|  |
| --- |
| 实验目的：  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()。 |
| 硬件环境：  惠普品牌型号笔记本  Intel Core i5-8300 CPU  8GB内存  512GB SSD |
| 软件环境：  宿主机：Windows 10 21H1 64位  虚拟机软件：VMware Workstation Pro 16.1.2 build-17966106  Linux：Ubuntu 14.04.6 LTS Desktop i386 (Trusty Tahr)  gcc/g++：(Ubuntu 4.8.4-2ubuntu1~14.04.4) 4.8.4  MIPS交叉编译器：gcc-2.8.1-mips.tar.gz  Nachos：Nachos-3.4-UALR-LW |
| 实验步骤与内容：  [Lab6 系统调用与多道用户程序](#lab6-系统调用与多道用户程序)  [6.1 实验内容](#Xca4140f1669fb5952c0796bf1b8fe5c2c974ceb)  [6.2 实验思路](#X63811bb9011f2b938511ec30064a73ffd3421b9)  [6.3 实验代码](#X6938418191aedc54007a62324ecac88d5b139a2)  [6.3.1 添加Print函数](#X6e1f0e21c9e4603e7d2c5e16a7010d7e05ca370)  [6.3.2 扩展Nachos AddSpace类](#Xeb283bbe192f3cc3882c2f981e70c960fce46df)  [6.3.3 实现系统调用Exec()](#X22017c2e9313b311cd9237e43f7efdf7cdc7b1e) Lab6 系统调用与多道用户程序6.1 实验内容  1. 扩展现有的class AddrSpace的实现，使得Nachos可以实现多道用户程序。 2. 按照实验指导书中的方法，完成class AddrSpace中的Print函数。 3. 实现Nachos 系统调用：Exec()。  6.2 实验思路  * **如何理解用户进程如何映射到一个核心线程** * 在 progtest.cc中的StartProcess 函数，在为用户程序初始化了地址空间之后，执行一条命令 currentThread->space = space，此命令即将用户进程映射到了核心线程之上。 * **如何理解当前进程的页表是如何与CPU使用的页表进行关联的** * 在创建用户进程的地址空间时，创建用户进程的页表项 * pageTable = new TranslationEntry[numPages];// 初始化页表数组 for (i = 0; i < numPages; i++) {// 对每个页表初始化  pageTable[i].virtualPage = i; //目前为止，物理地址等于虚拟逻辑地址  pageTable[i].physicalPage = i;  pageTable[i].valid = TRUE;  pageTable[i].use = FALSE;  pageTable[i].dirty = FALSE;  pageTable[i].readOnly = FALSE;  } * 创建完页表之后，回到 StartProcess 函数中，调用了 space->RestoreState() 命令，该命令将用户进程的页表赋值给了 Machine 页表，具体代码如下所示： * void AddrSpace::RestoreState()  {  machine->pageTable = pageTable;  machine->pageTableSize = numPages; } * 之后程序运行的过程就是通过 PC 寄存器中的虚拟地址通过 Machine 中的页表转化为物理地址，然后在将根据指令类型执行该指令。 * **如何实现内存页面分配** * 实现多进程机制比较关键的问题就是**页**的分配问题，**Nachos默认机制下的内存分配是每个进程的虚拟逻辑地址与实际物理地址相同**，且每个进程都是从0开始分配，这样如果有多道用户程序，内存中就会来回被覆盖： * pageTable[i].virtualPage = i; //目前为止，物理地址等于虚拟逻辑地址 pageTable[i].physicalPage = i; * 我们采用bitmap数据结构保存空余页：MIPS模拟机(machine类)内存中有32个物理页面，也就是需要一个int(32位)，可以保存32个内存页面的空闲情况。bitmap的大小为32bits。主要使用bitmap的find()方法 * #define BitsInByte 8 #define BitsInWord 32  class BitMap {  public:  BitMap(int nitems); // Initialize a bitmap, with "nitems" bits  // initially, all bits are cleared.  ~BitMap(); // De-allocate bitmap    void Mark(int which); // Set the "nth" bit  void Clear(int which); // Clear the "nth" bit  bool Test(int which); // Is the "nth" bit set?  int Find(); // Return the # of a clear bit, and as a side 从前到后找到第一个0的位置，顺便置为1  // effect, set the bit.   // If no bits are clear, return -1.  int NumClear(); // Return the number of clear bits   void Print(); // Print contents of bitmap    // These aren't needed until FILESYS, when we will need to read and   // write the bitmap to a file  void FetchFrom(OpenFile \*file); // fetch contents from disk   void WriteBack(OpenFile \*file); // write contents to disk   private:  int numBits; // number of bits in the bitmap  int numWords; // number of words of bitmap storage  // (rounded up if numBits is not a  // multiple of the number of bits in  // a word)  unsigned int \*map; // bit storage }; * **如何实现多道用户程序** * 在我们完成了多个程序同时驻留内存的内存分配算法后，我们就应当考虑用户父子进程并发执行的问题了。为了完成这一功能需要实现Exec系统调用的异常处理函数。  6.3 实验代码6.3.1 添加Print函数 研究一下../test 目录中 Makefile 文件的内容，../test 中现有 5 个 C 语言用户源程序，可以通过 make 命令一次性编译连接生成它们的可执行文件和其在该目录中的符号链接。  image-20211214131357041  image-20211214131357041  image-20211214131534461  image-20211214131534461  切换到./userprog/文件夹下，为了能够了解 Nachos 中多用户程序驻留内存的情况，可以在AddSpace类中增加以下打印成员函数Print()：  void AddrSpace::Print() {  printf("process spaceId: %d",spaceId);  printf("page table dump: %d pages in total\n", numPages);  printf("============================================\n");  printf("\tVirtPage, \tPhysPage\n");  for (int i=0; i < numPages; i++) {  printf("\t%d, \t\t%d\n", pageTable[i].virtualPage, pageTable[i].physicalPage);  }  printf("============================================\n\n"); }  在progtest.cc的StartProcess(char \*filename)方法中添加使得当为一个应用程序新建一个空间后，调用 Print() 函数来输出页表信息，具体修改结果如下所示：  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;  space->Print();// 打印内存使用情况   delete executable; // close file   space->InitRegisters(); // set the initial register values  space->RestoreState(); // load page table register   machine->Run(); // jump to the user progam  ASSERT(FALSE); // machine->Run never returns;  // the address space exits  // by doing the syscall "exit" }  编译Nachos内核，执行一个用户程序：  ./nachos -x ../test/halt.noff  image-20211214132137883  image-20211214132137883  查看 halt.noff 虚拟逻辑地址页号与物理地址页号的对应关系，以及 Nachos 为该程序分配的实页数。可以看到 Nachos 为该程序分配了 10 个实页。 6.3.2 扩展Nachos AddSpace类 对于实现内存页面的调用，添加两个静态全局变量Bitmap标识内存中的空页与进程空间池，添加一个成员变量spaceID表示当前进程的地址空间标识符，以及一个get函数：  int getSpaceId();  private:  static BitMap \*freeMap,\*spaceIdMap;  int spaceId; // 地址空间标识符  在AddSpace.cc的文件内进行初始化（全局变量不能在构造函数中初始化，类成员共有）  BitMap \*AddrSpace:: freeMap = new BitMap(NumPhysPages);// 初始化bitMap BitMap \*AddrSpace:: spaceIdMap = new BitMap(256);// 定义spaceID池，  //一个用户进程对应一个唯一的地址空间id，大小为256个  初始化AddSpace时给sapceID赋值，在位图SpaceIdMap中查找未被分配的id号：  // 分配进程空间标识符 ASSERT(spaceIdMap->NumClear() > 0);// 确认页面足够分配 spaceId = spaceIdMap->Find();  接下来我们修改虚实页面分配的代码，对于每一个虚页，**我们在位图freeMap中找一个未被分配的页面作为虚页映射。**除此之外，我们需要保证在分配之前物理内存中空闲页面数量大于等于我们需要的页面数量，具体代码如下所示。  pageTable = new TranslationEntry[numPages]; ASSERT(freeMap->NumClear()>= numPages);// 确认页面足够分配 for (i = 0; i < numPages; i++) {  pageTable[i].virtualPage = i;// 虚拟页面视图，从0开始  pageTable[i].physicalPage = freeMap->Find();// 在位图中找到空闲页分配  pageTable[i].valid = TRUE;  pageTable[i].use = FALSE;  pageTable[i].dirty = FALSE;  pageTable[i].readOnly = FALSE;  }  进行虚实页映射之后，我们需要将 noff 文件中的数据拷贝到machine的物理内存mainMemory中，因此我们需要将虚拟地址所对应的物理地址求出：**求出代码段或数据段的页表项，根据该页表项的物理页号 × 页大小 = 物理内存地址，再求出代码段或数据段的偏移量（无法被页大小整除的部分），相加得到起始物理内存地址。**  if (noffH.code.size > 0) {  // pageTable[noffH.code.virtualAddr/PageSize]表示起始的页表项  int pagePosition = pageTable[noffH.code.virtualAddr/PageSize].physicalPage \* PageSize;//计算出代码段在内存数组的起始下标  int offset = noffH.code.virtualAddr % PageSize;// 代码段的偏移量  DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",  (pagePosition+offset), noffH.code.size);// 修改DEBUG信息  executable->ReadAt(&(machine->mainMemory[pagePosition+offset]),  noffH.code.size, noffH.code.inFileAddr);//从pagePosition+offset处分配size大小的内存空间 } if (noffH.initData.size > 0) {  int pagePosition = pageTable[noffH.initData.virtualAddr/PageSize].physicalPage \* PageSize;//计算出代码段在内存数组的起始下标  int offset = noffH.initData.virtualAddr % PageSize;// 代码段的偏移量  DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",  (pagePosition+offset), noffH.initData.size);// 修改DEBUG信息  executable->ReadAt(&(machine->mainMemory[pagePosition+offset]),  noffH.initData.size, noffH.initData.inFileAddr);//从pagePosition+offset处分配size大小的内存空间 }  最后我们需要修改 AddrSpace 类的析构函数，我们需要在 AddrSpace 析构的时候将对应的位图的标识释放，具体代码如下所示：  AddrSpace::~AddrSpace() {  for(int i = 0;i < numPages;i++){  freeMap->Clear(pageTable[i].physicalPage);// 释放位图中的内存页标识  }  spaceIdMap->Clear(spaceId);// 释放位图中的进程地址空间标识符  delete [] pageTable; } 6.3.3 实现系统调用Exec() 为了完成这一功能首先可以准备一个作为父进程的用户程序exec.c：  #include "syscall.h" int main() {  SpaceId pid;  pid = Exec("../test/halt.noff");  Halt(); }  为了能和已有的用户 C 程序一起生成可执行文件，可以修改../test/Makefile 文件将 exec 加入到 targets 定义中，在../test 中重新 make 生成 exec.noff 可执行文件。  targets = halt shell matmult sort exec  **获取参数：**在exec.c中为了生成子进程 halt.noff，使用了 Nachos 的系统调用 Exec。它带有一个字符串参数，是一个可执行文件名。在发生系统调用时系统内核需要得到这个**参数**并根据它建立子进程。  这里我们先看对应于 exec.c 的汇编代码，了解一下 MIPS 机指令系统对于参数传递是如何安排的：  .file 1 "exec.c" gcc2\_compiled.: \_\_gnu\_compiled\_c:  .rdata  .align 2 $LC0:  .ascii "../test/exec.noff\000" # 用户地址空间  .text  .align 2 # 2 字节对齐，即 2\*2  .globl main # 全局变量  .ent main # main函数入口 main: # 汇编伪指令 frame 用来声明堆栈布局 # 该指令有三个参数： # （1）第一个参数 framereg: 声明用于访问局部堆栈的寄存器，一般为 $sp # （2）第二个参数 framesize: 声明该函数已分配堆栈的大小，符合 $sp+framesize = $sp # （3）第三个参数 returnreg: 这个寄存器用来保存返回地址 # $fp 为栈指针，该函数层栈大小为 32 字节，函数返回地址存放在 $31  .frame $fp,24,$31 # vars= 0, regs= 2/0, args= 16, extra= 0  .mask 0xc0000000,-4  .fmask 0x00000000,0  # 栈采用向下生长的方式，即由大地址向小地址生长，栈指针指向栈的最小地址  # $sp - 32 -> $sp，构造 main() 的栈 frame  # $sp 的原值应该是执行 main() 之前的栈  # 上一函数对应栈 frame 的顶（最小地址处）  subu $sp,$sp,24  sw $31,20($sp) # $31 -> memory[$sp+20]  sw $fp,16($sp) # $fp -> memory[$sp+16]  move $fp,$sp # $sp -> $fp，执行 Exec() 会修改 $sp  jal \_\_main # PC+4 -> $31，goto\_main  # $LCO -> $4，将 Exec("../test/halt.noff\000")的参数的地址传给$4  # $4 -> $7，传递函数的前四个参数给子程序，不够的用堆栈  la $4,$LC0  # 转到 start.s 中的 Exec 处执行  # PC+4 -> $31，goto Exec  # PC 是调用函数时的指令地址  # PC+4 是函数的下条指令地址，以便从函数返回时再调用  # 函数的下条指令开始继续执行原程序  jal Exec  jal Halt $L1: # $fp -> $sp  move $sp,$fp  # memory[$sp+20] -> $31, 取 main() 的返回值  lw $31,20($sp)  # memory[$sp+16] -> $fp，恢复 $fp  lw $fp,16($sp)  # $sp+24 -> $sp，释放 main() 对应的在栈中的 frame  addu $sp,$sp,24  # goto $31，main() 函数返回  j $31  .end main  因此我们可以从 **4 号寄存器**中获取参数在内存的地址，然后根据该地址读出该参数并执行，初代的系统中断处理函数如下（**不完整**）：  case SC\_Exec:{  int fileAddr = machine->ReadRegister(4);  char filename[50];  for (int i=0;;i++){  machine->ReadMem(fileAddr + i, 1, (int \*)&filename[i]);  if (filename[i] == '\0')  break;  }  printf("%s\n", filename);//输出文件名  interrupt->Halt();//停机  break; }  image-20211214203631304  image-20211214203631304  可以看到我们成功输出了系统调用的参数。还要注意在文件../machine/mipssim.cc 中处理系统调用模拟指令的操作是以 return 返回的：  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;  这意味着在执行完系统调用后**是否令程序计数器向前推进（PC 值会指向下一条指令所在地址）**的工作交给了对应的**系统调用处理函数ExceptionHandler(ExceptionType which)**去决定。所以你还应当在exception.cc准备一个函数 AdvancePC() 以便当系统调用成功后向前推进程序计数器：  void AdvancePC() {  machine->WriteRegister(PrevPCReg,machine->ReadRegister(PCReg));//前一个PC  machine->WriteRegister(PCReg, machine->ReadRegister(PCReg) + 4);//当前PC  machine->WriteRegister(NextPCReg, machine->ReadRegister(NextPCReg) + 4);//下一条PC，对应下一个执行的指令  }  下面我们就可以考虑怎样实现 Exec 系统调用的异常处理函数 Exec 了。由于**系统调用引发异常**属于一种中断处理，因此可以把这类处理函数都封装到Interrupt类中，作为Interrupt类的成员函数：  // 新线程执行用户进程初始化操作 void InitProcess(int spaceId) {  ASSERT(currentThread->space->getSpaceId() == spaceId);  currentThread->space->InitRegisters(); // 设置寄存器初值  currentThread->space->RestoreState(); // 加载页表寄存器  machine->Run(); // 运行  ASSERT(FALSE); }  // 硬件中断处理 void Interrupt::Exec() {  int fileAddr = machine->ReadRegister(4);  char filename[50];  for (int i=0;;i++){  machine->ReadMem(fileAddr + i, 1, (int \*)&filename[i]);  if (filename[i] == '\0')  break;  }  OpenFile \*executable = fileSystem->Open(filename);  if(executable == NULL) {  printf("Unable to open file %s\n",filename);  return;  }  printf("Exec(%s)\n",filename);  AddrSpace \*space = new AddrSpace(executable);// 建立新地址空间  space->Print(); // 输出新分配的地址空间  delete executable; // 关闭文件   Thread \*thread = new Thread(filename); // 建立新核心线程  thread->space = space; // 将用户进程映射到核心线程上   thread->Fork(InitProcess,(int)space->getSpaceId());  machine->WriteRegister(2,space->getSpaceId());// 返回地址空间标识符   currentThread->Yield();// 当前线程放弃CPU 切换到新的线程 }  **值得注意的是，当创建新的线程并且将其加入到等待队列后要将当前线程下CPU，否则执行无法调度新线程到RUNNING态，整个系统会卡死。**  当异常的类型是系统调用异常且系统调用类型是Exec时，调用interrupt->Exec()进行中断处理。  void ExceptionHandler(ExceptionType which) {  int type = machine->ReadRegister(2);   if (which == SyscallException) {  switch (type) {  case SC\_Halt:{  DEBUG('a', "Shutdown, initiated by user program.\n");  interrupt->Halt();  break;  }  case SC\_Exec:{  DEBUG('a', "Shutdown, initiated by user program.\n");  interrupt->Exec();// 触发硬件中断  AdvancePC();// 程序计数器向前推进  break;  }  default:{  printf("Unexpected user mode exception %d %d\n", which, type);  ASSERT(FALSE);  }   }  } else {  printf("Unexpected user mode exception %d %d\n", which, type);  ASSERT(FALSE);  } } |
| 结论分析与体会：  运行exec.noff后实验结果如图：  image-20211215102008456  可以看到系统在执行exec.noff可执行文件的同时执行了halt.noff  执行halt.noff的步骤是通过系统调用Exec()实现的  反思：   * **如何理解用户进程如何映射到一个核心线程** * 在 progtest.cc中的StartProcess 函数，在为用户程序初始化了地址空间之后，执行一条命令 currentThread->space = space，此命令即将用户进程映射到了核心线程之上。 * **如何理解当前进程的页表是如何与CPU使用的页表进行关联的** * 在创建用户进程的地址空间时，创建用户进程的页表项 * pageTable = new TranslationEntry[numPages];// 初始化页表数组 for (i = 0; i < numPages; i++) {// 对每个页表初始化  pageTable[i].virtualPage = i; //目前为止，物理地址等于虚拟逻辑地址  pageTable[i].physicalPage = i;  pageTable[i].valid = TRUE;  pageTable[i].use = FALSE;  pageTable[i].dirty = FALSE;  pageTable[i].readOnly = FALSE;  } * 创建完页表之后，回到 StartProcess 函数中，调用了 space->RestoreState() 命令，该命令将用户进程的页表赋值给了 Machine 页表，具体代码如下所示： * void AddrSpace::RestoreState()  {  machine->pageTable = pageTable;  machine->pageTableSize = numPages; } * 之后程序运行的过程就是通过 PC 寄存器中的虚拟地址通过 Machine 中的页表转化为物理地址，然后在将根据指令类型执行该指令。 * **如何实现内存页面分配** * 实现多进程机制比较关键的问题就是**页**的分配问题，**Nachos默认机制下的内存分配是每个进程的虚拟逻辑地址与实际物理地址相同**，且每个进程都是从0开始分配，这样如果有多道用户程序，内存中就会来回被覆盖： * **如何实现多道用户程序** * 在我们完成了多个程序同时驻留内存的内存分配算法后，我们就应当考虑用户父子进程并发执行的问题了。为了完成这一功能需要实现Exec系统调用的异常处理函数。 * **当创建新的线程并且将其加入到等待队列后要将当前线程下CPU，否则执行无法调度新线程到RUNNING态，整个系统会卡死** |