虚拟内存实习报告

姓名 张煌昭 学号 1400017707
日期 2017.11.13

目录

内容一：总体概述

本次 LAB 完成了对与 Nachos 系统虚拟内存的实现和改造优化。通过本次 LAB 较为深入地了解了操作系统虚存的相关概念，对于 TLB、页表、虚拟页和物理页等概念也进行了实现。

内容二：任务完成情况

任务完成列表（Y/N）

Exercise 1	Exercise 2	Exercise 3	Exercise 4	Exercise 5	Exercise 6	Exercise 7	Challenge 1	Challenge 2
Y	Y	Y	Y	Y	Y	Y	--	Y

具体 Exercise 的完成情况

第一部分. TLB 异常处理

Exercise1 源代码阅读。

- **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。**
- **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。**

用户程序执行的过程

用户在 code/userprog 目录下使用命令“./nachos -x USER_PROGRAM”执行用户程序，其中“USER_PROGRAM”为用户程序的路径；程序首先进入 code/threads/main.cc 中执行 main 函数，之后找到“-x”参数选项的入口 StartProcess()，该函数的定义位于 code/userprog/progtest.cc 之中；在 StartProcess()函数之中，打开该可执行文件，将其载入到地址空间 AddrSpace 之中，对地址空间进行必要的初始化后将其赋给当前执行的线程（此处为 main 线程），最后运行 machine->Run()开始运行用户程序。

地址空间 AddrSpace 类的定义位于 code/userprog/addrspace.h 和 code/userprog/addrspace.cc 之中，在该类中对各个线程的页表进行操作。构造函数之中新建可执行文件大小的页表，之后将可执行文件的 code 段和 initData 段全部装载于页表之中；RestoreState()函数将 machine 中的页表替换为该对象的页表，完成内存空间切换的工作。

Nachos 系统的 TLB 机制

Nachos 中目前并不支持 TLB 与页表共存（由于 code/machine/translate.cc 中 Translate()函数中对其进行了 ASSERT 检查，将其注释后可以使用），修改代码后才可以使用。

在 code/machine/translate.cc 中的 Translate()函数中，利用 TLB 和页表进行虚拟地址-物

理地址的转换。若 TLB 为空（说明没有使用 TLB，只使用了页表），则直接从页表中取 vpn 表项翻译物理地址即可，若 vpn 过大或 vpn 表项不可用则分别返回 AddressErrorException 和 PageFaultException；若 TLB 非空（说明使用了 TLB），则从 TLB 中查找虚拟页号为 vpn 的一项，若查找不到或该项不可用则返回 PageFaultException。

可以发现 TLB MISS 和页表缺页均会引起缺页异常（PageFaultException），因而对于缺页异常处理，需要两步进行，首先检查 TLB MISS，若 TLB 需要的页面不在页表之中，则再进行缺页异常处理，从磁盘中读取页面到页表之中，之后再加载到 TLB 中。

相关的代码文件

code/userprog/exception.cc 文件中定义了所有异常的处理函数 ExceptionHandler()，缺页异常和 TLB MISS 的处理在该函数中进行，此外所有系统调用也需要在该函数中进行实现。

code/machine/machine.h 和 code/machine/machine.cc 对 Nachos 的 MIPS 模拟器硬件进行定义和实现，其中具体包括了各种指令模拟、寄存器读写、内存读写的方法。

code/machine/translate.h 和 code/machine/translate.cc 对地址转换机制进行了实现，translate.h 文件中定义了页表项和 TLB 表项类；translate.cc 文件中实现了 Nachos 的 MIPS 模拟器的地址转换方法 translate()，该方法将虚拟地址转换为物理地址，并引起缺页异常和 TLB MISS 等异常。

code/userprog/progtest.cc 对用户程序的加载函数 StartProcess()进行了实现。

code/userprog/addrsapce.h 和 code/userprog/addrspace.cc 对地址空间进行定义，其本质是各个用户进程的页表，该页表在程序加载时完全加载，在之后的虚存的实现中，需要完成页表的不完全加载。

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）。

由于需要同时使用 TLB 和页表，因此注释掉 Translate()函数中的两个 ASSERT 检查，并在 code/userprog 中添加-DUSE_TLB 参数宏定义使用 TLB。此时，所有地址翻译都从 TLB 中进行查找，若查找不到对应的 vpn（即发生了 TLB MISS），则抛出 PageFaultException，等待异常处理函数进行处理。因此，对 TLB MISS 的处理可以全部在 ExceptionHandler()函数中完成。

首先确定 TLB MISS 的情况在 ExceptionHandler()函数中如何捕获，由于 TLB MISS 抛出的是 PageFaultException，因此需要捕获该异常，此外由于 TLB MISS 是在 TLB 非空的情况下抛出的，因此可以用 TLB 是否为空来判断是否是 TLB MISS。

捕获 TLB MISS 之后，计算 vpn，由于此时页表中页表项号就是 vpn，并且全部内存都加载在页表之中，因此可以直接读取 vpn 页表项放入 TLB。下面只剩下 TLB 的替换，替换算法在下一部分完成，假设此时已经完成了替换算法，那么直接将读出的页表项放入 TLB 中被替换的表项即可。

与下一部分 TLB 置换算法一起进行测试。

Exercise3　TLB 置换算法

为 TLB 机制实现至少两种置换算法，通过比较不同算法的置换次数比较算法的优劣。

实现了 FIFO，随机和 LRU 三种替换算法，详细情况如下

TLB FIFO 替换算法

TLB 中表项按先进先出的规则进行替换，将 TLB 按队列的形式进行组织，新加入的表项从数组的最后一项进入 TLB，最终从第 0 项退出；每有新表项进入时，TLB 中所有表项向下标减一的方向前进一步。宏定义 TLB_FIFO 允许使用 TLB FIFO 替换算法。

TLB 随机替换算法

TLB 中表项按随机的规则进行替换，不需要特殊的组织形式，每次决定被替换的表项时，利用 Random()函数生成伪随机数最为被替换的表项。宏定义 TLB_RANDOM 允许使用 TLB 随机替换算法。

TLB LRU 替换算法

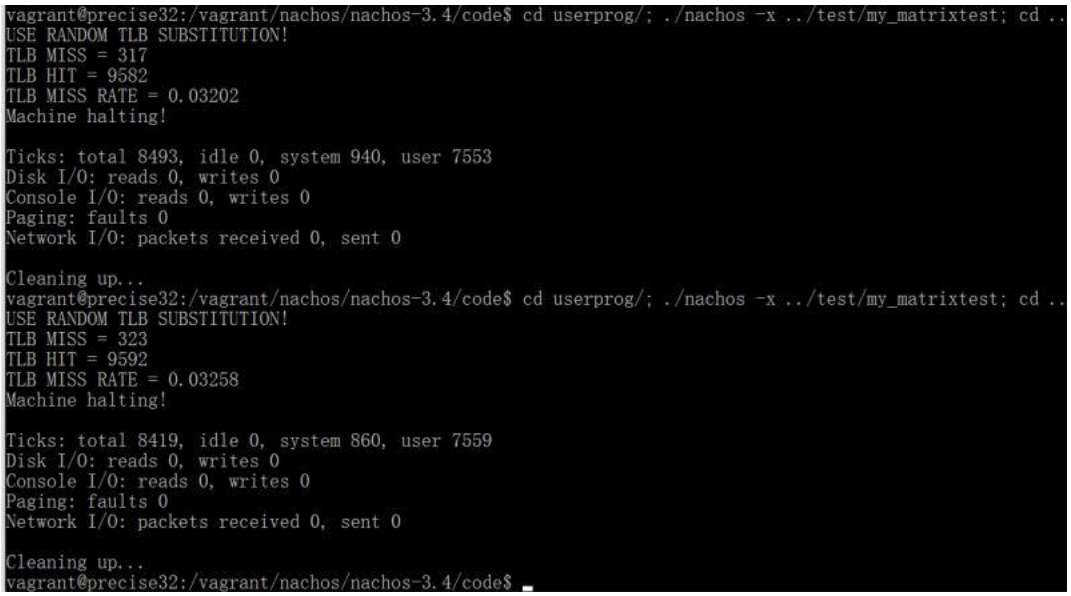
TLB 中表项按最不常使用的规则进行替换，需要添加一个 TLB 表项使用计数器，在 machine 类中添加 LRU_timer 数组进行计数。每次替换时，在 LRU_timer 数组中找到最大一项作为被替换表项，之后所有计数加一，被替换的表项的计数置零；此外在 translate 函数中也要对 LRU_timer 在 TLB HIT 的情况下进行维护，将命中表项的计数置零，其余的计数加一。所有添加的内容均在宏定义 TLB_LRU 之下，该宏定义允许使用 TLB LRU 替换算法。

在 code/test 中仿照其余程序添加测试代码 my_matrixtest.cc 用于 TLB 的测试，该代码中对一个二维数组进行按列遍历赋值。由于目前还没有实现所有的系统调用，因此测试代码退出时，使用已有的 Halt()系统调用。向 Translate()方法中 TLB MISS 和 TLB HIT 各添加一个计数，并在 ExceptionHandler()中的 Halt 系统调用中打印这两个计数。

使用三种替换算法的测试结果如下。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code$ cd userprog/; ./nachos -x ../test/my_matrixtest; cd ..
USE FIFO TLB SUBSTITUTION!
TLB MISS = 261
TLB HIT = 9546
TLB MISS RATE = 0.02661
Machine halting!

Ticks: total 8397, idle 0, system 900, user 7497
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...
vagrant@precise32:/vagrant/nachos/nachos-3.4/code$
```

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code$ cd userprog/; ./nachos -x ../test/my_matrixtest; cd ..
USE LRU TLB SUBSTITUTION!
TLB MISS = 162
TLB HIT = 9495
TLB MISS RATE = 0.01678
Machine halting!

Ticks: total 8198, idle 0, system 800, user 7398
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...
vagrant@precise32:/vagrant/nachos/nachos-3.4/code$
```



FIFO，随机和 LRU 三种 TLB 替换算法的 TLB MISS 率分别为 0.027，0.017 和 0.032，因此可以大致得出结论——在局部性原理的保护下，LRU 替换算法是这三种替换算法中的最佳算法。

第二部分. 分页式内存管理

目前 Nachos 系统中，类 Class Thread 的成员变量 AddrSpace* space 中使用 TranslationEntry* pageTable 来管理内存。应用程序的启动过程中，对其进行初始化；而在线程的切换过程中，亦会对该变量进行保存和恢复的操作（使得类 Class Machine 中定义的 Class Machine::TranslationEntry* pageTable 始终指向当前正在运行的线程的页表）。

Exercise 4 内存全局管理数据结构

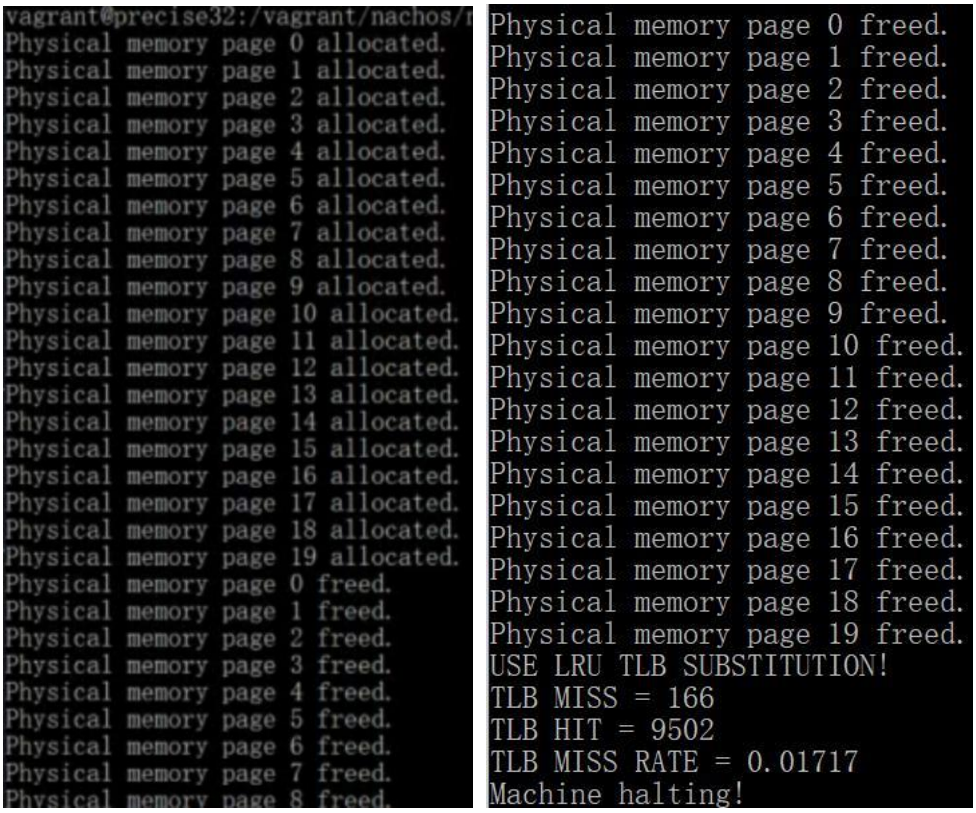
设计并实现一个全局性的数据结构（如空闲链表、位图等）来进行内存的分配和回收，并记录当前内存的使用状态。

使用位图来记录内存的分配情况，具体的实现为利用 code/userprog/bitmap.h 和 code/userprog/bitmap.cc 中实现的位图 BitMap。向 machine 类添加成员 bitmap 来记录物理内存页的分配情况，并向 machine 类添加两个成员方法 allocPhysPage()和 freePhysPage()用于分配和回收内存页面。

allocPhysPage()方法调用 bitmap->Find()方法寻找空闲内存页面，若找到的返回该页面号，否则返回-1；freePhysPage()方法将整个页表中所有有效且已分配的页面从 bitmap 中释放。

内存分配是在 AddrSpace 的构造函数中完成的，因此其中的页表项的物理页号应该使用 machine->allocPhysPage()方法进行分配，并检查该页号是否非法（-1）；页面回收应该在用户进程结束时完成，因此在 Halt 系统调用中调用 machine->freePhysPage()对内存页面进行释放。

使用之前编写的测试代码进行测试如下，可以看到用户程序执行前分配了 19 个物理页面，执行结束调用 Halt 方法后对这 19 个物理页面进行了回收。



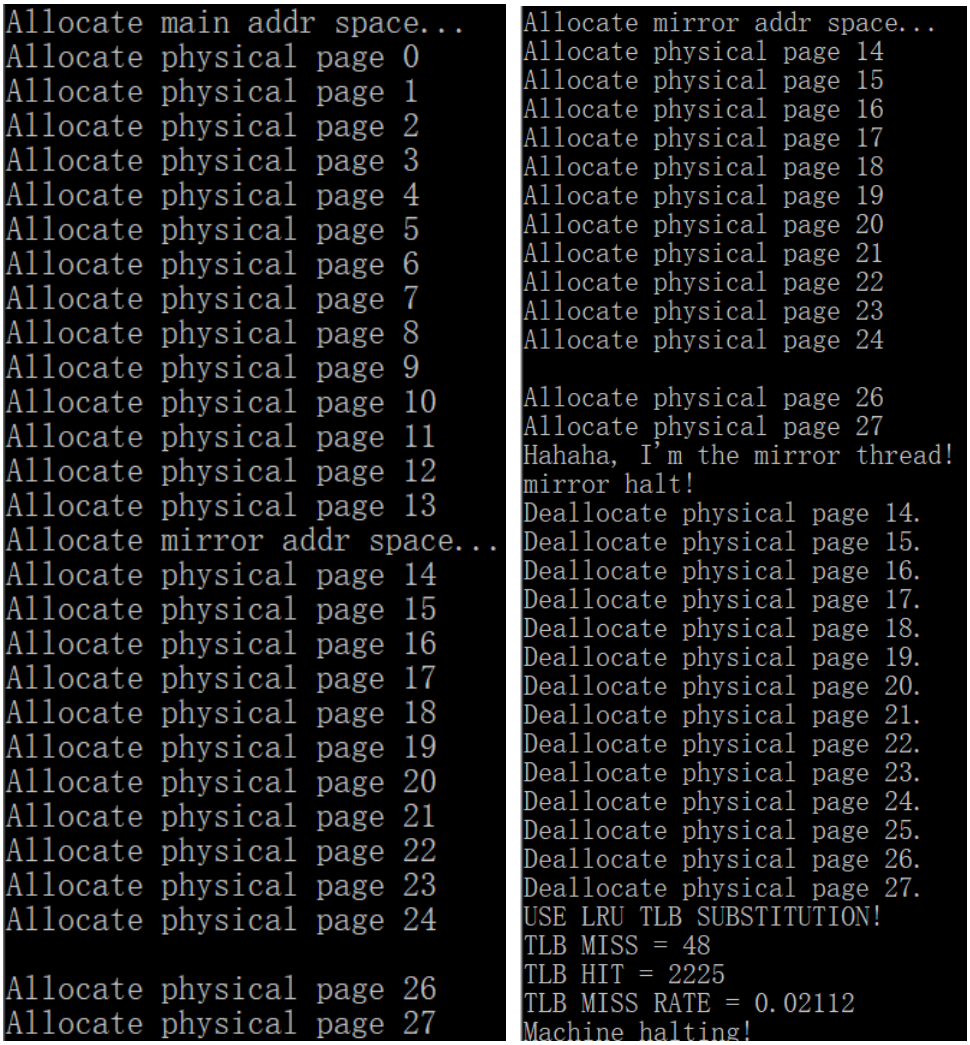
Exercise 5 多线程支持

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。

目前的 Nachos 系统在 AddrSpace 类的构造函数中，加载地址空间时直接连续地写入 code 和 initData 两段，即其默认线程的地址空间在物理内存中也是连续的，这是导致其内存中只能同时存在一个线程的原因。将其改为按页面写入 code 和 initData 即实现了线程地址空间在物理内存中页内连续，各页不连续的目标。

具体的操作为在写入 code 和 initData 时按 byte 向其应该归属的页面写入，这样虽然看起来比较繁琐，但实际上是一种简单的按页面载入的方法。此外由于需要支持多线程切换，TLB 在切换后一定失效，需要在 code/userprog/addrspace.cc 中的 SaveState()函数中将页表项全部置为无效。

测试时在 StartProcess 中创建一个镜像线程“mirror”，mirror 线程的地址空间完全拷贝自 main 线程，之后 mirror 线程运行至 Halt 停机。运行 my_matrixtest 程序，结果如下。



首先 main 线程加载内存空间至物理内存的 0-13 页，之后切换至 mirror 线程加载内存空间至物理内存的 14-27 页，mirror 线程执行，打印其信息，之后 mirror 线程运行至 Halt()，回收其内存空间，打印 TLB 信息，停机。

由于 main 线程实际并未运行，而 mirror 线程运行至 Halt()，Halt()系统调用直接使得系统停机，因此 main 线程的内存空间并未回收。这是一种不优雅的测试方法，但迫于没有系统调用，只能这样进行测试。

Exercise 6 缺页异常处理

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。

缺页异常发生时，同 TLB MISS 一样抛出 PageFaultException，由于已经定义使用 TLB 了，因此会默认进入 TLB MISS 的处理函数，又由于该页并不在页表之中，因而 TLB MISS 处理并不能找到需要的页表项；因此 PageFaultException 和 TLB 找不到需要的页表项即为缺页异常发生。缺页异常发生时，首先需要从磁盘中找到所需的页面并将其加载到物理内存之中（需要考虑替换算法），之后更新页表，最后更新 TLB。所做的更新如下。

首先需要构造磁盘上的存储，一般的会直接从打开的可执行文件进行加载，但为了简化

问题，我选择在 AddrSpace 的构造函数中，将可执行文件按照内存加载形式存放在一个额外的“虚存文件”之中。由于每个用户程序都会有一个自己的“虚存文件”，因此需要对其进行分配，在 machine 中添加新的 BitMap vmbitmap 用于分配和释放“虚存文件”，并添加对应的分配和释放方法。

之后修改 ExceptionHandler 中对缺页异常进行处理。缺页异常的标志为 TLB 为空或在页表中为找到所需的页表项，满足其一便进入缺页异常处理；缺页异常处理时需要从相应的“虚存文件”中利用 vpn 找到所需页面，将其载入到合适的物理内存位置，并修改页表和 TLB；TLB 利用在 TLB MISS 处理中替换算法找到的位置进行替换，页表利用页表替换算法进行替换。

最后说明我使用的页面替换算法，我使用的是“零页替换”，即寻找 ppn 为 0 的页面进行替换，这在多线程场景下是并不正确的，但目前仅有单线程，而这种替换是最简单易行的替换算法；寻找到 ppn=0 的页面之后，需要检查 dirty 位，若 dirty 位为真则需要将该页面写回到“虚存文件”之中。

以上所有更新均在宏定义 USE_VMEM 之下，测试部分与 Lazy Loading 组合，见下一部分。

第三部分. Lazy Loading

Exercise 7 实现 Lazy-loading

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。

在上一部分中已经实现了“虚存文件”，但在分配内存空间时不仅分配了虚存文件，也将其放入了物理内存之中，这是没有必要的；可以仅仅将其放在虚存文件之中，不分配物理内存，而利用 PageFault 将所需的页面即时调入内存之中进行使用。

因此所需的改动即为仅仅将可执行文件加载在虚存文件中（而不再向内存加载），使用 code/test/my_matrixtest 和 code/test/sort 程序进行测试，结果如下图。页面并没有在程序执行前完全加载进内存中，而是按照所需分步加载至内存中（一些在程序执行过程中不需要的页面就没有进行加载）。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code$ cd userprog/; ./nachos -x ../test/my_matrixtest; cd ..
Page fault... Load virtual page 0 into physical page 0.
Page fault... Load virtual page 1 into physical page 1.
Page fault... Load virtual page 13 into physical page 2.
Page fault... Load virtual page 2 into physical page 3.
Page fault... Load virtual page 3 into physical page 4.
Page fault... Load virtual page 4 into physical page 5.
Page fault... Load virtual page 5 into physical page 6.
USE LRU TLB SUBSTITUTION!
TLB MISS = 87
TLB HIT = 2247
TLB MISS RATE = 0.03728
Machine halting!
```

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code$ cd userprog/; ./nachos -x ../test/sort; cd ..
Page fault... Load virtual page 0 into physical page 0.
Page fault... Load virtual page 1 into physical page 1.
Page fault... Load virtual page 21 into physical page 2.
Page fault... Load virtual page 2 into physical page 3.
Page fault... Load virtual page 5 into physical page 4.
Page fault... Load virtual page 6 into physical page 5.
Page fault... Load virtual page 7 into physical page 6.
Page fault... Load virtual page 8 into physical page 7.
Page fault... Load virtual page 9 into physical page 8.
Page fault... Load virtual page 10 into physical page 9.
Page fault... Load virtual page 11 into physical page 10.
Page fault... Load virtual page 12 into physical page 11.
Page fault... Load virtual page 13 into physical page 12.
Page fault... Load virtual page 3 into physical page 13.
Page fault... Load virtual page 4 into physical page 14.
USE LRU TLB SUBSTITUTION!
TLB MISS = 118983
TLB HIT = 1562648
TLB MISS RATE = 0.07075
Machine halting!
```

Challenge 1 线程 SUSPENDED 状态

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。

我并未对此 Challenge 进行实现，下面仅对思路进行叙述。

挂起状态下，线程的内存空间被全部转移至磁盘，从而可以释放出物理内存空间给别的线程使用，而 TCB 可以保留在系统内存空间中，以便之后从磁盘装载回被挂起的线程。需要的更新如下。

由于线程 TCB 中已经有指向 AddrSpace 的指针，并且在 AddrSpace 中已有指向虚存文件的指针，因此可以直接在 Thread 类中添加向虚存文件装载和读取的方法。之后添加 Suspend() 方法，将内存空间装载至虚存文件，并释放页表中全部页面，最后将线程状态置为 Suspended_Ready 或 Suspended_Blocked；Unsuspend() 方法将 Suspended_Ready 或 Suspended_Blocked 状态置为 Ready 或 Blocked，由于 Lazy-loading 的存在，不需要重新加载内存。

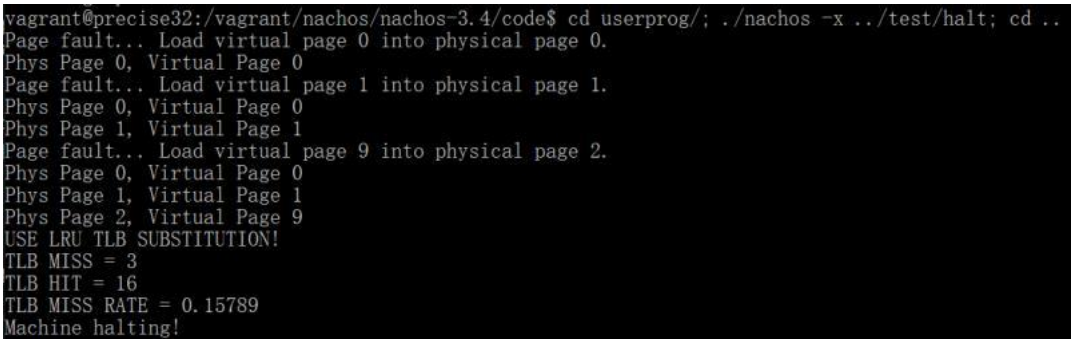
Challenge 2 倒排页表

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。

之前页表按照 vpn 为下标的方式进行组织，而倒排页表按照 ppn 为下标的方式进行组织，各个用户线程的页表空间是相同的空间（都为物理内存空间），而页表项互斥（没有任何两个页表的同一个 ppn 页表项的 valid 位同为真）。对代码的更新如下。

使用 ppn 作为下标对 AddrSpace 构造中的页表进行初始化，并利用 ppn 作为下标进行页表项替换和加载；此外在 TLB 替换时需要额外查找虚页号为 vpn 的页面（此前可以直接用 vpn 作为下标找到）。

使用 code/test/halt 程序进行测试，结果如下图。页表分配按照 ppn 下标进行分配。



内容三：遇到的困难以及解决方法

1. Nachos 中 TLB 和页表无法共存

在 Excersice 1 之中，修改 Makefile 添加 USE_TLB 宏定义后会引起 Translate()中的 ASSERT 检查报错。TLB 和页表无法共存的这一规定比较难以理解，在这两个断言被卡壳很长时间，之后决定按照思路使得 TLB 和页表共存，将其注释后可以急需正常进行。

2. Makefile 修改后并不会重新编译

在实验中使用多个宏定义进行控制（比如选择 TLB 替换算法），而仅仅更改 Makefile 中宏定义，不修改代码的话，重新 make 并不会引起使得工程重新编译。引起采用不同 TLB 替换算法，重新 make 后结果不变的错误。解决方法为修改 make all，将 make clean 作为先导，使得每次 make 都会重新编译所有文件。

3. 多线程支持无法进行测试

在完成 Exercise 5 后，希望进行测试，而此时并未实现 Fork 系统调用，并没有办法在用户程序代码中 Fork 得到多线程。这使得该测试进入尴尬的境地，最终通过选择上述的不优雅的方法在 StartProcess()中分裂出 mirror 线程进行测试。

4. 使用磁盘实现虚存

由于需要将虚存放置在磁盘之中，需要使用下一次 LAB 才需要的文件系统。

内容四：收获及感想

机制与策略分离的想法

在完成本次 LAB 的过程中采用了机制与策略分离的想法，提供 TLB、页表等机制，和不同的 TLB 替换算法等策略，通过宏定义的方式选择策略进行编译。对于“操作系统应将机制与策略分离”这一想法有了进一步的理解和体会。

内容五：对课程的意见和建议

暂无。

内容六：参考文献

暂无。