山东大学 软件 学院

**操作系统课程设计** 实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202000300358 | 姓名：梁思睿 | | 班级：软工20.6 |
| 学号：202022300316 | 姓名：王丁 | | 班级：软工20.6 |
| 实验编号：Lab7 | | | |
| 实验题目：虚拟内存 | | | |
| 实验学时：6 | | 实验日期：2022.xx.xx | |
| 实验目的：  1. 在未实现虚拟内存管理之前，Nachos在运行一个用户进程的时候，需要将程序运行所需全部内存空间一次性分配。虚拟内存实现将突破物理内存限制。本实验核心任务为根据理论学习中涉及的对换（Swapping）技术，在Lab6的基础上，设计并实现用户空间的虚拟内存管理。  2. 用户进程的帧数采用固定分配(建议5帧)，局部置换。  3. 实现“纯按需调页”(pure demand paging)。  4. 页置换算法可以采用LRU、增强型二次机会、二次机会、FIFO等算法之一，或自己认为合适的其他算法(不包括随机置换)。  5. 对class Statistics进行调用及修改，以便在程序结束时打出页故障次数及将牺牲页写入交换空间的次数。  6. 使用lab7目录中的示例程序n7(若lab7额外实现了多种算法，可用自己的lab7)，测试用户程序用同样ARRAYSIZE参数值的sort，但不同的页置换算法(详见code/lab7/n7readme.txt)多次运行n7。不同页置换算法运行结束时显示的user ticks数是否一样？解释这是为什么？  7. 最优页置换算法(OPT)有最低的页故障率，但需要未来的页面引用信息，因此不能用于实际环境，主要用于评估其他页置换算法的性能。在前述1-5实现的基础上，给出在Nachos中获得最优页置换算法页故障次数的具体实现方法(不要求实现可运行的代码。在实验报告中用文字描述即可，必要时可在文字中结合进关键代码片段、数据结构、对象等说明)。 | | | |
| 硬件环境：  设备名称 LAPTOP-UUHJO71O  处理器　　AMD Ryzen 5 4600U with Radeon Graphics 2.10 GHz  机带 RAM 16.0 GB (15.4 GB 可用)  设备 ID 63412E57-FC9E-4F92-A869-B35FDEF3303C  产品 ID 00342-35837-59360-AAOEM  系统类型 64 位操作系统, 基于 x64 的处理器  硬盘 512GB SSD | | | |
| 软件环境：  宿主机：Windows 10 21H2 64位  虚拟机软件：VMware Workstation Pro 16.1.2 build-17966106  Linux：Ubuntu 14.04.6 LTS Desktop i386 (Trusty Tahr)  gcc/g++：(Ubuntu 4.8.4-2ubuntu1~14.04.4) 4.8.4  MIPS交叉编译器：gcc-2.8.1-mips.tar.gz  Nachos：Nachos-3.4-UALR-2022 | | | |
| 实验步骤与内容：  **概述**  在上一次实验的时候，我们已经实现了多道用户程序，能够运行多道程序而不会产生内存的相互覆盖。但是目前的nachos依然有着一些缺陷。如果要运行的程序太大，那么还是无法运行，因为目前的nachos是先把程序的所有页都调入内存，再运行。如果程序太大，就会失败；我们可以用lab6的代码运行一下sort.noff试试：  这次的实验就是要实现虚拟内存，以达到更加灵活的程序运行效果。如果访问到不在内存中的页操作系统会发出一个页错误，然后nachos要处理这个页错误，就要用到页面置换算法来选择该进程在物理页中的某一页进行替换。这是上学期操作系统学过的知识。   1. 固定分配，局部置换。   固定分配指的是，对于每个程序，分配给的page数目是固定的，运行时不在增加或者减少。局部置换指的是，缺页的时候只能换出本程序的暂时不用的页，而不能“染指”其他程序的页面或者空闲的页。   1. 纯按需调页   只有程序执行的时候才需要调入页，在nachos中，也就是一开始并不分配几个初始的页，而是等到访问的时候发出缺页中断在进行调入。也成为懒交换。   1. 页置换算法   这里可以采用LRU，增强二次机会（CLOCK）、二次机会、FIFO等等算法。我们这次试验实现了CLOCK和FIFO算法来实现页面置换。  **Nachos 用户程序相关代码回顾**   1. **页表项**   要想实现虚拟内存，显然需要先看看TranslateEntry的结构。在上一个实验中已经可以注意到，里面有很多信息位，利用这些信息位就能做一些虚拟内存相关的工作。话不多说先看看代码：  class TranslationEntry {    public:      int virtualPage;      int physicalPage;      bool valid;  // If this bit is set, the translation is ignored.      // (In other words, the entry hasn't been initialized.)      bool readOnly;      bool use;      bool dirty;         // set every time the page is modified.  };  这里的 virtualPage, physicalPage 意义很简单，通过valid可以看出这一页是否有效，readonly指示这一页是否是只读的，如果被用过，use就置1，如果被写过，dirty就是1。  显然，利用use，dirty位可以构造虚拟内存。   1. **中断机制与系统调用**   经过实验 6 后，我们的ExceptionHandler长这个样子：  void ExceptionHandler(ExceptionType which) {      int type = machine->ReadRegister(2);      if ((which == SyscallException) && (type == SC\_Halt)) {          DEBUG('a', "Shutdown, initiated by user program.\n");          interrupt->Halt();      } else if ((which == SyscallException) && (type == SC\_Exec)) {          DEBUG('a', "Exec, initiated by user program.\n");          char filename[100];          int addr = machine->ReadRegister(4);          int i = 0;          machine->ReadMem(addr + i, 1, (int\*)&filename[i]);          while (filename[i++] != '\0')              machine->ReadMem(addr + i, 1, (int\*)&filename[i]);          interrupt->Exec(filename);          AdvancePC();      } else if ((which == SyscallException) && (type == SC\_PrintInt)) {          DEBUG('a', "Exit, initiated by user program.\n");          interrupt->PrintInt(machine->ReadRegister(4));          printf("reg4: %d\n", machine->ReadRegister(4));          AdvancePC();      } else {          printf("Unexpected user mode exception %d %d\n", which, type);          ASSERT(FALSE);      }  }  那么我们需要添加缺页中断的逻辑，就在这里添加。  关于bitmap做内存管理工具的思路，上一个实验已经比较相近了。我感觉这个思路很巧妙甚至有点类似于trick，但是又很好用，这次试验还会继续用，用法也和上次相同。  **解题思路**  要实现虚拟内存，首先得有个能存储当前占用内存空间的虚拟页的结构，并且让程序能够先部分装入。在这里我们要实现的是固定分配局部置换。那么怎么才能知道必须页的数量呢？我们只需要用max(initData + code + 1, pnperp)，即可。其中initData + code + 1是系统不发生颠簸的最小页数量。这里+1容易被遗忘，但非常有用；而 pnperp是**p**age **n**umber **per** **p**rocess的意思，这里为了简便我把它写成缩写。    要想实现增强二次机会算法，要把分配的物理页想成一个环，需要有一个指针来指向每一个页和移动。发生缺页的时候其实是发生一个异常，RaiseException 将该异常分派到对应的异常处理函数 ExceptionHandler 处理。在上次的实验中，我们通过修改exceptionHandler和Interrupt来添加了两个系统调用（PrintInt, Exec）。这次我们肯定也要修改这两个类，只不过修改的方式不一样。因为缺页中断本质上不是系统调用。当然，这一点只需要留意一下，因为只是在ExceptionHandler的不同的else分支写上处理逻辑。  此外，还需要实现页面交换函数 Swap，以及将当前要被置换的页的写回函数 WriteBack 等函数，以实现虚拟内存和增强二次机会算法（Enhanced Clock）。这两个算法在上学期都已经讲过，回忆一下写出来并不是很难。  **关键代码的解释**   1. **固定分配，局部置换**   老师要求的是采用固定分配（建议每个程序有 5 个帧的空间），那么我们就#define一个宏，把这个固定数值表示出来（也就是pnperp）。  #ifndef pnperp  #define pnperp 5  #endif  #ifndef SWAP\_STRATEGY  #define SWAP\_STRATEGY int  #define STR\_\_FIFO\_\_ 1  #define STR\_\_CLOCK\_\_ 2  #endif  这里稍微解释一下其他的数值是什么意思。因为这里要用各种页面置换算法，所以我把“是否实现了任何一种实现算法” 作为SWAP\_STRATEGY来进行暗示。FIFO定义为STR\_\_FIFO\_\_，CLOCK定义为STR\_\_CLOCK\_\_，具体后面讲到再说。  根据题意“实现‘纯按需调页’(pure demand paging)”，这提示我们在需要该页面时再把页面调入，而不是刚开始就分配好内存空间。  那显然要改构造函数：  AddrSpace::AddrSpace(OpenFile \*executable, char \*filename) {  … …  for (i = 0; i < numPages; i++) {          pageTable[i].virtualPage = i;          pageTable[i].physicalPage = -1;          pageTable[i].valid = false;          pageTable[i].use = false;          pageTable[i].dirty = false;          pageTable[i].readOnly = false;      }  也就是一开始就把所有valid位设置为false.  我们还需要设置一个临时的变量，目的是使用bitmap的find函数记录当前是否有空闲空间，如果有，是哪一个空闲页。如果有空闲空间，则无需置换旧页，直接在该空闲空间中写入新页即可；如果没有空闲空间。则需要页置换算法将旧页换出，将新页写入。  有空闲空间时处理的代码如下：  int AddrSpace::FIFO(int badVAddr) {      printf("--------------- FIFO Algorithm ---------------\n");      int temp = 0;      if ((temp = userMap->Find()) != -1) {          int newPage = badVAddr / PageSize;          printf("第 %d页写入,不需要写出旧页.\n", newPage);          virtualMem[p\_vm] = newPage;          p\_vm = (p\_vm + 1) % pnperp;          pageTable[newPage].physicalPage = temp;          OpenFile \*vm = fileSystem->Open("VMFile");          vm->ReadAt(              &(machine->mainMemory[pageTable[newPage].physicalPage  \* PageSize]),              PageSize, newPage \* PageSize);          delete vm;          pageTable[newPage].valid = true;          pageTable[newPage].use = true;          pageTable[newPage].dirty = false;          pageTable[newPage].readOnly = false;          Print();          return 0;      } else {  … …  }  }  根据题意“页置换算法可以采用LRU、增强型二次机会、二次机会、FIFO等算法之一，或自己认为合适的其他算法。我实现了两种页置换算法：FIFO置换算法和增强型二次机会置换算法。   1. **FIFO**   这个算法相对来说比较简单，实现也容易。我是这样实现的：每次都将第一个页取出，并且把新的页放在队首的位置；同时指针指向下一个位置，因此新被调入的页在之后的swap中是最不优先考虑的。如果旧页的dirty位是1，说明旧的页已经被修改了，需要把旧的页进行写回，并且修改其各种信息位，然后把新的页面载入内存。具体如下：          int oldPage = virtualMem[p\_vm];          int newPage = badVAddr / PageSize;          virtualMem[p\_vm] = newPage;          p\_vm = (p\_vm + 1) % pnperp;          a = Swap(oldPage, newPage);          OpenFile \*executable = fileSystem->Open("VMFile");          if (executable == NULL) {              printf("Unable to open filssse %s\n", filename);              return 3;          }          executable->ReadAt(              &(machine->mainMemory[pageTable[newPage].physicalPage  \* PageSize]),              PageSize, newPage \* PageSize);          delete executable;          Print();          return a + 1;  那么跑一下看看。这里我随机截了相邻的两张图，如图所示    这里要把page 4 换出，换入 page 6，把 page 4 的valid位设置为 0，把page 4 的位置让给page 6，并且把6的valid设置为1.     1. **CLOCK（增强型二次机会算法）**   CLOCK算法的算法步骤可以描述如下：  内存中的页面通过指针链接成一个循环队列，有访问位和修改位两个位， 除了考虑是否被访问过，还考虑页面是否被修改过。为 1 表示访问过或修改过， 用(used, dirt)的形式表示各页面状态，改进的时钟置换算法最多进行四轮扫描：   1. 第一轮：从指针当前位置开始扫描，尝试找到第一个（0,0）的页帧用 于替换，本轮扫描 不修改任何标志位。（既没被访问，又没被修改的页帧。） 2. 第二轮：如果第一轮扫描失败，第二轮扫描尝试找到第一个（0,1）的 页帧用于替换，并 且本轮扫描中所有被扫过的页帧访问位重置为 0。（没有被访问，但是被 修改过的页帧。） 3. 第三轮：若第二轮扫描失败，第三轮扫描尝试找到第一个（0,0）的页 帧用于替换，本轮 扫描不修改任何标志位。（被访问过，但是没有修改过的页面。） 4. 第四轮：若第三轮扫描失败，第四轮扫描尝试找到第一个（0,1）的页 帧用于替换，一定能找到。（被访问过，且被修改过的页面。）   需要注意的是， 第一轮同 CLOCK，每轮扫描修改的时候如果遇到了 OK 的页面，则置换后本 次置换扫描就停止了， 不是每轮检测都要转完一整圈。 ②淘汰页面并换入新页面以后访问位置为 1，没被修改修改位仍为 0，然 后后移指针一次。 我设置当页读入内存的时候设置 use 位为 1。  **关于问题 6**  **OPT 算法的实现思路** | | | |
| 结论分析与体会：  可写的包括但不限于：  解决xxx问题的一般做法；  获得的经验；  自己的体会与提高；  对操作系统相关问题的进一步理解；  等等… | | | |