【Linux】从Linux文件系统看文件读写过程 #18

Open justtreee opened this issue on 3 Aug 2018 · 4 comments

Assignees

Labels Linux

justtreee commented on 3 Aug 2018

从Linux文件系统看文件读写过程

> 提问： 在一个 txt 文件中，修改其中一个字，然后保存，这期间计算机内部到底发生了什么？

1. 答案：文件读写基本流程

1.1 读文件

1. 进程调用库函数向内核发起读文件请求；
2. 内核通过检查进程的文件描述符定位到虚拟文件系统的已打开文件列表表项；
3. 调用该文件可用的系统调用函数read()
4. read()函数通过文件表项链接到目录项模块，根据传入的文件路径，在目录项模块中检索，找到该文件的inode；
5. 在inode中，通过文件内容偏移量计算出要读取的页；
6. 通过inode找到文件对应的address_space；
7. 在address_space中访问该文件的页缓存树，查找对应的页缓存结点：
    - 如果页缓存命中，那么直接返回文件内容；
    - 如果页缓存缺失，那么产生一个页缺失异常，创建一个页缓存页，同时通过inode找到文件该页的磁盘地址，读取相应的页填充该缓存页；重新进行第6步查找页缓存；
8. 文件内容读取成功。

7. 如果页缓存命中，直接把文件内容修改更新在页缓存的页中。写文件就结束了。这时候文件修改位于页缓存，并没有写回到磁盘文件中去。

8. 如果页缓存缺失，那么产生一个页缺失异常，创建一个页缓存页，同时通过inode找到文件该页的磁盘地址，读取相应的页填充该缓存页。此时缓存页命中，进行第6步。

9. 一个页缓存中的页如果被修改，那么会被标记成脏页。脏页需要写回到磁盘中的文件块。有两种方式可以把脏页写回磁盘：

    - 手动调用sync()或者fsync()系统调用把脏页写回
    - pdflush进程会定时把脏页写回到磁盘

同时注意，脏页不能被置换出内存，如果脏页正在被写回，那么会被设置写回标记，这时候该页就被上锁，其他写请求被阻塞直到锁释放。

> 这里出现了几个概念：系统调用、虚拟文件系统中的innode、页缓冲Page Cache和Address Space

2.1 系统调用

操作系统的主要功能是为管理硬件资源和为应用程序开发人员提供良好的环境，但是计算机系统的各种硬件资源是有限的，因此为了保证每一个进程都能安全的执行。处理器设有两种模式：“用户模式”与“内核模式”。一些容易发生安全问题的操作都被限制在只有内核模式下才可以执行，例如I/O操作，修改基址寄存器内容等。**而连接用户模式和内核模式的接口称之为系统调用。**

应用程序代码运行在用户模式下，当应用程序需要实现内核模式下的指令时，先向操作系统发送调用请求。操作系统收到请求后，执行系统调用接口，使处理器进入内核模式。当处理器处理完系统调用操作后，操作系统会让处理器返回用户模式，继续执行用户代码。

进程的虚拟地址空间可分为两部分，内核空间和用户空间。内核空间中存放的是内核代码和数据，而进程的用户空间中存放的是用户程序的代码和数据。不管是内核空间还是用户空间，它们都处于虚拟空间中，都是对物理地址的映射。

应用程序中实现对文件的操作过程就是典型的系统调用过程。

附：linux的用户模式和内核模式

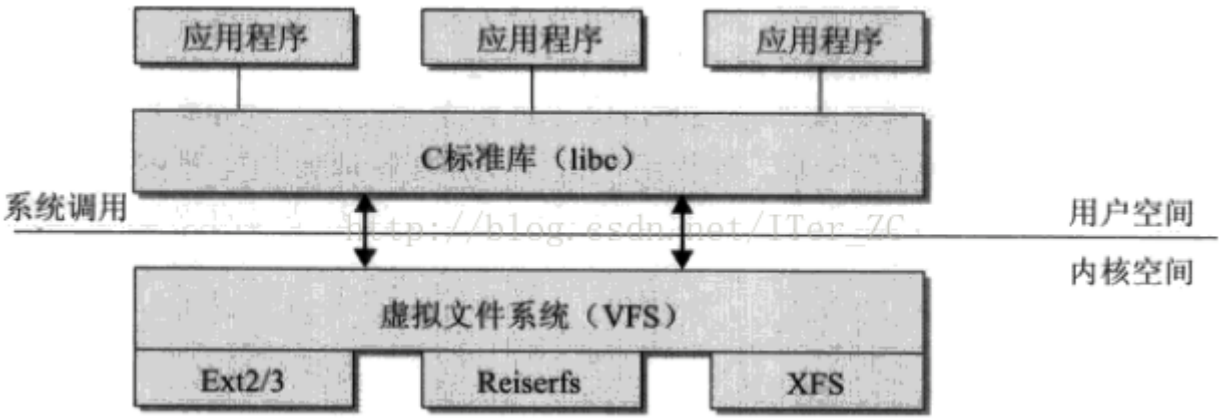
> 在Linux机器上，CPU要么处于受信任的内核模式，要么处于受限制的用户模式。除了内核本身处于内核模式以外，所有的用户进程都运行在用户模式之中。
>
> 处理器总处于以下状态中的一种：
>
> 1、内核态，运行于进程上下文，内核代表进程运行于内核空间；
>
> 2、内核态，运行于中断上下文，内核代表硬件运行于内核空间；
>
> 3、用户态，运行于用户空间。

> 时的环境等。
>
> 硬件通过触发信号，导致内核调用中断处理程序，进入内核空间。这个过程中，硬件的一些变量和参数也要传递给内核，内核通过这些参数进行中断处理。所谓的“中断上下文”，其实也可以看作就是硬件传递过来的这些参数和内核需要保存的一些其他环境（主要是当前被打断执行的进程环境）。

2.2 虚拟文件系统

一个操作系统可以支持多种底层不同的文件系统（比如NTFS, FAT, ext3, ext4），为了给内核和用户进程提供统一的文件系统视图，Linux在用户进程和底层文件系统之间加入了一个抽象层，即虚拟文件系统(Virtual File System, VFS)，进程所有的文件操作都通过VFS，由VFS来适配各种底层不同的文件系统，完成实际的文件操作。

通俗的说，VFS就是定义了一个通用文件系统的接口层和适配层，一方面为用户进程提供了一组统一的访问文件，目录和其他对象的统一方法，另一方面又要和不同的底层文件系统进行适配。如图所示：



Linux的EXT2文件系统

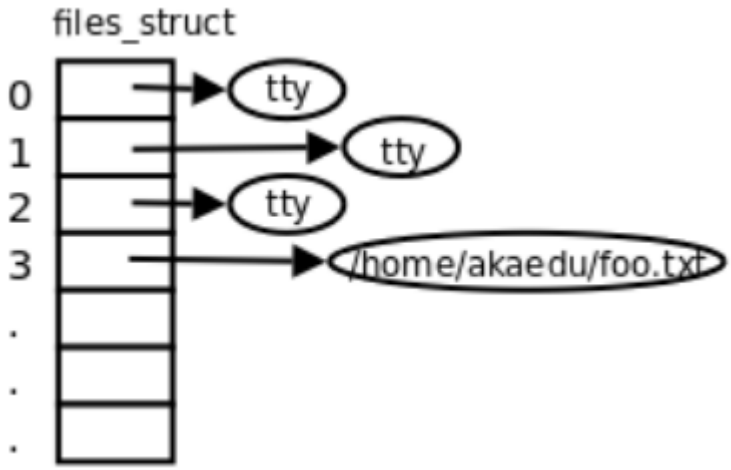
对于一个磁盘分区来说，在被指定为相应的文件系统后，整个分区被分为 1024，2048 和 4096 字节大小的块。根据块使用的不同，可分为：

- **超级块**(Superblock): 这是整个文件系统的第一块空间。包括整个文件系统的基本信息，如块大小，inode/block的总量、使用量、剩余量，指向空间 inode 和数据块的指针等相关信息。
- **inode块**(文件索引节点) : 文件系统索引,记录文件的属性。它是文件系统的最基本单元，是文件系统连接任何子目录、任何文件的桥梁。每个子目录和文件只有唯一的一个 inode 块。它包含了文件系统中文件的基本属性(文件的长度、创建及修改时间、权限、所属关系)、存放数据的位置等相关信息. 在 Linux 下可以通过 "ls -li" 命令查看文件的 inode 信息。硬连接和源文件具有相同的 inode 。
- **数据块**(Block) :实际记录文件的内容，若文件太大时，会占用多个 block。为了提高目录访问效率，Linux 还提供了表达路径与 inode 对应关系的 dentry 结构。它描述了路径信息并连接到节点 inode，它包括各种目录信息，还指向了 inode 和超级块。

就像一本书有封面、目录和正文一样。在文件系统中，超级块就相当于封面，从封面可以得知这本书的基本信息； inode 块相当于目录，从目录可以得知各章节内容的位置；而数据块则相当于书的正文，记录着具体内容。

体中维护了一个 files的指针（和“已打开文件列表”上的表项是不同的指针）来指向结构体files_struct，files_struct中包含文件描述符表和打开的文件对象信息。

2. file_struct中的文件描述符表实际是一个file类型的指针列表（和“已打开文件列表”上的表项是相同的指针），可以支持动态扩展，每一个指针指向虚拟文件系统中文件列表模块的某一个已打开的文件。



3. file结构一方面可从f_dentry链接到目录项模块以及inode模块，获取所有和文件相关的信息，另一方面链接file_operations子模块，其中包含所有可以使用的系统调用函数，从而最终完成对文件的操作。这样，从进程到进程的文件描述符表，再关联到已打开文件列表上对应的文件结构，从而调用其可执行的系统调用函数，实现对文件的各种操作。

进程、文件列表与Inode

1. 多个进程可以同时指向一个打开文件对象（文件列表表项），例如父进程和子进程间共享文件对象；

2. 一个进程可以多次打开一个文件，生成不同的文件描述符，每个文件描述符指向不同的文件列表表项。但是由于是同一个文件，inode唯一，所以这些文件列表表项都指向同一个inode。通过这样的方法实现文件共享（共享同一个磁盘文件）；

2.3 I/O 缓冲区

概念

如高速缓存（cache）产生的原理类似，在I/O过程中，读取磁盘的速度相对内存读取速度要慢的多。因此为了能够加快处理数据的速度，需要将读取过的数据缓存在内存里。而这些缓存在内存里的数据就是高速缓冲区（buffer cache），下面简称为“buffer”。

具体来说，buffer（缓冲区）是一个用于存储速度不同步的设备或优先级不同的设备之间传输数据的区域。一方面，通过缓冲区，可以使进程之间的相互等待变少，从而使从速度慢的设备读入数据时，速度快的设备的操作进程不发生间断。另一方面，可以保护硬盘或减少网络传输的次数。

Buffer和Cache

Buffer Cache和 Page Cache

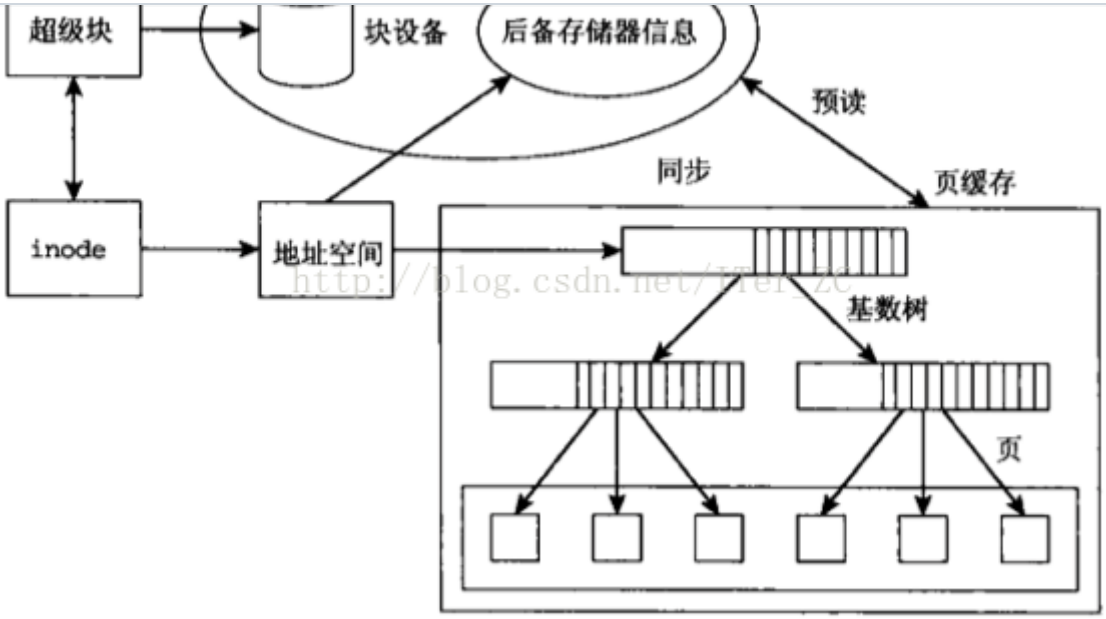
buffer cache和page cache都是为了处理设备和内存交互时高速访问的问题。buffer cache可称为块缓冲器，page cache可称为页缓冲器。在linux不支持虚拟内存机制之前，还没有页的概念，因此缓冲区以块为单位对设备进行。在linux采用虚拟内存的机制来管理内存后，页是虚拟内存管理的最小单位，开始采用页缓冲的机制来缓冲内存。Linux2.6之后内核将这两个缓存整合，页和块可以相互映射，同时，页缓存page cache面向的是虚拟内存，块I/O缓存Buffer cache是面向块设备。需要强调的是，页缓存和块缓存对进程来说就是一个存储系统，进程不需要关注底层的设备的读写。

buffer cache和page cache两者最大的区别是缓存的粒度。buffer cache面向的是文件系统的块。而内核的内存管理组件采用了比文件系统的块更高级别的抽象：页page，其处理的性能更高。因此和内存管理交互的缓存组件，都使用页缓存。

2.4 页缓存Page Cache

页缓存是面向文件，面向内存的。通俗来说，**它位于内存和文件之间缓冲区**，文件IO操作实际上只和page cache交互，不直接和内存交互。page cache可以用在所有以文件为单元的场景下，比如网络文件系统等等。page cache通过一系列的数据结构，比如inode, address_space, struct page，实现将一个文件映射到页的级别：

1. struct page结构标志一个物理内存页，通过page + offset就可以将此页帧定位到一个文件中的具体位置。同时struct page还有以下重要参数：
    - 标志位flags来记录该页是否是脏页，是否正在被写回等等；
    - mapping指向了地址空间address_space，表示这个页是一个页缓存中页，和一个文件的地址空间对应；
    - index记录这个页在文件中的页偏移量；
2. 文件系统的inode实际维护了这个文件所有的块block的块号，通过对文件偏移量offset取模可以很快定位到这个偏移量所在的文件系统的块号，磁盘的扇区号。同样，通过对文件偏移量offset进行取模可以计算出偏移量所在的页的偏移量。
3. page cache缓存组件抽象了地址空间address_space这个概念来作为文件系统和页缓存的中间桥梁。地址空间address_space通过指针可以方便的获取文件inode和struct page的信息，所以可以很方便地定位到一个文件的offset在各个组件中的位置，即通过：文件字节偏移量 --> 页偏移量 --> 文件系统块号 block --> 磁盘扇区号
4. 页缓存实际上就是采用了一个基数树结构将一个文件的内容组织起来存放在物理内存struct page中。一个文件inode对应一个地址空间address_space。而一个address_space对应一个页缓存基数树。



2.5 Address Space

下面我们总结已经讨论过的address_space所有功能。address_space是Linux内核中的一个关键抽象，它被作为文件系统和页缓存的中间适配器，用来指示一个文件在页缓存中已经缓存了的物理页。因此，它是页缓存和外部设备中文件系统的桥梁。如果将文件系统可以理解成数据源，那么address_space可以说关联了内存系统和文件系统。

由图中可以看到，地址空间address_space链接到页缓存基数树和inode，因此address_space通过指针可以方便的获取文件inode和page的信息。那么页缓存是如何通过address_space实现缓冲区功能的？我们再来看完整的文件读写流程。

3 回顾

应用程序需要修改文件A中的部分字段，首先应用程序将待写数据存放在其user buffer结构中，user buffer 通过MMU 映射，数据实际存放在物理内存中。现在应用程序需要将待写数据写入硬盘。

1. 程序进程调用内核函数write()，将待写文件标识（句柄）、待写数据相对文件首部的字节偏移量（offset xx)、待写数据长度(2KB)和待写数据的位置一并传给内核；

   > 注：在程序打开文件时，内核在PageCache中创建一个虚拟的文件 A′，这个文件A′从文件系统inode结构（下文讲）中映射出来，由若干个page组成，初始情况下文件A′存在与逻辑地址空间内，不占用物理内存。文件A′的逻辑长度参考文件实际长度占用page的整数倍，这里假设page大小为4KB。

2. 内核根据文件字节偏移量和上文提到的虚拟文件A′计算出待写数据占用的page1；（这里面待写数据只有2KB，小于page大小，因此待写数据落入page1中）

3. 计算出page号后，内核尝试找到page1对应的物理地址，以进行下一步操作。此时发现page1对应的数据并没有调入内存中，产生缺页，此时需要内核将page1对应的数据完整的从磁盘调入内存；（注意：此处和内存换页没有关系，这里可以看到使用操作系统Pagecache的写IO可能会产生IO读惩罚）

> 一定是连续的。因此文件系统需要一个链表来记录文件对应的物理块位置，这个结构在linux中就是inode

4. 文件系统将page号映射到对应的块，然后根据inode可查到文件块对应的真实物理地址LBA，内核将请求封装后转给设备驱动层（此步内核需要将Page所包含的所有字节都调入内存——"Page对齐"）。
5. 设备驱动将请求翻译成若干各个SCSI指令，驱动SAS控制器通过SAS总线向磁盘发送指令：
   SCSI Read() LBA0x****** Len=N N=读取字节大小/扇区大小

> 注：上述过程主要发生在CPU与内存之间，CPU从内存中读出指令并执行，最后CPU将指令通过PCIe总线发送给了SAS控制器，SAS控制器将指令发送到SAS总线上

6. 磁盘收到SCSI指令后，找到LBA对应的实际盘面和柱面，读出对应的扇区，发回SAS控制器；
7. 从磁盘读出的数据（这里是4KB大小）从原路返回，最后写入到page1对应的物理内存中；
8. 内核用代写2KB数据替换掉Page1对应的2KB待替换数据；
9. 此时内核向程序进程反馈：写入成功；
10. 内核在合适时机将内存中的脏页刷入磁盘。

> 注：9、10两步表示 Write Back模式，内核在没有将数据写入磁盘时就返回写入成功，以提高效率，相当于内核"欺骗"了应用程序。实际上不光内核会这样做，底层的很多环节也会有这样的情况，比如磁盘也会"欺骗"SAS控制器。如果此时发生系统掉电，所有易失性存储中的数据全部丢失，并未写入磁盘，而应用程序认为写IO已经完成了，下次开机时就会产生数据不一致。程序可以设置Write Through 模式，此时内核会等底层层层上报写入成功后，才会反馈写入成功。

参考链接

- linux的用户模式和内核模式
- 从内核文件系统看文件读写过程
- 计算机文件读写原理


justtreee added the Linux label on 3 Aug 2018


justtreee self-assigned this on 3 Aug 2018

hhyvs111 commented on 9 Nov 2019

总结的很好

通过inode存储的不是文件基本属性和 和block信息吗，相当于索引?
怎么找到address_space呢?

justtreee commented on 1 Dec 2019

> 通过inode存储的不是文件基本属性和 和block信息吗，相当于索引?
> 怎么找到address_space呢?

@Aleafboat
一句话：inode管磁盘，address_space接内存。两者互相指针链接。

Inode是文件系统（VFS）下的概念，通过“一个inode对应一个文件”使得文件管理按照类似索引的这种树形结构进行管理，通过inode快速的找到文件在磁盘扇区的位置；但是这种管理机制并不能满足读写的要求，因为我们修改文件的时候是先修改在内存里的，所以就有了页缓存机制，作为内存与文件的缓冲区。“address_space模块，它表示一个文件在页缓存中已经缓存了的物理页。它是页缓存和外部设备中文件系统的桥梁。如果将文件系统可以理解成数据源，那么address_space可以说关联了内存系统和文件系统。（见引用：https://www.cnblogs.com/huxiao-tee/p/4657851.html）”

ironboxer commented on 4 Feb 2021

文件读取的时候， 权限的判断在哪一步呢?

Assignees

justtreee

Labels

Linux

Projects

None yet

Milestone

No milestone

Development

No branches or pull requests