山东大学 软件 学院

操作系统 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202000300358 | 姓名：梁思睿 | | 班级：20.6 |
| 学号：202022300316 | 姓名：王丁 | | 班级：20.6 |
| 实验题目： NACHOS试验环境准备、安装与源码分析(Lab1) | | | |
| 实验学时：2 | | 实验日期：2022.10.26 | |
| 实验目的：   1. Nachos开发环境的安装测试(不含Linux系统本身及虚拟机软件的安装测试)。 2. Nachos实验代码框架(源码目录)的基本分析。 3. Nachos Makefile的基本分析。 4. 硬件机制模拟部分的实现原理分析，包括中断、时钟、CPU指令执行。 | | | |
| 硬件环境：  设备名称 Three  处理器 Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz  机带 RAM 16.0 GB (15.8 GB 可用)  设备 ID 8CA8307A-62B3-42B1-AEE1-871F308ACDF3  产品 ID 00342-35669-83174-AAOEM  系统类型 64 位操作系统, 基于 x64 的处理器  笔和触控 为 10 触摸点提供触控支持 | | | |
| 软件环境：  宿主机：Windows 10 21H1 64位  虚拟机软件：WSL2  Linux:Ubuntu20.04.1LTS (GNU/Linux 5.4.72-microsoft-standard-WSL2 x86\_64)  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：  **一、Nachos开发环境的安装测试安装编译基本工具包**  sudo apt-get install build-essential    安装c,c++多平台库    安装32位环境库  sudo apt-get install lib32ncurses5 lib32z1  sudo apt-get install zlib1g:i386 libstdc++6:i386  sudo apt-get install libc6:i386 libncurses5:i386  sudo apt-get install libgcc1:i386 libstdc++5:i386    安装低于 gcc5.0 的版本  sudo apt-get install gcc-4.8 gcc-4.8-multilib g++-4.8 g++-4.8-multilib    将低版本 gcc 与 g++切换为当前活跃版本  sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 40  sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8 40    检查版本是否正确  gcc -v  g++ -v    下载并解压nachos    进入解压后目录：  cd NachOS-4.1/code/build.linux/  make depend    make    建立交叉编译环境  拷贝mips-desxxxxxxx.tgz到根目录并解压，得到usr文件夹    进入到NachOS文件夹下面  cd coff2noff  ./configure    编译用户程序并测试    检查是否安装成功    **二、Nachos实验代码框架(源码目录)的基本分析**  Nachos本身是Linux宿主主机的一个进程，它包含一个MIPS虚拟机，用于执行用户程序，用户程序是C程序经过交叉编译后生成的MIPS可运行程序，Nachos Kernel负责用户程序的装载以及在MIPS模拟器上的执行。其体系结构可以概括如图：    Nachos源码目录分析  进行Nachos每个操作实验的学习之前，要先观察明白整个“森林”的全貌，即看一下Nachos的源码目录并明确每个目录包含的东西是完成什么功能的。Nachos共有五个功能模块，分别是机器模拟、线程管理、文件系统管理、用户程序和虚拟存储以及网络系统。  nachos-3.4-ualr-lw目录下包含的内容及其作用如下：   1. 一级文件夹： 2. c++example/：有关C++介绍和实例 3. code/：Nachos各部分代码，含MIPS可执行文件转换实用工具 4. doc/：原作者的简单介绍和原有的作业要求，有序需求可以参考 5. code目录下内容： 6. Makefile.dep：平台相关文件，包括x86 Linux平台。为各个Makefile所共用，一般不需要修改这个文件，Makefile.common是各个Makefile所共用的文件部分，一般不需要修改这个文件 7. bin/：Nachos用户MIP程序目标变换的实用程序及源码 8. filesys/：Nachos文件系统管理部分的代码 9. machine/：Nachos MIPS虚拟机模拟代码，一般不需要修改 10. network/：Nachos网络管理部分的代码 11. test/：一些测试用MIPS用户应用程序 12. threads/：线程管理部分的代码 13. userprog/：用户程序部分的代码 14. demo0/：一个用于演示Makefile的例子 15. lab2/至lab7/：待完成的lab2-lab7的代码放在对应的文件夹内，文件夹内已包含相应的Makefile，一般不需要修改就可以直接使用   注意，有些Nachos用户程序在实现虚拟内存之前无法运行，因为他们将使用过多的内存，而做过实验六我们就可以知道Nachos原来的物理帧只是定义了32个，用户进程所需内存超过这个大小的时候Nachos就会崩掉  Nachos常用命令  这里只分析以后实验中用到的或常用的Nachos命令：   1. -d：显示特定的调试信息。比如：-d f 为现实文件系统相关调试信息 2. -rs：使得线程间可以随机切换。比如：-rs 123，其中123为伪随机数种子 3. -x：执行一个用户程序。比如：./nachos -x ../test/halt.noff 4. -f：格式化模拟磁盘 5. -cp：将一个文件从宿主机拷贝到Nachos模拟磁盘上 6. -ap：将一个文件追加到模拟磁盘的另一个文件尾部 7. -D：打印出Nachos文件系统内容   **三、Nachos Makefile分析**   1. Makefile作用：一个工程中的源文件不计其数，按类型、功能、模块分别放在若干目录中，Makefile实现了“自动化编译”，一旦写好，只需要一个make命令，整个工程完全自动编译，极大的提高了软件开发效率。   make命令执行条件：make会检查磁盘上的文件，如果目标文件的时间戳（该文件生成或被改动的时间）比它的任何一个依赖文件旧的话，make就执行相应的命令，以便更新目标文件   1. Nachos中Makefile的编译器   code文件夹下不同的Makefile调用的C/C++编译器也不相同   |  |  | | --- | --- | | Makefile | 编译器 | | machine, threads, vm,  userprog, filrsys, network | Host g++ | | bin | Host gcc | | test | MPIS cross compiler gcc |  1. Makefile.dep   结合上课所学，查阅资料并结合本次实验所给的Makefile.dep文件学习可知，Makefile.dep文件根据安装Nachos时所使用的操作系统环境，定义一些相应的宏，供g++使用  Makefile.dep中，首先使用语句uname = $(shell uname)获取安装Nachos所使用的操作系统平台：  uname = $(shell uname)  然后用语句ifeq($(uname),xxxx)根据使用的平台定义给g++使用的相应的宏  然后用语句 ifeq ($(uname),xxxx)根据所使用的平台定义给 g++使用的相应  的宏：  ifeq ($(uname),OSF1)  HOST = -DHOST\_ALPHA  CPPFLAGS = $(INCDIR) -D HOST\_ALPHA  CPP=/usr/local/lib/gcc-lib/alpha-dec-osf4.0/2.7.2/cpp  arch = dec-alpha-osf  ifdef MAKEFILE\_TEST  GCCDIR = /home/unit/66204/gcc/bin/decstation-ultrix-  LDFLAGS = -T script -N  ASFLAGS = -mips2  endif  endif  在最后几行给出了 makefile.common 所使用的几个宏：  arch\_dir = arch/$(arch)  obj\_dir = $(arch\_dir)/objects  bin\_dir = $(arch\_dir)/bin  depends\_dir = $(arch\_dir)/depends   1. Makefile.common   结合上课所学，查阅资料并结合本次实验所给的 Makefile.common 文件学  习可知，Makefile.common 文件定义了编译链接生成一个完整的 Nachos 可执  行文件所需要的所有规则。重点部分分析如下：   1. 包含文件（44 行）：include ../Makefile.dep。 2. 定义编译时查找相关文件的路径（49--51 行）：   vpath %.cc ../network:../filesys:../vm:../userprog:../threads:../machine  vpath %.h  ../network:../filesys:../vm:../userprog:../threads:../machine  vpath %.s  ../network:../filesys:../vm:../userprog:../threads:../machine   1. 生成 Nachos 系统所需要产生的目标文件名及其存放路径（64--66 行）：   s\_ofiles = $(SFILES:%.s=$(obj\_dir)/%.o)  c\_ofiles = $(CFILES:%.c=$(obj\_dir)/%.o)  cc\_ofiles = $(CCFILES:%.cc=$(obj\_dir)/%.o)   1. 链接生成可执行文件（91--94 行）：   $(bin\_dir)/% :  @echo ">>> Linking" $@ "<<<"  $(LD) $(GCCOPT32) $^ $(LDFLAGS) -o $@  ln -sf $@ $(notdir $@)   1. 根据 C++源程序生成 Object 文件（98--100 行）：   $(obj\_dir)/%.o: %.cc  @echo ">>> Compiling" $< "<<<"  $(CC) $(CFLAGS) -c -o $@ $<   1. 根据 C 源程序生成 Object 文件（104--106 行）：   $(obj\_dir)/%.o: %.c  @echo ">>> Compiling" $< "<<<"  $(CC) $(CFLAGS) -c -o $@ $<   1. 根据汇编程序生成 Object 文件（114--118 行）：   $(obj\_dir)/%.o: %.s  @echo ">>> Assembling" $< "<<<"  $(CPP) $(CPPFLAGS) $< > $(obj\_dir)/tmp.s  $(AS) $(ASOPT32) -o $@ $(obj\_dir)/tmp.s  rm $(obj\_dir)/tmp.s   1. 根据 c++、c 和 s 文件及.h 生成 dependence 文件（139--152 行）：   $(depends\_dir)/%.d: %.cc  @echo ">>> Building dependency file for " $< "<<<"  @$(SHELL) -ec '$(CC) -MM $(CFLAGS) $< \  | sed '\''s@$\*.o[ ]\*:@$(depends\_dir)/$(notdir $@) $(obj\_dir)/&@g'\'' > $@'  $(depends\_dir)/%.d: %.c  @echo ">>> Building dependency file for" $< "<<<"  @$(SHELL) -ec '$(CC) -MM $(CFLAGS) $< \  | sed '\''s@$\*.o[ ]\*:@$(depends\_dir)/$(notdir $@) $(obj\_dir)/&@g'\'' > $@'  $(depends\_dir)/%.d: %.s  @echo ">>> Building dependency file for" $< "<<<"  @$(SHELL) -ec '$(CPP) -MM $(CPPFLAGS) $< \  | sed '\''s@$\*.o[ ]\*:@$(depends\_dir)/$(notdir $@) $(obj\_dir)/&@g'\'' > $@   1. 每个Lab中的Makefile   各个子目录中含有 MakeFile 文件及 MakeFile.local 文件，这两个文件会调用 code 根目录下的 MakeFile 文件。在终端下进入相应目录，利用 make 或 makeall 命令，可依据该目录下的 Makefile 文件生成包含相应功能的 Nachos 可执行程序。   1. 每个Lab中的Makefile.local   这些文件主要是对一些编译、链接及运行时所使用的宏进行定义。   1. CCFILES：指定在该目录下生成Nachos所涉及到的宏进行定义 2. INCPATH：指明所涉及的C++源程序中的头文件（.h文件）所在的路径，以便利用g++进行编译链接时通过这个路径查找这些头文件 3. DEFINES：传递各g++的一些标号或宏 4. **Nachos硬件机制模拟** 5. 中断   Nachos要处理的中断有时钟中断、磁盘中断、终端读/终端写以及网络接收/网络发送中断。  内存空间管理中学习的系统中断是由Translate模拟的，在程序执行时逻辑地址需要需要变换为物理空间。在真实的计算机中，这一工作是由 MMU硬件完成的。当变换发生错误时 MMU 会自动发出各种异常中断。Nachos 模拟带有 TLB 的页式内存管理。MMU 由函数 Translate 模拟。两个函数ReadMem 和 WriteMem在访问物理内存之前都要调用函数Translate将要访问的逻辑地址变换为物理地址。Translate 函数的代码可以在 machine /translate.cc中找到。如果变换发生错误函数 Translate 会返回一个异常。  Nachos 文件系统中的 I/O 中断驱动是由两个模块模拟的：Disk 、SynchDisk。Disk 模拟了设备控制和磁盘的自身驱动，SynchDisk 模拟了系统内核 I/O 系统的一部分。模拟硬盘驱动的源代码文件是：machine/disk.h 和machine/disk.cc。  网络发送部分的中断没有应用到，在此不做分析。   1. 时钟   文件 timer.cc 和 timer.h 中模拟了时钟中断。当生成出一个 Timer 类的实例时，就设计了一个模拟的时钟中断，当时钟中断时刻到来时，调用TimerHandler 函数，其调用 TimerExpired 方法，该方法将新的时钟中断插入到等待处理中断队列中，然后再调用真正的时钟中断处理函数。Nacho 虚拟机可以如同实际的硬件一样，每隔一定的时间会发生一次时钟中断，这是一个可选项，目前 Nachos 还没有充分发挥时钟中断的作用。   1. CPU 指令执行   CPU 指令的执行由 machine 类来实现，这个类模拟了 MIPS 机。在文件machine /mips.cc 第 30-45 行上的 Run()函数负责设置机器进入用户态，并且在一个无限循环中模拟取 CPU 指令和执行 CPU 指令的过程。执行一条 MIPS 指令调用的函数是 void OneInstuction(Instruction \*instr)，它模拟取下一条指令和执行一条指令的周期。  在取出一条指令和在指令解码之后，CPU 的处理进入一个大的 switch 语句中执行对应的操作。如果该指令执行成功，CPU 指令计数器被向前推进，准备执行下一条指令。  附加 关键源代码分析：   1. 线程调度类Scheduler   Scheduler负责线程调度  私有变量 List \*readyList：类别是 List，这个 readyList 就是就绪队列了。  ReadyToRun 方法：void ReadyToRun(Thread\* thread);  这个方法的作用是把传入的线程插入到就绪队列 readylist 里。原Nachos 的实现如下：  void  Scheduler::ReadyToRun (Thread \*thread)  {  DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());  thread->setStatus(READY);  readyList->Append((void \*)thread);  }  哪个线程来了，就将哪个线程放在队尾，即先来先服务  Thread \*  Scheduler::FindNextToRun ()  {  return (Thread \*)readyList->Remove();  }  将当前要上处理机的线程移出队列并将队列头指针指向下一个上处理机的线程   1. Nachos信号量   Nachos中的信号量实现：  Semaphore::Semaphore(const char\* debugName, int initialValue)  {  name = (char\*)debugName;  value = initialValue;  queue = new List;  }  void  Semaphore::P()  {  IntStatus oldLevel = interrupt->SetLevel(IntOff); // disable interrupts    while (value == 0) { // semaphore not available  queue->Append((void \*)currentThread); // so go to sleep  currentThread->Sleep();  }  value--; // semaphore available,  // consume its value    (void) interrupt->SetLevel(oldLevel); // re-enable interrupts  }  void  Semaphore::V()  {  Thread \*thread;  IntStatus oldLevel = interrupt->SetLevel(IntOff);  thread = (Thread \*)queue->Remove();  if (thread != NULL) // make thread ready, consuming the V immediately  scheduler->ReadyToRun(thread);  value++;  (void) interrupt->SetLevel(oldLevel);  }  value为信号量保存的值，queue是等待队列，PV操作直接用函数P()和函数V( )，初始时要给value赋值   1. 位示图BitMap   扇是磁盘中最小的存储单位，也是磁盘 I/O 操作最基本的单位。在Nachos 中以扇区为基本单位的自由存储空间是由位示图类 BitMap 管理的。首先明确位示图管理的原理：如果要占用一个扇则与其扇号对应的位 就置”1”, 如果要释放一个扇则与其扇号对应的位就置”0”。即 1 代表这个扇区被占用了，0 代表这个扇区还没有东西可以被分配。在bitmap中定义的成员变量：  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  变量 numBits 保存了能表示扇号的位数，numWords 保存了组成位示图的字数。需要特别注意的是最后一个 int 型指针，私有成员变量 map 指向一个保存位示图的内存。    int  BitMap::Find()  {  for (int i = 0; i < numBits; i++)  if (!Test(i)) {  Mark(i);  return i;  }  return -1;  }  find()方法的作用返回找到的第一个空闲位的索引同时将该位置“1”，而不是只返回空闲位  void  BitMap::FetchFrom(OpenFile \*file)  {  file->ReadAt((char \*)map, numWords \* sizeof(unsigned), 0);  }  void  BitMap::WriteBack(OpenFile \*file)  {  file->WriteAt((char \*)map, numWords \* sizeof(unsigned), 0);  }  由于内存是易失的，所以对应硬盘的位示图需要作为一个文件保存在磁盘上。它作为内核的一个特殊文件被管理。     1. Nachos异常处理   已知系统调用函数有用户进程发出，那个什么存根程序会引发一个异常或自陷，然后就是调用了 machine.cc 的 RaiseException 去处理（这个函数模拟硬件的动作，切换到系统态并且在异常处理完成后返回到用户态），这个RaiseException 里调用了 ExceptionHandler。ExceptionHandler(which)函数调用模拟硬件的动作发发一个异常中断到对应的异常处理程序，原来的定义如下：  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);  } | | | |
| 结论分析与体会：  本次实验过程中，我成功安装了vmware以及64位Ubuntu虚拟机，在虚拟机中成功安装Nachos  通过对源码的学习，我对Nachos各个部分有了大致的掌握，熟悉了Nachos的许多命令，为之后的实验做准备。通过对Nachos的Makefile分析与学习，我复习了上学期Makefile相关的知识并且尝试读懂和理解Nachos相关的Makefile文件，对Makefile有了更深刻的了解，并且熟悉了Nachos的编译过程。通过对Nachos模拟硬件机制的分析，我在读懂代码的同时对Nachos的原理有了更深入的了解，并回忆了操作系统中断相关知识。  在本次实验的学习过程中，我对接下来将要进行的操作系统实验有了一定的了解，并为接下来的实验过程打下基础。 | | | |