

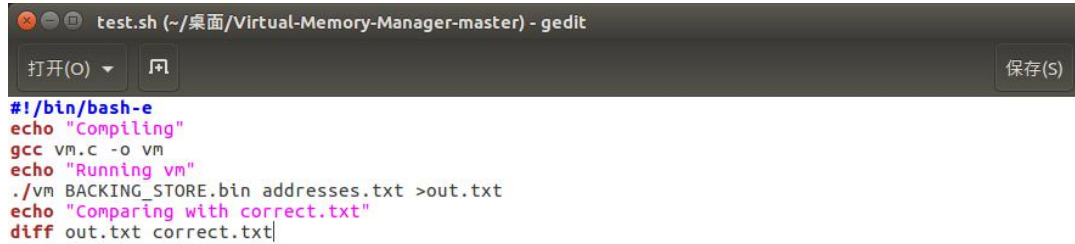
# 操作系统大作业 2

智能科学与技术 2 班 17363056 刘思瑶

## 一、虚存管理模拟程序

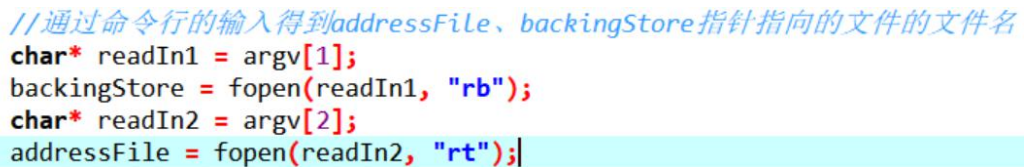
vm.c 文件中实现了通过命令行控制物理内存帧数(128 或者 256)和使用的策略 (FIFO 或者 LRU)，这里 TLB 和页面置换使用的是同一种策略。

在最初的时候按照第一小题要求的内容编写 test.sh 文件及对应的 vm.c 文件，此时 test.sh 文件内容如图 1 (a) 所示，这时 vm.c 中读取命令行输入的文件名的代码部分截图如图 1(b) 所示，其他部分代码同最终版代码相同。运行 test.sh 后的结果如图 1(c) 所示。



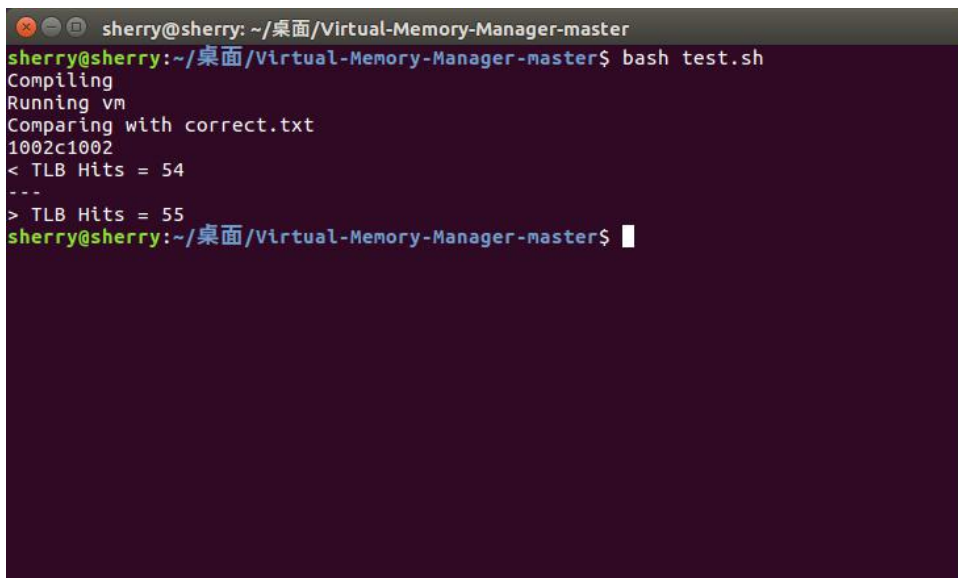
```
#!/bin/bash-e
echo "Compiling"
gcc vm.c -o vm
echo "Running vm"
./vm BACKING_STORE.bin addresses.txt >out.txt
echo "Comparing with correct.txt"
diff out.txt correct.txt
```

图 1 (a)



```
//通过命令行的输入得到addressFile、backingStore指针指向的文件的文件名
char* readIn1 = argv[1];
backingStore = fopen(readIn1, "rb");
char* readIn2 = argv[2];
addressFile = fopen(readIn2, "rt");
```

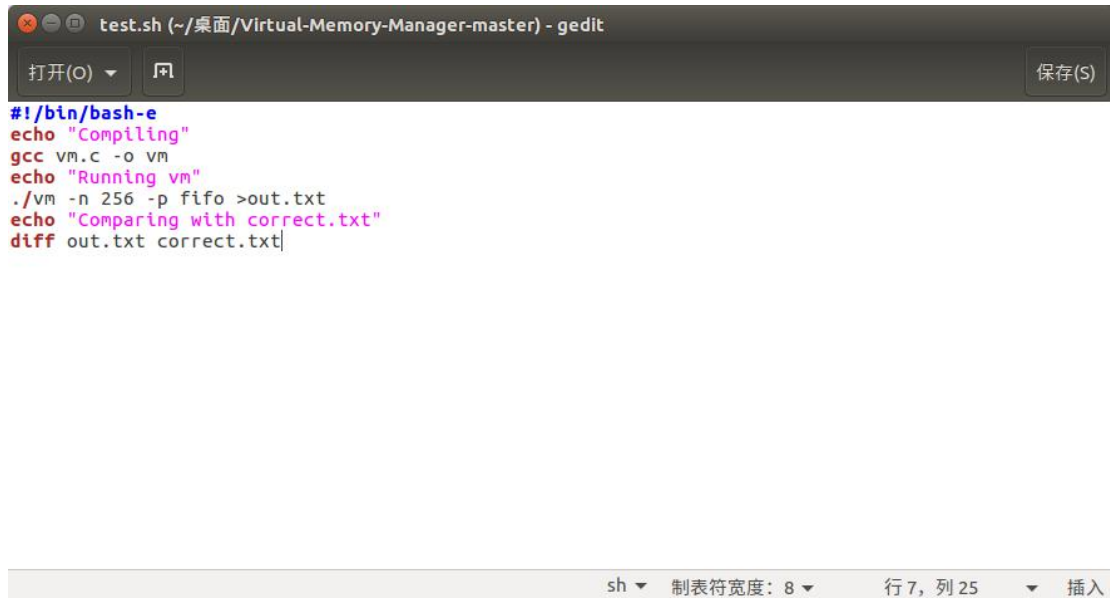
图 1 (b)



```
sherry@sherry: ~/.桌面/Virtual-Memory-Manager-master
sherry@sherry:~/.桌面/Virtual-Memory-Manager-master$ bash test.sh
Compiling
Running vm
Comparing with correct.txt
1002c1002
< TLB Hits = 54
---
> TLB Hits = 55
sherry@sherry:~/.桌面/Virtual-Memory-Manager-master$
```

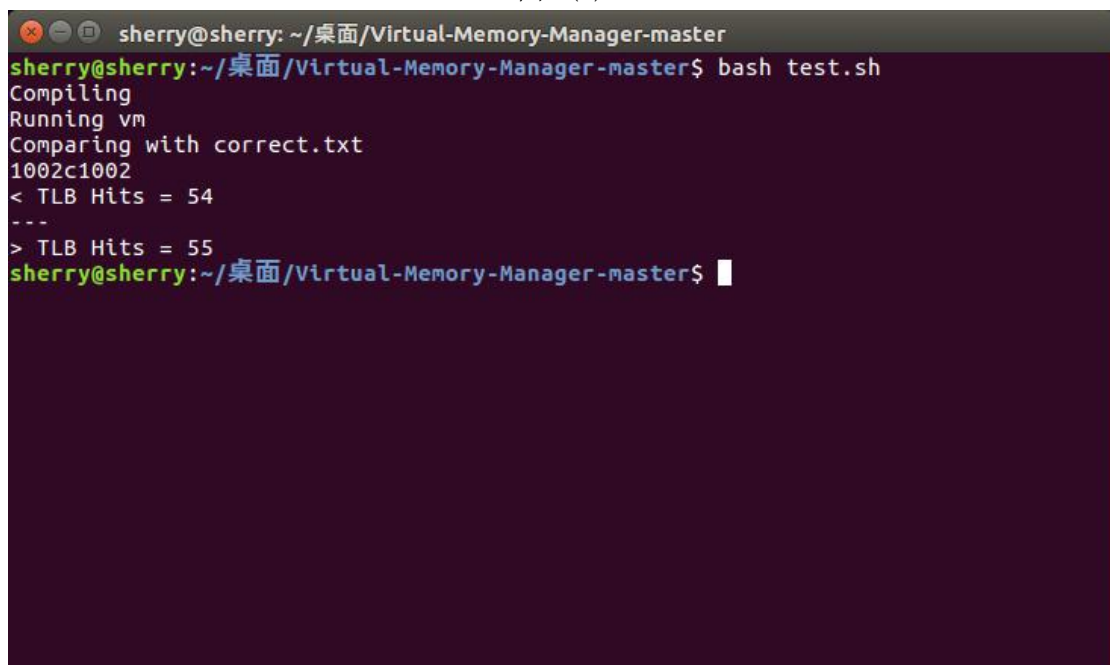
图 1 (c)

最终版本的 `vm.c` 按照老师要求的使用 `getopt` 函数，通过 `-n`、`-p` 来获取参数，进行操作，当物理帧数设置为 128 时需要页面转换。这里总共有四种组合，一是在物理帧数为 256 的情况下，采用 FIFO 策略，及与上面相同，此时的 `test.sh` 文件如图 2(a) 所示，运行结果如图 2(b) 所示；二是在物理帧数为 256 的情况下，采用 LRU 策略，此时的 `test.sh` 文件如图 3(a) 所示，运行结果如图 3(b) 所示；三是在物理帧数为 128 的情况下，采用 FIFO 策略，此时的 `test.sh` 文件如图 4(a) 所示，运行结果如图 4(b) 所示；四是在物理帧数为 128 的情况下，采用 LRU 策略，此时的 `test.sh` 文件如图 5(a) 所示，运行结果如图 5(b) 所示。



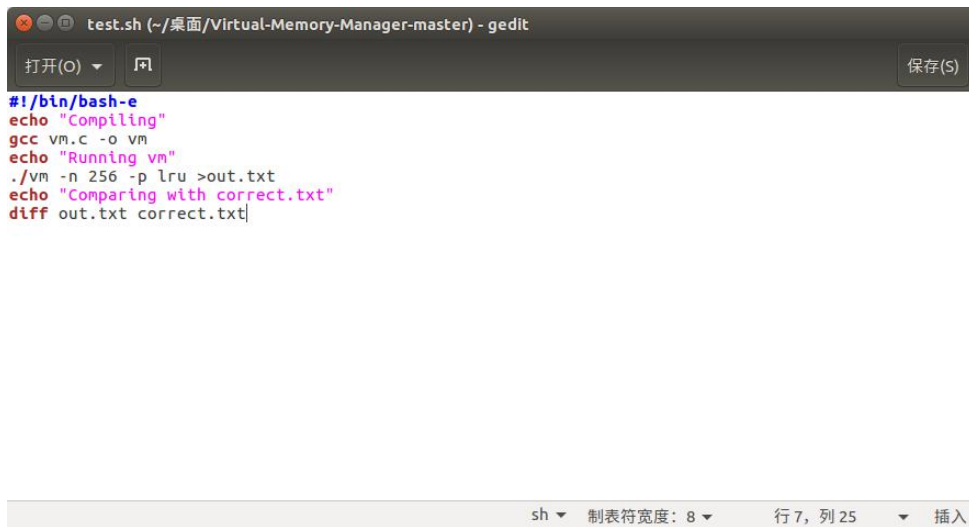
```
#!/bin/bash-e
echo "Compiling"
gcc vm.c -o vm
echo "Running vm"
./vm -n 256 -p fifo >out.txt
echo "Comparing with correct.txt"
diff out.txt correct.txt
```

图 2(a)



```
sherry@sherry: ~/桌面/Virtual-Memory-Manager-master
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$ bash test.sh
Compiling
Running vm
Comparing with correct.txt
1002c1002
< TLB Hits = 54
---
> TLB Hits = 55
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$
```

图 2(b)

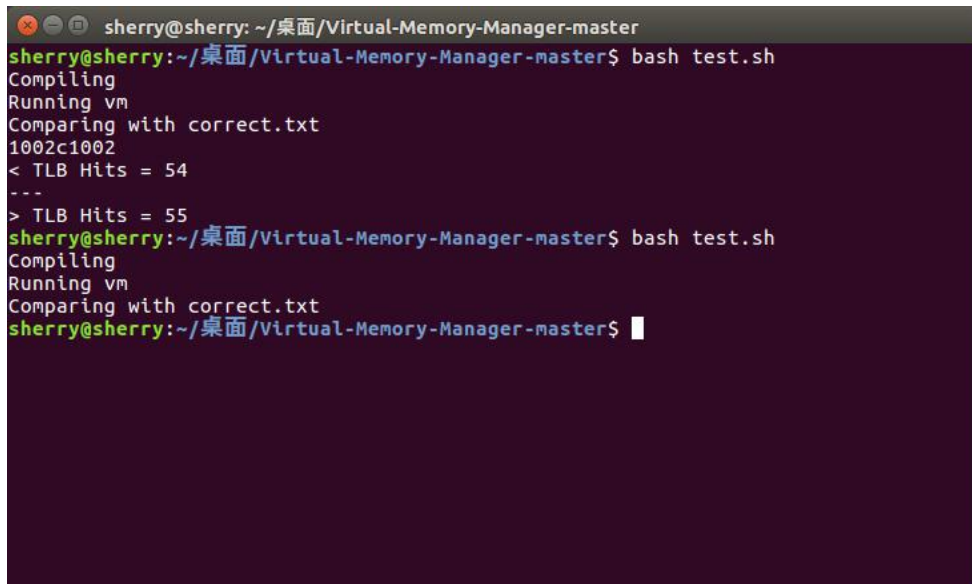


```
test.sh (~/.桌面/Virtual-Memory-Manager-master) - gedit
打开(O) 保存(S)

#!/bin/bash-e
echo "Compiling"
gcc vm.c -o vm
echo "Running vm"
./vm -n 256 -p lru >out.txt
echo "Comparing with correct.txt"
diff out.txt correct.txt
```

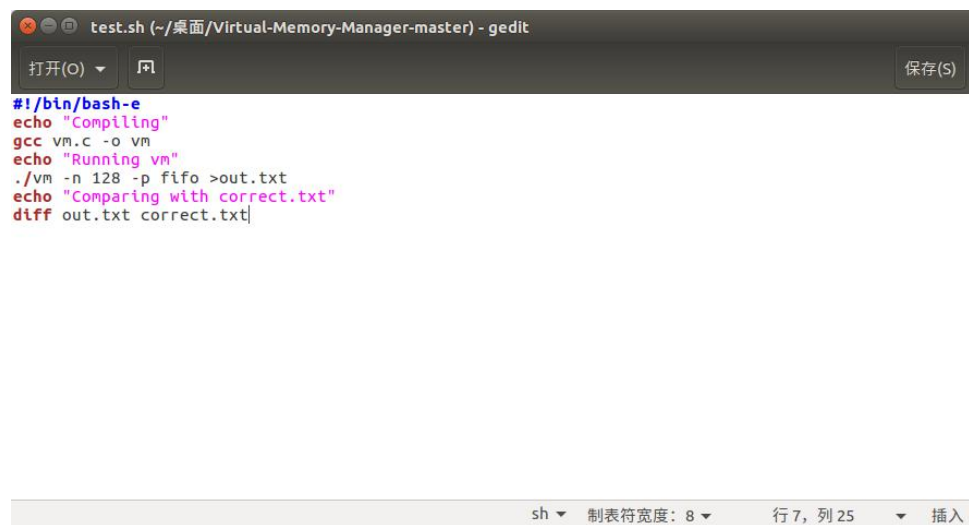
sh 制表符宽度: 8 行 7, 列 25 插入

图 3(a)



```
sherry@sherry: ~/桌面/Virtual-Memory-Manager-master
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$ bash test.sh
Compiling
Running vm
Comparing with correct.txt
1002c1002
< TLB Hits = 54
---
> TLB Hits = 55
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$ bash test.sh
Compiling
Running vm
Comparing with correct.txt
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$
```

图 3(b) (中间的命令行操作)



```
test.sh (~/.桌面/Virtual-Memory-Manager-master) - gedit
打开(O) 保存(S)

#!/bin/bash-e
echo "Compiling"
gcc vm.c -o vm
echo "Running vm"
./vm -n 128 -p fifo >out.txt
echo "Comparing with correct.txt"
diff out.txt correct.txt
```

sh 制表符宽度: 8 行 7, 列 25 插入

图 4(a)

```
sherry@sherry: ~/桌面/Virtual-Memory-Manager-master
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$ bash test.sh
Compiling
Running vm
Comparing with correct.txt
1002c1002
< TLB Hits = 54
---
> TLB Hits = 55
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$ bash test.sh
Compiling
Running vm
Comparing with correct.txt
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$ bash test.sh
Compiling
Running vm
Comparing with correct.txt
1001,1002c1001,1002
< Page Faults = 538
< TLB Hits = 54
---
> Page Faults = 244
> TLB Hits = 55
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$
```

图 4(b) (最下面的命令行操作)

```
test.sh (~/桌面/Virtual-Memory-Manager-master) - gedit
打开(O) 保存(S)

#!/bin/bash-e
echo "Compiling"
gcc vm.c -o vm
echo "Running vm"
./vm -n 128 -p lru >out.txt
echo "Comparing with correct.txt"
diff out.txt correct.txt
```

```
sh 制表符宽度: 8 行 7, 列 25 插入
```

图 5(a)

```
sherry@sherry: ~/桌面/Virtual-Memory-Manager-master
> TLB Hits = 55
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$ bash test.sh
Compiling
Running vm
Comparing with correct.txt
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$ bash test.sh
Compiling
Running vm
Comparing with correct.txt
1001,1002c1001,1002
< Page Faults = 538
< TLB Hits = 54
---
> Page Faults = 244
> TLB Hits = 55
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$ bash test.sh
Compiling
Running vm
Comparing with correct.txt
1001c1001
< Page Faults = 539
---
> Page Faults = 244
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$
```

图 5(b) (最下面的命令行操作)

test.sh 让每种情况下的输出均与 correct.txt 文件中的内容对比，而 correct.txt 中的 Page Faults = 244、TLB Hits = 55。所以通过上面所有的运行结果可以获得表 1。

Framenumber	策略	Page Faults	TLB Hits	Page-fault rate	TLB hit rate
256	FIFO	244	54	24.40%	5.40%
	LRU	244	55	24.40%	5.50%
128	FIFO	538	54	53.80%	5.40%
	LRU	539	55	53.90%	5.50%

表 1

Page-fault rate 和 TLB hit rate 的运行截图分别如图 6 所示：

```

sherry@sherry: ~/桌面/Virtual-Memory-Manager-master
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$ ./vm -n 256 -p fifo
Page-fault rate: 24.40%
TLB hit rate: 5.40%
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$ ./vm -n 256 -p lru
Page-fault rate: 24.40%
TLB hit rate: 5.50%
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$ ./vm -n 128 -p fifo
Page-fault rate: 53.80%
TLB hit rate: 5.40%
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$ ./vm -n 128 -p lru
Page-fault rate: 53.90%
TLB hit rate: 5.50%
sherry@sherry:~/桌面/Virtual-Memory-Manager-master$

```

图 6

## 二、Linux 内存管理实验

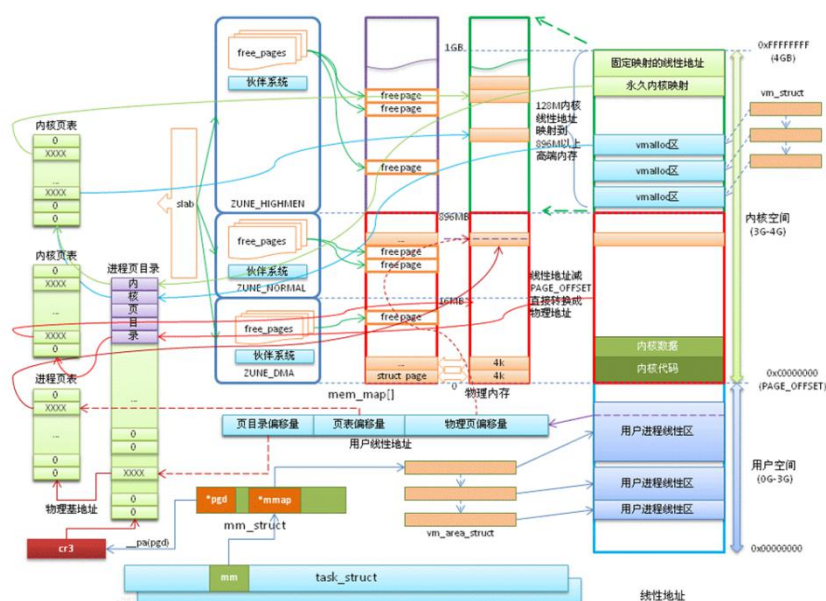


图 7

1、答：首先图中有一个明显错误，将 ZONE 打成 ZUNE。

从图中的右侧开始往左走，最右侧的框表明 os 分配给每个进程一个独立的、连续的、虚拟的地址内存空间，该大小一般是 4G，通常 32 位 Linux 内核虚拟地址空间划分 0~3G 为用户空间，3~4G 为内核空间（内核可以使用的线性地址只有 1G）。进程在用户态只能访问 0~3G，只有进入内核态才能访问 3G~4G，进程通过系统调用进入内核态，每个进程虚拟空间的 3G~4G 部分是相同的。高端内存 HIGH\_MEM 地址空间范围为 0xF8000000 ~ 0xFFFFFFFF（896MB~1024MB）。当内核想访问高于 896MB 物理地址内存时，从 0xF8000000 ~ 0xFFFFFFFF 地址空间范围内找一段相应大小空闲的逻辑地址空间，借用一会。借用这段逻辑地址空间，建立映射到想访问的那段物理内存（即填充内核 PTE 页面表）。内核将高端内存划分为 3 部分：VMALLOC\_START~VMALLOC\_END、KMAP\_BASE~FIXADDR\_START 和 FIXADDR\_START~4G。对于高端内存，可以通过 alloc\_page() 或者其它函数获得对应的 page，但是要想访问实际物理内存，还得把 page 转为线性地址才行，也就是说，我们需要为高端内存对应的 page 找一个线性空间，这个过程称为高端内存映射。对应高端内存的 3 部分，高端内存映射有三种方式：映射到“内核动态映射空间”（noncontiguous memory allocation）

这种方式很简单，因为通过 vmalloc()，在“内核动态映射空间”申请内存的时候，就可能从高端内存获得页面（参看 vmalloc 的实现），因此说高端内存有可能映射到“内核动态映射空间”中。持久内核映射（permanent kernel mapping）内核专门为此留出一块线性空间，从 PKMAP\_BASE 到 FIXADDR\_START，用于映射高端内存。在 2.6 内核上，这个地址范围是 4G-8M 到 4G-4M 之间。这个空间起叫“内核永久映射空间”或者“永久内核映射空间”。这个空间和其它空间使用同样的页目录表，对于内核来说，就是 swapper\_pg\_dir，对普通进程来说，通过 CR3 寄存器指向。通常情况下，这个空间是 4M 大小，因此仅仅需要一个页表即可，内核通过来 pkmap\_page\_table 寻找这个页表。通过 kmap()，可以把一个 page 映射到这个空间来。由于这个空间是 4M 大小，最多能同时映射 1024 个 page。因此，对于不使用的 page，及应该时从这个空间释放掉（也就是解除映射关系），通过 kunmap()，可以把一个 page 对应的线性地址从这个空间释放出来。临时映射（temporary kernel mapping）内核在 FIXADDR\_START 到 FIXADDR\_TOP 之间保留了一些线性空间用于特殊需求。这个空间称为“固定映射空间”在这个空间中，有一部分用于高端内存的临时映射。同时，内核空间包括内核数据和内核代码。

页(page)是内核的内存管理基本单位，其结构定义在 include/linux/mm\_types.h 文件中，部分代码如图 8 所示（完整定义太长）。其中 flags：页标志包含是不是脏的，是否被锁定等等，每一位单独表示一种状态，可同时表示出 32 种不同状态，定义在 linux/page-flags.h 中。virtual：页在虚拟内存中的地址，对于不能永久映射到内核空间的内存（比如高端内存），该值为 NULL；需要时必须动态映射这些内存。struct address\_space \*mapping：该页所在地址空间描述结构指针，用于内容为文件的页帧。pgoff\_t index：该页描述结构在地址空间 radix 树 page\_tree 中的对象索引号即页号 struct list\_head lru：最近最久未使用 struct slab 结构指针链表头变量。内核用 struct page 结构体表示每个物理页，struct page 结构体占 40 个字节。



```

69 struct page {
70     unsigned long flags;        /* Atomic flags, some possibly
71                                 * updated asynchronously */
72
73     union {
74         struct { /* Page cache and anonymous pages */
75             struct list_head lru;
76             /* See page-flags.h for PAGE_MAPPING_FLAGS */
77             struct address_space *mapping;
78             pgoff_t index;        /* Our offset within mapping. */
79             unsigned long private;
80         };
81         struct { /* page_pool used by netstack */
82             /**
83              * @dma_addr: might require a 64-bit value even on
84              * 32-bit architectures.
85              */
86             dma_addr_t dma_addr;
87         };
88         struct { /* slab, slob and slub */

```

图 8 (a)

```

167     unsigned int active;        /* SLAB */
168     int units;                  /* SLOB */
169 };
170
171 /* Usage count. *DO NOT USE DIRECTLY*. See page_ref.h */
172 atomic_t _refcount;
173
174 #ifdef CONFIG_MEMCG
175     struct mem_cgroup *mem_cgroup;
176 #endif
177
178 #if defined(WANT_PAGE_VIRTUAL)
179     void *virtual;              /* Kernel virtual address (NULL if
180                                 not kmapped, ie. highmem) */
181 #endif /* WANT_PAGE_VIRTUAL */
182
183 #ifdef LAST_CPUPID_NOT_IN_PAGE_FLAGS
184     int _last_cpupid;
185 #endif
186 } _struct_page_alignment;

```

图 8 (b)

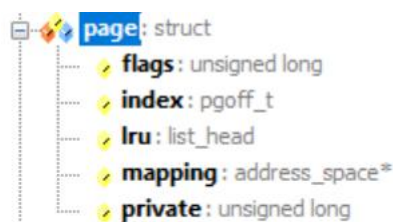


图 8 (c)

下面列举所有的页为单位进行连续物理内存分配，也称为低级页分配器，表 2 展示了页分配函数, 其中 `get_zeroed_page`: 对于用户空间，这个方法能保障系统敏感数据不会泄露 `page_address`: 把给定的页转换成逻辑地址。页释放函数如表 3。

页分配函数	描述
<code>alloc_pages(gfp_mask, order)</code>	分配 $2^{\text{order}}$ 个页，返回指向第一页的指针
<code>alloc_pages(gfp_mask)</code>	分配一页，返回指向页的指针
<code>__get_free_pages(gfp_mask, order)</code>	分配 $2^{\text{order}}$ 个页，返回指向其逻辑地址的指针
<code>__get_free_pages(gfp_mask)</code>	分配一页，返回指向其逻辑地址的指针
<code>get_zeroed_page(gfp_mask)</code>	分配一页，并填充内容为 0，返回指向其逻辑地址的指针

表 2

页释放函数	描述
<code>__free_pages(page, order)</code>	从 page 开始，释放 $2^{\text{order}}$ 个页
<code>free_pages(addr, order)</code>	从地址 addr 开始，释放 $2^{\text{order}}$ 个页
<code>free_page(addr)</code>	释放 addr 所在的那一页

表 3

物理内存只有进程真正去访问虚拟地址，发生缺页中断时，才分配实际的物理页面，建立物理内存和虚拟内存的映射关系。将虚拟地址转化为物理地址后，处理器才能解析地址访问请求。物理内存是通过分页机制实现的。理页在系统中由也结构 `struct page` 描述，所有的 page 都存储在数组 `mem_map[]` 中，可通过该数组找到系统中的每一页。虚拟进程空间：通过查询进程页表，获取实际物理内存地址；虚拟内核空间：通过查询内核页表，获取实际物理内存地址；物理内存映射区：物理内存映射区与实际物理去偏移量仅 `PAGE_OFFSET`，通过通过 `virt_to_phys()` 转化。虚拟内存与真实物理内存映射关系如图 9 所示。其中物理地址空间中除了 896M(`ZONE_DMA` + `ZONE_NORMAL`)的区域是绝对的物理连续，其他内存都不是物理内存连续。在虚拟内核地址空间中的安全保护区域的指针都是非法的，用于保证指针非法越界类的操作，`vm_struct` 是连续的虚拟内核空间，对应的物理页面可以不连续，地址范围  $(3G + 896M + 8M) \sim 4G$ ；另外在虚拟用户空间中 `vm_area_struct` 同样也是一块连续的虚拟进程空间，地址空间范围  $0 \sim 3G$ 。



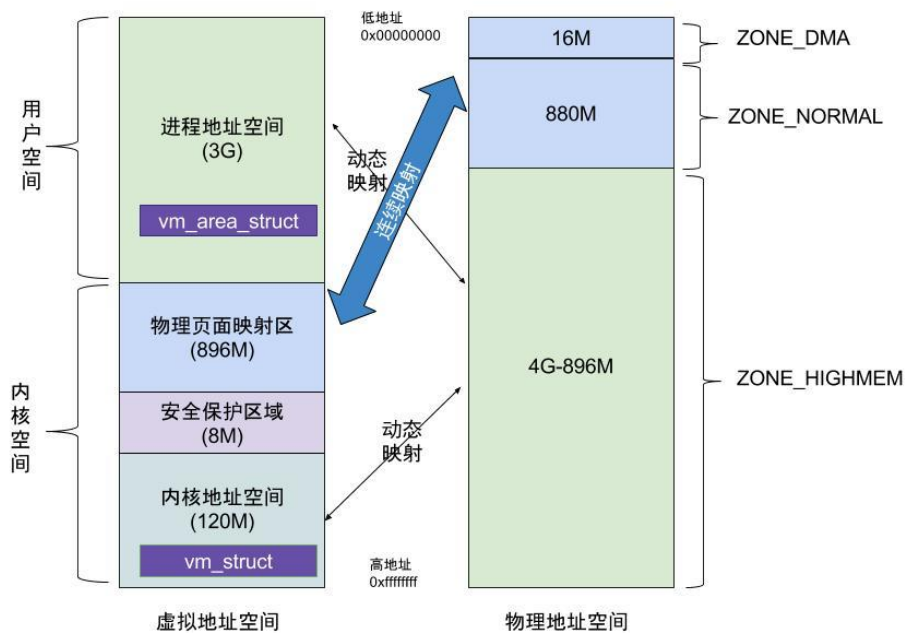


图 9

内核把页划分在不同的区(zone)，总共 3 个区，ZONE\_DMA 为 DMA 使用的页，物理内存小于 16MB；ZONE\_NORMAL 为可正常寻址的页，物理内存存在 16~896MB 之间；ZONE\_HIGHMEM 为动态映射的页，物理地址大于 896MB。执行 DMA 操作的内存必须从 ZONE\_DMA 区分配，一般内存，既可从 ZONE\_DMA，也可从 ZONE\_NORMAL 分配，但不能同时从两个区分配。

当在用户模式下运行进程请求额外内存时，从内核维护的空闲页帧列表上分配页面。这个列表通常使用页面置换算算法来填充，如前所述，它很可能包含散布在物理内存中的空闲页面。也要记住，如果用户进程请求单个字节内存，那么就会导致内部碎片，因为进程会得到整个帧。用于分配内核内存的空闲内存池通常不同于用于普通用户模式进程的列表。这有两个主要原因：内核需要为不同大小的数据结构请求内存，其中有的小于一页。因此，内核应保守地使用内存，并努力最小化碎片浪费。这一点非常重要，因为许多操作系统的内核代码或数据不受调页系统的控制。用户模式进程分配的页面不必位于连续物理内存。然而，有的硬件设备与物理内存直接交互，即无法享有虚拟内存接口带来的便利，因而可能要求内存常驻在连续物理内存中。管理用于内核进程的空闲内存的两个策略分别是伙伴系统和 slab 分配。伙伴系统从物理连续的大小固定的段上进行分配。从这个段上分配内存，采用 2 的幂分配器来满足请求分配单元的大小为 2 的幂（4KB、8KB、16KB 等）。请求单元的大小如不适当，就圆整到下一个更大的 2 的幂。例如，如果请求大小为 11KB，则按 16KB 的段来请求。伙伴系统的一个优点是：通过称为合并的技术，可以将相邻伙伴快速组合以形成更大分段。伙伴系统的明显缺点是：由于圆整到下一个 2 的幂，很可能造成分配段内的碎片。例如，33KB 的内存请求只能使用 64KB 段来满足。事实上，我们不能保证因内部碎片而浪费的单元一定少于 50%。分配内核内存的第

二种策略称为 slab 分配。每个 slab 由一个或多个物理连续的页面组成，每个 cache 由一个或多个 slab 组成，每个内核数据结构都有一个 cache。例如，用于表示进程描述符、文件对象、信号量等的数据结构都有各自单独的 cache。每个 cache 含有内核数据结构的对象实例（称为 object）。例如，信号量 cache 有信号量对象，进程描述符 cache 有进程描述符对象，等等。图 10 显示了 slab、cache 及 object 三者之间的关系。该图显示了 2 个大小为 3KB 的内核对象和 3 个大小为 7KB 的对象，它们位于各自的 cache 中。slab 分配算法采用 cache 来存储内核对象。在创建 cache 时，若干起初标记为 free 的对象被分配到 cache。cache 内的对象数量取决于相关 slab 的大小。例如，12KB slab（由 3 个连续的 4KB 页面组成）可以存储 6 个 2KB 对象。最初，cache 内的所有对象都标记为空闲。当需要内核数据结构的新对象时，分配器可以从 cache 上分配任何空闲对象以便满足请求。从 cache 上分配的对象标记为 used（使用）。内核为表示进程描述符的对象从 slab 分配器请求内存，在 Linux 系统中，进程描述符属于 struct task\_struct 类型，当 Linux 内核创建一个新任务时，它从 cache 中请求 struct task\_struct 对象的必要内存。cache 利用已经在 slab 中分配的并且标记为 free（空闲）的 struct task\_struct 对象来满足请求。在 Linux 中，slab 可以处于三种可能状态之一：满的：slab 的所有对象标记为使用；空的：slab 上的所有对象标记为空闲；部分：slab 上的对象有的标记为使用，有的标记为空闲。slab 分配器首先尝试在部分为空的 slab 中用空闲对象来满足请求。如果不存在，则从空的 slab 中分配空闲对象。如果没有空的 slab 可用，则从连续物理页面分配新的 slab，并将其分配给 cache；从这个 slab 上，再分配对象内存。slab 分配器提供两个主要优点：1、没有因碎片而引起内存浪费。碎片不是问题，因为每个内核数据结构都有关联的 cache，每个 cache 都由一个或多个 slab 组成，而 slab 按所表示对象的大小来分块。因此，当内核请求对象内存时，slab 分配器可以返回刚好表示对象的所需内存。2、可以快速满足内存请求。因此，当对象频繁地被分配和释放时，如来自内核请求的情况，slab 分配方案在管理内存时特别有效。分配和释放内存的动作可能是一个耗时过程。然而，由于对象已预先创建，因此可以从 cache 中快速分配。再者，当内核用完对象并释放它时，它被标记为空闲并返回到 cache，从而立即可用于后续的内核请求。

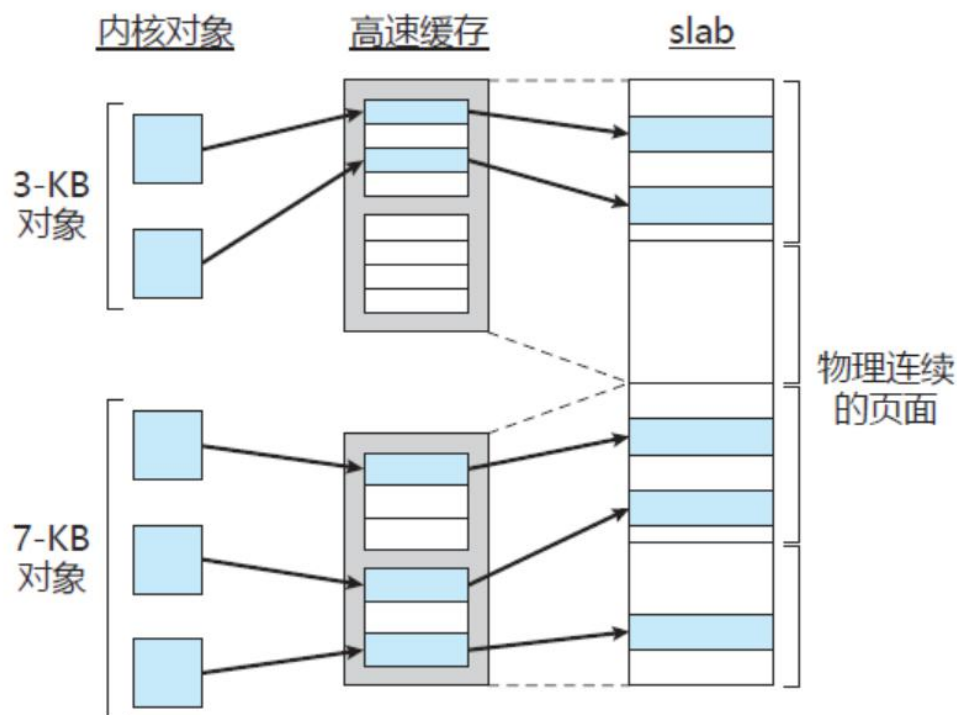


图 10

连续的物理页，使用低级页分配器 或 `kmalloc()`；高端内存分配，使用 `alloc_pages()`，返回 `page` 结构指针；想获取地址指针，应使用 `kmap()`，把高端内存映射到内核的逻辑地址空间；仅仅需要虚拟地址连续页，使用 `vmalloc()`，性能有所损失；频繁创建和撤销大量数据结构，考虑建立 slab 高速缓存。

进程由可执行的指令代码，数据和堆栈区组成。Linux 操作系统在内核中使用 `task_struct` 结构体来记录一个进程的各种信息，这个结构体的实例就是进程描述符，主要包括进程当前运行的状态信息、内核栈信息、进程使用状态、PID、优先级、锁、时间片、队列、信号量、内存管理信息、文件列表等等与进程管理、调度密切相关的信息。linux-5.3.8\include\linux 目录下的 `sched.h` 文件中定义了 `task_struct` 结构体，部分代码如下图：

```

637 struct task_struct {
638     #ifdef CONFIG_THREAD_INFO_IN_TASK
639         /*
640          * For reasons of header soup (see current_thread_info()), this
641          * must be the first element of task_struct.
642          */
643         struct thread_info    thread_info;
644     #endif
645     /* -1 unrunnable, 0 runnable, >0 stopped: */
646     volatile long            state;
647
648     /*
649      * This begins the randomizable portion of task_struct. Only
650      * scheduling-critical items should be added above here.
651      */
652     randomized_struct_fields_start
653
654     void                    *stack;
655     refcount_t              usage;

```

图 11 (a)

上述定义的 state 作为进程的状态标志，为-1 时表示当前进程不可运行，为 0 是可运行，大于 0 时表示进程已停止。

```

740 struct mm_struct    *mm;
741 struct mm_struct    *active_mm;

```

图 11 (b)

mm: 进程所拥有的用户空间内存描述符，active\_mm: 进程运行时所使用的内存描述符。对于普通进程，这两个指针变量相同，对于内核线程，不拥有任何内存描述符，mm 成员总是设为 NULL。当内核线程运行时，它的 active\_mm 成员被初始化为前一个运行进程的 active\_mm 值。每一个进程都会有自己独立的 mm\_struct，这样每一个进程都会有自己独立的地址空间，这样才能互不干扰。当进程之间的地址空间被共享的时候，我们可以理解为这个时候是多个进程使用一份地址空间，这就是线程。

一个进程的虚拟地址空间主要由两个数据结构来描述。一个是最高层次的：mm\_struct，一个是较高层次的：vm\_area\_structs。最高层次的 mm\_struct 结构描述了一个进程的整个虚拟地址空间。较高层次的结构 vm\_area\_struct 描述了虚拟地址空间的一个区间（简称虚拟区或线性区）。每个进程只有一个 mm\_struct 结构，在每个进程的 task\_struct 结构中，有一个指向该进程的结构。可以说，mm\_struct 结构是对整个用户空间的描述。对于进程可访问的地址空间称为内存区域。每个内存区域都具有对相关进程的可读、可写、可执行属性等相关权限设置。内存区域可包含的对象：代码段(text section)：可执行文件代码；数据段(data section)：可执行文件的已初始化全局变量（静态分配的变量和全局变量）；bss 段：程序中未初始化的全局变量，零页映射（页面的信息全部为 0 值）；进程用户空间栈的零页映射（进程的内核栈独立存在并由内核维护）；每一个诸如 C 库或动态连接程序等共享库的代码段、数据段和 bss 也会被载入进程的地址空间；任何内存映射文件；任何共享内存段；任何匿名的内存映射（比如由 malloc() 分配的内存）；这些内存区域不能相互覆盖，每一个进程都有不同的内存片段。

由 task\_struct 到 mm\_struct，进程的地址空间的分布，如图 12 所示。

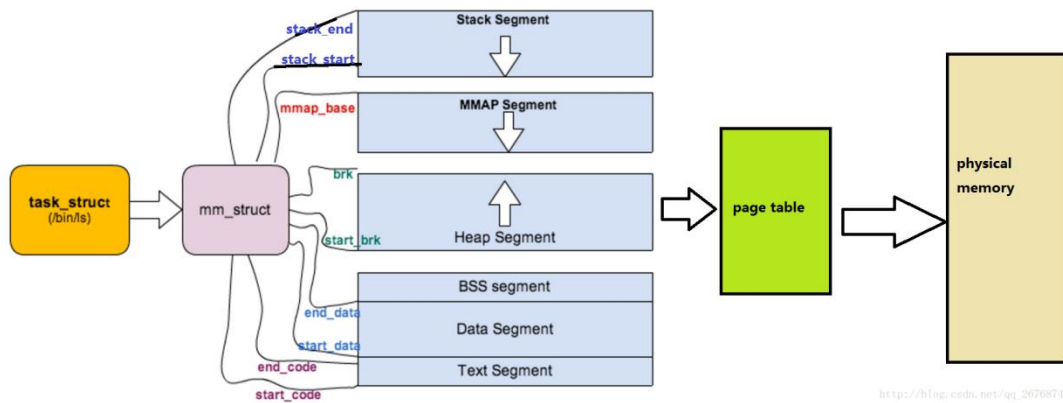


图 12

内存描述符由 `mm_struct` 结构体表示，定义在 `include/linux/mm_types.h` 文件中，其部分代码如图 13 所示。

```

365 struct mm_struct {
366     struct {
367         struct vm_area_struct *mmap;          /* List of VMAs */
368         struct rb_root mm_rb;
369         u64 vmacache_seqnum;                  /* per-thread vmacache */
370 #ifdef CONFIG_MMU
371         unsigned long (*get_unmapped_area) (struct file *filp,
372         unsigned long addr, unsigned long len,
373         unsigned long pgoff, unsigned long flags);
374 #endif
375         unsigned long mmap_base;              /* base of mmap area */
376         unsigned long mmap_legacy_base;       /* base of mmap area in bottom-up allocations */
377 #ifdef CONFIG_HAVE_ARCH_COMPAT_MMAP_BASES
378         /* Base addresses for compatible mmap() */
379         unsigned long mmap_compat_base;
380         unsigned long mmap_compat_legacy_base;
381 #endif
382         unsigned long task_size;              /* size of task vm space */

```

图 13 (a)

`mmap` 和 `mm_rb`：描述的对象都是相同的，`mmap` 以链表形式存放，利于高效地遍历所有元素，`mm_rb` 以红黑树形式存放，适合搜索指定元素。371 行开始的函数用来在进程地址空间中搜索有效的进程地址空间的函数。

```

395     atomic_t mm_users;
396
397     /**
398      * @mm_count: The number of references to &struct mm_struct
399      * (@mm_users count as 1).
400      *
401      * Use mmgrab()/mmdrop() to modify. When this drops to 0, the
402      * &struct mm_struct is freed.
403      */
404     atomic_t mm_count;

```

图 13 (b)

mm\_users: 代表正在使用该地址的进程数目, 当该值为 0 时 mm\_count 也变为 0; mm\_count: 代表 mm\_struct 的主引用计数, 当该值为 0 说明没有任何指向该 mm\_struct 结构体的引用, 结构体会被撤销。

```
416 | struct list_head mmlist; /* List of maybe swapped mm's. These
417 |                          * are globally strung together off
418 |                          * init_mm.mmlist, and are protected
419 |                          * by mmlist_lock
420 |                          */
```

图 13 (c)

mmlist: 所有的 mm\_struct 结构体都通过 mmlist 连接在一个双向链表中, 该链表的首元素是 init\_mm 内存描述符, 它代表 init 进程的地址空间。

在进程的进程描述符 task\_struct 结构体中, mm 域记录该进程使用的内存描述符。故 current->mm 代表当前进程的内存描述符。fork() 函数利用 copy\_mm 函数复制父进程的内存描述符, 子进程中的 mm\_struct 结构体通过 alloc\_mm() 从高速缓存中分配得到。通常, 每个进程都有唯一的 mm\_struct 结构体, 即唯一的进程地址空间。当子进程与父进程是共享地址空间, 可调用 clone(), 那么不再调用 alloc\_mm(), 而是仅仅是将 mm 域指向父进程的 mm, 即 tsk->mm = current->mm。相反地, 撤销内存是 exit\_mm() 函数, 该函数会进行常规的撤销工作, 更新一些统计量。

Linux 通过类型为 vm\_area\_struct 的结构体对象实现线性区, 该结构定义了内存 VMM 内存区域。每个 VM 区域/任务中有一个。VM 区域是进程虚拟内存空间的任何部分, 它具有页面错误处理程序的特殊规则 (即共享库, 可执行区域等)。我们可以把 vm\_area\_struct 称为线性区描述符, 它标识了一个线性地址区间。进程所拥有的线性区从来不重叠, 并且内核尽力把新分配的线性区与紧邻的现有线性区进行合并。如果两个相邻区的访问权限相匹配, 就能把他们合并在一起。进程所有的线性区是通过一个简单的链表链接在一起的, 出现在链表中的线性区是按内存地址的升序排列的。每两个线性区可以由未用的内存地址区隔开。内核通过进程的内存描述符的 mmap 字段来查找线性区, 其中 mmap 字段指向链表中的第一个线性区描述符。同时内存描述符的 map\_count 字段存放进程所拥有的线性区数目。对 vm\_area\_struct 的定义在 /include/linux/mm\_types.h 中, 部分代码如图 14 所示。



```

287 | struct vm_area_struct {
288 |     /* The first cache line has the info for VMA tree walking. */
289 |
290 |     unsigned long vm_start;    /* Our start address within vm_mm. */
291 |     unsigned long vm_end;      /* The first byte after our end address
292 |                                | within vm_mm. */
293 |
294 |     /* linked list of VM areas per task, sorted by address */
295 |     struct vm_area_struct *vm_next, *vm_prev;
296 |
297 |     struct rb_node vm_rb;

```

图 14 (a)

其中 `vm_start` 是区域的首地址, `vm_end` 是区域的尾地址。295 行定义为 VMA 链表, `vm_rb` 为 VMA 的红黑树结构。

```

309 | struct mm_struct *vm_mm;    /* The address space we belong to. */
310 | pgprot_t vm_page_prot;     /* Access permissions of this VMA. */
311 | unsigned long vm_flags;     /* Flags, see mm.h. */

```

图 14 (b)

其中 `vm_mm` 为内存描述符, `vm_page_prot` 为访问控制权限, `vm_flags` 为保护标志位和属性标志位。

```

332 | /* Function pointers to deal with this struct. */
333 | const struct vm_operations_struct *vm_ops;
334 |
335 | /* Information about our backing store: */
336 | unsigned long vm_pgoff;     /* Offset (within vm_file) in PAGE_SIZE
337 |                                | units */
338 | struct file * vm_file;      /* File we map to (can be NULL). */
339 | void * vm_private_data;     /* was vm_pte (shared mem) */

```

图 14 (c)

其中 `vm_ops` 为相关的操作表, `vm_file` 为指向被映射的文件的指针, `vm_private_data` 为设备驱动私有数据, 与内存管理无关。对 `vm_operations_struct` 的定义如下图所示, `vm_operations` 结构中包含的是函数指针; 其中, `open`、`close` 分别用于虚拟区间的打开、关闭, 而 `nopage` 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, 当 Linux 处理这一缺页异常时(请页机制), 就可以为新的虚拟内存区分配实际的物理内存。

```

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 write_access);
    ...
}

```

图 15

应用程序操作的对象时映射到物理内存之上的虚拟内存,而处理器直接操作的是物理内存。故应用程序访问一个虚拟地址时,需要将虚拟地址转换为物理地址,然后处理器才能解析地址访问请求,这个转换工作通过查询页表完成。Linux 使用三级页表完成地址转换。1、顶级页表:页全局目录(PGD),指向二级页目录;2、二级页表:中间页目录(PMD),指向 PTE 中的表项;3、最后一级:页表(PTE),指向物理页面。多数体系结构,搜索页表工作由硬件完成。每个进程都有自己的页表(线程会共享页表)。为了加快搜索,实现了翻译后缓冲器(TLB),作为将虚拟地址映射到物理地址的硬件缓存。还有写时拷贝方式共享页表,当 fork() 时,父子进程共享页表,只有当子进程或父进程试图修改特定页表项时,内核才创建该页表项的新拷贝,之后父子进程不再共享该页表项。可见,利用共享页表可以消除 fork() 操作中页表拷贝所带来的消耗。

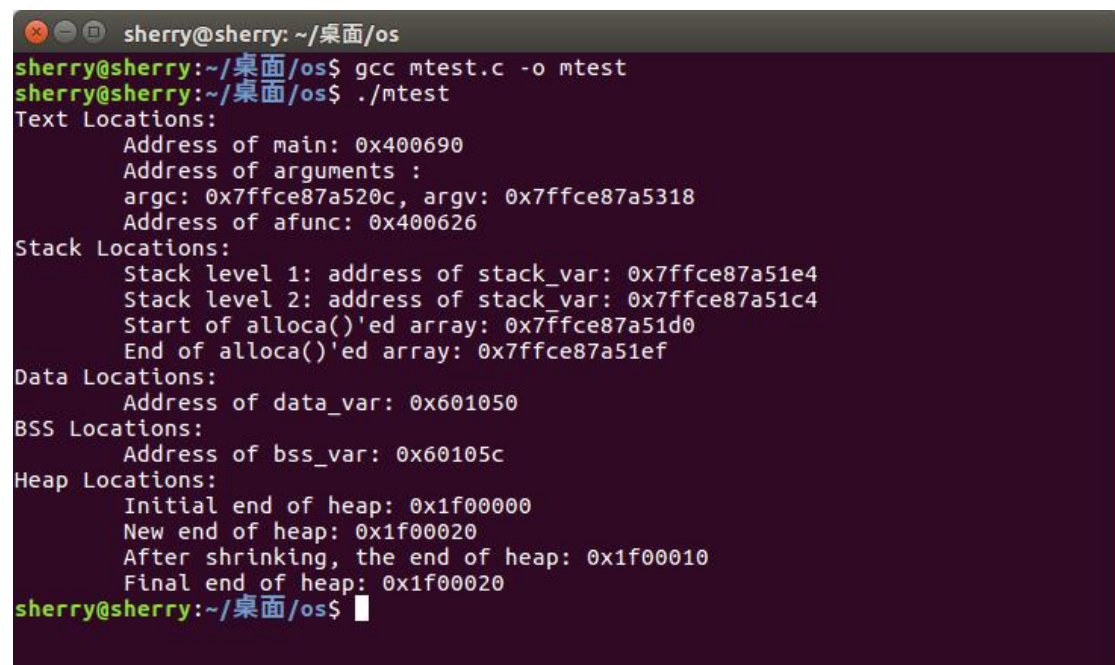
每个进程都有其自身的页面目录 PGD, Linux 将该目录的指针存放在与进程对应的内存结构 task\_struct. (struct mm\_struct)mm->pgd 中。每当一个进程被调度 (schedule()) 即将进入运行态时, Linux 内核都要用该进程的 PGD 指针设置 CR3 (switch\_mm())。当创建一个新的进程时,都要为新进程创建一个新的页面目录 PGD, 并从内核的页面目录 swapper\_pg\_dir 中复制内核区间页面目录项至新建进程页面目录 PGD 的相应位置, 具体过程如下: do\_fork() --> copy\_mm() --> mm\_init() --> pgd\_alloc() --> set\_pgd\_fast() --> get\_pgd\_slow() --> memcpy(&PGD + USER\_PTRS\_PER\_PGD, swapper\_pg\_dir + USER\_PTRS\_PER\_PGD, (PTRS\_PER\_PGD - USER\_PTRS\_PER\_PGD) \* sizeof(pgd\_t))。这样一来, 每个进程的页面目录就分成了两部分, 第一部分为“用户空间”, 用来映射其整个进程空间 (0x0000 0000—0xBFFF FFFF) 即 3G 字节的虚拟地址; 第二部分为“系统空间”, 用来映射 (0xC000 0000—0xFFFF FFFF) 1G 字节的虚拟地址。可以看出 Linux 系统中每个进程的页面目录的第二部分是相同的, 所以从进程的角度来看, 每个进程有 4G 字节的虚拟空间, 较低的 3G 字节是自己的用户空间, 最高的 1G 字节则为与所有进程以及内核共享的系统空间。

当系统启动时, Linux 内核映像被安装在物理地址 0x00100000 开始的地方, 即 1MB 开始的区间(第 1M 留作它用)。然而, 在正常运行时, 整个内核映像应该在虚拟内核空间中, 因此, 连接程序在连接内核映像时, 在所有的符号地址上加一个偏移量 PAGE\_OFFSET, 这样, 内核映像在内核空间的起始地址就为 0xC0100000。例如, 进程的页目录 PGD (属于内核数据结构) 就处于内核空间中。在进程切换时, 要将寄存器 CR3 设置成指向新进程的页目录

PGD，而该目录的起始地址在内核空间中是虚地址，但 CR3 所需要的是物理地址，这时候就要用 `__pa()` 进行地址转换。

2、答： `kmalloc` 和 `get_free_page` 申请的内存位于物理内存映射区域，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，因此存在较简单的转换关系，`virt_to_phys()` 可以实现内核虚拟地址转化为物理地址。而 `vmalloc` 申请的内存则位于 `vmalloc_start~vmalloc_end` 之间，与物理地址没有简单的转换关系，虽然在逻辑上它们也是连续的，但是在物理上它们不要求连续。`__get_free_page` 和 `kmalloc` 分配的是连续的物理内存，与物理内存存在直接映射关系。`brk` 和 `do_map` 申请的是进程虚拟内存，使用时对进程页表进行访问，缺页则使用 `get_free_page` 从物理内存中得到空闲页进行填充，最后建立起映射关系（虚拟地址到物理地址）。

3、答：在 User-Level Memory Management 中的原代码的基础上进行了如下改动：1）补充注释，有助于理解代码含义，并且将 `afunc` 函数定义移动到 `main` 函数前面。2）在打印代码段部分，补充打印了 `main` 函数参数的地址。3）在原代码中，最开始使用了一次 `sbrk` 函数扩大地址内存空间，之后进行了一次缩小地址内存空间的操作，分别打印出了最初的地址空间结束的位置地址、扩大后的地址空间结束的位置地址、缩小后的地址空间结束的位置地址。在此基础上，我在最后又加上了一次扩大地址内存空间的操作，让对地址内存空间增减的操作分别在上一次的地址内存空间的基础上分别为 +32、-16、+16，用于对比看出地址的变化情况。打印出的各部分地址如图 16 所示。从图中可以看出参数地址、`alloca` 数组地址在栈上；通过两个栈变量的地址对比可以看出栈向下增长；以及 BSS 段变量在 Data 段变量之上；Heap 紧接着在 BSS 段之上并且 Heap 向上增长；地址空间可以收缩，在经过添加的操作后的地址变回了第一次扩大地址内存空间后的地址。



```
sherry@sherry: ~/桌面/os
sherry@sherry:~/桌面/os$ gcc mtest.c -o mtest
sherry@sherry:~/桌面/os$ ./mtest
Text Locations:
    Address of main: 0x400690
    Address of arguments :
    argc: 0x7ffce87a520c, argv: 0x7ffce87a5318
    Address of afunc: 0x400626
Stack Locations:
    Stack level 1: address of stack_var: 0x7ffce87a51e4
    Stack level 2: address of stack_var: 0x7ffce87a51c4
    Start of alloca()'ed array: 0x7ffce87a51d0
    End of alloca()'ed array: 0x7ffce87a51ef
Data Locations:
    Address of data_var: 0x601050
BSS Locations:
    Address of bss_var: 0x60105c
Heap Locations:
    Initial end of heap: 0x1f00000
    New end of heap: 0x1f00020
    After shrinking, the end of heap: 0x1f00010
    Final end of heap: 0x1f00020
sherry@sherry:~/桌面/os$
```

图 16

4、答：在 3 中的 `mtest.c` 代码的最后一行加上 `sleep(10000)` 让 `mtest` 程序驻留内存，在对 `mtest.c` 进行编译后，在终端输入 `./mtest &` 将 `mtest` 在后台运行，输入命令后会先输出一行，可以从中知道 `pid`。然后打开另一个终端，输入 `cat /proc/pid_number/maps` 以输出需

要的信息，pid\_number 为前面获得的 pid。运行界面截图如图 17，输出的信息截图如图 18 所示。每一列的含义如表 4，对输出的每一行（共 18 行）分析如表 5 所示。

```
sherry@sherry: ~/桌面/os
sherry@sherry:~/桌面/os$ gcc mtest.c -o mtest
sherry@sherry:~/桌面/os$ ./mtest &
[1] 2959
sherry@sherry:~/桌面/os$ Text Locations:
    Address of main: 0x4006d0
    Address of arguments :
        argc: 0x7ffddb112dac, argv: 0x7ffddb112eb8
    Address of afunc: 0x400666
Stack Locations:
    Stack level 1: address of stack_var: 0x7ffddb112d84
    Stack level 2: address of stack_var: 0x7ffddb112d64
    Start of alloca()'ed array: 0x7ffddb112d70
    End of alloca()'ed array: 0x7ffddb112d8f
Data Locations:
    Address of data_var: 0x601058
BSS Locations:
    Address of bss_var: 0x601064
Heap Locations:
    Initial end of heap: 0x6a4000
    New end of heap: 0x6a4020
    After shrinking, the end of heap: 0x6a4010
    Final end of heap: 0x6a4020
```

图 17

```
sherry@sherry: ~/桌面/os
sherry@sherry:~/桌面/os$ cat /proc/2959/maps
00400000-00401000 r-xp 00000000 103:09 131812 /home/sherry/桌面/os/mtest
00600000-00601000 r--p 00000000 103:09 131812 /home/sherry/桌面/os/mtest
00601000-00602000 rw-p 00001000 103:09 131812 /home/sherry/桌面/os/mtest
00683000-006a5000 rw-p 00000000 00:00 0 [heap]
7feac15a1000-7feac1761000 r-xp 00000000 103:08 137063 /lib/x86_64-linux-gnu/libc-2.23.so
7feac1761000-7feac1961000 ---p 001c0000 103:08 137063 /lib/x86_64-linux-gnu/libc-2.23.so
7feac1961000-7feac1965000 r--p 001c0000 103:08 137063 /lib/x86_64-linux-gnu/libc-2.23.so
7feac1965000-7feac1967000 rw-p 001c4000 103:08 137063 /lib/x86_64-linux-gnu/libc-2.23.so
7feac1967000-7feac196b000 rw-p 00000000 00:00 0
7feac196b000-7feac1991000 r-xp 00000000 103:08 137035 /lib/x86_64-linux-gnu/ld-2.23.so
7feac1b4f000-7feac1b52000 rw-p 00000000 00:00 0
7feac1b90000-7feac1b91000 r--p 00025000 103:08 137035 /lib/x86_64-linux-gnu/ld-2.23.so
7feac1b91000-7feac1b92000 rw-p 00026000 103:08 137035 /lib/x86_64-linux-gnu/ld-2.23.so
7feac1b92000-7feac1b93000 rw-p 00000000 00:00 0
7ffddb0f4000-7ffddb115000 rw-p 00000000 00:00 0
7ffddb18a000-7ffddb18d000 r--p 00000000 00:00 0 [stack]
7ffddb18d000-7ffddb18f000 r-xp 00000000 00:00 0 [vvar]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vdso]
[vsyscall]
```

图 18

列数	含义
1	本段内存映射的虚拟地址空间范围,对应 vm_area_struct 中的 vm_start 和 vm_end。
2	权限 r-读, w-写 x-可执行 p-私有, 对应 vm_flags。
3	针对有名映射, 指本段映射地址在文件中的偏移, 对应 vm_pgoff。对匿名映射而言, 为 vm_area_struct->vm_start。
4	所映射的文件所属设备的设备号, 对应 vm_file->f_dentry->d_inode->i_sb->s_dev。: 前为主设备号, 后为次设备号。



5	-文件的索引节点号，对应 <code>vm_file-&gt;f_dentry-&gt;d_inode-&gt;i_ino</code> ，与 <code>ls -li</code> 显示的内容相符。匿名映射为 0。
6	所映射的文件名。对有名映射而言，是映射的文件名，对匿名映射来说，是此段内存存在进程中的作用。 <code>[stack]</code> 表示本段内存作为栈来使用， <code>[heap]</code> 作为堆来使用，其他情况则为无。

表 4

第 1 行		
00400000-00401000 代表本段内存映射的虚拟地址空间范围。	<code>r-xp</code> 表示权限为只读、可执行、私有，该段为代码段。	00000000 指本段映射地址在文件中的偏移。
103: 09 表示主设备号为 103，次设备号为 09。	131812 为文件的索引节点号，这里非 0 即有名映射。	<code>/home/sherry/桌面/os/mtest</code> 映射的文件名。
第 2 行		
00600000-00601000 代表本段内存映射的虚拟地址空间范围。	<code>r--p</code> 表示权限为只读、私有，该段为数据段。	00000000 指本段映射地址在文件中的偏移。
103: 09 表示主设备号为 103，次设备号为 09。	131812 为文件的索引节点号，这里非 0 即有名映射。	<code>/home/sherry/桌面/os/mtest</code> 映射的文件名。
第 3 行		
00601000-00602000 代表本段内存映射的虚拟地址空间范围。	<code>rw-p</code> 表示权限为可读写、私有，该段为数据段。	00001000 指本段映射地址在文件中的偏移。
103: 09 表示主设备号为 103，次设备号为 09。	131812 为文件的索引节点号，这里非 0 即有名映射。	<code>/home/sherry/桌面/os/mtest</code> 映射的文件名。
第 4 行		
00683000-006a5000 代表本段内存映射的虚拟地址空间范围。	<code>rw-p</code> 表示权限为可读写、私有。	00000000 为 <code>vm_area_struct-&gt;vm_start</code> 。
00: 00 表示为匿名映射。	0 即匿名映射。	<code>[heap]</code> 表示为堆段。
第 5 行		
7feac15a1000-7feac1761000 代表本段内存映射的虚拟地址空间范围。	<code>r-xp</code> 表示权限为只读、可执行、私有，该段为代码段。	00000000 指本段映射地址在文件中的偏移。
103: 08 表示主设备号为 103，次设备号为 08。	137063 为文件的索引节点号，这里非 0 即有名映射。	<code>/lib/x86_64-linux-gnu/libc-2.23.so</code> 是共享库名。
第 6 行		
7feac1761000-7feac1961000 代表本段内存映射的虚拟地址空	<code>---p</code> 表示权限为私有，为	001c0000 指本段映射地址

间范围。	空闲内存区。	在文件中的偏移。
103: 08 表示主设备号为 103, 次设备号为 08。	137063 为文件的索引节点号, 这里非 0 即有名映射。	/lib/x86_64-linux-gnu/libc-2.23.so 是共享库名。
第 7 行		
7feac1961000-7feac1965000 代表本段内存映射的虚拟地址空间范围。	r--p 表示权限为只读、私有, 该段为数据段。	001c0000 指本段映射地址在文件中的偏移。
103: 08 表示主设备号为 103, 次设备号为 08。	137063 为文件的索引节点号, 这里非 0 即有名映射。	/lib/x86_64-linux-gnu/libc-2.23.so 是共享库名。
第 8 行		
7feac1965000-7feac1967000 代表本段内存映射的虚拟地址空间范围。	rw-p 表示权限为可读写、私有, 该段为数据段。	001c4000 指本段映射地址在文件中的偏移。
103: 08 表示主设备号为 103, 次设备号为 08。	137063 为文件的索引节点号, 这里非 0 即有名映射。	/lib/x86_64-linux-gnu/libc-2.23.so 是共享库名。
第 9 行		
7feac1967000-7feac196b000 代表本段内存映射的虚拟地址空间范围。	rw-p 表示权限为可读写、私有。	00000000 为 vm_area_struct->vm_start。
00: 00 表示为匿名映射。	0 即匿名映射。	无表述则为其他情况。
第 10 行		
7feac196b000-7feac1991000 代表本段内存映射的虚拟地址空间范围。	r-xp 表示权限为只读、可执行、私有, 该段为代码段。	00000000 指本段映射地址在文件中的偏移。
103: 08 表示主设备号为 103, 次设备号为 08。	137035 为文件的索引节点号, 这里非 0 即有名映射。	/lib/x86_64-linux-gnu/libd-2.23.so 是共享库名。
第 11 行		
7feac1b4f000-7feac1b52000 代表本段内存映射的虚拟地址空间范围。	rw-p 表示权限为可读写、私有。	00000000 为 vm_area_struct->vm_start。
00: 00 表示为匿名映射。	0 即匿名映射。	无表述则为其他情况。
第 12 行		
7feac1b90000-7feac1b91000 代表本段内存映射的虚拟地址空	r--p 表示权限为只读、私有, 该段为数据段。	00025000 指本段映射地址在文件中的偏移。



间范围。		
103: 08 表示主设备号为 103, 次设备号为 08。	137035 为文件的索引节点号, 这里非 0 即有名映射。	/lib/x86_64-linux-gnu/lib-2.23.so 是共享库名。
第 13 行		
7feac1b91000-7feac1b92000 代表本段内存映射的虚拟地址空间范围。	rw-p 表示权限为可读写、私有, 该段为数据段。	00026000 指本段映射地址在文件中的偏移。
103: 08 表示主设备号为 103, 次设备号为 08。	137035 为文件的索引节点号, 这里非 0 即有名映射。	/lib/x86_64-linux-gnu/lib-2.23.so 是共享库名。
第 14 行		
7feac1b92000-7feac1b93000 代表本段内存映射的虚拟地址空间范围。	rw-p 表示权限为可读写、私有。	00000000 为 vm_area_struct->vm_start。
00: 00 表示为匿名映射。	0 即匿名映射。	无表述则为其他情况。
第 15 行		
7ffddb0f4000-7ffddb115000 代表本段内存映射的虚拟地址空间范围。	rw-p 表示权限为可读写、私有。	00000000 为 vm_area_struct->vm_start。
00: 00 表示为匿名映射。	0 即匿名映射。	[stack]为栈段。
第 16 行		
7ffddb18a000-7ffddb18d000 代表本段内存映射的虚拟地址空间范围。	r--p 表示权限为只读、私有。	00000000 为 vm_area_struct->vm_start。
00: 00 表示为匿名映射。	0 即匿名映射。	[vvar]
第 17 行		
7ffddb18d000-7ffddb18f000 代表本段内存映射的虚拟地址空间范围。	r-xp 表示权限为只读、可执行、私有。	00000000 为 vm_area_struct->vm_start。
00: 00 表示为匿名映射。	0 即匿名映射。	[vdso]虚拟动态共享对象。
第 18 行		
fffffffffff600000-fffffffffff601000 代表本段内存映射的虚拟地址空间范围。	r--xp 表示权限为只读、可执行、私有。	00000000 为 vm_area_struct->vm_start。

00: 00 表示为匿名映射。	0 即匿名映射。	[vsyscall]内核区。
-----------------	----------	----------------

表 5

5、答：（1）1）brk/sbrk 和 mmap 两种方式的区别是 brk 是将数据段(.data)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，称为文件映射区域的地方）找一块空闲的虚拟内存。2）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）。3）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）。

（2）从静态存储区域分配，内存存在程序编译的时候就已经分配好，这块内存存在程序的整个运行期间都存在。例如全局变量，static 变量；在栈上创建，在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。从堆上分配，亦称动态内存分配，程序在运行的时候用 malloc 或 new 申请任意多少的内存，程序员自己负责在何时用 free 或 delete 释放内存，动态内存的生存期由我们决定，使用非常灵活，但问题也最多。系统调用不能支持任意大小的内存分配（有的系统调用只支持固定大小以及其倍数的内存申请，这样的话，对于小内存的分配会造成浪费；系统调用申请内存代价昂贵，涉及到用户态和核心态的转换。

（3）malloc 的内存分配，是分配的虚拟内存，只有在真正访问一个地址的时候才建立这个地址的物理映射。先检查要访问的虚拟地址是否合法，合法的话就查找/分配一个物理页，然后填充物理页内容（读取磁盘，或者直接置 0，或者无操作），最后建立映射关系（虚拟地址到物理地址）。malloc 在分配用户传入的大小的时候，还分配了一个相关的用于管理的额外内存。

## 6、（附加题）

代码中定义了 myalloc/myfree 两个函数，用其模拟 malloc/free 函数。首先对模拟堆的内存大小继续初始化，然后 mymalloc(20)，因为最接近 20 的 8 的倍数为 24，所以给其分配的大小为 24，即 Block1 的大小为 24，标志为已分配，剩下为 Block2 大小为 40，未分配部分。后面的 myalloc 操作将内存分为一块块的，即使进行 myfree，也只是使这一块进行释放且标志为未分配，块数及对应大小并未改变，再进行 myalloc 时是寻找当前未分配块中是否有大小大于需要分配大小的块，因此会存在未分配内存大小和大于需要大小时无法实现分配的情况，这样对于内存的利用效率是很低的。

因此在代码中有 coalesce 函数，用于将为分配的内存进行合并以提高内存的利用率，具体的效果可从代码的运行结果图中看出（图 19）。

```
sherry@sherry: ~/桌面/os
sherry@sherry:~/桌面/os$ ./myalloc
initial allocation
Block 0: size 64    unallocated
malloc 20
Block 0: size 24    allocated
Block 1: size 40    unallocated

malloc 10
Block 0: size 24    allocated
Block 1: size 16    allocated
Block 2: size 24    unallocated

free 20
Block 0: size 24    unallocated
Block 1: size 16    allocated
Block 2: size 24    unallocated
malloc 4
Block 0: size 8     allocated
Block 1: size 16    unallocated
Block 2: size 16    allocated
Block 3: size 24    unallocated

free 10
Block 0: size 8     allocated
Block 1: size 16    unallocated
Block 2: size 16    unallocated
Block 3: size 24    unallocated
malloc 30
allocation failed

coalesce
Block 0: size 8     allocated
Block 1: size 56    unallocated
malloc 30
Block 0: size 8     allocated
Block 1: size 40    allocated
Block 2: size 16    unallocated

free everything
Block 0: size 8     unallocated
Block 1: size 40    unallocated
Block 2: size 16    unallocated
malloc 56
allocation failed

coalesce
Block 0: size 64    unallocated
malloc 56
Block 0: size 64    allocated

sherry@sherry:~/桌面/os$
```

图 19