虚拟内存实习报告

杨东升 1400012898

2017.3.27

目录

内容一：总体概述

本实验主要是对虚拟内存管理机制进行熟悉和实践。虚拟内存管理涉及到的项目有缺页异常处理、TLB 表的查询和替换、页表的查询、页面交换以及地址翻译等等。首先我们要理解 Nachos 源代码的异常处理机制和地址翻译机制，以及 TLB 和页表的建立。然后要注意 Nachos 是如何运行用户程序的。之后就是对 Nachos 源代码进行修改，实现 TLB 的查询和替换、页表的查询和交换等等。这次 lab 可以加强对整个虚拟内存管理机制的理解，了解具体的工作方式。

内容二：任务完成情况

任务完成列表（Y/N）

	Exercise1	Exercise2	Exercise3	Challenge1	Challenge2
第一部分	Y	Y	Y		
第二部分	Y	Y	Y		
第三部分	Y			Y	Y

（总之就是都完成了）

具体 Exercise 的完成情况

第一部分：

Exercise1: 源代码阅读

1. 阅读 code/userprog/progtest.cc，着重理解 nachos 执行用户程序的过程，以及该过程中与内存管理相关的要点。

执行用户程序的过程，首先是调用 StartProcess 函数。该函数打开通过 MIPS 体系结构编译的可执行文件，然后为进程申请并初始化一块地址空间，再初始化寄存器，装载页表，最后运行用户程序。运行用户程序后不会返回，因为进程在结束时会调用系统调用中的停机函数，执行回收空间等任务。

该过程中与内存管理相关的要点是创建 AddrSpace 对象。AddrSpace 构造函数首先读取可执行文件头，得到程序所需空间大小，然后上取整为页大小的整数倍。然后创建地址翻译表，并初始化是否有效，是否使用，是否修改等标志。最后把每一页的内容置零，并复制代

码和数据到相应的位置。

2. 阅读 code/machine 目录下的 machine.h(cc)，translate.h(cc)文件和 code/userprog 目录下的 exception.h(cc)，理解当前 Nachos 系统所采用的 TLB 机制和地址转换机制。

machine.h 定义了 nachos 虚拟机的运行环境。关于虚拟内存机制，它定义了 TLBSize 的宏，默认为 4，表示 TLB 中可以存放 4 个页表项；定义了 PageFaultException 异常；在 Machine 类中声明了 tlb 和 pageTable 指针，分别用于 TLB 地址翻译和线性地址翻译。

在 machine.cc 文件中，构造函数中建立了一个 TLB，valid 字段设置为 false；另外有 RaiseException 函数，用来为异常处理函数 ExceptionHandler 设置机器状态。在 exception.cc 中定义了 ExceptionHandler，用于处理产生的异常。

在 translate.h 文件中，定义了页表项的类，其中定义了实地址、虚地址、有效位、只读位、use（使用过）、 dirty（被修改）。

在 translate.cc 文件中，定义了 ReadMem、WriteMem 和 Translate 函数，分别用来读写内存，和地址转换。 其中 ReadMem 和 WriteMem 函数都需要 Translate 函数来实现地址转换。地址转换函数 Translate 完成的步骤主要有：检查对齐，即查看地址是否 2、4 字节对齐有效；计算虚拟页号和偏移量；在 TLB 或者页表中找到对应的页表表项；检查是否只读、超出地址范围；如果页表合法，就返回实地址。

地址转换机制的核心操作是 translate 中的翻译操作。Nachos 系统默认是只允许 TLB 和页表二者当且仅当一个存在，因此如果没有 TLB 就直接用虚拟页号作为索引找到页表的对应表项。如果有 TLB，就遍历 TLB 所有表项，看哪个表项符合，如果没有或者无效就返回 PageFaultException。

Exercise2: TLB MISS 异常处理

修改 code/userprog 目录下 exception.cc 中的 ExceptionHandler 函数，使得 Nachos 系统可以对 TLB 异常进行处理（TLB 异常时，Nachos 系统会抛出 PageFaultException，详见 code/machine/machine.cc）。

TLB 异常出现，表示 TLB 中没有对应的页表项，但是内容有可能存在于内存中。为了处理 TLB 的异常，就需要从页表中找到对应的页表项。如果该页表项有效，就让它导入 TLB，然后重新进行 TLB 的查找或者直接得到实地址。

为了区别 TLB 异常和页表产生的缺页异常，我在这里添加了一个新的异常类型 TLBMissException，表示在 TLB 中找不到对应页表项。

当 TLB 页表项缺失时，Translate 函数会返回一个 TLBMissException，ReadMem 和

WriteMem 函数会将造成异常的地址存放在 BadVAddrReg 寄存器中，并抛出这个异常。我需要在 ExceptionHandler 函数中加入处理这个异常的代码。具体的功能就是在页表中找到对应的页表项，如果无效就抛出缺页异常，否则就将其载入 TLB 中。TLB 替换策略采用轮转策略。代码如下：

```
if (which == TLBMissException)
{
    tlbAccess ++ ;
    int VAddr = machine->registers[BadVAddrReg] ;
    unsigned int vpn = VAddr / PageSize ;
    unsigned int offset = VAddr % PageSize ;
    TranslationEntry * entry ;
    if (vpn >= machine->pageTableSize) {
        DEBUG('a', "virtual page # %d too large for page table size %d!\n",
        VAddr, machine->pageTableSize);
        DEBUG('a', "Shutdown, initiated by user program.\n");
        interrupt->Halt();
    } else if (!machine->pageTable[vpn].valid) {
        DEBUG('a', "virtual page # %d is not valid!\n",
            VAddr, machine->pageTableSize);
        DEBUG('a', "Shutdown, initiated by user program.\n");
        interrupt->Halt();
    }
    printf("TLBAccess: %d, VPN: %d, ", tlbAccess, vpn) ;
    entry = &machine->pageTable[vpn];
    machine->tlb[turn] = *entry ;
    printf("TLBIndex: %d\n", turn) ;
    turn = (turn + 1) % TLBSize ;
}
```

需要注意的是：

1. 原本 TLB 和 PageTable 同时只能存在一个，这个限制被我去掉了。
2. 还需要定义 USETLB 让系统使用 TLB 做翻译。
3. 在 exception.cc 中需要加入处理 Exit 系统调用的代码。
4. 在异常处理函数中，如果出现页面错误，我选择直接终止程序。

使用 nachos 运行 sort（排序数目改成了 20）程序结果如下：

```
TLBAccess: 601, VPN: 4, TLBIndex: 0
TLBAccess: 602, VPN: 5, TLBIndex: 1
TLBAccess: 603, VPN: 2, TLBIndex: 2
TLBAccess: 604, VPN: 3, TLBIndex: 3
TLBAccess: 605, VPN: 14, TLBIndex: 0
TLBAccess: 606, VPN: 0, TLBIndex: 1
Machine halting!
```

从结果中可以看出 sort 正常运行，轮转替换策略产生了 606 次 TLB miss

Exercise3: 实现两种 TLB 替换策略：

除了 exercise2 中的轮转策略，我还实现了 LRU 和 LFU 替换策略。LRU 就是把最长时间没被使用的 TLB 项替换出去，LFU 就是把使用频率最少的 TLB 项替换出去。在 exception.cc 中添加的代码如下，功能是找到时间 time 或频率 freq 的最小值，并把该项 TLB 换出：

```
//2. LRU调度
int index = 0, min = 10000000 ;
for(int i = 0; i < TLBSize; i ++)
{
    if( (machine->tlb[i]).time <= min )
    {
        min = (machine->tlb[i]).time ;
        index = i ;
    }
}
machine->tlb[index] = *entry ;
printf("TLBIndex: %d\n", index) ;

//2. LFU调度
int index = 0, min = 10000000 ;
for(int i = 0; i < TLBSize; i ++)
{
    if( (machine->tlb[i]).freq <= min )
    {
        min = (machine->tlb[i]).freq ;
        index = i ;
    }
}
machine->tlb[index] = *entry ;
printf("TLBIndex: %d\n", index) ;
```

在 translation.cc 中添加维护 freq 和 time 的代码如下，每次命中一项 TLB 就更新它的值：

```
    tlb[i].time = TLBtime ++ ;
    tlb[i].freq ++ ;
```

在 translation.h 中的 translationEntry 类声明中加入 time 和 freq，在 machine.h 的类定义中加入全局的时间计数器 TLBtime 并在 machine 的构造函数中初始化为零。

LRU 算法运行 sort 的结果如下：

```
TLBAccess: 495, VPN: 3, TLBIndex: 2
TLBAccess: 496, VPN: 6, TLBIndex: 3
TLBAccess: 497, VPN: 4, TLBIndex: 0
TLBAccess: 498, VPN: 5, TLBIndex: 2
TLBAccess: 499, VPN: 2, TLBIndex: 0
TLBAccess: 500, VPN: 3, TLBIndex: 3
TLBAccess: 501, VPN: 0, TLBIndex: 3
Machine halting!
```

从结果可以看出 LRU 正常运行，产生了 501 次 TLB miss

LFU 算法运行 sort 会陷入死循环，因为在执行一段时间后 TLB 的某一项的频率是 1，而且两个访存在这个 TLB 项上来回替换。这两个访存可能分别是一行代码用来取指令和取数据，它们无法同时访存成功，导致这行代码一直不能运行。

综上，LRU 需要 501 次 TLB miss，轮转需要 606 次 TLBmiss，LFU 无法正常运行。

第二部分

Exercise 4 内存全局管理数据结构

设计并实现一个全局性的数据结构（如空闲链表、位图等）来进行内存的分配和回收，并记录当前内存的使用状态。

原来的物理内存分配方式是整体的，即把所有物理内存都给一个用户程序，并且顺序存放。现在我们需要使用页式内存分配，即以页为单位，可以不按顺序存放。因此，我们需要一个位图来标记哪些物理内存是空闲的，哪些已经被使用。

具体实现时，我在 machine.h 中定义了一个位图 memoryMap，在 machine.cc 中初始化为 NumPhysPage 大小。在 AddrSpace.cc 中的构造函数中，初始化 pageTable 时，我使用 memoryMap->Find 函数找到未使用的物理内存页，同时 Find 函数会自动把它标记为已使用。在 AddrSpace 析构函数中，我遍历所有虚拟内存对应的物理内存页，把它们在 memoryMap 中标记为未使用。

因为此时只有一个用户程序，因此还是顺序分配内存页，运行结果和之前一样。所以**我在这里不再展示代码和运行结果，留到下一个 exercise 中一并展示。**

Exercise 5 多线程支持

目前 Nachos 系统的内存中同时只能存在一个线程，我们希望打破这种限制，使得 Nachos 系统支持多个线程同时存在于内存中。

要支持多线程，必须改变 Nachos 的内存分配方式。原来的 Nachos 假定所有程序从第 0 页开始使用内存，并在第 0 页写入代码段和数据段。当有多个线程时，需要找到这个线程拥有的一个物理页，在这个物理页上写入代码段和数据段。为此，我们需要在分配地址空间时找到线程的一个物理页，并以此为基址写入代码和数据。

我在 AddrSpace 构造函数中插入代码如下：

```
for (i = 0; i < numPages; i++) {
    pageTable[i].virtualPage = i;

//pageTable[i].physicalPage = i;
//离散分配物理页
    pageTable[i].physicalPage = machine->memoryMap->Find();
    ASSERT(pageTable[i].physicalPage != -1) ;
    bzero(machine->mainMemory + pageTable[i].physicalPage*PageSize, PageSize);
    if(i == 0) initialPage = pageTable[i].physicalPage ;
```

我在 AddrSpace 构造函数中改变内存写入代码如下：

```
// 写入位置为第一个物理内存页
// 如果size大于一个物理页，可能出错

int codeVirtualAddr = noffH.code.virtualAddr+initialPage*PageSize;
if (noffH.code.size > 0) {
    DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
        codeVirtualAddr, noffH.code.size);
    executable->ReadAt(&(machine->mainMemory[codeVirtualAddr]),
        noffH.code.size, noffH.code.inFileAddr);
}
int initDateVirtualAddr = noffH.initData.virtualAddr+initialPage*PageSize;
if (noffH.initData.size > 0) {
    DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
        initDateVirtualAddr, noffH.initData.size);
    executable->ReadAt(&(machine->mainMemory[initDateVirtualAddr]),
        noffH.initData.size, noffH.initData.inFileAddr);
}
```

我在 AddrSpace 析构函数中插入代码如下

```
AddrSpace::~AddrSpace()
{
   printf("Before delete AddrSpace\n") ;
   machine->memoryMap->Print() ;
   printf("\nAfter delete AddrSpace\n") ;
   for( int i = 0 ; i < numPages ; i ++ )
   {
        if( pageTable[i].valid )
        {
            machine->memoryMap->Clear(pageTable[i].physicalPage) ;
        }
   }
   machine->memoryMap->Print() ;
   delete pageTable;
}
```

除了改动初始化内存的代码，我们还需要处理线程切换时 TLB 的清空。页表和快表都是翻译机制，但是有区别：页表是每个线程私有的，因此翻译结果不受其他线程影响；快表是全局唯一的，只能把一个线程的虚拟地址翻译成物理地址，如果一个新线程用它的虚拟页号查快表，会得到旧线程的物理地址，也就发生了错误。因此，每次线程切换时需要把快表清空。在 scheduler.cc 中的 run 函数里，我添加如下代码：

```
#ifdef USER_PROGRAM
    if( machine->tlb != NULL )
    {
        int i = 0 ;
        for( ; i < 4; i ++ ) // TLBSize = 4
        {
            (machine->tlb[i]).valid = 0 ;
        }
    }
#endif
```

现在需要测试内存分配算法是否正确，所以需要创建多个线程。为此，我在 userprog.cc 的 StartProccess 中新创建一个线程，用于再次执行 sort。创建的新线程运行另一个函数 MyStartProcess，且只会创建一次，避免死循环。代码如下：

```
void myStartProcess(int arg)
{
    if( arg == 1 ) StartProcess("../test/sort") ;
}

void
StartProcess(char *filename)
{
    //time = 1 ;
    Thread * newThread = new Thread("newThread", 0) ;
    newThread->Fork(myStartProcess, first) ;
    if( first ) first = 0 ;
```

为了让两个线程交叉执行，我在 mipssim.cc 的 Run 函数当中让线程每执行 400 条指令就 Yield，添加代码如下：

```
for (i = 1; 1; i++ ) {
    OneInstruction(instr);
    interrupt->OneTick();
    if (singleStep && (runUntilTime <= stats->totalTicks))
        Debugger();
    if( i % 400 == 0 ) currentThread->Yield() ;
}
```

为了让程序不直接退出，需要在 exception.cc 中加入处理 exit 的代码。

```
if ((which == SyscallException) && (type == SC_Exit)) {
    DEBUG('a', "Finish, initiated by user program.\n");
    printf("System call implement by Yang: exit\n") ;
    currentThread->Finish();
}
```

最后，在线程析构函数中，需要析构 AddrSpace 对象，代码只有一行所以就不展示了。

下面是程序运行结果。两个 sort 程序交替运行，一共产生了 1118 次 tlb miss。主线程 main 首先结束，调用 exit 系统调用退出。在 exit 中我令线程 Finish；接下来 newThread 上 CPU，析构主线程 Thread 对象，进而析构主线程的 AddrSpace 对象。从输出中可以看到，在 AddrSpace 析构后，内存管理的 BitMap 的前 15 个位置被归零了。接下来 newThread 也结束，运行终止。

```
Switching from thread "newThread" to thread "main"
TLBAccess: 1109, VPN: 3, TLBIndex: 0
TLBAccess: 1110, VPN: 14, TLBIndex: 3
TLBAccess: 1111, VPN: 5, TLBIndex: 1
TLBAccess: 1112, VPN: 2, TLBIndex: 2
TLBAccess: 1113, VPN: 0, TLBIndex: 0
System call implement by Yang: exit
Finish Thread

Switching from thread "main" to thread "newThread"
Before delete AddrSpace
Bitmap set:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,

After delete AddrSpace
Bitmap set:
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
TLBAccess: 1114, VPN: 3, TLBIndex: 3
TLBAccess: 1115, VPN: 14, TLBIndex: 2
TLBAccess: 1116, VPN: 5, TLBIndex: 1
TLBAccess: 1117, VPN: 2, TLBIndex: 0
TLBAccess: 1118, VPN: 0, TLBIndex: 3
System call implement by Yang: exit
Finish Thread
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

Exercise 6 缺页中断处理

基于 TLB 机制的异常处理和页面替换算法的实践，实现缺页中断处理（注意！TLB 机制的异常处理是将内存中已有的页面调入 TLB，而此处的缺页中断处理则是从磁盘中调入新的页面到内存）、页面替换算法等。

当产生 TLB miss 的时候，异常处理函数尝试从页表找到 miss 页面的信息。如果页表中也没有这个页面的信息，就会产生一个 PageFaultException，返回后再次尝试查找页面：

```
if( findEntry == FALSE )
{
    DEBUG('a', "virtual page # %d is not valid!\n",
        VAddr, machine->pageTableSize);
    machine->RaiseException(PageFaultException, VAddr);
    goto tryAgain ;
}
```

接下来，我们需要处理 PageFaultException。

首先，需要一个交换分区 swap，用来存储进程全部的页面。这个 swap 分区我加在了 addrspace.cc 中，因此是每个进程有独立的 swap。在 addrspace 构造函数中，需要把程序的代码和数据全部拷贝到 swap 分区中，代码如下：

```
swap = new char[size] ;
bzero(swap, size) ;
```

写入swap区

```
if (noffH.code.size > 0) {
    DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
        noffH.code.virtualAddr, noffH.code.size);
    executable->ReadAt(&swap[noffH.code.virtualAddr],
        noffH.code.size, noffH.code.inFileAddr);
}
if (noffH.initData.size > 0) {
    DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
        noffH.initData.virtualAddr, noffH.initData.size);
    executable->ReadAt(&swap[noffH.initData.virtualAddr],
        noffH.initData.size, noffH.initData.inFileAddr);
}
```

注意在析构函数中删除 swap 指针。

接下来，需要在 exception.cc 中加入对 PageFaultException 的处理。目标是在内存中找到一个页面，把缺页异常的页面从 swap 中复制到这个页面中。实现的思路是先遍历查询有无空页，如果有空页就选择该页；否则找到最久没有被调入 TLB 的页面并选择该页换出，也就是使用 LRU 策略。但是需要注意的，如果一个页面是很久之前被调入 TLB 的，但是一直没出去过 TLB，也会被我的算法误选择为换出的页面。为了解决这个问题，我需要对每个页面判断一下它是不是在 TLB 中，如果是的话，就不会把它换出。选择换出页的代码如下：

```
else if (which == PageFaultException)
{
    int virtualAddr = machine->registers[BadVAddrReg];
    int vpn = virtualAddr / PageSize ;
    int find = -1 ;
    int min = 1000000000 ;
    for( int i = 0 ; i < NumPhysPages; i ++ )
    {
        if( (machine->invertedList[i]).used == FALSE )
        {
            find = i ;
            break ;
        }
    }

    if( find == -1 )
    for( int i = 0 ; i < NumPhysPages; i ++ )
    {
        int cannot = 0 ;
        if( (machine->invertedList[i]).lastTime < min )
        {
            for(int j = 0 ; j < 4 ; j ++ )
                if( (machine->tlb[j]).virtualPage == (machine->invertedList[i]).vpn &&
                    currentThread->getTid() == (machine->invertedList[i]).tid)
                        cannot = 1 ;
            if( cannot == 1 ) continue ;
            min = (machine->invertedList[i]).lastTime ;
            find = i ;
        }
    }
```

选好换出页以后，如果它是脏页，需要写回到 swap 分区。最后，我们把缺页异常的页复制到这个物理页面上，并初始化一些标志。代码如下：

```
if( (machine->invertedList[find]).used )
{
    int tid = (machine->invertedList[find]).tid ;
    int ovpn = (machine->invertedList[find]).vpn ;
    if(freeTid[tid] == 0)
    {
        Thread * thread = Tpool[tid] ;
        bcopy(&machine->mainMemory[find*PageSize], &thread->space->swap[ovpn*PageSize], PageSize) ;
    }
}

bcopy(&currentThread->space->swap[vpn*PageSize], &machine->mainMemory[find*PageSize], PageSize) ;

(machine->invertedList[find]).used = TRUE ;
(machine->invertedList[find]).tid = currentThread->getTid() ;
(machine->invertedList[find]).vpn = vpn ;

(machine->invertedList[find]).entry.valid = TRUE;
(machine->invertedList[find]).entry.use = FALSE;
(machine->invertedList[find]).entry.dirty = FALSE;
(machine->invertedList[find]).entry.readOnly = FALSE;
(machine->invertedList[find]).entry.freq = 0;
(machine->invertedList[find]).entry.time = machine->TLBtime;
(machine->invertedList[find]).entry.physicalPage = find ;
(machine->invertedList[find]).entry.virtualPage = vpn ;
```

需要注意的是，写会脏页的时候需要判断该页所属的线程是否还存在，只有线程存在时才能写会。我通过我之前 lab 定义的全局线程标志 freeTid 的值来判断线程是否被析构。

可以发现，我在这个 exercise 中已经开始使用倒排页表了。因为必须要有一个数据结构来记录每个物理页的信息。关于倒排页表的详细描述我将在 chanllange 中说明。

测试部分留在后面展示。

三、Lazy-loading

Exercise 7 我们已经知道，Nachos 系统为用户程序分配内存必须在用户程序载入内存时一次性完成，故此，系统能够运行的用户程序的大小被严格限制在 4KB 以下。请实现 Lazy-loading 的内存分配算法，使得当且仅当程序运行过程中缺页中断发生时，才会将所需的页面从磁盘调入内存。

Lazy-loading 与之前的 pagefault 处理 exercise 同出一辙。首先，我们需要删除 addrspace 构造函数中对物理内存写入的代码。其次，我们需要去掉关于物理空间限制的 ASSERT，并分配足够的页面给 swap 分区。

其他的工作交给 PageFaultException 处理函数就好，在上面已经展示，这里不再赘述。

四、Challenges

Challenge 1 为线程增加挂起 SUSPENDED 状态，并在已完成的文件系统和内存管理功能的基础之上，实现线程在“SUSPENDED”，“READY”和“BLOCKED”状态之间的切换。

SUSPEND 状态意味着线程要释放所有资源，包括物理页；并且不能先于 READY 的线程运行。具体实现细节如下：我在 Thread.h 中增加 SUSPEND 状态，在 Thread.cc 中加入 oldprior

变量。我在 machine.cc 中加入 Thread::Suspend() 函数。该函数把内存中所有属于本线程的页面写会到 swap 区，然后把这些物理页置为可用状态。接着我把该线程优先级保存在 oldprior 里，把 prior 置为最低的优先级 6，修改线程状态为 SUSPEND。最后我调用 Yield 函数，把线程挂起。需要注意的是，由于它此时优先级最低，所以一定是在所有非 SUSPEND 线程执行完后才有机会运行。代码如下：

```
void Thread::Suspend ()
{
    printf("Suspend: %s\n", currentThread->getName()) ;
    for( int i = 0 ; i < NumPhysPages; i ++)
    {
        if( machine->invertedList[i].tid == tid )
        {
            bcopy(&machine->mainMemory[i*PageSize],
                &space->swap[machine->invertedList[i].vpn*PageSize], PageSize) ;
            machine->invertedList[i].used = 0 ;
        }
    }
    for( int i = 0 ; i < NumPhysPages ; i ++ )
        printf("PhysPage: %d, thread: %d, VirtPage: %d\n", i,
           (machine->invertedList[i]).tid, (machine->invertedList[i]).vpn) ;

    currentThread->oldprior = currentThread->prior ;
    currentThread->prior = 6 ;
    currentThread->status = SUSPEND ;
    Yield() ;
}
```

在 scheduler.cc 的 run 函数中，我们需要判断当前线程是不是被 SUSPEND 的线程。如果一个 SUSPEND 线程重新得到运行，我们需要恢复它原来的优先级。代码如下：

```
currentThread = nextThread;         // switch to the next thread
if(currentThread->status == SUSPEND){
    currentThread->prior = currentThread->oldprior ;
}
currentThread->setStatus(RUNNING);      // nextThread is now running
```

Challenge 2　多级页表的缺陷在于页表的大小与虚拟地址空间的大小成正比，为了节省物理内存在页表存储上的消耗，请在 Nachos 系统中实现倒排页表。

页表记录的是虚拟内存的状态，也就是每个虚拟页对应哪个物理页。倒排页表用于标识物理内存的状态，也就是每个物理页记录的是哪个虚拟页。

为了节省物理内存，需要用倒排页表代替页表。首先，我在 machine.h 中定义 InvertedList 的数据结构，包含是否被使用，虚拟页号，页的线程号，最后使用时间，以及一个页表条目用于向 TLB 赋值。代码如下所示：

```
class InvertedList{
    public:
        bool used ;
        int vpn ;
        int tid ;
        int lastTime ;
        TranslationEntry entry ;
        static int time ;
};
```

在 machine 的初始化时，创建一个 invertedList。需要注意的是，倒排页表是全局唯一的，所以应该在 machine.cc 中定义，而不是在 addrSpace 中定义。

```
invertedList = new InvertedList[NumPhysPages] ;
for (i = 0; i < NumPhysPages; i++)
{
    invertedList[i].used = FALSE ;
    invertedList[i].lastTime = 0 ;
}
```

在 machine 的析构函数中删除 invertedList，这里不再赘述。

在 AddrSpace 析构时，需要归还该线程的物理页。为此，我们需要遍历倒排页表，找到该线程使用的物理页，把使用状态置为 FALSE。另外有调试输出的代码。

```
for( int i = 0 ; i < NumPhysPages; i ++)
{
    if( machine->invertedList[i].tid == currentThread->getTid() ) machine->invertedList[i].used = 0 ;
}
printf("pageTime: %d\n", PageTime) ;
for( int i = 0 ; i < NumPhysPages ; i ++ )
    printf("PhysPage: %d, thread: %d, VirtPage: %d\n", i,
       (machine->invertedList[i]).tid, (machine->invertedList[i]).vpn) ;

delete swap ;
```

倒排页表的使用是在 TLB MIss 的异常处理函数中。系统首先试图在物理内存中找到 miss 的物理页，为此，需要查找 invertedList，找到虚拟页号和线程号与 miss 的页面相同的条目，然后把该条目复制到 TLB 中。代码如下：

```
tryAgain:
for( int i = 0; i < NumPhysPages; i ++ )
{
    if( (machine->invertedList[i]).vpn == vpn
        && (machine->invertedList[i]).used == TRUE
        && (machine->invertedList[i]).tid == currentThread->getTid() )
    {
        entry = (machine->invertedList[i]).entry ;
        findEntry = TRUE ;
        //printf("page in, vpn: %d, page: %d\n", vpn, i) ;

        (machine->invertedList[i]).lastTime = PageTime ;
        PageTime ++ ;
        break ;
    }
}
```

如果在物理内存中找到了页面，就可以完成 TLB miss 处理了。但是，如果该页不在物理内存中，就需要产生一个 PageFault 异常，把该页从 swap 分区中换到物理内存中。这部分在 exercise6 中已经介绍，这里不再赘述。处理好 PageFaultException 之后，跳转到 tryAgain，就可以在物理内存中找到 miss 的 TLB 页了。

需要特别留意的是，每次在 TLBmiss 处理函数中把一个物理页信息换入 TLB，需要对该物理页的最后访问时间进行更新，并且把 PageTime 变量加 1.

倒排页表的换入换出操作全在 exception.cc 之中，在 exercise6 已经讲过，这里不再赘述。但是，这里要仔细讲解一下对倒排页表的赋值。

```
(machine->invertedList[find]).used = TRUE ;
(machine->invertedList[find]).tid = currentThread->getTid() ;
(machine->invertedList[find]).vpn = vpn ;

(machine->invertedList[find]).entry.valid = TRUE;
(machine->invertedList[find]).entry.use = FALSE;
(machine->invertedList[find]).entry.dirty = FALSE;
(machine->invertedList[find]).entry.readOnly = FALSE;
(machine->invertedList[find]).entry.freq = 0;
(machine->invertedList[find]).entry.time = machine->TLBtime;
(machine->invertedList[find]).entry.physicalPage = find ;
(machine->invertedList[find]).entry.virtualPage = vpn ;
```

第一部分是描述物理页是否被占用，页的线程号是多少，页的虚拟页号是多少。

第二部分是页的具体信息，包括页面是有效的，页面未被访问，页面未被修改，页面不是只读，页面被访问频数为 0，页面最后一次被访问时间为当前时间，页面的物理页号和虚拟页号。

最后，我们需要把使用页表 PageTable 的代码都注释掉。这些代码包含在 addrSpace 类的各种函数中。

测试：Exercise6 – Challange2

现在，我对上面的几个练习一起测试。我的测试方案是使用倒排页表和 Lazy-loading 技术，运行两个所需空间超过 nachos 物理内存大小的进程（sort），期间进行 Suspand 操作。

首先，我在 progtest.cc 的 startProcess 中创建新线程，然后使用新线程再次调用 startProcess。为了避免死循环，需要用一个变量标记执行次数，第一次时创建新线程，第二次就不创建新线程了。代码如下：

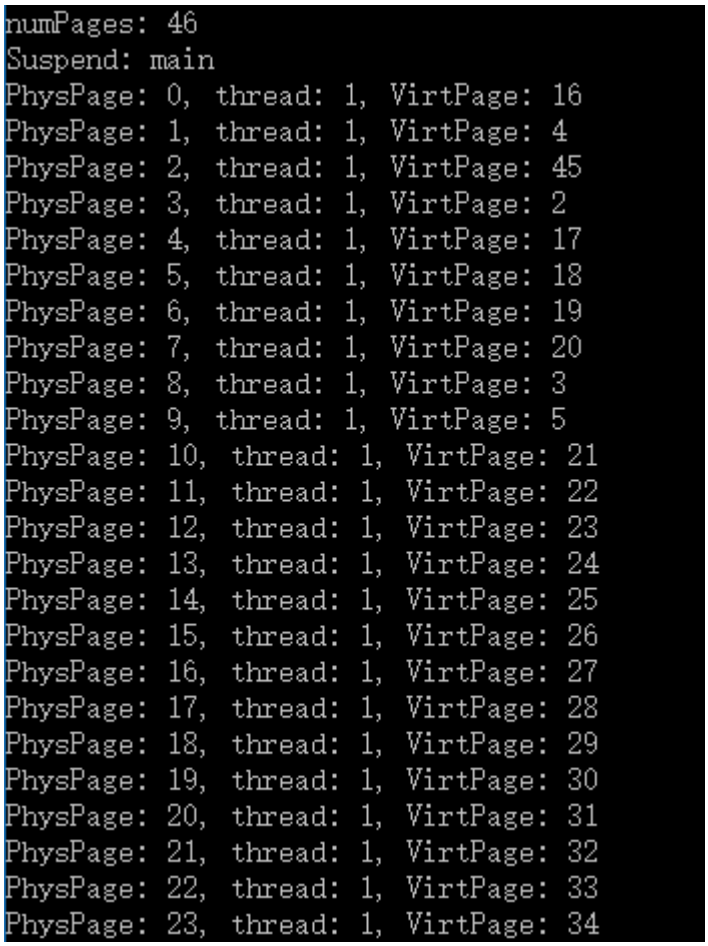
```
void myStartProcess(int arg)
{
    if (threadToBeDestroyed != NULL) {
        delete threadToBeDestroyed;
        threadToBeDestroyed = NULL;
    }
    if( arg == 1 )
    {
        StartProcess("../test/sort") ;
    }
}

void
StartProcess(char *filename)
{
    //time = 1 ;
    Thread * newThread = new Thread("newThread", 2) ;
    newThread->Fork(myStartProcess, first) ;
    if( first ) first = 0 ;
```

接下来，我需要在线程执行过程中调用 Suspend 函数。为此，我在 mipssim.cc 的 Run 函数中判断指令执行的次数，如果是两千万的整数倍就进行 Suspend:

```
    for (i = 1; 1; i++ ) {
        OneInstruction(instr);
        interrupt->OneTick();
        if (singleStep && (runUntilTime <= stats->totalTicks))
            Debugger();
        if( i % 20000000 == 0 ) currentThread->Suspend() ;
    }
```

在 addrSpace 析构函数中和 Suspend 函数中，我都会打印当前倒排页表的状态，运行结果如下：



从上面这张图片，我们可以看到 sort 函数对 1024 个元素排序时，虚拟内存有 46 页，但是物理内存只有 32 页。通过 PageFault 机制，sort 程序可以正常运行，说明 PageFault 的相关处理，包括 Lazy-Loading 机制，是成功的。在第一次 Suspend 时，倒排页表中各个虚拟页交错分布，说明 LRU 的页替换策略是有效的。

从上面的图片可以看到，主线程切换到新线程，新线程执行 sort，新线程再次 Suspend 时倒排页表中的虚拟页已经都属于新线程了。

```
System call implement by Yang: exit
Finish Thread: main

Switching from thread "main" to thread "newThread"
delete: main, 1
pageTime: 2636809
PhysPage: 0, thread: 1, VirtPage: 5
PhysPage: 1, thread: 1, VirtPage: 45
PhysPage: 2, thread: 1, VirtPage: 18
PhysPage: 3, thread: 1, VirtPage: 2
PhysPage: 4, thread: 1, VirtPage: 3
PhysPage: 5, thread: 1, VirtPage: 19
PhysPage: 6, thread: 1, VirtPage: 4
PhysPage: 7, thread: 1, VirtPage: 20
PhysPage: 8, thread: 1, VirtPage: 21
PhysPage: 9, thread: 1, VirtPage: 22
PhysPage: 10, thread: 1, VirtPage: 23
PhysPage: 11, thread: 1, VirtPage: 24
PhysPage: 12, thread: 1, VirtPage: 0
PhysPage: 13, thread: 2, VirtPage: 24
PhysPage: 14, thread: 2, VirtPage: 25
PhysPage: 15, thread: 2, VirtPage: 26
PhysPage: 16, thread: 2, VirtPage: 27
PhysPage: 17, thread: 2, VirtPage: 28
PhysPage: 18, thread: 2, VirtPage: 29
```

从上面的图片可以看到，主线程也就是 1 号线程执行结束了，新线程上 CPU 后把主线程析构。此时倒排页表中新老线程的虚拟页面共同存在，说明上次主线程上 CPU 后只使用了 12 个页面。其他新线程占据的物理页面虽然被标记为可使用了，但还没被替换出去。

从统计变量中可以看到，已经经历了 2636809 次 TLB miss。

```
Switching from thread "newThread" to thread "newThread"
delete: newThread, 2
pageTime: 2687014
PhysPage: 0, thread: 1, VirtPage: 5
PhysPage: 1, thread: 1, VirtPage: 45
PhysPage: 2, thread: 1, VirtPage: 18
PhysPage: 3, thread: 1, VirtPage: 2
PhysPage: 4, thread: 1, VirtPage: 3
PhysPage: 5, thread: 1, VirtPage: 19
PhysPage: 6, thread: 1, VirtPage: 4
PhysPage: 7, thread: 1, VirtPage: 20
PhysPage: 8, thread: 1, VirtPage: 21
PhysPage: 9, thread: 1, VirtPage: 22
PhysPage: 10, thread: 1, VirtPage: 23
PhysPage: 11, thread: 1, VirtPage: 24
PhysPage: 12, thread: 1, VirtPage: 0
PhysPage: 13, thread: 2, VirtPage: 5
PhysPage: 14, thread: 2, VirtPage: 45
PhysPage: 15, thread: 2, VirtPage: 18
PhysPage: 16, thread: 2, VirtPage: 2
PhysPage: 17, thread: 2, VirtPage: 3
PhysPage: 18, thread: 2, VirtPage: 19
PhysPage: 19, thread: 2, VirtPage: 4
PhysPage: 20, thread: 2, VirtPage: 20
PhysPage: 21, thread: 2, VirtPage: 21
PhysPage: 22, thread: 2, VirtPage: 22
PhysPage: 23, thread: 2, VirtPage: 23
PhysPage: 24, thread: 2, VirtPage: 24
```

最后，新线程的执行也结束了，由另一个新线程回收它的空间。从上图中可以看到，新线程，也就是 2 号线程，使用了从 13 号开始的页面，并且页面分布情况和 1 号线程相同。这说明 2 号线程的运行状态完全正常，进程切换以及对多线程的支持是成功的。需要说明的是，之所以 2 号线程从 13 开始使用页面而不是从 0 开始使用页面，是因为找空页面的算法是每次接着上次找到的位置开始找的，而不是从 0 开始找的，这样做可以加快找空页面的速度。这两个 sort 程序一共执行了 41580597 次指令，与 1024 的平方数量级相同，说明程序的运行是完整的，正确的。

本程序运行时间约为 9 秒。

内容三：遇到的困难以及解决方法

困难 1：低估了程序产生的 TLB miss 次数

当 sort 排序 1024 个数时，我最初还在调试输出 tlb miss，但是发现根本停不下来。我注释掉调试输出后，发现产生了死循环。经过仔细的排查，我才发现是我用 LRU 找最早使用的页面时，设置的 min 临时变量的初始值只有 100 万，而 tlb time 会达到 100 万以上，所以会找不到可以替换的页面，一直产生 tlb miss。设定 min 值的时候我没有考虑太多，没想到居然会溢出我的设定值。

困难 2：忘记改写析构函数

多线程时，当一个线程想换出一个已结束线程的页面时，会发送 segmentation fault。经过思考，我发现我忘记在线程析构时把它占用的物理页面标记为空闲，导致新线程想把这个页面换出时，会把这个页面写回已经被析构线程的 swap 分区，导致内存访问越界。

困难 3：测试程序比较难构造

多线程时，需要创造多个线程：我在 StartProcess 函数中使用 myStartPcrocess 函数创建了一个新线程；还需要交替上下 CPU 的场景：我在 mipssim 中判断指令执行的次数，到一定次数就下 CPU。最后，还需要直观地展示内存状态：我选择在线程析构时打印当前页表状态。

困难 4：编译链接的依赖关系比较复杂

在添加代码时，经常会遇到需要引用的头文件在另一个文件夹这种情况。而在 thread.cc 中引用 machine.h 和 bitmap.h 会导致编译不能通过。我没有找到好的解决方案，只好通过把相关的函数写到 machine.cc 中来解决这个问题。

困难 5：需要进行版本控制

后面的 exercise 有时会与前面的 exercise 冲突，比如要用倒排页表替代页表。这就导致完成了后面的 exercise 后前面的 exercise 就不能跑了。因此，我需要把所有版本加以保存。

使用 Git 是一个好的选择。

困难 6：代码繁杂，包罗万象

在添加一个新功能时，往往牵一发而动全身。比如添加对 page fault 的支持，就需要涉及到 machine，addrspace，exception 等等文件，需要考虑进程从构建到析构的全过程，多个进程存在时还需要考虑进程切换的后果。因此，改代码往往需要同时修改近 10 个文件，然后进行调试又是比较痛苦的过程。

内容四：收获及感想

通过这次 lab，我对虚拟内存的 tlb 和页表机制了解更加深入了。我在改写代码，维护大的工程方面的能力也有了很大提高。虚拟内存是非常复杂的，而现代操作系统又能让这一切井井有条地高效运行，使我对操作系统也有了由衷的赞叹之情。

内容五：对课程的意见和建议

我认为这次 lab 布置的顺序不太合理。倒排页表应该先于 page fault 实现。现在这样的顺序导致最后几个 lab 都杂糅再一起，难解难分。

我认为应该先做系统调用 lab 再做虚拟内存 lab。现在用户程序没办法输出到控制台，所以没办法测试用户程序运行的正确性，只能通过运行的时间大概判断一下程序是不是完整地运行了，但是结果对不对是不知道的。

另外，exit 系统调用 nachos 没有给实现，导致我在这个地方栽了一些跟头。最开始我令程序 halt，结果多线程的时候只有一个线程可以运行完。这时我才意识到应该在 exit 中让当前线程 Finish。我觉得这个应该由 nachos 事先实现好，否则与当前 lab 的主题就偏离了。

内容六：参考文献

【1】现代操作系统　　陈向群