虚拟内存实习报告

姓名 **彭广举** 学号 1200012964
日期 2015-4-20

目录

内容一：总体概述

这一部分主要的内容是通过阅读相关代码，了解 nachos 用户程序的执行过程，之后完成 TLB，页表和虚拟内存等的实现。

当前的 nachos 只支持单个用户程序，没有用到 TLB，并且每个用户程序不能超过物理内存 4kb 大小的限制 。为了解决如上的线程，需要引入虚拟内存，使得每个用户程序从自己本身来看都有无限大的内存空间可供使用，并且多个线程之间没有干扰 。有了虚拟内存之后，就需要有页表来完成虚拟地址和物理地址之间的映射，以便完成地址翻译 。同时需要有数据结构来管理物理内存，以便对物理内存的分配和回收 。虚拟内存的实现可以采用文件的方式，对物理内存的管理可以使用位图。

TLB 又叫快表，是为了加速地址翻译的过程，利用局部性原理，减少对内存中页表的访问 。TLB 可以是全相连或者组相连的，全相连的实现相对简单，组相连的性能会相对较好，但在 nachos 中 tlb 只有 4 个，所以采用全相连即可 。

【用简洁的语言描述本次 lab 的主要内容；阐述本次 lab 中涉及到的重要的概念，技术，原理等，以及其他你认为的最重要的知识点。这一部分主要是看大家对 lab 的总体的理解。

要求：简洁，不需要面面俱到，把重要的知识点阐述清楚即可。】

内容二：任务完成情况

任务完成列表（Y/N）

	Exercise1	Exercise2	Exercise3	Exercise4	Exercise5	Exercise6
第一部分	Y	Y	Y			
第二部分	Y	Y	Y			
第三部分	Y					
第四部分	N	Y				

具体 Exercise 的完成情况

第一部分

Exercise1

源代码阅读

code/userprog/progtest.cc：progtest.cc 是 nachos 中和用户程序执行相关的模块。首先是 StartProcess 函数，该函数的功能就是加载并执行一个用户程序 。

StartProcess 接受一个文件名，然后打开该文件，之后通过类 AddrSpace 的构造函数为用户程序分配并初始化地址空间，过程如下：

1） 将文件头中内容加载到一个 NoffHeader 的结构体中，该结构体定义了 Nachos 目标代码格式，并且在有必要的时候进行小端到大端的转换。

2） 计算用户程序所需要用户空间的大小，包括代码段，已初始化的数据段，为初始化的数据段和用户栈，并计算所需要的页的个数。

3） 建立从用户程序地址到物理地址的翻译机制，目前使用的是完全的一对一的。该页表定义在 AddrSpace 类中，当程序运行前，调用 RestoreState 函数，将 machine 的 pageTable 指向该 AddrSpace 对象的页表。

4） 初始化物理内存空间（machine->mainMemory），将程序的代码段和数据段拷贝到内存中。

5） 关闭文件，初始化用户寄存器的值，加载页表到 machine 中，调用 machine->Run 执行用户程序。

Progtest.cc 中还有一个函数 ConsoleTest，用于控制台输入输出的测试。

code/machine/machine.h(.cc)：这两个文件主要用于模拟 nachos 用户程序运行的机器。machine.h 中定义了内存大小，TLB 大小，物理页的个数和页的大小，还有一些异常的种类和一些特殊用户寄存器的编号。之后是类 Instruction 用于将目标代码转为 mips 可执行的代码。Machine 类模拟机器的运行，通过 Run 函数模拟 mips 指令的执行 。Run 调用 OneInstruction， OneInstruction 从 PC 寄存器所指向的地址调用 ReadMem 来读取指令，ReadMem 中调用 Translate 函数完成从虚拟地址到物理地址的转换，从物理地址中读取指令放入 Instruction 类的对象中进行解码，之后模拟 mips 执行。

code/machine/translate.h(.cc)：头文件中定义了页表 entry 的类，包含物理页号，虚拟页号和 vaild/use/dirty/readonly 属性。translate.cc 中主要实现了 machine 的读写内存和地址转换的功能。读写内存的粒度有 1/2/4，目前的实现就是根据 translate 中得到的物理地址进行简单的读或者写 。地址翻译机制的实现是首先判断对齐，然后计算虚拟页号和偏移，如果 tlb 为 null 则在页表中寻找，否则在 tlb 中找（目前还没有用到 TLB，所以 tlb=null），最后根据页表项计算出物理地址。在转换过程中可能发生各种意外，抛出很多异常，异常的处理则在 code/userprog/exception.cc 中的 ExceptionHandler 中处理，异常的处理发生在内核态。nachos 对于异常或者系统调用的处理过程是，当发生异常时，会调用 machie->RaiseException 函数，该函数得到异常的类型和发生异常处的虚拟地址，然后将虚拟地址存入 badAddr 寄存器中，DelayLoad 是结束当前过程中运行的任何东西，和 mips 模拟有关，之后设置当前状态为系统态（内核态），然后调用 ExceptionHandler 进行处理，处理返回后，重新设置为用户态。

Exercise2

TLB MISS 异常处理

Nachos 的异常处理函数是 exception.cc 中的 ExceptionHandler 函数，当前只有对 Halt 系统调用的处理。添加对 TLB 异常的 PageFaultException 的处理如下：

```
else if (which == PageFaultException) {
        int vaddr = machine->ReadRegister(BadVAddrReg);
        machine->LRUSwapTLB(vaddr);
}
```

当异常的类型是 PageFaultException 时，先获取发生异常的地址，然后调用 TLB

的 Swap 函数，这里调用的是 LRU 算法的处理函数。

Exercise3

TLB 置换算法

分别实现了 LRU 和 FIFO 两种置换算法。

首先，在 nachos 中，若使用了 TLB，那么每次访问都会去访问 TLB，如果 TLB 没有我们需要的页表项，那么进入 PageFaultException 处理函数，调用 TLB 的置换函数进行置换。若 TLB 中有空项，则进入页表，找对应的页表项是否有效，如果有效，即说明该页在内存中（当前都在内存中，为之后的虚拟内存做处理），则将对应的表项加入 TLB，TLB 中对应的项设为有效；若 TLB 没有空项，则根据置换算法，选取其中一项作为替换，将将要被替换的表项写回页表，在页表中找到需要用的页表项写入 TLB。再重新进行一遍 Translate，这次就会 TLB 命中。

LRU 的实现是在页表项中加入一个时间 lastUsedTime，记录该表项对应的页最近一次的使用时间，每次某个页被使用时，更新其在 TLB 中的页表项的 lastUsedTime 为当前的时钟 totalTicks。FIFO 是在页表项中增加了变量 inTLBTime 表示该项第一次进入 TLB 的时间，当一个表项放入 TLB 时，设定改变量为 totalTicks。

LRU：LRUSwapTLB 函数，如果 TLB 没满，则填充一条空的表项。若满了，则找到其中 lastUsedTime 最小的作为将要被替换的表项，将其写回页表。将新的表项从页表中取出，放入 TLB。在 Translate 中每次 TLB 命中时，更改对应的表项的 lastUsedTime 为当前的 totalTicks。

FIFO：FIFOSwapTLB 函数，同样若 TLB 没满，就填充空项，否则，找出 inTLBTime 最小的作为将要被替换的表项，将其写回页表，将新的表项从页表取出，写入 TLB，并设置该项的 inTLBTime 为当前的 totalTicks。

两种置换算法的对比会在之后实现完虚拟内存后给出。

第二部分

Exercise1

内存全局管理数据结构

用位图 BitMap 来管理内存，物理内存一共 32 页，所以用 32 位的位图来管理。每次申请物理内存时，调用 BitMap 的 find 函数，找到一块空闲的页，如果没有空闲页 find 函数返回-1。

另外为了记录各个物理页的状态，比如脏位，该内存上的内容所属的线程等，新建了一个类 PhysicalPage：

```
class PhysicalPage {
  public:
    int vaPageNum;
    bool valid;
    bool dirty;
    int lastUsedTime;
    int heldThreadID;

};
```

在 machine 中新建一个 32 个元素的 PhysicalPage 的数组 physPageTable，来更详细地管理内存，并且为虚拟内存和多线程的实现做准备。

先说一下虚拟内存的实现。

由于当前的 nachos 中虚拟内存和物理内存是一一对应的，而物理内存一共有 32 × 128=4KB，所以限制了可以执行的用户程序所用的内存空间只能是 4KB，并且只能运行一个程序。为了使 nachos 能够支持多道程序的执行，并且每个程序能够使用的内存能够无限大，而使用了虚拟内存的机制。

虚拟内存的实现:

首先对于 AddrSpace 类中，在构造函数中:删去一对一的页表的构造过程，对内存的初始化（因为如果支持多个程序运行的话，不能初始化为 0），写入内存的过程和对虚拟页表限制等的代码。对新建的页表进行初始化，然后新建一个文件，作为当前线程的虚拟地址空间，当内存不够用时，则把必要的信息写入到该文件中，在 AddrSpace 类中增加一个打开文件的指针 OpenFile* vaSpace;指向新建的文件。之后，把目标代码文件中的代码段和初始化的数据段拷贝到 vaSpace 打开的文件中，代码段在 vaSpace 中所在的文件位置从 0 开始，数据段从 noffH.code.size 开始，刚好对应于它们的虚拟地址，所以在 vaSpace 中，代码或者数据虚拟地址和它们在文件中的位置是相同的，对于不在内存中的部分的访问，都需要访问该文件，所以 vaSpace 的 delete 操作在 AddrSpace 的析构函数中执行。最后，计算代码段所占的虚拟页的个数，然后遍历这些虚页，用 BitMap 去找是否有空闲的物理页，如果有，则将该虚页内容写入物理页中，并把对应信息写入页表 pageTable 和 machine->physPageTable（建立映射关系，设置 valid 位为 TRUE，其余位为 FALSE，machine->physPageTable 的 heldThreadID 设置为当前线程 ID）。只写入代码段是因为如果是多线程时，一个线程的数据段太大会提前占住内存，而并没有被访问就发生了线程切换，新的线程则会把这些数据段换下来，就会造成很多的内存物理页的切换。

地址空间建立完成后，调用 InitRegisters()初始化用户寄存器，RestoreState()将 machine->pageTable 指向当前地址空间的 pageTable。然后调用 machine->Run()开始执行。machine->Run()会一条一条地从内存中读取指令，并模拟 mips 执行。

运行过程中对内存的读写过程是调用 Machine 类的 ReadMem 和 WriteMem 来执行的（在 translate.cc 中）。先调用 Translate 函数进行地址翻译，如果出现异常则调用 RaiseException 处理异常。对于 PageFaultException，第一次的异常处理是将对应的 TLB，页表，内存中的信息准备就绪，所以如果是 PageFaultException，则需要进行第二次的地址翻译，所以有:

```
exception = Translate(addr, &physicalAddress, size, FALSE);
if (exception != NoException) {
        machine->RaiseException(exception, addr);
        if (exception != PageFaultException)
                return FALSE;
        ExceptionType sdException =  Translate(addr, &physicalAddress, size, FALSE);
        if (sdException != NoException) {
                machine->RaiseException(sdException, addr);
                return FALSE;
        }
}
```

在 Translate 函数中，由于用到了 TLB（这里需要在 usrprog/Makefie 中添加-DUSE_TLB），所以直接进入 tlb!=NULL 的代码段。先遍历 TLB 查找要用到的表项是否在 TLB 中，若在，则得到对应的页表项，否则，抛出 PageFaultException，调用异常处理函

数，接着会调用 TLB 的置换函数（在 machine.cc)，以 LRU 为例，LRUSwapTLB：

先找到一个可用的 TLB 槽，空的或者 LRU 算法替换下来的，如果是替换下来的，则将对应的 tlb 中的项写回页表。从页表中取出新的页表项写入 TLB 对应的槽位中，当然可能会发生该虚拟地址对应的内容不在内存，即页表中对应的项是无效的，需要从虚拟地址空间中取出对应内容写入内存，再更新页表和 TLB，具体的做法在该部分的 Exercise3 中给出。

TLB 更新后，会再次调用 Translate 函数，这次就会发生命中，得到对应的表项，此时更新页表项的 lastUsedTime 和对应物理页的 lastUsedTime，之后计算出对应的物理页号，更新页表项的 use 位为 TRUE，如果是写操作，更新 dirty 位为 TRUE，同时更新物理页表对应的 dirty 位。最后根据偏移，计算出物理地址，返回。得到物理地址后， ReadMem 和 WriteMen 就可以从对应的内存中的位置访问数据了。

至此，虚拟内存基本实现了。

Exercise2

多线程支持

首先考虑到多个线程共用物理内存，所以共用物理页表 physPageTable，所以在 physPageTable 中 heldThreadID 指明了该内存页的内容所属的线程，因为之前在 system 中维护了一个全体线程的数组，所以可以简单根据线程号找到对应的线程信息。当一个线程运行时，是根据 machine->pageTable 指向该线程的地址空间的 pageTable 来实现的，运行不同的线程，就是 machine->pageTable 指向不同线程的页表，从而可以访问不同的数据。

因为 TLB 也是共用的，所以为了实现方便，当一个线程将要被切换下去时，将 tlb 内容全部写回对应页表，并清空 TLB，如果不这么做的话，每次访问 tlb 都需要判断该表项是否属于该线程，会比较麻烦。所以在 AddrSpace 的 SaveState 函数中调用 ClearTLB 函数清空 TLB。在线程机制实习中，我们知道，发生线程切换是在 Scheduler 的 Run 函数中，在 SWITCH 之前会调用 Thread 的 SaveUserState()保存用户寄存器，AddrSpace 的 SaveState 保存用户状态，SWITCH 结束后 RestoreUserState()恢复用户寄存器和 RestoreState()恢复用户状态（将页表切回）。

为了实现多个线程的同时运行，在 usrprog 中实现了一个函数 TestMultProcess()：

```
void
TestMultProcess()
{
    Thread *t1 = new Thread("array");
    Thread *t2 = new Thread("loop");
    t1->Fork(StartArray, t1->GetThreadID());
    t2->Fork(StartLoop, t2->GetThreadID());
}
```

用于测试给定的若干用户程序运行，这里这用到了两个 array 和 loop，这是为了测试方便，自己在 test 中添加的程序，具体的会在测试部分给出。StartArray 和 StartLoop 都和 StartProcess 一样，只是参数不同，分别执行 array 和 loop 用户程序（Fork 好像不能传字符串的参数，就采用这种粗暴的方法解决了。。。）。多线程之间的切换是利用时钟中断实现的，每执行一条用户指令会调用一次 OneTick()，所以时钟会前进的。

这里主要要解决的问题是，当一个线程发现内存满了时，需要将内存中的一页换下来（我采用的是 LRU，具体会在下个 Exercise 中给出），而要换下来的这一页内存是属于其他线程的，那么需要更改那个线程的页表，使得其对应的页表项为无效，若该页是 dirty 的，

则需要将该页写回它所属的线程的虚拟地址空间 vaSpace 中。具体代码会在下一个 Exercise 中给出。

Exercise3

缺页中断处理

已经知道，nachos 运行时，先检查 TLB，若在 TLB 中没有找到，则发生 PageFaultException，也就是说，每次读写内存，进行地址转换时，都是对 TLB 的访问，并没有直接访问页表。所以在 TLB 的异常处理过程中，要包括对于内存的缺页异常处理。同样 TLB 的置换采用 LRU，内存页的置换也采用 LRU。

之前说过，TLB 的 LRU 算法，是当 TLB 没有空槽时，找到 lastUsedTime 最小的槽去替换。已经找到了可以放页表项的 TLB 槽，现在去页表中找对应的页表项，如果该表项有效，那么直接写入 TLB 槽中，否则，说明还没有为该虚页分配物理页，则调用 AllocatePhysPage 分配物理页。

```
void Machine::AllocatePhysPage(int badVA)
{
        unsigned int vpn = (unsigned) badVA / PageSize;   // 虚拟页号
        int virAddr = vpn * PageSize;
        pageTable[vpn].virtualPage = vpn;

        int ppn = mBitMap->Find();

        if (ppn != -1)  {   // 有空闲块
            pageTable[vpn].physicalPage = ppn;
            pageTable[vpn].valid = TRUE;
        } else {            // 需要完成物理页的置换
            ppn = LRUSwapPages();
            pageTable[vpn].physicalPage = ppn;
            pageTable[vpn].valid = TRUE;
            pageTable[vpn].dirty = FALSE;
        }

        physPageTable[ppn].vaPageNum = vpn;

        currentThread->space->vaSpace->ReadAt(&(machine->mainMemory[ppn*PageSize]),
         PageSize, virAddr);

        physPageTable[ppn].valid =TRUE;
        physPageTable[ppn].dirty = FALSE;
        physPageTable[ppn].heldThreadID = currentThread->GetThreadID();
}
```

如代码所示，先用 BitMap 找空闲的物理页，若没有找到，说明物理内存已满，则调用 LRUSwapPages 进行物理页的切换，从而得到可用的物理页。然后 pageTable 和 physPageTable 的对应信息，并把虚拟地址空间中的内容读入内存。

```
int Machine::LRUSwapPages()
{
        int minTime = physPageTable[0].lastUsedTime;
        int swapPageNum= 0;
        for (int i = 0; i < NumPhysPages; ++i) {
            if (physPageTable[i].lastUsedTime < minTime) {
                minTime = physPageTable[i].lastUsedTime;
                swapPageNum = i;
            }
        }

        Thread* tempThread = threadsInfo[physPageTable[swapPageNum].heldThreadID];
        if (tempThread != NULL && physPageTable[swapPageNum].valid
        && physPageTable[swapPageNum].dirty) {       // 写回文件

            tempThread->space->vaSpace->WriteAt(&(machine->mainMemory[swapPageNum * PageSize]),
             PageSize,  physPageTable[swapPageNum].vaPageNum * PageSize);
        }
        int swapVpn = physPageTable[swapPageNum].vaPageNum;
        if (tempThread != NULL) {
        tempThread->space->pageTable[swapVpn].valid = FALSE;
        tempThread->space->pageTable[swapVpn].dirty = FALSE;
       }
        return swapPageNum;
}
```

LRUSwapPages 是用 LRU 的思想，遍历 physPageTable 找到 lastUsedTime 最小的页作为替换，并且如果要被替换的页是 dirty 的，则需要写回对应的地址空间。注意要先看该页所属的线程是哪个，然后如果该线程不为 NULL，即还未结束，则写回该线程的地址空间。若该线程没结束，还需要更新其页表，将对应的页表项置为无效。

第三部分

Lazy-loading：

其实之前的虚拟内存的实现，已经用到了 lazy-loading：在新建地址空间时，如果有空闲的物理页，则只把代码段写到空闲内存中，之后的数据页使用的时候才写入内存，即为 lazy-loading。这样的实现，使得用户程序可以用的内存，不在受物理页大小的限制，甚至可以无限大，也为多线程的实现创造了可能。具体的实现过程已经在上一部分叙述过了，就不在加以叙述了。

第四部分

Challenge 2：倒排页表

看到 Challenge 快哭了，，，Challenge 1 要在完成文件系统的基础上做，只能做 2 了。倒排页表，这意味着，我几乎要把之前写过的虚拟内存的实现，整个改一遍。。。

好吧，为了不把之前的改乱，就重新写一个版本吧。。。还好有之前的备份。

首先倒排页表可以是每个用户程序都有一个页表，但是考虑到所有用户程序共用一块物理内存，所以我就只用了一个整体的倒排表，既用于地址翻译，又用于内存管理，并且占用空间小。在 entry 中添加属性 lastUsedTime 用于 LRU 的置换算法，和 heldThreadID 指明该页内存中内容所属的线程号，用于支持多线程的处理。

考虑到之前虚拟地址空间的模拟是对于每个用户程序都单独新建一个文件作为其虚拟空间，但是如果同时运行很多用户程序的话，就需要很大的磁盘空间（当然有时候也是必需的）。这次我采用了一种不一样的处理，我在 Machine 类构造函数中新建一个固定大小的交换文件，我用的是 4 倍的内存大小，在 machine 类中增加一个维护该文件的 OpenFile 的指针 swapSpace。需要用到该交换空间的情况是，当内存满了的时候，需要在内存中选取一页去替换，我采用的是 LRU 算法，如果被选到的这一页脏位被设置了，说明被修改过，那么需要将这一页的内容保存在一个地方，这里就保存在了交换空间中。所以还需要增加对交换空间的维护，一个 128 位的 BitMap，和一个对于交换空间的倒排页表。

和之前的处理一样，在 AddrSpace 的构造函数中，当物理内存中有空闲块时，将代码段写入物理内存，并建立起对应关系。对 Translate 函数基本不做什么改变，只需要在 TLB 命中时，更新 tlb 中对应表项和倒排表中的对应表项的 lastUsedTime，在 writing 时，设置其 Dirty 位。由于每次线程切换下去时都会调用 SaveState 清空 tlb，所以每次 tlb 中有效的项都属于当前用户程序，所以不用增加判断。

当 TLB 未命中时，同样发生 PageFaultException，调用处理函数，然后调用 LRUSwapTLB 更新 tlb。LRU 算法和之前的基本一致，这里就不在叙述了。不同的是，在找该虚拟地址对应的页表项时，需要遍历倒排页表，找到有效的，虚拟页号相同的，且属于当前线程的页表项，把这个过程集成到了 GetPageEntry 函数中，如果没有找到匹配的则返回-1。如果返回的是-1，说明该虚页还没有调入内存，那么调用 AllocatePhysPage，为其分配物理页。

物理页分配，先用 BitMap 找是否有空页，有的话，则直接得到物理页号，否则调用 LRUSwapPage 置换一个物理页。被置换下来的物理页如果设置了 dirty 位，并且它所属的线程还未结束，则用交换区的 BitMap 找到交换区的一个空页将其写入，并在 swap 的页表中建立对应关系，若交换区满了则报错（所以这种处理应该有限制可运行的用户程序所用内存空间的大小）。之后将倒排页表中对应的物理页设为无效，为之后分配做准备。LRUSwapPage 返回一个可分配的物理页号。得到物理页号后，先要遍历 swap 区的倒排页表，看当前的地址是否在交换区内，如果在，则将其从交换区取出，并清空 BitMap 位和交换区倒排页表对应的页表项，如果不在交换区内，再判断其是否是代码段和初始化的数据段，如果是，则从目标代码文件中读出，写入物理内存，否则不写物理页。最后设置物理页对应的页表，并返回物理页号。至此，分配物理页已经完成，回到 LRUSwapTLB 中，更新 tlb。

```
int Machine::AllocatePhysPage(int badVAddr)
{
    unsigned int vpn = (unsigned) badVAddr / PageSize;

    int ppn = mBitMap->Find();

    if (ppn == -1)
        ppn = LRUSwapPage(vpn);
    int spn = GetSwapEntry(vpn);
    if (spn == -1) {
        if (badVAddr < (currentThread->space->noffH.code.size + currentThread->space->noffH.initData.size)) {
        currentThread->space->exe->ReadAt(&(machine->mainMemory[ppn*PageSize]), PageSize,
          currentThread->space->noffH.code.inFileAddr + vpn * PageSize);
      }
    }
    else {
      swapSpace->ReadAt(&(machine->mainMemory[ppn*PageSize]), PageSize, spn*PageSize);
      sBitMap->Clear(spn);
      swapTable[spn].valid = FALSE;
    }
    pageTable[ppn].valid = TRUE;
    pageTable[ppn].dirty = FALSE;
    pageTable[ppn].virtualPage = vpn;
    pageTable[ppn].physicalPage = ppn;
    pageTable[ppn].heldThreadID = currentThread->GetThreadID();
    return ppn;
}
```

```
int Machine::LRUSwapPage(int vpn)
{
    int minTime = pageTable[0].lastUsedTime;
    int i;
    int swapPageNum= 0;
    for (i = 0; i < NumPhysPages; ++i) {
            if (pageTable[i].lastUsedTime < minTime) {
                    minTime = pageTable[i].lastUsedTime;
                    swapPageNum = i;
             }
    }
    printf("Swap Page %d\n", swapPageNum);
    Thread* tempThread = threadsInfo[pageTable[swapPageNum].heldThreadID];
    if (tempThread != NULL && pageTable[swapPageNum].valid && pageTable[swapPageNum].dirty) {
        int spn = sBitMap->Find();
        ASSERT(spn != -1);
        swapTable[spn].virtualPage = pageTable[swapPageNum].virtualPage;
        swapTable[spn].physicalPage = spn;
        swapTable[spn].valid = TRUE;
        swapTable[spn].heldThreadID = currentThread->GetThreadID();
        swapSpace->WriteAt(&(machine->mainMemory[swapPageNum * PageSize]),
                    PageSize,  spn*PageSize);
    }
    pageTable[swapPageNum].valid = FALSE;
    pageTable[swapPageNum].dirty = FALSE;
    return swapPageNum;
}
```

由于现在只有一个全局的倒排页表了，所以在 AddrSpace 的 RestoreState 中就不能切换页表了，所以将这一部分注释掉，甚至可以注释掉 AddrSpace 中的 pageTable，以免发生混淆。

测试：

在测试之前，首先新增加了系统调用 Exit，在异常处理函数中增加代码如下：

```
else if ((which == SyscallException) && (type == SC_Exit)) {
    int exitStatus = machine->ReadRegister(4);
    printf("Exit %d\n", exitStatus);
    currentThread->Finish();
}
```

Exit 的语义是结束当前程序，这里为了测试正确性，将 Exit 的退出参数输出。

1. 对 **TLB** 置换算法的测试和对比

测试程序：

```
#include "syscall.h"

int A[32][32];

int main()
{
    int i, j;
    for (i = 0; i < 32; ++i)
        for (j = 0; j < 32; ++j)
            A[i][j] = i * j;
    Exit(A[10][10]);
}
```

对 32*32 的数组进行遍历赋值，先测试以行遍历：

LRU 置换算法：

```
Exit 100
TLB:  Hit 37660, Miss 133
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 31000, idle 148, system 1550, user 29302
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

FIFO 置换算法：

```
Exit 100
TLB:  Hit 37660, Miss 246
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 31000, idle 148, system 1550, user 29302
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

可以看出两者的退出时的值都为 100，说明执行结果是正确的。Hit 的次数都为 37660，说明有效的内存访问的次数是一致的，而 LRU 只有 133 次 Miss，FIFO 有 246 次。

按列遍历二维数组，得到的结果如下：

LRU：

```
Exit 100
TLB:  Hit 37660, Miss 1071
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 31000, idle 148, system 1550, user 29302
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

FIFO：

```
Exit 100
TLB:  Hit 37660, Miss 2599
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 31000, idle 148, system 1550, user 29302
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

同样两次的退出结果相同，Hit 的次数也相同，说明没有出错。然而两个算法 Miss 的次数都明显变大，分别为 1071 和 2599，也符合预期。

最后测试一下 matmult 程序：

LRU：

```
Exit 7220
TLB:  Hit 759799, Miss 66427
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 665400, idle 139, system 33270, user 631991
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

FIFO：

```
Exit 7220
TLB:  Hit 759799, Miss 67836
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 665400, idle 139, system 33270, user 631991
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

matmult 中同时有按行访问和按列访问二维数组，按列访问的话，不太符合空间局部性，所以两次 Miss 率大概分别为 8.7%和 8.9%，并没有差太多，但 LRU 稍微好点。

综上可以得出，LRU 的置换算法优于 FIFO 的置换算法。

所以之后的测试都是针对 LRU 算法的。

2. 单线程的正确性测试

上面对二维数组的遍历测试，其退出结果都为 100，也可以说明运行结果是正确的。

这次的测试主要是针对抗压和正确性测试，由于上面已经测试过了 matmult 程序，所以这里选择的测试程序是 test 中的 sort 函数。sort 的最坏情况下时间复杂度是 O(n^2)的，那么对于 1024 大小的数组的时间是百万级别的太耗时了，所以测试将数据大小改为了 256，结果如下：

```
Exit 255
TLB:  Hit 1522695, Miss 51983
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1289000, idle 96, system 64450, user 1224454
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

发现 Exit 退出时的结果好像不对啊，应该是 0 啊，仔细检查发现是 sort 函数本身写错了，内部循环 j 应该从 0 到 255-i，而不应该从 i 开始，正确的代码应该如下：

```c
#include "syscall.h"

int A[256];   /* size of physical memory; with code, we'll run out of space!*/

int
main()
{
   int i, j, tmp;

   /* first initialize the array, in reverse sorted order */
   for (i = 0; i < 256; i++)
      A[i] = 255 - i;

   /* then sort! */
   for (i = 0; i < 255; i++)
      for (j = 0; j < (255 - i); j++)
     if (A[j] > A[j + 1]) {   /* out of order -> need to swap ! */
        tmp = A[j];
        A[j] = A[j + 1];
        A[j + 1] = tmp;
         }

   Exit(A[0]);    /* and then we're done -- should be 0! */
}
```

结果如下：

```
Exit 0
TLB:  Hit 3017482, Miss 101256
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2554800, idle 172, system 127740, user 2426888
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

由于运行过程中的输出内容较多，所以只截取了最后结束的部分来说明执行后的结果，在执行过程中的输出结果大概都如下图所示：

```
VPN 5
Current Thread 0,  Read VPN 5 PPN 5
VPN 21
Current Thread 0,  Read VPN 21 PPN 6
VPN 5
Current Thread 0,  Read VPN 5 PPN 5
VPN 5
Current Thread 0,  Read VPN 5 PPN 5
VPN 5
Current Thread 0,  Read VPN 5 PPN 5
VPN 5
Current Thread 0,  Read VPN 5 PPN 5
VPN 21
Dirty Page 6
Current Thread 0,  Write VPN 21 PPN 6
```

3. 多线程测试

只测试了两个线程的同时运行：一个对 32*32 数组的遍历赋值 array，一个对大小为 32 的数组的赋值和排序 loop，时间片大小是 100，调度算法是之前线程调度算法，结果如下：

```
VPN 2
Current Thread 1,  Read VPN 2 PPN 2
VPN 2
Current Thread 1,  Read VPN 2 PPN 2
Ticks: 100, TimeInterrupt!
Current Thread:
array, 100
Ready list contents:
loop, 95
PageTable Address: 0x98a1d38
NumPages: 15
NoffHeader:
Code: 0x0, 0x28, 0x2f0
Data:  0xffffffff, 0xff7fc39e, 0x0
UninitDate: 0x2f0,  0x0,  0x80
hahhahaha
Thread 2 start to run
```

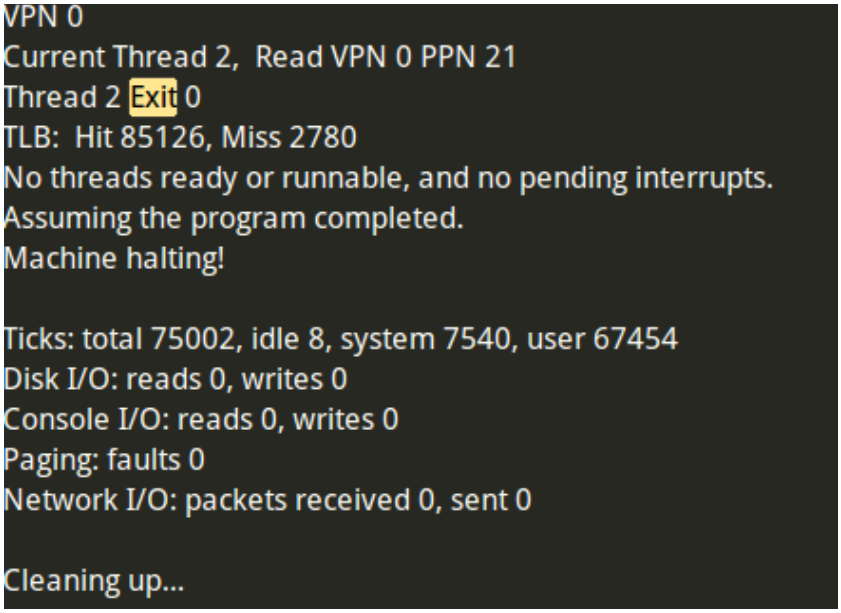
第一次时钟中断，Thread2 开始运行。

```
VPN 2
Current Thread 2,  Read VPN 2 PPN 7
VPN 14
Current Thread 2,  Read VPN 14 PPN 11
Ticks: 200, TimeInterrupt!
Current Thread:
loop, 95
Ready list contents:
array, 97
VPN 2
Current Thread 2,  Read VPN 2 PPN 7
VPN 2
Current Thread 2,  Read VPN 2 PPN 7
VPN 2
Current Thread 2,  Read VPN 2 PPN 7
```

```
VPN 0
Current Thread 1,  Read VPN 0 PPN 20
VPN 0
Current Thread 1,  Read VPN 0 PPN 20
Thread 1 Exit 100
Ticks: 65002, TimeInterrupt!
VPN 3
VPN 3
Current Thread 2,  Read VPN 3 PPN 14
VPN 14
VPN 14
Current Thread 2,  Read VPN 14 PPN 11
```

第二次时钟中断，由于优先级的原因，并没有发生切换。

Thread1 先运行结束退出。



Thread2 也运行结束，程序退出。两个程序退出的值都是正确的，也能说明程序正确执行了。

4. 倒排页表

对倒排页表的实现，测试了 matmult 和数组大小为 256 的 sort，测试都运行通过，由于运行结果是一样的，这里就不再粘贴结果。

【对于阅读代码类的 exercise，请对其中你认为重要的部分（比如某文件，或某个类、或某个变量、或某个函数……）做出说明。

对于要编程实现的 exercise，请对你增加或修改的内容作出说明。如果增加或修改了某个类，请写出这个类写在那个文件中，它的的功能是什么，你自己添加或修改的成员变量以及成员函数有哪些，它们的功能是什么；特别的，对于复杂的函数，请说明你的实现方法。不需要贴具体的实现代码。

要求：表述清楚即可， 可采用图表来辅助说明，不需要贴代码】

内容三：遇到的困难以及解决方法

困难 1

调试巨困难，当出现段错误或者非法指令的错误时，有的时候根本不能从出现错误的地方去找错误，它可能是在之前没有解决的错误，运行到此处才体现出来的。

解决方法：

1. 自己看代码，看一下是否有一些变量名写混的情况

2. 缩小测试用例的规模，并在代码运行中，打印各种有用的信息，然后把输出结果输出到文件中，自己模拟一下程序的运行，看是否相符，再去发现错误。

困难 2

……

【描述你在实习过程中遇到的困难，是与实习直接相关的技术方面的难题。突出说明你是如何攻克这些难关的。

要求：只需写一下有较深体会的困难。如果觉得整个过程都比较简单的话此部分可不用写。】

内容四：收获及感想

1. 对虚拟内存，TLB，以及用户程序的运行过程有了更详细，更直观的理解。

2. 整个对内存的管理中，都使用了很多宿主机的资源，比如把页表直接放在宿主机的内存等。如果是真实的操作系统的话，每个线程的页表都应该是属于该线程的用户空间的，甚至页表太大时，也是存在磁盘上的，当需要用到时才调到内存。在 nachos 中模拟的时候，就直接存在了宿主机的内存，并且发生切换时，直接将 machine 的页表指针指向了将要运行线程的页表，这些都依赖于宿主机的资源，使得整个实现过程也变得相对简单。

3. 在内存的管理中，还应该注重一些性能方面的东西，比如如何能减少内存页切换的次数，减少文件系统的使用，增大 TLB 命中率，减少物理资源的使用等。

【自己的收获，任何关于实习的感想，可以是技术方面的或非技术方面的，可随意发挥。

要求：内容不限，体裁不限，字数不限，多多益善，有感而发。】

内容五：对课程的意见和建议

……

【请写下你认为课程需要改进的地方，任何方面，比如进度安排、难易程度、课堂讲解、考核方式、题目设置……甚至如果你认为源代码哪里写得不好也欢迎提出。

各位同学反馈的信息对课程建设会有极大帮助。】

内容六：参考文献

操作系统课件

【我们希望大家不要使用复制粘贴来拼凑你的报告。详细地列出你在完成 lab 的过程中引用的书籍，网站，讲义，包括你咨询过的大牛们。

要求：诚实，格式尽量按照论文的要求，请参考"论文参考文献格式.doc"】