Linux 内存管理导读

1. 存储层次结构和 x86 存储管理硬件（MMU）

1.1 存储层次

高速缓存(cache) → 主存(main memory) → 磁盘(disk)

理解存储层次结构的根源：CPU 速度和存储器速度的差距。

层次结构可行的原因：局部性原理。

LINUX 的任务:

- 减小 footprint，提高 cache 命中率，充分利用局部性。
- 实现虚拟存储以满足进程的需求，有效地管理内存分配，力求最合理地利用有限的资源。

1.2 MMU 的作用

辅助操作系统进行内存管理，提供虚实地址转换等硬件支持。

1.3 x86 的地址

逻辑地址：出现在机器指令中，用来制定操作数的地址。段：偏移

线性地址：逻辑地址经过分段单元处理后(如将段地址左移四位与偏移量）得到线性地址，这是一个 32 位的无符号整数，可用于定位 4G 个存储单元。

物理地址：线性地址经过页表查找后得出物理地址，这个地址将被送到地址总线上指示所要访问的物理内存单元。

LINUX: 尽量避免使用段功能以提高可移植性。如通过使用基址为 0 的段，使逻辑地址==线性地址。

1.4 x86 的段与 Linux 下的段比较

x86 保护模式下的段：表索引+描述符项。不仅仅是一个基地址的原因是为了提供更多的信息：保护、长度限制、类型等。描述符项存放在一张表中(GDT 或 LDT)，表索引就是表的索引。段寄存器中存放的是表索引，在段寄存器装入的同时，描述符中的数据被装入一个不可见的寄存器以便 cpu 快速访问。

专用寄存器：GDTR(包含全局描述附表的首地址),
LDTR（当前进程的段描述附表首地址）,
TSR（指向当前进程的任务状态段）

LINUX 使用的四种段：

__KERNEL_CS： 内核代码段。范围 0-4G。可读、执行。DPL=0。

__KERNEL_DS：内核代码段。范围 0-4G。可读、写。DPL=0。

__USER_CS：内核代码段。范围 0-4G。可读、执行。DPL=3。

__USER_DS：内核代码段。范围 0-4G。可读、写。DPL=3。

TSS(任务状态段)：存储进程的硬件上下文，进程切换时使用。（因为 x86 硬件对 TSS 有一定支持，所有有这个特殊的段和相应的专用寄存器。）

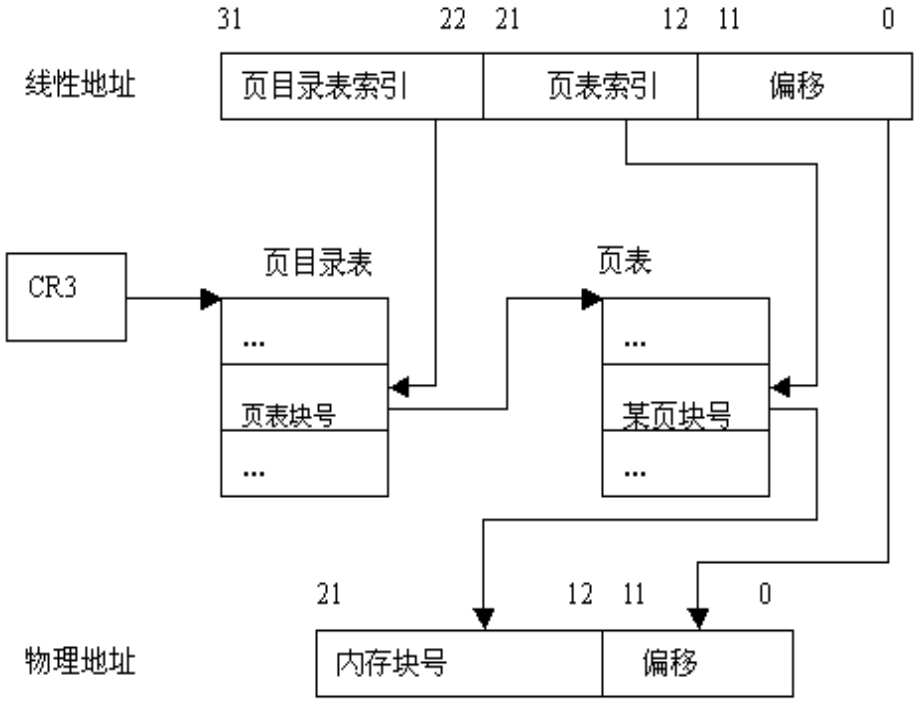
__USER_CS 和__USER_DS 段都是被所有在用户态下的进程共享的。但这种段的共享和进程空间的共享是有区别的：虽然各个进程使用同一个（种）段，但通过使

用不同的页表由分页机制保证了进程空间仍然是独立的。

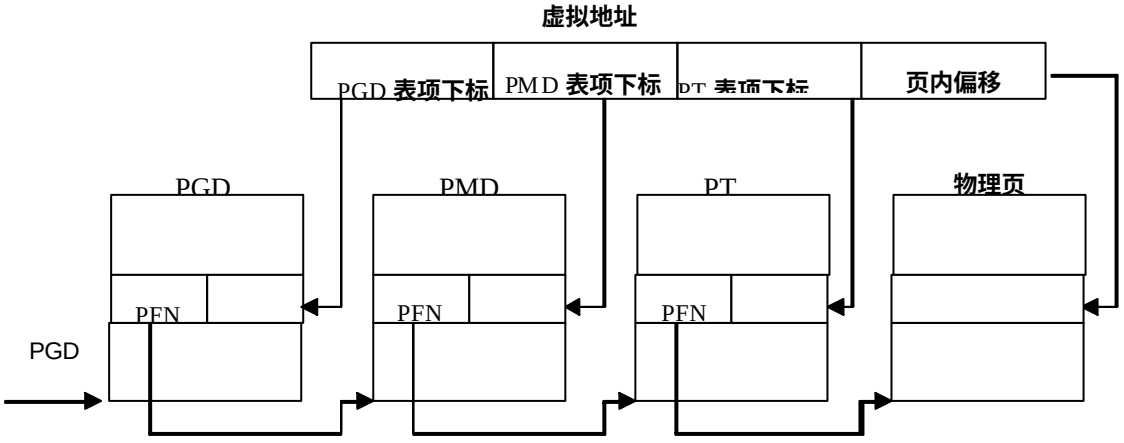
1.5 x86 的分页机制

x86 硬件支持两级页表，奔腾 pro 以上的型号还支持 Physical Address Extension Mode（PAE）和三级页表。所谓的硬件支持包括一些特殊寄存器(cr0-cr4)、以及 CPU 能够识别页表项中(页表项的高 20 位是页面索引，而低 12 为标志位)的一些标志位并根据访问情况做出反应等等。如读写 Present 位为 0 的页或者写 Read/Write 位为 0 的页将引起 CPU 发出 page fault 异常，访问完页面后自动设置 accessed 位等。

具体可参见《Paging Extensions for the Pentium Pro Processor (Overview).htm》



Linux 采用的是一个体系结构无关的三级页表模型（如图），使用一系列的宏来掩盖各种平台的细节。例如，为了适应 x86 的二级页面映射机制，Linux 内核通过把 PMD 看作只有一项的表并存储在 pgd 表项中（通常 pgd 表项中存放的应该是 pmd 表的首地址），页表的中间目录(pmd)被巧妙地‘折叠’到页表的全局目录(pgd)，从而适应了二级页表硬件。



1.6 TLB

TLB 全称是 Translation Look-aside Buffer,用来加速页表查找。这里关键的一点是：如果操作系统更改了页表内容，它必须相应的刷新 TLB 以使 CPU 不误用过时的表项。

以下是从 Intel 技术文档中摘出的图，是一个 TLB 下的线性地址映射过程图

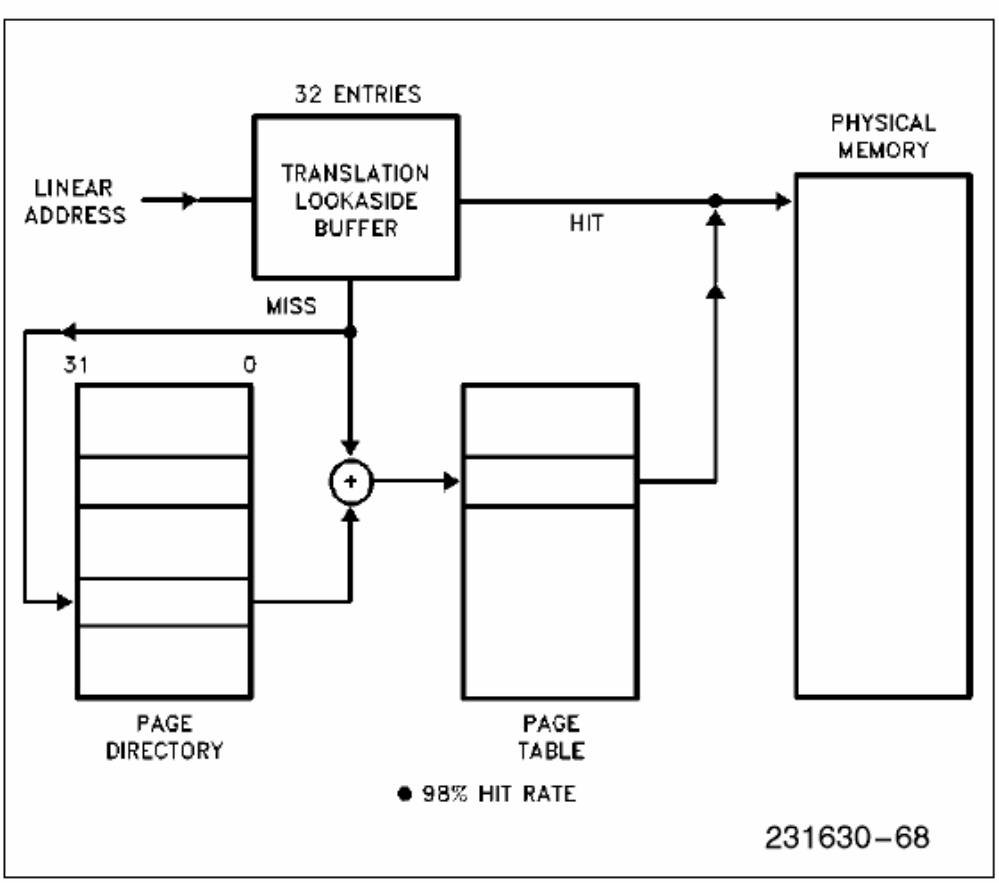


Figure 4-22. Translation Lookaside Buffer

1.7 Cache

Cache 基本上是对程序员透明的，但是不同的使用方法可以导致大不相同的性能。linux 有许多关键的地方对代码做了精心优化，其中很多就是为了减少对 cache 不必要的污染。如把只有出错情况下用到的代码放到.fixup section，把频繁同时使用的数据集中到一个 cache 行（如 struct task_struct），减少一些函数的存储器足迹（footprint），在 slab 分配器里头的 slab coloring 等。

另外，我们也必须知道什么时候 cache 要无效：新 map/remap 一页到某个地址、页面换出、页保护改变、进程切换等，也即当 cache 对应的那个地址的内容或含义有所变化时。当然，很多情况下不需要无效整个 cache，只需要无效某个地址或地址范围即可。实际上，intel 在这方面做得非常好用，cache 的一致性完全由硬件维护。

- 关于 x86 处理器更多信息，请参照其手册：Volume 3: Architecture and Programming Manual,可以从ftp://download.intel.com/design/pentium/MANUALS/24143004.pdf获得

1.8 Linux 相关实现

这一部分的代码和体系结构紧密相关，因此大多位于 arch 子目录下，而且大量以宏定义和 inline 函数形式存在于头文件中。以 i386 平台为例，主要的文件包括：

1.8.1 page.h

页大小、页掩码定义。主要是定义了一些宏，如 PAGE_SIZE,PAGE_SHIFT 和 PAGE_MASK。对页的操作，如清除页内容 clear_page、拷贝页 copy_page、页对齐 page_align 还有内核虚地址的起始点： PAGE_OFFSET（0xC0000000）和相关的内核中虚实地址转换的宏__pa 和__va.。

宏 virt_to_page 从一个内核虚地址得到该页的描述结构 struct page *。所有物理内存都由一个 memmap 数组来描述。这个宏就是通过计算给定地址的物理页在这个数组中的位置。另外这个文件也定义了一个简单的宏检查一个页是不是合法：VALID_PAGE(page)。如果 page 离 memmap 数组的开始太远以至于超过了最大物理页面应有的距离则是不合法的。

但页表项的定义也放在这里。有 pgd_t，pmd_t，pte_t 和存取它们值的宏 pte_val 等

1.8.2 pgtable.h　pgtable-2level.h　pgtable-3level.h

这些文件就是处理页表的，它们提供了一系列的宏来操作页表。pgtable-2level.h 和 pgtable-2level.h 则分别对应 x86 二级、三级页表的需求。在编译内核的时候通过编译条件来选择具体哪个文件。首先当然是表示每级页表有多少项的定义不同了。而且在 PAE 模式下，地址超过 32 位，页表项 pte_t 用 64 位来表示(pmd_t,pgd_t 不需要变)，一些对整个页表项的操作也就不同。

pgtable.h 的宏很多，但都比较直观，通常从名字就可以看出宏的意义。如 pte_xxx 宏的参数是 pte_t,而 ptep_xxx 的参数是 pte_t *。

pgtable.h 里除了页表操作的宏外，还有 cache 和 tlb 刷新操作，因为他们常常是在页表操作时使用。但关于 cache 的宏实现全是空的，前面也讲过基本都有硬件做了。

- #define flush_cache_all()　　do { } while (0)
- #define flush_cache_mm(mm)　　do { } while (0)

关于 tlb 操作是以__开始的，也就是说，内部使用的，真正对外接口在 pgalloc.h 中。

1.8.4 pgalloc.h

包括页表项的分配和释放宏/函数,值得注意的是表项高速缓存的使用：

pgd/pmd/pte_quicklist

还有上面提到的 tlb 刷新的接口

1.8.5 segment.h

定义 Linux 中使用的四个段的

```
#define __KERNEL_CS 0x10
#define __KERNEL_DS 0x18

#define __USER_CS      0x23
#define __USER_DS      0x2B
```

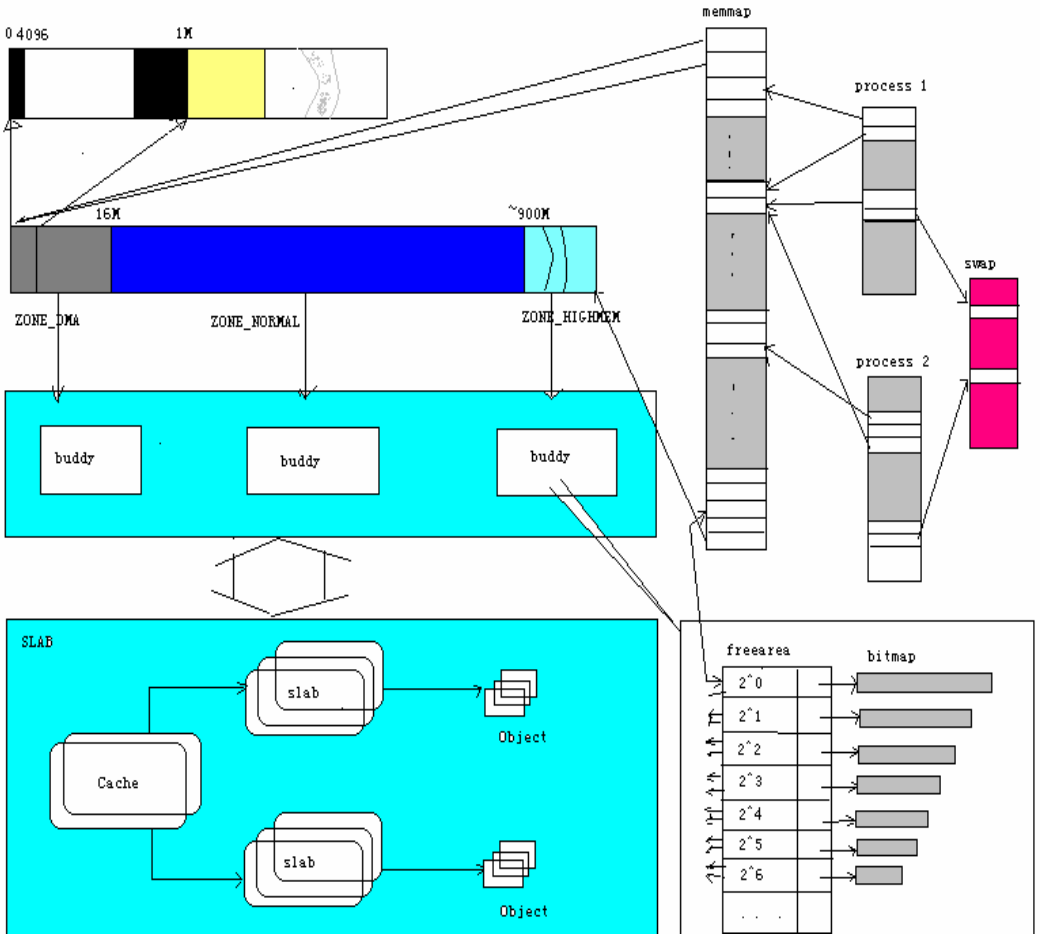
2. 物理内存的管理

2.1 物理内存数据结构及其联系

2.4.0 内核中内存管理有很大的变化。在物理页面管理上实现了基于区的伙伴系统（zone based buddy system）。区(zone)的是根据内存的不同使用类型划分的。对不同区的内存使用单独的伙伴系统(buddy system)管理,而且独立地监控空闲页等。

实际上更高一层还有 numa 支持。Numa(None Uniformed Memory Access)是一种体系结构，其中对系统里的每个处理器来说,不同的内存区域可能有不同的存取时间(一般是由内存和处理器的距离决定)。而一般的机器中内存叫做 DRAM，即动态随机存取存储器，对每个单元，CPU 用起来是一样快的。NUMA 中访问速度相同的一个内存区域称为一个 Node，支持这种结构的主要任务就是要尽量减少 Node 之间的通信，使得每个处理器要用到的数据尽可能放在对它来说最快的 Node 中。2.4.0 内核中 node—相应的数据结构是 pg_data_t，每个 node 拥有自己的 mem_map 数组，同时把自己节点的内存分成几个 zone，每个 zone 再用独立的伙伴系统管理物理页面。

一些重要的数据结构粗略地表示如下：



2.2 基于区的伙伴系统的设计—物理页面的管理

内存分配的两大问题是：分配效率、碎片问题。一个好的分配器应该能够快速的满足各种大小的分配要求，同时不能产生大量的碎片浪费空间。伙伴系统是一个常用的比

较好的算法。

引入区的概念是为了区分内存的不同使用类型，以便更有效地利用它们。2.4.0 内核中有三个区：DMA，Normal，HighMem。

DMA 区在 x86 体系结构中通常是小于 16 兆的物理内存区，因为 DMA 控制器只能使用这一段的内存。因为内核能访问的虚拟空间只有 1G，而且其中又有 128M 用来作为交换空间，所以实际能访问的物理内存是 896M，所以从 16M～896M 为 Normal 区。HighMem 是物理地址超过 896M 的高端内存。

由于 linux 实现的原因，高地址的内存不能直接被内核使用，如果选择了 CONFIG_HIGHMEM 选项，内核会使用一种特殊的办法来使用它们。HighMem 只用于 page cache 和用户进程。这样分开之后，我们将可以更有针对性地使用内存，而不至于出现把 DMA 可用的内存大量给无关的用户进程使用导致驱动程序没法得到足够的 DMA 内存等情况。此外，每个区都独立地监控本区内存的使用情况（每个区都有空闲页面队列 free_area[]和不活跃干净队列 inactive_clean_list），分配时系统会判断从哪个区分配比较合算，综合考虑用户的要求和系统现状。2.4.0 内核里分配页面时可能会和高层的 VM 代码交互(分配时根据空闲页面的情况，内核可能从伙伴系统里分配页面，也可能直接把已经分配的页收回—reclaim 等)。

整个分配器的主要接口是如下函数(mm.h page_alloc.c)：

- struct page * alloc_pages(int gfp_mask, unsigned long order)　根据 gftp_mask 的要求，从适当的区分配 2^order 个页面，返回第一个页的描述符指针。
- #define alloc_page(gfp_mask)　alloc_pages(gfp_mask,0)
- unsigned long __get_free_pages((int gfp_mask, unsigned long order)
  工作同 alloc_pages,但返回首地址。
- #define __get_free_page(gfp_mask) __get_free_pages(gfp_mask,0)
- get_free_page 分配一个已清零的页面。
- __free_page(s) 和 free_page(s)释放页面（一个/多个）前者以页面描述符为参数，后者以页面地址为参数。

http://home.earthlink.net/~jknapka/linux-mm/zonealloc.html。

2.3 Slab—内核缓冲区管理

2.3.1 slab 机制简介

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 k 不等，都取整到 2 的幂次个页面那是完全不现实的。2.0 的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。这样做的效率不高。有许多地方可以改进：

- 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。
- 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。
- 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类似 2.0 内核中的取整到 2 的幂次的通用缓存。
- 使用 2 的幂次大小的内存区域时高速缓存冲突的概率较大，有可能通过仔细安

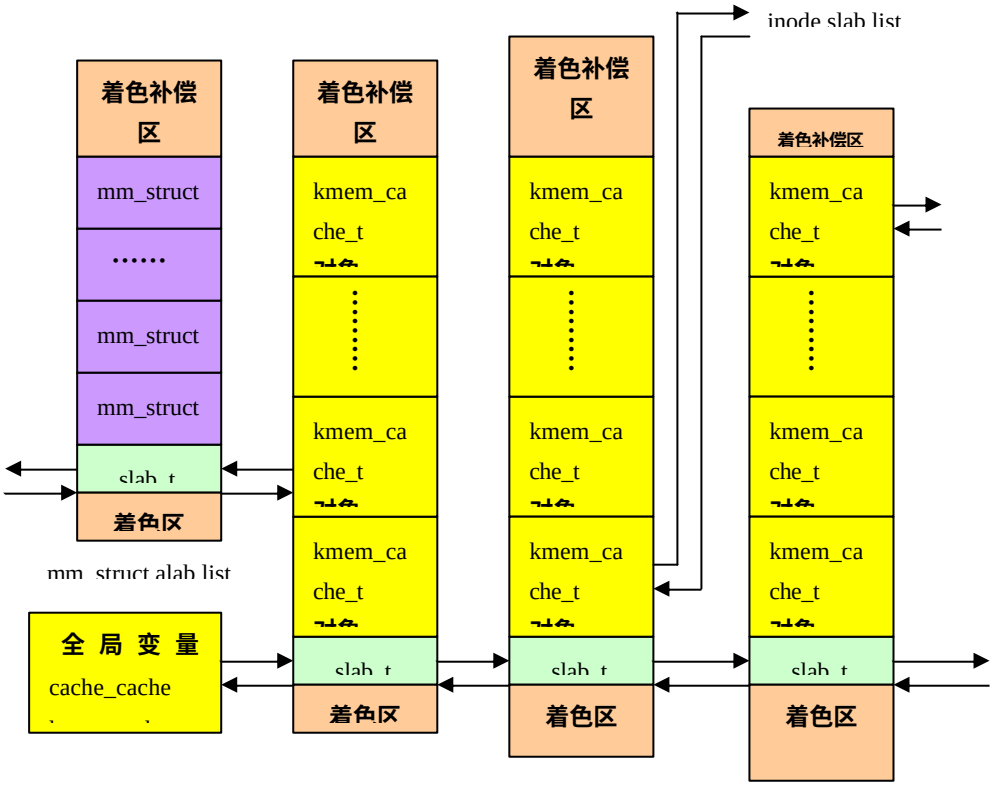
排内存区域的起始地址来减少高速缓存冲突。

- 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高速缓存污染。

2.2.0 内核实现的 slab 分配器体现了这些改进思想。

2.3.2 实例：小对象 slab 结构示意图

以下这个图体现了内核中小对象的 slab list 是如何组织的，内核中有一个 kmem_cache_t 全局变量 cache_cache 来作为最高成的管理；同时图中也反应了一个 slab list 的结构，以及一个 slab_t 所管理的对象之间的关系。



2.3.3 相关接口

kmem_cache_create/kmem_cache_destory

kmem_cache_grow/kmem_cache_reap　增长/缩减某类缓存的大小

kmem_cache_alloc/kmem_cache_free　从某类缓存分配/释放一个对象

kmalloc/kfree　通用缓存的分配、释放函数。

相关参考：

http://www.lisoleg.net/lisoleg/memory/slab.pdf　：Slab 发明者的论文。

AKA2000 年的讲座也有这个主题，请访问 aka 主页：www.aka.org.cn

2.4 页面的定期换出

2.4.1 两个重要线程“ kswapd ”和“ kreclaim ”

在系统初始化时，kswapd_init 创建了两个线程用于内存页面的置换管理，分别是“ kswapd ”和“ kreclaimd ”两个线程。

kswapd 的任务：当空闲页面低于一定值时，从 inactive_dirty_list、进程的地址空间以

及各类 cache 回收页面。

kreclaimd 的任务：从 inactive_clean_list **回收页面，由**__alloc_pages **唤醒。**

2.4.2 相关文件

mm/swap.c　kswapd **使用的各种参数以及操作页面年龄的函数。**

mm/swap_file.c　**交换分区/文件的操作。**

mm/page_io.c　**读或写一个交换页。**

mm/swap_state.c　swap cache **相关操作,加入/删除/查找一个** swap cache **等。**

mm/vmscan.c　**扫描进程的** vm_area**，试图换出一些页面（**kswapd **线程所作)。**

其中 reclaim_page**：从** inactive_clean_list **回收一个页面，放到** free_list

kclaimd **被唤醒后重复调用** reclaim_page **直到每个区的** zone->free_pages>= zone->pages_low

page_lauder **：由**__alloc_pages **和** try_to_free_pages **等调用。通常是由于** freepages + inactive_clean_list **的页太少了。**

功能 把 inactive_dirty_list **的页面转移到** inactive_clean_list**，首先把已经被写回文件或者交换区的页面**(by bdflush)**放到** inactive_clean_list**，如果** freepages **确实短缺，唤醒** bdflush**，再循环一遍把一定数量的** dirty **页写回。**

关于这几个队列(active_list，inactive_dirty_list，inactive_clean_list)**的逻辑，我的手写稿有一部分。**

2.5 页面的异常处理

2.5.1 页面异常情况

（1**）越界访问**

- **页面没有映射→这种情况就调用**__alloc_page **页面分配**
- **页面不在内存中→这种情况就调用** do_swap_page **页面换入**

（2**）越权访问**

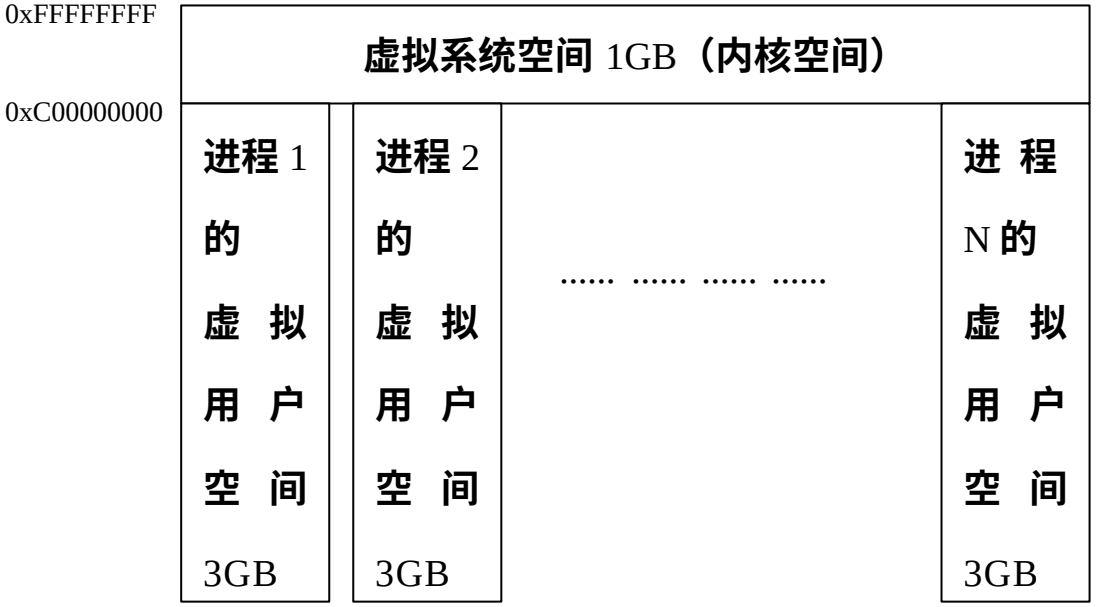
2.5.2 页面异常处理过程

当出现页面异常时，处理器会产生中断请求，由异常响应机制最终可以进入页面异常服务函数 do_page_fault()**，这个函数判断错误情况然后调用相应的函数来处理。**

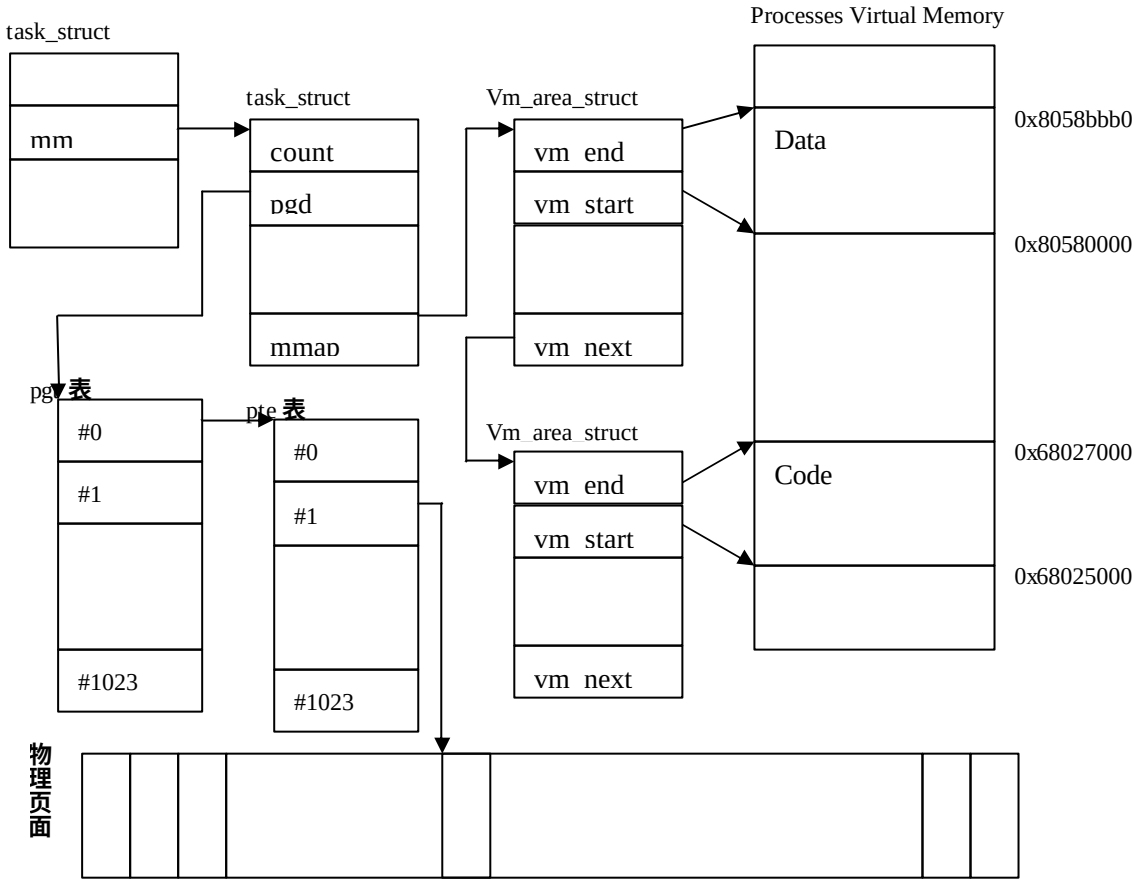
具体的中断响应这里不再介绍。

3. Linux 2.4.0 内核的虚拟内存管理

3.1 进程虚存空间示意图



3.2 虚拟内存管理中的几个数据结构之间的关系图



3.3 进程地址空间管理

3.3.1 虚拟区间的几个操作

进程的虚拟地址管理相对比较简单，主要有:

- 系统进程（内核空间）虚拟区间的创建和撤销，vmalloc 和 vfree 两个函数
- 用户进程（用户空间）虚拟区间的创建和撤销，brk 系统调用。brk 系统调用既可以用来创建虚拟区间也可以用来撤销。
- 内存映射，mmap 系统调用。mmap 系统调用分配一块虚拟区间并与文件等建立映射，这使得进程能访问文件，也能通过为两个进程建立相同的虚拟区间来实现共享内存的进程通信方式。
- 对虚拟区间的管理。主要有如修改区间属性、当撤销时修补区间等等。

3.3.2 相关文件

include/linux/mm.h：

- struct page 结构的定义，page 的标志位定义以及存取操作宏定义。
- struct vm_area_struct 定义。
- 内存管理子系统的一些函数原型声明。

include/linux/mman.h:

- vm_area_struct 结构的相关的常量宏定义。

memory.c：

- handle_mm_fault：页面异常处理相关的函数；
- 对一个区域 vm_area 的页表相关操作:
  zeromap_page_range: 把一个范围内的页全部映射到 zero_page
  remap_page_range：给定范围的页重新映射到另一块地址空间。
  zap_page_range：把给定范围内的用户页释放掉，页表清零。
- 其他等

mmap.c:：mmap/munmap/brk 系统调用。

- sys_brk: brk 系统调用的主要实现函数，申请虚拟区间并申请物理页面与虚拟
  区间建立映射；
- do_mmap_pgoff：mmap 实现体。
- 其他一些关于 vm_area_struct 的操作。

mprotect.c：

- mprotect 系统调用，修改一段虚拟区间的保护属性。

4. Linux 的 Cache 管理

4.1 page cache、buffer cache 和 swap cache 即其它

- page cache：读写文件时文件内容的 cache，大小为一个页。不一定在磁盘上连续。
- buffer cache：读写磁盘块的时候磁盘块内容的 cache，buffer cache 的内容对应磁盘上一个连续的区域，一个 buffer cache 大小可能从 512(扇区大小)到一个页。
- swap cache：是 page cache 的子集。用于多个进程共享的页面被换出到交换区的情况。
- 其它 cache：
  - pgd_quicklist: 全局变量用于快速分配 pdg，直接将该 cache 中的项分配
  - pmd_quicklist: 全局变量用于快速分配 pmd，直接将该 cache 中的项分配
  - pte_quicklist: 全局变量用于快速分配 pte，直接将该 cache 中的项分配
  - mmap_cache: 局部于 mm_struct 结构，是用户进程虚拟区间的 cache

4.2 cache 之间的关系

4.2.1 page cache 和 buffer cache 的关系

本质上是很不同的，buffer cache 缓冲磁盘块内容，page cache 缓冲文件的一页内容。page cache 写回磁盘时会使用临时的 buffer cache 来写磁盘。

4.2.2 两个与 cache 有关的线程：

bdflush： 把脏的 buffer cache 写回磁盘。通常只当脏的 buffer 太多或者需要更多的 buffer 而内存开始不足时运行。page_lauder 也可能唤醒它。

kupdate： 定时运行，把写回期限已经到了的脏 buffer 写回磁盘。

4.3 2.4.0 内核中针对 cache 的改进

在 2.4.0 内核中，page cache 和 buffer cache 耦合得更好。

在 2.2.0 内核里，磁盘文件的读使用 page cache，而写绕过 page cache，直接使用 buffer cache，因此带来了同步的问题：写完之后必须使用 update_vm_cache()更新可能有的 page cache。

2.4.0 内核中 page cache 做了比较大的改进，文件可以通过 page cache 直接写，page cache 优先使用 high memory。

2.4.0 内核引入了新的对象：address_space，它包含用来读写一整页数据的方法。这些方法考虑到了 inode 的更新、page cache 处理和临时 buffer 的使用。page cache 和 buffer cache 的同步问题就消除了。

5. SMP 结构下的内存管理（需要进一步分析）

在 SMP 结构中，所有 cpu 都是“对称”的，一般物理上也是同一种 cpu。所有的 cpu 通过同一条总线共享同一个内存和所有的外设。

在 SMP 结构下内存管理与单 cpu 并没有太大的区别，因为是共享的一块物理内存，所以不管是物理地址还是进程的虚拟地址对于所有的 cpu 都是一致的。

不过在 slab 管理这一块略有差别，就是在 SMP 结构中分配 slab 对象时可以批量的分配（好像是为每个 cpu 都分配？）。

不过在 SMP 中高速缓存和内存的一致性问题则比较重要了。为了减少访问内存的冲突，SMP 结构中的每个 cpu 都有自己的高速缓存，所以这带来了内存和缓存之间一致性的问题，需要有 Linux 在解决，具体细节就不介绍。

在 SMP 中讨论的最多的还是进程的调度和中断机制，这些以后在讨论到相应的模块时再关注吧。

6. 相关问题

6.1 在地址变换中给出 cr3 里的页面目录寄存器的具体使用。

下面图是从 Intel 的技术文档中摘出来的 CR2 和 CR3 寄存器，可以看到 CR3 的低 12 位没有使用，而高 20 位是用于存放 PGD 的基址。

在 Linux 中，每个进程都有一个 PGD，所以进程切换就需要将 CR3 中高 20 位内容置为新进程的 PGD 基址。

相关代码是一条汇编代码为

asm volatile (“movl %0, %%cr3” : : “r” (__pa(next_pdg)));

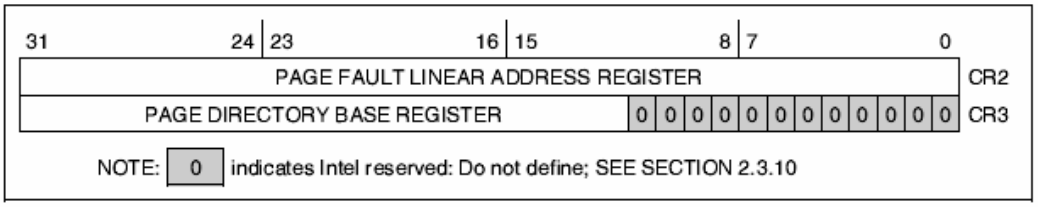
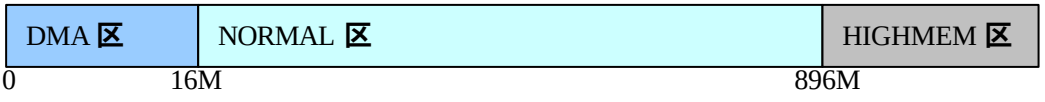


Figure 2-6. Control Registers 2 and 3

6.2 物理页面划分成几个管理区，划分各区大小的依据是什么？

前面已经介绍一个 pg_data_t 存储节点表示一块 UMA 内存节点（即访问速度一致），一个存储节点上的物理页面被分为三个管理区：DMA，NORMAL，HIGHMEM。如下图



DMA 区为 0～16M，因为 DMA 控制器地址线只有 24 位，所以只能访问 16M 空间；NORMAL 区为 16M～896M，因为内核虚拟空间为 1G，其中低 16M 已用于 DMA，而

系统又保留了 128M 作为交换空间，所以这样能访问到的上界为 1G－128M＝896M；

HIGHMEM 区为 896M 以上的内存，用户进程可以直接访问，系统进程要访问则需要通过一些手段。

6.3 mm_struct 中的 mmap_cache 是怎么使用维护的？

Linux 内核中有针对一些经常要访问的数据结构都设置了 cache，当然有的是链表，有的只有一项等等。前面已经介绍了一下比较重要的 cache，如 page cache，buffer cache 等。而像 mmap_cache 这样的 cache 维护很简单，仅仅根据存放了前次访问的虚拟区间指针。

6.4 有关内存管理的线程在初始化时的创建，如 kswapd 线程

在内核初始化时，需要把处理器的页面映射机制开，需要为物理内存建立内核所使用的 pgd，需要统一可用于分配的物理页面组织成区间管理方式等等。

当然也创建了一些线程用于维护内存管理的，主要是两个进程：kswapd 和 kreclaimd，它们的功能前面已经介绍。

6.5 页面调入调出是如何定位到磁盘呢？

有一个全局的交换设备（文件）数组，用户管理磁盘页面，其实可以把它们看作是磁盘上的文件。所以当要把一个页面交换到磁盘时，就变成了写文件；同样调入也变成了读文件。所以和文件操作建立的联系。