xiaolincoding.com

7.1 文件系统全家桶

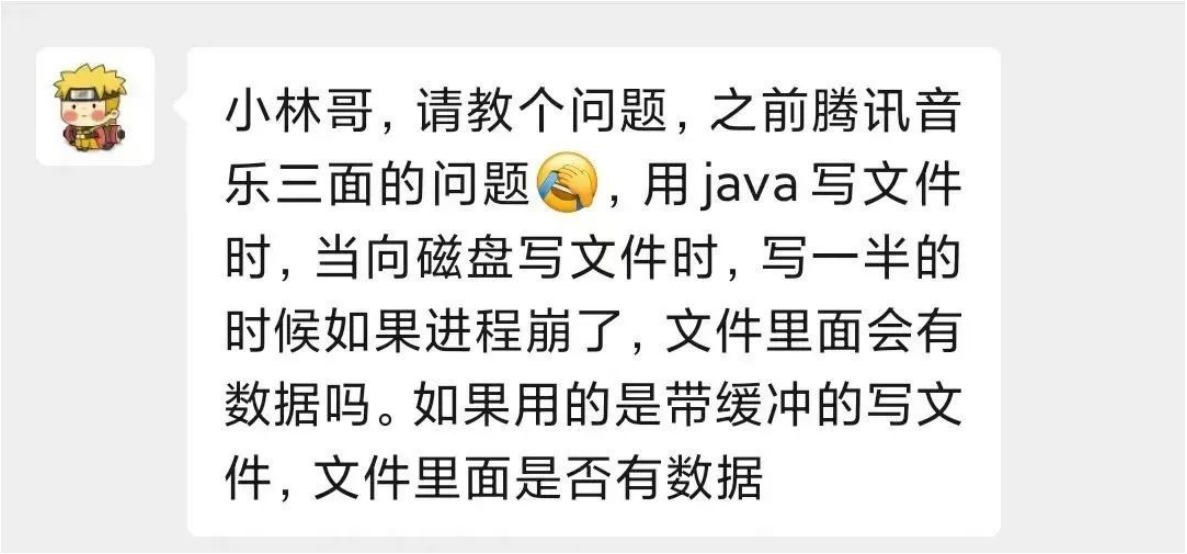
小林coding

26-32 minutes

7.2 进程写文件时，进程发生了崩溃，已写入的数据会丢失吗?

大家好，我是小林。

前几天，有位读者问了我这么个问题：



大概就是，进程写文件（使用缓冲 IO）过程中，写一半的时候，进程发生了崩溃，已写入的数据会丢失吗?

答案，是不会的。

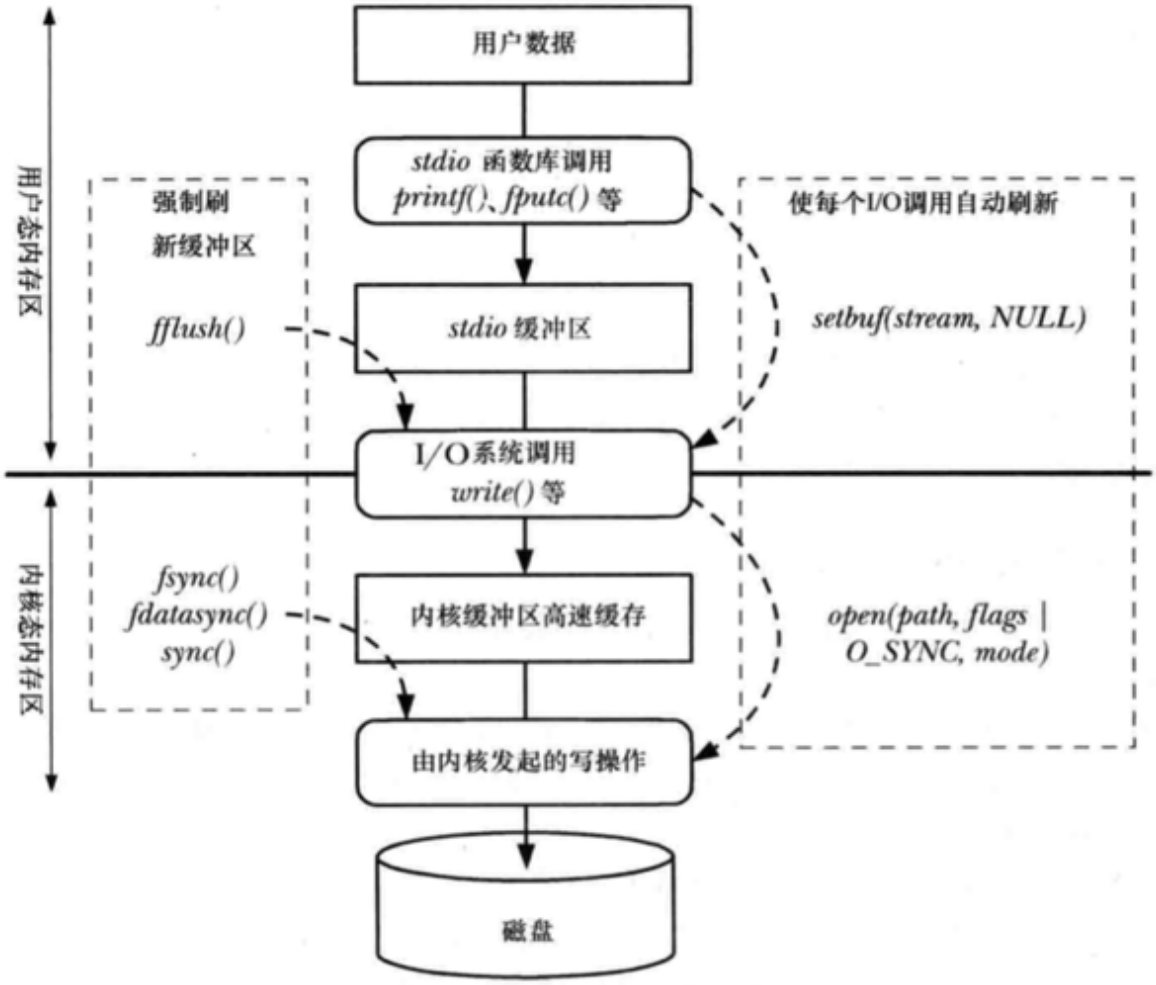


图 13-1：I/O 缓冲小结

因为进程在执行 write （使用缓冲 IO）系统调用的时候，实际上是将文件数据写到了内核的 page cache，它是文件系统中用于缓存文件数据的缓冲，所以即使进程崩溃了，文件数据还是保留在内核的 page cache，我们读数据的时候，也是从内核的 page cache 读取，因此还是依然读的进程崩溃前写入的数据。

内核会找个合适的时机，将 page cache 中的数据持久化到磁盘。但是如果 page cache 里的文件数据，在持久化到磁盘化到磁盘之前，系统发生了崩溃，那这部分数据就会丢失了。

当然， 我们也可以在程序里调用 fsync 函数，在写文文件的时候，立刻将文件数据持久化到磁盘，这样就可以解决系统崩溃导致的文件数据丢失的问题。

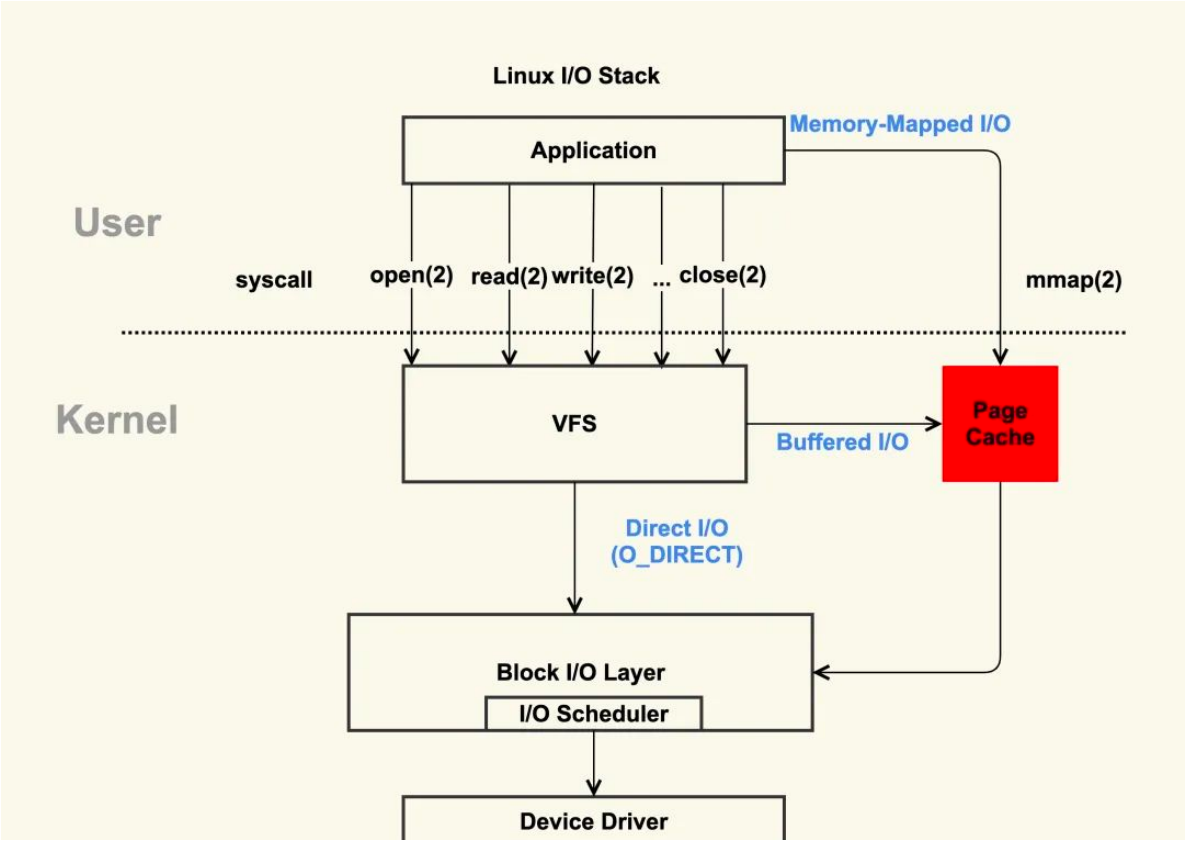
我在网上看到一篇介绍 page cache 很好的文章， 分享给大家一起学习。

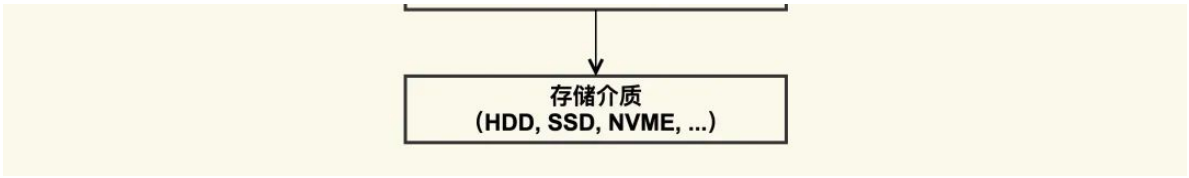
> 作者：spongecaptain
>
> 原文地址：Linux 的 Page Cache

Page Cache

Page Cache 是什么?

为了理解 Page Cache，我们不妨先看一下 Linux 的文件 I/O 系统，如下图所示：





上图中，红色部分为 Page Cache。可见 Page Cache 的本质是由 Linux 内核管理的内存区域。我们通过 mmap 以及 buffered I/O 将文件读取到内存空间实际上都是读取到 Page Cache 中。

如何查看系统的 Page Cache?

通过读取 `/proc/meminfo` 文件，能够实时获取系统内存情况：

根据上面的数据，你可以简单得出这样的公式（等式两边之和都是 112696 KB）：

两边等式都是 Page Cache，即：

通过阅读下面的小节，就能够理解为什么 SwapCached 与 Buffers 也是 Page Cache 的一部分。

page 与 Page Cache

page 是内存管理分配的基本单位， Page Cache 由多个 page 构成。page 在操作系统中通常为 4KB 大小（32bits/64bits），而 Page Cache 的大小则为 4KB 的整数倍。

另一方面，并不是所有 page 都被组织为 Page

Cache。

Linux 系统上供用户可访问的内存分为两个类型，即：

- File-backed pages：文件备份页也就是 Page Cache 中的 page，对应于磁盘上的若干数据块；对于这些页最大的问题是脏页回盘；
- Anonymous pages：匿名页不对应磁盘上的任何磁盘数据块，它们是进程的运行是内存空间（例如方法栈、局部变量表等属性）；

为什么 Linux 不把 Page Cache 称为 block cache，这不是更好吗?

这是因为从磁盘中加载到内存的数据不仅仅放在 Page Cache 中，还放在 buffer cache 中。

例如通过 Direct I/O 技术的磁盘文件就不会进入 Page Cache 中。当然，这个问题也有 Linux 历史设计的原因，毕竟这只是一个称呼，含义随着 Linux 系统的演进也逐渐不同。

下面比较一下 File-backed pages 与 Anonymous pages 在 Swap 机制下的性能。

内存是一种珍惜资源，当内存不够用时，内存管理单元（Memory Mangament Unit）需要提供调度算法来回收相关内存空间。内存空间回收的方式通常就是 swap，即交换到持久化存储设备上。

File-backed pages（Page Cache）的内存回收代价较低。Page Cache 通常对应于一个文件上的若干顺序块，因此可以通过顺序 I/O 的方式落盘。另一方面，如果 Page Cache 上没有进行写操作（所谓的没有脏页），甚至不会将 Page Cache 回盘，因为数据的内容完全可以通过再次读取磁盘文件得到。

Page Cache 的主要难点在于脏页回盘，这个内容会在后面进行详细说明。

Anonymous pages 的内存回收代价较高。这是因为 Anonymous pages 通常随机地写入持久化交换设备。另一方面，无论是否有写操作，为了确保数据不丢失，Anonymous pages 在 swap 时必须持久化到磁盘。

Swap 与缺页中断

Swap 机制指的是当物理内存不够用，内存管理单元（Memory Mangament Unit，MMU）需要提供调度算法来回收相关内存空间，然后将清理出来的内存空间给当前内存申请方。

Swap 机制存在的本质原因是 Linux 系统提供了虚拟内存管理机制，每一个进程认为其独占内存空间，因此所有进程的内存空间之和远远大于物理内存。所有进程的内存空间之和超过物理内存的部分就需要交换到磁盘上。

操作系统以 page 为单位管理内存，当进程发现需要访问的数据不在内存时，操作系统可能会将数据以页的方式加载到内存中。上述过程被称为**缺页中断**，当操作系统发生缺页中断时，就会通过系统调用将 page 再次读到内存中。

但主内存的空间是有限的，当主内存中不包含可以使用的空间时，操作系统会从选择合适的物理内存页驱逐回磁盘，为新的内存页让出位置，**选择待驱逐页的过程在操作系统中叫做页面替换（Page Replacement）**，替换操作又会触发 swap 机制。

如果物理内存足够大，那么可能不需要 Swap 机制，但是 Swap 在这种情况下还是有一定优势：对于有发生内存泄漏几率的应用程序（进程），Swap 交换分区更是重要，这可以确保内存泄露不至于导致物理内存不够用，最终导致系统崩溃。但内存泄露会引起频繁的 swap，此时非常影响操作系统的性能。

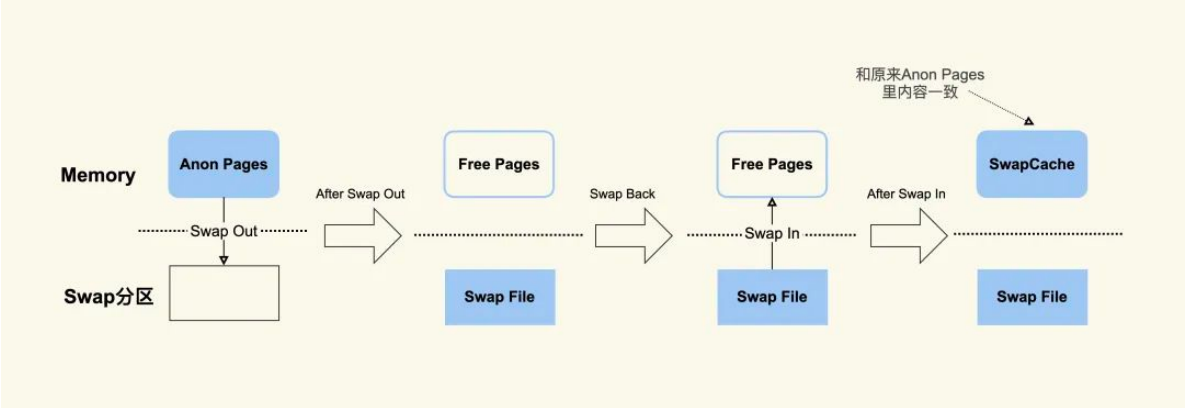
Linux 通过一个 swappiness 参数来控制 Swap 机制：这个参数值可为 0-100，控制系统 swap 的优先级：

- 高数值：较高频率的 swap，进程不活跃时主动将其转换出物理内存。
- 低数值：较低频率的 swap，这可以确保交互式不因为内存空间频繁地交换到磁盘而提高响应延迟。

最后，为什么 SwapCached 也是 Page Cache 的一部

分?

这是因为当匿名页（Inactive(anon) 以及 Active(anon)）先被交换（swap out）到磁盘上后，然后再加载回（swap in）内存中，由于读入到内存后原来的 Swap File 还在，所以 SwapCached 也可以认为是 File-backed page，即属于 Page Cache。这个过程如下图所示。



Page Cache 与 buffer cache

执行 free 命令，注意到会有两列名为 buffers 和 cached，也有一行名为 “-/+ buffers/cache”。

其中，cached 列表示当前的页缓存（Page Cache）占用量，buffers 列表示当前的块缓存（buffer cache）占用量。

用一句话来解释：**Page Cache 用于缓存文件的页数据，buffer cache 用于缓存块设备（如磁盘）的块数据。**

- 页是逻辑上的概念，因此 Page Cache 是与文件系统同

级的；

- 块是物理上的概念，因此 buffer cache 是与块设备驱动程序同级的。

  Page Cache 与 buffer cache 的**共同目的都是加速数据 I/O**：

- 写数据时首先写到缓存，将写入的页标记为 dirty，然后向外部存储 flush，也就是缓存写机制中的 write-back（另一种是 write-through，Linux 默认情况下不采用）；
- 读数据时首先读取缓存，如果未命中，再去外部存储读取，并且将读取来的数据也加入缓存。操作系统总是积极地将所有空闲内存都用作 Page Cache 和 buffer cache，当内存不够用时也会用 LRU 等算法淘汰缓存页。

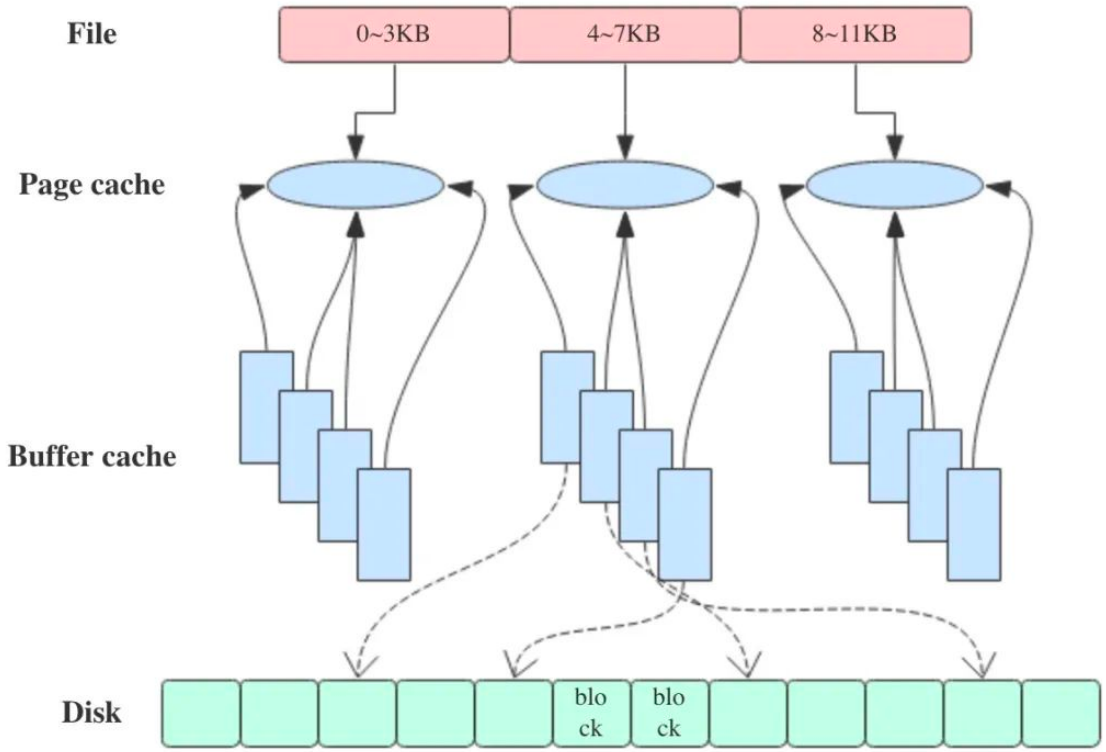
  在 Linux 2.4 版本的内核之前，Page Cache 与 buffer cache 是完全分离的。但是，块设备大多是磁盘，磁盘上的数据又大多通过文件系统来组织，这种设计导致很多数据被缓存了两次，浪费内存。

  所以在 2.4 版本内核之后，两块缓存近似融合在了一起：如果一个文件的页加载到了 Page Cache，那么同时 buffer cache 只需要维护块指向页的指针就可以了。只有那些没有文件表示的块，或者绕过了文件系统直接操作（如dd命令）的块，才会真正放到 buffer cache

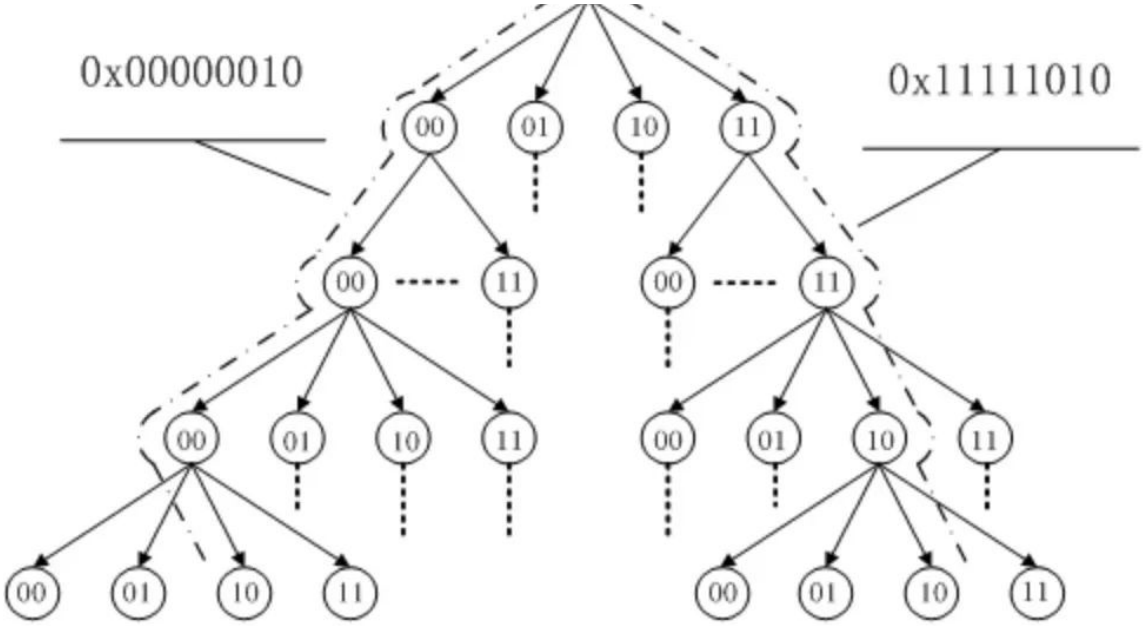
里。

因此，**我们现在提起 Page Cache，基本上都同时指 Page Cache 和 buffer cache 两者，本文之后也不再区分，直接统称为 Page Cache。**

下图近似地示出 32-bit Linux 系统中可能的一种 Page Cache 结构，其中 block size 大小为 1KB，page size 大小为 4KB。



Page Cache 中的每个文件都是一棵基数树（radix tree，本质上是多叉搜索树），树的每个节点都是一个页。根据文件内的偏移量就可以快速定位到所在的页，如下图所示。关于基数树的原理可以参见英文维基，这里就不细说了。

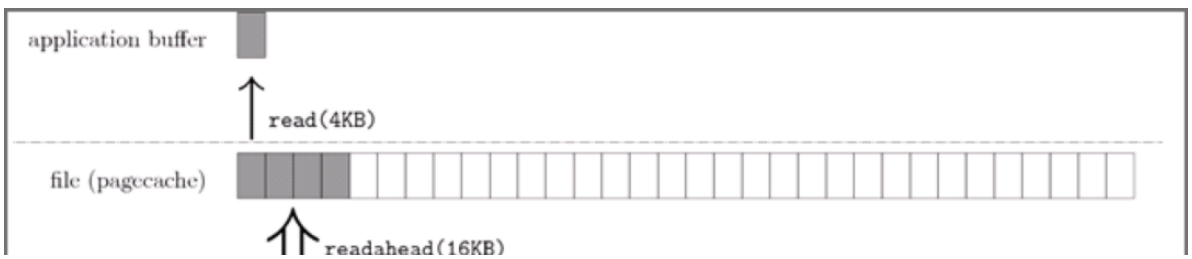


Page Cache 与预读

操作系统为基于 Page Cache 的读缓存机制提供**预读机制**（PAGE_READAHEAD），一个例子是：

- 用户线程仅仅请求读取磁盘上文件 A 的 offset 为 0-3KB 范围内的数据，由于磁盘的基本读写单位为 block（4KB），于是操作系统至少会读 0-4KB 的内容，这恰好可以在一个 page 中装下。
- 但是操作系统出于局部性原理会选择将磁盘块 offset [4KB,8KB)、[8KB,12KB) 以及 [12KB,16KB) 都加载到内存，于是额外在内存中申请了 3 个 page；

下图代表了操作系统的预读机制：



上图中，应用程序利用 read 系统调动读取 4KB 数据，实际上内核使用 readahead 机制完成了 16KB 数据的读取。

Page Cache 与文件持久化的一致性&可靠性

现代 Linux 的 Page Cache 正如其名，是对磁盘上 page（页）的内存缓存，同时可以用于读/写操作。

任何系统引入缓存，就会引发一致性问题：内存中的数据与磁盘中的数据不一致，例如常见后端架构中的 Redis 缓存与 MySQL 数据库就存在一致性问题。

Linux 提供多种机制来保证数据一致性，但无论是单机上的内存与磁盘一致性，还是分布式组件中节点 1 与节点 2 、节点 3 的数据一致性问题，理解的关键是 trade-off：吞吐量与数据一致性保证是一对矛盾。

首先，需要我们理解一下文件的数据。**文件 = 数据 + 元数据**。元数据用来描述文件的各种属性，也必须存储在磁盘上。因此，我们说保证文件一致性其实包含了两个方面：数据一致+元数据一致。

> 文件的元数据包括：文件大小、创建时间、访问时间、属主属组等信息。

我们考虑如下一致性问题：如果发生写操作并且对应的

数据在 Page Cache 中，那么写操作就会直接作用于 Page Cache 中，此时如果数据还没刷新到磁盘，那么内存中的数据就领先于磁盘，此时对应 page 就被称为 Dirty page。

当前 Linux 下以两种方式实现文件一致性：

1. **Write Through（写穿）**：向用户层提供特定接口，应用程序可主动调用接口来保证文件一致性；
2. **Write back（写回）**：系统中存在定期任务（表现形式为内核线程），周期性地同步文件系统中文件脏数据块，这是默认的 Linux 一致性方案；

上述两种方式最终都依赖于系统调用，主要分为如下三种系统调用：

方法	含义
fsync(intfd)	fsync(fd)：将 fd 代表的文件的脏数据和脏元数据全部刷新至磁盘中。
fdatasync(intfd)	fdatasync(fd)：将 fd 代表的文件的脏数据刷新至磁盘，同时对必要的元数据刷新至磁盘中，这里所说的必要的概念是指：对接下来访问文件有关键作用的信息，如文件大小，而文件修改时间等不属于必要信息
sync()	sync()：则是对系统中所有的脏的文件

方法	含义
	数据元数据刷新至磁盘中

上述三种系统调用可以分别由用户进程与内核进程发起。下面我们研究一下内核线程的相关特性。

1. 创建的针对回写任务的内核线程数由系统中持久存储设备决定，为每个存储设备创建单独的刷新线程；
2. 关于多线程的架构问题，Linux 内核采取了 Lighthttp 的做法，即系统中存在一个管理线程和多个刷新线程（每个持久存储设备对应一个刷新线程）。管理线程监控设备上的脏页面情况，若设备一段时间内没有产生脏页面，就销毁设备上的刷新线程；若监测到设备上有脏页面需要回写且尚未为该设备创建刷新线程，那么创建刷新线程处理脏页面回写。而刷新线程的任务较为单调，只负责将设备中的脏页面回写至持久存储设备中。
3. 刷新线程刷新设备上脏页面大致设计如下：

- 每个设备保存脏文件链表，保存的是该设备上存储的脏文件的 inode 节点。所谓的回写文件脏页面即回写该 inode 链表上的某些文件的脏页面；
- 系统中存在多个回写时机，第一是应用程序主动调用回写接口（fsync，fdatasync 以及 sync 等），第二管理线程周期性地唤醒设备上的回写线程进行回写，第三是某些应用程序/内核任务发现内存不足时要回收部分缓存页面而事先进行脏页面回写，设计一个统一的框架来管理

这些回写任务非常有必要。

Write Through 与 Write back 在持久化的可靠性上有所不同：

- Write Through 以牺牲系统 I/O 吞吐量作为代价，向上层应用确保一旦写入，数据就已经落盘，不会丢失；
- Write back 在系统发生宕机的情况下无法确保数据已经落盘，因此存在数据丢失的问题。不过，在程序挂了，例如被 kill -9，Page Cache 中的数据操作系统还是会确保落盘；

Page Cache 的优劣势

Page Cache 的优势

1.加快数据访问

如果数据能够在内存中进行缓存，那么下一次访问就不需要通过磁盘 I/O 了，直接命中内存缓存即可。

由于内存访问比磁盘访问快很多，因此加快数据访问是 Page Cache 的一大优势。

2.减少 I/O 次数，提高系统磁盘 I/O 吞吐量

得益于 Page Cache 的缓存以及预读能力，而程序又往往符合局部性原理，因此通过一次 I/O 将多个 page 装入 Page Cache 能够减少磁盘 I/O 次数， 进而提高系统磁盘 I/O 吞吐量。

Page Cache 的劣势

page cache 也有其劣势，最直接的缺点是需要占用额外物理内存空间，物理内存在比较紧俏的时候可能会导致频繁的 swap 操作，最终导致系统的磁盘 I/O 负载的上升。

Page Cache 的另一个缺陷是对应用层并没有提供很好的管理 API，几乎是透明管理。应用层即使想优化 Page Cache 的使用策略也很难进行。因此一些应用选择在用户空间实现自己的 page 管理，而不使用 page cache，例如 MySQL InnoDB 存储引擎以 16KB 的页进行管理。

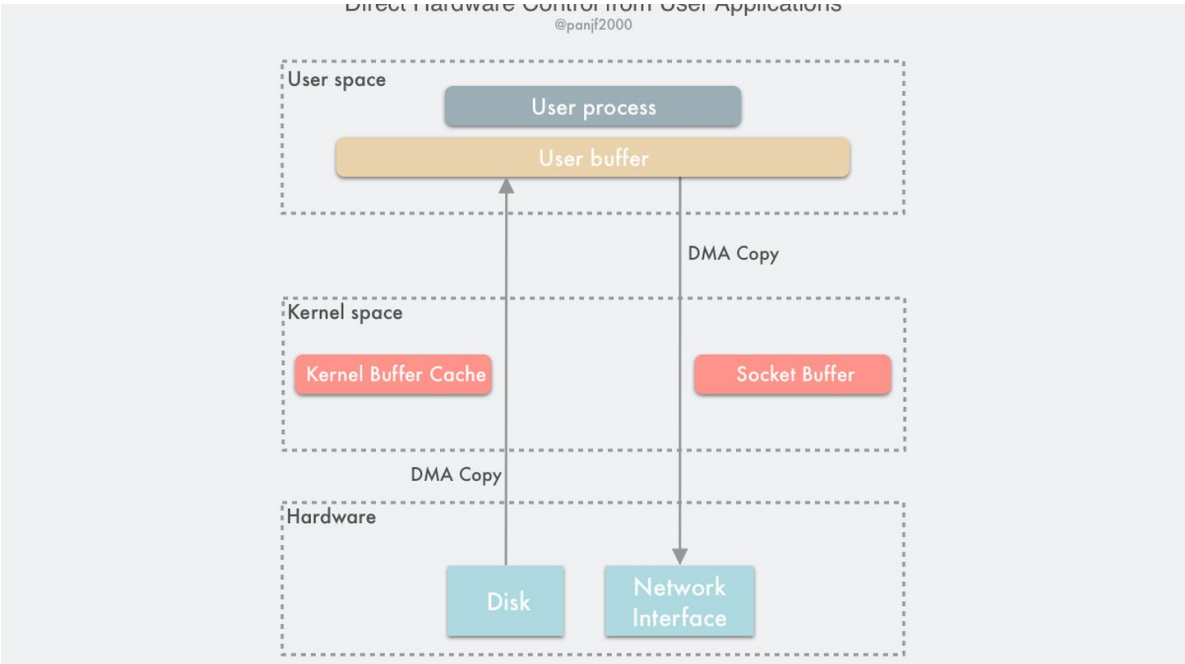
Page Cache 最后一个缺陷是在某些应用场景下比 Direct I/O 多一次磁盘读 I/O 以及磁盘写 I/O。

Direct I/O 即直接 I/O。其名字中的”直接”二字用于区分使用 page cache 机制的缓存 I/O。

- 缓存文件 I/O：用户空间要读写一个文件并**不直接**与磁盘交互，而是中间夹了一层缓存，即 page cache；
- 直接文件 I/O：用户空间读取的文件**直接**与磁盘交互，没有中间 page cache 层；

“直接”在这里还有另一层语义：其他所有技术中，数据至少需要在内核空间存储一份，但是在 Direct I/O 技术中，数据直接存储在用户空间中，绕过了内核。

Direct I/O 模式如下图所示：



此时用户空间直接通过 DMA 的方式与磁盘以及网卡进行数据拷贝。

Direct I/O 的读写非常有特点：

- Write 操作：由于其不使用 page cache，所以其进行写文件，如果返回成功，数据就真的落盘了（不考虑磁盘自带的缓存）；
- Read 操作：由于其不使用 page cache，每次读操作是真的从磁盘中读取，不会从文件系统的缓存中读取。

参考资料

- Linux内核技术实战课
- Reconsidering swapping
- 访问局部性
- DMA 与零拷贝技术

哈喽，我是小林，就爱图解计算机基础，如果觉得文章对你有帮助，欢迎微信搜索「小林coding」，关注后，回复「网络」再送你图解网络 PDF