Linux 内核的文件预读

007-11-20 10:05 吴峰光 软件世界

Linux 文件预读算法磁盘 I/O 性能的发展远远滞后于 CPU 和内存，因而成为现代计算机系统的一个主要瓶颈。预读可以有效的减少磁盘的寻道次数和应用程序的 I/O 等待时间，是改进磁盘读 I/O 性能的重要优化手段之一

编者按：Linux 文件预读算法磁盘 I/O 性能的发展远远滞后于 CPU 和内存，因而成为现代计算机系统的一个主要瓶颈。预读可以有效的减少磁盘的寻道次数和应用程序的 I/O 等待时间，是改进磁盘读 I/O 性能的重要优化手段之一。本文作者是中国科学技术大学自动化系的博士生，他在 1998 年开始学习 Linux，为了优化服务器的性能，他开始尝试改进 Linux kernel，并最终重写了内核的文件预读部分，这些改进被收录到 Linux Kernel 2.6.23 及其后续版本中。

从寄存器、L1/L2 高速缓存、内存、闪存，到磁盘/光盘/磁带/存储网络，计算机的各级存储器硬件组成了一个金字塔结构。越是底层存储容量越大。然而访问速度也越慢，具体表现为更小的带宽和更大的延迟。因而这很自然的便成为一个金字塔形的逐层缓存结构。由此产生了三类基本的缓存管理和优化问题：

◆预取(prefetching)算法，从慢速存储中加载数据到缓存；
◆替换(replacement)算法，从缓存中丢弃无用数据；
◆写回(writeback)算法，把脏数据从缓存中保存到慢速存储。

其中的预取算法，在磁盘这一层次尤为重要。磁盘的机械臂+旋转盘片的数据定位与读取方式，决定了它最突出的性能特点:擅长顺序读写，不善于随机 I/O，I/O 延迟非常大。由此而产生了两个方面的预读需求。

来自磁盘的需求

简单的说，磁盘的一个典型 I/O 操作由两个阶段组成：

1.数据定位

平均定位时间主要由两部分组成：平均寻道时间和平均转动延迟。寻道时间的典型值是 4.6ms。转动延迟则取决于磁盘的转速：普通 7200RPM 桌面硬盘的转动延迟是 4.2ms，而高端 10000RPM 的是 3ms。这些数字多年来一直徘徊不前，大概今后也无法有大的改善了。在下文中，我们不妨使用 8ms 作为典型定位时间。

2.数据传输

持续传输率主要取决于盘片的转速（线速度）和存储密度，最新的典型值为 80MB/s。虽然磁盘转速难以提高，但是存储密度却在逐年改善。巨磁阻、垂直磁记录等一系列新技术的采用，不但大大提高了磁盘容量，也同时带来了更高的持续传输率。

显然，I/O 的粒度越大，传输时间在总时间中的比重就会越大，因而磁盘利用率和吞吐量就会越大。简单的估算结果如表 1 所示。如果进行大量 4KB 的随机 I/O，那么磁盘在 99%以上的时间内都在忙着定位，单个磁盘的吞吐量不到 500KB/s。但是当 I/O 大小达到 1MB 的时候，吞吐量可接近 50MB/s。由此可见，采用更大的 I/O 粒度，可以把磁盘的利用效率和吞吐量提高整整 100 倍。因而必须尽一切可能避免小尺寸 I/O，这正是预读算法所要做的。

I/O大小	定位时间	传输时间	磁盘利用率	吞吐量	IOPS
4KB	8ms	0.05ms	0.6%	497KB/s	124
32KB	8ms	0.39ms	4.7%	3814KB/s	119
128KB	8ms	1.56ms	16.3%	13386KB/s	105
512KB	8ms	6.25ms	43.9%	35930KB/s	70
1024KB	8ms	12.50ms	61.0%	49951KB/s	49

表 1　　随机读大小与磁盘性能的关系

来自程序的需求

应用程序处理数据的一个典型流程是这样的:while(!done) { read(); compute(); }。假设这个循环要重复 5 次，总共处理 5 批数据，则程序运行的时序图可能如图 1 所示。

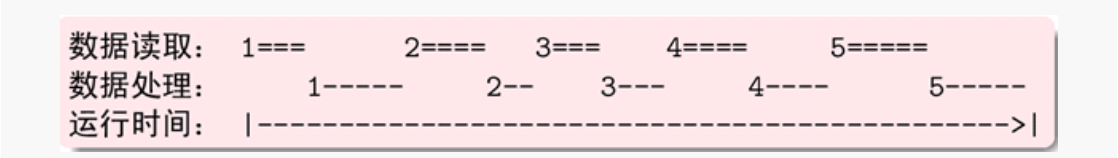

图 1 典型的 I/O 时序图

不难看出，磁盘和 CPU 是在交替忙碌：当进行磁盘 I/O 的时候，CPU 在等待；当 CPU 在计算和处理数据时，磁盘是空闲的。那么是不是可以让两者流水线作业，以便加快程序的执行速度？预读可以帮助达成这一目标。基本的方法是，当 CPU 开始处理第 1 批数据的时候，由内核的预读机制预加载下一批数据。这时候的预读是在后台异步进行的，如图 2 所示。

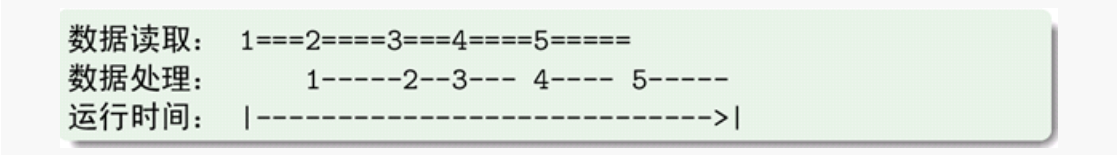

图 2 预读的流水线作业

注意，在这里我们并没有改变应用程序的行为：程序的下一个读请求仍然是在处理完当前的数据之后才发出的。只是这时候的被请求的数据可能已经在内核缓存中了，无须等待，直接就能复制过来用。在这里，异步预读的功能是对上层应用程序“隐藏”磁盘 I/O 的大延迟。虽然延迟事实上仍然存在，但是应用程序看不到了，因而运行的更流畅。

预读的概念

预取算法的涵义和应用非常广泛。它存在于 CPU、硬盘、内核、应用程序以及网络的各个层次。预取有两种方案：启发性的(heuristic prefetching)和知情的(informed prefetching)。前者自动自发的进行预读决策，对上层应用是透明的，但是对算法的要求较高，存在命中率的问题；后者则简单的提供 API 接口，而由上层程序给予明确的预读指示。在磁盘这个层次，Linux 为我们提供了三个 API 接口：posix_fadvise(2), readahead(2), madvise(2)。

不过真正使用上述预读 API 的应用程序并不多见：因为一般情况下，内核中的启发式算法工作的很好。预读(readahead)算法预测即将访问的页面，并提前把它们批量的读入缓存。

它的主要功能和任务可以用三个关键词来概括：

◆批量，也就是把小 I/O 聚集为大 I/O，以改善磁盘的利用率，提升系统的吞吐量。
◆提前，也就是对应用程序隐藏磁盘的 I/O 延迟，以加快程序运行。
◆预测，这是预读算法的核心任务。前两个功能的达成都有赖于准确的预测能力。当前包括 Linux、FreeBSD 和 Solaris 等主流操作系统都遵循了一个简单有效的原则：把读模式分为随机读和顺序读两大类，并只对顺序读进行预读。这一原则相对保守，但是可以保证很高的预读命中率，同时有效率/覆盖率也很好。因为顺序读是最简单而普遍的，而随机读在内核来说也确实是难以预测的。

Linux 的预读架构

Linux 内核的一大特色就是支持最多的文件系统，并拥有一个虚拟文件系统(VFS)层。早在 2002 年，也就是 2.5 内核的开发过程中，Andrew Morton 在 VFS 层引入了文件预读的基本框架，以统一支持各个文件系统。如图所示，Linux 内核会将它最近访问过的文件页面缓存在内存中一段时间，这个文件缓存被称为 pagecache。如图 3 所示。一般的 read()操作发生在应用程序提供的缓冲区与 pagecache 之间。而预读算法则负责填充这个 pagecache。

应用程序的读缓存一般都比较小，比如文件拷贝命令 cp 的读写粒度就是 4KB；内核的预读算法则会以它认为更合适的大小进行预读 I/O，比比如 16-128KB。

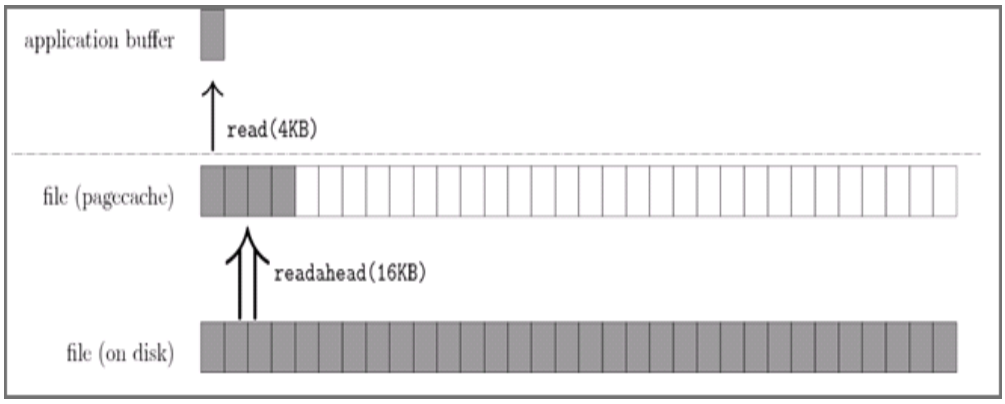

图 3 以 pagecache 为中心的读和预读

大约一年之后，Linus Torvalds 把 mmap 缺页 I/O 的预取算法单独列出，从而形成了 read-around/read-ahead 两个独立算法（图 4）。read-around 算法适用于那些以 mmap 方式访问的程序代码和数据，它们具有很强的局域性(locality of reference)特征。当有缺页事件发生时，它以当前页面为中心，往前往后预取共计 128KB 页面。而 readahead 算法主要针对 read()系统调用，它们一般都具有很好的顺序特性。但是随机和非典型的读取模式也大量存在，因而 readahead 算法必须具有很好的智能和适应性。

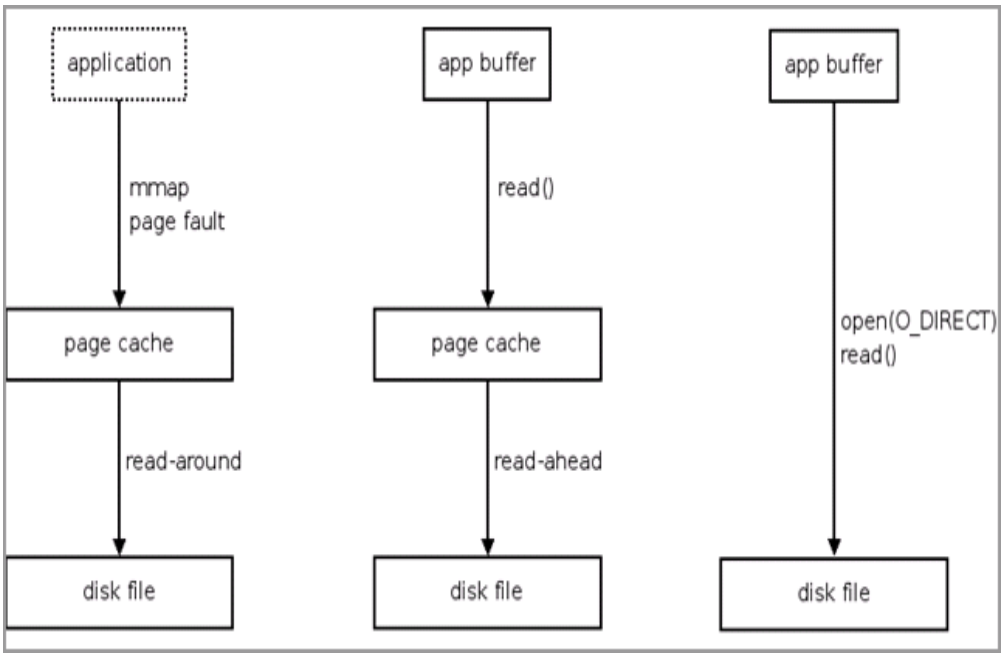

图 4 Linux 中的 read-around, read-ahead 和 direct read

又过了一年，通过 Steven Pratt、Ram Pai 等人的大量工作，readahead 算法进一步完善。其中最重要的一点是实现了对随机读的完好支持。随机读在数据库应用中处于非常突出

的地位。在此之前，预读算法以离散的读页面位置作为输入，一个多页面的随机读会触发“顺序预读”。这导致了预读 I/O 数的增加和命中率的下降。改进后的算法通过监控所有完整的 read()调用，同时得到读请求的页面偏移量和数量，因而能够更好的区分顺序读和随机读。

预读算法概要

这一节以 linux 2.6.22 为例，来剖析预读算法的几个要点。

1.顺序性检测

为了保证预读命中率，Linux 只对顺序读(sequential read)进行预读。内核通过验证如下两个条件来判定一个 read()是否顺序读：

◆这是文件被打开后的第一次读，并且读的是文件首部；

◆当前的读请求与前一（记录的）读请求在文件内的位置是连续的。

如果不满足上述顺序性条件，就判定为随机读。任何一个随机读都将终止当前的顺序序列，从而终止预读行为（而不是缩减预读大小）。注意这里的空间顺序性说的是文件内的偏移量，而不是指物理磁盘扇区的连续性。在这里 Linux 作了一种简化，它行之有效的基本前提是文件在磁盘上是基本连续存储的，没有严重的碎片化。

2.流水线预读

当程序在处理一批数据时，我们希望内核能在后台把下一批数据事先准备好，以便 CPU 和硬盘能流水线作业。Linux 用两个预读窗口来跟踪当前顺序流的预读状态：current 窗口和 ahead 窗口。其中的 ahead 窗口便是为流水线准备的：当应用程序工作在 current 窗口时，内核可能正在 ahead 窗口进行异步预读；一旦程序进入当前的 ahead 窗口，内核就会立即往前推进两个窗口，并在新的 ahead 窗口中启动预读 I/O。

3.预读的大小

当确定了要进行顺序预读(sequential readahead)时，就需要决定合适的预读大小。预读粒度太小的话，达不到应有的性能提升效果；预读太多，又有可能载入太多程序不需要的页面，造成资源浪费。为此，Linux 采用了一个快速的窗口扩张过程：

◆首次预读： readahead_size = read_size * 2; // or *4

预读窗口的初始值是读大小的二到四倍。这意味着在您的程序中使用较大的读粒度（比如 32KB）可以稍稍提升 I/O 效率。

◆后续预读： readahead_size *= 2;

后续的预读窗口将逐次倍增，直到达到系统设定的最大预读大小，其缺省值是 128KB。这个缺省值已经沿用至少五年了，在当前更快的硬盘和大容量内存面前，显得太过保守。比如西部数据公司近年推出的 WD Raptor 猛禽 10000RPM SATA 硬盘，在进行 128KB 随机读的时候，只能达到 16%的磁盘利用率（图 5）。所以如果您运行着 Linux 服务器或者桌面系统，不妨试着用如下命令把最大预读值提升到 1MB 看看，或许会有惊喜：

```
# blockdev  -setra 2048 /dev/sda
```

当然预读大小不是越大越好，在很多情况下，也需要同时考虑 I/O 延迟问题。

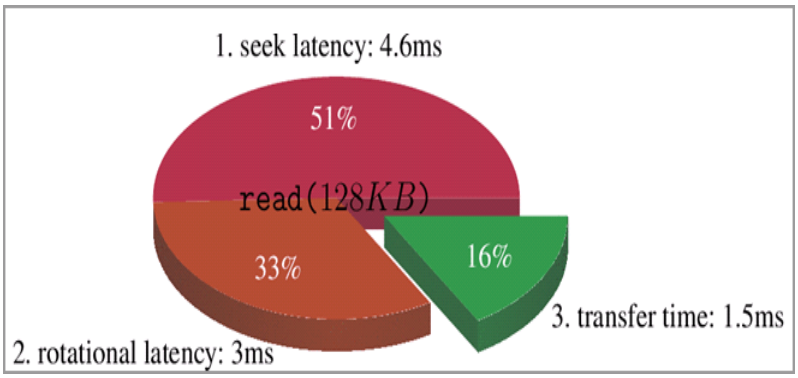

图 5 128KB I/O 的数据定位时间和传输时间比重

重新发现顺序读

上一节我们解决了是否 / 何时进行预读，以及读多少的基本问题。由于现实的复杂性，上述算法并不总能奏效，即使是对于顺序读的情况。例如最近发现的重试读(retried read)的问题。

重试读在异步 I/O 和非阻塞 I/O 中比较常见。它们允许内核中断一个读请求。这样一来，程序提交的后续读请求看起来会与前面被中断的读请求相重叠。如图 6 所示。

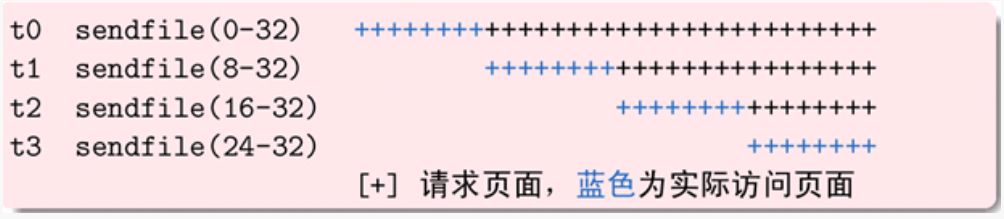

图 6 重试读(retried reads)

Linux 2.6.22 无法理解这种情况，于是把它误判为随机读。这里的问题在于“读请求”并不代表读取操作实实在在的发生了。预读的决策依据应为后者而非前者。最新发布的 2.6.23 对此作了改进。新的算法以当前读取的页面状态为主要决策依据，并为此新增了一

个页面标志位：PG_readahead，它是“请作异步预读”的一个提示。在每次进行新预读时，算法都会选择其中的一个新页面并标记之。预读规则相应的改为：

◆当读到缺失页面(missing page)，进行同步预读；

◆当读到预读页面(PG_readahead page)，进行异步预读。

这样一来，ahead 预读窗口就不需要了：它实际上是把预读大小和提前量两者作了不必要的绑定。新的标记机制允许我们灵活而精确地控制预读的提前量，这有助于将来引入对笔记本省电模式的支持。

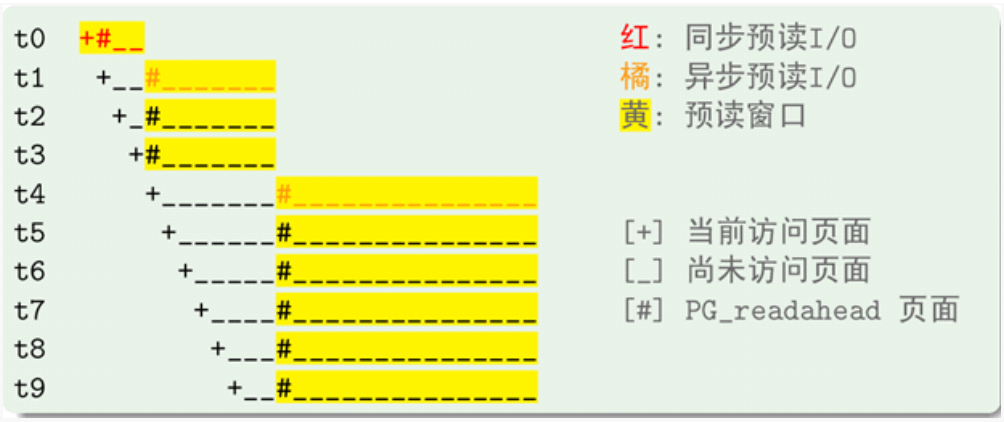

图 7 Linux 2.6.23 预读算法的工作动态

另一个越来越突出的问题来自于交织读(interleaved read)。这一读模式常见于多媒体/多线程应用。当在一个打开的文件中同时进行多个流(stream)的读取时，它们的读取请求会相互交织在一起，在内核看来好像是很多的随机读。更严重的是，目前的内核只能在一个打开的文件描述符中跟踪一个流的预读状态。因而即使内核对两个流进行预读，它们会相互覆盖和破坏对方的预读状态信息。对此，我们将在即将发布的 2.6.24 中作一定改进，利用页面和 pagecache 所提供的状态信息来支持多个流的交织读。

预读建议

预读建议(advice)	涵义	内核动作
POSIX_FADV_NORMAL	无特别建议	重置预读大小为默认值
POSIX_FADV_SEQUENTIAL	将要进行顺序操作	设最大预读大小为默认值的 2 倍
POSIX_FADV_RANDOM	将要进行随机操作	将最大预读大小清零（预读被禁止）
POSIX_FADV_NOREUSE	指定的数据将只访问一次	（无动作）
POSIX_FADV_WILLNEED	指定的数据即将被访问	立即预读数据到 pagecache
POSIX_FADV_DONTNEED	指定的数据近期不会被访问	立即从 pagecache 中丢弃数据

相关链接：

日期	版本号	作者	功能
2002-4	2.5.8	Andrew Morton	统一的预读架构；预读算法的雏形
2003-2	2.5.60	Andrew Morton	引入 posix_fadvise64()
2003-7	2.5.75	Linus Torvalds	独立的 mmap read-around 算法：更快更简单
2004-5	2.6.7	Andrew Morton	改进多线程并发读
2005-1	2.6.11	Steven Pratt, Ram Pai	引入读大小参数：代码简化及优化；支持随机读
2005-3	2.6.12	Oleg Nesterov	支持非对齐顺序读
2005-9	(2.6.13)	吴峰光	自适应预读算法 V1 (http://lwn.net/Articles/155510/)
2006-5	2.6.17-mm	吴峰光	自适应预读算法 V12 (http://kerneltrap.org/node/6642)
2007-5	2.6.22-mm	吴峰光	简化的自适应预读 (http://lwn.net/Articles/235164/)
2007-7	2.6.23	吴峰光	引入页面状态：代码简化及优化；改进重试读、预读抖动
2007-10	2.6.24	吴峰光	引入 pagecache 状态：改进多线程交织读