# 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: Lab7 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: 孙凯

## 学生学号: 23336212

## 实验地点: 实验中心B202

## 实验时间: 2025.5.23

## 一、实验要求：

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。

具体内容：

1. 物理页内存管理的实现
2. 二级分页机制的实现
3. 虚拟页内存管理的实现
4. 页面置换算法

## 二、预备知识和实验环境：

预备知识：qemu+gdb调试方法，内存管理，分页机制

**实验环境:**

```
1. 虚拟机版本/处理器型号：Ubuntu 20.04 LTS
2. 代码编辑环境：Vscode+nasm+C/C++插件+qemu仿真平台
3. 代码编译工具：gcc/g++ （64位）
4.  重要三方库信息：无
5. 代码程序调试工具：gdb
```

## 三、实验任务：

- 任务1：物理页内存管理的实现
- 任务2：二级分页机制的实现

- 任务3：虚拟页内存管理的实现
- 任务4：页面置换算法的实现（选做内容）

# 四、实验步骤和实验结果：

## 实验任务1：物理页内存管理的实现

代码放在`lab7/assignment1`中

- 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。
- 实验思路：

```
测试1：基本分配和释放
目的：验证单个页面的分配和释放的基本功能
过程：
分配1个内核页面并验证分配结果
释放该页面
再次分配1个页面，检查是否得到相同的地址（验证页面回收机制）
释放页面

测试2：连续多页分配
目的：测试连续物理内存分配功能
过程：
分配2个连续内核页面
额外分配1个内核页面（测试碎片处理能力）
释放之前分配的2个连续页面
再次分配2个连续页面，检查地址是否相同（验证多页连续内存的回收与重用）

测试3：用户内存分配与释放
目的：验证用户空间内存管理功能
过程：
分配4个用户空间页面
释放这些页面
验证整个过程是否成功

测试4：边界测试
目的：测试内存管理器对超大内存请求的处理能力
过程：
尝试分配16000页内核内存,实际上超出内核内存的大小
检查内存管理器如何应对这种极端请求
```

- 实验步骤：

1. 复现代码，分析实现思路：

**初始化过程分析:**

内存管理系统的初始化是从上到下、层层调用的过程，遵循以下层次结构：

### 1. MemoryManager初始化

初始化步骤:

1. 系统内存划分：系统总内存→预留内存→可用内存→内核池+用户池
2. 位图区域分配：为内核池和用户池分别分配位图存储区域
3. 初始化地址池：配置内核和用户地址池

```
//MemoryManager::initialize()函数完成整个内存管理系统的初始化:
void MemoryManager::initialize()
{
  // 获取系统总内存
  this->totalMemory = getTotalMemory();
  // 预留低端内存(256页+1MB)给系统使用
  int usedMemory = 256 * PAGE_SIZE + 0x100000;
  // 计算可用内存并分配
  int freeMemory = this->totalMemory - usedMemory;
  int freePages = freeMemory / PAGE_SIZE;
  // 内核和用户空间平分剩余内存
  int kernelPages = freePages / 2;
  int userPages = freePages - kernelPages;
  // 计算各个地址区域的起始点
  int kernelPhysicalStartAddress = usedMemory;
  int userPhysicalStartAddress = usedMemory + kernelPages * PAGE_SIZE;
  // 计算位图存储区域的位置
  int kernelPhysicalBitMapStart = BITMAP_START_ADDRESS;
  int userPhysicalBitMapStart = kernelPhysicalBitMapStart + ceil(kernelPages, 8);
  // 初始化两个地址池
  kernelPhysical.initialize((char *)kernelPhysicalBitMapStart,
             kernelPages, kernelPhysicalStartAddress);
  userPhysical.initialize((char *)userPhysicalBitMapStart,
             userPages, userPhysicalStartAddress);
}
```

### 2. AddressPool初始化

```
//AddressPool::initialize()接收三个参数:

- 位图存储区域的起始地址
- 管理的页面数量
- 物理地址起始点

void AddressPool::initialize(char **bitmap*, const int *length*, const int *startAddress*)
{
  resources.initialize(bitmap, length); // 初始化内部的位图
  this->startAddress = startAddress;   // 设置起始物理地址
}
```

### 3.BitMap初始化

```
//BitMap::initialize()是最底层的初始化:

void BitMap::initialize(char **bitmap*, const int *length*)
{
  this->bitmap = bitmap;  // 位图存储区域
  this->length = length;  // 总位数
  int bytes = ceil(length, 8); // 计算需要的字节数
  memset(bitmap, 0, bytes);   // 初始化为全0(表示未分配)
}
```

**物理页分配过程**

物理页分配是一个自顶向下的调用过程:

1. **从MemoryManager开始**

```
int MemoryManager::allocatePhysicalPages(enum AddressPoolType *type*, const int *count*)
{
  // 根据类型选择合适的地址池
  if (type == AddressPoolType::KERNEL) {
    start = kernelPhysical.allocate(count);
  } else {
    start = userPhysical.allocate(count);
  }
  return (start == -1) ? 0 : start;
}
```

2. **AddressPool分配**

```
int AddressPool::allocate(const int *count*)
{
  // 调用位图分配，并将位图索引转换为物理地址
  uint32 start = resources.allocate(count);
  return (start == -1) ? -1 : (start * PAGE_SIZE + startAddress);
}
```

3. **BitMap核心分配算法**

```
int BitMap::allocate(const int *count*)
{
  // 首次适应算法：寻找第一个足够大的连续空闲区域
  while (index < length) {
  // 跳过已分配区域
  while (index < length && get(index)) ++index;
  // 检查连续空闲区域大小
  empty = 0;
  start = index;
  while ((index < length) && (!get(index)) && (empty < count)) {
  ++empty;
  ++index;
}
  // 找到合适区域，标记为已分配并返回
```

```
  if (empty == count) {
    for (int i = 0; i < count; ++i) {
        set(start + i, true);
      }
    return start;
  }
  }
}
```

**物理页释放过程**

物理页释放也是自顶向下的过程：

1. **从MemoryManager开始**

```
void MemoryManager::releasePhysicalPages(enum AddressPoolType *type*, const int *paddr*,
const int *count*)
{
  // 选择对应地址池进行释放
  if (type == AddressPoolType::KERNEL) {
    kernelPhysical.release(paddr, count);
  } else {
    userPhysical.release(paddr, count);
  }
}
```

2. **AddressPool释放**

```
void AddressPool::release(const int *address*, const int *amount*)
{
  // 将物理地址转换回位图索引并释放
  resources.release((address - startAddress) / PAGE_SIZE, amount);
}
```

3. **BitMap标记释放**

```
void BitMap::release(const int *index*, const int *count*)
{
  // 简单循环，将对应位清零(标记为未分配)
  for (int i = 0; i < count; ++i) {
    set(index + i, false);
  }
}
```

2. 在`setup.cpp`添加测试代码：

测试了物理页基本分配和释放，连续多页分配，用户内存分配与释放，分配超过最大值四种情况。

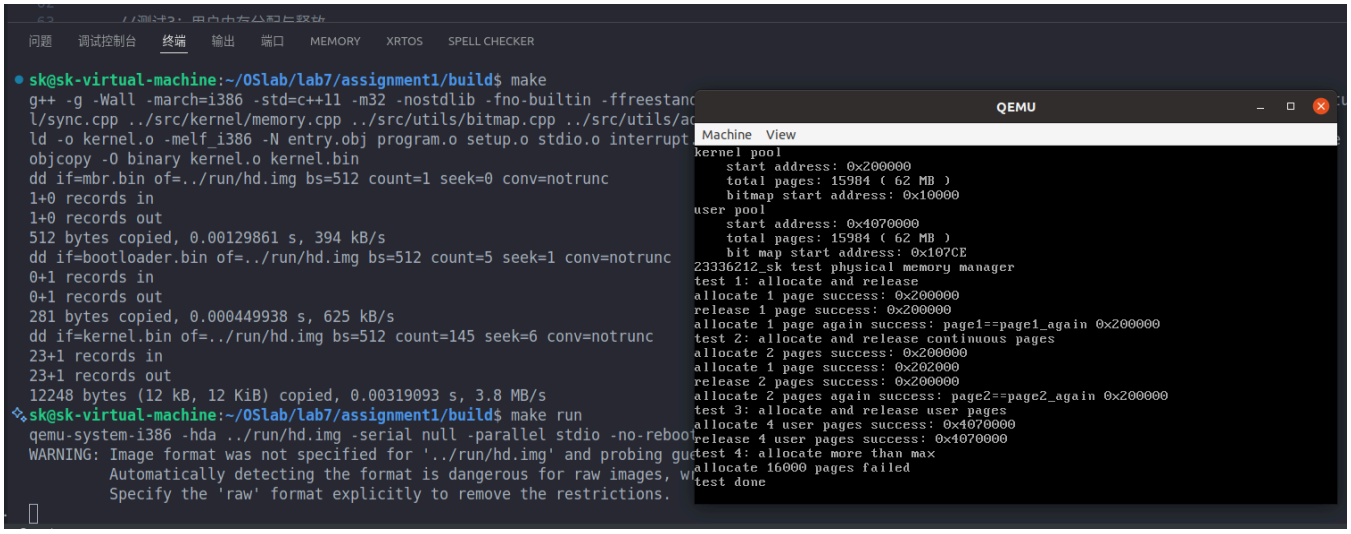
```
void first_thread(void *arg)
{
    printf("23336212_sk test physical memory manager\n");
    //测试1：基本分配和释放
```

```
    printf("test 1: allocate and release\n");
    int page1 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
    if(page1){
        printf("allocate 1 page success: 0x%x\n", page1);
        memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, page1, 1);
        printf("release 1 page success: 0x%x\n", page1);
        int page1_again = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
        if(page1==page1_again){
            printf("allocate 1 page again success: page1==page1_again 0x%x\n", page1_again);
        }else{
            printf("allocate 1 page again failed, new address: 0x%x\n", page1_again);
        }
        memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, page1_again, 1);
    }else{
        printf("allocate 1 page failed\n");
    }

    //测试2: 连续多页分配
    printf("test 2: allocate and release continuous pages\n");
    int page2 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 2);
    int page2_1 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
    if(page2){
        printf("allocate 2 pages success: 0x%x\n", page2);
        memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, page2, 2);
        printf("release 2 pages success: 0x%x\n", page2);
    }else{
        printf("allocate 2 pages failed\n");
    }

    int page2_again = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 2);
    if(page2==page2_again){
        printf("allocate 2 pages again success: page2==page2_again 0x%x\n", page2_again);
        memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, page2_again, 2);
    }else{
        printf("allocate 2 pages again failed, new address: 0x%x\n", page2_again);
    }
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, page2_1, 1);



    //测试3: 用户内存分配与释放
    printf("test 3: allocate and release user pages\n");
    int page3 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 4);
    if(page3){
        printf("allocate 4 user pages success: 0x%x\n", page3);
        memoryManager.releasePhysicalPages(AddressPoolType::USER, page3, 4);
        printf("release 4 user pages success: 0x%x\n", page3);
    }else{
        printf("allocate 4 user pages failed\n");
    }

    //测试4: 分配超过最大值
    printf("test 4: allocate more than max\n");
```

```
    int page4 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 16000);
    if(page4){
        printf("allocate 16000 pages success: 0x%x\n", page4);
        memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, page4, 16000);
        printf("release 16000 pages success: 0x%x\n", page4);
    }else{
        printf("allocate 16000 pages failed\n");
    }
    printf("test done\n");
    asm_halt();
}
```

实验结果如下：

测试情况为物理页单页分配和释放，连续分配和释放，用户内存分配与释放，分配超过最大值




测试情况1：可以看到，进程成功分配到1个物理页，起始地址为`0x200000`，然后成功释放物理页。

测试情况2：可以看到，进程先分配2个物理页和1个物理页，起始地址分别为`0x200000`和`0x202000`，符合连续分配的逻辑。然后释放2个物理页，再分配两个物理页，发现起始地址仍为`0x200000`，说明连续页释放逻辑是对的。

测试情况3：可以看到，进程分配2个物理页，起始地址为`0x407000`，与用户起始地址一致，说明用户内存分配成功。

测试情况4：可以看到，我们让进程分配16000个物理页，但是内核池中最多只有15984个物理页，超出了内存大小，所以分配失败。

## 实验任务2：二级分页机制的实现

代码放在`lab7/assignment2`中

- 任务要求：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。
- 实验思路：复现实验仓库里的代码，然后在分配页的函数中添加对应的输出信息。

- 实验步骤:

1. 复现代码

在x86架构中，二级分页的地址转换过程如下:

1. 线性地址被分为三部分：页目录索引(10位)、页表索引(10位)、页内偏移(12位)
2. 页目录索引用于在页目录表中查找页表的物理地址
3. 页表索引用于在页表中查找页框的物理地址
4. 页内偏移直接添加到页框地址上，得到最终物理地址

启动分页机制的流程如下所示:

- 规划好页目录表和页表在内存中的位置，然后初始化。
- 将页目录表的地址写入cr3。
- 将cr0的PG位置1。

2. 添加对应的输出信息

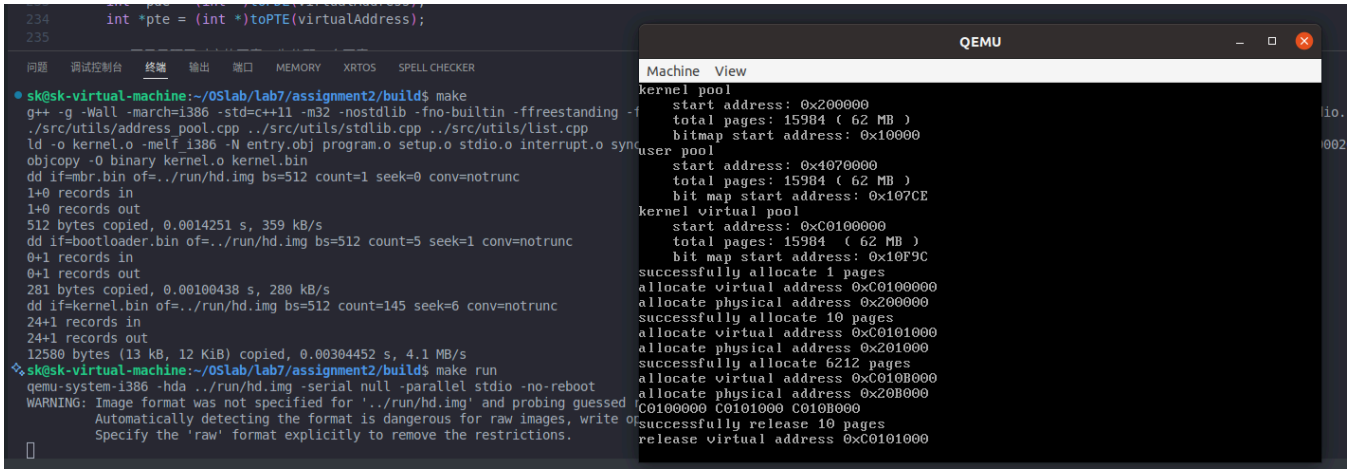
```
//在int MemoryManager::allocatePages(enum AddressPoolType type, const int count)中
//添加打印分配的虚拟页和物理页
    printf("successfully allocate %d pages\n", count);
    printf("allocate virtual address 0x%x\n", virtualAddress);
    printf("allocate physical address 0x%x\n", physicalPageAddress);

//在void MemoryManager::releasePages(enum AddressPoolType type, const int virtualAddress,
const int count)
//添加打印虚拟页释放
    printf("successfully release %d pages\n", count);
    printf("release virtual address 0x%x\n", virtualAddress);
```

3. 编写测试代码

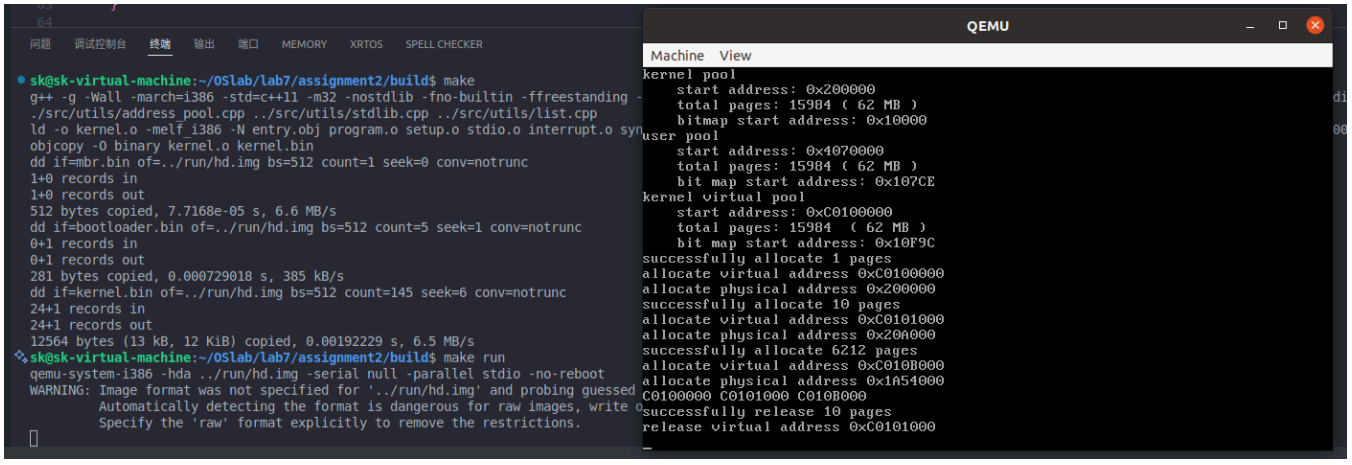
```
void first_thread(void *arg)
{
    char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
    char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
    char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 6212);

    printf("%x %x %x\n", p1, p2, p3);
    memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
    asm_halt();
}
```

实验结果如下:

这一张里的物理地址为起始地址




这里的物理页地址为最后一页的起始地址，用于说明页的数量分配是否正确。




可以看到内存虚拟池的起始地址为`0xc0100000`，依次分配到了1，10，6212（6212为我学号的后四位）个页，可以看到对应的物理页地址为`0x200000`，`0x20A000`，`0x1A5400`，地址之间差值符合1，10，6212个数；虚拟页地址为`0xc0100000`，`0xc0101000`，`0xc010B000`，这是它们对应的起始地址，所以也是符合逻辑的，第1页从`0xc0100000`起始，后面10页，从`0xc0101000`起始，再后面6212页从`0xc010B000`起始。内存的申请和释放成功实现。

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?

回答:

1. **节省内存空间**

- **一级页表**：需要为每个进程分配一个完整的页表，假设32位虚拟地址、4KB页大小，则每个进程的页表需要4MB，即使进程只用很少的虚拟空间，也要分配这么大。
- **二级页表**：只需为实际用到的虚拟空间分配页表。页目录只需4KB，页表按需分配，未用到的虚拟空间不分配页表，极大节省了内存。

2. **便于内存管理和扩展**

- 二级页表结构更灵活，便于操作系统动态分配和回收页表，适应进程虚拟空间的动态变化。
- 只需修改部分页表或页目录即可实现虚拟空间的扩展或收缩。

# 实验任务3：虚拟页内存管理的实现

代码放在`lab7/assignment3`中

- 任务要求：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。
- 实验思路：复现仓库代码，然后编写测例，测试是否能正确分配和释放页内存和物理内存,以及连续的虚拟地址能否映射到不连续的物理地址。
- 实验步骤：

## 1. 复现代码，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

### 一、虚拟页内存分配的三步过程

从代码来看，`MemoryManager::allocatePages()`函数实现了完整的虚拟内存分配过程，主要分为三个步骤：

**第一步：从虚拟地址池分配虚拟页**

```
// 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;
}
```

- 通过`allocateVirtualPages`从虚拟地址池中获取连续的虚拟页面
- 对于内核空间，从`kernelVirtual`地址池中分配(起始于0xc0100000)
- 分配失败则直接返回0

**第二步：为每个虚拟页分配物理页**

```
// 依次为每一个虚拟页指定物理页
for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE) {
  // 第二步：从物理地址池中分配一个物理页
  physicalPageAddress = allocatePhysicalPages(type, 1);
  if (!physicalPageAddress) {
    flag = false;
  }
```

- 循环为每个虚拟页分配对应的物理页

- 根据type参数决定从内核物理池还是用户物理池分配
- 对每个虚拟页单独分配物理页，确保灵活性

**第三步：建立虚拟页到物理页的映射**

```
// 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);
```

`connectPhysicalVirtualPage()`函数实现了映射建立：

1. 计算虚拟地址对应的页目录项和页表项地址
2. 如果页目录项未指向有效页表，先分配一个新页表
3. 将页表项设置为指向分配的物理页，设置访问权限标志

核心部分：

```
// 使页目录项指向页表
*pde = page | 0x7; // 设置U/S=1, R/W=1, P=1
// 使页表项指向物理页
*pte = physicalPageAddress | 0x7;
//此处0x7表示页面权限：用户可访问(U/S=1)、可读写(R/W=1)、存在(P=1)
```

错误处理与回滚：

分配过程中若任一步骤失败，会执行回滚操作，释放已分配资源：

```
// 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;
}
```

**二、虚拟页内存释放过程**

`releasePages()`函数实现了虚拟内存释放过程：

**第一步：释放物理页并清除映射**

```
// 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;
}
```

- 使用`vaddr2paddr`将虚拟地址转换为物理地址

- 调用`releasePhysicalPages`释放物理页
- 通过设置页表项为0，清除虚拟地址到物理地址的映射

**第二步：释放虚拟页**

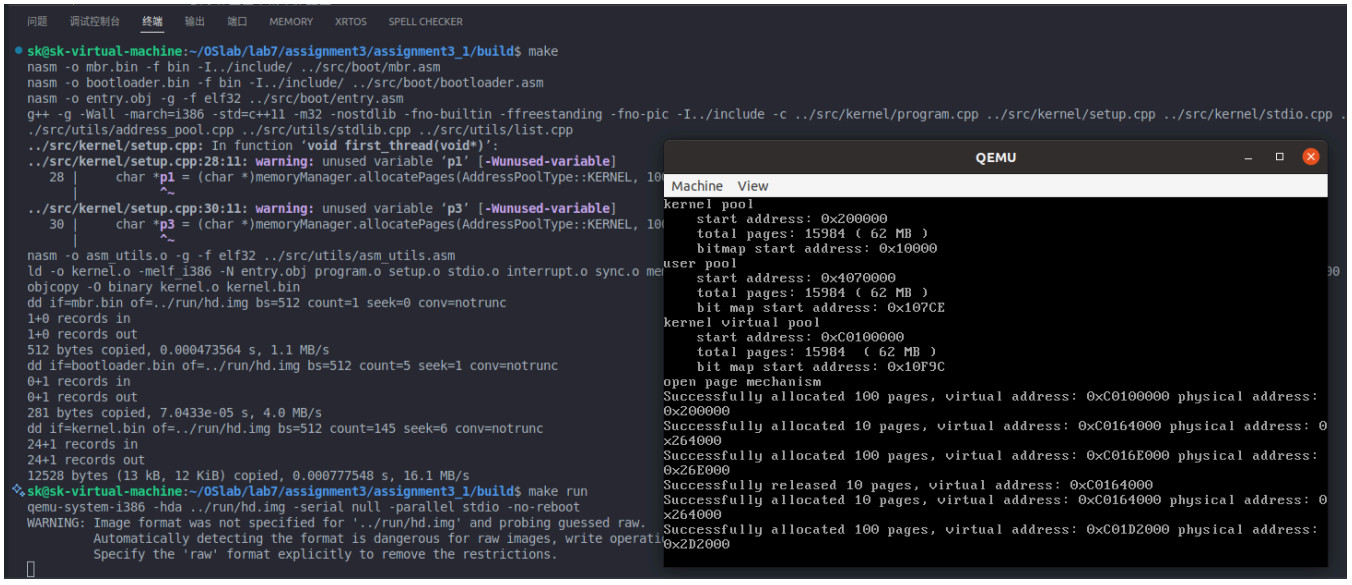
```
// 第二步，释放虚拟页
releaseVirtualPages(type, virtualAddress, count);
```

- 调用`releaseVirtualPages`将虚拟页标记为未使用
- 更新虚拟地址池的位图状态

## 2. 编写测试代码

```
void first_thread(void *arg)
{
    char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
    char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
    char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);

    memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
    //测试连续虚拟页分配可以对应不连续的物理页
    p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
    p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
    asm_halt();
}
```

实验结果如下




测例解析：

1. 测试虚拟页分配和释放：首先依次分配100，10，100个页，观察到对应的信息输出，说明虚拟页分配成功。然后释放中间的10页，可以看到成功释放。
2. 测试连续的虚拟地址可以对应不连续的物理地址：再次分配100，10页，可以观察到100页的起始地址是上面释放的10页的起始地址。在这种情况下，100页中的前10页分配到了之前的10页的空间，然后后90页分配到之前100页的地址后面，这样就证明了通过分页机制，我们可以把连续的地址变换到不连续的地址中。

### 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。

构造PDE(页目录项的虚拟地址):

首先，页目录项所在的物理页是页目录表，`virtual`的页目录号是`virtual[31:22]`，而每一个页目录项的大小是4个字节，这里的页目录项为141，因此页内偏移为

$$pde[11:0] = 4 \times virtual[31:22] = (4 \times 141)_{10} = 001000110100_2 = 234_{16}$$

接下来我们构造pde[21:12]。pde是位于页目录表的，那么，在现有的页表中，哪一个页表中的哪一个页表项指向了页目录表呢？实际上，页目录表是页表的一种特殊形式，页目录表的第1000个页目录项指向了页目录表。因此我们有

$$pde[21:12] = 1111101000_2$$

其中，下标2表示二进制，$2^{10}-1=1023$。

最后，我们来构造$pde[31:22]$。在构造$pde[21:12]$的时候，我们把页目录表当成了页表，而$pde[31:22]$是页目录项的序号，我们需要知道页目录表的哪一个页目录项指向了这个“页表”。显然，答案是第1023个页表项，因此我们有

$$pde[31:22] = 1111101000_2$$

则构建出来第141个页目录项的虚拟地址为:

$$pde = fA3E8234_{16}$$

构造PTE(页表项的虚拟地址):

首先，页表项所在的物理页是页表，`virtual`的页号是`virtual[21:12]`，而每一个页表项的大小是4个字节，这里对应的页号为109，因此我们有

$$pte[11:0] = 4 \times virtual[21:12] = (4 \times 109)_{10} = 000110110100_2 = 1B4_{16}$$

接下来我们构造pte[21:12]。pte是位于页表的，那么，在现有的页表中，哪一个页表的哪一个页表项指向了pte所在的页表呢？回忆起二级分页机制的地址变换过程我们发现，页目录表的第virutal[31:22]个页目录项指向了这个页表，这里页目录项为891，因此我们有

$$pte[21:12] = virtual[31:22] = 891_{10} = 001101111011_2 = 37B_{16}$$

最后，我们来构造pde[31:22]。我们实际上把页目录表当成了页表，而pte[31:22]是页目录项的序号，我们需要知道页目录表的哪一个页表项指向了这个“页表”显然，答案是第1000个页表项，因此我们有

$$pte[31:22] = 1111101000_2$$

则构建出来第891个页目录项指向的页表中第109个页表项的虚拟地址为:

$$pde = fA37B1B4_{16}$$

然后修改一下`toPDE`和`toPTE`函数

```
int MemoryManager::toPDE(const int virtualAddress)
{
    return (0xFA3E8000 + (((virtualAddress & 0xffc00000) >> 22) * 4));
}

int MemoryManager::toPTE(const int virtualAddress)
{
    return (0xFA000000 + ((virtualAddress & 0xffc00000) >> 10) + (((virtualAddress &
0x003ff000) >> 12) * 4));
}
```

1. **toPDE函数**:
    - 新基地址: `0xFA3E8000 = 0x3E8 << 22 | 0x3E8 << 12`
    - 这表示：页目录索引为1000，页表索引为1000
    - 通过这个虚拟地址，我们可以访问页目录表条目
2. **toPTE函数**:
    - 新基地址: `0xFA000000 = 0x3E8 << 22`
    - 加上`((virtualAddress & 0xffc00000) >> 10)`选择正确的页表
    - 再加上`(((virtualAddress & 0x003ff000) >> 12) * 4)`选择页表中的条目

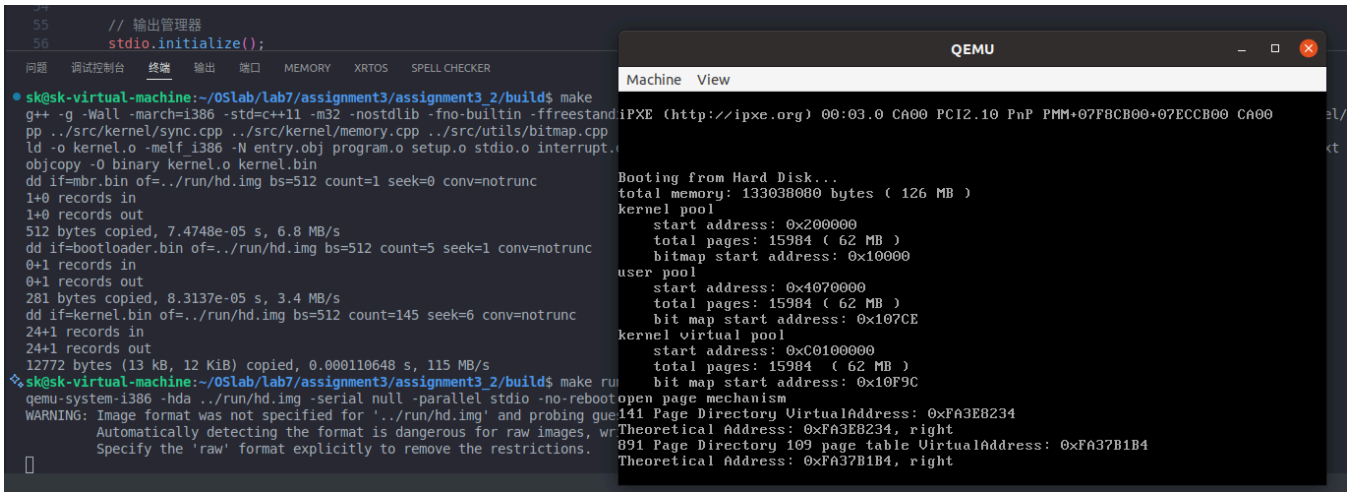
编写代码测试:

```
void first_thread(void *arg)
{
    // 测试页目录和页表虚拟地址构造
    // 1. 测试第141个页目录项
    int dir_index_141 = 141;
    int vaddr_141 = dir_index_141 << 22;  // 构造页目录索引为141的虚拟地址
    int pde_addr_141 = memoryManager.toPDE(vaddr_141);
    printf("141 Page Directory VirtualAddress: 0x%x\n", pde_addr_141);

    // 理论计算值
    int expected_pde_141 = 0xFA3E8000 + (dir_index_141 * 4);
    printf("Theoretical Address: 0x%x, %s\n",
           expected_pde_141,
           pde_addr_141 == expected_pde_141 ? "right" : "fault");

    // 2. 测试第891个页目录项指向的页表中第109个页表项
    int dir_index_891 = 891;
    int table_index_109 = 109;
    int vaddr_891_109 = (dir_index_891 << 22) | (table_index_109 << 12);
    int pte_addr_891_109 = memoryManager.toPTE(vaddr_891_109);
    printf("891 Page Directory 109 page table VirtualAddress: 0x%x\n", pte_addr_891_109);

    // 理论计算值
    int expected_pte_891_109 = 0xFA000000 + (dir_index_891 << 12) + (table_index_109 * 4);
    printf("Theoretical Address: 0x%x, %s\n",
           expected_pte_891_109,
           pte_addr_891_109 == expected_pte_891_109 ? "right" : "fault");
```

```
    asm_halt();
}
```

测试结果如下：

可以看到理论的虚拟地址与实际虚拟地址相同，说明构建成功。




# 实验任务4：页面置换算法的实现

代码放在`lab7/assignment4`中

- 任务要求：在Assignment 3的基础上，实现一种理论课上学习到的虚拟内存管理中的页面置换算法，在虚拟页内存中实现页面的置换，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)。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)。
- 实验思路：我这里实现的是FIFO置换算法

FIFO置换算法：

1. 维护一个队列
   系统为所有已分配的页面维护一个队列，队头是最早进入内存的页面，队尾是最新进入的页面。
2. 页面分配
   当有新的页面请求时，如果内存还有空闲页面，则直接分配，并将该页面加入队尾。
3. 页面置换
   如果内存已满（即队列已满），需要淘汰一个页面时，总是淘汰队头（最早进入内存的页面），然后把新页面插入队尾。

由于本次实验未涉及磁盘的读写，对于置换出来的页直接释放其物理页而不是换入外存。

- 实验步骤：

1. 实现队列，我用数组在原来代码下模拟一个FIFO队列

为了简单便于实现，我这里将空闲帧和FIFO队列集成在一起，空闲帧大小为3。

```
int FIFO[3];// FIFO队列
```

2. 添加必要的成员

```
int pagename[10];//页号名,用来标识对应的页
int virtual_address[10];// 页号名对应的虚拟地址，用来存储页对应的虚拟地址，便于在置换的时候进行地址替换
int size;// FIFO大小
```

3. 在`allocatePages`函数中实现FIFO算法：

核心思路如下：

1. **查找是否已存在该 name 的虚拟页**
   遍历 `pagename` 数组，如果找到与`name`相同的项，直接返回对应的虚拟地址，无需分配新页。
2. **如果不存在，则分配新的虚拟页**
   - 调用`allocateVirtualPages`分配新的虚拟页。
   - 判断 FIFO 队列是否已满（最多 3 个页）：
     - 未满：直接将新分配的虚拟地址、`name` 插入到 `virtual_address`、`pagename` 和 `FIFO` 数组末尾，`size++`。
     - 已满：
       - 选择 FIFO 队列头部（最早进入的页）为淘汰页（victim）。
       - 找到 victim 在 `virtual_address`/`pagename` 数组中的下标。
       - 调用 `releasePages` 释放 victim 的物理页和虚拟页。
       - FIFO 队列左移一位，末尾插入新分配的虚拟地址。
       - `virtual_address`/`pagename` 也同步左移，末尾插入新虚拟地址和`name`。
3. **分配物理页并建立映射**
   对新分配的虚拟页，逐页检查是否已分配物理页。若没有，则分配物理页并建立映射。若失败，回滚已分配的资源。

```
int MemoryManager::allocatePages(enum AddressPoolType type, const int count, int name)
{
    int virtualAddress = 0;
    int existIdx = -1;
    // 查找是否已存在该 name 的虚拟页
    for (int i = 0; i < this->size; i++) {
        if (this->pagename[i] == name) {
            virtualAddress = this->virtual_address[i];
            existIdx = i;
            break;
        }
    }

    // 如果不存在，则分配新的虚拟页
    if (virtualAddress == 0) {
        virtualAddress = allocateVirtualPages(type, count);
        if (!virtualAddress) return 0;

        // FIFO队列未满，直接插入
        if (this->size < 3) {
            this->virtual_address[this->size] = virtualAddress;
            this->pagename[this->size] = name;
```

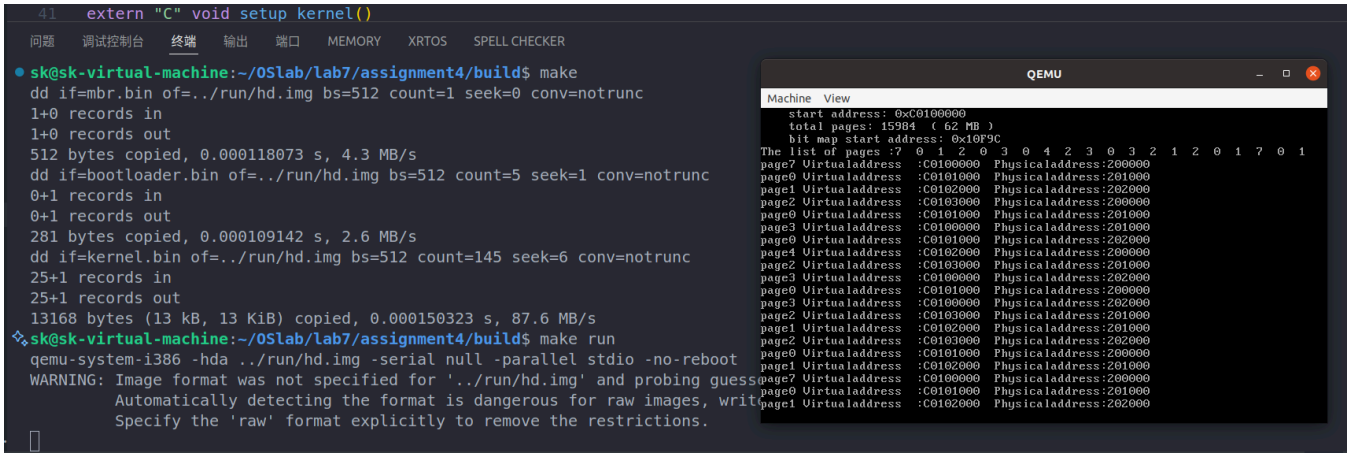
```
            this->FIFO[this->size] = virtualAddress;
            this->size++;
        } else {
            // FIFO队列已满，释放最早的虚拟页和物理页
            int victim_vaddr = this->FIFO[0];
            // 找到 victim_vaddr 在 virtual_address/pagename 中的下标
            int victim_idx = -1;
            for (int i = 0; i < 3; i++) {
                if (this->virtual_address[i] == victim_vaddr) {
                    victim_idx = i;
                    break;
                }
            }
            // 释放 victim 的物理页和虚拟页
            releasePages(type, victim_vaddr, count);

            // 队列左移
            for (int i = 1; i < 3; i++) {
                this->FIFO[i-1] = this->FIFO[i];
            }
            this->FIFO[2] = virtualAddress;

            // virtual_address/pagename 也要同步左移
            if (victim_idx != -1) {
                for (int i = victim_idx + 1; i < 3; i++) {
                    this->virtual_address[i-1] = this->virtual_address[i];
                    this->pagename[i-1] = this->pagename[i];
                }
                this->virtual_address[2] = virtualAddress;
                this->pagename[2] = name;
            }
        }
    }

    // 分配物理页并建立映射（如果还没有建立）
    int vaddress = virtualAddress;
    for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE) {
        int *pte = (int *)toPTE(vaddress);
        if ((*pte & 0x1) == 0) { // 还没有物理页
            int physicalPageAddress = allocatePhysicalPages(type, 1);
            if (!physicalPageAddress || !connectPhysicalVirtualPage(vaddress,
physicalPageAddress)) {
                // 分配失败，释放已分配的
                releasePages(type, virtualAddress, i);
                releaseVirtualPages(type, virtualAddress + i * PAGE_SIZE, count - i);
                return 0;
            }
        }
    }
    return virtualAddress;
}
```

测试的页面集合如下：

| 访问页面 | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 物理块 1 | 7 | 7 | 7 | 2 | | 2 | 2 | 4 | 4 | 4 | 0 | | | 0 | 0 | | | 7 | 7 | 7 |
| 物理块 2 | | 0 | 0 | 0 | | 3 | 3 | 3 | 2 | 2 | 2 | | | 1 | 1 | | | 1 | 0 | 0 |
| 物理块 3 | | | 1 | 1 | | 1 | 0 | 0 | 0 | 3 | 3 | | | 3 | 2 | | | 2 | 2 | 1 |
| 缺页否 | √ | √ | √ | √ | | √ | √ | √ | √ | √ | √ | | | √ | √ | | | √ | √ | √ |

图 3.23　FIFO 置换算法时的置换图

实验结果如下：




分析如下：

对应的页面和物理页地址如下，可以看到与上述FIFO置换图一致。

| 访问页面 | 0x200000 (物理块1) | 0x201000 (物理块2) | 0x202000 (物理块3) | 缺页否 |
|---|---|---|---|---|
| 7 | 7 | - | - | √ |
| 0 | 7 | 0 | - | √ |
| 1 | 7 | 0 | 1 | √ |
| 2 | 2 | 0 | 1 | √ |
| 0 | 2 | 0 | 1 | |
| 3 | 2 | 3 | 1 | √ |
| 0 | 2 | 3 | 0 | √ |
| 4 | 4 | 3 | 0 | √ |
| 3 | 4 | 3 | 0 | |
| 0 | 4 | 3 | 0 | |
| 3 | 4 | 3 | 0 | |
| 2 | 4 | 2 | 0 | √ |
| 3 | 4 | 2 | 3 | √ |

| 访问页面 | 0x200000 (物理块1) | 0x201000 (物理块2) | 0x202000 (物理块3) | 缺页否 |
| --- | --- | --- | --- | --- |
| 2 | 2 | 2 | 3 | √ |
| 0 | 2 | 0 | 3 | √ |
| 1 | 2 | 0 | 1 | √ |
| 7 | 7 | 0 | 1 | √ |
| 0 | 7 | 0 | 1 | |
| 1 | 7 | 0 | 1 | |

# 五、实验总结和心得体会

1. 通过这次实现，我对于虚拟地址和物理地址以及二者之间的转换有了清晰的认知，掌握二级分页机制的原理。特别是在实验三中，实践构建页目录表的虚拟地址和页表项的虚拟地址使我对于虚拟地址如何转换为物理地址的过程更加清晰。
2. 我觉得这次实验难度比较大，因为整个实验的内容和理论还是有点割离。在实现过程中还是有点难度的。而且，实验过程中还需考虑内存状态信息的存储、虚拟页地址和物理页地址的转换，以及二者之间真实的存储位置。
3. 在实现FIFO算法的时候，其实并不是真正的FIFO置换算法，只是在原有代码的基础上，实现了算法的模拟，因为时间有限，以后可能会花点时间从底层实现FIFO置换算法。因为真正的FIFO算法是要处理缺页的，但是实验代码基础上并没有设计缺页中断等解决方案，所以还是要从底层实现。

# 六、参考资料


1. https://www.bookstack.cn/read/simple_os_book/zh-chapter-3-implement_pages_mem_managment.md?wd=pa
2. https://zhuanlan.zhihu.com/p/480796773
3. https://blog.csdn.net/u012321457/article/details/80714640