



本科生实验报告

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: 熊彦钧

学生学号: 23336266

实验地点: 实验楼 B203

实验成绩:

报告时间: 2025 年 5 月 27 日

Section 1 实验概述

- 1. 复现参考代码，实现二级分页机制，并能够在虚拟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. 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。
- 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。
- 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 （对应 assignment1） -----

- 任务要求：复现参考代码，实现二级分页机制，并能够在虚拟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。
- 实验步骤：

首先。我们需要理解，二级分页机制的核心就是：允许程序使用连续的虚拟地址空间，而实际的物理内存可以分散或不连续，从而解决内存管理问题，而我们需要将虚拟地址动态映射成物理地址。

在二级页表结构中，我们有页目录表，页表，物理页三个数据结构。其中页目录表是一级表，包含 1024 个页目录项（PDE），每个 PDE 指向一个页表；而页表是二级表，包含 1024 个页表项（PTE），每个 PTE 指向一个 4KB 的物理页。

地址转换的过程如下：首先，系统需要定位页目录项：CPU 通过 cr3 寄存器

获取页目录项基地址。再通过虚拟地址的高 10 位得到 PDE 的偏移地址，读取 PDE 内容；随后再通过虚拟地址的中间 10 位得到 PTE 的偏移地址，读取 PTE 内容；最后通过物理页基地址的高 20 位和虚拟地址的低 12 位，最终得到物理地址。

了解了二级页表分页机制后，我们先把网站上的代码复现到虚拟机，随后修改 setup.cpp 编写测试代码，如下：

```
void memory_test_thread(void *arg) {  
    // 测试1: 分配单页内存  
    int addr1 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);  
    printf("Allocated 1 page at 0x%x\n", addr1);  
  
    int addr2 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 3);  
    printf("Allocated 3 pages at 0x%x\n", addr2);  
  
    int addr3 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 5);  
    printf("Allocated 5 pages at 0x%x\n", addr3);  
  
    int addr4 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 2);  
    printf("Allocated 2 pages at 0x%x\n", addr4);  
  
    printf("Releasing 3 page at address: 0x%x\n", addr2);  
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr2, 3);  
  
    printf("Releasing 5 pages at address: 0x%x\n", addr3);  
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr3, 5);  
  
    printf("Releasing 1 page at address: 0x%x\n", addr1);  
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr1, 1);  
  
    printf("Releasing 2 pages at address: 0x%x\n", addr4);  
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr4, 2);  
  
    printf("\nMemory released successfully\n");  
}
```

运行该程序，运行结果如下：

```
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
Created memory test thread, PID: 1
Allocated 1 page at 0x200000
Allocated 3 pages at 0x201000
Allocated 5 pages at 0x204000
Allocated 2 pages at 0x209000
Releasing 3 page at address: 0x201000
Releasing 5 pages at address: 0x204000
Releasing 1 page at address: 0x200000
Releasing 2 pages at address: 0x209000

Memory released successfully
```

通过观察程序输出结果，我们可以看到，系统能够成功在划分的内存空间分配和释放相应数量的页，二级分页机制成功得到实现和基本维护，实验任务完成。

----- 实验任务 2 （对应 assignment2） -----

- 任务要求：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。
- 实验步骤：

①实现 first-fit 分配算法：first-fit 分配算法的原理是：从内存空闲链表的头部开始顺序查找，找到第一个能满足请求大小的空闲块，并立即分配。在复现网站源码的过程中，我们不难看出网站的源码实现的就是 first-fit 分配算法，其中算法的核心代码位于 bitmap.cpp 的 allocate 函数中：

```

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;
}

```

我们编写了如下测试程序：

```

void memory_test_thread(void *arg) {
    int addr1 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 5);
    printf("Allocated 5 page at 0x%x\n", addr1);

    int addr2 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 2);
    printf("Allocated 2 pages at 0x%x\n", addr2);

    int addr3 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 4);
    printf("Allocated 4 pages at 0x%x\n", addr3);

    int addr4 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 3);
    printf("Allocated 3 pages at 0x%x\n", addr4);

    printf("Releasing 5 page at address: 0x%x\n", addr1);
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr1, 5);

    printf("Releasing 4 pages at address: 0x%x\n", addr3);
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr3, 4);

    int addr5 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 3);
    printf("Allocated 3 page at 0x%x\n", addr5);

    int addr6 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 7);
    printf("Allocated 7 page at 0x%x\n", addr6);

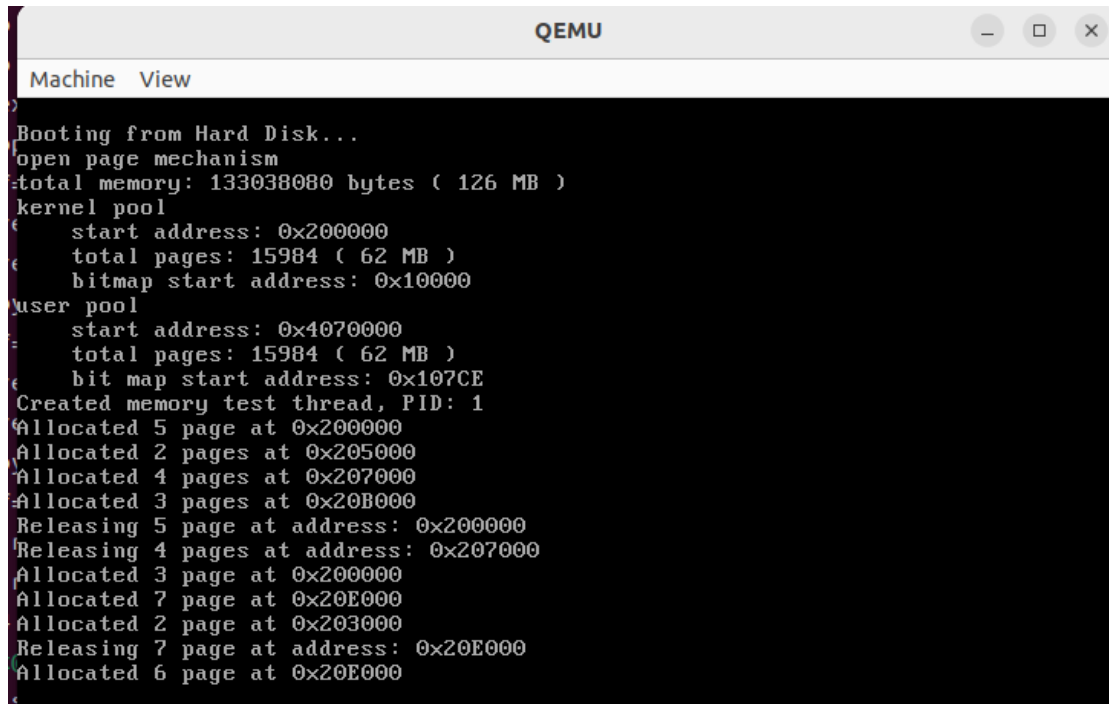
    int addr7 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 2);
    printf("Allocated 2 page at 0x%x\n", addr7);

    printf("Releasing 7 page at address: 0x%x\n", addr6);
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr6, 7);

    int addr8 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 6);
    printf("Allocated 6 page at 0x%x\n", addr8);
}

```

这一程序的运行结果如下：



```
QEMU

Machine View

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
}
Created memory test thread, PID: 1
Allocated 5 page at 0x200000
Allocated 2 pages at 0x205000
Allocated 4 pages at 0x207000
Allocated 3 pages at 0x20B000
Releasing 5 page at address: 0x200000
Releasing 4 pages at address: 0x207000
Allocated 3 page at 0x200000
Allocated 7 page at 0x20E000
Allocated 2 page at 0x203000
Releasing 7 page at address: 0x20E000
Allocated 6 page at 0x20E000
```

我们可以看到，系统每次都能找到内存地址中第一个满足大小的地址空间（注意，需要同时符合“第一个”和“满足大小”，详情请见上面的测试样例），并成功分配内存，**first-fit** 分配算法实现成功。

②实现 **best-fit** 分配算法：**best-fit** 分配算法的原理是：从所有空闲内存块中选择一个大小最接近且不小于请求大小的块进行分配，以最小化剩余碎片。根据这个原理，我们修改 `allocate` 函数，如下：

```
if (count == 0 || count > length)
    return -1;

int bestStart = -1;
int bestSize = length + 1; // 初始化为一个比可能的最大空闲区域还大的值

int index = 0;
while (index < length)
{
    // 跳过已分配的位
    while (index < length && get(index))
        ++index;

    // 记录空闲区域的起始位置
    int start = index;

    // 计算当前空闲区域的大小
    int empty = 0;
    while (index < length && !get(index))
    {
        ++empty;
        ++index;
    }

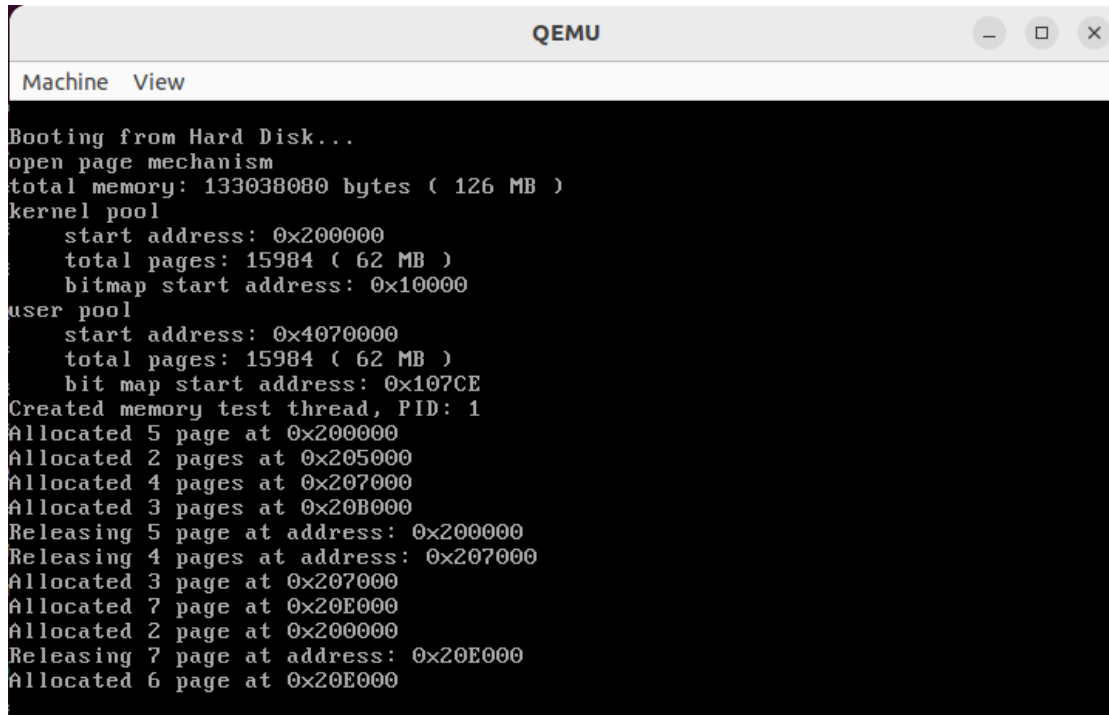
    // 如果这个空闲区域能满足请求，并且比之前找到的最佳块更小
    if (empty >= count && empty < bestSize)
    {
        bestStart = start;
        bestSize = empty;
    }

    // 如果找到一个正好大小的块，可以提前返回
    if (bestSize == count)
        break;
}

// 如果找到合适的空闲区域
if (bestStart != -1)
{
    // 标记为已分配
    for (int i = 0; i < count; ++i)
    {
        set(bestStart + i, true);
    }
    return bestStart;
}
```

（其中找到正好大小的块就提前返回的目的是：节约遍历时间成本，并且保证当内存空间存在多个正好大小的块的时候，优先使用第一个块）。

我们使用和 **first-fit** 完全一样的测试程序，以便于对比这两个算法之间的区别，从而更好的得出结论。程序运行结果如下：



```
QEMU
Machine View
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
Created memory test thread, PID: 1
Allocated 5 page at 0x200000
Allocated 2 pages at 0x205000
Allocated 4 pages at 0x207000
Allocated 3 pages at 0x20B000
Releasing 5 page at address: 0x200000
Releasing 4 pages at address: 0x207000
Allocated 3 page at 0x207000
Allocated 7 page at 0x20E000
Allocated 2 page at 0x200000
Releasing 7 page at address: 0x20E000
Allocated 6 page at 0x20E000
```

我们可以看到，程序主要的区别就在于分配三个页面的时候，**best-fit** 算法选择的不是位于 **0x200000** 处的地址空间，而是大小更为接近的 **0x207000** 处的地址空间。显而易见，**best-fit** 分配算法实现成功，实验任务完成。

----- 实验任务 3 （对应 assignment3） -----

- 任务要求：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，也可以提出自己的算法。
- 实验步骤： 在这里我们实现的是 **FIFO** 页面置换算法。

首先我们分析源代码，我们发现和前面的实验任务，这里主要的区别在于 **memory.cpp** 中。在前面的实验任务中，我们只是初步实现了内存管理，内存分配也只有 **bitmap** 上面简单的分配函数。而在这一任务中，我们才真正实现了虚拟地址到物理地址的映射，并且对于这两个地址，我们有相应的分配页面和释放页面的函数以维护地址空间。

通过观察源码并运行测试程序（即不断分配页面直到没有内存），我们可以

明显发现源码缺少实现页面置换的部分，当页面被使用完后，就无法再分配页面了。

为了实现 **FIFO** 页面置换算法，我们首先分析我们需要什么。**FIFO** 页面置换算法，顾名思义，就是在物理页面被用完或者可用物理页面不足以分配的时候，我们需要从最早被分配到页面开始释放，直到可分配的连续物理页满足请求分配的页数。因此，我们需要一个数据结构来记录页面被分配到先后顺序：遵循先入先出原则的队列显然是最优解；其次，我们需要维护这个队列：即在页面被分配时入队，页面被释放时出队——因此我们需要修改分配页面函数和释放页面函数。

理清了思路后，我们开始着手修改源码：

首先我们在 `memory.h` 中声明了一个队列，这个队列以 `ListItem` 为元素，并且具有头尾指针。

```
//修改：定义数据结构队列
struct Queue{
    ListItem que[999];
    int head;
    int tail;
};
```

随后，鉴于源码中划分了用户物理地址和内核物理地址，因此我们在 `memoryManager` 中分别声明了用户队列和内核队列：

```
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;
    // 内核虚拟地址池
    AddressPool kernelVirtual;

    Queue kernelQueue;
    Queue userQueue;
```

为了实现队列的维护，我们声明了入队函数和出队函数，这两个函数将在分配页面函数和释放页面函数中被调用：

```
//修改：入队函数
void push_page(enum AddressPoolType type, const int vaddr, const int count);

//修改：出队函数
int pop_page(enum AddressPoolType type);
```


与此同时，为了让队列的元素能够承载起始地址和页数这两个信息，以便于页内存释放操作，我们还修改了 `List.h` 文件，为结构体 `ListItem` 增加了两个变量内容：

```
struct ListItem
{
    //修改：增加存储信息
    int address;
    int count;
    ListItem *previous;
    ListItem *next;
};
```

做好了这些前置准备，我们正式修改核心文件 `memory.cpp`：

首先，对于 `memoryManager` 的初始化，我们需要初始化两个队列，即让他们的头尾指针都指向 0：

```
void MemoryManager::initialize()
{
    this->totalMemory = 0;
    this->totalMemory = getTotalMemory();

    //修改：初始化队列
    kernelQueue.head=0;
    kernelQueue.tail=0;
    userQueue.head=0;
    userQueue.tail=0;
}
```

随后，我们实现入队/出队函数：

```
void MemoryManager::push_page(enum AddressPoolType type, const int vaddr, const int count){
    ListItem temp;
    temp.address=vaddr;
    temp.count=count;
    temp.next=nullptr;
    temp.previous=nullptr;

    if(type=AddressPoolType::KERNEL){
        kernelQueue.que[kernelQueue.tail]=temp;
        kernelQueue.tail=(kernelQueue.tail+1)%999;
    }
    else if(type=AddressPoolType::USER){
        userQueue.que[userQueue.tail]=temp;
        userQueue.tail=(userQueue.tail+1)%999;
    }
    printf("push page start from:%x,number of pages:%d\n",temp.address,temp.count);
}
```

```

int MemoryManager::pop_page(enum AddressPoolType type){
    if(type==AddressPoolType::KERNEL){
        if(kernelQueue.head==kernelQueue.tail){
            return -1;
        }
        ListItem temp=kernelQueue.que[kernelQueue.head];
        kernelQueue.head=(kernelQueue.head+1)%999;
        releasePages(type,temp.address,temp.count);
        printf("pop page start from:%x,number of pages:%d\n",temp.address,temp.count);
        return 0;
    }
    else if(type==AddressPoolType::USER){
        if(userQueue.head==userQueue.tail){
            return -1;
        }
        ListItem temp=userQueue.que[userQueue.head];
        userQueue.head=(userQueue.head+1)%999;
        releasePages(type,temp.address,temp.count);
        printf("pop page start from:%x,number of pages:%d\n",temp.address,temp.count);
        return 0;
    }
}

```

这两个函数都需要根据用户态和内核态操作不同的队列，并且我们通过求余这一操作，把队列变成循环队列从而节约操作系统的内存空间。

最后，我们通过修改 `allocatePages` 函数，实现了在内存空间不足的时候进行页面置换的操作，从而实现页面置换算法。

```

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)
    {
        pop_page(type);
        return 0;
    }
    push_page(type,virtualAddress,count);
    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;
}

```

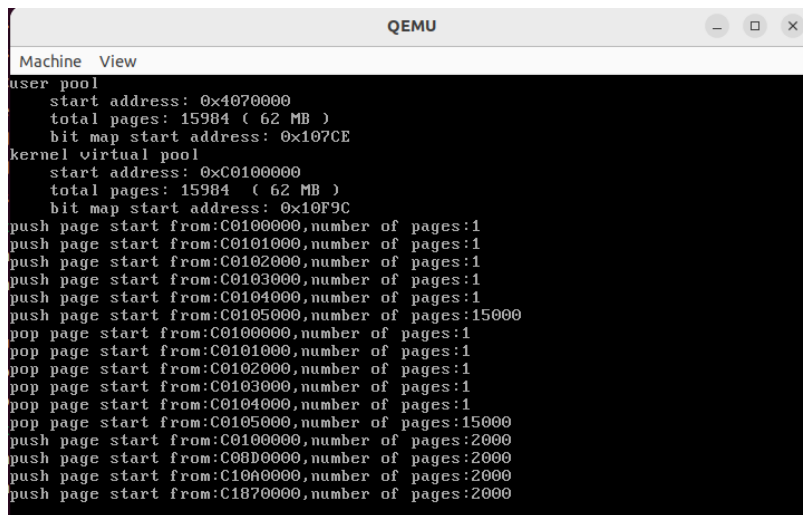
我们修改了测试样例，以测试页面置换算法的正确性：

```

void first_thread(void *arg)
{
    for(int i=0;i<5;i++){
        char *addr = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
    }
    char *addr = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 15000);
    for(int i=0;i<20;i++){
        char *addr = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 2000);
    }
    asm_halt();
}

```

程序运行结果如下：

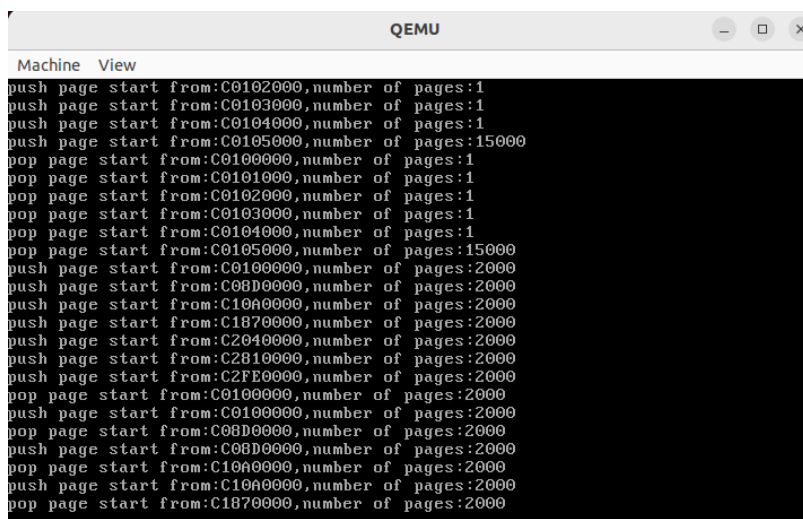


QEMU Machine View

```

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
push page start from:C0100000,number of pages:1
push page start from:C0101000,number of pages:1
push page start from:C0102000,number of pages:1
push page start from:C0103000,number of pages:1
push page start from:C0104000,number of pages:1
push page start from:C0105000,number of pages:15000
pop page start from:C0100000,number of pages:1
pop page start from:C0101000,number of pages:1
pop page start from:C0102000,number of pages:1
pop page start from:C0103000,number of pages:1
pop page start from:C0104000,number of pages:1
pop page start from:C0105000,number of pages:15000
push page start from:C0100000,number of pages:2000
push page start from:C08D0000,number of pages:2000
push page start from:C10A0000,number of pages:2000
push page start from:C1870000,number of pages:2000

```



QEMU Machine View

```

push page start from:C0102000,number of pages:1
push page start from:C0103000,number of pages:1
push page start from:C0104000,number of pages:1
push page start from:C0105000,number of pages:15000
pop page start from:C0100000,number of pages:1
pop page start from:C0101000,number of pages:1
pop page start from:C0102000,number of pages:1
pop page start from:C0103000,number of pages:1
pop page start from:C0104000,number of pages:1
pop page start from:C0105000,number of pages:15000
push page start from:C0100000,number of pages:2000
push page start from:C08D0000,number of pages:2000
push page start from:C10A0000,number of pages:2000
push page start from:C1870000,number of pages:2000
push page start from:C2840000,number of pages:2000
push page start from:C2810000,number of pages:2000
push page start from:C2FE0000,number of pages:2000
pop page start from:C0100000,number of pages:2000
push page start from:C0100000,number of pages:2000
pop page start from:C08D0000,number of pages:2000
push page start from:C08D0000,number of pages:2000
pop page start from:C10A0000,number of pages:2000
push page start from:C10A0000,number of pages:2000
pop page start from:C1870000,number of pages:2000

```

通过观察我们可以发现，系统两次实现了页面置换，并且符合 **FIFO** 页面置换算法的原理，实验任务完成。

----- 实验任务 4 （对应 assignment4） -----

- 任务要求： （1）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；（2）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。

- 实验步骤：

①分析代码：虚拟页分配函数如下：

```
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)
    {
        pop_page(type);
        return 0;
    }
    push_page(type, virtualAddress, count);
    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;
        }
    }
}
```

虚拟页分配的过程一共有三步：

第一步：分配内存页，即从虚拟地址池中分配连续的虚拟页，供后续映射物理页使用。详细操作是系统调用 `allocateVirtualPages(type, count)`，根据类型（内核/用户）从对应的虚拟地址池分配 `count` 个虚拟页。若分配成功，返回起始虚拟地址，否则返回 `0`（这也是 `assignment3` 源码在修改前运行的时候最后输出的地址为 `0` 的原理）。

```
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;
}
```

其中 `allocateVirtualPages` 函数调用的是 `bitmap` 中的 `allocate` 函数，即我们在 `assignment2` 中实现页面分配算法的函数。

第二步：分配物理页，即为每个虚拟地址分配物理页。详细操作是系统调用 `allocatePhysicalPages(type, count)`，根据类型（内核/用户）从对应的物理地址池分配 `count` 个物理页。若分配成功，返回起始物理地址，否则返回 0。

分配物理页函数（`allocatePhysicalPages`）的内容和前面的 `allocateVirtualPages` 几乎一模一样，唯一的区别是操作的地址空间不一样，一个是 `Virtual`，另一个是 `Physical`。

```
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;
}
```

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。详细操作是先根据虚拟地址的高 10 位和中间 10 位计算页目录项和页表项；随后检查页表是否存在：若页目录项无效（`*pde & 0x1 == 0`），分配一个物理页作为新页表，初始化并写入页目录项；最后设置页表项，将物理页地址写入页表项（`*pte = physicalPageAddress | 0x7`），标志位 `P=1`（存在）、`RW=1`（可读写）、`US=1`（用户可访问）。

```

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;
}

```

最后, 若某一步失败 (比如物理页不足), 系统则会回滚错误, 将刚刚已经分配的物理页和虚拟页释放, 防止资源泄露。

而虚拟页释放的过程如下:

```

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);

        // 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);
}

```

第一步, 对每一个虚拟页, 释放为其分配的物理页:

```

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);
    }
}

```

这里调用的是 **bitmap** 的 **release** 函数, 即通过起始地址和页数, 将这些地址设置为 **false** (未分配);

```

void BitMap::release(const int index, const int count)
{
    for (int i = 0; i < count; ++i)
    {
        set(index + i, false);
    }
}

```

于此同时，我们还需要设置页表项为不存在，防止被错误使用；

第二步：释放虚拟页：

```

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

```

②分析代码 bug：通过观察源代码，并借鉴往届学长的实验报告，我发现在分配内存函数的实现是先将起始地址赋值，随后再检查是否存在足够多的连续页面，因此如果在检查开始前，调用页分配函数的进程被调度走，后续进程再调用页分配算法的时候，由于此时这些页面还未被分配，因此可能会导致重复分配的现象。我们通过为 bitmap.cpp 的 allocate 函数增加如下延时：

```

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<10000;i++){
                for(int j=0;j<20000;j++){
                    // 延时操作
                }
            }
            for (int i = 0; i < count; ++i)
            {
                set(start + i, true);
            }

            return start;
        }
    }

    return -1;
}

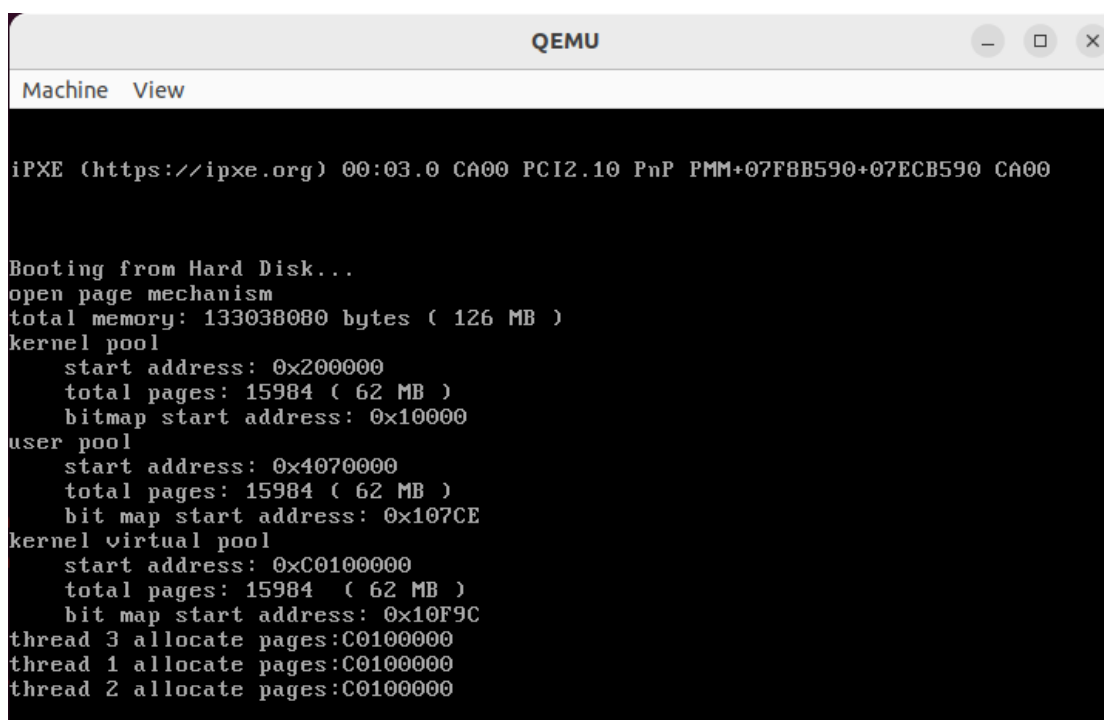
```


随后我们编写了这样一个测试样例，即有三个进程先后执行页分配函数：

```
void thread1(void *arg){
    char *p1=(char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
    printf("thread 1 allocate pages:%x\n",p1);
}

void thread2(void *arg){
    char *p2=(char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
    printf("thread 2 allocate pages:%x\n",p2);
}
void thread3(void *arg){
    char *p3=(char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
    printf("thread 3 allocate pages:%x\n",p3);
}
void first_thread(void *arg)
{
    programManager.executeThread(thread1, nullptr, "thread1", 1);
    programManager.executeThread(thread2, nullptr, "thread2", 1);
    programManager.executeThread(thread3, nullptr, "thread3", 1);
    asm_halt();
}
```

程序运行，我们发现三个进程得到的是同样的页，这明显不合理：



```
QEMU
Machine View

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
kernel virtual pool
    start address: 0xC0100000
    total pages: 15984 ( 62 MB )
    bit map start address: 0x10F9C
thread 3 allocate pages:C0100000
thread 1 allocate pages:C0100000
thread 2 allocate pages:C0100000
```

因此，我们尝试修改源码的这一错误。

在 lab6 中，我们学习的锁和信号量可以解决同步互斥问题，在这里刚好可以派上用场。因此，我们引入了信号量 Semaphore 充当互斥锁，并在 memoryManager 的页分配函数前后，分别加锁和解锁：

```
Semaphore allocate_lock;
```



```

int MemoryManager::allocatePages(enum AddressPoolType type, const int count)
{
    allocate_lock.P();

    // 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;
        }
    }
}

allocate_lock.V();

```

对于同样的测试样例，运行结果如下：

```

Machine View

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
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x10F9C
thread 1 allocate pages:C0100000
thread 2 allocate pages:C0101000
thread 3 allocate pages:C0102000
specify the 'raw' format explicitly to remove the restrictions

```

前面的 bug 得到了解决，实验任务完成。

Section 5 实验总结与心得体会

这次实验完成得较为“草率”，一来是因为临近期末，最近其他科的大作业各方面压力比较大，不得不减少在这门课上的钻研时间；另一方面是因为这个实

验的拓展难度确实不小，以下是我的一些思考或者可尝试探索的改进方向。

①在 `assignment2` 中，我们实现了 `first-fit` 和 `best-fit` 两种页面分配算法，这两个算法的核心都是从头遍历。而在源码中，我们不难看出系统是通过 `index` 作为遍历的指针，而 `length` 是遍历的最长次数；基于这一框架，我们或许可以通过建立一个全局变量以替代 `index`，而在每次分配页面后无需把全局变量清零，这样或许就可以实现 `next-fit` 页面分配算法；

②在 `assignment3` 中，我们只实现了 `FIFO` 这种最简单的页面置换算法（虽然他不一定是最劣算法，毕竟 `FIFO` 的最大优势是它的公平性），这不完全是偷懒，主要是因为我们在理论课上学习过的，常见的页面置换算法中，只有 `FIFO` 是基于时间的算法，而 `LRU`、`Clock`、`NRU`、工作集算法都是基于访问的算法。而我们纵观 `memoryManager` 的源码，我很难去想到一个很好的方法实现‘访问’和‘记录’这个操作。

本人在和助教交谈的过程中，只能想到以下这个比较“笨拙”的方法：写一个专门代表访问的函数，这个函数的传参是一个地址，传参的地址处于哪一页，就代表哪一页被访问了，而函数的内容就是给那一页的计数器变量+1（或者修改引用位）；随后我们再修改页的数据结构，引入访问次数的统计计数器。

但是这样的解决方案产生了两个问题：（1）我们仍然无法实现更贴近现实的“访问并修改”某一页的内容这一操作；（2）这样的操作会面临大量的修改，包括页的数据结构，包括如何统计一共访问了多少次，包括页分配函数的传参（当前的 `int addr` 肯定无法实现对它自己所有页的访问次数的遍历求和）。

随后，我还把目光看向了页目录项的数据结构；它的 32 位中似乎有一个访问位（A 位）和一个脏位（D 位）。我们当然可以学到如何触发这个访问位的改变，也知道脏位应该如何改变，而且这两个位都是操作系统自动修改，无需我们手动赋值。但是问题是我们在尝试页面置换的时候，应该用什么样的操作去读取当前页的这些位？是直接读取就可以了吗？还是说需要汇编代码（原子操作）的介入？我在有限的时间内没有想到解决这些问题的完整方案，因此最终就放弃了，这些思想暂时留在这里，等待更有实力的人，或者更有实力的自己去实现吧。

Section 6 附录：代码清单

代码请见附件。