北京郵電大學
Beijing University of Posts and Telecommunications

操作系统内核
-基于Linux

第5讲
内存管理

主讲：杨文川

内容

- 1 Linux内存管理机制
- 2 进程用户空间管理机制
- 3 物理内存分配与回收机制(上)
- 4 物理内存分配与回收机制(下)
- 5 动手实践-Linux内存映射

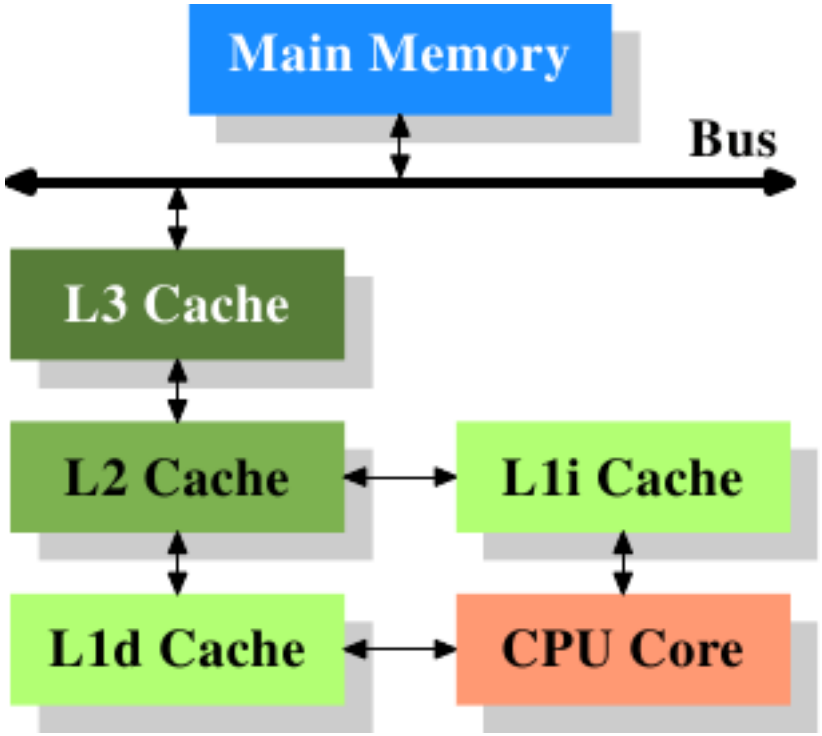
内容导航：

1 Linux内存管理机制

内存层次

- 内存(RAM)和外存相比，尽管内存比外存速度快很多，但还是无法与CPU的速度匹配，
- 因此CPU内部就需要更快的存储装置，这就是高速缓存(Cache)。

CPU中的Cache



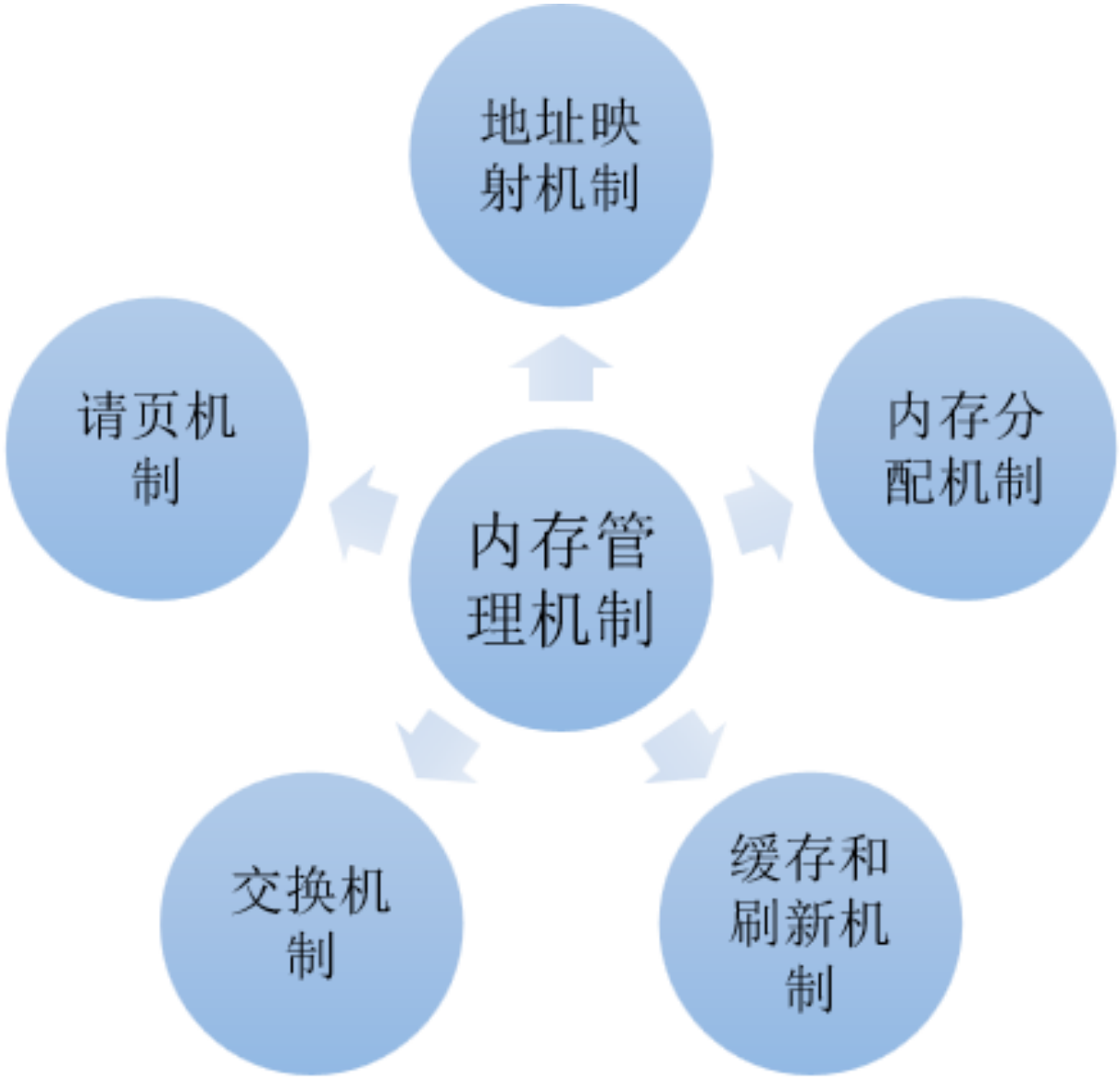
- 我们可以通过lscpu命令，查看内存的层次结构
- $lscpu

```
L1d cache:                       32K
L1i cache:                       32K
L2 cache:                        256K
L3 cache:                        3072K
NUMA node0 CPU(s):               0
```

- 从输出结果看，在x86机器上，L1d 和L1i cache，为一级数据和指令Cache，L2和L3 Cache为二级和三级Cache，大小各不同。

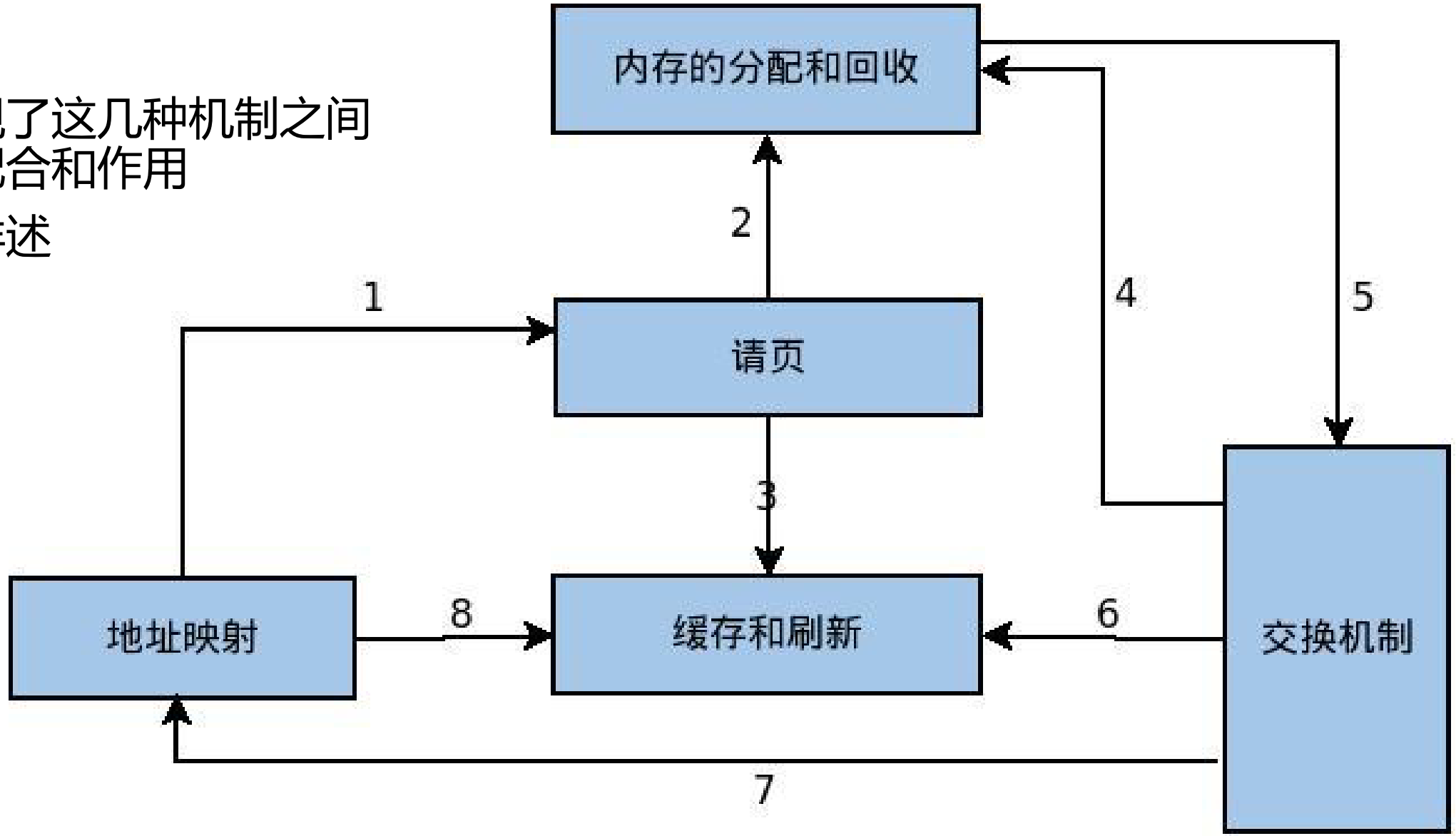
虚拟内存实现机制

- 那么Linux如何对虚拟内存进行管理，
- 我们总结出五种机制，它们分别是：



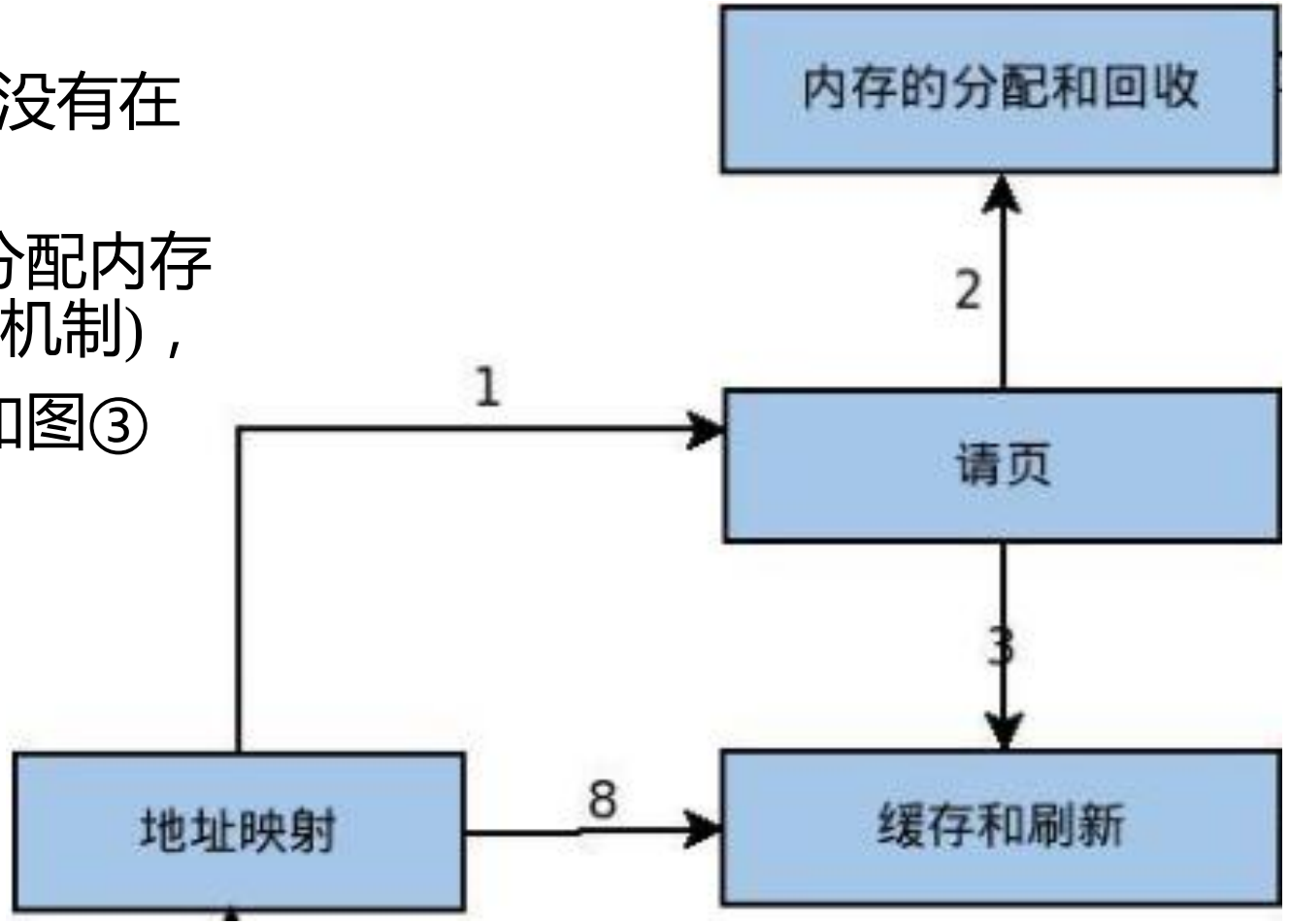
虚拟内存实现机制关系图

- 此图体现了这几种机制之间的相关配合和作用
- 下面将详述



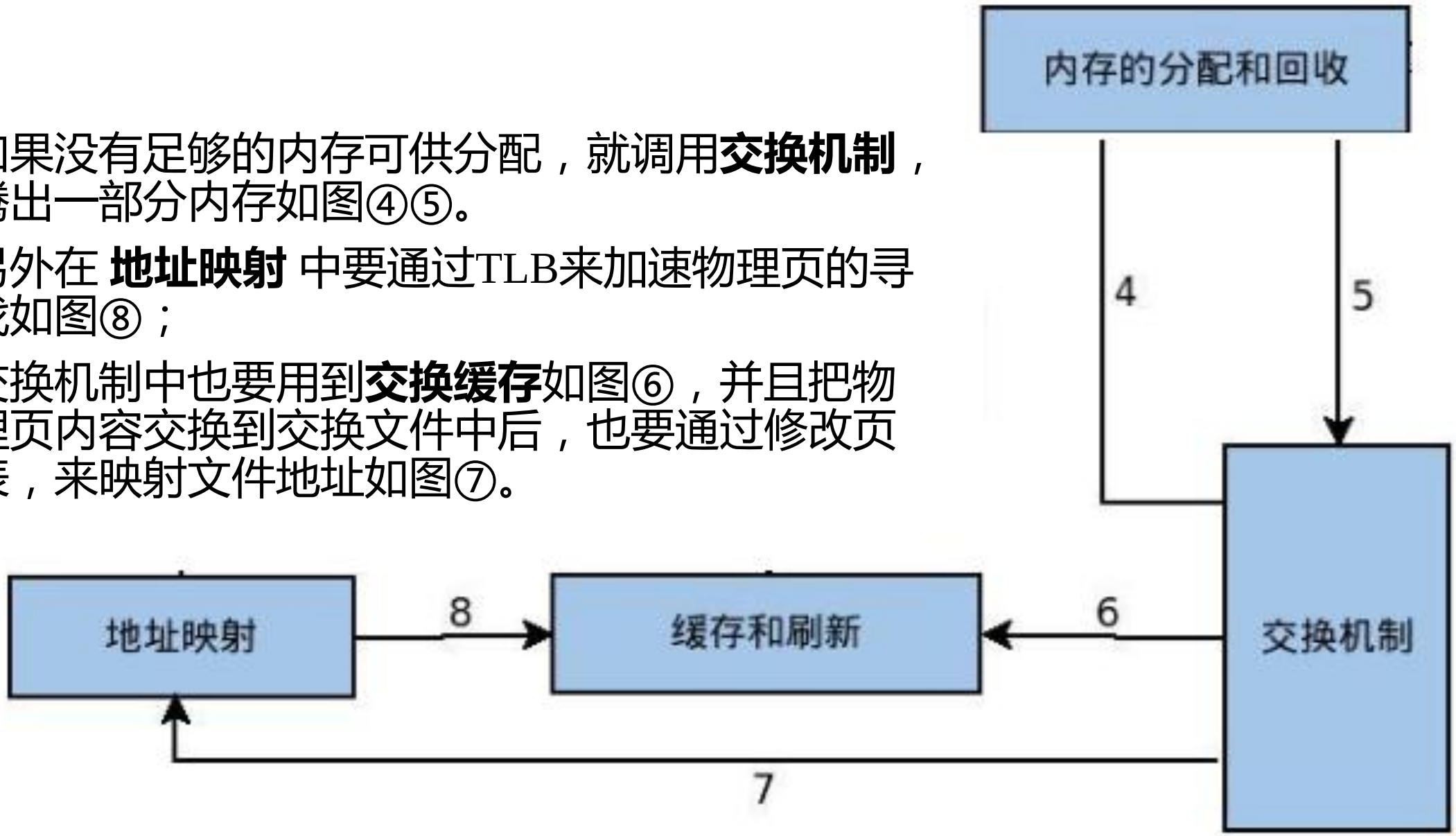
虚拟内存实现机制

- 首先内核通过映射机制，把进程从 **磁盘** 映射到 **虚拟地址空间**
- 当进程执行时，如果发现要访问的 **页** 没有在 **物理内存** 时，就发出 **请页** 如图①；
- 如果有空闲的内存可供分配，就请求分配内存如图②(于是用到了 **内存的分配和回收**机制)，
- 并把正在使用的 **页** 记录在 **页缓存** 中如图③(于是就使用了缓存机制)。



虚拟内存实现机制

- 如果没有足够的内存可供分配，就调用**交换机制**，腾出一部分内存如图④⑤。
- 另外在 **地址映射** 中要通过TLB来加速物理页的寻找如图⑧；
- 交换机制中也要用到**交换缓存**如图⑥，并且把物理页内容交换到交换文件中后，也要通过修改页表，来映射文件地址如图⑦。

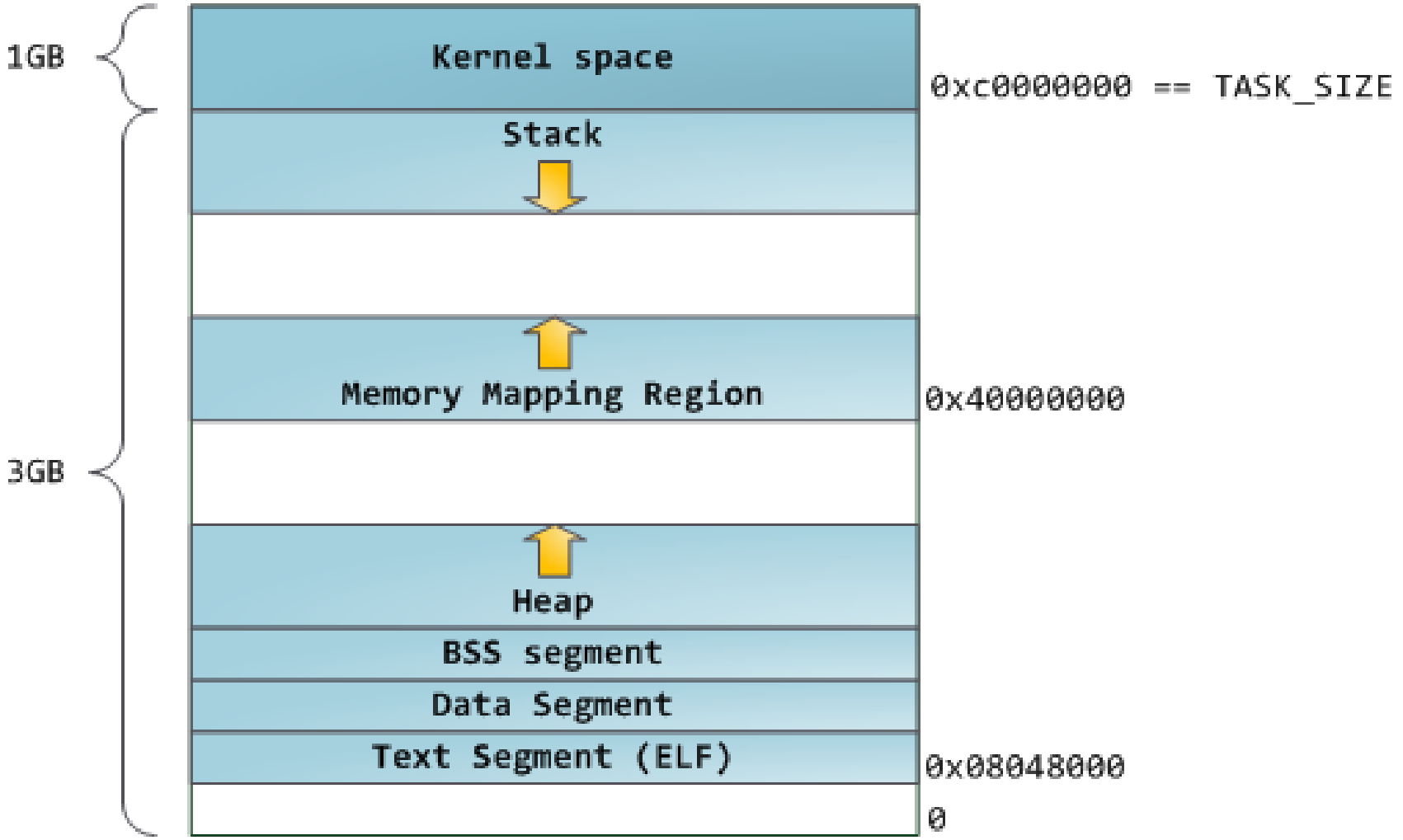


进程虚拟地址空间

- 程序一旦被执行就成为一个进程，内核就会为每个运行的进程提供了大小相同的虚拟地址空间，这使得多个进程可以同时运行而又不会互相干扰。
- 一个进程对某个地址的访问，绝不会干扰其他进程对同一地址的访问，下图是x86 的32位地址空间示意图。

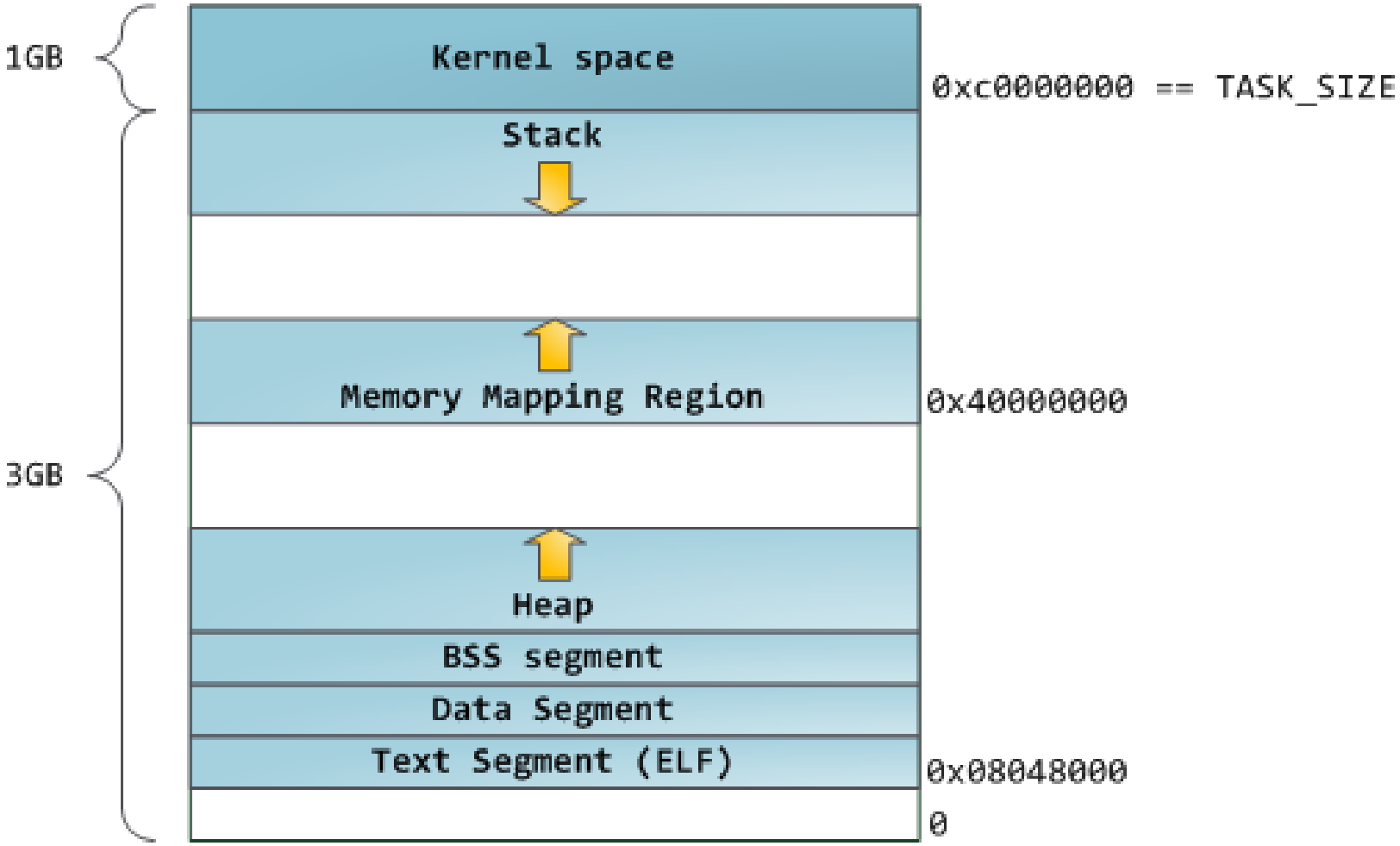
进程地址空间是如何布局的？

- 每个进程通过系统调用进入内核，Linux内核空间由系统内的所有进程共享。
- 从进程的角度看，每个进程拥有4GB的虚拟地址空间。
- 每个进程有各自的私有用户空间(0-3GB)，这个空间，对系统中的其他进程是不可见的。
- 最高的1GB内核空间为所有进程以及内核所共享。



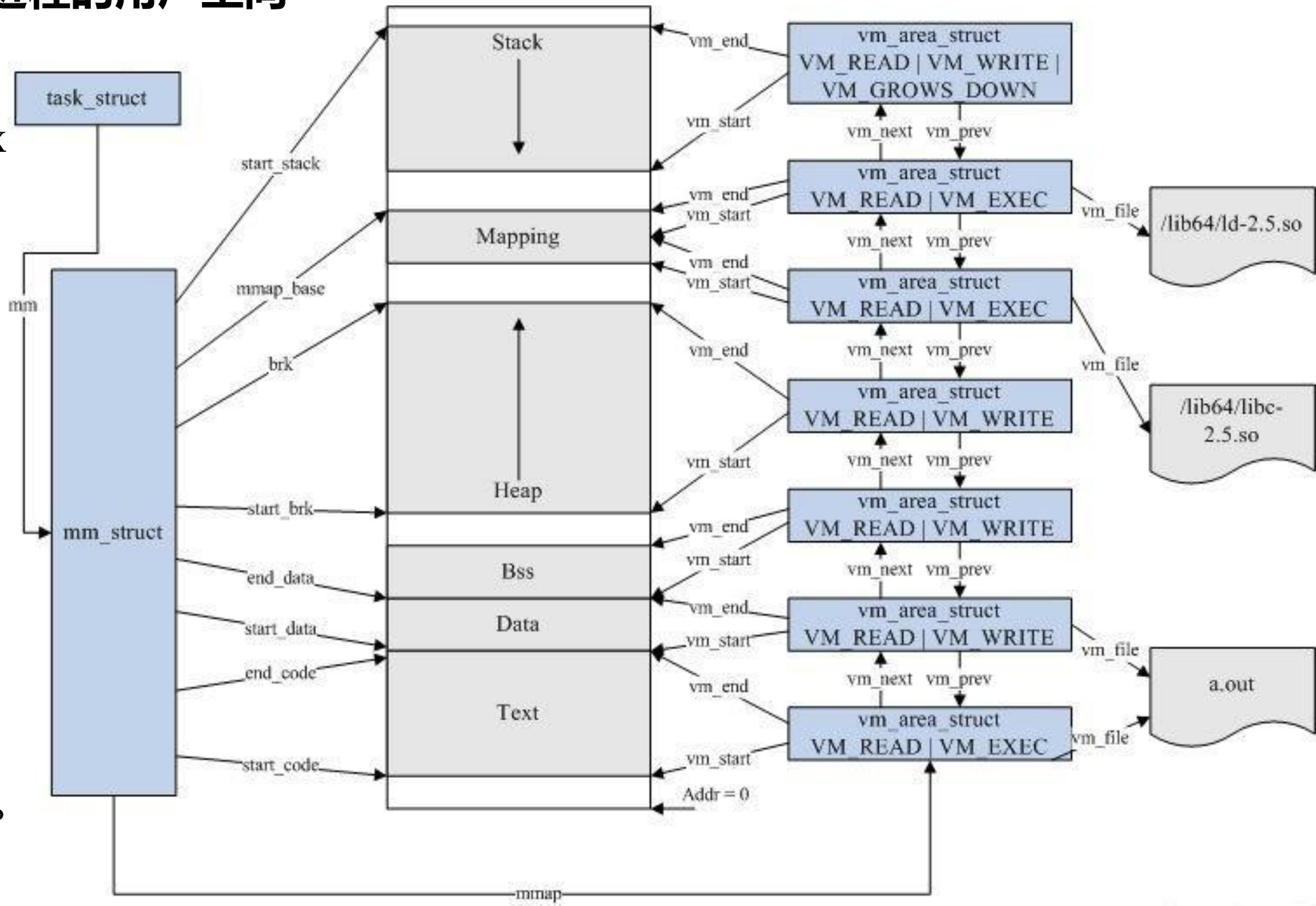
进程地址空间是如何布局的？

- 每个程序编译链接后形成的的二进制映像文件，有一个代码段(Text)和数据段(BSS和Data)。
- 进程运行时须有独占的堆(Heap)和栈(Stack)空间。
- 连接器和函数库都有自己的代码段(Text)和数据段(BSS和Data)。
- 进程要映射的文件被映射到内存映射区(Memory Mapping Region)。



如何用数据结构描述进程的用户空间

- 为便于管理，Linux把进程的用户空间划分为若干 **虚拟内存区域**(vma)。
- 进程 用户地址空间由mm_struct结构和 vm_area_structs结构 来描述。
- **mm_struct结构**对进程整个用户空间进行描述。
- **vm_area_structs结构**对用户空间中各个内存区进行描述。



mm_struct在源代码中的部分字段

- mm_struct定义在include/linux/mm_types.h

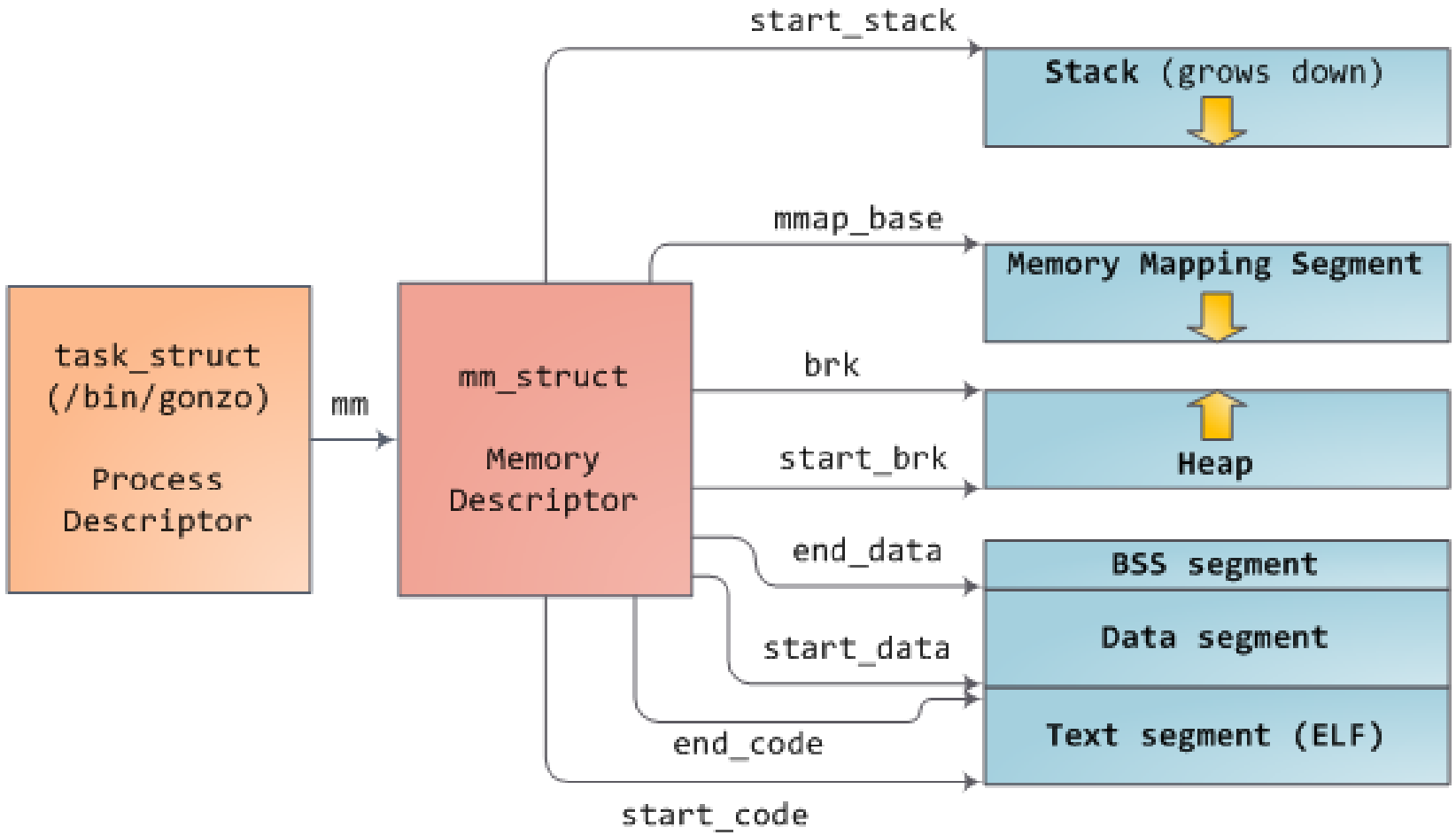
```
struct mm_struct {
        struct vm_area_struct * mmap;           / * list of VMAs */
        struct rb_root mm_rb;
        struct vm_area_struct * mmap_cache;     / * last find_vma result */
        unsigned long (*get_unmapped_area) (struct file *filp,
                        unsigned long addr, unsigned long len,
                        unsigned long pgoff, unsigned long flags);
        void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
        unsigned long mmap_base;                / * base of mmap area */
        unsigned long task_size;                / * size of task vm space */
        unsigned long cached_hole_size;             / * if non- zero, the largest hole
        unsigned long free_area_cache;              / * first hole of size cached_ho
        pgd_t * pgd;
        atomic_t mm_users;                  / * How many users with user space? */
        atomic_t mm_count;                  / * How many references to "struct mm_s
        int map_count;                      / * number of VMAs */
        struct rw_semaphore mmap_sem;
        spinlock_t page_table_lock;             / * Protects page tables and some c

        struct list_head mmlist;                / * List of maybe swapped mm's.
```

mm_struct基本字段的含义

- 下面对mm_struct结构中的一些字段含义给出简要描述，在这里只提及几个重要的字段：
- 一个进程的虚拟空间中可能有多个虚拟区间，对这些 虚拟区间的组织方式有两种：
- 当虚拟区较少时，采用 单链表，由**mmap指针**指向这个链表，
- 当虚拟区间多时，采用 红黑树结构，由**mm_rb**指向。
- **指针pgd**指向该进程的页目录，当调度程序调度一个 程序运行时，就将这个虚地址地址转成物理地址，并写 入控制寄存器(CR3)。
- 其他字段就不一一介绍了。

mm_struct如何描述地址空间

- 在进程的task_struct结构中，有一个字段mm指向mm_struct结构，
- mm_struct结构中各个区域的起始和结束字段，描述了进程地址空间的各个**虚存区(VMA)。**



虚存区(VM_AREA_STRUCT) 结构

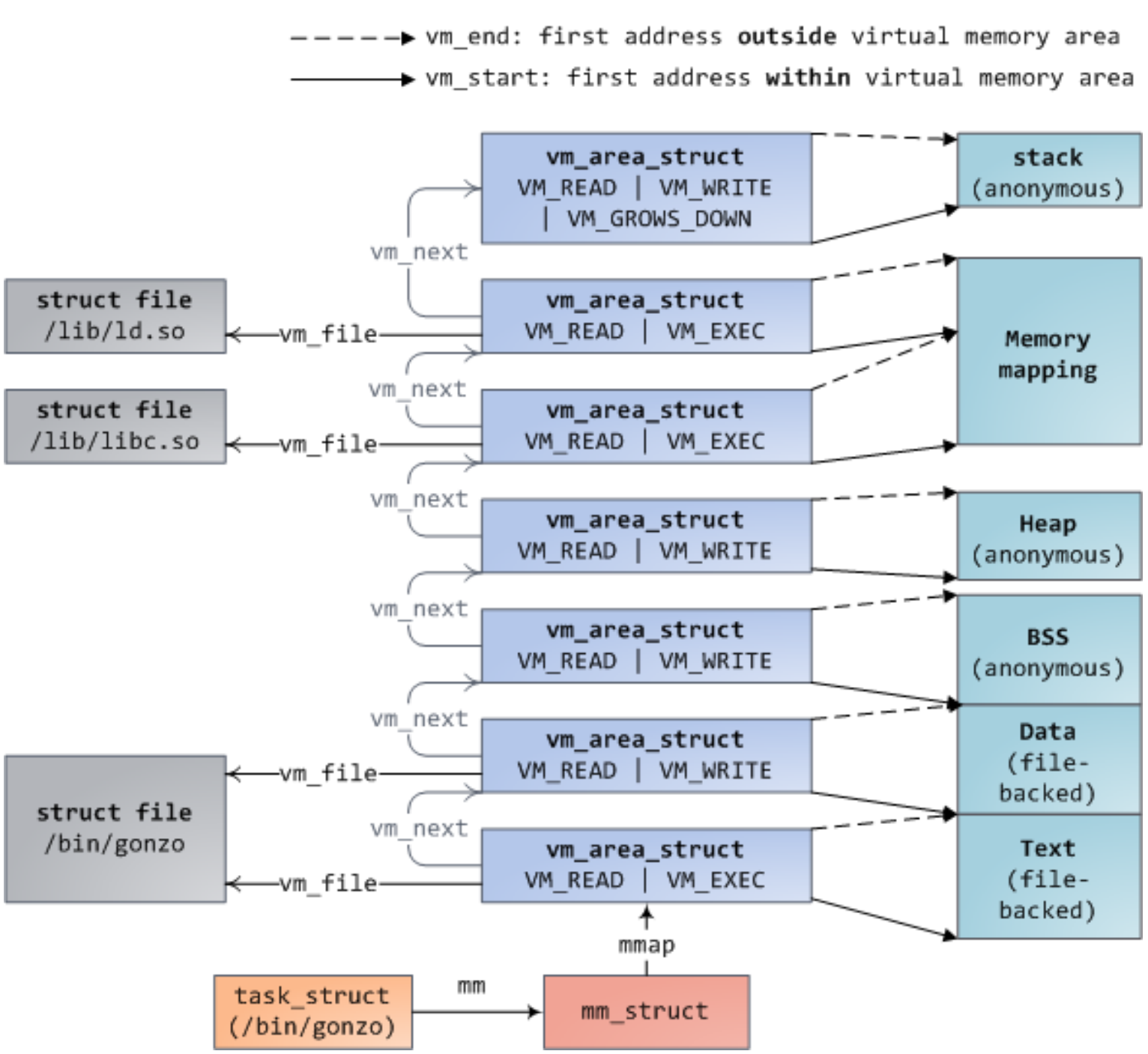
- vm_area_struct描述进程用户空间的一个**虚拟内存区间(**Virtual Memory Area, 简称VMA)，其定义如下：
  - struct vm_area_struct {
    - struct mm_struct * vm_mm;
    - unsigned long vm_start;
    - unsigned long vm_end;
    - struct vm_area_struct * vm_next;
    - pgprot_t vm_page_prot;
    - unsigned long vm_flags;
    - struct rb_node_t vm_rb;
    - struct vm_operations_struct * vm_ops; unsigned long vm_pgoff;
    - struct file * vm_file;
    - void * vm_private_data;
    - ……

在include/linux/mm_types.h

为什么要把进程的用户空间 划分为一个个区间？

- 这是因为每个虚存区可能来源不同： 有的可能来自 可执行映像，有的可能来自 共享库，而有的则可能是 动态分配的内存区，对不同的区间可能具有不同的访问权限，也可能有不同的操作。
- 因此Linux 把进程的用户空间分割管理，并利用了虚存区处理结构(vm_ops)，来抽象对不同来源虚存区的处理方法，其定义如下：
  - struct vm_operations_struct {
  - void (*open)(struct vm_area_struct * area);
  - void (*close)(struct vm_area_struct * area);
  - struct page * (*nopage)(struct vm_area_struct * area,
  - unsigned long address, int unused);
  - …;
  - };

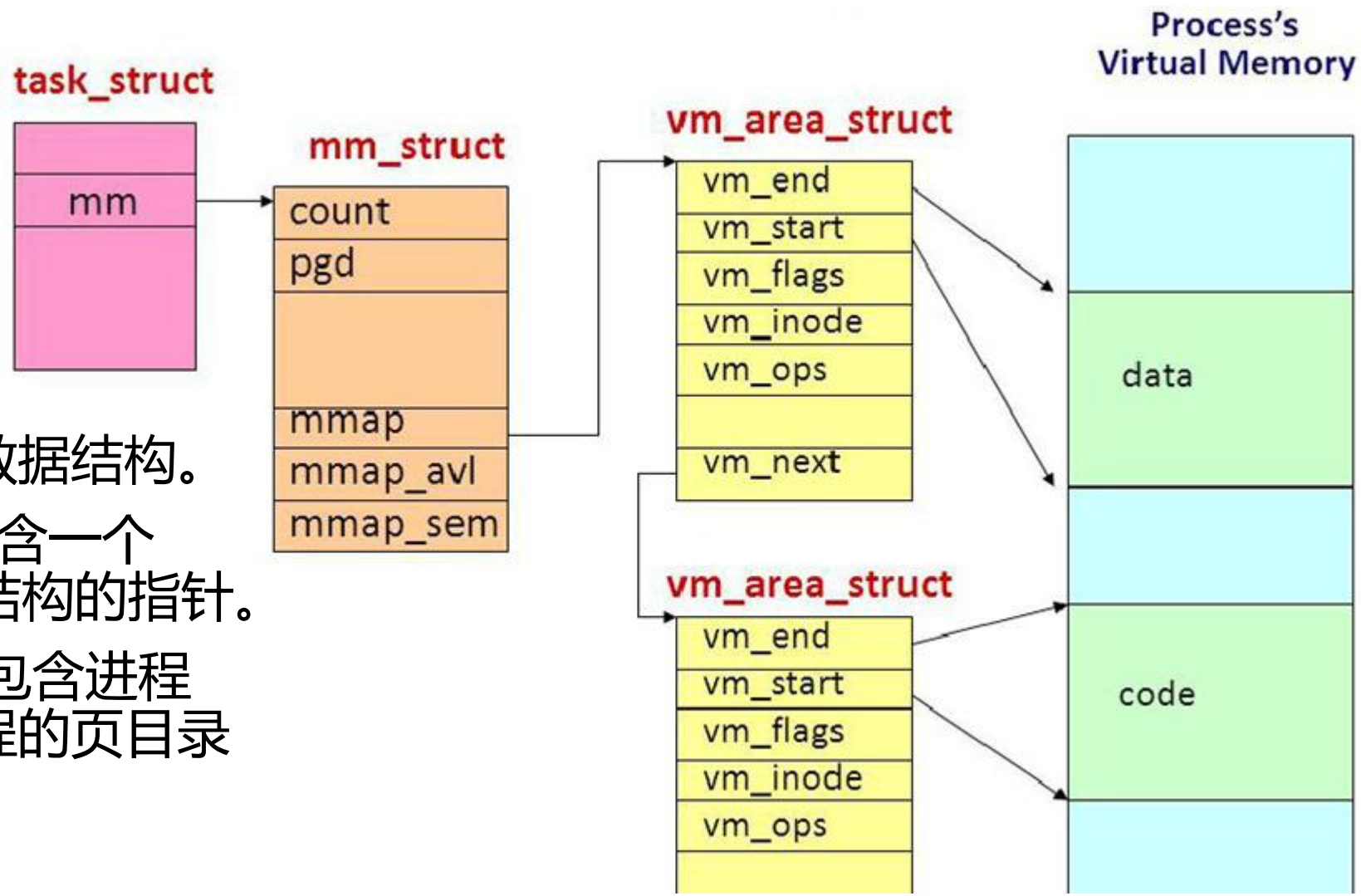
在include/linux/mm.h



内存区(VMA)如何映射到地址空间？

- 从这个图看出，mm_struct结构由一个个的VMA组成，
- 进程的代码段和数据段映射到Text段和Data段，
- 共享库(.so)映射到内存映射区。

相关数据结构的关系图



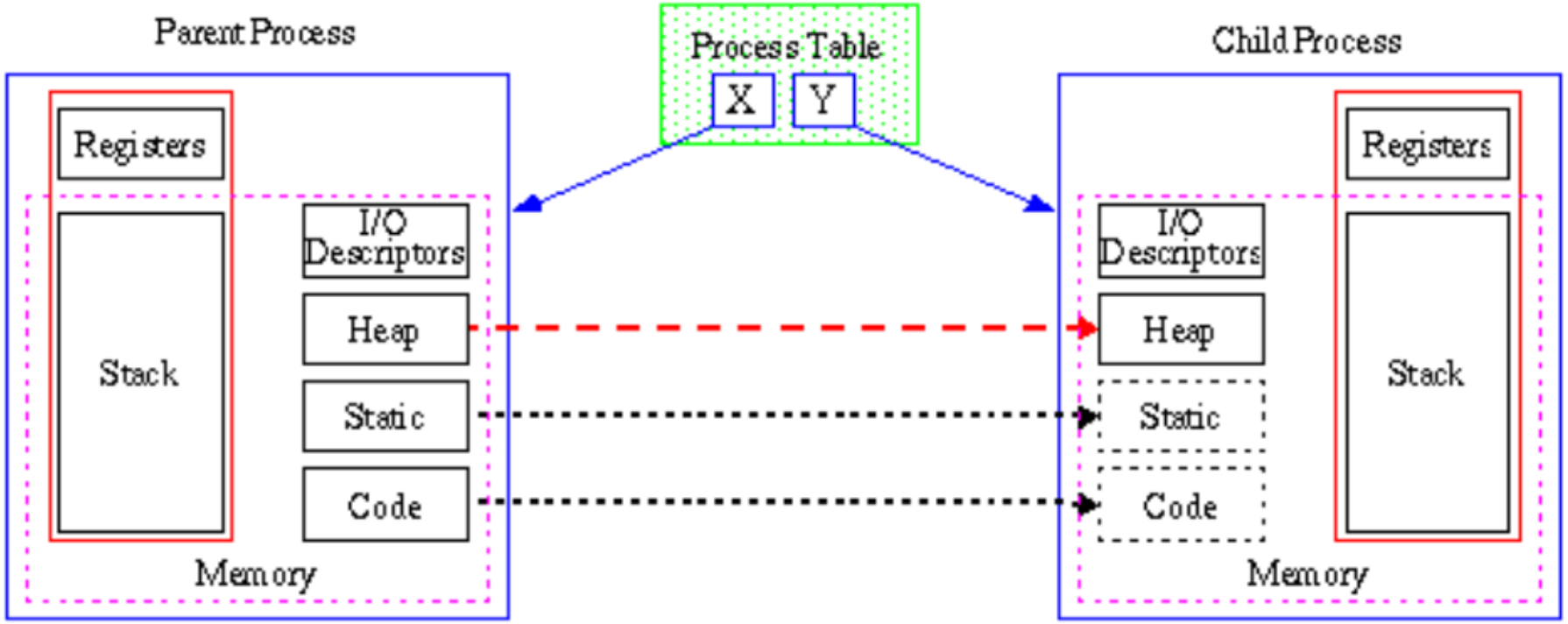
- 进程控制块是内核中的核心数据结构。
- 在进程的task_struct 结构中包含一个mm域，它是指向mm_struct 结构的指针。
- 而进程的mm_struct结构，则包含进程的可执行映像信息，以及进程的页目录指针pgd等。
- 该结构还包含有指向虚存区(vm_area_struct)结构的几个指针，每个VMA代表进程的一个虚拟地址区间。
- 这几个结构之间的关系如图所示。

内容导航：

2 进程用户空间的创建

创建进程用户空间

- 前面我们介绍了每个进程都有自己独立的地址空间，
- 那么，进程的地址空间到底是什么时候创建的？

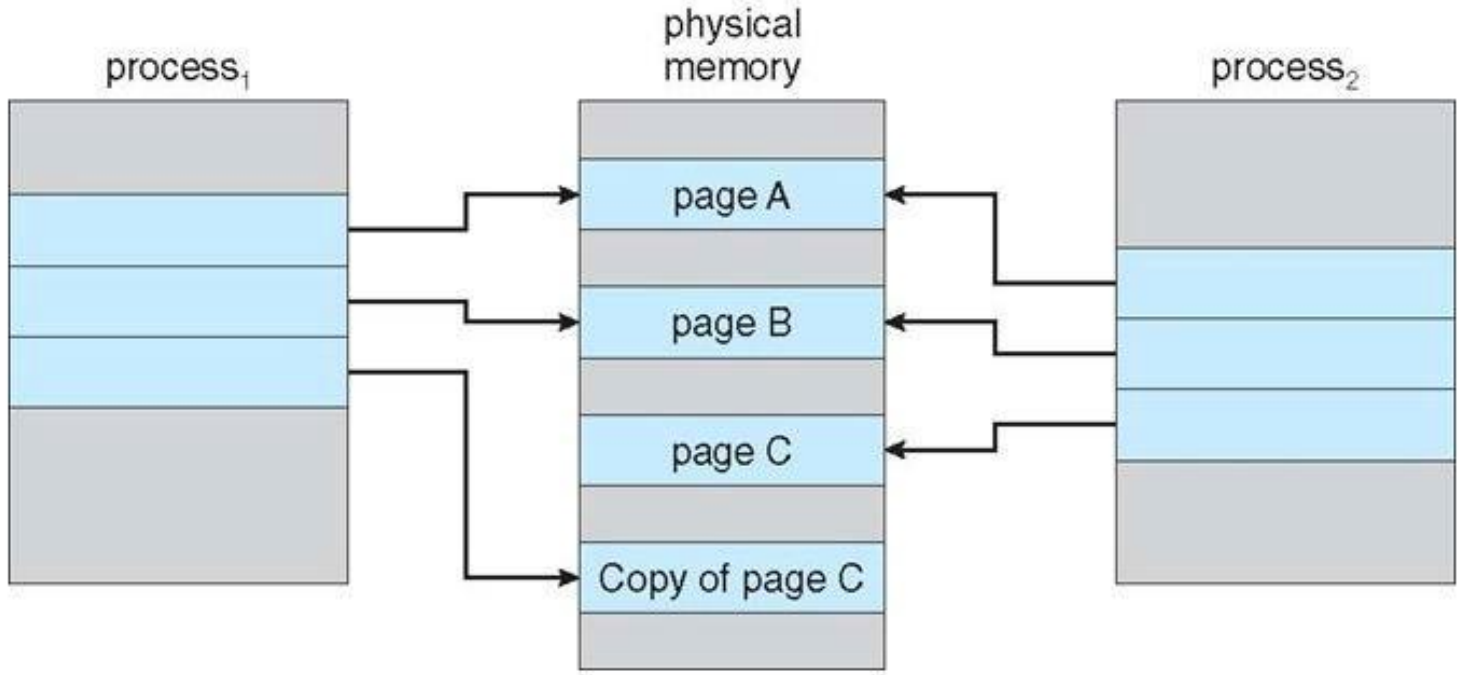


- 实际上，当fork()系统调用在创建新进程时，也为该进程创建完整的用户空间。
- 这个用户空间是如何被创建的？
- 　实际上是通过拷贝或共享父进程的用户空间来实现的，即内核调用copy_mm()函数，为新进程建立所有页表和mm_struct结构
- 　通常，每个进程都有自己的用户空间，但是调用clone( )函数创建内核线程时， 共享父进程的用户空间

- Linux利用“**写时复制**”技术来快速创建进程

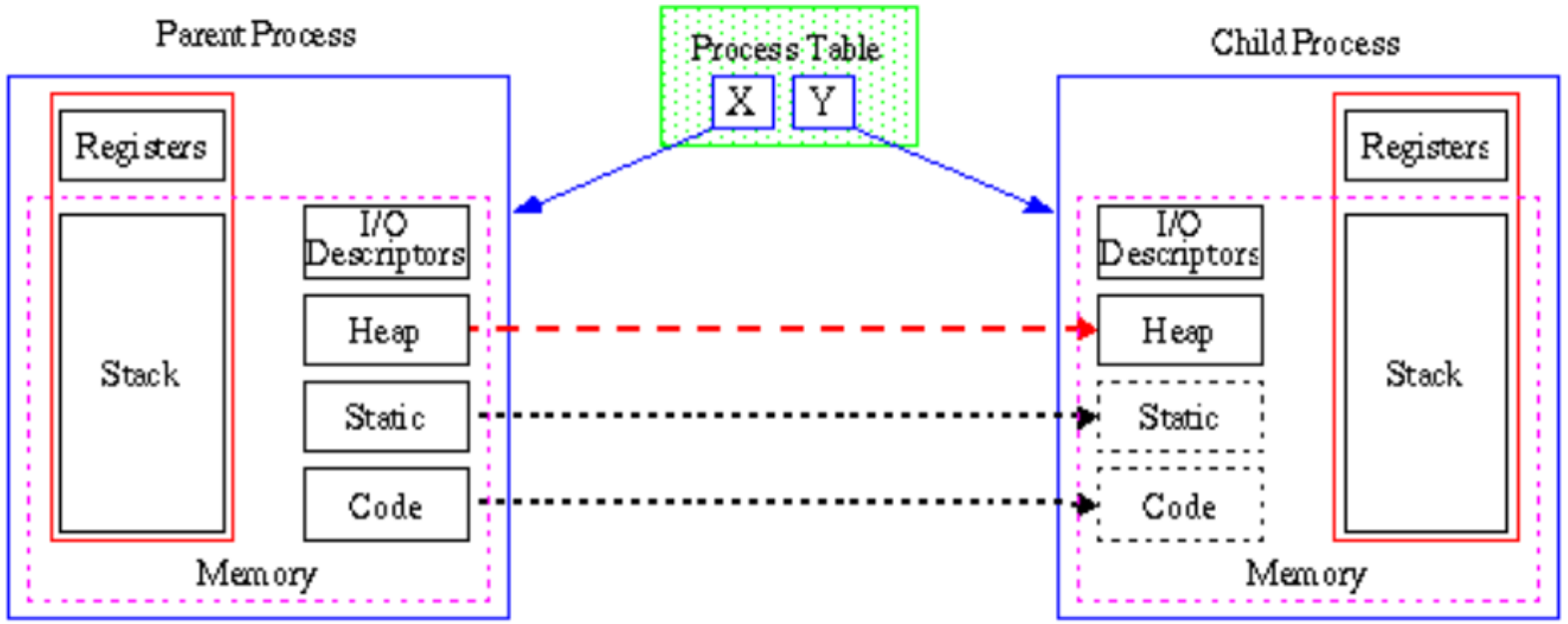
写时复制(Copy-on-Write)

- 子进程共享父进程的地址空间，只要其中任何一个进程要进行写入，则该页面被复制一份，
- 如图，子进程要写C页，则该页被复制一份。



fork()为什么能快速的创建进程？

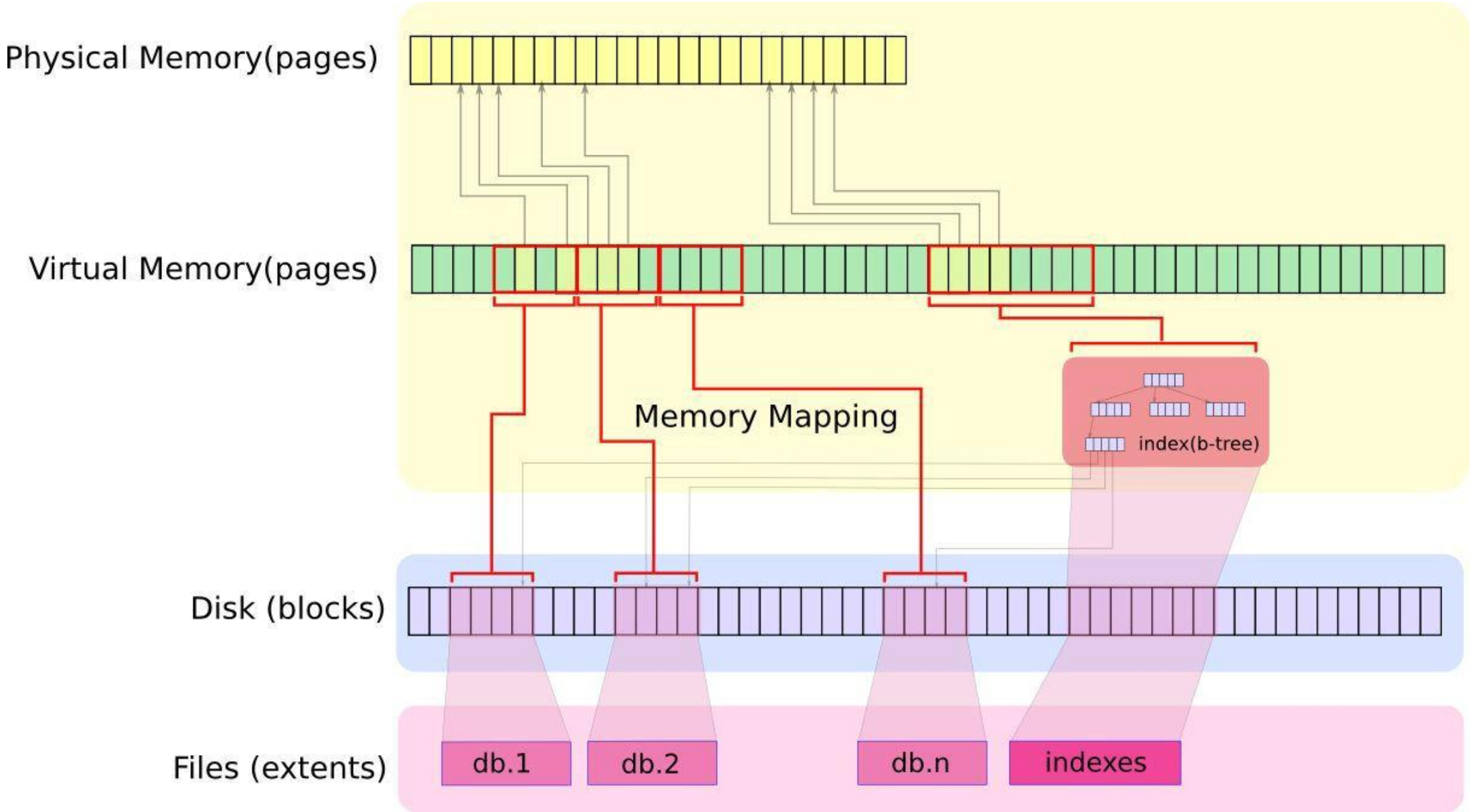
- 可以看出，进程用户空间的创建主要依赖于父进程，
- 而且，在创建的过程中所做的工作，仅仅是mm_struc结构的建立、vm_area_struct结构的建立，以及页目录和页表的建立，并没有真正地复制一个物理页面，
- 这也是为什么Linux内核，能迅速地创建进程的原因之一。



虚存映射

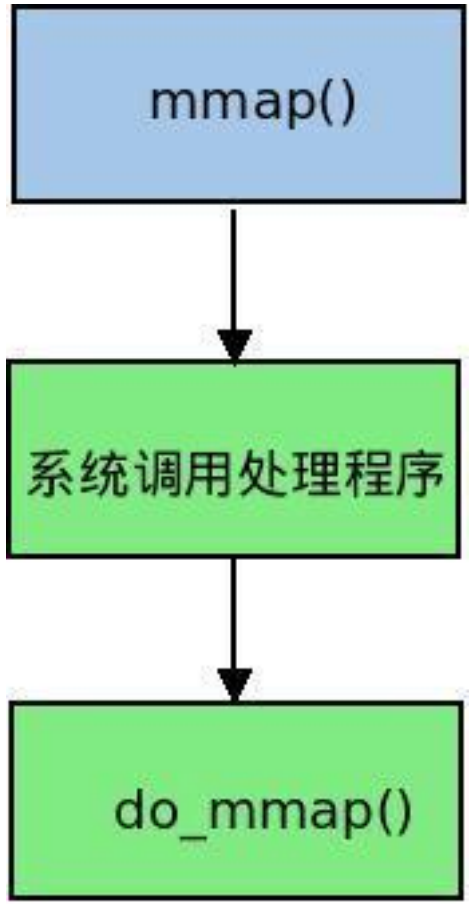
- 当调用exec()系统调用开始执行一个进程时，进程的可执行映像(包括代码段、数据段，堆和栈等)，必须装入到进程的用户地址空间。
- 如果该进程用到了任何一个共享库，则共享库也必须装入到进程的用户空间。
- 由此可看出，Linux并不将映像装入到物理内存，相反，可执行文件只是被映射到进程的用户空间中。
- 这种将可以执行文件映像，映射到进程用户空间的方法，被称为“**虚存映射**”。如后图所示。

虚存映射

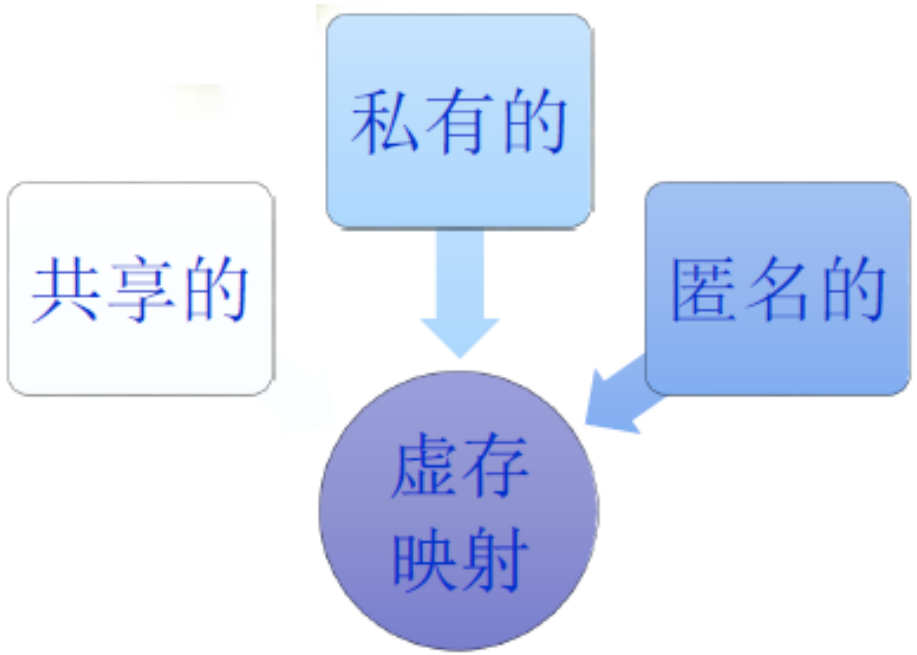

Physical Memory(pages)
Virtual Memory(pages)
Memory Mapping
index(b-tree)
Disk (blocks)
Files (extents)
db.1
db.2
db.n
indexes

VMA的新建方法

- 那么内核通过怎样的方式新建一个个虚存区(VMA)？
- 在用户空间可以通过mmap()系统调用获取do_mmap()(指向do_mmap)的功能
- 在内核空间，可以直接调用do_mmap()创建一个新的虚存区，其调用关系如图：

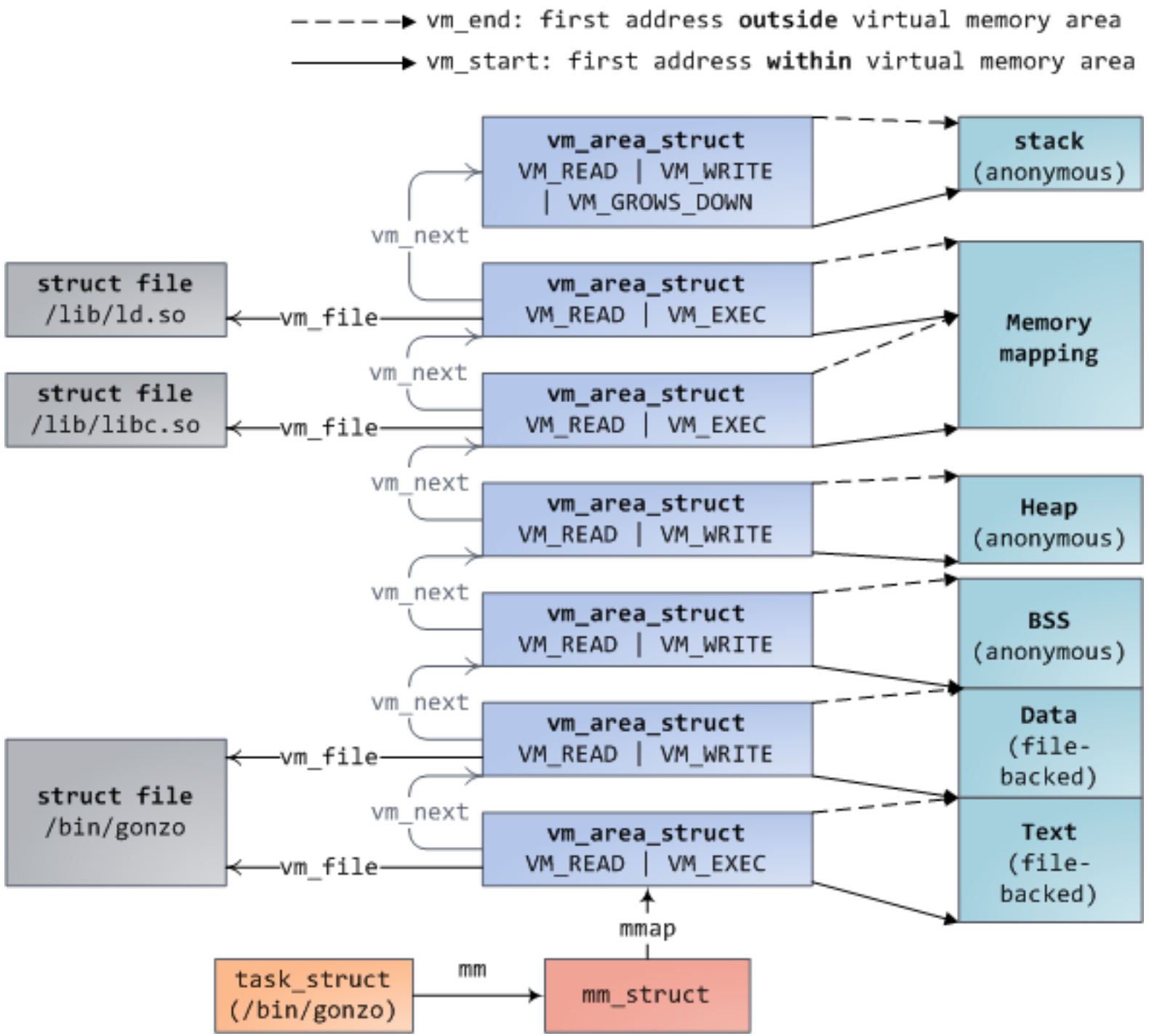


虚存映射包括 共享的、私有的虚存映射和匿名映射



- (1)共享的：有几个进程共享这一映射，也就是说，如果一个进程对共享的虚存区进行写，其它进程都能感觉到，而且会修改磁盘上对应的文件，文件的共享就可以采用这种方式。
- (2)私有的：进程创建的这种映射只是为了读文件，而不是写文件。 因此，对虚存区的写操作不会修改磁盘上的文件，由此可以看出，私有映射的效率要比共享映射的高。
- (3)匿名映射： 除了这两种映射外，如果映射与文件无关，就叫匿名映射。

虚存映射



- 当可执行映像映射到进程的用户空间时，将产生一组vm_area_struct结构来描述各虚拟区间的起始点和终止点。

进程的虚存区举例

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char **argv)
{
  int i;
  unsigned char *buff;
  buff = (char *)malloc(sizeof(char)*1024);
  printf("My pid is :%d\n", getpid());
  for (i = 0; i < 60; i++) {
       sleep(60);
  }
  return 0;
}
```

- 现在，我们通过一个例子来描述，Linux内核是如何把共享库及各个程序段映射到进程的用户空间的。
- 左边是一个C程序exam.c：
- 例子在samplej/41exam中

地址范围	许可权	偏移量	所映射的文件
08048000-08049000	r-xp	00000000	/home/test/exam
08049000-0804a000	rw-p	00001000	/home/test/exam
40000000-40015000	r-xp	00000000	/lib/ld-2.3.2.so
40015000-40016000	rw-p	00015000	/lib/ld-2.3.2.so
40016000-40017000	rw-p	00000000	匿名
4002a000-40159000	r-xp	00000000	/lib/libc-2.3.2.so
40159000-4015e000	rw-p	0012f000	/lib/libc-2.3.2.so
4015e000-40160000	rw-p	00000000	匿名
bfffe000-c0000000	rwxp	fffff000	匿名

进程的虚存区举例

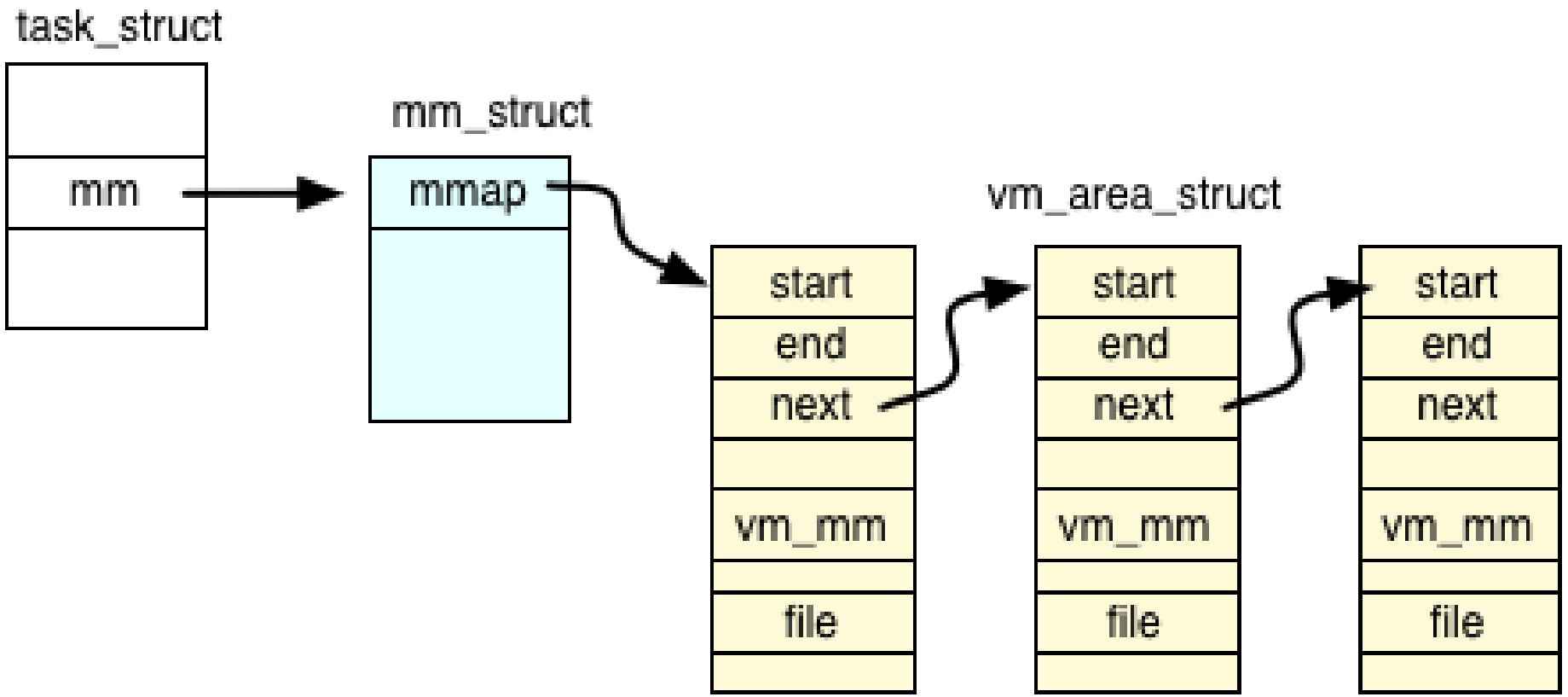
- 假设程序执行后其对应的PID为513，这个进程对应的虚存区如表所示,可以从/proc/513/maps得到这些信息
- 注意，这里列出的所有区都是私有虚存映射(在许可权列出现的字母p)。
- 这是因为，这些虚存区的存在，仅仅是为了给进程提供数据；
- 当进程执行指令时，可以修改这些虚存区的内容，但与它们相关的磁盘上的文件保持不变，这就是私有虚存映射所起的保护作用。

与用户空间相关的主要系统调用

系统调用	描述
fork()	创建具有新的用户空间的进程，用户空间中的所有页被标记为“写时复制”，且由父子进程共享。
mmap()	在进程的用户空间内创建一个新的虚存区。
munmap()	销毁一个完整的虚存区或其中的一部分，如果要取消的虚存区位于某个虚存区的中间，则这个虚存区被划分为两个虚存区。
exec()	装入新的可执行文件以代替当前用户空间

编写虚存区内核模块

- 我们可以编写内核模块，查看某一进程的虚拟区。



- 在进程的task_struct(PCB)结构中，有如下定义：
  - struct task_struct{
  - ……
  - struct mm_struct *mm; /*描述进程的整个用户空间*/
  - ……
  - }
    - 代码在include/linux/sched.h
- 在struct mm_struct结构中，又有如下定义：
  - struct mm_struct{
  - ……struct vm_area_struct * mmap; /*描述进程的虚存区*/
  - ……
  - }
    - 代码在include/linux/mm_types.h

编写虚存区内核模块

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/sched.h>
static int pid;
module_param(pid,int,0644);
static int __init memtest_init(void)
{
   struct task_struct *p;
   struct vm_area_struct *temp;
   printk("The virtual memory areas(VMA) are:\n");
   p = pid_task(find_vpid(pid), PIDTYPE_PID); /*
            该函数因内核版本而稍有不同*/
   temp = p->mm->mmap;
```

- 编写一个内核模块，打印进程的虚存区，其中通过模块参数把进程的pid传递给模块。
- 该模块中，通过传递的pid，可以获得对应的PCB，从而可以打印出其各个虚存区的起始地址。
- 代码在samplej/42memtest

编写虚存区内核模块

```
 while(temp) {
   printk("start:%p\tend:%p\n", (unsigned long *)temp->vm_start,
                                 (unsigned long *)temp->vm_end);
   temp = temp->vm_next;
 }
 return 0;
}
static void __exit memtest_exit(void)
{
   printk("Unloading my module.\n");
   return;
}
module_init(memtest_init);
module_exit(memtest_exit);
MODULE_LICENSE("GPL");
```

- 我们先运行前面写的程序exam,
- $ ./exam
- 然后把这个进程的pid(假设pid是9413)作为参数带入模块，如下：
  - $ sudo insmod memtest.ko pid=9413
  - $dmesg

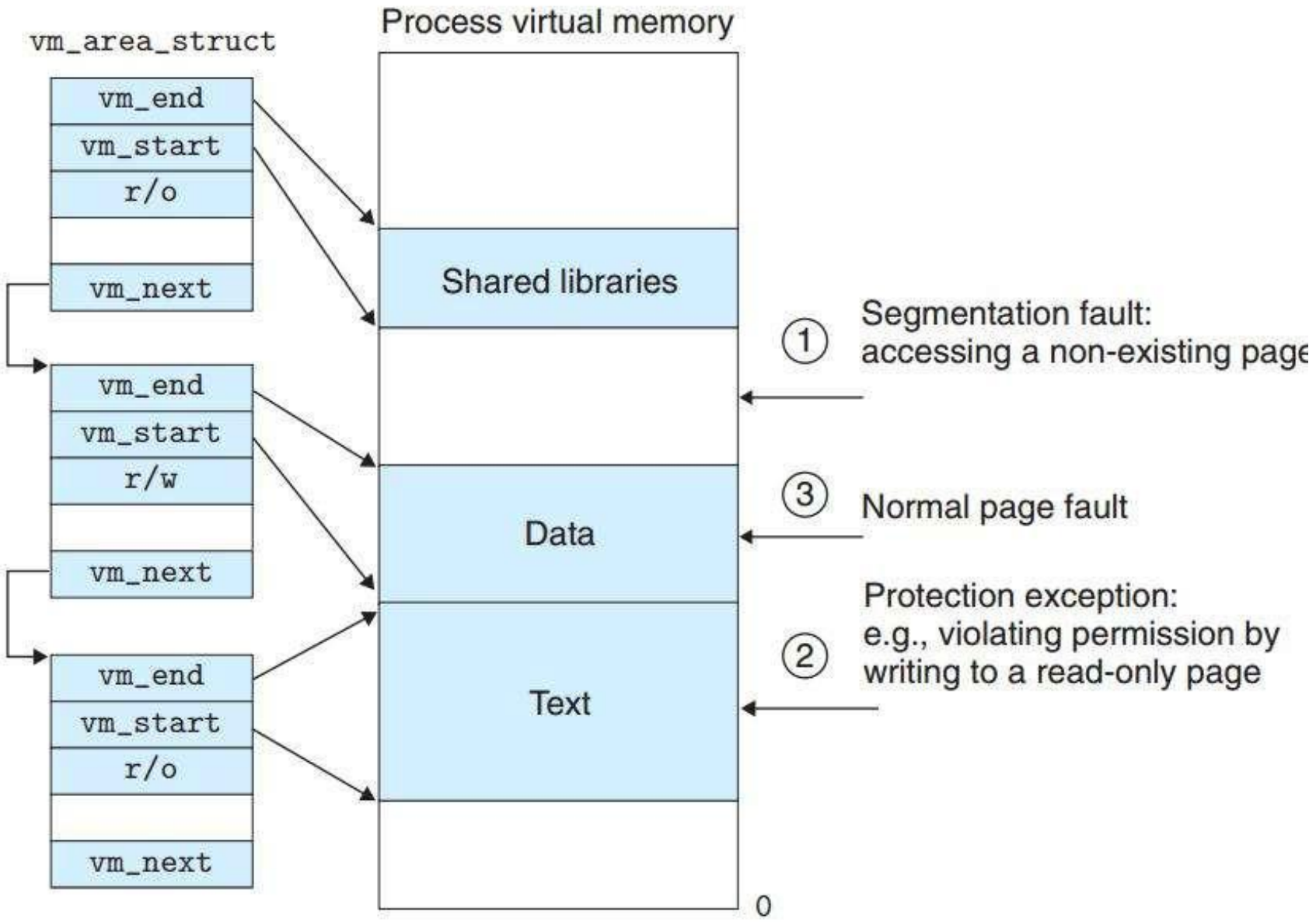
内核模块输出信息

地址范围	许可权	偏移量	所映射的文件
08048000-08049000	r-xp	00000000	/home/test/exam
08049000-0804a000	rw-p	00001000	/home/test/exam
40000000-40015000	r-xp	00000000	/lib/ld-2.3.2.so
40015000-40016000	rw-p	00015000	/lib/ld-2.3.2.so
40016000-40017000	rw-p	00000000	匿名
4002a000-40159000	r-xp	00000000	/lib/libc-2.3.2.so
40159000-4015e000	rw-p	0012f000	/lib/libc-2.3.2.so
4015e000-40160000	rw-p	00000000	匿名
bfffe000-c0000000	rwxp	fffff000	匿名

- 可以看出，输出的信息与前面从proc文件系统中所读取的信息是一致的。
- 进程的用户空间是由一个个的虚存区组成。
- 对进程用户空间的管理在很大程度上依赖于对虚存区的管理。

请页机制 - 实现虚存管理的重要手段

- 进程运行时，CPU访问的是用户空间的虚地址。
- Linux仅把当前要使用的少量页面装入内存，需要时再通过请页机制，将特定的页面调入内存。
- 当要访问的页不在内存时，产生一个页故障，并报告故障原因。
- 如图所示。

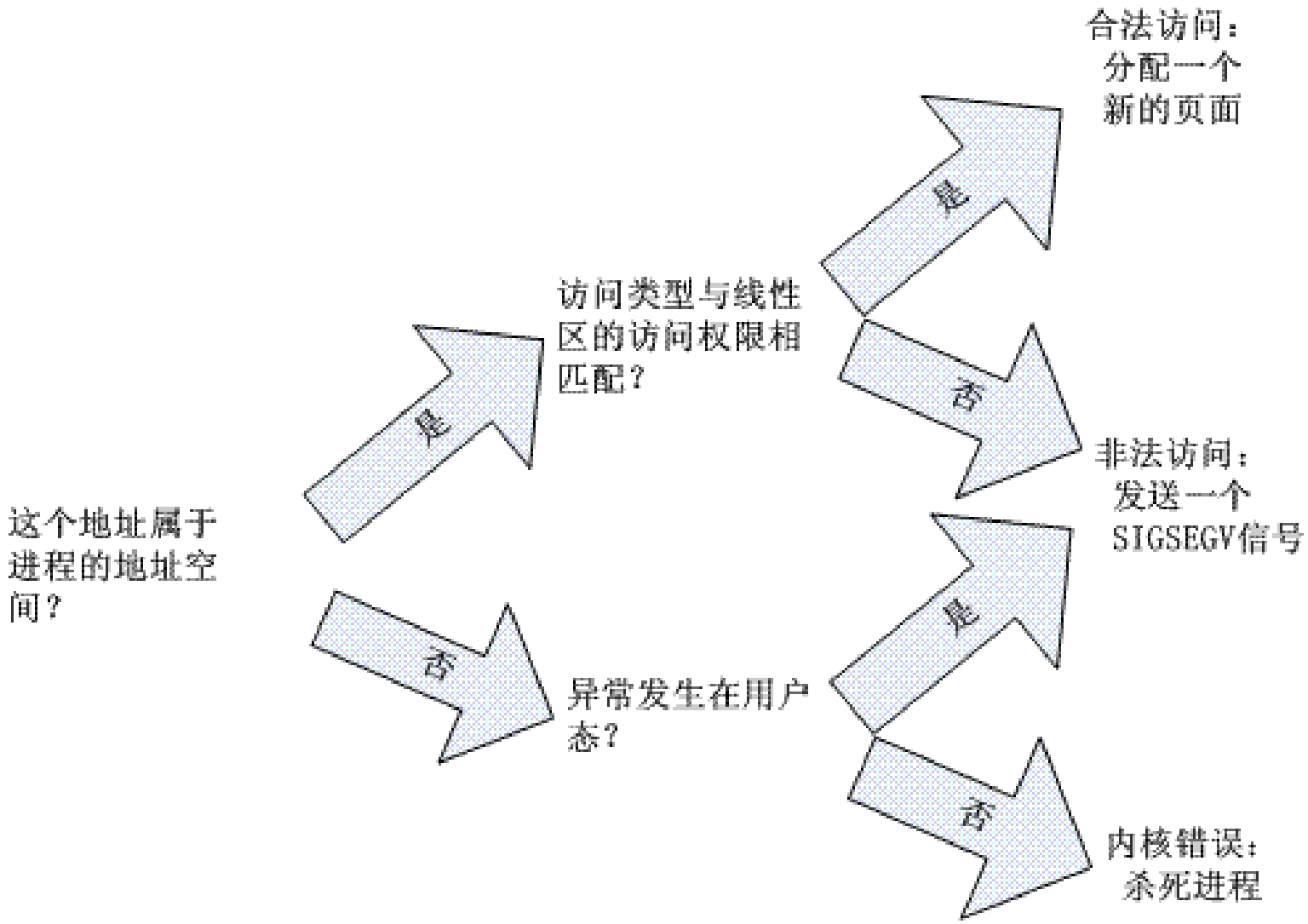


缺页异常处理

- Linux的缺页(Page Fault)异常处理程序区分以下两种情况：
- 　由编程错误所引起的异常，或是 缺页引起的异常；
- 如果是编程引起的异常，还发在内核态，毫不含糊的杀死该进程，如果发生在用户态，说明是无效的内存引用，程序也停止执行；
- 如果是缺页引起的异常，而且有合法的权限，则进入缺页异常处理程序。
- 如后图所示。

缺页异常处理

- 从图看出，很多分支都是处理缺页发生时异常情况，只有两个分支是真正的请求分页处理，
- 大家可以先查看这两个正常情况处理的分支，
- 然后再看异常情况的处理。



内容导航：

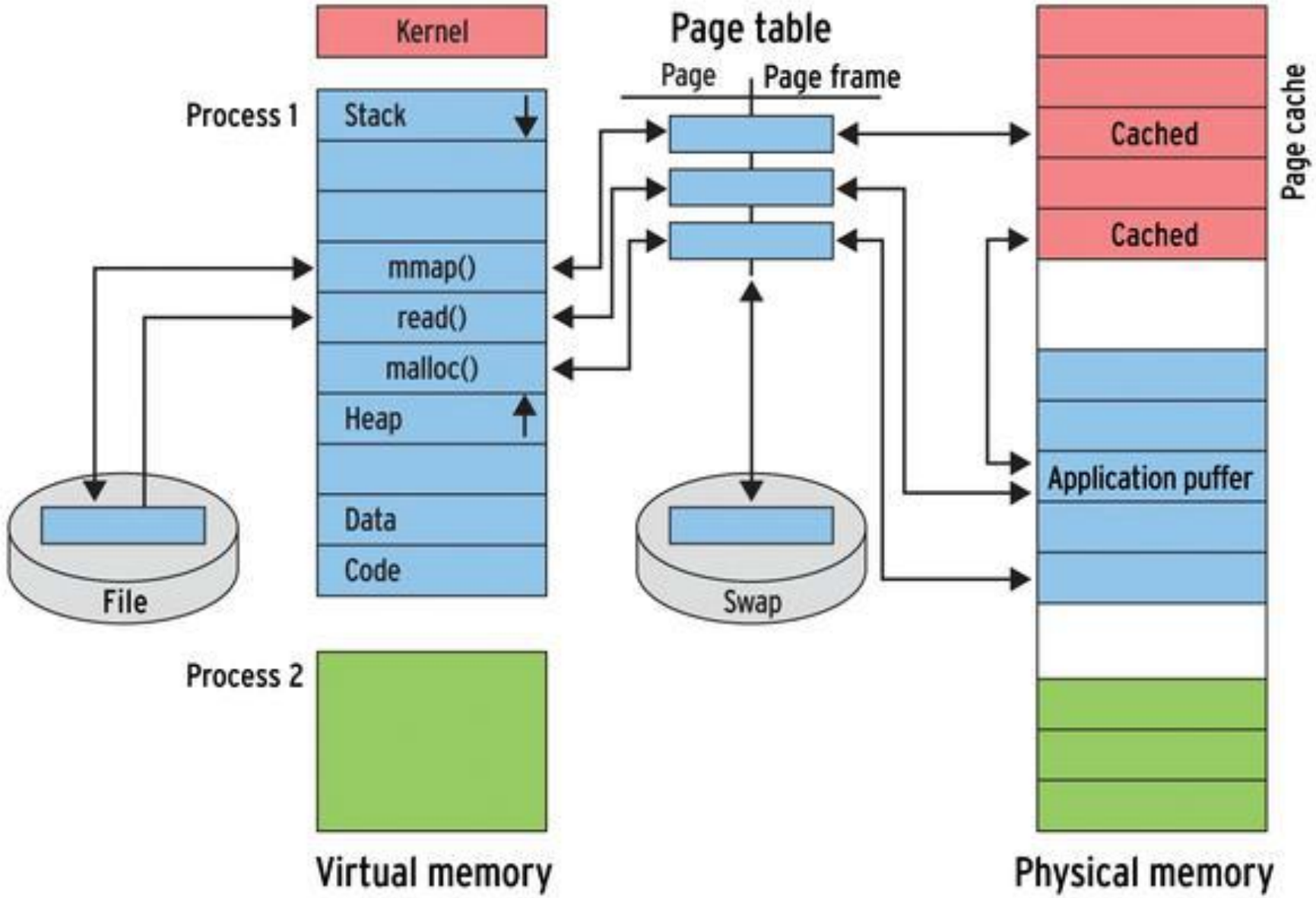
3 内存分配与回收机制(一)

进程在执行时的步骤

- 从操作系统的角度看，一个进程最关键的特征，是拥有独立的虚拟地址空间。
- 创建并执行一个进程通常需要执行步骤如下：
- 1、建立可执行文件与虚拟地址空间的映射
- 2、将指令寄存器设置为可执行文件入口，并启动运行。
- 下面将对这两步进行介绍。

进程在执行时的步骤

- 1、建立可执行文件与虚拟地址空间的映射
- 当执行一个程序时，加载器读取可执行文件(ELF)的头，建立虚拟空间和可执行文件的映射，调用的是do_mmap()函数，同时虚拟地址空间所需的数据结构mm_struct和vm_area_struct也填充相应的值， 如图左边虚拟内存部分
- 2、将指令寄存器设置为可执行文件入口，并启动运行。



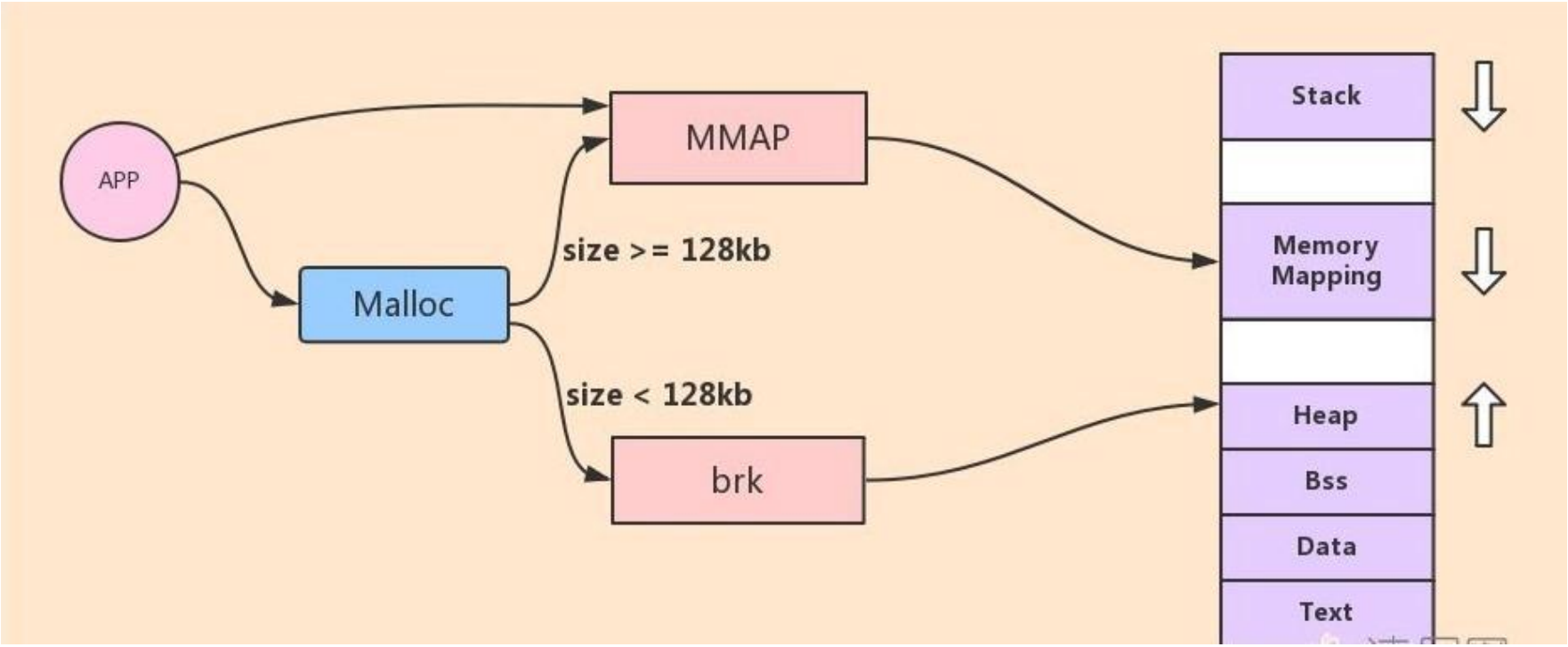
程序的装载执行过程

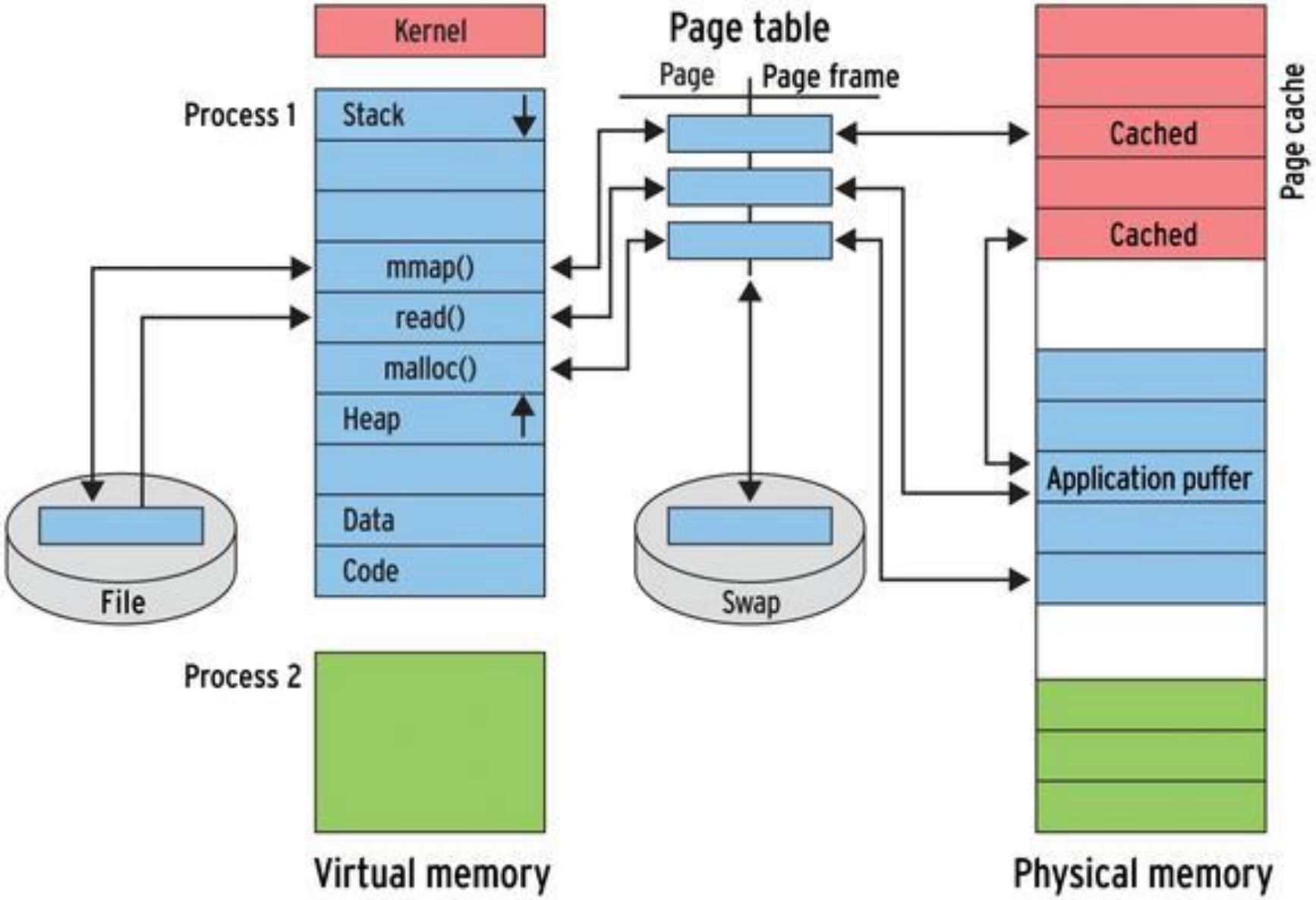
- 一个程序在装载，在做什么？
- 在上述步骤后，执行文件的指令和数据加载进内存，但并没有真正的装入物理内存，只是通过ELF文件头部信息，建立起可执行文件与虚拟地址空间的映射关系而已，
- 真正加载过程将在发生缺页异常处理时才进行，装载执行过程如下：
- 1、内核根据上面建立的映射关系，找到所需的内容在可执行文件中的位置。
- 2、分配一个物理内存页面，并将可执行文件内容装载到该内存页中。
- 3、建立该物理页面和虚拟地址空间的映射关系，也就是说填充页表，然后把控制权交换给进程。

- 前面已介绍过缺页异常处理机制，接下来介绍物理内存的分配与回收。

当调用malloc()时，内核做什么

- 当我们在用户程序申请内存时调用malloc()，内核实际上负责为进程动态的申请一块内存，操作系统从堆中分配一块内存，并把首地址返回给用户。
- malloc()申请内存大小不一样，最终调用的系统调用也不一样，如图所示，
- 内核不会立即为进程分配物理内存，依然是通过 请页机制。如图所示。

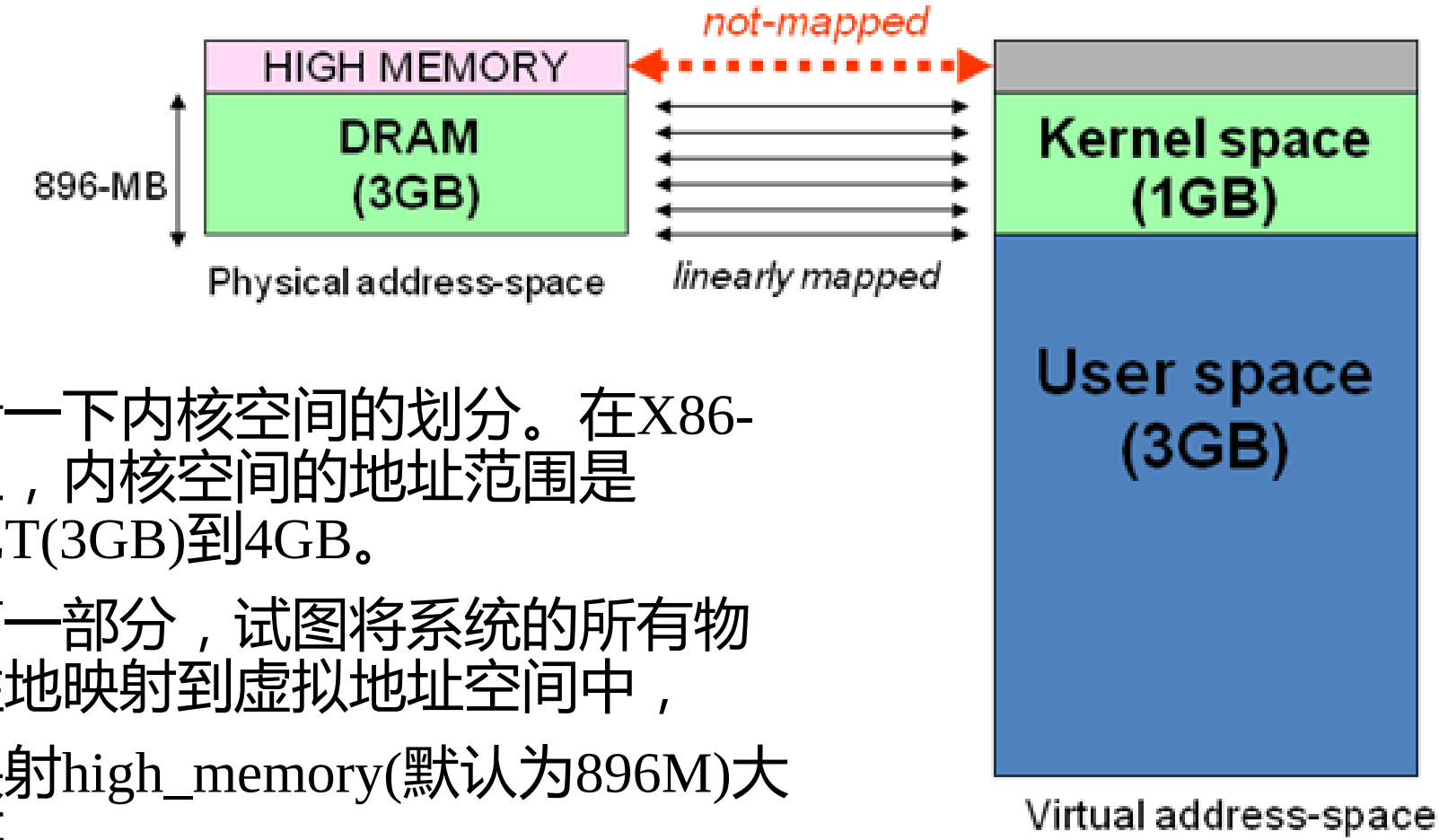




物理内存管理

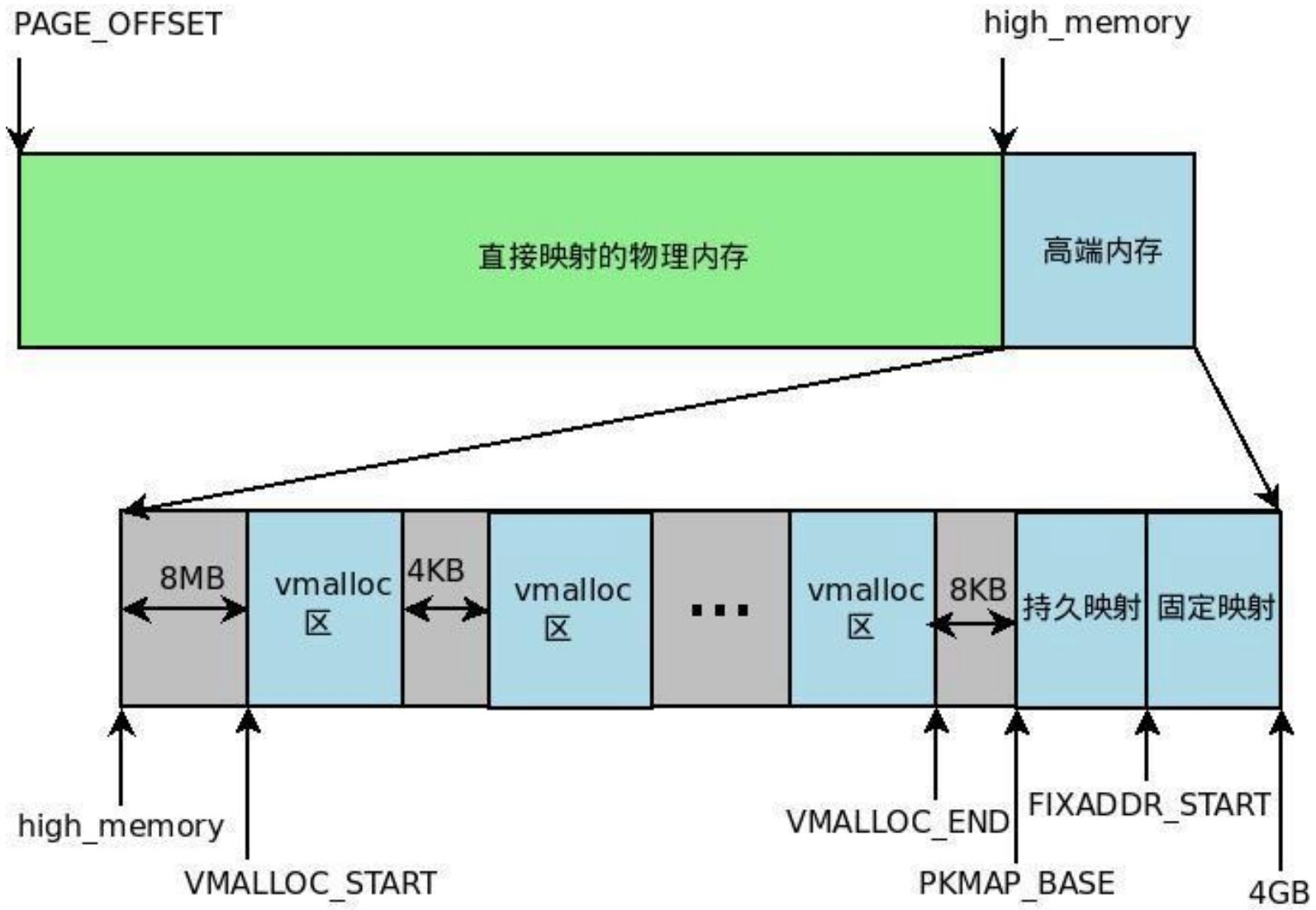
- 请页机制可以为进程请求物理内存，
- 那么物理内存在内核中，究竟是如何管理和分配的？
- 接下来将从物理内存的逻辑模型开始说起。

内核空间的划分



- 首先，我们看一下内核空间的划分。在X86-32体系架构上，内核空间的地址范围是PAGE_OFFSET(3GB)到4GB。
- 内核空间的第一部分，试图将系统的所有物理内存，线性地映射到虚拟地址空间中，
- 但最多只能映射high_memory(默认为896M)大小的物理内存。
- 大于high_memory的物理内存，将映射到内核空间的后部分。

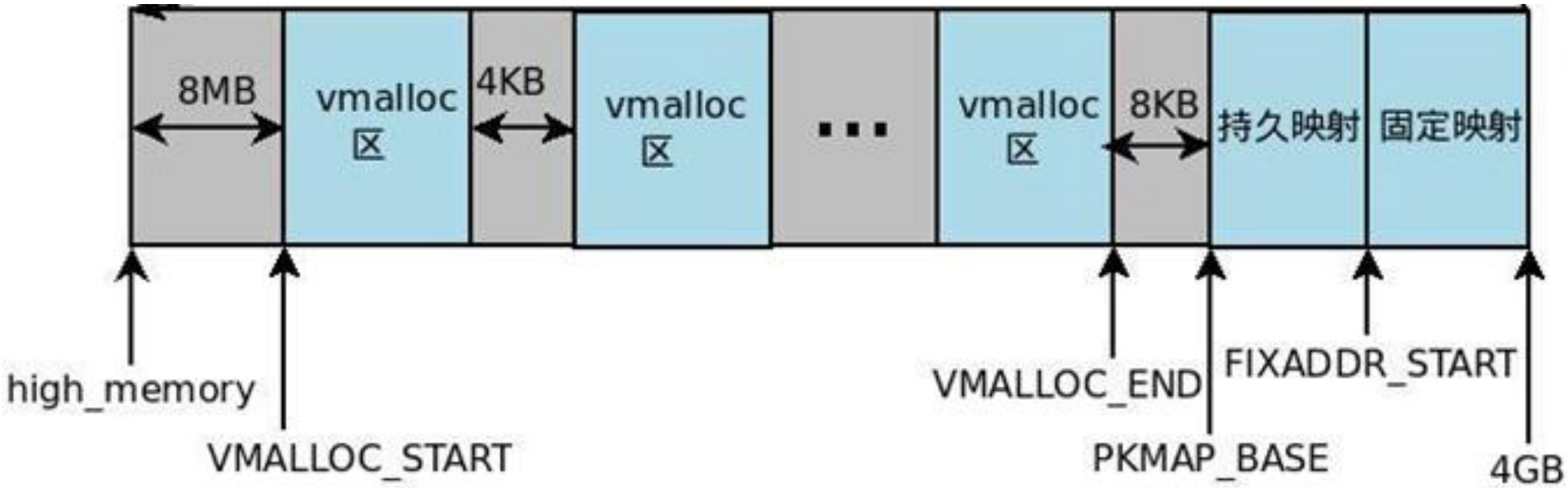
内核空间的划分图示



- 按照这样的映射规则，
- 0M到high_memory的物理内存称为低端内存，
- 大于high_memory的物理内存称为高端内存。

内核空间的划分图示

- 从图中可以看出，内核采用了三种机制将高端内存映射到内核空间：
- 1.永久内核映射
- 2.固定映射
- 3.vmalloc机制。



内核虚拟地址和物理地址的转换

- 内核虚拟地址和物理地址如何进行转换？内核为线性映射的内存区提供物理地址和虚拟地址的转换函数：
  - __pa(vaddr)：返回虚拟地址vaddr对应的物理地址。
  - __va(paddr)：返回物理地址paddr对应的虚拟地址。

- 内核地址空间是从PAGE_OFFSET开始的，因此上述两个地址转换函数的源码，在2.6版本中的实现如下：

```
#define __pa(x)          ((unsigned long) (x) - PAGE_OFFSET)
#define __va(x)          ((void *)((unsigned long) (x) + PAGE_OFFSET))
```

- 从这两个宏定义可以看出，这完全是一种线程关系。

物理内存管理机制

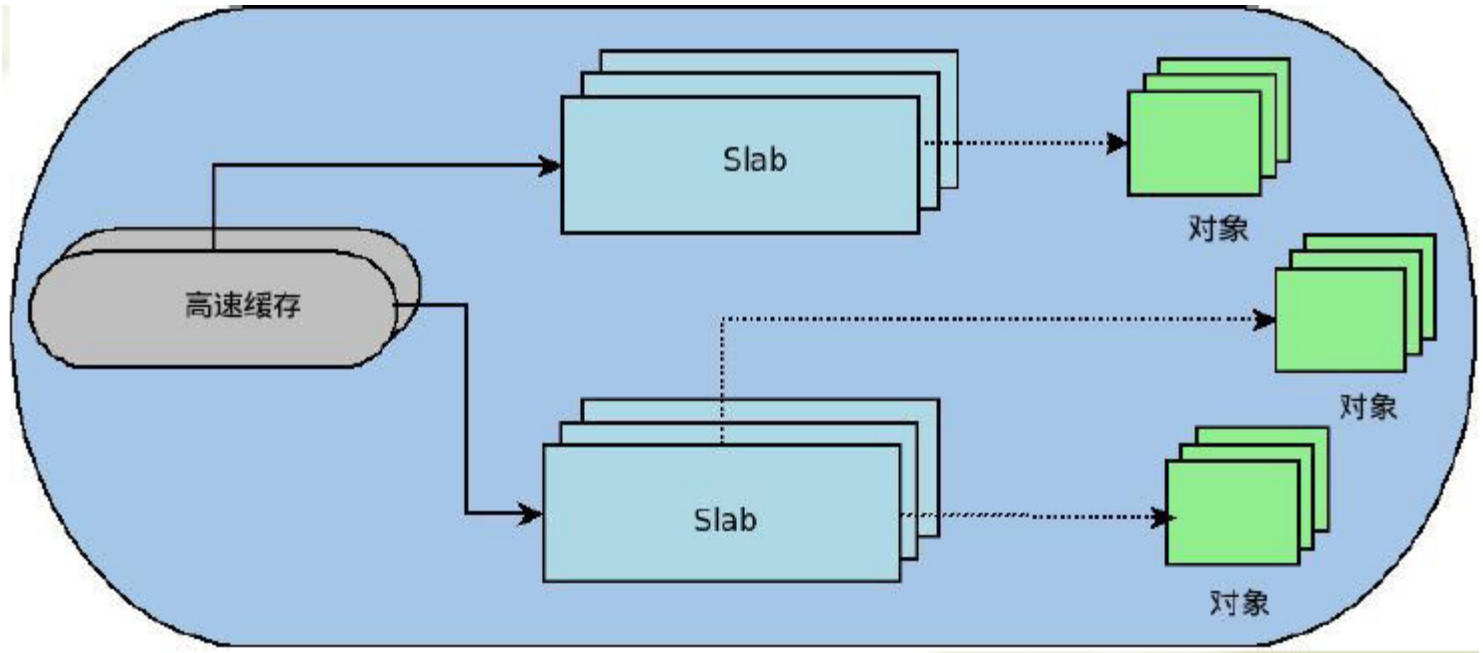
- 基于物理内存在内核空间中的映射原理，物理内存的管理方式也有所不同。
- 内核中物理内存的管理机制主要有以下四种：
- **伙伴算法**：负责大块连续物理内存的分配和释放，以**页框**为基本单位。该机制可以避免外部碎片。
- **per-CPU页框高速缓存**：内核经常请求和释放单个 **页框**，该缓存包含预先分配的页框，用于满足本地CPU发出的单一页框请求。
- **slab缓存**：负责小块物理内存的分配，并且它也作为高速缓存，主要针对内核中经常分配并释放的对象。
- **vmalloc机制**：vmalloc机制使得内核通过连续的线性地址，来访问非连续的物理**页框**，这样可以最大限度的使用高端物理内存。

Slab 分配机制 - 分配小内存

- 伙伴算法负责大块连续物理内存的分配和释放，以 **页框** 为基本单位，将在下一讲介绍，
- 那么分配小块内存怎么办呢？slab分配器就是为此而提出的。
- slab 分配器最初是为了解决物理内存的内部碎片而提出的，它将内核中常用的数据结构看做对象。
- slab分配器为每一种对象建立高速缓存。

Slab 分配机制 - 分配小内存

- 内核对该对象的分配和释放均是在这块高速缓存中操作，如图所示。
- 可以看到每种对象的高速缓存，是由若干个slab组成，每个slab是由若干个 **页框** 组成的。
- 虽然slab分配器可以分配比单个页框更小的内存块，但它所需的所有内存都是通过伙伴算法分配的。



Slab 分配机制 - 分配小内存

Slab通用缓存

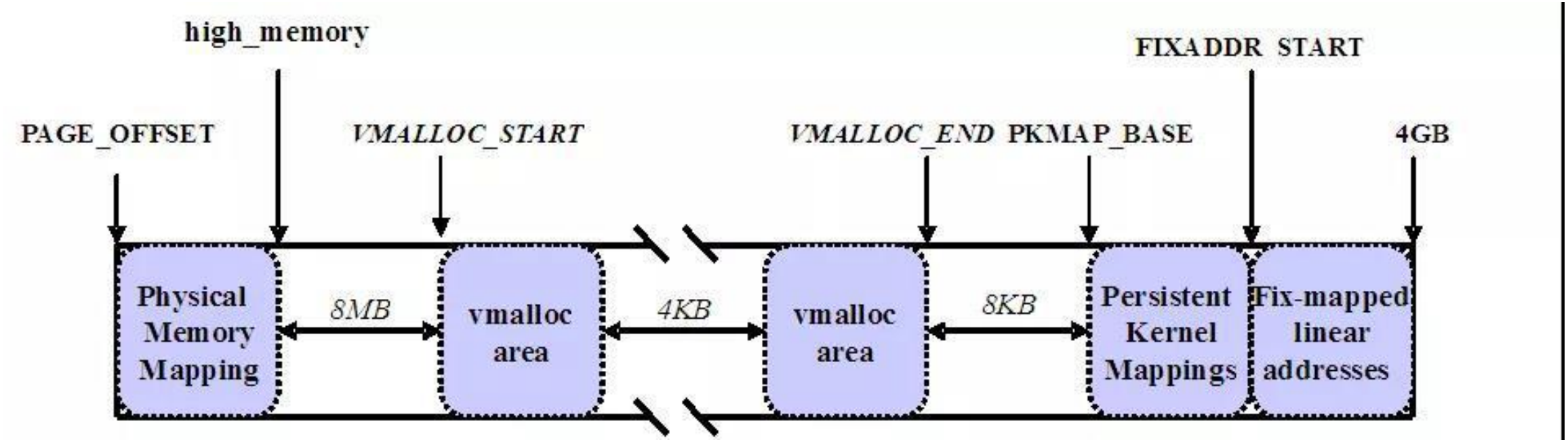
- 通用缓存是针对一般情况，适合分配任意大小的物理内存
- 其接口为kmalloc()

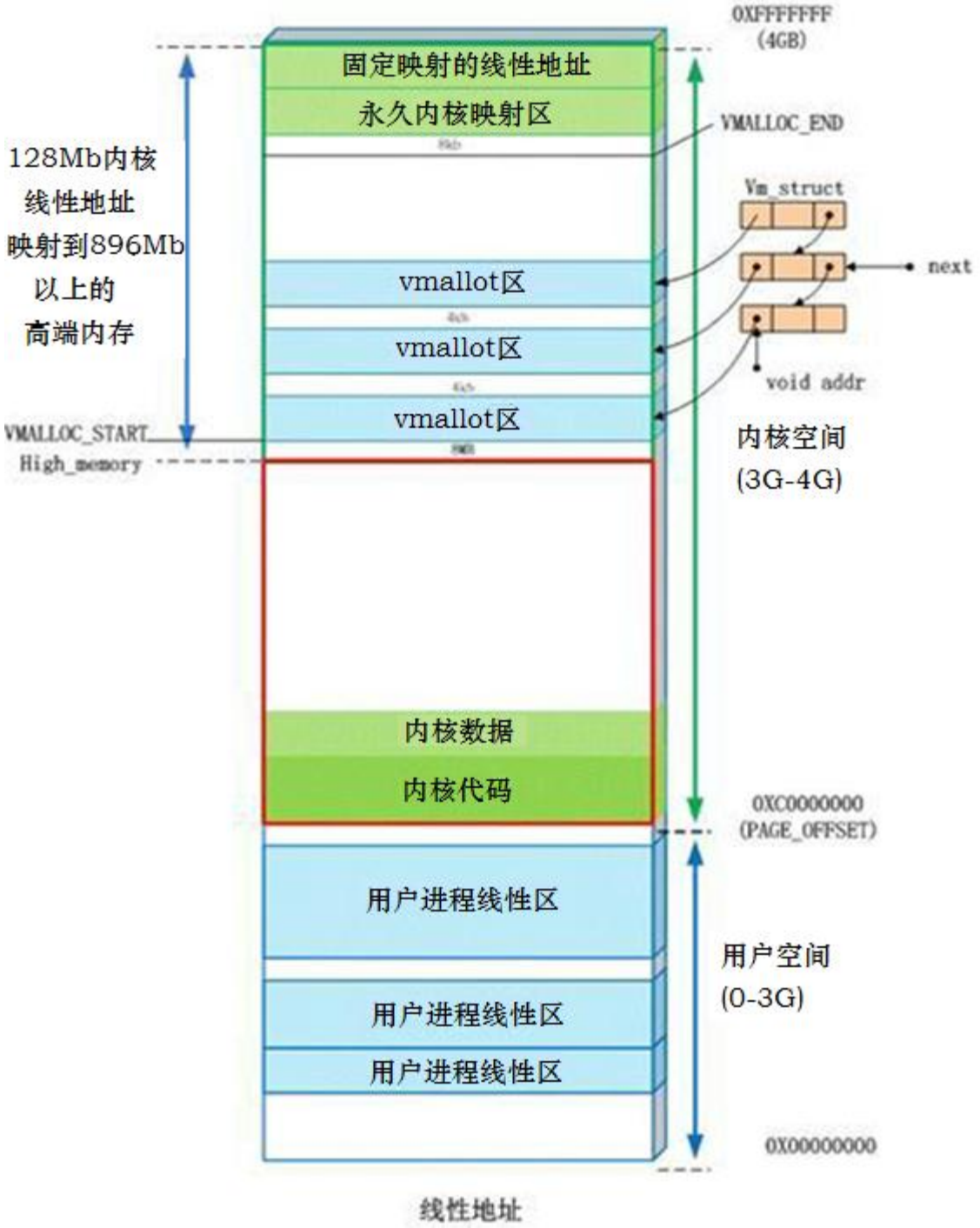
Slab专用缓存

- 专用缓存是对特定的对象。
- 比如为进程控制块创建高速缓存task_struct_cachep

内核空间非连续内存区的分配

- 非连续内存，处于3G到4G之间的内核空间的高端内存区。
- 我们知道物理上连续的映射对内核是最好的，但不是总能成功。
- 在分配一大块内存时，可能无法找到连续的内存块。内核使用vmalloc()接口函数，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。
- 使用vmalloc()最好的实例是为内核模块分配内存，因为模块可能在任何时候加载，如果模块数据较多，那么无法保证有足够的连续内存可用。如图所示。





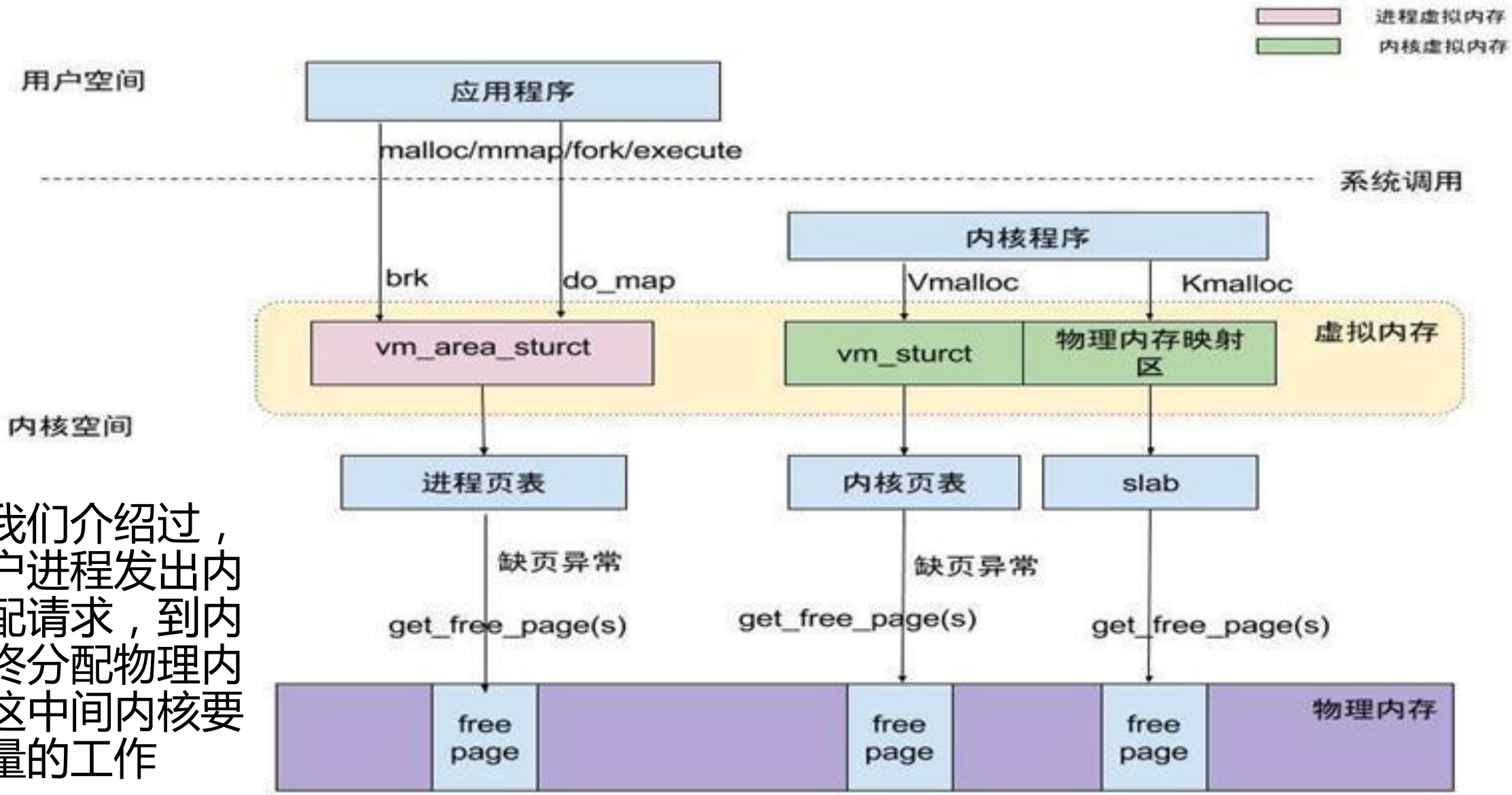
vmalloc()与 kmalloc()之区别

- vmalloc()与kmalloc()都可用于内核空间分配内存。
- kmalloc()分配的内存处于3GB ~ high_memory之间，这段内核空间与物理内存的映射一一对应，
- 而vmalloc()分配的内存在VMALLOC_START ~ VMALLOC_END之间，这段非连续内存区映射到物理内存也可能是非连续的
- vmalloc() 分配的物理地址无需连续，而kmalloc() 确保页在物理上是连续的。

内容导航：

4 内存分配与回收机制(二)

内存分配

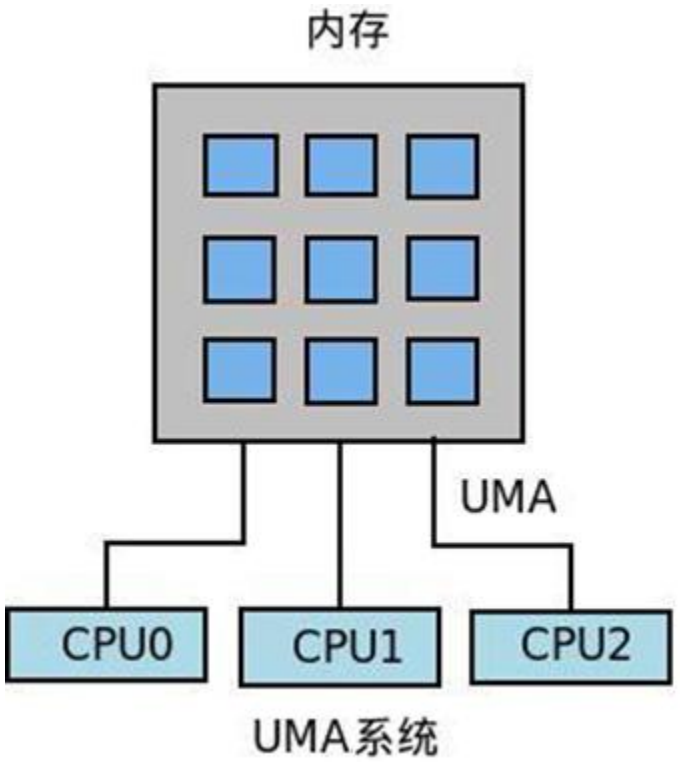


- 前面我们介绍过，从用户进程发出内存分配请求，到内核最终分配物理内存，这中间内核要做大量的工作

UMA和NUMA计算机的内存管理方式

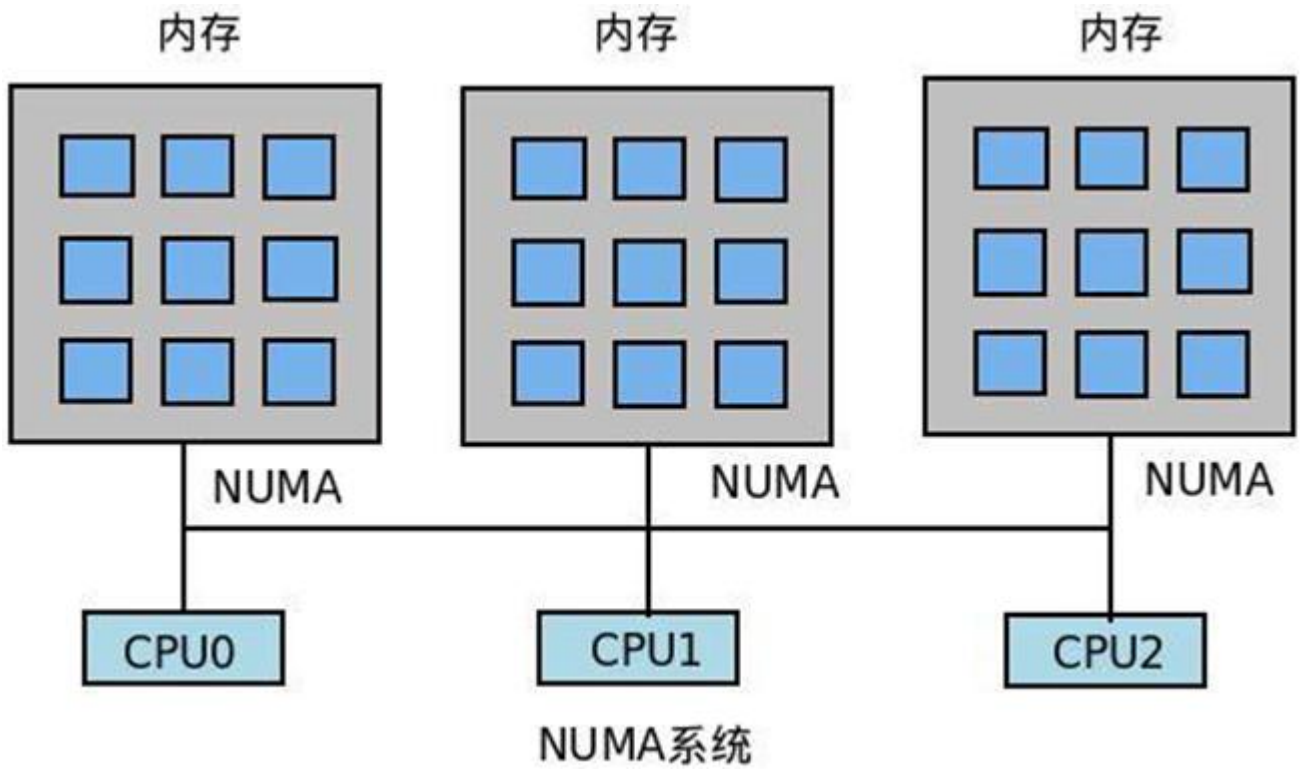
- 目前有**NUMA和UMA**两种类型的计算机，分别以不同的方法管理物理内存:

- **UMA计算机(uniform memory access)**：将可用内存以连续方式组织起来，如图所示。



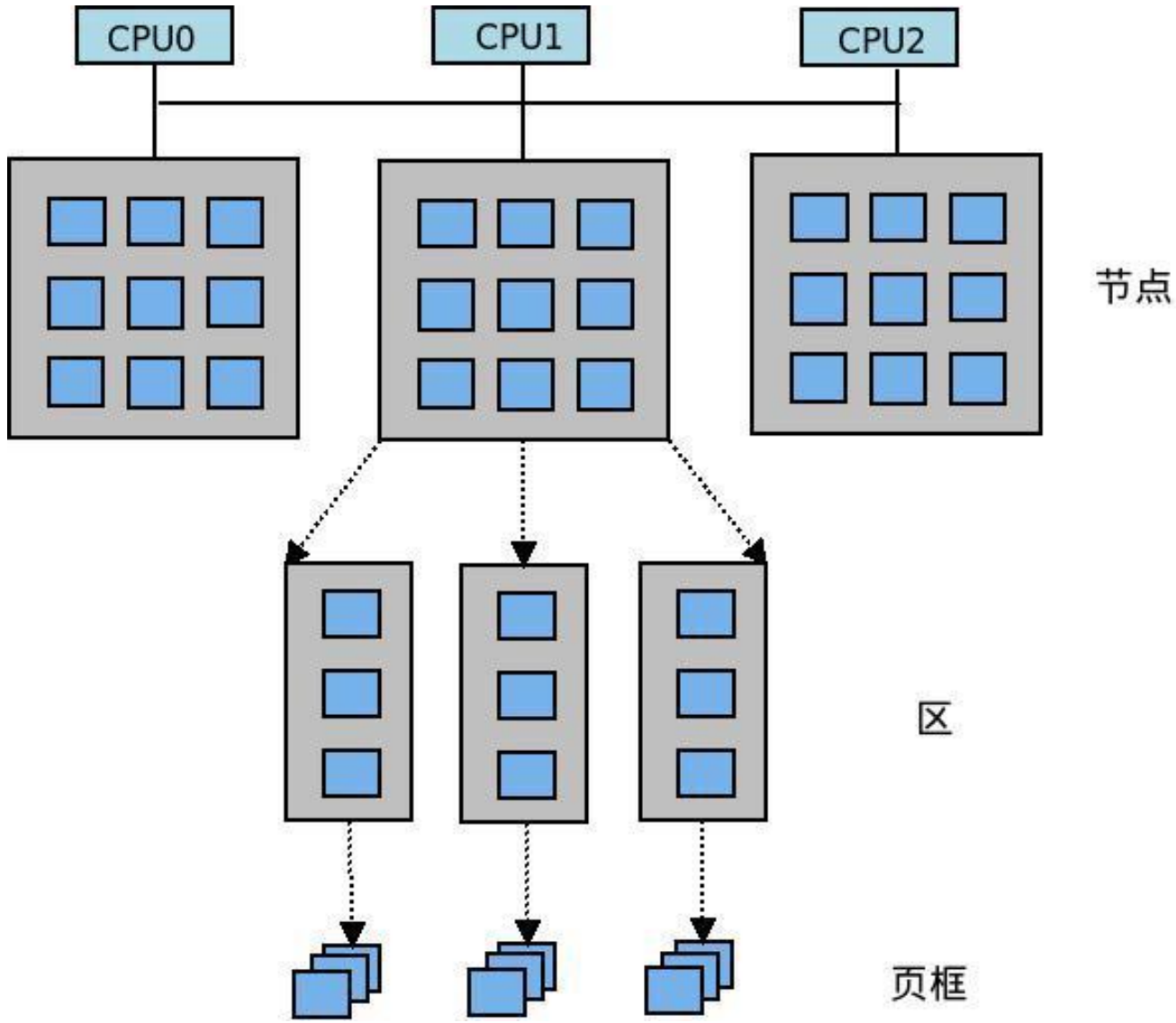
UMA和NUMA计算机的内存管理方式

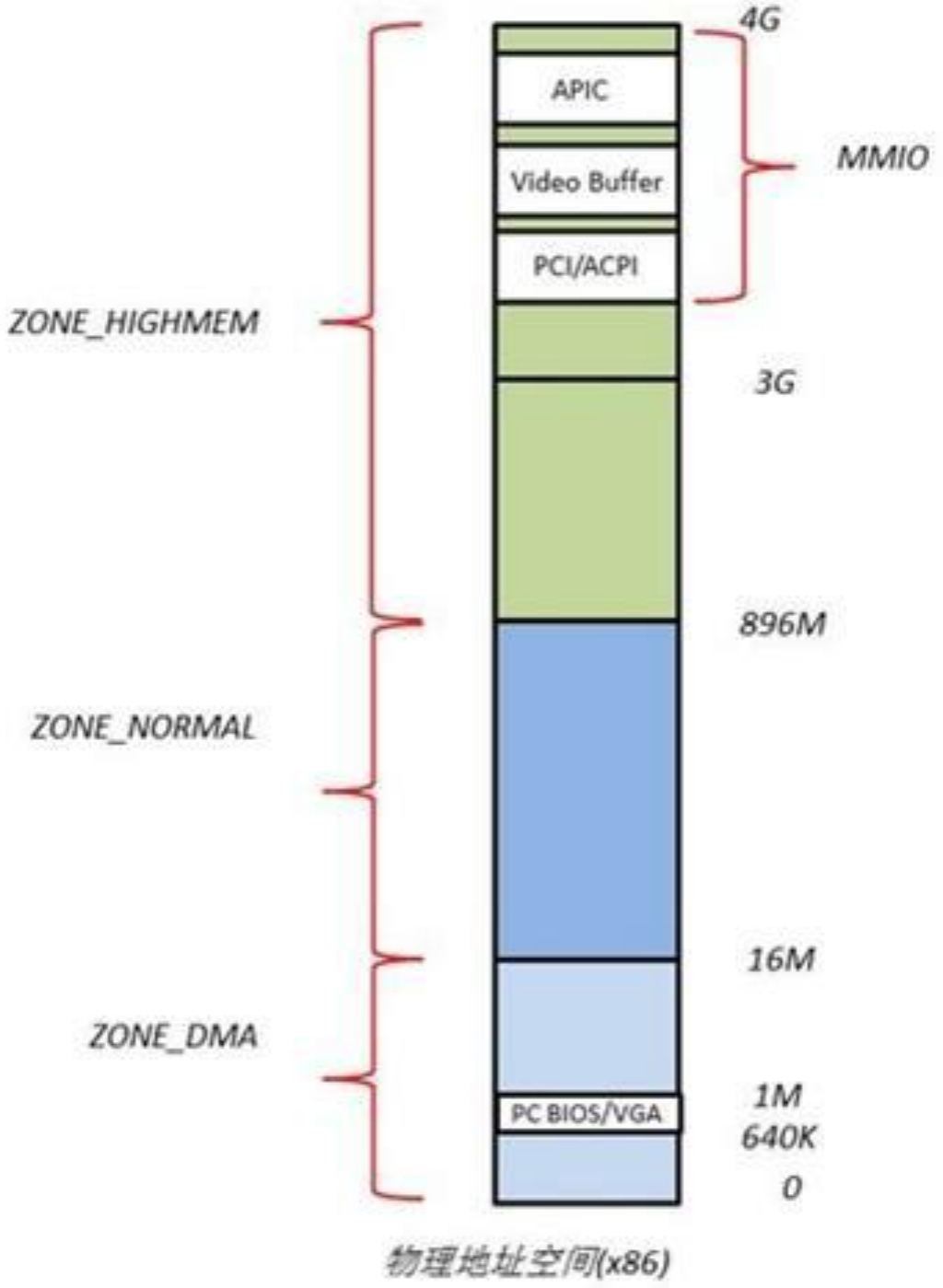
- **NUMA计算机(non-uniform memory access)**：是一种多处理器计算机，每个CPU拥有各自的本地内存。
- 这样的划分使每个CPU都能以较快的速度访问本地内存，各个CPU之间通过总线连接起来，
- 这样也可以访问其他CPU的本地内存，只不过速度比较慢而已。
- 如图所示。



物理内存的组织 及 内存节点

- 为兼容NUMA模型，内核引入了**内存节点Node**，每个节点关联一个CPU。
- 各个节点又被划分几个**内存区Zone**，每个内存区中又包含若干个**页框Page**。
- 物理内存在逻辑上被划分为三级结构，用pg_data_t，zone和page这三种数据结构，依次描述**节点Node**，**区Zone**和**页框Page**。
- NUMA计算机中每个CPU的物理内存称为一个**内存节点**，内核通过pg_data_t数据结构，来描述一个内存节点，系统内的所有结点形成一个双链表。
- UMA模型下的物理内存只对应一个节点，也就是整个物理内存形成一个节点，因此上述的节点链表中，只有一个元素。



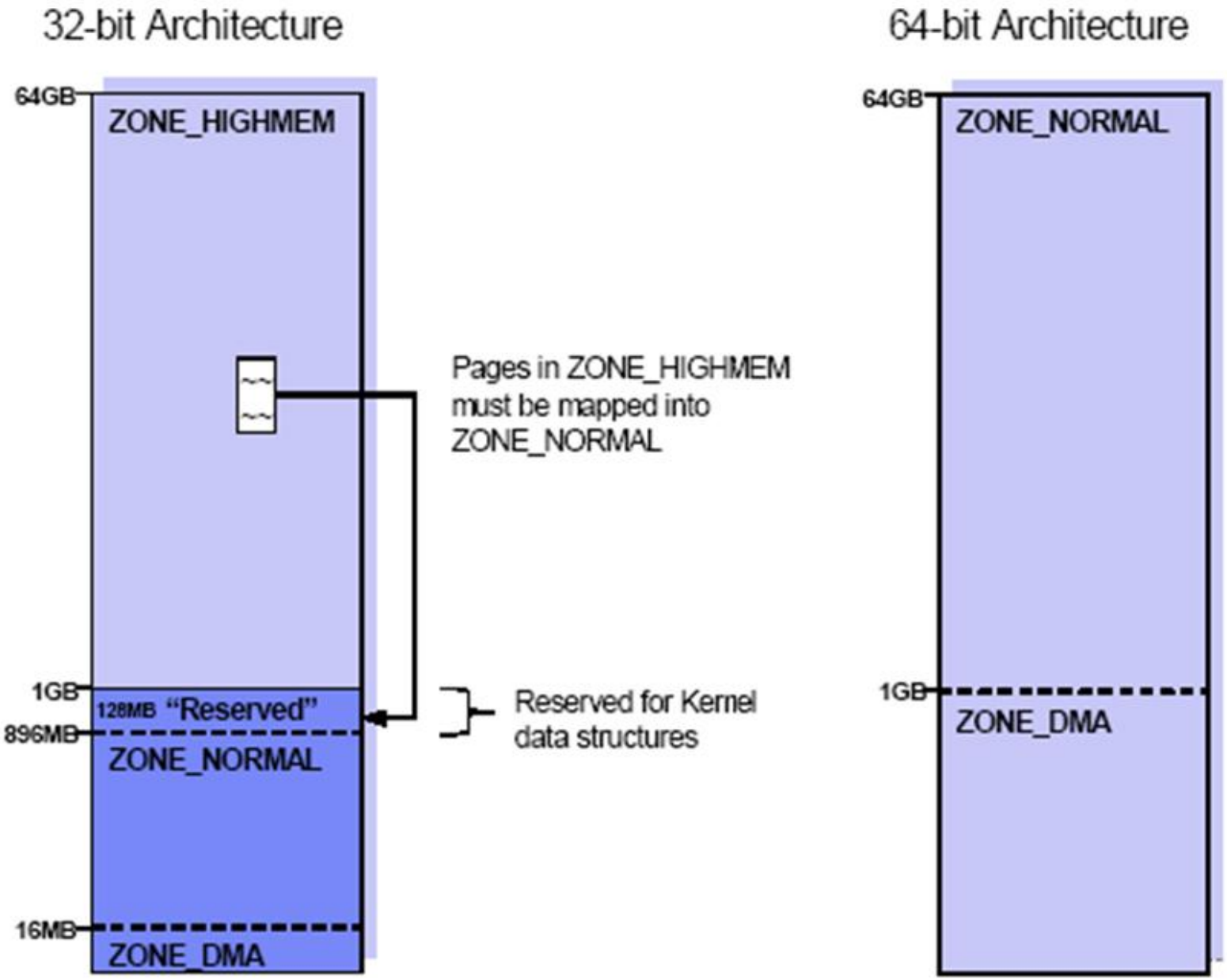


内存管理区

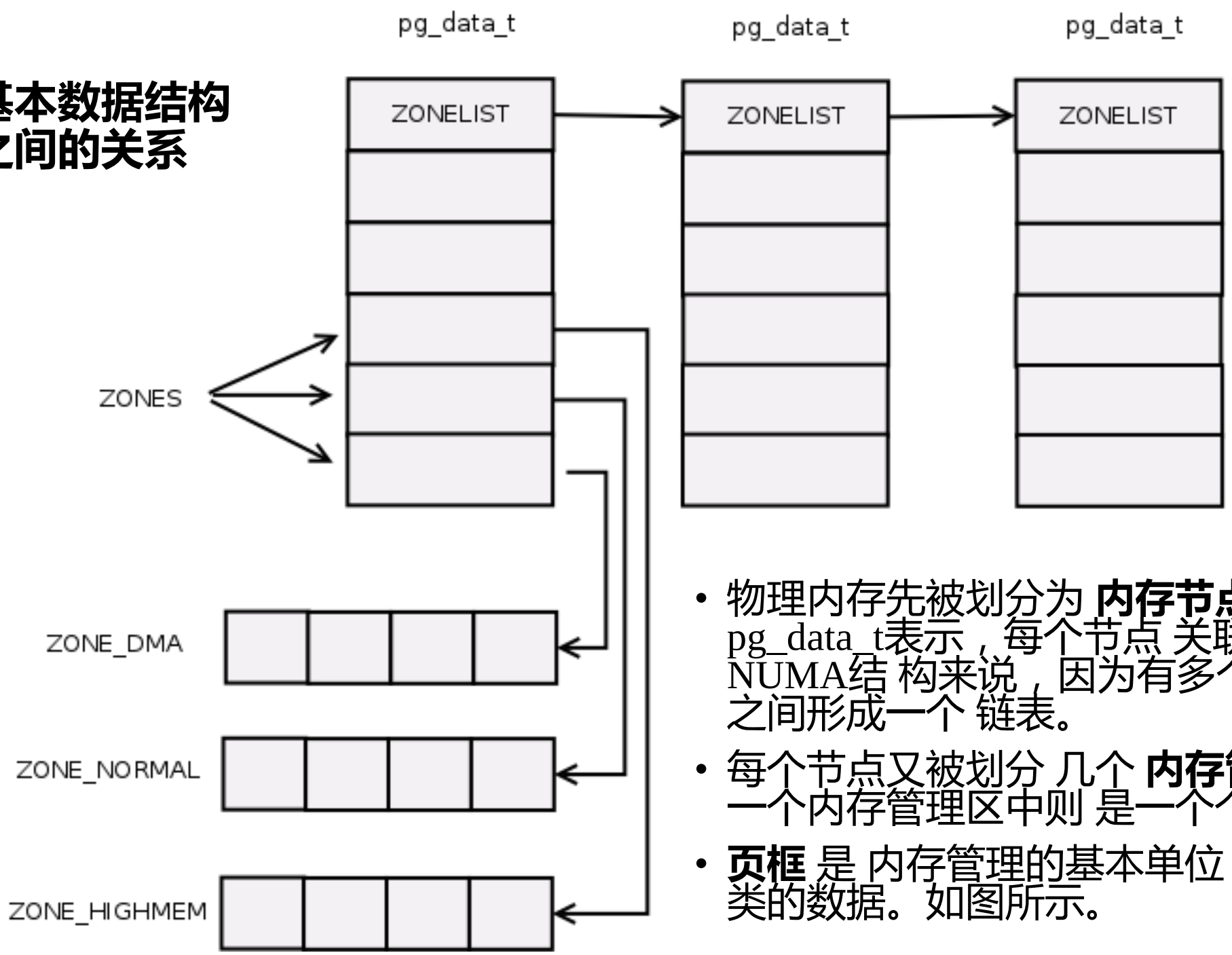
- 各个节点划分为若干个区，也是对物理内存的进一步细分。
- 通过下面几个宏，来标记物理内存不同的区：
- ZONE_DMA：标记适合DMA的内存区。
- ZONE_NORMAL：可以直接映射到内核空间的物理内存。
- ZONE_HIGHMEM：高端物理内存。

那么32位和64位操作系统对内存区管理上有什么差异？

- 最大的区别是64位操作系统，不再有高端内存的概念，可以支持大于4GB的内存寻址。
- ZONE_NORMAL空间将扩展到64GB或者128GB(64位系统上的映射更简单了)
- 如图所示。

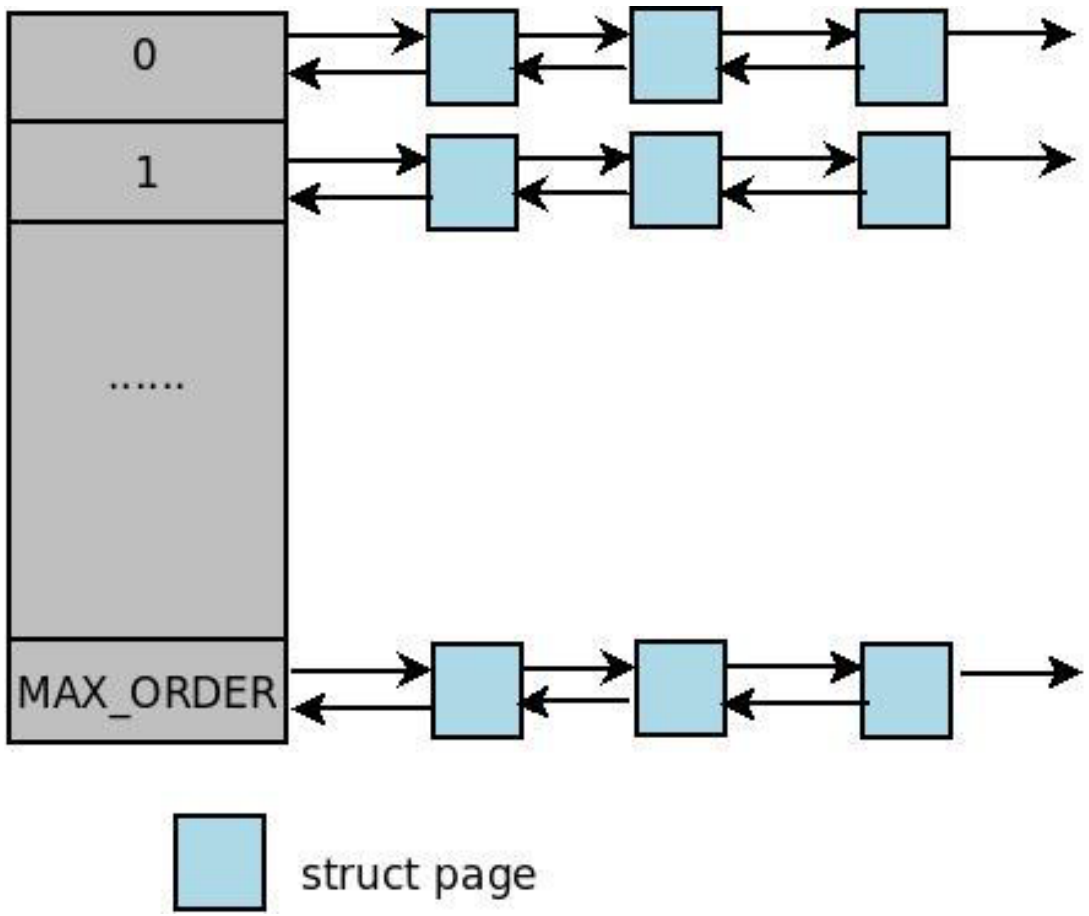


基本数据结构之间的关系



- 物理内存先被划分为 **内存节点**，内存区 用pg_data_t表示，每个节点 关联一个CPU，对于NUMA结 构来说，因为有多个节点， 因此各节点之间形成一个 链表。
- 每个节点又被划分 几个 **内存管理区**(ZONES) ，在一个内存管理区中则 是一个个的**页框**。
- **页框** 是 内存管理的基本单位，它 可以存放任何种类的数据。如图所示。

伙伴算法概述



- Linux内核中主要采用伙伴算法，来 分配 物理内存。
- 为啥叫伙伴算法呢？
- 大小相同、物理地址连续的两个页块被称为伙伴
- Linux的伙伴算法把所有的空闲页面分为多个块链表(该默认大小为11)个，
- 每个链表中的一个块，含有2的幂次个页面，即 **页块** 或简称 **块。**

与伙伴算法有关的数据结构

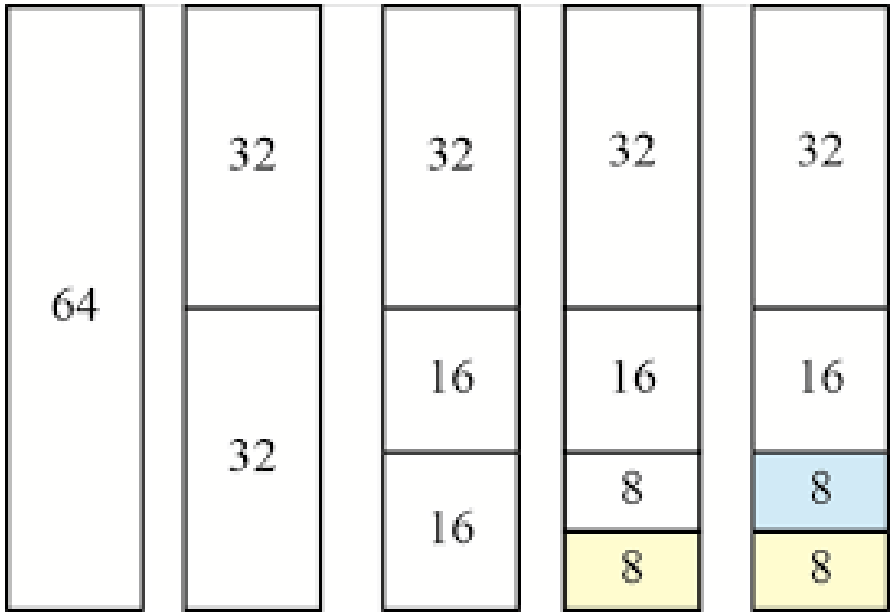
- 每个物理 **页框** 对应一个struct page实例。
- 每个内存区关联了一个struct zone实例，该结构中使用free_area数组对空闲**页框**进行管理

```
01 struct zone {
02 ……
03 struct free_area         free_area[MAX_ORDER];
04 ……
05 }
06
07 #ifndef CONFIG_FORCE_MAX_ZONEORDER
08 #define MAX_ORDER 11
09 #else
10 #define MAX_ORDER CONFIG_FORCE_MAX_ZONEORDER
11 #endif
```

```
1 struct free_area {
2         struct list_head        free_list[MIGRATE_TYPES];
3         unsigned long           nr_free;
4 };
```

在include/linux/mmzone.h

伙伴算法的分配原理



- 伙伴算法的分配原理是，如果分配阶为n的**页框**块，那么先从第n条**页框**块链表中，查找是否存在这么大小的空闲页块。
- 如果有则分配，否则在第n+1条链表中继续查找，直到找到为止。
- 如果申请大小为8的页块(分配阶为3)，但却在页块大小为32 (分配阶为5)的链表中，找到空闲块，
- 则先将这32个页面对半等分，前一半作为分配使用，另一半作为新元素插入下级大小为16的(分配阶为4)链表中；
- 继续将前一半大小为16的页块等分，一半分配，另一半插入大小为8的链表中。

课程思政

课程思政

发扬“敬业、精益、专注、创新”的工匠精神

2019年，国际政治关系的变化导致华为技术有限公司（以下简称华为）面临紧迫的Android（安卓）系统停供风险。在此背景下，华为毅然推出自主研发的操作系统——鸿蒙系统（HarmonyOS），以尽量减少Android系统停供所造成的损失。鸿蒙系统一经问世便引发产业界高度关注，这充分反映出大家对优秀国产操作系统的期盼。

近年来，我国软件产业的规模和效益同步提升，企业实力和创新能力显著增强。工业和信息化部数据显示，2018年，软件著作权登记量突破110万件，同比增长48%；2019年，软件业务收入已达71 768亿元，同比增长15.4%。由此可知，软件业的发展为国产操作系统的研发提供了基础性的技术支持。

发扬“敬业、精益、专注、创新”的工匠精神

随着云计算、大数据、人工智能等新一代信息技术与实体经济深度融合，工业互联网、工业大数据等新模式、新业态快速发展，工业已成为世界各国重点布局的领域。我国作为全世界唯一拥有联合国产业分类中所列全部工业门类的国家，在工业领域发展操作系统可谓具有独特优势。

国内大量处在工业2.0、3.0和4.0不同发展阶段的企业通过“企业上云”，加速实现数字化和网络化，伴随着工业互联网基础设施的快速迭代升级，国产操作系统不仅具有广阔的市场空间，也将推动传统制造业加速向数字化、网络化和智能化转变，为国产操作系统“换道超车”提供重大历史机遇，在这个过程中，我们一定要发扬“敬业、精益、专注、创新”的工匠精神。

课后练习题

1、请简述Linux对虚拟内存进行管理的五种机制。
2、进程运行时须有独占的堆和栈空间，请简述堆与栈的区别。
3、请简述一下虚拟内存区域(简称vma)是什么？
4、为什么把进程的用户空间要划分为一个个区间？
5、请对Linux快速创建进程的“写时复制”技术做一个介绍。
6、请简述什么是匿名映射？
7、有哪些情况会导致触发Linux的缺页(Page Fault)异常处理程序？
8、请对ELF可执行文件做一个介绍，其主要结构包含什么？
9、请简述内核采用了哪三种机制将高端内存映射到内核空间。
10、请简述为啥Linux操作系统需要引入slab机制？
11、请简述vmalloc()与kmalloc()之间的区别？
13、简述UMA和NUMA是啥含义？

谢 谢！

THANKS