



中山大學
SUN YAT-SEN UNIVERSITY

操作系统实验 7

内存管理

实验课程：操作系统原理实验

实验名称：内存管理

专业名称：计算机科学与技术

学生姓名：钟旺烜

学生学号：23336342

实验地点：实验楼 B203

实验成绩：

报告时间：2025 年 6 月 1 日

Section 1 实验概述

●实验任务 1:

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。

●实验任务 2:

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。

●实验任务 3:

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，也可以提出自己的算法。

●实验任务 4:

复现“虚拟页内存管理”一节的代码，完成如下要求。

- 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。
- （不做要求，对评分没有影响）如果你有想法，可以在自己的理解的基础上，参考 ucore，《操作系统真象还原》，《一个操作系统的实现》等资料来实现自己的虚拟页内存管理。在完成之后，你需要指明相比较于本教程，你的实现的虚拟页内存管理的特点所在。

Section 2 实验步骤与实验结果

----- 实验任务 1 -----

●任务要求:

✦Assignment 1

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。

●思路分析:

实现二级分页机制的代码已经在实验材料中给出，直接复制到虚拟机中使用 qemu 编译启动操作系统即可。

下面通过设计一个申请和释放内存的线程，通过运行操作系统来观察并查看线程是否能够正确的进行内存的申请和释放。

●实验步骤:

首先下面给出第一个线程有关内存的申请和释放的代码, 代码保存在 `setup.cpp` 中:

```
void first_thread(void *arg)
{
    // 第 1 个线程不可以返回
    // stdio.moveCursor(0);
    // for (int i = 0; i < 25 * 80; ++i)
    // {
    //     stdio.print(' ');
    // }
    // stdio.moveCursor(0);
    char *p1 = (char *)memoryManager.allocatePhysicalPages(AddressPoolType::USER, 2);
    char *p2 = (char *)memoryManager.allocatePhysicalPages(AddressPoolType::USER, 2);
    char *p3 = (char *)memoryManager.allocatePhysicalPages(AddressPoolType::USER, 2);
    memoryManager.releasePhysicalPages(AddressPoolType::USER, (int)p2, 2);
    char *p4 = (char *)memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);
    char *p5 = (char *)memoryManager.allocatePhysicalPages(AddressPoolType::USER, 2);

    asm_halt();
}
```

可以看到线程首先在用户内存空间中进行三次内存申请, 三次均为申请连续的 2 个页面。随后释放第二次申请的内存块的两个页面, 然后再次进行两次内存申请, 分别分配 1 个和 2 个页面。

为了更好的显示内存申请和释放的结果, 在 `allocatePhysicalPages` 与 `releasePhysicalPages` 函数中添加 `printf` 函数显示内存申请和释放的操作, 添加后两个函数的代码如下所示:

```
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);
    }
    if (start != -1)
        printf("allocate %d page(s) successfully, start_address = 0x%x\n", count, start);

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

void MemoryManager::releasePhysicalPages(enum AddressPoolType type, const int paddr, const int count)
{
    if (type == AddressPoolType::KERNEL)
```

```

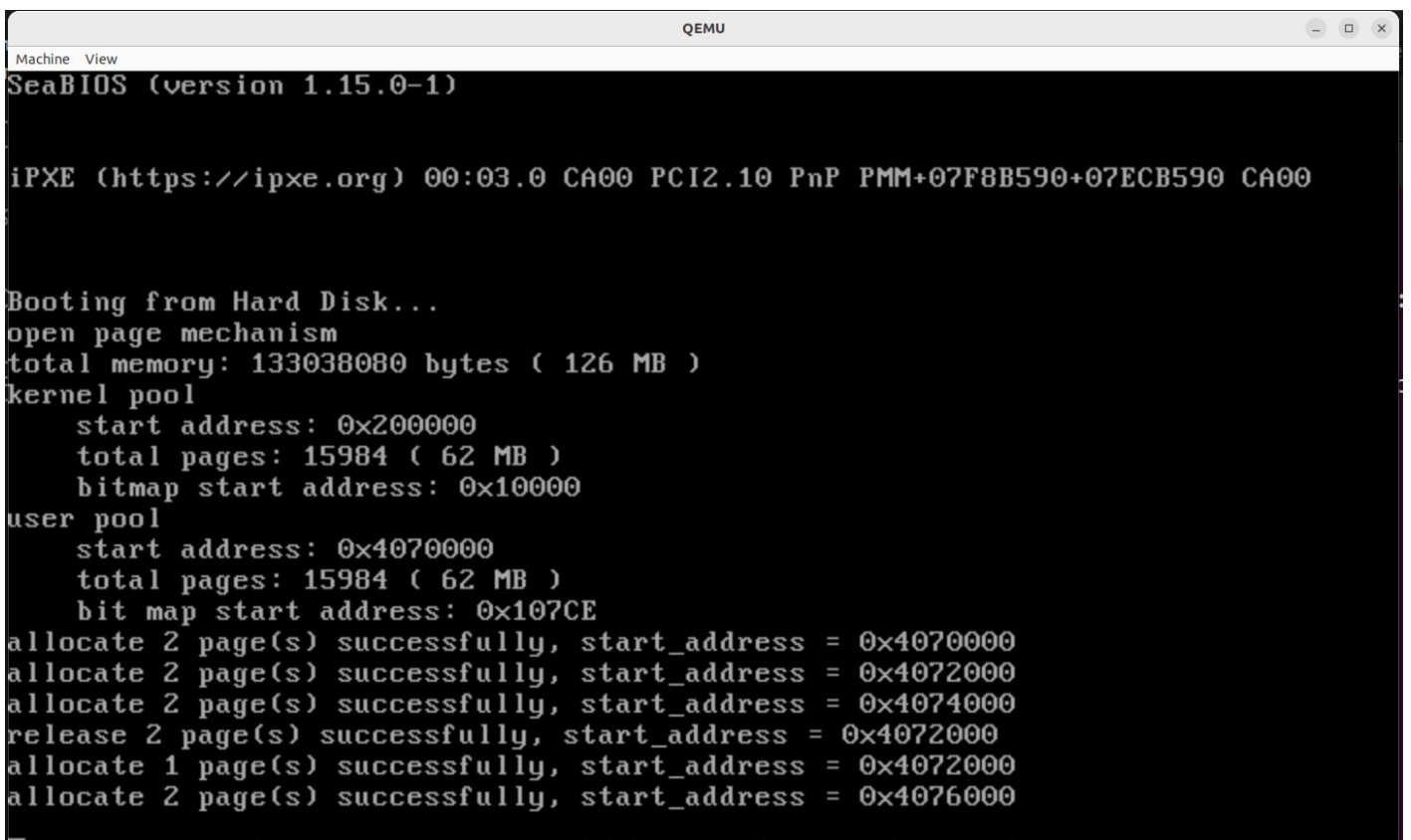
    kernelPhysical.release(paddr, count);
}
else if (type == AddressPoolType::USER)
{
    userPhysical.release(paddr, count);
}
printf("release %d page(s) successfully, start_address = 0x%x\n", count, paddr);
}

```

添加完毕后在 terminal 中使用相关 make 指令编译运行操作系统,可以在 qemu 显示屏上观察到相关内存分配和释放信息:

●实验结果展示:通过执行前述代码,可得下图结果。

通过运行我们设计好的线程进行内存的分配和释放后,可以得到下面显示器所示结果:



The screenshot shows a QEMU terminal window with the following output:

```

SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00

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
allocate 2 page(s) successfully, start_address = 0x4070000
allocate 2 page(s) successfully, start_address = 0x4072000
allocate 2 page(s) successfully, start_address = 0x4074000
release 2 page(s) successfully, start_address = 0x4072000
allocate 1 page(s) successfully, start_address = 0x4072000
allocate 2 page(s) successfully, start_address = 0x4076000

```

首先根据开启二级页表分页机制的显示结果,可以看到用户内存池的起始地址为 0x4070000。

因此,当使用 `allocatePhysicalPages` 函数向用户内存池中申请内存时,操作系统会从用户内存池中的 Bitmap 中找到连续分布的足够的空闲页,如果申请成功,便会将申请页在 bitmap 中的索引位置 1 表示已经使用并返回这连续分配的页的首个页的地址以供线程使用。

因此可以看到我们前三个申请内存的操作一共向用户内存池申请了 6 个页,起始地址分别为 0x4070000,0x4072000 和 0x4074000。

接着进行了释放第二个申请的内存块的操作:使用 `releasePhysicalPages` 进行释放内存操作时,传入释放内存块的起始地址以及释放页的数量,操作系统便会将相应内存池中的 bitmap 中对应页的索引处置零,表示页已经释放。如图所示,其实于 0x4072000 的两个页面成功释放。

然后,再进行两个内存申请操作。注意,实验材料中有关物理内存的分配实现的是 `first_fit` 动态分配,在这个前提下可以发现,当进行第四次内存申请 1 个页面时,内存的起始地址和进行第二次内存申请的起始地址一致,都是 0x4072000,这是因为第二次内存申请已经被提前释放,因此得到了从地

址 0x4072000 开始的两个空闲页面。

而进行第五次内存申请 2 个页面时，因为第四次内存申请了 1 个页面，在 0x4073000 只有一个页面空闲，因此第五次内存申请的起始地址为 0x4076000,申请到了此地址往后的两个空闲页面。

以上我们初步验证了线程在开启二级分页机制后能够正确进行了内存的申请和释放。

----- 实验任务 2 -----

●任务要求：

Assignment 2

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。

●思路分析：

实验资料提供的代码已经实现了 first-fit 动态分区算法，这里实现 best-fit 动态分配算法，并设计进行申请和释放内存操作的线程，通过分别使用两个算法来运行操作系统，来检验物理内存分配算法的正确性并体现两种算法的不同之处。

Best-Fit 算法 是一种经典的内存分配算法，主要用于操作系统的内存管理。它的核心思想是从空闲内存块中选择最小但足够满足需求的块进行分配，从而尽量减少内存碎片的产生。

对实现的 BitMap 类中的 allocate()函数进行修改，来实现 Best-fit 物理分配算法。

在 set()函数原来的 first-fit 物理分配算法的基础上，修改几条条件判定逻辑，并且新增 best_len 和 best_start 变量来记录遍历 BitMap 时找到的最适合内存块的页数以及起始地址。算法具体流程如下：

首先遍历整个 BitMap，在遍历过程中将那些连续且满足分配长度的内存块使用 best_len 和 best_start 变量记录下来，当发现比 best_len 更小且满足分配长度要求的内存块出现时，便更新 best_len 和 best_start 变量。遍历完成后，如果 best_len 大于内存要求分配长度，即分配内存成功，更新 BitMap 并返回内存块的起始地址。否则分配失败。

这里同样解释一下 first-fit 算法的工作原理：从内存的低地址开始顺序扫描空闲块链表，选择第一个足够大的空闲块进行分配。

●实验步骤：

在原来 BitMap 类的 allocate()函数基础上进行修改，来实现 Best-fit 内存分配算法进行内存的动态分配。相关代码保存在 bitmap.cpp 中，具体代码如下：

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

    int index, empty, start, best_len, best_start;

    best_len = length;
    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
    empty = 0;
    start = index;
    while ((index < length) && (!get(index)))
    {
        ++empty;
        ++index;
    }

    if (empty >= count && empty < best_len)
    {
        best_len = empty;
        best_start = start;
    }
}

if (best_len >= count)
{
    for (int i = 0; i < count; ++i)
    {
        set(best_start + i, true);
    }
    return best_start;
}
else
    return -1;
}

```

实现完 best-fit 内存动态分配算法后，我们设计一个能够体现 first-fit 和 best-fit 两种算法不同之处的对内存进行申请和释放的线程，线程代码保存在 setup.cpp 中，具体代码如下：

```

void first_thread(void *arg)
{
    char *p1 = (char *)memoryManager.allocatePhysicalPages(AddressPoolType::USER,1);
    char *p2 = (char *)memoryManager.allocatePhysicalPages(AddressPoolType::USER,3);
    char *p3 = (char *)memoryManager.allocatePhysicalPages(AddressPoolType::USER,1);
    char *p4 = (char *)memoryManager.allocatePhysicalPages(AddressPoolType::USER,2);
    char *p5 = (char *)memoryManager.allocatePhysicalPages(AddressPoolType::USER,1);

    memoryManager.releasePhysicalPages(AddressPoolType::USER,(int)p2,3);
}

```



```
memoryManager.releasePhysicalPages(AddressPoolType::USER, (int)p4, 2);

p2 = (char *)memoryManager.allocatePhysicalPages(AddressPoolType::USER, 2);

asm_halt();
}
```

线程对用户内存池进行申请和释放操作。可以看到线程首先进行了五次内存申请，分别申请了 1,3,1,2 和 1 个页面。随后，将申请的第二个和第四个内存块进行释放操作，最后再进行第六次内存申请，申请页面数为 2。

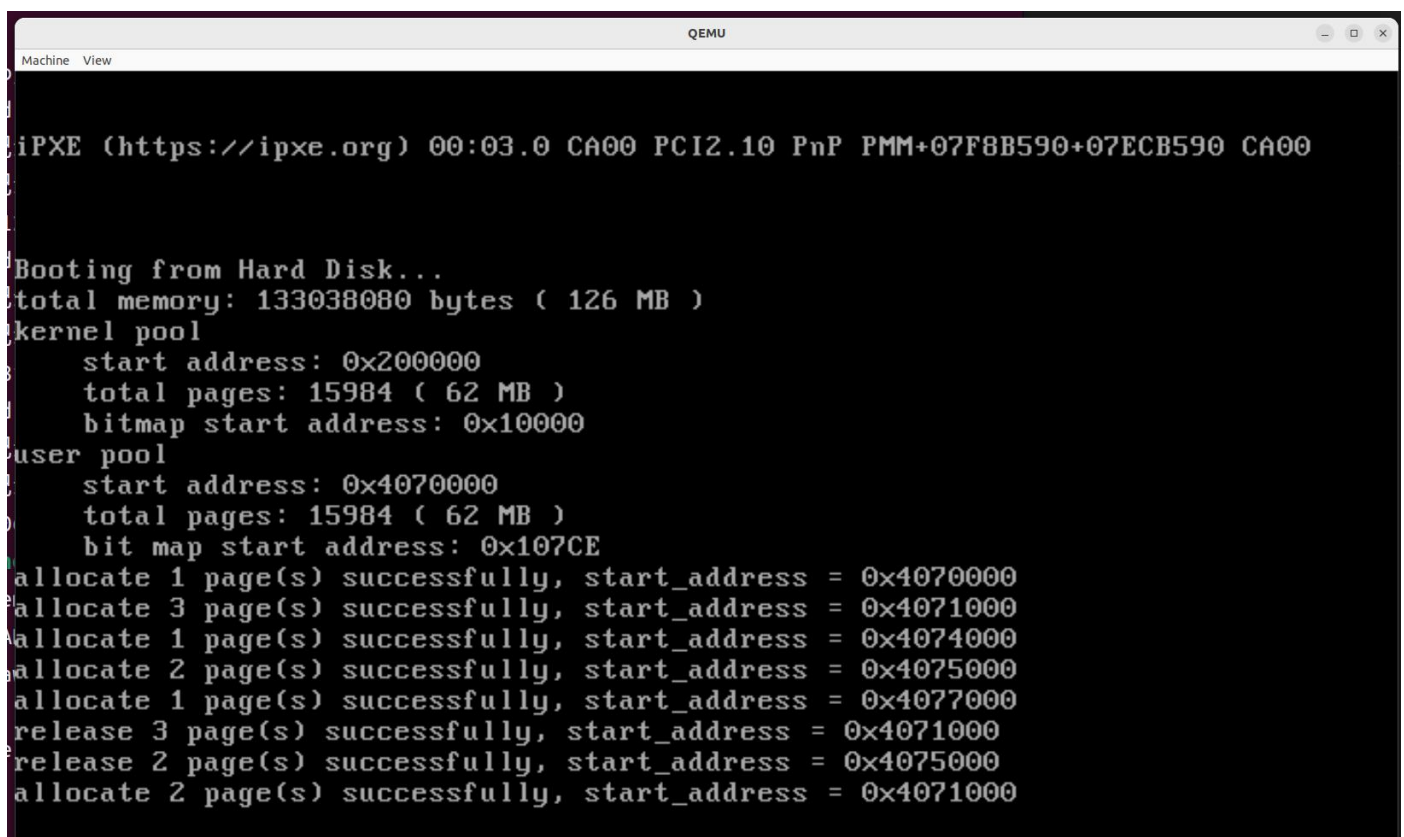
与实验 1 一样，在 `allocatePhysicalPages` 和 `releasePhysicalPages` 函数中添加了 `printf` 函数来更好的显示内存的申请和释放情况。

在虚拟机的 terminal 上进行 `make` 的相关操作即可得到相对应的 `qemu` 显示屏输出结果。

分别运行两次操作系统，分别采用 `first-fit` 和 `best-fit` 算法。

●实验结果展示：通过执行前述代码，可得下图结果。

首先是 `first-fit` 算法的结果：



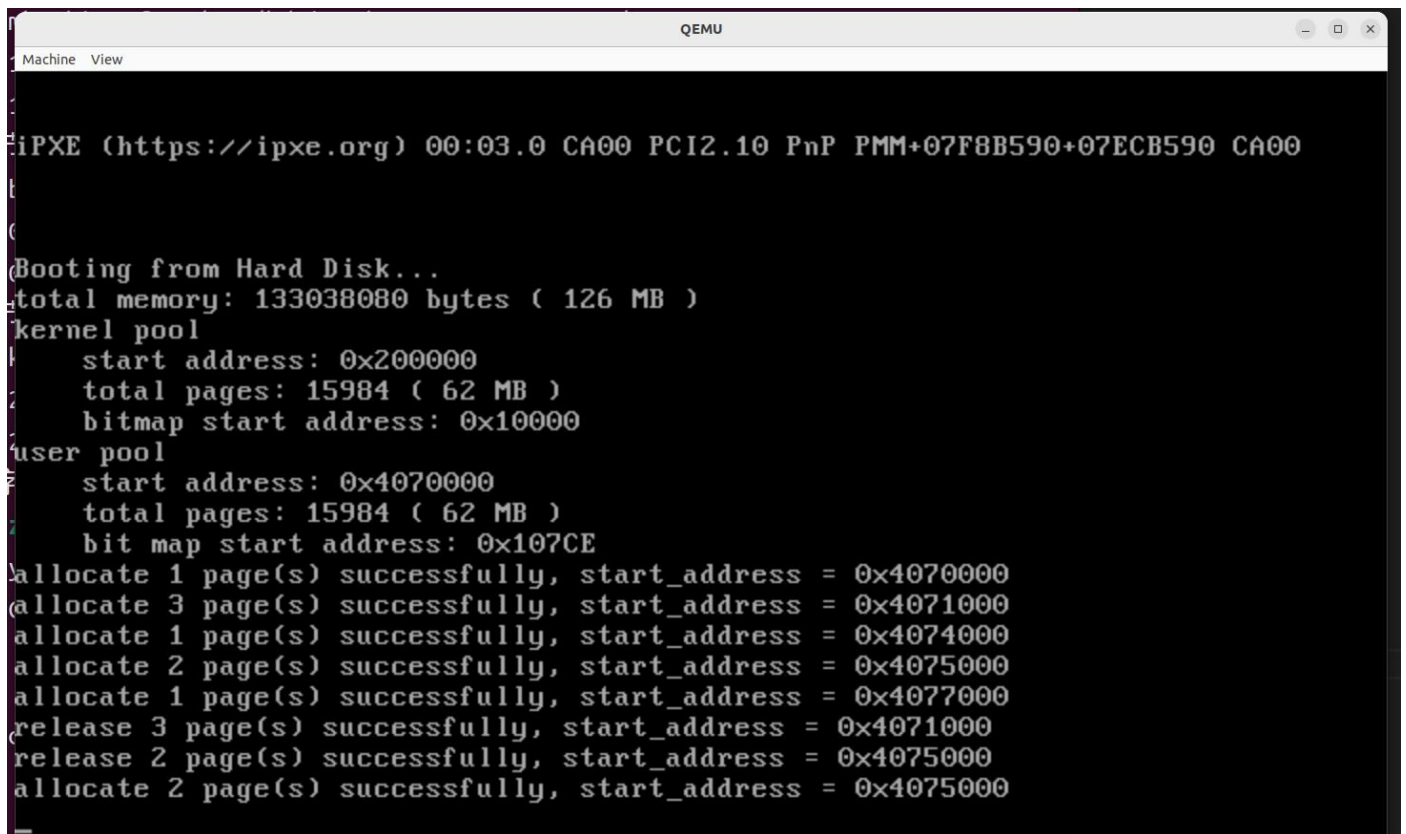
```
Machine View
QEMU

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00

Booting from Hard Disk...
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
allocate 1 page(s) successfully, start_address = 0x4070000
allocate 3 page(s) successfully, start_address = 0x4071000
allocate 1 page(s) successfully, start_address = 0x4074000
allocate 2 page(s) successfully, start_address = 0x4075000
allocate 1 page(s) successfully, start_address = 0x4077000
release 3 page(s) successfully, start_address = 0x4071000
release 2 page(s) successfully, start_address = 0x4075000
allocate 2 page(s) successfully, start_address = 0x4071000
```

可以看到，在前面的内存申请和释放操作后，第六次也就是最后一次的内存申请操作得到的内存起始地址为 `0x4071000`。这满足 `first-fit` 算法的工作原理的预期：在第二次和第四次申请的内存块释放后，从地址 `0x4071000` 和 `0x4075000` 分别空出了 3 个和 2 个空闲页面，在 `first-fit` 算法下，最后一次的内存申请将第一个满足申请长度条件的内存块分配给了线程，因此起始地址为 `0x4071000`

其次是 best-fit 算法的结果：



```
Machine View
QEMU

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00

(
Booting from Hard Disk...
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
allocate 1 page(s) successfully, start_address = 0x4070000
allocate 3 page(s) successfully, start_address = 0x4071000
allocate 1 page(s) successfully, start_address = 0x4074000
allocate 2 page(s) successfully, start_address = 0x4075000
allocate 1 page(s) successfully, start_address = 0x4077000
release 3 page(s) successfully, start_address = 0x4071000
release 2 page(s) successfully, start_address = 0x4075000
allocate 2 page(s) successfully, start_address = 0x4075000
```

通过上面的解释，发现对于相同的线程，这一回最后一次的内存申请操作得到的内存起始地址为 0x4075000。这满足 best-fit 算法的工作原理的预期：因为地址 0x4075000 开始正好有两个续的空闲页面，因此 best-fit 算法将这以内存块分配给线程。

以上我们便成功验证了我们实现的 best-fit 算法的正确性并且比较了 first-fit 和 best-fit 算法之间的差异之处。

----- 实验任务 3 -----

●任务要求：

Assignment 3

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，也可以提出自己的算法。

●思路分析：

在实现虚拟页内存管理后，现在来实现页面置换：这里使用 LRU 算法来实现页面置换：

LRU（Least Recently Used，最近最少使用）算法是一种常见的页面置换算法，用于管理缓存中的数据。其基本思想是：当缓存空间满时，选择最久未被访问的缓存数据进行淘汰，以便为新的数据腾出空间。

LRU 算法的实现通常依赖于一个双向链表结构，该链表按照页面访问的时间顺序进行排序。当一个页面被访问时，它会被移动到链表的头部。当需要进行页面置换时，链表尾部的页面（即最久未被访问的页面）会被弹出。

页面置换是指当内存已满，需要淘汰一些页面以便为新页面腾出空间的过程。页面置换算法的目标是最小化缺页次数，从而减少对外存的访问，提高系统的响应速度。

因为到此实验为止我们尚未学习如何把页面换出到磁盘中，因此本次实验不实现从磁盘换出换入页面的操作。并且，由于到本次实验为止还没有实现进程，因此无法实现当对已经被释放内存的虚拟地址进行访问时，产生缺页中断然后通过进程的內部表来进行磁盘操作进行页面的调度。一是我们没有实现进程，二是我们暂时没有实现磁盘的页面切换操作。

下面是我实现 LRU 算法的主要思路：

实现 LRU 算法，首先要实现一个双向链表结构，这个链表的每个链表项都代表着一个被分配的物理页，每个链表项链还要存储对应物理页的相应数据：物理页地址和对应虚拟页地址。因此实现这个链表需要进行相应的内存分配操作，以来保存链表储存的数据。

这里使用前面实验中已经实现的 List 类来实现双向链表功能，对于链表需要进行储存数据的要求，定义一个新的结构体 LRU_ListItem，里面包含 3 个变量：int 类型的 paddr 和 vaddr:分别储存物理页的地址(addr)和对应虚拟页的地址(vaddr);ListItem 类型的变量 tagLRUItem，作为双向链表的标志。实现一个宏定义：

```
Listltem2LRU(ADDRESS, LIST_ITEM) ((LRU_ListItem *)((int)(ADDRESS) - (int)&((LRU_ListItem *)0)->LIST_ITEM))
```

来进行从 ListItem 类型到 LRU_ListItem 类型的转换，以便访问和修改双向链表中的数据。

此外，实现一个 LRU 类，实现 LRU 算法的一系列操作函数以及管理与 LRU 双向链表有关的数据集。

LRU 类中定义了管理 LRU 链表的一系列数据结构：

- 1、List LRU_List：双向链表结构
- 2、enum AddressPoolType type：定义 LRU 管理的地址池类型
- 3、int len：LRU_list 管理的物理页数量
- 4、int LRU_start;分配 LRU_List 链表项的起始地址
- 5、bool LRU_SET_STATUS[20000];LRU_List 链表项内存分配标记表

LRU 类中定义的实现各项功能的函数：

- 1、void initialize(enum AddressPoolType type, int len, int addr);

初始 LRU 对象的基本信息。

- 2、void update();

根据物理内存访问情况更新 LRU 链表项（重点）

- 3、void push(int paddr, int vaddr);

当物理页分配时，创建 LRU_List 链表项并加入 LRU_List 中。

- 4、void pop();

需要进行页面置换时，将链表尾部的链表项弹出。

- 5、LRU_ListItem* allocateLRUitem();

分配 LRU_List 链表项的储存内存

- 6、void releaseLRUitem(LRU_ListItem *item);

释放 LRU_List 链表项的储存内存。

- 7、void printLRU();

打印 LRU_List,用于调试。

具体的函数实现以及调用方法在实验步骤中详细展示。

实现 LRU 类后，为用户内存池和内核内存池分别添加一个管理其的 LRU 对象，并对内存分配操

作函数进行对应修改，将 LRU 算法加入其中，实现页面置换操作。

●实验步骤：

首先，实现 LRU_ListItem 结构体和 LRU 类的定义声明，代码保存在 memory.h 中，如下所示：

```
struct LRU_ListItem
{
    int paddr;
    int vaddr;
    ListItem tagLRUItem;
};

class LRU
{
public:
    List LRU_List;
    enum AddressPoolType type;
    int len;
    int LRU_start;
    bool LRU_SET_STATUS[20000];
public:
    LRU();

    void initialize(enum AddressPoolType type, int len, int addr);

    void update();

    void push(int paddr, int vaddr);

    void pop();

    LRU_ListItem* allocateLRUitem();

    void releaseLRUitem(LRU_ListItem *item);

    void printLRU();
};
```

每个定义的函数和变量需实现的功能都在实验思路中阐述，下面将具体给出各个函数的具体实现代码：实现代码都保存在 mememory.cpp 下：

1、void initialize(enum AddressPoolType type, int len, int addr);

实现代码如下：

```
void LRU::initialize(enum AddressPoolType type,int len, int addr)
{
    this->type = type;
    this->len = len;
    this->LRU_start = addr;
```

```

    for(int i = 0;i < len;i++)
    {
        LRU_SET_STATUS[i] = false;
    }
}

```

此函数主要实现 LRU 对象的基本参数设置：设置 LRU 管理的地址池类型，管理物理页的数量以及分配 LRU_ListItem 的起始地址。

2、LRU_ListItem* allocateLRUitem();

```

LRU_ListItem* LRU::allocateLRUitem()
{
    for (int i = 0; i < len; ++i)
    {
        if (!LRU_SET_STATUS[i])
        {
            LRU_SET_STATUS[i] = true;
            return (LRU_ListItem *)((int)LRU_start + 16 * i);
        }
    }
    return nullptr;
}

```

此函数实现 LRU_ListItem 对象的分配。当调用此函数时，函数会遍历 LRU_SET_STATUS 状态表从中寻找没有被分配的地址(LRU_SET_STATUS[i] == 0 即为未分配)，寻找到后从对应地址开始分配大小为 16 字节 LRU_ListItem 对象并返回对应地址。

3、void LRU::releaseLRUitem(LRU_ListItem *item)

```

void LRU::releaseLRUitem(LRU_ListItem *item)
{
    int index = ((int)item - (int)LRU_start) / 16;
    LRU_SET_STATUS[index] = false;
}

```

此函数主要实现释放对应 LRU_ListItem 内存的功能。找到对应 LRU_ListItem 在状态表 LRU_SET_STATUS 的标号，将其设置为 false,新分配的 LRU_ListItem 即可覆盖它。

4、void LRU::printLRU()

```

void LRU::printLRU()
{
    printf("head: ");
    ListItem *temp = LRU_List.front();
    while(temp)
    {
        LRU_ListItem* tmp = ListItem2LRU(temp,tagLRUItem);
        printf("-> 0x%x ",tmp->paddr);
        temp = temp->next;
    }
}

```

```
    printf("end.\n");  
}
```

此函数实现打印 LRU_List 中管理的物理内存页的顺序情况，主要用于调试。

下面的三个函数是实现 LRU 页面置换算法的最重要的部分：

5、void LRU::push(int paddr, int vaddr)

```
void LRU::push(int paddr, int vaddr)  
{  
    printf("push: 0x%x, relateing vaddr: 0x%x\n",paddr, vaddr);  
    LRU_ListItem *tempLRUitem = allocateLRUitem();  
    tempLRUitem->paddr = paddr;  
    tempLRUitem->vaddr = vaddr;  
    LRU_List.push_front(&(tempLRUitem->tagLRUItem));  
}
```

址通过创建一个 LRU_ListItem 保存起来，并将 LRU_ListItem 中的 ListItem tagLRUItem 加入 LRU_List 的头部，即加入双向链表中来记录物理页的访问先后情况。

6、void LRU::pop()

```
void LRU::pop()  
{  
    if(LRU_List.empty())  
        return;  
    else  
    {  
        ListItem *temp = LRU_List.back();  
        LRU_ListItem* tmp = ListItem2LRU(temp,tagLRUItem);  
        while(tmp->vaddr == 0)  
        {  
            if(temp->previous && temp != &LRU_List.head)  
                temp = temp->previous;  
            else  
                return;  
            tmp = ListItem2LRU(temp,tagLRUItem);  
        }  
        printf("pop: 0x%x, relateing vaddr: 0x%x\n", tmp->paddr, tmp->vaddr);  
        memoryManager.releasePhysicalPages(type, tmp->paddr, 1);  
        int *pte = (int *)memoryManager.toPTE(tmp->vaddr);  
        *pte = *pte & 0xfffffffffe;  
        releaseLRUitem(tmp);  
        LRU_List.erase(temp);  
    }  
}
```

当物理内存满时，需要将 LRU_List 尾部的链表项给弹出，来提供更多内存。因为在二级分页机

制中一级页表也从物理页中分配，所以，为了避免将一级页表给弹出，在给一级页表分配内存的时候，设置其对应的虚拟地址为 0，那么在 `pop()` 函数中，遇到虚拟地址为 0 的 `LRU_ListItem` 项将会直接跳过它并访问后面的链表项。在确定弹出链表项并进行弹出操作时，我们要进行三步操作：

- 1、将对应的物理页释放
- 2、将物理页对应的虚拟页的页表项的存在位设置为 0
- 3、释放对应的 `LRU_ListItem`

以上我们便完成了 `pop()` 操作。

7、`void LRU::update()`

接下来是实现 LRU 算法的最重要的一部分：我们知道，LRU 算法的核心功能为：如果一个物理页面最近被访问，这个物理页面对应的链表项要放在链表的头部。我们知道，当通过分配了物理内存的虚拟地址来向其中添加数据时便进行了一个内存访问操作，可是我们的操作系统如何检测这一个内存访问操作发生了呢？我从实验材料中发现，页表项中存在一个 A 位（访问位），1 表示被访问过，0 表示未被访问，其由 CPU 自动置位。当我们通过虚拟地址访问其对应的物理地址时，CPU 会自动修改这个物理页对应的页表项的 A 位，这就为我实现检测内存访问是否发生提供了途径：

实现一个 `update()` 函数，把这个函数放入我们处理时间中断的函数中，这样操作系统便可以按一定的时间周期检查 `LRU_List` 中已经分配的物理内存是否被访问。

在 `update()` 函数中，会从 `LRU_List` 尾部开始依次检查中各个链表项的虚拟地址对应的页表项的 A 位是否被 CPU 置为 1，如果被置为了 1，说明此虚拟内存对应的物理页近期被访问过，于是将其移动到链表的头部。移动完成后，将页表项的 A 位重新置为 0，这样当此物理页再次被访问后，`update()` 函数就可以再次将此物理页对应的链表放在链表头部。

在实现将页表项的 A 位重新置为 0 来检测此物理页下次被访问的记录的过程中遇到了一些问题：虽然在操作系统中将页表项的 A 位重新置为零，但当再次通过虚拟地址访问同一个物理页时，CPU 并没有将此物理页的页表项的 A 位重新置为零，这和 CPU 所描述的功能不同。在查阅相关资料后，发现是因为虽然操作系统中将页表项的 A 位重新置为零，但 CPU 中的 TLB 保存的页表项并没有及时得到更新，CPU 的 TLB 中仍然保存着原来 A 位设置为 1 的页表项。因此我们要手动实现 TLB 的刷新。这里涉及到汇编指令 `invlpg` 的使用：

`invlpg` 指令用于使特定虚拟地址对应的页表项失效，从而强制 CPU 重新从内存中读取该页表项。

我们实现一个汇编函数 `asm_reflash_TLB` 供 `update()` 函数调用。`asm_reflash_TLB` 中使用 `invlpg` 指令，汇编函数读取 C 函数中传递的虚拟地址，并使用 `invlpg` 指令实现 TLB 的更新：

`asm_reflash_TLB` 函数的实现代码保存在 `asm_utils.asm` 中，代码如下

```
;void asm_reflash_TLB(uint32 virtualaddress)
asm_reflash_TLB:
    push ebp
    mov ebp, esp

    mov eax, [ebp + 4 * 2]
    invlpg [eax]

    pop ebp
    ret
```

以上我们便解决了 CPU 不更新 A 位的问题。

下面是 update()的代码:

```
void LRU::update()
{
    if(type == AddressPoolType::KERNEL)
    {
        if(LRU_List.size() == 0)
            return;

        ListItem *temp = LRU_List.back();
        while(temp != &LRU_List.head)
        {
            LRU_ListItem* tmp = ListItem2LRU(temp,tagLRUItem);
            int vaddr = tmp->vaddr;
            int *pde = (int *)memoryManager.toPDE(vaddr);
            int *pte = (int *)memoryManager.toPTE(vaddr);
            if(*pde == 0 || *pte == 0 || vaddr == 0)
            {
                temp = temp->previous;
                continue;
            }
            if((*pte & 0x00000020) == 0x00000020)
            {
                *pte = *pte & 0xffffffff;
                asm_reflash_TLB(vaddr);

                LRU_List.erase(temp);
                LRU_List.push_front(temp);
            }
            temp = temp->previous;
        }
    }
}
```

现在我们已经完成了 LRU 进行将最近访问的页面代表的链表项放在链表头部的功能，下面我们通过修改 MemoryManager 类的函数以及添加变量来使我们的操作系统的内存管理实现 LRU 页面置换:

首先在 MemoryManager 类中添加两个分别管理内核地址池和用户地址池的两个 LRU 数据对象:

```
LRU LRU_kernel;
LRU LRU_user;
```

接着我们在 void MemoryManager::initialize() 中添加初始化 LRU_kernel 和 LRU_user 的代码段:

```
int kernel_LRU_start = 0x40000; //设置 kernel_LRU 存放链表项的起始地址
// kernel_LRU_start 往后 kernelPages * 16 个 字节 存放 user_LRU_start
int user_LRU_start = kernel_LRU_start + kernelPages * 16;

LRU_kernel.initialize(AddressPoolType::KERNEL,kernelPages,kernel_LRU_start);
LRU_user.initialize(AddressPoolType::USER,userPages,user_LRU_start);
```


接着我们修改 `int MemoryManager::allocatePages(enum AddressPoolType type, const int count)` 函数，实现 LRU 页面置换：

```
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 -1;
    }

    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)
        {
            //printf("allocate physical page 0x%x\n", 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
        {
            int num = count - i;
            while(num--)
            {
                if (type == AddressPoolType::KERNEL)
                {
                    LRU_kernel.pop();
                }
                else if (type == AddressPoolType::USER)
                {
                    LRU_user.pop();
                }
            }
            i--;
            vaddress -= PAGE_SIZE;
            continue;
        }
    }
}
```

```

        // 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;
        }
        if (type == AddressPoolType::KERNEL)
            LRU_kernel.push(physicalPageAddress, vaddress);
        else if (type == AddressPoolType::USER)
            LRU_user.push(physicalPageAddress, vaddress);
    }

    return virtualAddress;
}

```

随后，因为我们在 `bool MemoryManager::connectPhysicalVirtualPage(const int virtualAddress, const int physicalPageAddress)` 函数中也进行了物理页的分配，因此也要添加对应的 LRU 的 push 操作：修改后的部分代码段如下：

```

.....
    // 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;
        LRU_kernel.push(page, 0);
    }
.....

```

以上我们对 MemoryManager 的修改大功告成，最后我们将 `update()` 函数放入处理时间中断函数中：

```

extern "C" void c_time_interrupt_handler()
{
    PCB *cur = programManager.running;

    if (cur->ticks)
    {
        --cur->ticks;
        ++cur->ticksPassedBy;
    }
    else
    {
        programManager.schedule();
    }
    memoryManager.LRU_kernel.update();
    memoryManager.LRU_user.update();
}

```

以上，我们的操作系统已经实现了 LRU 页面置换算法，下面是验证算法正确性的测例的设计：

首先，我们来验证操作系统实现的 LRU 页面置换是否能够将最近访问的物理页面代表的链表项放置到链表头部，设计的测试案例放在 `setup.cpp` 文件中第一个线程里，代码段如下：

```
char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
char *p4 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
char *p5 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);

memoryManager.LRU_kernel.printLRU();
p3[5] = '8';
printf("visit p3:0x%x\n", (int)p3);
int timedelay = 10000000;
while(timedelay--){}
memoryManager.LRU_kernel.printLRU();
p2[1] = '3';
printf("visit p2:0x%x\n", (int)p2);
timedelay = 10000000;
while(timedelay--){}
memoryManager.LRU_kernel.printLRU();
p3[3] = '2';
printf("visit p3:0x%x\n", (int)p3);
timedelay = 10000000;
while(timedelay--){}
memoryManager.LRU_kernel.printLRU();
```

可以看到，我们首先进行了五次虚拟页的分配，每次都分配一页，并将虚拟页的地址值以 `char` 类型分别赋值给 `p1-p5`。随后通过使用 `p3` 和 `p2` 实现通过访问虚拟地址来向物理页赋值实现物理页的访问操作，在访问操作进行前后分别使用 `LRU` 类中实现的 `printLRU()` 函数来打印 `LRU` 对应链表的物理页连接顺序情况。期间进行了延迟操作，因为在访问内存后到 `update()` 函数更新链表顺序存在一定的时间间隔，进行延迟操作能更好的展示测试结果。

接下来我们要测试当物理内存已经分配满时，我们的 `LRU` 页面置换能否弹出最久未被访问：也就是链表尾部的链表项来实现页面置换操作。

因为虚拟地址一般远大于物理地址，为了实现测试，我们首先要将虚拟内核内存进行扩容。

这里将虚拟内核地址池扩大一倍，从 62MB 扩容到 124MB

修改 `void MemoryManager::initialize()` 中有关 `kernelVirtual` 初始化的代码实现扩容操作：

```
.....
kernelVirtual.initialize(
    (char *)kernelVirtualBitMapStart,
    kernelPages * 2, //扩大一倍
    KERNEL_VIRTUAL_START);
.....
```

进行完虚拟内核地址池的扩容后，下面是我们的测试代码段：

```
timedelay = 1000000000;
while(timedelay--){

    for(int i = 0;i < 1600;i++)
    {
        memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
    }
}
```

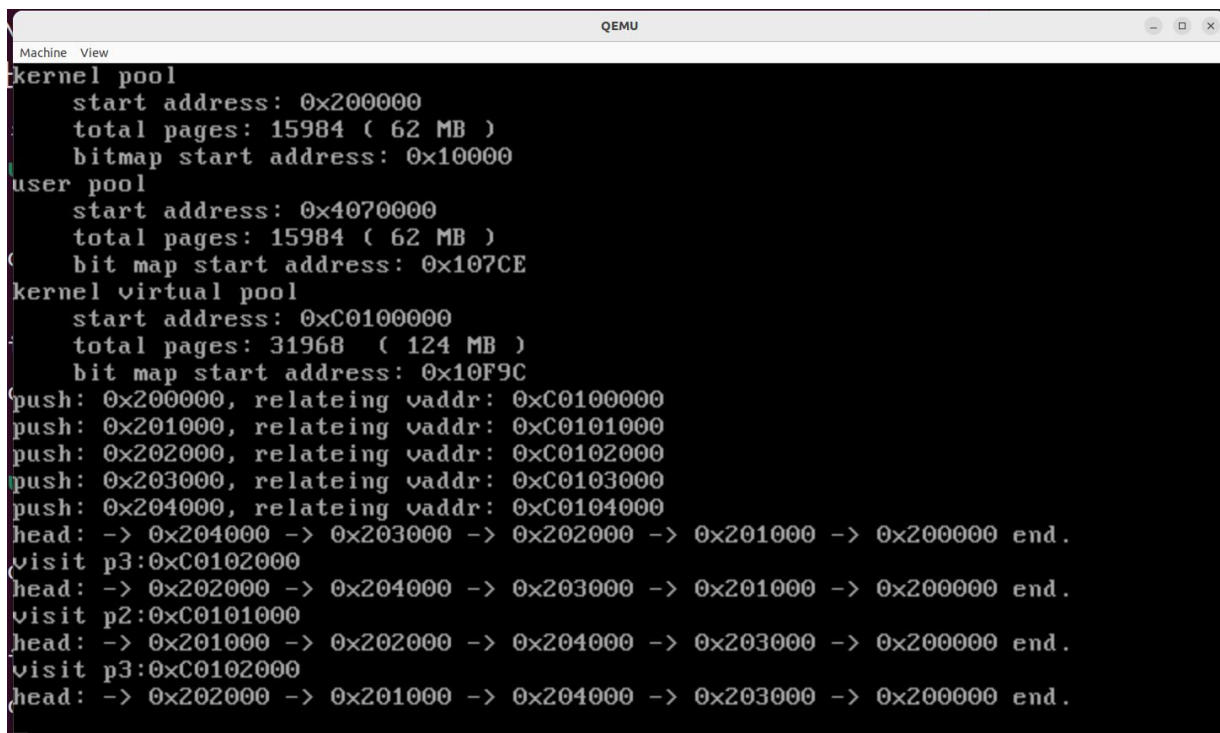
第一个延迟操作是为了和第一个测试案例分开执行，以便观察第一个测试案例的执行结果。

在这个测试案例中，我们进行 1600 次页面分配，每次申请 10 个虚拟页面。这样我们总共申请了 16000 个虚拟页面，对应 16000 个物理页面。但是从实验一中我们知道，我们的物理内核地址池只有 15986 个物理页面，因此我们的 LRU 页面置换算法会弹出最久未被访问的页面。

至此我们完成了测试案例的设计，在虚拟机中的 terminal 使用 make 相关指令编译运行操作系统，即可观察到测试结果。

●实验结果展示：通过执行前述代码，可得下图结果。

下面是测试案例一的结果：



```
Machine View
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: 31968 ( 124 MB )
  bit map start address: 0x10F9C
push: 0x200000, relateing vaddr: 0xC0100000
push: 0x201000, relateing vaddr: 0xC0101000
push: 0x202000, relateing vaddr: 0xC0102000
push: 0x203000, relateing vaddr: 0xC0103000
push: 0x204000, relateing vaddr: 0xC0104000
head: -> 0x204000 -> 0x203000 -> 0x202000 -> 0x201000 -> 0x200000 end.
visit p3:0xC0102000
head: -> 0x202000 -> 0x204000 -> 0x203000 -> 0x201000 -> 0x200000 end.
visit p2:0xC0101000
head: -> 0x201000 -> 0x202000 -> 0x204000 -> 0x203000 -> 0x200000 end.
visit p3:0xC0102000
head: -> 0x202000 -> 0x201000 -> 0x204000 -> 0x203000 -> 0x200000 end.
```

可以看到，在进行内存访问前，物理内存的链表项在链表中的排序为

head->0x204000->0x203000->0x202000->0x201000->0x200000

当对 p3 对应的物理地址页 0x202000 进行访问操作后，物理内存的链表项在链表中的排序变为：

head->0x202000->0x204000->0x203000->0x201000->0x200000

当对 p2 对应的物理地址页 0x201000 进行访问操作后，物理内存的链表项在链表中的排序变为：

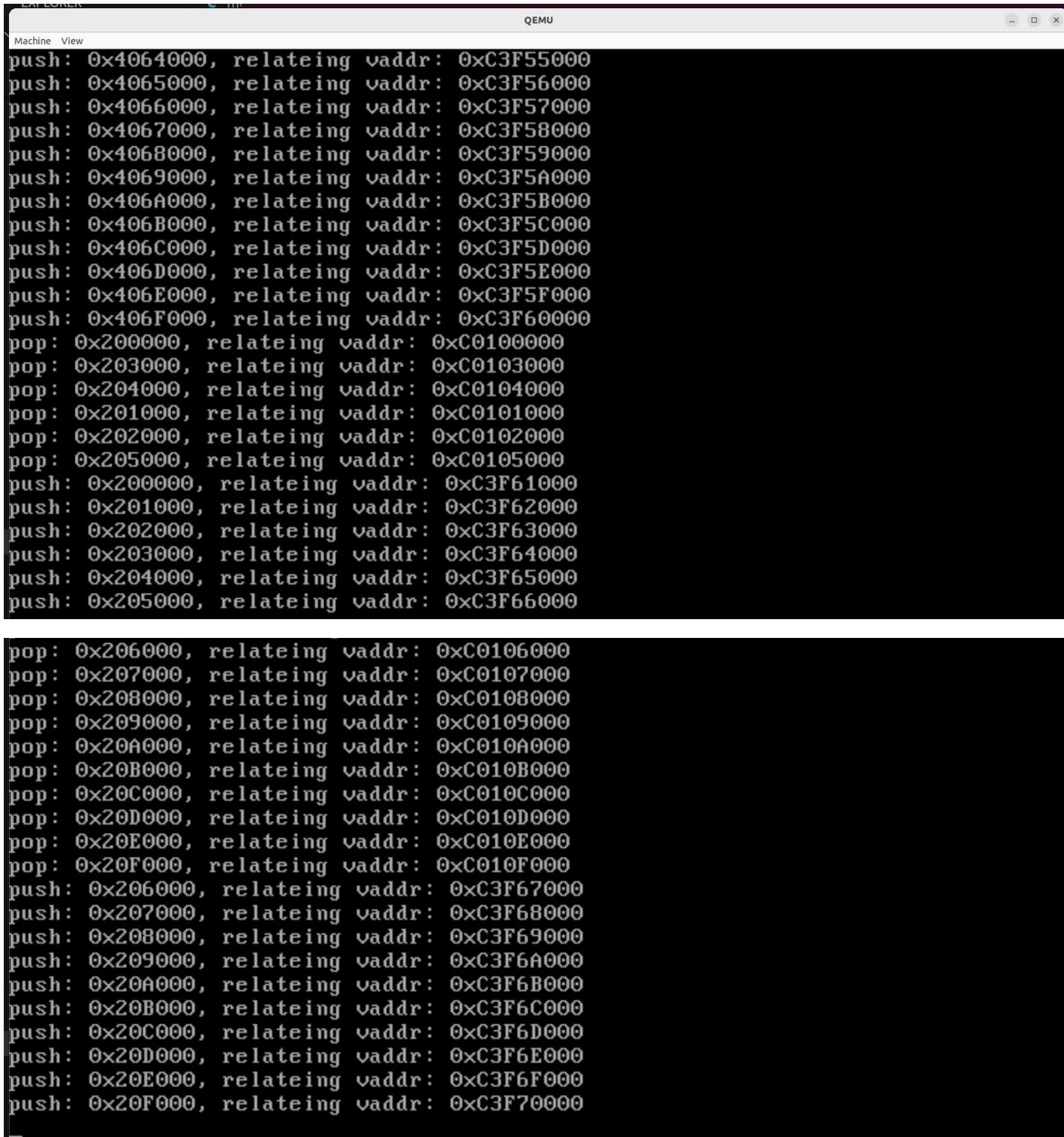
head->0x201000->0x202000->0x204000->0x203000->0x200000

当再一次对 p3 对应的物理地址页 0x202000 进行访问操作后，物理内存的链表项在链表中的排序变为：

head->0x202000->0x201000->0x204000->0x203000->0x200000

有上述的实验结果可以得出，操作系统的 LRU 页面置换算法实现了将最近访问的物理页代表的链表项放到链表头部的操作。

下面是测例 2 的结果：



```
Machine View
push: 0x4064000, relateing vaddr: 0xC3F55000
push: 0x4065000, relateing vaddr: 0xC3F56000
push: 0x4066000, relateing vaddr: 0xC3F57000
push: 0x4067000, relateing vaddr: 0xC3F58000
push: 0x4068000, relateing vaddr: 0xC3F59000
push: 0x4069000, relateing vaddr: 0xC3F5A000
push: 0x406A000, relateing vaddr: 0xC3F5B000
push: 0x406B000, relateing vaddr: 0xC3F5C000
push: 0x406C000, relateing vaddr: 0xC3F5D000
push: 0x406D000, relateing vaddr: 0xC3F5E000
push: 0x406E000, relateing vaddr: 0xC3F5F000
push: 0x406F000, relateing vaddr: 0xC3F60000
pop: 0x200000, relateing vaddr: 0xC0100000
pop: 0x203000, relateing vaddr: 0xC0103000
pop: 0x204000, relateing vaddr: 0xC0104000
pop: 0x201000, relateing vaddr: 0xC0101000
pop: 0x202000, relateing vaddr: 0xC0102000
pop: 0x205000, relateing vaddr: 0xC0105000
push: 0x200000, relateing vaddr: 0xC3F61000
push: 0x201000, relateing vaddr: 0xC3F62000
push: 0x202000, relateing vaddr: 0xC3F63000
push: 0x203000, relateing vaddr: 0xC3F64000
push: 0x204000, relateing vaddr: 0xC3F65000
push: 0x205000, relateing vaddr: 0xC3F66000

pop: 0x206000, relateing vaddr: 0xC0106000
pop: 0x207000, relateing vaddr: 0xC0107000
pop: 0x208000, relateing vaddr: 0xC0108000
pop: 0x209000, relateing vaddr: 0xC0109000
pop: 0x20A000, relateing vaddr: 0xC010A000
pop: 0x20B000, relateing vaddr: 0xC010B000
pop: 0x20C000, relateing vaddr: 0xC010C000
pop: 0x20D000, relateing vaddr: 0xC010D000
pop: 0x20E000, relateing vaddr: 0xC010E000
pop: 0x20F000, relateing vaddr: 0xC010F000
push: 0x206000, relateing vaddr: 0xC3F67000
push: 0x207000, relateing vaddr: 0xC3F68000
push: 0x208000, relateing vaddr: 0xC3F69000
push: 0x209000, relateing vaddr: 0xC3F6A000
push: 0x20A000, relateing vaddr: 0xC3F6B000
push: 0x20B000, relateing vaddr: 0xC3F6C000
push: 0x20C000, relateing vaddr: 0xC3F6D000
push: 0x20D000, relateing vaddr: 0xC3F6E000
push: 0x20E000, relateing vaddr: 0xC3F6F000
push: 0x20F000, relateing vaddr: 0xC3F70000
```



```

pop: 0x210000, relateing vaddr: 0xC0110000
pop: 0x211000, relateing vaddr: 0xC0111000
pop: 0x212000, relateing vaddr: 0xC0112000
pop: 0x213000, relateing vaddr: 0xC0113000
pop: 0x214000, relateing vaddr: 0xC0114000
pop: 0x215000, relateing vaddr: 0xC0115000
pop: 0x216000, relateing vaddr: 0xC0116000
pop: 0x217000, relateing vaddr: 0xC0117000
pop: 0x218000, relateing vaddr: 0xC0118000
pop: 0x219000, relateing vaddr: 0xC0119000
push: 0x210000, relateing vaddr: 0xC3F71000
push: 0x211000, relateing vaddr: 0xC3F72000
push: 0x212000, relateing vaddr: 0xC3F73000
push: 0x213000, relateing vaddr: 0xC3F74000
push: 0x214000, relateing vaddr: 0xC3F75000
push: 0x215000, relateing vaddr: 0xC3F76000
push: 0x216000, relateing vaddr: 0xC3F77000
push: 0x217000, relateing vaddr: 0xC3F78000
push: 0x218000, relateing vaddr: 0xC3F79000
push: 0x219000, relateing vaddr: 0xC3F7A000

```

可以从测试结果看到，当线程准备进行对虚拟页面地址 0xC3F61000 进行物理页面分配时，因为物理内核地址空间已经被分配完毕，因此 LRU 页面置换算法将此次虚拟页面分配还需要分配的 6 个虚拟页面对应的 6 个物理页面，通过释放链表尾部对应的 6 个物理页面来腾出空间满足最新的虚拟页面分配。可以看到，LRU_list 将位于链表尾部的：

.....->0x205000->0x201000->0x202000->0x204000->0x203000->0x200000->end

依次释放，这符合我们在测例 1 中最后链表尾部的排列顺序。

在此后的每 10 个虚拟页面分配中，LRU 页面置换都实现了现将满物理内存池最久未被访问，也就是处于链表尾部的 10 个物理页释放，再进行新的虚拟页同物理页的对应分配。

经过两个测试案例，我实现的 LRU 页面置换算法被证明是正确的。

----- 实验任务 4 -----

●任务要求：

✎Assignment 4

复现“虚拟页内存管理”一节的代码，完成如下要求。

- 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。
- （不做要求，对评分没有影响）如果你有想法，可以在自己的理解的基础上，参考ucore，《操作系统真象还原》，《一个操作系统的实现》等资料来实现自己的虚拟页内存管理。在完成之后，你需要指明相比较于本教程，你的实现的虚拟页内存管理的特点所在。

●思路分析：

虚拟页内存管理的代码已经在实验材料中给出，复制到虚拟机上进行编译运行即可。

首先结合给出的代码分析虚拟页内存分配的三步过程和虚拟页内存释放

在这次实验任务中，通过设计一个进行虚拟内存申请和释放的线程来分析虚拟页内存管理实现的正确性。通过使用 printf 函数来更好的显示虚拟内存管理的实现的正确性。

●实验步骤:

下面是结合给出的代码分析虚拟页内存分配的三步过程和虚拟页内存释放

实现虚拟页内存分配，首先调用 `int MemoryManager::allocatePages(enum AddressPoolType type, const int count)` 函数，输入调用虚拟页面的类型（内核或用户）以及一次调用的页数，该函数在分配成功的情况下会放回分配虚拟页面的起始地址，分配失败时会放回 0。下面是此函数的代码：

```
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)
        {
            printf("allocate physical page 0x%x\n", 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;
}
```

可以看到此函数分别依次执行虚拟页内存分配的三步过程：

第一步：从虚拟地址池中分配若干虚拟页：

首先通过 `allocateVirtualPages()` 函数分配虚拟页面，并得到返回的分配的虚拟页面的首地址
下面是 `allocateVirtualPages()` 函数的实现代码

```
int MemoryManager::allocateVirtualPages(enum AddressPoolType type, const int count)
{
    int start = -1;

    if (type == AddressPoolType::KERNEL)
    {
        start = kernelVirtual.allocate(count);
    }

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

可以看到，得到分配的虚拟页面地址的方法和得到物理页面地址的方法一致，都是通过在对应的地址池中通过访问对应保存的 **BitMap** 来实现页面的分配。

第二步：从物理地址池中分配一个物理页

成功得到分配的虚拟页面的首地址后，通过 `for` 循环为每一个虚拟页面地址找到一个物理页面地址。这时我们通过 `allocatePhysicalPages(type, 1)` 函数每一次在对应类型的物理地址池中分配一个空闲的物理页，得到其对应的地址，这样便可以进行接下来的第三步过程。

下面是 `allocatePhysicalPages(type, 1)` 函数的代码：

```
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;
}
```

可以看到代码和我们在实验 1 所看到的一模一样。因为从对应类型的物理地址池分配一个页面的过程已经解释过了，即通过 **BitMap** 来找到一个空闲的物理页并放回它的地址，这里不再赘述。

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

在成功得到分配的虚拟页面的首地址并得到一个物理页表的地址后，我们来为一个虚拟页面的地址与这个物理页面地址建立联系：主要通过 `connectPhysicalVirtualPage(vaddress, physicalPageAddress)` 函数来实现将虚拟页面和物理页面的一一对应。下面是 `connectPhysicalVirtualPage(vaddress, physicalPageAddress)` 的代码：

```
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;
}
```

可以看到，此函数在得到需要建立联系的虚拟页面地址和物理页面地址后，通过 `toPDE(virtualAddress)` 和 `toPTE(virtualAddress)` 两个转换函数，得到虚拟地址对应的页目录表项和页表项的访问虚拟地址，做这样的转换是因为我们的操作系统已经实现了二级分页机制，任何对地址内容的访问都会被 CPU 解释为虚拟地址。随后，我们对虚拟页面地址对应的页目录表项和页表项进行修改，将虚拟页面的地址和物理页面地址建立联系，这样在我们访问虚拟页面时 CPU 能找到对应的物理页面。

此函数在连接成功时放回 `true`，如果连接失败放回 `false`。如何放回了 `false`，说明无法建立虚拟页面同物理页面的联系，那么代表着整个分配虚拟页面任务的失败，于是会释放在这之前已经分配好的页面。

下面结合代码来解释释放页面的操作：

释放页面用到两个函数：`void MemoryManager::releasePages(enum AddressPoolType type, const int`

`virtualAddress, const int count)` 和 `void MemoryManager::releaseVirtualPages(enum AddressPoolType type, const int vaddr, const int count)`函数

这两个函数实现的功能不同：`void MemoryManager::releasePages(enum AddressPoolType type, const int virtualAddress, const int count)`函数的代码如下所示：

```
void MemoryManager::releasePages(enum AddressPoolType type, const int virtualAddress, const int count)
{
    int vaddr = virtualAddress;
    int *pte;
    for (int i = 0; i < count; ++i, vaddr += PAGE_SIZE)
    {
        // 第一步，对每一个虚拟页，释放为其分配的物理页
        releasePhysicalPages(type, vaddr2paddr(vaddr), 1);
        printf("release physical page 0x%x\n", vaddr2paddr(vaddr));
        // 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;
    }

    // 第二步，释放虚拟页
    releaseVirtualPages(type, virtualAddress, count);
}
```

可以看到，此函数实现的是释放已经建立起虚拟页和物理页联系页面的功能。首先对于每一个虚拟页，先释放其对应的物理页，在对虚拟页对应的页表项设置为零，避免释放后的再次利用。

可以看到，在此函数中使用了 `void MemoryManager::releaseVirtualPages(enum AddressPoolType type, const int vaddr, const int count)`函数，此函数的代码如下所示：

```
void MemoryManager::releaseVirtualPages(enum AddressPoolType type, const int vaddr, const int count)
{
    if (type == AddressPoolType::KERNEL)
    {
        kernelVirtual.release(vaddr, count);
    }
}
```

可以看到，此函数完成了将对应的虚拟页面从对应类型的虚拟地址池中释放的操作。

至此，我结合给出的代码分析了虚拟页内存分配的三步过程和虚拟页内存释放操作。

下面是测试虚拟内存管理的实现的正确性的线程代码，这里首先进行三次虚拟页面分配，申请分配页数分别为3、2和1页，随后对第二次分配的2个虚拟页面进行释放，再申请3个虚拟页面，再释放，最后再申请4个虚拟页面。

线程代码如下所示：

```
void first_thread(void *arg)
{
    // 第 1 个线程不可以返回
    // stdio.moveCursor(0);
    // for (int i = 0; i < 25 * 80; ++i)
    // {
    //     stdio.print(' ');
    // }
    // stdio.moveCursor(0);

    char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 3);
    char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 2);
    char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);

    printf("allocate virtual page(s) start from 0x%x,0x%x,0x%x\n",p1, p2, p3);

    memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 2);
    printf("release virtual page(s) start from 0x%x\n",p2);

    p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 3);
    printf("allocate virtual page(s) start from 0x%x\n",p2);

    memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 3);
    printf("release virtual page(s) start from 0x%x\n",p2);

    p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 4);
    printf("allocate virtual page(s) start from 0x%x\n",p2);

    asm_halt();
}
```

同样，为了更好的验证虚拟页面内存分配的正确性，我在分配和释放物理内存的函数中添加了对应的 `printf` 函数，可以更好的观察内存分配情况。

在虚拟机的 `terminal` 中使用 `make` 指令启动操作系统查看测试结果。

●实验结果展示：通过执行前述代码，可得下图结果。

线程运行结果如下所示：

```
Machine View
bit map start address: 0x10F9C
allocate physical page 0x200000
allocate physical page 0x201000
allocate physical page 0x202000
allocate physical page 0x203000
allocate physical page 0x204000
allocate physical page 0x205000
allocate virtual page(s) start from 0xC0100000,0xC0103000,0xC0105000
release physical page 0x203000
release physical page 0x204000
release virtual page(s) start from 0xC0103000
allocate physical page 0x203000
allocate physical page 0x204000
allocate physical page 0x206000
allocate virtual page(s) start from 0xC0106000
release physical page 0x203000
release physical page 0x204000
release physical page 0x206000
release virtual page(s) start from 0xC0106000
allocate physical page 0x203000
allocate physical page 0x204000
allocate physical page 0x206000
allocate physical page 0x207000
allocate virtual page(s) start from 0xC0106000
```

可以从测试结果中看到，我们的前三次虚拟页面分配首先在虚拟内存页面池中分配虚拟页面，其次在物理内存池中分配物理页面与每一个虚拟页面联系起来，地址为 0xC0100000-0xC0105000 的虚拟页面分别和地址为 0x200000-0x205000 的物理页面相联系。

在对第二次分配的页面释放后，地址为 0x203000 和 0x204000 物理页面同时被释放。

再次分配 3 个虚拟页面时，地址为 0x203000、0x204000 和 0x206000 的物理页面被联系分配

注意到这一次虚拟页面分配的起始地址为 0xC0106000，这是因为在联系分配虚拟地址是，因为虚拟内核地址池使用的 first-fit 算法，在前一步释放的两个连续虚拟页面的大小不足以满足连续分配 3 个虚拟页面的要求，所以这一次更配的虚拟页面起始地址为 0xC0106000。

因为在对虚拟页面同物理页面建立联系时，物理页面是一个一个分配的，因此物理的分配可以是不连续的，这便是为什么地址为 0xC0106000 到 0xC0108000 的虚拟页面对应的物理页面的地址分别为 0x203000、0x204000 和 0x206000。这也满足虚拟页面和物理页面并不总是连续对应的关系。

随后释放这 3 个虚拟页面，地址为 0x203000、0x204000 和 0x206000 的物理页面也被释放。

最后分配 4 个虚拟页面，地址为 0x203000、0x204000、0x206000 和 0x207000 的物理页面被联系分配。

Section 3 实验总结与心得体会

1、通过本次对于内存管理的实验，我对如何通过操作系统对内存进行管理有了清晰的认识，知晓了如何获取操作系统可管理的总内存大小。通过对位图和地址池的了解，我对操作系统管理内存的形式，如何便捷的实现内存的管理有了更加深入的了解。

2、通过对二级分页机制的了解，我对于如何更好的处理和分配内存有了更加全面的理解，理解了使用各种管理内存机制的优势，明白了二级分页机制能够有效地帮助我们来进行程序的动态重定位，并对于后面开启虚拟页面内存管理有很大的帮助。

3、通过对虚拟页内存管理的了解与实现，知晓了实现虚拟内存的好处以及如何将虚拟内存页面同物理内存页面通过二级分页机制联系起来，建立虚拟页和物理页之间的对应关系，从而真正实现虚拟页内存管理。

4、在实现 LRU 页面置换算法的途中，我遇到了许多问题:最先是如何检测物理页面是否被访问，这是实现 LRU 算法的关键。在仔细阅读实验资料后，我发现在通过虚拟地址访问对应的物理页面时，对应页表项的 A 位会被 CPU 自动设置为 1，这就为检测物理页面是否被访问提供了途径。而在将 A 位置零后再次访问页表项对应的物理页面时，遇到了 CPU 没有将页表项的 A 位设置为 1 的问题，通过查阅资料，发现是 CPU 内部的 TLB 并没有在操作系统将 A 位置零后将页表项更新导致的。解决这个问题我使用到了 `invlpg` 汇编指令来实现对于虚拟地址的 TLB 更新，解决了 CPU 不更新 A 位的问题。

5、通过实现 LRU 页面置换算法以及学习虚拟页内存管理，我对实现进程并进行内存分配的实验内容有了基本知识的积累与学习，为接下来的实验任务的完成有了更多的信心。

Section 4 对实验的改进建议和意见

实验任务的描述可以更加清晰，比如应该实现什么样的关键功能，什么功能暂时不用实现。

Section 5 附录：参考资料清单

https://github.com/linggm3/SYSU_Operate-System-Lab/blob/main/Lab7_%E5%86%85%E5%AD%98%E7%AE%A1%E7%90%86/Code/assignment3/src/kernel/memory.cpp

<https://jishuzhan.net/article/1892926228726288385>

https://blog.csdn.net/weixin_30243533/article/details/96893089

<https://gitee.com/apshuang/sysu-2025-spring-operating-system/tree/master/lab7>