

Linux操作系统分析

Chapter 8 内存管理

陈香兰 (xlanchen@ustc.edu.cn)

计算机应用教研室@计算机学院
嵌入式系统实验室@苏州研究院
中国科学技术大学
Fall 2014

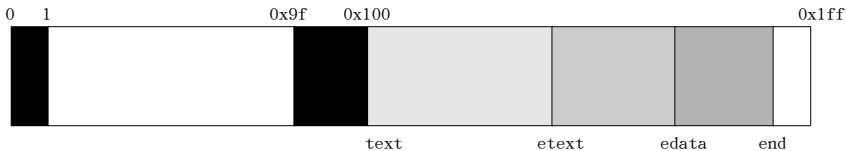
October 31, 2014






- 1 页框管理：页面级分配器
 - 页框和页描述符
 - 存储区(Memory Zones)
 - 页面级分配器接口
 - 页框管理算法：伙伴算法
- 2 内存区管理(memory area)
 - slab分配器
- 3 非连续存储区管理
- 4 小结和作业

动态存储器

- 在前面已经提到，Linux如何有效地利用x86的分段和分页机制把逻辑地址转换为物理地址
- RAM的某些部分永久地分配给内核，用以存放内核代码以及静态数据，这部分RAM在Linux运行期间不会被释放，也不能被分配用作其他用途
- RAM的其余部分称为**动态存储器 (dynamic memory)**

- Linux内核的前512个页框（假定内核所需内存<1MB，from ULK2）



-  Unavailable page frames
-  Available page frames
-  Kernel code
-  Initialized kernel data
-  Uninitialized kernel data

动态存储器

- 进程和内核的运行都需要动态存储器
- 属于稀缺资源
- 整个系统的性能取决于如何有效地管理动态存储器
 - 对动态存储器的使用要进行优化
 - 原则：对于动态存储器要尽可能做到
 - 按需分配，不需要时释放

- 内核如何给自己分配动态存储器
 - ① 页框管理
 - ② 小内存管理
 - ③ 非连续存储区管理

- 1 页框管理：页面级分配器
 - 页框和页描述符
 - 存储区(Memory Zones)
 - 页面级分配器接口
 - 页框管理算法：伙伴算法
- 2 内存区管理(memory area)
- 3 非连续存储区管理
- 4 小结和作业

Outline

- 1 页框管理：页面级分配器
 - 页框和页描述符
 - 存储区(Memory Zones)
 - 页面级分配器接口
 - 页框管理算法：伙伴算法
- 2 内存区管理(memory area)
 - slab分配器
- 3 非连续存储区管理
- 4 小结和作业

- Linux采用页作为内存管理的基本单位
- Linux采用的标准的页框大小为4KB
 - 由分页单元引发的缺页异常很容易得到解释
 - 4KB是大多数磁盘块大小的倍数
 - 传输效率高
 - 管理方便

```
/* PAGE_SHIFT determines the page size */  
#define PAGE_SHIFT      12  
#define PAGE_SIZE      (_AC(1,UL) << PAGE_SHIFT)  
#define PAGE_MASK      (~(PAGE_SIZE-1))
```

- 例如：512M的物理内存对应于???个页框？

- 内核必须记录每个页框的当前状态
 - 哪些属于进程，哪些存放了内核代码/数据
 - 对于动态存储器中的页框：是否空闲，即是否可用
 - 如果一个页框不可用，内核需要知道是谁在用这个页框
 - 用户态进程、动态分配的内核数据结构、静态的内核代码、页面cache、设备驱动程序缓冲的数据等等

页描述符

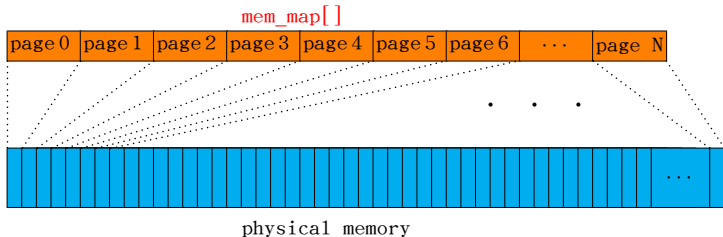
- 内核使用**页描述符**来跟踪管理物理内存
 - 每个物理页框都用一个页描述符表示
 - 页描述符用**struct page**的结构描述

include/linux/mm_types.h

```
/*
 * Each physical page in the system has a struct page associated with
 * it to keep track of whatever it is we are using the page for at the
 * moment. Note that we have no way to track which tasks are using
 * a page, though if it is a pagecache page, rmap structures can tell us
 * who is mapping it.
 */
struct page {
    unsigned long flags;    /* Atomic flags, some possibly
                           * updated asynchronously */
    atomic_t _count; /* Usage count, see below. */
    ...
    struct list_head lru;    /* Pageout list, eg. active_list
                           * protected by zone->lru_lock ! */
    ...
}
```

页描述符

- 所有物理页框描述符以（物理页号）自然序组织在`mem_map`数组中



mm/memory.c

```
#ifndef CONFIG_NEED_MULTIPLE_NODES
/* use the per-pgdat data instead for discontigmem - mbligh */
unsigned long max_mapnr;
struct page *mem_map;

EXPORT_SYMBOL(max_mapnr);
EXPORT_SYMBOL(mem_map);
#endif
```

页描述符

- mem_map数组的空间分配和初始化，mm/page_alloc.c

```
static void __init_refok alloc_node_mem_map(struct pglist_data *pgdat) {
    /* Skip empty nodes */
    if (!pgdat->node_spanned_pages)
        return;

#ifdef CONFIG_FLAT_NODE_MEM_MAP
    /* ia64 gets its own node_mem_map, before this, without bootmem */
    if (!pgdat->node_mem_map) {
        unsigned long size, start, end;
        struct page *map;

        /*
         * The zone's endpoints aren't required to be MAX_ORDER
         * aligned but the node_mem_map endpoints must be in order
         * for the buddy allocator to function correctly.
         */
        start = pgdat->node_start_pfn & ~(MAX_ORDER_NR_PAGES - 1);
        end = pgdat->node_start_pfn + pgdat->node_spanned_pages;
        end = ALIGN(end, MAX_ORDER_NR_PAGES);
        size = (end - start) * sizeof(struct page);
        map = alloc_remap(pgdat->node_id, size);
    }
#endif
}
```

页描述符

```
    if (!map)
        map = alloc_bootmem_node(pgdat, size);
    pgdat->node_mem_map = map + (pgdat->node_start_pfn - start);
}
#endif CONFIG_NEED_MULTIPLE_NODES
/*
 * With no DISCONTIG, the global mem_map is just set as node 0's
 */
if (pgdat == NODE_DATA(0)) {
    mem_map = NODE_DATA(0)->node_mem_map;
...
}
#endif
#endif /* CONFIG_FLAT_NODE_MEM_MAP */
}
```

页描述符

- 页描述符中的各个字段将在用到时再逐步介绍，首先看一下
 - count：页的使用引用计数器
 - 0：空闲
 - >0：页已经分配给一个或多个进程或用户某些内核数据结构
 - flags：页框状态，最多可以有32个，每个使用一个位表示

页描述符

页框状态：include/linux/page-flags.h

```
enum pageflags {
    PG_locked, /* Page is locked. Don' t touch. */
    PG_error,
    PG_referenced,
    PG_uptodate,
    PG_dirty,
    PG_lru,
    PG_active,
    PG_slab,
    PG_owner_priv_1, /* Owner use. If pagecache, fs may use*/
    PG_arch_1,
    PG_reserved,
    PG_private, /* If pagecache, has fs-private data */
    PG_writeback, /* Page is under writeback */
#ifdef CONFIG_PAGEFLAGS_EXTENDED
    PG_head, /* A head page */
    PG_tail, /* A tail page */
#else
    PG_compound, /* A compound page */
#endif
    PG_swapcache, /* Swap page: swp_entry_t in private */
    PG_mappedtodisk, /* Has blocks allocated on-disk */
    PG_reclaim, /* To be reclaimed asap */
    PG_buddy, /* Page is free, on buddy lists */
#ifdef CONFIG_IA64_UNCACHED_ALLOCATOR
    PG_uncached, /* Page has been mapped as uncached */
#endif
    __NR_PAGEFLAGS
};
```


- 页描述符将会占用很大的一段空间
 - 由于每个struct page结构小于64个字节，因此1MB的内存需要使用约4个页面来存放mem_map数组

$$\frac{1\text{MB}}{4\text{KB每页}} \times \frac{64\text{B每页}}{4\text{KB每页}} = 4\text{页}$$

- 假若系统中存在512MB的内存，那么大约需要??

- 本课程不考虑NUMA
- 物理内存被划分为若干个node
- 存取时间不等
- 考虑CPU局部性
- Node使用数据结构pg_data_t描述
- 每个node被划分成若干个zone

Outline

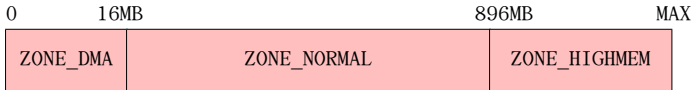
- 1 页框管理：页面级分配器
 - 页框和页描述符
 - 存储区(Memory Zones)
 - 页面级分配器接口
 - 页框管理算法：伙伴算法
- 2 内存区管理(memory area)
 - slab分配器
- 3 非连续存储区管理
- 4 小结和作业

存储区(Memory Zones)

- 在一个理想的体系结构中，一个页框就是一个物理存储单元，可以用于任何事情，例如
 - 存放内核数据/用户数据/缓存磁盘数据等
- 实际上存在硬件制约：一些页框由于自身的物理地址的原因不能被一些任务所使用，例如
 - ISA总线的DMA控制器只能对ram的前16M寻址
 - 在一些具有大容量ram的32位计算机中，CPU不能直接访问所有的物理存储器，因为线性地址空间不够
- 为了应付这种限制，Linux把具有同样性质的物理内存划分成——**区(zones)**

存储区 (Memory Zones)

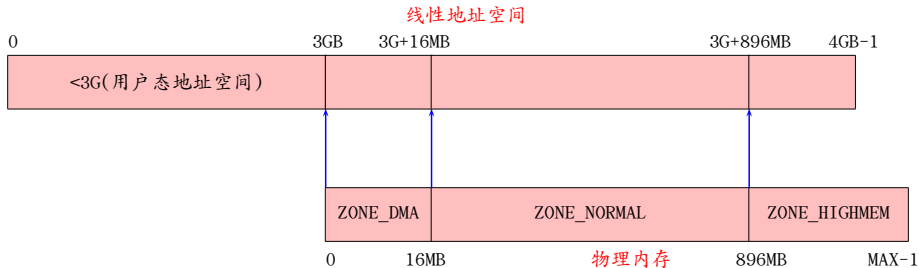
- 根据不同的体系结构和配置信息，zone的划分有所不同
 - 具体参见枚举类型zone_type
- 在32位x86中，一般有下列4个zone
 - ① ZONE_DMA (0-16MB)
 - ② ZONE_NORMAL (16MB-896MB)
 - ③ ZONE_HIGHMEM (>896MB)
 - ④ ZONE_MOVABLE (参见Documentation/memory-hotplug.txt)



物理内存

存储区 (Memory Zones)

- ZONE_DMA 和 ZONE_NORMAL 区包含存储器的“常规”页，通过把它们映射到线性地址空间的3GB以上，内核就可直接访问
- 而 ZONE_HIGHMEM 区中包含的存储器页面不能由内核直接访问



存储区(Memory Zones)

- 每个zone使用struct zone表示，记录了zone中内存相关的各种信息，包括zone的
 - 物理位置信息、
 - 空闲内存情况、
 - 内存使用和回收情况
 - 等等
 - 具体参见include/linux/mmzone.h::struct zone

```
struct zone {
    /* Fields commonly accessed by the page allocator */
    unsigned long pages_min, pages_low, pages_high;
    ...
    /*
     * free areas of different sizes
     */
    spinlock_t lock;
    ...
    struct free_area free_area[MAX_ORDER];
    ...
}
```

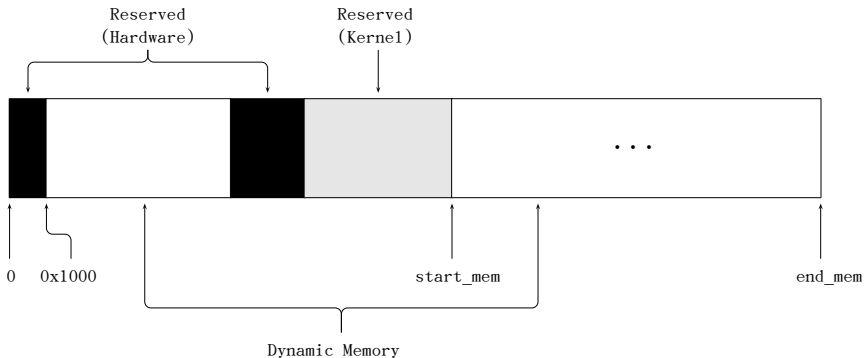
存储区(Memory Zones)

```
...
/* Fields commonly accessed by the page reclaim scanner */
spinlock_t lru_lock;
struct list_head active_list;
struct list_head inactive_list;
unsigned long nr_scan_active;
unsigned long nr_scan_inactive;
unsigned long pages_scanned; /* since last reclaim */
unsigned long flags; /* zone flags, see below */
...
/*
 * Discontig memory support fields.
 */
struct pglst_data *zone_pgdat;
/* zone_start_pfn == zone_start_paddr >> PAGE_SHIFT */
unsigned long zone_start_pfn;
...
} ___cacheline_internodealigned_in_smp;
```


存储区 (Memory Zones)

小结：struct page、Mem_map、node、zone之间的关系

- 内存布局：动态内存及引用它的一些值



Outline

- 1 页框管理：页面级分配器
 - 页框和页描述符
 - 存储区(Memory Zones)
 - 页面级分配器接口
 - 页框管理算法：伙伴算法
- 2 内存区管理(memory area)
 - slab分配器
- 3 非连续存储区管理
- 4 小结和作业

页面级分配器接口：请求和释放页框

- 内核实现了一种底层的内存分配机制：页面级分配器，并提供了几个接口供其他内核函数调用。
- 分配：
 - `alloc_pages/alloc_page/alloc_pages_node/alloc_pages_current/...`
 - `__get_free_pages/__get_free_page/__get_dma_pages/get_zeroed_page`
- 释放：
 - `free_pages/__free_pages/free_page/__free_page`

页面级分配器接口：请求和释放页框

```
include/linux/gfp.h
```

```
#define alloc_pages(gfp_mask, order) \  
    alloc_pages_node(numa_node_id(), gfp_mask, order)  
#define alloc_page_vma(gfp_mask, vma, addr) alloc_pages(gfp_mask, 0)  
...  
#define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)
```

```
static inline struct page *alloc_pages_node(int nid, gfp_t gfp_mask,  
    unsigned int order) {  
    if (unlikely(order >= MAX_ORDER)) return NULL;  
  
    /* Unknown node is current node */  
    if (nid < 0) nid = numa_node_id();  
  
    return __alloc_pages(gfp_mask, order, node_zonelist(nid, gfp_mask));  
}
```

```
mm/page_alloc.c
```

```
struct page *  
__alloc_pages(gfp_t gfp_mask, unsigned int order, struct zonelist *zonelist) {  
    return __alloc_pages_internal(gfp_mask, order, zonelist, NULL);  
}
```

页面级分配器接口：请求和释放页框

mm/page_alloc.c

```
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order) {
    struct page * page;
    page = alloc_pages(gfp_mask, order);
    if (!page)
        return 0;
    return (unsigned long) page_address(page);
}
```

include/linux/gfp.h

```
#define __get_free_page(gfp_mask) \
    __get_free_pages((gfp_mask),0)
#define __get_dma_pages(gfp_mask, order) \
    __get_free_pages((gfp_mask) | GFP_DMA,(order))
```

页面级分配器接口：请求和释放页框

mm/page_alloc.c

```
unsigned long get_zeroed_page(gfp_t gfp_mask) {
    struct page * page;
    /*
     * get_zeroed_page() returns a 32-bit address, which cannot represent
     * a highmem page
     */
    VM_BUG_ON((gfp_mask & __GFP_HIGHMEM) != 0);
    page = alloc_pages(gfp_mask | __GFP_ZERO, 0);
    if (page)
        return (unsigned long) page_address(page);
    return 0;
}
```

- 这个函数与__get_free_page基本等价，差别在于它把所分配页框中的页面数据全部清0。
这在给用户态的空间分配页框时候很有用。因为这个页框虽然是可被分配的，但是里面的数据可能包含了内核的一些敏感信息。
清0，可以避免信息泄露

关于unsigned int gfp_mask

在页面级分配器中，

gfp_mask常常用来指明可在何处并以何种方式查找空闲的页框

- **GFP_ATOMIC**，
这种分配是高优先级的并且不能睡眠。
一般在中断处理程序、下半部分和其他不能睡眠的场合下使用
- **GFP_KERNEL**，
这是普通的分配模式，允许睡眠。
一般在用户进程可能调用到的内核函数中使用，这个时候进程是可以安全的睡眠的
- **GFP_DMA**，
设备驱动程序需要DMA内存时使用
- 关于gfp_mask中更多标志位的含义，参见include/linux/gfp.h

释放页框

mm/page_alloc.c

```
void __free_pages(struct page *page, unsigned int order) {
    if (put_page_testzero(page)) {
        if (order == 0)
            free_hot_page(page);
        else
            __free_pages_ok(page, order);
    }
}
```

```
void free_pages(unsigned long addr, unsigned int order) {
    if (addr != 0) {
        VM_BUG_ON(!virt_addr_valid((void *)addr));
        __free_pages(virt_to_page((void *)addr), order);
    }
}
```

include/linux/gfp.h

```
#define __free_page(page) __free_pages((page), 0)
#define free_page(addr) free_pages((addr),0)
```


释放页框

- 在内核中释放页框时要非常小心，必须确保只释放了所请求的页框，否则内核可能会崩溃

```
page = __get_free_page(GFP_KERNEL,3);
If (!page){
    /*如果内存不足，分配失败，必须在这里处理这个失败*/
}
/*现在page变量指向了8个连续页框的起始线性地址*/

free_pages(page,3);
/*现在页框被释放，不应该再对page中存放的线性地址进行操作*/
```

Outline

- 1 页框管理：页面级分配器
 - 页框和页描述符
 - 存储区(Memory Zones)
 - 页面级分配器接口
 - 页框管理算法：伙伴算法
- 2 内存区管理(memory area)
 - slab分配器
- 3 非连续存储区管理
- 4 小结和作业

页框的管理算法的选择

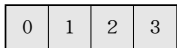
- 内核要为分配一组连续的页框建立一种**稳定、高效**的分配策略
- 这种策略**要解决（外部）碎片问题**：
 - 频繁的请求和释放不同大小的一组连续页框，必然导致在物理页框中分散许多小块的空闲页框
 - 这样，即使有足够的空闲页框满足请求，但要分配一个大块的连续页框可能就无法满足了
- **有两种办法可以避免这样的碎片**
 - ① 利用MMU把一组**非连续**的物理空闲页框映射到连续的线性地址空间
 - ② 使用一种适当的技术来**记录现存的空闲连续页框的情况**，以避免为满足对小块的请求而把大块的空闲块进行分割
- 基于下面的原因，Linux内核首选第二种方法
 - 在某些情况下，必须使用连续的页框，如DMA
 - 尽量少的修改内核页表

Linux中页面级分配器的buddy算法（伙伴算法）

- Linux使用著名的伙伴算法来解决碎片问题。
 - 把所有空闲页框分组为10（Linux2.6.26为11）个块链表，每个块链表分别包含大小为1，2，4，8，16，32，64，128，256和512个连续的页框
 - 每个块的第一个页框的物理地址是该块大小的整数倍
例如：大小为16个页框的块，其起址是16×4KB的倍数

伙伴的定义

- 例如：0和1是伙伴，1和2不是伙伴
- 两个伙伴的大小必须相同，物理地址必须连续
 - 假定伙伴的大小为b
 - 那么第一个伙伴的物理地址必须是 $2 \times b \times 4\text{KB}$ 对齐
- 事实上伙伴是通过对大块的物理内存划分获得的
 - 假如从第0个页面开始到第3个页面结束的内存



- 每次都对半划分，那么第一次划分获得大小为2页的伙伴
- 进一步划分，可以获得大小为1页的伙伴，例如0和1，2和3

相关数据结构和参数

- ① Linux为每个zone使用各自独立的伙伴系统，每个伙伴系统使用的主要数据结构为：连续空闲页框管理数组free_area

```
include/linux/mmzone.h
```

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

② MAX_ORDER取值

```
/* Free memory management - zoned buddy allocator. */  
#ifndef CONFIG_FORCE_MAX_ZONEORDER  
#define MAX_ORDER 11  
#else  
#define MAX_ORDER CONFIG_FORCE_MAX_ZONEORDER  
#endif  
#define MAX_ORDER_NR_PAGES (1 << (MAX_ORDER - 1))
```

③ mem_map数组

- 即页描述符数组，其中每个页描述符描述一个物理页框
- 整个mem_map数组描述所有zone中的物理内存

伙伴的合并

- 当一对伙伴都为空闲的时候，就合并成一个更大的块
- 该过程将一直进行，直到找不到可以合并的伙伴为止
- 寻找伙伴
 - 给定一个要释放的空闲块
 - 找到其伙伴
 - 查看其状态：合并 or 不合并

- 假设有128MB的ram。
128MB最多可以分成：
 $2^{15} = 32768$ 个页框，或者
 $2^{14} = 16384$ 个8KB（2页）的块，或者
 $2^{13} = 8192$ 个16KB（4页）的块，或者
...，或者
64个大小为512个页的块
- 假设要请求一个大小为 $2^7 = 128$ 个页框的块(0.5MB)。
 - ① 算法先到free_area[7]中检查是否有可用的空闲块，即大小为128个页框的连续空闲页框形成的空闲块
 - ② 若有，则分配并返回。
 - ③ 若没有，就到free_area[8]中找一个大一号的空间块（大小为256个页框）
 - ④ 若有，伙伴系统就把256个页框分成两等份（伙伴），一半用作满足请求，另一半插入free_area[7]中

- 5 如果在`free_area[8]`中也没有空闲块，就进一步到`free_area[9]`中找是否有空闲块。
- 6 若有，先将512个页框大小的空闲块分成一对大小为256个页框的伙伴，一个插入`free_area[8]`中，另一个进一步划分成一对大小为128个页框的伙伴，取其一插入`free_area[7]`中，另一个分配出去
- 7 如果`free_area[9]`也没有空闲块，则内存不够，返回一个错误信号

伙伴算法在Linux中的实现

- 伙伴系统相关数据结构的最初初始化

mm/page_alloc.c

```
static void __meminit zone_init_free_lists(struct zone *zone) {
    int order, t;
    for_each_migratetype_order(order, t) {
        INIT_LIST_HEAD(&zone->free_area[order].free_list[t]);
        zone->free_area[order].nr_free = 0;
    }
}
```

- 伙伴系统的释放式初始化

arch/x86/mm/init_32.c

```
void __init mem_init(void) {
    ...
    /* this will put all low memory onto the freelists */
    totalram_pages += free_all_bootmem();
    ...
}
```

伙伴算法在Linux中的实现

- 内存的分配与回收：

阅读相关代码，mm/page_alloc.c

- 关键：nr_free
- 释放：__free_one_page ← free_one_page ← __free_page_ok ← __free_pages
- 分配：__rmqueue ← buffered_rmqueue ← get_page_from_freelist ← __alloc_pages_internal ← __alloc_pages

```
/*
 * This is the 'heart' of the zoned buddy allocator.
 */
static struct page *
__alloc_pages_internal(gfp_t gfp_mask, unsigned int order,
                      struct zonelist *zonelist, nodemask_t *nodemask) {
    ...
    page = get_page_from_freelist(gfp_mask|__GFP_HARDWALL, nodemask, order,
                                  zonelist, high_zoneidx, ALLOC_WMARK_LOW|ALLOC_CPUSET);
    ...
}
```

伙伴算法在Linux中的实现

```
/*
 * get_page_from_freelist goes through the zonelist trying to allocate
 * a page.
 */
static struct page * get_page_from_freelist(gfp_t gfp_mask, nodemask_t *nodemask,
unsigned int order,
        struct zonelist *zonelist, int high_zoneidx, int alloc_flags) {
    ...
    page = buffered_rmqueue(preferred_zone, zone, order, gfp_mask);
    ...
}
```

伙伴算法在Linux中的实现

```
/*
 * Really, prep_compound_page() should be called from __rmqueue_bulk(). But
 * we cheat by calling it from here, in the order > 0 path. Saves a branch
 * or two.
 */
static struct page *buffered_rmqueue(struct zone *preferred_zone,
                                     struct zone *zone, int order, gfp_t gfp_flags) {
    ...
    if (likely(order == 0)) {
        ...
    } else {
        spin_lock_irqsave(&zone->lock, flags);
        page = __rmqueue(zone, order, migratetype);
        spin_unlock(&zone->lock);
        if (!page)
            goto failed;
    }
    ...
}
```

伙伴算法在Linux中的实现

```
/*
 * Do the hard work of removing an element from the buddy allocator.
 * Call me with the zone->lock already held.
 */
static struct page *__rmqueue(struct zone *zone, unsigned int order, int migratetype) {
    struct page *page;
    page = __rmqueue_smallest(zone, order, migratetype);
    if (unlikely(!page))
        page = __rmqueue_fallback(zone, order, migratetype);
    return page;
}
```

伙伴算法在Linux中的实现

- `__rmqueue_smallest()`

在`free_area`中找到满足要求的连续空闲页框

```
/*
 * Go through the free lists for the given migratetype and remove
 * the smallest available page from the freelists
 */
static struct page *__rmqueue_smallest(struct zone *zone, unsigned int order,
                                       int migratetype) {
    unsigned int current_order;
    struct free_area * area;
    struct page *page;

    /* Find a page of the appropriate size in the preferred list */
    for (current_order = order; current_order < MAX_ORDER; ++current_order) {
        area = &(zone->free_area[current_order]);
        if (list_empty(&area->free_list[migratetype])) continue;

        page = list_entry(area->free_list[migratetype].next, struct page, lru);
        list_del(&page->lru);
        rmv_page_order(page);
        area->nr_free--;
        __mod_zone_page_state(zone, NR_FREE_PAGES, - (1UL << order));
        expand(zone, page, order, current_order, area, migratetype);
        return page;
    }
    return NULL;
}
```


伙伴算法在Linux中的实现

- `expand()`将找到的连续空闲页框逐级划分成伙伴
 - 前一半被逐级划分，后一半保留在空闲链中

```
static inline void expand(struct zone *zone, struct page *page,
                        int low, int high, struct free_area *area, int migratetype) {
    unsigned long size = 1 << high;

    while (high > low) {
        area--;
        high--;
        size >>= 1;
        VM_BUG_ON(bad_range(zone, &page[size]));
        list_add(&page[size].lru, &area->free_list[migratetype]);
        area->nr_free++;
        set_page_order(&page[size], high);
    }
}
```

伙伴算法在Linux中的实现

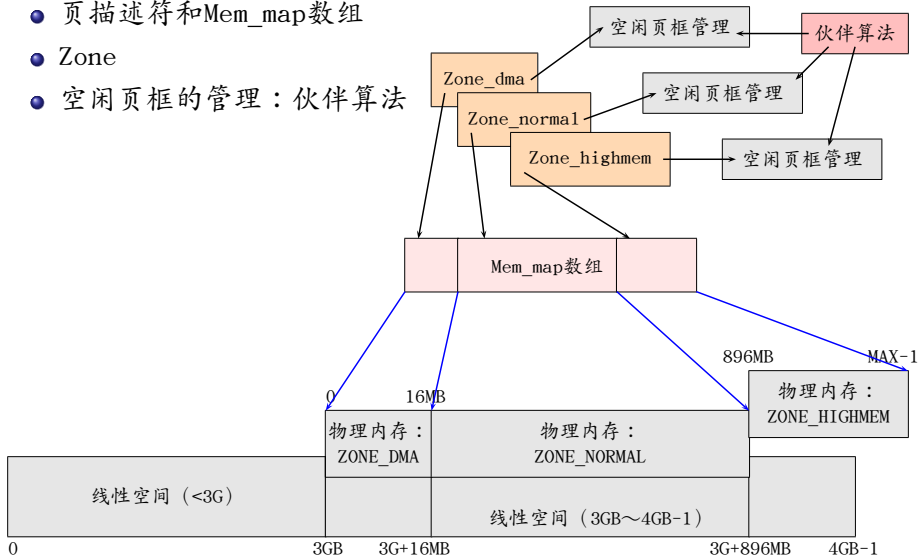
- `__free_one_page()` 释放一个连续页框块到 `free_area` 中
 - 考虑合并

```
static inline void __free_one_page(struct page *page,
                                  struct zone *zone, unsigned int order) {
    ...
    page_idx = page_to_pfn(page) & ((1 << MAX_ORDER) - 1);
    ...
    while (order < MAX_ORDER-1) {
        ...
        buddy = __page_find_buddy(page, page_idx, order);
        if (!page_is_buddy(page, buddy, order)) break;

        /* Move the buddy up one level. */
        list_del(&buddy->lru);
        zone->free_area[order].nr_free--;
        rmv_page_order(buddy);
        combined_idx = __find_combined_index(page_idx, order);
        page = page + (combined_idx - page_idx);
        page_idx = combined_idx;
        order++;
    }
    set_page_order(page, order);
    list_add(&page->lru,
            &zone->free_area[order].free_list[migratetype]);
    zone->free_area[order].nr_free++;
}
```

页框管理小结

- 页描述符和Mem_map数组
- Zone
- 空闲页框的管理：伙伴算法



- 察看 `/proc/buddyinfo`:
 - `cat /proc/buddyinfo`
- 参见 `Documentation/filesystems/proc.txt`
 - Kernel memory allocator information
 - Memory fragmentation is a problem under some workloads, and `buddyinfo` is a useful tool for helping diagnose these problems. `Buddyinfo` will give you a clue as to how big an area you can safely allocate, or why a previous allocation failed.
 - Each column represents the number of pages of a certain order which are available.

- 1 页框管理：页面级分配器
- 2 内存区管理(memory area)
 - slab分配器
- 3 非连续存储区管理
- 4 小结和作业

内存区管理(memory area)

- 单单分配页面的分配器肯定是不能满足要求的
- 内核中大量使用各种数据结构，大小从几个字节到几十上百KB不等，都取整到2的幂次个页面那是完全不现实的
- 早期内核的解决方法是提供大小为2,4,8,16,...,131056字节的内存区域
 - 需要新的内存区域时，内核从伙伴系统申请页面，把它们划分成一个个区域，取其中一个来满足需求
 - 如果某个页面中的内存区域都释放了，页面就交回到伙伴系统

内存区管理(memory area)

- 但这种分配方法有许多值得改进的地方：
 - ① **不同的数据类型**用不同的方法分配内存可能提高效率。
比如需要初始化的数据结构，释放后可以暂存着，再分配时就不必初始化了
 - ② 内核的函数常常**重复地使用**同一类型的内存区，缓存最近释放的对象可以加速分配和释放
 - ③ 对内存的请求可以按照**请求频率**来分类，频繁使用的类型使用专门的缓存，很少使用的可以使用通用缓存
 - ④ 使用2的幂次大小的内存区域时**硬件高速缓存冲突**的概率较大，有可能通过仔细安排内存区域的起始地址来减少硬件高速缓存冲突
 - ⑤ **缓存**一定数量的对象可以减少对buddy系统的调用，从而节省时间并减少由此引起的硬件高速缓存污染

Linux 2.6.26 中的内存区管理算法

- SLOB Allocator: Simple List Of Blocks
 - NUMA
- Slab
- Slub: slab 的一个变种
- 接口: Kmalloc/kfree 等
- 本课介绍基本的 **slab 算法**

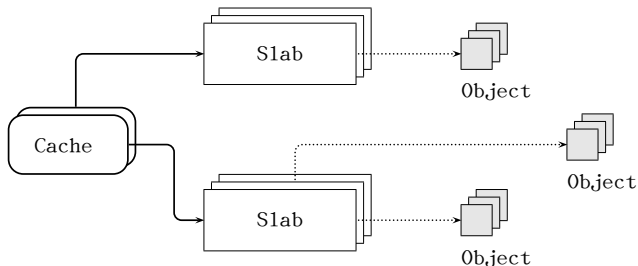
Outline

- 1 页框管理：页面级分配器
 - 页框和页描述符
 - 存储区(Memory Zones)
 - 页面级分配器接口
 - 页框管理算法：伙伴算法
- 2 内存区管理(memory area)
 - slab分配器
- 3 非连续存储区管理
- 4 小结和作业

- slab分配器最先由Jeff Bonwick在Solaris 2.4操作系统中引入，现在广泛用于Unix和Linux系统中
 - 在Linux-2.6.23之前为缺省的内存区分配器
- slab分配器体现了这些改进思想
 - slab分配器把内存区看成对象
 - slab分配器把对象分组放进高速缓存。
 - 每个高速缓存都是同种类型内存对象的一种“储备”
 - 例如当一个文件被打开时，存放相应“打开文件”对象所需的内存是从一个叫做filp(file pointer)的slab分配器的高速缓存中得到的
 - 也就是说每种对象类型对应一个高速缓存

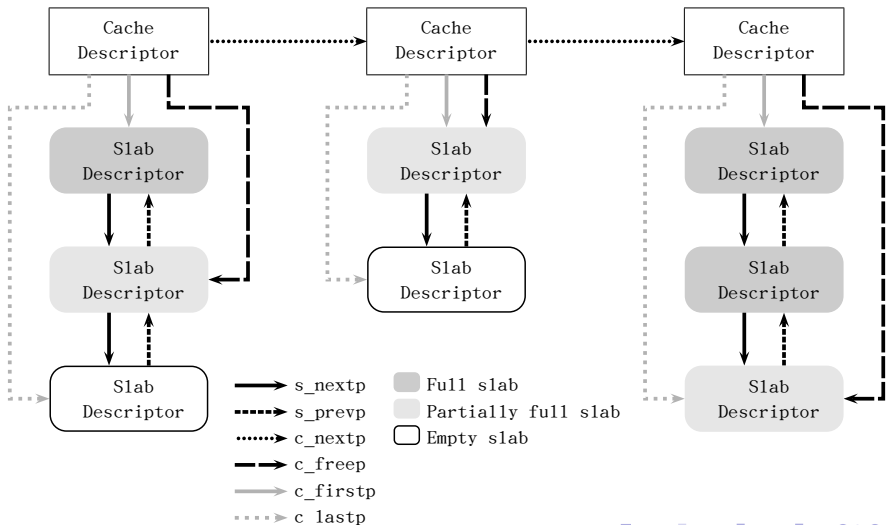
- 每个高速缓存被分成多个slabs，每个slab由一个或多个连续的页框组成，其中包含一定数目的对象

Figure 7-3. The slab allocator components



- 每个slab有三种状态：全满，半满，全空
 - 全满意味着slab中的对象全部已被分配出去
 - 全空意味着slab中的对象全部是可用的
 - 半满介于两者之间
- 当内核函数需要一个新的对象时，
 - 优先从半满的slab满足这个请求
 - 否则从全空的slab中取一个对象满足请求
 - 如果没有空的slab则向buddy系统申请页面生成一个新的slab

● 高速缓存描述符和slab描述符之间的关系



普通和专用高速缓存

- 每个高速缓存使用kmem_cache表示，参见mm/slab.c
- 普通高速缓存根据大小分配内存
 - 26个，2组（一组用于DMA分配，另一组用于常规分配）
 - 每组13个，大小从 $2^5 = 32$ 个字节，到 $2^{17} = 132017$ 个字节
 - 数据结构cache_sizes
 - 数组：malloc_sizes
- 专用高速缓存根据类型分配

普通和专用高速缓存

include/linux/slab_def.h

```
/* Size description struct for general caches. */
struct cache_sizes {
    size_t cs_size;
    struct kmem_cache *cs_cachep;
#ifdef CONFIG_ZONE_DMA
    struct kmem_cache *cs_dmacachep;
#endif
};
extern struct cache_sizes malloc_sizes[];
```

mm/slab.c

```
/*
 * These are the default caches for kmalloc. Custom caches can have other sizes.
 */
struct cache_sizes malloc_sizes[] = {
#define CACHE(x) { .cs_size = (x) },
#include <linux/kmalloc_sizes.h>
    CACHE(ULONG_MAX)
#undef CACHE
};
```

普通和专用高速缓存

```
/*
 * struct slab
 *
 * Manages the objs in a slab. Placed either at the beginning of mem allocated
 * for a slab, or allocated from an general cache.
 * Slabs are chained into three list: fully used, partial, fully free slabs.
 */
struct slab {
    struct list_head list;
    unsigned long colouroff;
    void *s_mem; /* including colour offset */
    unsigned int inuse; /* num of objs active in slab */
    kmem_bufctl_t free;
    unsigned short nodeid;
};
```


普通和专用高速缓存

```
/*
 * The slab lists for all objects.
 */
struct kmem_list3 {
    struct list_head slabs_partial; /* partial list first, better asm code */
    struct list_head slabs_full;
    struct list_head slabs_free;
    unsigned long free_objects;
    unsigned int free_limit;
    unsigned int colour_next; /* Per-node cache coloring */
    spinlock_t list_lock;
    struct array_cache *shared; /* shared per node */
    struct array_cache **alien; /* on other nodes */
    unsigned long next_reap; /* updated without locking */
    int free_touched; /* updated without locking */
};
```

slab分配器提供的接口

- 创建专用高速缓存：`kmem_cache_create`
- 撤销专用高速缓存：`kmem_cache_destroy`
 - 一般内核撤销一个模块时会调用这个函数撤销属于那个模块的cache类型
- 从专用高速缓冲中分配和释放
 - 从高速缓存中分配/释放一个内存对象
 - `kmem_cache_alloc/kmem_cache_free`
- 从普通高速缓存中分配和释放
 - `kmalloc/kfree`
- 举例说明使用情况
 - 如果编写的内核模块有许多创建和释放数据结构的操作，可以考虑调用前面所述的slab分配器的接口创建一个高速缓存这样可以大大减少内存的访问时间

slab分配器和伙伴系统的接口

- slab分配器调用`kmem_getpages()`来获取一组连续的空闲页框
- 相应的有`kmem_freepages()`来释放分配给slab分配器的页框

- Documentation/filesystems/proc.txt
 - The slabinfo file gives information about memory usage at the slab level. Linux uses slab pools for memory management above page level in version 2.2. Commonly used objects have their own slab pool (such as network buffers, directory cache, and so on).

Outline

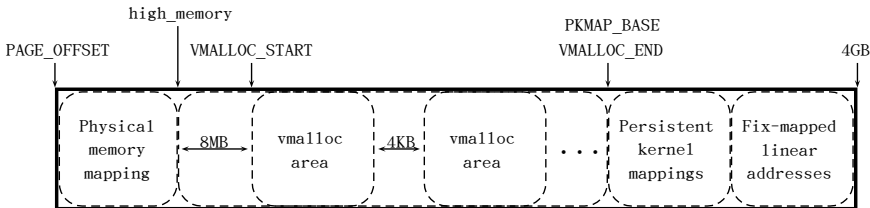
- 1 页框管理：页面级分配器
- 2 内存区管理(memory area)
- 3 非连续存储区管理**
- 4 小结和作业

3、非连续存储区管理

- 把线性空间映射到一组连续的页框是很好的选择
- 有时候不得不将线性空间映射到一组不连续的页框
 - 优点：避免碎片

为非连续内存区保留的线性地址空间

- VMALLOC_START~VMALLOC_END



非连续存储区的描述符

```
include/linux/vmalloc.h
```

```
struct vm_struct {  
    /* keep next,addr,size together to speedup lookups */  
    struct vm_struct *next;  
    void *addr;  
    unsigned long size;  
    unsigned long flags;  
    struct page **pages;  
    unsigned int nr_pages;  
    unsigned long phys_addr;  
    void *caller;  
};
```

```
DEFINE_RWLOCK(vmlist_lock);  
struct vm_struct *vmlist;
```

- mm/vmalloc.c
 - Vmalloc等分配一个非连续存储区
 - Vfree释放非连续线性区间

❶ vmalloc()分配一个非连续线性区

```
/**
 * vmalloc - allocate virtually contiguous memory
 * @size: allocation size
 * Allocate enough pages to cover @size from the page level
 * allocator and map them into contiguous kernel virtual space.
 *
 * For tight control over page level allocator and protection flags
 * use __vmalloc() instead.
 */
void *vmalloc(unsigned long size) {
    return __vmalloc_node(size, GFP_KERNEL | __GFP_HIGHMEM, PAGE_KERNEL,
        -1, __builtin_return_address(0));
}
```

② __vmalloc_node

- ① 调用 __get_vm_area_node() 分配 vm_struct 结构并分配线性地址空间
- ② 调用 __vmalloc_area_node() 分配物理页并修改相关页表映射关系

```
static void *__vmalloc_node(unsigned long size, gfp_t gfp_mask, pgprot_t prot,
                           int node, void *caller) {
    struct vm_struct *area;

    size = PAGE_ALIGN(size);
    if (!size || (size >> PAGE_SHIFT) > num_physpages)
        return NULL;

    area = __get_vm_area_node(size, VM_ALLOC, VMALLOC_START, VMALLOC_END,
                             node, gfp_mask, caller);

    if (!area)
        return NULL;

    return __vmalloc_area_node(area, gfp_mask, prot, node, caller);
}
```

⑧ __get_vm_area_node()分配vm_struct结构并分配线性地址空间

```

static struct vm_struct * __get_vm_area_node(unsigned long size, unsigned long flags,
      unsigned long start, unsigned long end, int node, gfp_t gfp_mask, void *caller) {
    struct vm_struct **p, *tmp, *area;
    unsigned long align = 1;
    unsigned long addr;

    BUG_ON(in_interrupt());
    ...
    addr = ALIGN(start, align);
    size = PAGE_ALIGN(size);
    ...
    area = kmalloc_node(sizeof(*area), gfp_mask & GFP_RECLAIM_MASK, node);
    ...
    /* * We always allocate a guard page. */
    size += PAGE_SIZE;

    write_lock(&vmlist_lock);
    for (p = &vmlist; (tmp = *p) != NULL ;p = &tmp->next) {
        if ((unsigned long)tmp->addr < addr) {
            if((unsigned long)tmp->addr + tmp->size >= addr)
                addr = ALIGN(tmp->size + (unsigned long)tmp->addr, align);
            continue;
        }
    }
}

```

⑥ __get_vm_area_node()分配vm_struct结构并分配线性地址空间

```

    if ((size + addr) < addr) goto out;
    if (size + addr <= (unsigned long)tmp->addr) goto found;
    addr = ALIGN(tmp->size + (unsigned long)tmp->addr, align);
    if (addr > end - size) goto out;
}
if ((size + addr) < addr) goto out;
if (addr > end - size) goto out;
found:
area->next = *p;
*p = area;
area->flags = flags;
area->addr = (void *)addr;
area->size = size;
area->pages = NULL;
area->nr_pages = 0;
area->phys_addr = 0;
area->caller = caller;
write_unlock(&vmlist_lock);
return area;
out:
...
}

```

④ __vmalloc_area_node()

- ① 以一个页框为单位分配所需物理页
- ② 调用map_vm_area()完成页表映射

```
static void *__vmalloc_area_node(struct vm_struct *area, gfp_t gfp_mask,
                                pgprot_t prot, int node, void *caller) {
    struct page **pages;
    unsigned int nr_pages, array_size, i;

    nr_pages = (area->size - PAGE_SIZE) >> PAGE_SHIFT;
    array_size = (nr_pages * sizeof(struct page *));

    area->nr_pages = nr_pages;
    /* Please note that the recursion is strictly bounded. */
    if (array_size > PAGE_SIZE) {
        pages = __vmalloc_node(array_size, gfp_mask | __GFP_ZERO, P
                                AGE_KERNEL, node, caller);
        area->flags |= VM_VPAGES;
    } else {
        pages = kmalloc_node(array_size,
                              (gfp_mask & GFP_RECLAIM_MASK) | __GFP_ZERO, node);
    }
}
```

● __vmalloc_area_node()

```
area->pages = pages;
area->caller = caller;
...
for (i = 0; i < area->nr_pages; i++) {
    struct page *page;
    if (node < 0)    page = alloc_page(gfp_mask);
    else    page = alloc_pages_node(node, gfp_mask, 0);
    ...
    area->pages[i] = page;
}
if (map_vm_area(area, prot, &pages))    goto fail;
return area->addr;

fail: ...
}
```

⑤ map_vm_area()完成页表映射

```

int map_vm_area(struct vm_struct *area, pgprot_t prot, struct page ***pages) {
    pgd_t *pgd; unsigned long next;
    unsigned long addr = (unsigned long) area->addr;
    unsigned long end = addr + area->size - PAGE_SIZE;
    int err;

    BUG_ON(addr >= end);
    pgd = pgd_offset_k(addr);
    do {
        next = pgd_addr_end(addr, end);
        err = vmap_pud_range(pgd, addr, next, prot, pages);
        if (err) break;
    } while (pgd++, addr = next, addr != end);
    flush_cache_vmap((unsigned long) area->addr, end);
    return err;
}

```

⑤ map_vm_area()完成页表映射

- vmap_pud_range→vmap_pmd_range→vmap_pte_range

```
static int vmap_pte_range(pmd_t *pmd, unsigned long addr,
                        unsigned long end, pgprot_t prot, struct page ***pages) {
    pte_t *pte;

    pte = pte_alloc_kernel(pmd, addr);
    if (!pte) return -ENOMEM;
    do {
        struct page *page = **pages;
        WARN_ON(!pte_none(*pte));
        if (!page) return -ENOMEM;
        set_pte_at(&init_mm, addr, pte, mk_pte(page, prot));
        (*pages)++;
    } while (pte++, addr += PAGE_SIZE, addr != end);
    return 0;
}
```


- vfree()释放非连续线性区
 - ① 释放相关数据结构
 - ② 释放物理页框

非连续线性区中的page_fault

- 非连续线性区需要进行页表映射
 - 惰性：只映射init_mm，其他按需。因此可能会发生缺页

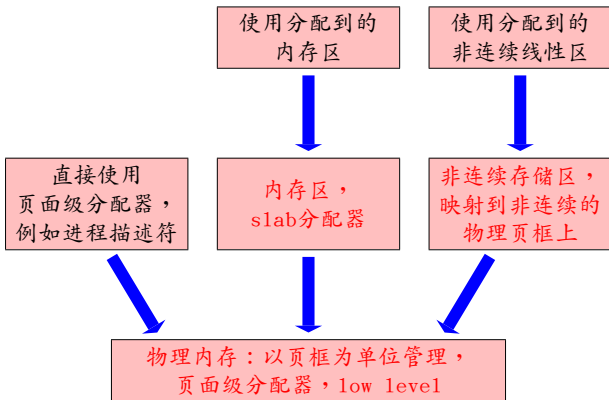
```
static int vmalloc_fault(unsigned long address) {
    unsigned long pgd_paddr;
    pmd_t *pmd_k;
    pte_t *pte_k;
    /* Make sure we are in vmalloc area */
    if (!(address >= VMALLOC_START && address < VMALLOC_END))    return -1;
    /*
     * Synchronize this task's top level page-table
     * with the 'reference' page table.
     *
     * Do _not_ use "current" here. We might be inside
     * an interrupt in the middle of a task switch..
     */
    pgd_paddr = read_cr3();
    pmd_k = vmalloc_sync_one(__va(pgd_paddr), address);
    if (!pmd_k)    return -1;
    pte_k = pte_offset_kernel(pmd_k, address);
    if (!pte_present(*pte_k))    return -1;
    return 0;
}
```

Outline

- 1 页框管理：页面级分配器
- 2 内存区管理(memory area)
- 3 非连续存储区管理
- 4 小结和作业

- 小结：

页框管理、内存区管理、非连续存储区管理之间的关系



- 察看 `/proc/meminfo`
 - Provides information about distribution and utilization of memory
 - `cat /proc/meminfo`
 - 关于每一项的解释，参见 `Documentation/filesystems/proc.txt`

- 已知物理内存大小>1GB，
对于线性地址0xC0123456，你能知道它对应的物理地址是多少么？
若知道，给出物理地址并说明该物理地址对应的物理页框号。
- Linux的页框管理采用什么算法？简述该算法。
- Linux中的slab算法的用途是什么？简述该算法。
- 什么是非连续内存区？
- Linux内核为进程分配内核栈（假定是8KB大小）时为什么不使用非连续线性区或者slab分配器接口而使用伙伴算法接口？

Thanks !

The end.