Linux内核源码中关于NUMA部分分析报告

NUMA（Non-Uniform Memory Access）即非一致内存访问结构，如图1所示，在NUMA架构中有多个CPU模块，每个CPU模块由多个CPU组成，并且具有独立的本地内存、I/O槽口等。由于其节点之间可以通过互联模块（如称为Crossbar Switch）进行连接和信息交互，因此每个CPU可以访问整个系统的内存。显然，访问本地内存的速度将远远高于访问远地内存（系统内其它节点的内存）的速度。

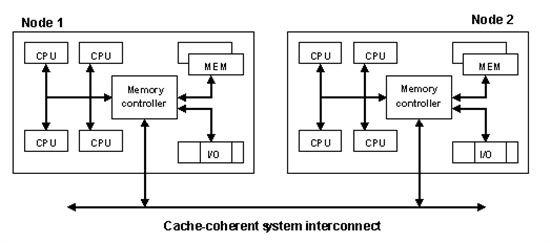


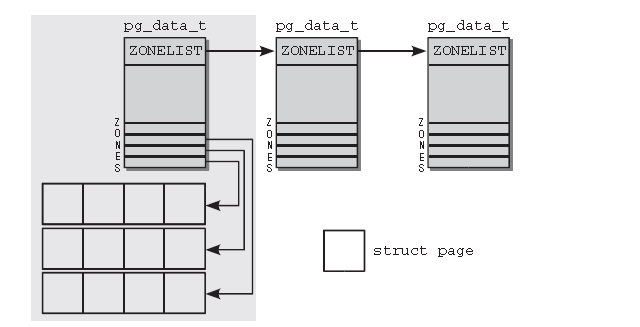
图1 NUMA 结构图

对于Linux 源码中关于NUMA部分的分析，从以下三个方面分析：内存管理、调度、cache。

# 内存管理

内存管理部分，包括物理空间、虚拟空间、越界访问、物理页面管理和分配、内核缓冲区管理、地址映像等部分。

## 物理空间



首先，内核划分为结点。每个结点关联到系统中的一个处理器，在内核中表示为pg\_data\_t（pglist\_data结构）的实例。各个结点又划分为内存域，是内存的进一步细分。Linux 对整个物理内存的管理通过mem\_map表描述。它本身是关于struct page mem\_map\_t的数组，每项对应一个关于内核态、用户态代码和数据等的页帧。物理内存以页帧为单位，页帧的长度固定，等于页长，对Intel CPU缺省为4KB。对每个物理页帧，内核定义page类型的mem\_map\_t数据结构进行描述。

1. pglist\_data定义在linux3.19/include/linux/mmzone.h中。

typedef struct pglist\_data {

struct zone node\_zones[MAX\_NR\_ZONES]; //是一个数组，包含了结点中各内存域的数据结构

struct zonelist node\_zonelists[MAX\_ZONELISTS]; //指点了备用结点及其内存域的列表，以便在当前结点没有可用空间时，在备用结点分配内存

int nr\_zones;

#ifdef CONFIG\_FLAT\_NODE\_MEM\_MAP /\* means !SPARSEMEM \*/

struct page \*node\_mem\_map; //指向page实例数组的指针，用于描述结点的所有物理内存页，它包含了结点中所有内存域的页。

……

} pg\_data\_t;

1. struct zone定义在linux3.19/include/linux/mmzone.h中。free\_area结构也定义在此文件中。

struct zone {

……

#ifdef CONFIG\_NUMA

int node;

#endif

unsigned int inactive\_ratio;

struct pglist\_data \*zone\_pgdat; //建立内存域和父结点之间的关联

struct per\_cpu\_pageset \_\_percpu \*pageset;

unsigned long dirty\_balance\_reserve;

#ifndef CONFIG\_SPARSEMEM

unsigned long \*pageblock\_flags;

#endif /\* CONFIG\_SPARSEMEM \*/

#ifdef CONFIG\_NUMA

/\*

\* zone reclaim becomes active if more unmapped pages exist.

\*/

unsigned long min\_unmapped\_pages;

unsigned long min\_slab\_pages;

#endif /\* CONFIG\_NUMA \*/

/\* zone\_start\_pfn == zone\_start\_paddr >> PAGE\_SHIFT \*/

unsigned long zone\_start\_pfn; //内存域第一个页帧的索引

unsigned long managed\_pages;

unsigned long spanned\_pages; //指定内存域中页的总数，但并非所有的都可用，因为有空洞

unsigned long present\_pages; //指定了内存域中实际上可用的页数目

……

/\* free areas of different sizes \*/

struct free\_area free\_area[MAX\_ORDER]; //每个数组元素都表示某种固定长度的一些连续内存区，对于包含在每个区域中的空闲内存页的管理，free\_area是一个起点。

……

} \_\_\_\_cacheline\_internodealigned\_in\_smp;

struct free\_area {

struct list\_head free\_list[MIGRATE\_TYPES];

unsigned long nr\_free;

};

1. 代表物理页面的 page结构定义在linux3.19/include/linux/mm\_types.h中。 系统在初始化时根据物理内存的大小建立起一个page结构数组的mem\_map，里面的每个page数据结构都代表着系统中的一个物理页面。所有的物理页面被划分成三个管理区zone。每个管理区都有一个数据结构zone\_struct。

struct page {

/\* First double word block \*/

unsigned long flags; /\* Atomic flags, some possibly updated asynchronously \*/

union {

struct address\_space \*mapping; /\* If low bit clear, points to inode address\_space, or NULL. If page mapped as anonymous memory, low bit is set, and it points to anon\_vma object:see PAGE\_MAPPING\_ANON below. \*/

void \*s\_mem; /\* slab first object \*/

};

/\* Second double word \*/

struct {

union {

pgoff\_t index; /\* Our offset within mapping. \*/

void \*freelist; /\* sl[aou]b first free object \*/

bool pfmemalloc; /\* If set by the page allocator, ALLOC\_NO\_WATERMARKS was set and the low watermark was not met implying that the system is under some pressure. The caller should try ensure this page is only used to free other pages. \*/

};

……

};

## 地址映像

进程可用虚存空间共有4GB，但这4GB空间并不是可以让用户态进程任意使用的，只是0至3GB之间的那一部分可以被直接使用，剩下的1GB空间则是属于内核的，用户态进程不能直接访问到。操作系统相关的代码，即内核部分的代码数据都会映射到内核空间，而用户进程则会映射到用户空间。

### 内核空间到物理空间的映射

内核空间中存放的是内核代码和数据，而进程的用户空间中存放的是用户程序的代码和数据。不管是内核空间还是用户空间，它们都处于虚拟空间中。内核空间占据了每个虚拟空间中的最高1GB字节，但映射到物理内存却总是从最低地址（0x00000000）开始。对内核空间来说，其地址映像是很简单的线性映像，PAGE\_OFFSET就是物理地址与线性地址之间的位移量。

linux-3.19\include\asm-generic\page.h

**#ifdef** CONFIG\_KERNEL\_RAM\_BASE\_ADDRESS

**#define** PAGE\_OFFSET (CONFIG\_KERNEL\_RAM\_BASE\_ADDRESS)

**#else**

**#define** PAGE\_OFFSET (0)

**#endif**

**#ifndef** ARCH\_PFN\_OFFSET

**#define** ARCH\_PFN\_OFFSET (PAGE\_OFFSET >> PAGE\_SHIFT)

**#endif**

### 用户空间到物理空间的映射

现在进入页式映射的过程了,Linux系统中的每个进程都有其自身的页面目录PGD,指向这个目录的指针保存在每个进程的mm\_struct数据结构中。每当调度一个进程进入运行的时候，内核都要为即将运行的进程设置好[控制寄存器](http://baike.baidu.com/subview/4092816/4092816.htm)cr3，而MMU的硬件则总是从cr3中取得指向当前页面目录的指针。

* + - 1. linux-3.19\arch\x86\include\asm\pgtable\_types.h

**#ifdef** CONFIG\_X86\_32

**# include** <asm/pgtable\_32\_types.h>

**#else**

**# include** <asm/pgtable\_64\_types.h>

**#endif**

* + - 1. 32位虚拟地址描述

1. linux-3.19\arch\x86\include\asm\pgtable\_32\_types.h

/\*

\* The Linux x86 paging architecture is 'compile-time dual-mode', it

\* implements both the traditional 2-level x86 page tables and the

\* newer 3-level PAE-mode page tables.

\*/

#ifdef CONFIG\_X86\_PAE

# include <asm/pgtable-3level\_types.h>

# define PMD\_SIZE (1UL << PMD\_SHIFT)

# define PMD\_MASK (~(PMD\_SIZE - 1))

#else

# include <asm/pgtable-2level\_types.h>

#endif

根据在编译Linux内核文件之前的系统配置（config）过程中的选择，编译时会把目录include/asm符号连接到具体cpu专用的文件目录。在配置系统还有一个选择项是关于PAE 的，根据此项选择，编译时从pgtable-3level\_types.h或pgtable-2level\_types.h中二者选一。

1. linux-3.19\arch\x86\include\asm\pgtable-3level\_types.h

**……**

**#define** PAGETABLE\_LEVELS 3 //三级页表

**#define** PGDIR\_SHIFT 30 // PGDIR\_SHIFT表示线性地址中PGD下标位段的起始位置，即第31位

**#define** PTRS\_PER\_PGD 4//每个PGD表中指针的个数是4，这与线性地址中的PGD位段的长度（2）相符的，即22=4

**#define** PMD\_SHIFT 21

**#define** PTRS\_PER\_PMD 512

**#define** PTRS\_PER\_PTE 512

1. linux-3.19\arch\x86\include\asm\pgtable-2level\_types.h

**#define SHARED\_KERNEL\_PMD 0**

**#define PAGETABLE\_LEVELS 2//两级页表**

**#define PGDIR\_SHIFT 22 // PGDIR\_SHIFT表示线性地址中PGD下标位段的起始位置，即第23位**

**#define PTRS\_PER\_PGD 1024//每个PGD表中指针的个数是1024，这与线性地址中的PGD位段的长度（10）相符的，即210=1024**

**#define PTRS\_PER\_PTE 1024**

* + - 1. 48位虚拟地址描述

linux-3.19\arch\x86\include\asm\pgtable\_64\_types.h

#define SHARED\_KERNEL\_PMD 0

#define PAGETABLE\_LEVELS 4 //4级页表

#define PGDIR\_SHIFT 39 // PGDIR\_SHIFT表示线性地址中PGD下标位段的起始位置，即第40位

#define PTRS\_PER\_PGD 512//每个PGD表中指针的个数是1024，这与线性地址中的PGD位段的长度（9）相符的，即29=512

#define PUD\_SHIFT 30

#define PTRS\_PER\_PUD 512

#define PMD\_SHIFT 21

#define PTRS\_PER\_PMD 512

#define PTRS\_PER\_PTE 512

* + - 1. linux-3.19\arch\x86\include\asm\pgtable\_types.h

typedef struct { pgdval\_t pgd; } pgd\_t;

**#if** PAGETABLE\_LEVELS > 3

**typedef** **struct** { pudval\_t pud; } pud\_t;

**#else**

**#include** <asm-generic/pgtable-nopud.h>

**static** **inline** pudval\_t native\_pud\_val(pud\_t pud)

{

**return** native\_pgd\_val(pud.pgd);

}

**#endif**

**#if** PAGETABLE\_LEVELS > 2

**typedef** **struct** { pmdval\_t pmd; } pmd\_t;

**#else**

**#include** <asm-generic/pgtable-nopmd.h>

**static** **inline** pmdval\_t native\_pmd\_val(pmd\_t pmd)

{

**return** native\_pgd\_val(pmd.pud.pgd);

}

**#endif**

* + - 1. linux-3.19\arch\x86\include\asm-generic\pgtable.h

**#define** pgd\_index(address) (((address) >> PGDIR\_SHIFT) & (PTRS\_PER\_PGD - 1))

**#define** pgd\_offset(mm, address) ((mm)->pgd + pgd\_index((address)))

**static** **inline** **unsigned** **long** pud\_index(**unsigned** **long** address)

{

**return** (address >> PUD\_SHIFT) & (PTRS\_PER\_PUD - 1);

}

**static** **inline** pud\_t \*pud\_offset(pgd\_t \*pgd, **unsigned** **long** address)

{

**return** (pud\_t \*)pgd\_page\_vaddr(\*pgd) + pud\_index(address);

}

**static** **inline** **unsigned** **long** **pmd\_index**(**unsigned** **long** address)

{

**return** (address >> PMD\_SHIFT) & (PTRS\_PER\_PMD - 1);

}

**static** **inline** pmd\_t \*pmd\_offset(pud\_t \*pud, **unsigned** **long** address)

{

**return** (pmd\_t \*)pud\_page\_vaddr(\*pud) + pmd\_index(address);

}

**static** **inline** **unsigned** **long** **pte\_index**(**unsigned** **long** address)

{

**return** (address >> PAGE\_SHIFT) & (PTRS\_PER\_PTE - 1);

}

## 虚拟空间

无论是内核线程还是用户进程，对于内核来说，无非都是task\_struct这个数据结构的一个实例而已，task\_struct被称为进程描述符（process descriptor),因为它记录了这个进程所有的context。其中有一个被称为“内存描述符”（memory descriptor)的数据结构mm\_struct，抽象并描述了Linux视角下管理进程地址空间的所有信息。

每当新建一个进程，LINUX 都为其分配一个task\_struct结构，此结构中内嵌mm\_struct，有关用户进程中与存储有关的信息都包含在mm\_struct中。

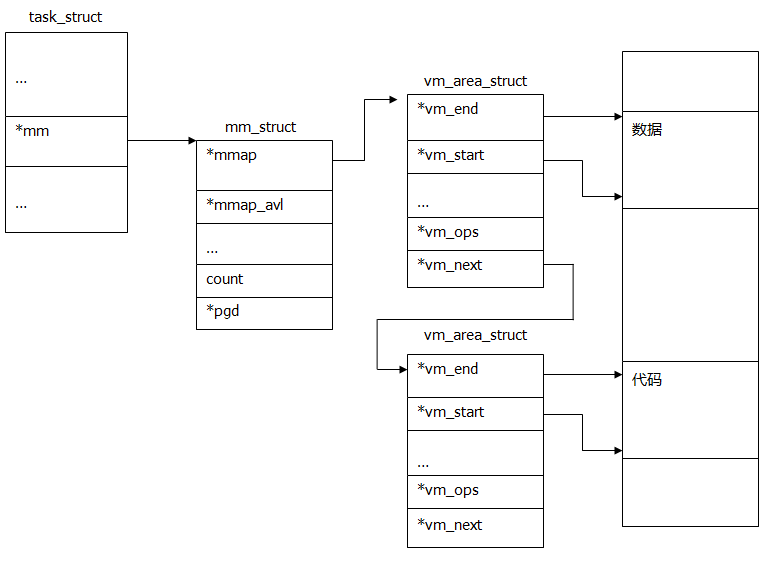


图 进程的虚拟内存

1. task\_struct定义在include/linux/sched.h中。

struct task\_struct {

volatile long state; /\* -1 unrunnable, 0 runnable, >0 stopped \*/

void \*stack;

atomic\_t usage;

unsigned int flags; /\* per process flags, defined below \*/

unsigned int ptrace;

……

struct mm\_struct \*mm, \*active\_mm; //每个进程都有一个active\_mm的成员用于管理该进程的虚拟内存空间

……

｝

1. mm\_struct定义在include/linux/mm\_types.h中，其中的域抽象了进程的地址空间。此mm\_struct结构包含一个进程虚存空间的基本面描述和相关的信息记录，它是后续虚存段管理的基础。vma为虚拟内存空间。

struct mm\_struct {

struct vm\_area\_struct \*mmap; /\* list of VMAs，指向VMA段双向链表的指针 \*/

struct rb\_root mm\_rb;/\*指向VMA段红黑树的指针, mm\_rb是红黑树的根，该进程的所有虚拟空间块（虚拟地址不连续）都以起始虚拟地址为key值挂在该红黑树上。该进程新申请的虚拟内存区间会插入到这棵树上，当然插入过程中可能会合并相邻的虚拟区域。删除时会从该树上摘除相应的node。为了提高查找效率，内核又以红黑树的形式组织内存区域，以便降低搜索耗时。\*/

……

unsigned long mmap\_base; /\* base of mmap area \*/

unsigned long mmap\_legacy\_base; /\* base of mmap area in bottom-up allocations \*/

unsigned long task\_size; /\* size of task vm space \*/

unsigned long highest\_vm\_end; /\* highest vma end address \*/

pgd\_t \* pgd; /\*进程页目录的起始地址\*/

atomic\_t mm\_users; /\*记录目前正在使用此mm\_struct结构的用户个数 \*/

atomic\_t mm\_count; /\* How many references to "struct mm\_struct" (users count as 1),mm\_struct被内核线程应用次数 \*/

atomic\_long\_t nr\_ptes; /\* Page table pages \*/

int map\_count; /\* 进程所使用的vma块的个数\*/

spinlock\_t page\_table\_lock; /\* 对此进程的页表操作时所需要的自旋锁Protects page tables and some counters \*/

struct rw\_semaphore mmap\_sem; /\*对mmap操作的互斥信号量，由down()和 up()更改\*/

struct list\_head mmlist; /\*task\_struct中的active\_mm域的链表

……

unsigned long start\_code, end\_code, start\_data, end\_data; /\*进程代码段的起始地址和结束地址、进程数据段的起始地址和结束地址\*/

unsigned long start\_brk, brk, start\_stack;/\*进程未初始化的数据段的起始地址和结束地址\*/

unsigned long arg\_start, arg\_end, env\_start, env\_end; /\*调用参数的起始地址和结束地址\*/

……

};

1. 一个进程在运行过程中使用到的物理内存一般是不连续的，用到的虚拟地址也不是连成一片的，而是被分成几块，进程通常占用几个虚存段，分别用于代码段、数据段、堆栈段等。每个进程的所有虚存段通过指针构成链表，虚存段在此链表中的排列顺序按照它们的地址增长顺序进行。linux定义了虚存段vma，即virtual memeory area。vma段是属于某个进程的一段连续的虚存空间，在这段虚存里的所有页面拥有一些相同的特征。用户程序对内存的操作都是对 mm 的操作，具体来说是对 mm 上的 vma（虚拟内存空间）的操作。

vm\_area\_struct定义在include/linux/mm\_types.h中。

struct vm\_area\_struct {

/\* The first cache line has the info for VMA tree walking. \*/

unsigned long vm\_start; /\*vma描述的虚拟内存段起始地址\*/

unsigned long vm\_end; /\* vma描述的虚拟内存段结束地址\*/

/\* linked list of VM areas per task, sorted by address \*/

struct vm\_area\_struct \*vm\_next, \*vm\_prev;

struct rb\_node vm\_rb; //用于对vma块进行rb操作的结构体

……

struct mm\_struct \*vm\_mm; /\*同一个进程的所有vma块用这个指针指向此进程的

mm\_struct结构\*/

pgprot\_t vm\_page\_prot; /\*此变量指示本vma块中所有页面的保护模式Access permissions of this VMA. \*/

unsigned long vm\_flags; /\*本vma块的属性标志位Flags, see mm.h. \*/

……

/\* Function pointers to deal with this struct. \*/

const struct vm\_operations\_struct \*vm\_ops; /\*指向一个结构体的指针，该结构体中是对vma段进行操作的函数指针的集合\*/

……

};

1. struct vm\_operation\_struct结构定义在 linux 3.19/include/linux/mm.h中，该结构体中是对vma段进行操作的函数指针的集合。

struct vm\_operations\_struct {

void (\*open)(struct vm\_area\_struct \* area);

void (\*close)(struct vm\_area\_struct \* area);

int (\*fault)(struct vm\_area\_struct \*vma, struct vm\_fault \*vmf);

void (\*map\_pages)(struct vm\_area\_struct \*vma, struct vm\_fault \*vmf);

int (\*page\_mkwrite)(struct vm\_area\_struct \*vma, struct vm\_fault \*vmf);

int (\*access)(struct vm\_area\_struct \*vma, unsigned long addr,

void \*buf, int len, int write);

const char \*(\*name)(struct vm\_area\_struct \*vma);

#ifdef CONFIG\_NUMA

int (\*set\_policy)(struct vm\_area\_struct \*vma, struct mempolicy \*new);

struct mempolicy \*(\*get\_policy)(struct vm\_area\_struct \*vma,

unsigned long addr);

#endif

……

};

当进程请求分配内存时，linux 并不直接为其分配物理内存。它只是创建一个vm\_area\_struct结构来描述此虚拟内存。此结构被链接到进程的虚拟内存链表中。当进程试图对新分配的虚拟内存进行写操作时，系统将产生页面中断，处理机会尝试解析此虚拟地址，如果找不到对应此虚拟地址的页表，处理机将放弃解析并产生页面错误异常。由linux 核心作进一步的处理。如果找到了页表项，linux继续查找此虚拟地址是否在当前进程的虚拟地址空间中。如果在，linux就创建正确的PTE,并为此进程分配物理页面。包含在此页面中的代码或数据可能要从文件系统或者交换磁盘上读出。然后进程将从页面错误处继续开始执行，由于物理内存已经存在，所以不会再产生页面异常。

## 物理页面管理和分配

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。内存分配的标志GFP（ Get Free Page）决定内存分配器、kswapd分配和回收页面时的行为。

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

**#ifdef** CONFIG\_NUMA

**extern** **struct** page \*alloc\_pages\_current(gfp\_t gfp\_mask, **unsigned** order);

**static** **inline** **struct** page \*

alloc\_pages(gfp\_t gfp\_mask, **unsigned** **int** order)

{

**return** alloc\_pages\_current(gfp\_mask, order);

}

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2MAX\_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2MAX\_ORDER－1即210个页面。

函数alloc\_pages\_current（）实现在文件mm/mempolicy.c中。

**struct** page \***alloc\_pages\_current**(gfp\_t gfp, **unsigned** order)

{

**struct** mempolicy \*pol = &default\_policy;

**struct** page \*page;

**unsigned** **int** cpuset\_mems\_cookie;

**if** (!in\_interrupt() && !(gfp & \_\_GFP\_THISNODE))

pol = get\_task\_policy(current);

retry\_cpuset:

cpuset\_mems\_cookie = read\_mems\_allowed\_begin();

/\*

\* No reference counting needed for current->mempolicy

\* nor system default\_policy

\*/

**if** (pol->mode == *MPOL\_INTERLEAVE*)

page = alloc\_page\_interleave(gfp, order, interleave\_nodes(pol));

**else**

page = \_\_alloc\_pages\_nodemask(gfp, order,

policy\_zonelist(gfp, pol, numa\_node\_id()),

policy\_nodemask(gfp, pol));

**if** (unlikely(!page && read\_mems\_allowed\_retry(cpuset\_mems\_cookie)))

**goto** retry\_cpuset;

**return** page;

}

EXPORT\_SYMBOL(alloc\_pages\_current);

内存策略就是指应用程度控制自己的内存分配。Linux3.19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。

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时，就是用系统默认的分配策略。默认的内存分配策略为MPOL\_PREFERRED。

**static** **struct** mempolicy default\_policy = {

.refcnt = ATOMIC\_INIT(1), /\* never free it \*/

.mode = *MPOL\_PREFERRED*,

.flags = MPOL\_F\_LOCAL,

};

NUMA默认内存分配策略下的情况下，代码会继续执行。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。

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。

**enum** {

*MIGRATE\_UNMOVABLE*,

*MIGRATE\_RECLAIMABLE*,

*MIGRATE\_MOVABLE*,

*MIGRATE\_PCPTYPES*, /\* the number of types on the pcp lists \*/

*MIGRATE\_RESERVE* = *MIGRATE\_PCPTYPES*,

**#ifdef** CONFIG\_CMA

/\*

\* MIGRATE\_CMA migration type is designed to mimic the way

\* ZONE\_MOVABLE works. Only movable pages can be allocated

\* from MIGRATE\_CMA pageblocks and page allocator never

\* implicitly change migration type of MIGRATE\_CMA pageblock.

\*

\* The way to use it is to change migratetype of a range of

\* pageblocks to MIGRATE\_CMA which can be done by

\* \_\_free\_pageblock\_cma() function. What is important though

\* is that a range of pageblocks must be aligned to

\* MAX\_ORDER\_NR\_PAGES should biggest page be bigger then

\* a single pageblock.

\*/

MIGRATE\_CMA,

**#endif**

**#ifdef** CONFIG\_MEMORY\_ISOLATION

MIGRATE\_ISOLATE, /\* can't allocate from here \*/

**#endif**

*MIGRATE\_TYPES*

};

继续分析函数\_\_alloc\_pages\_nodemask（）实现。

\_\_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。

**struct** page \*

\_\_alloc\_pages\_nodemask(gfp\_t gfp\_mask, **unsigned** **int** order,

**struct** zonelist \*zonelist, nodemask\_t \*nodemask)

{

**enum** zone\_type high\_zoneidx = gfp\_zone(gfp\_mask);

**struct** zone \*preferred\_zone;

**struct** zoneref \*preferred\_zoneref;

**struct** page \*page = NULL;

**int** migratetype = gfpflags\_to\_migratetype(gfp\_mask);

**unsigned** **int** cpuset\_mems\_cookie;

**int** alloc\_flags = ALLOC\_WMARK\_LOW|ALLOC\_CPUSET|ALLOC\_FAIR;

**int** classzone\_idx;

gfp\_mask &= gfp\_allowed\_mask;

lockdep\_trace\_alloc(gfp\_mask);

might\_sleep\_if(gfp\_mask & \_\_GFP\_WAIT);

**if** (should\_fail\_alloc\_page(gfp\_mask, order))

**return** NULL;

/\*

\* Check the zones suitable for the gfp\_mask contain at least one

\* valid zone. It's possible to have an empty zonelist as a result

\* of GFP\_THISNODE and a memoryless node

\*/

**if** (unlikely(!zonelist->\_zonerefs->zone))

**return** NULL;

**if** (IS\_ENABLED(CONFIG\_CMA) && migratetype == MIGRATE\_MOVABLE)

alloc\_flags |= ALLOC\_CMA;

retry\_cpuset:

cpuset\_mems\_cookie = read\_mems\_allowed\_begin();

/\* The preferred zone is used for statistics later \*/

preferred\_zoneref = first\_zones\_zonelist(zonelist, high\_zoneidx,

nodemask ? : &cpuset\_current\_mems\_allowed,

&preferred\_zone);

**if** (!preferred\_zone)

**goto** out;

classzone\_idx = zonelist\_zone\_idx(preferred\_zoneref);

/\* First allocation attempt \*/

page = get\_page\_from\_freelist(gfp\_mask|\_\_GFP\_HARDWALL, nodemask, order,

zonelist, high\_zoneidx, alloc\_flags,

preferred\_zone, classzone\_idx, migratetype);

**if** (unlikely(!page)) {

/\*

\* Runtime PM, block IO and its error handling path

\* can deadlock because I/O on the device might not

\* complete.

\*/

gfp\_mask = memalloc\_noio\_flags(gfp\_mask);

page = \_\_alloc\_pages\_slowpath(gfp\_mask, order,

zonelist, high\_zoneidx, nodemask,

preferred\_zone, classzone\_idx, migratetype);

}

trace\_mm\_page\_alloc(page, order, gfp\_mask, migratetype);

out:

/\*

\* When updating a task's mems\_allowed, it is possible to race with

\* parallel threads in such a way that an allocation can fail while

\* the mask is being updated. If a page allocation is about to fail,

\* check if the cpuset changed during allocation and if so, retry.

\*/

**if** (unlikely(!page && read\_mems\_allowed\_retry(cpuset\_mems\_cookie)))

**goto** retry\_cpuset;

**return** page;

}

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（）从区域列表中分配2order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。

### get\_page\_from\_freelist（）函数

get\_page\_from\_freelist（）函数也在文件mm/page\_alloc.c中。函数主体是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。

**static** **struct** page \*

get\_page\_from\_freelist(gfp\_t gfp\_mask, nodemask\_t \*nodemask, **unsigned** **int** order,

**struct** zonelist \*zonelist, **int** high\_zoneidx, **int** alloc\_flags,

**struct** zone \*preferred\_zone, **int** classzone\_idx, **int** migratetype)

{

**struct** zoneref \*z;

**struct** page \*page = NULL;

**struct** zone \*zone;

nodemask\_t \*allowednodes = NULL;/\* zonelist\_cache approximation \*/

**int** zlc\_active = 0; /\* set if using zonelist\_cache \*/

**int** did\_zlc\_setup = 0; /\* just call zlc\_setup() one time \*/

bool consider\_zone\_dirty = (alloc\_flags & ALLOC\_WMARK\_LOW) &&

(gfp\_mask & \_\_GFP\_WRITE);

**int** nr\_fair\_skipped = 0;

bool zonelist\_rescan;

zonelist\_scan:

zonelist\_rescan = false;

/\*

\* Scan zonelist, looking for a zone with enough free.

\* See also \_\_cpuset\_node\_allowed() comment in kernel/cpuset.c.

\*/

for\_each\_zone\_zonelist\_nodemask(zone, z, zonelist,

high\_zoneidx, nodemask) {

**unsigned** **long** mark;

**if** (IS\_ENABLED(CONFIG\_NUMA) && zlc\_active &&

!zlc\_zone\_worth\_trying(zonelist, z, allowednodes))

**continue**;

**if** (cpusets\_enabled() &&

(alloc\_flags & ALLOC\_CPUSET) &&

!cpuset\_zone\_allowed(zone, gfp\_mask))

**continue**;

/\*

\* Distribute pages in proportion to the individual

\* zone size to ensure fair page aging. The zone a

\* page was allocated in should have no effect on the

\* time the page has in memory before being reclaimed.

\*/

**if** (alloc\_flags & ALLOC\_FAIR) {

**if** (!zone\_local(preferred\_zone, zone))

**break**;

**if** (test\_bit(ZONE\_FAIR\_DEPLETED, &zone->flags)) {

nr\_fair\_skipped++;

**continue**;

}

}

/\*

\* When allocating a page cache page for writing, we

\* want to get it from a zone that is within its dirty

\* limit, such that no single zone holds more than its

\* proportional share of globally allowed dirty pages.

\* The dirty limits take into account the zone's

\* lowmem reserves and high watermark so that kswapd

\* should be able to balance it without having to

\* write pages from its LRU list.

\*

\* This may look like it could increase pressure on

\* lower zones by failing allocations in higher zones

\* before they are full. But the pages that do spill

\* over are limited as the lower zones are protected

\* by this very same mechanism. It should not become

\* a practical burden to them.

\*

\* **XXX**: For now, allow allocations to potentially

\* exceed the per-zone dirty limit in the slowpath

\* (ALLOC\_WMARK\_LOW unset) before going into reclaim,

\* which is important when on a NUMA setup the allowed

\* zones are together not big enough to reach the

\* global limit. The proper fix for these situations

\* will require awareness of zones in the

\* dirty-throttling and the flusher threads.

\*/

**if** (consider\_zone\_dirty && !zone\_dirty\_ok(zone))

**continue**;

mark = zone->watermark[alloc\_flags & ALLOC\_WMARK\_MASK];

**if** (!zone\_watermark\_ok(zone, order, mark,

classzone\_idx, alloc\_flags)) {

**int** ret;

/\* Checked here to keep the fast path fast \*/

BUILD\_BUG\_ON(ALLOC\_NO\_WATERMARKS < NR\_WMARK);

**if** (alloc\_flags & ALLOC\_NO\_WATERMARKS)

**goto** try\_this\_zone;

**if** (IS\_ENABLED(CONFIG\_NUMA) &&

!did\_zlc\_setup && nr\_online\_nodes > 1) {

/\*

\* we do zlc\_setup if there are multiple nodes

\* and before considering the first zone allowed

\* by the cpuset.

\*/

allowednodes = zlc\_setup(zonelist, alloc\_flags);

zlc\_active = 1;

did\_zlc\_setup = 1;

}

**if** (zone\_reclaim\_mode == 0 ||

!zone\_allows\_reclaim(preferred\_zone, zone))

**goto** this\_zone\_full;

/\*

\* As we may have just activated ZLC, check if the first

\* eligible zone has failed zone\_reclaim recently.

\*/

**if** (IS\_ENABLED(CONFIG\_NUMA) && zlc\_active &&

!zlc\_zone\_worth\_trying(zonelist, z, allowednodes))

**continue**;

ret = zone\_reclaim(zone, gfp\_mask, order);

**switch** (ret) {

**case** ZONE\_RECLAIM\_NOSCAN:

/\* did not scan \*/

**continue**;

**case** ZONE\_RECLAIM\_FULL:

/\* scanned but unreclaimable \*/

**continue**;

**default**:

/\* did we reclaim enough \*/

**if** (zone\_watermark\_ok(zone, order, mark,

classzone\_idx, alloc\_flags))

**goto** try\_this\_zone;

/\*

\* Failed to reclaim enough to meet watermark.

\* Only mark the zone full if checking the min

\* watermark or if we failed to reclaim just

\* 1<<order pages or else the page allocator

\* fastpath will prematurely mark zones full

\* when the watermark is between the low and

\* min watermarks.

\*/

**if** (((alloc\_flags & ALLOC\_WMARK\_MASK) == ALLOC\_WMARK\_MIN) ||

ret == ZONE\_RECLAIM\_SOME)

**goto** this\_zone\_full;

**continue**;

}

}

try\_this\_zone:

page = buffered\_rmqueue(preferred\_zone, zone, order,

gfp\_mask, migratetype);

**if** (page)

**break**;

this\_zone\_full:

**if** (IS\_ENABLED(CONFIG\_NUMA) && zlc\_active)

zlc\_mark\_zone\_full(zonelist, z);

}

**if** (page) {

/\*

\* page->pfmemalloc is set when ALLOC\_NO\_WATERMARKS was

\* necessary to allocate the page. The expectation is

\* that the caller is taking steps that will free more

\* memory. The caller should avoid the page being used

\* for !PFMEMALLOC purposes.

\*/

page->pfmemalloc = !!(alloc\_flags & ALLOC\_NO\_WATERMARKS);

**return** page;

}

/\*

\* The first pass makes sure allocations are spread fairly within the

\* local node. However, the local node might have free pages left

\* after the fairness batches are exhausted, and remote zones haven't

\* even been considered yet. Try once more without fairness, and

\* include remote zones now, before entering the slowpath and waking

\* kswapd: prefer spilling to a remote zone over swapping locally.

\*/

**if** (alloc\_flags & ALLOC\_FAIR) {

alloc\_flags &= ~ALLOC\_FAIR;

**if** (nr\_fair\_skipped) {

zonelist\_rescan = true;

reset\_alloc\_batches(preferred\_zone);

}

**if** (nr\_online\_nodes > 1)

zonelist\_rescan = true;

}

**if** (unlikely(IS\_ENABLED(CONFIG\_NUMA) && zlc\_active)) {

/\* Disable zlc cache for second zonelist scan \*/

zlc\_active = 0;

zonelist\_rescan = true;

}

**if** (zonelist\_rescan)

**goto** zonelist\_scan;

**return** NULL;

}

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。

接下来就要检查区域水准，看当前区域是否满足水准要求。

（1）若本区域水准满足要求，则直接尝试在本区域分配页面；  
 （2）若本区域水准不能满足要求，且区域zone\_reclaim\_mode的值为0，则跳转到this\_zone\_full；

（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。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；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； 若成功回收了部分内存，则重新检查区域水准。

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。

**static** **inline**

**struct** page \*buffered\_rmqueue(**struct** zone \*preferred\_zone,

**struct** zone \*zone, **unsigned** **int** order,

gfp\_t gfp\_flags, **int** migratetype)

{

**unsigned** **long** flags;

**struct** page \*page;

bool cold = ((gfp\_flags & \_\_GFP\_COLD) != 0);

again:

**if** (likely(order == 0)) {

**struct** per\_cpu\_pages \*pcp;

**struct** list\_head \*list;

local\_irq\_save(flags);

pcp = &this\_cpu\_ptr(zone->pageset)->pcp;

list = &pcp->lists[migratetype];

**if** (list\_empty(list)) {

pcp->count += rmqueue\_bulk(zone, 0,

pcp->batch, list,

migratetype, cold);

**if** (unlikely(list\_empty(list)))

**goto** failed;

}

**if** (cold)

page = list\_entry(list->prev, **struct** page, lru);

**else**

page = list\_entry(list->next, **struct** page, lru);

list\_del(&page->lru);

pcp->count--;

} **else** {

**if** (unlikely(gfp\_flags & \_\_GFP\_NOFAIL)) {

/\*

\* \_\_GFP\_NOFAIL is not to be used in new code.

\*

\* All \_\_GFP\_NOFAIL callers should be fixed so that they

\* properly detect and handle allocation failures.

\*

\* We most definitely don't want callers attempting to

\* allocate greater than order-1 page units with

\* \_\_GFP\_NOFAIL.

\*/

WARN\_ON\_ONCE(order > 1);

}

spin\_lock\_irqsave(&zone->lock, flags);

page = \_\_rmqueue(zone, order, migratetype);

spin\_unlock(&zone->lock);

**if** (!page)

**goto** failed;

\_\_mod\_zone\_freepage\_state(zone, -(1 << order),

get\_freepage\_migratetype(page));

}

\_\_mod\_zone\_page\_state(zone, NR\_ALLOC\_BATCH, -(1 << order));

**if** (atomic\_long\_read(&zone->vm\_stat[NR\_ALLOC\_BATCH]) <= 0 &&

!test\_bit(ZONE\_FAIR\_DEPLETED, &zone->flags))

set\_bit(ZONE\_FAIR\_DEPLETED, &zone->flags);

\_\_count\_zone\_vm\_events(PGALLOC, zone, 1 << order);

zone\_statistics(preferred\_zone, zone, gfp\_flags);

local\_irq\_restore(flags);

VM\_BUG\_ON\_PAGE(bad\_range(zone, page), page);

**if** (prep\_new\_page(page, order, gfp\_flags))

**goto** again;

**return** page;

failed:

local\_irq\_restore(flags);

**return** NULL;

}

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，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。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。

接下来就是通过\_\_rmqueue（）来从zone区域中分配2order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，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。

\_\_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。

**static** **struct** page \*\_\_rmqueue(**struct** zone \*zone, **unsigned** **int** order,

**int** migratetype)

{

**struct** page \*page;

retry\_reserve:

page = \_\_rmqueue\_smallest(zone, order, migratetype);

**if** (unlikely(!page) && migratetype != MIGRATE\_RESERVE) {

page = \_\_rmqueue\_fallback(zone, order, migratetype);

/\*

\* Use MIGRATE\_RESERVE rather than fail an allocation. goto

\* is used because \_\_rmqueue\_smallest is an inline function

\* and we want just one call site

\*/

**if** (!page) {

migratetype = MIGRATE\_RESERVE;

**goto** retry\_reserve;

}

}

trace\_mm\_page\_alloc\_zone\_locked(page, order, migratetype);

**return** page;

}

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，找不到给定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。

\_\_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Linux物理内存页面分配表上（expand（）函数），页面分配成功返回；  
（ 5）页面分配失败返回。  
 \_\_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。  
 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。

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。

### \_\_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。

**static** **inline** **struct** page \*

\_\_alloc\_pages\_slowpath(gfp\_t gfp\_mask, **unsigned** **int** order,

**struct** zonelist \*zonelist, **enum** zone\_type high\_zoneidx,

nodemask\_t \*nodemask, **struct** zone \*preferred\_zone,

**int** classzone\_idx, **int** migratetype)

{

**const** gfp\_t wait = gfp\_mask & \_\_GFP\_WAIT;

**struct** page \*page = NULL;

**int** alloc\_flags;

**unsigned** **long** pages\_reclaimed = 0;

**unsigned** **long** did\_some\_progress;

**enum** migrate\_mode migration\_mode = MIGRATE\_ASYNC;

bool deferred\_compaction = false;

**int** contended\_compaction = COMPACT\_CONTENDED\_NONE;

/\*

\* In the slowpath, we sanity check order to avoid ever trying to

\* reclaim >= MAX\_ORDER areas which will never succeed. Callers may

\* be using allocators in order of preference for an area that is

\* too large.

\*/

**if** (order >= MAX\_ORDER) {

WARN\_ON\_ONCE(!(gfp\_mask & \_\_GFP\_NOWARN));

**return** NULL;

}

/\*

\* GFP\_THISNODE (meaning \_\_GFP\_THISNODE, \_\_GFP\_NORETRY and

\* \_\_GFP\_NOWARN set) should not cause reclaim since the subsystem

\* (f.e. slab) using GFP\_THISNODE may choose to trigger reclaim

\* using a larger set of nodes after it has established that the

\* allowed per node queues are empty and that nodes are

\* over allocated.

\*/

**if** (IS\_ENABLED(CONFIG\_NUMA) &&

(gfp\_mask & GFP\_THISNODE) == GFP\_THISNODE)

**goto** nopage;

retry:

**if** (!(gfp\_mask & \_\_GFP\_NO\_KSWAPD))

wake\_all\_kswapds(order, zonelist, high\_zoneidx,

preferred\_zone, nodemask);

/\*

\* OK, we're below the kswapd watermark and have kicked background

\* reclaim. Now things get more complex, so set up alloc\_flags according

\* to how we want to proceed.

\*/

alloc\_flags = gfp\_to\_alloc\_flags(gfp\_mask);

/\*

\* Find the true preferred zone if the allocation is unconstrained by

\* cpusets.

\*/

**if** (!(alloc\_flags & ALLOC\_CPUSET) && !nodemask) {

**struct** zoneref \*preferred\_zoneref;

preferred\_zoneref = first\_zones\_zonelist(zonelist, high\_zoneidx,

NULL, &preferred\_zone);

classzone\_idx = zonelist\_zone\_idx(preferred\_zoneref);

}

/\* This is the last chance, in general, before the goto nopage. \*/

page = get\_page\_from\_freelist(gfp\_mask, nodemask, order, zonelist,

high\_zoneidx, alloc\_flags & ~ALLOC\_NO\_WATERMARKS,

preferred\_zone, classzone\_idx, migratetype);

**if** (page)

**goto** got\_pg;

/\* Allocate without watermarks if the context allows \*/

**if** (alloc\_flags & ALLOC\_NO\_WATERMARKS) {

/\*

\* Ignore mempolicies if ALLOC\_NO\_WATERMARKS on the grounds

\* the allocation is high priority and these type of

\* allocations are system rather than user orientated

\*/

zonelist = node\_zonelist(numa\_node\_id(), gfp\_mask);

page = \_\_alloc\_pages\_high\_priority(gfp\_mask, order,

zonelist, high\_zoneidx, nodemask,

preferred\_zone, classzone\_idx, migratetype);

**if** (page) {

**goto** got\_pg;

}

}

/\* Atomic allocations - we can't balance anything \*/

**if** (!wait) {

/\*

\* All existing users of the deprecated \_\_GFP\_NOFAIL are

\* blockable, so warn of any new users that actually allow this

\* type of allocation to fail.

\*/

WARN\_ON\_ONCE(gfp\_mask & \_\_GFP\_NOFAIL);

**goto** nopage;

}

/\* Avoid recursion of direct reclaim \*/

**if** (current->flags & PF\_MEMALLOC)

**goto** nopage;

/\* Avoid allocations with no watermarks from looping endlessly \*/

**if** (test\_thread\_flag(TIF\_MEMDIE) && !(gfp\_mask & \_\_GFP\_NOFAIL))

**goto** nopage;

/\*

\* Try direct compaction. The first pass is asynchronous. Subsequent

\* attempts after direct reclaim are synchronous

\*/

page = \_\_alloc\_pages\_direct\_compact(gfp\_mask, order, zonelist,

high\_zoneidx, nodemask, alloc\_flags,

preferred\_zone,

classzone\_idx, migratetype,

migration\_mode, &contended\_compaction,

&deferred\_compaction);

**if** (page)

**goto** got\_pg;

/\* Checks for THP-specific high-order allocations \*/

**if** ((gfp\_mask & GFP\_TRANSHUGE) == GFP\_TRANSHUGE) {

/\*

\* If compaction is deferred for high-order allocations, it is

\* because sync compaction recently failed. If this is the case

\* and the caller requested a THP allocation, we do not want

\* to heavily disrupt the system, so we fail the allocation

\* instead of entering direct reclaim.

\*/

**if** (deferred\_compaction)

**goto** nopage;

/\*

\* In all zones where compaction was attempted (and not

\* deferred or skipped), lock contention has been detected.

\* For THP allocation we do not want to disrupt the others

\* so we fallback to base pages instead.

\*/

**if** (contended\_compaction == COMPACT\_CONTENDED\_LOCK)

**goto** nopage;

/\*

\* If compaction was aborted due to need\_resched(), we do not

\* want to further increase allocation latency, unless it is

\* khugepaged trying to collapse.

\*/

**if** (contended\_compaction == COMPACT\_CONTENDED\_SCHED

&& !(current->flags & PF\_KTHREAD))

**goto** nopage;

}

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。

通过gfp\_mask标志，转换成对应的alloc\_flags，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Linux物理内存页面分配  
get\_page\_from\_freelist（）作了一层封装， alloc\_flags参数设置为  
ALLOC\_NO\_WATERMARKS。

（1）\_\_alloc\_pages\_direct\_compact（）

调用\_\_alloc\_pages\_direct\_compact（）函数来分配内存，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，再分配页面块。

函数主要两个函数try\_to\_compact\_pages（）合并小页面块，然后  
再次调用get\_page\_from\_freelist（）分配页面块。

（2）\_\_alloc\_pages\_direct\_reclaim（）

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。\_\_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。

# 调度

# Cache