

Lab5.文件系统

在进行本实验之前，你需要进行分支管理，请在工程目录下输入以下命令：

```
git fetch
git checkout lab5
make clean
```

💡 注意

在本次实验的代码中，`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，所以第一次`make qemu`之后启动xv6可能有点慢，请耐心等待，它并不是卡住了。

📖 扩展阅读

本实验开始前，如果对文件系统知识薄弱的同学，可以阅读[OSTEP](#)（有中文版）的相关章节，特别推荐阅读第39章。

概述

在之前的实验中，我们已经学习并接触到了两项关键操作系统技术的发展：进程，他是虚拟化的CPU；地址空间，他是虚拟化的内存。在这两种抽象的共同作用下，程序运行时好像它在自己的私有独立世界中一样，好像它有自己的处理器（或多处理器），好像它有自己的内存。

接下来，我们加上虚拟化中更关键的一步：持久存储（persistent storage）。我们知道让应用程序直接管理磁盘并不是个好主意——你可以想象程序之间需要协调磁盘的并发访问，并且如果有程序出了bug，其他程序的数据就可能被无故“摧毁”。因此，为了更好地帮助应用程序管理持久数据，一个很自然的想法就是对磁盘进行“虚拟化”，让应用程序访问“虚拟”的磁盘，实现应用与应用之间的隔离,文件系统就在此基础上诞生。

我们将通过解构xv6中的文件系统和阅读源码来对文件系统进行进一步的了解和实践。

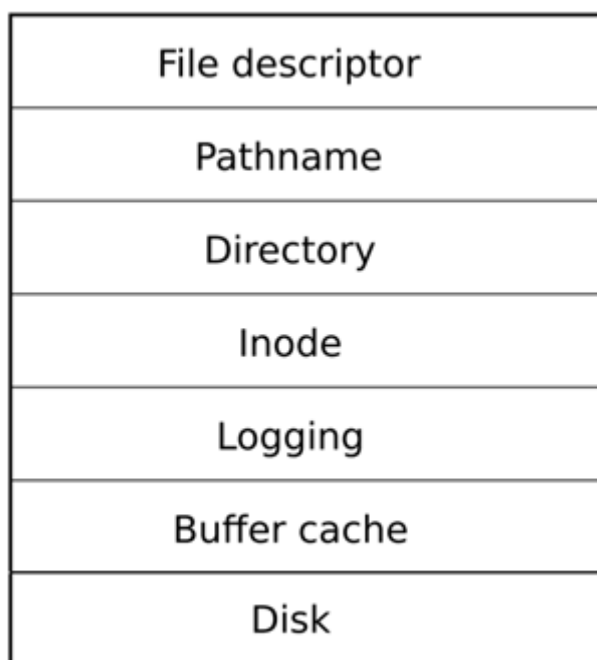
构建可靠的文件系统

首先在这里，我们思考一下：文件系统要实现在存储媒体（如硬盘、固态驱动器等）上组织文件和目录，并提供访问、读取、写入和删除这些文件和目录的方式，需要解决什么难题。

1. **数据结构和元数据管理：** 文件系统需要设计合适的数据结构来表示目录树、文件和空闲块信息。这包括了维护目录树的层次结构，跟踪每个文件使用的数据块，以及记录空闲磁盘区域。
2. **崩溃恢复：** 为了支持崩溃恢复，文件系统必须实现事务和日志记录机制。每个更新操作都应该被认为是一个事务，必须先记录在日志中，然后再在元数据上进行更新。如果发生崩溃，文件系统可以使用日志来回滚未完成的事务或者重新执行已提交的事务，以确保数据结构的一致性。
3. **并发控制：** 多个进程或线程可能同时操作文件系统，因此需要实现并发控制机制，如锁机制，以确保在同一时间只有一个进程能够修改文件系统的数据结构。这可以防止数据结构的不一致性和冲突。
4. **缓存管理：** 为了解决磁盘访问速度慢的问题，文件系统必须维护一个内存中的缓存 (cache) 来存储常被访问的数据块。缓存可以减少磁盘访问次数，提高文件系统的性能。然而，缓存管理也需要考虑缓存一致性和内存管理等问题。

为了解决这些问题，需要逐层实现一个完整的文件系统。接下来，我们将以xv6文件系统为基础，探讨如何有效地解决这些问题。

XV6的文件系统架构



如上图所示，在xv6中对文件系统的架构做出了如下的分层：

- 在最底层是磁盘，也就是一些实际保存数据的存储设备，正是这些设备提供了持久化存储。
- 在这之上是Buffer cache或者说Block cache，这些cache可以避免频繁的读写磁盘。这里我们将磁盘中的数据保存在了内存中。
- Logging层，是用于实现文件系统的日志和事务机制的重要组成部分。日志和事务机制是为了实现文件系统的崩溃恢复和一致性，保证在系统崩溃或其他异常情况下，文件系统数据的完整性和正确性。

- Inode层，inode可能指磁盘上的数据结构，包含文件大小和数据块编号列表；也可能指内存中的inode，包含磁盘上inode的副本和内核中需要的额外信息。
- Directory层，即为目录层，目录本身也是inode，只是它的数据包含了其目录下的文件。Directory层引入树状结构，并提供 `dirlookup` 函数（见 `kernel/fs.c`）方便搜寻。
- 再往上，就是文件名，和文件描述符操作。

不同的文件系统组织方式和每一层可能都略有不同，有的时候分层也没有那么严格，xv6中分层也不是很严格，但是从概念上来说这里的结构对于理解文件系统还是有帮助的。实际上所有的文件系统都有组件对应这里不同的分层，例如buffer cache，logging，inode和路径名。

接下来，我们会自底向上去分析。

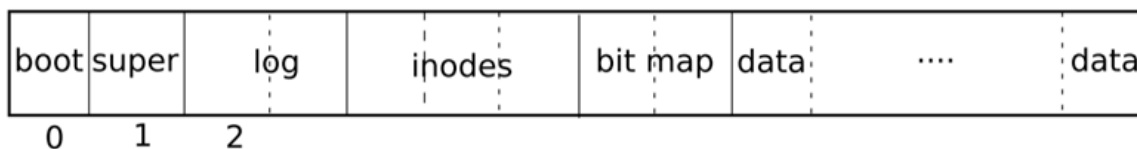
存储设备

实际中有非常多不同类型的存储设备，这些设备的区别在于性能，容量，数据保存的期限等。其中两种最常见，并且你们应该也挺熟悉的是SSD和HDD。

- sector通常是磁盘驱动可以读写的最小单元，它过去通常是512字节。
- block通常是操作系统或者文件系统视角的数据。它由文件系统定义，在xv6中它是1024字节。所以xv6中一个block对应两个sector。通常来说一个block对应了一个或者多个sector。

磁盘的区域划分

我们都知道刚刚拿到手的磁盘是不能直接使用的，必须经过物理格式化和逻辑格式化之后才能被使用。逻辑格式化就是在磁盘上写入一系列支持文件系统所需的数据，这些数据将磁盘的空间进行划分，为实现文件组织，存储空间的分配和回收提供相应的支持。xv6对磁盘空间划分方法如下图所示：



- block0要么没有用，要么被用作boot sector来启动操作系统。
- block1通常被称为super block，它描述了文件系统。它可能包含磁盘上有多少个block共同构成了文件系统这样的信息。我们之后会看到xv6在里面会存更多的信息，你可以通过block1构造出大部分的文件系统信息。我们可以在 `kernel/fs.h` 中看到 `super block` 的数据结构。
- 在xv6中，log从block2开始，到block32结束。实际上log的大小可能不同，这里在 `super block` 中会定义log就是30个block。

- 接下来在block32到block45之间，XV6存储了inode。多个inode会打包存在一个block中，一个inode是64字节。
- 之后是bitmap block，这是我们构建文件系统的默认方法，它只占据一个block。它记录了数据block是否空闲。
- 后面就全是数据block了，数据block存储了文件的内容和目录的内容。

一个小小的思考题

假设inode是64个字节，如果你想要读取inode11，那么你应该怎么推算出他所在的block呢？可以把你们的计算过程写在实验报告中。

Buffer cache层

磁盘存储以块（有时称为扇区）为单位，磁盘驱动通过读取多个块来执行操作，但磁盘I/O操作耗时，通常需要数毫秒。因此，Buffer cache层的核心思想是在内存中缓存部分块，隔离磁盘和上层应用，创造所有块都在内存中的错觉。此外，Buffer cache层还协调多进程对数据的并发访问。这种设计加快数据访问速度，避免频繁磁盘读写，减少操作延迟。同时，通过控制缓存数据的并发访问，确保数据一致性和正确性，提供更优系统性能。

在xv6中，Buffer cache层的工作原理涉及以下几个方面：

- 1. 缓存的数据结构：** 在xv6的源码中，Buffer cache层使用了 `struct buf`（见 `kernel/buf.h`）来表示缓存块。这个结构包含了块的状态（是否是脏块、是否已经读取等）、块的数据、块号等信息。
- 2. 缓存的初始化：** 在操作系统启动时，xv6会初始化一块内存区域作为缓存池，将其划分为多个缓存块。每个缓存块与一个磁盘块对应。
- 3. 缓存读取：** 当应用程序需要读取文件系统块时，xv6会首先检查缓存中是否已经存在对应的块。如果存在，就直接从缓存中读取；如果不存在，就从磁盘上读取数据，并将数据加载到缓存中。
- 4. 缓存写入：** 当应用程序修改了缓存中的块数据时，xv6会将这些块标记为“脏”，表示它们已经被修改。xv6会在适当的时机，比如在缓存块被替换时，将脏块的数据写回到磁盘。
- 5. 缓存替换策略：** xv6使用LRU（最近最少使用）策略来决定哪些缓存块应该被替换。当缓存满了且需要加载新的块时，xv6会选择最长时间未被访问的块进行替换。
- 6. 缓存同步：** 在一些需要同步的情况下，如系统关闭时，xv6会将所有脏块的数据写回到磁盘，以确保数据的一致性。

要深入了解xv6的缓存层工作原理，建议您查阅xv6操作系统的源代码 `kernel/bio.c`。通过仔细研究源代码，您将能够更好地理解xv6的缓存层是如何工作的。

💡 请认真阅读源码

这里不要偷懒啊，了解文件系统的最好方法就是回到源码中去研究，文件系统的代码确实有点多，但我们希望你还是可以认真读完各层架构中的代码，读完之后你会变得更强大。

Logging层

文件系统设计中，错误恢复是一个重要问题。由于文件系统需要多次磁盘写操作，崩溃可能导致磁盘数据不一致。xv6通过日志系统解决这个问题：系统调用将写操作封装成日志记录，写入日志后再写入磁盘。提交记录标志一次完整操作。崩溃前，日志未提交，不影响。崩溃后，恢复程序回放日志。这使得磁盘操作在崩溃前或后都是“原子”的，确保文件系统一致性。

在xv6源码中，logging层的关键部分可以在 `log.c` 文件中找到。这个文件包括了实现日志记录、事务提交、崩溃恢复等的代码。通过仔细研究 `kernel/log.c` 文件，您可以更深入地了解xv6的logging层是如何实现文件系统的可靠性和一致性的。

Inode

结合前面实验学到的知识，应用进程内，进程控制块PCB存有打开的文件数组 `ofile[]`，其中进程每调用一次 `open()` 就会返回一个整数，这个整数就是文件数组的下标索引，也称文件描述符（通常用 `fd` 表示）。每一个下标对应一个 `ofile[]` 中打开的文件。文件在进程中以线性编辑的视图进行写入，而实际过程中写入的文件数据会被分成等大（一般固定为某一个 `size`）的逻辑数据块，逻辑数据块通过文件中的映射表，转换为实际存储的物理盘块号。尽管物理盘块号是非线性的（比如有扇区等划分），但总体经过封装仍能以线性方式进行对物理盘块的位置映射。

在上述过程中，对多个文件，在逻辑数据块到物理盘块之间的转换仍有细节。操作系统的文件管理（文件系统）会保存一个静态数组，`ftable[file]`，来存储目前打开的每一个文件。即使是同一个文件，不同的进程访问，读写权限也不一样，游标位置可能也不一样，所以也可能有多个 `ftable[]` 项其实指代同一个文件。其中，`ftable[file]` 中的 `file`，既可以是管道类型的，也可以是其他没有 `inode` 的文件类型；有 `inode` 对应的文件，既可以是普通的文件，也可以是设备文件，也可以是专门存储了很多文件信息的文件（也即目录，其实是一种存储了很多文件的信息的特殊文件）。这就是所谓的 `unix` 的万物皆文件的哲学，无论是什么类型的数据读写、数据交换，都可以封装成文件的样式纳入这个体系中进行管理。

每个文件内部的数据结构都存有一个 `addrs[]`，用于存放本文件第 `i` 块数据块存在第 `addrs[i]` 块逻辑块的信息。

此时，文件就完成了操作系统内核的管理（`inode`）到应用程序的封装读写（`off`）之间的关联。

每一个文件的存储映射并不直接存在本文件结构的结构体中，而是再封装进了一个叫 inode 的结构体中（见 kernel/file.h），因此可见每个文件会映射一个 inode，映射数组 `addrs[]` 就在这个 inode 中。inode 存的就是这个文件中的数据在物理空间中的映射，所以无论是哪个进程访问、这个文件被以何种多种的形式打开访问（`int ref`:表示目前本文件已被引用次数），都是映射到同一个 inode，因为尽管以不同打开方式（读写游标位置不同(`uint off`)，读写权限不同（`char writeable/readable`），在 `file` 结构体中有定义），它们的数据块和内存块之间的虚拟物理存储映射位置关系是不变的（因为就是同一个文件），所以 `ftable[file]` 中要是打开了很多个文件，其中也可以有一些文件是指向同一个 inode 的，表现了一个静态文件多次动态打开的过程，而每一个动态打开的文件又通过 inode 关联到同一个静态的磁盘文件上。

所以，xv6 对文件的管理是通过在磁盘中保存一大堆的 inode，来记录各个文件的转换映射关系，然后在内存需要的时候将这些映射调出来用（在内存中有个 `icache` 结构体，就是用于缓存 inode 节点的）。而在磁盘中，这些 inode 用 `dinode` 来存储，inode 实际上是 `dinode` 提取到内存中使用后的形式，在硬盘中 `dinode` 也就是文件系统真正管理文件的信息结构（主要就是数据块的位置查找）。内存版本的 `dinode`（也就是 inode）比 `dinode` 要多一些信息：

```
struct inode {
    uint dev;           // Device number
    uint inum;          // Inode number
    int ref;            // Reference count
    struct sleeplock lock; // protects everything below here
    int valid;          // inode has been read from disk?

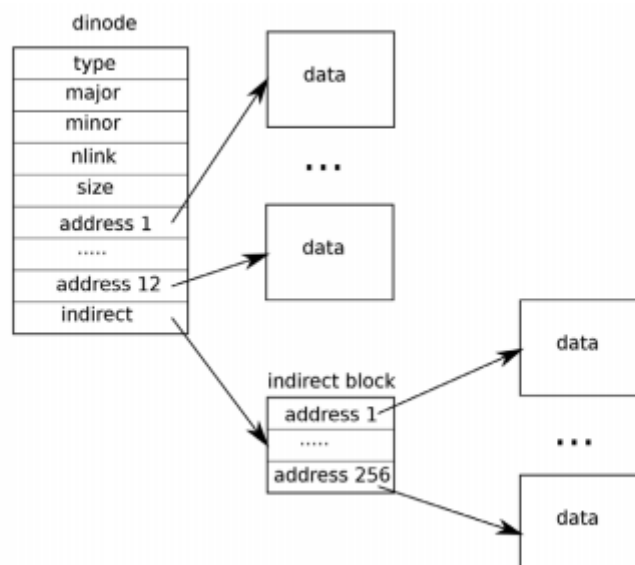
    short type;         // copy of disk inode
    short major;
    short minor;
    short nlink;
    uint size;
    uint addrs[NDIRECT+1];
};
```

```
// On-disk inode structure
struct dinode {
    short type;           // File type
    short major;          // Major device number (T_DEVICE only)
    short minor;          // Minor device number (T_DEVICE only)
    short nlink;          // Number of links to inode in file
    system
    uint size;            // Size of file (bytes)
    uint addrs[NDIRECT+1]; // Data block addresses
};
```

即多了 `uint dev`、`uint inum` (inode 节点号)、`int ref` (本文件被引用的次数)、`int flags` (标志位, 标志空闲)。

讲了这么多, 我们可以知道inode在文件系统架构中的重要性, 本次实验内容的关注点也会针对inode层来推出。而阅读完上述文字后你可能还不是那么清晰, 那就需要仔细阅读代码, 搞清楚代码间的逻辑关系, 文件系统比较复杂, 必须回到源码中进行探索。

请先看下图, 其为 xv6 文件系统的一个“文件”在存储设备 (一般为外存) 上的逻辑结构图。



此图分为三个子部分:

1. 左边的 `dinode`
2. 中间的数据块 `data` 和一级间接块 `indirect block`
3. 右边的、通过一级间接块索引的数据块 `data`

左上角的 `dinode` 表示的是与“文件”对应的索引节点（定义在 `kernel/fs.h`），其中字母 `d` 表示 `disk`，用来指明该结构用于存储在类似于磁盘的外存上。它对应的内存版本数据结构就是 `inode`，表示 `index node`——索引节点，我们在上文也有讲述过。`dinode` 存储在 `disk` 上的一个预先定义好的位置，通过它，可以方便地访问到和该文件相关的数据块（看到这里，大家可能会多少有些明白它的名字里为什么有一部分叫做“index”）。

`dinode` 是一个结构体，它包含了若干个成员变量：

- `type`，该索引节点对应的文件类型，可以是常规文件、目录或者设备文件（这里体现了 UNIX 的 `everything is file` 设计哲学）；
- `major`，如果 `type` 是设备类型的话，指主设备号；
- `minor`，如果 `type` 是设备类型的话，指从设备号；
- `nlink`，表示的是有多少个目录项指向该索引节点，这里可以理解为系统中有多少个“硬链接”指向该索引节点；
- `size`，表示的是文件的大小，单位是字节；
- `address1-12`——12 个“直接”数据块的地址，每个地址指向一个直接用于存放文件数据的数据块；
- `indirect`，1 个“一级间接”块的地址，该地址指向了一个间接块，该块包含了 256 个条目，每个条目分别为一个数据块的地址，所以，`xv6` 文件系统可以支持的文件最大尺寸为： $12 + 256 = 268$ 个数据块，而每个数据块的大小为 1024 字节（1KB），则 268 个数据块的大小为 $268 * 1KB = 268KB$ 。

可见，这是个很小的文件长度，实际的文件系统，文件最大的长度会大的多得多。

了解了上述基本知识后，我们可以验证一下，再 `xv6` 的终端里敲下如下命令：

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

其中，`bigfile` 是一个用户层的测试程序，它希望能够创建一个包含 65803 个块的文件，但目前，`xv6` 文件限制为 268 个块。此限制来自以下事实：一个 `xv6` 的 `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。所以很显然，该程序执行失败了。

现在本次实验的第一个任务已经浮出水面，就是要让 `xv6` 文件系统支持大文件的创建，以让上述用户层的测试程序能够正常运行。

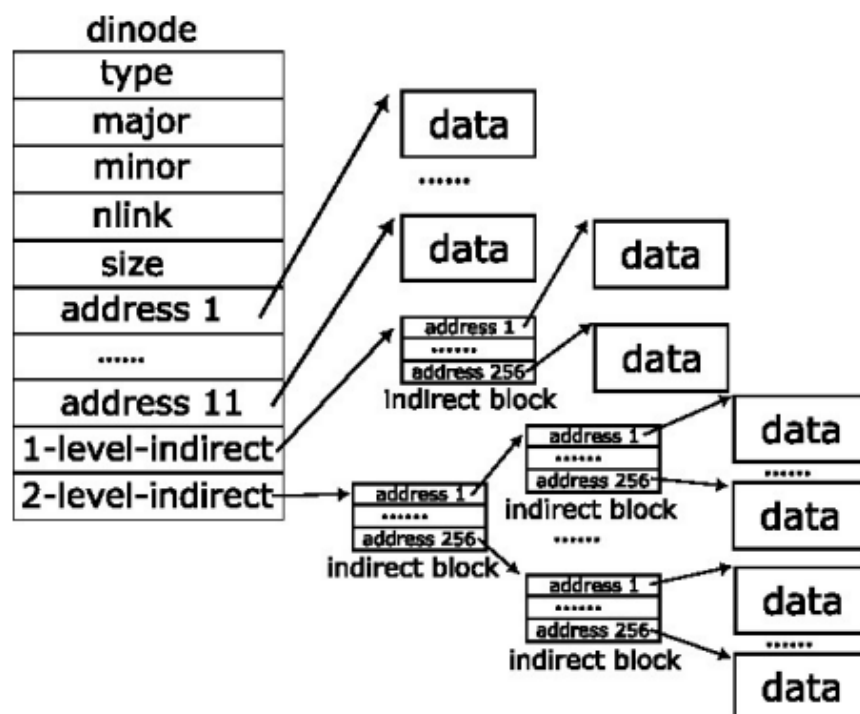
 **支持大文件的创建**

本次实验中你需要使xv6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 $256 \times 256 + 256 + 11$ 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）。

使用 `python3 grade-lab-5 bigfile` 可以测试你的程序，该测试时间可能略长，请耐心等待。

💡 一些提示和预备知识

(1) `kernel/fs.c` 中的 `bmap()` 函数是本任务的核心，在实验开始前请确保您理解 `bmap()`；它的主要工作是将在一个用户传下来的、相对的逻辑块号（针对该文件的偏移），转换为一个绝对的物理块号（针对该设备上的偏移，而一个设备可以包含多个文件）；目前，`bmap()` 里仅支持直接块和一级间接块的寻址，你需要添加对二级间接块的寻址；



(2) `dinode` 结构体里的直接块地址、一级间接块地址和二级间接块地址，都需要同时存在，你不能把原本用于“存放一级间接块地址”的成员变量直接修改为“存放二级间接块的地址”，但是可以修改一个原本用于“存放直接块地址”的成员变量，让其用于存放二级间接块地址（不能变化 `dinode` 结构体的大小）；

(3) `NDIRECT` 表示的是 `dinode` 里直接块地址的数目，而 `NINDIRECT` 表示的是 `dinode` 结构体里间接块地址的数目（都在 `kernel/fs.h` 里定义）。如果更改 `NDIRECT` 的定义，您可能必须更改 `file.h` 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。同时，如果你修改了 `NDIRECT` 的值，请重新生成 xv6 虚拟机的镜像文件——`fs.img`，生成的方法是在 xv6 的源码根目录下敲 `make clean`，它在清理一些临时的编译文件同时，会在主机系统里强制生成最新的 `fs.img`（这

个镜像文件里包含着整个 xv6 虚拟机的文件系统），其实在实验期间的任何时候你发现自己必须从头开始重建文件系统，你都可以这样做；

(4) 同样的地方，还有一个名为 `MAXFILE` 的宏定义，表示当前文件系统所支持的数据块数目，请查看此处并思考，当你要增加 xv6 文件系统对大文件的支持时，此宏定义是否也需要修改为对应值？

(5) `dinode` 是索引节点用于存放在外存上 (disk) 的结构，其实它在内存中也存在一个特定的版本——名为 `inode` 结构体 (定义在 `kernel/file.h`)，所以，当你为了支持大文件或者二级间接块修改了 `dinode` 之后，也请修改 `inode` 的结构体的相应的成员变量；（索引节点存在两个版本的原因，一个是，为了方便多进程之间的并发访问控制，你可以仔细查看下两个版本之间的代码差别，内存版本多了个锁的成员变量；另外一个原因则是，内存版本的访问速度更快，可以充当其 disk 版本的一个内存副本）；

(6) 除了分配块的 `bmap()` 函数，释放块的函数 `itrunc()` 也需要修改 (`kernel/fs.c`)，以支持释放或回收后面加入的二级间接块（可以参考 `itrunc` 是怎么处理一级间接块的）；

(7) 如果您的文件系统进入不良状态（可能是崩溃），请删除 `fs.img`（在 Unix 上执行此操作，而不是在 xv6 上执行此操作）。make 将为您构建一个新的干净的文件系统映像。

符号链接

符号链接 (Symbolic link)，又名软链接，它和硬链接之间的区别和联系在课前推荐的阅读材料里面已经有相关介绍，如果还是不清楚也可以看下面的文字，让自己对相关知识点有个大概的印象，或者也可以自行去搜索。

硬链接

- 具有相同 `inode` 节点号的多个文件互为硬链接文件；
- 删除硬链接文件或者删除源文件任意之一，文件实体并未被删除；
- 只有删除了源文件和所有对应的硬链接文件，文件实体才会被删除；
- 硬链接文件是文件的另一个入口；
- 可以通过给文件设置硬链接文件来防止重要文件被误删；
- 创建硬链接命令 `ln 源文件 硬链接文件`；
- 硬链接文件是普通文件，可以用 `rm` 删除；
- 对于静态文件（没有进程正在调用），当硬链接数为 0 时文件就被删除。（注意：如果有进程正在调用，则无法删除或者即使文件名被删除但空间不会释放。）

软连接

- 软链接类似Windows系统的快捷方式；
- 软链接里面存放的是源文件的路径，指向源文件；
- 删除源文件，软链接依然存在，但无法访问源文件内容；
- 创建软链接命令 `ln -s 源文件 软链接文件`（有一个s参数）；
- 软链接和源文件是不同的文件，文件类型也不同，inode号也不同；
- 软链接的文件类型是“l”，可以用 `rm` 删除。

软连接和硬链接的区别和联系

- 原理上，硬链接和源文件的inode号相同，两者互为硬链接；
- 软链接和源文件的inode节点号不同，进而指向的block也不同，软链接block中存放了源文件的路径名；实际上，硬链接和源文件是同一份文件，而软链接是独立的文件，类似于快捷方式，存储着源文件的位置信息便于指向；
- 使用限制：不能对目录创建硬链接，不能对不同文件系统创建硬链接，不能对不存在的文件创建硬链接；可以对目录创建软链接，可以跨文件系统创建软链接，可以对不存在的文件创建软链接。

在默认情况下，xv6已经支持对文件创建硬链接。但是符号链接在xv6中还没被创建，由此我们引入了第二个实验

在增加符号链接功能

增加一个名为 `symlink(char *target, char *path)` 的系统调用，以实现创建符号链接的功能。其中，`target` 指的是被创建符号链接的文件路径（包括文件名），而 `path` 指的是该符号链接创建后存放的路径名（包括符号链接本身文件名）。有关更多信息，请参阅 `symlink` 手册页（注：执行 `man symlink`）。

使用 `python3 grade-lab-5 symlinktest` 可以测试你的程序

一些提示和预备知识

(1) 按照lab2中给出的创建新系统调用的步骤，为 `symlink` 提供一个接口/入口，内部实现可以暂时为空（需要修改 `user/usys.pl`, `user/user.h` 和 `kernel/sysfile.c` 等文件）；

(2) 在 `kernel/stat.h` 中，创建一个名为 `(T_SYMLINK)` 文件类型，去代表 `symlink()` 所创建的符号链接类型；（大家可能会回忆起 xv6 原本支持的文件类型有三种，分别是常规文件、目录文件和设备文件，现在有了第四种）

(3) 在 `kernel/fcntl.h` 中，加入一个标志位——`O_NOFOLLOW`，用于 `open()` 在打开符号链接时，以表明打开的对象为符号链接本身，而不是该符号链接所指向的目标文件（没有该标志位的话，系统将无法区分上述两种情况，因为 `open` 函数的内部处理需要返回一个指向 inode 的指针，那么这个指针到底是应该指向符号链接文件本身，还是其所指向的对象文件？）；注意，在定

义 `O_NOFOLLOW` 时，注意不要和其他的标志位值发生冲突，因为 `open()` 系统调用可以通过“与”操作来将多个标志位进行重叠；

(4) 将 `user/` 下的 `symlinktest.c` 文件加入到 `Makefile` 里，可以编译，`xv6` 中输入 `symlinktest` 进行测试应该可以运行，但是会出错，因为 `symlink()` 系统调用的主体还没有实现；

(5) `symlink()` 打开的对象不一定非要存在，如果不存在，`symlink` 直接返回一个相关的错误就行，不必影响该系统调用的执行，类似于 `link` 和 `unlink`；

(6) 符号链接文件的“数据”应该是一个路径，所以你要选择一个合适的地方去保存着这个“路径”（这里可以借鉴向常规文件写入的函数 `writei` 去保存这个路径）；

(7) 可能存在一个嵌套或者递归的情况，你用 `open` 打开的符号链接，其指向的又是另外一个符号链接，我们最多允许 10 次递归，超过就报错；

(8) 如果存在递归，也可能存在另外一个情况，就是“循环指向”，遇到这种情况，也要坚决抛出错误（这里是否可以直接用前面最多递归 10 次的机制来报错？）

(9) 其他的系统调用，比如 `link` 或者 `unlink`，它们的操作对象如果是一个符号链接文件，那就必须作用于该符号链接本身，而不能操作该符号链接所指向的目标文件（这里要用到之前添加的标志位 `O_NOFOLLOW`）

(10) 本次实验，不用考虑符号链接文件指向一个目录的情况，只需要考虑常规文件即可（但是在实际系统中，符号链接也可以指向一个目录）。

Directory层

引入了基于文件名而非整数标识符的索引思想后，文件系统的组织方式发生了变化。这种新思想将文件按路径而非下标来索引，从而引入了目录的概念。在 `xv6` 中，目录的实现和文件的实现过程很像。目录的 `i` 节点的类型 `T_DIR`，它的数据是一系列的目录条目。每个条目是一个 `struct dirent` 结构体，包含一个名字和一个 `i` 节点编号。这个名字最多有 `DIRSIZ` 个字符；如果比较短，它将以 `NUL` 作为结尾字符。`i` 节点编号是 0 的条目都是可用的。

函数 `dirlookup`（见 `kernel/fs.c`）用于查找目录中指定名字的条目。给定一个目录的 `inode` 和需要找的文件名，首先挨个读取 `inode data block` 上的 `dirent`，然后判断是不是要找的，如果是的话，就得到了 `inode num`，通过 `iget` 返回指针。

函数 `dirlink`（见 `kernel/fs.c`）按照参数中的 `name` 和 `inum` 创建一个新的目录，如果有空闲的就写入，如果已经存在就报错。

路径名层

路径名查找通过一系列 `dirlookup` 实现。

`namei` 计算 `path` 并返回相应的 `inode`。

`nameiparent` 返回给定文件父节点的 `inode`。

`namex` 是上面两个结构体的主体部分，首先根据 `path` 起始字符判断是绝对路径还是相对路径，返回查找的起始 `inode`，然后递归查询，直到下一级不存在，最终返回 `inode`。

文件描述符层

经过以上一步步的抽象之后，`xv6`以更间接的方式提供了文件的接口。`xv6` 给每个进程都有一个自己的打开文件表，所有打开的文件保存在全局的文件表中。

文件的操作

在实际的操作系统中，有很多系统调用都涉及到了文件的操作，比如 `open`、`read`、`write`、`link` 等，调试这些系统调用是一个有助于了解文件结合文件系统进行操作的过程的好方法。

调试 `read` 系统调用

在深入理解 `xv6` 文件系统架构之后，我们可以去了解文件读写过程。我们可以通过调试 `sys_read` 函数（位于 `sysfile.c` 文件）去逐渐掌握细节。

首先，我们在 `sys_read` 函数中，具体在 `if(argfd(0, 0, &f) < 0)` 处，设置一个断点。这能够使我们在进入该分支时暂停执行，以便逐步审查程序的执行流程。

逐步调试并进入 `fileread()` 函数后，我们将焦点放在 `ilock(f->ip)`，在这里，我们关注的是确保 `inode` 在缓冲区 (buffer cache) 中的过程。因为在文件系统中进行读写操作时，我们必须确保相关的 `inode` 位于缓冲区中，否则我们需要将它从磁盘读取到缓冲区，然后再进行操作。这个过程的核心就是通过 `ilock()` 函数中判断 `ip->valid` 的值来确定 `inode` 是否在缓冲区中。

在探索这两种不同情况时，我们需要精心选择适合的断点位置。这样，我们可以清楚地观察在 `inode` 是否在缓冲区的不同状态下，程序的执行流程和数据变化。

把上述过程完整记录下来并写在实验报告之中，用自己的话总结发生了什么。

调试 write 系统调用

在我们深入探索 `sys_read()` 函数之后，接下来可以挑战更大的难题，即对 `sys_write()` 函数进行调试。我们的目标是进入 `sys_write()` 函数并跟踪到调用的 `filewrite()` 函数中。在 `filewrite()` 函数的关键位置，即位于第155行的 `int max = ((MAXOPBLOCKS-1-1-2) / 2) * BSIZE;` 处，我们将设置一个断点，以便逐步理解 `sys_write()` 的复杂过程。

`sys_write()` 函数的理解确实更为复杂，因为它牵涉到文件的更新操作。在执行文件写入时，我们需要修改inode的内容，写入数据，更新bitmap等，同时还涉及到缓冲机制和原子性操作等多个方面。这需要在调试代码的过程中，逐步深入理解，并将不同方面的操作关联起来。

我们将在调试中探索以下内容：

1. **缓冲机制：** 观察数据何时被写入缓冲区，如何更新缓冲区的内容，并确保数据的一致性和完整性。
2. **原子性操作：** 在多进程或多线程环境下，对共享资源的访问需要保证原子性。我们将关注在 `sys_write()` 中的操作是否会引起数据竞争和不一致。
3. **文件inode更新：** 我们将观察在文件写入时如何正确地更新相关的inode信息，以及如何确保文件的正确更新。
4. **数据写入和bitmap更新：** 文件写入时，涉及到数据的写入和bitmap的更新。我们将跟踪这些操作的顺序和机制。
5. **错误处理和异常情况：** 在 `sys_write()` 中，需要处理各种可能的错误情况，如磁盘空间不足、权限问题等。我们将关注这些情况下的程序行为和处理方式。

通过逐步调试，我们将更深入地理解文件写入的复杂过程，并将各个组成部分连接起来。虽然 `sys_write()` 的理解难度较大，但通过坚持调试和逐步分析，我们将能够掌握这个关键的操作系统知识点。

在上面调试了 `read` 和 `write` 两个系统调用可以对文件描述符和数据操作有更深刻的理解，当然，你也可以继续对 `open` 系统调用进行调试，有助于你了解路径解析、创建文件描述符，还可以对 `link` 系统调用进行调试了解文件元数据的操作。

文件系统是一个庞大且重要的体系，对于同学们而言，深入理解其工作原理和机制不仅仅是追求学术完善，更是为了能够在实际场景中更有效地应用和解决现实问题。上文所述只是文件系统复杂性的微观体现，我们鼓励同学们在学习过程中不断深入挖掘，以真正精通这一关键领域。

