操作系统实验七实验报告

个人信息

- 姓名：李浩辉
- 学号：21307018

实验要求

- 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
- 实现**动态分区算法**如first-fit, best-fit等
- 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或验证其正确性
- 实现**页面置换**如FIFO、LRU等
  （调整了assignment3和assignment4顺序）

实验过程

Assignment 1

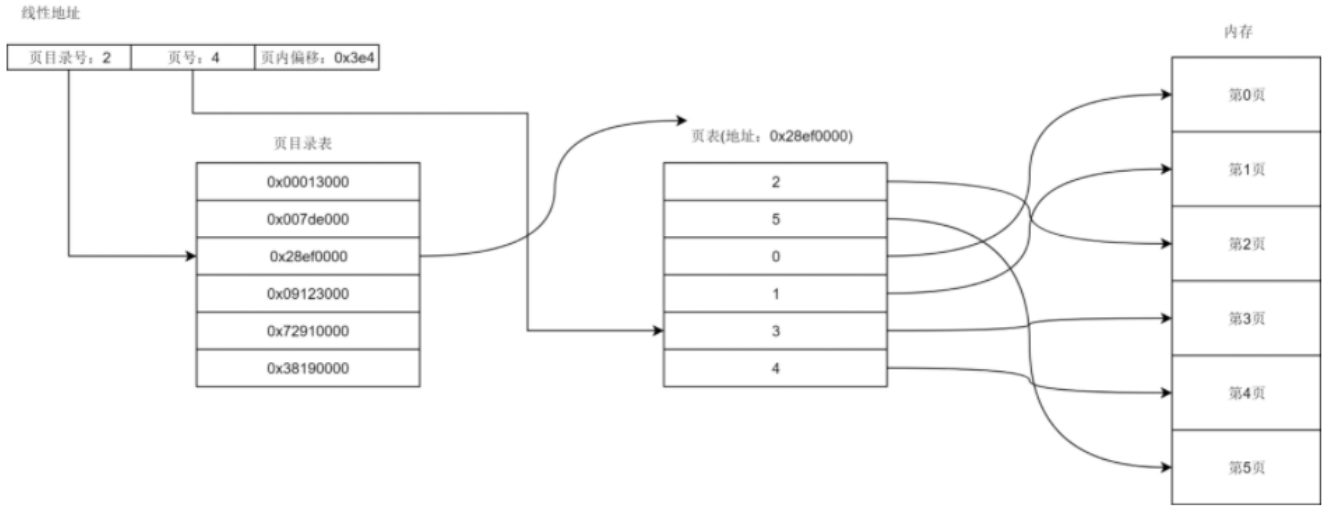
实验要求

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

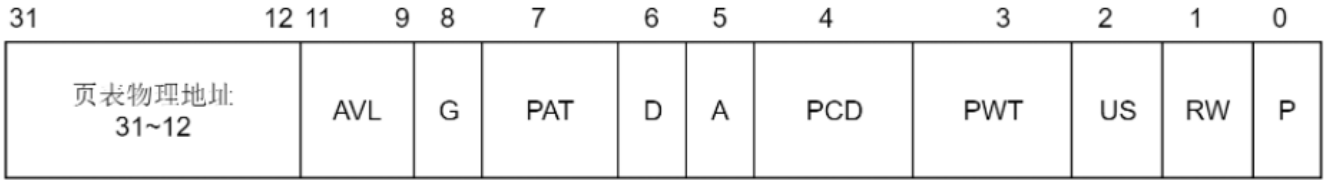
实验原理

在开启分页之前，我们首先进行了内存的探查，**位图**和**地址池**的实现。
简单来说，就是我们接下来通过地址池可以知道哪些地址也被分配，那些地址是空闲的，方便我们进行后续的分页。

下面介绍一下**二级页表**的原理和实现
二级页表寻址流程图



页目录项/页表项结构



一个线性地址拆分为3部分

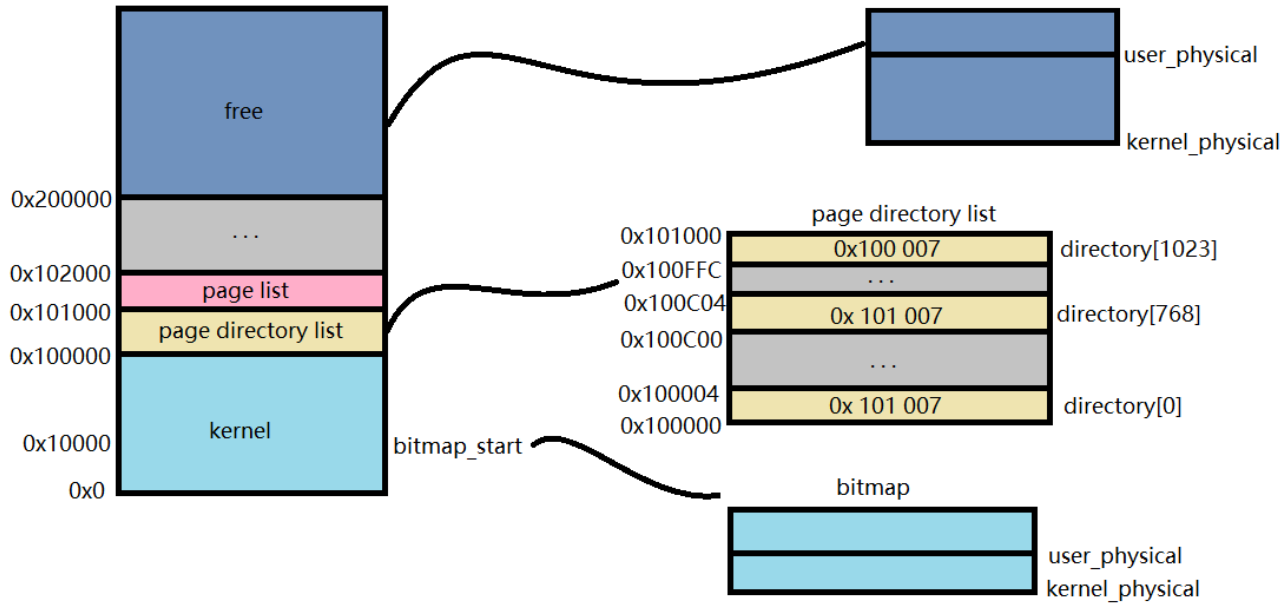
- 31-22位表示**页目录表中的序号**，共10bit，可以表示 $2^{10} = 1K$ 项
- 21-12位表示**页表中的序号**，共10bit，可以表示 $2^{10} = 1K$ 项
- 11-0位表示**页内偏移**，共12bit，可以表示 $2^{12} = 4K$ 项

开启分页只需要**将页目录表输入到 $cr3$，将 $cr0$ 的PG位（31）设置为1即可**打开

```
asm_init_page_reg:
    push ebp
    mov ebp, esp
    push eax
    mov eax, [ebp + 4 * 2]
    mov cr3, eax ; 放入页目录表地址
    mov eax, cr0
    or eax, 0x80000000
    mov cr0, eax            ; 置PG=1，开启分页机制
    pop eax
```

```
    pop ebp
    ret
```

下面讲下开启分页之前我们的安排，如图所示



- 我们将**位图**安排在 `0x10000` ，这里用来指示内存分配情况
- 我们的**内核空间**一共会占据1MB，我们把他放在了 `0x0` - `0x1000000`
- `0x200000` 以上的地址就是我们**空闲可分配的空间**，我们将其划分成了内核，用户两部分
- **页目录表**放在了 `0x100000` - `0x101000` 即内核空间往上的4KB内存
- 预留出了 `0x101000` - `0x200000` 的空间，共计255个页表
- `directory[0]` 指向第一个页表， `directory[768]` 指向第一个页表， `directory[1023]` 指向页目录表
- 安排了**第一个页表**，其中它前256项安排成前二十位从0到256依次自增1，这样将会**让线性地址0-1MB映射到实际地址0-1MB**

实验过程

位图以及地址池代码逻辑比较简单，只是简单的**记录分配情况**，不作赘述。
同时关于分页前的安排我在原理上已经写得比较详细，我们来看下过程。
首先写一段简单的**分配内存以及释放内存**

```
void test(void *arg)
{
    char* p1=(char*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL,
1);
```

```
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL,(int)p1,1);
    asm_halt();
}
```

分配内存

直接 `make debug` 通过设置断点跳转

```
18      void test(void *arg)
19      {
B+> 20          char* p1=(char*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
21          memoryManager.releasePhysicalPages(AddressPoolType::KERNEL,(int)p1,1);
22          asm_halt();
23      }
```

我们 `step` 进入 `allocatePhysicalPages` ，并且我们即将进入地址池的分配函数

```
终端
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
../src/kernel/memory.cpp
60                  userPhysicalBitMapStart);
61      }
62
63      int MemoryManager::allocatePhysicalPages(enum AddressPoolType type, const int count)
64      {
65          int start = -1;
66
67          if (type == AddressPoolType::KERNEL)
68          {
>69              start = kernelPhysical.allocate(count);
70          }
71          else if (type == AddressPoolType::USER)
72          {
remote Thread 1 In: MemoryManager::allocatePhysicalPages                L69   PC: 0x2135b
(gdb) c
Continuing.

Breakpoint 2, test (arg=0x0) at ../src/kernel/setup.cpp:20
(gdb) s
MemoryManager::allocatePhysicalPages (this=0x32a34 <memoryManager>,
    type=KERNEL, count=1) at ../src/kernel/memory.cpp:65
(gdb) n
(gdb) n
```

地址池分配如图，即从类中的位图 `bitmap resources` 去看哪个位置空闲有内存

至于位图的代码就不进入了，**位图的start地址在0-1MB因为我们的分页设置不会受到影响，所以只需要简单的数组设置1或者0即可**

```
15      // 从地址池中分配count个连续页
16      int AddressPool::allocate(const int count)
17      {
>18          int start = resources.allocate(count);
19          return (start == -1) ? -1 : (start * PAGE_SIZE + startAddress);
20      }
```

最后，地址池将会计算并返回地址

```
15      // 从地址池中分配count个连续页
16      int AddressPool::allocate(const int count)
17      {
18          int start = resources.allocate(count);
>19         return (start == -1) ? -1 : (start * PAGE_SIZE + startAddress);
20      }
21
22      // 释放若干页的空间
23      void AddressPool::release(const int address, const int amount)
24      {
25          resources.release((address - startAddress) / PAGE_SIZE, amount);
```

```
remote Thread 1 In: AddressPool::allocate                                L19   PC: 0x217cd
(gdb) s
MemoryManager::allocatePhysicalPages (this=0x32a34 <memoryManager>,
    type=KERNEL, count=1) at ../src/kernel/memory.cpp:65
(gdb) n
(gdb) n
(gdb) s
AddressPool::allocate (this=0x32a38 <memoryManager+4>, count=1) at ../src/utils/address_pool.cpp:18
(gdb) n
(gdb) p start
$1 = 0
(gdb)
```

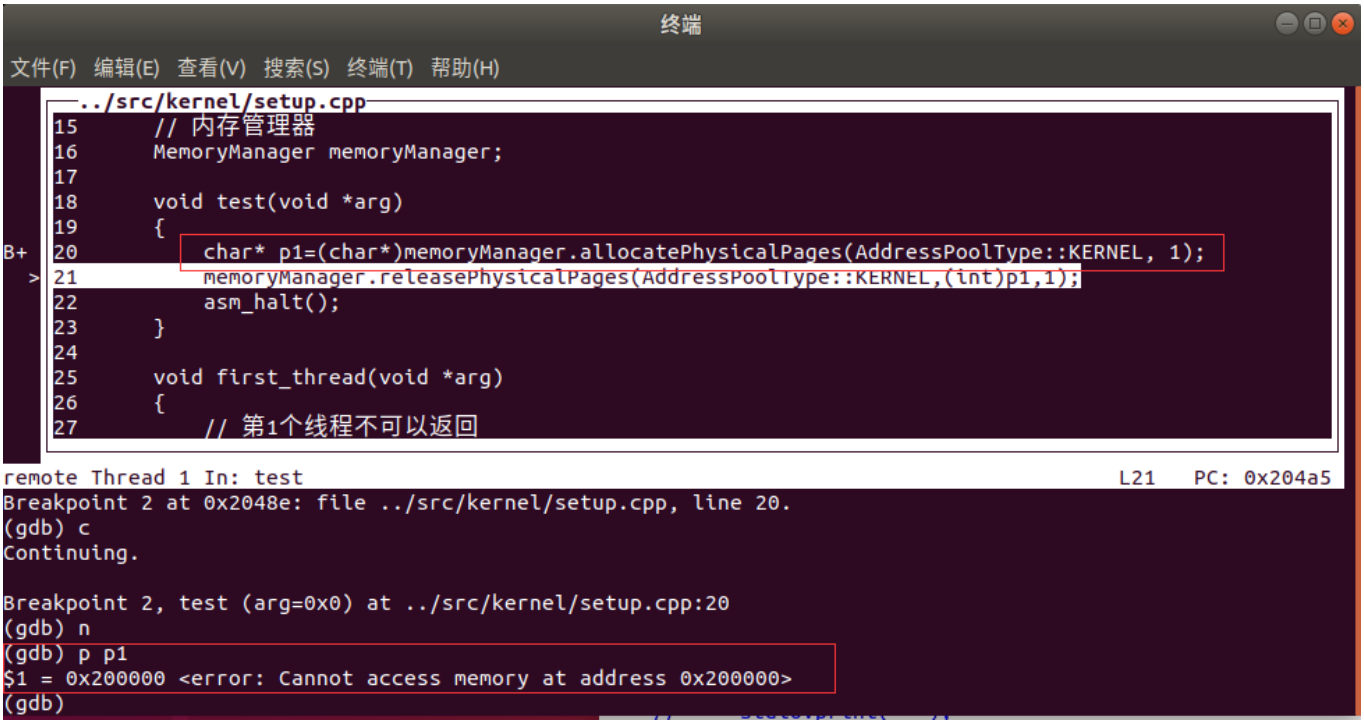
接着 `allocate` 函数也是返回同样的地址

```
终端
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
../src/kernel/memory.cpp
68          {
69              start = kernelPhysical.allocate(count);
70          }
71          else if (type == AddressPoolType::USER)
72          {
73              start = userPhysical.allocate(count);
74          }
75
>76          return (start == -1) ? 0 : start;
```

看下结果，`allocatePhysicalPages` 返回了地址 `0x200000`

这里有一个提示`Cannot access memory`，**因为我们得到的是一个实际地址，但是分页机制已经开**

启，我们自然就找不到这个地址！



释放内存

我们接着上面进入`releasePhysicalPages`
和分配内存一样，我们将会调用地址池的`release`函数

```
79      void MemoryManager::releasePhysicalPages(enum AddressPoolType type, const int paddr, const int
80      {
81          if (type == AddressPoolType::KERNEL)
82          {
83              kernelPhysical.release(paddr, count);
84          }
```

函数里面调用位图的释放函数，位图代码一样不展示了，原理一样就是**设置数组**

```
22      // 释放若干页的空间
23      void AddressPool::release(const int address, const int amount)
24      {
25          resources.release((address - startAddress) / PAGE_SIZE, amount);
26      }^?
27
```

接着依次退出即完成

在这里发下牢骚：
这里的二级分页算是完成了吗？严格意义上来说还没有！二级分页机制没有虚拟内存机制就是**把自己的仓库装了一把没有钥匙的锁。**
我们**分配的内存根本用不了**：因为我们还没有建立好对应的虚拟内存机制，但是返回的却是一个实际地址，我们无论是对他进行赋值还是访问，操作系统都会当他虚拟内存来寻址进而产生错误！
(问题解决在Assignment3中)

Assignment 2

实验要求

实现**动态分区算法**如first-fit, best-fit等

实验原理

$first\_fit$：从头开始遍历内存空间，遍历过程中如果发现有可容纳的连续内存空间则分配。即**分配第一个可容纳的内存块**
$best\_fit$：遍历内存空间，找到所有空闲内存块中**可容纳并且最小的内存块**来分配。

这里的分配算法我们只需要修改位图 `int BitMap::allocate(const int count)` 即可

实验过程

先看下 `first_fit` 的代码
逻辑比较简单，从前往后遍历：遇到空闲资源则开启新的循环检测，如果合适则分配；遇到占用资源或者不合适继续往前遍历

```
int BitMap::first_fit_allocate(const int count)
{
    if (count == 0)
        return -1;

    int index, empty, start;

    index = 0;
    while (index < length)
    {
        // 越过已经分配的资源
        while (index < length && get(index))
            ++index;

        // 不存在连续的count个资源
        if (index == length)
            return -1;

        // 找到1个未分配的资源
        // 检查是否存在从index开始的连续count个资源
        empty = 0;
        start = index;
        while ((index < length) && (!get(index)) && (empty < count))
        {
            ++empty;
            ++index;
        }
```

```
        // 存在连续的count个资源
        if (empty == count)
        {
            for (int i = 0; i < count; ++i)
            {
                set(start + i, true);
            }

            return start;
        }
    }

    return -1;
}
```

接着是`best_fit`的代码
其实只是在`first_fit`基础上增加一个`min_num`以及`min_index`遍历过程中记录空闲块的大小和位置即可

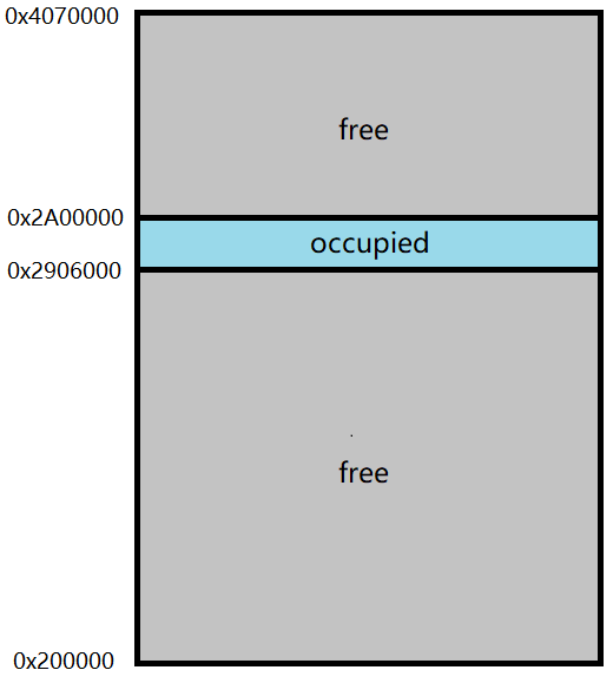
```
int BitMap::best_fit_allocate(const int count)
{
    if (count == 0)
        return -1;
    int index, empty, start;
    int min_index=-1;
    int min_num=length+1;
    index = 0;
    while (index < length)
    {
        // 越过已经分配的资源
        while (index < length && get(index))
            ++index;
        // 不存在连续的count个资源
        if (index == length)
            return -1;
        // 找到1个未分配的资源
        // 检查是否存在从index开始的连续count个资源
        empty = 0;
        start = index;
        while ((index < length) && (!get(index)))
        {
            ++empty;
            ++index;
        }
        // 存在连续的count个资源
```

```
        if (empty >= count&&empty<min_num)
        {
            min_num=empty;
            min_index=start;
        }
    }
    if(min_index!=-1)
    {
        for (int i = 0; i < count; ++i)
        {
            set(min_index + i, true);
        }
        return min_index;
    }
    return -1;
}
```

我们接下来来验证一下
内核空间为 `0x200000` 到 `0x4070000`
创建一个 $9990 * 4KB$，无论哪种算法都会占据 `0x200000` 到 `0x2906000`
继续创建 $250 * 4KB$，同样无论哪种都将继续占据 `0x2906000` 到 `0x2A00000`
然后我们释放掉p1即 `0x200000` 到 `0x2906000`

```
void test(void *arg)
{
    char* p1=(char*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL,
9990);
    char* p2=(char*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL,
250);
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL,(int)p1,9990);
    char* p3=(char*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL,
250);
    printf("%x\n%x\n%x\n",p1,p2,p3);
    asm_halt();
}
```

我们的 `Kernel_physical` 内存空间将会如下图所示



我们使用 `first_fit` 来测试结果如图
即最后p3占据了 `0x200000` 开始的空间，与我们的预期相同即找第一个合适的块

```
// 从地址池中分配count个连续页
int AddressPool::allocate(const int count)
{
    int start = resources.first_fit_allocate(count);
    return (start == -1) ? -1 : (start * PAGE_SIZE + startAddress);
}
```

```
200000
2906000
200000
```

改用 `best_fit` 进行测试
则会占据 `0x2A00000` 开始的内存块，与预期相同，找较小的合适块

```
int AddressPool::allocate(const int count)
{
    int start = resources.best_fit_allocate(count);
    return (start == -1) ? -1 : (start * PAGE_SIZE + startAddress);
}
```

```
200000
2906000
2A00000
```

Assignment 3

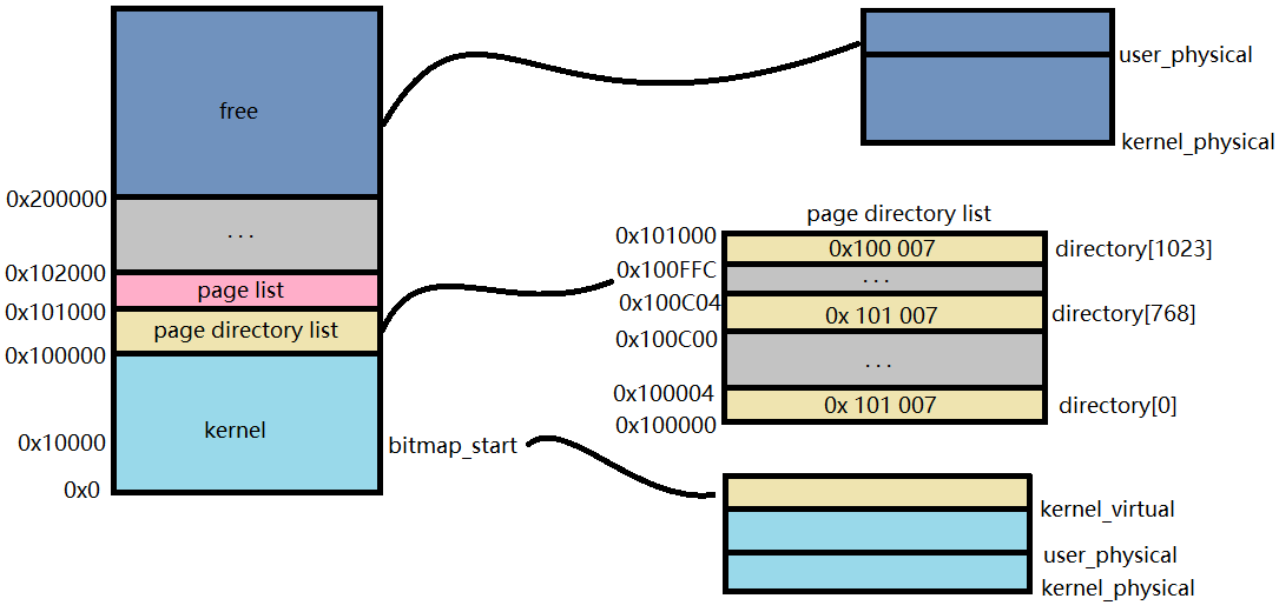
实验要求

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或验证其正确性

实验原理

前面Assignment 1中我们实现二级分页后就出现了问题——**我们申请的内存没法使用。**
原因就是我们**开启分页后，操作一个地址要用虚拟地址进行，却没有对应的虚拟页机制来操作对应的实际地址。**
我们的内存结构大体不变，但是位图多增加了kernel_virtual用来指示分配的内核虚拟地址，同时定义了新的宏

```
#define KERNEL_VIRTUAL_START 0xc0100000
```



在这里我们可以说明白**预留的内存**和**页目录表安排**的目的了：

- 我们希望有一块**所有进程（包括内核）共用的地方。**于是我们进行了个小设计：让kernel_virtual从 `0xc0100000` 开始分配，目的是为了他分配的页空间虚拟地址仅在3-4GB之间，同时任何进程3GB-4GB部分页目录表均以内核页目录表为准。
  （这样设计会有很多特别的事情，后面讨论）
- `directory[1023]` **指向页目录表本身**的目的：我们后续**建立虚拟地址与实际地址相关联**的时候，需要找到pde，pte即页目录项/页表项所在的虚拟地址。这样定义会带来方便（后面再分析）。

实验过程

内存分配

在这里分析一下内存分配的代码三步骤

- **分配虚拟页**
- **分配物理页**
- **建立联系**
  分配物理页和虚拟页具体就不做赘述了，其中没有涉及访存的地方，仅是更改数组和传递信息的功能，大致流程：
  分配函数 -> 地址池引用分配函数 -> 位图更改数组，返回地址

```
int MemoryManager::allocatePages(enum AddressPoolType type, const int count)
{
    // 第一步：从虚拟地址池中分配若干虚拟页
    int virtualAddress = allocateVirtualPages(type, count);
    if (!virtualAddress)
    {
        return 0;
    }
    bool flag;
    int physicalPageAddress;
    int vaddress = virtualAddress;
    // 依次为每一个虚拟页指定物理页
    for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE)
    {
        flag = false;
        // 第二步：从物理地址池中分配一个物理页
        physicalPageAddress = allocatePhysicalPages(type, 1);
        if (physicalPageAddress)
        {
            // 第三步：为虚拟页建立页目录项和页表项
            flag = connectPhysicalVirtualPage(vaddress, physicalPageAddress);
        }
        else
        {
            flag = false;
        }
        // 分配失败，释放前面已经分配的虚拟页和物理页表
        if (!flag)
        {
            // 前i个页表已经指定了物理页
            releasePages(type, virtualAddress, i);
            // 剩余的页表未指定物理页
            releaseVirtualPages(type, virtualAddress + i * PAGE_SIZE, count - i);
            return 0;
```

```
        }
    }
    return virtualAddress;
}
```

在分析如何建立联系之前，我们先弄懂如何构建pte,pde（页目录项/页表项的虚拟地址）
其实就一个思想：**页目录表当作特殊的页表，页表当作特殊的页**

- pde:
  页目录表 -> 页目录表 -> 页目录表 -> 页目录号
- pte:
  页目录表 -> 页目录表 -> 页表 -> 页表号

```
int toPDE(const int virtualAddress)
{
    return (0xfffff000 + (((virtualAddress & 0xffc00000) >> 22) * 4));
}
int toPTE(const int virtualAddress)
{
    return (0xffc00000 + ((virtualAddress & 0xffc00000) >> 10) + (((virtualAddress
& 0x003ff000) >> 12) * 4));
}
```

分析一下我们**如何建立联系**：

- 得到pde,pte
- 看页表是否已经建立，未建立则分配物理页并修改页目录项
- 修改页表项为物理地址
  这里有一点很奇妙的地方，pte我们是先计算出来的，它是页表的虚拟地址。**然而我们在这个函数运行过程中我们没有修改它的值，它的实际地址有可能会变化。**

```
bool MemoryManager::connectPhysicalVirtualPage(const int virtualAddress, const int
physicalPageAddress)
{
    // 计算页目录表/页表的虚拟地址
    int *pde = (int *)toPDE(virtualAddress);
    int *pte = (int *)toPTE(virtualAddress);
    // 页目录项无对应的页表，先分配一个页表
    if(!(*pde & 0x00000001))
    {
        // 从内核物理地址空间中分配一个页表
        int page = allocatePhysicalPages(AddressPoolType::KERNEL, 1);
        if (!page)
            return false;
```

```
        // 使页目录项指向页表
        *pde = page | 0x7;
        // 初始化页表
        char *pagePtr = (char *)(((int)pte) & 0xfffff000);
        memset(pagePtr, 0, PAGE_SIZE);
    }
    // 使页表项指向物理页
    *pte = physicalPageAddress | 0x7;
    return true;
}
```

内存释放

逻辑比较简单，**先释放物理页（提前计算物理地址），再释放虚拟页，最后设置页表项为空即可**，不作赘述

```
void MemoryManager::releasePages(enum AddressPoolType type, const int
virtualAddress, const int count)
{
    int vaddr = virtualAddress;
    int *pte, *pde;
    bool flag;
    const int ENTRY_NUM = PAGE_SIZE / sizeof(int);
    for (int i = 0; i < count; ++i, vaddr += PAGE_SIZE)
    {
        releasePhysicalPages(type, vaddr2paddr(vaddr), 1);
        // 设置页表项为不存在，防止释放后被再次使用
        pte = (int *)toPTE(vaddr);
        *pte = 0;
    }
    releaseVirtualPages(type, virtualAddress, count);
}
```

示例

简单一段代码

```
void first_thread(void *arg)
{
    char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
    char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
    char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
    printf("%x %x %x\n", p1, p2, p3);
    memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
    p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
```

```
    printf("%x\n", p2);
    p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
    printf("%x\n", p2);
    asm_halt();
}
```

我们输出的是虚拟地址，从虚拟地址上看与我们预期一致

```
C0100000 C0164000 C016E000
C01D2000
C0164000
```

通过 `make debug` 来看具体的物理地址是否符合预期

```
终端
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
../src/kernel/memory.cpp
179         // 依次为每一个虚拟页指定物理页
180         for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE)
181         {
182             flag = false;
183             // 第二步：从物理地址池中分配一个物理页
184             physicalPageAddress = allocatePhysicalPages(type, 1);
> 185           if (physicalPageAddress)
186             {
187                 //printf("allocate physical page 0x%x\n", physicalPageA
188
189                 // 第三步：为虚拟页建立页目录项和页表
190                 flag = connectPhysicalVirtualPage(vaddress, physicalPag
191             }
remote Thread 1 In: MemoryManager::allocatePages              L185  PC: 0x21654
MemoryManager::allocatePages (this=0x32f80 <memoryManager>, type=KERNEL,
    count=100) at ../src/kernel/memory.cpp:169
(gdb) n
(gdb) p physicalPageAddress
$1 = 2097152
(gdb) p/x physicalPageAddress
$2 = 0x200000
(gdb)
(gdb) p/x physicalPageAddress
$4 = 0x264000
(gdb) p/x physicalPageAddress
$6 = 0x26e000
```

主要关注的是p2第二次的物理内存，先是使用了 之前释放掉的内存 `0x264000`

```
(gdb) p/x physicalPageAddress
$2 = 0x264000
```

后面会用 `0x2d2000` 以后的内存继续补充，这样就可以达到**虚拟内存连续，但是物理内存不连续**的效果

```
(gdb) p /x physicalPageAddress
$5 = 0x2d3000
```

后面就不继续分析了，该代码物理地址与虚拟地址分开分配，释放也没有问题。

几点思考/疑惑

- 我们设计让内核虚拟地址从 `0xc0100000` 开始分配。其实内核的页目录表即 `0x100004` 到 `0x100C00` 这里**内核页目录表第1项到第767项都没有意义。**这样即使我们可管理内存有4GB往上（这里没有），我们内核也只能用最多1GB
- 我们设计了一个**页目录表->页目录表->页目录表->页目录表某一项**的寻找pde的方式，同时有一个**页目录表->页目录表->页目录表->页表**的寻找pte的方式。这些方式应该会引出要求**页表/页目录表大小要和页大小相一致。**但是实际操作系统还是这样吗?

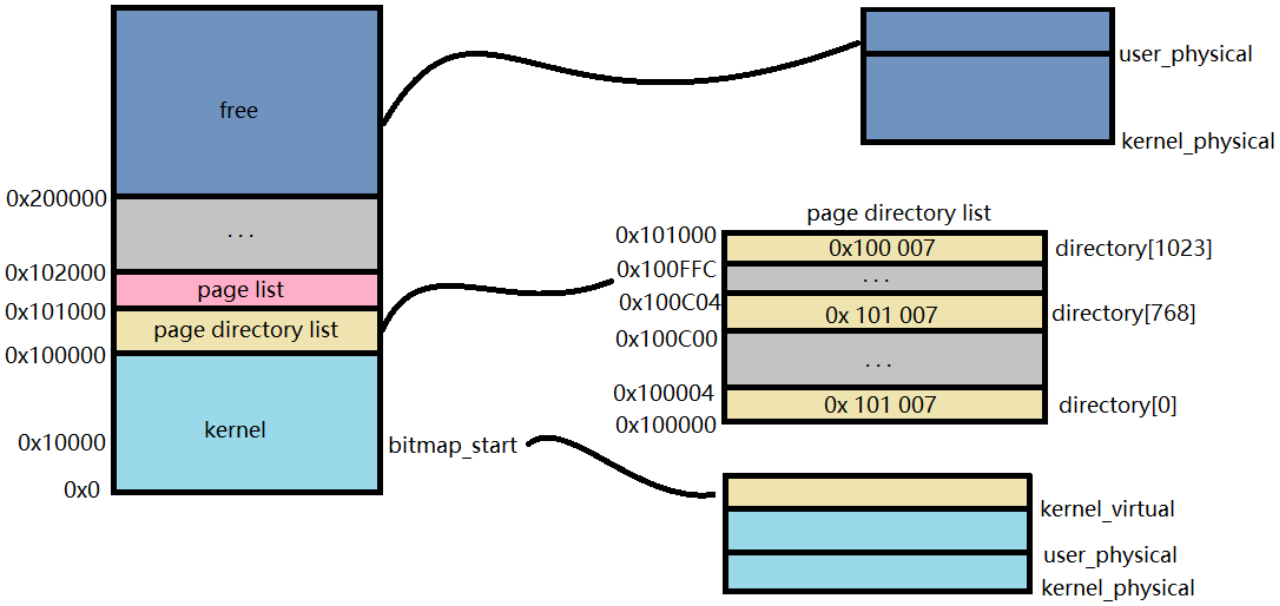
Assignment 4

实验要求

实现**页面置换**如FIFO、LRU等

实验原理

容易分析得到：我们的页目录表从768项开始分配一直到1023项，一共可以映射1GB的内容。
但是我们的内存大小全加起来只有126MB，**所以我们虚拟页分配到一定页数的时候，一定会出现物理页数量不足以继续分配的情况。**
所以我们的页面置换这样设定：在物理页内存已经装满，我们虚拟页还要继续分配空间，我们会根据算法替换掉某一个物理页。



而我们常用的页面置换算法有：LRU，FIFO等。
FIFO： 先进先出，可以直接替换掉最先分配的物理页
LRU： 根据页面最近使用情况来进行替换

实验过程

我们已经进入保护模式，进行磁盘读写已经不可以继续使用BIOS提供的中断。
为了简易实现，我们页面置换后的页直接进行释放其虚拟页和物理页。
（假如我们可以操作磁盘，那么其物理页就可以装进磁盘，虚拟页保留，缺页则继续替换）

FIFO算法简单实现
首先我们需要一个变量来记录找到最先进来的虚拟页

```
this->vir_to_exchange=KERNEL_VIRTUAL_START;
```

接着当我们虚拟页不够的时候就**顺着这个变量往下释放内存**，直到有足够的内存
释放的同时也要增长这个变量，用以保持它一直可以找到最先分配的内存

```
    while(!virtualAddress)
    {
        releasePages(type, vir_to_exchange, 1);
        vir_to_exchange+=PAGE_SIZE;
        if(vir_to_exchange==(0x4070000-0x200000+0xC0100000))
vir_to_exchange=0xC0100000;
        virtualAddress = allocateVirtualPages(type, count);
    }
```

效果
我们简单分配几个空间
这里p1，p2之后分配内存将超过地址空间，需要页面替换

```
    char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 15900);
    char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
    char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
    char *p4 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);

    printf("%x %x %x %x\n", p1, p2, p3,p4);
```

效果如图
p1之前最先分配的内存被p3，p4分配的页面替换掉