山东大学 软件 学院

**操作系统课程设计** 实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202000300125 | 姓名：贾星宇 | | 班级：计软20.5 |
| 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年11月14日 | |
| 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：  HUAWEI matebook14 2020笔记本  Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz 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 16.2.4 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：  根据各个实验的内容，以及实现的过程，可写的包括但不限于：  解决问题的思路；  实现步骤；  关键源代码及注释(尽量以文本格式，且必须与提交的源代码一致)；  程序说明，特别是自己认为的精彩之处；  调试过程及记录；  运行结果(可文本格式，必要时抓屏)；  等等…   1. （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。   多道用户程序指的是CPU一次读取多个程序放入内存，先运行第一个程序直到它出现了IO操作。如果系统不是多道程序，那么当我们在运行一个用户程序时就无法运行其他用户程序，那么当这个用户程序去占用IO等操作而不使用CPU时，CPU利用率会因此降低。因此，我们采用多道程序，让多个用户程序同时运行，如果一个程序不使用CPU了，那么另一个程序会占用CPU，大大提高了CPU的使用率。  观察当前程序，如果要运行一个用户程序，我们只能等待系统执行完指令后，再利用命令行运行下一个程序，因此为了实现多道用户程序，我们需要新增系统调用Exec。  在实现之前，我们首先实现一个Print函数来打印当前用户程序的内存占用情况，也就是当前用户程序的虚拟地址和实际物理地址的对应情况：    AddrSpace类负责为用户程序分配地址空间，因此，我们在AddrSpace类构造函数结束时调用此Print函数，打印出空间的分配情况：    （2）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。  我们从程序运行过程的角度分析如何实现系统调用。通过readme文件可以知道，我们的目的是运行exec.c所编译链接生成的exec.noff文件：  ./nachos -x ../test/exec.noff  因此，我们前往main函数查看-x指令调用了哪个方法：    随后我们前往StartProcesss方法：    在此方法中，首先利用AddrSpace类构造处相关用户程序文件的地址空间，随后进行初始化操作，并运行Run函数依次调用指令执行用户程序。因此我们修改的重点是分配地址空间的构造函数和运行时需要调用的函数。  AddrSpace构造函数：  在原构造函数中，仅是简单的计算出所需页的数量并分配到相应的物理块上，而现在由于我们要实现多道程序，因此我们需要记录每道程序所占用的物理块，并保证新的分配不会分配到已经占用的物理块。这时，我想到了先前文件系统中用到的类BitMap，在这个类中我可以定义一个指定长度的字符流，其中第n位为1表示占用。因此我可以构建一个长度为最大物理块大小的bitmap，并在分配物理内存时用Find方法分配，这样便做到了不冲突地分配物理地址。同时，在后续读取代码和数据段的地址也要做相应的修改，详细代码如下：   |  | | --- | | BitMap\* AddrSpace::bitmap = new BitMap(NumPhysPages);  AddrSpace::AddrSpace(OpenFile \*executable)  {  bool flag = false;  for (int i = 0; i < 128; i++)  {  if(!ThreadMap[i]){  ThreadMap[i] = 1;  flag = true;  spaceID = i;  break;  }  }  ASSERT(flag);  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 = bitmap->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  }  // zero out the entire address space, to zero the unitialized 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);  int codepage = noffH.code.virtualAddr / PageSize;  int codeOffset = noffH.code.virtualAddr % PageSize;  int codePhyAdd = pageTable[codepage].physicalPage \* PageSize + codeOffset;  }  if (noffH.initData.size > 0)  {  DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",  noffH.initData.virtualAddr, noffH.initData.size);  int datapage = noffH.initData.virtualAddr / PageSize;  int dataOffset = noffH.initData.virtualAddr % PageSize;  int dataPhyAdd = pageTable[datapage].physicalPage \*PageSize + dataOffset;  executable->ReadAt(&(machine->mainMemory[dataPhyAdd]),  noffH.initData.size, noffH.initData.inFileAddr);  }  Print();  } |   构造Exec系统调用：  可以看到，在syscal.h和start.s文件中皆已经写好了Exec，那么我们只需要首先在interrupt中编写Exec函数，随后仿照exception中的ExceptionHandler来调用interrupt中的Exec函数即可。调用方式：    Exec函数：此函数主要功能为新创建一块地址空间，并创建新线程然后在新地址空间里运行即可。创建地址空间并Run用户进程的过程我们可以仿照StartProcess方法进行书写，在创建结束需要运行时，我们根据实验二的经验来实现，具体代码如下：   |  | | --- | | Thread \*thread;  AddrSpace \*space;  int Interrupt::Exec()  {  printf("Execute system call of Exec() \n");  char filename[50];  int addr = machine->ReadRegister(4);  int i = 0;  do  {  machine->ReadMem(addr + i, 1, (int \*)&filename[i]);  } while (filename[i++] != '\0');  printf("Exec(%s):\n", filename);  OpenFile \*executable = fileSystem->Open(filename);  if (executable == NULL)  {  printf("Unable to open file %s\n", filename);  return 0;  }  space = new AddrSpace(executable);  delete executable;  thread = new Thread("forked thread");  scheduler->ReadyToRun(currentThread);  currentThread=thread;  thread->space=space;  space->InitRegisters();  space->RestoreState();  machine->WriteRegister(2, space->getSpaceID());  machine->Run();  ASSERT(FALSE);  AdvancePC();  } |  1. 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。   我们先前已经增加过系统调用Exec，因此我们仿照上述方式来增加PrintInt：  首先在syscal.h和start.s中仿照已有系统调用新增PrintInt：  Syscal.h:      Start.s:    随后我们前往ExceptionHandler添加一个if分支：    随后在interrupt类中添加方法PrintInt即可：    效果检验：make后输入指令./nachos -x ../test/exec.noff，打印效果如下：   |  | | --- | | u1@ubuntu:~/oscp/nachos-3.4-ualr-2022/code/lab6$ ./nachos -x ../test/exec.noff  spaceID: 0  page table dump: 11 pages in total  ============================================  VirtPage, PhysPage  0, 0  1, 1  2, 2  3, 3  4, 4  5, 5  6, 6  7, 7  8, 8  9, 9  10, 10  ============================================  12345  Execute system call of Exec()  Exec(../test/halt2.noff):  spaceID: 1  page table dump: 11 pages in total  ============================================  VirtPage, PhysPage  0, 11  1, 12  2, 13  3, 14  4, 15  5, 16  6, 17  7, 18  8, 19  9, 20  10, 21  ============================================  67890  Machine halting!  Ticks: total 48, idle 0, system 10, user 38  Disk I/O: reads 0, writes 0  Console I/O: reads 0, writes 0  Paging: faults 0  Network I/O: packets received 0, sent 0  Cleaning up... |  1. 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)。   首先，实现系统调用要修改syscal.h和start.s两个文件，可以看到这两个文件中已经存在Fork与exec。因此我们只需要现在exception类中添加if循环分支，随后前往interrupt类中构建所需要的方法。  传统的Fork函数和exec函数都是通过父进程生成子进程，区别在于fork函数会开辟新的地址空间并赋值父进程地址空间，而exec是装载一个新的进程覆盖当前进程的地址空间。  通常情况下，通过fork创建子进程后，子进程会通过exec函数覆盖当前地址空间并执行新的进程，那么我们就没有必要先前fork时复制一份父进程地址空间给子进程。因此，便出现了写时复制的方法，也就是说fork创建出子进程之后与父进程使用相同的地址空间，当父进程或者子进程有更改相应段的行为发生时，再为子进程分配地址空间。  我们来考虑写时复制在Nachos上的实现方式。当我们fork创建子进程时，我们不需要为新的进程初始化地址空间，而是获取当前进程的addrspace并赋予给子进程的space。但是，我们需要在AddrSpace类中增加监听方法。我们可以看到，当调用SaveState、RestoreState等方法时，说明父进程地址空间要做出修改或者子进程要更新自己的地址空间了，这时候我们新增一个allocateAddrSpace方法为子进程调用exec方法赋予内存空间。同时，我们也需要判断是父进程做出改变还是子进程，这个可以通过进程号或者地址空间号来判断，如果父进程地址空间要做出改变，那么子进程初始化的地址空间是父进程的，如果子进程要改变，则调用exec构造子进程要求构造地址空间的和用户程序。 | | | |
| 结论分析与体会：  通过实验六的设计，我对用户程序的运行及实现方式、多道用户程序的实现方式、虚拟内存的原理以及写时复制等操作系统的经典方法有了深入的理解，同时也拥有了更好的编程技能和代码阅读能力，对代码的封装性也有了更进一步的认识。  同时，我也深刻的认识到如果要解决一个问题，我们可以从问题的源头走起，一步步按照程序的执行方式来解决需要解决的问题。 | | | |