

国科大操作系统研讨课任务书

RISC-V 版本



版本 1.0

目录

P6 文件系统	1
1 实验说明	1
2 物理文件系统	1
2.1 superblock	3
2.2 sector/block map	3
2.3 inode map	3
2.4 inode	3
2.5 data	4
2.6 任务一：物理文件系统的实现	5
3 文件操作	8
3.1 文件描述符	8
3.2 读写文件	9
3.3 任务二：文件操作	9
4 关于 S-core 和 A-core 任务的说明	12
5 C-core 任务的说明	13
5.1 任务三：多线程的从网络向文件传输数据	13

Project 6

文件系统

1 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。

本次的任务书将 S-core 和 A-core 的任务需求总结成了单独一章进行介绍,请大家看完任务书之后理解任务要求并完成各个 core 需求的任务。另外, C-core 的任务是将大家本课程实现的几乎所有操作系统功能进行综合测试的一个任务,请大家自己按任务书介绍实现测试程序,并完成测试。具体请看任务书最后一章的介绍。

请准备做 C-core 的同学注意,如果你完成了 C-core,你可以不做前面两个任务的检查,只进行任务三的检查。这是因为 C-core 的任务中已经包含了一个二级的文件系统的需求。但是在任务三之外,你依然需要完成大文件的任务,请大家参见关于 S-core 和 A-core 的任务说明中对大文件任务的描述。

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。本次实验需要实现的功能如表P6-1所示。

2 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?图P6-1为我们提供供大家参考的物理文件系统结构图。

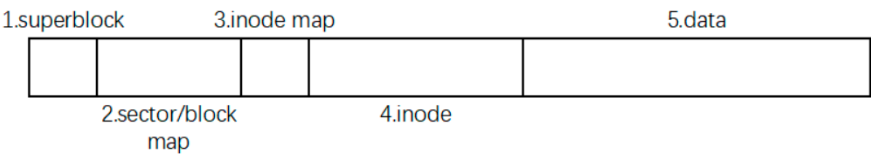


图 P6-1: 文件系统结构

目录操作		
函数名	shell 命令	说明
mkfs	mkfs	初始化文件系统
mkdir	mkdir	创建目录
rmdir	rmdir	删除目录
read_dir	ls	打印目录目录的内容
fs_info	statfs	打印文件系统信息，包括数据块的使用情况等
enter_fs	cd	进入目录

文件操作		
函数名	shell 命令	说明
mknod	touch	建立一个文件
cat	cat	将文件的内容打印到屏幕
open		打开一个文件
read		读一个文件
write		写一个文件
close		关闭一个文件

表 P6-1: 实验需要支持的指令或函数

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，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 不要放在一起，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 的结构大体如图2.4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：

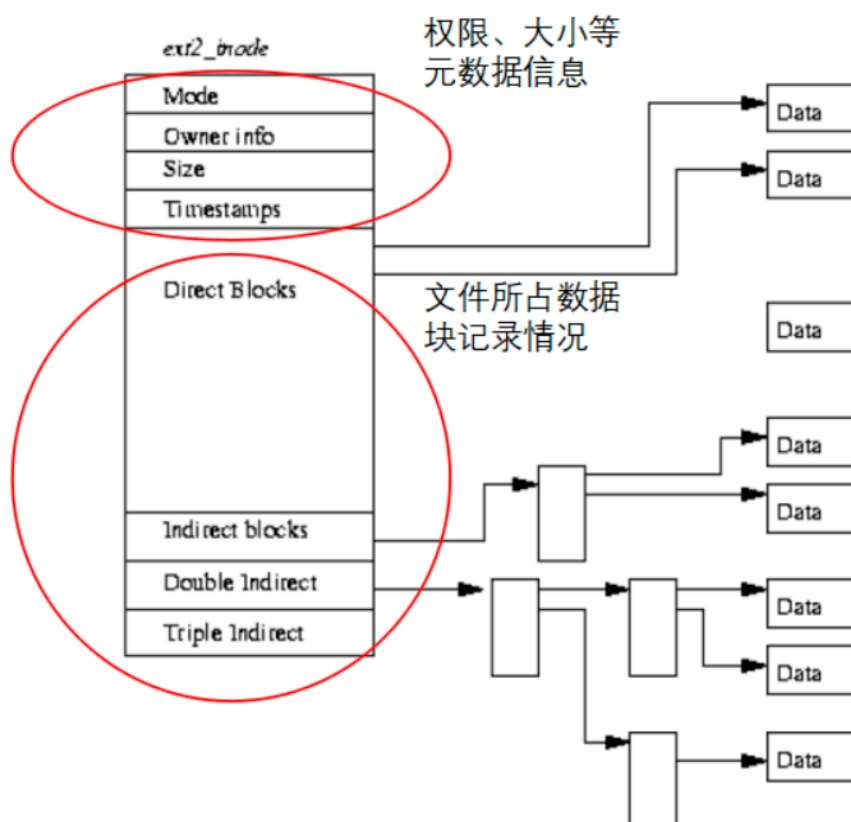


图 P6-2: ext2 文件系统的 inode 结构

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

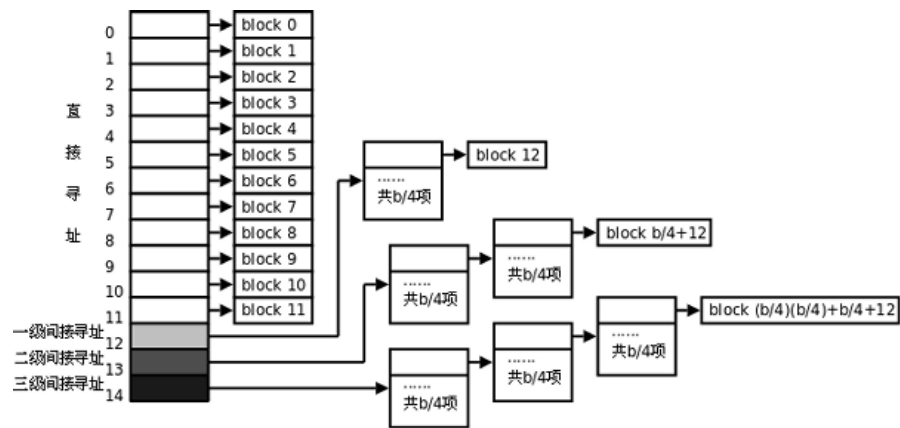


图 P6-3: 间接寻址

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

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，要求其实现表P6-2所列功能，对于所述的功能，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

表 P6-2: 任务一所需实现的命令

实验步骤

- mkfs 如下图所示，要求初始化文件系统时打印初始化的信息。可以参考图P6-4，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：
- 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

```

----- 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!

```

图 P6-4: mkfs

息。可以参考图P6-5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：

```

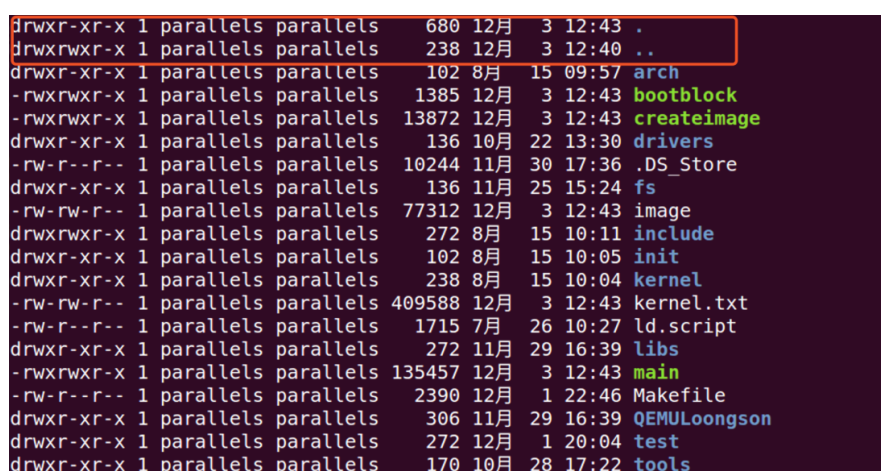
> root@UCAS_OS: statfs
magic : 0x66666666 (KFS)
used sector : 780/1048576, start sector : 1048576 (0x20000000)
inode map offset : 1, occupied sector : 1, used : 4/4096
sector map offset : 2, occupied sector : 256
inode offset : 258, occupied sector : 512
data offset : 770, occupied sector : 1047807
inode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

```

图 P6-5: statfs

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。
- 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。



```

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

```

图 P6-6: mkdir

- 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）。
- 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。
 - 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。

注意事项

- 本次实验需要使用对 SD 卡读、写的函数，对于 RISC-V 开发板，请大家直接使用 SBI 接口来读写 SD 卡 [1]。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 `sbi_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）。
- 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，具体逻辑如图 P6-7 所示。
- 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。

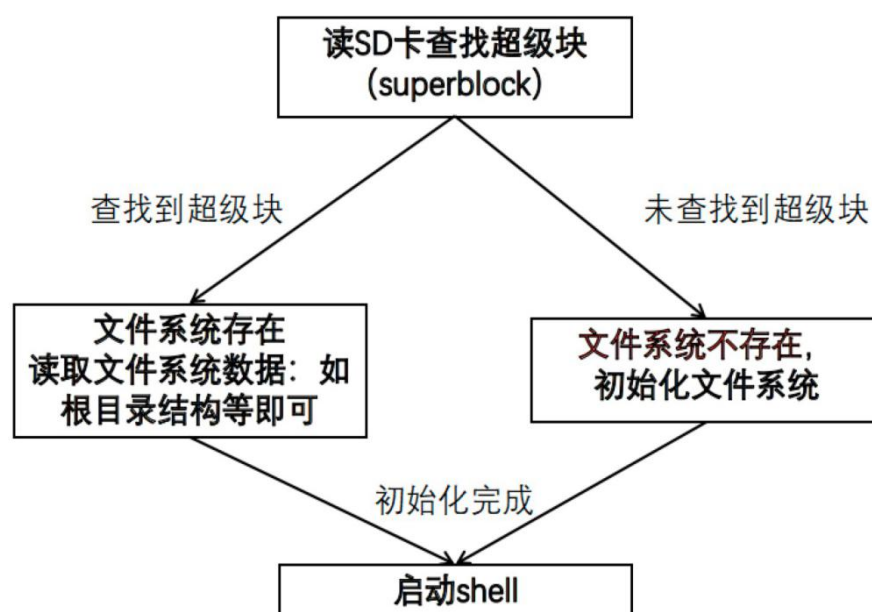


图 P6-7: 内核启动时的文件系统初始化流程

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`

3 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：



图 P6-8: 文件描述符

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

实验要求

实现文件系统文件操作，具体要求如表P6-3所示。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：

实验步骤

1. 使用 touch 指令建立一个目录，如图P6-9所示。

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

图 P6-9: touch

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P6-10图。
3. 重启开发板，使用 cat 指令打印出 1.txt 中的内容，如P6-11图。

ID	功能	对应 shell 命令	说明
1	建立一个空文件	<code>touch [filename]</code>	建立一个文件
2	打印文件内容	<code>cat [filename]</code>	打印一个文件到 shell
ID	功能	函数实现	说明
1	打开文件	<code>int open(char*name, int access)</code>	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	<code>int read(int fd, char *buff, int size)</code>	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	<code>int write(int fd, char *buff, int size)</code>	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	<code>void close(int fd)</code>	关闭一个文件，释放文件描述符。

表 P6-3: 文件操作

```
hello world!  
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:
```

图 P6-10: test_fs.c

```
----- 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!
```

图 P6-11: cat

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。

4 关于 S-core 和 A-core 任务的说明

以上我们介绍了本 Project 的任务。接下来我们将说明 S-core 和 A-core 任务的区别。

首先，S-core 的任务中可以只实现单级目录，即根目录下只有一级文件目录，文件的路径最多为/abc/123.txt 这种格式。A-core 则必须可以支持多级目录。

A-core 的同学除了参照上面的任务说明之外，还需要完成下面的额外任务：

(1) 额外功能的实现：

ID	功能	对应 Shell 命令	说明
1	硬链接	ln	建立一个文件的硬连接，具体实现参考 Linux 指令 ln
2	打印当前目录下文件的详细信息	ls -l	打印出文件的 inode 号、链接数、size（以字节为单位）
3	删除文件	rm	在本实验实现为，在父目录中移除文件名，即 dentry. 如果这个文件名是对应 inode 的最后一个硬链接，则 inode 被删除，它的资源也被释放。

表 P6-4: 要求的额外功能

(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 8MB 的文件，能进行文件创建和读写（在本实验中，我们要求直接指针的个数是同学们学号最后一位加上 5，一级间接指针、二级间接指针、三级间接指针的个数分别为 3, 2, 1 个）。

为了测试方便，需要完成 lseek 函数，用 lseek 定位大文件的读写指针，这样不需要写整个文件来测试大文件。lseek 函数定义见表P6-5。

功能	函数实现	说明
重定位文件指针	<code>int lseek(int fd,int offset,int whence)</code>	传入的参数为文件描述符 <code>fd</code> , <code>whence</code> 参数为下列一种: (1) <code>SEEK_SET(0)</code> , 参数 <code>offset</code> 即为新的读写指针所在的位置 (2) <code>SEEK_CUR(1)</code> , 以目前的读写指针所在的位置往后增加 <code>offset</code> 个偏移量 (3) <code>SEEK_END(2)</code> , 文件尾后再增加 <code>offset</code> 个偏移量作为新的读写指针所在的位置

表 P6-5: lseek 函数说明

5 C-core 任务的说明

由于 Project 6 是本课程的最后一个 Project, 因此我们为 P6 的 C-core 安排了综合目前实现的几乎所有操作系统功能的任务。考验大家实现的操作系统稳定性的同时, 也将展示同学们实现的操作系统丰富功能。

5.1 任务三：多线程的从网络向文件传输数据

本实验是本课程的最后一个实验任务, 需要 C-core 的同学完成。作为一个总结性的任务, 本任务需要用到课程中实现的几乎所有功能, 包括以下: OS 功能: 文件系统, 网络传输, 虚存管理, shell, 共享内存, mailbox, fork, 多线程, 双系统。本任务需要完成的具体功能流程如下:

- 1、编写测试程序并编译, 使用这次新发布的 `pkt` 程序的发送文件功能将它从本地发送至板卡上。

- 2、板卡上需要启动一个文件接收的程序, 从网络接收到这个文件, 并将其写入文件系统。

- 3、从文件系统中使用 **exec 文件名的格式**启动这个测试进程, 该测试程序的功能如下面所述:

- 4、申请一块大小为 8KB 的共享内存 `buffer`, 并新建一个文件。

- 5、fork 一个进程, 子进程用与父进程相同的 ID 共享这块 8KB 的内存 `buffer`, 并打开父进程创建的文件。(这里需要让子进程 fork 出来之后也能读到这个共享内存的 ID, 而非约定好一个相同的数字。)

- 6、父子两个进程各创建两个新线程。父进程的三个线程 1 个通过网卡中断方式读网络数据包、1 个把网络接收到的数据拷贝到共享内存中、1 个通过 mailbox 发送接收到的数据; 子进程 1 个线程读共享内存内容并将内容拷贝到写文件 `buffer`、1 个线程接

收 mailbox 发来的数据并将内容拷贝到写文件 buffer、1 个将写文件 buffer 的内容写文件落盘。注意多进程或者多线程读写同一块内存区域的时候需要加锁。网络包通过共享内存或 mailbox 传递数据取决于网络包中的端口号。

7、运行 pkt 程序，发送和 P5 一样的数据包。

8、在传输过程中使用 taskset 命令任意地将 6 个线程绑核。

9、重启板卡，通过双系统功能，用另一个操作系统中的 shell 命令行命令 cat 打开文件内容，判定网络数据和文件数据相等。为了让 cat 命令可以辨认文件内容，在第 6 步写文件的时候需要写成十六进制格式或者字符格式。

新发布的 pkt 程序的发送文件功能具体使用方法请见 pkt 文件夹中的使用方法.docx。

可以看出，P6 C-core 的功能大部分已经在之前的 Project 中完成，如果前面的 Project 已经完美的实现了各个功能的话，本任务只是将它们结合起来使用而已。如果有同学们觉得 P6 C-core 的功能难以完整实现，或者由于之前没有完成某个 Project 的 C-core 而导致功能缺失，我们鼓励大家只实现 C-core 任务的其中一部分功能，我们也会酌情给分。例如：1) 只通过 pkt 传输文件传输进来一个跑飞机的程序并从文件系统启动，而非复杂的测试程序；2) 两个进程并非使用 fork 创建，而是直接从 shell 启动了两个程序。3) 共享内存传输数据和 mailbox 发送数据只实现了其中之一。

参考文献

- [1] UG585, “Zynq-7000 soc.” https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, 2018. [Online; accessed 20-September-2021].