第 7卷 第 12期 2007年 6月
1671-1819(2007)12-2849-03
科 学 技 术 与 工 程
Science Technology and Engineering
Vol. 7 No. 12 June 2007

Linux内核内存池实现研究

刘 磊
(西北工业大学, 西安 710065)

摘 要 回顾了 Linux内核内存管理发展历程, 比较了早期的 Linux内存管理与最新的 Linux kernel 2.6.16内存管理的联系和差别。分析 Linux最新版本的内核内存管理策略, 重点研究了最新的 Linux 2.6.16版本内核中的内核内存池的实现。

关键词 Linux内核 内存池 存储管理
中图法分类号 TP332.11; 文献标志码 A

Linux早期的内存管理由 get_free_page()、free_page()、free_page_tables()、copy_page_tables()、put_page()、do_wp_page()、do_no_page()和 get_empty_page()这几个定义在 memory.c中的函数来实现, 并用 mem_map[]表对页面进行管理[1]。为了支持虚拟存储器, Linux系统采用 LRU分页替换算法载入进程。虚拟存储器由存储器管理机制及一个大容量的快速硬盘存储器支持, 它的实现基于局部性原理。早期的 Linux内存管理虽然没有 2.6.16版本复杂, 但已经奠定了写时复制、需求加载等内存管理的基本思想。Linux是针对有 MMU的处理器设计的, 对具有 MMU的处理器而言, 虚拟地址被送到内存管理单元 MMU, 把虚拟地址映射为物理地址。通过赋予每个任务不同的虚拟地址到物理地址的转换映射, 支持不同任务之间的保护。每个用户进程 4 GB长度的虚拟内存被划分成固定大小的页面。其中 0至 3 GB是用户态空间, 由各进程独占; 3 GB到 4 GB是内核态空间, 由所有进程共享, 但只有内核态进程才能访问。

1 Linux内存池的实现

Linux2.6内存页面有 page数据结构管理, 全局数组 mem_map[]用于存储 page页面结构。在操作系统的运行过程中, 经常会有到大量对象的重复生成、使用和释放问题。对象生成算法的改进, 可以在很大程度上提高整个系统的性能。这些对象往往在生成时, 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, 这些属性又恢复为未使用前的状态。因此, 如果我们能够用合适的方法使得在对象前后两次背使用时, 在同一块内存, 或同一类内存空间, 且保留了基本的数据结构, 就可以大大提高效率[2]。

1.1 内存池的数据结构

Linux内存池是在 2.6版内核中才有的, 数据结构定义在 include/linux/mempool.h中[3]。其中内存池的数据结构如下所示意:

```
typedef struct mempool_s {
    spinlock_t lock;
    int min_nr;    /* elements数组中的成员数量 */
    int curr_nr;   /* 当前 elements数组中空闲的成员数量 */
    void **elements;
    void *pool_data;
    mempool_alloc_t *alloc;
    mempool_free_t *free;   /* 内存释放函数, 其它同上 */
    wait_queue_head_t wait;   /* 任务等待队列 */
} mempool_t;
```

结构成员 elements用来存放内存成员的二维数组, 其长度为 min_nr, 宽度是上述各个内存对象的长度, 因为对于不同的对象类型, 我们会创建相应的内存池对象, 所以每个内存池对象实例的 element宽度都是跟其内存对象相关的。pool_data这个指针通常是指向这种内存对象对应的缓存区的指针。结构成员 alloc是用户在创建一个内存池对象时提供的内存分配函数, 这个函数可以用户编写, 也可以

2006年 12月 15日收到

采用内存池提供的分配函数。

1.2 内存池的创建

内存池由 mempool_create()函数创建,其函数定义为:

```
mempool_t * mempool_create(int min_nr, mempool_alloc_t * alloc_fn,
        mempool_free_t * free_fn, void * pool_data)
{
    return mempool_create_node(min_nr, alloc_fn, free_fn, pool_data,
    -1);
}
```

其中 min_nr是池中最少可以分配的对象数、alloc_fn由用户定义的对象分配函数名 free_fn由用户定义的释放对象函数名、pool_data是用户定义对象分配函数可以访问的可选私有数据。这个函数生成并分配一个一定大小并事先已经分配的内存池,该内存池是 mempool_alloc()函数和 mempool_free()函数的参数之一。 mempool_create_node()定义如下:

```
mempool_t * mempool_create_node(int min_nr, mempool_alloc_t * al-
loc_fn,
                    mempool_free_t * free_fn, void * pool_data,
                    int node_id)
{
    mempool_t * pool;
    pool = kmalloc_node(sizeof(* pool), GFP_KERNEL, node_id);
    if (! pool)
        return NULL;
    memset(pool, 0, sizeof(* pool));
    pool->elements = kmalloc_node(min_nr * sizeof(void * ),
                    GFP_KERNEL, node_id);
    if (! pool->elements) {
        kfree(pool);
        return NULL;
    }
    spin_lock_init(&pool->lock);
    pool->min_nr = min_nr;
    pool->pool_data = pool_data;
    init_waitqueue_head(&pool->wait);
    pool->alloc = alloc_fn;
    pool->free = free_fn;
    /* 产生特定大小的缓冲区 */
    while (pool->curr_nr < pool->min_nr) {
        void * element;
        element = pool->alloc(GFP_KERNEL, pool->pool_data);
        if (unlikely(! element)) {
            free_pool(pool);
            return NULL;
        }
        add_element(pool, element);
    }
    return pool;
}
```

1.3 内存池的使用

如果需要使用已经创建的内存池,则需要调用 mempool_alloc从内存池中申请内存以及调用 mempool_destroy()将用完的内存还给内存池。

```
void * mempool_alloc(mempool_t * pool, gfp_t gfp_mask)
{
    void * element;
    unsigned long flags;
    wait_queue_t wait;
    gfp_t gfp_temp;
    might_sleep_if(gfp_mask & __GFP_WAIT);
    gfp_mask |= __GFP_NOMEMALLOC;   /* don't allocate emer
    gency reserves */
    gfp_mask |= __GFP_NORETRY;   /* don't loop in __alloc_pa
    ges */
    gfp_mask |= __GFP_NOWARN;  /* failures are OK */
    gfp_temp = gfp_mask & ~(__GFP_WAIT | __GFP_IO);
repeat_alloc:
    element = pool->alloc(gfp_temp, pool->pool_data);
    if (likely(element != NULL))
        return element;
    spin_lock_irqsave(&pool->lock, flags);
    /* 内存池不为空,则从池中获取一个内存对象,返回给申请
    者 */
    if (likely(pool->curr_nr)) {
        element = remove_element(pool);
        spin_unlock_irqrestore(&pool->lock, flags);
        return element;
    }
    spin_unlock_irqrestore(&pool->lock, flags);
    /* 在 GFP_ATOMIC事件发生时,不能睡眠 */
    if (! (gfp_mask & __GFP_WAIT))
        return NULL;
    /* 重新声明页面 */
    gfp_temp = gfp_mask;
    init_wait(&wait);
    prepare_to_wait(&pool->wait, &wait, TASK_UNINTERRUPT-
    IBLE);
    smp_mb();
    if (! pool->curr_nr)
        io_schedule();
    finish_wait(&pool->wait, &wait);
```

```
    goto repeat_alloc;
}
```

参数 pool指向有 mempool_create()函数分配的内存池、gfp_mask为通用分配位掩码。如果申请者调用 mempool_free准备释放内存,实际上是将内存对象重新放到内存池中[4],源码如下:

```
void mempool_destroy(mempool_t * pool)
{
    if(pool->curr_nr != pool->min_nr)
        BUG();    /* There were outstanding elements */
    free_pool(pool);
}
void mempool_free(void * element,mempool_t * pool)
{
    unsigned long flags;
    mb();
    /* 如果当前内存池已满,则调用用户内存释放函数将内存还给系统 */
    if(pool->curr_nr < pool->min_nr) {
        spin_lock_irqsave(&pool->lock,flags);
        if(pool->curr_nr < pool->min_nr) {
            /* 如果内存池还有剩余的空间,则将内存对象放入池中,唤醒等待队列 */
            add_element(pool,element);
            spin_unlock_irqrestore(&pool->lock,flags);
            wake_up(&pool->wait);
            return;
        }
        spin_unlock_irqrestore(&pool->lock,flags);
    }
    pool->free(element,pool->pool_data);
}
```

2 Linux内存池实现总结

Linux内核的内存管理实现了二级分配机制,即如果用户申请的内存大于预定义的级别,则直接调用 malloc从堆中分配内存。而如果申请的内存大小在 128字节以内,则从最相近的内存大小中申请,如果该组的内存储量小于一定的值,就会根据算法,再次从堆中申请一部分内存加入内存池,保证内存池中有一定量的内存可用。 Linux的内存池与特定内存对象相关联,每一种内存对象都有其特定的大小以及初始化方法,内核根据实际的对象的大小来确定池中对象的大小。内核内存池初化始时从缓存区申请一定量的内存块,需要使用时从池中顺序查找空闲内存块并返回给申请者。回收时直接将内存插入池中,如果池已经满,则直接释放。内存池没有动态增加大小的能力,如果内存池中的内存耗尽,则直接从缓存区申请内存,内存池的容量不会随着使用量的增加而增加。

参 考 文 献

1 赵 炯. Linux内核完全剖析. 北京: 机械工业出版社, 684—685

2 http://www.linux.org Linux kernel 2.6.16

3 Bovet D P, Cesati M. Understanding the Linux kernel. 北京: 东南大学出版社, 294—342

4 Gorman M. 深入理解 Linux虚拟内存管理. 白 洛, 李俊奎, 刘森林. 北京: 北京航空航天大学出版社, 30—138

Study of the Implementation of the Memory Pool in Linux Kernel

LIU Lei

(Northwest Polytechnical University, Xi'an 710065, P. R. China)

[Abstract] The development of the memory management of the Linux kernel is reviewed. The difference of memory management between the last Linux kernel and the latest Linux kernel 2.6.16 is discovered here by comparing the old kernel and the latest kernel. The emphasis is to study the immplementation of the memory pool in the latest Linux kernel 2.6.16.

[Key words] Linux kernel memory pool memory management