Lab6: file system

本实验是这学期的最后一个实验，在本实验中，您将向xv6文件系统添加大型文件和符号链接。

> Attention
>
> 在编写代码之前，您应该阅读《xv6手册》中的《第八章：文件系统》，并学习相应的代码。

获取实验室的xv6源代码并切换到fs分支：

```
$ git checkout fs
$ make clean
```

Large files(moderate)

在本作业中，您将增加xv6文件的最大大小。目前，xv6文件限制为268个块或`268*BSIZE`字节（在xv6中`BSIZE`为1024）。此限制来自以下事实：一个xv6 inode包含12个“直接”块号和一个“间接”块号，“一级间接”块指一个最多可容纳256个块号的块，总共12+256=268个块。

`bigfile`命令可以创建最长的文件，并报告其大小：

```
$ bigfile
..
wrote 268 blocks
bigfile: file is too small
$
```

测试失败，因为`bigfile`希望能够创建一个包含65803个块的文件，但未修改的xv6将文件限制为268个块。

您将更改xv6文件系统代码，以支持每个inode中可包含256个一级间接块地址的“二级间接”块，每个一级间接块最多可以包含256个数据块地址。结果将是一个文件将能够包含多达65803个块，或256*256+256+11个块（11而不是12，因为我们将为二级间接块牺牲一个直接块号）。

预备

`mkfs`程序创建xv6文件系统磁盘映像，并确定文件系统的总块数；此大小由*kernel/param.h*中的`FSSIZE`控制。您将看到，该实验室存储库中的`FSSIZE`设置为200000个块。您应该在`make`输出中看到来自`mkfs/mkfs`的以下输出：

```
nmeta 70 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 25) blocks 199930 total 200000
```

这一行描述了`mkfs/mkfs`构建的文件系统：它有70个元数据块（用于描述文件系统的块）和199930个数据块，总计200000个块。

如果在实验期间的任何时候，您发现自己必须从头开始重建文件系统，您可以运行`make clean`，强制`make`重建*fs.img*。

看什么

磁盘索引节点的格式由*fs.h*中的`struct dinode`定义。您应当尤其对`NDIRECT`、`NINDIRECT`、`MAXFILE`和`struct dinode`的`addrs[]`元素感兴趣。查看《XV6手册》中的图8.3，了解标准xv6索引结点的示意图。

在磁盘上查找文件数据的代码位于*fs.c*的`bmap()`中。看看它，确保你明白它在做什么。在读取和写入文件时都会调用`bmap()`。写入时，`bmap()`会根据需要分配新块以保存文件内容，如果需要，还会分配间接块以保存块地址。

`bmap()`处理两种类型的块编号。`bn`参数是一个“逻辑块号”——文件中相对于文件开头的块号。`ip->addrs[]`中的块号和`bread()`的参数都是磁盘块号。您可以将`bmap()`视为将文件的逻辑块号映射到磁盘块号。

你的工作

修改 `bmap()`，以便除了直接块和一级间接块之外，它还实现二级间接块。你只需要有11个直接块，而不是12个，为你的新的二级间接块腾出空间；不允许更改磁盘inode的大小。 `ip->addrs[]` 的前11个元素应该是直接块；第12个应该是一个一级间接块（与当前的一样）；13号应该是你的新二级间接块。当 `bigfile` 写入65803个块并成功运行 `usertests` 时，此练习完成：

```
$ bigfile
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
........
wrote 65803 blocks
done; ok
$ usertests
...
ALL TESTS PASSED
$
```

运行 `bigfile` 至少需要一分钟半的时间。

提示：

- 确保您理解 `bmap()` 。写出 `ip->addrs[]` 、间接块、二级间接块和它所指向的一级间接块以及数据块之间的关系图。确保您理解为什么添加二级间接块会将最大文件大小增加256*256个块（实际上要-1，因为您必须将直接块的数量减少一个）。
- 考虑如何使用逻辑块号索引二级间接块及其指向的间接块。
- 如果更改 `NDIRECT` 的定义，则可能必须更改*file.h*文件中 `struct inode` 中 `addrs[]` 的声明。确保 `struct inode` 和 `struct dinode` 在其 `addrs[]` 数组中具有相同数量的元素。
- 如果更改 `NDIRECT` 的定义，请确保创建一个新的*fs.img*，因为 `mkfs` 使用 `NDIRECT` 构建文件系统。

- 如果您的文件系统进入坏状态，可能是由于崩溃，请删除*fs.img*（从Unix而不是xv6执行此操作）。 `make` 将为您构建一个新的干净文件系统映像。
- 别忘了把你 `bread()` 的每一个块都 `brelse()` 。
- 您应该仅根据需要分配间接块和二级间接块，就像原始的 `bmap()` 。
- 确保 `itrunc` 释放文件的所有块，包括二级间接块。

Symbolic links(moderate)

在本练习中，您将向xv6添加符号链接。符号链接（或软链接）是指按路径名链接的文件；当一个符号链接打开时，内核跟随该链接指向引用的文件。符号链接类似于硬链接，但硬链接仅限于指向同一磁盘上的文件，而符号链接可以跨磁盘设备。尽管xv6不支持多个设备，但实现此系统调用是了解路径名查找工作原理的一个很好的练习。

你的工作

YOUR JOB

您将实现 `symlink(char *target, char *path)` 系统调用，该调用在引用由 `target` 命名的文件的路径处创建一个新的符号链接。有关更多信息，请参阅 `symlink` 手册页（注：执行 `man symlink` ）。要进行测试，请将 `symlinktest` 添加到 ***Makefile***并运行它。当测试产生以下输出（包括 `usertests` 运行成功）时，您就完成本作业了。

```
$ symlinktest
Start: test symlinks
test symlinks: ok
Start: test concurrent symlinks
test concurrent symlinks: ok
$ usertests
...
ALL TESTS PASSED
$
```

提示：

- 首先，为`symlink`创建一个新的系统调用号，在*user/usys.pl*、*user/user.h*中添加一个条目，并在*kernel/sysfile.c*中实现一个空的`sys_symlink`。
- 向*kernel/stat.h*添加新的文件类型（`T_SYMLINK`）以表示符号链接。
- 在*kernel/fcntl.h*中添加一个新标志（`O_NOFOLLOW`），该标志可用于`open`系统调用。请注意，传递给`open`的标志使用按位或运算符组合，因此新标志不应与任何现有标志重叠。一旦将*user/symlinktest.c*添加到*Makefile*中，您就可以编译它。
- 实现`symlink(target, path)`系统调用，以在`path`处创建一个新的指向`target`的符号链接。请注意，系统调用的成功不需要`target`已经存在。您需要选择存储符号链接目标路径的位置，例如在inode的数据块中。`symlink`应返回一个表示成功（0）或失败（-1）的整数，类似于`link`和`unlink`。
- 修改`open`系统调用以处理路径指向符号链接的情况。如果文件不存在，则打开必须失败。当进程向`open`传递`O_NOFOLLOW`标志时，`open`应打开符号链接（而不是跟随符号链接）。
- 如果链接文件也是符号链接，则必须递归地跟随它，直到到达非链接文件为止。如果链接形成循环，则必须返回错误代码。你可以通过以下方式估算存在循环：通过在链接深度达到某个阈值（例如10）时返回错误代码。
- 其他系统调用（如`link`和`unlink`）不得跟随符号链接；这些系统调用对符号链接本身进行操作。
- 您不必处理指向此实验的目录的符号链接。