



NACHOS 指南

目录

NACHOS 指南	1
1. 概述	4
1) Nachos 简介	4
2) CS163 实验简介	5
3) 实验小技巧	6
2. 环境配置	8
1) Nachos 的 eclipse 配置	8
2) 交叉编译 c 文件	10
I. 概述	10
II. 交叉编译简单介绍	10
III. Makefile 简单介绍	10
IV. 编译 c 文件	12
3. Nachos 源码解读	12
1) 快速开始	12
I. 项目文件的目录结构	13
II. 项目配置文件	13
III. 启动程序	14
2) 内核线程与 TCB	16
I. 内核线程	17
II. 线程控制块	18
III. 懒惰线程与无限前进机制	20
3) 中断机制	21
I. 中断机制	21
II. 时钟中断	23
4) 用户程序	24
I. 用户程序的加载与运行	24
II. 指令执行流程介绍	28
III. 系统调用陷入内核的机制	28
IV. 增添自定义的系统调用	30
5) 内存读写	32
I. 物理内存的创建	32
II. 处理器内部的内存读写	33
III. 进程的内存读写方法	35
6) 文件系统	35
I. 文件系统的底层实现	35
7) 网络机制	36
I. 网卡机制	37
II. 网络收发机制	39
4. 实验指南	44
1) 实验一：内核线程	44
I. 题目：	44

II. 思路:	44
2) 实验二: 多道程序设计	45
I. 题目:	45
II. 思路:	46
3) 实验三: 虚拟内存.....	47
I. 题目:	47
II. 思路:	47
4) 实验四: 网络	48
I. 题目:	48
II. 思路:	49

1. 概述

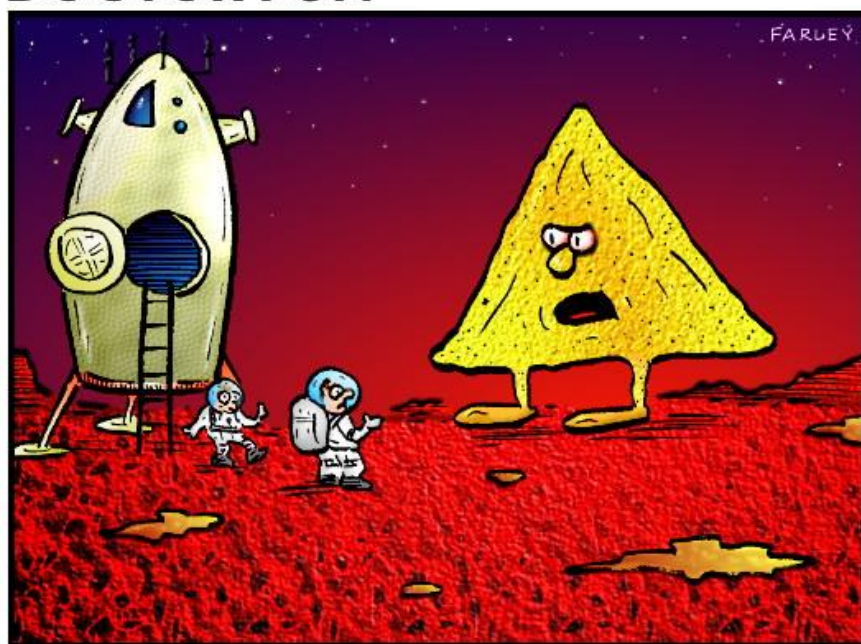
在这一章中，将对 Nachos 操作系统以及我们需要完成的操作系统课程设计实验进行一个简单的说明和介绍。

而在后面的三章里，将详细阐述 nachos 实验的环境配置方法，nachos 中较为关键的几个模块的代码解读，以及对我们要完成的四个实验进行简单的介绍和思路引导。

1) Nachos 简介

Nachos 是用于教授本科生和潜在的研究生级操作系统课程的教学软件。它由 Thomas Anderson 设计，在加州大学伯克利分校开发，并被世界各地的许多学校使用。

Nachos 最初是由 C++ 编写，作为主机进程在用户操作系统上运行。如今 Nachos 已经开发了 java 的版本，在 java 虚拟机上运行。对于 java 虚拟机或者用户操作系统而言，Nachos 就是一个普通的进程，但其很好的模拟了各项操作系统的功能，并且可以执行 MIPS 指令集下的用户程序，是一个很好的学习操作系统的教学软件。

DOCTOR FUN

© Copyright 1994 David Farley. World rights reserved.
This cartoon is made available on the Internet for personal viewing only.
dgf1@midway.uchicago.edu
Opinions expressed herein are not those of the University of Chicago
or the University of North Carolina.

"This is the planet where nachos rule."

图 1 Nachos 富有调侃意味的插图

2) CS163 实验简介

加州大学伯克利分校每一年都会开设一门编号为 CS163 的课程, 该课程是操作系统的课程设计课程。该课程每年均开设四个实验, 分别要求学生通过 Nachos 从内核线程设计、多道程序设计、虚拟内存、网络四个方面来了解操作系统的内核实现。

在官方网站中也给出了实验的三个小建议。第一点是先阅读作业中的代码, 直到你完全理解为止。第二点是在你明白你在做什么之前不要编码。设计, 设计, 设计第一, 只有这样你们才能分头编写代码。第三点是如果你陷入困境, 尽可能多地和其他人交谈。

- [Phase #1: Threads](#)
 - Initial design due 2/13
 - Design Reviews week of 2/13
 - Code due 3/2
 - Group evaluations, test cases, and final design docs due 3/3
 - [Autograder test cases](#)
- [Phase #2: Multiprogramming](#)
 - Initial design due 3/13
 - Design Reviews week of 3/13
 - Code due 3/23
 - Group evaluations, test cases, and final design docs due 3/24
 - [Autograder test cases](#)
- [Phase #3: Caching and Virtual Memory](#)
 - Initial design due 4/6
 - Design Reviews week of 4/10
 - Code due 4/18
 - Group evaluations, test cases, and final design docs due 4/19
 - [Autograder test cases](#)
- [Phase #4: Networks and Distributed Systems](#)
 - Initial design due 5/1
 - Design Reviews week of 5/2
 - Code due 5/10
 - Group evaluations, test cases, and final design docs due 5/11
 - [Autograder test cases](#)

图 2 CS163 课程 2006 年的四个实验

3) 实验小技巧

由于 Nachos 本身的源码解读文档较少，在我们实验的过程中，不妨有很多情况下需要自己去理解源码，来实现一些有趣的功能。在这里我将介绍一个自己阅读源码的小技巧，也是我完成操作系统课程设计所得到的收获之一。

阅读源码时，应该采取先总后分的方法。在进行实验时，我们首先应该找出那些有助于实验的 class，对这些 class 的总体结构进行理解，即理解每个 class 的总体结构，有什么变量和方法，每个变量代表什么含义，每个方法有什么作用。一般我们均采用 java 进行开发，以 eclipse 为例子，可以使用 eclipse 的 outline 对 class 的总体结构进行理解，对于大部分方法或者变量，在源码上会提供相应的注解。如下图所示。

理解了整体结构之后，对于那些需要我們进行修改，从而来实现某些特定功能的类方法，再细读其内部实现的具体代码，理解代码的逻辑，从而实现我们的

功能。有些实验还需要我们在类内添加一些新的方法和变量哟。

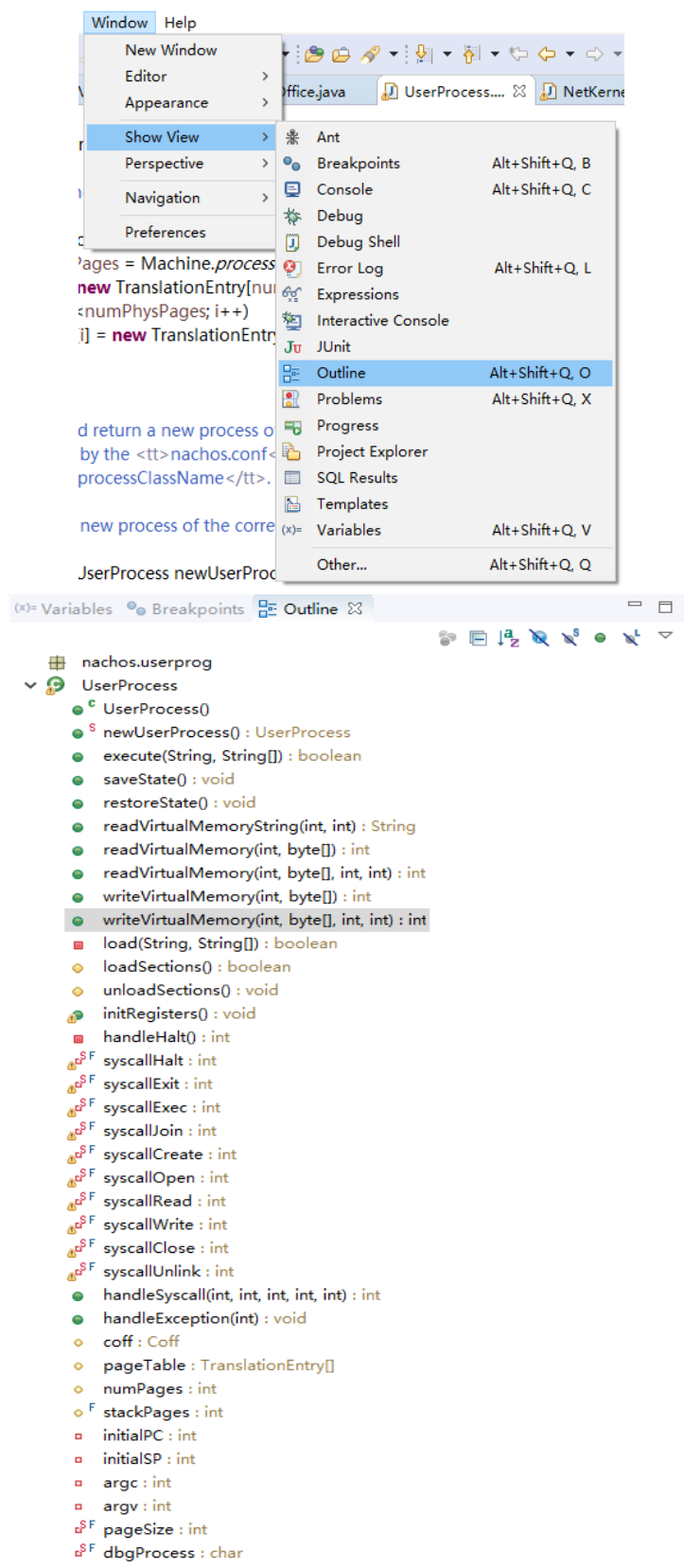


图 3 用 outline 对 class 的总体结构进行理解

```

*/
public class UserProcess {
    /**
     * Allocate a new process.
     */
    public UserProcess() {
        int numPhysPages = Machine.processor().getNumPhysPages();
        pageTable = new TranslationEntry[numPhysPages];
        for (int i=0; i<numPhysPages; i++)
            pageTable[i] = new TranslationEntry(i,i, true,false,false,false);
    }

    /**
     * Allocate and return a new process of the correct class. The class name
     * is specified by the <tt>nachos.conf</tt> key
     * <tt>Kernel.processClassName</tt>.
     *
     * @return a new process of the correct class.
     */
    public static UserProcess newUserProcess() {
        return (UserProcess)Lib.constructObject(Machine.getProcessClassName());
    }

    /**
     * Execute the specified program with the specified arguments. Attempts to
     * load the program, and then forks a thread to run it.
     *
     * @param name the name of the file containing the executable.
     * @param args the arguments to pass to the executable.
     * @return <tt>true</tt> if the program was successfully executed.
     */
    public boolean execute(String name, String[] args) {
        if (!load(name, args))
            return false;

        new UThread(this).setName(name).fork();

        return true;
    }
}

```

大部分方法均有相应的源码注解

图 4 方法在源码上的注解

2. 环境配置

1) Nachos 的 eclipse 配置

我们可以借助强大的工具——eclipse 对 Nachos 进行开发，只需要完成下列三个步骤即可完成配置。

第一步，首先创建一个文件夹，名字可以自定义，在这里将其命名为 nachos-java。然后从官网中下载 Nachos 的压缩包，将压缩包解压到新创建的文件夹中。进入官网 <https://www-inst.eecs.berkeley.edu/~cs162/sp06/>，点击左侧目录栏中

的 Projects and Nachos 选项进入项目页面，然后点击 nachos-java.tar.gz 便可下载。

第二步，解压缩会得到一个名为 nachos 的文件夹，将 nachos 文件夹内的 nachos.conf 文件复制一份到 nachos-java 文件夹中，得到目录结构如下图所示。

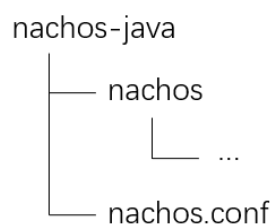


图 5 目录结构

第三步，利用 eclipse，使用打开 java 项目的方式打开 nachos-java 文件夹。然后进入到 nachos.machine 包的 Machine 类中执行其 main 函数。如果出现如下图所示的输出结果，代表配置基本完成。

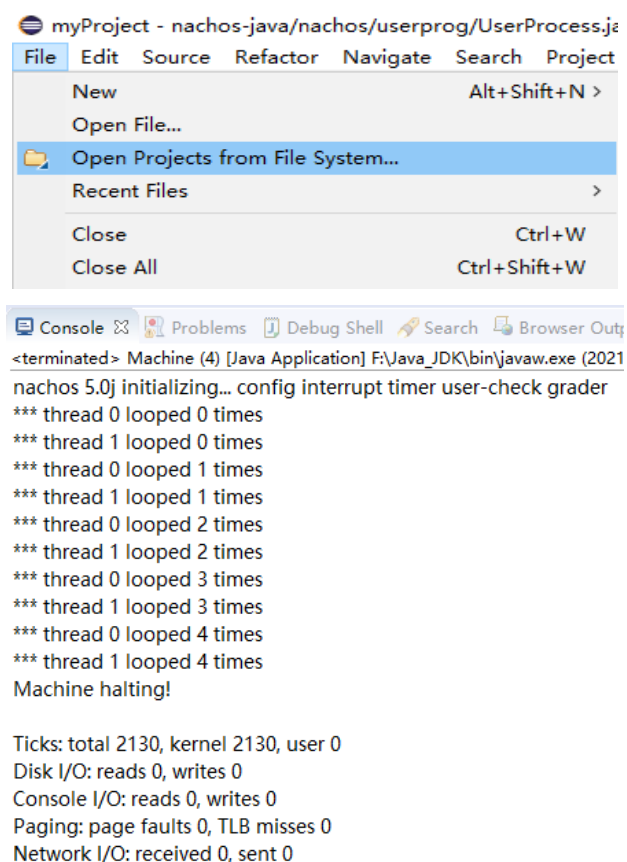


图 6 打开 java 项目的方式以及输出的结果

2) 交叉编译 c 文件

I.概述

在后续的实验中，我们需要编写一些 c 程序来让 nachos 执行，这些 c 程序需要借助交叉编译器编译成对应的.coff 文件才能在 nachos 操作系统中执行。下面我们便介绍这个编译的方法。

并且，在 nachos-java/nachos/test 目录结构中，nachos 的设计者早已经编写好了一些可供我们测试使用的 c 程序。其中有一个名为 sh.coff 的程序，该程序很好的模拟了操作系统的 shell 的功能。我们可以先运行 sh.coff 程序，在其基础上再打开其他的程序进行测试。

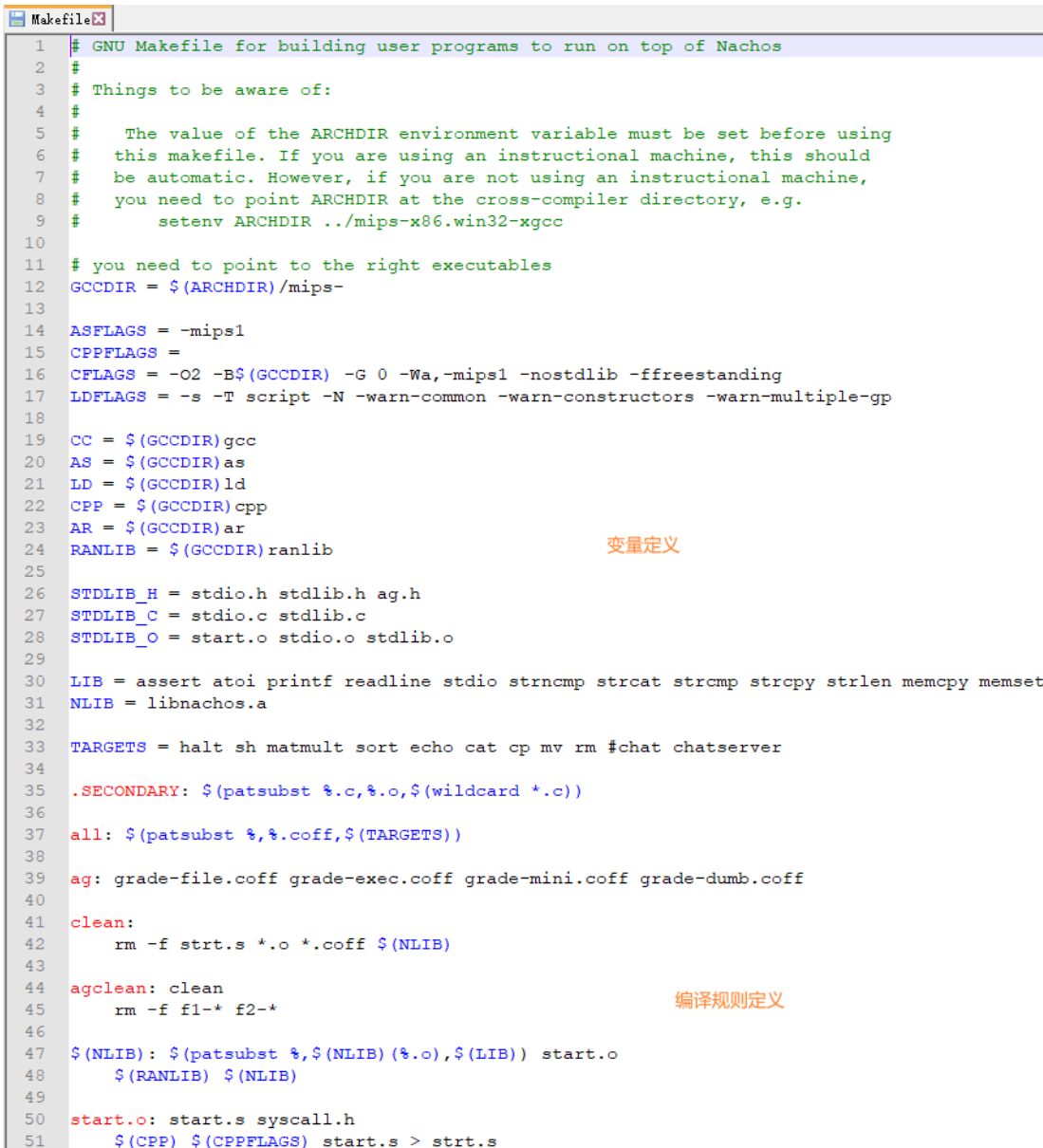
II.交叉编译简单介绍

首先介绍什么是交叉编译。交叉编译是在一个平台上生成另一个平台上的可执行代码。这样，我们在 windows 操作系统中便可以编译出适合在 nachos 运行的 MIPS 架构的目标程序。

III.Makefile 简单介绍

Makefile 文件的作用是定义了一系列的规则来指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至进行更复杂的功能操作。因此 makefile 文件就像一个含有多条编译指令的 Shell 脚本，可以同时执行多条 shell 指令，从而完成一系列的编译。采用 GUN Make 项目构建工具的 make 指令，便能对 Makefile 文件进行解析，然后对整个项目进行编译。

test 文件夹中 Makefile 文件定义的编译规则是对 test 文件夹的所有.c 和.s 文件进行编译。因此，我们只需要将我们写好的 c 程序放在 test 文件夹中，然后对 test 文件夹进行 make 构建，便可对我们的 c 程序进行编译。由于 Makefile 语法并不是重点，在此不详细阐述，有兴趣可以通过网上的一些博客查询一些教程。test 文件夹的 Makefile 文件如下图所示，主要分为变量定义和编译指令规则制定两部分。



```

1 # GNU Makefile for building user programs to run on top of Nachos
2 #
3 # Things to be aware of:
4 #
5 #   The value of the ARCHDIR environment variable must be set before using
6 #   this makefile. If you are using an instructional machine, this should
7 #   be automatic. However, if you are not using an instructional machine,
8 #   you need to point ARCHDIR at the cross-compiler directory, e.g.
9 #       setenv ARCHDIR ../mips-x86.win32-xgcc
10
11 # you need to point to the right executables
12 GCCDIR = $(ARCHDIR)/mips-
13
14 ASFLAGS = -mips1
15 CPPFLAGS =
16 CFLAGS = -O2 -B$(GCCDIR) -G 0 -Wa,-mips1 -nostdlib -ffreestanding
17 LDFLAGS = -s -T script -N -warn-common -warn-constructors -warn-multiple-gp
18
19 CC = $(GCCDIR)gcc
20 AS = $(GCCDIR)as
21 LD = $(GCCDIR)ld
22 CPP = $(GCCDIR)cpp
23 AR = $(GCCDIR)ar
24 RANLIB = $(GCCDIR)ranlib
25
26 STDLIB_H = stdio.h stdlib.h ag.h
27 STDLIB_C = stdio.c stdlib.c
28 STDLIB_O = start.o stdio.o stdlib.o
29
30 LIB = assert atoi printf readline stdio strncmp strcat strcmp strcpy strlen memcpy memset
31 NLIB = libnachos.a
32
33 TARGETS = halt sh matmult sort echo cat cp mv rm #chat chatserver
34
35 .SECONDARY: $(patsubst %.c,%.o,$(wildcard *.c))
36
37 all: $(patsubst %,%.coff,$(TARGETS))
38
39 ag: grade-file.coff grade-exec.coff grade-mini.coff grade-dumb.coff
40
41 clean:
42     rm -f strt.s *.o *.coff $(NLIB)
43
44 agclean: clean
45     rm -f f1-* f2-*
46
47 $(NLIB): $(patsubst %, $(NLIB) (%.o), $(LIB)) start.o
48     $(RANLIB) $(NLIB)
49
50 start.o: start.s syscall.h
51     $(CPP) $(CPPFLAGS) start.s > strt.s

```

图 7 test 文件夹的 Makefile 文件

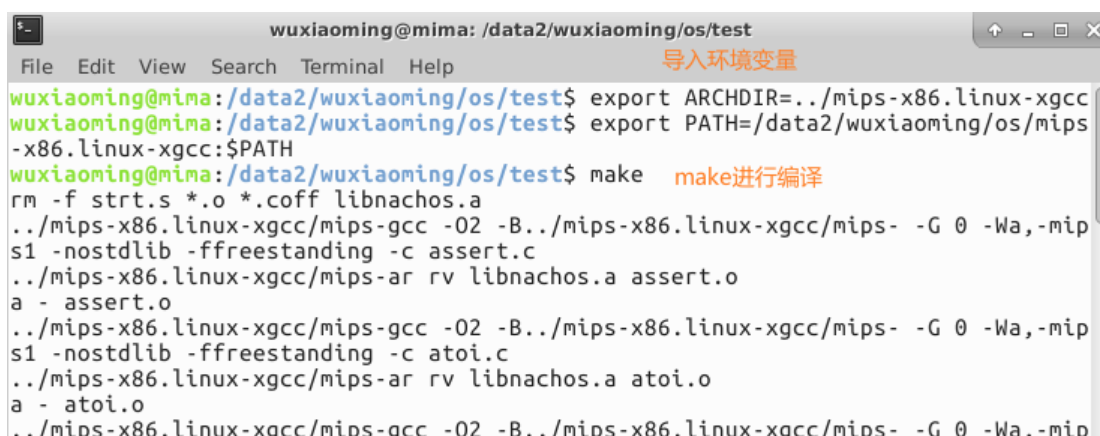
IV. 编译 c 文件

当我们编写好一个 c 文件后 (ps: nachos 不支持浮点数运算以及 new 的操作), 需要用交叉编译器将其编译为 coff 文件才能在 nachos 中运行。下面我们将介绍编译的步骤, 主要分为三步。

第一步, 从官网中下载交叉编译器。首先进入官网, 点击左侧目录栏中的 Projects and Nachos 选项进入项目页面, 从 MIPS Cross-compilers 栏目下中点击 this page 选项进入交叉编译器下载页面, 下载 x86.linux 版本的交叉编译器。

第二步, 配置 linux 的编译环境。交叉编译需要在 linux 系统下进行, 所以我们需要将 test 文件夹以及交叉编译器全部复制到 linux 系统中。

第三部, 编译。首先在 test 文件夹中打开 Terminal 进行, 然后进行下面的两小步, 第一步导入环境变量, 将交叉编译器的地址设置为环境变量; 第二步运行 make 指令进行编译。编译的具体指令如下图所示。



```
wuxiaoming@mima: /data2/wuxiaoming/os/test
File Edit View Search Terminal Help 导入环境变量
wuxiaoming@mima:/data2/wuxiaoming/os/test$ export ARCHDIR=./mips-x86.linux-xgcc
wuxiaoming@mima:/data2/wuxiaoming/os/test$ export PATH=/data2/wuxiaoming/os/mips-x86.linux-xgcc:$PATH
wuxiaoming@mima:/data2/wuxiaoming/os/test$ make make进行编译
rm -f strt.s *.o *.coff libnachos.a
./mips-x86.linux-xgcc/mips-gcc -O2 -B./mips-x86.linux-xgcc/mips- -G 0 -Wa,-mips1 -nostdlib -ffreestanding -c assert.c
./mips-x86.linux-xgcc/mips-ar rv libnachos.a assert.o
a - assert.o
./mips-x86.linux-xgcc/mips-gcc -O2 -B./mips-x86.linux-xgcc/mips- -G 0 -Wa,-mips1 -nostdlib -ffreestanding -c atoi.c
./mips-x86.linux-xgcc/mips-ar rv libnachos.a atoi.o
a - atoi.o
./mips-x86.linux-xgcc/mips-qcc -O2 -B./mips-x86.linux-xgcc/mips- -G 0 -Wa,-mip
```

图 8 编译的具体指令

3. Nachos 源码解读

1) 快速开始

I.项目文件的目录结构

首先我们先来了解下整个项目的目录结构，如下图所示。



图 9 项目目录结构

II.项目配置文件

在项目中，除了不同的包之外，还有一个称为 nachos.conf 的文件，该文件定义了 nachos 的总体配置文件。在 nachos 启动时，会自动加载该文件，然后导入基础的配置信息，模拟了真实操作系统中的 BIOS。

下面我们对 nachos.conf 文件的一些配置可选项进行一个解释，如下图所示。注意，并不是所有的配置都是必须的，有些配置可以不需要，也可以通过 main 函数的参数的形式提供给 nachos。

```

1 Machine.stubFileSystem = true
2 # 是否使用文件系统
3
4 Machine.processor = true
5 # 是否使用处理器来执行用户程序
6
7 Machine.console = true
8 # 是否使用控制台
9
10 Machine.disk = false
11 Machine.bank = false
12
13 Machine.networkLink = true
14 # 是否使用网络
15
16 ElevatorBank.allowElevatorGUI = true
17 NachosSecurityManager.fullySecure = false
18
19 ThreadedKernel.scheduler = nachos.threads.RoundRobinScheduler
20 # 使用哪一个调度程序类文件，不同实验则用该实验文件夹内的那个
21
22 Kernel.kernel = nachos.network.NetKernel
23 # 使用哪一个Kernel类文件，不同实验则用该实验文件夹内的那个
24
25 Kernel.processClassName = nachos.network.NetProcess
26 # 使用哪一个Process类文件，不同实验则用该实验文件夹内的那个
27
28 NetworkLink.reliability = 1.0
29 # 网络通讯的可靠度（从0到1），小于1会有信息丢失
30
31 Processor.numPhysPages = 12
32 # 有多少物理页（内存大小为多少页）
33
34 Kernel.shellProgram = sh.coff
35 # shell程序名称
36
37 FileSystem.testDirectory =
38 "D:\\Information\\projects\\Small Operation System\\nachos-java\\nachos\\test"
39 # test文件夹的地址

```

图 10 nachos.conf 文件的配置

III. 启动程序

配置完成后，我们可以从 Machine 的 main 函数启动 nachos 操作系统，下面我们将对 main 函数进行解读，进而理解内核整个启动的过程。如下图。

```

14 public final class Machine {
15     // nachos启动函数
16     public static void main(final String[] args) {
17         System.out.print("nachos 5.0j initializing...");
18
19         Lib.assertTrue(Machine.args == null);
20         Machine.args = args;
21         // 加载启动函数
22         processArgs();

```

```

24 Config.load(configFileName);
25 // 将地址保存为变量
26 //baseDirectory为项目地址
27 //nachosDirectory为nachos文件夹的地址
28 //testDirectoryName为test文件夹的地址
29 baseDirectory = new File(new File("").getAbsolutePath());
30 nachosDirectory = new File(baseDirectory, "nachos");
31 String testDirectoryName =
32     Config.getString("FileSystem.testDirectory");
33 if (testDirectoryName != null) {
34     testDirectory = new File(testDirectoryName);
35 }
36 else {
37     testDirectory = new File(baseDirectory.getParentFile(), "test");
38 }
39
40 //配置安全管理机制
41 securityManager = new NachosSecurityManager(testDirectory);
42 privilege = securityManager.getPrivilege();
43 privilege.machine = new MachinePrivilege();
44 TCB.givePrivilege(privilege);
45 privilege.stats = stats;
46 securityManager.enable();
47
48 //创建并初始化所有设备
49 createDevices();
50 checkUserClasses();
51
52 //加载ag模块
53 autoGrader = (AutoGrader) Lib.constructObject(autoGraderClassName)
54
55 //创建TCB，在这个线程中执行autoGrader.start()函数
56 //在autoGrader.start()函数中将会启动内核
57 new TCB().start(new Runnable() {
58     public void run() { autoGrader.start(privilege); }
59 });
60 }
61

```

图 11 Machine.java 的 main 函数详解

可以看到 main 函数最后会通过 new TCB().start()开启一个新的内核线程，并且在这个内核线程内，将运行 Autograder 类下面的 start()函数，从而启动内核。因此下面我们再简要解读下 Autograder 类的 start()函数。注意：Autograder 类在 nachos.ag 包下面。


```

31 public void start(Privilege privilege) {
32     Lib.assertTrue(this.privilege == null,
33         "start() called multiple times");
34     this.privilege = privilege;
35     String[] args = Machine.getCommandLineArguments();
36     extractArguments(args);
37     System.out.print(" grader");
38     init();
39     System.out.print("\n");
40
41     kernel =
42         (Kernel) Lib.constructObject(Config.getString("Kernel.kernel"));
43     //初始化内核
44     kernel.initialize(args);
45     //启动内核, 分别对内核进行自测, 运行, 关闭
46     run();
47 }
48
147 void run() {
148     kernel.selfTest();
149     kernel.run();
150     kernel.terminate();
151 }

```

图 12 Autograde.java 的 start 以及 run 函数详解

最后，在 run() 函数运行内核的过程中会根据 nachos.conf 内的配置不同而执行不同的内核函数。在原始的配置中，内核是 nachos.threads.ThreadedKernel，因此会分别执行 nachos.threads 包下的 ThreadedKernel 类的 selfTest()、run()、terminate() 函数，最终在 ThreadedKernel 类的 selfTest() 函数中运行 PingTest() 函数，会输出前面展示的配置成功后的那些输出信息。

2) 内核线程与 TCB

前面我们了解到了 Nachos 操作系统的启动程序是如何一步步将 Nachos 启动的整个过程。并且，从中我们还看到 Nachos 操作系统的启动最后是通过创建一个 TCB() 从而创建一个内核线程，然后在该内核线程内部将操作系统启动起

来的。在这一节中，我们将深入了解一下内核线程以及 TCB 的内在运行机理。

I. 内核线程

内核线程是 nachos.thread 包下的 KThread 类，该类管理着内核线程的各种函数，包括内核线程的状态变化函数，一些用于线程创建以及调度线程执行顺序的函数等等。在这一小节，我们将了解下内核线程的一些主要函数。

首先我们来看看内核线程的状态变化函数，我们在此仅介绍下如何通过一些方法去改变内核线程的状态，如下图所示，这些方法在实验中将会使用。具体这些方法的内部实现借助了 TCB 类的一些函数，下一小节我们将介绍 TCB 的函数，当理解了 TCB 类后，这些方法的实现将很容易理解。

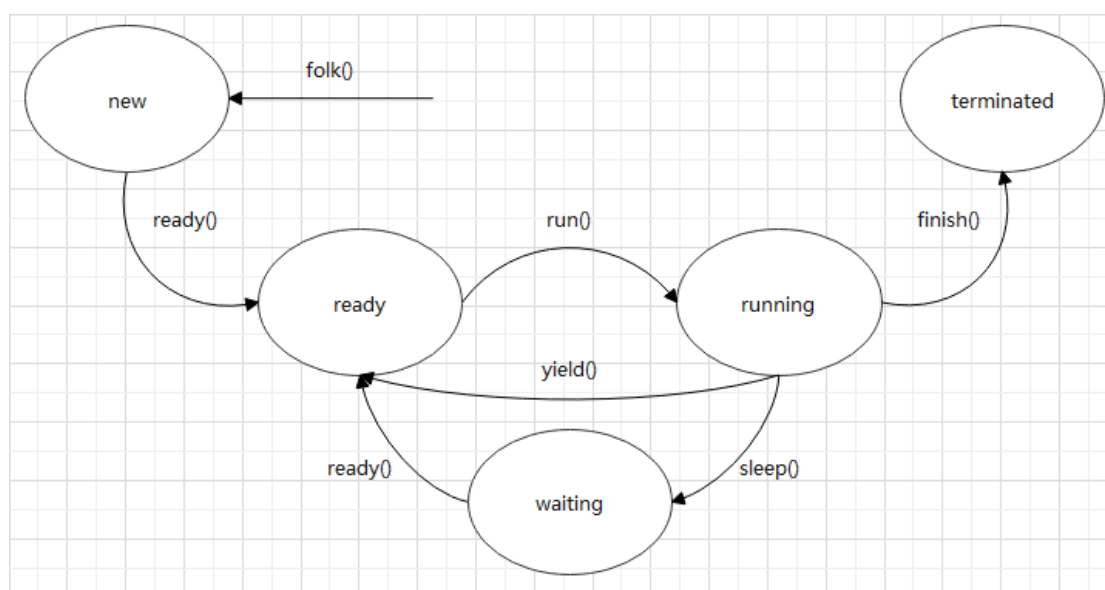


图 13 内核线程的状态变化以及导致变化的函数

另外，内核线程类中还有一个注意点，那便是等待队列。内核线程中有一个名为 readyQueue 的等待队列，所有 ready 状态的线程都在该等待队列中等待，该队列继承自同样在 nachos.thread 包下的 ThreadQueue 类，该类有几个方法，其中 waitForAccess(KThread) 方法用于将一个内核线程加入该队列。而 nextThread()

方法用于调度下一个线程，后面的一些调度算法的实验将需要的不同实现。

II.线程控制块

线程控制块是 nachos.machine 的 TCB 类，该类模拟了硬件对内核线程控制块的各种控制，包括内核控制块的启动、上下文交换等。

首先，应该了解到 TCB 背后是借助 java 线程实现的。每个 TCB 都对应着一个真正的 java 线程，存于 TCB 类的 javaThread 变量中。TCB 的等待和唤醒也是依靠着 javaThread 来实现的。如下图所示。因此，前面内核线程 KThread 类的状态改变的本质都是 java 线程的 wait()和 notify()。

```

263 //当前TCB等待唤醒，且在唤醒时检测是否已经被destory
264 //用于contextSwitch()以及threadroot()中
265 private void yield() {
266     waitForInterrupt();
267     if (done) {
268         currentTCB.interrupt();
269         throw new ThreadDeath();
270     }
271     currentTCB = this;
272 }
273
274 //当前TCB等待唤醒
275 private synchronized void waitForInterrupt() {
276     while (!running) {
277         try { wait(); } //利用了java线程的wait()实现
278         catch (InterruptedException e) {}
279     }
280 }
281
282 //唤醒当前TCB
283 private synchronized void interrupt() {
284     running = true;
285     notify(); //利用了java线程的notify()实现
286 }
287

```

图 14 TCB 底层利用 java 线程的方法实现等待和唤醒

然后，再来看看 TCB 的一些方法。TCB 有一个 start(Runnable target)方法，该

方法将启动 TCB，最终启动的 TCB 会创建一个与该 TCB 对应的 java 线程，在该 java 线程中执行 start 的参数中的 target 的 run() 方法。另外 TCB 还有 contextSwitch 方法，用于模拟上下文切换。具体代码解释如下图。因此，前面 KThread 类的创建函数 folk() 的实现本质便是调用了 TCB 的 start() 方法，而那些状态改变的方法中有一些也借助了 TCB 的 contextSwitch() 方法来实现。

```

45 //启动TCB
46 public void start(Runnable target) {
47 //各种检测
48 Lib.assertTrue(javaThread == null && !done);
49 Lib.assertTrue(runningThreads.size() < maxThreads);
50 isFirstTCB = (currentTCB == null);
51 if (!isFirstTCB)
52     Lib.assertTrue(currentTCB.javaThread == Thread.currentThread());
53
54 //将TCB加入runningThreads容器中
55 runningThreads.add(this);
56
57 this.target = target;
58
59 //无论是否是第一个TCB，最终都是执行threadroot()函数
60 //threadroot()函数最终会执行 target.run()
61 if (!isFirstTCB) {
62     tcbTarget = new Runnable() {
63         public void run() { threadroot(); }
64     };
65     privilege.doPrivileged(new Runnable() {
66         public void run() { javaThread = new Thread(tcbTarget); }
67     });
68     currentTCB.running = false;
69     this.javaThread.start();
70     currentTCB.waitForInterrupt();
71 }
72 else {
73     javaThread = Thread.currentThread();
74     threadroot();
75 }
76 }

```

图 15 TCB.start()函数

```

85 //上下文切换
86 public void contextSwitch() {
87 //各种检测
88 Lib.assertTrue(currentTCB.javaThread == Thread.currentThread());
89 Lib.assertTrue(currentTCB.associated);
90 currentTCB.associated = false;
91 if (this == currentTCB)
92     return;
93
94 //将正在执行的TCB陷入等待，将调用该方法的TCB唤醒
95 TCB previous = currentTCB;
96 previous.running = false;
97
98 this.interrupt();
99 previous.yield();
100 }

```

图 16 TCB 的 contextSwitch() 函数

III. 懒惰线程与无限前进机制

在内核线程 KThread 的构造函数中，我们可以看到有一个 createIdleThread 方法，该方法会创建一个懒惰线程，这个线程在一般的情况下并不会执行，但是一旦所有的内核线程都陷入等待时（也就是就绪队列上没有内核线程时），nachos 便会调度这个懒惰线程。

这个懒惰线程存在的目的是保证系统时钟的前进，后面我们会讲到，nachos 的时钟只会在两种情况下前进，第一种是关中断后重新开中断，另一种是执行一条用户程序的指令。因此，当 nachos 中所有的内核线程都陷入等待时，便没有机制可以让 nachos 的时钟前进，而一旦 nachos 的时钟不前进，那么等待的事件将永远处于等待，从而导致死锁。因此，nachos 中引入了懒惰线程的机制，保证当所有的内核线程都陷入等待时，仍有一个懒惰线程在执行，从而保证时钟继续前进。

3) 中断机制

前面第二节主要讲了 nachos 如何模拟操作系统的内核线程运行机理，在这一节中，将介绍 nachos 对中断机制的模拟。

1. 中断机制

中断机制的实现，最关键的便是 nachos.machine 包下的中断类 Interrupt。该类简要模拟了计算机的中断机制。

首先，Interrupt 类内有一个名为 pending 的容器，在该容器内，存着所有将要发生的中断（包括该中断的名字，中断将要发生的时间，以及中断的处理事件）。内核的硬件可以通过 Interrupt 类的 schedule 方法将中断加入 pending 容器中。

然后，Interrupt 类内有一个 tick()方法，该方法模拟计算机时钟的跳动。该方法会在两个地方被调用，第一个是 Interrupt 类的 setStatus 方法中，该方法会检测此时是否是从关中断到开中断，如果是，那么就会执行 tick()，第二个是在 Processor 类的 run()方法中，每执行一条指令便会执行一次 tick()方法。

```
88 public boolean setStatus(boolean status) {  
89     boolean oldStatus = enabled;  
90     enabled = status;  
91     //如果以前是关中断而现在是开中断，则tick()  
92     if (oldStatus == false && status == true)  
93         tick(true);  
94     return oldStatus;  
95 }
```

图 17 Interrupt 类的 setStatus 中调用 tick()

```

87 public void run() {
88     Lib.debug(dbgProcessor, "starting program in current thread");
89     //模拟pc=pc+1的硬件过程, 代码中+4是因为一个指令4个字节
90     registers[regNextPC] = registers[regPC] + 4;
91     Machine.autoGrader().runProcessor(privilege);
92     Instruction inst = new Instruction();
93     //循环执行每一条指令 (取指令, 解析指令, 执行指令)
94     while (true) {
95         try {
96             inst.run();
97         }
98         catch (MipsException e) {
99             e.handle();
100         }
101         //每次执行完instruction调用tick()
102         privilege.interrupt.tick(false);
103     }
104 }

```

图 18 Processor 类的 run()方法解读与调用 tick()

另外, tick()方法的每次执行, 都会调用 checkIfDue 方法, 该方法会检测 pending 容器中有哪些中断是已经到了其触发中断的时间, 将那些到了时间的中断取出并执行其中断处理函数。

```

128 //tick时钟前进
129 private void tick(boolean inKernelMode) {
130     Stats stats = privilege.stats;
131     //不同情况下前进的时钟数目不同
132     //inKernelMode==true指的是在关中断后开中断的时候tick, 此时前进10个系统时钟
133     //inKernelMode==false指的是在执行了一条用户程序的指令之后tick, 此时前进1个系统时钟
134     if (inKernelMode) {
135         stats.kernelTicks += Stats.KernelTick;
136         stats.totalTicks += Stats.KernelTick;
137     }
138     else {
139         stats.userTicks += Stats.UserTick;
140         stats.totalTicks += Stats.UserTick;
141     }
142     if (Lib.test(dbgInt))
143         System.out.println("== Tick " + stats.totalTicks + " ==");
144     enabled = false;
145     //调用checkIfDue检查中断
146     checkIfDue();
147     enabled = true;
148 }

```

图 19 tick 函数解读

```

150 private void checkIfDue() {
151     //检测一些东西
152     long time = privilege.stats.totalTicks;
153     Lib.assertTrue(disabled());
154     if (Lib.test(dbgInt))
155         print();
156     if (pending.isEmpty())
157         return;
158     if (((PendingInterrupt) pending.first()).time > time)
159         return;
160     Lib.debug(dbgInt, "Invoking interrupt handlers at time = " + time);
161
162     //从pending容器中取出中断, 并且将已经到了触发事件的中断触发
163     while (!pending.isEmpty() &&
164         ((PendingInterrupt) pending.first()).time <= time) {
165         PendingInterrupt next = (PendingInterrupt) pending.first();
166         pending.remove(next);
167         Lib.assertTrue(next.time <= time);
168         if (privilege.processor != null)
169             privilege.processor.flushPipe();
170         Lib.debug(dbgInt, " " + next.type);
171
172         next.handler.run(); //调用中断的处理函数
173     }

```

图 20 checkIfDue 函数解读

II. 时钟中断

时钟中断主要由 nachos.machine 包的 Timer 类来模拟实现。

在 Timer 类创建开始, 便会调用 scheduleInterrupt 方法, 将时钟中断加入到 pending 中进行等待, 中断发生时, 会调用中断处理函数 timerInterrupt, 中断处理函数会再一次将时钟中断加入到 pending 中, 如此循环, 从而实现时钟中断。

```

74 private void scheduleInterrupt() {
75     int delay = Stats.TimerTicks;
76     delay += Lib.random(delay/10) - (delay/20);
77     //将时钟中断加入pending中等待
78     //500个系统时钟后执行中断, 中断处理函数为timerInterrupt
79     privilege.interrupt.schedule(delay, "timer", timerInterrupt);
80 }

```

图 21 scheduleInterrupt() 方法解读


```

62 private void timerInterrupt() {
63     //再次将中断加入pending中，从而执行下一次中断
64     scheduleInterrupt();
65     scheduleAutoGraderInterrupt();
66
67     lastTimerInterrupt = getTime();
68
69     //执行handler的run()方法，从而可以实现一些自定义功能
70     if (handler != null)
71         handler.run();
72 }

```

图 22 timerInterrupt() 方法解读

4) 用户程序

前面第三节介绍了 nachos 模拟中断的机制，在第四节，将介绍 nachos 在内核启动后，如何加载用户程序并且执行的内在机理。

I. 用户程序的加载与运行

首先我们来看看内核如何一步步的启动我们的用户程序的。我们可以看到，在 nachos.userprog 包的 UserKernel 类的 run()方法中，该方法不再像之前的 ThreadKernel 一样什么都不执行，而是在内核启动后，建立一个用户进程来执行 shell 用户程序。注意，shell 程序的地址需要我们在 nachos.conf 中设置。

```

92 public void run() {
93     super.run();
94     //创建一个用户进程
95     UserProcess process = UserProcess.newUserProcess();
96     //在该用户进程中执行shell用户程序
97     String shellProgram = Machine.getShellProgramName();
98     Lib.assertTrue(process.execute(shellProgram, new String[] { }));
99
100     KThread.currentThread().finish();
101 }
102

```

图 23 UserKernel 类的 run()方法解读

然后，我们来看看用户进程 `UserProcess` 执行用户程序的方法 `execute()` 的内部是怎么实现的——加载 `coff` 文件，然后创建用户线程来启动处理器。具体解读如下图所示。

```

51 public boolean execute(String name, String[] args) {
52     //load加载用户程序coff文件到内存
53     if (!load(name, args))
54         return false;
55     //创建一个用户线程
56     //UThread内会让处理器执行，从而进入到取指令执行指令的过程
57     new UThread(this).setName(name).fork();
58
59     return true;
60 }

```

图 24 `UserProcess` 类的 `execute` 方法解读

```

15 public UThread(UserProcess process) {
16     super();
17     //执行runProgram(), 而runProgram方法会让处理器开始运行
18     setTarget(new Runnable() {
19         public void run() {
20             runProgram();
21         }
22     });
23     this.process = process;
24 }
25
26 private void runProgram() {
27     //初始化处理器的寄存器值
28     process.initRegisters();
29     process.restoreState();
30     //让处理器开始运行，从而不断取指令，执行指令
31     Machine.processor().run();
32     Lib.assertNotReached();
33 }

```

图 25 `UThread` 的构造函数解读（启动处理器）

最后，我们再来看看 `load` 方法是如何加载用户程序到内存的，`load` 方法的理解对于后面的页表、懒加载等实验都十分有帮助。解读如下图所示。

```

191 private boolean load(String name, String[] args) {
192     Lib.debug(dbgProcess, "UserProcess.load(\"" + name + "\");");
193     OpenFile executable = ThreadedKernel.fileSystem.open(name, false);
194     if (executable == null) {
195         Lib.debug(dbgProcess, "\topen failed");
196         return false;
197     }
198     //将用户程序coff加载成一个coff对象
199     try {
200         coff = new Coff(executable);
201     }
202     catch (EOFException e) {
203         executable.close();
204         Lib.debug(dbgProcess, "\tcoff load failed");
205         return false;
206     }
207     //coff文件内部是分段存储的，比如代码段，堆栈段等
208     //检测用户coff文件的每一段是否符合要求
209     numPages = 0;
210     for (int s=0; s<coff.getNumSections(); s++) {
211         CoffSection section = coff.getSection(s);
212         if (section.getFirstVPN() != numPages) {
213             coff.close();
214             Lib.debug(dbgProcess, "\tfragmented executable");
215             return false;
216         }
217         numPages += section.getLength();
218     }
219     //保证参数能在一页中存下
220     byte[][] argv = new byte[args.length][];
221     int argsSize = 0;
222     for (int i=0; i<args.length; i++) {
223         argv[i] = args[i].getBytes();
224         argsSize += 4 + argv[i].length + 1;
225     }
226     if (argsSize > pageSize) {
227         coff.close();
228         Lib.debug(dbgProcess, "\targuments too long");
229         return false;

```

图 26 load 方法的解读 1

```

231 // 初始化PC指针、堆栈指针的值
232 initialPC = coff.getEntryPoint();
233 numPages += stackPages;
234 initialSP = numPages*pageSize;
235 //留一页存储参数
236 numPages++;
237
238 //loadSection方法加载coff文件进入内存
239 if (!loadSections())
240     return false;
241
242 // 存储参数到最后一页
243 int entryOffset = (numPages-1)*pageSize;
244 int stringOffset = entryOffset + args.length*4;
245 this.argc = args.length;
246 this.argv = entryOffset;
247 for (int i=0; i<argv.length; i++) {
248     byte[] stringOffsetBytes = Lib.bytesFromInt(stringOffset);
249     Lib.assertTrue(writeVirtualMemory(entryOffset,stringOffsetBytes) == 4);
250     entryOffset += 4;
251     Lib.assertTrue(writeVirtualMemory(stringOffset, argv[i]) ==
252         argv[i].length);
253     stringOffset += argv[i].length;
254     Lib.assertTrue(writeVirtualMemory(stringOffset,new byte[] { 0 }) == 1);
255     stringOffset += 1;
256 }
257 return true;
258 }

```

图 27 load 方法的解读 2

```

260 protected boolean loadSections() {
261     //保证程序的页数不超出物理页范围
262     if (numPages > Machine.processor().getNumPhysPages()) {
263         coff.close();
264         Lib.debug(dbgProcess, "\tinsufficient physical memory");
265         return false;
266     }
267     //加载用户程序到内存
268     //这里的实现只是初始版本，后面的实验要求实现页表，需要修改该方法
269     for (int s=0; s<coff.getNumSections(); s++) {
270         CoffSection section = coff.getSection(s);
271         Lib.debug(dbgProcess, "\tinitializing " + section.getName()
272             + " section (" + section.getLength() + " pages)");
273         for (int i=0; i<section.getLength(); i++) {
274             int vpn = section.getFirstVPN()+i;
275             section.loadPage(i, vpn);
276         }
277     }
278     return true;
279 }

```

图 28 loadSection 方法的解读

II. 指令执行流程介绍

通过前面一小节我们知道，当一个用户程序加载后，会创建一个用户线程 Uthread，并且该在该线程内启动处理器。这一小节将看看处理器是怎么一步步执行我们用户程序的每一条指令的。

我们知道，Uthread 的构造方法最后会通过 Processor.run()方法启动处理器。通过图 18 Processor 类的 run()方法解读我们可以了解到处理器的指令执行流程，Processor.run()是通过循环执行 inst.run()来模拟处理器取指令、执行指令的循环过程的，Instruction 的 run()方法解读如下图所示。

```
606 public void run() throws MipsException {  
607     fetch(); //取指令  
608     decode(); //解析  
609     execute(); //执行指令  
610     writeBack();  
611 }
```

图 29 Instruction 类的 run()方法解读

III. 系统调用陷入内核的机制

通过前面一小节我们知道处理器最终会不断循环取指令、执行指令的过程来执行用户程序的。在这一小节，我们将看看在执行指令的过程中，nachos 是如何模拟系统调用陷入内核的。

通过对执行指令的方法——Instruction 类的 execute()方法进行解析，我们发现 Instruction 类的 execute()是通过前面对指令进行解析后得到的 operation 变量的不同值，从而执行不同的操作。当前面的指令是一个系统调用时，则会抛出异常，异常会在 UserProcess 那一层被捕捉然后处理，处理的程序则是执行响应的系统调用函数。

```

842⊖ private void execute() throws MipsException {
843     int value;
844     int preserved;
845     //根据operation的不同值，从而指令不同的操作
846     switch (operation) {
847         case Mips.ADD:
848             dst = src1 + src2;
849             break;
850     case Mips.SUB:
851         dst = src1 - src2;
852         break;
853     case Mips.MULT:
854         dst = src1 * src2;
855         registers[regLo] = (int) Lib.extract(dst, 0, 32);
856         registers[regHi] = (int) Lib.extract(dst, 32, 32);
857
858         //指令是系统调用，抛出异常
859         //异常会在UserProcess那一层被捕捉和处理
860     case Mips.SYSCALL:
861         throw new MipsException(exceptionSyscall);
862     }
863 }

```

图 30 Instruction 类的 execute()方法：执行指令的过程

```

375 //异常处理函数
376⊖ public void handleException(int cause) {
377     Processor processor = Machine.processor();
378     switch (cause) {
379         //该异常是系统调用
380         case Processor.exceptionSyscall:
381             //因此调用相应的系统调用函数
382             int result = handleSyscall(processor.readRegister(Processor.regV0,
383                 processor.readRegister(Processor.regA0,
384                 processor.readRegister(Processor.regA1,
385                 processor.readRegister(Processor.regA2,
386                 processor.readRegister(Processor.regA3
387                 ));
388             processor.writeRegister(Processor.regV0, result);
389             processor.advancePC();
390             break;
391
392         default:
393             Lib.debug(dbgProcess, "Unexpected exception: " +
394                 Processor.exceptionNames[cause]);
395             Lib.assertNotReached("Unexpected exception");
396     }
397 }
398 }

```

图 31 异常处理函数以及处理系统调用异常的过程

IV.增添自定义的系统调用

在这一小节我们来看看如何自己添加一个系统调用。根据上一小节的经验，只要我们自己编写的 `coff` 程序执行一个系统调用的指令，那么这个系统调用最终便会在 Nachos 中被抛出异常，最后被捕捉处理。

首先，我们来看看如何定义相应的系统调用函数，在 `test` 文件夹里面有一个名为 `syscall.h` 的文件，该文件里定义了用户程序内可以使用的的一系列系统调用函数以及相应的系统调用号（该号在 nachos 里使用，用于辨别是哪一个系统调用）。如下图所示。

知道了系统调用函数以及系统调用号的定义位置后，下面我们来看看如何将系统调用函数转化为真正的系统调用指令，并且将系统调用号也告知操作系统。这部分工作在一个称为 `start.s` 的程序中完成，该程序是一个汇编程序，里面主要分为两部分，第一部分的代码规定了我们编写的 `c` 程序中 `main` 函数是整个程序的入口，且保证了 `main` 函数完成后最后会执行退出函数 `exit()`；第二部分是通过预编译的方法，将系统的每个调用函数都转化为真正的系统调用指令，并且将系统调用号也传到相应的寄存器，nachos 通过对寄存器上的参数值进行解析，便可以知道触发了哪一个系统调用。如下图所示。

因此，要想增添自定义的系统调用，只需要先在 `syscall.h` 中定义好系统调用函数以及其系统调用号，然后在 `start.s` 的末尾加上该系统调用的预编译即可。这样操作之后，我们自定义的系统调用函数最终便会触发真正的系统调用，然后在执行到这一指令时，便会在 Nachos 中出现异常，我们对异常进行捕捉，再对我们

自定义的系统调用进行我们自定义的处理即可。

```

8
9 #ifndef SYSCALL_H
10 #define SYSCALL_H
11
12 #define syscallHalt      0
13 #define syscallExit      1
14 #define syscallExec      2
15 #define syscallJoin      3
16 #define syscallCreate     4
17 #define syscallOpen      5
18 #define syscallRead      6
19 #define syscallWrite      7
20 #define syscallClose      8
21 #define syscallUnlink     9
22 #define syscallMmap     10
23 #define syscallConnect   11
24 #define syscallAccept    12
25
26 #ifndef START_S
27 #define fdStandardInput  0
28 #define fdStandardOutput 1
29
30 void halt();
31
32 void exit(int status);
33
34 int exec(char *file, int argc, char *argv[]);
35
36 int join(int processID, int *status);
37
38 int creat(char *name);
39
40 int open(char *name);
41
42 int read(int fileDescriptor, void *buffer, int count);
43
44 int write(int fileDescriptor, void *buffer, int count);
45
46 int close(int fileDescriptor);
47
48 int unlink(char *name);
49
50 int mmap(int fileDescriptor, char *address);
51
52 int connect(int host, int port);
53
54 int accept(int port);

```

系统调用号的定义

系统调用函数的定义

图 32 syscall.h 文件对系统调用的定义


```

/* -----
 * __start
 *      Initialize running a C program, by calling "main".
 * -----
 */

        .globl  __start
        .ent    __start
__start:
        jal     main
        addu    $4,$2,$0
        jal     exit /* if we return from main, exit(return value) */
        .end    __start

        .globl  __main
        .ent    __main
__main:
        jr      $31
        .end    __main

#define SYSCALLSTUB(name, number) \
        .globl  name                ; \
        .ent    name                ; \
name:                                     ; \
        addiu    $2,$0,number        ; \
        syscall                               ; \
        j        $31                ; \
        .end    name

SYSCALLSTUB(halt, syscallHalt)
SYSCALLSTUB(exit, syscallExit)
SYSCALLSTUB(exec, syscallExec)
SYSCALLSTUB(join, syscallJoin)
SYSCALLSTUB(creat, syscallCreate)
SYSCALLSTUB(open, syscallOpen)
SYSCALLSTUB(read, syscallRead)
SYSCALLSTUB(write, syscallWrite)
SYSCALLSTUB(close, syscallClose)
SYSCALLSTUB(unlink, syscallUnlink)
SYSCALLSTUB(mmap, syscallMmap)
SYSCALLSTUB(connect, syscallConnect)
SYSCALLSTUB(accept, syscallAccept)

```

这一段代码将c程序的main函数作为入口，并且保证main函数运行结束后运行exit退出程序

将系统调用函数预编译成一段汇编程序，这段汇编程序的主要工作是将系统调用号传到相应的寄存器，然后触发syscall指令。

图 33 start.s 文件的简单解读

5) 内存读写

上一节，我们主要讲述了 nachos 对用户程序的加载和执行的一些相关内容。

在这一节，我们将看一看 nachos 中内存访问的相关内容。

I.物理内存的创建

在 nachos 中，物理内存是通过一个 byte 数组来模拟的，该内存会在处理器 Processor 类创建的时候顺带创建，并且会创建一个 translations 变量来模拟 TLB

或者页表（根据是否使用 TLB 的不同而不同）。

```

31 public Processor(Privilege privilege, int numPhysPages) {
32     System.out.print(" processor");
33     this.privilege = privilege;
34     privilege.processor = new ProcessorPrivilege();
35     Class<?> clsKernel = Lib.loadClass(Config.getString("Kernel.kernel"));
36     Class<?> clsVMKernel = Lib.tryLoadClass("nachos.vm.VMKernel");
37     usingTLB =
38         (clsVMKernel != null && clsVMKernel.isAssignableFrom(clsKernel));
39
40     //创建物理内存
41     this.numPhysPages = numPhysPages;
42     for (int i=0; i<numUserRegisters; i++)
43         registers[i] = 0;
44     mainMemory = new byte[pageSize * numPhysPages];
45
46     //如果使用TLB, translations则代表着TLB
47     //如果没有使用TLB, translations则代表着页表
48     //应该注意, 在上下文切换时, TLB应该设为无效, 页表应该换成另一个线程的页表
49     if (usingTLB) {
50         translations = new TranslationEntry[tlbSize];
51         for (int i=0; i<tlbSize; i++)
52             translations[i] = new TranslationEntry();
53     }
54     else {
55         //注意, 虽然在这里会设置为null
56         //但是在UserProcess中会将translations设置为自己的页表
57         translations = null;
58     }
59 }

```

图 34 Processor 类中内存的创建

II.处理器内部的内存读写

在处理器内部，读写内存主要通过私有方法 readMem(int, int)和 writeMem(int, int, int)来实现。在这两个方法内部主要通过 translate(int, int, boolean)方法来实现，translate(int, int, boolean)方法的主要功能是将虚拟地址转化为对应的物理地址。该函数的详解如下图所示。

```

278 //将虚拟地址转化为物理地址
279 private int translate(int vaddr, int size, boolean writing)
280 throws MipsException {
281 //各种检测
282 if (Lib.test(dbgProcessor))
283     System.out.println("\ttranslate vaddr=0x" + Lib.toHexString(vaddr)
284         + (writing ? ", write" : ", read..."));
285 if ((vaddr & (size-1)) != 0) {
286     Lib.debug(dbgProcessor, "\t\talignment error");
287     throw new MipsException(exceptionAddressError, vaddr);
288 }
289
290 //计算虚拟页以及页偏移
291 int vpn = pageFromAddress(vaddr);
292 int offset = offsetFromAddress(vaddr);
293
294 TranslationEntry entry = null;
295 // 如果不使用TLB, translations是页表, 则找出对应的页然后找出物理帧
296 if (!usingTLB) {
297     if (translations == null || vpn >= translations.length ||
298         translations[vpn] == null ||
299         !translations[vpn].valid) {
300         privilege.stats.numPageFaults++;
301         Lib.debug(dbgProcessor, "\t\tpage fault");
302         throw new MipsException(exceptionPageFault, vaddr);
303     }
304     entry = translations[vpn];
305 }
306 //若使用TLB, translations是TLB, 遍历TLB找出符合的页
307 else {
308     for (int i=0; i<tlbSize; i++) {
309         if (translations[i].valid && translations[i].vpn == vpn) {
310             entry = translations[i];
311             break;
312         }
313     }
314     if (entry == null) {
315         privilege.stats.numTLBMisses++;
316         Lib.debug(dbgProcessor, "\t\tTLB miss");
317         throw new MipsException(exceptionTLBMiss, vaddr);
318     }
319 }
320
321 // 再做一些合法性检查
322 if (entry.readOnly && writing) {
323     Lib.debug(dbgProcessor, "\t\tread-only exception");
324     throw new MipsException(exceptionReadOnly, vaddr);
325 }
326 int ppn = entry.ppn;
327 if (ppn < 0 || ppn >= numPhysPages) {
328     Lib.debug(dbgProcessor, "\t\tbad ppn");
329     throw new MipsException(exceptionBusError, vaddr);
330 }
331
332 //修改页表项的状态
333 entry.used = true;
334 if (writing)
335     entry.dirty = true;
336
337 int paddr = (ppn*pageSize) + offset;
338
339 if (Lib.test(dbgProcessor))
340     System.out.println("\t\tpaddr=0x" + Lib.toHexString(paddr));
341 return paddr;
342 }

```

图 35 translate 方法解读

III.进程的内存读写方法

另外，在用户进程 `UserProcess` 的层面，进程要想读写内存，则不能用硬件的方法，因此此时有另一套方法来进行内存读写，分别是 `readVirtualMemory` 和 `writeVirtualMemory`。这两个方法的初始版本并没有考虑页表，直接将虚拟内存和物理内存当成是同一个，因此直接对物理内存进行读写。在实验中需要使用页表时，我们需要对这两个方法进行改写。

6) 文件系统

前一节我们介绍了 nachos 的内存相关的部分，这一节将介绍 nachos 的文件系统。

I.文件系统的底层实现

Nachos 的文件系统主要是通过 `nachos.machine` 包下的 `StubFileSystem` 类来实现的。

该类对于文件系统的实现，底层主要是通过 java 本身的文件系统来实现的。在 `StubFileSystem` 类内部有一个私有类 `StubOpenFile`，该类模拟了一个打开文件，而该类对于文件的创建和读写，内部均借助了 java 本身的文件系统，如下图所示。而 `StubFileSystem` 类对于文件的操作，都是依赖 `StubOpenFile` 来实现的。

```

75  StubOpenFile(final String name, final boolean truncate)
76      throws IOException {
77      super(StubFileSystem.this, name);
78      //借助java的File类实现文件的创建
79      final File f = new File(directory, name);
80      //检测一些合法性并且维护一些变量的值
81      if (openCount == maxOpenFiles)
82          throw new IOException();
83  privilege.doPrivileged(new Runnable() {
84      public void run() { getRandomAccessFile(f, truncate); }
85  });
86      if (file == null)
87          throw new IOException();
88      open = true;
89      openCount++;
90  }

106 public int read(int pos, byte[] buf, int offset, int length) {
107     if (!open)
108         return -1;
109     try {
110         delay();
111         //借助java中File类的方法进行操作
112         file.seek(pos);
113         return Math.max(0, file.read(buf, offset, length));
114     }
115     catch (IOException e) {
116         return -1;
117     }
118 }

119
120 public int write(int pos, byte[] buf, int offset, int length) {
121     if (!open)
122         return -1;
123     try {
124         delay();
125         //借助java中File类的方法进行写操作
126         file.seek(pos);
127         file.write(buf, offset, length);
128         return length;
129     }
130     catch (IOException e) {
131         return -1;
132     }
133 }

```

图 36 StubOpenFile 的创建与读写

7) 网络机制

上一节中，我们介绍了 nachos 的文件系统的实现。在这一节中，我们将介绍 nachos 的网络机制。

I. 网卡机制

Nachos 通过 nachos.machine 包里的 NetworkLink 类实现对网卡的模拟。首先我们要知道, Nachos 底层是通过 java 的 socket 的 UDP 来实现对网卡的模拟的。每个 nachos 系统启动时，会创建一个属于自己的 UDP 的 socket, ip 地址是本机地址，端口号是 nachos 通过一个机制来自行分配，因此，不同的一个 nachos 系统之间，仅仅是本机的端口号不同，ip 地址是相同的，因此不同 nachos 系统之间收发信息，本质上仅仅是一个本机不同端口号的网络程序之间的通讯。相应的代码解读如下图所示。

```

62 public NetworkLink(Privilege privilege) {
63     System.out.print(" network");
64     this.privilege = privilege;
65     //获取localhost地址
66     try {
67         localhost = InetAddress.getLocalHost();
68     }
69     catch (UnknownHostException e) {
70         localhost = null;
71     }
72     Lib.assertTrue(localhost != null);
73     reliability = Config.getDouble("NetworkLink.reliability");
74     Lib.assertTrue(reliability > 0 && reliability <= 1.0);
75     //建立UDP的socket, IP地址是本地地址, 端口号是base端口号+link地址
76     socket = null;
77     for (linkAddress=0;linkAddress<Packet.linkAddressLimit;linkAddress++) {
78         try {
79             //用一个循环来创建socket
80             //通过寻找一个link地址, 去寻找一个没有被占用的端口号
81             socket = new DatagramSocket(portBase + linkAddress, localhost);
82             break;
83         }
84         catch (SocketException e) {
85         }
86     }
87     //检测socket的合法性
88     if (socket == null) {
89         System.out.println("");
90         System.out.println("Unable to acquire a link address!");
91         Lib.assertNotReached();
92     }
93     System.out.print("(" + linkAddress + ")");
94     //设置接受/发送的中断处理程序, 并且启动接受中断
95     receiveInterrupt = new Runnable() {
96         public void run() { receiveInterrupt(); }
97     };
98     sendInterrupt = new Runnable() {
99         public void run() { sendInterrupt(); }
100    };
101    scheduleReceiveInterrupt();
102    //新开一个thread用于不断检测是否有接受到的信息并存下来
103    Thread receiveThread = new Thread(new Runnable() {
104        public void run() { receiveLoop(); }
105    });
106    receiveThread.start();
107 }
108

```

图 37 NetworkLink 代码解读

但是 nachos 为了模拟真正的主机（既有 ip 地址也有主机端口号），nachos 采用了与主机端口号有关联的 link 地址作为每个 nachos 系统的网络地址，然后每

个 nachos 系统内部可以自定义不同的 nachos 端口号，每个信息包的发送必须注明源端和目标端的 link 地址和 nachos 端口号，并且，nachos 将这样的信息包封装成了一个 MailMessage，用 nachos.network 的 MailMessage 类表示。相应的关系如下图所示。

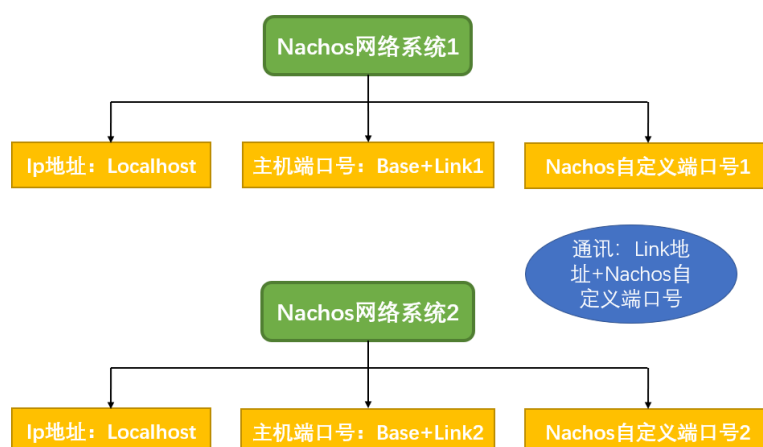


图 38 Nachos 网卡通信机制

II.网络收发机制

Nachos 通过 nachos.network 包里的 PostOffice 类实现对于网络内容的收发。并且其内部的实现借助了 NetworkLink 类内的一些方法来实现。

首先看看 Nachos 的接受机制，通过前面图 37 NetworkLink 代码解读的代码解读可以知道，NetworkLink 在创建时便启动了一个接受信息的线程，该线程执行 receiveLoop()的函数，不断循环接受信息并存起来。同时 NetworkLink 还启动了一个接受信息的中断处理程序，每过一段时间触发中断，中断处理函数会检测一下是否有信息被接收但是未处理，如果有，则调用 PostOffice 类内的 receiveInterrupt() 处理函数进行处理，该函数将接收到的信息按照相应的

Nachos 端口号放到一个与该 Nachos 端口号相关的队列中。相应的代码解读如下图所示。

```
190 private void receiveLoop() {  
191     //通过一个无限循环, 不断将接收到的信息放到packetBytes数组中  
192     while (true) {  
193         synchronized(this) {  
194             while (incomingBytes != null) {  
195                 try {  
196                     wait();  
197                 }  
198                 catch (InterruptedException e) {  
199                     }  
200             }  
201         }  
202         byte[] packetBytes;  
203         try {  
204             byte[] buffer = new byte[Packet.maxPacketLength];  
205             DatagramPacket dp = new DatagramPacket(buffer, buffer.length);  
206             socket.receive(dp);  
207             packetBytes = new byte[dp.getLength()];  
208             System.arraycopy(buffer, 0, packetBytes, 0, packetBytes.length);  
209         }  
210         catch (IOException e) {  
211             return;  
212         }  
213         //接收了信息后设置incomingBytes=packetBytes, 表示有信息接受但是未被处理  
214         synchronized(this) {  
215             incomingBytes = packetBytes;  
216         }  
217     }  
}
```

图 39 NetworkLink 启动的循环接受信息线程执行的 receiveLoop()函数代码解读


```

146 private synchronized void receiveInterrupt() {
147     Lib.assertTrue(incomingPacket == null);
148     //incomingBytes != null表示此时有消息接受了但是未被处理
149     //此时则启动receiveInterruptHandler处理函数去将信息处理
150     if (incomingBytes != null) {
151         if (Machine.autoGrader().canReceivePacket(privilege)) {
152             try {
153                 incomingPacket = new Packet(incomingBytes);
154                 privilege.stats.numPacketsReceived++;
155             }
156             catch (MalformedPacketException e) {
157             }
158         }
159         incomingBytes = null;
160         notify();
161         if (incomingPacket == null)
162             scheduleReceiveInterrupt();
163         else if (receiveInterruptHandler != null)
164             //该receiveInterruptHandler的处理函数在PostOffice中
165             receiveInterruptHandler.run();
166     }
167     //没有信息则不用处理，再次启动一个新的接受中断即可
168     else {
169         scheduleReceiveInterrupt();
170     }

```

图 40 NetworkLink 启动的接受信息的中断处理函数代码解读

```

69 //PostOffice创建时会启动一个线程，该线程会执行 postalDelivery()的函数
70 //postalDelivery()函数通过无限循环不断去接受信息
71 private void postalDelivery() {
72     while (true) {
73         //receiveInterrupt函数执行后，便可继续往下走
74         messageReceived.P();
75         Packet p = Machine.networkLink().receive();
76         MailMessage mail;
77         try {
78             mail = new MailMessage(p);
79         }
80         catch (MalformedPacketException e) {
81             continue;
82         }
83         if (Lib.test(dbgNet))
84             System.out.println("delivering mail to port " + mail.dstPort
85                               + ": " + mail);
86         //接受信息后存到对应端口号的队列中
87         queues[mail.dstPort].add(mail);
88     }
89 }
90
91 //接受信息后的中断处理函数
92 //当NetworkLink接受信息的中断触发后会调用
93 //将messageReceived信号量进行一次V操作
94 private void receiveInterrupt() {
95     messageReceived.V();
96 }

```

图 41 PostOffice 类的接受信息的中断处理函数代码解读

最后再看看 Nachos 的发送机制，首先 PostOffice 内有一个 send(MailMessage) 方法用于发送信息。该方法对调用 NetworkLink 类的 send 函数进行发送，后者的 send 方法启动一个发送信息的中断，在中断触发时执行发送信息的中断处理函数，进行信息的发送，发送后再调用 PostOffice 内的 sendInterrupt() 处理函数，该函数使得 PostOffice 的 send(MailMessage) 方法继续往下进行，意味着发送成功。相应的代码解读如下图所示。

```

98 //发送信息
99 public void send(MailMessage mail) {
100     if (Lib.test(dbgNet))
101         System.out.println("sending mail: " + mail);
102     sendLock.acquire();
103     Machine.networkLink().send(mail.packet); //调用networkLink的send发送
104     //sendInterrupt函数触发后便可继续往下执行
105     messageSent.P();
106     sendLock.release();
107 }
108
109 //当信息在网卡中发送成功，会调用这个发送的中断处理程序
110 //该程序对messageSent信号量进行V操作
111 private void sendInterrupt() {
112     messageSent.V();

```

图 42 PostOffice 的 send()函数以及中断处理函数代码解读

```

258 //启动发送信息的中断，在中断触发时将信息发送
259 public void send(Packet pkt) {
260     if (outgoingPacket == null)
261         scheduleSendInterrupt();
262
263     outgoingPacket = pkt;
264 }
265
266 //发送信息的中断触发时执行的函数
267 private void sendInterrupt() {
268     Lib.assertTrue(outgoingPacket != null);
269     //检测信息合法后将信息发送
270     if (Machine.autoGrader().canSendPacket(privilege) &&
271         Lib.random() <= reliability) {
272         privilege.doPrivileged(new Runnable() {
273             public void run() { sendPacket(); }
274         });
275     }
276     else {
277         outgoingPacket = null;
278     }
279     if (sendInterruptHandler != null)
280         //发送后调用发送的中断处理程序处理
281         //该程序在PostOffice中
282         sendInterruptHandler.run();
283 }

```

图 43 NetworkLink 中 send()函数以及发送信息触发的中断处理函数代码解读

4. 实验指南

这一章将主要对四个实验进行简单的介绍和思路引导。每个实验的具体细节要求可以从官网中查看。进入官网后，点击 Projects and Nachos 选项进入项目页面，页面的最下方就是四个实验的网址，内部有每个实验的具体细节要求。

1) 实验一：内核线程

I. 题目：

1. 实现 Kthread 的 join() 函数。join() 函数的作用是，B 线程调用了 A.join()，B 线程应该等待，直到 A 结束后再执行。
2. 提供一个不使用信号量工具来实现的条件变量。条件变量的作用是，当一个进程/线程执行过程中发现有一些前提条件没有完成，那么这个进程/线程可以使用条件变量挂起，直到其他进程/线程前提条件完成再将该进程/线程唤醒。
3. 实现 Alarm 中的 waitUntil(long) 方法。作用是使得一个内核线程等待一段时间后再继续执行。
4. 使用条件变量实现一个类似于生产者消费者功能的程序。
5. 实现优先级调度。
6. 利用同步工具实现一个过桥游戏的程序。

II. 思路：

1. 为每个 Kthread 增添一个等待队列，里面存放所有需要等待该线程执行结束

后才能继续执行的线程。然后当该线程执行结束后，将所有等待队列中的线程唤醒。

2. 为每个条件变量实现一个队列，使用条件变量的线程在该条件变量的队列上挂起，唤醒时从队列中移除。
3. 在 Alarm 中实现一个队列，所有 waitUntil()等待的线程都在队列里，然后每次时钟中断时检查队列，将到时间的线程唤醒。
4. 利用信号量或者锁等同步工具实现的一个进程同步问题的程序。更具体的说，是一个经典的生产者消费者问题。
5. 为每一个线程设定一个优先级的值，调度时找出优先级最高的线程进行调度。
该题还有一个注意点，便是优先级传递，例如：A 线程需要等待 B 线程执行结束后才能执行，并且 A 线程优先级比 B 线程要高，那么在 A 线程等待 B 线程执行的时候，B 线程的优先级应该提高到与 A 线程相同。在这里提供一个效率不高的思路，在每次调度时，先计算每个线程经过传递后的实际优先级，然后再从每个线程中找出实际优先级最大的线程进行调度。
6. 与第四题类似，是一个利用信号量或者锁等同步工具实现的一个进程同步问题。

2) 实验二：多道程序设计

I. 题目：

1. 实现 creat, open, read, write, close, 和 unlink 几个文件操作的系统调用。Creat 用于创建文件，open 用于打开文件，read 和 write 分别对应于文件读写，

close 是文件关闭，unlink 用于文件删除。

2. 实现页表及其加载、读写等方法，从而使 Nachos 支持多道程序。
3. 实现 exec, join, 和 exit 几个系统调用。exec 用于创建子用户程序，join 用于协调用户程序间的先后执行顺序，exit 用于退出用户程序。
4. 实现一个彩票调度。彩票调度规则如下：第一点是每个线程的实际优先级是所有等待该线程执行完毕才能执行的线程的优先级之和；第二点是采取抽签决定下一个执行的线程，并且线程被抽签抽到的概率等于其实际优先级比上所有线程实际优先级之和。

II.思路：

1. 借助 Nachos 文件系统提供的方法，可以很简单的实现这些系统调用。不过应该注意的是，在这里还需要实现并且维护一个线程的打开文件表，并且打开文件表数组的第一和第二项应该分别设定为控制台的读写。
2. 将页表 pageTable 变量修改成真正的页表，并且修改一些方法。第一个是修改加载用户程序的 loadSections()方法，将用户程序加载到内存的时候，先创建程序的页表，然后为每个页表项从物理帧中找出空闲帧将内容装入。第二个是修改读写内存的 readVirtualMemory()以及 writeVirtualMemory()方法，改成从页表中找到相应的页表项，再通过页表项转换为物理帧号，再进行读写。
3. 借助 Nachos 中已经实现的 Kthread 的 folk(), join()和 finish()方法，可以很简单的实现这些系统调用。不过应该注意的是，在系统调用的同时要维护对应的一些资源的改变。同时，我们还可以进行一些额外的拓展，实现一些自定

义的系统调用哟。

4. 借助前面优先级调度的基础，修改相应的方法可以很简单的实现彩票调度。

3) 实验三：虚拟内存

I.题目：

1. 实现 TLB 以及反向页表。反向页表指的是一个总体的页表，里面包含了所有进程所有虚拟页到物理帧的对应关系。
2. 实现懒加载和按需调页的机制。懒加载和按需调页其实就是在加载进程时，先只为其分配虚拟页而不加载物理页到内存，然后当真正需要使用该页时，再将其加载到内存。

II.思路：

其实 TLB、懒加载和按需调页三者是相互关联的，应该共同来实现。首先，根据实验的建议，我们可以使用哈希表来实现反向页表，并且在使用反向页表的情况下，可以不再使用原来每个进程特有的那个页表。然后，我们需要实现一个异常处理函数来处理 TLB 异常，整个流程如下：当 TLB 异常发生，我们需要找出导致错误的虚拟页，该虚拟页的信息在 TLB 中不存在，因此我们要找出该虚拟页对应的物理帧号，然后将其加载到 TLB 中，如果该虚拟页还没有装入内存，也就是说这时该虚拟页还没有一个对应的物理帧号，因此则需要先从内存中找出一个空闲帧将虚拟页的内容装入，如果内存已经满了，还需要先从内存中选择一个牺牲的物理帧换出到硬盘，然后空出一个新的空闲帧来将虚拟页的内容装入。另外

值得注意的是，将虚拟页的内容装入内存又分为两种方式，第一种是该虚拟页仍未加载（懒加载），这时应该将其加载然后装入内存，第二种是该虚拟页已经加载，不过其内容之前被换出到硬盘了，这时需要从硬盘的交换空间中将其换入回到内存中。具体流程图如下所示：

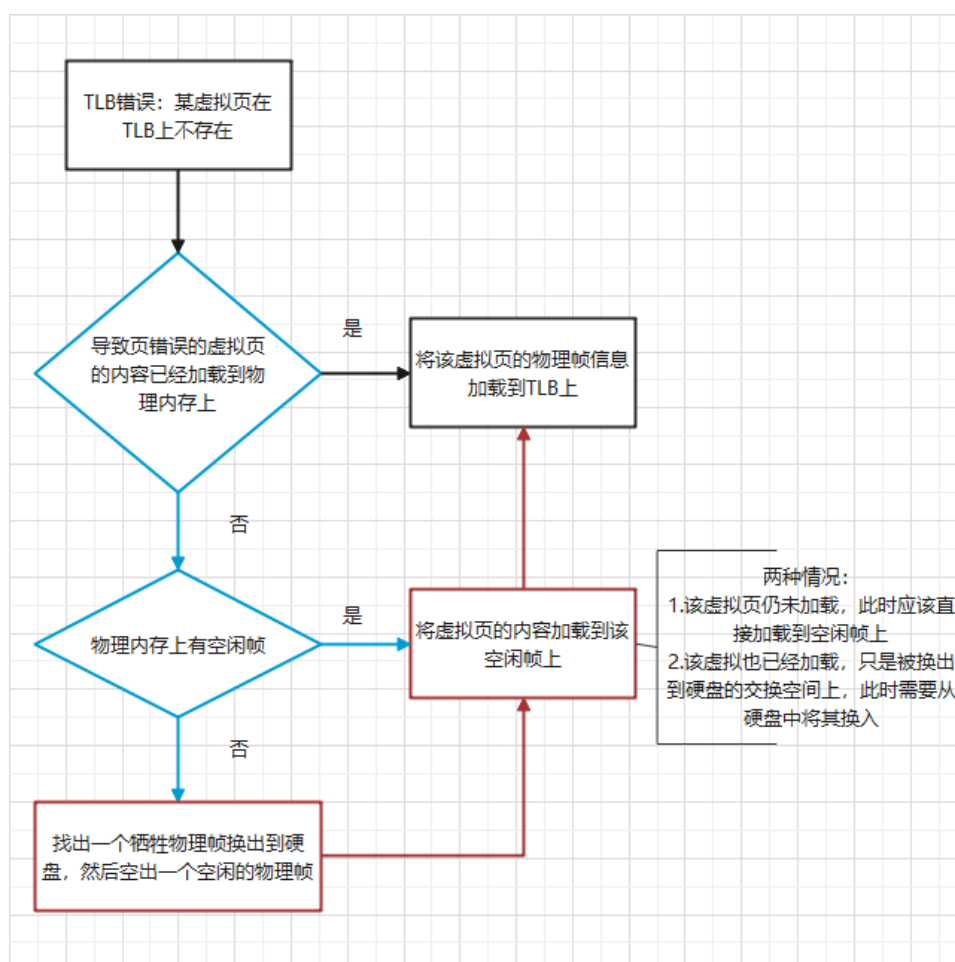


图 44 含 TLB 的页表按需调页流程图

4) 实验四：网络

I. 题目：

1. 实现 `connect()`, `accept()`, `close()`, `read()`, `write()` 这些网络相关系统调用。

2. 用两个 c 程序，借助前面的系统调用，实现服务器和主机的网络通信。

II.思路:

1. 通过 PostOffice 类的网络收发机制，我们可以很容易的实现网络之间的信息传递。不过有两个需要注意的地方，第一点是我们需要将建立的连接保存起来，在实验中要求我们将网络连接保存为文件，也就是说一个网络连接应该抽象为一个打开文件；第二点是我们需要保证收发的可靠性，因此还需要有超时重传的机制，如果有兴趣，还可以实现三次握手和三次挥手的连接关闭过程，模拟和体验 TCP 的机制。
2. 可以发挥自己的创意自己编写。