在 Linux 2.6.11 中进程调用 malloc 函数的情景分析

郑杰，郭益林，吴爱华
（厦门大学，厦门 361005）

摘　要： 在 Linux 系统中用户调用 malloc 函数的过程实际上是一个间接调用 brk 函数的过程，brk 函数在内核中的实现为 sys_brk 函数。详细探讨了 Linux 内核 2.6.11 版本中内存管理与分配的细节，重点分析了 sys_brk 函数的代码，并提供了 do_munmap 和 do_brk () 这两个主要的函数的流程图。sys_brk 函数可以用来分配空间，即把动态分配区底部的边界往上推，也可以用来释放内存，即归还空间。因此它的代码也大致上分成两部分，对相关函数调用与流程作了详细的分析。

关键词： 进程调用；进程空间；malloc；sys_brk；do_brk；do_munmap

The Scenario Analysis of Procedure Call of Malloc Function in Linux 2.6.11

ZHENG Jie, GUO Yilin, WU Aihua
(Xiamen University ,Xiamen 361005)

Abstract: User calls of the malloc function in the linux system is an indirect process of brk function, which the realization in the kernel is sys_brk function. We discusses the details of the memory management and allocation in the 2.6.11 version of linux kernel, focusing on an analysis of sys_brk function, and provides the two main flow chart of do_munmap and do_brk. Sys_brk function can be used to allocate space, that is, push up the border at the bottom of the dynamic allocation area , but also can be used to release memory, that is, return the space. As Code is also generally divided into two parts, this paper analyses the calls of related functions and processes in detail.

Keywords: Process call；Process space；malloc；sys_brk；do_brk；do_munmap

1　引言

要分析 malloc，这里需要注意，malloc 并非 linux 的系统调用，而是 C 标准库的函数。所以，如果要在 linux 内核代码中找 malloc 的实现，无疑是南辕北辙。

但可以知道，malloc 是通过系统调用 brk () 实现的。而 brk () 则是通过 sys_brk () 实现的[1]。“如果把 malloc () 想象成零售，brk () 则是批发。库函数 malloc () 为用户进程（malloc 本身就是该进程的一部分）维持一个小仓库，当进程需要使用更多的内存空间时就向小仓库要，小仓库中存量不足时就通过 brk () 向内核批发”[2]。虽然这段话是针对 2.4 内核的，但当分析完 sys_brk () 时就会发现，这句话同样适用于要分析的 2.6.11 的内核。

现在分析 sys_brk () 的实现。sys_brk () 函数的作用是重新设置进程地址空间中数据段的边界。传入的参数是一个 long 型的值，作为新的数据段的边界。返回值是数据段的新的边界。对用户程序而言在分析之前，需要了解的一些基本的知识。

2　基础知识

已知程序有自己的代码段和数据段。数据段中包含了所有静态分配的数据空间，即全局变量和所有声明为 static 的局部变量，这些空间是进程所必需的，是在建立一个进程的运行映像时就分配好的。包括虚存空间和物理页面的映射。除此之外，堆栈使用的空间也属于基本要求。所以也在建立进程时就分配好。所不同的是，堆栈空间被安置在虚存空间的顶部，运行时自顶向下延伸；代码段和数据段则在底部。而从数据段的顶部 end_data 到堆栈段的底部这段内存空洞就是在运行时动态分配的内存空间。也就是通常说的堆空间。分配是从 end_data 即数据段的顶部向上增长的。每次分配一段空间，顶部就向上生长一段，同时内核和进程都要记下当前边界的位置。在进程中由 malloc ()，free () 等管理，而在内核中则将当前的边界记录在进程的 mm_struct 结构中。在 mm_struct 中有一个域 brk 记录了当前的数据段边界（堆空间的终止地址）。

图 1　3GB 的用户空间

Linux 把进程的用户空间划分为一个个区间，便于管理。一个进程的用户地址空间主要由 mm_struct 结构和 vm_area_structs 结构来描述。

```
struct mm_struct
 {
  struct vm_area_struct *mmap; //list of vmas;
  struct rb_root mm_rb; //red black tree
```

本文收稿日期：2009-9-28

```
struct vm_area_struct* mmap_cache;//last find vma result
 pgd_t * pgd;
 atomic_t mm_users;  //how many users with user space
 atomic_t mm_count;  //how many refs to mm_struct
 struct rw_semaphore mmap_sem; //read/write semaphore
 unsigned long start_code,end_code,start_data,end_data;
 unsigned long start_brk,brk,start_stack;
 unsigned long arg_start,arg_end,env_start,env_end;
 ...
} ;
```

以上的 mm_struct 包含了主要的描述进程地址空间的字段。

其中 mmap 指向线性虚拟内存地址区间的链表头部。mm_struct 中的 mm_rb 字段是 Linux 为了对加快对进程线性区的搜索，把各进程的线性区组织成的红-黑树。红-黑树是一种平衡二叉树，满足如下规则：

(1) 每个节点必须为黑或红。

(2) 树的根节点必须为黑。

(3) 红节点的孩子必须为黑。

(4) 从一个节点到后代节点的每个路径上都包含相同数量的黑节点。

(5) 插入新节点时，新节点必须先作为叶子插入并着成红色。如果违反上述规则则必须移动或对节点重新着色。

线性区是通过 vm_rb 字段链入红黑树中。mmap_cache 则描述了最近使用的线性地址空间。mm_users 和 mm_count 字段分别描述了进程共享 mm_struct 的数量和内核线程对该 mm_struct 描述符的“借用”。内核线程在运行时会借用其他进程的 mm_struct,这样的线程叫“anonymous users”，因为他们不关心 mm_struct 指向的用户空间,也不会去访问这个用户空间，他们只是临时借用。mm_users 是对 mm_struct 所指向的用户空间进行共享的所有进程的计数。也就是说,会有多个进程共享同一个用户空间。

每个进程都有自己的用户空间，但是调用 clone () 函数创建的内核线程时共享父进程的用户空间。在写时复制方法中，子进程继承父进程的用户空间：只要页是只读的，就依然共享它们。当其中的一个进程试图写入某一个页时，这个页就被复制一份；一段时间之后，所创建的进程通常获得与父进程不一样的完全属于自己的用户空间。而对于内核线程来说，它使用父进程的用户空间，因此创建内核线程比创建普通进程相应要快得多，而且只要父进程和子进程谨慎地调整它们的访问顺序，就可以认为页的共享是有益的。

mm_struct 的以下的几个字段则是描述了程序运行时的环境和进程用户空间的分布状况。start_code,end_code 描述了代码段的分布；start_data,end_data 描述数据段的分布；start_brk,brk 描述了堆的分布；start_stack 描述了栈的分布；而 env_start,env_end,arg_start,arg_end 则是描述了环境变量和传入参数的分布。

现在看另一个和 mm_struct 密切相关的数据结构：struct vm_area_struct。

```
struct vm_area_struct
 {
 struct mm_struct * vm_mm;
 unsigned long vm_start;
 unsigned long vm_end;
 struct vm_area_struct * vm_next;
 pgprot_t vm_page_prot;
 unsigned long vm_flags;
 struct rb_node vm_rb;
 union {
     struct {
        struct list_head list;
        void * parent;
        struct vm_area_struct *head;
  } vm_set;
   struct raw_prio_tree_node prio_tree_node;
} shared;
    struct list_head anon_vma_node;
     struct anon_vma* anon_vma;
     struct vm_operations_struct * vm_ops;
     unsinged long vm_pgoff;
     struct file * vm_file;
     void * vm_private_data;
     unsigned long vm_truncate_count;
     ...
} ;
```

其中 vm_mm 描述了该 vm_area_struct 所属的 mm_struct。这里需要注意，为何一个 mm_struct 通过一个 vm_area_struct 的链表来描述。即为何用户空间被划分成了一个个的线性区。这是因为每个虚存区的来源可能不同，有的来自可执行映像，有的来自共享库，而有的则可能是动态分配的内存区；不同的区间可能具有不同的访问权限，不同的操作。因此 Linux 把进程的用户空间分割，并利用了虚存区处理函数 (vm_ops) 来抽象对不同来源的虚存区的处理方法。其中 vm_ops 的定义如下:

```
struct vm_operations_struct  {
      void  (*open)  (struct vm_area_struct * area) ;
      void  (*close)  (struct vm_area_struct * area) ;
 struct page *  (*nopage)  (struct vm_area_struct * area,
unsigned long address, int unused)}
```

vm_operations 结构中包含的是函数指针；其中，open、close 分别用于虚存区的打开、关闭，而 nopage 是当虚存页面不在物理内存而引起的“缺页异常”时所应该调用的函数。

vm_start 和 vm_end 分别表示该线性区的起点和终点。vm_next 构成线性链表的指针，按虚存区基址从小到大排列。vm_rb 用来连接红黑树的数据。vm_page_prot 表示该线性区中页的操作权限。

进程控制块是内核中的核心数据结构。在进程的 task_struct 结构中包含一个 mm 域，它是指向 mm_struct 结构的指针。而进程的 mm_struct 结构则包含进程的可执行映像信息以及进程的页目录指针 pgd 等。该结构还包含有指向 vm_area_struct 结构的几个指针，每个 vm_area_struct 代表进

程的一个虚拟地址区间。

下面，将进入具体的 sys_brk () 的代码分析。

3 sys_brk () 源码分析

```
asmlinkage unsigned long sys_brk (unsigned long brk)
{
        unsigned long rlim, retval;
        unsigned long newbrk, oldbrk;
        struct mm_struct *mm = current->mm;
        down_write (&mm->mmap_sem) ;
        if (brk < mm->end_code)
                goto out;
        newbrk = PAGE_ALIGN (brk) ;
        oldbrk = PAGE_ALIGN (mm->brk) ;
        if (oldbrk == newbrk)
                goto set_brk;
        /* Always allow shrinking brk. */
        if (brk <= mm->brk)    {
              if (! do_munmap (mm, newbrk, oldbrk-newbrk))
                        goto set_brk;
                goto out;
        }
        /* Check against rlimit.. */
         rlim = current ->signal ->rlim [ RLIMIT_DATA] .
rlim_cur;
         if ( rlim < RLIM_INFINITY && brk - mm ->
start_data > rlim)
            goto out;
        /* Check against existing mmap mappings. */
         if ( find_vma_intersection ( mm, oldbrk, newbrk +
PAGE_SIZE))
            goto out;
        /* Ok, looks good - let it rip. */
        if (do_brk (oldbrk, newbrk-oldbrk) ! = oldbrk)
            goto out;
                set_brk:
        mm->brk = brk;
                out:
            retval = mm->brk;
            up_write (&mm->mmap_sem) ;
            return retval;
}
```

在这里，先了解该函数的总体思路，然后逐步分析该函数的实现细节。

该函数的步骤如下：

(1) 获得当前进程的 mm_struct；其中的 current 是一个宏，在/include/asm-i386/current.h 中有它的定义。代表了 getcurrent () 函数。而 getcurrent () 则是返回一个 task_struct 的指针。所以可以认为 current 指的是当前进程的 task_struct。

(2) 对 mm_struct 做互斥访问；通过 down_write () 设置了 mm_struct 中的 mmap_sem。注意，因为要修改该内存区，而 mm_struct 本身是可以共享的，可能同时有多个进程访问。所以这里为了正确修改 mm_struct 中的属性值，需要为该 mm_struct 加锁。

(3) 检查传入的新的堆边界是否合法；如果比 mm_struct 的 end_code 的边界还小，不合法。如前所说，这里是在分配堆的空间。只能在数据段的上方 (end_data) 进行分配和释放。

(4) 判断是否要分配新的页；把新的边界和旧的边界按页对齐，PAGE_ALIGN 宏是按照页的大小进行对齐的。如果是同一页，意味着不需要分配释放或者分配新的页 (修改页面映射)，仅需设置一下在原来的页中的边界偏移即可。

(5) 否则，判断是分配新的页还是释放原来的页；如果传入的参数比原来的边界 mm->brk 小，意味着要释放多出的空间。通过 do_munmap () 释放从新的边界开始到旧的边界处的页 (除去该处的页面映射)。如果释放成功，则设置成新的边界；如果释放失败，仍使用原来的页边界，并释放锁，返回。

(6) 如果传入的参数比原来的边界大，意味着要分配新的页。则首先判断要分配的空间是否超过了该进程的数据段的段限长。在 task_struct 的 signal 中存储了该 task_struct 的各种限长。共同组成了 rlim 数组。如果超过了，也返回。

找到覆盖按页对齐后的新边界和老边界的第一个 vm_area_struct。如果存在，意味着该区段已经被分配出去。直接返回。find_vma_intersection ( ) 查找第一个与给定地址区间相交的线性区。参数：mm 是指向该进程的内存描述符的指针，start_addr 是给定地址区间的起始地址，end_addr 是给定地址区间的结束地址。返回类型是一个指向线性区的指针。如果这样的线性区不存在，将返回 NULL.

(7) 否则，调用 do_brk () 分配从 oldbrk 到新边界处的页面。具体的 do_brk 分析参照下面。

(8) 设置边界。设置返回值。释放 mm_struct 上的锁，并返回。

以上是对 sys_brk () 函数过程的总体分析，下面将进一步详细分析该调用过程各步骤。

4 do_munmap () 源码分析

4.1 do_munmap () 源码分析

do_munmap () 作用是释放 mm_struct 中相应线性区间的内存。传入参数 mm_struct 和线性区间的首地址以及线性区间的长度。返回 0 表示成功，否则失败。

主要有两个步骤：找到 mm_struct 的所有被该区间覆盖的 vm_area_struct，从 mm_struct 的链表和红黑树中删除；更改页表中对应该区间中的表项。删除相应的表项。

分成以下的步骤：

(1) 检查传入的线性地址区间的起点是否合法。如果没有对齐或者超过了 TASK_SIZE，或者线性区间的末端超过了 TASK_SIZE，直接返回错误。TASK_SIZE 宏标识了用户地址空间的界限，为 3G。

(2) 找到满足 mpnt->end>start 的 vma，通过 find_vma_prev 实现。

如果没有这样的 vma 或者存在这样的 vma，但该 vma 同这个区间没有任何交集，直接返回。如果线性区间同该 vma mpnt 有交集，即线性区间的起点在 mpnt 的内部，有 start>mp-

(下转到 37 页)

减 1，最终级数再到 0。程序运行效果如图 3 所示。

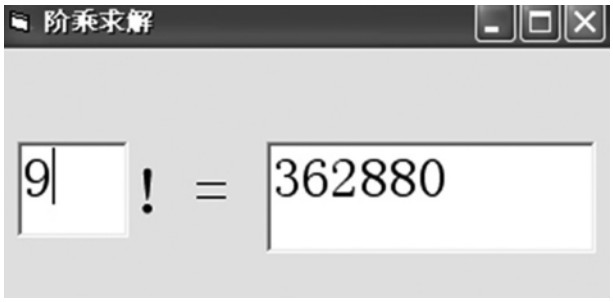

图 3 程序运行效果

4 结语

递归算法在很多方面有应用价值，其中典型的在数学中求解契比雪夫多项式、素数的求解、汉诺塔求解和积分求解等都十分有效。

参考文献

[1] 刘瑞新，汪远征. Visual Basic 程序设计教程 [M] .北京：机械工业出版社，2000.

[2] 谭浩强. Visual Basic 程序设计 [M] . 北京：清华大学出版社，2005.

[3] 丁民选. 录音技术在课件制作中的应用 [J] . 山西广播电视大学学报，2007，(1)：39-40.

作者简介

伍意，女 (1990-)，湖北常德人。

(上接第 26 页)

nt->vm_start 且 start<mpnt->end 则通过 split_vma () 将该 vma 分成两个不同的 vma。便于将要删除的区间直接从链表中卸掉。其中 split_vma 作用是将一个线性区间的 vma 分成两个小的 vma。接收参数一个 mm_struct，和相应的 vma，两个小的 vma 的边界 addr。和一个 new_below：标记新产生的 vma 是原来的 vma 区间的上半部分还是下半部分。

❸ 同时，找到线性区间的终点所在的 vma，如果找到，调用 split_vma 将该 vma 分成两个 vma。

❹ 修改 mpnt 的值，使它指向线性区间的第一个 vma。

❺ 调用 detach_vmas_to_be_unmapped () 从 mm_struct 的 vma 链表和红黑树中移除该线性区间覆盖的 vma。detach_vmas_to_be_unmapped () 的代码:

```
vma = mpnt;
insertion_point = (prev ? &prev->vm_next : &mm->mmap);
do {
    rb_erase(&vma->vm_rb, &mm->mm_rb);
    mm->map_count--;
    tail_vma = vma;
    vma = vma->next;
} while (vma && vma->start < end);
*insertion_point = vma;
    tail_vma->vm_next = NULL;
mm->map_cache = NULL;
```

❻ 获得该 mm_struct 的页表的自旋锁。调用 unmap_region () 删除该线性区间对应的页表项。unmap_region 作用是移除属于一个线性区间的所有页表项。它接收 5 个参数。其中 mm_struct 指示了要移除的线性区所属的进程地址空间。vma 则指示要移除的区间所覆盖的第一个 vm_area_struct。prev 指示了在该进程地址空间中的 vma 链表中的该 vma 的前一项。

❼ 释放 mm_struct 页表的自旋锁。

❽ 释放所有从链表中删除的 vam 的空间。

4.2 针对 do_brk () 函数的情景分析

do_brk () 函数用于分配线性区，do_brk () 函数实际上可以理解为是一个简化版本的 do_mmap ()。do_brk () 函数只处理了匿名内存区域，它的调用可以同等于：

do_mmap (NULL,oldbrk,newbrk -oldbrk,PROT_READ | PROT_WRITE|PROT_EXEC,MAP_FIXED|MAP_PRIVATE, 0)；

参数：addr 是需要建立映射的新区间的起始地址，len 为区间的长度.返回的值是若建立映射失败-ENOMEM，如果成功建立映射，就返回传入的线性地址。

进程调用 malloc 函数的过程最终归结为对 sys_brk 函数的调用。是用来分配和释放空间的函数，它的整体框架由 old-brk，newbrk 决定，当 oldbrk > newbrk 时调用 do_munmap 函数释放内存，反之则调用 do_brk 扩充内存。释放内存时，首先处理 MAX_MAP_COUNT 限制，若虚拟内存存在空洞则将其一分为二，然后将要释放的虚拟内存空间串成一个 free 链表，对 free 链表中要释放的虚拟内存空间一一调用 zap_page_range 函数释放其所对应的二级页表，再调用用 unmap_fixup 函数相应调整对应的 vm_area_struct {}，处理所有 free 链表中的虚存空间后，最终释放页面表。扩充内存时则调用 do_brk 函数，该函数能合并则合并，不能合并则再分配一个 vm_area_struct {}，插入 vm_area {} 中去，利用 handle_mm_fault () 来将相关页面调入内存。

参考文献

[1] Daniel P. Bovet，Marco Cesati. Unstanding the Linux kernel 3nd [M] . O'Reilly 2005.

[2] 毛德操，胡希明. LINUX 内核源代码情景分析 [M] . 浙江大学出版社，2001.

作者简介

郑杰，男 (1988-)，厦门大学软件学院软件工程本科生。

郭益林，男，厦门大学软件学院软件工程本科生。

吴爱华，女，厦门大学软件学院软件工程本科生。