

目录

1	实验概述	2
2	实验任务一	2
2.1	实验过程	2
2.2	实验结果展示	3
3	实验任务二	3
3.1	动态内存分配算法	3
3.2	实验结果展示	5
4	实验任务三	6
4.1	开启缺页中断	6
4.2	延迟分配物理页	7
4.3	请求调页机制 + 缺页中断处理 +FIFO 页面置换	7
4.4	时钟置换算法 (第二次机会算法)	9
4.5	LRU 页面置换	11
4.6	实验结果展示	12
4.6.1	FIFO 页面置换	12
4.6.2	时钟页面置换	13
4.7	LRU 页面置换	14
5	实验任务四	15
5.1	虚拟页内存分配的三步过程	15
5.2	虚拟页内存释放	16
5.3	虚拟页管理实现存在的 bug1: 未实现互斥	17
5.4	bug2:769-1022 个页目录项对应的页表未初始化	17
5.5	修复 bug1	17
5.6	修复 bug2	18
6	实验总结	19
6.1		19
6.2		19
6.3		19

# 1 实验概述

- 实验任务一：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
- 实验任务二：实现物理内存动态分配算法：首次适应、最佳适应、最差适应
- 实验任务三：实现虚拟内存管理的页面置换算法:FIFO、时钟置换算法、LRU 置换算法
- 实验任务四：复现“虚拟页内存管理”的代码，结合代码分析内存分配的过程与虚拟也内存释放，并构造测试例子分析虚拟页内存管理的实现是否存在 bug，并修复

## 2 实验任务一

### 2.1 实验过程

在开启分页之前，要先规划好内核所在位置与页表的位置：内核程序存放在物理内存的 0-1MB 处，因此要先建立好 0-1MB 的虚拟地址到物理地址的恒等映射：

0-1MB 的虚拟地址对应第 0 个页目录项以及第 0 个页表的前 256 项，将它映射到物理内存的 0-1MB 处

```
int *directory = (int *)PAGE_DIRECTORY;
int *page = (int *) (PAGE_DIRECTORY + PAGE_SIZE);
memset(directory, 0, PAGE_SIZE);
memset(page, 0, PAGE_SIZE);
int address = 0;
for (int i = 0; i < 256; ++i)
{
    // U/S = 1, R/W = 1, P = 1
    page[i] = address | 0x7;
    address += PAGE_SIZE;
}
```

将 0xc0100000 开始的高 3GB-4GB 地址划分为内核地址区域，因此将从第 768 个页目录表开始划分为内核地址区域，同时为了便于在虚拟地址中寻找页目录表，将第 1023 个页目录项指向页目录表的地址。

```
// 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;
```

规划完地址后，便可以在硬件层面开启分页，即将页目录表的地址写入 CR3, 并在 CR0 中置 PG 位 =1，开启分页。

```
1 ; void asm_init_page_reg(int *directory);
2 asm_init_page_reg:
3 push ebp
4 mov ebp, esp
5
6 push eax
7
8 mov eax, [ebp + 4 * 2]
9 mov cr3, eax ; 放入页目录表地址
10 mov eax, cr0
11 or eax, 0x80000000
12 mov cr0, eax ; 置PG=1，开启分页机制
```

```

13
14 pop eax
15 pop ebp
16
17 ret

```

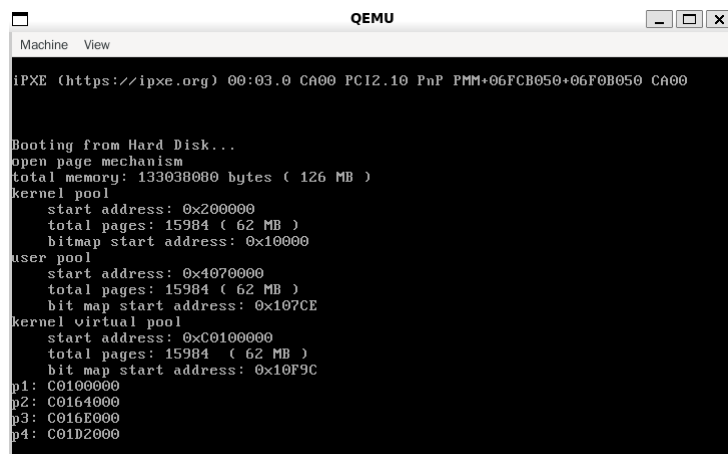
在内存的申请与释放中，用地址池来管理分配的内存的地址，用位图来压缩存储资源的状态以及进行资源的分配。在 setup.cpp 中进行内存资源的动态申请、释放：

```

char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
printf("p1: %x\n", p1);
char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
printf("p2: %x\n", p2);
char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
printf("p3: %x\n", p3);
memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
char *p4 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
printf("p4: %x\n", p4);

```

## 2.2 实验结果展示



如图，已经成功开启分页机制，且分别给内核用户的物理内存、内核的虚拟内存分配了页面，开启分页机制。并成功完成了内核虚拟空间的动态内存分配。

## 3 实验任务二

### 3.1 动态内存分配算法

本实验中，实现以页面个数为单位的动态内存分配。

在示例的地址池管理中，位图中的 allocate 函数是以首次适应的方式进行分配的，找到第一个有连续 count 个页的空间进行分配，在此我们扩展为支持首次适应，最佳适应，最差适应三种方式：

```

enum AllocateType
{
    FIRST_FIT,
    BEST_FIT,
    WORST_FIT
};

```

首次适应算法：从头开始找到第一个连续 count 个空闲页面的块进行分配：

```

int BitMap::firstFit(const int count)
{
    int cnt=0;
    for (int i=0;i<length;i++) {

```

```

        if (get(i)) {
            cnt=0;
        } else {
            cnt++;
            if (cnt==count) {
                return i-count+1;
            }
        }
    }
    return -1;
}

```

最佳适应算法: 从所有满足拥有连续大于 count 个页面的块中, 找最小的块进行分配:

```

int BitMap::bestFit(const int count)
{
    int min=1e9;
    int beststart=-1;
    int cnt=0;
    for (int i=0;i<length;i++) {
        if (get(i)) {
            if (cnt>=count) { //符合条件, 则找最小的
                if (cnt-count<min) {
                    min=cnt-count;
                    beststart=i-cnt;
                }
            }
            cnt=0;
        } else {
            cnt++;
        }
    }
    if (cnt>=count) {
        if (cnt-count<min) {
            min=cnt-count;
            beststart=length-cnt;
        }
    }
    return beststart;
}

```

最差适应算法: 从所有满足拥有连续大于 count 个页面的块中, 找最大的块进行分配:

```

int BitMap::worstFit(const int count)
{
    int max=0;
    int worststart=-1;
    int cnt=0;
    for (int i=0;i<length;i++) {
        if (get(i)) {
            if (cnt>=count) {
                if (cnt-count>max) {
                    max=cnt-count;
                    worststart=i-cnt;
                }
            }
            cnt=0;
        } else {
            cnt++;
        }
    }
    if (cnt>=count) {
        if (cnt-count>max) {
            max=cnt-count;
            worststart=length-cnt;
        }
    }
}

```

```

    }
    return worststart;
}

```

将三个方法集成到 allocate 函数中，根据选择来调用：

```

int BitMap::allocate(const int count)
{
    if (count == 0)
        return -1;
    int index=-1;
    switch (allocateType)
    {
        case FIRST_FIT:
            index=firstFit(count);
            break;
        case BEST_FIT:
            index=bestFit(count);
            break;
        case WORST_FIT:
            index=worstFit(count);
            break;
    }
    if (index!=-1) {
        for (int i=index;i<index+count;i++) {
            set(i,true);
        }
    }
    if (index!=-1) printf("Allocate count: %d, page %d to %d\n",count,index,index+count-1);
    return index;
}

```

## 3.2 实验结果展示

从实验任务一可知，总页数为 15984，构造的测试样例以及理论上三种算法的分配情况如下：

操作	首次适应算法	最佳适应算法	最差适应算法
分配 11000 页	0-10999	0-10999	0-10999
回收 2000-9999 共 8000 页	-	-	-
分配 3500 页	2000-5499	11000-14499	2000-5499
分配 2500 页	5500-7999	2000-4499	11000-13499

实验结果如下：



```
QEMU
Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+06FCB050+06F0B050 CA00

Booting from Hard Disk...
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
Allocate count: 11000, page 0 to 10999
Release count: 8000, page 2000 to 9999
Allocate count: 3500, page 11000 to 14499
Allocate count: 2500, page 2000 to 4499
```

最差适应算法:

```
QEMU
Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+06FCB050+06F0B050 CA00

Booting from Hard Disk...
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
Allocate count: 11000, page 0 to 10999
Release count: 8000, page 2000 to 9999
Allocate count: 3500, page 2000 to 5499
Allocate count: 2500, page 11000 to 13499
```

## 4 实验任务三

页面置换是在请求调页机制下，若访问的地址对应的物理页不在内存中，则从外存中调物理页到内存中。

示例中的虚拟页管理是一建立虚拟页则马上分配对应的一个物理页并建立映射关系。而请求调页机制下需要延迟物理页分配，在首次访问虚拟页地址时通过中断方式分配物理页。

若物理页已分配满，则需要根据页面置换算法，选择一个页面换出外存 (在本实验中未实现换出外存以及回写法的处理，只是简单的把物理页面置换掉)。

### 4.1 开启缺页中断

按照 Lab4-中断的方法，注册并开启缺页中断:

首先在中断处理器中，注册缺页中断:

```
setInterruptDescriptor(14, (uint32)asm_page_fault_handler, 0);
```

然后在 asm 中执行中断处理函数，这里要注意，缺页中断与时钟中断的不同点在于它还回返回一个中断的错误码在栈中，返回前需要进行弹栈弹出错误码:

```
1 asm_page_fault_handler:
2     pushad
3     call c_page_interrupt_handler
4     popad
5     ; 处理错误码
6     add esp, 4
7     iret
```

## 4.2 延迟分配物理页

分配虚拟页时不立即分配物理页

```
int MemoryManager::allocatePages(enum AddressPoolType type, const int count)
{
    int virtualAddress = allocateVirtualPages(type, count);
    if (!virtualAddress)
    {
        return 0;
    }

    return virtualAddress;
}
```

## 4.3 请求调页机制 + 缺页中断处理 + FIFO 页面置换

FIFO 页面置换只需要维护一个队列，页面分配时将其加入队列

```
struct item {
    item(int virtualAddress, int physicalAddress)
    {
        this->virtualAddress=virtualAddress;
        this->physicalAddress=physicalAddress;
    }
    item(){}
    int virtualAddress;
    int physicalAddress;
    int valid;
};

class Queue
{
public:
    item queue[MAX_QUEUE_SIZE];
    int front;
    int rear;
    Queue()
    {
        front=0;
        rear=0;
    }
    void push(item value)
    {
        queue[rear]=value;
        rear=(rear+1)%MAX_QUEUE_SIZE;
    }
    void pop()
    {
        front=(front+1)%MAX_QUEUE_SIZE;
    }
    item getFront()
    {
        while(size()>0&&queue[front].valid==0)
        {
            front=(front+1)%MAX_QUEUE_SIZE;
        }
        return queue[front];
    }
    bool empty()
    {
        return front==rear;
    }
    int size()
    {

```

```

        return (rear-front+MAX_QUEUE_SIZE)%MAX_QUEUE_SIZE;
    }
};

```

发生缺页中断时，中断发生所在的虚拟地址会存入寄存器 CR2，因此我们在中断处理时可通过 CR2 寄存器获取发生中断的虚拟地址：

```

uint32 read_cr2() {
    uint32 value;
    asm volatile("mov %%cr2, %0" : "=r"(value));
    return value;
}

```

系统会自动给访问过的页面进行 TLB 缓存，因此在释放页面之前要先刷新被替换掉的页面对应的 TLB

```

static inline void flush_tlb_single(int addr) {
    asm volatile("invlpg (%0)" :: "r" (addr) : "memory");
}

```

约定只有手动分配过的虚拟地址才是合法的虚拟地址，否则定义此地址不合法，因此在缺页中断处理步骤：

- 从 CR2 寄存器读取虚拟地址
- 判断虚拟地址是否合法 (是否分配过)
- 为此虚拟地址分配物理页
- 若可分配的物理页已满，则进行页面置换，弹出队首元素并清空其页表，同时清空 TLB 缓存，然后为虚拟地址新分配物理页
- 将新分配的物理页加入队列
- 将虚拟页与物理页的建立页目录表、页表联系

中断处理函数代码如下：

```

extern "C" void c_page_interrupt_handler()
{
    uint32 cr2=read_cr2();
    printf("page fault on virtual address :0x%x\n",cr2);
    bool is_valid = false;

    // 计算地址所在的页面
    uint32 page_addr = cr2 & 0xFFFFF000;

    // 检查是否是内核地址空间 (3GB以上)
    if (cr2 >= 0xC0000000) {
        // 内核地址空间，检查该地址是否在内核虚拟地址池的已分配范围内
        int page_index = (page_addr - memoryManager.kernelVirtual.startAddress) / 4096;
        if (page_index >= 0 && page_index < memoryManager.kernelVirtual.resources.length) {
            // 检查该页是否已分配 (即位图中该位是否为1)
            is_valid = memoryManager.kernelVirtual.resources.get(page_index);
        }
    } else {
        is_valid = false;
    }

    if (!is_valid) {
        printf("Illegal memory access at 0x%x, process terminated\n", cr2);
        asm_halt();
        return;
    }

    // 合法地址，为其分配物理页
    int physicalPageAddress;
}

```



```

bool flag=true;
physicalPageAddress = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
if (physicalPageAddress == 0) {
    flag=false;
}

if (!flag) {
    if (memoryManager.physicalPageQueue.empty()) {
        asm_halt();
    }
    item u=memoryManager.physicalPageQueue.getFront();
    memoryManager.physicalPageQueue.pop();
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, u.physicalAddress, 1);
    int *pte=(int *)memoryManager.toPTE(u.virtualAddress);
    *pte=0;
    physicalPageAddress=memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
    flush_tlb_single(u.virtualAddress);
}
memoryManager.physicalPageQueue.push(item(cr2,physicalPageAddress));
flag=memoryManager.connectPhysicalVirtualPage(cr2, physicalPageAddress);
if (!flag) {
    printf("connect with physical page failed\n");
    asm_halt();
}

printf("Connect virtual Page:%d with physical Page:%d\n",memoryManager.virtualPageID(cr2),memoryManager.
    physicalPageID(physicalPageAddress));
}

```

## 4.4 时钟置换算法 (第二次机会算法)

时钟置换算法是对 FIFO 页面置换算法的改进，当选择了一个页面时先检查它的引用位，若值为 0 则直接替换此页面；若引用位为 1 则给此页面第二次机会并将引用位清 0，然后将指针指向下一个页面，继续检查下一个页面的引用位。根据页表的结构，页表的第 6 低位为引用位，与 FIFO 页面置换算法相比，时钟置换需要实现以下几个函数：

```

//访问引用位
bool vis(int vaddr);
//设置引用位为0
void setVis(int vaddr);
//寻找置换的页面
int FindExchangePage();

```

引用位的查找与设置如下：

```

bool MemoryManager::vis(int vaddr)
{
    int *pte = (int *)toPTE(vaddr);
    return ((*pte) & 0x00000020)>0?true:false;
}
void MemoryManager::setVis(int vaddr)
{
    int *pte = (int *)toPTE(vaddr);
    *pte = *pte & 0xfffffddf;
}

```

寻找置换的页面时，需要在队列中维护一个 cur 指针，指向准备访问的位置，并按照时钟置换算法的规则进行查找：

```

int MemoryManager::FindExchangePage()
{
    int &cur=physicalPageQueue.cur;
    Queue &q=physicalPageQueue;
    while(1)
    {
        item u=q.queue[cur];

```

```

        if (!vis(u.virtualAddress)) return (int)cur;
        setVis(u.virtualAddress);
        cur=(cur+1)%MAX_QUEUE_SIZE;
        if (cur==q.rear) cur=q.front;
    }
}

```

在中断处理函数中，与 FIFO 的不同点在于，一是寻找替换页面的方式，用 FindExchangePage 函数来寻找，二是新分配的页面直接取代原页面在队列中的位置，而不是插入到队尾，同时 cur 指针指向新加入的元素的下位，从而实现“FIFO”的目的。

```

extern "C" void c_page_interrupt_handler()
{
    uint32 cr2=read_cr2();
    printf("page fault on virtual address :0x%x\n",cr2);
    bool is_valid = false;

    // 计算地址所在的页面
    uint32 page_addr = cr2 & 0xFFFFF000;

    // 检查是否是内核地址空间 (3GB以上)
    if (cr2 >= 0xC0000000) {
        // 内核地址空间，检查该地址是否在内核虚拟地址池的已分配范围内
        int page_index = (page_addr - memoryManager.kernelVirtual.startAddress) / 4096;
        if (page_index >= 0 && page_index < memoryManager.kernelVirtual.resources.length) {
            // 检查该页是否已分配 (即位图中该位是否为1)
            is_valid = memoryManager.kernelVirtual.resources.get(page_index);
        }
    } else {
        is_valid = false;
    }

    if (!is_valid) {
        printf("Illegal memory access at 0x%x, process terminated\n", cr2);
        asm_halt();
        return;
    }

    // 合法地址，为其分配物理页
    int physicalPageAddress;
    bool flag=true;
    physicalPageAddress = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
    if (physicalPageAddress == 0) {
        flag=false;
    }

    if (!flag) {
        if (memoryManager.physicalPageQueue.empty()) {
            asm_halt();
        }
        int index=memoryManager.FindExchangePage();
        item u=memoryManager.physicalPageQueue.queue[index];
        //删除该物理页框和清空页表
        memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, u.physicalAddress, 1);
        int *pte=(int *)memoryManager.toPTE(u.virtualAddress);
        *pte=0;
        physicalPageAddress=memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
        item v(cr2,physicalPageAddress);
        memoryManager.physicalPageQueue.queue[index]=v;
        memoryManager.physicalPageQueue.cur=(index+1)%MAX_QUEUE_SIZE;
        if (memoryManager.physicalPageQueue.cur==memoryManager.physicalPageQueue.rear) memoryManager.physicalPageQueue.
            cur=memoryManager.physicalPageQueue.front;
        flush_tlb_single(u.virtualAddress);
    } else { //物理地址没分配满

```

```

        memoryManager.physicalPageQueue.push(item(cr2,physicalPageAddress));
    }
    flag=memoryManager.connectPhysicalVirtualPage(cr2, physicalPageAddress);
    if (!flag) {
        printf("connect with physical page failed\n");
        asm_halt();
    }
    printf("Connect virtual Page:%d with physical Page:%d\n",memoryManager.virtualPageID(cr2),memoryManager.
        physicalPageID(physicalPageAddress));
    printf("curpos:%d\n",memoryManager.physicalPageQueue.getcurpos());
}

```

## 4.5 LRU 页面置换

LRU 页面置换中，对每个页面维护一个 LRU 位，每访问一次页面则将该页面的 LRU 位清零，其余页面的 LRU 位加 1。意味着 LRU 位越大，该页面则是越久未被访问，页面置换时选择 LRU 位最大的页面进行置换。

队列中元素的定义如下，增添了 LRU 位

```

struct item {
    item(int virtualAddress,int physicalAddress,int lru=0)
    {
        this->virtualAddress=virtualAddress;
        this->physicalAddress=physicalAddress;
        this->lru=lru;
    }
    item(){}
    int virtualAddress;
    int physicalAddress;
    int lru;
};

```

选择替换页面的函数

```

int MemoryManager::FindExchangePage()
{
    int index=-1;
    int mx=0;
    for (int i=physicalPageQueue.front;i!=physicalPageQueue.rear;i=(i+1)%MAX_QUEUE_SIZE)
    {
        if (physicalPageQueue.queue[i].lru>mx)
        {
            mx=physicalPageQueue.queue[i].lru;
            index=i;
        }
    }
    return index;
}

```

在队列的处理中，增加删除元素的方法

```

int del(int index)
{
    if (empty()) return -1;
    for (int i=index,next=(i+1)%MAX_QUEUE_SIZE;next!=rear;i=next,next=(i+1)%MAX_QUEUE_SIZE)
    {
        queue[i]=queue[next];
    }
    rear=(rear-1+MAX_QUEUE_SIZE)%MAX_QUEUE_SIZE;
    return 0;
}

```

在中断处理函数中，换走的页面要从队列中删除，新增的页面要插入到队尾

```

if (!flag) {
    if (memoryManager.physicalPageQueue.empty()) {

```

```

asm_halt();
}
int index=memoryManager.FindExchangePage();
item u=memoryManager.physicalPageQueue.queue[index];
memoryManager.physicalPageQueue.del(index);
memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, u.physicalAddress, 1);
int *pte=(int *)memoryManager.toPTE(u.virtualAddress);
*pte=0;
physicalPageAddress=memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
item v(cr2,physicalPageAddress);
memoryManager.physicalPageQueue.push(v);
flush_tlb_single(u.virtualAddress);
} else { //物理地址没分配满
memoryManager.physicalPageQueue.push(item(cr2,physicalPageAddress));
}
flag=memoryManager.connectPhysicalVirtualPage(cr2, physicalPageAddress);

```

每进行一次元素访问，就要维护 LRU 位

```

int count=0;
void visit(char *p,int pagenum) //访问第几页
{
count++;
printf("-----%d-----\n",count);
p[pagenum*PAGE_SIZE]='a';
for (int i=memoryManager.physicalPageQueue.front;i!=memoryManager.physicalPageQueue.rear;i=(i+1)%MAX_QUEUE_SIZE)
{
memoryManager.physicalPageQueue.queue[i].lru++;
item &u=memoryManager.physicalPageQueue.queue[i];
if (u.virtualAddress==(int)p+pagenum*PAGE_SIZE)
{
u.lru=0;
}
}
}
}

```

## 4.6 实验结果展示

### 4.6.1 FIFO 页面置换

为了方便测试算法的正确性，以及展示出物理页帧分配满的情况，在初始化内存管理时，只分配10个物理页

```
int kernelphysicalpage=10;
```

构造的测试样例如下：

首先分配 15 个虚拟页

```
char *p=(char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 15);
```

接着依次访问这十五个虚拟页对应的地址，会发生 15 次缺页中断，且第 11 次开始会发生页面置换

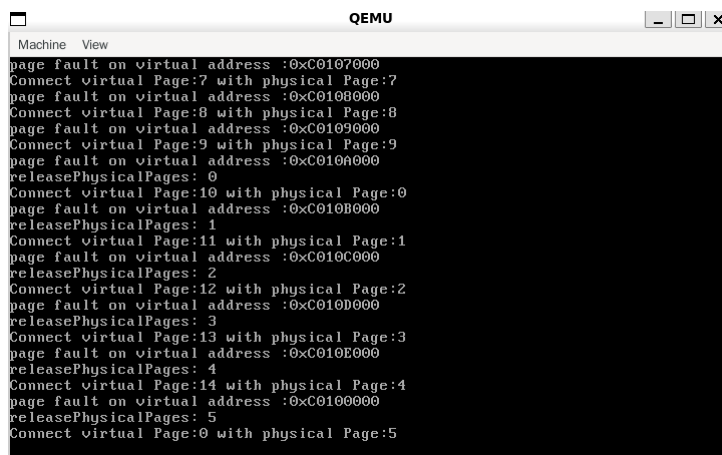
```
for (int i=0;i<15;i++)
{
p[i*PAGE_SIZE]='a';
}

```

最后再访问一次 p 指向的地址，由于此地址对应的物理页已经发生置换，因此这个地址访问也会产生缺页中断。

```
*p='a';
```

输出如下：



顺利完成了请求调页 → 缺页中断处理 → FIFO 页面置换

## 4.6.2 时钟页面置换

在 setup.cpp 中，为了保证在访问后一定会设置访问位（有时 MMU 有 bug?），于是便封装了访问函数：

```
void visit(char *p,int pagenum) //访问第几页
{
    p[pagenum*PAGE_SIZE]='a';
    // 页面被访问后，需要设置访问位
    int vaddr = (int)(p + pagenum*PAGE_SIZE);
    int *pte = (int *)memoryManager.toPTE(vaddr);
    *pte = *pte | 0x00000020; // 设置访问位为1
}
```

为了方便测试，初始化物理页帧即驻留集为5。

设置的样例如下：

```
char *p=(char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 8);
visit(p,1);
visit(p,3);
visit(p,4);
visit(p,2);
visit(p,5);
visit(p,6);
visit(p,3);
visit(p,4);
visit(p,7);
```

以下为此时钟置换算法的执行过程，其中红色代表当前指针位置  
前五次访问后，内存中信息如下：

物理页号	0	1	2	3	4
虚拟页号 (括号内为访问位)	1(1)	3(1)	4(1)	2(1)	5(1)

第六次访问后，内存中信息如下：

物理页号	0	1	2	3	4
虚拟页号 (括号内为访问位)	6(1)	3(0)	4(0)	2(0)	5(0)

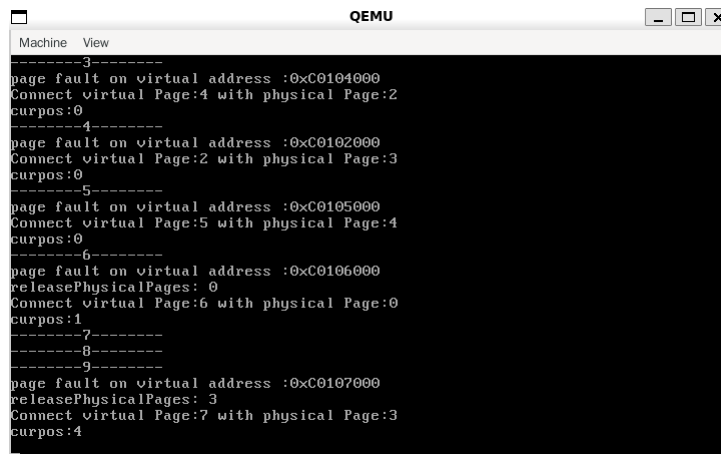
第七、八次访问后，内存中信息如下：

物理页号	0	1	2	3	4
虚拟页号 (括号内为访问位)	6(1)	3(1)	4(1)	2(0)	5(0)

最后一次访问后，内存中信息如下：

物理页号	0	1	2	3	4
虚拟页号 (括号内为访问位)	6(1)	3(0)	4(0)	7(1)	5(0)

运行结果如下：



## 4.7 LRU 页面置换

为了方便测试，初始化物理页帧即驻留集为4。

设置的样例如下：

```
char *p=(char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 9);
visit(p,1);
visit(p,8);
visit(p,1);
visit(p,7);
visit(p,8);
visit(p,2);
visit(p,7);
visit(p,2);
visit(p,1);
visit(p,8);
visit(p,3);
visit(p,8);
visit(p,2);
visit(p,1);
visit(p,3);
visit(p,1);
visit(p,7);
visit(p,1);
visit(p,3);
visit(p,7);
```

根据 LRU 算法，执行过程如下：

访问页面	1	8	1	7	8	2	7	2	1	8	3	8	2	1	3	1	7	1	3	7
页框 0	1	1		1		1					1						1			
页框 1		8		8		8					8						7			
页框 2				7		7					3						3			
页框 3						2					2						2			
是否发生缺页	√	√		√		√					√						√			

实验结果如下：

```
Machine View
-----1-----
page fault on virtual address :0xC0101000
Connect virtual Page:1 with physical Page:0
-----2-----
page fault on virtual address :0xC0100000
Connect virtual Page:8 with physical Page:1
-----3-----
-----4-----
page fault on virtual address :0xC0107000
Connect virtual Page:7 with physical Page:2
-----5-----
-----6-----
page fault on virtual address :0xC0102000
Connect virtual Page:2 with physical Page:3
-----7-----
-----8-----
-----9-----
-----10-----
-----11-----
page fault on virtual address :0xC0103000
releasePhysicalPages: 2
Connect virtual Page:3 with physical Page:2
-----12-----
-----13-----

-----12-----
-----13-----
-----14-----
-----15-----
-----16-----
-----17-----
page fault on virtual address :0xC0107000
releasePhysicalPages: 1
Connect virtual Page:7 with physical Page:1
-----18-----
-----19-----
-----20-----
```

实验结果符合预期

## 5 实验任务四

### 5.1 虚拟页内存分配的三步过程

第一步: 从虚拟地址池中分配虚拟页

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

第二步: 为每一个虚拟页分配物理页, 其中一个一个的分配物理页可以避免外部碎片的存在, 这也是分页机制的一大优点:

```
for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE)
{
    flag = false;
    physicalPageAddress = allocatePhysicalPages(type, 1);
```

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。

```
for (int i = 0; i < count; ++i, vaddress += PAGE_SIZE)
{
    flag = false;
    physicalPageAddress = allocatePhysicalPages(type, 1);
    if (physicalPageAddress)
    {
        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;
    }
}

```

其中建立页目录项和页表项的过程，涉及由地址的虚拟地址到页目录表和页表的虚拟地址的转换：

```

int MemoryManager::toPDE(const int virtualAddress)
{
    return (0xffff000 + (((virtualAddress & 0xffc00000) >> 22) * 4));
}

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

bool MemoryManager::connectPhysicalVirtualPage(const int virtualAddress, const int physicalPageAddress)
{
    int *pde = (int *)toPDE(virtualAddress);
    int *pte = (int *)toPTE(virtualAddress);
    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;
        char *pagePtr = (char *)(((int)pte) & 0xffff000);
        memset(pagePtr, 0, PAGE_SIZE);
    }

    // 使页表项指向物理页
    *pte = physicalPageAddress | 0x7;

    return true;
}

```

## 5.2 虚拟页内存释放

虚拟页的内存释放，需要先释放物理页，然后设置页表项为不存在，最后释放虚拟页

```

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);
}

```



### 5.3 虚拟页管理实现存在的 bug1: 未实现互斥

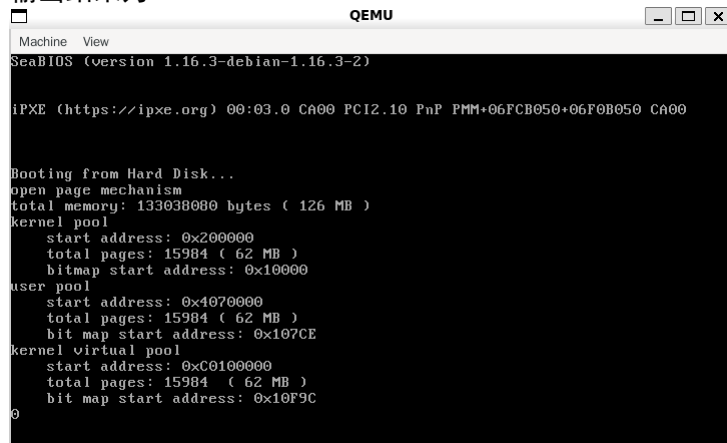
各个线程进行页面分配时，并不是互斥进行的，若在页面分配的过程中发生调度，可能会导致不同线程分配到了同一物理页，导致错误发生。

### 5.4 bug2:769-1022 个页目录项对应的页表未初始化

我们初始一共划分了 15984 个虚拟页，而当我们尝试分配 15984 个虚拟页时：

```
char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 15984);
printf("%x\n", p1);
```

输出结果为



结果说明分配失败了。去探查原因时，发现在建立虚拟页和物理页对应的页表项和页目录项时，也给页表分配了物理页，这与先前的想法自相矛盾了。

```
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;
    }
}
```

但我们事先在初始化内存管理时，以及预留了 256 个页表的空间紧跟在内核空间和页目录表的后面，这些页表原本是用来关联 3G-4G 的内核虚拟空间的，事实上我们并未对这部分进行初始化，导致这些空间未被利用。

```
int usedMemory = 256 * PAGE_SIZE + 0x100000;
```

### 5.5 修复 bug1

在页面分配的函数中进行上锁，保证页面分配的互斥性。

```
int MemoryManager::allocatePages(enum AddressPoolType type, const int count)
{
    Lock.lock();
    // 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;
    }
}
Lock.unlock();
return virtualAddress;
}

```

## 5.6 修复 bug2

在开启分页机制前, 建立起第 768-1023 个页目录项对应的页表, 提前维护好 3G-4G 的内核虚拟空间的页表:

```

// 0~1MB
directory[0] = ((int)page) | 0x07;
// 3GB 的内核空间
directory[768] = directory[0];

for (int i=769; i<1023; i++)
{
    page=page+PAGE_SIZE;
    memset(page, 0, PAGE_SIZE);
    directory[i]=((int)page)|0x07;
}
// 最后一个页目录项指向页目录表
directory[1023] = ((int)directory) | 0x7;

```

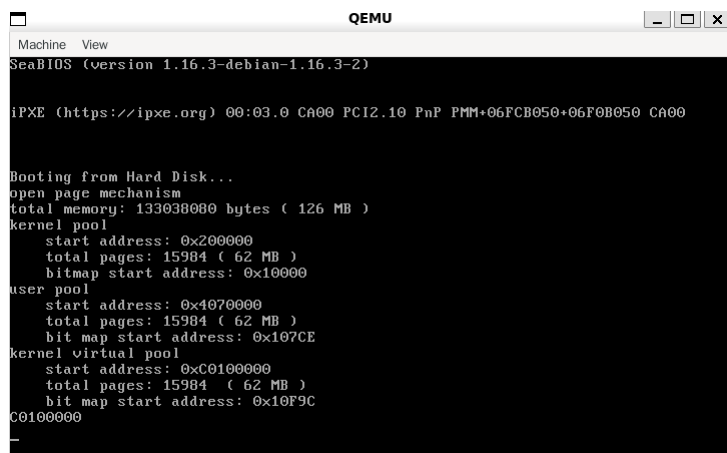
在 setup.cpp 中进行测试: 分配满 15984 个虚拟页:

```

char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 15984);
printf("%x\n", p1);

```

输出结果为



```
Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP FMM+06FCB050+06F0B050 CA00

Booting from Hard Disk...
open page mechanism
total memory: 133038000 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15904 ( 62 MB )
  bit map start address: 0x10000
user pool
  start address: 0x4070000
  total pages: 15904 ( 62 MB )
  bit map start address: 0x107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15904 ( 62 MB )
  bit map start address: 0x10F9C
C0100000
```

## 6 实验总结

### 6.1

本次实验综合了 Lab4 关于中断的处理，实现了请求调页与缺页中断的处理，但尚有以下不足

- 在动态内存分配中，只以页为单位进行动态内存分配，未直接对物理内存进行分配。
- 对于页表项访问位的维护，未采用 MMU 的方式，而是通过软件的方式来维护，与实际有所偏差。
- 由于磁盘等操作较为困难，本实验只实现了简单版的请求调页，仅仅将物理页置换，没有实现换出到外存以及在外存维护页信息的操作。

### 6.2

本实验为了更好展示测试结果，手动设置了驻留集大小为 10, 5, 4 等，实际中应根据系统的物理内存大小来设置驻留集大小。

### 6.3

开启了分页机制后，要时刻注意区分虚拟地址与物理地址，程序中涉及的地址都是虚拟地址。