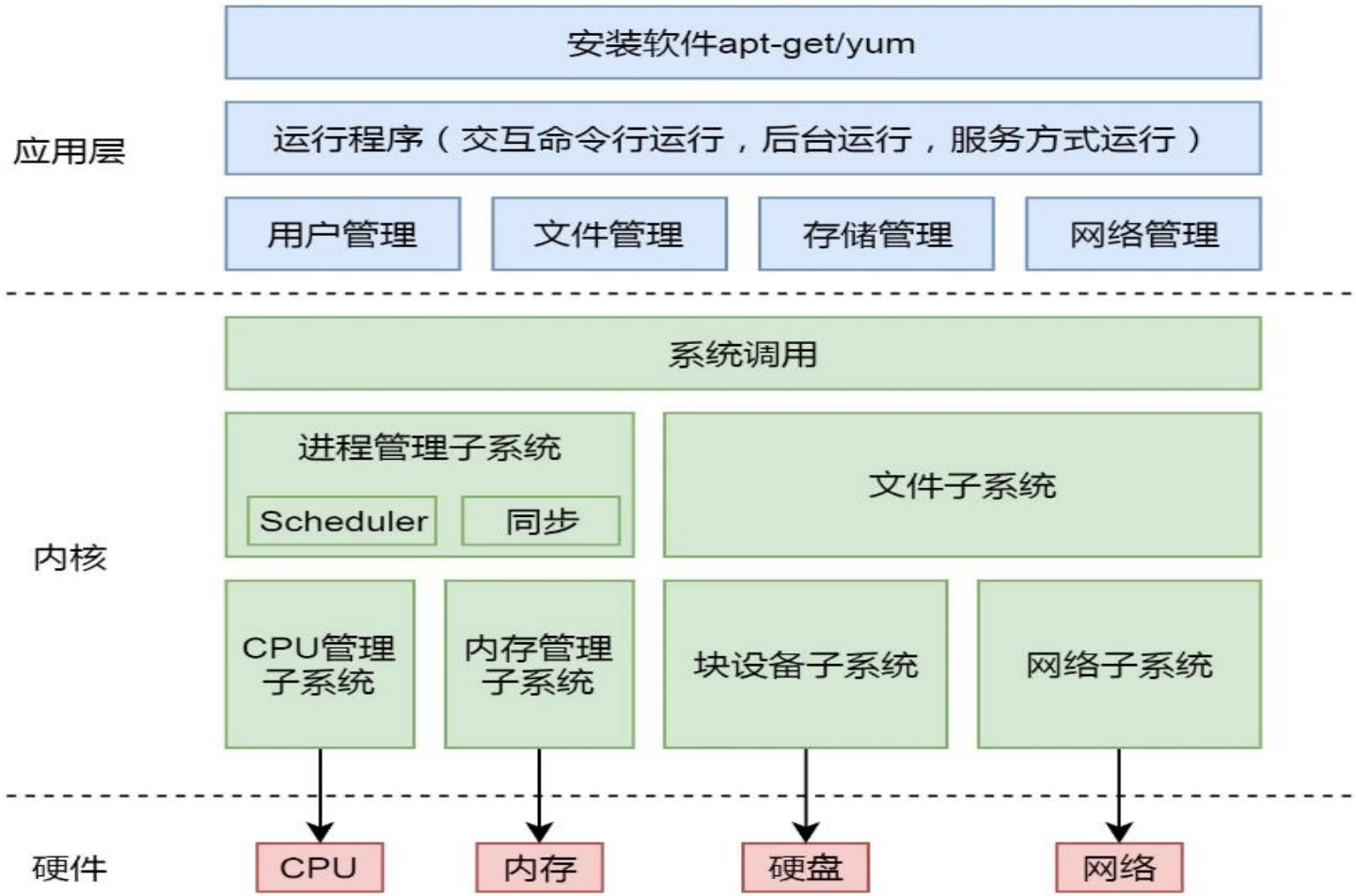
深入理解Linux内存子系统

By 公众号@极客重生

操作系统宏观视角



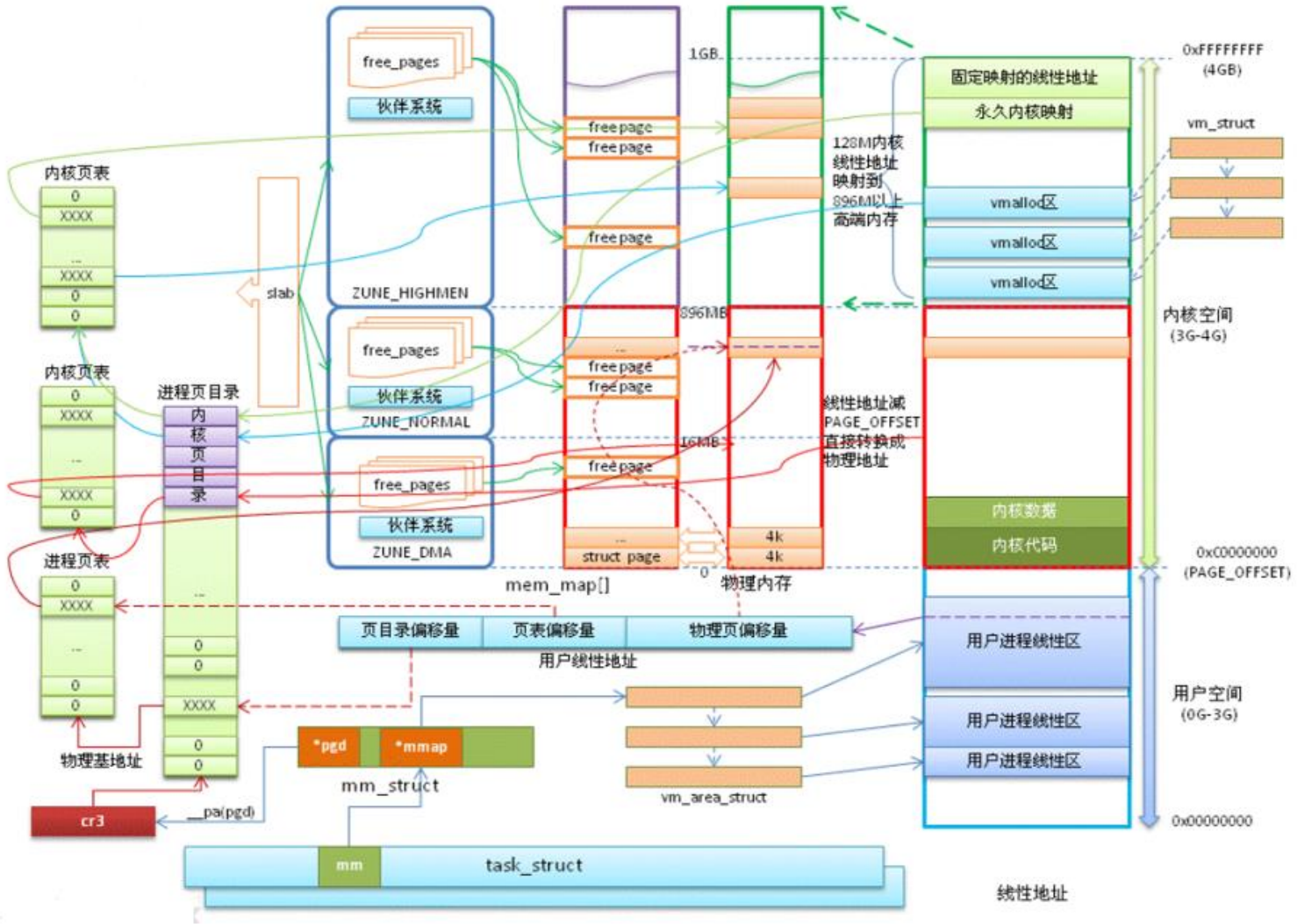
Linux操作系统

操作系统各大子系统

云计算开发	IaaS：虚拟化，计算，网络，存储 PaaS：虚拟化，容器，计算，网络，存储
Linux后台服务器开发	高性能，高并发
Linux嵌入式开发	内核裁剪，各种硬件接口适配，性能优化，手机，Iot
Linux SRE方向	运维，解决Linux稳定性问题。

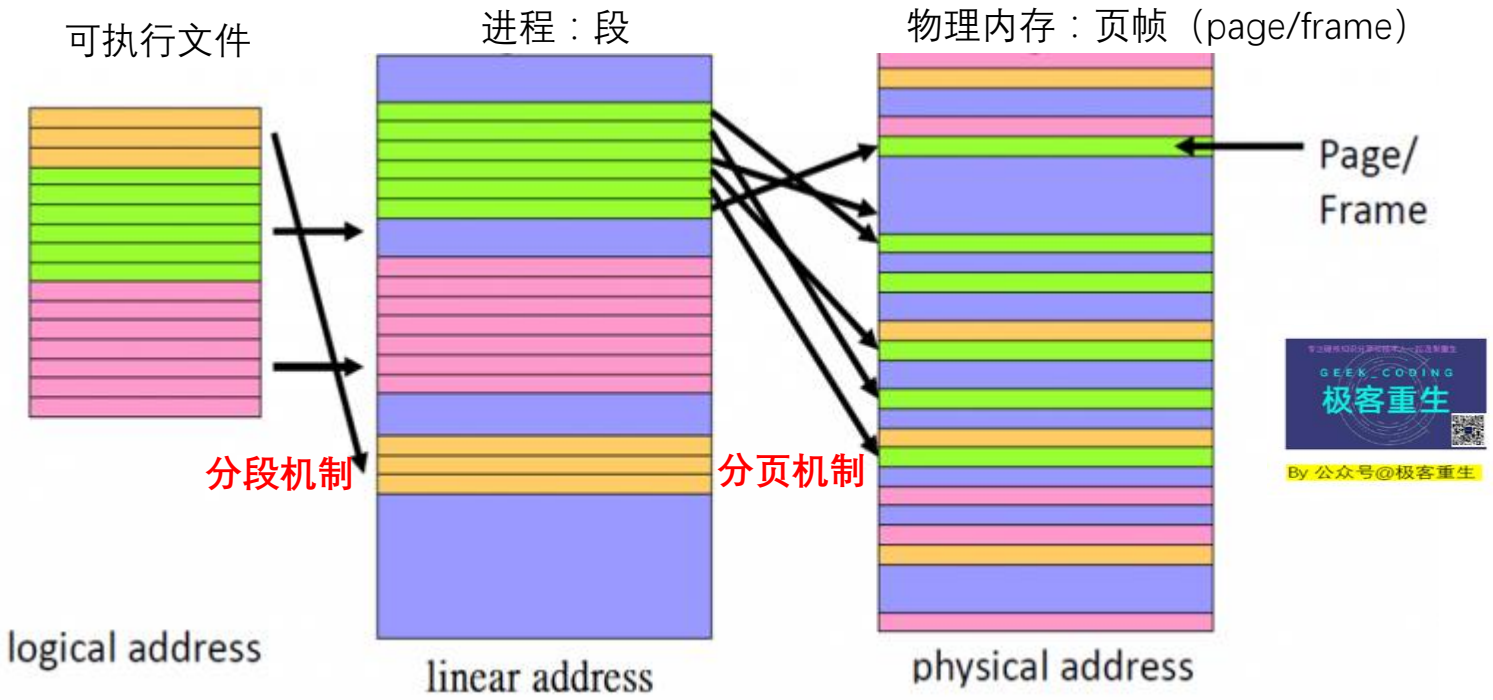
- 调度子系统
- 内存子系统
- IO子系统
- 网络子系统
- 驱动（设备管理）子系统

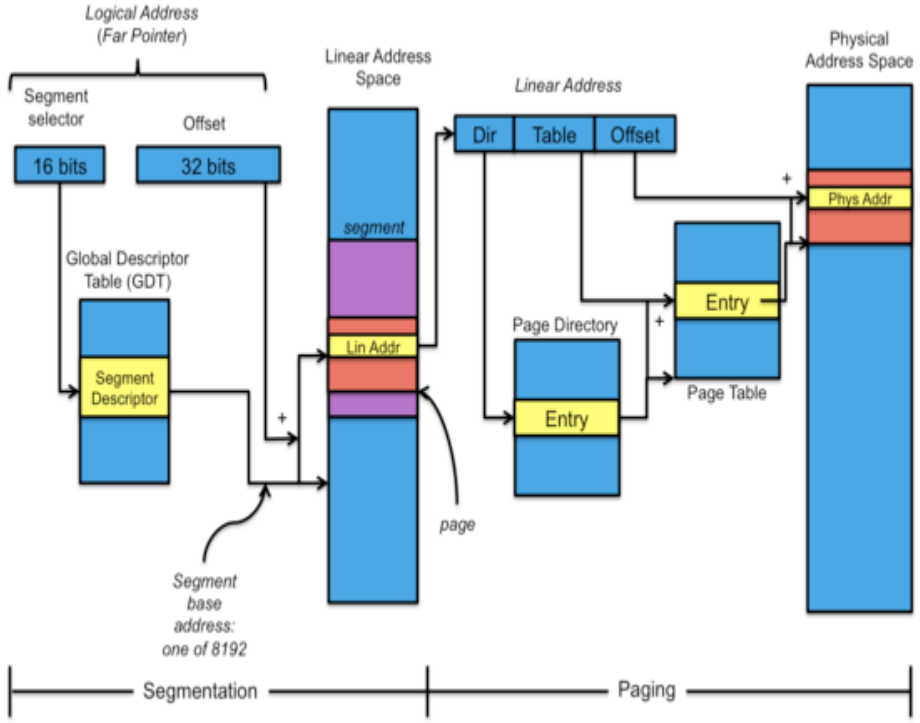
内存子系统



内存管理全景图

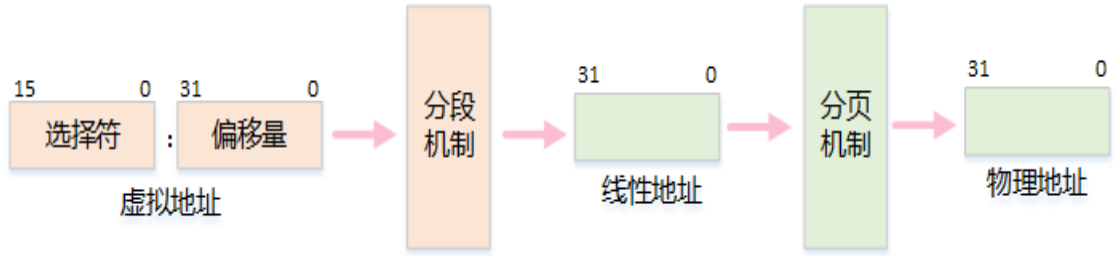
内存管理—宏观视角之内存地址





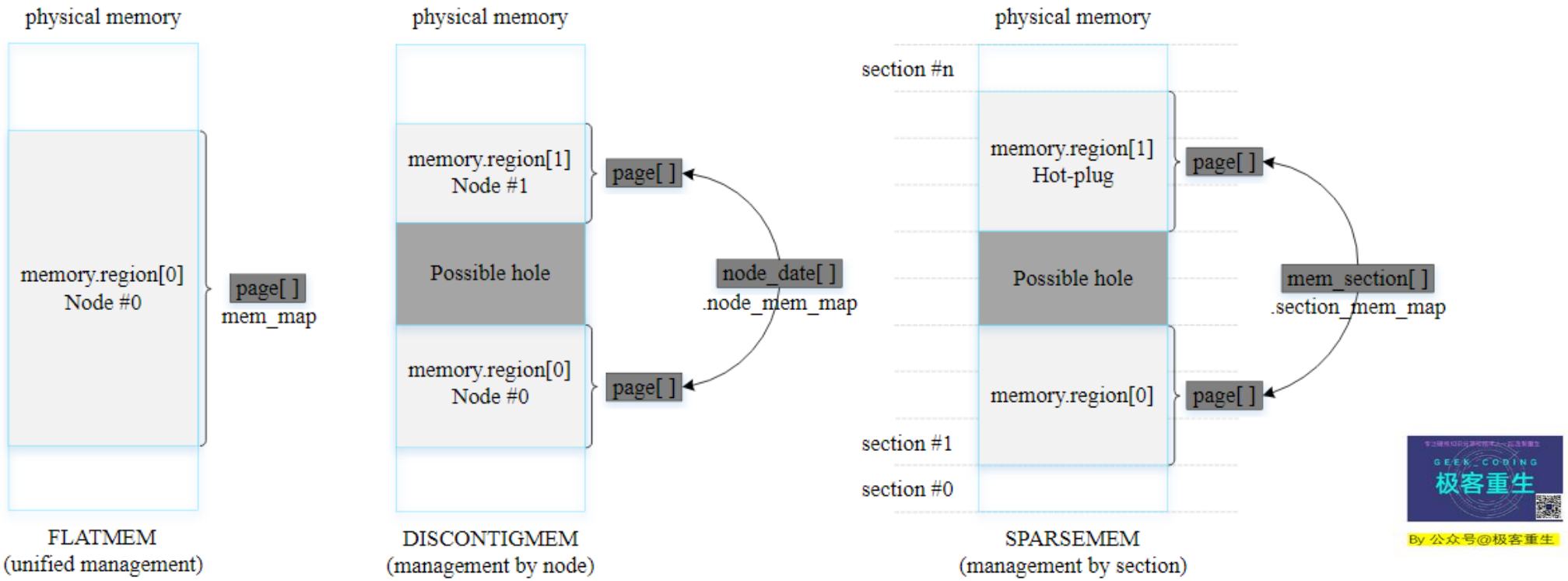
逻辑地址（程序视角）->虚拟地址（进程视角）->物理地址（硬件视角）

- **逻辑地址**：每个逻辑地址都由一个段（segment）和段内偏移量（offset）组成。编译器将程序数据分为若干段。
- **线性地址也叫虚拟地址**：操作系统抽象出来的地址为虚拟地址，由两部分组成：页和页内偏移量。
- **物理地址**：内存中实际存储单元的地址称为物理地址由于每个内存单元都有唯一的内存地址编号，因此物理地址空间是一个一维的线性空间。



虚拟化思想起源：本质是指资源的抽象化，要想资源充分利用，必须把资源最小单位化(池化)，这样上层才能按需使用资源，虚拟化不但解放了操作系统，也解放了物理硬件，大大提高了资源的利用率。

内存管理—内存模型



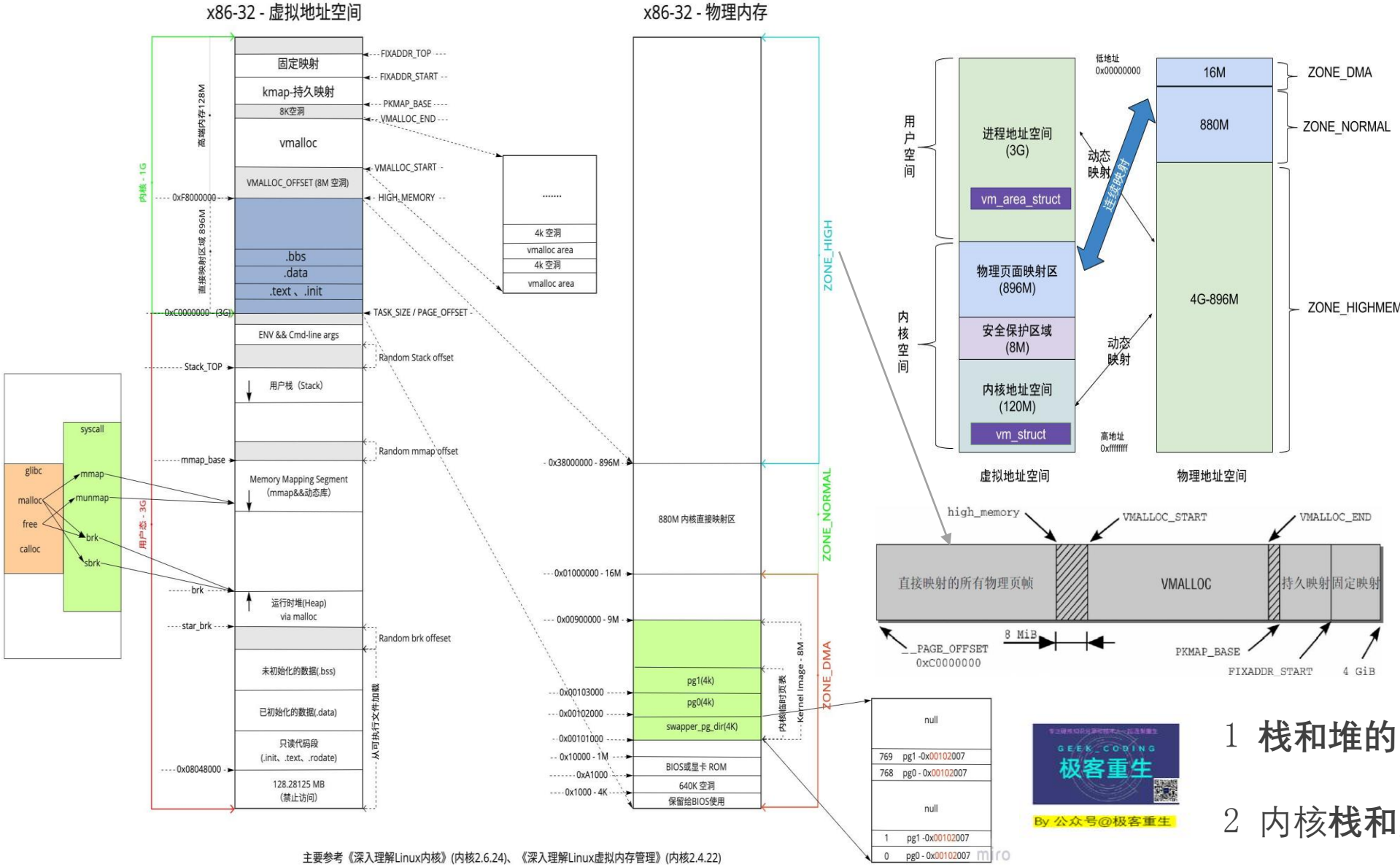
Linux内存模型发展经历三个模式：
FLATMEM（32位）->DISCONTIGMEM（32+numa）->SPARSEMEM(64位默认)

- 内存连续且不存在空隙
- 在大多数情况下，应用于UMA系统「Uniform Memory Access」

- 多个内存节点不连续并且存在空隙「hole」
- 适用于UMA系统和NUMA系统「Non-Uniform Memory Access」
- ARM在2010年已移除了对DISCONTIGMEM内存模型的支持

- 多个内存区域不连续并且存在空隙
- 支持内存热插拔「hot-plug memory」，但性能稍逊色于DISCONTIGMEM
- 在x86或ARM64内核采用了最近实现的SPARSEMEM_VMEMMAP变种，其性能比DISCONTIGMEM更优并且与FLATMEM相当
- 对于64位内核，默认选择SPARSEMEM内存模型
- 以section为单位管理online和hot-plug内存

内存管理—Linux虚拟地址空间-32位



主要参考《深入理解Linux内核》(内核2.6.24)、《深入理解Linux虚拟内存管理》(内核2.4.22)

直接映射

背景：在内核中我们往往要频繁地进行虚拟/物理地址操作，在这种情况下，快速高效的virtual to physical转换就很重要。

思路：将0xC0000000-0xFFFFFFFF的虚拟地址直接映射到0x00000000-0x3FFFFFFF，也就是将最高的1G地址全部映射到最低的1G，这样虚拟地址与物理地址之间就有固定的3G offset，每当遇到一个内核中的符号，我们需要得到其物理地址时，直接减去3G即可。

高端内存（动态映射）：

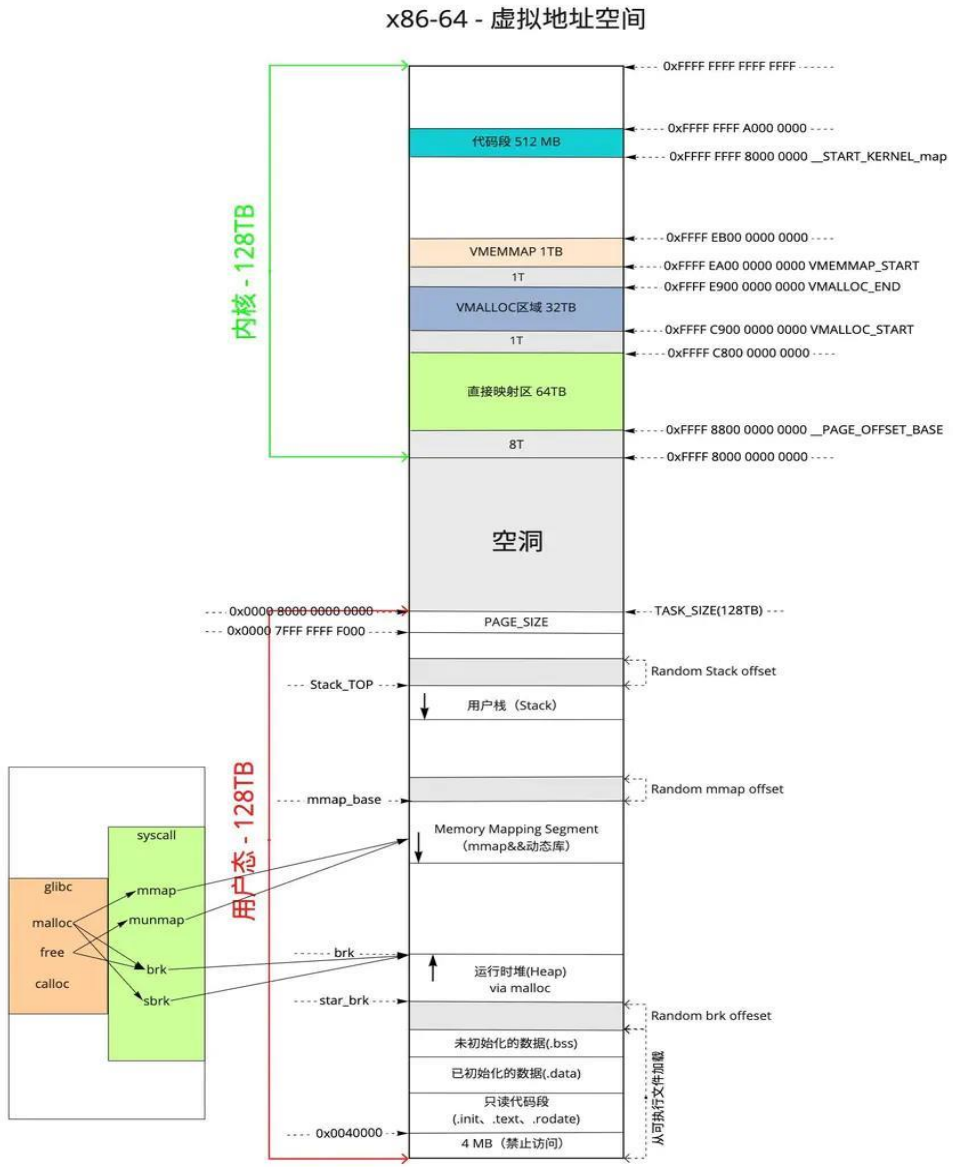
背景: 解决的是32位下**虚拟地址空间不足**带来的问题。

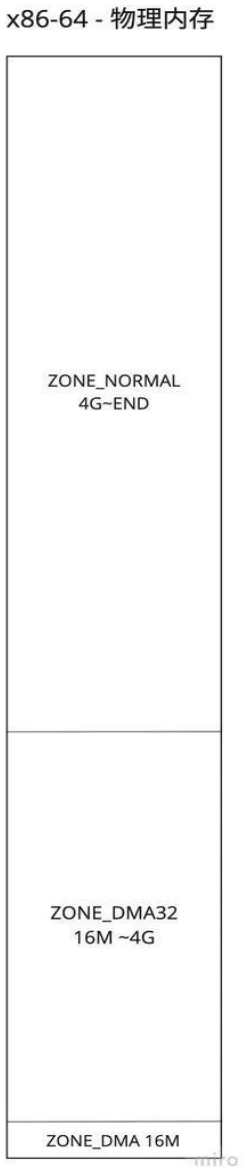
思想：借一段地址空间，建立临时地址映射，用完后释放，达到这段地址空间可以循环使用，访问所有物理内存。

1 **栈和堆的区别**

2 内核**栈和用户空间的栈区别**

内存管理—Linux虚拟地址空间-64位





内核空间

- 代码段：
- Vmemmap：
- Vmalloc：
- 直接映射区：

用户空间

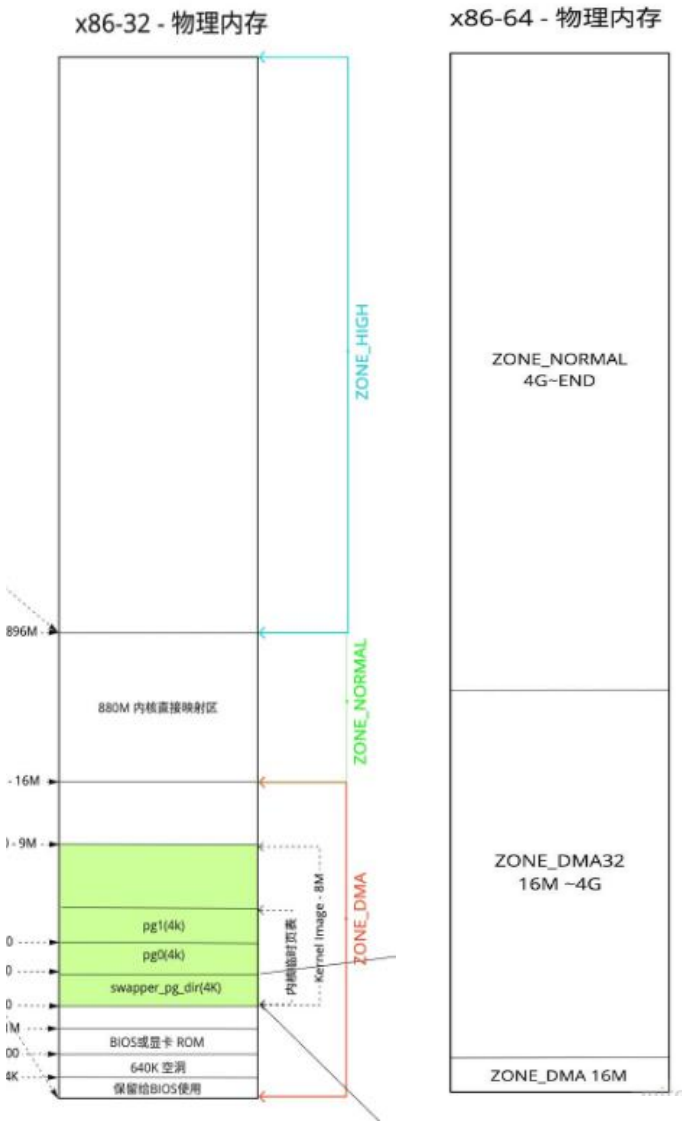
- 栈空间
- Memory mapping区
- Heap区：
- Bss区
- Data区
- 代码区

1 为什么x86-64系统的虚拟地址空间只有256TB？

2^48 = 256TB（内核 128TB， 用户态：128TB）

2 为什么64位系统没有划分ZONE_HIGHMEM区域？

内存管理—Linux 物理内存空间



内存节点(Node1)
- 内存区域（Zone1）
  - 页框(Page Frame1)(4K)
  - 页框(Page Frame2)(4K)
  - ...
- 内存区域（Zone2）
- ...

内存节点(Node2)

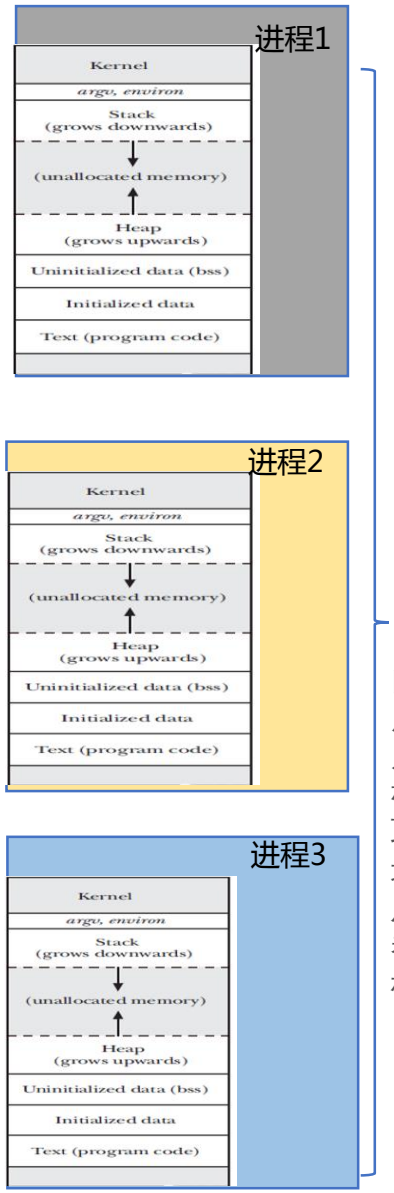
...

- **ZONE_DMA** 标记适合DMA的内存域。该区域的长度依赖于处理器类型。在IA-32计算机上，一般的限制是16 MiB，这是由古老的ISA设备强加的边界。但更现代的计算机也可能受这一限制的影响。
- **ZONE_DMA32** 标记了使用32位地址字可寻址、适合DMA的内存域。显然，只有在64位系统上，两种DMA内存域才有差别。在32位计算机上，本内存域是空的，即长度为0 MiB。在Alpha和AMD64系统上，该内存域的长度可能从0到4GiB。
- **ZONE_NORMAL**标记了可直接映射到内核段的普通内存域。这是在所有体系结构上保证都会存在的唯一内存域，但无法保证该地址范围对应了实际的物理内存。例如，如果AMD64系统有2 GiB内存，那么所有内存都属于ZONE_DMA32范围，而ZONE_NORMAL则为空

1 虚拟内存页和物理内存页有什么关系？

2 为什么要内存对齐？

内存管理-虚拟内存机制

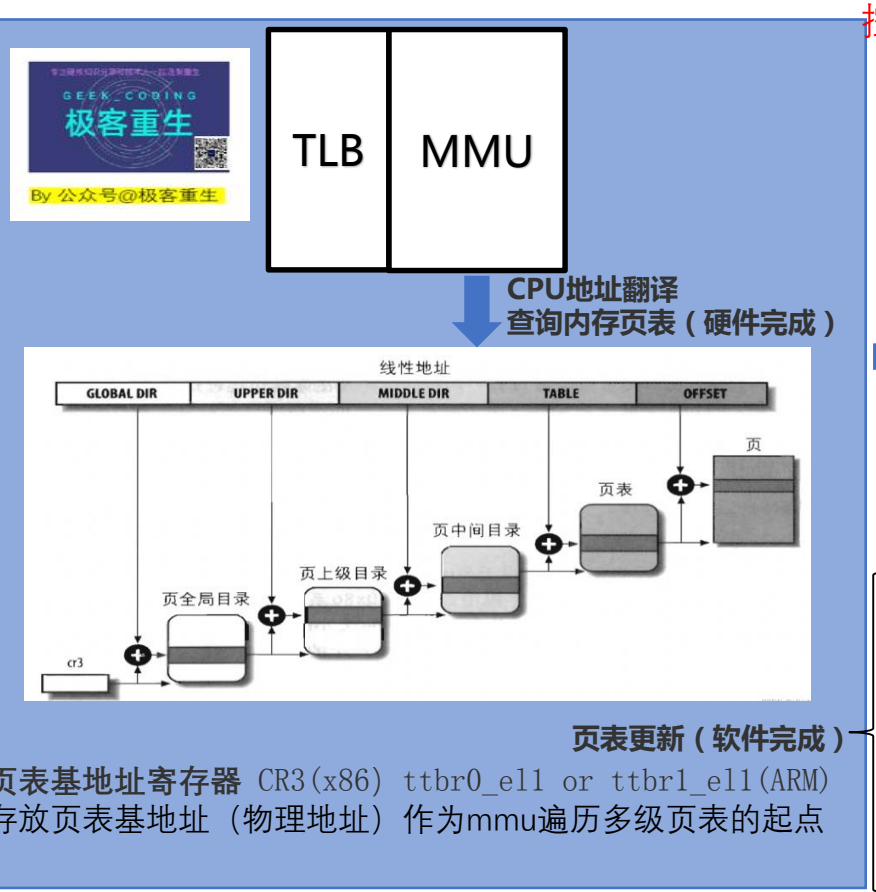


虚拟地址空间

1 为什么有MMU？
2 内核是如何访问内存的？
3 内核页表和普通的页表到底有什么区别？
4 为什么有缺页中断？
5 如何优化**页表查询？**

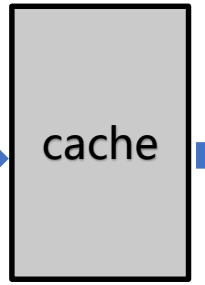
线性地址
(虚拟地址)

内核空间地址/用户空间地址，只是权限不一样而已，要获取它们的物理地址，
从**硬件层面**上，都要走MMU硬件机制。



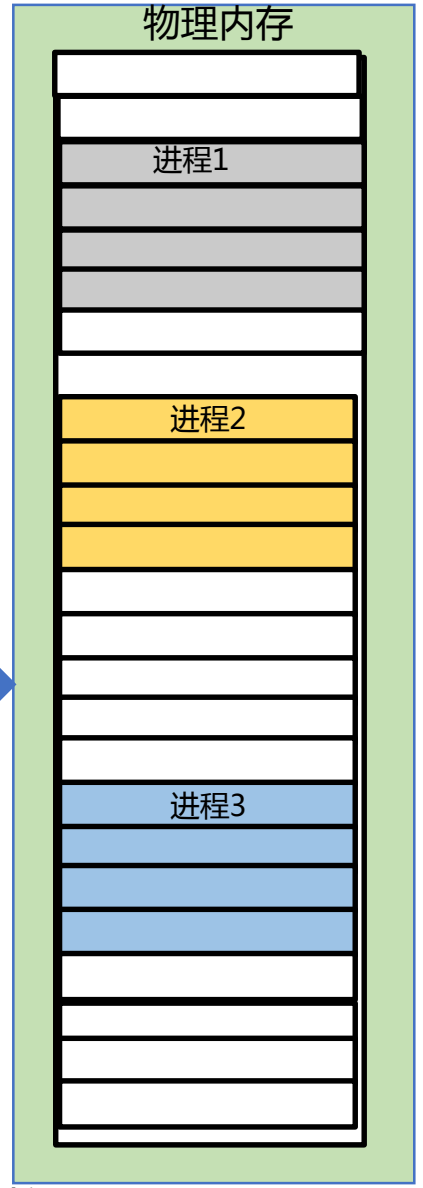
理解的核心：内存成本，物理内存共用，内核只有一个，多进程（多个虚拟空间）
页表的作用不仅仅是虚拟地址到物仅理地址的映射，还有关键的权限访问控制和页面属性的记录

物理地址



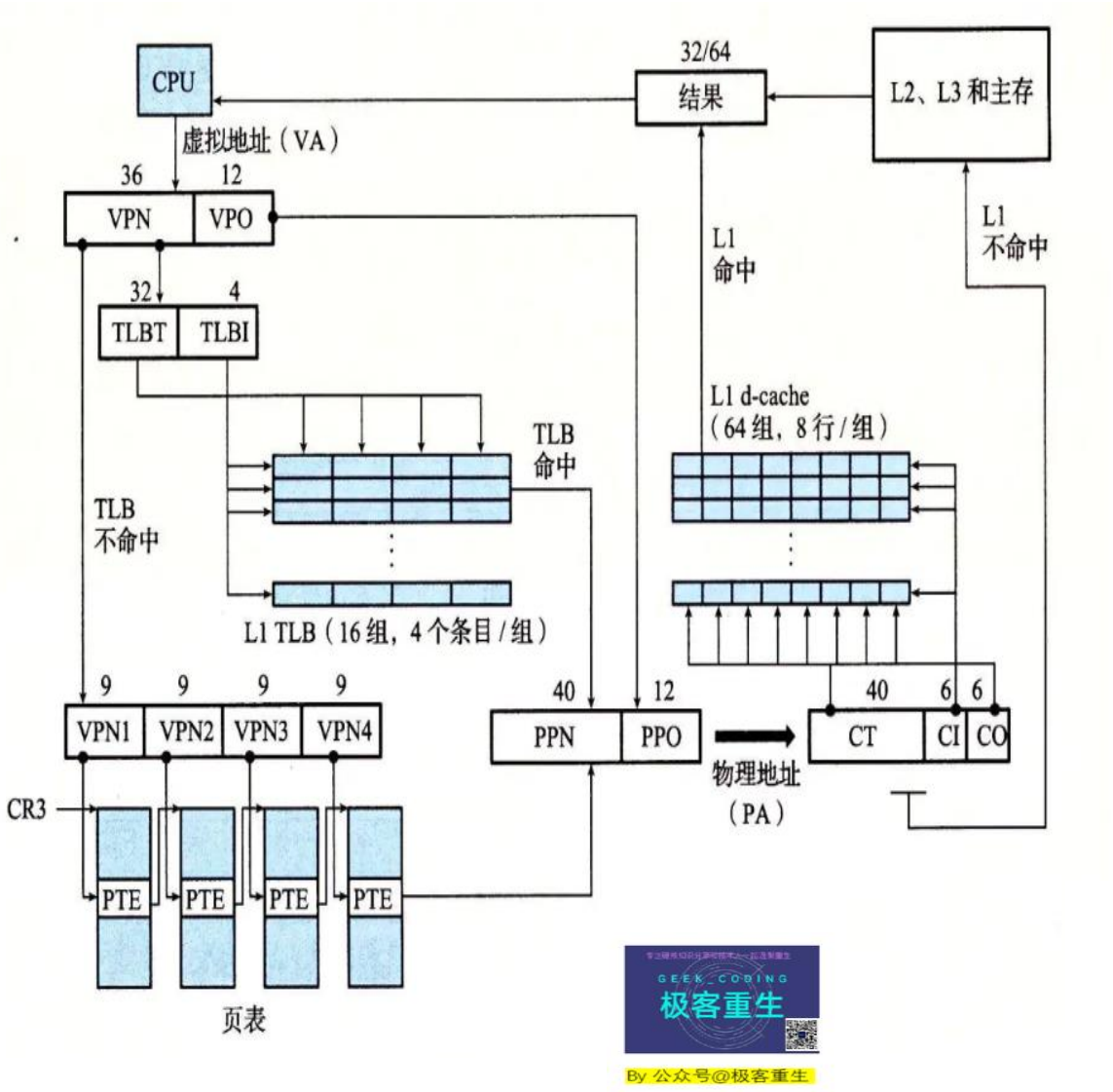
内核页表->初始化预分配

用户页表->按需分配，缺页中断



物理地址空间

内存管理-虚拟内存机制



TLB miss

mmu会进行多级页表遍历遍历过程如下：

1）首先在进程切换时将该进程的页目录起始物理地址放到CR3（x86）寄存器中；

2）当该进程指令访问一个地址时，先经过MMU分段单元转化成线性地址；

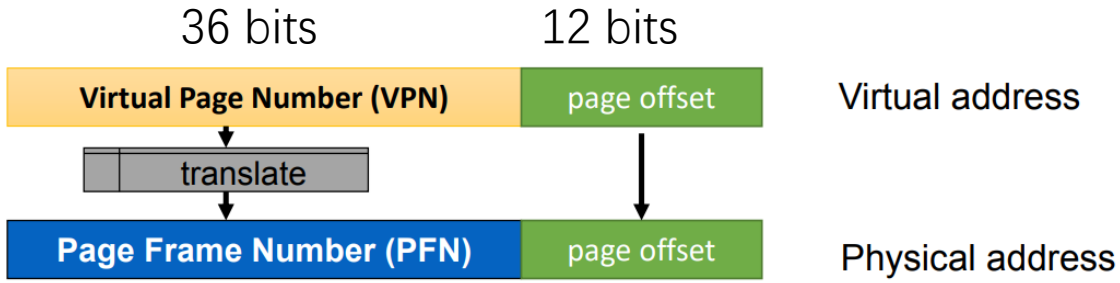
3）把线性地址中**PGD**项（9bit）作为偏移量，加上CR3寄存器中存储的页目录物理基地址，得到的**PUD**就是该地址所在页表的物理基地址；

4）把线性地址中**PUD**项（9bit）作为偏移量，加上3中得到的页表物理基地址，得到的PTE就是该地址所在页表的物理基地址；

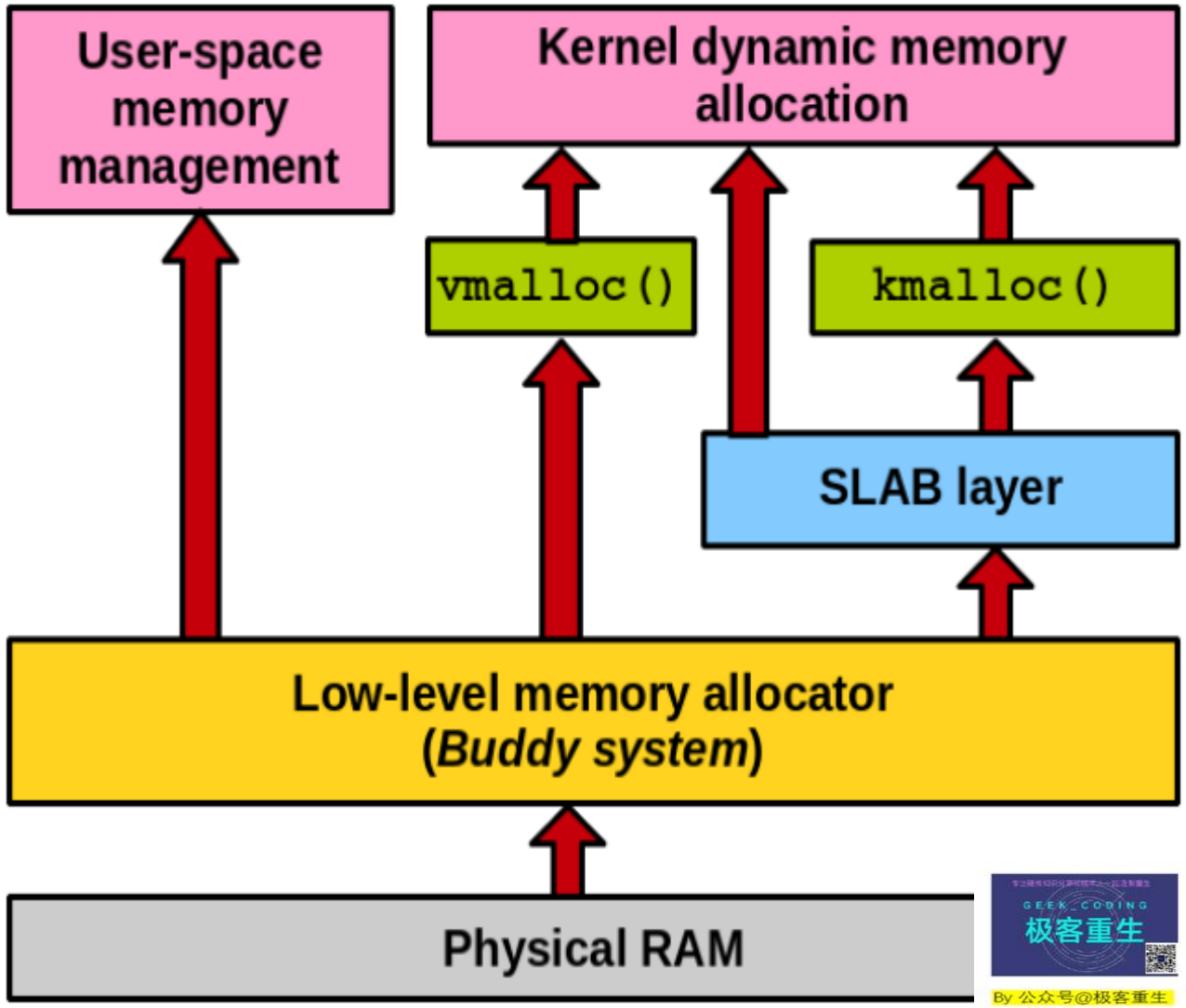
5）把线性地址中**PMD**项（9bit）作为偏移量，加上4中得到的页表物理基地址，得到的PTE就是该地址所在页表的物理基地址；

6）把线性地址中**PTE**项（9bit）作为偏移量，加上5中得到的页表物理基地址，得到的PTE就是该地址所在页表的物理基地址；

7）线性地址中page offset项（最后12bit-大小4K），加上上面得到的页框基地址就是该线性地址映射的物理内存地址。



Linux内存分配算法



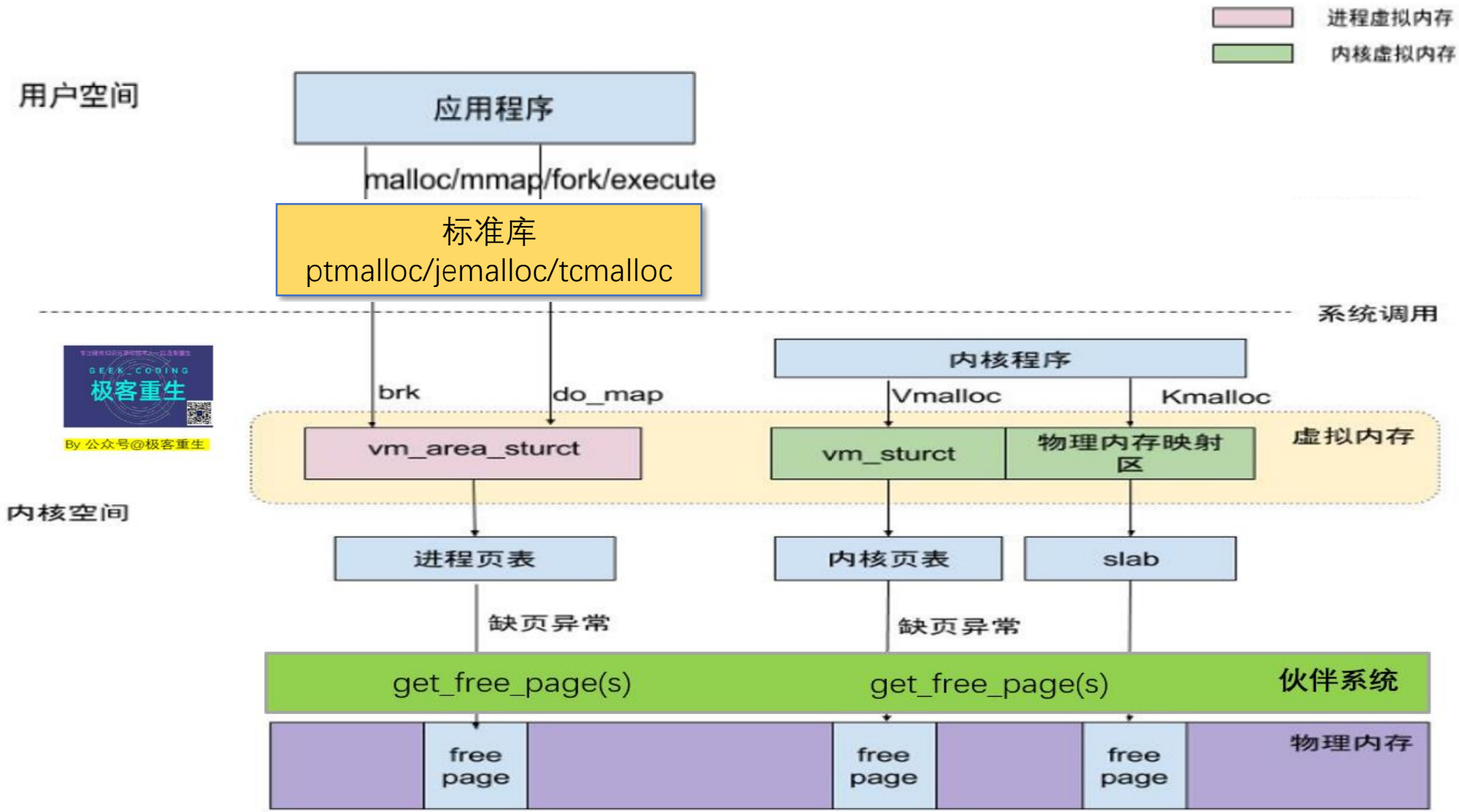
1) **基本原理**

- 产生原因：内存分配较小，并且分配的这些小的内存生存周期又较长，反复申请后将产生内存碎片的出现
- 优点：提高分配速度，便于内存管理，防止内存泄露
- 缺点：大量的内存碎片会使系统缓慢，内存使用率低，浪费大

2) **如何避免内存碎片**

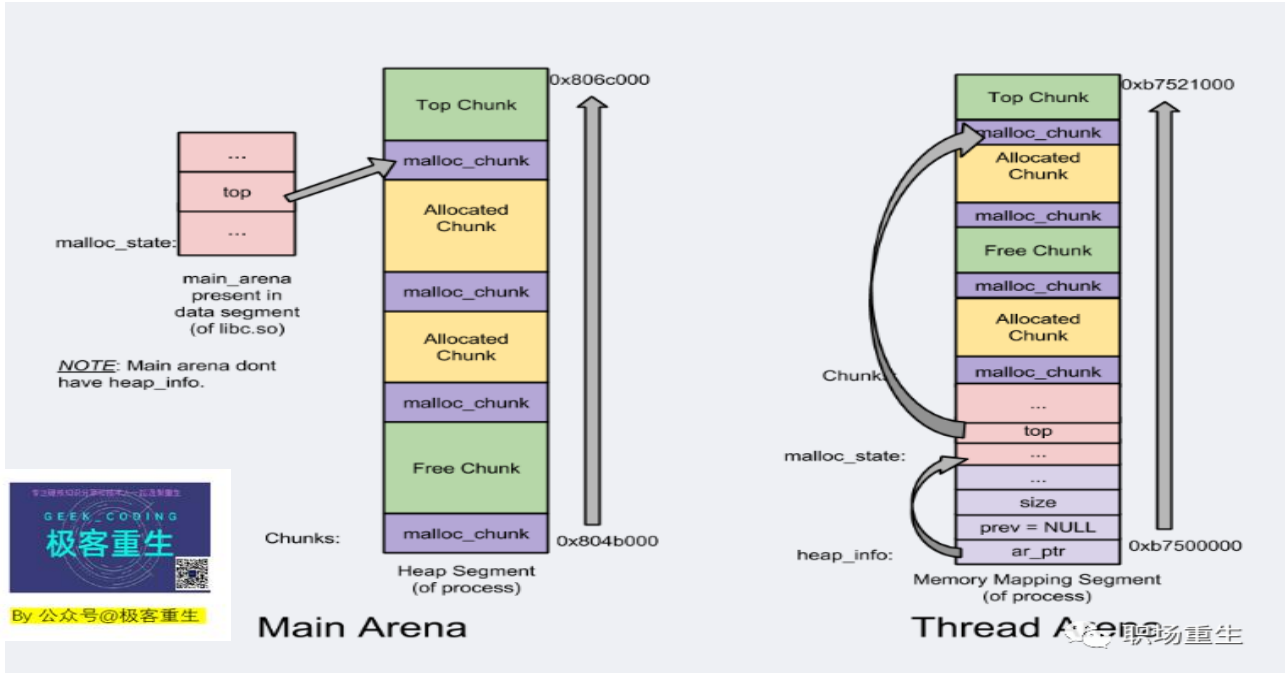
- 少用动态内存分配的函数(尽量使用栈空间)
- 分配内存和释放的内存尽量在同一个函数中
- 尽量一次性申请较大的内存，而不要反复申请小内存
- 尽可能申请大块的 2 的指数幂大小的内存空间
- 外部碎片避免——伙伴系统算法
- 内部碎片避免——slab 算法
- 自己进行内存管理工作，设计内存池

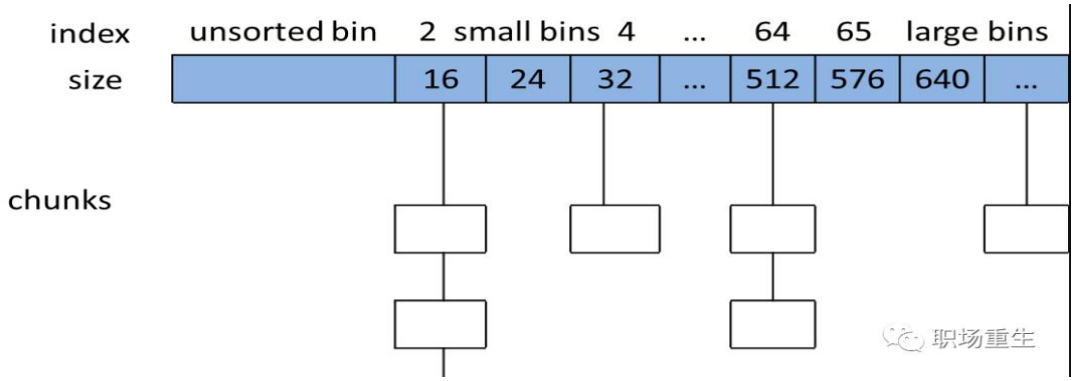
Linux内存分配算法



Linux动态内存分配-**ptmalloc**

是glibc的内存分配管理模块，主要核心技术点



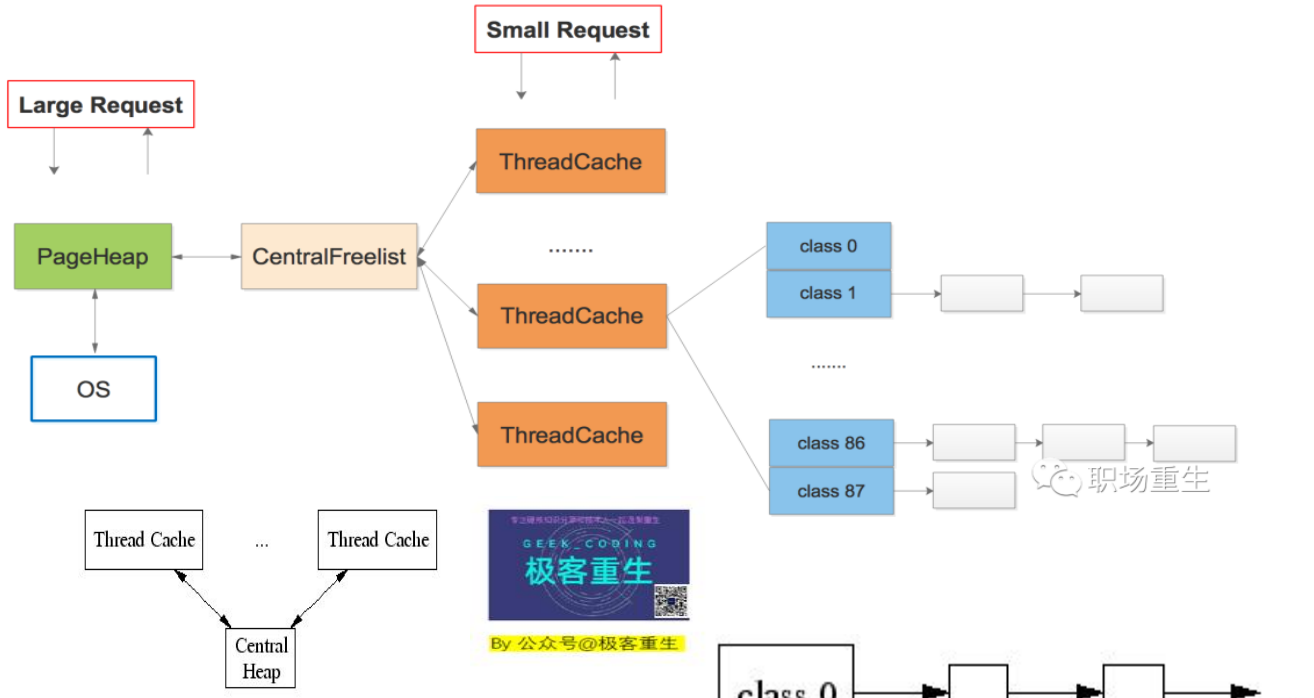


- Arena-main /thread；支持多线程
- Heap segments；for thread arena via by mmap call ；提高管理
- chunk/Top chunk/Last Remainder chunk；提高内存分配的局部性
- bins/fast bin/unsorted bin/small bin/large bin;提高分配效率

缺陷

- 后分配的内存先释放，因为ptmalloc收缩内存是从top chunk 开始，如果与 top chunk 相邻的 chunk 不能释放，top chunk 以下的 chunk 都无法释放。
- 多线程锁开销大， 需要避免多线程频繁分配释放。
- 内存从thread的areana中分配， 内存不能从一个arena移动到另一个arena， 就是说如果多线程使用内存不均衡，容易导致内存的浪费。比如说线程1使用了300M内存，完成任务后glibc没有释放给操作系统，线程2开始创建了一个新的arena， 但是线程1的300M却不能用了。
- 每个chunk至少8字节的开销很大
- 不定期分配长生命周期的内存容易造成内存碎片，不利于回收。64位系统最好分配32M以上内存，这是使用mmap的阈值。

Linux动态内存分配-tcmalloc



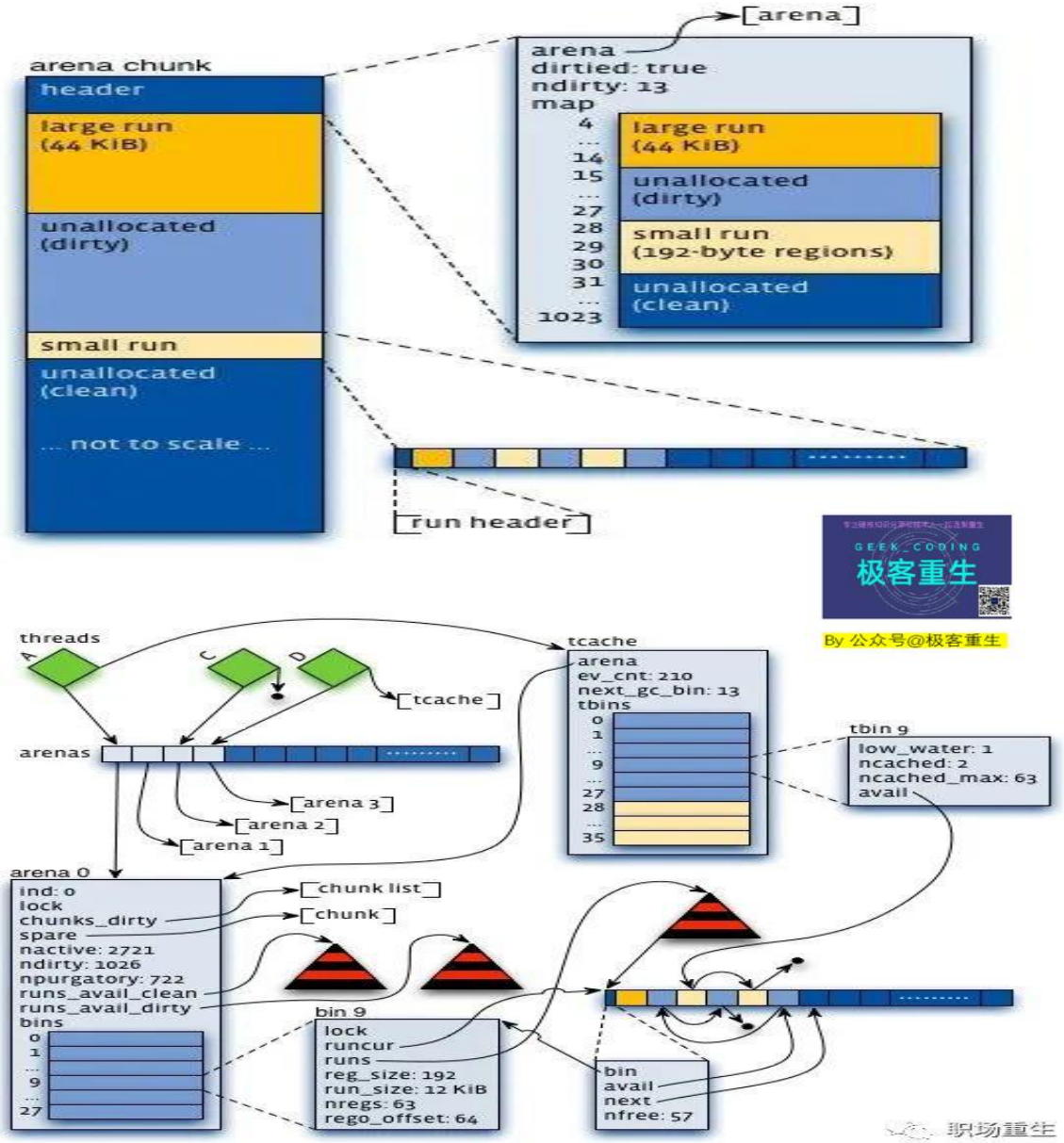
google的gperftools内存分配管理模块，主要核心技术点

1. thread-localcache/periodic garbagecollections/CentralFreeList；提高多线程性能,提高cache利用率
2. ThreadSpecific Free List/size-classes [8,16,32,…32k]: 更好小对象内存分配;
3. The central page heap：更好的大对象内存分配。
4. TCMalloc管理的堆由一系列页面组成。连续的页面由一个“跨度”（Span）对象来表示。一个跨度可以是已被分配或者是自由的。

tcmalloc的改进和问题

- ThreadCache会阶段性的回收内存到CentralCache里。解决了ptmalloc2中arena之间不能迁移的问题。
- Tcmalloc占用更少的额外空间。例如，分配N个8字节对象可能要使用大约8N * 1.01字节的空间。即，多用百分之一的空间。Ptmalloc2使用最少8字节描述一个chunk。
- 更快。小对象几乎无锁，>32KB的对象从CentralCache中分配使用自旋锁。并且>32KB对象都是页面对齐分配，多线程的时候应尽量避免频繁分配，否则也会造成自旋锁的竞争和页面对齐造成的浪费。

Linux动态内存分配-**jemalloc**



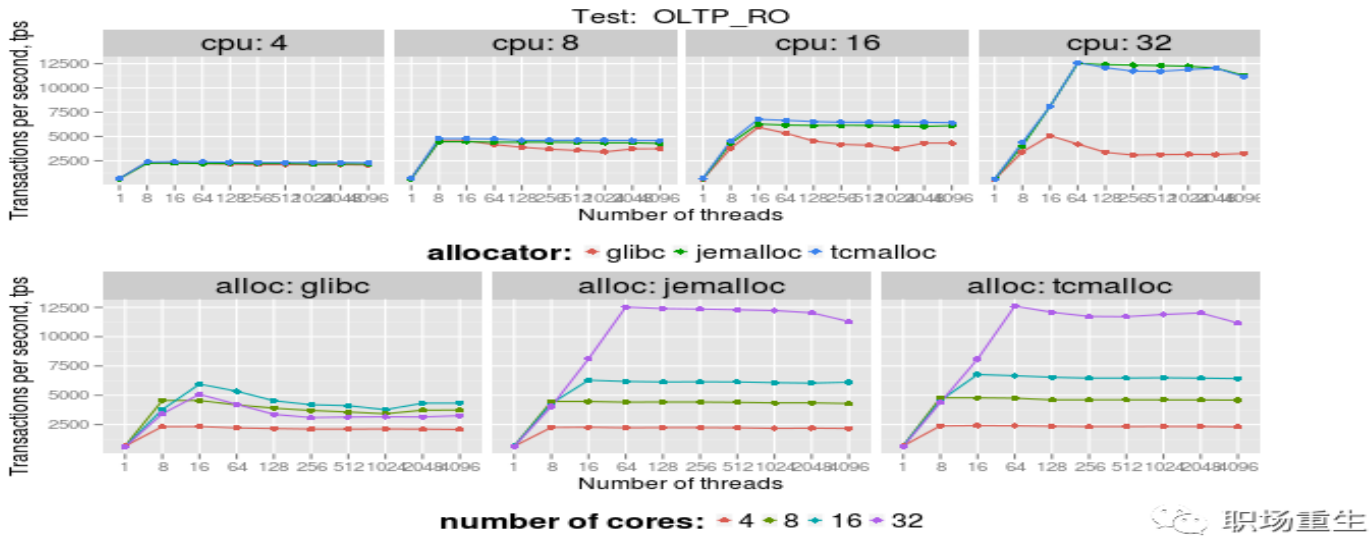
FreeBSD的提供的内存分配管理模块，主要核心技术点

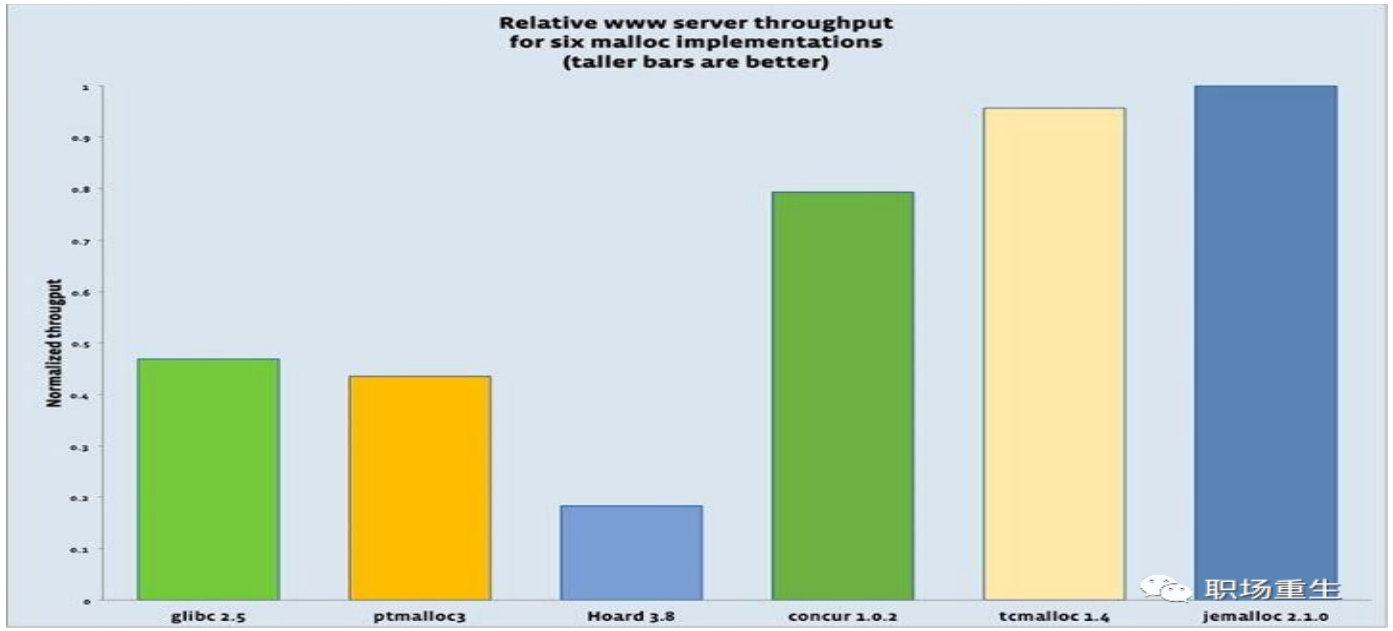
1. 与tcmalloc类似，每个线程同样在<32KB的时候无锁使用线程本地cache;
2. Jemalloc 在 64bits 系统上使用下面的 size-class 分类：
Small: [8], [16, 32, 48, ···, 128], [192, 256, 320, ···, 512] [768,1024, 1280, ···, 3840]
Large: [4 KiB, 8 KiB, 12 KiB, ···, 4072 KiB]
Huge: [4 MiB, 8 MiB, 12 MiB, ···]
3. small/large对象查找metadata需要常量时间， huge对象通过全局红黑树在对数时间内查找
4. 虚拟内存被逻辑上分割成chunks（默认是4MB，1024个4k页），应用线程通过round-robin算法在第一次malloc的时候分配arena， 每个arena都是相互独立的，维护自己的chunks， chunk切割pages到small/large对象。free()的内存总是返回到所属的arena中，而不管是哪个线程调用free().
5. 通过arena分配的时候需要对arena bin（每个small size-class一个，细粒度）加锁，或arena本身加锁。并且线程cache对象也会通过垃圾回收指数退让算法返回到arena中。

jemalloc的优化

- Jmalloc小对象也根据size-class，但是它使用了低地址优先的策略，来降低内存碎片化。
- Jemalloc大概需要2%的额外开销。（tcmalloc 1%，ptmalloc最少8B）.
- Jemalloc和tcmalloc类似的线程本地缓存，避免锁的竞争 .
- 相对未使用的页面，优先使用dirty page，提升缓存命中。

Linux动态内存分配-**性能比较**





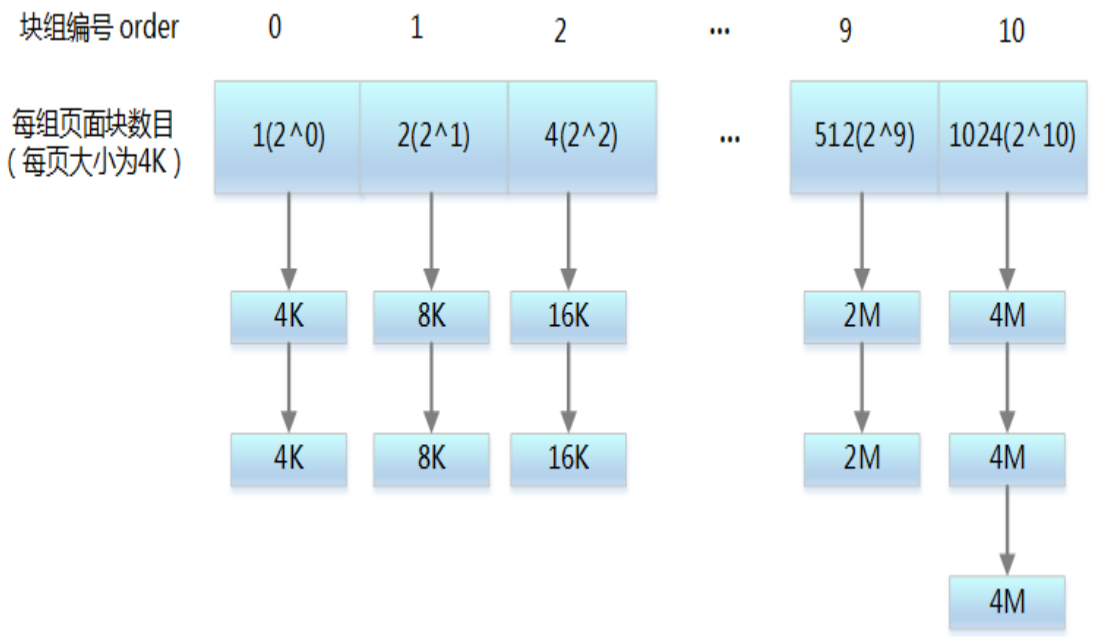
对比测试结果

总结

1总的来看，作为基础库的ptmalloc是最为稳定的内存管理器，无论在什么环境下都能适应，但是分配效率相对较低。
2 tcmalloc针对多核情况有所优化，性能有所提高，但是内存占用稍高，大内存分配容易出现CPU飙升。
3 jemalloc的内存占用更高，但是在多核多线程下的表现也最为优异
4 在多线程环境使用tcmalloc和jemalloc效果非常明显。一般支持多核多线程扩展情况下可以使用jemalloc；反之使用tcmalloc可能是更好的选择。

多核多线程的情况下，内存管理需要考虑内存分配加锁、异步内存释放、多线程之间的内存共享、线程的生命周期

Linux内核内存分配-Buddy系统（伙伴系统）

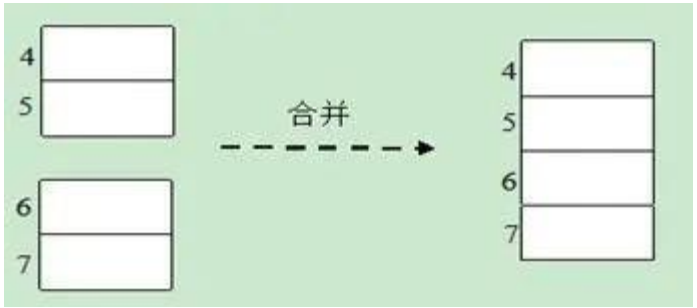


为什么会有伙伴系统？

申请算法

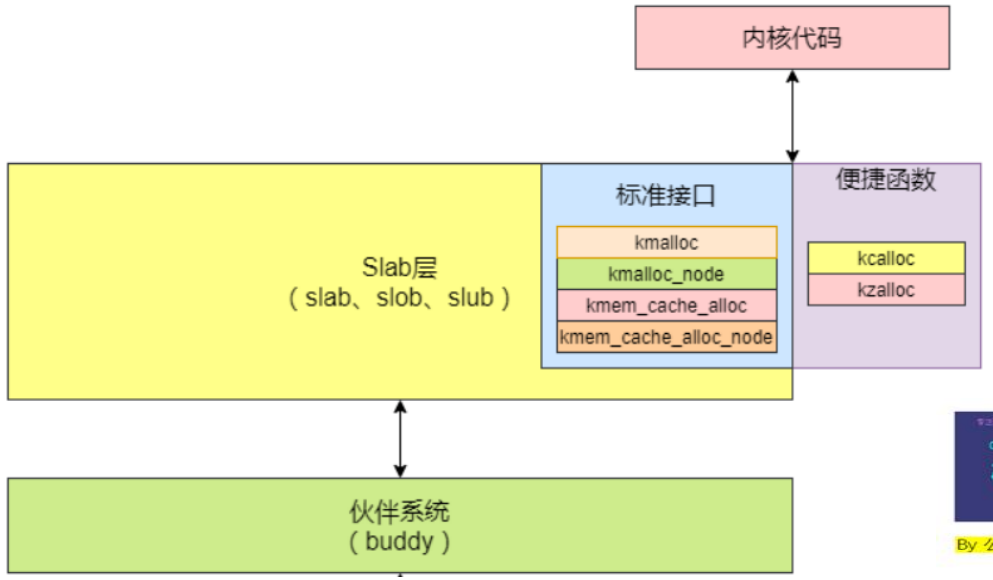
1. 申请 2^i 个页块存储空间，如果 2^i 对应的块链表有空闲页块，则分配给应用
2. 如果没有空闲页块，则查找 2^(i+1) 对应的块链表是否有空闲页块，如果有，则分配 2^i 块链表节点给应用，另外 2^i 块链表节点插入到 2^i 对应的块链表中
3. 如果 2^(i+1) 块链表中没有空闲页块，则重复步骤 2，直到找到有空闲页块的块链表
4. 如果仍然没有，则返回内存分配失败

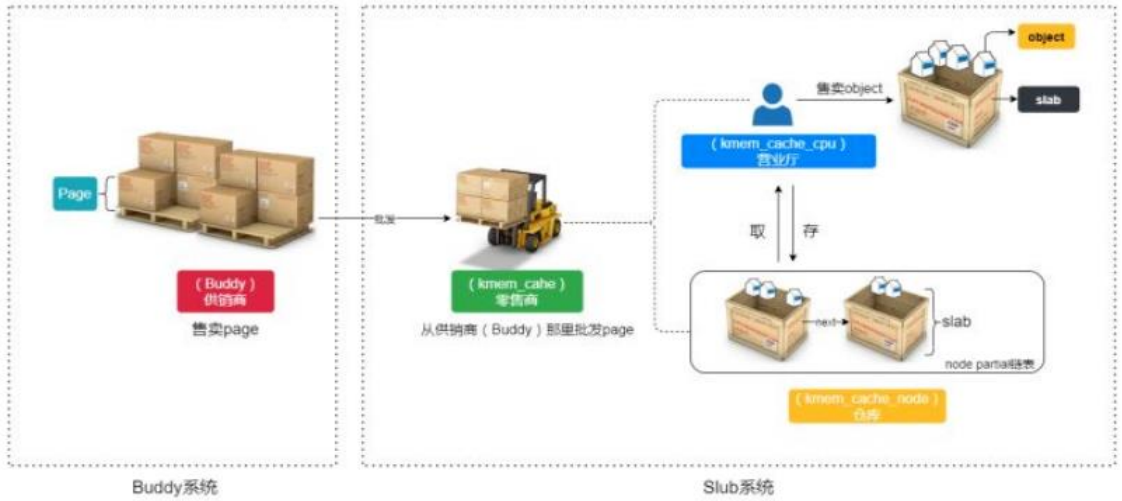
回收算法



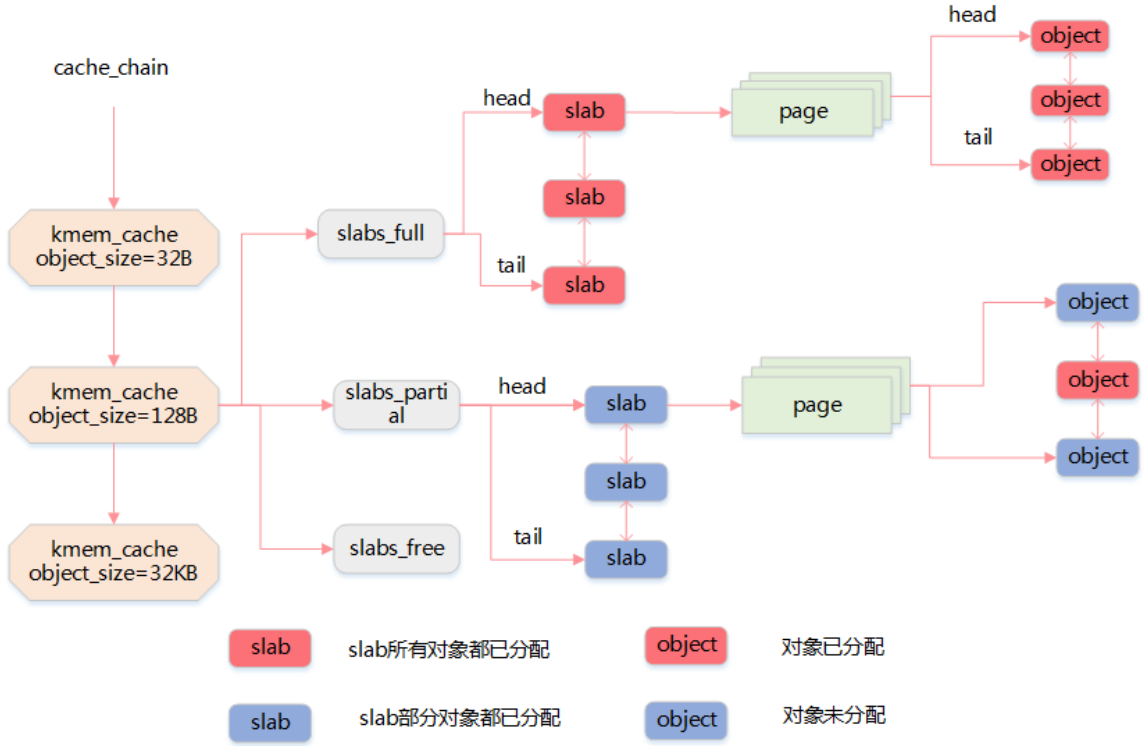
- 释放 2^i 个页块存储空间，查找 2^i 个页块对应的块链表，是否有与其物理地址是连续的页块，如果没有，则无需合并
- 如果有，则合并成 2^（i 1）的页块，以此类推，继续查找下一级块链接，直到不能合并为止。

Linux内核内存分配-Slab算法



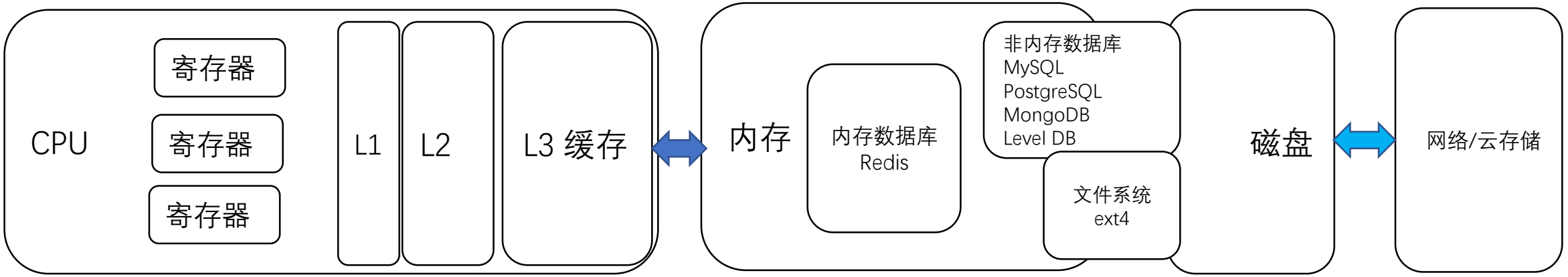


为什么会有slab系统？



- 已经被分配出去的的内存空间大于请求所需的内存空间
- 减少伙伴算法在分配小块连续内存时所产生的内部碎片
- 将频繁使用的对象缓存起来，减少分配、初始化和释放对象的时间开销
- 通过着色技术调整对象以更好的使用硬件高速缓存
- kmem_cacche_cpu：一个空闲对象链表，每个CPU一个的独享cache，分配释放对象无需加锁。
- kmem_cache_node：每个NUMA NODE的所有CPU共享的cache，单位为page(s)，获取后被提升到对应CPU的slab cache。

理解”存储”体系



存储器类型	位于哪里	存储容量	半导体工艺	访问时间	如何访问
CPU寄存器	位于CPU执行单元中。	CPU寄存器通常只有几个到几十个，每个寄存器的容量取决于CPU的字长，所以一共只有几十到几百字节。	“寄存器”这个名字就是一种数字电路的名字，它由一组触发器（Flip-flop）组成，每个触发器保存一个Bit的数据，可以做存取和移位等操作。计算机掉电时寄存器中保存的数据会丢失。	寄存器是访问速度最快的存储器，典型的访问时间是几纳秒。	使用哪个寄存器，如何使用寄存器，这些都是由指令决定的。
Cache	和MMU一样位于CPU核中。	Cache通常分为几级，最典型的是如上图所示的两级Cache，一级Cache更靠近CPU执行单元，二级Cache更靠近物理内存，通常一级Cache有几十到几百KB，二级Cache有几百KB到几MB。	Cache和内存都是由RAM（Random Access Memory）组成的，可以根据地址随机访问，计算机掉电时RAM中保存的数据会丢失。不同的是，Cache通常由SRAM（Static RAM，静态RAM）组成，而内存通常由DRAM（Dynamic RAM，动态RAM）组成。DRAM电路比SRAM简单，存储容量可以做得更大，但DRAM的访问速度比SRAM慢。	典型的访问时间是几十纳秒。	Cache缓存最近访问过的内存数据，由于Cache的访问速度是内存的几十倍，所以有效利用Cache可以大大提高计算机的整体性能。一级Cache是这样工作的：CPU执行单元要访问内存时首先发出VA，Cache利用VA查找相应的数据有没有被缓存，如果Cache中有就不需要访问物理内存了，如果是读操作就直接将Cache中的数据传给CPU寄存器，如果是写操作就直接改写到Cache中；如果Cache没有缓存该数据，就去物理内存中取数据，但并不是要哪个字节就取哪个字节，而是把相邻的几十个字节都取上来缓存着，以备下次用到，这称为一个Cache Line，典型的Cache Line大小是32~256字节。如果计算机还配置了二级缓存，则在访问物理内存之前先用PA去二级缓存中查找。一级缓存是用VA寻址的，二级缓存是用PA寻址的，这是它们的区别。Cache所做的工作是由硬件自动完成的，而不是像寄存器一样由指令决定先做什么后做什么。
内存	位于CPU外的芯片，与CPU通过地址和数据总线相连。	典型的存储容量是几百MB到几GB。	由DRAM组成，详见上面关于Cache的说明。	典型的访问时间是几百纳秒。	内存是通过地址来访问的，在启用MMU的情况下，程序指令中的地址是VA，而访问内存用的是PA，它们之间的映射关系由操作系统维护。
硬盘	位于设备总线上，并不直接和CPU相连，CPU通过设备总线的控制器访问硬盘。	典型的存储容量是几百GB到几TB。	硬盘由磁性介质和磁头组成，访问硬盘时存在机械运动，磁头要移动，磁性介质要旋转，机械运动的速度很难提高到电子的速度，所以访问速度很受限制。保存在硬盘上的数据掉电后不会丢失。	典型的访问时间是几毫秒，是寄存器访问时间的10^6倍。	由驱动程序操作设备总线控制器去访问。由于硬盘的访问速度较慢，操作系统通常一次从硬盘上读几个页面到内存中缓存起来，如果这几个页面后来都被程序访问到了，那么这一次读硬盘的时间就可以分摊（Amortize）给程序的多次访问了。

内存的使用场景

- page 管理
- slab（kmalloc、内存池）
- 用户态内存使用（malloc、relloc 文件映射、共享内存）
- 程序的内存 map（栈、堆、code、data）
- 内核和用户态的数据传递（copy_from_user、copy_to_user）
- 内存映射（硬件寄存器、保留内存）
- DMA 内存