



# 本科生实验报告

实验课程:\_\_\_\_\_操作系统原理实验\_\_\_\_\_

实验名称:\_\_\_\_\_实验 7 内存管理\_\_\_\_\_

专业名称:\_\_\_\_\_计算机科学与技术(超级计算方向) \_\_\_\_\_

学生姓名:\_\_\_\_\_林天皓\_\_\_\_\_

学生学号:\_\_\_\_\_18324034\_\_\_\_\_

实验地点:\_\_\_\_\_

实验成绩:\_\_\_\_\_

报告时间:\_\_\_\_\_2021. 6. 12\_\_\_\_\_

# 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。

## 1. 页表空间的规划和初始化

根据实验代码，首先人为划分一片地址空间用于装载页目录表和页表项，在本实验中起始地址分别为 0x100000 和 0x101000，大小均为 4KB. 共计可装载 1024\*1024 个页表项目，可寻址的空间为 4GB。对应代码为

```
// 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);
```

然后再设置 CPU 的 cr3 寄存器，将页目录表起始地址写入，完成二级页表的设置，此时的访存 CPU 会根据二级页表的查询方式寻址。

```
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
```

根据需求指向不同的物理空间，由于 32 位地址空间的二级页表转换规则，每一个页目录项对应了 4MB 的虚拟空间，为了将 0-1MB 直接映射到物理空间中，设置页目录项与页表项目。

```
for (int i = 0; i < 256; ++i)
{
    page[i] = address | 0x7; // U/S = 1, R/W = 1, P = 1
    address += PAGE_SIZE;
}
directory[0] = ((int)page) | 0x07;
```

然后将 3GB-3GB+1MB 的地址空间映射到 0-1MB 中，使用了 `directory[768] = directory[0]`。

基础的地址空间的划分完成，下面初始化内存的分配。通过内存探查查询现有内存，然后将内存均等的分配在用户页地址与内核页地址中。然后分别初始化用户物理地址，内核物理地址，内核虚拟地址的 bitmap 地址，页面数量，地址起始地址。

虚拟内存的初始化完成了，对于内存的申请与释放过程，将在 assignment4 中再做详细的代码介绍，此处给出运行过程。

## 2.运行测试，运行的主要测试代码如下

```
char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
printf_debug("first address:%x\n", p1);
printf_debug("second address:%x\n", p2);
printf_debug("third address:%x\n", p3);
memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 10);
p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
printf_debug("realloc p2 %x\n", p2);
p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 10);
printf_debug("realloc p2 %x\n", p2);
```

```
QEMU
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F909D0+0
)
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
first address:C0100000
second address:C0164000
third address:C016E000
realloc p2 C01D2000
realloc p2 C0164000
```

图 1-内存分配与释放测试

分析：P1 申请内存 100 页，占用空间为 400K，因此 P2 的地址为 P1+400K 的地址 C0164000。 P2 申请内存 10 页，占用空间 10K，因此 P3 的开始地址为 P2+40K=C016E000。然后释放 P2, P2 再申请 100 个页申请到了 P3 之后的位置。然后 P2 再申请 10 个页申请到了之前 P2 释放内存之后的位置。代表我们的内存释放后能被重新利用。

3.为了进一步了解页表的运行过程我们做以下的补充测试，测试 CPU 对页表的修改。

```
char *p1 = (char*)memoryManager.allocatePages(AddressPoolType::KERNEL,100);
printf_debug("first address:%x\n", p1);
brk();
*p1=10;
brk();
```

debug 运行，并且设置断点在第一个 brk 暂停。

```
QEMU [Stopped]
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F909D0

Booting from Hard Disk...
page directory at 100000
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
connect vaddr:C0100000 phyaddr:200000
pde at FFFFFFFC00
pte at FFF00400
first address:C0100000
```

图 2-运行到断点 brk 函数

这时候我们在 gdb 中尝试查看当前的页表，

```
(gdb) x/1x 0x100000
0x100000: 0x00000000
(gdb)
```

图 3-查看 1MB 中的值

只能查看到 0 是因为这时候我们不再能直接访问物理地址，0x10000 为虚拟地址转换后的地址 page[512]的地址，为 NULL 地址，值为 0。

我们可以根据虚拟地址访问访问到其中的内容，这是因为该部分已经被映射

```
(gdb) x/1x 0xfffffc00
0xfffffc00: 0x00101067
(gdb) x/1x 0xffff00400
0xffff00400: 0x00200007
```

图 3-查看虚拟地址 pde 和 pte 中的值

因此我们如果想要暂时查看内存中的值，需要暂时关闭分页机制，方便对页表的 debug 工作。

```
asm_stop_page_reg:
    mov eax, cr0
    and eax, 0x7fffffff
    mov cr0, eax          ; 置 PG=0, 关闭分页机制
    ret

asm_open_page_reg:
    mov eax, cr0
    or eax, 0x80000000
    mov cr0, eax          ; 置 PG=1, 开启分页机制
    ret
```

更改测试函数，加入关闭分页的断点。

```
char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
printf_debug("first address:%x\n", p1);
asm_stop_page_reg();
brk();
asm_open_page_reg();
```

在 brk 断点后，根据 p1 的虚拟地址 c0100000 对应的页目录表为第 768 项，页目录表地址为  $0x100000 + 768 * 4 = 0x100c00$ 。

```
0x200000: 0x00000000
(gdb) x/1x 0x100c00
0x100c00: 0x00101067
```

图 4-通过 pde 地址的内容查找页表

页表地址为  $0x101000 + 0x100 * 4 = 0x100400$

```
(gdb) x/1x 0x101400
0x101400: 0x00200007
```

图 5-通过页表查找虚拟地址对应的物理地址

根据该页目录表规则，

31	12	11	9	8	7	6	5	4	3	2	1	0
物理页的物理地址 31~12	AVL		G		PAT	D	A	PCD	PWT	US	RW	P

图 6-页表项的结构

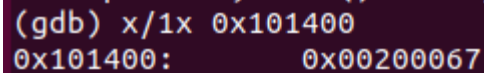
该页面对应的物理地址为  $0x200000 + 4 * 0 = 0x200000$ 。

正好对应了我们申请到的物理地址 0x200000，这就是页表寻址的过程。

接下来通过改变该地址的值测试页表项目中的控制位变化，执行以下代码。

```
asm_open_page_reg();
*p1=10;
asm_stop_page_reg();
brk();
```

再次查看页表中的内容，可见，页表中的脏页位和访问位被设置为 1。



(gdb) x/1x 0x101400  
0x101400: 0x00200067

图 7-页表项中的脏页位和访问位为 1

综上，该实验完成了二级页表的启用与初始化，并完成了内存空间的申请与释放，并通过写入操作观察了页表中脏位和访问位的变化。

## 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。

### 1.测试使用 best-fit 方法。修改 allocate 函数

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

    int index, empty, start;
    int ans = -1;
    int best = 999999999;
    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;
    }
    // 存在大于连续的 count 个资源,
    if (empty >= count && empty - count < best) //best—fit
    {
        best = empty - count;
        ans = start;
    }
}
if (empty >= count && empty - count < best) //best—fit
{
    best = empty - count;
    ans = start;
}
if (ans != -1)
{
    for (int i = 0; i < count; ++i)
    {
        set(start + i, true);
    }
}
return ans;
}

```

寻找大小最靠近所需内存的内存空洞返回地址。

运行测试代码。

```

char *p1 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
printf_debug("1 address:%x\n", p1);
char *p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 80);
printf_debug("2 address:%x\n", p2);
char *p3 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
printf_debug("3 address:%x\n", p3);

```



```

char *p4 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 40);
printf_debug("4 address:%x\n", p4);
char *p5 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 100);
printf_debug("5 address:%x\n", p5);
memoryManager.releasePages(AddressPoolType::KERNEL, (int)p2, 80);
memoryManager.releasePages(AddressPoolType::KERNEL, (int)p4, 40);
p2 = (char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 35);
printf_debug("realloc p2 %x\n", p2);

```

先分别申请 100, 80, 100, 40, 100 的内存空间, 然后释放 80 和 40 的空间构造内存空洞, 然后申请一个 35 的空间测试, 若使用 best\_fit, 则应该被填写进入大小为 40 的内存空洞。

```

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
}
1 address:C0100000
2 address:C0164000
3 address:C01B4000
4 address:C0218000
5 address:C0240000
realloc p2 C0218000

```

图 8-测试使用 best\_fit 方法申请内存

分析: 经过测试, P2 申请到了 40 的内存空洞位置, 符合 best\_fit 方法。

## 2.使用线段树算法完成对物理内存的分配

由于目前使用的内存分配算法的时间复杂度为  $O(mn)$ ,  $m$  为内存总页数,  $n$  为申请页面大小。考虑对时间复杂度的优化, 我们可以对物理页面的申请使用线段树算法。建立 Physical\_AddressPool 类, 并将其中的 resources 改为 SegTree 类

```
class Physical_AddressPool
{
public:
    SegTree resources;
    int startAddress;
```

线段树类的成员基本添加了 leave\_nodes 表示叶子节点的数量，其他与 bitmap 类差距不大。

```
class SegTree
{
public:
    // 被管理的资源个数
    int length;
    int bytes;
    //叶子节点的个数
    int leave_nodes;
    // bitmap 的起始地址
    char *seg;
```

成员函数添加了 void up(int index,bool status);为了处理线段树上传维护的情况，其他接口不变。

线段树的初始化，申请内存，释放内存与 bitmap 实现不同。

### 一.初始化过程如下

1. 创建一个  $\text{len}=2^k$  (大于所需管理的内存数量) 数量的 bitmap, 储存线段树节点, 其中索引为  $\text{len}/2$  以上的节点为叶节点
2. 将  $2^k$  多出来的部分设置为已经占用
3. 维护上一步中应该受到影响的内部节点

```
void SegTree::initialize(char *s, const int length)
{
    this->seg = s;
    this->length = length;
    this->leave_nodes = next_power_of_two(length);
    printf_debug("length %d leave %d\n", length, leave_nodes);
```

```

int bytes = ceil(length, 8);
for(int i=0;i<leave_nodes*2;i++){
    set(i,false);
}
for(int i=length;i<leave_nodes;i++){
    set(i,true);
}
for(int i=leave_nodes;i>0;i--){
    bool sta=get(2*i) && get(2*i+1) && get(i);
    set(i,sta);
}
}

```

二.申请内存过程如下，该算法只支持一个一个申请内存、

1. 从根节点开始，查找 false 状态的子节点，直到索引大于  $\text{len}/2$
2. 标记索引节点为 true,并向上维护线段树。
3. 返回值为  $\text{index}-\text{len}/2$ ,表示是第几个叶节点

```

int SegTree::allocate(const int count)
{
    if(count!=1){
        printf_error("SegmentTree allocator alloc 1 address once\n");
        return -1;
    }
    if (count == 0)
        return -1;
    int index;
    index = 1;
    if(get(index) == 1){
        return -1;
    }
    else{
        while(index < length /2){
            if(!get(index*2)){
                index*=2;// printf_debug("1segtree index%d\n",index);
            }
            else if(!get(index*2+1)){
                index=index*2+1;// printf_debug("2segtree index%d\n",index);
            }
            else{
                printf_error("SegmentTree BAD\n");
            }
        }
    }
}

```

```

        while(1);
    }
}
}
up(index,true);
// printf_debug("segtree alloc index%d\n",index-leave_nodes/2);brk();
return index-leave_nodes/2;
}

```

释放内存的过程如下：

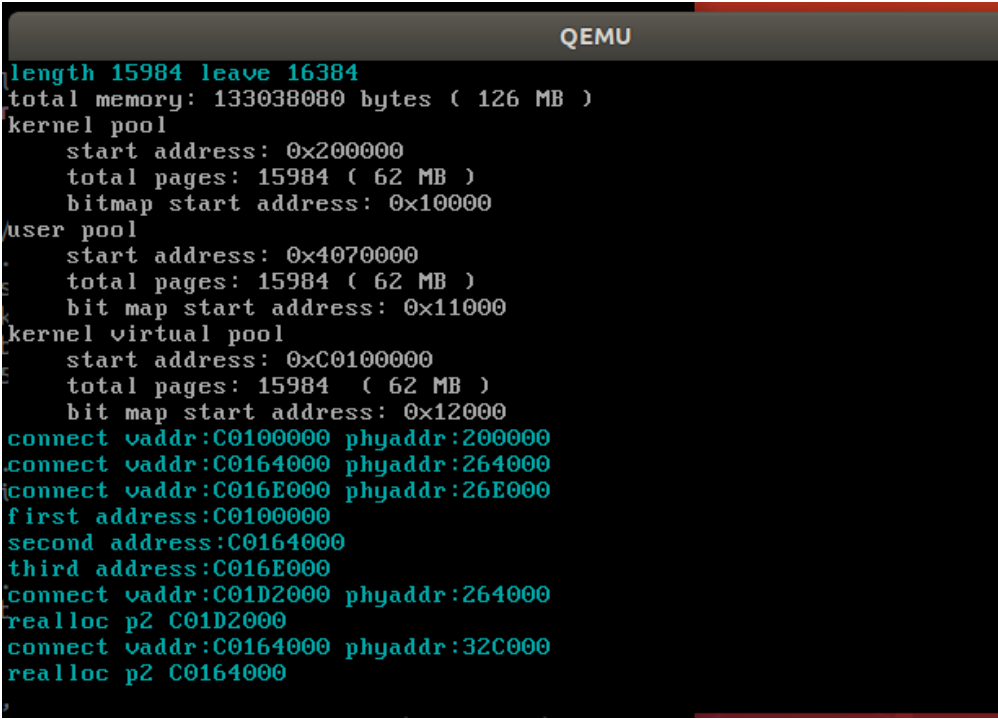
先设置叶子节点 false，然后向上维护线段树状态。

```

void SegTree::release(const int index, const int count)
{
    int node = index+leave_nodes/2;
    up(node,false);}

```

下面运行测试：同样使用在 assignment1 中的测试代码，测试结果如下：



```

QEMU
length 15984 leave 16384
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: 0x11000
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x12000
connect vaddr:C0100000 phyaddr:200000
connect vaddr:C0164000 phyaddr:264000
connect vaddr:C016E000 phyaddr:26E000
first address:C0100000
second address:C0164000
third address:C016E000
connect vaddr:C01D2000 phyaddr:264000
realloc p2 C01D2000
connect vaddr:C0164000 phyaddr:32C000
realloc p2 C0164000

```

图 9-线段树算法分配物理内存

经过对比该地址均和 assignment1 中的一致，我们此时使用线段树分配算法分配了物理内存。

通过更多的 debug 信息，我们可以追踪线段树节点的变化过程。

```
QEMU [Stopped]
lsegtree index32
lsegtree index64
lsegtree index128
lsegtree index256
lsegtree index512
lsegtree index1024
lsegtree index2048
lsegtree index4096
lsegtree index8192
up set index8192 status:1
up set index4096 status:0
up set index2048 status:0
up set index1024 status:0
up set index512 status:0
up set index256 status:0
up set index128 status:0
up set index64 status:0
up set index32 status:0
up set index16 status:0
up set index8 status:0
up set index4 status:0
up set index2 status:0
up set index1 status:0
segtree alloc index0
```

图 10-跟踪线段树算法的维护过程

上面为线段树向下寻找空闲节点的过程，不断寻找左子树，最终申请到了内存地址 0，下面一部分为自底向上维护线段树状态的过程，可见除了 8192 节点以外，其他的祖先节点均有空闲，状态设置为 0，与预期一致。

综上,该部分实现了 **best\_fit** 内存分配算法,并且将物理内存的分配方法由 **BitMap** 实现改写为了基于线段树的实现。

## 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。

由于我们并没有文件系统，所以自然无法将内存中的内容换出到磁盘上，因此这里通过输出将要被换出的页面，并且将页表的有效为置 0，触发缺页故障。

### 一、一个实现过程中遇到的错误

原本想通过分配完所有的内存然后记录所有的内存进行分页，其中分分配页面的函数如下：

```
void first_thread(void *arg)
```

```

{
    // 第 1 个线程不可以返回
    char *p[20000];
    for(int i=0;i<20000;i++){
        p[i]=(char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
        if(p[i]==0){
            printf_debug("i=%d no mem\n",i);
            while(1);
        }
        printf_debug("i=%d\n",i);
    }
}

```

然而这段代码导致分配器出现了错误。

```

i=792
start 57
start 793
connect vaddr:C0139000 phyaddr:519000
i=793
start 57
start 794
connect vaddr:C0139000 phyaddr:51A000
i=794
start 57
start 795
start 795
connect vaddr:C0139000 phyaddr:51B000
i=795

```

图 11-Start 为 bitmap allocator 中返回值，一直为 57 发生错误

一直写入同一个虚拟地址。经过查找，原来是 char \*p[20000]所使用的内存过多，以及溢出了 PCB 所开设的栈空间 4096，导致错误的内存修改破坏了 bitmap，引发故障。

```

(gdb) print &p[19999]
$4 = (char **) 0x24884 <PCB_SET+4068>
(gdb) print &p[0]
$5 = (char **) 0x11008
(gdb) 

```

图 12-使用占栈空间过多，超过 PCB 规定的 4096Byte

## 2.实现页面换出

所以下面对于页面置换的测试，我们通过软件限制只能对 10 页进行测试。

由于我们的操作系统没有实现文件系统并不能将换出的页储存，因此我们的置换

策略只执行换出而不执行换入。

在我们的操作系统中并不方便实现 LRU 策略，因为要想统计到每一次程序所访问的页面是较困难的。而 FIFO 策略和页面的访问完全无关，较容易实现。因此我使用以下的页面换出算法，定时检测页面是否被访问决定是否换出，综合了页面的读取情况和实现难度。



图 12-决定页面是否换出的算法

判断页面是否被访问使用页表中的访问（ACCESSED）位

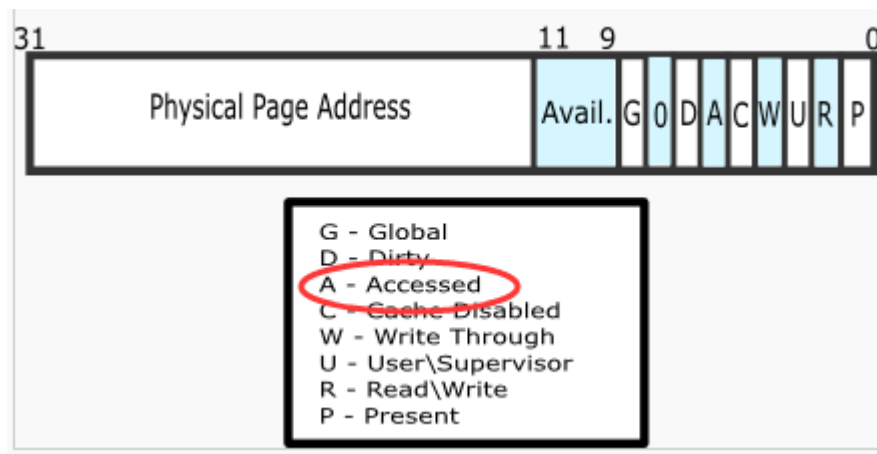


图 13-通过访问位判断页面活跃情况

实现页面换出测试类如下

```
class SwapOutTest
{
public:
    char *p[10]; // 页面
    int count; // 页面数量
    int vaild[10]; // 页面是否被换出
    char *unactive[10]; // 闲置页面
    int unactivecount; // 闲置页面数量
```

```

SwapOutTest(/* args */); //构造函数
void SwapOutCheck(); //检查页面是否被访问
void SwapOut(char *vaddr); //换出页面
bool IsActive(char *vaddr); //判断页面是否被访问过
};

```

实现换出逻辑的方法如下。

```

void SwapOutTest::SwapOutCheck(){
    printf_debug("Unactive vaddr is"); //检测闲置页面
    int check_count=0;
    char *temp[10];
    for(int i=0;i<count;i++){
        if(vaild[i] && !IsActive(p[i])){
            temp[check_count++]=p[i];
            printf_debug("%d ",i);
        }
    }
    printf_info("\n");
    for(int i=0;i<check_count;i++){
        for(int j=0;j<unactivecount;j++){
            if(unactive[j]==temp[i]){ //检测闲置页面是否被闲置两次
                printf_error("Double unactive Swap out vaddr %x\n",temp
[i]);
                SwapOut(temp[i]); //换出
            }
        }
    }
    unactivecount=0;
    for(int i=0;i<check_count;i++){
        unactive[unactivecount++]=temp[i];
    }
}

```

换出页面的具体操作和检测页面是否被访问方法如下。

```

void SwapOutTest::SwapOut(char *vaddr){
    int *pte = (int *)toPTE(vaddr);
    *pte=*pte&(~1); //有效位清零
    for(int i=0;i<count;i++){
        if(vaddr==p[i]){
            vaild[i]=0;
        }
    }
}

bool SwapOutTest::IsActive(char *vaddr){

```



```

    int *pte = (int *)toPTE(vaddr);
    //printf_info("pte = %x\n",*pte);
    bool ans=(*pte)&(1<<5);//获取访问位
    *pte=*pte&(~(3<<5));//访问位和写入位清零
    return ans;//返回访问位
}

```

运行的时候，通过在主程序中创建一个进程，该进程执行换出检测的操作。进程执行的代码如下。

```

void SwapOutTester_thread(void *arg)
{
    while(1){
        int t=999999999;
        while (t--);
        printf_info("\n");
        SwapOutTester.SwapOutCheck();
        printf_info("access addr");
    }
}

```

构造的测试如下，通过手动构造的随机访问对申请到的页面中进行写操作。

```

int random_series[100]={6,2,5,3,8,5,3,6,9,1,3……此处省略 };
void first_thread(void *arg)
{
    int k=0;
    for(int i=0;i<10;i++){
        SwapOutTester.p[i]=(char *)memoryManager.allocatePages(AddressPoolType::KERNEL, 1);
        SwapOutTester.vaild[i]=1;
        SwapOutTester.count++;
    }
    printf_info("access addr ");
    for(int i=0;i<500;i++){
        int t=999999999;
        while (t--);
        int now=random_series[i%100];
        if(SwapOutTester.vaild[now]==1){
            *(SwapOutTester.p[random_series[i]])=k++;
            printf_info("%x ",random_series[i]);
        }
    }
    printf_info("end");
    asm_halt();
}

```

```
}
```

### 3.运行测试

```
QEMU
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
connect vaddr:C0100000 phyaddr:200000
connect vaddr:C0101000 phyaddr:201000
connect vaddr:C0102000 phyaddr:202000
connect vaddr:C0103000 phyaddr:203000
connect vaddr:C0104000 phyaddr:204000
connect vaddr:C0105000 phyaddr:205000
connect vaddr:C0106000 phyaddr:206000
connect vaddr:C0107000 phyaddr:207000
connect vaddr:C0108000 phyaddr:208000
connect vaddr:C0109000 phyaddr:209000
access addr 9 8 0
Unactive vaddr is 1 2 3 4 5 6 7
access addr 3 2 5 6
Unactive vaddr is 0 1 4 7 8 9
Double unactive Swap out vaddr C0101000
Double unactive Swap out vaddr C0104000
Double unactive Swap out vaddr C0107000
```

图 14- 页面换出计算过程与换出决定

分析：可见在第一次检测的时候我们发现 1, 2, 3, 4, 5, 6, 7 页面没有被访问过，标记为闲置页面，在第二次检测时候，发现 0, 1, 4, 7, 8, 9 页面没有被访问过，其中 1, 4, 7 两次都没有被访问过，因此被换出，符合预期现象。

综上，该部分实现了一个基于页面最近是否被闲置，来选择页面的换出的算法。

## 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。

开启分页机制代码运行已经在 assignment1 中完成，此处通过代码介绍分页机制

的内存申请和释放的过程。

### 1.申请过程:

首先申请一片连续的  $n$  个虚拟地址，然后申请  $n$  个物理地址，然后写入页表。

申请的过程位查询对应的 bitmap 有没有被使用，找到联系的  $n$  个 bitmap 为 0 的空间即为一片未被占用的内存。

Bitmap 的查询算法如下

```
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;
}

```

### 写入页表过程如下

分别计算物理地址对应的页目录项和页表项，如果页表不存在，需要分配一个页表，使页目录项指向页表，再将页表指向物理地址。

```

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;
}

```

## 2.释放内存空间过程

将物理页表与虚拟页表中的内容的有效位标记为 0,, 这里为了方便直接将整个页表设置为 0。

还需要重置 bitmap 中的对应标记。

```

void MemoryManager::releasePages(enum AddressPoolType type, const int v
irtualAddress, 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);
}

```

### 3. 页面内存管理多线程竞争 bug

上述算法在单线程运行下能够完成对内存空间的申请和释放,但是在多线程环境下,很容易出现对内存 bitmap 的竞争访问现象,导致 bitmap 返回了已经被占用的地址,出现错误,使用的测试如下。

```

void first_thread(void *arg)
{
    for(int i=0;i<2;i++){
        char *p1 = (char *)memoryManager.allocatePages(AddressPoolType:
:KERNEL, 100);
        printf_debug("first_thread alloc p:%x\n",p1);
    }
    asm_halt();
}

void second_thread(void *arg)
{
    for(int i=0;i<2;i++){
        char *p1 = (char *)memoryManager.allocatePages(AddressPoolType:
:KERNEL, 100);
        printf_debug("second_thread alloc p:%x\n",p1);
    }
    asm_halt();
}

```

首先使用单线程申请页表，申请大小为 100 的页表 4 次，得到的地址如下：

```
QEMU
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07

Booting from Hard Disk...
page directory at 100000
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
first_thread alloc p:C0100000
first_thread alloc p:C0164000
first_thread alloc p:C01C8000
first_thread alloc p:C022C000
```

图 14 – 单线程内存申请 4 次每次 100 页

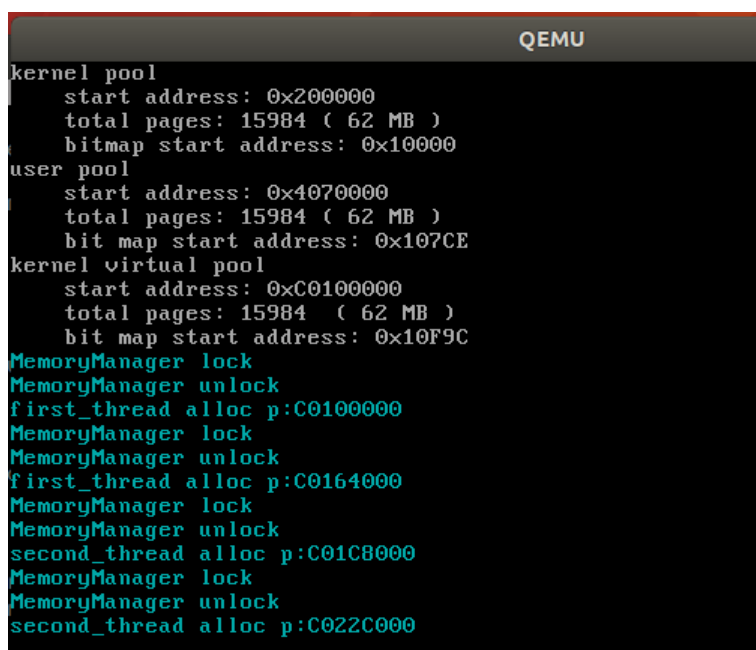
再建立一个线程，也申请大小为 100 页的内存，由于竞争现象的出现，导致了两个线程获取了同样的内存地址，这是一个不可接受的 bug。

```
QEMU
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F

Booting from Hard Disk...
page directory at 100000
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
first_thread alloc p:C0100000
second_thread alloc p:C0164000
first_thread alloc p:C01C8000
second_thread alloc p:C01C8000
```

图 15 – 双线程每线程内存申请 2 次每次 100 页，竞争

为了防止这种现象的发生，我们需要在内存的分配过程中加锁保证互斥。加入自旋锁后再次运行测试。



```
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
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0x10F9C
MemoryManager lock
MemoryManager unlock
first_thread alloc p:C0100000
MemoryManager lock
MemoryManager unlock
first_thread alloc p:C0164000
MemoryManager lock
MemoryManager unlock
second_thread alloc p:C01C8000
MemoryManager lock
MemoryManager unlock
second_thread alloc p:C022C000
```

图 15 – 双线程每线程内存申请 2 次每次 100 页，加锁

可见此时已经可以保证内存申请的互斥，可以运用在多线程环境中了。

综上，该部分完成了对实现页面申请与释放过程代码解读，并且通过添加自旋锁解决了内存申请在多线程环境中的竞争 bug 现象。

## 实验感想

本次实验了解了 x86 的二级页表机制，并且在该机制下实现了页表的初始化以及内存的申请和释放功能。

在 assignemnt1 中，实现了开启分页并初始化，并通过一个具体的例子阐述了虚拟地址是如何被转换为物理地址。通过内存管理，我们可以超越内存容量完成寻址功能，不再需要精密的内存规划，而是通过 MMU 动态申请与释放内存。

在 assignment2 中，通过简单的改写将原本的 first\_fit 改为 best\_fit 内存分配

算法。并且由于对物理地址的页都是逐个申请，使用了线段树算法进行内存分配提高了分配的速度。

在 assignment3 中，由于没有辅存无法实现换入，使用了一种基于页面是否活跃的算法决定换出的页面。其实在这部分实现中仍然有问题，CPU 会在访问虚拟地址时将页表的 A 标志设置为 1，然后我们的操作系统就可以得知该页面被访问过，但是被操作系统手动清零后，经过测试，再次访问该页面，CPU 不再改变这一位，导致换出算法不能继续正确执行，没有查询到原因，也没有找到有关 linux 如何得知 CPU 访问什么页表来实现 LRU 的方法。

在 assignment4 中，对内存申请和释放的代码做了阐述，并且发现了原本的内存申请方法在多线程运行下的竞争现象，并最终通过自旋锁修复 bug。想查询有没有不如此粗暴的方式解决这种问题，网络上已经很难找到相关资料。

## 参考资料

[https://wiki.osdev.org/Detecting\\_Memory\\_\(x86\)](https://wiki.osdev.org/Detecting_Memory_(x86))

[https://wiki.osdev.org/Page\\_Frame\\_Allocation](https://wiki.osdev.org/Page_Frame_Allocation)

<http://www.cs.albany.edu/~sdc/CSI500/Fall10/Classes/C22/intelpaging83-94.pdf>

<https://zhuanlan.zhihu.com/p/350697474>

<https://wiki.osdev.org/Paging>

<https://blog.csdn.net/zhoutaopower/article/details/93429620>