

操作系统原理实验报告

**实验名称:** 实验七 内存管理

**授课教师：** 张青

**学生姓名:** 曾慧蕾

**学生学号:** 21307358

1. **实验要求**

**Assignment 1 实现二级分页机制**

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 实现动态分区算法**

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 实现页面置换算法**

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。

**Assignment 4 实现虚拟页内存管理**

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。
* 构造测试例来分析虚拟页内存管理的实现是否有bug。若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。

1. **实验过程+关键代码+结果展示**

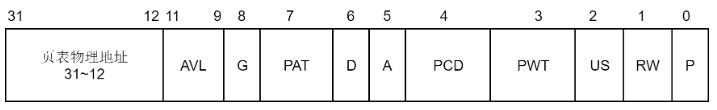
* **Assignment 1 实现二级分页机制**

要实现二级分页机制，首先确定好流程：

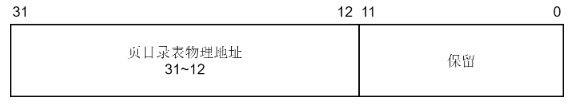
1. 规划好页目录表和页表在内存中的位置，然后初始化。
2. 将页目录表的地址写入cr3。
3. 将cr0的PG位置1。

第一步，规划好页目录表和页表在内存中的位置并写入内容。首先，我们需要明确一点，页目录表和页表是需要在内存中特意地划分出位置来存放的。所以，我们需要在内存中指定页目录表和页表存放的位置。同时，页目录表和页表的物理地址必须是4KB的整数倍，也就是低12位为0。

规定了页目录表的位置后，我们根据线性地址空间的大小来确定需要分配的页表的数量和位置，不必一开始就分配完1024个页表给页目录表。规划好了页目录表的位置后，我们向页目录表中写入页表对应的页目录项。页目录项的结构如下。



第二步，将页目录表的地址写入cr3。cr3寄存器保存的是页目录表的地址，使得CPU的MMU能够找到页目录表的地址，然后自动地将线性地址转换成物理地址。我们在建立页目录表和页表后，需要将页目录表地址放到CPU所约定的地方，而这个地方是cr3。cr3又被称为页目录基址寄存器 PDBR，其内容如下。



cr3可以直接使用mov指令赋值。

第三步，将cr0的PG位置1。启动分页机制的开关是将控制寄存器 cr0 的 PG 位置 1，PG 位是cr0寄存器的第31位，PG位为1后便进入了内存分页运行机制。

综合上述3步骤后便可以开启分页机制，我们下面使用代码实现。

class MemoryManager{

    ...

    // 开启分页机制

    void openPageMechanism();

};

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");

}

extern "C" void setup\_kernel(){

    ...

    // 内存管理器

    memoryManager.openPageMechanism();

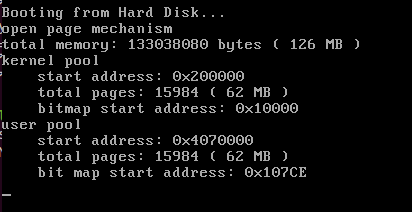
    memoryManager.initialize();

    // 创建第一个线程

    ...

}

将上述代码编译后，在终端运行结果如下所示：



可见二级分页机制已成功开启，各页地址与大小已成功输出。

* **Assignment 2 实现动态分区算法**

首先，在bigmap.cpp中实现first fit和best fit算法。明确：

1. first-fit：找到第一个能容得下的连续未分配页就分配；
2. best-fit：找到能满足大小需求且大小最小的空闲空间

first fit：

int BitMap::first\_fit(const int count)//first fit{

    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;

}

Best fit：

int BitMap::best\_fit(const int count)//best fit{

    if (count == 0)

        return -1;

    int index, empty, start;

    int address = -1;

    int address\_len = -1;

    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;

            ++index;

        }

        // 空闲的内存足够大，根据历史记录来判断是否要更新

        if (empty >=count &&(address==-1 || empty<address\_len)){

            address=start;

            address\_len=empty;

        }

    }

    if (address!=-1){

        for (int i = 0; i < count; ++i){

                set(address + i, true);

            }

        return address;

    }

    else return -1;

}

然后，在setup.cpp中编写样例，使用两种动态分区算法。

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);

    // 都设为内核程序，从内核空间中分配物理页

    char \*p0=(char \*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 128);

    printf("Allocated 128 pages for p0, starting at %d.\n", p0);

    char \*p1=(char \*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 64);

    printf("Allocated 64 pages for p1, starting at %d.\n", p1);

    char \*p2=(char \*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 16);

    printf("Allocated 16 pages for p2, starting at %d.\n", p2);

    char \*p3=(char \*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 8);

    printf("Allocated 8 pages for p3, starting at %d.\n", p3);

    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, int(p0), 128);

    printf("Released 128 pages for p0.\n");

    memoryManager.releasePhysicalPages(AddressPoolType::KERNEL, int(p2), 16);

    printf("Released 16 pages for p2.\n");

    char \*p4=(char \*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 16);

    printf("Allocated 16 pages for p4, starting at %d.\n", p4);

    char \*p5=(char \*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 129);

    printf("Allocated 129 pages for p5, starting at %d.\n", p5);

    asm\_halt();

}

在memoryManager中，分配新的物理页时会自动调用动态分区算法，如下所示：

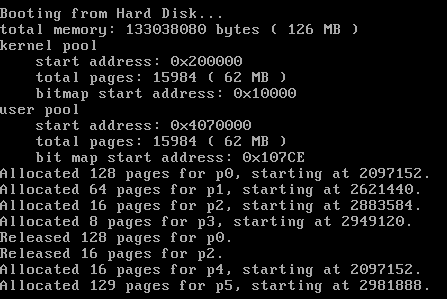
int BitMap::allocate(const int count){

    //return first\_fit(count);

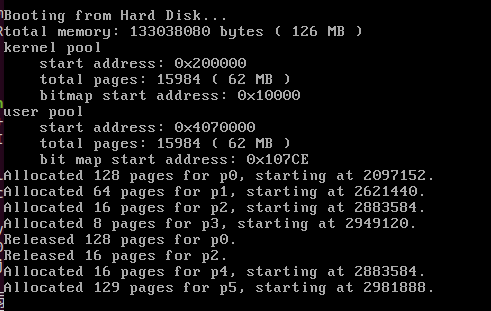
     return best\_fit(count);

}

以下为实验结果。当使用算法为first fit时：



当使用算法为best fit时：



根据页的起始地址可看出，两种动态分配算法符合预期效果。

* **Assignment 3 实现页面置换算法**

进程运行时，若其访问的页面不在内存中而需要将其调入，但内存已经无空闲空间时，就需要从内存中调出一页程序或者数据，送入磁盘的对换区。这就叫做页面置换算法。

我选用的页面置换算法有FIFO算法和LRU算法，这两个算法比较容易实现，原理也较简单。

首先在memory.h中定义新的数据结构，确定存储页面的内存块大小。

struct cell

{

    int index;

    int time;

    char \*p;

};

class MemoryManager{

    ......

    //内存块 大小为3

    cell memory[3];

    ......

}

FIFO：优先淘汰那些最早进入内存的页面，即淘汰在内存中驻留时间最久的页面。该算法实现简单，只需把已调入内存的页面根据先后次序链接成队列，设置一个指针总是指向最老的页面。

void MemoryManager::FIFO(int id, char \*px){

    int i=0;

    int flag=0;

    for(i=0;i<3;i++){

        if(this->memory[i].index==id)

        {

            flag=1;

            break;

        }

    }

    if(flag==0){ //发生缺页

        if(this->memory[2].index!=-1) //已满

    {

        for(i=0;i<2;i++)

        {

            this->memory[i]=this->memory[i+1];

        }

        this->memory[2].index=id;

        this->memory[2].time=0;

        this->memory[2].p=px;

    }

    else{

        i=0;

        while(this->memory[i].index==-1) i++;

        this->memory[i].index=id;

        this->memory[i].time=0;

        this->memory[i].p=px;

        }

    }

    printf("Now have the pages:");

    for(i=0;i<3;i++)

    {

        if(this->memory[i].index==-1) break;

        printf("%d ",this->memory[i].index);

    }

    printf("\n");

}

LRU：选择最近最长时间未访问过的页面予以淘汰，它认为过去一段时间内未访问过的页面，在最近的将来可能也不会被访问。该算法为每个页面设置一个访问字段，用来记录页面自上次被访问以来所经历的时间，淘汰页面时选择现有页面中值最大的予以淘汰。

实现这个算法时主要是根据各cell的time值来判断调换。

void MemoryManager::LRU(int id, char \*px){

    int i=0;

    int flag=0;

    int min=999;

    int id\_min=999;

    for(i=0;i<3;i++){

        if(this->memory[i].index==id) // 查找到，未缺页

        {

            flag=1;

            this->memory[i].time+=1;

            break;

        }

        if(memory[i].time<min)

        {

            min=this->memory[i].time;

            id\_min=i;

        }

    }

    if(flag==0){ //发生缺页

        if(this->memory[2].index!=-1) //已满

    {

        this->memory[id\_min].index=id;

        this->memory[id\_min].time=0;

        this->memory[id\_min].p=px;

    }

    else

        {

        i=0;

        while(this->memory[i].index==-1) i++;

        this->memory[i].index=id;

        this->memory[i].time=0;

        this->memory[i].p=px;

        }

    }

    printf("Now have the pages:");

    for(i=0;i<3;i++)

    {

        if(this->memory[i].index==-1) break;

        printf("%d ",this->memory[i].index);

    }

    printf("\n");

}

然后，在setup.cpp中编写测试用例。

void first\_thread(void \*arg){

    // 都设为内核程序，从内核空间中分配物理页

    char \*p0=(char \*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 16);

    printf("Allocated 16 pages for p0, starting at %d.\n", p0);

    char \*p1=(char \*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 16);

    printf("Allocated 16 pages for p1, starting at %d.\n", p1);

    char \*p2=(char \*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 16);

    printf("Allocated 16 pages for p2, starting at %d.\n", p2);

    char \*p3=(char \*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 16);

    printf("Allocated 16 pages for p3, starting at %d.\n", p3);

    char \*p4=(char \*)memoryManager.allocatePhysicalPages(AddressPoolType::KERNEL, 16);

    printf("Allocated 16 pages for p4, starting at %d.\n", p4);

    char \*list[5]={p0,p1,p2,p3,p4};

    int index[10]={3,6,8,0,1,4,2,5,9,7};

    for(int i=0;i<9;i++){

       int j=index[i]%5; //3,1,3,0,1,4,2,0,4,2

       memoryManager.FIFO(j, list[j]);

       //memoryManager.LRU(j,list[j]);

        }

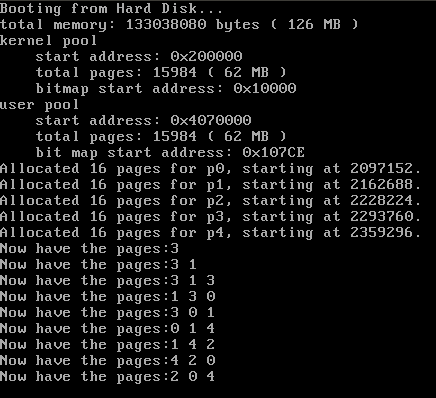
}

    asm\_halt();

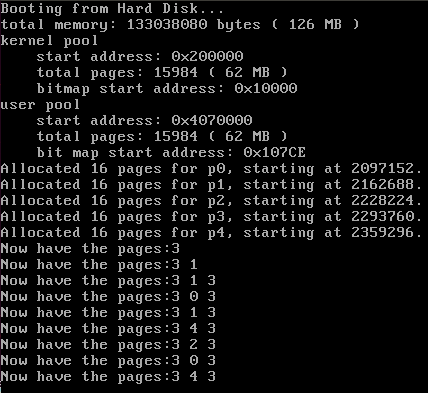
}

根据数组所示，调入的页面顺序应该为：3 1 3 0 1 4 2 0 4 2.

下面是实验结果。首先是使用FIFO算法：



然后是使用LRU算法：



由于我的样例是刚好卡在每个用例出现两次，所以当一开始页面3占据优势地位后就会一直留着，但是通过每次调用中间的出现次数最少的页面可以看出LRU算法也符合预期。

* **Assignment 4 实现虚拟页内存管理**

**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。**

开启了分页机制后，程序使用的地址是虚拟地址，虚拟地址都需要经过二级页表转换成物理地址后CPU才可以正常访问指令和数据。由此而产生的效果是，连续的虚拟地址可以对应不连续的物理地址，这种对应关系由二级页表来维护。

由于需要同时处理两个地址空间（虚拟地址空间和物理地址空间）的内容，进行页内存分配时，需要分别标识虚拟地址的分配状态和物理地址的分配状态，由此而产生了两种地址池——虚拟地址池和物理地址池。当需要进行连续的页内存分配时，我们会做如下工作。

* **从虚拟地址池中分配连续的多个虚拟页。注意，虚拟页之间的虚拟地址是连续的。**
* **从物理地址池中为每一个虚拟页分配相应大小的物理页。**
* **在页目录表和页表中建立虚拟页和物理页之间的对应关系。此时，由于分页机制的存在，物理页的地址可以不连续。CPU的MMU会在程序执行过程中将虚拟地址翻译成物理地址。**

完成了内存管理器的初始化后，接下来是页内存分配的过程。先确定步骤：

* **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 = kernelVrirtual.allocate(count);

    }

    return (start == -1) ? 0 : start;

}

由于没有实现用户进程，此时能够分配页内存的地址池仅内核虚拟地址池。因此，对于其他类型地址池一律返回0，即分配失败。

* **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，这里便不再赘述。我们的物理地址池有两个，用户物理地址池和内核物理地址池，因此在分配物理页的时候也应该区别对待。

* **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;

}

在寻址过程中，由于CPU还是会将物理地址当作虚拟地址来对待，为了访问页目录项和页表项，我们要根据页目录项的序号构造出页目录项的虚拟地址与根据页目录项和页表项的序号构造出页表项的虚拟地址

对于一个虚拟地址，需要构造三部分的信息，页目录号virtual[31:22]，页号virtual[21:12]和偏移地址virtual[11:0]。这里的构造方法在实验七教程中都有，这里便不再赘述。

有了页分配算法后自然也有页内存的释放。这一过程分两个步骤完成。

* **对每一个虚拟页，释放为其分配的物理页。**

int MemoryManager::vaddr2paddr(int vaddr)

{

    int \*pte = (int \*)toPTE(vaddr);

    int page = (\*pte) & 0xfffff000;

    int offset = vaddr & 0xfff;

    return (page + offset);

}

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，取页表项内容的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。

然后我们释放物理页。释放了物理页后，我们就要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。

* **释放虚拟页。**

void MemoryManager::releaseVirtualPages(enum AddressPoolType type, const int vaddr, const int count)

{

    if (type == AddressPoolType::KERNEL)

    {

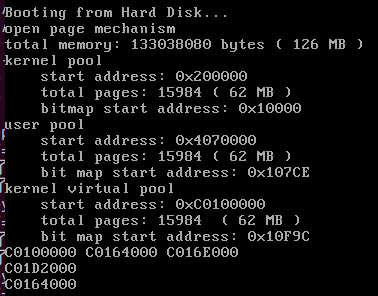
        kernelVirtual.release(vaddr, count);

    }

}

我们现在还没实现用户进程，而每一个用户进程都有自己独立的虚拟地址池，因此这里我们只处理内核虚拟地址池中的地址，等到我们实现了用户进程后再修改这部分的代码。

至此，页内存分配的实现已经完成。以下是代码运行结果。



输出为线程申请的虚拟地址。

**构造测试例子来分析虚拟页内存管理的实现是否存在bug。**

在上述的代码中，对于并发的代码部分没有任何一个锁。这在代码实际运行的过程中可能会触发bug。

于是。在bigmap的连续块与分配连续块之间添加延迟后，可以通过观察若干线程申请的虚拟地址是否产生冲突来判断bug。

在分配资源前，进行延时：

for (int i=0; i<10000;i++)

    for(int j=0;j<10000;j++)

在setup.cpp中编写用例测试：

template<int id>

void thread(void \*arg){

    char \*p = (char \*)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);

    printf("Now thread %d's pages address is %x\n",id,p);

}

void first\_thread(void \*arg)

{

    programManager.executeThread(thread<0>,nullptr,"p0",1);

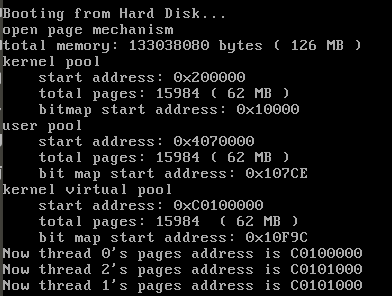
    programManager.executeThread(thread<1>,nullptr,"p1",1);

    programManager.executeThread(thread<2>,nullptr,"p2",1);

    asm\_halt();

}

运行结果如下：



可见在p1与p2的申请中产生了虚拟内存冲突

为了解决这个问题，我们可以直接在线程运行时加锁，要求一次只执行一个线程，这样就不会产生虚拟内存地址申请冲突。

在memory.cpp中：

#include "sync.h"

Semaphore semaphore;

MemoryManager::MemoryManager()

{

    initialize();

    semaphore.initialize(1);

}

void MemoryManager::initialize(){

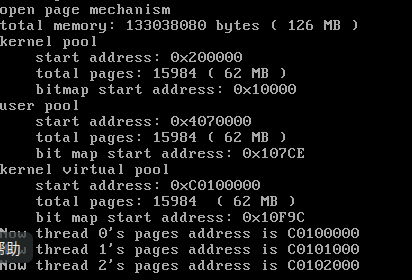
    semaphore.P();

    ...

    semaphore.V();

}

运行结果如下：



1. **总结**

通过本次实验我了解了二级分页机制的具体实验过程，并通过编写动态内存分配算法（first fit，best fit）和虚拟内存管理算法（FIFO、LRU）进一步加深了对内存管理的理解，并且还通过虚拟页内存管理部分了解了虚拟页管理的实现机制，以及如何用信号量来加强算法安全性，收获颇多。