# 1 Linux 内存相关问题汇总

这篇文章是对 Linux 内存相关问题的集合，工作中会有很大的帮助。关注公号的朋友应该知道之前我写过从内核态到用户态 Linux 内存管理相关的基础文章，在阅读前最好浏览下，链接如下：

**CPU 是如何访问内存的？**

**物理地址和虚拟地址的分布**

**Linux 内核内存管理算法 Buddy 和 Slab**

**Linux 用户态进程的内存管理**

linux 内存是后台开发人员，需要深入了解的计算机资源。合理的使用内存，有助于提升机器的性能和稳定性。本文主要介绍 linux 内存组织结构和页面布局，内存碎片产生原因和优化算法，linux 内核几种内存管理的方法，内存使用场景以及内存使用的那些坑。从内存的原理和结构，到内存的算法优化，再到使用场景，去探寻内存管理的机制和奥秘。

## 1.1 一、走进 linux 内存

1、内存是什么？

1)内存又称主存，是 CPU 能直接寻址的存储空间，由半导体器件制成

2)内存的特点是存取速率快

**2、内存的作用**

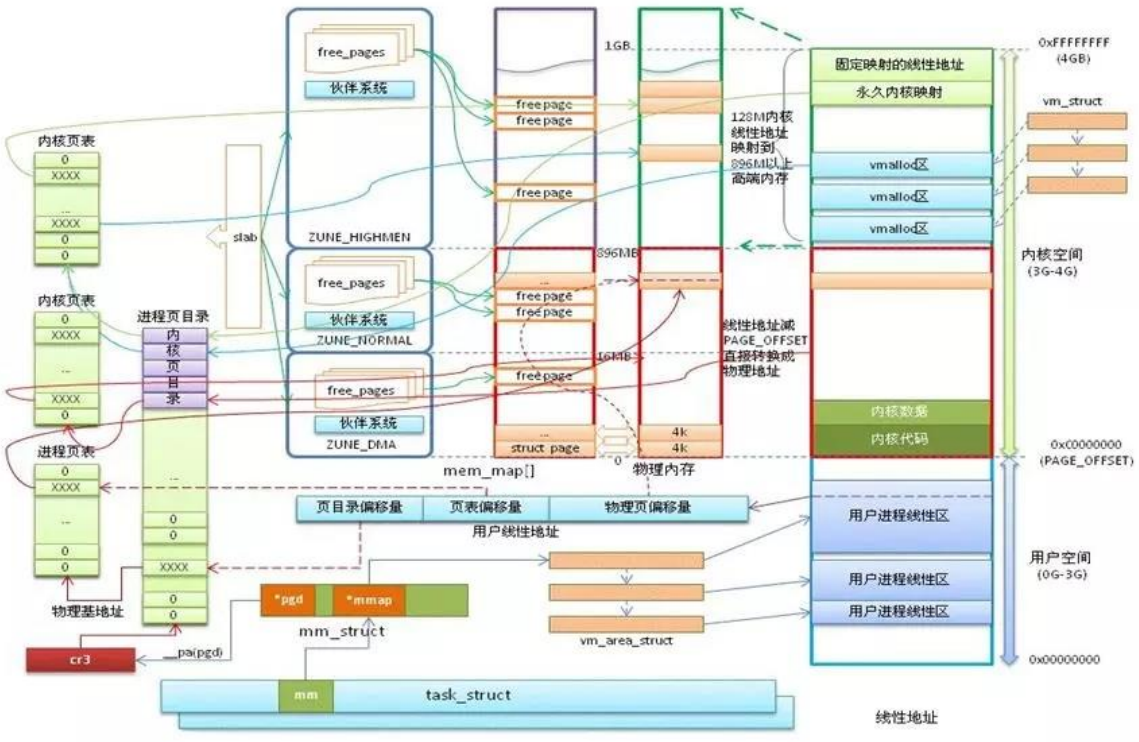
1)暂时存放 cpu 的运算数据

2)硬盘等外部存储器交换的数据

3)保障 cpu 计算的稳定性和高性能

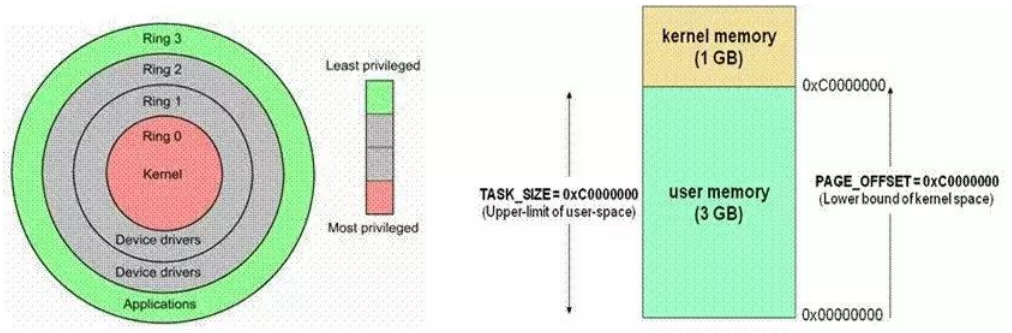
## 1.2 二、 linux 内存地址空间

### 1、linux 内存地址空间 Linux 内存管理全貌



## 2、内存地址——用户态&内核态

- 用户态：Ring3 运行于用户态的代码则要受到处理器的诸多
- 内核态：Ring0 在处理器的存储保护中，核心态
- 用户态切换到内核态的 3 种方式：系统调用、异常、外设中断
- 区别：每个进程都有完全属于自己的，独立的，不被干扰的内存空间；用户态的程序就不能随意操作内核地址空间，具有一定的安全保护作用；内核态线程共享内核地址空间；




## 3、内存地址——MMU 地址转换

- MMU 是一种硬件电路，它包含两个部件，一个是分段部件，一个是分页部件
- 分段机制把一个逻辑地址转换为线性地址
- 分页机制把一个线性地址转换为物理地址
- 

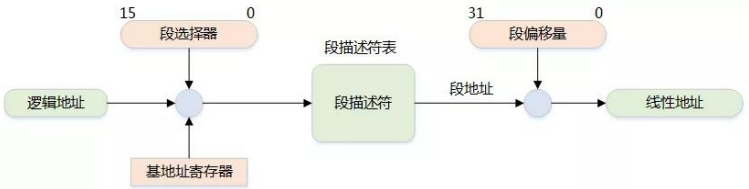

## 4、内存地址——分段机制

1) **段选择符**

- 为了方便快速检索段选择符，处理器提供了 6 个分段寄存器来缓存段选择符，它们是:cs,ss,ds,es,fs 和 gs
- 段的基地址(Base Address)：在线性地址空间中段的起始地址
- 段的界限(Limit)：在虚拟地址空间中，段内可以使用的最大偏移量

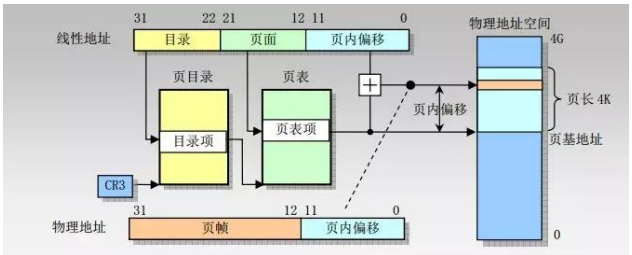
**2)分段实现**

- 逻辑地址的段寄存器中的值提供段描述符，然后从段描述符中得到段基址和段界限，然后加上逻辑地址的偏移量，就得到了线性地址



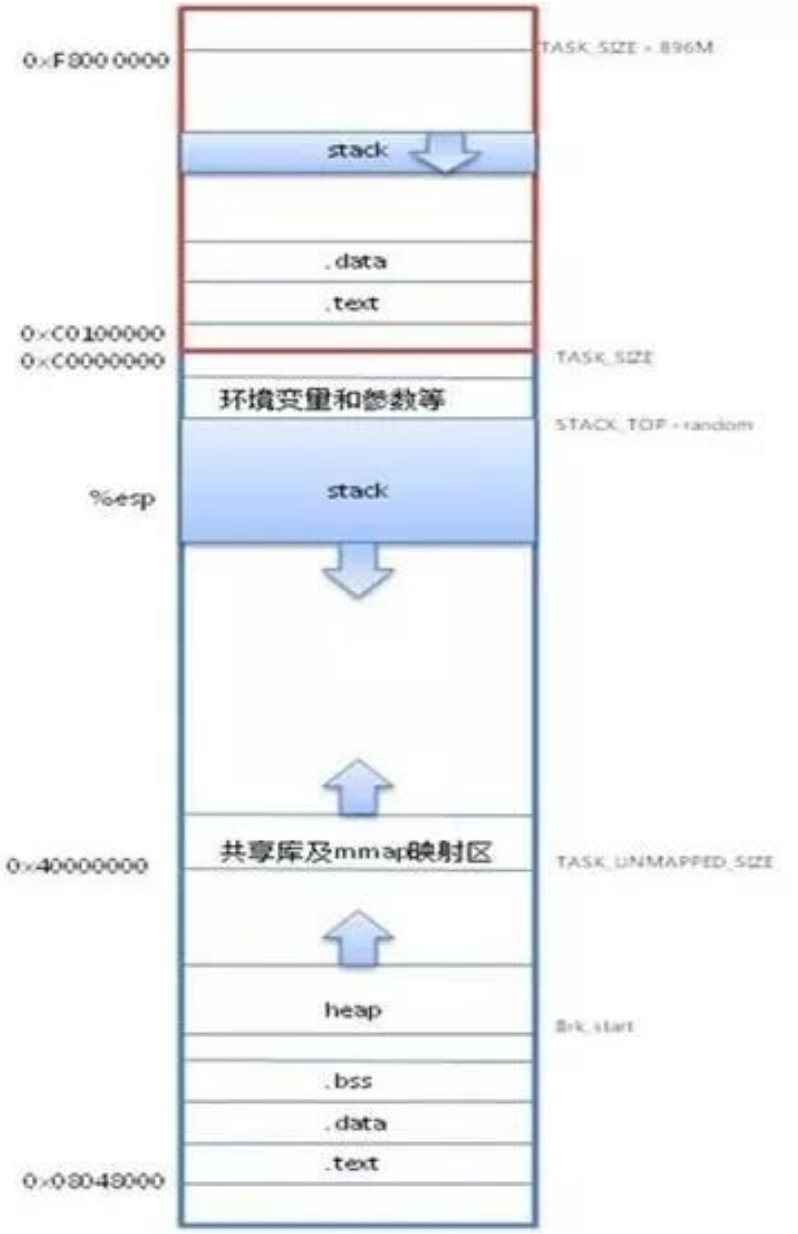

## 5、内存地址——分页机制（32 位）

- 分页机制是在分段机制之后进行的，它进一步将线性地址转换为物理地址
- 10 位页目录，10 位页表项， 12 位页偏移地址
- 单页的大小为 4KB

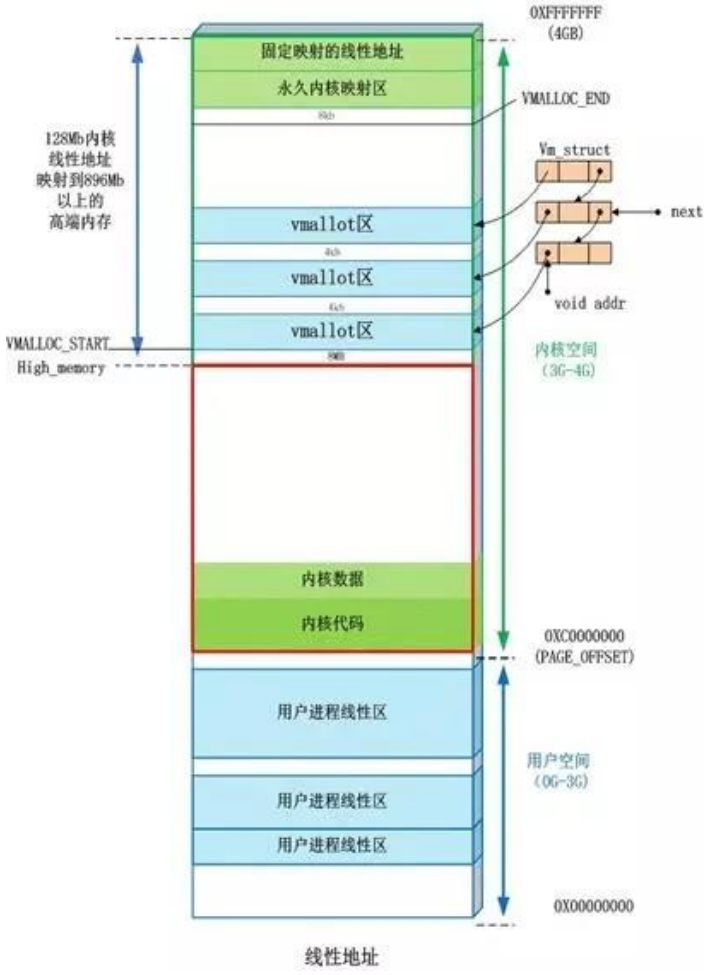



## 6、用户态地址空间

- TEXT：代码段可执行代码、字符串字面值、只读变量
- DATA：数据段，映射程序中已经初始化的全局变量
- BSS 段：存放程序中未初始化的全局变量
- HEAP：运行时的堆，在程序运行中使用 malloc 申请的内存区域
- MMAP：共享库及匿名文件的映射区域
- STACK：用户进程栈



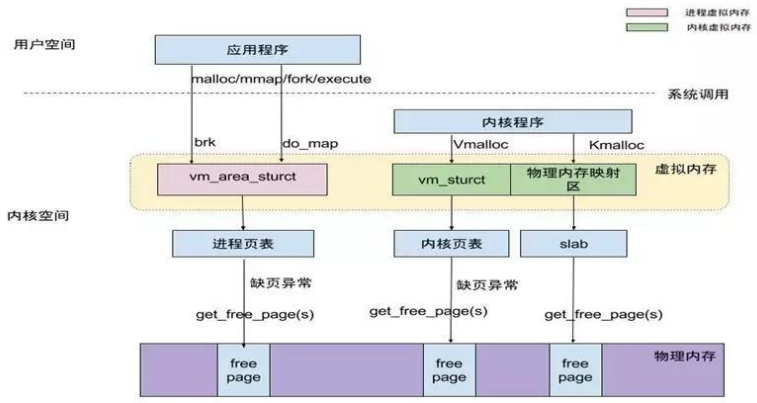

## 7、内核态地址空间




- 直接映射区：线性空间中从 3G 开始最大 896M 的区间，为直接内存映射区
- 动态内存映射区：该区域由内核函数 vmalloc 来分配
- 永久内存映射区：该区域可访问高端内存
- 固定映射区：该区域和 4G 的顶端只有 4k 的隔离带，其每个地址项都服务于特定的用途，如： ACPI_BASE 等

## 8、进程内存空间

- 用户进程通常情况只能访问用户空间的虚拟地址，不能访问内核空间虚拟地址
- 内核空间是由内核负责映射，不会跟着进程变化；内核空间地址有自己对应的页表，用户进程各自有不同额页表




## 1.3 三、 Linux 内存分配算法

内存管理算法——对讨厌自己管理内存的人来说是天赐的礼物

1、内存碎片

**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

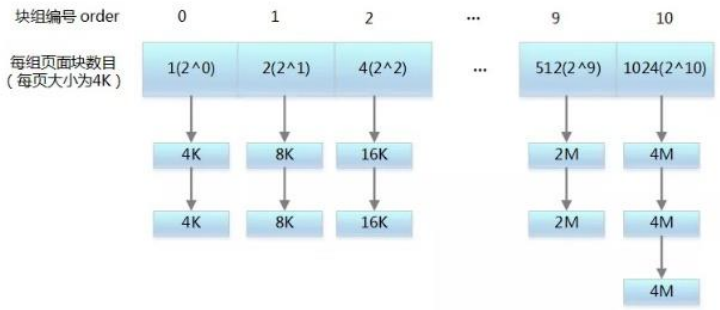
## 2、伙伴系统算法——组织结构

1) 概念

- 为内核提供了一种用于分配一组连续的页而建立的一种高效的分配策略，并有效的解决了外碎片问题
- 分配的内存区是以页框为基本单位的

2) 外部碎片

- 外部碎片指的是还没有被分配出去（不属于任何进程），但由于太小了无法分配给申请内存空间的新进程的内存空闲区域 3)组织结构
- 把所有的空闲页分组为 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 个连续页框的页块。最大可以申请 1024 个连续页，对应 4MB 大小的连续内存



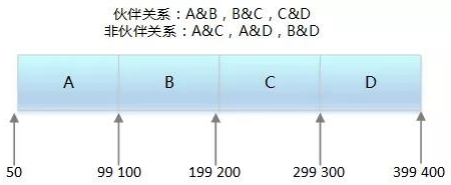

## 3、伙伴系统算法——申请和回收

**1) 申请算法**

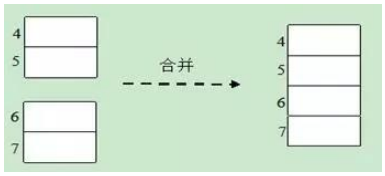
- 申请 2^i 个页块存储空间，如果 2^i 对应的块链表有空闲页块，则分配给应用
- 如果没有空闲页块，则查找 2^(i+1) 对应的块链表是否有空闲页块，如果有，则分配 2^i 块链表节点给应用，另外 2^i 块链表节点插入到 2^i 对应的块链表中
- 如果 2^(i 1) 块链表中没有空闲页块，则重复步骤 2，直到找到有空闲页块的块链表
- 如果仍然没有，则返回内存分配失败

**2) 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




3) 条件

- 两个块具有相同的大小
- 它们的物理地址是连续的
- 页块大小相同

## 4、如何分配 4M 以上内存?

1) 为何限制大块内存分配

- 分配的内存越大，失败的可能性越大
- 大块内存使用场景少

2) 内核中获取 4M 以上大内存的方法

- 修改 MAX_ORDER，重新编译内核
- 内核启动选型传递"mem="参数，如"mem=80M，预留部分内存；然后通过 request_mem_region 和 ioremap_nocache 将预留的内存映射到模块中。需要修改内核启动参数，无需重新编译内核. 但这种方法不支持 x86 架构，只支持 ARM，PowerPC 等非 x86 架构
- 在 start_kernel 中 mem_init 函数之前调用 alloc_boot_mem 函数预分配大块内存，需要重新编译内核 vmalloc 函数，内核代码使用它来分配在虚拟内存中连续但在物理内存中不一定连续的内存

## 5、伙伴系统——反碎片机制

1) 不可移动页

- 这些页在内存中有固定的位置，不能够移动，也不可回收
- 内核代码段，数据段，内核 kmalloc() 出来的内存，内核线程占用的内存等

2) 可回收页

- 这些页不能移动，但可以删除。内核在回收页占据了太多的内存时或者内存短缺时进行页面回收 3)可移动页
- 这些页可以任意移动，用户空间应用程序使用的页都属于该类别。它们是通过页表映射的
- 当它们移动到新的位置，页表项也会相应的更新

## 6、slab 算法——基本原理

1) 基本概念

- Linux 所使用的 slab 分配器的基础是 Jeff Bonwick 为 SunOS 操作系统首次引入的一种算法
- 它的基本思想是将内核中经常使用的对象放到高速缓存中，并且由系统保持为初始的可利用状态。比如进程描述符，内核中会频繁对此数据进行申请和释放

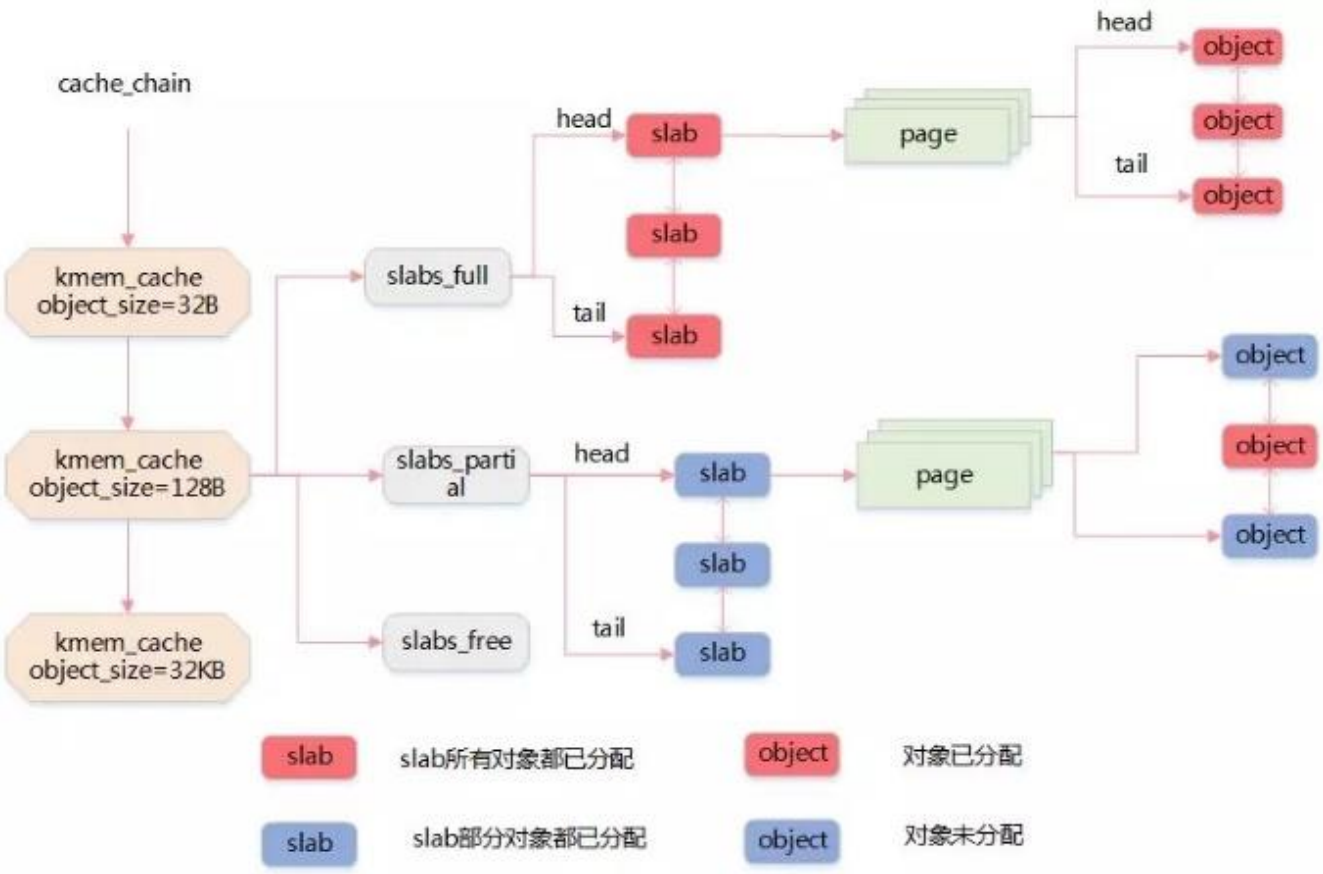
2) 内部碎片

- 已经被分配出去的的内存空间大于请求所需的内存空间 3)基本目标
- 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

## 7、slab 分配器的结构

- 由于对象是从 slab 中分配和释放的，因此单个 slab 可以在 slab 列表之间进行移动
- slabs_empty 列表中的 slab 是进行回收（reaping）的主要备选对象
- slab 还支持通用对象的初始化，从而避免了为同一目而对一个对象重复进行初始化




## 8、slab 高速缓存

1) 普通高速缓存

●slab 分配器所提供的小块连续内存的分配是通过通用高速缓存实现的

●通用高速缓存所提供的对象具有几何分布的大小，范围为 32 到 131072 字节。

●内核中提供了 kmalloc() 和 kfree() 两个接口分别进行内存的申请和释放

2) 专用高速缓存

●内核为专用高速缓存的申请和释放提供了一套完整的接口，根据所传入的参数为具体的对象分配 slab 缓存

●kmem_cache_create() 用于对一个指定的对象创建高速缓存。它从 cache_cache 普通高速缓存中为新的专有缓存分配一个高速缓存描述符，并把这个描述符插入到高速缓存描述符形成的 cache_chain 链表中

●kmem_cache_alloc() 在其参数所指定的高速缓存中分配一个 slab。相反， kmem_cache_free() 在其参数所指定的高速缓存中释放一个 slab

## 9、内核态内存池

1) 基本原理

●先申请分配一定数量的、大小相等(一般情况下) 的内存块留作备用

●当有新的内存需求时，就从内存池中分出一部分内存块，若内存块不够再继续申请新的内存

●这样做的一个显著优点是尽量避免了内存碎片，使得内存分配效率得到提升

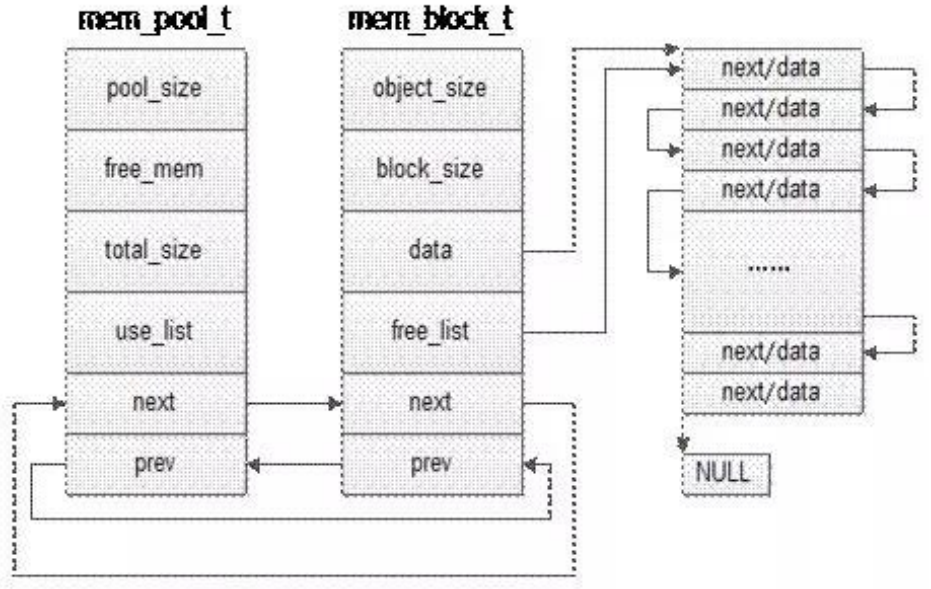
2) 内核 API

●mempool_create 创建内存池对象

●mempool_alloc 分配函数获得该对象

●mempool_free 释放一个对象

●mempool_destroy 销毁内存池




## 10、用户态内存池

1) C++ 实例

```
template <int N>
class heappool
{
private:
        typedef struct { char data[N]; } block_type;
        block_type* ptr;
private:
        static size_t count;
        static std::list<block_type*> L;
public:
        heappool() {
                if(L.empty()) ptr=new block_type; else { ptr=L.back(); L.pop_back(); }
        }
        ~heappool() {
                L.push_back(ptr); if(L.size()>count) { delete L.front(); L.pop_front(); }
        }
        static void set_block_count(size_t cnt) {count=cnt;}
public:
        char* data() {return (char*)ptr;}
        size_t size() {return N;}
};
```

## 11、DMA 内存

### 1) 什么是 DMA

●直接内存访问是一种硬件机制，它允许外围设备和主内存之间直接传输它们的 I/O 数据，而不需要系统处理器的参与 2) DMA 控制器的功能

●能向 CPU 发出系统保持（HOLD）信号，提出总线接管请求

●当 CPU 发出允许接管信号后，负责对总线的控制，进入 DMA 方式

●能对存储器寻址及能修改地址指针，实现对内存的读写操作

●能决定本次 DMA 传送的字节数，判断 DMA 传送是否结束

●发出 DMA 结束信号，使 CPU 恢复正常工作状态

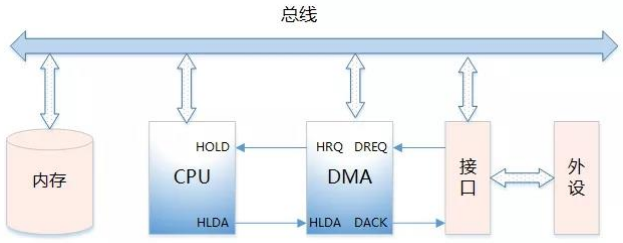
### 2) DMA 信号

●DREQ：DMA 请求信号。是外设向 DMA 控制器提出要求，DMA 操作的申请信号

●DACK：DMA 响应信号。是 DMA 控制器向提出 DMA 请求的外设表示已收到请求和正进行处理的信号

●HRQ：DMA 控制器向 CPU 发出的信号，要求接管总线的请求信号。

●HLDA：CPU 向 DMA 控制器发出的信号，允许接管总线的应答信号：




## 1.4 四、 内存使用场景

**out of memory 的时代过去了吗？no，内存再充足也不可任性使用。**

### 1、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

### 2、用户态内存分配函数

- alloca 是向栈申请内存,因此无需释放
- malloc 所分配的内存空间未被初始化，使用 malloc() 函数的程序开始时(内存空间还没有被重新分配) 能正常运行，但经过一段时间后(内存空间已被重新分配) 可能会出现问题
- calloc 会将所分配的内存空间中的每一位都初始化为零
- realloc 扩展现有内存空间大小

**a)如果当前连续内存块足够 realloc 的话**，只是将 p 所指向的空间扩大，并返回 p 的指针地址。这个时候 q 和 p 指向的地址是一样的

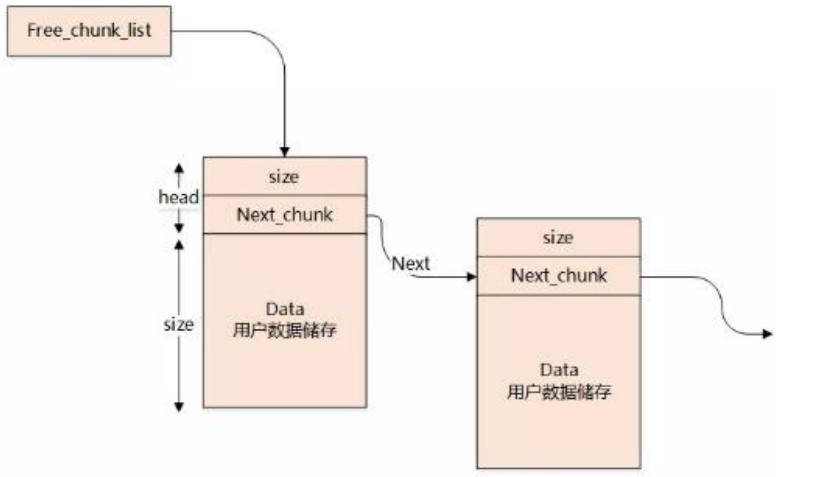
**b)如果当前连续内存块不够长度，再找一**个足够长的地方，分配一块新的内存，q，并将 p 指向的内容 copy 到 q，返回 q。并将 p 所指向的内存空间删除

### 3、内核态内存分配函数

函数分配原理最大内存其他_get_free_pages 直接对页框进行操作 4MB 适用于分配较大量的连续物理内存 kmem_cache_alloc 基于 slab 机制实现 128KB 适合需要频繁申请释放相同大小内存块时使用 kmalloc 基于 kmem_cache_alloc 实现 128KB 最常见的分配方式，需要小于页框大小的内存时可以使用 vmalloc 建立非连续物理内存到虚拟地址的映射物理不连续，适合需要大内存，但是对地址连续性没有要求的场合 dma_alloc_coherent 基于_alloc_pages 实现 4MB 适用于 DMA 操作 ioremap 实现已知物理地址到虚拟地址的映射适用于物理地址已知的场合，如设备驱动 alloc_bootmem 在启动 kernel 时，预留一段内存，内核看不见小于物理内存大小，内存管理要求较高

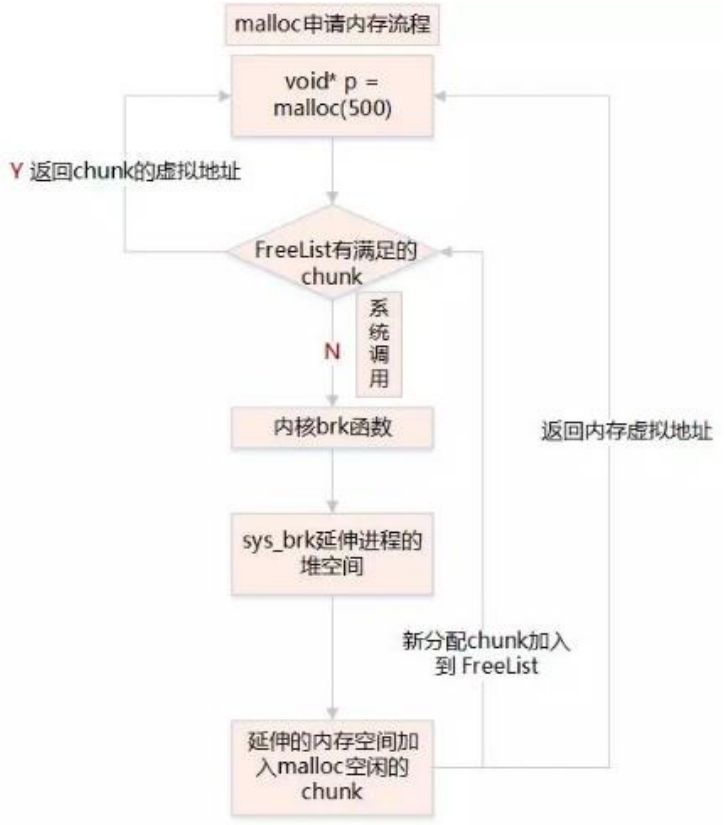
### 4、malloc 申请内存

调用 malloc 函数时，它沿 free_chuck_list 连接表寻找一个大到足以满足用户请求所需要的内存块




free_chuck_list 连接表的主要工作是维护一个空闲的堆空间缓冲区链表

如果空间缓冲区链表没有找到对应的节点，需要通过系统调用 sys_brk 延伸进程的栈空间



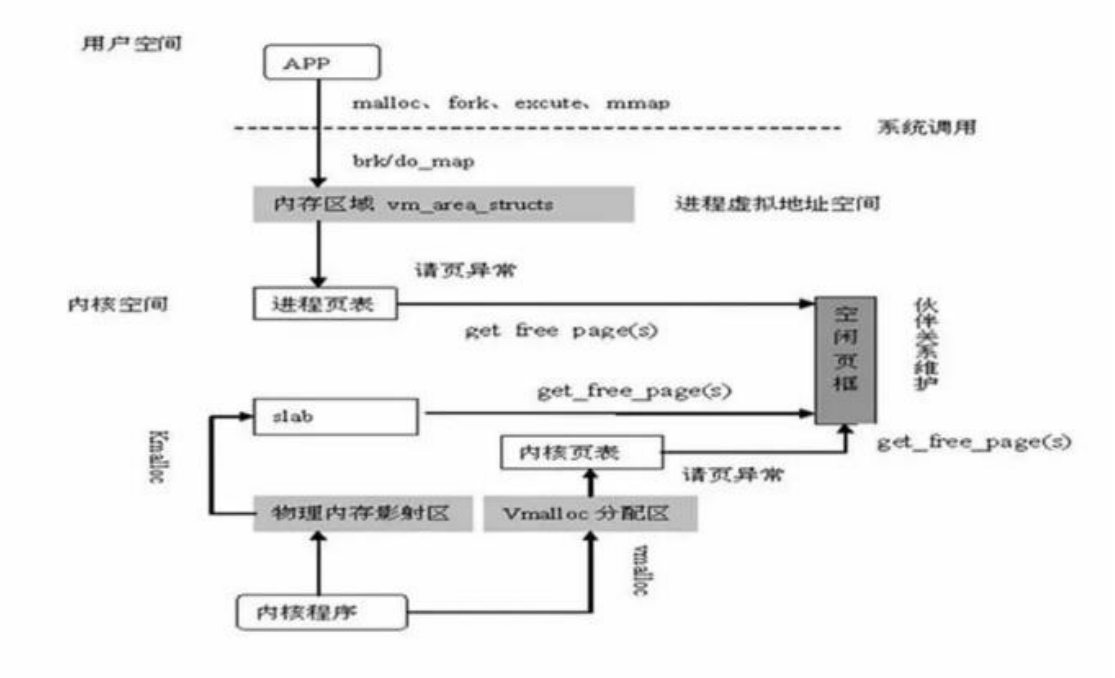

## 5、缺页异常

●通过 get_free_pages 申请一个或多个物理页面

●换算 addr 在进程 pdg 映射中所在的 pte 地址

●将 addr 对应的 pte 设置为物理页面的首地址

●系统调用：Brk—申请内存小于等于 128kb，do_map—申请内存大于 128kb

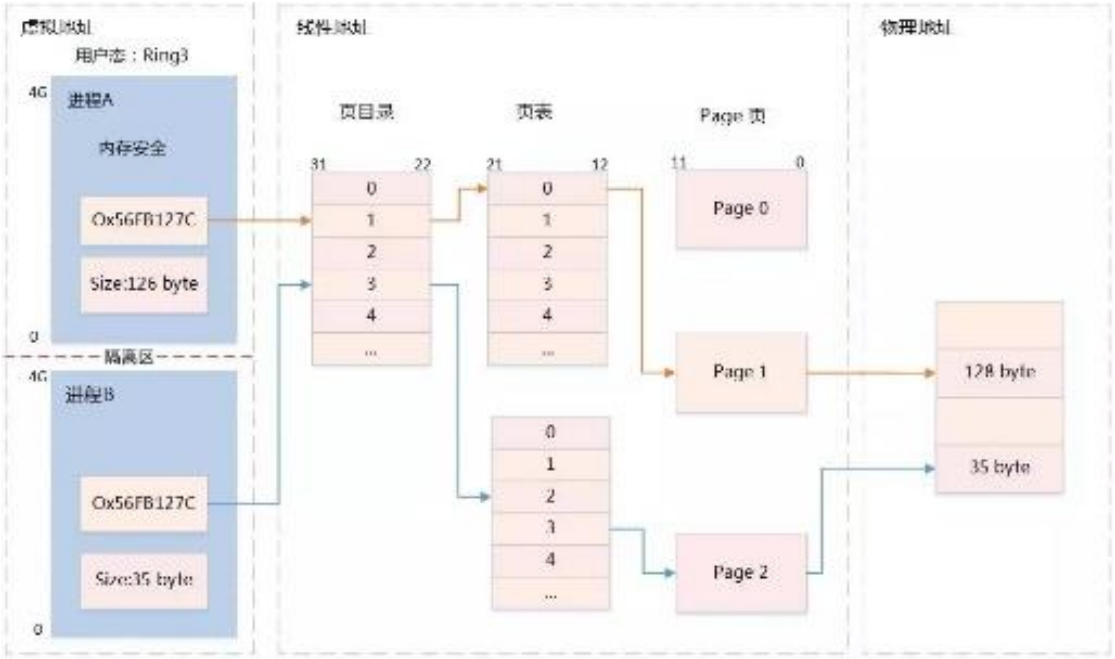



## 6、用户进程访问内存分析

●用户态进程独占虚拟地址空间，两个进程的虚拟地址可相同

●在访问用户态虚拟地址空间时，如果没有映射物理地址，通过系统调用发出缺页异常

●缺页异常陷入内核，分配物理地址空间，与用户态虚拟地址建立映射



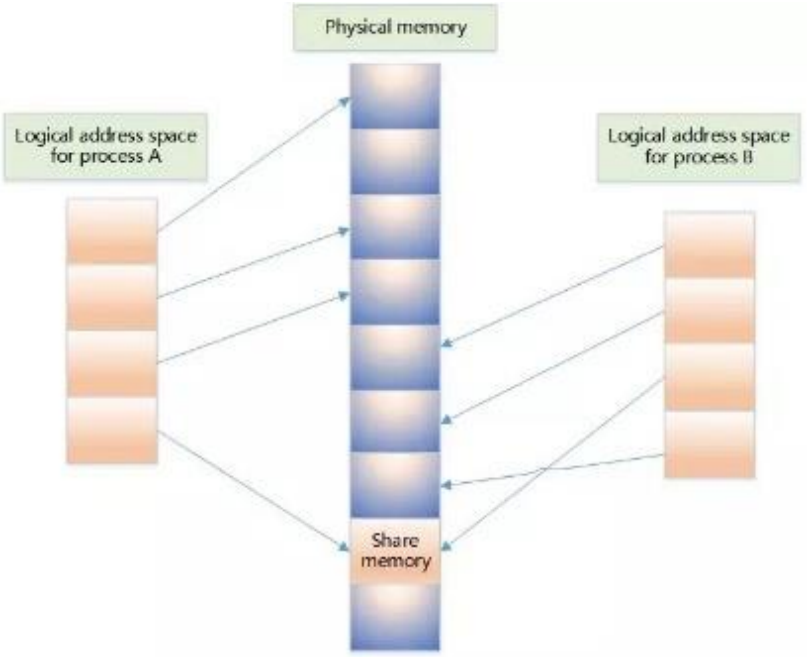

## 7、共享内存

1) 原理

●它允许多个不相关的进程去访问同一部分逻辑内存

●两个运行中的进程之间传输数据，共享内存将是一种效率极高的解决方案

●两个运行中的进程共享数据，是进程间通信的高效方法，可有效减少数据拷贝的次数




2) shm 接口

●shmget 创建共享内存

●shmat 启动对该共享内存的访问，并把共享内存连接到当前进程的地址空间

●shmdt 将共享内存从当前进程中分离

## 1.5 五、 内存使用那些坑

### 1、C 内存泄露

● 在类的构造函数和析构函数中没有匹配地调用 new 和 delete 函数

```
//购买岛
Json::Value jRspData;
CApolloIslandLogicMT* p_worldFactoryMT= new CApolloIslandLogicMT;
p_worldFactoryMT->Init(tmpbuf.data(),tmpbuf.size(),uin,APOLLO_GOODS_TYPE_ISLAND,implat,);
ret=p_worldFactoryMT->BuyWorldItem(tmpbuf.data(),tmpbuf.size(),uin,seriesId,jRspData);
if(ret < 0){
    LOG_UINERROR(uin, "p_worldFactoryMT->BuyWorldItem fail. ret=%d, uin=%lu", ret, uin);
    //delete p_worldFactoryMT;
    MCRET(ret,"");
}
delete p_worldFactoryMT;
//p_worldFactoryMT=NULL;
```

●没有正确地清除嵌套的对象指针

●没有将基类的析构函数定义为虚函数

●当基类的指针指向子类对象时，如果基类的析构函数不是 virtual，那么子类的析构函数将不会被调用，子类的资源没有得到正确释放，因此造成内存泄露

●缺少拷贝构造函数，按值传递会调用（拷贝）构造函数，引用传递不会调用

●指向对象的指针数组不等同于对象数组，数组中存放的是指向对象的指针，不仅要释放每个对象的空间，还要释放每个指针的空间

●缺少重载赋值运算符，也是逐个成员拷贝的方式复制对象，如果这个类的大小是可变的，那么结果就是造成内存泄露

### 2、C 野指针

●指针变量没有初始化

●指针被 free 或 delete 后，没有设置为 NULL

●指针操作超越了变量的作用范围，比如返回指向栈内存的指针就是野指针

●访问空指针（需要做空判断）

●sizeof 无法获取数组的大小

●试图修改常量，如：char  p="1234";p='1';

## 3、C 资源访问冲突

●多线程共享变量没有用 valotile 修饰

●多线程访问全局变量未加锁

●全局变量仅对单进程有效

●多进程写共享内存数据，未做同步处理

●mmap 内存映射，多进程不安全

## 4、STL 迭代器失效

●被删除的迭代器失效

●添加元素（insert/push_back 等）、删除元素导致顺序容器迭代器失效

●错误示例：删除当前迭代器，迭代器会失效

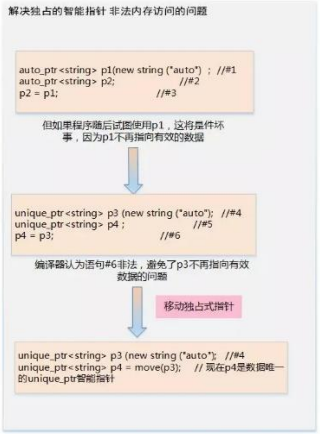
```
#include <iostream>
#include <vector>
using namespace std;
void main( ) {
    vector<int> vectInt;
    int i;
    //   初始化vector容器
    for (i = 0; i < 5; i++ ) {
        vectInt.push_back( i );
    }
    //   以下代码是要删除所有值为4的元素
    vector<int>::iterator itVect = vectInt.begin();
    for ( ; itVect != vectInt.end();  ++itVect ) {
        if ( *itVect == 4 ) {
            vectInt.erase( itVect );
        }
    }
    int iSize = vectInt.size();
    for (  i = 0 ; i < iSize; i++ )  {
            cout << " i= " <<i <<  "," << vectInt[ i ] << endl;
    }
}
```

- *正确示例：迭代器 erase 时，需保存下一个迭代器*

```
#include <iostream>
#include <vector>
using namespace std;
void main( ) {
    vector<int> vectInt;
    int i;
    for (  i = 0; i < 5; i++ ) {
        vectInt.push_back( i );
        if ( 3 == i ) {
            //     使3的元素有两个，并且相临。
            vectInt.push_back( i );
        }
    }
    vector<int>::iterator itVect = vectInt.begin();

    //   以下代码是要删除所有值为3的元素

    for ( ; itVect != vectInt.end();  ) {  // 删除 ++itVect;

        if ( *itVect == 3 ) {

            itVect = vectInt.erase( itVect );
        }
        else {

            ++itVect;
        }
    }
    //   把vectInt.size()放在for循环中

    for (  i = 0 ; i < vectInt.size(); i++ )  {
            cout << " i= " <<i <<  "," << vectInt[ i ] << endl;
    }
```

## 5、C++ 11 智能指针

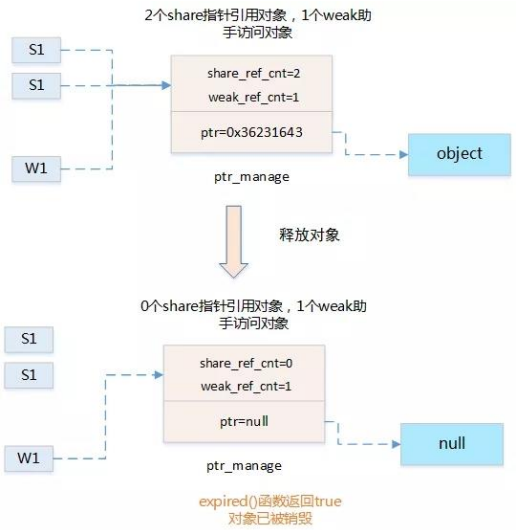
- *auto_ptr 替换为 unique_ptr*




- *使用 make_shared 初始化一个 shared_ptr*

1.性能：当您创建新的对象，然后创建一个shared_ptr，有这种情况发生两个动态内存分配：一个是来自新对象本身，再进行第二次由shared_ptr的构造函数创建的管理器对象。
2. 当你使用make_shared，C ++编译器的单一内存分配大到足以容纳两个管理对象和新对象

- *weak_ptr 智能指针助手（1）原理分析：*




（2）数据结构：

boost::weak_ptr

boost::weak_ptr<T>是boost提供的一个弱引用的智能指针，它的声明可以简化如下：

```
namespace boost {

  template<typename T> class weak_ptr {
  public:
    template <typename Y>
    weak_ptr(const shared_ptr<Y>& r);

    weak_ptr(const weak_ptr& r);

    ~weak_ptr();

    T* get() const;                 //1. 用于访问智能指针对象
    bool expired() const;           //2. 用于检测所管理的对象是否已经释放
    shared_ptr<T> lock() const;     //3. 用于获取所管理的对象的强引用指针
  };
}
```

（3）使用方法：

a. lock() 获取所管理的对象的强引用指针

b. expired() 检测所管理的对象是否已经释放 c. get() 访问智能指针对象

## 6、C++ 11 更小更快更安全

- std::atomic 原子数据类型 多线程安全
- std::array 定长数组开销比 array 小和 std::vector 不同的是 array 的长度是固定的，不能动态拓展
- std::vector vector 瘦身 shrink_to_fit()：将 capacity 减少为于 size() 相同的大小
- td::forward_list

forward_list 是单链表（std::list 是双链表），只需要顺序遍历的场合，forward_list 能更加节省内存，插入和删除的性能高于 list

- std::unordered_map、std::unordered_set 用 hash 实现的无序的容器，插入、删除和查找的时间复杂度都是 O(1)，在不关注容器内元素顺序的场合，使用 unordered 的容器能获得更高的性能六、 如何查看内存
- 系统中内存使用情况：/proc/meminfo

```
/usr/local/services]$ cat /proc/meminfo
MemTotal:        8045260 kB
MemFree:          183616 kB
Buffers:          363700 kB
Cached:          4448656 kB
SwapCached:            0 kB
Active:          4174176 kB
Inactive:        2481228 kB
Active(anon):    1949420 kB
Inactive(anon):   260964 kB
Active(file):    2224756 kB
Inactive(file):  2220264 kB
Unevictable:          12 kB
Mlocked:              12 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:               512 kB
Writeback:             0 kB
AnonPages:       1843120 kB
Mapped:           308940 kB
Shmem:            367328 kB
Slab:             267060 kB
SReclaimable:     234684 kB
SUnreclaim:        32376 kB
KernelStack:        1720 kB
PageTables:        13504 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     4022628 kB
Committed_AS:    3632540 kB
VmallocTotal:   34359738367 kB
VmallocUsed:       32880 kB
VmallocChunk:   34359694020 kB
HardwareCorrupted:     0 kB
AnonHugePages:    913408 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:       40952 kB
DirectMap2M:     8347648 kB
```

- 进程的内存使用情况：/proc/28040/status
- 查询内存总使用率：free

```
[user_00@VM_10_56_230_114_tencent_tlinux_release_1_2_final /usr/local/services]$ free
             total       used       free     shared    buffers     cached
Mem:       8045260    7877052     168208          0     363780    4458548
-/+ buffers/cache:    3054724    4990536
Swap:            0          0          0
```

- 查询进程 cpu 和内存使用占比：top
- 虚拟内存统计：vmstat
- 进程消耗内存占比和排序：ps aux  - sort -rss
- 释放系统内存缓存：/proc/sys/vm/drop_caches

To free pagecache, use echo 1 > /proc/sys/vm/drop_caches

To free dentries and inodes, use echo 2 > /proc/sys/vm/drop_caches

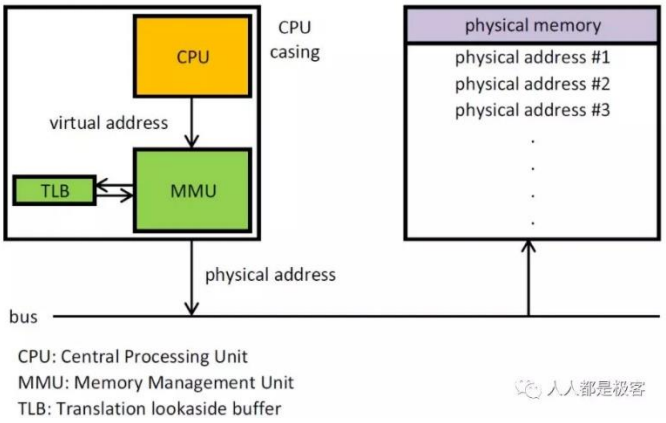
To free pagecache, dentries and inodes, use echo 3 >/proc/sys/vm/drop_caches

# 2 CPU 是如何访问内存的？

内存管理可以说是一个比较难学的模块，之所以比较难学。一是内存管理涉及到硬件的实现原理和软件的复杂算法，二是网上关于内存管理的解释有太多错误的解释。希望可以做个内存管理的系列，从硬件实现到底层内存分配算法，再从内核分配算法到应用程序内存划分，一直到内存和硬盘如何交互等，彻底理解内存管理的整个脉络框架。本节主要讲解硬件原理和分页管理。

## 2.1 CPU 通过 MMU 访问内存

我们先来看一张图：




从图中可以清晰地看出，CPU、MMU、DDR 这三部分在硬件上是如何分布的。首先 CPU 在访问内存的时候都需要通过 MMU 把虚拟地址转化为物理地址，然后通过总线访问内存。MMU 开启后 CPU 看到的所有地址都是虚拟地址，CPU 把这个虚拟地址发给 MMU 后，MMU 会通过页表在页表里查出这个虚拟地址对应的物理地址是什么，从而去访问外面的 DDR（内存条）。

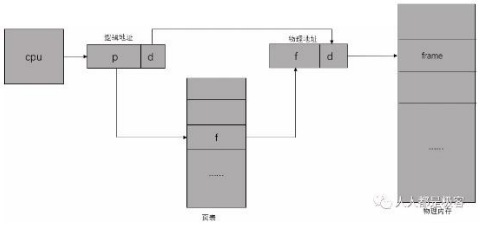
所以搞懂了 MMU 如何把虚拟地址转化为物理地址也就明白了 CPU 是如何通过 MMU 来访问内存的。

MMU 是通过页表把虚拟地址转换成物理地址，页表是一种特殊的数据结构，放在系统空间的页表区存放逻辑页与物理页帧的对应关系，每一个进程都有一个自己的页表。

CPU 访问的虚拟地址可以分为：p（页号），用来作为页表的索引；d（页偏移），该页内的地址偏移。现在我们假设每一页的大小是 4KB，而且页表只有一级，那么页表长成下面这个样子（页表的每一行是 32 个 bit，前 20 bit 表示页号 p，后面 12 bit 表示页偏移 d）：


CPU，虚拟地址，页表和物理地址的关系如下图：

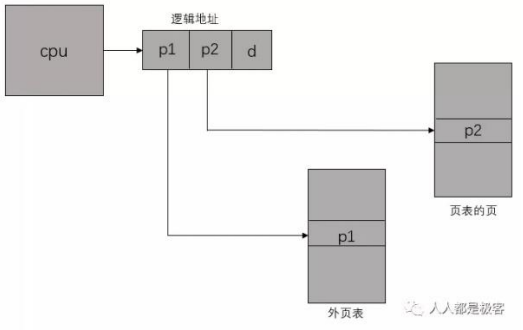
页表包含每页所在物理内存的基地址，这些基地址与页偏移的组合形成物理地址，就可送交物理单元。




上面我们发现，如果采用一级页表的话，每个进程都需要 1 个 4MB 的页表(假如虚拟地址空间为 32 位（即 4GB）、每个页面映射 4KB 以及每条页表项占 4B，则进程需要 1M 个页表项（4GB / 4KB = 1M），即页表（每个进程都有一个页表）占用 4MB（1M * 4B = 4MB）的内存空间)。然而对于大多数程序来说，其使用到的空间远未达到 4GB，何必去映射不可能用到的空间呢？也就是说，一级页表覆盖了整个 4GB 虚拟地址空间，但如果某个一级页表的页表项没

有被用到，也就不需要创建这个页表项对应的二级页表了，即可以在需要时才创建二级页表。做个简单的计算，假设只有20%的一级页表项被用到了，那么页表占用的内存空间就只有0.804MB（1K * 4B + 0.2 * 1K * 1K * 4B = 0.804MB）。除了在需要的时候创建二级页表外，还可以通过将此页面从磁盘调入到内存，只有一级页表在内存中，二级页表仅有一个在内存中，其余全在磁盘中（虽然这样效率非常低），则此时页表占用了 8KB（1K * 4B + 1 * 1K * 4B = 8KB），对比上一步的 0.804MB，占用空间又缩小了好多倍！总而言之，采用多级页表可以节省内存。

二级页表就是将页表再分页。仍以之前的 32 位系统为例，一个逻辑地址被分为 20 位的页码和 12 位的页偏移 d。因为要对页表进行再分页，该页号可分为 10 位的页码 p1 和 10 位的页偏移 p2。其中 p1 用来访问外部页表的索引，而 p2 是是外部页表的页偏移。



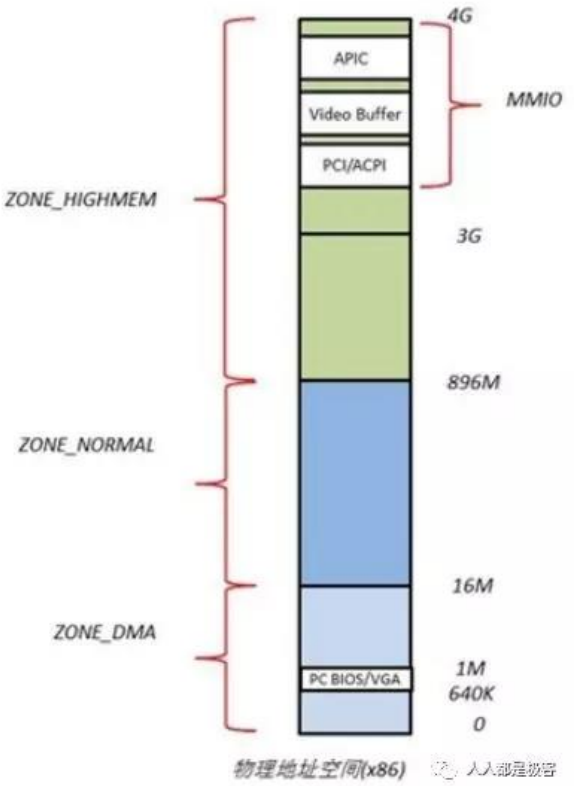

# 3 物理地址和虚拟地址的分布

上一节内容的学习我们知道了 CPU 是如何访问内存的，CPU 拿到内存后就可以向其它人（kernel 的其它模块、内核线程、用户空间进程、等等）提供服务，主要包括：

- 以虚拟地址（VA）的形式，为应用程序提供远大于物理内存的虚拟地址空间（Virtual Address Space）
- 每个进程都有独立的虚拟地址空间，不会相互影响，进而可提供非常好的内存保护（memory protection）
- 提供内存映射（Memory Mapping）机制，以便把物理内存、I/O 空间、Kernel Image、文件等对象映射到相应进程的地址空间中，方便进程的访问
- 提供公平、高效的物理内存分配（Physical Memory Allocation）算法
- 提供进程间内存共享的方法（以虚拟内存的形式），也称作 Shared Virtual Memory
- 在提供这些服务之前需要对内存进行合理的划分和管理，下面让我们看下是如何划分的。

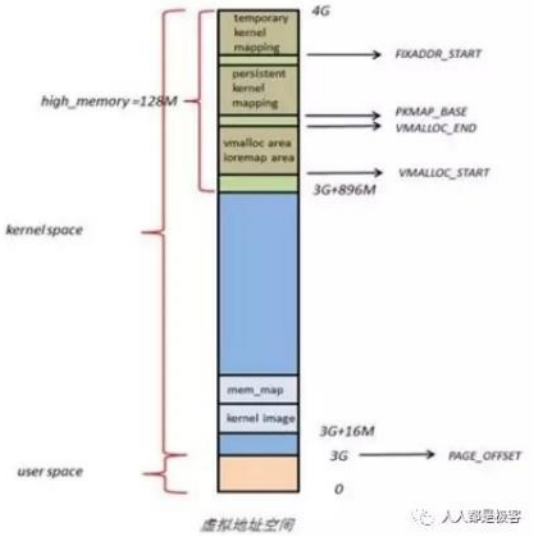
## 3.1 物理地址空间布局

Linux 系统在初始化时，会根据实际的物理内存的大小，为每个物理页面创建一个 page 对象，所有的 page 对象构成一个 mem_map 数组。进一步，针对不同的用途，Linux 内核将所有的物理页面划分到 3 类内存管理区中，如图，分别为 ZONE_DMA，ZONE_NORMAL，ZONE_HIGHMEM。




- ZONE_DMA 的范围是 0~16M，该区域的物理页面专门供 I/O 设备的 DMA 使用。之所以需要单独管理 DMA 的物理页面，是因为 DMA 使用物理地址访问内存，不经过 MMU，并且需要连续的缓冲区，所以为了能够提供物理上连续的缓冲区，必须从物理地址空间专门划分一段区域用于 DMA。
- ZONE_NORMAL 的范围是 16M~896M，该区域的物理页面是内核能够直接使用的。
- ZONE_HIGHMEM 的范围是 896M~结束，该区域即为高端内存，内核不能直接使用。

## 3.2 Linux 内核空间虚拟地址分布

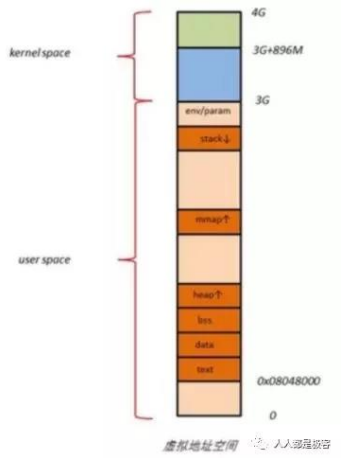



在 Kernel Image 下面有 16M 的内核空间用于 DMA 操作。位于内核空间高端的 128M 地址主要由 3 部分组成，分别为 vmalloc area、持久化内核映射区、临时内核映射区。

由于 ZONE_NORMAL 和内核线性空间存在直接映射关系，所以内核会将频繁使用的数据如 Kernel 代码、GDT、IDT、PGD、mem_map 数组等放在 ZONE_NORMAL 里。而将用户数据、页表（PT）等不常用数据放在 ZONE_HIGHMEM 里，

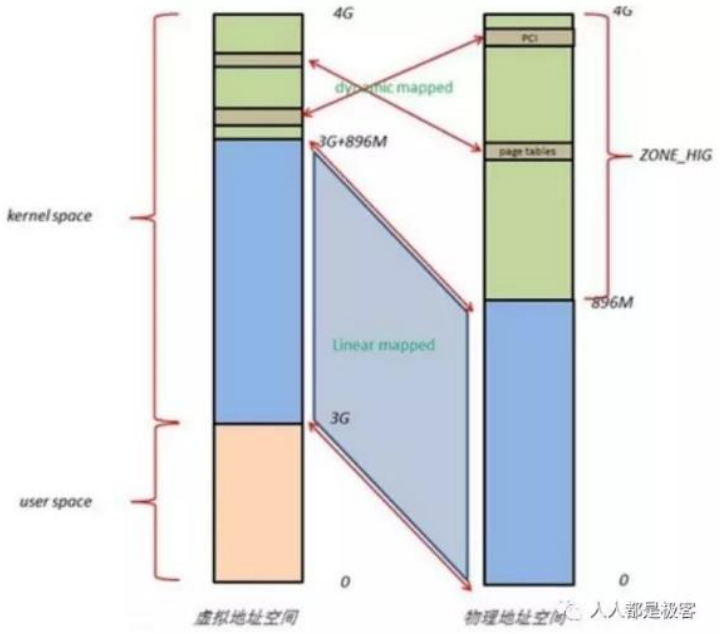
只在要访问这些数据时才建立映射关系（kmap()）。比如，当内核要访问 I/O 设备存储空间时，就使用 ioremap() 将位于物理地址高端的 mmio 区内存映射到内核空间的 vmalloc area 中，在使用完之后便断开映射关系。

## 3.3 Linux 用户空间虚拟地址分布




用户进程的代码区一般从虚拟地址空间的 0x08048000 开始，这是为了便于检查空指针。代码区之上便是数据区，未初始化数据区，堆区，栈区，以及参数、全局环境变量。

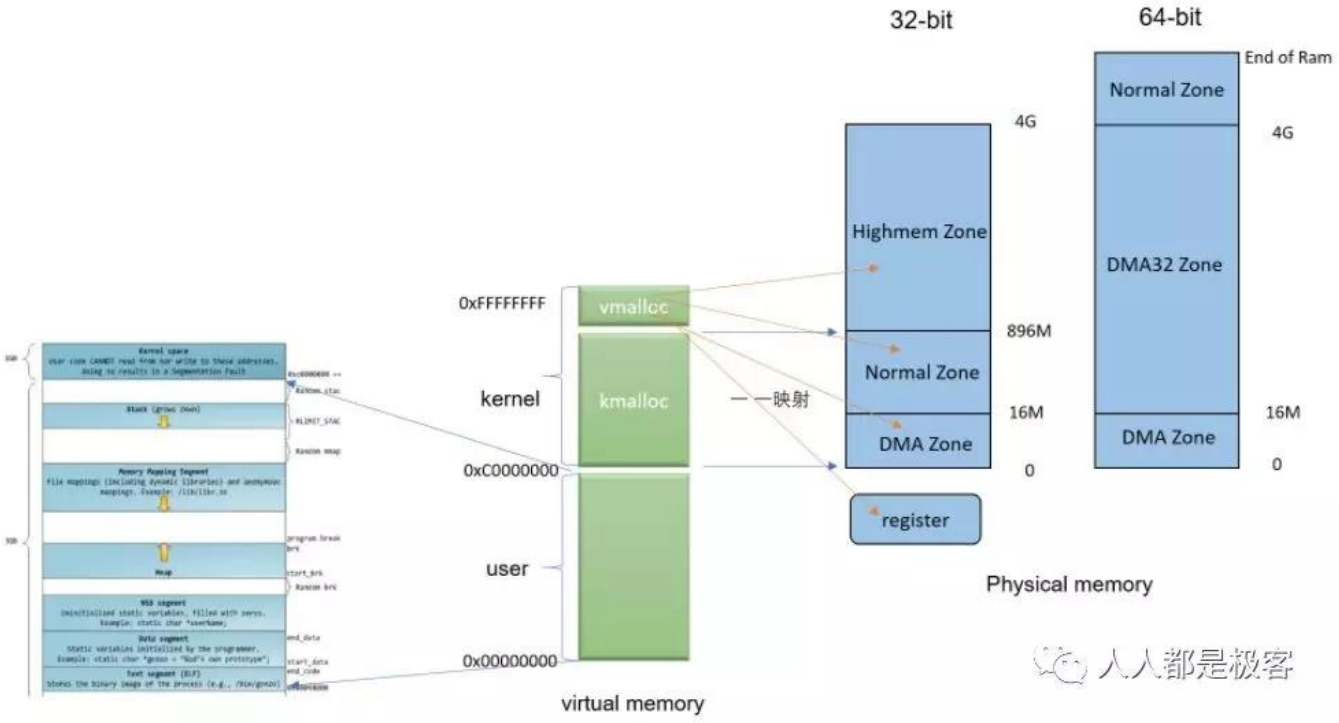
## 3.4 Linux 物理地址和虚拟地址的关系




Linux 将 4G 的线性地址空间分为 2 部分，0~3G 为 user space，3G~4G 为 kernel space。

由于开启了分页机制，内核想要访问物理地址空间的话，必须先建立映射关系，然后通过虚拟地址来访问。为了能够访问所有的物理地址空间，就要将全部物理地址空间映射到 1G 的内核线性空间中，这显然不可能。于是，内核将 0~896M 的物理地址空间一对一映射到自己的线性地址空间中，这样它便可以随时访问 ZONE_DMA 和 ZONE_NORMAL 里的物理页面；此时内核剩下的 128M 线性地址空间不足以完全映射所有的 ZONE_HIGHMEM，Linux 采取了动态映射的方法，即按需的将 ZONE_HIGHMEM 里的物理页面映射到 kernel space 的最后 128M 线性地址空间里，使用完之后释放映射关系，以供其它物理页面映射。虽然这样存在效率的问题，但是内核毕竟可以正常的访问所有的物理地址空间了。

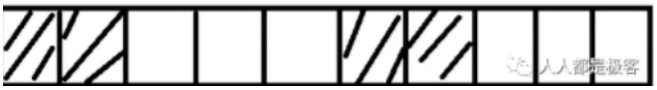
到这里我们应该知道了 Linux 是如何用虚拟地址来映射物理地址的，最后我们用一张图来总结一下：



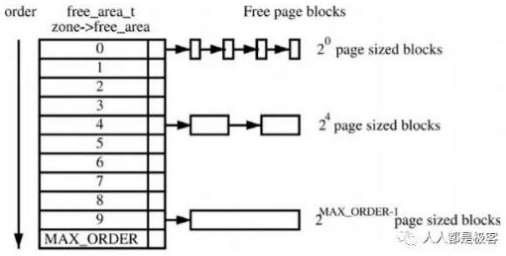

# 4 Linux 内核内存管理算法 Buddy 和 Slab

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。

## 4.1 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：

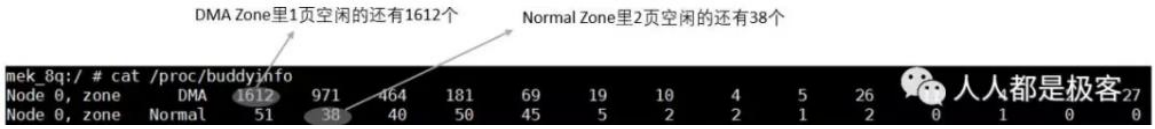



假设要申请一个 256 个页框的块，先从 256 个页框的链表中查找空闲块，如果没有，就去 512 个页框的链表中找，找到了则将页框块分为 2 个 256 个页框的块，一个分配给应用，另外一个移到 256 个页框的链表中。如果 512 个页框的链表中仍没有空闲块，继续向 1024 个页框的链表查找，如果仍然没有，则返回错误。页框块在释放时，会

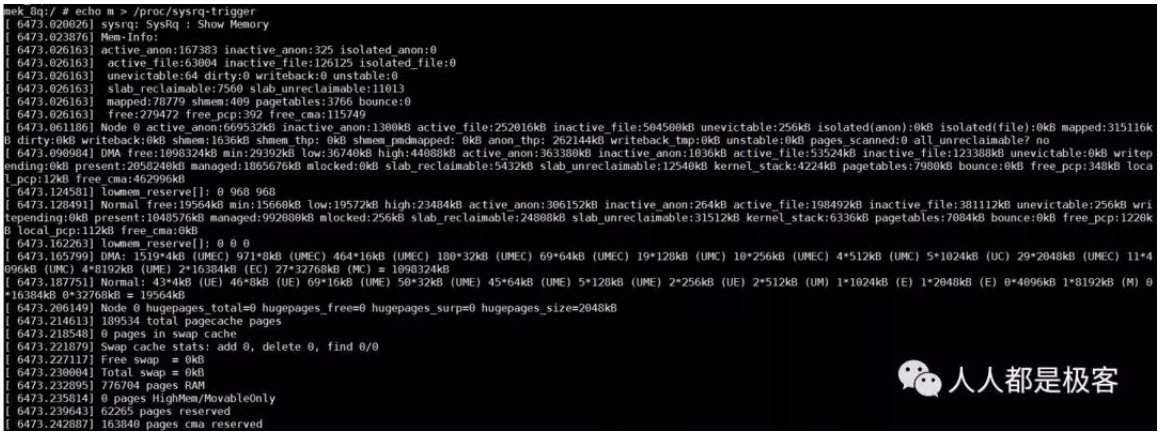
主动将两个连续的页框块合并为一个较大的页框块。

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：



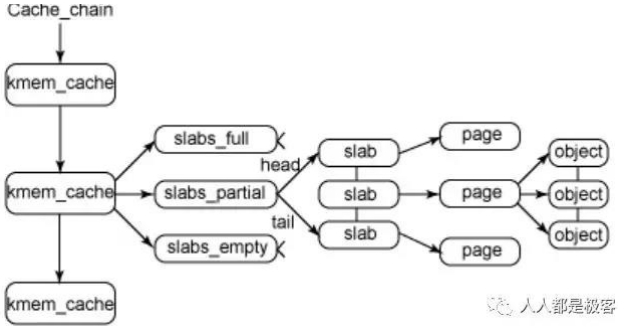

## 4.2 CMA

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。

## 4.3 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：

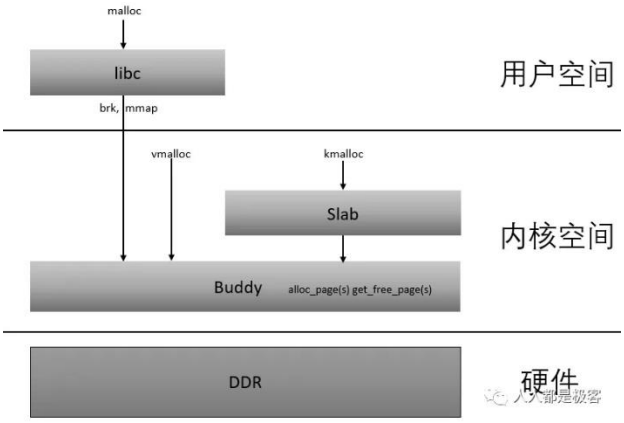
```
root@imx8qxpmek:~# cat /proc/slabinfo
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount> <sharedfacto
>
ip6-frags              0      0    200   20    1 : tunables    0    0    0 : slabdata      0      0      0
UDPv6                120    120   1088   30    8 : tunables    0    0    0 : slabdata      4      4      0
tw_sock_TCPv6          0      0    240   17    1 : tunables    0    0    0 : slabdata      0      0      0
request_sock_TCPv6     0      0    304   26    2 : tunables    0    0    0 : slabdata      0      0      0
TCPv6                 48     48   2048   16    8 : tunables    0    0    0 : slabdata      3      3      0
ext4_groupinfo_4k     28     28    144   28    1 : tunables    0    0    0 : slabdata      1      1      0
ubifs_inode_slab       0      0    720   22    4 : tunables    0    0    0 : slabdata      0      0      0
can_gw                 0      0    568   28    4 : tunables    0    0    0 : slabdata      0      0      0
isp1760_qtd            0      0     72   56    1 : tunables    0    0    0 : slabdata      0      0      0
isp1760_urb_listitem     0      0     24  170    1 : tunables    0    0    0 : slabdata      0      0      0
bsg_cmd                0      0    312   26    2 : tunables    0    0    0 : slabdata      0      0      0
mqueue_inode_cache    18     18    896   18    4 : tunables    0    0    0 : slabdata      1      1      0
```

```
dma-kmalloc-8192       0      0   8192    4    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-4096       0      0   4096    8    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-2048       0      0   2048   16    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-1024       0      0   1024   16    4 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-512        0      0    512   16    2 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-256        0      0    256   16    1 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-128        0      0    128   32    1 : tunables    0    0    0 : slabdata      0      0      0
kmalloc-8192          44     44   8192    4    8 : tunables    0    0    0 : slabdata     11     11      0
kmalloc-4096         165    176   4096    8    8 : tunables    0    0    0 : slabdata     22     22      0
kmalloc-2048        1904   1904   2048   16    8 : tunables    0    0    0 : slabdata    119    119      0
kmalloc-1024        1341   1360   1024   16    4 : tunables    0    0    0 : slabdata     85     85      0
```



## 4.4 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：



# 5 Linux 用户态进程的内存管理

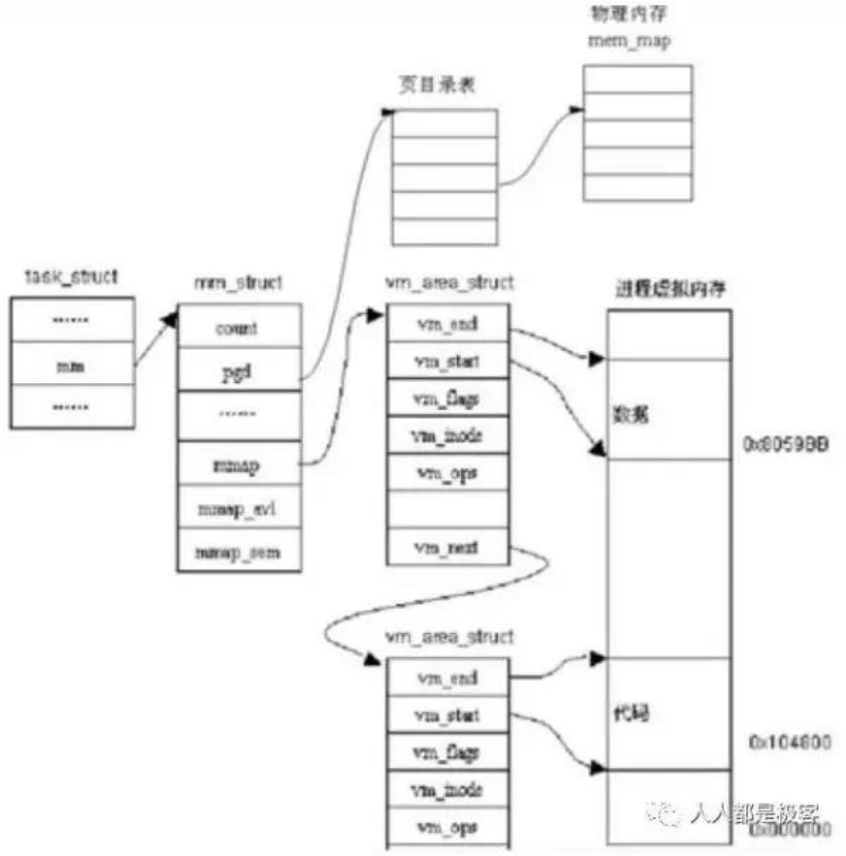
我们了解了内存在内核态是如何管理的，本篇文章我们一起来看下内存在用户态的使用情况，如果上一篇文章说是内核驱动工程师经常面对的内存管理问题，那本篇就是应用工程师常面对的问题。

相信大家都知道对用户态的内存消耗对象是进程，应用开发者面对的所有代码操作最后的落脚点都是进程，这也是说为什么内存和进程两个知识点的重要性，理解了内存和进程两大法宝，对所有软件开发的理解都会有了全局观（关于进程的知识以后再整理和大家分享）。

下面闲话少说，开始本篇的内容——进程的内存消耗和泄漏

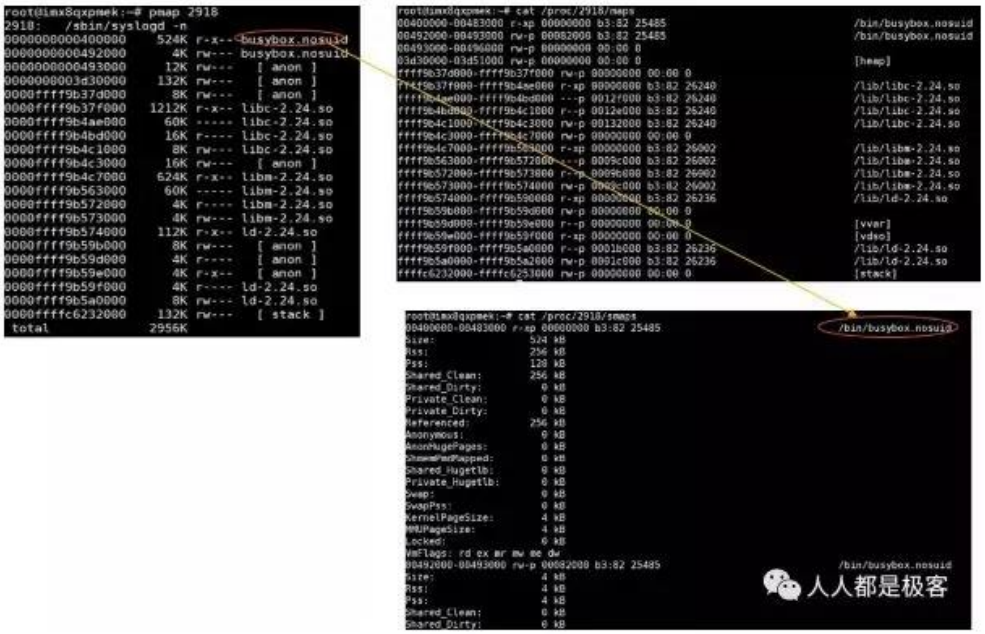
## 5.1 进程的虚拟地址空间 VMA（Virtual Memory Area）

在 linux 操作系统中，每个进程都通过一个 task_struct 的结构体描叙，每个进程的地址空间都通过一个 mm_struct 描叙，c 语言中的每个段空间都通过 vm_area_struct 表示，他们关系如下 ：

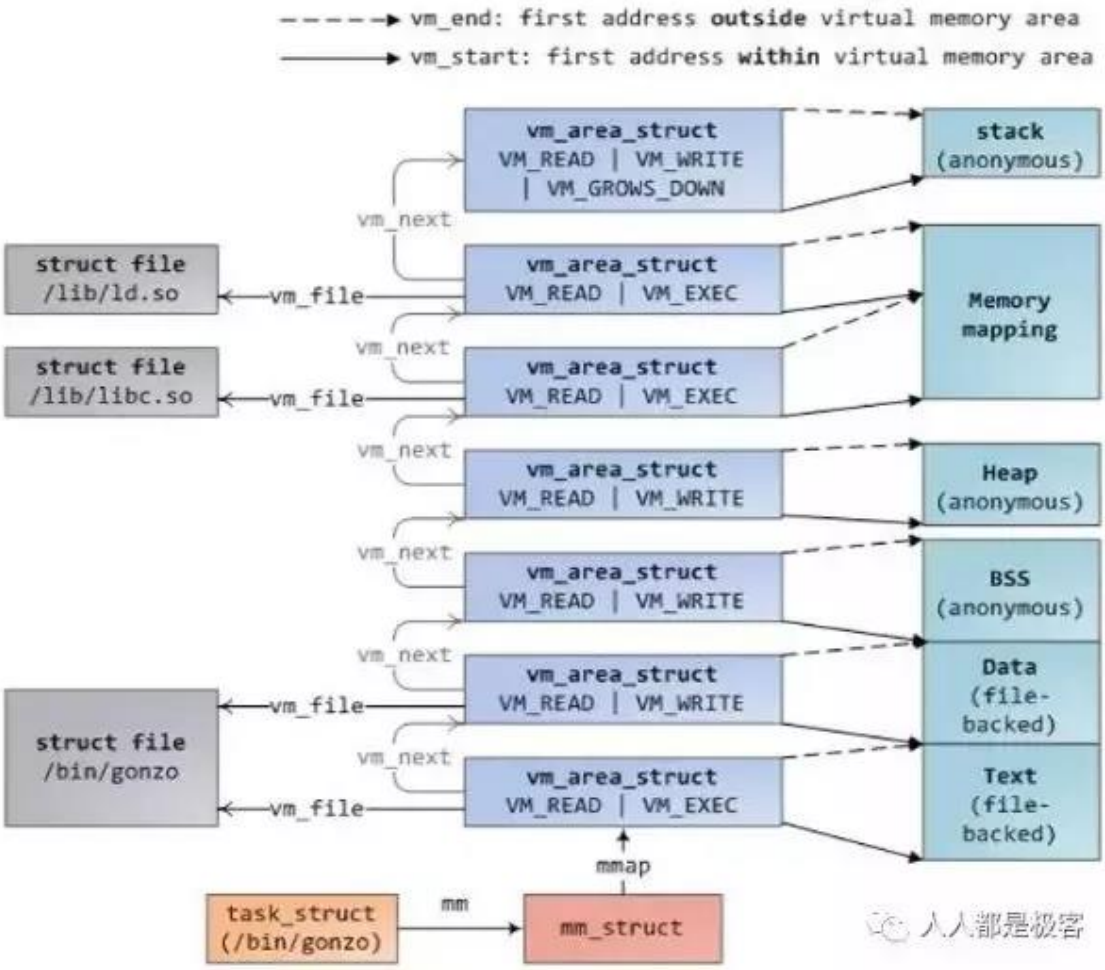



上图中，task_struct 中的 mm_struct 就代表进程的整个内存资源，mm_struct 中的 pgd 为页表，mmap 指针指向的 vm_area_struct 链表的每一个节点就代表进程的一个虚拟地址空间，即一个 VMA。一个 VMA 最终可能对应 ELF 可执行程序的数据段、代码段、堆、栈、或者动态链接库的某个部分。

VMA 的分布情况可以有通过 pmap 命令，及 maps，smaps 文件查看，如下图：




另，VMA 的具体内容可参考下图。




## 5.2 page fault 的几种可能性

我们先来看张图：


（此图来源于宋宝华老师）

如，调用 malloc 申请 100M 内存，IA32 下在 0~3G 虚拟地址中立刻就会占用到大小为 100M 的 VMA，且符合堆的定义，这一段 VMA 的权限是 R+W 的。但由于 Lazy 机制，这 100M 其实并没有获得，这 100M 全部映射到一个物理地址相同的零页，且在页表中记录的权限为只读的。当 100M 中任何一页发生写操作时，MMU 会给 CPU 发 page fault（MMU 可以从寄存器读出发生 page fault 的地址；MMU 可以读出发生 page fault 的原因），Linux 内核收到缺页中断，在缺页中断的处理程序中读出虚拟地址和原因，去 VMA 中查，发现是用户程序在写 malloc 的合法区域且有写权限，Linux 内核就真正的申请内存，页表中对应一页的权限也修改为 R+W。

如，程序中有野指针飞到了此程序运行时进程的 VMA 以外的非法区域，硬件就会收到 page fault，进程会收到 SIGSEGV 信号报段错误并终止。如，程序中有野指针飞到了此程序运行时进程的 VMA 以外的非法区域，硬件就会收到 page fault，进程会收到 SIGSEGV 信号报段错误并终止。

如，代码段在 VMA 中权限为 R+X，如果程序中有野指针飞到此区域去写，则也会发生段错误。（另，malloc 堆区在 VMA 中权限为 R+W，如果程序的 PC 指针飞到此区域去执行，同样发生段错误。）

如，执行代码段时会发生缺页，Linux 申请 1 页内存，并从硬盘读取出代码段，此时产生了 IO 操作，为 major 主缺页。如，执行代码段时会发生缺页，Linux 申请 1 页内存，并从硬盘读取出代码段，此时产生了 IO 操作，为 major 主缺页。

| 缺页 | |
|---|---|
| major主缺页 | minior次缺页 |
| 发生缺页必须读硬盘<br>如：上4）<br>有IO行为，处理时间长 | 发生缺页直接申请到内存<br>如：malloc内存，写时发生<br>无IO行为，处理时间相对major短 |

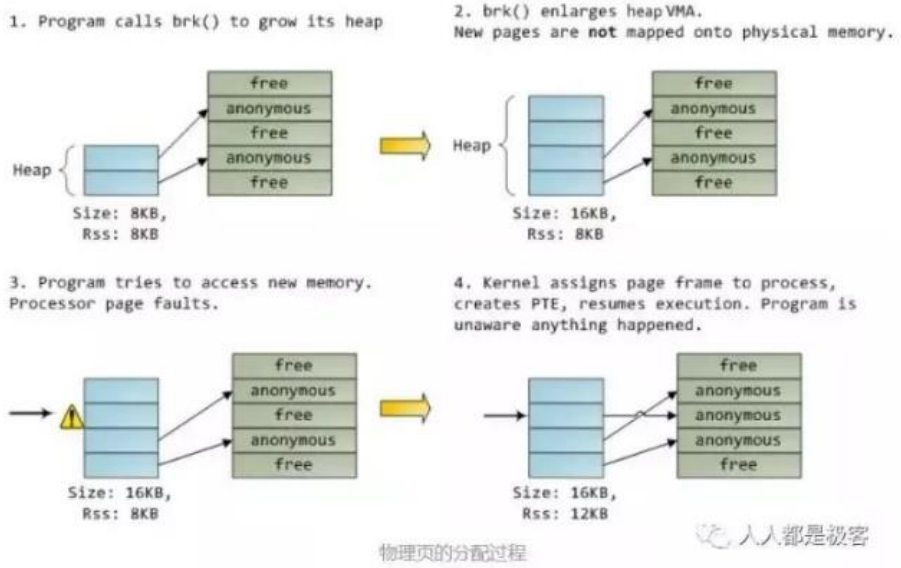
（此图来源于宋宝华老师）

综上，page fault 后，Linux 会查 VMA，也会比对 VMA 中和页表中的权限，体现出 VMA 的重要作用。

## 5.3 malloc 分配的原理

malloc 的过程其实就是把 VMA 分配到各种段当中，这时候是没有真正分配物理地址的。malloc 调用后，只是

分配了内存的逻辑地址，在内核的 mm_struct 链表中插入 vm_area_struct 结构体，没有分配实际的内存。当分配的区域写入数据是，引发页中断，建立物理页和逻辑地址的映射。下图表示了这个过程。



物理页的分配过程

从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：brk 和 mmap（不考虑共享内存）。

- malloc 小于 128k 的内存，使用 brk 分配内存，将_edata 往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系)
- malloc 大于 128k 的内存，使用 mmap 分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为 0)

## 5.4 内存的消耗 VSS RSS PSS USS

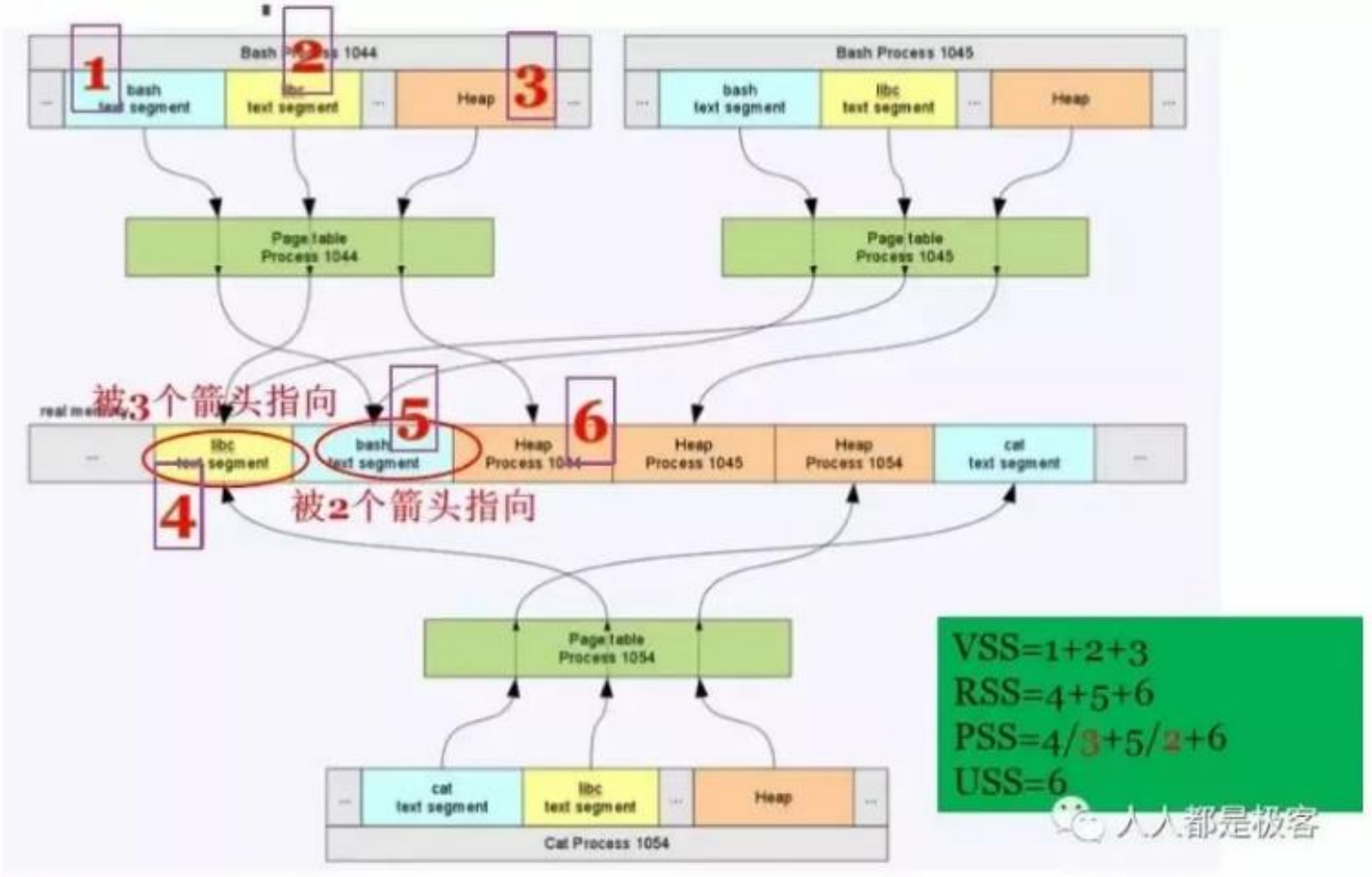
首先，我们评估一个进程的内存消耗都是指用户空间的内存，不包括内核空间的内存消耗 。这里我们用工具 procrank 先来看下 Linux 进程的内存占用量 。

```
# procrank
procrank
PID    Vss     Rss     Pss     Uss cmdline
481  31536K  30936K  14337K   9956K system_server
475  26128K  26128K  10046K   5992K zygote
526  25108K  25108K   9225K   5384K android.process.acore
523  22388K  22388K   7166K   3432K com.android.phone
574  21632K  21632K   6109K   2468K com.android.settings
521  20816K  20816K   6050K   2776K jp.co.omronsoft.openwnn
474   3304K   3304K   1097K    624K /system/bin/mediaserver
37    304K    304K    289K    288K /sbin/adbd
29    720K    720K    261K    212K /system/bin/rild
601    412K    412K    225K    216K procrank
  1    204K    204K    185K    184K /init
35    388K    388K    182K    172K /system/bin/qemud
284    384K    384K    160K    148K top
27    376K    376K    148K    136K /system/bin/vold
261    332K    332K    123K    112K logcat
33    396K    396K    105K     80K /system/bin/keystore
32    316K    316K    100K     88K /system/bin/installd
269    328K    328K     95K     72K /system/bin/sh
26    280K    280K     93K     84K /system/bin/servicemanager
45    304K    304K     91K     80K /system/bin/qemu-props
34    324K    324K     91K     68K /system/bin/sh
260    324K    324K     91K     68K /system/bin/sh
600    324K    324K     91K     68K /system/bin/sh
25    308K    308K     88K     68K /system/bin/sh
28    232K    232K     67K     60K /system/bin/debuggerd
```

- VSS -Virtual Set Size 虚拟耗用内存（包含共享库占用的内存）
- RSS -Resident Set Size 实际使用物理内存（包含共享库占用的内存）
- PSS -Proportional Set Size 实际使用的物理内存（比例分配共享库占用的内存）
- USS -Unique Set Size 进程独自占用的物理内存（不包含共享库占用的内存）

下面再用一张图来更好的解释 VSS,RSS,PSS,USS 之间的区别：


有了对 VSS,RSS,PSS,USS 的了解，我们趁热打铁来看下内存在进程中是如何被瓜分的：




（此图来源于宋宝华老师）

1044，1045，1054 三个进程，每个进程都有一个页表，对应其虚拟地址如何向 real memory 上去转换。

process 1044 的 1，2，3 都在虚拟地址空间，所以其 VSS=1+2+3。

process 1044 的 4，5，6 都在 real memory 上，所以其 RSS=4+5+6。

分析 real memory 的具体瓜分情况：

4 libc 代码段，1044，1045，1054 三个进程都使用了 libc 的代码段，被三个进程分享。

5 bash shell 的代码段，1044，1045 都是 bash shell，被两个进程分享。

6 1044 独占

所以，上图中 4+5+6 并不全是 1044 进程消耗的内存，因为 4 明显被 3 个进程指向，5 明显被 2 个进程指向，衍

生出了 PSS（按比例计算的驻留内存）的概念。进程 1044 的 PSS 为 4/3 +5/2 +6。

最后，进程 1044 独占且驻留的内存 USS 为 6。

一般来说内存占用大小有如下规律：VSS >= RSS >= PSS >= USS