

操作系统大作业2

陈政培 17363011 智能科学与技术 智科1班

操作系统大作业2

1. 虚存管理模拟程序

地址转换测试——vm.c

原理和具体实现

getopt.h库运行前键入参数

程序运行结果

包含LRU和FIFO的TLB内存管理程序以及页面置换Page Replacement——vm-final.c

原理和具体实现

程序运行结果

运行，测试128情况下的FIFO

运行，测试128情况下的LRU

附加题——简单trace生成器程序

原理和具体实现

绘制图表

运行结果

addresses-locality.txt输出

生成的图表

在vm中测试FIFO和LRU的性能

FIFO

LRU

2. Linux内存管理实验程序

Linux物理内存的组织形式

存储节点——Node

管理区——Zone

页面——Page

基于三个基本单元的管理优化

① 2级页表解析

slab层

高端内存的划分

页表映射

② 内核层不同内存分配接口的区别

__get_free_pages, alloc_page与get_free_page

kmalloc, vmalloc

③ mtest.c 打印代码段、数据段、BSS，栈、堆等的相关地址

增加的内容

代码运行结果：

④ 分析mtest各个内存段

运行结果

分析列

分析行

第六列为什么会有相同的文件路径

⑤ 内存分配相关问题

⑥ 模仿malloc接口，实现简单的 myalloc/myfree

mempool.c部分细节

test.c测试代码和测试函数

运行结果：

init阶段

1. 虚存管理模拟程序

地址转换测试——vm.c

原理和具体实现

- (1) 根据用户输入的进程号找到相应的页表，在该页表中查询该页号的页面是否在主存 中，若已在主存中，则直接访问，修改贮存队列中的访问时间。
- (2) 若不在主存队列中，则需要将其装入至主存队列，若可以找到一块空闲帧，则将其 装入至主存队列中，修改在主存中的信息
- (3) 若未在主存队列中，找到空闲帧，则需在驻留集队列中替换一个页面，根据FIFO算法，替换队头结点，同时将待访问页面构造为一个新结点，插入至队尾，同时根据所删除结点的帧号在主存中队列中相应帧信息

getopt.h库运行前键入参数

运用库提供的功能从main函数传入的字符串中得到预设的参数

```
int opt;
FILE *ftp;
FILE *value;
char *usage = "USAGE: sim -f tracefile -b valuefile -n numberframe -p
policy-algorithm\n";

while ((opt = getopt(argc, argv, "f:b:n:p:")) != -1) {
    switch (opt) {
        case 'f':
            tracefile = optarg;
            break;
        case 'b':
            valuefile = optarg;
            break;
        case 'n':
            numberframe = (unsigned)strtoul(optarg, NULL, 10);
            break;
        case 'p':
            replacement_alg = optarg;
            break;
        default:
            fprintf(stderr, "%s", usage);
            exit(1);
    }
}
```

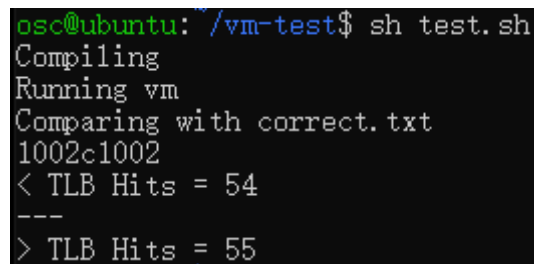
基本逻辑就是键入trace文件和内存信息文件，然后预设页框数目限制和使用的算法。vm.c程序中只设计了FIFO算法，所以最后一个-p policy选项暂时没有含义

```
osc@ubuntu:~/vm-test$ ./vm -f addresses.txt -b BACKING_STORE.bin -n 256 -p fifo
```

相应的test.sh文件也需要稍作改动

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

程序运行结果



```
osc@ubuntu:~/vm-test$ sh test.sh
Compiling
Running vm
Comparing with correct.txt
1002c1002
< TLB Hits = 54
---
> TLB Hits = 55
```

correct文件比对后，左后的TLB Hits和样例correct.txt不符合，但是经过和其他同学对比，应该正确答案就是54次命中

包含LRU和FIFO的TLB内存管理程序以及页面置换Page Replacement——vm-final.c

原理和具体实现

LRU部分的实现使用了双向链表数据结构

```
typedef struct Double_Link_List
{
    int table[2];
    struct Double_Link_List *next;
    struct Double_Link_List *front;
}DL_List;
```

LRU的实现是建立在vm.c的基础上，如果程序选择FIFO则运行老代码，如果选择LRU则申请双向链表

```
        else
        {
            frame_pos = Frame_List_pos-> table[1];
            if(policy_flag)
                Frame_head =
DL_List_Update(Frame_head,Frame_List_pos); //LRU的页面置换
        }
        TLB_head = TLB_head->next;
        TLB_head ->table[0] = page_pos;
        TLB_head -> table[1] = frame_pos;
    }
```

```

else
{
    frame_pos = page_table[page_pos];
    if(policy_flag)
    {
        Frame_List_pos = DL_List_find(page_pos, Frame_head);
        Frame_head = DL_List_Update(Frame_head, Frame_List_pos);
//LRU的页面置换
    }

    }
    TLB_head = TLB_head->next;
    TLB_head ->table[0] = page_pos;
    TLB_head -> table[1] = frame_pos;
}
else
{
    Hit_count += 1;
    frame_pos = TLB_pos->table[1];
    if(policy_flag)
        TLB_head = DL_List_Update(TLB_head, TLB_pos); //LRU的TLB
}
physical_address = frame_pos * 256 + page_offset;
printf("%d\n", mem[frame_pos][page_offset]);

```

程序运行结果

运行，测试128情况下的FIFO

```
osc@ubuntu:~/vm-test$ ./vm2 -f addresses.txt -b BACKING_STORE.bin -n 128 -p fifo
```

```

0
0
-85
0
0
126
-46
Page Faults = 538
TLB Hits = 40

```

Page Fault发生了538次

运行，测试128情况下的LRU

```
osc@ubuntu:~/vm-test$ ./vm2 -f addresses.txt -b BACKING_STORE.bin -n 128 -p lru
```

```

0
-85
0
0
126
-46
Page Faults = 537
TLB Hits = 41
osc@ubuntu:~/vm-test$

```

Page Fault发生了537次，快表命中次数也多一次，LRU相对会更优一些

附加题——简单trace生成器程序

原理和具体实现

借用vm-final.c文件的代码，直接申请10000个双向列表，以输出每一次申请的地址

```
int main(int argc, char **argv)
{
    ftp = fopen("addresses-locality.txt", "w");

    DL_List *TLB_head = DL_List_Init(TLB_SIZE);
    DL_List *Frame_head = DL_List_Init(numberframe);
    Frame_Init(Frame_head);
    Frame_head = Frame_head->front;
    DL_List *TLB_pos, *Frame_List_pos;

    fclose(ftp);
    return 0;
}

DL_List* DL_List_Init(int size)
{
    int i;
    DL_List *new_node, *cur_node, *head;
    new_node = (DL_List *)malloc(sizeof(DL_List));
    new_node->table[0] = -1;
    new_node->table[1] = -1;
    new_node->front = new_node;
    new_node->next = new_node;
    head = cur_node = new_node;
    for(i = 0; i < size - 1; i++)
    {
        new_node = (DL_List *)malloc(sizeof(DL_List));
        new_node->table[0] = -1;
        new_node->table[1] = -1;
        cur_node->next = new_node;
        new_node->front = cur_node;
        new_node->next = head;
        head->front = new_node;
        cur_node = new_node;
        fprintf(ftp, "%p\n", new_node);
    }
    return head;
}
```

绘制图表

先通过c语言将地址数据保存为10进制的数字形式方便绘制

```
fprintf(ftp, "%d\n", (int)new_node); //trace.c文件只需要修改这一句就可以
```

为了快速绘制，使用python读取生成的 addresses-locality.txt 文件，并通过matplotlib工具绘制


```
import matplotlib.pyplot as plt

res = []
with open('addresses-locality.txt', 'r', encoding='utf-8') as f:
    lines = f.readlines()
    for line in lines:
        res.append(int(line.split('\n')[0]))

plt.scatter(range(len(res)), res)
plt.show()
```

运行结果

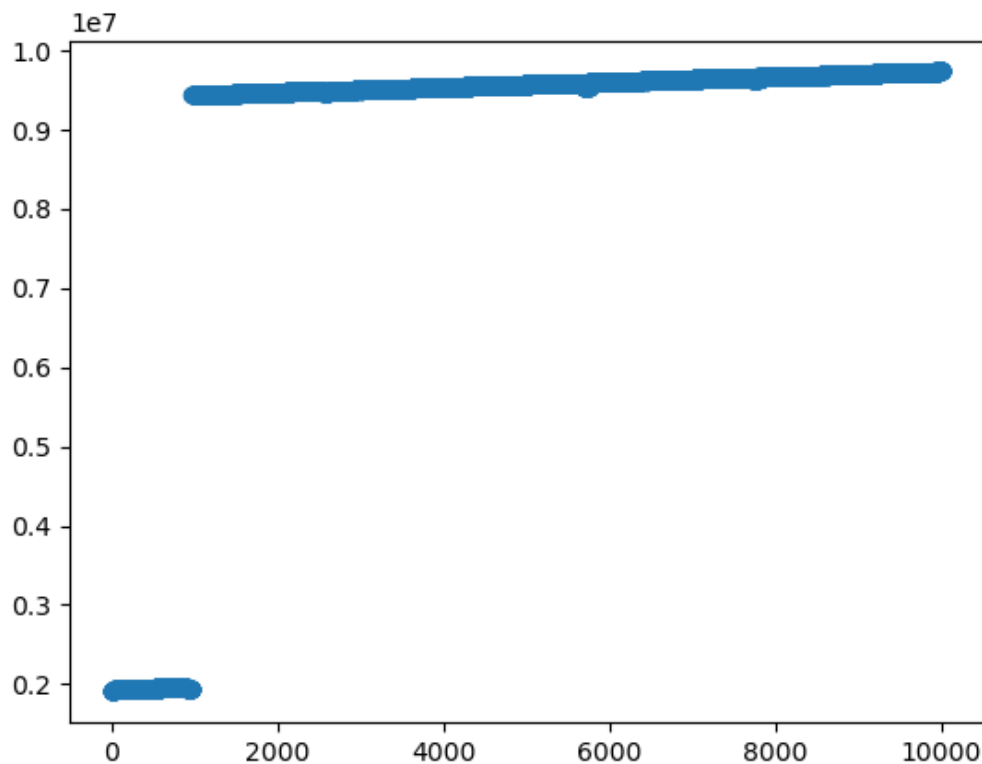
addresses-locality.txt输出

 addresses-locality.txt

文件(F) 编辑(E) 格式(O)

```
0x23e3260
0x23e4290
0x23e42b0
0x23e42d0
0x23e42f0
0x23e4310
0x23e4330
0x23e4350
0x23e4370
0x23e4390
0x23e43b0
0x23e43d0
0x23e43f0
0x23e4410
0x23e4430
0x23e4470
0x23e4490
0x23e44b0
0x23e44d0
0x23e44f0
0x23e4510
0x23e4530
0x23e4550
0x23e4570
0x23e4590
0x23e45b0
0x23e45d0
0x23e45f0
0x23e4610
```

生成的图表



在vm中测试FIFO和LRU的性能

注释掉打印的value后只输出 Page Faults 和 TLB Hits

FIFO

```
osc@ubuntu:~/vm-test$ ./vm2 -f addresses-locality.txt -b BACKING_STORE.bin -n 128 -p fifo
Page Faults = 2459
TLB Hits = 6417
```

LRU

```
osc@ubuntu:~/vm-test$ ./vm2 -f addresses-locality.txt -b BACKING_STORE.bin -n 128 -p lru
Page Faults = 2460
TLB Hits = 8191
```

虽然FIFO比LRU的Page Fault还少一个，但是LRU明显有更多的TLB Hits运行时间浪费就会少一些

2. Linux内存管理实验程序

Linux物理内存的组织形式

节选自 [linux/include/linux/mmzone.h](#) 文件中

存储节点——Node

首先, 内存被划分为结点. 每个结点关联到系统中的一个处理器, 内核中表示为 `pg_data_t` 的实例. 系统中每个结点被链接到一个以NULL结尾的 `pgdat_list` 链表中

```
typedef struct pglist_data {
    struct zone node_zones[MAX_NR_ZONES];
    struct zonelist node_zonelists[MAX_ZONELISTS];
    int nr_zones;
    ...
    struct per_cpu_nodestat __percpu *per_cpu_nodestats;
    atomic_long_t vm_stat[NR_VM_NODE_STAT_ITEMS];
} pg_data_t;
```

管理区——Zone

接着各个节点又被划分为内存管理区域, 一个管理区域通过 `struct zone_struct` 描述, 其被定义为 `zone_t`, 用以表示内存的某个范围, 低端范围的16MB被描述为 `ZONE_DMA`, 最后是超出了内核段的物理地址域 `ZONE_HIGHMEM`, 被称为高端内存.

```
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
};

struct zone {
    ...
} ____cacheline_internodealigned_in_smp;
```

而管理区的初始化则在 `mm/page_alloc.c` 文件中

```
/* In mm/page_alloc.c */
zone_t *zone_table[MAX_NR_ZONES*MAX_NR_NODES];
EXPORT_SYMBOL(zone_table);
```

页面——Page

最后页帧——page frame代表了系统内存的最小单位, 堆内存中的每个页都会创建一个 `struct page` 的一个实例

```
/* The array of struct pages - for discontigmem use pgdat->lmem_map */
extern struct page *mem_map;
```

每个物理的页由一个 `struct page` 的数据结构对象来描述。页的数据结构对象都保存在 `mem_map` 全局数组中

Page结构体中用到了非常多的联合体union，用于节省内存，代码太长节选了部分

```
struct page {
    unsigned long flags;
    union {
        struct { /* Page cache and anonymous pages */
            struct list_head lru;
            /* See page-flags.h for PAGE_MAPPING_FLAGS */
            struct address_space *mapping;
            pgoff_t index; /* Our offset within mapping. */
            unsigned long private;
        };
        struct { /* page_pool used by netstack */
            dma_addr_t dma_addr;
        };
        struct { /* slab, slob and slub */
            union {
                struct list_head slab_list; /* uses lru */
                struct { /* Partial pages */
                    struct page *next;
#ifdef CONFIG_64BIT
                    int pages; /* Nr of pages left */
                    int pobjects; /* Approximate count */
#else
                    short int pages;
                    short int pobjects;
#endif
                };
            };
            struct kmem_cache *slab_cache; /* not slob */
            /* Double-word boundary */
            void *freelist; /* first free object */
            union {
                void *s_mem; /* slab: first object */
                unsigned long counters; /* SLUB */
                struct { /* SLUB */
                    unsigned inuse:16;
                    unsigned objects:15;
                    unsigned frozen:1;
                };
            };
        };
    };
    struct { /* Tail pages of compound page */
        unsigned long compound_head; /* Bit zero is set */
        /* First tail page only */
        unsigned char compound_dtor;
        unsigned char compound_order;
        atomic_t compound_mapcount;
    };
    struct { /* Second tail page of compound page */
        unsigned long _compound_pad_1; /* compound_head */
        unsigned long _compound_pad_2;
        struct list_head deferred_list;
    };
    ...
};
union { /* This union is 4 bytes in size. */
```

```

    /*
     * If the page can be mapped to userspace, encodes the number
     * of times this page is referenced by a page table.
     */
    atomic_t _mapcount;
    /*
     * If the page is neither PageSlab nor mappable to userspace,
     * the value stored here may help determine what this page
     * is used for. See page-flags.h for a list of page types
     * which are currently stored here.
     */
    unsigned int page_type;
    unsigned int active;          /* SLAB */
    int units;                   /* SLOB */
};
...
};

```

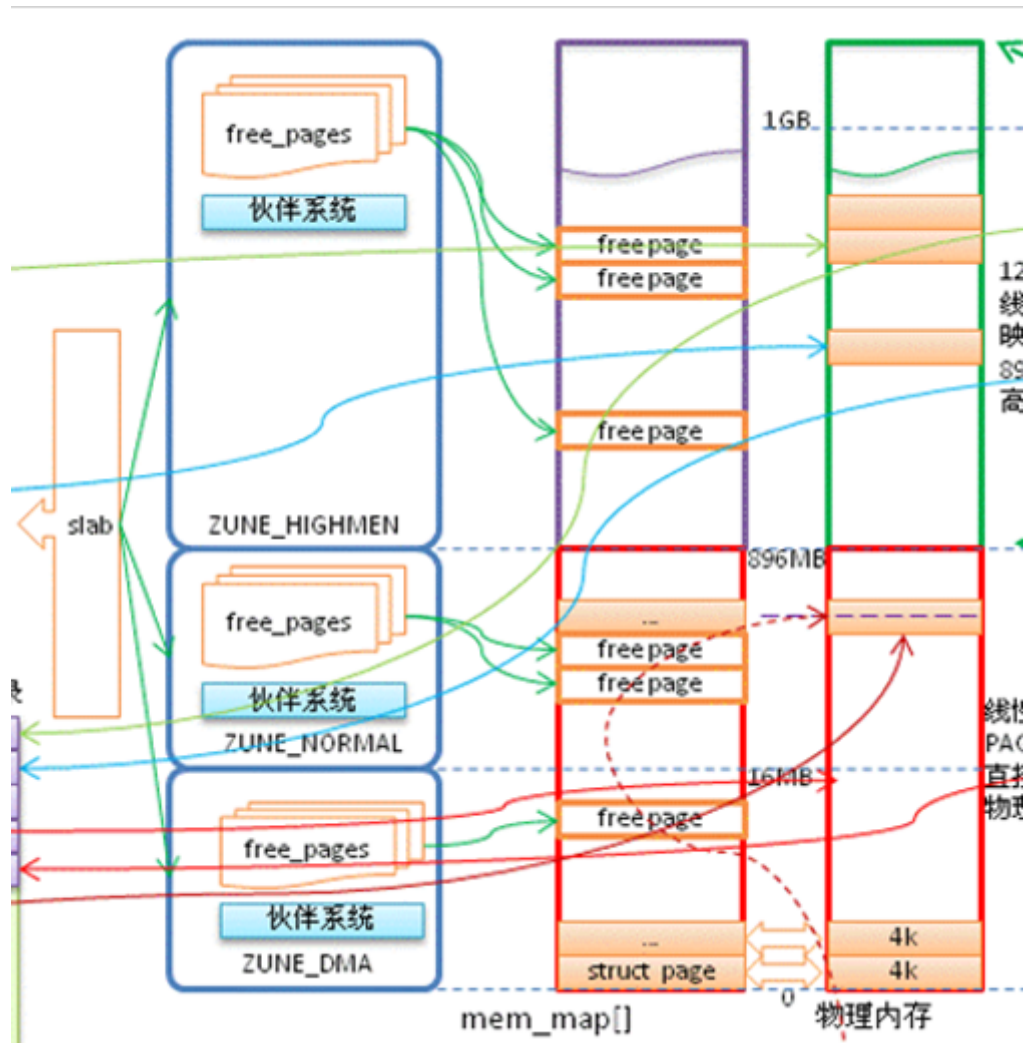
而其中需要特别提到lru字段，通过lru链表page可以实现伙伴系统中的链接作用，内存回收等

当一个页框属于一个slab的时候，这个页框的lru的prev指向的就是这个页框所属的slab。当页框属于cache的时候，页框的lru的next指向的就是这个页框所属的slab

基于三个基本单元的管理优化

1. 应用程序线性地址和动态内存分配
2. 高端内存和低端内存的划分
 - 映射到内核动态映射空间
 - 持久内核映射
 - 临时映射
3. 页框管理，保留的页框池
4. 虚拟内存管理
5. 分页机制

① 2级页表解析



slab层

有些多次会用到的数据结构如果频繁分配内存必然导致效率低下。slab层就是用于解决频繁分配和释放数据结构的问题

简单的说，物理内存中有多个高速缓存，每个高速缓存都是一个结构体类型，一个高速缓存中会有一个或多个slab，slab通常为一页，其中存放着数据结构类型的实例化对象。

分配高速缓存的接口，它返回的是 `kmem_cache` 结构体。第一个参数是缓存的名字，第二个参数是高速缓存中每个对象的大小，第三个参数是slab内第一个对象的偏移量

```
struct kmem_cache kmem_cache_create (const char *name, size_t size, size_t
align,unsigned long flags, void (*ctor)(void ))

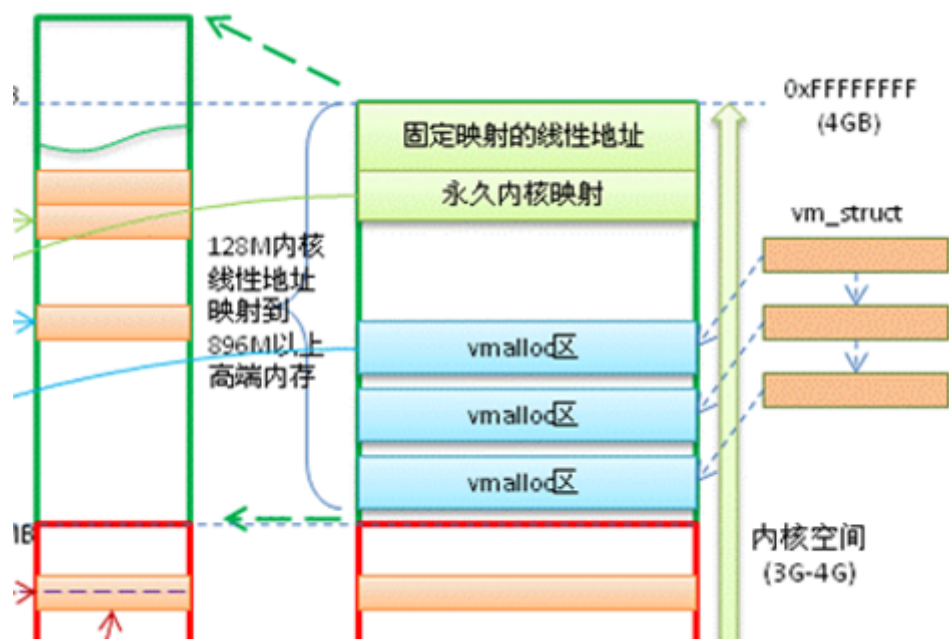
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
```

第二个是缓存中分配实例化的对象的接口。参数是 `kmem_cache` 结构体，也就是分配好的高速缓存，`flags`是标志位

在linux/include/linux/mmzone.h 中的伙伴系统的辅助数据结构

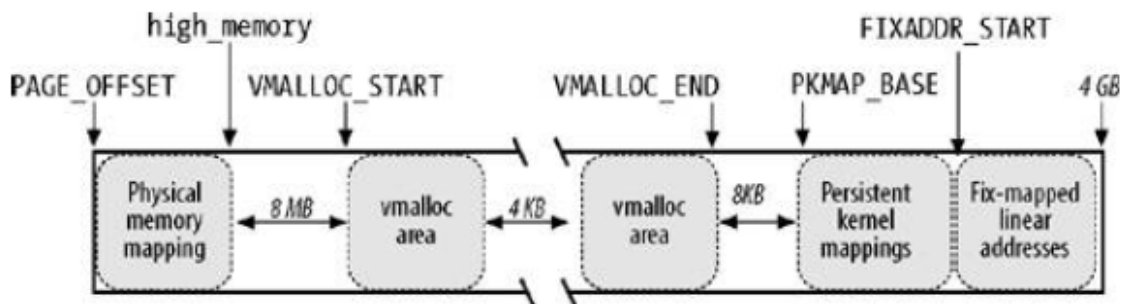
```
struct free_area {
    struct list_head    free_list[MIGRATE_TYPES];
    unsigned long       nr_free;
};
```

slab层按照伙伴系统，完成了大块内存的二分划分，就会出现一些零散的 freepage，等待被使用，或者长期空闲形成碎片。每个物理的页由一个 struct page 的数据结构对象来描述。页的数据结构对象都保存在 mem_map 全局数组中



高端内存的划分

在32位的系统上, 内核占有从第3GB~第4GB的线性地址空间, 共1GB大小



详细的划分和上图彩色图片对应

- 固定映射的线性地址和永久内核映射对应持久内核映射

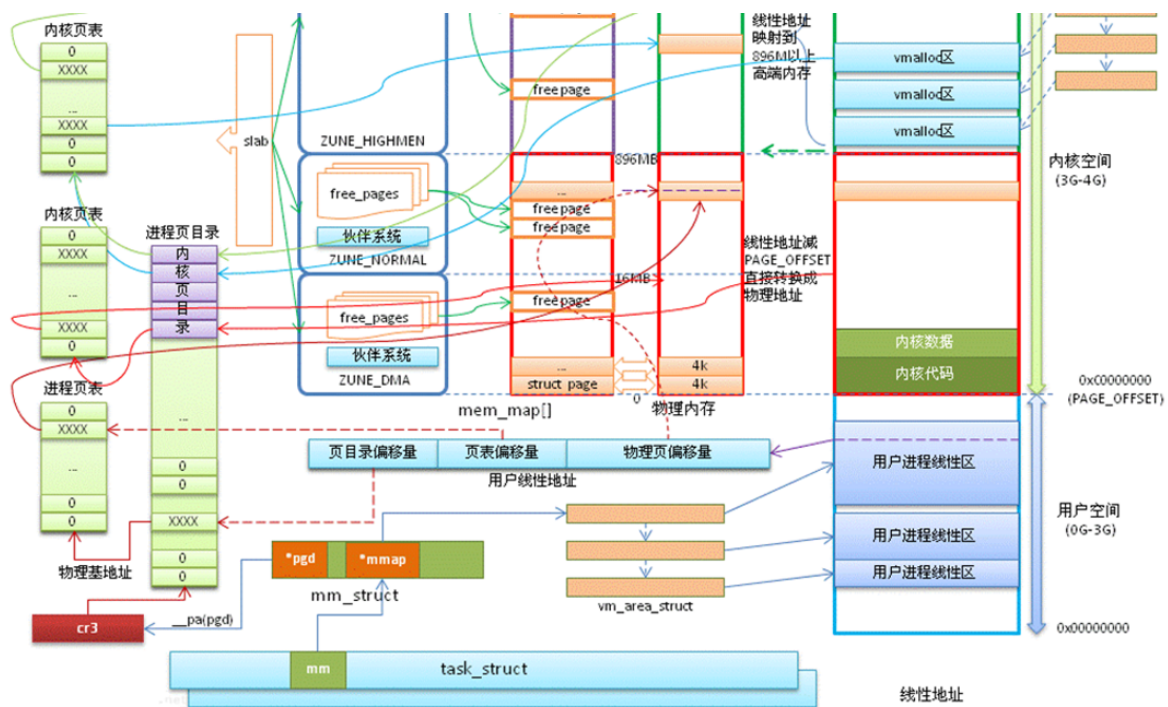
内核专门为此留出一块线性空间，从PKMAP_BASE 到 FIXADDR_START, 用于映射高端内存

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

- vmalloc区部分则对应内核动态映射

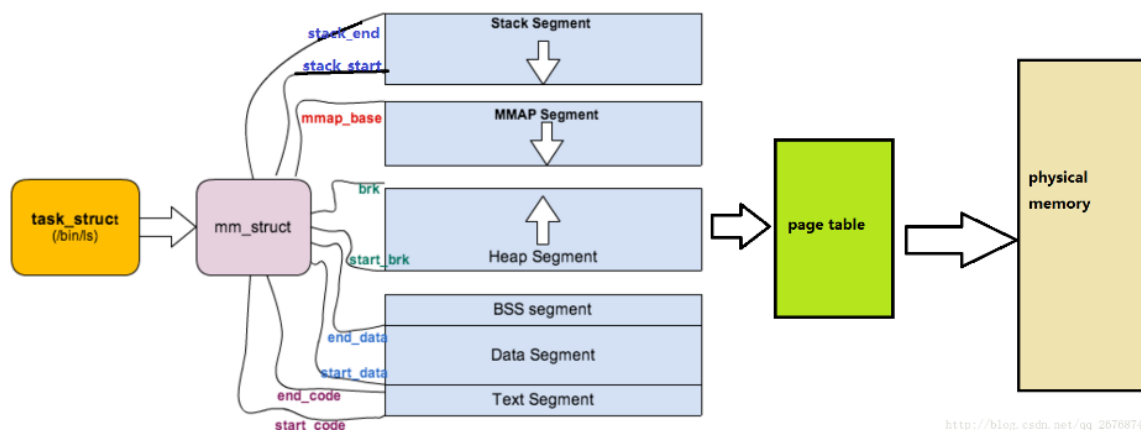
这种方式很简单，因为通过 vmalloc()，在“内核动态映射空间”申请内存的时候，就可能从高端内存获得页面

- 其他部分对应临时映射

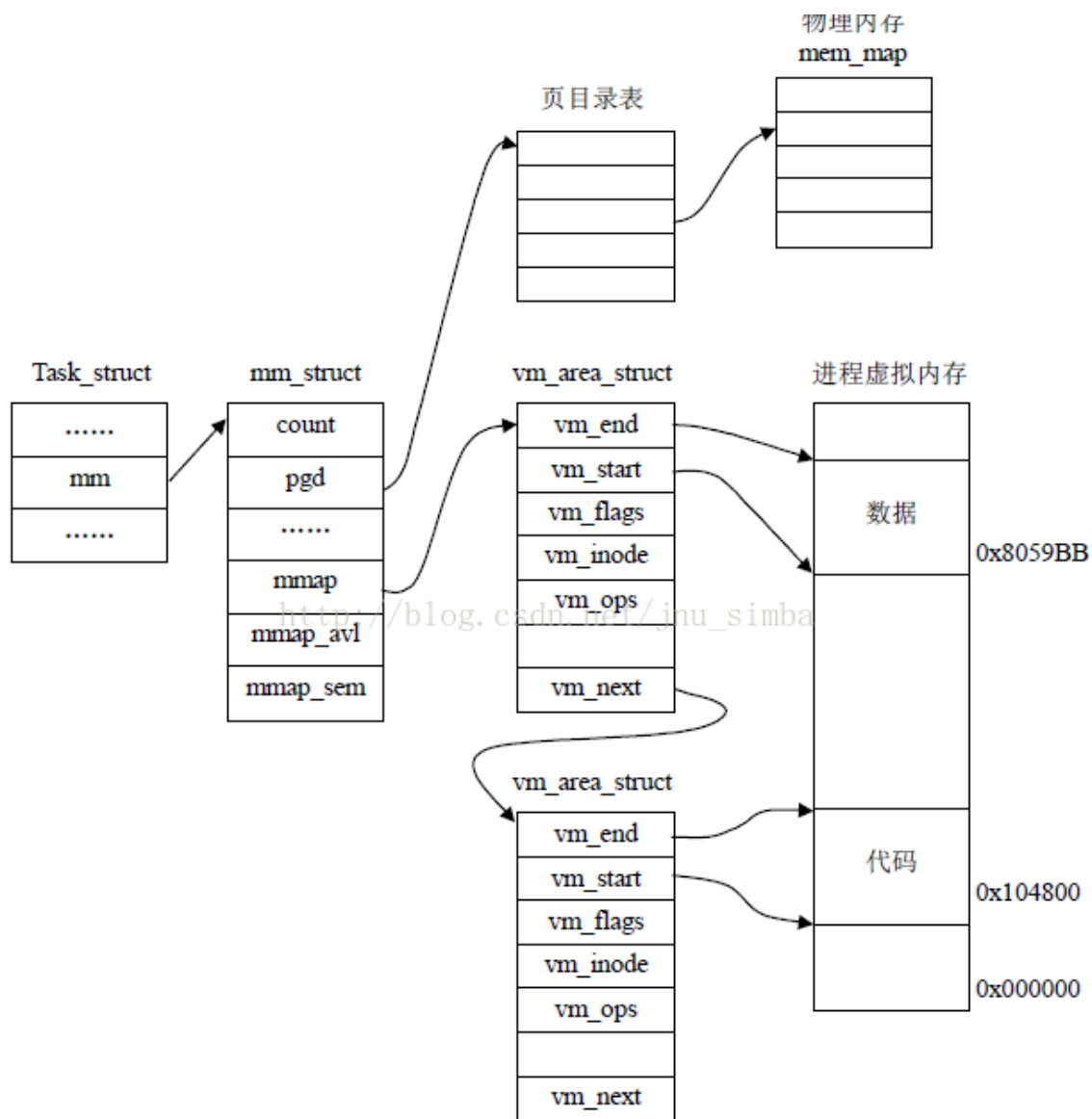


页表映射

1. 当一个任务task_struct来临，将任务



一个进程的虚拟地址空间主要由两个数据结果来描述。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对应图中的用户进程线性区。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



`mm_struct` 和 `vm_area_struct` 所在的路径是 [linux/include/linux/mm_types.h](http://blog.csdn.net/jnu_simba)

`vm_operations_struct` 在 [linux/include/linux/mm.h](http://blog.csdn.net/jnu_simba)

```
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 start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    ...
};

struct vm_area_struct
{
    struct mm_struct *vm_mm;               /* The address space we belong to. */
    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;
    ....
}
```

```

/* describe the permissible operation */
unsigned long vm_flags;
/* operations on this area */
struct vm_operations_struct * vm_ops;

struct file * vm_file; /* File we map to (can be NULL). */
} ;

/*
 * These are the virtual MM functions - opening of an area, closing and
 * unmapping it (needed to keep files on disk up-to-date etc), pointer
 * to the functions called when a no-page or a wp-page exception occurs.
 */
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);
};

```

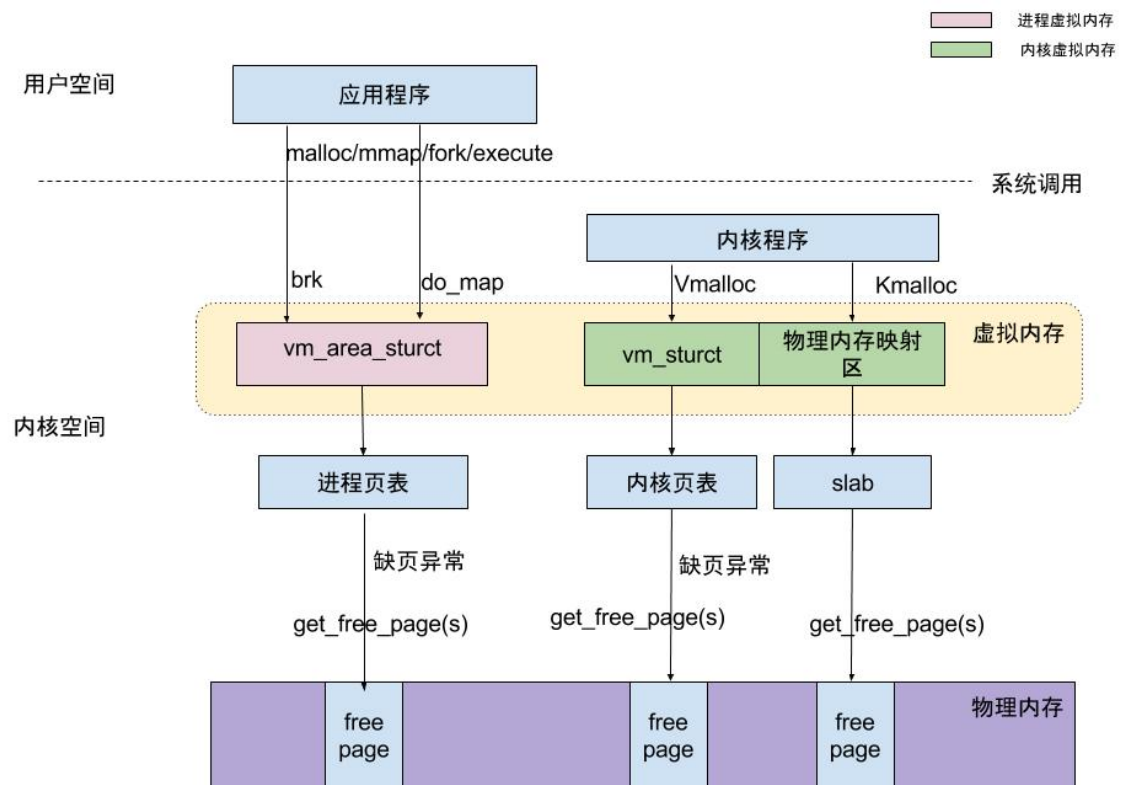
2.结合在CR3寄存器中存放的页目录的物理地址，再加上从虚拟地址中抽出高10位叫做页目录表项pgd的部分作为偏移，即定位到可以描述该地址的pgd；

3.从该pgd中可以获取可以描述该地址的页表的物理地址，再加上从虚拟地址中抽取中间10位作为偏移，即定位到可以描述该地址的pte；

4.在这个pte中即可获取该地址对应的页的物理地址，加上从虚拟地址中抽取的最后12位，即形成该页的页内偏移，即可最终完成从虚拟地址到物理地址的转换

唯一的不同在于内核和用户进程访问的页表不同

② 内核层不同内存分配接口的区别



__get_free_pages, alloc_page与get_free_page

在 [linux/mm/page_alloc.c](#) 中能够找到 `get_free_pages()`

其中用到的在 `alloc_pages()` [linux/include/linux/gfp.h](#) 中

```
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)
{
    struct page *page;
    page = alloc_pages(gfp_mask & ~__GFP_HIGHMEM, order);
    if (!page)
        return 0;
    return (unsigned long) page_address(page);
}

static inline struct page *
alloc_pages(gfp_t gfp_mask, unsigned int order)
{
    return alloc_pages_current(gfp_mask, order);
}
```

其功能简单说就是获得页与释放页获得页，是用户级的页操作。而`alloc_page`函数与`get_free_page`函数功能相似，只是相当于上述 `get_free_pages` 中参数`order`设置为0

kmalloc, vmalloc

前面讲的那些接口都是以页为单位进行内存分配与释放的。而在实际中内核需要的内存不一定是整个页，可能只是以字节为单位的一片区域。`kmalloc`，`vmalloc` 这两个函数就是实现这样的目的。不同之处在于，`kmalloc` 分配的是虚拟地址连续，物理地址也连续的一片区域，`vmalloc` 分配的是虚拟地址连续，物理地址不一定连续的一片区域

在 [linux/include/linux/slab.h](#) 中就能找到 `kmalloc`

在 [linux/mm/nommu.c](#) 中找到 `vmalloc`

```
static __always_inline void *kmalloc(size_t size, gfp_t flags)
{
    if (__builtin_constant_p(size)) {
#ifdef CONFIG_SLOB
        unsigned int index;
#endif
        if (size > KMALLOC_MAX_CACHE_SIZE)
            return kmalloc_large(size, flags);
#ifdef CONFIG_SLOB
        index = kmalloc_index(size);
        if (!index)
            return ZERO_SIZE_PTR;
        return kmem_cache_alloc_trace(
            kmalloc_caches[kmalloc_type(flags)][index],
            flags, size);
#endif
    }
    return __kmalloc(size, flags);
}

void *vmalloc(unsigned long size)
{
    return __vmalloc(size, GFP_KERNEL | __GFP_HIGHMEM, PAGE_KERNEL);
}
```

③ mtest.c 打印代码段、数据段、BSS，栈、堆等的相关地址

增加的内容

额外增加了常量的地址信息打印

```
const int MAX = 1024;
const int * add=&MAX;

printf("Data Locations:\n");
printf("\tAddress of const_int: %p\n", add);
```

并适当改变递归的次数

```
void afunc(void)
{
    static int level = 0;
    auto int stack_var;

    if(++level == 5)
        return;

    printf("\tStack level %d: address of stack_var: %p\n", level, &stack_var);
    afunc();
}
```

代码运行结果：

```

osc@ubuntu:~/vm-test$ ./mtest
PID: 2509
Data Locations:
    Address of const_int: 0x7fffd3891850
Text Locations:
    Address of main: 0x4006b6
    Address of afunc: 0x4008bc
Stack Locations:
    Stack level 1: address of stack_var: 0x7fffd3891824
    Stack level 2: address of stack_var: 0x7fffd3891804
    Stack level 3: address of stack_var: 0x7fffd38917e4
    Stack level 4: address of stack_var: 0x7fffd38917c4
    Start of alloca()'ed array: 0x7fffd3891810
    End of alloca()'ed array: 0x7fffd389182f
Data Locations:
    Address of data_var: 0x601060
BSS Locations:
    Address of bss_var: 0x60106c
Heap Locations:
    Initial end of heap: 0x94a000
    New end of heap: 0x94a020
    Final end of heap: 0x94a010

```

④ 分析mtest各个内存段

为了能够获取mtest进程的pid_number

```

pid_t pid = (int)syscall(__NR_getpid);
printf("PID: %d\n", pid);

```

然后在另一个终端输入

```

osc@ubuntu:~/vm-test$ cat /proc/2510/maps

```

运行结果

```

osc@ubuntu:~/vm-test$ cat /proc/2510/maps
00400000-00401000 r-xp 00000000 08:01 1182899 /home/osc/vm-test/mtest
00600000-00601000 r--p 00000000 08:01 1182899 /home/osc/vm-test/mtest
00601000-00602000 rw-p 00001000 08:01 1182899 /home/osc/vm-test/mtest
01ca1000-01cc4000 rw-p 00000000 00:00 0 [heap]
7f9bec591000-7f9bec751000 r-xp 00000000 08:01 134524 /lib/x86_64-linux-gnu/libc-2.23.so
7f9bec751000-7f9bec951000 ---p 001c0000 08:01 134524 /lib/x86_64-linux-gnu/libc-2.23.so
7f9bec951000-7f9bec955000 r--p 001c0000 08:01 134524 /lib/x86_64-linux-gnu/libc-2.23.so
7f9bec955000-7f9bec957000 rw-p 001c4000 08:01 134524 /lib/x86_64-linux-gnu/libc-2.23.so
7f9bec957000-7f9bec95b000 rw-p 00000000 00:00 0
7f9bec95b000-7f9bec981000 r-xp 00000000 08:01 134521 /lib/x86_64-linux-gnu/libd-2.23.so
7f9bec981000-7f9becb77000 rw-p 00000000 00:00 0
7f9becb77000-7f9becb81000 r--p 00025000 08:01 134521 /lib/x86_64-linux-gnu/libd-2.23.so
7f9becb81000-7f9becb82000 rw-p 00026000 08:01 134521 /lib/x86_64-linux-gnu/libd-2.23.so
7f9becb82000-7f9becb83000 rw-p 00000000 00:00 0
7fff11398000-7fff113b9000 rw-p 00000000 00:00 0 [stack]
7fff113d4000-7fff113d6000 r--p 00000000 00:00 0 [vvar]
7fff113d6000-7fff113d8000 r-xp 00000000 00:00 0 [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

```

分析列

一共有6列

- 第一列代表内存段的虚拟地址，对应vm_area_struct中的vm_start和vm_end
- 第二列代表执行权限，rw-p---权限 r-读，w-写 x-可执行 p-私有，s-共享。不用说，heap和stack段不应该有x，否则就容易被xx
- 第三列代表在进程地址里的偏移量，对应vm_pgoff
- 第四列映射文件的主设备号和次设备号，其中fd为主设备号，00为次设备号
- 第五列映像文件的节点号，即inode，对应vm_file->f_dentry->d_inode->i_ino

- 第六列是映像文件的路径，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。[stack]表示本段内存作为栈来使用，[heap]作为堆来使用，其他情况则为无

分析行

第六列为什么会有相同的文件路径

```
00400000-00401000 r-xp 00000000 08:01 1182899
/home/osc/vm-test/mtest
00600000-00601000 r--p 00000000 08:01 1182899
/home/osc/vm-test/mtest
00601000-00602000 rw-p 00001000 08:01 1182899
/home/osc/vm-test/mtest
```

一个是只读的，是代码段，一个是读写的，是数据段

- 堆[heap]段
- 共享库 .so
- 栈段[stack]

⑤ 内存分配相关问题

- 1、brk是将数据段(.data)的最高地址指针_edata往高地址推
- 2、sbrk不是系统调用，是C库函数。系统调用通常提供一种最小功能，而库函数通常提供比较复杂的功能
- 3、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。

这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系

⑥ 模仿malloc接口，实现简单的 myalloc/myfree

mempool.c部分细节

```
// Best fit
void * _mymallocBestFit(int flag)
{
    struct chunk * scp = g_chunk_head;
    struct chunk * mp = NULL;
    int bestSize = MEMPOOL_SIZE + 1;

    while (scp)
    {
        if ((FREE == scp->state) && scp->size > flag + _chunkSize && bestSize >
scp->size)
        {
            bestSize = scp->size;
            mp = scp;
        }

        scp = scp->next;
    }

    return _alloc(mp, flag);
}
```

```

}

// 测试函数
void showMempoolChunkInfo(struct chunk *scp)
{
    printf("%s%s%p%s%d%s%d",
           "\n=====chunk=====",
           "\naddress: ", scp,
           "\n state: ", scp->state,
           "\n size : ", scp->size);
}

void showMempoolInfo()
{
    struct chunk *scp = g_chunk_head;

    printf("\n***** begin *****\n");
    printf(" %s%d%s%d",
           "mempool total size: ", MEMPOOL_SIZE,
           "\n      chunk size: ", _chunkSize);
    int index = 0;

    while (scp)
    {
        printf("\n\n[index: %d]", index++);
        showMempoolChunkInfo(scp);
        scp = scp->next;
    }

    printf("\n-----end-----\n\n");
}

```

test.c测试代码和测试函数

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#include "mempool.h"

int main (int argc, const char * argv[])
{
    pid_t pid = (int)syscall(__NR_getpid);
    printf("PID: %d\n", pid);
    sleep(10);

    printf("myinit\n");
    myinit(BEST);

    int *p1 = (int *)mymalloc(sizeof(int) * 100);
    int *p2 = (int *)mymalloc(sizeof(int) * 200);
    int *p3 = (int *)mymalloc(sizeof(int) * 50);

    myfree(p1);
    sleep(10);
    //myfree(p2); //

```

```

    printf("myfree\n");
    myfree(p3);

    showMempoolInfo();

    return 0;
}

```

运行结果:

init阶段

```

osc@ubuntu: /vm-test$ cat /proc/2525/maps
00400000-00402000 r-xp 00000000 08:01 1182903 /home/osc/vm-test/test
00601000-00602000 r--p 00001000 08:01 1182903 /home/osc/vm-test/test
00602000-00603000 rw-p 00002000 08:01 1182903 /home/osc/vm-test/test
00603000-00803000 rw-p 00000000 00:00 0
02551000-02573000 rw-p 00000000 00:00 0
7f03371af000-7f033736f000 r-xp 00000000 08:01 134524 /lib/x86_64-linux-gnu/libc-2.23.so
7f033736f000-7f033756f000 --p 001c0000 08:01 134524 /lib/x86_64-linux-gnu/libc-2.23.so
7f033756f000-7f0337573000 r--p 001c0000 08:01 134524 /lib/x86_64-linux-gnu/libc-2.23.so
7f0337573000-7f0337575000 rw-p 001c4000 08:01 134524 /lib/x86_64-linux-gnu/libc-2.23.so
7f0337575000-7f0337579000 rw-p 00000000 00:00 0
7f0337579000-7f033759f000 r-xp 00000000 08:01 134521 /lib/x86_64-linux-gnu/ld-2.23.so
7f0337792000-7f0337795000 rw-p 00000000 00:00 0
7f033779e000-7f033779f000 r--p 00025000 08:01 134521 /lib/x86_64-linux-gnu/ld-2.23.so
7f033779f000-7f03377a0000 rw-p 00026000 08:01 134521 /lib/x86_64-linux-gnu/ld-2.23.so
7f03377a0000-7f03377a1000 rw-p 00000000 00:00 0
7ffee6381000-7ffee63a2000 rw-p 00000000 00:00 0 [stack]
7ffee63aa000-7ffee63ac000 r--p 00000000 00:00 0 [vvar]
7ffee63ac000-7ffee63ae000 r-xp 00000000 00:00 0 [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

```

myfree阶段

```

osc@ubuntu: /vm-test$ cat /proc/2525/maps
00400000-00402000 r-xp 00000000 08:01 1182903 /home/osc/vm-test/test
00601000-00602000 r--p 00001000 08:01 1182903 /home/osc/vm-test/test
00602000-00603000 rw-p 00002000 08:01 1182903 /home/osc/vm-test/test
00603000-00803000 rw-p 00000000 00:00 0
02551000-02573000 rw-p 00000000 00:00 0
7f03371af000-7f033736f000 r-xp 00000000 08:01 134524 /lib/x86_64-linux-gnu/libc-2.23.so
7f033736f000-7f033756f000 --p 001c0000 08:01 134524 /lib/x86_64-linux-gnu/libc-2.23.so
7f033756f000-7f0337573000 r--p 001c0000 08:01 134524 /lib/x86_64-linux-gnu/libc-2.23.so
7f0337573000-7f0337575000 rw-p 001c4000 08:01 134524 /lib/x86_64-linux-gnu/libc-2.23.so
7f0337575000-7f0337579000 rw-p 00000000 00:00 0
7f0337579000-7f033759f000 r-xp 00000000 08:01 134521 /lib/x86_64-linux-gnu/ld-2.23.so
7f0337792000-7f0337795000 rw-p 00000000 00:00 0
7f033779e000-7f033779f000 r--p 00025000 08:01 134521 /lib/x86_64-linux-gnu/ld-2.23.so
7f033779f000-7f03377a0000 rw-p 00026000 08:01 134521 /lib/x86_64-linux-gnu/ld-2.23.so
7f03377a0000-7f03377a1000 rw-p 00000000 00:00 0
7ffee6381000-7ffee63a2000 rw-p 00000000 00:00 0 [stack]
7ffee63aa000-7ffee63ac000 r--p 00000000 00:00 0 [vvar]
7ffee63ac000-7ffee63ae000 r-xp 00000000 00:00 0 [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

```

内存块管理方面优化

```

void myinit(int flag)
{
    static char poolState = 0; // 防止多次初始化
    if (poolState) return;

    poolState = 1;
    atexit(memoryLeak); // 检测出内存泄漏
    _chunkSize = sizeof(struct chunk);

    g_algorithm = flag;
    g_chunk_head = (struct chunk *)g_mempool;
    strncpy(g_chunk_head->signature, "OSEX", 4);
    g_chunk_head->next = NULL;
    g_chunk_head->state = FREE;
    g_chunk_head->size = MEMPOOL_SIZE - _chunkSize; // 实际可用内存
}

```

加入了内存溢出泄露的检测

参考文献

1. <https://blog.csdn.net/gatieme/article/details/52384058> Linux内存描述之概述--Linux内存管理 (一) —— Linux内存管理(二十一)
2. <https://blog.csdn.net/hty46565/article/details/74938642> 内存管理 (Linux内核源码分析)
3. https://blog.csdn.net/qq_26768741/article/details/54375524 进程—内存描述符 (mm_struct)
4. <https://www.cnblogs.com/feng9exe/p/6379650.html> 内存管理概述、内存分配与释放、地址映射机制 (mm_struct, vm_area_struct) 、 malloc/free 的实现
5. http://www.sohu.com/a/191140670_467784 内存管理 (内核中分配与回收内存的函数)
6. <https://code.woboq.org/linux/linux/> Linux内核代码在线查找
7. <https://blog.csdn.net/unbutun/article/details/4901800> 简单解读linux的/proc下的statm、maps、memmap 内存信息文件分析
8. <https://bbs.csdn.net/topics/360002818> CSDN论坛