嵌入式操作系统

陈香兰

Fall 2009

嵌入式系统实验室

EMBEDDED SYSTEM LABORATORY

SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

内存管理

嵌入式系统实验室

EMBEDDED SYSTEM LABORATORY

SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC

内存管理

- 二级页表
- 动态存储器
- Slab 算法
- 非连续存储区

内存管理

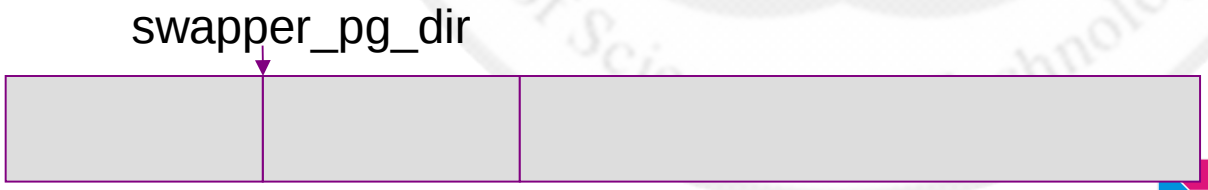
- RAM 的某些部分永久地分配给内核，用以存放内核代码以及静态数据
- RAM 的其余部分称为动态存储器（ dynamic memory ）

Arm 存储系统之粗粒度的 2 级页表

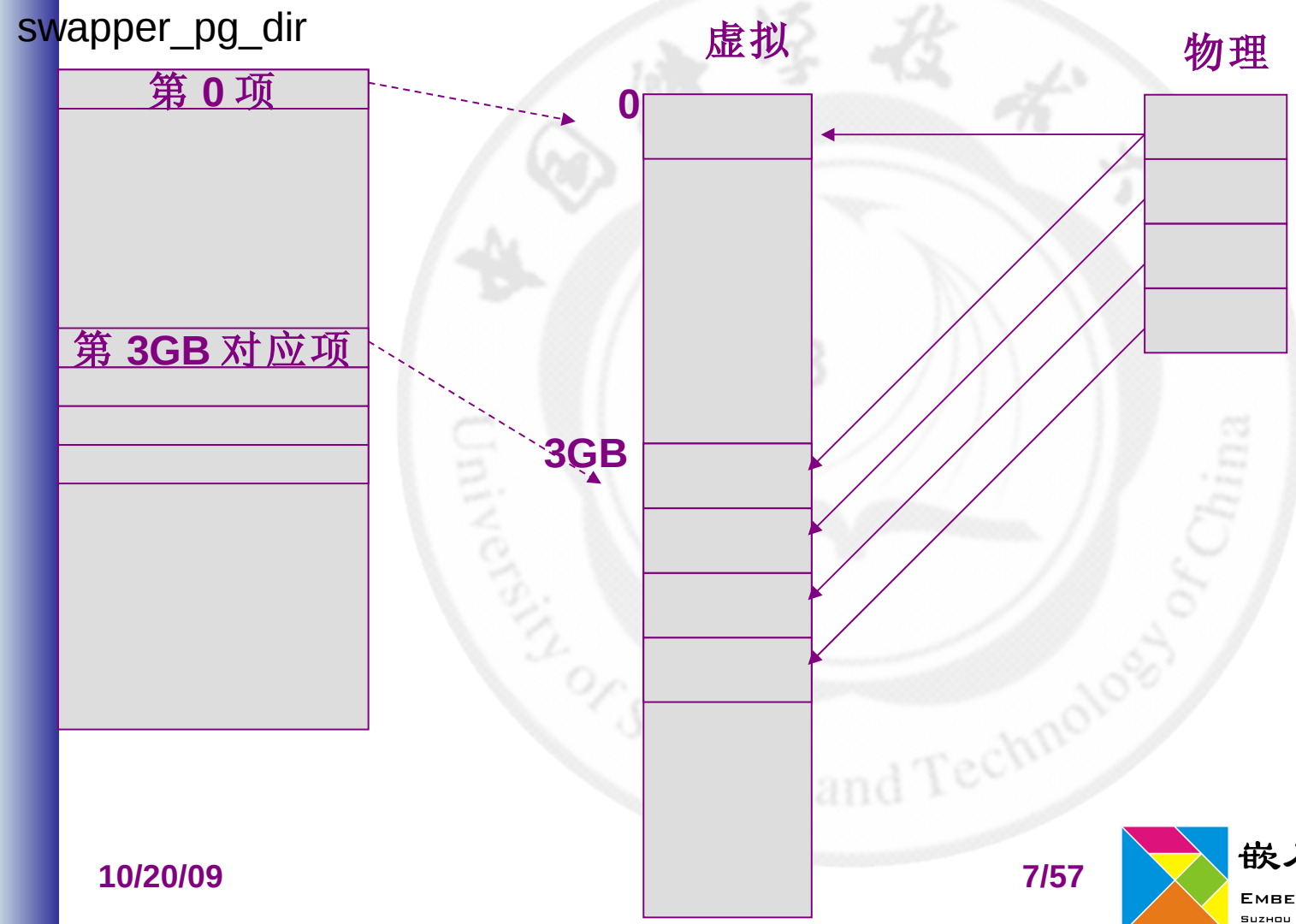
- 第一级页表:
  - 每一项描述 1MB 空间的映射关系
  - 每个条目 4B
  - 页表大小： 16KB
- 第二级页表:
  - 页框大小 4KB
  - 每个条目大小 4B
  - 页表大小： 1KB

Linux 中

- 虚拟地址空间:
  - KERNEL_RAM_VADDR :
    - 3GB 以上 + TEXT_OFFSET （大多为 0x8000 ）
- swapper_pg_dir
  - KERNEL_RAM_VADDR － 0x4000， 大小为 16KB
- Head.S 中： __create_page_tables



__create_page_tables



动态存储器

- 进程和内核都需要动态存储器
- 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

- 对于动态存储器要尽可能做到:
  - 按需分配，不需要时释放

主要内容

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

页框管理

- 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
  - 4KB 是大多数磁盘块大小的倍数
  - 传输效率高
  - 管理方便
- 例如： 512M 的物理内存对应于 128K 个页框
- 算法：伙伴算法

请求页框

- 内核实现了一种底层的内存分配机制，并提供了几个接口供其他内核函数调用。

- 分配：
  - alloc_pages/alloc_page
  - __get_free_pages/__get_free_page/__get_dma_pages/get_zeroed_page
- 释放
  - free_pages/__free_pages/free_page__free_page

页框数据结构

- 内核必须记录每个页框当前的状态
  - 哪些属于进程，哪些存放了内核代码/数据
  - 是否空闲，即是否可用
  - 如果不可用，内核需要知道是谁在用这个页框
  - 这个页框可能的使用者有用户态进程、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

- 页描述符： struct page
  - 每个物理页框都用一个页描述符表示
    - 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
      - 参见枚举类型 pageflags

- 当内核调用一个页框分配函数时，必须指明请求页框所在的区。这个一般是通过一些 flag 标志来指定的

GFP_XXX

关于 NUMA

- 不考虑

- 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

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

zone

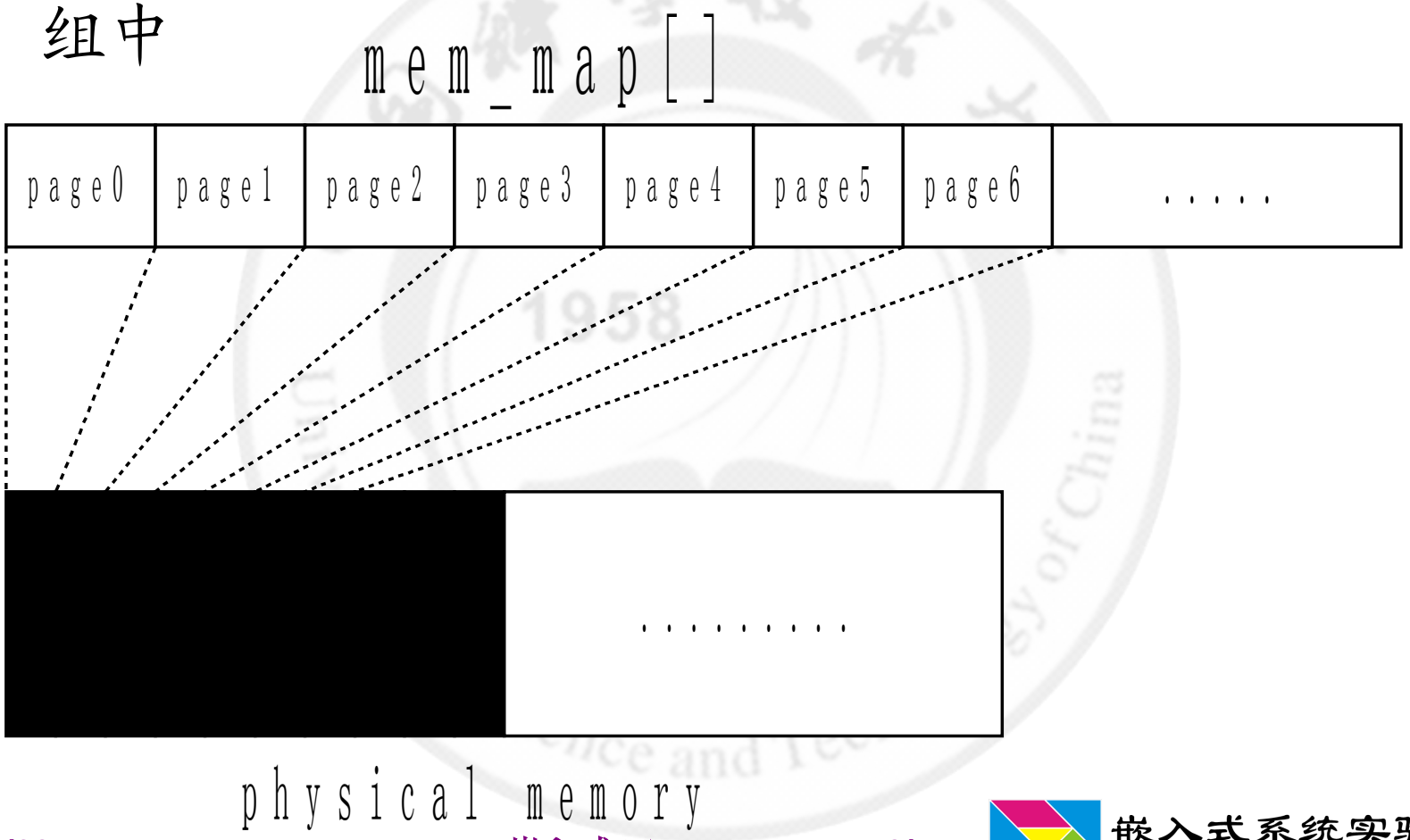
- 为了应付这种限制， Linux 把具有同样性质的物理内存划分成——区 (zones)
- Linux 把物理存储器划分为 4 个区
  - ZONE_DMA
  - ZONE_DMA32（未见用）
  - ZONE_NORMAL
  - ZONE_HIGHMEM
- 参见枚举类型 zone_type

- 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_HIGHMEN 区
  中包含的存储器页面不能由内核直接访问

- 每个 zone 使用 struct zone 表示
  - 关键: free_area

mem_map 数组

- 所有物理页框的描述符，组织在 mem_map 的数组中



- mem_map 的定义和初始化

arch/arm/mm/mmu.c

start_kernel→setup_arch→paging_init→bootmem_init
→bootmem_init_node→free_area_init_node
→alloc_node_mem_map

mm/page_alloc.c

- 页描述符将会占用很大的一段空间

- Mem_map 、 node 、 zone 之间的关系

请求页框

- 内核实现了一种底层的内存分配机制，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

关于 unsigned int gfp_mask

- 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

- 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 */
```

页框管理算法

- 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
- 这种策略要解决碎片问题：即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
  - 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
  - 使用一种适当的技术来记录现存的空闲连续页框的情况，以尽量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

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 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×b×4KB 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

数据结构

- 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:
  - 空闲内存管理数组 free_area

- mem_map 数组
  - 前面介绍过的页描述符数组
  - 每个页描述符描述一个物理页框
  - 整个 mem_map 数组描述整个 zone 中的所有的物理内存
- 空闲内存管理数组
  - 空闲内存按照伙伴管理的方法进行组织
  - 使用 free_area 结构

伙伴

- 当两个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

举例

- 假设有 128MB 的 ram 。

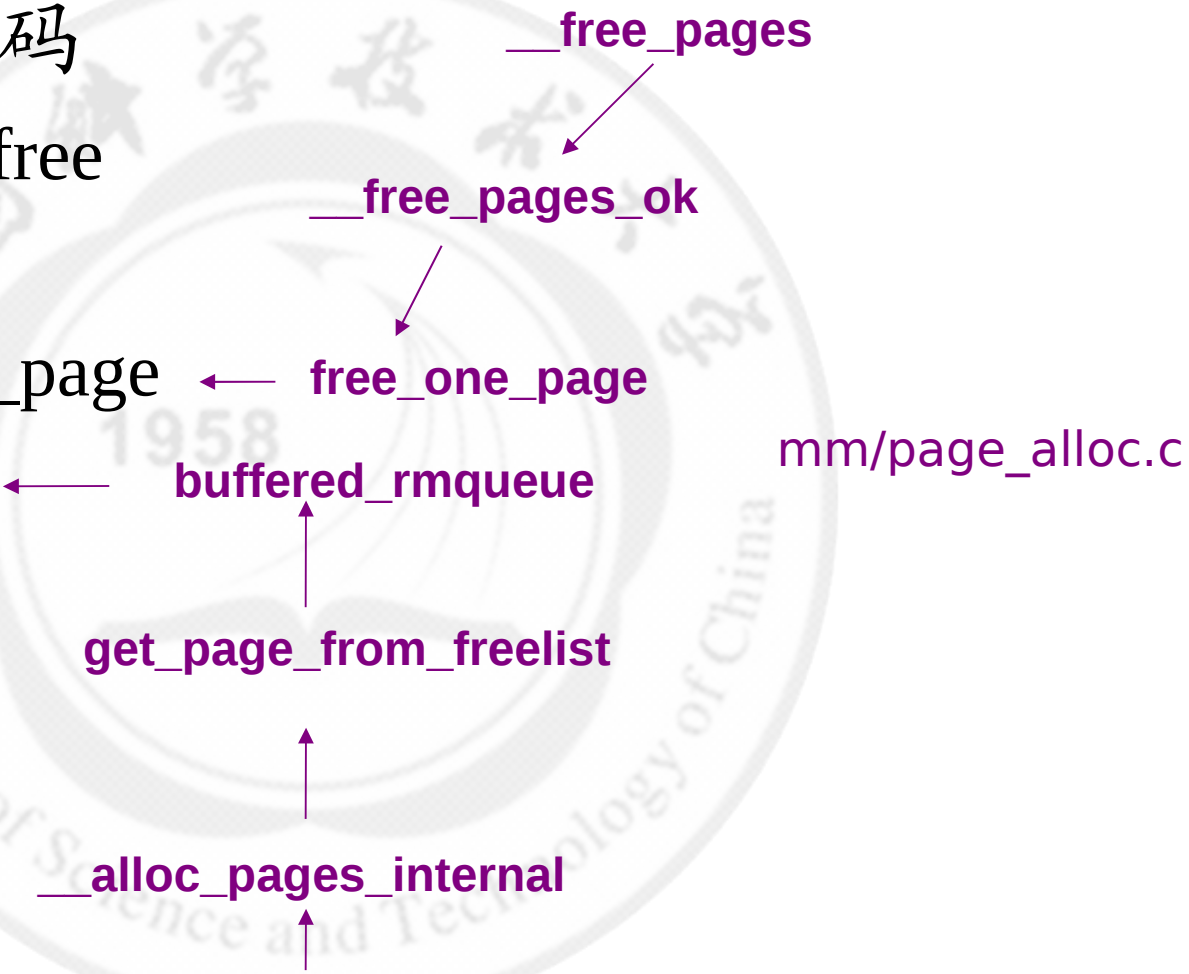
  128MB 最多可以分成 2^{15}=32768 个页框， 2^{14}=16384 个 8KB （ 2 页）的块或 2^{13}=8192 个 16KB （ 4 页）的块，直至 64 个大小为 512 个页的块

- 假设要请求一个大小为 128 个页框的块 (0.5MB) 。
  - 算法先 free_area[7] 中检查是否有空闲块（块大小为 128 个页框）
  - 若没有，就到 free_area[8] 中找一个空闲块（块大小为 256 个页框）
  - 若存在这样的块，内核就把 256 个页框分成两等份，一半用作满足请求，另一半插入 free_area[7] 中
  - 如果在 free_area[8] 中也没有空闲块，就继续找 free_area[9] 中是否有空闲块。

- 若有，先将 512 分成伙伴，一个插入 free_area[8] 中，另一个进一步划分成伙伴，取其一插入 free_area[7] 中，另一个分配出去
- 如果 free_area[9] 也没有空闲块，内存不够，返回一个错误信号

内存的分配与回收

- 阅读相关代码
  - 关键： nr_free
  - __free_one_page
  - __rmqueue



页框管理小结

- Mem_map
- Zone
- Free_area
- 伙伴算法

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 k 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

- 但这种分配方法有许多值得改进的地方：
  - 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通用缓存
  - 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
  - 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

- SLOB Allocator: Simple List Of Blocks
  - NUMA
- Slab
- Slub : slab 的一个变种

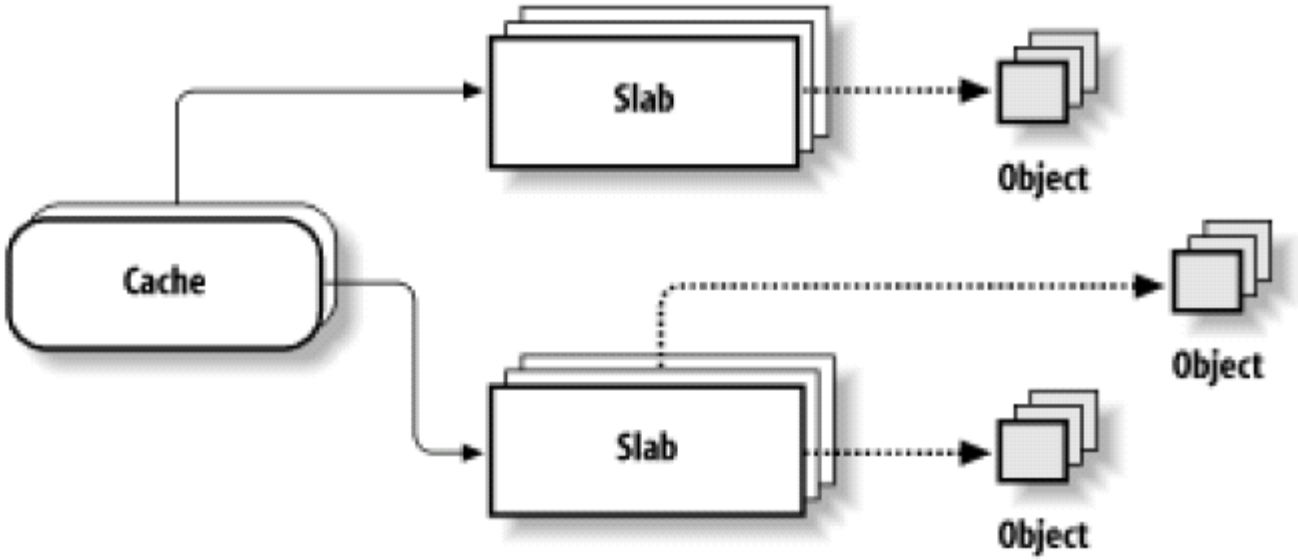
- Kmalloc/kfree
- 本课介绍基本的 slab 算法

slab 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

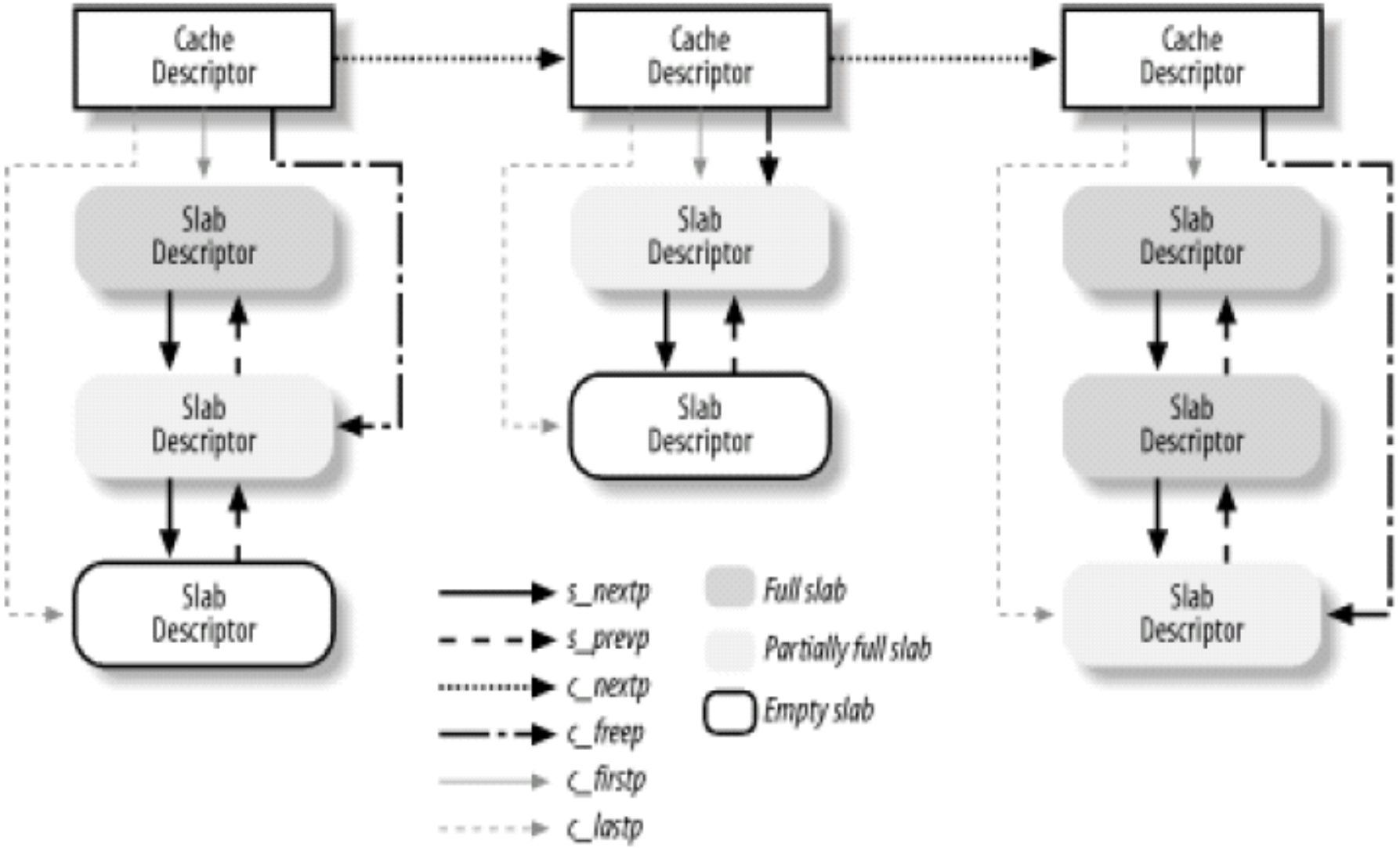
Figure 7-3. The slab allocator components



普通和专用高速缓存

- 每个高速缓存使用 kmem_cache_s 表示
- 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

高速缓存描述符和 slab 描述符之间的关系



- 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

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

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

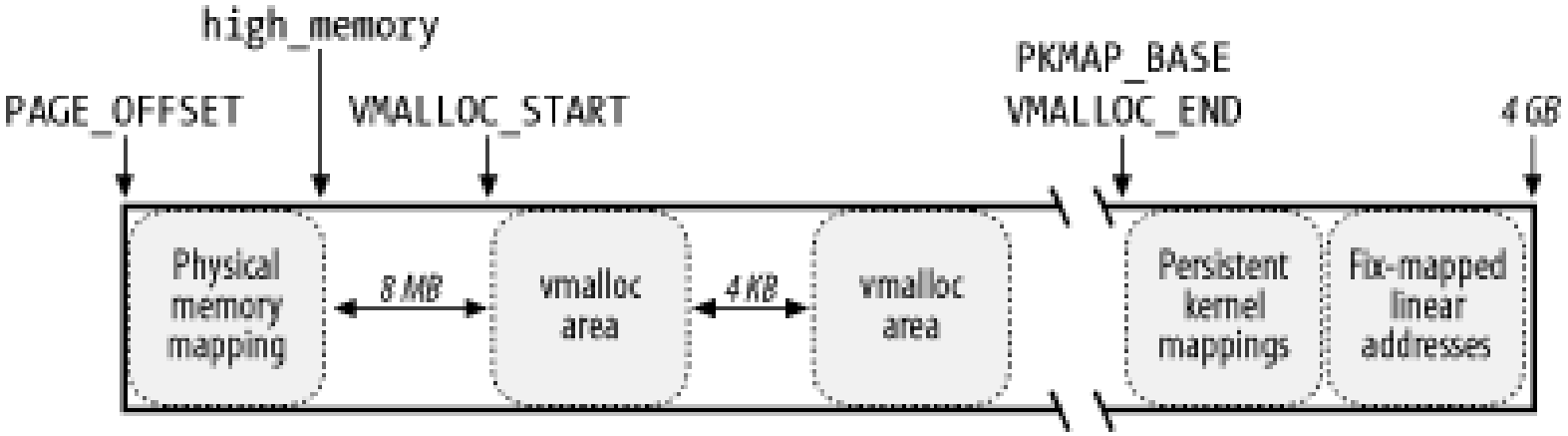
  这样可以大大减少内存的访问时间

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



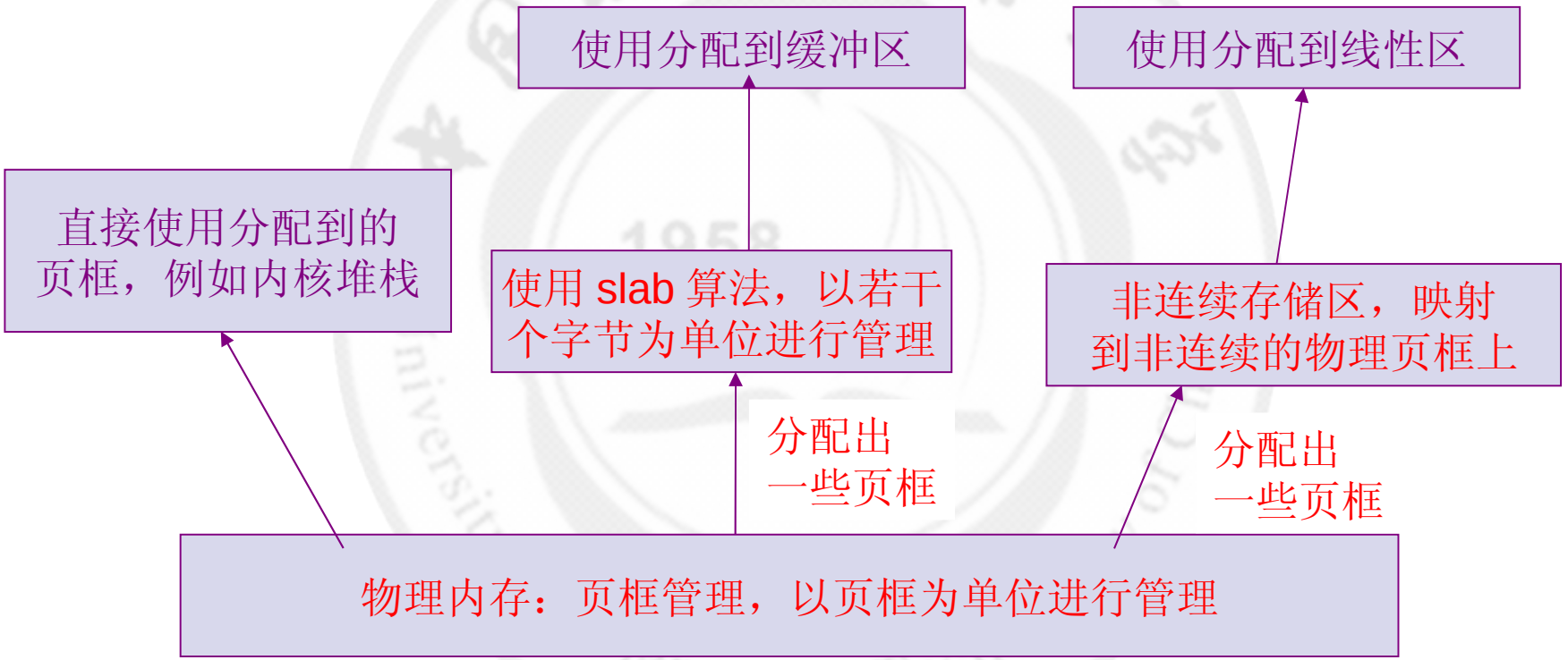
- 非连续存储区的描述符 vm_struct

include/linux/vmalloc.h

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

页框管理、内存区管理、非连续存储区管理之间的关系



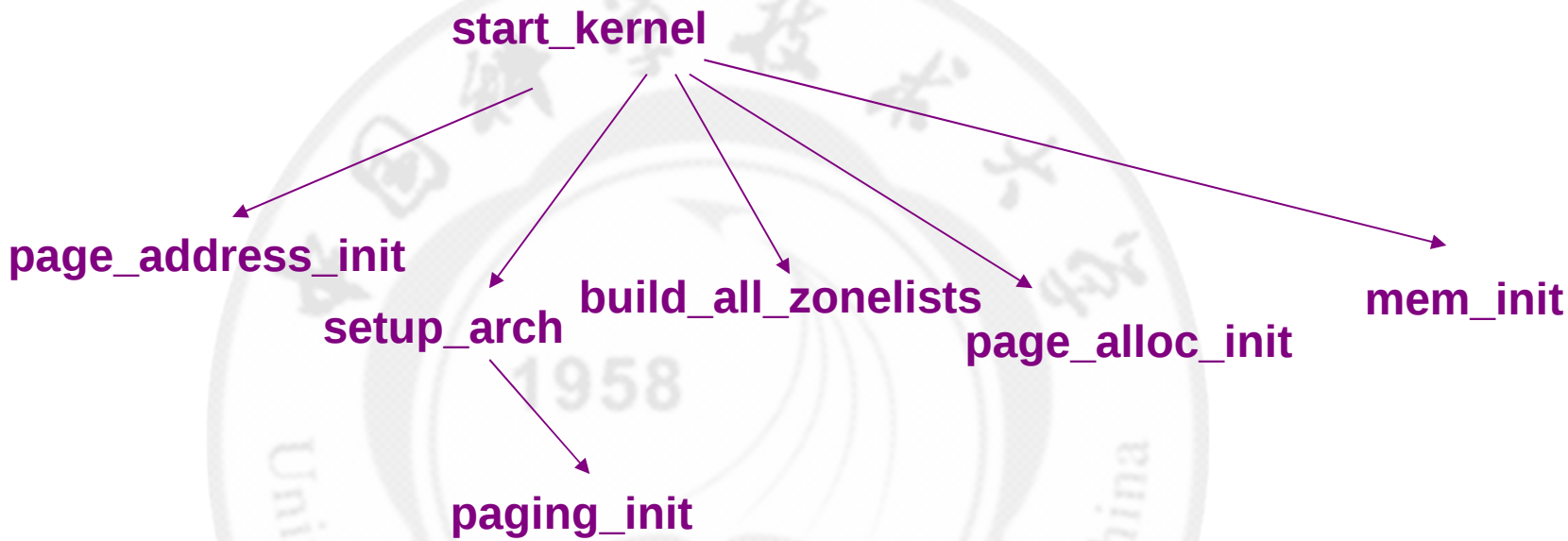
Thanks !

The end.

嵌入式系统实验室

EMBEDDED SYSTEM LABORATORY

SUZHOU INSTITUTE FOR ADVANCED STUDY OF USTC



page_address_init

- 在 include/linux/mm.h 文件中

```
00585: #if defined(CONFIG_HIGHMEM) && !defined(WANT_PAGE_VIRTUAL)
00586: #define HASHED_PAGE_VIRTUAL
00587: #endif

00589: #if defined(WANT_PAGE_VIRTUAL)
00590: #define page_address(page) ((page)->virtual)
00591: #define set_page_address(page, address)                  \
00592:         do {                                             \
00593:                 (page)->virtual = (address);             \
00594:         } while(0)
00595: #define page_address_init()  do { } while(0)
00596: #endif
00598: #if defined(HASHED_PAGE_VIRTUAL)
00599: void *page_address(struct page *page);
00600: void set_page_address(struct page *page, void *virtual);
00601: void page_address_init(void);
00602: #endif
```

```
00604: #if !defined(HASHED_PAGE_VIRTUAL) && !defined(WANT_PAGE_VIRTUAL)
00605: #define page_address(page) lowmem_page_address(page)
00606: #define set_page_address(page, address)  do { } while(0)
00607: #define page_address_init()  do { } while(0)
00608: #endif
```

我们不考虑高端内存

Paging_init

- Arm/mm/mmu.c

```
00747: /*
00748:  * paging_init() sets up the page tables, initialises the zone memory
00749:  * maps, and sets up the zero page, bad page and bad page tables.
00750:  */
00751: void __init paging_init(struct meminfo *mi, struct machine_desc *mdesc)
00752: {
00753:         void *zero_page;
00754:
00755:         build_mem_type_table();
00756:         prepare_page_table(mi);
00757:         bootmem_init(mi);
00758:         devicemaps_init(mdesc);
00759:
00760:         top_pmd = pmd_off_k(0xffff0000);
00761:
00762:         /*
00763:          * allocate the zero page.  Note that we count on this going ok.
00764:          */
00765:         zero_page = alloc_bootmem_low_pages(PAGE_SIZE);
00766:         memzero(zero_page, PAGE_SIZE);
00767:         empty_zero_page = virt_to_page(zero_page);
00768:         flush_dcache_page(empty_zero_page);
00769: }
```

Arm 的 cache

- Cachepolicy 数据结构

arch/arm/mm/mmu.c

```
00059: struct cachepolicy {
00060:         const char      policy[16];
00061:         unsigned int    cr_mask;
00062:         unsigned int    pmd;
00063:         unsigned int    pte;
00064: };
```

- Cachepolicy 变量

arch/arm/mm/mmu.c

```
00051: static unsigned int cachepolicy __initdata = CPOLICY_WRITEBACK;
```

- cache_policies[]

```
00045: #define CPOLICY_UNCACHED        0
00046: #define CPOLICY_BUFFERED        1
00047: #define CPOLICY_WRITETHROUGH    2
00048: #define CPOLICY_WRITEBACK       3
00049: #define CPOLICY_WRITEALLOC      4
```

cpu_architecture 获得 CPU 体系结构信息

- 根据 processor_id 来判断

```
00064: unsigned int processor_id;
00065: EXPORT_SYMBOL(processor_id);
```

  - processor_id 中间 5 个比特，掩码为 0x0008F000
- processor_id 在 head_common.S 中 switch_data 被 __mmap_switched 中被初始化

CPU 体系结构信息

include/asm-arm/system.h

```
00008: #define CPU_ARCH_UNKNOWN   0
00009: #define CPU_ARCH_ARMv3     1
00010: #define CPU_ARCH_ARMv4     2
00011: #define CPU_ARCH_ARMv4T    3
00012: #define CPU_ARCH_ARMv5     4
00013: #define CPU_ARCH_ARMv5T    5
00014: #define CPU_ARCH_ARMv5TE   6
00015: #define CPU_ARCH_ARMv5TEJ  7
00016: #define CPU_ARCH_ARMv6     8
00017: #define CPU_ARCH_ARMv7     9
```

若掩码对应的 bit 全 0

掩码末尾 3 位全 1，根据第 24 位是 0 还是 1

掩码最高位 0，根据掩码末 3 位

掩码最高位 1 及其后 3 位全 1，根据 mmfr0 判断