

操作系统大作业 2 实验报告

智能工程学院 自动化 1 班 黄舒怡 17364029

○、GitHub 上传情况

 sysu17364029 操作系统大作业2

 locality.c	操作系统大作业2
 mtest.c	操作系统大作业2
 myalloc.c	操作系统大作业2
 readme.txt	操作系统大作业2
 vm.c	操作系统大作业2
 vmm.sh	操作系统大作业2
 vmmp.sh	操作系统大作业2

 [readme.txt](#)

【虚存管理模拟程序】

vm.c包含FIFO和LRU的TLB及Page Replacement

运行vmm.sh可直接打印FIFO和LRU分别运行vm（TLB和页置换统一策略）的Page-fault rate和TLB hit rate

[附加题]

locality.c用来生成addresses_locality.txt

运行vmmp.sh可直接用locality.c生成addresses_locality.txt，并打印用vm.c测试的结果

【Linux内存管理实验】

mtest.c为自己创建的例子，打印代码段、数据段、BSS，栈、堆等的相关地址

[附加题]

myalloc.c包括一对简单的函数myalloc/myfree，实现堆上的动态内存分配和释放，并含有测试函数

一共上传了 7 个文件，包含一个 readme.txt，各代码具体用法也在 readme.txt 中展示。

目录

一、虚存管理模拟程序	3
1. 使用 FIFO 和 LRU 分别执行 TLB 和页表换统一策略, 打印比较 Page-fault rate 和 TLB hit rate。	3
2. 附加题: trace 生成器	4
二、Linux 内存管理实验	6
1. 分析图 1, 解释每一类方框和箭头含义, 在代码树中寻找相关数据结构片段, 做简单解释。	6
1.1 虚拟内存	6
1.1.1 用户空间	7
1.1.2 内核空间	8
1.2 物理内存	9
1.2.1 物理内存管理逻辑	9
1.2.2 分页机制	12
1.3 由虚变实 (地址映射机制)	13
1.3.1 基本结构体和指针	14
1.3.2 用户空间	17
1.3.3 内核空间	18
2. 参考图 9 解释内核层不同内存分配接口的区别	20
2.1 get_free_pages	20
2.2 kmalloc	20
2.3 vmalloc	21
3. 写一个实验程序 mtest.c	22
4. 分析 mtest 各个内存段	23
5. Q & A	24
5.1 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?	24
5.2 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?	24
5.3 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?	25
6. 附加题: 模仿 malloc 接口, 实现一对简单函数 myalloc/myfree	25

一、虚存管理模拟程序

1. 使用 FIFO 和 LRU 分别执行 TLB 和页表换统一策略,打印比较 Page-fault rate 和 TLB hit rate。

```
(1) osc@ubuntu:~$ ./vm BACKING_STORE.bin addresses.txt -p fifo -n 256
Page Faults = 244
TLB Hits = 54
osc@ubuntu:~$ ./vm BACKING_STORE.bin addresses.txt
Page Faults = 244
TLB Hits = 54
(2) osc@ubuntu:~$ ./vm BACKING_STORE.bin addresses.txt -n 256 -p LRU
Page Faults = 244
TLB Hits = 55
(3) osc@ubuntu:~$ ./vm BACKING_STORE.bin addresses.txt -p lru -n 128
Page Faults = 537
TLB Hits = 55
(5) osc@ubuntu:~$ ./vm BACKING_STORE.bin addresses.txt -p FIFO -n 128
Page Faults = 538
TLB Hits = 54
osc@ubuntu:~$ ./vm BACKING_STORE.bin addresses.txt -n 128
Page Faults = 538
TLB Hits = 54
```

图 a

如图 [a](#), (1) 为 FIFO 的 TLB 的运行结果, (2) 为 LRU 的 TLB 的结果, (3) 为基于 LRU 的 Page Replacement 的结果, (5) 为基于 FIFO 的 Page Replacement 的结果。

两种策略的缺页率均略微超过 50%。由于 addresses.txt 随机生成, 缺页率几乎不受置换策略影响, 且应为物理页数/虚拟页数=128/256=50%左右。

为使结果更加清晰, 改写 vmm.sh, 运行得到如下图 [b](#) 结果(与上图 [a](#) 一致)。

```
osc@ubuntu:~$ bash vmm.sh
256 physical pages with FIFO replacement:
Page Faults = 244
TLB Hits = 54
256 physical pages with LRU replacement:
Page Faults = 244
TLB Hits = 55
128 physical pages with FIFO replacement:
Page Faults = 538
TLB Hits = 54
128 physical pages with LRU replacement:
Page Faults = 537
TLB Hits = 55
```

图 b

2. 附加题：trace 生成器

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <alloca.h>
4 #include <malloc.h>
5 #include <unistd.h>
6
7 int a[10][10000], c[10005], b[10000][10], ans[10][10];
8
9
10 int main() {
11     freopen("addresses_locality.txt", "w", stdout);
12     int cnt = 0;
13     for (int i = 0; i < 10; ++i) {
14         for (int j = 0; j < 10; ++j) {
15             for (int k = 0; k < 10000; ++k) {
16                 int *x = (int*)sbrk(sizeof(int)), *y = (int*)sbrk(sizeof(int)), *z = (int*)sbrk(sizeof(int));
17                 sbrk(256);
18                 *x = (long long)(&a[i][k]) % 65536; *y = (long long)(&b[k][j]) % 65536; *z = (long long)(&ans[i][j]) % 65536;
19                 cnt += 6;
20                 printf("%d\n%d\n%d\n", *x % 65536, *y % 65536, *z % 65536);
21                 printf("%lld\n%lld\n%lld\n", (long long)(x) % 65536, (long long)(y) % 65536, (long long)(z) % 65536);
22                 if (cnt >= 10000) return 0;
23             }
24         }
25     }
26 }
```

图 c

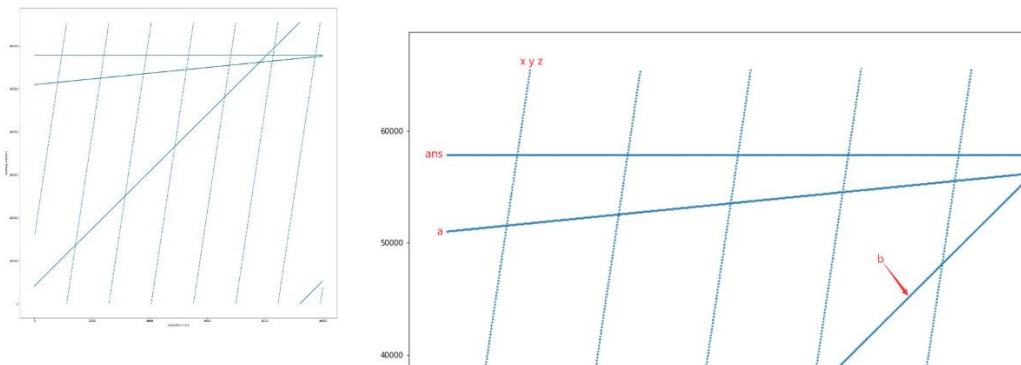


图 d

图 d 为运行时间-访问地址图。

(注：因为地址关于 65536 取余，所以每次运行结果会有微小区别。)

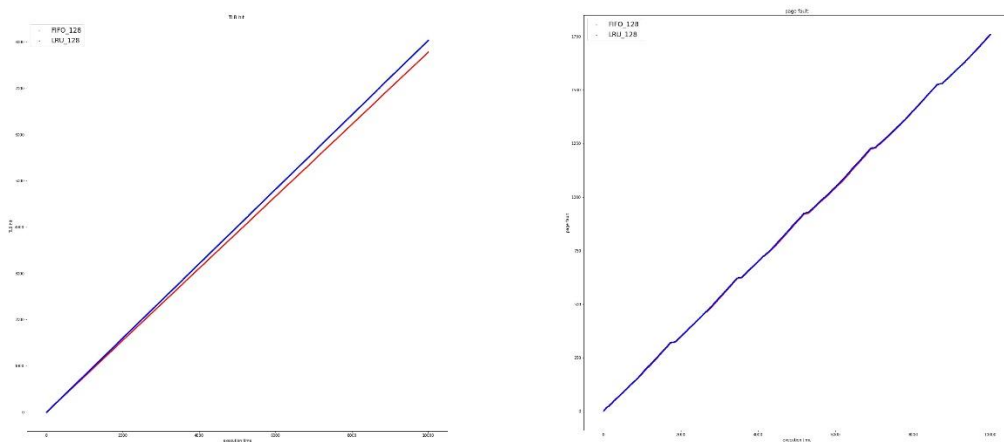


图 e

图 e 两张图依次为 FIFO (红) 和 LRU (蓝) 执行 TLB 的 TLB hit、page fault。

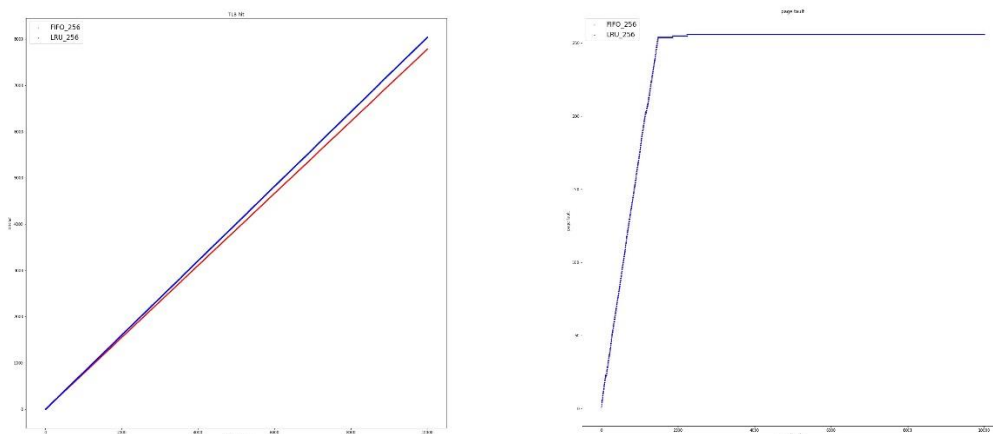


图 f

图 f 两张图依次为 FIFO (红) 和 LRU (蓝) 页置换的 TLB hit、page fault。

FIFO 为先进先出算法，当必须置换页面时，将选择最旧的页面。LRU 为最近最少使用算法，将置换最长时间没有使用的页。

从图 d 可以清楚地看出，ans 在短时间内会被重复访问（时间局部性）。当使用 FIFO 算法时，每隔 16 次将会置换最旧的页面，此时，当再次访问 ans 时，会出现没有命中的情况；而 LRU 则没有这种情况。由此，LRU 的 TLB hit 可能会略高于 FIFO。

因为 page size 较大，所以 page fault 没有看出显著影响。

二、Linux 内存管理实验

1. 分析图 1，解释每一类方框和箭头含义，在代码树中寻找相关数据结构片段，做简单解释。

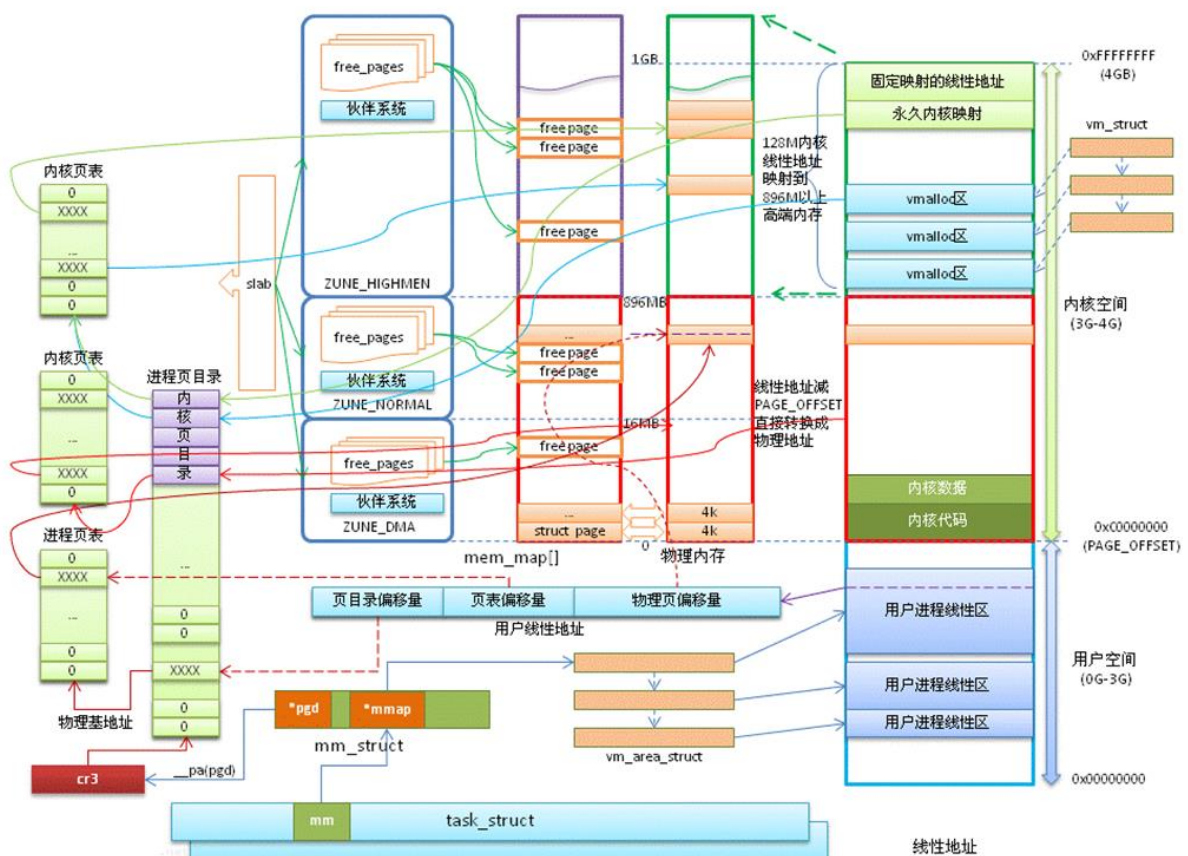


图 1. Linux 内核内存管理示意图 (IA_32)

以下将基本按从右往左、从上到下的顺序进行解释，即依次介绍：虚拟内存、物理内存、地址映射机制。

1.1 虚拟内存

Linux 操作系统采用**虚拟内存管理技术**，使得**每个进程**都有各自**互不干涉的**进程地址空间。该空间是块大小为 4G 的线性虚拟空间，用户所看到和接触到的都是该虚拟地址，无法看到实际的物理内存地址。利用这种虚拟地址不但能起到保护操作系统的效果，而且更重要的是，用户程序可使用比实际物理内存更大的

地址空间。

4G 的进程地址空间被人为的分为两个部分——**用户空间与内核空间**。如图 2 所示，用户空间从 0 到 3G（0xC0000000），内核空间占据 3G 到 4G。用户进程通常情况下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。只有用户进程进行系统调用等时刻可以访问到内核空间。

如果进程地址空间中的一个区域被分配给了进程，则内核会创建一个对应的线性区，也叫虚拟内存区（VMA）。属于同一个进程地址空间的线性区形成一个链表。

用户空间对应进程，所以每当进程切换，用户空间就会跟着变化；而内核空间是由内核负责映射，它并不会跟着进程改变，是固定的，Linux 以此方式让内核态进程共享代码段和数据段。内核空间地址有自己对应的页表，用户进程各自有不同的页表，如图 3 所示。

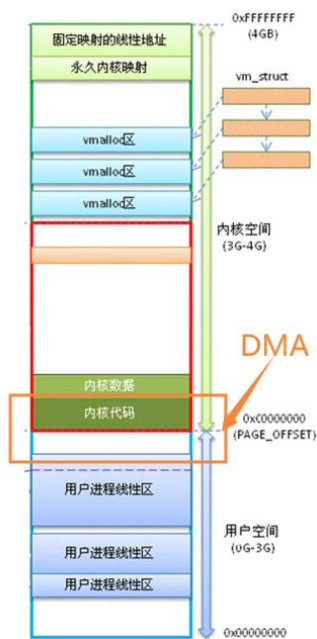


图 2 进程地址空间



图 3 页表和页目录

1.1.1 用户空间

每个进程的用户空间都是完全独立、互不相干的。

进程内存管理的对象是进程线性地址空间上的内存镜像，这些内存镜像其实就是进程使用的虚拟内存区域。为了方便管理，虚拟空间被划分为许多大小可变的(但必须是 4096 的倍数)内存区域，这些区域在进程线性地址中像停车位一样

有序排列。这些区域的划分原则是“将访问属性一致的地址空间存放在一起”，所谓访问属性在这里无非指的是“可读、可写、可执行等”。

1.1.2 内核空间

在内核空间中，我们分为三个部分——3G~3G+896M、3G+896M~3G+896M+8M 及 3G+896M+8M~4G。

第一个范围的地址属低端虚存，用来映射连续的物理页面，该区地址减 PAGE_OFFSET 直接转化为物理地址，从下到上依次为：16M 的 DMA 区、内核代码区、内核数据区。

第二个范围为安全保护区域，这 8M 大小的 gap 用于防止越界。

而最后一部分是高端虚存，包含 vmalloc 区、永久内核映射区、固定映射的线性地址区等。

我们从图 2 中看到，在高端虚存的 vmalloc 区存在一个结构体 vm_struct。可映射非连续的物理界面，常用于 vmalloc/vfree 的操作。

```
struct vm_struct {
    struct vm_struct *next;
    void *addr;
    unsigned long size;
    unsigned long flags;
    struct page **pages;
    unsigned int nr_pages;
    phys_addr_t phys_addr;
    const void *caller;
};
```

***next:** 指向下一个 vm，所有的 vm 组成一个链表。

***addr:** 指向虚拟地址。

size: 大小。

****pages:** 指向 vm 所映射的 page。

nr_pages: vm 所映射的 page 的个数。

phys_addr: 对应起始的物理地址，和*addr 相对应。

***caller:** 当前调用 vm 的指针。

1.2 物理内存

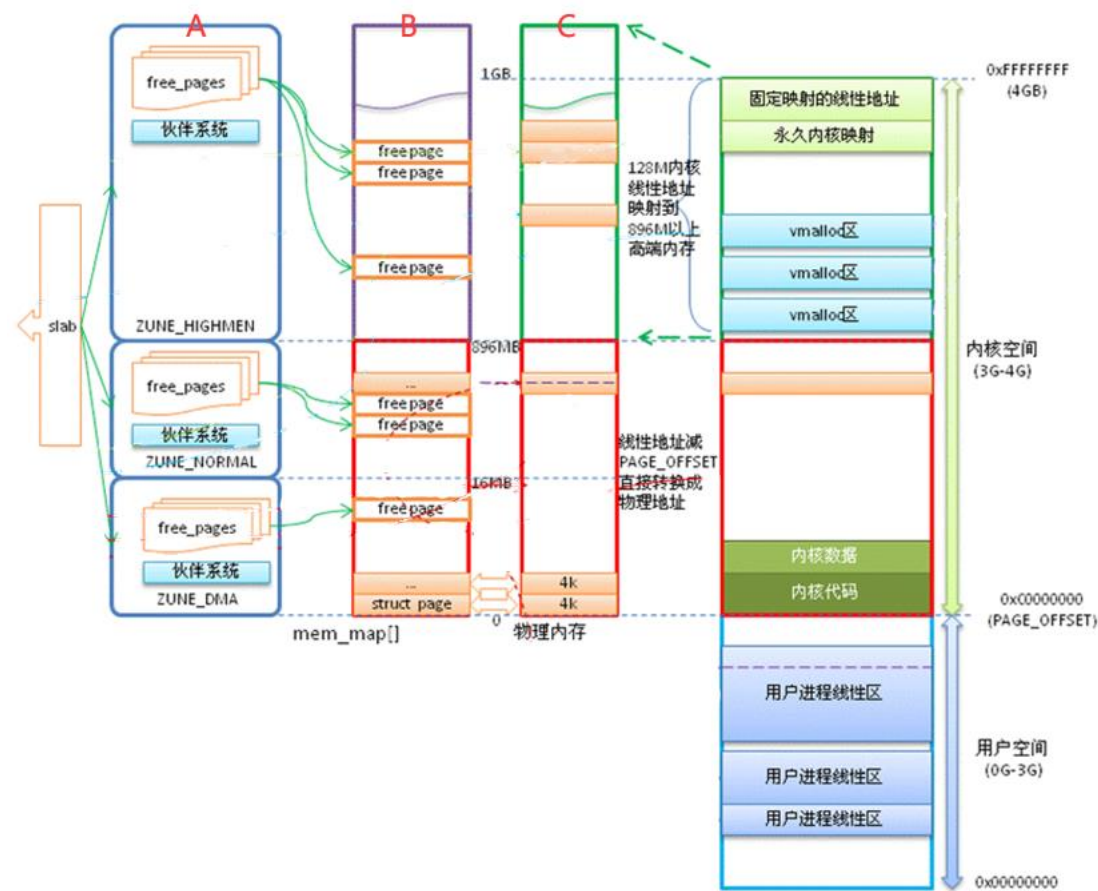


图 4 物理内存

物理内存是相对于虚拟内存而言的，指通过物理内存条而获得的内存空间。图 4 中 C 框对应的就是内存条。可以看到，128M 内核线性地址映射到 896M 以上高端内存，即绿色虚线标注的部分。

1.2.1 物理内存管理逻辑

Linux 中，用 node 对应一个内存 bank。node 在逻辑上分为管理区 zone，而每个 zone 又被划分为一系列 page，每个 page 大小为 4k。

(1) zone

如 A 框所示，zone 的类型有：

ZONE_DMA: 0~16M，该区域的物理页面专门供 I/O 设备的 DMA 使用。

ZONE_NORMAL: 16M~896M，该区域的物理页面是内核能够直接使用的，即可被内核直接映射到自己的虚拟地址空间的地址。

ZONE_HIGHMEM: 896M~结束, 该区域即为高端内存, 内核不能直接使用, 即不能被直接映射到内核的虚拟地址空间的地址。

三者中, ZONE_NORMAL 是影响系统性能最重要的管理区。

```
enum zone_type {
#ifdef CONFIG_ZONE_DMA
    ZONE_DMA,
#endif
#ifdef CONFIG_ZONE_DMA32
    ZONE_DMA32,
#endif
    ZONE_NORMAL,
#ifdef CONFIG_HIGHMEM
    ZONE_HIGHMEM,
#endif
    ZONE_MOVABLE,
#ifdef CONFIG_ZONE_DEVICE
    ZONE_DEVICE,
#endif
    __MAX_NR_ZONES
};
```

(2) page

[B](#)框对应的就是 page 结构。Linux 中, 运用 page 结构体来保存内核需要知道的所有物理内存信息, 对系统中每个物理页, 都有一个 page 结构体相对应。

```
struct page {
    /* First double word block */
    unsigned long flags;
    union {
        struct address_space *mapping;
        void *s_mem; /* slab first object */
        atomic_t compound_mapcount; /* first tail page */
        /* page_deferred_list().next -- second tail page */
    };
    /* Second double word */
    union {
        pgoff_t index; /* Our offset within mapping. */
        void *freelist; /* sl[aou]b first free object */
        /* page_deferred_list().prev -- second tail page */
    };
    union {
```

```

_slub_counter_t counters;
unsigned int active;      /* SLAB */
struct {                  /* SLUB */
    unsigned inuse : 16;
    unsigned objects : 15;
    unsigned frozen : 1;
};
struct {                  /* Page cache */
    atomic_t _mapcount;
    atomic_t _refcount;
};
};
union {
    struct list_head lru;
    struct dev_pagemap *pgmap;
    struct {              /* slub per cpu partial pages */
        struct page *next; /* Next partial slab */
    };
};
union {
    unsigned long private;
    struct kmem_cache *slab_cache; /* SL[AU]B: Pointer to slab */
};
#ifdef WANT_PAGE_VIRTUAL
    void *virtual;
#endif /* WANT_PAGE_VIRTUAL */
};

```

flags: page 状态的标志信息，如 dirty、locked in memory 等。每一位表示一种状态，所以至少可以同时表示出 32 种不同的状态。

***mapping:** 指向与该页相关的 address_space 对象。

index: 如果 page 是 file mapping 的一部分，它指明在文件中的偏移。如果 page 是交换缓存，则它指明在 address_space 所声明的对象(swapper_space)中的偏移。如果这个 page 是一个特殊的进程将要释放的一个 page 块，则这是一个 将要释放的 page 块的序列值，这个值在__free_page_ok()函数中设置。

_mapcount: 被页表映射的次数，也就是说该 page 同时被多少个进程共享。初始值为-1，如果只被一个进程的页表映射了，该值为 0。如果该 page 处于伙伴系统中，该值为 PAGE_BUDDY_MAPCOUNT_VALUE (-128)，内核通过判断该值是否为 PAGE_BUDDY_MAPCOUNT_VALUE 来确定该 page 是否属于伙伴系统

_refcount: 引用计数，表示内核中引用该 page 的次数。

lru: 链表头，用于在各种链表上维护该页，以便于按页将不同类别分组，主要有 3 个用途：伙伴算法，slab 分配器，被用户态使用或被当做页缓存使用

private: 私有数据指针，由应用场景确定其具体的含义，保存了一些和 mapping 有关的特定信息。

virtual: 对于如果物理内存可以直接映射内核的系统，我们可以之间映射出虚拟地址与物理地址的管理，但是对于需要使用高端内存区域的页，即无法直接映射到内核的虚拟地址空间，因此需要用 virtual 保存该页的虚拟地址。

mem_map 是一个内核全局变量，包含系统中所有的物理内存对应的 page 结构的数组，存放了所有的页描述符。一个页对应一个页描述符，即每个数组元素对应一个物理页，数组的索引对应于该物理页在物理内存中的位置，比如物理内存的第一个页对应于 mem_map[0]

```
#ifndef CONFIG_NEED_MULTIPLE_NODES
/* use the per-pgdat data instead for discontigmem - mbligh */
unsigned long max_mapnr;
EXPORT_SYMBOL(max_mapnr);

struct page *mem_map;
EXPORT_SYMBOL(mem_map);
#endif
```

1.2.2 分页机制

Linux 内核管理物理内存是通过**分页机制**实现的，它将整个内存划分成无数个 4k 大小的页，从而分配和回收内存的基本单位便是内存页了。利用分页管理有助于灵活分配内存地址，因为分配时不必要求必须有大块的连续内存，系统可以东一页、西一页的凑出所需要的内存供进程使用。虽然如此，但是实际上系统使用内存时还是倾向于分配连续的内存块，因为分配连续内存时，页表不需要更改，因此能降低 TLB 的刷新率。

(1) 伙伴系统

鉴于上述需求，内核分配物理页面时为了尽量减少不连续情况，采用了“伙

伴”关系来管理空闲页面。Linux 中空闲页面的组织和管理利用了伙伴关系，因此空闲页面分配时也需要遵循伙伴关系，最小单位只能是 2 的幂倍页面大小。

（2）页内内存分配——slab

所谓尺有所长，寸有所短。以页为最小单位分配内存对于内核管理系统中的物理内存来说的确比较方便，但内核自身最常使用的内存却往往是很小（远远小于一页）的内存块——比如存放文件描述符、进程描述符、虚拟内存区域描述符等行为所需的内存都不足一页。这些用来存放描述符的内存相比页面而言，就好比是面包屑与面包。一个整页中可以聚集多个这些小块内存；而且这些小块内存块也和面包屑一样频繁地生成/销毁。

为了满足内核对这种小内存块的需要，Linux 系统采用了一种被称为 slab 分配器的技术。Slab 分配器的核心思想就是“存储池”的运用。内存片段（小块内存）被看作对象，当被使用完后，并不直接释放而是被缓存到“存储池”里，留做下次使用，这无疑避免了频繁创建与销毁对象所带来的额外负载。

Slab 技术不但避免了内存内部分片带来的不便（引入 Slab 分配器的主要目的是为了减少对伙伴系统分配算法的调用次数——频繁分配和回收必然会导致内存碎片——难以找到大块连续的可用内存），而且可以很好地利用硬件缓存提高访问速度。

Slab 并非是脱离伙伴关系而独立存在的一种内存分配方式，slab 仍然是建立在页面基础之上，换句话说，Slab 将页面（来自于伙伴关系管理的空闲页面链表）撕碎成众多小内存块以供分配，slab 中的对象分配和销毁使用 `kmem_cache_alloc` 与 `kmem_cache_free`。

1.3 由虚变实（地址映射机制）

进程所能直接操作的地址都为虚拟地址。当进程需要内存时，从内核获得的仅仅是虚拟的内存区域，而不是实际的物理地址，进程并没有获得物理内存，获得的仅仅是对一个新的线性地址区间的使用权。虽然应用程序操作的对象是映射到物理内存之上的虚拟内存，但是处理器直接操作的却是物理内存。所以当应用程序访问一个虚拟地址时，首先必须将虚拟地址转化成物理地址，然后处理器才能解析地址访问请求。

实际的物理内存只有当进程真的去访问新获取的虚拟地址时，才会由“请求页机制”产生“缺页”异常，从而进入分配实际页面的例程。该异常是虚拟内存机制赖以存在的基本保证——它会告诉内核去真正为进程分配物理页，并建立对应的页表，这之后虚拟地址才实实在在地映射到了系统的物理内存上。

这种请求页机制把页面的分配推迟到不能再推迟为止，并不急于把所有的事情都一次做完。之所以能这么做是利用了内存访问的“局部性原理”，请求页带来的好处是节约了空闲内存，提高了系统的吞吐率。而实现这种机制，就基于下文将提到的 `fault` 操作。

1.3.1 基本结构体和指针

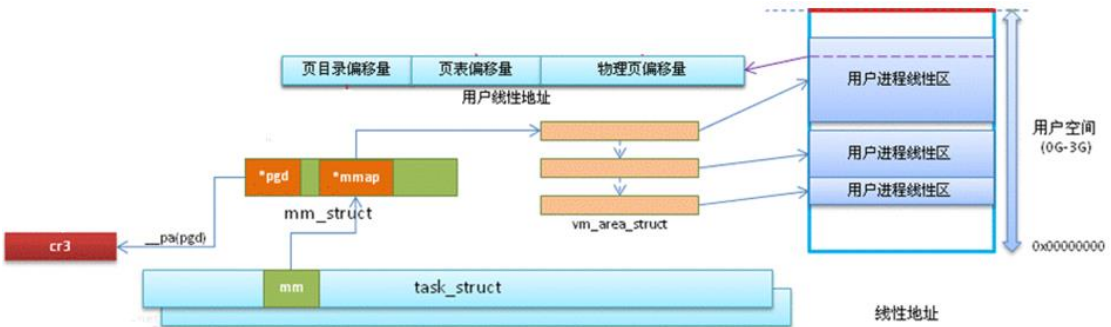


图 5 基本结构体

(1) task_struct

地址的转换工作需要通过查询页表才能完成。Linux 内核为每个进程，分配一个 `task_struct` 数据结构，并创建页表目录、页表。

```
struct task_struct {
    struct mm_struct    *mm;
    struct mm_struct    *active_mm;
    /* Per-thread vma caching: */
    struct vmacache     vmacache;
};
```

如上文所说，进程描述符 `task_struct` 用于描述一个进程。在进程控制块 `task_struct` 里定义了一个指向内存描述符 `mm_struct` 的指针 `*mm`。

(2) mm_struct

与进程地址空间有关的全部信息都包含在内存描述符 `mm_struct` 中。`mm_struct` 用于描述一个进程虚拟地址空间的布局。

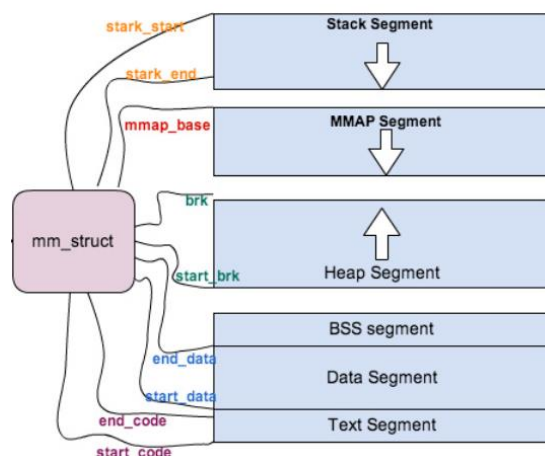


图 6 mm_struct 部分结构示意图

```
struct mm_struct {
    struct vm_area_struct *mmap;    /* list of VMAs */
    struct rb_root mm_rb;
    u32 vmacache_seqnum;           /* per-thread vmacache */
    pgd_t * pgd;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;
};
```

***mmap:** 指向内存线性区链表的首部。mmap 链表中的一个节点 vm_area_struct 记录了实际分配的一个内存区域。

mm_rb: 指向线性区对象的红黑树。

vmacache_seqnum: 内核在 3.16 版本后对 vma 的查找进行了优化，添加了此值而删去了 * mmap_cache。猜测此值和原来的 * mmap_cache 作用相似，即利用局部性原理存储最近用到的虚拟区间结构，提高效率。

*** pgd:** 指向的就是进程的页全局目录，当调度程序调度一个程序运行时，就将这个地址转成物理地址，并写入控制寄存器（CR3），如图 5 所示。

start_code, end_code, start_data, end_data: 分别为代码段、数据段的首地址和终止地址。

start_brk, brk, start_stack: 分别为进程堆的起始地址、堆的结束地址、用户态堆栈的起始地址。

arg_start, arg_end, env_start, env_end: 分别为命令行参数的起始地址、结束地址，环境变量的起始地址、结束地址。

(3) vm_area_struct

在 Linux 内核中对应进程内存区域的数据结构是：vm_area_struct，内核将每个内存区域作为一个单独的内存对象管理，相应的操作也都一致。

struct vm_area_struct 是虚存管理的最基本管理单元，它描述的是一段连续的、具有相同访问属性的虚存空间，该虚存空间的大小为物理内存页面的整数倍。

```
struct vm_area_struct {
    unsigned long vm_start;      /* Our start address within vm_mm. */
    unsigned long vm_end;        /* The first byte after our end address
                                   within vm_mm. */
    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next, *vm_prev;
    struct rb_node vm_rb;
    struct mm_struct *vm_mm;     /* The address space we belong to. */
    pgprot_t vm_page_prot;       /* Access permissions of this VMA. */
    unsigned long vm_flags;       /* Flags, see mm.h. */
    const struct vm_operations_struct *vm_ops; /* Function pointers to deal
                                                with this struct. */
    struct file * vm_file;        /* File we map to (can be NULL). */
};
```

vm_start 和 vm_end: 它们分别保存了该虚存空间的首地址和末地址后第一个字节的地址，以字节为单位，所以虚存空间范围可以用[vm_start, vm_end)表示。

***vm_next 和*vm_prev:** 分别纸箱 VMA 链表的前后成员。通常，进程所使用到的虚存空间不连续，且各部分虚存空间的访问属性也可能不同。所以一个进程的虚存空间需要多个 vm_area_struct 结构来描述。在 vm_area_struct 结构的数目较少的时候，各个 vm_area_struct 按照升序排序，以单链表的形式组织数据，vm_next 和 vm_prev 分别指向当前节点的下一个节点、上一个节点。

vm_rb: 当 vm_area_struct 结构的数据较多的时候，仍然采用链表组织的化，势必会影响到它的搜索速度。针对这个问题，每个进程结构体 mm_struct 中都创建一个红黑树，将本 VMA 作为一个节点加入到红黑树中，以提高 vm_area_struct 的搜索速度。

***vm_mm:** 指向 VMA 所属进程的 struct mm_struct 结构体。

vm_page_prot: 描述 VMA 访问权限，即虚存区域的页面保护特性。用于创建

区域中各页目录、页表项和存取控制标志，如 R/W, U/S, A, D, G 位等。

vm_flags: 主要保存 VMA 标志位，指出了虚存区域的操作特性。

***vm_ops:** 指向 vm_operations 结构。

***vm_file:** 指向被映射的文件，可以为空。

(4) vm_operations_struct

不同的虚拟区间其处理操作可能不同，Linux 在这里利用了面向对象的思想，即把一个虚拟区间看成一个对象，用 vm_area_structs 描述了这个对象的属性，其中的 vm_operation 结构描述了在这个对象上的操作。

```
struct vm_operations_struct {
    void(*open)(struct vm_area_struct * area);
    void(*close)(struct vm_area_struct * area);
    int(*fault)(struct vm_fault *vmf);
};
```

***open、*close:** 分别用于虚拟区间的打开、关闭

***fault:** 指向用于当虚存页面不在物理内存而引起的“缺页异常”时所应该调用的函数。

1.3.2 用户空间

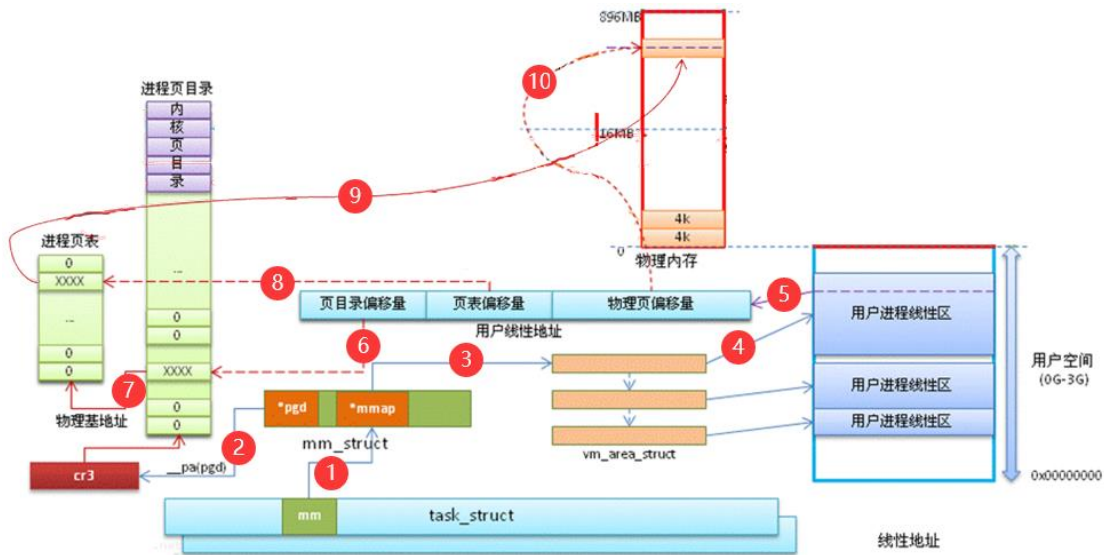


图 7 用户进程线性地址→物理地址

如上文所说，访问新获取的虚拟地址时，由“请求页机制”产生“缺页”异常，从而进入分配实际页面的例程。

- ① 从该进程的 task_struct 的 mm_struct 中读取*pgd 和*mmap;
 - ② *pgd 指向的就是进程的页全局目录，当调度程序调度一个程序运行时，就将这个地址转成物理地址，并写入控制寄存器（CR3），并获取进程页目录基地址；
 - ③ 从*mmap 中读取链表 vm_area_struct;
 - ④ & ⑤ 利用 vm_area_struct 在用户进程线性区读取用户线性地址，获取偏移量信息；
 - ⑥ & ⑦ 通过页目录偏移量在进程页目录中读取相应位置的信息，得到进程页表基地址；
 - ⑧ 通过页表偏移量，在进程页表中读取相应位置的信息；
 - ⑨ 通过进程页表中的信息，找到物理内存对应的物理页位置；
 - ⑩ 通过物理页偏移量，最终将线性地址转化为对应的物理地址。
- 理论上，图 7 的物理内存区应为内核的低端内存区，图片可能存在问题。

1.3.3 内核空间

(1) 低端内存

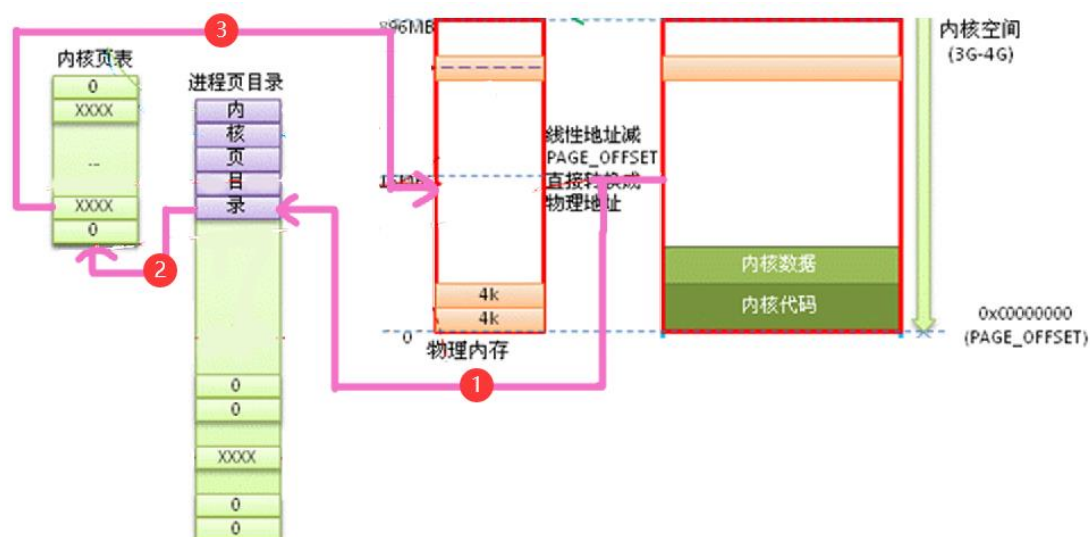


图 8 内核空间线性地址转换为物理地址

同用户进程线性区类似，先从进程页目录读出内核页表位置，在内核页表相应位置读出物理页位置。

理论上，图 8 展示的内核的直接映射区，线性地址减 PAGE_OFFSET 直接转换

成物理地址。猜测可能存在与用户进程线性区类似的转换机制。

(2) 高端内存

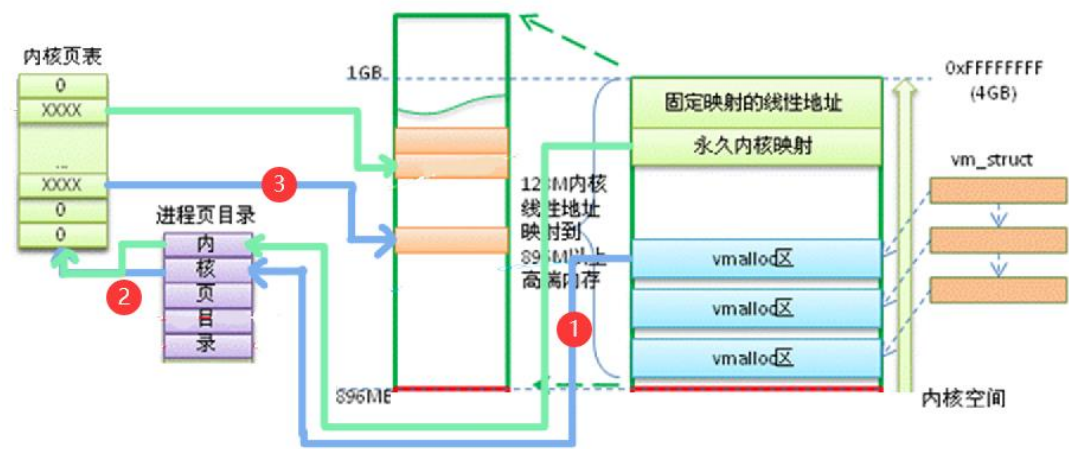


图 9 内核空间高端内存线性地址转换为物理地址

图 9 分别展示了 vmalloc 区和永久内核映射区的地址转换过程。具体过程与上文相仿，即“内核页目录-内核页表-物理内存”。可以看到，两个区域可能指向同一个内核页表。

主要参考链接：https://blog.csdn.net/csdn_whb/article/details/81251713

2. 参考图 9 解释内核层不同内存分配接口的区别

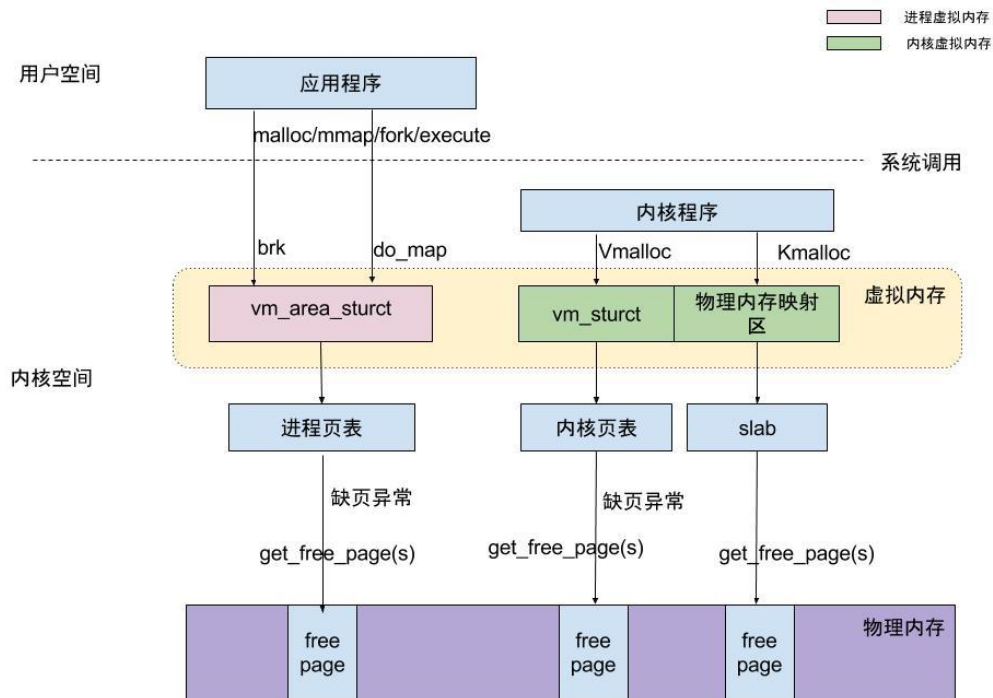


图 9. 用户空间和内核空间内存分配示意图

2.1 get_free_pages

内核中分配空闲页面的基本函数是 `get_free_page/get_free_pages`，它们或是分配单页或是分配指定的页面（2、4、8...512 页），一页的大小一般是 128K。

该函数申请的内存位于物理内存的映射区域（3G+896M）（低端内存），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。

2.2 kmalloc

与 `get_free_pages` 函数一样，`kmalloc` 函数申请的内存也位于物理内存的映射区域。

```
static __always_inline void *kmalloc(size_t size, gfp_t flags)
```

`kmalloc` 第一个参数是要分配块的大小，以字节为单位进行分配；第二个参数为分配标志，用于控制 `kmalloc` 的行为——最常用的分配标志是 `GFP_KERNEL`，

其含义是内核空间的进程中申请内存。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，使用的是内存分配器 slab 的一小片。使用 kmalloc 函数之后使用 kfree 函数。

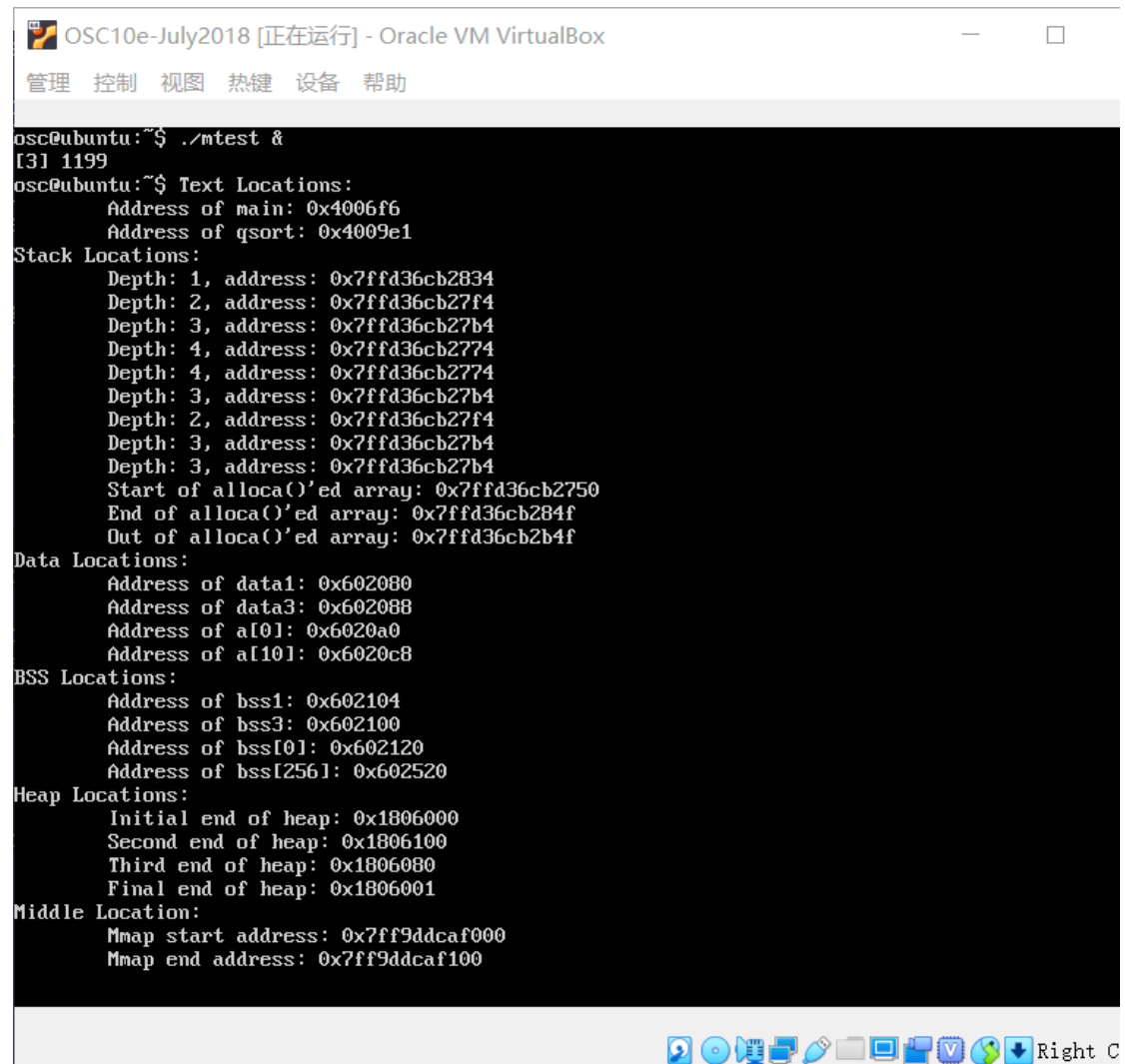
从内核内存分配的角度来讲，kmalloc 可被看成是 get_free_page(s) 的一个有效补充，内存分配粒度更灵活了。

2.3 vmalloc

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，位于高端内存（3G+896M 以上的内存）。

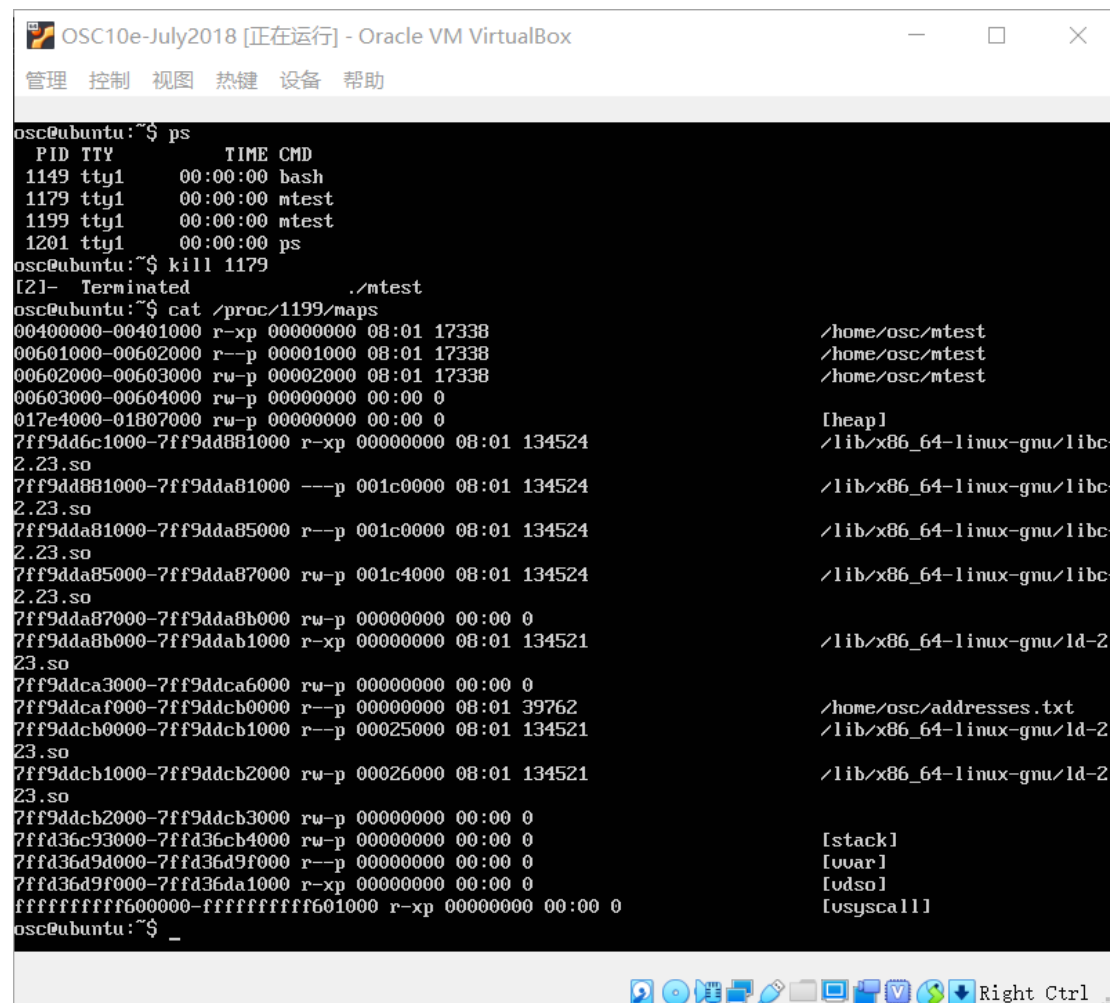
vmalloc() 一般用在只存在于软件中的较大顺序缓冲区（可远大于 128K，但必须是页大小的倍数）分配内存，vmalloc() 远大于 get_free_pages() 的开销。与用户进程相似，内核也有一个名为 init_mm 的 mm_struct 结构来描述内核地址空间，其中页表项 pgd=swapper_pg_dir 包含了系统内核空间（3G-4G）的映射关系。因此 vmalloc 分配内核虚拟地址必须更新内核页表。所以效率没有 kmalloc 和 __get_free_page 效率高。

3. 写一个实验程序 mtest.c



```
osc@ubuntu:~$ ./mtest &
[3] 1199
osc@ubuntu:~$ Text Locations:
    Address of main: 0x4006f6
    Address of qsort: 0x4009e1
Stack Locations:
    Depth: 1, address: 0x7ffd36cb2834
    Depth: 2, address: 0x7ffd36cb27f4
    Depth: 3, address: 0x7ffd36cb27b4
    Depth: 4, address: 0x7ffd36cb2774
    Depth: 4, address: 0x7ffd36cb2774
    Depth: 3, address: 0x7ffd36cb27b4
    Depth: 2, address: 0x7ffd36cb27f4
    Depth: 3, address: 0x7ffd36cb27b4
    Depth: 3, address: 0x7ffd36cb27b4
    Start of alloca()'ed array: 0x7ffd36cb2750
    End of alloca()'ed array: 0x7ffd36cb284f
    Out of alloca()'ed array: 0x7ffd36cb2b4f
Data Locations:
    Address of data1: 0x602080
    Address of data3: 0x602088
    Address of a[0]: 0x6020a0
    Address of a[10]: 0x6020c8
BSS Locations:
    Address of bss1: 0x602104
    Address of bss3: 0x602100
    Address of bss[0]: 0x602120
    Address of bss[256]: 0x602520
Heap Locations:
    Initial end of heap: 0x1806000
    Second end of heap: 0x1806100
    Third end of heap: 0x1806080
    Final end of heap: 0x1806001
Middle Location:
    Mmap start address: 0x7ff9ddcaf000
    Mmap end address: 0x7ff9ddcaf100
```

4. 分析 mtest 各个内存段



```
OSC10e-July2018 [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助

osc@ubuntu:~$ ps
  PID TTY          TIME CMD
 1149 tty1      00:00:00 bash
 1179 tty1      00:00:00 mtest
 1199 tty1      00:00:00 mtest
 1201 tty1      00:00:00 ps
osc@ubuntu:~$ kill 1179
[2]-  Terminated                  ./mtest
osc@ubuntu:~$ cat /proc/1199/maps
00400000-00401000 r-xp 00000000 08:01 17338          /home/osc/mtest
00601000-00602000 r--p 00001000 08:01 17338          /home/osc/mtest
00602000-00603000 rw-p 00002000 08:01 17338          /home/osc/mtest
00603000-00604000 rw-p 00000000 00:00 0
017e4000-01807000 rw-p 00000000 00:00 0
7ff9dd6c1000-7ff9dd881000 r-xp 00000000 08:01 134524      /lib/x86_64-linux-gnu/libc-
2.23.so
7ff9dd881000-7ff9dda81000 ---p 001c0000 08:01 134524      /lib/x86_64-linux-gnu/libc-
2.23.so
7ff9dda81000-7ff9dda85000 r--p 001c0000 08:01 134524      /lib/x86_64-linux-gnu/libc-
2.23.so
7ff9dda85000-7ff9dda87000 rw-p 001c4000 08:01 134524      /lib/x86_64-linux-gnu/libc-
2.23.so
7ff9dda87000-7ff9dda8b000 rw-p 00000000 00:00 0
7ff9dda8b000-7ff9ddab1000 r-xp 00000000 08:01 134521      /lib/x86_64-linux-gnu/ld-2.
23.so
7ff9ddca3000-7ff9ddca6000 rw-p 00000000 00:00 0
7ff9ddcaf000-7ff9ddcb0000 r--p 00000000 08:01 39762          /home/osc/addresses.txt
7ff9ddcb0000-7ff9ddcb1000 r--p 00025000 08:01 134521      /lib/x86_64-linux-gnu/ld-2.
23.so
7ff9ddcb1000-7ff9ddcb2000 rw-p 00026000 08:01 134521      /lib/x86_64-linux-gnu/ld-2.
23.so
7ff9ddcb2000-7ff9ddcb3000 rw-p 00000000 00:00 0
7ffd36c93000-7ffd36cb4000 rw-p 00000000 00:00 0
7ffd36d9d000-7ffd36d9f000 r--p 00000000 00:00 0
7ffd36d9f000-7ffd36da1000 r-xp 00000000 00:00 0
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0
osc@ubuntu:~$ _
```

每一行为一个 vm_area_struct。

第一列：起始地址-结束地址，本段内存映射的虚拟地址空间范围，对应 vm_start 和 vm_end；

第二列：权限，r 读，w 写，x 可执行，p 私有，对应 vm_flags；

第三列：若 vm_area_struct 自文件映射，该列表示地址在文件中的偏移，对应 vm_pgoff，否则对应 vm_start；

第四列：主设备号和次设备号；

第五列：文件的索引节点号；

第六列：若自文件映射，则为文件名；否则为作用。

这段程序从上到下依次为：

① mtest 代码段（可读可执行）

- ② mtest 数据段（可读不可写）
- ③ mtest BSS 段（可读可写）
- ④ brk_offset
- ⑤ 堆
- ⑥ 几个外部库的组件
- ⑦ mmap_offset
- ⑧ 几个外部库的组件，中间有一个替换成了一个自己链接的文件
- ⑨ mmap_offset
- ⑩ 栈
- ⑪ vvar, vdso, vsyscall, 是内核中一些组件的外部虚拟化，这样在调用的时候就不用切换到内核状态，提高效率

5. Q & A

5.1 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？

（1）brk/sbrk 直接移动堆指针的位置，如果向高位移动，实际上扩张了堆，也就是在堆的高地址端分配了一部分内存。由于 brk/sbrk 直接移动堆指针位置，因此在释放内存时，低位内存必须在高位内存释放后才能释放。

（2）mmap 在堆和栈中间分配一块连续的内存区域，独立于程序的已有区域（实际上新建了一个 vm_area_struct），并映射到物理页。mmap 申请的内存可以自由释放。

5.2 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？

在 malloc 中，当分配小块内存（小于一个阈值，似乎是 M_MMAP_THRESHOLD=128K）时，调用 brk/sbrk 分配内存，而分配大块内存（大于上述阈值）时，调用 mmap 分配内存，一方面避免用 mmap 分配内存带来内存碎片，同时防止频繁使用 mmap 和 munmap 分配和回收资源导致大量缺页中断；另一方面，malloc 本身

维护一个内存资源库，通过堆上的内存碎片重用，减少 brk/sbrk 和 mmap 的调用次数，通过降低系统调用频率提高效率。

开发应用程序时，往往面临较为复杂的内存管理场景，若还需要思考何时使用何种系统调用分配内存，将大大增加工作难度。malloc 已经提供了高效的内存分配策略，因此开发应用程序时直接使用，可以增加开发效率，提高程序性能。

5.3 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？

malloc 的实现与具体硬件无关，分配的是虚拟内存，只有当分配的内存区域首次被访问时，才会发生缺页中断，页表中更新对应表项，同时数据实际装入物理内存。当页表项被换出，物理内存装入其它数据，此时虽然内存未被回收，但其对应的物理内存已经失效，只具有虚拟内存的地址。

6. 附加题：模仿 malloc 接口，实现一对简单函数 myalloc/myfree

```
ossc@ubuntu:~$ gcc myalloc.c -o myalloc -w
ossc@ubuntu:~$ ./myalloc
Allocate int a...
Init 0x11fd000
Allocate: 0x11fd000-0x11fd020, prev: (nil)
011db000-011fe000 rw-p 00000000 00:00 0 [heap]
Allocate long b...
Allocate: 0x11fd020-0x11fd040, prev: 0x11fd000
011db000-011fe000 rw-p 00000000 00:00 0 [heap]
Allocate int c...
Allocate: 0x11fd040-0x11fd060, prev: 0x11fd020
011db000-011fe000 rw-p 00000000 00:00 0 [heap]
Allocate int[65536] d...
Allocate: 0x11fd060-0x123d080, prev: 0x11fd040
011db000-0123e000 rw-p 00000000 00:00 0 [heap]
Allocate char e...
Allocate: 0x123d080-0x123d0a0, prev: 0x11fd060
011db000-0123e000 rw-p 00000000 00:00 0 [heap]
10
Free starts.
    end: 0x123d0a0, cb + size: 0x11fd040
Free Ends.
Free starts.
    end: 0x123d0a0, cb + size: 0x11fd060
Free Ends.
Free starts.
    end: 0x123d0a0, cb + size: 0x123d0a0
    Free 0x123d080-0x123d0a0
    tail: 0x11fd060, end: 0x123d080, free: 0
Free Ends.
Allocate 10 f...
Allocate: 0x11fd020-0x11fd040, prev: 0x11fd000
011db000-0123e000 rw-p 00000000 00:00 0 [heap]
Free starts.
    end: 0x123d080, cb + size: 0x123d080
```

```

Free 0x11fd060-0x123d080
tail: 0x11fd040, end: 0x11fd060, free: 1
Free 0x11fd040-0x11fd060
tail: 0x11fd020, end: 0x11fd040, free: 0
Free Ends.
Free starts.
end: 0x11fd040, cb + size: 0x11fd040
Free 0x11fd020-0x11fd040
tail: 0x11fd000, end: 0x11fd020, free: 0
Free Ends.
Free starts.
end: 0x11fd020, cb + size: 0x11fd020
Free 0x11fd000-0x11fd020
tail: (nil), end: 0x11fd000
Free Ends.
0x11fd000 0x11fd000
011db000-011fd000 rw-p 00000000 00:00 0 [heap]

```

malloc 维护一个块链表，每次先从表中找第一个大小足够的空闲块分配，否则 sbrk() 开辟新块，保证块大小为 16 的倍数。free 设置对应块为空闲，同时如果堆顶已经空闲，就 sbrk() 一路释放掉堆顶的空闲块。

运行过程中可以发现：释放 b 和 c 并没有实际上的虚拟空间释放，释放 e 的时候则会有；f 被分配到了原先 b 的位置，因此释放 d 的时候会先后释放 d 和 c 的空间，然后释放 f 的时候会释放原先 b 的空间，最后释放 a。