blog.51cto.com

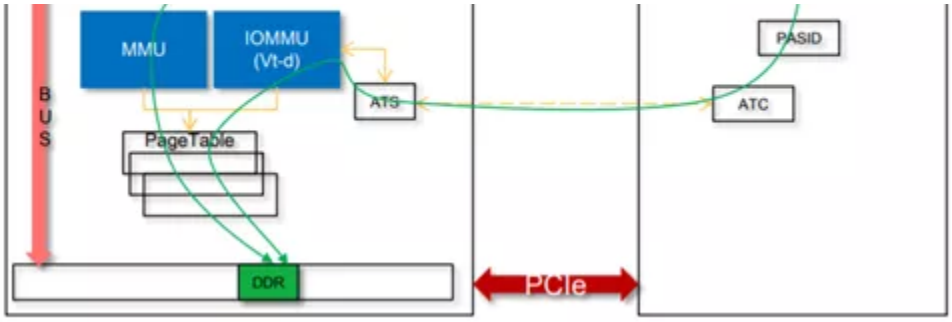
宋宝华：论Linux的页迁移（Page Migration）完整版_mb6066e53e0c12a的技术博客_51CTO博客

55–68 minutes

本文目录 ●●

对于用户空间的应用程序，我们通常根本不关心page的物理存放位置，因为我们用的是虚拟地址。所以，只要虚拟地址不变，哪怕这个页在物理上从DDR的这里飞到DDR的那里，用户都基本不感知。那么，为什么要写一篇论述页迁移的文章呢?

我认为有2种场景下，你会关注这个Page迁移的问题：一个是在Linux里面写实时程序，尤其是Linux的RT补丁打上后的情况，你希望你的应用有一个确定的时延，不希望跑着跑着你的Page正在换位置而导致的延迟；再一个场景就是在用户空间做DMA的场景，尤其是SVA（SharedVirtual Addressing），设备和CPU共享页表，设备共享进程的虚拟地址空间的场景，如果你DMA的page跑来跑去，势必导致设备DMA的暂停，设备的传输性能出现严重抖动。这种场景下，设备的IOMMU和CPU的MMU会共享Page table：



1. CoW导致的页面迁移

1.1 fork

典型的CoW（写时拷贝）与fork()相关，当父子兄弟进程共享一部分page，而这些page本身又应该是具备独占属性的时候，这样的page会被标注为只读的，并在某进程进行写动作的时候，产生page fault，其后内核为其申请新的page。比如下面的代码中，把10写成20的进程，在写的过程中，会得到一页新的内存，data原本的虚拟地址会指向新的物理地址，从而发生page的migration。

```
int data = 10;

int child_process()
{
        printf("Child process %d, data %d\n",getpid(),data);
        data = 20;
        printf("Child process %d, data %d\n",getpid(),data);
        _exit(0);
}

int main(int argc, char* argv[])
{
        int pid;
        pid = fork();

        if(pid==0) {
                child_process();
        }
        else{
                sleep(1);
                printf("Parent process %d, data %d\n",getpid(), data);
                exit(0);
        }
}
```

类似的场景还有页面的私有映射引发的CoW：

```
int main(int argc, char **argv)
```

```
{
        unsigned char *m;
        int i;
        pid_t pid;

#define MAP_SIZE 1024*1024
        m = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE,
                        MAP_PRIVATE | MAP_ANON, -1, 0);

        for (i = 0; i < MAP_SIZE; i++)
                m[i] = 10;

        pid = fork();
        if (pid == -1)
                exit(1);
        else if (pid == 0) {
                while(1);
        } else {
                sleep(30);
                /* CoW */
                for (i = 0; i < MAP_SIZE; i++)
                        m[i] = 20;
                printf("cow is done in parent process\n");
                while(1);
        }
}
```

运行这个程序，通过smem观察，可以明显看到，父进程执行新的写后（程序打印“cow is done in parent process”后），父子进程不再共享相关page，他们的USS/PSS都显著增大：

由fork衍生的写时拷贝场景相对来说比较基础，在此我们不再赘述。

1.2 KSM

其他的CoW的场景有KSM（Kernel same-page merging）。KSM会扫描多个进程的内存，如果发现有page的内容是一模一样的，则会将其merge为一个page，并将其标注为写保护的。之后对这个page执行CoW，谁写谁得到新的拷贝。比如，你在用qemu启动

一个虚拟机的时候，使用mem-merge=on，就可以促使多个VM共享许多page，从而有利于实现“超卖”。

sudo /x86_64-softmmu/qemu-system-x86_64 \-enable-kvm -m 1G \-machine mem-merge=on

- 1.

不过这本身也引起了虚拟机的一些安全漏洞，可被side-channel攻击。

比如，把下面的代码编译为a.out，并且启动两份a.out进程

./a.out&./a.out

- 1.

代码：

我们看到这2个a.out的内存消耗情况如下：

但是，如果我们把中间的if 0改为if 1，也就是暗示mmap()的这1MB内存可能要merge，则耗费内存的情况发生显著变化：

耗费的内存大大减小了。

我们可以看看pageshare的情况：

Merge发生在进程内部，也发生在进程之间。

当然，如果在page已经被merge的情况下，谁再写merge过的page，则会引起写时拷贝，比如如下代码中的p[0]=100这句话。

2. 内存规整导致的页面迁移

2.1 CMA引起的内存迁移

CMA (The *Contiguous* Memory Allocator)可运行作为dma_alloc_coherent()的后端，它的好处在于，CMA区域的空闲部分，可以被应用程序拿来申请MOVABLE的page。如下图中的一个CMA区域的红色部分已经被设备驱动通过dma_alloc_coherent()拿走，但是蓝色部分目前被用户进程通过malloc()等形式拿走。

一旦设备驱动继续通过dma_alloc_coherent()申请更多的内存，则内核必须从别的非CMA区域里面申请一些page，然后把蓝色的区域往新申请的page移走。用户进程占有的蓝色page发现了迁移。

CMA在内核的配置选项中依赖于MMU，且会自动使能MIGRATION（Pagemigration）和MEMORY_ISOLATION：

2.2 alloc_pages

当内核使能了COMPACTION，则Linux的底层buddy分配器会在alloc_pages()中尝试进行内存迁移以得到连续的大内存。COMPACTION这个选项也会使能CMA一节提及的MIGRATION选项。

从代码的顺序上来看，alloc_pages()分配order比较高的连续内存的时候，是优先考虑COMPACTION，再次考虑RECLAIM的。

2.3 /proc/sys/vm/compact_memory

当然，上面alloc_pages所提及的compaction也可以被用户手动的触发，触发方式：

echo 1 >/proc/sys/vm/compact_memory

- 1.

将1写入compact_memory文件，则内核会对各个zone进行规整，以便能够尽可能地提供连续内存块。

我的Ubuntu已经运行了一段时间，内存稍微有些碎片化了，我们来对比下手动执行

compact_memory前后，buddy的情况：

可以清晰地看出来，执行compact_memory后，DMA32 ZONE和NORMAL ZONE里面，order比较大的连续page数量都明显增大了。

2.4 huge page

再次展开内核的COMPACTION选型，你会发现COMPACTION会被透明巨页自动选中：

这说明透明巨页是依赖于COMPACTION选项的。

所谓透明巨页，无非就是应用程序在运行的时候，神不知鬼不觉地偷偷地就使用到了Hugepage的功能，这个过程对用户是透明的。与透明对应的无非就是不透明的巨页，这种方式下，应用程序需要显示地告诉内核我需要使用巨页。

我们先来看看不透明的巨页是怎么玩的？一般用户程序可以这样写，在mmap里面会加上MAP_HUGETLB的Flag，当然这个巨页也必须是提前预设好的，否则mmap就会失败。

ptr_ = mmap(NULL, memory_size_, PROT_READ | PROT_W

- 1.

比如下面的代码我们想申请2MB的巨页：

程序执行的时候会返回错误，打印如下：

$ ./a.out Hugetlb: Cannot allocate memory

- 1.

原因很简单，因为现在系统里面2MB的巨页数量和free的数量都是0：

我们如何让它申请成功呢？我们首先需要保证系统里面有一定数量的巨页。这个时候我们可以写nr_hugepages得到巨页：

我们现在让系统得到了10个大小为2048K的巨页。

现在来重新运行a.out，就不在出错了，而且系统里面巨页的数量发生了变化：

Free的数量从10页变成了9页。

聪明的童鞋应该想到了，当我们尝试预留巨页的时候，它最终还是要走到buddy，假设系统里面没有连续的大内存，系统是否会进行内存迁移以帮忙规整出来巨页

呢？这显然符合前面说的alloc_pages()的逻辑。从alloc_buddy_huge_page()函数的实现也可以看出这一点：

另外，**这种巨页的特点是“预留式”的，不会free给系统，也不会被swap。**因此可有效防止用户态DMA的性能抖动。对于DPDK这样的场景，人们喜欢这种巨页分配，减少了页面的数量和TLB的miss，缩短了虚拟地址到物理地址的重定位的转换时间，因此提高了性能。

当然，我们在运行时通过写nr_hugepages的方法设置巨页，这种方法未必一定能够成功。所以，工程中也可以考虑通过内核启动的bootargs来设置巨页，这样Linux开机的过程中，就可以直接从bootmem里面分配巨页，而不必在运行时通过order较高的alloc_pages()来获取。这个在内核文档的kernel-parameters.txt说的比较清楚，你可以在bootargs里面设置各种不同hugepagesize有多少个页数：

透明巨页听起来是比较牛逼的，因为它不需要你在应用程序里面通过MAP_HUGETLB来显式地指定，但是实际的使用场景则未必这么牛逼。

使用透明巨页的最激进的方法莫过于把enabled和defrag都设置为always：

echo always >/sys/kernel/mm/transparent_hugepage/enabledecho always >/sys/kernel

/mm/transparent_hugepage/defrag

- 1.

  enabled写入always暗示对所有的区域都尽可能使用透明巨页，defrag写入always暗示内核会激进地在用户申请内存的时候进行内存回收（RECLAIM）和规整（COMPACTION）来获得THP（透明巨页）。

  我们来前面的例子代码稍微进行更改，mmap16MB内存，并且去掉MAP_HUGETLB：

  运行这个程序，并且得到它的pmap情况：

  我们发现从00007f46b0744000开始，有16MB的anon内存区域，显然对应着我们代码里面的mmap(16*1024*1024)的区域。

  我们进一步最终/proc/15371/smaps，可以得到该区域的内存分布情况：

  显然该区域是THPeligible的，并且获得了透明巨页。内核文档filesystems/proc.rst对THPeligible的描述如下：

  "THPeligible" indicates whether the mapping is eligible for allocating THP pages - 1 if true, 0 otherwise. It just shows the current status.

  透明巨页的生成，显然会涉及到前面的内存COMPACTION过程。透明巨页在实际的用户场景里面，可能反而因为内存的RECLAIM和COMPACTION而

降低了性能，比如有些VMA区域的寿命很短申请完使用后很快释放，或者某些使用大内存的进程是短命鬼，进行规整花了很久，而跑起来就释放了这部分内存，显然是不值得的。类似《*权力的游戏*》中的夜王，花了那么多季进行内存规整准备干夜王这个透明巨页，结果夜王上来就被秒杀了，你说我花了多时间追剧冤不冤?

所以，透明巨页在实际的工程中，又引入了一个半透明的因子，就是内核可以只针对用户通过madvise()暗示了需要巨页的区间进行透明巨页分配，暗示的时候使用的参数是MADV_HUGEPAGE：

所以，默认情况下，许多系统会把enabled和defrag都设置为madvise：

echo madvise >/sys/kernel/mm/transparent_hugepage/enabledecho madvise >/sys/kernel/mm/transparent_hugepage/defrag

- 1.

或者干脆把透明巨页的功能关闭掉：

echo never >/sys/kernel/mm/transparent_hugepage/enabledecho never >/sys/kernel/mm/transparent_hugepage/defrag

- 1.

如果我们只对madvise的区域采用透明巨页，则用户的代码可以这么写：

既然我都已经这么写代码了，我还透明个什么鬼？所以，我宁可为了某种确定性，而去追求预留式的，非swap的巨页了。

3. NUMA Balancing引起的页面迁移

在一个典型的NUMA系统中，存在多个NODE，很可能每个NODE都有CPU和Memory，NODE和NODE之间通过某种总线再互联。下面中的NUMA系统有4个NODE，每个NODE有24个CPU和1个内存，NODE之间通过红线互联：

在这样的系统中，通常CPU访问本地NODE节点的memory会比较快，而跨NODE访问memory则会慢很多（红色总线慢）。所以Linux的NUMA自动均衡机制，会尝试将内存迁移到正在访问它的CPU节点所在的NODE，如下图中绿色的memory经常被CPU24访问，但是它位于NODE0的memory：

则Linux内核可能会将绿色内存迁移到CPU24所在的本地memory：

这样CPU24访问它的时候就会快很多。

显然NUMA_BALANCING也是依赖MIGRATION机制的：

下面我们来写个多线程的程序，这个程序里面有28个线程（一个主线程，26个dummy线程执行死循环，以及一个写内存的线程）：

我们开那么多线程的目的，无非是为了让write_thread_start对应的线程，尽可能地不被分配到主线程所在的NUMA节点。

这个程序的主线程最开始写了64MB申请的内存，30秒后，通过write_done=1来暗示write_thread_start()线程你可以开始写了，write_thread_start()则会把这64MB也写一遍，如果主线程和write_thread_start()线程不在一个NODE节点的话，内存迁移就有可能发生。

这是我们刚开始2秒的时候获得的该进程的numastat，可以看出，这64MB内存几乎都在NODE3上面：

但是30秒后，我们再次看它的NUMA状态，则发生了巨大的变化：

64MB内存跑到NODE1上面去了。由此我们可以推断，write_thread_start()线程应该是在NODE1上面跑，从而引起了这个迁移的发生。

当然，我们也可以通过numactl--cpunodebind=2类似的命令来规避这个问题，比如：

numactl --cpunodebind=2 ./a.out

- 1.

NUMA Balancing的原理是通过把进程的内存一部分一部分地周期性地进行unmap（比如每次256MB），在页表里面把扫描的部分的PTE设置为 “no access permission”，以在其后访问它的时候，强制产生

pagefault，进而探测page fault发生在本地NODE还是远端NODE，来获知CPU和memory是否较远的。这说明，哪怕没有真实的迁移发生，NUMA balancing也会导致进程的内存访问出现Page fault。

4. Page migration究竟是怎么做的?

内存规整和NUMA平衡等引发的Page migration的过程，一言以蔽之，就是把一个page从A位置移动到B位置。正所谓“当官的一动嘴，当差的跑断腿”。这个过程真地是一点都不简单。在一个真实的工程场景中，由于共享内存等原因，一个page可能被多个进程share，被映射到多个进程的VMA里面去，所以这个迁移过程，要伴随多个进程的虚实映射的更改（迁移前后虚拟地址都是不变的）以及相关address_space的radix tree从指向A到指向B的变更。此外，新page对应的一些flags，应该和旧page是一样的。

如果时间停止，比如一旦我们开启了迁移page A到page B的过程，进程1,2,3,4都站那里不动了，内核其他人也站那里不动了，这个过程还相对来说比较简单。真实的情况是，在我们把A迁移到B的过程中，进程1,2,3,4应该有可能还会访问迁移中的page对应的虚拟地址。当迁移正在进行中，“树欲静而风不止”，用户进程和内核其他子系统不会因为你“想静静”就给你“静静”。

迁移的步骤非常繁杂，我们抓主干放弃细节，主要以

clean的page的迁移为分析目标，抛开dirty页面的writeback问题。整个迁移的过程中，除了进程1-4可以访问page里面的内容，内核内存管理子系统其他的代码，也可能访问page对应的元数据。应该避免内核的内存管理的其他子系统进来干扰。比如，我正在迁移pageA，结果你的内存管理子系统还在LRU里面扫描A，比如把pageA从inactiveLRU移到activeLRU。明天新皇帝就要打进京城了，老皇帝今天晚上还在修皇宫，这么无私奉献的皇帝也有？所以，迁移的过程中，我们先要用__isolate_lru_page()把page从LRU里面隔离出来，不要再在明天就要挂的皇帝上面费心思了。因为，你一边在修改老page的元数据，一边老page在灭亡中，这也太乱了吧？还有一种情况，A皇明天就要禅位给B皇，结果你今天晚上把A皇灭了，比如pageA被回收释放了，那么明天的皇帝交接仪式还如何进行?

__isolate_lru_page()的核心代码如下，它阻止了Page A被释放，同时也避免LRU的扫描动作:

```
        if (likely(get_page_unless_zero(page))) {                /*
```

1.

它把page的_refcount进行了增加，这样这个pageA就不会在迁移给B的过程中，被内存回收子系统free掉。另外，ClearPageLRU()会清除掉page的PG_lru标记，使得PageLRU(page)为假，从而阻止了很多类似如下的LRU代码路径：

隔离的准备工作完成，我们开始进行迁移，然后就到了令人痛不欲生的内核函数：

int migrate_pages(struct list_head *from, new_page_t get_nev

1.

如果你看内核的

https://www.kernel.org/doc/html/latest/vm/page_migration.html

文档，相信看到这个地方，一共有20步，一定懵逼了：

这个步骤太多，比较难以理顺，所以我们重点是抓核心矛盾：迁移必须是无缝的，不能因为你的迁移导致进程不能正常运行，迁移过程中，进程还是可能访问正在迁移的page！当页面A正在迁移到B的过程，原先的进程1-4访问对应的虚拟地址的时候，这个访问其实没有什么不合法。所以，我们必须保证它还是可以访问，当然这个访问可能是有延后的。大体可以分为3个阶段：

在迁移的早期阶段，进程的页表项还是指向page A的，映射了page A的用户进程还是可以无障碍地访问到A；

在迁移的中期阶段，他们访问不到A了，因为A被unmap了，但是这个时候它其实也访问不了B，因为B还没有map。有那么一段时间是一个皇位悬空的状态。但是进程1-4访问这个虚拟地址没有什么不合法啊！我们甚至都不需要让进程1-4感知到我们正在迁移。所以，在这个悬空的期间，进程1-4对虚拟地址进行访问的时候，我们应

该能感知到这个访问的发生并让他们稍作等待，以便新皇登基后他们继续访问；

在迁移的末期阶段，page B被map，新皇登基完成，先皇被抛弃，我们应该唤醒在“中期阶段”的进程1-4对该虚拟地址的访问，并让进程1-4能访问到page B。

第2阶段，根据被迁移的page寻找进程，寻找VMA，以操作他们的PTE，用到了反向映射技术RMAP；而PTE修改后，用户进程继续访问相关page对应的虚拟地址，则会发生page fault，发生page fault后，进程阻塞等待，等待在发生page fault的page的lock上面，这一点从__migration_entry_wait()这个page fault处理函数的核心函数的注释和代码流程可以看出：

put_and_wait_on_page_locked()会将进程放入等待队列，等待page的PG_locked标记被清楚，也就是等待第3阶段的完成，从而实现了无缝地交接。

5. 如何规避页迁移

5.1 mlock可以吗?

mlock()的主要作用是是防止swap。我们可以用mlock()锁住匿名页或者有文件背景的页面。如果匿名页被mlock住了，对应的内存不会被写入swap分区；如果文件背景的页面被mlock住了，相关的pagecache不会被reclaim。有文件背景的页面的mlock稍微有点难理解，所以我们用如下代码演示了有文件背景的mmap区域被

mlock：

由于我们mlock了，所以这4096个字节必须常驻内存！但是mlock()并不能保证这4096个字节对应的物理页面不迁移，它最主要的作用是防止page被swap。这一点从mlock()的manpage上面看地非常清楚：mlock(),mlock2(), and mlockall() lock part or all of the calling process's virtualaddress space into RAM, preventing that memory from being paged to the swaparea.

为了实际证明被mlock()的page还是可能被迁移，我们把前面演示NUMA Balance页迁移的程序稍微进行改动，把malloc的64MB内存进行mlock():

再运行这个程序，效果如下：

我们明显看出，Node0上面的64MB内存，随着write_thread_start()的执行，被迁移到了NODE1。

在Linux中，执行mlock()操作的时候，相应的VMA会被设置VM_LOCKED标记。当然，用户mlock()的区域大小，并不一定正好等于mlock()对应区域的VMA大小，所以mlock()的过程中，会进行VMA的拆分或合并（原因是同一个VMA里面的flags应该是一样的）。我们都本着眼见为实的原则，用代码来说话。把前面那段文件背景页面的被mlock()前后进程的VMAdump一下，我们稍微改一下代码，mmap 1M但是只mlock其中的4K：

在mlock前后我们dump这个进程的VMA。mlock前，我们看到一段1024K的VMA区域，背景为/home/baohua/file：

在mlock()后我们再次观察，我们看到这个1024k被拆分了1020k+4k：

开始的4K由于有了单独的VMA特性，1024被拆分为4+1020两个VMA。

mlock()锁定区域的page，也会被mark成PG_mlocked（注意不是PG_locked），相关页面会被放入unevictable_list链表，page变成不可回收的page，从LRU剥离。与unevictable相反的page是可回收的，此类page会进入inactive或者activeLRU，内核的swap机制会扫描相关page和LRU决定对最不活跃的page进行回收。

根据内核文档

https://www.kernel.org/doc/Documentation/vm/unevictable-lru.txt，

Linux supports migration of mlocked pages and otherunevictable pages. This involves simplymoving the PG_mlocked and PG_unevictable states from the old page to the newpage.

所以，对于mlock()的page，还是可能被迁移，只是迁移后，在迁移完成的目标page上面，保留了原先的PG_mlocked标记，新的page还是LRU剥离的，仍然不

会被swap机制扫描到。

至于这种不可回收page的内存规整动作是否应该进行，则可以由/proc/sys/vm/compact_unevictable_allowed参数进行控制。如果我们使能了它，内存规整的代码也会扫描unevictableLRU，对不可回收page进行页面迁移。

透明巨页是一个神奇的存在，假设我们透明巨页的大小是2MB，如果我们mlock()其中的1MB，则没有被mlock()的1MB是可以被回收的，当内核进行内存回收动作的时候，可以将2MB再次拆分为一个个的4KBpage，从而保证没有被mlock()的部分可以被回收。

5.2 GUP可以吗?

我们已经发现mlock()不具备防止迁移的功能。那么，如果我们要用用户空间的buffer做DMA，常规的途径是什么呢？Linux内核可以用GUP（get_user_pages的衍生变体），来pin住page，从而避免相关的page被迁移或被swap代码释放。这实际上给针对用户空间的buffer进行DMA提供了某种形式的安全保障。

前面我们说过，mlock()可以避免page的swap，但是没法避免page的迁移和old page的释放，所以，如果我们把用户空间的一片被mlock()的区域交给外设去做DMA，比如DMA的方向是DMA_FROM_DEVICE，由于设备感知到的是page A（app通过参数传递的方式给驱动传递了用户空间buffer，传递的时候buffer还对应page A），

但是实际在DMA发生前，可能内存已经被迁移到了B，Page A甚至都释放给buddy或者被别的程序申请走了，这个时候设备做DMA，还是往A里面写，那么极有可能导致莫名其妙的崩溃发生。

所以，对于用户空间的buffer，直接进行DMA的场景，我们必须保证

1. 当我们在app里把buffer传递给driver的时候，假设这个时候buffer对应page A
2. 在我们做DMA的时刻，以及DMA进行中的时刻，pageA都必须是一直存在的。这个pageA**不能被释放**、重新分配给第三者。

具体是如何做到的呢？我们可以看看V4L2驱动，如果用户空间传一个指针进来进行DMA，V4L2驱动的内核态是如何处理的,相关代码位于drivers/media/v4l2-core/videobuf-dma-sg.c：

videobuf_dma_init_user_locked()根据用户空间buffer的开始地址和大小，计算出page的数量，然后调用pin_user_pages()把这些page在内核pin住。

pin_user_pages()这个函数位于mm/gup.c，属于GUP家族函数的一部分。该函数的功能，顾名思义，就是把用户的page pin住。Pin住和mlock()是完全不同的概率，mlock()是针对VMA添加VMA_LOCKED属性，针对VMA映射的page添加PG_mlocked属性，要求page不可

reclaim不被swap，但是从pageA到page B的迁移仍然可能发生，pageA仍然可能被释放，但是这个释放也无关紧要，因为我们已经有了B。所以本质上，**mlock(p, size)是对虚拟地址p ~ p+size-1常驻内存的一种保障，或者说是对有VM_LOCKED属性的VMA区域有常驻内存的page的一种保障，至于常驻的是A还是B，其实都不违背mlock()的语义。**

mlock强调长期有人值班，不能掉链子，而不在乎你换不换班。Pin是完全不同的语义，pin强调的是贞节牌坊式的坚守。pin是把buffer对应的page本身让它无法释放，因为pin的过程，会导致这个page的引用计数增加GUP_PIN_COUNTING_BIAS（值为1024，我理解是一种代码的hack，以区分常规的get_user_pages和pin_user_pages，常规的get只是+1，而pin的主要目的是为了用user space的buffer做DMA操作）：

Pin page并没有改变page所在VMA区域的语义，这是和mlock()之间很大的区别，mlock()会导致VMA区域得到VM_LOCKED，但是我们并没有一个VM_PINNED这样的东西。从用户场景上来看，内核驱动Pin住了USERPTR对应的memory后，进一步，V4L2的代码videobuf_dma_map()会把这个buffer转换为sg进行map_sg操作，之后就可以DMA了。所以pin最主要的作用是保证DMA动作不至于访问内存的时候踏空。

如果我们不再需要这些userbuffer来进行DMA，相关的

page应该被unpin：

Unpin显然会导致一次引用计数减去GUP_PIN_COUNTING_BIAS的动作，如果ref减去GUP_PIN_COUNTING_BIAS的结果成为0，证明我们是最后一个用户，应该调用__put_page()促使page被正确释放：

这里还有一个特别值得关注的地方，如果我们进行的是DMA_FROM_DEVICE，也就是设备到内存的DMA，则相关的page可能被设备而不是CPU写dirty了，我们把unpin_user_pages_dirty_lock()的最后一个参数设置为了TRUE，这样unpin_user_pages_dirty_lock()会把相关Page的dirty标记置上，否则Linux内核甚至都不知道设备写过这个page。

pin_user_pages()在内核态增加了page的refcount，从而让内存管理的其他子系统不会释放这个page（尽管它没有像mlock那样把page从LRU上面拿下来，但是swap的代码也不可能释放它），也成功地阻止了page migration在这一个page上面的发生，甚至我们想去hot remove被pin的page所在的内存，都不可能成功了。

对于我们关注的页面迁移问题，pagemigration的代码会检查page的refcount，refcount不合适的page（pin本身导致了refcount不符合expected_count），不会被迁移：

所以，被pin的userspace page总体是DMA安全的。这里还有2个比较tricky的问题：

1. 就是用户的page，被内核pin住了，用户在unpin之前进行unmap呢？首先我们不欢迎这样的动作，对于pin的主要场景如RDMA、VFIO等，我们强调pin住的page是LONGTERM的，一般用户空间弄好buffer，应该一直复用这个buffer。详见内核文档：core-api/pin_user_pages.rst

也可以看include/linux/mm.h的一段注释：

实际上，用户空间应该控制这个buffer是indefinite的。如果万一pin住的页面用户层面真地在unpin之前就unmap了呢？pin的refcount其实也阻止了这个page被释放，只是这个page所在对应的虚拟地址由于被unmap了，所以不再对CPU可见了。

当然，还有一个madvise(p, size, MADV_DONTNEED)，其行为更加诡异，曾经臭名昭著的DirtyCoW漏洞(CVE-2016-5195)，就与它息息相关。它并不是unmap，但是暗示内核它对某段地址访问暂时完成了使命很长一段时间不想访问了，并且去掉了页表的映射。这样后期再次访问这个地址p~ p+size-1的时候，如果p底层对应的是一个共享映射，可以获得老page的值；如果底层对应的是一个MAP_PRIVATE的映射，则按需获得一个0填充的页面：

MADV_DONTNEED显然也是与pin的语义是违背的，这属于不“under userspace control”的情况了。如果我们真地对私有映射的pin区域执行了MADV_DONTNEED呢？显然pin也阻止了相关page被释放，但是当CPU重新访问page对应的虚拟地址的时候，MADV_DONTNEED却要求“the caller will see zero-fill-on-demand pages upon subsequent page references”，所以你不能指望先用MADV_DONTNEED暗示了自己不要这个memory了，然后再反过来期待DMA后CPU还能获取到自己曾经抛弃的内存，因为“是你当初说分手，分手就分手，现在又要用真爱把我哄回来，爱情不是你想卖，想买就能卖，让我挣开，让我明白，放手你的爱”。

下面的代码映射100MB，但是只对其中的前2页执行MADV_DONTNEED：

```c
#define SOMESIZE (100*1024*1024)        // 100MB

int main(int argc, char *argv[])
{
        unsigned char *buffer;
        int i;
        int ret;

        buffer = mmap(0, SOMESIZE, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, -1, 0);
        if (!buffer)
                exit(-1);

        /*
         * Touch each page in this piece of memory to get it
         * mapped into RAM
         */
        for (i = 0; i < SOMESIZE; i += 4 * 1024)
                buffer[i] = 100;

        printf("before madvise: %d %d %d\n", buffer[0], buffer[4096], buffer[8192]);

        madvise(buffer, 8192, MADV_DONTNEED);

        printf("after madvise: %d %d %d\n", buffer[0], buffer[4096], buffer[8192]);

        while(1);
        return 0;
}
```

程序运行打印的结果如下：

```
baohua@baohua-VirtualBox:~$ ./a.out
before madvise: 100 100 100
after madvise: 0 0 100
```

前2页变成了0，第3页还是100。

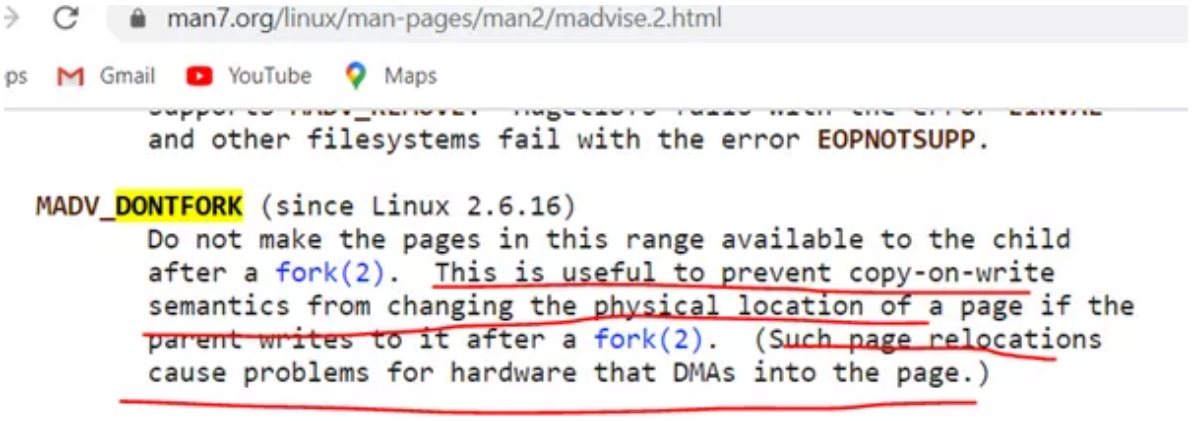
2.如果pin住的区域是一个私有映射的区域（mmap的时候，使用了MAP_PRIVATE），然后在此区域执行了CoW会怎样?

在下面的代码中，我们有个假想的设备驱动/dev/kernel-driver-with-pin-support，它支持一个IOCTL来让用户pin userspace的page。此例中，pin的这部分内存是私有映射的，而父进程又fork了子进程，之后父进程尝试写自己pin过的区域，这时会引发写时拷贝（CoW），从而让p对应的底层page发生变化：

```
void cow_pin_example(void)
{
        u8 *p;
        int fd = open("/dev/kernel-driver-with-pin-support", O_RDWR);
        struct gup_param gup;
        pid_t pid;
        int page_nr =  SIZE_1M / 4096;

        p = mmap(NULL, SIZE_1M, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS , -1, 0);

        // some write operations
        for (i = 0; i < page_nr; i++);
                p[i] = 10;

        // call driver to pin private anon pages
        gup.addr = p;
        gup.page_nr = SIZE_1M/4096;
        ioctl(fd, LONGTERM_PIN,  &gup);

        pid = fork();
        if (pid==-1)    {
                perror("Cannot create new process");
                exit(1);
        } else  if (pid==0) {
                // child
        } else {
                // parent
                // some write operations
                for (i = 0; i < page_nr; i++);
                        p[i] = 20; // cow
                ...
        }
}
```

这种编程方式，实际上也违背了用户空间应该控制这个

buffer是indefinite的原则，只是更加隐晦，所以要用userspace buffer做DMA的话，也应该尽可能避免这种编程方式，一般要对用户态buffer做DMA的进程，应该是叶子进程。如果一定要fork，可以考虑对pin住的区域执行madvise(p, size, MADV_DONTFORK)，以避免CoW，具体的原理在madvise的man page上说地很清楚：



5.3 使用巨页?

我们如果用户空间是用非透明的巨页的话，实际上这种巨页是“预留式”的，因此可有效的避免内存的回收、swap和因为memory compaction、Automatic NUMA balancing等导致的page migration等。一般用户这么获得hugetlb内存：

base_ptr_ = mmap(NULL, memory_size_, PROT_READ | PR(

- 1.

它实际上免去了对pin、mlock()等的需求。由于巨页本身可减小TLB miss以及walk page table的路径长度，想必可带来性能提升。

一般情况下，当我们想在系统预留n个巨页的时候，若系统有m个NUMA节点，Linux会倾向于每个NUMA节点预留n/m个巨页。如果采用内存本地化的NUMA策略，任务运行在节点A的话，内核会倾向于从节点A预留的巨页向目标任务分配，所以我们最好是对任务进行一下NUMA的绑定。

6. 最后的话

在对用户空间代码进行轻微控制的情况下，GUP是比较理想的、抗性能抖动的、安全的针对用户空间buffer进行DMA的途径。当然，某些场景下，用户空间程序具有很大的灵活多变性，比如经常要动态malloc、free，我们又要在这样的heap上面去做DMA，每次都去pin和unpin显然是不现实的。这个时候我们可以借用Linux实时编程技术里面常常采用的，让malloc/free在一个存在的堆池发生分配和释放的技术：

```
#define SOMESIZE (100*1024*1024)        // 100MB

int main(int argc, char *argv[])
{
        unsigned char *buffer;
        int i;
        int fd = open("/dev/driver-with-pin".....);

        if (!mlockall(MCL_CURRENT | MCL_FUTURE))
                mallopt(M_TRIM_THRESHOLD, -1UL);
        mallopt(M_MMAP_MAX, 0);

        buffer = malloc(SOMESIZE);
        if (!buffer)
                exit(-1);

        /*
         * Touch each page in this piece of memory to get it
         * mapped into RAM
         */
        for (i = 0; i < SOMESIZE; i += 4 * 1024)
                buffer[i] = 0;

        /* call driver to pin userspace heap */
        ioctl(fd, PIN, to_gup(buffer));

        free(buffer); /* M_TRIM_THRESHOLD will help hold the heap in userspace */

        /* the below malloc/buffer will occur in the existing heap */
```

```
        q = malloc(...);
        do_dma(q);
        free(q);
        ...

        return 0;
}
```

诀窍就在于先申请一个大堆，通过mallopt(M_TRIM_THRESHOLD, -1UL)避免大堆在free的时候还给内核。之后所有的malloc,free，都是在事先预留好的大堆进行，不再需要每次malloc/free区域单独pin，它带来了类似SVA的编程灵活性。

如果用户空间采用无GUP的SVA方案，我们可能需考虑屏蔽如下功能以防止性能抖动：

- 避免使用transparent huge pages (THP)；
- 避免使用kernel samepage merging (KSM)；
- 避免使用automatic NUMA balancing；
- 避免运行时修改非透明巨页的数量；
- 避免因为CMA而触发的页迁移；
- 对需要做DMA的区域执行mlock()避免其被swap。

当然上述问题几乎都可以被用户应用使用非透明巨页来规避，因为非透明巨页天然获得“预留式”内存的特征。当然这种方法绑死了用户应用必须使用hugetlbfs的逻辑，可能要求用户修改bootargs等来在系统启动过程中预留巨页。