[**Linux内核剖析之回收页框**](http://blog.csdn.net/zhaobryant/article/details/40859051)

**一、页框回收算法**

**1、为何要有页框回收算法？**

    Linux在为用户态与内核分配动态内存时，检查得并不严谨。

    例如：

    (1)、对单个用户创建的进程的RAM使用的总量并不作严格的检查（进程资源的限制只针对单个进程）；

    (2)、对内核使用的许多磁盘高速缓存和内存高速缓存大小也同样不做限制。

**2、为何要减少控制？**

    可以使内核以最好的可行方式使用可用的RAM：

    (1)、当系统负载较低时，RAM的大部分由磁盘高速缓存占用，较少的正在运行的进程可以获益；

    (2)、当系统负载增加时，RAM的大部分则由进程页占用，高速缓存就要缩小给进程让出空间。

    进程和高速缓存会持续的获得页框，但无法从正在运行的进程中获得已经不再使用的页框。

    Linux内核调用**页框回收算法（PFRA）**采取从用户态进程和内核高速缓存窃取页框的办法补充伙伴系统的空闲块列表。

    用完所有的空闲内存之前，就必须执行页框回收算法：

    ===>>>要释放一个页框，内核要把页框的数据写入磁盘，但是，为了完成这个操作，内核要请求另一个页框。

**3、选择目标页**

    PFRA按照页框所含的内容以不同的方式处理页框。

**不可回收页**---->不允许也无需回收

    空闲页（包含在伙伴系统列表之中）

    保留页（PG\_reserved标志置位）

    内核动态分配页

    进程内核态堆栈页

    临时锁定页

    内存锁定页

**可交换页**---->将页的内容保存在交换区

    用户态地址的匿名页

    tmpfs文件系统的映射页（IPC共享内存的页）

**可同步页**---->必要时，与磁盘映像同步这些页

    用户态地址空间的映射页

    存有磁盘文件数据且在页高速缓存中的页

    块设备缓冲区页

    某些磁盘高速缓存的页

**可丢弃页**---->无需操作

    内存高速缓存中的未使用页（如slab分配器高速缓存）

    目录项高速缓存的未使用页

**解释：**

    映射页：页映射了一个文件的某个部分。

        例如：属于文件内存映射的用户态地址空间中所有的页都是映射页；

                    映射页差不多都是可同步的：为了回收页框，内核必须检查页是否为脏，必要时将页的内容写到相应的磁盘文件中。

    匿名页：属于某个进程的匿名线性区。

        例如：进程的用户态堆和堆栈中的所有页为匿名页。

                为了回收页框，内核必须把页的内容保存在一个专门的磁盘分区或磁盘文件----交换区。

                所有的匿名页都是可交换的。

**4、特殊文件系统：**

    内核产生但不存在于硬盘上（存在于内存中）的文件系统。

    特殊文件系统并不管理磁盘空间（无论是磁盘的还是在网络上的），它们在UNIX操作系统上大量使用。这些文件系统通常由系统内核或者应用程序动态管理，以达到反映系统运行状况、进行进程间通信、获取临时文件空间等目的。常见的这类特殊文件系统有：proc文件系统、tmpfs文件系统、devfs文件系统、rootfs文件系统等。

    proc文件系统：

        存储的是当前内核运行状态的一系列特殊的文件；

        可以通过这些文件查看有关系统硬件及当前正在运行进程的信息；

        也可以更改其中文件来改变内核的运行状态。

    tmpfs文件系统：

        标准的挂载点是/dev/shm，默认是实际物理内存的一半；

        tmpfs可以使用mem（物理内存）或者swap（交换分区）（free -m查看）。

      几大特性：

            tmpfs构建在内存中，所以存放在tmpfs的数据在卸载或者断电后就消失；

            快速读写能力，内存的访问速度要远快于磁盘的IO操作；

            动态收缩特性，给它分配20M的空间，并不是一开始就占用20M，而是一开始使用很少的空间。随着文件的增加，tmpfs会分配更多的内存。

    rootfs文件系统：

        根文件系统

    devfs文件系统：

        设备文件系统

    特殊文件系统中的页是不可回收的。tmpfs中的除外，它可以被保存在交换分区中后被回收。

**5、PFRA（页框回收算法）设计**

    选择合适的目标页是内核设计中最精巧的问题。

        --->保证内存需求有限但对系统响应有苛刻要求的计算机性能可接受；

        --->保证对内存有巨大需求的计算机性能可接受。

**PFRA设计的总体原则**

        (1)、首先释放无害页

            进程用户地址空间的页回收之前，先回收没有被任何进程使用的磁盘与内存高速缓存中的页；

            回收内存与高速缓存的页框不需要修改任何页表项。

        (2)、将用户态进程的所有页设定为可回收页

            除了锁定页，PFRA必须能够窃取任何用户态的进程页，包括匿名页；

            睡眠较长的进程将失去所有页框（或者说一个进程的部分线性区长时间没有被使用，页框将被回收）。

        (3)、同时取消引用一个共享页框的所有页表项的映射，就可以回收该共享页框

            当PFRA要释放几个进程共享的页框时，它就清空引用该页框的所有页表项，然后回收该页框。

        (4)、只回收“未用”页

            使用简化的最近最少使用（LRU）置换算法，PFRA将页分为“在用”和“未用”。如果某页很长时间没有被访问，那么它将来被访问的可能性就比较小，就可以将他看做未用；另一方面，如果某页最近被访问过，那么它将来被访问的可能性较大，就必须把它看做在用。PFRA只回收未用页。

        LRU算法：

            用一个计数器来存放RAM中每一页的页年龄，即上一次访问该页到现在已经过的时间。

            此计数器可以使PFRA只回收任何进程的最旧页。

            80x86处理器不提供这样的硬件功能，Linux内核不能依赖计数器记录页年龄。

            Linux每个页表都一个访问标志位；页年龄由页描述符在链表中的位置来表示(后面会提到)。

        集中启发式方法的混合：

            谨慎选择检查高速缓存的顺序；

            基于页年龄的变化顺序（释放最近最少使用的页）；

            区别对待不同状态的页。

**二、反向映射**

    释放共享页框：Linux内核必须能够快速定位指向同一页框的所有页表项，此过程叫做反向映射。

    最简单的方法：

        在页描述符中引入附加字段，将某页描述符确定的页框所对应的多个页表项连接起来。

    面向对象的反向映射：

        用户态页---->线性区---->内存描述符---->页全局目录。

        线性区描述符比页描述符少的多。

   如何实现？

        PFRA需要确定页是共享的还是非共享的，映射页或是匿名页。

            页描述符中存在两个字段：

                \_mapcount：引用页框的页表数目。

                mapping：用于确定页是匿名的还是映射的。

                    若为null：该页面属于交换高速缓存；

                    最后一位是1：匿名页，mapping指向anon\_vma描述符指针；

                    最后一位是0：映射页，mapping指向对应文件的address\_space对象。

**1、匿名页反向映射===>>>**

    将引用同一个页框的所有匿名页连接起来的策略：

    将该页框所在的线性区存放在一个双向循环链表中（即使一个匿名线性区存有不同的页，也始终只有一个反向映射链表用于该区域的所有页框）。

    当匿名线性区分配一个页（物理页）的时候，内核会创建一个anon\_vma结构，其head字段是线性区的双向链表的表头。

    内核将此匿名线性区vm\_area\_struct描述符插入anon\_vma中。

    为了实现这个目的，vm\_area\_struct本身含有两个字段：

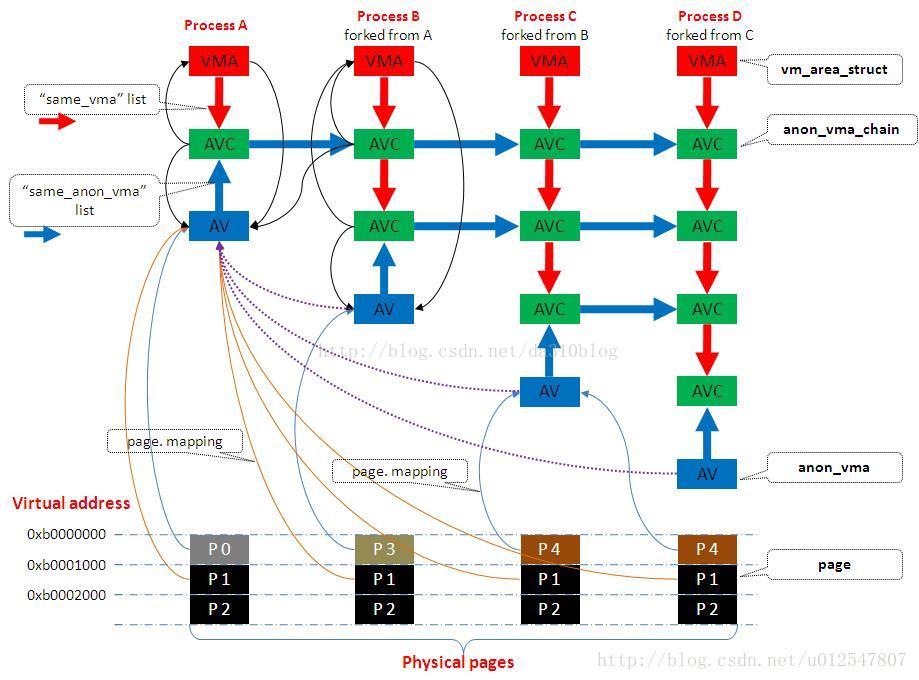
       anon\_vm\_node：存放指针：前驱和后继

       anon\_vma：指向anon\_vma链表头

    最后，将anon\_vma指向page页描述符的mapping字段。

    当已经被一个进程引用的页框插入到另一个页表项时（fork()系统调用），内核只是将子进程中共享了同一个页框的线性区的描述符插入page->anon\_vma双向链表中；一个AVC通常包括了不同进程的线性区。

（图中，VMA表示虚拟内存区域，AVC表示anon\_vma\_chain，AV表示anon\_vma）



    相关函数：

    try\_to\_unmap\_anon(page);

        回收匿名页框，PFRA需要扫描anon\_vma的所有线性区。

        函数执行步骤：

        1、获得anon\_vma的自旋锁，通过page->anon\_vma获得；

        2、扫描线性区的链表，对每一个线性区，用try\_to\_unmap\_one(vma, page)函数：

        ==>>try\_to\_unmap\_anon()函数<匿名页>和try\_to\_unmap\_file()函数<映射页>重复调用！

        try\_to\_unmap\_one(vma, page);

            (1)、计算出回收页的线性地址，根据以下参数：

              vma->vm\_start：线性区的起始线性地址

             vma->vm\_pgoff：被映射文件的线性区偏移量

             page->index：页磁盘映像或匿名区中标识存放在页框中数据的位置

                对于匿名页：

             vma->vm\_pgoff为0

             page->index是区域内的页索引

          (2)、如果目标页时匿名页，则检查页的线性地址是否在线性区内。如果不是，则返回；

          (3)、通过vma->vm\_mm得到内存描述符地址，获得保护页表的自旋锁；

          (4)、调用pgb\_offset()、pud\_oddset()……获得对应目标页线性地址的页表项地址；

          (5)、执行一些检查来验证目标页可有效回收；

          (6)、页可以被回收，清空页表项；

          (7)、递减页描述符的\_mapcount字段，递减也描述符\_count字段中的页框使用计数器；

          (8)、调用pte\_unmap()释放内核映射。

**2、映射页的反向映射**

    共享匿名页框的数量不是很大，线性区组织在双向链表中。

    映射页是经常共享的。这里的映射页不单单指文件，不同进程会经常共享同一个程序代码（几乎所有的进程都会包括标准C库代码的页）。

    线性区使用优先搜索树组织。映射页使用优先搜索树迅速查找共享通一个页框的的所有线性区。

    相关函数：

    try\_to\_unmap\_file();

    函数执行步骤：

    1、获得page->mapping->i\_mmap\_lock自旋锁

    2、对搜索树应用vma\_prio\_tree\_foreach()宏，搜索树的跟在page->mapping->i\_mmap字段

    3、对宏发现的每个vm\_area\_struct描述符，使用try\_to\_unmap\_one()，尝试对该页所在的线性区页表清0

    ......

**三、PFRA（页框回收算法）实现**

    PFRA有几个入口，页框回收算法的执行有三个基本情形：

    1、内存紧缺回收：内核发现内存紧缺。

         激活情形：

         grow\_buffers()函数无法获得新的缓冲区页；

         alloc\_page\_buffers()函数无法获得页临时缓冲区首部；

         \_\_alloc\_pages()函数无法在给定的内存管理区中分配一组连续的页框。

    2、睡眠回收（不做进一步讨论）。

    3、周期回收：周期性的激活内核线程执行内存回收算法。

         由下面两种不同的内核线程激活：

             kswapd内核线程，检查某个内存管理区中空闲的页框数是否低于pages\_high值。

             events内核线程，预定义工作队列的工作者线程。

           PFRA周期性的调度用于回收slab分配器处理的位于内存高速缓存中的所有空闲的slab。

1、**最近最少使用的（LRU）链表**

    LRU链表：活动链表与非活动链表。

    页的活动链表和非活动链表是页框回收算法的核心数据结构。

    页必须从非活动链表中窃取。

    数据结构的组织：

    这两个链表的表头存放在zone->active\_list和inactive\_list字段，

    zone->nr\_active和zone->inactive字段表示存放在两个链表中的页数。

    zone：内存节点描述符。

    每个内存节点描述符管理”许多“页框”；

    每个内存节点根据页框的分布划分为三个内存管理区：16M|896M|大于896M。

    如果页属于LRU链表，则设置page（页描述符）的PG\_lru标识。

        相应的，如果页属于活动链表，设置PG\_active标识；如果页属于非活动链表，清除PG\_active标识。

    页描述符page中的lru字段负责指向LRU链表中的下一个元素和前一个元素的指针。

    常用函数：

    add\_page\_to\_active\_list()

        将页加入管理区的活动链表头部并递增管理区描述符的nr\_active字段

    add\_page\_to\_inactive\_list()

        将页加入管理区的非活动链表头部并递增管理区描述符的nr\_inactive字段

    del\_page\_from\_active\_list()

        从管理区的活动链表中删除页并递减管理区描述符的nr\_active字段

    del\_page\_from\_inactive\_list()

        从管理区的非活动链表中删除页并递减管理区描述符的nr\_inactive字段

    del\_page\_from\_lru()

        检查页的PG\_active标识。根据检查结果，将页从活动或非活动链表中删除，递减管理区描述符的nr\_active或者nr\_inactive字段，且如有必要，将PG\_active清0

    activate\_page()

        检查PG\_active标识，如果没置位，将页移到活动链表中

    lru\_cache\_add()

        如果页不在LRU链表中，将PG\_lru标志置位，得到管理区的lru\_lock自旋锁，调用add\_page\_to\_inactive\_list()把页插入管理区的非活动链表

    lru\_cache\_add\_active()

        如果页不在LRU链表中，将PG\_lru和PG\_active标志置位，得到管理区的lru\_lock自旋锁，调用add\_page\_to\_active\_list()把页插入管理区的活动链表

    在LRU链表之间移动页时，PFRA将最近访问过的页集中放在活动链表上，将很长时间没有访问过的页集中放在非活动链表上。但是，两个状态不足以描述页的所有情况，例如：

        每隔一小时写一次数据到一个页中，尽管一小时没有访问，但也不能回收。

    对于此问题，没有通用的解决方法，因为PFRA不能预测用户的行为。

    页描述符中的PG\_referenced标志：

        用来把一个页从非活动链表移到活动链表所需的访问次数加倍，也把一个页从活动链表移动到非活动链表所需的“丢失访问”次数加倍。

    例如：

        如果非活动链表的PG\_referenced为0，第一次访问把这个标志置为1，但这一页仍然留在非活动链表；第二次访问这个标志为1时，才移动到活动链表。

        偶尔一次的访问并不能说明这个页是“活动的”。

    相关函数：

    mark\_page\_accessed();

        内核调用此函数把一个页标记为访问过了。

        在如下情况会调用此函数，例如：

            按需装入进程的一个匿名页；

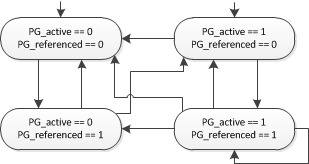
            按需装入内存映射文件的一个页；

            从文件读取数据页；

            换入一个页；

            在页的高速缓存中搜索一个缓冲区页。

        函数内部会检查PG\_referenced标志位，若置位，会调用activate\_page()。



        这个函数只是四种情况会调用：

            非活动链表访问标志为0，非活动链表访问标志位1，活动链表访问标志位1，活动链表访问标志位0

    page\_referenced();

        PFRA扫描页调用。

        返回值：如果PG\_referenced标志或者页表项中的某些Accessed标志位置位，返回1.

        函数功能：

        如果PG\_referenced标志置位则清0，对引用该页的所有用户态页表项中的Accessed标志位进行检查并清0。

        该函数只是负责清0。从活动链表到非活动链表不使用这个函数。

    refill\_inactive\_zone(zone, sc);

        //zone：指向内存管理区描述符（活动链表和非活动链表每个zone有一个）。

        //sc(scan\_control结构)：结构存放着回收操作执行时的有关信息。

        被shrink\_zone()调用（函数对页高速缓存和用户态地址空间进行页回收）。

        从活动链表到非活动链表移动页。

        函数的工作至关重要：

            若函数的掠夺性过强，会有过多的页从活动链表被移动到非活动链表，页框被大量回收，系统性能受到影响。

          若函数太懒惰，没有足够的页来补充非活动链表，PFRA不能回收足够的页。

        为此函数可以调整自己的行为：

            开始时，对每次调用，扫描非活动链表中少量的页；当PFRA很难回收内存时，该函数在每次调用时会逐渐增加扫描的活动页数（scan结构中有一个优先级字段）。

**试探法：**

            确定函数是移动所有的页还是只移动不属于用户态地址空间的页。

            计算管理区内的交换倾向值：

                交换倾向值 = 映射比率/2 + 负荷值 + 交换值

                映射比率：

                LRU链表中有两类页：一类是属于用户态地址空间的页，另一类是不属于任何用户态进程且在页高速缓存中的页。

                映射比率指的是用户态地址空间所有内存管理区的页占所有可分配叶框数的百分比。

                负荷值：

                PFRA在管理区中回收页框的效率，依据是前一次PFRA运行时管理区扫描的优先级：

                     12...7   6   5   4    3     2    1     0

                        0      1   3   6    12   25  50  100

                交换值是一个用户定义的常数，通常是60。

              可见PFRA会优先回收页高速缓存中的页，避免对进程产生影响。当管理区交换值大于等于100时，页才从进程地址空间处回收。

        函数执行步骤：

        1、对zone->active\_list中的页进行首次扫描（根据sc进行扫描），从链表的底部开始向上。

            为什么是底部？底部页是active\_list中距离上一次访问时间最长的页，近期不大可能被访问。

            直到链表为空或sc->nr\_to\_scan的页扫描完毕（扫描到的每一页要增加引用计数器）。

            在zone->active\_list中删除页描述符，加入临时局部链表l\_hold中。

            zone->nr\_active中减去移入l\_hold中的页数。

        2、计算交换倾向值

        3、l\_hold中的页根据交换倾向值分配到l\_active和l\_inactive中。

              属于某个进程用户态地址空间的页（page->\_mapcount为非负数）被加入l\_active的条件：

                交换值小于100或者应用于page\_referenced()函数返回正数（表明页最近被访问过），否则加入l\_inactive链表。

        4、对l\_inactive链表执行循环，加入到zone->inactive\_list，更新zone->nr\_inactive字段。

        5、对l\_active链表执行循环，加入zone->active\_list，更新zone->nr\_active字段，递减页框计数器。

**2、内存紧缺回收**

    当内存分配失败的时候会激活内存紧缺回收。

    例如：

        从伙伴系统分配一个或多个页框的时候，调用try\_to\_free\_pages()。

**函数try\_to\_free\_pages();**

    函数分析：

        参数：

        zones：要回收的页所在的内存管理区的链表。

        gfp\_mask：用于失败的内存的一组分配标志。

        order：未用。

        函数目标：

            通过重复调用shrink\_caches()和shrink\_slab()函数释放至少32个页框；

            每次调用后优先级会比前一次提高：从12到0；

            如果这个函数没能在某次（最多13次）调用shrink\_caches()和shrink\_slab()回收到至少32个页框，那么PFRA就没有办法了，只能杀死进程来释放页框 。

        函数执行步骤：

            1、分配和初始化一个scan\_control描述符；

            2、对zones链表中的每个管理区，将管理区描述符的优先级字段初始化为12，计算LRU链表的总页数；

            3、从优先级12到0，执行最多13次循环；

                a、更新scan\_control描述符的一些字段：把用户态进程的总页数存入nr\_mapped字段，把本次迭代的优先级存入priority字段；

                b、调用shrink\_caches()，传给它zones链表和scan\_control描述符地址作为参数，扫描管理区的非活动页；

                c、调用shrink\_slab()从可压缩内核高速缓存中回收页；

                d、回收页存储在scan\_control描述符的nr\_reclaimed字段；

                e、如果已达到目标，跳出循环（nr\_reclaimed大于等于32）；

                f、未达目标，做相应处理，这里略去。。。

            4、把每个管理区描述符的prev\_priority字段设为上一次调用的shrink\_caches()使用的优先级；

**函数shrink\_caches(zones，(scan\_control)sc);**

    函数分析：

        针对zones中的每一个zone用shrink\_zone()函数；

        调用shrink\_zone()之前，用sc->priority字段更新管理区描述符的优先级字段，即当前优先级；

        管理区描述符中的all\_unreclaimable标志置位，且当前优先级小于12，则shrink\_zone()不必调用；

        即：如果能够提前知道这个zone的页都是不可回收页，则没有必要扫描。

    函数shrink\_zone(zone, sc);

**函数分析：**

        函数的目标是从管理区非活动链表回收32页；

        每次在更大的一段管理区非活动链表上重复调用辅助函数shrink\_cache()，达到目标；

        shrink\_zone()会重复调用refill\_inactive\_zone()函数补充非活动链表。

    执行步骤：

        递增zone->nr\_scan\_active，增量取决于当前优先级；

        递增zone->nr\_scan\_inactive，增量取决于当前优先级；

        zone->nr\_scan\_active字段大于等于32，就把该字段赋值给：nr\_active，并把该字段设为0，否则nr\_active=0；

        zone->nr\_scan\_inactive字段大于等于32，就把该字段赋值给：nr\_inactive，并把该字段设为0，否则nr\_inactive=0；

        若nr\_active和nr\_inactive都为0，函数结束；

        若nr\_active大于等于32，则补充管理区非活动链表；

        若nr\_inactive大于等于32，则尝试从非活动链表回收32页（调用shrink\_cache()函数）；

        若成功回收32页，结束，都则跳回第六步。

    特点：为提高效率，函数采用批量操作，每32页一批   。

**函数shrink\_cache();**

    函数分析：

        从管理区非活动链表取出一组页，把它们放入一个临时链表，然后调用shrink\_list()函数 对这个链表中的每一页进行页框回收。

    函数执行步骤：

        1、处理非活动链表中的页（最多32项），对于每一页，递增引用计数器，把页从管理区活动链表移动到一个局部链表；

        2、更新zone->nr\_inactive；

        3、对搜集的页使用shrink\_list()函数；

        4、将shrink\_list()中没有成功释放的页放回到非活动链表中，如果没有成功回收目标页数，返回1。

**函数shrink\_list();**

    函数分析：

        页框回收的核心部分；

        前面的函数的目的是找到一组合适回收的候选页，shrink\_list()则是从page\_list链表中尝试回收这些页，当函数返回时，page\_list链表中剩下的是无法回收的页。

    函数执行步骤：

        1、执行循环，处理page\_list链表中的每一页。对其中每个元素，从链表上删除页描述符并尝试回收该页框。如果页框不能释放，则把该页框插入到一个局部链表中；

        2、函数把页描述符从局部链表移回page\_list。

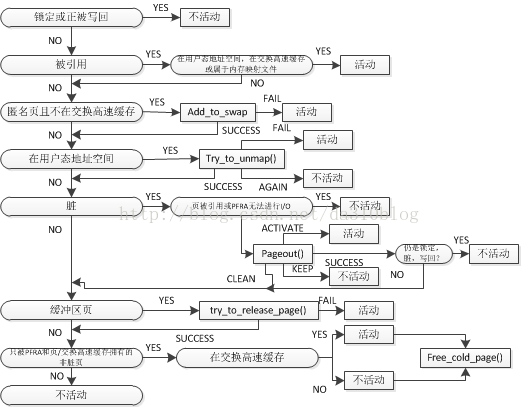
    shrink\_list()处理的页框可能的三种结果：

        1、调用free\_cold\_page()函数，把页释放到管理区伙伴系统；

        2、页没有被回收，因此被重新插入到page\_list链表，但是shrink\_list()认为不久就可以回收，则让页描述符的PG\_active保持清0，这样的页会被重新放到非活动链表中；

        3、页没有被回收，因此被重新插入到page\_list链表，但是页正在被使用，或者近期内无法回收，将PG\_active置1 ，这样的页会被放入活动链表。

    回收页框流程图：



**函数pageout();**

    函数分析：

        当一个脏页必须写回磁盘时，调用pageout()函数。

**3、周期回收**

    kswapd内核线程：调用shrink\_zone()和shrink\_slab()从LRU链表中回收页。

    cache\_reap函数：周期性的调用以便从slab分配器中回收未用的slab。

    kswapd内核线程

        有些内存分配请求是由中断和异常处理程序执行的，它们不会阻塞等待释放页框的当前进程。即：try\_to\_free\_pages()会引发阻塞。

        每个内存节点对应各自的kswapd内核线程。通常睡眠在等待队列中。

        如果\_\_alloc\_pages()发现页框不够，则会激活线程；安全页框数大于一个阈值，线程睡眠。

   cache\_reap()函数

        回收slab分配器里的页。在工作队列中被周期性调度

**4、内存不足删除程序**

    页框回收实在无法完成，调用删除程序。

    程序会挑选合适的进程进行删除：

        拥有大量页框；

        删除它只损失少量的工作成果（工作了几天的进程尽量不删除） ；

        具有较低的静态优先级；

        不是root特权的进程；

        不是直接访问硬件的进程；

        不是进程0，进程1，以及内核线程 。

**5、交换标记**

    交换失效现象：

        内存不足时，PFRA会全力地把页写入磁盘以释放内存并从进程中窃取页获得页框。

            进程要运行，也要全力地访问它们的页；

            页无休止的被写入磁盘并且再度读回，大部分时间都耗在访问磁盘上了。

        将交换标记赋予进程，该标记可以使该进程避免页框被回收，这样页框被持有的时间要长一些。

**四、交换**

    交换用来为非映射页在磁盘上提供备份，有三类页必须由交换子系统处理：

        \*进程匿名线性区（用户态堆栈）的页

        \*进程私有内存映射的脏页

        \*IPC共享内存的页

    交换对于程序必须是透明的，程序本身不知道被交换了。

    交换子系统的主要功能：

        \*在磁盘上建立交换区（swap area），用于存放没有磁盘映像的页；

        \*管理交换区空间。当需求发生时，分配与释放页槽；

        \*提供函数用于从RAM中把页换出到交换区或从交换区换入到RAM中；

        \*利用页表项（已被换出的换出页页表项）中的换出页标识符跟踪数据在交换区中的位置。

    为什么必须要用到交换？

        确保进程的所有页框都能被PFRA随意回收，而不仅仅是回收有磁盘映像的页（这种页直接写硬盘就好了）。

    交换分区拓展了内存地址空间，进程可以有效的使用，但是性能远远比不上RAM。

**1、交换区**

    实现：

        使用磁盘分区虚拟；

        包含在大型分区中的文件。

    可以定义多个交换区，最大个数由MAX\_SWAPFILES确定（一般是32）。

        多个交换区允许并发操作

        每个交换区都有一组页槽（4KB）组成，每块包含一个换出页。交换区的第一个页槽用来存放交换区的信息，其格式由swap\_header联合体（info和magic组成）来描述。

        swap\_header联合体（info、magic）：

**info：**

            bootbits

                交换算法不使用该字段。

            version

                版本

            last\_page

                可以有效使用的最后的一个页槽

            nr\_badpages

                有缺陷的页槽的个数

            padding

                填充字节

            badpages

                指定有缺陷页槽位置

**magic：**

            提供了一个字符串，用来把磁盘某部分明确标记为交换区。

**2、创建与激活交换区**

    系统关闭的时候，进程全部被杀死，所以交换区中的数据也会被丢弃。

    通常在创建linux系统中的其他分区的时候都会创建一个交换分区，使用mkswap把这个磁盘区设置成交换区，同时对swap\_header中的字段（第一个页槽）进行初始化。但是，mkswap命令会把交换区设置成非激活状态。

    每个交换区有一个或者多个交换子区组成，每个交换子区由一个swap\_extent描述符表示，每个子区对应一组槽，磁盘上物理相邻。

    swap\_extent描述符的组成：

            交换区的子区首页索引、子区的页数和子区的起始磁盘扇区号。

    激活交换分区的同时，组成交换区的有序子区链表也被创建。

    一般的，

        存放在磁盘分区中的交换区只有一个子区；

        但是存放在普通文件中的交换区则可能有多个子区（因为文件系统可能没把该文件全部分配在磁盘的一组连续快中）。

**3、如何在交换区中分布页**

    换出时，内核尽力把换出的页存放在相邻的页槽中，可以减少在访问交换区时磁盘的寻道时间，这是高效交换算法的一个重要因素。

    系统如果使用了多个交换区，事情变得复杂：

        查找空闲页槽的时候，从优先级最高的交换区中开始搜索。

**4、交换区描述符**

    swap\_info\_struct描述符——>>>

    重要字段：

        flags字段：

            SWP\_USED：交换区是否是活动的；

            SWP\_WRITEOK：交换区是否可写；

            SWP\_ACTIVE：若前面两个标识置位，那么置位。

        swap\_map字段：

                指向一个计数器数组，交换区每个页槽对应一个元素。

                若计数器值等于0，那么这个页槽空闲；如果计数器为正数，表示换出的页填充了这个页槽。

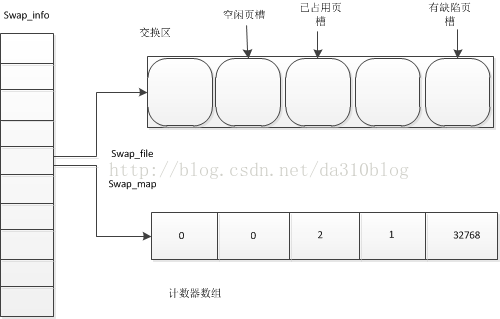
                页槽计数器的值表明了共享换出页的进程数

                计数器的值为32767，表明存放在页槽中的页是永久，不能删除；32768：页槽有缺陷，不能用

        prio字段：

                有符号整数，表示交换区子系统依据这个值考虑每个交换区的次序。

    swap\_info数组包括MAX\_SWAPFILES个交换区描述符。设置了SWP\_USED标识的交换区才被使用。



    活动交换区描述符也被插入到按交换区优先级排序的链表中。

    交换区描述符的next字段实现，这个next是int型，表明swap\_info数组中下一个描述符的索引。

    这个next和其他链表的实现有不同，依赖于swap\_info数组。

    swap\_list\_t类型的swap\_list：

        head:第一个链表元素在swap\_info数组中的下标

        next：换出页下一个交换区在数组中的下标。用于在具有空闲页槽的最大优先级的交换区之间实现轮询算法

        max字段：以页为单位交换区的大小

        pages字段：可用页槽的数目

        变量nr\_swap\_pages包含所有活动交换区中可用（空闲并且无缺陷）的页槽数

        变量total\_swap\_pages包含无缺陷页槽总输

**5、换出页标识符**

    如何唯一的标识一个换出页？

        swap\_info数字中找到交换区描述符，交换区描述符找到指定的页槽索引（第一个可用槽索引是1）

        31--------8|7--------1|0

            页槽索引      区号

        swp\_entry(type,offset)宏负责从交换区索引type和页槽索引offset构造出页标识符，swp\_type和swp\_offset宏正好相反。

        当页被换出的时候，标识符就作为页的表项插入到页表中。

        页标识符：

            空项：该页不属于进程地址空间，或相应的页框还没有分配给进程；

            前31个最高位不全等于0，最后一位等于0：页被换出；

            最低位等于1：该页包含在RAM中。

        同一个页可以属于几个进程的地址空间，可以把一个页换出多次，但只在swap中存储一次，剩下的增加swap\_map计数器。

**6、激活和禁用交换区**

    sys\_swapon()：激活交换区

    sys\_swapoff()：禁用交换区

**try\_to\_unuse()函数**

        使用一个索引参数，该参数标识待清空的交换区；

        该函数换入页并更新已换出页的进程的所有页表；

        函数从init\_mm内存描述符开始，访问所有内核线程和进程的地址空间。这个函数相当的耗时。

     函数扫描交换区的swap\_map数组。找到一个“在用的”页槽时，首先换入其中的页，然后查找该页的进程。

    页被加锁，没有进程可以访问它，它不会被另一个内核控制路径再次换出。

    交换区被标识为不可写，没有进程可以对这个页槽进行换出。

    如何实现？参考后面页高速缓存

    执行结果：swap数组中的每个页槽引用计数器都为空

**7、分配与释放页槽**

    释放内存的时候，内核要在很短的时间内把很多页都交换出去。因此需要尽力把页存放在相邻的页槽中

    如何扫描空闲页槽？

        Linux总是从最后一个已分配的页槽开始查找（除非已经到到交换区的末尾）

    交换区的swap\_info\_struct描述符的cluster\_nr字段存放空闲页槽数，当函数从交换区的开头重新分配时该字段置0。

    cluster\_next字段存放在下一次分配时要检查的第一个页槽索引。

    lowest\_bit和highest\_bit定义了最后一个和第一个可能为空的页槽，换计划说，所有在这个范围之外的认为是已经分配了

    scan\_swap\_map(交换区描述符)

        用来在给定的交换区中查找一个空闲页槽。返回一个空闲页槽的索引。

        函数执行步骤：

            1、检查交换区的cluster\_nr字段，若是正数，就从cluster\_next索引处元素开始对计数器的swap\_map数组进行扫描。查找一个空项。若找到了，则减少cluster\_nr字段的值。

            2、cluster\_nr为空，就把cluster\_nr重新初始化为256，再重新找

            3、cluster\_next开始搜索后找不到空项，从lowest\_bit索引处重新扫描，查找单独页槽，还没有，就把lowest\_bit设为数组最大索引，highest\_bit设为0，返回0

            4、已经找到了空项，减少nr\_swap\_pages的值，更新lowest\_bit和highest\_bit，inused\_page,cluster\_next。

    get\_swap\_page()

        搜索所有的活动的交换区来查找一个空闲页槽，返回新近分配的换出页标识符。

        搜索过程中考虑交换区优先级。

        是对scan\_swap\_map的调用。

        搜索过程：要执行两边搜索：

            第一遍，先查第一个交换区，若没有空页槽，查下一个，此时只在查找优先级和第一个相同的（最高优先级）

            第二遍，第一遍没找到，则从链表头开始查找全部的交换区。

       swap\_free(换出页标识符)

        换入页

            对相应的swap\_map计数器进行减一的操作，若计数器值为0，页槽的标识符不包含在任何进程页表项中。因此页槽就变得空闲。

**8、交换高速缓存**

    要解决的问题：

        多重换入：两个进程可能同时要换入一个页

        同时换入换出：一个进程可能换入正由PFGA换出的页

    多重换入：

        当第一个进程试图访问页的时候，内核开始换入页，第一步要检查该页在不在交换高速缓存中，

        若不在，那么内核就分配一个新的页框并把他插入到交换高速缓存，然后开始I/O操作，从交换区读入页

        第二个进程访问该共享匿名页，内核检查该页在不在交换高速缓存中，现在在了，因此内核只是访问也框描述符

        在PG\_locked标识清0之前，让进程睡眠

    同时换入换出：

        同理：A和B进程共享P页。B进行换出操作时，页会被存入页高速缓存，此时A不能换入

        若A没有换入操作，P会从页高速缓存中删除。

(1)、换出页

        PFRA已经确定了匿名页是否该被换出。内核如何执行换出操作？（这一步应该在回收页框之前）

        在shink\_list()内执行.

        第一步：准备交换高速缓存

            add\_to\_swap()函数在交换区中分配一个页槽，并把页框（页描述符地址）插入交换高速缓存

        第二步：调用try\_to\_unmap()，确定引用匿名页的每个用户态页表项地址，将换出页标识符写入其中

        第三步：将页写入交换区，调用swap\_writepage()实现

        第四步：从交换高速缓存中删除页框

(2)、换入页

        当进程对一个已经被换出的页进行寻址时，必然会发生页的换入。

        缺页异常处理程序引发换入操作。

        引起异常的地址是一个有效的页，属于当前进程的一个线性区

        页不在内存中时，页表项的Present标识被清除。

        与页有关的页表项不为空，但Dirty位清0。页表项包含一个换出页标识符，调用do\_swap\_page()。