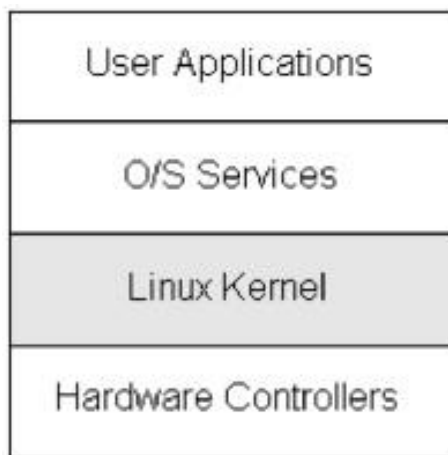




## 第0018讲 反碎片技术



零声学院讲师: Vico老师



## 一、虚拟可移动区域

## 二、内存碎片整理



内存碎片可分为内部碎片和外部碎片，其中内部碎片指内存页里面的碎片，外部碎片指空闲的内存页分散，很难找到一组物理地址连续的空闲内存页，无法满足超过一页的内存分配请示。

**为解决外部碎片问题，内核引入反碎片技术：**

a)2.6.23版本引入虚拟可移动区域；

b)2.6.23版本引入成块回收（集中回收），从3.5版本废除，被内存碎片整理技术取代。

虚拟可移动区域和根据可移动性分组是预防外部碎片的技术，成块回收和内存碎片整理是在出现外部碎片以后消除外部碎片的技术。



# 一、虚拟可移动区域

可移动区域 (ZONE\_MOVABLE) 是一个伪内存区域，基本思想是：把物理内存分为两个区域，一个区域用于分配不可移动的页，另一个区域用于分配可移动的页，防止不可移动页向可移动区域引入碎片。

## 技术原理：

可移动区域 (ZONE\_MOVABLE) 没有包含任何物理内存，所以我们说它是伪内存区域，或者说是虚拟的内存区域。可移动区域借用最高内存区域的内存，在32位系统上最高的内存区域通常是高端内存区域 (ZONE\_HIGHMEM)，在64位系统上最高的内存区域通常是普通区域 (ZONE\_NORMAL)。



## 1、解析内核引导参数

函数cmdline\_parse\_kernelcore解析内核引导参数 “kernelcore” , 其内核源码分析

如下:

```
C page_alloc.c X
mm > C page_alloc.c
6622 static int __init cmdline_parse_kernelcore(char *p)
6623 {
6624     /* parse kernelcore=mirror */
6625     if (parse_option_str(p, "mirror")) {
6626         mirrored_kernelcore = true;
6627         return 0;
6628     }
6629
6630     return cmdline_parse_core(p, &required_kernelcore);
6631 }
6632
```



## 成员函数源码分析:

C page\_alloc.c X

mm > C page\_alloc.c

```
6637 static int __init cmdline_parse_movablecore(char *p)
6638 {
6639     return cmdline_parse_core(p, &required_movablecore);
6640 }
6641
6642 early_param("kernelcore", cmdline_parse_kernelcore);
6643 early_param("movablecore", cmdline_parse_movablecore);
6644
6645 #endif /* CONFIG_HAVE_MEMBLOCK_NODE_MAP */
```



内核函数cmdline\_parse\_movable\_node解析内核引导参数“movable\_node”，把全局变量movable\_node\_enabled设置为真，表示所有可以热插拔的物理内存都作为可移动区域，源码如下：

```
mm > C memory_hotplug.c
1763 static int __init cmdline_parse_movable_node(char *p)
1764 {
1765     #ifdef CONFIG_MOVABLE_NODE
1766         movable_node_enabled = true;
1767     #else
1768         pr_warn("movable_node option not supported\n");
1769     #endif
1770     return 0;
1771 }
1772 early_param("movable_node", cmdline_parse_movable_node);
```





## 2、确定可移动区域的范围

函数find\_zone\_movable\_pfns\_for\_nodes确定可移动区域的范围

确定可移动域从哪个内存区域借用物理页：调用find\_usable\_zone\_for\_movable以查找包含物理页的最高内存区域，全局变量movable\_zone保存借用区域的索引。

```
mm > C page_alloc.c
5655 static void __init find_usable_zone_for_movable(void)
5656 {
5657     int zone_index;
5658     for (zone_index = MAX_NR_ZONES - 1; zone_index >= 0; zone_index--) {
5659         if (zone_index == ZONE_MOVABLE)
5660             continue;
5661
5662         if (arch_zone_highest_possible_pfn[zone_index] >
5663             arch_zone_lowest_possible_pfn[zone_index])
5664             break;
5665     }
5666
5667     VM_BUG_ON(zone_index == -1);
5668     movable_zone = zone_index;
5669 }
5670
```





确定每个内存节点中可移动区域的起始物理页号，使用全局数组保存如下：

```
mm > C page_alloc.c
273 static unsigned long __initdata required_kernelcore;
274 static unsigned long __initdata required_movablecore;
275 static unsigned long __meminitdata zone_movable_pfn[MAX_NUMNODES];
276 static bool mirrored_kernelcore;
277
```



### 3、从可移动区域分配物理页

申请物理页的时候，如果同时指定分配标志 `_GFP_HIGHMEM` 和 `_GFP_MOVABLE`，页分配器的核心函数如下图所示，计算出首选的内存区域是可移动区域，首先尝试从可移动区域分配物理页。如果可移动区域分配失败，从备用的内存区域借用物理页。

```
mm > C page_alloc.c
4003  */
4004  struct page *
4005  __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
4006                       struct zonelist *zonelist, nodemask_t *nodemask)
4007  {
4008      struct page *page;
4009      unsigned int alloc_flags = ALLOC_WMARK_LOW;
```

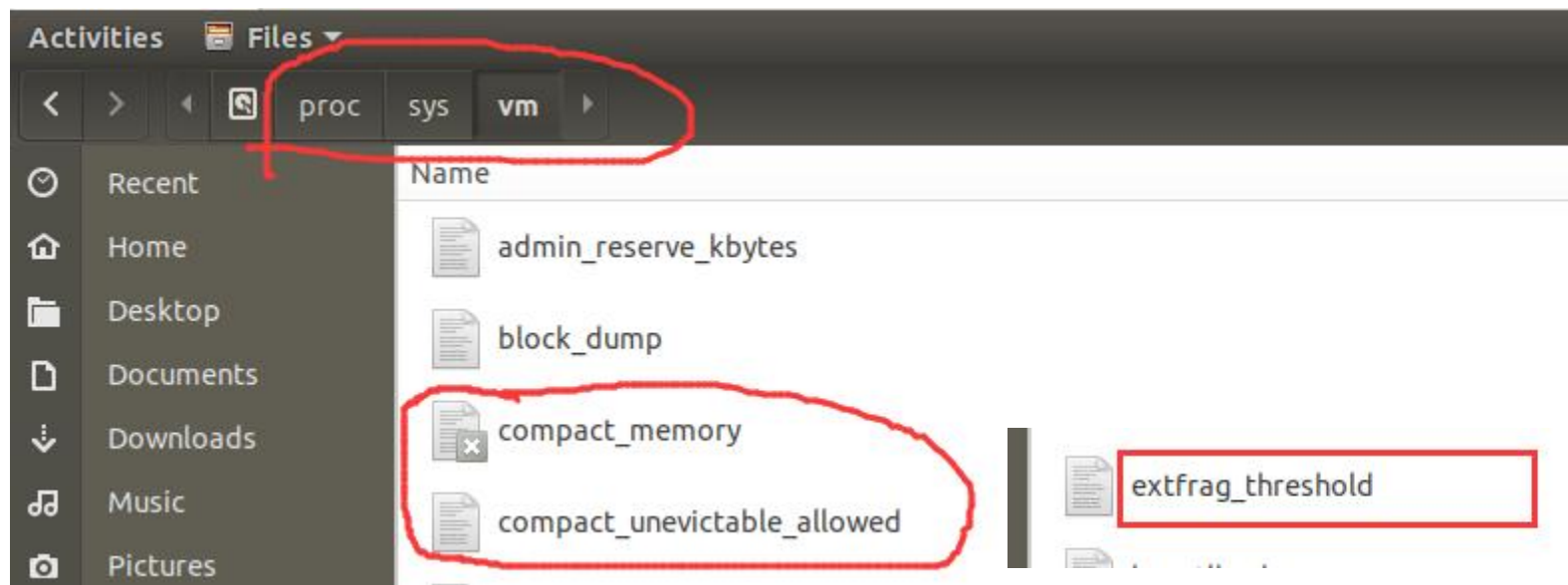


## 二、内存碎片整理



内存碎片整理 (memory compaction, 又称为内存紧缩) 的基本思想: 从内存区域的底部扫描已分配的可移动页, 从内存区域的顶部扫描空闲页, 把底部的可移动页移到顶部的空闲页, 在底部形成连续的空闲页。

系统内存碎片整理配置文件如下:





## 内存碎片整理算法:

- a. 首先从内存区域的底部向顶部以页块为单位扫描, 在页块内部从起始页向结束页扫描, 把这个页块里面的可移动页组成一条链表;
- b. 然后从内存区域的顶部向底部以页块为单位扫描, 在页块内部也是从起始页向结束页扫描, 把空闲页组成一条链表;
- c. 最后把底部的可移动页的数据复制到顶部的空闲页, 修改进程的页表, 把虚拟页映射到新的物理页。

## 内存碎片整理优先级:

- a. 完全同步模式
- b. 轻量级同步模式
- c. 异步模式



## 何时执行内存碎片整理???

### 内存碎片整理结束条件如下:

- a. 如果迁移扫描器和空闲扫描器相遇, 那么内存碎片整理结束
- b. 如果迁移扫描器和空闲扫描器没有相遇, 但是申请或备用的迁移类型至少有一个足够大的空闲页块, 那么内存碎片整理结束。



## 1、内存碎片整理推迟

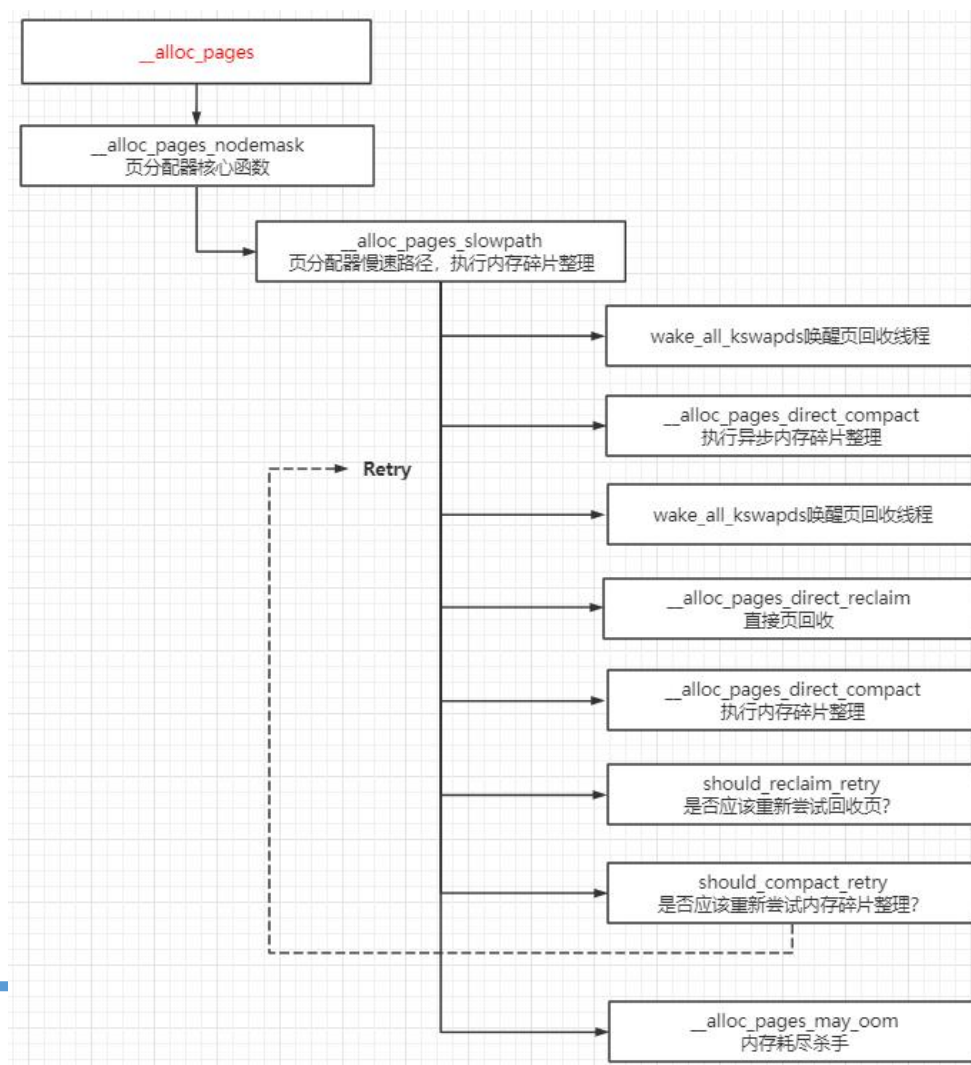
内存碎片整理成功的标准：（空闲页数-申请页数）大于或等于水线，并且申请或备用的迁移类型至少有一个足够大的空闲页块。

内核在内存区域增加3上成员来标记内存碎片整理推迟信息：

```
C mmzone.h 2 X
include > linux > C mmzone.h > zone > compact_cached_free_pfn
4//
478 #ifdef CONFIG_COMPACTION
479     /*
480     * On compaction failure, 1<<compact_defer_shift compactions
481     * are skipped before trying again. The number attempted since
482     * last failure is tracked with compact_considered.
483     */
484     unsigned int    compact_considered;
485     unsigned int    compact_defer_shift;
486     int             compact_order_failed;
487 #endif
```



## 2、页分配器执行内存碎片整理流程源码分析








## 页分配器执行内存碎片整理流程源码分析（续）



函数 `__alloc_pages_nodemask` 是页分配器的核心函数，函数 `__alloc_pages_slowpath` 是页分配器慢速路径，执行内存碎片整理，页分配器执行内存碎片整理流程如下：

```
include > linux > C gfp.h > arch_alloc_page(page *, int)
438 static inline struct page *
439 __alloc_pages(gfp_t gfp_mask, unsigned int order,
440              struct zonelist *zonelist)
441 {
442     return __alloc_pages_nodemask(gfp_mask, order, zonelist, NULL);
443 }
444
```



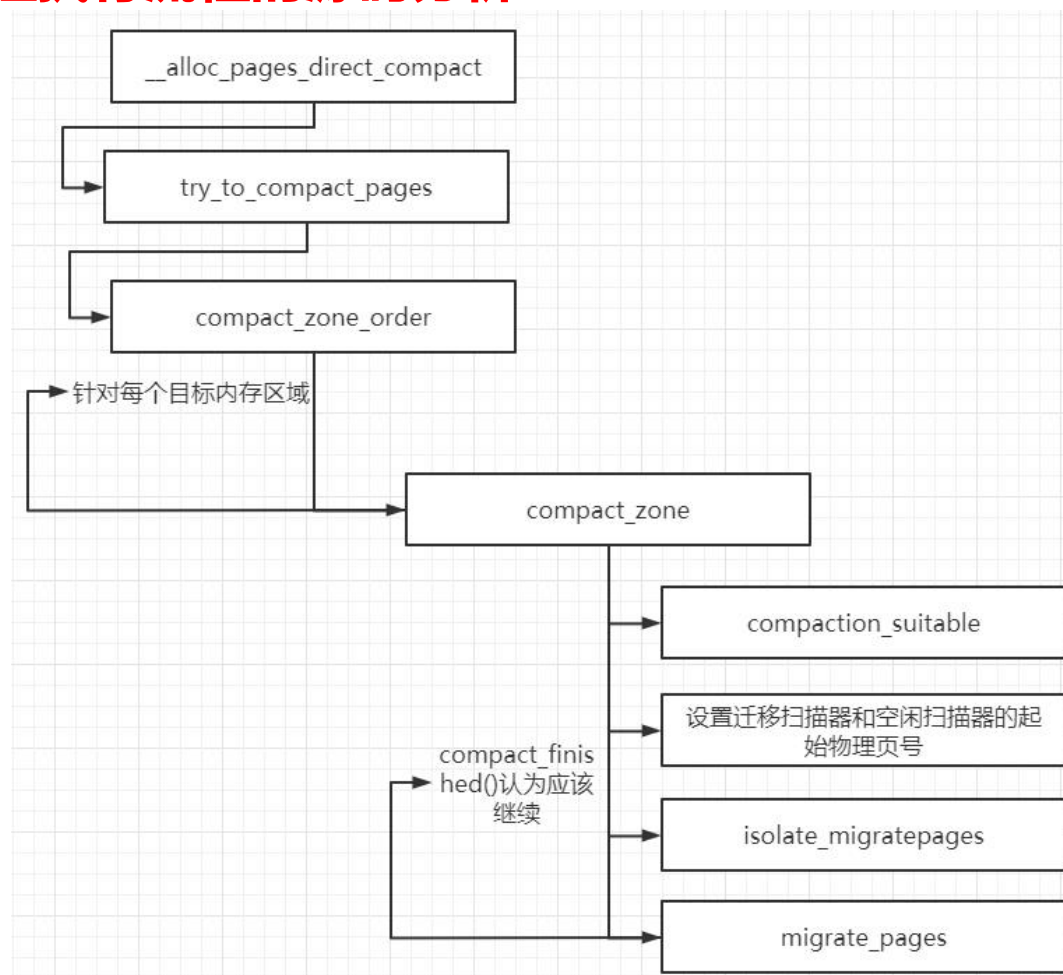
## 页分配器执行内存碎片整理流程源码分析 (续)

```
mm > C page_alloc.c >  __alloc_pages_nodemask(gfp_t, unsigned int, zonelist *, nodemask_t *)  
4004 struct page *  
4005 __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,  
4006 struct zonelist *zonelist, nodemask_t *nodemask)  
4007 {  
4008     struct page *page;  
4009     unsigned int alloc_flags = ALLOC_WMARK_LOW;  
4010     gfp_t alloc_mask = gfp_mask; /* The gfp_t that was actually used for allocation */  
4011     struct alloc_context ac = { };  
4012
```

```
mm > C page_alloc.c >  __alloc_pages_nodemask(gfp_t, unsigned int, zonelist *, nodemask_t *)  
4039  
4040     page =  __alloc_pages_slowpath(alloc_mask, order, &ac);  
4041
```



### 3、内存碎片整理执行流程的源码分析





办学宗旨：一切只为渴望更优秀的你

办学愿景：让技术简单易懂