



本科生实验报告

实验课程：_____操作系统_____

实验名称：_____lab9_____

专业名称：_____计算机科学与技术_____

学生姓名：_____叶彦烈_____

学生学号：_____21307074_____

实验地点：_____VMware 虚拟机_____

实验成绩：_____

报告时间：_____2023/6/24_____

1. 实验要求

1) 虚拟内存的完善，实现页在磁盘的换入换出；

2. 实验过程

由于本次实验综合性比较强，而且一些默认性的 cpu 规则比较多，我先将自己从网上大量学习和搜索理解到的问题知识罗列。

首先，这个代码在之前实验完成置换算法的基础上进行修改。

在看到这个任务时，我想，上一次实验我模拟了一个情景，就是在一页一页申请物理页，如果有一页不成功时，我会将这个任务认为失败，然后释放之前申请的所有与虚拟页链接成功的物理页，并且把页表项 pte 置为 0。那么如果我要实现换入换出的话，那么之前成功连接的就要保留下来。

所以，我实现的页面置换在下面情况下会发生：

访问到对应进程的页但是不在内存中，如果在磁盘里的话需要换入并踢出页，如果不在磁盘里那么要申请新的一页。

这时候有了第一个问题，怎么知道读到了虚拟地址后就知道它不在内存里（也就是缺页）呢？

Physical Page	Rsv	G	D	A	P	W	U/S	P
Address							R/W	

上述是 pte 的具体位的意思，可以看到 P 位代表是否在物理内存中。我们可以看到，只要没有进行虚拟页和物理页连接，即下列代码的话，这时候访问虚拟地址系统是会判断到不在物理内存中的。

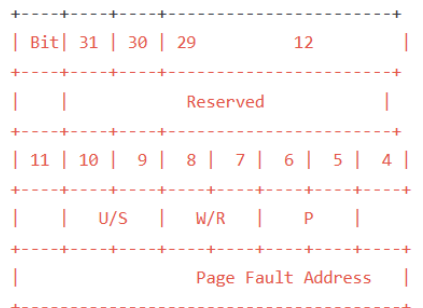
```
// 使页表项指向物理页
*pte = physicalPageAddress | 0x7;
printf("Connecting VP: 0x%x with PP: 0x%x\n", virtualAddress, physicalPageAddress, *pte);
```

第一个问题已经解决，第二个问题来了，知道不在物理内存会怎么办呢？

经过查阅 intel 手册以及网上 Ucore 的代码教程，我知道了系统会默认进行 int14 的缺页中断来处理这个缺页。

接下来是缺页中断的过程：

将当前的 EFLAGS（标志寄存器）当前的 CS（代码段寄存器）当前的 EIP（指令指针寄存器）保存到栈中。将错误码（errorCode）压栈。错误码提供了关于出现缺页中断的具体原因和类型的信息。同时 CR2 寄存器保存着引发缺页的线性地址。下图是 errorCode 具体含义



这是上面缺页中断的默认信息，然后我们要怎么处理呢？

思考了许久，其实就是自己完善好缺页中断的代码，首先是关闭中断，然后保护现场，跳出相应的中断处理程序，并且根据 cr2 保存的错误地址来确定哪一个页需要给他找到物理页去分配，然后接下来是找到分配的物理页，如果物理页不够，需要用页面置换算法进行换入换出。

刚刚说到，缺页可以是该虚拟页没有对物理页或者该页在磁盘中，对于没有物理页，很简答直接分配一个并连接就好，对于在磁盘中的话，怎么知道哪一个磁盘的页对应这个虚拟页的，我去网上搜索后，发现一种方法，是根据 PTE 保存，因为我们可以看到，在 PTE 最后一位置零表示不在物理内存之后，我们可以用前面的位数进行保存对应的磁盘的位置。我们便可以找到这个对应磁盘中的位置进行换入换出。



还有一个问题：就如理论课所说，我们还需要定义一个磁盘的交换

空间的管理，于是我借用了位图的设计，来定义了一个磁盘的交换空间管理器，用于管理是否分配的情况。

同时，在搜索学习之后还发现一点前人踩过的坑，就是要在更新

pte 后刷新 TLB。刷新 TLB 是因为 TLB 中存储的是虚拟地址到物理地址的映射关系。TLB 是一个高速缓存，用于加速页面的转换过程。每当修改了 PTE 时，可能会导致旧的 TLB 项与新的页表不一致，从而引发错误的地址转换结果。接下来是代码部分的介绍。

3. 关键代码

1. 对于 Assignment1 系统调用代码见 src/1 中的实现

1) 缺页中断的初始化以及程序的实现

可以注意到传进来的参数有压入栈的错误码以及 cr2 寄存器的地址以及判断是用户态还是内核态缺页。对于在磁盘的页，直接换

入函数即可，对于是新分配的虚拟页，则需要重新分配一个物理页与其连接。

```
// 中断处理函数
extern "C" void c_pageFault_handler(uint32 pageFault_Code, uint32
pageFault_Addr, uint32 OSMod)
{
    pageFault_Addr = pageFault_Addr & 0xfffff000;
    printf("[Page Fault] Catch the fault page 0x%x\n",
pageFault_Addr);
    bool inKer_Flag = ((OSMod & 3) == 0); // in kernel or in user
    bool inDis_Flag =
((*(int*)memoryManager.toPTE(pageFault_Addr)) & 2) == 2;
    enum AddressPoolType type = OSMod == 1 ?
AddressPoolType::KERNEL : AddressPoolType::USER;
    if(inDis_Flag)
    {
        memoryManager.swapIn(pageFault_Addr, inKer_Flag);
        return;
    }

    int physicalPageAddress =
memoryManager.allocatePhysicalPages(type, 1);
    // int physicalPageAddress =
allocatePhysicalPages(AddressPoolType::USER, 1);
    if (physicalPageAddress == 0)
    {
        /*
        not enough physical pages
        find one page swapout
        */
        int swapOutPage = 0;
        if(type == 1)
        {
            //kernel
            swapOutPage = memoryManager.kernelVirtual.Out();
        }
        else
        {
            //user
            swapOutPage = programManager.running-
>userVirtual.Out();
        }

        memoryManager.swapOut(swapOutPage, type);
        physicalPageAddress =
memoryManager.allocatePhysicalPages(type, 1);
    }
}
```

```

    }
    memoryManager.connectPhysicalVirtualPage((int)pageFault_Addr,
physicalPageAddress);
    asm_update_tlb();
}

```

2) Swapin 和 Swapout 函数

换入函数中，根据错误的地址转换 PTE 得到磁盘的位置，如果此时物理内存有空闲，则直接换入，否则需要找到换出的页换出。

```

int MemoryManager::swapOut(uint32 vaddr, int mod)
{
    enum AddressPoolType type = mod == 1 ? AddressPoolType::KERNEL :
AddressPoolType:: USER;
    int *pte = (int *)toPTE(vaddr);
    int index = swapResources.allocate(8); //one page equal eight
sections
    if (index == -1)
    {
        printf("Swapping Out Failed ,due to disk swap space is not
enough\n");
        return -1;
    }
    if (mod == 1){
        printf("[Mod Kernel]Swapping out Page: 0x%x to Sector %d\n",
vaddr, index + beginSector);
    }
    else{
        printf("[Mod User]Swapping out Page: 0x%x to Sector %d\n",
vaddr, index + beginSector);
    }
    for (int i = 0; i < 8; i++)
    {
        char *ptr = (char *)vaddr + i * 512;
        Disk::write(index + i + beginSector, (void *)ptr);
    }
    releasePhysicalPages(type, vaddr2paddr(vaddr), 1);
    if (type == AddressPoolType::KERNEL)
    {
        int index = (vaddr - kernelVirtual.startAddress) / PAGE_SIZE;
        int i = 0;
        for (i = 0; i < MAX_PAGES && kernelVirtual.lruindex[i] !=
index; i++);
        kernelVirtual.lruindex[i] = -1;
    }
}

```

```

        else if (type == AddressPoolType::USER)
        {
            int index = (vaddr - (programManager.running)-
>userVirtual.startAddress) / PAGE_SIZE;
            int i = 0;
            for (i = 0; i < MAX_PAGES && (programManager.running)-
>userVirtual.lruindex[i] != index; i++);
            printf("Realease index %d page\n",index);
            (programManager.running)->userVirtual.lruindex[i] = -1;
        }
        //store pte and significance bit
        *pte = (index << 20) + 2;
        // 刷新 TLB
        asm_update_tlb();
        return 0;
    }
}

int MemoryManager::swapIn(uint32 vaddr, int mod)
{
    enum AddressPoolType type = mod == 1 ? AddressPoolType::KERNEL :
AddressPoolType:: USER;
    int *pte = (int *)toPTE(vaddr);
    int index = (*pte) >> 20;
    printf("[Mod %d]Swapping in Page: 0x%x from Sector %d\n",!mod,
vaddr, index + beginSector);
    int physicalPageAddress = allocatePhysicalPages(type, 1);
    // int physicalPageAddress =
allocatePhysicalPages(AddressPoolType::USER, 1);
    if (physicalPageAddress == 0)
    {
        /*
        not enough physical pages
        find one page swapout
        */
        int swapOutPage = 0;
        if(type == 1)
        {
            // kernel
            swapOutPage = memoryManager.kernelVirtual.Out();
        }
        else
        {
            // user
            swapOutPage = programManager.running->userVirtual.Out();
        }
        swapOut(swapOutPage, type);
        physicalPageAddress = allocatePhysicalPages(type, 1);
    }
    connectPhysicalVirtualPage((int)vaddr, physicalPageAddress);
}

```

```

    for (int i = 0; i < 8; i++)
    {
        char *ptr = (char *)vaddr + i * 512;
        Disk::read(index + i + beginSector, (void *)ptr);
    }
    swapResources.release(index, 8);
    if (type == AddressPoolType::KERNEL)
    {
        int index = (vaddr - kernelVirtual.startAddress) / PAGE_SIZE;
        int i = 0;
        for (i = 0; i < MAX_PAGES && kernelVirtual.lruindex[i] != -1;
i++);
        kernelVirtual.lruindex[i] = index;
    }
    else if (type == AddressPoolType::USER)
    {
        int index = (vaddr - (programManager.running)-
>userVirtual.startAddress) / PAGE_SIZE;
        int i = 0;
        for (i = 0; i < MAX_PAGES && (programManager.running)-
>userVirtual.lruindex[i] != -1; i++);
        (programManager.running)->userVirtual.lruindex[i] = index;
    }
    // 刷新 TLB
    asm_update_tlb();
}

```

磁盘的通过端口读写的函数，详见 Disk.h

```

static void write(int start, void *buf)
{
    byte *buffer = (byte *)buf;
    int temp = 0;
    int high, low;

    // 请求硬盘写入一个扇区，等待硬盘就绪
    bool flag = waitForDisk(start, 1, 0x30);
    if (!flag)
    {
        return;
    }

    for (int i = 0; i < SECTOR_SIZE; i += 2)
    {
        high = buffer[i+1];
        high = high & 0xff;
        high = high << 8;

        low = buffer[i];
    }
}

```

```

        low = low & 0xff;

        temp = high | low;

        // 每次需要向 0x1f0 写入一个字 (2 个字节)
        asm_outw_port(0x1f0, temp);
        // 硬盘的状态可以从 0x1f7 读入
        // 最低位是 err 位
        asm_in_port(0x1f7, (uint8 *)&temp);

        if (temp & 0x1)
        {
            asm_in_port(0x1f1, (uint8 *)&temp);
            printf("disk error, error code: %x\n", (temp & 0xff));
            return;
        }
    }

    busyWait();
}

// 以扇区为单位读出, 每次读取一个扇区
// 参数 start: 起始逻辑扇区号
// 参数 buf: 读出的数据写入的起始地址
static void read(int start, void *buf)
{
    byte *buffer = (byte *)buf;
    int temp;

    // 请求硬盘读出一个扇区, 等待硬盘就绪
    bool flag = waitForDisk(start, 1, 0x20);
    if (!flag)
    {
        return;
    }

    for (int i = 0; i < SECTOR_SIZE; i += 2)
    {
        // 从 0x1f0 读入一个字
        asm_inw_port(0x1f0, buffer + i);
        // 硬盘的状态可以从 0x1f7 读入
        // 最低位是 err 位
        asm_in_port(0x1f7, (uint8 *)&temp);
        if (temp & 0x1)
        {
            asm_in_port(0x1f1, (uint8 *)&temp);
            printf("disk error, error code: %x\n", (temp & 0xff));
            return;
        }
    }
}

```



```

    }
}

    busyWait();
}

```

汇编代码辅助函数，包括刷新 TLB 函数，中断处理函数

asm_pageFault_handler:

```

    cli ;incase time interupt stop us

    push ebp
    mov ebp, esp
    pushad
    push ds
    push es
    push fs
    push gs

    mov eax, cr2
    mov ebx, [ebp + 4]
    mov ecx, [ebp + 4 * 3]
    push ecx
    push eax
    push ebx
    call c_pageFault_handler
    pop ebx
    pop eax
    pop ecx

    pop gs
    pop fs
    pop es
    pop ds
    popad
    pop ebp
    add esp, 4 ;remove the error code from interupt stack
    sti
    iret

```

; void asm_update_tlb();

asm_update_tlb:

```

    cli
    push eax
    mov eax, cr3
    mov cr3, eax
    pop eax

```

```
sti
ret
```

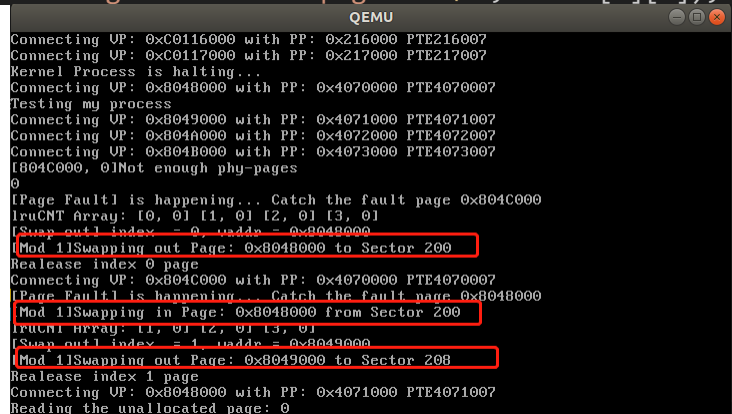
4. 测试代码和实验结果解释

1. 缺页中断的测试代码

可以看到，首先为了测试方便，我将用户池规定为 4 页的物理帧，于是在第一个进程分配时，首先会有一页分配给特定的栈结构，然后再在程序中手动分配 4 页，会发现最后一页应该是只有虚拟页但是没有对应的物理页的，所以手动进行访问，发现进行缺页中断，并且换出页腾出物理空间，结果如图。

```
printf("Testing my process\n");
enum AddressPoolType type = AddressPoolType::USER;
char* buffer[4];
for (int i = 0; i < 4; i++)
{
    buffer[i] = (char*)memoryManager.allocatePages(USER, 1);
}

printf("%x\n",*(int*)memoryManager.toPTE((int)&buffer[3][0]));
printf("Reading unallocated page: %d\n",buffer[3][0]);
```



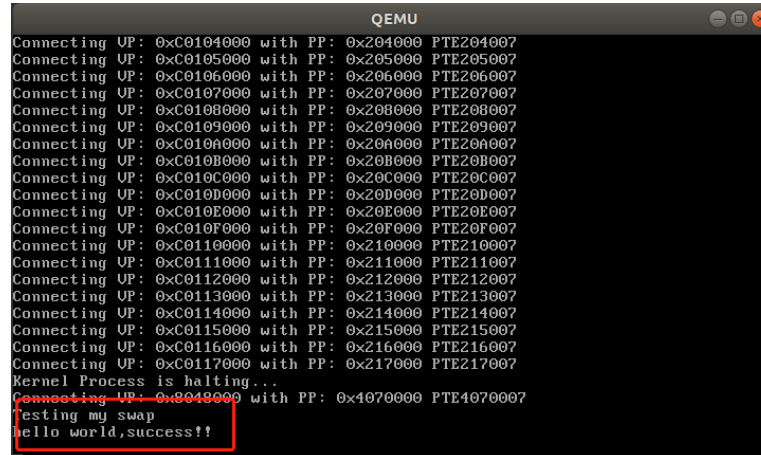
```
QEMU
Connecting UP: 0xC0116000 with PP: 0x216000 PTE216007
Connecting UP: 0xC0117000 with PP: 0x217000 PTE217007
Kernel Process is halting...
Connecting UP: 0x8048000 with PP: 0x4070000 PTE4070007
Testing my process
Connecting UP: 0x8049000 with PP: 0x4071000 PTE4071007
Connecting UP: 0x804A000 with PP: 0x4072000 PTE4072007
Connecting UP: 0x804B000 with PP: 0x4073000 PTE4073007
[804C000, 0]Not enough phy-pages
0
[Page Fault] is happening... Catch the fault page 0x804C000
truCNT Array: [0, 0] [1, 0] [2, 0] [3, 0]
[Swap out] index = 0, address = 0x8048000
[Mod 1]Swapping out Page: 0x8048000 to Sector 200
Release Index 0 page
Connecting UP: 0x804C000 with PP: 0x4070000 PTE4070007
[Page Fault] is happening... Catch the fault page 0x8048000
[Mod 1]Swapping in Page: 0x8048000 from Sector 200
truCNT Array: [1, 0] [2, 0] [3, 0]
[Swap out] index = 1, address = 0x8049000
[Mod 1]Swapping out Page: 0x8049000 to Sector 200
Release Index 1 page
Connecting UP: 0x8048000 with PP: 0x4071000 PTE4071007
Reading the unallocated page: 0
```

2. 测试磁盘读写是否正确

```
printf("Testing my swap\n");
enum AddressPoolType type = AddressPoolType::USER;
char buffer[512] = "hello world";
char newbuffer[512];
Disk::write(200,buffer);
Disk::read(200,newbuffer);
bool flag = 1;
for(int i = 0;i < 512;i++){
    if(buffer[i] != newbuffer[i]){
        flag = 0;
        break;
    }
}
```

```
if(flag){
    printf("%s,success!!\n",buffer);
}
```

这个测试进行写入读出然后对比前后是否一样，测试完发现成功通过测试，说明程序正确。



```
QEMU
Connecting UP: 0xC0104000 with PP: 0x204000 PTE204007
Connecting UP: 0xC0105000 with PP: 0x205000 PTE205007
Connecting UP: 0xC0106000 with PP: 0x206000 PTE206007
Connecting UP: 0xC0107000 with PP: 0x207000 PTE207007
Connecting UP: 0xC0108000 with PP: 0x208000 PTE208007
Connecting UP: 0xC0109000 with PP: 0x209000 PTE209007
Connecting UP: 0xC010A000 with PP: 0x20A000 PTE20A007
Connecting UP: 0xC010B000 with PP: 0x20B000 PTE20B007
Connecting UP: 0xC010C000 with PP: 0x20C000 PTE20C007
Connecting UP: 0xC010D000 with PP: 0x20D000 PTE20D007
Connecting UP: 0xC010E000 with PP: 0x20E000 PTE20E007
Connecting UP: 0xC010F000 with PP: 0x20F000 PTE20F007
Connecting UP: 0xC0110000 with PP: 0x210000 PTE210007
Connecting UP: 0xC0111000 with PP: 0x211000 PTE211007
Connecting UP: 0xC0112000 with PP: 0x212000 PTE212007
Connecting UP: 0xC0113000 with PP: 0x213000 PTE213007
Connecting UP: 0xC0114000 with PP: 0x214000 PTE214007
Connecting UP: 0xC0115000 with PP: 0x215000 PTE215007
Connecting UP: 0xC0116000 with PP: 0x216000 PTE216007
Connecting UP: 0xC0117000 with PP: 0x217000 PTE217007
Kernel Process is halting...
Connecting UP: 0x0040000 with PP: 0x4070000 PTE4070007
Testing my swap
hello world,success!!
```

5. 总结

这次实验算是本课程最后一个实验，从难度上来说，综合性很高，更难得是从 0-1 的跨越。虽然第一部分的实验思路在写这个实验报告的时候很顺利，但是其实中间的摸索过程是十分艰辛的。如果不是查阅以及学习网上前辈们的经验代码，不可能知道是 int14 中断，也不会知道缺页中断 cpu 的默认处理，这些默认规范是十分棘手的部分，对于错误码，错误信息的处理，以及添加缺页中断处理程序，汇编代码的难度写起来是十分大的，但是一旦摸索出来，整个实验的思路其实是十分明朗的。

在这次实验中真实模拟了页面置换，是对前面实验的进一步更新与完善，初步实现了自己简单的操作系统。可以看到前人为我们留下的经验十分宝贵，同时也希望自己能够在接下来的路中创新，再接再厉！！