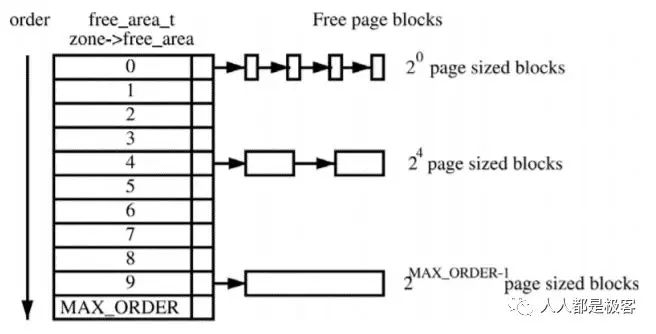
**Linux内核内存管理算法Buddy和Slab**

有了前两节的学习相信读者已经知道CPU所有的操作都是建立在虚拟地址上处理(这里的虚拟地址分为内核态虚拟地址和用户态虚拟地址)，CPU看到的内存管理都是对page的管理，接下来我们看一下用来管理page的经典算法--Buddy。

Buddy分配算法

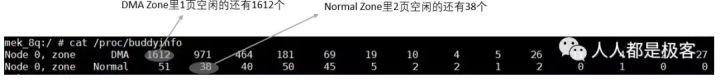


假设这是一段连续的页框，阴影部分表示已经被使用的页框，现在需要申请一个连续的5个页框。这个时候，在这段内存上不能找到连续的5个空闲的页框，就会去另一段内存上去寻找5个连续的页框，这样子，久而久之就形成了页框的浪费。为了避免出现这种情况，Linux内核中引入了伙伴系统算法(Buddy system)。把所有的空闲页框分组为11个块链表，每个块链表分别包含大小为1，2，4，8，16，32，64，128，256，512和1024个连续页框的页框块。最大可以申请1024个连续页框，对应4MB大小的连续内存。每个页框块的第一个页框的物理地址是该块大小的整数倍，如图：

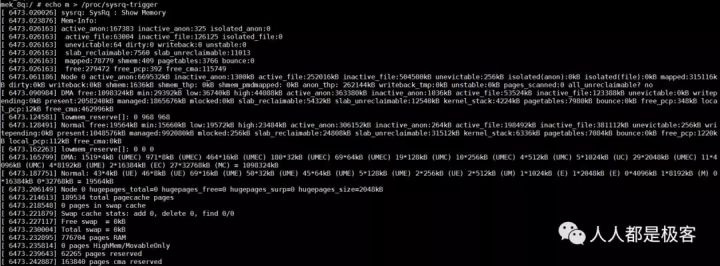


假设要申请一个256个页框的块，先从256个页框的链表中查找空闲块，如果没有，就去512个页框的链表中找，找到了则将页框块分为2个256个页框的块，一个分配给应用，另外一个移到256个页框的链表中。如果512个页框的链表中仍没有空闲块，继续向1024个页框的链表查找，如果仍然没有，则返回错误。页框块在释放时，会主动将两个连续的页框块合并为一个较大的页框块。

从上面可以知道Buddy算法一直在对页框做拆开合并拆开合并的动作。Buddy算法牛逼就牛逼在运用了世界上任何正整数都可以由2^n的和组成。这也是Buddy算法管理空闲页表的本质。 空闲内存的信息我们可以通过以下命令获取：



也可以通过echo m > /proc/sysrq-trigger来观察buddy状态，与/proc/buddyinfo的信息是一致的：



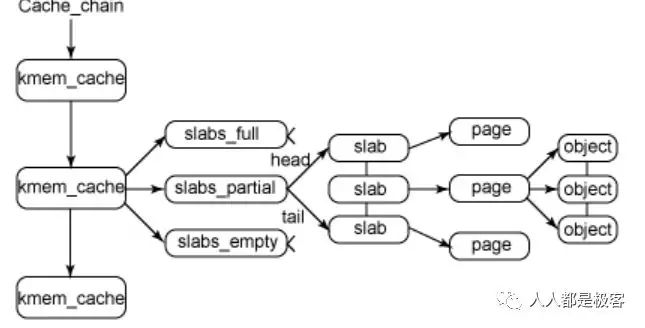
**CMA**

细心的读者或许会发现当Buddy算法对内存拆拆合合的过程中会造成碎片化的现象，以至于内存后来没有了大块的连续内存，全是小块内存。当然这对应用程序是不影响的(前面我们讲过用页表可以把不连续的物理地址在虚拟地址上连续起来)，但是内核态就没有办法获取大块连续的内存（比如DMA, Camera, GPU都需要大块物理地址连续的内存）。

在嵌入式设备中一般用CMA来解决上述的问题。CMA的全称是contiguous memory allocator， 其工作原理是：预留一段的内存给驱动使用，但当驱动不用的时候，CMA区域可以分配给用户进程用作匿名内存或者页缓存。而当驱动需要使用时，就将进程占用的内存通过回收或者迁移的方式将之前占用的预留内存腾出来，供驱动使用。

**Slab**

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。我们先来看一张图：



kmem\_cache是一个cache\_chain的链表，描述了一个高速缓存，每个高速缓存包含了一个slabs的列表，这通常是一段连续的内存块。存在3种slab：

slabs\_full(完全分配的slab)

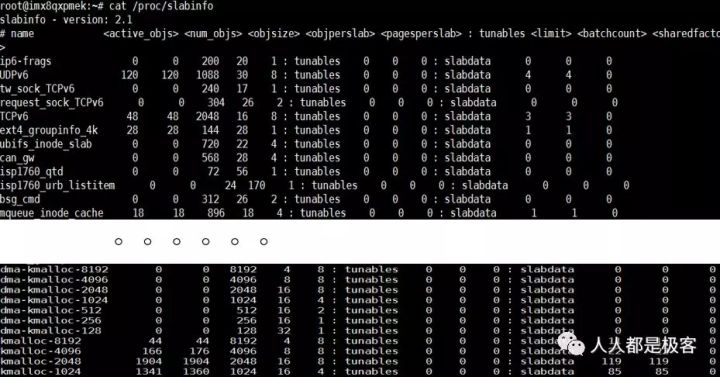
slabs\_partial(部分分配的slab)

slabs\_empty(空slab,或者没有对象被分配)。

slab是slab分配器的最小单位，在实现上一个slab有一个货多个连续的物理页组成（通常只有一页）。单个slab可以在slab链表之间移动，例如如果一个半满slab被分配了对象后变满了，就要从slabs\_partial中被删除，同时插入到slabs\_full中去。

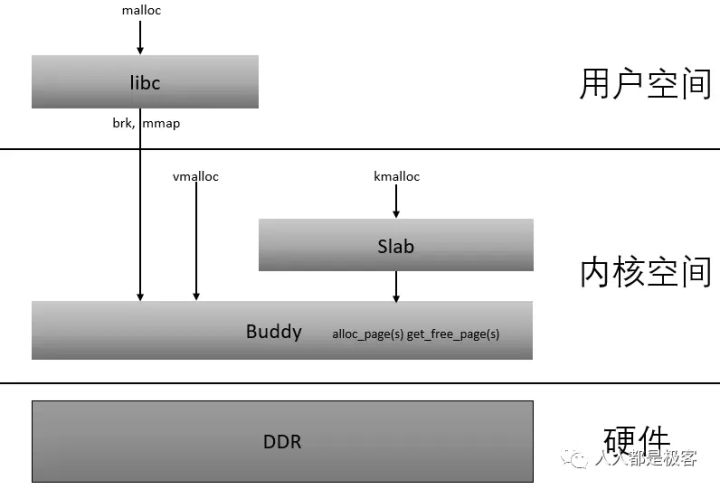
为了进一步解释，这里举个例子来说明，用struct kmem\_cache结构描述的一段内存就称作一个slab缓存池。一个slab缓存池就像是一箱牛奶，一箱牛奶中有很多瓶牛奶，每瓶牛奶就是一个object。分配内存的时候，就相当于从牛奶箱中拿一瓶。总有拿完的一天。当箱子空的时候，你就需要去超市再买一箱回来。超市就相当于partial链表，超市存储着很多箱牛奶。如果超市也卖完了，自然就要从厂家进货，然后出售给你。厂家就相当于伙伴系统。

可以通过下面命令查看slab缓存的信息：



总结

从内存DDR分为不同的ZONE，到CPU访问的Page通过页表来映射ZONE，再到通过Buddy算法和Slab算法对这些Page进行管理，我们应该可以从感官的角度理解了下图：



# 伙伴系统和slab内存分配机制详解

当在用户模式下运行进程请求额外内存时，从内核维护的空闲页帧列表上分配页面。这个列表通常使用页面置换算法来填充，如前所述，它很可能包含散布在物理内存中的空闲页面。也要记住，如果用户进程请求单个字节内存，那么就会导致内部碎片，因为进程会得到整个帧。

用于分配内核内存的空闲内存池通常不同于用于普通用户模式进程的列表。这有两个主要原因：

内核需要为不同大小的数据结构请求内存，其中有的小于一页。因此，内核应保守地使用内存，并努力最小化碎片浪费。这一点非常重要，因为许多操作系统的内核代码或数据不受调页系统的控制。

用户模式进程分配的页面不必位于连续物理内存。然而，有的硬件设备与物理内存直接交互，即无法享有虚拟内存接口带来的便利，因而可能要求内存常驻在连续物理内存中。

下面讨论两个策略，以便管理用于内核进程的空闲内存：“伙伴系统”和 slab 分配。

伙伴系统

伙伴系统从物理连续的大小固定的段上进行分配。从这个段上分配内存，采用 2 的幂分配器来满足请求分配单元的大小为 2 的幂（4KB、 8KB、16KB 等）。请求单元的大小如不适当，就圆整到下一个更大的 2 的幂。例如，如果请求大小为 11KB，则按 16KB 的段来请求。

让我们考虑一个简单例子。假设内存段的大小最初为 256KB，内核请求 21KB 的内存。最初，这个段分为两个伙伴，称为 AL 和 AR，每个的大小都为 128KB；这两个伙伴之一进一步分成两个 64KB 的伙伴，即 BL 和 BR。然而，从 21KB 开始的下一个大的 2 的幂是 32KB，因此 BL 或 BR 再次划分为两个 32KB 的伙伴 CL 和 CR。因此，其中一个 32KB 的段可用于满足 21KB 请求。这种方案如图 1 所示，其中 CL 段是分配给 21KB 请求的。

伙伴系统分配

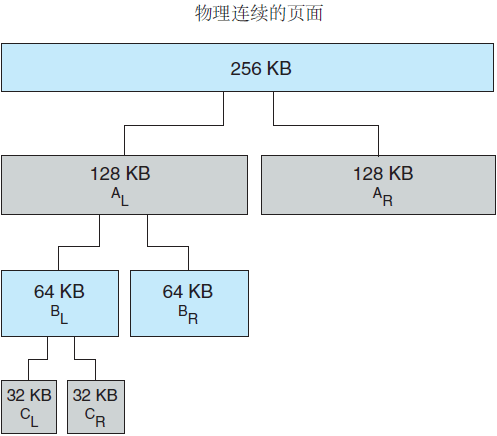


图 1 伙伴系统分配

伙伴系统的一个优点是：通过称为合并的技术，可以将相邻伙伴快速组合以形成更大分段。例如，在图 1 中，当内核释放已被分配的 CL 时，系统可以将 CL 和 CR 合并成 64KB 的段。段 BL 继而可以与伙伴 BR 合并，以形成 128KB 段。最终，可以得到原来的 256KB 段。

伙伴系统的明显缺点是：由于圆整到下一个 2 的幂，很可能造成分配段内的碎片。例如，33KB 的内存请求只能使用 64KB 段来满足。事实上，我们不能保证因内部碎片而浪费的单元一定少于 50%。

slab分配

分配内核内存的第二种策略称为slab分配。每个 slab 由一个或多个物理连续的页面组成，每个 cache 由一个或多个 slab 组成，每个内核数据结构都有一个 cache。

例如，用于表示进程描述符、文件对象、信号量等的数据结构都有各自单独的 cache。每个 cache 含有内核数据结构的对象实例（称为 object）。例如，信号量 cache 有信号量对象，进程描述符 cache 有进程描述符对象，等等。

slab分配

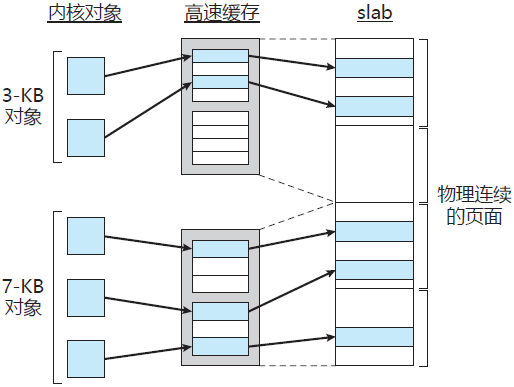


图 2 slab 分配

图 2 显示了 slab、cache 及 object 三者之间的关系。该图显示了 2 个大小为 3KB 的内核对象和 3 个大小为 7KB 的对象，它们位于各自的 cache 中。

slab 分配算法采用 cache 来存储内核对象。在创建 cache 时，若干起初标记为 free 的对象被分配到 cache。cache 内的对象数量取决于相关 slab 的大小。例如，12KB slab（由 3 个连续的 4KB 页面组成）可以存储 6 个 2KB 对象。最初，cache 内的所有对象都标记为空闲。当需要内核数据结构的新对象时，分配器可以从 cache 上分配任何空闲对象以便满足请求。从 cache 上分配的对象标记为 used（使用）。

让我们考虑一个场景，这里内核为表示进程描述符的对象从 slab 分配器请求内存。在 Linux 系统中，进程描述符属于 struct task\_struct 类型，它需要大约 1.7KB 的内存。当 Linux 内核创建一个新任务时，它从 cache 中请求 struct task\_struct 对象的必要内存。cache 利用已经在 slab 中分配的并且标记为 free (空闲）的 struct task\_struct 对象来满足请求。

在 Linux 中，slab 可以处于三种可能状态之一：

**满的：slab 的所有对象标记为使用。**

**空的：slab 上的所有对象标记为空闲。**

**部分：slab 上的对象有的标记为使用，有的标记为空闲。**

slab 分配器首先尝试在部分为空的 slab 中用空闲对象来满足请求。如果不存在，则从空的 slab 中分配空闲对象。如果没有空的 slab 可用，则从连续物理页面分配新的 slab，并将其分配给 cache；从这个 slab 上，再分配对象内存。

slab 分配器提供两个主要优点：

没有因碎片而引起内存浪费。碎片不是问题，因为每个内核数据结构都有关联的 cache，每个 cache 都由一个或多个 slab 组成，而 slab 按所表示对象的大小来分块。因此，当内核请求对象内存时，slab 分配器可以返回刚好表示对象的所需内存。

可以快速满足内存请求。因此，当对象频繁地被分配和释放时，如来自内核请求的情况，slab 分配方案在管理内存时特别有效。分配和释放内存的动作可能是一个耗时过程。然而，由于对象已预先创建，因此可以从 cache 中快速分配。再者，当内核用完对象并释放它时，它被标记为空闲并返回到 cache，从而立即可用于后续的内核请求。

slab 分配器首先出现在 Solaris 2.4 内核中。由于通用性质，Solaris 现在也将这种分配器用于某些用户模式的内存请求。最初，Linux 使用的是伙伴系统；然而，从版本 2.2 开始，Linux 内核采用 slab 分配器。

现在，最近发布的 Linux 也包括另外两个内核内存分配器，SLOB 和 SLUB 分配器（Linux 将 slab 实现称为 SLAB）。

简单块列表（SLOB）分配器用于有限内存的系统，例如嵌入式系统。SLOB 工作采用 3 个对象列表：小（用于小于 256 字节的对象）、中（用于小于 1024 字节的对象）和大（用于小于页面大小的对象）。内存请求采用首先适应策略，从适当大小的列表上分配对象。

从版本 2.6.24 开始，SLUB 分配器取代 SLAB，成为 Linux 内核的默认分配器。SLUB 通过减少 SLAB 分配器所需的大量开销，来解决 slab 分配的性能问题，一个改变是，在 SLAB 分配下每个 slab 存储的元数据，移到 Linux 内核用于每个页面的结构 page。此外，对于 SLAB 分配器，每个 CPU 都有队列以维护每个 cache 内的对象，SLUB 会删除这些队列。

对于具有大量处理器的系统，分配给这些队列的内存量是很重要的。因此，随着系统处理器数量的增加，SLUB 性能也更好。