

23 | 物理内存管理（上）：会议室管理员如何分配会议室？

2019-05-20 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 19:09 大小 17.55M



前一节，我们讲了如何从项目经理的角度看内存，看到的是虚拟地址空间，这些虚拟的地址，总是要映射到物理的页面。这一节，我们来看，物理的页面是如何管理的。

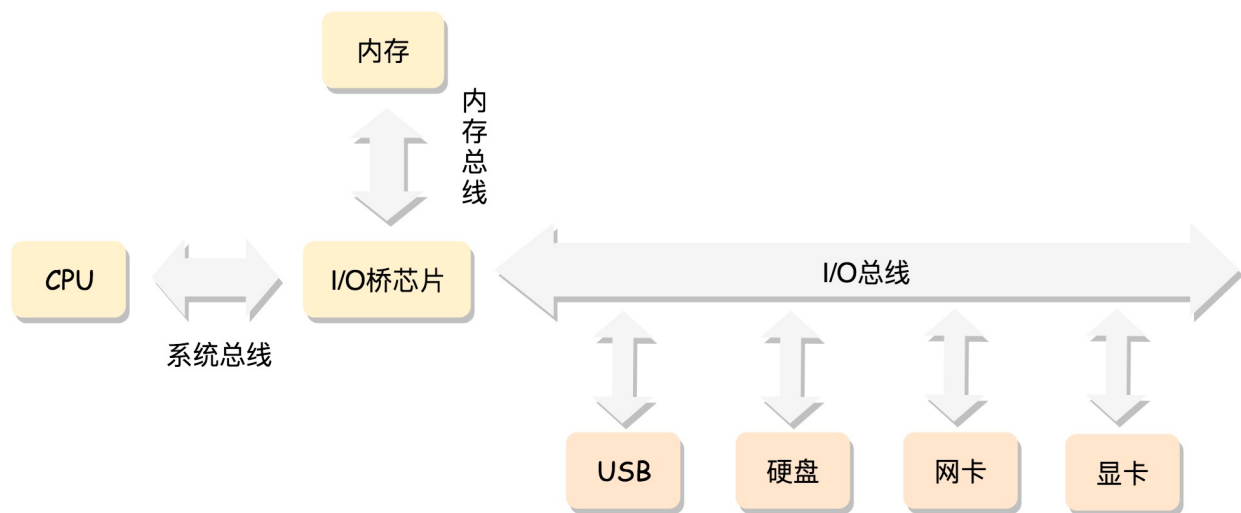
物理内存的组织方式

前面咱们讲虚拟内存，涉及物理内存的映射的时候，我们总是把内存想象成它是由连续的一页一页的块组成的。我们可以从 0 开始对物理页编号，这样每个物理页都会有个页号。

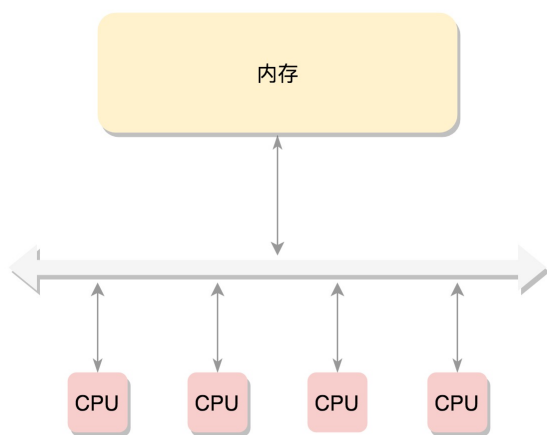
由于物理地址是连续的，页也是连续的，每个页大小也是一样的。因而对于任何一个地址，只要直接除一下每页的大小，很容易直接算出在哪一页。每个页有一个结构 `struct page` 表示，这个结构也是放在一个数组里面，这样根据页号，很容易通过下标找到相应的 `struct page` 结构。

如果是这样，整个物理内存的布局就非常简单、易管理，这就是最经典的**平坦内存模型** (Flat Memory Model) 。

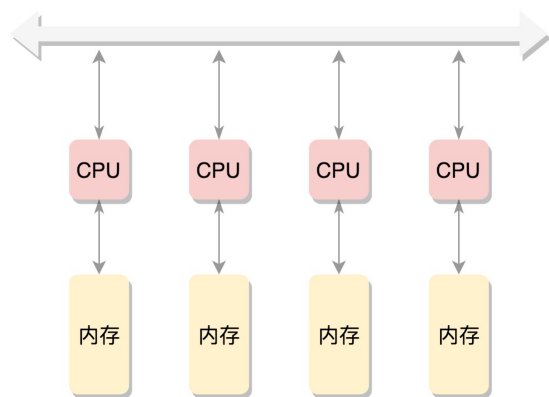
我们讲 x86 的工作模式的时候，讲过 CPU 是通过总线去访问内存的，这就是最经典的内存使用方式。



在这种模式下，CPU 也会有多个，在总线的一侧。所有的内存条组成一大片内存，在总线的另一侧，所有的 CPU 访问内存都要过总线，而且距离都是一样的，这种模式称为 **SMP** (Symmetric multiprocessing)，即对称多处理器。当然，它也有一个显著的缺点，就是总线会成为瓶颈，因为数据都要走它。



对称多处理器SMP



非统一内存访问NUMA

为了提高性能和可扩展性，后来有了一种更高级的模式，**NUMA** (Non-uniform memory access)，非一致内存访问。在这种模式下，内存不是一整块。每个 CPU 都有自己的本地内存，CPU 访问本地内存不用过总线，因而速度要快很多，每个 CPU 和内存在一起，称为一个 NUMA 节点。但是，在本地内存不足的情况下，每个 CPU 都可以去另外的 NUMA 节点申请内存，这个时候访问延时就会比较长。

这样，内存被分成了多个节点，每个节点再被分成一个一个的页面。由于页需要全局唯一定位，页还是需要有全局唯一的页号的。但是由于物理内存不是连起来的了，页号也就不再连续了。于是内存模型就变成了非连续内存模型，管理起来就复杂一些。

这里需要指出的是，NUMA 往往是非连续内存模型。而非连续内存模型不一定是 NUMA，有时候一大片内存的情况下，也会有物理内存地址不连续的情况。

后来内存技术牛了，可以支持热插拔了。这个时候，不连续成为常态，于是就有了稀疏内存模型。

节点

我们主要解析当前的主流场景，NUMA 方式。我们首先要能够表示 NUMA 节点的概念，于是有了下面这个结构 `typedef struct pglist_data pg_data_t`，它里面有以下的成员变量：

每一个节点都有自己的 ID: `node_id`;

`node_mem_map` 就是这个节点的 `struct page` 数组，用于描述这个节点里面的所有的页；

`node_start_pfn` 是这个节点的起始页号；

`node_spanned_pages` 是这个节点中包含不连续的物理内存地址的页面数；

`node_present_pages` 是真正可用的物理页面的数目。


例如，64M 物理内存隔着一个 4M 的空洞，然后是另外的 64M 物理内存。这样换算成页面数目就是，16K 个页面隔着 1K 个页面，然后是另外 16K 个页面。这种情况下，`node_spanned_pages` 就是 33K 个页面，`node_present_pages` 就是 32K 个页面。

```

2     struct zone node_zones[MAX_NR_ZONES];
3     struct zonelist node_zonelists[MAX_ZONELISTS];
4     int nr_zones;
5     struct page *node_mem_map;
6     unsigned long node_start_pfn;
7     unsigned long node_present_pages; /* total number of physical pages */
8     unsigned long node_spanned_pages; /* total size of physical page range, including
9     int node_id;
10    .....
11 } pg_data_t;

```

每一个节点分成一个个区域 zone，放在数组 node_zones 里面。这个数组的大小为 MAX_NR_ZONES。我们来看区域的定义。

 复制代码

```

1 enum zone_type {
2 #ifdef CONFIG_ZONE_DMA
3     ZONE_DMA,
4 #endif
5 #ifdef CONFIG_ZONE_DMA32
6     ZONE_DMA32,
7 #endif
8     ZONE_NORMAL,
9 #ifdef CONFIG_HIGHMEM
10    ZONE_HIGHMEM,
11 #endif
12    ZONE_MOVABLE,
13    __MAX_NR_ZONES
14 };

```

ZONE_DMA 是指可用于作 DMA（Direct Memory Access，直接内存存取）的内存。DMA 是这样一种机制：要把外设的数据读入内存或把内存的数据传送到外设，原来都要通过 CPU 控制完成，但是这会占用 CPU，影响 CPU 处理其他事情，所以有了 DMA 模式。CPU 只需向 DMA 控制器下达指令，让 DMA 控制器来处理数据的传送，数据传送完毕再把信息反馈给 CPU，这样就可以解放 CPU。

对于 64 位系统，有两个 DMA 区域。除了上面说的 ZONE_DMA，还有 ZONE_DMA32。在这里你大概理解 DMA 的原理就可以，不必纠结，我们后面会讲 DMA 的机制。

ZONE_NORMAL 是直接映射区，就是上一节讲的，从物理内存到虚拟内存的内核区域，通过加上一个常量直接映射。

ZONE_HIGHMEM 是高端内存区，就是上一节讲的，对于 32 位系统来说超过 896M 的地方，对于 64 位没必要有的一段区域。

ZONE_MOVABLE 是可移动区域，通过将物理内存划分为可移动分配区域和不可移动分配区域来避免内存碎片。

这里你需要注意一下，我们刚才对于区域的划分，都是针对物理内存的。

nr_zones 表示当前节点的区域的数量。node_zonelists 是备用节点和它的内存区域的情况。前面讲 NUMA 的时候，我们讲了 CPU 访问内存，本节点速度最快，但是如果本节点内存不够怎么办，还是需要去其他节点进行分配。毕竟，就算在备用节点里面选择，慢了点也比没有强。

既然整个内存被分成了多个节点，那 pglist_data 应该放在一个数组里面。每个节点一项，就像下面代码里面一样：

 复制代码

```
1 struct pglist_data *node_data[MAX_NUMNODES] __read_mostly;
```

区域

到这里，我们把内存分成了节点，把节点分成了区域。接下来我们来看，一个区域里面是如何组织的。

表示区域的数据结构 zone 的定义如下：

 复制代码

```
1 struct zone {
2     .....
3     struct pglist_data      *zone_pgdat;
4     struct per_cpu_pageset __percpu *pageset;
5
6 }
```

```

7         unsigned long            zone_start_pfn;
8
9
10        /*
11         * spanned_pages is the total pages spanned by the zone, including
12         * holes, which is calculated as:
13         *     spanned_pages = zone_end_pfn - zone_start_pfn;
14         *
15         * present_pages is physical pages existing within the zone, which
16         * is calculated as:
17         *     present_pages = spanned_pages - absent_pages(pages in holes);
18         *
19         * managed_pages is present pages managed by the buddy system, which
20         * is calculated as (reserved_pages includes pages allocated by the
21         * bootmem allocator):
22         *     managed_pages = present_pages - reserved_pages;
23         *
24         */
25        unsigned long            managed_pages;
26        unsigned long            spanned_pages;
27        unsigned long            present_pages;
28
29
30        const char                *name;
31        .....
32        /* free areas of different sizes */
33        struct free_area        free_area[MAX_ORDER];
34
35
36        /* zone flags, see below */
37        unsigned long            flags;
38
39
40        /* Primarily protects free_area */
41        spinlock_t                lock;
42        .....
43 } ____cacheline_internodealigned_in_

```

在一个 zone 里面，zone_start_pfn 表示属于这个 zone 的第一个页。

如果我们仔细看代码的注释，可以看到，spanned_pages = zone_end_pfn - zone_start_pfn，也即 spanned_pages 指的是不管中间有没有物理内存空洞，反正就是最后的页号减去起始的页号。

$\text{present_pages} = \text{spanned_pages} - \text{absent_pages}(\text{pages in holes})$, 也即 present_pages 是这个 zone 在物理内存中真实存在的所有 page 数目。

$\text{managed_pages} = \text{present_pages} - \text{reserved_pages}$, 也即 managed_pages 是这个 zone 被伙伴系统管理的所有的 page 数目, 伙伴系统的工作机制我们后面会讲。

per_cpu_pageset 用于区分冷热页。什么叫冷热页呢? 咱们讲 x86 体系结构的时候讲过, 为了让 CPU 快速访问段描述符, 在 CPU 里面有段描述符缓存。CPU 访问这个缓存的速度比内存快得多。同样对于页面来讲, 也是这样的。如果一个页被加载到 CPU 高速缓存里面, 这就是一个热页 (Hot Page), CPU 读起来速度会快很多, 如果没有就是冷页 (Cold Page)。由于每个 CPU 都有自己的高速缓存, 因而 per_cpu_pageset 也是每个 CPU 一个。

页

了解了区域 zone, 接下来我们就到了组成物理内存的基本单位, 页的数据结构 struct page 。这是一个特别复杂的结构, 里面有很多的 union, union 结构是在 C 语言中被用于同一块内存根据情况保存不同类型数据的一种方式。这里之所以用了 union, 是因为一个物理页面使用模式有多种。

第一种模式, 要用就用一整页。这一整页的内存, 或者直接和虚拟地址空间建立映射关系, 我们把这种称为匿名页 (Anonymous Page)。或者用于关联一个文件, 然后再和虚拟地址空间建立映射关系, 这样的文件, 我们称为内存映射文件 (Memory-mapped File)。

如果某一页是这种使用模式, 则会使用 union 中的以下变量:

$\text{struct address_space} * \text{mapping}$ 就是用于内存映射, 如果是匿名页, 最低位为 1; 如果是映射文件, 最低位为 0;

pgoff_t index 是在映射区的偏移量;

atomic_t_mapcount , 每个进程都有自己的页表, 这里指有多少个页表项指向了这个页;

$\text{struct list_head lru}$ 表示这一页应该在一个链表上, 例如这个页面被换出, 就在换出页的链表中;

compound 相关的变量用于复合页（Compound Page），就是将物理上连续的两个或多个页看成一个独立的大页。

第二种模式，仅需分配小块内存。有时候，我们不需要一下子分配这么多的内存，例如分配一个 task_struct 结构，只需要分配小块的内存，去存储这个进程描述结构的对象。为了满足对这种小内存块的需要，Linux 系统采用了一种被称为**slab allocator**的技术，用于分配称为 slab 的一小块内存。它的基本原理是从内存管理模块申请一整块页，然后划分成多个小块的存储池，用复杂的队列来维护这些小块的状态（状态包括：被分配了 / 被放回池子 / 应该被回收）。

也正是因为 slab allocator 对于队列的维护过于复杂，后来就有了一种不使用队列的分配器 slub allocator，后面我们会解析这个分配器。但是你会发现，它里面还是用了很多 slab 的字眼，因为它保留了 slab 的用户接口，可以看成 slab allocator 的另一种实现。

还有一种小块内存的分配器称为**slob**，非常简单，主要使用在小型的嵌入式系统。

如果某一页是用于分割成一小块一小块的内存进行分配的使用模式，则会使用 union 中的以下变量：

s_mem 是已经分配了正在使用的 slab 的第一个对象；

freelist 是池子中的空闲对象；

rcu_head 是需要释放的列表。

 复制代码

```
1  struct page {
2      unsigned long flags;
3      union {
4          struct address_space *mapping;
5          void *s_mem;                /* slab first object */
6          atomic_t compound_mapcount; /* first tail page */
7      };
8      union {
9          pgoff_t index;              /* Our offset within mapping. */
10         void *freelist;              /* sl[aou]b first free object */
11     };
12     union {
13         unsigned counters;
14         struct {
15             union {
```



```

16         atomic_t _mapcount;
17         unsigned int active;           /* SLAB */
18         struct {                       /* SLUB */
19             unsigned inuse:16;
20             unsigned objects:15;
21             unsigned frozen:1;
22         };
23         int units;                     /* SLOB */
24     };
25     atomic_t _refcount;
26 };
27 };
28 union {
29     struct list_head lru; /* Pageout list */
30     struct dev_pagemap *pgmap;
31     struct {               /* slub per cpu partial pages */
32         struct page *next; /* Next partial slab */
33         int pages;         /* Nr of partial slabs left */
34         int pobjects;      /* Approximate # of objects */
35     };
36     struct rcu_head rcu_head;
37     struct {
38         unsigned long compound_head; /* If bit zero is set */
39         unsigned int compound_dtor;
40         unsigned int compound_order;
41     };
42 };
43 union {
44     unsigned long private;
45     struct kmem_cache *slab_cache; /* SL[AU]B: Pointer to slab */
46 };
47 .....
48 }

```

页的分配

好了，前面我们讲了物理内存的组织，从节点到区域到页到小块。接下来，我们来看物理内存的分配。


对于要分配比较大的内存，例如到分配页级别的，可以使用**伙伴系统**（Buddy System）。

Linux 中的内存管理的“页”大小为 4KB。把所有的空闲页分组为 11 个页块链表，每个块链表分别包含很多个大小的页块，有 1、2、4、8、16、32、64、128、256、512 和

1024 个连续页的页块。最大可以申请 1024 个连续页，对应 4MB 大小的连续内存。每个页块的第一个页的物理地址是该页块大小的整数倍。


第 i 个页块链表中，页块中页的数目为 2^i 。

在 struct zone 里面有以下定义：

 复制代码

```
1 struct free_area      free_area[MAX_ORDER];
```

MAX_ORDER 就是指数。


 复制代码

```
1 #define MAX_ORDER 11
```

当向内核请求分配 $(2^{(i-1)}, 2^i]$ 数目的页块时，按照 2^i 页块请求处理。如果对应的页块链表中没有空闲页块，那我们就在更大的页块链表中去寻找。当分配的页块中有多余的页时，伙伴系统会根据多余的页块大小插入到对应的空闲页块链表中。

例如，要请求一个 128 个页的页块时，先检查 128 个页的页块链表是否有空闲块。如果没有，则查 256 个页的页块链表；如果有空闲块的话，则将 256 个页的页块分成两份，一份使用，一份插入 128 个页的页块链表中。如果还是没有，就查 512 个页的页块链表；如果有的话，就分裂为 128、128、256 三个页块，一个 128 的使用，剩余两个插入对应页块链表。

上面这个过程，我们可以在分配页的函数 alloc_pages 中看到。

 复制代码

```
1 static inline struct page *
2 alloc_pages(gfp_t gfp_mask, unsigned int order)
3 {
4     return alloc_pages_current(gfp_mask, order);
5 }
```

```

6
7
8 /**
9  *      alloc_pages_current - Allocate pages.
10 *
11 *      @gfp:
12 *          %GFP_USER    user allocation,
13 *          %GFP_KERNEL  kernel allocation,
14 *          %GFP_HIGHMEM highmem allocation,
15 *          %GFP_FS      don't call back into a file system.
16 *          %GFP_ATOMIC don't sleep.
17 *      @order: Power of two of allocation size in pages. 0 is a single page.
18 *
19 *      Allocate a page from the kernel page pool. When not in
20 *      interrupt context and apply the current process NUMA policy.
21 *      Returns NULL when no page can be allocated.
22 */
23 struct page *alloc_pages_current(gfp_t gfp, unsigned order)
24 {
25     struct mempolicy *pol = &default_policy;
26     struct page *page;
27     .....
28     page = __alloc_pages_nodemask(gfp, order,
29                                   policy_node(gfp, pol, numa_node_id()),
30                                   policy_nodemask(gfp, pol));
31     .....
32     return page;
33 }

```

alloc_pages 会调用 alloc_pages_current，这里的注释比较容易看懂了，gfp 表示希望在那个区域中分配这个内存：

GFP_USER 用于分配一个页映射到用户进程的虚拟地址空间，并且希望直接被内核或者硬件访问，主要用于一个用户进程希望通过内存映射的方式，访问某些硬件的缓存，例如显卡缓存；


GFP_KERNEL 用于内核中分配页，主要分配 ZONE_NORMAL 区域，也即直接映射区；

GFP_HIGHMEM，顾名思义就是主要分配高端区域的内存。

另一个参数 order，就是表示分配 2 的 order 次方个页。

接下来调用 __alloc_pages_nodemask。这是伙伴系统的核心方法。它会调用 get_page_from_freelist。这里的逻辑也很容易理解，就是在一个循环中先看当前节点的


zone。如果找不到空闲页，则再看备用节点的 zone。

 复制代码

```
1 static struct page *
2 get_page_from_freelist(gfp_t gfp_mask, unsigned int order, int alloc_flags,
3                         const struct alloc_context *ac)
4 {
5     .....
6     for_next_zone_zonelist_nodemask(zone, z, ac->zonelist, ac->high_zoneidx, ac->no
7         struct page *page;
8     .....
9         page = rmqueue(ac->preferred_zoneref->zone, zone, order,
10                        gfp_mask, alloc_flags, ac->migratetype);
11     .....
12 }
```

每一个 zone，都有伙伴系统维护的各种大小的队列，就像上面伙伴系统原理里讲的那样。这里调用 rmqueue 就很好理解了，就是找到合适大小的那个队列，把页面取下来。

接下来的调用链是 rmqueue->__rmqueue->__rmqueue_smallest。在这里，我们能清楚看到伙伴系统的逻辑。

 复制代码


```
1 static inline
2 struct page *__rmqueue_smallest(struct zone *zone, unsigned int order,
3                                 int migratetype)
4 {
5     unsigned int current_order;
6     struct free_area *area;
7     struct page *page;
8
9
10    /* Find a page of the appropriate size in the preferred list */
11    for (current_order = order; current_order < MAX_ORDER; ++current_order) {
12        area = &(zone->free_area[current_order]);
13        page = list_first_entry_or_null(&area->free_list[migratetype],
14                                        struct page, lru);
15        if (!page)
16            continue;
17        list_del(&page->lru);
18        rmv_page_order(page);
19        area->nr_free--;
20        expand(zone, page, order, current_order, area, migratetype);
21        set_pcppage_migratetype(page, migratetype);
```

```

22         return page;
23     }
24
25
26     return NULL;

```

从当前的 order，也即指数开始，在伙伴系统的 free_area 找 2^{order} 大小的页块。如果链表的第一个不为空，就找到了；如果为空，就到更大的 order 的页块链表里面去找。找到以后，除了将页块从链表中取下来，我们还要把多余的部分放到其他页块链表里面。expand 就是干这个事情的。area 就是伙伴系统那个表里面的前一项，前一项里面的页块大小是当前项的页块大小除以 2，size 右移一位也就是除以 2，list_add 就是加到链表上，nr_free++ 就是计数加 1。

 复制代码

```

1 static inline void expand(struct zone *zone, struct page *page,
2     int low, int high, struct free_area *area,
3     int migratetype)
4 {
5     unsigned long size = 1 << high;
6
7
8     while (high > low) {
9         area--;
10        high--;
11        size >>= 1;
12        .....
13        list_add(&page[size].lru, &area->free_list[migratetype]);
14        area->nr_free++;
15        set_page_order(&page[size], high);
16    }
17 }

```

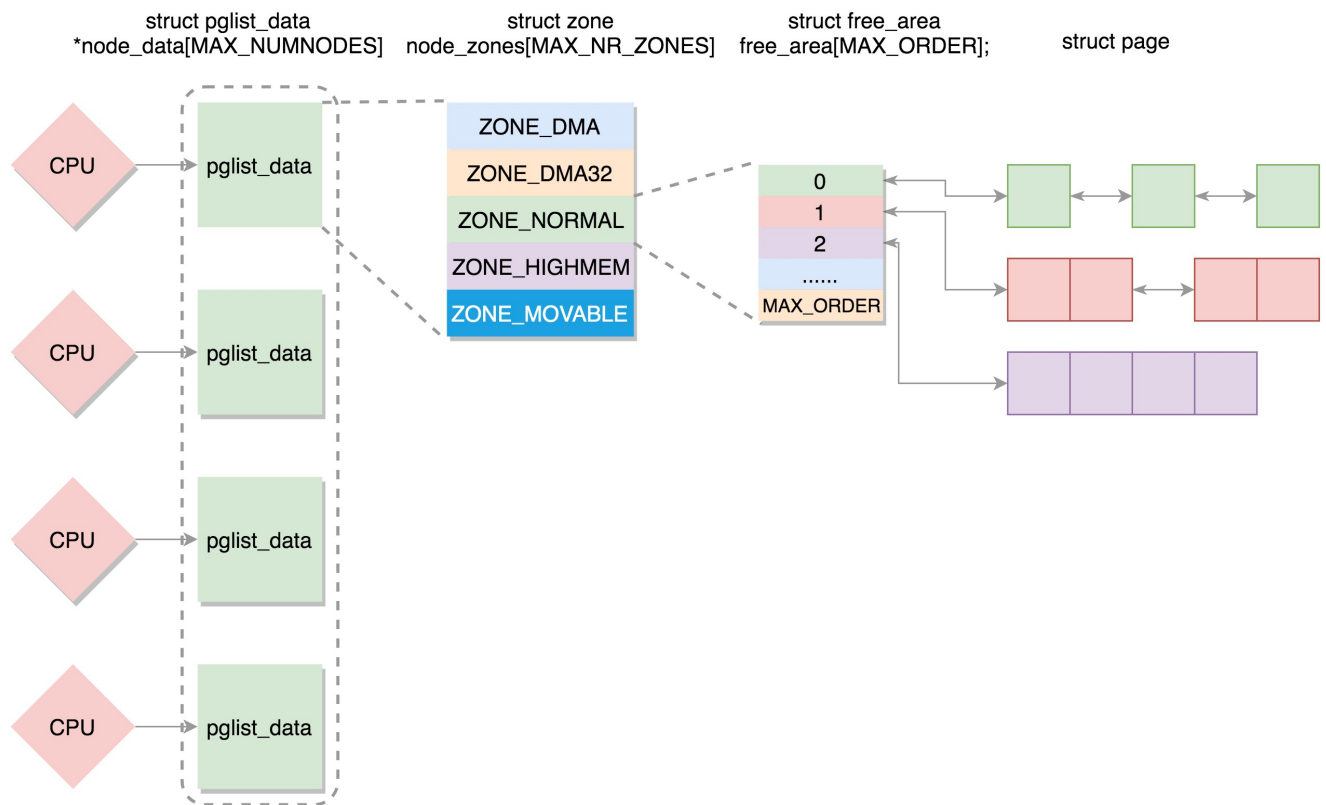
总结时刻

对于物理内存的管理的讲解，到这里要告一段落了。这一节我们主要讲了物理内存的组织形式，就像下面图中展示的一样。

如果有多个 CPU，那就有多个节点。每个节点用 struct pglist_data 表示，放在一个数组里面。

每个节点分为多个区域，每个区域用 struct zone 表示，也放在一个数组里面。

每个区域分为多个页。为了方便分配，空闲页放在 struct free_area 里面，使用伙伴系统进行管理和分配，每一页用 struct page 表示。



课堂练习

伙伴系统是一种非常精妙的实现方式，无论你使用什么语言，请自己实现一个这样的分配系统，说不定哪天你在做某个系统的时候，就用到了。

欢迎留言和我分享你的疑惑和见解，也欢迎你收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习、进步。

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院

云计算技术部首席架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 22 | 进程空间管理：项目组还可以自行布置会议室

下一篇 24 | 物理内存管理（下）：会议室管理员如何分配会议室？

精选留言 (11)

写留言



why

2019-05-23

3

- 物理内存组织方式

- 每个物理页由 struct page 表示
- 物理页连续, page 放入一个数组中, 称为平坦内存模型
- 多个 CPU 通过总线访问内存, 称为 SMP 对称多处理器(采用平坦内存模型, 总线成为瓶颈)...

展开



有铭

2019-05-20

2

还是没理解那个“伙伴系统”为何会命名为伙伴系统，没感觉到有“伙伴”的感觉



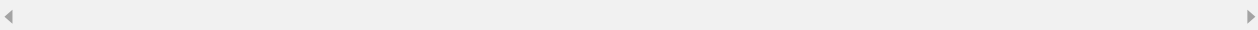
Virtue

2019-05-28



学了进程管理和内存管理，在看前面内核初始化的内容，感觉理解又深刻了点。

作者回复: 赞，常复习常新



CHEN

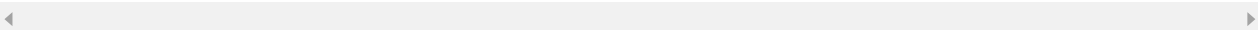
2019-05-25



周六打卡，这周的课程让我感觉自己智商欠费😓

展开 ∨

作者回复: 一个是CPU，一个是内存，比较绕，多看几遍就好了



Sharry

2019-05-23



老师, 从这节课我看到 slab slob 和 slub... 这...是同一个吗?

展开 ∨

作者回复: 不是同一个



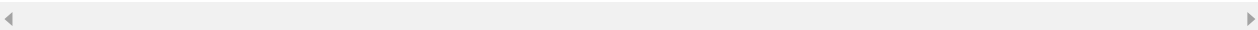
冷月流光

2019-05-22



老师好，伙伴系统的页分配我好像理解了，但是它是怎么回收(回收后要合并吗?)还是没想明白，这个伙伴系统与netty的PoolChunk分配机制有什么异同?

作者回复: 会合并的





Leon 📷

2019-05-22



伙伴系统的意思就是劫富济贫

展开 ▾



bradleyz...

2019-05-21



感觉伙伴系统的命名是说，每次请求内存时，要么是请求到别的伙伴分剩下的内存，要么是自己切好拿走一块，剩下的给将来的伙伴。



Linuxer

2019-05-21



有些应用会要求关闭numa那么这里numanode就为1吗？这时候是所有CPU共用一个pglist？



Linuxer

2019-05-21



每个页块的第一个页的物理地址是该页块大小的整数倍。怎么理解

展开 ▾



Brigand

2019-05-20



Linux可以理解为每页能存4KB吗？

展开 ▾