

OS Lab3 - File System

- OS Lab3 - File System
 - NachOS 文件系统简介
 - 实验内容与要求
 - Part1 多级索引
 - 主要任务
 - 重点与难点
 - 测试方法
 - Part2 多级目录与数据恢复
 - 主要任务
 - 重点与难点
 - 测试方法
 - 详细培训-那么我们需要干嘛呢??
 - 多级索引
 - 你需要做的-补洞!
 - 在补洞之前
 - 补洞

NachOS 文件系统简介

在 NachOS 里，有两份文件系统的实现：

- 利用宿主机的 File System 接口实现 NachOS 文件操作，直接读写宿主操作系统上的文件
- 维护一个宿主机上的 `DISK` 文件作为 NachOS 的模拟磁盘，在其上进行基于 Sector 的操作

在lab2代码的基础上，将 `code/build.linux/Makefile` line 194 中的 `-DFILESYS_STUB` 去掉，并将 `code/filesys/filesys.h` 中的 Line 67 ~ 94 的 `Write()` 与 `Read()` 两个函数定义拷至该文件后半部分的 `class FileSystem` 定义中，重新编译即可使用内置的虚拟磁盘：

```
$ cd code/build.linux
$ make clean
$ make -j
$ cd ../test
$ ../build.linux/nachos -f
$ ../build.linux/nachos -cp fork fork_in_virtual_disk
$ ../build.linux/nachos -x fork_in_virtual_dist
```

其中，`-f` 参数将格式化 NachOS 虚拟磁盘，`-cp` 将文件从 UNIX 文件系统拷贝到 NachOS 文件系统中，`-x` 参数使 NachOS 在虚拟磁盘中寻找对应名字的可执行文件并执行。更多参数可以参照 `code/threads/main.cc` 中的注释

实验内容与要求

NachOS 自带的文件系统限制诸多，例如：不支持变长文件；不支持目录；不支持多级索引。

本次实验分为两个阶段：

- Part 1：多级索引
- Part 2：多级目录与数据恢复

在本次实验中，你需要对助教提供的 Lab3 部分代码进行完善，对 NachOS 文件系统进行功能扩充。

我们提供了两个阶段的完整代码，需要将已完成的阶段一的代码复制到新项目的相应位置（将完成的6个洞的代码复制到对应位置即可）。

Part1 多级索引

主要任务

你需要阅读lab3源码中的如下文件：

```
- <root>
  | NachOS-4.0
  | code
  | filesystem
>>   | filehdr.cc
>>   | filehrd.h
>>   | filesystem.cc
>>   | filesystem.h
>>   | openfile.cc
>>   | openfile.h
>>   | synchdisk.cc
    | 其他已有的文件
```

与原版NachOS进行比对，理解NachOS一级索引的工作流程。同时仔细阅读提供的实验代码的注释，思考多级索引与一级索引的不同，并根据提示完成注释中提示的空缺部分

重点与难点

- 当写文件的长度超过文件现有长度，如何对文件进行扩充？
- 多级索引文件偏移如何映射到实际的sector号？

测试方法

```
$ cd code/test
$ ./toTestFileSys.sh > testOutcome.txt; cat testOutcome.txt
$ # toTestFileSys脚本内容如下：
$ nl toTestFileSys.sh
  1  #!/bin/bash
  2  nachos='../build.linux/nachos'
  3  $nachos -f
  4  $nachos -cp h.txt h.txt
  5  $nachos -cp testFileSys testFileSys
  6  $nachos -x testFileSys
```

正确的测试结果见主页的 `testOutcome.txt`

Part2 多级目录与数据恢复

主要任务

阶段二相较于阶段一，主要是下面的若干文件发生了改动：

```
- <root>
  | NachOS-4.0
    | code
      | filesystems
>>      | directory.cc
>>      | directory.h
>>      | filesystems.cc
```

理解NachOS创建根目录的工作流程，同时仔细阅读提供的实验代码的注释，思考如何创建一个新目录，以及如何恢复已删除数据，并根据提示完成注释中提示的空缺部分。

- 多级目录：

NachOS采用特殊的文件来表示一个目录，该特殊文件采用固定大小，文件用于存放固定个数的目录项（`DirectoryEntry`），NachOS中创建一个目录就是创建一个相应的特殊文件。

- 数据恢复：

NachOS通过 `FileSystem::Remove` 函数来删除一个文件，该函数的作用是将已分配给该文件的扇区释放，然后在当前目录中将该文件的相应目录项的 `inUse` 字段置为 `FALSE`。可以看到，删除文件后仍残留着一些信息可以供我们做数据恢复。为了避免恢复时覆盖丢失数据，我们将恢复出的数据写入到原生Linux系统中的一个文件中。

重点与难点

- 如何在一个目录中创建一个新的目录？
- NachOS删除一个文件时做了哪些操作？有哪些残留信息可以帮助我们恢复文件？

测试方法

```
$ cd code/test
$ ./toTestDirectoryAndRecovery.sh;
$ cat recovery.txt
```

toTestDirectoryAndRecovery.sh 脚本通过创建文件 folder1/folder2/file，对其进行读写来验证多级目录相关代码的正确性，随后通过删除该文件，再恢复到 recovery.txt 中来验证 recovery功能是否正常。若 recovery.txt 中包含10行字符串 we write contents to folder1/folder2/file 则说明recovery功能是正常的。

详细培训-那么我们需要干嘛呢??

多级索引

索引是一种将文件内容组织在磁盘上的方式。

当用户想要访问一个指定文件的指定偏移时，需要查找索引表，然后根据索引表的提供的sector号去访问文件内容。

- 一级索引：

在文件Header中有一个sector号的数组。

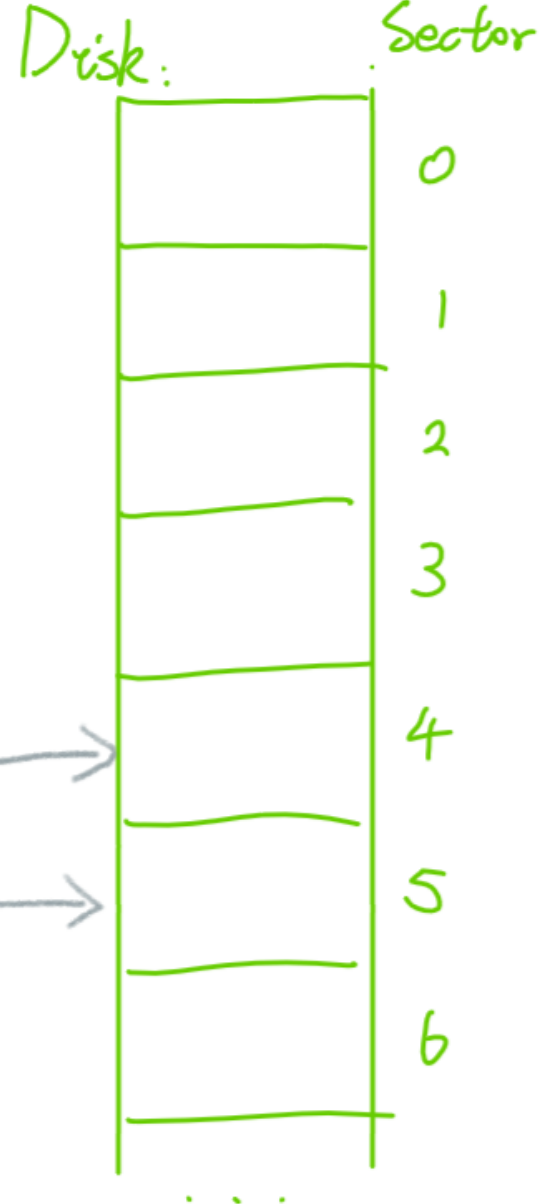
由于每个sector大小固定，访问指定偏移时，只需要将偏移除以sector大小即可知道sector号在数组第几个，根据该sector号去这个sector上获取文件内容。该sector称为 dataSector

Each Sector = 128 B



Offset 379 = $2 \times 128 + 123$

So: Sector 5's 123B



这样带来的好处是文件不用连续存放，可以分散在磁盘各个位置，提高利用率，并且访问每次访问指定偏移所需的读取磁盘的次数恒定为2(访问header算一次)。

坏处则十分显然：文件最大大小受到header大小的限制

- 二级索引：

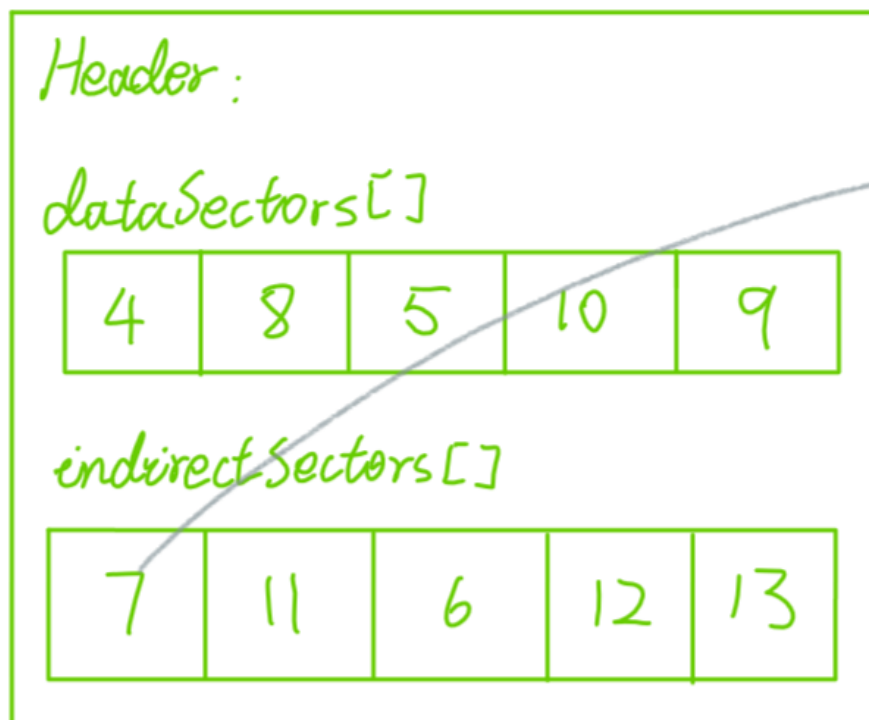
相比一级索引，二级索引就是将sector号数组放在磁盘上，而不是放在header中。这个存放sector号数组的sector称为 `indirectSector`。

访问指定偏移时，首先看该偏移是否在一级索引可以寻找到的位置，若不在，则计算sector号其应该在哪个 `indirectSector`，在其中获取sector号数组，接下来类似一级索引

Each Sector = 128 B

Each indirectSector has 32 sector number

$$32 \times 128 = 4096 \text{ B}$$



Offset: $900 = 5 \times 128 + 2 \times 128 + 4$

Sector number in sector 7.

你需要做的-补洞!

首先,如果你用grep(global search regular expression(RE) and print out the line)命令匹配"洞",就会得到:

```
# zevin @ ubuntu in ~/Desktop/lab3pt1/NachOS-4.0/code [10:13:18]
$ grep -r 洞 ./
./filesystem/filehdr.cc:           // 洞1:begin
./filesystem/filehdr.cc:           //   这个洞要注意边界条件(如doneSec等)
./filesystem/filehdr.cc:           // 洞1:end
./filesystem/filehdr.cc:           // 洞2:begin
./filesystem/filehdr.cc:           // 洞2:end
./filesystem/filehdr.cc:           // 洞3:begin
./filesystem/filehdr.cc:           //   这个洞需要的控制/判断逻辑很少,主要是计算,
./filesystem/filehdr.cc:           // 洞3:end
./filesystem/filehdr.cc:           // 洞4:begin
./filesystem/filehdr.cc:           //   这个洞要注意边界条件(如doneSec等)
./filesystem/filehdr.cc:           // 洞4:end
./filesystem/openfile.cc:          // 洞5:begin
./filesystem/openfile.cc:          // 洞5:end
./filesystem/filesys.cc:          // 洞6:begin
./filesystem/filesys.cc:          // 洞6:end
```

所以,这次实验一共有11个(阶段一6个+阶段二5个)"洞"需要勇敢的你来填补!
阶段一的6个洞按照顺序分别在这六个成员函数中.

```
FileHeader::Allocate()  
FileHeader::Deallocate()  
FileHeader::ByteToSector()  
FileHeader::expandFile()  
OpenFile::WriteAt()  
FileSystem::Open()
```

阶段二的5个洞按照顺序分别在这五个成员函数中(其中有三个洞的需要填的内容基本一样).

```
FileSystem::Create()  
FileSystem::CreateFolder()  
FileSystem::Open()  
FileSystem::Remove()  
FileSystem::Recover()
```

等等!

在补洞之前

当然了,在开始补洞之前,你还需要理解这个洞,也就要阅读相关源码.

这次你需要关注的lab3相关源码是哪些呢?也许聪明的你已经想到了,就是这些:

如果你用grep递归匹配 `lab\s?3` (这是正则表达式,以后如果掌握了可能大大提高字符串匹配效率),就会得到如下结果:


```
# zevin @ ubuntu in ~/Desktop/lab3pt1/NachOS-4.0/code [10:18:34]
$ grep -r -E "LAB\s?3" ./
./filesystems/synchdisk.cc:/* ++++++ LAB 3 请阅读如下两个成员函数 ++++++ */
./filesystems/filehdr.cc:/* ++++++ LAB 3 ++++++ */
./filesystems/filehdr.cc:/* ++++++ LAB3 ++++++ */
./filesystems/filehdr.cc:/* ++++++ LAB3 ++++++ */
./filesystems/filehdr.cc:/* ++++++ LAB 3 ++++++ */
./filesystems/filehdr.cc:/* ++++++ LAB3 ++++++ */
./filesystems/openfile.cc:/*+++++ LAB3 ++++++*/
./filesystems/filesys.h:/* ++++++ LAB3 ++++++ */
./filesystems/filesys.h://LAB3
./filesystems/openfile.h://LAB3
./filesystems/filesys.cc:/* ++++++ LAB3 ++++++ */
./filesystems/filesys.cc:/* ++++++ LAB3 可能需要阅读这里 ++++++ */
./filesystems/filehdr.h:/* LAB3 */
./filesystems/filehdr.h:/* +++LAB3+++ */
./filesystems/filehdr.h:/* ++++++ LAB3 ++++++ */
./userprog/ksyscall.h:/* ++++++ LAB3 ++++++ */
./userprog/syscall.h://LAB3
```

你可以参照前几次实验同样位置的源码，理解清楚本次多级索引在此基础上增加了哪些工作量

补洞

如果你看得差不多了,勇敢的你就可以开始补洞了!

我们的建议是按照洞的序号进行填补.

一开始在填补洞1和洞2时,你可能会有一点点吃力.这时不要怀疑自己,静下心来看看洞1和洞2所在的成员函数中已经实现了的部分.

理解了这些内容,再结合上一小节阅读的源码,相信聪明的你马上就能敲击键盘,完成这几个洞了.

这次的注释像上次实验一样给的很详尽,流程和需要考虑的问题也很清晰.只有个别地方可能需要你小小的动动脑筋.

下面开始给出进一步的提示:

- 洞1: `FileHeader::Allocate`

首先,你需要对 `indirectSectors` 进行处理, 方法类似于:

```
dataSectors[i] = freeMap->FindAndSet();
```

然后, 对 `sectors[NumInDirectIndex]` 数组元素一次进行类似操作, 注意继续统计 `doneSec`, 直到这次的 `sectors[]` 用完, 或者 `doneSec` 达到 `numSectors` 的大小。

最后, 你需要将 `sectors[]`, 用 `SynchDisk` 中的某接口写入到 `indirectSectors[j]` 中。

- 洞2: `FileHeader::Deallocate`

和洞1很相似,这里为了 `deallocate`, 首先要读 `indirectSectors[j]` (用什么接口呢?), 然后进行 `allocate` 的逆操作, 然后将对应的使用了的 `sector` 进行 `clear` 操作, 最后别忘了 `indirectSectors[j]` 也要 `clear`

- 洞3: `FileHeader::ByteToSector`

返回 `offset` 这个字节所在的sector号。

分为在`direct sector`和在`indirect sector`两种情况。这里需要理解多级索引的地址计算，加油！

- 洞4: `FileHeader::expandFile`

在write的文件大小超过原本大小的时候变长。找到 `indirectSectors[j]!=-1` 时的sector，或者 `indirectSectors[j]==-1` 的第一个sector，并进行修改，修改的操作和 `allocate` 洞1十分相似，别忘了将修改的 `secetors[]` 写回到 `indirectSectors[j]` 的块中。

- 洞5: `OpenFile::WriteAt`

先计算需要的sector总数目，然后调用 `FileHeader` 类里的 `expandFile()` 函数。之后修改hdr，也就是写回(`FileHeader::WriteBack`)，最后写回freeMap

- 洞6: `FileSystem::Open`

这里处理open。

根据得到的sector，添加到 `openedFile` 里面。

注意不要修改 `openedFile[0,1,2]`，从 `openedFile[3]` 开始分配

- 洞7: `FileSystem::Create(char *name, int initialSize)`

将name包含的目录名解析出来，然后从根目录开始跳转定位到当前目录。

若name中包含的目录不存在，则认为失败并返回FALSE。

例如，name为“folder1/folder2/file1”，则需要从根目录出发，通过其下的folder1到达folder2，然后在folder2目录下创建文件file1，若folder1或folder2不存在，则失败返回FALSE。

- 洞8: `FileSystem::CreateFolder(char *name)`

将name包含的目录名解析出来，然后从根目录开始跳转定位到当前目录，并在当前目录下创建相应的新目录，具体过程可参考NachOS的根目录创建过程。

例如，name为“folder1/folder2/folder3”，则需要从根目录出发，通过其下的folder1到达folder2，然后在folder2目录下创建名为folder3的文件（目录就是一个文件），若folder1或folder2不存在，则失败返回FALSE。

注：实现该函数可以参考 `FileSystem::Create` 函数。

- 洞9: `FileSystem::Open(char *name)`

将name包含的目录名解析出来，然后从根目录开始跳转定位到当前目录。

若name中包含的目录不存在，则认为失败并返回FALSE。

- 洞10: `FileSystem::Remove(char *name)`

将name包含的目录名解析出来，然后从根目录开始跳转定位到当前目录。

若name中包含的目录不存在，则认为失败并返回FALSE。

- 洞11: `FileSystem::Recover(char *srcName, char *dstName)`

将srcName包含的目录名解析出来，然后从根目录开始跳转定位到当前目录，并在当前目录下判断是否能够恢复（即目录表项是否还有条目的名字为待恢复文件的名称），若能的话，恢复相应的文件并存放到原生Linux系统下的dstName文件中。

注：实现该函数可以参考 `FileSystem::Create` 函数。