|  |  |
| --- | --- |
| 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：2022.12.1 |
| 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：  联想小新Air 15ALC 2021  设备名称 LAPTOP-OPORL5Q4  处理器 AMD Ryzen 7 5700U with Radeon Graphics 1.80 GHz  机带 RAM 16.0 GB (13.9 GB 可用)  硬盘 1TB SSD | |
| 软件环境：  宿主机：Windows 11 家庭中文版 21H2 22000.1219  虚拟机软件：VMware® Workstation 16 Pro 16.0.0 build-16894299  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： 一、扩展class AddrSpace1.1 解决问题的思路 实验要求扩展AddrSpace使得可以实现多道用户程序，也就是说本来的nachos不支持多道用户程序，问题出在AddrSpace里，推测是由于没有页表功能，每个进程的空间都是从头开始分配物理空间，导致反复覆盖导致的。我们需要查看nachos原本为用户进程分配空间的逻辑，对此加以修改，让多个程序可以同时驻留内存。  首先查看nachos原来是如何分配空间的：  本程序的测试命令行为./nachos -x (用户程序位置)，也就是在progtest.cc中有处理执行用户程序的相关逻辑。查看代码，逻辑在progtest.cc的StartProccess中：   1. space = new AddrSpace(executable); 2. currentThread->space = space;   可见，以上程序中space是用户程序的地址空间，初始化后将其映射到了核心线程上，完成了将用户程序映射到核心线程上的操作。  那么要想实现多道用户程序，就应该修改用户程序地址空间的初始化部分，查看userprog/addrspace.cc中构造方法：   1. size = noffH.code.size + noffH.initData.size + noffH.uninitData.size + UserStackSize; *// we need to increase the size* 2. *// to leave room for the stack* 3. numPages = divRoundUp(size, PageSize); 4. size = numPages \* PageSize;   可见，其为用户程序分配的虚拟空间大小就是其代码空间+初始数据+非初始数据+栈空间，再由以下程序将虚拟页映射到物理页：   1. pageTable = new TranslationEntry[numPages]; 2. for (i = 0; i < numPages; i++) { 3. pageTable[i].virtualPage = i; *// for now, virtual page # = phys page #* 4. pageTable[i].physicalPage = i; 5. pageTable[i].valid = TRUE; 6. pageTable[i].use = FALSE; 7. pageTable[i].dirty = FALSE; 8. pageTable[i].readOnly = FALSE;  *// if the code segment was entirely on* 9. *// a separate page, we could set its* 10. *// pages to be read-only* 11. }   此时虚拟页号=物理页号，其实此处有虚拟页表，但没什么用处，每个新的地址空间都会覆盖原来地址空间的物理页，所以不能实现多道用户程序。 1.2 实现步骤与代码说明 根据分析，我们需要修改addrspace.cc中将虚拟页映射到物理页的逻辑，即构造方法中的部分逻辑。  1、将addrspace.h/.cc拷贝到lab6目录下。  2、考虑如何将不同的用户进程的虚拟页映射到不同的物理页上。逻辑其实很简单，新进程从未被分配的物理页开始占用即可。很容易想到nachos源码中自带的bitmap数据结构，用一个静态的bitmap对象管理物理页的占用情况即可。  将管理物理页占用情况的全局freeMap其添加到addrspace.h中：   1. private: 2. ... 3. static BitMap \*freeMap;   并在addrspace.cc中对其进行初始化：   1. BitMap \*AddrSpace::freeMap=new BitMap(NumPhysPages);   其中，NumPhysPages为machine/machine.h中的常数，默认为32，是nachos模拟的硬件的一部分，我们暂时不需要关心这个数字。  3、在构造方法中使用上述freeMap为每个虚拟页分配物理页，注意到原本下方还有将其空间置0的逻辑，将物理空间0~size的字节全部置0，在虚拟内存已经起作用的情况下就不能简单地如此做了，于是将其删除掉：   1. *//bzero(machine->mainMemory, size);*   然后在for循环中实现虚拟页映射到物理页的具体逻辑：   1. pageTable = new TranslationEntry[numPages]; 2. ASSERT(freeMap->NumClear()>=numPages);*//ensure to have enough physical pages* 3. for (i = 0; i < numPages; i++) { 4. pageTable[i].virtualPage = i; 5. pageTable[i].physicalPage = freeMap->Find();*//lab6: could be different from vPage* 6. pageTable[i].valid = TRUE; 7. pageTable[i].use = FALSE; 8. pageTable[i].dirty = FALSE; 9. pageTable[i].readOnly = FALSE;  *// if the code segment was entirely on* 10. *// a separate page, we could set its* 11. *// pages to be read-only* 12. bzero(&(machine->mainMemory[pageTable[i].physicalPage\*PageSize]),PageSize); 13. }   如上，只需要简单的freeMap->Find()方法就可以找到一个未被占用的物理页，就可以实现虚拟页到物理页的映射了。然后再利用bzero方法对每个映射到的物理页分别置0。  4、完成分配后，需要将二进制文件中的代码段和初始数据段写入到内存中。由于不清楚二进制文件的代码段/初始数据段的起始位置是否会正好是某个页的起始位置，也不知道具体的物理地址是否连续，我们就不能简单地从某个物理页的开始写入一整段内容，而是需要逐页写入对应的内容：   1. if (noffH.code.size > 0) { 2. Segment seg=noffH.code; 3. int startPos=seg.virtualAddr; 4. int endPos=startPos+seg.size; 5. int firstVPage=startPos/PageSize,lastVPage=divRoundUp(endPos,PageSize); 6. int curAddr=seg.inFileAddr;*//read from this addr of file* 7. for(int vPage=firstVPage;vPage<=lastVPage;vPage++){ 8. int pagePos=pageTable[vPage].physicalPage\*PageSize;*//physical pos* 9. if(vPage==firstVPage)pagePos+=startPos%PageSize; 10. int curSize=(vPage==lastVPage?(endPos%PageSize):PageSize)-(vPage==firstVPage?(startPos%PageSize):0);*//size* 11. executable->ReadAt(&machine->mainMemory[pagePos],curSize,curAddr); 12. curAddr+=curSize; 13. } 14. }   虽然代码看起来有些麻烦，但就是计算文件位置逐页写入对应物理位置的转换，逻辑其实很简单。对于初始数据的处理与上述代码一致，不再赘述。  5、进程结束后，地址空间释放的同时也应该释放freeMap中的资源，因此在AddrSpace的析构函数中添加擦除占用的物理页对应的位的逻辑：   1. for(int i=0;i<numPages;i++)freeMap->Clear(pageTable[i].physicalPage);  二、完成AddrSpace::Print()2.1 解决问题的思路 此函数用于打印当前地址空间的虚拟页对物理页的映射情况。最终实现效果应与示例程序一致：   1. Page table dump: 11 pages in total 2. ============================================ 3. VirtPage, PhysPage 4. 0,        0 5. 1,        1 6. 2,        2 7. 3,        3 8. 4,        4 9. 5,        5 10. 6,        6 11. 7,        7 12. 8,        8 13. 9,        9 14. 10,       10 15. ============================================   可见只是对当前AddrSpace对象某些状态的打印。 2.2 实验步骤与代码说明 1、参照示例程序的输出，编写AddrSpace::Print()函数的主要逻辑如下：   1. printf("page table dump: %d pages in total\n",numPages); 2. printf("============================================\n"); 3. printf("\tVirtPage, \tPhysPage\n"); 4. for(int i=0;i<numPages;i++) 5. printf("\t%d, \t\t%d\n",pageTable[i].virtualPage,pageTable[i].physicalPage); 6. printf("============================================\n");   然后为了让其在合适的位置进行输出，在AddrSpace类的构造方法中加入此方法：   1. Print();   应该就可以实现示例程序的效果了。当然也可以在进程分配地址空间后手动调用等。 三、实现Nachos系统调用：Exec()3.1 解决问题的思路 系统调用是用户程序与内核程序的边界之一，系统通过异常机制向内核提交请求，而系统调用（一种陷入）就是一种异常。操作系统进入内核的边界就是异常，异常中有中断、陷入、故障、终止等。Nachos中有一个程序负责处理所有异常：userprog/exception.cc：   1. void 2. ExceptionHandler(ExceptionType which) 3. { 4. int type = machine->ReadRegister(2); 5. if ((which == SyscallException) && (type == SC\_Halt)) { 6. DEBUG('a', "Shutdown, initiated by user program.\n"); 7. interrupt->Halt(); 8. } else { 9. printf("Unexpected user mode exception %d %d\n", which, type); 10. ASSERT(FALSE); 11. } 12. }   可见只有Halt()终止的处理程序，nachos将其视为一种SC\_Halt系统调用，而不是一种特殊的异常。由此处可知，系统调用的类型诸如SC\_Halt在userprog/syscall.h中维护：   1. #define SC\_Halt  0 2. #define SC\_Exit  1 3. #define SC\_Exec  2 4. #define SC\_Join  3 5. #define SC\_Create 4 6. #define SC\_Open  5 7. #define SC\_Read  6 8. #define SC\_Write 7 9. #define SC\_Close 8 10. #define SC\_Fork  9 11. #define SC\_Yield 10 12. #define SC\_PrintInt 11   由此处可见，Exec()类型的系统调用已经预留了，我们只需要在exception.cc的异常处理程序中加上对SC\_Exec系统调用的处理即可。  参考SC\_Halt系统调用的处理，其逻辑在interrupt类中，可见nachos认为来自软件的系统调用也是一种中断（interrupt）。类似地，我们可以在interrupt.cc中编写Exec()系统调用的逻辑。  但是Exec()相比Halt()需要多一个参数（可执行文件的路径），需要获取到这个参数才能进行逻辑的编写。  首先exception.cc中已经给我们提示了：   1. *//  For system calls, the following is the calling convention:* 2. *//* 3. *//  system call code -- r2* 4. *//  arg1 -- r4* 5. *//  arg2 -- r5* 6. *//  arg3 -- r6* 7. *//  arg4 -- r7* 8. *//* 9. *// The result of the system call, if any, must be put back into r2.*   系统调用的第一个参数在4号寄存器中。所以就像获取系统调用类型一样，直接machine->ReadRegister(4)就是文件的路径了。  然后starts.s中也有类似的提示：   1. \* System call stubs: 2. \* Assembly language assist to make system calls to the Nachos kernel. 3. \* There is one stub per system call, that places the code for the 4. \* system call into register r2, and leaves the arguments to the 5. \* system call alone (in other words, arg1 is in r4, arg2 is 6. \* in r5, arg3 is in r6, arg4 is in r7)   我看汇编的能力有限，也难以深究为什么是第4个寄存器存参数，然后返回值放到第2个寄存器了，照做就是了。  获取文件名后，就可以仿照progtest.cc中的方法创建新的用户进程了。  最后一个问题，就是如何创建新线程以将用户进程绑定。经队友提示，可以利用Thread类中的Fork()启动新的内核线程。此函数还需要一个线程号作为参数，所以在创建用户进程空间时需要再维护一个编号供其使用。 3.2 实验步骤与代码说明 1、维护地址空间的编号。  在AddrSpace类中添加类似进程号的成员变量spaceId：   1. private: 2. ... 3. int spaceId;   由于整个系统有最大进程数量，也需要使用位图数据结构对可用的进程号进行维护：   1. private: 2. ... 3. static BitMap \*spaceIdMap;   然后在addrspace.cc中对该位图进行初始化，参数为最大进程数：   1. BitMap \*AddrSpace::spaceIdMap=new BitMap(NumProcess);   其中最大进程数NumProcess是我在addrspace.h中自定义的常数，值为256：   1. #define NumProcess 256   在AddrSpace类的构造函数中，为其spaceId成员变量进行初始化：   1. *//allocate spaceId* 2. ASSERT(spaceIdMap->NumClear()>0); 3. spaceId=spaceIdMap->Find();   在Print()函数中添加spaceId的打印：   1. printf("SpaceId:%d\n",spaceId);   最后在地址空间销毁时，也就是在析构函数中释放该编号：   1. spaceIdMap->Clear(spaceId);   2、获取Exec()的参数，即文件路径。  将interrupt.cc/.h拷贝到lab6目录下。  添加方法Interrupt::Exec()，在其中添加以下逻辑：   1. int fileAddr=machine->ReadRegister(4); 2. char filename[50]; 3. *//read in the filename from memory* 4. int i=0; 5. machine->ReadMem(fileAddr,1,(int\*)&filename[0]); 6. while(filename[i++]!='\0') 7. machine->ReadMem(fileAddr+i,1,(int\*)&filename[i]);   经过测试，由于是字符串类型，第四个寄存器存放的是文件路径的地址。需要利用c++的字符串机制从该地址开始读到’\0’结束，将这一段内容存入filename，即可获得文件路径。  3、编写创建用户进程的逻辑。  初始化一个线程控制块，以便稍后进行Fork()为其分配栈空间。  仿照progtest.cc中的逻辑，创建地址空间并将其绑定到内核线程，并将其页表关联到CPU页表：   1. OpenFile \*executable=fileSystem->Open(filename); 2. if(executable==NULL){ 3. printf("Unable to open file %s\n",filename); 4. return; 5. } 6. AddrSpace \*space=new AddrSpace(executable);*//allocate new addrspace* 7. delete executable;*//close file* 8. Thread \*thread=new Thread(filename);*//new kernel thread* 9. thread->space=space;*//user thread map to kernel thread* 10. thread->space->InitRegisters(); 11. thread->space->RestoreState();   4、编写执行该进程的逻辑。  首先创建一个令Fork()执行的方法：   1. void 2. InitProcess(int spaceId){ 3. machine->Run();*//jump to it* 4. ASSERT(false); 5. }   将此函数作为Fork()的参数，执行其中方法时currentThread就是执行Fork()的线程，而Fork()可以分配栈空间，关中断，将进程加入到就绪队列，恢复中断状态值。此处的machine->Run()是跳转到currentThread的空间并转为用户模式，这样就将新的用户进程变为了就绪态。  然后在Interrupt::Exec()中编写：   1. printf("Exec(%s):\n",filename); 2. thread->Fork(InitProcess,space->GetSpaceId()); 3. machine->WriteRegister(2,space->GetSpaceId());*//return spaceid to reg2* 4. currentThread->Yield();*//release cpu to next thread*   其中还输出了执行信息、并将进程号写回寄存器2号。最后释放cpu执行新的线程，将现在的线程变为就绪态，否则新的线程只会在就绪队列。  5、在异常处理函数中调用该方法。  将exception.cc拷贝到lab6目录下。对其中的ExceptionHandler()方法进行修改：   1. if (which == SyscallException) { 2. switch(type){ 3. case SC\_Halt:{ 4. DEBUG('a', "Shutdown, initiated by user program.\n"); 5. interrupt->Halt(); 6. break; 7. } 8. case SC\_Exec:{ 9. printf("Executing SysCall Exec()\n"); 10. interrupt->Exec(); 11. IncrementPC(); 12. break; 13. } 14. default:{ 15. printf("Unexpected user mode exception %d %d\n", which, type); 16. ASSERT(FALSE); 17. } 18. } 19. } else { 20. printf("Unexpected user mode exception %d %d\n", which, type); 21. ASSERT(FALSE); 22. }   调用中断处理函数后，还调用了自定义的IncrementPC()方法。是看到了异常处理函数上方的注释：   1. *// And don't forget to increment the pc before returning. (Or else you'll* 2. *// loop making the same system call forever!*   而且根据machine/mipssim.cc中对于系统调用的处理，也应该手动前移程序计数器：   1. case OP\_SYSCALL: 2. RaiseException(SyscallException, 0); 3. return; 4. ... 5. registers[PrevPCReg] = registers[PCReg]; *// for debugging, in case we* 6. *// are jumping into lala-land* 7. registers[PCReg] = registers[NextPCReg]; 8. registers[NextPCReg] = pcAfter;   由以上代码可以得出，程序在模拟完系统调用之后直接return了，没有机会执行下面的PC+1逻辑，也就是说是否有下一条指令需由具体的系统调用决定。此逻辑最后的pcAfter是mipssim.cc中的一个临时变量，计算出下一个PC指向的地址实际上应该是+4：   1. int pcAfter = registers[NextPCReg] + 4;   那么就可以将以上代码的PC+1逻辑整合到exception.cc中作为一个函数：   1. void IncrementPC(){ 2. machine->WriteRegister(PrevPCReg,machine->ReadRegister(PCReg)); 3. machine->WriteRegister(PCReg,machine->ReadRegister(NextPCReg)); 4. machine->WriteRegister(NextPCReg,machine->ReadRegister(NextPCReg)+4); 5. }   这样在处理系统调用时，就可以根据需要调用该方法，使PC+1。 四、增加新的系统调用PrintInt()4.1 解决问题的思路 和Exec()系统调用类似，增加一个新的调用即可。而且他们被调用的格式完全一致，见test/exec.c：   1. PrintInt(12345); 2. pid = Exec("../test/halt2.noff");   因为系统调用的入口、参数位置等都知道了，逻辑也很简单，只需要一个printf就可以。所以在异常处理程序里加上此种系统调用类型的处理逻辑即可。  主要问题是系统默认没有PrintInt类型的系统调用，需要修改系统某个部分的常量等参数，加上此种系统调用。根据实验指导书提示，可以原位修改syscall.h和starts.s。那么只要“入乡随俗”地加上这一种调用即可。  userprog/syscall.h：   1. #define SC\_Halt  0 2. #define SC\_Exit  1 3. #define SC\_Exec  2 4. #define SC\_Join  3 5. ...   此处即定义系统调用类型的地方，在这个地方加以修改即可。  test/starts.s：   1. ... 2. .globl Exec 3. .ent Exec 4. Exec: 5. addiu $2,$0,SC\_Exec 6. syscall 7. j $31 8. .end Exec 9. ...   应该是各种系统调用的入口，仿照这个格式添加即可。 4.2 实验步骤与代码说明 1、在syscall.h中添加PrintInt调用类型。   1. #define SC\_PrintInt 11   即顺延编号为11。  2、在starts.s中添加PrintInt调用的入口。   1. .globl PrintInt 2. .ent PrintInt 3. PrintInt: 4. addiu $2,$0,SC\_PrintInt 5. syscall 6. j $31 7. .end PrintInt   仿照其他调用编写即可，除了名称以外没有区别。  3、在exception.cc中添加打印整数的逻辑：   1. case SC\_PrintInt:{ 2. int num=machine->ReadRegister(4); 3. printf("%d\n",num); 4. IncrementPC(); 5. break; 6. }   需要注意的是，在测试中发现，此系统调用的参数就是要打印的数本身（类似于立即数），而不是这个数的地址。  逻辑简单，不需要转换成内核态，直接在此处处理即可。  同样地，此系统调用之后也需要增加程序计数器的值。 五、调试过程及记录 写实验的时候并没有记录这个…但期间用了git做版本控制（其实就是隔一段时间备份一下），可以回忆一下出的各种错误：    利用BitMap不熟练导致的笔误：    将二进制文件写入内存时的逻辑错误。之前认为新分配的物理空间一定是连续的，但事实上有可能不是，比如一个占用空间较小的进程被释放后，下一个进程的空间就会不连续：    错误地认为PrintInt的参数还是地址，导致执行时出现了段错误，因为那个地址恰好不允许访问（还好不允许，不然错误的地方可能不好找）：    事实上，这个实验调试起来还是比较顺利的，本身也没有太多的复杂逻辑，主要是一些原有函数的调用。  最终执行的二进制文件exec.noff的源码如下：   1. #include "syscall.h" 2. int 3. main() 4. { 5. SpaceId pid; 6. PrintInt(12345); 7. pid = Exec("../test/halt2.noff"); 8. Halt(); 9. */\* not reached \*/* 10. }   其中调用了halt2.noff，其源码：   1. #include "syscall.h" 2. int 3. main() 4. { 5. PrintInt(67890); 6. Halt(); 7. */\* not reached \*/* 8. }  六、运行结果 [zhangboning@ubuntu ~/nachos-3.4-ualr-2022/code/lab6]$ make  make: “arch/unknown-i386-linux/bin/nachos”是最新的。  [zhangboning@ubuntu ~/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  Executing SysCall Exec()  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  ============================================  Exec(../test/halt2.noff):  67890  Machine halting!  Ticks: total 68, idle 0, system 30, 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... 七、类Unix/Linux的Fork()： Unix中的fork()即建立一个与主进程基本一致的子进程，但子进程并不执行fork()之前的语句，只是复制了主进程的数据而已。而且父进程中fork()返回子进程的进程号，子进程中则返回0。 7.1 异常处理 其类型与实验中的Exec()类似，都可以作为中断处理，那么也通过异常处理判断类型后调用interrupt中的自定义函数完成操作。  其中异常判断比较简单，代码应该是这样的：   1. case SC\_Fork:{ 2. interrupt->Fork(); 3. IncrementPC(); 4. break; 5. }   就是简单的调用中断函数，PC+1。 7.2 中断处理 基于此逻辑，Fork()中断处理程序Interrupt::Fork()应具有以下功能：  1、创建子进程thread；  2、将子进程的父进程设置为currentThread；  3、拷贝父进程的地址空间，但要先将所有pageTable[i].readOnly设为TRUE；  4、将进程绑定到内核线程；  5、为子进程初始化寄存器、恢复进程空间；  6、thread->Fork()分配栈空间；  7、写返回值（新进程的编号）到reg2中。  其中，thread->Fork()的两个参数为接下来定义的InitFork和地址空间编号。  InitFork()应具有以下功能：  1、写返回值0到reg2中；  2、增加程序计数器；  3、machine->Run()。  其代码大概如下：   1. void 2. InitFork(){ 3. machine->WriteRegister(2,0); 4. machine->IncrementPC(); 5. machine->Run(); 6. }   以上就完成了fork()在子进程中返回0的逻辑，并且由于地址空间是复制的父进程，那么访问的代码空间也是父进程的，也就是会继续执行父进程的逻辑。 八、写时复制机制8.1 异常捕获 在（七）中，由于将父子进程的地址空间的页表都设置为了只读，那么在任何一个进程试图在某个页进行写操作的时候，会触发nachos的只读异常（ReadOnlyException）。  而此时地址异常寄存器（BadVAddrReg）会记录异常产生的虚拟页号，那么此异常在异常处理类exception.cc中可以进行捕获并指定中断处理程序：   1. if(which==ReadOnlyException){ 2. int badVAddr=machine->ReadRegister(BadVAddrReg); 3. interrupt->CopyOnWrite(badVAddr); 4. }   由于fork()后内核将子进程放在队列的前面，因此触发此异常的一般也就是子进程，此机制也就为子进程分配了最低必要的物理空间，而只读数据/程序等一直与父进程共享物理内存，节省了空间。  至于真正的访问只读页异常（程序逻辑错误等，例如段错误），还需要在进程等地方加上其他参数加以区分，但这就不是写时复制机制的讨论范围了。 8.2 中断处理程序 至于中断处理程序Interrupt::CopyOnWrite(int badVAddr)，应该具有以下功能：  1、将子进程的只读异常页映射到新的物理页；  2、将父子进程的该页设为可写（readOnly=false）。  其代码应该如下：   1. int vPage=badVAddr/PageSize; 2. currentThread->space->CopyOnWrite(vPage); 3. currentThread->parent->space->SetWritable(vPage);   其中，映射到新的物理页时其状态就应该为可写，然后将其父进程的虚拟页设为可写即可。 8.3 操作页表的具体逻辑 至于此处AddrSpace::CopyOnWrite(int vPage)，其逻辑为：  1、找到一个新的空闲物理页，即freeMap->Find()；  2、将此页对应内容复制到新物理页；  3、将此页映射到新物理页；  4、将此页状态设为可写。  其代码逻辑应该类似：   1. int newPhysPage=freeMap->Find(); 2. memcpy(&(machine->mainMemory[newPhysPage\*PageSize]), 3. &(machine->mainMemory[pageTable[vPage].physicalPage\*PageSize]), 4. pageSize); 5. pageTable[vPage].physicalPage=newPhysPage; 6. pageTable[vPage].readOnly=FALSE;   而AddrSpace::SetWritable(int vPage)仅仅是将第vPage页设为可写而已，因为子进程的此页已经映射到另一个物理位置了，所以当前页对应的物理页就可以被父进程修改而不产生冲突。 九、类Unix/Linux的Exec() 在Unix/Linux下，进程调用exec()族函数后，将由一个新进程完全替代当前进程，并不创建新进程。也就是说当前进程的空间可以完全被新进程代替。事实上，exec()经常在fork()函数之后调用，以此创建新的程序运行环境，也可以提高代码复用程度、内核的内聚程度。  由于系统调用类型与本实验之前的Exec()等完全一致，所以修改Interrupt::Exec()中的内容就可以完成逻辑的修改。  此中断处理程序应包含以下逻辑：  1、获取可执行文件名，方法在实验报告中已经说明；  2、将可执行文件的内容覆盖写入地址空间，必要时进行调整；  3、初始化寄存器、恢复进程空间；  4、分配栈空间并运行（machine->Run()）；  5、将进程号（spaceId）写回reg2。  这样就完成了将新进程替代到原有进程的操作，进程号与原进程一致。  这样，返回到原位置（此处为异常处理函数）时，PC+1后的指令就是新的可执行文件中的指令了。  以上步骤中“必要时进行调整”指的是：  1、若新文件需要更大的空间，则继续在位图中寻找可用的页；  2、若新文件需要的空间较少，则在位图中释放多余的页。  与之前实现的在另一空间执行用户程序的Exec()相比，此处实现的直接复用了空间、进程号。而且并没有创建新线程然后Fork()，直接分配栈空间然后运行即可。 | |
| 结论分析与体会：  经过本次实验，我体会到了想要解决“增加、完善功能”问题的一般做法，就是先要尽可能熟悉原有的系统，并尽量了解可以使用的工具（例如本实验中使用了现有的位图代码）。在做好备份的前提下，逐步在原有的逻辑上添加合适的分支，必要时做一些修改。  我还注意到，原有的代码对增加逻辑非常开放，甚至对于修改也很开放。一般的程序设计能做到对增加开放已经不容易了，可见nachos的成功得益于其优质的代码质量和完备的注释。  在编写代码的过程中，我了解了Makefile的使用，这是一种将一些c++文件链接起来的手段。当然也仅限于了解，主要是感叹原有系统设计的巧妙，能将各个实验分开，使我们可以分别独立完成这些实验。  对于c++的语法有了更多的了解，例如#ifdef等元素是此前很少接触的。并且体会到了面向对象程序设计的优势，特别是在开闭原则等原则被很好遵守的情况下。难以想象如果这个系统是面向过程的，那修改起来会有多么复杂。  此外，我还提高了英文能力。Nachos的注释十分完备，虽然一些代码可以自解释，在不得已的情况下只能去读英文，所幸英文看起来并没有想象中复杂，更多的是心理障碍。另外我的linux系统中文输入法有问题（只能设置双拼，很奇怪），导致我没办法好好输入中文，所以我的注释也都使用了英文，也算是间接练习了英文能力吧。而且阅读英文使我了解了很多操作系统的专有名词的拼写，算是意外收获。  对于操作系统，我对“将用户进程绑定到核心线程”、页表、内存页面分配、多道用户程序、进程状态、系统异常、系统中断这些问题有了进一步的理解。  在其模拟机器、MIPS的逻辑中，我也加深了一些计算机组成原理的知识的理解，例如寄存器（程序计数器、存放参数）、内存管理、汇编代码（寻址方式）等。  我也体会到了开发系统内核工作量之大，需要考虑的问题之多，对体力、脑力、精力都是很大的考验。 | |