**Linux内存分配小结--malloc、brk、mmap**

**Linux 的虚拟内存管理有几个关键概念：**

1、每个进程都有独立的虚拟地址空间，进程访问的虚拟地址并不是真正的物理地址；

2、虚拟地址可通过每个进程上的页表(在每个进程的内核虚拟地址空间)与物理地址进行映射，获得真正物理地址；

3、如果虚拟地址对应物理地址不在物理内存中，则产生缺页中断，真正分配物理地址，同时更新进程的页表；如果此时物理内存已耗尽，则根据内存替换算法淘汰部分页面至物理磁盘中。

基于以上认识，进行了如下分析：

一、Linux 虚拟地址空间如何分布？

Linux 使用虚拟地址空间，大大增加了进程的寻址空间，由低地址到高地址分别为：

**1、只读段：**该部分空间只能读，不可写；(包括：代码段、rodata 段(C常量字符串和#define定义的常量) )

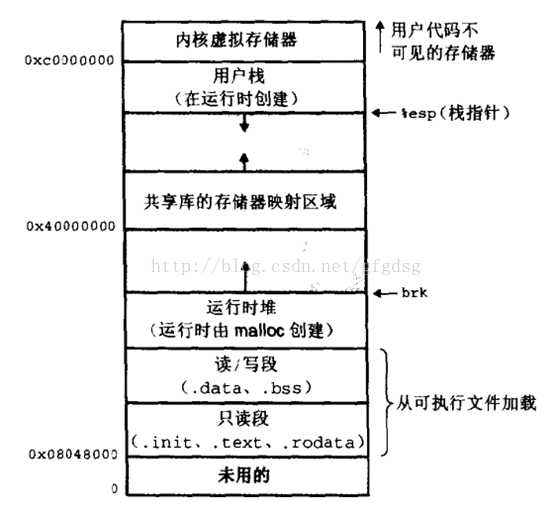
**2、数据段：**保存全局变量、静态变量的空间；

**3、堆** ：就是平时所说的动态内存， malloc/new 大部分都来源于此。其中堆顶的位置可通过函数 brk 和 sbrk 进行动态调整。

**4、文件映射区域 ：**如动态库、共享内存等映射物理空间的内存，一般是 mmap 函数所分配的虚拟地址空间。

**5、栈：**用于维护函数调用的上下文空间，一般为 8M ，可通过 ulimit –s 查看。

**6、内核虚拟空间：**用户代码不可见的内存区域，由内核管理(页表就存放在内核虚拟空间)。

****

32 位系统有4G 的地址空间::

      其中 0x08048000~0xbfffffff 是用户空间，0xc0000000~0xffffffff 是内核空间，包括内核代码和数据、与进程相关的数据结构（如页表、内核栈）等。另外，**%esp 执行栈顶，往低地址方向变化；brk/sbrk 函数控制堆顶\_edata往高地址方向变化。**

**64位系统结果怎样呢？ 64 位系统是否拥有 2^64 的地址空间吗？**

事实上， 64 位系统的虚拟地址空间划分发生了改变：

**1、地址空间大小不是2^32，也不是2^64，而一般是2^48。**因为并不需要 2^64 这么大的寻址空间，过大空间只会导致资源的浪费。64位Linux一般使用48位来表示虚拟地址空间，40位表示物理地址，

这可通过 /proc/cpuinfo 来查看

address sizes   : 40 bits physical, 48 bits virtual

2、其中，0x0000000000000000~0x00007fffffffffff 表示用户空间， 0xFFFF800000000000~ 0xFFFFFFFFFFFFFFFF 表示内核空间，共提供 256TB(2^48) 的寻址空间。

这两个区间的特点是，第 47 位与 48~63 位相同，若这些位为 0 表示用户空间，否则表示内核空间。

3、用户空间由低地址到高地址仍然是只读段、数据段、堆、文件映射区域和栈；

二、malloc和free是如何分配和释放内存？

如何查看进程发生缺页中断的次数？

         用ps -o majflt,minflt -C program命令查看。

          majflt代表major fault，中文名叫大错误，minflt代表minor fault，中文名叫小错误。

          这两个数值表示一个进程自启动以来所发生的缺页中断的次数。

**发成缺页中断后，执行了那些操作？**

**当一个进程发生缺页中断的时候，进程会陷入内核态，执行以下操作：**

1、检查要访问的虚拟地址是否合法

2、查找/分配一个物理页

3、填充物理页内容（读取磁盘，或者直接置0，或者啥也不干）

4、建立映射关系（虚拟地址到物理地址）

重新执行发生缺页中断的那条指令

**如果第3步，需要读取磁盘，那么这次缺页中断就是majflt，否则就是minflt。**

**内存分配的原理**

从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：**brk和mmap（不考虑共享内存）。**

1、brk是将数据段(.data)的最高地址指针\_edata往高地址推；

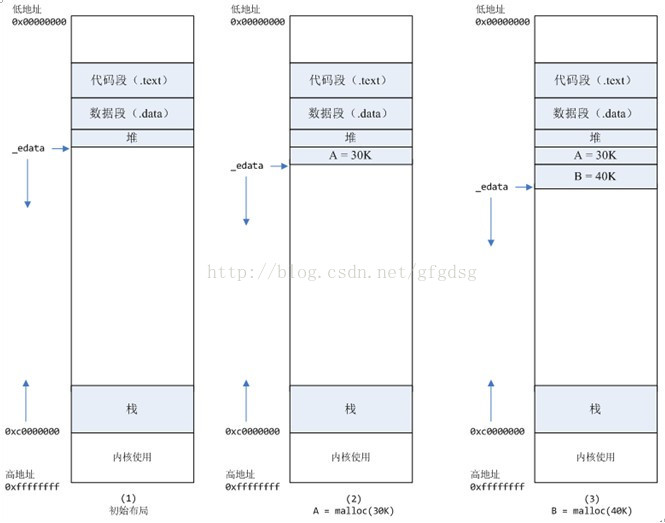
2、mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。

      这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。

在标准C库中，提供了malloc/free函数分配释放内存，这两个函数底层是由brk，mmap，munmap这些系统调用实现的。

**下面以一个例子来说明内存分配的原理：**

情况一、malloc小于128k的内存，使用brk分配内存，将\_edata往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系)，如下图：



1、进程启动的时候，其（虚拟）内存空间的初始布局如图1所示。

      其中，mmap内存映射文件是在堆和栈的中间（例如libc-2.2.93.so，其它数据文件等），为了简单起见，省略了内存映射文件。

**\_edata指针（glibc里面定义）指向数据段的最高地址。**

2、进程调用A=malloc(30K)以后，内存空间如图2：

      malloc函数会调用brk系统调用，将\_edata指针往高地址推30K，就完成虚拟内存分配。

      你可能会问：**只要把\_edata+30K就完成内存分配了？**

      事实是这样的，\_edata+30K只是完成虚拟地址的分配，A这块内存现在还是没有物理页与之对应的，等到进程第一次读写A这块内存的时候，发生缺页中断，这个时候，内核才分配A这块内存对应的物理页。**也就是说，如果用malloc分配了A这块内容，然后从来不访问它，那么，A对应的物理页是不会被分配的。**

3、进程调用B=malloc(40K)以后，内存空间如图3。

**情况二、malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为0)，如下图：**

**4、进程调用C=malloc(200K)以后，内存空间如图4：**

**默认情况下，malloc函数分配内存，如果请求内存大于128K（可由M\_MMAP\_THRESHOLD选项调节），那就不是去推\_edata指针了，而是利用mmap系统调用，从堆和栈的中间分配一块虚拟内存。**

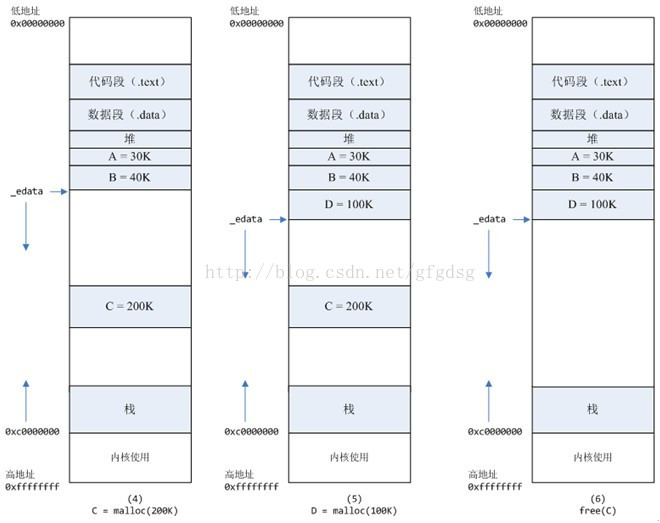
**这样子做主要是因为::**

**brk分配的内存需要等到高地址内存释放以后才能释放（例如，在B释放之前，A是不可能释放的，这就是内存碎片产生的原因，什么时候紧缩看下面），而mmap分配的内存可以单独释放。**

**当然，还有其它的好处，也有坏处，再具体下去，有兴趣的同学可以去看glibc里面malloc的代码了。**

**5、进程调用D=malloc(100K)以后，内存空间如图5；**

**6、进程调用free(C)以后，C对应的虚拟内存和物理内存一起释放。**



7、进程调用free(B)以后，如图7所示：

**B对应的虚拟内存和物理内存都没有释放，因为只有一个\_edata指针，如果往回推，那么D这块内存怎么办呢？**

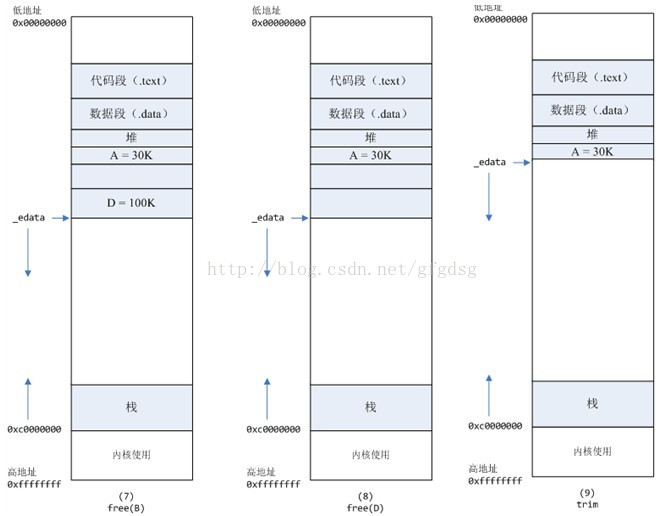
当然，B这块内存，是可以重用的，如果这个时候再来一个40K的请求，那么malloc很可能就把B这块内存返回回去了。

8、进程调用free(D)以后，如图8所示：

        B和D连接起来，变成一块140K的空闲内存。

9、默认情况下：

       当最高地址空间的空闲内存超过128K（可由M\_TRIM\_THRESHOLD选项调节）时，执行内存紧缩操作（trim）。在上一个步骤free的时候，发现最高地址空闲内存超过128K，于是内存紧缩，变成图9所示。



**三、既然堆内内存brk和sbrk不能直接释放，为什么不全部使用 mmap 来分配，munmap直接释放呢？**

        既然堆内碎片不能直接释放，导致疑似“内存泄露”问题，为什么 malloc 不全部使用 mmap 来实现呢(mmap分配的内存可以会通过 munmap 进行 free ，实现真正释放)？而是仅仅对于大于 128k 的大块内存才使用 mmap ？

其实，进程向 OS 申请和释放地址空间的接口 sbrk/mmap/munmap 都是系统调用，频繁调用系统调用都比较消耗系统资源的。并且， mmap 申请的内存被 munmap 后，重新申请会产生更多的缺页中断。例如使用 mmap 分配 1M 空间，第一次调用产生了大量缺页中断 (1M/4K 次 ) ，当munmap 后再次分配 1M 空间，会再次产生大量缺页中断。**缺页中断是内核行为，会导致内核态CPU消耗较大。**另外，如果使用 mmap 分配小内存，会导致地址空间的分片更多，内核的管理负担更大。

**同时堆是一个连续空间，并且堆内碎片由于没有归还 OS ，如果可重用碎片，再次访问该内存很可能不需产生任何系统调用和缺页中断，这将大大降低 CPU 的消耗。** 因此， glibc 的 malloc 实现中，充分考虑了 sbrk 和 mmap 行为上的差异及优缺点，默认分配大块内存 (128k) 才使用 mmap 获得地址空间，也可通过 mallopt(M\_MMAP\_THRESHOLD, <SIZE>) 来修改这个临界值。

**四、如何查看进程的缺页中断信息？**

可通过以下命令查看缺页中断信息

ps -o majflt,minflt -C <program\_name>

ps -o majflt,minflt -p <pid>

其中:: majflt 代表 major fault ，指大错误；

           minflt 代表 minor fault ，指小错误。

这两个数值表示一个进程自启动以来所发生的缺页中断的次数。

其中 majflt 与 minflt 的不同是::

        majflt 表示需要读写磁盘，可能是内存对应页面在磁盘中需要load 到物理内存中，也可能是此时物理内存不足，需要淘汰部分物理页面至磁盘中。

**五、C语言的内存分配方式与malloc**

C语言跟内存分配方式

（1） 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static变量。

（2） 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运

算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

（3）从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多

     C语言跟内存申请相关的函数主要有 alloc,calloc,malloc,free,realloc,sbrk等.其中alloc是向栈申请内存,因此无需释放. malloc分配的内存是位于堆中的,并且没有初始化内存的内容,因此基本上malloc之后,调用函数memset来初始化这部分的内存空间.calloc则将初始化这部分的内存,设置为0. 而realloc则对malloc申请的内存进行大小的调整.申请的内存最终需要通过函数free来释放. 而sbrk则是增加数据段的大小;

       malloc/calloc/free基本上都是C函数库实现的,跟OS无关.C函数库内部通过一定的结构来保存当前有多少可用内存.如果程序 malloc的大小超出了库里所留存的空间,那么将首先调用brk系统调用来增加可用空间,然后再分配空间.free时,释放的内存并不立即返回给os, 而是保留在内部结构中. 可以打个比方: brk类似于批发,一次性的向OS申请大的内存,而malloc等函数则类似于零售,满足程序运行时的要求.这套机制类似于缓冲.

使用这套机制的原因: 系统调用不能支持任意大小的内存分配(有的系统调用只支持固定大小以及其倍数的内存申请,这样的话,对于小内存的分配会造成浪费; 系统调用申请内存代价昂贵,涉及到用户态和核心态的转换.

函数malloc()和calloc()都可以用来分配动态内存空间，但两者稍有区别。

在Linux系统上，程序被载入内存时，内核为用户进程地址空间建立了代码段、数据段和堆栈段，在数据段与堆栈段之间的空闲区域用于动态内存分配。

      内核数据结构mm\_struct中的成员变量start\_code和end\_code是进程代码段的起始和终止地址，start\_data和 end\_data是进程数据段的起始和终止地址，start\_stack是进程堆栈段起始地址，start\_brk是进程动态内存分配起始地址（堆的起始 地址），还有一个 brk（堆的当前最后地址），就是动态内存分配当前的终止地址。

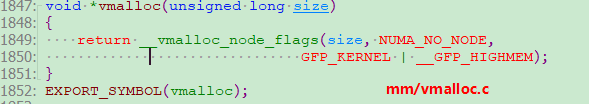
C语言的动态内存分配基本函数是malloc()，在Linux上的基本实现是通过内核的brk系统调用。brk()是一个非常简单的系统调用，只是简单地改变mm\_struct结构的成员变量brk的值。

      mmap系统调用实现了更有用的动态内存分配功能，可以将一个磁盘文件的全部或部分内容映射到用户空间中，进程读写文件的操作变成了读写内存的操作。在 linux/mm/mmap.c文件的do\_mmap\_pgoff()函数，是mmap系统调用实现的核心。do\_mmap\_pgoff()的代码，只是新建了一个vm\_area\_struct结构，并把file结构的参数赋值给其成员变量m\_file，并没有把文件内容实际装入内存。

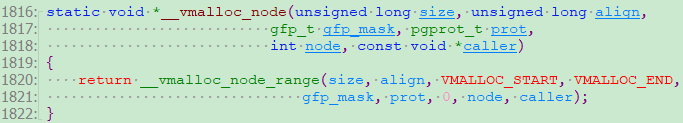
Linux内存管理的基本思想之一，是只有在真正访问一个地址的时候才建立这个地址的物理映射。

**2.6 VMOLLOC**

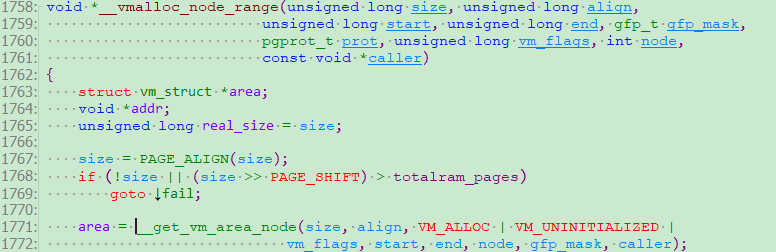
函数：kmalloc,vmalloc和malloc 这3个常用API函数具有相当的分量。三者看上去很相似，但是实现上可大有讲究。Kmalloc 基于slab分配器，slab缓冲区建立 在一个连续物理地址的大块内存之上，所以其缓存对象也是物理地址连续的，如果在内核中不需要连续的物理地址，而仅仅需要内核空间是连续虚拟地址内存块，该 如何处理呢？这时vmalloc就派上用场所了。



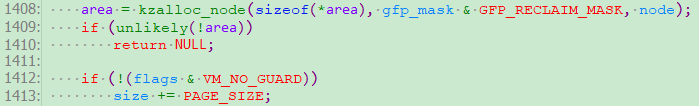
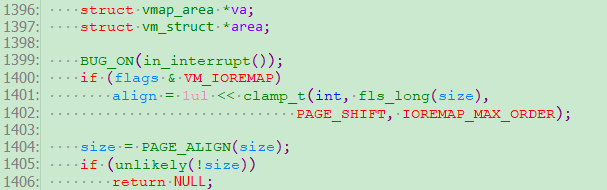
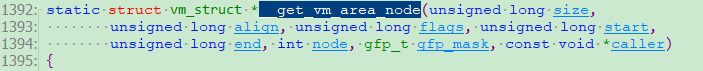
函数使用分配掩码是” GFP\_KERNEL | \_\_GFP\_HIGHMEM”,说明会优先使用高端内存high memory.

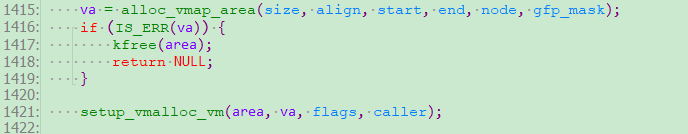


这里的VMALLOC\_START, VMALLOC\_END是vmalloc中很重要的宏，这两个宏定义在[arch/arm/include/pgtable.h]头文件，ARM64 [arch/arm64/include/pgtable.h]。 VMALLOC\_START 是vmalloc 区域的开始地址，它是在high\_memory 指定的高端内存开始地址再加上8MB大小的安全区域(VMALLOC\_OFFSET),





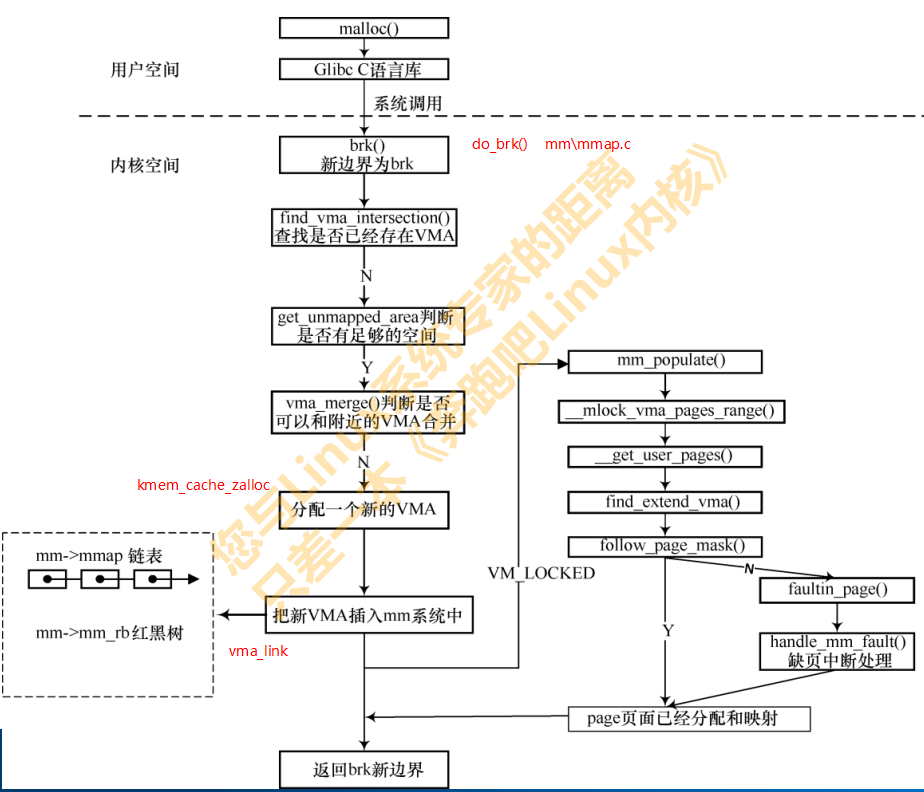
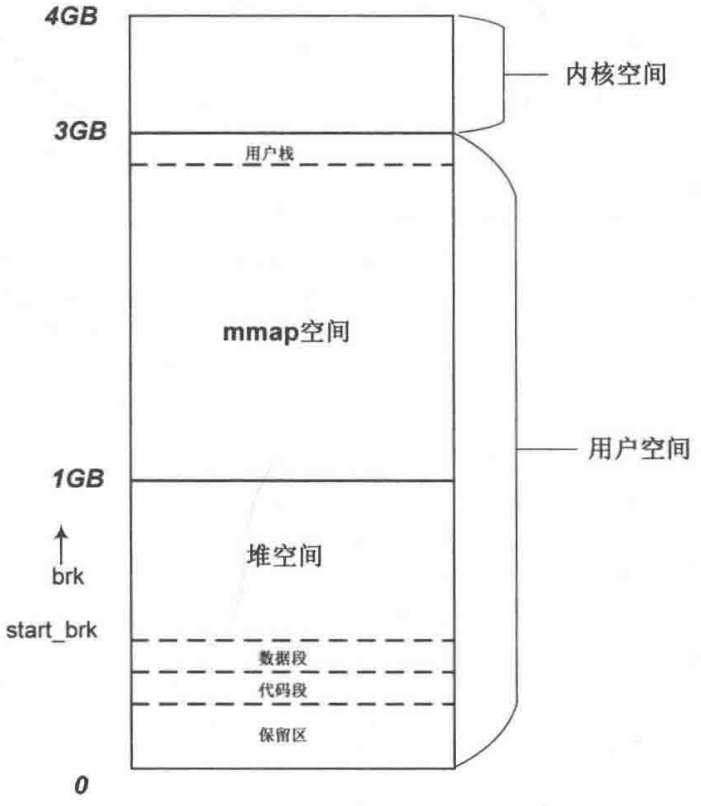


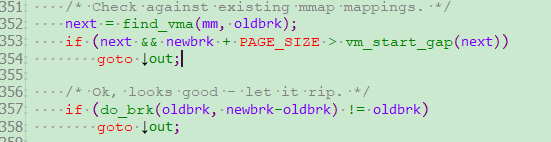


1408行代码分配一个struct vm\_struct 数据结构中描述这个vmalloc区域。

1412 如果flags中没有定义VM\_NO\_GUARD标志位，那么要多分配一个页来做安全垫，例如要分配4KB大小内存，vmalloc分配了8KB内存块。







函数：find\_vma函数以老边界old\_brk地址去查找系统中有没有一块已经存在vma,它通过find\_vma来查找当前用户进程中是否已经有一块vma和start-addr有重叠。

函数do\_brk()函数是这里核心函数

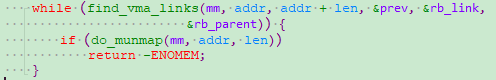




函数声明：do\_brk()函数中，申请分配内存大小要以页面大小对齐。函数



函数用来判断 虚拟内存空间是否足够的空间。返回一段没有映射过的空间的超始地址，这个函数会调用到具体体系结构中实现。



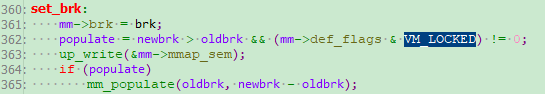
代码中find\_vma\_links()函数之前已经阅读过，它循环遍历用户进程红黑树中的VMAS,然后根据addr来查找最合适插入到红黑树的节点，最终rb\_link指针指向最合适节点rb\_left或者 rb\_rights指针本身的地址。返回 0 表示寻找到最合适的插入节点，返回-ENOMEN表示和现有的VMA重叠，这时会调用do\_munnap()函数来释放这段重叠的空间。



函数vm\_merage()函数去找有没有可能合并addr附近vma。如果没有办法合并，那么只能新创建的一个vma,vma的地址空间就是[addr,addr+len].



新创建的vma需要加入到mm->mmap链表和红黑树中，vm\_link函数实现 这个功能，

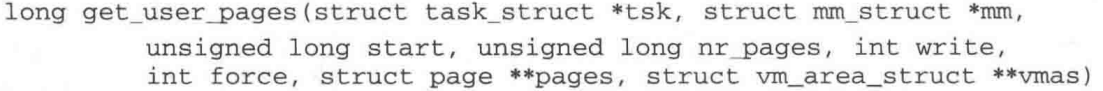


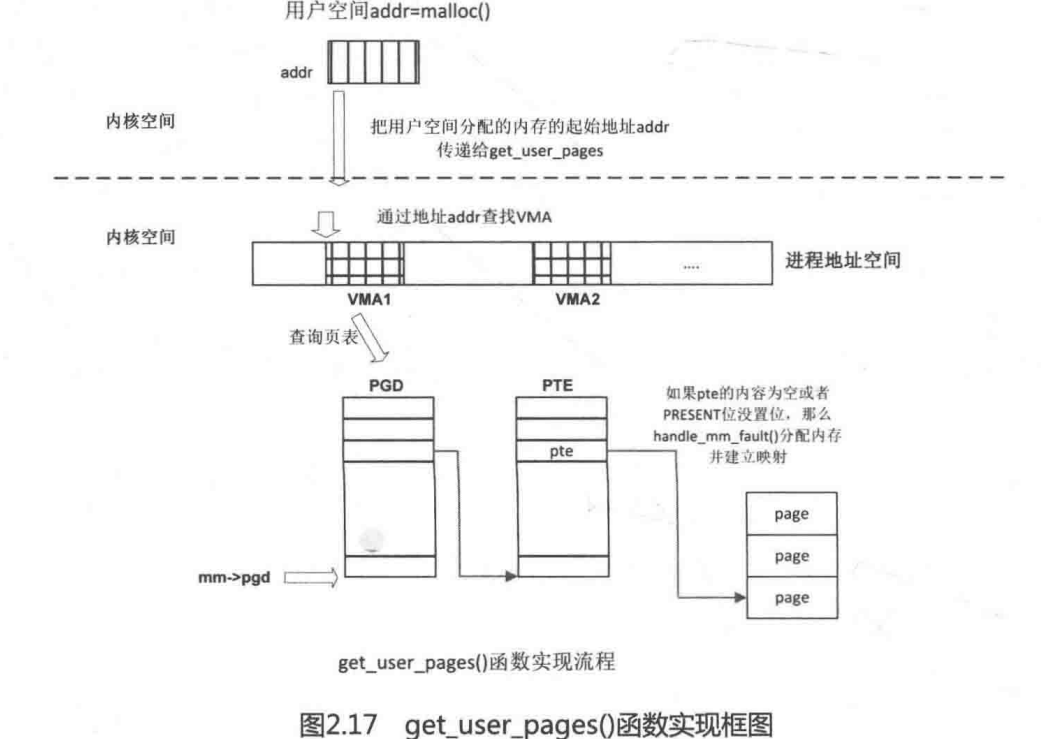
这里判断 flag是否置 位VM\_LOCKED，这个VM\_LOCKED通常从mlockall系统调用中设置而来，如果有，那么需要用户mm\_populate().马上分配物理内存并建立 映射。通常用户程序很使用VM\_LOCKED分配掩码，所以brk不会为这个用户进程立马分配物理页面。而是一直将分配物理页面的工作推延到用户进程需要访问这些虚拟页面时。发生缺页中断才会分配物理内存。并和虚拟地址建立 映射关系。

小结：

1. get\_user\_pages()函数，

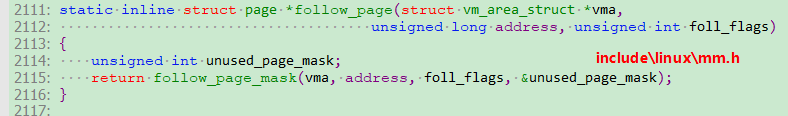
用于把用户空间的虚拟内存空间传到内核空间。内核空间为其分配物理内存并建立 相应的映射关系，实现过程如图，例如camera 驱动，V4L2核心架构中可以使用用户空间内存类型（）来分配 物理内存，其驱动的实现 使用的是get\_user\_pages()函数。





（2）follow\_page()函数

通过虚拟地址addr寻找相应的物理页面，返回normal mapping 页面对应的struct page 数据结构，该函数会查询页表。



（3）vm\_normal\_page()函数

该函数由pte返回normal mapping 的struct page 数据结构，主要目的是过虑掉那些令人讨论special mapping 页面。



上述是内存管理中最经典的3个函数，值得读者细细品味