课程内容

日期	知识模块	知识点
2月20日	课程介绍	Linux 内核基本结构、Linux的历史、开源基础知识简介、驱动程序介绍
2月27日	实验课一	内核编译，内核补丁
3月5日	内核编程基础知识概述	内核调试技术、模块编程、源代码阅读工具、linux启动过程、git简介
3月12日	实验课2	内核调试
3月19日 3月26日	进程管理与调度	Linux进程基本概念、进程的生命周期、进程上下文切换、Linux 进程调度策略、调度算法、调度相关的调用
4月2日	实验课3	提取进程信息
4月9日	系统调用、中断处理	系统调用内核支持机制、系统调用实现、Linux中断处理、下半部
4月16日	实验课4	添加系统调用、显示系统缺页次数
4月23日	内核同步	原子操作、自旋锁、RCU、内存屏障等linux内核同步机制
4月30日	内存管理1	内存寻址、Linux物理内存和虚拟内存的组织、伙伴系统、vmalloc
5月14日	内存管理2	Slab分配器、进程地址空间
5月21日	实验课5	观察内存映射、逻辑地址与物理地址的对应
5月28日	文件系统	Linux虚拟文件系统、Ext2/Ext3/Ext4文件系统结构与特性
6月4日	Linux设备驱动基础字符设备驱动程序设计	Linux设备驱动基础、字符设备创建和加载、字符设备的操作、对字符设备进行poll 和select的实现、字符设备访问控制、IOCTL、阻塞IO、异步事件等
	基于linux的容器平台技术概述+实验课6	虚拟化技术与容器、Docker概述、Kubernetes概述 实验：Docker对容器的资源限制
6月11日	报告课	期末课程报告

Agenda

1. **Memory Addressing**
2. **Introduction to Linux Physical and Virtual Memory**
3. **Allocators**
   a) **vmalloc(Noncontiguous memory area management)**
   b) **Physical Page Allocation**
   c) **sla/ub**
4. **Process address space**

Slab allocator

- 目的：提高**内存分配**性能，减少**内部碎片**。
- 思想：把**经常使用**的数据结构当成**对象**，连续地存放在**缓存**中。分配某种数据结构或者对象的内存就是从 slab cache 中“抓”一个对象出来；释放某种数据结构或对象就是把对象再放到 slab cache 中，但不是将其完全地释放掉。内核周期性地扫描slab cache，释放空slab对应的页框。
  - 避免了单独分配一块内存产生的**内部碎片**。
  - 避免了每次都使用 buddy allocator 来分配和释放页面，从而在一定程度上**提高了效率**。

Slab allocator试图在几个基本原则之间寻求一种平衡：

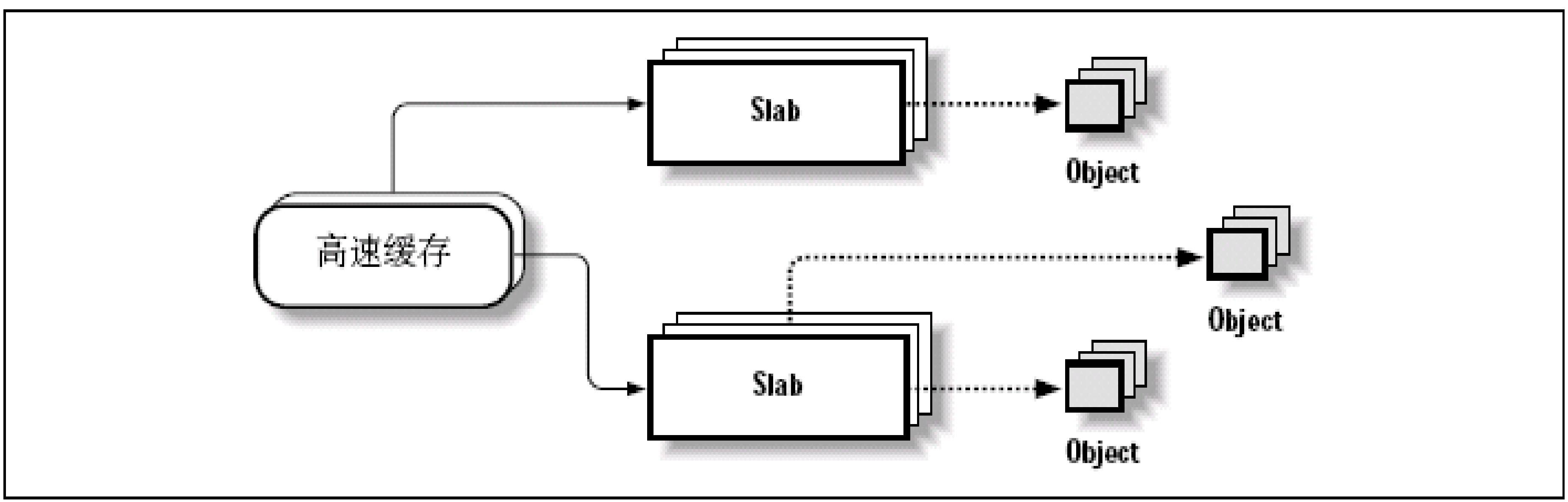
- 频繁使用的数据结构也会频繁分配和释放，因此应当缓存他们；
- 频繁分配和回收会导致内存碎片，为了避免，数据结构在缓存中连续存放，释放的时候放回缓存，不会导致碎片；
- 回收的对象可以立即投入下次分配，提高性能；
- 如果分配器知道对象大小、页大小和总的缓存大小等，有利于决策；
- 如果让部分缓存具有per-cpu属性，则分配和释放就可以不用加自旋锁；
- 如果分配器是与NUMA相关，可从相同的内存节点为请求者分配；

Linux的**slab**层在设计和实现时充分考虑了上述原则。

三层结构

- Cache
- Slab (>= 1 page frame, contiguous)
- Object

Cache ——高速缓存
Object - - - - - 对象



2.1
Slab allocator Caches

两种cache

slab.c

- **General cache**
  - **kmem_cache:** 它的对象是其它 **cache** 的描述符。由 **kmem_cache_boot** 在系统初始化时静态描述，由全局变量 **kmem_cache**指向。
  - 其它通用**cache**在系统初始化期间调用**kmem_cache_init()** 来建立，通用**cache**使用**kmalloc()**分配对象。
- **Specific cache**
  - 内核其他部分**(**频繁被使用的对象**)**使用的 **cache**。
  - **kmem_cache_create()** 用来创建专用的**cache**。

从 **/proc/slabinfo** 可以获得通用和专用 **cache** 的名字和其他的一些属性。

/mm/slab_common.c

所有的**cache**在**slab_caches**链表里 LIST_HEAD(slab_caches);

Slab information

```
root@ubuntu:/home/andrew# head /proc/slabinfo
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab>
 : tunables <limit> <batchcount> <sharedfactor> : slabdata <active_slabs> <num_s
labs> <sharedavail>
ext4_groupinfo_4k    392    392    144   56    2 : tunables    0    0    0 : sla
bdata      7      7      0
ip6-frags              0      0    216   37    2 : tunables    0    0    0 : sla
bdata      0      0      0
UDPLITEv6              0      0   1088   30    8 : tunables    0    0    0 : sla
bdata      0      0      0
UDPv6                 30     30   1088   30    8 : tunables    0    0    0 : sla
bdata      1      1      0
tw_sock_TCPv6          0      0    280   58    4 : tunables    0    0    0 : sla
bdata      0      0      0
TCPv6                 14     14   2240   14    8 : tunables    0    0    0 : sla
bdata      1      1      0
kcopyd_job             0      0   3312    9    8 : tunables    0    0    0 : sla
bdata      0      0      0
dm_uevent              0      0   2632   12    8 : tunables    0    0    0 : sla
bdata      0      0      0
root@ubuntu:/home/andrew#
```

Cache Descriptor

Slab_def.h

无论是通用cache还是专用cache，都需要cache描述符来描述cache

```
struct kmem_cache {
 //per-CPU数据，每次分配、释放期间访问
        struct array_cache __percpu *cpu_cache;
 /* 1) Cache tunables. Protected by cache_chain_mutex */
        unsigned int batchcount;
        unsigned int limit;
        unsigned int shared;

        unsigned int size;
        struct reciprocal_value reciprocal_buffer_size;
 /* 2) touched by every alloc & free from the backend */
        unsigned int flags;          /* constant flags */
        unsigned int num;            /* # of objs per slab */
......

     struct kmem_cache_node *node[MAX_NUMNODES];


};
```

The fields of the kmem_cache descriptor

Type	Name	Description
Struct array_cache __percpu	*cpu_cache	__percpu
unsigned int	batchcount	从本地高速缓存交换的对象的数量
unsigned int	limit	本地高速缓存中空闲对象的数量
unsigned int	shared	是否存在共享CPU高速缓存
unsigned int	size	对象获得实际内存的大小
struct reciprocal_value	reciprocal_buffer_size	buffer_size的倒数，系统中没有使用
unsigned int	num	对象数目
unsigned int	gfporder	每个slab包含的页框数取2为底的对数
gfp_t	allocflags	GFP flags
size_t	colour	slab使用的颜色个数
unsigned int	colour_off	slab中的基本对齐偏移
const char *	name	cache creation/removal */
struct list_head	list	
int	refcount	
int	object_size	
int	align	
kmem_cache_node *	node[MAX_NUMNODES]	kmem_cache_node数组

struct kmem_cache_node

```
/*
 * The slab lists for all objects.
 */
struct kmem_cache_node {
	spinlock_t list_lock;

#ifdef CONFIG_SLAB
	struct list_head slabs_partial;	/* partial list first, better asm code */
	struct list_head slabs_full;
	struct list_head slabs_free;
	unsigned long free_objects;
	unsigned int free_limit;
	unsigned int colour_next;	/* Per-node cache coloring */
	struct array_cache *shared;	/* shared per node */
	struct alien_cache **alien;	/* on other nodes */
	unsigned long next_reap;	/* updated without locking */
	int free_touched;		/* updated without locking */
#endif
```

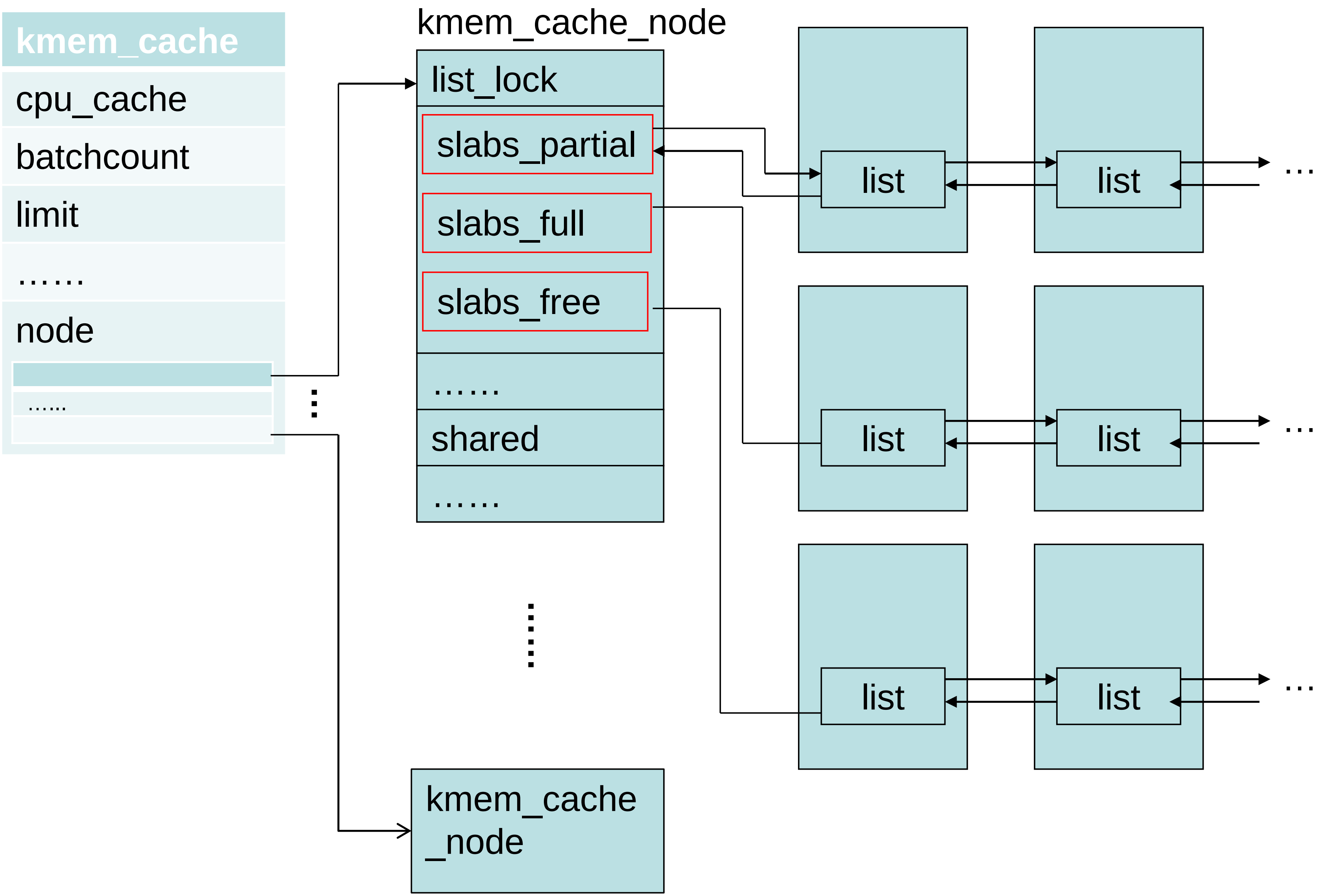
kmem_cache_node中的可用对象的个数

kmem_cache_node中的所有slab的可用对象数上限

```
//编译选项
#ifdef CONFIG_SLUB
        unsigned long nr_partial;
        struct list_head partial;
#ifdef CONFIG_SLUB_DEBUG
        atomic_long_t nr_slabs;
        atomic_long_t total_objects;
        struct list_head full;
#endif
#endif

};
```

struct kmem_cache_node



Kmalloc_caches

Slab_common.c

- **kmalloc/kfree: general cache**
- 工作于 **slab**和**slub**分配器之上
- 内核初始化时，创建一组通用对象的缓冲区。**kmalloc_caches** 数组存放了这些缓冲区的 **kmem_cache** 数据结构的指针**,**其中
  - **kmalloc_caches[0]** 代表的缓冲区专门分配 **kmem_cache_node** 结构
  - **kmalloc_caches[1]** 缓冲区对象大小为**64**，**kmalloc_caches[2]** 缓冲区对象大小为**192**，其余第 **i**（**3-13**）号缓冲区对象大小为 **2^i**
  - 如果请求分配超过物理页面大小的对象，直接调用页框分配器
  - 为了满足老式 **ISA** 设备的需要，内核还使用 **DMA** 内存创建了 通用对象的缓冲区，用 **kmalloc_caches[KMALLOC_DMA]**数组存放相应的 **kmem_cache** 结构。

通用cache

Slab_common.c

```
struct kmem_cache *
kmalloc_caches[NR_KMALLOC_TYPES][KMALLOC_SHIFT_HIGH + 1]
    __ro_after_init = {};
EXPORT_SYMBOL(kmalloc_caches);
```

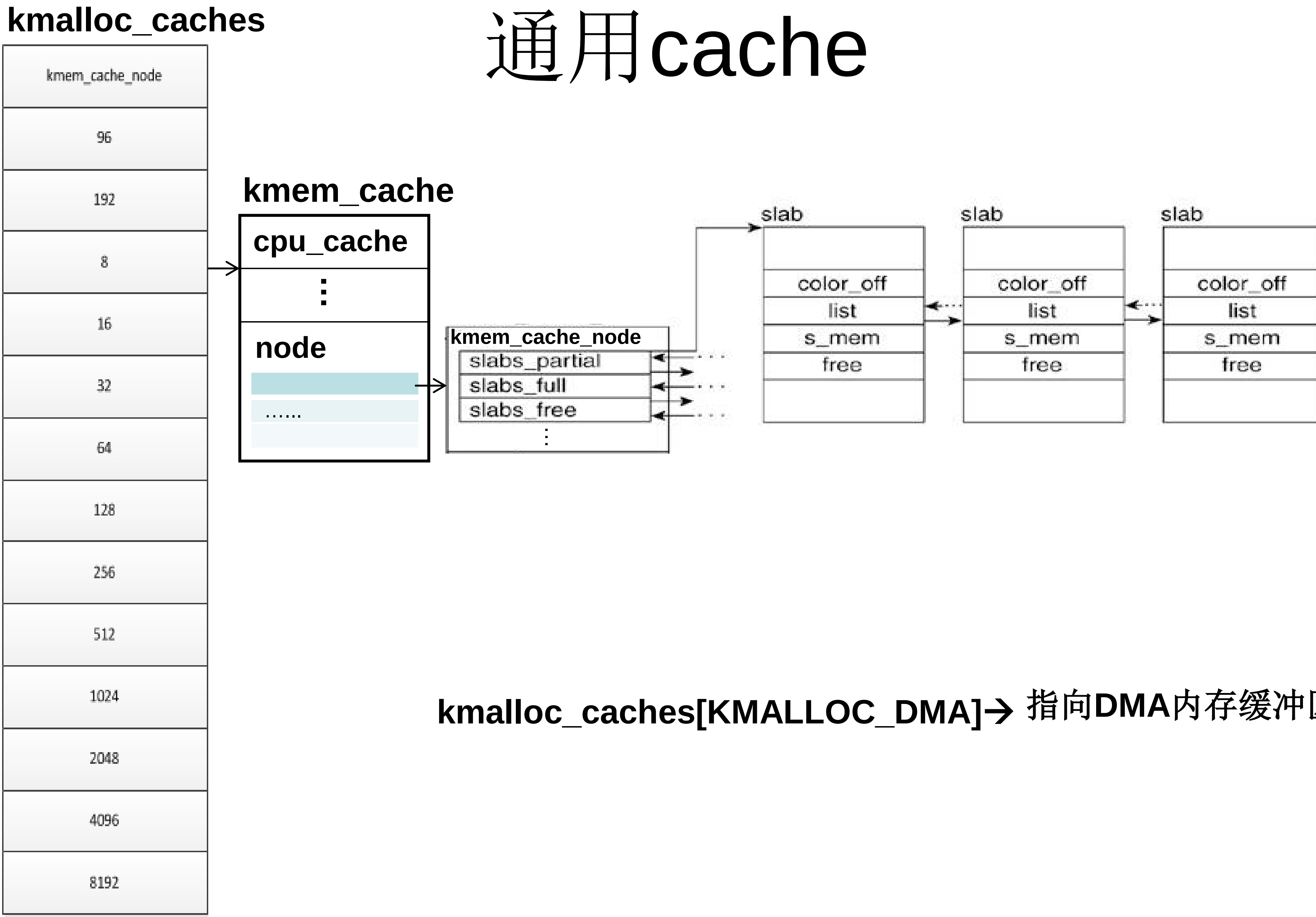
```
enum kmalloc_cache_type {
        KMALLOC_NORMAL = 0,
        KMALLOC_RECLAIM,
#ifdef CONFIG_ZONE_DMA
        KMALLOC_DMA,
#endif
        NR_KMALLOC_TYPES
};
```

kmalloc_caches[KMALLOC_NORMAL]:
指向普通内存缓冲区ZONE_NORMAL

kmalloc_caches[KMALLOC_DMA]:
指向DMA内存缓冲区，ZONE_DMA

slab/slub: **introduce kmalloc-reclaimable caches**
On Wed, Jul 18, 2018 at 03:36:15PM +0200, Vlastimil Babka wrote:
> Kmem caches can be created with a SLAB_RECLAIM_ACCOUNT flag, which indicates
> they contain objects which can be reclaimed under memory pressure (typically
> through a shrinker). This makes the slab pages accounted as NR_SLAB_RECLAIMABLE
> in vmstat, which is reflected also the MemAvailable meminfo counter and in
> overcommit decisions. The slab pages are also allocated with __GFP_RECLAIMABLE,
> which is good for **anti-fragmentation** through grouping pages by mobility.
>
> The generic kmalloc-X caches are created without this flag, but sometimes are
> used also for objects that can be reclaimed, which due to varying size cannot
> have a dedicated kmem cache with SLAB_RECLAIM_ACCOUNT flag. A prominent example
> are dcache external names, which prompted the creation of a new, manually
> managed vmstat counter NR_INDIRECTLY_RECLAIMABLE_BYTES in commit f1782c9bc547
> ("dcache: account external names as indirectly reclaimable memory").
>
> To better handle this and any other similar cases, this patch introduces
> SLAB_RECLAIM_ACCOUNT variants of kmalloc caches, named kmalloc-rcl-X.
> They are used whenever the kmalloc() call passes **__GFP_RECLAIMABLE** among gfp
> flags. They are added to the kmalloc_caches array as a new type. Allocations
> with both __GFP_DMA and __GFP_RECLAIMABLE will use a dma type cache.
>
> This change only applies to SLAB and SLUB, not SLOB. This is fine, since SLOB's
> target are tiny system and this patch does add some overhead of kmem management
> objects.

通用cache



kmalloc_caches[KMALLOC_DMA]→ 指向DMA内存缓冲区

普通内存缓冲区

Local Caches of Free Slab Objects

- **each cache includes a per-CPU data structure called the slab local cache.**
  - **To reduce spin lock contention among processors**
  - **to make better use of the hardware caches**
  - **consisting of a small array of pointers to freed objects**
- **Most allocations and releases of slab objects affect the local cache only**
  - **the slab data structures get involved only when the local cache underflows or overflows.**

```
struct kmem_cache {
        struct array_cache __percpu *cpu_cache;
        ……
```

Per-CPU data structure array_cache

- Array_cache 是 slab local cache，存放指向某些可用的对象的指针

Slab.c

```
struct array_cache {
    unsigned int avail;   /* 在local cache 中的可用对象个数  */
    unsigned int limit;    /*  local cache 可保存的最大指针个数 */
    unsigned int batchcount;  /* 用来清空或填充cache 的Chunk size的大小 */
    unsigned int touched;  /* 如果当前的 local cache 正在被使用，设置为 1 */

   void *entry[];   /*
                     * Must have this definition in here for the proper
                     * alignment of array_cache. Also simplifies accessing
                     * the entries.
                     */
};
```

Slab.c

kmem_cache_boot

```
/* internal cache of cache description objs */
static struct kmem_cache kmem_cache_boot = {
    .batchcount = 1
    .limit = BOOT_CPUCACHE_ENTRIES,
    .shared = 1
    .size = sizeof(struct kmem_cache), /*对象大小 */
    .name = “kmem_cache”, /* cache 的名字*/
};
```

在Local cache 中的最大可用对象数

Cache中的每个对象的大小，这里可以看到，kmem_cache_boot中的对象是 kmem_cache 描述符

Kmem_cache_node初始化

```
static int init_cache_node_node(int node) //初始化对应于某node的kmem_cache_node
{
	int ret;
	struct kmem_cache *cachep;

	list_for_each_entry(cachep, &slab_caches, list) {
		ret = init_cache_node(cachep, node, GFP_KERNEL);
		if (ret)
			return ret;
	}

	return 0;
}
```

```
static int init_cache_node(struct kmem_cache *cachep, int node, gfp_t gfp)
{
        struct kmem_cache_node *n;
        n = get_node(cachep, node);//根据node号找到相应的kmem_cache_node
        if (n) {
                spin_lock_irq(&n->list_lock);
                n->free_limit = (1 + nr_cpus_node(node)) * cachep->batchcount +
                                cachep->num;
                spin_unlock_irq(&n->list_lock);

                return 0;
        }
        n = kmalloc_node(sizeof(struct kmem_cache_node), gfp, node);//分配描述符

        if (!n)         return -ENOMEM;
        kmem_cache_node_init(n); //初始化kmem_cache_node
        ……
        cachep->node[node] = n;

        return 0;
}
```

```
static void kmem_cache_node_init(struct kmem_cache_node *parent)
{               INIT_LIST_HEAD(&parent->slabs_full);
                INIT_LIST_HEAD(&parent->slabs_partial);
                INIT_LIST_HEAD(&parent->slabs_free);
                parent->total_slabs = 0;
                parent->free_slabs = 0;
                parent->shared = NULL;
                parent->alien = NULL;
                parent->colour_next = 0;
                spin_lock_init(&parent->list_lock);
                parent->free_objects = 0;
                parent->free_touched = 0;  }
```

```
static __always_inline void *kmalloc_node(size_t size, gfp_t flags, int node)
{
#ifndef CONFIG_SLOB
        if (__builtin_constant_p(size) &&
                size <= KMALLOC_MAX_CACHE_SIZE) {
                unsigned int i = kmalloc_index(size);

                if (!i)
                        return ZERO_SIZE_PTR;

                return kmem_cache_alloc_node_trace(
                                kmalloc_caches[kmalloc_type(flags)][i],
                                                flags, node, size);
        }
#endif
        return __kmalloc_node(size, flags, node);
}
```

Slab.h

```
//根据size求取索引号
static __always_inline unsigned int kmalloc_index(size_t size)
{
        if (!size)                          return 0;

        if (size <= KMALLOC_MIN_SIZE)                 return KMALLOC_SHIFT_LOW;

        if (KMALLOC_MIN_SIZE <= 32 && size > 64 && size <= 96)           return 1;
        if (KMALLOC_MIN_SIZE <= 64 && size > 128 && size <= 192)         return 2;
        if (size <=          8) return 3;
        if (size <=         16) return 4;
        ……
        if (size <=        512) return 9;
        if (size <=       1024) return 10;
        if (size <=   2 * 1024) return 11;
        if (size <=   4 * 1024) return 12;
        ……
                if (size <= 1024 * 1024) return 20;
        if (size <=  2 * 1024 * 1024) return 21;
        if (size <=  4 * 1024 * 1024) return 22;
        if (size <=  8 * 1024 * 1024) return 23;
        if (size <=  16 * 1024 * 1024) return 24;
        if (size <=  32 * 1024 * 1024) return 25;
        if (size <=  64 * 1024 * 1024) return 26;
        BUG();

        /* Will never be reached. Needed because the compiler may complain */
        return -1;
}
#endif
```

2.2
Slab allocator

Slabs and objects

- Slab是一个或者多个连续的物理页框
- Slab没有额外的描述结构，而是在代表物理页框的 page 结构中加入 s_mem, **freelist**等字段，分别代表第一个对象和第一个空闲对象的指针，所以 slab 的第一个物理页框的 page 结构就可以描述自己。

/include/linux/mm_types.h

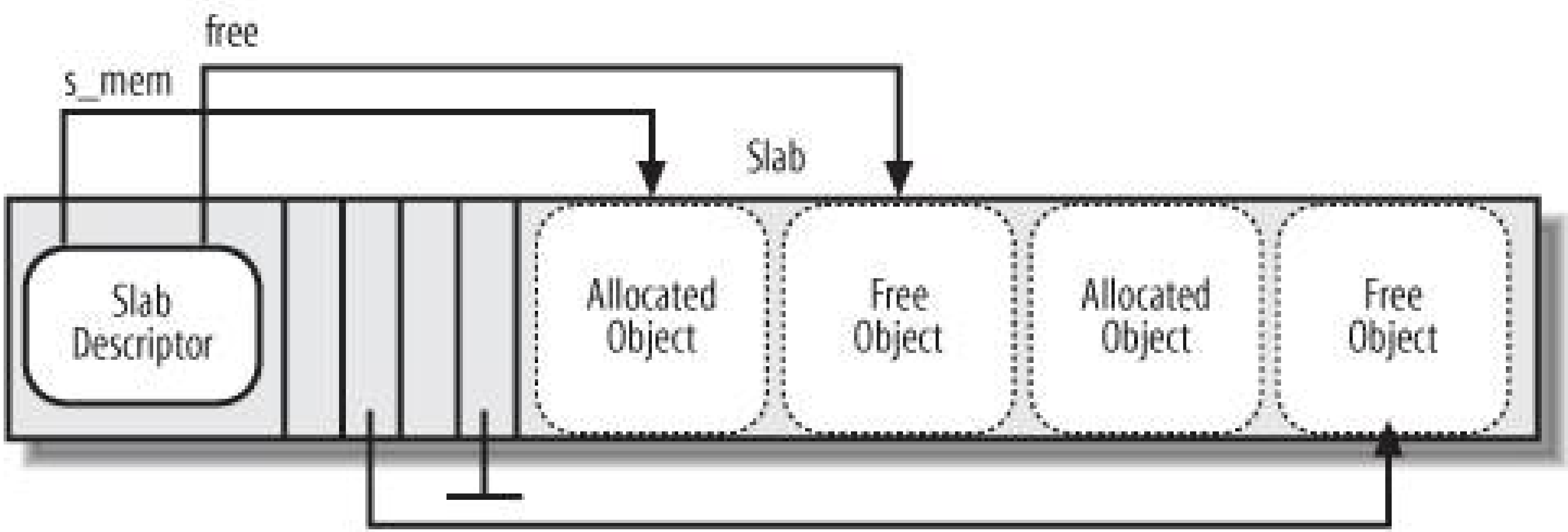
```
struct page {
	……
	struct {		/* slab, slob and slub */
		union {
			struct list_head slab_list;
			struct {	/* Partial pages */
				struct page *next;
				……
			};
		};
		struct kmem_cache *slab_cache; /* not slob */
		/* Double-word boundary */
		void *freelist;			/* first free object */
		union {
			void *s_mem;	/* slab: first object */
			unsigned long counters;		/* SLUB */
			struct {			/* SLUB */
				unsigned inuse:16;
				unsigned objects:15;
				unsigned frozen:1;
			};
		};
	};
	……
}
```

```
struct page {                                              /include/linux/mm_types.h
......
        union {
                /* See page-flags.h for the definition of PAGE_MAPPING_FLAGS */
                struct address_space *mapping;
                void *s_mem;                        /* slab first object */
                atomic_t compound_mapcount;         /* first tail page */
                /* page_deferred_list().next         -- second tail page */
        };
        /* Second double word */
        union {
                pgoff_t index;                /* Our offset within mapping. */
                void *freelist;               /* sl[aou]b first free object */
                /* page_deferred_list().prev        -- second tail page */
        };
        union {
                _slub_counter_t counters;
                unsigned int active;                /* SLAB */
                struct {                      /* SLUB */
                        unsigned inuse:16;
                        unsigned objects:15;
                        unsigned frozen:1;
                };
......
```

对象描述符

- 每个对象都有类型为**kmem_bufctl_t**的一个描述符。
- 是一个无符号整数，只有在对象空闲时才有意义：下一个空闲对象在**slab**中的下标，因此实现了**slab**内部空闲对象的一个简单链表。
  - 空闲对象链表中的最后一个元素的对象描述符用常规值**BUFCTL_END**（**0xffff**）标记

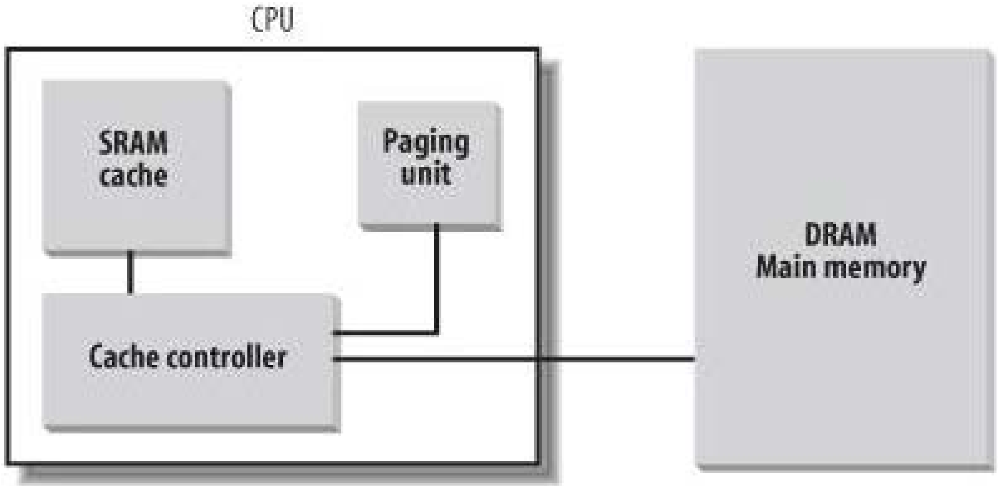
```
typedef unsigned int kmem_bufctl_t;
Index of next free object in the slab,
or BUFCTL_END if there are no free objects left
```

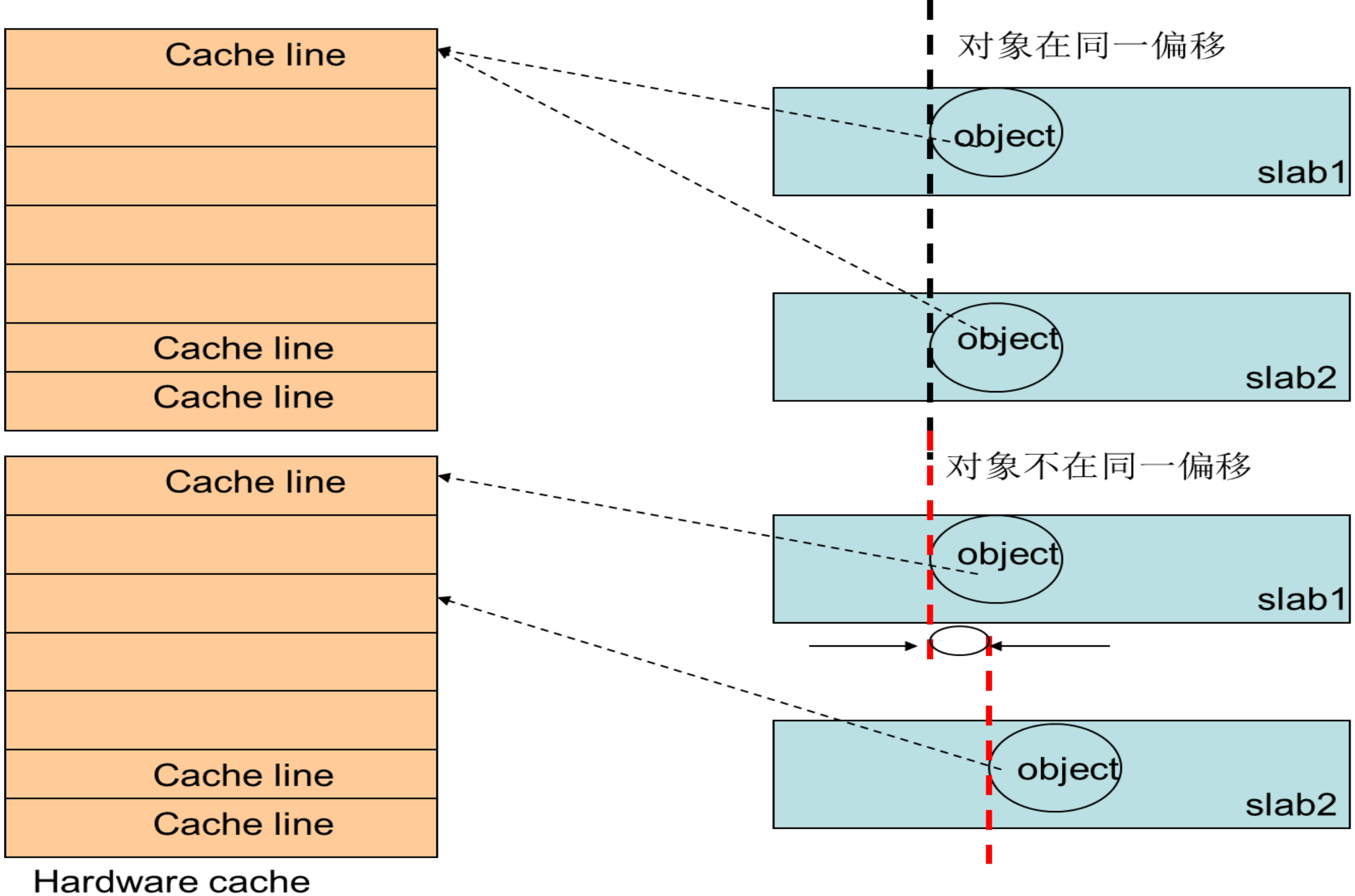


free
s_mem
Slab
Slab
Descriptor
Allocated
Object
Free
Object
Allocated
Object
Free
Object

Slab 着色

- 为什么要 slab 着色？
  - 不同slab中在**<u>相同偏移位置的对象，有可能映射到同一个 hardware cache line 中</u>**。那么 hardware cache就可能把处于同一个cache line 中的对象来回存放或者取出于不同的内存位置，这样就浪费了时间，且其他的 cache line 却没有得到利用。
  - 所以，给 slab 着色，让**不同的slab相同位置的object** 可以映射到**不同的 hardware cache line** 中





拥有不同颜色的 Slab 保存第一个对象于不同的内存位置，且满足对齐限制。

struct kmem_cache

- color: 可用颜色数， 是free/aln, 存放在cache descriptor。
- colour_off: 在 cache descriptor 中， 表示对齐常数 aln。

解释—对齐（align）

- slab分配器所管理的对象可以在内存中进行对齐，也就是说，**存放它们的内存单元的起始物理地址是一个给定常量的倍数**， 通常是2的倍数——alignment factor。
- slab分配器所允许的最大对齐因子是4096。
- **通常情况下，如果内存单元的物理地址是字大小（即计算机的内部内存总线的宽度）对齐的，那么，微机对内存单元的存取会非常快。**因此，缺省情况下，kmem_cache_create()函数根据BYTES_PER_WORD宏所指定的字大小来对齐对象。对于80x86处理器，这个宏产生的值为4，因为字长是32 位。
- 当创建一个新的slab高速缓存时， 就可以让它所包含的对象在第一级硬件高速缓存中对齐。为了做到这点，设置SLAB_HWCACHE_ALIGN高速缓存描述符标志。kmem_cache_create()函数按如下方式处理请求：
  - 如果对象的大小大于高速缓存行 （cache line） 的一半， 就在RAM中根据L1_CACHE_BYTES的倍数（也就是行的开始）对齐对象。
  - 否则，对象的大小就是L1_CACHE_BYTES的因子取整。这可以保证一个小对象不会横跨两个高速缓存行。

slab分配器以内存空间换取访问时间，即通过人为地增加对象的大小来获得较好的高速缓存性能，由此也引起额外的内碎片。

2.3
Slab allocator Functions

Destroy a cache

- destroy a cache and remove it from a cache chain list by invoking kmem_cache_destroy( ).
  - This function is mostly useful to modules that create their own caches when loaded and destroy them when unloaded.
- To avoid wasting memory space, the kernel must destroy all slabs before destroying the cache itself. The kmem_cache_shrink( ) function destroys all the slabs in a cache by invoking slab_destroy( ) iteratively

Releasing a Slab from a Cache

- Slabs can be destroyed in two cases:
  - There are too many free objects in the cache.
  - A **timer** function invoked periodically determines whether there are fully unused slabs that can be released.
- the slab_destroy( ) function is invoked to
  - destroy a slab，release all **objects** in a slab;
  - release the corresponding **page** frames to the zoned page frame allocator.

Allocating a Slab to a Cache

- A newly created cache does not contain a slab and therefore does not contain any free objects. New slabs are assigned to a cache only when **both** of the following are true:
  - A request has been issued to allocate a new object.
  - The cache does not include a free object.
- the slab allocator assigns a new slab to the cache by invoking cache_grow_begin().
  - This function calls kmem_ getpages( ) to obtain from the zoned page frame allocator the group of page frames needed to store a single slab

Interfacing the Slab Allocator with the Zoned Page Frame Allocator

- When the slab allocator creates a new slab, it relies on the zoned page frame allocator to obtain a group of free contiguous page frames. For this purpose, it invokes the kmem_getpages( ) function.
- In the reverse operation, page frames assigned to a slab can be released by invoking the kmem_freepages( ) function.

Allocating a Slab Object

- New objects may be obtained by invoking the kmem_cache_alloc( ) function.
  - parameter cachep points to the cache descriptor from which the new free object must be obtained
  - parameter flags represents the flags to be passed to the zoned page frame allocator functions, should all slabs of the cache be full

Freeing a Slab Object

- The kmem_cache_free( ) function releases an object previously allocated by the slab allocator to some kernel function.
  - Its parameters are cachep, the address of the cache descriptor, and objp, the address of the object to be released.
  - The function checks first whether the local cache has room for an additional pointer to a free object.
  - If so, the pointer is added to the local cache and the function returns.
  - Otherwise it first invokes cache_flusharray( ) to deplete the local cache and then adds the pointer to the local cache.

General Purpose Objects

- <u>Infrequent requests</u> for memory areas are handled through a group of general caches whose objects have geometrically distributed sizes.
- Objects of this type are obtained by invoking the kmalloc( ) function
  - The function first locates the nearest power-of-2 size to the requested size.
  - It then allocates the object, the main parameter is either the cache descriptor for the page frames usable for ISA DMA or the cache descriptor for the "normal" page frames, depending on whether the caller specified the _ _GFP_DMA flag.
- Objects obtained by invoking kmalloc( ) can be released by calling kfree( )

2.4 Slub

Slab 缺点

- 较多复杂的队列管理；
- slab 管理数据的存储开销比较大；
- 缓冲区内存回收比较复杂；
- 对 NUMA 的支持非常复杂；
- 冗余的 Partial 队列；
- 性能调优比较困难；
- 调试功能比较难使用。

Slub allocator

- **Christoph Lameter设计，于2.6.22引入。**
- **特点**
  - **SLUB 分配器简化了kmem_cache，slab 等相关的管理数据结构，摒弃了SLAB 分配器中众多的队列概念，并针对多处理器、NUMA 系统进行优化，从而提高了性能和可扩展性并降低了内存的浪费。**
  - **保留 SLAB 分配器的基本思想：**
    - **每个缓冲区由多个小的 slab 组成，每个 slab 包含固定数目的对象。**
    - **为了保证内核其它模块能够无缝迁移到 SLUB 分配器，SLUB 还保留了原有 SLAB 分配器所有的接口 API 函数。**

Slub_def.h

```
struct kmem_cache {
    struct kmem_cache_cpu _percpu *cpu_slab;
    /* Used for retriving partial slabs etc */
    unsigned long flags;
    unsigned long min_partial; /* minimum num of pgs of node’s partial list*/
    int size;        /*分配给对象的内存大小（可能大于对象的实际大小） */
    int object_size;;                    /*对象的实际大小*/
    int offset;                   /*指向下一个空闲对象的指针的位移*/
    int cpu_partial;           /*Number of per cpu partial objects to keep around */
    struct kmem_cache_order_objects oo;
    /* Allocation and freeing of slabs */
    struct kmem_cache_order_objects max;
    struct kmem_cache_order_objects min;

    gfp_t allocflags;          /*gfp flags to use on each alloc */
    int refcount;                 /* Refcount for slab cache destroy，-1: can’t reuse */
    void (*ctor)(struct kmem_cache *, void *);
```

Slab的物理页个数以及其slab中的对象数，2^order个连续页面（默认oo，不能满足则min）

```
        unsigned int inuse;/* Offset to metadata, generally at the end of the obj */
        unsigned int align;                     /* Alignment */
        unsigned int red_left_pad;  /* Left redzone padding size */
        const char *name;           /* Name (only for display!) */
        struct list_head list;      /* List of slab caches */#ifdef #ifdef
CONFIG_SYSFS
        struct kobject kobj;        /* For sysfs */
        struct work_struct kobj_remove_work;
#endif
……

#ifdef CONFIG_NUMA
        /* defragmentation by allocating from a remote node. */
        int remote_node_defrag_ratio;
#endif
        struct kmem_cache_node *node[MAX_NUMNODES];
};
```

Slab.h

```
struct kmem_cache_node {
                 …
    #ifdef CONFIG_SLUB
    unsigned long nr_partial;
    struct list_head partial;
#ifdef CONFIG_SLUB_DEBUG
    atomic_long_t nr_slabs;
    atomic_long_t total_objects;
    struct list_head full;
#endif
#endif
};
```

slab 的公共缓存队列由 **kmem_cache_node** 结构维护。

- 部分对象被使用的**slab**（可能还有**free**的**slab**，相当于原**partial**链+**free**链）处于**kmem_cache_node**结构的**partial**队列中。
- 全部对象被用完的**slab**处于**kmem_cache_node**结构的**full**队列中，只有调试的时候为了便于查看分配器状态才有此队列
  - 可以从**obj**直接得到**page**结构地址，所以不再维护**full**队列

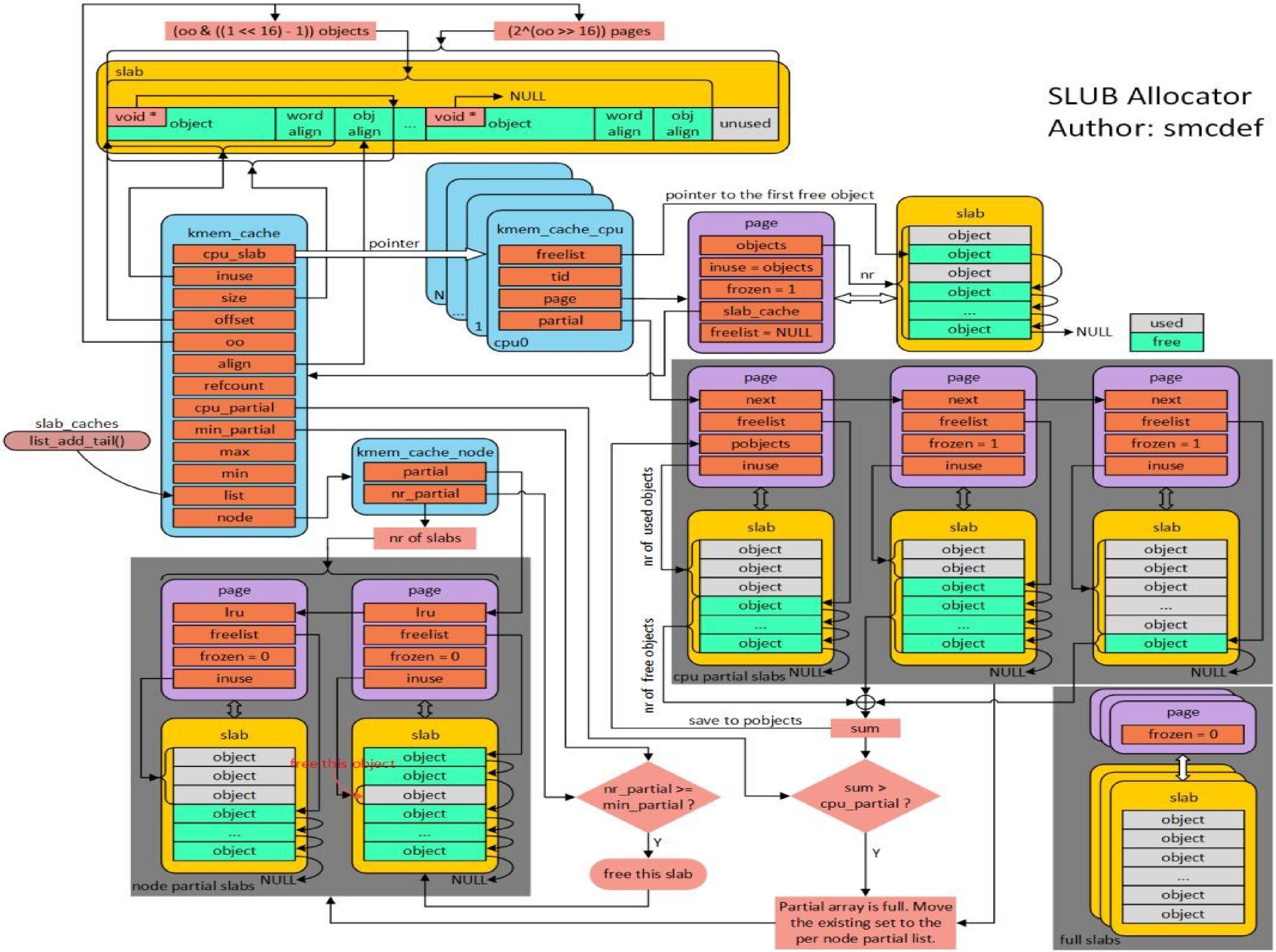
```
struct kmem_cache_cpu {
    void **freelist;    /*空闲对象队列的指针，即第一个空闲对象的指
    针*/
    unsigned long tid; /* Globally unique transaction id */
    struct page *page  /* The slab from which we are allocating */
    struct page *partial;       /* Partially allocated frozen slabs */
    #ifdef CONFIG_SLUB_STATS
    unsigned stat[NR_SLUB_STAT_ITEMS];
#endif
};
```

每个处理器都有一个本地活动cache——cpu_slab，里面有一个活动slab（page指向），处理器申请对象时都是从本地活动slab中申请，slab的第一个空闲对象由freelist指向。如果没有可供申请的对象，则向partial求救。

/include/linux/mm_types.h

```
struct page {
        ……
        struct {        /* slab, slob and slub */
                union {
                        struct list_head slab_list;
                        struct {  /* Partial pages */
                                struct page *next;
                                ……
                        };
                };
                struct kmem_cache *slab_cache; /* not slob */
                /* Double-word boundary */
                void *freelist;                 /* first free object */
                union {
                        void *s_mem;    /* slab: first object */
                        unsigned long counters;         /* SLUB */
                        struct {                        /* SLUB */
                                unsigned inuse:16;  //正在使用的对象数量
                                unsigned objects:15; //总的对象数量
                                unsigned frozen:1; //是否被冻结——在cpu_slab的partial
                                                   //被该cpu独占
                        };
                };
        };……
}
```

slab 没有额外的描述结构，因为 SLUB 分配器在代表物理页框的 page 结构中加入 **freelist**，inuse等字段，分别代表页框中的第一个空闲对象的指针，正在使用的对象数目等，所以 slab 的第一个物理页框的 page 结构就可以描述自己。


SLUB Allocator
Author: smcdef
(oo & ((1 << 16) - 1)) objects
(2^(oo >> 16)) pages
slab
void *
object
word align
obj align
...
NULL
unused
kmem_cache
cpu_slab
inuse
size
offset
oo
align
refcount
cpu_partial
min_partial
max
min
list
node
slab_caches
list_add_tail()
pointer
kmem_cache_cpu
freelist
tid
page
partial
cpu0
N
1
pointer to the first free object
page
objects
inuse = objects
frozen = 1
slab_cache
freelist = NULL
nr
used
free
kmem_cache_node
partial
nr_partial
nr of slabs
next
pobjects
inuse
nr of used objects
nr of free objects
cpu partial slabs
lru
frozen = 0
free this object
node partial slabs
save to pobjects
sum
nr_partial >= min_partial ?
sum > cpu_partial ?
Y
free this slab
Partial array is full. Move the existing set to the per node partial list.
full slabs

cache重用

- **当内核执行请求创建新的缓存 C2 时，Slub 分配机制会先搜索已创建的缓存，如果发现某缓存 C1 的对象大小对齐后等于 C2，且满足一定条件，则重用 C1。**
- **测试表明，这项功能减少了大约 50% 的缓存数目，从而减少了 slab碎片并提高了内存利用率。**
- **该项功能使得着色变得不那么重要，所以在slub中被去掉（也是为了节省page空间占用）。**
- **kmem_cache中的refcount为-1表示不可重用，比如kmem_cache和kmem_cache_node的专有cache不可重用**

start_kernel中在buddy启用之后调用kmem_cache_init进行slab机制初始化：

/mm/slub.c

```
 void __init kmem_cache_init(void)
{          static __initdata struct kmem_cache boot_kmem_cache,
                     boot_kmem_cache_node;
           ……
           kmem_cache_node = &boot_kmem_cache_node;
           kmem_cache = &boot_kmem_cache;        ……
           create_boot_cache(kmem_cache_node, "kmem_cache_node",
                     sizeof(struct kmem_cache_node), SLAB_HWCACHE_ALIGN, 0, 0);
           ……
           create_boot_cache(kmem_cache, “kmem_cache”, //填写属性值，申请内存
                               offsetof(struct kmem_cache, node) +
                                         nr_node_ids * sizeof(struct kmem_cache_node *),
                          SLAB_HWCACHE_ALIGN, 0, 0);

           kmem_cache = bootstrap(&boot_kmem_cache); //静态缓存拷贝，建立链接
           kmem_cache_node = bootstrap(&boot_kmem_cache_node);
           /* Now we can use the kmem_cache to allocate kmalloc slabs */
           setup_kmalloc_cache_index_table();
           create_kmalloc_caches(0);
           …..
};
```

```
static struct kmem_cache * __init bootstrap(struct kmem_cache *static_cache)
{
	int node;
	struct kmem_cache *s = kmem_cache_zalloc(kmem_cache, GFP_NOWAIT);
	struct kmem_cache_node *n;

	memcpy(s, static_cache, kmem_cache->object_size);
	__flush_cpu_slab(s, smp_processor_id());
	for_each_kmem_cache_node(s, node, n) {
		struct page *p;
		list_for_each_entry(p, &n->partial, slab_list)
			p->slab_cache = s;

#ifdef CONFIG_SLUB_DEBUG
		list_for_each_entry(p, &n->full, slab_list)
			p->slab_cache = s;
#endif
	}
	slab_init_memcg_params(s);
	list_add(&s->list, &slab_caches);
	memcg_link_cache(s, NULL);
	return s;
}
```

Slab分配器

分配：

本地active slab $\xrightarrow{\text{没有空闲对象了}}$ 本地partial链 $\xrightarrow{\text{没有空闲对象了}}$ node的 partial链 $\xrightarrow{\text{没有空闲对象了}}$ 新建slab

释放：

属于本地active slab则放回 $\xrightarrow{\text{不属于}}$ 放回到page的freelist里

第一次释放？ →加到partial链

都释放回来了？ →释放slab

kmem_cache_cpu.freelist（kf）和kmem_cache_cpu.page.freelist(kpf):

管理本地cpu的对象　　　　　　　　　　管理page的对象

1、本地active slab的分配和释放都是针对kf的，有可能多个page。

2、所以需要将kpf中的空闲对象都分配给kf才能够进行正常的分配操作。

3、kpf中的空闲对象分配给kf之后，kpf=NULL。

4、active slab有两种后果：

1)还有可供分配的空闲对象（partial的状态），则分配和释放都发生在kf。

2)没有可供分配的空闲对象（full的状态），则被从kp上flush下来，之后只能进行释放的操作。当第一次有对象被释放（pf==NULL?），则将其加到partial链里，以后该slab被释放掉的对象会被陆续加到pf中。当pf中的对象满了，则将此slab释放掉。

kmalloc

- 参数size（申请分配的内存大小）如果是常数，则：
  - 通过size找到合适的cache（kmalloc_slab()）；
  - **通过kmem_cache_alloc()分配**（参数为指定的cache以及用于物理页框分配器的gfpflags，核心函数为**slab_alloc()**）。
- 否则（size是变量或者从DMA区分配） __kmalloc()
  - get_slab()找到合适的cache；
  - 通过**slab_alloc()**分配。

slab_alloc() （**快速分配**，从本地active slab分配）

- 得到本地cpu对应的active slab (即该cache中的本地kmem_cache_cpu)；
- 本地cpu的active slab没有空闲对象（kf为空），则通过__slab_alloc进行**慢速分配**（求助cpu或node的partial链或者新建slab）；
- 否则从本地cpu的active slab（kmem_cache_cpu的freelist）中分配（快速路径）；

__slab_alloc()（慢速分配）

- 如果本地cpu的active slab中没有空闲对象，则求助本地的partial链，找到一个合适的slab，挂载到kmem_cache_cpu上；
- 如果partial链也没有合适的slab，则通过求助node的partial和物理页框分配器新建一个slab，挂载到kmem_cache_cpu上；
- 将新挂载的slab中的第二个空闲对象开始的所有空闲对象（由slab第一个物理页框页描述符page的freelist指向）分配给kmem_cache_cpu的freelist，以后再分配空闲对象可以直接从这里分配；
- 将page的freelist置为NULL，将第一个空闲对象返回。

Kmalloc函数

Slab.h

```
static __always_inline void *kmalloc(size_t size, gfp_t flags)
{           //检测size是变量还是常量, 为常量则执行if
            if (__builtin_constant_p(size)) {
                        //检测申请的大小是否超过1页内存的大小
                        if (size > KMALLOC_MAX_CACHE_SIZE)
                                    return kmalloc_large(size, flags); //调用大块内存分配
#ifndef CONFIG_SLOB
                        index = kmalloc_index(size);   //获取索引号

                        if (!index)
                                    return ZERO_SIZE_PTR;
                        return kmem_cache_alloc_trace(
                                                kmalloc_caches[kmalloc_type(flags)][index],
                                                flags, size);
#endif
            }
            return __kmalloc(size, flags);
}
```

```
void *__kmalloc(size_t size, gfp_t flags)
{
        struct kmem_cache *s;
        void *ret;

        if (unlikely(size > KMALLOC_MAX_CACHE_SIZE))
                return kmalloc_large(size, flags);

        s = kmalloc_slab(size, flags);// //找到合适的cache

        if (unlikely(ZERO_OR_NULL_PTR(s)))
                return s;

        ret = slab_alloc(s, flags, _RET_IP_);

        trace_kmalloc(_RET_IP_, ret, size, s->size, flags);

        ret = kasan_kmalloc(s, ret, size, flags);

        return ret;
}
EXPORT_SYMBOL(__kmalloc);
```

Slab.h

```
//根据size求取索引号
static __always_inline unsigned int kmalloc_index(size_t size)
{
        if (!size)                  return 0;

        if (size <= KMALLOC_MIN_SIZE)           return KMALLOC_SHIFT_LOW;

        if (KMALLOC_MIN_SIZE <= 32 && size > 64 && size <= 96)        return 1;
        if (KMALLOC_MIN_SIZE <= 64 && size > 128 && size <= 192)      return 2;
        if (size <=          8) return 3;
        if (size <=         16) return 4;
        ……
        if (size <=        512) return 9;
        if (size <=       1024) return 10;
        if (size <=   2 * 1024) return 11;
        if (size <=   4 * 1024) return 12;
        ……
                if (size <= 1024 * 1024) return 20;
        if (size <=  2 * 1024 * 1024) return 21;
        if (size <=  4 * 1024 * 1024) return 22;
        if (size <=  8 * 1024 * 1024) return 23;
        if (size <=  16 * 1024 * 1024) return 24;
        if (size <=  32 * 1024 * 1024) return 25;
        if (size <=  64 * 1024 * 1024) return 26;
        BUG();

        /* Will never be reached. Needed because the compiler may complain */
        return -1;
}
#endif
```

Slub.c

```
void *kmem_cache_alloc_trace(struct kmem_cache *s, gfp_t gfpflags,
size_t size)
{
        void *ret = slab_alloc(s, gfpflags, _RET_IP_);
        trace_kmalloc(_RET_IP_, ret, size, s->size, gfpflags);
        kasan_kmalloc(s, ret, size);
        return ret;
}

static __always_inline void *slab_alloc(struct kmem_cache *s,
                gfp_t gfpflags, unsigned long addr)
{
        return slab_alloc_node(s, gfpflags, NUMA_NO_NODE, addr);
}
```

```
static __always_inline void *slab_alloc_node(struct kmem_cache *s,
                        gfp_t gfpflags, int node, unsigned long addr)
{
          void **object;
          struct kmem_cache_cpu *c;
          struct page *page;
          unsigned long tid;

          s = slab_pre_alloc_hook(s, gfpflags);   //预处理，涉及到memcg，返回合适
                                                  //的kmem_cache
          if (!s)
                    return NULL;
redo:
          do {
                    tid = this_cpu_read(s->cpu_slab->tid); /* Globally unique
                                                              transaction id */
                    c = raw_cpu_ptr(s->cpu_slab); //取得本地cpu_slab
          } while (IS_ENABLED(CONFIG_PREEMPT) &&
                     unlikely(tid != READ_ONCE(c->tid)));
```

```
    barrier();
    object = c->freelist;      //获取本地cpu的active slab的空闲对象
    page = c->page;
    if (unlikely(!object || !node_match(page, node))) {
            object = __slab_alloc(s, gfpflags, node, addr, c);   //慢速分配
            stat(s, ALLOC_SLOWPATH);
    } else {
            void *next_object = get_freepointer_safe(s, object);  //下一个object地址
             if (unlikely(!this_cpu_cmpxchg_double(
                                    s->cpu_slab->freelist, s->cpu_slab->tid,
                                    object, tid,
                                    next_object, next_tid(tid)))) {

                    note_cmpxchg_failure("slab_alloc", s, tid);
                    goto redo;
            }
            prefetch_freepointer(s, next_object);
            stat(s, ALLOC_FASTPATH);
    }

    if (unlikely(gfpflags & __GFP_ZERO) && object)
            memset(object, 0, s->object_size);

    slab_post_alloc_hook(s, gfpflags, object);

    return object;
}
```

```
//本地cpu的active slab中没有空闲对象或node不匹配
static void *__slab_alloc(struct kmem_cache *s, gfp_t gfpflags, int
node, unsigned long addr, struct kmem_cache_cpu *c)
{
        void *freelist;
        struct page *page;

        page = c->page;
        if (!page) {  // 本地cpu的active slab为空

                /* if the node is not online or has no normal memory,
                 *just ignore the node constraint */

                if (unlikely(node != NUMA_NO_NODE &&
                             !node_state(node, N_NORMAL_MEMORY)))
                        node = NUMA_NO_NODE;
                goto new_slab; //从partial链找一个slab或者新建一个slab
        }
```

```
redo: //page不为空

        if (unlikely(!node_match(page, node))) {  //如果node不匹配
                /*
                 * same as above but node_match() being false already
                 * implies node != NUMA_NO_NODE
                 */
                if (!node_state(node, N_NORMAL_MEMORY)) {//如果
                                                    //没有normal区内存
                        node = NUMA_NO_NODE;
                        goto redo;
                } else { //不匹配但是有normal内存
                        stat(s, ALLOC_NODE_MISMATCH);
                        deactivate_slab(s, page, c->freelist, c);// Remove
                                                    //the cpu slab，KF→KPF
                        goto new_slab;
                }
        }
```

```
//page不为空且node匹配则

if (unlikely(!pfmemalloc_match(page, gfpflags))) {
        deactivate_slab(s, page, c->freelist, c);
        goto new_slab;
}

/* must check again c->freelist in case of cpu migration or IRQ */
freelist = c->freelist;
if (freelist)
        goto load_freelist; //取对象返回，并处理下一个指针

freelist = get_freelist(s, page);

if (!freelist) {
        c->page = NULL;
        stat(s, DEACTIVATE_BYPASS);
        goto new_slab;
}

stat(s, ALLOC_REFILL);
```

```
load_freelist: //取对象返回，并处理下一个指针
	VM_BUG_ON(!c->page->frozen);
	c->freelist = get_freepointer(s, freelist);  //取出当前对象，返回下一个对象

	c->tid = next_tid(c->tid);

	return freelist;

new_slab:

		if (slub_percpu_partial(c)) {//cpu partial
			page = c->page = slub_percpu_partial(c); //从cpu partial分配
			slub_set_percpu_partial(c, page);
			stat(s, CPU_PARTIAL_ALLOC);
			goto redo;
		}
```

```
        //！ cpu partial
         freelist = new_slab_objects(s, gfpflags, node, &c);   //得到freelist，
                                                               //把KPF给KF
        if (unlikely(!freelist)) {
                 slab_out_of_memory(s, gfpflags, node);
                 return NULL;
         }
        page = c->page;
         if (likely(!kmem_cache_debug(s) && pfmemalloc_match(page, gfpflags)))
                 goto load_freelist;

         /* Only entered in the debug case */
         if (kmem_cache_debug(s) &&
                         !alloc_debug_processing(s, page, freelist, addr))
                 goto new_slab;   /* Slab failed checks. Next slab needed */

         deactivate_slab(s, page, get_freepointer(s, freelist), c);
         return freelist;
}
```

kfree

- 参数x为释放内存的地址；
- 通过x得到其所属物理页框的页结构指针p（如果属于slab则找到其头页框的页结构指针）；
- 该页框不属于slab分配器（通过检查pageslab属性），则将该页框引用计数减一；
- 否则通过**slab_free()**释放。

void kfree(const void *x);

slab_free→do_slab_free

- 如果p就挂载在本地kmem_cache_cpu上，则直接释放到其freelist里（快速释放）；
- 否则通过__slab_free()慢速释放。

Slub.c

```
void kfree(const void *x)
{     struct page *page;
     void *object = (void *)x;
     trace_kfree(_RET_IP_, x);
     if (unlikely(ZERO_OR_NULL_PTR(x)))   return;  //地址检查
     page = virt_to_head_page(x); //得到page结构指针
     if (unlikely(!PageSlab(page))) {  //页面是否属于slab分配器
          unsigned int order = compound_order(page);
          BUG_ON(!PageCompound(page));
          kfree_hook(object);
          mod_node_page_state(page_pgdat(page), NR_SLAB_UNRECLAIMABLE,
                                   -(1 << order));
          __free_pages(page,order);//不属于则该页框引用计数减一
          return;
     }
     slab_free(page->slab_cache, page, object,NULL, 1,  _RET_IP_);
}
EXPORT_SYMBOL(kfree);
```

调试

- 激活“slab_debug”选项，用户就可以很方便地选择单个或一组指定的缓冲区进行动态调试。

Agenda

1. **Memory Addressing**
2. **Introduction to Linux Physical and Virtual Memory**
3. **Allocators**
   a) **vmalloc(Noncontiguous memory area management)**
   b) **Physical Page Allocation**
   c) **sla/ub**
4. **Process address space**

分配内存

- 内核获取内存方式
  - _get_free_pages( ) or alloc_pages( ) .
    - 从分区页框分配器获取内存
  - kmem_cache_alloc( ) or kmalloc( ).
    - 使用slab分配器为专用或通用对象分配内存
  - vmalloc( ) or vmalloc_32( ) .
    - 获取一块非连续内存区.
  - 内核申请内存使用这些简单方法基于以下两个原因:
    - **内核是操作系统中优先级最高的成分**。如果内核申请动态内存，那么必定有正当理由。因此，**没有理由推迟处理这个请求**。
    - **内核信任自己**。所有内核函数都是假定没有错误的，因此内核函数不必插入针对编程错误的保护措施。

分配内存

- 用户模式进程
  - –进程对动态内存的请求被认为是**不紧迫**的。
    - 例如，当一个进程的可执行文件被加载时，进程不可能立刻处理所有的代码页。
    - 同样, 当进程调用 malloc()来获得额外的动态内存，并不意味着进程将很快访问所有获得的额外内存。

  因此,内核总是尽量**推迟**给用户态进程分配动态内存

  .

  –用户进程是**不可信任的**，因此，内核必须能随时准备捕捉用户进程引起的所有寻址错误。

进程地址空间

- 进程的地址空间（Address Space）由允许进程使用的全部线性地址组成。

- 一个进程使用的进程地址空间与另一个进程使用的进程地址空间之间没有关系。

- 内核可以通过增加或删除某些**线性地址区间**来动态修改进程的地址空间。

- 内核通过所谓memory region的资源来表示线性地址空间，它由三部分组成：
  - 起始线性地址
  - 长度
  - 相应访问权限

- 为了提高效率，起始地址和长度都是4096（一页）的整数倍。

some typical situations a process gets new memory regions

- When the user types a command at the console,
  - the shell process **creates a new process** to execute the command.
  - As a result, a fresh address space, and thus a set of memory regions, is assigned to the new process.
- A running process may decide to load an entirely different program.
  - In this case, the process ID remains **unchanged**,
  - but the memory regions used before loading the program are **released**
  - and a new set of memory regions is **assigned** to the process.
- A running process may perform a "memory mapping" on a file (or on a portion of it).
  - In such cases, the kernel assigns **a new memory region** to the process to map the file.

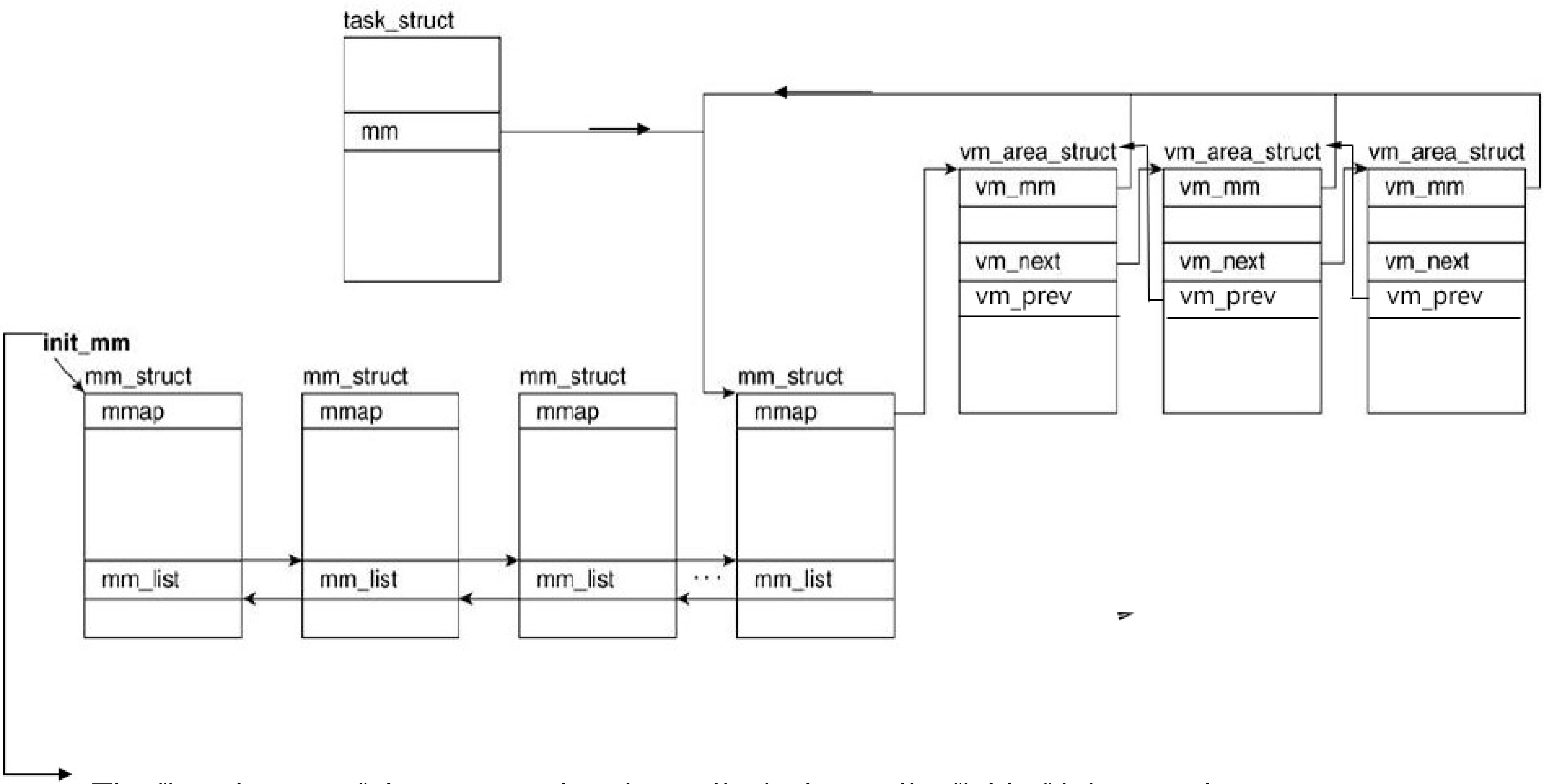
some typical situations a process gets new memory regions

- A process may keep **adding data** on its User Mode **stack** until all addresses in the memory region that map the stack have been used. In this case, the kernel may decide to **expand the size of that memory region**.
- A process may expand its dynamic area (the **heap**) through a function such as malloc( ). As a result, the kernel may decide to **expand the size of the memory region** assigned to the heap.
- A process may **create an IPC-shared memory region** to share data with other cooperating processes. In this case, the kernel **assigns a new memory region** to the process to implement this construct.

与创建、删除memory region相关的系统调用

System call	Description
brk( )	Changes the heap size of the process
execve( )	Loads a new executable file, thus changing the process address space
_exit( )	Terminates the current process and destroys its address space
fork( )	Creates a new process, and thus a new address space
mmap( ), mmap2( )	Creates a memory mapping for a file, thus enlarging the process address space
mremap( )	Expands or shrinks a memory region
remap_file_pages( )	Creates a non-linear mapping for a file (see Chapter 16)
munmap( )	Destroys a memory mapping for a file, thus contracting the process address space
shmat( )	Attaches a shared memory region
shmdt( )	Detaches a shared memory region

Process-Related Memory Structures



The first element of the memory descriptors list is the mmlist field of init_mm, the memory descriptor used by process 0 in the initialization phase.链表的第一个元素是init_mm的mmlist字段，init_mm是初始化阶段进程0使用的内存描述符。

mm field in the process descriptor

- 所有的内存描述符存放在一个双向链表中。
- 进程描述符中的**mm**字段指向进程所拥有的内存描述符，而**active_mm**字段指向进程运行时所使用的内存描述符。
- 对于普通进程，这两个字段存放相同的指针。但是，对于内核线程，由于内核线程不拥有任何内存描述符，因此，它们的**mm**字段总是**NULL**。当内核线程运行时，它的**active_mm**字段被初始化为前一个运行进程的**active_mm**值。

memory descriptor

[{linuxsrcdir}/include/linux/mm_types.h]

Type	Field	Description
struct vm_area_struct *	mmap	Pointer to the head of the list of memory region objects
struct rb_root	mm_rb	Pointer to the root of the red-black tree of memory region objects
u32	vmacache_seqnum	Per-thread vmacache
unsigned long (*) ( )	get_unmapped_area	Method that searches an available linear address interval in the process address space
unsigned long	highest_vm_end	Highest vma end address
unsigned long	mmap_base	Identifies the linear address of the first allocated anonymous memory region or file memory mapping
unsigned long	task_size	Size of task vm space
pgd_t *	pgd	Pointer to the Page Global Directory
atomic_t	mm_users	Secondary usage counter
atomic_t	mm_count	Main usage counter
int	map_count	Number of memory regions
struct rw_semaphore	mmap_sem	Memory regions' read/write semaphore
spinlock_t	page_table_lock	Memory regions' and Page Tables' spin lock
struct list_head	mmlist	Pointers to adjacent elements in the list of memory descriptors
unsigned long	start_code	Initial address of executable code
unsigned long	end_code	Final address of executable code
unsigned long	start_data	Initial address of initialized data

unsigned long	end_data	Final address of initialized data
unsigned long	start_brk	Initial address of the heap
unsigned long	brk	Current final address of the heap
unsigned long	start_stack	Initial address of User Mode stack
unsigned long	arg_start	Initial address of command-line arguments
unsigned long	arg_end	Final address of command-line arguments
unsigned long	env_start	Initial address of environment variables
unsigned long	env_end	Final address of environment variables
unsigned long	rss	Number of page frames allocated to the process
unsigned long	anon_rss	Number of page frames assigned to anonymous memory mappings
unsigned long	total_vm	Size of the process address space (number of pages)
unsigned long	locked_vm	Number of "locked" pages that cannot be swapped out (see Chapter 17)
unsigned long	shared_vm	Number of pages in shared file memory mappings
unsigned long	exec_vm	Number of pages in executable memory mappings
unsigned long	stack_vm	Number of pages in the User Mode stack
unsigned long	reserved_vm	Number of pages in reserved or special memory regions
unsigned long	def_flags	Default access flags of the memory regions
unsigned long	nr_ptes	Number of Page Tables of this process
unsigned long []	saved_auxv	Used when starting the execution of an ELF program (see Chapter 20)
unsigned int	dumpable	Flag that specifies whether the process can produce a core dump of the memory
cpumask_t	cpu_vm_mask	Bit mask for lazy TLB switches (see Chapter 2)
mm_context_t	context	Pointer to table for architecture-specific information (e.g., LDT's address in 80 86 platforms)

memory region

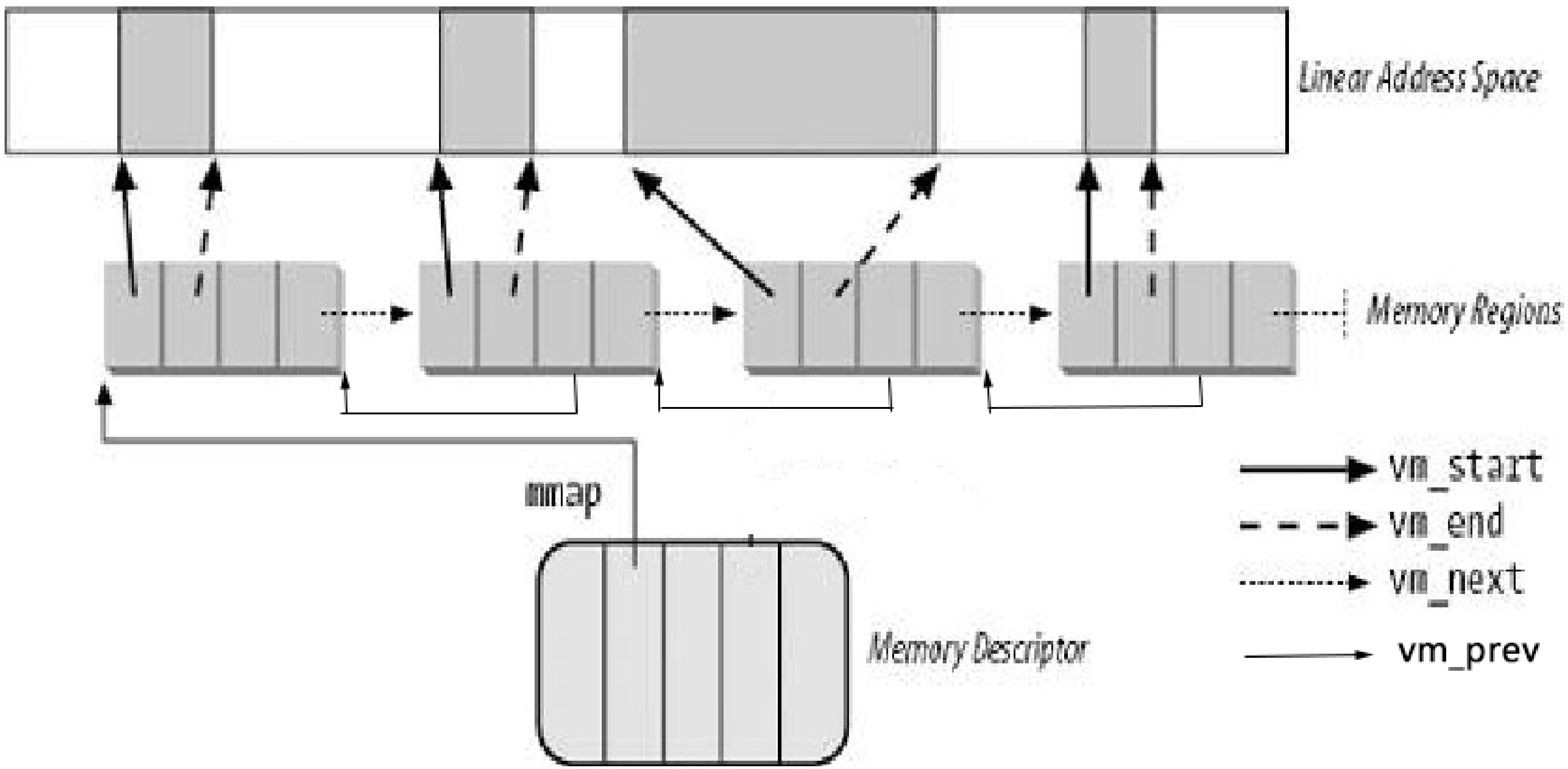
- Linux通过类型为vm_area_struct的对象实现一个memory region。
- 每个memory region描述符表示一个线性地址区间。
- vm_start字段指向线性区的第一个线性地址，而vm_end字　段指向线性区之后的第一个线性地址。因此vm_end - vm_start表示memory region的长度。
- vm_mm字段指向拥有这个区间的进程的mm_struct内存描述符。
- 进程所拥有的memory region从来不重叠，并且内核尽力把新分配的memory region与紧邻的现有memory region进行合并。两个相邻memory region的访问权限如果相匹配，就能把它们合并在一起。

vm_area_struct

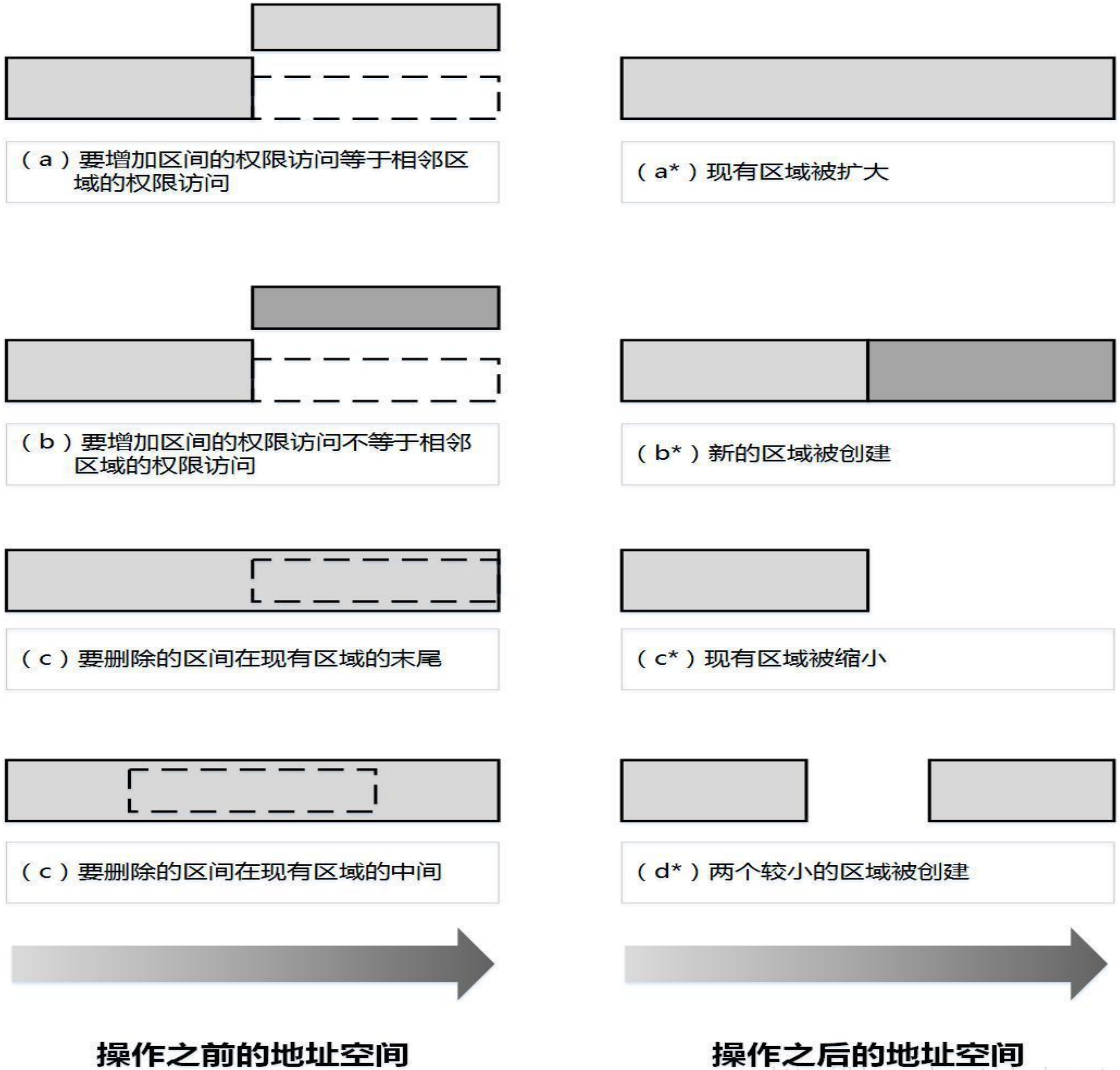
{linuxsrcdir}/include/linux/mm_types.h

Type	Field	Description
struct mm_struct *	vm_mm	Pointer to the memory descriptor that owns the region.
unsigned long	vm_start	First linear address inside the region.
unsigned long	vm_end	First linear address after the region.
struct vm_area_struct *	vm_next	Next region in the process list.
pgprot_t	vm_page_prot	Access permissions for the page frames of the region.
unsigned long	vm_flags	Flags of the region.
struct rb_node	vm_rb	Data for the red-black tree (see later in this chapter).
union	shared	Links to the data structures used for reverse mapping (see the section "Reverse Mapping for Mapped Pages" in Chapter 17).
struct list_head	anon_vma_node	Pointers for the list of anonymous memory regions (see the section "Reverse Mapping for Anonymous Pages" in Chapter 17).
struct anon_vma *	anon_vma	Pointer to the anon_vma data structure (see the section "Reverse Mapping for Anonymous Pages" in Chapter 17).
struct vm_operations_struct*	vm_ops	Pointer to the methods of the memory region.
unsigned long	vm_pgoff	Offset in mapped file (see Chapter 16). For anonymous pages, it is either zero or equal to vm_start/PAGE_SIZE (see Chapter 17).
struct file *	vm_file	Pointer to the file object of the mapped file, if any.
void *	vm_private_data	Pointer to private data of the memory region.

Descriptors related to the address space of a process



增加或删除一个线性区


（a）要增加区间的权限访问等于相邻区域的权限访问
（a*）现有区域被扩大
（b）要增加区间的权限访问不等于相邻区域的权限访问
（b*）新的区域被创建
（c）要删除的区间在现有区域的末尾
（c*）现有区域被缩小
（c）要删除的区间在现有区域的中间
（d*）两个较小的区域被创建
操作之前的地址空间
操作之后的地址空间

Memory region 数据结构

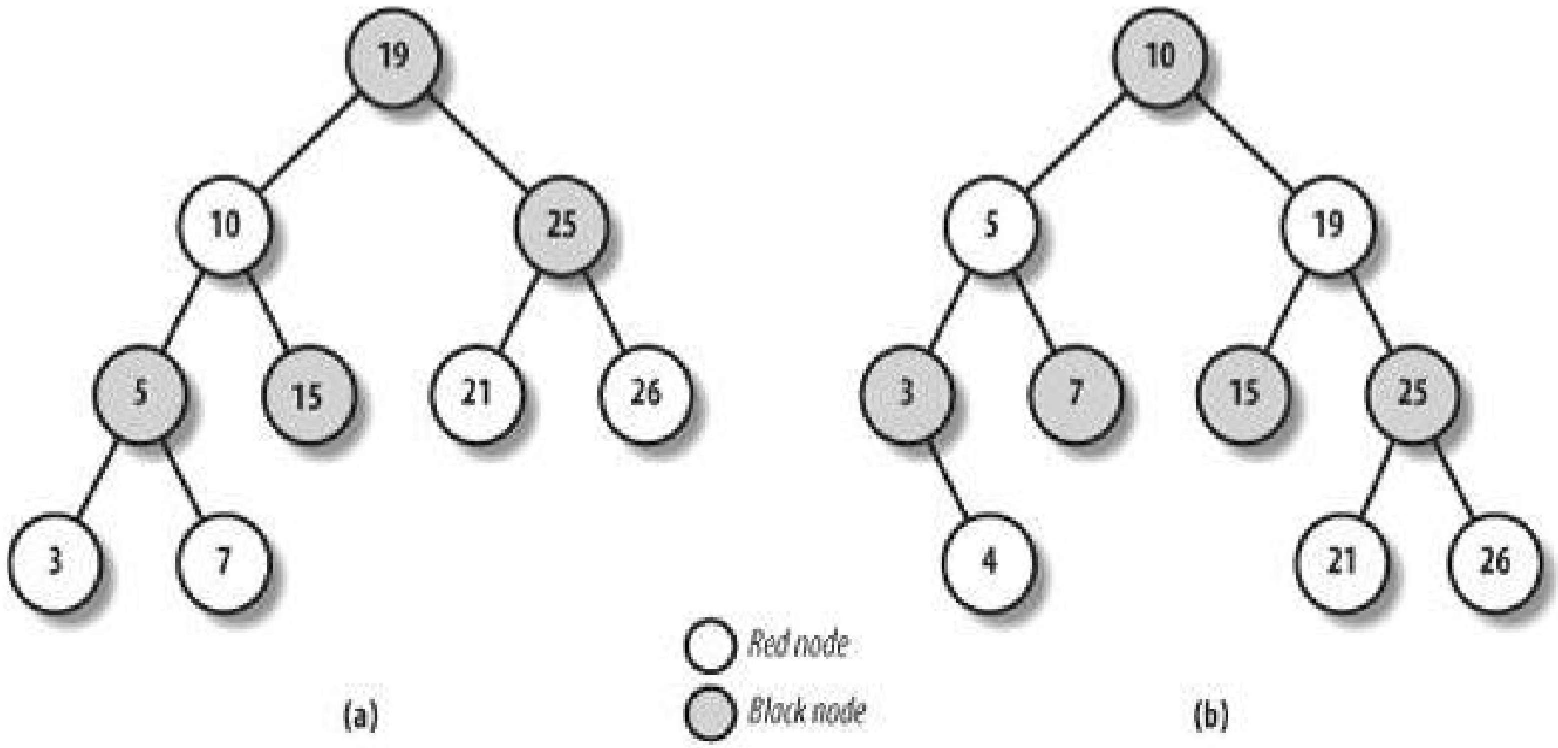
为了存放进程的线性区，Linux既使用了链表，也使用了红黑树。这两种数据结构包含指向同一线性区描述符的指针，当插入或删除一个线性区描述符时，内核通过红黑树搜索前后元素，并用搜索结果快速更新链表而不用扫描链表。

链表：链表的头由内存描述符的mmap字段所指向。任何线性区对象都在vm_next字段存放指向链表下一个元素的指针，在vm_prev字段存放指向链表上一个元素的指针。

红黑树：红黑树的首部由内存描述符中的mm_rb字段所指向。任何线性区对象都在类型为rb_node的vm_rb字段中存放颜色以及指向左孩子和右孩子的指针。

Red-black tree

大部分应用程序只用很少的 memory regions，但是有些大型应用软件需要几百到上千的 memory regions。在这种情况下，使用链表来管理所有的memory regions显然从效率的角度来说不够完美。为了进一步提高效率，把memory region的描述符使用红黑树来存放，大大提高了查找、插入、删除效率。*O(lg(n))*



Red-black tree 定义

- A nearly-balanced tree that uses an extra bit per node to maintain balance. No leaf is more than twice as far from the root as any other.
- A red-black tree with n internal nodes has height at most **$2\log_2(n+1)$.**
- **Also known as symmetric binary B-tree**.
- **Red-black tree** 是一种 **B** 树；**AVL** 树是**Red-black tree** 的特殊形式。(AVL 树的定义：A balanced binary search tree where the height of the two subtrees (children) of a node **differs by at most one**. Look-up, insertion, and deletion are O(log n), where n is the number of nodes in the tree.)

什么是 Red-black tree？

- Red-black tree 是二叉树。
- 一棵二叉树是 red-black tree 的条件：
  - 每个节点都有一个值；
  - 任何节点的值都大于其左孩子的值，小于其右孩子的值；
  - 根节点是黑色的；
  - 每个节点或者是红色的，或者是黑色的；
  - 每个红色节点只能拥有黑色节点的孩子；
  - 从某节点到其子孙叶节点必须包含同样数目的黑色节点。
- 插入删除操作是数据结构的内容，这里就不再叙述，有兴趣的同学请看：
  - **http://www.ececs.uc.edu/~franco/C321/html/RedBlack/redblack.html**

Struct rb_node

{linuxsrcdir}/include/linux/rbtree.h

```
struct rb_node {
    unsigned long  __rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));
   /* The alignment might seem pointless, but allegedly CRIS needs it */

struct rb_root {
    struct rb_node *rb_node;
};
```

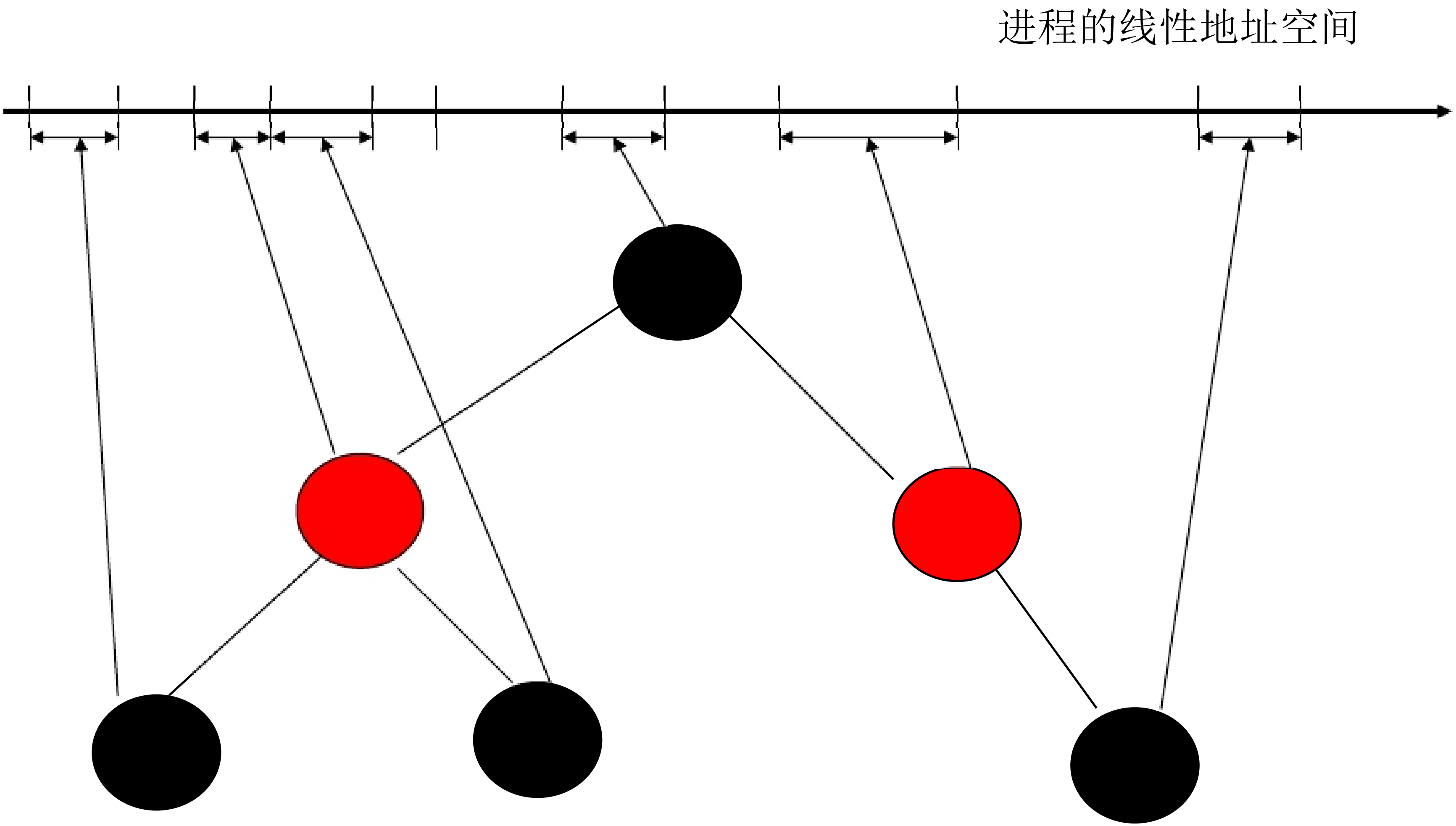
Page 和 Memory Region的关系

- 每个 Memory Region 指示的线性地址范围是几个有连续页号的页的集合，当然这些页可能在内存中存在，可能在内存中不存在。
- Memory region 有一些 flags 可以控制读写权限等性质，这些性质最终要体现到页表相关数据结构的一些性质；也影响缺页中断产生后造成的结果。

Memory Region Handling

- 涉及到的操作包括：
  - 根据线性地址寻找合适的 memory area；
  - 分配线性地址 interval；
  - 释放线性地址 interval；
  - 插入region；
  - 删除 region;
  - ……

红黑树组织的vma和线性地址空间



分配线性地址 interval get_unmapped_area( )

- **get_unmapped_area( )**
  - **搜索进程地址空间，发现一个长度为len的可用线性地址间隔(interval)，如果成功，返回线性地址；否则返回ENOMEM.**
  - **调用arch_get_unmapped_area或 arch_get_unmapped_area_topdown。**

arch_get_unmapped_area( ) 从低地址到高地址进行地址映射。

arch_get_unmapped_area_topdown( )从高地址到低地址映射。

具体调用哪个，要看进程的内存布局

The memory region layouts in the 80 x 86 architecture

Type of memory region	Classical layout	Flexible layout
Text segment (ELF)	Starts from 0x08048000	
Data and bss segments	Starts right after the text segment	
Heap	Starts right after the data and bss segments	
File memory mappings and anonymous memory regions	Starts from 0x40000000 (this address corresponds to 1/3 of the whole User Mode address space); libraries added at successively higher addresses	Starts near the end (lowest address) the User Mode stack; libraries added successively lower addresses
User Mode stack	Starts at 0xc0000000 and grows towards lower addresses	

The **flexible memory region layout** has been introduced in the kernel version 2.6.9
Why has the flexible layout been introduced? Its main advantage is that it allows a process to **make better use of the User Mode linear address space**. In the classical layout the heap is limited to less than 1 GB, while the other memory regions can fill up to about 2 GB (minus the stack size). In the flexible layout, these constraints are gone: both the heap and the other memory regions can freely expand until all the linear addresses left unused by the User Mode stack and the program's fixed-size segments are taken.

```
void arch_pick_mmap_layout(struct mm_struct *mm, struct rlimit *rlim_stack)
{
	if (mmap_is_legacy())
		mm->get_unmapped_area = arch_get_unmapped_area;
	else
		mm->get_unmapped_area = arch_get_unmapped_area_topdown;

	arch_pick_mmap_base(&mm->mmap_base, &mm->mmap_legacy_base,
			arch_rnd(mmap64_rnd_bits), task_size_64bit(0),
			rlim_stack);

#ifdef CONFIG_HAVE_ARCH_COMPAT_MMAP_BASES
	/*
	 * The mmap syscall mapping base decision depends solely on the
	 * syscall type (64-bit or compat). This applies for 64bit
	 * applications and 32bit applications. The 64bit syscall uses
	 * mmap_base, the compat syscall uses mmap_compat_base.
	 */
	arch_pick_mmap_base(&mm->mmap_compat_base, &mm-
>mmap_compat_legacy_base, arch_rnd(mmap32_rnd_bits), task_size_32bit(),
			rlim_stack);
#endif
}
```

arch/x86/mm/mmap.c

insert_vm_struct( )

在线性区对象链表和内存描述符的红黑树中插入一个vm_area_struct结构。这个函数使用两个参数：mm 指定进程内存描述符的地址，vma指定要插入的vm_area_struct对象的地址。其基本思路：

1. 利用find_vma_links()寻找出将要插入的结点位置，其前驱结点和其父结点。

2. If the memory region is **anonymous**, inserts the region in the list headed at the corresponding anon_vma data structure;

3. 利用vma_link通过__vma_link_list()和__vma_link_rb()将结点分别插入**链表**和**红黑树**中。

4. 线性区计数加一。

Anonymous memory region

- The default pager handles *nonpersistent memory*, known as **anonymous memory**. Anonymous memory is zero-initialized, and it exists **only during the life of a task**.
- A page is said to be anonymous if it belongs to an anonymous memory region of a process (*for instance, all pages in the User Mode heap or stack of a process are anonymous*).
- In order to reclaim the page frame, the kernel must save the page contents in a dedicated disk partition or disk file called "swap area" ; therefore, **all anonymous pages are swappable**.

进程的地址空间，和可执行程序的关系？

- 要想了解进程的地址空间， memory regions，以及可执行程序的关系，必须先了解可执行程序是怎么加载到内存中的，是如何开始运行的。

Process Image Layout and Linear Address Space

- Text. 又称代码段，这个段持有可执行指令，有可执行和可读属性。Linux 允许多个进程共享 text segment。mm_struct 的 start_code 和 end_code 域存放 text 段的开始和结束地址。
- Data. 这个 section 保存所有的初始化的数据，这些数据包括静态分配和全局分配的数据。

Process Image Layout and Linear Address Space

example1.c

```
int gvar = 10;
int main()
{
  ...
}
```

在 data section 中

Process Image Layout and Linear Address Space

- BSS. This section holds uninitialized data.

example2.c

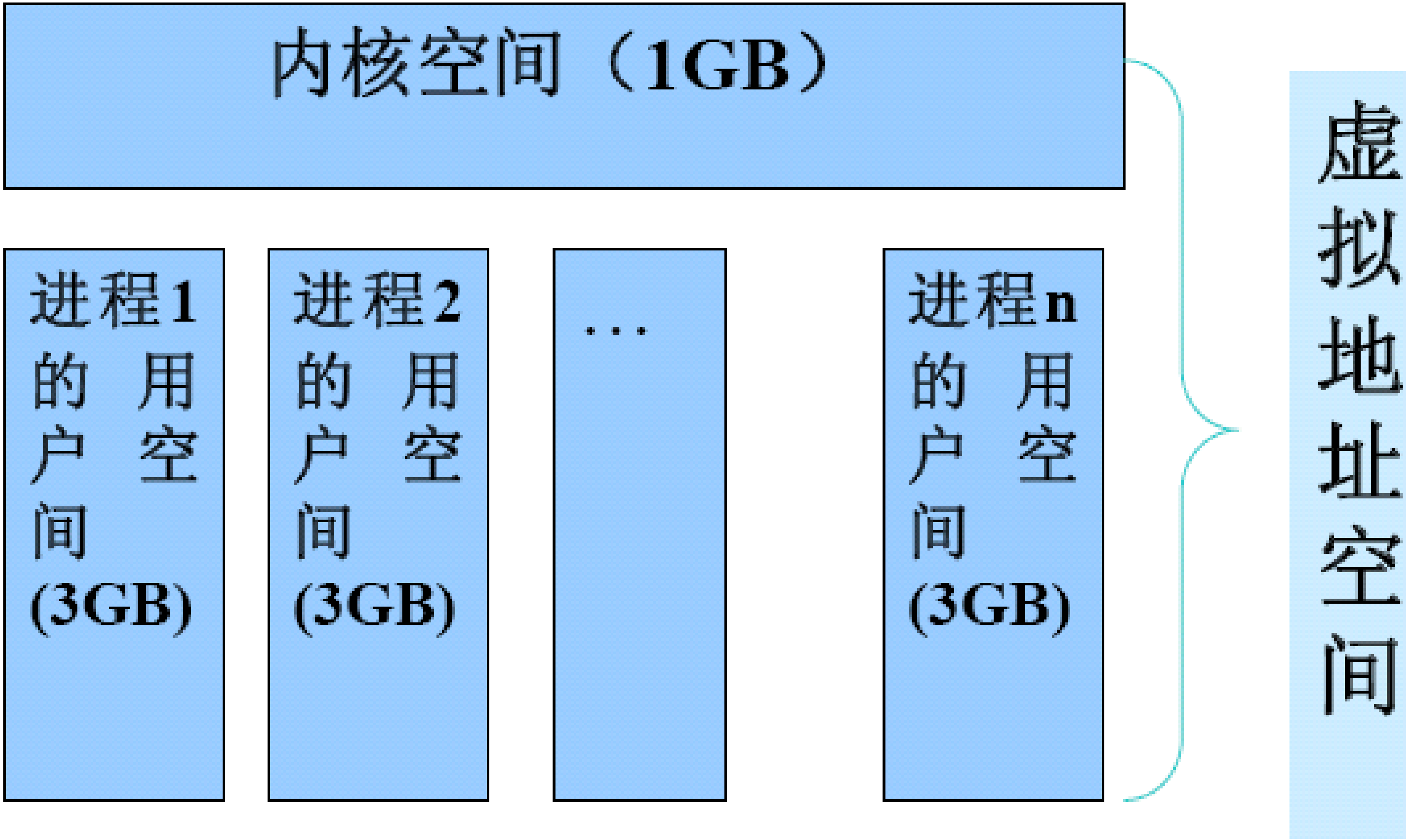
```
int gvar1[10];
long gvar2;
int main()
{
...
}
```

Process Image Layout and Linear Address Space

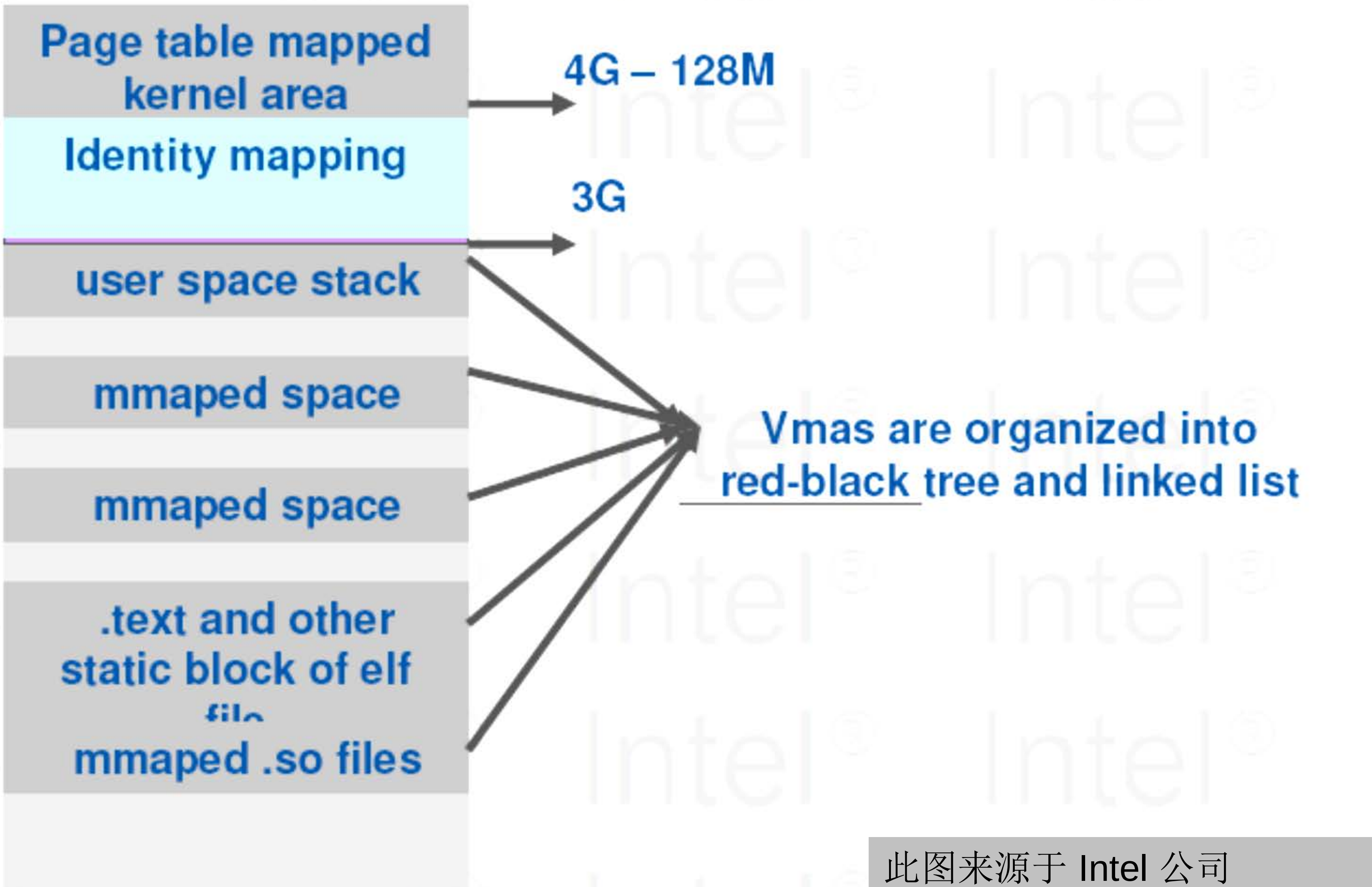
- Heap. 堆的分配是按照线性地址的增加来分配的，当应用程序使用 malloc 分配动态内存区域的时候，那么内存放到堆中。
  - mm_struct 的 start_brk 和 brk 域保存了堆的开始和结束地址。
  - 当调用 malloc() 的时候，调用系统调用 sys_brk()，增加 brk 指针，增加了堆

Process Image Layout and Linear Address Space

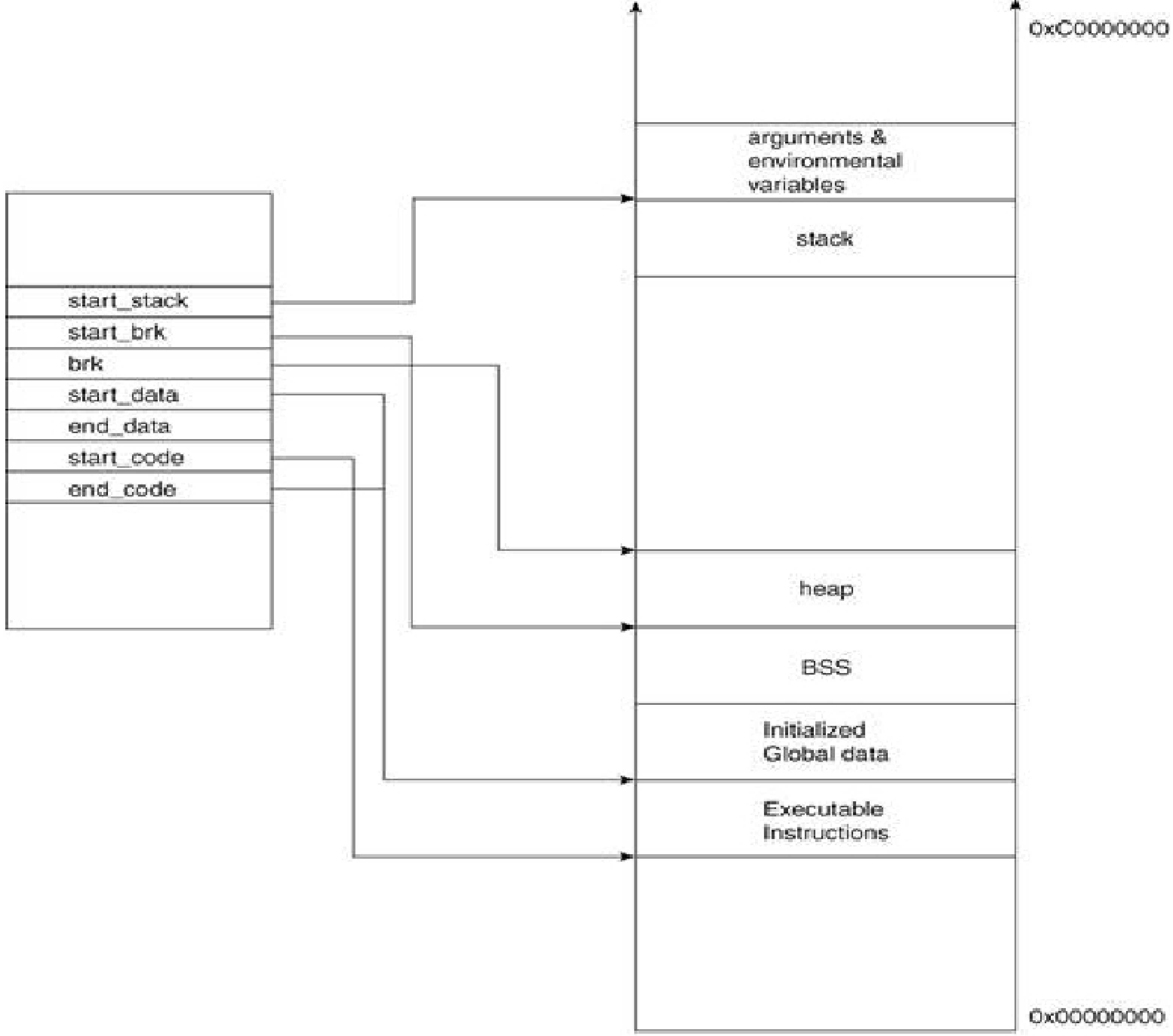
- Stack. 栈保存局 部 分 配的变量。当函数调用的时候，此函数的局部变量压到栈中。直到函数执行结束，和函数相关的局部变量从栈中弹出。当前其他的一些信息比如函数的参数、返回地址都保存在栈中。
  - mm_struct 的 start_stack 域标明了进程栈的开始地址。


内核空间（1GB）
进程1的用户空间(3GB)
进程2的用户空间(3GB)
…
进程n的用户空间(3GB)
虚拟地址空间

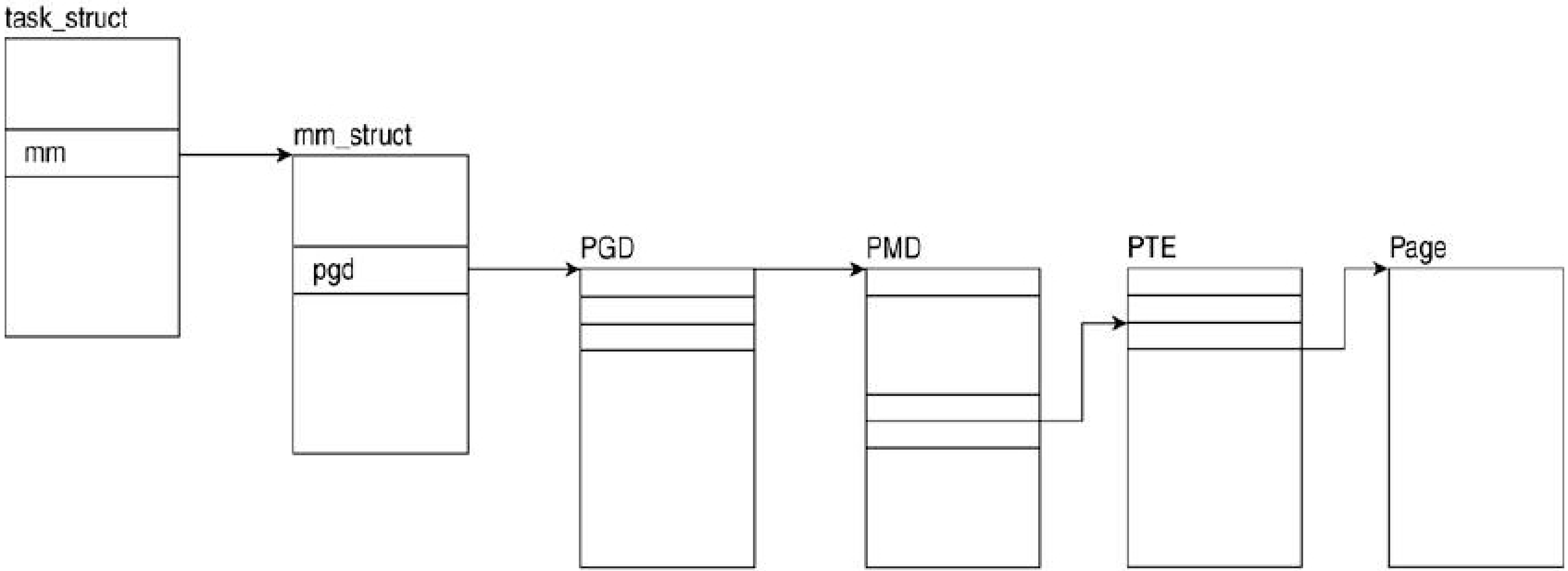
用户空间的虚拟内存



此图来源于 Intel 公司


0xC0000000
arguments &
environmental
variables
stack
start_stack
start_brk
brk
start_data
end_data
start_code
end_code
heap
BSS
Initialized
Global data
Executable
Instructions
0x00000000

Linux 进程页表



顶级页表是页全局目录(PGD)，二级页表是中间页目录(PMD).最后一级是页表(PTE),该页表结构指向物理页。上图中的页表对应的结构体定义在文件asm/page.h中。

Executable Format

- The standard Linux executable format is named **Executable and Linking Format (ELF)**.
- An executable format is described by an object of type linux_binfmt, which essentially provides three methods:
  - load_binary
    - **Sets up a new execution environment** for the current process by reading the information stored in an executable file.
  - load_shlib
    - **Dynamically binds a shared library** to an already running process; it is activated by the uselib( ) system call.
  - core_dump
    - **Stores the execution context** of the current process in a file named core

/include/linux/binfmts.h

```
struct linux_binfmt {
        struct list_head lh;
        struct module *module;
        int (*load_binary)(struct linux_binprm *);
        int (*load_shlib)(struct file *);
        int (*core_dump)(struct coredump_params *cprm);
        unsigned long min_coredump;        /* minimal dump size */
```

Executable Format

- All linux_binfmt objects are included in a singly linked list, and the address of the first element is stored in the **formats** variable.
- Elements can be inserted and removed in the list by invoking the register_binfmt( ) and unregister_binfmt( ) functions.
- The **register_binfmt**( ) function is executed
  - during system startup for each executable format compiled into the kernel
  - when a module implementing a new executable format is being loaded
- the **unregister_binfmt**( ) function is invoked when the module is unloaded

Executable Format

- Linux allows users to register their own **custom executable formats**.
- When the kernel determines that the executable file has a custom format, it starts the proper **interpreter program**.
- The interpreter program runs in User Mode, receives as its parameter the pathname of the executable file, and carries on the computation.

多种可执行程序格式

格式	linux_binfmt定义	load_binary	load_shlib	core_dump
a.out	aout_format	load_aout_binary	load_aout_library	aout_core_dump
flat style executables	flat_format	load_flat_binary	load_flat_shared_library	flat_core_dump
script脚本	script_format	load_script	无	无
misc_format	misc_format	load_misc_binary	无	无
em86	em86_format	load_format	无	无
elf_fdpic	elf_fdpic_format	load_elf_fdpic_binary	无	elf_fdpic_core_dump
elf	elf_format	load_elf_binary	load_elf_binary	elf_core_dump

Execution Domains

- 在非linux系统中编译的**POSIX**兼容的程序可以在linux上被很容易的执行，因为它们遵循同一套API（“应该”遵守，实际上有例外情况）——只有少数的不同：系统调用如何被调用；信号如何被编号等等
- 这些**不同的信息**被存储在类型为exec_domain 的执行域描述符中
- 进程：
  - 设置描述符中的personality域
  - 将相应的**exec_domain**的地址存储在thread_info的exec_domain域中

Personality	Operating system
PER_LINUX	Standard execution domain
PER_LINUX_32BIT	Linux with 32-bit physical addresses in 64-bit architectures
PER_LINUX_FDPIC	Linux program in ELF FDPIC format
PER_SVR4	System V Release 4
PER_SVR3	System V Release 3

linux_binprm

{linuxsrcdir}/include/linux/binfmts.h

/* This structure is used to **hold the arguments that are used when loading binaries**.用来保存要执行的文件相关的信息, 包括可执行程序的路径, 参数和环境变量的信息 */

```
struct linux_binprm {
#ifdef CONFIG_MMU
        struct vm_area_struct *vma;
        unsigned long vma_pages;
#else
# define MAX_ARG_PAGES          32
        struct page *page[MAX_ARG_PAGES];
#endif
        struct mm_struct *mm;
        unsigned long p; /* current top of mem */
        unsigned long argmin; /* rlimit marker for copy_strings() */
        ……
        char buf[BINPRM_BUF_SIZE];
} __randomize_layout;
```

execve的入口函数sys_execve

系统调用号(体系结构相关)   /arch/x86/entry/syscalls/syscall_64.tbl

59   execve   __x64_sys_execve/ptregs

入口函数声明   /include/linux/syscalls.h, line 842

```
asmlinkage long sys_execve(const char __user *filename,
                           const char __user *const __user *argv,
                           const char __user *const __user *envp);
```

系统调用实现   /fs/exec.c, line 1710

```
SYSCALL_DEFINE3(execve,
 const char __user *, filename,  //可执行程序的名称
 const char __user *const __user *, argv, //程序的参数
 const char __user *const __user *, envp) //环境变量
{
    return do_execve(getname(filename), argv, envp);
 }
```

execve加载可执行程序的过程

内核中实际执行**execv()**或**execve()**系统调用的程序是**do_execve()**：

- 打开目标映像文件
- 调用函数**search_binary_handler()** 搜索可执行文件类型队列，如果类型匹配，则调用**load_binary**函数指针所指向的处理函数来处理目标映像文件

sys_execve() > do_execve() > do_execveat_common > __do_execve_file > exec_binprm() > search_binary_handler() > load_binary() ELF文件格式处理函数是**load_elf_binary**

作业5

深入剖析（可选其一）

- kfree
- malloc
- mmap
- do_execve()