# 一、物理内存描述

内存访问分为两种体系结构：一致性内存访问（UMA）和非一致性内存访问（NUMA）。NUMA指CPU对不同内存单元的访问时间可能不一样，因而这些物理内存被划分为几个节点，每个节点里的内存访问时间一致，NUMA体系结构主要存在大型机器、alpha等，嵌入式的基本都是UMA。UMA也使用了节点概念，只是永远都只有1个节点。本文讲的是UMA模型的嵌入式平台，linux版本：3.10.y。

每个节点又将物理内存划分为3个管理区，在x86机器上管理区如下：

ZONE\_DMA：0~16MB

ZONE\_NORMAL：16MB~896MB

ZONE\_HIGHMEM：896~末尾（对于64位，则不需要高端内存，虚拟地址足够直接映射）

而在hi3536嵌入式平台上实际的使用时由于系统mem没有超过896M，因此只有一个管理区ZONE\_NORMAL（cat /proc/buddyinfo）：

Node 0, zone Normal 70 70 36 22 11 10 2 2 3 2 37

每个管理区管理该内存区域内的所有页面（linux中每个页面的大小为4KB）。

以下linux物理内存组织结构标准的关系图：



在Documentation/arm/memory.txt文件中定义了arm平台的线性地址空间布局情况，其与物理内存关系图：



当用户空间和内核空间比例3:1时，PAGE\_OFFSET=0xC0000000。上面橙色虚线地址是一一映射关系，而高于highmem则需要vmalloc申请使用。

PHYS\_OFFSET=0x40000000

ZRELADDR == virt\_to\_phys(PAGE\_OFFSET + TEXT\_OFFSET) = virt\_to\_phys (TEXT\_ADDR) = 0x40008000

INITRD\_PHYS = 0x00800000

PARAMS\_PHYS = 0x00000100

high\_memory和VMALLOC\_START之间保留了8M空间间隙（实际测试只有6M）

high\_memory=PAGE\_OFFSET + highmem

当系统内存在0~896MB时，highmem=系统内存

当系统内存>896MB时，highmem=896MB，多余的内存需要vmalloc映射访问

VMALLOC\_START 和VMALLOC\_END-1之间用于vmalloc() / ioremap() space映射。

cat /proc/vmallocinfo：

0xfb000000-0xfe190000 51970048 iotable\_init+0x0/0xb0 phys=10000000 ioremap

映射了CPU寄存器空间

## 1、节点

每个节点由struct pglist\_data定义（include/linux/mmzone.h）：

typedef struct pglist\_data {

#define MAX\_NR\_ZONES 3

struct zone node\_zones[MAX\_NR\_ZONES]; //分别代表3个管理区

#define MAX\_ZONELISTS 1

struct zonelist node\_zonelists[MAX\_ZONELISTS]; //按照分配时的管理区顺序排列，如果在ZONE\_HIGHMEM中分配失败，就有可能还原成ZONE\_NORMAL或ZONE\_DMA。

int nr\_zones; //表示管理区的数目，值为1、2、3。

struct page \*node\_mem\_map; //指向该节点第一个物理页面

struct bootmem\_data \*bdata; //指向内存引导程序

unsigned long node\_start\_pfn; //该节点的起始页编号

//calculate\_node\_totalpages中对以下两个值进行计算

unsigned long node\_present\_pages; /\* total number of physical pages \*/

unsigned long node\_spanned\_pages; /\* total size of physical pagerange, including holes \*/

int node\_id; //节点号，在嵌入式上一般为0

nodemask\_t reclaim\_nodes; /\* Nodes allowed to reclaim from \*/

wait\_queue\_head\_t kswapd\_wait; //交换守护进程kswapd使用的等待队列

wait\_queue\_head\_t pfmemalloc\_wait;

struct task\_struct \*kswapd; //指向交换守护进程描述符

int kswapd\_max\_order;

enum zone\_type classzone\_idx; //管理区类型

} pg\_data\_t;

include/linux/mmzone.h中定义了全局节点：

extern struct pglist\_data contig\_page\_data;

#define NODE\_DATA(nid) (&contig\_page\_data)

mm/bootmem.c进行全局节点定义：

struct pglist\_data \_\_refdata contig\_page\_data = {

.bdata = &bootmem\_node\_data[0]

};

EXPORT\_SYMBOL(contig\_page\_data);

## 2、管理区

每个管理区由struct zone描述（include/linux/mmzone.h）：

struct zone {

unsigned long watermark[NR\_WMARK]; //该管理区的三个水平线值，min, low, high

unsigned long percpu\_drift\_mark;

unsigned long lowmem\_reserve[MAX\_NR\_ZONES]; //每个管理区必须保留的页框数

unsigned long dirty\_balance\_reserve;

struct per\_cpu\_pageset \_\_percpu \*pageset; //CPU的页面缓存

spinlock\_t lock; //保护该管理区的自旋锁

int all\_unreclaimable; /\* All pages pinned \*/

struct free\_area free\_area[MAX\_ORDER]; //通过伙伴算法管理的空闲页面

ZONE\_PADDING(\_pad1\_)

spinlock\_t lru\_lock;

struct lruvec lruvec;

unsigned long pages\_scanned; //管理区回收页框时使用的计数器，记录上一次回收一同扫描过的页框

unsigned long flags; /\* zone flags, see below \*/

/\* Zone statistics \*/

atomic\_long\_t vm\_stat[NR\_VM\_ZONE\_STAT\_ITEMS];

unsigned int inactive\_ratio;

ZONE\_PADDING(\_pad2\_)

wait\_queue\_head\_t \* wait\_table; //进程等待的散列表，这些进程正在等待管理区中的某页

unsigned long wait\_table\_hash\_nr\_entries; //散列表数组的大小

unsigned long wait\_table\_bits; //散列表数组的大小对2取log的结果

struct pglist\_data \*zone\_pgdat; //管理区属于的节点

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

unsigned long zone\_start\_pfn; //管理区的起始页号

unsigned long spanned\_pages; //管理区的大小包括洞

unsigned long present\_pages; //管理区的大小不包括洞

unsigned long managed\_pages;

const char \*name; //管理区名字，DMA、NORMAL or HIGHMEM

}

当系统中的可用内存很少时，守护程序kswapd被唤醒释放页面。每个管理区通过数组watermark来决定唤醒还是睡眠kswapd守护进程，这个数组通过下面的枚举来分别代表page\_min, page\_low, page\_high，他们的之间的关系图如下图：

enum zone\_watermarks {

WMARK\_MIN,

WMARK\_LOW,

WMARK\_HIGH,

NR\_WMARK

};



page\_low：当空闲页面数达到pages\_low时，伙伴算法分配器就会唤醒kswapd释放页面。

page\_min：当达到pages\_min时，kswap没唤醒则唤醒，同时同步进程释放内存，如果申请内存远远超过实际内存，就会出现out\_of\_memory。page\_min包含里管理区最小保留内存，因此这是GFP\_ATOMIC还可以分配出内存。

page\_high：当释放页面达到这个值，认为该管理区已经平衡，kswapd开始睡眠。

当多个进程对同一个页面进行IO操作（个人理解觉得更应该是有写操作）时，比如页面换入或换出，为了防止访问数据的不一致性，该页面会被锁住，而其他进程则通过wait\_on\_page()函数被添加到等待队列中。如果每个页面都有等待队列则系统会花费大量的内存存放，linux则是将等待队列存储在管理区的wait\_table散列表中，这样就只有一个等待队列。其简单流程图如下：



当申请分配标志为GFP\_ATOMIC时则不会睡眠，因为该标志不能睡眠的内存分配标志，用在中断处理程序、下半部、持有自旋锁以及其他不能睡眠的地方。

## 3、物理页面

系统中的每个物理页面都由struct page用以记录该页面的状态，结构体包含了很多union，因为现在有slab/slob/slub三种分配器使用其中一种即可，定义如下（include/linux/mm\_types.h）：

struct page {

/\* First double word block \*/

unsigned long flags; //存放页的状态

struct address\_space \*mapping; //如果低位是clear的，则内存映射到文件或设备，它指向文件或设备节点inode的address\_space，或者NULL；如果内存映射到匿名则低位被设置，指向匿名对象objects。

/\* Second double word \*/

struct {

union {

pgoff\_t index; //页面是文件映射的一部分，它就是页面在文件中的偏移；页面是交换高速缓存一部分，它就是在交换地址空间中address\_space的偏移量

void \*freelist; //指向slub/slob第一个空闲对象

bool pfmemalloc;

};

union {

unsigned counters;

struct {

union {

atomic\_t \_mapcount;

struct { /\* SLUB \*/

unsigned inuse:16;

unsigned objects:15;

unsigned frozen:1;

};

int units; /\* SLOB \*/

};

atomic\_t \_count; //页的引用计数

};

};

};

/\* Third double word block \*/

union {

struct list\_head lru; //最近最少使用（LRU）链表的链表首部

struct { /\* slub per cpu partial pages \*/

struct page \*next; /\* Next partial slab \*/

short int pages;

short int pobjects;

};

struct list\_head list; //slob列表

struct slab \*slab\_page; /\* slab fields \*/

};

/\* Remainder is not double word aligned \*/

union {

unsigned long private;

struct kmem\_cache \*slab\_cache; /\* SL[AU]B: Pointer to slab \*/

struct page \*first\_page; /\* Compound tail pages \*/

};

#if defined(WANT\_PAGE\_VIRTUAL)

void \*virtual; /\* Kernel virtual address (NULL if

not kmapped, ie. highmem) \*/

#endif /\* WANT\_PAGE\_VIRTUAL \*/

}

Flags的最高ZONES\_SHIFT位记录该页面所属的管理区。set\_page\_zone函数设置页面的管理区。

## 4、源码解析

linux内核的内存管理分三个阶段。  
A. 启动---->bootmem初始化完成为第一阶段。此阶段只能使用memblock\_reserve函数分配内存。  
B. bootmem初始化完--->buddy完成前。该阶段使用引导内存分配器（boot memory allocator）分配内存。  
C. 全部内存初始化完毕，可以用cache和buddy分配内存。

(1)~(4)点完成第一阶段，该阶段主要在start\_kernel —> setup\_arch函数实现:

### （1）获取总内存

parse\_early\_param —> early\_mem函数根据uboot传递进来的命令行参数mem=size@start计算出起始和内存大小通过arm\_add\_memory添加到meminfo全局数组里。

这里需要注意parse\_early\_param之前setup\_machine\_tags里parse\_tags中的parse\_tag\_mem32也会解析uboot通过tags方法传递进来的系统内存，但是被early\_mem覆盖重新计算。

### （2）计算高端内存

sanity\_check\_meminfo（arch/arm/mm/mmu.c）对meminfo里的内存进行判断，是否需要划分为高端内存。

### （3）计算保留内存块

保留内存块包括内核（数据段，代码段等）、页目录等占用的内存块。通过arm\_memblock\_init（arch/arm/mm/init.c）实现：

1. 将meminfo内存通过memblock\_add加到memblock.memory类型内存块里（包含全部物理内存）

B．通过memblock\_reserve将保留内存添加到memblock.reserved类型内存块里

memblock\_reserve(\_\_pa(\_stext), \_end - \_stext); 内核通过查看System.map可以看到

arm\_mm\_memblock\_reserve(); //页表

最终结果显示如下：

MEMBLOCK configuration:

memory size = 0xfa00000（250M） reserved size = 0x6ba3cc

memory.cnt = 0x1

memory[0x0] [0x00000040000000-0x0000004f9fffff], 0xfa00000 bytes

reserved.cnt = 0x2

reserved[0x0] [0x00000040004000-0x00000040007fff], 0x4000 bytes //页表

reserved[0x1] [0x000000400081c0-0x000000406be58b], 0x6b63cc bytes //内核

在伙伴算法完成之前，内存的分配通过该方式进行。

### （4）paging\_init

该函数arch/arm/mm/mmu.c实现，完成页表设置、管理区zone、设置零页等。

/\*

\* paging\_init() sets up the page tables, initialises the zone memory

\* maps, and sets up the zero page, bad page and bad page tables.

\*/

void \_\_init paging\_init(struct machine\_desc \*mdesc)

{

void \*zero\_page;

memblock\_set\_current\_limit(arm\_lowmem\_limit);

build\_mem\_type\_table(); //建立各种类型页表的属性（内存MEMORY类型，设备DEVICE，中断向量表是HIGH\_VECTORS），根据不同arm体系进行初始化，这是因为地址转换由MMU硬件单元根据页表处理，每个体系的MMU单元有差别（不是很明白）。

/\* 以下见第二章页表管理 \*/

prepare\_page\_table(); //清除在这之前建立的临时页表，以便下面建立正式链表

map\_lowmem(); //为低端内存建立一一的映射表（0~ arm\_lowmem\_limit），存放在swapper\_pg\_dir位置，用到前面的页表类型：MT\_MEMORY

dma\_contiguous\_remap(); //建立DMA映射表，类型为：MT\_MEMORY\_DMA\_READY

devicemaps\_init(mdesc); //完成ffff0000开始的中断向量映射表，完成CPU IO映射表（从0x10000000~0x13190000就是控制器地址空间）、刷新一下TLB

kmap\_init(); //申请kmap高端内存永久映射的页表项，虚拟空间为2M（0xbfe00000 - 0xc0000000），如果未定义高端内存（CONFIG\_HIGHMEM）什么也不做。

tcm\_init(); //do nothing

//关于页表见第二章，此处返回中断向量表0xffff0000在pgd中的偏移量

top\_pmd = pmd\_off\_k(0xffff0000);

/\* allocate the zero page. \*/

zero\_page = early\_alloc(PAGE\_SIZE);

bootmem\_init(); //见下一节重点分析

//一种特殊的页，供初始化为0的数据和写时复制使用

empty\_zero\_page = virt\_to\_page(zero\_page);

\_\_flush\_dcache\_page(NULL, empty\_zero\_page);

}

#### （4-1）bootmem\_init

在arch/arm/mm/init.c中定义，完成buddy管理内存需要的工作。

void \_\_init bootmem\_init(void)

{

unsigned long min, max\_low, max\_high;

max\_low = max\_high = 0;

find\_limits(&min, &max\_low, &max\_high); //最小物理页号，低端内存最大物理页号，高端内存最大物理页号

/\* 申请低端内存所需位图的空间，然后赋值给节点pgdat->bdata（pgdat = NODE\_DATA(0);只有一个节点，实际为一个全局变量contig\_page\_data）并将位图所在内存空间保留（memblock.reserved），bdata->node\_bootmem\_map指向位图空间（bdata 为struct bootmem\_data）；先将memblock.memory内存（所有物理内存）的位图清零—表示未使用，再将memblock.reserved 内存的位图置1—表示使用

node\_bootmem\_map:cf9f7000，该部分内存空间在建立伙伴系统的时候释放，所以放在内存的末端

\*/

arm\_bootmem\_init(min, max\_low);

arm\_memory\_present(); //do nothing

sparse\_init(); //do nothing

arm\_bootmem\_free(min, max\_low, max\_high); //见下一节介绍

max\_low\_pfn = max\_low - PHYS\_PFN\_OFFSET; //低端内存的物理页数目

max\_pfn = max\_high - PHYS\_PFN\_OFFSET; //高端内存的物理页数目

}

#### （4-2）arm\_bootmem\_free

该函数（arch/arm/mm/init.c）先计算zone\_size和zhole\_size，然后调用free\_area\_init\_node（mm/page\_alloc.c），因此这里主要分析free\_area\_init\_node函数：

void \_\_paginginit free\_area\_init\_node(int nid, unsigned long \*zones\_size,

unsigned long node\_start\_pfn, unsigned long \*zholes\_size)

{

pg\_data\_t \*pgdat = NODE\_DATA(nid);

pgdat->node\_id = nid;

pgdat->node\_start\_pfn = node\_start\_pfn;

init\_zone\_allows\_reclaim(nid);

//计算节点的node\_spanned\_pages和node\_present\_pages，没有高端内存两者相等

calculate\_node\_totalpages(pgdat, zones\_size, zholes\_size);

//申请所有物理页描述结构体（struct page）所需要的内存空间，mem\_map = NODE\_DATA(0)->node\_mem\_map指向该空间

alloc\_node\_mem\_map(pgdat); //空间大小：page size:200000 //2M

printk("free\_area\_init\_node: node %d, pgdat %08lx, node\_mem\_map %08lx\n",

nid, (unsigned long)pgdat, (unsigned long)pgdat->node\_mem\_map);

free\_area\_init\_node: node 0, pgdat c06535c0（内核数据区）, node\_mem\_map c06bf000（内核bss后面申请的一个内存地址）

free\_area\_init\_core(pgdat, zones\_size, zholes\_size); //见下节

}

#### （4-3）free\_area\_init\_core

该函数（mm/page\_alloc.c）实现如下：

static void \_\_paginginit free\_area\_init\_core(struct pglist\_data \*pgdat,

unsigned long \*zones\_size, unsigned long \*zholes\_size)

{

enum zone\_type j;

int nid = pgdat->node\_id;

unsigned long zone\_start\_pfn = pgdat->node\_start\_pfn;

int ret;

//初始化自旋锁和相关等待队列

pgdat\_resize\_init(pgdat);

init\_waitqueue\_head(&pgdat->kswapd\_wait);

init\_waitqueue\_head(&pgdat->pfmemalloc\_wait);

pgdat\_page\_cgroup\_init(pgdat);

//初始化各个管理区

for (j = 0; j < MAX\_NR\_ZONES; j++) {

struct zone \*zone = pgdat->node\_zones + j;

unsigned long size, realsize, freesize, memmap\_pages;

/\* 重新计算管理区的大小（减去struct page所占用的内存空间）\*/

size = zone\_spanned\_pages\_in\_node(nid, j, zones\_size);

realsize = freesize = size - zone\_absent\_pages\_in\_node(nid, j,

zholes\_size);

memmap\_pages = calc\_memmap\_size(size, realsize);

if (freesize >= memmap\_pages) {

freesize -= memmap\_pages;

}

if (!is\_highmem\_idx(j))

nr\_kernel\_pages += freesize;

/\* Charge for highmem memmap if there are enough kernel pages \*/

else if (nr\_kernel\_pages > memmap\_pages \* 2)

nr\_kernel\_pages -= memmap\_pages;

nr\_all\_pages += freesize;

zone->spanned\_pages = size;

zone->present\_pages = realsize;

/\*

\* Set an approximate value for lowmem here, it will be adjusted

\* when the bootmem allocator frees pages into the buddy system.

\* And all highmem pages will be managed by the buddy system.

\*/

zone->managed\_pages = is\_highmem\_idx(j) ? realsize : freesize;

zone->name = zone\_names[j];

//初始化其他自旋锁

spin\_lock\_init(&zone->lock);

spin\_lock\_init(&zone->lru\_lock);

zone\_seqlock\_init(zone);

//指向节点

zone->zone\_pgdat = pgdat;

zone\_pcp\_init(zone); //初始化CPU页面缓存

lruvec\_init(&zone->lruvec); //初始化lru

if (!size)

continue;

set\_pageblock\_order();

setup\_usemap(pgdat, zone, zone\_start\_pfn, size);

//初始化该管理区的wait\_table和free\_area(伙伴算法)

ret = init\_currently\_empty\_zone(zone, zone\_start\_pfn,

size, MEMMAP\_EARLY);

BUG\_ON(ret);

//初始化该管理区的所有物理页结构体struct page

memmap\_init(size, nid, j, zone\_start\_pfn);

zone\_start\_pfn += size;

}

}

### （5）setup\_per\_cpu\_areas

在init/main.c的start\_kernel调用，在mm/percpu.c中定义，只有在SMP系统下才有用，在UP不做任何处理。

该函数主要是为每个处理器设置per-cpu数据区域，per\_cpu数据由各个CPU独立使用，即使不锁访问，十分有效。per-cpu数据按照不同的CPU类型使用，以将性能低下引发的缓存一致性问题减小到最小。

### （6）build\_all\_zonelists

在init/main.c的start\_kernel调用，在mm/page\_alloc.c中定义。初始化每个节点内的zonelists。

### （7）mm\_init 设置内存分配器

在init/main.c中定义，主要用于设设置伙伴算法内存分配器。源码如下：

static void \_\_init mm\_init(void)

{

/\*

\* page\_cgroup requires contiguous pages,

\* bigger than MAX\_ORDER unless SPARSEMEM.

\*/

page\_cgroup\_init\_flatmem();

mem\_init(); //见下面

kmem\_cache\_init(); //建立kmem\_cache和kmem\_cache\_node两个高速缓存（slab分配器使用）及kmalloc数组高速缓存。

percpu\_init\_late();

pgtable\_cache\_init(); //do nothing

vmalloc\_init(); //vmalloc分配的内存虚拟地址连续，而物理地址无需连续，这里初始vmalloc要用的相关链表等准备工作。

}

void \_\_init mem\_init(void)

{

unsigned long reserved\_pages, free\_pages;

struct memblock\_region \*reg;

int i;

max\_mapnr = pfn\_to\_page(max\_pfn + PHYS\_PFN\_OFFSET) - mem\_map;

//以下两者都是释放空闲内存到伙伴系统：memblock的空闲内存，后者是bootmem的空闲内存，其实两者有交叠，伙伴系统会尝试与前后连续页框组成更大的页框块。

/\* bootmem分配器核心就是node\_bootmem\_map这个位图，每一位代表这个node的一个页，当需要分配时就会扫描这个位图，然后获取一段物理页框进行分配，一般都会从开始处向后进行分配，并没有什么特殊的算法在其中。而伙伴系统初始化时页会根据这个位图，将位图中空闲的页释放回到伙伴系统中，而已经分配出去的页则不会在初始化阶段释放回伙伴系统，不过有可能会在系统运行过程中释放回伙伴系统中。\*/

free\_unused\_memmap(&meminfo); //根据代码只有一个bank，所以没做啥事情

totalram\_pages += free\_all\_bootmem();

//释放高端内存到伙伴系统

free\_highpages();

reserved\_pages = free\_pages = 0;

//统计空闲内存和保留内存并打印出来

for\_each\_bank(i, &meminfo) {

struct membank \*bank = &meminfo.bank[i];

unsigned int pfn1, pfn2;

struct page \*page, \*end;

pfn1 = bank\_pfn\_start(bank);

pfn2 = bank\_pfn\_end(bank);

page = pfn\_to\_page(pfn1);

end = pfn\_to\_page(pfn2 - 1) + 1;

do {

if (PageReserved(page))

reserved\_pages++;

else if (!page\_count(page))

free\_pages++;

page++;

} while (page < end);

}

}

### （8）伙伴算法建立后物理内存分布

经过上面源码分析后，下图是该阶段物理内存分布情况：



4000：内核全局页表swapper\_pg\_dir起始地址，共16K

8000~6be58c：内核存放空间（text\data\bss等）

6bf000~6bf000+200000：存放所有页框描述数据结构struct page

f9f7000：存放早期bootmem分配内存使用的位图空间起始地址，建立伙伴算法后会释放掉

916000~fa00000：空闲页表，全部有伙伴系统管理

前面的9304K保留内存永远不释放，也不归伙伴系统管理

内核信息打印如下：

Memory: 250MB = 250MB total

Memory: 246696k/246696k available, 9304k reserved, 0

Virtual kernel memory layout:

vector : 0xffff0000 - 0xffff1000 ( 4 kB)

fixmap : 0xfff00000 - 0xfffe0000 ( 896 kB)

vmalloc : 0xd0000000 - 0xff000000 ( 752 MB)

lowmem : 0xc0000000 - 0xcfa00000 ( 250 MB)

pkmap : 0xbfe00000 - 0xc0000000 ( 2 MB)

modules : 0xbf000000 - 0xbfe00000 ( 14 MB)

.text : 0xc0008000 - 0xc05e85c8 (6018 kB)

.init : 0xc05e9000 - 0xc0617ec0 ( 188 kB)

.data : 0xc0618000 - 0xc06542c0 ( 241 kB)

.bss : 0xc06542c0 - 0xc06be58c ( 425 kB)

# 二、页表管理

linux采用了一种同时适用于32位和64位系统的普通分页模型。对于像32位arm系统来说两级页表已经足够，但64位系统需要更多数量的分页级别。2.6.10版本以前，linux采用三级分页的模型，从2.6.11版本开始采用4级分页模型。4种页表如下：

页全局目录（pgd）

页上级目录（pud）

页中间目录（pmd）

页表（pte）

其分页模型如下：



图中没有显示具体位数，主要是每一部分大小与体系结构相关。

对于没有启用物理地址扩展的32位系统使用两级页表，下面主要介绍基于32位的arm系统。

启动了物理地址扩展的32位系统使用三级页表。

64位系统使用三级还是四级分页取决与硬件对线性地址的位划分。

以下分析arm的二级页表：

## 1、四级页表与二级页表

linux模型是四级页表，而32位的arm是使用二级页表，之间如何对应起来：

pgd = pgd\_offset\_k(addr) = init\_mm.pgd + ((addr)>> PGDIR\_SHIFT)

#define pgd\_index(addr) ((addr) >> PGDIR\_SHIFT)

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

#define pgd\_offset\_k(addr) pgd\_offset(&init\_mm, addr)

pud\_t \*pud = pud\_offset(pgd, addr) = pgd

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

{

return (pud\_t \*)pgd;

}

pmd\_t \*pmd = pmd\_offset(pud, addr) = pud = pgd

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

{

return (pmd\_t \*)pud;

}

pte\_t \*pte = start\_pte + pte\_index(addr);

#define pte\_index(addr) (((addr) >> PAGE\_SHIFT) & (PTRS\_PER\_PTE - 1))

从上面代码可以看出二级页表时，没有pud和pmd只剩下了pgd和pte。

## 2、页表类型

第一级页表pgd是由虚拟地址的高12bit（bits[31:20]）组成，共有4096个表项，每个表项4个字节，因此4096 \*4 = 16K，这就是上面说的swapper\_pg\_dir总共为16K的原因，每个表项支持1MB，4096\*1MB=4G。每个entry的最低2bit用来区分具体是什么种类的页表项，2bit可以区分4种格式页表项。

第一级页表具体每种表项的结构如下：



第二级页表具体每种表项的结构如下：



large page：64K，small page：4K，tiny page: 1K

arm体系第一级页表主要用到了section entry和coarse page table类型，第二级页表用到了small page类型。在第一级页表中前者在linux内核启动的初始化阶段，建立临时页表使用；后者则是正常工作使用（见下一节分析）。

arch/arm/kernel/head.S中\_\_create\_page\_tables创建了section entry类型临时页表，供linux内核初始化执行环境，一个表项可以映射1MB的物理空间，mmu硬件执行虚拟地址转物理地址的过程如下：



## 3、arm 二级页表

上面建立的临时页表在paging\_init中被prepare\_page\_table清除，然后map\_lowmem建立正常工作的二级页表（create\_mapping），该函数为物理内存从0到lowmem\_limit建立一个一一映射的映射表，就是物理地址和虚拟地址偏差一个固定偏移量PAGE\_OFFSET。而有高端内存时，则通过VMALLOC\_START~VMALLOC\_END线性空间进行动态映射，建立mmu硬件使用的页表，访问完后，将映射清除线性空间释放，达到访问高端内存目的。

一级页表项：4096项16K，二级页表项256项1K，每个表项指向4K 空间（一个page），mmu将虚拟空间转化为物理空间如下：



TLB：存放最近转化后的映射表，可以认为是一个页表的子集(一个高速硬件cache)，加速查找不必每次都需要MMU进行硬件转化。

CPU通过地址来访问内存中的单元，如果CPU没有MMU，或者有MMU但没有启动，那么CPU内核在取指令或者访问内存时发出的地址(此时必须是物理地址，假如是虚拟地址，那么当前的动作无效)将直接传到CPU芯片的外部地址引脚上，直接被内存芯片(物理内存)接收，这时候的地址就是物理地址。如果CPU启用了MMU(一般是在bootloader中的eboot阶段的进入main()函数的时候启用)，CPU内核发出的地址将被MMU截获，这时候从CPU到MMU的地址称为虚拟地址，而MMU将这个VA翻译成为PA发到CPU芯片的外部地址引脚上，也就是将VA映射到PA中。MMU将VA映射到PA是以页(page)为单位的，对于32位的CPU，通常一页为4k，物理内存中的一个物理页面称页为一个页框(page frame)。虚拟地址空间划分成称为页（page）的单位,而相应的物理地址空间也被进行划分，单位是页框(frame).页和页框的大小必须相同

对于所有进程kernel部分的页表都是一样的，在内核do\_fork的时候，申请pgd空间后，先清0，然后将拷贝init\_mm内核部分的页表项到创建进程的页表项；而用户部分每个进程有自己的页表从而达到进程地址空间独立。

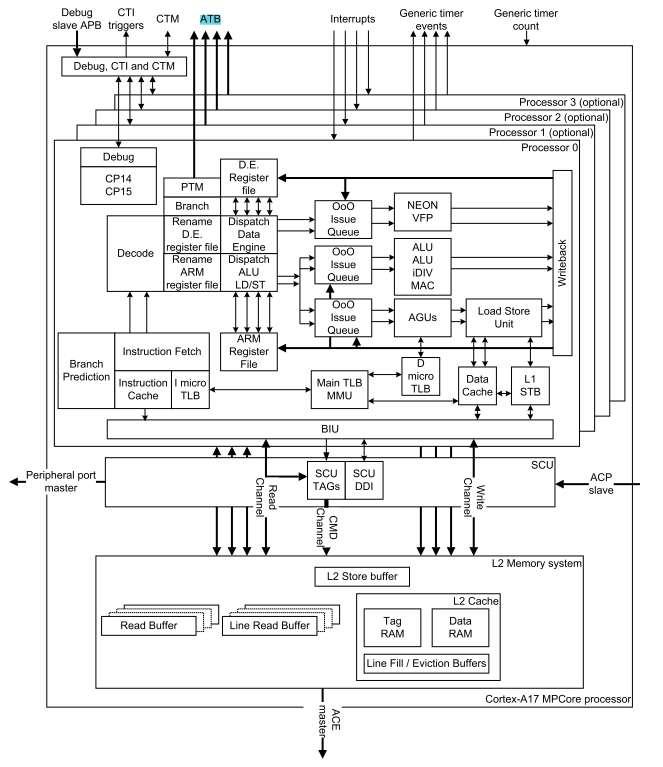
## 4、最新内核优化

上面是arm典型的mmu映射框图，linux在此基础上进行了重要调整（细节看源码）：

第一级页表从4096个项变成2048个项，每个项从4个字节变成8个字节，还是16K大小。

第二级页表从1个256项变成2个256项，每项还是4个字节，这样总计256 \* 2 \* 4 = 2KB，放置在page页的下半部，而上部分放置对应的linux内存管理系统使用页表，mmu硬件是不会去使用它，刚好占满4KB，不浪费空间。

ARM Cortex-A17的拓扑结构：



# 三、物理页面分配（伙伴算法）

内存碎片通常分为外部碎片和内部碎片两种。外部碎片指内存全部是分散小块而无法满足大块内存申请的需求；内部碎片指申请到一页或几页内存后只使用了部分内存造成其他内存浪费。linux因此引入了伙伴算法解决外部碎片问题，slab方式解决内部碎片问题，本章主要讲解伙伴算法，下一章讲解slab层。

伙伴算法是一个结合了2的方幂分配器与空闲缓冲区合并技术的分配方案。内存被分成了含有很多页面的大块，每一个块都是2的方幂个页面大小。如果找不到想要的块，上一级的大块会被分成两部分，这两部分彼此就成了伙伴，其中一半被用来分配而另外一半空闲放入本级空闲链表，这些块会继续被二分直至产生一个所需大小的块。当一个块被最终释放时，其伙伴将被检测出来，如果空闲就合并两者。

## 1、管理空闲块

伙伴系统维护空闲页面所组成的块，每一块都是2的方幂个页面。每个管理区struct zone中struct free\_area free\_area[MAX\_ORDER]数组就用于管理伙伴算法：

#define MAX\_ORDER CONFIG\_FORCE\_MAX\_ZONEORDER =11

struct free\_area {

struct list\_head free\_list[MIGRATE\_TYPES]; //空闲链表

unsigned long nr\_free; //空闲链表个数

};

MAX\_ORDER = 11意味这最大支持的连续物理快2^11 \* 4K = 8M。其示意图如下：



由于本arm体系结构只有一个ZONE\_NORMAL管理区，在内核伙伴算法初始化完成后，其伙伴算法的空闲链表数目如下：

free\_area:0, nr\_free:2

free\_area:1, nr\_free:0

free\_area:2, nr\_free:2

free\_area:3, nr\_free:2

free\_area:4, nr\_free:1

free\_area:5, nr\_free:2

free\_area:6, nr\_free:2

free\_area:7, nr\_free:2

free\_area:8, nr\_free:1

free\_area:9, nr\_free:1

free\_area:10, nr\_free:1

free\_area:11, nr\_free:29

## 2、操作函数

（1）alloc\_page

#define alloc\_page(gfp\_mask) alloc\_pages(gfp\_mask, 0)

分配一个页面并返回一个页地址

（2）alloc\_pages

#define alloc\_pages(gfp\_mask, order) \

alloc\_pages\_node(numa\_node\_id(), gfp\_mask, order)

分配2^order数量的页面并返回第一个页地址

（3）page\_address

该函数返回一个指针，指向给定物理页当前所在的逻辑地址

（4）get\_zeroed\_page

unsigned long get\_zeroed\_page(gfp\_t gfp\_mask)

分配一个页面，对其初始化成0值，返回一个虚拟地址

（5）\_\_get\_free\_pages

unsigned long \_\_get\_free\_pages(gfp\_t gfp\_mask, unsigned int order);

分配2^order数量的页面并返回一个虚拟地址

（6）\_\_get\_free\_page

#define \_\_get\_free\_page(gfp\_mask) \

\_\_get\_free\_pages((gfp\_mask), 0)

分配一个页面并返回虚拟地址

（7）\_\_free\_pages

void \_\_free\_pages(struct page \*page, unsigned int order);

从给定页面中释放2^order页面大小的块

（8）free\_pages

void free\_pages(unsigned long addr, unsigned int order);

从给定虚拟地址空间释放2^order页面大小的块

（9）\_\_free\_page

#define \_\_free\_page(page) \_\_free\_pages((page), 0)

从给定页面中释放1个页面

（10）free\_page

#define free\_page(addr) free\_pages((addr), 0)

从给定的虚拟地址空间释放一个页面

释放需要注意的是：只能释放当初申请的页，如果传递了错误的参数，可能会导致系统崩溃。

# 四、slab层

上一章主要讲述以页为最小单位进行内存分配的伙伴管理算法，较大程度上避免了内存碎片问题。而实际上对内存的申请却不是每次都申请一个页面的（比如文件节点，任务描述符等结构体内存），通常是远小于一个内存页面的大小，此外更可能会频繁地申请释放这些内存。对于这种情况，每次分配小于一个页面的都统一分配一个页面的空间是过于浪费且不切实际的，因此必须充分利用未被使用的空闲空间，同时也要避免过多地访问操作页面分配。基于该问题的考虑，内核需要一个缓冲池对小块内存进行有效的管理起来，于是就有了slab内存分配算法。每次小块内存的分配优先来自于该内存分配器，小块内存的释放也是先缓存至该内存分配器，留作下次申请时进行分配，避免了频繁分配和释放小块内存所带来的额外负载。而这些被管理的小块内存在管理算法中被视之为“对象”

slab/slub/slob（本系统用了slub）：

slab是基础，是最早从Sun OS那引进的；

slob是被改进的slab，占用资源少，使用内存较少的嵌入式设备

slub是在slab上进行的改进简化，在大型机上表现出色，并且能更好的适应large NUMA系统；SLUB相对于SLAB有5%-10%的性能提升和减少50%的内存占用

slab分配器有三个基本目标：

A．减少伙伴系统分配小块内存时所产生的内部碎片

B．把进程使用的对象缓存起来，减少分配、初始化以及释放对象的时间开销

C．调整对象以更好的使用功能L1和L2硬件高速缓存

## 1、高速缓存

slab分配器形象得说就是先由伙伴算法申请部分空闲内存空间，然后slab按照相同数据类型大小对申请的内存进行分割，最后再用一些数据结构进行管理。这些进行分割后的内存称之为高速缓存。

每一种高速缓存存放相同类型的对象，不同高速缓存组成了slab的高速缓存组，通过链表的形式组织，链表头为slab\_caches。通过/proc/slabinfo可以查看当前系统所有的高速缓存：

# name <active\_objs> <num\_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount> <sharedfactor> : slabdata <active\_slabs> <num\_slabs> <sharedavail>

kmalloc-8192 20 20 8192 4 8 : tunables 0 0 0 : slabdata 5 5 0

kmem\_cache\_node 192 192 64 64 1 : tunables 0 0 0 : slabdata 3 3 0

kmem\_cache 128 128 128 32 1 : tunables 0 0 0 : slabdata 4 4 0

name：高速缓存名字

active\_objs：正在使用的对象数目

num\_objs：总共对象数目

objsize：每个对象大小

objperslab：每个slab对象数目

pageperslab：每个slab需要的pages数目

active\_slabs：活动的slab数目

num\_slabs：slab数目

从上面可知，每种高速缓存由1个或多个slab组成，而每个slab又有一页或多页组成，最终被划分为n个对象。如图所示：



## 2、slab

### （1）数据结构描述

从上面看高速缓存和slab划分很清晰，实际代码则比较模糊，主要通过kmem\_cache的数据结构来描述每种slab。该结构定义如下（include/linux/slub\_def.h）：

struct kmem\_cache {

struct kmem\_cache\_cpu \_\_percpu \*cpu\_slab; //per CPU slab结构，用于各个CPU的缓存管理

/\* Used for retriving partial slabs etc \*/

unsigned long flags;

unsigned long min\_partial;

int size; /\* 对象大小，包括meta data元数据 \*/

int object\_size; /\* slab对象纯大小 \*/

int offset; /\* 空闲对象的指针偏移 \*/

int cpu\_partial; /\* 每个CPU持有量 \*/

struct kmem\_cache\_order\_objects oo; //存放分配给slab页框的阶数（高16位）和slab中对象数量（低16位）

/\* Allocation and freeing of slabs \*/

struct kmem\_cache\_order\_objects max;

struct kmem\_cache\_order\_objects min;

gfp\_t allocflags; /\* 申请页面时使用的GFP标识 \*/

int refcount; /\* 缓冲区计数器，当用户请求创建新的缓冲区时SLUB分配器重用已创建的相似大小的缓冲区从而减少缓冲区个数 \*/

void (\*ctor)(void \*); //创建对象的回调函数

int inuse; /\* 元数据meta data偏移量 \*/

int align; /\* 对齐值 \*/

int reserved; /\* Reserved bytes at the end of slabs \*/

const char \*name; /\* slab缓存名称 \*/

struct list\_head list; /\* slab caches管理链表 \*/

#ifdef CONFIG\_SYSFS

struct kobject kobj; /\* For sysfs \*/

#endif

struct kmem\_cache\_node \*node[MAX\_NUMNODES]; //各个内存管理节点的slub信息

};

其中cpu\_slab的结构类型是kmem\_cache\_cpu，每个CPU类型数据，各个CPU都有自己独立的一个结构，用于管理本地的对象缓存。定义如下：

struct kmem\_cache\_cpu {

void \*\*freelist; /\* 空闲对象队列的指针 \*/

unsigned long tid; /\* Globally unique transaction id 标识CPU，保证只有一个且在正确的CPU上申请 \*/

struct page \*page; /\* 指向slab对象来源的内存页面 \*/

struct page \*partial; /\* 指向曾分配完所有的对象，但当前已回收至少一个对象的page \*/

};

node用于管理节点所有对象的slab缓冲区，定义如下：

struct kmem\_cache\_node {

spinlock\_t list\_lock;

#ifdef CONFIG\_SLUB

unsigned long nr\_partial; //本节点的partial slab的数目

struct list\_head partial; //partial slab的双向循环队列

#ifdef CONFIG\_SLUB\_DEBUG

atomic\_long\_t nr\_slabs; //slab总数

atomic\_long\_t total\_objects; //slab对象数目

struct list\_head full; //slab full列表

#endif

#endif

};

Slub分配管理中，每个CPU都有自己的缓存管理（即kmem\_cache\_cpu数据结构管理）；而每个node节点也有自己的缓存管理（即kmem\_cache\_node数据结构管理）。

分配对象：

A．当前CPU缓存有满足申请要求的对象时，将会首先从kmem\_cache\_cpu的空闲链表freelist将对象分配出去。

B．如果对象不够时，将会向伙伴管理算法中申请内存页面，申请来的页面将会先填充到node节点中，然后从node节点取出对象到CPU的缓存空闲链表中

C．如果原来申请的node节点A的对象，现在改为申请node节点B的，那么将会把node节点A的对象释放后再申请。

释放对象：

A．会先将对象释放到CPU上面，如果释放的对象恰好与CPU的缓存来自相同的页面，则直接添加到列表

B．如果释放的对象不是当前CPU缓存的页面，则会把当前的CPU缓存对象放到node节点上面，然后再把该对象释放到本地的cache中

为了避免过多的空闲对象缓存在管理框架中，slub设置的阈值，如果空闲对象个数达到了峰值，将会把当前缓存释放到node节点中，当node节点也过了阈值，将会把node节点的对象释放到伙伴管理算法中。

其实现框图（CPU和node管理细节？）如下：



### （2）slab初始化

在start\_kernel中mm\_init的kmem\_cache\_init实现了slab的初始化，它是在实现伙伴算法之后进行初始化。kmem\_cache\_init在slab.c、slob.c和slub.c都有实现，不同算法其初始化各异，本系统主要用了slub分配算法，因此这里分析slub.c中的实现。其代码原型如下：

void \_\_init kmem\_cache\_init(void)

{

static \_\_initdata struct kmem\_cache boot\_kmem\_cache,

boot\_kmem\_cache\_node;

if (debug\_guardpage\_minorder())

slub\_max\_order = 0;

/\* 由于在这之前只是初始化好伙伴算法，因而无法使用slab分配高速缓存kmem\_cache结构体。这里使用定义了一个静态变量变量boot\_kmem\_cache\_node和boot\_kmem\_cache来临时管理slab。kmem\_cache：主要用于kmem\_cache\_create创建高速缓存时，从该高速缓存分配对象kmem\_cache，描述待创建的高速缓存 \*/

kmem\_cache\_node = &boot\_kmem\_cache\_node;

kmem\_cache = &boot\_kmem\_cache;

/\* 函数用于创建分配算法缓存，主要是把上面两个变量结构初始化 \*/

create\_boot\_cache(kmem\_cache\_node, "kmem\_cache\_node",

sizeof(struct kmem\_cache\_node), SLAB\_HWCACHE\_ALIGN);

register\_hotmemory\_notifier(&slab\_memory\_callback\_nb);

/\* Able to allocate the per node structures \*/

slab\_state = PARTIAL;

create\_boot\_cache(kmem\_cache, "kmem\_cache",

offsetof(struct kmem\_cache, node) +

nr\_node\_ids \* sizeof(struct kmem\_cache\_node \*),

SLAB\_HWCACHE\_ALIGN);

/\* 上面临时初始化后就可以进行slab分配kmem\_cache对象。该函数主要作用是将临时kmem\_cache和kmem\_cache\_node向最终的迁移，并修正相关指针，使其指向最终的kmem\_cache和kmem\_cache\_node，并将两者添加到slab\_caches全局链表中 \*/

kmem\_cache = bootstrap(&boot\_kmem\_cache);

kmem\_cache\_node = bootstrap(&boot\_kmem\_cache\_node);

/\* Now we can use the kmem\_cache to allocate kmalloc slabs \*/

/\* 初始化一批后期内存中需要用到的不同大小的slab缓存（即kmalloc），可以通过/proc/slabinfo查看当前kmalloc支持的大小{64,128,192,256,512,1024,4096,8192} \*/

create\_kmalloc\_caches(0);

#ifdef CONFIG\_SMP

register\_cpu\_notifier(&slab\_notifier);

#endif

printk(KERN\_INFO

"SLUB: HWalign=%d, Order=%d-%d, MinObjects=%d,"

" CPUs=%d, Nodes=%d\n",

cache\_line\_size(),

slub\_min\_order, slub\_max\_order, slub\_min\_objects,

nr\_cpu\_ids, nr\_node\_ids);

}

上面初始化完后，就可以创建自己的高速缓存，可以只用kmalloc申请内存空间。

### （3）操作函数

A．kmem\_cache\_create

创建一个新的高速缓存

B．kmem\_cache\_destroy

销毁一个高速缓存

C．kmem\_cache\_alloc

从高速缓存中分配对象，当高速缓存中空间不足时，会调用alloc\_slab\_page申请一个新的slab页。

D．kmem\_cache\_free

释放一个对象

### （4）slab着色

在slub分配算法中已经看不到着色相关信息，也可能是水平有限未找到。但是这里还是讲一下slab着色原理，有助于对硬件cache理解。

假定L1 data cache为32K，cache line 64B，就有512个cache line(指一次性可以读取/写入的数据量)。CPU访问时先访问硬件cache，没有命中时才访问内存。

硬件cache对应到内存的位置不是任意的：

cache line0 对应到内存地址：0~63，32K~32K+63，…

cache line1 对应到内存地址：64~127，32K+64~32K+127，…

cache line2 对应到内存地址：128~191，32K+128~32K+191，…

…

cache line512 对应到内存地址：32704~32767，32K+32703~32K+32767，…

从上可以看出假定slab对象A在0地址，对象B在64地址，而对象C在32K地址，那么对象A和对象C则会使用相同的硬件cache line0，如果频繁的访问A和C那么就会不停的切入和换出A和C，导致命中效率低下。所谓着色就是将C进行偏移存放在cache line2，这样就不会进行频繁切换，提高命中效率。当然如果cache已经用完，那么进行着色效果也比价差。

## 3、kmalloc和kfree

kmalloc是基于slab/slob/slub分配算法上实现，如前面章节描述kmalloc存在不同固定大小的kmalloc-N的高速缓存。kmalloc在slab/slob/slub上都有实现，这里主要描述slub的实现方式：

static \_\_always\_inline void \*kmalloc(size\_t size, gfp\_t flags)

{

if (\_\_builtin\_constant\_p(size)) { //gcc内建函数，用于判断一个值是否为编译时常量

if (size > KMALLOC\_MAX\_CACHE\_SIZE) //大于8192（即kmalloc最大cache）通过kmalloc\_large函数申请，该方式通过\_\_get\_free\_pages接口，即通过buddy伙伴算法申请所需内存空间（默认最大是8M）

return kmalloc\_large(size, flags);

if (!(flags & GFP\_DMA)) {

int index = kmalloc\_index(size);

if (!index)

return ZERO\_SIZE\_PTR;

//如果<=8192则通过kmalloc cache进行分配

return kmem\_cache\_alloc\_trace(kmalloc\_caches[index],

flags, size);

}

}

return \_\_kmalloc(size, flags);//也是上面的结合，只是在size非常量情况下。

}

size为表示申请的空间大小，而flags则表示分配标志。分配标志众多，每个标志标识特定的bit位，可以通过|进行组合，常见的有如下：

GFP\_KERNEL：首选标志，内核内存分配，会阻塞引起睡眠

GFP\_ATOMIC：不会引起睡眠，但会使用紧急内存池，一般用于中断处理等不能睡眠的地方

从上可以看出kmalloc实现较为简单，分配说的的内存不仅虚拟地址是连续空间，物理地址上也是连续空间。

kfree则是用于释放由kmalloc申请的内存块，在slab/slob/slub都有对应的实现，这里主要展示slub的实现：

void kfree(const void \*x)

{

struct page \*page;

void \*object = (void \*)x;

trace\_kfree(\_RET\_IP\_, x);

if (unlikely(ZERO\_OR\_NULL\_PTR(x))) //判断是否为NULL

return;

page = virt\_to\_head\_page(x); //将虚拟地址转化成对应的页地址

if (unlikely(!PageSlab(page))) { //是否属于slab管理页，不属于则进入if处理

BUG\_ON(!PageCompound(page));

kmemleak\_free(x); //对该虚拟地址进行释放前处理

\_\_free\_memcg\_kmem\_pages(page, compound\_order(page)); // 调用\_\_free\_pages释放页到伙伴算法管理里

return;

}

//如果属于slab管理页，就使用slab方式释放对象

slab\_free(page->slab\_cache, page, object, \_RET\_IP\_);

}

如果释放的内存不是kmalloc分配的，或者想要释放的内存早就被释放了，那么调用该函数会导致严重的后果。注意，调用kfree(NULL)是安全的。

# 五、高端物理内存与非连续内存分配

## 1、高端物理内存

在第一章已经讲过，在usr/kernel为3:1的情况下，在一台32位的体系结构上最多只有896MB的内存可以直接访问，而超过部分就只能通过映射后进行访问。而64位机器有足够多的虚拟地址空间，就不会存在这个问题。

结合第一章线性空间布局，内核空间映射情况如下(以本系统为例)：



直接内存映射区：PAGE\_OFFSET ~ PAGE\_OFFSET + highmem（最大0~896M）: 0xc0000000 - 0xcfa00000。

动态映射区：VMALLOC\_START ~ VMALLOC\_END ： 0xd0000000 - 0xff000000

永久映射区：PAGE\_OFFSET-2M ~ PAGE\_OFFSET：0xbfe00000 - 0xc0000000

固定映射区：FIXADDR\_START ~ FIXADDR\_TOP：0xfff00000 - 0xfffe0000

（1）动态映射区

通过vmalloc/vfree实现，见下一节。

（2）永久映射区

用于将高端内存长久映射到内存虚拟地址。通过一下函数实现：

void \*kmap(struct page \*page)

将一个给定页映射到内核地址空间，一般用于高端内存也可以用于低端内存。如果page对应的是低端内存中的一页，函数只会单纯的返回该页的虚拟地址。如果页位于高端内存，则会建立永久映射，再返回地址。该函数可以睡眠只用于进程上下文中。

该区域只有2M数量有限，当不再需要高端内存时，应该通过下面函数解除映射：

void kunmap(struct page \*page)

（3）固定映射区

主要解决持久映射不能用于中断处理程序而增加的临时内核映射。通过下面函数实现：

void \*kmap\_atomic(struct page \*page)

void \_\_kunmap\_atomic(void \*kvaddr)

## 2、非连续内存分配

伙伴算法解决外部碎片问题，slab算法解决内部碎片问题，但是在使用过程中还是会产生碎片，从而导致申请大块连续内存将可能持续失败。因此，引入了不连续页面管理算法即vmalloc机制。

vmalloc和kmalloc工作方式类似，只是前者虚拟地址连续而物理地址不一定连续，后者虚拟地址和物理地址都连续。vmalloc为了把物理上不连续的页转化成虚拟地址空间连续的页，必须专门建立页表项进行一个一个映射，直接导致比直接映射大的多的TLB抖动，影响性能。内核一般使用kmalloc，系统/proc/vmallocinfo可以查看当前通过vmalloc申请的映射区域：

…

0xbf214000-0xbf222000 57344 module\_alloc\_update\_bounds+0xc/0x5c pages=13 vmalloc

0xbf225000-0xbf228000 12288 module\_alloc\_update\_bounds+0xc/0x5c pages=2 vmalloc

0xd000a000-0xd004b000 266240 atomic\_pool\_init+0x0/0x108 phys=4f500000 user

0xd004b000-0xd0057000 49152 cramfs\_uncompress\_init+0x30/0x64 pages=11 vmalloc

…

从上面可以看出，加载模块时也使用vmalloc机制，但是映射的虚拟地址不在VMALLOC\_START ~ VMALLOC\_END，而是专门的模块地址：MODULE\_VADDR ~ PKMAP\_BASE（PAGE\_OFFSET -16M ~ PAGE\_OFFSET-2M）共14M大小，其他申请的虚拟地址都在VMALLOC\_START ~ VMALLOC\_END。

### （1）数据结构描述

非连续内存区通过vm\_struct结构体描述：

struct vm\_struct {

struct vm\_struct \*next; //指向下一个vm\_struct结构体

void \*addr; //非连续内存区的虚拟地址空间起始地址

unsigned long size; //申请的内存区大小+page\_size（page\_size保留安全间隙）

unsigned long flags; //映射的内存类型VM\_ALLOC(vmalloc) or VM\_IOREMAP(ioremap)

struct page \*\*pages; //直线nr\_pages数组指针，该数组由指向页描述符指针组成即每个不连续的物理页

unsigned int nr\_pages; //页个数

phys\_addr\_t phys\_addr; //映射硬件设备时I/O共享内存，否则一般都为0

const void \*caller;

};

非连续内存区虚拟空间管理通过vmap\_area结构体描述：

struct vmap\_area {

unsigned long va\_start; //虚拟空间起始地址

unsigned long va\_end; //虚拟空间结束地址

unsigned long flags; //映射的内存类型

struct rb\_node rb\_node; //红黑树管理虚拟地址，根据地址排序

struct list\_head list; //链表方式管理虚拟地址，根据地址排序

struct list\_head purge\_list; /\* "lazy purge" list \*/

struct vm\_struct \*vm; //指向非连续内存区描述符

struct rcu\_head rcu\_head;

};

两者关系图如下:



### （2）初始化

在start\_kernal的mm\_init通过vmalloc\_init实现初始化，mm/vmalloc.c中定义如下：

void \_\_init vmalloc\_init(void)

{

struct vmap\_area \*va;

struct vm\_struct \*tmp;

int i;

for\_each\_possible\_cpu(i) {

struct vmap\_block\_queue \*vbq;

struct vfree\_deferred \*p;

vbq = &per\_cpu(vmap\_block\_queue, i);

spin\_lock\_init(&vbq->lock);

INIT\_LIST\_HEAD(&vbq->free);

p = &per\_cpu(vfree\_deferred, i);

init\_llist\_head(&p->list);

INIT\_WORK(&p->wq, free\_work);

}

/\* Import existing vmlist entries. \*/

for (tmp = vmlist; tmp; tmp = tmp->next) {

va = kzalloc(sizeof(struct vmap\_area), GFP\_NOWAIT);

va->flags = VM\_VM\_AREA;

va->va\_start = (unsigned long)tmp->addr;

va->va\_end = va->va\_start + tmp->size;

va->vm = tmp;

\_\_insert\_vmap\_area(va);

}

vmap\_area\_pcpu\_hole = VMALLOC\_END;

vmap\_initialized = true;

}

该函数先是遍历每CPU的vmap\_block\_queue和vfree\_deferred变量及初始化。其中vmap\_block\_queue是非连续内存块队列管理结构，主要是队列以及对应的保护锁；vfree\_deferred是vmalloc的内存延迟释放管理，除了队列初始外，还创建了一个free\_work()工作队列用于异步释放内存。接着，将已经存在于vmlist链表的各项非连续区通过\_\_insert\_vmap\_area插入到非连续内存块的管理中。

### （3）vmalloc解析

vmalloc用于申请非连续区域，当有高端内存时先从高端内存申请，没有高端内存再从低端内存中申请。

void \*vmalloc(unsigned long size)

{

return \_\_vmalloc\_node\_flags(size, NUMA\_NO\_NODE,

GFP\_KERNEL | \_\_GFP\_HIGHMEM);

}

\_\_vmalloc\_node\_flags -> \_\_vmalloc\_node -> \_\_vmalloc\_node\_range是最终的实现，其源码如下：

void \*\_\_vmalloc\_node\_range(unsigned long size, unsigned long align,

unsigned long start, unsigned long end, gfp\_t gfp\_mask,

pgprot\_t prot, int node, const void \*caller)

{

struct vm\_struct \*area;

void \*addr;

unsigned long real\_size = size;

size = PAGE\_ALIGN(size);

if (!size || (size >> PAGE\_SHIFT) > totalram\_pages)

goto fail;

/\* 申请虚拟地址空间，并添加到地址排序的红黑树中进行管理 \*/

area = \_\_get\_vm\_area\_node(size, align, VM\_ALLOC | VM\_UNLIST,

start, end, node, gfp\_mask, caller);

if (!area)

goto fail;

/\* 申请对应的物理内存：首先计算需要的物理页数nr\_pages及存储等量页面的页描述符数组空间大小pages；接着，根据页面数循环申请物理页面空间；最后对申请的页面进行页表更新 \*/

addr = \_\_vmalloc\_area\_node(area, gfp\_mask, prot, node, caller);

if (!addr)

return NULL;

clear\_vm\_unlist(area);

//内存泄露监测

kmemleak\_alloc(addr, real\_size, 3, gfp\_mask);

return addr;

fail:

return NULL;

}

需要注意的一点：所有进程都共享内核空间，因此这里更新的是内核的主页表即init\_mm所在的swap\_pg\_dir页表，进程则是拷贝它。

（4）vfree

vfree用于释放有vmalloc申请的非连续空间，是vmalloc的倒序操作。其实现如下:

void vfree(const void \*addr)

{

BUG\_ON(in\_nmi());

kmemleak\_free(addr);

if (!addr)

return;

if (unlikely(in\_interrupt())) {

struct vfree\_deferred \*p = &\_\_get\_cpu\_var(vfree\_deferred);

llist\_add((struct llist\_node \*)addr, &p->list);

schedule\_work(&p->wq);

} else

\_\_vunmap(addr, 1);

}

先撤销内存泄漏监测；如果当前释放操作在中断中，那么将释放的内存空间加入到当前的CPU的vfree\_deferred管理列表中，继而通过schedule\_work唤醒free\_work工作队列，对内存进行异步释放操作；如果不在中断，直接通过\_\_vunmap()进行内存释放：删除红黑树对应的虚拟地址空间管理区，释放物理页面，释放管理对象页面。

# 附录 A

内存管理参考博客：http://m.blog.chinaunix.net/uid/26859697.html

Slab着色参考博客：http://blog.csdn.net/zqy2000zqy/article/details/1137895