山东大学 软件 学院

**操作系统课程设计** 实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202022300316 | 姓名：王丁 | | 班级：软工20.6 |
| 学号：202000300358 | 姓名：梁思睿 | | 班级：软工20.6 |
| 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：2022.xx.xx | |
| 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：  设备名称 LAPTOP-UUHJO71O  处理器　　AMD Ryzen 5 4600U with Radeon Graphics 2.10 GHz  机带 RAM 16.0 GB (15.4 GB 可用)  设备 ID 63412E57-FC9E-4F92-A869-B35FDEF3303C  产品 ID 00342-35837-59360-AAOEM  系统类型 64 位操作系统, 基于 x64 的处理器  硬盘 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：  **概述**  这次实验涉及系统调用，难度大而且代码跨度大。不仅要阅读nachos内核c++代码，还需要阅读、更改MIPS汇编，以实现新的系统调用。不仅要更改现有的页表分派方式，还需要时刻考虑nachos执行用户程序的方式，不能疏漏任何一个系统调用的细节。  **Nachos用户程序分析**  首先说一下 .noff 文件的特征。 .noff 文件以一个“魔法数”0xbadfad开头，暗示这个文件是一个 .noff 文件。Noff 格式主要定义了 Nachos 系统中用户可执行文件的段表格式以及段的类型。Nachos系统将根据这一定义将 Noff 格式的文件加载到 Nachos的用户内存解释执行。  要想产生 .noff 文件，只需要在用户程序源代码文件夹（code/test）下make一下就行。目录中的 Makefile 文件声明将用coff2noff 和 coff2float 程序转换 Coff 格式的 MIPS 可执行文件到 Noff和 flat 格式的可执行文件 make 完成后 test/目录中将产生与.c 文件相对应的一系列.noff 和.flat 文件。如果写了一些原来目录中没有的文件，编译时需要更改makefile，把新的程序放到targets中。   1. 首先，我们需要明确AddressSpace的作用和机制。   类 AddrSpace 定义的是用户程序的地址空间。其中的几个主要方法：AddrSpace()：根据打开的用户可执行文件构造用户内存空间。  InitRegisters()：初始化用户 CPU 寄存器。  SaveState()：保存用户空间现场。  RestoreState()：恢复用户空间现场。  我们主要要做的是读懂 AddrSpace()函数，这个函数前半部分是在计算地址空间大小，我们重点要关注其后半部分，以知道为什么现在的 Nachos 不能支持多到用户程序：      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;      }  在原始的AddressSpace当中，页表的分配实际上形同虚设，因为虚拟页面和物理页面实际上是同一编号的页面，这也导致原有的nachos必然不支持多到用户程序，因为如果这样分配内存空间，原有的程序在内存中的位置会被替换掉。  我们需要想办法更改nachos的内存分配方式。根据实验指导书的提示，我们可以使用 bitmap 来改进 Nachos 现有的内存分配方法。通过回顾实验4可以知道，Bitmap本来是磁盘空间管理的工具类。但是由于磁盘的sector和内存的页大小相同，所以我们可以巧妙的利用这个类来管理、记录已经分配的内存空间。通过 bitmap 申请物理帧，就能让不同的程序能够装入到不同的物理空间中。   1. 阅读TranslationEntry 类   通过学习，我们已经知道基本的 Nachos 系统采用页式内存分配方式管理用户内存空间，刚分析的AddrSpace中的一个指向TranslationEntry 类的指针pageTable 给出了页表数组的起始地址。Nachos 系统中的页表结构是由TranslationEntry 类定义的：  class TranslationEntry {    public:      int virtualPage;    // The page number in virtual memory.      int physicalPage;   // The page number in real memory      bool valid;         // set when translation is ignored.      bool readOnly;      bool use;           // set every time page is referenced or modified.      bool dirty;         // set when modified.  };  其中 virtualPage 是用户逻辑页号，physicalPage 是对应的物理块号。valid 指示该页是否有效，readOnly 指示该页是否可写。use 是引用位，dirty 是改写位，利用这两位信息可以构造虚拟内存。use 和 dirty 位将在实验7中用到。   1. 阅读StartProcess函数   在main程序发起之后，在本次实验中，会调用StartProcess()方法。本方法将会读取一个.noff文件并且作为可执行程序运行。在运行之前，会分配寄存器和内存页。  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 progam      ASSERT(FALSE);   // machine->Run never returns; the address space  // exits by doing the syscall "exit"  }   1. Nachos系统调用机制   那么，nachos是怎么实现系统调用的呢？通过阅读源码、阅读实验指导书可以得知，nachos系统调用首先在syscal.h中定义，并且用一个数字进行指代。通过向寄存器传递参数，就能分辨程序使用了哪个系统调用。  #define SC\_Close    8  #define SC\_Fork     9  ... ...  void Halt();  void Exit(int status);  typedef int SpaceId;  SpaceId Exec(char \*name);  ... ...  然后，这些函数实际上并没有C语言实现，而是在MIPS汇编文件：start.s中编写。start.s会在程序编译时与用户程序进行链接。例如 C 程序 halt.c 被编译为 haltt.o，同时 start.s 也被汇编为 start.o。之后两个目标模块被连接成可执行的 Coff 格式的可执行文件，最后这个 Coff 文件又被转换为 Noff 格式的 Nachos 可执行文件。      .globl \_\_start      .ent    \_\_start  \_\_start:      jal main      move    $4,$0      jal Exit     /\* if we return from main, exit(0) \*/      .end \_\_start      .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  但是这些也只是把信息放在了寄存器当中呀，那么最终是谁来处理这些系统调用的呢？答案是中断处理模块。每个系统调用都会产生一个中断，在中断处理模块（interupt类）当中，就蕴含着处理系统调用的逻辑。  exception.cc  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);      }  但是这个中断处理逻辑还不完全，目前仅仅支持halt一个系统调用。我们要做的，就是补充中断处理逻辑，以支持更多的系统调用。而且根据这个代码风格可以看出，不能直接在这个函数里写中断处理的逻辑，而是需要封装在interupt当中。所以，如果需要写新的系统调用，主要的处理逻辑应该在interupt内新加函数来实现。  那么万事俱备，开始实现Exec()吧。  **实验内容1: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，以实现nachos的多到用户程序。**  要想完成这个实验，首先需要阅读实验指导书和nachos手册，通过阅读这些文档，我得知了最好定义一个Print函数用来打印页表，还需要定义一个AdvancePC函数来使得程序计数器向前推进。这两个工具函数写好之后，我们再来添加exec逻辑。  按照上面的分析，首先需要在interupt.h, interupt.cc, syscall.h, start.s中定义Exec().具体如下：  interupt.h  void Exec(char\* filename);  // run a user program  interupt.cc 这里是Exec的真正逻辑  void Interrupt::Exec(char\* filename) {      OpenFile\* executable = fileSystem->Open(filename);      // printf("$$$$$$$$ In Exec\n");      printf("Exec filename: %s\n", filename);      if (executable == NULL) {          printf("Unable to open file %s\n", filename);          return;      }      space = new AddrSpace(executable);      delete executable;  // close file      thread = new Thread("another thread");      // printf("$$$$$$$$ another thread created\n");      thread->Fork(StartProcess, 1);      machine->WriteRegister(2, space->getSpaceId());      currentThread->Yield();  }  （一些打印信息用来辅助debug）  syscall.h  typedef int SpaceId;  SpaceId Exec(char \*name);  start.s      .globl Exec      .ent    Exec  Exec:      addiu $2,$0,SC\_Exec      syscall      j   $31      .end Exec  然后，在exception.cc的异常处理逻辑中，加上Exec 的处理。  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 if ((which == SyscallException) && (type == SC\_Exec)) {          DEBUG('a', "Exec, initiated by user program.\n");          char filename[100];          int addr = machine->ReadRegister(4);          int i = 0;          machine->ReadMem(addr + i, 1, (int\*)&filename[i]);          while (filename[i] != '\0') {              i++;              machine->ReadMem(addr + i, 1, (int\*)&filename[i]);          }          printf("----before exec ----\n");          interrupt->Exec(filename);          printf("----after exec  ----\n");          AdvancePC();      }  另外，我们还需要更改AddressSpace的内存分配方式。  在头文件中，我定义了两个全局变量：  extern bool\* spaceIds;  extern int spaceIdCount;  用来指示现在已经有了的space序号。在AddrSpace当中，我定义了一个int成员，用来存放自己的spaceId.  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);      int codePageNumber = divRoundUp(noffH.code.size, PageSize);      int initDataPageNumber = divRoundUp(noffH.initData.size, PageSize);  size = codePageNumber \* PageSize + initDataPageNumber \* PageSize +             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      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;          pageTable[i].physicalPage = memoryMap->Find();// 支持多道程序          pageTable[i].valid = TRUE;          pageTable[i].use = FALSE;          pageTable[i].dirty = FALSE;          pageTable[i].readOnly = FALSE;      }  if (noffH.initData.size > 0) {          DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",                noffH.initData.virtualAddr, noffH.initData.size);          for (int i = 0; i < initDataPageNumber; i++) {              executable->ReadAt(                  &(machine->mainMemory[pageTable[codePageNumber  + i].physicalPage \* PageSize]),  // 这里假设分配到的物理内存不连续，然后分别赋予读进数据。                  PageSize, noffH.initData.inFileAddr + i \* PageSize);          }      }      Print();// 看下分配情况      mySpaceId = spaceIdCount++;      if (mySpaceId >= MAX\_THREAD\_COUNT) {          // set printf color to red          printf("\033[31m");          printf("Too many threads!\n");          printf("Please increase MAX\_THREAD\_COUNT in addrspace.h\n");          printf("machine halt\n");          // set printf color to white          printf("\033[0m");          ASSERT(false);      }      spaceIds[spaceIdCount] = true;  // 这个bool数组用来指示已经使用了的space Id.      printf("mySpaceId: %d\n", mySpaceId);  }  然后我们还需要一个程序用来引导。也就是，初始化寄存器和存储状态。  void StartProcess(int n) {      currentThread->space = space;      currentThread->space->InitRegisters();      currentThread->space->RestoreState();      // printf("$$$$$$$$ In another thread\n");      machine->Run();      ASSERT(FALSE);  }  最后，测试一下看看。    程序正常工作，大功告成。  **实验内容2: Nachos实现新的系统调用：PrintInt()，在用户程序中打印一个整数值。**  这里，实际上就是再加一个系统调用。有了上面exec的经验，加一个PrintInt实际上非常简单。话不多说直接上代码。  首先在start.s 里面加入汇编引导程序:      .globl PrintInt      .ent    PrintInt  PrintInt:      addiu $2,$0,SC\_PrintInt      syscall      j   $31      .end PrintInt  然后再在syscall、interupt里面注册：  void PrintInt(int n);  在interupt.cc里面加入具体实现：  void Interrupt::PrintInt(int n) {      printf("----- PrintInt: %d -----\n", n);  }  稍微更改一下execption，在ExceptionHandler里面添加PrintInt中断处理逻辑：  if ((which == SyscallException) && (type == SC\_Halt)) {          DEBUG('a', "Shutdown, initiated by user program.\n");          interrupt->Halt();      } else if ((which == SyscallException) && (type == SC\_Exec)) {         ... ... 这里是exec处理逻辑      } else if ((which == SyscallException) && (type == SC\_PrintInt)) {          DEBUG('a', "Exit, initiated by user program.\n");          interrupt->PrintInt(machine->ReadRegister(4));          printf("reg4: %d\n", machine->ReadRegister(4));          AdvancePC();      } else {          printf("Unexpected user mode exception %d %d\n", which, type);          ASSERT(FALSE);  }  大功告成。现在修改一下测试程序exec.c：  int main() {      SpaceId pid;      PrintInt(12345);      pid = Exec("../test/halt.noff");      PrintInt(114514);   // This line should not be printed      Halt();      /\* not reached \*/  }  效果：    新的系统调用PrintInt成功。  **实验内容3: 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。**  要想实现这些功能，首先需要了解什么是copy-on-write。  OS 领域 copy-on-write 核心思想是 lazy copy。也就是说，只要不进行修改，在形式上进行复制的时候，实际上可以利用同一份物理内存空间。  传统的fork()函数是这样进行的：创建一个子进程，子进程和父进程共享代码段，复制数据段、堆、栈到子进程。    Linux的fork()函数则是写时复制(copy-on-write)。创建一个子进程，内核只为子进程创建虚拟空间，不分配物理内存，和父进程共享物理空间，当父进程中有更改相应段的行为发生时，才为子进程分配物理空间。  根据以上知识，我们可以推断在nachos实现copy-on-write的fork和exec的思路。   1. copy-on-write的fork     传统的fork函数直接把所有资源复制给新的进程，效率很低下。写时拷贝在需要写入时，数据才会被复制，没有数据写入时，fork()的开销实际只是复制父进程的页表以及给子进程创建唯一的进程描述符。有数据要写入前，会将将要改变的数据页复制给子进程。  那么就要好好想想fork怎么和copy-on-write结合。仔细思考不难发现，fork的情况下，代码段实际上是不会被更改的，会被更改的只是数据段。因此，将代码段和数据段分别装在不同的分页里面就比较有利于区分处理，而这一点我们已经在AddrSpace的构造函数里面实现。  在fork的时候，显然代码段是不会被修改的，数据段则不一定，但是也有很多不用修改的情况。那么，fork之后我们只需要新建一个新的页表，页表的物理页还是指向原来的物理页；当程序试图修改原来的页面的时候，我们再进行复制即可。  问题是我怎么知道什么时候修改了共享的页面呢？一种可能的思路是借助中断来实现。观察页表不难发现，有一个readonly位。我们可以在共享页面之后，把共享的页面标记为readonly，每当试图修改共享的页面的时候，就产生一个中断；在这个中断的处理中，加上复制页面的逻辑。  但是另一方面，也不能每次产生readOnlyException都复制一个新的页面。万一这个页面本来就是只读，而且不希望被他人共享呢？所以我的想法是在页表中加一个copyOnWrite位，用来指示这个页表管理的页面究竟是不是copyOnWrite机制下的共享页面。如果是，那么产生readOnlyException的时候附加上这个信息，方便系统进行页面的复制。  那么fork的copyOnWrite思路就是这样。   1. copy-on-write的exec   exec听起来好像和copy-on-write不沾边，毕竟exec执行的是另外一个程序，它怎么会涉及到复制呢。  让我们回想一下Linux的exec是什么策略。Linux的exec不像我们在nachos中刚刚实现的exec；它唤起的新的程序实际上是替代了原来的程序的空间。我们在上学期做OS实验2的时候，曾经用过这个函数来执行另一个程序，当时它的表现让我印象深刻，在执行完另一个程序之后，原来的进程就不能用了，因为它的物理空间已经被覆盖，这有些出乎意料。  那么这里也可以把exec的逻辑改一改。我们**不**分配新的物理内存、保留原有的进程空间，而是直接进行覆盖！这样一来，每次需要执行一个新的程序、但是需要保留原来的程序的时候，就需要先fork然后紧接着exec。  实际上，这也是合乎情理和实践的：通过上学期的实验不难知道，在实际的Linux系统中，我们的确就是这么干的。这样修改exec逻辑之后，它就能和copy-on-write的fork结合，尽力不在内存复制上做无用功。  我在nachos系统中实现fork和exec的copy-on-write思路就是这样。 | | | |
| 结论分析与体会：  这次试验花费了大量的时间，但是也收获了很多。学习了各种系统调用，以及 Nachos 是如何模拟硬件进行的系统调用等操作的，通过查询资料和反复阅读，对汇编部分的代码也能有一定的理解了。最重要的是我彻底弄明白了系统调用的执行过程，以及各个类的作用分别是什么。  系统调用和之前函数调用的实现思路可以说是非常不一样了。在上学期学过操作系统之后，我自以为已经很了解系统调用究竟是什么了；但是这次做了两个nachos系统调用之后才发觉还是“纸上得来终觉浅，绝知此事要躬行”。  通过阅读Linux的fork和exec源码、查阅网上资料，我也了解了copy-on-write机制，并且对于程序的执行和内存的复制以及中断机制带来的便利有了更深刻的认识，并且得出了nachos实现copy-on-write的大概思路。我想，这些工作对于下一个实验或许也有不少作用吧。 | | | |