Linux 内存管理知识总结

2013.3 Version 1.00
yalung929@gmail.com

1 基本概念

1.1 Page

page是内存管理的基本单位。类似于磁盘的block。一个page在x86上通常是4KB。每一个page都有一个描述符——struct page描述它。主要记录一些使用信息。在我分析的内核(2.6.32 x86_64)上它的大小是56字节。这就意味着页描述符占去总内存的1%多。

1.2 地址空间和地址映射

此处省略XX字。详情请参考本人博文《Linux进程管理知识总结》1.6小节。

1.2.1 内核空间的地址映射

在发生进程切换时，CR3会改变，被放入将要运行进程的页表基址。在发生用户态到内核态的切换之后，CR3寄存器并不需要改变。这是因为每一个进程的页表中，以内核空间起始地址0xffff880000000000[1]开始的逻辑地址都被映射至相同的物理地址。*详细的来说，每一个进程的一级页表PGD上，以0xffff880000000000对应索引位置开始的内容都是一样的。这样经MMU计算的结果就是一样的。*

然后为了方便管理，0xffff880000000000开始的地址连续的映射至整个物理内存。即0xffff880000000000 + x 对应物理地址x。这样内核可以通过简单的宏计算在逻辑地址和物理地址之间转换。

```
42 #define __va(x)          ((void *)((unsigned long)(x)+PAGE_OFFSET)) //物理地址到逻辑地址
"arch/x86/include/asm/page.h" 64 lines --7%--
```

[1] 这个值，在不同内核版本、不同体系结构上可能不一样。

1.3 Zone

X86_64上物理内存被分为三个Zone，DMA、DMA32、NORMAL。

[5439462.513669] Node 0 **DMA**: 0*4kB 0*8kB 0*16kB 1*32kB 2*64kB 1*128kB 1*256kB 0*512kB 1*1024kB 1*2048kB 3*4096kB = 15904kB

[5439462.513679] Node 0 **DMA32**: 1164*4kB 1266*8kB 1072*16kB 742*32kB 357*64kB 125*128kB 26*256kB 8*512kB 0*1024kB 0*2048kB 0*4096kB = 105280kB

[5439462.513688] Node 0 **Normal**: 1022*4kB 208*8kB 157*16kB 46*32kB 12*64kB 0*128kB 0*256kB 0*512kB 0*1024kB 0*2048kB 0*4096kB = 10504kB

DMA：物理内存0~16M的部分，用于一些老外设的DMA映射。

DMA32：物理内存16M ~ 4G的部分，用于外设DMA映射。

NORMAL：物理内存高于4G的部分，通用内存。

1.4 Node

在NUMA架构下，物理内存首先按照NUMA node进行组织。每个node上再做zone划分：

[344847.674642] **Node 0 DMA**: 1*4kB 2*8kB 1*16kB 2*32kB 1*64kB 1*128kB 0*256kB 0*512kB 1*1024kB 1*2048kB 3*4096kB = 15652kB

[344847.674654] **Node 0 DMA32**: 6*4kB 6*8kB 67*16kB 4*32kB 5*64kB 5*128kB 5*256kB 5*512kB 6*1024kB 5*2048kB 124*4096kB = 530360kB

[344847.674666] **Node 0 Normal**: 295*4kB 411*8kB 476*16kB 251*32kB 122*64kB 27*128kB 572*256kB 7611*512kB 5853*1024kB 3989*2048kB 9634*4096kB = 57698452kB

[344847.674679] **Node 1 Normal**: 24722*4kB 904*8kB 517*16kB 61*32kB 19*64kB 5*128kB 18*256kB 2*512kB 3*1024kB 134*2048kB 127*4096kB = 921528kB

但可以看到node 0上有DMA和DMA32，其他node上只有NORMAL。

1.5 伙伴算法（buddy system）

每个Node上的每个Zone上的空闲物理页使用伙伴算法管理。一共11个链表，分别存放大小为2^0（4K）~ 2^{10}（4096K）个连续页面。最开始，系统初始化的时候，大块连续的内存很多，随着系统运行，内存会变的非常零散，一些大的申请请求可能就无法满足，所以就需要把空闲的地址连续的小块内存合并。比如两个4K的会被合并为8K放入8K的链表。如果8K的还可以合并，就继续合并为16K，一直到最终的4096K。伙伴算法是为了解决外部碎片。

无论内核态还是用户态，物理内存的使用最终都是通过buddy system获得。alloc_page/alloc_pages_node是buddy system对外的接口。

1.5.1 关键实现细节

假设1 2 3 4一共4个块，这4个块地址依次相邻，且大小相同。1和2互为伙伴、3和4互为伙伴。不能出现2和3互为伙伴的情况，虽然他们的地址相邻，因为一旦2和3合并了，1和4就没法合并了，这四个块将不能恢复为原来的大块。下面的函数就是**伙伴**算法的奥义所在。

```
410 /*
411  * Locate the struct page for both the matching buddy in our
412  * pair (buddy1) and the combined O(n+1) page they form (page).
413  *
414  * 1) Any buddy B1 will have an order O twin B2 which satisfies
415  * the following equation:
416  *      B2 = B1 ^ (1 << O)
417  * For example, if the starting buddy (buddy2) is #8 its order
418  * 1 buddy is #10:
419  *      B2 = 8 ^ (1 << 1) = 8 ^ 2 = 10
420  *
421  * 2) Any buddy B will have an order O+1 parent P which
422  * satisfies the following equation:
423  *      P = B & ~(1 << O)
424  *
425  * Assumption: *_mem_map is contiguous at least up to MAX_ORDER
426  */
427 static inline struct page *
428 __page_find_buddy(struct page *page, unsigned long page_idx, unsigned int order)
429 {
430          unsigned long buddy_idx = page_idx ^ (1 << order);
431
432          return page + (buddy_idx - page_idx);
435 static inline unsigned long
436 __find_combined_index(unsigned long page_idx, unsigned int order)
437 {
438          return (page_idx & ~(1 << order));
439 }
"mm/page_alloc.c" 5447 lines --7%--
```

所有的page描述符按照其所描述内存页的地址顺序地放在一个数组中。

第一个公式通过异或操作让下标（index）：

0和1，2和3……互为伙伴，对于4KB而言。1和2不会成为伙伴。

0和2，4和6……互为伙伴，对于8KB而言。2和4不会成为伙伴。

第二个公式通过与操作直接获得合并为上一级大小的块后的下标。

1.6 Slab（Slab allocator）

Slab是内核提供的供内核各个子系统使用的一个对象内存池。其主要作用有两个：提升内存分配性能和减少内部碎片。

内核中维护了大量的对象（结构体），比如task_struct、inode、dentry……，这些对象频繁的建立和释放对系统性能造成很大的影响。slab为每一个对象提供一个缓冲池，初始化时，申请一大块连续的内存，然后零售给使用者。释放的时候，先放到池子里面，暂不交还给buddy system，这样下次申请，就可以很快的获得。

通过slabtop命令或/proc/slabinfo文件可以查看系统中的slab信息：

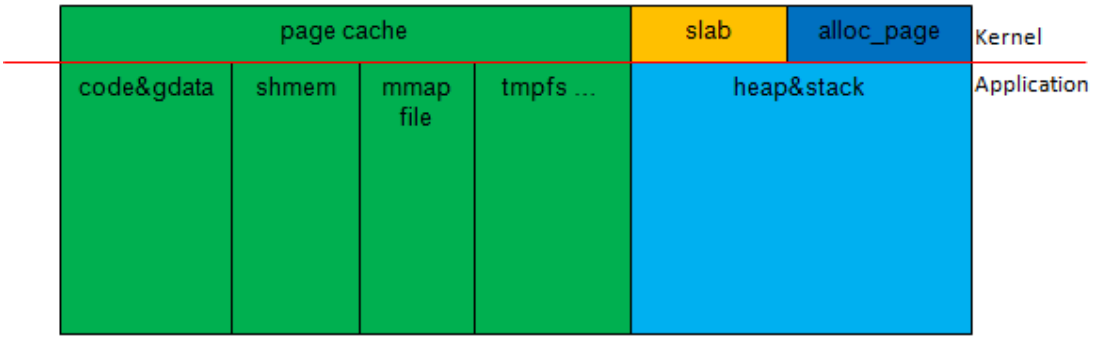
```
Active / Total Objects (% used)    : 337009 / 370755 (90.9%)
Active / Total Slabs (% used)      : 26739 / 26741 (100.0%)
Active / Total Caches (% used)     : 90 / 175 (51.4%)
Active / Total Size (% used)       : 103794.30K / 109417.20K (94.9%)
Minimum / Average / Maximum Object : 0.02K / 0.29K / 4096.00K
 OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
123506 108570  87%    0.10K   3338       37     13352K buffer_head
64355  64249  99%    0.80K  12871        5     51484K ext3_inode_cache
54260  46221  85%    0.19K   2713       20     10852K dentry
41654  41205  98%    0.06K    706       59      2824K size-64
18697  17360  92%    0.55K   2671        7     10684K radix_tree_node
14520  14290  98%    0.12K    484       30      1936K size-128
```

Slab的代码定义位置如下，三个文件分别表示三种不同的内部实现：

```
# ls mm/sl*b.c
mm/slab.c  mm/slob.c  mm/slub.c
```

2 内存的使用分类和统计

内存去哪里了？谁用了？我们经常遇到这样问题。有必要把系统对内存使用的分类搞清楚。



首先按照责任者内存分为两大类：

凡是由应用程序（用户态代码）申请和释放的称作应用内存。

凡是由内核（内核态代码）申请和释放的称作内核内存。

上图红线之上是内核内存，红线之下是应用内存。

内核内存中包括了slab、page cache，以及其他直接从buddy system获取的内存——例如DMA什么的。其中slab可以通过/proc/slabinfo以及/proc/meminfo的Slab字段查看。

应用内存通过top[2]命令显示的RES字段统计。应用内存中的heap&stack进程私有。而code&gdata（global data）、通过shmget获取的共享内存、mmap至文件的内存，**可能被共享**，在系统内存中只会有一份，这部分统计到top的SHR字段。

比较特别的是，tmpfs等类似的内存文件系统，文件直接存于内存中。虽然不统计在进程的内存使用上，但我们依然把它归属于应用内存。因为这部分内存，内核不负责释放，由应用程序调用文件系统接口释放。

2.1 Page Cache

不得不重点说一下page cache，大家注意到，它跨越了红线。就是因为这个东西，才让内存使用的分类和统计变得复杂，难以统一。

page cache本身非常简单，一句话可以概括，page cache就是**所有映射至文件的物理内存的总和，无论这个文件是在磁盘上、还是在内存中；无论有实际意义的文件、还是只是底层实现封装的抽象的虚拟文件**。正是这种简单的概括，让它包括了七七八八的内存，有些并不具备缓存属性，并不能由内核自动释放。比如共享内存（shmem）和tmpfs。这给内存占用率的正确统计造成了相当的麻烦。我们不能简单的认为free命令显示的free+cached之和就都是**可用**的。显然**应用内存中属于page cache的部分是不可用的。**比较好的是，我所分析的2.6.32内核上，通过/proc/meminfo文件的Shmem字段已经可以获取到shmem和tmpfs等内存文件系统的统计值，使用它可以减少统计误差[3]。

```
cat /proc/meminfo |grep Shmem
Shmem:                 6604 kB
```

[2] top命令内存统计字段的含义参见本人博文《剖析top命令显示的VIRT RES SHR值》。

[3] 详情参考本人博文《Linux可用内存统计方法》。

2.2 Shmem

《Linux进程管理知识总结》中谈到了Shmem的实现，实际上就是在tmpfs这种文件系统上创建一个文件，然后mmap之。内核封装了这个过程，提供shmget/shmat接口，不再赘述。但要知道，这是共享内存被统计到Cached字段的原因——谁让你和文件沾上边！

2.3 top 的 SHR 字段

再看top会为什么把共享内存、代码段、数据段、mmap file都算作SHR，其实无非是这些内存都有一个文件与之对应，而文件是系统全局唯一的，系统中只需要一份物理内存与之对应，不需要重复分配，都是**共享**的，谁用谁就把自己的地址空间（虚拟内存）映射过去好了。 但SHR中的内存有可能只有自己在使用，只是以后别人也用时，可以共享。如果想精确统计一个进程的内存中共享和不共享的内存数量，可以使用本人博文《剖析top命令显示的VIRT RES SHR值》中提到的解析/proc/pid/smaps文件的方法。

3 内存的回收

可以回收的内存包括**所有的应用内存、 所有的page cache、 以及slab中未被使用的部分。**

slab的回收由slab系统单独管理。

3.1 LRU List

其它的可回收内存，都被放到两个LRU链表内管理——Inactive和Active。通过/proc/meminfo文件可以看到这两个统计值。在需要回收的时候，会从inactive链表上回收。回收的时候，如果映射的是磁盘上的文件就flush，然后直接丢掉内存；如果不是就写到swap分区（交换）。

3.1.1 实现

内核并没有一个进程负责周期性的扫描LRU链表，把它们转移到active或inactive。那么如何确定一个page长时间没被访问，而需要进入inactive呢？这个时间又是多长呢？

对于第一个问题，依靠CPU来解决。每个page对应的页表项（最后一级页表项）——pte的低12位都是0，所以可用做特殊用途。其中一个bit就表示Accessed——访问标记。每次CPU

访问这一个页，都会导致这个bit被设置。

对于第二个问题，由于没有周期性的动作，所以这个时间是不确定的。内核是这样处理的：

假设一开始page被放入active中：Accessed=1。

如果内存充足，回收算法永远不会被执行。这些page一直是这个状态。

在内存不足而触发回收算法执行的时候，它会从inative队列上回收内存。如果inative上不够或者inactive相对active比较少，就会从active队列上摘一些下来，检查它们的Accessed标记，如果有，就设置为0，并把它再放入active的尾部。如果没有Accessed，就认为很久没访问了，加入inactive队列。

所以所谓一个page很长时间不被访问就inative了，这个时间在内存充足时可能是永远，在内存不充足时，就是两次从active**队列**尾部游走到头部的时间。

如果刚进入inactive后，内存突然充足了，回收算法永远不会被执行。进入到inactive的page一直是inactive状态。即使它刚刚被访问，并且Accessed已经是1。

时机还是等到，内存不足触发回收的时候，内核从inactive的头部开始回收内存，如果发现page的Accessed是1，就把它插入active的尾部。

3.1.2 PG_referenced

这个标记保存在页描述符struct page中，由内核自己维护。这个标记就是记录page是否被访问的，不是有pte中的Accessed吗？这是因为很多情况下对一个page的访问是通过内核代码操作的。比如你去读取一个文件的内容，其对应的page就会被内核代码访问，这时候不需要CPU设置Accessed标记，它马上就可以设置PG_referenced标记（mark_page_accessed）。如果这个标记被设置了两次，就会从inactive进入active。即使没有触发回收算法。

3.2 kswapd

```
# ps -elf |grep kswapd
1 S root      239     2  0  80   0 -     0 kswapd Feb22 ?        00:00:00 [kswapd0]
1 S root      240     2  0  80   0 -     0 kswapd Feb22 ?        00:00:00 [kswapd1]
```

kswapd是一个内核线程，每一个NUMA node对应一个。在内存不足的时候，除了执行上面的过程获取当次申请所需要的内存之外，还会唤醒对应的kswapd线程继续执行回收算法回收更多的内存。

3.3 多 LRU list

每一个Zone上都有一个inactive和active队列，回收是以Zone为单位进行。inactive和active队列又都划分为file和anon两类，file指映射至（磁盘）文件的内存，anon指没有映射至（磁盘）文件的**匿名内存**。这种划分可以进行一些精细化控制。比如优先回收file，因为这些内存如果不是脏的就可以直接丢掉，而不需要交换到swap分区，其回收的开销远小于anon[4]。

3.4 vm.min_free_kbytes

vm.min_free_kbytes这个内核参数，在启动时候自动内核根据内存总数计算得来。一旦内存低于这个数值[5]就会触发回收，而不是在内存完全耗尽的时候。这主要是保证系统紧急情况下仍然可以获取到内存执行一些紧急任务。

有些人会手工把这个值调整的很大，这完全没必要。你调整到1G就等于内存少于1G的时候就开始回收、就可能交换了。而这1G内存的大部分，内核都用不到，应用也无法使用到。浪费了。

3.5 OOM-killer

如果回收失败，alloc_page无法获取到内存，就会执行OOM-killer，全称是out of memory killer，它会杀死进程，释放内存。通常我们不期望这种情况的发生。如果发生了，那么要么是内存严重泄漏，要么是业务容量规划有问题，应该排查解决。实际上swap也应该避免。

3.6 代码简要说明

内存回收的代码定义在"mm/vmscan.c"内。入口函数是和try_to_free_pages和balance_pgdat。try_to_free_pages由alloc_page触发。balance_pgdat由kswapd调用。二者最终都是执行shrink_slab回收slab以及shrink_zone回收LRU list上的内存。

---

[4] 早期内核中不支持这种区分，可能引入一些问题。参见本人博文《SUSE10（2.6.16）上的内存回收和脏数据回写问题》。

[5] 实际上这个值，会被按比例分摊到不同的zone。

4 附录

4.1 内核对应用程序暴露的内存使用接口

4.1.1 mmap/ munmap

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

mmap和munmap分别用于申请和释放一段连续的地址空间。如果fd是一个文件句柄，其所指向的文件内容会被映射至这一段地址空间上。然后访问这一段地址空间，就是访问对应的文件内容。实际上是文件对应的page cache用户态化了。这样就避免了read和write方式访问文件时，从用户buffer到内核态page cache拷贝的过程，提高文件IO的性能。

但这种时候一定要注意文件的保护，一旦一个文件被其他进程截断，本进程mmap的地址空间对应的物理内容就不存在了，内核没法给你捏，就会给你发SIGBUS信号，导致进程coredump。在不强制内存对齐访问的X86平台上这是发生SIGBUS错误的最主要原因。

4.1.2 brk

```
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t increment);
```

brk用于增长或收缩一个地址空间。这个地址空间就是我们进程的堆段。它一般在bss段之后。其内部实现和mmap是一致的。

4.2 C 库 malloc 的基本原理[6]

1. 通过 brk 或 mmap 从系统批发一整段连续的内存（地址空间），作为原始堆。
2. 用户申请时，从原始堆上切割下来一小片。这切下来的一小片称作一个 chunk。
3. 用户释放时，释放的 chunk 会先被放入空闲 chunk 链表，除非正好挨着原始堆（最后被切下来），那样的话，会合入原始堆；另外如果与释放 chunk 地址相邻的 chunk 是未使用的就会合并。

[6] 详细情况请参考本人博文《glibc malloc管理结构设计浅谈》。

4. 后续的申请优先从空闲链表上获取。

5. 满足特定条件后收缩或完全释放原始堆给系统。

4.3 一段应用代码的执行过程

```
char *p = (char*)malloc(1024);
memset(p, 0, 1024);
```

malloc 可能直接从空闲链表上找到需要的内存块返回；也可能从预分配的堆上切下来一片返回，或者调用 brk|mmap 从系统批发后再切下来。

然后memet访问malloc返回的地址，CPU MMU根据CR3所指的页表基址从内存中读取页表并计算得出物理地址，然后访问。如果页表项不存在，说明还没建立物理映射，产生缺页异常，进入内核注册的异常处理函数do_page_fault。导致异常访问的地址会被放入CR3，然后内核为这个地址建立页表，最后异常返回，CPU继续执行之前失败访存指令，这时候页表也OK了，CPU MMU可以得到物理地址，成功访问。

考虑一个页是4kB，所以memset最多导致两次缺页异常。缺页异常会导致上下文切换，几条[7]简单访存指令演变为数千甚至上万条“乱起八糟”指令的执行，相对开销十分巨大。某些性能关键场合，预分配**并预访问**是个必要的措施。

MMU并非每次都需要从内存中读取页表，映射关系会被缓存，负责缓存的CPU单元叫做TLB（Translation Lookaside Buffers）。如果不缓存，每次1个访存指令，背后可能是4~5次访存操作[8]。TLB可能使访存开销节约数倍。

4.4 大页（hugepage）

TLB能够缓存的页表项是有限的，在TLB固定容量下，提高页的大小，可以提高TLB命中，提高内存访问性能，尤其在数据库等大内存应用的场景下。例如，假设TLB容量就是1个页表项，在4KB连续访问的时候意味着，第一个字节需要访问页表，而后续4K – 1个字节都直接TLB命中。而如果4KB变为2M呢，甚至更多呢？TLB命中率会被增加。

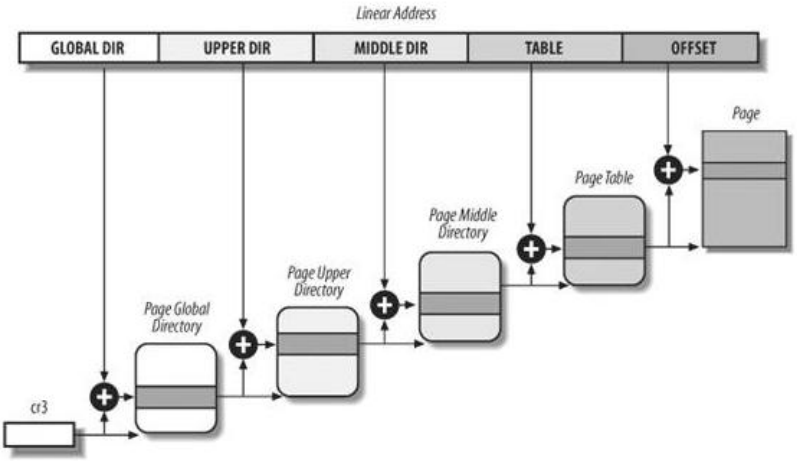
使用测试程序随机访问1G的内存，在一个Intel(R) Xeon(R) CPU E5-2658 0 @ 2.10GHz的环境上，2M的page相对4K的page大概有50%的性能提升。

[7] 包括MMU对页表的访问。

[8] 不一定是真的访问内存，多数情况应该是命中CPU Cache。但这个相对直接使用之前缓存的映射结果，也是慢的。

4.4.1 大页的实现

在x86_64上，只支持2M和1G的大页面，2M需要CPU支持"pse"。1G需要CPU支持"pdpe1gb"。为什么是2M和1G？这两个数字恰好是第4级和3级页表的偏移。所以这个实现其实很简单，就是2M的时候，由原来的4级页表变为3级，即下图所示地址中的的table和offset段都变为offset，共21个bit = 2M。同理1G的时候，3级继续变为2级。offset变为30个bit。



Linux页映射图[9]

4.4.2 大页的使用方法

系统配置

在使用大页面特性之前，需要设置一些系统参数：

设置 vm.nr_overcommit_hugepages 内核参数，单位是页数。

例如1000，表示系统允许1000个大页，如果页面大小是2M，就是2G。

方法1：mmap

使用mmap系统调用申请内存，并加入MAP_HUGETLB标志：

```
#include <sys/mman.h>
#include <linux/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

示例程序：

[9] 抄自《Understanding.the.Linux.Kernel.3rd.Edition》

```
# gcc map_hugetlb.c -o map_hugetlb
# ./map_hugetlb
# pmap 28967
28967: map_hugetlb
START                  SIZE      RSS      PSS    DIRTY      SWAP PERM MAPPING
00007f9a44600000 102400K        0K       0K       0K       0K rw-p /anon_hugepage //区别于普通的内存段。
00007f9a44488000       12K        8K       8K       8K        0K rw-p [anon]
```

方法2：shmget – 共享内存

使用shmget系统调用申请内存，并加入SHM_HUGETLB标志：

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

此方法适用于使用大量共享内存的应用，只需要创建时增加SHM_HUGETLB标志。

方法3：hugetlbfs – 文件系统

使用hugetlbfs虚拟文件系统：

```
mount -t hugetlbfs none /hugetlbfs
```

/hugetlbfs目录的文件直接存于内存，并且使用大页，但要求必须通过mmap的方式访问文件，而不能使用普通read和write调用。

方法4：libhugetlbfs – 库

libhugetlbfs库基于hugetlbfs文件系统和shmget系统调用，做了封装，它的基本原理是：

1. 截获所有对C库malloc – morecore函数的调用，由原来的通过调用brk或mmap申请内存改为在hugetlbfs文件系统上申请内存。
2. 截获所有对shmget的调用，添加SHM_HUGETLB标志。

使用方法：

1. 在程序的启动脚本内，增加并export以下环境变量，程序不需要重新编译和链接。

```
LD_PRELOAD=/usr/lib64/libhugetlbfs.so          #预加载这个库
HUGETLB_SHM=yes                                #拦截shmget
HUGETLB_MORECORE=yes                           #拦截malloc – morecore
HUGETLB_MORECORE_SHRINK=yes                    #允许收缩moecore创建的内存段。
```

注：如果需要拦截malloc – morecore，则需要创建一个hugetlbfs文件系统。

2. 在程序链接阶段增加如下选项，libhugetlbfs.so库会被链接入程序：

-B /usr/share/libhugetlbfs/ -Wl,--hugetlbfs-align

示例：

```
# gcc -B /usr/share/libhugetlbfs/ -Wl,--hugetlbfs-align test.c
# ldd a.out
```

```
linux-vdso.so.1 =>  (0x00007fff91dff000)
libc.so.6 => /lib64/libc.so.6 (0x00007fcc3bea5000)
libhugetlbfs.so => /usr/lib64/libhugetlbfs.so (0x00007fcc3bc8b000)
lib64/ld-linux-x86-64.so.2 (0x00007fcc3c203000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007fcc3ba87000)
```

运行时仍然需要增加以下环境变量，否则不起作用：

```
HUGETLB_SHM=yes                    #拦截shmget
HUGETLB_MORECORE=yes               #拦截malloc – morecore
HUGETLB_MORECORE_SHRINK=yes        #允许收缩moecore创建的内存段。
```

并支持一个额外的变量，用于让程序的数据段和代码段也建立在大页上：

```
HUGETLB_ELFMAP=RW                  #程序代码段和数据段也是使用大页
```

由于glibc的内存管理器malloc在多线程时会有多个分配区，但是只有主分配区会调用morecore扩展，其他分配区都是用mmap，这就意味着libhugetlbfs对于多线程程序通过malloc/new申请的内存不能完全起作用。

1G大页面的支持方法：

在确认CPU支持“pdpe1gb”[10]内核引导参数增加hugepagesz=1G hugepages=10。顺序要对，然后按照方法3挂载hugetlbfs的时候指定pagesize=1g这个挂载参数，即可。

4.4.3 透明大页（THP）

Transparent Huge Pages (THP)在SUSE11SP2上开始支持，在设置了以下内核参数之后，不需要应用进行任何设置和接口调用，内核自动判断并分配2MB的大页。

```
# cat /sys/kernel/mm/transparent_hugepage/enabled
[always] madvise never
```

5 参考资料

Linux Source Code 2.6.32

Understanding.the.Linux.Kernel.3rd.Edition

Linux.Kernel.Development.3rd.Edition.

[10] 通过/proc/cpuinfo文件查看。