#### 操作系统课程设计报告-实验一

目录

[一、实验准备 3](#_Toc56502482)

[1.选择环境 3](#_Toc56502483)

[2.nachos导入eclipse 3](#_Toc56502484)

[3.研读nachos源代码 3](#_Toc56502485)

[二、设计实现 4](#_Toc56502486)

[Task1.1 Kthread.join() 4](#_Toc56502487)

[１．实验要求： 4](#_Toc56502488)

[２．要求分析： 4](#_Toc56502489)

[３．设计与实现： 5](#_Toc56502490)

[4．测试： 7](#_Toc56502491)

[Task1.2 condition2类 8](#_Toc56502492)

[１．实验要求： 8](#_Toc56502493)

[２．要求分析： 8](#_Toc56502494)

[３．设计与实现： 9](#_Toc56502495)

[4．测试： 10](#_Toc56502496)

[Task1.3 alarm类 12](#_Toc56502497)

[１．实验要求： 12](#_Toc56502498)

[２．要求分析： 12](#_Toc56502499)

[3.设计与实现： 13](#_Toc56502500)

[4．测试： 15](#_Toc56502501)

[Task1.4 communicator类 16](#_Toc56502502)

[１．实验要求： 16](#_Toc56502503)

[２．要求分析： 17](#_Toc56502504)

[３．设计与实现： 17](#_Toc56502505)

[4．测试： 19](#_Toc56502506)

[Task1.5 priority scheduler类 20](#_Toc56502507)

[１．实验要求： 20](#_Toc56502508)

[２．要求分析： 22](#_Toc56502509)

[3．设计与实现： 23](#_Toc56502510)

[4．测试： 27](#_Toc56502511)

[Task1.6 boat类 28](#_Toc56502512)

[１．实验要求： 28](#_Toc56502513)

[２．要求分析： 31](#_Toc56502514)

[３．设计与实现： 32](#_Toc56502515)

[4.测试： 36](#_Toc56502516)

[1.7 拓展.寻找产生时间延迟的原因： 37](#_Toc56502517)

[三、总结 38](#_Toc56502518)

# 一、实验准备

## 1.选择环境

虽然上学期的操作系统实验都是在Ubuntu虚拟机上完成的，但是在虚拟机上写代码有一定的困难，并且在大一的时有过Java多线程、文件和网络编程的经验，综上我选择nachos-java作为这次课程设计的环境。

## 2.nachos导入eclipse

导入

步骤一：准备一个nachos（java版）操作系统， eclipse编译工具。

步骤二：在eclipse上新建一个工程，工程名字可以任意，如TestNachos。

（路径如下：D:\os\TestNachos）

步骤三：将nachos系统解压，将获得一个nachos文件夹。

（可以解压在任意位置）

步骤四：将步骤三的nachos文件夹拷贝或剪切到工程TestNachos下的src文件夹下。

（路径如下：D:\os\TestNachos\src\nachos）

步骤五：在src/nachos.proj1下面将nachos.conf拷贝到工程TestNachos文件夹下。

## 3.研读nachos源代码

1. nachos虚拟机

Nachos模拟一个真实的CPU和硬件设备，包括中断和内存管理.Java中的包nachos.machine提供了这个模拟器。

Nachos模拟器用nachos.conf文件安装在不同的工程上,它详细描述了模拟器中包含哪些硬件设备和Nachos内核如何使用硬件,工程的说明书包括适当的配置文件，如果必要的话，可以改动工程的宣传册文件需要完成这个工程。

Nachos的启动进程类似于真实的虚拟机,nachos.machine.Machine.class的一个实例用来启动Nachos,计算机硬件（虚拟机类）首先初始化设备包括中断控制器，定时器，电梯控制器，MIPS进程，和文件系统。

虚拟机类对象把控制权交给特定的自动分级AutoGrader装置，相当于从硬盘上载入启动引导程序的代码, 是AutoGrader产生了Nachos内核，并开始运行系统,学生们不必关心这一步启动工作，真正有趣的部分从内核开始。

Nachos的内核仅仅是nachos.machine.Kernal的一个子类。比如说，线程项目用的是nachos.threads.ThreadedKernel。

Nachos虚拟机包括很多硬件设备。有一些在现代计算机中普遍存在（如：网络接口），但有些（比如说电梯控制）是Nachos特有的。Machine文件中的大多数类是硬件模拟的一部分，而在machine文件之外的类是Nachos操作系统的。

Nachos中断管理是通过nachos.machine.Interrupt 类通过维持一个事件队列来模拟。

1. nachos运行过程

从Machine.java的main函数分析源代码，进入AutoGrader.java , 分析出selfTest进行演示，进程创建进入Kernel.java ；

因为Kernel是一个抽象类，所以肯定有实现Kernel的实例,而在nachos.proj1的nachos.conf配置文件中包含Kernel.kernel = nachos.threads.ThreadedKernel，在threadedKernel.java中，找到selfTest()进入Kthread.java；

通过selfTest()找到PingTest，在PingTest里就包含了完整的运行proj1的信息 ；PingTest是继承runnable接口的类，runnable是一个Java中的接口.

# 二、设计实现

## Task1.1 Kthread.join()

### １．实验要求：

(5%, 5 lines)

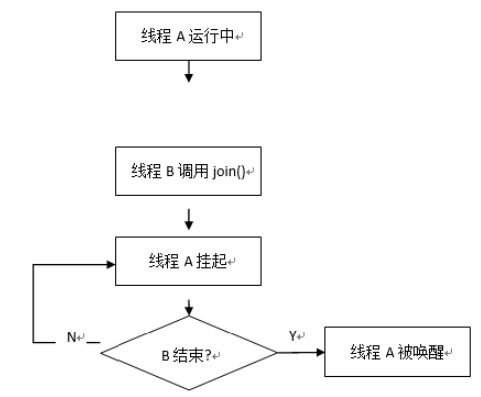
* Implement KThread.join().
* Note that another thread does not have to call join(), but if it is called, it must be called only once. The result of calling join() a second time on the same thread is undefined, even if the second caller is a different thread than the first caller.
* A thread must finish executing normally whether or not it is joined.
* 实现KThread.join().
* 请注意，另一个线程不必调用join（），但是如果调用了该线程，则只能调用一次。即使第二个调用者是与第一个调用者不同的线程，也无法在同一线程上第二次调用join（）的结果。
* 无论线程是否连接，线程必须正常完成执行。

### ２．要求分析：

从实验要求中我们可以得知：join函数的作用为等待调用此join函数的线程运行完毕，即当前线程处于睡眠状态，待另外一个线程运行完毕才会被唤醒继续运行。

调用join()函数的线程不能是当前线程；join()函数只能被调用一次，第二次调用的结果是没有定义的。

举个例子：当前线程 (唯一一个正在运行的线程) A调用另一个线程 (处于就绪状态)B的join()函数时 (A 和 B 在Nachos中均为KThread类型对象)，A被挂起，直到B运行结束后， join()函数返回，A才能继续运行。如果这个线程已经结束，马上返回。



每个线程都维护自己的资源，（在Linux和Windows操作系统中，线程维护的资源存在于TCB中）换言之，每个线程拥有自己的waitJoinQueue，里面存放了由于自己的join而被阻塞无法执行的线程。

由于我们对每个线程添加了waitJoinQueue切里面存有在等待的线程，故而我们需要修改finish函数，在线程完成运行即将终止时检查其waitJoinQueue，唤醒队列中在等待的线程

### ３．设计与实现：

在KThread创建一个ThreadQueue和int类型的变量用来分别表示等待队列和调用join函数的次数。在join方法中实现讲当前线程放到等待队列中（阻塞），将调用join方法的线程进行运行（独享）。在finish方法中取出ThreadQueue队列头线程进行运行，同时将ThreadQueue中的所有线程唤醒

①变量声明：

// 定义变量

**private** **static** ThreadQueue *waitJoinQueue* = **null**;

**private** **int** joinCount = 0;// 定义调用join的次数

构造方法里实现

//1.1start

// 抽象类由于java派生于抽象类的对象无法实例化，在运行时很可能出现空指针异常，该语句不可缺少

*waitJoinQueue* = ThreadedKernel.*scheduler*.newThreadQueue(**false**);

//1.1end

②方法实现:

join()方法:

**public** **void** join() {

Lib.*debug*(***dbgThread***, "Joining to thread: " + toString());

Lib.*assertTrue*(**this** != *currentThread*);

// 1.1 start

joinCount++;

**if** (joinCount > 1) {

System.***out***.println("提示:join函数只能被调用1次!");

**return**;

}

**boolean** intStatus = Machine.*interrupt*().disable();// 系统关中断(关中断是指在此中断处理完成前，不处理其它中断)

**if** (status != ***statusFinished***) {

// 让this线程(join方法调用之前正在运行的线程)成为waitForJoinQueue队列的头，

//表明只有先执行this线程，才会去执行队列里的线程

*waitJoinQueue*.acquire(**this**);//this线程获得资源

*waitJoinQueue*.waitForAccess(*currentThread*);//// 将当前线程加入到waitJoinQueue队列里，暂时阻塞

*currentThread*.*sleep*();// 当前进程睡眠等待被调用进程结束

}

Machine.*interrupt*().enable();// 系统开中断

// 1.1 end

}

finish()方法:

**public** **static** **void** finish() {

Lib.*debug*(***dbgThread***, "Finishing thread: " + *currentThread*.toString());

Machine.*interrupt*().disable();// 系统关中断

Machine.*autoGrader*().finishingCurrentThread();// 当前进程运行结束的标志

Lib.*assertTrue*(*toBeDestroyed* == **null**);

*toBeDestroyed* = *currentThread*;

*currentThread*.status = ***statusFinished***;// 当前进程的状态修改为运行结束

// 1.1 START

KThread waitThread = *currentThread*.*waitJoinQueue*.nextThread(); // 调用等待队列上的第一个进程;

**while** (waitThread != **null**) // while等待队列上有进程

{

waitThread.ready(); // 唤醒等待队列上所有被阻塞的进程

waitThread = *currentThread*.*waitJoinQueue*.nextThread(); // 调用等待队列上的下一个进程

}

// 1.1 END

*sleep*();

}

### 4．测试：

测试代码

// 1.1test start

**public** **static** **void** testJoin() {

**final** KThread b = **new** KThread(**new** PingTest(1));

System.***out***.println("thread b 启动");

b.fork();

System.***out***.println("调用join方法，当前线程阻塞，thread b 执行结束后,thread a 再执行(thread a 为主线程)");

b.join();//如果不使用join方法，a，b线程会交替执行

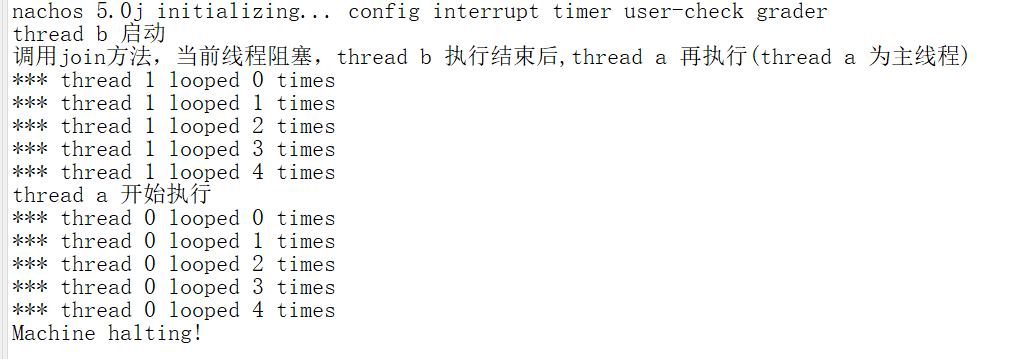
System.***out***.println("thread a 开始执行");

**new** PingTest(0).run();

}

// 1.1test end

测试结果



## Task1.2 condition2类

### １．实验要求：

(5%, 20 lines)

* Implement condition variables directly, by using interrupt enable and disable to provide atomicity.
* We have provided a sample implementation that uses semaphores; your job is to provide an equivalent implementation without directly using semaphores (you may of course still use locks, even though they indirectly use semaphores).
* Once you are done, you will have two alternative implementations that provide the exact same functionality.
* Your second implementation of condition variables must reside in class nachos.threads.Condition2.
* 通过使用中断启用和禁用来提供原子性，直接实现条件变量。
* 我们提供了一个使用信号量的示例实现； 您的工作是在不直接使用信号量的情况下提供等效的实现（当然，即使它们间接使用信号量，您也可能仍使用锁）。
* 完成后，将有两个提供完全相同功能的替代实现。
* 条件变量的第二个实现必须驻留在nachos.threads.Condition2类中。

### ２．要求分析：

条件变量使我们可以睡眠等待某种条件出现,是利用线程间共享的全局变量进行同步的一种机制,主要包括两个动作: 一个线程等待“条件变量的条件成立”而挂起；另一个线程使“条件成立”(给出条件成立信号)。我们要做的就是提供等效的实现与Condition.java。为了防止竞争,条件变量的使用总是和一个互斥锁结合在一起。

1. Lock conditionLock: 利用锁完成保证同时只有一个线程访问临界代码区

2. ThreadQueue waitQueue: 该条件变量等待的线程构成的队列

3. sleep(): 其他线程在该条件变量上的等待，即进入该条件变量的waitQueue陷入阻塞状态

  4. wake(): 唤醒该条件变量中等待的某个线程

5. wakeAll(): 即对wake()方法的循环调用，直到队列中没有线程为止

### ３．设计与实现：

在condition2中创建一个线程等待队列waitQueue，在sleep方法中暂时释放锁，但只有当重新获取锁后该方法才会返回。在wake方法中将队列中的第一个线程唤醒（但是不释放锁），wakeAll方法唤醒队列中的所有线程（同样也不释放锁），只有release方法才释放锁。

①变量声明：

**private** Lock conditionLock;

//1.2 start

**private** ThreadQueue waitQueue=**null**;

//1.2 end

在构造器中初始化：

**this**.conditionLock = conditionLock;

//1.2 start

waitQueue=ThreadedKernel.*scheduler*.newThreadQueue(**false**);

//1.2 end

②方法实现：（关键代码主要为sleep()方法，wake()方法和wakeAll()方法）

**public** **void** sleep() {

Lib.*assertTrue*(conditionLock.isHeldByCurrentThread());//检测当前线程是否获得锁

conditionLock.release();

//1.2 start

**boolean** preState = Machine.*interrupt*().disable();//关中断

waitQueue.waitForAccess(KThread.*currentThread*());//将当前线程加入到waitQueue中

KThread.*currentThread*().*sleep*();//让当前线程睡眠,返回就是有线程使用wake唤醒

Machine.*interrupt*().restore(preState);//恢复中断

//1.2 end

conditionLock.acquire();//重新获得锁才能执行

}

**public** **void** wake() {

Lib.*assertTrue*(conditionLock.isHeldByCurrentThread());

//1.2 start

**boolean** preState = Machine.*interrupt*().disable();//关中断

KThread thread = waitQueue.nextThread();

**if**(!(thread==**null**))

thread.ready();//只是将线程加入就绪队列但不释放锁

Machine.*interrupt*().restore(preState);//恢复中断

//1.2 end

}

**public** **void** wakeAll() {

Lib.*assertTrue*(conditionLock.isHeldByCurrentThread());

//1.2 start

**boolean** preState = Machine.*interrupt*().disable();//关中断

KThread thread = waitQueue.nextThread();

**while**(thread!=**null**){//将waitQueue中的所有线程均唤醒

thread.ready();

thread=waitQueue.nextThread();

}

Machine.*interrupt*().restore(preState);//恢复中断

//1.2 end

}

### 4．测试：

代码：

//1.2test start

**public** **static** **void** testCondition2(){

Lib.*debug*(***dbgThread***, "Enter KThread.selfTest");

Lock lock=**new** Lock();

Condition2 con=**new** Condition2(lock);

KThread thread\_A = **new** KThread(**new** Runnable(){

**public** **void** run(){

lock.acquire();

System.***out***.println("-----Now we begin to testCondition2()-----");

System.***out***.println("thread\_A will sleep");

con.sleep();

System.***out***.println("thread\_A is waked up");

// cdt.wake();

lock.release();

System.***out***.println("thread\_A execute successful!");

}

});

KThread thread\_B = **new** KThread(**new** Runnable(){

**public** **void** run(){

lock.acquire();

System.***out***.println("thread\_B will sleep");

con.sleep();

//KThread.currentThread.yield();

System.***out***.println("thread\_B is waked up");

// lock.release();

System.***out***.println("thread\_B execute successful!");

}

});

KThread thread\_MM=**new** KThread(**new** Runnable(){

**public** **void** run(){

lock.acquire();

System.***out***.println("Thread\_Wake:I will wake up all of the threads");

con.wakeAll();

lock.release();

//KThread.currentThread.yield();

System.***out***.println("thread\_Wake execute successful!");

// System.out.println("successful!");

}

});

thread\_A.fork();

thread\_B.fork();

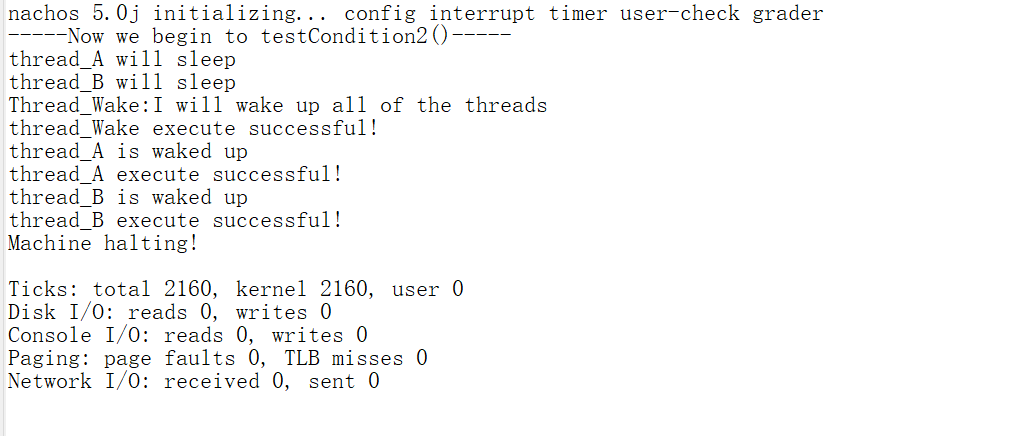
thread\_MM.fork();

// thread\_A.join();

}

//1.2test end

结果：



## Task1.3 alarm类

### １．实验要求：

(10%, 40 lines)

* Complete the implementation of the Alarm class, by implementing the waitUntil(long x) method.
* A thread calls waitUntil to suspend its own execution until time has advanced to at least now + x.
* This is useful for threads that operate in real-time, for example, for blinking the cursor once per second.
* There is no requirement that threads start running immediately after waking up; just put them on the ready queue *in the timer interrupt handler* after they have waited for at least the right amount of time.
* Do not fork any additional threads to implement waitUntil();
* you need only modify waitUntil() and the timer interrupt handler.
* waitUntil is not limited to one thread; any number of threads may call it and be suspended at any one time.
* 通过实现waitUntil（long x）方法，完成Alarm类的实现。
* 线程调用waitUntil来暂停其执行，直到时间至少提前到现在 + x。
* 这对于实时操作的线程很有用，例如，每秒使光标闪烁一次。
* 不需要在唤醒后立即开始运行线程。在它们等待了至少适当的时间后，只需将它们放在计时器中断处理程序中的就绪队列中即可。
* 不要派生任何其他线程来实现waitUntil（）;
* 您只需要修改waitUntil（）和计时器中断处理程序。
* waitUntil不限于一个线程；任何数量的线程都可以调用它，并且可以随时将其暂停。

### ２．要求分析：

Alarm 类使用硬件定时器提供抢占，并允许线程挂起到某个时间。我们要完成的就是一个线程在调用了waitUntil(x)函数之后，就代表它想至少等待x个滴答之后再运行,调用之后会暂时让它睡眠。

定时器中断处理程序被称为机器计时器定期（大约每 500时钟周期)，每500个滴答，时间中断方法自动执行，立马去检查等待队列上的线程有没有到唤醒时间（调用waitUntil的时间+(x)），如果到了，将其加入ready队列。

### 3.设计与实现：

在Alarm类中创建一个数据结构Waiter用来记录等待线程的信息，包括线程号和唤醒时间，用一个链表来储存每个线程的waiter。在waitUntil(**long** x)方法中将线程休眠，将线程的相关信息放到waiter中将waiter放到链表中储存。在timerInterrupt()方法中将到达休眠时间的线程唤醒并放到就绪队列中。系统Timer类硬件计时器大约每500个时钟滴答产生一次CPU计时器中断。

代码：

//1.3start

**private** LinkedList<Waiter> waitlist = **new** LinkedList();/\*!!!注意new\*/

/\*\*

\*存储等待线程的信息，包括线程号和唤醒时间

\*内部类,存放线程信息

\*/

**private** **class** Waiter{

**private** KThread thread;//等待线程

**private** **long** wakeTime;//唤醒时间

**public** Waiter(KThread thread,**long** wakeTime){

**this**.thread=thread;

**this**.wakeTime=wakeTime;

}

**public** KThread getThread() {

**return** thread;

}

**public** **void** setThread(KThread thread) {

**this**.thread = thread;

}

**public** **long** getWakeTime() {

**return** wakeTime;

}

**public** **void** setWakeTime(**long** time) {

**this**.wakeTime = wakeTime;

}

}

//1.3end

/\*好啦，时钟中断了！！

\* 快点去检查等待队列上的线程有没有到唤醒时间？

\* 到唤醒时间的线程 从 waitlist中移除 加入ready队列！！！

\* \*/

**public** **void** timerInterrupt() {//产生时钟中断 回调函数

//1.3start

**boolean** preState = Machine.*interrupt*().disable();//关中断

Waiter waiter;

//将等待队列上应该唤醒的队列加入到就绪队列

**for**(**int** i=0;i<waitlist.size();i++){

waiter = waitlist.getFirst();

**if**(waiter.wakeTime<=Machine.*timer*().getTime()){//如果达到唤醒时间，将其从waitlist中移除并唤醒该线程

System.***out***.println("唤醒线程："+waiter.thread.getName()+",时间为："+Machine.*timer*().getTime());

waitlist.removeFirst();//移除这个已到唤醒时间的线程并加入ready

waiter.thread.ready();//加入ready,线程进入就绪状态

}

}

Machine.*interrupt*().restore(preState);//恢复中断

//1.3end

KThread.*currentThread*().*yield*();

}

/\*!!某线程调用我了！！

\* 我要记住它的当前时间+x = 它的唤醒时间

\* 并把它加入到waitlist上!!

\* \*/

**public** **void** waitUntil(**long** x) {

// for now, cheat just to get something working (busy waiting is bad)

**long** wakeTime = Machine.*timer*().getTime() + x; /\*确定唤醒时间\*/

// while (wakeTime > Machine.timer().getTime())

// KThread.yield();

//1.3start

**boolean** preState = Machine.*interrupt*().disable();//关中断

Waiter waiter = **new** Waiter(KThread.*currentThread*(), wakeTime);

waitlist.add(waiter);//加入到等待列表waitlist

System.***out***.println(KThread.*currentThread*().getName()+"线程休眠，时间为："+Machine.*timer*().getTime()+",应在"+wakeTime+"时间醒来。");

KThread.*sleep*();//该线程休眠！！

Machine.*interrupt*().restore(preState);//恢复中断

//1.3end

}

### 4．测试：

代码：

//1.3test start

**public** **static** **void** testAlarm() {

KThread alarmThread\_1 = **new** KThread(**new** Runnable() {

**int** wait = 10;

**public** **void** run() {

System.***out***.println("-----Now we begin to testAlarm()-----");

System.***out***.println("alarmThread\_1进入睡眠,时间:" + Machine.*timer*().getTime() + "等待时间:" + wait);

ThreadedKernel.*alarm*.waitUntil(wait);

System.***out***.println("alarmThread\_1执行结束后的系统时间:" + Machine.*timer*().getTime());

}

}).setName("alarmThread\_1");

KThread alarmThread\_2 = **new** KThread(**new** Runnable() {

**int** wait = 1500

**public** **void** run() {

System.***out***.println("alarmThread\_2进入睡眠,时间:" + Machine.*timer*().getTime() + "等待时间:" + wait);

ThreadedKernel.*alarm*.waitUntil(wait);

System.***out***.println("alarmThread\_2执行结束后的系统时间:" + Machine.*timer*().getTime());

}

}).setName("alarmThread\_2");

alarmThread\_1.fork();

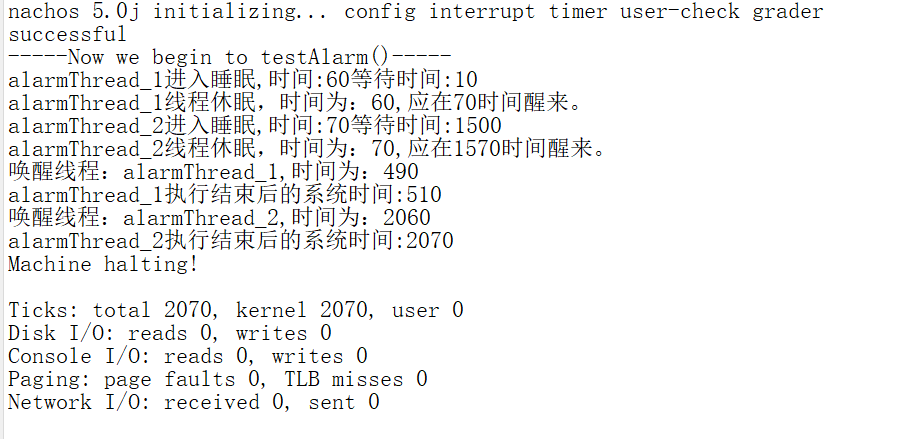
alarmThread\_2.fork();

System.***out***.println("successful");

}

//1.3 test end! 结果：

结果



## Task1.4 communicator类

### １．实验要求：

(20%, 40 lines)

* Implement synchronous send and receive of one word messages (also known as Ada-style rendezvous),

--using **condition variables** (don't use semaphores!).

--Implement the Communicator class with operations,

--void speak(int word) and int listen().

* speak() atomically waits until listen() is called on the same Communicator object, and then transfers the word over to listen(). Once the transfer is made, both can return.
* Similarly, listen() waits until speak() is called, at which point the transfer is made, and both can return (listen() returns the word).
* This means that neither thread may return from listen() or speak() until the word transfer has been made.
* Your solution should work even if there are multiple speakers and listeners for the same Communicator
* (note: this is equivalent to a zero-length bounded buffer; since the buffer has no room, the producer and consumer must interact directly, requiring that they wait for one another). Each communicator should only use exactly **one** lock. If you're using more than one lock, you're making things too complicated.
* 使用条件变量（不要使用信号量！）来实现一个单词消息的同步发送和接收（也称为Ada风格的集合点）。通过以下操作实现Communicator类：void speak(int word) 和int listen（）。
* speak（）原子地等待直到在同一Communicator对象上调用listen（），然后将单词转移到listen（）。转移完成后，双方都可以返回。
* 类似地，listen（）等待直到调用speak（），然后进行传输，并且两者都可以返回（listen（）返回单词）。
* 这意味着在完成字传输之前，任何线程都不能从listen（）或speak（）返回。即使同一Communicator有多个说话者和听者，您的解决方案也应能工作。
* （注意：这等效于零长度的有界缓冲区；由于缓冲区没有空间，生产者和使用者必须直接交互，要求他们等待另一个）。每个通信器应仅使用一个锁。如果您使用多个锁，那么会使事情变得过于复杂。

### ２．要求分析：

使用条件变量来实现一个字长信息的发送和接收同步。使用void speak(int word)和int listen函数来实现通讯（Communicator)类的通讯操作。

本题目相当于缓冲区为0的生产者消费者问题，既然缓冲区没用空间，那么需要生产者和消费者直接进行交互，要求他们相互等待。简单来讲，如果有speaker说话，有listener就相当于交流成功，否则阻塞speaker同时等待listener；如果有listener，有speaker就相当于交流成功，否则阻塞listener同时等待speaker。

### ３．设计与实现：

在Communicator类中添加变量Lock和speakerNum，listenerNum，words以及条件变量listener和speaker。在构造方法中将其初始化。在speak方法中先获得锁，将数据放到链表中，如果没有听的人，进入睡眠。如果有听的人将听的人唤醒。在listen方法中，先获得锁，如果有说的人，将其唤醒，进行数据传输，将链表的最后一位数据删除，否则进入睡眠。

//1.4 start

**private** Lock lock;//互斥锁

**private** **int** speakerNum;//说者数量

**private** **int** listenerNum;//听者数量

**private** LinkedList<Integer> words;//保存说者话语

Condition2 listener;//说者条件变量

Condition2 speaker;//听者条件变量

//1.4 end

/\*\*

\* Allocate a new communicator.

\*/

**public** Communicator() {

//1.4 start

lock=**new** Lock();

words=**new** LinkedList<Integer>();

speaker=**new** Condition2(lock);

listener=**new** Condition2(lock);

speakerNum=0;

listenerNum=0;

//1.4 end

}

**public** **void** speak(**int** word) {

//1.4 start

**boolean** preState = Machine.*interrupt*().disable();//关中断

lock.acquire();//拿到锁

words.add(word);//说者的word加入words链表

**if**(listenerNum == 0){//没有听者，说者睡眠，队列存储说者的话

System.***out***.println("暂时没有收听者，等待收听");

speakerNum++;//说者人数+1

speaker.sleep();

listenerNum--;

}

**else**{//有听者

System.***out***.println("有收听者，作为speaker我要直接唤醒它，且word传给它！");

speakerNum++;//说者人数+1

listener.wake();

listenerNum--;

}

lock.release();

Machine.*interrupt*().restore(preState);//开中断

**return**;

//1.4 end

}

**public** **int** listen() {

//1.4 start

//如果有人说话,则让说者说话，得到word，听者返回word；否则听者等待

**boolean** preState = Machine.*interrupt*().disable();//关中断

lock.acquire();//拿到锁

**if**(speakerNum == 0){

System.***out***.println("暂时没有说话者，等待说话");

listenerNum++;//listener++

listener.sleep();

speakerNum--;

}

**else**{

System.***out***.println("有说者，我要唤醒它！");

listenerNum++;//listener++

speaker.wake();//如果说话者不为0，则wake一个speaker说话

speakerNum--;

}

lock.release();//释放锁

Machine.*interrupt*().restore(preState);//开中断

**return** words.removeLast();

//1.4 end

}

### 4．测试：

测试代码：(特意添加了Thread本身的sleep方法测试对话是否是一对一的)

//1.4 test start

**public** **static** **void** testCommunicator() {

System.***out***.println("-----Now we begin to testCommunicator()-----");

Communicator c = **new** Communicator();

**new** KThread(**new** Runnable() {

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 5; ++i) {

System.***out***.println("speaker 大声说出 : " + i);

c.speak(i);

KThread.*yield*();

}

}

}).setName("Speaker").fork();

**for** (**int** i = 0; i < 5; ++i) {

**try** {

Thread.*sleep*(1000);//发现对话确实是一对一对话的

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("我是listener" + i + ",我来听speaker说话.");

**int** x = c.listen();

System.***out***.println("listener 听到了, word = " + x);

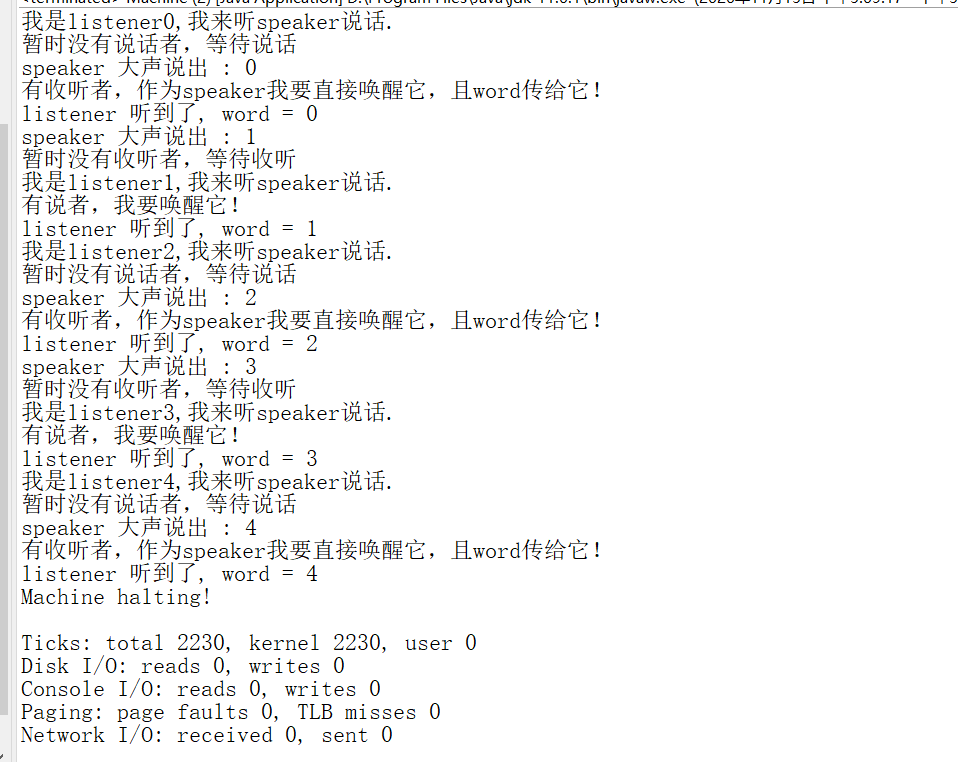
KThread.*yield*();

}

}

//1.4 test end!

测试结果：



## Task1.5 priority scheduler类

### １．实验要求：

* (35%, 125 lines)

Implement priority scheduling in Nachos by completing the PriorityScheduler class. Priority scheduling is a key building block in real-time systems. Note that in order to use your priority scheduler, you will need to change a line in nachos.conf that specifies the scheduler class to use. The ThreadedKernel.scheduler key is initially equal to nachos.threads.RoundRobinScheduler. You need to change this to nachos.threads.PriorityScheduler when you're ready to run Nachos with priority scheduling.

* Note that all scheduler classes extend the abstract class nachos.threads.Scheduler. You must implement the methods getPriority(), getEffectivePriority(), and setPriority(). You may optionally also implement increasePriority() and decreasePriority() (these are not required). In choosing which thread to dequeue, the scheduler should always choose a thread of the highest effective priority. If multiple threads with the same highest priority are waiting, the scheduler should choose the one that has been waiting in the queue the longest.
* An issue with priority scheduling is priority inversion. If a high priority thread needs to wait for a low priority thread (for instance, for a lock held by a low priority thread), and another high priority thread is on the ready list, then the high priority thread will never get the CPU because the low priority thread will not get any CPU time. A partial fix for this problem is to have the waiting thread donate its priority to the low priority thread while it is holding the lock.
* Implement the priority scheduler so that it donates priority, where possible. Be sure to implement Scheduler.getEffectivePriority(), which returns the priority of a thread after taking into account all the donations it is receiving.
* Note that while solving the priority donation problem, you will find a point where you can easily calculate the effective priority for a thread, but this calculation takes a long time. To receive full credit for the design aspect of this project, you need to speed this up by caching the effective priority and only recalculating a thread's effective priority when it is possible for it to change.
* It is important that you do not break the abstraction barriers while doing this part -- the Lock class does not need to be modified. Priority donation should be accomplished by creating a subclass of ThreadQueue that will accomplish priority donation when used with the existing Lock class, and still work correctly when used with the existing Semaphore and Condition classes. Priority should also be donated through thread joins.
* Priority Donation Implementation Details:

1) A thread's effective priority is calculated by taking the max of the donor's and the recipient's priority. If thread A with priority 4 donates to thread B with priority 2, then thread B's effective priority is now 4. Note that thread A's priority is also still 4. A thread that donates priority to another thread does not lose any of its own priority. For these reasons, the term "priority inheritance" is in many ways a more appropriate name than the term "priority donation".

2）Priority donation is transitive. If thread A donates to thread B and then thread B donates to thread C, thread B will be donating its new effective priority (which it received from thread A) to thread C.

* 通过完成PriorityScheduler类，在Nachos中实现优先级调度。优先级调度是实时系统中的关键构建块。请注意，为了使用优先级调度程序，您将需要在nachos.conf中更改一行以指定要使用的调度程序类。 ThreadedKernel.scheduler密钥最初等于nachos.threads.RoundRobinScheduler。准备使用优先级调度运行Nachos时，需要将其更改为nachos.threads.PriorityScheduler。
* 请注意，所有调度程序类都扩展了抽象类nachos.threads.Scheduler。您必须实现方法getPriority（），getEffectivePriority（）和setPriority（）。您也可以选择实现crementPriority（）和reducerPriority（）（这些不是必需的）。在选择要出队的线程时，调度程序应始终选择具有最高有效优先级的线程。如果多个具有相同最高优先级的线程正在等待，则调度程序应选择队列中等待时间最长的线程。
* 优先级调度的一个问题是优先级倒置。如果高优先级线程需要等待低优先级线程（例如，由低优先级线程持有的锁），并且另一个高优先级线程在就绪列表中，则高优先级线程将永远无法获得CPU，因为低优先级线程将不会获得任何CPU时间。**此问题的部分解决方法是让等待线程在持有锁时将其优先级捐赠给低优先级线程。**
* 实现优先级调度程序，以便在可能的情况下提供优先级。确保实现Scheduler.getEffectivePriority（），该函数在考虑线程收到的所有捐赠后返回线程的优先级。
* 请注意，在解决优先级捐赠问题时，您会发现可以轻松计算线程的有效优先级的一点，但是此计算需要很长时间。要获得此项目设计方面的全部功劳，您需要通过缓存有效优先级并仅在可能更改线程时才重新计算线程的有效优先级来加快速度。
* 重要的是，在执行此部分时，请不要打破抽象的障碍-无需修改Lock类。优先级捐赠应该通过创建ThreadQueue的子类来完成，该子类在与现有的Lock类一起使用时将完成优先级捐赠，而在与现有的Semaphore和Condition类一起使用时仍可以正常工作。优先级也应通过线程连接捐赠。
* **优先捐赠实施细节：**

**1）线程的有效优先级是通过计算供者和受者优先级的最大值来计算的。如果具有优先级4的线程A捐赠给具有优先级2的线程B，则线程B的有效优先级现在为4。请注意，线程A的优先级也仍然为4。向其他线程捐赠优先级的线程不会失去其自身的优先级。由于这些原因，术语“优先继承”在许多方面比术语“优先捐赠”更合适。**

**2）优先捐赠是可转让的。如果线程A捐献给线程B，然后线程B捐献给线程C，则线程B将把其新的有效优先级（从线程A接收到）捐献给线程C。**

### ２．要求分析：

通过完成PriorityScheduler类来实现优先级调度；

为了使用优先级调度，需要在 nachos.conf 文件中更改一行内容，需要将ThredededKernel.scheduler的值由nachos.threads.RoundRobinScheduler更改为nachos.threads.PriorityScheduler；

优先级调度应该选择一个有效优先级最高的线程出队列，如果多个线程有效优先级相同，则应该选择等待时间最久的；

另外，优先级调度中需要处理优先级倒置问题（可以通过优先级捐赠的方法解决）。

### 3．设计与实现：

将transferPriority设置为true，同时声明一个队列头（但不在队列中）lockHolder还有一个等待链表waitList，有效优先级和无效优先级，一个储存所有等待链表的Hashset。通过优先级寻找下一个线程pickNextThread：通过两个遍历器来获取所有等待线程的优先级，找到最高优先级（相同的优先级时间早的先出来）。获得有效的优先级数getEffectivePriority：只有成为过队列头的线程才能获取等待队列中比它优先级更高的优先级作为它的优先级。

1. 变量声明

PriorityQueue类

/\*

\* 此队列是否应将优先级从等待线程转移到拥有线程。

\*/

**public** **boolean** transferPriority = **true**;

//1.5 start

**protected** KThread lockHolder = **null**; //队列头

**protected** LinkedList<KThread> waitList = **new** LinkedList<KThread>();

//1.5 end

ThreadState类

//1.5 start

**protected** **int** effectivePriority = -2;//有效优先级初始化为-2

**protected** **final** **int** invalidPriority = -1;//无效优先级初始化为-1

//等待该线程的所有优先队列（每个优先队列里有等待线程）,包括等待锁，等待join方法的队列

**protected** HashSet<nachos.threads.PriorityScheduler.PriorityQueue> acquired = **new** HashSet<nachos.threads.PriorityScheduler.PriorityQueue>();

//1.5 end

②实现方法

PriorityQueue类

/\* nextThread():

\* 决定this队列中下一个最可能执行的线程【如果this是readyQueue，则选择是下一个执行的线程】，

\* 最可能执行的线程是队列中有效优先级最高的线程。

\* \*/

**public** KThread nextThread() {//选择下一个有效优先级最高的线程 - 线程

Lib.*assertTrue*(Machine.*interrupt*().disabled());

// implement me

//1.5 start

ThreadState x = pickNextThread();//下一个选择的线程

**if**(x == **null**)//如果为null,则返回null

**return** **null**;

**else**{

KThread thread = x.thread;

getThreadState(thread).acquire(**this**);//将得到的线程改为this线程队列的队列头

**return** thread;//将该线程返回

}

//1.5 end

// return null;

}

/\*

\* 选择下一个有效优先级最高的线程；

\* 实现：

\* 扫描this队列中的每一个线程，通过比较每个线程的有效优先级，来选出最大有效优先级对应的线程。

\* 根据题意如果有效优先级相同，则应该选择等待时间最长的线程。

\* （对于这一点，我没有设置每个线程的等待时间，因为我的线程队列的底层实现是LinkedList，

\* 所以相同有效优先级，只要优先选择该链表前面的线程即可解决这个问题。）

\* （在代码中是这样实现的，只有当后一个线程的有效优先级大于前一个线程的有效优先级，

\* 变量nextThread的值才会改变）。

\* \*/

**protected** ThreadState pickNextThread() {//选择下一个有效优先级最高的线程 - 优先级

// implement me

//1.5 start

java.util.Iterator i = waitList.iterator();

KThread nextthread;

**if**(i.hasNext()){

nextthread = (KThread)i.next();//取出下一个线程

KThread x = **null**;

**while**(i.hasNext()){//比较线程的有效优先级，选出最大的，如果优先级相同，则选择等待时间最长的

x = (KThread)i.next();

//System.out.println(x.getName());

**int** a = getThreadState(nextthread).getEffectivePriority();

**int** b = getThreadState(x).getEffectivePriority();

**if**(a<b){

nextthread = x;

}

}

**return** getThreadState(nextthread);

}**else**

//1.5 end

**return** **null**;

}

ThreadState类

/\* 方法解释：

\* 该方法通过比较this线程的acquired中所有线程的优先级，将最高优先级捐献给this线程。

\* 【acquired为一个HashSet容器，该容器中装有所有等待该线程的队列，这些队列包括

\* 锁队列（如果该线程持有几把锁，那么等待该锁的线程队列便会在里面），

\* waitForJoin队列（如果A线程join了该线程，则A线程会进入该线程waitForJoin队列）等，

\* 队列里放着等待的线程】

\* \*/

**public** **int** getEffectivePriority() {

// implement me

// 1.5 start

Lib.*assertTrue*(Machine.*interrupt*().disabled());

//ThreadState 有一个PriorityQueue类型的 waitQueue, PriorityQueue 中有个LinkedList的

//该优先级线程队列存在优先级捐赠吗？？

**if**(effectivePriority == invalidPriority && !acquired.isEmpty()){

effectivePriority = priority;//先将自己的优先级赋给有效优先级

**for**(Iterator i = acquired.iterator();i.hasNext();){//比较acquired中的所有等待队列中的所有线程的优先级

**for**(Iterator j = ((PriorityQueue)i.next()).waitList.iterator();j.hasNext();){

ThreadState ts = getThreadState((KThread)j.next());

**if**(ts.priority>effectivePriority){

effectivePriority = ts.priority;

}

}

}

**return** effectivePriority;

}**else**{

**if**(effectivePriority==-2) //表明该优先级线程队列不存在优先级捐赠

**return** priority;

**else**

**return** effectivePriority;//如果该线程没有执行，那么它之前算的有效优先级不必重新再算一遍

}

// 1.5 end

// return priority;

}

**public** **void** waitForAccess(PriorityQueue waitQueue) {

// implement me

//1.5 start

waitQueue.waitList.add(**this**.thread);//将this调用线程加入到等待队列

//1.5 end

}

/\*方法解释：设置waitQueue的队列头为this线程。【在代码实现中队列头为lockHolder】

\*

\* 使用范围: 只有两处使用到该方法

\* a、PriorityQueue类中acquire(KThread thread)方法调用了该方法；

\* b、PriorityQueue类中nextThread()方法调用了该方法。

\*

\* 代码实现解释

\* 先将this线程从waitQueue中移除（队列头不能在队列里），在代码中使用remove(this)方法，

\* 如果this线程本来就不在waitQueue队列中则返回false，否则返回true；

\* 然后将队列的队列头更换为this线程；判断waitQueue队列是否允许优先级捐赠，

\* 如果允许，则将this线程的有效优先级设置为无效优先级，并将该队列加入到acquired容器中。

\* 否则不作处理【即如果不允许优先级捐赠，则有效优先级仍然为初始值-2】。

\*

\* \*/

**public** **void** acquire(PriorityQueue waitQueue) {

// implement me

//1.5 start

waitQueue.waitList.remove(**this**.thread);//如果这个队列中存在该线程，删除

waitQueue.lockHolder = **this**.thread;//对于readyQueue来讲，lockHolder为执行线程；对于Lock类的waitQueue来讲，lockHolder为持锁者；对于waitForJoin队列来讲，lockHolder为执行join方法的线程。

**if**(waitQueue.transferPriority){//如果存在优先级翻转，则执行下面操作

**this**.effectivePriority = invalidPriority;

acquired.add(waitQueue);//将等待该线程的队列加入该线程的等待队列集合中

}

//1.5 end

}

### 4．测试：

代码

//1.5 test start

**public** **static** **void** testPriority() {

System.***out***.println("-----Now begin the testPriority()-----");

**boolean** status = Machine.*interrupt*().disable();// 关中断，setPriority()函数中要求关中断

**final** KThread a = **new** KThread(**new** PingTest(1)).setName("thread1");

**new** PriorityScheduler().setPriority(a, 3);

System.***out***.println("thread1的优先级为：" + **new** PriorityScheduler().getThreadState(a).priority);

KThread b = **new** KThread(**new** PingTest(2)).setName("thread2");

**new** PriorityScheduler().setPriority(b, 4);

System.***out***.println("thread2的优先级为：" + **new** PriorityScheduler().getThreadState(b).priority);

KThread c = **new** KThread(**new** Runnable() {

**public** **void** run() {

**for** (**int** i = 0; i < 5; i++) {

**if** (i == 2)

a.join();//并没有发生优先级翻转！！！！！！！！！！

System.***out***.println("\*\*\* thread 3 looped " + i + " times");

KThread.*currentThread*().*yield*();

}

}

}).setName("thread3");

**new** PriorityScheduler().setPriority(c, 5);

System.***out***.println("thread3的优先级为：" + **new** PriorityScheduler().getThreadState(c).priority);

a.fork();

b.fork();

c.fork();

Machine.*interrupt*().restore(status);

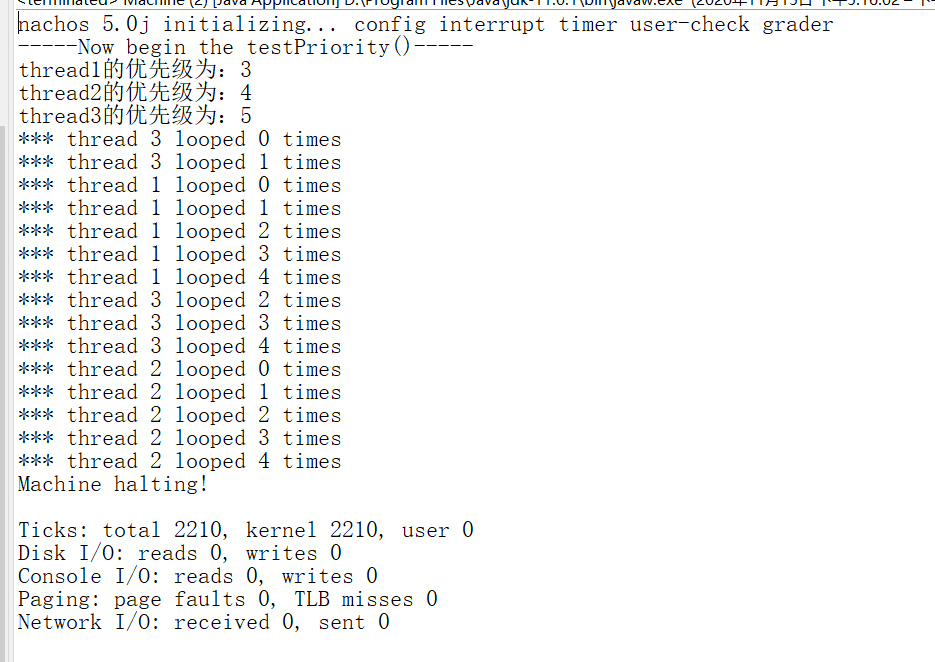
}

//1.5test end!

结果

使用join（）方法来对编写的程序进行测试。

首先给三个线程分别设置了优先级为3，4，5；如果不执行join()方法，必定会按照优先级高低顺序执行（线程3先执行，然后执行线程2，最后执行线程1）；但在测试程序中当优先级为5的线程（线程3）执行到第3次时，调用了a.join()方法；如果没有发生优先级捐赠的话，线程2应该先执行，因为它的优先级是4，但由于发生了优先级捐赠，线程3将优先级捐赠给了线程1，所以线程1的有效优先级现在是5，故结果中出现了线程1先执行，然后线程3继续执行，最后线程2才执行。



## Task1.6 boat类

### １．实验要求：

(25%, 150 lines)

* Now that you have all of these synchronization devices, use them to solve this problem. You will find condition variables to be the most useful synchronization method for this problem.
* A number of Hawaiian adults and children are trying to get from Oahu to Molokai. Unfortunately, they have only one boat which can carry maximally two children or one adult (but not one child and one adult). The boat can be rowed back to Oahu, but it requires a pilot to do so.
* Arrange a solution to transfer everyone from Oahu to Molokai. You may assume that there are at least two children.
* The method Boat.begin() should fork off a thread for each child or adult. We will refer to the thread that called Boat.begin() as the parent thread.
* Your mechanism cannot rely on knowing how many children or adults are present beforehand, although you are free to attempt to determine this among the threads (i.e. you can't pass the parameters adults and children in the method begin() to your threads, but you are free to have each thread increment shared variables to try and determine this value, if you wish).
* To show that the trip is properly synchronized, make calls to the appropriate BoatGrader methods every time someone crosses the channel.
* When a child pilots the boat from Oahu to Molokai, call ChildRowToMolokai. When a child rides as a passenger from Oahu to Molokai, call ChildRideToMolokai.
* Make sure that when a boat with two people on it crosses, the pilot calls the ...RowTo... method before the passenger calls the ...RideTo... method.
* Your solution must have no busy waiting, and it must eventually end. The simulation ends when the parent thread finishes running. Note that it is not necessary to terminate all the forked threads -- you can leave them blocked waiting for a condition variable. While you cannot pass the number of threads created to the threads representing adults and children, you can and probably will need to use this number in begin() in order to determine when all the adults and children are across and you can return.
* The idea behind this task is to use independent threads to solve a problem.
* You are to program the logic that a child or an adult would follow if that person were in this situation.
* For example, it is reasonable to allow a person to see how many children or adults are on the same island they are on.
* A person could see whether the boat is at their island.
* A person can know which island they are on.
* All of this information may be stored with each individual thread or in shared variables.
* So a counter that holds the number of children on Oahu would be allowed, so long as only threads that represent people on Oahu could access it.
* What is not allowed is a thread which executes a "top-down" strategy for the simulation. For example, you may not create threads for children and adults, then have a controller thread simply send commands to them through communicators. The threads must act as if they were individuals. This also means you cannot serialize your code by disabling interrupts inside Boat.java explicitly, for any reason. Every person must be thinking, and acting, independently at all times.
* Information which is not possible in the real world is also not allowed. For example, a child on Molokai cannot magically see all of the people on Oahu. That child may remember the number of people that he or she has seen leaving, but the child may not view people on Oahu as if it were there. (Assume that the people do not have any technology other than a boat!)
* You will reach a point in your simulation where the adult and child threads believe that everyone is across on Molokai. At this point, you are allowed to do one-way communication from the adult/child threads to begin() (the parent thread) in order to inform it that the simulation may be over. It may be possible, however, that your adult and child threads are incorrect. Your simulation must handle this case without requiring explict or implict communication from begin() (the parent thread) to the adult/child threads.
* 现在，您已经拥有所有这些同步设备，请使用它们来解决此问题。您会发现条件变量是此问题最有用的同步方法。
* 许多夏威夷成人和儿童正试图从瓦胡岛Oahu到莫洛凯岛Molokai。不幸的是，他们只有一艘船，最多可搭载两名儿童或一名成人（但不能携带一名儿童和一名成人）。该船可以划回瓦胡岛，但需要领航员这样做。
* 安排解决方案，将所有人从瓦胡岛转移到莫洛凯岛。您可以假设至少有两个孩子。
* **方法Boat.begin（）应该为每个孩子或成人派生一个线程。我们将称为Boat.begin（）的线程称为父线程。**尽管您可以随意尝试在线程之间确定这一点，但是您的机制不能依赖于事先知道有多少个孩子或成人（即，您不能将begin（）方法中的adult和children参数传递给线程，但是您可以随意让每个线程递增共享变量来尝试确定该值）。
* 为了显示行程已正确同步，**每当有人越过通道时，都要调用相应的BoatGrader中的方法**。
* 当一个孩子驾驶从瓦胡岛到莫洛凯岛的船时，请致电ChildRowToMolokai。当一个孩子乘车从瓦胡岛到莫洛凯岛时，请致电ChildRideToMolokai。
* 确保当载有两个人的小船越过时，飞行员在乘客调用... RideTo ...方法之前先调用... RowTo ...方法。
* 您的解决方案必须没有繁忙的等待时间，并且最终必须结束。当父线程完成运行时，模拟结束。请注意，不必终止所有分支的线程-您可以将它们保留为等待条件变量。虽然您无法将创建的线程数传递给表示成人和儿童的线程，但您可能并且可能需要在begin（）中使用此数字，以确定何时所有成人和儿童都可以通过并可以返回。
* 该任务背后的想法是使用独立线程来解决问题。
* 您要编写逻辑，以使儿童或成人在这种情况下会遵循的逻辑。
* 例如，允许一个人看到他们所在的同一个岛上有多少个孩子或成人是合理的。
* 一个人可以看到船是否在他们的岛上。
* 一个人可以知道他们在哪个岛上。
* 所有这些信息都可以与每个单独的线程或共享变量一起存储。
* 因此，只要拥有代表瓦胡岛人的线程的访问权限，就可以允许一个计数器来保存瓦胡岛上的孩子人数。
* 不允许执行模拟的“自上而下”策略的线程。例如，您可能没有为儿童和成人创建线程，然后让控制器线程仅通过通信器向他们发送命令。线程必须像个人一样操作。这也意味着您不能出于任何原因通过显式禁用Boat.java中的中断来序列化代码。每个人都必须始终独立思考和行动。
* 也不允许在现实世界中无法获得的信息。例如，莫洛凯岛上的一个孩子无法神奇地看到瓦胡岛上的所有人。该孩子可能记得他或她离开的人数，但该孩子可能不会像在瓦胡岛上那样看待人们。 （假设人们除了小船以外没有任何其他技术！）
* 您将在模拟中达到成人和儿童线程相信每个人都参与Molokai的地步。此时，您可以从成人/儿童线程到begin（）（父线程）进行单向通信，以告知模拟可能已结束。但是，您的成人和儿童线程可能不正确。您的仿真必须处理这种情况，而无需从begin（）（父线程）到成人/子线程进行显式或隐式通信。

### ２．要求分析：

利用前面所学同步方法，以及条件变量解决过河问题。一群夏威夷人想要从瓦胡岛（Oahu）到摩洛凯岛(Molokai)，只有一艘船，该船可载最多2个小孩或一个成人，且必须有一个开船的人(默认每个人都会开船）。设计一个解决方案,使得每个人都能从 Oahu 到Molokai。假设至少有两个小孩，每一个人都会划船。

要为每个大人和孩子创建一个线程，这个线程就相当于一个人，他们能自己判断（思考），什么时候可以过河，而不是通过调度程序的调度。在解决问题的过程中不能出现忙等待，而且问题最终一定要解决。不符合逻辑的事情不能发生，比如在这个岛的人能知道对面岛的情况。

需要记录的信息有：

1. 0 岛上大人/小孩的人数和M岛上大人/小孩的人数

2. 船的位置(在0岛还是M岛) 和状态(空/半满/全满) (半满指只有一个

小孩，全满指有两个小孩或一个大人)

3. 是否最后一次运输

4. 孩子、大人信息的条件变量，并且需要一个锁

运送过程大体如下：

1. 所以我们可以尽可能多的先从 0岛把孩子运到M岛(M岛到0岛需要一个child 驾驶员，要　　　避免出现当大人到M岛上而没有孩子在M岛上出现这种情况)
2. 然后开始运输大人，并且让孩子把船开回0岛
3. 重复1，2

4. 最后将两个孩子运到M岛

对于大人，若满足以下条件则独自乘船过河(每个大人过且仅过一次河，线程即结束)，否则(在0 岛) 等待：

1)0岛上只有一个小孩或没有小孩

2)船在0 岛

3)船为空

对于小孩, 满足以下情况则乘船过河：

1)初始条件，小孩全在ｏ岛，船在ｏ岛，且M岛没有小孩

2)某小孩在ｏ岛，船在ｏ岛 船为空，ｏ岛上的小孩数大于等于2，则该小孩上船等另外一个小孩上船后,两人一起划船过河到M岛

3)某小孩在ｏ岛，船在ｏ岛，船为空，ｏ岛上没有大人：该小孩上船过河

4)当所有的大人运完了之后，ｏ岛出现了两个孩子，这个时候这两个孩子划船过河。

### ３．设计与实现：

在Boat中声明相关变量，在begin方法中进行初始化，同时开启每个大人和小孩的线程。之后进行运行判断。注意先尽可能的将小孩运往O岛。只有当M岛的成人和孩子数同时为零时，该线程结束。

**变量声明：**

//1.6 start

**static** **int** *num\_children\_O*; //在Oahu岛上孩子的数量

**static** **int** *num\_alduts\_O*;//在Oahu岛上大人的数量

**static** **int** *num\_children\_M*;//在Molokai岛上孩子的数量

**static** **int** *num\_alduts\_M*;//在Molokai岛上大人的数量

**static** Lock *lock*;

**static** Condition *children\_condition\_o*;//孩子在Oahu岛上的条件变量

**static** Condition *children\_condition\_m*;//孩子在Molokai岛上的条件变量

**static** Condition *alduts\_condition\_o*; //大人在Oahu岛上的条件变量

**static** **boolean** *boatInO*;//船在O岛吗？

**static** **boolean** *is\_pilot*;//是舵手吗？

**static** **boolean** *is\_adult\_go*;//大人要过河（O->M）吗

**static** **boolean** *gameover*;//游戏结束了吗？

//1.6 end

**方法实现：**

**public** **static** **void** begin( **int** adults, **int** children, BoatGrader b )

{

// Store the externally generated autograder in a class

// variable to be accessible by children.

*bg* = b;

Lib.*assertTrue*(children>=2);//小孩>=2个

// Instantiate global variables here

// Create threads here. See section 3.4 of the Nachos for Java

// Walkthrough linked from the projects page.

//1.6 start

*num\_children\_O*=children;

*num\_alduts\_O* = adults;

*num\_alduts\_M* = 0;

*num\_children\_M* = 0;

*lock* = **new** Lock();

*children\_condition\_o* = **new** Condition(*lock*);

*children\_condition\_m* = **new** Condition(*lock*);

*alduts\_condition\_o* = **new** Condition(*lock*);

*boatInO* = **true**;

*is\_pilot* = **true**;

*is\_adult\_go* = **false**;

*gameover* = **false**;

//为每位大人和小孩创建线程，并启动

**for**(**int** i = 0;i<adults;i++){//每个大人为一个线程

**new** KThread(**new** Runnable(){

**public** **void** run(){

*AdultItinerary*();

}

}).fork();;

}

**for**(**int** i = 0;i<children;i++){//每个小孩为一个线程

**new** KThread(**new** Runnable(){

**public** **void** run(){

*ChildItinerary*();

}

}).fork();;

}

//1.6 end

}

**static** **void** AdultItinerary()

{

*bg*.initializeAdult(); //Required for autograder interface. Must be the first thing called.

//DO NOT PUT ANYTHING ABOVE THIS LINE.

/\* This is where you should put your solutions. Make calls

to the BoatGrader to show that it is synchronized. For

example:

bg.AdultRowToMolokai();

indicates that an adult has rowed the boat across to Molokai

\*/

//1.6 start

*lock*.acquire();//申请锁

//如果大人不满足过河条件或者船不在O岛，大人在O岛不走睡眠

**if**(!(*is\_adult\_go*&&*boatInO*)){

*alduts\_condition\_o*.sleep();

}

//否则大人划至M岛

*bg*.AdultRowToMolokai();

*num\_alduts\_M*++;//M岛的大人数量+1

*num\_alduts\_O*--;//O岛的大人数量—1

*boatInO* = **false**;//船改至M岛

*children\_condition\_m*.wake();//唤醒M岛的孩子线程

*is\_adult\_go* = **false**;//下一次船再到O岛时，必定是小孩走

*lock*.release();//释放锁

//1.6 end

}

**static** **void** ChildItinerary()

{

*bg*.initializeChild(); //Required for autograder interface. Must be the first thing called.

//DO NOT PUT ANYTHING ABOVE THIS LINE.

//1.6 start

*lock*.acquire();//申请锁

**while**(!*gameover*){

**if**(*boatInO*){//如果船在O岛

**if**(*is\_adult\_go*){//如果大人满足过河条件，能走，则将O岛的大人线程唤醒，O岛的孩子线程睡眠

*alduts\_condition\_o*.wake();

*children\_condition\_o*.sleep();

}

**if**(*is\_pilot*){//如果是第一个小孩，则设为舵手

*bg*.ChildRowToMolokai();

*num\_children\_O*--;//O岛小孩数量-1

*num\_children\_M*++;//M岛小孩数+1

*is\_pilot* = **false**;//将舵手设为false

*children\_condition\_o*.wake();//唤醒O岛的其他小孩线程

*children\_condition\_m*.sleep();//让自己睡眠在M岛

}**else**{//如果是第二个小孩，则设为游客

*bg*.ChildRideToMolokai();

*boatInO* = **false**;//将船改为在M岛

*num\_children\_O*--;//O岛的小孩数量-1

*num\_children\_M*++;//M岛的小孩数量+1

*is\_pilot*=**true**;//将舵手设为true

**if**(*num\_alduts\_O*==0&&*num\_children\_O*==0){//如果O岛的孩子和大人数量均为0，则游戏结束

*gameover* = **true**;

}

**if**(*gameover*){//如果游戏结束，则打印成功过河

System.***out***.println("成功过河！！！");

*children\_condition\_o*.sleep();

}

**if**(*num\_alduts\_O*!=0&&*num\_children\_O*==0){//如果O岛的大人还有，但小孩线程为0，则大人可走

*is\_adult\_go* = **true**;

}

*children\_condition\_m*.wake();//将M岛的其他孩子线程唤醒

*children\_condition\_m*.sleep();//将自己睡眠在M岛

}

}

**else**//如果船在M岛

{

*bg*.ChildRowToOahu();

*boatInO* = **true**;//设置船在O岛

*num\_children\_O* ++;//O岛孩子数量+1

*num\_children\_M* --;//M岛孩子线程数量-1

**if**(*is\_adult\_go*){//如果大人可以走，则将O岛的大人线程唤醒

*alduts\_condition\_o*.wake();

}**else**{//否则，唤醒O岛的孩子线程

*children\_condition\_o*.wake();

}

*children\_condition\_o*.sleep();//让自己睡眠在O岛

}

}

*lock*.release();//释放锁

//1.6 end

}

### 4.测试：

代码：

//1.6test start

**public** **static** **void** testBoat() {

System.***out***.println("-----Now we begin to testBoat()-----");

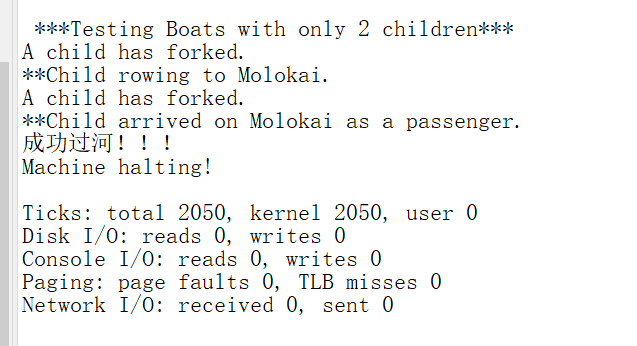
Boat.*selfTest*();

}

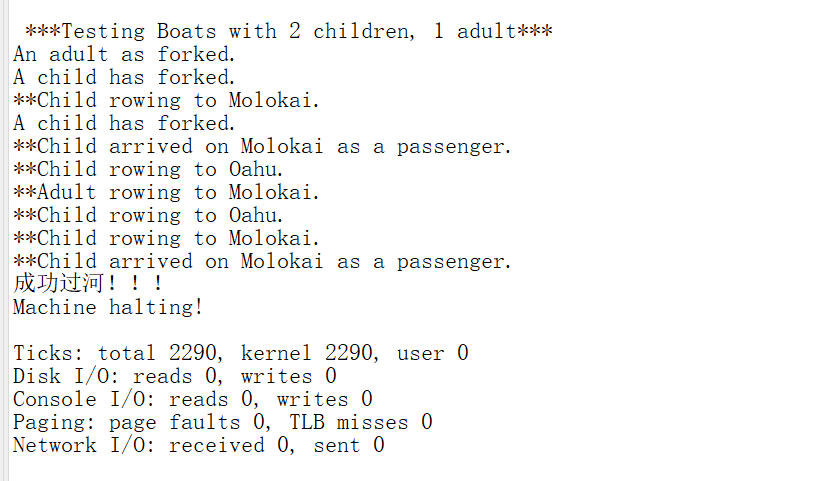
//1.6test end

结果（分别测试了实验给出的三个例子）：

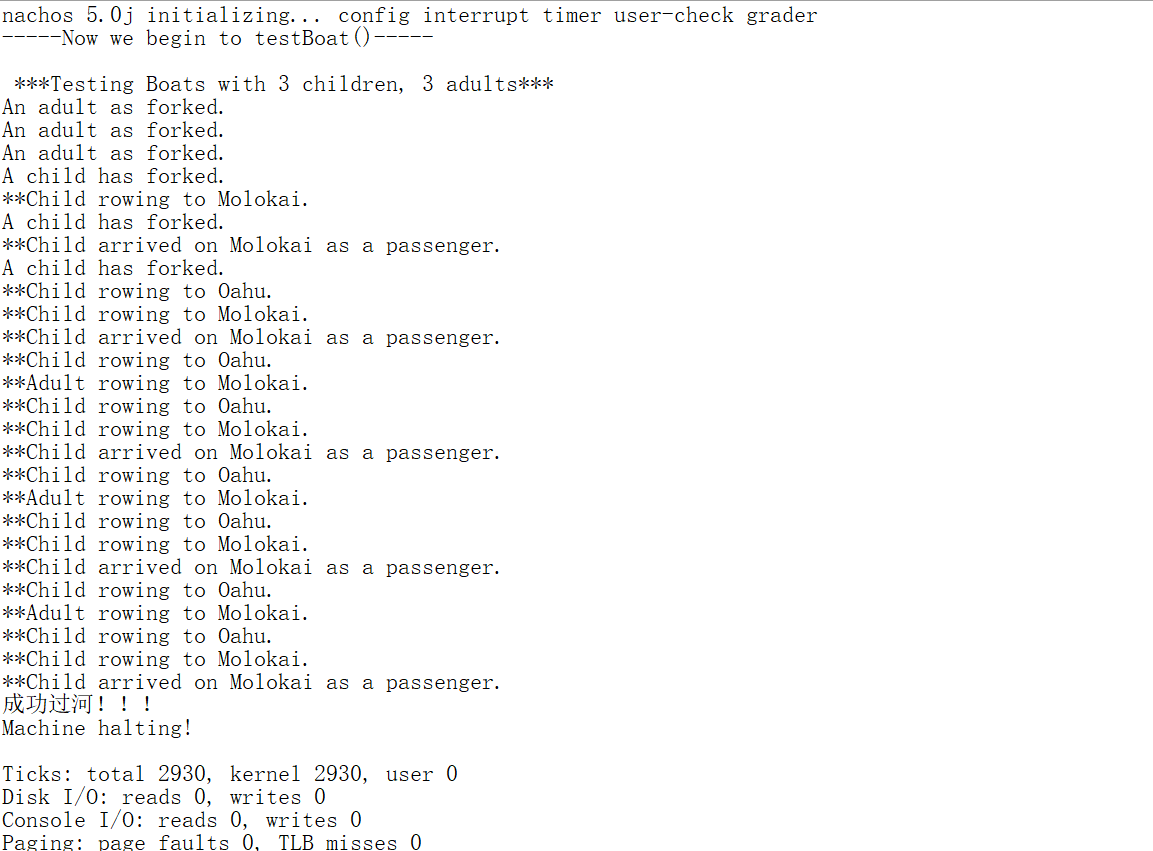
0个成人和2个小孩：



一个成人两个小孩：



三个成人和三个孩子：



## 1.7 拓展.寻找产生时间延迟的原因：

我们发现每次运行代码，就算修改等待时间，线程的唤醒都是需要时间延迟。当演示时老师提出这个问题，我们没有很好的回答，回来之后有仔细研读了nachos的代码。发现Alarm是通过timerInterrupt方法判断那个线程到达了睡眠时间，而Alarm构造方法创建一个硬件计时器timer，在计时器里开启一个线程独立运行timerInterrupt方法。于是进入硬件计时器timer进行观察。

硬件计时器大约每500个时钟滴答产生一次CPU计时器中断。 这意味着它可用于实现时间分片，或用于使线程在特定时间段内进入睡眠状态。我看到timer中的scheduleInterrupt()方法在Timer的构造函数中使用，并且代码如下：

**private** **void** scheduleInterrupt() {

**int** delay = Stats.***TimerTicks***;

delay += Lib.*random*(delay/10) - (delay/20);

privilege.interrupt.schedule(delay, "timer", timerInterrupt);

}

Delay时间延迟，于是进入Stats类中：

/\*\*

\* 计时器中断之间模拟的平均时间量。\*/

**public** **static** **final** **int** ***TimerTicks*** = 500;

这样就找到了产生时间延迟到原因就是因为硬件计时器每隔500ms产生一次滴答（脉冲），这样即使线程到达醒来的时间，也会因为没有脉冲仍然在睡眠。

修改TimerTicks为极小的一个值，观察结果：

代码：

**public** **static** **final** **int** ***TimerTicks*** = 5;

**private** **void** scheduleInterrupt() {

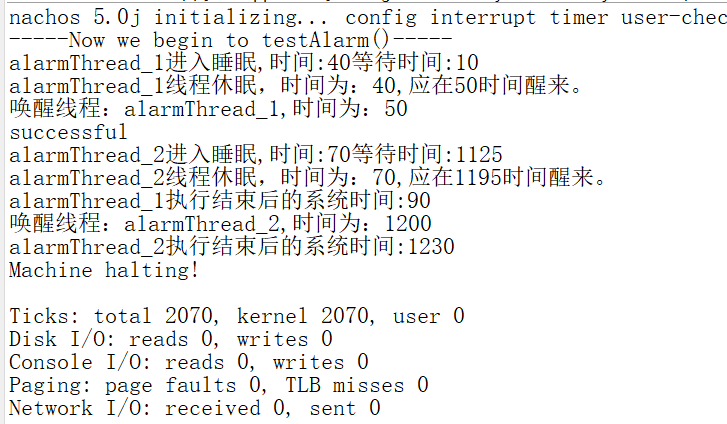
**int** delay = Stats.***TimerTicks***;

delay += Lib.*random*(delay/1) - (delay/2);

privilege.interrupt.schedule(delay, "timer", timerInterrupt);

}

结果：果然线程一在指定的时间被唤醒，找到了原因。



# 三、总结

刚开始，我们显示讨论了使用什么环境，发现网上linux教程比较多，但是如果做linux的话需要在虚拟机里做，如果是一个小实验还可以应对，但是如果是一个大型课程设计就显得很麻烦，于是我们还是选择了java。

其实，所有的困难更多来源内心的恐惧，通过自己一步步跟随一些网上博客以及看老师发的PPT，我们发现其实没有想象中那么困难，只要认真去读、去看、去研究，随着时间的推移，我们还是可以看得懂的。

在本Task1，主要是在nachos.threads文件夹里做文章，所以重点也是其内包含的java文件和java对于线程的控制Thread类。 于是我先去复习了java的多线程有关的基础知识，从Thread类，Runnable借口，静态代理，sleep方法到同步锁……一系列基础知识，通过掌握java本身过的线程知识来帮助掌握threads包。

同时，我们又复习了上学期的许多操作系统知识，对操作系统有了更深刻的理解。比如：我们又重新认识了线程与进程，关于进程之间的同步与互斥，通过条件变量，我们又有了新的认识，我们再次复习了一些简单的进程调度算法（如：RR调度、优先级调度等），需要在nachos配置文件里修改默认的调度方式。同时，对于一些零散的概念，我们又加深了印象，有了新的感悟，这一切都来源于我们亲身去设计、去做操作系统课设，才对之前觉得比较抽象的概念感觉有了新体会。

总的来说，受益匪浅，未来可期，继续加油！