内存管理

在内核里分配内存可不像其他地方分配内存那么容易，造成这种局面的因素很多，从根本上讲，是因为内核本身不能像用户空间那样奢侈的使用内存，内核与用户空间不同，他不具备这种能力，它不支持简单便捷的分配方式

页

内核把物理页作为内存管理的基本单位，大多数32位体系结构支持4kb的页，而64位体系结构一般会支持8kb的页。这就意味着，在支持4kb页的大小并有1gb物理内存的机器上，物理内存会被划分为262144个页。

内核用struct page结构表示系统中每个物理页。

- flag域用来存放页的状态。这些状态包括页是不是脏的，是不是被锁定在内存中等
- _count域存放页的引用计数，也就是这一页被引用了多少次
- virtual域是页的虚拟地址，通常情况下，他就是页在虚拟内存的地址
- page结构与物理页相关，而并非与虚拟页相关

区

由于硬件的限制，内核并不能对所有的页都一视同仁。有些页位于内存特定的物理地址上。所以内核又把页划分为不同的区。

之所以这么做，是为了解决下面两个问题

- 一些硬件只能用于某些特定内存地址来执行DMA（直接内存访问）
- 一些体系结构的内存的物理寻址范围比虚拟寻址要大得多，这样就有一些内存不能永久的映射到内核空间上

因此，Linux使用了四种区，Linux把系统的页划分为区，形成不同的内存池，这样可以根据用途去进行分配了。注意：这种区的划分没有任何物理意义，只是内核为了管理页而采取的一种逻辑上的分组

- ZONE-DMA这个区用来进行直接内存访问
- ZONE-DMA32和上一个相似，但是这些页面只能被32位设备访问
- ZONE-NORMAL这个区包含的都是能正常映射的页
- ZONE-HIGHEM这个区包括告诉俺内存——其中的页并不能永久地映射到内核地址空间

表 12-1　x86-32 上的区

区	描　述	物理内存
ZONE_DMA	DMA 使用的页	<16MB
ZONE_NORMAL	正常可寻址的页	16 ～ 896MB
ZONE_HIGHMEM	动态映射的页	>896MB

某些分配可能需要从特定的区中获取页，而另外一些分配则可以从多个区获取页，不过不可能同时从两个区分配，因为分配是不能够跨越界限的，但是，如果可供分配的资源不够用了，那么内核就会去占用其他可用区的内存。

不是所有的体系结构都定义了全部去。比如64位系统的话可以处理64位的内存空间，所以对于64位操作系统来说，只有ZONE-DMA和ZONE-NORMAL区。

每个区都用对应的结构——struct-zone表示。关键的几个结构如下

- lock：这个域是一个自旋锁，他能防止结构被并发访问
- watemark：这是一个数组，简单来说，记录对应区的内存占用水位
- name域：名字

获得页

底层访问机制

内核提供了一种请求内存的底层机制，并且对他进行访问的几个接口

- alloc-page获得页的代表，他会分配2的x的方的连续物理页，x位传入的参数
- free-page：释放页：注意释放页的时候需要释放自己的页，不能传递错地址，否则内核会停止运行

kmalloc

kmalloc函数与用户空间的malloc一族函数非常类似，只不过他多了一个flags函数。它可以获得以字节位单位的一块内核内存

kmalloc还提供了分配器标志（alloc-page）也有，他们的作用是提示内核如何区分配内存包括但不限于

- 不能睡眠
- 从哪里分配等功能

kmalloc函数确保页在物理地址上是连续的，并且虚拟内存中也是连续的

vmalloc

vmallloc函数的工作方式类似于kmalloc，只不过前者分配的内存虚拟地址是连续的，而物理地址则无需连续，这也是用户空间分配函数的工作方式。

但是与kmalloc不同的是，vmalloc返回的内训虚拟地址是连续的，但是物理内存不是连续的。

slab

为了数据频繁的分配和回收，编程人员常常会用到空闲链表，空闲链表包含可供使用的，已经分配好的数据块。Linux的slab公司就是解决这个工作的

slab层把不同的对象划分为所谓的高速缓存组，每个高速缓存组都存放不同的类型的对象。kmalloc接口建立在slab层之上，使用了一组通用高速缓存

当内核某一部分需要一个新的对象的时候，先从部分满的slab中进行分配，然后没有部分满的slab，就从空的slab中进行分配。这种策略能减少碎片

每个高速缓存都使用kmem-cache结构表示，这个结构包含了三个链表。slab描述符要么在slab之外分配，要么放在slab自身开始的地方，如果slab很小，足够有空间容纳，那么就放里面。

slab分配器可以创建新的slab，这是通过底层的访问机制的分配器实现的

图示如下

对象之间的关系。

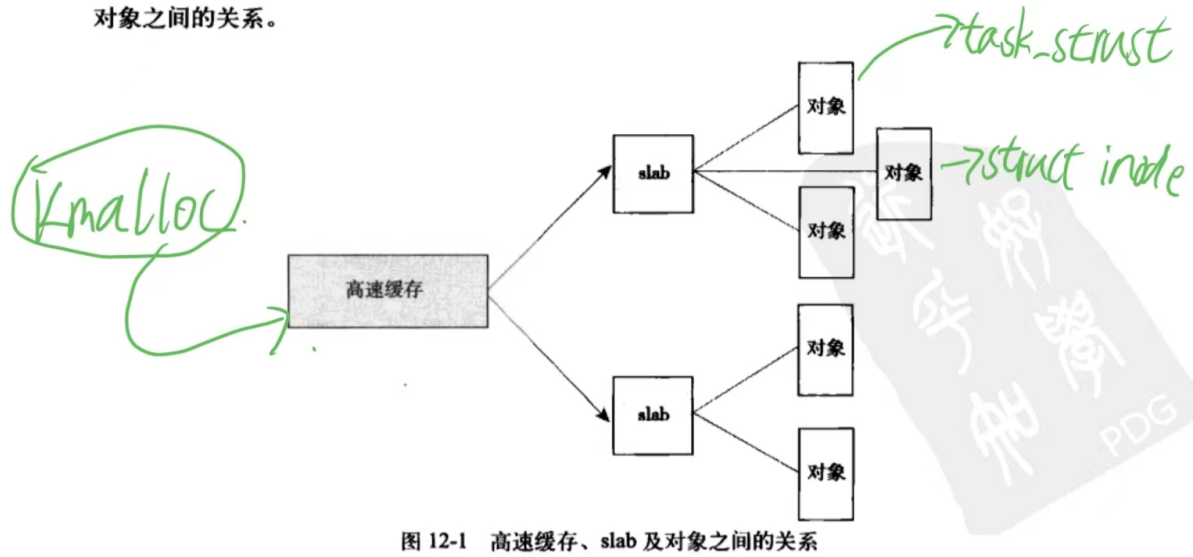

图 12-1　高速缓存、slab 及对象之间的关系

在栈上的静态分配

对于内核，内核栈小而且固定，当给每个进程分配一个固定大小的栈后，不但可以减少内存的消耗，而且内核也无需负担台中的栈管理任务

32位的栈位8kb，64的为16kb

对于中断程序，我们有专门的中断栈，中断栈为每一个进程提供一个可以用于中断处理程序的栈

在任意一个函数中，我们都必须要尽可能的节约栈资源，因为内核没有用户空间那样的保护，那么，就需要确保栈不会溢出。防止数据破坏等

每个CPU的分配

一般来说，每个CPU的数据存放在一个数组中，数组中的每一项对应着系统上一个存在的处理器。按当前处理器好确定这个数组的当前元素，这就是内核处理每个CPU数据的方式

所以Linux提供了接口可以分配每一个CPU，这个可以查

这么做的好处有

- 首先减少了数据锁定，因为在单一CPU上运行，就不需要啥锁
- 使用每个CPU数据可以大大减少缓存失效
- 注意任务是不能睡眠的

虚拟文件系统

虚拟文件系统作为内核子系统，为用户空间程序提供文件和文件系统相关的接口

之所以可以使用这种通用接口对所有类型的文件系统进行操作，是因为内核在它的底层文件系统接口上建立了一个抽象层，我们把这个文件系统叫做VFS

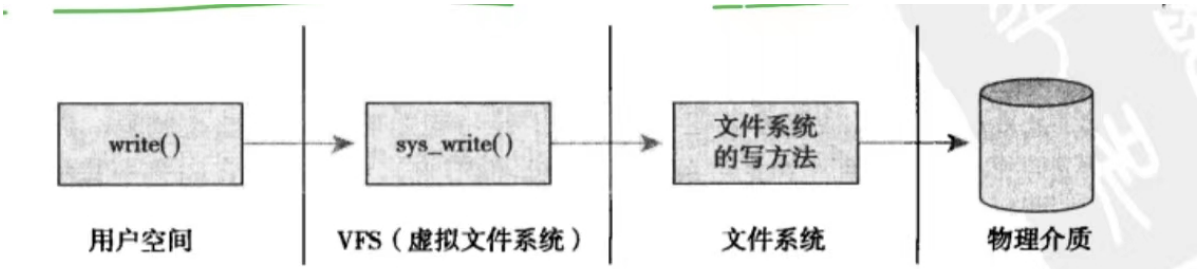

图 13-2 write() 调用将来自用户空间的数据流，首先通过 VFS 的通用系统调用，其次通过文件系统的特殊写法，最后写入物理介质中

unix文件系统

unix使用了四种和文件系统相关的传统抽象概念：文件，目录项，索引节点，安装点

从本质上讲文件系统是特殊的数据分层储存结构，它包含文件，目录和相关控制信息。在unix中，我呢见系统被安装在一个特定的安装点上，该安装点在全局结构层次中被称作命名空间，所有已安装的文件系统作为根文件数的枝叶出现在系统中

文件通过目录组织起来，在Linux中，目录属于普通文件。目录其中的每一项都叫做目录项。

Linux把文件本身和文件相关信息区分开了

- 文件相关信息：访问权限等储存在一个单独的数据结构中：inode

文件系统的控制信息储存在超级块中，Linux的VFS的设计目标是要能保持和实现这些概念的文件系统协同工作，像FAT和NTFS这样的就必须经过一些封装

VFS对象

VFS采用面向对象的设计思路，它是由C语言的结构体实现，包括

- 超级块对象：代表一个具体已经安装的文件系统
- 索引节点对象：代表一个具体的文件
- 目录项：他代表一个目录项，是路径组成的一部分
- 文件对象：它代表由进程打开的文件

超级块对象

各种文件系统都必须实现超级块对象，用于储存特定文件系统的信息，通常对应于存放在磁盘特定扇区中的文件系统超级块/文件系统控制块

索引节点对象

包含了内核在操作文件或目录时需要的全部信息，文件系统都必须从中提取这些信息。没有索引节点的文件系统通常将文件的描述作为文件的一部分来存放

索引节点对象必须在内存中创建，以便文件系统使用

目录项对象

VFS把目录当作文件对待，所以在路径/bin/vi中，bin和vi都属于文件，但是他们也属于目录项对象，在路径中，每一部分都是目录项对象

目录项由三种有效状态：被使用，未被使用，负状态

目录项缓存

为了方便的查找所有的目录项，我们就有了目录项缓存，包括三个部分

- 被使用的目录项链表
- 最近被使用的双向链表：该链表含有未被使用的和负状态的目录项对象
- 散列表和相应的散列函数用来快速的将给定路径解析为相关目录项对象

VFS会先在目录项缓存中搜索路径名，如果找到了就没必要花费力气去找了，如果找不到再去遍历

和进程相关的数据结构

系统中的每一个进程都有自己一组打开的文件，像根文件系统，当前工作目录，安装点等，有三个数据结构将VFS层和系统的进程紧密联系在一起，他们分别是：file-struct，fs-struct和namespace结构体，其中

- fd-array数组指针指向已打开的文件对象
- fs-struct包含了文件系统和进程相关的信息，该结构也包含了当前进程的当前工作目录（pwd）和根目录
- namespace结构体能看到文件系统层次结构

块IO层

块IO层简介

系统中能够随机访问固定大小的数据片的硬件设备称作块设备。这些固定大小的数据片就称作块，最常见的块设备就是硬盘。除此以外，还有软盘驱动器，蓝光光驱等其他设备

另外一种设备是字符设备，字符设备能够按照字符流的方式被有序访问，像串口和键盘就属于字符设备。如果一个硬件设备是以字符流的方式被访问的话，那就应该将他归于字符设备。反过来，如果一个设备是随机无须访问的，那么他就属于块设备

内核管理块设备要比管理字符设备复杂的多，因为字符设备只需要控制一个位置：当前位置，而块设备需要在介质之间不断的移动。需要有一个专门的子系统

块设备中最小的单元是扇区，扇区大小一般是2的整数倍

缓冲区和缓冲区头

当一个块被调入内存时候，它要储存在一个缓冲区中，每一个缓冲区与一个块对应，块的大小不能够超过一个页面，因为内核在处理数据的时候需要一些相关的控制信息，所以每一个缓冲区都有一个对应的描述符

bio结构体

目前内核中块IO的操作基本容器由bio结构体表示，该结构体代表了正在现场以片断链表形式组织的块IO操作，一个片段是一小块连续的内存缓冲区，其中一些重要的东西

- bi-io-vec域指向一个bio-vec结构体数组，该结构体链表包含一个特定IO操作所需要的使用到的所有片段
- 每一个bio-vec都是一个形式为<page,offset,len>的vector

总而言之，每一个块IO请求都通过一个bio结构体表示，每个请求都包含一个或者多个块，这些块储存在bio-vec结构体数组中，这些结构体描述了每个片段在物理页中的实际位置，并且像vector一样组成在一起

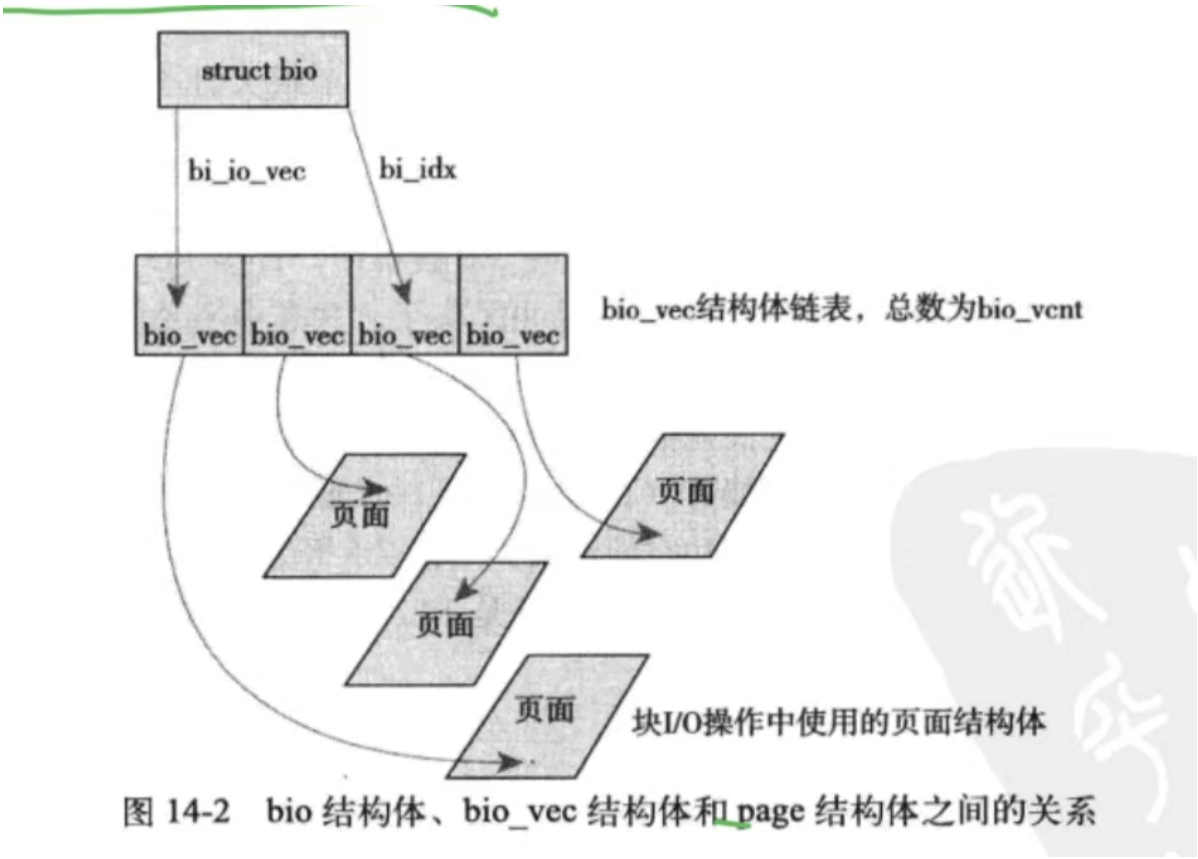

图 14-2　bio 结构体、bio_vec 结构体和 page 结构体之间的关系

bio结构体有下面几个好处

- bio结构体很容易处理高端内存，因为它处理的是物理页面而不是直接指针
- bio结构体既可以代表普通页IO，同时也可以代表直接IO
- bio结构体便于执行分散-集中的块IO操作，操作中数据可取自多个物理页面
- bio结构体相比缓冲区头属于轻量级结构体

请求队列

块设备将他们挂起的块IO请求保存在请求队列中，该队列由reques-queue结构体表示，包含一个双向请求链表以及相关控制信息

因为一个请求可能需要操作多个连续的磁盘块，所以每个请求都由bio结构体组成

IO调度程序

如果简单地以内核产生的请求的次序直接将请求发给块设备的话，性能肯定让人难以接受，

为了优化寻址操作，内核既不会简单的请求接收次序，也不会立即将其交给磁盘。他会提前进行合并操作。

IO调度程序就是用来管理请求队列的，其中比较知名的调度有下面几个

- linus电梯调度
    - Linus电梯能执行合并与排序预处理。当有新的请求加入队列时候，他会先检查每一个挂起的请求是否可以合并
    - 如果队列中已经存在一个对相邻磁盘扇区操作的请求，那么新请求将和这个已经存在的请求合并为一个请求
    - 如果队列中存在一个驻留时间过长的请求，那么新的请求就会被插入队列尾部，以往其他旧的请求发生饥饿
    - 如果队列中以扇区方向为序存在合适的插入位置，那么新的请求将被插入到该位置，保证队列中的请求是以访问磁盘物理位置为序进行排列的
- 最终期限IO调度程序
    - Linus电梯存在一些问题，比如
        - 假设我在某个磁盘区域上进行繁重的操作，无疑会使得磁盘其他位置上的操作请求永远得不到机会，这是一种饥饿情况
        - 此外，在进行读写操作的时候，虽然写请求对应用程序性能带来的影响不大，但是应用程序必须等待读请求结束后才能运行其他程序，所以读操作响应时间对系统的性能非常重要
        - 但是读请求会相互依靠，这就导致可能还是有饥饿的情况
    - 所以有了最后期限IO调度
        - 它类似于Linus电梯，但是每个请求都有一个最后期限，它维护三个队列，读，写，排序FIFO队列
        - 如果在写FIFO或者读FIFO队列请求超时的话，那么最后期限调度程序就会从FIFO队列中提取请求服务，将它推入队列
        - 虽然这样，但是这个最后期限FIFO调度算法也不能严格保证请求响应的时间，但是通常情况下可以京可能避免
- 预测IO调度程序
    - 和最终期限IO调度程序一样，但是预测IO会观察现在IO的操作习惯，这种预测靠启发来工作
    - 随着IO次数的增多，预测IO就会记录这个应用的IO行为，并且预测未来行为。
- 完全公平排队IO调度
    - 顾名思义，但是这种它每一个进程都有自己的IO队列
    - 他以时间片轮转进行，每个队列选取请求数（一般为4）然后一次轮转
- 空操作IO调度
    - 空IO基本不排序，他只是执行合并
    - 类似于malloc一样，只是检查上下并且尝试核并

进程地址空间

地址空间

进程地址空间由进程可寻址的虚拟内存组成，而且内核允许进程使用这种虚拟内存的地址，现在操作系统采用的都是平坦地址空间

如果一个进程的地址空间和另外一个进程一样但是他们不相干，那么就会被称为线程

我们把虚拟地址中可以合法访问的趋于叫做内存区域。进程可以给自己的地址空间动态增加和减少内存区域

内存区域可以包含各种内存对象，比如

- 代码段：可执行文件代码的内存映射
- 数据段：可执行文件已初始化的全局变量的内存映射
- 内存映射：包含未初始化的全局变量，也就是bss段的零页
- 用于用户空间的进程的栈的零页内存映射
- 地址空间：动态连接程序用的
- 其他内存映射文件
- 其他共享的内存段
- malloc分配的内存

内存描述符

内核使用内存描述符结构体（mm-struct）表示进程的地址空间，该结构包含了和进程地址空间有关的全部信息，有几个关键的域

- mmusers域记录了正在使用的进程数目，比如说两个进程共享的话那就是2
- mmap和mmrb这两个不同的数据结构体描述对象是相同的，前者是以链表的形式存放，后者是红黑树的形式存放
    - mmap结构体可以简单高效的遍历元素，而mmrb适合用于查找元素

mm-struct与内核进程

内核进程没有进程地址空间，所以也没有相关的内存描述符，所以内核进程的进程描述符中mm域为空。这也是内核线程的含义：他们没有用户上下文

但是如果访问内核内存，那么内核线程也还是需要一些数据，比如页表，如果使用的时候内核会直接用进程的描述符

当一个进程被调度的时候，该进程的mm域指向的地址空间被装在到内存，然后进程描述符中的active-mm会更新，指向新的地址空间

对于内核来说，因为不直接访问用户空间的内存，所以内核线程可以直接使用前一个进程的页表

虚拟内存区域

内存区域由vm-area-struct结构体描述，它指定了地址空间内连续区间的一个独立内存范围，内核将每个内存区域作为一个单独的内存对象管理，每个内存区域都拥有一致的属性，比如访问权限等。另外，相应的操作也是一致的

每个内存描述符都对应于进程地址空间中唯一的区间

其中有一种很重要的标志，VMA

VMA标志

VMA标志是一种特殊的标志，它包含在vm-flags域内，标志了内存区域所包含的页面的行为和信息，和物理页的访问权限不同，VMA标志反应了内核处理页面所需要遵守的行为准则，而不是硬件要求

其中包括可读写等

创建内存区域

内核使用do-mmap()函数创建一个新的线性地址空间

使用它后会在虚拟内存中分配一个合适的内存区域。

页表

当一个程序访问虚拟地址的时候，需要先将虚拟地址转化为物理地址，所以我们需要进行页表查询

Linux使用三级页表完成地址转换

- 顶级页表是页全局目录（PGD），指向页二级：PMD
- 二级页表是中间项目录，指向页三级（PTE）
- 最后一个就是页表

查询图片如下

所以，当请求访问一个虚拟地址的时候，就需要使用一个缓冲器（TLB）。每当请求一个虚拟地址的时候，处理器首先检查TLB中是否缓存了该虚拟地址到物理地址的映射，如果缓存有那就命中，没有就返回

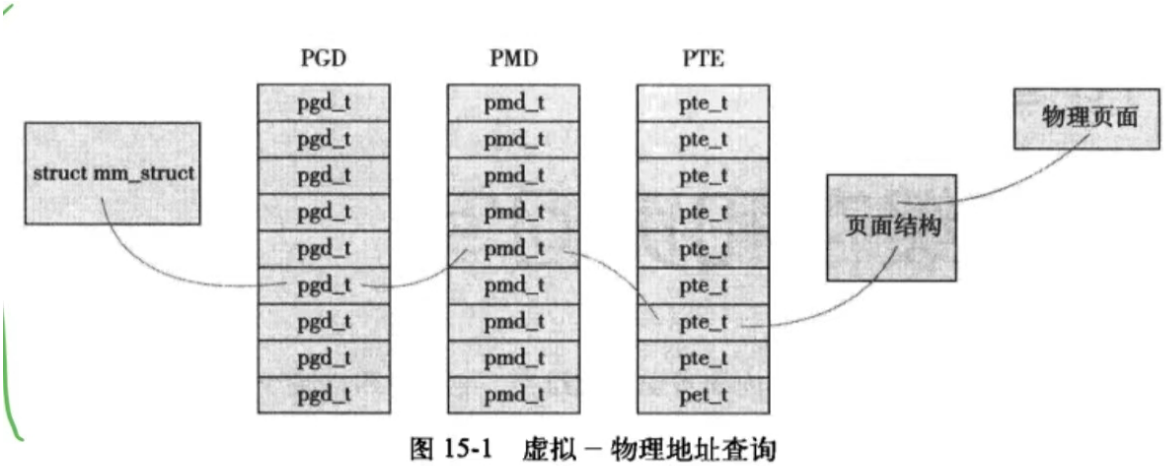

图 15-1　虚拟－物理地址查询

页高速缓存和页回写

页高速缓存是Linux内核实现磁盘缓存，主要是缓解访问速度不平衡，减少IO操作的

写缓存

Linux采用的写回的方法

- 在这种策略下，程序执行写操作直接写到缓存中，后端储存不会直接更新。而是标记称为脏的，随后加入脏页链表中
- 之后再进行一次性的回写

缓存回收的时候采用的策略

最好的算法就是LRU算法了，这种算法是比较好的相对理想的

Linux实现了一个双链LRU，类似于数据库的。它维护两个链表：old和young链表

- 在young链表上的被认为是热，他们不会被换出
- 在old链表上被认为是冷的，他们会被换出
- 他们以一种伪LRU原则：实际上是FIFO，两者都是从尾部进入头部出来
- 当活跃链表过多的时候，那么链表头就要去非活跃的，反之亦然

address-space对象

在页高速缓存中可能包括多个不连续的物理磁盘块，也正是由于页面中映射的磁盘不一定连续，所以在页高速缓存中检查数据不能用特定信息

Linux采用了一个新的对象管理缓存项和页面IO操作，这个对象是address-space结构体

方式

- 首先在页高速缓存中搜索需要的页，如果需要的页不在高速缓存中，那么内核在高速缓存中新分配一个空闲项，然后进行写请求最后写入磁盘

什么时候写回?

- 当空闲内存低于一个特定的阈值的时候，内核必须将脏页写回磁盘以便释放内存。
- 当脏页在内存中驻留时间超过一个特定的阈值的时候，会将其写入磁盘
- 使用sync和fsync时候，立刻写回