|  |
| --- |
| 实验目的：  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所有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，设计并实现用户空间的虚拟内存管理。  页置换算法可以采用FIFO、二次机会、增强型二次机会、LRU等算法之一，或自己认为合适的其他算法。 |
| 硬件环境：  惠普品牌型号笔记本  Intel Core i5-8300 CPU  8GB内存  512GB SSD |
| 软件环境：  宿主机：Windows 10 21H1 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-LW |
| 实验步骤与内容：  目录：  [Lab7 虚拟内存](#lab7-虚拟内存)  [7.1 实验内容](#X3a1e2d1f0b154e169a8d72c6e62533961f26d5f)  [7.2 实验思路](#Xe66cf01dcf67818f4e16a26471bead8899be478)  [7.3 实验代码](#X98158ab022000a8ba1a58789a3d74b7008a248c)  [7.3.1 添加页统计信息](#X10e22b94c7c5c671af271cfffb45ac9d275bb7e)  [7.3.2 添加缺页错误处理](#X4ab64f5b793f3eda8b0db3a98cf6543c534318c)  [7.3.3 修改AddrSpace类](#X3a0140e580161a8fc794c1e0f849cd5270741de)  [7.3.4 修改progtest.cc](#X53cc1f1e18f541e0a16619345ea4f34c8e7d27b)  [7.4 实验结果](#X56cca31fe4fffb8d53621369bf62a0fc676b990) Lab7 虚拟内存7.1 实验内容 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所有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，设计并实现用户空间的虚拟内存管理。  页置换算法可以采用FIFO、二次机会、增强型二次机会、LRU等算法之一，或自己认为合适的其他算法。  注：可以用耗费用户内存比较多的code/test/sort.c作为虚拟内存的用户测试程序。但需要改两个地方。一是ARRAYSIZE原来定义的值1024太大，以至于程序输出刷屏太厉害，需要改小。二是最后的Exit系统调用还没有实现，可暂时简单地改为Halt系统调用 7.2 实验思路 虚拟内存的页面置换算法采用FIFO的页面置换算法，Nachos默认不使用TLB，需要在execption.cc中添加一个PageFaultException缺页错误的处理。  void ExceptionHandler(ExceptionType which) {  int type = machine->ReadRegister(2);   if (which == SyscallException) {  ...  }else if(which == PageFaultException){  printf("page fault exception %d %d\n", which, type);  // 这里处理缺页错误  }  else {  printf("Unexpected user mode exception %d %d\n", which, type);  ASSERT(FALSE);  } }  对于FIFO算法，我们使用自带的List类，将可交换的页号添加到队列中。同时在为了获取页命中率，需要定义两个全局变量记录页命中次数与总的页请求次数。 7.3 实验代码7.3.1 添加页统计信息 我在machine.h中定义了PageHitCount = 0;(记录页 Hit次数)；TranslateCount = 0(记录进程页面访问次数), 并在machine.cc中对它们初始化为0。每当调用translate.cc中的translate()函数时TranslateCount++，每当tlb命中时，PageHitCount++  int TLBHitCount; int TranslateCount;  Machine::Machine(bool debug) {  int i;   PageHitCount = 0;  TranslateCount = 0;    ... }  在停机中断函数中添加对页命中信息的打印：  void Interrupt::Halt() {  // 在程序执行结束后打印TLB信息  printf("Page Miss: %d, Page Hit: %d, Total Translate: %d, Page Hit Rate: %.2lf%%\n",  machine->TranslateCount-machine->PageHitCount,   machine->PageHitCount,   Tmachine->TranslateCount,   (double)(machine->PageHitCount\*100)/(machine->TranslateCount));   printf("Machine halting!\n\n");  stats->Print();  Cleanup(); // Never returns. } 7.3.2 添加缺页错误处理 1.13节分析过：Machine::RaiseException()会将发生异常的虚拟地址存入 Register[BadVAddrReg] ，因此我们可以从 Register[BadVAddrReg] 中获取发生异常的虚拟地址。因为缺页异常也会引发中断，因此需要交给中断处理：  void ExceptionHandler(ExceptionType which) {  int type = machine->ReadRegister(2);   if (which == SyscallException) {  switch (type) {  case SC\_Halt:{  DEBUG('a', "Shutdown, initiated by user program.\n");  interrupt->Halt();  break;  }  case SC\_Exec:{  DEBUG('a', "Shutdown, initiated by user program.\n");  printf("Execute System Call of Exec()\n");  interrupt->Exec();  AdvancePC();// 程序计数器向前推进  break;  }  default:{  printf("Unexpected user mode exception %d %d\n", which, type);  ASSERT(FALSE);  }   }  }else if(which == PageFaultException){  int BadVAddr = machine->ReadRegister(BadVAddrReg);  printf("page fault exception badAddress:%d\n", BadVAddr);  interrupt->PageFault(BadVAddr);   }  else {  printf("Unexpected user mode exception %d %d\n", which, type);  ASSERT(FALSE);  } }  PageFault(BadVAddr)的实现如下：  /\*\*  \* 系统发生缺页错误触发的中断函数  \* @param badVAddr 发生缺页错误的虚拟逻辑地址  \*/ void Interrupt::PageFault(int badVAddr) {  // 获取当前执行用户进程的虚拟页表  TranslationEntry \* virtualPageTable = currentThread->space->getPageTable();    // 虚拟页表中该页的valid=FALSE  int newPage = badVAddr / PageSize;    //调用FIFO算法确定置换页  unsigned int oldPage = currentThread->space->FIFO(newPage);   printf("发生页置换，虚拟页号: %d换出，虚拟页号: %d换入\n",oldPage,newPage);  currentThread->space->writeBack(oldPage);  virtualPageTable[oldPage].valid=false;  virtualPageTable[newPage].physicalPage = virtualPageTable[oldPage].physicalPage;  virtualPageTable[newPage].valid=true;  virtualPageTable[newPage].dirty = false;  virtualPageTable[newPage].readOnly = false;   //读取需要置换的到内存中  currentThread->space->readIn(newPage);  // 打印相关信息  currentThread->space->Print(); } 7.3.3 修改AddrSpace类 先添加**用户进程可执行文件名与页号队列**  private:   char\* fileName;  List\* pageList;// 当前内存页号队列  需要着重修改该类的构造函数，将构造函数传入的参数改为**可执行的文件名，char\* FileName**，已修改的部分如下，未修改的部分...省略。  每个用户进程最多可被分配max(4,代码段+初始化数据段的页长度=initPages)个可用的真实页表项，定义为necessaryPages，真实可用的页表项用valid=TRUE表示：  注意**代码段+初始化数据段的页表永远无法被换出**，自定义可换页长度为3，在自定义可换页初始化的时候将页号添加到FIFO队列中。  AddrSpace::AddrSpace(char\* FileName) {  fileName = FileName;// 赋值  pageList = new List;// 初始化一个FIFO队列  OpenFile \*executable = fileSystem->Open(FileName);    ...    // 整个用户进程的虚拟页长度  numPages = divRoundUp(size, PageSize);   // 用户进程的初始化部分（代码段+初始化数据段的页长度）,永远不能被换出  int initPages = divRoundUp(noffH.code.size + noffH.initData.size,PageSize);   // 用户进程无法初始化（代码段+数据段大于目前可用页长度）  ASSERT(initPages <= NumPhysPages && initPages <= freeMap->NumClear());   DEBUG('a', "Initializing address space, num pages %d, size %d\n",  numPages, size);   // 每个用户进程必须的页数（包含了之后可自由置换的页：3个）  int necessaryPages = max(4,initPages+3);   size = numPages \* PageSize;  // 声明虚拟页表  pageTable = new TranslationEntry[numPages]; // ASSERT(freeMap->NumClear() >= numPages);// 确认页面足够分配  for (i = 0; i < necessaryPages; i++) {    if(i >= initPages)// 可置换的页放入优先队列  pageList->Append((void\*)i);   // 用户进程的初始化部分的虚拟页表初始化  pageTable[i].virtualPage = i;  pageTable[i].physicalPage = freeMap->Find();  pageTable[i].valid = TRUE;  pageTable[i].use = FALSE;  pageTable[i].dirty = FALSE;  pageTable[i].readOnly = FALSE; // if the code segment was entirely on  // a separate page, we could set its  // pages to be read-only  }  for(i;i<numPages;i++){  // 用户进程的其他部分的虚拟页表的初始化  pageTable[i].virtualPage = i;  pageTable[i].physicalPage = -1;  pageTable[i].valid = false;  pageTable[i].use = FALSE;  pageTable[i].dirty = FALSE;  pageTable[i].readOnly = FALSE;  }   ... }  AddrSpace需要添加的函数如下：  public:  // 获取虚拟页表  TranslationEntry \*getPageTable();  // 打印  void Print();  // 被修改的页写回磁盘  void writeBack(int oldPage);  // 读取需要置换的页到内存中  void readIn(int newPage);  // 先进先出的置换  int FIFO(int newPage);   * getPageTable()：返回该用户进程的虚拟页表 * TranslationEntry \* AddrSpace::getPageTable(){  return pageTable; } * Print()：打印该进程的虚拟页表使用情况，**其中valid=True的页表项有效，视为在真实页表中** * void AddrSpace::Print() {  printf("spaceID: %d\n",spaceId);  printf("page table dump: %d pages in total\n", numPages);  printf("============================================\n");  printf(" VirtPage, PhysPage, valid\n");  for (int i=0; i < numPages; i++) {  printf("\t%d,\t%d,\t%d\n", pageTable[i].virtualPage,pageTable[i].physicalPage,pageTable[i].valid);  }  printf("============================================\n\n"); } * void writeBack(int oldPage)：如果修改了，将虚拟页表中oldPage号页重新写入文件 * void AddrSpace::writeBack(int oldPage) {  if(pageTable[oldPage].dirty){  printf("被修改 dirty!写回磁盘，spaceId:%d,oldPage:%d\n",spaceId,oldPage);  OpenFile \*executable = fileSystem->Open(fileName);   if (executable == NULL) {  printf("Unable to open file %s\n", fileName);  return;  }  executable->WriteAt(&(machine->mainMemory[pageTable[oldPage].physicalPage]),PageSize,oldPage\*PageSize);  delete executable;  }else{  printf("无需修改\n");  } } * void readIn(int newPage)：读取需要置换的第newPage号虚拟页表到内存中： * void AddrSpace::readIn(int newPage) {  OpenFile \*executable = fileSystem->Open(fileName);  if (executable == NULL) {  printf("Unable to open file %s\n", fileName);  return;  }  executable->ReadAt(&(machine->mainMemory[pageTable[newPage].physicalPage]),PageSize,newPage\*PageSize);  delete executable;  printf("置换页已写入内存!"); } * int FIFO(int newPage)：返回队列头部的页表项 * int AddrSpace::FIFO(int newPage) {  pageList->Append((void\*)newPage);  return (int)pageList->Remove(); }  7.3.4 修改progtest.cc 因为修改了AddrSpace的构造函数，所以要对progtest.cc进行修改，我们在命令行输入  Nachos -x 可执行文件路径(../test/exec.noff)  会自动调用这个函数  void StartProcess(char \*filename) {  ...    space = new AddrSpace(filename);   ... } |
| 结论分析与体会：  测试时需要注意的是，因为没有实现Exit()系统调用，sort.c源文件需要将Exit()改为Halt()停机。同时缩小数组规模方便观察：执行sort.noff结果如下：  **初始虚拟页表情况，此时队列为6-7-8：**  image-20211223224631299  **第一次缺页，6换出15换入，此时FIFO队列为7-8-15**：  image-20211223224754962  **第二次缺页，7换出6换入，此时FIFO队列为8-15-6：**  image-20211223224957653  **第三次缺页，8换出7换入，系统停机，可以看到一共发送3次缺页，命中率约等于100%：**  image-20211223225136957 |