操作系统大作业实验报告

自动化-**17364068**-王耀浩

实验一：虚存管理模拟程序：

1、文件：

vm1.c：未实现页置换，**TLB** 使用 **FIFO** 策略

vm2.c：未实现页置换，**TLB** 使用 **LRU** 策略

vm3.c：实现基于 **FIFO** 的页置换，**TLB** 使用 **FIFO** 策略

vm4.c：实现基于 **LRU** 的页置换，**TLB** 使用 **LRU** 策略

2、实验结果：

vm1.c：

```
Virtual address: 30032 Physical address: 592 Value: 0
Virtual address: 48065 Physical address: 25793 Value: 0
Virtual address: 6957 Physical address: 26413 Value: 0
Virtual address: 2301 Physical address: 35325 Value: 0
Virtual address: 7736 Physical address: 57912 Value: 0
Virtual address: 31260 Physical address: 23324 Value: 0
Virtual address: 17071 Physical address: 175 Value: -85
Virtual address: 8940 Physical address: 46572 Value: 0
Virtual address: 9929 Physical address: 44745 Value: 0
Virtual address: 45563 Physical address: 46075 Value: 126
Virtual address: 12107 Physical address: 2635 Value: -46
Number of Translated Addresses = 1000
Page Faults = 244
Page Fault Rate = 0.244
TLB Hits = 54
TLB Hit Rate = 0.054
```

vm2.c：

```
Virtual address: 48065 Physical address: 25793 Value: 0
Virtual address: 6957 Physical address: 26413 Value: 0
Virtual address: 2301 Physical address: 35325 Value: 0
Virtual address: 7736 Physical address: 57912 Value: 0
Virtual address: 31260 Physical address: 23324 Value: 0
Virtual address: 17071 Physical address: 175 Value: -85
Virtual address: 8940 Physical address: 46572 Value: 0
Virtual address: 9929 Physical address: 44745 Value: 0
Virtual address: 45563 Physical address: 46075 Value: 126
Virtual address: 12107 Physical address: 2635 Value: -46
Number of Translated Addresses = 1000
Page Faults = 244
Page Fault Rate = 0.244
TLB Hits = 55
TLB Hit Rate = 0.055
```

vm3.c:

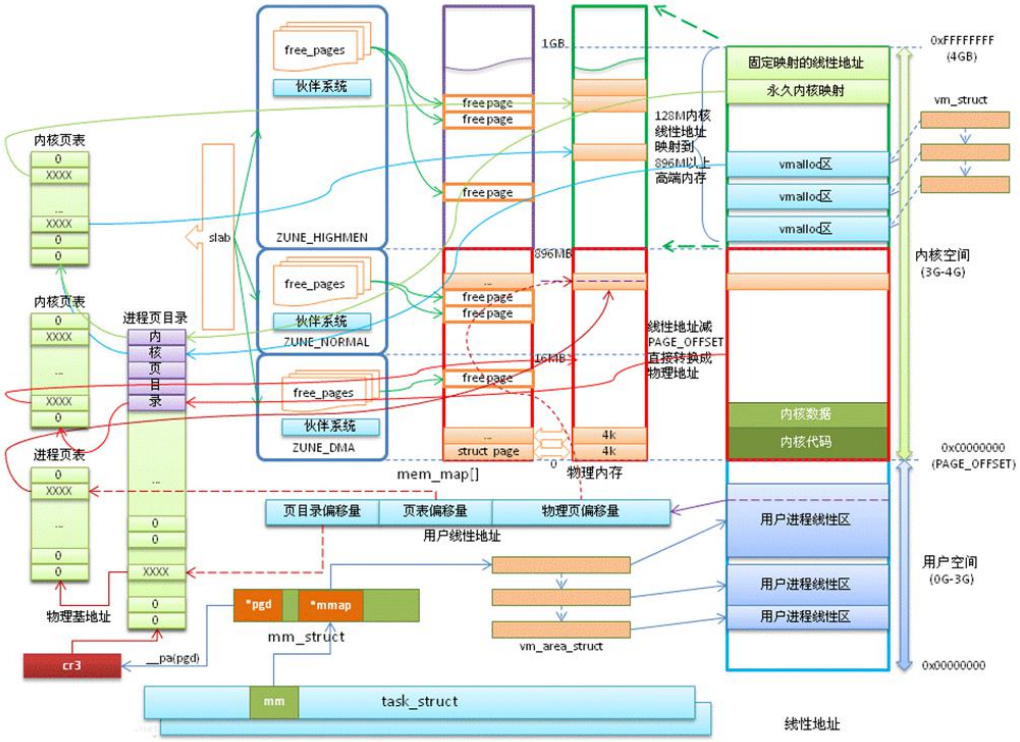
```
Virtual address: 30032 Physical address: 592 Value: 0
Virtual address: 48065 Physical address: 25793 Value: 0
Virtual address: 6957 Physical address: 26413 Value: 0
Virtual address: 2301 Physical address: 35325 Value: 0
Virtual address: 7736 Physical address: 57912 Value: 0
Virtual address: 31260 Physical address: 23324 Value: 0
Virtual address: 17071 Physical address: 175 Value: -85
Virtual address: 8940 Physical address: 46572 Value: 0
Virtual address: 9929 Physical address: 44745 Value: 0
Virtual address: 45563 Physical address: 46075 Value: 126
Virtual address: 12107 Physical address: 2635 Value: -46
Number of Translated Addresses = 1000
Page Faults = 244
Page Fault Rate = 0.244
TLB Hits = 54
TLB Hit Rate = 0.054
```

vm4.c:

```
Virtual address: 48065 Physical address: 25793 Value: 0
Virtual address: 6957 Physical address: 26413 Value: 0
Virtual address: 2301 Physical address: 35325 Value: 0
Virtual address: 7736 Physical address: 57912 Value: 0
Virtual address: 31260 Physical address: 23324 Value: 0
Virtual address: 17071 Physical address: 175 Value: -85
Virtual address: 8940 Physical address: 46572 Value: 0
Virtual address: 9929 Physical address: 44745 Value: 0
Virtual address: 45563 Physical address: 46075 Value: 126
Virtual address: 12107 Physical address: 2635 Value: -46
Number of Translated Addresses = 1000
Page Faults = 244
Page Fault Rate = 0.244
TLB Hits = 55
TLB Hit Rate = 0.055
```

实验二：Linux 内存管理实验：

1、解释图中每一类方框和箭头的含义，在代码树中寻找相关数据结构片段，做简单解释。



我们从右下到左上开始逐块分析了解

最下面 task_struct：

task_struct 是 Linux 内核的一种数据结构。它放在 RAM(运行内存)里并包含着进程的信息。每个进程都把自己的信息放在 task_struct 数据结构里。该结构体的定义位置是在内核 sched.h 中，地址：/usr/src/kernels/3.10.0-514.21.1.el7.x86_64/include/linux（3.10.0-514.21.1.el7.x86_64 是内核版本）

该结构体的主要信息：

1、进程状态 ,将纪录进程在等待,运行,或死锁

2、调度信息，由哪个调度函数调度,怎样调度等

3、进程的通讯状况

4、因为要插入进程树,必须有联系父子兄弟的指针，当然是 task_struct 型

5、时间信息，比如计算好执行的时间，以便 cpu 分配

6、标号 ,决定改进程归属

7、可以读写打开的一些文件信息

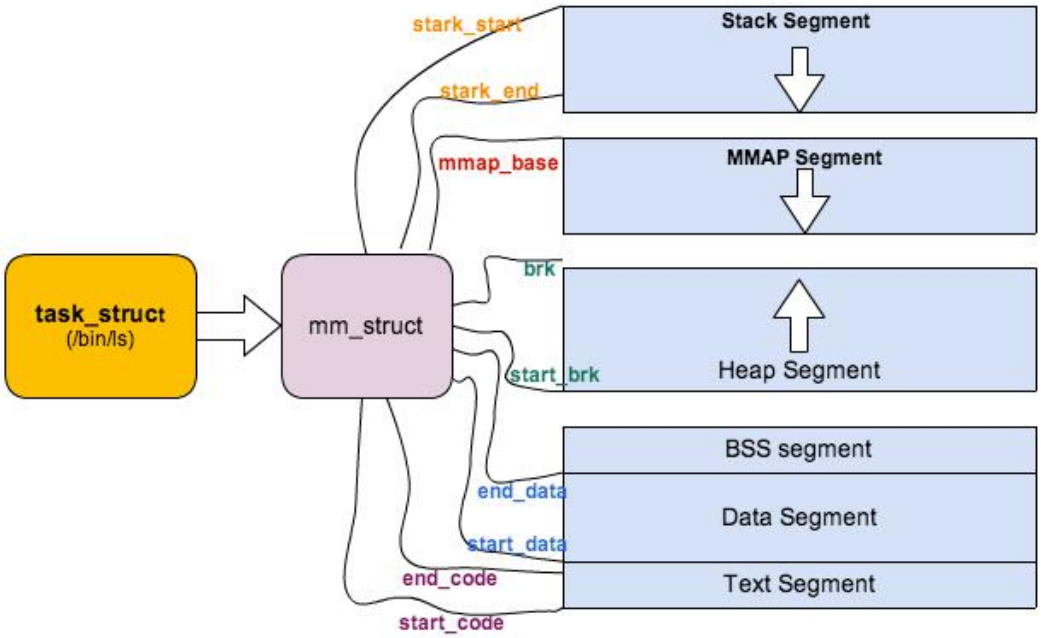
8、进程上下文和内核上下文

9、处理器上下文

10、内存信息

向上 mm_struct:

对于内核来说，内核线程与用户进程 都是 task_struct 这个数据结构的一个实例，task_struct 被称为进程描述符（process descriptor),因为它记录了这个进程所有的 context。其中有一个被称为'内存描述符'（memory descriptor)的数据结构 mm_struct，抽象并描述了 Linux 视角下管理进程地址空间的所有信息。该结构体定义在 include/linux/mm_types.h 中，其中的域抽象了进程的地址空间。

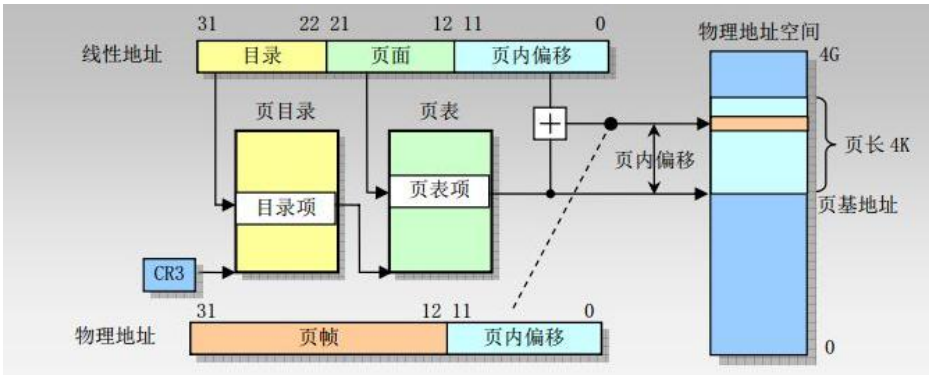


向左，*pdg-->cr3：

每个进程都有其自身的页面目录 PGD，Linux 将该目录的指针存放在与进程对应的内存结构 task_struct.(struct mm_struct)mm->pgd 中。每当一个进程被调度（schedule()）即将进入运行态时，Linux 内核都要用该进程的 PGD 指针设置 CR3（switch_mm()）。

当创建一个新的进程时，都要为新进程创建一个新的页面目录 PGD，并从内核的页面目录 swapper_pg_dir 中复制内核区间页面目录项至新建进程页面目录 PGD 的相应位置。

cr3 向上，进程页目录：



Linux 内存地址分页机制：

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。

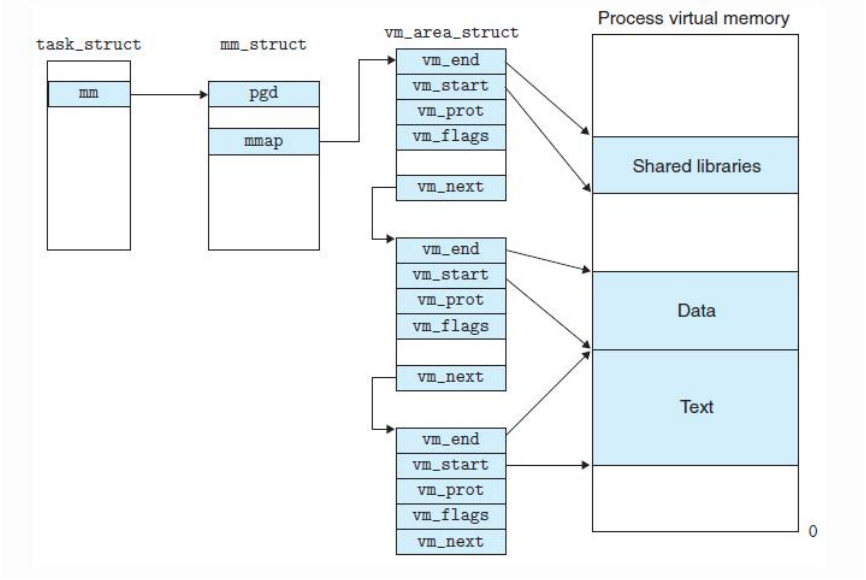
分页机制支持虚拟存储技术，在使用虚拟存储的环境中，大容量的线性地址空间需要使用小块的物理内存（RAM 或 ROM）以及某些外部存储空间来模拟。当使用分页时，每个段被划分成页面（通常每页为 4K 大小），页面会被存储于物理内存中或硬盘中。操作系统通过维护一个页目录和一些页表来留意这些页面。当程序（或任务）试图访问线性地址空间中的一个地址位置时，处理器就会使用页目录和页表把线性地址转换成一个物理地址，然后在该内存位置上执行所要求的操作。

mm_struct 向右，*mmap->vm_area_struct:

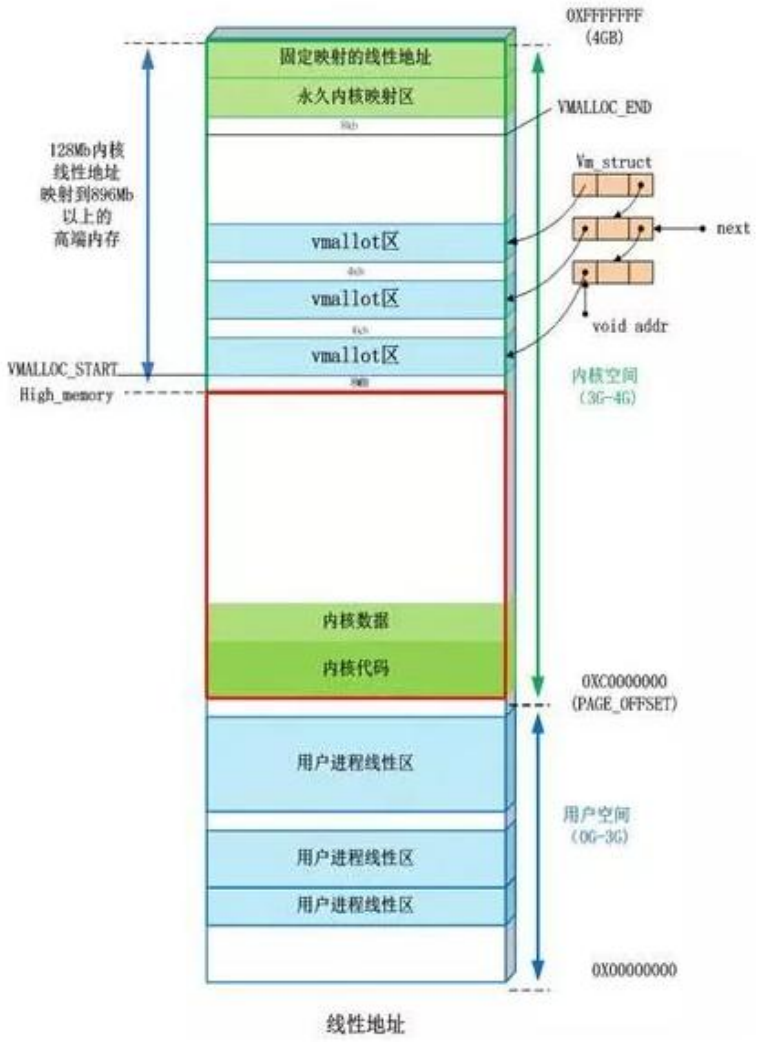
vm_area_struct 结构体是 include\linux\mm_types.h 文件中定义的结构体，简称 VMA，也被称为进程地址空间或进程线性区，在创建之后会插入到 mm->mm_rb 红黑树和 mm->mmap 链表中。该结构体用来定义 VMM 内存区域，是 linux 虚存管理的最基本的管理单元，它描述的是一段连续的、具有相同访问属性的虚拟内存空间，大小为物理内存页面的整数倍。

vm_area_struct 结构中包含区域起始和终止地址以及其他相关信息，同时也包含一个 vm_ops 指针，其内部可引出所有针对这个区域可以使用的系统调用函数。这样，进程对某一虚拟内存区域的任何操作需要用要的信息，都可以从 vm_area_struct 中获得。mmap 函数就是要创建一个新的 vm_area_struct 结构，并将其与文件的物理磁盘地址相连。

linux 内核使用 vm_area_struct 结构来表示一个独立的虚拟内存区域，由于每个不同质的虚拟内存区域功能和内部机制都不同，因此一个进程使用多个 vm_area_struct 结构来分别表示不同类型的虚拟内存区域。各个 vm_area_struct 结构使用链表或者树形结构链接，方便进程快速访问，如下图所示：



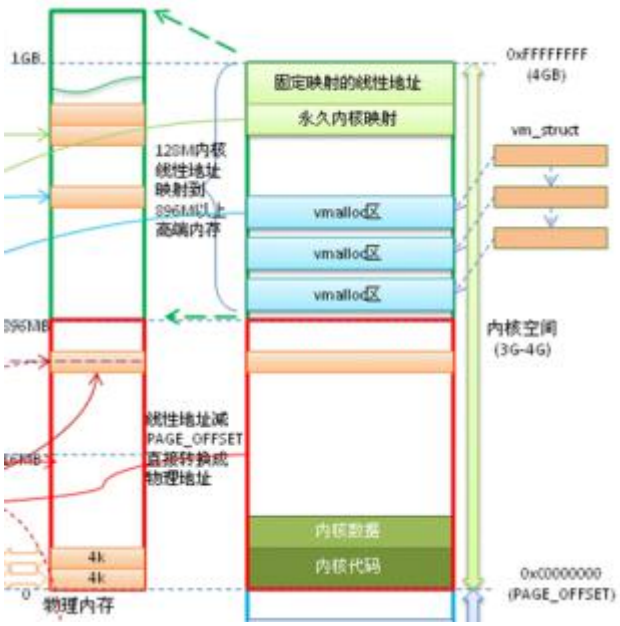
vm_area_struct 向右，内核态地址空间：



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

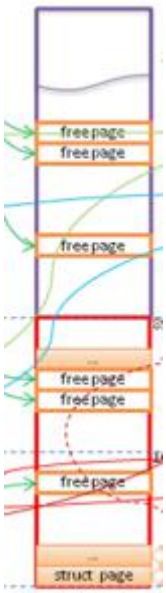
内核态地址空间向左展开，物理内存：
IA32 架构中内核虚拟地址空间只有 1GB 大小（从 3GB 到 4GB），因此可以直接将 1GB 大小的物理内存（即常规内存）映射到内核地址空间，但超出 1GB 大小的物理内存（即高端内存）就不能映射到内核空间。为此，内核采取了下面的方法使得内核可以使用所有的物理内存。
1).高端内存不能全部映射到内核空间，也就是说这些物理内存没有对应的线性地址。不过，内核为每个物理页框都分配了对应的页框描述符，所有的页框描述符都保存在 mem_map 数组中，因此每个页框描述符的线性地址都是固定存在的。内核此时可以使用 alloc_pages() 和 alloc_page()来分配高端内存，因为这些函数返回页框描述符的线性地址。
2).内核地址空间的后 128MB 专门用于映射高端内存，否则，没有线性地址的高端内存不能被内核所访问。这些高端内存的内核映射显然是暂时映射的，否则也只能映射 128MB 的高

端内存。当内核需要访问高端内存时就临时在这个区域进行地址映射，使用完毕之后再用来进行其他高端内存的映射。



物理内存向左，mem_map[]:

在 linux 内核中，所有的物理内存都用 struct page 结构来描述，这些对象以数组形式存放，而这个数组的地址就是 mem_map。

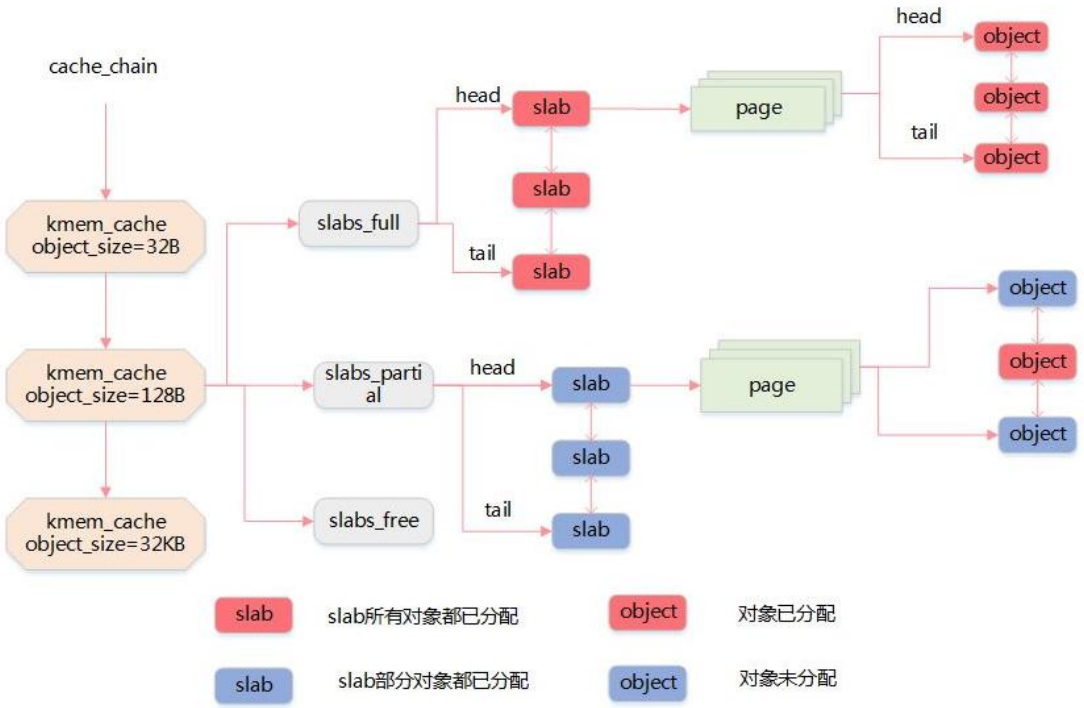


mem_map[]向左，伙伴系统：

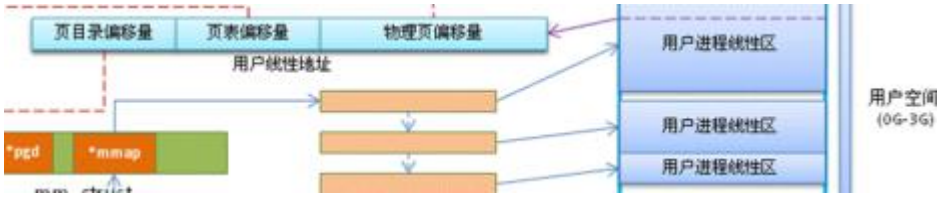
伙伴系统是一个结合了 2 的方幂个分配器和空闲缓冲区合并计技术的内存分配方案，其基本思想很简单. 内存被分成含有很多页面的大块，每一块都是 2 个页面大小的方幂. 如果找不到想要的块，一个大块会被分成两部分，这两部分彼此就成为伙伴. 其中一半被用来分配，而另一半则空闲. 这些块在以后分配的过程中会继续被二分直至产生一个所需大小的块. 当一个块被最终释放时，其伙伴将被检测出来，如果伙伴也空闲则合并两者.

伙伴系统向左，slab：

slab 是 Linux 操作系统的一种内存分配机制。其工作是针对一些经常分配并释放的对象，如进程描述符等，这些对象的大小一般比较小，如果直接采用伙伴系统来进行分配和释放，不仅会造成大量的内存碎片，而且处理速度也太慢。而 slab 分配器是基于对象进行管理的，相同类型的对象归为一类(如进程描述符就是一类)，每当要申请这样一个对象，slab 分配器就从一个 slab 列表中分配一个这样大小的单元出去，而当要释放时，将其重新保存在该列表中，而不是直接返回给伙伴系统，从而避免这些内碎片。slab 分配器并不丢弃已分配的对象，而是释放并把它们保存在内存中。当以后又要请求新的对象时，就可以从内存直接获取而不用重复初始化。

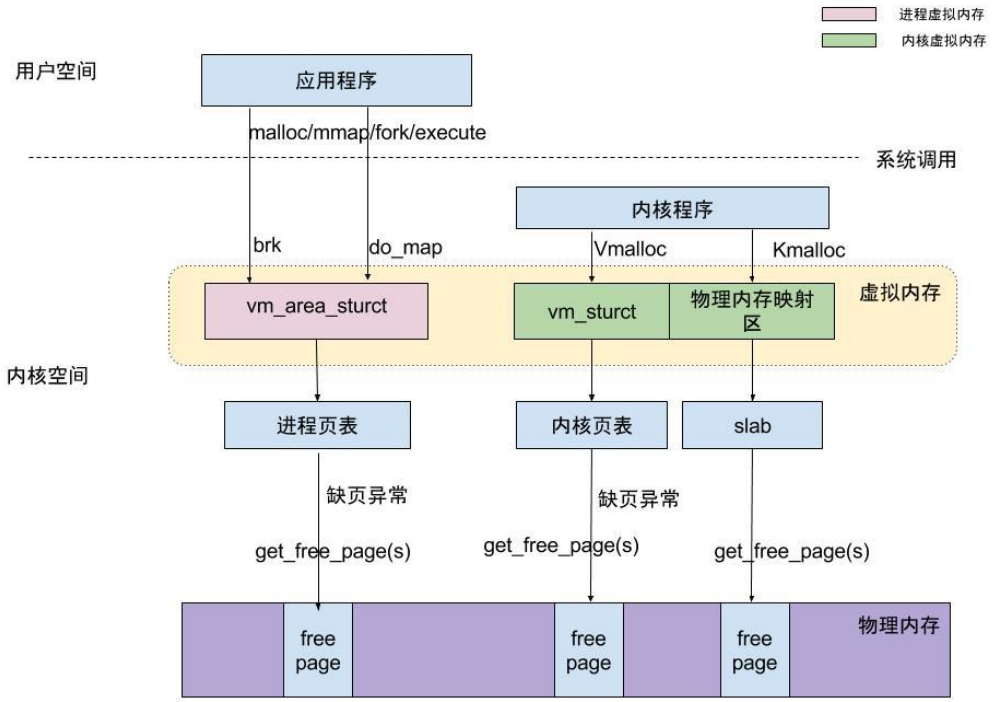


用户线性地址：



线性地址是一个 32 位无符号整数，可以用来表示高达 4GB 的地址，也就是，高达 4294967296 个内存单元。线性地址通常用十六进制数字表示，值的范围从 0x00000000 到 0xffffffff）。 [1] 程序代码会产生逻辑地址，通过逻辑地址变换就可以生成一个线性地址。如果启用了分页机制，那么线性地址可以再经过变换以产生一个物理地址。当采用 4KB 分页大小的时候，线性地址的高 10 位为页目录项在页目录表中的编号，中间 10 位为页表中的页号，其低 12 位则为偏移地址。如果是使用 4MB 分页机制，则高 10 位页号，低 22 位为偏移地址。如果没有启用分页机制，那么线性地址直接就是物理地址。

2、参考图 2 解释内核层不同内存分配接口的区别，包括__get_free_pages，kmalloc，vmalloc 等



__get_free_pages()函数：申请的内存位于物理内存的映射区域，而且在物理上也是连续的，它们与真实的物理地址只有一个固定的偏移，因此存在简单的线性关系；其申请的内存是一整页，一般为 128KB。

kmalloc()函数：用于申请较小的、连续的物理内存;申请的内存位置与__get_free_pages()的位置特性一样；

其有两个参数

void *kmalloc(size_t size, int flags);

size_t size：分配块大小

int flags：分配标志，用于控制 kmalloc 的行为

kmalloc()的底层依赖__get_free_page()实现，分配标志的前缀 GFP 正好是底层函数的缩写。kmalloc 申请的是较小的连续的物理内存，内存物理地址上连续，虚拟地址上也是连续的，使用的是内存分配器 slab 的一小片。申请的内存位于物理内存的映射区域。其真正的物理地址只相差一个固定的偏移。

vmalloc()函数：申请的虚拟内存与物理内存之间也没有简单的换算关系；vmalloc()一般用在只存在于软件中的较大顺序缓冲区分配内存，vmalloc()远大于__get_free_pages()的开销，为了完成 vmalloc()，新的页表需要被建立。所以效率没有 kmalloc 和__get_free_page 效率高。

3、参考 Anatomy of a Program in Memory 和 User-Level Memory Management 中例程，写一个实验程序 mtest.c，生成可执行程序 mtest；打印代码段、数据段、BSS，栈、堆等的相关地址

代码：见 mtest.c

实验结果：

```
osc@ubuntu:~/final-src-osc10e/ch10$ vim mtest.c
osc@ubuntu:~/final-src-osc10e/ch10$ gcc mtest.c -o mtest.o
osc@ubuntu:~/final-src-osc10e/ch10$ ./mtest.o
代码段地址
        主函数代码段:  0x400690
        function代码段 0x400626
栈地址
        栈级 1: stack_var地址: 0x7ffffed0a6c4
        栈级 2: stack_var地址: 0x7ffffed0a6a4
         alloca()开始: 0x7ffffed0a6b0
         alloca()结束: 0x7ffffed0a6cf
数据段地址
        data_var的地址: 0x601050
BSS地址:
        bss_var的地址: 0x60105c
堆地址:
        开始时堆末地址: 0x2539000
        新堆末地址: 0x2539020
        结束时堆末地址: 0x2539010
```

4、参考 How The Kernel Manages Your Memory，通过/proc/pid_number/maps，分析 mtest 各个内存段（参考链接）。绘制图表，解释输出的每一段的各种属性，包括每一列的内容。为了让 mtest 程序驻留内存，可以在程序末尾加上长时睡眠，并将 mtest 在后台运行。

在 mtest.c 的主函数尾部加入 sleep(600)

通过相关资料可知，maps 一共有六列

第一列代表内存段的虚拟地址

第二列代表执行权限

第三列代表在进程地址里的偏移量

第四列映射文件的主设备号和次设备号

第五列映像文件的节点号，即 inode

第六列是映像文件的路径

```
[2] 20056
[1]   Exit 127               ./mest.o
osc@ubuntu:~/final-src-osc10e/ch10$ 代码段地址
        主函数代码段:  0x4006d0
        function代码段 0x400666
栈地址
        栈级 1: stack_var地址: 0x7fff58f45a54
        栈级 2: stack_var地址: 0x7fff58f45a34
         alloca()开始: 0x7fff58f45a40
         alloca()结束: 0x7fff58f45a5f
数据段地址
        data_var的地址: 0x601058
BSS地址:
        bss_var的地址: 0x601064
堆地址:
        开始时堆末地址: 0x7c8000
        新堆末地址: 0x7c8020
        结束时堆末地址: 0x7c8010
#ps
osc@ubuntu:~/final-src-osc10e/ch10$ cd //
osc@ubuntu://$ cd proc/20056
osc@ubuntu://proc/20056$ sudo cat maps
[sudo] password for osc:
00400000-00401000 r-xp 00000000 08:01 23767                              /home/osc/final-src-osc10e/ch10/mtest.o
00600000-00601000 r--p 00000000 08:01 23767                              /home/osc/final-src-osc10e/ch10/mtest.o
00601000-00602000 rw-p 00001000 08:01 23767                              /home/osc/final-src-osc10e/ch10/mtest.o
007a7000-007c9000 rw-p 00000000 00:00 0                                  [heap]
7f027dfc6000-7f027e186000 r-xp 00000000 08:01 134524                     /lib/x86_64-linux-gnu/libc-2.23.so
7f027e186000-7f027e386000 ---p 001c0000 08:01 134524                     /lib/x86_64-linux-gnu/libc-2.23.so
7f027e386000-7f027e38a000 r--p 001c0000 08:01 134524                     /lib/x86_64-linux-gnu/libc-2.23.so
7f027e38a000-7f027e38c000 rw-p 001c4000 08:01 134524                     /lib/x86_64-linux-gnu/libc-2.23.so
7f027e38c000-7f027e390000 rw-p 00000000 00:00 0
7f027e390000-7f027e3b6000 r-xp 00000000 08:01 134521                     /lib/x86_64-linux-gnu/ld-2.23.so
7f027e5a9000-7f027e5ac000 rw-p 00000000 00:00 0
7f027e5b5000-7f027e5b6000 r--p 00025000 08:01 134521                     /lib/x86_64-linux-gnu/ld-2.23.so
7f027e5b6000-7f027e5b7000 rw-p 00026000 08:01 134521                     /lib/x86_64-linux-gnu/ld-2.23.so
7f027e5b7000-7f027e5b8000 rw-p 00000000 00:00 0
7fff58f27000-7fff58f48000 rw-p 00000000 00:00 0                          [stack]
7fff58ff1000-7fff58ff3000 r--p 00000000 00:00 0                          [vvar]
7fff58ff3000-7fff58ff5000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
osc@ubuntu://proc/20056$
```

5、参考 A Malloc Tutorial 以及相关资料（如链接）回答以下问题：

（1）用户程序的内存分配涉及 brk/sbrk 和 mmap 两个系统调用，这两种方式的区别是什么，什么时候用 brk/sbrk，什么时候用 mmap？

（2）应用程序开发时，为什么需要用标准库里的 malloc 而不是直接用这些系统调用接口？malloc 额外做了哪些工作？

（3）malloc 的内存分配，是分配的虚拟内存还是物理内存？两者之间如何转换？

（1） brk 是将数据段的最高地址指针_edata 往高地址推；

mmap 是在进程的虚拟地址空间中（堆和栈中间，文件映射区）找一块空闲的虚拟内存。

（2）malloc 可以按照需要分配内存大小，有些时候事先并不知道需要多大的内存。

（3）分配的是虚拟内存，通过 MMU 进行连接