|  |  |
| --- | --- |
| 实验编号：Lab7 | |
| 实验题目：虚拟内存 | |
| 实验学时：6 | 实验日期：2022.12.01 |
| 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：  联想小新Air 15ALC 2021  设备名称 LAPTOP-OPORL5Q4  处理器 AMD Ryzen 7 5700U with Radeon Graphics 1.80 GHz  机带 RAM 16.0 GB (13.9 GB 可用)  硬盘 1TB SSD | |
| 软件环境：  宿主机：Windows 11 家庭中文版 21H2 22000.1219  虚拟机软件：VMware® Workstation 16 Pro 16.0.0 build-16894299  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： 一、纯按需调页-FIFO1.1 解决问题的思路 在实验六中，虚拟页与物理页的对应关系在进程运行过程中是不会改变的，因此虽然使多道用户程序可以在内存中共存，但每个进程占用的物理页数量都是固定等于其虚拟页数的，物理内存资源被消耗得很快。因此需要一定的调页算法，使更多用户程序可以同时驻留在内存中。根据实验要求，每个用户进程最多固定占用5个物理页。  实验要求纯按需调页，说明初始化地址空间时，没有需要的内存，那么所有虚拟页均需设为无效，这样nachos在访存时会抛出缺页异常（PageFaultException）。此时为其分配合适的物理页供其使用即可。  缺页异常的处理程序，根据实验六的分析，入口在userprog/exception.cc中，同时在BadVAddrReg寄存器中会有缺页异常的地址，可以经由此处转中断处理程序，也就是machine/interrupt.cc中处理页错误的函数。  置换出的内存页的内容，参照示例代码，可以放入磁盘中的SWAPn（n为用户程序的SpaceId号）中。相较于内存，磁盘空间我们认为是足量的，所以这样做没有问题。  我们选择使用FIFO作为调页算法，此算法需要使用的队列可以使用threads/list.h中定义的队列，对其进行修改（增加实例变量size维护其长度）就可以为调页算法使用了。 1.2 实现步骤与代码说明1.2.1 为AddrSpace类绑定交换文件 参照示例程序，编号为n的地址空间绑定的虚拟内存交换文件名为SWAPn，在进程执行的全过程均有可能用到该文件，每次重新生成该文件名也会造成开销，且该命名规则保证了进程执行过程中不会有其他进程占用此文件名，所以将该名称作为一个AddrSpace实例变量以便随时在磁盘中找到该文件：   1. private: 2. ... 3. char swapFileName[20];   即一个长度最大为20的字符串，显然对于这个模拟的内存大小来说完全足够了。  我们规定交换文件的内容与虚拟内存内容的对应关系为：将虚拟内存某位置的内容完全对应到交换文件的该位置，那么交换文件的大小应被初始化为虚拟内存大小，这样在进程运行过程中大小就不会改变。我们就可以在AddrSpace对象初始化时同时初始化交换文件：   1. ... 2. spaceId=spaceIdMap->Find(); 3. sprintf(swapFileName,"SWAP%d",spaceId); 4. ... 5. size = noffH.code.size + noffH.initData.size + noffH.uninitData.size 6. + UserStackSize; *// we need to increase the size* 7. *// to leave room for the stack* 8. numPages = divRoundUp(size, PageSize); 9. size=numPages\*PageSize; 10. *//create swap file* 11. fileSystem->Remove(swapFileName); 12. fileSystem->Create(swapFileName,size); 13. ...   以上代码就包括了生成文件名、获取大小、删除原文件、创建文件的逻辑。  有了交换文件，我们就可以将内容从内存写到交换文件，或从交换文件读到内存中了。我们需要将这两个函数进行编写，以便于页置换算法调用。 1.2.2 将内容从内存写出到文件 在页置换算法中，被置换出的页如果被修改了（“脏”了），就需要对其进行缓存，也就是写出到交换文件中，以备下一次使用。而如果此页内容没有被写，就不需要缓存。所以在页置换算法中调用此方法时，只有在旧页脏的时候才需要将其写入交换文件。Nachos会在进程运行过程中修改的页面的dirty标志设为true。  最终编写的方法如下，此方法负责将页号为oldPage的虚拟页置换出去：   1. void 2. AddrSpace::writeOut(int oldPage){ 3. printf("swapping out vPage:%d ...\t",oldPage); 4. if(pageTable[oldPage].dirty){ 5. printf("Dirty! It will be written into disk\n"); 6. OpenFile \*swapFile=fileSystem->Open(swapFileName); 7. if(swapFile==NULL){ 8. printf("Unable to open swap file %s\n",swapFileName); 9. return; 10. } 11. swapFile->WriteAt(&(machine->mainMemory[pageTable[oldPage].physicalPage\*PageSize]),PageSize,oldPage\*PageSize); 12. stats->numPageWriteOuts++; 13. delete swapFile; 14. }else{ 15. printf("Clean! No need to write into disk\n"); 16. } 17. }   其中还包括了文件读取异常等的检测以及调试信息的输出。核心逻辑为：swapFile->WriteAt(&(machine->mainMemory[pageTable[oldPage].physicalPage\*PageSize]),PageSize,oldPage\*PageSize);  其中，pageTable[oldPage].physicalPage\*PageSize是此页内容在物理内存中的起始字节的位置,从主存的此位置开始，写一页大小（PageSize）的内容放入交换文件中该虚拟页对应的位置（oldPage\*PageSize）中。 1.2.3 将内容从文件读取到内存 页置换算法中，将某页换入时需要将其历史内容重新写入内存中，也就是从交换文件的某个位置转移到物理内存中：   1. void 2. AddrSpace::readIn(int newPage){ 3. OpenFile \*swapFile=fileSystem->Open(swapFileName); 4. if(swapFile==NULL){ 5. printf("Unable to open swap file %s\n",swapFileName); 6. return; 7. } 8. swapFile->ReadAt(&(machine->mainMemory[pageTable[newPage].physicalPage\*PageSize]),PageSize,newPage\*PageSize); 9. delete swapFile; 10. printf("vPage:%d has been read into mem\n",newPage); 11. }   由于不需要判断是否需要读取（即使是第一次非代码段/初始数据段，读取的也是空数据，且仅会造成一次额外开销），所以逻辑较写入简单一些。 1.2.4 初始化地址空间 对于纯按需调页，初始化时所有虚拟页均为无效，以便在执行中触发页错误，将需要的内存页写入内存。即在初始化时将有效位置非：   1. pageTable[i].valid = FALSE;   然而这带来了一些问题，这样一来代码段和初始数据段无法在一开始写入内存。然而我们可以将这部分写到交换文件中，这样在访问代码段触发页错误的时候就可以将这部分内容读入到内存的正确位置：   1. OpenFile \*swapFile=fileSystem->Open(swapFileName); 2. if(swapFile==NULL){ 3. printf("Unable to open swap file %s\n",swapFileName); 4. return; 5. } 6. if(noffH.code.size>0){ 7. Segment seg=noffH.code; 8. char tmpBuff[seg.size]; 9. executable->ReadAt(tmpBuff,seg.size,seg.inFileAddr); 10. swapFile->WriteAt(tmpBuff,seg.size,seg.virtualAddr); 11. }   以上代码在操作时使用了一个临时字符串tmpBuff作为中转。  将初始数据段写入交换文件的逻辑与以上代码一致，在此不加赘述。 1.2.5 FIFO页置换算法 为了实现这一算法，我们需要维护一个长度有上限的队列（也就是为每个进程分配的帧数，根据实验指导书建议，此值设为5），我们为此在addrspace.h中定义了一个常数：   1. #define NumUserProcessFrame 5   队列的维护上，我们选用了在线程调度时用过的链表实现的队列，但是需要额外维护一个队列长度作为实例变量，因为每次遍历链表获得长度开销较大，在list.h中：   1. private: 2. ... 3. int size;   然后在入队和出队方法中对其进行增减即可，还有它的get方法，在此不加赘述。  然后实现算法本身。我们将该算法封装为AddrSpace::FIFO(int newPage)，它可以完成将新页换入、旧页换出相关的逻辑。  需要进行页置换时，首先将新页号入队，然后判断当前进程占用的帧是否超过为其固定分配的值（也就是判断队列长度是否超过上限）。如果队列过长，将队尾的页出队：   1. pageQueue->Append((void\*)newPage); 2. int oldPage=-1; 3. if(pageQueue->GetSize()>NumUserProcessFrame) 4. oldPage=(int)pageQueue->Remove();   至此，我们完成了队列结构的维护并获取了需要换出的页号，还需要根据这个页号对内存进行一些操作。  需要换出旧页时，我们需要将其内容写入磁盘、有效位置非、将物理页号传递到新页。而不需要换出时（当前进程占用的帧未达到上限），新页对应的物理页从位图中分配：   1. if(oldPage!=-1){*//need to swap out an old page* 2. printf("\tout:vNum: %d, physPage:%d\n",oldPage,pageTable[oldPage].physicalPage); 3. writeOut(oldPage); 4. pageTable[oldPage].valid=FALSE; 5. pageTable[newPage].physicalPage=pageTable[oldPage].physicalPage; 6. } 7. else pageTable[newPage].physicalPage=freeMap->Find();*//limit not reached*   然后，将新页的各个标志位重新设置为可用、不脏、非只读：   1. pageTable[newPage].valid=TRUE; 2. pageTable[newPage].dirty=FALSE; 3. pageTable[newPage].readOnly=FALSE;   最后将新页内容从交换文件读出：   1. readIn(newPage);   如此就完成了整个FIFO算法的数据结构维护与内存、磁盘的操作。 1.2.6 缺页异常的捕获与处理 根据分析，进程在试图访问无效页时抛出缺页异常，并将此次访存的地址写入BadVAddrReg寄存器。我们需要在exception.cc中对其进行捕获，并转页错误的中断处理程序：   1. ... 2. if(which==PageFaultException){ 3. int badVAddr=machine->ReadRegister(BadVAddrReg); 4. printf("page fault exception badVAddr:%d\n",badVAddr); 5. interrupt->PageFault(badVAddr); 6. }   在interrupt.cc中，我们实现了Interrupt::PageFault(int badVAddr)函数，它将调用FIFO算法将此地址所在虚拟页换入内存，使程序得以继续运行。   1. void 2. Interrupt::PageFault(int badVAddr){ 3. int newPage=badVAddr/PageSize; 4. currentThread->space->FIFO(newPage); 5. }   至此所有关于页置换的逻辑均已编写完毕。 1.2.7 增加打印内容 根据示例程序，虚拟页表的打印相较于lab6添加了状态位的打印，我们相应地修改AddrSpace::Print()使其在打印虚拟页表时打印状态位信息：   1. void 2. AddrSpace::Print(){ 3. printf("SpaceId:%d\n",spaceId); 4. printf("page table dump: %d pages in total\n",numPages); 5. printf("========================================================================================\n"); 6. printf("\tVirtPage, \tPhysPage, \tValid, \t\tUse, \t\tDirty\n"); 7. for(int i=0;i<numPages;i++) 8. printf("\t%d, \t\t%d, \t\t%d, \t\t%d, \t\t%d\n",pageTable[i].virtualPage,pageTable[i].physicalPage,pageTable[i].valid,pageTable[i].use,pageTable[i].dirty); 9. printf("========================================================================================\n"); 10. }  二、修改统计类2.1 解决问题的思路 首先查看stats.h，其中有很多系统运行时的统计信息：   1. int numDiskReads;  *// number of disk read requests* 2. int numDiskWrites;  *// number of disk write requests* 3. int numConsoleCharsRead; *// number of characters read from the keyboard* 4. int numConsoleCharsWritten; *// number of characters written to the display* 5. int numPacketsSent;  *// number of packets sent over the network* 6. int numPacketsRecvd; *// number of packets received over the network*   我们要统计页故障次数、将牺牲页写入交换空间的次数，只需要在其中加入两个实例变量即可。  由于Statistics对象会在系统启动时初始化，并且运行时一直只有这一个单例，那么在需要修改其中状态值的时候直接在方法中修改即可。  最后在打印统计信息的方法中，加上我们需要统计的信息。 2.2 实验步骤与代码说明 首先在stats.h中加入存储页故障、写入交换空间次数的实例变量：   1. public: 2. ... 3. int numPageWriteOuts; *// number of virtual memory page write into disk when dirty* 4. int numPageFaults;  *// number of virtual memory page faults*   相应地，在最后的打印部分stats.cc的Statistics::Print()函数中输出他们的信息：   1. printf("Paging: faults %d, write out %d\n", numPageFaults,numPageWriteOuts);   然后就要在对应位置进行页错误、交换次数的统计了。  页错误统计：在exception.cc的异常处理函数中，捕获到缺页异常时：   1. if(which==PageFaultException){ 2. stats->numPageFaults++; 3. ...   交换次数统计：在AddrSpace::WriteOut()函数中：   1. if(pageTable[oldPage].dirty){ 2. ... 3. stats->numPageWriteOuts++; 4. ... 5. }   这样就完成了统计信息的增加，新增的统计信息会在nachos停止时随其他信息一起打印。 三、调试过程及记录 在VCS中可以看到编写和调试的主要过程：    调试过程中最让我们头疼的就是一个笔误性质的东西：    从磁盘读入/写出的时候，物理内存地址忘记从页转换到字节了…为此我们还用上了vscode的Hexdump插件，比较示例程序和我们的程序生成的交换文件的内容。一度以为是初始化代码段进入交换文件的问题，还更换了把二进制文件写入交换文件的方法：先读到内存里，再逐页写出…然后发现把页内容打印出来时内容没有问题，但写到文件里就出问题了。  经由内存的交换文件：    不经由内存的交换文件：    最终我们定位出问题出在写磁盘的方法上，就发现了这个问题，然后程序就可以正常运行了。 四、运行结果 输出的行数太多，就输出到文件，只截取前后一些部分放到实验报告：    此文件最终有39960行，显然不适合全部贴出：    文件从头开始：  SpaceId:0  page table dump: 18 pages in total  ========================================================================================  VirtPage, PhysPage, Valid, Use, Dirty  0, -1, 0, 0, 0  1, -1, 0, 0, 0  2, -1, 0, 0, 0  3, -1, 0, 0, 0  4, -1, 0, 0, 0  5, -1, 0, 0, 0  6, -1, 0, 0, 0  7, -1, 0, 0, 0  8, -1, 0, 0, 0  9, -1, 0, 0, 0  10, -1, 0, 0, 0  11, -1, 0, 0, 0  12, -1, 0, 0, 0  13, -1, 0, 0, 0  14, -1, 0, 0, 0  15, -1, 0, 0, 0  16, -1, 0, 0, 0  17, -1, 0, 0, 0  ========================================================================================  page fault exception badVAddr:0  page swapping...  in:vNum: 0  vPage:0 has been read into mem  SpaceId:0  page table dump: 18 pages in total  ========================================================================================  VirtPage, PhysPage, Valid, Use, Dirty  0, 0, 1, 0, 0  1, -1, 0, 0, 0  2, -1, 0, 0, 0  3, -1, 0, 0, 0  4, -1, 0, 0, 0  5, -1, 0, 0, 0  6, -1, 0, 0, 0  7, -1, 0, 0, 0  8, -1, 0, 0, 0  9, -1, 0, 0, 0  10, -1, 0, 0, 0  11, -1, 0, 0, 0  12, -1, 0, 0, 0  13, -1, 0, 0, 0  14, -1, 0, 0, 0  15, -1, 0, 0, 0  16, -1, 0, 0, 0  17, -1, 0, 0, 0  ========================================================================================  page fault exception badVAddr:224  page swapping...  in:vNum: 1  vPage:1 has been read into mem  SpaceId:0  page table dump: 18 pages in total  ========================================================================================  VirtPage, PhysPage, Valid, Use, Dirty  0, 0, 1, 1, 0  1, 1, 1, 0, 0  2, -1, 0, 0, 0  3, -1, 0, 0, 0  4, -1, 0, 0, 0  5, -1, 0, 0, 0  6, -1, 0, 0, 0  7, -1, 0, 0, 0  8, -1, 0, 0, 0  9, -1, 0, 0, 0  10, -1, 0, 0, 0  11, -1, 0, 0, 0  12, -1, 0, 0, 0  13, -1, 0, 0, 0  14, -1, 0, 0, 0  15, -1, 0, 0, 0  16, -1, 0, 0, 0  17, -1, 0, 0, 0  ========================================================================================  page fault exception badVAddr:2284  page swapping...  in:vNum: 17  vPage:17 has been read into mem  SpaceId:0  page table dump: 18 pages in total  ========================================================================================  VirtPage, PhysPage, Valid, Use, Dirty  0, 0, 1, 1, 0  1, 1, 1, 1, 0  2, -1, 0, 0, 0  3, -1, 0, 0, 0  4, -1, 0, 0, 0  5, -1, 0, 0, 0  6, -1, 0, 0, 0  7, -1, 0, 0, 0  8, -1, 0, 0, 0  9, -1, 0, 0, 0  10, -1, 0, 0, 0  11, -1, 0, 0, 0  12, -1, 0, 0, 0  13, -1, 0, 0, 0  14, -1, 0, 0, 0  15, -1, 0, 0, 0  16, -1, 0, 0, 0  17, 2, 1, 0, 0  ========================================================================================  page fault exception badVAddr:256  page swapping...  in:vNum: 2  vPage:2 has been read into mem  SpaceId:0  page table dump: 18 pages in total  ========================================================================================  VirtPage, PhysPage, Valid, Use, Dirty  0, 0, 1, 1, 0  1, 1, 1, 1, 0  2, 3, 1, 0, 0  3, -1, 0, 0, 0  4, -1, 0, 0, 0  5, -1, 0, 0, 0  6, -1, 0, 0, 0  7, -1, 0, 0, 0  8, -1, 0, 0, 0  9, -1, 0, 0, 0  10, -1, 0, 0, 0  11, -1, 0, 0, 0  12, -1, 0, 0, 0  13, -1, 0, 0, 0  14, -1, 0, 0, 0  15, -1, 0, 0, 0  16, -1, 0, 0, 0  17, 2, 1, 1, 1  ========================================================================================  page fault exception badVAddr:784  page swapping...  in:vNum: 6  vPage:6 has been read into mem  SpaceId:0  page table dump: 18 pages in total  ========================================================================================  VirtPage, PhysPage, Valid, Use, Dirty  0, 0, 1, 1, 0  1, 1, 1, 1, 0  2, 3, 1, 1, 0  3, -1, 0, 0, 0  4, -1, 0, 0, 0  5, -1, 0, 0, 0  6, 4, 1, 0, 0  7, -1, 0, 0, 0  8, -1, 0, 0, 0  9, -1, 0, 0, 0  10, -1, 0, 0, 0  11, -1, 0, 0, 0  12, -1, 0, 0, 0  13, -1, 0, 0, 0  14, -1, 0, 0, 0  15, -1, 0, 0, 0  16, -1, 0, 0, 0  17, 2, 1, 1, 1  ========================================================================================  page fault exception badVAddr:896  page swapping...  in:vNum: 7  out:vNum: 0, physPage:0  swapping out vPage:0 ... Clean! No need to write into disk  vPage:7 has been read into mem  （中间部分略过）  SpaceId:0  page table dump: 18 pages in total  ========================================================================================  VirtPage, PhysPage, Valid, Use, Dirty  0, 0, 0, 1, 0  1, 3, 1, 1, 0  2, 1, 1, 1, 0  3, 4, 1, 1, 0  4, 2, 0, 1, 0  5, 3, 0, 1, 0  6, 2, 1, 1, 0  7, 3, 0, 1, 1  8, 3, 0, 1, 1  9, 0, 0, 1, 1  10, -1, 0, 0, 0  11, -1, 0, 0, 0  12, -1, 0, 0, 0  13, -1, 0, 0, 0  14, -1, 0, 0, 0  15, -1, 0, 0, 0  16, -1, 0, 0, 0  17, 0, 1, 1, 1  ========================================================================================  0  page fault exception badVAddr:692  page swapping...  in:vNum: 5  out:vNum: 3, physPage:4  swapping out vPage:3 ... Clean! No need to write into disk  vPage:5 has been read into mem  SpaceId:0  page table dump: 18 pages in total  ========================================================================================  VirtPage, PhysPage, Valid, Use, Dirty  0, 0, 0, 1, 0  1, 3, 1, 1, 0  2, 1, 1, 1, 0  3, 4, 0, 1, 0  4, 2, 0, 1, 0  5, 4, 1, 1, 0  6, 2, 1, 1, 0  7, 3, 0, 1, 1  8, 3, 0, 1, 1  9, 0, 0, 1, 1  10, -1, 0, 0, 0  11, -1, 0, 0, 0  12, -1, 0, 0, 0  13, -1, 0, 0, 0  14, -1, 0, 0, 0  15, -1, 0, 0, 0  16, -1, 0, 0, 0  17, 0, 1, 1, 1  ========================================================================================  1  page fault exception badVAddr:1176  page swapping...  in:vNum: 9  out:vNum: 17, physPage:0  swapping out vPage:17 ... Dirty! It will be written into disk  vPage:9 has been read into mem  SpaceId:0  page table dump: 18 pages in total  ========================================================================================  VirtPage, PhysPage, Valid, Use, Dirty  0, 0, 0, 1, 0  1, 3, 1, 1, 0  2, 1, 1, 1, 0  3, 4, 0, 1, 0  4, 2, 0, 1, 0  5, 4, 1, 1, 0  6, 2, 1, 1, 0  7, 3, 0, 1, 1  8, 3, 0, 1, 1  9, 0, 1, 1, 0  10, -1, 0, 0, 0  11, -1, 0, 0, 0  12, -1, 0, 0, 0  13, -1, 0, 0, 0  14, -1, 0, 0, 0  15, -1, 0, 0, 0  16, -1, 0, 0, 0  17, 0, 0, 1, 1  ========================================================================================  98  99  100  page fault exception badVAddr:16  page swapping...  in:vNum: 0  out:vNum: 2, physPage:1  swapping out vPage:2 ... Clean! No need to write into disk  vPage:0 has been read into mem  SpaceId:0  page table dump: 18 pages in total  ========================================================================================  VirtPage, PhysPage, Valid, Use, Dirty  0, 1, 1, 1, 0  1, 3, 1, 1, 0  2, 1, 0, 1, 0  3, 4, 0, 1, 0  4, 2, 0, 1, 0  5, 4, 1, 1, 0  6, 2, 1, 1, 0  7, 3, 0, 1, 1  8, 3, 0, 1, 1  9, 0, 1, 1, 0  10, -1, 0, 0, 0  11, -1, 0, 0, 0  12, -1, 0, 0, 0  13, -1, 0, 0, 0  14, -1, 0, 0, 0  15, -1, 0, 0, 0  16, -1, 0, 0, 0  17, 0, 0, 1, 1  ========================================================================================  Machine halting!  Ticks: total 332531, idle 0, system 10, user 332521  Disk I/O: reads 0, writes 0  Console I/O: reads 0, writes 0  Paging: faults 1377, write out 610  Network I/O: packets received 0, sent 0  Cleaning up...  文件到此结束。  注意到页表信息以外，还是输出了正确结果的：    对应排序程序的以下部分：   1. PrintInt(A[0]); */\* and then we're done -- should be 0! \*/* 2. PrintInt(A[1]); */\* should be 1 \*/* 3. PrintInt(A[ARRAYSIZE - 2]); */\* should be ARRAYSIZE - 2 \*/* 4. PrintInt(A[ARRAYSIZE - 1]); */\* should be ARRAYSIZE - 1 \*/* 5. PrintInt(ARRAYSIZE); */\* should be ARRAYSIZE \*/*   默认ARRAYSIZE为100，所以认为结果正确。  此外我们还发现，我们的页错误数与写入交换空间的次数与示例程序同样条件下一致（即./n7 -pra 1 -x ../test/sort.noff）：    均为1377/610。说明我们的页置换算法实现的也比较合理。 五、user ticks相关的分析 ARRAYSIZE均为100，固定默认5帧。  FIFO：332521    二次机会：332367    增强二次机会：332200    LRU：332033    可见，不同算法的user ticks是不同的。  而系统跳数是一样的，且页置换算法对于这个用户排序程序的逻辑是透明的，不同页置换算法不影响排序算法的逻辑。那么我们推测原因是页错误的处理程序是在用户空间进行的，而不同页置换算法淘汰页的机制不同，在同样的程序下表现可能会不一样（包括换出的次数不一样、页置换算法逻辑不一样）。 六、最优页置换的实现思路 在已知引用串的情况下，可以实现最优页置换算法。  页置换算法的实现思路为贪心算法，即在引用串的某个位置选择牺牲页时，选择（当前位置起）第一次出现位置最靠后的页A牺牲掉，若某页自此以后再也没出现，那么其位置视为无穷靠后。即牺牲自此以后最长时间不出现的页。  可以证明，如果选择次靠后的页B牺牲，结果不会更优，因为在页A触发页错误之前，页B一定会触发至少一次页错误。  具体实现时，可以先从后往前遍历引用串，使每个位置记录下一次出现相同页的位置。若某位置为最后一次出现，则下一次位置为正无穷。  设引用串长n，虚拟页共m页，引用串为ref[n]，下一次出现位置为nxtPos[n]，此页最近出现位置pos[m]：   1. for(int i=n-1;i;i--){ 2. nxtPos[i]=pos[ref[i]]; 3. pos[ref[i]]=i; 4. }   此前需要将pos数组初始化为正无穷。则此代码可以完成其初始化。  那么在进程运行时，我们维护一个优先队列，具体可用大根堆实现，其节点权值为“下一次出现位置”，附带页号信息，即（位置，页号）。当第i页需要换入，若需要牺牲，则牺牲堆顶节点的页号，然后将（下一次位置nxtPos[i]，新页号ref[i]）入堆：   1. struct Node{int nxtPos,pageNum;} 2. priority\_queue<Node> pq; 3. void OPT(int i){ 4. if(分配的帧已用完){ 5. Node sacrifice=pq.front(); 6. pq.pop(); 7. swapOut(sacrifice.pageNum); 8. } 9. pq.push((Node){nxtPos[i],ref[i]}); 10. swapIn(ref[i]); 11. }   则全程时间复杂度为O(nlogn)，空间复杂度为O(n)。如果为每个进程分配的页数较少，则相比起读写磁盘的开销，几乎可以忽略不计。 | |
| 结论分析与体会：  经过本次实验，我们进一步体验了在较复杂系统上增加功能的过程，需要先对此前系统的机制与实现方式了解透彻，修改起来才更加方便。实现的时候需要注意细节，毕竟不是自己的系统，不要想当然地认为参数是什么样的。  这次实验又是一个c++的小工程，对于此前不熟悉c++的我们来说还是有一些困难的，经常出现小错误，写完后可以说又积累了一些c++的经验。这次出现的错误让我们尝试了gdb的使用（主要用于分析“核心已转储”的“核心”文件），但没有什么收获。gcc是比较早期的编译器，出现段错误等也不会告诉你对应源代码的位置，调试起来有些辛苦。  当然最终看到自己的页置换算法让一个用户程序运行起来时，还是很有收获感的。此前一直在使用前人的智慧，这次尝试模仿前人，有了一些效果。  亲自实现页置换算法，让我们对虚拟内存的理解更加深刻，尤其是FIFO算法需要注意的一些细节，以前根本不会考虑。还有示例程序的多种实现的对比，也让我们了解了不同页置换算法的表现和其理论上限如何获取。  这就是最后一个实验了，即使如此还只是实现了操作系统的一部分基本功能，比现代操作系统还差的很远。而真正的操作系统linux/windows的内核也仅仅是由几个人编写的。我们要好好利用这笔宝贵的财富，为信息世界增砖添瓦。 | |