Linux 内存管理的主要数据结构分析

陈 绮
(华南热带农业大学，海南 海口 571737)

【摘 要】 Linux 的研究方兴未艾，对 Linux 源代码的全面剖析是很繁琐但同时又是很有意义的一项工程。本文以 Linux 内核 2.4.18 为蓝本，着重对 Linux 内存管理中的几个主要数据结构进行分析，让 Linux 爱好者从一个侧面了解 Linux 的内存管理实现。

【关键词】 Linux 内存 数据结构

一直以来，由于人们所需要的内存数目远远大于物理内存，人们设计出了各种各样的策略来解决此问题，其中最成功的是虚拟内存技术。每个进程的虚拟内存用一个 mm_struct 数据结构表示。这包括当前执行的映像的信息和指向一组 vm_area_struct 结构的指针。每一个 vm_area_struct 的数据结构都描述了内存区域的起始、进程对于内存区域的访问权限和对于这段内存的操作，通过这两个主要结构，能够把虚存段很好的组织和管理起来。但是，在操作系统运作过程中，经常会涉及到大量对象的重复生成、使用和释放问题，为提高对象的使用效率，改善整个系统的性能，内存管理采用了 slab 算法(关于内存管理中的主要算法将另具文章讨论)，从而大大提高了内存的利用率以及硬件缓存区级系统总线的利用率，为此引入了一些数据结构，其中较主要的有 kmem_cache_s 结构。

尽管 Linux 采用虚拟存储管理策略，有些操作仍然需要直接针对物理内存。例如，为刚创建的进程分配页目录，为装入进程的代码段分配空间，为 I/O 操作准备缓冲区等等。物理内存以页帧为单位，页帧的长度固定，等于页长，对 Intel CPU 缺省为 4KB。对每个物理页帧，内核定义 page 类型的 mem_map_t 数据结构进行描述。下面我们来看看所提到的这几个主要数据结构：

1、**虚拟内存的 mm_struct 结构：** (include/linux/sched.h)

每当新建一个进程，Linux 都为其分配一个 task_struct 结构，此结构中内嵌 mm_struct，有关用户进程中与存储有关的信息都包含在 mm_struct 中。

```
struct mm_struct {
    struct vm_area_struct * mmap;   /* 指向 VMA 段双向链表的指针 */
    rb_root_t mm_rb;                /*指向 VMA 段红黑树的指针 */
    struct vm_area_struct * mmap_cache;
    /* 存储上次对 vma 块的查找操作的结果 */
    pgd_t * pgd;                    /*进程页目录的起始地址 */
    atomic_t mm_users; /* 记录目前正在使用此 mm_struct 结构的用户个数 */
    atomic_t mm_count; /* mm_struct 被内核线程引用次数 */
    int map_count;                  /* 进程所使用的 VMA 块的个数 */
    struct rw_semaphore mmap_sem;
    /*对 mmap 操作的互斥信号量，由 down()和 up()更改*/
    spinlock_t page_table_lock;   /*对此进程的页表操作时所需要的自旋锁*/
    struct list_head mmlist;        /* task_struct 中的 active_mm 域的链表 */
    unsigned long start_code, end_code, start_data, end_data;
/* 进程代码段的起始地址和结束地址、进程数据段的起始地址和结束地址*/
    unsigned long start_brk, brk, start_stack;
    /* 进程未初始化的数据段的起始地址和结束地址 */
    unsigned long arg_start, arg_end, env_start, env_end;
    /* 调用参数的起始地址和结束地址 */
    unsigned long rss, total_vm, locked_vm;
    /* rss 进程内容驻留在物理内存的页面总数 */
    unsigned long def_flags; unsigned long cpu_vm_mask; unsigned long swap_address;
    /* 页面换出过程中用到的交换空间地址 */
    unsigned dumpable:1; mm_context_t context;
    /* 存放当前进程使用的段起始地址 */
};
```

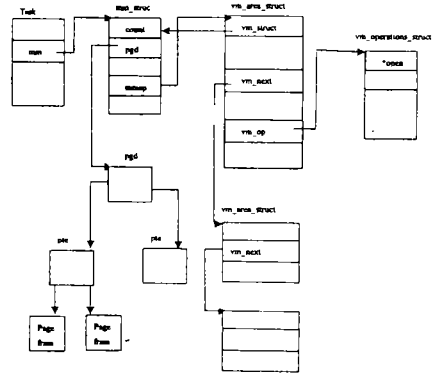
此 mm_struct 结构包含一个进程虚存空间的基本面描述和相关的信息记录，它是后续虚存段管理的基础。

2、**虚存段的数据结构：** (include/linux/mm.h)

进程可用虚存空间共有 4GB，但这 4GB 空间并不是可以让用户态进程任意使用的，只是 0 至 3GB 之间的那一部分可以被直接使用，剩下的 1GB 空间则是属于内核的，用户态进程不能直接访问到。所有进程的 3GB 至 4GB 的虚存空间的映像都是相同的，以此方式共享核心的代码段和数据段。一个进程在运行过程中使用到的物理内存一般是不连续的，用到的虚拟地址也不是连成一片的，而是被分成几块，进程通常占用几个虚存段，分别用于代码段、数据段、堆栈段等。每个进程的所有虚存段通过指针构成链表，虚存段在此链表中的排列顺序按照它们的地址增长顺序进行。Linux 定义了虚存段 vma，即 virtual memory area。vma 段是属于某个进程的一段连续的虚存空间，在这段虚存里的所有页面拥有一些相同的特征，例如，属于同一进程，相同的访问权限，同时被锁定(locked)，同时受保护(protected)等。vma 段由数据结构 vm_area_struct 描述如下：

```
struct vm_area_struct {
    struct mm_struct * vm_mm;
    /* 同一个进程的所有 VMA 块用这个指针指向此进程的 mm_struct 结构 */
    unsigned long vm_start;   /* VMA 描述的虚拟内存段起始地址 */
    unsigned long vm_end;     /* VMA 描述的虚拟内存段结束地址 */
    struct vm_area_struct *vm_next;
    /* 进程所使用的按地址排序的 vm_area 链表指针 */
    pgprot_t vm_page_prot;    /* 此变量指示本 VMA. 块中所有页面的保护模式 */
    unsigned long vm_flags;   /* 本 VMA 块的属性标志位 . */
    rb_node_t vm_rb;          /* 用于对 VMA 块进行 rb(Red Black Tree) 操作的
结构体，其定义位置在 include/linux/rbtree.h, line 109 */
    struct vm_area_struct *vm_next_share;
    /* 指向共享块链表中的前一个 VMA 块的指针 */
    struct vm_area_struct **vm_pprev_share;
    /* 指向共享块链表中的后一个 VMA 块的指针 */
    struct vm_operations_struct * vm_ops;
    /* 指向一个结构体的指针，该结构体中是对 VMA 段进行操作的函数指针
的集合。参见 include/linux/mm.h 中的 line 130 struct vm_operations_struct. */
    unsigned long vm_pgoff;
    /* Offset (within vm_file) in PAGE_SIZE units, *not* PAGE_CACHE_SIZE */
    struct file * vm_file;
/* 如果此 VMA 段是对某个文件的映射，vm_file 为指向这个文件结构的指针 */
    unsigned long vm_raend;   /* XXX: put full readahead info here. */
    void * vm_private_data;   /* was vm_pte (shared mem) */
};
```

当一个执行映像映射到进程的虚拟地址空间时，产生一组 vm_area_struct 数据结构。每一个 vm_area_struct 结构表示执行映像的一部分：执行代码、初始化数据（变量）、未初始化数据等等。Linux 支持一系列标准的虚拟内存操作，当 vm_area_struct 数据结构创建时，一组正确的虚拟内存操作就和它们关联在一起。(参见下图)



进程的虚存管理数据结构

3、**对象缓存中的 kmem_cache_s:** (linux/mm/slab.c)

Linux 系统中所用到的对象，一般说来种类相对稳定（如 inode、task_struct 等），然而每类对象的数量却是大量的，并且在初始化与析构时要做大量的工作，所占时间比例大大超过内存分配所占用的时间，但是这些对象往往具有这样的一个性质，即它们在生成时，所包括的成员属性值一般都赋成确定的数值，并且在使用完毕，释放结构前、这些成员属性又恢复为未使用前的状态。为此，内存管理中引入了 slab 算法，以解决对象重复生成、使用和释放问题。Slab 算法思路中最基本的一点被称为 object - caching，即对象缓存，其中一个重要数据结构如下：

```
struct kmem_cache_s {
/* 1) each alloc & free */
        /* full, partial first, then free */
        struct list_head   slabs_full;      /* slab 中的完全块 */
        struct list_head   slabs_partial;   /* slab 中的部分块 */
        struct list_head   slabs_free;      /* slab 中的空闲块 */
        unsigned int       objsize;         /* 此 slab 块中对象的大小 */
        unsigned int       flags;           /* 属性标志 */
        unsigned int       num;             /* 此缓存区中的每个 slab 中的对象个数 */
        spinlock_t         spinlock;        /* 互斥锁。所有的 cache 都位于一条链表上，在访问 cache 时使用这个互斥信号量来避免同时访问这条链表。*/
#ifdef CONFIG_SMP
        unsigned int       batchcount;
#endif
/* 2) slab additions /removals */
        unsigned int       gfporder;        /* 该缓存区中每个 slab 块中包含页面个数相对于 2 的幂次方 */
        unsigned int       gfpflags;        /* 申请页面时用到的申请优先数 */
        size_t             colour;          /* 染色计数器可以达到的最大值 */
        unsigned int       colour_off;      /* 缓存区中 slab 块之间的相对距离 */
        unsigned int       colour_next;     /* 染色计数器 */
        kmem_cache_t       *slabp_cache; /* 对于 off slab 方式此指针指向一块公共的 cache_slabp 缓存区，此缓存区用以存放与每个 slab 块相对应的 slab_t 结构。*/
        unsigned int       growing;         /* 正在对此缓存区增加新的 slab 块的时候设置此标志，用于防止同时对此 slab 块的收缩操作。 */
        unsigned int       dflags;          /* 动态属性标志 */
        void ( *ctor)(void *, kmem_cache_t *, unsigned long);
                                            /* 对象的构造函数，在创建新的 slab 块之后给这个块中的每个对象调用此构造函数 */
        void ( *dtor)(void *, kmem_cache_t *, unsigned long);
                                            /* 对象的析构函数 */
        unsigned long      failures;
/* 3) cache creation/removal */
        char               name[CACHE_NAMELEN]; /* 此缓存区的名字。它将出现在/proc/slabinfo 文件中。 */
        struct list_head   next;            /* 指向下一个缓存区结构的指针 */
#ifdef CONFIG_SMP
/* 4) per-cpu data */                       /* 以下是对多处理器的支持部分，注释略 */
        cpucache_t         *cpudata[NR_CPUS];
#endif
#if STATS
        unsigned long      num_active;
        unsigned long      num_allocations;
        unsigned long      high_mark;
        unsigned long      grown;
        unsigned long      reaped;
        unsigned long      errors;
#ifdef CONFIG_SMP
        atomic_t           allochit;
        atomic_t           allocmiss;
        atomic_t           freehit;
        atomic_t           freemiss;
#endif
#endif
    };
```

在 slab 算法中，一种对象的所有实例都存在同一个缓存区中。不同的对象，即使大小相同，也放在不同的缓存区中。每一缓存区有若干个 slab 块，按照满、半满、空的顺序排列。Slab 块是内核内存分配与页面级分配的接口。每个 slab 块的大小都为页面大小的整数倍，由若干对象组成。引入 slab 块的目的之一就是通过染色机制使 slab 块在硬件缓存区中均匀分布，从而也使对象的分布达到均匀状态，以达到提高硬件缓存区级系统总路线的利用率。在这里，所谓的染色机制是指，按照对象要求的对齐字节数，分配合适的着色区，（即偏移量），以使该类对象有良好的地址分布，便于硬件操作。

4、物理空间管理中的 mem_map_t：（include/linux/mm.h）

Linux 对整个物理内存的管理通过 mem_map 表描述。它本身是关于 struct page mem_map_t（如下结构）的数组，每项对应一个关于内核态、用户态代码和数据等的页帧。

```
typedef struct page {
        struct list_head list; /* 这个 list 用来将这页挂在不同用途的链表中 */
        struct address_space *mapping;
            /* 当此页帧的内容是文件时，由这个 mapping 所指的 address_space 型变量记录了 inode cache(也就是 page 缓存区)链表的表头指针 . */
        unsigned long index;            /* 当此页被换出到交换空间中时，这个 index 对其在交换空间中的位置进行索引 */
        struct page *next_hash;         /* 用于 page 缓存区中链表元素的连接 */
        atomic_t count;                 /* 指明目前使用该页面的用户数 */
        unsigned long flags;            /* 此变量中的不同的位描述此页属性 */
        struct list_head lru;           /* Pageout list, eg. active_list; protected by pagemap_lru_lock !! */
        struct page **pprev_hash;       /*  *next_hash. 的辅助指针 */
        struct buffer_head * buffers;   /* 映射磁盘块的缓冲区 */
        /*
        * On machines where all RAM is mapped into kernel address space,
        * we can simply calculate the virtual address. On machines with
        * highmem some memory is mapped into kernel virtual memory
        * dynamically, so we need a place to store that address.
        * Note that this field could be 16 bits on x86 ... ;)
        *
        * Architectures with slow multiplication can define
        * WANT_PAGE_VIRTUAL in asm/page.h
        */
#if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)
        void *virtual;                  /* Kernel virtual address (NULL if not kmapped, ie. highmem) */
#endif /* CONFIG_HIGMEM || WANT_PAGE_VIRTUAL */
} mem_map_t;
```

在一个虚拟内存系统中，所有的地址都是虚拟地址而非物理地址。处理器通过操作系统保存的一组信息将虚拟地址转换为物理地址，这可以通过页表来实现。这部分的知识在操作系统课程里有专门的介绍。

Linux 内存管理中所用到的数据结构当然不止这几个，但这些是比较重要，有的也比较不易理解，其它还有一些数据结构，相对比较简单，并与上面所提的这些数据结构联系紧密，感兴趣的读者可以参看其源代码。

参考文献

[1] 《Linux 内核 2.4 版源代码分析大全》，李善平等著，2002 年 1 月机械工业出版社
[2] 《现代操作系统教程》，滕至阳著，2000 年 5 月高等教育出版社