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：2022.12.05 | |
| 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、二次机会、FIFO等等算法。我们这次试验实现了增强二次机会和FIFO算法来进行页面置换。  **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。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。在这里要实现的是固定分配局部置换。程序中定义了一些特殊的变量，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，所以我们把“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 newVPN = badVAddr / PageSize;          printf("%d页写入,不需要写出旧页.\n", newVPN);          virtualMem[p\_vm] = newVPN;          advancePtr();          pageTable[newVPN].physicalPage = temp;          OpenFile \*vm = fileSystem->Open("VMFile");          vm->ReadAt(&(machine->mainMemory[pageTable[newVPN]  .physicalPage \* PageSize]), PageSize, newVPN \* PageSize);          delete vm;          pageTable[newVPN].valid = true;          pageTable[newVPN].use = true;          pageTable[newVPN].dirty = false;          pageTable[newVPN].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。我们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。我们是这样实现的：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 oldVPN = virtualMem[p\_vm];          int newVPN = badVAddr / PageSize;          virtualMem[p\_vm] = newVPN;          advancePtr();          a = Swap(oldVPN, newVPN);          OpenFile \*executable = fileSystem->Open("VMFile");          if (executable == NULL) {              printf("Unable to open filssse %s\n", filename);              return 3;          }          executable->ReadAt(              &(machine->mainMemory[pageTable[newVPN]  .physicalPage \* PageSize]), PageSize, newVPN \* PageSize);          delete executable;          Print();          return a + 1;  那么跑一下看看。这里我们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，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。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。 我们设置当页读入内存的时候设置 use 位为 1。  CLOCK 的具体实现代码如下:  int AddrSpace::clock(int badVAddr) {      printf("--------------- CLOCK Algorithm ---------------\n");      int temp = 0;      if ((temp = userMap->Find()) != -1) {          directSwapInRoutine(badVAddr, temp);          return 0;      }      int oldVPN;      int count = 0;  // circle count      // search from (0, 0)      for (int i = 0; i < pnperp; ++i) {          if (notUsednotDirty()) {              oldVPN = ptrVPN();              printf("第一轮，找到的要替换的页是：%d \n", oldVPN);              break;          }          advancePtr();          count++;      }      if (count == pnperp) {  // 2th          count = 0;          for (int i = 0; i < pnperp; i++) {              if (notUsedbutDirty()) {                  oldVPN = ptrVPN();                  printf("第二轮，找到的要替换的页是：%d \n", oldVPN);                  break;              }              pageTable[virtualMem[p\_vm]].use = 0;              advancePtr();              count++;          }      }      if (count == pnperp) {          count = 0;          for (int i = 0; i < pnperp; ++i) {              if (notUsednotDirty()) {                  oldVPN = ptrVPN();                  printf("第三轮，找到的要替换的页是：%d \n", oldVPN);                  break;              }              advancePtr();              count++;          }      }      if (count == pnperp) {          for (int i = 0; i < pnperp; ++i) {              if (notUsedbutDirty()) {                  oldVPN = ptrVPN();                  printf("第四轮，找到的要替换的页是：%d \n", oldVPN);                  break;              }              advancePtr();          }      }      int newVPN = badVAddr / PageSize;      ASSERT(newVPN < numPages);      virtualMem[p\_vm] = newVPN;      advancePtr();  // moveback pointer      writeBacked = Swap(oldVPN, newVPN);      OpenFile \*executable = fileSystem->Open("VMFile");      if (executable == NULL) {          printf("Unable to open filssse %s\n", filename);          return 3;      }      executable->ReadAt(          &(machine->mainMemory[pageTable[newVPN].physicalPage \* PageSize]),          PageSize, newVPN \* PageSize);      delete executable;      Print();      return 1 + writeBacked;  }  这里提一下，不管是FIFO还是CLOCK都返回了一个INT数值，这个数值是暗示的“写回”次数，方便统计类进行统计。  关于Swap() 和writeBack() 函数，逻辑比较简单，实现也容易，这里放出源代码，不再进行解释。  Swap():  int AddrSpace::Swap(int oldVPN, int newVPN) {      // if dirty, writeback and return 1.      // if not dirty, refuse to writeback and return 0.      int writeBacked = writeBack(oldVPN);      pageTable[newVPN].physicalPage = pageTable[oldVPN].physicalPage;      printf("Swap out oldVPN: %d, Swap in newVPN: %d (frame %d)\n", oldVPN,             newVPN, pageTable[oldVPN].physicalPage);      pageTable[oldVPN].valid = false;      pageTable[newVPN].physicalPage = pageTable[oldVPN].physicalPage;      pageTable[newVPN].valid = true;      pageTable[newVPN].use = true;      pageTable[newVPN].dirty = false;      return writeBacked;  }  writeBack():  // if dirty bit set to true, write back to disk  int AddrSpace::writeBack(int oldVPN) {      // if dirty, writeback and return 1.      // if not dirty, refuse to writeback and return 0.      if (pageTable[oldVPN].dirty) {          OpenFile \*executable = fileSystem->Open("VMFile");          if (executable == NULL) {              printf("Unable to open files %s\n", filename);              return 0;          }          executable->WriteAt(              &(machine->mainMemory[pageTable[oldVPN].physicalPage \* PageSize]), PageSize, oldVPN \* PageSize);          delete executable;          return 1;      }      return 0;  }  到现在，两个具体算法已经实现出来了。那么剩下的当然是要更改异常处理机制，使得其可以处理缺页中断：  void ExceptionHandler(ExceptionType which) {  ... ...  else if ((which == PageFaultException)) {          bool k = interrupt->PageFault();          DEBUG('a', "PageFault.\n");      } else {          printf("Unexpected user mode exception %d %d\n", which, type);          ASSERT(FALSE);  }  }  缺页中断的具体策略选择：  bool Interrupt::PageFault() {      int badVAddr = machine->ReadRegister(BadVAddrReg);      AddrSpace \*space = currentThread->space;   ★ stats->numPageFaults++;      int t = -1;      if (swap\_strategy == STR\_\_FIFO\_\_) {          t = space->FIFO(badVAddr);      } else if (swap\_strategy == STR\_\_CLOCK\_\_) {          t = space->clock(badVAddr);      } else {          printf(              "Unknown swap swap\_strategy: %d, expect 1 for FIFO or 2 for  CLOCK.\n", swap\_strategy);          ASSERT(false);      }      if (t) {          if (t == 2) {   ★         stats->numWriteBacks++;              return true;          }      } else return false;  }  **关于统计信息**  根据题意“对class Statistics进行调用及修改，以便在程序结束时打印出页故障次数及将牺牲页写入交换空间的次数”，我们在stats.h中增加了一个变量numWriteBacks，用于记录旧页写回的次数。同时，对addrspace.cc进行了更改，如果页置换算法写回了旧页，就返回2，并对interrupt.cc中的缺页异常处理函数进行修改。在上一个代码段的“★”行就是统计信息相关的行。  那么如此一来，就能打印出pagefault次数和实际将page swap出交换空间的次数了。效果如下（这里是运行了sort.noff，并且CLOCK算法）：    然后将运行结果与n7二进制文件进行比较。这里简单提一下：关于增强二次机会算法和FIFO之间的切换，我们是通过main函数里面的85、86行实现的，要想切换页调度策略，需要注释掉其中的一行。   1. FIFO的运行情况   第一张图是 ./nachos -x ../test/sort.noff  第二张图是 ./n7 -pra 1 -x ../test/sort.noff      结果显示我们的FIFO算法和n7的FIFO算法等价。   1. CLOCK的运行情况：   第一张图是 ./nachos -x ../test/sort.noff的运行情况，  第二张图是 ./n7 -pra 3 -x ../test/sort.noff的运行情况。      可以看出两者是等价的。试验成功。  **关于问题6**  问题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?  首先，不同页置换算法运行结束时显示的user ticks数不一样。  下面运行不同的命令来测试：  ./n7 -pra 4 -x ../test/sort.noff (LRU (栈)页置换)    ./n7 -pra 0 -x ../test/sort.noff(OPT最优置换)    ./n7 -pra 1 -x ../test/sort.noff (FIFO)    ./n7 -pra 2 -x ../test/sort.noff(CLOCK)    ./n7 -pra 3 -x ../test/sort.noff(增强二次机会)    ./n7 -pra 5 -x ../test/sort.noff (随机的页面置换)    原因也很简单，nachos的内核代码是直接执行的。在内核中花费的时间，和系统调用的次数紧密相关。不同的页置换算法缺页次数不同，所以抛出异常的数量自然不同，因此花费的时间也不一样。所以user ticks不同。  **OPT 算法的实现思路**  关于OPT算法的定义，上学期已经学过，这里就不再赘述。OPT算法听起来就很不好实现，这个问题也困扰了我们很久，觉得这似乎是不可能完成的任务——直到看到了LAB7的README文件，老师的实现也是根据“已经有过的一次代码执行页面调度流程记录”才能OPT调度，到这里才豁然开朗。  我们之前实现OPT的思路是，通过代码审查，结合pageNum（如果采取固定分配策略的话）推算出哪些页面将会先访问，从而达到OPT。但是这样导致的语义分析、自动化代码审查、统计信息工作量是非常庞大的，可以说更具体的细节一点思路都没有。  不过实际上可以先获得“程序执行一次的page reference列表”，再根据这个去实现OPT! 所以可以按照这样的思路（以下讨论在不会引起歧义的情况下，将RAM的用户空间部分简称为RAM）：对于每次将要使用一个页，做出以下判断：如果已经在RAM内，则什么都不做，只是简单地将命中数量记录+1. 如果发生缺页中断，则看看现在RAM里面有没有将来永远不会访问到的页。如果有，则毫无疑问需要将它调出，换成新的页。然而大多数情况下都没有，这种时候就看看现有的RAM内哪个页面在“最远的将来”被访问，这样的页面一定存在，我们将这个页面从RAM中换出，并且将需要的新页换入。  这明显是一个贪心算法。那么这个贪心到底能不能保证缺页次数最小呢？下面来讨论。  假如有一次调入新页面(记作x)的时候，我们**不**调出一个在“最远的将来”被访问的页(记作a)而是调出任意一个其他页(记作b)，我们将这个新的“最优”调度策略记作Algo，在这一步操作之后，b已经在磁盘中而不是RAM内。那么Algo调度算法下一次引用b从而造成缺页中断的时间一定早于OPT算法中下次引用a的时间。对于任意一次缺页中断，以上论断都是成立的，从而证明了OPT至少一定不会比Algo差。因为我们假设Algo是最优的，所以可以得出结论：OPT也是缺页中断次数最少的页面调度算法。  实际上，这个Algo调度算法并不一定比OPT差。可以设想一种情况：针对某些特定的reference序列，Algo调度算法和OPT可能仅仅有一两次调度的区别，而这些区别并不一定造成更多的缺页中断。不过我们已经证明了OPT是缺页中断最少的调度算法，所以Algo调度算法怎样其实已经不重要。  那么如何在nachos中实现呢？其实也很简单，只需要随便指定一个其他策略的页调度算法（比如FIFO或者增强二次机会），更改其缺页中断部分的逻辑，使得其在调度过程中，每次缺页中断都会在指定文件中写入一个记录即可。这个文件的文件名和要执行的noff程序相关，再次使用OPT调度算法执行noff文件的时候，就根据这个reference文件进行调度。这实际上也和老师给出的二进制文件n7采用的策略一致。  那么LAB7至此就圆满完成。 | | | |
| 结论分析与体会：  通过本次实验，我们实现了 nachos的虚拟内存，对操作系统的用户进程空间管理有了更深的理解。同时，我们复习了多种页面置换算法并实现了增强二次机会算法，在编写代码的过程中也更加熟悉了 nachos的页表结构并可以熟练应用 nachos的页表。  这学期的nachos实验至此圆满完成。通过这一系列循序渐进的实验，我们花费了相当多的时间，不仅复习了C++、Makefile的运用，还通过git进行项目管理与协作，还通过gdb进行了调试，可以说光是C++技术栈和项目管理的内容就学到了很多。另外，我们还对于各种Linux命令更加熟悉了。  一系列实验跟着做下来，我们对操作系统的线程调度、文件管理、系统调用、异常处理和多道用户程序乃至虚拟内存，不仅有了更加深刻的理解，而且在nachos里面添加了实际代码并且运行成功。通过实验的调试，提升了应对问题、解决问题的能力，我们对操作系统实践部分有了深刻的认识，并经过分析与思考有了自己的理解，使我们对操作系统有了很深刻的体会。收获颇丰。 | | | |