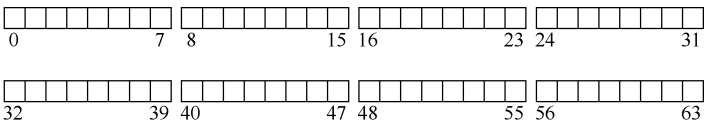
心智模型，这是系统思维的一个关键部分。在深入研究我们的第一个实现时，请尝试建立你的心智模型。

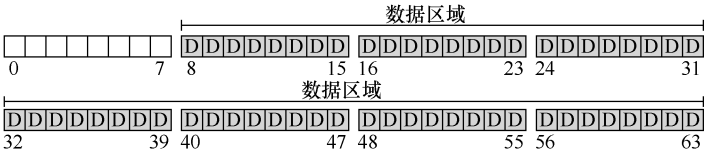
40.2 整体组织

我们现在来开发 VSFS 文件系统在磁盘上的数据结构的整体组织。我们需要做的第一件事是将磁盘分成块（block）。简单的文件系统只使用一种块大小，这里正是这样做的。我们选择常用的 4KB。

因此，我们对构建文件系统的磁盘分区的看法很简单：一系列块，每块大小为 4KB。在大小为 N 个 4KB 块的分区中，这些块的地址为从 0 到 $N-1$。假设我们有一个非常小的磁盘，只有 64 块：

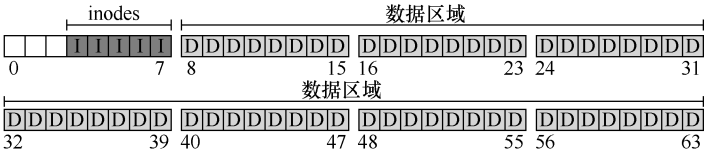


现在让我们考虑一下，为了构建文件系统，需要在这些块中存储什么。当然，首先想到的是用户数据。实际上，任何文件系统中的大多数空间都是（并且应该是）用户数据。我们将用于存放用户数据的磁盘区域称为数据区域（data region），简单起见，将磁盘的固定部分留给这些块，例如磁盘上 64 个块的最后 56 个：



正如我们在第 39 章中了解到的，文件系统必须记录每个文件的信息。该信息是元数据（metadata）的关键部分，并且记录诸如文件包含哪些数据块（在数据区域中）、文件的大小，其所有者和访问权限、访问和修改时间以及其他类似信息的事情。为了存储这些信息，文件系统通常有一个名为 inode 的结构（后面会详细介绍 inode）。

为了存放 inode，我们还需要在磁盘上留出一些空间。我们将这部分磁盘称为 inode 表（inode table），它只是保存了一个磁盘上 inode 的数组。因此，假设我们将 64 个块中的 5 块用于 inode，磁盘映像现在看起来如下：

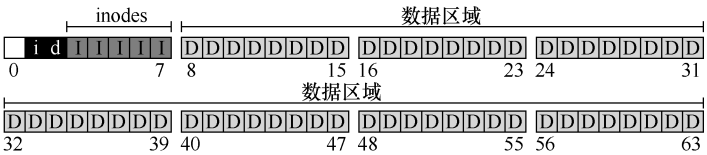


在这里应该指出，inode 通常不是那么大，例如，只有 128 或 256 字节。假设每个 inode 有 256 字节，一个 4KB 块可以容纳 16 个 inode，而我们上面的文件系统则包含 80 个 inode。在我们简单的文件系统中，建立在一个小小的 64 块分区上，这个数字表示文件系统中可以

拥有的最大文件数量。但是请注意，建立在更大磁盘上的相同文件系统可以简单地分配更大的 inode 表，从而容纳更多文件。

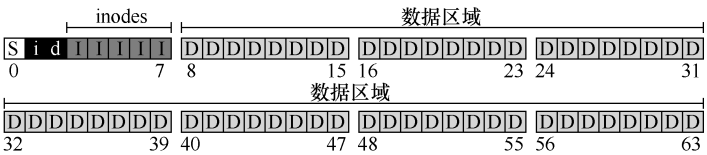
到目前为止，我们的文件系统有了数据块（D）和 inode（I），但还缺一些东西。你可能已经猜到，还需要某种方法来记录 inode 或数据块是空闲还是已分配。因此，这种分配结构（allocation structure）是所有文件系统中必需的部分。

当然，可能有许多分配记录方法。例如，我们可以用一个空闲列表（free list），指向第一个空闲块，然后它又指向下一个空闲块，依此类推。我们选择一种简单而流行的结构，称为位图（bitmap），一种用于数据区域（数据位图，data bitmap），另一种用于 inode 表（inode 位图，inode bitmap）。位图是一种简单的结构：每个位用于指示相应的对象/块是空闲（0）还是正在使用（1）。因此新的磁盘布局如下，包含 inode 位图（i）和数据位图（d）：



你可能会注意到，对这些位图使用整个 4KB 块是有点杀鸡用牛刀。这样的位图可以记录 32KB 对象是否分配，但我们只有 80 个 inode 和 56 个数据块。但是，简单起见，我们就为每个位图使用整个 4KB 块。

细心的读者可能已经注意到，在极简文件系统的磁盘结构设计中，还有一块。我们将它保留给超级块（superblock），在下图中用 S 表示。超级块包含关于该特定文件系统的信息，包括例如文件系统中有多少个 inode 和数据块（在这个例子中分别为 80 和 56）、inode 表的开始位置（块 3）等等。它可能还包括一些幻数，来标识文件系统类型（在本例中为 VSFS）。



因此，在挂载文件系统时，操作系统将首先读取超级块，初始化各种参数，然后将该卷添加到文件系统树中。当卷中的文件被访问时，系统就会知道在哪里查找所需的磁盘上的结构。

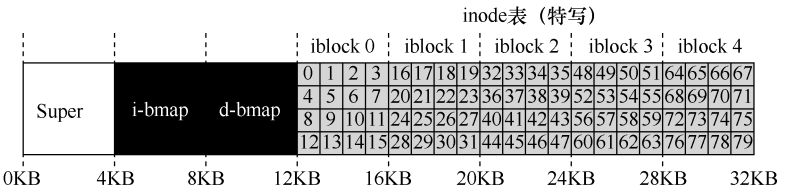
40.3 文件组织：inode

文件系统最重要的磁盘结构之一是 inode，几乎所有的文件系统都有类似的结构。名称 inode 是 index node（索引节点）的缩写，它是由 UNIX 开发人员 Ken Thompson [RT74] 给出的历史性名称，因为这些节点最初放在一个数组中，在访问特定 inode 时会用到该数组的索引。

补充：数据结构——inode

inode 是许多文件系统中使用的通用名称，用于描述保存给定文件的元数据的结构，例如其长度、权限以及其组成块的位置。这个名称至少可以追溯到 UNIX（如果不是早期的系统，可能还会追溯到 Multics）。它是 index node（索引节点）的缩写，因为 inode 号用于索引磁盘上的 inode 数组，以便查找该 inode 号对应的 inode。我们将看到，inode 的设计是文件系统设计的一个关键部分。大多数现代系统对于它们记录的每个文件都有这样的结构，但也许用了不同的名字（如 dnodes、fnodes 等）。

每个 inode 都由一个数字（称为 inumber）隐式引用，我们之前称之为文件的低级名称（low-level name）。在 VSFS（和其他简单的文件系统）中，给定一个 inumber，你应该能够直接计算磁盘上相应节点的位置。例如，如上所述，获取 VSFS 的 inode 表：大小为 20KB（5 个 4KB 块），因此由 80 个 inode（假设每个 inode 为 256 字节）组成。进一步假设 inode 区域从 12KB 开始（即超级块从 0KB 开始，inode 位图在 4KB 地址，数据位图在 8KB，因此 inode 表紧随其后）。因此，在 VSFS 中，我们为文件系统分区的开头提供了以下布局（特写视图）：



要读取 inode 号 32，文件系统首先会计算 inode 区域的偏移量（32×inode 的大小，即 8192），将它加上磁盘 inode 表的起始地址（inodeStartAddr = 12KB），从而得到希望的 inode 块的正确字节地址：20KB。回想一下，磁盘不是按字节可寻址的，而是由大量可寻址扇区组成，通常是 512 字节。因此，为了获取包含索引节点 32 的索引节点块，文件系统将向节点（即 40）发出一个读取请求，取得期望的 inode 块。更一般地说，inode 块的扇区地址 iaddr 可以计算如下：

```
blk    = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

在每个 inode 中，实际上是所有关于文件的信息：文件类型（例如，常规文件、目录等）、大小、分配给它的块数、保护信息（如谁拥有该文件以及谁可以访问它）、一些时间信息（包括文件创建、修改或上次访问的时间文件下），以及有关其数据块驻留在磁盘上的位置的信息（如某种类型的指针）。我们将所有关于文件的信息称为元数据（metadata）。实际上，文件系统中除了纯粹的用户数据外，其他任何信息通常都称为元数据。表 40.1 所示的是 ext2 [P09]的 inode 的例子。

设计 inode 时，最重要的决定之一是它如何引用数据块的位置。一种简单的方法是在 inode 中有一个或多个直接指针（磁盘地址）。每个指针指向属于该文件的一个磁盘块。这种方法有局限：例如，如果你想要一个非常大的文件（例如，大于块的大小乘以直接指针数），那就不走运了。

表 40.1 ext2 的 inode

大小（字节）	名称	inode 字段的用途
2	mode	该文件是否可以读/写/执行
2	uid	谁拥有该文件
4	size	该文件有多少字节
4	time	该文件最近的访问时间是什么时候
4	ctime	该文件的创建时间是什么时候
4	mtime	该文件最近的修改时间是什么时候
4	dtime	该 inode 被删除的时间是什么时候
2	gid	该文件属于哪个分组
2	links count	该文件有多少硬链接
4	blocks	为该文件分配了多少块
4	flags	ext2 将如何使用该 inode
4	osd1	OS 相关的字段
60	block	一组磁盘指针（共 15 个）
4	generation	文件版本（用于 NFS）
4	file acl	一种新的许可模式，除了 mode 位
4	dir acl	称为访问控制列表
4	faddr	未支持字段
12	i osd2	另一个 OS 相关字段

多级索引

为了支持更大的文件，文件系统设计者必须在 inode 中引入不同的结构。一个常见的思路是有一个称为间接指针（indirect pointer）的特殊指针。它不是指向包含用户数据的块，而是指向包含更多指针的块，每个指针指向用户数据。因此，inode 可以有一些固定数量（例如 12 个）的直接指针和一个间接指针。如果文件变得足够大，则会分配一个间接块（来自磁盘的数据块区域），并将 inode 的间接指针设置为指向它。假设一个块是 4KB，磁盘地址是 4 字节，那就增加了 1024 个指针。文件可以增长到（12 + 1024）×4KB，即 4144KB。

提示：考虑基于范围的方法

另一种方法是使用范围（extent）而不是指针。范围就是一个磁盘指针加一个长度（以块为单位）。因此，不需要指向文件的每个块的指针，只需要指针和长度来指定文件的磁盘位置。只有一个范围是有局限的，因为分配文件时可能无法找到连续的磁盘可用空间块。因此，基于范围的文件系统通常允许多个范围，从而在文件分配期间给予文件系统更多的自由。

这两种方法相比较，基于指针的方法是最灵活的，但是每个文件使用大量元数据（尤其是大文件）。基于范围的方法不够灵活但更紧凑。特别是，如果磁盘上有足够的可用空间并且文件可以连续布局（无论如何，这实际上是所有文件分配策略的目标），基于范围的方法都能正常工作。

毫不奇怪，在这种方法中，你可能希望支持更大的文件。为此，只需添加另一个指向 inode 的指针：双重间接指针（double indirect pointer）。该指针指的是一个包含间接块指针的块，每个间接块都包含指向数据块的指针。因此，双重间接块提供了可能性，允许使用额外的 1024×1024 个 4KB 块来增长文件，换言之，支持超过 4GB 大小的文件。不过，你可能想要更多，我们打赌你知道怎么办：三重间接指针（triple indirect pointer）。

总之，这种不平衡树被称为指向文件块的多级索引（multi-level index）方法。我们来看一个例子，它有 12 个直接指针，以及一个间接块和一个双重间接块。假设块大小为 4KB，并且指针为 4 字节，则该结构可以容纳一个刚好超过 4GB 的文件，即（$12 + 1024 + 1024^2$）× 4KB。增加一个三重间接块，你是否能弄清楚支持多大的文件？（提示：很大）

许多文件系统使用多级索引，包括常用的文件系统，如 Linux ext2 [P09]和 ext3，NetApp 的 WAFL，以及原始的 UNIX 文件系统。其他文件系统，包括 SGI XFS 和 Linux ext4，使用范围而不是简单的指针。有关基于范围的方案如何工作的详细信息，请参阅前面的内容（它们类似于讨论虚拟内存时的段）。

你可能想知道：为什么使用这样的不平衡树？为什么不采用不同的方法？好吧，事实证明，许多研究人员已经研究过文件系统以及它们的使用方式，几乎每次他们都发现了某些“真相”，几十年来都是如此。其中一个真相是，大多数文件很小。这种不平衡的设计反映了这样的现实。如果大多数文件确实很小，那么为这种情况优化是有意义的。因此，使用少量的直接指针（12 是一个典型的数字），inode 可以直接指向 48KB 的数据，需要一个（或多个）间接块来处理较大的文件。参见 Agrawal 等人最近的研究[A+07]。表 40.2 总结了这些结果。

表 40.2 文件系统测量汇总

大多数文件很小	大约 2KB 是常见大小
平均文件大小在增长	几乎平均增长 200KB
大多数字节保存在大文件中	少数大文件使用了大部分空间
文件系统包含许多文件	几乎平均 100KB
文件系统大约一半是满的	尽管磁盘在增长，文件系统仍保持约 50%是满的
目录通常很小	许多只有少量条目，大多数少于 20 个条目

补充：基于链接的方法

设计 inode 有另一个更简单的方法，即使用链表（linked list）。这样，在一个 inode 中，不是有多个指针，只需要一个，指向文件的第一个块。要处理较大的文件，就在该数据块的末尾添加另一个指针等，这样就可以支持大文件。

你可能已经猜到，链接式文件分配对于某些工作负载表现不佳。例如，考虑读取文件的最后一个块，或者就是进行随机访问。因此，为了使链接式分配更好地工作，一些系统在内存中保留链接信息表，而不是将下一个指针与数据块本身一起存储。该表用数据块 D 的地址来索引，一个条目的内容就是 D 的下一个指针，即 D 后面的文件中的下一个块的地址。那里也可以是空值（表示文件结束），或用其他标记来表示一个特定的块是空闲的。拥有这样的下一块指针表，使得链接分配机制可以有效地进行随机文件访问，只需首先扫描（在内存中）表来查找所需的块，然后直接访问（在磁盘上）。

这样的表听起来很熟悉吗？我们描述的是所谓的文件分配表（File Allocation Table，FAT）——文件系统的基本结构。是的，在 NTFS [C94]之前，这款经典的旧 Windows 文件系统基于简单的基于链接的分配方案。它与标准 UNIX 文件系统还有其他不同之处。例如，本身没有 inode，而是存储关于文件的元数据的目录条目，并且直接指向所述文件的第一个块，这导致不可能创建硬链接。参见 Brouwer 的著作 [B02]，了解更多不够优雅的细节。

当然，在 inode 设计的空间中，存在许多其他可能性。毕竟，inode 只是一个数据结构，任何存储相关信息并可以有效查询的数据结构就足够了。由于文件系统软件很容易改变，如果工作负载或技术发生变化，你应该愿意探索不同的设计。

40.4 目录组织

在 VSFS 中（像许多文件系统一样），目录的组织很简单。一个目录基本上只包含一个二元组（条目名称，inode 号）的列表。对于给定目录中的每个文件或目录，目录的数据块中都有一个字符串和一个数字。对于每个字符串，可能还有一个长度（假定采用可变大小的名称）。

例如，假设目录 dir（inode 号是 5）中有 3 个文件（foo、bar 和 foobar），它们的 inode 号分别为 12、13 和 24。dir 在磁盘上的数据可能如下所示：

```
inum | reclen | strlen | name
   5       4        2     .
   2       4        3     ..
  12       4        4     foo
  13       4        4     bar
  24       8        7     foobar
```

在这个例子中，每个条目都有一个 inode 号，记录长度（名称的总字节数加上所有的剩余空间），字符串长度（名称的实际长度），最后是条目的名称。请注意，每个目录有两个额外的条目：.（点）和..（点点）。点目录就是当前目录（在本例中为 dir），而点点是父目录（在本例中是根目录）。

删除一个文件（例如调用 unlink()）会在目录中间留下一段空白空间，因此应该有一些方法来标记它（例如，用一个保留的 inode 号，比如 0）。这种删除是使用记录长度的一个原因：新条目可能会重复使用旧的、更大的条目，从而在其中留有额外的空间。

你可能想知道确切的目录存储在哪里。通常，文件系统将目录视为特殊类型的文件。因此，目录有一个 inode，位于 inode 表中的某处（inode 表中的 inode 标记为“目录”的类型字段，而不是“常规文件”）。该目录具有由 inode 指向的数据块（也可能是间接块）。这些数据块存在于我们的简单文件系统的数据块区域中。我们的磁盘结构因此保持不变。

我们还应该再次指出，这个简单的线性目录列表并不是存储这些信息的唯一方法。像以前一样，任何数据结构都是可能的。例如，XFS [S+96]以 B 树形式存储目录，使文件创建操作（必须确保文件名在创建之前未被使用）快于使用简单列表的系统，因为后者必须扫描其中的条目。

40.5 空闲空间管理

文件系统必须记录哪些 inode 和数据块是空闲的，哪些不是，这样在分配新文件或目录时，就可以为它找到空间。因此，空闲空间管理（free space management）对于所有文件系统都很重要。在 VSFS 中，我们用两个简单的位图来完成这个任务。

补充：空闲空间管理

管理空闲空间可以有很多方法，位图只是其中一种。一些早期的文件系统使用空闲列表（free list），其中超级块中的单个指针保持指向第一个空闲块。在该块内部保留下一个空闲指针，从而通过系统的空闲块形成列表。在需要块时，使用头块并相应地更新列表。

现代文件系统使用更复杂的数据结构。例如，SGI 的 XFS [S+96]使用某种形式的 B 树（B-tree）来紧凑地表示磁盘的哪些块是空闲的。与所有数据结构一样，不同的时间-空间折中也是可能的。

例如，当我们创建一个文件时，我们必须为该文件分配一个 inode。文件系统将通过位图搜索一个空闲的内容，并将其分配给该文件。文件系统必须将 inode 标记为已使用（用 1），并最终用正确的信息更新磁盘上的位图。分配数据块时会发生类似的一组活动。

为新文件分配数据块时，还可能会考虑其他一些注意事项。例如，一些 Linux 文件系统（如 ext2 和 ext3）在创建新文件并需要数据块时，会寻找一系列空闲块（如 8 块）。通过找到这样一系列空闲块，然后将它们分配给新创建的文件，文件系统保证文件的一部分将在磁盘上并且是连续的，从而提高性能。因此，这种预分配（pre-allocation）策略，是为数据块分配空间时的常用启发式方法。

40.6 访问路径：读取和写入

现在我们已经知道文件和目录如何存储在磁盘上，我们应该能够明白读取或写入文件的操作过程。理解这个访问路径（access path）上发生的事情，是开发人员理解文件系统如何工作的第二个关键。请注意！

对于下面的例子，我们假设文件系统已经挂载，因此超级块已经在内存中。其他所有内容（如 inode、目录）仍在磁盘上。

从磁盘读取文件

在这个简单的例子中，让我们先假设你只是想打开一个文件（例如/foo/bar，读取它，然后关闭它）。对于这个简单的例子，假设文件的大小只有 4KB（即 1 块）。

当你发出一个 open("/foo/bar", O_RDONLY)调用时，文件系统首先需要找到文件 bar 的 inode，从而获取关于该文件的一些基本信息（权限信息、文件大小等等）。为此，文件系统

必须能够找到 inode，但它现在只有完整的路径名。文件系统必须遍历（traverse）路径名，从而找到所需的 inode。

所有遍历都从文件系统的根开始，即根目录（root directory），它就记为/。因此，文件系统的第一次磁盘读取是根目录的 inode。但是这个 inode 在哪里？要找到 inode，我们必须知道它的 i-number。通常，我们在其父目录中找到文件或目录的 i-number。根没有父目录（根据定义）。因此，根的 inode 号必须是“众所周知的”。在挂载文件系统时，文件系统必须知道它是什么。在大多数 UNIX 文件系统中，根的 inode 号为 2。因此，要开始该过程，文件系统会读入 inode 号 2 的块（第一个 inode 块）。

一旦 inode 被读入，文件系统可以在其中查找指向数据块的指针，数据块包含根目录的内容。因此，文件系统将使用这些磁盘上的指针来读取目录，在这个例子中，寻找 foo 的条目。通过读入一个或多个目录数据块，它将找到 foo 的条目。一旦找到，文件系统也会找到下一个需要的 foo 的 inode 号（假定是 44）。

下一步是递归遍历路径名，直到找到所需的 inode。在这个例子中，文件系统读取包含 foo 的 inode 及其目录数据的块，最后找到 bar 的 inode 号。open()的最后一步是将 bar 的 inode 读入内存。然后文件系统进行最后的权限检查，在每个进程的打开文件表中，为此进程分配一个文件描述符，并将它返回给用户。

打开后，程序可以发出 read()系统调用，从文件中读取。第一次读取（除非 lseek()已被调用，则在偏移量 0 处）将在文件的第一个块中读取，查阅 inode 以查找这个块的位置。它也会用新的最后访问时间更新 inode。读取将进一步更新此文件描述符在内存中的打开文件表，更新文件偏移量，以便下一次读取会读取第二个文件块，等等。

补充：读取不会访问分配结构

我们曾见过许多学生对分配结构（如位图）感到困惑。特别是，许多人经常认为，只是简单地读取文件而不分配任何新块时，也会查询位图。不是这样的！分配结构（如位图）只有在需要分配时才会访问。inode、目录和间接块具有完成读请求所需的所有信息。inode 已经指向一个块，不需要再次确认它已分配。

在某个时候，文件将被关闭。这里要做的工作要少得多。很明显，文件描述符应该被释放，但现在，这就是 FS 真正要做的。没有磁盘 I/O 发生。

整个过程如表 40.3 所示（向下时间递增）。在该表中，打开导致了多次读取，以便最终找到文件的 inode。之后，读取每个块需要文件系统首先查询 inode，然后读取该块，再使用写入更新 inode 的最后访问时间字段。花一些时间，试着理解发生了什么。

另外请注意，open 导致的 I/O 量与路径名的长度成正比。对于路径中的每个增加的目录，我们都必须读取它的 inode 及其数据。更糟糕的是，会出现大型目录。在这里，我们只需要读取一个块来获取目录的内容，而对于大型目录，我们可能需要读取很多数据块才能找到所需的条目。是的，读取文件时生活会变得非常糟糕。你会发现，写入一个文件（尤其是创建一个新文件）更糟糕。

表 40.3 文件读取时间线（向下时间递增）

	data inode	root foo bar	root foo bar bar bar
	bitmap bitmap	inode inode inode	data data data[0] data[1] data[2]
open(bar)		read read read	read read
read()		read write	read
read()		read write	read
read()		read write	read

写入磁盘

写入文件是一个类似的过程。首先，文件必须打开（如上所述）。其次，应用程序可以发出 write()调用以用新内容更新文件。最后，关闭该文件。

与读取不同，写入文件也可能会分配（allocate）一个块（除非块被覆写）。当写入一个新文件时，每次写入操作不仅需要将数据写入磁盘，还必须首先决定将哪个块分配给文件，从而相应地更新磁盘的其他结构（例如数据位图和 inode）。因此，每次写入文件在逻辑上会导致 5 个 I/O：一个读取数据位图（然后更新以标记新分配的块被使用），一个写入位图（将它的新状态存入磁盘），再是两次读取，然后写入 inode（用新块的位置更新），最后一次写入真正的数据块本身。

考虑简单和常见的操作（例如文件创建），写入的工作量更大。要创建一个文件，文件系统不仅要分配一个 inode，还要在包含新文件的目录中分配空间。这样做的 I/O 工作总量非常大：一个读取 inode 位图（查找空闲 inode），一个写入 inode 位图（将其标记为已分配），一个写入新的 inode 本身（初始化它），一个写入目录的数据（将文件的高级名称链接到它的 inode 号），以及一个读写目录 inode 以便更新它。如果目录需要增长以容纳新条目，则还需要额外的 I/O（即数据位图和新目录块）。所有这些只是为了创建一个文件！

我们来看一个具体的例子，其中创建了 file/foo/bar，并且向它写入了 3 个块。表 40.4 展示了在 open()（创建文件）期间和在 3 个 4KB 写入期间发生的情况。

在该表中，对磁盘的读取和写入放在导致它们发生的系统调用之下，它们可能发生的大致顺序从表的顶部到底部依次进行。你可以看到创建该文件需要多少工作：在这种情况下，有 10 次 I/O，用于遍历路径名，然后创建文件。你还可以看到每个分配写入需要 5 次 I/O：一对读取和更新 inode，另一对读取和更新数据位图，最后写入数据本身。文件系统如何以合理的效率完成这些任务？

表 40.4 文件创建时间线（向下时间递增）

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
			read			read				
				read			read			
create (/foo/bar)		read write					write			
					read					
					write					
				write						
write()	read write				read write			write		
write()	read write				read write			write		
write()	read write				read write					write

关键问题：如何降低文件系统 I/O 成本

即使是最简单的操作，如打开、读取或写入文件，也会产生大量 I/O 操作，分散在磁盘上。文件系统可以做些什么，来降低执行如此多 I/O 的高成本？

40.7 缓存和缓冲

如上面的例子所示，读取和写入文件可能是昂贵的，会导致（慢速）磁盘的许多 I/O。这显然是一个巨大的性能问题，为了弥补，大多数文件系统积极使用系统内存（DRAM）来缓存重要的块。

想象一下上面的打开示例：没有缓存，每个打开的文件都需要对目录层次结构中的每个级别至少进行两次读取（一次读取相关目录的 inode，并且至少有一次读取其数据）。使用长路径名（例如，/1/2/3/…/100/file.txt），文件系统只是为了打开文件，就要执行数百次读取！

早期的文件系统因此引入了一个固定大小的缓存（fixed-size cache）来保存常用的块。正如我们在讨论虚拟内存时一样，LRU 及不同变体策略会决定哪些块保留在缓存中。这个

固定大小的缓存通常会在启动时分配，大约占总内存的 10%。

然而，这种静态的内存划分（static partitioning）可能导致浪费。如果文件系统在给定的时间点不需要 10%的内存，该怎么办？使用上述固定大小的方法，文件高速缓存中的未使用页面不能被重新用于其他一些用途，因此导致浪费。

相比之下，现代系统采用动态划分（dynamic partitioning）方法。具体来说，许多现代操作系统将虚拟内存页面和文件系统页面集成到统一页面缓存中（unified page cache）[S00]。通过这种方式，可以在虚拟内存和文件系统之间更灵活地分配内存，具体取决于在给定时间哪种内存需要更多的内存。

现在想象一下有缓存的文件打开的例子。第一次打开可能会产生很多 I/O 流量，来读取目录的 inode 和数据，但是随后文件打开的同一文件（或同一目录中的文件），大部分会命中缓存，因此不需要 I/O。

我们也考虑一下缓存对写入的影响。尽管可以通过足够大的缓存完全避免读取 I/O，但写入流量必须进入磁盘，才能实现持久。因此，高速缓存不能减少写入流量，像对读取那样。虽然这么说，写缓冲（write buffering，人们有时这么说）肯定有许多优点。首先，通过延迟写入，文件系统可以将一些更新编成一批（batch），放入一组较小的 I/O 中。例如，如果在创建一个文件时，inode 位图被更新，稍后在创建另一个文件时又被更新，则文件系统会在第一次更新后延迟写入，从而节省一次 I/O。其次，通过将一些写入缓冲在内存中，系统可以调度（schedule）后续的 I/O，从而提高性能。最后，一些写入可以通过拖延来完全避免。例如，如果应用程序创建文件并将其删除，则将文件创建延迟写入磁盘，可以完全避免（avoid）写入。在这种情况下，懒惰（在将块写入磁盘时）是一种美德。

提示：理解静态划分与动态划分

在不同客户端/用户之间划分资源时，可以使用静态划分（static partitioning）或动态划分（dynamic partitioning）。静态方法简单地将资源一次分成固定的比例。例如，如果有两个可能的内存用户，则可以给一个用户固定的内存部分，其余的则分配给另一个用户。动态方法更灵活，随着时间的推移提供不同数量的资源。例如，一个用户可能会在一段时间内获得更高的磁盘带宽百分比，但是之后，系统可能会切换，决定为不同的用户提供更大比例的可用磁盘带宽。

每种方法都有其优点。静态划分可确保每个用户共享一些资源，通常提供更可预测的性能，也更易于实现。动态划分可以实现更好的利用率（通过让资源匮乏的用户占用其他空闲资源），但实现起来可能会更复杂，并且可能导致空闲资源被其他用户占用，然后在需要时花费很长时间收回，从而导致这些用户性能很差。像通常一样，没有最好的方法。你应该考虑手头的问题，并确定哪种方法最适合。实际上，你不是应该一直这样做吗？

由于上述原因，大多数现代文件系统将写入在内存中缓冲 5～30s，这代表了另一种折中：如果系统在更新传递到磁盘之前崩溃，更新就会丢失。但是，将内存写入时间延长，则可以通过批处理、调度甚至避免写入，提高性能。

某些应用程序（如数据库）不喜欢这种折中。因此，为了避免由于写入缓冲导致的意外数据丢失，它们就强制写入磁盘，通过调用 fsync()，使用绕过缓存的直接 I/O（direct I/O）

接口，或者使用原始磁盘（raw disk）接口并完全避免使用文件系统[①]。虽然大多数应用程序能接受文件系统的折中，但是如果默认设置不能令人满意，那么有足够的控制可以让系统按照你的要求进行操作。

提示：了解耐用性/性能权衡

存储系统通常会向用户提供耐用性/性能折中。如果用户希望写入的数据立即持久，则系统必须尽全力将新写入的数据提交到磁盘，因此写入速度很慢（但是安全）。但是，如果用户可以容忍丢失少量数据，系统可以缓冲内存中的写入一段时间，然后将其写入磁盘（在后台）。这样做可以使写入快速完成，从而提高感受到的性能。但是，如果发生崩溃，尚未提交到磁盘的写入操作将丢失，因此需要进行折中。要理解如何正确地进行这种折中，最好了解使用存储系统的应用程序需要什么。例如，虽然丢失网络浏览器下载的最后几张图像可以忍受，但丢失部分数据库交易、让你的银行账户不能增加资金，这不能忍。当然，除非你很有钱。如果你很有钱，为什么要特别关心积攒每一分钱？

40.8 小结

我们已经看到了构建文件系统所需的基本机制。需要有关于每个文件（元数据）的一些信息，这通常存储在名为 inode 的结构中。目录只是“存储名称→inode 号”映射的特定类型的文件。其他结构也是需要的。例如，文件系统通常使用诸如位图的结构，来记录哪些 inode 或数据块是空闲的或已分配的。

文件系统设计的极好方面是它的自由。接下来的章节中探讨的文件系统，都利用了这种自由来优化文件系统的某些方面。显然，我们还有很多尚未探讨的策略决定。例如，创建一个新文件时，它应该放在磁盘上的什么位置？这一策略和其他策略会成为未来章节的主题吗？

参考资料

[A+07] Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch A Five-Year Study of File-System Metadata

FAST ’07, pages 31–45, February 2007, San Jose, CA

最近对文件系统实际使用方式的一个很好的分析。利用其中的文献目录可以追溯到 20 世纪 80 年代早期的文件系统分析论文。

[B07]“ZFS: The Last Word in File Systems”Jeff Bonwick and Bill Moore

最新的重要文件系统之一，功能丰富，性能卓越。我们应该为它写一章，也许很快就会有这么一章。

① 选修一门数据库课程，了解更多有关传统数据库的知识，以及它们过去对避开操作系统和自己控制一切的坚持。但要小心！有些搞数据库的人总是试图说操作系统的坏话。

[B02]“The FAT File System”Andries Brouwer, September, 2002
关于 FAT 的很好、很漂亮的描述。文件系统的类型，不是培根的类型。但你必须承认，培根可能味道更好。

[C94]“Inside the Windows NT File System”, Helen Custer
Microsoft Press, 1994
一本关于 NTFS 的小书，其他书中可能有更多技术 s 细节。

[H+88]“Scale and Performance in a Distributed File System”
John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West.
ACM Transactions on Computing Systems (ACM TOCS), page 51-81, Volume 6, Number 1, February 1988
经典的分布式文件系统，我们稍后会更多地了解它，不用担心。

[P09]“The Second Extended File System: Internal Layout”Dave Poirier, 2009
有关 ext2 的详细信息，这是一个非常简单的基于 FFS 的 Linux 文件系统，即 Berkeley Fast File System。我们将在第 41 章中详细解读。

[RT74]“The UNIX Time-Sharing System”
M．Ritchie and K. Thompson
CACM, Volume 17:7, pages 365-375, 1974
关于 UNIX 的较早的论文。阅读它，能了解许多现代操作系统的基础知识。

[S00]“UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD”Chuck Silvers
FREENIX, 2000
一篇关于 NetBSD 集成文件系统缓冲区缓存和虚拟内存页面缓存的好文章。许多其他系统做了同样的事情。

[S+96]“Scalability in the XFS File System”
Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, Geoff Peck
USENIX ’96, January 1996, San Diego, CA
第一次尝试让操作具有可伸缩性，其中包括在目录中拥有数百万个文件这样的事情，这是核心关注点。它是一个把想法推向极致的好例子。这个文件系统的关键思想是：一切都是树。我们也应该为这个文件系统写一章内容。

作业

使用工具 vsfs.py 来研究文件系统状态如何随着各种操作的发生而改变。文件系统以空状态开始，只有一个根目录。模拟发生时，会执行各种操作，从而慢慢改变文件系统的磁盘状态。详情请参阅 README 文件。

问题

1．用一些不同的随机种子（比如 17、18、19、20）运行模拟器，看看你是否能确定每次状态变化之间一定发生了哪些操作。

2．现在使用不同的随机种子（比如 21、22、23、24），但使用-r 标志运行，这样做可以让你在显示操作时猜测状态的变化。关于 inode 和数据块分配算法，根据它们喜欢分配的块，你可以得出什么结论？

3．现在将文件系统中的数据块数量减少到非常少（比如两个），并用 100 个左右的请求来运行模拟器。在这种高度约束的布局中，哪些类型的文件最终会出现在文件系统中？什么类型的操作会失败？

4．现在做同样的事情，但针对 inodes。只有非常少的 inode，什么类型的操作才能成功？哪些通常会失败？文件系统的最终状态可能是什么？

第 41 章　局部性和快速文件系统

当 UNIX 操作系统首次引入时，UNIX“魔法师”Ken Thompson 编写了第一个文件系统。我们称之为“老 UNIX 文件系统”，它非常简单，基本上，它的数据结构在磁盘上看起来像这样：

超级块（S）包含有关整个文件系统的信息：卷的大小、有多少 inode、指向空闲列表块的头部的指针等等。磁盘的 inode 区域包含文件系统的所有 inode。最后，大部分磁盘都被数据块占用。

老文件系统的好处在于它很简单，支持文件系统试图提供的基本抽象：文件和目录层次结构。与过去笨拙的基于记录的存储系统相比，这个易于使用的系统真正向前迈出了一步。与早期系统提供的更简单的单层次层次结构相比，目录层次结构是真正的进步。

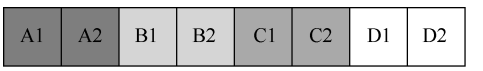
41.1　问题：性能不佳

问题是：性能很糟糕。根据 Kirk McKusick 和他在伯克利的同事[MJLF84]的测量，性能开始就不行，随着时间的推移变得更糟，直到文件系统仅提供总磁盘带宽的 2%！

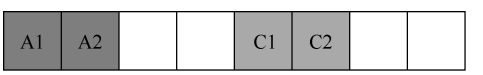
主要问题是老 UNIX 文件系统将磁盘当成随机存取内存。数据遍布各处，而不考虑保存数据的介质是磁盘的事实，因此具有实实在在的、昂贵的定位成本。例如，文件的数据块通常离其 inode 非常远，因此每当第一次读取 inode 然后读取文件的数据块（非常常见的操作）时，就会导致昂贵的寻道。

更糟糕的是，文件系统最终会变得非常碎片化（fragmented），因为空闲空间没有得到精心管理。空闲列表最终会指向遍布磁盘的一堆块，并且随着文件的分配，它们只会占用下一个空闲块。结果是在磁盘上来回访问逻辑上连续的文件，从而大大降低了性能。

例如，假设以下数据块区域包含 4 个文件（A、B、C 和 D），每个文件大小为两个块：



如果删除 B 和 D，则生成的布局为：



如你所见，可用空间被分成两块构成的两大块，而不是很好的连续 4 块。假设我们现在希望分配一个大小为 4 块的文件 E：