NachOS Lab4 实习报告

史杨勋惟

1200012741 信息科学技术学院

2016,04,30

目录

1 总体概述

这次的 Lab 主要内容是虚拟内存上各种机制的实现，也是有史以来内容最多的一个 Lab。

2 任务完成列表

Exercise 1	Exercise 2	Exercise 3	Exercise 4	Exercise 5	Exercise 6
Y	Y	Y	Y	Y	Y

Exercise 7	Challenge 1	Challenge 2
Y	N	N

3 完成情况

Exercise 1

progtest.cc 中的 StartProcess 函数是启动用户程序的入口。从代码来看，StartProcess 主要做了以下操作。

- 检查这是不是可执行文件;
- 为其分配页表, 初始化页表项;
- 装入数据段、程序段;
- 调用 machine->Run() 方法模拟用户程序在处理器上运行。

machine.cc 是对运行用户程序的硬件进行的模拟。这里我们发现, TLB 在默认下是不开启的, 需要通过宏定义开启。当 TLB 不开其的时候, Nachos 使用 PageTable，如果 TLB 开启，Nachos 使用 TLB。Machine 类负责创建 TLB。TLB 的入口是 RaiseException, 当系统收到一种 Exception(例如 PageFault) 时, 此方法会被调用。从实现上来看，这也是我们要执行页面置换算法的入口。

translate.cc 定义了页表项和翻译规则。这里的页表项和我们在操作系统课上学到的一致。其翻译规则（虚拟地址到物理地址的转换规则）和我们操作系统课上学到的也类似。注意到，PageTable 和 TLB 中的页表项结构是一样的。

Exercise 2 & Exercise 3

Exercise 2 的内容是捕获异常，我们只需在 Exception 类中中国添加一个新的 Exception 枚举体即可，这里我命名为 TLBMissException。在 ExceptionHandler 中，加入一些条件判断，我们就可以确定异常类型是不是 TLBMissException 了。

随后就要对 TLBMissException 设计相应的异常处理函数，也就是设计页面置换算法。这里我选择了两种最简单也是最经典的页面置换算法，FIFO 和 LRU。

在具体实现上，我们需要设计两个函数 TLBSwapFIFO 和 TLBSwapLRU, 这两个函数首先计算出虚拟页号，通过索引得到页表项。然后在 TLB 中找有没有 invalid 的页表项，有的话就直接插入此处，没有的话，就需要选择一个页面置换。

FIFO

对于 FIFO 算法，选择队头，所以用一个常量维护当前索引即可, 算法如下：

```
1     int slot=TLBhead;
2     TLBhead=(TLBhead+1)%TLBSize;
3     int slot = 0;
4     int min = tlb[0].comingTime;
5     TranslationEntry *entry = NULL;
6     for(int i = 0; i< TLBSize;i++){
7         if(tlb[i].valid == FALSE){        //Found an empty TLB entry
8             slot = i;
9           DEBUG('a',"Slot %d is empty.\n",slot);
10            break;
11        }
12    }
13    entry = &tlb[slot];
```

LRU

对于 LRU 算法，需要为页表项维护一个上次访问时间的变量，一个比较直观的办法是把这个时间设成当时的 totalTicks。实现过程只需要扫一遍 TLB，选取最小值作为 slot 即可。算法如下：

```
1     for(int i = 0; i< TLBSize;i++){
```

```
        if(tlb[i].valid == FALSE){        //Found an empty TLB entry
            slot = i;
            DEBUG('a',"Slot %d is empty.\n",slot);
            break;
        }
        if(tlb[i].lastAccessTime < min){
            min = tlb[i].lastAccessTime;
            slot = i;
        }
    }
    entry = &tlb[slot];
```

以下为 FIFO 测试结果:

```
TLB Miss!
put the miss entry in slot 0
TLB Miss!
put the miss entry in slot 1
TLB Miss!
put the miss entry in slot 2
TLB Miss!
put the miss entry in slot 3
TLB Miss!
put the miss entry in slot 0
```

以下为 LRU 测试结果:

```
TLB Miss!
slot 0, last access time: 150
slot 1, last access time: 113
slot 2, last access time: 142
slot 3, last access time: 126
put the miss entry in slot 1
TLB Miss!
```

结果对比

我把排序算法的数的个数射程了 4，对比发现 LRU 的 TLBMiss 比 FIFO 少，对于排序算法来说这个结果是符合常理的。

Exercise 4

我设计的数据结构是 PageManager 类，我的实现是基于位图的，而 Nachos 中的 bitmap.cc 提供了相关的接口。

以下是 PageManager 的接口：

```
class PageManager
{
public:
        PageManager();
        ~PageManager();
        int findPage();
        int numCleanPage();
        void markPage(int pagenum);
        void cleanPage(int pagenum);
private:
        BitMap *manager;
};
```

在实现过程中，首先需要在 Initialize 中初始化一个全局的 PageManager 对象，pageManager。在初始化过程中，我设定的位置大小为 4096，即每个地址都是一个位。但是在分配的过程中还是基于页分配的，也就是每次以 128 位为一个单元分配。当然，这里本质上只需要 4096/128 = 32 个位就行了，但是 4096 本质上并不是一个大数字，所以我还是选择了每个地址映射到一个位。

为了简化 Makefile 的修改，我将 PageManager 类放到了 machine.h 和 machine.cc 中，而不是单独一个文件。

Exercise 5

Nachos 不支持多线程的本质在于虚拟内存的初始化过程中清零了整个物理内存。所以我修改了这个函数，只清零这个进程被分配的页号所对应的物理内存。

修改之后还是不支持多线程的运行的，我分析了很就，发现了 Nachos 不支持多线程的另一个原因。Nachos 不支持多线程的另一个原因在于写入代码段和数据段的时候并不是按照页写的，这样就可能覆盖其他线程的代码段和数据段了，所以我们也需要修改这里。

测试过程中，我们需要模拟多线程运行。一种方法是如在命令行中添加多线程的执行方法，但是我觉得这种方法执行起来比较麻烦。我选择的在系统中模拟多线程，也就是在 StartProcess 函数中添加代码，模拟已有运行的一个用户程序。我为 sort 测试例程建立一个新的地址空间，并且新建一个 thread 使得它的用户空间指向这个空间，然后，让新建立的线程 Fork，从而可以加入调度队列。在这个新进程的伊始，AddrSpace 类的 RestoreState()

方法切换页表, 然后调用 machine->Run() 执行。两个线程执行的都是 sort 程序。还有注意的真是线程刚刚创建的时候不会设置用户寄存器，为了支持多线程，这个部分的代码需要加上。

测试结果如下:

```
New Thread enter translating: vpn:2, ppn:16
Current thread yields
change Pagetable
main enter translating: vpn:2, ppn:2
```

可见正常切换。

Exercise 6 & Exercise 7

这两个练习都是针对缺页处理的，可以一起完成。

lazy-loading 有点类似与 copy-on-write，在初始化页表的过程中，不确定每一个页表条目对应的物理页号, 也不在此时将可执行文件的代码、数据等载入, 而是只等到出现 PageFault 时, 由对应的处理函数载入对应页面。

在 ExceptionHandler 中捕获 PageFault。捕获之后, 如果需要置换页面, 就利用页面置换算法找到应该置换的页面。这里需要自行对 lazy-loading 进行进一步设计，我设计的策略是每个用户程序最多只能占用一定数目的物理页面, 同时在替换时只从自己占用的页面中选取, 而不干扰别的用户程序。知道要置换掉哪个页面之后, 首先看看页面是否为 dirty , 是的话要写回“磁盘”，也就是写入文件中。然后从文件中读入页面信息，读入内存。

具体实现中，首先需要对 AddrSpace 类的构造函数进行修改, 取消分配物理页号，装载数据段等操作。同时增加创建文件的操作，通过 TID 标示文件名。在 TLB miss 处理函数中判断对应表项在 pageTable 中是否为 valid , 为 PageManager 新增三个函数载入页面 (loadPage) 、选择置换页面 (getSwapPage) 、置换页面 (swapPage)。并在异常处理程序中加入相应的调用。

具体代码如下:

```
1 void
2 PageManager::loadPage(int address)
3 {
4     char *filename = currentThread->space->getFileName();
5     // get the executable file name of the thread
6     int phynum;
7     int vpn = address/ PageSize;
8     printf("PageFault from virtual page %d!\n", vpn);
```

```
9
10      // find a clean page in the memory for the program
11      int availPage=currentThread->space->getAvailPageNum();
12      ASSERT(availPage>=0);
13      if(availPage==0) {
14          phynum = swapPage();
15          ASSERT(phynum!=-1);
16      }else{
17          phynum = findPage();
18          currentThread->space->setAvailPageNum(availPage-1);
19          ASSERT(phynum!= -1);
20      }
21
22      markPage(phynum);       //set the page
23
24      printf("Load virtual page %d to physical page %d from %s\n",
25       vpn, phynum, filename);
26
27      //open file and load the page
28      OpenFile *executable = fileSystem -> Open(filename);
29      executable ->ReadAt(&(machine ->mainMemory[phynum * PageSize]),
30       PageSize, vpn * PageSize);
31
32      // set the pagetable
33      machine ->pageTable[vpn].valid = TRUE;
34      machine ->pageTable[vpn].physicalPage = phynum;
35      machine ->pageTable[vpn].virtualPage = vpn;
36      machine ->pageTable[vpn].use = FALSE;
37      machine ->pageTable[vpn].dirty = FALSE;
38      machine ->pageTable[vpn].readOnly = FALSE;
39      machine->pageTable[vpn].comingTime=stats->totalTicks;
40
41      DumpState();
42      delete executable;
43  }
44
45
46  int
47  PageManager::swapPage()
48  {
49      int  entryid, phynum, vpn, pagenum;
50      pagenum = machine->pageTableSize;
51      //entryid= getSwapPageFIFO();
52      entryid = getSwapPageLRU();
53      ASSERT(entryid != -1 && machine->pageTable[entryid].valid==TRUE);
54
55      vpn=machine->pageTable[entryid].virtualPage;
```

```
56        phynum=machine->pageTable[entryid].physicalPage;
57
58       printf("Page Table is full! Page %d will be swapped out!\n",phynum);
59        machine ->pageTable[entryid].valid = FALSE;
60        //open file and write memory data, if the page is dirty
61        if(machine ->pageTable[entryid].dirty == TRUE)
62        {
63           char *filename=currentThread->space->getFileName();
64           printf("Page %d is dirty, write back to %s\n",phynum, filename);
65
66            OpenFile *executable = fileSystem -> Open(filename);
67            executable ->WriteAt(&(machine ->mainMemory[phynum * PageSize]),
68            PageSize, vpn * PageSize);
69            delete executable;
70        }
71        //clean the page
72        cleanPage(phynum);
73        return phynum;
74  }
75
76  int
77  PageManager::getSwapPageFIFO()
78  {
79      DEBUG('a',"PageFault, use FIFO swap\n");
80      int slot=0;
81      int min = stats->totalTicks;
82      int pagenum=machine->pageTableSize;
83      for(int i = 0; i< pagenum;i++){
84          if(machine->pageTable[i].valid==TRUE &&
85          machine->pageTable[i].comingTime < min){
86              min = machine->pageTable[i].comingTime;
87              slot = i;
88            }
89      }
90      return slot;
91  }
92
93  int
94  PageManager::getSwapPageLRU()
95  {
96      DEBUG('a',"PageFault, use LRU swap\n");
97      int slot=0;
98      int min = stats->totalTicks;
99      int pagenum=machine->pageTableSize;
100     for(int i = 0; i< pagenum;i++){
101         if(machine->pageTable[i].valid==TRUE &&
102         machine->pageTable[i].lastAccessTime < min){
```

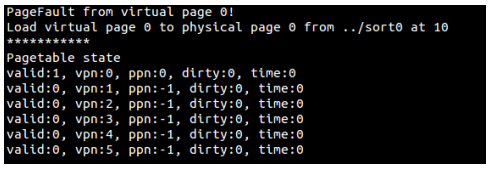
```
            min = machine->pageTable[i].lastAccessTime;
            slot = i;
          }
    }
    return slot;
}
```

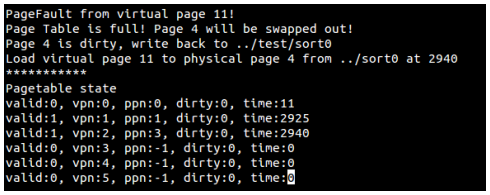
在异常处理程序中添加

```
    }else if(which==PageFaultException){
        (stats->numPageFaults) ++;
        int address=machine->registers[BadVAddrReg];
        pageManager->loadPage(address);
        };
```

这里的测试比较复杂，我把调试信息打到了文件中，我们可以从文件中判断是否 lazy-loading 正确。

下面两张图分别是程序一开始的页表状态和中间段的页表状态，我们发现一上来都是 PageFault，属于正常的 lazy-loading 情况。到了中间，当程序缺页的时候，会从替换页面写到文件。测试情况符合逻辑。





4 遇到的困难以及解决办法

Makefile 的修改

虽然助教在第一节课上讲过怎么样添加一个自己的 C 文件，这次在实现 PageManager 上，我尝试过把 PageManager 单独写成一个 cc 文件，然后修改 Makefile，不过其中的依赖关系实在比较复杂，我一直没有成功编

译。最后我采用了投机的办法，把 PagaManager 类和它的实现都加入了 machine.h 和 machine.cc，从而不用修改 Makefile 就能够通过编译。

调试

这次的 Lab 内容比较多，所以各种错误也比较多，尤其是牵涉到用户程序和内存管理，需要一步一步慢慢地分析。在调试方面我也花了很多时间。

5 收获与感想

这次的 Lab 内容比较多，而且都是实实在在的内容，不花时间是不行的。作为大四学生，我还需要重温两年前学习的操作系统的很多细节内容，也是挺花时间的。也导致了我没有做 Challenge 了，也是一个遗憾的地方。

当然，这次 Lab 也让我们进一步理解了操作系统虚拟内存管理上的很多东西，其实 Nachos 在虚拟内存上的控制已经算比较简单的了，真实的系统中十分复杂。当然，其实这次 Lab 也对我的毕业设计有一点帮助的地方。在我毕业设计中，我需要为 fork() 系统调用设计一个文件级别的 copy-on-write，这个和本次实验的 lazy-loading 有一点类似之处，当然我还没有开始设计我的工具，但多少也可以给我一点启发。

6 意见与建议

无。

7 参考资料

- 《现代操作系统》