山东大学 软件 学院

**操作系统课程设计** 实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202100300063 | 姓名：李彦浩 | | 班级：工软 21 |
| 学号：202100300340 | 姓名：黄幸兒 | | 班级：工软 21 |
| 实验编号：Lab1 | | | |
| 实验题目：Nachos实验环境准备、安装与源码分析 | | | |
| 实验学时：5 | | 实验日期：2023.10.28 | |
| 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：  联想Thinkpad笔记本  Intel Core i5-8250 CPU 8核  8GB内存  1907GB SSD | | | |
| 软件环境：  宿主机：Windows 11 22H2 64位  虚拟机软件：VMware Workstation Pro 16.2.2 build-19200509  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： 目录 [1. Nachos开发环境安装测试 2](#_Toc23754)  [1.1. 开发工具的安装过程 2](#_Toc19022)  [1.2. Nachos安装工程 2](#_Toc18555)  [1.2.1. 安装g++并查看 2](#_Toc14969)  [1.2.2. 安装MIPS交叉编译器 3](#_Toc4818)  [1.2.3. 安装Nachos3.4 3](#_Toc18184)  [1.2.4. 测试Nachos Thread 3](#_Toc24518)  [1.2.5. 生成实用程序coff2noff和coff2flat 4](#_Toc20056)  [1.2.6. Make Nachos for User Program 4](#_Toc3439)  [1.2.7. Make User(MIPS) Test Programs 4](#_Toc20531)  [2. Nachos实验代码框架（源码目录）分析 5](#_Toc26239)  [3. Nachos Makefile的基本分析 6](#_Toc11201)  [3.1. Makefile.dep 7](#_Toc6069)  [3.2. Makefile.common 9](#_Toc25627)  [3.3. Makefile 12](#_Toc17364)  [3.4. Makefile.local 12](#_Toc13528)  [4. 硬件机制模拟部分原理分析 15](#_Toc21617)  [4.1. 时钟中断 15](#_Toc29741)  [4.1.1. 从启动内核开始 15](#_Toc22035)  [4.1.2 注入时钟与中断模块实例 18](#_Toc30466)  [4.1.3 时间片是怎么过期的？ 23](#_Toc25214)  [4.1.4 系统时钟是怎么推移的？ 26](#_Toc3746)  [4.2. CPU指令模拟 28](#_Toc29212)  [4.2.1. 指令集 28](#_Toc13551)  [4.2.2. 模拟指令 35](#_Toc30174) 1. Nachos开发环境安装测试1.1. 开发工具的安装过程 虚拟机ubuntu14是上学期装上的，版本为14.04.6  Untitled 1.2. Nachos安装工程1.2.1. 安装g++并查看 #更新ubuntu的源 sudo apt update #安装g++ sudo install g++ #安装后查看版本检查是否成功安装 g++ --version Untitled 11.2.2. 安装MIPS交叉编译器 先将资源包中的gcc-2.8.1-mips.tar.gz 复制到一个合适的文件夹中，比如Home位置。然后使用cd /usr/local打开目标文件夹，再使用sudo tar -xzvf ~/gcc-2.8.1-mips.tar.gz将压缩包解压到local目录下。  Untitled 2 1.2.3. 安装Nachos3.4 使用cd /home打开主文件夹，使用mkdir oscp新建文件夹，将资源包中的nachos-3.4-ualr-2022.tar.gz 复制到/home/oscp,使用tar -xzvf nachos-3.4-ualr-2022.tar.gz解压缩。  Untitled 3 1.2.4. 测试Nachos Thread Untitled 4 1.2.5. 生成实用程序coff2noff和coff2flat Untitled 5 1.2.6. Make Nachos for User Program Untitled 6  Untitled 7 1.2.7. Make User(MIPS) Test Programs Untitled 8  Untitled 9  Untitled 10 2. Nachos实验代码框架（源码目录）分析 本次课设使用nachos-3.4-ualr-2022。下属目录：  Untitled 11  其中code为源码目录。  主要分为四大部分：   1. **Makefile**：编译连接使用，根据不同的参数条件编译到不同处理器架构、系统上，提升Nachos兼容性与可移植性。    1. Makefile.common    2. Makefile.dep    3. Makefile.local    4. Makefile 2. **Source Code**：Nachos源码。系Nachos官方给出的示例源码，实现了一部分操作系统中的常见机制，如线程调度内核、信号量、互斥锁、文件系统、网络等，保留了许多机制未实现，留待学生自己探索。    1. threads：线程管理    2. machine：软件模拟机器    3. filesys：文件系统    4. network：网络管理    5. userprog：用户程序    6. test：测试应用 3. **Labs**：实验源码及程序存放处。按照给出的要求完成实验，在每个文件夹下编写源码以及Makefile，随后编译链接得到二进制可执行文件。    1. Lab2：线程优先级调度（静态优先级/老化)    2. Lab3：N线程屏障（同步问题、随机数种子问题）    3. Lab4：文件系统（大小可变、时间戳）    4. Lab5：二级索引（支持二级索引、统计信息、读写权限）    5. Lab6：系统调用与多道用户程序    6. Lab7：虚拟内存（内页置换、置换统计、OPT算法） 4. **MISC**：杂项。包括示例程序，作者附加的一些利用源码实现的应用程序（比如生产者消费者的monitor）以及目标码转换的程序    1. demo0/demo1：教你怎么迁移并编写自己的内核代码，含源码文件、Makefile、局部编译等。    2. monitor：管程实现样例，利用nachos/threads下的sync代码实现了一个生产者消费者队列（[ring.cc](http://ring.cc)）    3. bin：包含有用户程序目标码变换的程序   Nachos源码结构图如下：  Untitled 12 3. Nachos Makefile的基本分析 Nachos下，code目录有2个所有子目录公用的Makefile文件：   1. Makefile.common 2. Makefile.local   而在不同的子目录下，都有本地的两个Makefile文件：   1. Makefile 2. Makefile.local   结构如下图：  Untitled 13  先来看公用的Makefile。 3.1. Makefile.dep 用于定义由 g++使用的系统依赖关系的宏。这个文件会作为Makefile.common文件的本地依赖。这些依赖关系的宏有：   * HOST 主机系统架构 * arch 文档存放路径 * CPP C++编译器的名字 * CPPFLAGS C++编译开关 * GCCDIR g++安装路径 * LDFLAGS 程序链接开关 * ASFLAGS 汇编开关   我们使用的Ubuntu是i386处理器，linux操作系统，那么依赖宏就是：   |  | | --- | | ifeq ($(uname),Linux)  HOST\_LINUX=-linux  HOST = -DHOST\_i386 -DHOST\_LINUX  CPP=/lib/cpp  CPPFLAGS = $(INCDIR) -D HOST\_i386 -D HOST\_LINUX  arch = unknown-i386-linux  endif |  1. ifeq：if equals的缩写，uname是主机操作系统名称，这句话的意思是如果主机名称是Linux，那么定义以下依赖宏。   我们可以查看一下操作系统名称：  Untitled 14  是Linux，那么这个条件依赖将触发。   1. HOST\_LINUX：定义变量HOST\_LINUX，并将其设置为-linux ，用于后续的构建过程，以标识目标平台是Linux。 2. HOST：定义变量HOST，并将其设置为-DHOST\_i386 -DHOST\_LINUX，用于指定编译器选项，以标识目标平台是i386架构的Linux系统。 3. CPP：定义编译器变量CPP，并将其设置为/lib/cpp，用于指定预处理器的路径。   Untitled 15   1. CPPFLAGS：定义编译开关变量，设置为$(INCDIR) -D HOST\_i386 -D HOST\_LINUX，前者是包含头文件路径的变量，后者是编译器标志。 2. arch：定义归档变量，将编译产生的依赖文件、目标文件以及最后的可执行文件置入unknown-i386-linux这个文件夹。   对于这个arch，我们想把依赖文件.d，目标文件.o，以及二进制文件存放到不同的目录，那么就要：   |  | | --- | | arch\_dir = arch/$(arch)  obj\_dir = $(arch\_dir)/objects  bin\_dir = $(arch\_dir)/bin  depends\_dir = $(arch\_dir)/depends |  1. 先定义一个相对根路径变量arch\_dir，它的值等于arch/$(arch)，而在上一步，我们定义了arch变量为unkown-i386-linux。因此arch\_dir = arch/unkown-i386-linux。 2. 定义目标文件存放路径变量obj\_dir = arch/unkown-i386-linux/objects 3. 定义依赖文件存放路径变量depends\_dir = arch/unkown-i386-linux/depends 4. 定义二进制可执行文件存放路径变量bin\_dir = arch/unkown-i386-linux/bin   编译结束后，就会按照这个配置存放生成的文件。  Untitled 16  这里用命令行展示了这个文件目录的结构。 3.2. Makefile.common 之前提到过Makefile.dep是Makefile.common的依赖文件。Makefile.common负责使用之前条件定义的宏编写编译规则。  首先是vpath，定义了编译所需的源码文件的位置，如果在当前目录找不到源码文件，则逐一检索指定的路径。   |  | | --- | | vpath %.cc ../network:../filesys:../userprog:../threads:../machine  vpath %.h ../network:../filesys:../userprog:../threads:../machine  vpath %.s ../network:../filesys:../userprog:../threads:../machine |   这个文件中还定义了编译开关宏 CFLAGS、目标文件规则宏 ofile ，和最终目标程序 规则宏 program。   |  | | --- | | CFLAGS = $(GCCOPT32) -g -Wall -Wshadow $(INCPATH) $(DEFINES) $(HOST) -DCHANGED  s\_ofiles = $(SFILES:%.s=$(obj\_dir)/%.o)  c\_ofiles = $(CFILES:%.c=$(obj\_dir)/%.o)  cc\_ofiles = $(CCFILES:%.cc=$(obj\_dir)/%.o)  ofiles = $(cc\_ofiles) $(c\_ofiles) $(s\_ofiles)  program = $(bin\_dir)/nachos  $(program): $(ofiles) |  1. CFLGAS：编译开关宏 2. s/c/cc\_ofiles：目标文件位置，Makefile会根据我们提供的源码，将其中的汇编语言.s，C语言.c，C++语言.cc编译成目标文件，放置在之前在Makefile.dep中定义的obj\_dir下面。 3. ofiles：定义全体目标文件集合，也就是刚才2中生成的目标文件。 4. program：定义二进制可执行文件路径及文件名，比如这里是之前按Makefile.dep中定义的bin\_dir下，生成一个名叫nachos的可执行文件。 5. $(program)：按照根据ofiles集合中的所有目标文件，构造链接出一个可执行文件，并存放在4中定义的路径下。   刚才这些这不过是一些变量/集合/路径的定义，接下来定义的是真正的构造链接规则：   |  | | --- | | $(bin\_dir)/% :  @echo ">>> Linking" $@ "<<<"  $(LD) $(GCCOPT32) $^ $(LDFLAGS) -o $@  ln -sf $@ $(notdir $@) |  1. $(bin\_dir)/% :：这行定义了一个模式规则。$(bin\_dir)/%表示在$(bin\_dir)目录下的所有文件，无论文件名是什么。%是一个通配符，代表任何字符。 2. @echo ">>> Linking" $@ "<<<"：这行会在命令执行时打印出一条消息，显示正在链接的文件名。$@是规则中的模式变量，代表当前正在处理的文件名。 3. $(LD) $(GCCOPT32) $^ $(LDFLAGS) -o $@：使用g++（这里用$(LD)代表）链接所有的.o文件（这里用$^代表），生成可执行目标文件nachos（这里用$@代表）。 4. ln -sf $@ $(notdir $@)：当前目录中产生一个与可执行文件nachos同名的链接文件。实际扩展为 shell 命令：  |  | | --- | | ln –sf arch/unknown-i386-linux/bin/nachos nachos |   那么怎么编译出所有需要的目标文件.o呢？这里定义了编译构造规则：   |  | | --- | | $(obj\_dir)/%.o: %.cc  @echo ">>> Compiling" $< "<<<"  $(CC) $(CFLAGS) -c -o $@ $<  $(obj\_dir)/%.o: %.c  @echo ">>> Compiling" $< "<<<"  $(CC) $(CFLAGS) -c -o $@ $<  $(obj\_dir)/%.o: %.s  @echo ">>> Assembling" $< "<<<"  $(CPP) $(CPPFLAGS) $< > $(obj\_dir)/tmp.s  $(AS) $(ASOPT32) -o $@ $(obj\_dir)/tmp.s  rm $(obj\_dir)/tmp.s |   这一规则说明了怎样从main.cc产生arch/unknown-i386-linux/objects/main.o。这里不仅是C++文件，还有C文件以及汇编文件。  但我们知道源码之间是有依赖的，编译时需要根据这些依赖编译成对应的目标文件，然后才能链接。所以还需要定义依赖规则：  通过g++ 的MM开关列出对应.cc文件的依赖关系清单并通过管道将清单发送给sed命令，sed命令在清单的前面插入规定的目录路径后在指定的目录中生成对应的依赖关系文件.d。  然后Makefile.common将所有文件的依赖关系定义在变量dfile中，通过语句：   |  | | --- | | include $(dfiles) |   将所有它所建立的依赖关系文件都括进来。 3.3. Makefile 这个文件只是把通用Makefile.common和本地的Makefile.local放在一起，相当于一个过渡文件：   |  | | --- | | include Makefile.local  include ../Makefile.common |  3.4. Makefile.local 由于每个子目录中的源码结构都不同（比如依赖关系），因此预定义了一些特有的本地变量：   * **CCFILE** 构造本目录中 Nachos 系统所用到的 C++源文件的文件名串 * **INCPATH** 指示 g++编译器查找 C++源程序中括入的.h 文件的路径名串 * **DEFINES** 传递给 g++编译器的标号串，一般用于条件编译。  |  | | --- | | SFILES = switch$(HOST\_LINUX).s  # If you add new files, you need to add them to CCFILES,  # you can define CFILES if you choose to make .c files instead.  CCFILES = main.cc\\  list.cc\\  scheduler.cc\\  synch.cc\\  synchlist.cc\\  system.cc\\  thread.cc\\  utility.cc\\  threadtest.cc\\  synchtest.cc\\  interrupt.cc\\  sysdep.cc\\  stats.cc\\  timer.cc\\  prodcons++.cc\\  ring.cc  INCPATH += -I- -I../monitor -I../threads -I../machine  DEFINES += -DTHREADS |   这里放一张编译的图。  Untitled 17  可以看到上述分析的部分都有体现在这张图中，比如Compiling、Linking、ln -sf等。 4. 硬件机制模拟部分原理分析4.1. 时钟中断 这是一个很复杂的过程，尽管在问题一中采用了汇编语言来编写将PC切换到新线程的部分，但实质上他们属于同一个模块（可以把switch.s看作scheduler.cc的扩展）。  但在这个问题中，牵涉到了:   1. [线程控制模块thread.cc](http://xn--thread-vs3j025aq11a0lolj5b11l.cc) 2. [线程调度模块scheduler.cc](http://xn--scheduler-zq5pe75dxp5afi7bu0nq30c.cc) 3. [时钟模块timer.cc](http://xn--timer-b52ik78f5mhgz3h.cc) 4. 时钟统计模块(实际上可能不止这个功能）[stats.cc](http://stats.cc) 5. [中断控制模块interrupt.cc](http://xn--interrupt-pl6n986agppyx1bg6fwus.cc)   其实可能有人会觉得1和2不属于时钟中断的部分，这牵涉到时钟中断到底在做一件什么事情的问题。我觉得时钟中断本质上服务于线程以及线程切换，即：  多个线程并发执行的情况下，当前正在执行的线程用完了他的时间片（这个时间片是由时钟模块和时钟统计模块共同协作授予的），因此中断控制模块应该处理这件事，把他扔回ready queue，然后挑选出下一个应该得到执行的线程（如果我们采用静态优先级调度，可能就是扔回去的那个线程本身！）。纵观这整个过程，显然我们需要1和2的协助，在后面我们将看到，中断模块中将当前系统时钟向后推移的函数OneTick中调用了1中的Yield，而后者调用了2中的Run。  而可能会有人觉得时钟中断应该仅涉及到授予时间片以及开关中断的部分，这样ta就会把1和2这两个模块划分出去。我认为这是一个见仁见智的部分，但5个模块的划分更加贴合于源码。 4.1.1. 从启动内核开始 如果我们要给当前执行的线程分配时间片，那么当前至少要有一个正在执行的线程。当然，我们已经有一个main线程了。这是因为：   |  | | --- | | //main.cc  int  main(int argc, char \*\*argv)  {  int argCount; // the number of arguments  // for a particular command  DEBUG('t', "Entering main");  (void) Initialize(argc, argv);  //system.cc  currentThread = new Thread("main");  currentThread->setStatus(RUNNING); |   不过我们最好是复用这份代码而不是修改它。我们完全可以在测试环节创建新的线程，观察时钟中断是如何作用于他们的，也就是说：   |  | | --- | | //main.cc  #ifdef THREADS  ThreadTest(); |   修改这一部分的代码。（当然可以改一下条件编译的表达式，这个是源码）  我们将在ThreadTest中创建三个线程（需求），给他们注入适当的优先级，观察其执行过程。  这里就涉及到两个问题：   1. 不同于问题一中我们手动Yield了main线程，这里main线程不结束，怎么切换到我们Fork出来的三个线程？ 2. 系统时钟是怎样向后推移的？   对于问题一，nachos官方用了一个小trick。   |  | | --- | | //main.cc  // NOTE: if the procedure "main"  // returns, then the program "nachos"  // will exit (as any other normal program  // would). But there may be other  // threads on the ready list. We switch  // to those threads by saying that the  // "main" thread is finished, preventing  // it from returning.  currentThread->Finish(); |   观察这个代码块之前的代码，可以发现currentThread实际上就是main。可以看下官方给的注释：如果main线程返回，我们启动的nachos虚拟机也就退出了。但ready queue中仍然可能有其他线程，我们通过声明main线程已经结束了来把PC切换到这些线程上，以此防止虚拟机启动入口函数返回。  之前我们疑惑的点是main线程一直占着CPU不放，我们又没有手动让他放弃CPU（Yield）。那其他线程怎么获得CPU？答案就是声明main线程已经结束了，这样CPU就空出来了。  看下Finish函数的代码：   |  | | --- | | //thread.cc  void  Thread::Finish ()  {  (void) interrupt->SetLevel(IntOff);  ASSERT(this == currentThread);    DEBUG('t', "Finishing thread \\"%s\\"\\n", getName());    threadToBeDestroyed = currentThread;  Sleep(); // invokes SWITCH  // not reached  }  void  Thread::Sleep ()  {  Thread \*nextThread;    ASSERT(this == currentThread);  ASSERT(interrupt->getLevel() == IntOff);    DEBUG('t', "Sleeping thread \\"%s\\"\\n", getName());  status = BLOCKED;  while ((nextThread = scheduler->FindNextToRun()) == NULL)  interrupt->Idle(); // no one to run, wait for an interrupt    scheduler->Run(nextThread); // returns when we've been signalled  } |   这里先不用管while ((nextThread = scheduler->FindNextToRun()) == NULL)，毕竟我们还有三个线程等着跑呢。在Finish中，我们本质上是挂起了该线程，也就是调用Sleep时的status = BLOCKED; ，然后就可以去运行其他的线程了。  对于问题二，答案是：手动推移。但具体的方式需要结合其他模块的代码来讲。 4.1.2 注入时钟与中断模块实例 既然之前提到3、4、5模块将协助我们实现时钟中断，那么我们需要为虚拟机注入这三个模块的实例。   |  | | --- | | //system.cc  //Function:Initialize  stats = new Statistics(); // collect statistics  interrupt = new Interrupt; // start up interrupt handling  scheduler = new Scheduler(); // initialize the ready queue  if (randomYield) // start the timer (if needed)  timer = new Timer(TimerInterruptHandler, 0, randomYield); |   这里可以注意到timer模块实例(负责时钟模块）的注入是有条件的，也就是这个虚拟机可能压根没有时钟模块，但为了实现时钟中断，怎么可能没有时钟，这一点是我们后续coding过程中可能需要改造的。  这里对于stats的注入比较简单，注册了当前系统时钟和空闲总时间、kernel总时间、user总时间等一系列的统计量，不赘述了。   |  | | --- | | //stats.cc  Statistics::Statistics()  {  totalTicks = idleTicks = systemTicks = userTicks = 0;  numDiskReads = numDiskWrites = 0;  numConsoleCharsRead = numConsoleCharsWritten = 0;  numPageFaults = numPacketsSent = numPacketsRecvd = 0;  } |   而对于scheduler的注入也很简单，不赘述了。   |  | | --- | | //scheduler.cc  Scheduler::Scheduler()  {  readyList = new List;  } |  注入中断模块 重点在于interrupt和timer的注入，先看interrupt。   |  | | --- | | //interrupt.cc  Interrupt::Interrupt()  {  level = IntOff;  pending = new List();  inHandler = FALSE;  yieldOnReturn = FALSE;  status = SystemMode;  } |   各字段作用如下:   1. level:是否允许中断 2. pending:待处理的中断（后面我们会讨论这个List中的对象实例应该是怎样的） 3. inHandler:short for interrupt handler。简单来说就是是否在运行一个中断处理的句柄。 4. yieldOnReturn:当前线程是否要求Yield，或者更准确地说：当前线程的时间片是否要求他Yield。 5. status:当前线程的mode(idle,kernel/system,user)   后面我们会看到这些字段是怎么使用的。 2. 注入中断模块  |  | | --- | | //timer.cc  Timer::Timer(VoidFunctionPtr timerHandler, \_int callArg, bool doRandom)  {  randomize = doRandom;  handler = timerHandler;  arg = callArg;  // schedule the first interrupt from the timer device  interrupt->Schedule(TimerHandler, (\_int) this, TimeOfNextInterrupt(),  TimerInt);  } |   各字段作用如下:   1. randomize:时间片长度是否随机（也就导致了时钟中断发生的时间变得随机） 2. handler:处理中断发生的句柄。 3. arg:处理中断发生的句柄所需要的参数（实际上可能根本不需要参数，冗余了）   这里需要注意的是在Schedule中传递的TimerHandler函数本质上不是timerHandler参数，而是一个全局的函数。他们之间的关系是，当时间片过期时，TimerHandler调用timerHandler，而后者往往是声明并定义在system.cc中的全局函数TimerInterruptHandler。为什么这么设计呢？可能是为了方便扩展，毕竟一个时钟模块的处理中断的句柄的实现可能改变，甚至重新声明并定义一个句柄，这样timerHandler就可以不是TimerInterruptHandler了，实现解耦。  看一下这两个函数的代码：   |  | | --- | | //system.cc  static void  TimerInterruptHandler(\_int dummy)  {  if (interrupt->getStatus() != IdleMode)  interrupt->YieldOnReturn();  }  //timer.cc  // dummy function because C++ does not allow pointers to member functions  static void TimerHandler(\_int arg)  { Timer \*p = (Timer \*)arg; p->TimerExpired(); }  void  Timer::TimerExpired()  {  // schedule the next timer device interrupt  interrupt->Schedule(TimerHandler, (\_int) this, TimeOfNextInterrupt(),  TimerInt);  // invoke the Nachos interrupt handler for this device  (\*handler)(arg);  } |   这里由于Cpp特性，不能引用成员函数，所以为了把TimerExpired函数作为句柄传递，我们只能开个全局函数把他包装一下。  而调用的TimerExpired函数在分配下一个时间片后，调用了处理中断的函数句柄TimerInterruptHandler。（实际上这是由于我们注入处理中断的函数句柄时就用的这个函数）而后者的作用就是通知推进系统时钟的函数OneTick 。它应当以时间片过期为由，将当前线程扔回ready queue，然后重新取一个线程出来继续执行。这个后面再说。先来看看重新分配时间片的部分：   |  | | --- | | //interrupt.cc  void  Interrupt::Schedule(VoidFunctionPtr handler, \_int arg, int fromNow, IntType type)  {  int when = stats->totalTicks + fromNow;  PendingInterrupt \*toOccur = new PendingInterrupt(handler, arg, when, type);  DEBUG('i', "Scheduling interrupt handler the %s at time = %d\\n",  intTypeNames[type], when);  ASSERT(fromNow > 0);  pending->SortedInsert(toOccur, when);  } |   其实我觉得这里这个函数的功能和它的名字有点不对应。实际上当我们将一个函数命名为Schedule时，我们更希望它能负责生命周期更长的调度流程，比如将一个（中断）任务置入pending list；以及从pending list中取出（中断）任务，分析是否需要执行，然后采取不同的操作。  但这里的Schedule仅仅是将一个中断任务加入了pending list，在我的感觉里是不太合适的，但这是人家国外名校的教授大佬写的，很有可能是我理解不对。  Anyway，这就是这个函数的功能：生成一个中断任务，并将其加入pending list。  具体怎么生成的呢？注意fromNow，对于这个参数，我们一直都给它传递的一个时间片的长度：SystemTick=100ms。而在中断任务列表中的每个元素里，都有一个字段，存储了这个任务应该被执行的时间，也就是这个函数中声明的变量when 。when是一个绝对的时间（戳），而并非一个偏移量，偏移量是fromNow。因此，中断任务执行时间，或者，更形象地，当前线程执行截止时间when=totalTicks+fromNow。其中totalTicks是当前系统时间。  然后我们再来关注这个pending list里的元素应该是什么样的。   |  | | --- | | PendingInterrupt::PendingInterrupt(VoidFunctionPtr func, \_int param, int time, IntType kind)  {  handler = func;  arg = param;  when = time;  type = kind;  } |   各字段作用如下：   1. handler:分配下一个时间片以及处理中断的函数句柄。 2. arg:handler的参数（源码里根本用不上） 3. when:中断任务执行时间，或者，更形象地，当前线程执行截止时间 4. type:中断类型（时钟，磁盘，控制台读写，网络消息传递）   其实后半部分已经和这一个小节没什么关系了，但为了讲明白我们注入实例时，到底注入了一个什么东西，所以额外写了一点。  以下是时钟与中断交互的结构：  Untitled 18   1. 在中断模块实例中有一个成员变量叫做YieldOnReturn。这个变量的值标识了当前线程是否需要放弃CPU（可能会切换到其他线程上）。而这个变量的值是通过一个开关函数进行切换的（也就是上图的Yield Switch)，这个开关函数定义在了system.cc中，名叫TimerInterruptHandler。当调用这个开关函数时，它将声明当前线程需要放弃CPU。 2. 上述的开关函数被注入到Timer模块的实例中，在时间片过期时会调用这个表示当前线程时间片用完，应当失去CPU。这在用户视角下有点像一个回调函数。 3. 而调用这个开关函数的方式是在Timer实例的成员函数TimerExpired中调用，TimerExpired函数做了两件事情：    1. 将新的时间片加入时间片队列    2. 调用开关函数声明当前线程应当放弃CPU 4. 而Timer模块的实例本身被作为一个参数注入到了时间片中断事件的实例中，也就是Timer的实例被注入到了PendingInterrupt对象中（上图中的Pending List中的元素） 5. 在检查时间片过期的函数CheckIfDue中，如果System Tick抵达了Expire Time，那么就从Pending List中取出当前时间片的PendingInterrupt对象，此时这个PendingInterrupt对象中包括了Timer的实例，而这个实例中有成员函数TimerExpired，随后我们调用TimerExpired，他就会做它要做的两件事 6. 图中的Function wrapper就是令Timer模块实例调用TimerExpired的包装函数。  4.1.3 时间片是怎么过期的？ 一般来说，当我们说（快）到ddl了，实际上在说这样一件事：   1. 我们知道当前的时间 2. 我们知道截止的时间 3. 如果当前时间超过（大于等于）截止时间，那么相当于任务过期了   上一节我们提到，中断任务实例中有一个字段when，保存了这个中断任务何时执行，或者，当前线程何时截止。我们可以利用这个字段来判断时间片是否过期。流程如下：   1. 获取当前系统时间 2. 获取当前线程截止时间 3. 比较二者，如果后者大于等于前者，则时间片过期   看似这个流程很简单，但我们要考虑到一个问题：我们需要动态更新当前系统时间。因为nachos本质上是个虚拟机，没有任何模块或硬件会帮你更新现在虚拟机的时间。  时间看起来是一个很“连续”的东西，但我们又无法无限细分它，然后不断更新，所以我们更新的粒度（或者说步长）会大一些，源码中是10ms。   |  | | --- | | #define UserTick 1 // advance for each user-level instruction  #define SystemTick 10 // advance each time interrupts are enabled  #define RotationTime 500 // time disk takes to rotate one sector  #define SeekTime 500 // time disk takes to seek past one track  #define ConsoleTime 100 // time to read or write one character  #define NetworkTime 100 // time to send or receive one packet  #define TimerTicks 100 // (average) time between timer interrupts |   这就涉及到我们之前提过的一个问题了，怎样让虚拟机不断地把SystemTick加到totalTicks上？也就是说，时钟的推移。  我们可以先简单看一下后面那个动作：把SystemTick加到totalTicks上。Interrupt类中封装了一个公有的成员函数OneTick。   |  | | --- | | void  Interrupt::OneTick()  {  MachineStatus old = status;  // advance simulated time  if (status == SystemMode) {  stats->totalTicks += SystemTick;  stats->systemTicks += SystemTick;  } else { // USER\_PROGRAM  stats->totalTicks += UserTick;  stats->userTicks += UserTick;  }  DEBUG('i', "\\n== Tick %d ==\\n", stats->totalTicks);  // check any pending interrupts are now ready to fire  ChangeLevel(IntOn, IntOff); // first, turn off interrupts  // (interrupt handlers run with  // interrupts disabled)  while (CheckIfDue(FALSE)) // check for pending interrupts  ;  ChangeLevel(IntOff, IntOn); // re-enable interrupts  if (yieldOnReturn) { // if the timer device handler asked  // for a context switch, ok to do it now  yieldOnReturn = FALSE;  status = SystemMode; // yield is a kernel routine  currentThread->Yield();  status = old;  }  } |   假设现在是kernel态，那么OneTick就执行了这样的语句：stats->totalTicks += SystemTick;。随后它会检查pending list中是否有过期的时间片。这一段单独封装了一个CheckIfDue函数。   |  | | --- | | bool  Interrupt::CheckIfDue(bool advanceClock)  {  MachineStatus old = status;  int when;  ASSERT(level == IntOff); // interrupts need to be disabled,  // to invoke an interrupt handler  if (DebugIsEnabled('i'))  DumpState();  PendingInterrupt \*toOccur =  (PendingInterrupt \*)pending->SortedRemove(&when);  if (toOccur == NULL) // no pending interrupts  return FALSE;  if (advanceClock && when > stats->totalTicks) { // advance the clock  stats->idleTicks += (when - stats->totalTicks);  stats->totalTicks = when;  } else if (when > stats->totalTicks) { // not time yet, put it back  pending->SortedInsert(toOccur, when);  return FALSE;  }  // Check if there is nothing more to do, and if so, quit  if ((status == IdleMode) && (toOccur->type == TimerInt)  && pending->IsEmpty()) {  pending->SortedInsert(toOccur, when);  return FALSE;  }  DEBUG('i', "Invoking interrupt handler for the %s at time %d\\n",  intTypeNames[toOccur->type], toOccur->when);  #ifdef USER\_PROGRAM  if (machine != NULL)  machine->DelayedLoad(0, 0);  #endif  inHandler = TRUE;  status = SystemMode; // whatever we were doing,  // we are now going to be  // running in the kernel  (\*(toOccur->handler))(toOccur->arg); // call the interrupt handler  status = old; // restore the machine status  inHandler = FALSE;  delete toOccur;  return TRUE;  } |   其中，官方对advanceClock的解释是："advanceClock" -- if TRUE, there is nothing in the ready queue,so we should simply advance the clock to when the next pending interrupt would occur (if any). If the pending interrupt is just the time-slice daemon, however, then we're done!  简单来说就是，如果ready queue中没有线程，也就是说没有其他线程等待使用CPU，那么我们就可以简单地把时钟推移到这个时间片结束的时间，如果这个任务仅仅就是个时间片过期的守护线程，那么我们直接返回False。也就是通知虚拟机时间片没过期，因为即便真的过期了，也不会有其他线程需要下一个时间片了！  另外，当没有时间片过期的中断任务或者时间片还没过期时，我们也返回False。这样函数调用结束，返回OneTick时，yieldOnReturn为false（CheckIfDue没有改变它，直接返回了False)，当前线程不会结束。  但如果时间片确实过期了，我们就需要执行之间注入的TimerHandler句柄了。也就是：   |  | | --- | | (\*(toOccur->handler))(toOccur->arg); // call the interrupt handler |   如上文提到的，他会分配下一个时间片，然后调用向虚拟机声明时间片已经过期，当前线程需要切换的函数TimerExpired，注意，这个函数仅仅声明当前线程需要切换，并没有真的切换线程！  而切换线程的部分写到了OneTick中，这看起来有些混乱，实际上不是，如果你把CheckIfDue这个封装好的函数直接内联到OneTick中，这就是一个很清晰的流程：检查是否已经过期，如果已经过期，声明切换线程的意愿；如果声明了这个意愿，切换线程。 4.1.4 系统时钟是怎么推移的？ 上一节中提到的OneTick函数，实际上只是推移系统时钟并在恰当的时机切换线程这个动作本身。就如同我们的四肢会做动作一样。但如果没有一个大脑（线程）去指挥四肢做这个动作（调用这个函数），四肢是不会莫名其妙在没有任何外界干扰的情况下动起来的（系统时钟是不会自动更新的）。  一开始困扰我的部分在于，我找遍了整个库，都没有发现有哪里一直调用OneTick，或者说，这个函数的调用时机很隐蔽，所以我没能理解系统时钟是如何持续推进的，直到我发现实现这个过程是需要我们自己去编码的。  因此我们需要做的就是：尝试手动调用OneTick（尝试让大脑指挥四肢做动作）。  但这里的调用OneTick，不是直接调用这个函数，nachos官方对此给出了解释：   |  | | --- | | //----------------------------------------------------------------------  // Interrupt::OneTick  // Advance simulated time and check if there are any pending  // interrupts to be called.  //  // Two things can cause OneTick to be called:  // interrupts are re-enabled  // a user instruction is executed  //---------------------------------------------------------------------- |   只有两件事情可以调用OneTick：   1. 重新开中断 2. 执行用户指令   后面这个暂时用不到，就不谈了。主要在于重新开中断上。  源码中对应的函数为SetLevel。   |  | | --- | | IntStatus  Interrupt::SetLevel(IntStatus now)  {  IntStatus old = level;    ASSERT((now == IntOff) || (inHandler == FALSE));// interrupt handlers are  // prohibited from enabling  // interrupts  ChangeLevel(old, now); // change to new state  if ((now == IntOn) && (old == IntOff))  OneTick(); // advance simulated time  return old;  } |   如果当前中断的状态为开并且之前中断的状态为关，那么系统时钟向前推进。这很好理解，因为当我们打开中断时，就是线程继续执行的时刻，也就是系统时间推进的时刻。  把状态控制标记转换和为了方便调试的断言去了，这个函数就是在做上面说的这件事情。  那么什么时候我们会开关中断呢？比如，我们在设置一个线程Ready态，或者调度某个线程执行时，不希望被中断，那么就是这个时候会有开关中断的操作。  那什么时候调度某个线程执行呢？一般来说是把一个旧线程切换到一个新线程的时候。  那什么时候切换呢？时间片过期的时候？这里不可以是，别忘了我们分析这么一堆，就是为了写一段代码，可以推移系统时间，然后判断时间片是否过期。  那就只剩下线程自动放弃CPU了。也就是说，一个线程每执行完一段，我们都要让他放弃一次CPU，推移一次系统时间，然后因为我们的线程是具备了静态优先级的，所以如果它还没执行完，下次调度的线程还是它，它就继续执行，直到时间片用完。（如果不是静态优先级，我们得把这个线程插入到ready queue队首，因为它放弃CPU只是为了让我们推移系统时间的，不是真的不打算继续执行了） 4.2. CPU指令模拟4.2.1. 指令集 **1. OP Code**  Nachos选择使用软件去模拟MIPS指令集，但不同于实际使用的MIPS指令集（总共支持约111条指令），Nachos仅支持了63条指令。（其中只有61条是有效的，一条还没有被模拟器实现的，另一条保留的）我们知道指令需要操作码（Operation Code，OP Code)，操作码和实际指令一一对应。这些操作码定义在了/code/machie/mipssim.h文件中（顾名思义，mips simulator，MIPS指令集模拟器）   |  | | --- | | #define OP\_ADD 1  #define OP\_ADDI 2  #define OP\_ADDIU 3  #define OP\_ADDU 4  #define OP\_AND 5  #define OP\_ANDI 6  #define OP\_BEQ 7  #define OP\_BGEZ 8  #define OP\_BGEZAL 9  #define OP\_BGTZ 10  #define OP\_BLEZ 11  #define OP\_BLTZ 12  #define OP\_BLTZAL 13  #define OP\_BNE 14  #define OP\_DIV 16  #define OP\_DIVU 17  #define OP\_J 18  #define OP\_JAL 19  #define OP\_JALR 20  #define OP\_JR 21  #define OP\_LB 22  #define OP\_LBU 23  #define OP\_LH 24  #define OP\_LHU 25  #define OP\_LUI 26  #define OP\_LW 27  #define OP\_LWL 28  #define OP\_LWR 29  #define OP\_MFHI 31  #define OP\_MFLO 32  #define OP\_MTHI 34  #define OP\_MTLO 35  #define OP\_MULT 36  #define OP\_MULTU 37  #define OP\_NOR 38  #define OP\_OR 39  #define OP\_ORI 40  #define OP\_RFE 41  #define OP\_SB 42  #define OP\_SH 43  #define OP\_SLL 44  #define OP\_SLLV 45  #define OP\_SLT 46  #define OP\_SLTI 47  #define OP\_SLTIU 48  #define OP\_SLTU 49  #define OP\_SRA 50  #define OP\_SRAV 51  #define OP\_SRL 52  #define OP\_SRLV 53  #define OP\_SUB 54  #define OP\_SUBU 55  #define OP\_SW 56  #define OP\_SWL 57  #define OP\_SWR 58  #define OP\_XOR 59  #define OP\_XORI 60  #define OP\_SYSCALL 61  #define OP\_UNIMP 62  #define OP\_RES 63  #define MaxOpcode 63 |   其中OP\_UNIMP就是之前提到的合法但还没被实现的指令对应的操作码（Unimplemented）。OP\_RES是保留操作码（Reserved）。  **2. MIPS指令格式**  首先先来看MIPS适用于什么架构下的CPU：   |  | | --- | | 32个寄存器：  $s0---$s7(16~23) 存放变量的寄存器  $t0---$t9 (8~15) 存放临时变量的寄存器  用于过程（函数）中：  $a0---$a3 用于传递参数的4个寄存器  $v0---$v1 用于返回值的两个值寄存器 (val)  $ra 用于返回起始点的返回地址寄存器 (register address，向PC返回调用点的地址)  $zero 存放常数0的寄存器  $gp: 静态数据的全局指针寄存器(reg 28) global pointer for static data (reg 28)  $sp: 堆栈指针寄存器stack pointer (reg 29)  $fp: 帧指针寄存器(frame pointer) ，保存过程帧的第一个字 (reg 30) |   例如：add $t0,$s0,$s1代表将t0寄存器中的数加到s0寄存器中的数上，结果保存到s1。  MIPS指令集有三种类型的指令：   1. R型：一般做算术逻辑运算 2. I型：一般做立即数寻址 3. J型：一般做跳转  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 操作码 | OP | Rs | Rt | Rd | shamt | funct | | 长度 | 6b | 5b | 5b | 5b | 5b | 6b | | 含义 | R型全0 | 第一个源操作数 | 第二个源操作数 | 保存结果 | 移位常数 | 功能码 |   其实按我的理解，功能码就是扩展操作码。比如6位OP不够用了，就再加6位funct进行扩展。举个可能在实际中不存在的例子：  两条指令的目标都是ADD。但是其中一个是要Rs-Rt，另一个Rs+Rt。大家OP都一样，怎么办？那这个时候可以用funct码来进行区分。（实际可能不是这样，我假想的，只是为了说明funct的作用）  举个例子：   | **指令** | **[31:26]** | **[25:21]** | **[20:16]** | **[15:11]** | **[10:6]** | **[5:0]** | **指令功能** | | --- | --- | --- | --- | --- | --- | --- | --- | | add | 000000 | rs | rt | rd | 00000 | 100000 | 寄存器加 |   （**2） MIPS I型指令**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 操作码 | OP | Rs | Rt | imm(立即数) | | 长度 | 6b | 5b | 5b | 16b | | 含义 | 操作码 | 源操作数，寄存器中 | 保存位置 | 源操作数，立即数 |   举个例子：   | **指令** | **[31:26]** | **[25:21]** | **[20:16]** | **[15:0]** | **指令功能** | | --- | --- | --- | --- | --- | --- | | addi | 001000 | rs | rt | imm | 寄存器和立即数加 |   imm16位也说明寄存器16位，上限65535。  （**3）MIPS J型指令**   | **操作码** | **OP** | **imm** | | --- | --- | --- | | 长度 | 6b | 26b | | 含义 | 操作码 | 跳转地址 |   举个例子：   | **指令** | **[31:26]** | **[25:0]** | **指令功能** | | --- | --- | --- | --- | | j | 000010 | addr | 无条件跳转 |   addr26位说明地址总线至少26位了。  **3. Nachos对于指令集的定义**  在mipssim中，定义了不同指令的类型：   |  | | --- | | #define SPECIAL 100  #define BCOND 101  #define IFMT 1  #define JFMT 2  #define RFMT 3  struct OpInfo {  int opCode; /\* Translated op code. \*/  int format; /\* Format type (IFMT or JFMT or RFMT) \*/  }; |   其中I/J/RFMT就是上文中说的I/J/R型指令。  随后它打了一个表，用来将OP码映射到类型上：   |  | | --- | | struct OpInfo {  int opCode; /\* Translated op code. \*/  int format; /\* Format type (IFMT or JFMT or RFMT) \*/  };  static OpInfo opTable[] = {  {SPECIAL, RFMT}, {BCOND, IFMT}, {OP\_J, JFMT}, {OP\_JAL, JFMT},  {OP\_BEQ, IFMT}, {OP\_BNE, IFMT}, {OP\_BLEZ, IFMT}, {OP\_BGTZ, IFMT},  {OP\_ADDI, IFMT}, {OP\_ADDIU, IFMT}, {OP\_SLTI, IFMT}, {OP\_SLTIU, IFMT},  {OP\_ANDI, IFMT}, {OP\_ORI, IFMT}, {OP\_XORI, IFMT}, {OP\_LUI, IFMT},  {OP\_UNIMP, IFMT}, {OP\_UNIMP, IFMT}, {OP\_UNIMP, IFMT}, {OP\_UNIMP, IFMT},  {OP\_RES, IFMT}, {OP\_RES, IFMT}, {OP\_RES, IFMT}, {OP\_RES, IFMT},  {OP\_RES, IFMT}, {OP\_RES, IFMT}, {OP\_RES, IFMT}, {OP\_RES, IFMT},  {OP\_RES, IFMT}, {OP\_RES, IFMT}, {OP\_RES, IFMT}, {OP\_RES, IFMT},  {OP\_LB, IFMT}, {OP\_LH, IFMT}, {OP\_LWL, IFMT}, {OP\_LW, IFMT},  {OP\_LBU, IFMT}, {OP\_LHU, IFMT}, {OP\_LWR, IFMT}, {OP\_RES, IFMT},  {OP\_SB, IFMT}, {OP\_SH, IFMT}, {OP\_SWL, IFMT}, {OP\_SW, IFMT},  {OP\_RES, IFMT}, {OP\_RES, IFMT}, {OP\_SWR, IFMT}, {OP\_RES, IFMT},  {OP\_UNIMP, IFMT}, {OP\_UNIMP, IFMT}, {OP\_UNIMP, IFMT}, {OP\_UNIMP, IFMT},  {OP\_RES, IFMT}, {OP\_RES, IFMT}, {OP\_RES, IFMT}, {OP\_RES, IFMT},  {OP\_UNIMP, IFMT}, {OP\_UNIMP, IFMT}, {OP\_UNIMP, IFMT}, {OP\_UNIMP, IFMT},  {OP\_RES, IFMT}, {OP\_RES, IFMT}, {OP\_RES, IFMT}, {OP\_RES, IFMT}  }; |   这个仅仅是操作码和指令类型对应，那么功能码呢？  又打了个表，把功能码按索引和操作码对应上了。（我有点奇怪为什么它打表不打同一张表，而是拆开来，后来我想想有可能是要解耦合，因为操作码和类型是绑死的，这个不会动，但是功能码可能有扩展甚至改动的需求，所以拆了两张表）   |  | | --- | | /\*  \* The table below is used to convert the "funct" field of SPECIAL  \* instructions into the "opCode" field of a MemWord.  \*/  static int specialTable[] = {  OP\_SLL, OP\_RES, OP\_SRL, OP\_SRA, OP\_SLLV, OP\_RES, OP\_SRLV, OP\_SRAV,  OP\_JR, OP\_JALR, OP\_RES, OP\_RES, OP\_SYSCALL, OP\_UNIMP, OP\_RES, OP\_RES,  OP\_MFHI, OP\_MTHI, OP\_MFLO, OP\_MTLO, OP\_RES, OP\_RES, OP\_RES, OP\_RES,  OP\_MULT, OP\_MULTU, OP\_DIV, OP\_DIVU, OP\_RES, OP\_RES, OP\_RES, OP\_RES,  OP\_ADD, OP\_ADDU, OP\_SUB, OP\_SUBU, OP\_AND, OP\_OR, OP\_XOR, OP\_NOR,  OP\_RES, OP\_RES, OP\_SLT, OP\_SLTU, OP\_RES, OP\_RES, OP\_RES, OP\_RES,  OP\_RES, OP\_RES, OP\_RES, OP\_RES, OP\_RES, OP\_RES, OP\_RES, OP\_RES,  OP\_RES, OP\_RES, OP\_RES, OP\_RES, OP\_RES, OP\_RES, OP\_RES, OP\_RES  }; |   举个例子，第一条指令，（OP,TYPE,FUNCT) = (SPECIAL,RFMT,OP\_SLL)。其实就是shift left logical，逻辑左移指令的类型是R型，操作码是常量SPECIAL。  然后为了让用户看到每条指令都代表哪条MIPS汇编，他还打了个实际功能含义的映射表：   |  | | --- | | enum RegType { NONE, RS, RT, RD, EXTRA };  struct OpString {  const char \*string; // Printed version of instruction  RegType args[3];  };  static struct OpString opStrings[] = {  {"Shouldn't happen", {NONE, NONE, NONE}},  {"ADD r%d,r%d,r%d", {RD, RS, RT}},  {"ADDI r%d,r%d,%d", {RT, RS, EXTRA}},  {"ADDIU r%d,r%d,%d", {RT, RS, EXTRA}},  {"ADDU r%d,r%d,r%d", {RD, RS, RT}},  {"AND r%d,r%d,r%d", {RD, RS, RT}},  {"ANDI r%d,r%d,%d", {RT, RS, EXTRA}},  {"BEQ r%d,r%d,%d", {RS, RT, EXTRA}},  {"BGEZ r%d,%d", {RS, EXTRA, NONE}},  {"BGEZAL r%d,%d", {RS, EXTRA, NONE}},  {"BGTZ r%d,%d", {RS, EXTRA, NONE}},  {"BLEZ r%d,%d", {RS, EXTRA, NONE}},  {"BLTZ r%d,%d", {RS, EXTRA, NONE}},  {"BLTZAL r%d,%d", {RS, EXTRA, NONE}},  {"BNE r%d,r%d,%d", {RS, RT, EXTRA}},  {"Shouldn't happen", {NONE, NONE, NONE}},  {"DIV r%d,r%d", {RS, RT, NONE}},  {"DIVU r%d,r%d", {RS, RT, NONE}},  {"J %d", {EXTRA, NONE, NONE}},  {"JAL %d", {EXTRA, NONE, NONE}},  {"JALR r%d,r%d", {RD, RS, NONE}},  {"JR r%d,r%d", {RD, RS, NONE}},  {"LB r%d,%d(r%d)", {RT, EXTRA, RS}},  {"LBU r%d,%d(r%d)", {RT, EXTRA, RS}},  {"LH r%d,%d(r%d)", {RT, EXTRA, RS}},  {"LHU r%d,%d(r%d)", {RT, EXTRA, RS}},  {"LUI r%d,%d", {RT, EXTRA, NONE}},  {"LW r%d,%d(r%d)", {RT, EXTRA, RS}},  {"LWL r%d,%d(r%d)", {RT, EXTRA, RS}},  {"LWR r%d,%d(r%d)", {RT, EXTRA, RS}},  {"Shouldn't happen", {NONE, NONE, NONE}},  {"MFHI r%d", {RD, NONE, NONE}},  {"MFLO r%d", {RD, NONE, NONE}},  {"Shouldn't happen", {NONE, NONE, NONE}},  {"MTHI r%d", {RS, NONE, NONE}},  {"MTLO r%d", {RS, NONE, NONE}},  {"MULT r%d,r%d", {RS, RT, NONE}},  {"MULTU r%d,r%d", {RS, RT, NONE}},  {"NOR r%d,r%d,r%d", {RD, RS, RT}},  {"OR r%d,r%d,r%d", {RD, RS, RT}},  {"ORI r%d,r%d,%d", {RT, RS, EXTRA}},  {"RFE", {NONE, NONE, NONE}},  {"SB r%d,%d(r%d)", {RT, EXTRA, RS}},  {"SH r%d,%d(r%d)", {RT, EXTRA, RS}},  {"SLL r%d,r%d,%d", {RD, RT, EXTRA}},  {"SLLV r%d,r%d,r%d", {RD, RT, RS}},  {"SLT r%d,r%d,r%d", {RD, RS, RT}},  {"SLTI r%d,r%d,%d", {RT, RS, EXTRA}},  {"SLTIU r%d,r%d,%d", {RT, RS, EXTRA}},  {"SLTU r%d,r%d,r%d", {RD, RS, RT}},  {"SRA r%d,r%d,%d", {RD, RT, EXTRA}},  {"SRAV r%d,r%d,r%d", {RD, RT, RS}},  {"SRL r%d,r%d,%d", {RD, RT, EXTRA}},  {"SRLV r%d,r%d,r%d", {RD, RT, RS}},  {"SUB r%d,r%d,r%d", {RD, RS, RT}},  {"SUBU r%d,r%d,r%d", {RD, RS, RT}},  {"SW r%d,%d(r%d)", {RT, EXTRA, RS}},  {"SWL r%d,%d(r%d)", {RT, EXTRA, RS}},  {"SWR r%d,%d(r%d)", {RT, EXTRA, RS}},  {"XOR r%d,r%d,r%d", {RD, RS, RT}},  {"XORI r%d,r%d,%d", {RT, RS, EXTRA}},  {"SYSCALL", {NONE, NONE, NONE}},  {"Unimplemented", {NONE, NONE, NONE}},  {"Reserved", {NONE, NONE, NONE}}  }; |   这里有点让我好奇的是第15条，它打了个：   |  | | --- | | {"Shouldn't happen", {NONE, NONE, NONE}}, |   我就在想什么指令不能发生？后来我去查了一下，对应：   |  | | --- | | (OPC,TYPE,FUNCT) = (OP\_RES,IFMT,OP\_RES) |   操作码和功能码全被保留了，那就是这个指令根本不存在/无意义。所以会提示这么一句话，其他的OPC和FUNCT至少有一个有意义。 4.2.2. 模拟指令 实现在mipssim.cc中。本质上是Machine(Nachos虚拟机)的一个成员函数。当nachos带有用户程序条件编译的可执行文件运行时，将会注入虚拟机模块。   |  | | --- | | //system.cc:: Initialize  #ifdef USER\_PROGRAM  machine = new Machine(debugUserProg); // this must come first  #endif |   这个虚拟机里封装了一堆用户程序的函数,把Debug和处理Exception的去掉，就剩下：   |  | | --- | | // Routines callable by the Nachos kernel  void Run(); // Run a user program  int ReadRegister(int num); // read the contents of a CPU register  void WriteRegister(int num, int value);  // store a value into a CPU register  void OneInstruction(Instruction \*instr);  // Run one instruction of a user program.  void DelayedLoad(int nextReg, int nextVal);  // Do a pending delayed load (modifying a reg)    bool ReadMem(int addr, int size, int\* value);  bool WriteMem(int addr, int size, int value);  // Read or write 1, 2, or 4 bytes of virtual  // memory (at addr). Return FALSE if a  // correct translation couldn't be found. |   四个读写内存/寄存器的，一个运行多条的，一个运行一条指令的。前者由后者实现。这次定位到最重要的部分了：OneInstruction函数。   |  | | --- | | void  Machine::OneInstruction(Instruction \*instr)  {  int raw;  int nextLoadReg = 0;  int nextLoadValue = 0; // record delayed load operation, to apply  // in the future  // Fetch instruction  if (!machine->ReadMem(registers[PCReg], 4, &raw))  return; // exception occurred  instr->value = raw;  instr->Decode();  if (DebugIsEnabled('m')) {  struct OpString \*str = &opStrings[instr->opCode];  ASSERT(instr->opCode <= MaxOpcode);  printf("At PC = 0x%x: ", registers[PCReg]);  printf(str->string, TypeToReg(str->args[0], instr),  TypeToReg(str->args[1], instr), TypeToReg(str->args[2], instr));  printf("\\n");  }    // Compute next pc, but don't install in case there's an error or branch.  int pcAfter = registers[NextPCReg] + 4;  int sum, diff, tmp, value;  unsigned int rs, rt, imm;  // Execute the instruction (cf. Kane's book)  switch (instr->opCode) {    case OP\_ADD:  sum = registers[instr->rs] + registers[instr->rt];  if (!((registers[instr->rs] ^ registers[instr->rt]) & SIGN\_BIT) &&  ((registers[instr->rs] ^ sum) & SIGN\_BIT)) {  RaiseException(OverflowException, 0);  return;  }  registers[instr->rd] = sum;  break;  //后面的我就不放上来了，太长了 |   这个东西非常长，但是是合理的，因为一旦到了底层，那就要硬编码了，不然机器是不会自动把操作码和执行逻辑对应起来的。  这段代码简单来说就是：   1. 从内存读一条指令  |  | | --- | | // Fetch instruction  if (!machine->ReadMem(registers[PCReg], 4, &raw))  return; // exception occurred  instr->value = raw;  instr->Decode(); |  1. 更新程序计数器指针  |  | | --- | | // Compute next pc, but don't install in case there's an error or branch.  int pcAfter = registers[NextPCReg] + 4; |  1. 执行读取到的指令(代码不贴了，一大堆case太长了）   现在我们可以看一下一条简单的指令的实现：   |  | | --- | | case OP\_ADD:  sum = registers[instr->rs] + registers[instr->rt];  if (!((registers[instr->rs] ^ registers[instr->rt]) & SIGN\_BIT) &&  ((registers[instr->rs] ^ sum) & SIGN\_BIT)) {  RaiseException(OverflowException, 0);  return;  }  registers[instr->rd] = sum;  break; |  1. 首先从rs,rt两个源操作数寄存器中读取出操作数，加和暂存到sum变量。  |  | | --- | | sum = registers[instr->rs] + registers[instr->rt]; |  1. 异常处理：整型溢出  |  | | --- | | if (!((registers[instr->rs] ^ registers[instr->rt]) & SIGN\_BIT) &&  ((registers[instr->rs] ^ sum) & SIGN\_BIT)) {  RaiseException(OverflowException, 0);  return;  } |  1. 如果没有异常或者错误发生，把sum存到rd(也就是register destinied)目标寄存器中  |  | | --- | | registers[instr->rd] = sum; |   这样我们就用软件模拟出来一条MIPS指令ADD了。 | | | |
| 结论分析与体会：  本次实验我初步探究了nachos源码的基本结构，包括文件目录结构以及用于编译链接归档的Makefile目录。上个学期的操作系统实验我也写过一些Makefile，但那都是比较“小型”的，只有一个单文件，而且要声明的变量很少，只需要声明几个源文件，几个头文件，然后编译出目标文件进行链接就能得到最终的可执行文件了。  而nachos的Makefile分为各模块共有的Makefile.dep、Makefile.common以及每个模块私有的Makefile和Makefile.local。这样做可以将各模块的编译调试独立开来，更加细粒度地编写整个系统。  这其中也不乏我看不懂的Makefile的写法，参考了老师提供的编写Makefile的资料以及网络上的一些资源，我逐行剖析了所有的Makefile中的指令，打通了逻辑。  这之后，根据实验指导书的引领，我找到了machine模块，开始分析nachos官方作者造的轮子，包括时钟模块，中断模块以及指令模块。  在时钟模块中，我学习到了回调函数的实现方式，比如时间片过期/时钟中断的回调函数就是通过给Pending List中的PendingInterrupt实例注入一个时钟模块实例以及对应处理句柄，在其中调用TimerExpired时间片过期函数进行中断处理，随后我们只需要在时钟推进时，检查是否有时间片过期即可。如果有，则执行这个回调函数。  在指令模块中，我看到nachos官方使用软件模拟了MIPS指令集，首先注册了一个全局的驱动表，用来标识每条指令的操作码Operation Code，每条指令的类型（I型指令/J型指令/R型指令），每条指令的功能码Function Code。学习到了将OP Code和类型Type强绑定而与Function Code分离的解耦方式。  随后，nachos官方在Instruction中的OneInstruction函数编写模拟了MIPS指令集中的大部分指令，比如add指令是通过从寄存器数组中的rs寄存器和rt寄存器取出数，加和存到rd寄存器。  阅读nachos源码的过程时，我时常感到作者实现方式之精妙，各个模块内部高度聚合的同时模块间耦合度也尽可能降到了最小，这不仅是码力和基本功强就可以做到，同时也需要很多经验以及对整个system架构的深入理解，这些都是我们新人程序员值得学习的。 | | | |