

24 | 物理内存管理（下）：会议室管理员如何分配会议室？

2019-05-22 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 13:43 大小 12.57M




前一节，前面我们解析了整页的分配机制。如果遇到小的对象，物理内存是如何分配的呢？这一节，我们一起来看一看。

小内存的分配

前面我们讲过，如果遇到小的对象，会使用 slub 分配器进行分配。那我们就先来解析它的工作原理。

还记得咱们创建进程的时候，会调用 `dup_task_struct`，它想要试图复制一个 `task_struct` 对象，需要先调用 `alloc_task_struct_node`，分配一个 `task_struct` 对象。

从这段代码可以看出，它调用了 `kmem_cache_alloc_node` 函数，在 `task_struct` 的缓存区域 `task_struct_cachep` 分配了一块内存。

 复制代码


```
1 static struct kmem_cache *task_struct_cachep;
2
3 task_struct_cachep = kmem_cache_create("task_struct",
4                                     arch_task_struct_size, align,
5                                     SLAB_PANIC|SLAB_NOTRACK|SLAB_ACCOUNT, NULL);
6
7 static inline struct task_struct *alloc_task_struct_node(int node)
8 {
9     return kmem_cache_alloc_node(task_struct_cachep, GFP_KERNEL, node);
10 }
11
12 static inline void free_task_struct(struct task_struct *tsk)
13 {
14     kmem_cache_free(task_struct_cachep, tsk);
15 }
```

在系统初始化的时候，`task_struct_cachep` 会被 `kmem_cache_create` 函数创建。这个函数也比较容易看懂，专门用于分配 `task_struct` 对象的缓存。这个缓存区的名字就叫 `task_struct`。缓存区中每一块的大小正好等于 `task_struct` 的大小，也即 `arch_task_struct_size`。

有了这个缓存区，每次创建 `task_struct` 的时候，我们不用到内存里面去分配，先在缓存里面看看有没有直接可用的，这就是 **`kmem_cache_alloc_node`** 的作用。

当一个进程结束，`task_struct` 也不用直接被销毁，而是放回到缓存中，这就是 **`kmem_cache_free`** 的作用。这样，新进程创建的时候，我们就可以直接用现成的缓存中的 `task_struct` 了。

我们来仔细看看，缓存区 `struct kmem_cache` 到底是什么样子。

 复制代码

```
1 struct kmem_cache {
2     struct kmem_cache_cpu __percpu *cpu_slab;
3     /* Used for retriving partial slabs etc */
4     unsigned long flags;
5     unsigned long min_partial;
```

```

6      int size;                /* The size of an object including meta data */
7      int object_size;         /* The size of an object without meta data */
8      int offset;              /* Free pointer offset. */
9  #ifdef CONFIG_SLUB_CPU_PARTIAL
10     int cpu_partial;          /* Number of per cpu partial objects to keep around */
11 #endif
12     struct kmem_cache_order_objects oo;
13     /* Allocation and freeing of slabs */
14     struct kmem_cache_order_objects max;
15     struct kmem_cache_order_objects min;
16     gfp_t allocflags;          /* gfp flags to use on each alloc */
17     int refcount;              /* Refcount for slab cache destroy */
18     void (*ctor)(void *);
19     .....
20     const char *name;          /* Name (only for display!) */
21     struct list_head list;     /* List of slab caches */
22     .....
23     struct kmem_cache_node *node[MAX_NUMNODES];
24 };
25

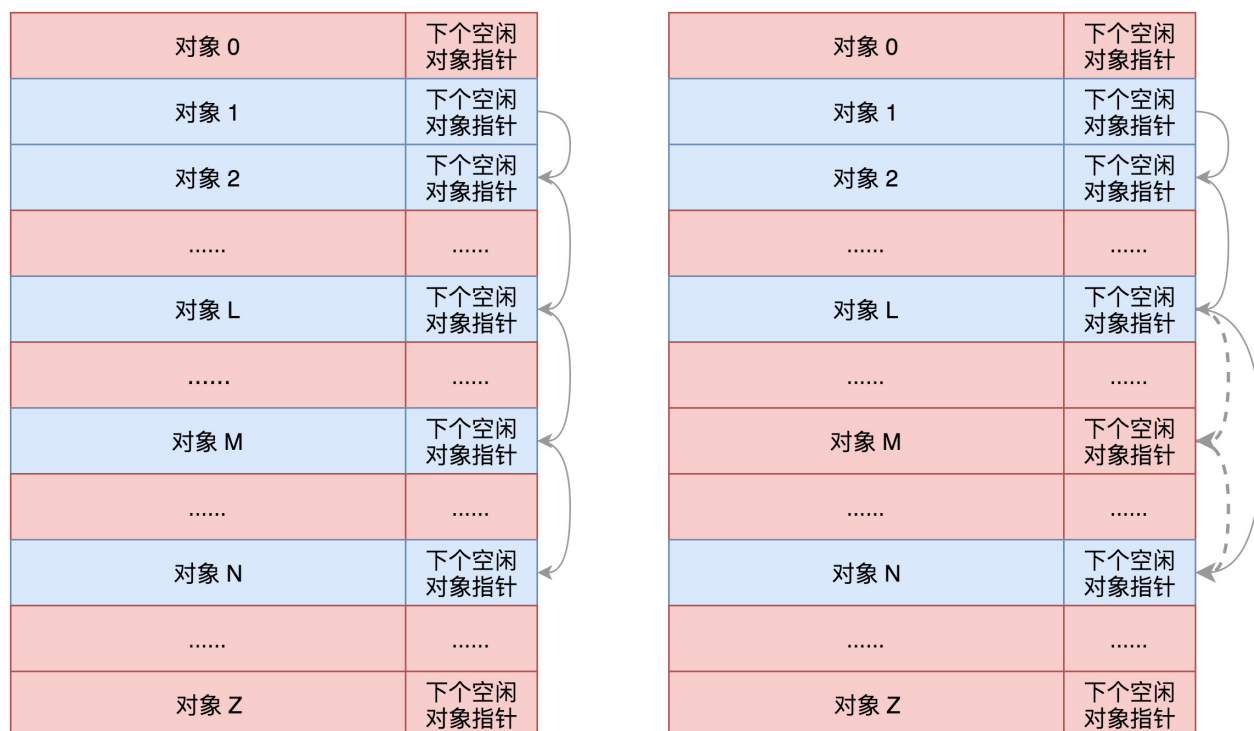
```

在 struct kmem_cache 里面，有个变量 struct list_head list，这个结构我们已经看到过多次了。我们可以想象一下，对于操作系统来讲，要创建和管理的缓存绝对不止 task_struct。难道 mm_struct 就不需要吗？fs_struct 就不需要吗？都需要。因此，所有的缓存最后都会放在一个链表里面，也就是 LIST_HEAD(slab_caches)。

对于缓存来讲，其实就是分配了连续几页的大内存块，然后根据缓存对象的大小，切成小内存块。

所以，我们这里有三个 kmem_cache_order_objects 类型的变量。这里面的 order，就是 2 的 order 次方个页面的大内存块，objects 就是能够存放的缓存对象的数量。

最终，我们将大内存块切分成小内存块，样子就像下面这样。

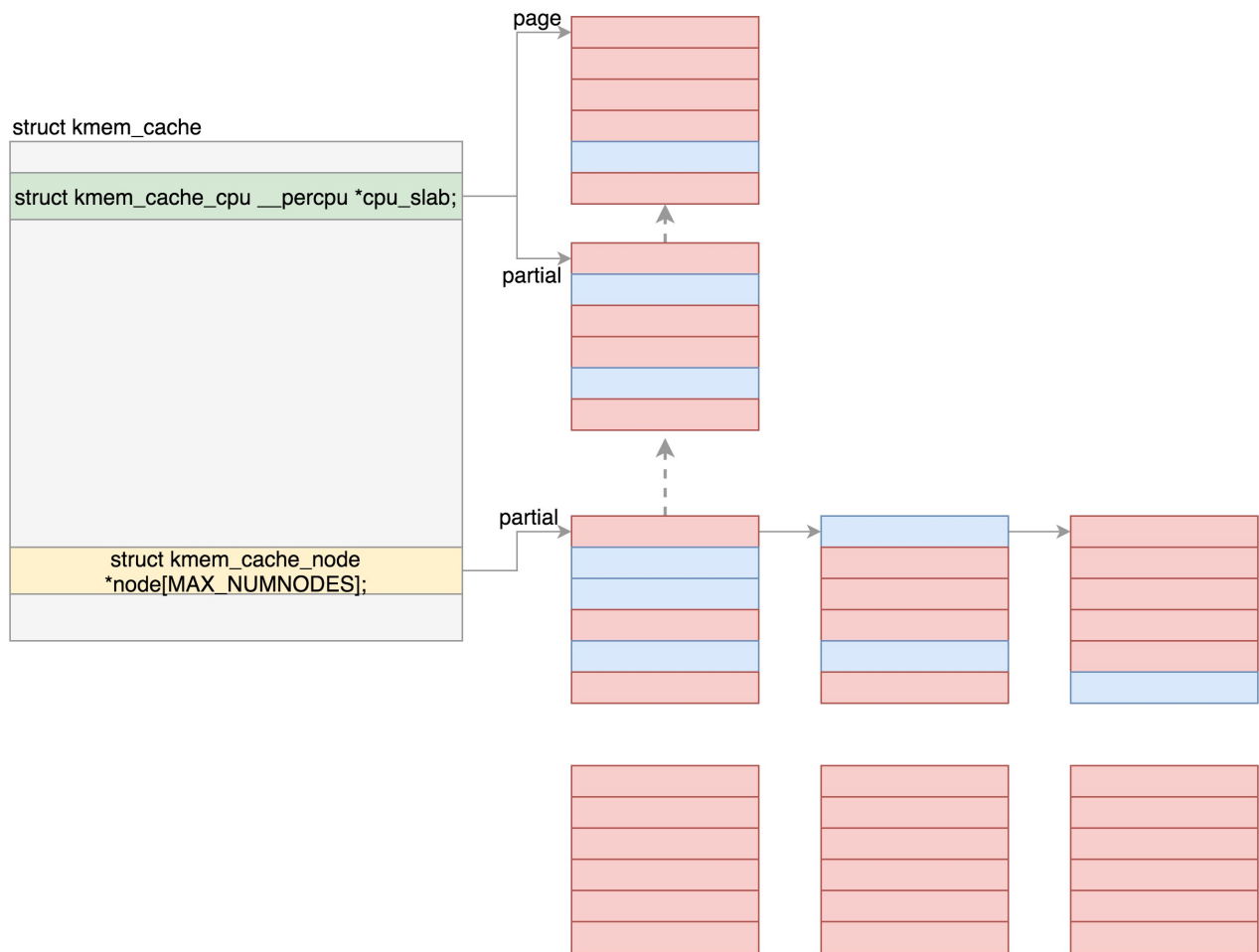


每一项的结构都是缓存对象后面跟一个下一个空闲对象的指针，这样非常方便将所有的空闲对象链成一个链。其实，这就相当于咱们数据结构里面学的，用数组实现一个可随机插入和删除的链表。

所以，这里面就有三个变量：`size` 是包含这个指针的大小，`object_size` 是纯对象的大小，`offset` 就是把下一个空闲对象的指针存放在这一项里的偏移量。

那这些缓存对象哪些被分配了、哪些在空着，什么情况下整个大内存块都被分配完了，需要向伙伴系统申请几个页形成新的大内存块？这些信息该由谁来维护呢？

接下来就是最重要的两个成员变量出场的时候了。`kmem_cache_cpu` 和 `kmem_cache_node`，它们都是每个 NUMA 节点上有一个，我们只需要看一个节点里面的情况。



在分配缓存块的时候，要分两种路径，**fast path**和**slow path**，也就是**快速通道**和**普通通道**。其中 `kmem_cache_cpu` 就是快速通道，`kmem_cache_node` 是普通通道。每次分配的时候，要先从 `kmem_cache_cpu` 进行分配。如果 `kmem_cache_cpu` 里面没有空闲的块，那就到 `kmem_cache_node` 中进行分配；如果还是没有空闲的块，才去伙伴系统分配新的页。

我们来看一下，`kmem_cache_cpu` 里面是如何存放缓存块的。


[复制代码](#)

```
1 struct kmem_cache_cpu {
2     void **freelist;           /* Pointer to next available object */
3     unsigned long tid;        /* Globally unique transaction id */
4     struct page *page;        /* The slab from which we are allocating */
5 #ifdef CONFIG_SLUB_CPU_PARTIAL
6     struct page *partial;      /* Partially allocated frozen slabs */
7 #endif
8     .....
9 };
```

在这里，page 指向大内存块的第一个页，缓存块就是从里面分配的。freelist 指向大内存块里面第一个空闲的项。按照上面说的，这一项会有指针指向下一个空闲的项，最终所有空闲的项会形成一个链表。

partial 指向的也是大内存块的第一个页，之所以名字叫 partial（部分），就是因为它里面部分被分配出去了，部分是空的。这是一个备用列表，当 page 满了，就会从这里找。


我们再来看 kmem_cache_node 的定义。

 复制代码

```
1 struct kmem_cache_node {
2     spinlock_t list_lock;
3     .....
4 #ifdef CONFIG_SLUB
5     unsigned long nr_partial;
6     struct list_head partial;
7     .....
8 #endif
9 };
```

这里面也有一个 partial，是一个链表。这个链表里存放的是部分空闲的大内存块。这是 kmem_cache_cpu 里面的 partial 的备用列表，如果那里没有，就到这里来找。

下面我们就来看看这个分配过程。kmem_cache_alloc_node 会调用 slab_alloc_node。你还是先重点看这里面的注释，这里面说的就是快速通道和普通通道的概念。

 复制代码

```
1 /*
2  * Inlined fastpath so that allocation functions (kmalloc, kmem_cache_alloc)
3  * have the fastpath folded into their functions. So no function call
4  * overhead for requests that can be satisfied on the fastpath.
5  *
6  * The fastpath works by first checking if the lockless freelist can be used.
7  * If not then __slab_alloc is called for slow processing.
8  *
9  * Otherwise we can simply pick the next object from the lockless free list.
10 */
11 static __always_inline void *slab_alloc_node(struct kmem_cache *s,
12     gfp_t gfpflags, int node, unsigned long addr)
```

```

13 {
14     void *object;
15     struct kmem_cache_cpu *c;
16     struct page *page;
17     unsigned long tid;
18     .....
19     tid = this_cpu_read(s->cpu_slab->tid);
20     c = raw_cpu_ptr(s->cpu_slab);
21     .....
22     object = c->freelist;
23     page = c->page;
24     if (unlikely(!object || !node_match(page, node))) {
25         object = __slab_alloc(s, gfpflags, node, addr, c);
26         stat(s, ALLOC_SLOWPATH);
27     }
28     .....
29     return object;
30 }

```

快速通道很简单，取出 `cpu_slab` 也即 `kmem_cache_cpu` 的 `freelist`，这就是第一个空闲的项，可以直接返回了。如果没有空闲的了，则只好进入普通通道，调用 `__slab_alloc`。

 复制代码

```

1 static void *__slab_alloc(struct kmem_cache *s, gfp_t gfpflags, int node,
2                          unsigned long addr, struct kmem_cache_cpu *c)
3 {
4     void *freelist;
5     struct page *page;
6     .....
7 redo:
8     .....
9     /* must check again c->freelist in case of cpu migration or IRQ */
10    freelist = c->freelist;
11    if (freelist)
12        goto load_freelist;
13
14
15    freelist = get_freelist(s, page);
16
17
18    if (!freelist) {
19        c->page = NULL;
20        stat(s, DEACTIVATE_BYPASS);
21        goto new_slab;
22    }
23
24

```



```


25 load_freelist:
26     c->freelist = get_freepointer(s, freelist);
27     c->tid = next_tid(c->tid);
28     return freelist;
29
30
31 new_slab:
32
33
34     if (slub_percpu_partial(c)) {
35         page = c->page = slub_percpu_partial(c);
36         slub_set_percpu_partial(c, page);
37         stat(s, CPU_PARTIAL_ALLOC);
38         goto redo;
39     }
40
41
42     freelist = new_slab_objects(s, gfpflags, node, &c);
43     .....
44     return freeli
45

```

在这里，我们首先再次尝试一下 `kmem_cache_cpu` 的 `freelist`。为什么呢？万一当前进程被中断，等回来的时候，别人已经释放了一些缓存，说不定又有空间了呢。如果找到了，就跳到 `load_freelist`，在这里将 `freelist` 指向下一个空闲项，返回就可以了。

如果 `freelist` 还是没有，则跳到 `new_slab` 里面去。这里面我们先去 `kmem_cache_cpu` 的 `partial` 里面看。如果 `partial` 不是空的，那就将 `kmem_cache_cpu` 的 `page`，也就是快速通道的那一大块内存，替换为 `partial` 里面的大块内存。然后 `redo`，重新试下。这次应该就可以成功了。

如果真的还不行，那就要到 `new_slab_objects` 了。

 复制代码

```

1 static inline void *new_slab_objects(struct kmem_cache *s, gfp_t flags,
2                                     int node, struct kmem_cache_cpu **pc)
3 {
4     void *freelist;
5     struct kmem_cache_cpu *c = *pc;
6     struct page *page;
7
8
9     freelist = get_partial(s, flags, node, c);

```




```

10
11
12     if (freelist)
13         return freelist;
14
15
16     page = new_slab(s, flags, node);
17     if (page) {
18         c = raw_cpu_ptr(s->cpu_slab);
19         if (c->page)
20             flush_slab(s, c);
21
22
23         freelist = page->freelist;
24         page->freelist = NULL;
25
26
27         stat(s, ALLOC_SLAB);
28         c->page = page;
29         *pc = c;
30     } else
31         freelist = NULL;
32
33
34     return freelis

```

在这里面，get_partial 会根据 node id，找到相应的 kmem_cache_node，然后调用 get_partial_node，开始在这个节点进行分配。

 复制代码

```

1  /*
2   * Try to allocate a partial slab from a specific node.
3   */
4  static void *get_partial_node(struct kmem_cache *s, struct kmem_cache_node *n,
5                               struct kmem_cache_cpu *c, gfp_t flags)
6  {
7      struct page *page, *page2;
8      void *object = NULL;
9      int available = 0;
10     int objects;
11     .....
12     list_for_each_entry_safe(page, page2, &n->partial, lru) {
13         void *t;
14
15
16         t = acquire_slab(s, n, page, object == NULL, &objects);
17         if (!t)

```


```

18             break;
19
20
21         available += objects;
22         if (!object) {
23             c->page = page;
24             stat(s, ALLOC_FROM_PARTIAL);
25             object = t;
26         } else {
27             put_cpu_partial(s, page, 0);
28             stat(s, CPU_PARTIAL_NODE);
29         }
30         if (!kmem_cache_has_cpu_partial(s)
31             || available > slub_cpu_partial(s) / 2)
32             break;
33     }
34     .....
35     return object;

```

acquire_slab 会从 kmem_cache_node 的 partial 链表中拿下一大块内存来，并且将 freelist，也就是第一块空闲的缓存块，赋值给 t。并且当第一轮循环的时候，将 kmem_cache_cpu 的 page 指向取下来的这一大块内存，返回的 object 就是这块内存里面的第一个缓存块 t。如果 kmem_cache_cpu 也有一个 partial，就会进行第二轮，再次取下一大块内存来，这次调用 put_cpu_partial，放到 kmem_cache_cpu 的 partial 里面。

如果 kmem_cache_node 里面也没有空闲的内存，这就说明原来分配的页里面都放满了，就要回到 new_slab_objects 函数，里面 new_slab 函数会调用 allocate_slab。

 复制代码

```

1 static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags, int node)
2 {
3     struct page *page;
4     struct kmem_cache_order_objects oo = s->oo;
5     gfp_t alloc_gfp;
6     void *start, *p;
7     int idx, order;
8     bool shuffle;
9
10
11     flags &= gfp_allowed_mask;
12     .....
13     page = alloc_slab_page(s, alloc_gfp, node, oo);
14     if (unlikely(!page)) {
15         oo = s->min;

```

```

16         alloc_gfp = flags;
17         /*
18          * Allocation may have failed due to fragmentation.
19          * Try a lower order alloc if possible
20          */
21         page = alloc_slab_page(s, alloc_gfp, node, oo);
22         if (unlikely(!page))
23             goto out;
24         stat(s, ORDER_FALLBACK);
25     }
26     .....
27     return page;
28 }

```

在这里，我们看到了 `alloc_slab_page` 分配页面。分配的时候，要按 `kmem_cache_order_objects` 里面的 `order` 来。如果第一次分配不成功，说明内存已经很紧张了，那就换成 `min` 版本的 `kmem_cache_order_objects`。

好了，这个复杂的层层分配机制，我们就讲到这里，你理解到这里也就够用了。

页面换出


另一个物理内存管理必须要处理的事情就是，页面换出。每个进程都有自己的虚拟地址空间，无论是 32 位还是 64 位，虚拟地址空间都非常大，物理内存不可能有这么多的空间放得下。所以，一般情况下，页面只有在被使用的时候，才会放在物理内存中。如果过了一段时间不被使用，即使用户进程并没有释放它，物理内存管理也有责任做一定的干预。例如，将这些物理内存中的页面换出到硬盘上去；将空出的物理内存，交给活跃的进程去使用。

什么情况下会触发页面换出呢？

可以想象，最常见的情况就是，分配内存的时候，发现没有地方了，就试图回收一下。例如，咱们解析申请一个页面的时候，会调用 `get_page_from_freelist`，接下来的调用链为 `get_page_from_freelist->node_reclaim->__node_reclaim->shrink_node`，通过这个调用链可以看出，页面换出也是以内存节点为单位的。

当然还有一种情况，就是作为内存管理系统应该主动去做的，而不能等真的出了事儿再做，这就是内核线程 **kswapd**。这个内核线程，在系统初始化的时候就被创建。这样它会进入一

个无限循环，直到系统停止。在这个循环中，如果内存使用没有那么紧张，那它就可以放心睡大觉；如果内存紧张了，就需要去检查一下内存，看看是否需要换出一些内存页。

 复制代码

```
1  /*
2   * The background pageout daemon, started as a kernel thread
3   * from the init process.
4   *
5   * This basically trickles out pages so that we have _some_
6   * free memory available even if there is no other activity
7   * that frees anything up. This is needed for things like routing
8   * etc, where we otherwise might have all activity going on in
9   * asynchronous contexts that cannot page things out.
10  *
11  * If there are applications that are active memory-allocators
12  * (most normal use), this basically shouldn't matter.
13  */
14 static int kswapd(void *p)
15 {
16     unsigned int alloc_order, reclaim_order;
17     unsigned int classzone_idx = MAX_NR_ZONES - 1;
18     pg_data_t *pgdat = (pg_data_t*)p;
19     struct task_struct *tsk = current;
20
21
22     for ( ; ; ) {
23         .....
24         kswapd_try_to_sleep(pgdat, alloc_order, reclaim_order,
25                             classzone_idx);
26         .....
27         reclaim_order = balance_pgdat(pgdat, alloc_order, classzone_idx);
28         .....
29     }
30 }
31
```

这里的调用链是 `balance_pgdat->kswapd_shrink_node->shrink_node`，是以内存节点为单位的，最后也是调用 `shrink_node`。

`shrink_node` 会调用 `shrink_node_memcg`。这里面有一个循环处理页面的列表，看这个函数的注释，其实和上面我们想表达的内存换出是一样的。

 复制代码

```

1  /*
2   * This is a basic per-node page freer.  Used by both kswapd and direct reclaim.
3   */
4  static void shrink_node_memcg(struct pglist_data *pgdat, struct mem_cgroup *memcg,
5                               struct scan_control *sc, unsigned long *lru_pages)
6  {
7      .....
8          unsigned long nr[NR_LRU_LISTS];
9          enum lru_list lru;
10     .....
11     while (nr[LRU_INACTIVE_ANON] || nr[LRU_ACTIVE_FILE] ||
12           nr[LRU_INACTIVE_FILE]) {
13         unsigned long nr_anon, nr_file, percentage;
14         unsigned long nr_scanned;
15
16
17         for_each_evictable_lru(lru) {
18             if (nr[lru]) {
19                 nr_to_scan = min(nr[lru], SWAP_CLUSTER_MAX);
20                 nr[lru] -= nr_to_scan;
21
22
23                 nr_reclaimed += shrink_list(lru, nr_to_scan,
24                                             lruvec, memcg, sc);
25             }
26         }
27     .....
28     }
29     .....

```

这里面有个 lru 列表。从下面的定义，我们可以想象，所有的页面都被挂在 LRU 列表中。LRU 是 Least Recent Use，也就是最近最少使用。也就是说，这个列表里面会按照活跃程度进行排序，这样就容易把不怎么用的内存页拿出来做处理。

内存页总共分两类，一类是**匿名页**，和虚拟地址空间进行关联；一类是**内存映射**，不但和虚拟地址空间关联，还和文件管理关联。

它们每一类都有两个列表，一个是 active，一个是 inactive。顾名思义，active 就是比较活跃的，inactive 就是不怎么活跃的。这两个里面的页会变化，过一段时间，活跃的可能变为不活跃，不活跃的可能变为活跃。如果要换出内存，那就是从不活跃的列表中找出最不活跃的，换出到硬盘上。

```

1 enum lru_list {
2     LRU_INACTIVE_ANON = LRU_BASE,
3     LRU_ACTIVE_ANON = LRU_BASE + LRU_ACTIVE,
4     LRU_INACTIVE_FILE = LRU_BASE + LRU_FILE,
5     LRU_ACTIVE_FILE = LRU_BASE + LRU_FILE + LRU_ACTIVE,
6     LRU_UNEVICTABLE,
7     NR_LRU_LISTS
8 };
9
10
11 #define for_each_evictable_lru(lru) for (lru = 0; lru <= LRU_ACTIVE_FILE; lru++)
12
13
14 static unsigned long shrink_list(enum lru_list lru, unsigned long nr_to_scan,
15                                 struct lruvec *lruvec, struct mem_cgroup *memcg,
16                                 struct scan_control *sc)
17 {
18     if (is_active_lru(lru)) {
19         if (inactive_list_is_low(lruvec, is_file_lru(lru),
20                                 memcg, sc, true))
21             shrink_active_list(nr_to_scan, lruvec, sc, lru);
22         return 0;
23     }
24
25     return shrink_inactive_list(nr_to_scan, lruvec, sc, lru);
26

```

从上面的代码可以看出，shrink_list 会先缩减活跃页面列表，再压缩不活跃的页面列表。对于不活跃列表的缩减，shrink_inactive_list 就需要对页面进行回收；对于匿名页来讲，需要分配 swap，将内存页写入文件系统；对于内存映射关联了文件的，我们需要将在内存中对于文件的修改写回到文件中。

总结时刻

好了，对于物理内存的管理就讲到这里了，我们来总结一下。对于物理内存来讲，从下层到上层的关系及分配模式如下：

物理内存分 NUMA 节点，分别进行管理；

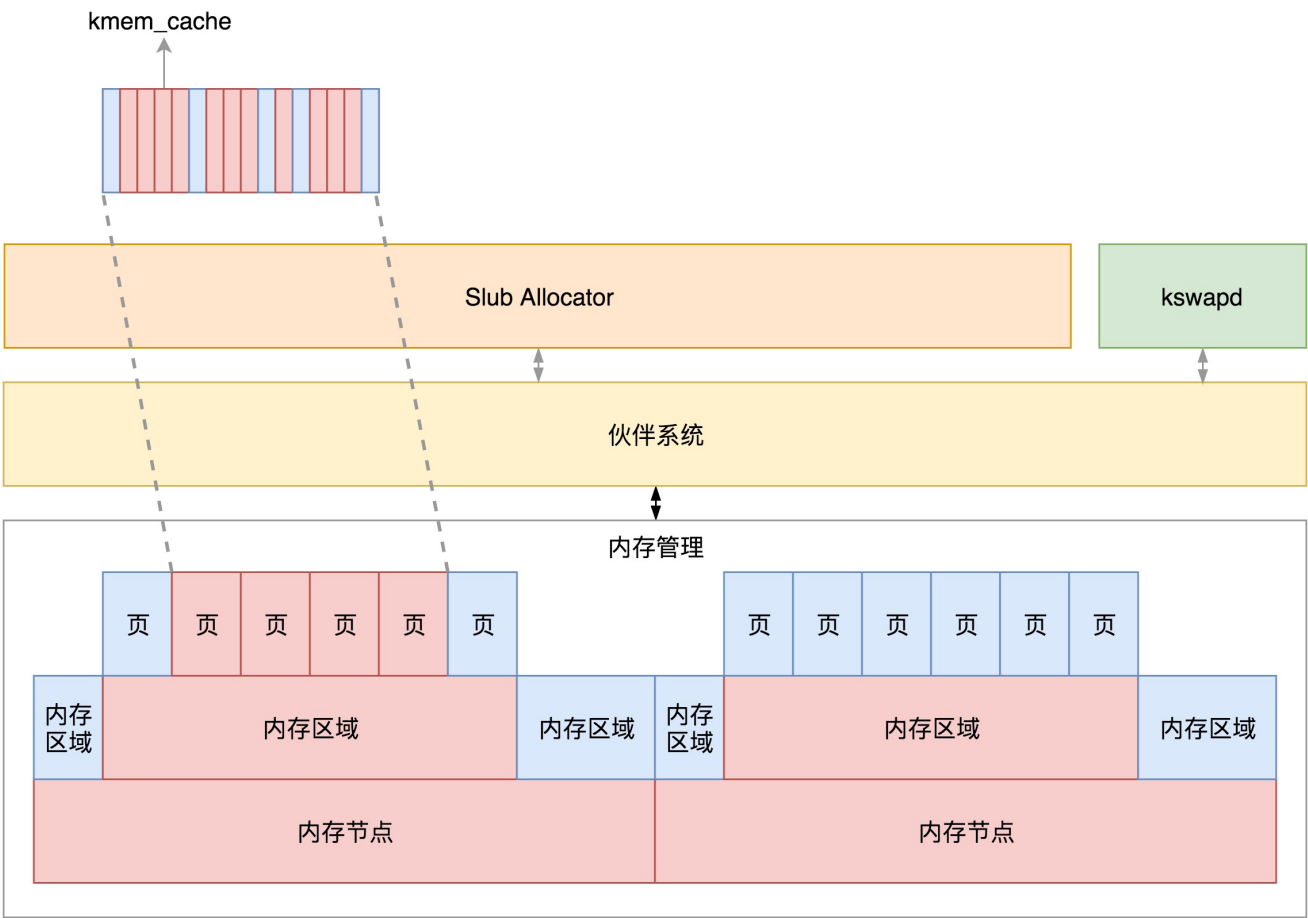
每个 NUMA 节点分成多个内存区域；

每个内存区域分成多个物理页面；

伙伴系统将多个连续的页面作为一个大的内存块分配给上层；

kswapd 负责物理页面的换入换出；

Slub Allocator 将从伙伴系统申请的大内存块切成小块，分配给其他系统。



课堂练习

内存的换入和换出涉及 swap 分区，那你知道如何检查当前 swap 分区情况，如何启用和关闭 swap 区域，如何调整 swappiness 吗？

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 23 | 物理内存管理（上）：会议室管理员如何分配会议室？

下一篇 25 | 用户态内存映射：如何找到正确的会议室？

精选留言 (13)

写留言



W.jyao

2019-05-22

👍 6

真心看不懂了，😞😞😞

展开



...

2019-05-22

👍 5

看到现在，代码越来越多，关于代码的阅读需要么？怎么去做呢

展开



why
2019-05-24



- 小内存分配, 例如分配 task_struct 对象
 - 会调用 kmem_cache_alloc_node 函数, 从 task_struct 缓存区域 task_struct_cachep(在系统初始化时, 由 kmem_cache_create 创建) 分配一块内存
 - 使用 task_struct 完毕后, 调用 kmem_cache_free 回收至缓存池中
 - struct kmem_cache 用于表示缓存区信息, 缓存区即分配连续几个页的大块内存, 再...

展开 ∨



刘強
2019-05-22



这几节看起来吃力了, 需要理解的外围知识很多!

展开 ∨



鲍勃
2019-05-22



果然和你的网络课程一样, 越到后面越hold不住😂

展开 ∨



活的潇洒
2019-05-22



花了4个多小时终于把笔记做完
分享给大家:

<https://www.cnblogs.com/luoahong/p/10907734.html>

展开 ∨



guojiun
2019-05-22



<https://events.static.linuxfound.org/sites/events/files/slides/slaballocators.pdf> 這裡有清楚的視意圖, 對照著看會更清楚!



zhj
2019-05-30



麻烦问下, 换出不活跃物理页的时候, 对于原来进程A是无感知的, A中的页表保存了虚拟

地址到物理地址的映射，在A视角看来还是原来大小的内存，但当A需要访问这个不活跃的页时，这个映射关系已经不成立，所以我的问题就是当换出发生的时候，对A的页表是进行了什么特殊处理，望老师答疑下



川云

2019-05-28



讲得真好，深入到代码层面，否则要是自己研究太困难了，不过也确实需要一定的Linux基础，根据老师的讲解再去看下linux内核的书籍，会提升的更快的

作者回复: 谢谢，这门课主要强调流程，代码是个印证，不适合作为特别严肃的代码分析书籍看，为了突出内核工作机制和流程，删除和省略了很多内核代码



周平

2019-05-27



老师，有个疑问，这些内存小块的分配及释放，都是在物理内存中完成的呢，还是在虚拟内存中完成再映射对对应的物理内存呢？

作者回复: 物理内存分配完毕一整页的时候，会通过page_address分配一个虚拟地址，小块都是基于这个虚拟地址的。



chengzise

2019-05-23



老师好，这两节讲的是物理内存的管理和分配，大概逻辑是看的懂的，细节需要继续反复研读才行。其中有个问题，kmem_cache部分已经属于页内小内存分配，这个分配算法属于虚拟内存分配，不算物理内存管理范畴吧？就是说先分配物理内存页，映射到虚拟内存空间，再在这个虚拟内存分配小内存给程序逻辑使用。希望老师解答一下。

展开 ∨

作者回复: 是的，会映射成为虚拟地址，是整页会有虚拟地址



安排



2019-05-23

slab分配器内容很多，这样讲一讲大体框架也不错，要不然就更看不懂了

展开 ▾



Brigand

2019-05-22



怎么自定义页面换出？

展开 ▾