



本科生实验报告

实验课程：_____操作系统_____

实验名称：_____内存管理_____

专业名称：_____计算机科学与技术_____

学生姓名：_____凌国明_____

学生学号：_____21307077_____

实验地点：_____教室_____

实验成绩：_____

报告时间：_____2023. 06. 08_____

1. 实验要求

- 学习如何使用位图和地址池来管理资源
- 实现在物理地址空间下的内存管理
- 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

实验任务

- 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。
- 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。
- 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。
- 复现“虚拟页内存管理”一节的代码，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

1) 二级分页机制

- 第一步，在 include 中编写 bitmap.h，在 src/utils 中实现 bitmap.cpp

```
public:
    // 被管理的资源个数, bitmap的总位数
    int length;
    // bitmap的起始地址
    char *bitmap;
public:
    // 初始化
    BitMap();
    // 设置BitMap, bitmap=起始地址, length=总位数(即被管理的资源个数)
    void initialize(char *bitmap, const int length);
    // 获取第index个资源的状态, true=allocated, false=free
    bool get(const int index) const;
    // 设置第index个资源的状态, true=allocated, false=free
    void set(const int index, const bool status);
    // 分配count个连续的资源, 若没有则返回-1, 否则返回分配的第1个资源单元序号
    int allocate(const int count);
    // 释放第index个资源开始的count个资源
    void release(const int index, const int count);
    // 返回Bitmap存储区域
    char *getBitmap();
    // 返回Bitmap的大小
    int size() const;
private:
    // 禁止Bitmap之间的赋值
    BitMap(const BitMap &) {}
    void operator=(const BitMap&) {}
};
```

- 初始化，申请 $\text{ceil}(\text{length}/8)$ 个 char 的空间，一个 char 一个字节有 8 位，可以表示 8 个资源的占用情况

```
void BitMap::initialize(char *bitmap, const int length)
{
    this->bitmap = bitmap;
    this->length = length;

    int bytes = ceil(length, 8);

    for (int i = 0; i < bytes; ++i)
    {
        bitmap[i] = 0;
    }
}
```

- get 单个资源的状态：先找到资源对应的 bitmap 字节，然后将该字节取出，通过“与运算”将该字节的第 offset 位取出即可

```
bool BitMap::get(const int index) const
{
    int pos = index / 8;
    int offset = index % 8;

    return (bitmap[pos] & (1 << offset));
}
```

- set 单个资源的状态，寻址跟 get 的逻辑一样，然后如果是 set0 则通过与运算，set1 可以通过或运算，比如 set0 时，offset 为 1，则将该字节和 10111111 进行与运算，则完成置 0

```
void BitMap::set(const int index, const bool status)
{
    int pos = index / 8;
    int offset = index % 8;

    // 清0
    bitmap[pos] = bitmap[pos] & ~(1 << offset);

    // 置1
    if (status)
    {
        bitmap[pos] = bitmap[pos] | (1 << offset);
    }
}
```

- allocate: 先越过已经分配的资源，直到找到一个未分配的资源，位置为 p；然后从这个位置 p 开始，计算一共有多少个连续的未分配内存，如果够 count 个，则分配成功；如果不够 count 个，只有 n 个 (n < count) 则继续从 p+n 的位置继续上面的逻辑；如果直到到达 length 的边界还没有分配成功，则分配失败

```
int BitMap::allocate(const 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
    }
```

- 第二步，在 include 中编写 address_pool.h，在 src/utils 中实现 address_pool.cpp

```
class AddressPool
{
public:
    BitMap resources;
    int startAddress;

public:
    AddressPool();
    // 初始化地址池
    void initialize(char *bitmap, const int length, const int startAddress);
    // 从地址池中分配count个连续页，成功则返回第一个页的地址，失败则返回-1
    int allocate(const int count);
    // 释放若干页的空间
    void release(const int address, const int amount);
};

#endif
```

- 相比 bitmap，增加了 start_address，其他的实现都差不多，初始化调用 bitmap 的初始化，分配和释放都是调用 bitmap 的分配和释放

```
// 设置地址池BitMap
void AddressPool::initialize(char *bitmap, const int length, const int startAddress)
{
    resources.initialize(bitmap, length);
    this->startAddress = startAddress;
}

// 从地址池中分配count个连续页
int AddressPool::allocate(const int count)
{
    int start = resources.allocate(count);
    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)
{
    resources.release((address - startAddress) / PAGE_SIZE, amount);
}
```

- 第三步，在 include 中编写 memory.h，在 src/kernel 中实现 memory.cpp

```
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;
```

- MemoryManager 初始化：先预留部分内存，然后将剩下的内存对半分给内核空间和用户空间，然后分别设置起始地址，初始化用户空间和内核空间的 address_pool。申请内存：分两个空间的情况来申请

```
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);
}
```

- 二级分页机制：页目录表，页表，页都是 4KB 的。一个页目录表有 $4KB/4B=1024$ 项，一个页表有 $4KB/4B=1024$ 项，则一个页目录表可以放 2^{20} 个页，每个页 4KB，刚好 4GB。这样算的话，一个数据的地址 32 位，前 10 位是页目录项，中间 10 位是页表项，最后 12 位是页内偏移

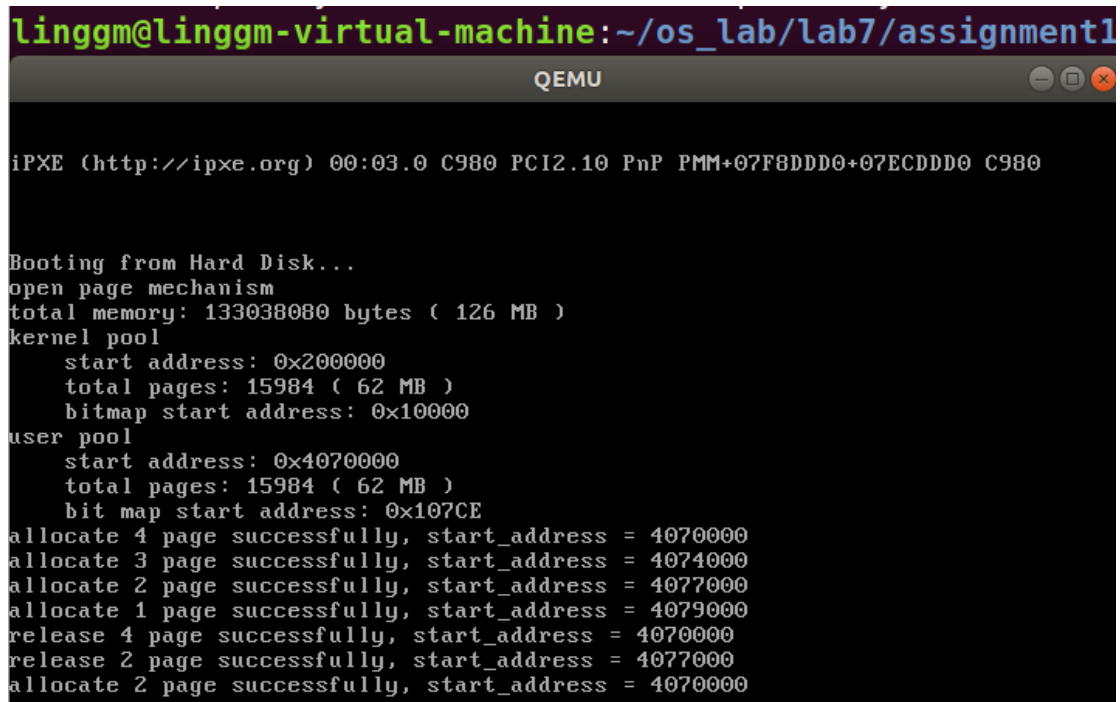
```
// 初始化页目录表
memset(directory, 0, PAGE_SIZE);
// 初始化线性地址0~4MB对应的页表
memset(page, 0, PAGE_SIZE);

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;
}
```

- 在第一个线程中申请用户空间的内存

```
void first_thread(void *arg)
{
    // 第1个线程不可以返回
    // stdio.moveCursor(0);
    // for (int i = 0; i < 25 * 80; ++i)
    // {
    //     stdio.print(' ');
    // }
    // stdio.moveCursor(0);
    int start_add_1 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 4);
    int start_add_2 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 3);
    int start_add_3 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 2);
    int start_add_4 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);
    memoryManager.releasePhysicalPages(AddressPoolType::USER, start_add_1, 4);
    memoryManager.releasePhysicalPages(AddressPoolType::USER, start_add_3, 2);
    int start_add_5 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 2);
    asm_halt();
}
```

- 运行结果



```
linggm@linggm-virtual-machine:~/os_lab/lab7/assignment1
QEMU

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD0+07ECDDDD0 C980

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
allocate 4 page successfully, start_address = 4070000
allocate 3 page successfully, start_address = 4074000
allocate 2 page successfully, start_address = 4077000
allocate 1 page successfully, start_address = 4079000
release 4 page successfully, start_address = 4070000
release 2 page successfully, start_address = 4077000
allocate 2 page successfully, start_address = 4070000
```

2) 任务二: 教程的 bitmap 中实现了 first-fit 算法, 我来实现 best-fit 算法

- 思想: 遍历所有空缺(空缺指连续的一段未分配的页), 在这些空缺中找出放得下"申请的页数量"的最小空缺, 完成分配。
- 具体实现: 用 smallest_k 记录遇到过的能放下申请页数的最小空缺, 初始为-1。先越过已经分配的资源, 直到找到一个未分配的资源, 位置为 p; 然后从这个位置 p 开始, 计算一共有 k 个连续的未分配内存; 如果 k 大于 count, 而且 k 是遇到过的 k 中最小的, 则将 smallest_k 置为 k; 继续从 p+k 的位置继续上面的逻辑, 直到到达 length 的边界; 如果 smallest_k == -1, 则分配失败, 否则成功

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

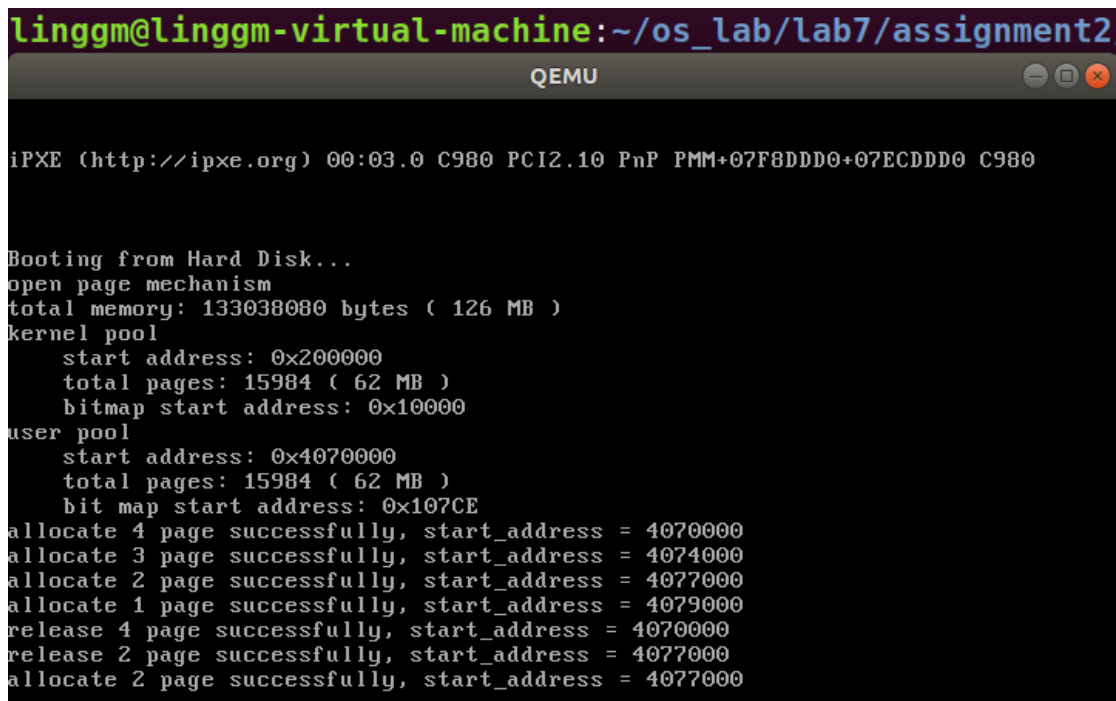
    // 不存在连续的count个资源
    if (index == length)
        break;

    // 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;
    // 直到搜索到头 或者 搜索到已分配资源
    while ((index < length) && !get(index))
    {
        ++empty;
        ++index;
    }

    // 存在大于等于count个的空位, 且空位数量小于已知最小数量
    if (empty >= count && empty < smallest_fit_empty)
    {
        smallest_fit_empty = empty;
        smallest_fit_start = start;
    }
}
if (smallest_fit_start != -1) {
    for (int i = 0; i < count; ++i)
    {
        set(smallest_fit_start + i, true);
    }
}
return smallest_fit_start;
}
```


- 先后申请 4、3、2、1 个页，然后释放掉第一次申请的 4 个页，释放第三次申请的 2 个页，再申请 2 个页，bitmap 中的 allocate 算法是 best-fit 算法，所以申请的 2 个页应该会在用户空间地址 0x4077000

```
void first_thread(void *arg)
{
    // 第1个线程不可以返回
    // stdio.moveCursor(0);
    // for (int i = 0; i < 25 * 80; ++i)
    // {
    //     stdio.print(' ');
    // }
    // stdio.moveCursor(0);
    int start_add_1 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 4);
    int start_add_2 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 3);
    int start_add_3 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 2);
    int start_add_4 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 1);
    memoryManager.releasePhysicalPages(AddressPoolType::USER, start_add_1, 4);
    memoryManager.releasePhysicalPages(AddressPoolType::USER, start_add_3, 2);
    int start_add_5 = memoryManager.allocatePhysicalPages(AddressPoolType::USER, 2);
    asm_halt();
}
```



The screenshot shows a QEMU terminal window with the following output:

```
linggm@linggm-virtual-machine:~/os_lab/lab7/assignment2
QEMU

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD0+07ECDDDD0 C980

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
allocate 4 page successfully, start_address = 4070000
allocate 3 page successfully, start_address = 4074000
allocate 2 page successfully, start_address = 4077000
allocate 1 page successfully, start_address = 4079000
release 4 page successfully, start_address = 4070000
release 2 page successfully, start_address = 4077000
allocate 2 page successfully, start_address = 4077000
```

申请的 2 个页在用户空间地址 0x4077000，符合预期

3) 任务三, FIFO 页面置换

- 在 MemoryManager 中加入两个队列

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

```
// 将页面加入到queue中  
void push(enum AddressPoolType type, int addr, int count);  
  
// 将页面从queue中pop调  
int pop(enum AddressPoolType type);
```

- 将从 addr 开始的 count 个页面加入 queue

```
// 将页面加入到queue中  
void MemoryManager::push(enum AddressPoolType type, int addr, int count){  
    printf("push: %x, count: %d\n", addr, count);  
    static ListItem *tmp;  
    tmp->addr = addr;  
    tmp->num = count;  
    if (type == AddressPoolType::KERNEL)  
        kernel_queue.push_back(tmp);  
    else if (type == AddressPoolType::USER)  
        user_queue.push_back(tmp);  
}
```

- 将 queue 头部的页面 pop 掉

```
// 将页面从queue中pop  
int MemoryManager::pop(enum AddressPoolType type){  
    // printf("enter pop\n");  
    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;  
        }  
    }  
}
```

- 虚拟地址池分配失败时, 说明虚拟地址池中不存在连续的 count 个页面, 这个时候就要 pop 掉 kernel_queue 里的页面, 来使得虚拟地址池有足够的空间分配给新页面

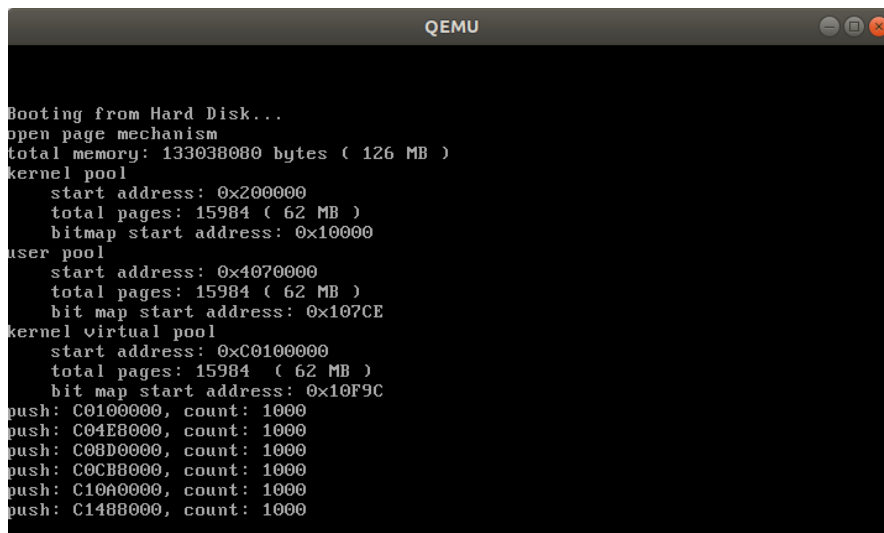
```
// 第一步：从虚拟地址池中分配若干虚拟页
int virtualAddress = allocateVirtualPages(type, count);
while(!virtualAddress)
{
    pop(type);
    virtualAddress = allocateVirtualPages(type, count);
}
```

因为虚拟地址池和物理地址池一样大，所以这里虚拟地址分配成功的话，物理地址池是一定有空闲页框分配给虚拟页的，而无需额外处理

- Setup 函数中，申请 20 次，每次申请 1000 页

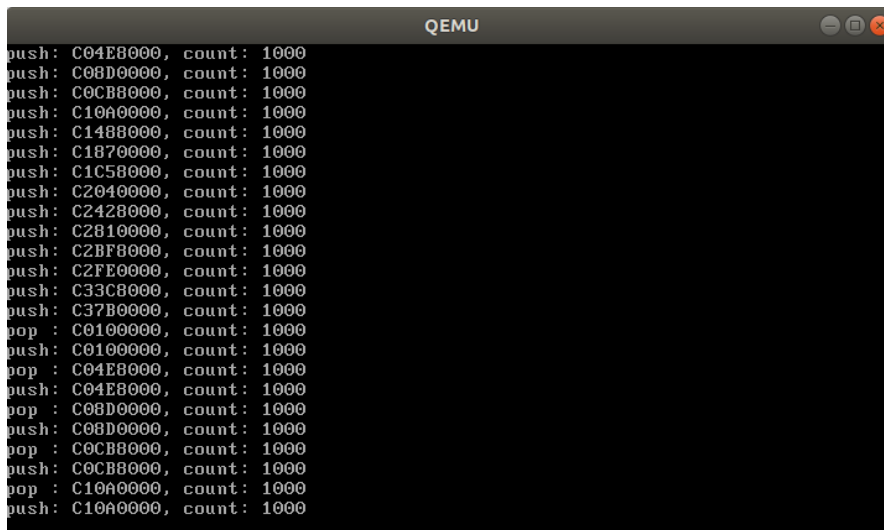
```
void first_thread(void *arg)
{
    for(int i = 0; i < 20; i++){
        char *addr = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1000);
    }

    asm_halt();
}
```



```
QEMU

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
kernel virtual pool
    start address: 0xC0100000
    total pages: 15984 ( 62 MB )
    bit map start address: 0x10F9C
push: C0100000, count: 1000
push: C04E8000, count: 1000
push: C08D0000, count: 1000
push: C0CB8000, count: 1000
push: C10A0000, count: 1000
push: C1488000, count: 1000
```



```
QEMU

push: C04E8000, count: 1000
push: C08D0000, count: 1000
push: C0CB8000, count: 1000
push: C10A0000, count: 1000
push: C1488000, count: 1000
push: C1870000, count: 1000
push: C1C58000, count: 1000
push: C2040000, count: 1000
push: C2428000, count: 1000
push: C2810000, count: 1000
push: C2BF8000, count: 1000
push: C2FE0000, count: 1000
push: C33C8000, count: 1000
push: C37B0000, count: 1000
pop : C0100000, count: 1000
push: C0100000, count: 1000
pop : C04E8000, count: 1000
push: C04E8000, count: 1000
pop : C08D0000, count: 1000
push: C08D0000, count: 1000
pop : C0CB8000, count: 1000
push: C0CB8000, count: 1000
pop : C10A0000, count: 1000
push: C10A0000, count: 1000
```

4) 虚拟内存管理代码分析

- 首先看分配内存：先从虚拟地址池中分配连续的 count 页

```
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;
    }

    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;
    }
}
```

- 然后为每个虚拟页，分配一个物理页。然后在页目录表和页表中建立起虚拟页和物理页之间的联系

```
// 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

        flag = connectPhysicalVirtualPage(vaddress, physicalPageAddress);
    }
    else
    {
        flag = false;
    }
}
```

- 如果分配物理页失败或者建立联系失败，则 flag 为 false，release 之前分配的页面

```
// 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;
}
```

- 再看释放页面：先释放每个虚拟页对应的物理页，再释放所有虚拟页

```
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);
}
```

释放物理页，释放所有虚拟页的代码也没有问题。

ToPDE 和 ToPTE 也没有问题，就是位运算截取前十位，中间十位

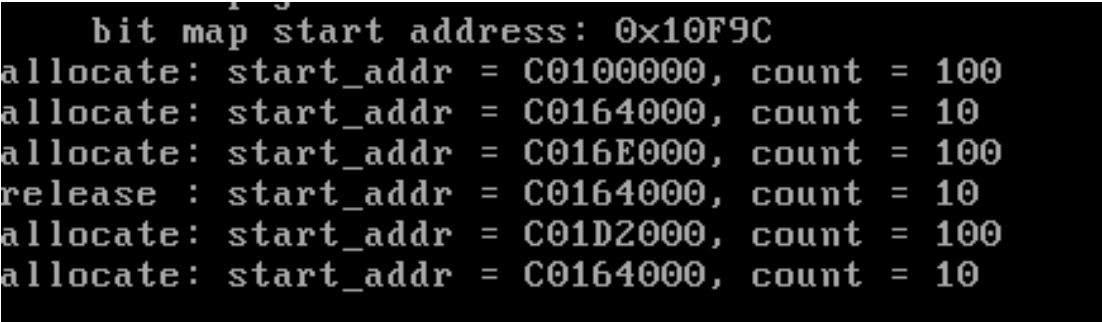
- 测试

```
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);

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

    asm_halt();
}
```



```
bit map start address: 0x10F9C
allocate: start_addr = C0100000, count = 100
allocate: start_addr = C0164000, count = 10
allocate: start_addr = C016E000, count = 100
release : start_addr = C0164000, count = 10
allocate: start_addr = C01D2000, count = 100
allocate: start_addr = C0164000, count = 10
```

现在看来，虚拟内存管理的代码没什么问题

3 关键代码

1) First-fit 算法

先越过已经分配的资源，直到找到一个未分配的资源，位置为 p；然后从这个位置 p 开始，计算一共有多少个连续的未分配内存，如果够 count 个，则分配成功；如果不够 count 个，只有 n 个 ($n < \text{count}$) 则继续从 p+n 的位置继续上面的逻辑；如果直到到达 length 的边界还没有分配成功，则分配失败

```
int BitMap::allocate(const 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
    }
```

2) Best-fit 算法

思想：遍历所有空缺（空缺指连续的一段未分配的页），在这些空缺中找出放得下“申请的页数量”的最小空缺，完成分配。

实现：用 `smallest_k` 记录遇到过的能放下申请页数的最小空缺，初始为-1。先越过已经分配的资源，直到找到一个未分配的资源，位置为 `p`；然后从这个位置 `p` 开始，计算一共有 `k` 个连续的未分配内存；如果 `k` 大于 `count`，而且 `k` 是遇到过的 `k` 中最小的，则将 `smallest_k` 置为 `k`；继续从 `p+k` 的位置继续上面的逻辑，直到到达 `length` 的边界；如果 `smallest_k == -1`，则分配失败，否则成功

```
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)
        break;

    // 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;
    // 直到搜索到头 或者 搜索到已分配资源
    while ((index < length) && !get(index))
    {
        ++empty;
        ++index;
    }

    // 存在大于等于count个的空位，且空位数量小于已知最小数量
    if (empty >= count && empty < smallest_fit_empty)
    {
        smallest_fit_empty = empty;
        smallest_fit_start = start;
    }
}
if(smallest_fit_start != -1){
    for (int i = 0; i < count; ++i)
    {
        set(smallest_fit_start + i, true);
    }
}
return smallest_fit_start;
```

3) 任务三中, FIFO 的 push 和 pop

每次 allocatePages 时, 将这次申请的虚拟地址, 页数 push 到队列里

```
// 将页面加入到queue中
void MemoryManager::push(enum AddressPoolType type, int addr, int count){
    printf("push: %x, count: %d\n", addr, count);
    static ListItem *tmp;
    tmp->addr = addr;
    tmp->num = count;
    if (type == AddressPoolType::KERNEL)
        kernel_queue.push_back(tmp);
    else if (type == AddressPoolType::USER)
        user_queue.push_back(tmp);
}
```

将队列里的头部元素 对应的所有页 都 release 掉, 然后队列 pop 掉头部元素。

```
// 将页面从queue中pop
int MemoryManager::pop(enum AddressPoolType type){
    // printf("enter pop\n");
    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;
        }
    }
}
```

当 allocatePages 中, 申请虚拟内存失败时, 将队列里的头部元素 对应的所有页 都 release 掉, 然后队列 pop 掉头部元素。再重新尝试申请虚拟内存, 如果再次失败, 则再次执行上述逻辑。

```
// 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);
}
```

因为虚拟地址池和物理地址池一样大, 所以这里虚拟地址分配成功的话, 物理地址池是一定有空闲页框分配给虚拟页的, 而无需额外处理

4 实验结果

1) Assignment1

```
linggm@linggm-virtual-machine:~/os_lab/lab7/assignment1
QEMU

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980

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
allocate 4 page successfully, start_address = 4070000
allocate 3 page successfully, start_address = 4074000
allocate 2 page successfully, start_address = 4077000
allocate 1 page successfully, start_address = 4079000
release 4 page successfully, start_address = 4070000
release 2 page successfully, start_address = 4077000
allocate 2 page successfully, start_address = 4070000
```

2) Assignment2

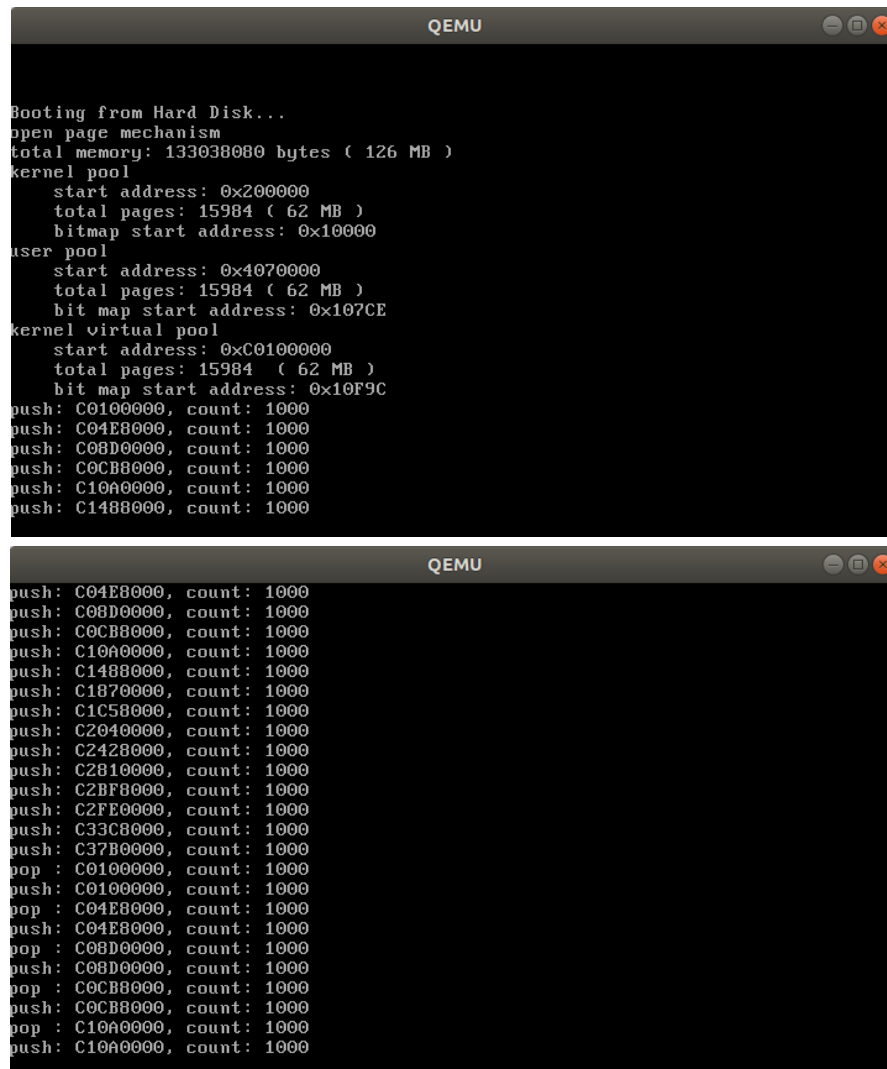
```
linggm@linggm-virtual-machine:~/os_lab/lab7/assignment2
QEMU

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980

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
allocate 4 page successfully, start_address = 4070000
allocate 3 page successfully, start_address = 4074000
allocate 2 page successfully, start_address = 4077000
allocate 1 page successfully, start_address = 4079000
release 4 page successfully, start_address = 4070000
release 2 page successfully, start_address = 4077000
allocate 2 page successfully, start_address = 4077000
```

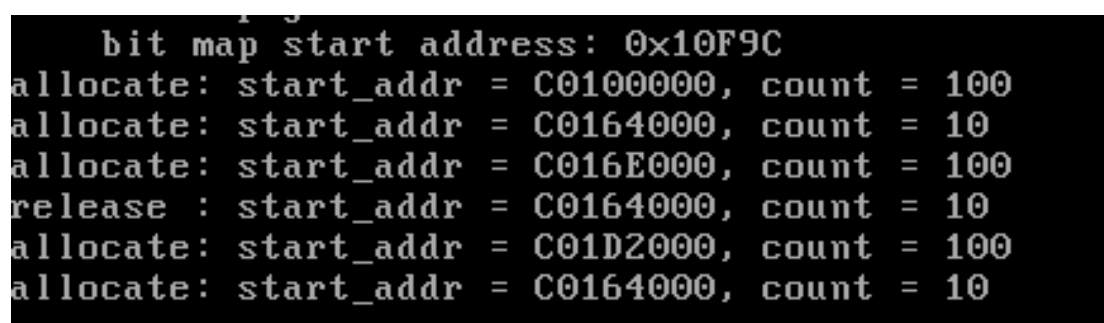
3) Assignment3

连续申请 20 次，每次申请 1000 页，则前 15 次申请都成功，后 5 次申请要先 pop，也就是要先进行页面置换，才能申请成功。结果符合预期。



```
QEMU
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
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x10F9C
push: C0100000, count: 1000
push: C04E8000, count: 1000
push: C08D0000, count: 1000
push: C0CB8000, count: 1000
push: C10A0000, count: 1000
push: C1488000, count: 1000
push: C04E8000, count: 1000
push: C08D0000, count: 1000
push: C0CB8000, count: 1000
push: C10A0000, count: 1000
push: C1488000, count: 1000
push: C1870000, count: 1000
push: C1C58000, count: 1000
push: C2040000, count: 1000
push: C2428000, count: 1000
push: C2810000, count: 1000
push: C2BF8000, count: 1000
push: C2FE0000, count: 1000
push: C33C8000, count: 1000
push: C37B0000, count: 1000
pop : C0100000, count: 1000
push: C0100000, count: 1000
pop : C04E8000, count: 1000
push: C04E8000, count: 1000
pop : C08D0000, count: 1000
push: C08D0000, count: 1000
pop : C0CB8000, count: 1000
push: C0CB8000, count: 1000
pop : C10A0000, count: 1000
push: C10A0000, count: 1000
```

4) Assignment4



```
bit map start address: 0x10F9C
allocate: start_addr = C0100000, count = 100
allocate: start_addr = C0164000, count = 10
allocate: start_addr = C016E000, count = 100
release : start_addr = C0164000, count = 10
allocate: start_addr = C01D2000, count = 100
allocate: start_addr = C0164000, count = 10
```

认为无异常