# Linux内存模型

## 一、前言

在linux内核中支持3中内存模型，分别是flat memory model，Discontiguous memory model和sparse memory model。所谓memory model，其实就是从cpu的角度看，其物理内存的分布情况，在linux kernel中，使用什么的方式来管理这些物理内存。另外，需要说明的是：本文主要focus在share memory的系统，也就是说所有的CPUs共享一片物理地址空间的。

本文的内容安排如下：为了能够清楚的解析内存模型，我们对一些基本的术语进行了描述，这在第二章。第三章则对三种内存模型的工作原理进行阐述，最后一章是代码解析，代码来自4.4.6内核，对于体系结构相关的代码，我们采用ARM64进行分析。

## 二、和内存模型相关的术语

1、什么是page frame？

操作系统最重要的作用之一就是管理计算机系统中的各种资源，做为最重要的资源：内存，我们必须管理起来。在linux操作系统中，物理内存是按照page size来管理的，具体page size是多少是和硬件以及linux系统配置相关的，4k是最经典的设定。因此，对于物理内存，我们将其分成一个个按page size排列的page，每一个物理内存中的page size的内存区域我们称之page frame。我们针对每一个物理的page frame建立一个struct page的数据结构来跟踪每一个物理页面的使用情况：是用于内核的正文段？还是用于进程的页表？是用于各种file cache还是处于free状态……

每一个page frame有一个一一对应的page数据结构，系统中定义了page\_to\_pfn和pfn\_to\_page的宏用来在page frame number和page数据结构之间进行转换，具体如何转换是和memory modle相关，我们会在第三章详细描述linux kernel中的3种内存模型。

2、什么是PFN？

对于一个计算机系统，其整个物理地址空间应该是从0开始，到实际系统能支持的最大物理空间为止的一段地址空间。在ARM系统中，假设物理地址是32个bit，那么其物理地址空间就是4G，在ARM64系统中，如果支持的物理地址bit数目是48个，那么其物理地址空间就是256T。当然，实际上这么大的物理地址空间并不是都用于内存，有些也属于I/O空间（当然，有些cpu arch有自己独立的io address space）。因此，内存所占据的物理地址空间应该是一个有限的区间，不可能覆盖整个物理地址空间。

PFN是page frame number的缩写，所谓page frame，就是针对物理内存而言的，把物理内存分成一个个的page size的区域，并且给每一个page 编号，这个号码就是PFN。假设物理内存从0地址开始，那么PFN等于0的那个页帧就是0地址（物理地址）开始的那个page。假设物理内存从x地址开始，那么第一个页帧号码就是（x>>PAGE\_SHIFT）。

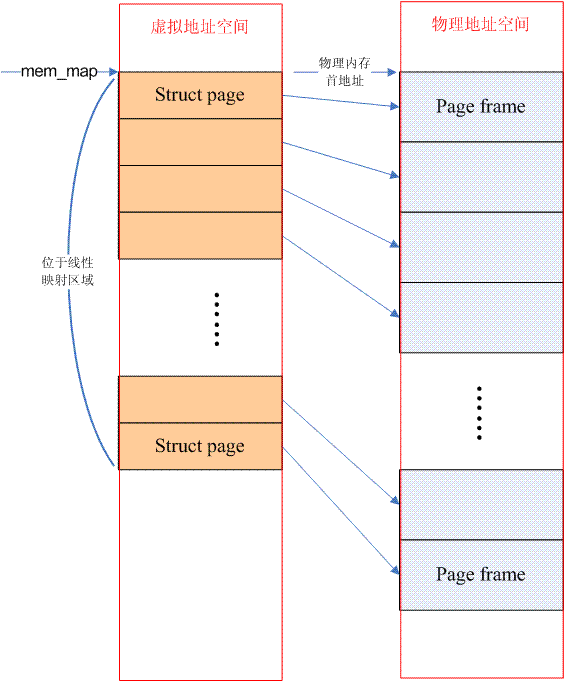
3、什么是NUMA？

在为multiprocessors系统设计内存架构的时候有两种选择：一种就是UMA（Uniform memory access），系统中的所有的processor共享一个统一的，一致的物理内存空间，无论从哪一个processor发起访问，对内存地址的访问时间都是一样的。NUMA（Non-uniform memory access）和UMA不同，对某个内存地址的访问是和该memory与processor之间的相对位置有关的。例如，对于某个节点（node）上的processor而言，访问local memory要比访问那些remote memory的速度要快。

## 三、Linux 内核中的三种memory model

1、什么是FLAT memory model？

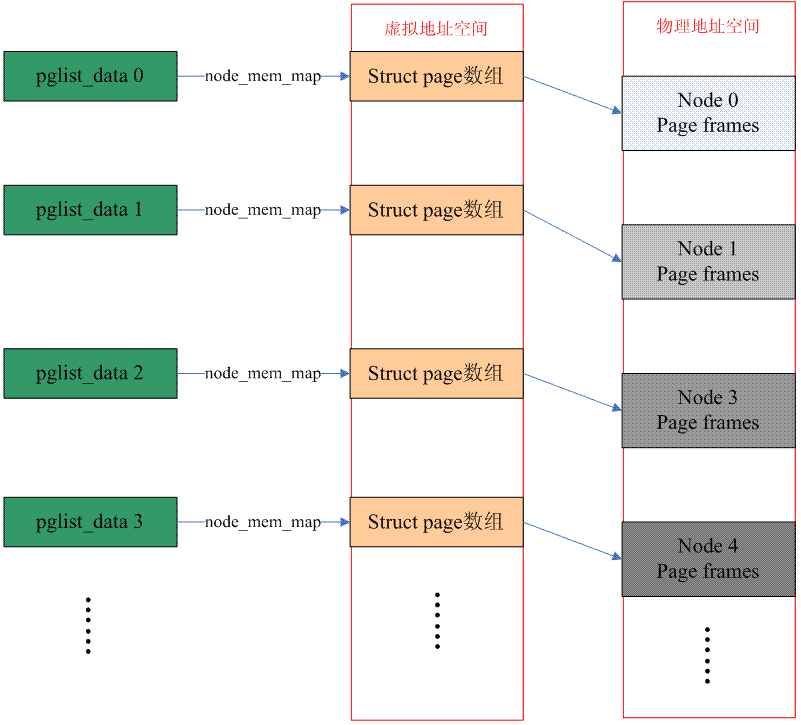
如果从系统中任意一个processor的角度来看，当它访问物理内存的时候，物理地址空间是一个连续的，没有空洞的地址空间，那么这种计算机系统的**内存模型就是Flat memory**。这种内存模型下，物理内存的管理比较简单，每一个物理页帧都会有一个page数据结构来抽象，因此系统中存在一个struct page的数组（mem\_map），每一个数组条目指向一个实际的物理页帧（page frame）。在flat memory的情况下，PFN（page frame number）和mem\_map数组index的关系是线性的（有一个固定偏移，如果内存对应的物理地址等于0，那么PFN就是数组index）。因此从PFN到对应的page数据结构是非常容易的，反之亦然，具体可以参考page\_to\_pfn和pfn\_to\_page的定义。此外，对于flat memory model，节点（struct pglist\_data）只有一个（为了和Discontiguous Memory Model采用同样的机制）。下面的图片描述了flat memory的情况：



需要强调的是struct page所占用的内存位于直接映射（directly mapped）区间，因此操作系统不需要再为其建立page table。

**2、什么是Discontiguous Memory Model？**

如果cpu在访问物理内存的时候，其地址空间有一些空洞，是不连续的，那么这种计算机系统的内存模型就是Discontiguous memory。一般而言，NUMA架构的计算机系统的memory model都是选择Discontiguous Memory，不过，这两个概念其实是不同的。NUMA强调的是memory和processor的位置关系，和内存模型其实是没有关系的，只不过，由于同一node上的memory和processor有更紧密的耦合关系（访问更快），因此需要多个node来管理。Discontiguous memory本质上是flat memory内存模型的扩展，整个物理内存的address space大部分是成片的大块内存，中间会有一些空洞，每一个成片的memory address space属于一个node（如果局限在一个node内部，其内存模型是flat memory）。下面的图片描述了Discontiguous memory的情况：



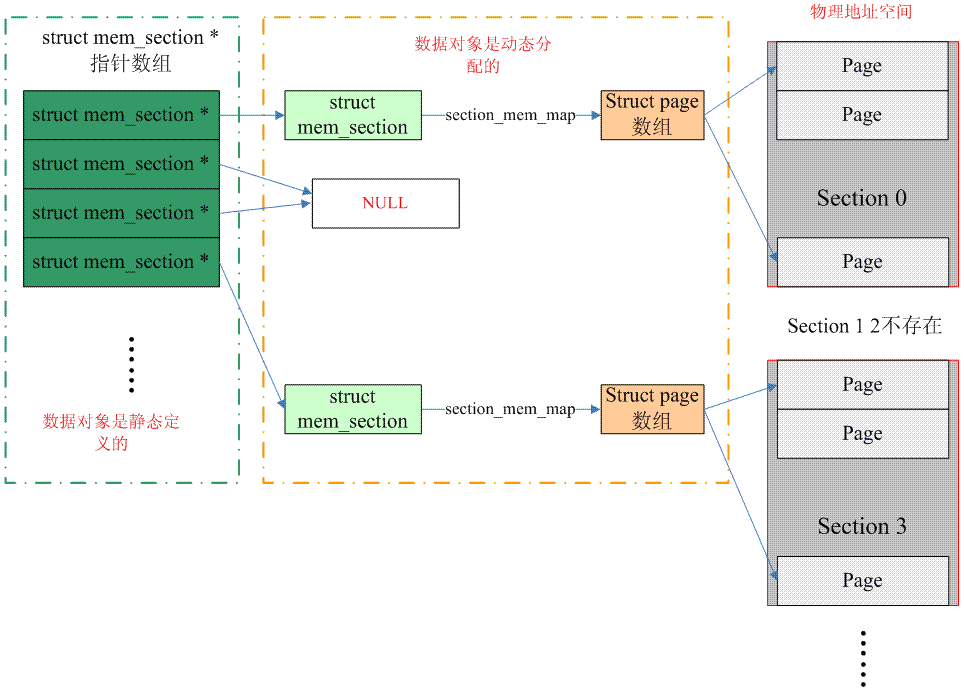
因此，这种内存模型下，节点数据（struct pglist\_data）有多个，宏定义NODE\_DATA可以得到指定节点的struct pglist\_data。而，每个节点管理的物理内存保存在struct pglist\_data 数据结构的node\_mem\_map成员中（概念类似flat memory中的mem\_map）。这时候，从PFN转换到具体的struct page会稍微复杂一点，我们首先要从PFN得到node ID，然后根据这个ID找到对于的pglist\_data 数据结构，也就找到了对应的page数组，之后的方法就类似flat memory了。

**3、什么是Sparse Memory Model？**

Memory model也是一个演进过程，刚开始的时候，使用flat memory去抽象一个连续的内存地址空间（mem\_maps[]），出现NUMA之后，整个不连续的内存空间被分成若干个node，每个node上是连续的内存地址空间，也就是说，原来的单一的一个mem\_maps[]变成了若干个mem\_maps[]了。一切看起来已经完美了，但是memory hotplug的出现让原来完美的设计变得不完美了，因为即便是一个node中的mem\_maps[]也有可能是不连续了。其实，在出现了sparse memory之后，Discontiguous memory内存模型已经不是那么重要了，按理说sparse memory最终可以替代Discontiguous memory的，这个替代过程正在进行中，4.4的内核仍然是有3中内存模型可以选择。

为什么说sparse memory最终可以替代Discontiguous memory呢？实际上在sparse memory内存模型下，连续的地址空间按照SECTION（例如1G）被分成了一段一段的，其中每一section都是hotplug的，因此sparse memory下，内存地址空间可以被切分的更细，支持更离散的Discontiguous memory。此外，在sparse memory没有出现之前，NUMA和Discontiguous memory总是剪不断，理还乱的关系：NUMA并没有规定其内存的连续性，而Discontiguous memory系统也并非一定是NUMA系统，但是这两种配置都是multi node的。有了sparse memory之后，我们终于可以把内存的连续性和NUMA的概念剥离开来：一个NUMA系统可以是flat memory，也可以是sparse memory，而一个sparse memory系统可以是NUMA，也可以是UMA的。

下面的图片说明了sparse memory是如何管理page frame的（配置了SPARSEMEM\_EXTREME）：

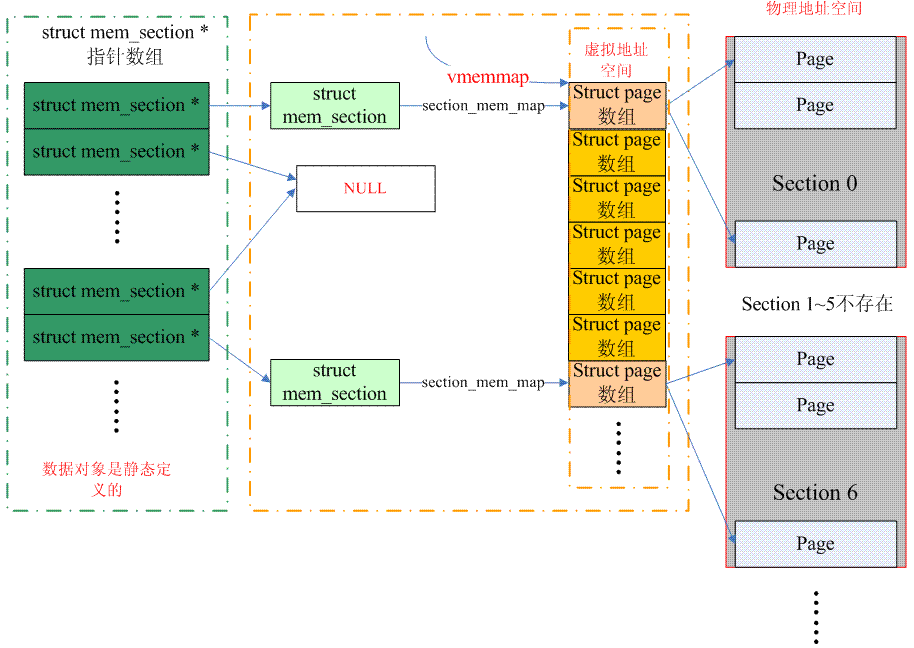


（注意：上图中的一个mem\_section指针应该指向一个page，而一个page中有若干个struct mem\_section数据单元）

整个连续的物理地址空间是按照一个section一个section来切断的，每一个section内部，其memory是连续的（即符合flat memory的特点），因此，mem\_map的page数组依附于section结构（struct mem\_section）而不是node结构了（struct pglist\_data）。当然，无论哪一种memory model，都需要处理PFN和page之间的对应关系，只不过sparse memory多了一个section的概念，让转换变成了PFN<--->Section<--->page。

我们首先看看如何从PFN到page结构的转换：kernel中静态定义了一个mem\_section的指针数组，一个section中往往包括多个page，因此需要通过右移将PFN转换成section number，用section number做为index在mem\_section指针数组可以找到该PFN对应的section数据结构。找到section之后，沿着其section\_mem\_map就可以找到对应的page数据结构。顺便一提的是，在开始的时候，sparse memory使用了一维的memory\_section数组（不是指针数组），这样的实现对于特别稀疏（CONFIG\_SPARSEMEM\_EXTREME）的系统非常浪费内存。此外，保存指针对hotplug的支持是比较方便的，指针等于NULL就意味着该section不存在。上面的图片描述的是一维mem\_section指针数组的情况（配置了SPARSEMEM\_EXTREME），对于非SPARSEMEM\_EXTREME配置，概念是类似的，具体操作大家可以自行阅读代码。

从page到PFN稍微有一点麻烦，实际上PFN分成两个部分：一部分是section index，另外一个部分是page在该section的偏移。我们需要首先从page得到section index，也就得到对应的memory\_section，知道了memory\_section也就知道该page在section\_mem\_map，也就知道了page在该section的偏移，最后可以合成PFN。对于page到section index的转换，sparse memory有2种方案，我们先看看经典的方案，也就是保存在page->flags中（配置了SECTION\_IN\_PAGE\_FLAGS）。这种方法的最大的问题是page->flags中的bit数目不一定够用，因为这个flag中承载了太多的信息，各种page flag，node id，zone id现在又增加一个section id，在不同的architecture中无法实现一致性的算法，有没有一种通用的算法呢？这就是CONFIG\_SPARSEMEM\_VMEMMAP。具体的算法可以参考下图：



（上面的图片有一点问题，vmemmap只有在PHYS\_OFFSET等于0的情况下才指向第一个struct page数组，一般而言，应该有一个offset的，不过，懒得改了，哈哈）

对于经典的sparse memory模型，一个section的struct page数组所占用的内存来自directly mapped区域，页表在初始化的时候就建立好了，分配了page frame也就是分配了虚拟地址。但是，对于SPARSEMEM\_VMEMMAP而言，虚拟地址一开始就分配好了，是vmemmap开始的一段连续的虚拟地址空间，每一个page都有一个对应的struct page，当然，只有虚拟地址，没有物理地址。因此，当一个section被发现后，可以立刻找到对应的struct page的虚拟地址，当然，还需要分配一个物理的page frame，然后建立页表什么的，因此，对于这种sparse memory，开销会稍微大一些（多了个建立映射的过程）。

## 四、代码分析

我们的代码分析主要是通过include/asm-generic/memory\_model.h展开的。

1、flat memory。代码如下：

#define \_\_pfn\_to\_page(pfn)    (mem\_map + ((pfn) - ARCH\_PFN\_OFFSET))   
#define \_\_page\_to\_pfn(page)    ((unsigned long)((page) - mem\_map) + ARCH\_PFN\_OFFSET)

由代码可知，PFN和struct page数组（mem\_map）index是线性关系，有一个固定的偏移就是ARCH\_PFN\_OFFSET，这个偏移是和估计的architecture有关。对于ARM64，定义在arch/arm/include/asm/memory.h文件中，当然，这个定义是和内存所占据的物理地址空间有关（即和PHYS\_OFFSET的定义有关）。

2、Discontiguous Memory Model。代码如下：

#define \_\_pfn\_to\_page(pfn)            \   
({    unsigned long \_\_pfn = (pfn);        \   
    unsigned long \_\_nid = arch\_pfn\_to\_nid(\_\_pfn);  \   
    NODE\_DATA(\_\_nid)->node\_mem\_map + arch\_local\_page\_offset(\_\_pfn, \_\_nid);\   
})

#define \_\_page\_to\_pfn(pg)                        \   
({    const struct page \*\_\_pg = (pg);                    \   
    struct pglist\_data \*\_\_pgdat = NODE\_DATA(page\_to\_nid(\_\_pg));    \   
    (unsigned long)(\_\_pg - \_\_pgdat->node\_mem\_map) +            \   
     \_\_pgdat->node\_start\_pfn;                    \   
})

Discontiguous Memory Model需要获取node id，只要找到node id，一切都好办了，比对flat memory model进行就OK了。因此对于\_\_pfn\_to\_page的定义，可以首先通过arch\_pfn\_to\_nid将PFN转换成node id，通过NODE\_DATA宏定义可以找到该node对应的pglist\_data数据结构，该数据结构的node\_start\_pfn记录了该node的第一个page frame number，因此，也就可以得到其对应struct page在node\_mem\_map的偏移。\_\_page\_to\_pfn类似，大家可以自己分析。

3、Sparse Memory Model。经典算法的代码我们就不看了，一起看看配置了SPARSEMEM\_VMEMMAP的代码，如下：

#define \_\_pfn\_to\_page(pfn)    (vmemmap + (pfn))   
#define \_\_page\_to\_pfn(page)    (unsigned long)((page) - vmemmap)

简单而清晰，PFN就是vmemmap这个struct page数组的index啊。对于ARM64而言，vmemmap定义如下：

#define vmemmap            ((struct page \*)VMEMMAP\_START - \   
                 SECTION\_ALIGN\_DOWN(memstart\_addr >> PAGE\_SHIFT))

毫无疑问，我们需要在虚拟地址空间中分配一段地址来安放struct page数组（该数组包含了所有物理内存跨度空间page），也就是VMEMMAP\_START的定义。

# Fix-Mapped Addresses

## 一、前言

某天，wowo同学突然来了一句：如果要在start\_kernel中点LED，ioremap在什么时间点才能调用呢？我想他应该是想通过点LED灯来调试start\_kernel之后的初始化的代码（例如DTB解析部分的代码）。那天，我们两个花了二十分钟的时间，讨论相关的问题，我觉得很有意思，因此决定写fix mapped address这样的一份文档。

在汇编代码中，由于没有打开MMU，想怎么访问外设都很简单，直接使用物理地址即可，然而，进入start kernel之后（打开了MMU），想要访问硬件都是那么的不方便，至少需要通过ioremap获取了虚拟地址之后才可以访问。但是，实际上，在内核的启动的初始阶段，内存管理子系统还没有ready，ioremap还不能调用（在mm\_init之后可以正常使用）。

实际上，这个需求是和early ioremap模块相关，此外，还有一些其他的需求，内核合并了这些需求并提出了fix mapped address的概念。本文就是描述关于fix mapped address的方方面面，BTW，本文的代码来自4.4.6内核，体系结构相关的代码依然选择的是ARM64。

## 二、什么是fixmap？

Fix map中的fix指的是固定的意思，那么固定什么东西呢？其实就是虚拟地址是固定的，也就是说，有些虚拟地址在编译（compile-time）的时候就固定下来了，而这些虚拟地址对应的物理地址不是固定的，是在kernel启动过程中被确定的。为了方便大家理解fixmap，我们提供一个具体的例子：DTB image的处理。

实际上DTB的处理一开始并没有使用fixmap，而是放在了kernel image附近。更具体的要求是必须放在kernel image起始地址的512M内，8字节对齐，dtb image不能越过2M section size的边界。之所以这么要求，主要是想借用kernel image页表的“东风”，反正建立kernel image所需要的PGD/PUD/PMD页表都已经静态分配了，对dtb image的要求可以不需要建立新的页表，只是在原有的页表中增加一个entry而已，不过这样的设计存在下面的问题：

（1）对dtb image的位置有限制，不是很灵活（谁不想自由自在呢？bootload copy dtb img的时候当然想了无牵挂啊）。

（2）对dtb image的位置检查在head.S中，需要汇编实现（谁想写汇编啊，用c实现多买易懂啊，可维护性，可移植性多么好啊）。

现在，既然内核有了early fixmap的支持，那么上面的限制都可以放宽了。整个dtb image的处理过程如下：

（1）bootloader copy dtb image到memory的某个位置上。具体的位置随便，当然还是要满足8字节对齐，dtb image不能越过2M section size的边界的要求，毕竟我们也想一条section mapping就搞定dtb image。

（2）bootload通过寄存器x0传递dtb image的物理地址，dtb image的虚拟地址在编译kernel image的时候就确定了。

（3）汇编初始化阶段不对dtb image做任何处理

（4）在start kernel之后的初始化代码中（具体在setup\_arch--->setup\_machine\_fdt中），创建dtb image的相关Translation tables，之后就可以自由的访问dtb image了。

## 三、为何有fixmap这个概念？

动态分配虚拟地址以及建立地址映射是一个复杂的过程，在内核完全启动之后，内存管理可以提供各种丰富的API让内核的其他模块可以完成虚拟地址分配和建立地址映射的功能，但是，在内核的启动过程中，有些模块需要使用虚拟内存并mapping到指定的物理地址上，而且，这些模块也没有办法等待完整的内存管理模块初始化之后再进行地址映射。因此，linux kernel固定分配了一些fixmap的虚拟地址，这些地址有固定的用途，使用该地址的模块在初始化的时候，讲这些固定分配的地址mapping到指定的物理地址上去。

最直观的需求来自初始化代码的调试，想一想当我们来到start\_kernel的时候我们面临的处境：

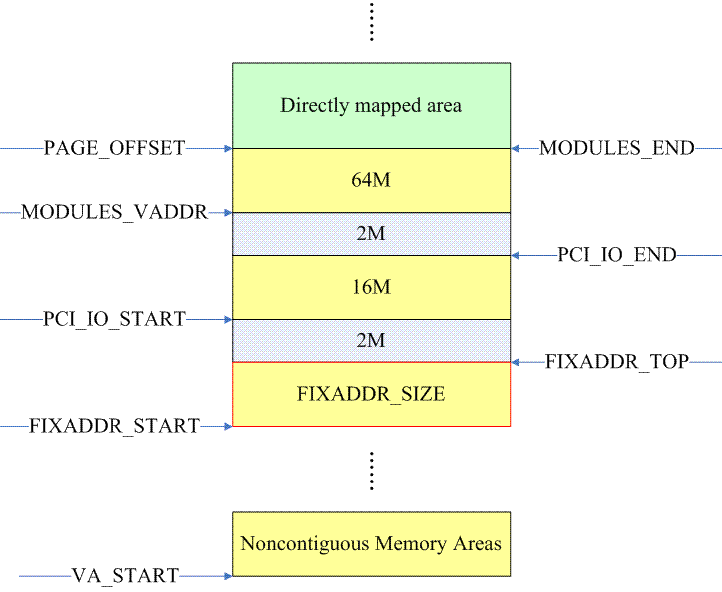
（1）我们不能访问全部的内存，只能访问kernel image附近的memory。

（2）我们不能访问任何的硬件，所有的io memory还没有mapping

想要通过串口控制台输出错误信息？sorry，现在离console驱动的初始化还早着呢。想点个LED灯看看内核运行情况？sorry，ioremp函数需要kmalloc分配内存，但是伙伴系统还没有初始化呢。怎么办？一个最简洁的方法就是简化虚拟内存的分配和管理（ioremp中使用的管理虚拟内存地址的方法太重了），而最简单的方法就是fix virtual address。

四、fixmap的具体位置在那里？

fixmap的地址区域位于FIXADDR\_START和FIXADDR\_TOP之间，具体可以参考下图：



上图中，红色框的block就是fixmap address的具体位置。

五、fixmap具体应用在哪些场景？

fixmap的地址区域有被进一步细分，如下：

enum fixed\_addresses {   
    FIX\_HOLE,    
    FIX\_FDT\_END,   
    FIX\_FDT = FIX\_FDT\_END + FIX\_FDT\_SIZE / PAGE\_SIZE - 1,

    FIX\_EARLYCON\_MEM\_BASE,   
    FIX\_TEXT\_POKE0,   
    \_\_end\_of\_permanent\_fixed\_addresses,

    FIX\_BTMAP\_END = \_\_end\_of\_permanent\_fixed\_addresses,   
    FIX\_BTMAP\_BEGIN = FIX\_BTMAP\_END + TOTAL\_FIX\_BTMAPS - 1,   
    \_\_end\_of\_fixed\_addresses   
};

由定义可知，fixmap地址区域又分成了两个部分，一部分叫做permanent fixed address，是用于具体的某个内核模块的，使用关系是永久性的。另外一个叫做temporary fixed address，各个内核模块都可以使用，用完之后就释放，模块和虚拟地址之间是动态的关系。

permanent fixed address主要涉及的模块包括：

（1）dtb解析模块。

（2）early console模块。标准的串口控制台驱动的初始化在整个kernel初始化过程中是很靠后的事情了，如果能够在kernel启动阶段的初期就有一个console，能够输出各种debug信息是多买美妙的事情啊，early console就能满足你的这个愿望，这个模块是使用early param来初始化该模块的功能的，因此可以很早就投入使用，从而协助工程师了解内核的启动过程。

（3）动态打补丁的模块。正文段一般都被映射成read only的，该模块可以使用fix mapped address来映射RW的正文段，从动态修改程序正文段，从而完成动态打补丁的功能。

temporary fixed address主要用于early ioremap模块。linux kernel在fix map区域的虚拟地址空间中开了FIX\_BTMAPS\_SLOTS个的slot（每个slot的size是NR\_FIX\_BTMAPS），内核中的模块都能够通过early\_ioremap、early\_iounmap的接口来申请或者释放对某个slot 虚拟地址的使用。

# TLB flush操作

## 一、前言

Linux VM subsystem在很多场合都需要对TLB进行flush操作，本文希望能够把这个知识点相关的方方面面描述清楚。第二章描述了一些TLB的基本概念，第三章描述了ARM64中TLB的具体硬件实现，第四章描述了linux中和TLB flush相关的软件接口。内核版本依然是4.4.6版本。

## 二、基本概念

1、什么是TLB？

TLB的全称是Translation Lookaside Buffer，我们知道，处理器在取指或者执行访问memory指令的时候都需要进行地址翻译，即把虚拟地址翻译成物理地址。而地址翻译是一个漫长的过程，需要遍历几个level的Translation table，从而产生严重的开销。为了提高性能，我们会在MMU中增加一个TLB的单元，把地址翻译关系保存在这个高速缓存中，从而省略了对内存中页表的访问。

2、为什么有TLB？

TLB这个术语有些迷惑，但是其本质上就是一种cache，既然是一种cache，那么就没有什么好说的，当然其存在就是为了更高的performance了。不同于instruction cache和data cache，它是Translation cache。对于instruction cache，它是解决cpu获取main memory中的指令数据（地址保存在PC寄存器中）的速度比较慢的问题而设立的。同样的data cache是为了解决数据访问指令比较慢而设立的。但是实际上这只是事情的一部分，我们来仔细看看程序中的数据访问指令（例如说是把）的执行过程，这个过程可以分成如下几个步骤：

（1）将PC中的虚拟地址翻译成物理地址

（2）从memory中获取数据访问指令（假设该指令需要访问地址x）

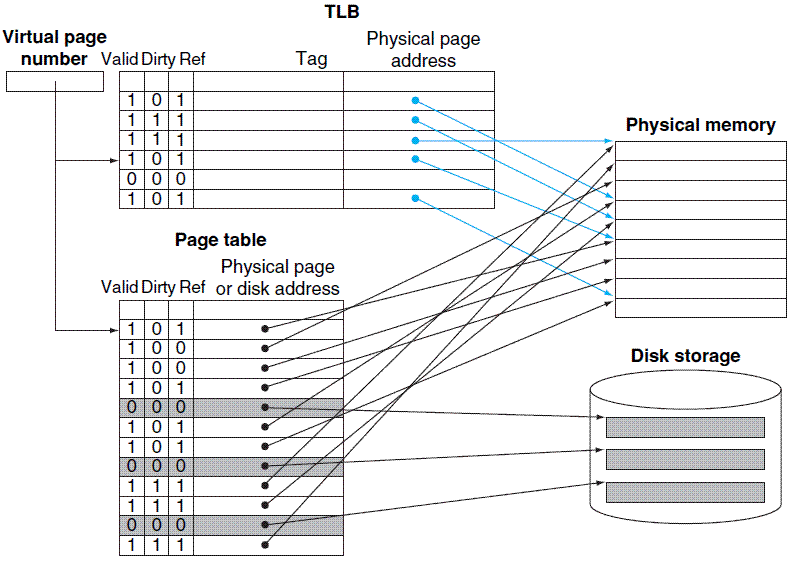
（3）将虚拟地址x翻译成物理地址y

（4）从location y的memory中获取具体的数据

instruction cache解决了step （2）的性能问题，data cache解决了step （4）中的性能问题，当然，复杂的系统会设立了各个level的cache用来缓存main memory中的数据，因此，实际上unified cache同时可以加快step （2）和（4）的速度。Anyway，这只是解决了部分的问题，IC设计工程师怎么会忽略step （1）和step （3）呢，这也就是TLB的由来。如果CPU core发起的地址翻译过程能够在TLB（translation cache）中命中（cache hit），那么CPU不需要访问慢速的main memory从而加快了CPU的performance。

3、TLB工作原理

大概的原理图如下（图片来自Computer Organization and Design 5th）：



当需要转换VA到PA的时候，首先在TLB中找是否有匹配的条目，如果有，那么我们称之TLB hit，这时候不需要再去访问页表来完成地址翻译。不过TLB始终是全部页表的一个子集，因此也有可能在TLB中找不到。如果没有在TLB中找到对应的item，那么称之TLB miss，那么就需要去访问memory中的page table来完成地址翻译，同时将翻译结果放入TLB，如果TLB已经满了，那么还要设计替换算法来决定让哪一个TLB entry失效，从而加载新的页表项。简单的描述就是这样了，我们可以对TLB entry中的内容进行详细描述，它主要包括：

（1） 物理地址（更准确的说是physical page number）。这是地址翻译的结果。

（2） 虚拟地址（更准确的说是virtual page number）。用cache的术语来描述的话应该叫做Tag，进行匹配的时候就是对比Tag。

（3） Memory attribute（例如：memory type，cache policies，access permissions）

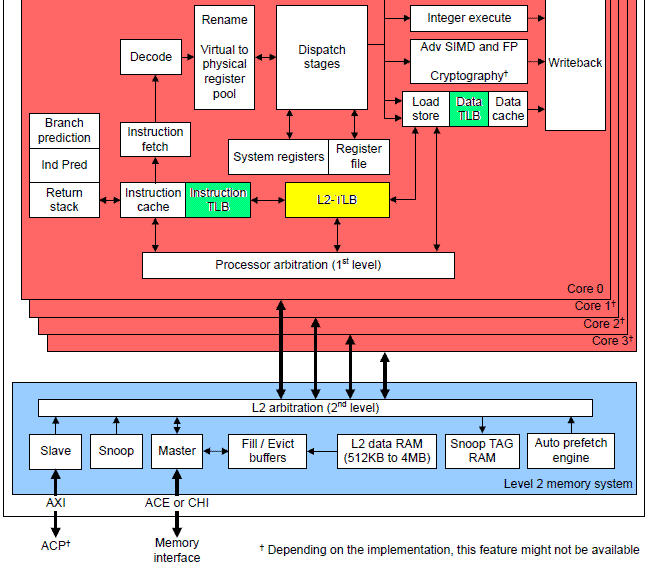
（4） status bits（例如：Valid、dirty和reference bits）

（5） 其他相关信息。例如ASID、VMID，下面的章节会进一步描述。

## 三、ARMv8的TLB

我们选择Cortex-A72 processor来描述ARMv8的TLB的组成结构以及维护TLB的指令。

1、TLB的组成结构。下图是A72的功能block：



A72实现了2个level的TLB，绿色是L1 TLB，包括L1 instruction TLB（48-entry fully-associative）和L1 data TLB（32-entry fully-associative）。黄色block是L2 unified TLB，它要大一些，可以容纳1024个entry，是4-way set-associative的。当L1 TLB发生TLB miss的时候，L2 TLB是它们坚强的后盾。

通过上图，我们还可以看出：对于多核CPU，每个processor core都有自己的TLB。

2、如何确定TLB match

整个地址翻译过程并非简单的VA到PA的映射那么简单，其实系统中的虚拟地址空间有很多，而每个地址空间的翻译都是独立的：

（1）操作系统中的每一个进程都有自己独立的虚拟地址空间。在各个进程不同的虚拟地址空间中，相同的VA被翻译成不同的PA。

（2）如果支持虚拟化，系统中存在一个host OS和多个guest OS，不同OS之间，地址翻译是不同的，而对于一个guest OS内部，其地址空间的情况请参考（1）。

（3）如果支持TrustZone，secure monitor、secure world以及normal world是不同的虚拟地址空间。

当然，我们可以在TLB匹配过程中，不考虑上面的复杂情况，比如在进程切换的时候，在切换虚拟机的时候，或者在切换secure/normal world的时候，将TLB中的所有内容全部flush掉（全部置为无效），这样的设计当然很清爽，但是性能会大打折扣。因此，实际上在设计TLB的时候，往往让TLB entry包括了和虚拟地址空间context相关的信息。在A72中，只有满足了下面的条件，才能说匹配了一个TLB entry：

（1）请求进行地址翻译的VA page number等于TLB entry中的VA page number

（2）请求进行地址翻译的memory space identifier等于TLB entry中的memory space identifier。所谓memory space identifier其实就是区分请求是来自EL3 Exception level、Nonsecure EL2 Exception level或者是Secure and Non-secure EL0还是EL1 Exception levels。

（3）如果该entry被标记为non-Global，那么请求进行地址翻译的ASID（保存在TTBRx中）等于TLB entry中的ASID。

（4）请求进行地址翻译的VMID（保存在VTTBR寄存器中）等于TLB entry中的VMID

3、进程切换和ASID(Address Space Identifier)

如果了解OS的基本知识，那么我们都知道：每个进程都有自己独立的虚拟地址空间。如果TLB不标识虚拟地址空间，那么在进程切换的时候，虚拟地址空间也发生了变化，因此TLB中的所有的条目都应该是无效了，可以考虑invalidate all。但是，这么做从功能上看当然没有问题，但是性能收到了很大的影响。

一个比较好的方案是区分Global pages （内核地址空间）和Process-specific pages（参考页表描述符的nG的定义）。对于Global pages，地址翻译对所有操作系统中的进程都是一样的，因此，进程切换的时候，下一个进程仍然需要这些TLB entry，因而不需要flush掉。对于那些Process-specific pages对应的TLB entry，一旦发生切换，而TLB又不能识别的话，那么必须要flush掉上一个进程虚拟地址空间的TLB entry。如果支持了ASID，那么情况就不一样了：对于那些nG的地址映射，它会有一个ASID，对于TLB的entry而言，即便是保存多个相同虚拟地址到不同物理地址的映射也是OK的，只要他们有不同的ASID。

切换虚拟机和VMID的概念是类似的，这里就不多说了。

4、TLB的一致性问题

TLB也是一种cache，有cache也就意味着数据有多个copy，因此存在一致性（coherence）问题。和数据、指令cache或者unified cache不同的是，硬件并不维护TLB的coherence，一旦软件修改了page table，那么软件也需要进行TLB invalidate操作，从而维护了TLB一致性。

5、TLB操作过程

我们以一个普通内存访问指令为例，说明TLB的操作过程，在执行该内存访问指令的过程中，第一件需要完成的任务就是将要访问的虚拟地址翻译成物理地址，具体操作步骤如下：

（1）首先在L1 data TLB中寻找匹配的TLB entry（如果是取指操作，那么会在L1 instruction TLB中寻找），如果运气足够好，TLB hit，那么一切都结束了，否者进入下一步

（2）在L2 TLB中寻找匹配的TLB entry。如果不能命中，那么就需要启动hardware translation table walk了

（3）在执行hardware translation table walk的时候，是直接访问main memory还是通过L2 cache 访问呢？其实都可以的，这和系统配置有关（具体参考TCR\_ELx）。如果配置的是Normal memory, Inner Write-Back Cacheable，那么可以在L2 cache中来寻找page table。如果配置的是Normal memory, Inner Write-Through Cacheable或者Non-cacheable，那么hardware translation table walk将直接和external main memory。

6、维护TLB的指令

我们将在下一章，配合linux的标准TLB flush接口来描述。

四、TLB flush API

和TLB flush操作相关的接口API主要包括：

1、 void flush\_tlb\_all(void)。

这个接口用来invalidate TLB cache中的所有的条目。执行完毕了该接口之后，由于TLB cache中没有缓存任何的VA到PA的转换信息，因此，调用该接口API之前的所有的对page table的修改都可以被CPU感知到。注：该接口是大杀器，不要随便使用。

对于ARM64，flush\_tlb\_all接口使用的底层命令是：tlbi vmalle1is。Tlbi是TLB Invalidate指令，vmalle1is是参数，指明要invalidate那些TLB。vm表示本次invalidate操作对象是当前VMID，all表示要invalidate所有的TLB entry，e1是表示要flush的TLB entry的memory space identifier是EL0和EL1，regime stage 1的TLB entry。is是inner shareable的意思，表示要invalidate所有inner shareable内的所有PEs的TLB。如果没有is，则表示要flush的是local TLB，其他processor core的TLB则不受影响。

flush\_tlb\_all接口有一个变种：local\_flush\_tlb\_all。flush\_tlb\_all是invalidate系统中所有的TLB（各个PEs上的TLB），而local\_flush\_tlb\_all仅仅是invalidate本CPU core上的TLB。local\_flush\_tlb\_all对应的底层接口是：tlbi vmalle1，没有is参数。

2、 void flush\_tlb\_mm(struct mm\_struct \*mm)。

这个接口用来invalidate TLB cache中所有和mm这个进程地址空间相关的条目。执行完毕了该接口之后，由于TLB cache中没有缓存任何的mm地址空间中VA到PA的转换信息，因此，调用该接口API之前的所有的对mm地址空间的page table的修改都可以被CPU感知到。

对于ARM64，flush\_tlb\_mm接口使用的底层命令是：tlbi aside1is, asid。is是inner shareable的意思，表示该操作要广播到inner shareable domain中的所有PEs。asid表示该操作范围是根据asid来进行的。

3、void flush\_tlb\_page(struct vm\_area\_struct \*vma, unsigned long addr)。

flush\_tlb\_page接口函数对addr对应的TLB entry进行flush操作。这个函数类似于flush\_tlb\_range，只不过flush\_tlb\_range操作了一片VA区域，涉及若干TLB entry，而flush\_tlb\_page对range进行了限定（range的size就是一个page），因此，也就只是invalidate addr对应的tlb entry。

对于ARM64，flush\_tlb\_page接口使用的底层命令是：tlbi vale1is, addr。va参数是virtual address的意思，表示本次flush tlb的操作是针对当前asid中的某个virtual address而进行的，addr给出具体要操作的地址和ASID信息。l表示last level，也就是说用户修改了last level Translation table的内容（一般而言，PTE就是last level，在某些情况下，例如section map，last level是PMD），那么我们仅仅需要flush最后一级的页表（page table walk可以有多级）。

4、 void flush\_tlb\_range(struct vm\_area\_struct \*vma, unsigned long start, unsigned long end)。

flush\_tlb\_range接口是一个flush强度比flush\_tlb\_mm要弱的接口，flush\_tlb\_range不是invalidate整个地址空间的TBL，而是针对该地址空间中的一段虚拟内存（start到end-1）在TLB中的entry进行flush。

ARM64并没有直接flush一个range的硬件操作接口，因此，在ARM64的代码中，flush一个range是通过flush一个个的page来实现的。

五、参考文献

1、Cortex A72 TRM

2、ARM ARM

3、Documentation/cachetlb.txt

# 逆向映射的演进

## 一、前言

数学大师陈省身有一句话是这样说的：了解历史的变化是了解这门学科的一个步骤。今天，我把这句话应用到一个具体的Linux模块：了解逆向映射的最好的方法是了解它的历史。本文介绍了Linux内核中的逆向映射机制如何从无到有，如何从笨重到轻盈的历史过程，通过这些历史的演进过程，希望能对逆向映射有更加深入的理解。

## 二、基础知识

在切入逆向映射的历史之前，我们还是简单看看一些基础的概念，这主要包括两个方面：一个是逆向映射的定义，另外一个是引入逆向映射的原因。

1、什么是逆向映射（reverse mapping）？

在聊逆向映射之前，我们先聊聊正向映射好了，当你明白了正向映射，逆向映射的概念也就易如反掌了。所谓正向映射，就是在已知虚拟地址和物理地址（或者page number、page struct）的情况下，为地址映射建立起完整的页表的过程。例如，进程分配了一段VMA之后，并无对应的page frame（即没有分配物理地址），直到程序访问了这段VMA之后，产生异常，由内核为其分配物理页面并建立起所有的各级的translation table。通过正向映射，我们可以将进程虚拟地址空间中的虚拟页面映射到对应的物理页面（page frame）。

逆向映射相反，在已知page frame的情况下（可能是PFN、可能是指向page descriptor的指针，也可能是物理地址，内核有各种宏定义用于在它们之间进行转换），找到映射到该物理页面的虚拟页面们。由于一个page frame可以在多个进程之间共享，因此逆向映射的任务是把分散在各个进程地址空间中的所有的page table entry全部找出来。

一般来说，一个进程的地址空间内不会把两个虚拟地址mapping到一个page frame上去，如果有多个mapping，那么多半是这个page被多个进程共享。最简单的例子就是采用COW的进程fork，在进程没有写的动作之前，内核是不会分配新的page frame的，因此父子进程共享一个物理页面。还有一个例子和c lib相关，由于c lib是基础库，它会file mapping到很多进程地址空间中，那么c lib中的程序正文段对应的page frame应该会有非常多的page table entries与之对应。

2、为何需要逆向映射？

之所以建立逆向映射机制主要是为了方便页面回收。当页面回收机制启动之后，如果回收的page frame是位于内核中的各种内存cache中（例如 slab内存分配器），那么这些页面其实是可以直接回收，没有相关的页表操作。如果回收的是用户进程空间的page frame，那么在回收之前，内核需要对该page frame进行unmapping的操作，即找到所有的page table entries，然后进行对应的修改操作。当然，如果页面是dirty的，我们还需要一些必要的磁盘IO操作。

可以给出一个实际的例子，例如swapping机制，在释放一个匿名映射页面的时候，要求对所有相关的页表项进行更改，将swap area和page slot index写入页表项中。只有在所有指向该page frame的页表项修改完毕后才可以将该页交换到磁盘，并且回收这个page frame。demand paging的场景是类似的，只不过是需要把所有的page table entry清零，这里就不赘述了。

## 三、史前文明

盘古开天辟地之前，宇宙混沌一片。对于逆向映射这个场景，我们的问题就是：没有逆向映射之前，混沌的内核世界是怎样的呢？这一章主要是回答这个问题的，分析的基础是2.4.18内核的源代码。

1、没有逆向映射，系统如何运作？

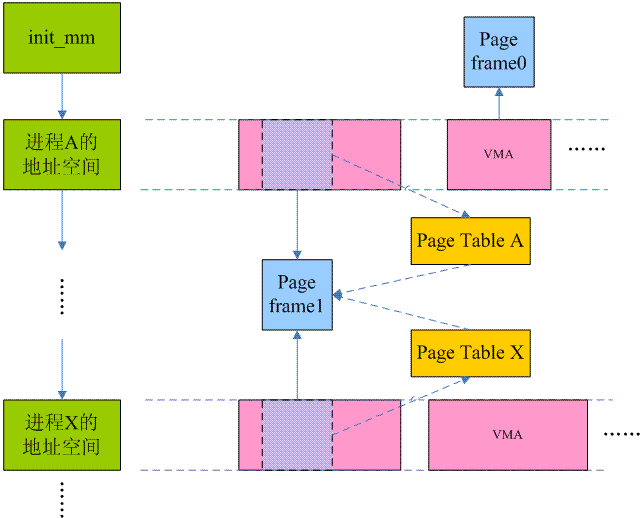
也许年轻的内核工程师很难想象没有逆向映射的内核世界，但实际上2.4时期的内核就是这样的。让我们想象一下，我们自己就是page reclaim机制的维护者，看看我们目前的困境：如果没有逆向映射机制，那么struct page中没有维护任何的逆向映射的数据。这种情况下，内核不可能通过简单的方法来找到page frame所对应的那些PTEs。当回收一个被多个进程共享的page frame，我们该怎么办呢？

本身回收用户进程的物理页帧并不复杂，这需要memory mapping和swapping机制的支持。这两种机制的工作原理类似，只不过一个用于file mapped page，另外一个用于anonymous page。不过对于页面回收而言，他们的工作原理类似：就是把某些进程不常使用的page frame交换到磁盘上去，同时解除进程和这个page frame的一切关系，完成这两步之后，这个物理页帧已经自由了，可以回收到伙伴系统中。

OK，了解了基本原理，现在需要看看如何具体实现：不常使用的page frame很好找（inactive lru链表），不过断绝page frame和进程们之间的关系很难，因为没有逆向映射。不过这难不倒Linux内核开发人员，他们选择了扫描整个系统的各个进程的地址空间的方法。

2、如何对进程地址空间进行扫描？

下图是一个对进程地址空间进行扫描的示意图：

[](http://www.wowotech.net/content/uploadfile/201711/b5e9b4f86ce43ca65bd79c894c4a924c20171117074715.gif)

系统中的所有进程地址空间（memory descriptor）被串成一个链表，链表头就是init\_mm，系统中所有的进程地址空间都挂在了这个链表中。所谓scan当然就是沿着这条mm链表进行了。当然，页面回收算法尽量不scan整个系统的全部进程地址空间，毕竟那是一个比较笨的办法。回收算法可以考虑收缩内存cache，也可以遍历inactive\_list来试图完成本次reclaim数目的要求（该链表中有些page不和任何进程相关），如果通过这些方法释放了足够多的page frame，那么一切都搞定了，不需要scan进程地址空间。当然，情况并非总是那么美好，有时候，必须启动进程物理页面回收过程才能满足页面回收的要求。

进程物理页面回收过程是通过调用swap\_out函数完成的，而scan进程地址空间的代码也是开始于这个函数。该函数是一个三层嵌套结构：

（1） 首先沿着init\_mm，对每一个进程地址空间进行扫描

（2） 在扫描一个进程地址空间的时候，对属于该进程地址空间的每一个VMA进行扫描

（3） 在扫描每一个VMA的时候，对属于该VMA的每一个page进行扫描

在扫描过程中，如果命中了进程A的page frame0，由于该page只是被进程A 使用（即只是被A进程mapping），那么可直接unmap并回收该page。对于共享页面，我们不能这么处理了，例如上图中的page frame 1，但scan A进程的时候，如果条件符合，那么我们会unmap该page，解除它和进程A的关系，当然，这时候不能回收该page，因为进程X还在使用该page。直到scan过程历经千山万水来到进程X，完成对page frame 1的unmaping操作，该物理页面才可以真正会伙伴系统的怀抱。

**3、地址空间扫描的细节问题**

首先，第一个问题：到底scan多少虚拟地址空间才停止scan呢？当目标已经达到的时候，例如本次scan打算reclaim 32个page frame，如果目标达到，那么scan停止，不需scan全部虚拟地址空间。还有一种比较悲惨的情况，那就是scan了系统中所有的地址空间之后，仍然没有达成目标，这时候也就可以停止了，不过这属于OOM的处理了。为了确保系统中的进程被均匀的scan（毕竟swap out会影响进程性能，我们肯定不能只逮住部分进程薅其羊毛），每次scan完成后，记录当前scan的位置（保存在swap\_mm变量），等下次又启动scan过程的时候，从swap\_mm开始继续scan。

由于对性能有影响，swap out需要雨露均沾，各个进程都跑不掉。同样的道理，对于一个进程的地址空间，我们一样也是需要公平对待，因此需要保存每次scan的虚拟地址（mm->swap\_address），这样，每次重启scan的时候，总是从swap\_mm那个地址空间的mm->swap\_address虚拟地址开始scan。

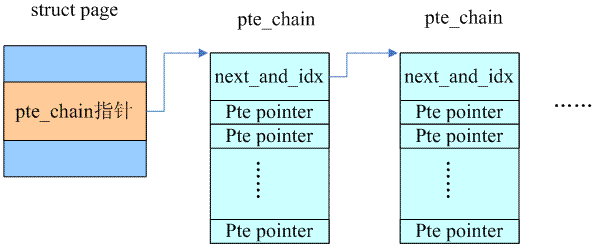
具体对一个page frame进行swap out的代码位于try\_to\_swap\_out函数中，在这个函数中，如果条件满足，会解除该page frame的该进程之间的关系，完成必要的IO操作，该page reference count减一，对应的pte清零或者设定swap entry等。当然，swap out一个page之后，我们并非一定能够回收它，因为这个page很可能被多个进程共享。而在scan过程中，如果碰巧找到了该page对应的所有的页面表条目，那么说明该页面已经不被任何进程引用，这时候该page frame就会被逐出磁盘，从而完成一个页面的回收。

## 四、开天辟地

时间又回到2002年1月，那时VM大神Rik van Riel遭遇了人生中的一次重大挫折，他的耗费心血维护的代码被一个全新的VM子系统取代了。不过Rik van Riel并没有消沉下去，他在憋大招，也就是传说中的reverse mapping（后文简称rmap）。本章主要描述第一个版本的rmap，代码来自Linux 2.6.0。

1、设计概念

如何构建rmap？最直观的想法就是针对每一个page frame，我们维护一个链表，保存属于该page的所有PTEs。因此，Rik van Riel给struct page增加了一个pte chain的成员，以便把所有mapping到该page的pte entry指针给串起来。这样想要unmap一个page就易如反掌了，沿着这个pte chain就可以找到所有的mappings。一个基本的示意图如下，下面的小节会给出更详细的解释。

[](http://www.wowotech.net/content/uploadfile/201711/274a01ad7ad7ad7d73d5f0b399ae5db220171117074717.gif)

2、对Struct page的修改

Struct page的修改如下：

struct page {

……

union {

struct pte\_chain \*chain;

pte\_addr\_t direct;

} pte;

……

当然，很多页面都不是共享的，只有一个pte entry，因此direct直接指向那个pte entry就OK了。如果存在页面共享的情况，那么chain成员则会指向一个struct pte\_chain的链表。

3、定义struct pte\_chain

struct pte\_chain {

unsigned long next\_and\_idx;

pte\_addr\_t ptes[NRPTE];

} \_\_\_\_cacheline\_aligned;

如果pte\_chain只保存一个pte entry的指针那么就太浪费了，比较好的方法是把struct pte\_chain对齐在cache line并让整个struct pte\_chain占用一个cache line。除了next\_and\_idx用于指向下一个pte\_chain，形成链表之外，其余的空间都用于保存pte entry指针。由于pte entry指针形成了数组，因此我们还需要一个index指示下一个空闲的pte entry pointer的位置，由于pte\_chain对齐在cache line，因此next\_and\_idx的LSB的若干个bit是等于0的，可以复用做index。

**4、页面回收算法的修改**

在进入基于rmap的页面回收算法之前，让我们先回忆一下痛苦的过去。假设一个物理页面P被A和B两个进程共享，在过去，释放P这个物理页面需要扫描进程地址空间，首先scan到A进程，解除P和A进程的关系，但是这时候不能回收，B进程还在使用该page frame。当然扫描过程最终会来到B进程，只有在这时候才有机会回收这个物理页面P。你可能会问：如果scan B进程地址空间的时候，A进程又访问了P从而导致映射建立。然后scan A的时候，B进程又再次访问，如此反反复复，那么P不就永远无法回收了吗？这个怎么办呢？这个……理论上是这样的，别问我，其实我也很绝望。

有了rmap，页面回收算法顿时感觉轻松多了，只要是页面回收算法看中的page frame，总是能够通过try\_to\_unmap解除和所有进程的关联，从而将其回收到伙伴系统中。如果该page frame没有共享（page flag设定PG\_direct flag），那么page->pte.direct直接命中pte entry，调用try\_to\_unmap\_one来进行unmap的操作。如果映射到了多个虚拟地址空间，那么沿着pte\_chain依次调用try\_to\_unmap\_one来进行unmap的操作。

## 五、女娲补天

虽然Rik van Riel开辟了逆向映射的新天地，但是，天和地都有着巨大的窟窿，需要有人修补。首先让我们看看这个“巨大的窟窿”是什么？

在引入第一个版本的rmap之后，Linux的页面回收变得简单、可控了，但是这个简单的设计是有代价的：每一个struct page增加一个指针成员，在32bit的系统上也就是增加了4B。考虑到系统为了管理内存会为每一个page frame建立一个struct page对象，引入rmap而导致的内存开销也不是一个小数目啊。此外，share page需要建立pte\_chain链表，也是一个不小的内存开销。除了内存方面的压力，第一个版本的rmap对性能也造成了一定的影响。例如：在fork操作的时候，父子进程共享了很多的page frame，这样，在copy page table的时候就会伴随大量的pte\_chain的操作，从而让fork的速度变得缓慢。

本章就是带领大家看看object-based reverse mapping（后文简称objrmap）是如何填补那个“巨大的窟窿”。本章的代码来自2.6.11版本的内核。

1、问题的引入

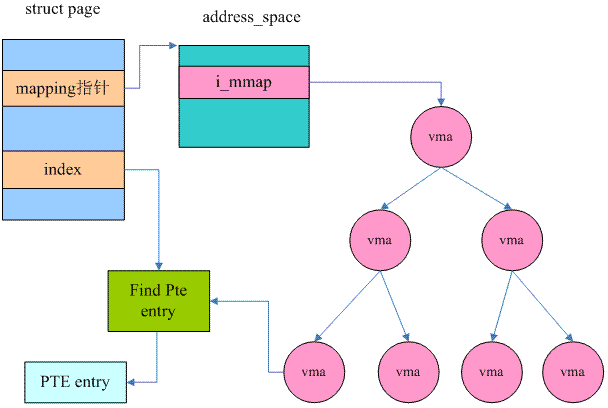
推动rmap优化的动力来自内存方面的压力，与此相关的问题是：32-bit的Linux内核是否支持4G以上的memory。在1999年，Linus的决定是：32-bit的Linux内核永远也不会支持2G以上的内存。不过历史的洪流不可阻挡，处理器厂商设计了扩展模块以便寻址更多的内存，高端的服务器也配置了越来越多的内存。这也迫使Linus改变之前的思路，让Linux内核支持更大的内存。

红帽公司的Andrea Arcangeli当时正在做的工作就是让32-bit的Linux运行在配置超过32G内存的公司服务器上。在这些服务器上往往启动大量的进程，共享了大量的物理页帧，消耗了大量的内存。对于Andrea Arcangeli来说，内存消耗的真正元凶是明确的：逆向映射模块，这个模块消耗了太多的low memory，从而导致了系统的各种crash。为了让自己的工作继续推进，他必须解决rmap引入的内存扩展性（memory scalability）问题。

2、file mapped的优化

并非只有Andrea Arcangeli关注到了rmap的内存问题，在2.5版本的开发过程中，IBM公司的Dave McCracken就已经提交了patch，试图在保证逆向映射功能的基础上，同时又能修正rmap带来的各种问题。

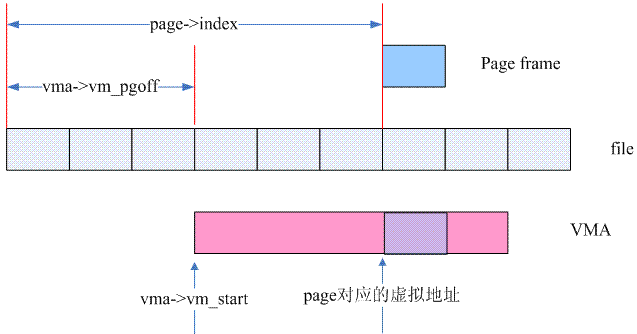
Dave McCracken的方案是一种基于对象的逆向映射机制。在过去，通过rmap，我们可以从struct page直接获取其对应的ptes，objrmap的方法借助其他的数据对象来完成从struct page检索到其对应ptes的过程，这个过程的示意图如下：

[](http://www.wowotech.net/content/uploadfile/201711/068ae40523a24c9ef54edefd375e542d20171117074718.gif)

对于objrmap而言，寻找一个page frame的mappings是一个比较长的路径，它借助了VMA（struct vm\_area\_struct）这个数据对象。我们知道对于某些page frame是有后备文件的，这种类型的页面和某个文件相关，例如进程的正文段和该进程的可执行文件相关。此外，进程可以调用mmap()对某个文件进行mapping。对于这些页帧我们称之file mapped page。

对于这些文件映射页面，其struct page中有一个成员mapping指向一个struct address\_space，address\_space是和文件相关的，它保存了文件page cache相关的信息。当然，我们这个场景主要关注一个叫做i\_mmap的成员。一个文件可能会被映射到多个进程的多个VMA中，所有的这些VMA都被挂入到i\_mmap指向的Priority search tree中。

当然，我们最终的目标是PTEs，下面这幅图展示了如何从VMA和struct page中的信息导出该page frame的虚拟地址的：

[](http://www.wowotech.net/content/uploadfile/201711/8f173d0eaffc4b90c0c0361b8f37cc1720171117074720.gif)

而在linux kernel中，函数vma\_address可以完成这个功能：

static inline unsigned long

vma\_address(struct page \*page, struct vm\_area\_struct \*vma)

{

pgoff\_t pgoff = page->index << (PAGE\_CACHE\_SHIFT - PAGE\_SHIFT);

unsigned long address;

address = vma->vm\_start + ((pgoff - vma->vm\_pgoff) << PAGE\_SHIFT);

return address;

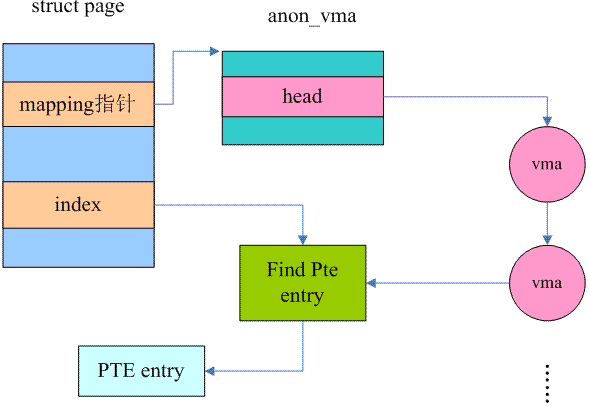
}

对于file mapped page，page->index表示的是映射到文件内的偏移（Byte为单位），而vma->vm\_pgoff表示的是该VMA映射到文件内的偏移（page为单位），因此，通过vma->vm\_pgoff和page->index可以得到该page frame在VMA中的地址偏移，再加上vma->vm\_start就可以得到该page frame的虚拟地址。有了虚拟地址和地址空间（vma->vm\_mm），我们就可以通过各级页表找到该page对应的pte entry。

**3、匿名页面的优化**

我们都知道，用户空间进程的页面主要有两种，一种是file mapped page，另外一种是anonymous mapped page。Dave McCracken的objrmap方案虽好，但是只是适用于file mapped page，对于匿名映射页面，这个方案无能为力。因此，我们必须为匿名映射页面也设计一种基于对象的逆向映射机制，最后形成full objrmap方案。

为了解决内存扩展性的问题，Andrea Arcangeli全力工作在full objrmap方案上，不过他还有一个竞争对手，Hugh Dickins，同时也提交了一系列full objrmap补丁，试图并入内核主线，显然，在匿名映射页面上，最后胜出的是Andrea Arcangeli，他的匿名映射方案如下图所示：

[](http://www.wowotech.net/content/uploadfile/201711/6a0fe32c3736658bde52846b7df05e0120171117074722.gif)

和file mapped类似，anonymous page也是通过VMA来寻找page frame对应的pte entry。由于文件映射页面的VMA数量可能非常大，因此我们采用Priority search tree这样的数据结构。对于匿名映射页面，其数量一般不会太大，所以使用链表结构就OK了。

为了节省内存，我们复用了struct page中的mapping指针：一个page frame如果是file mapped，其mapping指针指向对应文件的address\_space数据结构。如果是anonymous page，那么mapping指针指向anon\_vma数据结构。虽然节省了内存，但是降低了可读性，但是由于内核会为每一个page frame建立一个对应的struct page数据对象，该数据结构即便是增加4B对整个系统的内存消耗都是巨大的，因此内核还是采用了较为丑陋的方式来定义mapping这个成员。通过struct page中的mapping成员我们可以获得该page映射相关的信息，总结如下：

（1） 等于NULL，表示该page frame不再内存中，而是被swap out到磁盘去了。

（2） 如果不等于NULL，并且least signification bit等于1，表示该page frame是匿名映射页面，mapping指向了一个anon\_vma的数据结构。

（3） 如果不等于NULL，并且least signification bit等于0，表示该page frame是文件映射页面，mapping指向了一个该文件的address\_space数据结构。

通过anon\_vma数据结构，我们可以得到映射到该page的所有的VMA，至此，匿名映射和file mapped汇合，进一步解决的问题仅仅是如何从VMA到pte entry而已。上一节，我们描述了vma\_address函数如何获取file mapped page的虚拟地址，其实anonymous page的逻辑是一样的，只不过vma->vm\_pgoff和page->index的基础点不一样了，对于file mapped的场景，这个基础点是文件起始位置。对于匿名映射，起始点有两种情况，一种是share anonymous mapping，起点位置是0。另外一种是private anonymous mapping，起点位置是mapping的虚拟地址（除以page size）。但是不管如何，从VMA和struct page得到对应虚拟地址的算法概念是类似的。

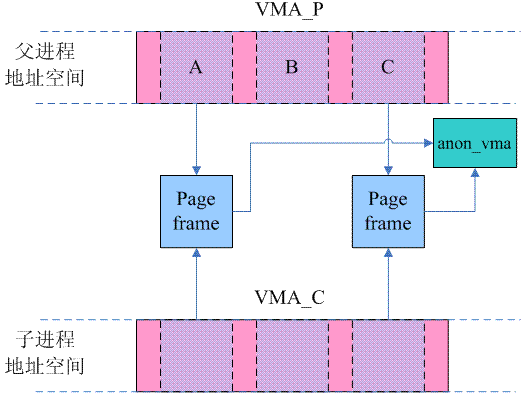
**六、卷土重来**

full objrmap进入内核之后，看起来一切都很完美了，比起她的前任，Rik van Riel的rmap方案，objrmap各方面的指标都是全面碾压rmap。首次将逆向映射引入内核的大神Rik van Riel遭受了第二次的打击，不过他依然斗志昂扬并试图东山再起。

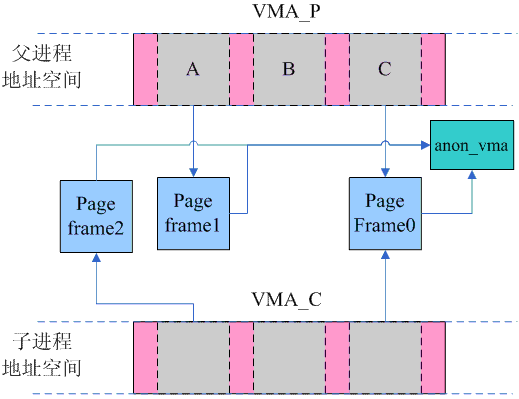
Objrmap虽然完美，不过晴朗的天空中飘着一朵乌云。大神Rik van Riel敏锐的看到了逆向映射的那朵“乌云“，提出了自己的解决方案。本章主要描述新的anon\_vma机制，代码来自4.4.6内核。

1、旧anon\_vma机制有什么问题？

我们先一起来看看旧anon\_vma机制下，系统是如何运作的。VMA\_P是父进程的一个匿名映射的VMA，A和C都已经分配了page frame，而其他的page都还都没有分配物理页面。在fork之后，子进程copy了VMA\_P，当然由于采用了COW技术，这时候父子进程的匿名页面会共享，同时在父子进程地址空间对应的pte entry中标注write protect的标记，如下图所示：

[](http://www.wowotech.net/content/uploadfile/201711/bde7cf9049e122a728d29f2213bb2b6720171117074724.gif)

按理说不同进程的匿名页面（例如stack、heap）是私有的，不会共享，但是为了节省内存，在父进程fork子进程之后，父子进程对该页面执行写操作之前，父子进程的匿名页是共享的，所以这些page frame指向同一个anon\_vma。当然，共享只是短暂的，一旦有write操作就会产生异常，并在异常处理中分配page frame，解除父子进程匿名页面的共享，具体如下图的page A所示：

[](http://www.wowotech.net/content/uploadfile/201711/e746f170e2a60a973f6f1b2abdbfa5a720171117074726.gif)

这时候由于写操作，父子进程原本共享的page frame已经不再共享，然而，这两个page却仍然指向同一个anon\_vma，不仅如此，对于B这样的页面，一开始就没有在父子进程之间共享，当首次访问的时候（无论是父进程还是子进程），通过do\_anonymous\_page函数分配的page frame也是同样的指向一个anon\_vma。也就是说，父子进程的VMA共享一个anon\_vma。

在这种情况下，我们看看unmap page frame1会发生什么。毫无疑问，page frame1对应的struct page的mapping成员指向了上图中的anon\_vma，遍历anon\_vma会命VMA\_P和VMA\_C，这里面，VMA\_C是无效的VMA，本来就不应该匹配到。如果anon\_vma的链表没有那么长，那么整体性能也OK。然而，在有些网路服务器中，系统非常依赖fork，某个服务程序可能会fork巨大数量的子进程来处理服务请求，在这种情况下，系统性能严重下降。Rik van Riel给出了一个具体的示例：系统中有1000进程，都是通过fork生成的，每个进程的VMA有 1000个匿名页。根据目前的软件架构，anon\_vma链表中会有1000个vma 的节点，而系统中有一百万个匿名页面属于同一个anon\_vma。

这样的系统会导致什么样的问题呢？我们一起来看看try\_to\_unmap\_anon函数，其代码框架如下：

static int try\_to\_unmap\_anon(struct page \*page)

{……

anon\_vma = page\_lock\_anon\_vma(page);

list\_for\_each\_entry(vma, &anon\_vma->head, anon\_vma\_node) {

ret = try\_to\_unmap\_one(page, vma);

}

spin\_unlock(&anon\_vma->lock);

return ret;

}

当系统中的一个CPU在执行try\_to\_unmap\_anon函数的时候，需要遍历VMA链表，这时会持有anon\_vma->lock这个自旋锁。由于anon\_vma存有了很多根本无关的VMA，通过，page table的检索过程，你就会发现这个VMA根本和准备unmap的page无关，因此只能scan下一个VMA，整个过程需要消耗大量的时间，延长了临界区（复杂度是O（N））。与此同时，其他CPU在试获取这把锁的时候，基本会被卡住，这时候整个系统的性能可想而知了。更加糟糕的是内核中并非只有unmap匿名页面的时候会上锁、遍历VMA链表，还有一些其他的场景也会这样（例如page\_referenced函数）。想象一下，一百万个页面共享这一个anon\_vma，对anon\_vma->lock自旋锁的竞争那是相当的激烈啊。

2、改进的方案

旧的方案的症结所在是anon\_vma承载了太多进程的VMA了，如果能将其变成per-process的，那么问题就解决了。Rik van Riel的解决办法是为每一个进程创建一个anon\_vma结构并通过各种数据结构把父子进程的anon\_vma（后面简称AV）以及VMA链接在一起。为了链接anon\_vma，内核引入了一个新的结构，称为anon\_vma\_chain（后面简称AVC）：

struct anon\_vma\_chain {

struct vm\_area\_struct \*vma;――指向该AVC对应的VMA

struct anon\_vma \*anon\_vma;――指向该AVC对应的AV

struct list\_head same\_vma; ――链接入VMA链表的节点

struct rb\_node rb;―――链接入AV红黑树的节点

unsigned long rb\_subtree\_last;

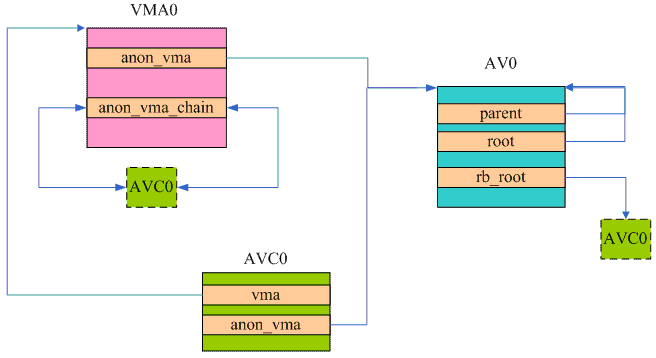
};

AVC是一个神奇的结构，每个AVC都有其对应的VMA和AV。所有指向相同VMA的AVC会被链接到一个链表中，链表头就是VMA的anon\_vma\_chain成员。而一个AV会管理若干的VMA，所有相关的VMA（其子进程或者孙进程）都挂入红黑树，根节点就是AV的rb\_root成员。

这样的描述非常枯燥，估计第一次接触逆向映射的同学是不会明白的，不如我们一起来看看AV、AVC和VMA的“大厦”是如何搭建起来的。

3、当VMA和VA首次相遇

由于采用了COW技术，子进程和父进程的匿名页面往往是共享的，直到其中之一发起写操作。但是如果子进程执行了exec的系统调用，加载了自己的二进制image，这时候，子进程和父进程的执行环境（包括匿名页面）就分道扬镳了（参考flush\_old\_exec函数），我们的场景就是从这么一个全新的exec后的进程开始。当该进程的匿名映射VMA通过page fault分配第一个page frame的时候，内核会构建下图所示的数据关系：

[](http://www.wowotech.net/content/uploadfile/201711/40af9ca5d4af4f53bc05e0396f2436f020171117074727.gif)

上图中的AV0就是该进程的anon\_vma，由于它是一个顶级结构，因此它的root和parent都是指向了自己。AV这个数据结构当然为了管理VMA了，不过新机制中，这是通过AVC进行中转的。上图中的AVC0搭建了该进程VMA和AV之间的桥梁，分别有指针指向了VMA0和AV0，此外，AVC0插入到AV的红黑树，同时也会插入到VMA的链表中。

对于这个新分配的page frame而言，它会mapping到VMA对应的某个虚拟地址上去，为了维护逆向映射的关系，struct page中的mapping指向了AV0，index成员指向了该page在整个VMA0中的偏移。图中没有画出这个关系，主要因为这是老生常谈了，相信大家都已经熟悉。

VMA0中随后可能会有若干的page frame被mapping到该VMA的某个虚拟页面，不过上面的结构不会变化，只不过每一个page中的mapping都指向了上图中的AV0。另外，上图中那个虚线绿色block的AVC0其实等于那个绿色实线的AVC0 block，也就是说这时候该VMA只有一个anon\_vma\_chain，即AVC0，上图只是方便表示该AVC也会被挂入VMA的链表，挂入anon\_vma的红黑树而已。

如果想参考相关的代码可以仔细看看do\_anonymous\_page或者do\_cow\_fault。

4、在fork的时候，匿名映射的VMA经历了什么？

一旦fork，那么子进程会copy父进程的VMA（参考函数dup\_mmap），子进程会有自己的VMA，同时也会分配自己的AV（旧的机制下，多个进程共享一个AV，而新的机制中，AV是per process的），然后建立父子进程之间的VMA、VA的“大厦”，主要的步骤如下：

（1） 调用anon\_vma\_clone函数，建立子进程VMA和“父进程们”VA的关系

（2） 建立子进程VMA和子进程VA的关系

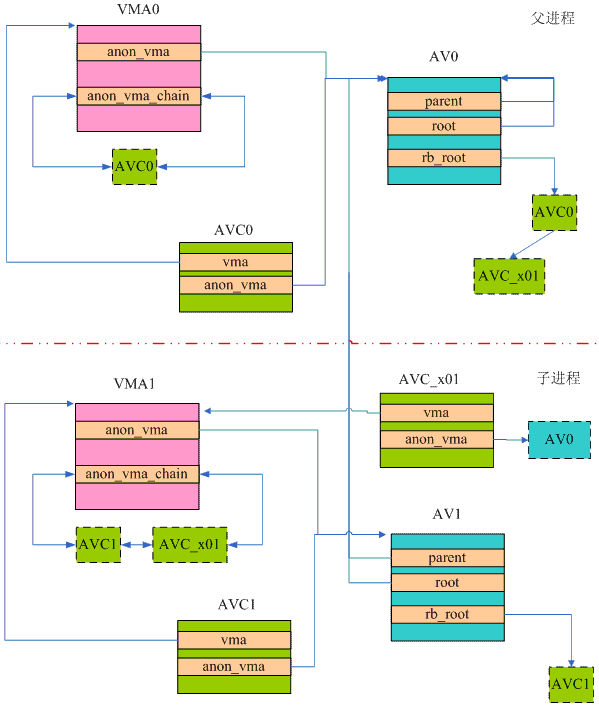
怎样叫做建立VMA和VA的关系？其实就是anon\_vma\_chain\_link函数的调用过程，步骤如下：

（1） 分配一个AVC结构，成员指针指向对应的VMA和VA

（2） 将该AVC加入VMA链表

（3） 将该AVC加入VA红黑树

我们一开始先别把事情搞得太复杂，先看看一个全新进程fork子进程的场景。这时候，内核会构建下图所示的数据关系：

[](http://www.wowotech.net/content/uploadfile/201711/3019f422209a6593f02fad5560f36b7920171117074729.gif)

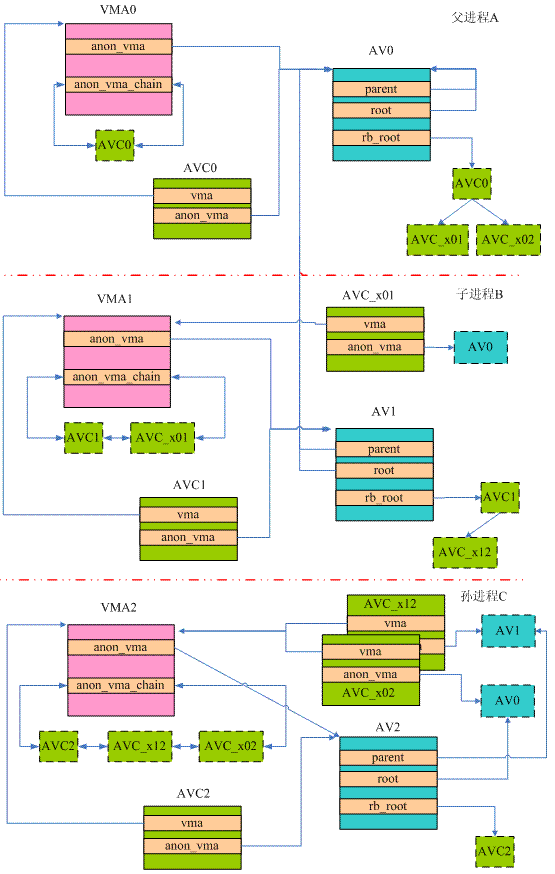
首先看看如何建立子进程VMA1和父进程AV0的关系，这里需要遍历VMA0的anon\_vma\_chain链表，当然现在这个链表只有一个AVC0（link到AV0），为了建立和父进程的联系，我们分配了AVC\_x01，它是一个桥梁，连接了父子进程。（注：AVC\_x01中的x表示连接，01表示连接level 0和level 1）。通过这个桥梁，父进程可以找到子进程的VMA（因为AVC\_x01插入AV0的红黑树中），而子进程也可以找到父进程的AV（因为AVC\_x01插入VMA1的链表中）。

当然，自己的anon\_vma也需要创建。在上图中，AV1就是子进程的anon\_vma，同时分配一个AVC1来连接该子进程的VMA1和AV1，并调用anon\_vma\_chain\_link函数将AVC1插入VMA1的链表和AV1的红黑树中。

父进程也会创建其他新的子进程，新创建的子进程的层次和VMA1、VA1的类似，这里就不描述了。不过需要注意的是：父进程每创建一个子进程，AV0的红黑树中会增加每一个起“桥梁”作用的AVC，以此连接到子进程的VMA。

5、构建三层大厦

上一节描述了父进程创建子进程的情况，如果子进程再次fork，那么整个VMA-VA的大厦将形成三层结构，具体如下图所示：

[](http://www.wowotech.net/content/uploadfile/201711/575eda61e93c7ad9d05928580213fd9b20171117074731.gif)

当然，首先要进行的仍然是建立孙进程VMA和“父进程们”VA的关系，这里的“父进程们”其实是泛指孙进程的上层的那些进程们。对于这个场景，“父进程们”指的就是上图中的A进程和B进程。如何建立？在fork的时候，我们进行VMA的拷贝：即分配VMA2并以VMA1为原型copy到VMA2中。Copy是沿着VMA1的AVC链表进行的，该链表有两个元素：AVC1和 AVC\_x01，分别和父进程A和子进程B的AV关联。因此，在孙进程C中，我们会分配AVC\_x02和AVC\_x12两个AVC，并建立level 2层和level 0层以及level 1层之间的关系。

同样的，自己level的anon\_vma也需要创建。在上图中，AV2就是孙进程C的anon\_vma，同时分配一个AVC2来连接该孙进程的VMA2和AV2，并调用anon\_vma\_chain\_link函数将AVC2插入VMA2的链表和AV2的红黑树中。

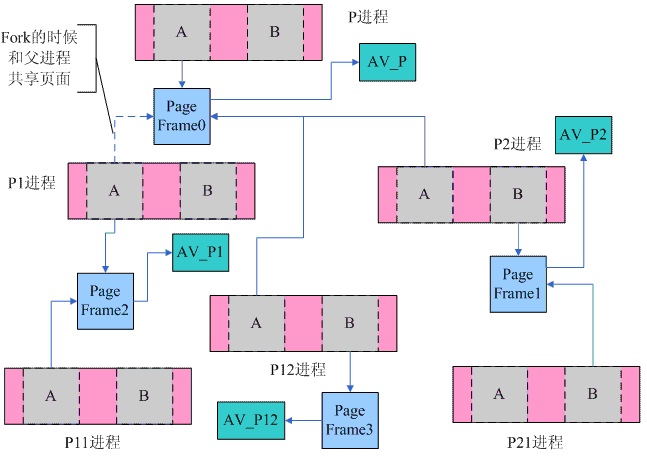
AV2中的root指向root AV，也就是进程A的AV。Parent成员指向其B进程（C的父进程）的AV。通过Parent这样的指针，不同level的AV建立了父子关系，而通过root指针，每一个level的AV都可以寻找找到root AV。

6、page frame是如何加入“大厦”中？

前面几个小节重点讨论了hierarchy AV的结构是如何搭建起来的，也就是描述fork的过程中，父子进程的VMA、AVC和AV是如何联系的。本小节我们将一起来看看父子进程之一访问页面，发生了page fault的处理过程。这个处理过程有两个场景，一个是父子进程都没有page frame，这时候，内核代码会调用do\_anonymous\_page分配page frame并调用page\_add\_new\_anon\_rmap函数建立该page和对应VMA的关系。第二个场景复杂一点，是父子共享匿名页面的场景，当发生write fault的时候，也是分配page frame并调用page\_add\_new\_anon\_rmap函数建立该page和对应VMA的关系，具体代码位于do\_wp\_page函数。无论哪一个场景，最终都是将该page的mapping成员指向了该进程所属的AV结构。

7、为何建立如此复杂的“大厦”？

如果你能坚持读到这里，那么说明你对枯燥文字的忍受能力还是很强的，哈哈。Page、VMA、VAC、VA组成了如此复杂的层次结构到底是为什么呢？是为了打击你学习内核的兴趣吗？非也，让我们还是用一个实际的场景来说明这个“大厦”的功能。

[](http://www.wowotech.net/content/uploadfile/201711/007b98d50fd1de35a654e100103ddac620171117074735.gif)

我们通过下面的步骤建立起上图的结构：

（1） P进程的某个VMA中有两类页面： 一类是有真实的物理页面的，另外一类是还没有配备物理页面的。上图中，我们分别跟踪有物理页面的A以及还没有分配物理页面的B。

（2） P进程fork了P1和P2

（3） P1进程fork了P12进程

（4） P1进程访问了A页面，分配了page frame2

（5） P12进程访问了B页面，分配了page frame3

（6） P2进程访问了B页面，分配了page frame1

（7） P2进程fork了P21进程

经过上面的这一些动作之后，我们来看看page frame共享的情况：对于P进程的page frame（是指该page 的mapping成员指向P进程的AV，即上图中的AV\_P）而言，他可能会被任何一个level的的子进程VMA中的page所有共享，因此AV\_P需要包括其子进程、孙进程……的所有的VMA。而对于P1进程而言，AV\_P1则需要包括P1子进程、孙进程……的所有的VMA，有一点可以确认：至少父进程P和兄弟进程P2的VMA不需要包括在其中。

现在我们回头看看AV结构的大厦，实际上是符合上面的需求的。

8、页面回收的时候，如何unmap一个page frame的所有的映射？

搭建了那么复杂的数据结构大厦就是为了应用，我们一起看看页面回收的场景。这个场景需要通过page frame找到所有映射到该物理页面的VMAs。有了前面的铺垫，这并不复杂，通过struct page中的mapping成员可以找到该page对应的AV，在该AV的红黑树中，包含了所有的可能共享匿名页面的VMAs。遍历该红黑树，对每一个VMA调用try\_to\_unmap\_one函数就可以解除该物理页帧的所有映射。

OK，我们再次回到这一章的开始，看看那个长临界区导致的性能问题。假设我们的服务器上有一个服务进程A，它fork了999个子进程来为世界各地的网友服务，进程A有一个VMA，有1000个page。下面我们就一起来对比新旧机制的处理过程。

首先，百万page共享一个anon\_vma的情况在新机制中已经解决，每一个进程都有自己特有的anon\_vma对象，每一个进程的page都指向自己特有的anon\_vma对象。在旧的机制中，每次unmap一个page都需要扫描1000个VMA，而在新的机制中，只有顶层的父进程A的AV中有1000个VMA，其他的子进程的VMA的数目都只有1个，这大大降低了临界区的长度。

七、后记

本文带领大家一起简略的了解了逆向映射的发展过程。当然，时间的车轮永不停息，逆向映射机制还在不断的修正，如果你愿意，也可以了解其演进过程的基础上，提出自己的优化方案，在其历史上留下自己的印记。

# KASAN实现原理

## 1. 前言

KASAN是一个动态检测内存错误的工具。KASAN可以检测全局变量、栈、堆分配的内存发生越界访问等问题。功能比SLUB DEBUG齐全并且支持实时检测。越界访问的严重性和危害性通过我之前的文章（SLUB DEBUG技术）应该有所了解。正是由于SLUB DEBUG缺陷，因此我们需要一种更加强大的检测工具。难道你不想吗？KASAN就是其中一种。KASAN的使用真的很简单。但是我是一个追求刨根问底的人。仅仅止步于使用的层面，我是不愿意的，只有更清楚的了解实现原理才能更加熟练的使用工具。不止是KASAN，其他方面我也是这么认为。但是，说实话，写这篇文章是有点底气不足的。因为从我查阅的资料来说，国内没有一篇文章说KASAN的工作原理，国外也是没有什么文章关注KASAN的原理。大家好像都在说How to use。由于本人水平有限，就根据现有的资料以及自己阅读代码揣摩其中的意思。本文章作为抛准引玉，如果有不合理的地方还请指正。  
注：文章代码分析基于linux-4.15.0-rc3。

图片显示有点走形，请点开查看。

## 2. 简介

KernelAddressSANitizer（KASAN）是一个动态检测内存错误的工具。它为找到use-after-free和out-of-bounds问题提供了一个快速和全面的解决方案。KASAN使用编译时检测每个内存访问，因此您需要GCC 4.9.2或更高版本。检测堆栈或全局变量的越界访问需要GCC 5.0或更高版本。目前KASAN仅支持x86\_64和arm64架构（linux 4.4版本合入）。你使用ARM64架构，那么就需要保证linux版本在4.4以上。当然了，如果你使用的linux也有可能打过KASAN的补丁。例如，使用高通平台做手机的厂商使用linux 3.18同样支持KASAN。

## 3. 如何使用

使用KASAN工具是比较简单的，只需要添加kernel以下配置项。

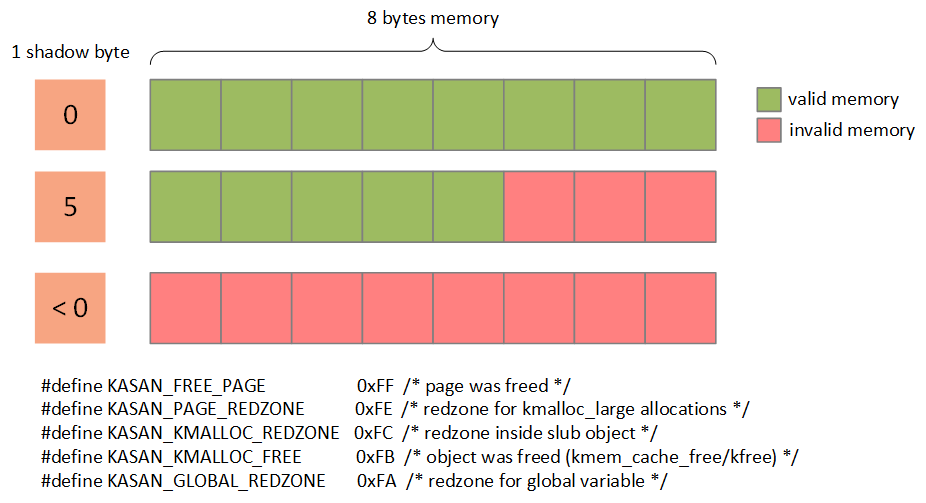
CONFIG\_SLUB\_DEBUG=y

CONFIG\_KASAN=y

为什么这里必须打开SLUB\_DEBUG呢？是因为有段时间KASAN是依赖SLUBU\_DEBUG的，什么意思呢？就是在Kconfig中使用了depends on，明白了吧。不过最新的代码已经不需要依赖了，可以看下提交。但是我建议你打开该选项，因为log可以输出更多有用的信息。重新编译kernel即可，编译之后你会发现boot.img（Android环境）大小大了一倍左右。所以说，影响效率不是没有道理的。不过我们可以作为产品发布前的最后检查，也可以排查越界访问等问题。我们可以查看内核日志内容是否包含KASAN检查出的bugs信息。

## 4. KASAN是如何实现检测的？

KASAN的原理是利用额外的内存标记可用内存的状态。这部分额外的内存被称作shadow memory（影子区）。KASAN将1/8的内存用作shadow memory。使用特殊的magic num填充shadow memory，在每一次load/store（load/store检查指令由编译器插入）内存的时候检测对应的shadow memory确定操作是否valid。连续8 bytes内存（8 bytes align）使用1 byte shadow memory标记。如果8 bytes内存都可以访问，则shadow memory的值为0；如果连续N(1 =< N <= 7) bytes可以访问，则shadow memory的值为N；如果8 bytes内存访问都是invalid，则shadow memory的值为负数。

[](http://www.wowotech.net/content/uploadfile/201802/4a471518360139.png)

在代码运行时，每一次memory access都会检测对应的shawdow memory的值是否valid。这就需要编译器为我们做些工作。编译的时候，在每一次memory access前编译器会帮我们插入\_\_asan\_load##size()或者\_\_asan\_store##size()函数调用（size是访问内存字节的数量）。这也是要求更新版本gcc的原因，只有更新的版本才支持自动插入。

mov x0, #0x5678  
movk x0, #0x1234, lsl #16  
movk x0, #0x8000, lsl #32  
movk x0, #0xffff, lsl #48  
mov w1, #0x5  
bl \_\_asan\_store1  
strb w1, [x0]  
上面一段汇编指令是往0xffff800012345678地址写5。在KASAN打开的情况下，编译器会帮我们自动插入bl \_\_asan\_store1指令，\_\_asan\_store1函数就是检测一个地址对应的shadow memory的值是否允许写1 byte。蓝色汇编指令就是真正的内存访问。因此KASAN可以在out-of-bounds的时候及时检测。\_\_asan\_load##size()和\_\_asan\_store##size()的代码在mm/kasan/kasan.c文件实现。

4.1. 如何根据shadow memory的值判断内存访问操作是否valid？

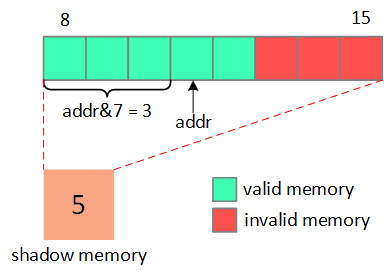
shadow memory检测原理的实现主要就是\_\_asan\_load##size()和\_\_asan\_store##size()函数的实现。那么KASAN是如何根据访问的address以及对应的shadow memory的状态值来判断访问是否合法呢？首先看一种最简单的情况。访问8 bytes内存。

long \*addr = (long \*)0xffff800012345678;  
\*addr = 0;

以上代码是访问8 bytes情况，检测原理如下：

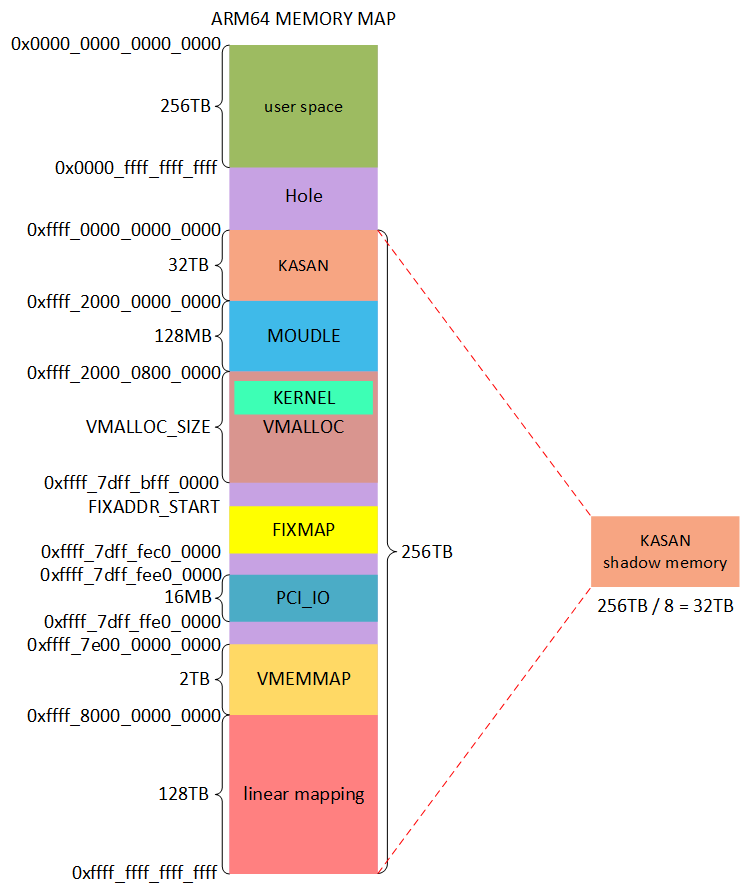
long \*addr = (long \*)0xffff800012345678;  
char \*shadow = (char \*)(((unsigned long)addr >> 3) + KASAN\_SHADOW\_OFFSE);  
if (\*shadow)  
    report\_bug();  
\*addr = 0;

红色区域类似是编译器插入的指令。既然是访问8 bytes，必须要保证对应的shadow mempry的值必须是0，否则肯定是有问题。那么如果访问的是1,2 or 4 bytes该如何检查呢？也很简单，我们只需要修改一下if判断条件即可。修改如下：  
if (\*shadow && \*shadow < ((unsigned long)addr & 7) + N); //N = 1,2,4  
如果\*shadow的值为0代表8 bytes均可以访问，自然就不需要report bug。addr & 7是计算访问地址相对于8字节对齐地址的偏移。还是使用下图来说明关系吧。假设内存是从地址8~15一共8 bytes。对应的shadow memory值为5，现在访问11地址。那么这里的N只要大于2就是invalid。

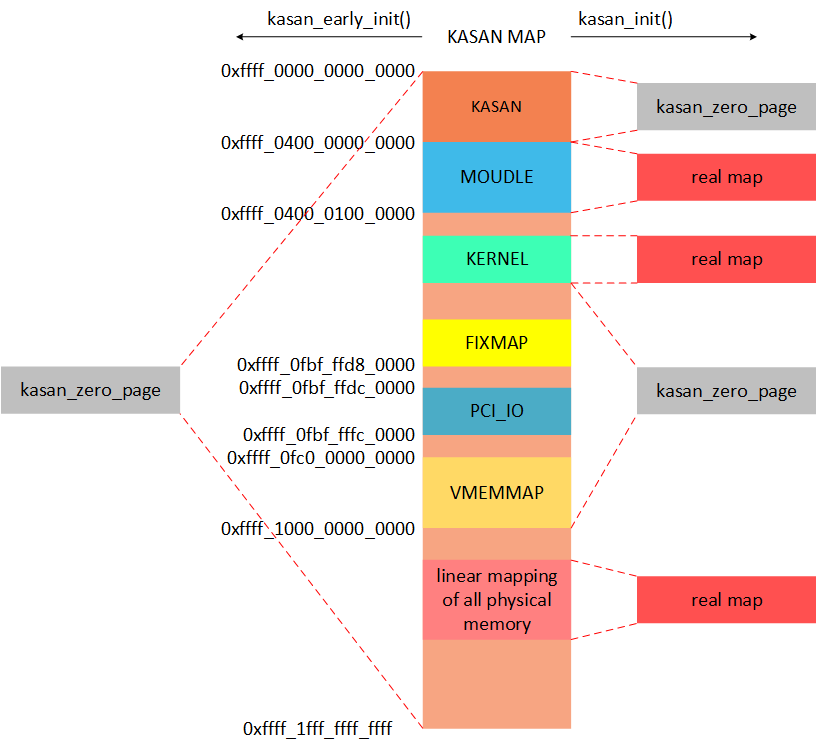
[](http://www.wowotech.net/content/uploadfile/201802/fb5c1519483501.png)  
   
4.2. shadow memory内存如何分配？

在ARM64中，假设VA\_BITS配置成48。那么kernel space空间大小是256TB，因此shadow memory的内存需要32TB。我们需要在虚拟地址空间为KASAN shadow memory分配地址空间。所以我们有必要了解一下ARM64 memory layout。

基于linux-4.15.0-rc3的代码分析，我绘制了如下memory layout（VA\_BITS = 48）。kernel space起始虚拟地址是0xffff\_0000\_0000\_0000，kernel space被分成几个部分分别是KASAN、MODULE、VMALLOC、FIXMAP、PCI\_IO、VMEMMAP以及linear mapping。其中KASAN的大小是32TB，正好是kernel space大小的1/8。不知道你注意到没有，KERNEL的位置相对以前是不是有所不一样。你的印象中，KERNEL是不是位于linear mapping区域，这里怎么变成了VMALLOC区域？这里是Ard Biesheuvel提交的修改。主要是为了迎接ARM64世界的KASLR（which allows the kernel image to be located anywhere in the vmalloc area）的到来。

[](http://www.wowotech.net/content/uploadfile/201802/10fb1518360141.png)  
 4.3. 如何建立shadow memory的映射关系？

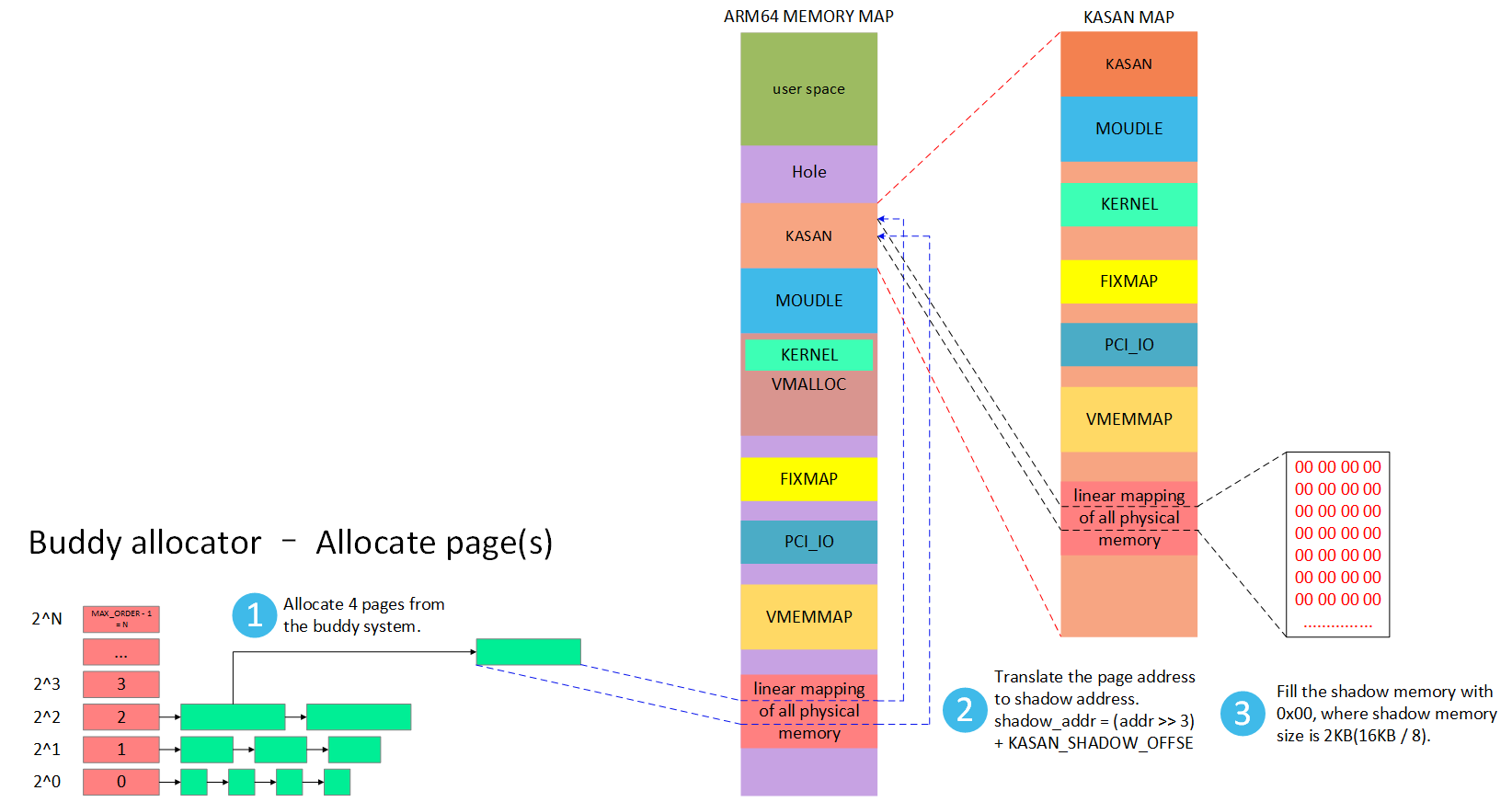
当打开KASAN的时候，KASAN区域位于kernel space首地址处，从0xffff\_0000\_0000\_0000地址开始，大小是32TB。shadow memory和kernel address转换关系是：shadow\_addr = (kaddr >> 3)  + KASAN\_SHADOW\_OFFSE。为了将[0xffff\_0000\_0000\_0000, 0xffff\_ffff\_ffff\_ffff]和[0xffff\_0000\_0000\_0000, 0xffff\_1fff\_ffff\_ffff]对应起来，因此计算KASAN\_SHADOW\_OFFSE的值为0xdfff\_2000\_0000\_0000。我们将KASAN区域放大，如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/09dd1518360142.png)

KASAN区域仅仅是分配的虚拟地址，在访问的时候必须建立和物理地址的映射才可以访问。上图就是KASAN建立的映射布局。左边是系统启动初期建立的映射。在kasan\_early\_init()函数中，将所有的KASAN区域映射到kasan\_zero\_page物理页面。因此系统启动初期，KASAN并不能工作。右侧是在kasan\_init()函数中建立的映射关系，kasan\_init()函数执行结束就预示着KASAN的正常工作。我们将不需要address sanitizer功能的区域同样还是映射到kasan\_zero\_page物理页面，并且是readonly。我们主要是检测kernel和物理内存是否存在UAF或者OOB问题。所以建立KERNEL和linear mapping（仅仅是所有的物理地址建立的映射区域）区域对应的shadow memory建立真实的映射关系。MOUDLE区域对应的shadow memory的映射关系也是需要创建的，但是映射关系建立是动态的，他在module加载的时候才会去创建映射关系。

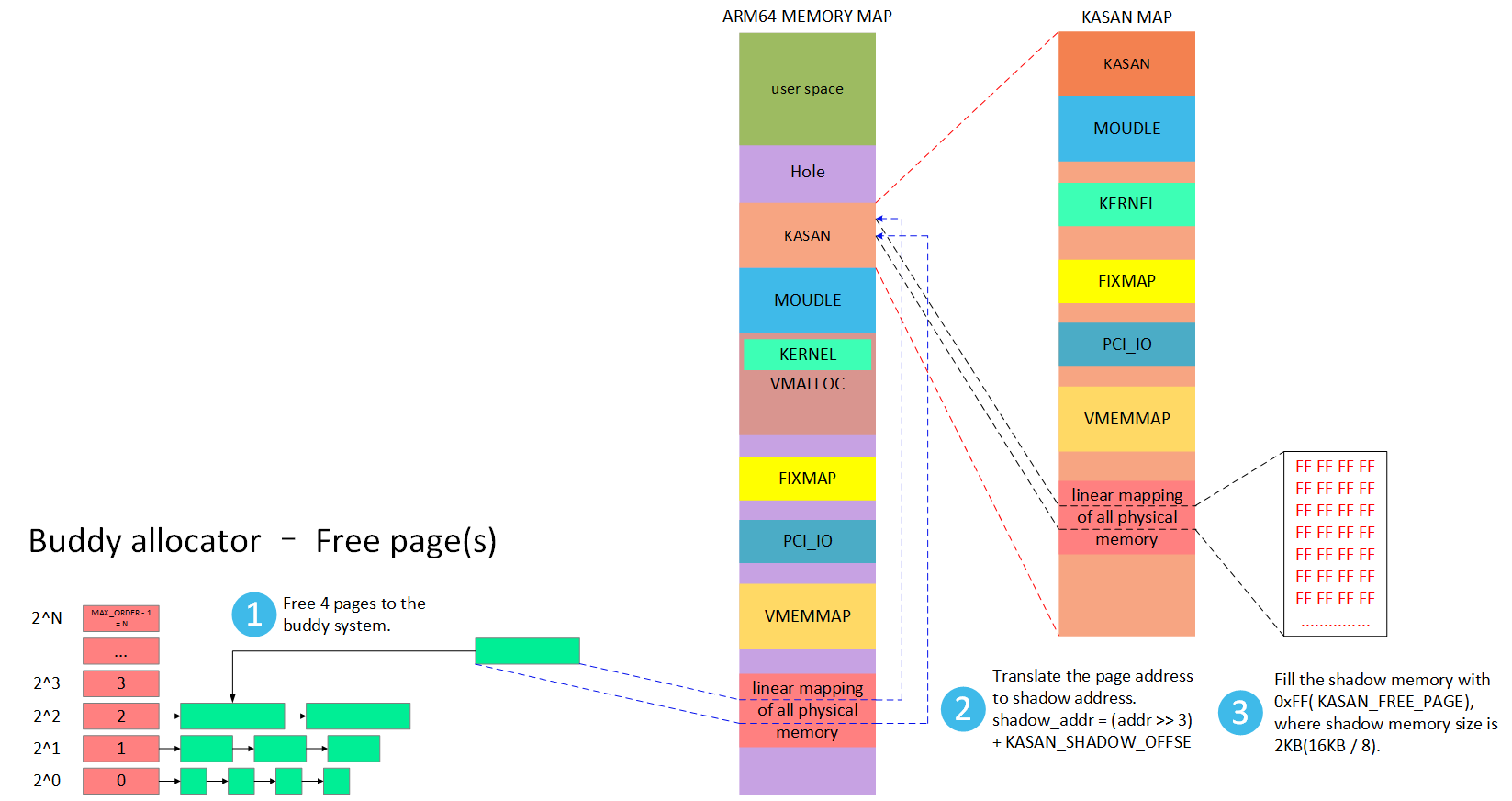
4.4. 伙伴系统分配的内存的shadow memory值如何填充？

既然shadow memory已经建立映射，接下来的事情就是探究各种内存分配器向shadow memory填充什么数据了。首先看一下伙伴系统allocate page(s)函数填充shadow memory情况。

[](http://www.wowotech.net/content/uploadfile/201802/82661518360143.png)

假设我们从buddy system分配4 pages。系统首先从order=2的链表中摘下一块内存，然后根据shadow memory address和memory address之间的对应的关系找对应的shadow memory。这里shadow memory的大小将会是2KB，系统会全部填充0代表内存可以访问。我们对分配的内存的任意地址内存进行访问的时候，首先都会找到对应的shadow memory，然后根据shadow memory value判断访问内存操作是否valid。

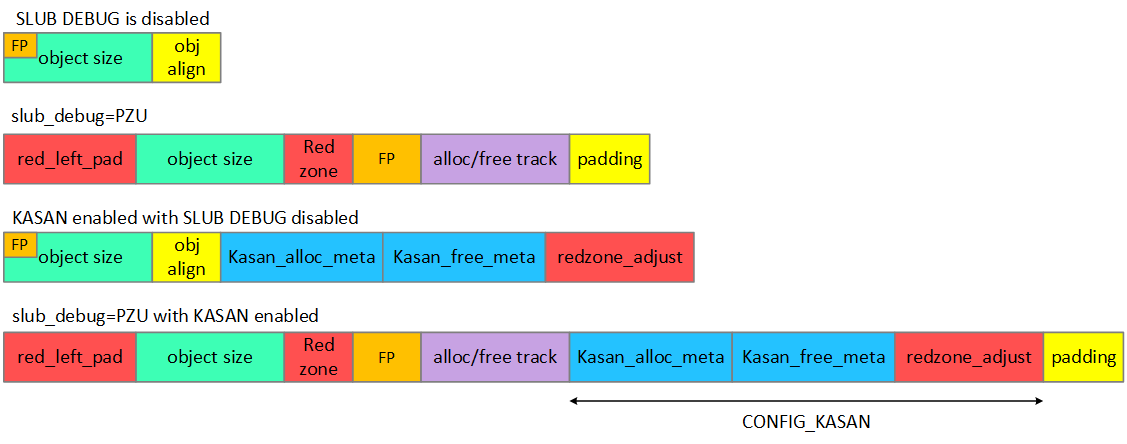
如果释放pages，情况又是如何呢？

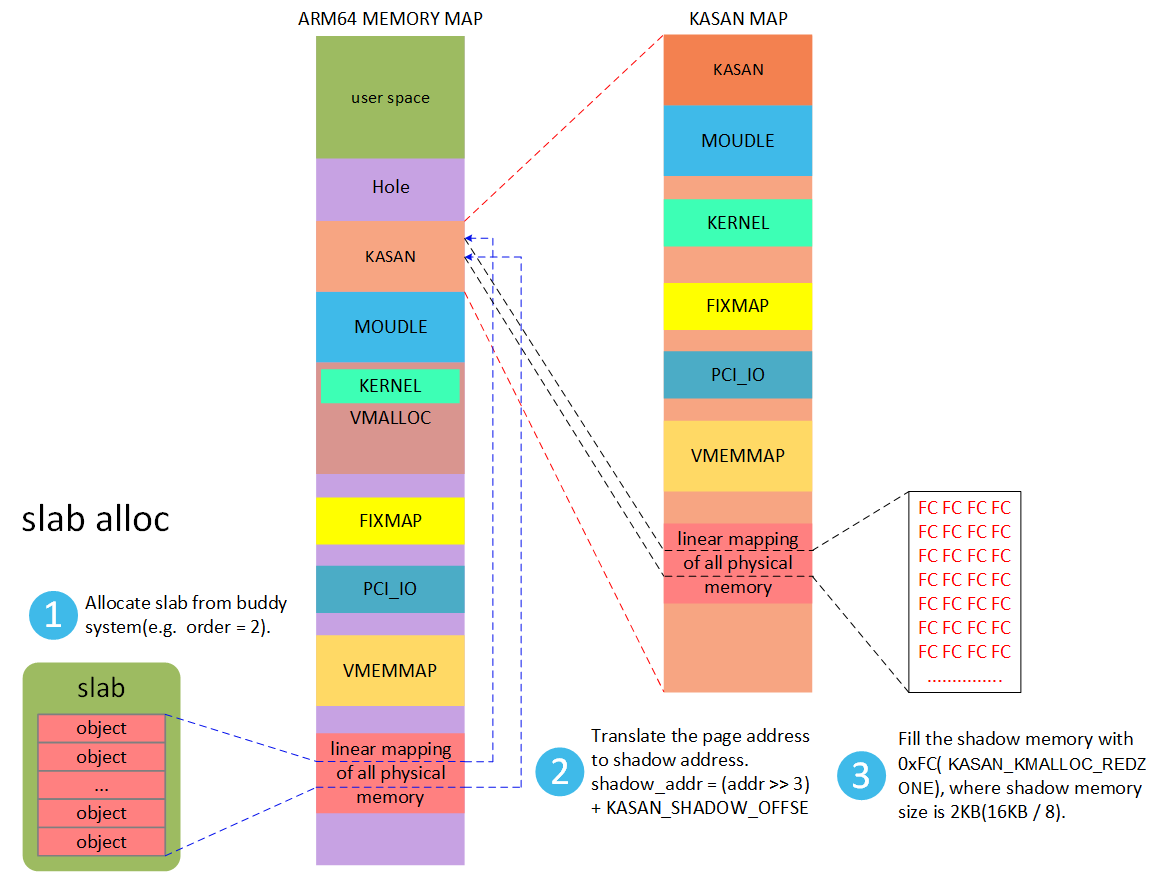
[](http://www.wowotech.net/content/uploadfile/201802/f19c1518360145.png)

同样的，当释放pages的时候，会填充shadow memory的值为0xFF。如果释放之后，依然访问内存的话，此时KASAN根据shadow memory的值是0xFF就可以断，这是一个use-after-free问题。

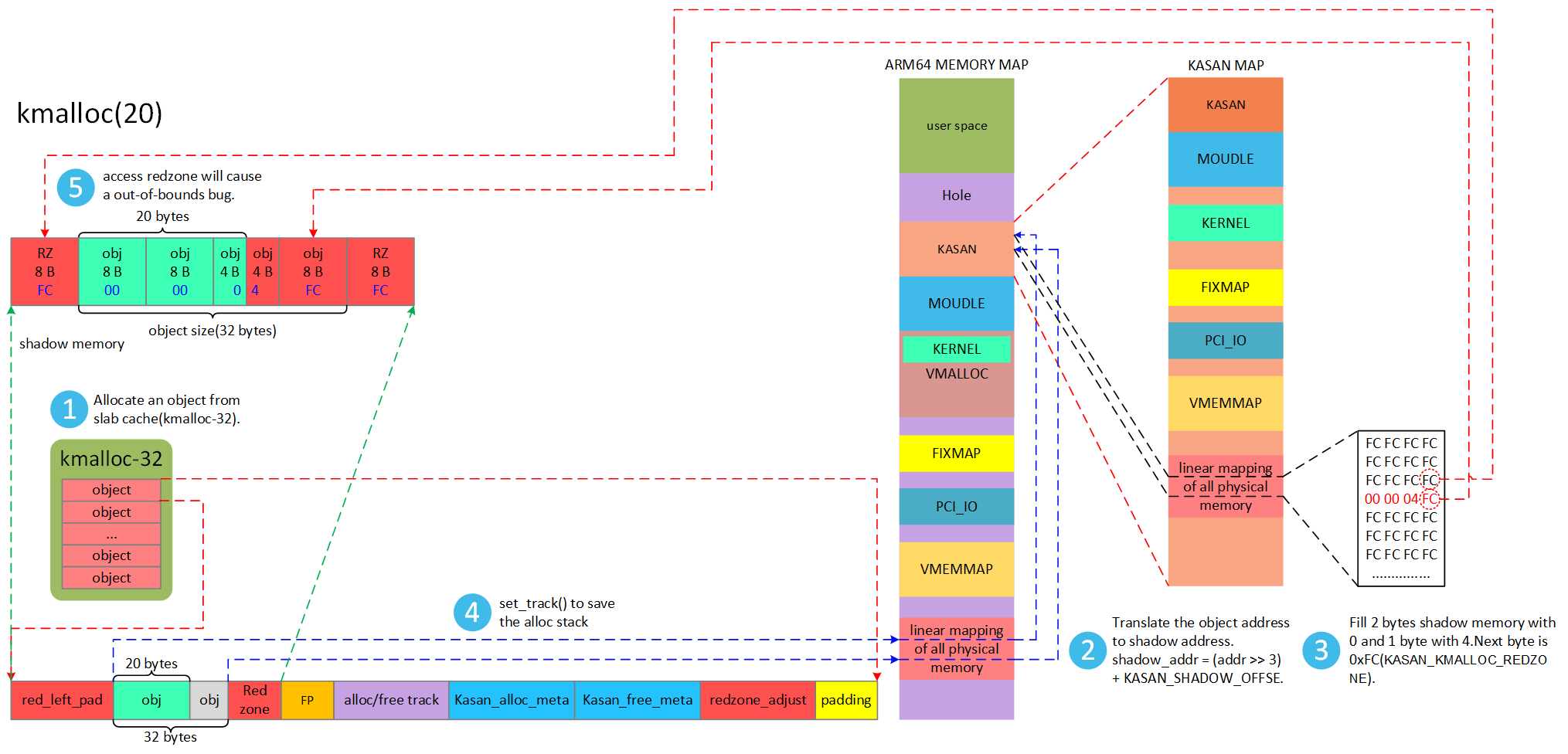
4.5. SLUB分配对象的内存的shadow memory值如何填充？

当我们打开KASAN的时候，SLUB Allocator管理的object layout将会放生一定的变化。如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/9eb91518360146.png)

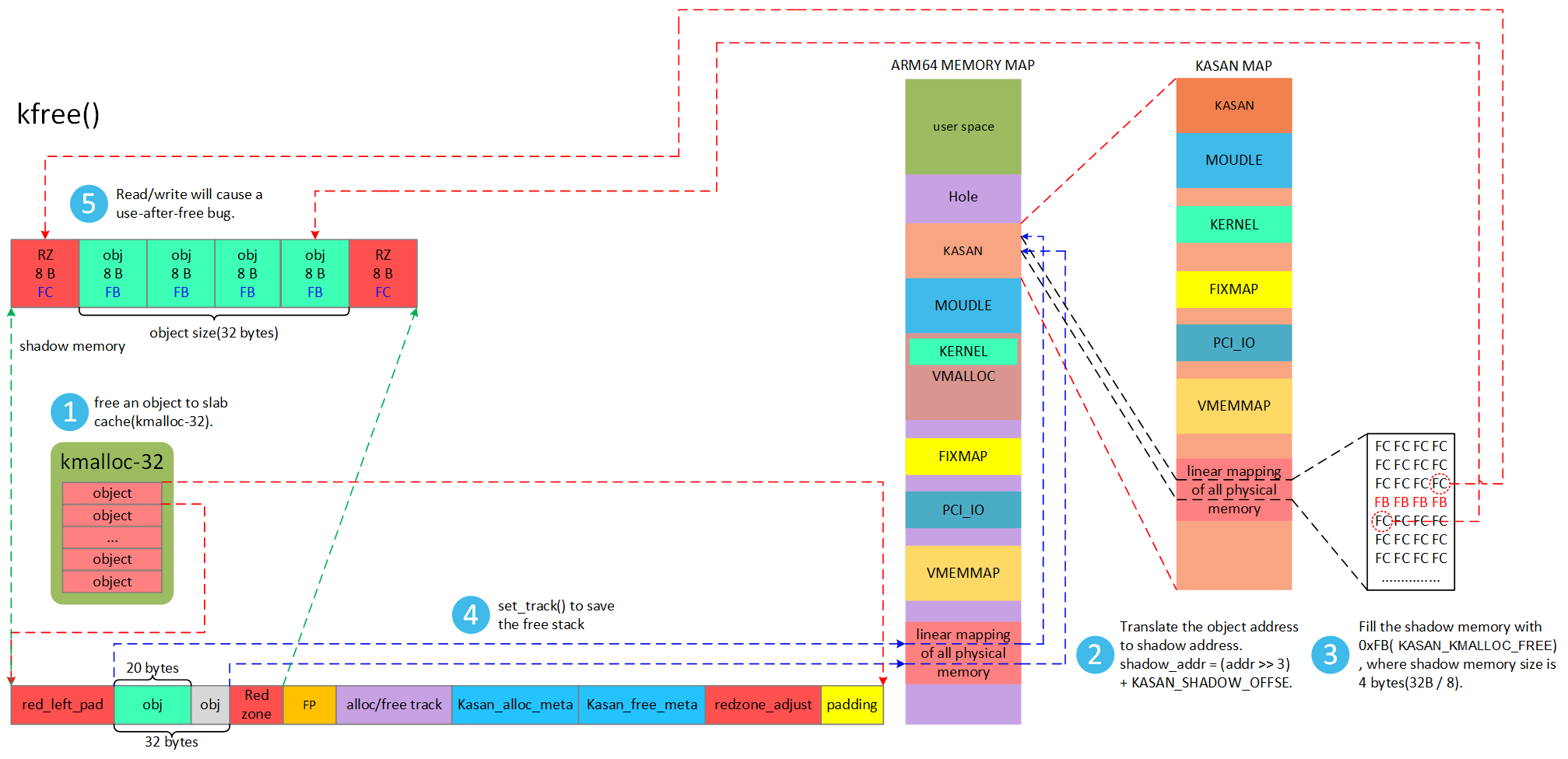
在打开SLUB\_DEBUG的时候，object就增加很多内存，KASAN打开之后，在此基础上又加了一截。为什么这里必须打开SLUB\_DEBUG呢？是因为有段时间KASAN是依赖SLUBU\_DEBUG的，什么意思呢？就是在Kconfig中使用了depends on，明白了吧。不过最新的代码已经不需要依赖了，可以看下提交。  
当我们第一次创建slab缓存池的时候，系统会调用kasan\_poison\_slab()函数初始化shadow memory为下图的模样。整个slab对应的shadow memory都填充0xFC。  
 [](http://www.wowotech.net/content/uploadfile/201802/602e1518360148.png)

上述步骤虽然填充了0xFC，但是接下来初始化object的时候，会改变一些shadow memory的值。我们先看一下kmalloc(20)的情况。我们知道kmalloc()就是基于SLUB Allocator实现的，所以会从kmalloc-32的kmem\_cache中分配一个32 bytes object。

[](http://www.wowotech.net/content/uploadfile/201802/7afb1518360149.png)

首先调用kmalloc(20)函数会匹配到kmalloc-32的kmem\_cache，因此实际分配的object大小是32 bytes。KASAN同样会标记剩下的12 bytes的shadow memory为不可访问状态。根据object的地址，计算shadow memory的地址，并开始填充数值。由于kmalloc()返回的object的size是32 bytes，由于kmalloc(20)只申请了20 bytes，剩下的12 bytes不能使用。KASAN必须标记shadow memory这种情况。object对应的4 bytes shadow memory分别填充00 00 04 FC。00代表8个连续的字节可以访问。04代表前4个字节可以访问。作为越界访问的检测的方法。总共加在一起是正好是20 bytes可访问。0xFC是Redzone标记。如果访问了Redzone区域KASAN就会检测out-of-bounds的发生。

当申请使用之后，现在调用kfree()释放之后的shadow memory情况是怎样的呢？看下图。

[](http://www.wowotech.net/content/uploadfile/201802/586e1518360151.png)

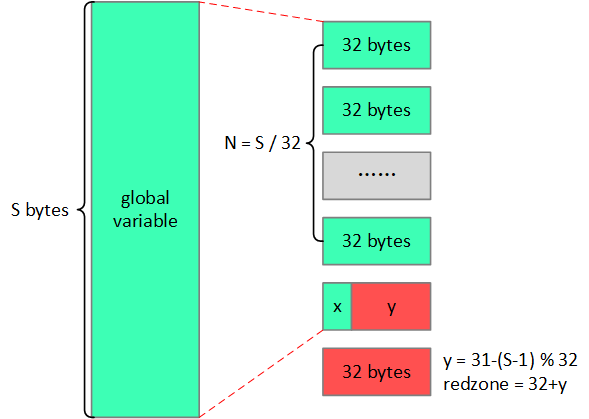
根据object首地址找到对应的shadow memory，32 bytes object对应4 bytes的shadow memory，现在填充0xFB标记内存是释放的状态。此时如果继续访问object，那么根据shadow memory的状态值既可以确定是use-after-free问题。

4.6. 全局变量的shadow memory值如何填充？

前面的分析都是基于内存分配器的，Redzone都会随着内存分配器一起分配。那么global variables如何检测呢？global variable的Redzone在哪里呢？这就需要编译器下手了。编译器会帮我们填充Redzone区域。例如我们定义一个全局变量a，编译器会帮我们填充成下面的样子。  
char a[4];  
转换

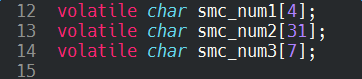
1. **struct** {
2. **char** original[4];
3. **char** redzone[60];
4. } a; //32 bytes aligned

如果这里你问我为什么填充60 bytes。其实我也不知道。这个转换例子也是从KASAN作者的PPT中拿过来的。估计要涉及编译器相关的知识，我无能为力了，但是下面做实验来猜吧。当然了，PPT的内容也需要验证才具有说服力。尽信书则不如无书。我特地写三个全局变量来验证。发现System.map分配地址之间的差值正好是0x40。因此这里的确是填充60 bytes。

另外从我的测试发现，如果上述的数组a的大小是33的时候，填充的redzone就是63 bytes。所以我推测，填充的原理是这样的。全局变量实际占用内存总数S（以byte为单位）按照每块32 bytes平均分成N块。假设最后一块内存距离目标32 bytes还差y bytes（if S%32 == 0，y = 0），那么redzone填充的大小就是(y + 32) bytes。画图示意如下（S%32 != 0）。因此总结的规律是：redzone = 63 – (S - 1) % 32。  
 [](http://www.wowotech.net/content/uploadfile/201802/59b21519483509.png)

全局变量redzone区域对应的shadow memory是在什么填充的呢？又是如何调用的呢？这部分是由编译器帮我们完成的。编译器会为每一个全局变量创建一个函数，函数名称是：\_GLOBAL\_\_sub\_I\_65535\_1\_##global\_variable\_name。这个函数中通过调用\_\_asan\_register\_globals()函数完成shadow memory标记。并且将自动生成的这个函数的首地址放在.init\_array段。在kernel启动阶段，通过以下代调用关系最终调用所有全局变量的构造函数。kernel\_init\_freeable()->do\_basic\_setup() ->do\_ctors()。do\_ctors()代码实现如下：

1. **static** **void** \_\_init do\_ctors(**void**)
2. {
3. ctor\_fn\_t \*fn = (ctor\_fn\_t \*) \_\_ctors\_start;
4. **for** (; fn < (ctor\_fn\_t \*) \_\_ctors\_end; fn++)
5. (\*fn)();
6. }

这里的代码意思对于轻车熟路的你再熟悉不过了吧。因为内核中这么搞的太多了。便利\_\_ctors\_start和\_\_ctors\_end之间的所有数据，作为函数地址进行调用，即完成了所有的global variables的shadow memory初始化。我们可以从链接脚本中知道\_\_ctors\_start和\_\_ctors\_end的意思。  
#define KERNEL\_CTORS()  . = ALIGN(8);              \  
            VMLINUX\_SYMBOL(\_\_ctors\_start) = .; \  
            KEEP(\*(.ctors))            \  
            KEEP(\*(SORT(.init\_array.\*)))       \  
            KEEP(\*(.init\_array))           \  
            VMLINUX\_SYMBOL(\_\_ctors\_end) = .;  
上面说了这么多，不知道你是否产生了疑心？怎么都是猜啊！猜的能准确吗？是的，我也这么觉得。是骡子是马，拉出来溜溜呗！现在用事实说话。首先我创建一个c文件drivers/input/smc.c。在smc.c文件中创建3个全局变量如下：  
 [](http://www.wowotech.net/content/uploadfile/201802/9eb61518360152.png)

然后就随便使用吧！编译kernel，我们先看看System.map文件中，3个全局变量分配的地址。  
ffff200009f540e0 B smc\_num1  
ffff200009f54120 B smc\_num2  
ffff200009f54160 B smc\_num3  
还记得上面说会有一个形如\_GLOBAL\_\_sub\_I\_65535\_1\_##global\_variable\_name的函数吗？在System.map文件文件中，我看到了\_GLOBAL\_\_sub\_I\_65535\_1\_smc\_num1符号。但是没有smc\_num2和smc\_num3的构造函数。你是不是很奇怪，不是每一个全局变量都会创建一个类似的构造函数吗？马上为你揭晓。我们先执行aarch64-linux-gnu-objdump –s –x –d vmlinux > vmlinux.txt命令得到反编译文件。现在好多重要的信息在vmlinux.txt。现在主要就是查看vmlinux.txt文件。先看一下\_GLOBAL\_\_sub\_I\_65535\_1\_smc\_num1函数的实现。

ffff200009381df0 <\_GLOBAL\_\_sub\_I\_65535\_1\_smc\_num1>:

ffff200009381df0:   a9bf7bfd    stp x29, x30, [sp,#-16]!  
ffff200009381df4:   b0001800    adrp    x0, ffff200009682000  
ffff200009381df8:   91308000    add x0, x0, #0xc20  
ffff200009381dfc:   d2800061    mov x1, #0x3                    // #3  
ffff200009381e00:   910003fd    mov x29, sp  
ffff200009381e04:   9100c000    add x0, x0, #0x30  
ffff200009381e08:   97c09fb8    bl  ffff2000083a9ce8 <\_\_asan\_register\_globals>  
ffff200009381e0c:   a8c17bfd    ldp x29, x30, [sp],#16  
ffff200009381e10:   d65f03c0    ret

汇编和C语言传递参数在ARM64平台使用的是x0~x7。通过上面的汇编计算一下，x0=0xffff200009682c50，x1=3。然后调用\_\_asan\_register\_globals()函数，x0和x1就是传递的参数。我们看一下\_\_asan\_register\_globals()函数实现。

1. **void** \_\_asan\_register\_globals(**struct** kasan\_global \*globals, size\_t size)
2. {
3. int i;
4. **for** (i = 0; i < size; i++)
5. register\_global(&globals[i]);
6. }

size是3就是要初始化全局变量的个数，所以这里只需要一个构造函数即可。一次性将3个全局变量全部搞定。这里再说一点猜测吧！我猜测是以文件为单位编译器创建一个构造函数即可，将本文件全局变量一次性全部打包初始化。第一个参数globals是0xffff200009682c50，继续从vmlinux.txt中查看该地址处的数据。struct kasan\_global是编译器帮我们自动创建的结构体，每一个全局变量对应一个struct kasan\_global结构体。struct kasan\_global结构体存放的位置是.data段，因此我们可以从.data段查找当前地址对应的数据。数据如下：  
ffff200009682c50 6041f509 0020ffff 07000000 00000000  
ffff200009682c60 40000000 00000000 d0d62b09 0020ffff  
ffff200009682c70 b8d62b09 0020ffff 00000000 00000000  
ffff200009682c80 202c6809 0020ffff 2041f509 0020ffff  
ffff200009682c90 1f000000 00000000 40000000 00000000  
ffff200009682ca0 e0d62b09 0020ffff b8d62b09 0020ffff  
ffff200009682cb0 00000000 00000000 302c6809 0020ffff  
ffff200009682cc0 e040f509 0020ffff 04000000 00000000  
ffff200009682cd0 40000000 00000000 f0d62b09 0020ffff  
ffff200009682ce0 b8d62b09 0020ffff 00000000 00000000

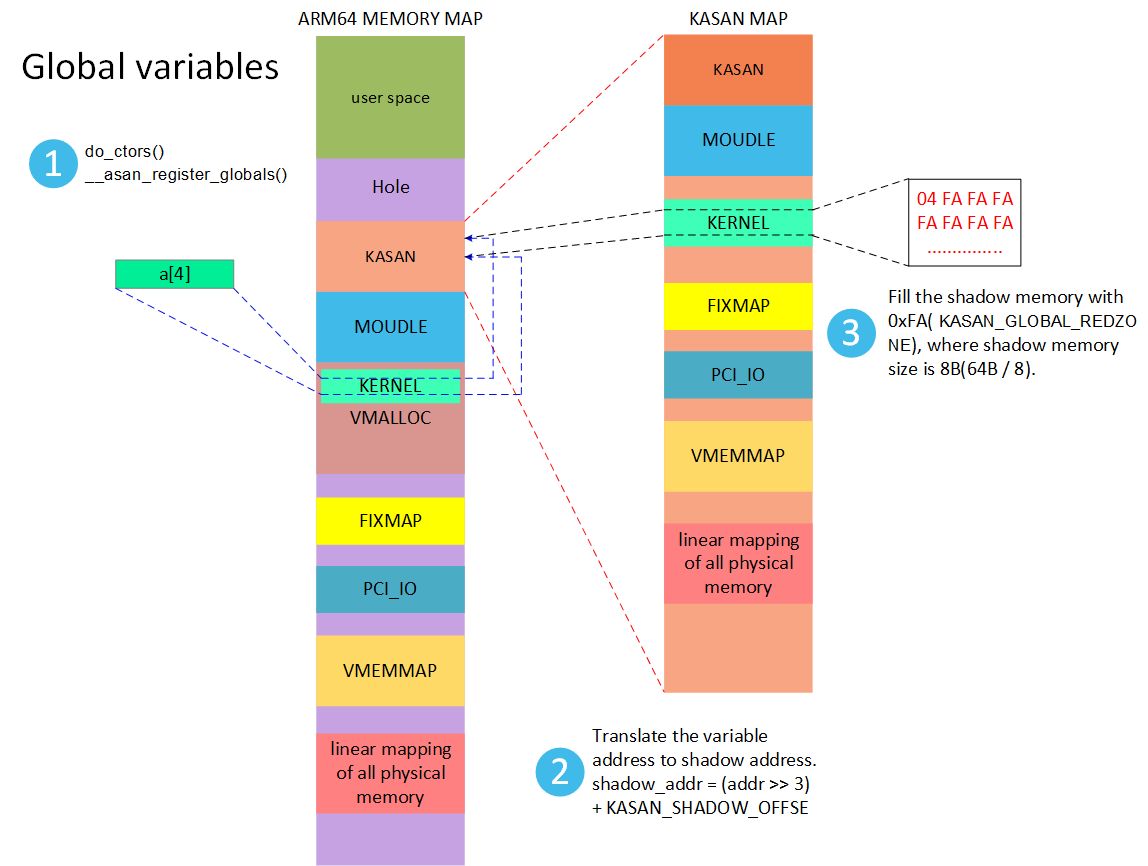
首先ffff200009682c50对应的第一个数据6041f509 0020ffff，这是个啥？其实是一个地址数据，你是不是又疑问了，ARM64的kernel space地址不是ffff开头吗？这个怎么60开头？其实这个地址数据是反过来的，你应该从右向左看。这个地址其实是ffff200009f54160。这不正是smc\_num3的地址嘛！解析这段数据之前需要了解一下struct kasan\_global结构体。

1. /\* The layout of struct dictated by compiler \*/
2. **struct** kasan\_global {
3. **const** **void** \*beg; /\* Address of the beginning of the global variable. \*/
4. size\_t size; /\* Size of the global variable. \*/
5. size\_t size\_with\_redzone; /\* Size of the variable + size of the red zone. 32 bytes aligned \*/
6. **const** **void** \*name;
7. **const** **void** \*module\_name; /\* Name of the module where the global variable is declared. \*/
8. **unsigned** **long** has\_dynamic\_init; /\* This needed for C++ \*/
9. #if KASAN\_ABI\_VERSION >= 4
10. **struct** kasan\_source\_location \*location;
11. #endif
12. };

第一个成员beg就是全局变量的首地址。跟上面的分析一致。第二个成员size从上面数据看出是7，正好对应我们定义的smc\_num3[7]，正好7 bytes。size\_with\_redzone的值是0x40，正好是64。根据上面猜测redzone=63-(7-1)%32=57。加上size正好是64，说明之前猜测的redzone计算方法没错。name成员对应的地址是ffff2000092bd6d0。看下ffff2000092bd6d0存储的是什么。

ffff2000092bd6d0 736d635f 6e756d33 00000000 00000000  smc\_num3........  
所以name就是全局变量的名称转换成字符串。同样的方式得到module\_name的地址是ffff2000092bd6b8。继续看看这段地址存储的数据。  
ffff2000092bd6b0 65000000 00000000 64726976 6572732f  e.......drivers/  
ffff2000092bd6c0 696e7075 742f736d 632e6300 00000000  input/smc.c.....  
一目了然，module\_name是文件的路径。has\_dynamic\_init的值就是0，这是C++需要的。我用的GCC版本是5.0左右，所以这里的KASAN\_ABI\_VERSION=4。这里location成员的地址是ffff200009682c20，继续追踪该地址的数据。  
ffff200009682c20 b8d62b09 0020ffff 0e000000 0f000000  
解析这段数据之前要先了解struct kasan\_source\_location结构体。

1. /\* The layout of struct dictated by compiler \*/
2. **struct** kasan\_source\_location {
3. **const** **char** \*filename;
4. int line\_no;
5. int column\_no;
6. };

第一个成员filename地址是ffff2000092bd6b8和module\_name一样的数据。剩下两个数据分别是14和15，分别代表全局变量定义地方的行号和列号。现在回到上面我定义变量的截图，仔细数数列号是不是15，行号截图中也有哦！特地截出来给你看的。剩下的struct kasan\_global数据就是smc\_num1和smc\_num2的数据。分析就不说了。前面说\_GLOBAL\_\_sub\_I\_65535\_1\_smc\_num1函数会被自动调用，该地址数据填充在\_\_ctors\_start和\_\_ctors\_end之间。现在也证明一下观点。先从System.map得到符号的地址数据。  
ffff2000093ac5d8 T \_\_ctors\_start  
ffff2000093ae860 T \_\_ctors\_end  
然后搜索一下\_GLOBAL\_\_sub\_I\_65535\_1\_smc\_num1的地址ffff200009381df0被存储在什么位置，记得搜索的关键字是f01d3809 0020ffff。  
ffff2000093ae0c0 f01d3809 0020ffff 181e3809 0020ffff  
可以看出ffff2000093ae0c0地址处存储着\_GLOBAL\_\_sub\_I\_65535\_1\_smc\_num1函数地址。这个地址不是正好位于\_\_ctors\_start和\_\_ctors\_end之间嘛！  
现在就剩下\_\_asan\_register\_globals()函数到底是是怎么初始化shadow memory的呢？以char a[4]为例，如下图所示。  
 [](http://www.wowotech.net/content/uploadfile/201802/c00b1519483390.png)

a[4]只有4 bytes可以访问，所以对应的shadow memory的第一个byte值是4，后面的redzone就填充0xFA作为越界检测。a[4]只有4 bytes可以访问，所以对应的shadow memory的第一个byte值是4，后面的redzone就填充0xFA作为越界检测。因为这里是全局变量，因此分配的内存区域位于kernel区域。

4.7. 栈分配变量的readzone是如何分配的？

从栈中分配的变量同样和全局变量一样需要填充一些内存作为redzone区域。下面继续举个例子说明编译器怎么填充。首先来一段正常的代码，没有编译器的插手。

1. **void** foo()
2. {
3. **char** a[328];
4. }

再来看看编译器插了哪些东西进去。

void foo()  
{  
    char rz1[32];  
    char a[328];  
    char rz2[56];  
    int \*shadow = （&rz1 >> 3）+ KASAN\_SHADOW\_OFFSE;  
    shadow[0] = 0xffffffff;  
    shadow[11] = 0xffffff00;  
    shadow[12] = 0xffffffff;  
⇓------------------------使用完毕----------------------------------------◊  
    shadow[0] = shadow[11] = shadow[12] = 0;  
}

红色部分是编译器填充内存，rz2是56，可以根据上一节全局变量的公式套用计算得到。但是这里在变量前面竟然还有32 bytes的rz1。这个是和全局变量的不同，我猜测这里是为了检测栈变量左边界越界问题。蓝色部分代码也是编译器填充，初始化shadow memory。栈的填充就没有探究那么深入了，有兴趣的读者可以自己探究。

5. Error log信息包含哪些信息？

从kernel的Documentation文档找份典型的KASAN bug输出的log信息如下。  
==================================================================  
BUG: AddressSanitizer: out of bounds access in kmalloc\_oob\_right+0x65/0x75 [test\_kasan] at addr ffff8800693bc5d3  
Write of size 1 by task modprobe/1689  
=============================================================================  
BUG kmalloc-128 (Not tainted): kasan error  
-----------------------------------------------------------------------------

Disabling lock debugging due to kernel taint  
INFO: Allocated in kmalloc\_oob\_right+0x3d/0x75 [test\_kasan] age=0 cpu=0 pid=1689  
 \_\_slab\_alloc+0x4b4/0x4f0  
 kmem\_cache\_alloc\_trace+0x10b/0x190  
 kmalloc\_oob\_right+0x3d/0x75 [test\_kasan]  
 init\_module+0x9/0x47 [test\_kasan]  
 do\_one\_initcall+0x99/0x200  
 load\_module+0x2cb3/0x3b20  
 SyS\_finit\_module+0x76/0x80  
 system\_call\_fastpath+0x12/0x17  
INFO: Slab 0xffffea0001a4ef00 objects=17 used=7 fp=0xffff8800693bd728 flags=0x100000000004080  
INFO: Object 0xffff8800693bc558 @offset=1368 fp=0xffff8800693bc720

Bytes b4 ffff8800693bc548: 00 00 00 00 00 00 00 00 5a 5a 5a 5a 5a 5a 5a 5a  ........ZZZZZZZZ  
Object ffff8800693bc558: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk  
Object ffff8800693bc568: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk  
Object ffff8800693bc578: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk  
Object ffff8800693bc588: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk  
Object ffff8800693bc598: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk  
Object ffff8800693bc5a8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk  
Object ffff8800693bc5b8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkkkkkkkkk  
Object ffff8800693bc5c8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b a5  kkkkkkkkkkkkkkk.  
Redzone ffff8800693bc5d8: cc cc cc cc cc cc cc cc                          ........  
Padding ffff8800693bc718: 5a 5a 5a 5a 5a 5a 5a 5a                          ZZZZZZZZ  
CPU: 0 PID: 1689 Comm: modprobe Tainted: G    B          3.18.0-rc1-mm1+ #98  
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS rel-1.7.5-0-ge51488c-20140602\_164612-nilsson.home.kraxel.org 04/01/2014  
 ffff8800693bc000 0000000000000000 ffff8800693bc558 ffff88006923bb78  
 ffffffff81cc68ae 00000000000000f3 ffff88006d407600 ffff88006923bba8  
 ffffffff811fd848 ffff88006d407600 ffffea0001a4ef00 ffff8800693bc558  
Call Trace:  
 [<ffffffff81cc68ae>] dump\_stack+0x46/0x58  
 [<ffffffff811fd848>] print\_trailer+0xf8/0x160  
 [<ffffffffa00026a7>] ? kmem\_cache\_oob+0xc3/0xc3 [test\_kasan]  
 [<ffffffff811ff0f5>] object\_err+0x35/0x40  
 [<ffffffffa0002065>] ? kmalloc\_oob\_right+0x65/0x75 [test\_kasan]  
 [<ffffffff8120b9fa>] kasan\_report\_error+0x38a/0x3f0  
 [<ffffffff8120a79f>] ? kasan\_poison\_shadow+0x2f/0x40  
 [<ffffffff8120b344>] ? kasan\_unpoison\_shadow+0x14/0x40  
 [<ffffffff8120a79f>] ? kasan\_poison\_shadow+0x2f/0x40  
 [<ffffffffa00026a7>] ? kmem\_cache\_oob+0xc3/0xc3 [test\_kasan]  
 [<ffffffff8120a995>] \_\_asan\_store1+0x75/0xb0  
 [<ffffffffa0002601>] ? kmem\_cache\_oob+0x1d/0xc3 [test\_kasan]  
 [<ffffffffa0002065>] ? kmalloc\_oob\_right+0x65/0x75 [test\_kasan]  
 [<ffffffffa0002065>] kmalloc\_oob\_right+0x65/0x75 [test\_kasan]  
 [<ffffffffa00026b0>] init\_module+0x9/0x47 [test\_kasan]  
 [<ffffffff810002d9>] do\_one\_initcall+0x99/0x200  
 [<ffffffff811e4e5c>] ? \_\_vunmap+0xec/0x160  
 [<ffffffff81114f63>] load\_module+0x2cb3/0x3b20  
 [<ffffffff8110fd70>] ? m\_show+0x240/0x240  
 [<ffffffff81115f06>] SyS\_finit\_module+0x76/0x80  
 [<ffffffff81cd3129>] system\_call\_fastpath+0x12/0x17  
Memory state around the buggy address:  
 ffff8800693bc300: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc  
 ffff8800693bc380: fc fc 00 00 00 00 00 00 00 00 00 00 00 00 00 fc  
 ffff8800693bc400: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc  
 ffff8800693bc480: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc  
 ffff8800693bc500: fc fc fc fc fc fc fc fc fc fc fc 00 00 00 00 00  
>ffff8800693bc580: 00 00 00 00 00 00 00 00 00 00 03 fc fc fc fc fc  
                                                                           ^  
 ffff8800693bc600: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc  
 ffff8800693bc680: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc  
 ffff8800693bc700: fc fc fc fc fb fb fb fb fb fb fb fb fb fb fb fb  
 ffff8800693bc780: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb  
 ffff8800693bc800: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb  
==================================================================  
输出的信息很丰富，包含了bug发生的类型、SLUB输出的object内存信息、Call Trace以及shadow memory的状态值。其中红色信息都是比较重要的信息。我没有写demo历程，而是找了一份log信息，不是我想偷懒，而是锻炼自己。怎么锻炼呢？我想问的是，从这份log中你可以推测代码应该是怎么样的？我可以得到一下信息：  
1) 程序是通过kmalloc接口申请内存的；  
2) 申请的内存大小是123 bytes，即p = kamlloc(123);  
3) 代码中类似往p[123]中写1 bytes导致越界访问的bug；  
4) 在3)步骤发生前没有任何的对该内存的写操作；  
如果你也能得到以上4点猜测，我觉的我写的这几篇文章你是真的看明白了。首先输出信息是有SLUB的信息，所以应该是通过kmalloc()接口；在打印的shadow memory的值中，我们看到连续的15个0和一个3，所以申请的内存size就是15x8+3=123；由于是往ffff8800693bc5d3地址写1个字节，并且object首地址是ffff8800693bc558，所以推测是往p[123]写1 byte出问题；由于log中将object中所有的128 bytes数据全部打印出来，一共是127个0x6b和一个0xa5（SLUB DEBUG文章介绍的内容）。所以我推测在3)步骤发生前没有任何的对该内存的写操作。

6. 补充

我看了linux-4.18的代码，KASAN的log输出已经发生了部分变化。例如：上面举例的SLUB的object的内容就不会打印了。我们用一下的程序展示这些变化（实际上就是上面举例用的程序）。

1. **static** noinline **void** \_\_init kmalloc\_oob\_right(**void**)
2. {
3. **char** \*ptr;
4. size\_t size = 123;
6. ptr = kmalloc(size, GFP\_KERNEL);
7. **if** (!ptr) {
8. pr\_err("Allocation failed\n");
9. **return**;
10. }
12. ptr[size] = 'x';
13. kfree(ptr);
14. }

针对以上代码，KASAN检测到bug后的输出log如下：

1. ==================================================================
2. BUG: KASAN: slab-**out**-of-bounds **in** kmalloc\_oob\_right+0x6c/0x8c
3. Write of size 1 at addr ffffffc0cb114d7b **by** task swapper/0/1
5. CPU: 4 PID: 1 Comm: swapper/0 Tainted: G S W 4.9.82-perf+ #310
6. Hardware name: Qualcomm Technologies, Inc. SDM632 PMI632
7. Call trace:
8. [<ffffff90cf88d9f8>] dump\_backtrace+0x0/0x320
9. [<ffffff90cf88dd2c>] show\_stack+0x14/0x20
10. [<ffffff90cfdd1148>] dump\_stack+0xa8/0xd0
11. [<ffffff90cfabf298>] print\_address\_description+0x60/0x250
12. [<ffffff90cfabf6a0>] kasan\_report.part.2+0x218/0x2f0
13. [<ffffff90cfabfac0>] kasan\_report+0x20/0x28
14. [<ffffff90cfabdc64>] \_\_asan\_store1+0x4c/0x58
15. [<ffffff90d1a4f760>] kmalloc\_oob\_right+0x6c/0x8c
16. [<ffffff90d1a50448>] kmalloc\_tests\_init+0xc/0x68
17. [<ffffff90cf8845dc>] do\_one\_initcall+0xa4/0x1f0
18. [<ffffff90d1a011ac>] kernel\_init\_freeable+0x244/0x300
19. [<ffffff90d0d6da70>] kernel\_init+0x10/0x110
20. [<ffffff90cf8842a0>] ret\_from\_fork+0x10/0x30
22. Allocated **by** task 1:
23. kasan\_kmalloc+0xd8/0x188
24. kmem\_cache\_alloc\_trace+0x130/0x248
25. kmalloc\_oob\_right+0x4c/0x8c
26. kmalloc\_tests\_init+0xc/0x68
27. do\_one\_initcall+0xa4/0x1f0
28. kernel\_init\_freeable+0x244/0x300
29. kernel\_init+0x10/0x110
30. ret\_from\_fork+0x10/0x30
32. Freed **by** task 1:
33. kasan\_slab\_free+0x88/0x178
34. kfree+0x84/0x298
35. kobject\_uevent\_env+0x144/0x620
36. kobject\_uevent+0x10/0x18
37. device\_add+0x5f8/0x860
38. amba\_device\_try\_add+0x22c/0x2f8
39. amba\_device\_add+0x20/0x128
40. of\_platform\_bus\_create+0x390/0x478
41. of\_platform\_bus\_create+0x21c/0x478
42. of\_platform\_populate+0x4c/0xb8
43. of\_platform\_default\_populate\_init+0x78/0x8c
44. do\_one\_initcall+0xa4/0x1f0
45. kernel\_init\_freeable+0x244/0x300
46. kernel\_init+0x10/0x110
47. ret\_from\_fork+0x10/0x30
49. The buggy address belongs to the **object** at ffffffc0cb114d00
50. which belongs to the cache kmalloc-128 of size 128
51. The buggy address **is** located 123 bytes inside of
52. 128-**byte** region [ffffffc0cb114d00, ffffffc0cb114d80)
53. The buggy address belongs to the page:
54. page:ffffffbf032c4500 count:1 mapcount:0 mapping: (**null**) index:0xffffffc0cb115200 compound\_mapcount: 0
55. flags: 0x4080(slab|head)
56. page dumped because: kasan: bad access detected
58. Memory state around the buggy address:
59. ffffffc0cb114c00: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
60. ffffffc0cb114c80: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
61. >ffffffc0cb114d00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03
62. ^
63. ffffffc0cb114d80: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
64. ffffffc0cb114e00: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
65. ==================================================================

我们从上面的log可以分析如下数据：

* line2：发生越界访问位置。
* line3：越界写1个字节，写的地址是0xffffffc0cb114d7b。当前进程是comm是swapper/0，pid是1。
* line7：Call trace，方便定位出问题的函数调用关系。
* line22：该object分配的调用栈，并指出分配内存的进程pid是1。
* line32：释放该object的调用栈（上次释放），并指出释放内存的进程pid是1。
* line49：指出slub相关的信息，从“kmalloc-28”的kmem\_cache分配的object。object起始地址是0xffffffc0cb114d00。
* line51：访问出问题的地址位于object起始地址偏移123 bytes的位置。object的地址范围是[0xffffffc0cb114d00, 0xffffffc0cb114d80)。object实际大小是128 bytes。
* line61：出问题地址对应的shadow memory的值，可以确定申请内存的实际大小是123 bytes。

参考文献：

1．How to use KASAN to debug memory corruption in OpenStack environment.pdf

2．KernelAddressSanitizer (KASan) a fast memory error detector for the Linux kernel.pdf

# fixmap addresses原理

作者：[smcdef](http://www.wowotech.net/author/531) 发布于：2018-4-29 20:35 分类：[内存管理](http://www.wowotech.net/sort/memory_management)

## 引言

fixmap是一段固定地址映射。kernel预留一段虚拟地址空间。因此虚拟地址是在编译的时候确定。fixmap可以用来做什么？kernel启动初期，由于此时的kernel已经运行在虚拟地址上。因此我们访问具体的物理地址是不行的，必须建立虚拟地址和物理地址的映射，然后通过虚拟地址访问才可以。例如：dtb中包含bootloader传递过来的内存信息，我们需要解析dtb，但是我们得到的是dtb的物理地址。因此访问之前必须创建映射，创建映射又需要内存。但是由于所有的内存管理子系统还没有ready。因此我们不能使用ioremap接口创建映射。为此kernel提出fixmap的解决方案。

注：文章代码分析基于linux-4.15，架构基于aarch64（ARM64）。

## fixmap空间分配

fixmap虚拟地址空间又被平均分成两个部分permanent fixed addresses和temporary fixed addresses。permanent fixed addresses是永久映射，temporary fixed addresses是临时映射。永久映射是指在建立的映射关系在kernel阶段不会改变，仅供特定模块一直使用。临时映射就是模块使用前创建映射，使用后解除映射。

fixmap区域又被继续细分，分配给不同模块使用。kernel中定义枚举类型作为index，根据index可以计算该模拟在fixmap区域的虚拟地址。

1. **enum** fixed\_addresses {
2. FIX\_HOLE,
3. #define FIX\_FDT\_SIZE (MAX\_FDT\_SIZE + SZ\_2M)
4. FIX\_FDT\_END,
5. FIX\_FDT = FIX\_FDT\_END + FIX\_FDT\_SIZE / PAGE\_SIZE - 1, /\* 1 \*/
6. FIX\_EARLYCON\_MEM\_BASE, /\* 2 \*/
7. FIX\_TEXT\_POKE0,
8. \_\_end\_of\_permanent\_fixed\_addresses,
9. #define NR\_FIX\_BTMAPS (SZ\_256K / PAGE\_SIZE)
10. #define FIX\_BTMAPS\_SLOTS 7
11. #define TOTAL\_FIX\_BTMAPS (NR\_FIX\_BTMAPS \* FIX\_BTMAPS\_SLOTS)
12. FIX\_BTMAP\_END = \_\_end\_of\_permanent\_fixed\_addresses,
13. FIX\_BTMAP\_BEGIN = FIX\_BTMAP\_END + TOTAL\_FIX\_BTMAPS - 1, /\* 3 \*/
14. FIX\_PTE,
15. FIX\_PMD,
16. FIX\_PUD,
17. FIX\_PGD,
18. \_\_end\_of\_fixed\_addresses
19. };
20. #define FIXADDR\_SIZE (\_\_end\_of\_permanent\_fixed\_addresses << PAGE\_SHIFT)
21. #define FIXADDR\_START (FIXADDR\_TOP - FIXADDR\_SIZE)

* FIX\_FDT映射设备树使用。在ARM64架构，大小是4M。
* early console使用，大小1页。1页虚拟地址空间完全够了，毕竟串口操作相关寄存器没有几个。
* early\_ioremap()接口使用，这部分属于动态映射。大小是7 × 256KB。

fixmap区域是地址空间范围从FIXADDR\_START到FIXADDR\_TOP结束。FIXADDR\_SIZE是permanent fixed addresses区域的大小。我对这个地方很奇怪，为什么不包括temporary fixed addresses区域的大小呢？如果你知道可以告诉我。fixmap区域可以想象成一块内存以页为单位被平均分成\_\_end\_of\_permanent\_fixed\_addresses块。而这些枚举值就是这块内存的index。因此虚拟地址可以根据index进行计算。计算方法如下。

1. #define \_\_fix\_to\_virt(x) (FIXADDR\_TOP - ((x) << PAGE\_SHIFT))
2. #define \_\_virt\_to\_fix(x) ((FIXADDR\_TOP - ((x) & PAGE\_MASK)) >> PAGE\_SHIFT)

FIX\_FDT是给dtb创建映射的区域。例如需要得到FDT的虚拟地址，即可以利用\_\_fix\_to\_virt(FIX\_FDT)得到虚拟地址。之所以FIX\_FDT放在枚举的最前面，是因为我们针对dtb映射采用section mapping要求虚拟地址2M对齐。FIXADDR\_TOP地址本身是2M对齐的，因此FIXADDR\_TOP - (FIX\_FDT << PAGE\_SHIFT)可以很容易2M对齐。

## fixmap初始化

fixmap初始化操作在early\_fixmap\_init函数中完成。主要是建立PGD/PUD/PMD页表。early\_fixmap\_init实现如下（简化部分代码逻辑）。

1. **static** pte\_t bm\_pte[PTRS\_PER\_PTE] \_\_page\_aligned\_bss; /\* 1 \*/
2. **static** pmd\_t bm\_pmd[PTRS\_PER\_PMD] \_\_page\_aligned\_bss \_\_maybe\_unused;/\* 1 \*/
3. **static** pud\_t bm\_pud[PTRS\_PER\_PUD] \_\_page\_aligned\_bss \_\_maybe\_unused;/\* 1 \*/
5. **void** \_\_init early\_fixmap\_init(**void**)
6. {
7. pgd\_t \*pgd;
8. pud\_t \*pud;
9. pmd\_t \*pmd;
10. **unsigned** **long** addr = FIXADDR\_START; /\* 2 \*/
12. pgd = pgd\_offset\_k(addr);
13. **if** (pgd\_none(\*pgd))
14. \_\_pgd\_populate(pgd, \_\_pa\_symbol(bm\_pud), PUD\_TYPE\_TABLE); /\* 2 \*/
15. pud = fixmap\_pud(addr);
16. **if** (pud\_none(\*pud))
17. \_\_pud\_populate(pud, \_\_pa\_symbol(bm\_pmd), PMD\_TYPE\_TABLE); /\* 2 \*/
18. pmd = fixmap\_pmd(addr);
19. \_\_pmd\_populate(pmd, \_\_pa\_symbol(bm\_pte), PMD\_TYPE\_TABLE); /\* 2 \*/
21. BUILD\_BUG\_ON((\_\_fix\_to\_virt(FIX\_BTMAP\_BEGIN) >> PMD\_SHIFT)
22. != (\_\_fix\_to\_virt(FIX\_BTMAP\_END) >> PMD\_SHIFT)); /\* 3 \*/
24. **if** ((pmd != fixmap\_pmd(fix\_to\_virt(FIX\_BTMAP\_BEGIN)))
25. || pmd != fixmap\_pmd(fix\_to\_virt(FIX\_BTMAP\_END))) { /\* 3 \*/
26. WARN\_ON(1);
27. }
28. }
29. 静态定义数组作为页表（PUD/PMD/PTE）使用。PGD页表使用swapper\_pg\_dir。
30. 以FIXADDR\_START为虚拟地址，建立页表映射关系。假设计算FDT的虚拟地址为addr = \_\_fix\_to\_virt(FIX\_FDT)，必然addr是2M对齐的一个地址。由上面的分析可知，FIXADDR\_START的值位于[addr - 2M, addr]之间。因此访问[addr - 2M, addr]之间的虚拟地址不需要再建立PUD/PMD页表项。只需要设置PTE页表对应的页表项即可。
31. [FIX\_BTMAP\_BEGIN, FIX\_BTMAP\_END]区域给动态映射使用，保证该区域正好位于[addr - 2M, addr]之间，必须检查动态映射区域小于2M。定义的PTE页表数组其实是给动态映射使用的。当我们需要访问物理地址A，从[addr - 2M, addr]区域找到一个合适的虚拟地址B，填充B地址对应的PTE页表项即可访问。这也是early\_ioremap的实现原理。

经过early\_fixmap\_init函数的探究，我们也可以得到一个结论：为了以page为单位进行映射，必须保证FIX\_FDT和\_\_end\_of\_fixed\_addresses之间的虚拟地址空间必须小于2M。如果有超过2M的部分就要使用section mapping（因为只有一个PTE页表），以2M为单位映射。

## early ioremap初始化

如果你希望kernel启动早期使用ioremap操作，其实是不行的。我们必须借助early ioremap接口。early ioremap是基于fixmap实现。初始化在early\_ioremap\_init完成。简化部分代码如下。

1. **static** **void** \_\_iomem \*prev\_map[FIX\_BTMAPS\_SLOTS] \_\_initdata;
2. **static** **unsigned** **long** prev\_size[FIX\_BTMAPS\_SLOTS] \_\_initdata;
3. **static** **unsigned** **long** slot\_virt[FIX\_BTMAPS\_SLOTS] \_\_initdata;
5. **void** \_\_init early\_ioremap\_setup(**void**)
6. {
7. int i;
9. **for** (i = 0; i < FIX\_BTMAPS\_SLOTS; i++)
10. slot\_virt[i] = \_\_fix\_to\_virt(FIX\_BTMAP\_BEGIN - NR\_FIX\_BTMAPS\*i);
11. }
13. **void** \_\_init early\_ioremap\_init(**void**)
14. {
15. early\_ioremap\_setup();
16. }

early ioremap利用slot管理映射，最多支持FIX\_BTMAPS\_SLOTS个映射，每个映射最大支持映射256KB。slot\_virt数组存储每个slot的虚拟地址首地址。prev\_map数组用来记录已经分配出去的虚拟地址，数组值为0代表没有分配。prev\_size记录映射的size。

## 创建FDT映射

创建FDT映射的函数是\_\_fixmap\_remap\_fdt，实现如下。

1. **void** \*\_\_init \_\_fixmap\_remap\_fdt(phys\_addr\_t dt\_phys,
2. int \*size, pgprot\_t prot)
3. {
4. **const** u64 dt\_virt\_base = \_\_fix\_to\_virt(FIX\_FDT); /\* 1 \*/
5. int offset;
6. **void** \*dt\_virt;
8. BUILD\_BUG\_ON(dt\_virt\_base % SZ\_2M); /\* 2 \*/
10. BUILD\_BUG\_ON(\_\_fix\_to\_virt(FIX\_FDT\_END) >> SWAPPER\_TABLE\_SHIFT !=
11. \_\_fix\_to\_virt(FIX\_BTMAP\_BEGIN) >> SWAPPER\_TABLE\_SHIFT); /\* 3 \*/
13. offset = dt\_phys % SWAPPER\_BLOCK\_SIZE;
14. dt\_virt = (**void** \*)dt\_virt\_base + offset;
16. create\_mapping\_noalloc(round\_down(dt\_phys, SWAPPER\_BLOCK\_SIZE), /\* 4 \*/
17. dt\_virt\_base, SWAPPER\_BLOCK\_SIZE, prot);
19. \*size = fdt\_totalsize(dt\_virt);
20. **if** (offset + \*size > SWAPPER\_BLOCK\_SIZE)
21. create\_mapping\_noalloc(round\_down(dt\_phys, SWAPPER\_BLOCK\_SIZE), dt\_virt\_base,
22. round\_up(offset + \*size, SWAPPER\_BLOCK\_SIZE), prot); /\* 5 \*/
24. **return** dt\_virt;
25. }
26. 根据\_\_fix\_to\_virt计算fdt虚拟地址。
27. 虚拟地址必须2M对齐。主要是为了section mapping。
28. 在early\_fixmap\_init函数中，已经建立了PUD/PMD页表。为了让映射过程不用分配额外的PMD页表内存，这里必须保证FDT所在的虚拟地址范围落在early\_fixmap\_init函数建立的PMD页表范围内。
29. 创建页表。这里映射2M空间。
30. 如果dtb文件结尾地址超过上一个建立映射的地址范围，就必须紧接着再映射2M空间。

# slub分配器

Linux的物理内存管理采用了以页为单位的buddy system（伙伴系统），但是很多情况下，内核仅仅需要一个较小的对象空间，而且这些小块的空间对于不同对象又是变化的、不可预测的，所以需要一种类似用户空间堆内存的管理机制(malloc/free)。然而内核对对象的管理又有一定的特殊性，有些对象的访问非常频繁，需要采用缓冲机制；对象的组织需要考虑硬件cache的影响；需要考虑多处理器以及NUMA架构的影响。90年代初期，在Solaris 2.4操作系统中，采用了一种称为“slab”（原意是大块的混凝土）的缓冲区分配和管理方法，在相当程度上满足了内核的特殊需求。

多年以来，SLAB成为linux kernel对象缓冲区管理的主流算法，甚至长时间没有人愿意去修改，因为它实在是非常复杂，而且在大多数情况下，它的工作完成的相当不错。但是，随着大规模多处理器系统和 NUMA系统的广泛应用，SLAB 分配器逐渐暴露出自身的严重不足：

1). 缓存队列管理复杂；

2). 管理数据存储开销大；

3). 对NUMA支持复杂；

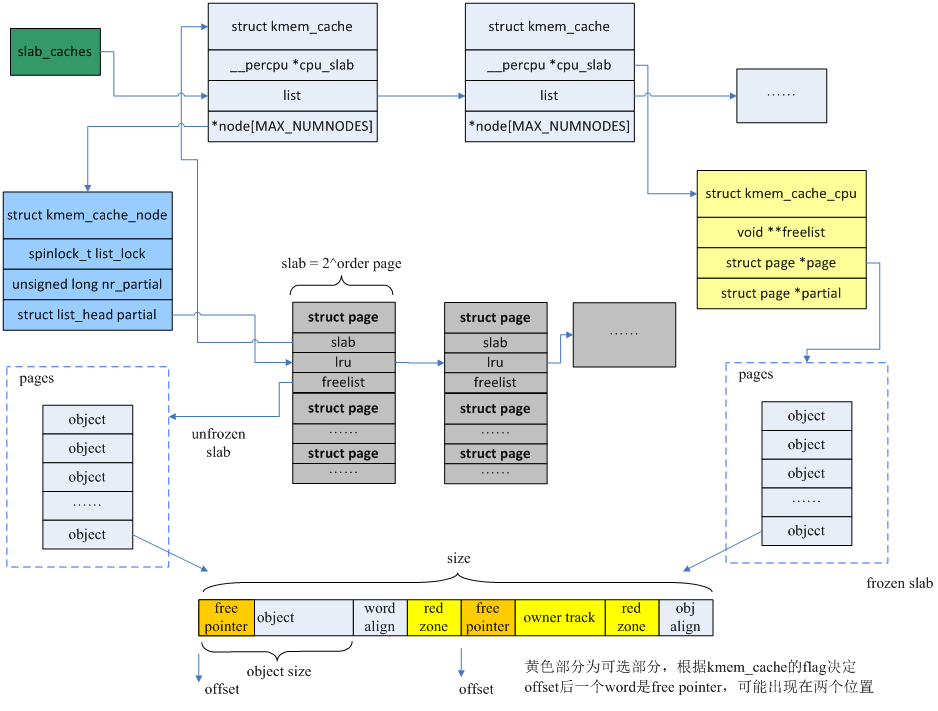
4). 调试调优困难；

5). 摒弃了效果不太明显的slab着色机制；

针对这些SLAB不足，内核开发人员Christoph Lameter在Linux内核2.6.22版本中引入一种新的解决方案：SLUB分配器。SLUB分配器特点是简化设计理念，同时保留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。SLUB分配器简化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。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。

（注：本文源码分析基于linux-4.1.x）

整体数据结构关系如下图所示：



## 1 SLUB分配器的初始化

SLUB初始化有两个重要的工作：第一，创建用于申请struct kmem\_cache和struct kmem\_cache\_node的kmem\_cache；第二，创建用于常规kmalloc的kmem\_cache。

**1.1 申请kmem\_cache的kmem\_cache**

第一个工作涉及到一个“先有鸡还是先有蛋”的问题，因为创建kmem\_cache需要从kmem\_cache的缓冲区申请，而这时候还没有创建kmem\_cache的缓冲区。kernel的解决办法是先用两个静态变量boot\_kmem\_cache和boot\_kmem\_cache\_node来保存struct kmem\_cach和struct kmem\_cache\_node缓冲区管理数据，以两个静态变量为基础申请大小为struct kmem\_cache和struct kmem\_cache\_node对象大小的slub缓冲区，随后再从这些缓冲区中分别申请两个kmem\_cache，然后把boot\_kmem\_cache和boot\_kmem\_cache\_node中的内容拷贝到新申请的对象中，从而完成了struct kmem\_cache和struct kmem\_cache\_node管理结构的bootstrap（自引导）。

1. **void** \_\_init kmem\_cache\_init(**void**)
2. {
3. //声明静态变量，存储临时kmem\_cache管理结构
4. **static** \_\_initdata **struct** kmem\_cache boot\_kmem\_cache,
5. boot\_kmem\_cache\_node;
6. •••
7. kmem\_cache\_node = &boot\_kmem\_cache\_node;
8. kmem\_cache = &boot\_kmem\_cache;
10. //申请slub缓冲区，管理数据放在临时结构中
11. create\_boot\_cache(kmem\_cache\_node, "kmem\_cache\_node",
12. **sizeof**(**struct** kmem\_cache\_node), SLAB\_HWCACHE\_ALIGN);
13. create\_boot\_cache(kmem\_cache, "kmem\_cache",
14. offsetof(**struct** kmem\_cache, node) +
15. nr\_node\_ids \* **sizeof**(**struct** kmem\_cache\_node \*),
16. SLAB\_HWCACHE\_ALIGN);
18. //从刚才挂在临时结构的缓冲区中申请kmem\_cache的kmem\_cache，并将管理数据拷贝到新申请的内存中
19. kmem\_cache = bootstrap(&boot\_kmem\_cache);
21. //从刚才挂在临时结构的缓冲区中申请kmem\_cache\_node的kmem\_cache，并将管理数据拷贝到新申请的内存中
22. kmem\_cache\_node = bootstrap(&boot\_kmem\_cache\_node);
23. •••
24. }

**1.2 创建kmalloc常规缓存**

原则上系统会为每个2次幂大小的内存块申请一个缓存，但是内存块过小时，会产生很多碎片浪费，所以系统为96B和192B也各自创建了一个缓存。于是利用了一个size\_index数组来帮助确定小于192B的内存块使用哪个缓存

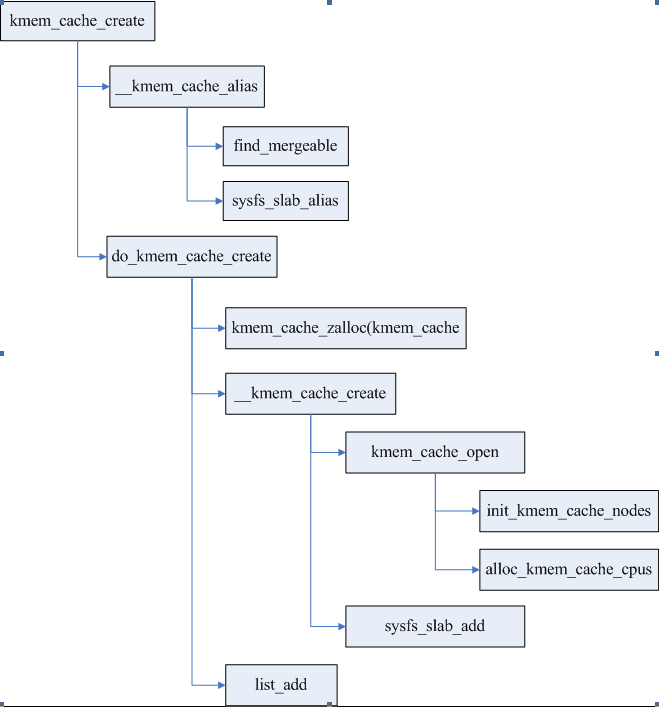
1. **void** \_\_init create\_kmalloc\_caches(**unsigned** **long** flags)
2. {
3. •••
4. /\*使用SLUB时KMALLOC\_SHIFT\_LOW=3，KMALLOC\_SHIFT\_HIGH=13
5. 也就是说使用kmalloc能够申请的最小内存是8B，最大内存是8KB
6. 申请内存是向上对其2的n次幂，创建对应大小缓存保存在kmalloc\_caches [n]\*/
7. **for** (i = KMALLOC\_SHIFT\_LOW; i <= KMALLOC\_SHIFT\_HIGH; i++) {
8. **if** (!kmalloc\_caches[i]) {
9. kmalloc\_caches[i] = create\_kmalloc\_cache(NULL,
10. 1 << i, flags);
11. }
13. /\*
14. \* Caches that are not of the two-to-the-power-of size.
15. \* These have to be created immediately after the
16. \* earlier power of two caches
17. \*/
18. /\*有两个例外，大小为64~96B和128B~192B，单独创建了两个缓存
19. 保存在kmalloc\_caches [1]和kmalloc\_caches [2]\*/
20. **if** (KMALLOC\_MIN\_SIZE <= 32 && !kmalloc\_caches[1] && i == 6)
21. kmalloc\_caches[1] = create\_kmalloc\_cache(NULL, 96, flags);
23. **if** (KMALLOC\_MIN\_SIZE <= 64 && !kmalloc\_caches[2] && i == 7)
24. kmalloc\_caches[2] = create\_kmalloc\_cache(NULL, 192, flags);
25. }
26. •••
27. }

## 2 缓存的创建与销毁

**2.1 缓存的创建**

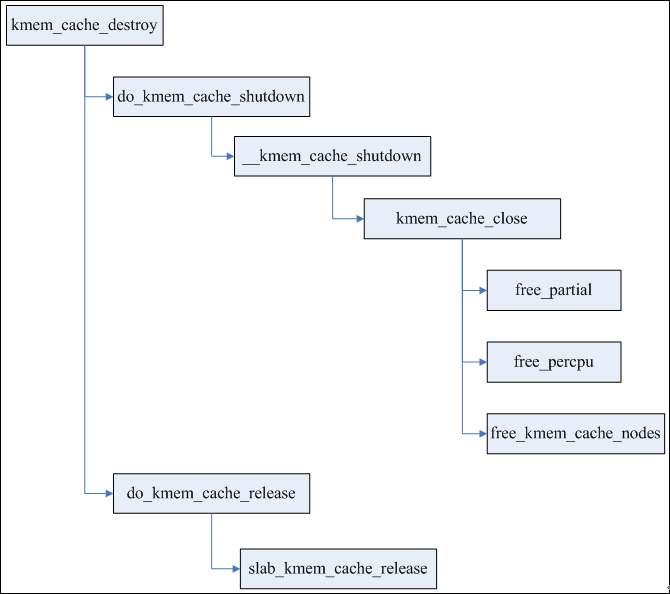
创建缓存通过接口kmem\_cache\_create进行，在创建新的缓存以前，尝试找到可以合并的缓存，合并条件包括对对象大小以及缓存属性的判断，如果可以合并则直接返回已存在的kmem\_cache，并创建一个kobj链接指向同一个节点。

创建新的缓存主要是申请管理结构暂用的空间，并初始化，这些管理结构包括kmem\_cache、kmem\_cache\_nodes、kmem\_cache\_cpu。同时在sysfs创建kobject节点。最后把kmem\_cache加入到全局cahce链表slab\_caches中。



2.2 缓存的销毁

销毁过程比创建过程简单的多，主要工作是释放partial队列所有page，释放kmem\_cache\_cpu，释放每个node的kmem\_cache\_node，最后释放kmem\_cache本身。



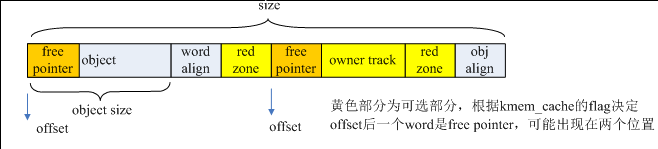
## 3 申请对象

对象是SLUB分配器中可分配的内存单元，与SLAB相比，SLUB对象的组织非常简洁，申请过程更加高效。SLUB没有任何管理区结构来管理对象，而是将对象之间的关联嵌入在对象本身的内存中，因为申请者并不关心对象在分配之前内存的内容是什么。而且各个SLUB之间的关联，也利用page自身结构进行处理。

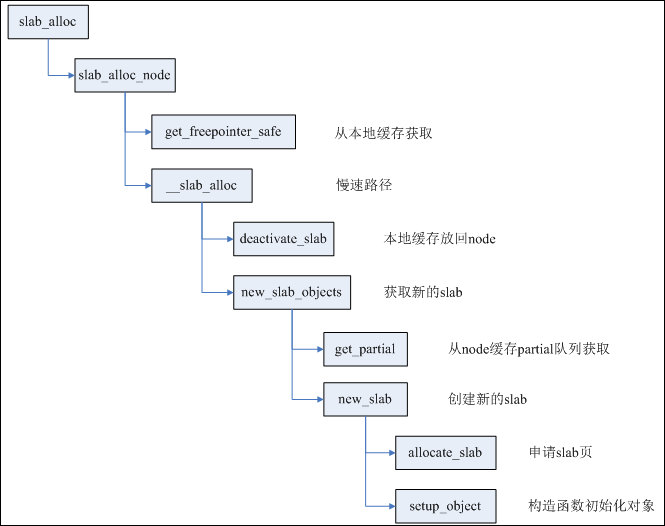
每个CPU都有一个slab作为本地高速缓存，只要slab所在的node与申请者要求的node匹配，同时该slab还有空闲对象，则直接在cpu\_slab中取出空闲对象，否则就进入慢速路径。

每个对象内存的offset偏移位置都存放着下一个空闲对象，offset通常为0，也就是复用对象内存的第一个字来保存下一个空闲对象的指针，当满足条件(flags & (SLAB\_DESTROY\_BY\_RCU | SLAB\_POISON)) 或者有对象构造函数时，offset不为0，每个对象的结构如下图。

cpu\_slab的freelist则保存着当前第一个空闲对象的地址。



如果本地CPU缓存没有空闲对象，则申请新的slab；如果有空闲对象，但是内存node不相符，则deactive当前cpu\_slab，再申请新的slab。



deactivate\_slab主要进行两步工作：

第一步，将cpu\_slab的freelist全部释放回page->freelist；

第二部，根据page(slab)的状态进行不同操作，如果该slab有部分空闲对象，则将page移到kmem\_cache\_node的partial队列；如果该slab全部空闲，则直接释放该slab；如果该slab全部占用，而且开启了CONFIG\_SLUB\_DEBUG编译选项，则将page移到full队列。

page的状态也从frozen改变为unfrozen。（frozen代表slab在cpu\_slub，unfroze代表在partial队列或者full队列）。

申请新的slab有两种情况，如果cpu\_slab的partial队列不为空，则取出队列中下一个page作为新的cpu\_slab，同时再次检测内存node是否相符，还不相符则循环处理。

如果cpu\_slab的partial队列为空，则查看本node的partial队列是否为空，如果不空，则取出page；如果为空，则看一定距离范围内其它node的partial队列，如果还为空，则需要创建新slab。

 创建新slab其实就是申请对应order的内存页，用来放足够数量的对象。值得注意的是其中order以及对象数量的确定，这两者又是相互影响的。order和object数量同时存放在kmem\_cache成员kmem\_cache\_order\_objects中，低16位用于存放object数量，高位存放order。order与object数量的关系非常简单：((PAGE\_SIZE << order) - reserved) / size。

下面重点看calculate\_order这个函数

1. **static** **inline** int calculate\_order(int size, int reserved)
2. {
3. •••
4. //尝试找到order与object数量的最佳配合方案
5. //期望的效果就是剩余的碎片最小
6. min\_objects = slub\_min\_objects;
7. **if** (!min\_objects)
8. min\_objects = 4 \* (fls(nr\_cpu\_ids) + 1);
9. max\_objects = order\_objects(slub\_max\_order, size, reserved);
10. min\_objects = min(min\_objects, max\_objects);
12. //fraction是碎片因子，需要满足的条件是碎片部分乘以fraction小于slab大小
13. // (slab\_size - reserved) % size <= slab\_size / fraction
14. **while** (min\_objects > 1) {
15. fraction = 16;
16. **while** (fraction >= 4) {
17. order = slab\_order(size, min\_objects,
18. slub\_max\_order, fraction, reserved);
19. **if** (order <= slub\_max\_order)
20. **return** order;
21. //放宽条件，容忍的碎片大小增倍
22. fraction /= 2;
23. }
24. min\_objects--;
25. }
27. //尝试一个slab只包含一个对象
28. order = slab\_order(size, 1, slub\_max\_order, 1, reserved);
29. **if** (order <= slub\_max\_order)
30. **return** order;
32. //使用MAX\_ORDER且一个slab只含一个对象
33. order = slab\_order(size, 1, MAX\_ORDER, 1, reserved);
34. **if** (order < MAX\_ORDER)
35. **return** order;
36. **return** -ENOSYS;
37. }

## 4 释放对象

从上面申请对象的流程也可以看出，释放的object有几个去处：

1）cpu本地缓存slab，也就是cpu\_slab；

2）放回object所在的page（也就是slab）中；另外要处理所在的slab：

2.1）如果放回之后，slab完全为空，则直接销毁该slab；

2.2）如果放回之前，slab为满，则判断slab是否已被冻结；如果已冻结，则不需要做其他事；如果未冻结，则将其冻结，放入cpu\_slab的partial队列；如果cpu\_slab partial队列过多，则将队列中所有slab一次性解冻到各自node的partial队列中。

值得注意的是cpu partial队列的功能是个可选项，依赖于内核选项CONFIG\_SLUB\_CPU\_PARTIAL，如果没有开启，则不使用cpu partial队列，直接使用各个node的partial队列。

# 图解slub

## **1.前言**

在Linux中，伙伴系统（buddy system）是以页为单位管理和分配内存。但是现实的需求却以字节为单位，假如我们需要申请20Bytes，总不能分配一页吧！那岂不是严重浪费内存。那么该如何分配呢？slab分配器就应运而生了，专为小内存分配而生。slab分配器分配内存以Byte为单位。但是slab分配器并没有脱离伙伴系统，而是基于伙伴系统分配的大内存进一步细分成小内存分配。

前段时间学习了下slab分配器工作原理。因为自己本身是做手机的，发现现在好像都在使用slub分配器，想想还是再研究一下slub的工作原理。之前看了代码，感觉挺多数据结构和成员的。成员的意思是什么？数据结构之间的关系是什么？不知道你是否感觉云里雾里。既然代码阅读起来晦涩难懂，如果有精美的配图，不知是否有助于阁下理解slub的来龙去脉呢？我想表达的意思就是文章图多，图多，图多。我们只说原理，尽量不看代码。因为所有代码中包含的内容我都会用图来说明。你感兴趣绝对有助于你看代码。

说明：slub是slab中的一种，slab也是slab中的一种。有时候用slab来统称slab, slub和slob。slab, slub和slob仅仅是分配内存策略不同。本篇文章中说的是slub分配器工作的原理。但是针对分配器管理的内存，下文统称为slab缓存池。所以文章中slub和slab会混用，表示同一个意思。

注：文章代码分析基于linux-4.15.0-rc3。 图片有点走形，请单独点开图片查看。

## **2. slub数据结构**

slub的数据结构相对于slab来说要简单很多。并且对外接口和slab兼容。所以说，从slab的系统更换到slub，可以说是易如反掌。

### 2.1. kmem\_cache

现在假如从伙伴系统分配一页内存供slub分配器管理。对于slub分配器来说，就是将这段连续内存平均分成若干大小相等的object（对象）进行管理。可是我们总得知道每一个object的size吧！管理的内存页数也是需要知道的吧！不然怎么知道如何分配呢！因此需要一个数据结构管理。那就是struct kmem\_cache。kmem\_cache数据结构描述如下：

1. **struct** kmem\_cache {
2. **struct** kmem\_cache\_cpu \_\_percpu \*cpu\_slab;
3. /\* Used for retriving partial slabs etc \*/
4. slab\_flags\_t flags;
5. **unsigned** **long** min\_partial;
6. int size; /\* The size of an object including meta data \*/
7. int object\_size; /\* The size of an object without meta data \*/
8. int offset; /\* Free pointer offset. \*/
9. #ifdef CONFIG\_SLUB\_CPU\_PARTIAL
10. int cpu\_partial; /\* Number of per cpu partial objects to keep around \*/
11. #endif
12. **struct** kmem\_cache\_order\_objects oo;
13. /\* Allocation and freeing of slabs \*/
14. **struct** kmem\_cache\_order\_objects max;
15. **struct** kmem\_cache\_order\_objects min;
16. gfp\_t allocflags; /\* gfp flags to use on each alloc \*/
17. int refcount; /\* Refcount for slab cache destroy \*/
18. **void** (\*ctor)(**void** \*);
19. int inuse; /\* Offset to metadata \*/
20. int align; /\* Alignment \*/
21. int reserved; /\* Reserved bytes at the end of slabs \*/
22. **const** **char** \*name; /\* Name (only for display!) \*/
23. **struct** list\_head list; /\* List of slab caches \*/
24. **struct** kmem\_cache\_node \*node[MAX\_NUMNODES];
25. };

1)     cpu\_slab：一个per cpu变量，对于每个cpu来说，相当于一个本地内存缓存池。当分配内存的时候优先从本地cpu分配内存以保证cache的命中率。

2)     flags：object分配掩码，例如经常使用的SLAB\_HWCACHE\_ALIGN标志位，代表创建的kmem\_cache管理的object按照硬件cache 对齐，一切都是为了速度。

3)     min\_partial：限制struct kmem\_cache\_node中的partial链表slab的数量。虽说是mini\_partial，但是代码的本意告诉我这个变量是kmem\_cache\_node中partial链表最大slab数量，如果大于这个mini\_partial的值，那么多余的slab就会被释放。

4)     size：分配的object size

5)     object\_size：实际的object size，就是创建kmem\_cache时候传递进来的参数。和size的关系就是，size是各种地址对齐之后的大小。因此，size要大于等于object\_size。

6)     offset：slub分配在管理object的时候采用的方法是：既然每个object在没有分配之前不在乎每个object中存储的内容，那么完全可以在每个object中存储下一个object内存首地址，就形成了一个单链表。很巧妙的设计。那么这个地址数据存储在object什么位置呢？offset就是存储下个object地址数据相对于这个object首地址的偏移。

7)     cpu\_partial：per cpu partial中所有slab的free object的数量的最大值，超过这个值就会将所有的slab转移到kmem\_cache\_node的partial链表。

8)     oo：低16位代表一个slab中所有object的数量（oo & ((1 << 16) - 1)），高16位代表一个slab管理的page数量（(2^(oo  16)) pages）。

9)     max：看了代码好像就是等于oo。

10)  min：当按照oo大小分配内存的时候出现内存不足就会考虑min大小方式分配。min只需要可以容纳一个object即可。

11)  allocflags：从伙伴系统分配内存掩码。

12)  inuse：object\_size按照word对齐之后的大小。

13)  align：字节对齐大小。

14)  name：sysfs文件系统显示使用。

15)  list：系统有一个slab\_caches链表，所有的slab都会挂入此链表。

16)  node：slab节点。在NUMA系统中，每个node都有一个struct kmem\_cache\_node数据结构。

### 2.2. kmem\_cache\_cpu

struct kmem\_cache\_cpu是对本地内存缓存池的描述，每一个cpu对应一个结构体。其数据结构如下：

1. **struct** kmem\_cache\_cpu {
2. **void** \*\*freelist; /\* Pointer to next available object \*/
3. **unsigned** **long** tid; /\* Globally unique transaction id \*/
4. **struct** page \*page; /\* The slab from which we are allocating \*/
5. #ifdef CONFIG\_SLUB\_CPU\_PARTIAL
6. **struct** page \*partial; /\* Partially allocated frozen slabs \*/
7. #endif
8. };

1)     freelist：指向下一个可用的object。

2)     tid：一个神奇的数字，主要用来同步作用的。

3)     page：slab内存的page指针。

4)     partial：本地slab partial链表。主要是一些部分使用object的slab。

### 2.3. kmem\_cache\_node

slab节点使用struct kmem\_cache\_node结构体描述。对于slub分配器来说，成员很少，远比slab分配器简洁。

1. **struct** kmem\_cache\_node {
2. spinlock\_t list\_lock;
3. **unsigned** **long** nr\_partial;
4. **struct** list\_head partial;
5. };

1)     list\_lock：自旋锁，保护数据。

2)     nr\_partial：slab节点中slab的数量。

3)     partial：slab节点的slab partial链表，和struct kmem\_cache\_cpu的partial链表功能类似。

### 2.4. slub接口

了解了基本的数据结构，再来看看slub提供的API。如果你了解slub，我想这几个接口你是再熟悉不过了。

1. **struct** kmem\_cache \*kmem\_cache\_create(**const** **char** \*name,
2. size\_t size,
3. size\_t align,
4. **unsigned** **long** flags,
5. **void** (\*ctor)(**void** \*));
6. **void** kmem\_cache\_destroy(**struct** kmem\_cache \*);
7. **void** \*kmem\_cache\_alloc(**struct** kmem\_cache \*cachep, int flags);
8. **void** kmem\_cache\_free(**struct** kmem\_cache \*cachep, **void** \*objp);

1) kmem\_cache\_create是创建kmem\_cache数据结构，参数描述如下：

    name：kmem\_cache的名称

    size ：slab管理对象的大小

    align：slab分配器分配内存的对齐字节数(以align字节对齐)

    flags：分配内存掩码

    ctor ：分配对象的构造回调函数

2) kmem\_cache\_destroy作用和kmem\_cache\_create相反，就是销毁创建的kmem\_cache。

3) kmem\_cache\_alloc是从cachep参数指定的kmem\_cache管理的内存缓存池中分配一个对象，其中flags是分配掩码，GFP\_KERNEL是不是很熟悉的掩码？

4) kmem\_cache\_free是kmem\_cache\_alloc的反操作

slab分配器提供的接口该如何使用呢？其实很简单，总结分成以下几个步骤：

1) kmem\_cache\_create创建一个kmem\_cache数据结构。

2) 使用kmem\_cache\_alloc接口分配内存，kmem\_cache\_free接口释放内存。

3) release第一步创建的kmem\_cache数据结构。

再来一段demo示例代码就更好了。

1. /\*
2. \*　This is a demo for how to use kmem\_cache\_create
3. \*/
4. **void** slab\_demo(**void**)
5. {
6. **struct** kmem\_cache \*kmem\_cache\_16 = kmem\_cache\_create("kmem\_cache\_16", 16,
7. 8, ARCH\_KMALLOC\_FLAGS,
8. NULL);
10. /\* now you can alloc memory, the buf points to 16 bytes of memory\*/
11. **char** \*buf = kmeme\_cache\_alloc(kmem\_cache\_16, GFP\_KERNEL);
13. /\*
14. \* do something what you what, don't forget to release the memory after use
15. \*/
16. kmem\_cache\_free(kmem\_cache\_16, buf);
17. kmem\_cache\_destroy(kmem\_cache\_16);
18. }

1) 首先使用kmem\_cache\_create创建名称为kmem\_cache\_16的kmem\_cache，该kmem\_cache主要是描述如何管理一堆对象，其实就是slab的布局。每个对象都是16字节，并且分配的对象地址按照8字节对齐，也就是说从kmem\_cache\_16中分配的对象大小全是16字节。不管你要申请多少，反正就是16Bytes。当然，kmem\_cache\_create仅仅是创建了一个描述slab缓存池布局的数据结构，并没有从伙伴系统申请内存，具体的申请内存操作是在kmeme\_cache\_alloc中完成的。

2) kmeme\_cache\_alloc从kmem\_cache\_16分配一个对象。

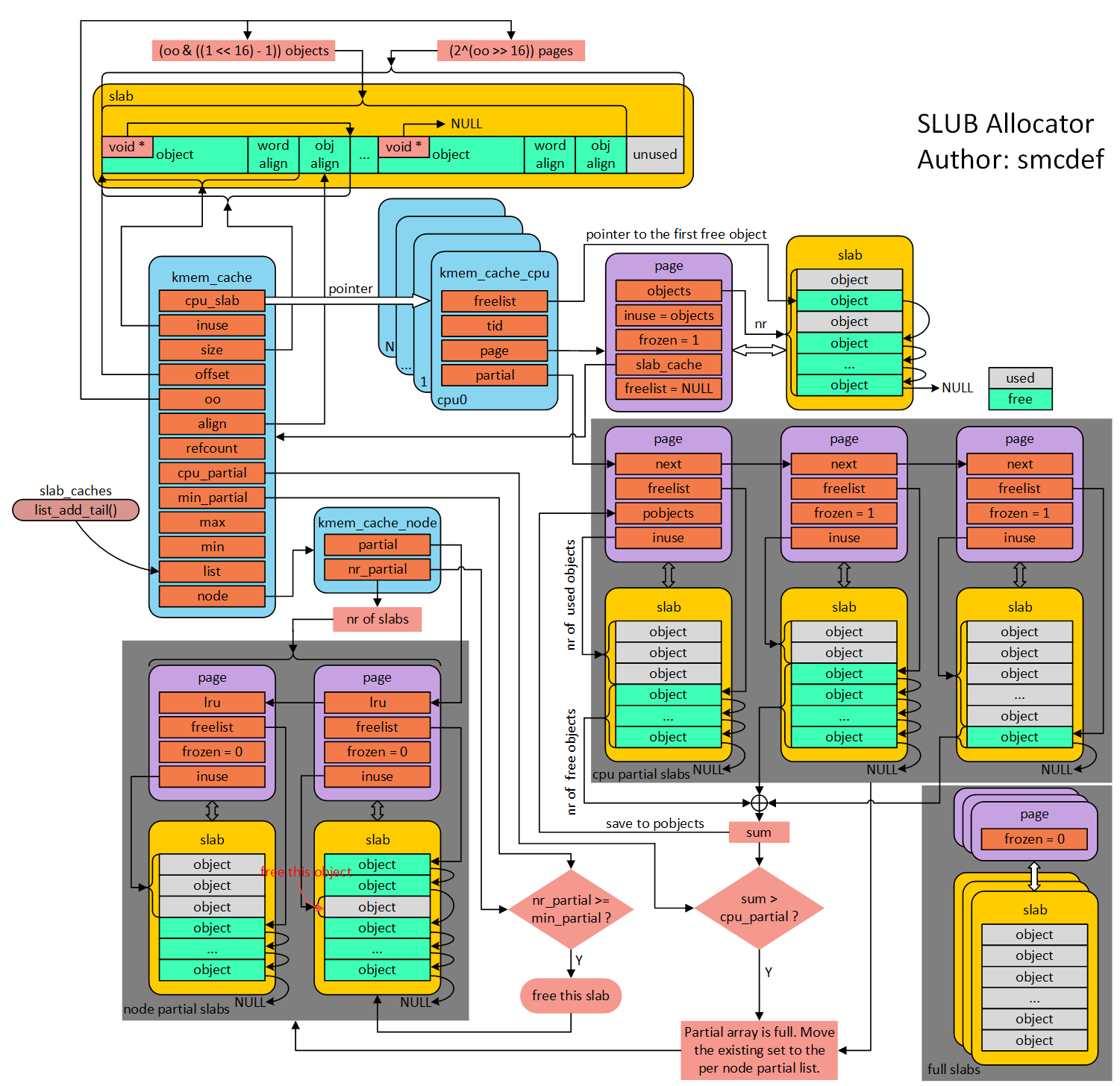
3) 内存使用结束记得kmem\_cache\_free释放。

4) 如果不需要这个kmem\_cache的话，就可以调用kmem\_cache\_destroy进行销毁吧。在释放kmem\_cache之前要保证从该kmem\_cache中分配的对象全部释放了，否则无法释放kmem\_cache。

## **3. slub数据结构之间关系**

什么是slab缓存池呢？我的解释是使用struct kmem\_cache结构描述的一段内存就称作一个slab缓存池。一个slab缓存池就像是一箱牛奶，一箱牛奶中有很多瓶牛奶，每瓶牛奶就是一个object。分配内存的时候，就相当于从牛奶箱中拿一瓶。总有拿完的一天。当箱子空的时候，你就需要去超市再买一箱回来。超市就相当于partial链表，超市存储着很多箱牛奶。如果超市也卖完了，自然就要从厂家进货，然后出售给你。厂家就相当于伙伴系统。

说了这么多终于要抛出辛辛苦苦画的美图了。

[](http://www.wowotech.net/content/uploadfile/201803/4a471520078976.png)

好了，后面说的大部分内容请看这张图。足以表明数据结构之间的关系了。看懂了这张图，就可以理清数据结构之间的关系了。

### 3.1. slub管理object方法

在图片的左上角就是一个slub缓存池中object的分布以及数据结构和kmem\_cache之间的关系。首先一个slab缓存池包含的页数是由oo决定的。oo拆分为两部分，低16位代表一个slab缓存池中object的数量，高16位代表包含的页数。使用kmem\_cache\_create()接口创建kmem\_cache的时候需要指出obj的size和对齐align。也就是传入的参数。kmem\_cache\_create()主要是就是填充kmem\_cache结构体成员。既然从伙伴系统得到(2^(oo >> 16)) pages大小内存，按照size大小进行平分。一般来说都不会整除，因此剩下的就是图中灰色所示。由于每一个object的大小至少8字节，当然可以用来存储下一个object的首地址。就像图中所示的，形成单链表。图中所示下个obj地址存放的位置位于每个obj首地址处，在内核中称作指针内置式。同时，下个obj地址存放的位置和obj首地址之间的偏移存储在kmem\_cache的offset成员。两外一种方式是指针外置式，即下个obj的首地址存储的位置位于obj尾部，也就是在obj尾部再分配sizeof(void \*)字节大小的内存。对于外置式则offset就等于kmem\_cache的inuse成员。

### 3.2. per cpu freelist

针对每一个cpu都会分配一个struct kmem\_cacche\_cpu的结构体。可以称作是本地缓存池。当内存申请的时候，优先从本地cpu缓存池申请。在分配初期，本地缓存池为空，自然要从伙伴系统分配一定页数的内存。内核会为每一个物理页帧创建一个struct page的结构体。kmem\_cacche\_cpu中page就会指向正在使用的slab的页帧。freelist成员指向第一个可用内存obj首地址。处于正在使用的slab的struct page结构体中的freelist会置成NULL，因为没有其他地方使用。struct page结构体中inuse代表已经使用的obj数量。这地方有个很有意思的地方，在刚从伙伴系统分配的slab的 inuse在分配初期就置成obj的总数，在分配obj的时候并不会改变。你是不是觉得很奇怪，既然表示已经使用obj的数量，为什么一直是obj的总数呢？你想想，slab中的对象总有分配完的时候，那个时候就直接脱离kmem\_cache\_cpu了。此时的inuse不就名副其实了嘛！对于full slab就像图的右下角，就像无人看管的孩子，没有任何链表来管理。

### 3.3. per cpu partial

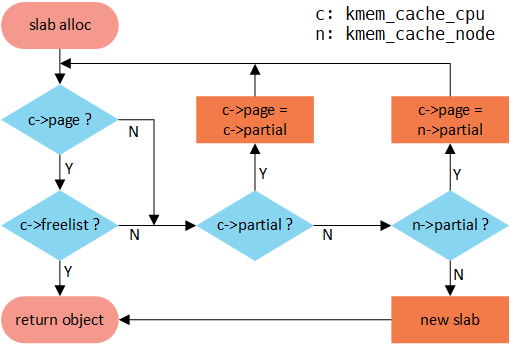
当图中右下角full slab释放obj的时候，首先就会将slab挂入per cpu partial链表管理。通过struct page中next成员形成单链表。per cpu partial链表指向的第一个page中会存放一些特殊的数据。例如：pobjects存储着per cpu partial链表中所有slab可供分配obj的总数，如图所示。当然还有一个图中没有体现的pages成员存储per cpu partial链表中所有slab缓存池的个数。pobjects到底有什么用呢？我们从full slab中释放一个obj就添加到per cpu partial链表，总不能无限制的添加吧！因此，每次添加的时候都会判断当前的pobjects是否大于kmem\_cache的cpu\_partial成员，如果大于，那么就会将此时per cpu partial链表中所有的slab移送到kmem\_cache\_node的partial链表，然后再将刚刚释放obj的slab插入到per cpu partial链表。如果不大于，则更新pobjects和pages成员，并将slab插入到per cpu partial链表。

### 3.4. per node partial

per node partia链表类似per cpu partial，区别是node中的slab是所有cpu共享的，而per cpu是每个cpu独占的。假如现在的slab布局如上图所示。假如现在如红色箭头指向的obj将会释放，那么就是一个empty slab，此时判断kmem\_cache\_node的nr\_partial是否大于kmem\_cache的min\_partial，如果大于则会释放该slab的内存。

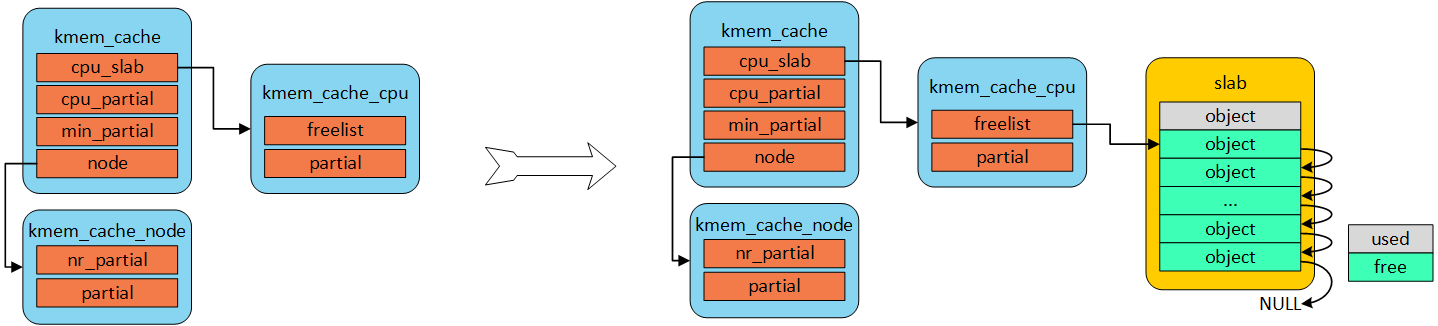
## **4. slub分配内存原理**

当调用kmem\_cache\_alloc()分配内存的时候，我们可以从正在使用slab分配，也可以从per cpu partial分配，同样还可以从per node partial分配，那么分配的顺序是什么呢？我们可以用下图表示。

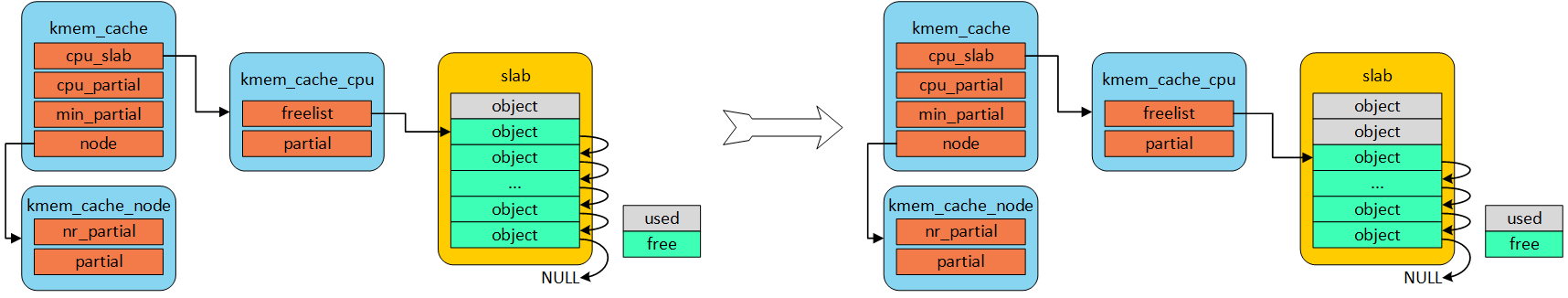
[](http://www.wowotech.net/content/uploadfile/201802/fb5c1519305301.png)

首先从cpu 本地缓存池分配，如果freelist不存在，就会转向per cpu partial分配，如果per cpu partial也没有可用对象，继续查看per node partial，如果很不幸也不没有可用对象的话，就只能从伙伴系统分配一个slab了，并挂入per cpu freelist。我们详细看一下这几种情况。

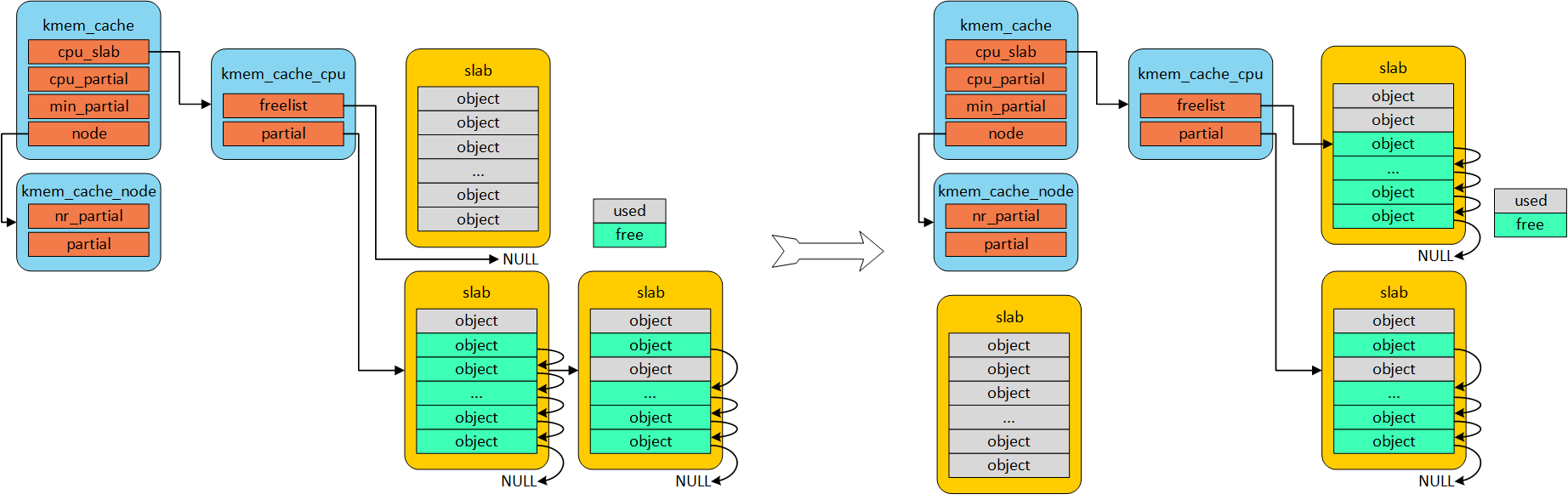
1)     kmem\_cache刚刚建立，还没有任何对象可供分配，此时只能从伙伴系统分配一个slab，如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/10fb1519305301.png)

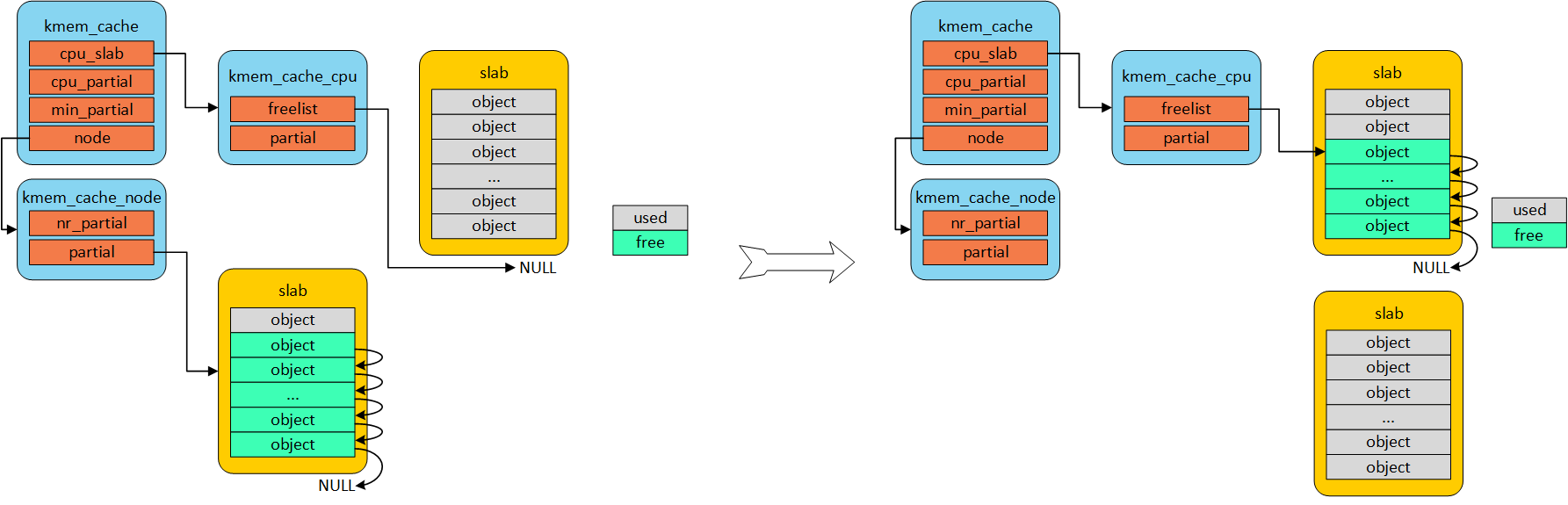
2)     如果正在使用的slab有free obj，那么就直接分配即可，这种是最简单快捷的。如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/09dd1519305302.png)

3)     随着正在使用的slab中obj的一个个分配出去，最终会无obj可分配，此时per cpu partial链表中有可用slab用于分配，那么就会从per cpu partial链表中取下一个slab用于分配obj。如下图所示。

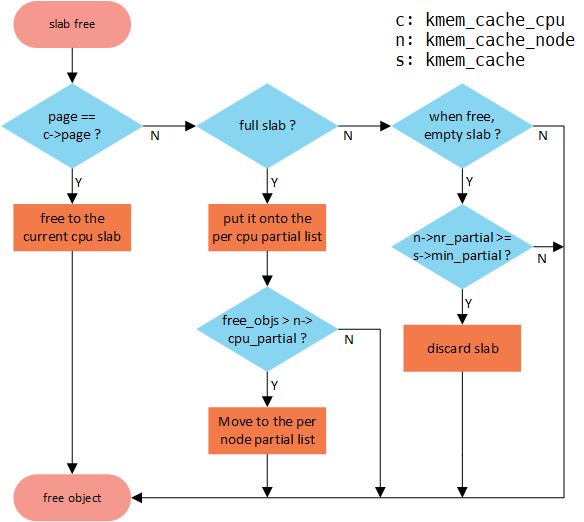
[](http://www.wowotech.net/content/uploadfile/201802/82661519305303.png)

4)     随着正在使用的slab中obj的一个个分配出去，最终会无obj可分配，此时per cpu partial链表也为空，此时发现per node partial链表中有可用slab用于分配，那么就会从per node partial链表中取下一个slab用于分配obj。如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/f19c1519305304.png)

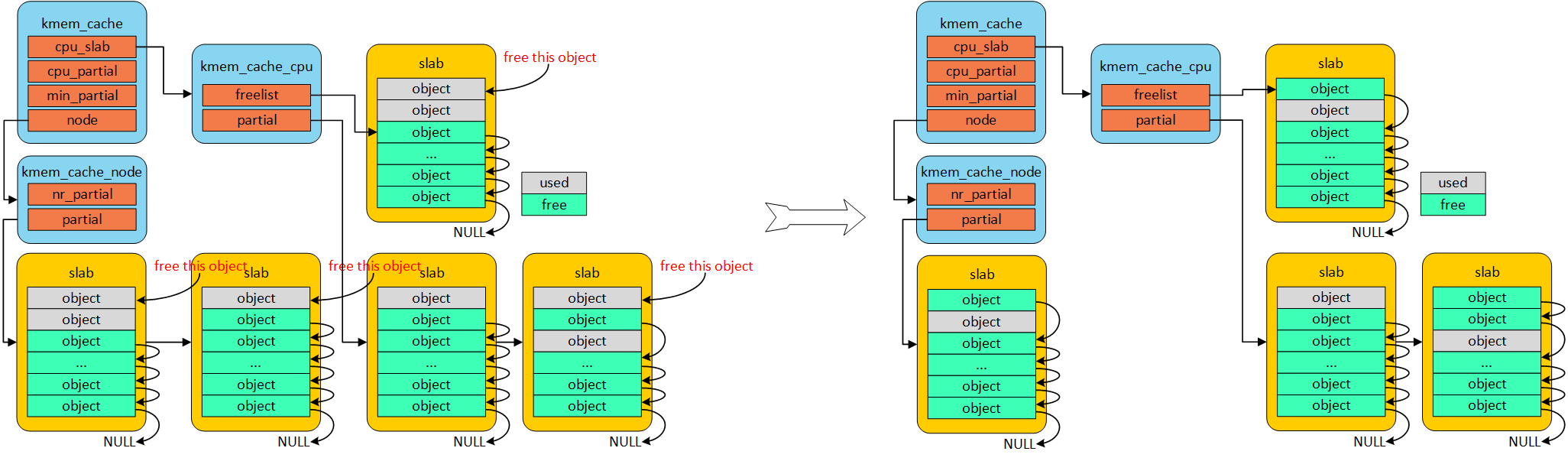
## **5. slub释放内存原理**

我们可以通过kmem\_cache\_free()接口释放申请的obj对象。释放对象的流程如下图所示。

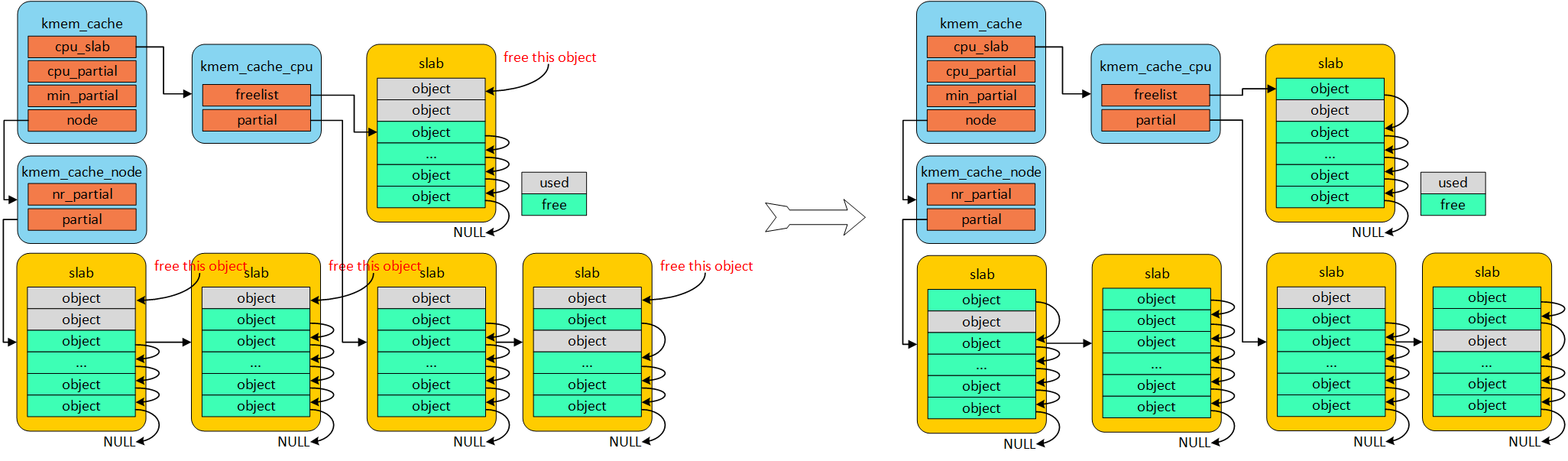
[](http://www.wowotech.net/content/uploadfile/201802/9eb91519305304.png)

如果释放的obj就是属于正在使用cpu上的slab，那么直接释放即可，非常简单；如果不是的话，首先判断所属slub是不是full状态，因为full slab是没妈的孩子，释放之后就变成partial empty，急需要找个链表领养啊！这个妈就是per cpu partial链表。如果per cpu partial链表管理的所有slab的free object数量超过kmem\_cache的cpu\_partial成员的话，就需要将per cpu partial链表管理的所有slab移动到per node partial链表管理；如果不是full slab的话，继续判断释放当前obj后的slab是否是empty slab，如果是empty slab，那么在满足kmem\_cache\_node的nr\_partial大于kmem\_cache的min\_partial的情况下，则会释放该slab的内存。其他情况就直接释放即可。

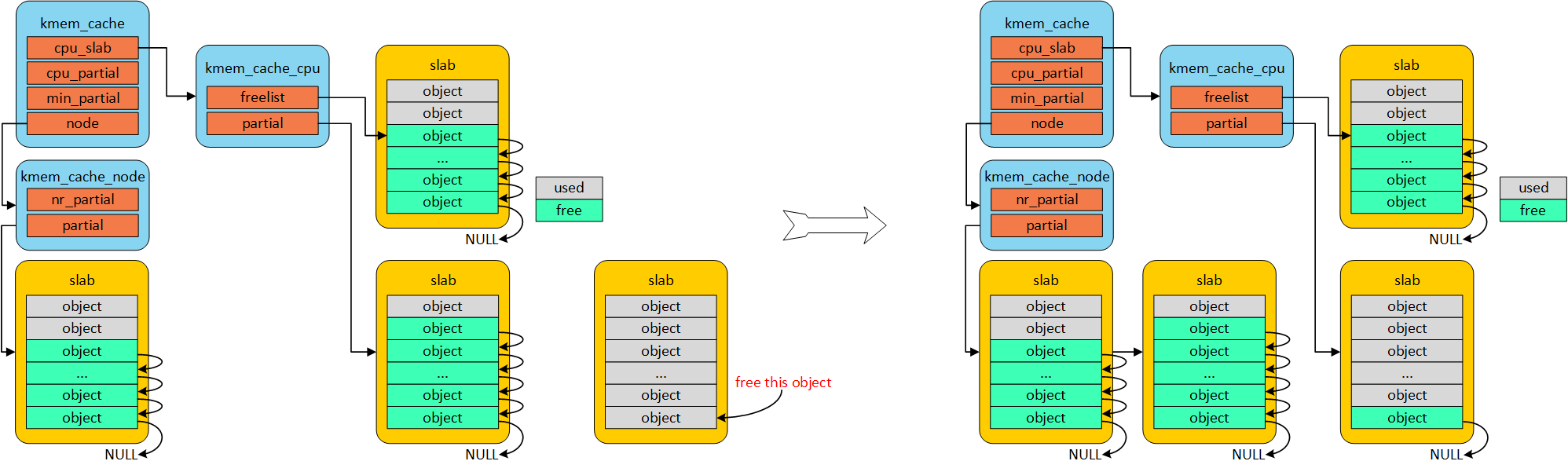
1)     假设下图左边的情况下释放obj，如果满足kmem\_cache\_node的nr\_partial大于kmem\_cache的min\_partial的话，释放情况如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/602e1519305305.png)

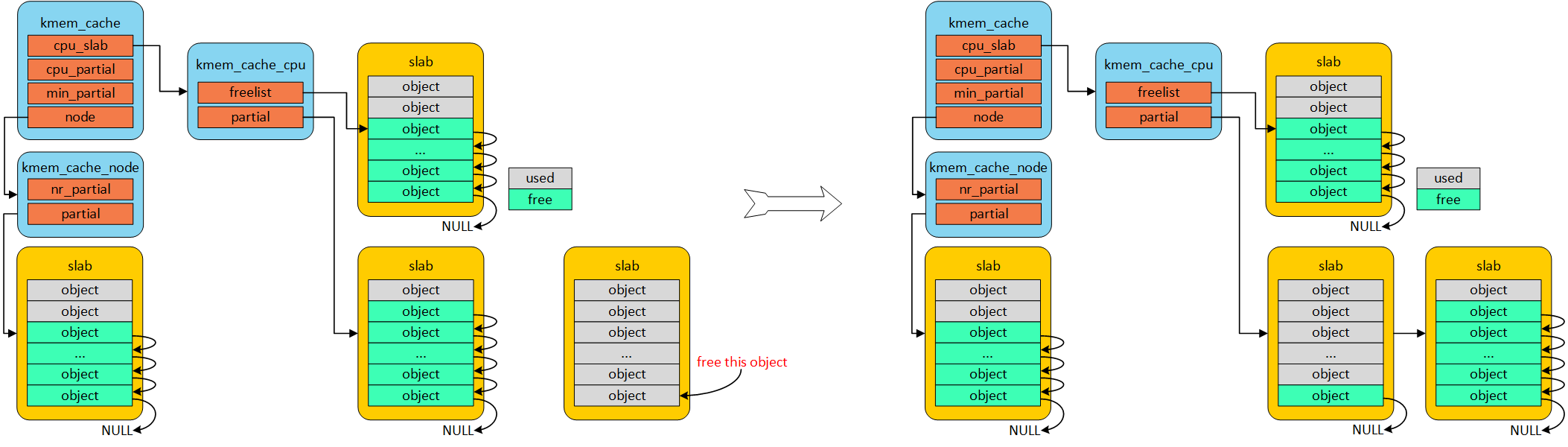
2)     假设下图左边的情况下释放obj，如果不满足kmem\_cache\_node的nr\_partial大于kmem\_cache的min\_partial的话，释放情况如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/7afb1519305306.png)

3)     假设下图从full slab释放obj的话，如果满足per cpu partial管理的所有slab的free object数量大于kmem\_cache的cpu\_partial成员的话的话，将per cpu partial链表管理的所有slab移动到per node partial链表管理，释放情况如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/586e1519305307.png)

4)     假设下图从full slab释放obj的话，如果不满足per cpu partial管理的所有slab的free object数量大于kmem\_cache的cpu\_partial成员的话的话，释放情况如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/59b21519305308.png)

## **6. kmalloc**

好了，说了这么多，估计你会感觉slab好像跟我们没什么关系。如果作为一个驱动开发者，是不是感觉自己写的driver从来没有使用过这些接口呢？其实我们经常使用，只不过隐藏在kmalloc的面具之下。

kmalloc的内存分配就是基于slab分配器，在系统启动初期调用create\_kmalloc\_caches ()创建多个管理不同大小对象的kmem\_cache，例如：8B、16B、32B、64B、…、64MB等大小。当然默认配置情况下，系统系统启动之后创建的最大size的kmem\_cache是kmalloc-8192。因此，通过slab接口分配的最大内存是8192 bytes。那么通过kmalloc接口申请的内存大于8192 bytes该怎么办呢？其实kmalloc会判断申请的内存是否大于8192 bytes，如果大于的话就会通过alloc\_pages接口申请内存。kmem\_cache的名称以及大小使用struct kmalloc\_info\_struct管理。所有管理不同大小对象的kmem\_cache的名称如下：

1. **const** **struct** kmalloc\_info\_struct kmalloc\_info[] \_\_initconst = {
2. {NULL, 0}, {"kmalloc-96", 96},
3. {"kmalloc-192", 192}, {"kmalloc-8", 8},
4. {"kmalloc-16", 16}, {"kmalloc-32", 32},
5. {"kmalloc-64", 64}, {"kmalloc-128", 128},
6. {"kmalloc-256", 256}, {"kmalloc-512", 512},
7. {"kmalloc-1024", 1024}, {"kmalloc-2048", 2048},
8. {"kmalloc-4096", 4096}, {"kmalloc-8192", 8192},
9. {"kmalloc-16384", 16384}, {"kmalloc-32768", 32768},
10. {"kmalloc-65536", 65536}, {"kmalloc-131072", 131072},
11. {"kmalloc-262144", 262144}, {"kmalloc-524288", 524288},
12. {"kmalloc-1048576", 1048576}, {"kmalloc-2097152", 2097152},
13. {"kmalloc-4194304", 4194304}, {"kmalloc-8388608", 8388608},
14. {"kmalloc-16777216", 16777216}, {"kmalloc-33554432", 33554432},
15. {"kmalloc-67108864", 67108864}
16. };

经过create\_kmalloc\_caches ()函数之后，系统通过create\_kmalloc\_cache()创建以上不同size的kmem\_cache，并将这些kmem\_cache存储在kmalloc\_caches全局变量中以备后续kmalloc分配内存。现在假如通过kmalloc(17, GFP\_KERNEL)申请内存，系统会从名称“kmalloc-32”管理的slab缓存池中分配一个对象。即使浪费了15Byte。

我们来看看kmalloc的实现方式。

1. **static** \_\_always\_inline **void** \*kmalloc(size\_t size, gfp\_t flags)
2. {
3. **if** (\_\_builtin\_constant\_p(size)) {
4. **if** (size > KMALLOC\_MAX\_CACHE\_SIZE)
5. **return** kmalloc\_large(size, flags);
6. **if** (!(flags & GFP\_DMA)) {
7. int index = kmalloc\_index(size);
8. **if** (!index)
9. **return** ZERO\_SIZE\_PTR;
10. **return** kmem\_cache\_alloc\_trace(kmalloc\_caches[index], flags, size);
11. }
12. }
13. **return** \_\_kmalloc(size, flags);
14. }

1)     \_\_builtin\_constant\_p是gcc工具用来判断参数是否是一个常数，毕竟有些操作对于常数来说是可以优化的。

2)     通过kmalloc\_index函数查找符合满足分配大小的最小kmem\_cache。

3)     将index作为下表从kmalloc\_caches数组中找到符合的kmem\_cache，并从slab缓存池中分配对象。

我们再看一下kmalloc\_index的实现。

1. **static** \_\_always\_inline int kmalloc\_index(size\_t size)
2. {
3. **if** (!size)
4. **return** 0;
5. **if** (size <= KMALLOC\_MIN\_SIZE)
6. **return** KMALLOC\_SHIFT\_LOW;
7. **if** (KMALLOC\_MIN\_SIZE <= 32 && size > 64 && size <= 96)
8. **return** 1;
9. **if** (KMALLOC\_MIN\_SIZE <= 64 && size > 128 && size <= 192)
10. **return** 2;
11. **if** (size <= 8) **return** 3;
12. **if** (size <= 16) **return** 4;
13. **if** (size <= 32) **return** 5;
14. **if** (size <= 64) **return** 6;
15. **if** (size <= 128) **return** 7;
16. **if** (size <= 256) **return** 8;
17. **if** (size <= 512) **return** 9;
18. **if** (size <= 1024) **return** 10;
19. **if** (size <= 2 \* 1024) **return** 11;
20. **if** (size <= 4 \* 1024) **return** 12;
21. **if** (size <= 8 \* 1024) **return** 13;
22. **if** (size <= 16 \* 1024) **return** 14;
23. **if** (size <= 32 \* 1024) **return** 15;
24. **if** (size <= 64 \* 1024) **return** 16;
25. **if** (size <= 128 \* 1024) **return** 17;
26. **if** (size <= 256 \* 1024) **return** 18;
27. **if** (size <= 512 \* 1024) **return** 19;
28. **if** (size <= 1024 \* 1024) **return** 20;
29. **if** (size <= 2 \* 1024 \* 1024) **return** 21;
30. **if** (size <= 4 \* 1024 \* 1024) **return** 22;
31. **if** (size <= 8 \* 1024 \* 1024) **return** 23;
32. **if** (size <= 16 \* 1024 \* 1024) **return** 24;
33. **if** (size <= 32 \* 1024 \* 1024) **return** 25;
34. **if** (size <= 64 \* 1024 \* 1024) **return** 26;
35. /\* Will never be reached. Needed because the compiler may complain \*/
36. **return** -1;
37. }

# SLUB DEBUG原理

## 1. 前言

在工作中，经常会遇到由于越界导致的各种奇怪的问题。为什么越界访问导致的问题很奇怪呢？在工作差不多半年的时间里我就遇到了很多越界访问导致的问题（不得不吐槽下IC厂商提供的driver，总是隐藏着bug）。比如说越界访问导致的死机问题，这种问题的出现一般需要长时间测试才能发现，而且发现的时候即使有panic log。你也没什么头绪。这是为什么呢？假设驱动A通过kmalloc()申请了一段内存，不注意越界改写了与其相邻的object的数据（经过我之前一篇SLUB的文章分析，你应该明白kmalloc基于kmem\_cache实现的），假设被改写的object是B驱动使用的，巧合B驱动使用object存储的是地址数据，如果B驱动访问这个地址。那么完了，B驱动死了，panic也是怪B驱动。试想一下，这块被改写的object是哪个驱动使用，是不是哪个驱动就倒霉了？并且每一次死机的log中panic极有可能发生在不同的模块。但是真正的元凶却是A驱动，他没事你还不知道，是不是很恐怖？简直是借刀杀人啊！当然，越界访问也不一定会死机。之前就遇到一个很奇怪的问题。有两个全局数组变量（用作存储字符串）分别被模块C和D使用。这两个数组是上层需要显示的name信息。当C和D模块都工作的时候，发现C模块的name显示不对，但是D模块的name显示正常。将D模块remove，发现C模块的name显示正确。当时看了下System.map文件，发现这两个全局数组变量分配的内存是在一起的，由于D模块越界写导致的。而这种情况就不会死机。但是当你遇到这种情况的时候，你很惊讶，怎么会这样？两个模块之间根本就没关系啊！如果完全不借助检测工具去查找问题是相当费时间的。而且有可能还没什么头绪。这种问题我们该怎么定位？因此我们遇到一种debug的手段，可以检测out-of-bounds（oob）问题。刚才的第一种情况就可以SLUB自带debug功能。针对第二种情况就需要借助更加强大的KASAN工具（后续会有文章介绍）。

因此，我们需要一种debug手段帮助我们定位问题。SLUB DEBUG就是其中的一种。但是SLUB DEBUG仅仅针对从slub分配器分配的内存，如果你需要检测从栈中或者数据区分配内存的问题，就不行了。当然了，你可以选择KASAN。本文主要关注SLUB DEBUG的原理，如何定位这些问题的。

SLUB DEBUG检测oob问题原理也很简单，既然为了发现是否越界，那么就在分配出去的内存尾部添加一段额外的内存，填充特殊数字（magic num）。我们只需要检测这块额外的内存的数据是否被修改就可以知道是否发生了oob情况。而这段额外的内存就叫做Redzone。直译过来“红色区域”是不是有种神圣不可侵犯的感觉。  
说明：slab是最早加入linux的,在那时只有slab的存在。随着时间的推移slub出现了，slub是在slab基础上进行的改进，在大型机上表现出色。而slob是针对小型系统设计的。由于slub实现的接口和slab接口保持一致（虽然你用的是slub分配器，但是很多函数名称和数据结构还是依然和slab一致），所以有时候用slab来统称slab, slub和slob。slab, slub和slob仅仅是分配内存策略不同。管理的思想基本一致。本篇文章中说的是slub分配器debug原理。但是针对分配器管理的内存，下文统称为slab缓存池。所以文章中slub和slab会混用，表示同一个意思。

注：文章代码分析基于linux-4.15.0-rc3。图片有点走形，请单独点开图片查看。

## 2. SLUB DEBUG功能

SLUB DEBUG可以检测内存越界（out-of-bounds）和访问已经释放的内存（use-after-free）等问题。  
2.1. 如何打开功能

重新配置kernel选项，打开如下选项即可。

CONFIG\_SLUB=y

CONFIG\_SLUB\_DEBUG=y

CONFIG\_SLUB\_DEBUG\_ON=y

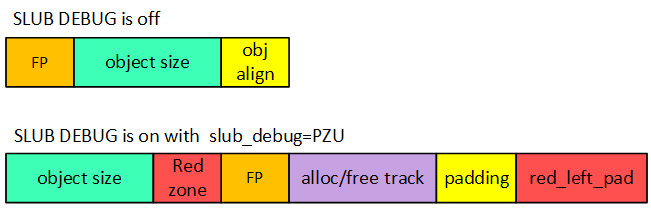
2.2. 如何使用

程序中的bug如果想用SLUB DEBUG去检测，还需要slabinfo命令。因为，SLUB内存检测功能在某些情况下不能立刻检测出来，必须主动触发，因此我们需要借助slabinfo命令触发SLUB allocator检测功能。和KASAN相比较而言，这也是SLUB DEBUG的一个劣势。毕竟KASAN可以做到在越界问题出现时就报出问题。  
slabinfo工具源码位于tools/vm目录。可以使用如下命令编译slabinfo工具（针对ARM64 architecture）。  
aarch64-linux-gnu-gcc -o slabinfo slabinfo.c

当系统开机之后，就可以运行slabinfo –v命令触发SLUB allocator检测所有的object，并将log信息输出到syslog。接下来的任务就是查看log信息是否包含SLUB allocator输出的bug log。其实有些bug是不需要运行slabinfo命令即可捕捉，但是有些却必须使用slabinfo –v命令才可以。下一节将会介绍SLUB DEBUG的原理，为你揭开哪些bug不需要slabinfo命令。

## 3. object layout

配置kernel选项CONFIG\_SLUB\_DEBUG\_ON后，在创建kmem\_cache的时候会传递很多flags（SLAB\_CONSISTENCY\_CHECKS、SLAB\_RED\_ZONE、SLAB\_POISON、SLAB\_STORE\_USER）。针对这些flags，SLUB allocator管理的object对象的format将会发生变化。如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/9eb61519307945.png)

SLUBU DEBUG关闭的情况下，free pointer是内嵌在object之中的，但是SLUB DEBUG打开之后，free pointer是在object之外，并且多了很多其他的内存，例如red zone、trace和red\_left\_pad等。这里之所以将FP后移就是因为为了检测use-after-free问题,当free object时会在将object填充magic num(0x6b)。如果不后移的话，岂不是破坏了object之间的单链表关系。

3.1. Red zone有什么用

从图中我们可以看到在object后面紧接着就是Red zone区域，那么Red zone有什么作用呢?既然紧随其后，自然是检测右边界越界访问（right out-of-bounds access）。原理很简单，在Red zone区域填充magic num，检查Red zone区域数据是否被修改即可知道是否发生right oob。

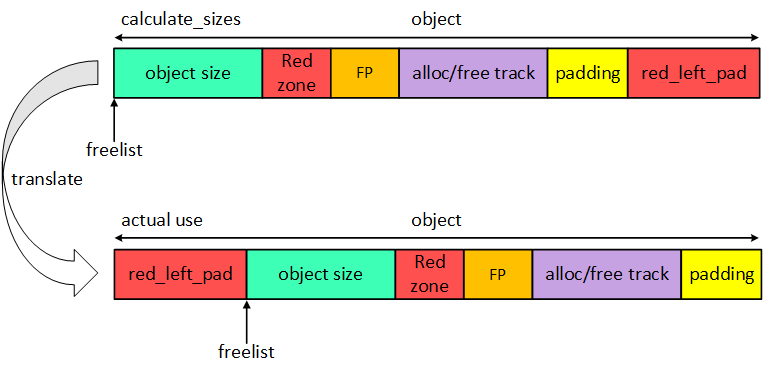
可能你会想到如果越过Red zone，直接改写了FP，岂不是检测不到oob了，并且链表结构也被破坏了。其实在check\_object()函数中会调用check\_valid\_pointer()来检查FP是否valid，如果invalid，同样会print error syslog。

3.2. padding有什么用

padding是sizeof(void \*) bytes的填充区域，在分配slab缓存池时，会将所有的内存填充0x5a。同样在free/alloc object的时候作为检测的一种途径。如果padding区域的数据不是0x5a，就代表发生了“Object padding overwritten”问题。这也是有可能，越界跨度很大。

3.3. red\_left\_pad有什么用

red\_left\_pad和Red zone的作用一致。都是为了检测oob。区别就是Red zone检测right oob，而red\_left\_pad是检测left oob。如果仅仅看到上面图片中object layout。你可能会好奇，如果发生left oob，那么应该是前一个object的red\_left\_pad区域被改写，而不是当前object的red\_left\_pad。如果你注意到这个问题，还是很机智的，这都被你发现了。为了避免这种情况的发生，SLUB allocator在初始化slab缓存池的时候会做一个转换。

[](http://www.wowotech.net/content/uploadfile/201802/c00b1519307827.png)

如果你去追踪kmem\_cache\_create()，在calculate\_sizes()中布局object。区域划分的layout就如同你看到上图的上半部分。当我第一次看到这段代码的时候，我也这么认为。实际上却不是这样的。在struct page结构中有一个freelist指针，freelist会指向第一个available object。在构建object之间的单链表的时候，object首地址实际上都会加上一个red\_left\_pad的偏移，这样实际的layout就如同图片中转换之后的layout。为什么会这样呢？因为在有SLUB DEBUG功能的时候，并没有检测left oob功能。这种转换是后续一个补丁的修改。补丁就是为了增加left oob检测功能。

做了转换之后的red\_left\_pad就可以检测left oob。检测的方法和Red zone区域一样，填充的magic num也一样，差别只是检测的区域不一样而已。

## 4. SLUB DEBUG原理

经过上一节分析应该很清楚了大概的原理了。从high level考虑，SLUB就是利用特殊区域填充特殊的magic num，在每一次alloc/free的时候检查magic num是否被意外修改。

4.1. magic num

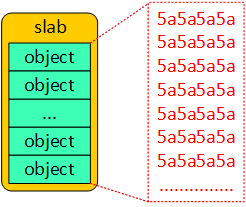
SLUB 中有哪些magic num呢?所有使用的magic num都宏定义在include/linux/poison.h文件。

1. #define SLUB\_RED\_INACTIVE 0xbb
2. #define SLUB\_RED\_ACTIVE 0xcc
3. /\* ...and for poisoning \*/
4. #define POISON\_INUSE 0x5a /\* for use-uninitialised poisoning \*/
5. #define POISON\_FREE 0x6b /\* for use-after-free poisoning \*/
6. #define POISON\_END 0xa5 /\* end-byte of poisoning \*/

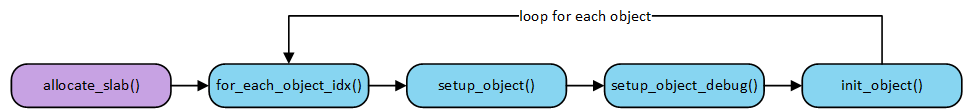
SLUB\_RED\_INACTIVE和SLUB\_RED\_ACTIVE用来填充Red zone和red\_left\_pad，目的是检测oob。POISON\_INUSE用来填充padding区域，同样可以用来检测oob，只不过是poison overwrite。POISON\_FREE作用是检测use-after-free问题。POISON\_END是object可用区域的最后一个字节填充。

4.2. slab缓存池填充

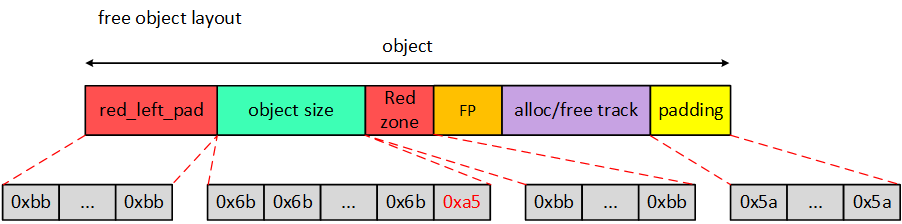
当SLUB allocator申请一块内存作为slab 缓存池的时候，会将整块内存填充POISON\_INUSE。如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/7b6f1519307828.png)

然后通过init\_object()函数将相关的区域填充成free object的情况，并且建立单链表。注意freelist指针指向的位置，SLUB\_DEBUG on和off的情况下是不一样的。主要就是3.3节提到的转换关系。为什么这里填充成free object的情况呢？其实就是为了假装我这里都是free的object，也是符合情理的。object初始化流程如下。

[](http://www.wowotech.net/content/uploadfile/201802/d6421519307830.png)  
4.3. free object layout

刚分配slab缓存池和free object之后，系统都会通过调用init\_object()函数初始化object各个区域，主要是填充magic num。free object layout如下图所示。

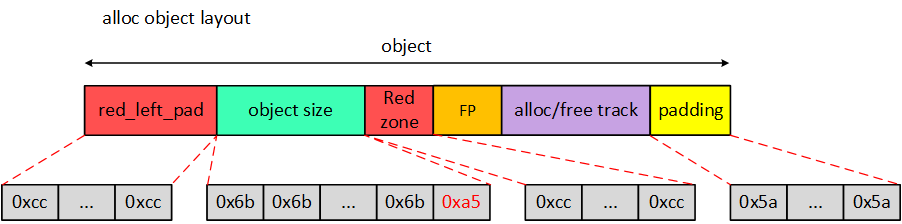
[](http://www.wowotech.net/content/uploadfile/201802/1e411519307832.png)  
1) red\_left\_pad和Red zone填充了SLUB\_RED\_INACTIVE（0xbb）；

2) object填充了POISON\_FREE（0x6b），但是最后一个byte填充POISON\_END（0xa5）；

3) padding在allocate\_slab的时候就已经被填充POISON\_INUSE（0x5a），如果程序意外改变，当检测到padding被改变的时候，会output error syslog并继续填充0x5a。

**4.4. alloc object layout**

当从SLUB allocator申请一个object时，系统同样会调用init\_object()初始化成想要的模样。alloc object layout如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/c9ba1519307834.png)

1) red\_left\_pad和Red zone填充了SLUB\_RED\_ACTIVE（0xcc）；

2) object填充了POISON\_FREE（0x6b），但是最后一个byte填充POISON\_END（0xa5）；

3) padding在allocate\_slab的时候就已经被填充POISON\_INUSE（0x5a），如果程序意外改变，当检测到padding被改变的时候，会output error syslog并继续填充0x5a。

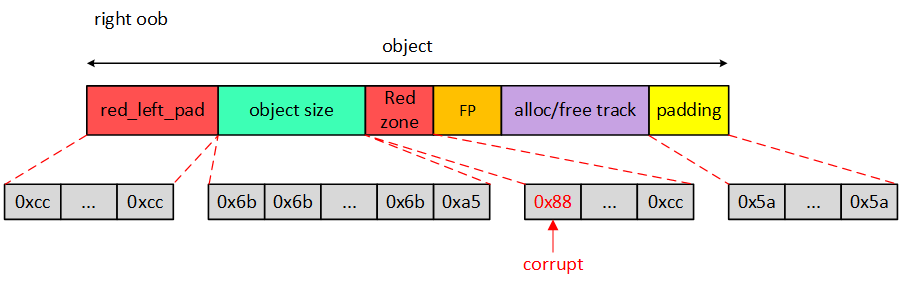
alloc object layout和free object layout相比较而言，也仅仅是red\_left\_pad和Red zone的不同。既然该填充的数据都搞定了，下面就是如何检查oob、use-after-free等问题了。

4.5. out-of-bounds bugs detect

下面使用demo例程来说明oob检测。我们使用kmalloc分配32 bytes内存，然后制造越界访问第33个元素，必然会越界访问。由于kmalloc是基于SLUB allocator，因此此bug可以检测。

1. **void** right\_oob(**void**)
2. {
3. **char** \*p = kmalloc(32, GFP\_KERNEL);
4. **if** (!p)
5. **return**;
6. p[32] = 0x88;
7. kfree(p);
8. }

运行后的object layout如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/88391519307837.png)

我们可以看到，Red zone区域本来应该0xcc的地方被修改成了0x88。很明显这是一个Redzone overwritten问题。那么系统什么时候会检测到这个严重的bug呢？就在你kfree()之后。kfree()中会去检测释放的object中各个区域的值是否valid。Red zone区域的值全是0xcc就是valid，因此这里会检测0x88不是0xcc，进而输出error syslog。kfree()最终会调用free\_consistency\_checks()检测object。free\_consistency\_checks()函数如下。

1. **static** **inline** int free\_consistency\_checks(**struct** kmem\_cache \*s,
2. **struct** page \*page, **void** \*object, **unsigned** **long** addr)
3. {
4. **if** (!check\_valid\_pointer(s, page, object)) {
5. slab\_err(s, page, "Invalid object pointer 0x%p", object);
6. **return** 0;
7. }
8. **if** (on\_freelist(s, page, object)) {
9. object\_err(s, page, object, "Object already free");
10. **return** 0;
11. }
12. **if** (!check\_object(s, page, object, SLUB\_RED\_ACTIVE))
13. **return** 0;
14. **return** 1;
15. }

1) check\_valid\_pointer()负责检测object的free pointer指针数据是否valid。oob是有可能导致这种情况得到发生。

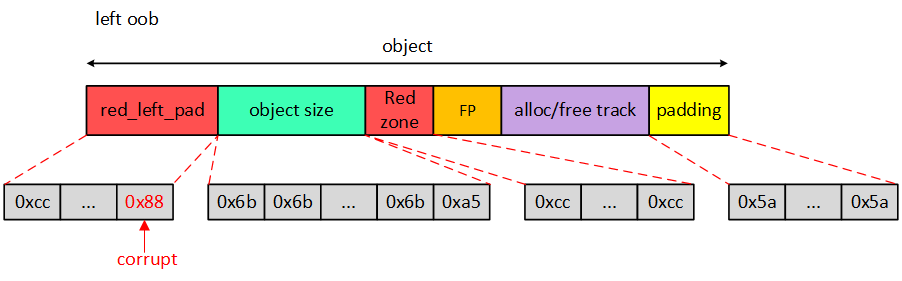
2) on\_freelist()检测object是否已经free，可以检测多次free的bug。

3) check\_object()会检测Red zone区域的数值是否被改变，因此这里就会报出bug。

如果是左边界越界访问，是否也同样可以检测出呢？可以测试以下demo例程。

1. **void** left\_oob(**void**)
2. {
3. **char** \*p = kmalloc(32, GFP\_KERNEL);
4. **if** (!p)
5. **return**;
6. p[-1] = 0x88;
7. kfree(p);
8. }

运行后的object layout如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/ba6b1519307842.png)

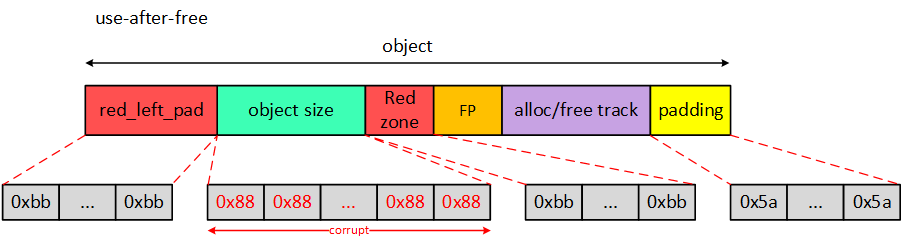
检测方法大同小异，这里也是最终在free\_consistency\_checks()函数中通过检测red\_left\_pad区域发现left oob问题。

可能你会想如果我只申请内存不释放的话，这个bug还能检测到吗？其实这里是不行的。我们只能借助slabinfo工具主动触发检测功能。所以，这也是SLUB DEBUG的一个劣势，它不能做到动态监测。它的检测机制是被动的。

4.6. use-after-free bugs detect

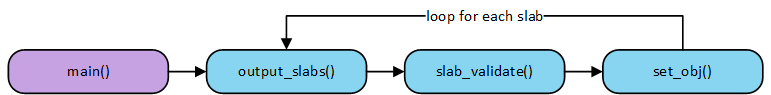
如果是use-after-free问题，我们该如何检测呢？首先上demo例程。

1. **void** use\_after\_free(**void**)
2. {
3. **char** \*p = kmalloc(32, GFP\_KERNEL);
4. **if** (!p)
5. **return**;
7. kfree(p);
8. memset(p, 0x88, 32);
9. }

运行之后object layout如下图所示。  
 [](http://www.wowotech.net/content/uploadfile/201802/079f1519307843.png)  
还记得上面说的吗？SLUB DEBUG是被动的。因此这里就要选择slabinfo工具了。命令中断输入slabinfo –v即可。slabinfo检测的原理也很简单，便利所有已经释放的object，检查object区域是否全是0x6b（最后一个字节oxa5）即可，如果不是的话，自然就是use-after-free。

## 5. slabinfo

我们看一下slabinfo –v命令的实现方式以及检查的流程。slabinfo源码位于tools/vm/slabinfo.c文件。slabinfo –v命令执行流程如下图所示。

[](http://www.wowotech.net/content/uploadfile/201802/71341519307845.png)  
针对系统中每一个slab都会执行set\_obj()函数。set\_obj()代码如下：

1. **static** **void** set\_obj(**struct** slabinfo \*s, **const** **char** \*name, int n)
2. {
3. **char** x[100];
4. FILE \*f;
5. snprintf(x, 100, "%s/%s", s->name, name);
6. f = fopen(x, "w");
7. **if** (!f)
8. fatal("Cannot write to %s\n", x);
9. fprintf(f, "%d\n", n);
10. fclose(f);
11. }

set\_obj()参数name传递的是“validate”，n传递的是1。作用就是向/sys/kernel/slab/<slab name>/ validate节点写“1”触发slab检测功能。根据validate节点找到写入口函数validate\_store()。调用函数执行流如下图所示。  
 [22.png](http://www.wowotech.net/content/uploadfile/201802/75c11519307847.png)

validate\_slab()代码如下：

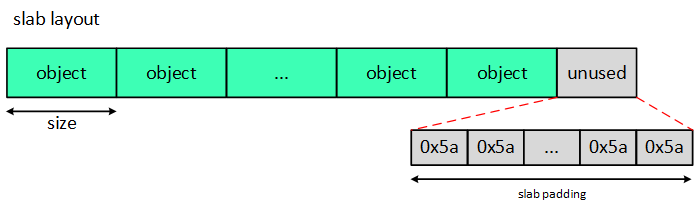
1. **static** int validate\_slab(**struct** kmem\_cache \*s, **struct** page \*page, **unsigned** **long** \*map)
2. {
3. **void** \*p;
4. **void** \*addr = page\_address(page);
5. **if** (!check\_slab(s, page) ||
6. !on\_freelist(s, page, NULL))
7. **return** 0;
8. /\* Now we know that a valid freelist exists \*/
9. bitmap\_zero(map, page->objects);
10. get\_map(s, page, map);
11. for\_each\_object(p, s, addr, page->objects) {
12. **if** (test\_bit(slab\_index(p, s, addr), map))
13. **if** (!check\_object(s, page, p, SLUB\_RED\_INACTIVE))
14. **return** 0;
15. }
16. for\_each\_object(p, s, addr, page->objects)
17. **if** (!test\_bit(slab\_index(p, s, addr), map))
18. **if** (!check\_object(s, page, p, SLUB\_RED\_ACTIVE))
19. **return** 0;
20. **return** 1;
21. }

1) check\_slab()会调用slab\_pad\_check()检查slab padding区域。slab padding和object里面的pading不是一回事。如果说从buddy system分配的页按照SLUB规则平分成很多object，那么有可能不能整除，那么剩下的unused区域就是slab padding。valid的数值是0x5a。如下图所示。

2) get\_map()利用bitmap标记所有的available object。例如，slab缓存池一共有10个对象，按地址大小排序标号0-9（相当于index）。假设5和8号object已经被分配出去。那么bitmap中除了bit5和bit8以为，其余位为1。

3) 第一个for循环历所有的available object是否有oob、use-after-free、object padding overwritten等问题发生。

4) 第二个for循环遍历所有已经分配出去的object是否发生oob问题。

[](http://www.wowotech.net/content/uploadfile/201802/7ae51519307847.png)