《深入理解LINUX内存管理》学习笔记

引子

为什么要写这个笔记：

1，这本书的中文版翻译了太垃圾，没法阅读。阅读英文原版，可以很好的理解作者的思路。作此笔记备忘

2，一直以来学习LINUX kernel的知识缺乏系统化，借对这本书的学习，系统化的学习一下LINUX kernel。

3，自己一直在做一个too small，too simple的单进程，特权模式，64bit保护模式的称不上OS的ＯＳ，已经做完了bootloader, 构思kernel的实现的时候，困惑在内存管理的实现上，阅读这本书，希望能有利于自己的OS的编写。

4，克服惰性，多读书，希望一天能阅读5页，争取半年内阅读完这本原版700多页的巨著。

不足：

我不可能完全理解LINUX 内存管理的精髓，肯定有很多地方理解错误。希望大家能够指正，以便提高，谢谢。

学习方法：

可能您第一次阅读的时候很多地方都不理解，不用担心。那您可能需要阅读一些文件系统的知识。

或者阅读全部笔记后，再回头阅读，有些地方您就理解了。

言归正传：

目录：

一、概要

可用工具

CodeViz: 生成代码调用关系图的工具，这个工具我现在还没有去使用，有兴趣的可以自己试试去建立调用关系图。

http://www.csn.ul.ie/~mel/projects/codeviz/

Linux cross reference (LXR): 以web的方式阅读和查找LINUX内核源代码的工具。这个工具安装相当麻烦，我建议直接到它的官方网站直接读代码。

http://lxr.linux.no/linux+v2.6.24/

模块

LINUX内存管理代码模块主要分为4个部分：

1.Out of memory 代码在mm/oom_kill.c 貌似用于杀进程的时候对内存的操作

2.虚拟内存的分配 代码在mm/vmalloc.c

3.物理内存页面分配 代码在mm/page_alloc.cVMA（virtual memory addresses)的 创建和进程内的内存区域的管理

4.这些模块，贯穿与其他kernel代码之中，形成更复杂的系统模块，如页面替换策略，buffer的输入输出等

二、物理内存

从硬件角度看内存系统，有2种主流的体系结构，不一致的内存访问系统（NUMA），我不知道什么系统在用这样模式，这种系统将内存系统分割成2块区域（BANK），一块是专门给CPU去访问，一块是给外围设备板卡的DMA去访问。另外一种体系结构，是一致的内存访问系统（UMA），PC都是用的这种结构，这种结构的对于CPU和其他外围设备访问的内存在一块内存条上，没有任何不同。

LINUX内核需要支持这2种体系结构。它引入了一个概念称为node，一个node对应一个bank，对于UMA体系的，系统中只有一个node。在LINUX中引入一个数据结构“struct pglist_data” ,来描述一个node，定义在include/linux/mmzone.h 文件中。（这个结构被typedef pg_data_t）

对于NUMA系统来讲， 整个系统的内存由一个node_data 的pg_data_t指针数组来管理。（因为可能有多个node）对于PC这样的UMA系统，使用struct pglist_datacontig_page_data ，作为系统唯一的node管理所有的内存区域。（UMA系统中中只有一个node）

每个node又被分成多个zone，它们各自描述在内存中的范围。zone由struct zone_struct 数据结构来描述。zone的类型由zone_t表示，有ZONE_DMA, ZONE_NORMAL, ZONE_HIGHMEM这三种类型。它们之间的用途是不一样的，ZONE_DMA类型的内存区域在物理内存的低端，主要是ISA设备只能用低端的地址做DMA操作。ZONE_NORMAL类型的内存区域直接被内核映射到线性地址空间上面的区域（line address space），以后的章节将详细描述。ZONE_HIGHMEM将保留给系统使用。

在PC系统中，内存区域类型如下分布：

ZONE_DMA 0-16MB

ZONE_NORMAL 16MB-896MB

ZONE_HIGHMEM 896MB-物理内存结束

大多数kernel的操作只使用ZONE_NORMAL区域，系统内存由很多固定大小的内存块组成的，这样的内存块称作为“页”（PAGE），x86体系结构中，page的大小为4096个字节。每个物理的页由一个struct page的数据结构对象来描述。页的数据结构对象都保存在mem_map全局数组中。从载入内核的低地址内存区域的后面内存区域，也就是ZONE_NORMAL开始的地方的内存的页的数据结构对象，都保存在这个全局数组中。

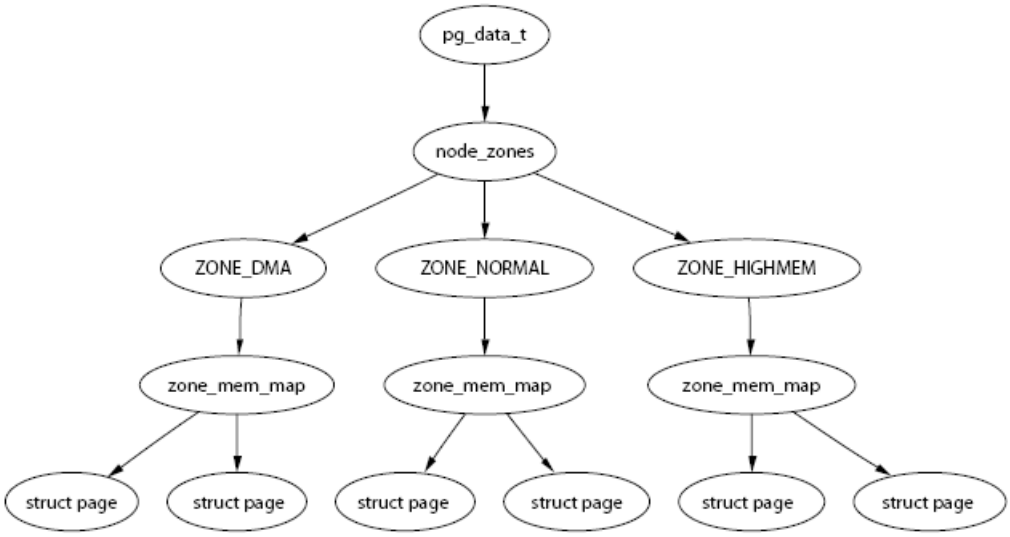


Figure 2.1. Relationship Between Nodes, Zones and Pages

因为ZONE_NORMAL区域的内存空间也是有限的，所以LINUX也支持High memory的访问，这个下面章节会描述，这个章节，将主要描述node，zone，page及它们之间的关联

Nodes

表示node的数据结构为pg_data_t, 也就是struct pglist_data, 这个结构定义在<linux/mmzone.h>中：

```
typedef struct pglist_data
 {
    struct zone
 node_zones
[MAX_NR_ZONES
];
    struct zonelist
 node_zonelists
[MAX_ZONELISTS
];
    int nr_zones
;
    struct page
 *node_mem_map
;
    struct bootmem_data
 *bdata
;
    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
;
    wait_queue_head_t
 kswapd_wait
;
    struct task_struct
 *kswapd
;
    int kswapd_max_order
;
} pg_data_t
;
```

node_zones: 分别为ZONE_DMA,ZONE_NORMAL,ZONE_HIGHMEM

node_zonelists: 分配内存操作时的区域顺序，当调用free_area_init_core()时，由mm/page_alloc.c文件中的build_zonelists()函数设置。

nr_zones: node中的zone的数量，1到3个之间。并不是所有的node都有3个zone的，比如有些就没有ZONE_DMA区域。

node_mem_map: node中的第一个page，它可以指向mem_map中的任何一个page。

bdata: 这个仅用于boot 的内存分配，下面再描述

node_start_pfn: pfn是page frame number的缩写。这个成员是用于表示node中的开始那个page在物理内存中的位置的。

2.4以前的版本，用物理地址来表示的，后来由于硬件的发展，物理内存很可能大于32bit所表示4G的内存地址，所以改为以页为单位表示。

node_present_pages: node中的真正可以使用的page数量

node_spanned_pages: node中所有存在的Page的数量，包括可用的，也包括被后面讲到的mem_map所占用的，dma所占用的区域的。（做了修正）

英文原版是这么描述的："node spanned pages" is the total area that is addressed by the node, including any holes that may exist.可能是包括hold的node可以访问的区域的数量吧。

node_id: node的NODE ID，从0开始

kswapd_wait: node的等待队列

对于单一node的系统，contig_page_data 是系统唯一的node数据结构对象。

Zone

每个zone都由一个struct zone数据结构对象描述。zone对象里面保存着内存使用状态信息，如page使用统计，未使用的内存区域，互斥访问的锁（LOCKS）等。struct zone在<linux/mmzone.h>中定义（把不关心的NUMA和memory hotplug相关的成员给省略掉了）：

```
struct zone {
        unsigned long           free_pages;
        unsigned long           pages_min, pages_low, pages_high;
        unsigned long           lowmem_reserve[MAX_NR_ZONES];

       struct per_cpu_pageset  pageset[NR_CPUS];

       spinlock_t              lock;

       struct free_area        free_area[MAX_ORDER];

       ZONE_PADDING(_pad1_)  //用于字节对齐

        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           nr_active;
        unsigned long           nr_inactive;
        unsigned long           pages_scanned;
        int                     all_unreclaimable;

        atomic_t                reclaim_in_progress;

        atomic_long_t           vm_stat[NR_VM_ZONE_STAT_ITEMS];

        int prev_priority;


        ZONE_PADDING(_pad2_)  //用于字节对齐
        wait_queue_head_t       * wait_table;
        unsigned long           wait_table_hash_nr_entries;
        unsigned long           wait_table_bits;

        struct pglist_data      *zone_pgdat;
        unsigned long           zone_start_pfn;

        unsigned long           spanned_pages;
        unsigned long           present_pages;

        const char              *name;
 } ____cacheline_internodealigned_in_smp;
```

free_pages：未分配使用的page的数量。

pages_min, pages_low and pages_high： zone对page管理调度的一些参数，下面章节将讲到。

lowmem_reserve[MAX_NR_ZONES]： 为了防止一些代码必须运行在低地址区域，所以事先保留一些低地址区域的内存。

pageset[NR_CPUS]： page管理的数据结构对象，内部有一个page的列表(list)来管理。每个CPU维护一个page list，避免自旋锁的冲突。这个数组的大小和NR_CPUS(CPU的数量）有关，这个值是编译的时候确定的。

lock: 对zone并发访问的保护的自旋锁

free_area： 页面使用状态的信息，以每个bit标识对应的page是否可以分配

lru_lock: LRU（最近最少使用算法)的自旋锁

reclaim_in_progress: 回收操作的原子锁

active_list: 活跃的page的list

inactive_list: 不活跃的page的list

refill_counter:从活跃的page list中移除的page的数量

nr_active: 活跃的page的数量

nr_inactive: 不活跃的page的数量

pressure： 检查回收page的指标

all_unreclaimable: 如果检测2次还是不能回收zone的page的话，则设置为1

pages_scanned: 上次回收page后，扫描过的page的数量。

wait_table：等待一个page释放的等待队列哈希表。它会被wait_on_page(),unlock_page()函数使用. 用哈希表，而不用一个等待队列的原因，防止进程长期等待资源。

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

zone_pgdat: 指向这个zone所在的pglist_data对象。

zone_start_pfn： 和node_start_pfn的含义一样。这个成员是用于表示zone中的开始那个page在物理内存中的位置的
present_pages, spanned_pages: 和node中的类似的成员含义一样。

zone: zone的名字，字符串表示： "DMA","Normal" 和"HighMem"

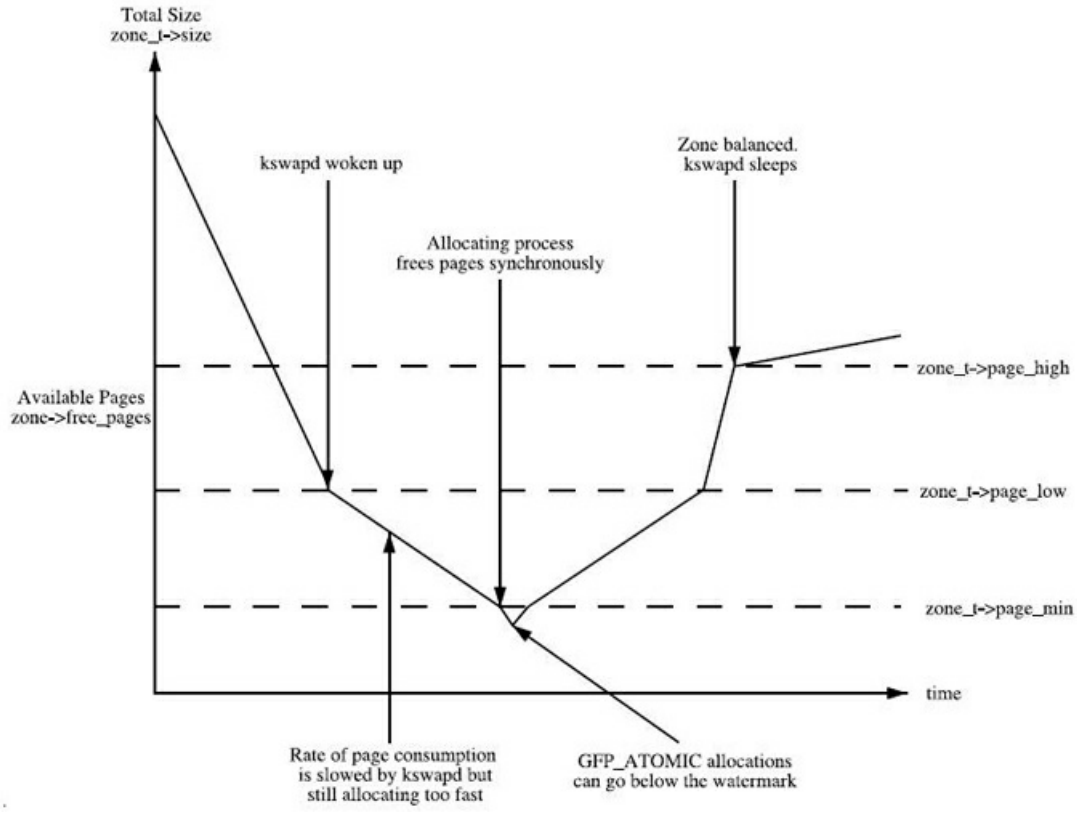
ZONE_PADDING： 由于自旋锁频繁的被使用，因此为了性能上的考虑，将某些成员对齐到cache line中，有助于提高执行的性能。使用这个宏，可以确定zone->lock,zone->lru_lock,zone->pageset这些成员使用不同的cache line.

Zone的管理调度的一些参数：（Zone watermarks),

英文直译为zone的水平，打个比喻，就像一个水库，水存量很小的时候加大进水量，水存量达到一个标准的时候，减小进水量，当快要满的时候，可能就关闭了进水口。pages_min, pages_low and pages_high就类似与这个标准。

当系统中可用内存很少的时候，系统代码kswapd被唤醒，开始回收释放page。pages_min, pages_low and pages_high这些参数影响着这个代码的行为。

每个zone有三个水平标准：pages_min, pages_low and pages_high，帮助确定zone中内存分配使用的压力状态。kswapd和这3个参数的互动关系如下图：



page_min中所表示的page的数量值，是在内存初始化的过程中调用free_area_init_core()中计算的。这个数值是根据zone中的page的数量除以一个>1的系数来确定的。通常是这样初始化的ZoneSizeInPages/128。

page_low: 当空闲页面的数量达到page_low所标定的数量的时候，kswapd线程将被唤醒，并开始释放回收页面。这个值默认是page_min的2倍。

page_min: 当空闲页面的数量达到page_min所标定的数量的时候, 分配页面的动作和kswapd线程同步运行

page_high: 当空闲页面的数量达到page_high所标定的数量的时候, kswapd线程将重新休眠，通常这个数值是page_min的3倍。

zone的大小的计算

setup_memory()函数计算每个zone的大小：

$$\text{wait\_table\_size} = \log_2(\frac{\text{NoPages} * 2}{\text{PAGE\_PER\_WAITQUEUE}} - 1)$$

PFN是物理内存以Page为单位的偏移量。系统可用的第一个PFN是min_low_pfn变量，开始与_end标号的后面，也就是kernel结束的地方。在文件mm/bootmem.c中对这个变量作初始化。系统可用的最后一个PFN是max_pfn变量，这个变量的初始化完全依赖与硬件的体系结构。x86的系统中，find_max_pfn()函数通过读取e820表获得最高的page frame的数值。同样在文件mm/bootmem.c中对这个变量作初始化。e820表是由BIOS创建的。

x86中，max_low_pfn变量是由find_max_low_pfn()函数计算并且初始化的，它被初始化成ZONE_NORMAL的最后一个page的位置。这个位置是kernel直接访问的物理内存，也是关系到kernel/userspace通过“PAGE_OFFSET宏”把线性地址内存空间分开的内存地址位置。（原文：This is the physical memory directly accessible by the kernel and is related to the kernel/userspace split in the linear address space marked by PAGE OFFSET.）我理解为这段地址kernel可以直接访问，可以通过PAGE_OFFSET宏直接将kernel所用的虚拟地址转换成物理地址的区段。在文件mm/bootmem.c中对这个变量作初始化。在内存比较小的系统中max_pfn和max_low_pfn的值相同

min_low_pfn, max_pfn和max_low_pfn这3个值，也要用于对高端内存（high memory)的起止位置的计算。在arch/i386/mm/init.c文件中会对类似的highstart_pfn和highend_pfn变量作初始化。这些变量用于对高端内存页面的分配。后面将描述。

Zone等待队列表（zone wait queue table)

当对一个page做I/O操作的时候，I/O操作需要被锁住，防止不正确的数据被访问。进程在访问page前，调用wait_on_page()函数，使进程加入一个等待队列。访问完成后，UnlockPage()函数解锁其他进程对page的访问。其他正在等待队列中的进程被唤醒。每个page都可以有一个等待队列，但是太多的分离的等待队列使得花费太多的内存访问周期。替代的解决方法，就是将所有的队列放在struct zone数据结构中。

也可以有一种可能，就是struct zone中只有一个队列，但是这就意味着，当一个page unlock的时候，访问这个zone里内存page的所有休眠的进程将都被唤醒，这样就会出现拥堵（thundering herd）的问题。建立一个哈希表管理多个等待队列，能解决这个问题，zone->wait_table就是这个哈希表。哈希表的方法可能还是会造成一些进程不必要的唤醒。但是这种事情发生的机率不是很频繁的。下面这个图就是进程及等待队列的运行关系：

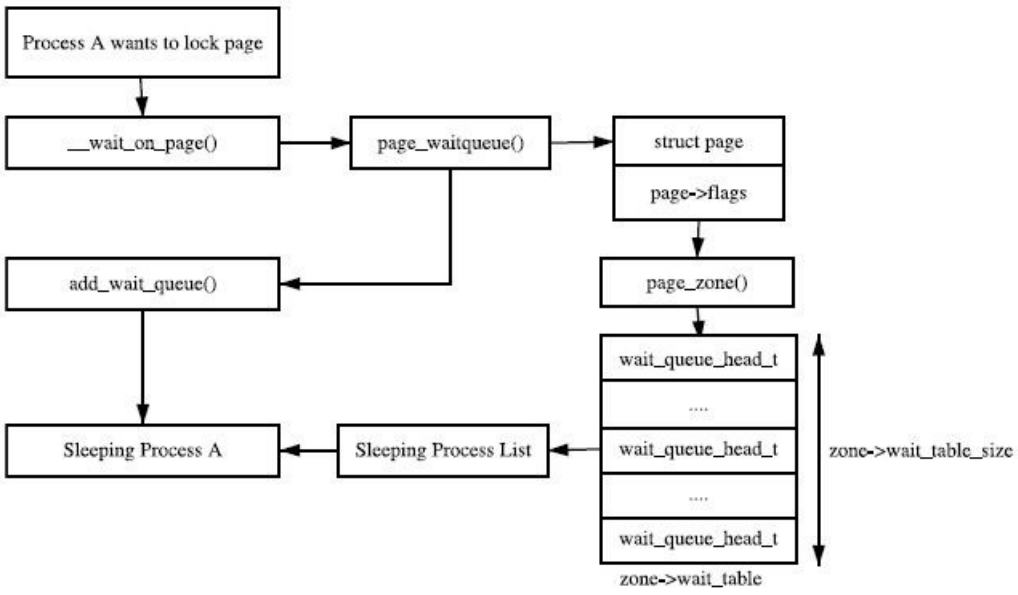


Figure 2.4. Sleeping on a Locked Page

等待队列的哈希表的分配和建立在free_area_init_core()函数中进行。哈希表的表项的数量在wait_table_size() 函数中计算，并且保持在zone->wait_table_size成员中。最大4096个等待队列。最小是NoPages / PAGES_PER_WAITQUEUE的2次方，NoPages是zone管理的page的数量，PAGES_PER_WAITQUEUE被定义256。（原文：For smaller tables, the size of the table is the minimum power of 2 required to store NoPages / PAGES PER WAITQUEUE number of queues, where NoPages is the number of pages in the zone and PAGE PER WAITQUEUE is defined to be 256.）

下面这个公式可以用于计算这个值：

$$\text{wait\_table\_size} = \log_2(\frac{\text{NoPages} * 2}{\text{PAGE\_PER\_WAITQUEUE}} - 1)$$

zone->wait_table_bits用于计算：根据page 地址得到需要使用的等待队列在哈希表中的索引的算法因子。page_waitqueue()函数负责返回zone中page所对应等待队列。它用一个基于struct page虚拟地址的简单的乘法哈希算法来确定等待队列的。

page_waitqueue()函数用GOLDEN_RATIO_PRIME的地址和“右移zone→wait_table_bits一个索引值”的一个乘积来确定等待队列在哈希表中的索引的。

Zone的初始化

在kernel page table通过paging_init()函数完全建立起z来以后，zone被初始化。下面章节将描述这个。当然不同的体系结构这个过程肯定也是不一样的，但它们的目的却是相同的：确定什么参数需要传递给free_area_init()函数（对于UMA体系结构）或者free_area_init_node()函数（对于NUMA体系结构）。这里省略掉NUMA体系结构的说明。

free_area_init()函数的参数：

unsigned long *zones_sizes: 系统中每个zone所管理的page的数量的数组。这个时候，还没能确定zone中那些page是可以分配使用的（free）。这个信息知道boot memory allocator完成之前还无法知道。

《深入理解LINUX内存管理》学习笔记（二）

初始化mem_map

mem_map是一个struct page的数组，管理着系统中所有的物理内存页面。在系统启动的过程中，创建和分配mem_map的内存区域。UMA体系结构中，free_area_init()函数在系统唯一的struct node对象contig_page_data中node_mem_map成员赋值给全局的mem_map变量。调用的关系图：

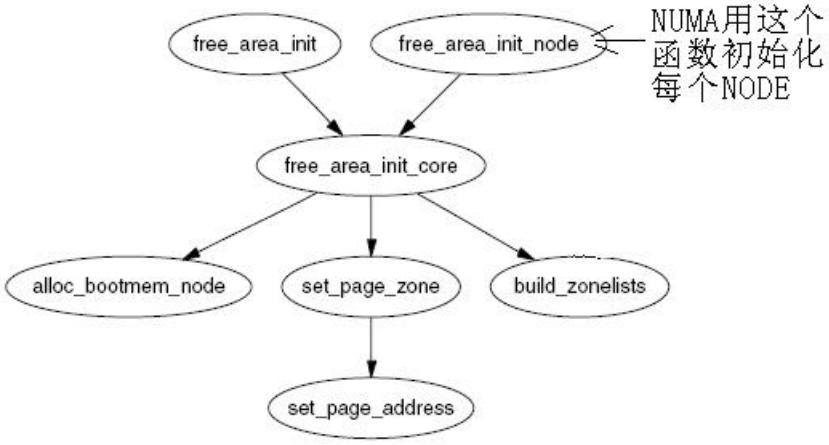


Figure 2.5. Call Graph: free_area_init()

主要的核心函数free_area_init_core(),为node的初始化过程分配本地的lmem_map（node->node_mem_map)。数组的内存在boot memory 分配的alloc_bootmem_node()函数分配.在UMA体系结构中，这个新分配的lmem_map成为全局的mem_map. 对于NUMA体系，lmem_map赋值给每一个node的node_mem_map成员，而这个情况下mem_map就被简单的赋值为PAGE_OFFSET(有兴趣理解NUMA体系结构的可以阅读英文原版，了解更多信息）。UMA体系中，node中的各个zone的zone_mem_map就指向mem_map中的某些元素作为zone所管理的第一个page的地址。

Page

系统中的每个物理页面用struct page数据结构对象来表示，并且跟踪page使用的状态：（省略了一些特定平台用到的成员）

```
struct page {
        unsigned long flags;
       atomic_t _count;
        union {
               atomic_t _mapcount;
              unsigned int inuse;
       };
        union {
           struct {
               unsigned long private;
               struct address_space *mapping;
           };
           struct kmem_cache *slab;    /* SLUB: Pointer to slab */
           struct page *first_page;    /* Compound tail pages */
        };
        union {
               pgoff_t index;          /* Our offset within mapping. */
               void *freelist;         /* SLUB: freelist req. slab lock */
        };
        struct list_head lru;
 #if defined
(WANT_PAGE_VIRTUAL
)
       void *virtual; #endif
};
union {
               atomic_t _mapcount;
              unsigned int inuse;
      } : 和页表转换有关的PTE链，下面章节将描述。
```

index：这个成员根据page的使用的目的有2种可能的含义。第一种情况：如果page是file mapping的一部分，它指明在文件中的偏移。如果page是交换缓存，则它指明在address_space所声明的对象:swapper_space（交换地址空间）中的偏移。第二种情况：如果这个page是一个特殊的进程将要释放的一个page块，则这是一个将要释放的page块的序列值，这个值在__free_page_ok()函数中设置。

mapping: 当文件或设备需要内存映射，文件或设备的inode对象有一个address_space类型的成员。如果page属于这个文件或设备，mapping将指向inode中这个成员。如果page不属于任何文件或设备，但是 mapping被设置了，则mapping指向了一个address_space类型的swapper_space对象，则page用于管理交换地址空间(swap address space)了。

lru: page交换调度策略使用。page可能被调度到active_list或者inactive_list队列里。就是使用lru这个list_head。

private：这个保存了一些和mapping（文件mapping到内存）有关的一些特定的信息。如果page是一个buffer page,则它就保存了一个指向buffer_head的指针。

virtual： 不再用于将high memory的映射到ZONE_NORMAL区域的作用了，除了一些其他的体系结构会用到外。

count: page的访问计数，当为0是，说明page是空闲的，当大于0的时候，说明page被一个或多个进程真正使用或者kernel用于在等待I/O。

flags: page状态的标志信息。kernel代码里定义了大量的宏用于设置，清楚，检测flag成员中的各个位所表示的page状态信息。特别提示一下，SetPageUptodate(),它需要调用一个和体系结构有关的函数：

arch_set_page_uptodate().

Bit Name	Description
PG_active	This bit is set if a page is on the active_list LRU and cleared when it is removed. It marks a page as being hot.
PG_arch_1	Quoting directly from the code, PG_arch_1 is an architecture-specific page state bit. The generic code guarantees that this bit is cleared for a page when it first is entered into the page cache. This allows an architecture to defer the flushing of the D-Cache (See Section 3.9) until the page is mapped by a process.
PG_checked	This is only used by the Ext2 file system.
PG_dirty	This indicates if a page needs to be flushed to disk. When a page is written to that is backed by disk, it is not flushed immediately. This bit is needed to ensure a dirty page is not freed before it is written out.
PG_error	If an error occurs during disk I/O, this bit is set.
PG_fs_1	This bit is reserved for a file system to use for its own purposes. Currently, only NFS uses it to indicate if a page is in sync with the remote server.
PG_highmem	Pages in high memory cannot be mapped permanently by the kernel. Pages that are in high memory are flagged with this bit during mem_init().
PG_launder	This bit is important only to the page replacement policy. When the VM wants to swap out a page, it will set this bit and call the writepage() function. When scanning, if it encounters a page with this bit and PG_locked set, it will wait for the I/O to complete.
PG_locked	This bit is set when the page must be locked in memory for disk I/O. When I/O starts, this bit is set and released when it completes.
PG_lru	If a page is on either the active_list or the inactive_list, this bit will be set.
PG_referenced	If a page is mapped and it is referenced through the mapping, index hash table, this bit is set. It is used during page replacement for moving the page around the LRU lists.
PG_reserved	This is set for pages that can never be swapped out. It is set by the boot memory allocator (See Chapter 5) for pages allocated during system startup. Later it is used to flag empty pages or ones that do not even exist.
PG_slab	This will flag a page as being used by the slab allocator.
PG_skip	This was used by some Sparc architectures to skip over parts of the address space but is no longer used. In 2.6, it is totally removed.
PG_unused	This bit is literally unused.
PG_uptodate	When a page is read from disk without error, this bit will be set.

Table 2.1. Flags Describing Page Status

Bit Name	Set	Test	Clear
PG_active	SetPageActive()	PageActive()	ClearPageActive()
PG_arch_1	None	None	None
PG_checked	SetPageChecked()	PageChecked()	None
PG_dirty	SetPageDirty()	PageDirty()	ClearPageDirty()
PG_error	SetPageError()	PageError()	ClearPageError()
PG_highmem	None	PageHighMem()	None
PG_launder	SetPageLaunder()	PageLaunder()	ClearPageLaunder()
PG_locked	LockPage()	PageLocked()	UnlockPage()
PG_lru	TestSetPageLRU()	PageLRU()	TestClearPageLRU()
PG_referenced	SetPageReferenced()	PageReferenced()	ClearPageReferenced()
PG_reserved	SetPageReserved()	PageReserved()	ClearPageReserved()
PG_skip	None	None	None
PG_slab	PageSetSlab()	PageSlab()	PageClearSlab()
PG_unused	None	None	None
PG_uptodate	SetPageUptodate()	PageUptodate()	ClearPageUptodate()

Table 2.2. Macros for Testing, Setting and Clearing page→flags Status Bits

映射页面到zone（Mapping page to zones）

在2.4.18内核之前，struct page数据结构中有一个zone的成员，后来证明这样做会无谓的浪费大量的内存空间，因为系统中会有大量的page对象，所以以后版本的page中不在有这样的成员了，而是有一个索引表示，这个索引保存在flag成员中的某些位段中，这个索引占用8个位。2.6.19版本的kernel系统中建立了一个全局的zone数组：

```
struct zone
  *zone_table
[1 <<ZONETABLE_SHIFT ] __read_mostly
; EXPORT_SYMBOL
(zone_table
 );
```

EXPORT_SYMBOL宏的作用，是让zone_table能够被其他载入的模块使用。free_area_init_core()函数对node里的所有page做初始化。 zone_table[nid * MAX_NR_ZONES + j] = zone; //对zone_table做初始化。 nid是node ID。 j是zone的索引。 对每个page调用set_page_zone()初始化page中的zone的索引值（在page->flag中）。 set_page_zone(page, nid * MAX_NR_ZONES + j); 但是2.6.20后就不用这一套了，mm/sparse.c文件中做了一套管理系统。新的方法将多个page组成section来管理。

这里略微描述一下，有兴趣的，可以详细阅读sparse.c的源代码。kernel将所有的page分成多个section管理，对于x86平台，有64个section，每个section管理着(1<<26)个或(1<<30)个（对于支持PAE的情况下）内存区域。

以下是几个主要的define：

```
include/asm-x86/sparsemem_32.h:

#ifdef CONFIG_X86_PAE
 #define SECTION_SIZE_BITS       30
 #define MAX_PHYSADDR_BITS       36
 #define MAX_PHYSMEM_BITS        36
 #else
 #define SECTION_SIZE_BITS       26
 #define MAX_PHYSADDR_BITS       32
 #define MAX_PHYSMEM_BITS        32
 #endifinclude/linux/mmzone.h:#define SECTIONS_SHIFT
    (MAX_PHYSMEM_BITS - SECTION_SIZE_BITS)#define NR_MEM_SECTIONS
    (1UL << SECTIONS_SHIFT)#ifdef CONFIG_SPARSEMEM_EXTREME
 #define SECTIONS_PER_ROOT
     (PAGE_SIZE / sizeof (struct mem_section))
 #else
 #define SECTIONS_PER_ROOT       1
 #endif#define SECTION_NR_TO_ROOT(sec) ((sec) / SECTIONS_PER_ROOT)
 #define NR_SECTION_ROOTS        (NR_MEM_SECTIONS / SECTIONS_PER_ROOT)
 #define SECTION_ROOT_MASK       (SECTIONS_PER_ROOT - 1)

首先声明了一个全局的mem_section的全局数组。
 struct mem_section *mem_section[NR_SECTION_ROOTS] ;

调用sparse_add_one_section()函数，分配mem_section，并且初始化。
```

《深入理解LINUX内存管理》学习笔记（三）

页表管理（Page table management)

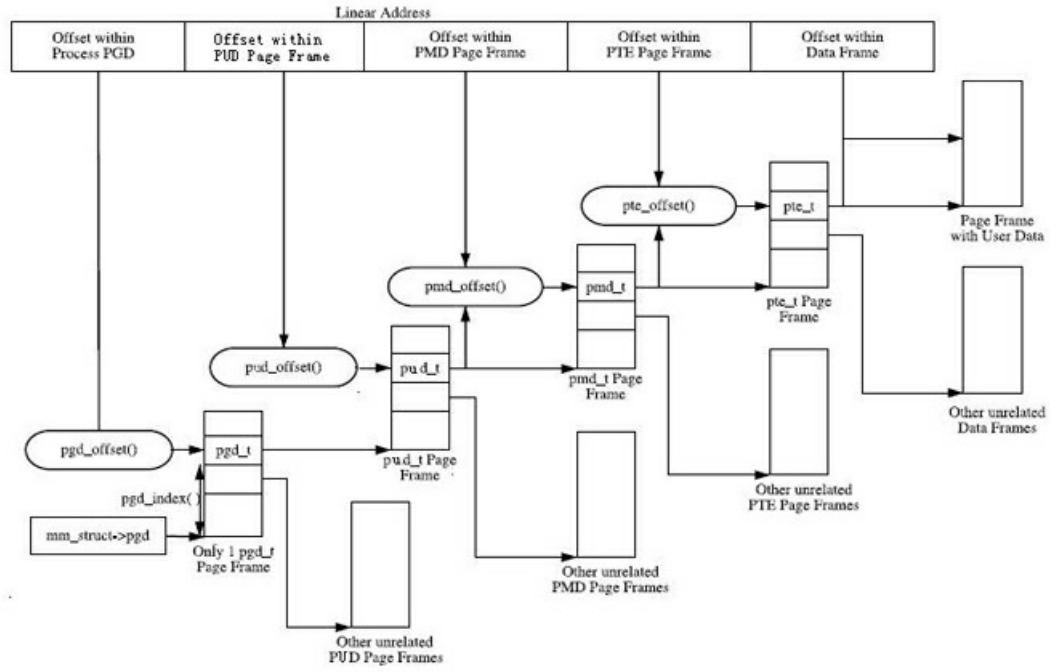
Linux内核软件架构习惯与分成硬件相关层和硬件无关层。对于页表管理，2.6.10以前（包括2.6.10）在硬件无关层使用了3级页表目录管理的方式，它不管底层硬件是否实现的也是3级的页表管理：

- Page Global Directory (PGD)
- Page Middle Directory (PMD)
- Page Table (PTE)

从2.6.11开始，为了配合64位CPU的体系结构，硬件无关层则使用了4级页表目录管理的方式：

- Page Global Directory (PGD)
- Page Upper Directory (PUD)
- Page Middle Directory (PMD)
- Page Table (PTE)

PGD每个条目中指向一个PUD，PUD的每个条目指向一个PMD，PMD的每个条目指向一个PTE，PTE的每个条目指向一个页面(Page)的物理首地址。因此一个线性地址被分为了5个部分，如下图：



PGD，PUD，PMD，PTE中到底有几个条目，不同的CPU体系结构有不同的定义。

虽然硬件无关层是这么设计的，但是底层硬件未必也是这样实现的。如x86体系结构，如果不使用PAE（Physical Address Extension)特性，则硬件底层实现的是2级的页表目录管理，事实上，只有PGD，PTE才是真正有意义的。

页目录（Page directory）

每个进程所代表的上下文数据结构中都有一个指针（mm_struct->pgd），其指向这个进程所使用的PGD的一个页（page frame）。这个页面中包含了一个类型为pgd_t的数组。pgd的载入到CPU的方式完全和体系结构相关。x86，进程的页表地址从mm_struct->pgd载入到CR3寄存器，载入页表地址的同时，会引起TLB（快表，是对页目录，页表缓存的缓冲区）也被强制刷新。事实上，这也是__flush_tlb()函数，实现的机制。

PGD中的每个条目指向一个页（page frame），这个页是“由类型为pud_t的条目组成的PUD”。 PUD中的每个条目同样指向一个页，这个页是“由类型为pmd_t的条目组成的PMD”。PMD的每个条目指向一个页，这个页是“由类型为pte_t的条目组成的PTE”。PTE的每个条目就指向了真正的数据或指令所在的页面的首地址的了，这也不是100%的，如果所需要的页面被交换到磁盘空间去后，这个条目就包含的内容是在当page fault发生后，传入需要调用的 do_swap_page()函数，找到包含页面数据的交换空间。

将线性地址转换成物理地址，需要将线性地址分成5个部分，其中4个的值是在各级页表中的索引或者也可以看成是偏移(OFFSET)，另外一个是数据在页中的偏移。为了分别析出这5个部分，各级页表和页中偏移都拥有特定的几个宏：SHIFT，SIZE和MASK。SHIFT宏表示各级页表或页中偏移所占用的bit数。

MASK的值和线性地址做AND运算，获得一个各级的高位部分，一般用于页面，页表对齐。SIZE宏表示各级所能管理的内存空间的字节数。

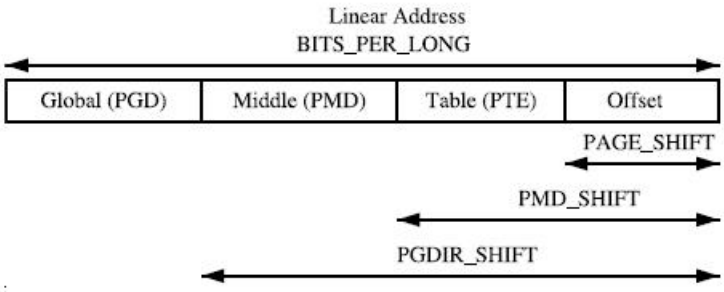


Figure 3.2. Linear Address Bit Size Macros

MASK和SIZE都是有SHIFT计算得到，如x86体系结构是这样的：

#define PAGE_SHIFT 12

#define PAGE_SIZE (1UL << PAGE_SHIFT)

#define PAGE_MASK (~ (PAGE_SIZE - 1))

PAGE_SHIFT是线性地址中偏移(offset)的位的位数，x86系统是12位。page的字节数计算很简单：2PAGE_SHIFT（和1<<PAGE_SHIFT是同样的结果）。如果需要对一个地址做页边界的对齐，则使用PAGE_ALIGN()宏，这个宏将地址加上PAGE_SIZE-1再和PAGE_MASK做AND操作即可。事实上PAGE_ALIGN()宏是和下一个页的边界对齐的。

PMD_SHIFT是线性地址中第三级页表的所占的位数，PMD_SIZE和PMD_MARK是由这个宏计算得到的。

PUD_SHIFT是线性地址中第二级页表的所占的位数，PUD_SIZE和PUD_MARK是由这个宏计算得到的。

PGD_SHIFT是线性地址中第一级页表的所占的位数，PGD_SIZE和PGD_MARK是由这个宏计算得到的。

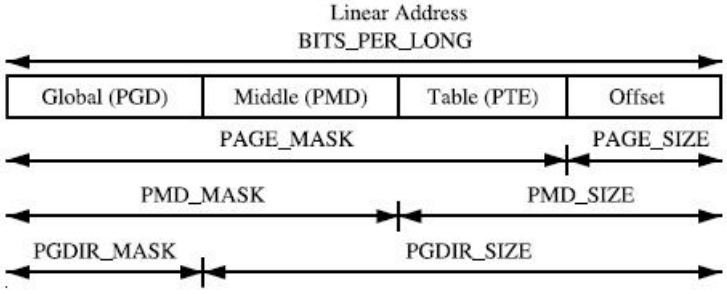


Figure 3.3. Linear Address Size and Mask Macros

最后介绍4个重要的宏：PTRS_PER_PGD,PTRS_PER_PUD, PTRS_PER_PMD,PTRS_PER_PTE。它们用于确定每级页表有多少条目。

不使能PAE特性的x86体系结构这几个宏定义如下：

```
#define PTRS_PER_PGD  1024
 #define PTRS_PER_PUD    1  //这种情况下PUD事实不起作用，为了代码的硬件无关性，设置为1。

                        //在include/asm-generic/pgtable-nopud.h中定义
 #define PTRS_PER_PMD  1  //这种情况下PMD事实不起作用，为了代码的硬件无关性，设置为1。

                       //在include/asm-generic/pgtable-nopmd.h中定义

#define PTRS_PER_PTE  1024
```

页表条目(Page table entry)

页表的每个条目都是一个声明为数据结构的对象：pgd_t,pud_t,pmd_t和pte_t分别对应PGD,PUD,PMD和PTE。

虽然这些数据结构常常只有一个无符号整数，它们被定义成数据结构有2个原因：第一，类型保护，防止被不合适的方式使用。第二，容易扩展每个条目所占字节的数量，如x86使能PAE，则需要另外加入4位(原书是说4位,但是我觉得应该是错误的，应该是加入了4个字节），以使得能够访问多余4GB的物理内存。

为了保持一些保护位，定义了pgprot_t数据结构，它保存相关的标志，通常会保持在页表条目的低位区域。

为了类型的计算，在文件asm/page_32.h或者asm/page_64.h中定义了5个宏。传入上述的类型，返回相应的数据结构中的部分数值：pte_val(),pmd_val(),pud_val()和pgprot_val(). 相反的操作的计算的宏：__pte(),__pmd(),__pud(),__pgd()和__pgprot()。

条目中的状态位，完全是和体系结构相关的。下面解释一下不使能PAE的x86体系结构下，各个状态位的含义。

没有使能PAE的x86，pte_t数据结构中只有一个32位的整数。每个PTE中的类型为pte_t的指针指向一个页面的首地址，也就是说指向的地址总是页面对齐的。因此，在这个整数中PAGE_SHIFT指定数目的位数，也就是12位，是给页表条目中的状态位。列表如下：

Bit	Function
_PAGE_PRESENT	Page is resident in memory and not swapped out.
_PAGE_PROTNONE	Page is resident, but not accessible.
_PAGE_RW	Set if the page may be written to
_PAGE_USER	Set if the page is accessible from userspace
_PAGE_DIRTY	Set if the page is written to
_PAGE_ACCESSED	Set if the page is accessed

Table 3.1. Page Table Entry Protection and Status Bits

比较费解的是_PAGE_PROTNONE这个状态位，x86的体系结构上并不存在这个状态位，LINUX内核借用了PAT位作为这个来使用。这里还有一个问题如果有PSE位被设置，则PAT位的位置就会使用另外一个位置，幸运的是，LINUX内核不会在用户页面中使用PSE特性

LINUX内核挪用这个位的目的是：确定一个虚拟内存的页面在物理内存中是存在的，但是用户空间的进程不能访问它，如同对一段内存区域调用mprotect() API函数并传入PROT_NONE标志一样。当一段内存区域被要求保护，_PAGE_PRESENT为被清除，_PAGE_PROTNONE位被置一。pte_present()宏会同时检测这2位的设置情况，让kernel能够自己知道对应的PTE是否可用：

#define pte_present(x) ((x).pte_low & (_PAGE_PRESENT | _PAGE_PROTNONE))

如果正好是用户空间不能访问的页面，这就相当巧妙了，但是也相当的重要考量。因为硬件状态为_PAGE_PRESENT已经被清除，当试图访问这个页面的时候，会产生一个page fault的异常，LINUX内核强制的保护了页面访问，但是内核还是知道页面是存在的，如果需要交换到磁盘或者进程退出释放页面，能够做出正确的动作。

《深入理解LINUX内存管理》学习笔记（四）

页表条目的操作

X86体系结构的情况下，在include/asm-x86/pgtable.h文件中，定义了“析出”或者“检查”页表条目中的值的几个宏(在2.6.24版本的内核中，由于体系结构的关系，这几个宏可能分布在几个相关的头文件中）。

通过 4 个宏，把一个线性地址从第一级页目录表 (Page directories) 追巡最后一级页目录表。

pgd_offset: 通过线性地址 (其中的一部分指出了需要访问的内存地址在的 PGD 中的索引) 和进程的 mm_struct 数据结构对象，返回一个指向 PGD 条目的地址，内容是某个 PUD 页面的首地址。

pud_offset: 通过线性地址（也是索引）和指向 PGD 条目的地址，返回一个指向 PUD 条目的地址，内容是某个 PMD 页面的首地址。如果硬件系统并不支持 PUD ，则直接返回指向 PGD 条目的地址。也就是通过巧妙的直接返回 PGD 的方式，使得不

同体系结构下，统一的软件架构。

pmd_offset: 通过线性地址（也是索引）和指向 PUD 条目的地址，返回一个指向 PMD 条目的地址，内容是某个 PTE 页面的首地址。如果硬件系统并不支持 PMD ，则直接返回指向 PUD 条目的地址。也就是通过巧妙的直接返回 PUD 的方式，使得不同体系结构下，统一的软件架构。

pte_offset: 通过线性地址（也是索引）和指向 PMD 条目的地址（内容是 PMD 页面的首地址），返回一个指向 PTE 条目的地址，内容是某个需要访问的数据内存页面的首地址（物理地址），这个地址和线性地址的低位部分的数据在内存页面中的偏移相加，就获得了数据真正所在的物理地址了。

第二组宏，用于检测页表条目是否存在或者是否可用的信息。

- pte_none(), pmd_none(), pud_none(),pgd_none() ，如果对应的条目不存在，则返回 1 。宏的定义只是检测条目的内容是否全 0.
- pte_present(), pmd_present(), pud_present(),pgd_present(), 如果条目中的 PRESENT 位被设置，则返回 1.
- pte_clear(), pmd_clear(), pud_clean(),pgd_clear(), 对相应条目清零。
- pmd_bad(), pud_bad(),pgd_bad(),