# Linux物理内存
# 页面分配

http://www.ilinuxkernel.com

原文链接地址：http://ilinuxkernel.com/?p=1371

欢迎各位批评指正或留下问题。

# 目　　录

# 1 概述

在用户态C语言程序中，我们对内存分配函数malloc（）或calloc（）非常熟悉；函数执行成功，就会返回需要的内存起始地址。显然这些函数在在内核态没法运行，在内核态有专门的内存申请/释放函数。

Linux内核中，如何分配和回收内存？空闲内存如何管理？本文以linux 2.6.32-220.el6版本内核源码为基础，介绍Linux内核中如何分配物理内存页面。

# 2 内核页面分配与回收API

我们先来了解一下内核中有哪些内存页面分配与回收API，常用的API如下表：

表1 物理内存页面分配与回收API

| 函数 | 描述 |
|---|---|
| alloc_page(gfp_mask) | 分配一个页面，返回页面数据结构 |
| alloc_pages(gfp_mask,order) | 分配 2 order 个页面，返回第一个页面的数据结构 |
| __get_free_page(gfp_mask) | 分配一个页面，且返回页面的逻辑地址 |
| __get_free_pages(gfp_mask, | 分配 2 order 个页面，且返回页面的逻辑地址 |
| get_zeroed_page(gfp_mask) | 分配 1 个页面，数据清零、且返回逻辑地址 |
| get_dma_pages(gfp_mask,order) | 分配适合 DMA 操作的页面 |
| | |
| __free_pages(page,order) | 释放 2 order 个页面，第一个参数为页面地址 |
| free_pages(addr,order) | 释放 2 order 个页面，第一个参数为逻辑地址 |
| free_page(addr) | 释放 2 order 个页面 |

注：在内核编程中，通常不直接和这些函数打交道，而常用kmalloc（）、vmalloc（）、kmem_cache_alloc（）等函数。这里我们只介绍页面分配，在slab机制中会介绍kmalloc（）等函数。

页面分配各个函数之间关系如下图：

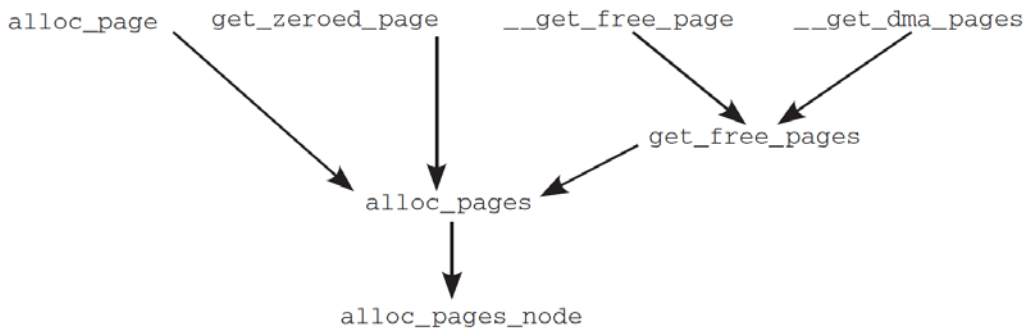


图2 页面分配函数之间关系

页面回收各个函数之间关系如下图：

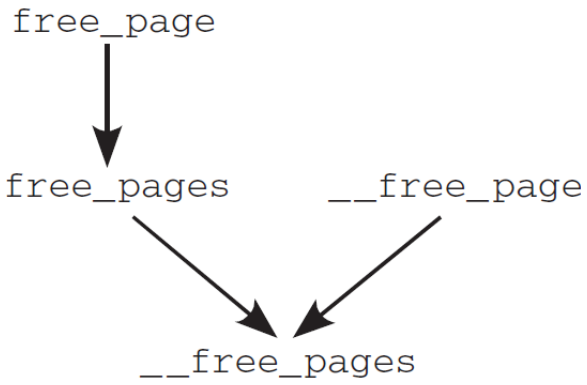


图3 页面回收函数之间关系

# 3 空闲页面的管理

## 3.1 物理内存空间描述

在《Linux物理内存描述》（http://ilinuxkernel.com/?p=1332）中，我们详细介绍了内核将物理分为三个层次：节点（Node）、区域（Zone）和页面（Page）。

物理内存空间描述处于最高层的为结点（Nodes），然后结点中包括多个区域（Zones），最后区域中包含很多页面（Pages），三者关系下图所示。

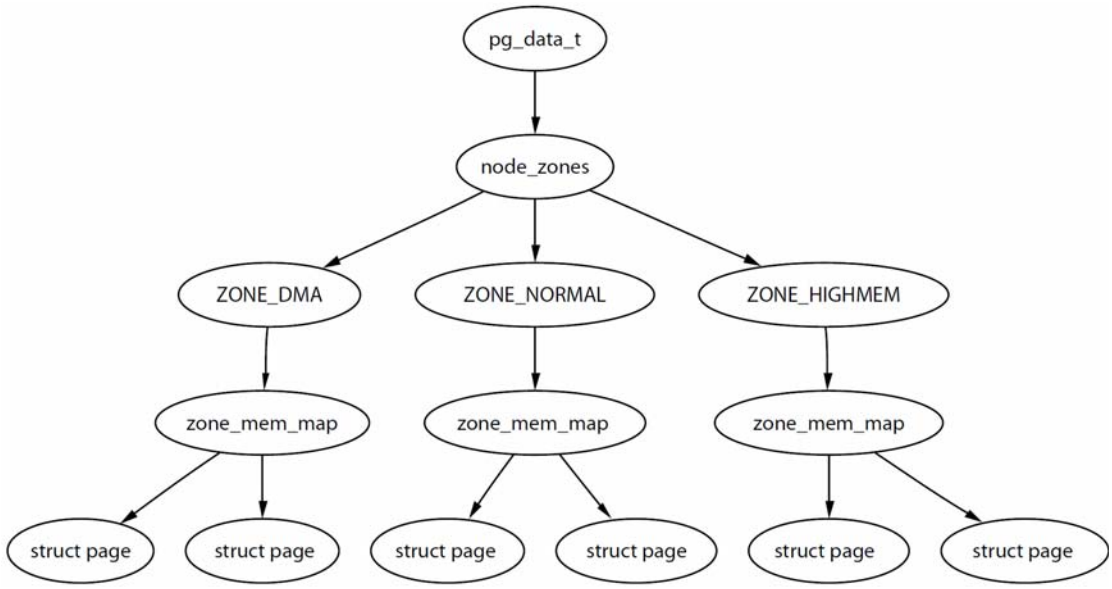


图4 结点、区域及页面关系图

## 3.2 空闲页面的管理

在页面分配与回收时，必然涉及到空闲页面是如何管理的。

在超市购物时，我们可以留意一下收银员是如何管理钱（组织、存放）的。通常会把5角的全部放在一起、1元的全部放在一起 ... ... 面值100元的全部放在一起。Linux内核空闲内存管理的思想类似。

在Linux内核中，空闲内存管理的基本单位是页面（x86/x86-64 CPU定义的页面），即以页面为单位来管理物理内存（kmalloc等slab/slub机制，是比页面更小的细分）。

同超市收银员管理钱款一样，Linux内核管理的每个内存空闲块都是2的幂次方个页面，幂次方的大小为**order**。把1个空闲页面的放在一起、2个空闲页面（物理地址连续）放在一起、4个空闲页面（物理地址连续）放在一起 ... ... $2^{MAX_ORDER-1}$个页面（物理地址连续）放在一起。空闲页面组织，如下图所示。

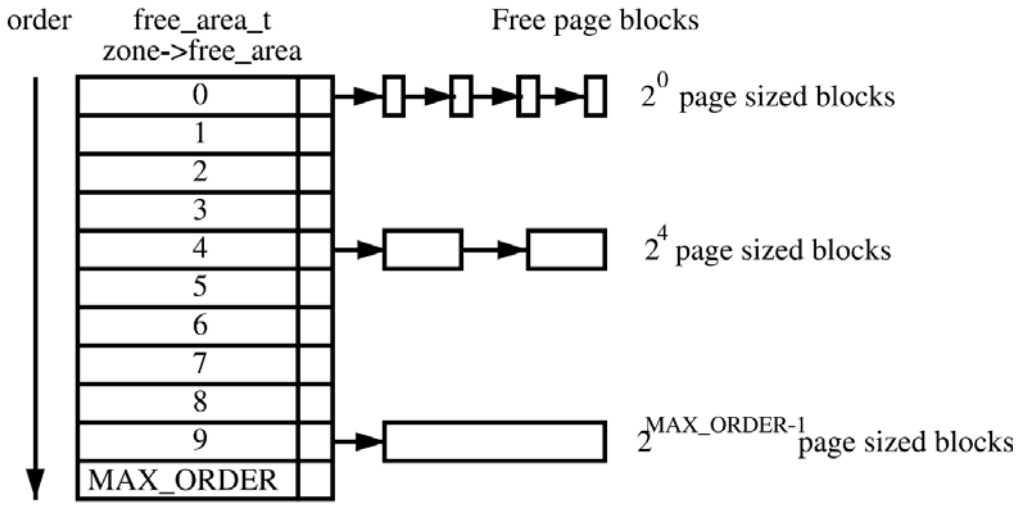


图5 空闲页面块的管理

在2.6.32-220.el6内核中，MAX_ORDER通常定义为11，内核管理最大的连续空闲物理内存大小为$2^{11-1}$个页面，即4MB。

```
00022: /* Free memory management - zoned buddy allocator. */
00023: #ifndef CONFIG_FORCE_MAX_ZONEORDER
00024: #define MAX_ORDER 11
00025: #else
00026: #define MAX_ORDER CONFIG_FORCE_MAX_ZONEORDER
00027: #endif
00028: #define MAX_ORDER_NR_PAGES (1 << (MAX_ORDER - 1))
```

## 区域（zone）与空闲页面

在区域（zone）的数据结构中，有个数组**free_area**[MAX_ORDER]来保存每个空闲内存块链表，

```
00287: struct zone {
00288:         /* Fields commonly accessed by the page allocator */
00289:
    ... ...
00331:         struct free_area     free_area[MAX_ORDER];
    ... ...
00445:         unsigned long padding[16];
00446: } ?     end zone ? ____cacheline_internodealigned_in_smp;
```

这样**free_area**[MAX_ORDER]数组中的第1个元素，指向内存块大小为$2^0$即1个页面的空闲页面链表；数组中的第2个元素指向内存块大小为$2^0$即1个页面的空闲页面链表；最后一个元素指向最大的空闲内存块链表，大小为$2^{MAX_ORDER-1}$个页面，目前MAX_ORDER的值定义为11。

**每个区域（zone）都有一个**free_area[MAX_ORDER]数组，其数据类型free_area结构体定义如下：

```
00057: struct free_area {
00058:         struct list_head     free_list[MIGRATE_TYPES];
00059:         unsigned long        nr_free;
00060: };
```

各成员变量含义如下：

***free_list***：空闲页面块的双链表；

***nr_free***：该区域中的空闲页面块数量；

每个空闲页面链表上各个元素（大小相同的连续物理页面），通过struct page中的双链表成员变量来连接，如下图所示。

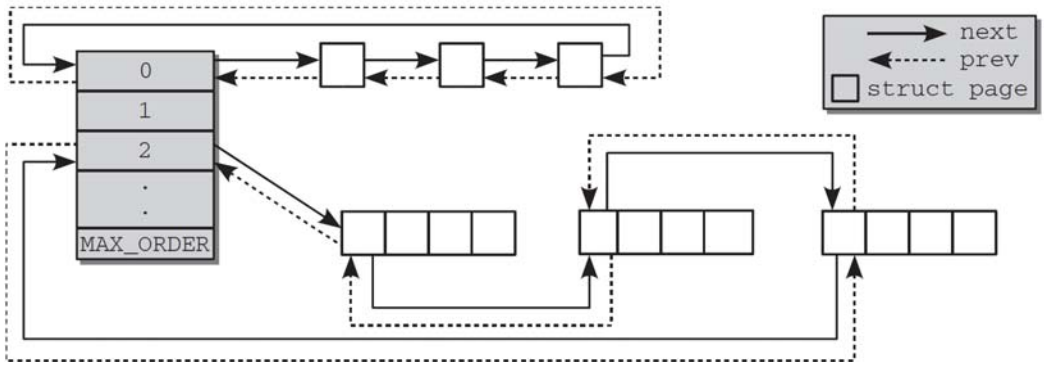


图6 空闲页面链表的管理

我们知道Linux内核描述物理内存有三个层次：节点、区域和页面。空闲页面的管理只是在区域（Zone）这一层，节点（Node）下的每个区域都管理着自己的空闲物理页面。空闲页面管理与节点、区域之间的关系如下图。

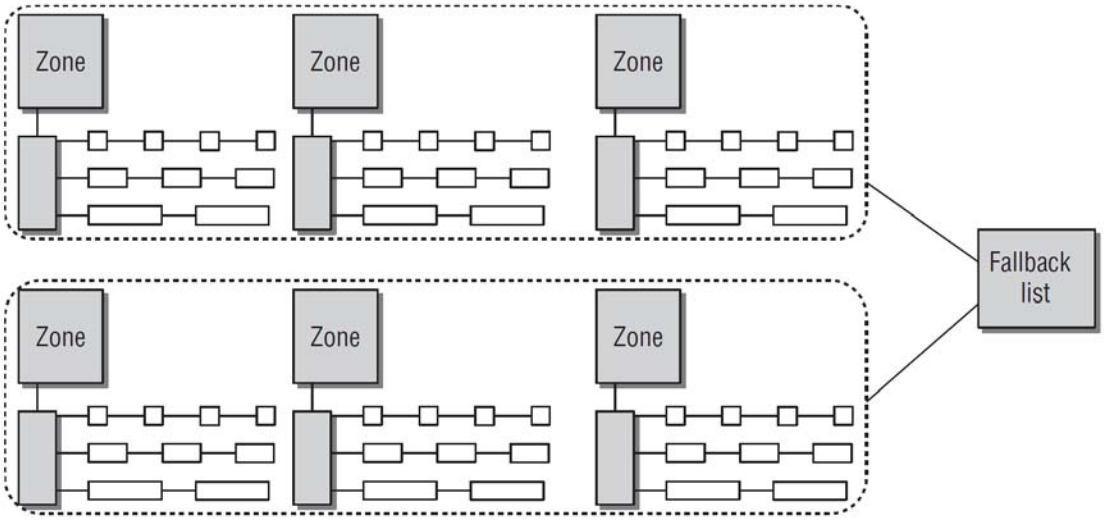


图7 空闲页面管理与节点、区域之间的关系

# 4 伙伴算法

## 4.1 Buddy System

伙伴系统（Buddy System）在理论上是非常简单的内存分配算法。它的用途主要是尽可能减少外部碎片（external fragmentation），同时允许快速分配与回收物理页面。为了减少外部碎片，连续的空闲页面，根据空闲块（由连续的空闲页面组成）大小，组织成不同的链表（或者orders）。前面一节介绍的空闲物理页面管理就是伙伴系统的一部分，

这样所有的2个页面大小的空闲块在一个链表中，4个页面大小的空闲块在另外一个链表中，以此类推。注意，不同大小的块在空间上，不会有重叠。下图为空闲页面的分配示意

图。

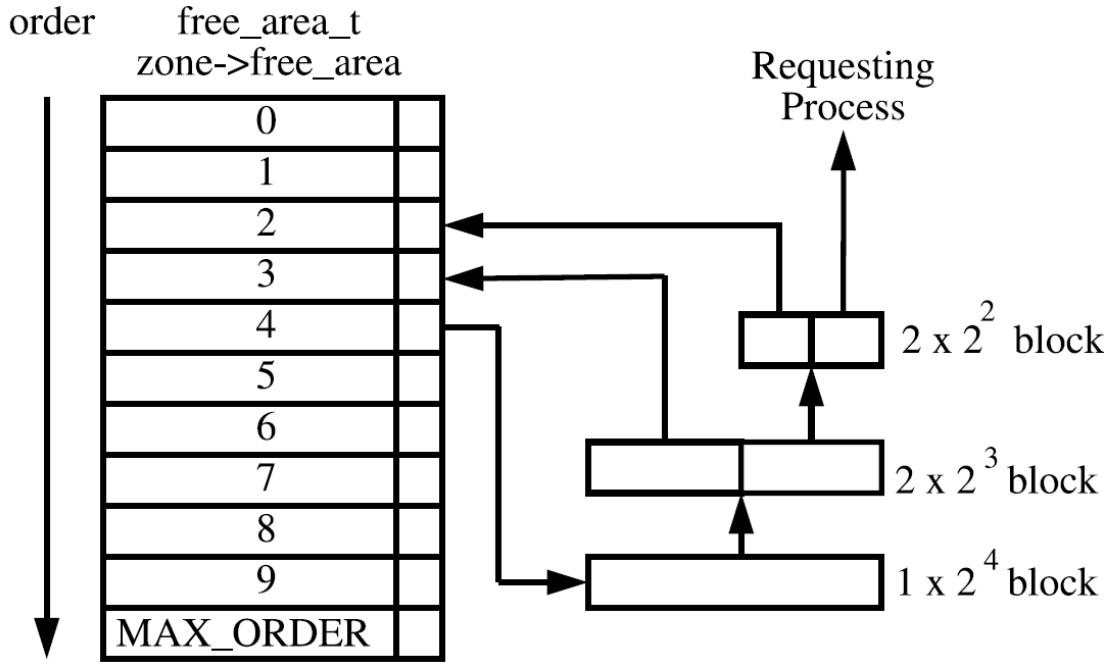


图8 空闲页面的分配

当一个需求为4个连续页面时，检查是否有大小为$2^{3-1}$个页面的空闲块而快速满足请求。若该链表上（每个结点都是大小为4页面的块）有空闲的块，则分配给用户，否则向下一个级别（order）的链表中查找。若存在（8页面的）空闲块（现处于另外一个级别的链表上），则将该页面块分裂为两个4页面的块，一块分配给请求者，另外一块加入到4页面的块链表中。这样可以避免分裂大的空闲块，而此时有可以满足需求的小页面块，从而减少外面碎片。

## 4.2 伙伴算法举例

前面的文字描述显得较为抽象，现拿一个具体例子来说明伙伴算法的内容。假设我们的系统内存只有32个页面RAM，物理页面使用情况如下图所示。


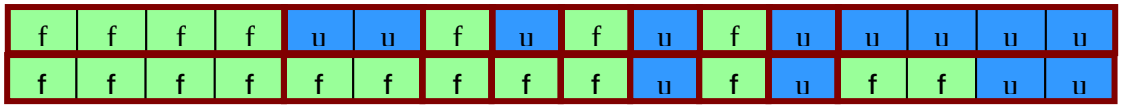


图9 伙伴算法示例

此时空闲内存页面组织如下图。order=1的链表（大小$2^{1-1}$个页面）上共有5个结点；order=2的链表（大小$2^{2-1}$个页面）上共有3个结点；order=3的链表为空；order=4的链表（大

小$2^{4-1}$个页面）上共有1个结点；order=5的链表为空；order=6的链表为空。

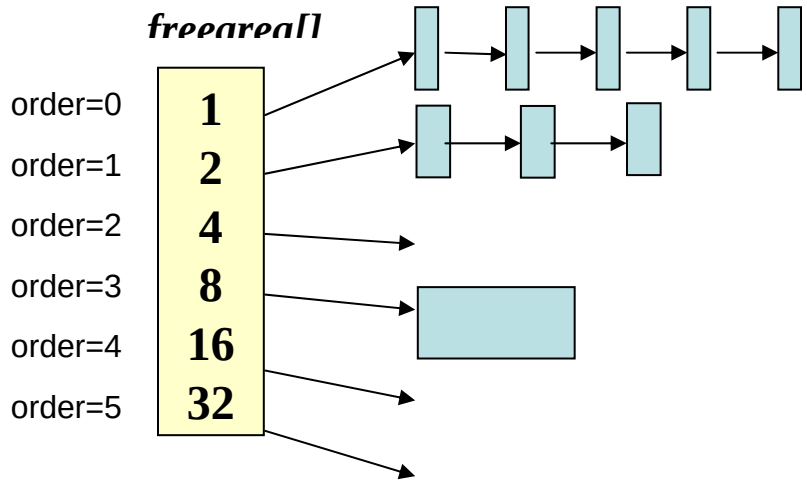


图10 伙伴算法示例 - 空闲页面组织

### 4.2.1 页面分配过程

现在上层请求分配4个**地址连续**的空闲物理页面块。

分配页面块步骤如下：

（1）4 = $2^{3-1}$，因此从Order = 3的空闲块链表上开始找空闲的块；

（2）由于在order = 3的链表上，没有空闲块；需要到上一级order查找是否有空闲块；

（2）从Order = 4的链表上开始查找，有一个空闲结点；但该链表上的每个节点块大小为8个页面，分配4个页面给上层，标记该页面表为已使用。

（3）还剩4个页面。此时将该剩下的4个页面，放入order=3的链表上；

（4）更新相关统计信息。

注：分配页面完成后，还要更新其他数据结构，在代码详细分析时会介绍，这里关注页面分配过程。

分配4个页面块给上层后，空闲页面组织如下图。

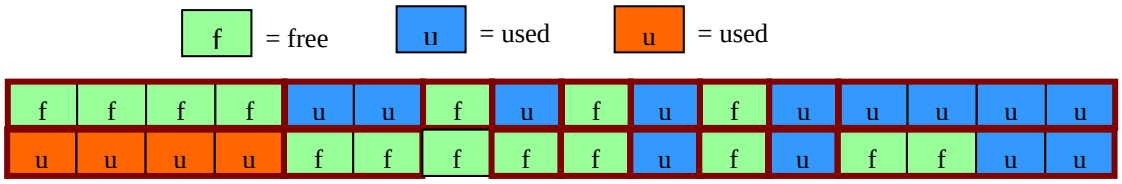


图11 伙伴算法分配页面块示例

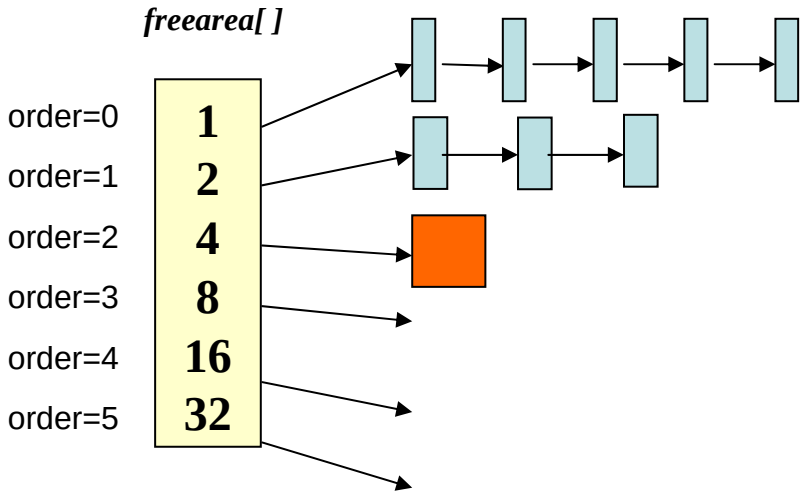


图12 伙伴算法示例 - 分配一个块后空闲页面组织

问题：是否存在非2的幂次方个页面申请请求？如上层一次性申请6个页面。在Linux内核中，是不存在这样的内存大小请求，API函数中保证了必须是申请$2^{order-1}$个页面。若上层确实要申请6个页面，则可以一次性申请8页面，也可以申请4（order =3 ）+2（order = 2）个页面（但要保证这6个页面物理地址连续）。

### 4.2.2 页面回收过程

现在上层释放了一个物理页面，如紫色标注。

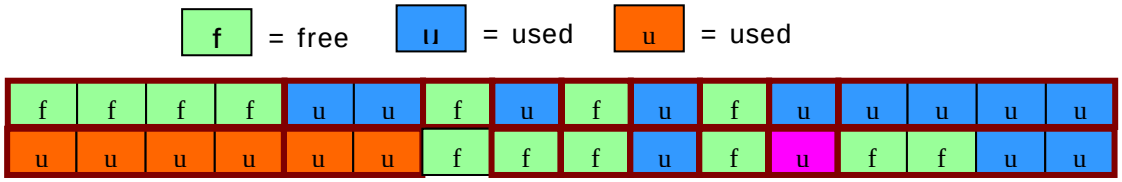


图13 伙伴算法回收页面块示例

回收页面块步骤如下：

（1）标记该页面块为空闲；

（2）检查相邻物理页面是否为空闲；若相邻物理页面为空闲，则尝试合并成更大的连续物理页面块（这样可以避免内存碎片化）；

（3）若有合并，则要更新freearea [] 中链表元素；

（4）更新相关统计信息。

从上面例子，可以看到释放的页面前后页面都是空闲，这样可以合并成4个连续空闲物

理页面（order = 3，$2^{3-1}$ ）。 回收页面，且合并后空闲页面组织如下图：

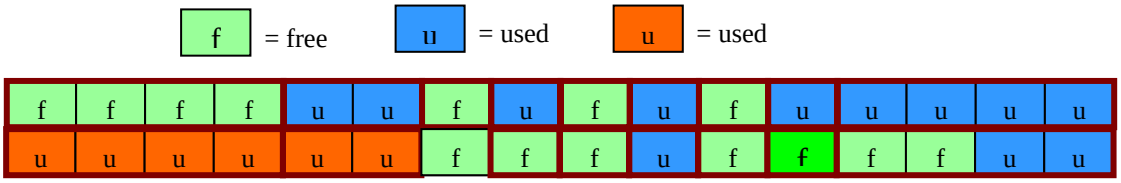


图14 系统物理内存页面状态

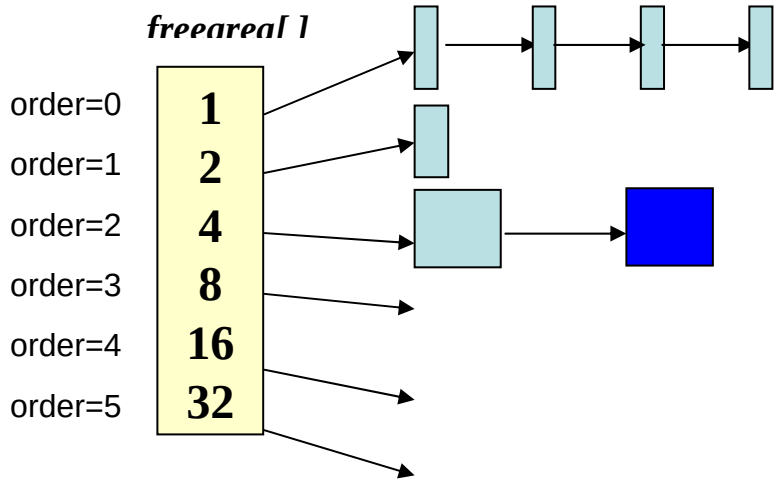


图15 伙伴算法示例 - 回收一个块后空闲页面组织

## 4.3 Buddy系统信息查看

我们可以查看**/proc/buddyinfo**文件内容来了解当前系统Buddy系统状态。

```
root@yiquan-ThinkPad-X200:/proc# cat buddyinfo
Node 0, zone      DMA      1      0      0      1      2      1      1      0      1      1      3
Node 0, zone    DMA32     10      7      5      8      8      7      6      7      3      3    731
Node 0, zone   Normal    244    135     61     33     18     13     12      9      2      1     33
root@yiquan-ThinkPad-X200:/proc#
```

也可以通过echo m > /proc/sysrq-trigger来观察Buddy系统状态。与/proc/buddyinfo的信息是一致。

*[ 134.154722] Node 0 DMA: 1*4kB 0*8kB 0*16kB 1*32kB 2*64kB 1*128kB 1*256kB 0*512kB 1*1024kB 1*2048kB 3*4096kB = 15908kB*
*[ 134.154747] Node 0 DMA32: 10*4kB 7*8kB 5*16kB 8*32kB 8*64kB 7*128kB 6*256kB 7*512kB 3*1024kB 3*2048kB 731*4096kB = 3010352kB*
*[ 134.154770] Node 0 Normal: 202*4kB 227*8kB 51*16kB 120*32kB 90*64kB 44*128kB 19*256kB 8*512kB 5*1024kB 3*2048kB 18*4096kB = 112624kB*

# 5 页面分配

Linux提供了一系列的API来分配物理页面，这些API都是基于函数alloc_pages（）。所有的API函数都使用gfp_mask参数，这个参数决定分配器的行为。在后面，我们会详细介绍内存分配的标志GFP（Get Free Page）标志，这些标志决定内存分配器和kswapd分配和回收页面时的行为。

alloc_pagess（）等函数定义在文件include/linux/gfp.h中。

```
00304: #ifdef CONFIG_NUMA
00305: extern struct page *alloc_pages_current(gfp_t gfp_mask,
unsigned order);
00306:
00307: static inline struct page *
00308: alloc_pages(gfp_t gfp_mask, unsigned int order)
00309: {
00310:         return alloc_pages_current(gfp_mask, order);
00311: }
00312: extern struct page *alloc_pages_vma(gfp_t gfp_mask, int
order,
00313:                         struct vm_area_struct *vma, unsigned long addr,
00314:                         int node);
00315: #else
00316: #define alloc_pages(gfp_mask, order) \
00317:                 alloc_pages_node(numa_node_id(), gfp_mask, order)
00318: #define alloc_pages_vma(gfp_mask, order, vma, addr, node) \
00319:         alloc_pages(gfp_mask, order)
00320: #endif
00321: #define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)
00322: #define alloc_page_vma(gfp_mask, vma, addr)                 \
00323:         alloc_pages_vma(gfp_mask, 0, vma, addr, numa_node_id())
00324: #define alloc_page_vma_node(gfp_mask, vma, addr, node)          \
00325:         alloc_pages_vma(gfp_mask, 0, vma, addr, node)
00326:
00327: extern unsigned long ___get_free_pages(gfp_t gfp_mask,
unsigned int order);
00328: extern unsigned long get_zeroed_page(gfp_t gfp_mask);
```

## 5.1 UMA页面分配

UMA架构下alloc_pages（）最终会__alloc_pages_nodemask（）函数。与NUMA架构下使用相同的函数。

```
00316: #define alloc_pages( gfp_mask,  order) \
00317:          alloc_pages_node( numa_node_id(), gfp_mask, order)
```

```
00286: static inline struct page *alloc_pages_node( int  nid, gfp_t
gfp_mask,
00287:                                  unsigned int order)
00288: {
00289:      /* Unknown node is current node */
00290:      if (nid < 0)
00291:          nid = numa_node_id();
00292:
00293:      return __alloc_pages( gfp_mask, order, node_zonelist( nid,
gfp_mask));
00294: }
```

```
00279: static inline struct page *
00280: __alloc_pages( gfp_t  gfp_mask, unsigned int order,
00281:          struct zonelist *zonelist)
00282: {
00283:      return __alloc_pages_nodemask( gfp_mask, order,
zonelist, NULL);
00284: }
```

## 5.2 NUMA页面分配

这里我们分析NUMA架构中的页面分配，即alloc_pages（）调用alloc_pages_current（）。

当一个进程需要分配若干连续的物理页面时，可以通过alloc_pages（）来完成。因为最大页面块是$2^{MAX_ORDER-1}$，若order大于或等于MAX_ORDER，显然超出范围。MAX_ORDER定义为11，故用户向内核请求页面分配时，每次最多能请求$2^{MAX_ORDER-1}$即$2^{10}$个页面。在4K页面大小的系统中，每次**最多分配4MB的连续物理内存**。不禁有人要问，如内核需要的内存大于4MB怎么办？这时只能连续多次申请4MB内存拼成大块内存，而且检查并保证物理地址连续。

一般情况下驱动申请内存最好不要超过256KB的连续内存，因为随着系统的运行，内存页面的使用会碎片化，难以找到大于256KB的连续物理内存空间。

```
00307: static inline struct page *
00308: alloc_pages(gfp_t gfp_mask, unsigned int order)
00309: {
00310:      return alloc_pages_current(gfp_mask, order);
```

```
00311: }
```

### 5.2.1 NUMA策略与cpuset功能

内存策略就是指应用程度控制自己的内存分配。可以通过mbind和set_mempolicyl两个系统调用来设置内存策略，可以设置整个进程或一段地址空间的策略，内存策略会从父进程继承。libnuma库和其numactl小工具可以方便操作NUMA内存。

系统支持MPOL_DEFAULT、MPOL_PREFERRED、MPOL_INTERLEAVE和MPOL_BIND四种分配策略，在mm/mempolicy.c文件中实现。

表2 NUMA内存分配策略

| 策略 | 含义 |
|---|---|
| MPOL_DEFAULT | 默认策略。也就是应该从当前节点当前节点分配内存，当前节点没有空闲内存时，从最近有空闲内存的节点分配。 |
| MPOL_PREFERRED | 从指定节点上分配内存，若该节点上没有空闲内存，则其他任何一个节点都可以。 |
| MPOL_INTERLEAVE | 内存分配要覆盖所有节点。该策略通常用于共享内存区域，分配的内存覆盖所有区域用来保证不会有节点过载，同时每个节点上用的内存大小相同。 |
| MPOL_BIND | 内存分配指定在特定的节点集（即某几个节点）中。当这些节点不能提供所需要的内存时，内存分配就会失败。 |

cpuset是2.6内核版本中的一个模块，它可以让使用者将多cpu的系统划分成不同区域，每个区域包括了cpu和物理内存段。可以设置某个进程只能在特定的区域执行，而且该进程不会使用区域之外的计算资源。一般的应用，如web服务器，或则NUMA架构中的高性能运算服务器，都可以使用cpuset来提升性能。可以在内核config配置文件中，查看CONFIG_CPUSET来确定内存是否打开了CPUSET功能。

cpuset有如下特点：

- 限定一组任务所允许使用的内存node和cpu资源；
- cpuset的限定优先级高于内存策略(与node相关)和绑定(与cpu相关)；

### 5.2.2 alloc_pages_current（）

函数alloc_pages_current（）实现在文件mm/mempolicy.c中。

```
01862: struct page *alloc_pages_current(gfp_t gfp, unsigned order)
```

```
01863: {
01864:         struct mempolicy *pol = current->mempolicy;
01865:         struct page *page;
01866:
01867:         if (!pol || !in_interrupt() || (gfp & __GFP_THISNODE))
01868:                 pol = &default_policy;
01869:
01870:         get_mems_allowed();
01871:         /*
01872:          * No reference counting needed for current->mempolicy
01873:          * nor system default_policy
01874:          */
01875:         if (pol->mode == MPOL_INTERLEAVE)
01876:                 page = alloc_page_interleave(gfp, order, interleave_nodes(pol));
01877:         else
01878:                 page = __alloc_pages_nodemask(gfp, order,
01879:                                 policy_zonelist(gfp, pol, numa_node_id()),
01880:                                 policy_nodemask(gfp, pol));
01881:         put_mems_allowed();
01882:         return page;
01883: } ?     end alloc_pages_current ?
01884: EXPORT_SYMBOL(alloc_pages_current);
```

前面介绍了四种NUMA内存分配策略，当内存分配标志为置有__GFP_THISNODE，明确在当前节点上申请内存，或代码申请是在中断中，或当前进程的内存分配策略为空时（1867行）；就是用系统默认的分配策略。默认的内存分配策略为MPOL_PREFERRED。

```
00115: struct mempolicy default_policy = {
00116:         .refcnt = ATOMIC_INIT(1), /* never free it */
00117:         .mode = MPOL_PREFERRED,
00118:         .flags = MPOL_F_LOCAL,
00119: };
```

在1870行、1881行，分别调用两个函数get_mems_allowed（）和put_mems_allowed（），这两个函数与cpu set有关。

若内核打开了cpuset功能，则get_mems_allowed（）和put_mems_allowed（）两个函数功能就是增加和减少当前进程mems_allowed_change_disable的计数。若内核关闭了cpuset功能，则该两个函数实现为空。mems_allowed_change_disable的计数目的就是在内存分配过程中，避免上层更改内存分配策略。

```
00094: static inline void get_mems_allowed(void)
00095: {
00096:         current->mems_allowed_change_disable++;
```

```
00097:
00106:          smp_mb();
00107: }


00109: static inline void put_mems_allowed(void)
00110: {
00119:          smp_mb();
00120:          --ACCESS_ONCE(current->mems_allowed_change_disable);
00121: }
```

这里我们分析NUMA默认内存分配策略下的情况，即代码会执行1878行。两个参数policy_zonelist(gfp, pol, numa_node_id())和policy_nodemask(gfp, pol)，是根据NUMA策略来确定在哪些节点、哪个区域上分配内存。

接下来我们分析函数__alloc_pages_nodemask（）实现。

## 5.3 __alloc_pages_nodemask（）

### 5.3.1 内存迁移类型与lockdep

在NUMA架构中，可以将内存在节点间移动，以使页面对使用的进程而言就更好的本地性。当多个进程在一个节点集上运行，然后某个进程结束，导致内存使用不均衡，此时就需要将部分内存从一个节点移到另外一个节点，用来恢复内存分布均衡和降低NUMA延迟。页面迁移类型有5种，定义在文件include/linux/mmzone.h。

```
00038: #define MIGRATE_UNMOVABLE     0
00039: #define MIGRATE_RECLAIMABLE   1
00040: #define MIGRATE_MOVABLE       2
00041: #define MIGRATE_PCPTYPES      3 /* the number of types on the pcp lists */
00042: #define MIGRATE_RESERVE       3
00043: #define MIGRATE_ISOLATE       4 /* can't allocate from here */
00044: #define MIGRATE_TYPES         5
```

lockdep是linux内核的一个调试模块，用来检查内核互斥机制尤其是自旋锁潜在的死锁问题。 自旋锁（spin lock）由于是查询方式等待，不释放处理器，比一般的互斥机制更容易死锁， 故引入lockdep检查以下几种情况可能的死锁：

- 同一个进程递归地加锁同一把锁
- 一把锁既在中断（或中断下半部）使能的情况下执行过加锁操作，又在中断（或中断下半部）里执行过加锁操作。这样该锁有可能在锁定时由于中断发生又试图在同

一处理器上加锁；

- 加锁后导致依赖图产生成闭环，这是典型的死锁现象。

lockdep的支持，需要在内核打开配置CONFIG_LOCKDEP_SUPPORT。

### 5.3.2 __alloc_pages_nodemask（）

__alloc_pages_nodemask（）是Linux内核Buddy分配器的核心函数，源码在文件mm/page_alloc.c。

```
02190: /*
02191:  * This is the 'heart' of the zoned buddy allocator.
02192:  */
02193: struct page *
02194: __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
02195:                 struct zonelist *zonelist, nodemask_t *nodemask)
02196: {
02197:         enum zone_type high_zoneidx = gfp_zone(gfp_mask);
02198:         struct zone *preferred_zone;
02199:         struct page *page;
02200:         int migratetype = allocflags_to_migratetype(gfp_mask);
02201:
02202:         gfp_mask &= gfp_allowed_mask;
02203:
02204:         lockdep_trace_alloc(gfp_mask);
02205:
02206:         might_sleep_if(gfp_mask & __GFP_WAIT);
02207:
```

2197行：根据参数gfp_mask，找到适合的区域（区域包括ZONE_DMA、ZONE_DMA32、ZONE_NORMAL、ZONE_HIGHMEM），在这里用作给定节点数组node_zonelists[ ]的下标。

2200行：根据页面申请标志gfp_mask转换为相应的内存迁移类型（migrate type）；

2202行：更新GFP（Get Free Page）标志；

2204行：当前进程内存申请lockdep检查；

2206行：当gfp_mask中设置__GFP_WAIT时，就设置当前函数可以睡眠；

```
02208:         if (should_fail_alloc_page(gfp_mask, order))
02209:                 return NULL;
02210:
```

```
02211:        /*
02212:         * Check the zones suitable for the gfp_mask contain at least one
02213:         * valid zone. It's possible to have an empty zonelist as a result
02214:         * of GFP_THISNODE and a memoryless node
02215:         */
02216:        if (unlikely(!zonelist->_zonerefs->zone))
02217:                return NULL;
02218:
02219:        get_mems_allowed();
02220:        /* The preferred zone is used for statistics later */
02221:        first_zones_zonelist(zonelist, high_zoneidx, nodemask,
&preferred_zone);
02222:        if (!preferred_zone) {
02223:                put_mems_allowed();
02224:                return NULL;
02225:        }
02226:
```

2208~2209行：在真正尝试分配页面之前，先根据gfp_mask和order值，检查是否能够满足页面分配请求，若不满足直接返回空；

2216~2217行：检查针对gfp_mask，至少有一个有效的区域；若没有有效区域，则说明不能满足请求，返回NULL；

2219、2223行：分别为增加和减少当前进程mems_allowed_change_disable的计数；

2221行：在区域列表中，根据nodemask，找到合适的小于或等于highest_zoneidx区域，值保存在preferred_zone变量中；

假设找到了满足请求条件的区域，我们继续往下分析。

```
02227:        /* First allocation attempt */
02228:        page = get_page_from_freelist(gfp_mask| __GFP_HARDWALL,
nodemask, order,
02229:                        zonelist, high_zoneidx, ALLOC_WMARK_LOW|
ALLOC_CPUSET,
02230:                        preferred_zone, migratetype);
02231:        if (unlikely(!page))
02232:                page = __alloc_pages_slowpath(gfp_mask, order,
02233:                        zonelist, high_zoneidx, nodemask,
02234:                        preferred_zone, migratetype);
02235:        put_mems_allowed();
02236:
02237:        trace_mm_page_alloc(page, order, gfp_mask, migratetype);
02238:        return page;
02239: } ?    end __alloc_pages_nodemask ?
02240: EXPORT_SYMBOL(__alloc_pages_nodemask);
```

前面的代码都是一些基本检查工作，检查通过后，就尝试通过get_page_from_freelist（）（2228行）从区域列表中分配$2^{order}$个**物理地址连续**的页面。即首先尝试在空闲页面链表中分配页面。显然随着系统的运行，空闲页面会越来越少（如Page Cache会占用内存，即用作块设备I/O缓存），通过get_page_from_freelist（）分配内存很可能失败，此时就要调用__alloc_pages_slowpath（）函数从全局内存池中分配页面，其中的工作包括回收物理内存页面。

下面我们分别分析get_page_from_freelist（）和__alloc_pages_slowpath（）两个函数的实现。

## 5.4 get_page_from_freelist（）

### 5.4.1 区域（Zone）水准

当系统中的空用内存变低时，kswapd守护进程就会被唤醒去释放页面。若内存空闲率很低， kswapd就会同步地释放内存，有时称为直接回收（direct-reclaim）路径。

每个区域都有三个水准：pages_low，pages_min和pages_high，三个水准反映了一个区域所面临的压力。区域水准类似与水库的水位，干旱时，水库有低位告警；也可以对水库的不同低水位分不同告警级别。

```
00159: enum zone_watermarks {
00160:         WMARK_MIN,
00161:         WMARK_LOW,
00162:         WMARK_HIGH,
00163:         NR_WMARK
00164: };


00166: #define min_wmark_pages(z) (z- >watermark[WMARK_MIN])
00167: #define low_wmark_pages(z) (z- >watermark[WMARK_LOW])
00168: #define high_wmark_pages(z) (z- >watermark[WMARK_HIGH])
00169:
```

三者的关系如下图所示。

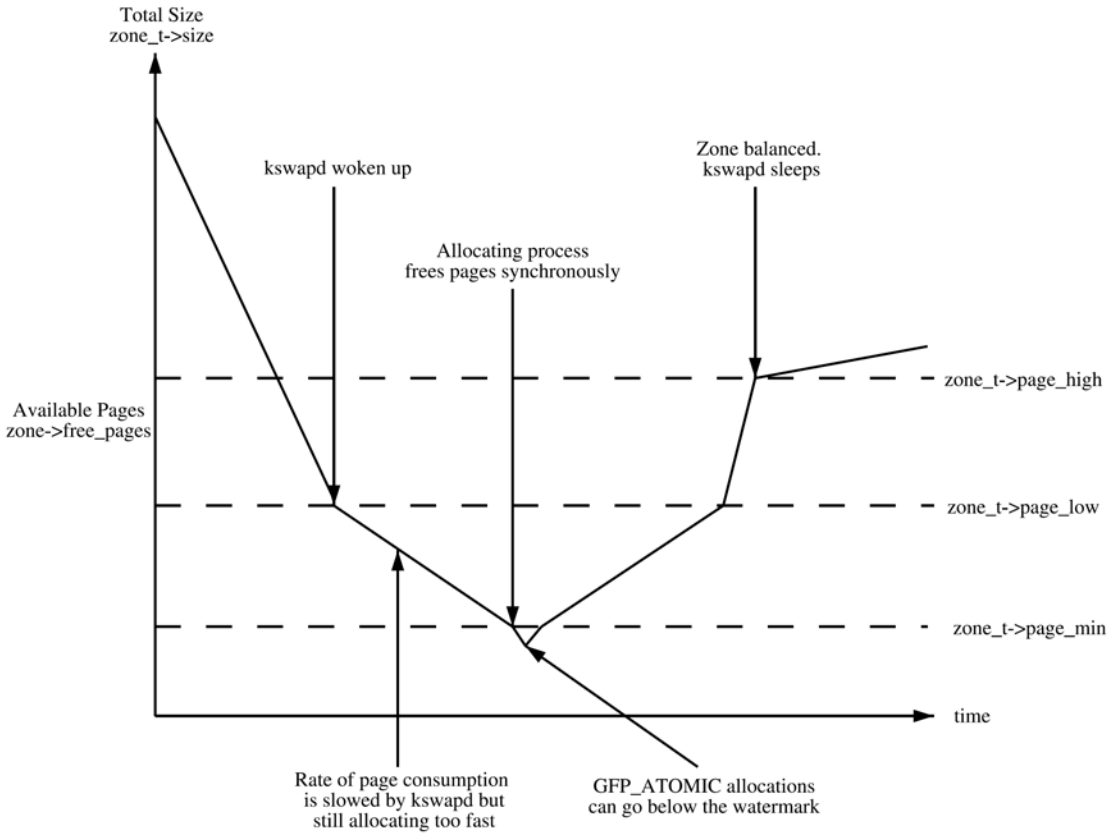


图16 区域水准图

在函数setup_per_zone_wmarks（）函数中初始化时设置page_min和pages_low的值。

```
04972: void setup_per_zone_wmarks( void)
04973: {
04974:         unsigned long pages_min = min_free_kbytes >>
                        (PAGE_SHIFT - 10);
04975:         unsigned long pages_low = extra_free_kbytes >>
                        (PAGE_SHIFT - 10);
```

在不同空闲页面的水准上，会采用不同的动作。

**pages_low**：当空闲页面达到pages_low时，buddy分配器唤醒kswapd守护进程来回收页面。默认值为pages_min的两倍；

**pages_min**：当空闲页面达到pages_min时，分配器就会唤醒kswapd以同步方式工作，有时称为直接回收（direct-reclaim）路径。

**pages_high**：唤醒kswapd进程之后，空闲页面达到pages_high时，就不会认为需要平衡区域。当达到这个水准后，kswapd就会进入休眠状态；pages_high的默认值为pages_min的三倍。

### 5.4.2 Hot-N-Cold页面

当某个物理内存页面数据在CPU Cache中，CPU访问该页数据就可以快速直接从Cache中读取，此时该页面称为**热（Hot）页面**；反之，页面不在CPU Cache中称为**冷（Cold）页面**。在多CPU系统中，每个CPU都有自己的Cache，因此冷热页面按CPU来管理，每个cpu都维持着一个冷/热页面的内存池。

struct zone结构体中成员pageset[NR_CPUS]是用来实现热 / 冷页管理。NR_CPUS并不是当前系统CPU数量，而是当前内核支持的最大CPU数量。

```
00287: struct zone {
  ...
00312: #ifdef CONFIG_NUMA
  ...
00319:      struct per_cpu_pageset   *pageset[NR_CPUS];
00320: #else
00321:      struct per_cpu_pageset   pageset[NR_CPUS];
00322: #endif
 ...
00446:  } ?    end zone ?
```

数据结构per_cpu_pageset的定义在文件include/linux/mmzone.h中。

```
00179: struct per_cpu_pageset {
00180:      struct per_cpu_pages pcp;
00181: #ifdef CONFIG_NUMA
00182:      s8 expire;
00183: #endif
00184: #ifdef CONFIG_SMP
00185:      s8 stat_threshold;
00186:      s8 vm_stat_diff[NR_VM_ZONE_STAT_ITEMS];
00187: #endif
00188: } ____cacheline_aligned_in_smp;
```

有用的数据结构per_cpu_pages定义也在mmzone.h。

```
00041: #define MIGRATE_PCPTYPES       3 /* the number of types on the pcp lists */

00170: struct per_cpu_pages {
00171:      int count;      /* number of pages in the list */
00172:      int high;       /* high watermark, emptying needed */
00173:      int batch;      /* chunk size for buddy add/remove */
```

```
00174:
00175:          /* Lists of pages, one per migrate type stored on the pcp-lists */
00176:          struct list_head lists[MIGRATE_PCPTYPES];
00177: };
```

count为链表中的页面数量；high为水准，若count > high，则表示链表中的页面太多了，需要清除一部分；batch针对buddy算法添加/删除的块大小；list为页面链表头。

下图为两颗CPU系统中，冷热页面统计信息及数据结构。

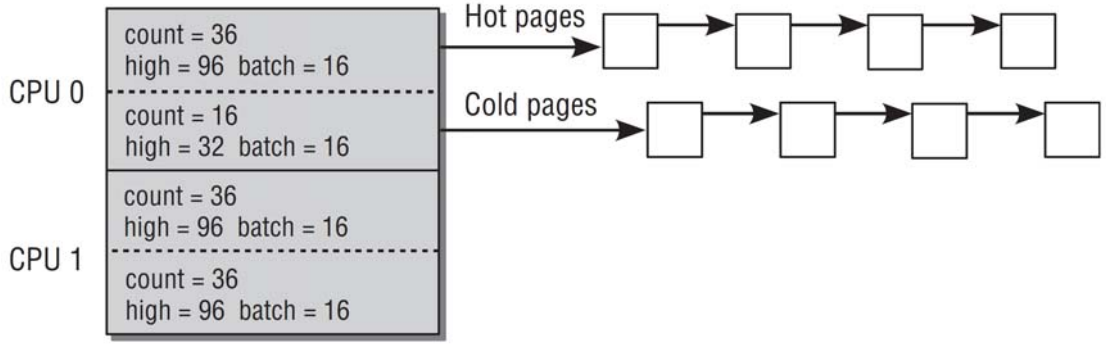


图17 双CPU系统中的Per CPU Cache

我们可以通过echo m > /proc/sysrq-trigger来观察当前系统中的冷/热页面情况。下面是一个实例。

```
Node 1 DMA per-cpu:
CPU    0: hi:    0, btch:   1 usd:   0
CPU    1: hi:    0, btch:   1 usd:   0
Node 1 DMA32 per-cpu:
CPU    0: hi:  186, btch:  31 usd: 212
CPU    1: hi:  186, btch:  31 usd:   0
Node 1 Normal per-cpu:
CPU    0: hi:  186, btch:  31 usd: 138
CPU    1: hi:  186, btch:  31 usd:  52
```

### 5.4.3 get_page_from_freelist（）

get_page_from_freelist（）函数也在文件mm/page_alloc.c中。函数主体是1666行的for_each_zone_zonelist_nodemask（）循环语句，在nodemask确定的节点中所有区域，找到满足请求数量的空闲页面。

```
01643: /*
01644:  * get_page_from_freelist goes through the zonelist trying to allocate
01645:  * a page.
01646:  */
01647: static struct page *
01648: get_page_from_freelist(gfp_t gfp_mask, nodemask_t
```

```
01648:          *nodemask, unsigned int order,
01649:             struct zonelist *zonelist, int high_zoneidx, int alloc_flags,
01650:             struct zone *preferred_zone, int migratetype)
01651: {
01652:         struct zoneref *z;
01653:         struct page *page = NULL;
01654:         int classzone_idx;
01655:         struct zone *zone;
01656:         nodemask_t *allowednodes = NULL;/* zonelist_cache
approximation */
01657:         int zlc_active = 0;         /* set if using zonelist_cache */
01658:         int did_zlc_setup = 0;          /* just call zlc_setup() one time */
01659:
01660:         classzone_idx = zone_idx(preferred_zone);
01661: zonelist_scan:
01662:         /*
01663:          * Scan zonelist, looking for a zone with enough free.
01664:          * See also cpuset_zone_allowed() comment in kernel/cpuset.c.
01665:          */
01666:         for_each_zone_zonelist_nodemask(zone, z, zonelist,
01667:                                         high_zoneidx, nodemask) {
01668:             if (NUMA_BUILD && zlc_active &&
01669:                 !zlc_zone_worth_trying(zonelist, z, allowednodes))
01670:                     continue;
01671:             if ((alloc_flags & ALLOC_CPUSET) &&
01672:                 !cpuset_zone_allowed_softwall(zone, gfp_mask))
01673:                     goto ↓try_next_zone;
01674:
01675:             BUILD_BUG_ON(ALLOC_NO_WATERMARKS <
NR_WMARK);
01676:             if (!(alloc_flags & ALLOC_NO_WATERMARKS)) {
01677:                 unsigned long mark;
01678:                 int ret;
01680:                 mark = zone->watermark[alloc_flags &
ALLOC_WMARK_MASK];
01681:                 if (zone_watermark_ok(zone, order, mark,
01682:                             classzone_idx, alloc_flags))
01683:                     goto ↓try_this_zone;
01684:
01685:                 if (zone_reclaim_mode == 0)
01686:                     goto ↓this_zone_full;
01687:
01688:                 ret = zone_reclaim(zone, gfp_mask, order);
01689:                 switch (ret) {
01690:                 case ZONE_RECLAIM_NOSCAN:
01691:                     /* did not scan */
01692:                     goto ↓try_next_zone;
01693:                 case ZONE_RECLAIM_FULL:
01694:                     /* scanned but unreclaimable */
01695:                     goto ↓this_zone_full;
01696:                 default:
01697:                     /* did we reclaim enough */
```

```
01698:                  if (! zone_watermark_ok(zone, order, mark,
01699:                              classzone_idx, alloc_flags))
01700:                      goto ↓this_zone_full;
01701:              }
01702:          } ?    end if !(alloc_flags&ALLOC_N... ?
01703:
```

在真正查找空闲页面之前，先做一些基本检查。若内核打开了NUMA，就通过zlc_zone_worth_trying（）函数来快速检查该区域是否值得去更进一步查找空闲内存；若该区域不值得查找，则到下一个区域执行同样的动作（1669~1670行）。

接下来就要检查区域水准，看当前区域是否满足水准要求（1676~1702行）。

（1）若本区域水准满足要求，则直接尝试在本区域分配页面（1681~1683行）；

（2）若本区域水准不能满足要求，且区域zone_reclaim_mode的值为0（1685~1686行），则跳转到this_zone_full；

注：zone_reclaim_mode的值可以通过/proc/sys/vm/zone_reclaim_mode更改，默认值为0。

（3）若上面两个条件都不满足，则要通过调用zone_reclaim（）尝试回收本区域内存（1681~1701行）。若返回值为区域未扫描ZONE_RECLAIM_NOSCAN，则跳过这个区域尝试下一个区域（1690行）；若返回值为ZONE_RECLAIM_FULL没法回收，就标记该区域已满，下次别人就不要再浪费时间扫描这个区域了（1693行）； 若成功回收了部分内存，则重新检查区域水准。

我们继续看代码，前面检查了区域水准，且认为当前区域能够有足够空闲页面满足请求，于是调用buffered_rmqueue（）函数来从区域分配页面。

```
01704: try_this_zone:
01705:          page = buffered_rmqueue(preferred_zone, zone, order,
01706:                              gfp_mask, migratetype);
01707:          if (page)
01708:              break;
01709: this_zone_full:
01710:          if (NUMA_BUILD)
01711:              zlc_mark_zone_full(zonelist, z);
01712: try_next_zone:
01713:          if (NUMA_BUILD && ! did_zlc_setup && nr_online_nodes > 1)
                {
01714:              /*
01715:               * we do zlc_setup after the first zone is tried but only
01716:               * if there are multiple nodes make it worthwhile
01717:               */
01718:              allowednodes = zlc_setup(zonelist, alloc_flags);
01719:              zlc_active = 1;
```

```
01720:                  did_zlc_setup = 1;
01721:              }
01722:          }
01723:
01724:          if (unlikely(NUMA_BUILD && page == NULL && zlc_active)) {
01725:              /* Disable zlc cache for second zonelist scan */
01726:              zlc_active = 0;
01727:              goto ↑zonelist_scan;
01728:          }
01729:          return page;
01730: } ?    end get_page_from_freelist ?
```

在下一小节详细分析buffered_rmqueue（）函数实现。

1. buffered_rmqueue（）

buffered_rmqueue（）函数源码在mm/page_alloc.c中。

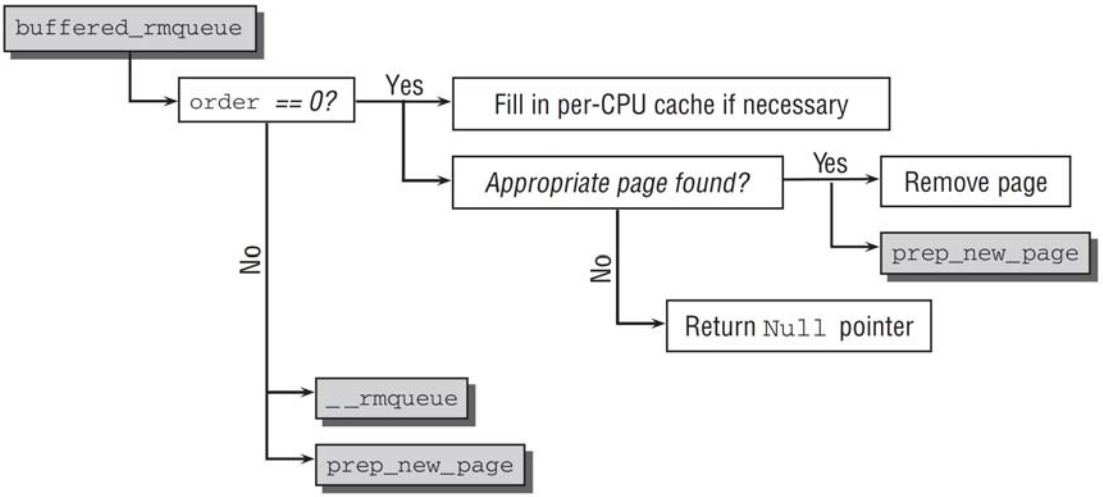


图18 buffered_rmqueue（）函数调用关系

```
01292: static inline
01293: struct page *buffered_rmqueue(struct zone *preferred_zone,
01294:                  struct zone *zone, int order, gfp_t gfp_flags,
01295:                  int migratetype)
01296: {
01297:          unsigned long flags;
01298:          struct page *page;
01299:          int cold = ! ! (gfp_flags & __GFP_COLD);
01300:          int cpu;
01301:
01302: again:
01303:          cpu   = get_cpu();
01304:          if (likely(order == 0)) {
01305:              struct per_cpu_pages *pcp;
```

```
01306:          struct list_head *list;
01307:
01308:          pcp = &zone_pcp(zone, cpu)->pcp;
01309:          list = &pcp->lists[migratetype];
01310:          local_irq_save(flags);
01311:          if (list_empty(list)) {
01312:                  pcp->count += rmqueue_bulk(zone, 0,
01313:                                  pcp->batch, list,
01314:                                  migratetype, cold);
01315:                  if (unlikely(list_empty(list)))
01316:                          goto ↓failed;
01317:          }
01318:
01319:          if (cold)
01320:                  page = list_entry(list->prev, struct page, lru);
01321:          else
01322:                  page = list_entry(list->next, struct page, lru);
01323:
01324:          list_del(&page->lru);
01325:          pcp->count--;
```

首先判断是否只请求分配一个页面（1304行）。每个CPU都维护着一个迁移页面链表，有以下三种。

```
00038: #define MIGRATE_UNMOVABLE     0
00039: #define MIGRATE_RECLAIMABLE   1
00040: #define MIGRATE_MOVABLE       2
```

在指定的migratetype迁移页面链表上查找是否有空闲页面，若有空闲页面（1308~1311行），就分配一个页面，同时更新迁移链表中的计数。另外根据是否指定申请冷/热页面，返回相应类型页面（1319~1322行）。

若申请的内存不止一个页面，则执行1326行后面的代码。通常情况下，很少使用__GFP_NOFAIL标志，即允许页面分配失败。

```
01326:      } else {
01327:          if (unlikely(gfp_flags & __GFP_NOFAIL)) {
01338:              WARN_ON_ONCE(order > 1);
01339:          }
01340:          spin_lock_irqsave(&zone->lock, flags);
01341:          page = __rmqueue(zone, order, migratetype);
01342:          spin_unlock(&zone->lock);
01343:          if (!page)
01344:              goto ↓failed;
01345:          __mod_zone_page_state(zone, NR_FREE_PAGES, -(1 <<
order));
01346:      } ?     end else ?
```

接下来就是通过__rmqueue（）来从zone区域中分配$2^{order}$个物理地址连续的页面，在执行页面分配前，要对区域上锁，防止其他CPU也操作该区域（1340行），分配页面完成后，再释放zone->lock锁。分析到这里，我们发现前面折腾了那么久，还没有执行真正分配物理页面。

```
01348:      __count_zone_vm_events(PGALLOC, zone, 1 << order);
01349:      zone_statistics(preferred_zone, zone, gfp_flags);
01350:      local_irq_restore(flags);
01351:      put_cpu();
01352:
01353:      VM_BUG_ON(bad_range(zone, page));
01354:      if (prep_new_page(page, order, gfp_flags))
01355:          goto ↑again;
01356:      return page;
01357:
01358: failed:
01359:      local_irq_restore(flags);
01360:      put_cpu();
01361:      return NULL;
01362: } ?   end buffered_rmqueue ?
```

1348~1349行，是对区域的一些统计信息。页面申请成功后，直接返回页面块中的第一个页面地址（1356行）。

### 5.4.4 __rmqueue（）

__rmqueue（）函数才是从相应zone中取得多页面的操作,它是整个页面分配过程的真正分配页面的核心代码。源码同样在文件mm/page_alloc.c中。

在NUMA系统中，内核总是首先尝试从进程所在CPU的节点上分配内存，这样内存性能更好。但这种分配策略并不能保证每次都成功，对于不能从本节点分配内存的情形，每个节点都提供一个**fallback链表**。该链表中包含其他节点及区域，可以用作内存分配的替代选择。

首先通过__rmqueue_smallest（）函数（982行），尝试找到恰好满足给定order大小、migratetype类型的页面块。

若在区域链表中，找不到给你order大小和migraterype类型的页面块，就要调用__rmqueue_fallback（）从fallback链表中分配指定order和migrate页面块。

```
00972: /*
00973:  * Do the hard work of removing an element from the buddy
allocator.
00974:  * Call me with the zone->lock already held.
00975:  */
00976: static struct page *__rmqueue(struct zone *zone, unsigned int
order,
00977:                                 int migratetype)
00978: {
00979:         struct page *page;
00980:
00981: retry_reserve:
00982:         page = __rmqueue_smallest(zone, order, migratetype);
00983:
00984:         if (unlikely(! page) && migratetype ! = MIGRATE_RESERVE) {
00985:                 page = __rmqueue_fallback(zone, order, migratetype);
00986:
00987:                 /*
00988:                  * Use MIGRATE_RESERVE rather than fail an allocation.
goto
00989:                  * is used because __rmqueue_smallest is an inline function
00990:                  * and we want just one call site
00991:                  */
00992:                 if (! page) {
00993:                         migratetype = MIGRATE_RESERVE;
00994:                         goto ↑retry_reserve;
00995:                 }
00996:         }
00997:
00998:         trace_mm_page_alloc_zone_locked(page, order, migratetype);
00999:         return page;
01000: } ?    end __rmqueue ?
```

下面我们分别分析__rmqueue_smallest（）和__rmqueue_fallback（）函数。

1. __rmqueue_smallest（）

__rmqueue_smallest（）源码同样在文件mm/page_alloc.c中。

分配页面块步骤如下：

（1）从order开始的空闲块链表上开始找空闲的块；

（2）若当前order上有空闲页面块，则摘除空闲块，然后跳转到步骤（4）；

（3）若当前order上没有有空闲页面块，则order=order+1，跳转需要到上一级查找是否有空闲块；跳转到步骤（2）执行；若order > MAX_ORDER-1，则跳转到步骤（5）；

（4）若分配页面块所在order大于请求值，还要将剩余部分页面块放在更低的order链

表上（802行expand（）函数），页面分配成功返回；

（5）页面分配失败返回。

```
00779: /*
00780:  * Go through the free lists for the given migratetype and remove
00781:  * the smallest available page from the freelists
00782:  */
00783: static inline
00784: struct page *___rmqueue_smallest(struct zone *zone,
unsigned int order,
00785:                                 int migratetype)
00786: {
00787:         unsigned int current_order;
00788:         struct free_area * area;
00789:         struct page *page;
00790:
00791:         /* Find a page of the appropriate size in the preferred list */
00792:         for (current_order = order; current_order < MAX_ORDER;
++current_order) {
00793:                 area = &(zone->free_area[current_order]);
00794:                 if (list_empty(&area->free_list[migratetype]))
00795:                         continue;
00796:
00797:                 page = list_entry(area->free_list[migratetype].next,
00798:                                         struct page, lru);
00799:                 list_del(&page->lru);
00800:                 rmv_page_order(page);
00801:                 area->nr_free--;
00802:                 expand(zone, page, order, current_order, area, migratetype);
00803:                 return page;
00804:         }
00805:
00806:         return NULL;
00807: } ?    end __rmqueue_smallest ?
```

2. __rmqueue_fallback（）

__rmqueue_fallback（）函数的分配页面块过程和__rmqueue_smallest（）类似，区别在于在fallback链表中进行，而不是本节点上的zone区域。这里不作详细解释。

```
00902: /* Remove an element from the buddy allocator from the fallback list
*/
00903: static inline struct page *
00904: ___rmqueue_fallback(struct zone *zone, int order, int
start_migratetype)
00905: {
00906:         struct free_area * area;
00907:         int current_order;
```

```
00908:      struct page *page;
00909:      int migratetype, i;
00910:
00911:      /* Find the largest possible block of pages in the other list */
00912:      for (current_order = MAX_ORDER-1; current_order >= order;
00913:                                  --current_order) {
00914:          for (i = 0; i < MIGRATE_TYPES - 1; i++) {
00915:              migratetype = fallbacks[start_migratetype][i];
00916:
00917:              /* MIGRATE_RESERVE handled later if necessary */
00918:              if (migratetype == MIGRATE_RESERVE)
00919:                  continue;
00920:
00921:              area = &(zone->free_area[current_order]);
00922:              if (list_empty(&area->free_list[migratetype]))
00923:                  continue;
00924:
00925:              page = list_entry(area->free_list[migratetype].next,
00926:                      struct page, lru);
00927:              area->nr_free--;
00928:
00929:              /*
00930:               * If breaking a large block of pages, move all free
00931:               * pages to the preferred allocation list. If falling
00932:               * back for a reclaimable kernel allocation, be more
00933:               * agressive about taking ownership of free pages
00934:               */
00935:              if (unlikely(current_order >= (pageblock_order >> 1)) ||
00936:                      start_migratetype ==
MIGRATE_RECLAIMABLE ||
00937:                      page_group_by_mobility_disabled) {
00938:                  unsigned long pages;
00939:                  pages = move_freepages_block(zone, page,
00940:                                  start_migratetype);
00941:
00942:                  /* Claim the whole block if over half of it is free */
00943:                  if (pages >= (1 << (pageblock_order-1)) ||
00944:                              page_group_by_mobility_disabled)
00945:                      set_pageblock_migratetype(page,
00946:                                  start_migratetype);
00947:
00948:                  migratetype = start_migratetype;
00949:              }
00950:
00951:  /* Remove the page from the freelists */
00952:              list_del(&page->lru);
00953:              rmv_page_order(page);
00954:
00955:              /* Take ownership for orders >= pageblock_order */
00956:              if (current_order >= pageblock_order)
00957:                  change_pageblock_range(page, current_order,
```

```
00958:                                  start_migratetype);
00959:
00960:                  expand(zone, page, order, current_order, area, migratetype);
00961:
00962:                  trace_mm_page_alloc_extfrag(page, order, current_order,
00963:                          start_migratetype, migratetype);
00964:
00965:                  return page;
00966:          } ?     end for i=0;i<MIGRATE_TYPES-1...    ?
00967:      } ?     end for current_order=MAX_ORD...    ?
00968:
00969:      return NULL;
00970: } ?     end __rmqueue_fallback ?
```

3. expand（）

expand（）函数的主要作用是将空闲链表上的页面块分配一部分后，再将空闲部分放到区域更低order空闲页面块链表中。

```
00722: static inline void expand(struct zone *zone, struct page *page,
00723:      int low, int high, struct free_area *area,
00724:      int migratetype)
00725: {
00726:      unsigned long size = 1 << high;
00727:
00728:      while (high > low) {
00729:          area--;
00730:          high--;
00731:          size >>= 1;
00732:          VM_BUG_ON(bad_range(zone, &page[size]));
00733:          list_add(&page[size].lru, &area-
>free_list[migratetype]);
00734:          area->nr_free++;
00735:          set_page_order(&page[size], high);
00736:      }
00737: }
```

调用参数表中的low对应于表示所属页面块大小的order，而high则对应于表示当前空闲区队列（也就是从中得到满足要求的页面块的队列）的curr_order。当两者相符时，从782行开始的while循环就被跳过了。若是分配到的页面块大于所需的大小（不可能小于所需的大小），那就将该页面块链入低一挡（小的order），也就是物理块大小减半的空闲队列中去。然后从该物理块中切去一半，而以其后半部作为一个新的物理块，而后开始下一轮循环，也就是处理更低一挡的空闲页面块链表。这样，最后必有high与low两者相等，也就是实际剩下的物理块与要求恰好相等的时候，循环结束了。

## 5.5 __alloc_pages_slowpath（）

显然随着系统的运行，空闲页面会越来越少（如Page Cache会占用内存，即用作块设备I/O缓存），通过get_page_from_freelist（）分配内存很可能失败，此时就要调用__alloc_pages_slowpath（）函数，函数源码仍在文件mm/page_alloc.c。从函数名就可以看出，在这里分配内存页面过程相对比较慢。

```
02024: static inline struct page *
02025:  __alloc_pages_slowpath( gfp_t gfp_mask, unsigned int order,
02026:         struct zonelist *zonelist, enum zone_type high_zoneidx,
02027:         nodemask_t *nodemask, struct zone *preferred_zone,
02028:         int migratetype)
02029: {
02030:         const gfp_t wait = gfp_mask & __GFP_WAIT;
02031:         struct page *page = NULL;
02032:         int alloc_flags;
02033:         unsigned long pages_reclaimed = 0;
02034:         unsigned long did_some_progress;
02035:         struct task_struct *p = current;
02036:         bool sync_migration = false;
02037:
02038:         /*
02039:          * In the slowpath, we sanity check order to avoid ever trying to
02040:          * reclaim >= MAX_ORDER areas which will never succeed. Callers may
02041:          * be using allocators in order of preference for an area that is
02042:          * too large.
02043:          */
02044:         if (order >= MAX_ORDER) {
02045:                 WARN_ON_ONCE(!(gfp_mask & __GFP_NOWARN));
02046:                 return NULL;
02047:         }
02048:
02057:         if (NUMA_BUILD && (gfp_mask & GFP_THISNODE) == GFP_THISNODE)
02058:                 goto ↓nopage;
02059:
02060: restart:
02061:         if (!(gfp_mask & __GFP_NO_KSWAPD))
02062:                 wake_all_kswapd(order, zonelist, high_zoneidx);
02063:
02065:          * OK, we're below the kswapd watermark and have kicked background
02066:          * reclaim. Now things get more complex, so set up alloc_flags according
02067:          * to how we want to proceed.
```

```
02068:          */
02069:         alloc_flags = gfp_to_alloc_flags(gfp_mask);
02070:
02071:         /* This is the last chance, in general, before the goto nopage. */
02072:         page = get_page_from_freelist(gfp_mask, nodemask,
02073:                         order, zonelist, high_zoneidx, alloc_flags &
~ALLOC_NO_WATERMARKS,
02074:                         preferred_zone, migratetype);
02075:         if (page)
02076:                 goto ↓got_pg;
02077:
```

在尝试分配内存之前，仍然要进行一些必要的检查（2044~2058行）。若gfp_mask中，没有指定不让调用kswapd内核线程回收内存，则唤醒kswapd线程在后台回收内存。此时，仍然尝试一次调用get_page_from_freelist（）来快速分配内存，若仍然没有分配到内存，此时就要继续往下走。

```
02078: rebalance:
02079:         /* Allocate without watermarks if the context allows */
02080:         if (alloc_flags & ALLOC_NO_WATERMARKS) {
02081:                 page = __alloc_pages_high_priority(gfp_mask, order,
02082:                                 zonelist, high_zoneidx, nodemask,
02083:                                 preferred_zone, migratetype);
02084:                 if (page)
02085:                         goto ↓got_pg;
02086:         }
02087:
02088:         /* Atomic allocations - we can't balance anything */
02089:         if (! wait)
02090:                 goto ↓nopage;
02091:
02092:         /* Avoid recursion of direct reclaim */
02093:         if (p- >flags & PF_MEMALLOC)
02094:                 goto ↓nopage;
02095:
02096:         /* Avoid allocations with no watermarks from looping endlessly */
02097:         if (test_thread_flag(TIF_MEMDIE) && !(gfp_mask &
__GFP_NOFAIL))
02098:                 goto ↓nopage;
02099:
```

通过gfp_mask标志，转换成对应的alloc_flags（2069行），若分配标志中没有ALLOC_NO_WATERMARKS标志，则表明这是最高优先级的内存申请，直接调用__alloc_pages_high_priority（）分配内存。__alloc_pages_high_priority（）仅是多函数

get_page_from_freelist（）作了一层封装，alloc_flags参数设置为ALLOC_NO_WATERMARKS。

前面尝试了很多次，仍然失败的话，说明内存申请困难，就像筹钱也来越困难，手里的钱快没了:-( 。

```
02100:      /*
02101:       * Try direct compaction. The first pass is asynchronous. Subsequent
02102:       * attempts after direct reclaim are synchronous
02103:       */
02104:      page = __alloc_pages_direct_compact(gfp_mask, order,
02105:                          zonelist, high_zoneidx,
02106:                          nodemask,
02107:                          alloc_flags, preferred_zone,
02108:                          migratetype, &did_some_progress,
02109:                          sync_migration);
02110:      if (page)
02111:          goto ↓got_pg;
02112:      sync_migration = true;
02113:
```

现在还没到最困难的时候，先调用__alloc_pages_direct_compact（）函数来分配内存（2104~2109行），该函数是异步方式执行，主要工作是将空闲页面链表中的小页面块合并成大页面表（如将两个order为2的页面块合并成1个order为3的页面块），再分配页面块。

若合并小页面块也不能成功，那只能做最大力度的尝试了，以同步方式分配内存（2115~2119行）。__alloc_pages_direct_reclaim（）的工作就是先通过try_to_free_pages（）回收一些最近很少用的和page cache中的页面，以便在物理内存中腾出更多的空间。接着，内核会再次调用get_page_from_freelist（）尝试分配内存。

```
02114:      /* Try direct reclaim and then allocating */
02115:      page = __alloc_pages_direct_reclaim(gfp_mask, order,
02116:                          zonelist, high_zoneidx,
02117:                          nodemask,
02118:                          alloc_flags, preferred_zone,
02119:                          migratetype, &did_some_progress);
02120:      if (page)
02121:          goto ↓got_pg;
02122:
```

如果内核进行了上述的回收和重新分配的过程后，仍未分配成功，即

did_some_progress为0，那么此时内核不的不考虑是否发生了OOM(Out of Memory)。若gfp_mask允许VFS I/O操作且允许重新尝试（2128行）的情况下，我们还可以继续尝试。

检查是否设置oom_killer_disabled，若不允许杀死进程，直接跳转到nopage。否则就调用__alloc_pages_may_oom（）分配内存，此时再失败的话，就要调用out_of_memory（）杀死申请内存最多的进程（可能杀死多个进程）。并且跳转到restart处，重新进行内存分配。

```
02123:          /*
02124:           * If we failed to make any progress reclaiming, then we are
02125:           * running out of options and have to consider going OOM
02126:           */
02127:          if (!did_some_progress) {
02128:                  if ((gfp_mask & __GFP_FS) && !(gfp_mask &
__GFP_NORETRY)) {
02129:                          if (oom_killer_disabled)
02130:                                  goto ↓nopage;
02131:                          page = __alloc_pages_may_oom(gfp_mask, order,
02132:                                          zonelist, high_zoneidx,
02133:                                          nodemask, preferred_zone,
02134:                                          migratetype);
02135:                          if (page)
02136:                                  goto ↓got_pg;
02137:
02144:                          if (order > PAGE_ALLOC_COSTLY_ORDER &&
02145:                                          !(gfp_mask & __GFP_NOFAIL))
02146:                                  goto ↓nopage;
02147:
02148:                          goto ↑restart;
02149:                  } ?    end if (gfp_mask&____GFP_FS)&&... ?
02150:          } ?    end if !did_some_progress ?
02151:
```

此时再次判断是否要重新进行一次内存申请。如果有这个必要，那么等待写操作完成后再次跳到rebalance处重试（2154~2157行）。

```
02152:          /* Check if we should retry the allocation */
02153:          pages_reclaimed += did_some_progress;
02154:          if (should_alloc_retry(gfp_mask, order, pages_reclaimed)) {
02155:                  /* Too much pressure, back off a bit at let reclaimers do work */
02156:                  wait_iff_congested(preferred_zone, BLK_RW_ASYNC,
HZ/50);
02157:                  goto ↑rebalance;
02158:          }
02159:
```

页面块分配函数结束时候有两种情况，第一种情形分配失败，并没有得到所需页面块，于是打印一些内存分配失败的信息，并打印当前栈信息（2161~2181行）。

另一种情形是分配页面块成功，那么直接返回页面块的第一页page结构（2183~2186行）。

```
02160: nopage:
02161:         if (!(gfp_mask & __GFP_NOWARN) && printk_ratelimit()) {
02162:                 unsigned int filter = SHOW_MEM_FILTER_NODES;
02163:
02164:                 /*
02165:                  * This documents exceptions given to allocations in certain
02166:                  * contexts that are allowed to allocate outside current's set
02167:                  * of allowed nodes.
02168:                  */
02169:                 if (!(gfp_mask & __GFP_NOMEMALLOC))
02170:                         if (test_thread_flag(TIF_MEMDIE) ||
02171:                             (current->flags & (PF_MEMALLOC | PF_EXITING)))
02172:                                 filter &= ~SHOW_MEM_FILTER_NODES;
02173:                 if (in_interrupt() || !wait)
02174:                         filter &= ~SHOW_MEM_FILTER_NODES;
02175:
02176:                 pr_warning("%s: page allocation failure. order:%d,
mode:0x%x\n",
02177:                         p->comm, order, gfp_mask);
02178:                 dump_stack();
02179:                 if (!should_suppress_show_mem())
02180:                         show_mem(filter);
02181:         }
02182:         return page;
02183: got_pg:
02184:         if (kmemcheck_enabled)
02185:                 kmemcheck_pagealloc_alloc(page, order, gfp_mask);
02186:         return page;
02187:
02188: } ?    end ____alloc_pages_slowpath ?
02189:
```

### 5.5.1 __alloc_pages_direct_compact（）

__alloc_pages_direct_compact（）函数主要工作是合并小页面块成大页面块，再分配页面块。内核默认情况下是打开CONFIG_COMPACTION选项的。

函数主要两个函数try_to_compact_pages（）合并小页面块（1846~1847行），然后再次调用get_page_from_freelist（）分配页面块（1855~1858行）。这里不再详细分析相关函数实现。

```
01830: #ifdef CONFIG_COMPACTION
01831: /* Try memory compaction for high-order allocations before reclaim */
01832: static struct page *
01833: __alloc_pages_direct_compact(gfp_t gfp_mask,
unsigned int order,
01834:         struct zonelist *zonelist, enum zone_type high_zoneidx,
01835:         nodemask_t *nodemask, int alloc_flags, struct zone
*preferred_zone,
01836:         int migratetype, unsigned long *did_some_progress,
01837:         bool sync_migration)
01838: {
01839:         struct page *page;
01840:         struct task_struct *p = current;
01841:
01842:         if (!order || compaction_deferred(preferred_zone))
01843:                 return NULL;
01844:
01845:         p->flags |= PF_MEMALLOC;
01846:         *did_some_progress = try_to_compact_pages(zonelist,
order, gfp_mask,
01847:                                         nodemask, sync_migration);
01848:         p->flags &= ~PF_MEMALLOC;
01849:         if (*did_some_progress != COMPACT_SKIPPED) {
01850:
01851:                 /* Page migration frees to the PCP lists but we want merging */
01852:                 drain_pages(get_cpu());
01853:                 put_cpu();
01854:
01855:                 page = get_page_from_freelist(gfp_mask, nodemask,
01856:                                 order, zonelist, high_zoneidx,
01857:                                 alloc_flags, preferred_zone,
01858:                                 migratetype);
01859:                 if (page) {
01860:                         preferred_zone->compact_considered = 0;
01861:                         preferred_zone->compact_defer_shift = 0;
01862:                         count_vm_event(COMPACTSUCCESS);
01863:                         return page;
01864:                 }
01865:
01866:                 /*
01867:                  * It's bad if compaction run occurs and fails.
01868:                  * The most likely reason is that pages exist,
01869:                  * but not enough to satisfy watermarks.
```

```
01870:              */
01871:             count_vm_event(COMPACTFAIL);
01872:             defer_compaction(preferred_zone);
01873:
01874:             cond_resched();
01875:         } ?    end if *did_some_progress! =C... ?
01876:
01877:         return NULL;
01878: } ?    end  ____alloc_pages_direct_compact ?
```

### 5.5.2 __alloc_pages_direct_reclaim（）

函数__alloc_pages_direct_reclaim（）的功能就是先通过try_to_free_pages（）回收页面，释放一些内存（1912行）。接着，内核会再次调用get_page_from_freelist（）尝试分配内存（1927~1930行）。

```
01891: / * The really slow allocator path where we enter direct reclaim */
01892: static inline struct page *
01893: __alloc_pages_direct_reclaim( gfp_t gfp_mask,
unsigned int order,
01894:         struct zonelist *zonelist, enum zone_type high_zoneidx,
01895:         nodemask_t *nodemask, int alloc_flags, struct zone
*preferred_zone,
01896:         int migratetype, unsigned long *did_some_progress)
01897: {
01898:         struct page *page = NULL;
01899:         struct reclaim_state reclaim_state;
01900:         struct task_struct *p = current;
01901:         bool drained = false;
01902:
01903:         cond_resched();
01904:
01905:         / * We now go into synchronous reclaim */
01906:         cpuset_memory_pressure_bump();
01907:         p- >flags | = PF_MEMALLOC;
01908:         lockdep_set_current_reclaim_state( gfp_mask);
01909:         reclaim_state.reclaimed_slab = 0;
01910:         p- >reclaim_state = &reclaim_state;
01911:
01912:         *did_some_progress = try_to_free_pages( zonelist, order,
gfp_mask,
01912: nodemask);
01913:
01914:         p- >reclaim_state = NULL;
01915:         lockdep_clear_current_reclaim_state();
01916:         p- >flags &= ~PF_MEMALLOC;
```

```
01917:
01918:          cond_resched();
01919:
01920:          if (order != 0)
01921:                  drain_all_pages();
01922:
01923:          if (unlikely(!(*did_some_progress)))
01924:                  return NULL;
01925:
01926: retry:
01927:          page = get_page_from_freelist(gfp_mask, nodemask,
order,
01928:                                        zonelist, high_zoneidx,
01929:                                        alloc_flags, preferred_zone,
01930:                                        migratetype);
01931:
01932:          /*
01933:           * If an allocation failed after direct reclaim, it could be because
01934:           * pages are pinned on the per-cpu lists. Drain them and try again
01935:           */
01936:          if (!page && !drained) {
01937:                  drain_all_pages();
01938:                  drained = true;
01939:                  goto ↑retry;
01940:          }
01941:
01942:          return page;
01943: } ?    end ____alloc_pages_direct_reclaim ?
```

# 6 影响页面分配行为的GFP标志

在前面内存页面分配代码分析过程中，必然用到**gfp_mask**参数，这个参数是贯穿于整个虚拟内存的一个概念。GFP（Get Free Page）标志，这些标志决定内存分配器和kswapd分配和回收页面时的行为，也就是指定在哪些区域申请内存、申请什么样的内存、用作什么用途等。如，一个中断处理程序不能进入睡眠，则它就不能设置为__GRP_WAIT标志，因为该标志表明调用者可以进入睡眠状态。

所有GFP标志定义在文件include/linux/gfp.h。

```
00040: #define __GFP_WAIT ((__force gfp_t)0x10u) /* Can wait and reschedule? */
00041: #define __GFP_HIGH ((__force gfp_t)0x20u) /* Should access emergency
pools? */
00042: #define __GFP_IO   ((__force gfp_t)0x40u) /* Can start physical IO? */
00043: #define __GFP_FS   ((__force gfp_t)0x80u) /* Can call down to low-level
FS? */
00044: #define __GFP_COLD ((__force gfp_t)0x100u) /* Cache-cold page required
```

```
*/
00045: #define ___GFP_NOWARN     ((__force gfp_t)0x200u)

00046: #define ___GFP_REPEAT     ((__force gfp_t)0x400u) /* See above */

00047: #define ___GFP_NOFAIL     ((__force gfp_t)0x800u) /* See above */
00048: #define ___GFP_NORETRY    ((__force gfp_t)0x1000u)/* See above */
00049: #define ___GFP_COMP ((__force gfp_t)0x4000u)/* Add compound page
metadata */
00050: #define ___GFP_ZERO  ((__force gfp_t)0x8000u)/* Return zeroed page on
success */
00051: #define ___GFP_NOMEMALLOC ((__force gfp_t)0x10000u) /* Don't
use emergency reserves */
00052: #define ___GFP_HARDWALL   ((__force gfp_t)0x20000u)
00052: /* Enforce hardwall cpuset memory allocs */
00053: #define ___GFP_THISNODE((__force gfp_t)0x40000u)/* No fallback, no
policies */
00054: #define ___GFP_RECLAIMABLE ((__force gfp_t)0x80000u) /* Page is
reclaimable */
00062: #define ___GFP_NO_KSWAPD  ((__force gfp_t)0x400000u)
00063: #define ___GFP_OTHER_NODE ((__force gfp_t)0x800000u)

00021: #define ___GFP_DMA   ((__force gfp_t)0x01u)
00022: #define ___GFP_HIGHMEM ((__force gfp_t)0x02u)
00023: #define ___GFP_DMA32      ((__force gfp_t)0x04u)
00024: #define ___GFP_MOVABLE  ((__force gfp_t)0x08u)   /* Page is movable */
00025: #define GFP_ZONEMASK   (__GFP_DMA| __GFP_HIGHMEM|
__GFP_DMA32| __GFP_MOVABLE)

00062: #define ___GFP_NO_KSWAPD  ((__force gfp_t)0x400000u)
00063: #define ___GFP_OTHER_NODE ((__force gfp_t)0x800000u)
00064:

00069: #define ___GFP_NOTRACK_FALSE_POSITIVE (__GFP_NOTRACK)
00070:
00071: #define ___GFP_BITS_SHIFT 23       /* Room for 23 ____GFP_FOO bits */
00072: #define ___GFP_BITS_MASK ((__force gfp_t)((1 <<
__GFP_BITS_SHIFT) - 1))
```

主要GFP标志含义如下表：

表2 基本GFP标志含义

| 标志位 | 标志位含义 |
|---|---|
| __GFP_WAIT | 内存申请可以被中断。即此时可以调度其他进程运行，也可以被更重要的事件中断；进程也可以被阻塞。 |
| __GFP_HIGH | 该请求非常重要，即内核紧急需要内存。若内存申请失败给内核带来严重影响甚至崩溃，就通常使用该标志。内存申请不允许不允许中断（该标志与 HIGHMEM 无关） |
| __GFP_IO | 在尝试找到空闲内存之前，可以执行磁盘 I/O 操作 |

| | |
|---|---|
| __GFP_FS | 允许执行文件系统 I/O 操作，该标志避免在 VFS 层应用，因为可能会带来死循环递归调用 |
| __GFP_COLD | 要求使用冷页 |
| __GFP_NOWARN | 分配失败不产生告警信息 |
| __GFP_REPEAT | 分配失败后，会尝试几次再分配，再失败，就停止 |
| __GFP_NOFAIL | 分配失败反复尝试，直到成功 |
| __GFP_NORETRY | 分配失败，不再尝试 |
| __GFP_NO_GROW | Slab 机制内部使用 |
| __GFP_COMP | 用于大页面 |
| __GFP_ZERO | 分配的页面用零填充 |
| __GFP_NOMEMALLOC | 不要使用用作紧急使用的页面 |
| __GFP_HARDWALL | 只在 NUMA 中有效，限制进程在节点上的内存分配，该进程绑定在指定 CPU 上运行。若进程运行在所有 CPU 上运行（这是默认值），该标志无效 |
| __GFP_THISNODE | 只在 NUMA 中有效，内存只能在当前节点或指定节点上进行 |
| __GFP_RECLAIMABLE | 该内存分配后，可以被回收 |
| __GFP_MOVABLE | 该内存分配后，可以被移动 |

除了基本GFP标志外，内核还定义了一些组合标志。

```
00073:
00074: /* This equals 0, but use constants in case they ever change */
00075: #define GFP_NOWAIT (GFP_ATOMIC & ~__GFP_HIGH)
00076: /* GFP_ATOMIC means both ! wait (____GFP_WAIT not set) and use emergency pool */
00077: #define GFP_ATOMIC  (__GFP_HIGH)
00078: #define GFP_NOIO    (__GFP_WAIT)
00079: #define GFP_NOFS    (__GFP_WAIT | __GFP_IO)
00080: #define GFP_KERNEL  (__GFP_WAIT | __GFP_IO | __GFP_FS)
00081: #define GFP_TEMPORARY (__GFP_WAIT | __GFP_IO | __GFP_FS |
\
00082:                        __GFP_RECLAIMABLE)
00083: #define GFP_USER    (__GFP_WAIT | __GFP_IO | __GFP_FS |
__GFP_HARDWALL)
00084: #define GFP_HIGHUSER    (__GFP_WAIT | __GFP_IO | __GFP_FS |
__GFP_HARDWALL | \
00085:                        __GFP_HIGHMEM)
00086: #define GFP_HIGHUSER_MOVABLE( __GFP_WAIT | __GFP_IO |
__GFP_FS | \
00087:                           __GFP_HARDWALL | __GFP_HIGHMEM | \
00088:                           __GFP_MOVABLE)
00089: #define GFP_IOFS    (__GFP_IO | __GFP_FS)
00090: #define GFP_TRANSHUGE (GFP_HIGHUSER_MOVABLE |
__GFP_COMP | \
00091:                        __GFP_NOMEMALLOC | __GFP_NORETRY |
__GFP_NOWARN | \
00092:                        __GFP_NO_KSWAPD)
00093:
```

```
00094: #ifdef CONFIG_NUMA
00095: #define GFP_THISNODE    (__GFP_THISNODE | __GFP_NOWARN |
__GFP_NORETRY)
00096: #else

00097: #define GFP_THISNODE    ((__force gfp_t)0)

00098: #endif
00099:
```

表3 组合GFP标志含义

| 组合标志 | 含义 |
|---|---|
| GFP_ATOMIC | 用于原子内存分配操作，即任何情况下不允许被中断，且可能会使用“紧急保留”内存 |
| GFP_NOIO | 不允许 I/O 操作，但可以被中断 |
| GFP_NOFS | 不允许 VFS 操作，但可以被中断 |
| GFP_KERNEL | 用于内核空间内存申请，该标志是内核代码中最常用的标志 |
| GFP_USER | 用于用户空间内存申请 |
| GFP_HIGHUSER | 对 GFP_USER 的扩展，表示可以使用高端内存区域 |
| GFP_IOFS | 用于 VFS 的 I/O 操作 |
| GFP_THISNODE | __GFP_THISNODE \| __GFP_NOWARN \| __GFP_NORETRY |
| GFP_DMA | 用于 DMA 操作 |
| GFP_DMA32 | 用于 DMA32 操作（x86_64 架构中） |