Linux 2.6.39 内存管理源码阅读

唐凯成，PB16001695

一、Linux 内存管理概述

1. 内存管理核心功能

Linux 中的所有进程都必须占用一定数量的内存，内存中或是存放从磁盘载入的程序代码，或是存放取自用户输入的数据等。不过进程对这些内存的管理方式因内存用途不一而不尽相同，例如有的内存是事先静态分配和统一回收的，而有的则是按需要动态分配和回收的。Linux 中内存管理的核心功能主要有以下几点：

（1）实现对虚拟内存和虚拟内存的管理。

（2）实现虚拟内存到物理内存的映射。

（3）实现进程内存的分配和回收，并尽量减少内存中的“分片”现象、减少小内存块。

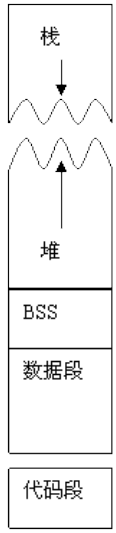
（4）不同进程间的共享虚拟内存。

（5）对进程代码和数据的保护及内存相关异常的管理。

2. 内存管理实现原理

（1）内存空间的划分

为实现以上内存管理功能，Linux 系统内存管理模块中将每个进程涉及的数据在内存空间中分为以下五种数据段：



A）代码段：代码段用来存放可执行文件的操作指令，即为可执行程序在内存中的镜像。代码段需要防止在运行时被非法修改，所以只准许读取操作，而不允许写入（修改）操作——它是不可写的，因此也起到了对进程的保护作用。

B）数据段：数据段用来存放可执行文件中已初始化全局变量，即存放程序静态分配的变量和全局变量。

C）BSS 段：BSS 段包含了程序中未初始化的全局变量，在内存中 BSS 段全部置零。

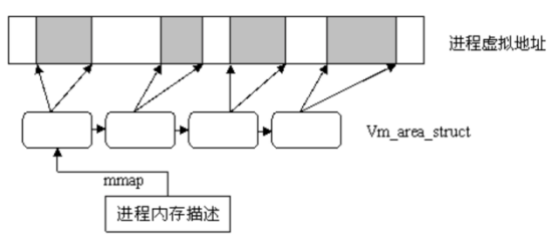
D）堆：堆用于存放进程运行中被动态分配的内存段，其大小并不固定，可进行动态扩张或缩减。当进程调用 malloc 等函数分配内存时，新分配的内存就被动态添加到堆上，当利用 free 等函数释放内存时，被释放的内存从堆中被剔除。

E）栈：栈是用户存放程序临时创建的局部变量。在函数被调用时，其参数会被压入发起调用的进程栈中，待到调用结束后，函数的返回值也会被存放回栈中。由于栈的特点，所以课用来保存/恢复调用现场。

上述几种内存区域中数据段、BSS 和堆通常是被连续存储的，如右图所示。

（2）虚拟内存的管理

为了存放以上数据段，Linux 定义了内核中对应进程内存区域的数据结构 vm_area_struct，将每个内存区域作为一个单独的内存对象管理，不同的内存结构间以链表形式链接，同时内核又以红黑树的形式组织内存区域，以便降低搜索耗时，该结构体将在后面介绍。右图即为两种形式组合成的进程地址空间的管理模型。



（3）物理内存的管理

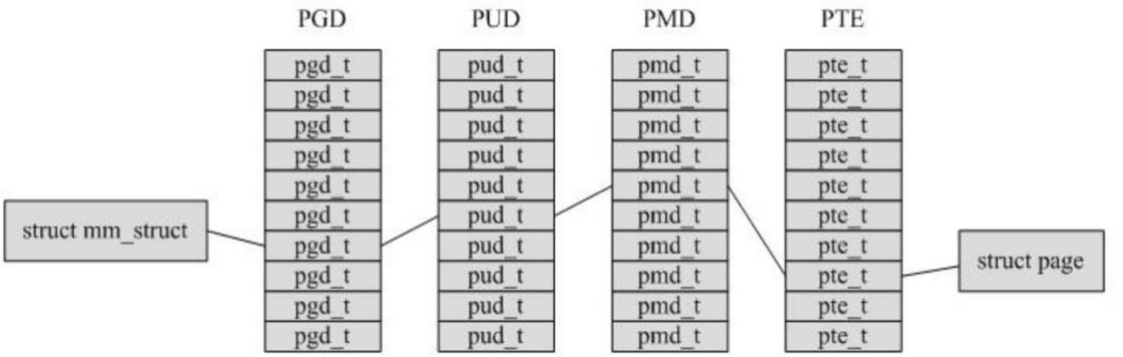
Linux 内核的物理内存管理是通过分页机制实现的，它将整个内存划分成无数个 4K 大小的页，分配和回收内存的基本单位为内存页。利用分页管理可以灵活地分配内存地址，如此在分配时不必要求必须有大块的连续内存，系统可以组合零散的内存供进程使用。

物理页在系统中由页结构 struct page 描述，系统中所有的页面都存储在数组 mem_map[]中，可以通过该数组找到系统中的每一页）。而其中的空闲页面则可由以伙伴关系组织的空闲页链表（free_area[MAX_ORDER]）来索引。

（4）虚拟内存到物理内存的映射

Linux 系统采用虚拟内存管理技术，使得每个进程都有各自互不干涉的进程地址空间。该空间是块大小为 4G 的线性虚拟空间，用户所看到和接触到的都是该虚拟地址，无法看到实际的物理内存地址。从用户向内核看，所使用的内存表象形式会依次经历“逻辑地址”——“线性地址”——“物理地址”几种形式。逻辑地址经过段机制转化成线性地址；线性地址又经过页机制转化为物理地址。

当应用程序访问一个虚拟地址时，必须将虚拟地址转化成物理地址，处理器才能解析地址访问请求。地址的转换工作通过查询页表来完成，简单理解就是地址转换需要将虚拟地址分段，使每段虚地址都作为一个索引指向页表，而页表项则指向下一级别的页表或者指向最终的物理页面。下图即为进程地址空间到物理页之间的转换关系：



（5）进程内存的分配与回收

内核中分配空闲页面的基本函数是 get_free_page/get_free_pages，它们或是分配单页或是分配指定的页面（2、4、8…512 页）。另外还有两个常见的分配方法：kmalloc 和 malloc，分别适用于小块内存的分配和大块内存的“连续”分配。具体的方法在后续介绍。

二、Linux 内存管理中的主要数据结构

1. 物理内存的数据结构

Linux 把物理内存划分为三个层次来管理，分别为：

（1）存储节点（Node）

在 Linux 操作系统中 CPU 被划分为多个节点(node)，内存则被分簇，每个 CPU 对应一个本地物理内存，即一个 CPU-node 对应一个内存簇 bank，即每个内存簇被认为是一个节点。

LINUX 中引入一个数据结构 struct pglist_data ，来描述一个 node，定义在文件目录：include/linux/mmzone.h 中，pglist_data 被 typedef 为 pg_data_t。

其中定义的内存管理域及其成员变量为：（截图为 struct pglist_data 的内存管理域）

```
typedef struct pglist_data {
        struct zone node_zones[MAX_NR_ZONES];
        struct zonelist node_zonelists[MAX_ZONELISTS];
        int nr_zones;
```

node_zones：该数组保存了节点中各个内存域 zone 的数据结构。

node_zonelists：备用节点及其内存域 zone 的列表，当当前节点的内存不够分配时，会选取访问代价最低的内存进行分配。

nr_zones：当前节点中不同内存域 zone 的数量，定义在 1 到 3 个之间。

定义的结点的内存页面及其重要成员变量：（截图为 struct pglist_data 的内存页面申明）

```
#ifdef CONFIG_FLAT_NODE_MEM_MAP /* means !SPARSEMEM */
        struct page *node_mem_map;
#ifdef CONFIG_CGROUP_MEM_RES_CTLR
        struct page_cgroup *node_page_cgroup;
#endif
#endif
unsigned long node_start_pfn;
unsigned long node_present_pages; /* total number of physical pages */
unsigned long node_spanned_pages; /* total size of physical page
                                     range, including holes */
int node_id;
```

node_mem_map：指向 page 实例数组的指针，用于描述结点的所有物理内存页，它包含了结点中所有内存域的页。

node_start_pfn：该 NUMA 结点的第一个页帧的逻辑编号。

node_present_pages：该结点中页帧的数目。

node_spanned_pages：该结点以页帧为单位计算的长度.

node_id：全局结点的 ID。

该结构中的其余变量则为其余结构所用，具体还有交换守护进程的相关变量等。

对于和节点 node 其他相关的结构和接口函数，还有：

结点状态标识：标记了内存结点所有可能的状态信息。

定义在：include /linux/nodemask.h。

查找内存结点：通过宏查找，见右图：

定义在：include/linux/mmzone.h。

```
#define for_each_online_pgdat(pgdat)
        for (pgdat = first_online_pgdat();
             pgdat;
             pgdat = next_online_pgdat(pgdat))
```

（2）管理区域（Zone）

在内存被划分为结点后，各个节点又被划分为内存管理区域 zone，用于表示不同范围的内存，内核可以使用不同的映射方式映射物理内存。

每一个管理区域通过 struct zone_struct 描述，其被定义为 zone_t，用以表示内存的某个范围，低端范围的 16MB 被描述为 ZONE_DMA，然后是可直接映射到内核的普通内存域 ZONE_NORMAL，最后是超出了内核段的物理地址域 ZONE_HIGHMEM，即高端内存。

首先分析一下内存管理区类型 zone_type，其定义在 include/linux/mmzone.h 中，全部由 #ifdef 语句构成，主要是在编译时根据不同的配置加入不同的管理区类型。

接下来分析管理区结构 zone，定义在 include/linux/mmzone.h 中。

其重要的成员函数为：

```
wait_queue_head_t       * wait_table;
unsigned long           wait_table_hash_nr_entries;
unsigned long           wait_table_bits;
```

wait_table：等待一个 page 来释放的等待队列哈希表。

wait_table_hash_nr_entries：哈希表中的等待队列的数量。

wait_table_bits：等待队列散列表数组大小。

另外，内核使用 pg_data_t 中的 zonelist 数组来表示所描述的层次结构。node_zonelists 数组对每种可能的内存域类型，都配置了一个独立的数组项，其定义在 include/linux/mmzone.h 中。

```
struct zonelist {
        struct zonelist_cache *zlcache_ptr;
        struct zoneref _zonerefs[MAX_ZONES_PER_ZONELIST + 1];
#ifdef CONFIG_NUMA
        struct zonelist_cache zlcache;
#endif
};
```

（3）页面（Page）

在节点和管理区域之下，内存被细分为多个页面帧，页面是系统内存的最小单元。

内核用 struct page 结构表示系统中的每个物理页，定义在：include/linux/mm_types.h 中，其中重要的成员函数有：（由于成员函数较为分散故不便插入截图）

flag：用来存放页的状态，每一位代表一种状态，可以同时表示出 32 中不同的状态。

virtual：高端内存区域的页 virtual 保存该页的虚拟地址。

_mapcount：被页表映射的次数，即该 page 同时被多少个进程共享。

index：在映射的虚拟空间（vma_area）内的偏移。

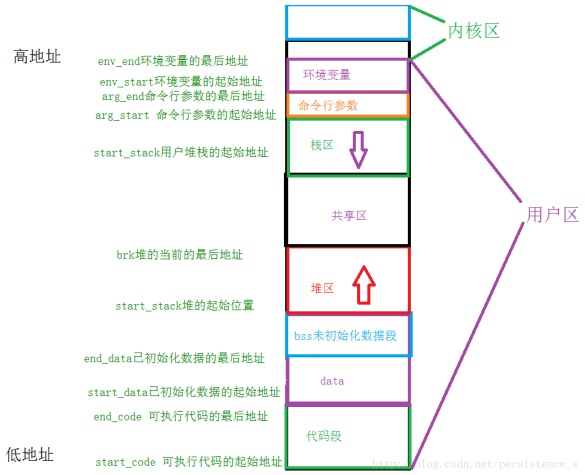
lru：链表头，用于在各种链表上维护该页，以便于按页将不同类别分组。

mapping：指向与该页相关的 address_space 对象。

2. 虚拟内存中的数据结构

（1）mm_struct 结构

mm_sruct 结构是内存描述符。每个进程的 task_struct 里均会有一个 struct mm_struct* 指向每个进程自己的 mm_struct，使得每个进程都有自己独立的虚拟的地址空间。在每个 mm_struct 中都有一个 pgd_t * 使其指向页表，通过此页表实现从虚拟地址到物理地址的映射。mm_struct 结构的用户控制分布图见右，主要由内核区和用户区组成，用户区即由五种基本数据段组成，



关于内核区的成员变量，主要搭建了第链表与红黑树两种形式并行地内存管理模式。其定义在 include/linux/mm_types.h 中：

```
struct mm_struct {
        struct vm_area_struct * mmap;           /* list of VMAs */
        struct rb_root mm_rb;
        struct vm_area_struct * mmap_cache;     /* last find_vma result */
#ifdef CONFIG_MMU
        unsigned long (*get_unmapped_area) (struct file *filp,
                                unsigned long addr, unsigned long len,
                                unsigned long pgoff, unsigned long flags);
        void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
#endif
        unsigned long mmap_base;                /* base of mmap area */
        unsigned long task_size;                /* size of task vm space */
        unsigned long cached_hole_size;         /* if non-zero, the largest
        unsigned long free_area_cache;          /* first hole of size cache
        pgd_t * pgd;
        atomic_t mm_users;                      /* How many users with user
        atomic_t mm_count;                      /* How many references to "
        int map_count;                          /* number of VMAs */

        spinlock_t page_table_lock;             /* Protects page tables and
        struct rw_semaphore mmap_sem;

        struct list_head mmlist;                /* List of maybe swapped mm
```

其中的重要成员变量为：

mmap：指向虚拟区间(VMA)的链表。

mm_rb：指向线性区对象红黑树的根。

mmap_cache：指向最近找到的虚拟区间。

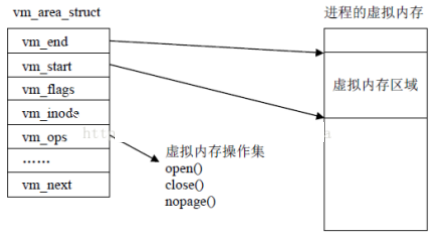
get_unmapped_area：在进程地址空间中搜索有效线性地址区。

free_area_cache：内核从这个地址开始搜索进程地址空间中线性地址的空闲区域。

list_head mmlist：指向内存描述符链表中的相邻元素

（2）vm_area_struct 结构

每个 vm_area_struct 代表进程的一个虚拟地址区间，vm_area_struct 之间构成链表。且每个进程的 mm_struct 中均有指向该结构的指针，该结构中存储了虚拟内存的起点与终点以及一系列虚拟内存的操作集。具体的对应关系如右图。



vm_area_struct 中重要成员变量意义如下：
vm_mm：指向虚拟内存的 mm_struct 指针。
vm_start：在虚拟内存中的起始地址。
vm_end：在虚拟内存中的结束地址。
vm_next：指向链表下一个单元的指针。
vm_prev：指向链表上一个单元的指针。
list：定义的一个虚拟内存操作集合。

```
struct vm_area_struct {
        struct mm_struct * vm_mm;       /* The address spa
        unsigned long vm_start;         /* Our start addre
        unsigned long vm_end;           /* The first byte
                                           within vm_mm. *

        /* linked list of VM areas per task, sorted by ada
        struct vm_area_struct *vm_next, *vm_prev;

        pgprot_t vm_page_prot;          /* Access permissi
        unsigned long vm_flags;         /* Flags, see mm.h

        struct rb_node vm_rb;

        /*
         * For areas with an address space and backing sta
         * linkage into the address_space->i_mmap prio tre
         * linkage to the list of like vmas hanging off it
         * linkage of vma in the address_space->i_mmap_non
         */
        union {
                struct {
                        struct list_head list;
                        void *parent;   /* aligns with pri
                        struct vm_area_struct *head;
                } vm_set;

                struct raw_prio_tree_node prio_tree_node;
        } shared;
```

三、Linux 内存管理的分配和收回

1. 伙伴算法

由于前文提到的 Linux 内核管理物理内存是通过分页机制实现的，因此其对内存的分配可以一页页不连续地依次分配。但是实际上系统使用内存时还是倾向于分配连续的内存块，因为分配连续内存时，页表不需要更改，因此能有效降低 TLB 的刷新率，保证进程的访问速度不受影响。鉴于上述需求，内核分配物理页面时为了尽量减少不连续情况，实际内存分配时采用了伙伴算法来管理空闲页面。

在 Linux 中采用伙伴算法把所有的空闲页框分为 11 个块链表，每块链表中分布包含特定的连续页框地址空间，比如第 0 个块链表包含大小为 2^0 个连续的页框，第 1 个块链表中，每个链表元素包含 2 个页框大小的连续地址空间，…，第 10 个块链表中，则每个链表元素代表 4M 的连续地址空间。每个链表中元素的个数在系统初始化时决定，在执行过程中，动态变化。伙伴算法每次只能分配 2 的幂次页的空间，比如一次分配 1 页，2 页，4 页，8 页，…，1024 页等，由于每页大小一般为 4K，所以伙伴算法最多一次能够分配 4M 的内存空间。

2. 实现伙伴算法的数据结构

为实现伙伴算法，在 include/linux/mmzone.h 中定义了 free_area 结构，其中 nr_free 代表当前对应大小空闲块的数目，并在 zone 中申明了该结构体的数组，其中 MAX_ORDER 对应空闲块的大小，例如当 MAX_ORDER=2 时，代表此时大小为 4 页的空闲块链表，所有大小为 4 页的空闲块构成双向链表，且链表内 nr_free 均相等。

```
struct free_area {
        struct list_head        free_list[MIGRATE_TYPES];
        unsigned long           nr_free;
};


    struct free_area            free_area[MAX_ORDER];
```

3. 内存空间的分配过程

有了 free_area 的结构，即可实现伙伴算法来分配内存空间，具体实现如下：

例如系统需要分配 4 页（16K）的内存空间，算法会先优从 free_area[2]中查看 nr_free 是否为空，如果有空闲块，则从中分配，如果没有空闲块，就从它的上一级 free_area[3]（每块 32K）中分配出 16K 内存空间，并将多余的内存（16K）加入到 free_area[2]中去，使得 free_area[2]里的 nr_free 变为 1。如果 free_area[3]也没有空闲，则再从更上一级申请空间，依次递推，直到 free_area[max_order]，如果顶级都没有空间，则就返回内存分配失败。

实际 Linux 的实现代码在 arch/s390/mm/page-states.c 中，代码如下：

其中额外加入了一个锁用以解决内存分配中的冲突问题，其余算法均和上面一样，使用 free_area[order]从初始分配大小不断用 for 循环向上尝试，直至完成内存的分配或搜索结束内存分配失败。

```
void arch_set_page_states(int make_stable)
{
        unsigned long flags, order, t;
        struct list_head *l;
        struct page *page;
        struct zone *zone;

        if (!cmma_flag)
                return;
        if (make_stable)
                drain_local_pages(NULL);
        for_each_populated_zone(zone) {
                spin_lock_irqsave(&zone->lock, flags);
                for_each_migratetype_order(order, t) {
                        list_for_each(l, &zone->free_area[order].free_list[t]) {
                                page = list_entry(l, struct page, lru);
                                if (make_stable)
                                        set_page_stable(page, order);
                                else
                                        set_page_unstable(page, order);
                        }
                }
                spin_unlock_irqrestore(&zone->lock, flags);
        }
}
```

4. 内存空间的释放过程

释放是分配的逆过程，故好内存申请几乎相同。当释放一个内存块时，先在其对于的 free_area 链表中查找是否有伙伴存在（伙伴块指两个内存块位码相邻且为伙伴位码）。如果没有伙伴块，直接将释放的块插入链表头。如果有伙伴块的存在，则将其从链表中取出，合并成一个大块，然后继续查找合并后的块在更大一级链表中是否有伙伴的存在，直至不能合并或者已经合并至最大块 1024 为止。

四、几种不同的内存分配函数

1. Kmalloc 函数

以页为最小单位分配内存对于内核管理系统中的物理内存来说的确比较方便，但内核自身最常使用的内存却往往是很小（远远小于一页）的内存块——比如存放文件描述符、进程描述符、虚拟内存区域描述符等行为所需的内存都不足一页，浪费了一定资源。

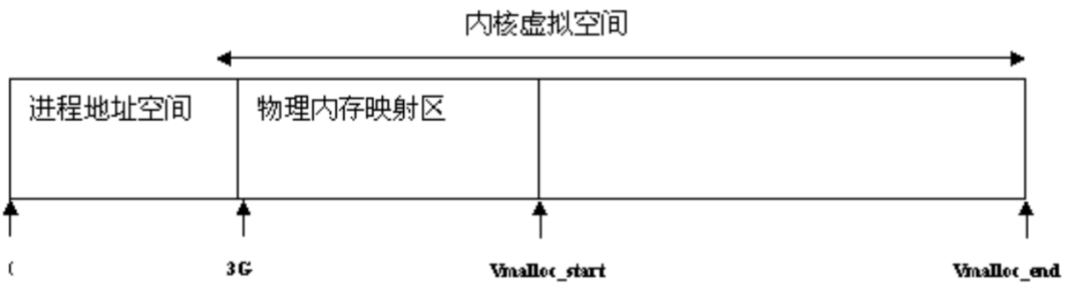
为了满足内核对这种小内存块的需要，Linux 系统采用了 Slab 分配器的技术。Slab 分配器的实现相当复杂，但原理不难，其核心思想就是“存储池”的运用。内存片段（小块内存）被看作对象，当被使用完后，并不直接释放而是被缓存到“存储池”里，留做下次使用，这无疑避免了频繁创建与销毁对象所带来的额外负载，提高了访问速度。

2. Vmalloc 函数

伙伴关系与 slab 技术从内存管理理论角度而言目的基本是一致的，都是为了防止“分片”。分片又分为外部分片和内部分片之说，所谓内部分片是说系统为了满足一小段内存区（连续）的需要，不得不分配了一大区域连续内存给它，从而造成了空间浪费；外部分片是指系统虽有足够的内存，但却是分散的碎片，无法满足对大块“连续内存”的需求。无论何种分片都是系统有效利用内存的障碍。

所以避免外部分片的最终思路还是落到了如何利用不连续的内存块组合成“看起来很大的内存块”——这里的情况很类似于用户空间分配虚拟内存，内存逻辑上连续，其实映射到并不一定连续的物理内存上。Linux 内核借用了这个技术，允许内核程序在内核地址空间中分配虚拟地址，同样也利用页表（内核页表）将虚拟地址映射到分散的内存页上。以此完美地解决了内核内存使用中的外部分片问题。内核提供 vmalloc 函数分配内核虚拟内存，该函数不同于 kmalloc，它可以分配较 Kmalloc 大得多的内存空间（可远大于 128K，但必须是页大小的倍数），但相比 Kmalloc 来说，Vmalloc 需要对内核虚拟地址进行重映射，必须更新内核页表，因此分配效率上要低一些（用空间换时间）。

与进程相似,内核也有一个名为 init_mm 的 mm_strcut 结构来描述内核地址空间，其中页表项 pdg=swapper_pg_dir 包含了系统内核空间（3G-4G）的映射关系。因此 vmalloc 分配内核虚拟地址必须更新内核页表，而 kmalloc 或 get_free_page 由于分配的连续内存，所以不需要更新内核页表。Vmalloc 更新内核页表如下：



进程空间地址分布从 0 到 3 G，从 3G 到 vmalloc_start 这段地址是物理内存映射区域（该区域中包含了内核镜像、物理页面表 mem_map 等等）。

3. 系统调用 mmap

系统调用 mmap()将一个文件或者其它对象映射进内存。文件被映射到多个页上，如果文件的大小不是所有页的大小之和，最后一个页不被使用的空间将会清零。其函数原型为：

void *mmap(void *start, size_t length, int prot, int flags,int fd, off_t offset);

成功执行时，mmap()返回被映射区的指针，munmap()返回 0。失败时，mmap()返回 MAP_FAILED[其值为(void *)-1]。mmap()常用于内存共享和内存映射。

参考文献:

[1] Abraham Silberschatz. 操作系统概念. 高等教育出版社, 2007.3.
[2]Linux 操作系统分析.陈香兰
[3] Linux 内核源代码简单分析
https://blog.csdn.net/wy19910326/article/details/7342082
[4]Linux 内核的整体架构简介
https://blog.csdn.net/changexhao/article/details/78321295