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: 第六章 内存管理

专业名称: 信息与计算科学

学生姓名: 郑鸿鑫

学生学号: 22336313

实验地点: 实验中心 D503

实验时间: 2024/5/21

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：x86 汇编语言程序设计、IA-32 处理器体系结构
- 实验环境：
  - 虚拟机版本/处理器型号：
    11th Gen Intel® Core™ i5-11320H @ 3.20GHz × 2
  - 代码编辑环境：VS Code
  - 代码编译工具：g++
  - 重要三方库信息：Linux 内核版本号：linux-5.10.210
    Ubuntu 版本号：Ubuntu 18.04.6LTS，Busybox 版本号：Busybox_1_33_0

Section 3 实验任务

- 实验任务 1: 物理页内存管理的实现
- 实验任务 2：二级分页机制的实现
- 实验任务 3：虚拟页内存管理的实现

- 实验任务 4：页面置换算法的实现

Section 4 实验步骤与实验结果

---------- 实验任务 1 ----------

- 任务要求：

Assignment 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。

- 实验步骤：

1. 我们使用类 Memory Manager 来执行内存管理。

```
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 startAddress,
const int count);
    // 获取内存总容量
    int getTotalMemory();
};
```

2. 对 Memory Manager 的初始化如下所示：

```
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);
}
```

MemoryManager 的初始化主要分为两部分，预留空间的内存分配和内核，用户物理地址空间的内存分配。0x000000 到 0x100000 存放系统内核，然后接着存放了 256 个内核页表。这些预留空间不参与内核，用户地址池的内存分配。

其中绿色加粗标注出来的是对位图和地址池的初始化，第一句设置了内核物理页面位图的起始地址，第二句计算用户空间物理页面位图的起始地址。它从内核位图的起始地址开始，并加上了内核页面数量（kernelPages）向上取整到最近的 8 的倍数（ceil(kernelPages, 8)）所表示的字节数。这样做是为了确保用户位图的起始地址在内核位图之后，并且对齐到下一个 8 位字节的边界，因为位图的每个位对应 8 个连续的字节。后两行初始化内核物理池还有用户物理池，第一个参数为内存起始地址，第二个参数为物理页面总数，第三个参数为物理页面的起始地址。

3. 对物理页的分配与释放的代码：

```
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;
}
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);
    }
}
```

物理页分配与释放的思路如下：allocatePhysicalPages 函数根据 type 的值，选择正确的地址池进行分配，allocate 方法尝试在相应的地址池中找到 count 个连续的未分配页，并返回第一个未分配页的起始地址。如果找不到足够数量的连续未分配页，allocate 方法返回-1。如果 start 为-1（表示分配失败），则返回 0；否则返回起始地址 start。

ReleasePhysicalPages 函数根据 type 的值，选择正确的地址池进行释放，release 方法将在地址池中标记指定的物理页为未分配状态，这样它们可以被未来的分配请求重新使用。

4. 我们在第一个线程函数中增加一些语句，用于测试内存和页分配是否存在 bug：

```
void first_thread(void *arg)
{
    // 第1个线程不可以返回
    // 执行边界条件测试
    testBoundaryConditions(memoryManager,AddressPoolType::KERNEL);
    int page = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
    if (page != 0) {
        printf("Allocated page at address: %x\n", page);
    } else {
        printf("Allocation failed\n");
    }
    // 释放页面
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, page, 1);
    printf("Released page\n");
    asm_halt();
}
```

其中测试了分配和释放一张页面是否正常，调用的函数 testBoundaryCondition 用于检测边界性条件，代码如下：

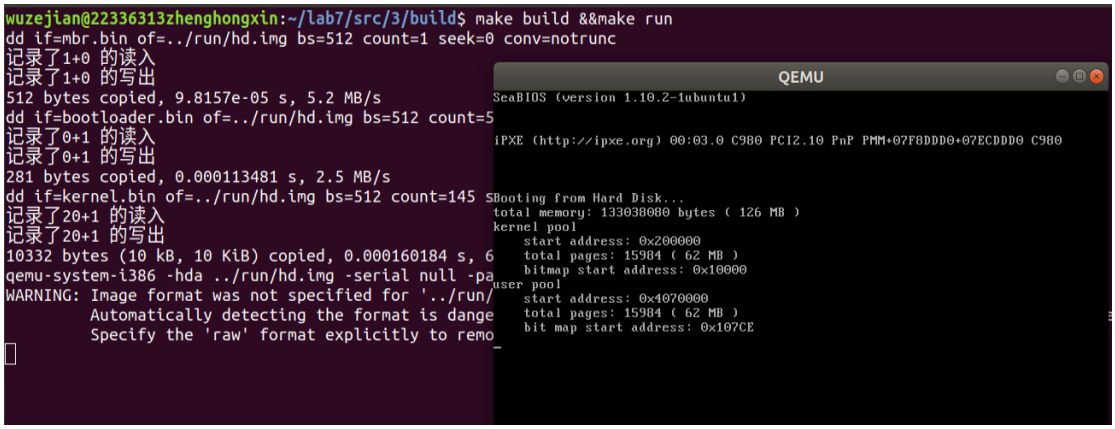
```
void testBoundaryConditions(MemoryManager&memoryManager,AddressPoolType
type) {
    // 分配0个页面
```

```
    int zeroPages = memoryManager.allocatePhysicalPages(type, 0);
    if (zeroPages != 0) {
        printf("Allocated 0 pages, but got %x\n", zeroPages);
        return;
    }
    // 分配超过总量的页面
    int tooManyPages = memoryManager.allocatePhysicalPages(type,
memoryManager.getTotalMemory() / PAGE_SIZE + 1);
    if (tooManyPages != 0) {
        printf("Allocated too many pages, but got %x\n", tooManyPages);
        return;
    }
    printf("Boundary conditions test passed\n");
}
```

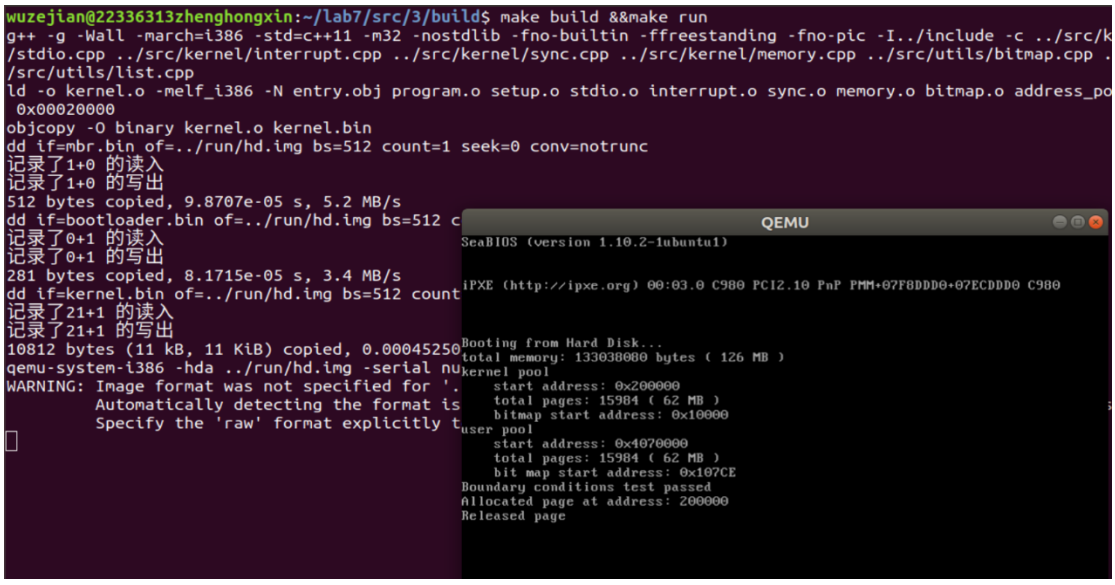
其中分别分配 0 个页面和超过总量的页面，检测系统是否会出现异常，如果没有则打印通过边界性检测的信息。

- 实验结果展示：

  复现代码的运行结果如下：



测试内存页的分配释放和边界性结果打印如下：



可以看到对页的分配释放正常，也能顺利通过边界性检测。

---------- 实验任务 2 ----------

- 任务要求：

Assignment 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。

- 思路分析：参照参考书的指引复现代码，实现分页机制并加以分析。
- 实验步骤：

1. 先利用以下代码复现分页机制：

```
 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");

}
```

2. 编写代码测试物理页的分配与释放：

```
   memoryManager.allocatePhysicalPages(KERNEL,15984);
    memoryManager.releasePhysicalPages(KERNEL,0x2233631,3);
    memoryManager.allocatePhysicalPages(KERNEL,1);
```

```
    memoryManager.allocatePhysicalPages(KERNEL,5);
```

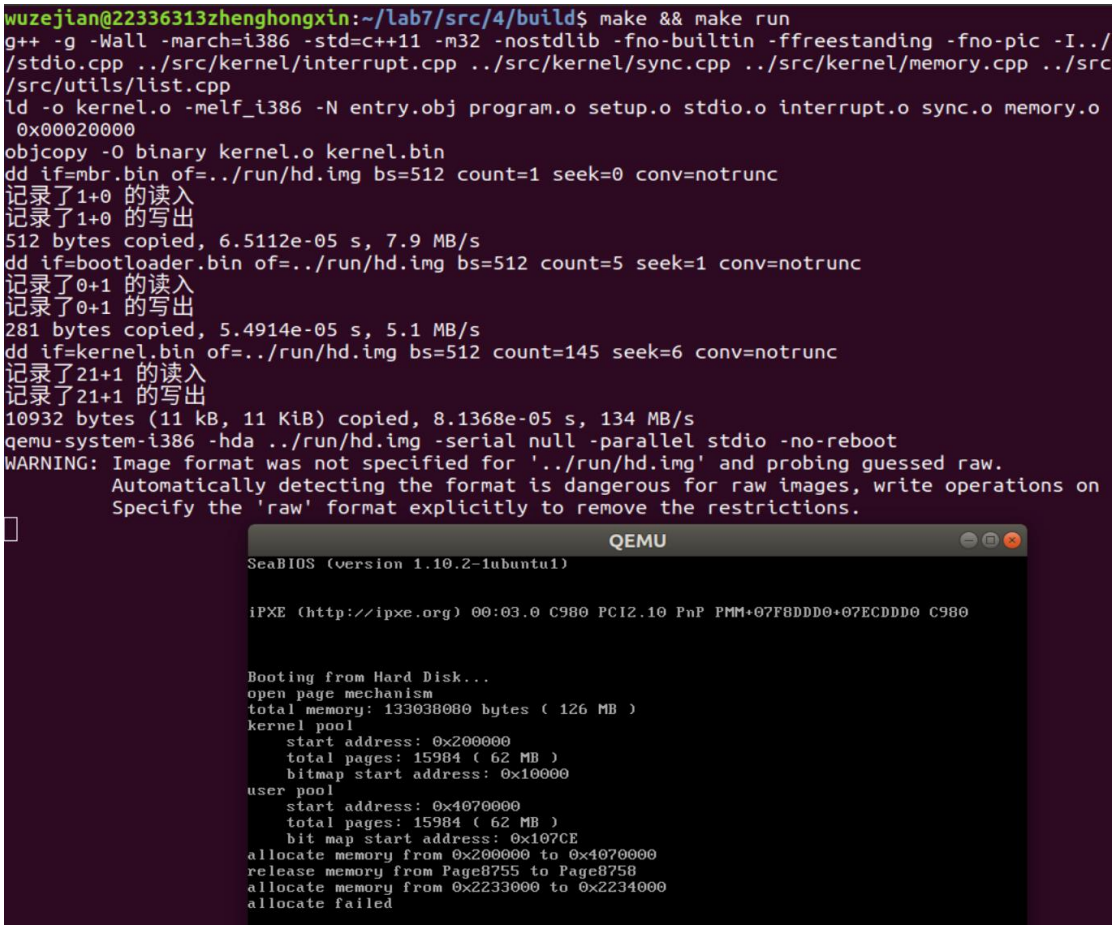
对测试代码的解释详见实验结果展示与分析

3. 添加一些额外的打印信息的语句：

```
// 从地址池中分配 count 个连续页
int AddressPool::allocate(const int count)
{
    int start = resources.allocate(count);
    if(start == -1) printf("allocate failed");
    else  printf("allocate memory from 0x%x to 0x%x\n",start * PAGE_SIZE +
startAddress,(start+count) * PAGE_SIZE + startAddress);
    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)
{
    printf("release memory from Page%d to Page%d\n",address/PAGE_SIZE,
address/PAGE_SIZE+amount);
    resources.release((address - startAddress) / PAGE_SIZE, amount);
}
```

4. 分析二级页表对比一级页表的优势：

- 减少内存占用：在一级页表机制中，如果虚拟地址空间很大，即使只使用了少量的物理内存，页表也需要为整个虚拟地址空间分配空间。这可能导致大量未使用的内存被分配给页表。而二级页表通过引入页目录表，只有当前活跃的页表（即被访问的页表）才会被加载到内存中，从而减少了内存的占用。
- 动态内存管理：二级页表允许操作系统动态地管理内存，只有在需要时才分配页表。这意味着操作系统可以根据当前的内存使用情况灵活地分配和回收页表项，提高内存的使用效率。
- 灵活性和扩展性：二级页表提供了更好的灵活性和扩展性。操作系统可以根据需要调整页目录和页表的大小，以适应不同的内存管理策略和优化内存访问性能。
- 实验结果展示：



图中显示的总内存大小和内核地址池及用户地址池均表示物理内存。

测试代码首先将内核空间中的所有空闲页面全部分配完毕，然后在 0x2233631 地址开始释放 3 页（22336313 为本人学号），上面打印的页号 8755 到 8758 分别为 2233 到 2236 由十六进制转化为 10 进制。释放后我们再申请 1 页内存，可以看到是分配了 0x2233000 到 0x2234000 的这一页（由于页的大小为 4KB，也就是 0x1000），再申请 5 页则会由于内存中没有这个数目的连续的空闲页而打印分配失败。

故我们实现了二级分页机制并分析了其相比一级分页的优势所在。

---------- 实验任务 3 ----------

- 任务要求：

Assignment 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。

- 实验步骤：

1. 要求一中描述虚拟页内存管理的三个基本步骤，实现虚拟页内存管理的代码如下：

```
 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)
    {
        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;
}
```

虚拟页内存释放的三个步骤可以总结为：a 分配虚拟页，b 为每个虚拟页指定物理页，c 设置页目录项（PDE）和页表项（PTE）来建立虚拟页和物理页的映射。

a. `int virtualAddress = allocateVirtualPages(type, count);`

这一步从虚拟地址池中分配连续的虚拟页。allocateVirtualPages()函数根据请求的类型（KERNEL 或 USER）从相应的虚拟地址池（kernelVirtual）中分配 count 数量的虚拟页，并返回起始虚拟地址。如果无法分配足够的页，则返回 0。

b. `physicalPageAddress = allocatePhysicalPages(type, 1);`

对于每个虚拟页，需要从物理地址池中分配一个对应的物理页。allocatePhysicalPages 函数负责这项工作，它返回一个物理页的起始地址。如果物理页分配失败，函数返回 0。

c.

```
 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
```

最后一步是建立虚拟页到物理页的映射。这通过设置页目录项（PDE）和页表项（PTE）来完成，确保虚拟地址能够转换为正确的物理地址。connectPhysicalVirtualPage 函数负责创建这个映射。

2. 要求一中虚拟页内存的释放过程，实现代码如下：

```
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);
}

int MemoryManager::vaddr2paddr(int vaddr)
{
    int *pte = (int *)toPTE(vaddr);
    int page = (*pte) & 0xfffff000;
    int offset = vaddr & 0xfff;
    return (page + offset);
}
```

首先释放对应的物理页，释放物理页后需要清除对应的页表项，通过将页表项设置为 0 来实现（`pte = (int *)toPTE(vaddr);*pte = 0;`）。最后一步是释放虚拟页本身。

3. 要求 2，我们使用以下代码来测试虚拟页的内存管理：

```
    char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
    char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 6313);
    char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
    printf("allocate: p1 start at %x \nallocate: p2 start at %x\nallocate: p3 start
at %x\n", p1, p2, p3);
    memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
    printf("after released: p3 start at %x\n", p3);
     char *p4 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
    printf("reallocate: p2 start at %x\n", p2);
    printf("reallocate: p4 start at %x\n", p4);
    asm_halt();
```

我们前后分别将 p4 分配的页面数设置为 100 和 10，以测试是否可以正常分配页面，测试的结果和具体分析见实验结果展示部分。

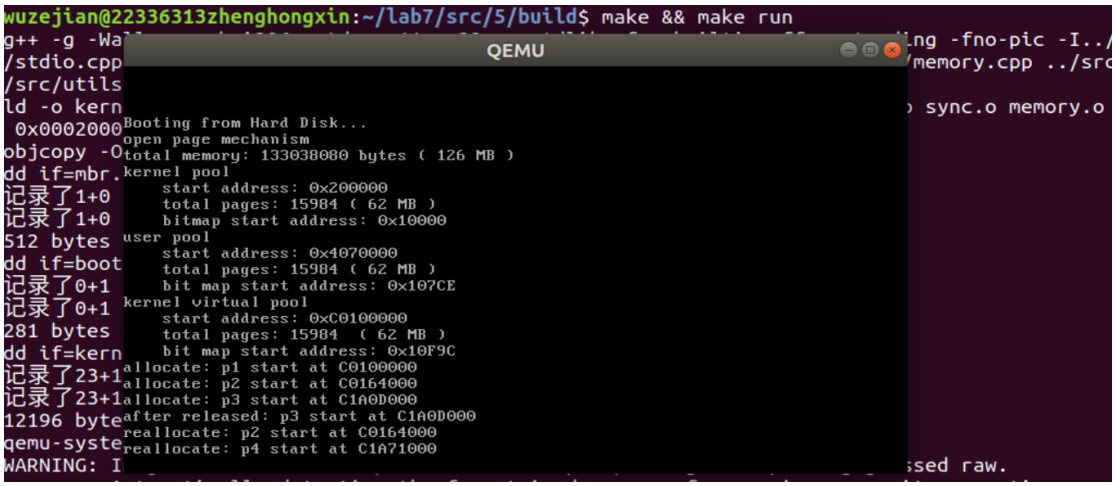
4. 要求 3，页目录项和页表项的构造

首先我们要清楚二级分页机制中的虚拟地址与物理地址的映射关系，高 32 位到 22 位是页目录号，21 位到 12 位是页表号，最后低 12 位是页内偏移量。

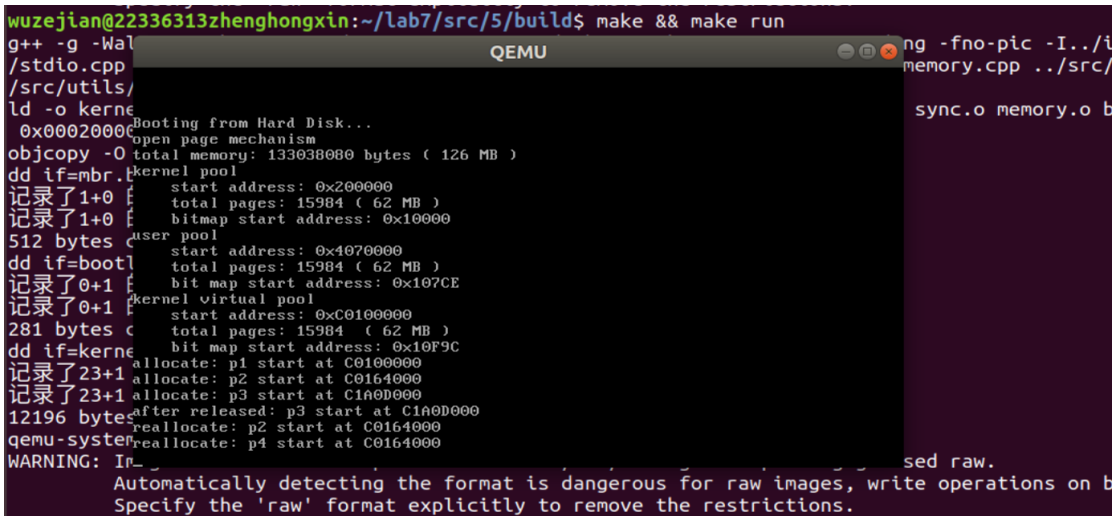
- 构造第 141 个页目录项的虚拟地址，由于页目录表的基址存放在第 1000 个页表项中，所以 PDE 的[31:22]以及[21:12]均为 0x3e8，也就是十进制的 1000。第 141 个页目录项所在的物理页是页目录表，第 141 个页目录项的页内偏移量是 4*141 = 564，转换为 16 进制为 0x234，作为 PDE 的低十二位。所以得到第 141 个页目录项的虚拟地址为 0xfa3e8234。
- 构造第 891 个页目录项指向的页表的第 109 个页表项的虚拟地址，由于第 1000 个页目录项指向它本身，所以 PDE 的[31:22]位是 0x3e8，即十进制的 1000。然后中间的[21:12]位应该为 0x37b（十进制为 891），最后第 109 个页表项的页 内偏移量应该为 4*109 = 436（十六进制为 0x1b4）。所以得到第 891 个页目录项指向的页表的第 109 个页表项的虚拟地址为 0xfa37b1b4。
- 实验结果展示：

要求 2 中对虚拟页内存的测试结果如下：

P4 分配 100 页的结果如下：



P4 分配 10 页的结果如下：

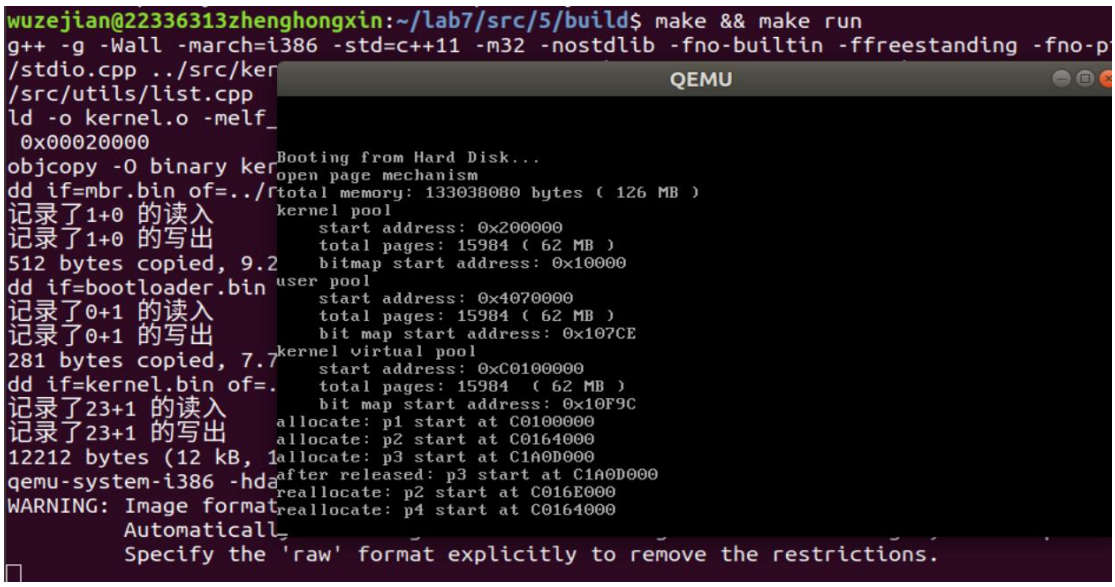


首先 p1 分配 100 页后，可以看到 p2 的起始地址为 C0164000，因为这与 p1 的起始地址 C0100000 相差刚好为 0x64000，由于一页的大小为 4KB，则相差的地址由 16 进制转换后刚好为 10 进制的 100 页，所以分配没问题，p2 分配了 6313 页（学号后四位），p2 与 p3 起始地址相差 0x1849000，转化后为 10 进制的 6313 页，故也符合预期。但是在后续我们对 p2 释放了 10 页之后，在 p2 和 p3 之间存在一个大小为 10 页的孔，当 p4 申请分配 100 页时，p4 跟在 p3 的后面，没有问题；当 p4 申请分配 10 页时，按理应该刚好被分配到 p2 与 p3 之间的空闲页面，但是打印的结果显示，p4 的起始地址和 p2 一样，这是由于页面的释放算法只对页进行释放，而不会更新 p2 指针的值。

故我们在释放后手动更新指针的值：

```
    memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
    p2  = (char*)((int)p2+0xa000 );
```

重新编译运行检查结果：



可以看到 p2 的起始地址已经正确更新（为原地址加上 10 页后即 0xa000 得到的地址）。

---------- 实验任务 4 ----------

- 任务要求：

Assignment 4 页面置换算法的实现

（**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：

1. 创建两个队列（作为 MemoryManager 的成员变量出现）用于管理页面的置换：

```
    List user_queue;
    List kernel_queue;
```

2. 为页面的置换编写 push 和 pop 函数来支持页面的调度：

```
void MemoryManager::push(enum AddressPoolType type,int addr,int count){
    printf("push: %x ,count: %d\n",addr,count);
    static ListItem tmp  [20];
    tmp[kernel_queue.size()].addr = addr;
    tmp[kernel_queue.size()].num = count;
    if (type == AddressPoolType::KERNEL)
{kernel_queue.push_back(&tmp[kernel_queue.size()]);}
    else user_queue.push_back(&tmp[user_queue.size()]);
}

int MemoryManager::pop(enum AddressPoolType type){
    if (type == AddressPoolType::KERNEL){
        if (kernel_queue.empty()) return -1;
        else{
            ListItem *tmp = kernel_queue.front();
            kernel_queue.pop_front();
            releasePages(type,tmp->addr,tmp-> num);
            printf("pop: %x,count : %d\n",tmp -> addr,tmp -> num);
            return 0;
        }
    }
    else return 1;
}
```

值得说明的是，这里每一次页分配时用的 ListItem 是存放在一个静态数组内，如果不是静态，则每次链表（队列）的节点会作为局部变量被销毁，无法实现算法。按照 C 语言本来应该是利用 malloc，free 语句动态分配和释放，但是操作系统内核不支持这两个语句，所以退而求其次使用一个预定义大小的静态或全局数组来存储 ListItem 对象。

3. 在每次成功分配页面后调用 push 函数

```
push(type,virtualAddress,count);
```

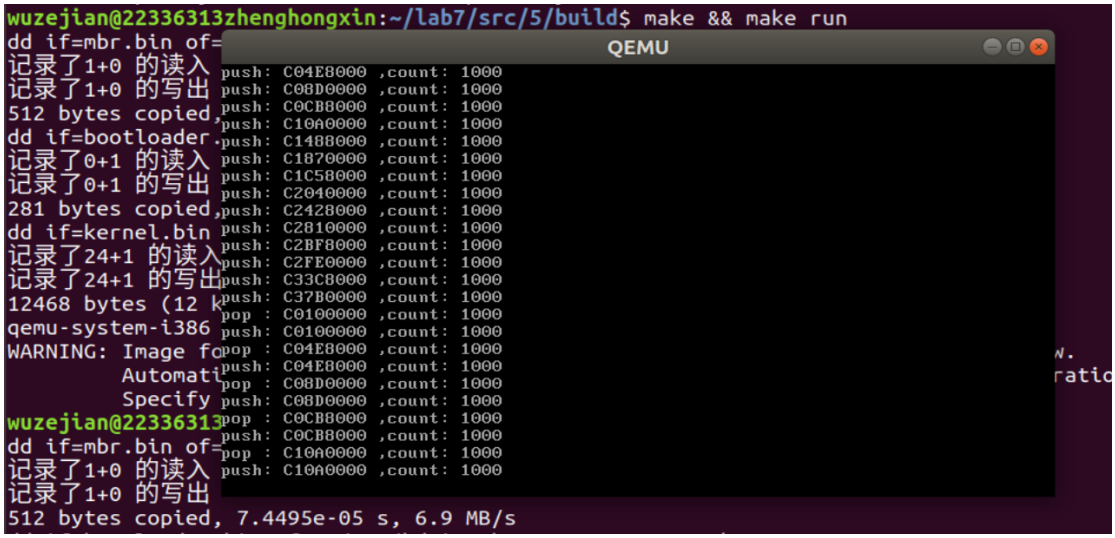
4. 每次空闲页面不足时调用 pop 函数进行页面置换，并且由于队列的性质，所以符合 FIFO 的特性。

```
// 第一步：从虚拟地址池中分配若干虚拟页
    int virtualAddress = allocateVirtualPages(type, count);
    while (!virtualAddress)
    {
        pop(type);
        virtualAddress = allocateVirtualPages(type,count);
    }
```

5. 编写测试代码以测试是否正常进行页的置换调度：

```
for(int i = 0;i < 20;i++){
      char* addr = (char*)memoryManager.allocatePages(AddressPoolType::KERNEL,1000);
}
```

- 实验结果展示：



分析：首先可以看到虚拟地址的起始地址为 0xC0100000， 从这里开始分配 20 次内存，每次分配 1000 页，可以看到分配一次后地址变为了 0xC04E8000，这是由于一页占 4KB，1000 页所占的地址空间转化为 16 进制就是 0x3E8000。由于总共只有 15984 页，而我们总计需要分配 20000 页，所以在后续肯定会出现空闲页面不足的情况，在 0xC37B0000 分配完 1000 页后，剩余地址空间已不足以分配接下来的 1000 页。所以置换算法根据 FIFO 先进先出的原则，将最早分配的 1000 页给释放掉，然后再进行分配，后续输出同理。

Section 5 实验总结与心得体会

通过这次内存管理实验，我对操作系统中的内存管理机制有了更深入的理解。实验过程中，我学习了位图和地址池这两种资源管理技术，并亲自实现了物理地址空间下的内存管理。这些知识让我对计算机如何分配和追踪内存资源有了直观的认识。

在学习二级分页机制时，我体会到了虚拟内存的重要性。通过开启分页机制，我了解到了虚拟地址与物理地址之间的映射关系，以及如何通过页目录表和页表进行地址转换。这个过程虽然复杂，但却是现代操作系统中不可或缺的一部分，它不仅提高了内存的使用效率，还增强了系统的安全性。

在实现虚拟地址空间下的内存管理时，我面临了诸多挑战。特别是在构造页目录项和页表项的虚拟地址时，我需要仔细理解虚拟地址的构造规则，并确保每个步骤都准确无误。这个过程锻炼了我的逻辑思维能力和问题解决能力。

Section 6 附录：参考资料清单

1. 页面大小、页表项、虚拟地址和物理地址之间的关系（转） - jiamian22 - 博客园 (cnblogs.com)
2. 【操作系统】虚拟地址和页表项的关系_页表项和虚拟地址的关系-CSDN 博客
3. SYSU-2023-Spring-Operating-System: 中山大学 2023 学年春季操作系统课程 - Gitee.com