

19335030_陈至雪_lab7

实验概述：

在本次实验中，我们首先学习如何使用位图和地址池来管理资源。然后，我们将实现在物理地址空间下的内存管理。接着，我们将会学习并开启二级分页机制。在开启分页机制后，我们将实现在虚拟地址空间下的内存管理。

本次实验最精彩的地方在于分页机制。基于分页机制，我们可以将连续的虚拟地址空间映射到不连续的物理地址空间。同时，对于同一个虚拟地址，在不同的页目录表和页表下，我们会得到不同的物理地址。这为实现虚拟地址空间的隔离奠定了基础。但是，本实验最令人困惑的地方也在于分页机制。开启了分页机制后，程序中使用的地址是虚拟地址。我们需要结合页目录表和页表才能确定虚拟地址对应的物理地址。而我们常常会忘记这一点，导致了不知道某些虚拟地址表示的具体含义。

实验要求：

- DDL：
 - 提交的内容：将4个assignment的代码和实验报告放到压缩包中，命名为“lab7-姓名-学号”，并交到课程网站上[\[http://course.dds-sysu.tech/course/3/homework\]](http://course.dds-sysu.tech/course/3/homework)
 - 材料的代码放置在 `src` 目录下。
1. 实验不限语言，C/C++/Rust都可以。
 2. 实验不限平台，Windows、Linux和MacOS等都可以。
 3. 实验不限CPU，ARM/Intel/Risc-V都可以。

实验内容：

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。

实验步骤:

Assignment1:

1、进行内存探查

进行内存探查，获取系统中可管理的内存容量。

在实模式下通过 `int 15h` 来获取内存大小，然后将中断返回的内存大小的信息保存在寄存器 `ax` 和 `bx` 中。其中 `ax` 存放0~15MB的内存大小，`bx` 存放16MB~4GB的内存大小，单位是64KB。

由于要在是模式下获取内存大小，因此在进入保护模式之前，就要先进行中断得到内存大小的信息，然后将信息保存在固定的地址（这里放在 `0x7c00`），好让我们进入保护模式后仍然能访问该信息。此过程在 `mbr.asm` 中实现。

```
19  load_bootloader:
20      push ax
21      push bx
22      call asm_read_hard_disk ; 读取硬盘
23      add sp, 4
24      inc ax
25      add bx, 512
26      loop load_bootloader
27
28      ;get memory size
29      mov ax, 0xe801
30      int 15h
31      mov [0x7c00], ax
32      mov [0x7c00+2], bx
33
34      jmp 0x0000:0x7e00 ; 跳转到bootloader
35
36      jmp $ ; 死循环
```

然后在 `include/os_constant.h` 中定义保存内存大小信息的地址 `#define MEMORY_SIZE_ADDRESS 0x7c00`。

然后 `first_thread` 中读取内存大小，然后对内存大小进行计算，最后输出内存大小。

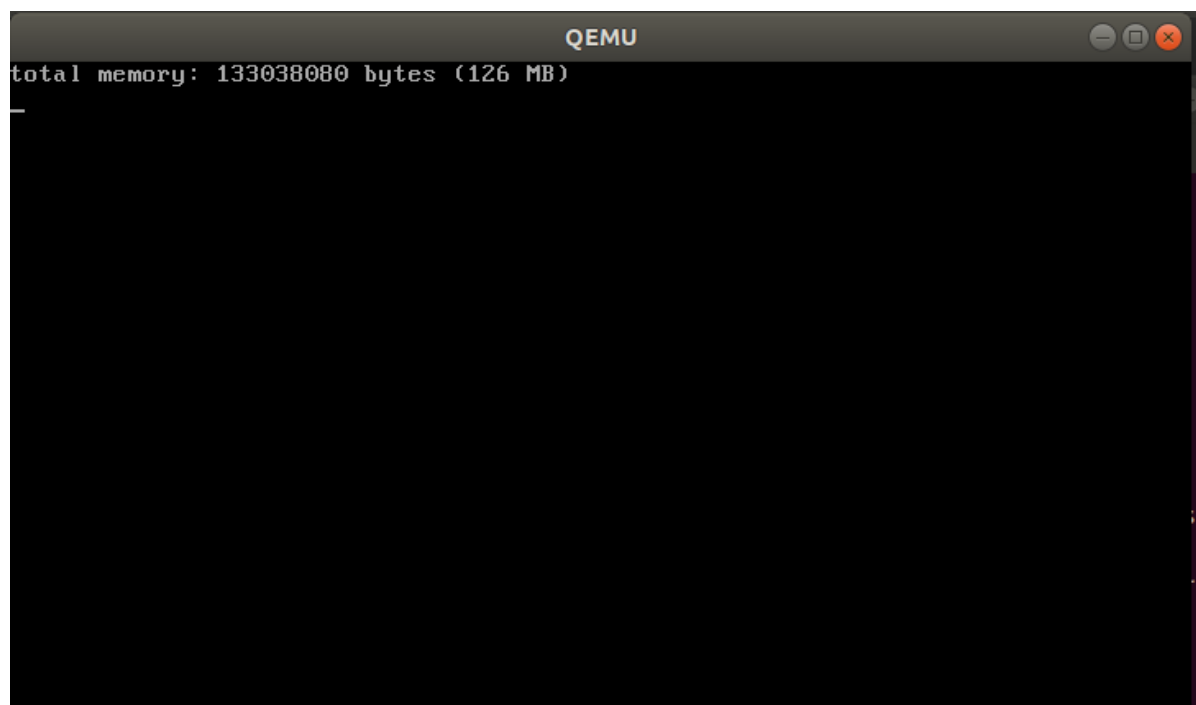
内存总容量 = $(ax \cdot 1024 + bx \cdot 64 \cdot 1024) \text{bytes}$ 。

```

15 void first_thread(void *arg)
16 {
17     // 第1个线程不可以返回
18     stdio.moveCursor(0);
19     for (int i = 0; i < 25 * 80; ++i)
20     {
21         stdio.print(' ');
22     }
23     stdio.moveCursor(0);
24
25     int memory = *((uint32 *)MEMORY_SIZE_ADDRESS);
26     // ax寄存器保存的内容
27     int low = memory & 0xffff;
28     // bx寄存器保存的内容
29     int high = (memory >> 16) & 0xffff;
30     memory = low * 1024 + high * 64 * 1024;
31     printf("total memory: %d bytes (%d MB)\n", memory, memory / 1024 / 1024);
32
33     asm_halt();
34 }

```

在终端运行查看结果：



```

QEMU
total memory: 133038080 bytes (126 MB)

```

总共的内存大小有126MB。

2、实现位图BitMap

位图是为每个资源单位分配一位，用于标记是否被分配。一个资源单位大小为4KB。

在 `include/bitmap.h` 中声明：

```

#ifndef BITMAP_H
#define BITMAP_H

#include "os_type.h"

```

```

class BitMap
{
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&) {}
};

#endif

```

实现的主要功能有在申请资源时对资源进行被申请标记，在释放资源时，对资源进行可用标记。

具体实现在 `utils/bitmap.cpp` 中

```

#include "bitmap.h"
#include "stdlib.h"
#include "stdio.h"

BitMap::BitMap()
{
}

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;
    }
}

```

```

bool BitMap::get( const int index ) const{
    int pos = index/8;
    int offset = index % 8;
    return ( bitmap[pos] & ( 1<<offset ) );
}

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

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

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

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

    int index,empty,start;

    index = 0;
    while( index < length )
    {
        //find the position that have not been allocate
        while( index < length && get( index ) )
            index++;

        if( index == length )
            return -1;

        empty = 0;
        start = index;
        while( ( index < length ) && ( !get( index ) ) && ( empty < count ) )
        {
            ++ empty;
            ++ index;
        }

        //we can find count empty neicun
        if( empty == count )
        {
            for( int i = 0; i < count; ++i )
            {
                set(start+i,true);
            }
            return start;
        }
    }
    return -1;
}

```

```

}

void BitMap::release( const int index, const int count )
{
    for( int i = 0; i < count; ++i )
    {
        set( index + i, false );
    }
}

char* BitMap::getBitmap(){
    return (char*)bitmap;
}

int BitMap::size()const{
    return length;
}

```

一个bitmap的存储区域为一个字节，因此给定一个资源单元的序列号*i*，无法通过bitmap[i]直接修改资源单元的状态。正确的做法是先定位到存储第*i*个资源单元的字节序号pos，然后再确定第*i*个资源单元的状态位在第pos字节中的偏移量offset。即 $i = 8 \cdot pos + offset, 0 \leq offset < 8$ 。一般 $pos = i/8, offset = index$ 。

3、实现地址池

地址池的功能是实现了对地址空间的管理。BitMap的功能是对资源的管理，地址池的实现用到了BitMap。

声明在 `include/address_pool.h` 中：

```

#ifndef ADDRESS_POOL_H
#define ADDRESS_POOL_H

#include "bitmap.h"
#include "os_type.h"

class AddressPool
{
public:
    BitMap resources;
    int startAddress;

public:
    AddressPool();
    void initialize( char *bitmap, const int length, const int startAddress );
    int allocate( const int count );
    void release( const int address, const int amount );
};

#endif

```

函数主要有初始化地址池，这个函数很关键，它指明了这个地址池有多大，能分配的资源有多少。然后就是向地址池申请资源的函数，以及释放资源的函数。

实现在 `utils/address_pool.cpp` 中:

```
#include "address_pool.h"
#include "os_constant.h"

AddressPool::AddressPool()
{
}

void AddressPool::initialize( char *bitmap, const int length, const int startAddress)
{
    resources.initialize( bitmap, length );
    this->startAddress = startAddress;
}

int AddressPool::allocate( const int count )
{
    int start = resources.allocate(count);
    return( start == -1 ) ? -1 : ( start* PAGE_SIZE + startAddress );
}

void AddressPool::release( const int address, const int amount )
{
    resources.release( ( address - startAddress ) / PAGE_SIZE, amount );
}
```

4、实现物理页内存管理

将物理内存划分为内核空间和用户空间。一个空间用一个地址池表示。声明在 `include/memory.h` 中。

```
#ifndef MEMORY_H
#define MEMORY_H
#include "address_pool.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;
public:
    MemoryManager();
    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);

// 释放从paddr开始的count个物理页
void releasePhysicalPages( enum AddressPoolType type, const int
startAddress, const int count );

// 获取内存总容量
int getTotalMemory();
};

#endif

```

实现:

```

#include "memory.h"
#include "os_constant.h"
#include "stdlib.h"
#include "asm_utils.h"
#include "stdio.h"
#include "program.h"
#include "os_modules.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);

    kernelPhysical.initialize( (char*)kernelPhysicalBitMapStart, kernelPages,
kernelPhysicalStartAddress );
    userPhysical.initialize( (char*)userPhysicalBitMapStart, userPages,
userPhysicalStartAddress );
}

```



```

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);
}

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 );
    }
}

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;
}

```

在进行初始化的时候，我们先用第一步实现的内存探查对获取内存的大小，然后预留可用的空间，将预留内存分成两等分，分别划分为内核物理地址空间和用户物理地址空间。然后再在1MB以下的空间处人为划分存放位图的区域，用来存放内核地址空间和用户空间的位图。最后打印内存管理的基本信息。

5、实现二级分页机制

在 `MemoryManager` 类中添加函数 `void openPageMechanism()` .实现如下：

```

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");
}

```

其中，常量定义在 `include/os_constant.h` 下，如下所示：

```

#define PAGE_DIRECTORY 0x100000

```

先建立好内核所在地址的页目录表和页表。然后对它们进行初始化建立虚地址到物理地址的恒等映射。最后将页目录表的地址放入cr3寄存器，然后将cr0寄存器的PG位置1开启分页机制。开启分页机制的函数实现如下：

```
asm_init_page_reg:
    push ebp
    mov ebp, esp

    push eax

    mov eax, [ebp + 4 * 2]
    mov cr3, eax ; 放入页目录表地址
    mov eax, cr0
    or eax, 0x80000000
    mov cr0, eax ; 置PG=1, 开启分页机制

    pop eax
    pop ebp

    ret
```

最后，在 `setup.cpp` 中开启并验证：

```
extern "C" void setup_kernel()
{
    // 中断管理器
    interruptManager.initialize();
    interruptManager.enableTimeInterrupt();
    interruptManager.setTimeInterrupt((void *)asm_time_interrupt_handler);

    // 输出管理器
    stdio.initialize();

    // 进程/线程管理器
    programManager.initialize();

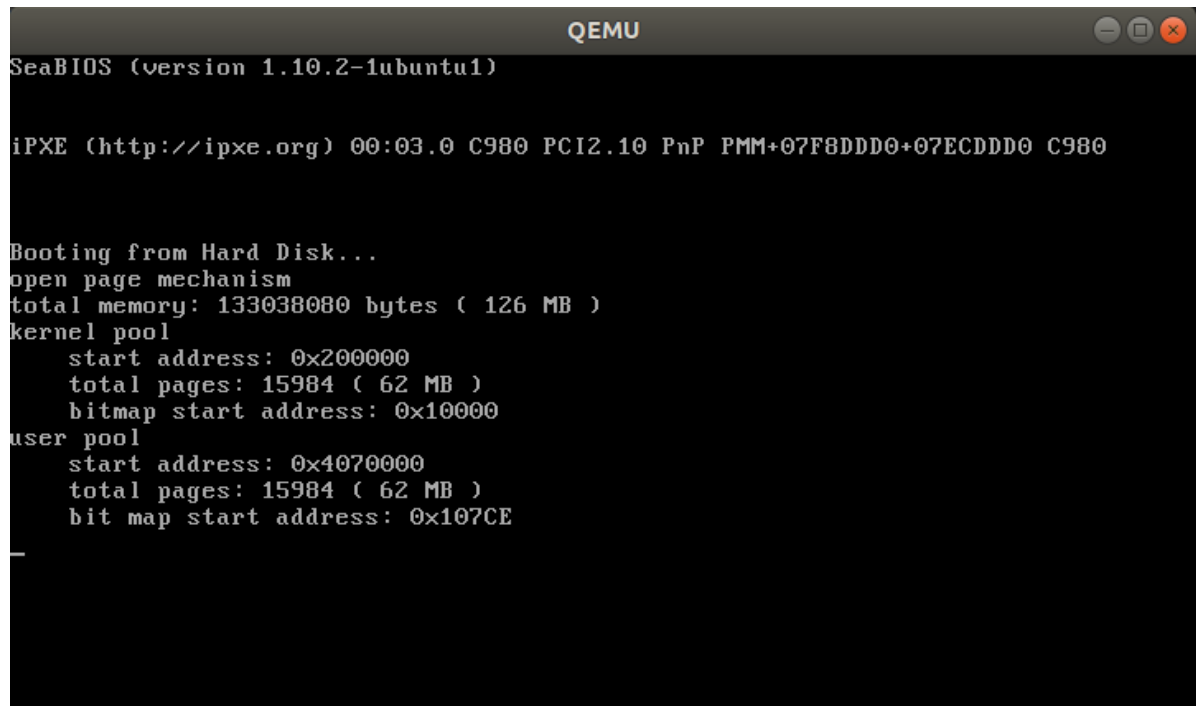
    // 内存管理器
    memoryManager.openPageMechanism();
    memoryManager.initialize();

    // 创建第一个线程
    int pid = programManager.executeThread(first_thread, nullptr, "first
thread", 1);
    if (pid == -1)
    {
        printf("can not execute thread\n");
        asm_halt();
    }

    ListItem *item = programManager.readyPrograms.front();
    PCB *firstThread = ListItem2PCB(item, tagInGeneralList);
    firstThread->status = RUNNING;
    programManager.readyPrograms.pop_front();
    programManager.running = firstThread;
    asm_switch_thread(0, firstThread);
}
```

```
asm_halt();  
}
```

编译运行，查看输出：



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

```
SeaBIOS (version 1.10.2-1ubuntu1)  
  
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
```

可以看到，我们开启了分页机制，并预留了124MB的空间，将预留的内存空间划分为了大小相等的两部分，一部分为内核的空间，一部分为用户的空间。也可以看到两个地址空间的起始地址和位图的起始地址。

Assignment2:

first_fit的实现和assignment1的实现一样，一次这里实现了best_fit算法。

实现best_fit:

```
int BitMap::best_fit( const int count )  
{  
    if (count == 0)  
        return -1;  
  
    int index, empty, start, min;  
    int temp_start;  
  
    index = empty = temp_start = 0;  
    start = min = -1;  
    while( index < length ){  
        while( index < length && get( index ) )  
            index ++;  
  
        if( index == length )  
            break;  
        //return start;  
  
        empty = 0;  
        temp_start = index;
```

```

while( ( index < length ) && (!get(index)) && (empty < count) )
{
    ++empty;
    ++index;
}

if( index == length )
    break ;

if( empty == count ){
    if( start == -1 )
        start = temp_start;

    int t = 0;
    while( ( index < length ) && (!get(index)) ){
        t ++;
        index++;
    }

    //initialize min
    if( min == -1 ) min = t;
    start = min < t ? start : temp_start;
    min = min < t ? min : t ;
}

if( start != -1 ){
    for (int i = 0; i < count; ++i)
    {
        set(start + i, true);
    }
}

return start;
}

```

对内存区域进行一次遍历。每次申请资源都从头开始扫描内存区域，遍历过程中记录最小的可以装下申请数量的连续资源块的起始地址。遍历完后，将得到的以start开始的连续count块内存区域设置为已被申请的状态。

验证：

在setup.cpp中分4次申请内存，每次申请10个单元。前三次连着一一起申请，然后对第二次申请的资源进行释放。然后申请第四次资源，最后按照第一次，第三次，第四次申请的顺序依次释放资源。按照best_fit的思想，第四次申请的资源的start应该和第二次一样。

```

int count = 10;
int start1 ,start2, start3, start4;
//1
printf( "1: allocate %d memory in kernel.\n",count );
start1 = memoryManager.allocatePhysicalPages( AddressPoolType::KERNEL,
count);
printf( "1: My memory start is %d. \n", start1 );
//2
printf( "2: allocate %d memory in kernel.\n",count );
start2 = memoryManager.allocatePhysicalPages( AddressPoolType::KERNEL,
count);
printf( "2: My memory start is %d. \n", start2 );

```

```

//3
printf( "3: allocate %d memory in kernel.\n",count );
start3 = memoryManager.allocatePhysicalPages( AddressPoolType::KERNEL,
count);
printf( "3: My memory start is %d. \n", start3 );

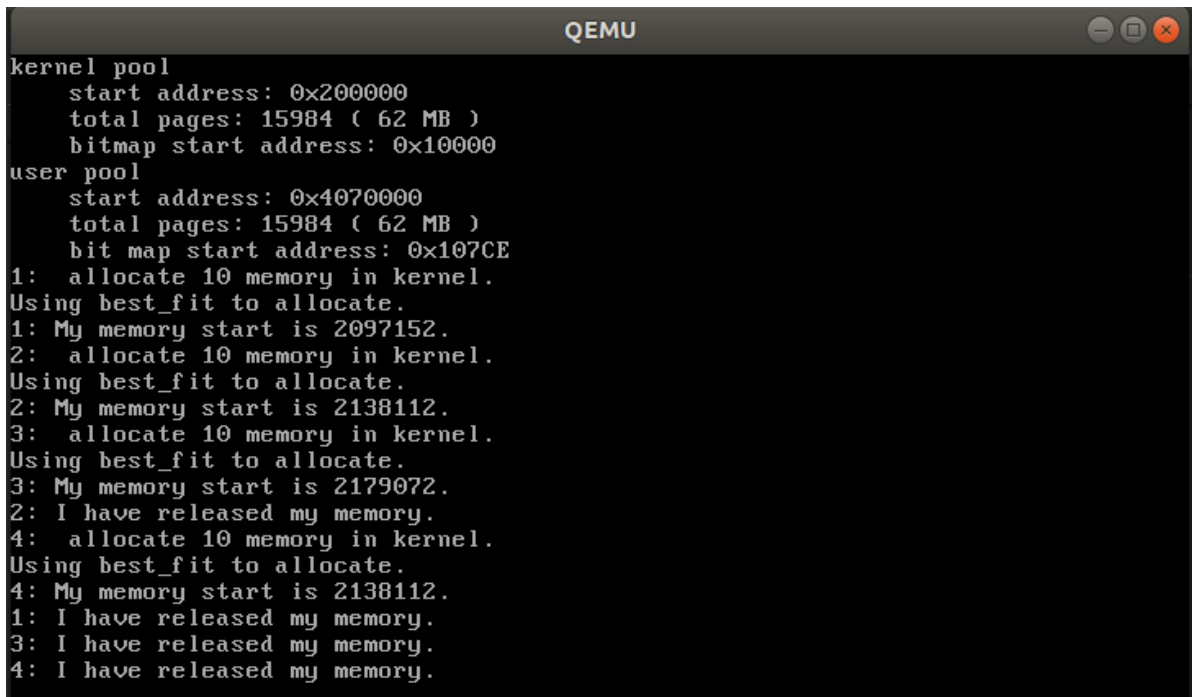
//release 2
memoryManager.releasePhysicalPages( AddressPoolType::KERNEL , start2,
count);
printf( "2: I have released my memory.\n" );

//4
printf( "4: allocate %d memory in kernel.\n",count );
start4 = memoryManager.allocatePhysicalPages( AddressPoolType::KERNEL,
count);
printf( "4: My memory start is %d. \n", start4 );

// release 1
memoryManager.releasePhysicalPages( AddressPoolType::KERNEL , start1,
count);
printf( "1: I have released my memory.\n" );
//release 3
memoryManager.releasePhysicalPages( AddressPoolType::KERNEL , start3,
count);
printf( "3: I have released my memory.\n" );
//release 4
memoryManager.releasePhysicalPages( AddressPoolType::KERNEL , start4,
count);
printf( "4: I have released my memory.\n" );

```

编译运行，查看结果：



```

QEMU
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
1: allocate 10 memory in kernel.
Using best_fit to allocate.
1: My memory start is 2097152.
2: allocate 10 memory in kernel.
Using best_fit to allocate.
2: My memory start is 2138112.
3: allocate 10 memory in kernel.
Using best_fit to allocate.
3: My memory start is 2179072.
2: I have released my memory.
4: allocate 10 memory in kernel.
Using best_fit to allocate.
4: My memory start is 2138112.
1: I have released my memory.
3: I have released my memory.
4: I have released my memory.

```

可以看到第四次申请的资源的start位置和第2次申请的资源的start位置是一样的。说明了实现的正确性。

Assignment3:

实现FIFO换页机制。

1、在内存中分配一块区域，用于记录每一个页被申请的时间长短。在使用FIFO时只要找到被申请时间最长的页进行替换即可。

```
int kernelPhysicalBitMapStart = BITMAP_START_ADDRESS;
int kernelPhysicalTimeStart = kernelPhysicalBitMapStart + ceil(kernelPages,
8);
int userPhysicalBitMapStart = kernelPhysicalTimeStart + ceil(kernelPages,
8);
int userPhysicalTimeStart = userPhysicalBitMapStart + ceil(userPages, 8);
int kernelVirtualBitMapStart = userPhysicalBitMapStart + ceil(userPages, 8);
int kernelVirtualTimeStart = kernelVirtualBitMapStart + ceil( kernelPages,
8 );
```

在每次申请物理页时，就会对已被申请的物理页的时间进行更新：

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

            return start;
        }
    }
}
```

```

    }

    return -1;
}

```

对时间进行更新的函数:

```

void BitMap::add(){
    int index = 0;
    int bytes = ceil(length, 8);
    while( index < bytes )
    {
        if( time[index] != 0 )
            time[index] ++;
        index ++;
    }
}

```

```

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);
        add();
        time[pos] = time[pos]++;
    }
    if( !status )
        time[pos] = 0;
}

```

每次释放物理页，就将该物理页的时间设置为0.

```

int BitMap::maxtime(){
    int index = 0;
    int pos = 0;
    int bytes = ceil(length, 8);
    int max = time[index];
    while( index < bytes )
    {
        if( time[index] > max ){
            max = time[index];
            pos = index;
        }
        index ++;
    }
    return pos;
}

```


3、在address_pool中进行封装:

```
int AddressPool::maxtime(){
    int start = resources.maxtime();
    return start;
}
```

4、在memory.cpp中实现FIFO:

先对页进行访问

```
void MemoryManager::visit( int vaddress, enum AddressPoolType type ) {
    int *pte;
    pte = (int *)toPTE(vaddress);
    if( *pte == 0 ){
        printf("Can't find paddr, ask FIFO to help!\n");
        int paddr = FIFO( type, vaddress );
        connectPhysicalVirtualPage( vaddress, paddr );
        printf( "FIFO give me a paddr, it is %d .\n",paddr );
    }
    else{
        int paddr = vaddr2paddr( vaddress );
        printf( "I can find paddr in %d .\n", paddr );
    }
}
```

FIFO的实现:

```
int MemoryManager::FIFO( enum AddressPoolType type, const int virtualAddress)
{
    int phy;
    if (type == AddressPoolType::KERNEL)
    {
        int addr = kernelVirtual.maxtime();
        releasePhysicalPages(type, addr, 1);
        phy = allocatePhysicalPages(type, 1);
        //connectPhysicalVirtualPage( virtualAddress, phy );
    }
    if( type == AddressPoolType::USER )
    {
        int addr = userPhysical.maxtime();
        releasePhysicalPages(type, addr, 1);
        phy = allocatePhysicalPages(type, 1);
        //connectPhysicalVirtualPage( virtualAddress, phy );
    }
    return phy;
}
```

找到时间片最长的物理页进行删除，然后重新分配一个物理页给虚拟地址。

Assignment4:

实现虚拟内存管理

由于实现了分页机制，我们需要虚拟地址池来管理物理地址池。由上面的实现知道，我们有内核物理地址池和用户物理地址池。因此虚拟地址池也必须分内核虚拟地址池和用户虚拟地址池。不一样的是，内核虚拟地址池是全局的，用户虚拟地址池则为每个进程所拥有。

每个进程有自己的虚拟用户地址空间，在 `MemoryManager` 中管理全局的用户物理地址空间、内核虚拟地址空间和内核物理地址空间。

在前面的 `MemoryManager` 的基础上加入内核虚拟地址池。

```
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;
public:
```

然后，在初始化函数中，添加初始化内核虚拟地址池的代码：

```
void MemoryManager::initialize()
{
    ...
    int kernelVirtualBitMapStart = userPhysicalBitMapStart + ceil(userPages, 8);
    ...
    ...
    kernelVirtual.initialize( ( char * )kernelVirtualBitMapStart, kernelPages,
    KERNEL_VIRTUAL_START );
    ...
    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,
        kernelBitMapStart);
}
```

在 `os_constant.h` 中声明常量 `#define KERNEL_VIRTUAL_START 0xc0100000`

编译执行，查看结果：

```
QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

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
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x10F9C
-
```

可以看到我们创建了一个内核虚拟地址池。

然后实现页内存分配。

由于现在实现的是分页机制，所以程序只能通过虚拟地址来访问数据，因此实现页内存分配需要解决三个问题。分别是

- 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。

从虚拟地址池中分配若干连续的虚拟页：

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

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

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

由于还没有实现用户进程，因此暂时没有用户地址池的分配。对于非内核地址池的分配，返回0，表示返回失败。对于内核地址池的分配，用函数 `kernelVirtual.allocate(count)` 实现。

接下来，对每一个虚拟页，从物理地址池中分配1页。通过函数 `allocatePhysicalPages` 实现。

然后，为虚拟页建立页目录项和页表项。

由虚拟地址到物理地址的变换如下：

- CPU先取虚拟地址的31-22位`virtual[31:22]`，在`cr3`寄存器中找到页目录表的物理地址，然后根据页目录表的物理地址在页目录表中找到序号为`virtual[31:22]`的页目录项，读取页目录项中的页表的物理地址。

- CPU再取虚拟地址的21-12位virtual[21:12]virtual[21:12]，根据第一步取出的页表的物理地址，在页表中找到序号为virtual[21:12]virtual[21:12]的页表项，读取页表项中的物理页的物理地址physicalphysical。
- 用物理页的物理地址的31-12位physical[31:12]physical[31:12]替换虚拟地址的31-12位virtual[31:12]virtual[31:12]得到的结果就是虚拟地址对应的物理地址。

用函数toPDE找到页目录表项。

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

用函数 toPTE找到页表项。

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

找到pde和pte后，建立从虚拟页到物理页的映射。用函数

MemoryManager::connectPhysicalVirtualPage(const int virtualAddress, const int physicalPageAddress) 实现。

```
bool MemoryManager::connectPhysicalVirtualPage(const int virtualAddress, const
int physicalPageAddress)
{
    // 计算虚拟地址对应的页目录表项和页表项
    int *pde = (int *)toPDE(virtualAddress);
    int *pte = (int *)toPTE(virtualAddress);

    // 页目录表 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;
}
```

先检查pde中是否有对应的页表。若没有，则先分配一个物理页，然后初始化刚分配的物理页，并将其写入pde，作为pde指向的页表。若pde对应的页表存在，则将之前为虚拟页分配的物理页地址写入pte。

然后，就可以实现页内存的申请了，函数实现如下：

```
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;

    // 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;
        }

        // 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;
}
```

第一步先从相应的虚拟地址池中分配相应数量的页。若申请不成功则返回0；

第二步为申请成功的虚拟页指定相应数量的物理页，同时建立页表目录项和页表项，使虚拟页内的地址可以通过二级分页机制变换到物理页内。若不能为每个虚拟页指定物理页，则将前面已经指定好的物理页释放，同时将虚拟页表中分配出去的虚拟页表项释放。

接下来实现页内存释放。

在分配页内存时，如果遇到物理页无法分配的情况，之前成功分配的虚拟页和物理页都要释放。否则就会造成内存泄漏，这部分内存无法再被分配。

页内存的释放分两个步骤完成：

- 对每一个虚拟页，释放为其分配的物理页。
- 释放虚拟页。

下面实现对每一个虚拟页，释放为其分配的物理页。

为了释放物理页，找到虚拟页对应的物理页的物理地址。根据分页机制，一个虚拟地址对应的物理页的地址是存放在页表项中的。因此，我们先求出虚拟地址的页表项的虚拟地址，然后访问页表项，页表项内容的31-12位就是物理页的物理地址，最后替换虚拟地址的31-12位即可得到虚拟地址对应的物理地址。

```
int MemoryManager::vaddr2paddr(int vaddr)
{
    int *pte = (int *)toPTE(vaddr);
    int page = (*pte) & 0xfffff000;
    int offset = vaddr & 0xfff;
    return (page + offset);
}
```

释放虚拟页：

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

因为还没有实现用户进程，这里只处理了内核虚拟地址池中的地址。

最后，释放页内存的代码如下：

```
void MemoryManager::releasePages(enum AddressPoolType type, const int virtualAddress, const int count)
{
    int vaddr = virtualAddress;
    int *pte, *pde;
    bool flag;
    const int ENTRY_NUM = PAGE_SIZE / sizeof(int);

    for (int i = 0; i < count; ++i, vaddr += PAGE_SIZE)
    {
        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;
}

releaseVirtualPages(type, virtualAddress, count);
}
```

完成了虚拟地址到物理地址的转换，然后释放物理页。释放了物理页后，将虚拟页对应的页表项置0，这是为了防止在虚拟页释放后被再次寻址。

这里对内存的分配每次都是在相应的分区里从头开始扫描，直到找到大小适合的块。若申请的大小不足，则将申请到一半的内存释放，这样就不会空占内存。释放内存时，会把页表项设置为不存在，防止释放后再次被使用，这样保障了安全性。由Assignment2的测试样例可以看到，内存能正常分配，这样看来，程序不存在bug。

实验感想：

这次实验让我对操作系统的内存管理有了深刻的理解。

- 1、操作系统将内存区域手动划分为两个区域，分别是内核区域和用户区域。为了记录内存的使用情况，要在实现内存管理之前，手动分配一块内存区域用于记录。比如记录物理页内存是否被访问的bitmap就处于这块内存区域。
- 2、在对页进行分配时，如果分配失败，之前分配好的虚拟页和物理页都需要被释放，否则将会产生内存泄漏。
- 3、为了实现对内存的合理利用，我们采用二级分页机制。第一级是页目录表，第二级是页表，第三级是物理内存。一个虚拟地址要找到物理地址，需要通过前10位在页目录表里找到对应的页表物理地址，再通过这个虚拟地址的中间10位在这个页表里找到它的在内存的页框号，再补上虚拟地址的后12位页内偏移量，然后去内存里找对应位置的数据。
- 4、在二级页表机制下，页目录表、页表、物理页的大小均为4KB。页目录项和页表项的大小均为4B。因此，在每一个页表中，页表项的数目为 $4KB/4B=1024$ 。同样，一个页目录表中有页目录项1024个。因此一共有 $1024*1024$ 个页表，每一个页表对应一个物理页，因此一共可以映射 $1024*1024$ 个物理页，每个物理页大小为4KB，因此能映射 $1024*1024*4KB=4GB$ 大小的内存。
- 5、Assignment3的实现对我来说非常困难，这里的实现虽然在分配物理页上可以做到FIFO，但是由于实现是从底向上进行的，即直接在物理页判断应该换掉哪一页，这样做的缺陷是无法根据应该换掉的物理页找到与它映射的虚拟地址。也就是说，当用FIFO把一页换下内存时，实际上这里并没有把虚拟地址所指的值置为0。