山东大学 软件 学院

**操作系统课程设计** 实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202100300340 | 姓名：黄幸兒 | | 班级：工软 21 |
| 学号：202100300063 | 姓名：李彦浩 | | 班级：工软 21 |
| 实验编号：Lab6 | | | |
| 实验题目：系统调用与多道用户程序 | | | |
| 实验学时：5 | | 实验日期：2023.11.17 | |
| 实验目的：  1. 扩展现有的class AddrSpace的实现，使得Nachos可以实现多道用户程序。按照实验指导书中的方法，完成class AddrSpace中的Print函数。实现Nachos系统调用：Exec()，一个用户程序启动另一个用户程序。注意本实验要求实现的Exec()系统调用，是在另一个地址空间运行指定的另一个用户程序，新程序并没有覆盖调用者的地址空间。这与Unix/Linux的系统调用exec()不同。  2. 在Nachos中增加并实现一个新的系统调用：PrintInt()，在用户程序中打印一个整数值。  3. 在实现了多道用户程序的基础上，若要求在Nachos中实现与Unix/Linux 的fork()/exec()功能类似的Nachos系统调用Fork()/Exec()，及写时复制 (copy-on-write) 机制，请给出在Nachos中实现的具体方法(实现时假定有足够的物理内存，无需页面置换。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关键代码片段、数据结构、对象等说明)。 | | | |
| 硬件环境：  联想Thinkpad笔记本  Intel Core i5-8250 CPU 8核  8GB内存  1907GB 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. 源码阅读 3](#_Toc24133)  [1.1. Nachos虚拟机 3](#_Toc19978)  [1.1.1. 从一个进程到虚拟机 3](#_Toc27743)  [1.1.2. Nachos虚拟机模拟实现 3](#_Toc27303)  [1.1.2.1. 模拟一台虚拟机的内存空间 3](#_Toc15719)  [1.1.2.2. 模拟一台虚拟机的CPU 4](#_Toc4107)  [1.2. 页式内存管理 6](#_Toc23649)  [1.2.1. 模拟MMU 6](#_Toc31688)  [1.3. 从内核线程到用户进程 9](#_Toc23755)  [1.3.1. Nachos用户进程的地址空间 9](#_Toc22221)  [1.3.1.1. Nachos专属可执行文件 9](#_Toc16788)  [1.3.1.2. 用户进程的地址空间 10](#_Toc9988)  [1.3.2. 从内核线程到用户进程 15](#_Toc13525)  [1.3.2.1. 为内核线程分配用户代码 15](#_Toc688)  [1.3.3. 通过命令行启动用户程序 16](#_Toc24102)  [1.3.4. 如何编译用户程序 18](#_Toc15329)  [1.3.4.1. 调用用户程序的main函数 18](#_Toc21703)  [1.3.4.2. 系统调用的存根 19](#_Toc11394)  [1.3.4.3. 如何进行系统调用 20](#_Toc21499)  [2. Nachos系统调用：Exec(char\* filename) 22](#_Toc20420)  [2.1. 多道用户程序的地址空间 22](#_Toc19263)  [2.1.1. 原有的Nachos地址空间分配策略 22](#_Toc31733)  [2.1.2. 位图分配空闲内存页 24](#_Toc29006)  [2.1.3. 用户程序的物理地址 25](#_Toc26631)  [2.2. 系统调用的参数是怎么传递的？ 27](#_Toc30722)  [2.2.1. 参数指针传递到参数寄存器 27](#_Toc16976)  [2.2.2. 参数值传递到Nachos虚拟机内存 28](#_Toc12442)  [2.3. 比较重要的Nachos-MIPS汇编指令 28](#_Toc2635)  [2.3.1. 汇编指令解析 28](#_Toc10999)  [2.3.2. 空间较大的参数传入内存的时机 31](#_Toc11620)  [2.4. StartProcess入口与嵌套Exec调用 35](#_Toc28603)  [2.4.1. 从Exec系统调用的视角看StartProcess函数 36](#_Toc10000)  [2.4.2. 嵌套使用Exec系统调用 37](#_Toc13431)  [2.5. 系统调用总体流程一览 38](#_Toc9922)  [3. Nachos系统调用：PrintInt(int num) 41](#_Toc8784)  [3.1. 增加系统调用代（码）号和定义 41](#_Toc4739)  [3.2. 增加存根 41](#_Toc19140)  [3.3. Tips:PC自增 41](#_Toc13099)  [3.4. 实现 42](#_Toc869)  [4. 复现Linux/Unix下的Fork()/Exec()/Copy-On-Write 43](#_Toc32662)  [4.1. 复现Fork(void (\*func)()) 43](#_Toc26838)  [4.1.1. 修改EH驱动表 43](#_Toc26184)  [4.1.2. 实现Fork逻辑 44](#_Toc2639)  [4.1.3. 编写测试程序 45](#_Toc12630)  [4.1.4. 一点小的疑问：为什么能正确执行 46](#_Toc2654)  [4.2. 复现Unix下Exec(char\* name) 48](#_Toc16027)  [4.2.1. 增加系统调用号 48](#_Toc9942)  [4.2.2. 增加存根 48](#_Toc22138)  [4.2.3. 修改EH驱动表 49](#_Toc26885)  [4.2.4. 实现 49](#_Toc32182)  [4.2.5. 编写测试程序 50](#_Toc2202)  [4.3. 复现Copy-On-Write机制 51](#_Toc17207)  [4.3.1. 什么是COW？ 51](#_Toc7924)  [4.3.2. 一点理论上的疑问 53](#_Toc20651)  [4.3.3. 标识父线程的物理页 53](#_Toc30127)  [4.3.4. MMU映射检查 54](#_Toc32092) 源码阅读Nachos虚拟机从一个进程到虚拟机 之前的所有Lab完成的事情，都并没有让Nachos看起来“像”一个操作系统。具体来说就是我们甚至要自己编写测试程序，然后把测试的程序编译到内核中。而测试程序本身其实不应该是内核的一部分，这应该是用户态下的部分。  而从Lab6开始，我们将要实现执行Nachos操作系统下专属的可执行文件（Nachos Object File Format,noff)，并可以通过命令行的方式提供系统调用的接口等。这就让Nachos变得更像一台虚拟机（即便我们知道它仍然是宿主机上的进程）。 Nachos虚拟机模拟实现模拟一台虚拟机的内存空间 既然可以执行自己专属的可执行文件，就要有自己专属的内存空间来装入这些程序。因此首先要实现内存管理机制。  Nachos采用页式内存分配方式，为了加快查询速度，外挂快表(TLB），这一部分的数据结构定义在Nachos虚拟机的模拟类Machine中：   |  | | --- | | // machine.h  char \*mainMemory; // physical memory to store user program,code and data, while executing  TranslationEntry \*tlb;  TranslationEntry \*pageTable;  unsigned int pageTableSize; |   而内存大小也是固定的，这一点可以从初始化中看出来：   |  | | --- | | #define SectorSize 128  #define PageSize SectorSize  #define NumPhysPages 32  #define MemorySize (NumPhysPages \* PageSize)  Machine::Machine(bool debug)  {  //...  mainMemory = new char[MemorySize]; |   也就是32\*128 = 2^12B = 4KB。 模拟一台虚拟机的CPU 在Lab1的时候分析过，Nachos采用软件模拟的方式实现了MIPS指令集中的大部分指令。这一块就不再赘述了。现在主要讲CPU是如何执行指令的。  程序要执行，首先离不开PC（Program Counter，程序计数器），既然Nachos用软件模拟CPU，那么就需要一块宿主机内存空间来装载这些寄存器的值，这部分代码在：   |  | | --- | | // machine.h  #define StackReg 29 // User's stack pointer  #define RetAddrReg 31 // Holds return address for procedure calls  #define NumGPRegs 32 // 32 general purpose registers on MIPS  #define HiReg 32 // Double register to hold multiply result  #define LoReg 33  #define PCReg 34 // Current program counter  #define NextPCReg 35 // Next program counter (for branch delay)  #define PrevPCReg 36 // Previous program counter (for debugging)  #define LoadReg 37 // The register target of a delayed load.  #define LoadValueReg 38 // The value to be loaded by a delayed load.  #define BadVAddrReg 39 // The failing virtual address on an exception  #define NumTotalRegs 40  int registers[NumTotalRegs]; |   不仅定义了CPU内置寄存器的数量，还规定了其中的几个特殊寄存器，比如栈底指针、乘积高位寄存器、乘积低位寄存器、程序计数器等。  这里要注意的是，Nachos不但有PC，还有PC指针的下一个位置的寄存器以及PC指针上一个指针的寄存器。前者用来标识一个分支（如果当前处在一个分支的话）后程序计数器的位置，后者用来DEBUG。  接下来是CPU如何执行一条指令，当然执行的前提是指令已经装入了内存，我们这里先不管指令是怎样装入内存的，就假设内存中已经装载了指令。   |  | | --- | | // mipssim.cc  void  Machine::Run()  {  Instruction \*instr = new Instruction; // storage for decoded instruction  if(DebugIsEnabled('m'))  printf("Starting thread \"%s\" at time %d\n",  currentThread->getName(), stats->totalTicks);  interrupt->setStatus(UserMode);  for (;;) {  OneInstruction(instr);  interrupt->OneTick();  if (singleStep && (runUntilTime <= stats->totalTicks))  Debugger();  }  }  void  Machine::OneInstruction(Instruction \*instr)  {  int raw;  int nextLoadReg = 0;  int nextLoadValue = 0; // record delayed load operation, to apply  // in the future  // Fetch instruction  if (!machine->ReadMem(registers[PCReg], 4, &raw))  return; // exception occurred  instr->value = raw;  instr->Decode();  //...  // Advance program counters.  registers[PrevPCReg] = registers[PCReg]; // for debugging, in case we  // are jumping into lala-land  registers[PCReg] = registers[NextPCReg];  registers[NextPCReg] = pcAfter;  } |   这里OneInstruction就是之前Lab1分析过指令模拟的部分，即用一个巨大的驱动表和一段非常长的switch代码块模拟的MIPS指令，这里就不多赘述了。主要是执行用户程序的流程：   1. 执行用户程序当然要设置当前模式是用户态。 2. 死循环，执行到停机为止。 3. 首先是取指周期。对于每一条指令（OneInstruction），首先先要从内存中读取指令出来，放到IR（Instruction Register，指令寄存器）中，由于我们是软件模拟，所以用一个指针指向内存数组mainMemory中索引为CPU数组中索引为PCReg（也即PC寄存器）的值的元素，相当于IR了。 4. 然后是执行周期。经过一个解码器（要让CPU知道执行什么类型的指令，具体是哪一条），这部分不是这里的重点，先略去。 5. 执行完成后，PC向前推动，别忘了我们有三个PC指针，都要前推。这样准备进入下一个取指周期为止。   如下图所示：  **Untitled** 页式内存管理模拟MMU 无论怎样的内存管理机制都有MMU，我们现在要模拟的是页式内存管理的MMU：   |  | | --- | | ExceptionType  Machine::Translate(int virtAddr, int\* physAddr, int size, bool writing)  {  int i;  unsigned int vpn, offset;  TranslationEntry \*entry;  unsigned int pageFrame;  DEBUG('a', "\tTranslate 0x%x, %s: ", virtAddr, writing ? "write" : "read");  // check for alignment errors  if (((size == 4) && (virtAddr & 0x3)) || ((size == 2) && (virtAddr & 0x1))){  DEBUG('a', "alignment problem at %d, size %d!\n", virtAddr, size);  return AddressErrorException;  }    // we must have either a TLB or a page table, but not both!  ASSERT(tlb == NULL || pageTable == NULL);  ASSERT(tlb != NULL || pageTable != NULL);  // calculate the virtual page number, and offset within the page,  // from the virtual address  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",  virtAddr, pageTableSize);  return AddressErrorException;  } else if (!pageTable[vpn].valid) {  DEBUG('a', "virtual page # %d too large for page table size %d!\n",  virtAddr, pageTableSize);  return PageFaultException;  }  entry = &pageTable[vpn];  } else {  for (entry = NULL, i = 0; i < TLBSize; i++)  if (tlb[i].valid && ((unsigned int)tlb[i].virtualPage == vpn)) {  entry = &tlb[i]; // FOUND!  break;  }  if (entry == NULL) { // not found  DEBUG('a', "\*\*\* no valid TLB entry found for this virtual page!\n");  return PageFaultException; // really, this is a TLB fault,  // the page may be in memory,  // but not in the TLB  }  }  if (entry->readOnly && writing) { // trying to write to a read-only page  DEBUG('a', "%d mapped read-only at %d in TLB!\n", virtAddr, i);  return ReadOnlyException;  }  pageFrame = entry->physicalPage;  // if the pageFrame is too big, there is something really wrong!  // An invalid translation was loaded into the page table or TLB.  if (pageFrame >= NumPhysPages) {  DEBUG('a', "\*\*\* frame %d > %d!\n", pageFrame, NumPhysPages);  return BusErrorException;  }  entry->use = TRUE; // set the use, dirty bits  if (writing)  entry->dirty = TRUE;  \*physAddr = pageFrame \* PageSize + offset;  ASSERT((\*physAddr >= 0) && ((\*physAddr + size) <= MemorySize));  DEBUG('a', "phys addr = 0x%x\n", \*physAddr);  return NoException;  } |   流程如下：   1. 内存管理中有一个很重要的概念叫做对齐。对齐是指将数据存储在内存中时，按照一定的规则让数据排列在规定的地址上，以提高数据访问的效率和速度。这些规则包括每个成员变量按照其自身所占用的字节数对齐，如果结构体的大小不是某个字节对齐周期的倍数，则自动填充字节使得结构体大小为对齐周期的倍数，结构体的首地址要和对齐周期的倍数相同。对齐周期指的是数据类型的大小，例如int类型大小为4字节，则以4字节为对齐周期进行内存对齐。   这里首先检查了对齐错误，不是重点，先略过。   1. 在Nachos源码中，快表和页表只能选一个使用，不能两个都不用或两个都用。这就是那两条断言的作用。  |  | | --- | | // we must have either a TLB or a page table, but not both!  ASSERT(tlb == NULL || pageTable == NULL);  ASSERT(tlb != NULL || pageTable != NULL); |  1. 首先根据入参的虚拟地址（以字节为单位）计算虚拟页号是多少。然后取余计算页内偏移量是多少。  |  | | --- | | // calculate the virtual page number, and offset within the page,  // from the virtual address  vpn = (unsigned) virtAddr / PageSize;  offset = (unsigned) virtAddr % PageSize; |  1. 然后要分两种情况了： 2. 使用页表：由于页表是顺序存储页号的，所以如果之前计算出来页号为x，那么直接访问页表中索引为x的表项即可。  |  | | --- | | if (tlb == NULL) { // => page table => vpn is index into table  // ... exceptions check  entry = &pageTable[vpn]; |  1. 使用快表：遍历快表，查询对应的虚拟页号对应的物理帧号(如果没找到返回Missing的PageFaultException  |  | | --- | | else {  for (entry = NULL, i = 0; i < TLBSize; i++)  if (tlb[i].valid && ((unsigned int)tlb[i].virtualPage == vpn)) {  entry = &tlb[i]; // FOUND!  break;  } |  1. 获取到这个表项中的物理帧号。声明当前这个帧已经被使用。 2. 计算物理地址并返回。   后面我们需要实现既可以使用TLB又能使用页表的内存管理机制，因此这个函数中的断言需要改变。另外，还需要实现多道用户程序的装载，因此之前分配地址空间的函数的分配方式也需要修改。 从内核线程到用户进程Nachos用户进程的地址空间Nachos专属可执行文件 其实只是把C语言编译链接出来的可执行文件通过官方给出的轮子coff2noff 和coff2float 转换成适配于Nachos软件模拟的CPU和指令集的可执行文件。但是在结构上还是一般的可执行文件的结构：   |  | | --- | | #define NOFFMAGIC 0xbadfad /\* magic number denoting Nachos  \* object code file  \*/  typedef struct segment {  int virtualAddr; /\* location of segment in virt addr space \*/  int inFileAddr; /\* location of segment in this file \*/  int size; /\* size of segment \*/  } Segment;  typedef struct noffHeader {  int noffMagic; /\* should be NOFFMAGIC \*/  Segment code; /\* executable code segment \*/  Segment initData; /\* initialized data segment \*/  Segment uninitData; /\* uninitialized data segment --  \* should be zero'ed before use  \*/  } NoffHeader; |   首先是一个特殊的文件头部编码，用来标识这是一个Nachos专属的可执行文件。然后是三个大段：   1. code：代码段 2. initData：初始化的数据段 3. uninitData：未初始化的数据段   每个大段都有几个属性：   1. virtualAddr：内存中的虚拟帧号，用来在装载到内存后找到这个段。 2. inFileAddr：在noff文件中的偏移量。 3. size：段的大小。  用户进程的地址空间 每个用户进程都分配得到了内存中的一部分空间。因此需要一个数据结构来标识该进程的内存空间的分配结果，也就是页表。在Nachos中，一个线程/用户进程对应一个页表，一个页表也对应一个线程/用户进程。  而线程的集合等于以下集合之并集：   1. currentThread指针指向的线程 2. readyList中的线程 3. 各同步机制中（如Semaphore,lock,Condition variable)中的等待队列中的线程   因此Nachos的页表并没有存放在自己的虚拟机内存（注意是虚拟机内存不是虚拟内存）中，因为上述数据结构都存放在宿主机的内存中。这是Nachos内存管理机制和一般操作系统不同之处。   |  | | --- | | class AddrSpace {  public:  AddrSpace(OpenFile \*executable); // Create an address space,  // initializing it with the program  // stored in the file "executable"  ~AddrSpace(); // De-allocate an address space  void InitRegisters(); // Initialize user-level CPU registers,  // before jumping to user code  void SaveState(); // Save/restore address space-specific  void RestoreState(); // info on a context switch  private:  TranslationEntry \*pageTable; // Assume linear page table translation  // for now!  unsigned int numPages; // Number of pages in the virtual  // address space  }; |   对于地址空间实现来说，要建立一个页表并保存其页数（大小）。装载可执行文件到内存时，使用这个页表记录可执行文件各数据段分配到的帧。   |  | | --- | | AddrSpace::AddrSpace(OpenFile \*executable)  {  NoffHeader noffH;  unsigned int i, size;  executable->ReadAt((char \*)&noffH, sizeof(noffH), 0);  if ((noffH.noffMagic != NOFFMAGIC) &&  (WordToHost(noffH.noffMagic) == NOFFMAGIC))  SwapHeader(&noffH);  ASSERT(noffH.noffMagic == NOFFMAGIC);  // how big is address space?  size = noffH.code.size + noffH.initData.size + noffH.uninitData.size  + UserStackSize; // we need to increase the size  // to leave room for the stack  numPages = divRoundUp(size, PageSize);  size = numPages \* PageSize;  ASSERT(numPages <= NumPhysPages); // check we're not trying  // to run anything too big --  // at least until we have  // virtual memory  DEBUG('a', "Initializing address space, num pages %d, size %d\n",  numPages, size);  // first, set up the translation  pageTable = new TranslationEntry[numPages];  for (i = 0; i < numPages; i++) {  pageTable[i].virtualPage = i; // for now, virtual page # = phys page #  pageTable[i].physicalPage = i;  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  }    // zero out the entire address space, to zero the uninitialized data segment  // and the stack segment  bzero(machine->mainMemory, size);  // then, copy in the code and data segments into memory  if (noffH.code.size > 0) {  DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",  noffH.code.virtualAddr, noffH.code.size);  executable->ReadAt(&(machine->mainMemory[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(&(machine->mainMemory[noffH.initData.virtualAddr]),  noffH.initData.size, noffH.initData.inFileAddr);  }  }  //translate.h  class TranslationEntry {  public:  int virtualPage; // The page number in virtual memory.  int physicalPage; // The page number in real memory (relative to the  // start of "mainMemory"  bool valid; // If this bit is set, the translation is ignored.  // (In other words, the entry hasn't been initialized.)  bool readOnly; // If this bit is set, the user program is not allowed  // to modify the contents of the page.  bool use; // This bit is set by the hardware every time the  // page is referenced or modified.  bool dirty; // This bit is set by the hardware every time the  // page is modified.  }; |   在AddrSpace构造函数中，将读取指定的可执行文件，然后将它装入内存：   1. 首先Nachos可执行文件有个特定的文件头NoffHeader，里面保存了程序段、数据段、未初始化数据段的具体信息，所以要先把这些信息获取到，就得读取这个文件头。（注意这里的文件头和Lab4和Lab5的文件头不是一个概念，这个文件头不是Nachos文件头，而是Unix文件数据中的文件头）  |  | | --- | | executable->ReadAt((char \*)&noffH, sizeof(noffH), 0); |  1. 后面是不同处理器架构上会采用不同的内存管理机制，有的是大端法，有的是小端法。我们的机器采用小端法，因此进行了转换。这里不是重点，略过。 2. 随后要计算这个文件的总大小，也就是程序段、数据段、未初始化数据段三个段的大小加起来。计算页数。  |  | | --- | | size = noffH.code.size + noffH.initData.size + noffH.uninitData.size  + UserStackSize; // we need to increase the size  // to leave room for the stack  numPages = divRoundUp(size, PageSize);  size = numPages \* PageSize; |  1. 之后是分配页表，在当前实现中，虚拟页号就是真实帧号（pt[i].virtP = physT = i)  |  | | --- | | pageTable = new TranslationEntry[numPages];  for (i = 0; i < numPages; i++) {  pageTable[i].virtualPage = i; // for now, virtual page # = phys page #  pageTable[i].physicalPage = i;  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  } |   其实这是快表的数据结构。因为页表本质上默认顺序存储页号，类似于：（图来自上学期理论课PPT）  Untitled  而快表是一个虚拟页号和物理帧号的哈希表，类似于：  Untitled 1  所以上述的实现更偏向于快表的结构而不是页表，但也能做页表。   1. Nachos源码中没有实现多道用户程序，因为每次分配地址空间的时候，都会清空一次内存，所以如果之前内存中有其他的数据，那么在这次装配时将会清空。清空内存是为了使得未初始化的数据段的初值为0。  |  | | --- | | // zero out the entire address space, to zero the uninitialized data segment  // and the stack segment  bzero(machine->mainMemory, size); |  1. 之后是向内存中写入程序和数据，这里比较简单，不再赘述。  |  | | --- | | // then, copy in the code and data segments into memory  if (noffH.code.size > 0) {  DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",  noffH.code.virtualAddr, noffH.code.size);  executable->ReadAt(&(machine->mainMemory[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(&(machine->mainMemory[noffH.initData.virtualAddr]),  noffH.initData.size, noffH.initData.inFileAddr);  } |  从内核线程到用户进程为内核线程分配用户代码 一个线程要执行一个任务，这个任务在之前的Lab中是我们自己编写了测试程序并指派给一些线程的。这只是为了单元测试，测试线程这个模块的行为是否正确。但测试程序本身不应该被编译到内核代码中。一个内核线程除了执行内核代码中的任务外，只能对接到一个用户进程上，执行用户的代码。  因此，一个线程需要保存用户的代码到内存，并且能够按照页表的指引找到这些代码。这部分数据结构的定义在thread.h下：   |  | | --- | | #ifdef USER\_PROGRAM  // A thread running a user program actually has \*two\* sets of CPU registers --  // one for its state while executing user code, one for its state  // while executing kernel code.  int userRegisters[NumTotalRegs]; // user-level CPU register state  public:  void SaveUserState(); // save user-level register state  void RestoreUserState(); // restore user-level register state  AddrSpace \*space; // User code this thread is running.  #endif |   这里注意，线程是可能发生上下文切换的。因此我们要把当时Nachos虚拟CPU寄存器中的值保存下来。也就是这个userRegisters 数组。另外就是保存用户程序和恢复用户程序到主存，这个也是为了上下文切换服务的。  之后最重要的，页表，也就是用户代码的地址空间。  从逻辑上来说，用户进程的地址空间的注册位置一定是某个线程（区分注册位置和实际位置的区别，注册位置是找到页表的位置，实际位置在Nachos虚拟机内存中）。而线程的注册位置是那个(currentThread+readyList+waitingList)的大集合中，因此结构上如下图所示：  Untitled 2 通过命令行启动用户程序 在main函数中允许我们通过-x选项来执行Nachos专属的可执行文件。随后会调用StartProcess函数，该函数具体实现：   |  | | --- | | void  StartProcess(char \*filename)  {  OpenFile \*executable = fileSystem->Open(filename);  AddrSpace \*space;  if (executable == NULL) {  printf("Unable to open file %s\n", filename);  return;  }  space = new AddrSpace(executable);  currentThread->space = space;  delete executable; // close file  space->InitRegisters(); // set the initial register values  space->RestoreState(); // load page table register  machine->Run(); // jump to the user program  ASSERT(FALSE); // machine->Run never returns;  // the address space exits  // by doing the syscall "exit"  } |   流程如下：   1. 首先使用文件系统打开这个可执行的Nachos文件。 2. 调用之前实现的AddrSpace的构造函数向Nachos虚拟机内存中装入这个可执行文件。 3. 使得当前线程的空间地址（页表）为该可执行文件的空间地址（页表），这样当前线程就承担了执行该可执行文件的责任。如果发生上下文切换，当前线程也有责任保存现场，以便下次切换时继续执行。 4. 初始化用户寄存器  |  | | --- | | 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);  } |   首先先让CPU中的所有用户寄存器清零。注意装入的code段是从内存中0号索引处开始的，所以初始化PC指针为0和下一个PC指针为4。初始化PC指针后，会从用户程序的第一条指令开始执行。最后初始化栈底指针。   1. 将当前虚拟机执行程序时需要用到的页表置换为当前程序的页表。 2. 执行程序。Run函数在之前已经详细分析过。   示意图如下：  Untitled 3 如何编译用户程序调用用户程序的main函数 在Nachos中，所有用户程序在编译后都将在目标文件前连接上一个由start.s生成的start.o的目标模块，相较于将用户程序的main函数作为入口地址，这里把start符号作为启动入口，在这个start函数中调用main函数：   |  | | --- | | .globl \_\_start  .ent\_\_start  \_\_start:  jal main  move $4,$0  jal Exit /\* if we return from main, exit(0) \*/  .end \_\_start |  1. jal main：jump and link main。将move $4,$0这条指令的地址置入$ra(返回地址寄存器中)，然后跳转到用户程序的main函数入口处。执行完main之后将跳转回来。 2. move $4,$0：把用户main函数的返回值置入$4寄存器（一般来说存放在$0寄存器中） 3. jal Exit：下面我们将看到Exit的存根与实现  |  | | --- | | // start.s stub of Exit  .globl Exit  .ent Exit  Exit:  addiu $2,$0,SC\_Exit  syscall  j $31  .end Exit  // syscall.h def of Exit  /\* This user program is done (status = 0 means exited normally). \*/  void Exit(int status);  // sysdep.cc impl of Exit  void  Exit(int exitCode)  {  exit(exitCode);  } |   我们可以看到，Exit函数是要一个参数的，这个参数代表了系统是否正常退出。而系统是否正常退出完全由用户程序中的main函数决定，因为start函数只是调用了main函数。因此第二步中的$4寄存器中存放的就是Exit函数的参数。 系统调用的存根 用户程序可能会调用一些Nachos操作系统开放的系统调用API，这些系统调用一式两份，一份是存根，位于start.s文件中；另一份是定义，位于syscall.h中。至于实现那位置就不确定了。  先来看存根部分：   |  | | --- | | .globl Halt  .ent Halt  Halt:  addiu $2,$0,SC\_Halt  syscall  j $31  .end Halt  .globl Exit  .ent Exit  Exit:  addiu $2,$0,SC\_Exit  syscall  j $31  .end Exit  .globl Exec  .ent Exec  Exec:  addiu $2,$0,SC\_Exec  syscall  j $31  .end Exec |   这里就放几个上来。可以看下是如何调用的。就以Halt为例：   1. 注意$2寄存器是函数返回值的一个寄存器，我们先把SC\_Halt的系统调用标记放到这个寄存器中，随后系统调用结束后把Halt函数的返回值（如果有）放到$2寄存器中。 2. 系统调用，具体调用哪个要看$2中的值。 3. 跳转到返回地址寄存器中对应的地址。 4. 结束系统调用  如何进行系统调用 系统调用通过操作系统的异常和自陷管理来调用的。这一部分是个hard code的驱动表，首先先定义了一个枚举类型：   |  | | --- | | // machine.h  enum ExceptionType { NoException, // Everything ok!  SyscallException, // A program executed a system call.  PageFaultException, // No valid translation found  ReadOnlyException, // Write attempted to page marked  // "read-only"  BusErrorException, // Translation resulted in an  // invalid physical address  AddressErrorException, // Unaligned reference or one that  // was beyond the end of the  // address space  OverflowException, // Integer overflow in add or sub.  IllegalInstrException, // Unimplemented or reserved instr.    NumExceptionTypes  }; |   可以看到有一个异常叫做SyscallException 来进行系统调用。  当我们遇到一条系统调用的指令时：   |  | | --- | | //mipssim.cc :: OneInstruction  case OP\_SYSCALL:  RaiseException(SyscallException, 0);  return;  // machine.cc :: RaiseException  void  Machine::RaiseException(ExceptionType which, int badVAddr)  {  DEBUG('m', "Exception: %s\n", exceptionNames[which]);    // 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);  }  // exception.cc :: ExceptionHandler  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 {  printf("Unexpected user mode exception %d %d\n", which, type);  ASSERT(FALSE);  }  } |   可以看到这里先读取了2号寄存器，也就是之前设定SC\_Halt的寄存器。   |  | | --- | | int type = machine->ReadRegister(2); |   通过异常处理句柄，根据之前我们设定好的系统调用编号，硬编码进行系统调用。  整个流程如下：   1. 在用户程序中，调用了Halt()  |  | | --- | | // user.cc  //...  Halt();  //... |   这段代码编译得到user.o文件，和start.o连接，然后通过coff2noff转换为Nachos可执行文件user.noff   1. 当内存中装载这段代码之后，每次取一条指令，直到取到Halt()对应的指令 2. Halt指令先执行存根中的汇编，也就是把SC\_Halt放到了2号寄存器 3. 解码之后发现操作码是OP\_SYSCALL ，调用RaiseException。 4. RaiseExcepiton转调ExceptionHandler。 5. ExceptionHandler查二号寄存器，发现type是SC\_Halt，那么调用Halt()函数（Halt函数由操作系统内核负责实现）  Nachos系统调用：Exec(char\* filename) 在当前用户进程A中使用Exec系统调用指定一个可执行文件，然后启动另一个用户进程B执行该文件，在流程和逻辑上有两个难点：   1. 多道用户程序：当B进程执行结束，应该回到A进程继续执行，这要求不同用户程序有不同的地址空间。而当前Nachos仅支持**uniProgramming**，需要改进。 2. 系统调用参数：Exec系统调用需要指定文件名然后执行，想要执行这个文件势必要先将文件名参数加载到CPU寄存器中，然后才能装载这个文件到内存。细化到指令级别，怎样将这个参数传递给CPU寄存器？  多道用户程序的地址空间 首先先来看如何实现多道用户程序的地址分配，我们来看看为什么Nachos原有的地址分配策略不满足多道用户程序的需求。 原有的Nachos地址空间分配策略 地址分配的代码在AddressSpace的构造函数中（构造一个地址空间就相当于分配地址空间），分两部分：   1. 分配页表  |  | | --- | | // first, set up the translation  pageTable = new TranslationEntry[numPages];  for (i = 0; i < numPages; i++) {  pageTable[i].virtualPage = i; // for now, virtual page # = phys page #  pageTable[i].physicalPage = i;  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  } |   可以看到，在分配页表时，该用户程序的物理页physicalPage直接从i=0开始分配。  而在多道用户程序中，假设程序A先分配地址空间，假设程序A有3页。那么内存中0-2的物理页就分配给了A。   | **Memory Phy page** | **0** | **1** | **2** | **…** | | --- | --- | --- | --- | --- | | owner | A | A | A | / |   而之后我们想要装载另一个用户程序B到内存，假设B有两页。那么根据上述代码，内存中0-1页就分配给了B。   | **Memory Phy page** | **0** | **1** | **2** | **…** | | --- | --- | --- | --- | --- | | owner | B | B | A | / |   可是这样B的代码和数据就覆盖了A原有的数据。这样B执行完，A想要再去执行时，我们发现原先A的数据已经全部丢失了，那么A程序继续执行下去的逻辑可能和原先完全不一样，甚至可能崩溃。  这也是我们在多道用户程序中必须改进页表分配的原因。   1. 装载文件数据到内存  |  | | --- | | if (noffH.code.size > 0) {  DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",  noffH.code.virtualAddr, noffH.code.size);  executable->ReadAt(&(machine->mainMemory[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(&(machine->mainMemory[noffH.initData.virtualAddr]),  noffH.initData.size, noffH.initData.inFileAddr);  } |   这里要注意的一点是，每个文件的code段的virtualAddr一定是0而不是其他值。我先放个截图证明一下：  Untitled  这其实很容易理解，既然不同地址空间的页表的索引都是从0开始的，那code段的地址也可以从0开始，之后再根据内存的实际情况进行装入，这也是为什么这个字段被称作virtual的原因，是因为它不是装入内存的实际地址，而是一个虚拟地址。  可能有人问，既然我们有这个量了，为什么还要有inFileAddr这个量呢？这两个字段的作用不重复吗？需要注意的是，noff文件的头部一定连接了一个start.o的目标模块（这个之前已经详细阐释过了），因此段头的逻辑偏移量（virtualAddress）并不就是它在文件中的偏移量（inFileAddr），如下图所示：  在以上代码中，文件装入内存的位置就是这个虚拟地址，那么在多道用户程序中逻辑就有问题，详细逻辑在上一步中已经解释清楚了：新程序的数据会覆盖旧程序的数据。Untitled 1  在以上代码中，文件装入内存的位置就是这个虚拟地址，那么在多道用户程序中逻辑就有问题，详细逻辑在上一步中已经解释清楚了：新程序的数据会覆盖旧程序的数据。 **2.1.2. 位图分配空闲内存页** 既然我们的问题是新程序的数据会覆盖旧程序的数据。那么只要记录下每个程序占据的内存地址空间，就可以在分配新程序的地址时避开这些内存中的物理页。这就很容易让人想到之前在文件系统中接触过的位示图（bit map）。  简单来说，每当我们分配一个新程序的物理页时，先去位图中找有哪些内存页没有使用过，然后把这些物理页当作自己的地址空间，并标记这些页已被使用。  由于这个位图要被各种不同的用户程序访问，所以它不能是某个私有字段，我选择直接将它开成整个系统中的一个单例，也就是Nachos整个操作系统只使用某一个位图来管理它的物理内存分配情况。  由于实验要求中有对地址空间进行编号的需求，因此我还开了个全局的ID，用来分配给不同的地址空间。   |  | | --- | | //system.cc  #ifdef USER\_PROGRAM // requires either FILESYS or FILESYS\_STUB  Machine \*machine; // user program memory and registers  BitMap\* pageMap;  int ASID = 0;  #endif |   随后我们就可以使用这个位图来分配地址空间了：   |  | | --- | | //addrspace.cc :: AddrSpace  pageTable = new TranslationEntry[numPages];  for (i = 0; i < numPages; i++) {  /\*\*  \* Lab6:mup  \* the physicalPage should be an idle page on the page bit map  \*/  pageTable[i].virtualPage = i;  pageTable[i].physicalPage = pageMap->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  } |   还记得Find函数吗？它不仅找到了空闲的位，还把这个空闲的位标记为非空闲。 2.1.3. 用户程序的物理地址 分配完内存地址空间，我们就可以实际装入可执行文件的code和data段了。   |  | | --- | | if (noffH.code.size > 0) {  DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",  noffH.code.virtualAddr, noffH.code.size);  /\*\*  \* Lab6:mup  \* calculate the demanding number of pages  \* check the page table  \* then load the exe file's segment to the corresponding mem addr  \*/  pageStart = noffH.code.virtualAddr/PageSize;  pOffset = noffH.code.virtualAddr % PageSize;  frameStart = pageTable[pageStart].physicalPage;  executable->ReadAt(&(machine->mainMemory[frameStart\*PageSize+pOffset]),  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);  pageStart = noffH.initData.virtualAddr/PageSize;  pOffset = noffH.initData.virtualAddr % PageSize;  frameStart = pageTable[pageStart].physicalPage;  executable->ReadAt(&(machine->mainMemory[frameStart\*PageSize+pOffset]),  noffH.initData.size, noffH.initData.inFileAddr);  } |   之前提到了，virtualAddr只是一个段的虚拟地址，我们要把它转换到内存中实际有效的物理地址：   1. 先计算出这个虚拟地址之前的段用掉了多少页（整数）以及多少字节 2. 找到对应物理内存页，向内存读入该段   公式如下：   |  | | --- | | physAddr = pt\_phys\_page[floor(virtualAddr/PageSize)] \* PageSize + virtualAddr % PageSize |   示意图如下：  Untitled 2  可以看到，code段占了1页多三个字节，那么init data段的开始位置应该是b页开始往后三个字节。也就是pt\_phys\_page[floor( (128+3)/128 )] + (128+3) % 128 = b\*128+3字节的位置。 系统调用的参数是怎么传递的？ 现在我们解决了多道用户程序中的“多道”的问题，是时候解决参数传递的问题了。实验指导书告诉我们系统调用的声明将传递参数，具体原话如下：   |  | | --- | | 现在我们已经看到了系统调用 Halt 的汇编代码，它没有携带任何要传送到内核中  的参数。如果系统调用携带参数，编译又怎样来生成传送参数的指令呢？  所有 Nachos 系统调用的接口原型都定义在文件 userprog/syscall.h 中。当编译用户  程序时编译器会括入这个文件并取得这些系统调用接口原型的信息（这就是为什么  当你编译 halt.c 是 Makefile 文件中必须加入-I../userprog 定义）。 |   也就是如果我们定义了接口原型，那么编译器将生成对应的汇编指令，把我们传递的参数置入参数寄存器中以供我们后续调用。  但问题是，Exec系统调用是需要去根据入参的文件名去执行一个可执行文件的！那如果文件名特别长，存不进寄存器，应该怎么办？ 2.2.1. 参数指针传递到参数寄存器 其实仔细观察会发现，Exec系统调用传递的不是字符串本身，而是字符串数组的首地址（字符指针）:   |  | | --- | | SpaceId Exec(char \*name); |   字符指针的长度和字符串长度可差远了，char\* 类型的长度一般也就1个字节，这对于一个四个字节的寄存器来说很容易就能放下。  之后的1.3节中我将结合具体的MIPS模拟汇编指令，并辅以截图证明，进一步印证这个说法。 2.2.2. 参数值传递到Nachos虚拟机内存 好了，现在我们可以把这个字符串的指针作为参数放到寄存器中，随后的指令就能找到那个对应的可执行文件在哪了。  但问题是，既然没有一个寄存器能放得下文件名，我们势必要把文件名保存到Nachos虚拟机的内存中（注意是虚拟机内存，不是虚拟内存！前者是Nachos作为一个虚拟的操作系统，借用宿主机的内存；后者是在物理上不属于，但在逻辑上属于操作系统的内存。这两个概念是完全不相同的！），这样就可以根据寄存器中的指针去读对应的内存地址了，也就获得了文件名。  那么文件名是什么时候保存到内存中的呢？这个也要等到下面的1.3节来讲解。我会放上自己调试时的截图，并根据日志信息推理得到这个时机。 比较重要的Nachos-MIPS汇编指令 本小节承接1.2小节中关于系统调用的参数传递的部分，另外还观察了一些其他重要的汇编指令。 汇编指令解析 虽然此时我们还没有实现Exec系统调用，但我们可以摆一个空架子上去，然后观察在系统调用之前的日志信息。  在Nachos中的-d选项中（用于DEBUG），提供了m类型的DEBUG方式，将打印所有正在执行的Nachos-MIPS模拟汇编指令。   |  | | --- | | // mipssim.cc :: OneInstruction  if (DebugIsEnabled('m')) {  struct OpString \*str = &opStrings[instr->opCode];  ASSERT(instr->opCode <= MaxOpcode);  printf("At PC = 0x%x: ", registers[PCReg]);  printf(str->string, TypeToReg(str->args[0], instr),  TypeToReg(str->args[1], instr), TypeToReg(str->args[2], instr));  printf("\n");  } |   可以看到，我们把指令的类型、使用的寄存器、立即数都打印了出来。现在让我们看看系统调用之前的指令都有什么吧。  Untitled 3  先不要管一直在打印的Main Mem start from sa:那句log信息，这是我自己加的，后续将成为文件名装入内存时机的有力证据。  现在我们的注意力就在我用蓝色框标识出的两条log上。红色框的log说明此时处在第一次系统调用之前。  **ADDIU rt,rs,extra**  这条指令是一条ADD家族的指令，只不过是立即数加法。含义为：将rs寄存器中的值加上extra寄存器中的值，然后保存到rt寄存器中。  在这里，r0 = 0，r4 = r0 + 12345 = 0 + 12345 = 12345。而我们知道r4寄存器是参数寄存器之一，这在源码中作者已经说过了：   |  | | --- | | // exception.cc  // system call code -- r2  // arg1 -- r4  // arg2 -- r5  // arg3 -- r6  // arg4 -- r7 |   那为什么把12345保存到r4中？是因为我们的用户程序有PrintInt这个系统调用，参数就是12345。   |  | | --- | | // exec.cc  #include "syscall.h"  int  main()  {  SpaceId pid;  PrintInt(12345);  pid = Exec("../test/halt2.noff");  Halt();  /\* not reached \*/  } |   关于PrintInt的实现，详情见于第二大节。这里展示的用意是为了说明Nachos模拟CPU是如何把参数装入寄存器的。  现在再来看第二个篮筐中的log信息：**ADDIU r2,r0,11**。我们知道r2是系统调用返回值寄存器，也是系统调用代（码）号寄存器。而之前没提到的是，PrintInt的系统调用代码号就是11。   |  | | --- | | /\*  \* Lab6:mup  \* define the PrintInt index here  \*/  #define SC\_PrintInt 11 |   那么当Nachos模拟CPU抓取到SYSCALL这条指令时，它会根据驱动表去执行：   |  | | --- | | case OP\_SYSCALL:  RaiseException(SyscallException, 0);  return; |   之前已经提到过，Nachos进行系统调用的方式是异常处理。它会报告一个异常，然后把异常类型告诉处理函数。   |  | | --- | | void  Machine::RaiseException(ExceptionType which, int badVAddr)  {  DEBUG('m', "Exception: %s\n", exceptionNames[which]);    // 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);  } |   处理函数根据异常类型和r2的内容，决定这是哪一个异常（停机Halt系统调用？执行程序Exec系统调用？打印一个整数PrintInt系统调用？这要看r2寄存器的值决定了）：   |  | | --- | | 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 {  printf("Unexpected user mode exception %d %d\n", which, type);  ASSERT(FALSE);  }  } |   当然源码中还没有实现Exec或者PrintInt的系统调用，后续我们要在这里加上这两个分支，不过这已经是后面要讲的内容了。  这里我提及这一块内容的原因，是想说明系统调用的前置条件：CPU已经执行过装载系统调用代码号到r2寄存器的指令了。 空间较大的参数传入内存的时机 现在我们可以回顾一下1.2.2节中的部分了。之所以放在这里讲，是因为我要先展示汇编指令的执行，然后才能进行逻辑推理，证明我的想法。  先说结论：Exec(char\* filename)中的filename参数是在AddressSpace实例初始化时装入内存的。接下来我来证明这一点。  还记得1.3.1节中的截图吗？当时我说先忽略Main Mem start from sa:不看，现在这句log就有用了。  由于我通过控制台信息知道了文件名指针被装在内存空间的第304字节，截图为证：  **Untitled 4**  这个是我已经完成PrintInt后的截图，可以看到Exec是第二个系统调用，在此之前r4中的内容是304，这证明文件名指针在内存304字节处。  因此我编写了一个函数，专门用来打印内存的内容，每执行一条指令前都打印一次。这样我就能知道是什么指令执行完之后把文件名装入内存的了。   |  | | --- | | void printMainMemory(int sa,int size){  char buf[size];  for(int i=0;i<size;i++){  machine->ReadMem(sa+i,1,(int\*)&buf[i]);  }  printf("Main Mem start from sa: %d,size: %d,contents: %s\n",sa,size,buf);  } |   既然我都知道文件名在304字节处，调用时直接传sa=304，文件名我自己设定的最大长度50，那么size就传50。随后修改Run函数如下：   |  | | --- | | void  Machine::Run()  {  Instruction \*instr = new Instruction; // storage for decoded instruction  if(DebugIsEnabled('m'))  printf("Starting thread \"%s\" at time %d\n",  currentThread->getName(), stats->totalTicks);  interrupt->setStatus(UserMode);  for (;;) {  OneInstruction(instr);  if(DebugIsEnabled('m')){  printMainMemory(304,50);  }  interrupt->OneTick();  if (singleStep && (runUntilTime <= stats->totalTicks))  Debugger();  }  } |   然后我们来看看log信息变成啥样了。  Untitled 5  很神奇的是，在第一条指令执行时，该内存地址的内容就已经是这个文件名了（../test/halt2.noff）。这说明装入时机并不是在执行用户程序时。  那么装入时机一定在此之前，我们可以看下Run之前都干了什么：   |  | | --- | | void  StartProcess(char \*filename)  {  DEBUG('u',"Starting user process...\n");  OpenFile \*executable = fileSystem->Open(filename);  AddrSpace \*space;  if (executable == NULL) {  printf("Unable to open file %s\n", filename);  return;  }  space = new AddrSpace(executable);  currentThread->space = space;  delete executable; // close file  space->InitRegisters(); // set the initial register values  space->RestoreState(); // load page table register  if(DebugIsEnabled('m')){  printMainMemory(304,50);  }  DEBUG('u',"ready to start user process\n");  machine->Run(); // jump to the user program  ASSERT(FALSE); // machine->Run never returns;  // the address space exits  // by doing the syscall "exit"  } |   machine→Run是在StartProcess中调用的，之前这个函数做了：   1. 从磁盘打开指定的可执行文件（这里打开的是../test/exec.noff文件） 2. 为该文件分配内存地址空间 3. 让当前线程接管这个用户程序 4. 初始化寄存器 5. 将虚拟机当前使用的页表置换为当前用户程序的页表   OK，那么我们知道文件名是在内存中的，所以我们只要找之前这五步到底是**谁动的内存**就行了。  1是**文件系统和磁盘交互**，和内存无关。  2动了内存，往内存中写入了可执行文件的code和initData段。  3只是**声明谁负责这个用户程序**，没动内存。  4是**寄存器操作**，和内存没关系。  5只是**置换了页表**而已，没改内存。  那么究竟是谁把文件名写入内存中的，已经不言自明了。就是分配地址空间的构造函数！再来回顾一下：   |  | | --- | | int pageStart,frameStart,pOffset;  // then, copy in the code and data segments into memory  if (noffH.code.size > 0) {  DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",  noffH.code.virtualAddr, noffH.code.size);  /\*\*  \* Lab6:mup  \* calculate the demanding number of pages  \* check the page table  \* then load the exe file's segment to the corresponding mem addr  \*/  pageStart = noffH.code.virtualAddr/PageSize;  pOffset = noffH.code.virtualAddr % PageSize;  frameStart = pageTable[pageStart].physicalPage;  executable->ReadAt(&(machine->mainMemory[frameStart\*PageSize+pOffset]),  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);  pageStart = noffH.initData.virtualAddr/PageSize;  pOffset = noffH.initData.virtualAddr % PageSize;  frameStart = pageTable[pageStart].physicalPage;  executable->ReadAt(&(machine->mainMemory[frameStart\*PageSize+pOffset]),  noffH.initData.size, noffH.initData.inFileAddr);  } |   所以实际的流程是，**编译器在生成可执行文件时就将文件名作为常量编译到了可执行文件，一并编译的还有该文件名字符串的指针**。因此在之前看到的ADDIU这条汇编指令中才会有304这个常量。  还有一个证据可以证明这个观点，我们用vim打开这个Unix文件：  Untitled 6  Exec的参数文件名赫然在目。 StartProcess入口与嵌套Exec调用 Nachos通过-x选项提供了执行用户程序的功能，而负责-x选项的函数就是StartProcess函数。这也是为什么我称这个函数为”入口“的原因。这个函数之前展示过不止一次了，这里不再赘述。 从Exec系统调用的视角看StartProcess函数 其实，StartProcess又何尝不是一种Exec系统调用呢？只不过StartProcess面向命令行参数，Exec面向用户代码而已。二者的流程本质上几乎一样，可以归纳为：   1. 打开指定的可执行文件 2. 为这个可执行文件分配地址空间 3. 初始化虚拟机CPU寄存器 4. 置换当前使用的页表 5. 起一个内核线程负责这个用户程序（当然StartProcess起的这个线程就是当前线程）   不过还有一点不一样，Exec函数是带返回值的，而返回值放在r2寄存器里，因此我们还得把地址空间id写到2号寄存器里面去。  接下来就可以按照上述流程编码了：   |  | | --- | | 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();  } |   在这里为了让Exec可以立即得到执行，把当前线程踢出了CPU，让新起的线程执行。而后者的函数为：   |  | | --- | | // dummy arg,useless  void userFunc(\_int dummy){  // run the user prog  machine->Run();  // never returns  ASSERT(FALSE);  } |   即让Nachos虚拟机按照当前页表以及内存中的数据执行用户程序。注意，Fork的时候无法指定用户程序，因为内核是不知道用户程序具体内容是什么的。用户程序的具体内容在虚拟机内存里。 嵌套使用Exec系统调用 这个系统调用很明显可以嵌套使用，既然我能在StartProcess里套一个Exec，为什么不能再Exec中套一个Exec呢？  我编写了一个简单的测试程序，在其中调用Exec   |  | | --- | | #include "syscall.h"  int  main()  {  SpaceId pid;  PrintInt(54321);  pid = Exec("../test/exec.noff");  Halt();  /\* not reached \*/  } |   **Untitled 7**  可以看到，先打印了我的测试程序中的54321，然后打印的exec.noff中的12345，当然下面还有再Exec(../test/halt2.noff)的嵌套调用，不过截图太长我就没放上来了。 系统调用总体流程一览 接下来我将从编译期开始，说明系统调用的整体流程，假设可执行文件的名字叫做exec.c。   1. gcc编译得到目标文件exec.o 2. 通过bin下Nachos官方造的轮子coff2flat和coff2noff，把exec.o转成noff可执行文件exec.noff，主要是在开头连接上了start.o目标模块 3. 通过-x选项执行exec.noff文件 4. 打开exec.noff文件 5. 分配地址空间 6. 初始化CPU寄存器 7. 置换页表 8. 起一个内核线程执行用户程序 9. 取指令 10. 执行指令 11. 普通指令，按正常逻辑执行 12. 系统调用指令，根据r2中的系统调用号进行对应的系统调用逻辑，这一段代码在os内核中 13. 结束用户程序执行。   流程图如下：  **Untitled 8** Nachos系统调用：PrintInt(int num) 在Nachos源码中，是没有PrintInt这个系统调用的，所以一些配置方面的代码还得我们自己写。 增加系统调用代（码）号和定义 syscall.h文件中配置系统调用码，主要是为了给存根用的。定义是为了让编译器知道这个系统调用有参数。   |  | | --- | | /\*  \* Lab6:mup  \* define the PrintInt index here  \*/  #define SC\_PrintInt 11  /\*\*  \* Lab6:mup  \* syscall PrintInt  \*/  void PrintInt(int num); |  增加存根 start.s中配置存根，主要是为了把系统调用码存进r2寄存器。这样异常发生后可以知道是哪个系统调用。   |  | | --- | | .globl PrintInt  .ent PrintInt  PrintInt:  addiu $2,$0,SC\_PrintInt  syscall  j $31  .end PrintInt |  Tips:PC自增 之前OneInstruction函数中最后有PC自增的部分。但现在系统调用会陷入异常，所以我们在系统调用结束后要在异常处理函数中PC自增（Halt不自增是因为停机了，没必要自增）。不然就会一直在系统调用（毕竟取指周期不会PC自增）   |  | | --- | | /\*\*  \* Lab6:mup  \* From contents for the ExceptionHandler:  \* And don't forget to increment the pc before returning. (Or else you'll  \* loop making the same system call forever!  \* since the registers field is public,just get access to it!  \*/  void IncPc(){  machine->registers[PrevPCReg] = machine->registers[PCReg];  machine->registers[PCReg] = machine->registers[NextPCReg];  machine->registers[NextPCReg] += 4;  } |  实现 首先ExceptionHandler中的驱动表肯定是要的，然后就是调用自己实现的函数。   |  | | --- | | void  ExceptionHandler(ExceptionType which)  {  int type = machine->ReadRegister(2);  if(which == SyscallException){  if(type == SC\_Halt){  DEBUG('a', "Shutdown, initiated by user program.\n");  interrupt->Halt();  // never reached  return;  }  else if(type == SC\_Exec){  interrupt->Exec();  }  else if(type == SC\_PrintInt){  interrupt->PrintInt();  }  // Note that the PC increases  IncPc();  return;  }else{  printf("Unexpected user mode exception %d %d\n", which, type);  ASSERT(FALSE);  }  }  void  Interrupt::PrintInt() {  // the arg of syscall is in reg 4  printf("%d\n",machine->ReadRegister(4));  } |   这个系统调用逻辑很简单，没啥好说的。 复现Linux/Unix下的Fork()/Exec()/Copy-On-Write复现Fork(void (\*func)()) 这里Nachos官方给出注释了，要求在原地址空间下进行Fork：   |  | | --- | | /\* Fork a thread to run a procedure ("func") in the \*same\* address space  \* as the current thread.  \*/  void Fork(void (\*func)()); |   其实这样反而好做，毕竟func已经保存到当前的地址空间中了，重开一个地址空间还得把func复制过去，既浪费内存又浪费时间。  存根和定义都做好了，现在就是做两件事情：   1. 修改ExceptionHandler驱动表 2. 实现Fork系统调用的内部逻辑  修改EH驱动表  |  | | --- | | void  ExceptionHandler(ExceptionType which)  {  int type = machine->ReadRegister(2);  if(which == SyscallException){  if(type == SC\_Halt){  DEBUG('a', "Shutdown, initiated by user program.\n");  interrupt->Halt();  // never reached  return;  }  else if(type == SC\_Exec){  interrupt->Exec();  }  else if(type == SC\_PrintInt){  interrupt->PrintInt();  }else if(type == SC\_Fork){  interrupt->Fork();  }  // Note that the PC increases  IncPc();  return;  }else{  printf("Unexpected user mode exception %d %d\n", which, type);  ASSERT(FALSE);  }  } |   其实就加了一行，type == SC\_Fork。 实现Fork逻辑 一开始的套路都是一样的，我们要执行一个func函数，而这个函数已经在StartProcess中被装入Nachos虚拟机内存了，这一部分之前已经详述过，不再赘述。所以我们先从参数寄存器r4中拿到这个函数的指针:   |  | | --- | | void  Interrupt::Fork(){  printf("Execute system call of Exec()\n");  /\*\*  \* The start addr of func is in the r4 reg  \*/  int funcReg = 4;  int funcAddr = machine->ReadRegister(funcReg); |   之后的问题是，拿到这个函数的首地址后，怎么开始执行这个函数？  从代码的角度来看，有点难想。但是**从指令的角度**看，就很容易了：这个函数经过编译后无非也就是一堆指令，我已经拿到这堆指令的首地址了，那么直接让PC跳到这个首地址上不久可以执行这个函数了吗？  所以下面一步就是PC跳转：   |  | | --- | | /\*\*  \* Let the PC jump to the function entry  \*/  machine->registers[PCReg] = funcAddr;  machine->registers[PrevPCReg] = (funcAddr>=4)?(funcAddr-4):0;  machine->registers[NextPCReg] = funcAddr+4; |   由于我们只是和父进程共享了同一块地址空间，但本质上为了并发效率，还是要Fork一个kernel线程来执行这个用户程序中的func，完整代码:   |  | | --- | | void  Interrupt::Fork(){  printf("Execute system call of Exec()\n");  /\*\*  \* The start addr of func is in the r4 reg  \*/  int funcReg = 4;  int funcAddr = machine->ReadRegister(funcReg);  /\*\*  \* Let the PC jump to the function entry  \*/  machine->registers[PCReg] = funcAddr;  machine->registers[PrevPCReg] = (funcAddr>=4)?(funcAddr-4):0;  machine->registers[NextPCReg] = funcAddr+4;  /\*\*  \* new thread sharing the same space with the old thread  \*/  Thread\* tfu = new Thread(currentThread->getName());  tfu->space = currentThread->space;  tfu->Fork(userFunc,0);  // LET'S GO!  currentThread->Yield();  } |  编写测试程序 我编写的测试程序如下，用户进程Fork一个子线程，然后子线程调用之前实现过的PrintInt系统调用，用户进程和子线程并发调用PrintInt。   |  | | --- | | #include "syscall.h"  void func(){  PrintInt(54321);  }  int  main()  {  Fork(func);  PrintInt(12138);  Halt();  /\* not reached \*/  } |   测试结果如下：  IMG_256  首先可以看到，只使用了一个地址空间。其次可以看到，子线程打印54321，父进程打印12138。测试通过。 一点小的疑问：为什么能正确执行 其实起初我并不确定是否能正确执行，随手跑了一遍没想到过了。当时疑惑的点在于：子线程Fork后，我并没有更新ra的值（返回地址寄存器），为什么PC能回到之前执行过的地方？  IMG_256  如图，Fork系统调用结束后，怎么回到的RA？我在代码中没有手动更新过？  为了解决这个问题，我加入-d选项进行DEBUG信息打印，得到：  IMG_256  那么JR指令干了什么？   |  | | --- | | case OP\_JR:  pcAfter = registers[instr->rs];  break; |   pcAfter即将赋值给NextPCReg，也就是说把r31，也就是ra的值给了指针的下一个位置上。我们可以从DEBUG信息中看出一些端倪：   1. PC = 0X18 2. PC = 0X1C 3. PC = 0XF8   1到2还是正常自增（一次4个字节），而2到3就是突然的跳转了。  这是因为1的时候NextPC = 0XF8。但结束时PC更新的值仍然是0X1C。但2的时候PC的值更新为NextPC，就变成0XF8了。   |  | | --- | | registers[PrevPCReg] = registers[PCReg]; // for debugging, in case we  // are jumping into lala-land  registers[PCReg] = registers[NextPCReg];  registers[NextPCReg] = pcAfter; |   这样就相当于跳回到主进程中了。  那么这个JR哪里来的呢？请看存根。   |  | | --- | | .globl Fork  .ent Fork  Fork:  addiu $2,$0,SC\_Fork  syscall  j $31  .end Fork |   原来早在存根中，就已经写好syscall系统调用之后PC返回RA了。 复现Unix下Exec(char\* name) 实验指导书中提到过，Unix/Linux下的exec系统调用会覆盖调用者的地址空间执行。这是和之前我们实现的Exec系统调用不一样的地方，其他的逻辑（比如取文件名指针等）都可以仿制。 增加系统调用号  |  | | --- | | /\*\*  \* Lab6:mup extension  \* define the UExec index here  \*/  #define SC\_UExec 12  /\*\*  \* Lab6:mup extension  \* define the unix exec syscall here  \*/  SpaceId UExec(char \*name); |  增加存根  |  | | --- | | .globl UExec  .ent UExec  UExec:  addiu $2,$0,SC\_UExec  syscall  j $31  .end UExec |  修改EH驱动表  |  | | --- | | }else if(type == SC\_UExec){  interrupt->UExec();  } |  实现 这里我采用的实现方式是直接释放原有的地址空间的物理页，然后分配给现在的用户程序。之所以不直接替换，是因为当前用户程序的内存占用大小可能和原先的内存占用大小不同。重分配看似效率比较低，但是可以保证逻辑正确，强于直接替换。  除此之外，并无与原先Exec系统调用不同之处:   |  | | --- | | /\*\*  \* Lab6:mup extension  \* simulate the exec() syscall on Unix  \* cover the cur thread's addr space to run  \*/  void  Interrupt::UExec() {  printf("Execute system call of Unix 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("Unix Exec(%s):\n",filename);  OpenFile\* exe = fileSystem->Open(filename);  if(exe == NULL){  printf("no such file: %s\n",filename);  return;  }  // get the cur page table  TranslationEntry\* cpt = currentThread->space->getPT();  // deallocate the pages on this page table  unsigned int np = currentThread->space->getNumPages();  for(int i=0;i<np;i++){  pageMap->Clear(cpt[i].physicalPage);  }  // reallocate 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();  } |  编写测试程序 仿照exec.c，编写了一版用于Unix Exec的用户测试程序:   |  | | --- | | #include "syscall.h"  int  main()  {  SpaceId pid;  PrintInt(12345);  pid = UExec("../test/halt2.noff");  Halt();  /\* not reached \*/  } |   测试结果如下：  IMG_256  可以看到，新地址空间和旧地址空间用的都是物理页的0-10页。且用户程序执行逻辑正确，测试通过。 复现Copy-On-Write机制什么是COW？ Copy-On-Write机制是一种延迟策略（Delayed Strategy），或者可以说是一种按需分配内存的策略（不同于虚拟内存的按需分配）。  假设我们有一对父子线程。在之前的实现中，我们大多数时候都是为两个线程分配不同的地址空间，但我们知道，父子线程可能使用一些共享的变量或字段，这些部分都是可以复用的，没有必要开两块内存空间给共享的变量。于是就有了COW机制。  简单来说，我们先为父子线程统一分配共享的地址空间，但我们把地址空间中的每一页都标记为只读（Read-Only），如果子线程需要向地址空间写入时，就会建立属于自己的地址空间，然后复制一份父进程的数据到自己的地址空间。这样的优势是如果子线程根本无需写入的话，就只需要父线程的一份地址空间即可，大大省下了内存空间。  示意图如下（我从网上查资料时找到的）：  IMG_256  IMG_256  可以看到，一开始父子进程/线程共享页表和实际物理内存。但一旦子线程想要更改空间，那么我们要分配额外的物理空间给子线程，并且更新子线程的页表，让它指向自己现在的物理内存空间。 一点理论上的疑问 之前我根据自己的理解，一直以为子线程一开始没有自己的地址空间，那么既然没有地址空间，又怎么取指令执行子线程呢？后来我查阅了资料得知，子线程一开始不是没有地址空间，而是和父线程共享地址空间，也就是子线程地址空间的指针指向父线程。只有在需要Write时才会复制一份新的出来 标识父线程的物理页 既然有父子线程，那么肯定得有Fork函数。不过之前实现的Fork函数已经让子线程共享父线程了，所以不需要更改。  现在的问题是，我们要给父线程的物理页上打上标记，让子线程不能修改这些物理页，而是给子线程分配额外的专属物理页。此外，很重要的一点，我们不能简单的标记这些页为只读，因为一旦标记了只读，那么父线程也不能修改自己的物理页了，所以关键在于找到一种标记方法，让父线程能修改自己的物理页的同时子线程不能修改这些页。  **修改线程数据结构**  在原先的Nachos实现中，线程是没有自己的PID的，标识线程的字段是name（线程名称），而我们知道name这个字段是很容易重名的，尤其是父子线程重名的可能性非常大，所以第一点我们要给线程加上一个唯一标识符，用来区分不同线程（尤其是父子线程）。   |  | | --- | | // thread.h  private:  int\* stack;  ThreadStatus status; // ready, running or blocked  char\* name;  int pid; // global unique identifer of thread |   其他一些基本的getter\setter就不赘述了。  **为页表项注入PID**  我想通过让页表项有一个owner属性来标识这一页属于哪个线程，这样如果不属于当前线程，并且当前线程想要更改这一页，我们就给当前线程分配一页新的物理页。  因此修改TranslationEntry数据结构如下：   |  | | --- | | class TranslationEntry {  public:  int virtualPage; // The page number in virtual memory.  int physicalPage; // The page number in real memory (relative to the  // start of "mainMemory"  bool valid; // If this bit is set, the translation is ignored.  // (In other words, the entry hasn't been initialized.)  bool readOnly; // If this bit is set, the user program is not allowed  // to modify the contents of the page.  bool use; // This bit is set by the hardware every time the  // page is referenced or modified.  bool dirty; // This bit is set by the hardware every time the  // page is modified.  int ownerPid; // belong to which thread  }; |   新增了一个字段ownerPid，用来标识该页属于哪个线程。有可能父子线程共享内存，所以某个表项的拥有者可能不止一个，因此若父子线程共享某块内存空间，对应的表项的ownerPid=-1(专门用来标识共享内存）。  **为页表项注册线程**  我们应该在为线程分配地址空间时为某一个物理页注册拥有者，除了这个拥有者，没有线程有对该内存空间的Write权限。   |  | | --- | | pageTable[i].virtualPage = i;  pageTable[i].physicalPage = pageMap->Find();  pageTable[i].valid = TRUE;  pageTable[i].use = FALSE;  pageTable[i].dirty = FALSE;  pageTable[i].readOnly = FALSE;  pageTable[i].ownerPid = currentThread->getPID(); |   这样我们就标识了每个线程的物理页。此外，我们还要保证地址空间的id就是当前线程的id，这样在后续MMU转换检查时，我们就可以根据当前线程的地址空间的id是否是当前线程的id来进行Write权限检查了。   |  | | --- | | asId = owner->getPid();  printf("spaceId:%d\n",asId); |  MMU映射检查 在Nachos中，任何操作内存的操作都要经过MMU映射，因此，我们要让MMU替我们把关，如果一个线程妄图更改不属于自己的物理页，我们应该复制一份物理页给它，然后它可以更改副本，而非原先的地址空间（copy-on-write）。   |  | | --- | | if (writing){  entry->dirty = TRUE;  /\*\*  \* Lab6:mup extension  \* copy on writing  \* check if the cur page table belongs to cur thread  \* if not,copy the page table to the cur thread's page table  \* then the cur thread can modify the mem data on it's copy  \*/  // check if cur thread is writing the space which is not belong to it  int oid = machine->pageTable[virtAddr/PageSize].ownerPid  bool flag = oid != SHARING && oid != currentThread->getPID();  if(flag){  TranslationEntry\* oas = currentThread->space;  // check if whether cur thread has its own addr space  if(currentThread->getPID()!=oas->getASID()){  // allocate a new addr space for cur thread  TranslationEntry\* nPT = new TranslationEntry[oas->getNumPages()];  for(int i=0;i<oas->getNumPages;i++){  nPT[i].virtualPage = i;  nPT[i].physicalPage = pageMap->Find();  //copy the data  for(int j=0;j<PageSize;j++){  machine->mainMemory[nPT[i].physicalPage\*PageSize+j]=  machine->mainMemory[oas->getPT()[i].physicalPage\*PageSize+j];  }  nPT[i].valid = TRUE;  nPT[i].use = FALSE;  nPT[i].dirty = FALSE;  nPT[i].readOnly = FALSE;  nPT[i].ownerPid = currentThread->getPID();  }  currentThread->space = new AddrSpace();// consume we support the constructor without arg  currentThread->space->setPT(nPT);  // update the as id for next check  currentThread->space->setASID(currentThread->getPID());  }  }  } |   流程如下：   1. 首先我们检查的是写权限，因为读权限无论哪个子线程都有。如果当前线程正在写，则进入检查。 2. 把当前页表项置为dirty（这个是为了虚存页置换，和检查无关） 3. 检查当前线程是否在写不属于自己的内存（子线程可能在写共享内存，如果写共享内存就不用copy on write了） 4. 如果当前线程在写不属于自己的内存，我们应该为它分配额外物理空间。 5. 在分配额外地址空间时，先将原内存空间的数据复制过去以便子线程修改。并且要标识新地址空间表项的拥有者。   流程图如下：  IMG_256 | | | |
| 结论分析与体会：  在Lab1探寻Nachos模拟CPU指令时，我曾略微看过一点Machine类。Lab6要求我们更深入地了解这个模块，以达到最后能熟练运用的程度。  Nachos开辟了宿主机内存空间中的一部分作为自己的虚拟机内存空间，并采用代码的方式模拟了一个简易的MIPS架构的CPU（之所以说简易是因为只实现了部分指令）。并通过自己编写的coff2noff等工具，将start.o目标模块连接到Unix/Linux的可执行文件头部，形成专属于Nachos的可执行文件(.noff文件)。这样CPU可以通过读取noff文件，并把程序、数据段装载到虚拟机内存中，之后CPU就可以进入取指周期、执行周期、下一个取指周期…..进而执行用户程序。  在Nachos中，管理用户程序地址空间采用了最简单的一级页表，这是一个可以改进的点，我们可以尝试把这一块改进为多级页表（这个多级页表的级数也不一定要是定死的，可以根据用户程序的大小来调整），不过最终我没有实现这个改进点。  在用户进程执行时，先把虚拟机当前使用的页表替换为该用户程序的页表（存放在执行用户程序的内核线程中），然后初始化用户寄存器，即可执行用户程序。有意思的是，初始化用户寄存器的时机也很重要，在这一点上我在Lab7中遇到了一个很有意思的BUG，这个BUG的复现以及解决方案我已经写在了Lab7的实验报告中。  在了解了Nachos虚拟机的架构以及工作原理后，我尝试动手实现Exec系统调用。首先要修改地址空间分配物理页方面的代码。这一块我选择使用Lab4/5中用到的位图来标记空闲也。随后我通过了DEBUG日志信息，探寻了系统调用的参数是何时、以何种形式传递到用户寄存器的。最终我了解到r4寄存器保存了系统调用参数的指针，通过该指针就可以先经过MMU，然后在虚拟机内存中找到对应的参数。  在这个过程中我遇到了一些BUG，我通过-d a和-d m选项观察了寄存器、虚拟机内存以及汇编指令的情况。这一部分一开始对于我来说还是挺困难的，因为我根本没写过汇编。因此我只能一条条的查找MIPS指令集的汇编命令的用法，最终搞清楚了JAL\SYSCALL\ADDIU\SW等汇编指令的用法，解决了问题。  我不仅完成了实验要求的基本内容。还实现了一些额外功能。比如，我实现的Exec系统调用是可以嵌套调用的。我还实现了Fork系统调用、Unix/Linux下覆盖原地址空间的Exec系统调用，这些内容以及测试过程我都写在了这篇报告中。  最后，我根据上学期理论课的内容，研究了COW（Copy-On-Write）机制，并修改了页表项和MMU部分的代码，尝试复现了COW机制。  Lab6的内容让我感觉Nachos真的成为了一台虚拟机上的操作系统，之前的实验中我们只是完成了Nachos里的各个模块，但并没有有机地结合在一起。Lab6将之前实验中的线程、位图、模拟CPU、汇编指令结合起来，让我对用户程序的装载和执行有了更进一步的认知。 | | | |