



本科生实验报告

实验课程：____操作系统原理实验____

实验名称：____内存管理____

专业名称：____计算机科学与技术____

学生姓名：____数据删除____

学生学号：____数据删除____

实验成绩：____

报告时间：____2025 年 5 月 22 日____

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。

2. 实验过程

Assignment 1

程序分配物理页的过程为：

1. 调用 `MemoryManager::allocatePhysicalPages`，根据申请分配的内存类型（内核或用户）调用响应的 `AddressPool` 对象进行分配。

2. `AddressPool::allocate` 会调用内部 `bitmap` 的 `allocate` 函数,后者使用 `first-fit` 算法从位图中找到第一个满足申请大小的空闲连续块,返回第一个块的起始地址。若没有找到,则返回空。

虽然打开了二级分页,但尚未实现页表管理,因此尽管能分配物理页但无法使用,因为程序使用的是虚拟地址而不是物理地址,所以只能简单测试物理页的分配和释放。

测试思路为先分配若干个物理页,首先检测分配的内存区段是否有重叠,然后释放最前面的物理页后申请若干个小块,检测物理页是否能够复用。

Assignment 2

选择 `best-fit` 分配算法进行实现,当进程请求内存时,该算法会遍历所有空闲内存块,查找能够满足请求大小且大小最接近请求大小的空闲块。

实现同样采用位图 `Bitmap` 来管理空闲物理页。位图中的每一位对应一个物理页,0 表示空闲,1 表示已分配。内存分配器 `Allocator` 在初始化时首先获取最大可用内存量,分配一部分(1MB)作为内核保留内存,剩下的按照 `PAGE_SIZE` 进行分块。内存分配时,`Allocator` 遍历整个 `Bitmap`,提取出所有空闲段并逐一与请求的长度进行比较,选取满足 `best-fit` 要求的最优段并返回其起始地址的指针,若没有满足的空闲段,返回 `nullptr`。

测试时首先分配若干个块,检验算法连续分配能力,之后将第一个块释放并重分配,检验 `Allocator` 是否能复用释放后的内存,以及 `best-fit` 是否生效。

Assignment 3

选择 `LRU` 算法进行实现,该算法在发生缺页中断,需要调入页面而页表已满时,选择最近一段时间最久未被访问过的页面进行淘汰。

实现上,用一个数组 `recent` 表示页表中的一个页面从上次访问以来经过的时间,每次访问时,被访问到的页(命中,或缺页换入后)的 `recent` 值置 0,其余页 `recent` 值加 1。缺页时,`recent` 最大的页就是最近最久违背访问过的页,遍历页表将其替换。

基于该算法编写模拟程序，首先输入页表大小和询问次数，每次询问输入一个请求页号，调页函数返回两个值，分别表示是否命中、未命中时替换的页号，用于正确性验证。

Assignment 4

虚拟页内存分配分为以下三个步骤：

1. 从虚拟地址池中分配若干连续的虚拟页。

- a) 调用 `MemoryManager::allocateVirtualPages` 从指定类型（目前只实现了 `KERNEL`）的虚拟地址池中分配 `count` 个连续的虚拟页。
- b) 如果虚拟地址分配失败，返回空地址；否则返回分配的虚拟页的起始地址。

2. 对每一个虚拟页，从物理地址池中分配 1 页。

调用 `MemoryManager::allocatePhysicalPages(KERNEL, 1)` 分配一个物理页。这里不要求物理页连续，所以是按照一个一个页的顺序进行分配。

3. 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。

- 1. 调用 `MemoryManager::connectPhysicalVirtualPage` 函数，传入虚拟地址和物理地址，函数内先计算虚拟地址对应的页目录表 PDE 和页表 PTE 地址，若 PDE 不存在（P 位为 0），则从内核地址空间中分配一个新的页表。
- 2. 映射过程中可能会出错（内核内存不足，无法创建新的页表），此时需要回滚以防止已经分配的内存无法被再次使用。回滚分为两部分，首先使用 `MemoryManager::releasePages` 释放已经成功映射的前 `i` 个页面（包括虚拟页和物理页），对于剩下的 `count-i` 个已分配虚拟页但未映射到物理页的页面，则调用 `MemoryManager::releaseVirtualPages` 来释放。

虚拟页内存的释放步骤为：

3. 关键代码

Assignment 1

bitmap 的 allocate 函数:

```
int BitMap::allocate(int count) {
    if (count == 0) return -1;

    int index, empty, start;

    index = 0;
    while (index < length) {
        // 越过已经分配的资源
        while (index < length && get(index))
            ++index;

        // 不存在连续的 count 个资源
        if (index == length)
            return -1;

        // 找到 1 个未分配的资源
        // 检查是否存在从 index 开始的连续 count 个资源
        empty = 0;
        start = index;
        while ((index < length) && (!get(index)) && (empty < count)) {
            ++empty;
            ++index;
        }

        // 存在连续的 count 个资源
        if (empty == count) {
            for (int i = 0; i < count; ++i) {
                set(start + i, true);
            }

            return start;
        }
    }

    return -1;
}
```

测试代码:

```

void first_thread(void*) {
    auto page1 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
    auto page2 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
    printf("page1: %x\npage2: %x\n", page1, page2);
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, page1, 1);
    auto page3 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 1);
    printf("page3: %x\n", page3);
    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, page3, 1);
    auto page4 = memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 2);
    printf("page4: %x\n", page4);

    auto page5 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);
    auto page6 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);
    printf("page5: %x\npage6: %x\n", page5, page6);
    memoryManager.releasePhysicalPages(AddressPoolType::USER, page5, 1);
    auto page7 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);
    printf("page7: %x\n", page7);
    memoryManager.releasePhysicalPages(AddressPoolType::USER, page7, 1);
    auto page8 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 2);
    printf("page8: %x\n", page8);

    asm_halt();
}

```

Assignment 2

内存分配器：

```

class Allocator {
public:
    int pageCount;
    int PhysicalStartAddress;
    uint8* bitmap;

public:
    Allocator() = default;
    void initialize() {
        int memory = *((int*)MEMORY_SIZE_ADDRESS);
        int low = memory & 0xffff;
        int high = (memory >> 16) & 0xffff;
        int totalMemory = low * 1024 + high * 64 * 1024;
        int usedMemory = 256 * PAGE_SIZE + 0x100000;
        int freeMemory = totalMemory - usedMemory;

        pageCount = freeMemory / PAGE_SIZE;

        PhysicalStartAddress = usedMemory;
    }
}

```

```

    bitmap = (uint8*)BITMAP_START_ADDRESS;

    printf("totalMemory: %d MB\n", totalMemory / 1024 / 1024);
    printf("freeMemory: %d MB\n", freeMemory / 1024 / 1024);
    printf("pageCount: %d\n", pageCount);
    printf("PhysicalStartAddress: %x\n", PhysicalStartAddress);
    printf("bitmapAddress: %x\n", bitmap);

    for (int i = 0; i < pageCount; i++)
        bitmap[i] = 0;
}

void* allocate(int count) const {
    if (count <= 0) return nullptr;
    // best-fit
    int best_index = -1, best_count = 0;
    for (int i = 0; i < pageCount; i++) {
        int empty = 0;
        while (i < pageCount && bitmap[i] == 1) {
            i++;
        }
        while (i < pageCount && bitmap[i] == 0) {
            empty++;
            i++;
        }
        if (empty < count) continue;
        if (best_index == -1 || empty < best_count) {
            best_index = i - empty;
            best_count = empty;
        }
    }
    if (best_index == -1) return nullptr;
    for (int i = best_index; i < best_index + count; i++) {
        bitmap[i] = 1;
    }
    return (void*)(PhysicalStartAddress + best_index * PAGE_SIZE);
}

void deallocate(const void* address, int count) const {
    if (count <= 0) return;
    int index = ((int)address - PhysicalStartAddress) / PAGE_SIZE;
    for (int i = index; i < index + count; i++) {
        bitmap[i] = 0;
    }
}

} allocator;

```

测试代码：

```

void first_thread(void*) {
    printf("first thread\n");
    int* p1 = (int*)allocator.allocate(1000);
    int* p2 = (int*)allocator.allocate(1000);
    int* p3 = (int*)allocator.allocate(1000);
    printf("p1: %x\n", p1);
    printf("p2: %x\n", p2);
    printf("p3: %x\n", p3);
    allocator.deallocate(p2, 1000);
    int* p4 = (int*)allocator.allocate(200);
    printf("p4: %x\n", p4);
    allocator.deallocate(p1, 1000);
    int* p5 = (int*)allocator.allocate(500);
    printf("p5: %x\n", p5);
    int* p6 = (int*)allocator.allocate(1500);
    printf("p6: %x\n", p6);
    asm_halt();
}

```

Assignment 3

模拟代码：

```

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class LRUCache {
    int capacity;           // 页表大小
    vector<int> pageTable;  // 存储页号，-1 表示空
    vector<int> recent;     // 记录每个页的最近访问值

public:
    explicit LRUCache(int capacity)
        : capacity(capacity), pageTable(capacity, -1), recent(capacity, 0) {}

    // 请求页面，返回是否命中和被替换的页（-1 表示无替换）
    pair<bool, int> request(int page) {
        for (auto& i : recent) i++;
        int max_recent = -1, index = -1;
        for (int i = 0; i < capacity; i++) {
            if (pageTable[i] == page) {
                recent[i] = 0; // 命中，重置最近访问值
                return { true, -1 };
            }
            if (recent[i] > max_recent) max_recent = recent[i], index = i;
        }
    }
}

```



```

    }
    int replaced = std::exchange(pageTable[index], page); // 替换页
    recent[index] = 0; // 重置最近访问值
    return { false, replaced };
}

// 打印当前页表
void printTable() {
    cout << "Page Table: ";
    for (int i = 0; i < capacity; ++i) {
        cout << "Page " << i << " = " << pageTable[i] << " Recent= " << recent[i]
<< "\n";
    }
    cout << endl;
}

};

int main() {
    int lru_size, query_count;
    cin >> lru_size >> query_count;
    LRUCache lru(lru_size);
    for (int i = 0; i < query_count; i++) {
        int page;
        cin >> page;
        auto&& [hit, replaced] = lru.request(page);
        if (hit) {
            cout << "Page " << page << " hit\n";
        }
        else {
            cout << "Page " << page << " miss, replaced page " << replaced << "\n";
        }
    }
    lru.printTable();
    return 0;
}

```

Assignment 4

测试代码 1：在 Assignment1 的基础上加入了读写操作，检查页表是否映射成功。

```

auto page1 = memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
int* ptr = (int*)page1;
*ptr = 0x12345678;
printf("ptr: %x\n", *ptr);
auto page2 = memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
printf("page1: %x\npage2: %x\n", page1, page2);
memoryManager.releasePages(AddressPoolType::KERNEL, page1, 1);

```

```
auto page3 = memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
printf("page3: %x\n", page3);
memoryManager.releasePages(AddressPoolType::KERNEL, page3, 1);
auto page4 = memoryManager.allocatePages(AddressPoolType::KERNEL, 2);
printf("page4: %x\n", page4);
```

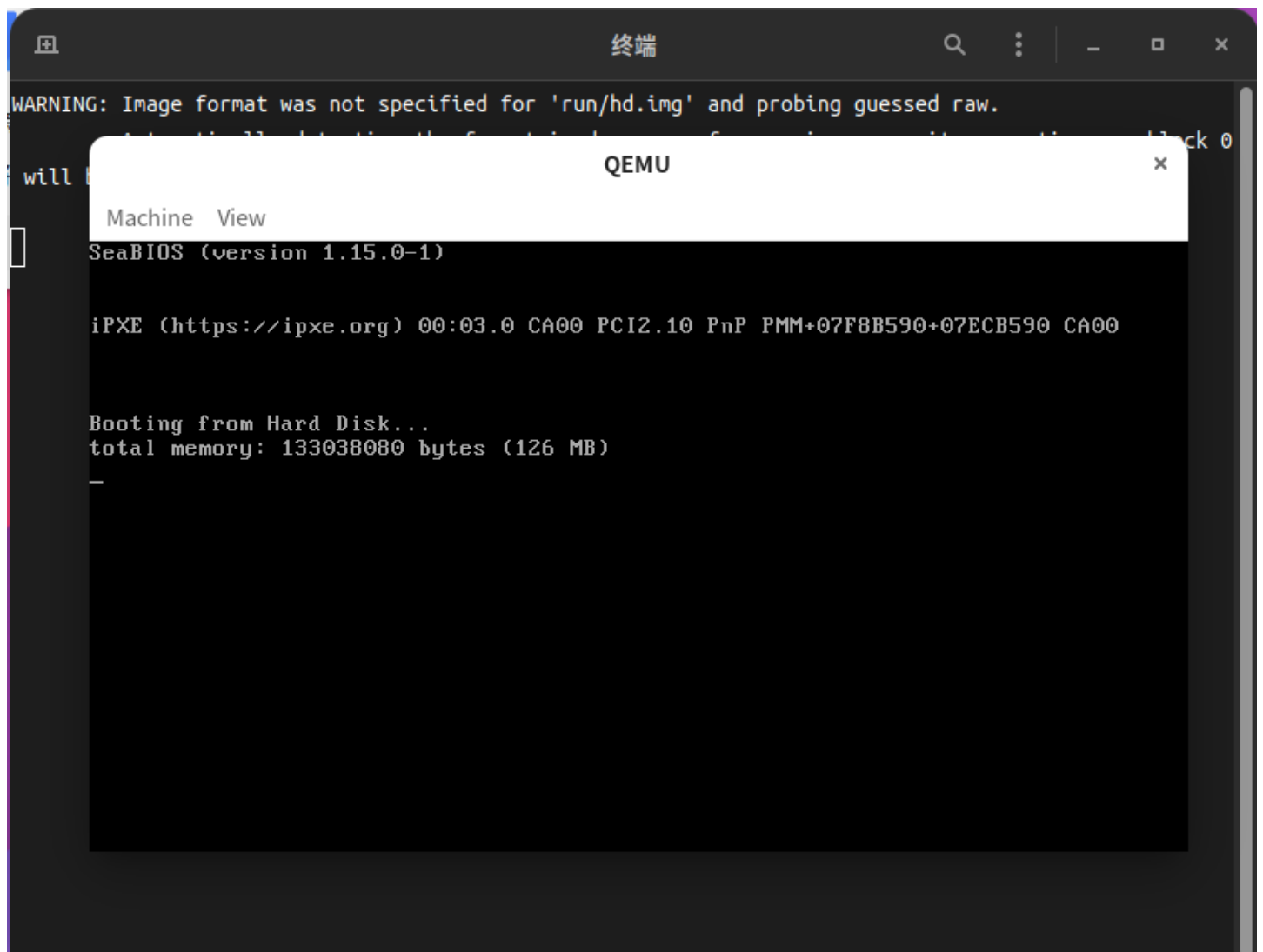
测试代码 2：大量分配和释放，检查虚拟页、物理页释放后能否再次使用：

```
while (true) {
    void* pages[100];
    for (int i = 1; i <= 100; i++) {
        pages[i - 1] = memoryManager.allocatePages(AddressPoolType::KERNEL, i * 2);
        printf("page%d: %x\n", i, pages[i - 1]);
    }
    for (int i = 1; i <= 100; i++) {
        memoryManager.releasePages(AddressPoolType::KERNEL, pages[i - 1], i * 2);
    }
}
```

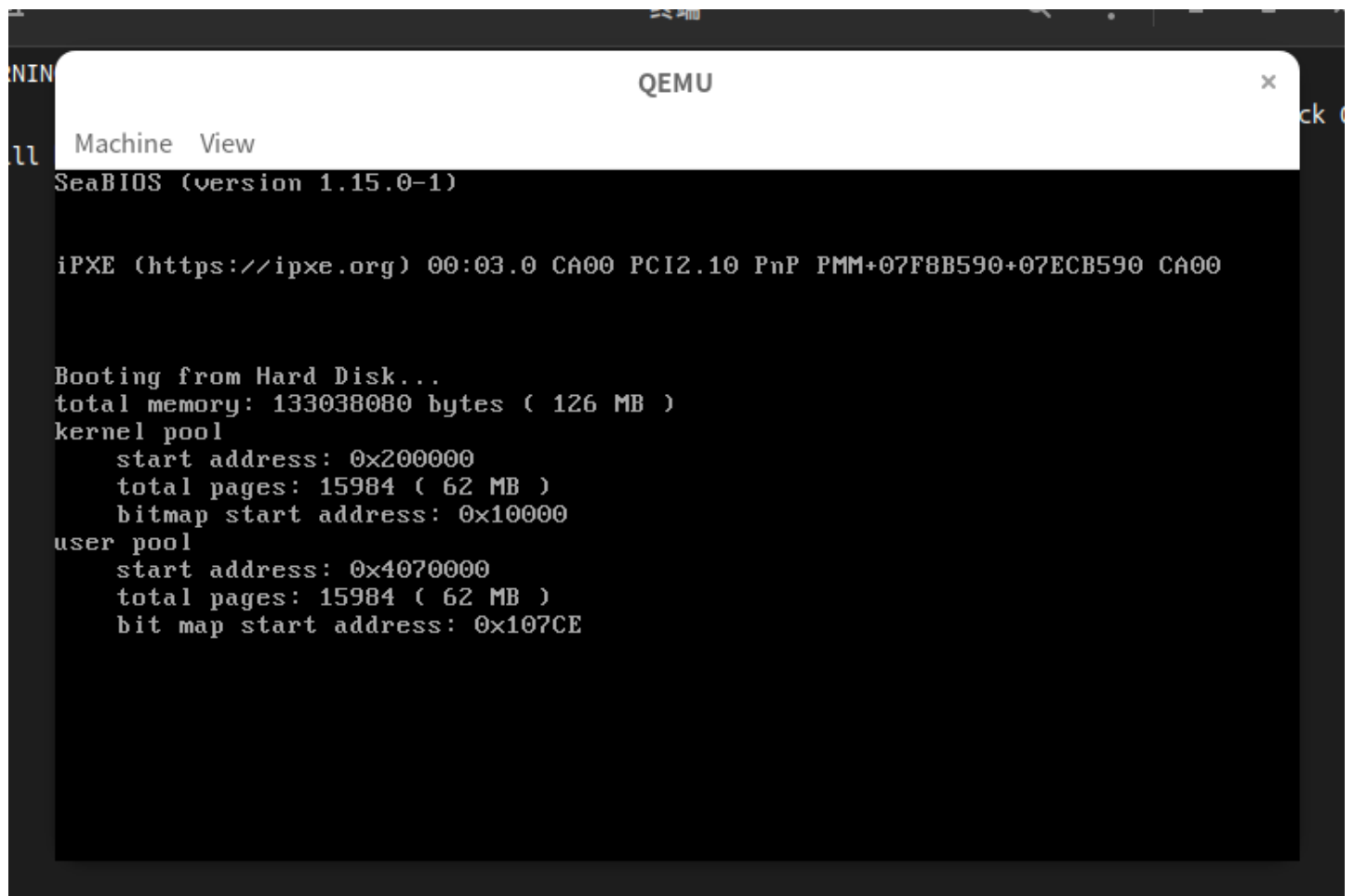
4. 实验结果

Assignment 1

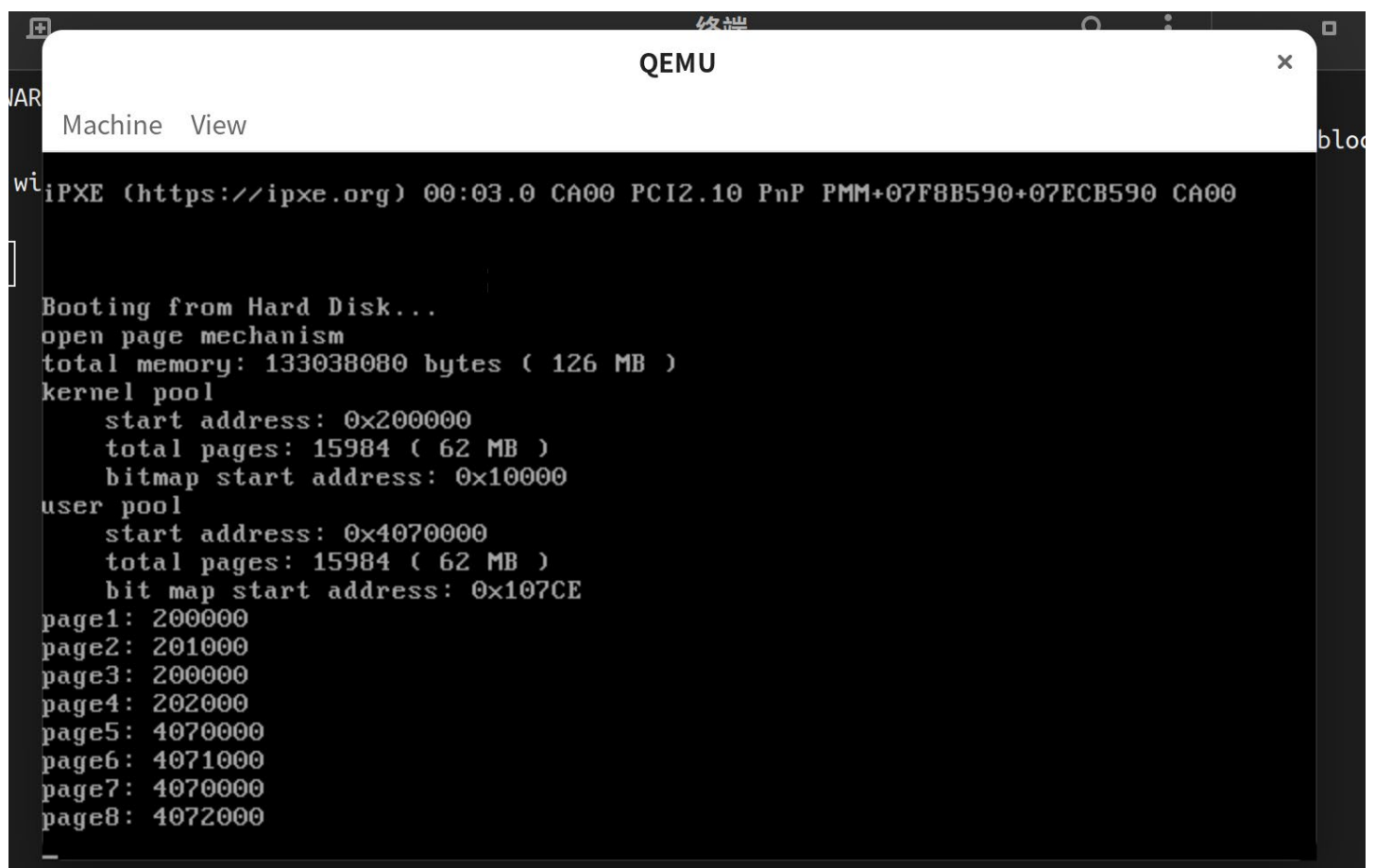
获取最大可用内存：



开启二级分页：



内存分配和释放测试：

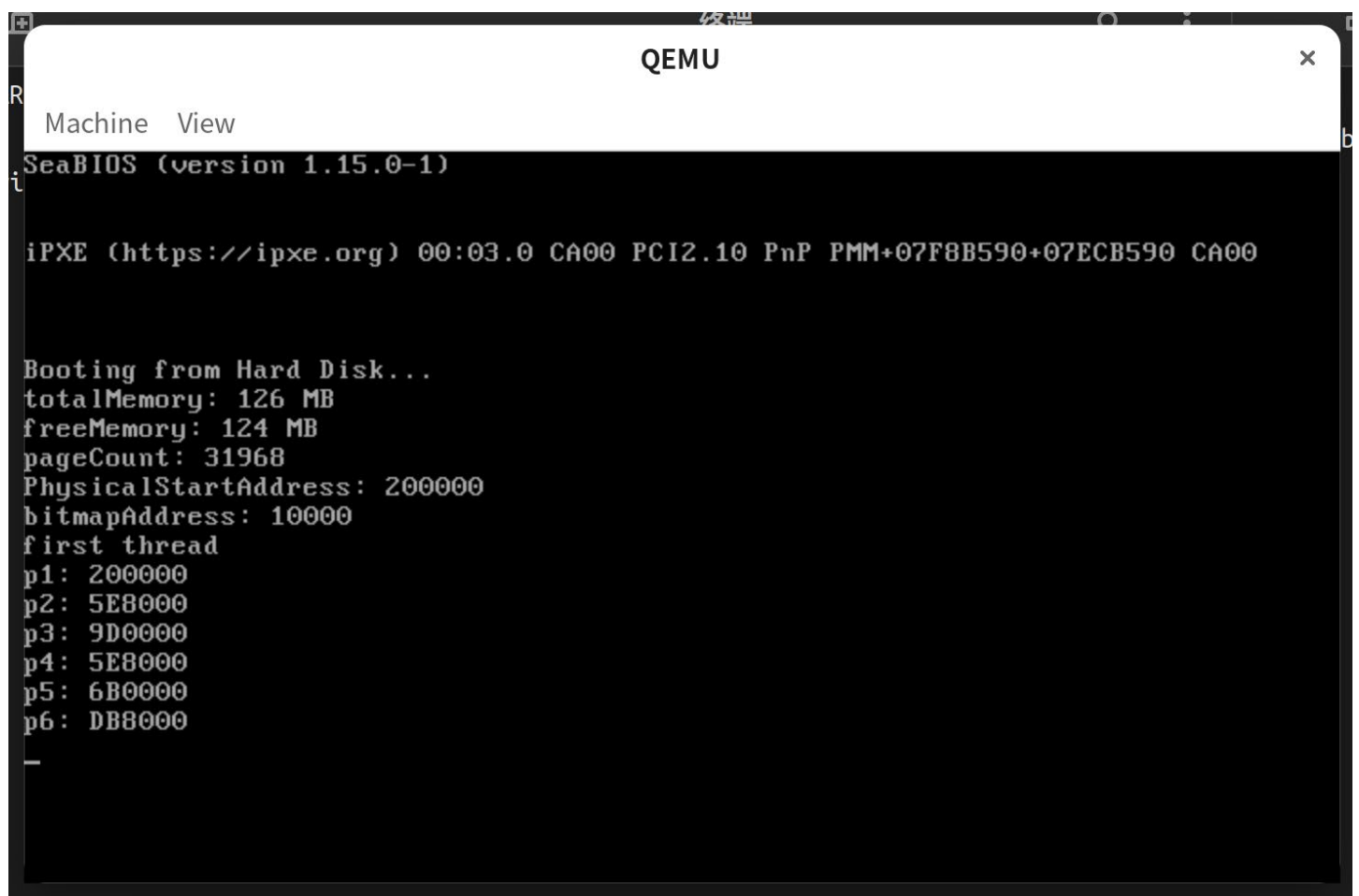


前四个 page 位于内核区，后四个位于用户区，首先可以确定程序正确区分了不同区域的内存分配。

首先按顺序分配了两个页 page1 和 page2，page1 大小为 1 (1*4096)，两个 page 的起始地址恰好也为 4096，符合预期；然后将 page1 释放，申请 page3，page3 得到和 page1 一样的物理地址，说明释放的物理块能够被复用，且 first-fit 算法起效；将 page3 释放，请求大小为 2 的页给 page4，page4 得到 202000，位于 page2 后，说明算法发现 page3 释放后的空洞不足以容纳 page4，同样符合预期。

后四个 page 同理，正确性显然。

Assignment 2



Assignment 3

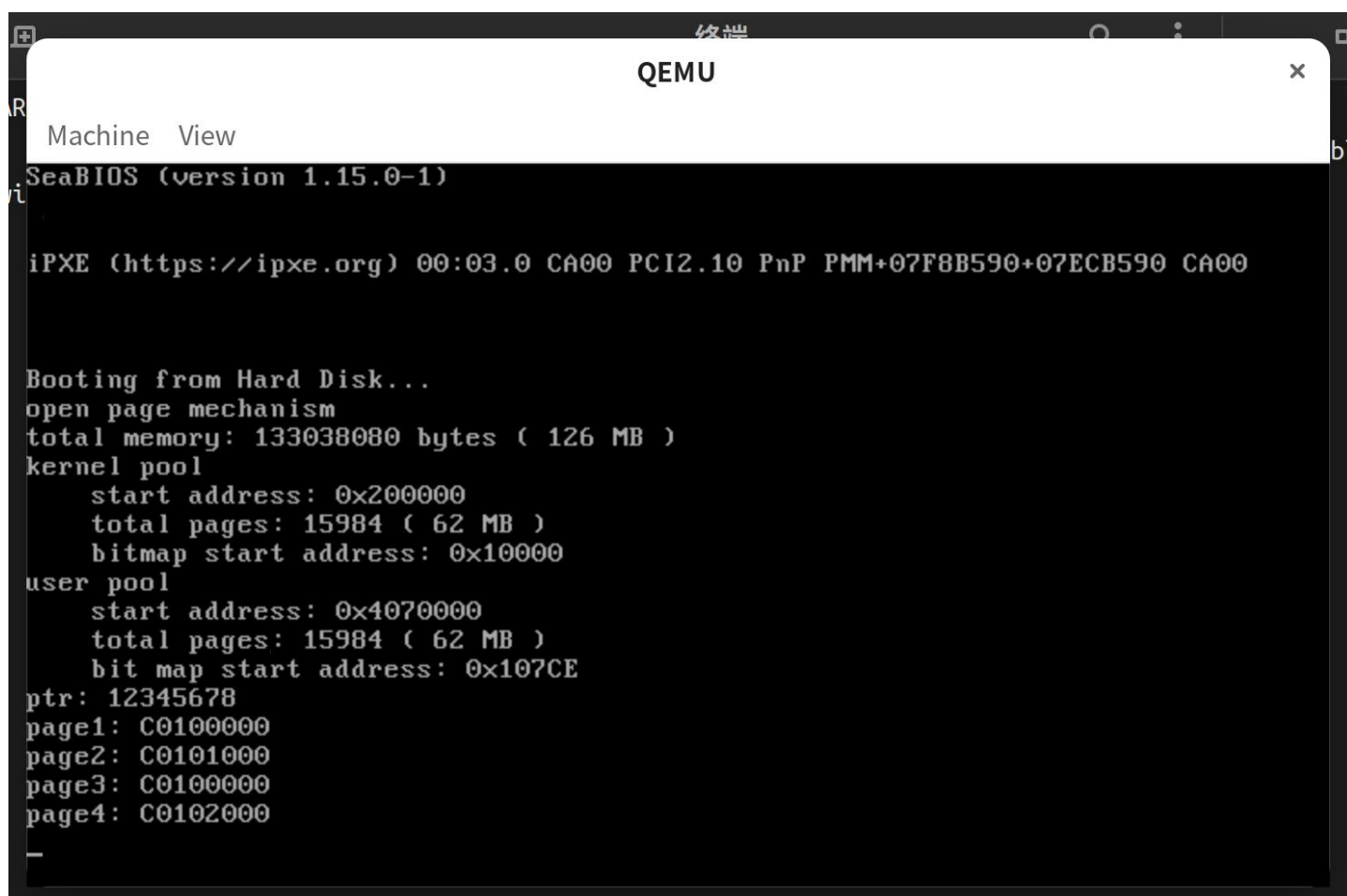
模拟输入数据和输出如下：

```
• bakabaka@Ubuntu22:~/lab$ xmake run lab7-3
3 10
1 3 5 2 4 2 4 3 1 5
Page 1 miss, replaced page -1
Page 3 miss, replaced page -1
Page 5 miss, replaced page -1
Page 2 miss, replaced page 1
Page 4 miss, replaced page 3
Page 2 hit
Page 4 hit
Page 3 miss, replaced page 5
Page 1 miss, replaced page 2
Page 5 miss, replaced page 4
Page Table:
Page 0 = 1 Recent= 1
Page 1 = 5 Recent= 0
Page 2 = 3 Recent= 2
```

与手动模拟的结果一致。

Assignment 4

测试 1:



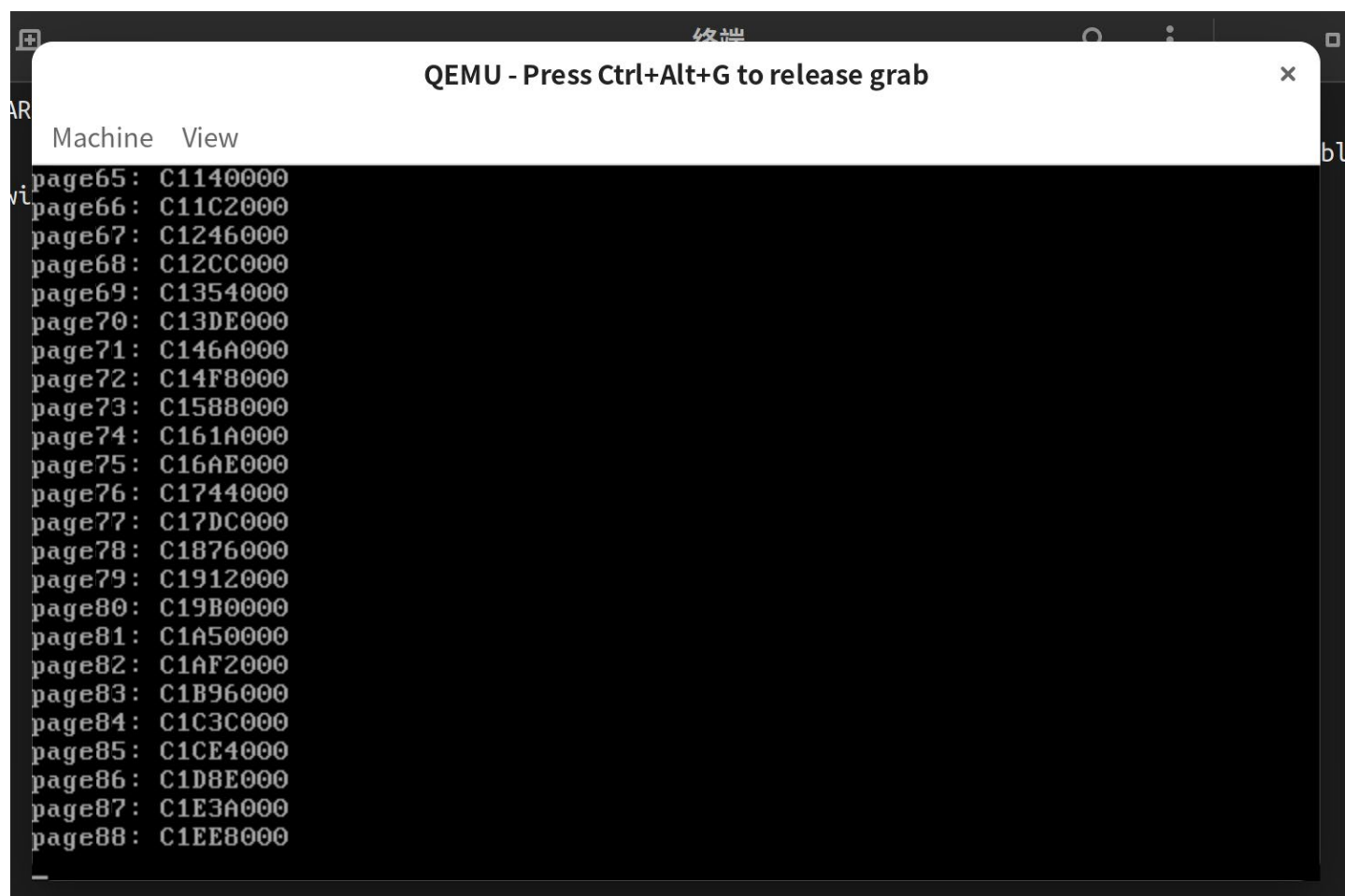
```
QEMU
Machine View
SeaBIOS (version 1.15.0-1)

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

Booting from Hard Disk...
open page mechanism
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0x10000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x107CE
ptr: 12345678
page1: C0100000
page2: C0101000
page3: C0100000
page4: C0102000
-
```

ptr 成功输出 12345678，且页面地址以 C0 开头，说明成功提升到了 3GB 的虚拟地址且页表工作正常。

测试 2:



反复申请和释放后相同 page 的虚拟页地址不变且能稳定运行，说明虚拟页和物理页可以复用。

5. 总结

通过本次实验，我对内存管理的原理和实现有了更深入的理解，包括二级分页、动态分区、页面置换算法和虚拟页管理多个模块，在理论课上学习到的内容在实现时有很多细节需要考虑，因此设计测试样例也是一个不小的挑战。