



本科生实验报告

学生姓名： 丁晓琪

学生学号： 22336057

专业名称： 计科

Assignment1

- 1.实验要求:
- 2.实验过程:
- 3.关键代码:
- 4.实验结果:
- 5.总结

Assignment2

- 1.实验要求:
- 2.实验过程:
- 3.关键代码:
- 4.实验结果:
- 5.总结

Assignment3

- 1.实验要求:
- 2.实验过程:
- 3.关键代码:
- 4.实验结果:
- 5.总结:

Assignment4

- 1.实验要求:
- 2.实验过程:
- 3.关键代码:
- 4.实验结果:
- 5.总结:

## Assignment1

### 1.实验要求:

复现参考代码，实现二级分页机制，并能够在虚拟机地址空间中进行内存管理，包括内存的申请和释放等，截图并给出过程解释。

## 2.实验过程：

- 1.实现位图
- 2.实现地址池
- 3.初始化页表，开启二级页表机制

## 3.关键代码：

### 3.1实现位图：

- 作用：用一个bit记录一个分区的分配情况，0为未分配，1为已分配，在分页机制中就是记录每一页的分配情况，便于开启分页机制后管理页的分配和释放
- 数据结构：  
`char* bitmap`：一块记录分区分配情况的空间，注意由于是1bit记录一个分区，则操作精度要到每个bit上
- 主要函数：  
`int allocate(const int count)`：遍历 `bitmap`，查找连续的count个未被分配的分区，并将其分配  
`void release(const int index, const int count)`：释放第index个分区开始的count个跟去
- 注意：单纯的位图实现中并没有说明每个资源的含义和大小，只是将资源做离散化管理，具体地对资源管理的一些定义仍需进一步设置定义

```
1  class BitMap
2  {
3  public:
4      // 被管理的资源个数，bitmap的总位数
5      int length;
6      // bitmap的起始地址
7      char *bitmap;
8  public:
9      // 初始化
10     BitMap();
11     // 设置BitMap, bitmap=起始地址, length=总位数(即被管理的资源个数)
12     void initialize(char *bitmap, const int length);
13     // 获取第index个资源的状态, true=allocated, false=free
14     bool get(const int index) const;
15     // 设置第index个资源的状态, true=allocated, false=free
16     void set(const int index, const bool status);
17     // 分配count个连续的资源, 若没有则返回-1, 否则返回分配的第1个资源单元序号
18     int allocate(const int count);
19     // 释放第index个资源开始的count个资源
20     void release(const int index, const int count);
21     // 返回Bitmap存储区域
22     char *getBitmap();
23     // 返回Bitmap的大小
24     int size() const;
25 private:
26     // 禁止Bitmap之间的赋值
27     BitMap(const BitMap &) {}
```

```

28     void operator=(const BitMap&) {}
29 };

```

### 3.2地址池实现

- 作用：在位图的基础上进一步申明管理的资源的含义和大小，管理的资源是内存且管理粒度为一页4kb

- 数据结构：

`BitMap resources`: 标识内存中划分出每一页的分配情况

`int startAddress`: 记录地址池管理的页的共同起始地址。（索引地址池第*i*页的起始地址：  
`address=startAddress+i×PAGESIZE`）

- 主要函数：

`int allocate(const int count)`: 在地址池中分配count页内存，通过位图 `resources` 的分配函数返回连续count个空闲页开头索引 `start`，再根据 `startAddress` 和 `start` 计算被分配的空闲页的开头地址

`int allocate(const int count)`: 释放开始地址为address的amount页。先根据pagesize, address, startAddress计算出释放页在位图中的索引 `index`,再通过位图 `resources` 的释放函数将这些页的状态都置为未分配。

```

1  class AddressPool
2  {
3  public:
4      BitMap resources;
5      int startAddress;
6
7  public:
8      AddressPool();
9      // 初始化地址池
10     void initialize(char *bitmap, const int length, const int
startAddress);
11     // 从地址池中分配count个连续页，成功则返回第一个页的地址，失败则返回-1
12     int allocate(const int count);
13     // 释放若干页的空间
14     void release(const int address, const int amount);
15 };

```

### 3.3 物理内存的实现：

- 作用：地址池实现对整个物理内存划分为页，需要进一步划分用户空间和内核空间

- 数据结构：

`AddressPool kernelPhysical`; 内核地址池

`AddressPool userPhysical`; 物理地址池

- 主要函数：

`void initialize()`: 留出给内核的0x100000,再预留出1MB（256个4kb页）给内核页目录表和页表，剩下的物理内存内核和用户地址池平分，同时指定内核和物理位图的位置起始0x10000。根据计算出的内存和用户地址池的开始地址和大小以及定义好的位图位置初始化 `kernelPhysical`, `userPhysical`。

`int allocatePhysicalPages(enum AddressPoolType type, const int count)`:根据指定类型去指定地址池调用地址池分配函数分配count页, 并且返回页的开始物理地址

`void releasePhysicalPages(enum AddressPoolType type, const int paddr, const int count)`: 根据指定的类型去指定地址池调用释放函数, 释放起始地址为 paddr 的 count 页

```
1  class MemoryManager
2  {
3  public:
4      // 可管理的内存容量
5      int totalMemory;
6      // 内核物理地址池
7      AddressPool kernelPhysical;
8      // 用户物理地址池
9      AddressPool userPhysical;
10 public:
11     MemoryManager();
12     // 初始化地址池
13     void initialize();
14     // 从type类型的物理地址池中分配count个连续的页
15     // 成功, 返回起始地址; 失败, 返回0
16     int allocatePhysicalPages(enum AddressPoolType type, const int count);
17     // 释放从paddr开始的count个物理页
18     void releasePhysicalPages(enum AddressPoolType type, const int paddr,
19 const int count);
19     // 获取内存总容量
20     int getTotalMemory();
21 };
```

### 3.4开启二级分页机制:

- 作用:

一级页表:

一个页表4K大小, 虚拟地址中需要12位表示页内偏移  
32-12=20, 剩下的20位是用来索引虚拟页到物理页的映射在页表中的偏移  
页表项一共有 $2^{20}$ 项, 每一项占4个字节  
一个页表能够查找1M (页的数目) \* 4KB(一个页大小)=4G大小的空间

虚拟地址到物理地址映射: 读取高20位索引页表项, 找到页表项对应的物理地址  
从虚拟地址低12位读出页内偏移, 页地址+页内偏移=物理地址

缺点: 需要4MB存储一个页表, 而每个进程都需要页表

二级页表:

页目录表, 页表, 物理页大小都为4KB  
一个页表有1KB的页表项, 能表示 $1KB * 4KB = 4M$ 的虚拟空间大小  
一个页目录表有1KB的页目录项, 能表示 $1KB * 4M = 4G$ 的虚拟空间大小  
还能节省空间, 一整页都没有项的页在页目录表中对应的为0, 而不需要再把这个页给创造出来, 浪费空间

32位的虚拟地址划分:

高10bit: 表示1024个页目录项 ( $2^{10}$ ), 对应着页表的起始距离  
中10bit: 页表项在页表中的偏移, 对应每个页对应的物理页的物理地址  
低12bit: 表示页内偏移

- 步骤:

1.初始化内核页表和页目录表:

- 目前物理内存的使用情况：只有内核在使用而且不超过1MB，所以要为1MB内存做好页表和页目录的映射
  - 页目录初始化：前1MB的高10位始终为0，则需要初始化第0个页目录项，将其映射到内核第一个页表的起始地址。页目录表实际上也是一个页表，也要能通过页目录表寻找，所以将页目录表的最后一项映射为页目录表的地址
  - 内核第一个页表初始化：前1mb的中间10位只有后8位有值， $1\text{mb}/4\text{kb}=256$ 页，所以需要初始化256个页表项，且定义0-1MB是恒等映射
- 2.将页目录表的地址写入cr3，让MMU能找到页目录表，自动将CPU生成的逻辑地址转化为物理地址
- 3.将cr0的PG位置1，开启分页机制

```

1 void MemoryManager::openPageMechanism()
2 {
3     // 页目录表指针
4     int *directory = (int *)PAGE_DIRECTORY; //放到0x100000的位置（十六进制也就是
    在1MB位置处）
5     //线性地址0~4MB对应的页表
6     int *page = (int *) (PAGE_DIRECTORY + PAGE_SIZE); //之前预留出了 $2^8 \times 4\text{KB} = 1\text{MB}$ 的
    位置，前面4KB代表页目录指针，其他都是代表页表
7     // 初始化页目录表
8     memset(directory, 0, PAGE_SIZE);
9     // 初始化线性地址0~4MB对应的页表
10    memset(page, 0, PAGE_SIZE); //一个页表可以表示4MB的虚拟内存，现在这个页表要映射内
    存中前4MB包括预留给内核的1MB的物理内存
11    int address = 0;
12    // 将线性地址0~1MB恒等映射到物理地址0~1MB（预留给内核的1MB的位置是虚拟地址和物理地址
    相同） $1\text{MB}/4\text{KB} = 2^8$ 个页
13    //前1MB 二进制：|00100 00000|00000 00000 00 中间10位只有后8位有值，所以需要初
    始化 $2^8 = 256$ 个表项
14    //初始化页表
15    for (int i = 0; i < 256; ++i)
16    {
17        // U/S = 1, R/W = 1, P = 1
18        page[i] = address | 0x7; //信息位的初始化
19        address += PAGE_SIZE; //加载高20位的页基地址，由于是4KB的倍数，所以只关心高20
    位
20    }
21    // 初始化页目录项
22    //前1MB的高10位始终是000，则页目录项只要初始化第0项
23    // 0~1MB
24    directory[0] = ((int)page) | 0x07;
25    // 3GB的内核空间
26    directory[768] = directory[0];
27    // 最后一个页目录项指向页目录表
28    directory[1023] = ((int)directory) | 0x7;
29
30    // 初始化cr3, cr0, 开启分页机制
31    asm_init_page_reg(directory);
32
33    printf("open page mechanism\n");
34
35 }
36

```

```

1  asm_init_page_reg:
2      push ebp
3      mov ebp, esp
4      push eax
5      mov eax, [ebp + 4 * 2]
6      mov cr3, eax ; 放入页目录表地址
7      mov eax, cr0
8      or eax, 0x80000000 ; 31位变成1
9      mov cr0, eax ; 置PG=1, 开启分页机制
10     pop eax
11     pop ebp
12     ret

```

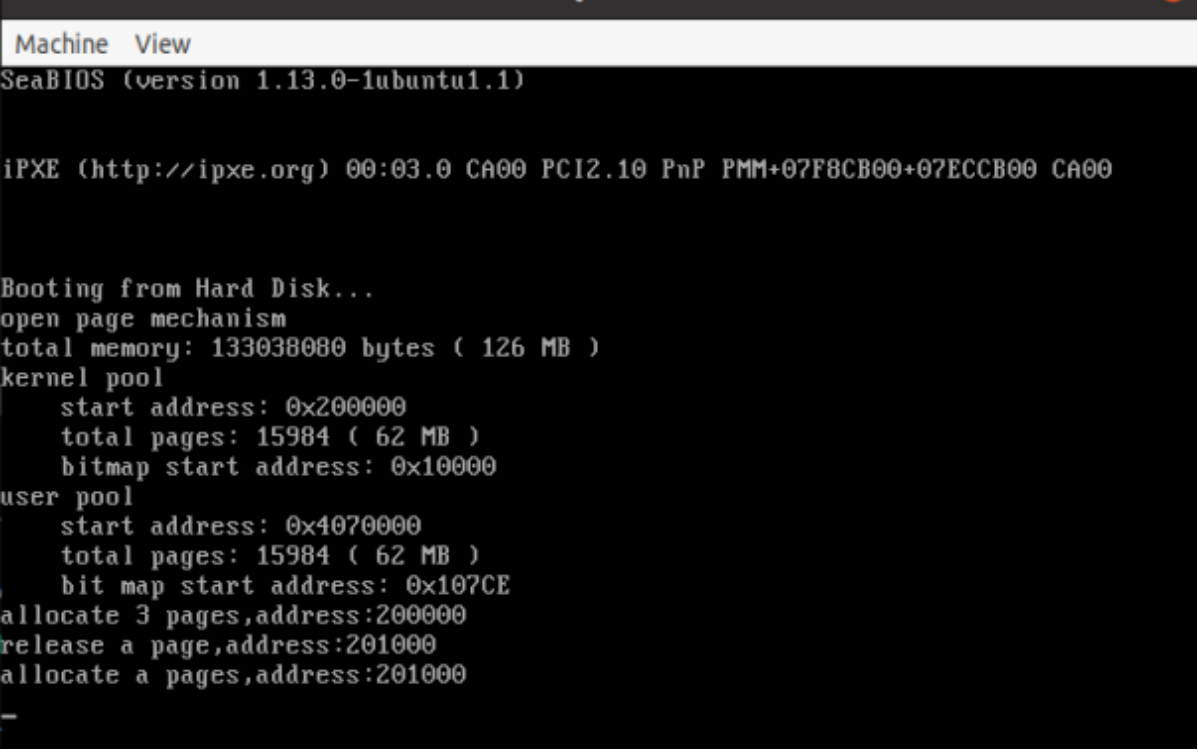
## 4.实验结果：

成功分配和释放内存

```

1  int p1=memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL,3);
2  printf("allocate 3 pages,address:%x\n",p1);
3  int p2=p1+PAGE_SIZE;
4  memoryManager.releasePhysicalPages(AddressPoolType::KERNEL,p2, 1);
5  printf("release a page,address:%x\n",p2);
6  int p3=memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL,1);
7  printf("allocate a pages,address:%x\n",p3);

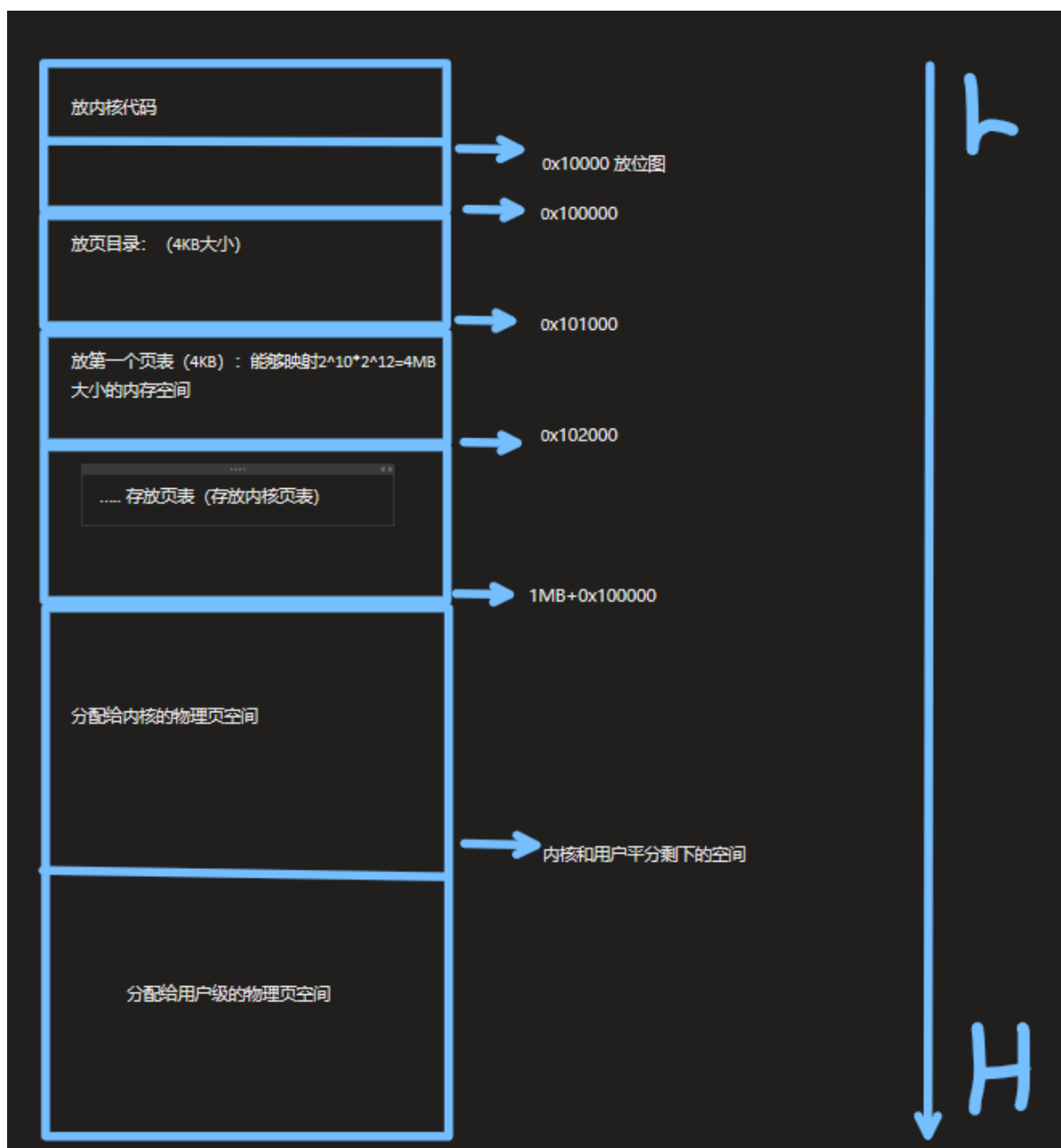
```



The screenshot shows the SeaBIOS boot interface. At the top, it says "Machine View" and "SeaBIOS (version 1.13.0-1ubuntu1.1)". Below that, it lists supported features: "iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00". The main part of the screen shows the boot process: "Booting from Hard Disk...", "open page mechanism", "total memory: 133038080 bytes ( 126 MB )", "kernel pool" (start address: 0x200000, total pages: 15984 ( 62 MB ), bitmap start address: 0x10000), "user pool" (start address: 0x4070000, total pages: 15984 ( 62 MB ), bit map start address: 0x107CE). The bottom of the screen shows the output of the memory management operations: "allocate 3 pages,address:200000", "release a page,address:201000", and "allocate a pages,address:201000".

## 5.总结

开启分页机制后物理内存图解：



## Assignment2

### 1.实验要求:

参照理论课上的学习的物理内存分配算法如first-fit, best-fit等实现动态分区算法等，或者自行提出自己的算法。这里实现first-fit动态分区

### 2.实验过程:

1. 实现和初始化存储内存中孔（分区）的信息的数据结构（Fit\_List 链表），构建管理分区的类 `First_fit`
2. 实现 first\_fit 算法，分配 `Fit_List::allocate(int size)` 和释放 `Fit_List::release(int start_address, int size)`
3. 测试算法，尝试在内存分区中分配和释放空洞

### 3.关键代码:

#### 3.1.1 实现和初始化 `Fit_List`:

- 链表节点 `Fit_ListItem`:

每个链表节点代表内存中的一个分区，里面保存分区的开始地址，大小，是否被分配

```
1 struct Fit_ListItem
2 {
3     int begin_address; //空闲孔开始地址
4     int size; //空闲孔的大小
5     int is_allocate; //是否被分配, 0为未被分配, 1为被分配
6     Fit_ListItem *previous;
7     Fit_ListItem *next;
8 };
```

- 分区链表 `Fit_List` 的初始化:

预先在全局中分配100个孔洞分区对象 `Fit_Item item[100]`，初始化链表头节点和 `item[100]`

```
1 Fit_ListItem item[100]; //全局变量, 避免局部变量被释放
2 void Fit_List::initialize()
3 {
4     head->next = head->previous = 0; //有头节点的
5     head->size=0;
6     //初始化孔洞表
7     for(int i=0; i<100; i++){
8         item[i].begin_address=-1;
9         item[i].is_allocate=0;
10        item[i].previous=0;
11        item[i].next=0;
12    }
13 }
```

### 3.1.2 实现和初始化 `First_fit`:

- `First_fit` 数据结构:

- 类成员: 保存维护分区信息的链表 `resources`，管理的连续内存分区的开始地址 `startAddress`，`resources` 的头节点 `start`，`resources` 的第一个节点 `one`

- 类成员函数:

`initialize`: 初始化动态分区管理类，参数为管理的连续内存的开始地址和长度

`allocate`: 分配一个size大小的分区

`release`: 释放一个指定开始地址的指定大小的分区

- 为什么会有 `start` 成员:

由于在内存管理中需要通过 `resources` 分配，我们需要指定和保存 `resources` 的地址信息，这个start节点就是为了记录下 `resources` 的地址信息

```
1 class First_fit
2 {
3 public:
4     Fit_List resources;
5     int startAddress;
6     Fit_ListItem start; //
7     Fit_ListItem one;
```



```

8 public:
9     // 初始化地址池
10    void initialize(const int length, const int startAddress);
11    // 从地址池中分配count个连续页，成功则返回第一个页的地址，失败则返回-1
12    int allocate( int size);
13    // 释放若干页的空间
14    void release(const int address,int size);
15    void print();
16 };

```

- 初始化 First\_fit:

在resources中建立一个分区节点，代表一开始没有分配还是一整块连续分区的整块被管理的连续内存

```

1 void First_fit::initialize(const int length, const int startAddress)
2 { //start为链表存放位置
3     resources.head=&start; //更换链表开始位置
4     resources.initialize();
5     one.begin_address=startAddress;
6     one.size=length;
7     one.is_allocate=0;
8     one.previous=resources.head;
9     one.next=0;
10    resources.head->next=&one;
11 }

```

### 3.2分区的分配与释放

- 分区的分配:

在 resources 链表中查找到第一个能装申请空间且未被分配的空洞

```

1 Fit_ListItem* Fit_List::find_fit(int size)
2 {
3     int pos = 0;
4     Fit_ListItem *temp = head->next;
5     while (temp&&(temp->size<size || temp->is_allocate==1 ))
6     {
7         temp = temp->next;
8         ++pos;
9     }
10    if (temp && temp->size>=size && temp->is_allocate==0)
11    {
12        return temp;
13    }
14    else
15    {
16        return 0;
17    }
18 }

```

如果分配的孔洞比申请的空间还要大，会产生内部碎片，需要在 resources 中插入新的分区节点。

```

1 int Fit_List::allocate(int size)

```

```

2  {
3      Fit_ListItem* itemPtr=find_fit(size);
4      while(itemPtr==0){
5          itemPtr=find_fit(size);
6      }//分配到孔和位置
7      int hole_size=itemPtr->size;
8      if(hole_size==size){
9          itemPtr->is_allocate=1;
10         return itemPtr->begin_address;
11     }
12     else{
13         //生成一个新的孔
14         Fit_ListItem* new_hole=find_hole();//在已经初始化好的分区节点对象中挑选一个改造插入
15         new_hole->begin_address=itemPtr->begin_address+size;
16         new_hole->size=hole_size-size;
17         new_hole->previous=itemPtr;
18         new_hole->next=itemPtr->next;
19         itemPtr->next=new_hole;
20         (new_hole->next)->previous=new_hole;
21         new_hole->is_allocate=0;
22         itemPtr->is_allocate=1;
23         itemPtr->size=size;
24         return itemPtr->begin_address;
25     }
26 }
27 int First_fit::allocate(const int size)
28 {
29     resources.allocate(size);
30 }

```

- 分区的释放:

参数: 释放的分区的起始地址, 释放的空间大小

实现: 在 `resources` 中找到要释放分区所在节点, 处理好释放时产生的碎片和释放后与前后空闲分区的合并

```

1  void Fit_List::release(int start_address,int size){
2      Fit_ListItem* itemPtr=find_release(start_address);
3      if (itemPtr==0){
4          printf("error\n");
5      }
6      if(itemPtr->size<size){
7          printf("error\n");
8      }
9      itemPtr->is_allocate=0;
10     Fit_ListItem* previous=itemPtr->previous;
11     Fit_ListItem* next=itemPtr->next;
12     Fit_ListItem* new_hole=find_hole();
13     if(itemPtr->size==size){//无碎片
14         if(previous==head){
15             itemPtr->is_allocate=0;
16         }
17         else if(previous->is_allocate==0){//与前面分区合并
18             itemPtr->begin_address=previous->begin_address;
19             itemPtr->size=itemPtr->size+previous->size;

```

```

20     erase(previous);
21 }
22 if(next&&next->is_allocate==0){//与后面分区合并
23     itemPtr->size=itemPtr->size+next->size;
24     erase(next);
25 }
26 }
27 else{ //有碎片
28     //合并
29     if(previous!=head&&previous->is_allocate==0){
30         previous->size=previous->size+next->size;
31         //更新
32         itemPtr->is_allocate=1;
33         itemPtr->begin_address=itemPtr->begin_address+next->size;
34         itemPtr->size=itemPtr->size-next->size;
35     }else{
36         new_hole->begin_address=itemPtr->begin_address;
37         new_hole->size=next->size;
38         new_hole->is_allocate=0;
39         itemPtr->is_allocate=1;
40         itemPtr->begin_address=itemPtr->begin_address+next->size;
41         itemPtr->size=itemPtr->size-next->size;
42         new_hole->previous=previous;
43         new_hole->next=itemPtr;
44         previous->next=new_hole;
45         itemPtr->previous=new_hole;
46     }
47 }
48 }
49 void First_fit::release(const int address,int size)
50 {
51     resources.release(address,size);
52 }

```

## 4.实验结果：

- 测试：

初始化：管理126MB和起始地址为0的连续内存

分配：分配一个大小为3的分区

释放：释放一个起始地址为0，大小为1的内存分区

```

1 first_fit.initialize(126,0);
2 printf("before allocate:\n");
3 first_fit.print();
4 printf("\nallocate 3\n");
5 first_fit.allocate(3);
6 first_fit.print();
7 printf("\nrelease 3\n");
8 first_fit.release(0,1);
9 first_fit.print();

```

- 结果：

```

booting from hard disk...
total memory: 133038080 bytes (126 MB)
before allocate:
1_hole:
  startAddress:0 || is_allocate:0 || size:126

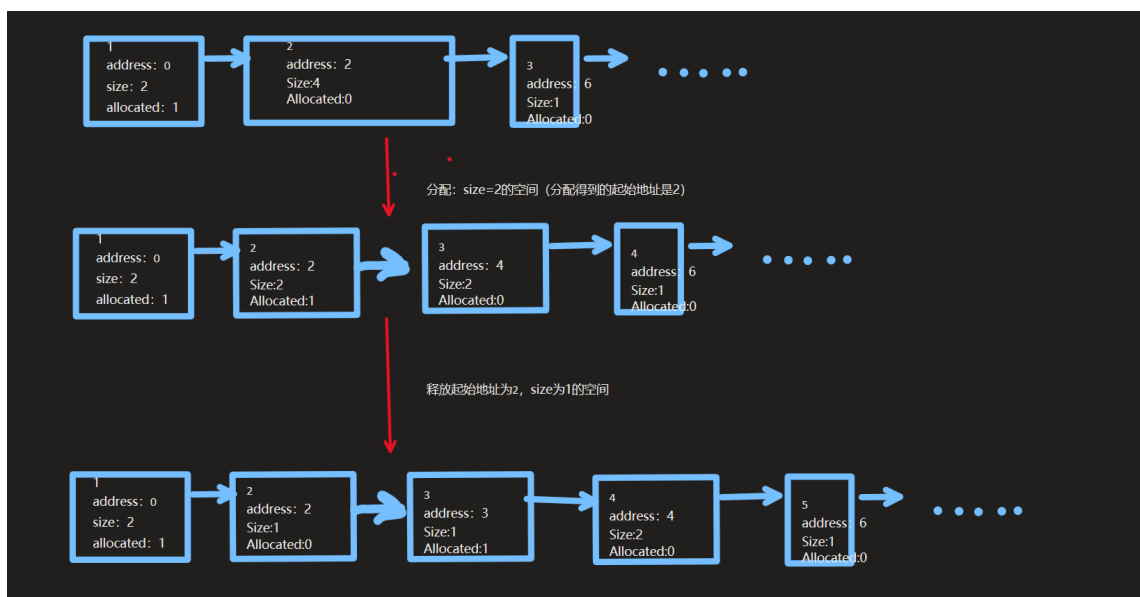
allocate 3
1_hole:
  startAddress:0 || is_allocate:1 || size:3
2_hole:
  startAddress:3 || is_allocate:0 || size:123

release 1
1_hole:
  startAddress:0 || is_allocate:0 || size:1
2_hole:
  startAddress:1 || is_allocate:1 || size:2
3_hole:
  startAddress:3 || is_allocate:0 || size:123

```

## 5.总结

- 注意：`resources` 中新的节点不能在直接在分配函数中定义生成，不然，节点将会是局部变量，离开函数后就会释放掉，节点的数据将会无法预测。所以一开始就初始化好一个 `item[100]` 全局变量，生成新节点时从其中获取。
- 补充图解：



# Assignment3

## 1.实验要求：

参照理论课上虚拟内存管理的页面置换算法如FIFO、LRU等，实现页面置换。

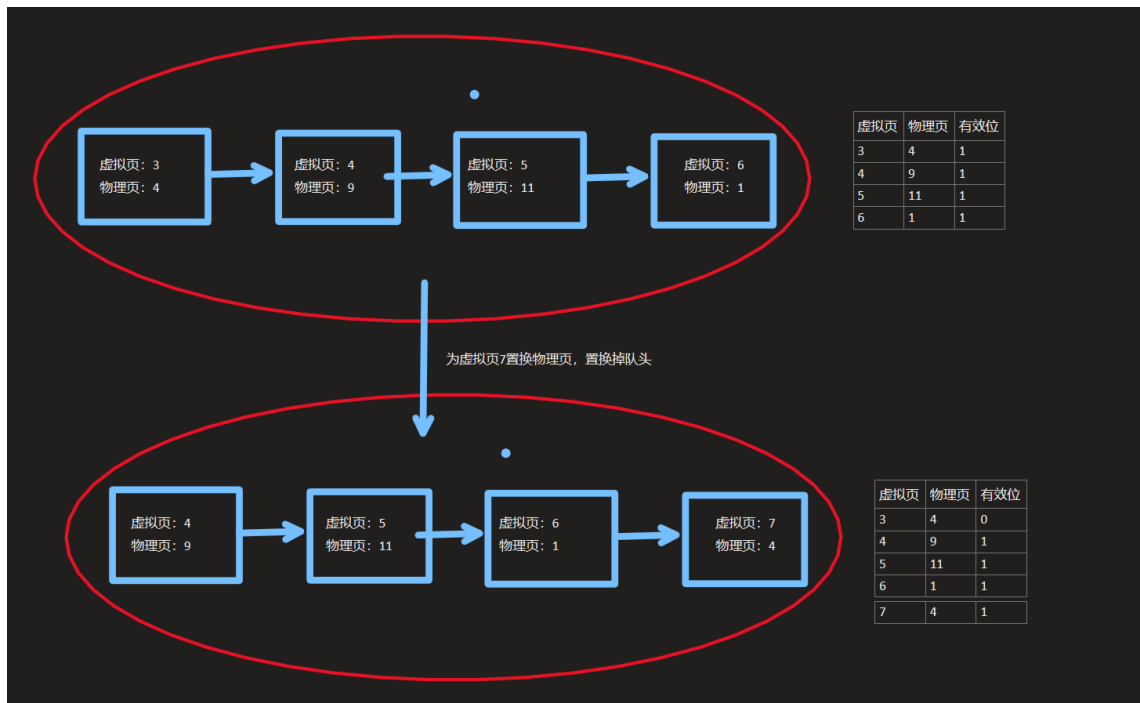
## 2.实验过程：

在内核的虚拟内存中实现空闲池版FIFO：

1. 创建：基于虚拟内存创建空闲池，空闲池的数据结构为用链表实现的队列（[为什么基于虚拟内存实现](#)）

2. 置换：从空闲池的队头处选择页面释放置换，置换后对应的物理页地址不变，该物理页仍然需要在空闲池中，所以将对应的物理页压入队尾。

3. 图解：



## 3.关键代码：

### 3.1创建：

- 数据结构：(见注释)

```
1 List FIFO;//FIFO空闲帧池，最大为66个帧，初始化的时候只需要给出需要空闲帧池的虚拟起始地址
2 struct ListItem//空闲帧池中的页
3 {
4     ListItem *previous;
5     ListItem *next;
6     int start_address;//物理页映射的虚拟页起始地址
7 };
8 class List
9 {
10 public:
11     ListItem head;
12 }
```

- 初始化：

选定一块连续的66页虚拟内存对应的物理页创建空闲帧池

```
1 void MemoryManager::init_FIFO(int start_address){//传进来的是虚拟地址
2     for (int i=0;i<66;i++){
3         item[i].start_address=start_address+PAGE_SIZE*i;
4         FIFO.push_back(&item[i]);
5     }
6     printf("initialize the free page pool successfully!\n the size:66
7     pages the start address:%x\n",start_address );
8 }
```

### 3.2 置换

释放队头虚拟页对应的物理页，并且置页表项无效（页表项的有效位在最低位）。置换后的页更新映射虚拟页，并将其从队头移动到队尾

```
1 void MemoryManager::displace_page(int count,int vaddr){//只是针对内核的，用户还没实现
2     for(int i=0;i<count;i++){
3         ListItem* tmp=FIFO.front();
4         releasePages_FIFO(KERNEL,tmp->start_address,1);
5         FIFO.pop_front();
6         FIFO.push_back(tmp);
7         printf("for displacing,a page(virtual start_address:%x
physical_address:%x) releases\n",tmp->start_address,vaddr2paddr(tmp-
>start_address));
8         tmp->start_address=vaddr;
9     }
10 }
11 void MemoryManager::releasePages_FIFO(enum AddressPoolType type, const int
virtualAddress, const int count)
12 {
13     int vaddr = virtualAddress;
14     int *pte;
15     for (int i = 0; i < count; ++i, vaddr += PAGE_SIZE)
16     {
17         // 第一步，对每一个虚拟页，释放为其分配的物理页
18         releasePhysicalPages(type, vaddr2paddr(vaddr), 1);
19
20         // 设置页表项为无效，防止释放后被再次使用
21         pte = (int *)toPTE(vaddr);
22         *pte = *pte&0xfffffffffe;
23     }
24 }
```

### 3.3 总置换流程

1. 申请到虚拟页后，逐页申请物理页（MemoryManager::allocatePages）

```
1     flag = false;
2     // 第二步：从物理地址池中分配一个物理页//由于物理页是不连续的，所以可以一个一个分
3     physicalPageAddress = allocatePhysicalPages(type, 1,vaddress);
4     if (physicalPageAddress)
5     {
6         //printf("allocate physical page 0x%x\n", physicalPageAddress);
7
8         // 第三步：为虚拟页建立项目录项和页表项，使虚拟页内的地址经过分页机制变换到物
理页内。
9         flag = connectPhysicalVirtualPage(vaddress,
physicalPageAddress);
10    }
```

2. 申请物理页时，遇到无空闲物理页的情况,进行页面置换，释放页再分配（MemoryManager::allocatePhysicalPages）

```

1     if (type == AddressPoolType::KERNEL)
2     {
3         start = kernelPhysical.allocate(count);
4         while(start==0&&count==1){//专门针对情况
5             displace_page(1,vaddr);//置换
6             start = kernelPhysical.allocate(count);//重新分配
7         }
8     }

```

## 4.实验结果：

### 1. 测试：

经过测试，内核的物理空间最多分配15969页

p1分配了15969页后，内核的物理空间将无空闲页

p2想要申请五页时，需要从空闲池置换，空闲池的物理页映射的虚拟页开始地址初始化0xc0100000，则p2会置换掉0xc0100000开始的五个虚拟页的物理页

```

1     memoryManager.init_FIFO(0xc0100000);
2     char *p1 = (char
*)memoryManager.allocatePages(AddressPoolType::KERNEL, 15969);
3     char *p2 = (char
*)memoryManager.allocatePages(AddressPoolType::KERNEL, 5);
4     printf("%x %x \n", p1,p2);

```

### 2. 结果：

置换了0xc0100000开始的五个虚拟页的物理页，和预测符合

```

initialize the free page pool successfully!
the size:66 pages  the start address:C0100000
for displacing,a page(virtual start_address:C0100000  physical_address:200000) r
leases
for displacing,a page(virtual start_address:C0101000  physical_address:201000) r
leases
for displacing,a page(virtual start_address:C0102000  physical_address:202000) r
leases
for displacing,a page(virtual start_address:C0103000  physical_address:203000) r
leases
for displacing,a page(virtual start_address:C0104000  physical_address:204000) r
leases
C0100000 C3F61000

```

## 5.总结：

- 由于物理内存中没有空闲帧，所以需要将挑选内存的部分物理页换出置换，将被置换的物理页对应的虚拟页的页表条目录为无效，并将被置换的物理页分配给其他虚拟页。如果在物理内存没有空闲帧的情况下申请了连续的多个虚拟页空间，由于虚拟页和物理页是分离的，虚拟页连续物理页可以不连续，所以可以在物理内存中置换出不连续的页，但是映射到连续的虚拟空间中。上述操作都是从物理内存的角度去考虑页面置换，为什么在实现的FIFO直接对已经分配好的虚拟页进行操作，而不是物理页？[跳转锚点](#)

- **合理性：**因为物理页与虚拟页是映射关系，所以可以通过对虚拟页的操作间接操作物理页。基于虚拟页建立空闲池，其实是对虚拟页映射的物理页建立空闲帧池。进行置换时，对空闲帧池中物理页的释放可以变成对空闲池中虚拟页的“释放”。但是需要注意对虚拟页的“释放”，是把和它对应的物理页解绑，先把物理页的内容写入磁盘，然后将页表项置为无效。

- **优点：**基于虚拟地址建立很快就能找到页表项，修改页表项，不然还需要根据物理地址遍历页表找到页表项修改更新。而且开启分页机制后，程序面对的是虚拟地址，这样操作起来更方便。
- **注意：**这里是将页表项有效位置为0而不是直接整个页表项清0。有效位置0代表虚拟地址还在，但是访问的时候会出现缺页错误，需要从磁盘调页，如果直接将页表项清0是程序释放了这段内存，虚拟内存也没有了。

每次置换后池内页的虚拟页地址会变化，物理页地址不变。由于是基于虚拟页地址进行操作的，所以置换后要把池内页的虚拟页地址进行更新，更新为物理页现在映射的虚拟页，保证池内空闲帧的不变

## 2. 实现中遇到的bug:

开始实验实现是基于给出的虚拟内存管理 `src/5` ,**在这里面给内核分配的物理内存和虚拟内存一样大。如果出现物理帧不够的情况，也意味着虚拟内存不够。**如果按照理论将物理页置换，保留虚拟页，将页表项置为无效，即使把整个空闲池释放掉，还是会由于虚拟空间不足没有办法分配空间。

改进：把虚拟页的页数设置的比物理页大

```
kernelPhysical.initialize(  
    (char *)kernelPhysicalBitMapStart,  
    kernelPages,  
    kernelPhysicalStartAddress);  
  
userPhysical.initialize(  
    (char *)userPhysicalBitMapStart,  
    userPages,  
    userPhysicalStartAddress);  
  
kernelVirtual.initialize(  
    (char *)kernelVirtualBitMapStart,  
    kernelPages,  
    KERNEL_VIRTUAL_START);
```

# Assignment4

## 1.实验要求:

- 结合代码分析虚拟页内存分配的三步过程和虚拟页内存释放。
- 构造测试例子来分析虚拟页内存管理的实现是否存在bug。如果存在，则尝试修复并再次测试。否则，结合测例简要分析虚拟页内存管理的实现的正确性。

## 2.实验过程:

(只是对内核管理的)



- 1.虚拟内存管理的初始化
- 2.虚拟内存页分配
- 3.虚拟内存页释放

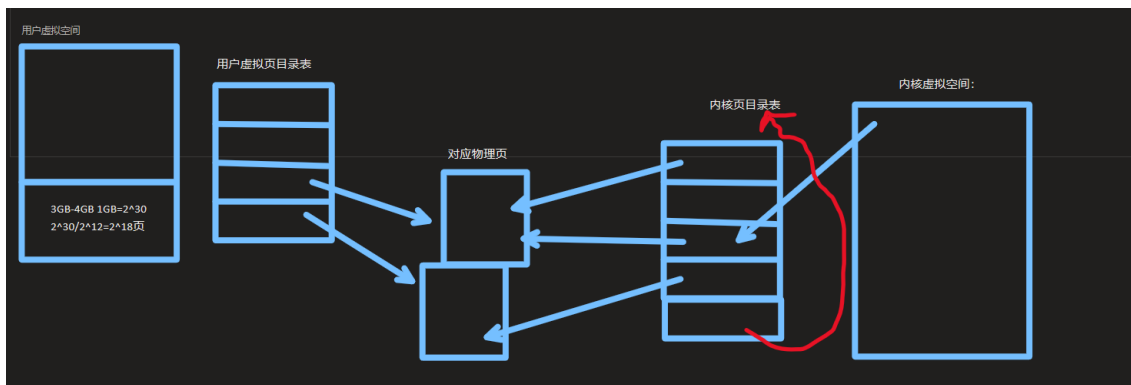
## 3.关键代码:

### 3.1虚拟内存管理的初始化

- 实现：建立内核的虚拟地址池（初始化：大小和内核的物理地址池一样大，起始地址为0xC0100000）
- 为什么起始地址为0xC100000?

将内核虚拟地址空间映射到用户虚拟地址空间3GB到4GB（用户进程通信的共享公共区域，由于需要告知内核，所以也需要在内核虚拟地址中映射）。为了所有进程都能共享到内核分配的页内存，则内核虚拟空间起始地址不是从0x100000开始，而是要同样提高到3GB到4GB的范围内。则开始地址为0xC0100000

- 注意：由于内核的地址映射是直接映射的，那么3GB-4GB的虚拟地址空间在页目录表中涵盖第768-第1023个页目录项（一个页目录项能涵盖4MB的虚拟空间映射 $3 * 2^{30} / 2^{22} = 3 * 2^8 = 768$ ）。一共256个页目录项：前255个页目录项映射255个页表（和页目录表中0-255的页目录项映射的内容一致（和内核真正的虚拟地址对应）），第256个页目录项映射页目录表（页目录表也是一个页表）
- 图解：



```
1 kernelVirtual.initialize(  
2     (char *)kernelVirtualBitMapStart,  
3     kernelPages,  
4     KERNEL_VIRTUAL_START);
```

### 3.2页内存分配

注意开启分页机制后程序看到的是虚拟地址

- 第一步：从虚拟地址池中分配若干连续的虚拟页

根据申请的空间类型，直接调用对应虚拟地址池的分配函数，返回分配到的虚拟地址

```
1 // 第一步：从虚拟地址池中分配若干虚拟页  
2 int virtualAddress = allocateVirtualPages(type, count);  
3 if (!virtualAddress)  
4 {  
5     return 0;  
6 }
```

```

7  int MemoryManager::allocateVirtualPages(enum AddressPoolType type, const
   int count)
8  {
9      int start = -1;
10
11     if (type == AddressPoolType::KERNEL)
12     { //只实现了内核的
13         start = kernelVirtual.allocate(count);
14     }
15     //从虚拟内核地址池里面分配连续地址空间
16     return (start == -1) ? 0 : start;
17 }

```

- 第二步：为每一个虚拟页，从物理地址池分配1页

原理：虚拟地址连续的，对应的物理地址可以不连续，体现优点避免外部碎片

根据申请的空间类型，直接调用对应物理地址池的分配函数，返回分配到的物理地址

```

1      flag = false;
2      // 第二步：从物理地址池中分配一个物理页//由于物理页是不连续的，所以可以一个一个分
   physicalPageAddress = allocatePhysicalPages(type, 1);
3  int MemoryManager::allocatePhysicalPages(enum AddressPoolType type,
   const int count)
4  {
5      int start = -1;
6
7      if (type == AddressPoolType::KERNEL)
8      {
9          start = kernelPhysical.allocate(count);
10     }
11     else if (type == AddressPoolType::USER)
12     {
13         start = userPhysical.allocate(count);
14     }
15
16     return (start == -1) ? 0 : start;
17 }
18

```

- 第三步：为虚拟页建立页目录项和页表项，使虚拟页内的地址经过分页机制变换到物理页内

1.如果分配过程中有一个虚拟页分配物理页失败了，要释放掉前i个已经分配的物理页，申请分配内存失败

```

1      if (physicalPageAddress)
2      {
3          // 第三步：为虚拟页建立页目录项和页表项，使虚拟页内的地址经过分页机制变换到物理页内。
4          flag = connectPhysicalVirtualPage(vaddress,
   physicalPageAddress);
5      }
6      else
7      {
8          flag = false;
9      }
10

```

```

11 // 分配失败，释放前面已经分配的虚拟页和物理页表
12 if (!flag)
13 {
14     // 前i个页表已经指定了物理页
15     releasePages(type, virtualAddress, i);
16     // 剩余的页表未指定物理页
17     releaseVirtualPages(type, virtualAddress + i * PAGE_SIZE, count -
18 i);
19     return 0;
20 }

```

## 2.为虚拟页建立页目录项和页表项

注意页表和页目录表都是4KB，所以它们的起始地址都是高二十位不为0，低十二位都为0

(1) 总：

```

1 bool MemoryManager::connectPhysicalVirtualPage(const int virtualAddress,
2 const int physicalPageAddress)
3 {
4     // 计算虚拟地址对应的页目录项和页表项
5     int *pde = (int *)toPDE(virtualAddress);
6     int *pte = (int *)toPTE(virtualAddress);
7     // 页目录项无对应的页表，先分配一个页表
8     if(!(*pde & 0x00000001)) //看页目录项的最后一个有效位是否有效
9     {
10         // 从内核物理地址空间中分配一个页表
11         int page = allocatePhysicalPages(AddressPoolType::KERNEL, 1);
12         if (!page)
13             return false;
14
15         // 使页目录项指向页表
16         *pde = page | 0x7; //分了个页表，写入页目录
17         // 初始化页表
18         char *pagePtr = (char *)(((int)pte) & 0xfffff000); //取出页表项地址的高二
19 十位也就是物理页表地址
20         memset(pagePtr, 0, PAGE_SIZE);
21     }
22     // 使页表项指向物理页
23     *pte = physicalPageAddress | 0x7; //在页表中建立
24
25     return true;
26 }

```

(2) 根据虚拟页的地址计算页目录项和页表项的虚拟地址：

注意：虚拟地址：高十位是对应页目录项在页目录表中的偏移，中十位是对应页表项在页表中的偏移，低十二位是在页内偏移

构造页目录项的虚拟地址：低十二位在页内的偏移：页目录项在页目录表内偏移；中十位：页表项在页表内的偏移：指向页目录表的页目录项在页目录表内的偏移（指向页目录表的页目录项是第1023个页目录项）；高十位：页目录表项在页目录表中的偏移：指向页目录表的页目录项在页目录表中的偏移

构造页表项的虚拟地址: 低十二位在页内偏移: 页目录项在页表内偏移; 中十位: 页表项在页表内的偏移: 指向页表的页目录项在页目录中的偏移; 高十位: 页目录表项在页目录内的偏移: 指向页目录表的页目录项在在页目录表中的偏移

```
1  int MemoryManager::toPDE(const int virtualAddress)//构造页目录项的虚拟的地址
2  {
3      return (0xfffff000 + (((virtualAddress & 0xffc00000) >> 22) * 4));
4  }//(virtualAddress & 0xffc00000)>> 22 提取出虚拟地址的高10位, 并且挪到低10位, 得到的是在页目录中的排序
5  /*4说明这个对应的页目录项在页目录的偏移位置//
6  //现在要构造的是对应页表项的虚拟地址(程序只能看虚拟地址), 低12位为页内偏移(就是页目录项在页目录的偏移(页目录也是页表))
7  //中10位为页在页表内的偏移, 在这里就是表示指向页目录表的页表项在页表内的偏移, (页目录表页的映射在页目录表的最后一项), 则为111
8  //高10位为页表项的映射在页目录表的偏移位置, 也是页目录的最后一项, 也是1111111
9  /
10 int MemoryManager::toPTE(const int virtualAddress)//构造页表项的虚拟地址
11 {
12     return (0xffc00000 + ((virtualAddress & 0xffc00000) >> 10) +
13         (((virtualAddress & 0x003ff000) >> 12) * 4));
14 }//把虚拟地址的中10位提取出来(((virtualAddress & 0x003ff000) >> 12) * 4) 这个是虚拟地址在页表中的偏移(低10位), 下面把页表项地址构造成虚拟地址
15 //中10位为页表项在页表中的偏移, 这里表示为对应页表在页目录表中的偏移,
16 //高10位为页目录项在页目录中的偏移, 这里表示为页目录表的映射项在页目录中的偏移 111111
```

### 3.3页内存释放

总: 先将虚拟页地址转为物理页地址到物理地址中释放, 再将对应的页表项抹0, 最后取虚拟地址池中释放虚拟页

```
1  void MemoryManager::releasePages(enum AddressPoolType type, const int virtualAddress, const int count)
2  {
3      int vaddr = virtualAddress;
4      int *pte;
5      for (int i = 0; i < count; ++i, vaddr += PAGE_SIZE)
6      {
7          // 第一步, 对每一个虚拟页, 释放为其分配的物理页
8          releasePhysicalPages(type, vaddr2paddr(vaddr), 1);
9
10         // 设置页表项为不存在, 防止释放后被再次使用
11         pte = (int *)toPTE(vaddr);
12         *pte = 0;
13     }
14
15     // 第二步, 释放虚拟页
16     releaseVirtualPages(type, virtualAddress, count);
17 }
```

虚拟地址到物理地址的转换: 根据高十位到页目录表中找到页表项得到页表地址, 根据中10位和页表地址到页表中找到页表项得到物理页地址, 物理页地址加上低十二位页内偏移得到物理地址

```

1  int MemoryManager::vaddr2paddr(int vaddr)
2  {
3      int *pte = (int *)toPTE(vaddr);
4      int page = (*pte) & 0xfffff000;
5      int offset = vaddr & 0xfff;
6      return (page + offset);
7  }

```

## 4.实验结果:

- 默认测试用例结果正常:

```

1  char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL,
2  100);
3      printf("allocate 100 pages for p1,address:%x\n",p1 );
4      char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL,
5  10);
6      printf("allocate 10 pages for p2,address:%x\n",p2 );
7      char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL,
8  100);
9      printf("allocate 100 pages for p3,address:%x\n",p3 );
10
11     memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
12     printf("release 10 pages from p2\n");
13     p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
14     printf("allocate 100 pages for p2,address:%x\n",p2 );
15     // printf("%x\n", p2);
16
17     p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
18     printf("allocate 10 pages for p2,address:%x\n",p2 );

```

成功打印关于地址池的相关信息。为p1分配100页，起始地址即为定义的虚拟空间起始地址 0xc0100000，终止地址为 0xc0164000(400kb是0x64000)；为p2分配10页起始地址为0xc0164000，终止地址为0xc016E000；为p3分配100页起始地址为0xc016E000，终止地址为0xc01D2000。

释放p2再给它分配100页时，p1和p3之间的10页不够发配，只能在p3后面分配。

再给p2分配10页时，分配的就是p1和p3之间空闲的10页

```

Booting from Hard Disk...
open page mechanism
total memory: 133038080 bytes ( 126 MB )
kernel pool
    start address: 0x2000000
    total pages: 15984 ( 62 MB )
    bitmap start address: 0x10000
user pool
    start address: 0x4070000
    total pages: 15984 ( 62 MB )
    bit map start address: 0x107CE
kernel virtual pool
    start address: 0xC0100000
    total pages: 15984 ( 62 MB )
    bit map start address: 0x10F9C
allocate 100 pages for p1,address:C0100000
allocate 10 pages for p2,address:C0164000
allocate 100 pages for p3,address:C016E000
release 10 pages from p2
allocate 100 pages for p2,address:C01D2000
allocate 10 pages for p2,address:C0164000

```

- bug:

设置：给bitmap的set函数中加了循环耗时

```

1 void BitMap::set(const int index, const bool status)
2 {
3     //加个循环,耗时
4     int pos = index / 8;
5     int offset = index % 8;
6     int delay = 0xffffffff;
7     while (delay)
8         --delay;
9     // 清0
10    bitmap[pos] = bitmap[pos] & ~(1 << offset);
11
12    // 置1
13    if (status)
14    {
15        bitmap[pos] = bitmap[pos] | (1 << offset);
16    }
17 }

```

触发bug：同时给两个线程分配100页

```

1 void second_thread(void *arg)
2 {
3     char *p2 = (char
4     *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
5     printf("allocate 100 pages for p2,address:%x\n",p2);
6 }

```

```

7 void first_thread(void *arg)
8 {
9     programManager.executeThread(second_thread, nullptr, "second
10 thread", 2);
11     char *p1 = (char
12 *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
13     printf("allocate 100 pages for p1,address:%x\n",p1 );
14     asm_halt();
15 }

```

问题：两个线程分配到的地址虽然不同，但是根据计算，两个线程分配到的内核空间重叠：按照计算两个起始地址应该是0xc0100000和0xc0164000，但是会出现一个地址是0xc010A000。

```

Booting from Hard Disk...
open page mechanism
total memory: 133038080 bytes ( 126 MB )
kernel pool
    start address: 0x200000
    total pages: 15984 ( 62 MB )
    bitmap start address: 0x10000
user pool
    start address: 0x4070000
    total pages: 15984 ( 62 MB )
    bit map start address: 0x107CE
kernel virtual pool
    start address: 0xC0100000
    total pages: 15984 ( 62 MB )
    bit map start address: 0x10F9C
allocate 100 pages for p2,address:C010A000
allocate 100 pages for p1,address:C0100000

```

出现原因：由于寻找和分配空间的过程并不是原子的，而且进程调度是时间片调度。这可能出现在为某个进程分配空间中途切换了其他进程。其他进程打断了上一个进程的虚拟空间分配的连续性，造成它们分配到的虚拟空间是重叠的

## 5.总结:

问题：为什么内核的物理页不能完全分配，实际上可分配的比理论少