LAB7 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：物理内存与虚拟内存管理

专业名称：计算机科学与技术

学生姓名：马福泉

学生学号：　23336179

实验地点：实验中心 B202

实验时间：　2025.6.3

Section 1 实验概述

- 在本次实验中，我们首先学习如何使用位图和地址池来管理资源。然后，我们将实现在物理地址空间下的内存管理。接着，我们将会学习并开启二级分页机制。在开启分页机制后，我们将实现在虚拟地址空间下的内存管理。
- 本次实验最精彩的地方在于分页机制。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，我们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。但是，本实验最令人困惑的地方也在于分页机制。开启了分页机制后，程序中使用的地址是虚拟地址。我们需要结合页目录表和页表才能确定虚拟地址对应的物理地址。而我们常常会忘记这一点，导致了我们不知道某些虚拟地址表示的具体含义。

Section 2 预备知识与实验环境

- 预备知识：汇编语言基础，计算机体系结构，操作系统基础，调试工具的使用（如 gd
- 实验环境：
  - 虚拟机版本/处理器型号：VMware-Ubuntu22.04.5/i386（32 位）
  - 代码编辑环境： 编辑器：VSCode
    插件：C/C++插件，汇编插件
  - 代码编译工具： 编译器：gcc
    工具链：binutils、make、qemu、nasm 等
    调试工具：gdb
  - 重要三方库信息：GNU binutils：工具集（如 as、ld）
    gdb：调试工具
    QEMU、Bochs：模拟器用于测试 等

Section 3 实验任务

- 实验任务 1：物理页内存管理的实现

复现实验 7 指导书中“物理页内存管理”一节的代码，实现物理页内存的管理，具体要求如下：
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。
3. （不强制要求，对实验完成度评分无影响）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物理页内存管理。完成之后，你需要指明相比指导书，你实现的物理页内存管理的特点。

- 实验任务 2：二级分页机制的实现

复现实验 7 指导书中“二级分页机制”一节的代码，实现二级分页机制，具体要求如下：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）。注意：建议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），作为报告独立完成的个人信息表征。
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

复现实验 7 指导书中“虚拟页内存管理”一节的代码，实现虚拟页内存的管理，具体要求如下：
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。
4. （不强制要求，对实验完成度评分无影响）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，你需要指明相比于指导书，你实现的虚拟页内存管理的特点。

- 实验任务 4：（选做内容，如果完成，可附加实验完成度评分）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，比如下面所列算法的其中一种：先进先出页面置换(FIFO).最优页面置换(OPR).最近最少使用页面置换(LRU)最不经常使用页面置换(LFU)。上述置换算法的细节参见理论课教材(《操作系统概

念》，原书第 9 版，中文)第 272-280 页，你也可以实现一种自己设计的置换算法。要求:描述你的设计思路并展示页面置换结果的截图(也可以统计缺页错误发生的次数作为输出）。

Section 4 实验步骤与实验结果

---------- 实验任务 1 ----------

- 任务要求：复现实验 7 指导书中“物理页内存管理”一节的代码，实现物理页内存的管理，具体要求如下：

  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：先复制代码进行复现实验，分析代码结构分析位图，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，编写样例分析物理页内存管理的实现是否存在 bug

- 实验步骤：

  1.复制代码，复现物理页内存管理

```
make build
make run
```

2. 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:

(1) 分析初始化过程

在 MemoryManager::initialize 函数中：

读取总内存：通过 getTotalMemory 函数读取之前保存的内存大小。

预留内存：预留一部分内存用于内核和其他必要用途。

计算剩余内存：计算剩余的空闲内存，并将其划分为内核空间和用户空间。

初始化地址池：为内核空间和用户空间分别初始化地址池，设置位图的起始地址和大小。

打印信息：打印内存管理的基本信息，包括内核空间和用户空间的起始地址、总页数和位图的起始地址。

(2) 物理页分配和释放的实现思路

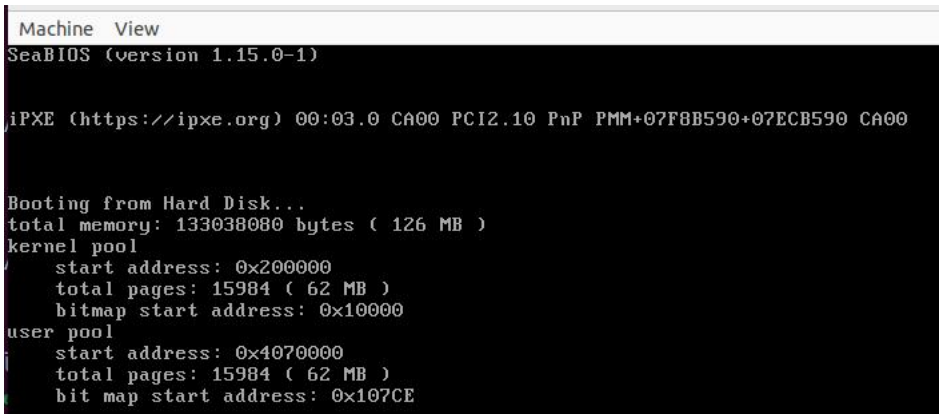
分配：调用 AddressPool::allocate 函数，从指定的地址池中分配连续的页。如果分配成功，返回分配的起始地址；否则返回 0。

释放：调用 AddressPool::release 函数，将指定的页返回到地址池中。更新位图，将这些页标记为“空闲”。

3. 编写样例分析物理页内存管理的实现是否存在 bug

```
// 测试用例1：验证单个页的分配和释放是否正确
    printf("=== Test Case 1: Single Page Allocation and Release ===\n");
    int addr1 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
    printf("Allocate 1 page, address: 0x%x\n", addr1);
    bool test1_part1 = (addr1 != 0);
    printTestResult("Single page allocation", test1_part1);

    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr1, 1);
    printf("Release 1 page, address: 0x%x\n", addr1);

    int addr1_2 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
    printf("Reallocate 1 page, address: 0x%x\n", addr1_2);
    bool test1_part2 = (addr1 == addr1_2);
    printTestResult("Single page reallocation after release", test1_part2);
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr1_2, 1);

    printf("\n");

    // 测试用例2：验证多个连续页的分配和释放是否正确
    printf("=== Test Case 2: Multiple Contiguous Pages Allocation and Release ===\n");
    int addr2 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 5);
    printf("Allocate 5 pages, start address: 0x%x\n", addr2);
    bool test2_part1 = (addr2 != 0);
    printTestResult("Multi-page contiguous allocation", test2_part1);

    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr2, 5);
    printf("Release 5 pages, start address: 0x%x\n", addr2);

    int addr2_2 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 5);
    printf("Reallocate 5 pages, start address: 0x%x\n", addr2_2);
    bool test2_part2 = (addr2 == addr2_2);
    printTestResult("Multi-page reallocation after release", test2_part2);
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr2_2, 5);

    printf("\n");

    // 测试用例3：验证分配超过可用页时是否正确返回失败
    printf("=== Test Case 3: Allocation Beyond Available Pages ===\n");
    // 先分配一些页面，避免一次性请求太多
    int addr3 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 10);
    printf("Allocate 10 pages, start address: 0x%x\n", addr3);

    // 尝试分配一个超大的页数（应该会失败）
    int huge_pages = 1000000; // 一个非常大的值
    int addr3_huge = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, huge_pages);
    printf("Try to allocate %d pages, return: 0x%x\n", huge_pages, addr3_huge);
    bool test3 = (addr3_huge == 0);
    printTestResult("Excessive allocation failure detection", test3);

    // 释放之前分配的内存
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr3, 10);
    printf("Release 10 pages, start address: 0x%x\n", addr3);

    printf("\n");

    // 测试用例4：验证释放未分配的页是否会导致系统崩溃
    printf("=== Test Case 4: Releasing Unallocated Pages ===\n");
    // 选择一个合理但未分配的地址（这里使用之前测试中得到的地址加一个偏移）
    int unallocated_addr = addr1 + PAGE_SIZE * 100;
    printf("Try to release unallocated address: 0x%x\n", unallocated_addr);

    // 直接调用释放函数，然后检查系统是否仍在运行
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, unallocated_addr, 1);
    printf("Release unallocated page completed\n");

    // 如果代码执行到这里，说明系统没有崩溃
    bool test4 = true;
    printTestResult("Release unallocated page", test4);

    printf("\n=== Physical Memory Page Allocation Test Completed ===\n");
```

- 实验结果展示：



上图已成功复现总内存：系统识别出总内存为 133038080 字节（大约 126 MB）。

（1） 内核空间（kernel pool）：

起始地址（start address）：0x200000

总页数（total pages）：15984 页（大约 62 MB）

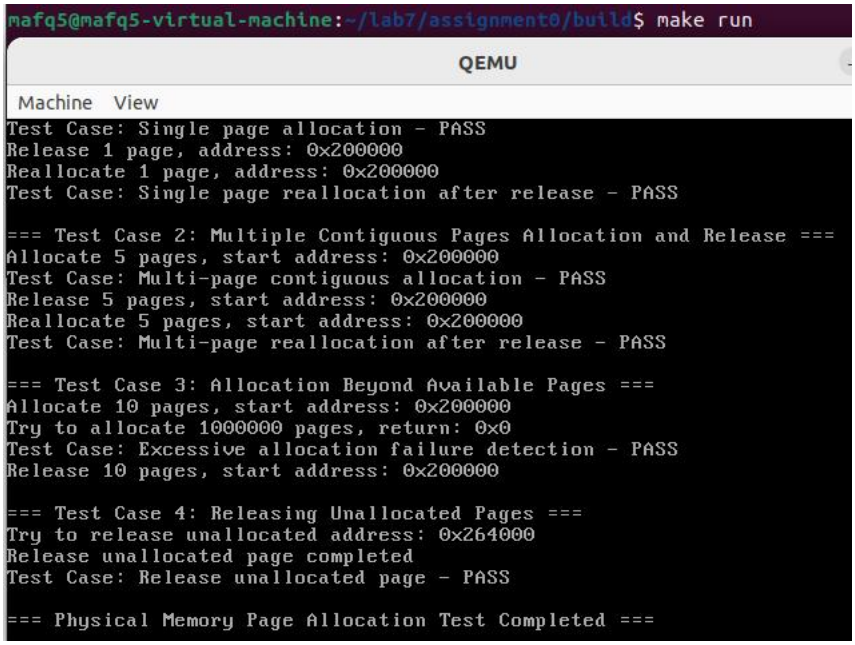
位图起始地址（bitmap start address）：0x100000

（2） 用户空间（user pool）：

起始地址（start address）：0x40700000

总页数（total pages）：15984 页（大约 62 MB）

位图起始地址（bitmap start address）：0x107CE



如上图，实现正确

测试用例 1：验证单个页的分配和释放是否正确。

测试用例 2：验证多个连续页的分配和释放是否正确。

测试用例 3：验证分配超过可用页时是否正确返回失败。

测试用例 4：验证释放未分配的页是否会引发错误。

---------- 实验任务 2 ----------

- 任务要求：二级分页机制的实现

复现实验 7 指导书中“二级分页机制”一节的代码，实现二级分页机制，具体要求如下：

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）。注意：建议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），作为报告独立完成的个人信息表征。
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：

分配不同数量的物理页，并在每次分配后添加一个固定的偏移量（6179）。
打印出分配的物理页地址，包括添加了偏移量后的地址。
释放之前分配的物理页，并在释放时减去偏移量，以确保释放正确的原始地址。
打印出释放的物理页地址，包括原始地址和添加了偏移量后的地址。

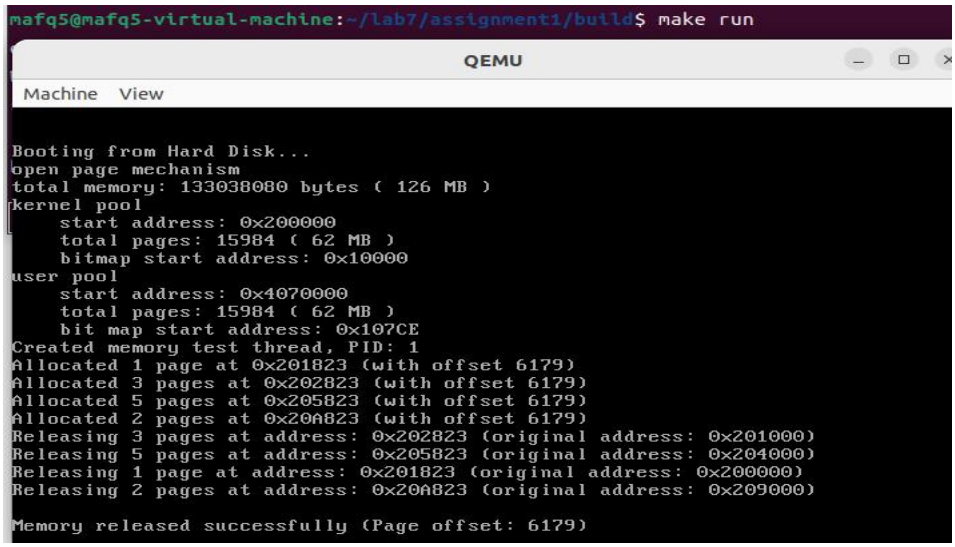
- 实验步骤：

编写 setup.cpp 代码

```
void memory_test_thread(void *arg) {
    // Test 1: Allocate single page memory
    int addr1 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
    addr1 += 6179;  // Add page offset
    printf("Allocated 1 page at 0x%x (with offset 6179)\n", addr1);

    int addr2 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 3);
    addr2 += 6179;  // Add page offset
    printf("Allocated 3 pages at 0x%x (with offset 6179)\n", addr2);

    int addr3 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 5);
    addr3 += 6179;  // Add page offset
    printf("Allocated 5 pages at 0x%x (with offset 6179)\n", addr3);

    int addr4 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 2);
    addr4 += 6179;  // Add page offset
    printf("Allocated 2 pages at 0x%x (with offset 6179)\n", addr4);


    printf("Releasing 3 pages at address: 0x%x (original address: 0x%x)\n", addr2, addr2 - 6179);
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr2 - 6179, 3);

    printf("Releasing 5 pages at address: 0x%x (original address: 0x%x)\n", addr3, addr3 - 6179);
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr3 - 6179, 5);

    printf("Releasing 1 page at address: 0x%x (original address: 0x%x)\n", addr1, addr1 - 6179);
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr1 - 6179, 1);

    printf("Releasing 2 pages at address: 0x%x (original address: 0x%x)\n", addr4, addr4 - 6179);
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, addr4 - 6179, 2);

    printf("\nMemory released successfully (Page offset: 6179)\n");
}
```

```
make build
make run
```

- 实验结果展示：



如图：分配不同数量的物理页，并在每次分配后添加一个固定的偏移量（6179）。打印出分配的物理页地址，包括添加了偏移量后的地址。释放之前分配的物理页，并在释放时减去偏移量，以确保释放正确的原始地址。打印出释放的物理页地址，包括原始地址和添加了偏移量后的地址。

二级页表的优势

- 内存使用效率：

一级页表需要为整个虚拟地址空间分配一个完整的页表，这在虚拟地址空间很大时会浪费大量内存。二级页表（页目录表和页表）允许延迟分配页表项，只有在实际需要映射虚拟页到物理页时才分配页表项，从而节省内存。

- 灵活性和扩展性：

二级页表允许更灵活的内存映射。例如，可以为不同的进程创建不同的页目录表，实现进程间的地址空间隔离。这种结构也更容易扩展到更大的地址空间，如从 32 位到 64 位系统。

- 安全性：

每个进程可以有自己的页目录表，这增强了进程间的隔离，提高了系统的安全性。页表项中的权限位（如用户/超级用户、读写/只读）可以控制对内存页的访问，防止恶意程序或错误操作破坏系统。

- 支持多任务和多用户：

二级页表结构支持多任务操作系统，可以为每个任务或用户进程创建独立的页表，实现内存隔离和保护。这对于多用户系统尤其重要，因为每个用户可能运行不同的程序，需要隔离他们的地址空间。

- 动态内存分配：

二级页表支持动态内存分配，操作系统可以根据需要动态地创建和销毁页表项，适应不同的内存需求。

---------- 实验任务 3 ----------

- 任务要求：虚拟页内存管理的实现

复现实验 7 指导书中“虚拟页内存管理”一节的代码，实现虚拟页内存的管理，具体要求如下：

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。

- 思路分析：

1. 结合代码，学习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
3. 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。
4. 将索引页目录表本身的页目录项从 1023 改为 1000

计算第 141 个页目录项的虚拟地址
计算第 891 个页目录项指向的页表中第 109 个页表项的虚拟地址
在代码中添加这部分逻辑并打印结果

- 实验步骤：

1. 虚拟页内存分配的三个基本步骤

从虚拟地址池中分配虚拟页：分配连续的虚拟页，这些虚拟页在虚拟地址空间中是连续的。

为每个虚拟页分配物理页：从物理地址池中为每个虚拟页分配相应的物理页，这些物理页可以是不连续的。

建立虚拟页和物理页之间的映射关系：在页目录表和页表中建立映射关系，使得虚拟地址可以通过 MMU（内存管理单元）转换为物理地址。

2. 虚拟页内存的释放过程

释放物理页：对于每个虚拟页，找到其对应的物理页并释放。

清除页表项：将虚拟页对应的页表项设置为无效，防止再次被访问。

释放虚拟页：释放虚拟地址空间中的虚拟页。

3. 构造测试用例来分析虚拟页内存管理的实现是否存在 bug

```cpp
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();
}
```

```
mafq5@mafq5-virtual-machine:~/lab7/task3(bug)/build$ make run
```

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 1 allocate pages:C0100000
thread 2 allocate pages:C0100000
thread 3 allocate pages:C0100000
```

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。

4. Debug: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：

```cpp
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();

    return virtualAddress;
}
```

```
mafq5@mafq5-virtual-machine:~/lab7/task3(correct)/build$ make run
```

QEMU

Machine View

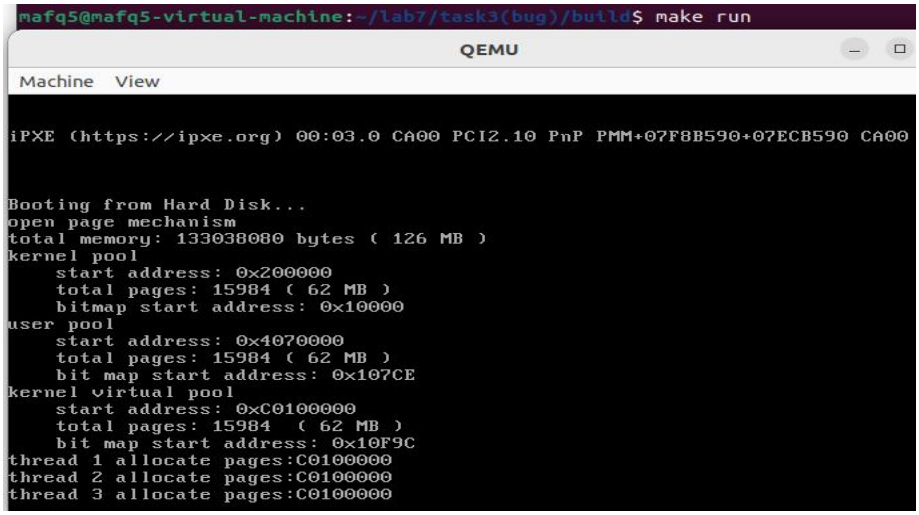
```
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00


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 1 allocate pages:C0100000
thread 2 allocate pages:C0101000
thread 3 allocate pages:C0102000
```

bug 得到解决。

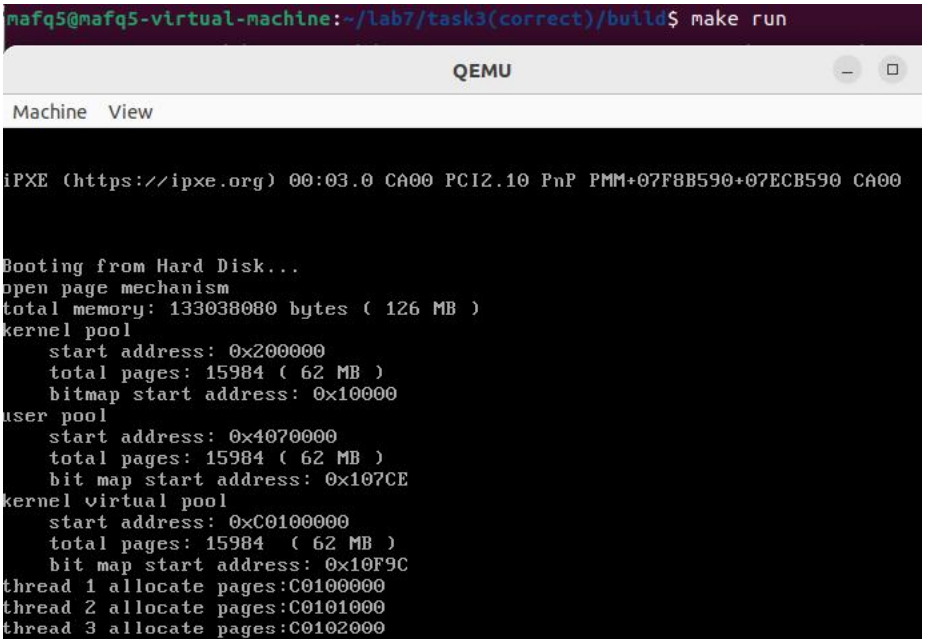
5. 将索引页目录表本身的页目录项从 1023 改为 1000
计算第 141 个页目录项的虚拟地址
计算第 891 个页目录项指向的页表中第 109 个页表项的虚拟地址
在代码中添加这部分逻辑并打印结果

- 实验结果展示：

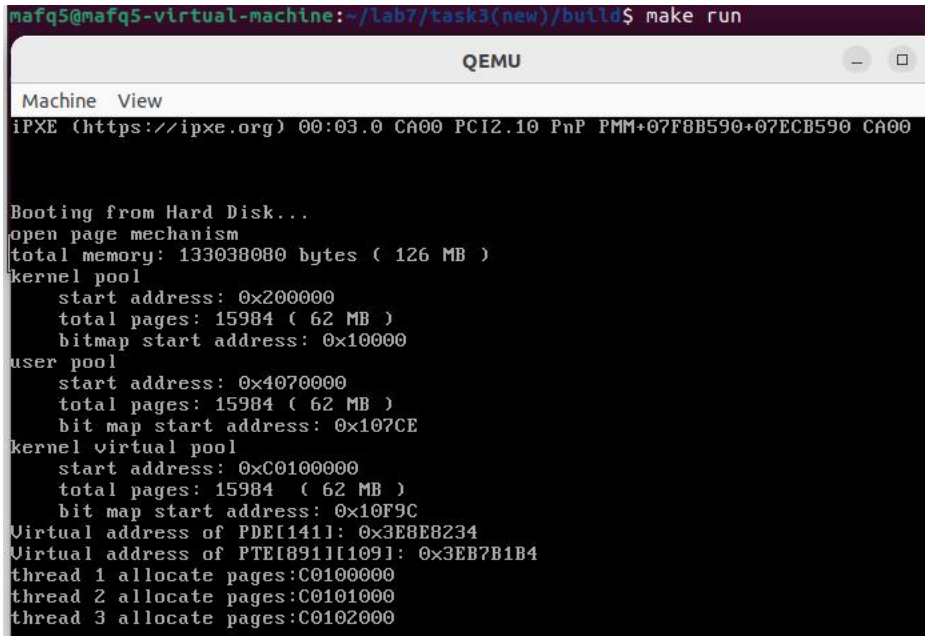
bug:task3（bug）



Debug 后:task3（debug）



task3（new）：



---------- 实验任务 4 ----------

- 任务要求：

（选做内容，如果完成，可附加实验完成度评分）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，比如下面所列算法的其中一种:先进先出页面置换(FIFO).最优页面置换(OPR).最近最少使用页面置换(LRU)最不经常使用页面置换(LFU)。上述置换算法的细节参见理论课教材(《操作系统概念》，原书第 9 版，中文)第 272-280 页，你也可以实现一种自己设计的置换算法。要求:描述你的设计思路并展示页面置换结果的截图(也可以统计缺页错误发生的次数作为输出）。

- 实验步骤：实现 FIFO

① 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。我们在 memoryManager 中分别声明了用户队列和内核队列。

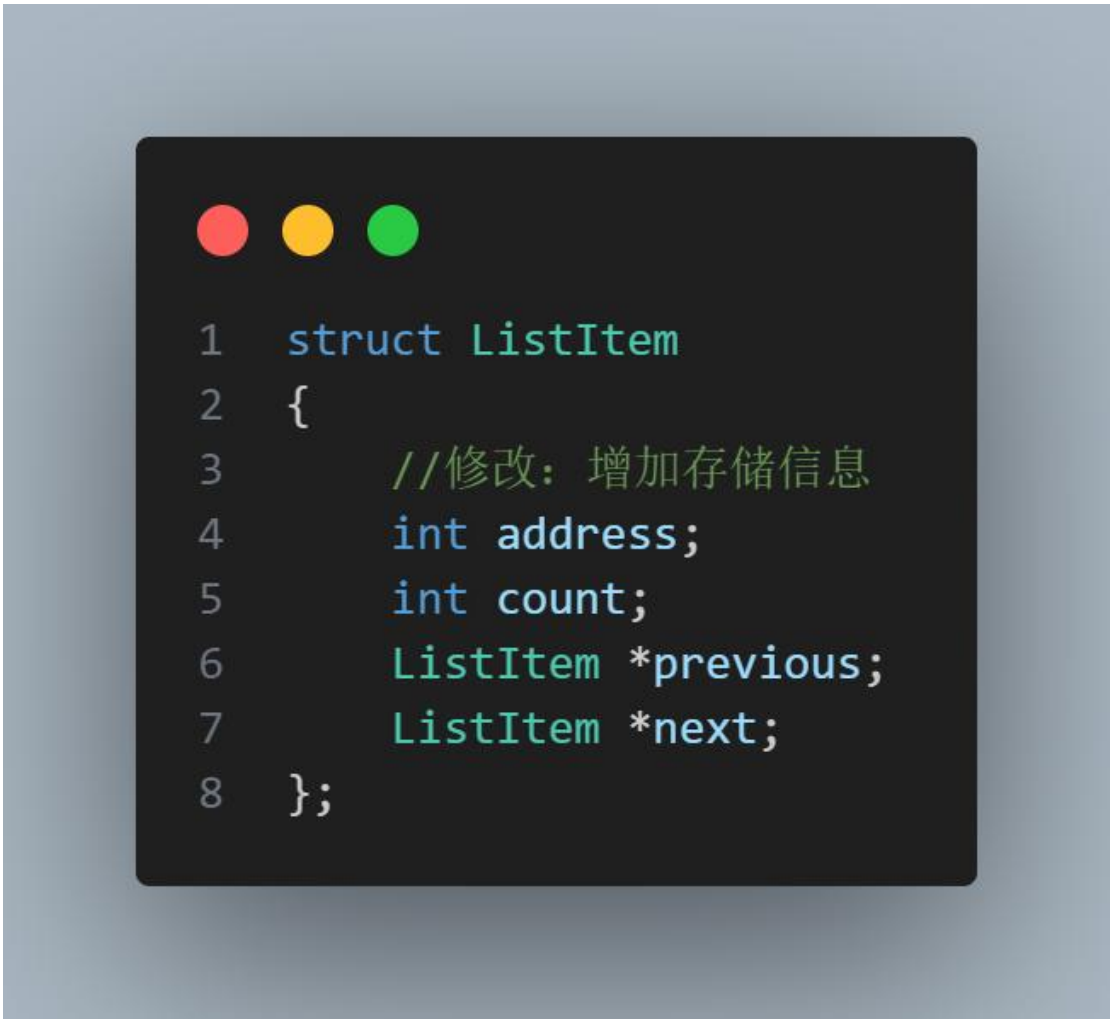
```
//修改：定义数据结构队列
struct Queue{
    ListItem que[999];
    int head;
    int tail;
};
```

```
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;

```

② 声明入队函数和出队函数实现队列的维护，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);
```

③ 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，修改 `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;
};
```

④ 修改核心文件 `memory.cpp`：

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：

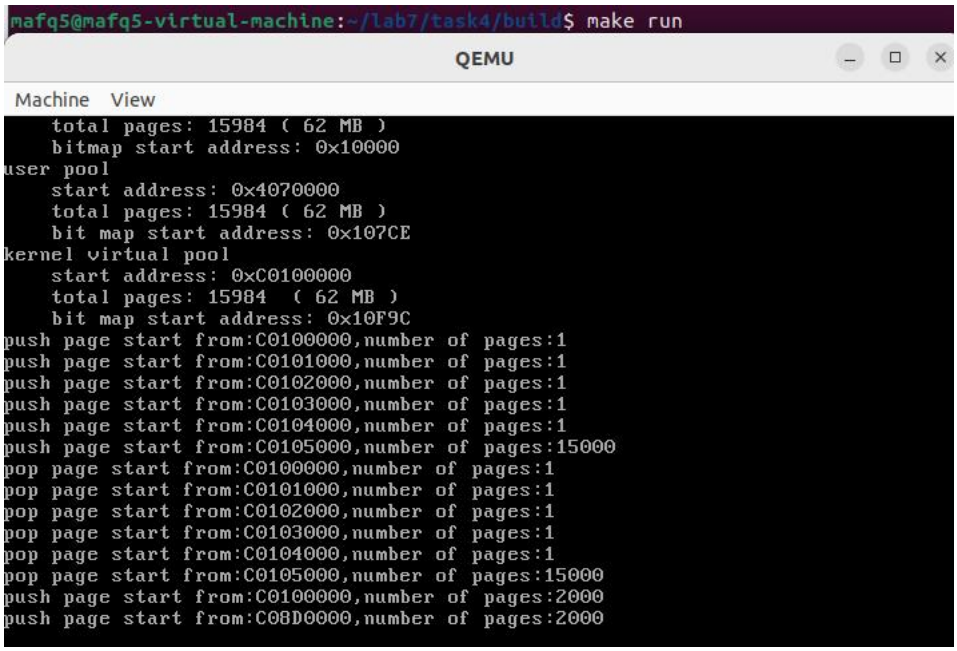
```
void MemoryManager::push_page(enum AddressPoolType type, const int vaddr, const int count){
    ListItem temp;
    temp.address=vaddr;
    temp.count=count;
    temp.next=nullptr;
    temp.previous=nullptr;

    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;
    }
}
```

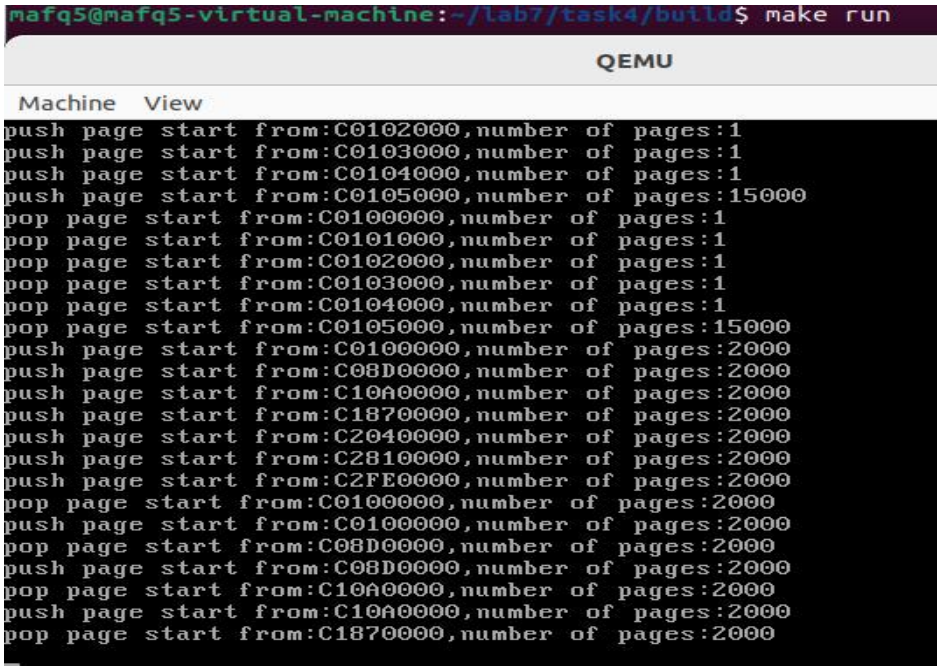
这两个函数都需要根据用户态和内核态操作不同的队列，通过求余这一操作，把队列变成循环队列从而节约操作系统的内存空间。
最后修改 `allocatePages` 函数，实现在内存空间不足的时候 进行页面置换的操作，从而实现页面置换算法。

```
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;
}
```

- 实验结果展示：

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。





1. Section 5 实验总结与心得体会

1. 页表和页目录的准确初始化

在实验中，正确初始化页表和页目录是至关重要的。这包括为页表分配内存、设置正确的页表项（PTE）和页目录项（PDE），以及确保它们正确地映射到物理内存。初始化过程中的任何错误都可能导致系统无法正确地进行虚拟地址到物理地址的转换，进而引发崩溃或其他不可预测的行为。

- 为页表分配连续的物理内存：页表需要连续的物理内存，因为它们会被 CPU 高速访问。不连续的页表可能会导致 TLB（Translation Lookaside Buffer）效

率降低，进而影响性能。

- 正确设置页表项和页目录项：每个 PTE 和 PDE 都必须正确设置，包括存在位、读写权限、用户/超级用户权限等。这些设置决定了虚拟地址到物理地址的映射是否有效，以及进程是否可以访问特定的内存区域。

2. 内存泄漏的避免

- 在内存管理中，内存泄漏是一个常见的问题。内存泄漏发生在分配的内存没有被正确释放时，这会导致随着时间的推移，可用内存越来越少，最终可能导致系统性能下降或崩溃。
- 确保每次分配都有对应的释放：每当通过 allocatePages 函数分配内存后，都应该在适当的时机调用 releasePages 函数来释放内存。这不仅包括显式的内存释放调用，还包括在异常或错误处理路径中释放内存。
- 检查内存分配函数的返回值：在分配内存时，应该检查函数的返回值以确保内存分配成功。如果内存分配失败（例如，由于内存不足），应该采取适当的错误处理措施，如释放已分配的资源、返回错误码等。

Section 6 附录：参考资料清单

1. 部分代码 debug 与指令除了参考实验指导书，还参考了大语言模型

2. 参考文章，博客部分如下：

【操作系统】实验：内存管理_内存管理实验-CSDN 博客

实验五_内存管理实验-CSDN 博客