山东大学 软件 学院

**操作系统课程设计** 实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202100300063 | 姓名：李彦浩 | | 班级：工软 21 |
| 学号：202100300340 | 姓名：黄幸兒 | | 班级：工软 21 |
| 实验编号：Lab7 | | | |
| 实验题目：虚拟内存 | | | |
| 实验学时：6 | | 实验日期：2023.11.20 | |
| 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：  Xxx品牌yyy型号笔记本  Intel Core i7-xxxx CPU  8GB内存  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： 目录 [1. 虚拟内存管理设计思路 3](#_Toc7903)  [1.1. 什么是虚拟内存 3](#_Toc23746)  [1.2. 虚拟内存的物理形式 3](#_Toc8899)  [1.3. 虚存-实存交互 4](#_Toc9078)  [1.3.1. 几个重要问题：如何交互 5](#_Toc29738)  [1.3.2. 交互逻辑 5](#_Toc26836)  [1.4. 逻辑上非持久化的磁盘空间 7](#_Toc30964)  [2. 用户地址空间的物理限制 8](#_Toc22250)  [2.1. 进程实存帧数 8](#_Toc18427)  [2.1.1. 应该为用户进程分配多少帧？ 9](#_Toc29218)  [2.1.2. Nachos中每个用户进程的帧数 9](#_Toc24843)  [2.2. 运行最小帧数（Minimum Number of Frames） 9](#_Toc23519)  [2.3. Thrashing 11](#_Toc9638)  [3. Pure Demand Paging 12](#_Toc9917)  [3.1. 什么是PDP（纯按需调页）？ 12](#_Toc8356)  [3.2. 缺页中断 13](#_Toc12822)  [3.2.1. 什么时候会发生缺页中断？ 14](#_Toc23613)  [3.2.2. 引发异常：RaiseException 14](#_Toc28288)  [3.2.3. 处理异常：ExceptionHandler 15](#_Toc16640)  [3.3. Partial Exchange/Entire Exchange(PE/EE) 15](#_Toc15192)  [3.3.1. 需要内页置换的情况 15](#_Toc3509)  [3.3.2. 直接装入内存的情况 17](#_Toc19872)  [4. 内页置换算法 18](#_Toc14157)  [4.1. 实现：LRU with Stack 19](#_Toc21165)  [4.2. 构想：其他算法 21](#_Toc14661)  [4.2.1. FIFO（First-In-First-Out） 21](#_Toc21891)  [4.2.2. LRU with Counter 22](#_Toc25127)  [4.2.3. Additional-Reference-Bits 22](#_Toc25551)  [4.2.4. Second Chance/Clock 22](#_Toc17679)  [4.2.5. Enhanced Second Chance 23](#_Toc6102)  [5. Page Fault和Write Back统计 24](#_Toc94)  [5.1. LRU-Stack的统计结果对比 24](#_Toc5854)  [5.1.1. 自主实现的统计结果 24](#_Toc19202)  [5.1.2. 示例程序的统计结果 25](#_Toc20442)  [5.2. 其他算法的统计结果 25](#_Toc1274)  [5.2.1. OPT 25](#_Toc25685)  [5.2.2. FIFO 25](#_Toc11519)  [5.2.3. Second Chance/Clock 26](#_Toc13822)  [5.2.4. Enhanced Second Chance 26](#_Toc27772)  [5.2.5. Random 26](#_Toc9533)  [6. 算法性能比较 27](#_Toc13370)  [7. OPT算法 28](#_Toc12723) 1. 虚拟内存管理设计思路 虚拟内存（简称虚存）的目的都是扩展计算机内存，使其突破物理限制。但实现细节上有各种不同的解决方案。因此在实现前要先探讨清楚虚存的概念，并说明我对于虚存管理的设计与实现思路。 1.1. 什么是虚拟内存 首先先来回顾一下什么是虚拟内存：虚拟内存是对物理内存的扩展，本质上不是内存空间，而是磁盘空间（外村）。  这是由于用户进程在运行时不一定需要将所有的程序、数据段保存在物理内存中，每条指令可能只使用部分程序或数据。因此我们可以暂时把不需要使用的程序、数据保存在磁盘（外存）中，等到需要使用了，再根据映射表按图索骥地去磁盘空间寻找对应的程序、数据。这样可以极大地节省内存空间。  打个比方，物理内存就8页。我现在有两个8页大小的用户进程需要执行，且每条指令执行用到的程序或数据不超过2页（也即最小帧数为2）。如果把一个进程的所有程序、数据装入物理内存，那么我们同时只能运行一个用户进程。  但如果我每个进程装入4页，剩下的部分放到磁盘。每次发现所需的程序、数据不在物理内存中就去磁盘找到对应页然后换到物理内存中。并且4>2，不会影响用户进程的执行。这样同时我们可以跑两个用户进程。  这里附上上学期理论课中的一张示意图，很形象地说明了上述思路：  Untitled 1.2. 虚拟内存的物理形式 在1.1节中提到，虚拟内存本质上不是内存，而是外村/磁盘空间。那么虚存一定是位于磁盘上的。而磁盘空间在逻辑上分为三种：  1.在物理磁盘上开辟单独的一块作为虚拟内存（有别于文件系统）  2.将一个独立的磁盘分区作为虚拟内存  3.将文件系统中的一个文件作为虚拟内存  1和2的区别其实不大，前者是获得磁盘空间后在逻辑上将一部分空间分离出来，作为虚存；而后者是直接将物理上的独立空间作为虚存。但第三者和前两者区别都比较大，它要求我们首先实现文件系统，其实相当于在磁盘上又封装一层，然后通过文件系统层去和虚存磁盘空间打交道。  听从了实验指导书的建议，我采用了第二种方式作为虚存的物理形式。在Lab4/5的文件系统实验中我们知道，建立一张盘片后，会在当前文件夹下产生一个叫做DISK的Unix文件，以此作为Nachos虚拟磁盘。  Untitled 1  那么仿照这种方式，如果系统支持虚存，就专门生成一张盘片来作为虚存管理的独立分区，我把它起名叫做SWAP。实现后运行可执行程序，会自动生成该文件：  Untitled 2  后续我们的所有与虚存的操作，全部都通过之前封装好的Nachos虚拟磁盘来进行。  在Nachos中，创建一个带有同步机制的虚拟磁盘的方式在之前的Lab4/5中也已经提到了：   |  | | --- | | // system.cc  /\*\*  \* Lab7:vmem  \* swap space for user progs  \* this one formats every Nachos setup  \* this is a global variable  \*/  SynchDisk\* swap; |   很显然，多个进程可能都要访问虚存空间，也就是swap，所以这应该是一个全局变量。 1.3. 虚存-实存交互1.3.1. 几个重要问题：如何交互 之前只是阐明了虚存的概念以及存在的物理形式。那虚存是如何与实存交互的呢？为了完成交互操作我们应该如何组织虚实存的代码结构？需要哪些数据结构来支持？这些部分还是模糊不清的。我总结了几个虚实存交互时比较重要的问题：  1.如何判断当前指令使用到的页是否处于物理内存中：如果是，我们就可以直接使用想要的数据；如果不是，我们要想办法把对应页装载进内存。  2.如何建立用户进程的逻辑地址空间与虚存的映射关系：我们知道，页表是用户逻辑地址空间与真实物理内存的映射关系，有了页表，我们才能在指令传达一个逻辑地址时将其转化为物理内存地址。那么当我们需要将需要的页装入内存时，也需要这么一个映射关系，用来找到某个逻辑地址对应的虚存地址。  3.如何确定用户进程帧达到物理上限后如何置换物理页与虚存页：前面提到过，我们不可能给某个用户进程无限分配物理内存，这可能导致其他用户进程没有可分配的地址空间。因此当当前进程装入了过多的虚存页到物理内存后，我们要想办法在下一次它再装入另外的虚存页时替换物理内存中某些之前装入的页。  4.如何为用户进程分配虚存空间：我们知道，当一个用户进程仍在执行时，其他进程是不允许随意修改它的地址空间下的程序或数据的。这一点对于虚存同样适用。某个进程的虚存不能随意被其他进程更改。这就需要我们为虚存空间标明属主，每个进程都只能读入、写回或修改自己下属的虚存空间。 1.3.2. 交互逻辑 我将分别对1.3.1节中提到的问题作出解答：  1.我们可以使用一个(in)valid-bit位来标记当前页表中的某一项是否处于物理内存中。当一个页从虚存装载或置换进物理内存时，将该页表位置为1；当一个页置换出物理内存时，将该页表位置为0。页表中所有项该位初始值都是0。  2.我们可以在页表中加一项来标记物理内存-地址空间-虚存这三者的映射关系，其中地址空间是桥梁，物理内存和虚存都转换到用户的逻辑地址空间，然后再通过MMU进行转换，进而得到物理内存/虚存的映射结果。  3.这涉及到页面置换算法。由于页面置换算法有很多种，我们可以采用面向对象的思想，先提供一个类似于接口的模块，这样就可以通过依赖于抽象从而任意替换不同的页面置换算法。  4.如同为用户进程分配物理内存一样，我们依然可以采用一个位图（bitmap）来标识虚存中哪些页已经被某个用户进程占据。  示意图如下：  Untitled 3  对于具体实现方面：   1. 在Nachos源代码中，已经提供了valid位：  |  | | --- | | // translation.h  bool valid;  // If this bit is set, the translation is ignored.  //(In other words, the entry hasn't been initialized.) |   我们需要做的就是：  1.在初始化地址空间时将该位设置为False（因为我们是Pure Demand Paging）  2.之后在每次装入或置换时，将进入物理内存的页的valid置为true，换出物理内存的页的valid置为false。  2.这里我选择在页表项中添加一项，用来标识虚存页号，例如：   |  |  |  | | --- | --- | --- | | Physical | Logical | Virtual Memory | | x | 0 | α | | y | 1 | β |   这意味着用户地址空间的第0页，对应物理内存的第x帧，对应虚存中第α帧。   |  | | --- | | // translate.h  class TranslationEntry {  public:  int virtualPage;  int physicalPage;  /\*\*  \* Lab7:vmem  \* where to find this page in the virtual memory  \*/  int vMemPage;  bool valid;  bool readOnly;  bool use;  bool dirty;  }; |   3.我们可以来看一下页面置换算法的特点：不需要输入，给出一个被置换页面的索引。我们把这个特点抽象出来，就可以封装出一个类似于接口的函数：   |  | | --- | | // replace.cc  int calVictim(){  // calculate the vicitm  // results depend on different algorithm  // ...  } |   我们现在还不用在意具体的实现算法。重点在于我们要向缺页中断的处理方提供这样一个接口，这样我们就可以把注意力暂时放在整个虚存管理架构上，而不是算法的细节上。  4.显然这个管理虚存空间的位图应该是个全局的单例，因为我的设想是所有用户进程都是用同一块独立出来的磁盘分区，所以每个进程都能访问这个位图：   |  | | --- | | // system.cc  /\*\*  \* Lab7:vmem  \* virtual mem bit map to notificate the usage of sectors on the swap disk  \* also formats every Nachos setup  \*/  BitMap\* vmMap |   需要注意的是，本节中的内容只涉及了虚存管理架构的设计与一些定义的声明，具体实现还要看之后的内容。 1.4. 逻辑上非持久化的磁盘空间 本节是对1.3节的一个补充。在1.3节中我们提到要用一个位图来管理虚存空间，但我们知道位图在每次Nachos虚拟机启动时都会被重新初始化，那也就是说我们丢失了之前分配虚存空间的结果？  事实上确实丢失了，但这对虚存没有任何影响。这是因为虚存是用来暂存用户进程的，一个用户进程需要运行时我们才会在虚存中为其分配空间，当用户进程结束我们要在虚存中释放为其分配的空间。也就是说虚存里的数据是不需要持久化的。上一次虚拟机虚存中的数据可能和本次运行时虚存中的数据完全不同！所以，**即便虚存的物理形式是磁盘空间，但在逻辑上，它是非持久化的**。直到用户进程发生缺页，否则我们不会关心虚存中的数据。 2. 用户地址空间的物理限制 之前在第1节中提到了，如果我们部分装载用户进程的程序或数据到物理内存，就可以剩下内存并分配给其它进程。所以虽然虚存在逻辑上扩大了物理内存空间，但在实际上反而还需要压缩每个用户进程的物理内存限制。  举个例子，虚存将内存从4KB扩展到了128KB，其中一页等于一帧，大小为128B。现在有两个线程，每个地址空间4KB。采用虚存，我们可以每个进程分配16页，同时跑两个进程，每个进程2KB内存大小。不采用虚存，我们只能跑一个进程，但是这一个进程的内存占用大小是4KB，大于虚存管理下的2KB。  但实际上我们知道，虚存管理下两个进程的另外2KB空间全部在虚存里，所以本质上物理内存是扩大了的，但是体现到实际物理内存中，每个进程的物理内存反而缩小了（从4KB到2KB）。  本节将介绍一些概念，虽然Nachos在实现这一部分时只是做了最简单的模拟，并且我在扩展时也没有专门为这些概念设计模块、算法等，但了解这些概念可以更清楚地理解虚存管理，以及这些概念对于虚存管理性能提升的帮助。 2.1. 进程实存帧数 首先我们要明确一点，进程发生缺页中断的概率是随该进程物理内存空间的大小的提升而递减的。说简单点，我们为一个进程分配的实际物理帧越多，它发生缺页中断的概率越小。如果我们把一个进程的所有程序或数据都装入物理内存，那么它一定不会发生缺页中断。如下图：  Untitled 4 **2.1.1. 应该为用户进程分配多少帧？**  所以看似用户进程的帧数越大越好？肯定不是，第1节已经强调了，用户进程的帧数越少，我们就能同时执行越多的进程。  所以一个进程的实存帧数其实要权衡缺页中断与并发性能来确定，太小了经常缺页，太大了并发能力太差。  我们可以通过先确定并发能力的方式来确定分配的帧数。现在假设物理内存页有64帧。我们的系统想要支持至少16个线程的并发能力。那么最好是每个进程拥有64/16=4帧。 2.1.2. Nachos中每个用户进程的帧数 为了简单起见，在实现Nachos虚存管理时，我硬性分配了Nachos每个用户进程在物理内存中最多拥有5帧，需要再多就得内页置换了。   |  | | --- | | // addrspace.h  #define maxInUse 5 |   但实际上分配帧没有这么简单，不同进程在同一时间使用的帧数也是不一样的，这就是即将在2.2节中提到的运行最小帧数。 2.2. 运行最小帧数（Minimum Number of Frames） 进程要正常运行，是需要一个最小帧数下限的，为该进程分配的帧数如果低于此下限，进程将陷入永无止境的缺页中断。  现在我来举个例子说明MoF的必要性：  假设一个进程需要执行一条指令：  LW r0 r37 304  这条指令本身属于程序段。但它需要去读逻辑内存地址为(r0+304)字节的内容。  1.现在假设我们只给当前进程分配了1帧（0帧肯定是不可能的），那么在抓取指令时将会从PC指针的位置往后读4个字节：   |  | | --- | | if (!machine->ReadMem(registers[PCReg], 4, &raw)) |   2.随后它会尝试去读对应逻辑地址：   |  | | --- | | // mipssim.cc  case OP\_LW:  tmp = registers[instr->rs] + instr->extra;  if (tmp & 0x3) {  RaiseException(AddressErrorException, tmp);  return;  }  // translate.cc  bool  Machine::ReadMem(int addr, int size, int \*value)  {  // ...  exception = Translate(addr, &physicalAddress, size, FALSE);  // ... |   3.但Translate函数（也即MMU）会告诉它，你要读的(r0+304)内存地址，目前没有被装载到内存中，如果你现在去读，就是错的，所以我会先帮你把虚存中的数据先装载到对应内容，然后你再去读。   |  | | --- | | // translate.cc  ExceptionType  Machine::Translate(int virtAddr, int\* physAddr, int size, bool writing)  {  //...  else if (!pageTable[vpn].valid) {  DEBUG('a', "virtual page # %d too large for page table size %d!\n",  virtAddr, pageTableSize);  return PageFaultException;  } |   4.那么缺页中断处理函数就会装载对应的虚存页到实存中。但是注意，当前进程的帧数只有1帧！所以它会替换现在的这一帧，也就是说现在指令LW不在内存里了！  5.好了，现在(r0+304)地址上的内容已经从虚存装载到实存里了。接下来可以继续执行指令了。但是，现在指令不在内存里了！当CPU重新读取指令时，MMU会告诉它，你想读取的指令已经不在内存里了，我先帮你把它从虚存置换进来吧！随后MMU就把该指令对应帧置换进来，那么(r0+304)地址对应帧就又从内存出去了。  6.当置换结束后，CPU终于能抓到这条指令了，解析完一看，原来是LW指令，那就需要去读对应的(r0+304）地址了，但是(r0+304)地址刚刚在第5步时又被换出去了。注意，CPU是不会记得它是否读过这个地址并把对应内容装载到寄存器里的，也就是说它不记得自己第4步时候的所作所为，也就是说它不知道当前r37寄存器中的内容已经是它想要的内容这件事。  那(r0+304)不在内存里怎么办呢？当然是继续置换……  欸？这个过程是不是有点眼熟，第6步做的事情和第3步完全一样！都是把当前页换到虚存，然后把需要的缺页从虚存换回内存。这就造成了永无止境的缺页中断。  也就是说CPU是无记忆的，它只是按部就班地按照内存中的程序执行而不做任何干涉，以上的过程就可以理解成：  CPU发现数据不在内存，他就调动缺页中断处理函数去找数据，但找到了数据他就把指令忘了。因此还得重新找指令，一找到指令就又重新把数据忘了，那还得重新找数据……循环往复。  为了避免CPU的“健忘症”，我们就得把每条指令要用到的内容全部保存到内存中，防止这样永无止境的寻找-忘记-再寻找的过程。这就是最小运行帧数MoF的含义。 2.3. Thrashing Thrashing(颠簸)其实是承接2.2节的概念。2.2节我们介绍了缺页中断最极限的情况，也就是死循环的情况。颠簸虽然没有达到这个极限，但是也出现频繁的缺页，极大地影响了系统性能。  颠簸现象的本质原因是由CPU利用率（CPU Utilization）和缺页中断的矛盾造成的。举个例子，假设现在有多个进程，其中A正在运行，其他的处于ready态：  1.系统发现现在CPU利用率很低，同时还有另外的进程准备执行，为了提高CPU利用率，它就把一个进程B的程序和数据装载到内存。但这需要抢占A的一部分内存空间资源，因此A、B都把一部分程序或数据装入了虚存。两个进程并发。  2.由于一部分内存资源被抢走，A进程的实存帧数变少，因此缺页次数就开始增多。而缺页中断和内页置换本质上是不算在CPU的有效利用里的。这就导致CPU利用率开始下降。  3.为了提高CPU利用率，系统就又把别的进程的程序和数据装载到内存。这就导致A进程的帧数更少了，缺页次数也更多，CPU利用率更低。这个过程循环往复。  我们可以发现，为了提升CPU利用率，OS就尝试更高的并发，但CPU利用率和并发量不总是正相关关系，而是在到达某个节点后呈负相关。也就是并发越大，CPU利用率越低，如下图：  Untitled 5  这样就导致了大部分进程的缺页置换时间甚至大于了真正的执行时间，也就是颠簸现象。  颠簸现象也在提醒我们：要做好并发量和缺页中断的平衡，否则系统很容易失衡，导致性能急剧下降。 3. Pure Demand Paging 现在回到我们需要编码的部分。实验要求我们实现纯按需调度页。这影响到了具体的实现细节。我们需要先明确PDP的概念，然后才能进一步设计并实现PDP。 3.1. 什么是PDP（纯按需调页）？ PDP的核心思想就一句话：除非进程执行过程中需要该页，否则永远都不要把这一页装入到内存。  换句话说，进程的执行一定是从一次缺页中断开始的，因为进程创建时还不需要任何程序段中的指令来执行该进程，所以一开始该进程的程序或数据都在虚存，实存里一页都没有。  这一部分实现我放在了用户地址空间初始化的时候，其实也就是用户进程创建的时候：   |  | | --- | | // addrspace.cc  pageTable = new TranslationEntry[numPages];  for (i = 0; i < numPages; i++) {  /\*\*  \* Lab7:vmem  \* pure demand paging  \*/  pageTable[i].virtualPage = i;  pageTable[i].physicalPage = IllegalPhysPage;  pageTable[i].vMemPage = vmMap->Find();  pageTable[i].valid = FALSE;  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  } |   1.可以看到，用户地址空间初始化时，我在页表中为每一个逻辑页都注明了对应的物理页为IllegalPhysPage，这个值定义为-1。   |  | | --- | | pageTable[i].physicalPage = IllegalPhysPage; |   2.另外我们还要给页表的每一项找一个虚存扇区，把该页存放到该扇区中。也就是一开始用户进程的程序和数据全部都在虚存里，实存中一点没有。  3.之后我们就可以向对应虚存中写入用户程序了。   |  | | --- | | /\*\*  \* Lab7:vmem  \* Initially we load the noff file to the swap disk  \* when there's a page fault  \* fetch the data on the disk to main mem  \*/  char buf[size];  bzero(buf,size);  if (noffH.code.size > 0) {  DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",  noffH.code.virtualAddr, noffH.code.size);  executable->ReadAt(&(buf[noffH.code.virtualAddr]),  noffH.code.size, noffH.code.inFileAddr);  }  if (noffH.initData.size > 0) {  DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",  noffH.initData.virtualAddr, noffH.initData.size);  executable->ReadAt(&(buf[noffH.initData.virtualAddr]),  noffH.initData.size, noffH.initData.inFileAddr);  }  // load the buf to the swap disk  for(int i=0;i<numPages;i++){  vmWrite(pageTable[i].vMemPage,&buf[i\*PageSize]);  } |   这里附上一张截图，证明在我实现的PDP调度下，每个进程一定会以一次缺页中断开始执行。  Untitled 6  可以看到上图中，执行开始后一上来就缺第0页。随后第0页被装入内存，其他页仍在虚存。 3.2. 缺页中断 现在来看看发生缺页中断时应该如何处理。 3.2.1. 什么时候会发生缺页中断？ 先确定何时发生缺页中断。显然是MMU将逻辑地址转换为物理地址时，发现物理地址对应的程序或数据并不在内存中，那么可以判定为缺页中断。   |  | | --- | | vpn = (unsigned) virtAddr / PageSize;  offset = (unsigned) virtAddr % PageSize;  if (tlb == NULL) { // => page table => vpn is index into table  if (vpn >= pageTableSize) {  DEBUG('a', "virtual page # %d too large for page table size %d!\n",  vpn, pageTableSize);  return AddressErrorException;  } else if (!pageTable[vpn].valid) {  DEBUG('a', "virtual page # %d cause page fault trap!\n",  vpn, pageTableSize);  return PageFaultException;  }  entry = &pageTable[vpn]; |   1.首先我们把逻辑地址转换为逻辑页号，即virtAddr → vpn。  2，随后检查页表中是否有该项，如果有，检查valid为是否为1(true)，如果valid位为0(false)，代表该逻辑页不在内存里，需要从虚存中装载进来。也就是return PageFaultException 。 3.2.2. 引发异常：RaiseException 该函数只在读写内存时用到，其实这也容易理解，如果根本不是对内存的操作，那么也不可能引发缺页中断或其他异常。   |  | | --- | | void  Machine::RaiseException(ExceptionType which, int badVAddr)  {  DEBUG('m', "Exception: %s\n", exceptionNames[which]);  // if(DebugIsEnabled('r')){  // for(int i=0;i<NumTotalRegs;i++){  // printf("reg %d : %d ,",i,registers[i]);  // }  // printf("\n");  // }    // ASSERT(interrupt->getStatus() == UserMode);  registers[BadVAddrReg] = badVAddr;  DelayedLoad(0, 0); // finish anything in progress  interrupt->setStatus(SystemMode);  ExceptionHandler(which); // interrupts are enabled at this point  interrupt->setStatus(UserMode);  } |   该函数会将错误的地址传入给异常处理函数。 3.2.3. 处理异常：ExceptionHandler 在之前的Lab6中，我们在这个函数中实现了一些系统调用。现在我们仍需要修改这个函数，使它能够处理缺页异常。   |  | | --- | | else if(which == PageFaultException){  /\*\*  \* Lab7:vmem  \* page fault handle  \*/  //DEBUG('r',"bad vAddr: %d\n",machine->registers[BadVAddrReg]);  loadPage(machine->registers[BadVAddrReg]);  stats->numPageFaults++;  // there's no need to IncPc,since we should restart the instruction  // just keep the PCs as where they stay now  } |   这里需要注意的是，当缺页中断发生时，当前的指令是没被执行完的！所以不同于Lab中我们推进PC指针的做法，在缺页中断发生时，我们要保留PC的位置，以便在内页置换后可以重新执行一遍该指令。 3.3. Partial Exchange/Entire Exchange(PE/EE) 内页置换也分为多种，一种是全体置换，另一种是局部置换：  1.全体置换：将整个进程置换入内存  2.局部置换：将缺页部分置换入内存  很明显，PDP策略采用局部置换。  接下来就是缺页中断具体的处理逻辑了。显然，根据当前进程帧数以及用户进程帧数上限，可以分为以下两种情况：  1.当前进程帧数已经达到上限，需要进行内页置换  2.当前进程帧数未达到上限，无需内页置换，直接从虚存中装入即可。 3.3.1. 需要内页置换的情况  |  | | --- | | // swap.cc  /\*\*  \* Lab7: vmem  \* @param vAddr provided by the caller,calculating based on the badVAddrReg  \* @return the logical page cause page fault  \*/  int swapPage(int vAddr){  TranslationEntry\* vmpt = machine->pageTable;  // step 1: find a victim by replacement algorithm  int rPage; // rPage is short for replaced page  rPage = calVictim();  DEBUG('s',"swap victim: %d\n",rPage);  // step 2: check if victim is dirty(need to write back)  if(vmpt[rPage].dirty){  dirty2vm(rPage,vmpt);  }  int logPage = vm2Mem(vAddr,vmpt,vmpt[rPage].physicalPage\*PageSize);  // step 3: set the valid bits of both rPage and logPage,also the dirty bit  vmpt[rPage].valid = false;  vmpt[logPage].valid = true;  vmpt[rPage].dirty = false;  // step 4: update the nachos virtual mem page table  vmpt[logPage].physicalPage = vmpt[rPage].physicalPage;  vmpt[rPage].physicalPage = IllegalPhysPage;  printf("Swap page %d out, demand page %d in frame(%d)\n",rPage,logPage,vmpt[logPage].physicalPage);  return logPage;  } |   1.由于内页置换需要修改页表项的属性，因此我们先把页表拿到。  2.根据内页置换算法计算被置换的页（victim）。不过当前我们并不在意这个算法的细节，现在我们只是在完成置换页面的常规流程。  3.如果被置换的页被修改过（dirty位），那么则写回虚存。  4.然后我们把虚存中对应于缺页的内容读取到内存中。   |  | | --- | | // swap.cc  // Fetch data in vm space to main mem  int vm2Mem(int vAddr,TranslationEntry\* vmpt,int desPos){  // step 1: get the page no. of vAddr in the swap space  int logPage = vAddr/PageSize; // log is short for logical  int tPage = vmpt[logPage].vMemPage; // t is short for target  // step 2: Fetch the page in the swap space  char buf[PageSize];  DEBUG('v',"the page caused page fault is : %d,target vmem page number is : %d\n",logPage,tPage);  vmFetch(tPage,buf);  // step 3: rewrite the data in the buffer back to the physical main mem  physMemCopy(buf,machine->mainMemory,PageSize,0,desPos);  // return the logical page of vAddr  return logPage;  } |   这部分函数的具体实现如下：  1.先找到缺页的逻辑页号对应的虚存页号。  2.从虚存中读取数据，这里直接调用SynchDisk类中封装的ReadSectors函数。   |  | | --- | | void vmFetch(int secNo,char\* data){  swap->ReadSector(secNo,data);  } |   3.把这段数据复制到内存中对应的位置。  4.返回缺页的逻辑页号。  5.这样victim就被置换出去了，我们要把它的valid位和dirty位改为false。而缺页被换了进来，因此我们把它的valid位改为true。  这样我们就完成了置换部分的流程。 3.3.2. 直接装入内存的情况  |  | | --- | | //exception.cc  /\*\*  \* Lab7: vmem  \* load the page in the swap place to physical mem  \* use func:swapPage or not depends there are still enough spaces for the refStk  \*/  void loadPage(int vAddr){  // step1: check if there is enough space for the refStk  AddrSpace\* cas = currentThread->space;  DEBUG('a',"current addr space id: %d\n",cas->getAsId());  int logPage;  DEBUG('u',"current user prog frame number-in-use: %d\n",cas->numInUse);  if(cas->numInUse < maxInUse){  TranslationEntry\* vmpt = machine->pageTable;  // make sure there are enough pages on the page map  ASSERT(pageMap->NumClear()>0);  // find an idle page on the page map  int physPage = pageMap->Find();  DEBUG('u',"phys page: %d allocated to cur addr space: %d\n",physPage,cas->getAsId());  // fetch the corresponding page of vAddr on the swap disk to this phys page  logPage = vm2Mem(vAddr,vmpt,physPage\*PageSize);  // user prog's page-in-use inc  cas->numInUse++;  // update the pt  vmpt[logPage].physicalPage = physPage;  // now the logPage is in the mem  vmpt[logPage].valid = true;  printf("Demand page %d in(frame %d)\n",logPage,physPage);  }else{  logPage = swapPage(vAddr);  }  // remember to add the new ref into the refStk  refPush(logPage);  if(true){ //DebugIsEnabled('m')||DebugIsEnabled('a')  cas->Print();  }  } |   这一部分由于比较简单，我就直接写在缺页异常处理的总逻辑里了：  1.获取当前页表，因为装入页后我们需要改页表项字段。  2.由于是新装入一页，因此要在物理内存里找空闲的页分配给当前进程。通过位图实现。  3.把虚存中对应的数据装载到内存。  4.用户进程帧数加一。  5.修改页表，使得缺页的物理页为位图为其分配的页，且valid位为true。  最后的refPush是LRU算法的实现细节，这里不需要在意。  这样我们就完成了不需要内页置换，直接把用户程序装入内存的部分。 4. 内页置换算法 在第3节中，我们实现了虚存向实存装入/置换页的总体流程。但具体的寻找victim的算法细节还没有实现。内页置换算法有很多种，我们一种一种来看。 4.1. 实现：LRU with Stack 我实现的内页置换算法是基于Stack的LRU（Least Recently Used）算法。每次寻找最长时间没有用过的页置换出去。对于这类时序问题的求解，可以想到Stack这样的数据结构。越靠近栈底的引用则越长时间没有使用；越远离栈底的引用则最近被使用过。  对于具体细节：  1.每次出现之前没有用过的页时，将其置入栈顶（入栈）  2.需要置换页面时，取出栈底的引用（出栈）  3.当使用了已经在内存的页时，该页将成为最近使用过的页，我们要调整栈的结构。  示意图如下：  Untitled 7  我这里采用了数组模拟栈，其实STL的各种容器性能会更好。但我觉得我们现在在做底层的东西，直接用现成的库不太好，还是使用了比较原始的数据结构对Stack进行模拟。  首先不同进程的LRU-Stack的内容都不一样，所以我选择在每个用户进程的地址空间实例AddrSpace中维护这个数组：   |  | | --- | | // addrspace.cc  /\*\*  \* Lab7:vmem  \* LRU replacement algorithm required  \*/  int\* refStk;  int stkSize; |   refStk即数组本身，stkSize为栈当前的大小，这个值是不能超过用户进程最大帧数的。  现在我们就来实现之前所说的三个细节：  1.入栈操作：直接在栈顶添加一个引用   |  | | --- | | void refPush(int ref){  DEBUG('r',"new ref: %d\n",ref);  AddrSpace\* as = currentThread->space;  // still enough space  as->refStk[as->stkSize++] = ref;  } |   2.出栈操作：把栈底的元素取出来，并且后面的元素向栈底顺移   |  | | --- | | int refPop(){  AddrSpace\* as = currentThread->space;  int ret = as->refStk[0];  // Each element moves forward  for(int i=1;i<as->stkSize;i++){  as->refStk[i-1] = as->refStk[i];  }  as->stkSize--;  return ret;  } |   3.调整栈结构：找到目标元素，把它后面的元素向前顺移，并把目标元素放在栈顶   |  | | --- | | /\*\*  \* Lab7:vmem  \* @param refPage virtual/logical page need to be updated when there's no page fault exception  \*/  void  AddrSpace::updateRefStk(int refPage) {  int target;  for(target=0;target<stkSize;target++){  if(refStk[target]==refPage){  break;  }  }  ASSERT(refStk[target]==refPage);  int tPage = refStk[target];  for(int i=target+1;i<stkSize;i++){  refStk[i-1] = refStk[i];  }  refStk[stkSize-1]=tPage;  } |   使用该算法计算victim时，只需要简单地取出栈底元素即可。   |  | | --- | | int calVictim(){  return refPop();  } |   而从虚存向内存装载页时，也不要忘记Push最新的页面引用（第3节末我说不要在意的那行代码，现在需要注意了）。 4.2. 构想：其他算法 除了LRU-Stack算法外，还有一些其他的算法可以用来使用。  其实除了不可能应用到实际的OPT算法，没有最好的内页置换算法这一说，不同的页面置换算法面对不同的场景，表现也是不一样的，有点像ML里的NFL（No Free Lunch，没有免费的午餐）定理一样，看似笨拙的算法，也可能在特定场景下有更好的表现。 4.2.1. FIFO（First-In-First-Out） FIFO比较简单。本质就是置换当前内存页中最早用到的页，实现方式即LRU算法去掉更新栈结构之后剩下的部分  示意图如下：  Untitled 8 4.2.2. LRU with Counter 这个和LRU with Stack一样，只是给不同引用加上了个时间戳，每次选择时间戳最小的置换。既然涉及到时间，则需要计时器这样的硬件支持，这里就没有选择实现。 4.2.3. Additional-Reference-Bits 这个算法是LRU算法的扩展。它在页表中维护了一个8位的0-1串，并且设置了一个计时器，当时钟中断发生时，将0-1串整体右移，地位丢弃，将引用位（reference bit）置入该串高位。而只有当前正在使用的页的引用位为1，其他都是0。置换时选择0-1串最小的进行置换。这个算法也需要硬件支持。  示意图如下：  Untitled 9 4.2.4. Second Chance/Clock 每个在内存的页都有一个引用位：  1.如果当前使用的页在内存中，将其引用位置为1  2.如果当前缺页，遍历所有在内存的页，直到找到一个引用位为0的页，将其置换出去，在遍历途中，把沿途所有引用位为1的页引用位置为0。  示意图如下：  Untitled 10 4.2.5. Enhanced Second Chance 引用位有两位，分别代表：   |  | | --- | | (used,modified) |   置换优先级如下：Untitled 11 5. Page Fault和Write Back统计 在Nachos中，有一个模块stats，专门用来统计Nachos运行时的各项数据，比如缺页次数、系统时间、用户程序执行时间、空闲时间等。   |  | | --- | | class Statistics {  public:  int totalTicks; // Total time running Nachos  int idleTicks; // Time spent idle (no threads to run)  int systemTicks; // Time spent executing system code  int userTicks; // Time spent executing user code  // (this is also equal to # of  // user instructions executed)  int numDiskReads; // number of disk read requests  int numDiskWrites; // number of disk write requests  int numConsoleCharsRead; // number of characters read from the keyboard  int numConsoleCharsWritten; // number of characters written to the display  int numPageFaults; // number of virtual memory page faults  int numVictimWriteBack;  int numPacketsSent; // number of packets sent over the network  int numPacketsRecvd; // number of packets received over the network  Statistics(); // initialize everything to zero  void Print(); // print collected statistics  }; |   在这里面,numPageFaults是缺页次数，numVictimWriteBack是内存页写回虚存的次数。  实现对这两个字段的统计也很简单，缺页异常中断时缺页次数增一，内页置换检查是否dirty时若为dirty写回次数增一。  下面所有的统计结果均使用用户进程帧数为5，算法各异。 5.1. LRU-Stack的统计结果对比5.1.1. 自主实现的统计结果 加入该功能后，先来看看我自己实现的LRU算法这两项的表现如何：  Untitled 12  缺页次数889次，写回次数224次。 5.1.2. 示例程序的统计结果 Untitled 13  由于运行的是同一个可执行文件，因此示例程序的结果可以作为标准答案来比对。缺页次数889，写回次数224。和我实现的一样。这代表我实现的LRU算法是正确的。 5.2. 其他算法的统计结果5.2.1. OPT 实际上OPT算法是不可能运用于实际的，但是示例程序通过手动记录引用串的方式预先生成了对应的ref string，既然都知道进程执行需要的页面了，就可以使用OPT了。  Untitled 14  缺页次数450，写回次数195。 5.2.2. FIFO Untitled 15  缺页次数1377，写回次数610。 5.2.3. Second Chance/Clock Untitled 16  缺页次数1233，写回次数526。 5.2.4. Enhanced Second Chance Untitled 17  缺页次数1056，写回次数196。 5.2.5. Random 随机挑一个置换的置换算法。  Untitled 18  其实统计这个算法没有意义，因为这个算法不具有幂等性。同一个可执行文件，随机置换算法运行的统计结果可能大不相同。  其实正如我在第4节中提到的，这只是在当前可执行文件下，每个用户进程采用5帧的统计结果。除了OPT这个无法真正实现的算法，没有最好的内页置换算法这一说。这次表现不好的算法，可能换个用户进程表现就变好了。甚至最不稳定的Random算法，在你运气好的时候，性能可以做到不输OPT呢。 6. 算法性能比较 在第5节时，我们已经跑出了不同内页置换算法的userTick。这里我就用示例程序的结果了（因为我自己实现的方式不一样，userTick没有任何可比性）。列表如下：   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Algorithm | UserTick | Page Faults | Write Back | PF + WB | | OPT | 331594 | 450 | 195 | 645 | | FIFO | 332521 | 1377 | 610 | 1987 | | CLOCK | 332367 | 1233 | 526 | 1759 | | ENHANCED CLOCK | 332200 | 1056 | 196 | 1252 | | LRU-STACK | 332033 | 889 | 224 | 1113 | | RANDOM | / | / | / | / |   之前已经提到过，统计Random的任何数据都没有意义，因为这个算法没有幂等性。因此这里就把它排除在外了。  观察上表数据，可以发现，UserTick随PF+WB增加严格单调增加，换句话说，发生缺页中断和虚存写回次数越多，UserTick也越多。  这个现象是可以理解的，因为发生缺页中断时会读磁盘，读磁盘是需要时间的；发生虚存写回时也要写磁盘，写磁盘也是要时间的。   |  | | --- | | void  SynchDisk::WriteSector(int sectorNumber, char\* data)  {  lock->Acquire(); // only one disk I/O at a time  disk->WriteRequest(sectorNumber, data);  semaphore->P(); // wait for interrupt  lock->Release();  }  void  Semaphore::P()  {  IntStatus oldLevel = interrupt->SetLevel(IntOff); // disable interrupts    while (value == 0) { // semaphore not available  queue->Append((void \*)currentThread); // so go to sleep  currentThread->Sleep();  }  value--; // semaphore available,  // consume its value    (void) interrupt->SetLevel(oldLevel); // re-enable interrupts  } |   像Nachos源码实现中，信号量的PV操作要开关中断，早在Lab2我们就提到过，开过中断会引发OneTick，也就是系统时钟推进。而我们在执行用户程序时处于用户态下，时间自然算到UserTick里了。 7. OPT算法 虽然OPT算法没有办法实际应用于操作系统中，但是我们可以通过“马后炮”的方式来使用OPT算法。并观察性能最好的算法缺页中断的次数，以此来评估其他算法的性能。  大致思路如下：  1.跑其他内页置换算法时，把每条指令执行需要用到的页（无论是程序段还是数据段）都记录到一个引用串中。  2.在停机前，使用这个已经记录好的引用串，跑OPT算法。当然我们只能记录下OPT算法的缺页次数，没法记录写回次数（因为没有Dirty位，无法确定是否写回）。  对于前者，由于对于内存的操作都需要经过MMU，因此我们可以在Translate函数尾部添加一个记录引用串的函数。并且不必担心缺页中断时不会执行这个函数。因为缺页中断PC不会推进，下一次执行的指令还是当前的指令，这个指令需要用到的页的引用一定会被记录下来。  编码如下：   |  | | --- | | void  AddrSpace::updateRefStr(int refPage) {  // unnecessary to record two same contiguous refs  // if the first one has been loaded to mem  // then the second one also did  if(lastVPN == refPage){  return;  }  lastVPN = refPage;  //DEBUG('A',"refStrLen: %d\n",refStrLen);  // if there's still enough space in the refStr arr  // just use it  if(refStrLen<maxLen){  refString[refStrLen++] = refPage;  return;  }  // Otherwise,create a new arr to replace the cur one  maxLen\*=2;  int\* narr = new int[maxLen];  for(int i=0;i<refStrLen;i++){  narr[i] = refString[i];  }  narr[refStrLen++] = refPage;  delete[] refString;  refString = narr;  } |   1.这里有一个很重要的trick。**如果一个引用串有连续两个相同的引用，我们只需要记录其中一个就可以了**。想想看这是为什么，因为前面的引用一定会被置换到内存中，因此后面那个一定不会引发缺页中断。但我们统计的就是缺页中断。因此我们可以忽略后者。  2.如果和上一次引用不相同。检查引用串数组是否需要扩容？  a.如果不需要，直接在尾部加上当前引用串即可  b.需要扩容，复制数组到一个更大的数组中，并在尾部加上当前引用串。  这样我们就完成了对引用串的记录了。  对于第二部分，其实就是拿着已知的引用串跑OPT模拟，我也没有用什么高深的算法来模拟OPT算法，就是直接写了个暴力。   |  | | --- | | /\*\*  \* Lab7:vmem  \* calculate page faults and write back for OPT algorithm  \*/  int  AddrSpace::calOPTPf() {  if(DebugIsEnabled('A')){  printf("len: %d,ref str: ",refStrLen);  // for(int i=0;i<refStrLen;i++){  // printf("%d ",refString[i]);  // }  // printf("\n");  }  int ans = 0;  int opt[maxInUse];  int optSize = 0;  for(int i=0;i<maxInUse;i++){  opt[i] = -1; // vpn will never be -1  }  int victim,max,cmp;  for(int i=0;i<refStrLen;i++){  // check if vpn is already in the mem  int ref = refString[i];  //DEBUG('A',"cur ref: %d\n",ref);  bool flag = false;  for(int j=0;j<maxInUse;j++){  if(opt[j] == ref){  flag = true;  break;  }  }  if(!flag){  ans++; // not in the mem  // still enough space?  if(optSize<maxInUse){  opt[optSize++] = ref;  continue;  }  // need paging,find victim  victim = 0;  max = 0;  for(int j=0;j<maxInUse;j++){  cmp = 0;  for(int k= i+1;k<refStrLen;k++){  if(refString[k]!=opt[j]){  cmp++;  }else{  break;  }  }  if(cmp>max){  max = cmp;  victim = j;  }  }  // swap the victim and the vpn  opt[victim] = ref;  }  }  return ans;  } |   流程如下：  1.首先初始化OPT帧数组，用户进程的所有引用都会被安排到这个数组中。长度为5。  2.一开始OPT数组中没有实际引用，-1。  3.遍历之前记录的引用串，对于每一个引用  a.检查是否已经在帧数组中  i.是，则不会引发缺页中断。  ii.否，继续。  b.引发缺页中断，无论是直接装载还是内页置换，缺页次数增一  c.如果当前帧数组内的引用数量还没有到达最大值，则直接装载，并回到3  d.如果当前帧数组内的引用数量到达最大值，则需要内页替换  i.对于帧数组内的每个引用，向后遍历，每经过一个元素给cmp变量增一  ii.如果最后cmp变量的值大于max变量，将max变量的值置为cmp变量的值，并选中当前元素作为被置换的页（这其实就是一个很简单的找最大值的算法）  e.置换当前选择的被置换的页  这样我们就实现了OPT算法的模拟。  在停机前，我打印了当前用户进程OPT算法下的缺页次数：   |  | | --- | | void  Interrupt::Halt()  {  printf("Machine halting!\n\n");  stats->Print();  #ifdef OPT  printf("OPT Page faults: %d\n",currentThread->space->calOPTPf());  #endif  Cleanup(); // Never returns.  } |   来和示例程序对一下答案，我的：  Untitled 19  示例程序的：  Untitled 14  正确，测试通过。  至此我们完成了Lab7。 | | | |
| 结论分析与体会：  在完成Lab7的过程中我遇到了一个很有意思的BUG，分享如下： 系统调用对虚存管理的适配 为了测试我实现的虚存管理是否正确，我选择了Lab6中的几个系统调用（有原本就有的，也有我自己实现的）来测试虚存管理是否支持这些系统调用（比如不覆盖原地址空间的Exec，类似于Unix/Linux下的覆盖原地址空间的Exec，Fork，PrintInt，Halt等）。  在这个过程中，我发现了一个BUG，当时卡了整整一天，后续我通过DEBUG日志信息，结合源码，才终于想明白这个过程。 报BUG：系统调用Exec/UExec无限调用PrintInt 如题，当我们执行exec.noff和UExec.noff时，会出现在系统调用Exec/UExec后在控制台无限循环输出0。如下图。  Untitled 定位BUG 在解决这个BUG时，用时最长的就是定位BUG。这里我把我定位BUG的方式和过程记录在这里。  **1.2.1. 0是哪里输出的？**  通过1.1小节标题，你会很容易知道这是PrintInt系统调用打印的。但当时我开发时，Nachos系统中已经有了各种各样的日志信息。我并不知道这是谁打印的。因此首要问题就是定位到具体打印0的代码。  观察到这行打印的格式是:   |  | | --- | | "%d/n" |   通过使用IDE提供的Ctrl+Shift+F的全局搜索功能，搜索”%d\n”，排除有其他输出信息的代码。只剩下一个，即PrintInt系统调用。  **1.2.2. 在PrintInt之前**  根据我DEBUG的经验，通常情况下当某一步出现问题时，很有可能不是当前这一步的代码有BUG，而是很早之前的代码出现了问题。  放到这个例子里，很显然PrintInt系统调用只是简单地使用了printf标准库函数。这是不可能出现问题的，问题出在了这个调用之前。  因此我尝试打印了这个BUG之前的日志信息。可是日志信息实在太多了！经常出现下面的日志信息覆盖上面的情况。  因此这里我使用Linux下控制台的一个命令，它可以帮助我们每次只看一段（一个屏幕大小）的日志信息，当你想要看之后的日志信息，每按以此Enter键，就多打印一行:   |  | | --- | | |more  ./nachos -x ../test/exec.noff -d m|more |   Untitled  **1.2.3. PC指针值错误**  通过-d m运行选项，我打印了包括但不仅限于汇编指令、系统异常、内核线程等日志信息。  之后我发现，在系统调用Exec之后，PC指针的值并没有从0x0开始，而是从0x4开始。如下图。  **Untitled 2**  这很明显是错误的，因为Exec系统调用后，我们会新起一个内核线程，然后让它承载这个用户程序，为它分配地址空间，最后初始化寄存器，然后把当前用户程序的页表作为Nachos虚拟机的页表，这样该用户程序就能开始执行了。  也就是说，Exec会调用该函数：   |  | | --- | | void  AddrSpace::InitRegisters()  {  int i;  for (i = 0; i < NumTotalRegs; i++)  machine->WriteRegister(i, 0);  // Initial program counter -- must be location of "Start"  machine->WriteRegister(PCReg, 0);  // Need to also tell MIPS where next instruction is, because  // of branch delay possibility  machine->WriteRegister(NextPCReg, 4);  // Set the stack register to the end of the address space, where we  // allocated the stack; but subtract off a bit, to make sure we don't  // accidentally reference off the end!  machine->WriteRegister(StackReg, numPages \* PageSize - 16);  DEBUG('a', "Initializing stack register to %d\n", numPages \* PageSize - 16);  } |   也就是说PC寄存器的值一定是0。但刚刚截图中放出来的却是4。现在我们仅仅暂时记下这个问题，之后我会来解释发生这个错误的原因。  **1.2.4. 新内核线程没有缺页中断**  错误不仅是如此，还记得之前我提到过，由于Pure Demand Paging，我们的每个用户程序的执行一定是由以此缺页中断开始的吗？  在处理缺页中断的函数中，我打印了一次页表，用来显示某次缺页中断处理后页表的状态。   |  | | --- | | // exception.cc :: loadPage  void loadPage(int vAddr){  AddrSpace\* cas = currentThread->space;  //...  cas->Print();  } |   也就是说，除了分配地址空间时打印的页表，其他每看到一次页表，就代表了系统处理了一次缺页中断。但在这次BUG中，日志信息中虽然报了缺页异常，但根本没有打印页表！就跳到了At PC=…，继续执行了。  Untitled 3  其实报了缺页中断，就代表缺页中断将会被系统处理。但是没打印页表，说明系统压根没处理完缺页中断。这是什么原因？  为了定位到底缺页中断的处理函数执行到哪里就不再继续往下执行了，我在代码中加入了一些日志信息。   |  | | --- | | // execption.cc :: loadPage  DEBUG('u',"phys page: %d allocated to cur addr space: %d\n",physPage,cas->getAsId());  // swap.cc :: vm2Mem  DEBUG('v',"the page caused page fault is : %d,target vmem page number is : %d\n",logPage,tPage);  // exception.cc :: loadPage  printf("Demand page %d in(frame %d)\n",logPage,physPage); |   可以发现的是，当我想把虚存中的所需的页装入内存中时，也就是vm2Mem函数执行前，以上代码的第二行是被输出出来的。但是之后第三行代码却没有输出。这说明程序没有执行到第三行代码。  Untitled 4  因此我认为vm2Mem函数出现了问题，这里我们再回顾一下vm2Mem：   |  | | --- | | // Fetch data in vm space to main mem  int vm2Mem(int vAddr,TranslationEntry\* vmpt,int desPos){  // step 1: get the page no. of vAddr in the swap space  int logPage = vAddr/PageSize; // log is short for logical  int tPage = vmpt[logPage].vMemPage; // t is short for target  // step 2: Fetch the page in the swap space  char buf[PageSize];  DEBUG('v',"the page caused page fault is : %d,target vmem page number is : %d\n",logPage,tPage);  vmFetch(tPage,buf);  // step 3: rewrite the data in the buffer back to the physical main mem  physMemCopy(buf,machine->mainMemory,PageSize,0,desPos);  // return the logical page of vAddr  return logPage;  } |   这里既然DEBUG日志打印了，那么即将进入vmFetch函数。而physMemCopy函数只是简单地把buffer里的内容复制到Nachos虚拟机内存中，这是不可能出问题的。再之后就return了。  因此我进一步把问题定位到vmFetch中，为了查看vmFetch函数执行情况，我在该函数开头和结尾处都加入了打印信息。   |  | | --- | | void vmFetch(int secNo,char\* data){  printf("executing here1\n");  swap->ReadSector(secNo,data);  printf("executing here2\n");  } |   再次运行，我发现executing here1被打印了出来，但是executing here2没有。  Untitled 5  也就是说问题进一步被定位到SynchDisk类的ReadSector函数。其实到了这里，我已经隐约感觉到真正的问题并没有出在这里了，因为ReadSector是官方封装的比较底层的函数，这基本上是不会出错的。  为了查看ReadSector函数的执行情况，我在获取互斥锁、信号量等待磁盘中断、磁盘中断回调、释放互斥锁处都加入了打印信息。   |  | | --- | | void  SynchDisk::ReadSector(int sectorNumber, char\* data)  {  DEBUG('v',"rlock P\n");  lock->Acquire(); // only one disk I/O at a time  disk->ReadRequest(sectorNumber, data);  DEBUG('v',"rs P\n");  semaphore->P(); // wait for interrupt  lock->Release();  DEBUG('v',"rlock V\n");  }  void  SynchDisk::RequestDone()  {  DEBUG('v',"V awake\n");  semaphore->V();  } |   再次运行，我发现新内核线程执行发生缺页中断时，在ReadSector时，只打印了rlock P和rs P，但没有V awak和rlock V。  Untitled 6  至此，我终于想明白了BUG出现的本质原因。如果你还没有想明白，请看下文的分析。 解决BUG 我们先来梳理一下1.2节中列举的各步骤，把程序执行的流程复现一遍。  **1.3.1. PC指针值错误的真正原因**  这里由于Exec函数之前都没有什么问题，我就直接从Exec函数分析了。先来回顾一下Exec函数：   |  | | --- | | void  Interrupt::Exec() {  printf("Execute system call of Exec()\n");  int maxFnLen = 50;  char filename[maxFnLen];  /\*\*  \* The start address of filename is in the $4 register  \*/  int fnReg = 4;  int fnAddr = machine->ReadRegister(fnReg);  DEBUG('u',"file name address is: %d\n",fnAddr);  int i;  for(i=0;i<maxFnLen;i++){  machine->ReadMem(fnAddr+i,1,(int\*)&filename[i]);  if(i==maxFnLen-1){  // check if the file name oversize  ASSERT(filename[i]=='\0');  }  if(filename[i]=='\0') // end of file name  break;  }  printf("Exec(%s):\n",filename);  OpenFile\* exe = fileSystem->Open(filename);  if(exe == NULL){  printf("no such file: %s\n",filename);  return;  }  // allocate the page table for exe  AddrSpace\* as = new AddrSpace(exe);  // init the vm registers and restore the page table  as->InitRegisters();  as->RestoreState();  // new kernel thread for executing user thread  Thread\* tfu = new Thread(filename);  // load the as to tfu  tfu->space = as;  int dummy = 0;  tfu->Fork(userFunc,0);  // free exe  delete exe;  // since the nachos Exec() syscall has a ret val  // we should write the as id back to the ret reg  machine->WriteRegister(2,as->getAsId());  // yield currentThread for executing the thread for user  currentThread->Yield();  } |   我们知道Exec调用会新起一个线程。我们把**调用Exec的线程称作t1，把新起的线程称作t2；把t1对应的用户程序地址空间称作a1，把t2对应的用户程序地址空间称作a2**。  我们直接从:   |  | | --- | | // allocate the page table for exe  AddrSpace\* as = new AddrSpace(exe); |   这里开始看，前面的代码很显然没有错误。   1. 首先，为Exec系统调用指定的可执行文件分配地址空间  |  | | --- | | // allocate the page table for exe  AddrSpace\* as = new AddrSpace(exe); |  1. 重置Nachos虚拟机CPU中的寄存器，替换当前页表。（我可以先剧透一下，问题就出在这里，如果你想不明白，暂时记住这一部分很重要就可以了）  |  | | --- | | // init the vm registers and restore the page table  as->InitRegisters();  as->RestoreState(); |  1. 新起一个线程，让它负责该用户程序，并执行machine→Run()（当这个线程得到调度，获得CPU时）。执行完成后把地址空间ID写到2号寄存器中。  |  | | --- | | // new kernel thread for executing user thread  Thread\* tfu = new Thread(filename);  // load the as to tfu  tfu->space = as;  int dummy = 0;  tfu->Fork(userFunc,0);  // free exe  delete exe;  // since the nachos Exec() syscall has a ret val  // we should write the as id back to the ret reg  machine->WriteRegister(2,as->getAsId()); |  1. 让当前线程，也就是t1，主动放弃CPU，把CPU交给t2执行。  |  | | --- | | // yield currentThread for executing the thread for user  currentThread->Yield(); |   问题就出在这个Yield上。我们知道Yield的实现是，找一个ReadyList中的线程，当前是t2。然后Run函数中调用SWITCH切换上下文。   |  | | --- | | void  Thread::Yield ()  {  Thread \*nextThread;  IntStatus oldLevel = interrupt->SetLevel(IntOff);    ASSERT(this == currentThread);    DEBUG('t', "Yielding thread \"%s\"\n", getName());    nextThread = scheduler->FindNextToRun();  if (nextThread != NULL) {  scheduler->ReadyToRun(this);  scheduler->Run(nextThread);  }  (void) interrupt->SetLevel(oldLevel);  }  // scheduler.cc :: Run  void  Scheduler::Run (Thread \*nextThread)  {  Thread \*oldThread = currentThread;    #ifdef USER\_PROGRAM // ignore until running user programs  if (currentThread->space != NULL) { // if this thread is a user program,  currentThread->SaveUserState(); // save the user's CPU registers  currentThread->space->SaveState();  }  #endif    oldThread->CheckOverflow(); // check if the old thread  // had an undetected stack overflow  currentThread = nextThread; // switch to the next thread  currentThread->setStatus(RUNNING); // nextThread is now running    DEBUG('t', "Switching from thread \"%s\" to thread \"%s\"\n",  oldThread->getName(), nextThread->getName());    // This is a machine-dependent assembly language routine defined  // in switch.s. You may have to think  // a bit to figure out what happens after this, both from the point  // of view of the thread and from the perspective of the "outside world".  SWITCH(oldThread, nextThread);    //DEBUG('t', "Now in thread \"%s\"\n", currentThread->getName());  // If the old thread gave up the processor because it was finishing,  // we need to delete its carcass. Note we cannot delete the thread  // before now (for example, in Thread::Finish()), because up to this  // point, we were still running on the old thread's stack!  if (threadToBeDestroyed != NULL) {  delete threadToBeDestroyed;  threadToBeDestroyed = NULL;  }    #ifdef USER\_PROGRAM  if (currentThread->space != NULL) { // if there is an address space  currentThread->RestoreUserState(); // to restore, do it.  currentThread->space->RestoreState();  }  #endif  } |   注意Run函数中的这一段代码：   |  | | --- | | #ifdef USER\_PROGRAM // ignore until running user programs  if (currentThread->space != NULL) { // if this thread is a user program,  currentThread->SaveUserState(); // save the user's CPU registers  currentThread->space->SaveState();  }  #endif |   还记得SaveUserState吗？我们来回顾一下：   |  | | --- | | void  Thread::SaveUserState()  {  for (int i = 0; i < NumTotalRegs; i++){  userRegisters[i] = machine->ReadRegister(i);  }  if(DebugIsEnabled('t')){  printf("Save PC State: %d\n",userRegisters[PCReg]);  }  } |   它把当前CPU寄存器里的值全部保存到当前线程的用户寄存器里了。这样下次切换回该线程时，我们就可以用户寄存器中取出之前执行的值，继续执行下去。  但是，还记得之前第2步时，我们把CPU寄存器初始化了吗？   |  | | --- | | // init the vm registers and restore the page table  as->InitRegisters();  as->RestoreState(); |   而InitRegisters把PC的值初始化为0了。也就是说我们并没有把t1此刻应有的PC值保存到用户寄存器（这个值应该是68，如下图），而是把应该初始化给t2的0保存到了t1的用户寄存器！  我们接着往下看，马上就知道为什么PC从0x4开始了。  正如上文所说的，由于是Pure Demand Paging，t2一开始发现自己所需的页并不在内存中。因此引发了缺页中断。随后它就要调用ReadSectors去读取虚存中的数据。  但是，SynchDisk的semaphore实例的初始值为0：   |  | | --- | | SynchDisk::SynchDisk(const char\* name)  {  semaphore = new Semaphore("synch disk", 0); |   而Semaphore的P操作会在值为0时将当前线程阻塞在一个等待队列中：   |  | | --- | | void  Semaphore::P()  {  IntStatus oldLevel = interrupt->SetLevel(IntOff); // disable interrupts    while (value == 0) { // semaphore not available  queue->Append((void \*)currentThread); // so go to sleep  currentThread->Sleep();  }  value--; // semaphore available,  // consume its value    (void) interrupt->SetLevel(oldLevel); // re-enable interrupts  } |   这一部分其实早在Lab4就分析过了，这里只是回顾一下，不多阐释。  因此，t2会被阻塞在队列queue中，随后它会Sleep，而我们知道Sleep函数会去寻找下一个ready list中的线程，这里就是t1。   |  | | --- | | void  Thread::Sleep ()  {  Thread \*nextThread;    ASSERT(this == currentThread);  ASSERT(interrupt->getLevel() == IntOff);    DEBUG('t', "Sleeping thread \"%s\"\n", getName());  status = BLOCKED;  while ((nextThread = scheduler->FindNextToRun()) == NULL)  interrupt->Idle(); // no one to run, wait for an interrupt    scheduler->Run(nextThread); // returns when we've been signalled  } |   其实实际意义就是，t2由于缺页中断，需要等待读磁盘，在这段时间里os调度t1进行执行。这下真相终于水落石出了，原来PC=0x4是t1执行时打印的。那么为什么是0x4呢？  首先，我们在线程切换时，错误地保存了用户寄存器的值，这个我上面已经分析过了，保存了0进去。  而t1在执行用户程序时，由于系统调用异常，切换到了t2，因此切换回来时，恰好就是系统调用结束的位置：   |  | | --- | | // mipssim.cc :: OneInstruction  void  Machine::OneInstruction(Instruction \*instr)  {  //...  case OP\_SYSCALL:  RaiseException(SyscallException, 0);  return;  //...  // Advance program counters.  registers[PrevPCReg] = registers[PCReg]; // for debugging, in case we  // are jumping into lala-land  registers[PCReg] = registers[NextPCReg];  registers[NextPCReg] = pcAfter; |   因此下一步就是把PC指针往前挪4个字节，这样下一次打印的时候，PC=0x4而不是0x0了。  示意图如下：  Untitled 7  **1.3.2. 修复该BUG**  正如上文所说，问题并没有出在读磁盘上，而是出现在了错误地保存已经初始化过的寄存器到原有线程（t1）中的用户寄存器。因此我们只需要修改初始化的时机，让切换线程时的Nachos虚拟机的寄存器仍是原有线程的数据即可。  这里我就只展示如何修复Exec系统调用了，UExec系统调用同理：   |  | | --- | | // dummy arg,useless  void userFunc(\_int dummy){  /\*\*  \* Lab7:vmem  \* patch for lab6  \* when we yield the current thread  \* it save the user registers before context switch  \* Also,we init the regs by using current thread's addr space  \* thus,we save the wrong state to cur thread's addr space  \* so it's necessary to init the regs by using new thread's addr space  \* after the context switching  \*/  AddrSpace\* as = currentThread->space;  // init the vm registers and restore the page table  as->InitRegisters();  as->RestoreState();  // run the user prog  machine->Run();  // never returns  ASSERT(FALSE);  } |   只需要把初始化寄存器和替换页表的部分放到t2线程的func里即可。我们来想一下为什么这样就可以解决上述BUG。首先我们在Exec函数中没有错误地把为t2初始化的寄存器保存到t1的用户空间中；其次当t2执行时，我们在它执行的func中为它初始化了用户寄存器，因此t2的执行也是正确的。  至此我们修复了这个BUG。 2. 后记 上述的BUG曾使我周六整整一天都坐在电脑前，不停地向代码中加入各种日志信息，然后观察打印结果，再抓耳挠腮地苦思冥想一阵子。直到那周周日中午才解决这个问题。后面Lab7的其他内容就一路高歌猛进了，虽然也遇到一些问题，但很快都得到解决。  Lab7是整个os课设的最后一个实验，因此我想在这里写一点对于整个课设的心得体会。  最首要的一点，“纸上得来终觉浅，绝知此事要躬行”。道理总是知易行难的，举个例子，上学期os理论课PPT里有一句话：“操作系统是中断驱动的。”当时不求甚解，为了应付考试，死记硬背了下来。其实到了这学期，这句话已经几乎遗忘了。  但就是在处理Lab7的这个BUG时，历经各种困难，我来到了SynchDisk的ReadSector函数面前。看着lock→acquire,semaphore→P，回想起Lab2和Lab3中线程调度、切换的种种，“操作系统是中断驱动的”这句话几乎是在一瞬间就打到了我的脑海中，直到此时我才更进一步了解了“调度”这个词的含义：os并不是主动掌握着计算机软件或硬件的生杀大权的独裁者，而是一个协调者，更准确地说，“调度者”。只有当某个事件/中断发生时，计算机才会请它出山，摆平一切。TimerInt请os找出下一个将要执行的线程并切换上下文；DiskInt请os唤醒正在等待读/写磁盘的线程，让它得以从waiting queue回到ready list中……中断发生时，才轮到os工作。  如果在上学期问我，为什么操作系统是中断驱动的？我可能会拿出教科书上的那一套八股文敷衍一番。如果你再细问，可能我就只能无奈地说：”书上是这么说的。“但如果这学期你问我这个问题，我就会拿这次课设的内容举例，列举一大堆证据来佐证这个观点。这是我坐在电脑前一整天的成果。  之后的第二点想法是，”人的进步是由错误驱动的“。老师之前已经把答案发出来了，对着答案写，我觉得堪堪10小时内就可以结束这个课设，这样能省出来大量的时间复习期末考试。但我觉得这样做没有任何意义。别人的代码再优雅，用了再厉害的算法、再复杂的数据结构、再高明的设计模式，终究不是自己的，也许自己写的代码，连编译都过不去，更不要提逻辑是否正确了。  ”不撞南墙不回头，不到黄河不死心；不当家不知柴米油盐贵，不踩坑不知道正路好走。“自己完整地写一遍代码，才能把握住细节，心里才踏实。只看结果，是花了整整一天，仅改了两行代码，甚至代码内容都没变，就只是换了个位置而已；但看过程，是积累了DEBUG的经验，掌握了DEBUG的工具，更进一步地理解了系统调用→线程切换→缺页中断→读磁盘延迟→线程切换的过程。虽然为了试错我花费了大量时间，但我觉得这是值得的。  Nachos不止于此，其实官方还给出了很多其它的模块等待我们去发掘，已经做过Lab的模块，也可以自己额外添加其他机制，尝试向现代操作系统靠近。今后的时间里，我会尝试把Nachos继续写下去。 | | | |