# 1. 实验要求

## Assignment 1

复现参考代码，实现二级分页机制，并能够在虚拟机地址空间中进行内存管理，包括内存的申请和释放等，截图并给出过程解释。

## Assignment 2

参照理论课上的学习的物理内存分配算法如first-fit, best-fit等实现动态分区算法等，或者自行提出自己的算法。

## Assignment 3

参照理论课上虚拟内存管理的页面置换算法如FIFO、LRU等，实现页面置换，也可以提出自己的算法。

## Assignment 4

复现“虚拟页内存管理”一节的代码，完成如下要求。

- 结合代码分析虚拟页内存分配的三步过程和虚拟页内存释放。
- 构造测试例子来分析虚拟页内存管理的实现是否存在bug。如果存在，则尝试修复并再次测试。否则，结合测例简要分析虚拟页内存管理的实现的正确性。
- （**不做要求，对评分没有影响**）如果你有想法，可以在自己的理解的基础上，参考ucore，《操作系统真象还原》，《一个操作系统的实现》等资料来实现自己的虚拟页内存管理。在完成之后，你需要指明相比较于本教程，你的实现的虚拟页内存管理的特点所在。

# 2. 实验过程

## Assignment 1

使用cd命令，在终端中进入相应代码文件夹下的build目录，再使用make与make run命令编译运行即可。

## Assignment 2

利用链表数据结构，跟踪已分配的内存的地址，实现动态分区算法first-fit。

## Assignment 3

利用链表数据结构，跟踪页面分配的先后顺序，实现FIFO页面置换。

## Assignment 4

- 分析虚拟页内存分配的三步过程和虚拟页内存释放。
- 结合测例简要分析虚拟页内存管理的实现的正确性。

# 3. 关键代码

# Assignment 2

需要建立简单分段机制，利用链表结构，跟踪已分配的内存快所在位置，List类的声明代码如下：

```
#ifndef LIST_H
#define LIST_H

class ListItem
{
public:
    ListItem *previous;
    char *start;
    char *end;
    ListItem *next;
};

class List
{
public:
    ListItem head;
public:
    // 初始化List
    List();
    // 显式初始化List
    void initialize();
    // 返回List元素个数
    int size();
    // 返回List是否为空
    bool empty();
    // 返回指向List最后一个元素的指针
    // 若没有，则返回nullptr
    ListItem *back();
    // 将一个元素加入到List的结尾
    void push_back(ListItem *itemPtr);
    // 删除List最后一个元素
    void pop_back();
    // 返回指向List第一个元素的指针
    // 若没有，则返回nullptr
    ListItem *front();
    // 将一个元素加入到List的头部
    void push_front(ListItem *itemPtr);
    // 删除List第一个元素
    void pop_front();
    // 将一个元素插入到pos的位置处
    void insert(int pos, ListItem *itemPtr);
    // 删除pos位置处的元素
    void erase(int pos);
    void erase(ListItem *itemPtr);
    // 返回指向pos位置处的元素的指针
    ListItem *at(int pos);
    // 返回给定元素在List中的序号
    int find(ListItem *itemPtr);
};

ListItem *allocateListItem();

void releaseListItem(ListItem *temp);
```

```
#endif
```

由代码可见，ListItem中含有start和end数据成员，用于跟踪已分配内存的起始地址与结束地址。

List的实现如下：

```
#include "list.h"
#include "os_constant.h"
#include "stdio.h"

const int LISTITEM_SIZE = 16;
char LISTITEM_SET[LISTITEM_SIZE * MAX_PROGRAM_AMOUNT];
bool LISTITEM_SET_STATUS[MAX_PROGRAM_AMOUNT];

ListItem *allocateListItem()
{
    printf("allocate!\n");
    for (int i = 0; i < MAX_PROGRAM_AMOUNT; ++i)
    {
        if (!LISTITEM_SET_STATUS[i])
        {
            LISTITEM_SET_STATUS[i] = true;
            return (ListItem *)((int)LISTITEM_SET + LISTITEM_SIZE * i);
        }
    }

    return nullptr;
}

void releaseListItem(ListItem *temp)
{
    int index = ((int)temp - (int)LISTITEM_SET) / LISTITEM_SIZE;
    LISTITEM_SET_STATUS[index] = false;
}


List::List()
{
    head.next = head.previous = 0;
}

void List::initialize()
{
    head.next = head.previous = 0;
    for (int i = 0; i < MAX_PROGRAM_AMOUNT; ++i)
    {
        LISTITEM_SET_STATUS[i] = false;
    }
}

int List::size()
{
    ListItem *temp = head.next;
    int counter = 0;

    while (temp)
```

```
    {
        temp = temp->next;
        ++counter;
    }

    return counter;
}

bool List::empty()
{
    return size() == 0;
}

ListItem *List::back()
{
    ListItem *temp = head.next;
    if (!temp)
        return nullptr;

    while (temp->next)
    {
        temp = temp->next;
    }

    return temp;
}

void List::push_back(ListItem *itemPtr)
{
    ListItem *temp = back();
    if (temp == nullptr)
        temp = &head;
    temp->next = itemPtr;
    itemPtr->previous = temp;
    itemPtr->next = nullptr;
}

void List::pop_back()
{
    ListItem *temp = back();
    if (temp)
    {
        temp->previous->next = nullptr;
        temp->previous = temp->next = nullptr;
        releaseListItem(temp);
    }
}

ListItem *List::front()
{
    return head.next;
}

void List::push_front(ListItem *itemPtr)
{
    ListItem *temp = head.next;
    if (temp)
    {
```

```
        temp->previous = itemPtr;
    }
    head.next = itemPtr;
    itemPtr->previous = &head;
    itemPtr->next = temp;
}

void List::pop_front()
{
    ListItem *temp = head.next;
    if (temp)
    {
        if (temp->next)
        {
            temp->next->previous = &head;
        }
        head.next = temp->next;
        temp->previous = temp->next = nullptr;
        releaseListItem(temp);
    }

}

void List::insert(int pos, ListItem *itemPtr)
{
    if (pos == 0)
    {
        push_front(itemPtr);
    }
    else
    {
        int length = size();
        if (pos == length)
        {
            push_back(itemPtr);
        }
        else if (pos < length)
        {
            ListItem *temp = at(pos);

            itemPtr->previous = temp->previous;
            itemPtr->next = temp;
            temp->previous->next = itemPtr;
            temp->previous = itemPtr;
        }
    }
}

void List::erase(int pos)
{
    if (pos == 0)
    {
        pop_front();
    }
    else
    {
        int length = size();
        if (pos < length)
```

```
        {
            ListItem *temp = at(pos);

            temp->previous->next = temp->next;
            if (temp->next)
            {
                temp->next->previous = temp->previous;
            }
            releaseListItem(temp);
        }
    }
}

void List::erase(ListItem *itemPtr)
{
    ListItem *temp = head.next;

    while (temp && temp != itemPtr)
    {
        temp = temp->next;
    }

    if (temp)
    {
        temp->previous->next = temp->next;
        if (temp->next)
        {
            temp->next->previous = temp->previous;
        }
        releaseListItem(temp);
    }
}
ListItem *List::at(int pos)
{
    ListItem *temp = head.next;

    for (int i = 0; (i < pos) && temp; ++i, temp = temp->next)
    {
    }

    return temp;
}

int List::find(ListItem *itemPtr)
{
    int pos = 0;
    ListItem *temp = head.next;
    while (temp && temp != itemPtr)
    {
        temp = temp->next;
        ++pos;
    }

    if (temp && temp == itemPtr)
    {
        return pos;
    }
    else
```

```
    {
        return -1;
    }
}
```

由于还没有实现动态内存分配机制，这里新建的ListItem需要从划分好的用来存储ListItem的数组空间LISTITEM_SET中分配内存，并用LISTITEM_SET_STATUS跟踪已分配情况。

接着实现类BlockList，用于实现分段机制，声明如下：

```
class BlockList
{
public:
    List Blocks;
    char *blocklist;
    int space;
public:
    // 初始化
    BlockList();
    void initialize(char *blocklist, const int space);
    ListItem * allocate(const int count);
    // 释放第index个资源开始的count个资源
    void release(ListItem * item);
    char *getBlockList();
    int size() const;
private:
    // 禁止BlockList之间的赋值
    BlockList(const BlockList &) {}
    void operator=(const BlockList&) {}
};
```

实现如下：

```
BlockList::BlockList() {}

void BlockList::initialize(char *blocklist, const int space)
{
    this->blocklist = blocklist;
    this->space = space;
    this->Blocks.initialize();
}

ListItem * BlockList::allocate(const int count)
{
    if (count > space){
        return nullptr;
    }

    if (Blocks.head.next == 0){
        ListItem * b = allocateListItem();
        b->start = blocklist;
        b->end = blocklist + count;
        Blocks.push_back(b);
        return b;
    }
```

```
    ListItem *temp = Blocks.head.next;
    int i = 1;

    while (temp->next)
    {
        if (temp->next->start - temp->end >= count )
        {
            ListItem * b = allocateListItem();
            b->start = temp->end;
            b->end = b->start + count;
            Blocks.insert(i, b);
            return b;
        }
        temp = temp->next;
        i++;
    }
    if ( blocklist + space - temp->end >= count )
    {
        ListItem * b = allocateListItem();
        b->start = temp->end;
        b->end = b->start + count;
        Blocks.push_back(b);
        return b;
    }
    return nullptr;
}

void BlockList::release(ListItem * item)
{
    Blocks.erase(item);
}

char *BlockList::getBlockList()
{
    return (char *)blocklist;
}

int BlockList::size() const
{
    return space;
}
```

注意分配内存时，ListItem需要先在LISTITEM_SET中获取内存。

最后在MemoryManager中实现分段机制，使用first_fit分配算法。声明如下：

```
#ifndef MEMORY_H
#define MEMORY_H

#include "address_pool.h"
#include "bitmap.h"
#include "list.h"

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;
    // 内核虚拟地址池
    AddressPool kernelVirtual;
    // kernel partition
    BlockList kernelBlocks;
    // user partition
    BlockList userBlocks;

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);

    ListItem * allocatePhysicalBlocks(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);

    void releasePhysicalBlocks(enum AddressPoolType type, ListItem * block);

    // 获取内存总容量
    int getTotalMemory();

    // 开启分页机制
    void openPageMechanism();

    // 页内存分配
    int allocatePages(enum AddressPoolType type, const int count);

    int allocateBlocks(enum AddressPoolType type, const int count);

    // 虚拟页分配
    int allocateVirtualPages(enum AddressPoolType type, const int count);

    // 建立虚拟页到物理页的联系
    bool connectPhysicalVirtualPage(const int virtualAddress, const int
physicalPageAddress);

    // 计算virtualAddress的页目录项的虚拟地址
    int toPDE(const int virtualAddress);
```

```
    // 计算virtualAddress的页表项的虚拟地址
    int toPTE(const int virtualAddress);

    // 页内存释放

    void releasePages(enum AddressPoolType type, const int virtualAddress, const
int count);

    // 找到虚拟地址对应的物理地址
    int vaddr2paddr(int vaddr);

    // 释放虚拟页
    void releaseVirtualPages(enum AddressPoolType type, const int vaddr, const
int count);
};

#endif
```

实现如下：

```
#include "memory.h"
#include "os_constant.h"
#include "stdlib.h"
#include "asm_utils.h"
#include "stdio.h"
#include "program.h"
#include "os_modules.h"
#include "list.h"

MemoryManager::MemoryManager()
{
    initialize();
}

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);
    int kernelVirtualBitMapStart = userPhysicalBitMapStart + ceil(userPages, 8);

    kernelPhysical.initialize(
        (char *)kernelPhysicalBitMapStart,
        kernelPages,
        kernelPhysicalStartAddress);

    userPhysical.initialize(
        (char *)userPhysicalBitMapStart,
        userPages,
        userPhysicalStartAddress);

    kernelVirtual.initialize(
        (char *)kernelVirtualBitMapStart,
        kernelPages,
        KERNEL_VIRTUAL_START);

    kernelBlocks.initialize(
        (char *)kernelPhysicalStartAddress,
        kernelPages * PAGE_SIZE);

    userBlocks.initialize(
        (char *)userPhysicalStartAddress,
        userPages * PAGE_SIZE);

    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);

    printf("kernel virtual pool\n"
           "    start address: 0x%x\n"
           "    total pages: %d  ( %d MB ) \n"
           "    bit map start address: 0x%x\n",
           KERNEL_VIRTUAL_START,
           userPages, kernelPages * PAGE_SIZE / 1024 / 1024,
           kernelVirtualBitMapStart);
}

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);
    }
}

ListItem * MemoryManager::allocatePhysicalBlocks(enum AddressPoolType type,
const int count)
{
    ListItem * start = nullptr;

    if (type == AddressPoolType::KERNEL)
    {
        start = kernelBlocks.allocate(count);
    }
    else if (type == AddressPoolType::USER)
    {
        start = userBlocks.allocate(count);
    }

    return (start == nullptr) ? 0 : start;
}

void MemoryManager::releasePhysicalBlocks(enum AddressPoolType type, ListItem *
block)
{
    if (type == AddressPoolType::KERNEL)
    {
        kernelBlocks.release(block);
    }
    else if (type == AddressPoolType::USER)
    {

        userBlocks.release(block);
    }
```

```
}

int MemoryManager::getTotalMemory()
{

    if (!this->totalMemory)
    {
        int memory = *((int *)MEMORY_SIZE_ADDRESS);
        // ax寄存器保存的内容
        int low = memory & 0xffff;
        // bx寄存器保存的内容
        int high = (memory >> 16) & 0xffff;

        this->totalMemory = low * 1024 + high * 64 * 1024;
    }

    return this->totalMemory;
}

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");
}

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;
}

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;
}
```

```
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;
}

int MemoryManager::toPDE(const int virtualAddress)
{
    return (0xfffff000 + (((virtualAddress & 0xffc00000) >> 22) * 4));
}

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

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);
}

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

# Assignment 3

使用链表结构跟踪页的分配先后顺序，用于实现FIFO页面置换算法，声明如下：

```
#ifndef LIST_H
#define LIST_H

class ListItem
{
public:
    ListItem *previous;
    int index;
    int count;
    ListItem *next;
};


class List
{
public:
    ListItem head;

public:
    // 初始化List
    List();
    // 显式初始化List
    void initialize();
    // 返回List元素个数
    int size();
    // 返回List是否为空
    bool empty();
    // 返回指向List最后一个元素的指针
    // 若没有，则返回nullptr
    ListItem *back();
    // 将一个元素加入到List的结尾
    void push_back(ListItem *itemPtr);
    // 删除List最后一个元素
    void pop_back();
```

```
    // 返回指向List第一个元素的指针
    // 若没有，则返回nullptr
    ListItem *front();
    // 将一个元素加入到List的头部
    void push_front(ListItem *itemPtr);
    // 删除List第一个元素
    void pop_front();
    // 将一个元素插入到pos的位置处
    void insert(int pos, ListItem *itemPtr);
    // 删除pos位置处的元素
    void erase(int pos);
    void erase(ListItem *itemPtr);
    // 返回指向pos位置处的元素的指针
    ListItem *at(int pos);
    // 返回给定元素在List中的序号
    int find(ListItem *itemPtr);
    int find(int index);
};

ListItem *allocateListItem();

void releaseListItem(ListItem *temp);

#endif
```

其中ListItem包含数据成员index和count，分别为页面的下标和连续分配页面的数量。实现代码与assignment 2类似，这里不再赘述了。

接下来在BitMap与AddressPool中实现FIFO置换算法，代码如下：

```
void BitMap::releaseFIFO(const int count)
{
    int released = 0;
    while( released < count )
    {
       released += stack.head.next->count;
       release(stack.head.next->index, stack.head.next->count);
    }
    //printf("Released %d pages FIFO\n", released);
}
```

```
void AddressPool::releaseFIFO(const int amount)
{
    resources.releaseFIFO(amount);
}
```

最后在MemoryManager中完成页面置换的实现，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;

    int i = 0;
    // 依次为每一个虚拟页指定物理页
    while( i < count)
    //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
            releasePagesFIFO(type, 1);
            // 剩余的页表未指定物理页
            //releaseVirtualPages(type, virtualAddress + i * PAGE_SIZE, count - i);
            //return 0;
        }
        else{
            ++i, vaddress += PAGE_SIZE;
        }
    }

    return virtualAddress;
}

void MemoryManager::releasePagesFIFO(enum AddressPoolType type, const int count)
{
    if( type == AddressPoolType::KERNEL)
    {
        kernelVirtual.releaseFIFO(count);
        kernelPhysical.releaseFIFO(count);
    }
    else{
        userPhysical.releaseFIFO(count);
    }
}
```

注意，由于要实现页面置换策略，内核虚拟内存需要比物理内存大，由于实验需要，我们将freeMemory定为200个页，作为物理内存大小（内核+用户），内核虚拟内存大小不变，代码如下：

```
    // 剩余的空闲的内存
    int freeMemory = 4096*200;

    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);
    int kernelVirtualBitMapStart = userPhysicalBitMapStart + ceil(userPages, 8);

    kernelPhysical.initialize(
        (char *)kernelPhysicalBitMapStart,
        kernelPages,
        kernelPhysicalStartAddress);

    userPhysical.initialize(
        (char *)userPhysicalBitMapStart,
        userPages,
        userPhysicalStartAddress);

    kernelVirtual.initialize(
        (char *)kernelVirtualBitMapStart,
        (totalMemory - usedMemory)/PAGE_SIZE,
        KERNEL_VIRTUAL_START);
```

# Assignment 4

根据实验文档，页内存分配分为以下3步。

- **从虚拟地址池中分配若干连续的虚拟页。**
- **对每一个虚拟页，从物理地址池中分配1页。**
- **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。**

负责页内存分配的函数如下所示。

```
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;
}
```

下面分别来看页内存分配的每个步骤。

**第一步，从虚拟地址池中分配若干连续的虚拟页。**虚拟页的分配通过函数`allocateVirtualPages`来实现：

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

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

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

此时能够分配页内存的地址池只有内核虚拟地址池。因此，对于其他类型的地址池，一律返回0，即分配失败。同时，文档中解释了`MemoryManager`没有用户虚拟地址空间的原因：每一个进程都有自己的用户虚拟地址池，因此用户虚拟地址池并不是全局的，而是被放到了PCB中。内存管理器`MemoryManager`只需要关心全局的地址空间即可，即用户物理地址空间，内核虚拟地址空间，内核物理地址空间，由此而

产生了3个地址池`kernelPhysical`，`kernelVirtual`和`userPhysical`。

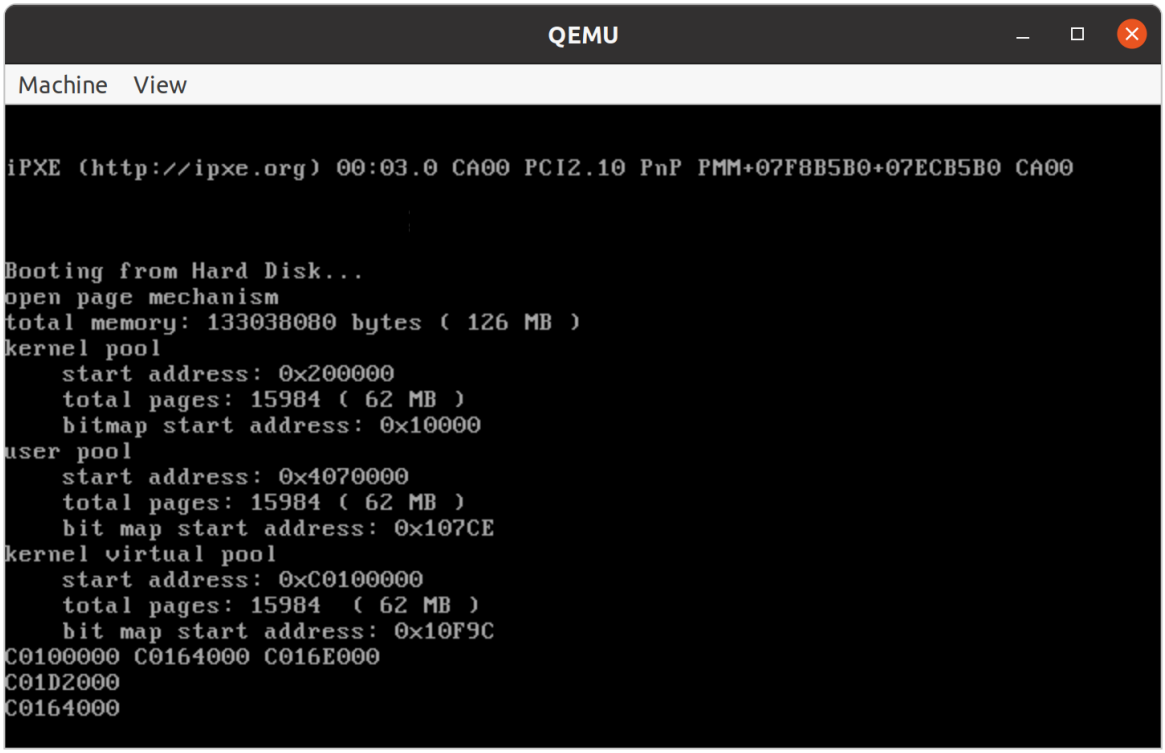
**第二步，对每一个虚拟页，从物理地址池中分配1页。**物理页的分配通过函数`allocatePhysicalPages`来实现。

**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。**建立虚拟页到物理页的映射关系通过函数`connectPhysicalVirtualPage`来实现，如下所示。

```
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;
}
```

# 4. 实验结果

## Assignment 1 & 4




对应代码如下：

```
    char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::USER, 100);
    char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::USER, 10);
    char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::USER, 100);

    printf("%x %x %x\n", p1, p2, p3);

    memoryManager.releasePages(AddressPoolType::USER, (int)p2, 10);
    p2 = (char *)memoryManager.allocatePages(AddressPoolType::USER, 100);

    printf("%x\n", p2);

    p2 = (char *)memoryManager.allocatePages(AddressPoolType::USER, 10);

    printf("%x\n", p2);
```

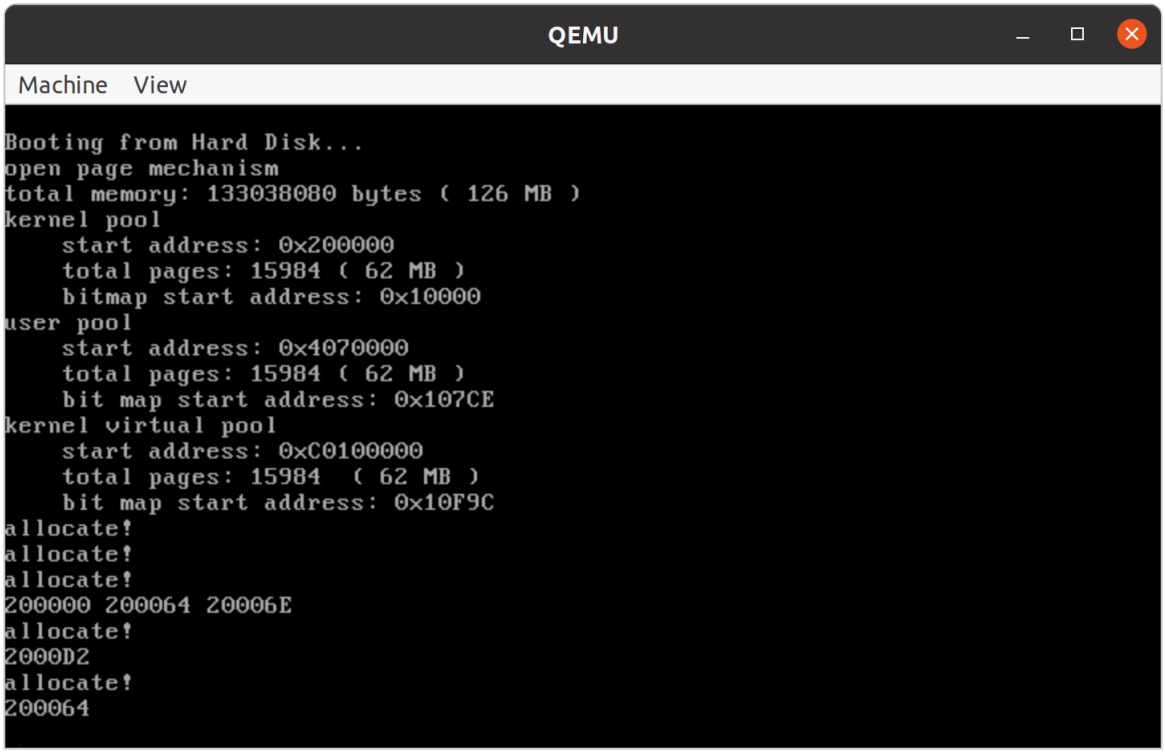
如图所示，成功实现了分页机制并进行了虚拟页内存分配。

代码中首先分配了100个页，10个页，100个页，打印出的地址C0100000, C0164000, C016E000（虚拟地址）间隔分别为100个页，10个页，分配正确。

接着释放了中间的10页，再分配了100页，打印出的地址C01D2000与C016E000间隔100个页，分配正确。

最后又分配了10页，这次MemoryManager又重新利用了已释放的10页空间，打印出的地址为C0164000。

# Assignment 2




```
    ListItem *p1 = memoryManager.allocatePhysicalBlocks(AddressPoolType::KERNEL,
100);
    ListItem *p2 = memoryManager.allocatePhysicalBlocks(AddressPoolType::KERNEL,
10);
    ListItem *p3 = memoryManager.allocatePhysicalBlocks(AddressPoolType::KERNEL,
100);

    printf("%x %x %x\n", p1->start, p2->start, p3->start);

    memoryManager.releasePhysicalBlocks(AddressPoolType::KERNEL, p2);
    p2 = memoryManager.allocatePhysicalBlocks(AddressPoolType::KERNEL, 100);

    printf("%x\n", p2->start);

    p2 = memoryManager.allocatePhysicalBlocks(AddressPoolType::KERNEL, 10);

    printf("%x\n", p2->start);
```

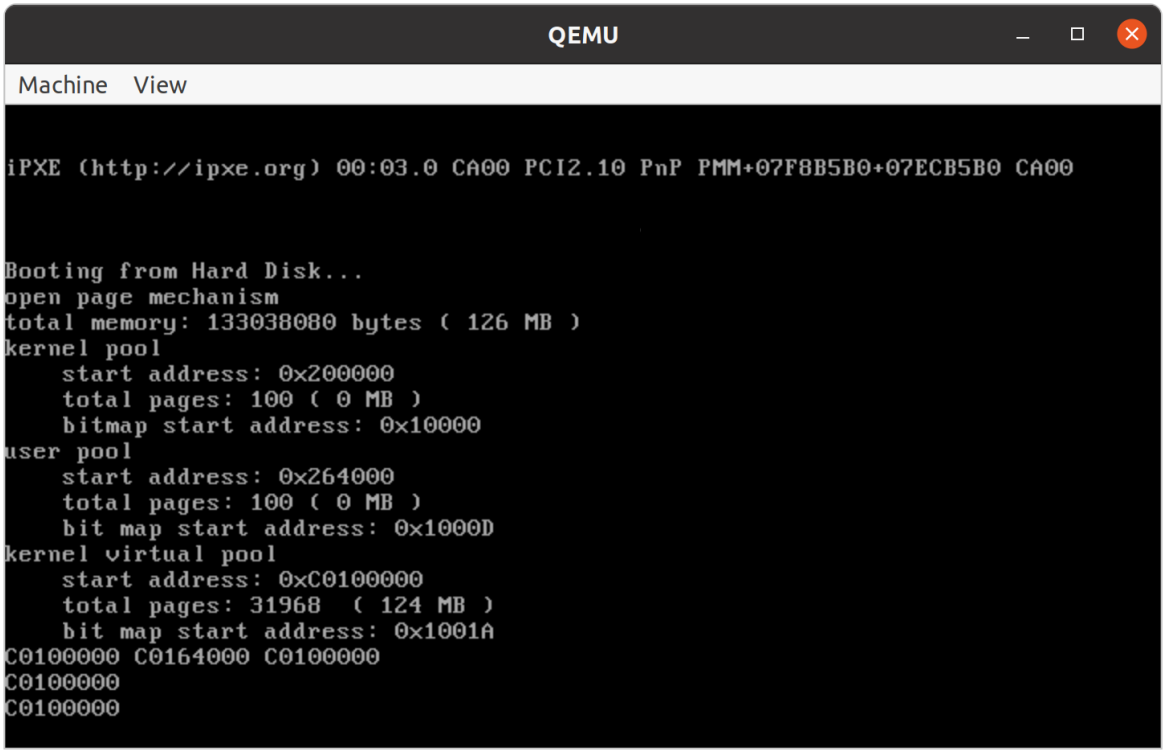
如图所示，动态分区算法按字节分配内存。

代码中首先分配了100个字节，10个字节，100个字节，打印出的地址200000, 200064, 20006E（虚拟地址）间隔分别为100字节，10个字节，分配正确。

接着释放了中间的10个字节，再分配了100个字节，打印出的地址2000D2与20006E间隔100个字节，分配正确。

最后又分配了10个字节，这次MemoryManager又重新利用了已释放的10个字节空间，打印出的地址为200064。

# Assignment 3




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

    printf("%x %x %x\n", p1, p2, p3);

    memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
    p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);

    printf("%x\n", p2);

    p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);

    printf("%x\n", p2);
```

如图所示，成功实现了页面置换算法。

代码中首先分配了100个页，10个页，100个页，打印出的地址C0100000, C0164000, C0100000（虚拟地址）。这是因为内核物理地址空间只有100个页，所以分配p3时，按照FIFO的算法，将p1释放了（由于返回的是虚拟地址，此处释放时将p1的虚拟内存也释放了，否则我们看到的p3虚拟地址与没有页面置换时相同）。

接着释放了中间的10页，再分配了100页，打印出的地址为C0100000，此时已将p3的物理内存释放。

最后又分配了10页，这次MemoryManager又将之前p2的10页物理内存释放，打印出的地址为C0100000。

# 5. 实验总结

## 5.1 实验中遇到的问题

## 1. BitMap为什么不使用静态分配内存

> 为什么我们需要从外界向BitMap提供存储区域呢？这是因为我们使用BitMap来实现内存管理，所以我们无法使用动态内存分配来在BitMap的初始化函数中分配一个存储区域。在后面的实现中可以看到，我们会在内存中手动划分出一块区域来存储BitMap用来标识资源分配情况的位。

BitMap不直接用声明数组的方式实现，是因为此时我们还不知道内存有多大，所以不能确定静态分配的大小。另外，BitMap需要和进程内存地址空间隔离开来，需要从外界提供。

## 2. 参考代码中为什么内核虚拟内存和物理内存一样大

代码如下：

```
    kernelPhysical.initialize(
        (char *)kernelPhysicalBitMapStart,
        kernelPages,
        kernelPhysicalStartAddress);

    userPhysical.initialize(
        (char *)userPhysicalBitMapStart,
        userPages,
        userPhysicalStartAddress);

    kernelVirtual.initialize(
        (char *)kernelVirtualBitMapStart,
        kernelPages,
        KERNEL_VIRTUAL_START);
```

内核的虚拟内存和物理内存一样大，是因为示例代码中还没有实现页面置换策略，不存在换入换出，所以一样大。

# 5.2 实验总结

在这次实验中，我进一步深入了解了理论课上学到的操作系统内存管理中关于分段和分页的相关内容，进一步熟悉了 Ubuntu 下的开发，接触到了操作系统在这一模块的实现的一些思想方法。我同时也锻炼了自己的分析和解决问题，以及请教他人的沟通能力。感谢老师和助教在我遇到困难的时候的耐心解答，我会再接再厉，争取学好这门课程。