虚拟内存实习报告

姓名 马少鹏　　学号 1500012893

日期 2018.4.17

目录

内容一：总体概述

本实习希望通过修改 Nachos 系统平台的底层源代码，达到“实现虚拟存储系统”的目标。

内容二：任务完成情况

任务完成列表（Y/N）

Exercise1	Exercise2	Exercise3	Exercise4	Exercise5	Exercise6	Exercise6	Challenge1
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

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

nachos 执行用户程序的过程：

在 StartProcess 中：首先打开一个可执行文件，如果打开成功，为其申请地址空间，并将地址空间赋给当前线程地址空间，关闭文件。初始化用户水平的 cpu 寄存器，加载页表寄存器，用于上下文切换。然后调用 machine->Run()跳转到用户程序执行。

其中与内存相关的就是 space->RestoreState()，加载页表寄存器。

machine.h(cc)、translate.h(cc)、exception.h(cc) 理解当前 Nachos 系统所采用的 TLB 机制和地址转换机制。

TLB 机制：

Translate.h 中定义了类 TranslationEntry，其中的成员变量记录了内存虚拟页号对应的物理页号。以及 valid,readOnly,use,dirty 的信息。

TLB 的初始化在类 Machine 的构造函数中完成，申请 TLB 空间并且初始化 valid 为 FALSE。

TLB 的使用：translate.cc 中 Machine::translate 中，遍历 TLB，如果有与目标相同的虚拟页号 vpn，则 TLB 命中。否则 TLB miss，返回缺页异常 PageFaultException。

地址转换机制：

translate.cc 中 Machine::translate 中，首先根据虚拟地址得到虚拟页号 vpn 和偏移量 offset，然后通过 Pagetable 或 TLB 得到 vpn 对应的物理页号 ppn，其中虚拟页号就是页表的索引。然后根据 ppn 和 offset 计算得到物理地址，完成地址转换。

Exercise2 TLB MISS 异常处理

源代码中，当 TLBmiss 时，发出 PageFaultException。当 Pagetable 中找不到目标项时，也是这个异常。所以当检测到 PageFaultException 时，进行判断：if(machine->tlb == NULL)，如果不为 NULL，即发生了 TLBmiss，就调用 TLBmiss 处理程序。该处理程序如下：

```
87  void
88  TlbMissHandler()
89  {
90      int virtAddr,vpn,offset;
91      virtAddr = machine->ReadRegister(BadVAddrReg);
92      machine->fifo(virtAddr);
93      //machine->lru(virtAddr);
94  }
95
```

处理程序需要在 Pagetable 中找到目标项，然后进行 TLB 的替换。替换的算法在 exercise3 中实现。

Exercise3 置换算法

咋 machine 类中实现算法，为 machine 的成员函数。

先进先出算法（FIFO）：

遍历 TLB，如果找到未使用的位置，即 valid == FALSE，直接替换。否则，删除第一个项，其他所有项向前移一位，然后替换到最后一个项。

LRU 算法：

为每个项维护一个时间戳：在 TranslationEntry 类中增加成员变量 int last_use_time; 当程序在 TLB 中访问该项时，就把该项修改为当前时间：totalTicks。

遍历 TLB，如果找到未使用的位置，即 valid == FALSE，直接替换。否则，找到上次使用时间最早的项进行替换。具体实现如下：

```
246  void Machine::lru(int addr)
247  {
248      //for(int i=0;i<TLBSize;++i) printf("tlb%d valid: %d page: %d\n",i,tlb[i].va
249      int tmp = tlb[0].last_use_time;
250      int vpn = addr/PageSize;
251      int target = 0;
252      for(int i=0; i<TLBSize; ++i)
253      {
254          if(tlb[i].valid == FALSE)
255          {
256              target = i;
257              break;
258          }
259          if(tlb[i].last_use_time < tmp)
260          {
261              target = i;
262              tmp = tlb[i].last_use_time;
263          }
264      }
265      if(tlb[target].valid == TRUE)
266          pageTable[tlb[target].virtualPage] = tlb[target];
267      tlb[target] = pageTable[vpn];
268      tlb[target].last_use_time = stats->totalTicks;
269      //printf("tlb_lru replaces %d times\n",++tlb_count_lru);
270      //for(int i=0;i<TLBSize;++i) printf("tlb%d valid: %d page: %d after\n",i,tlb
271  }
```

二者的效率比较;

测试程序使用 test 文件夹下的 matmult.c 文件，该程序进行了矩阵乘法，矩阵大小为 Dim*Dim。Dim 本来设置为 20，但是这样会导致可执行文件的 size 超过 nachos 设置的物理内存页数 32，所以修改 Dim 为 5。分别用两种替换算法执行该程序，记录 TLb 访问次数和命中次数。结果如下：

FIFO：

```
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -x ../test/matmult
Machine halting!

total TLB visit: 15556 total TLB hit: 14480

Ticks: total 12630, idle 0, system 10, user 12620
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...
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog$
```

访问数：15556 命中数：14480 命中率：0.93083

LRU：

```
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -x ../test/matmult
Machine halting!

total TLB visit: 15439 total TLB hit: 14435

Ticks: total 12558, idle 0, system 10, user 12548
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...
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog$
```

访问数：15439 命中数：14435 命中率：0.93450

可以看出，LRU 的效率优于 FIFO。

继续使用 test 文件夹下的 sort 文件作为测试程序，结果如下：

FIFO：

```
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog$ ./na
chos -x ../test/sort
Machine halting!

total TLB visit: 26308700 total TLB hit: 24719407

Ticks: total 21036091, idle 0, system 10, user 21036081
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...
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog$
```

访问数：26308700 命中数：24719407 命中率:0.93959

LRU:

```
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog$ ./na
chos -x ../test/sort
Machine halting!

total TLB visit: 25817043 total TLB hit: 24473595

Ticks: total 20790246, idle 0, system 10, user 20790236
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...
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog$
```

访问数：25817043 命中数：24473595 命中率：0.94796

同样可以看出：LRU 优于 FIFO。

综上，LRU 优于 FIFO。

第二部分 分页式内存管理

Exercise4 内存全局管理数据结构

定义一个全局的数据结构 BitMap *memory = new BitMap(NumPhysPages)，采用 bitmap.h 中定义的位图数据结构。大小初始化为物理页数。

在 AddrSpace 类的构造函数中，分配物理页时，不再分配和虚拟页号相同的物理页号，而是改为用 BitMap 类的 Find()方法。在 memory 中寻找第一个没有被分配的物理页。

内存的回收：因为执行用户程序时，machine->Run()不会返回，地址空间是通过系统调用 Exit 释放的，所以修改 exception.cc 中的 ExceptionHandler 函数，增加对 Exit 系统调用的处理：调用 currentThread->finish()方法。并修改 Thread 类的 Finish 方法，在 Finish 中回收内存，即调用 BitMap 的 clear 方法清空 memory 即可（需要判断当前线程是否是用户程序）。需要在 Thread 类中记录该线程占用的物理页，所以在 addrspace 类增加成员变量： int *pPage。若该线程占有物理页 i，则 pPage[i] = 1。

运行 test 中的 Halt.cc，把最后调用的 Halt()改为 Exit(0)，使用这个测试程序的原因是使用内存比较少，其他的测试程序使用内存大，不方便截图。结果如下：

```
msp@msp-virtual-machine: ~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog$ ./na
chos -x ../test/halt
physicalPage 0 is allocated!
physicalPage 1 is allocated!
physicalPage 2 is allocated!
physicalPage 3 is allocated!
physicalPage 4 is allocated!
physicalPage 5 is allocated!
physicalPage 6 is allocated!
physicalPage 7 is allocated!
physicalPage 8 is allocated!
physicalPage 9 is allocated!
physicalPage 10 is allocated!
physicalPage 0 is clear!
physicalPage 1 is clear!
physicalPage 2 is clear!
physicalPage 3 is clear!
physicalPage 4 is clear!
physicalPage 5 is clear!
physicalPage 6 is clear!
physicalPage 7 is clear!
physicalPage 8 is clear!
physicalPage 9 is clear!
physicalPage 10 is clear!
Machine halting!

total TLB visit: 21 total TLB hit: 18

Ticks: total 26, idle 0, system 10, user 16
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...
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog$
```

因为只有一个用户程序在执行，所以目前物理页号仍和虚拟页号相同，按顺序分配。当多线程时，可以看出不同。

Exercise5 多线程支持

首先，目前 Nachos 只支持单个线程同时在内存中，在 AddrSpace 的构造函数中，在拷贝代码段和数据段到内存之前，调用：bzero(machine->mainMemory, size)清空了内存，所以每次有新的用户程序时，就清空内存，则内存中只能有一个线程。

现在注释掉这行语句，并修改拷贝数据的代码如下：

```
123   // lab4 copy in the code and data segment into memory
124   if (noffH.code.size > 0) {
125           DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
126               noffH.code.virtualAddr, noffH.code.size);
127           int code_address = noffH.code.inFileAddr;
128           for(int i=0;i<noffH.code.size;++i)
129           {
130               int vpn = (noffH.code.virtualAddr + i) / PageSize;
131               int offset = (noffH.code.virtualAddr + i) % PageSize;
132               int ppn = pageTable[vpn].physicalPage;
133               int physical_address = ppn * PageSize + offset;
134               executable->ReadAt(&(machine->mainMemory[physical_address]),1, code_address++);
135           }
136       }
```

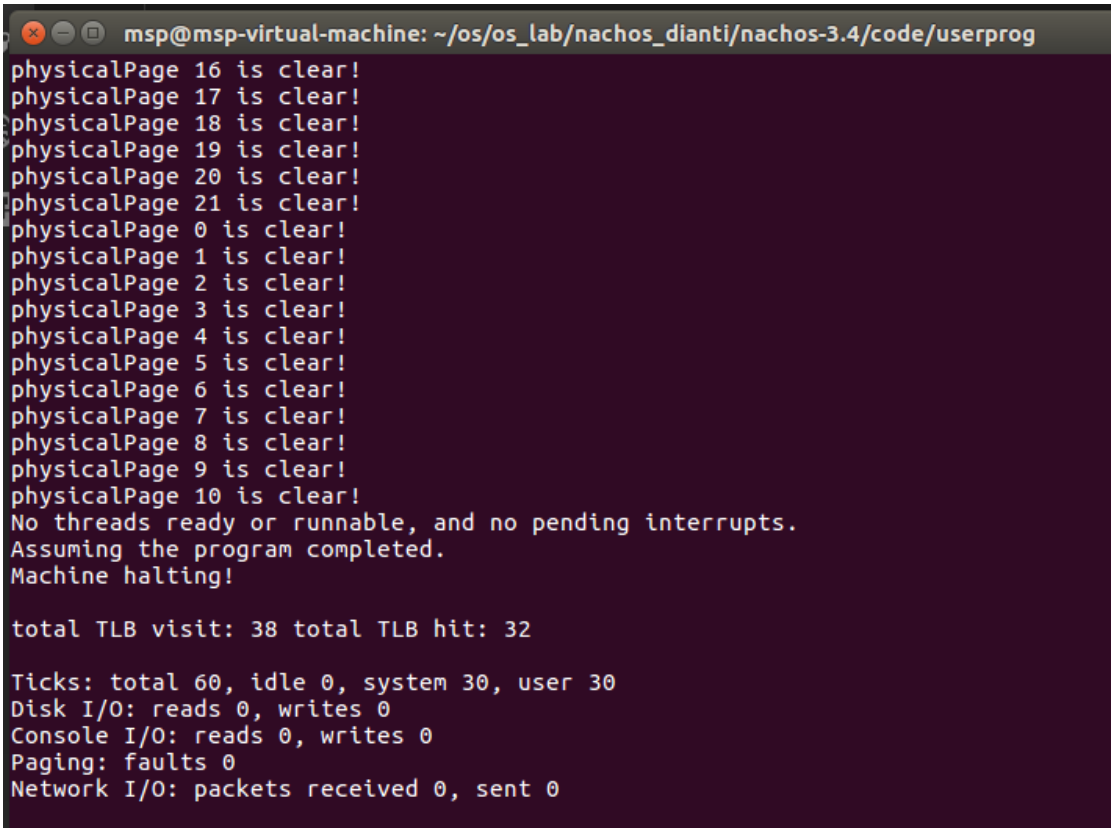
因有多个线程之后，虚拟页号与物理页号并不是相同的了。不同的物理页可能对应相同的虚拟页，因为它们可能对应不同线程的虚拟页。所以现在需要根据虚拟地址计算得到物理地址，并按字节拷贝到内存中。

对 TLB 的支持：修改 AddrSpace 类的 SaveState 函数，在这个函数中将 TLB 所有项的 vald 置为 FALSE。阅读代码 scheduler.cc 中的 Run()，可以发现，在线程切换上下文前，如果是用户程序的切换，会调用 SaveState 方法，所以在这里，我们清空 TLB。

为了测试修改后的代码：修改 progtest.cc 中的 StartProcess 函数，在之前打开一个可执行文件的基础上，再打开另一个可执行文件（这里打开"halt"），并将该用户程序加载到一个新创建的线程上，并将这个线程的优先级设置为 129（以 lab2 实现基于优先级的抢占式进程调度为基础，主线程的优先级默认为最高 128，这里为了测试，将新创建的线程的优先级设置为 129）。

运行结果如下：

```
Cleaning up...
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog$ ./na
chos -x ../test/halt
physicalPage 0 is allocated!
physicalPage 1 is allocated!
physicalPage 2 is allocated!
physicalPage 3 is allocated!
physicalPage 4 is allocated!
physicalPage 5 is allocated!
physicalPage 6 is allocated!
physicalPage 7 is allocated!
physicalPage 8 is allocated!
physicalPage 9 is allocated!
physicalPage 10 is allocated!
physicalPage 11 is allocated!
physicalPage 12 is allocated!
physicalPage 13 is allocated!
physicalPage 14 is allocated!
physicalPage 15 is allocated!
physicalPage 16 is allocated!
physicalPage 17 is allocated!
physicalPage 18 is allocated!
physicalPage 19 is allocated!
physicalPage 20 is allocated!
physicalPage 21 is allocated!
thread main id:0 yield to thread halt id: 1
thread name: halt id: 1 is running!
physicalPage 11 is clear!
```



如图，主线程执行了 halt，同时创建线程 halt 执行 halt。首先分配给主线程内存物理页 0-10，然后分配给 halt 线程内存物理页 11-21。Halt 线程创建后，由于优先级更高，此时发生抢占式调度，主线程让出 cpu，halt 线程执行直到结束。结束后释放内存，然后主线程继续执行到结束，释放内存。可以看到：halt 线程后创建并分配内存，并且先结束并回收内存，说明多线程机制实现成功。

Exercise6 缺页中断处理

在之前处理 PageFaultException 时，只对 TLB miss 的情况进行了处理，即编写了 TlbMissHandler 函数，而 PageFaultHandler 函数只是写了直接结束。现在重写这个函数，目前只实现了 LRU 替换算法。

当发生缺页中断时，首先调用 memory 的 Find 方法寻找空闲的物理页，如果找到，则直接从磁盘中拷贝需要的虚拟页到该物理页中。如果所有物理页已经被占用，则进行 LRU 替换。具体实现：

为每个 pageTable 表象维护一个变量 last_use_time。之前实现 TLB 的时候已经定义过这个变量，而 Nachos 中 TLB 和 pageTABLE 的表项是同一种数据结构，所以可以直接使用。每次访问某个表项时，就把 last_use_time 设置为当前系统时间。需要替换时，遍历 pageTable，寻找 last_use_time 最小，也就是上次使用时间最早的表项，将这个表项指向的虚拟页作为牺牲页，写回牺牲页，并把新的虚拟页加载到牺牲页所在的物理页。

测试：由于当前 Nachos 在用户进程创建时已经为程序分配了所有内存并初始化了页表，所以不会发生缺页中断。在后面的 lazy loading 中则会有缺页异常需要页面替换，所以测试工作放在 lazy loading 部分一并实现。

第三部分

Exercise7 lazy-loading

在 addrspace 类中增加成员变量 int code_address。用来记录可执行文件代码段的起始位置，用于加载可执行文件时的寻址，

修改 addrspace 类的构造函数：

创建页表后并不直接分配，而是将所有表项设置为无效（valid = FALSE），也不加载可执行文件。如图：

```
89  // lab4
90      code_address = noffH.code.inFileAddr;
91      pPage = new int[NumPhysPages];
92      for(int i=0;i<NumPhysPages;++i) pPage[i] = 0;  // for memory recovery
93      pageTable = new TranslationEntry[numPages];
94      for(int i=0;i<numPages;++i)
95      {
96  //printf("i: %d\n",i,NumPhysPages);
97          pageTable[i].virtualPage = -1;
98          pageTable[i].physicalPage = -1;
99          pageTable[i].valid = FALSE;
100         pageTable[i].use = FALSE;
101         pageTable[i].dirty = FALSE;
102         pageTable[i].readOnly = FALSE;
103         pageTable[i].last_use_time = 2147483647;  // need big enough
104     }
```

如图，页表的大小仍然与用户程序的地址空间成正比。其中，code_address 初始化为可执行文件的代码段的起始地址（为虚拟地址）。

然后修改 pagefault_lru()函数，这是 LRU 页面替换算法的实现。因为代码改为在创建用户线程时不分配内存，所以程序一开始执行就会发生缺页异常，所以在缺页异常处理函数中为线程分配内存，具体分配的就是线程需要访问的地址所在的那一页，将这一页加载到物理页中。具体代码如下：

```
315 // lab4
316 void pagefault_lru(int addr, char *filename)
317 {
318     NoffHeader noffH;
319     OpenFile *executable = fileSystem->Open(filename);
320
321     executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
322     if ((noffH.noffMagic != NOFFMAGIC) &&
323         (WordToHost(noffH.noffMagic) == NOFFMAGIC))
324         SwapHeader(&noffH);
325     ASSERT(noffH.noffMagic == NOFFMAGIC);
326
327     int vpn = addr/PageSize;
328
329     int page = memory->Find();
330 printf("pPage find: %d\n",page);
331     if(page>=0)     // some physical page empty
332     {
333         machine->pageTable[vpn].virtualPage = vpn;
334         machine->pageTable[vpn].valid = TRUE;
335         machine->pageTable[vpn].physicalPage = page;
336         machine->pageTable[vpn].last_use_time = stats->totalTicks;
337         currentThread->space->pPage[page] = 1;
338 printf("physicalPage %d is allocated by thread %s(tid: %d) for vPage %d!\n",page,currentThread->getName(),
339 currentThread->get_thread_id(),vpn);
340     }
```

这部分代码实现了当找到空闲物理页时，直接分配，并完成相关变量的更改工作。

```
    else   // lru
    {
printf("###LRU\n");
        int target = 0;
        int tmp = 2147483647;
        for(int i=0;i<machine->pageTableSize;++i)
        {
//printf("pageTable[%d]: vpn %d ppn %d valid: %d\n",i,machine->pageTable[i].virtualPage,machine->pageTable[i].physicalPage,
//machine->pageTable[i].valid);
            if((machine->pageTable[i].last_use_time<tmp) && (machine->pageTable[i].valid))
            {
                target = i;
                tmp = machine->pageTable[i].last_use_time;
            }
        }
        //write back
        int write_back_vpn = machine->pageTable[target].virtualPage;
        int write_back_offset = currentThread->space->code_address;
        executable->WriteAt(&(machine->mainMemory[page*PageSize]),PageSize, write_back_offset + write_back_vpn*PageSize);
        page = machine->pageTable[target].physicalPage;
        machine->pageTable[vpn].virtualPage = vpn;    //in
        machine->pageTable[vpn].valid = TRUE;
        machine->pageTable[vpn].physicalPage = page;
        machine->pageTable[vpn].last_use_time = stats->totalTicks;
        machine->pageTable[target].valid = FALSE;   //out
printf("vPage %d replace vPage %d in pPage %d!\n\n",vpn,machine->pageTable[target].virtualPage,page);
    }
    executable->ReadAt(&(machine->mainMemory[page*PageSize]),PageSize, currentThread->space->code_address + vpn*PageSize);
    delete executable;
}
```

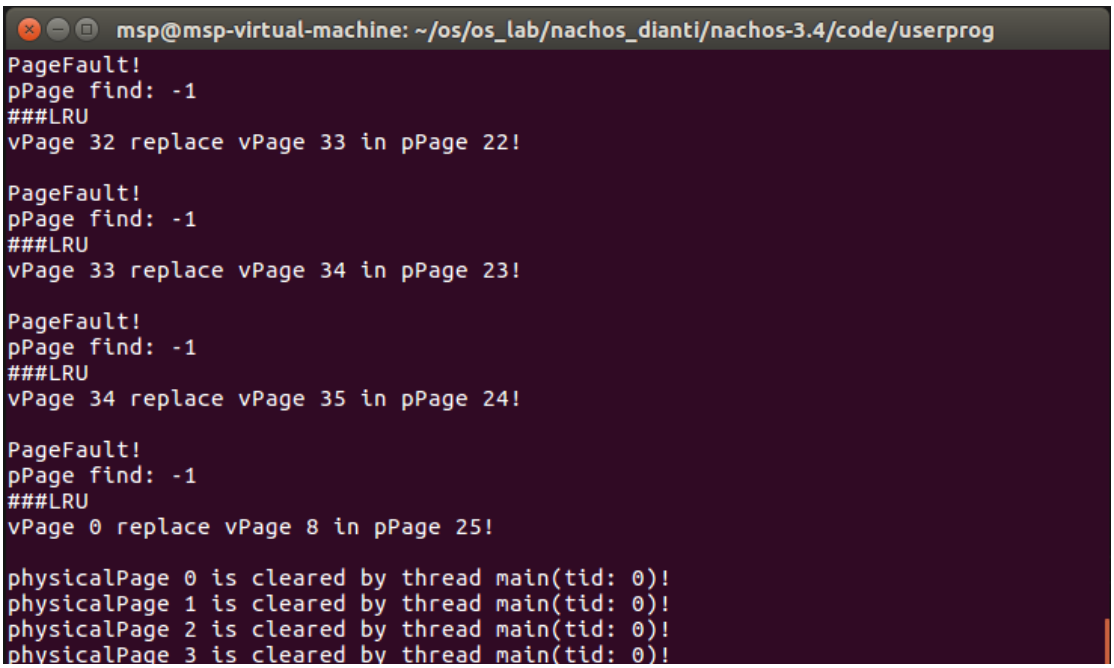
这部分代码实现了 LRU 替换算法。牺牲页进行写回，具体使用 OpenFile 类的 WriteAt 方法写回。并在最后进行了目标页的加载，具体使用 OpenFile 类的 ReadAt 方法加载。

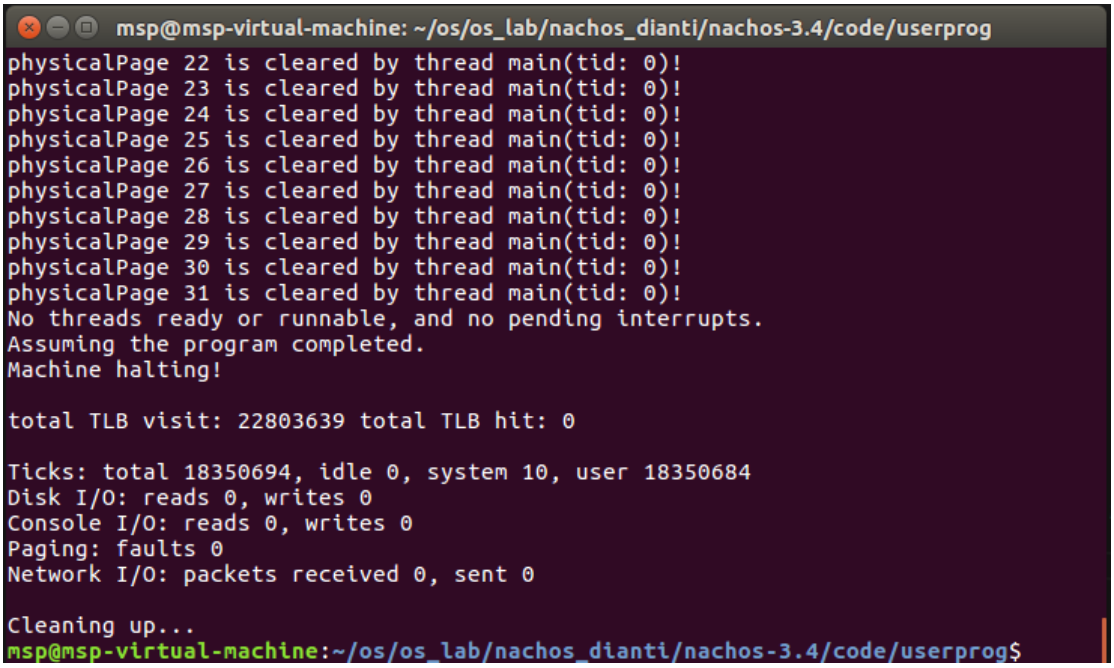
测试：使用 halt 测试：

```
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog$ ./nach
os -x ../test/halt
###RestoreState!
read vPage 0
PageFault!
pPage find: 0
physicalPage 0 is allocated by thread main(tid: 0) for vPage 0!
read vPage 0
read vPage 0
read vPage 1
PageFault!
pPage find: 1
physicalPage 1 is allocated by thread main(tid: 0) for vPage 1!
read vPage 1
read vPage 1
write vPage 10
PageFault!
pPage find: 2
physicalPage 2 is allocated by thread main(tid: 0) for vPage 10!
read vPage 1
write vPage 10
read vPage 1
write vPage 10
read vPage 1
```

可以看到线程试图读内存时，发生缺页异常，然后寻找物理页，并分配内存，每次只分配目标页那一页。每次读写还未分配的内存时，都会发生缺页异常并分配该页，说明 lazy-loading 实现成功。

使用 sort 测试：





由于 sort 程序访问内存太多，结果截图截不到开始部分，从最后的部分可以看出：发生缺页异常时（find 的返回结果为-1）使用了 LRU 替换算法进行替换，虚拟页号和物理页号都可以看到。这说明替换算法实现成功。并且最后进行了内存的回收。

对多线程支持的测试：

```
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/userprog$ ./na
chos -x ../test/halt
thread main(tid:0) yield to thread halt(tid: 1)
thread halt(tid: 1) is running!
PageFault!
pPage find: 0
physicalPage 0 is allocated by thread halt(tid: 1) for vPage 0!
PageFault!
pPage find: 1
physicalPage 1 is allocated by thread halt(tid: 1) for vPage 1!
PageFault!
pPage find: 2
physicalPage 2 is allocated by thread halt(tid: 1) for vPage 10!
PageFault!
pPage find: 3
physicalPage 3 is allocated by thread halt(tid: 1) for vPage 2!
physicalPage 0 is cleared by thread halt(tid: 1)!
physicalPage 1 is cleared by thread halt(tid: 1)!
physicalPage 2 is cleared by thread halt(tid: 1)!
physicalPage 3 is cleared by thread halt(tid: 1)!
PageFault!
pPage find: 0
physicalPage 0 is allocated by thread main(tid: 0) for vPage 0!
PageFault!
pPage find: 1
physicalPage 1 is allocated by thread main(tid: 0) for vPage 1!
PageFault!
pPage find: 2
physicalPage 2 is allocated by thread main(tid: 0) for vPage 10!
PageFault!
pPage find: 3
physicalPage 3 is allocated by thread main(tid: 0) for vPage 2!
physicalPage 0 is cleared by thread main(tid: 0)!
physicalPage 1 is cleared by thread main(tid: 0)!
physicalPage 2 is cleared by thread main(tid: 0)!
physicalPage 3 is cleared by thread main(tid: 0)!
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

与之前多线程的测试相同，不过增加了缺页异常处理和 lazy-loading。可以看到，优先级更高的线程 halt 抢占了 CPU，并 lazy-loading，执行完并回收内存后，线程 main 继续执行，且可以使用线程 halt 已经释放的物理内存。说明改进之后的系统仍旧支持多线程。

第四部分

Challenge1 增加 SUSPENDED 状态

SUSPENDED 状态和 BLOCKED 状态的区别是：

阻塞的线程仍在内存中，而挂起的线程不在内存中，需要写入磁盘。所以对线程的挂起操作的主要工作就是把该线程在内存中的数据写入磁盘。

首先，定义两个全局的列表：List *suspended_ready，和 List *suspended_blocked。分别记录就绪挂起和阻塞挂起的线程队列，然后再线程状态中增加 SUSPENDE_READY 和 SUSPENDED_BLOCKED 这两个状态。

当前线程主动挂起：

Suspend 函数：

```
374  void Suspend()
375  {
376      Thread *nextThread;
377
378      ASSERT(interrupt->getLevel() == IntOff);
379
380      DEBUG('t', "Suspending thread \"%s\"\n", currentThread->getName());
381
382      currentThread->setStatus(SUSPENDED);
383      suspended->SortedInsert2(currentThread,currentThread->get_priority());
384
385      NoffHeader noffH;
386      OpenFile *executable = fileSystem->Open(currentThread->filename);
387
388      executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
389      if ((noffH.noffMagic != NOFFMAGIC) &&
390          (WordToHost(noffH.noffMagic) == NOFFMAGIC))
391          SwapHeader(&noffH);
392      ASSERT(noffH.noffMagic == NOFFMAGIC);
393
394      for(int i=0;i<currentThread->space->numPages;++i)
395      {
396          if(currentThread->space->pageTable[i].valid)
397          {
398              int write_back_ppn = currentThread->space->pageTable[i].physicalPage;
399              int write_back_vpn = currentThread->space->pageTable[i].virtualPage;
400              int write_back_offset = currentThread->space->code_address;
401              executable->WriteAt(&(machine->mainMemory[write_back_ppn*PageSize]),PageSize,
402      write_back_offset + write_back_vpn*PageSize);
403
404          }
405      }
406      while ((nextThread = scheduler->FindNextToRun()) == NULL)
407      interrupt->Idle();
408
409      scheduler->ReadyToRun2(nextThread); // returns when we've been signalled
410  }
```

主要的操作就是一个 for 循环，遍历当前线程的页表，把当前线程占用的物理页的内容全部写回到磁盘中相应的位置。完成后寻找下一个可执行的线程加入 ReadyList 队列。

SUSPENDED_READY 到 READY 的转换：

修改 Scheduler 类中的 FindNextToRun 函数，寻找下一个可以执行的线程时，不仅仅在 ReadyList 中寻找，也在 suspended_ready 队列中寻找，并选两个队列中优先级最高的加入 ReadyList 队列，如果挂起线程优先级高，则完成状态转换。

BLOCKED 到 SUSPENDED_BLOCKED 的转换：

只需要把 Interrupt 类的 pending 队列中，即阻塞线程队列中的所有线程的内存数据写回磁盘并更改线程状态，并把这些线程加入到 suspended_blocked 队列中。具体函数同 Suspend 函数十分相似，只是操作对象不同，一个是当前线程，一个是阻塞的线程。

SUSPENDED_BLOCKED 到 SUSPENDED_READY 的转换：

当阻塞挂起的线程被其他线程的操作唤醒时，如条件变量。发现该线程的状态是阻塞挂起，则在 suspended_blocked 队列中找到该线程，并把该 tcb 移动到 suspended_ready 队列中，并更改线程状态。

内容三：遇到的困难以及解决方法

困难 1

这次的实习流程很长，且前后的任务都有联系，在修改代码时我没有写好注释和提前写实习报告，导致做到后面时已经对改过的代码不够熟悉。以后写代码要及时写好详细的注释，并提前写报告。

内容四：收获及感想

通过这次实习，我对操作系统内存管理有了更深刻的理解，尤其是页表部分，当初学 ics 的时候就理解的不够透彻，现在自己动手写了代码，才清楚的理解了其中的原理和运行机制。

此外，这次实习因为流程较长，比较难，所以我和队友进行了好几次讨论，交换了意见，这更加深了我对这套系统的理解。

内容五：对课程的意见和建议

目前对课程非常满意，没有意见。

内容六：参考文献

[1] Andrew S. Tanenbaum 著．陈向群 马洪兵 译 .现代操作系统［M］. 北京：机械工业出版社，2011：47-95

[2] 和队友刘雨曦同学的讨论