

**本科生实验报告**

实验课程：操作系统原理实验

任课教师：刘宁

实验题目：内存管理

专业名称：计算机科学与技术

学生姓名：

学生学号：

实验地点：实验中心B202

实验时间：2025.5.28

**Section 1 实验概述**

在本次实验中，首先学习位图和地址池的原理和概念，并使用位图和地址池来管理资源。然后，本次实验将将实现在物理地址空间下的内存管理。接着，在这基础之上，本次实验将会学习并开启二级分页机制。在开启分页机制后，将实现在虚拟地址空间下的内存管理。

本次实验最核心的地方在于二级分页机制。基于分页机制，我们可以将连续的虚拟地址空间映射到不连续的物理地址空间。同时，对于同一个虚拟地址，在不同的页目录表和页表下，会得到不同的物理地址。这为实现虚拟地址空间的隔离奠定了基础。

**Section 2 预备知识与实验环境**

* 预备知识：x86汇编语言、IA-32处理器体系结构、虚拟存储器、分页机制
* 实验环境：
  + 虚拟机版本/处理器型号：Virtualbox7.0.6、Ubuntu18.04
  + 代码编辑环境：gedit文本编辑器、gdb调试器、qemu
  + 代码编译工具：nasm

**Section 3 实验任务**

* **实验任务1：物理页内存管理的实现**

1. 结合代码分析位图，地址池，物理页管理的初始化过程，以及物理页进行分配和释 放的实现思路

2. 构造测试用例来分析物理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，结合测试用例简要分析物理页内存管理的实现的正确性

* **实验任务2：二级分页机制的实现**

1. 实现内存的申请和释放，保存实验截图并对能够在虚拟地址空间中进行内存管理， 截图并给出过程解释

2. 相比于一级页表，二级页表的开销是增大了的，但操作系统中往往使用的是二级页 表而不是一级页表。结合你自己的实验过程，说说相比于一级页表，使用二级页表会 带来哪些优势

* **实验任务3：虚拟页内存管理的实现**

1. 结合代码，描述虚拟页内存分配的三个基本步骤，以及虚拟页内存的释放的过程。

2. 构造测试用例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，结合测试用例简要分析虚拟页内存管理的实现的正确性

3. 请同学们借助第1000个页目录项，构造出第141 个页目录项的虚拟地址，和第891 个页目录项指向的页表中第109个页表项的虚拟地址

* **实验任务4：页面置换算法的实现**

在Assignment 3的基础上，实现一种理论课上学习到的虚拟内存管理中的页面置换算 法，在虚拟页内存中实现页面的置换.

**Section 4 实验步骤与实验结果**

------------------------- **实验任务1** -------------------------

* 任务要求：

1. 结合代码分析位图，地址池，物理页管理的初始化过程，以及物理页进行分配和释 放的实现思路

2. 构造测试用例来分析物理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，结合测试用例简要分析物理页内存管理的实现的正确性

* 思路分析：

实现物理页管理，首先我们需要实现bitmap位图工具来管理我们的资源，然后利用位图来管理我们的物理页来实现地址池工具实现物理页地址的管理.然后使用地址池将我们的内存空间划分为内核和用户区，分别用一个地址池来管理其中的物理页，实现物理页的分配和释放.

* 实验步骤：

**任务1：结合代码分析位图、地址池、物理页管理器的初始化过程，以及物理页进行分配和释放的实现思路**

**Step1：实现位图BitMap管理资源**

**位图的数据结构**：

class BitMap

{

public:

// 被管理的资源个数，bitmap的总位数

int length;

// bitmap的起始地址

char \*bitmap;

public:

// 初始化

BitMap();

// 设置BitMap，bitmap=起始地址，length=总位数(被管理的资源个数)

void initialize(char \*bitmap, const int length);

// 获取第index个资源的状态，true=allocated，false=free

bool get(const int index) const;

// 设置第index个资源的状态，true=allocated，false=free

void set(const int index, const bool status);

// 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

int allocate(const int count);

// 释放第index个资源开始的count个资源

void release(const int index, const int count);

// 返回Bitmap存储区域

char \*getBitmap();

// 返回Bitmap的大小

int size() const;

private:

// 禁止Bitmap之间的赋值

BitMap(const BitMap &) {}

void operator=(const BitMap&) {}

};

**BitMap成员函数的实现**：

位图的初始化：

位图的初始化就是传入一个连续的字符数组的地址和它的长度，然后将其初始化为0.

需要注意的是：一般情况下，成员涉及指针的对象的赋值都需要使用动态内存分配获得一个新的指针，但我们还没有实现动态内存分配。所以，将拷贝构造和赋值重载定义为private，以禁止BitMap之间的直接赋值。这也是为什么我们在BitMap的初始化函数initialize中需要提供BitMap的存储区域。

BitMap::BitMap(){

initialize(nullptr, 0);

}

void BitMap::initialize(char \*bitmap, const int length){

this->bitmap = bitmap;

this->length = length;

int bytes = ceil(length, 8);

memset(bitmap, 0, bytes);

}

其他成员函数：包括获取第index个资源的状态、设置第index个资源的状态、分配count个连续的资源、释放第index个资源开始的count个资源、返回Bitmap存储区域和大小.

bool BitMap::get(const int index) const{

int pos = index / 8;

int offset = index % 8;

return (bitmap[pos] & (1 << offset));

}

void BitMap::set(const int index, const bool status){

int pos = index / 8;

int offset = index % 8;

// 清0

bitmap[pos] = bitmap[pos] & (~(1 << offset));

// 置1

if (status){

bitmap[pos] = bitmap[pos] | (1 << offset);

}

}

int BitMap::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;

}

void BitMap::release(const int index, const int count){

for (int i = 0; i < count; ++i){

set(index + i, false);

}

}

char \*BitMap::getBitmap(){

return (char \*)bitmap;

}

int BitMap::size() const{

return length;

}

**Step2：实现地址池的来管理地址空间**

地址池的类的定义：

class AddressPool{

public:

BitMap resources;

int startAddress;

public:

AddressPool();

// 初始化地址池

void initialize(char \*bitmap, const int length,const int startAddress);

// 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

int allocate(const int count);

// 释放若干页的空间

void release(const int address, const int amount);

};

我们借助BitMap工具来管理地址池中的物理页资源

**地址池的初始化**：

地址池的初始化也比较简单，只需要指定一个位图和给定一个地址池的起始地址即可.

AddressPool::AddressPool(){}

// 设置地址池BitMap

void AddressPool::initialize(char \*bitmap, const int length, const int startAddress){

resources.initialize(bitmap, length);

this->startAddress = startAddress;

}

**其他函数的实现**：包括物理页的分配和释放

// 从地址池中分配count个连续页

int AddressPool::allocate(const int count){

uint32 start = resources.allocate(count);

return (start == -1) ? -1 : (start \* PAGE\_SIZE + startAddress);

}

// 释放若干页的空间

void AddressPool::release(const int address, const int amount){

resources.release((address - startAddress) / PAGE\_SIZE, amount);

**Step3：实现物理页管理**

利用物理页管理器对使用两个地址池对内核代码和用户代码进行区分和管理.

**类MemoryManager的定义**：

enum AddressPoolType{

USER,

KERNEL

};

class MemoryManager{

public:

// 可管理的内存容量

int totalMemory;

// 内核物理地址池

AddressPool kernelPhysical;

// 用户物理地址池

AddressPool userPhysical;

public:

MemoryManager();

// 初始化地址池

void initialize();

// 从type类型的物理地址池中分配count个连续的页

// 成功，返回起始地址；失败，返回0

int allocatePhysicalPages(enum AddressPoolType type, const int count);

// 释放从paddr开始的count个物理页

void releasePhysicalPages(enum AddressPoolType type, const int paddr, const int count);

// 获取内存总容量

int getTotalMemory();

};

**物理页管理的初始化过程**：

首先，我们读取之前在实模式下使用中断获取的内存大小，我们需要在在内存中预留一部分内存，0x00000000~0x00100000存放的是我们的内核，在预留内存中，1MB以上的剩余部分存放内核页表。

然后，为了实现的内存管理是页内存管理，并且我们将物理内存空间等分为两部分，内核物理地址空间和用户物理地址空间，同时我们计算两个物理地址空间的起始地址，用户物理地址空间紧跟在内核物理地址空间后面。

接着，我们在1MB以下的空间处人为划分了存放位图的区域，用来存放内核空间和用户空间的位图（BitMap），并我们对两部分空间的地址池进行初始化，最后我们将内存管理的基本信息打印出来.

MemoryManager::MemoryManager() {

initialize();

}

void MemoryManager::initialize(){

this->totalMemory = 0;

this->totalMemory = getTotalMemory();

// 预留的内存

int usedMemory = 256 \* PAGE\_SIZE + 0x100000;

if(this->totalMemory < usedMemory) {

printf("memory is too small, halt.\n");

asm\_halt();

}

// 剩余的空闲的内存

int freeMemory = this->totalMemory - usedMemory;

int freePages = freeMemory / PAGE\_SIZE;

int kernelPages = freePages / 2;

int userPages = freePages - kernelPages;

int kernelPhysicalStartAddress = usedMemory;

int userPhysicalStartAddress = usedMemory + kernelPages \* PAGE\_SIZE;

int kernelPhysicalBitMapStart = BITMAP\_START\_ADDRESS;

int userPhysicalBitMapStart = kernelPhysicalBitMapStart + ceil(kernelPages, 8);

kernelPhysical.initialize((char\*)kernelPhysicalBitMapStart, kernelPages, kernelPhysicalStartAddress);

userPhysical.initialize((char\*)userPhysicalBitMapStart, userPages, userPhysicalStartAddress);

printf("total memory: %d bytes ( %d MB )\n",

this->totalMemory,

this->totalMemory / 1024 / 1024);

printf("kernel pool\n"

" start address: 0x%x\n"

" total pages: %d ( %d MB )\n"

" bitmap start address: 0x%x\n",

kernelPhysicalStartAddress,

kernelPages, kernelPages \* PAGE\_SIZE / 1024 / 1024,

kernelPhysicalBitMapStart);

printf("user pool\n"

" start address: 0x%x\n"

" total pages: %d ( %d MB )\n"

" bit map start address: 0x%x\n",

userPhysicalStartAddress,

userPages, userPages \* PAGE\_SIZE / 1024 / 1024,

userPhysicalBitMapStart);

}

**物理页的分配**：

我们根据传入的地址空间的类型是用户还是内核，然后在其对应的地址池申请count数量的物理页即可.

int MemoryManager::allocatePhysicalPages(enum AddressPoolType type, const int count)

{

int start = -1;

if (type == AddressPoolType::KERNEL)

{

start = kernelPhysical.allocate(count);

}

else if (type == AddressPoolType::USER)

{

start = userPhysical.allocate(count);

}

return (start == -1) ? 0 : start;

}

**物理页的释放**：

页的释放同理，根据传入的地址空间的类型是用户还是内核，然后在其对应的地址池释放count数量的物理页即可.

void MemoryManager::releasePhysicalPages(enum AddressPoolType type, const int paddr, const int count)

{

if (type == AddressPoolType::KERNEL)

{

kernelPhysical.release(paddr, count);

}

else if (type == AddressPoolType::USER)

{

userPhysical.release(paddr, count);

}

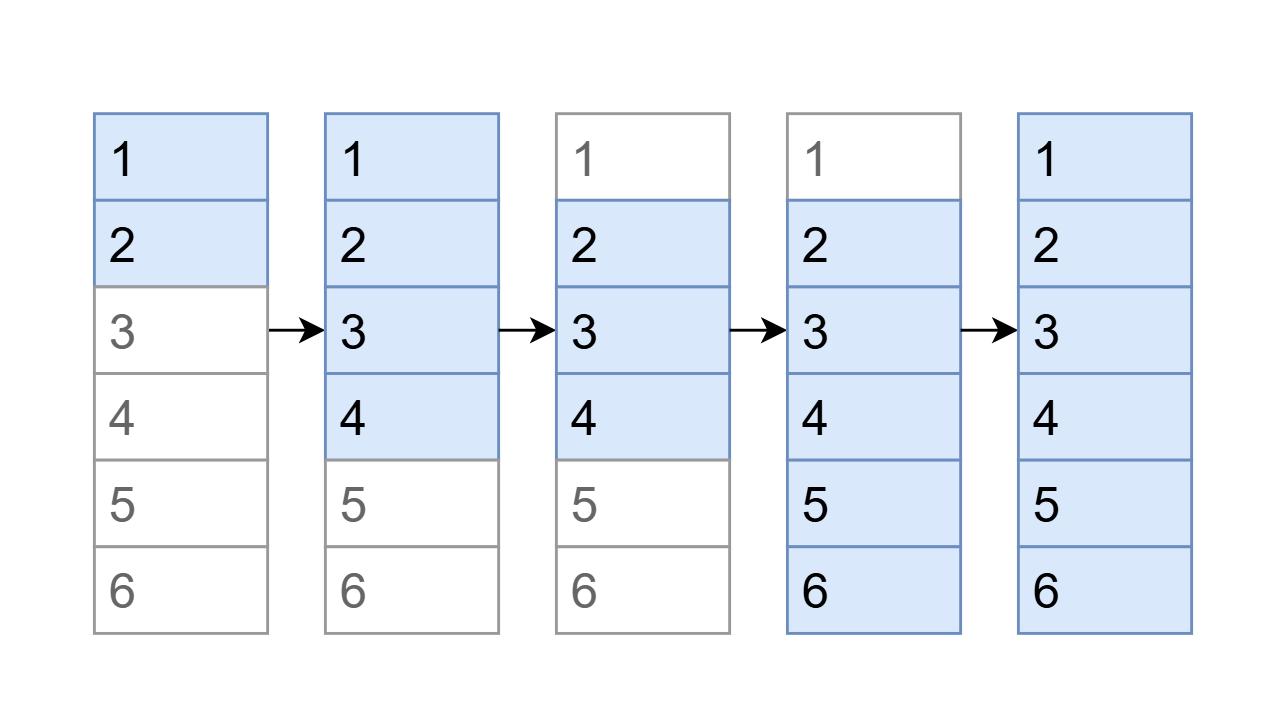
}

**任务2：构造测试用例测试物理页内存管理器.**

我构造了下面的一个分配例子，先连续申请2页、2页、然后从起始地址开始释放1页，然后再依次申请2页、1页.



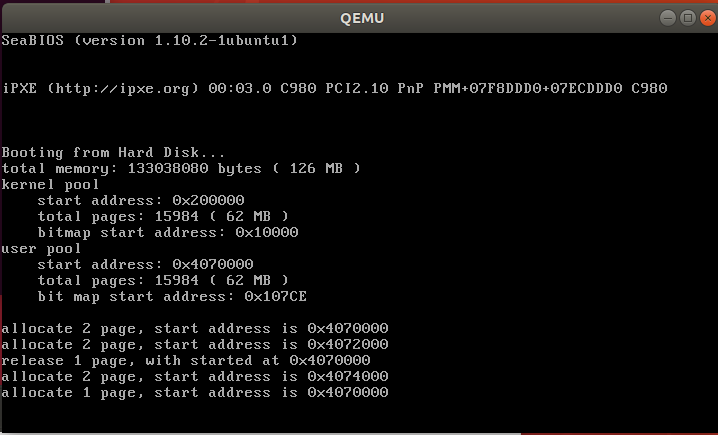
这样会出现的预计效果是这样子的



接着可以在实验结果展示里观察申请、释放的结果，可以观察到和我们预计的是一样的，这说明了物理页管理器没有什么bug.

* 实验结果展示：通过执行前述代码，可得下图结果。

可以看到我们正确的分配和释放了物理页，并没有什么bug



------------------------- **实验任务2** -------------------------

* 任务要求：

1. 实现内存的申请和释放，保存实验截图并对能够在虚拟地址空间中进行内存管理， 截图并给出过程解释（比如：说明哪些输出信息描述虚拟地址，哪些输出信息描述物 理地址），使用的物理地址或虚拟地址信息与学号相关联（比如学号后四位作为页内 偏移）

2. 相比于一级页表，二级页表的开销是增大了的，但操作系统中往往使用的是二级页 表而不是一级页表。结合你自己的实验过程，说说相比于一级页表，使用二级页表会 带来哪些优势

* 思路分析：

开启二级分页机制，我们需要实现页表、页目录这两个工具，然后将页目录的起始地址放入cr3寄存器中，使得MMU能够找到页目录表的起始地址，最后将cr0的PG置为1开始CPU中自带的二级分页寻址机制，完成这三步就可以开启分页机制了.

* 实验步骤：

**任务1：实现内存的申请和释放，说明哪些输出信息描述虚拟地址，哪些输出信息描述物理地址**

**Step1：规划好页目录表和页表在内存中的位置，然后初始化**

我们将页目录表放在1MB处。由于内核很小，假设内核只会放在0~1MB的内存区域。然后，为了方便，我们设置虚拟地址和物理地址相同。

首先，虚拟地址0~1MB的二进制表示是0x00000000~0x000fffff，其31-22位均为0，对应第0个页目录项。因此只需要初始化第0个页目录项和其对应的页表即可。第0个页目录项被放在页目录表之后，地址是PAGE\_DIRECTORY + PAGE\_SIZE。然后我们取21~12位，范围从0x000~0xfff，涉及256个页表项。由于我们希望线性地址经过翻译后的物理地址依然和线性地址相同。因此，这256个页表项分别指向物理页的第0-255页.同时我们要设置U/S，R/W和P位为1，也就是|0x7

最后利用汇编函数将页目录表的地址放入cr3寄存器，然后将cr0的PG位置1便可开启分页机制

void MemoryManager::openPageMechanism(){

// 页目录表指针

int \*directory = (int \*)PAGE\_DIRECTORY;

//线性地址0~4MB对应的页表

int \*page = (int \*)(PAGE\_DIRECTORY + PAGE\_SIZE);

// 初始化页目录表

memset(directory, 0, PAGE\_SIZE);

// 初始化线性地址0~4MB对应的页表

memset(page, 0, PAGE\_SIZE);

int address = 0;

// 将线性地址0~1MB恒等映射到物理地址0~1MB

for (int i = 0; i < 256; ++i){

// U/S = 1, R/W = 1, P = 1

page[i] = address | 0x7;

address += PAGE\_SIZE;

}

// 初始化页目录项

// 0~1MB

directory[0] = ((int)page) | 0x07;

// 3GB的内核空间

directory[768] = directory[0];

// 最后一个页目录项指向页目录表

directory[1023] = ((int)directory) | 0x7;

// 初始化cr3，cr0，开启分页机制

asm\_init\_page\_reg(directory);

printf("open page mechanism\n");

}

**Step2：利用汇编将页目录表的地址写入cr3，同时将cr0的PG位置1开启二级分页机制.**

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

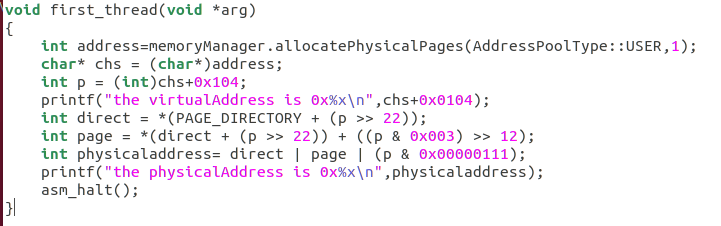
pop eax

pop ebp

ret

**Step3：接着我们编写一个测试程序，看看我们的二级分页的虚拟地址和物理地址的关系.**

首先我们分配一个页，然后查看数组中第0x104个元素的地址，以及通过人工转换的物理地址，然后再释放掉.



然后我们就使用实验任务2的例子来测试二级分页机制的内存申请和释放的功能.

运行的结果在结果展示部分展出，可以看到虚拟地址和物理地址的页内偏移是一样的，这符合二级分页机制的特点.

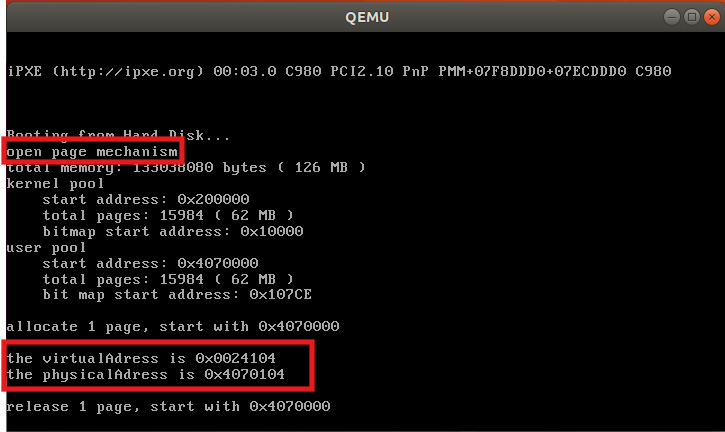
**任务2：二级页表的优势**

一级页表可以满足分页存储的思想，但是一级页表有一个问题，假设一级页表中可容纳1M个页表项，每个页表项是4字节。如果页表项全满的话，便是4MB大小，需要4MB/4KB=1K页，也就是说光是页表就要占用1k个页还必须是连续的，注意到每个进程都有自己的页表，当进程数量增多时，光是页表占用的内存空间就不小了，这显然是不能接受的.所以在这基础上设计除了二级分页机制.

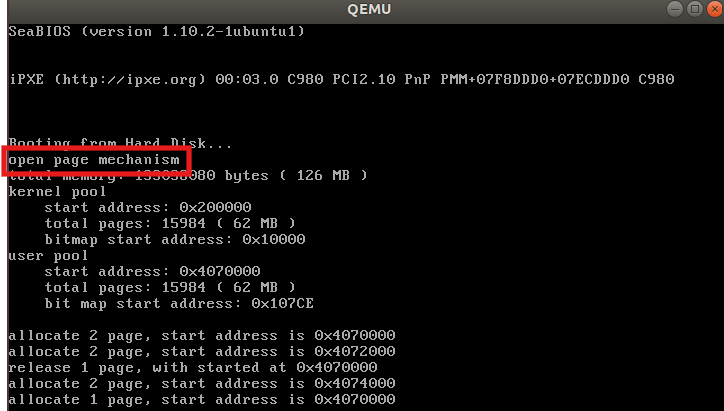
所以，我们不希望在一开始就创建所有的页表项，而是根据需要动态地创建页表项。二级页表为页表也创建了一个页表，称为页目录，页目录需要1K个页目录项，也就是4KB的内存，刚好等于一页.有了二级页表，这样我们的一级页表的页表项就可以放在磁盘空间中，当需要用到时，再根据页表目录项的地址将其所在的页表调入内存.这样就减少了内存占用.

综上所述，我们可以发现二级页表相较于一级页表，**减少了页表所占用的连续的内存空间**.避免了浪费内存空间去存储无用的页表项.节省内存空间.

* 实验结果展示：通过执行前述代码，可得下图结果。



可以看到我们的虚拟地址前十位为0000 0000 00，表示在第一个页目录项，中间的十位表示在所在的页表项，最后12位0x104表示页内偏移也就是我的学号。根据二级页表的特性物理地址和虚拟地址的页内偏移应该是一样的都是0x104，这说明二级分页机制是有效的.



可以看到我们开启了二级分页机制，同时内存的申请和释放都是顺利的进行着.

------------------------- **实验任务3** -------------------------

* 任务要求：

1. 结合代码，描述虚拟页内存分配的三个基本步骤，以及虚拟页内存的释放的过程。

2. 构造测试用例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，结合测试用例简要分析虚拟页内存管理的实现的正确性

3. 在PDE(页目录项)和PTE(页表项)的虚拟地址构造中，我们使用了第1023个页目录 项。第1023个页目录项指向了页目录表本身，从而使得我们可以构造出PDE和PTE 的虚拟地址。现在，我们将这个指向页目录表本身的页目录项放入第1000个页目录项， 而不再是放入了第1023个页目录项。请借助第1000个页目录项，**构造出第141个页 目录项的虚拟地址**，和**第891个页目录项指向的页表中第109个页表项的虚拟地址**

* 思路分析：

实现虚拟页内存管理首先我们的内存空间已经拓展到了四块即用户虚拟地址空间，用户物理地址空间，内核虚拟地址空间，内核物理地址空间，内存管理器MemoryManager只需要关心全局的地址空间即可，即用户物理地址空间，内核虚拟地址空间，内核物理地址空间，因为用户虚拟地址空间并不是全局的而是在PCB中.之后我们需要在虚拟地址空间分配虚拟页，同时我们要为已分配的虚拟页分配一个物理页，同时通过二级分页机制，将虚拟页与物理页连接起来，这样就实现了虚拟页内存管理.

* 实验步骤：

**任务1：结合代码，描述虚拟页内存分配的三个基本步骤和释放的过程**

**Step1：修改MemoryManage，增加虚拟地址池来管理虚拟页，同时也要修改我们的初始化函数.**

void MemoryManager::initialize(){

this->totalMemory = 0;

this->totalMemory = getTotalMemory();

// 预留的内存

int usedMemory = 256 \* PAGE\_SIZE + 0x100000;

if (this->totalMemory < usedMemory){

printf("memory is too small, halt.\n");

asm\_halt();

}

// 剩余的空闲的内存

int freeMemory = this->totalMemory - usedMemory;

int freePages = freeMemory / PAGE\_SIZE;

int kernelPages = freePages / 2;

int userPages = freePages - kernelPages;

int kernelPhysicalStartAddress = usedMemory;

int userPhysicalStartAddress = usedMemory + kernelPages \* PAGE\_SIZE;

int kernelPhysicalBitMapStart = BITMAP\_START\_ADDRESS;

int userPhysicalBitMapStart = kernelPhysicalBitMapStart + ceil(kernelPages, 8);

int kernelVirtualBitMapStart = userPhysicalBitMapStart + ceil(userPages, 8);

kernelPhysical.initialize(

(char \*)kernelPhysicalBitMapStart,

kernelPages,

kernelPhysicalStartAddress);

userPhysical.initialize(

(char \*)userPhysicalBitMapStart,

userPages,

userPhysicalStartAddress);

kernelVirtual.initialize(

(char \*)kernelVirtualBitMapStart,

kernelPages,

KERNEL\_VIRTUAL\_START);

printf("total memory: %d bytes ( %d MB )\n",

this->totalMemory,

this->totalMemory / 1024 / 1024);

printf("kernel pool\n"

" start address: 0x%x\n"

" total pages: %d ( %d MB )\n"

" bitmap start address: 0x%x\n",

kernelPhysicalStartAddress,

kernelPages, kernelPages \* PAGE\_SIZE / 1024 / 1024,

kernelPhysicalBitMapStart);

printf("user pool\n"

" start address: 0x%x\n"

" total pages: %d ( %d MB )\n"

" bit map start address: 0x%x\n",

userPhysicalStartAddress,

userPages, userPages \* PAGE\_SIZE / 1024 / 1024,

userPhysicalBitMapStart);

printf("kernel virtual pool\n"

" start address: 0x%x\n"

" total pages: %d ( %d MB ) \n"

" bit map start address: 0x%x\n",

KERNEL\_VIRTUAL\_START,

userPages, kernelPages \* PAGE\_SIZE / 1024 / 1024,

kernelVirtualBitMapStart);

}

**Step2：实现虚拟页内存分配**

虚拟页内存分配总共有3个步骤：

* 从虚拟地址池中分配若干连续的虚拟页
* 对每一个虚拟页，从物理地址池中分配1页
* 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

**第一步：allocateVirtualPages从虚拟地址池中分配若干连续的虚拟页**

int allocateVirtualPages(enum AddressPoolType type, const int count){

int start = -1;

if (type == AddressPoolType::KERNEL) {

start = kernelVrirtual.allocate(count);

}

return (start == -1) ? 0 : start;

}

**第二步：allocatePhysicalPages对每一个虚拟页都分配1页物理页**

这一步比较简单，就是调用地址池的成员函数即可.

**第三步：connectPhysicalVirtualPage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**

首先，我们检查pde中是否有对应的页表，如果没有，就要先分配一个物理页，然后初始化新分配的物理页并将其地址写入pde，以作为pde指向的页表，在pde对应的物理页存在的前提下，我们将之前为虚拟页分配的物理页地址写入pte即可.

bool MemoryManager::connectPhysicalVirtualPage(const int virtualAddress, const int physicalPageAddress){

// 计算虚拟地址对应的页目录项和页表项

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;

}

由于我们开启的了二级分页机制，所以我们对虚拟地址的页目录项和页表项寻址也需要虚拟地址，但是我们的所知的页表目录的起始地址是物理地址，所以我们需要构造出虚拟地址的页目录项和页表项的虚拟地址.

构造原理在任务2中会详细说明.

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));

}

**最后将三个步骤组合起来就是我们的虚拟页分配函数**：

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，使虚拟页内的地址经过分页机制变换到物理页内。

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;

}

**Step3：实现虚拟页内存的释放**

虚拟页内存释放主要分为两步：

* 对每一个虚拟页，释放为其分配的物理页
* 释放虚拟页

**第一步：对每一个虚拟页，释放为其分配的物理页**.

为了找到虚拟页对应的物理页，我们需要通过页目录和页表找到其对应的物理地址.

根据分页机制，一个虚拟地址对应的物理页的地址是存放在页表项中的。因此，我们先求出虚拟地址的页表项的虚拟地址，然后访问页表项，取页表项内容的31-12位就是物理页的物理地址，最后替换虚拟地址的31-12位即可得到虚拟地址对应的物理地址.

int MemoryManager::vaddr2paddr(int vaddr){

int \*pte = (int \*)toPTE(vaddr);

int page = (\*pte) & 0xfffff000;

int offset = vaddr & 0xfff;

return (page + offset);

}

然后我们释放物理页，这里比较简单直接调用地址池的函数即可

**第二步：释放虚拟页**.

这一步也比较简单，直接调用虚拟地址池的成员函数release即可.

void MemoryManager::releaseVirtualPages(enum AddressPoolType type, const int vaddr, const int count){

if (type == AddressPoolType::KERNEL){

kernelVirtual.release(vaddr, count);

}

}

**最后将两步结合起来就是我们的虚拟页释放函数**：

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;

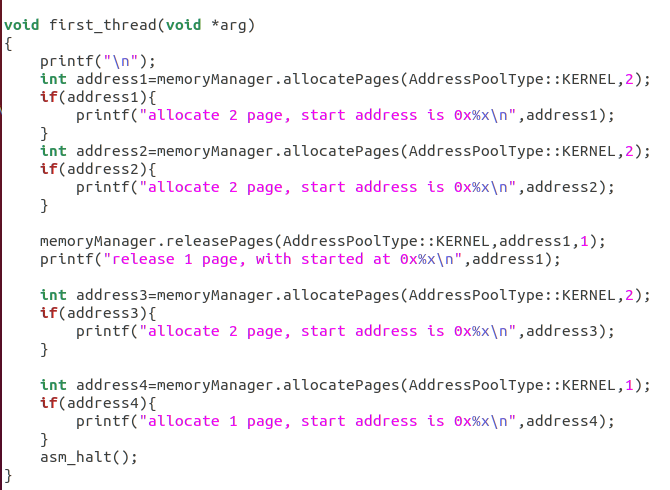
}

releaseVirtualPages(type, virtualAddress, count);

}

**任务2：构造测试用例来分析虚拟页内存管理**

这个样例和实验任务1中的例子申请、释放的页的顺序、数量都是一致的，不同的是，这里使用了虚拟页管理器来管理我们的内存.在实验结果截图中可以看到我们的起始地址都变成了虚拟地址，证明了虚拟页内存分配的有效性.



最后可以在实验结果展示里观察申请、释放的结果，可以观察到和我们预计的是一样的，这说明了物理页管理器没有什么bug.

**任务3：构造虚拟地址**

**3.1 首先构建第141个页目录项的虚拟地址**

①首先，页目录项所在的物理页是页目录表，第141个页目录项的页目录号是141，而每一个页目录项的大小是4个字节，因此页内偏移为

0010 0011 0100(141\*4)

②接下来我们构造

页目录表的第1000个页目录项指向了页目录表,因此我们有

=11 1110 1000(1000)

③最后，我们来构造

是页目录项的序号，第1000个页表项指向了页目录这个“页表”。因此我们有

=11 1110 1000

至此，我们已经完成了第141个页目录项的虚拟地址构建：

11 1110 1000|11 1110 1000|0010 0011 0100 **0xFA3E 8234**

**3.2 构建第891个页目录项指向的页表中第109个页表项的虚拟地址**

①首先，页表项所在的物理页是页表，第891个页目录项指向的页表中第109个页表项的页号是109，而每一个页表项的大小是4个字节，因此我们有

= 0001 1011 0100 (109\*4)

②接下来我们构造

是位于页表的，页目录表的第891个页目录项指向了这个页表，因此有

=11 0111 1011 (891)

③最后，我们来构造

是页目录项的序号，第1000个页目录项指向了页目录这个特殊的页表，因此我们有

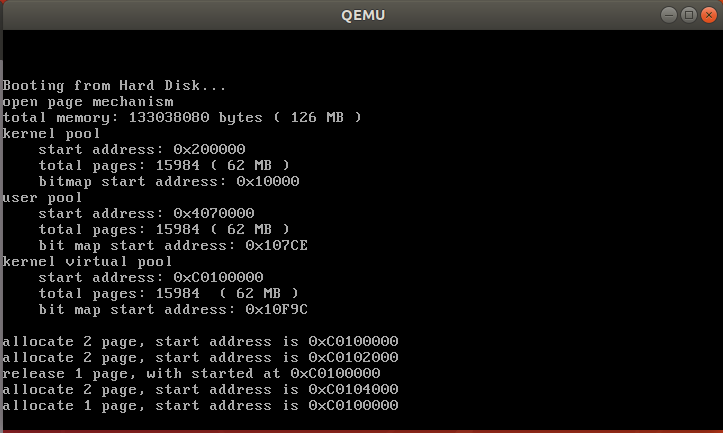
=11 1110 1000 (1000)

至此，我们已经完成了第891个页目录项指向的页表中第109个页表项的虚拟地址构建：

11 1110 1000|11 0111 1011|0001 1011 0100 **0xFA37 B1B4**

* 实验结果展示：通过执行前述代码，可得下图结果，

可以看到我们成功在虚拟地址空间申请和释放了内存，同时我们的地址全部都变成了虚拟地址，不在是物理地址.



------------------------- **实验任务4** -------------------------

* 任务要求：

在Assignment 3的基础上，实现一种理论课上学习到的虚拟内存管理中的页 面置换算法，在虚拟页内存中实现页面的置换.

* 思路分析：

页面置换算法有很多，比如OPR、LFU等等，这里我选择了FIFO先进先出的页面置换算法.

FIFO的页面置换算法，就是利用队列先进先出的特性，每次申请一页内存时，就将其加入队列，当我们的虚拟页的数量不足时，将队头的虚拟内存释放掉，再来分配内存，实现虚拟内存中页面的置换.

考虑到我们虚拟地址空间是连续的，这是一个天然的队列，我们只需要维护一个队头指针和一个队尾指针，就可以让虚拟地址空间拥有循环队列的特性.这样我们再修改虚拟页分配规则：只允许从队尾处寻找内存进行分配，如果虚拟页不够，就从队头开始释放足够的虚拟页然后再进行分配，实现FIFO的页面置换.

* 实验步骤：

**Step1:修改分配页的规则，寻找是否分配成功时，我们只允许从队尾处开始寻找是否有足够的资源.**

int BitMap::allocate(const int count)

{

......

empty = 0;

start = queueTail;

while ((queueTail < length) && (!get(queueTail)) && (empty < count))

{

      ++empty;

      ++queueTail;

 }

......

 return -1;

}

**Step2：修改MemoryManager**，我们修改一下虚拟页分配的规则：queueHead 是定义的队头指针，如果虚拟页分配失败，那么就释放掉队头的指针指向的虚拟页，同时让队头指针指向下一页，

queueHead = (queueHead == VirtualAddressEnd) ? KERNEL\_VIRTUAL\_START : queueHead + PAGE\_SIZE;

是这个算法的核心，它实现了一个循环队列，使得当队头指针到达了虚拟地址池的结尾后，能够返回到开始处，从而实现了FIFO的页面置换算法.

int MemoryManager::allocatePages(enum AddressPoolType type, const int count)

{

.......

    int cnt = 0;

    while (!virtualAddress)

    {

        releasePages(type, queueHead ,1);

        cnt += 1;

int VirtualAddressSize = userPhysicalStartAddress - kernelPhysicalStartAddress;

int VirtualAddressEnd = VirtualAddressSize + KERNEL\_VIRTUAL\_START;

queueHead = (queueHead == VirtualAddressEnd) ? KERNEL\_VIRTUAL\_START : queueHead + PAGE\_SIZE;

        virtualAddress = allocateVirtualPages(type, count);

    }

    if(cnt){

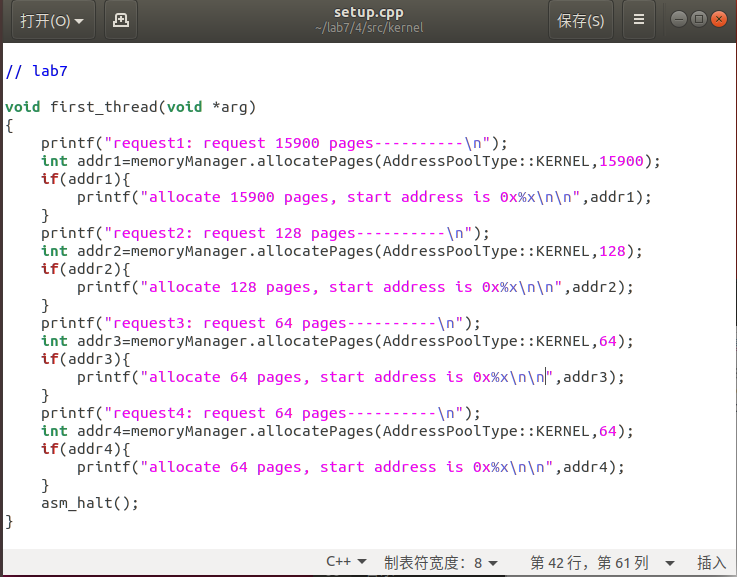
        printf("release %d page with start address 0x%x\n",cnt, queueHead-cnt \* PAGE\_SIZE);

    }

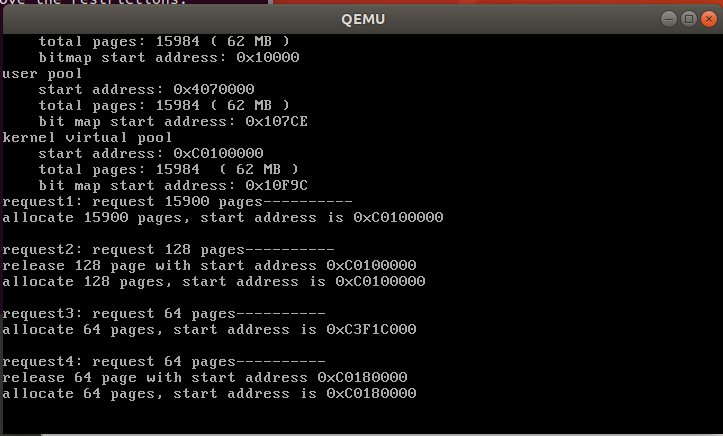
  ......

}

**Step3：测试FIFO页面置换算法，设计了如下的测试用例来测试**



* 实验结果展示：通过执行前述代码，可得下图结果，



从图中的结果我们可以看到

* 第一次我分配了15900页，此时还剩下84页（15984-15900），几乎把所有虚拟页都分配完了
* 接着我分配128页虚拟页时，由于虚拟页不够，所以从起始位置开始，这也是最先分配的一页，释放了128页虚拟页后，再分配128页虚拟页，这样就完成了页面的置换.
* 第三次分配申请了64页，由于前面还剩下84页，所以足够分配，所以就没有进行置换，直接分配.
* 最后一次，申请了64页，但是虚拟页不够，所以从上一次释放128页的位置继续释放64页，然后将这64页分配，完成页面的置换.

实验的输出结果页符合FIFO的页面置换算法

**Section 5 实验总结与心得体会**

本次实验我们实现了操作系统的内存管理.实验的一开始我们实现了位图工具实现了对内存资源的管理，然后为了开启分页机制，本次实验又实现了地址池的工具来管理我们地址空间的物理页.然后本次实验将地址空间对半分成了两个部分内核空间和用户空间，并实现了内存管理器来管理这两部分的内存.这样就完成了物理页内存管理.

在完成了物理页内存管理的基础上，然后开始开启我们的二级本页机制，来使得我们的CPU能够使用二级分页机制来实现内存的访问.开启二级分页机制主要是实现我们的页目录基址寄存器和我们的第一个页表，然后将页目录基址寄存器的地址写入cr3，然后将cr0置为1开启了CPU的二级分页寻址方式.

在开启了二级分页机制之后，接着本次实验就要实现我们的虚拟页内存管理.为了实现虚拟页内存管理，我们要在我们的内存管理器中创建一个虚拟页的地址池来管理我们的虚拟页，然后我们要实现物理页和虚拟页之间的连接，其实就是在我们的页目录和页表中存入相应的地址，使得我们的CPU能通过页目录和页表来找到我们实际物理页的地址，然后再加上页内偏移就可以实现我们的内存的寻址.

总之，本次实验学习和复习到了操作系统中的分页管理，虚拟内存管理、二级分页机制的实现和原理.同时对于操作系统的内存管理有了更加深入的理解.

**Section 6 附录：代码清单**

由于代码文件较多， 代码量较大， 为了篇幅的精简， 具体代码以附件的形式给出