转

陈延伟：任督二脉之内存管理总结笔记

2018年04月20日 00:00:00 Linuxer_

任督二脉之内存管理第一节课总结

本文是任督二脉之内存管理课程第一节课的总结说明，由于水平有限，可能无法对宋老师所讲完全理解通透，如有错误，请及时指证。

本文从5个方面进行说明：

1、 物理/虚拟/总线地址概念说明。

2、 MMU是什么，为什么，怎么做。

3、 内存分区和内存映射区。

4、 Buddy算法是个什么鬼。

5、 CMA的工作原理。

物理/虚拟/总线地址概念说明

所谓一花一世界，一叶一菩提，相同的事物在不同的角度可能会有不同的看法，对于物理地址，虚拟地址，总线地址的概念也是如此。

物理地址是MMU的视角所看到的内存地址。

虚拟地址是存在MMU的前提下CPU所看到的内存地址，当然我们实际编程的时候操作的也是虚拟地址。

总线地址是设备的视角所看到的内存地址。

比如一块内存，物理地址是0，在设备端看起来是0x80000000，而物理地址0又通常被映射为虚拟地址0xc0000000，从而同一地址就具备了三个身份，但他们在物理上指的是同一片区域。

归根结底，不论是MMU，CPU或程序员，还是设备，他们的终极目的是操作内存，至于怎么操作，它们又都有各自的比较舒服的操作方式，就是所谓的物理地址，虚拟地址和总线地址，至于为什么要通过这种方式操作内存，请参考下一节，MMU是什么，为什么，怎么做。

MMU是什么，为什么，怎么做

通常情况下，应用程序并不需要关心内存实际的物理地址，从应用程序的角度，“我需要的时候你就要给我，至于你是如何分配的，还有多少空闲，我不管”，MMU使这种需求成为可能。

我们知道应用程序的每一个进程都有自己的一张页表，通常0-3G为用户空间，3-4G为内核空间，每一个进程都傻傻的以为自己独自拥有4G的内存空间，从而使得程序员在写程序时不需要考虑计算机中物理内存的实际容量，但是我们真的没有这么大的内存啊，怎么办？没关系，MMU可以解决。

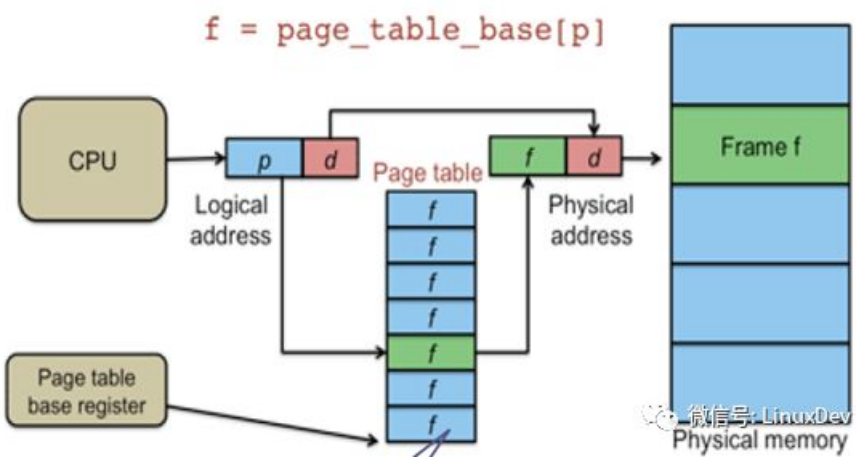
MMU提供了虚拟地址和物理地址的映射功能，这个功能使每个进程都拥有“4G独立的内存空间”成为可能。另外MMU还提供内存权限保护，用户权限保护和Cache缓存控制等功能。

我们使用C语言定义一个const变量，MMU(应该是内核，而不是MMU)会标记该变量所在的内存区间为readonly，当另外一个文件单元通过虚拟地址尝试写这个变量，MMU在把虚拟地址转换为物理地址的过程中发现，这段内存区域是readonly的，那么，不好意思，你无权写入，并产生一个fault，内核收到这个fault向应用程序发送一个SIGSEGV，应用程序产生段错误并结束。

用户权限保护，同理，MMU会标记内核空间的内存，当一个用户程序尝试访问内核空间内存，也会被拒绝。

另外，MMU还提供Cache缓存控制功能。我们知道设备可通过DMA直接访问内存，而不需要CPU；另一方面读写内存是非常耗时的（相对cache来说），如果我们的CPU存在高速缓存，把最近经常使用的内存缓冲的Cache可以大大的提高程序的效率。但是这时出现了一个问题，如何保证DMA和Cache的一致性问题？MMU提供了Cache是否命中的检查，从而进一步可以保证DMA与Cache的一致性。

那么MMU是如何实现物理地址到虚拟地址的映射的呢，请看下图：



对于一个虚拟地址，0x12345670，地址的高20位表示页表的物理地址，也就是0x12345（对应图中的p），通过这个地址找到页表所在位置，读取该位置中的数据，这个数据指向物理地址的索引。虚拟地址的低12位表示物理地址索引的偏移值，也就是0x670（对应图中的d），我们现在有了物理地址的索引值和偏移值，自然就可以找到所对应的物理内存位置。

另外MMU比较重要的一个组成部分需要介绍一下，TLB（Translation Lookaside Buffer）转换旁路缓存，顾名思义，他是一个物理地址和虚拟地址转换关系的缓存，是上图中Page table的cache，也被称为快表。

最后，附上宋老师的总结：http://mp.weixin.qq.com/s/SdsT6IsOVG84WlzcAkNCJA

内存分区和内存映射区

首先明确内存分区和内存映射区的区别，内存分区指的是实际的物理内存的分区，内存映射区指的是每一个进程所拥有的虚拟地址空间的分区情况。

对于一个运行了linux的设备，通常存在如下分区：DMAzone、Normal zone、HighMem zone。

DMAzone存在的原因是有些设备存在硬件缺陷，无法通过DMA访问全部的内存空间，为了让这些设备在需要内存的时候能够每次都申请到访问能力范围内的内存空间，内核规定了一个DMA zone，当这些设备申请内存的时候，指定分配flag为GFP_DMA，它就可以拿到DMA zone的内存。也就是说如果我们的设备不存在这样的存在缺陷的设备，我们就不需要DMA zone，或者说整个Normal zone都是DMA zone。一般我们称DMA zong + Normal zone为Low memory zone。

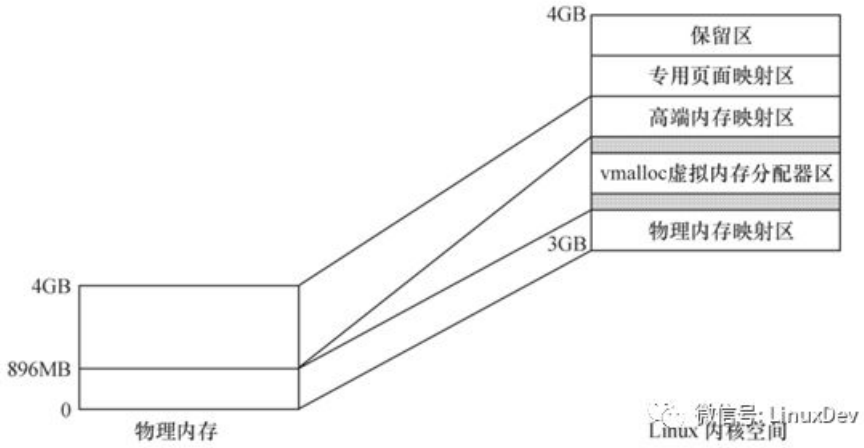
HighMemzone存在原因是，当内存较大时，内核空间的3-4G空间无法把所有物理内存地址一一映射到内核空间，只能映射一部分，那么无法映射的那部分就是高端内存。

可以一一映射到内核空间的那部分内存，除去DMA zone就是Normal zone。

内存映射区可以简单的理解为，每个进程都拥有的4G空间。其中3-4G为内核空间，这部分空间又被划分为多个区域，其中与DMA zone和Normal zone存在一一映射关系的区域是DMA+常规内存映射区或者low memory映射区，同时也专门有个区域可以映射HighMem zone，但是并不存在一一映射的关系，这个区域是高端内存映射区。

所谓的一一映射，是指虚拟地址和物理地址只是存在一个物理上的偏移。

在x86系统中，内存分区和内存映射区存在如下的关系：



在arm linux中，内存分区和内存映射区的关系请参考内核文档Documentation/arm/memory.txt

Buddy算法是个什么鬼

Linux的最底层的内存分配算法叫做Buddy算法，它以2的n次方页为单位对空闲内存进行管理。就是说不管我在应用程序中分了多大的空间，1字节，100KB，或者其它任意的大小，底层实际分配的是以2的n次方个页对齐的空间，当然并不是每一次用户空间申请内存都会引起底层的内存分配，slab算法就可以在buddy算法的基础上对内存进行二次管理，分配更小的内存空间，当然C库也可以对分配的空间二次利用，比如指定mallopt函数的第一个参数为M_TRIM_THRESHOLD，并设置真正释放内存给系统的阀值。。

Buddy算法的优点是避免了内存的外部碎片，但是长期运行后，大片的内存会比较少，而1页，2页，4页这种内存会非常多，当我们分配大片连续内存的时候就会出问题，具体解决办法请参考下一节-CMA的工作原理。

在linux系统中，我们可以通过/proc/buddy文件来查看当前系统空闲的连续内存空间剩余情况。

CMA的工作原理

应用程序中申请一块内存，在应用程序看来是连续的，因为虚拟地址本身是连续的，但实际的内存空间中，所申请的这片内存未必是连续的，不过这对应用程序来说是没关系的，因为应用程序不需要关心实际的内存情况，只要MMU把物理地址映射成虚拟地址就好了。但是如果没有MMU的情况呢，我们又需要一片连续的内存空间，比如设备通过DMA直接访问内存，这种情况下应该怎么办呢？

CMA机制就是为了解决上面提到的问题而产生的。DMA zone并不是DMA专属，其它的程序也可以申请该zone的内存，如果当设备要申请DMA zone空间的一大片连续的内存时候，已经没有连续的大片内存了，只有1页，2页，4页的这种连续的小内存。解决办法就是我们标记某一片连续区域为CMA区域，这部分区域在没有大片连续内存申请的时候只给moveable的程序使用，当大片连续内存请求来的时候，我们去这片区域，把所有moveable的小片内存移动到其它的非CMA区域，更改对应的程序的页表，然后再把空出来的CMA区域给设备，从而实现了DMA大片连续内存的分配。

CMA机制并不是单独存在的，它通常服务于DMA设备，在设备调用dma_alloc_coherent函数申请一块内存后，为了得到一片连续的内存，CMA机制被调用，它保证了申请的内存的连续性。

另外CMA区域通常被分配在高端内存。

任督二脉之内存管理第二节课总结

本文是任督二脉之内存管理课程第二节课的总结说明，由于水平有限，可能无法对宋老师所讲完全理解通透，如有错误，请及时指证。

本文从4个方面进行说明：

1、 Slab的基本原理以及它的文件接口说明

2、 kmalloc、vmalloc、malloc比较

3、 OOM是什么，为什么，怎么做

4、 FAQ：群里经常问到的，也是比较容易误解的问题

slab的基本原理以及它的文件接口说明

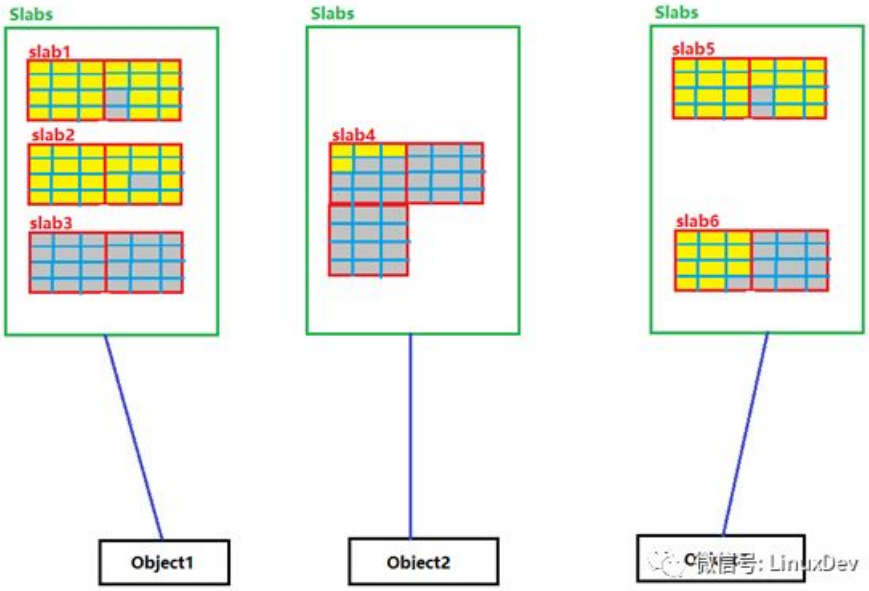
在第一节课中，我们了解到，Linux的最底层，由Buddy算法管理着所有的空闲页面，最小单位是2的0次方页，就是1页，4K，但是很多时候，我们为一个结构分配空间，也只需要几十个字节，按页分配无疑是浪费空间；另外，当我们频繁的分配和释放一个结构，我们希望在释放的时候，这部分内存不要立刻还给Buddy，而是提供一种类似C库的管理机制，在下一次在分配的时候还可以拿到同一块内存且保留着基本的数据结构。基于上面两点，Slab应运而生。总结一下，Slab主要提供以下两个功能：

A. 对从Buddy拿到的内存进行二次管理，以更小的单位进行分配和回收（注意，是回收而不是释放），防止了空间的浪费。

B. 让频繁使用的对象尽量分配在同一块内存区间并保留基本数据结构，提高程序效率。

那么，Slab是如何工作的呢？

如果某个结构被频繁的使用，内核源码就可以针对这个结构建立一个或者多个Slab分区（姑且这么叫），每一个Slab分区从Buddy拿到1页或者多页内存，并把这些内存划分为多个等分的这个结构大小的小块内存，这些Slab分区只用于分配给这个结构，通常称这个结构为object，每一次当有该object的分配请求，内核就从对应的Slab分区拿一小块内存给object，这样就实现了在同一片内存区间为频繁使用的object分配内存。请看下图



黑框表示频繁使用的结构；红框表示slab分区，一个结构内核可能为它分配一个或多个Slab；每个Slab分区有可能包含多个page，被分隔开的多个红框表示Slab分区的多个pages；蓝框表示Slab分区为对应的Object划分的一个一个的小内存块。填充黄色的框表示active的object，灰色填充的框表示未active的object，如果整个Slab分区的所有蓝框都是灰色的，表示这个Slab分区是未active的。

Linux为用户提供了Slab的文件查看接口，和命令接口。

文件接口：/proc/slabinfo

```
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount
ctive_slabs> <num_slabs> <sharedavail>
isofs_inode_cache     44     44    360   22    2 : tunables    0    0    0 : slabdata      2      2      0
ext4_groupinfo_4k    156    156    104   39    1 : tunables    0    0    0 : slabdata      4      4      0
UDPLITEv6              0      0    768   21    4 : tunables    0    0    0 : slabdata      0      0      0
UDPv6                 84     84    768   21    4 : tunables    0    0    0 : slabdata      4      4      0
tw_sock_TCPv6          0      0    192   21    1 : tunables    0    0    0 : slabdata      0      0      0
TCPv6                 44     44   1472   22    8 : tunables    0    0    0 : slabdata      2      2      0
zcache_objnode         0      0    272   30    2 : tunables    0    0    0 : slabdata      0      0      0
kcopyd_job             0      0   2344   13    8 : tunables    0    0    0 : slabdata      0      0      0
dm_uevent              0      0   2464   13    8 : tunables    0    0    0 : slabdata      0      0      0
dm_rq_clone_bio_info      0      0     88   46    1 : tunables    0    0    0 : slabdata      0      0      0
dm_rq_target_io        0      0    264   31    2 : tunables    0    0    0 : slabdata      0      0      0
bsg_cmd                0      0    288   28    2 : tunables    0    0    0 : slabdata      0      0      0
mqueue_inode_cache     28     28    576   28    4 : tunables    0    0    0 : slabdata      1      1      0
fuse_request          42     42    376   21    2 : tunables    0    0    0 : slabdata      2      2      0
fuse_inode            36     36    448   18    2 : tunables    0    0    0 : slabdata      2      2      0
ecryptfs_inode_cache      0      0    640   25    4 : tunables    0    0    0 : slabdata      0      0      0
fat_inode_cache        0      0    416   19    2 : tunables    0    0    0 : slabdata      0      0      0
```


上图所示为slabinfo文件的内容，第一行为表头：

Name：Object名字

Active_objs：已经激活的投入使用的object个数

Num_objs：为这个object分配的小内存块个数

Objsize：每一个内存块的大小

Objperslab：每一个Slab分区包含的object个数

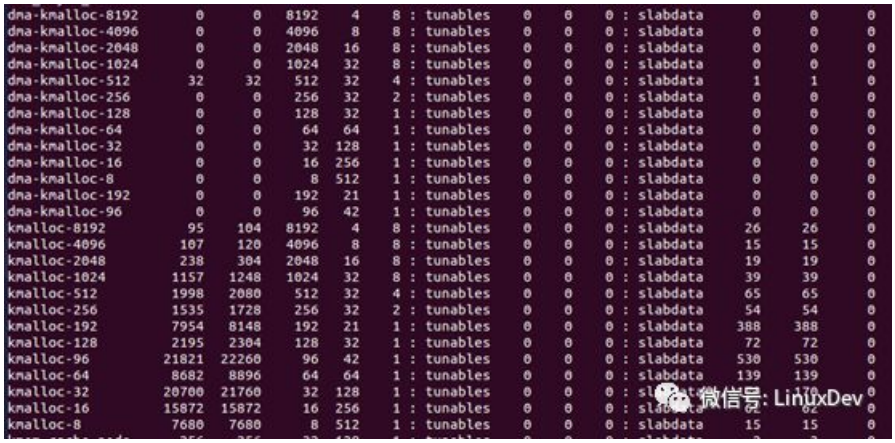
Pagesperslab：每个Slab分区包含的page的个数

Active_slabs：已经激活的投入使用的Slab分区个数

Num_slabs：为这个object分配的Slab分区个数

我在查看Slabinfo文件的时候，发现有的Num_objs为0，正常Active_objs为0是可以理解的，但是总的object数不应该为0啊，然后继续看，Active_slabs和Num_slabs也都为0，也就是说，这个时候内存还没有为这个结构分配Slab分区，一切就都解释的通了。

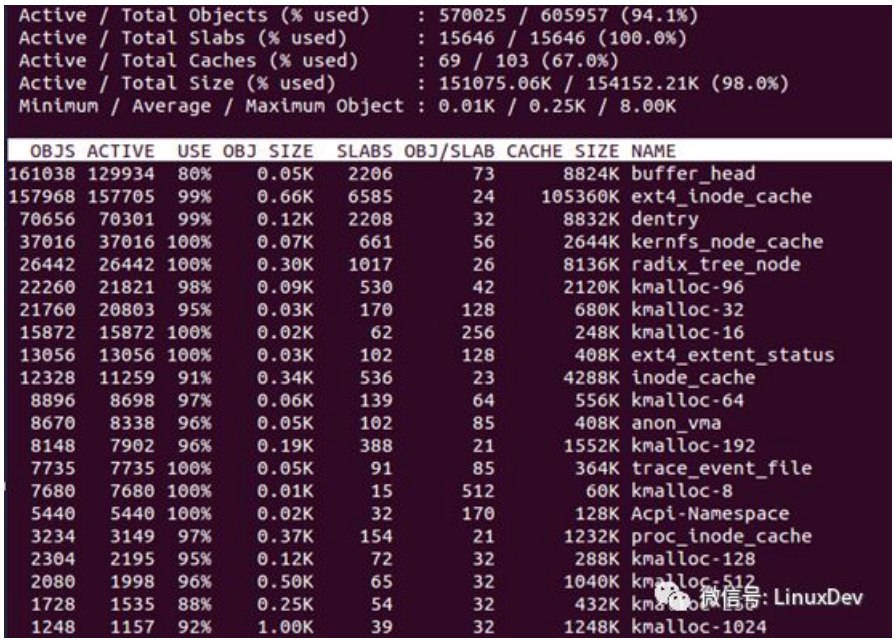
另外还有一部分slabinfo的内容是这样的：



就是说，除了经常频繁使用的结构，内核为他们分配了slabs，还同时定义了一些特定的slabs供驱动使用。

命令接口：slabtop

直接运行slabtop命令（要加sudo，上面查看slabinfo文件同样），内容如下

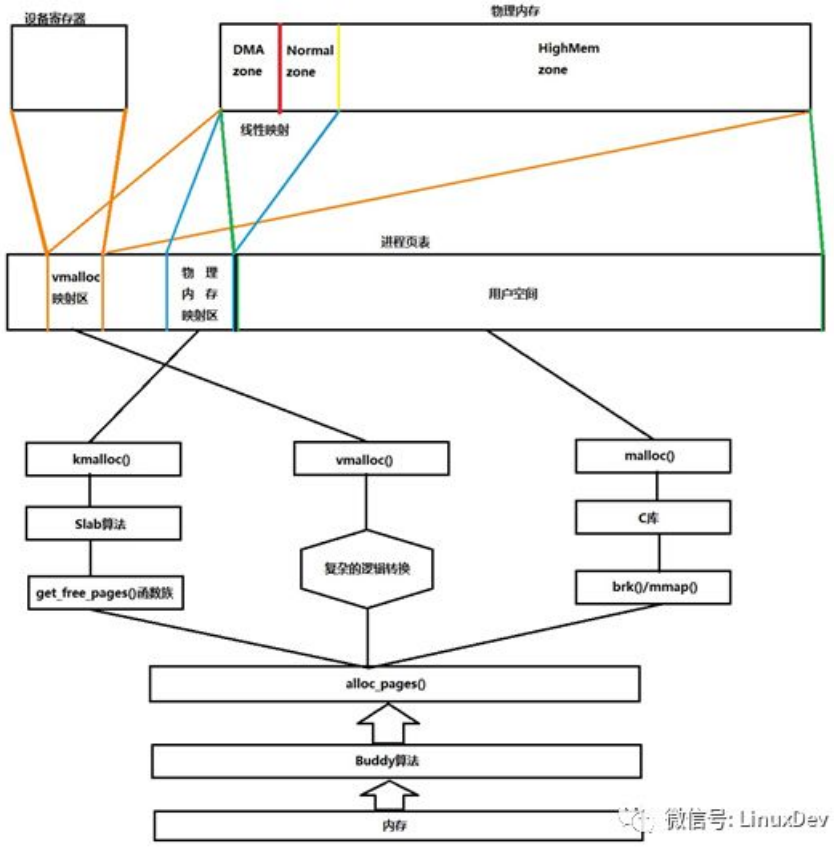


有点类似top命令，按照使用内存的多少进行排序。

最后再说一句，slab只用于分配低端内存，所分配的内存也只会被映射到物理内存映射区，所以vmalloc跟slab一毛钱关系都没有。

kmalloc、vmalloc、malloc比较

这部分内容牵连太多，也不好区分，直接上图：



(此图有误：highmem不是映射到vmalloc)

如上图所示：

如上图所示：

A. kmalloc函数是基于slab算法的，从物理内存的low mem获取内存，并线性映射到物理内存映射区（映射过程开机就已经完成了），由于是线性映射，物理地址和虚拟地址存在简单的转换关系（物理地址和虚拟地址的值只是相差了一个固定的偏移），所以使用kmalloc分配内存是十分高效的。

B. vmalloc函数分配内存的过程需要先通过alloc_pages函数获取内存（获取范围是整个内存条），然后在通过复杂的逻辑转换（注意，vmalloc并不是简单的线性映射，它获取的内存并不是连续的），把物理内存映射到vmalloc映射区。这个过程比较复杂，所以，如果只是使用vmalloc分配很小的内存空间是不合适的。

C. malloc是标准C库的函数，C库对申请的内存做二次管理，类似Slab。但是注意一点，当我们使用malloc函数申请一片内存时，实际上是从C库获取的内存，就是说，调用malloc返回后，系统未必给你一片真正的内存，分两种情况

a. C库还持有足够的内存，那么malloc就可以直接分配到C库现有的内存

b. C库没有足够的内存，malloc返回时，系统只是把要申请的内存大小的虚拟地址空间全部映射到同一块已经清零的物理内存，当我们实际要写这片内存的时候，才通过brk/mmap系统调用分配真实的物理内存，并改写进程页表。

D. 另外有一点值得一提，vmalloc映射区，除了vmalloc函数分配的内存会映射在该区域，设备的寄存器也同样会通过ioremap映射到该区域。

E. 根据上面要点C的描述，在编写实时程序时，我们可以通过下面这种方式，减少系统内存的频繁分配，而是基于C库管理的内存。

```
#include<malloc.h>

#include<sys/mman.h>
```

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


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

    if (!mlockall(MCL_CURRENT | MCL_FUTURE))//锁定进程当前和将来所有的内存
        mallopt(M_TRIM_THRESHOLD, -1UL);//设置C库释放内存的阀值为最大正整数
    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)//由于COW，确保内存被真实分配
        buffer[i] = 0;
    free(buffer);
    /* <do your RT-thing> */
    /* 接下来的所有内存分配动作都不是触发系统内存的真是分配，而是从C库获取
     *大大提高程序的效率，确保程序实时性。
         */
    while(1);
    return 0;
}
```

OOM是什么，为什么，怎么做

什么是OOM：上面提到，当我们使用malloc分配内存时，系统并没有真正的分配内存，而是采用欺骗性的手段，拖延分配内存的时机，防止无谓的内存消耗，只有在我们写入的时候，产生page fault才会拿到真实的内存，且是写多少才分配多少，那么，问题来了，当我们通过malloc获取一片内存并成功返回，然后开始逐步使用内存，系统也逐步的分配真实的内存给进程，但这个过程中，另外一个耗内存的程序快速的拿走所有的内存，导致我的进程在逐步写入的过程发现刚刚说好给我的内存现在没有了，这种情况就是OOM，out of memory!

在Linux系统，每一个进程都有一个oomscore，这个数值越高，说明进程消耗的内存越多，在发生OOM的情况下，oom score越高的进程就越有可能被系统干掉，从而缓解系统的内存压力。我们可以通过/proc/pid/oom_score文件查看进程的oom score。

那么有没有什么办法可以调整进程的oom score，就算这个进程比较耗内存，但是在OOM时候，这个进程仍然不会干掉。系统提供两个接口文件给用户：

/proc/pid/oom_ adj：可配置范围是-17到15，设置为15，oom score最大，最容易被干掉，设置-16，oom score最小，设置-17为禁止使用OOM杀死该进程。

/proc/pid/oom_score_adj：oom score会加上这个值，也可以设置负数，但如果负数的绝对值大于oom score，oom score最小为0。

FAQ

Q. kfree和free函数调用后，内存是否还给了Buddy?

A. kmalloc分配的内存是基于slab的，malloc分配的内存是基于C库的，slab和C库都会对内存进行二次管理，实际到底有没有被释放，只有Slab和C库他们自己知道。

Q. kmalloc，vmalloc，malloc他们从哪个zone申请物理内存，然后映射到那个映射区？

A. kmalloc从low mem获取物理内存，然后映射到内核空间的物理内存映射区，vmalloc和malloc都可以从整个内存条获取内存，vmalloc申请的内存映射到vmalloc映射区，malloc申请的内存映射到进程的用户空间。

写到这里，群里已经发出了第二节课问答集，其它更多内容参考该文档。

任督二脉之内存管理第三节课总结

本文是任督二脉之内存管理课程第三节课的总结说明，由于水平有限，可能无法对宋老师所讲完全理解通透，如有错误，请及时指证。

本文从7个方面进行说明：

1、 VMA到底是个什么鬼？

2、 Linux提供的VMA文件接口和命令接口说明。

3、 Page fault的产生原因分析以及与VMA的关系。

4、 物理内存、页表、进程之间的爱恨情仇。

5、 VSS、RSS、PSS、USS概念说明和实际的应用场景

6、 进程内存使用情况命令接口smem。

7、 内存泄漏的界定和监测办法。

VMA到底是个什么鬼？

VMA是Virtual MemoryAreas的缩写，虚拟内存区域，指的是用户空间0-3G范围内进程所拥有的多个全部零散分布的连续的虚拟内存空间。

注意上面这句话的三个定语：

用户空间0-3G范围内进程所拥有的：VMA区域存在于用户空间，当然“所拥有”并不是独占，也有可能是共享的。

多个全部零散分布的：进程拥有多个VMA区域，但是零散分布在0-3G空间。

连续的：这里说的连续，指的是单个VMA区域在虚拟地址空间是连续的。

看完上面的内容基本知道VMA是个什么东西了，那么VMA产生的原因是什么，为什么要搞个这个概念出来，VMA区域是客观存在的，你不定义VMA的struct，进程的代码段，数据段，堆，栈都是客观存在的，进程只要在代码段按序执行就好了，我管你叫什么名字，所以进程并不需要关心这个概念，真正需要VMA概念的是内核，通过VMA这个概念方便实现对所有进程内存空间的管理。

我们知道一个进程被fork出来，内核会维护一个taskstruct，这个结构的mmap成员维护了一个vm_area_struct的链表，这就是VMA的结构，内核通过维护这个结构来实现对进程的内存资源的管理、隔离和共享。

举个例子：进程使用malloc分配内存，这时C库没有足够的内存，内核的Lazy机制采用欺骗性手段，拖延分配内存的时机，这时，内存并没有被真正分配，内核只是把所有要分配的页表都映射到同一片已经清零的物理地址，并标记页表权限为readonly，但是当malloc返回的时候，对应的heap的VMA区域已经产生了，且已经进入内核管理的vm_area_struct链表中了，且权限被标记为读写权限，当我们向这片内存写入的时候，MMU在虚拟地址转换到物理地址的过程，发现页表的权限标记为readonly，而你要写入，这是不被允许的，于是产生Page fault，内核收到Page fault，查看对应的VMA链表，发现进程实际是有写的权限，于是分配页面，并改写页表，但是这整个过程，进程是不知道的，进程只是傻傻的以为，哈哈，老子又拿到了一片内存！

Linux提供的VMA文件接口和命令接口

Linux为用户提供了VMA查看的命令接口和文件接口。

命令接口：pmap

文件接口：/proc/pid/maps 、 /proc/pid/smaps

这些接口都可以看到进程的VMA区域分布情况，占用空间大小，和相应的权限

此处不做过多说明。

Page fault的产生原因分析以及与VMA的关系

其实上面的“VMA到底是个什么鬼”章节，已经介绍了一种Pagefault产生的原因，也说明了与VMA的关系。下面列举产生Page fault的4中原因。

A. 动态分配内存，第一次写入，由于内核的Lazy机制，页表的权限为readonly，VMA权限为r+w，MMU产生Page fault，这种情况是真实的缺页，下面还会介绍并不是真实的缺页的情况。这种缺页也叫做Minor Page fault。

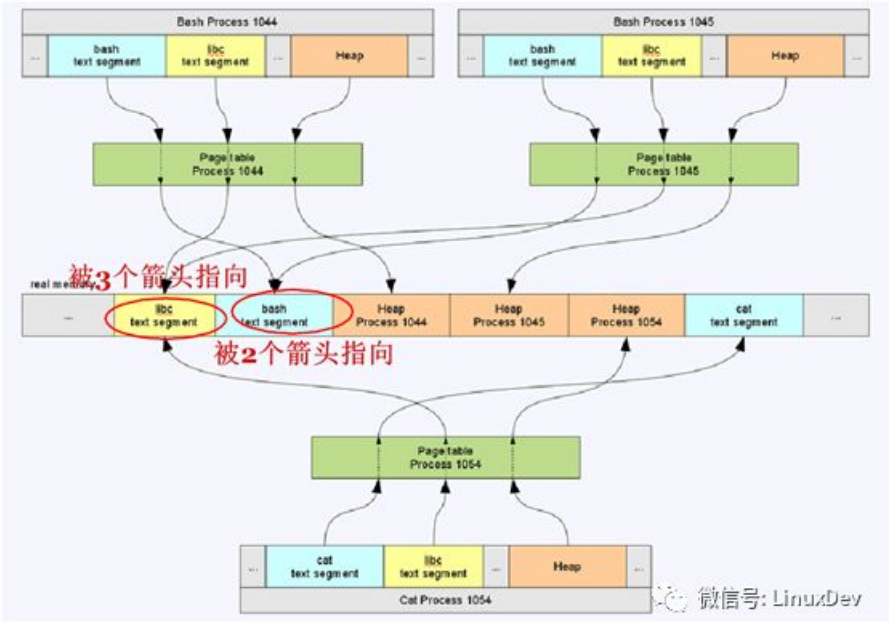
B. 进程访问自己的VMA区域以外的空间，这种行为被认为是非法的，同样会产生Page fault，但不会像A中一样引起真正的内存分配，反而会收到一个segv，程序被干掉。这种情况其实并不是真正的缺页。

C. 进程访问自己的VMA区域，但是并没有执行该操作的权限，比如进程尝试写代码段或者跳转到数据段执行，这也被认为是非法的，同样收到segv，程序被干掉。这种情况也不是真正的缺页。

D. 进程访问自己的VMA区域，且权限正确，但是对应的物理内存内容被swap到硬盘，这种情况毫无疑问才是彻彻底底的缺页，也会产生Page fault。这种缺页也叫做Major Page fault。

物理内存、页表、进程之间的爱恨情仇

此部分说起来比较复杂，直接上图：

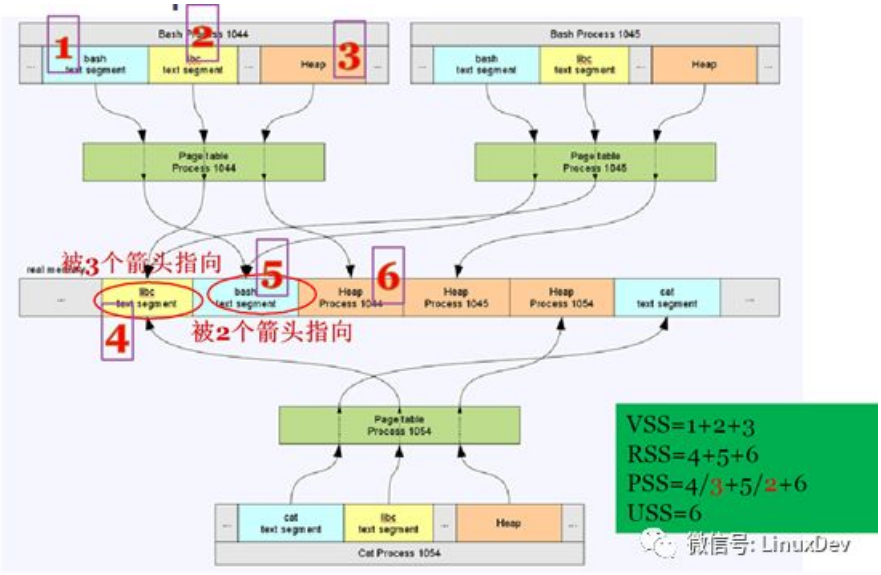


上图中，process1044，1045，1054是进程的虚拟地址空间，绿色框图是他们各自的页表，图片最中间的是实际的物理内存。进程1044，1045，1054在虚拟地址空间各自拥有多个VMA区域，但是多个进程的VMA可能通过各自页表指向同一片内存区域，比如上图中libc代码段，三个进程的libc的VMA区域都通过页表指向同一片内存，就是说这三个进程共享这段内存。当然进程的VMA通过页表指向的内存也有可能是被这个进程独占的，比如上面三个进程的堆，都是各自独占的。

举个例子：假设上图中的内存是女神，女神为了衬托自己漂亮，身边难免要有几个丑女闺蜜，丑女闺蜜就是页表，屌丝process 1044，1045，1054对女神爱慕已久，只是苦于女神高高在上，无法接近，那么怎么办，曲线救国，先接近她的闺蜜，三个屌丝通过女神的三个闺蜜都轻松的获取到女神的基本爱好，喜欢吃什么，喜欢什么颜色，三个屌丝虽然都各自获取到一份信息，但是这个信息是客观存在的只有一份（这就是上面说的共享的情况），这时，其中一个屌丝1044，辛苦加班12个月，攒钱给闺蜜买了个大金链子，这个大金链子就是屌丝1044跟女神所拥有的独家美好记忆（这就是上面说的独占的情况）。

VSS、RSS、PSS、USS概念说明和实际的应用场景

话不多说，还是上图吧：



如图所示：

VSS是进程看到的自己在虚拟内存空间所占用的内存。

RSS是进程实际真正使用的内存。

PSS是多进程共享一片内存的容量取平均数，在加上自己独占的内存。

USS是进程所独占的内存容量。

通常，VSS≥RSS>PSS>USS，VSS之所以大于等于RSS，是考虑内核的Lazy机制并没有真正的分配内存以及内存被换出等情况。

好！继续举例子：女神怀了高富帅的娃，苦于腹中胎儿一天天长大，高富帅又不值得托付，所以决定在众多屌丝中择一名形象气质佳的男士作为配偶，屌丝1044，1045，1054踊跃报名，由于三人形象上都不分伯仲，所以女神的主要考察点改为：谁更在乎自己多一点。于是：

VSS：屌丝自以为对女神的在乎程度，知道女神的爱好，还给女神买大金链子都计算在内。

RSS：每个人的感知程度不一样，屌丝纵然万般宠爱，可女神没感觉到也是白搭，这个值是女神感受到的在乎程度。

PSS：请来裁判，考察屌丝日常的在乎程度，知道女神爱好+1分，送大金链子+3分，这个分数是比较客观的。

USS：单独考量每个屌丝送多少大金链子。

所以，综上，我们知道PSS值是比较客观的值，VSS是一个虚拟的值，RSS是一个实际的值，USS是独占的值。

进程内存使用情况命令接口smem

Linux提供命令接口smem来查看系统中进程的VSS、RSS、PSS、USS值。

```
  PID User     Command                        Swap      USS      PSS      RSS
 3474 baohua   ./a.out                           0       68       76     1028
 3352 baohua   /bin/cat                          0      112      143     1896
 2750 baohua   /usr/bin/VBoxClient --seaml       0       84      217     1164
 2743 baohua   /usr/bin/VBoxClient --displ       0       84      219     1172
 2755 baohua   /usr/bin/VBoxClient --draga       0       92      219     1140
 2733 baohua   /usr/bin/VBoxClient --clipb       0      120      236     1160
 2838 baohua   upstart-dbus-bridge --daemo       0      208      271     1696
 2839 baohua   upstart-dbus-bridge --daemo       0      228      295     1784
 2787 baohua   upstart-event-bridge              0      264      315     2928
 3049 baohua   /sbin/initctl emit indicato       0      232      327     2988
 2836 baohua   upstart-file-bridge --daemo       0      268      346     1772
 2752 baohua   /usr/bin/VBoxClient --seaml       0      244      467     3296
 2746 baohua   /usr/bin/VBoxClient --displ       0      248      474     3384
 2830 baohua   gpg-agent --daemon --sh          32      528      543     1704
 2758 baohua   /usr/bin/VBoxClient --draga      32      332      563     3572
 2887 baohua   /bin/dbus-daemon --config-f       0      404      581     3320
 3054 baohua   /usr/lib/i386-linux-gnu/ind       0      524      602     5484
 3287 baohua   /usr/lib/gvfs/gvfsd-metadat       0      544      618     5580
 2892 baohua   /usr/lib/at-spi2-core/at-sp       0      540      640     5044
 2960 baohua   /usr/lib/ibus/ibus-engine-s       0      556      691     6364
```


还可以使用—pie选项和—bar选项进程图形化显示，更加一目了然。

内存泄漏的界定和监测办法

进程运行时申请的内存，在进程结束后会被全部释放。内存泄漏指的是运行的程序，随着时间的推移，占用的内存容量呈现线性增长，原因是程序中的申请和释放不成对。

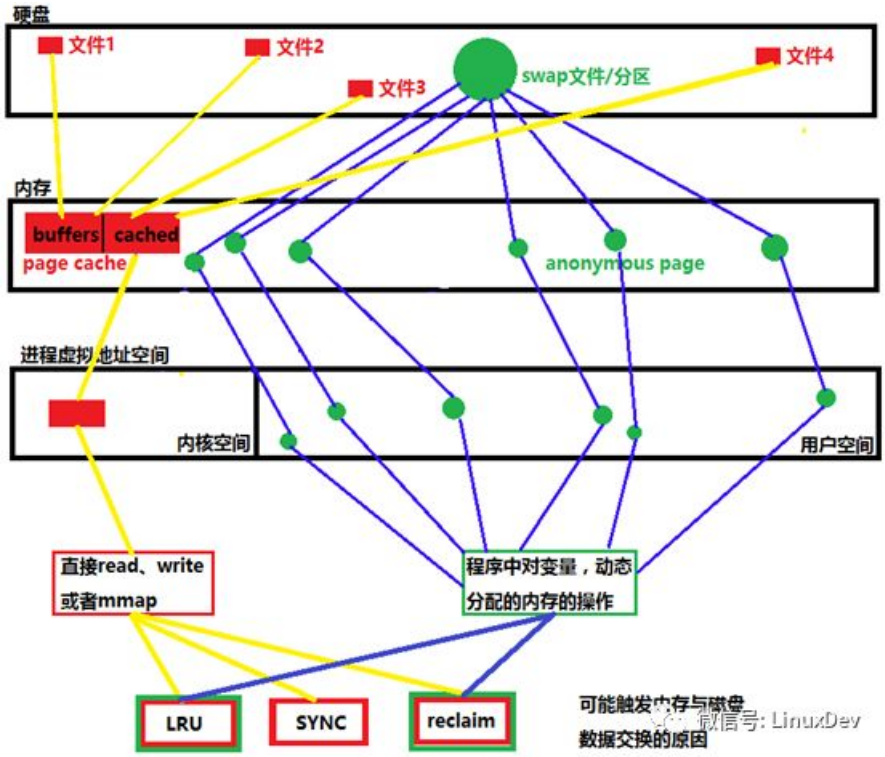
如何监测程序是否出现了内存泄漏的情况，一方面我们可以通过上文提到的smem命令，连续的在多个时间点采样，记录USS的变化情况，如果这个值在连续的很长时间里呈现出持续增长，基本就可以断定程序存在内存泄漏的情况，然后你就可以手动去程序中查找泄漏位置。

另一方面，如果程序代码量较大，不方便查找定位内存泄漏点，可以使用valgrind和addresssanitizer来查找程序的内存泄漏。两种方式各有优劣，valgrind在虚拟机中运行程序，所以程序运行效率下降；addresssanitizer则需要改动源码，在源码中包含sanitizer/lsan_interface.h文件，然后在需要检查内存泄漏的地方调用函数__lsan_do_leak_check。两种方式都可以定位内存泄漏的位置。

任督二脉之内存管理第四节课总结

本文是任督二脉之内存管理课程第四节课的总结说明，由于水平有限，可能无法对宋老师所讲完全理解通透，如有错误，请及时指证。

发了前几天的总结后，有群里的朋友@jeff表示，我这样大篇幅的文字描述，估计没几个人有耐心看下去，想想也是，内存管理本身就比较复杂，枯燥，我听了宋老师的课，了解个一知半解，转述的过程可能也不到位。所以这一次尽量使用图片进行说明，然后逐步展开。话不多说，先上图吧。



如图所示：有两条脉络，分别用黄线和蓝线标识，黄线的脉络为有文件背景的数据交换过程，蓝线为无文件背景的数据交换过程。

那么何为有文件背景的页File-backedpage，何为无文件背景的页，也就是匿名页anonymous page，直接盗用宋老师课件图片：

特别说明一下程序的代码段，程序运行的时候，实际上是把ELF文件的代码段加载到物理内存的page cache，然后映射到内核空间的page cache页，所以程序的代码段也是file backed的。

接下来分两条脉络来进行说明和扩展。

Filebacked

在硬盘中能对应到实际的文件的，归纳为file backed，文件在open后，会被加载到物理内存，我们称这片内存为page cache页，在3.14版本以前的内核，page cache又被划分为buffers和cache，3.14版本以后不做区分，全部看作page cache。Page cache被映射到内核空间的虚拟地址，用户通过两种方式访问磁盘上的文件，直接读写和mmap到用户空间，可能触发page cache页面和磁盘数据交换的有LRU，手动同步SYNC和内存回收reclaim。

上面的一段话，描述了filebacked的整体脉络，有一下几个问题需要单独说明：

A. 3.14版本以前的pagecache划分为buffers和cache，他们有什么区别？

要说明这个问题，先明确一个概念，page cache作为磁盘的一个缓冲，它缓冲过来的文件内容是可以被牺牲掉的，就是说我内存空间不足的时候，我可以回收这部分内存资源，所以在Linux操作系统看来，这部分内存虽然被占用，但是仍然是available的。

```
chenyw@LEPdocker:~/barrySong/memoryCourses/memory-courses-master/day4$ uname -r
3.13.0-24-generic
chenyw@LEPdocker:~/barrySong/memoryCourses/memory-courses-master/day4$ free
             total       used       free     shared    buffers     cached
Mem:       1003108     773212     229896       1428      21604     183656
-/+ buffers/cache:     567952     435156
Swap:      1045500     331280     714220
```


通过free命令，可以看到3.13版本分别列出了buffers和cached的数值，buffers值是直接操作裸分区的pagecache的大小，比如我们通过dd命令读写SD卡；cached值是操作文件系统上的文件的page cache的大小，比如直接open dev/sda1/hello.c文件，这个文件的page cache被叫做cached。

看上图，free命令的第三行，有一个-/+buffers/cache，这个值是如何计算的呢？

上面说过，page cache是可牺牲的，这个值就是page cache被回收后的内存的used和free的值，盗用宋老师课件again。

- Used5=1-3-4
- Free6=2+3+4

其实这种划分没有什么特别的意义，他们的区别是各自的background不同而已，所以3.14版本之后的内核不做buffers和cached的区分，free命令的-/+ buffers/cache这一行也不再需要，只是单独搞出一个available值，用于表示当前系统中包括已经使用的page cache（可牺牲），到底有多少可利用的内存。

```
yocto@yocto-pc:~$ uname -r
4.2.0-27-generic
yocto@yocto-pc:~$ free
             total       used       free     shared    buffers     cached
Mem:       2062592    1996248      66344      17968     350780     942772
-/+ buffers/cache:     702696    1359896
Swap:      1046524      62516     984008
```


原谅我的free还不够给力，并没有列出available值。

B. mmap和直接读写有什么区别？

Pagecache被映射到内核空间后，用户想要操作对应的文件，实际上是对内存里page cache的操作，系统会在合适的时机回写到磁盘中对应的文件。操作的方式有两种，直接读写，和通过mmap把page cache在映射到用户空间一份。

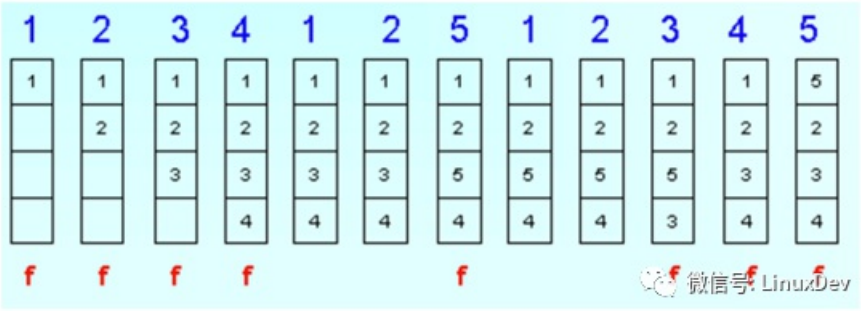
程序调用read、write函数的时候，陷入到内核空间，实际调用的是file operation结构的read、write接口，这两个接口必然要做的一件事就是copy_from_user和copy_to_user，我们知道这两个函数是会引起内核空间与用户空间的数据拷贝的，就是说每一次read和write都要进行一次拷贝。

mmap则完全不同，它把pagecache直接映射到用户空间，现在用户空间和内核空间的页表都可以对应到page cache物理内存。然后通过mmap返回的指针来读写page cache，这个过程是没有用户空间和内核空间的内存拷贝的。

通常在操作显存设备的时候会使用mmap，比如我可以通过mmap把/dev/fb0映射到用户空间，通过读写mmap返回的指针来实现对屏幕的显示。

C. LRU是个什么鬼？

LUR，LeastRecently Used，翻译过来就是最近最少使用，顾名思义，内核把最近最少使用的page cache内容或者anonymous页面交换出去。上图，盗图three



现在cache的大小是4页，前四次，1，2，3，4文件被一次使用，注意第七次，5文件被使用，系统评估最近最少被使用的文件是3，那么不好意思，3被swap出去，5加载进来，依次类推。

所以LRU可能会触发pagecache或者anonymous页与对应文件的数据交换。

D. SYNC指的什么，如何触发pagecache与文件的数据交换？

系统为用户提供了，数据从pagecache回写到文件的接口，sync命令。只要运行sync命令就可以触发page cache与文件的数据交换。

另外，我们还可以通过向/proc/sys/vm/drop_caches写入数字来清空对应的caches，一般写入之前，最好执行一下sync命令来同步一下，以便释放更多的空间，因为drop_caches只回收cleanpages，不回收dirtypages，所以如果想回收更多的cache，应该在drop_caches之前先执行"sync"命令，把dirtypages变成cleanpages。

```
echo 1 > /proc/sys/vm/drop_caches   //清空 pagecache

echo 2 > /proc/sys/vm/drop_caches   //清空 dentries 和 inodes

echo 3 > /proc/sys/vm/drop_caches   //清空所有缓存（pagecache、dentries 和 inodes）
```

anonymous

在进程中定义的堆，栈等没有文件背景的页面，如果系统没有创建swap分区或者swap文件，那不好意思，匿名页只能常驻内存，直到程序退出，或者发生OOM程序被干掉。与file backed不同的是，anonymous本身就在内存中，而file backed是磁盘中的文件，为了提高效率把内存作为磁盘的缓冲区，就是page cache，page cache可以往对应文件交换，anonymous页如果过大的话，可以往swap分区或者创建的swap文件中交换。要操作这些匿名页，直接在程序源码中操作变量和动态分配内存的指针就好了。

那么，对于anonymous页，在什么情况下会触发数据交换呢？

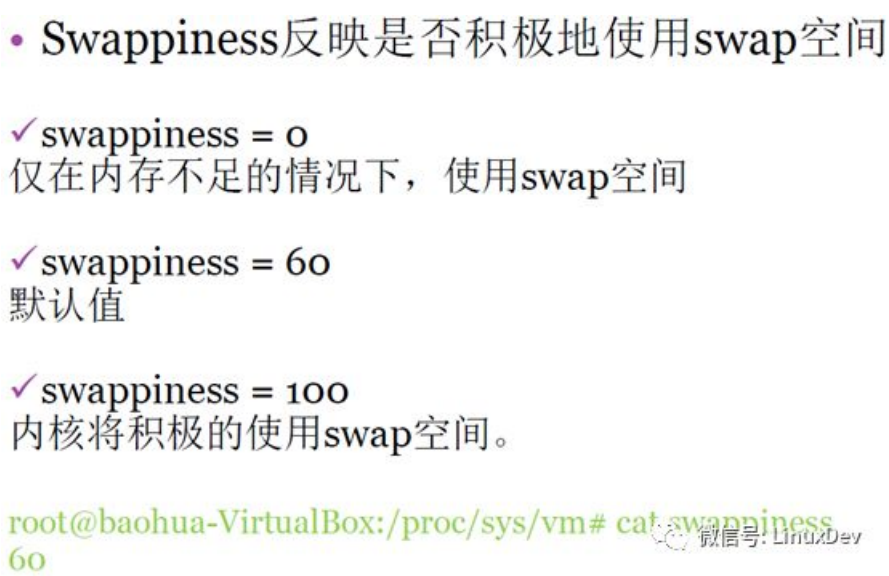
除了LRU会产生匿名页的交换，内存回收也会引发数据交换，reclaim，Linux有一个后台进程kswapd，负责回收page cache和匿名页，回收速度较慢，但是不会影响程序运行，程序不会被delay，当系统内存资源异常紧张时，会触发Direct reclaim，这个过程回收速度较快，但是进程会被直接delay，直到回收够足够的内存。

至于何时kswapd开始回收内存，何时Directreclaim，有三个门限值，min，low，hight，当内存的水位达到low，说明内存紧张，这时kswapd开始工作，慢慢回收内存，直到水位达到high，当系统内存异常紧张时，达到min水位，Direct reclaim被触发。

Swappiness

当系统内存不足时，可以从filebacked pages或者anonymous pages回收内存，不论哪个被回收，再次被加载进内存一定都会影响程序的效率。具体从哪里回收，Linux提供swappiness值作为衡量标准。

Swappiness越大，**越倾向于回收匿名页；swappiness越小，越倾向于回收file-backed的页面。**当然，它们的回收方法都是一样的LRU算法。盗图four。



顺便附上一个参考链接：http://mp.weixin.qq.com/s/BixMlSiPz3sR9FDNfVSJ6w

zRamswap

对于嵌入式设备，它的磁盘是SD卡，MMC，一方面速度较慢，另一方面，有使用寿命的问题，不太适合做swap分区。嵌入式设备一般会从内存中拿出一小部分当作虚拟内存，这个就是zRam。但是这样直接用又没有什么意义，因为虚拟内存的目的就是在内存不足时“扩展内存”，现在内存还是那片内存就是换了个说法，所以为了“扩展内存”，当系统把内存交换到这个虚拟内存，通常是以压缩的方式存储，当swap in时在解压。这样就某种程度的“扩展了内存”，但是缺点是增加了CPU的压力，需要进行压缩和解压缩。

任督二脉之内存管理第五节课总结

本文是任督二脉之内存管理课程第五节课的总结说明，由于水平有限，可能无法对宋老师所讲完全理解通透，如有错误，请及时指证。

第五节课的内容多且杂，其实完全可以合并到前四节课中。但考虑前四篇总结已经完成，章节插入不方便，所以还是多写一篇。

本文分成两部分来论述

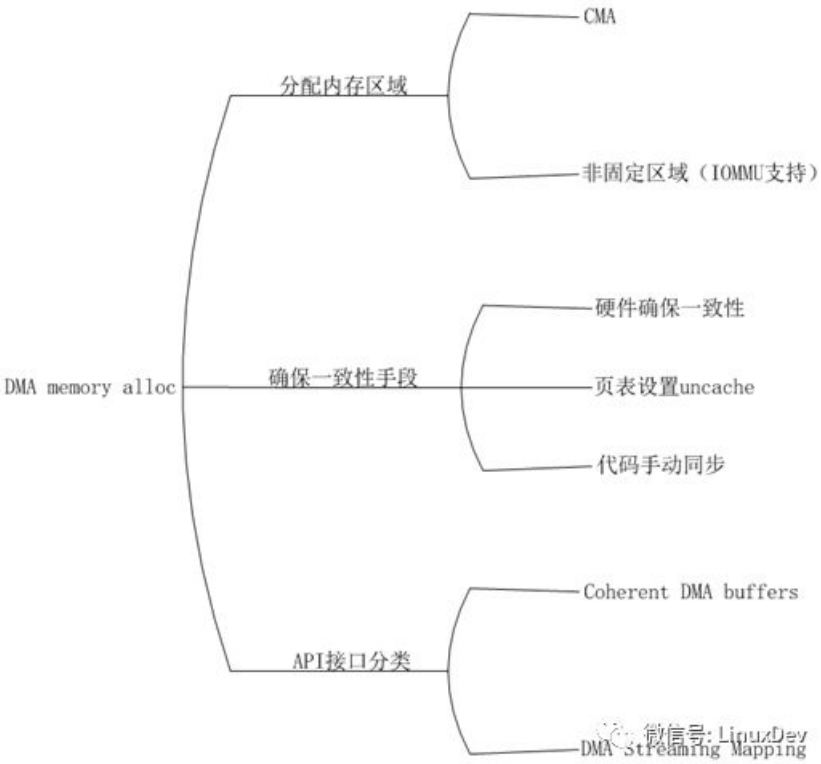
1、 DMA与Cache一致性问题。

2、 常用的命令接口和文件接口简要说明。

DMA与Cache一致性问题

关于这一部分，宋老师的文章已经讲解的非常细致，我在写也无非是画蛇添足，所以此处只做简单总结。附上文章连接，http://mp.weixin.qq.com/s/5K7rlPXo2yIcoIXXgqqLfQ

而实际上，如果你不是在IC公司，大部分时候你只需要在驱动程序中轻松敲下dma_alloc_coherent来获取一片能确保DMA和cache一致性的内存就可以了，具体实现细节对你来说可能并不重要。请看下图：



A．DMA的内存分配区域

不论是应用程序还是驱动程序，在获取内存时都需要获得一片连续地址的内存，但是由

于DMA设备访问内存不经过MMU，所以也无法把不连续的物理地址映射为连续的虚拟地址，解决这个问题有两种方式：

a. 在CMA区域申请DMA内存，因为CMA本身是一片连续的物理内存，CMA通常被分配在高端内存，这个时候这片内存会被映射到vmalloc映射区，如果CMA在低端内存，则不需要重新映射，因为低端内存在开机时已经与low Memory映射区建立了一一映射关系。

b. 如果设备存在IOMMU，那么做DMA内存分配时则不需要关心具体的内存分配区域，IOMMU会让设备看到一片连续的地址范围，它的功能类似MMU，只不过MMU是把物理地址转换为连续的虚拟地址供CPU使用，而IOMMU是把物理地址转换为连续的总线地址供设备使用。

B．确保DMA和cache一致性的手段

确保DMA和cache一致性的手段有以下三种：

a. 页表设置uncache

要保证DMA和cache一致性最简单办法，在申请到内存后，修改对应的页表，将页表的cache属性改为uncache，这样，当CPU在访问该片内存时就不会从cache取数据。

b. 硬件确保一致性

有的设备提供了确保一致性的硬件机制，这时我们申请内存后则不需要修改页表的cache属性，一致性由硬件来保证。

c. 代码手动同步

如果对应的内存区域已经申请好了，设备直接使用，那么驱动就无法更改对应页表的cache属性，这时解决一致性的手段时在每次访问这篇内存前手动同步cache内容到内存，同时禁止CPU对这片内存的访问，直到设备访问完成。实际上下面提到的DMA streaming mapping就是通过这种方式实现的。

C. API接口

当我们的驱动自己获取内存，可以使用一致性DMA缓冲区API接口，就是

dam_alloc_coherent()；如果对应的内存已经被成功分配，我们在使用前需要调用DMA流映射API接口，确保cache的内容被成功flush到内存，对应的函数有dma_map_sg()和dma_map_single()，他们两个的区别是，sg映射的内存是分散/聚集的，分散在不同的位置，single映射的内存是连续的一片内存，通常是CMA。

常用的命令接口和文件接口简要说明

A. 文件Dirty数据写回配置接口

/proc/sys/vm/dirty_expire_centisecs：设置Dirty数据的写回时间期限，超过这个时间，在flusher线程下次唤醒后，写回这部分数据，单位是百分之一秒，厘秒。

/proc/sys/vm/dirty_writeback_centisecs：flusher线程周期性唤醒的时间，单位是厘秒，设置为0，表示禁止定期写回。Flusher线程唤醒后会把超过期限的脏页和进程超过dirty_background_ratio值的脏页写回。

/proc/sys/vm/dirty_background_ratio：进程持有的脏页的个数阀值，单位是页，超过这个值，flusher线程在下次唤醒后会对脏页进行写回。

/proc/sys/vm/dirty_ ratio：进程持有的脏页的个数阀值，单位是页，超过这个值，进程delay，无法在进行任何的写操作，并且进程自行完成脏页的回写。

B. Memory Cgroup的使用

控制group的最大使用内存示例如下：

```
$:cd /sys/fs/cgroup/memory

$:mkdir A                                                    //创建一个分组

$:cd A/

$echo $((200*1024*1024)) >memory.limit_in_bytes    //设置该组最大可使用内存200M

$:cgexec  -g memory:A ./a.out                              //将a.out添加到组A并执行
```

C. 内存回收接口说明

在内存管理四章节中提到内存回收有三个水位，min，low和hight，当内存的水位达到low，说明内存紧张，这时kswapd开始工作，在后台慢慢回收内存，直到水位达到high，当系统内存异常紧张时，达到min水位，程序被堵住，Direct reclaim被触发。具体相关接口如下：

/proc/zoneinfo //该文件可以查看到各个zone的情况，包括各个zone的三个水位设置

/proc/sys/vm/min_free_kbytes //可用于查看和设置min水位的值

其中low水位和high水位没有对应的设置接口，是通过计算得来的。

Low = min*5/4

High = min*6/4

各个zone的水位标准是按总的水位标准等比例划分的，比如normal zone是800M，内存一共1G，min_free_kbytes被设置为30720，即30M，那么，normal zone的min水位就等于，800/1024 * 30720，就是24000

D. Swappiness接口

Swappiness越大，**越倾向于回收匿名页；swappiness越小，越倾向于回收file-backed的页面**。当然，它们的回收方法都是一样的LRU算法。Swappiness的接口有两个，一个是cgroup里的swappiness，一个是/proc/sys/vm下的swappiness，他们的区别是作用域不同，cgroup里的swappiness只控制组内程序的回收倾向，而/proc/sys/vm/swappiness控制当前整个系统，除了在cgroup被重新定义的程序。

E. Getdelays工具

要使用该工具需要打开内核选项CONFIG_TASK_DELAY_ACCT和CONFIG_TASKSTATS。该工具的源文件在LinuxKernelSource/Documentation/accounting下。使用getdelays可以查看当前系统或者某个进程调度的延时，IO的delay情况，swap和内存回收的delay情况，帮助用户查看程序的耗时情况。

F. Vmstat命令

该命令可周期性的查看swap in/out和block in/out的情况。
