山东 大学

操作系统实验报告

———Nachos For Java

专业：计算机科学与技术

年级：2014级3班

姓名：吴成敏

学号：201400130031

目录

Ps:由于第二个实验的测试程序及结果比较特殊，是前三个一起测试的，执行的是 coff 文件，没有测试代码，所以没有按照 Project 1 的格式写。关于 Project 2 的实验总结写在了最后的总结中。

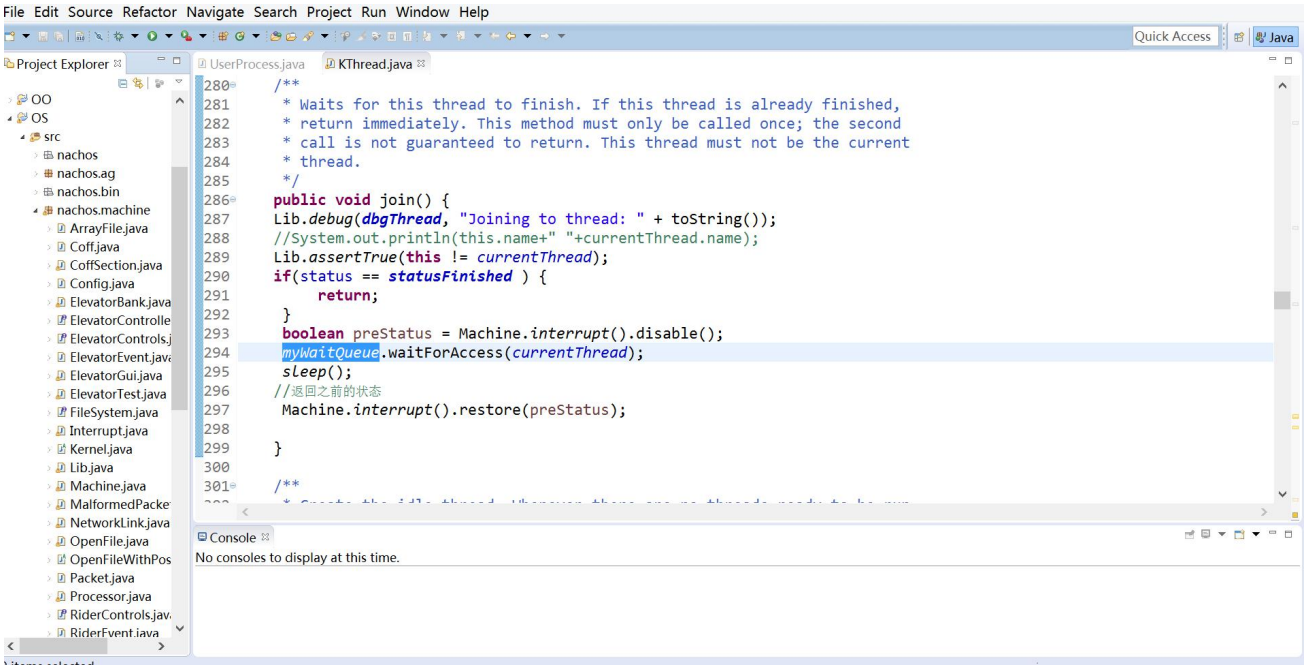
一、前期准备

1. 环境搭建

下载安装 eclipse

2. Eclipse 中导入 nachos

将指导老师所传的工程包解压，并导入 eclipse 中，基本的环境就搭建完成。



3. 前期关于 nachos 的准备：

通过老师的介绍以及自己搜查资料，对 nachos 系统有了一定的了解：nachos 是建立在一个软件（eclipse）模拟的虚拟机上，模拟了 MIPSR2/3000 的指令集、主存、中断系统、网络以及磁盘系统等操作系统所必须的硬件系统。在所给伯克利官网下载的 nachos 系统并不是一个完善的系统。

自己 debug 调试整个系统来观察具体类的作用是什么：在 nachos.machine 中的 Machine 相当于整个系统的主机，提供的功能有：读写寄存器、读写主存、运行一条用户程序对的汇编指令、运行用户程序、单步调试用户程序等。Machine 类实现方法是在宿主机上分配了两块内存分别作为虚拟机的寄存器和物理内存。运行用户程序时，先将用户程序从 nachos 文件系统中读出，写入模拟的物理内存中，然后调用指令模拟模块对每一条用户指令解释执行。将用户程序的读写内存要求，转变为对物理内存地址的读写。

一开始调试时在 machine 类中设置断点：

```
new TCB().start(new Runnable() {
        public void run() {
            autoGrader.start(privilege);
            }
    });
        }
```

一步步调试发现，在 nachos 中的线程实际上是由三个部分组成的：TCB、KThread 以及软件所真正创建的线程（java 线程）。如果创建的是第一个，则顺序是：javathread、TCB、KThread。会进行判断是否是第一个进程如果是首先使用 Thread.currentThread() 方法创建一个 java 线程，赋值给 javathread 变量，之后调用 threadroot 方法，在该方法中我们可以看到，第一次时会将 tcb 赋值给 currentTCB 变量，设置状态为 true：

```
 private void threadroot() {
   // this should be running the current thread
   Lib.assertTrue(javaThread == Thread.currentThread());

   if (!isFirstTCB) {
       currentTCB.interrupt();
       this.yield();
   }
   else {
       currentTCB = this;//第一次将 tcb 赋值给 currenttcb，因为第一次创建顺序为：javathread、tcb、kthread
       running = true;
   }

   try {
       target.run();
       // no way out of here without going throw one of the catch blocks
       Lib.assertNotReached();

   }
```

在 autoGrader 的 start 方法执行时，autoGrader 首先会解析启动命令传入的参数，接着进行初始化操作，然后从配置文件中读取 Kernel.kernel 的值，构造内核，并且执行内核的初始化方进阶着执行 run：

一开始在 machine 中设置断点处的线程参数：继续 run：

```
 void run() {

   kernel.selfTest();//测试线程之间是否正常。

   kernel.run();

   kernel.terminate();

    }
```

在跟进调试后发现它继续执行 selftest 方法，我们的 KThread 类和 UserKernel 都具体实现了这个方法，即不同 project 使用的内核不同，所以各个内核的效果也是不一样的。对于 project 1 实验整个过程中我的测试函数都在 KThread 中进行编写的，因为在 kernel 抽象类中 ，会检查 Kernel 中的静态成员变量 kernel 是否为空，并把当前对象赋值给 Kernel，这也决定了内核可以通过 kernel.kernel 调用。在构造器结束后，autograder 会执行 kernel 的 initialize 方法，对于 project1 的 ThreadedKernel 而言，在该方法中初始化了调度器，文件系统，alarm 等。在执行完 initialize 方法后，会进入 autograder 中的 run 方法，一次执行内核自检、运行、停机操作。

对于 project 1 的 ThreadedKernel 而言，run 方法是空的，也就是执行完开机检测后就自动停机了。具体的实验内容在下面介绍。

在调试过程中我对开中断以及关中断做了详细的研究，发现在很多操作都会进行中断的开关：

在 nachos 中有个 Interrupt 类，是用来模拟硬件中断系统的，在这个中断系统中，中断状态有开，关两种，中断类型有时钟中断、磁盘中断、控制台写中断、控制台读中断、网络发送中断以及网络接收中断。机器状态有用户态，核心态和空闲态。中断系统提供的功能有开/关中断，读/写机器状态，将一个即将发生中断放入中断队列，以及使机器时钟前进一步。

在 Interrupt 类中有一个记录即将发生中断的队列，称为中断等待队列。中断等待队列中每个等待处理的中断包含中断类型、中断处理程序的地址及参数、中断应当发生的时间等信息。一般是由硬件设备模拟程序把将要发生的中断放入中断队列。中断系统提供了一个模拟机器时钟，机器时钟在下列情况下前进

1 用户程序打开中断

2 执行一条用户指令

3 处理机没有进程正在运行

所以，在 Nachos 中只有在时钟前进时，才会检查是否有中断会发生，而 Nachos 模拟时钟前进的时机不是任意的，这样即使打开了中断，中断也不能在

任意时刻发生。只有在模拟时钟前进的时候才能处理等待着的中断。通过以后的叙述我们可以看到，在执行非用户代码的大部分时间里，系统不会被中断。这意味着不正确的同步代码可能在这个硬件模拟环境下工作正常，而实际上在真正的硬件上是无法正确运行的。

在一开始的时候会发现创建第一个线程之后会创建一个闲逛线程 private static void createIdleThread() {

```
Lib.assertTrue(idleThread == null);

idleThread = new KThread(new Runnable() {

    public void run() { while (true) yield(); }

});

idleThread.setName("idle");

Machine.autoGrader().setIdleThread(idleThread);

idleThread.fork();

 }
```

这个线程的作用是在没有要执行的线程时不断获取资源再不断放弃。注意这是整个系统的第二个线程，其创建的顺序不同于第一个线程了，先 KThead、再 TCB、后 java 线程。在一个线程调用 TCB.contextSwitch（）时，该线程停止执行，另一个线程开始运行。

KThread nachos 的一个线程，但是用 java 写的，所以具体的实现还是依靠 java，tcb 指硬件实现，具体指一个 KThread 对应一个 tcb，通过 tcb 来操作 java 线程。

4.makefile 文件的理解:

在 nachos 中还有个重要的文件 makefile，由于老师在课堂上介绍的比较简略，所以查阅了大量的资料，以下是我对 makefile 的理解：

makefile 关系到了整个工程的编译规则。一个工程中的源文件不计数，其按类型、功能、模块分别放在若干个目录中，makefile 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，甚至于进行更复杂的功能操作，因为 makefile 就像一个 Shell 脚本一样，其中也可以执行操作系统的命令。

makefile 带来的好处就是——“自动化编译”，一旦写好，只需要一个 make 命令，整个工程完全自动编译，极大的提高了软件开发的效率。make 是 一个命

令工具，是一个解释 makefile 中指令的命令工具，一般来说，大多数的 IDE 都有这个命令，比如：Delphi 的 make，Visual C++的 nmake，Linux 下 GNU 的 make。可见，makefile 都成为了一种在工程方面的编译方法。

Makefile 里主要包含了五个东西：显式规则、隐晦规则、变量定义、文件指示和注释。

1、显式规则。显式规则说明了，如何生成一个或多的的目标文件。这是由 Makefile 的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。

2、隐晦规则。由于我们的 make 有自动推导的功能，所以隐晦的规则可以让我们比较粗糙地简略地书写 Makefile，这是由 make 所支持的。

3、变量的定义。在 Makefile 中我们要定义一系列的变量，变量一般都是字符串，这个有点你 C 语言中的宏，当 Makefile 被执行时，其中的变量都会被扩展到相应的引用位置上。

4、文件指示。其包括了三个部分，一个是在一个 Makefile 中引用另一个 Makefile，就像 C 语言中的 include 一样；另一个是指根 据某些情况指定 Makefile 中的有效部分，就像 C 语言中的预编译#if 一样；还有就是定义一个多行的命令。有关这一部分的内容，我会在后续的部分中讲 述。

5、注释。Makefile 中只有行注释，和 UNIX 的 Shell 脚本一样，其注释是用“#”字符，这个就像 C/C++中的“//”一样。如果你要在你的 Makefile 中使用“#”字符，可以用反斜框进行转义，如：“\#”。

最后，还值得一提的是，在 Makefile 中的命令，必须要以[Tab]键开始。

原文对 makefile 的介绍如下:

Experienced programmers are familiar with makefiles. A makefile stores compiler and linker options and expresses all the interrelationships among source files. (A source code file needs specific include files, an executable file requires certain object modules and libraries, and so forth.) A make program reads the makefile and then invokes the compiler, assembler, resource compiler, and linker to produce the final output, which is generally an executable file. The make program uses built-in inference rules that tell it, for example, to invoke the compiler to generate an OBJ file from a specified CPP file.

从上面不难看出，makefile 文件保存了编译器和连接器的参数选项,还表述了所有源文件之间的关系(源代码文件需要的特定的包含文件,可执行文件要求包含的目标文件模块及库等).创建程序(make 程序)首先读取 makefile 文件,然后再激活编译器,汇编器,资源编译器和连接器以便产生最后的输出,最后输出并生成的通常是可执行文件.创建程序利用内置的推理规则来激活编译器,以便通过对特定 CPP 文件的编译来产生特定的 OBJ 文件.

在默认的方式下，也就是我们只输入 make 命令。那么：

1、make 会在当前目录下找名字叫“Makefile”或“makefile”的文件。

2、如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到“edit”这个文件，并把这个文件作为最终的目标文件。

3、如果 edit 文件不存在，或是 edit 所依赖的后面的 .o 文件的文件修改

时间要比 edit 这个文件新，那么，他就会执行后面所定义的命令来生成 edit 这个文件。

4、如果 edit 所依赖的.o 文件也不存在，那么 make 会在当前文件中找目标为.o 文件的依赖性，如果找到则再根据那一个规则生成.o 文件。（这有点像一个堆栈的过程）

5、当然，你的 C 文件和 H 文件是存在的啦，于是 make 会生成 .o 文件，然后再用 .o 文件生成 make 的终极任务，也就是执行文件 edit 了。

这就是整个 make 的依赖性，make 会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件。在找寻的过程中，如果出现错误，比如最后被依赖的文件找不到，那么 make 就会直接退出，并报错，而对于所定义的命令的错误，或是编译不成功，make 根本不理。make 只管文件的依赖性，即，如果在我找了依赖关系之后，冒号后面的文件还是不在，那么对不起，我就不工作啦。

通过上述分析，我们知道，像 clean 这种，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，我们可以显示要 make 执行。即命令——“make clean”，以此来清除所有的目标文件，以便重编译。

于是在我们编程中，如果这个工程已被编译过了，当我们修改了其中一个源文件，比如 file.c，那么根据我们的依赖性，我们的目标 file.o 会被重编译（也就是在这个依性关系后面所定义的命令），于是 file.o 的文件也是最新的啦，于是 file.o 的文件修改时间要比 edit 要新，所以 edit 也会被重新链接了（详见 edit 目标文件后定义的命令）。

而如果我们改变了“command.h”，那么，kdb.o、command.o 和 files.o 都会被重编译，并且，edit 会被重链接。

二、Project 1

一、实现 KThread.join():

***开始的开始：**

在开始做本实验前，虽然之前对 nachos 整体有了较为细致的了解，但是具体在操作中刚开始还是有点懵，所以我又去伯克利官网上看了相应的 api 介绍，对 KThread.java 的所有函数进行了一定的了解：

Method Summary	
int	compareTo(Object o) Deterministically and consistently compare this thread to another thread.
static KThread	currentThread() Get the current thread.
static void	finish() Finish the current thread and schedule it to be destroyed when it is safe to do so.
void	fork() Causes this thread to begin execution.
String	getName() Get the name of this thread.
void	join() Waits for this thread to finish.
void	ready() Moves this thread to the ready state and adds this to the scheduler's ready queue.
protected void	restoreState() Prepare this thread to be run.
protected void	saveState() Prepare this thread to give up the processor.
static void	selfTest() Tests whether this module is working.
KThread	setName(String name) Set the name of this thread.
KThread	setTarget(Runnable target) Set the target of this thread.
static void	sleep() Relinquish the CPU, because the current thread has either finished or it is blocked.
String	toString() Get the full name of this thread.

其中的 join 函数就是我们要实现的，但在准备写的时候发现 join 只能实现将一个线程挂起放入另一个线程的等待队列中去，但是该被挂起的线程也是需要执行的，后来发现 finish 方法，也就是每个线程执行结束后会自动调用的一个方法，我觉得单个 join 方法无法完成实验 1 的要求，必须得结合 finish 一起修改才可以实现真正的完成。

a、问题介绍及实验要求：

实现 join（）方法，其他线程没必要调用 join 函数，但是如果它被调用的话，也只能被调用一次。join（）方法第二次调用的结果是不被定义的，即使第二次调用的线程和第一次调用的线程是不同的。无论有没有被 join，一个进程都能够正常结束。

b、具体代码实现及分析：

需要注意的是 join 方法要和 finish 方法综合来用，当调用 join 方法时应该将当前正在执行的进程先放入到调用进程的等待队列，之后将当前进程挂起，之后调度程序再从就绪队列中去调度新的进程来执行，当执行了刚刚调用函数的那个进程时，在其执行完自动退出时会去检查它的等待队列，如果其等待队列有进程，则将该进程重新唤醒，放入就绪队列中。需要注意的是，由于 我在测试的时候只创建了一个进程，加上 main 进程（不考虑闲逛进程），总共只有两个进程，所以在时间上看，就是在 B 中 A.join(),会立马执行 A，再继续执行 B，但是实际上只是将其唤醒加入就绪队列，并不一定是立马执行。

实现代码：

```
public static void finish() {
      Lib.debug(dbgThread, "Finishing thread: "
+currentThread.toString());
      //系统关中断
      boolean preStatus = Machine.interrupt().disable();
      //唤起在等待队列的所有线程
      KThread th = null;
      do{
         th = myWaitQueue.nextThread();
          if(th != null) {
          th.ready();
          }
       }while(th != null);
      Machine.autoGrader().finishingCurrentThread();
       Lib.assertTrue(toBeDestroyed == null);
      toBeDestroyed = currentThread;
      currentThread.status = statusFinished;
      sleep();
      Machine.interrupt().restore(preStatus);
   }
```

join 方法的实现：

```
 public void join() {
   Lib.debug(dbgThread, "Joining to thread: " + toString());
    //System.out.println(this.name+" "+currentThread.name);
   Lib.assertTrue(this != currentThread);
   if(status == statusFinished ) {
       return;
    }
    boolean preStatus = Machine.interrupt().disable();
     myWaitQueue.waitForAccess(currentThread);
     sleep();
```

```
    //关中断
     Machine.interrupt().restore(preStatus);
    }
```

c、测试代码：

```
 public static void selfTest() {
       Lib.debug(dbgThread, "Enter KThread.selfTest");
       selfTest_1();

    }
    public static void selfTest_1() {
    //Lib.debug(dbgThread, "Enter KThread.selfTest");

    KThread th1=new KThread(new PingTest(1));
    th1.setName("forked thread").fork();
    //创建一个 KThread-tcb-javathread(即一个新的线程)来执行
PingTEST(1)程序，
    th1.join();
    new PingTest(0).run();//在现有线程中执行 pingtest 的 run 方法。

        }
```

d、测试结果及分析：

打印出了第一个执行线程的名字和第二个执行线程的名字，先执行主线程 thread0，但是在执行 run 之前，使用了 join，则主线程挂起，执行 th1。执行完后，调用 finish 方法会唤醒之前挂起的主线程，继续往下执行：

```
nachos 5.0j initializing... config interrupt timer user-check grader
forked thread main
*** thread 1 looped 0 times
*** thread 1 looped 1 times
*** thread 1 looped 2 times
*** thread 1 looped 3 times
*** thread 1 looped 4 times
*** thread 0 looped 0 times
*** thread 0 looped 1 times
*** thread 0 looped 2 times
*** thread 0 looped 3 times
*** thread 0 looped 4 times
```

注意：在使用 join 的时候，创建的等待队列使用的是 nachos 所给的等待队列：

```
    private static ThreadQueue myWaitQueue =
ThreadedKernel.scheduler.newThreadQueue(true);
```

为什么在这里使用的是这个，是因为和第五问结合起来使用的，后面的参数设置为 true，表示可以进行优先级反转。

e、收获与感想：

join()函数的作用是等待一个线程执行完毕后再继续执行，比如对于线程 a

和线程 b 并发运行，线程 b 调用 a.join()后，则线程 b 会被挂起，等待 a 线程执行完毕后 b 线程继续执行。

然而每个 KThread 线程只能调用一次 join()函数，多次调用没有相关的定义，而且线程自身不能调用自己的 join()函数。所以，完成该目标，需要解读提供的系统代码，在理解的基础上去自己实现系统中的 join()函数及其相关的函数 finish()定义，完善相关的逻辑设计并在 KThread.selfTest()中实现对于该系统调用的实际线程的测试。

二、实现 project1 中的 Condition2 类

a. 实验题目、实验要求：

直接实现条件变量，通过使用中断启用和禁用提供原子。我们已经提供了使用信号的样本实施;你的工作是提供等效实施而不直接使用信号量（你可能当然仍然使用锁，即使他们间接使用信号灯）。一旦你完成，你将有两种可供选择的实现，提供完全相同的功能。你的第二个实施条件变量必须驻留在类 nachos.threads.Condition2。

b. 问题分析及代码实现：

在实现的时候也要用到队列，但这个时候的队列并无什么要求所以我用到的是 java 类库中自带的一个：

```
private LinkedList<KThread> myWaitQueue=new LinkedList<KThread>();
```

sleep 方法因为要保证原子性所以在在调用开始的时候关中断，之后将线程加入队列中，我使用的是 offer 方法，就是加入，但现在官网上推荐这个方法在此我也安利一下。

```
public void sleep() {
Lib.assertTrue(conditionLock.isHeldByCurrentThread());
 boolean preStatus = Machine.interrupt().disable();  //关中断
 //System.out.println(KThread.currentThread().getName()+"要睡觉了");
 myWaitQueue.offer(KThread.currentThread());
 conditionLock.release();
 KThread.sleep(); //让调用本方法的线程 sleep
 conditionLock.acquire();
//返回之前的状态
 Machine.interrupt().restore(preStatus); //开中断

}
```

与 sleep 方法进行对比来着的此方法，同样也是要保证原子性，所以函数

开始处要开中断，最后关中断。

```
public void wake() {
    Lib.assertTrue(conditionLock.isHeldByCurrentThread());
    //系统关中断
    boolean preStatus = Machine.interrupt().disable();
    //唤起在等待队列的所有线程
    KThread th = null;
    th = myWaitQueue.poll();//取出队首的元素

            if(th != null) {
               System.out.println("要唤醒线程的名字:
"+th.getName());
            th.ready();  }//把线程放入就绪队列
            //else System.out.println("没有多余的人");
    Machine.interrupt().restore(preStatus);

        }

 public void wakeAll() {
    Lib.assertTrue(conditionLock.isHeldByCurrentThread());
    //系统关中断
    boolean preStatus = Machine.interrupt().disable();
    //唤起在等待队列的所有线程
    KThread th = null;
    do{
    th = myWaitQueue.poll();//取出队首的元素
            if(th != null) {
            th.ready();  }//把线程放入就绪队列
     }while(!myWaitQueue.isEmpty());
    Machine.interrupt().restore(preStatus);

        }
```

在测试程序之前，我在原先的 Pingtest 类中增加了一个构造函数：

```
PingTest(int which,Condition2 condition,Lock lock) {
        this.which = which;
        this.co=condition;
        this.l=lock;
           }
```

也就是会传进来一个 condition 和一个锁，所以会让两个线程共用一个条件变量及锁。

C、测试代码：

```
public static void selfTest_2() {
        Lock conditionLock=new Lock();
        Condition2 condition=new Condition2(conditionLock);
        KThread th1=new KThread(new
PingTest(1,condition,conditionLock));
        th1.setName("shiyan1.2").fork();//fork 后该线程会加入到就绪
队列 ，并不立即执行
        System.out.println("主线程要睡觉了");
        conditionLock.acquire();
        condition.sleep();
        conditionLock.release();
        System.out.println("主线程被唤醒了");

         }
```

d、测试结果及分析：

```
<terminated> Machine [Java Application] C:\Program Files\Java\jdk1.7.0_79\bin\javaw.exe (2016年11月23日 上
nachos 5.0j initializing... config interrupt timer user-check grader
主线程要睡觉了
*** thread 1 looped 0 times
*** thread 1 looped 1 times
*** thread 1 looped 2 times
*** thread 1 looped 3 times
*** thread 1 looped 4 times
要唤醒线程的名字：main
主线程被唤醒了
Machine halting!
```

观察上面的结果会发现首先执行主线程，之后主线程 sleep，因为要保证调用 sleep 方法的原子性，所以在先获得锁，但为了保证锁可以在本线程睡眠后能被另一个线程获得，所以在 sleep 方法中在真正调用函数睡眠时会先释放一下锁，这就保证了另外一个进程可以获得锁。

e. 收获与感想：

• 重点的知识：某个信号量拥有自己的等待队列，只能自己 wake 自己 sleep 的线程。

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，另一种线程使“条件成立”（给出条件成立信号）。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。

Thread.Lock 类提供了锁以保证互斥。在临界代码区的两端执行 Lock.acquire 和 Lock.release 即可保证同时只有一个线程访问临界代码区。

条件变量建立在锁之上，由 thread.Condition 实现，用来保证同步。每一个条件变量拥有一个锁变量。

• sleep 方法在条件变量的控制下函数自动释放相关锁并进入睡眠状态，直到另一个线程用 wake 方法唤醒它，需要在函数的开始和结尾处关、开中断以保证原子性。当前线程必须拥有相关锁，阻塞进程在 sleep 函数返回之前线程将会自动再次拥有锁。即调用这个方法的线程将自己挂起等待队列，阻塞自己的同时将自己拥有的锁交出去。之后等待锁的分配。拿到锁的线程再次拥有权限接受检验。

• wake 唤醒条件变量队列中第一个线程，在实验中利用从线程队列中取出线程用 ready 唤醒并加入就绪队列。wake'函数也要保证原子性。

• wakeall 的实现就是将等待队列中所有的线程依次唤醒，用 wake 方法。

三、实现 Alarm

a. 实验题目、实验要求：

完善和实现 Alarm 类，通过利用 waitUntil(long x)方法。当一个线程调用 waitUntil 方法后被挂起 x 毫秒，直到时间到达后才会接着执行。这样的调度线程策略在实时的系统中是非常有效的。当然，这里的线程在 x 毫秒后不一定会立即执行，只需要将其添加到就绪队列中就好。对于不同的多个线程可以分别调用 waitUntil 函数。我们要做的是去实现 waitUntil 方法。

b. 问题分析及代码实现：

waitUntil()

进程可调用的函数，在该函数中保存函数的进程名和进程唤醒时间。设计步骤如下：首先关闭中断保证原子性，然后创建一个 Waiter 类的对象类存储当前信息，然后再开中断。

```
 public void waitUntil(long x) {
       //系统关中断
    boolean preStatus = Machine.interrupt().disable();
    long wakeTime = Machine.timer().getTime() + x;
    NewKThread newkthread=new
NewKThread(KThread.currentThread(),wakeTime);
    myWaitQueue.offer(newkthread);

    KThread.sleep();
    Machine.interrupt().restore(preStatus);//开中断
    }
     private LinkedList<NewKThread> myWaitQueue=new
LinkedList<NewKThread>();

  class NewKThread{
     public KThread thread;
     public long wakeTime;
     public NewKThread(KThread thread,long wakeTime){
```

```
        this.thread=thread;
        this.wakeTime=wakeTime;
      }
       }
```

timerInterrupt()

实现将阻塞队列中的线程在正确的时间按照正确的顺序唤醒线程。为了能够对队列中的线程查找判断，需要循环遍历等待队列。遍历过程中，如果该进程的唤醒时间大于当前的时间，则继续循环，否则就唤醒该进程，并将其从等待队列中移除。

```
 public void timerInterrupt() {
    boolean preStatus = Machine.interrupt().disable();
    Stack<Integer> xuhao=new Stack<Integer>();
    for(int i=0;i<myWaitQueue.size();i++){

   if(myWaitQueue.get(i).wakeTime<=Machine.timer().getTime())
       {
          xuhao.push(i);
       }
    }
    while(!xuhao.isEmpty()){
       int temp=xuhao.pop();
       NewKThread t=myWaitQueue.get(temp);
       myWaitQueue.remove(t);

       t.thread.ready();
    }
    Machine.interrupt().restore(preStatus);
   KThread.currentThread().yield();
        }
```

NewKThread 类

用来存储要阻塞的线程的名和睡眠时间。

```
class NewKThread{
     public KThread thread;
     public long wakeTime;
     public NewKThread(KThread thread,long wakeTime){
        this.thread=thread;
        this.wakeTime=wakeTime;
     }
     }
```

C、测试代码：

```
 public static void selfTest_3() {
       Alarm alarm=new Alarm();
       alarm.waitUntil(10000);
```

```
        System.out.println("main 重新开始执行了！ ");
        System.out.println("现在是： "+Machine.timer().getTime());
        }
```

d、测试结果及分析：

```
nachos 5.0j initializing... config interrupt timer user-check grader
沉睡时间：10
唤醒时间：10100
main重新开始执行了！
现在是：10110
Machine halting!
```

可以看到在 10 的时候沉睡，之后沉睡时间设置的是 10000，所以在 100010 的时候唤醒，但这个时候是将该线程加入就绪队列，并不立马执行，之后真正执行的时候打印出执行时刻，可以看到这两个时间之间存在一定的时间差。

注意：

a.线程进入等待的规律是：早调用 waiUntil 方法的线程早进入等待，如果同一时刻有两个线程都满足等待完成条件，由于早调用方法的线程更早进入等待序列，这意味着他等待了更久的时间，那么他会早一步离开等待序列，这满足了题目的要求。

b.要检查调用函数的线程等待时间是不是已经过去了

e. 收获与感想：

要完善和设计 Alarm 类，了解和 Alarm 类相关的 machine.Timer 类，这个类是整个系统的时钟类，它每 500 个时钟调用回调函数。

实现 waitUntil，需要在 Alarm 类中实现一个队列 myWaitQueue，队列中的每个项目是（线程，唤醒时间）的二元组．在调用 waitUntil(x) 函数时，首先得到关于以上二元组的信息：(线程：当前线程，唤醒时间：当前时间+x)，然后将该元组放入队列中，并对条件变量 sleep 操作使当前线程挂起．在时钟回调函数中（大约每 500 个时钟间隔调用一次）则依次检查队列中的每个三元组．如果唤醒时间大于当前时间,则将元组移出队列并对元组中的条件变量执行 wake 操作将相应线程唤醒．

四、实现 Communicator 类

a. 实验题目、实验要求：

Communicator 对象中的 speak 和 listen，speak 一直处于阻塞监听状态知道属于同一个对象的 listen 被调用，然后再将这个信息传输给 listen。一旦传输完成，这两个方法都要返回。相同的，speak 被调用时候，同样的传输动作发生。也就是说任何一个线程都要等待 speak 和 listen 传输完成后才能返回。

Communicator 上有多个 spaker 和 listener 的情形．此时的 speaker 和 listener 只能是一对一的，即一个 speaker 只能将数据发送到一个 listener，一个 listener 也只能接收来自一个 spekaer 的数据，其余的 speakers 和 listeners 都需要等待。

b. 问题分析及代码实现：

代码：

```
 public void speak(int word) {
       lock.acquire();
       if(listener>0)//有听者
       {   words.offer(word);
          listen.wake();
       }
       else{
          words.offer(word);
          speak.sleep();//让说的那个线程等待
          }
       lock.release();
    }

public int listen() {
       lock.acquire();
       if(!words.isEmpty()){
          speak.wake();//把说者加入就绪队列中
          lock.release();
          return words.poll();
       }else
          {listener++;
          listen.sleep();
          listener--;
          lock.release();
          return words.poll();
          }
    }
```

增加了一组变量：

```
   private int listener=0;
   private LinkedList<Integer> words=new LinkedList<Integer>();
   private Lock lock=new Lock();
   private Condition2 listen=new Condition2(lock);//listen
   private Condition2 speak=new Condition2(lock);//speak
```

C、测试代码：

```
public static void selfTest_4() {
    Communicator com=new Communicator();
    KThread th1=new KThread(new PingTest(4,com));
   th1.setName("shiyan1.4").fork();//fork 后该线程会加入到就绪队
列 ， 并不立即执行
```

```
//     KThread th2=new KThread(new PingTest(3,com));
//     th2.setName("shiyan1.42").fork();//fork 后该线程会加入到就
绪队列 ，并不立即执行
    System.out.println("我是听者!");
    int result=com.listen();
    System.out.println("我听到说者说了： "+result);
    }
```

d、测试结果及分析：

```
nachos 5.0j initializing... config interrupt timer user-check grader
我是听者!
我是说者，我说话了！说的内容是123
要唤醒线程的名字：main
我听到说者说了：123
Machine halting!

Ticks: total 2080, kernel 2080, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: page faults 0, TLB misses 0
Network I/O: received 0, sent 0
```

e. 收获与感想：

每个 speak 一个 Word，listen 接受一个 Word，并返回，这两个操作一对一操作并且要求互斥，所以每个 Communicator 有一个锁（保证操作的原子性）和与该锁联系的两个条件变量 s,l 用于保证 speaker 和 listener 间的同步. 在 speak 函数中 ，首先检查若已经有一个 speaker 在等待(prepareData 变量) 或无 listener 等待，则挂起. 否则设置 prepareData 变量，准备数据并唤醒一个 listener. 在 listen 函数中，增加一个 listener 后，首先唤醒 speaker，然后将自己挂起以等待 speaker 准备好数据再将自己唤醒. 这个问题其实是一个缓冲区长度为 0 的生产者/消费者。

五、实现 Priority：

a. 实验题目、实验要求：

完成 PriorityScheduler 类在 Nachos 中实现优先级调度。实现中，需要注意的是：为了完成优先级调度需要修改 nachso.conf 文件中的相应行以启动指定的调度类， ThreadedKernel.scheduler 中的 key 按 nachos.threads.RoundRobinScheduler 一样被初始化。当修改完 nachos 文件后，要实现 getPriority(),getEffectivePriority()和 setPriority()函数，同时可以选择实现 increasePriority()和 decreasePriority()函数。在这种调度中，选择具有最高优先级的线程来执行，如果多个线程具有相同的优先级则调度其中在队列中等待最长时间的那个。另外，需要值得注意的是，优先级调度的优先级馈赠问题，就是如果一个高优先级的线程需要等待低优先级的，

另一个次高优先级的线程到达后，那么这个次高优先级的线程永远得不到 cpu，因为高优先级的线程得不到 cpu 执行。为了实现优先级的馈赠，需要实现 Scheduler.getEffectivePriority()。

b. 问题分析及代码实现：

首先实现基本的优先级调度：分析 PriorityScheduler 类，发现其中又包含了两个内部类：

①：PriorityQueue，其注释为——根据优先级将线程分类。我们用一个大根堆来存储所有的线程及其优先级，每次其堆顶是优先级最高的线程，我们将将 KTread 对象与其相应的优先级（EffectivePriority）加入大根堆中，根据优先级大小排序。该类主要需实现的方法有 nextThread()、pickNextThread（）。

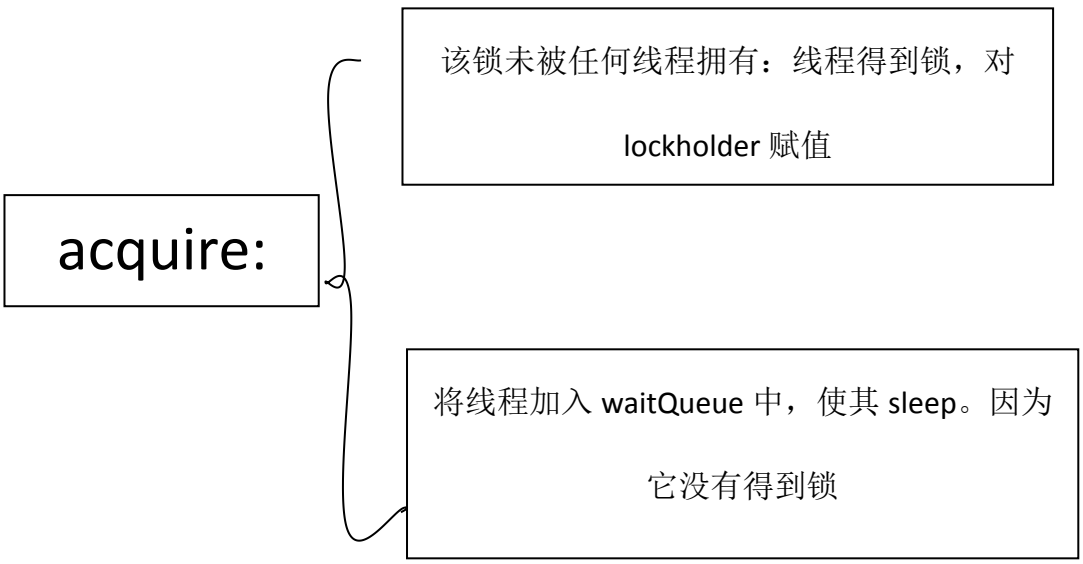
②：ThreadState——保存一个线程和它的优先级、EffectivePriority、waitQueue。主要需要实现的方法：getEffectivePriority（）、setPriority(int priority)、waitForAccess(PriorityQueue waitQueue)、acquire()。

逐一实现：此处对一些较为复杂的实现方法进行说明：ThreadState 中的 setPriority 方法：当我们设置一个线程的优先级时，如果该线程原先的优先级小于重新设置的，则需要更改 EffectivePriority，此时大根堆需要刷新一遍。并且如果此时的锁具有拥有者且其优先级低于新设置值，说明在堆中（即就绪队列）存在比 lockholder 高优先级的线程，要对 lockholder 进行优先级反转。waitForAccess(PriorityQueue waitQueue)方法：获得就绪队列，向其加入线程，当就绪队列满足优先级反转的条件进行优先级反转。当调用本类中的 acquire（）方法，说明调用该方法的线程拥有锁。

PriorityQueue 中 nextThread()方法是获得最大优先级的线程并将其设置成锁的拥有者（lockholder）。

此外重点说明 getThreadState(KThread thread)方法：返回包含参数线程的 ThreadState 对象。

代码基本实现后简单介绍一下程序整个逻辑。由于是通过锁 Lock 来完成整个线程的调度，查看 Lock 类：



同样的 release 方法：从等待锁的队列中选择一个线程，唤醒它。（等价于给一个线程锁）

增加了一个封装线程及其优先级的变量：

```
protected class WaitingItem{
        protected KThread thread;
        protected long time;
        public WaitingItem(KThread thread,long time){
            this.thread=thread;
            this.time=time;
        }
    }
```

Proriority 类的实现：

```
//MOD
        protected ThreadState lockHolder;
        protected java.util.PriorityQueue<WaitingItem> queue;
        //MOD

        PriorityQueue(boolean transferPriority) {
            this.transferPriority = transferPriority;

            //MOD
            queue=new java.util.PriorityQueue<>(1,new
Comparator<WaitingItem>() {

                @Override
                public int compare(WaitingItem o1, WaitingItem o2) {
                    ThreadState ts1=getThreadState(o1.thread);
                    ThreadState ts2=getThreadState(o2.thread);

    if(ts1.getEffectivePriority()>ts2.getEffectivePriority()){
                        return -1;
                    }else
if(ts1.getEffectivePriority()<ts2.getEffectivePriority()){
                        return 1;
                    }else{
                        if(o1.time>o2.time){
                            return 1;
                        }else if(o1.time<o2.time){
                            return -1;
                        }else{
                            return 0;
                        }
                    }
                }
```

```
            });
            //MOD
        }
public KThread nextThread() {
            Lib.assertTrue(Machine.interrupt().disabled());
            // implement me DONE
            WaitingItem item = queue.poll();
            if(item==null)
                return null;
            KThread thread = item.thread;
            getThreadState(thread).waitQueue=null;
            getThreadState(thread).acquire(this);
            return thread;
        }
protected ThreadState pickNextThread() {
            // implement me DONE
            if(queue.isEmpty())
                return null;
            return getThreadState(queue.peek().thread);
        }
```

ThreadState 实现：

```
        protected int effectivePriority=-1;
        protected PriorityQueue waitQueue;
public void setPriority(int priority) {
            if (this.priority == priority)
                return;

            this.priority = priority;

            // implement me DONE
            if(waitQueue!=null && waitQueue.transferPriority){
                if(priority > getEffectivePriority()){
                    effectivePriority=priority;
                }
                if(waitQueue.lockHolder!=null){

    if(waitQueue.lockHolder.getEffectivePriority()<getEffective
Priority()){
                        WaitingItem wi=waitQueue.queue.poll();

    waitQueue.lockHolder.effectivePriority=getEffectivePriority
();
                        if(wi!=null){
                            waitQueue.queue.offer(wi);
```

```
                    }
                }
            }
        }else{
            effectivePriority=priority;
        }
    }
public void waitForAccess(PriorityQueue waitQueue) {
        // implement me DONE
        this.waitQueue=waitQueue;
        if(waitQueue.transferPriority){
            if(waitQueue.lockHolder!=null){

   if(waitQueue.lockHolder.getEffectivePriority()<getEffective
Priority()){

   waitQueue.lockHolder.effectivePriority=getEffectivePriority
();
                }
            }
        }
        waitQueue.queue.offer(new WaitingItem(thread,
Machine.timer().getTime()));
    }
public void acquire(PriorityQueue waitQueue) {
        // implement me DONE
        waitQueue.lockHolder=getThreadState(thread);
    }
```

C、测试代码：见下图

d、测试结果及分析：

```
public static void selfTest_5() {
    Lock lock=new Lock();
    KThread th1=new KThread(new PingTest(51,null,lock));
    th1.setName("shiyan1.51").fork();//fork后该线程会加入到就绪队列，并不立即执行
    KThread th2=new KThread(new PingTest(52,null,lock));
    th2.setName("shiyan1.52").fork();//fork后该线程会加入到就绪队列，并不立即执行
    boolean preStatus = Machine.interrupt().disable();
    ThreadedKernel.scheduler.setPriority(1);
    ThreadedKernel.scheduler.setPriority(th1, 4);
    ThreadedKernel.scheduler.setPriority(th2, 5);
    lock.acquire();//主线程拿到锁
    System.out.println("main1 " + ThreadedKernel.scheduler.getEffectivePriority());
    yield();
    System.out.println("main2 " + ThreadedKernel.scheduler.getEffectivePriority());
    System.out.println("th1 " + ThreadedKernel.scheduler.getEffectivePriority(th1));
    lock.release();
    Machine.interrupt().restore(preStatus);
}
```

```
*** thread 52 looped 0 times
*** thread 52 looped 1 times
*** thread 52 looped 2 times
*** thread 52 looped 3 times
*** thread 52 looped 4 times
*** thread 51 looped 0 times
*** thread 51 looped 1 times
*** thread 51 looped 2 times
*** thread 51 looped 3 times
*** thread 51 looped 4 times
Machine halting!
```

结果：

开始主线程执行，获取到锁，此时主线程优先级最低，为 1。进行 yield（）后，调度程序开始选择线程执行：注意此时 main 线程的优先级低但由于其拥有锁，进行了优先级反转并执行。主线程执行并释放锁，此时从就绪队列中找到优先级最大的线程，给其锁并执行，所以先执行 th2（优先级为 5）.打印出主线程的 EffectivePriority，会发现它变成了 5。

e. 收获与感想：

题目要求先是完成 PriorityScheduler 类在 Nachos 中实现优先级调度。首先在这种调度下，每次调度程序从就绪队列中应该选择优先级最高的那个。此处会出现一种需要我们解决的问题：优先级反转——持有锁的线程的优先级低于就绪队列中的其他线程，则我们将最高优先级暂时性地赋予给之，低优先级的 lockholder 变成高优先级线程（暂时性），在代码中我们除了 priority 属性，特设有 EffectivePriority 一属性，在优先级反转过程中用到该变量。

六、小船问题：

a. 实验题目、实验要求：

使用条件变量解决过河问题。O 岛有一群人要到 M 岛去，但是只有一艘船，这艘船一次只能带一个大人或者两个孩子。开始必须假设至少有两个孩子（否则问题无法解决），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。

b. 问题分析及代码实现：

```
public class Boat
{
    static BoatGrader bg;
    private static KThread parentThread;// 父进程
    private static int children_number_O;// 在 O 岛上孩子的数量
    private static int adult_number_O;// 在 O 岛上成人的数量
    private static int children_number_M;// 在 M 岛上孩子的数量
    private static int adult_number_M;// 在 M 岛上成人的数量
    private static Condition2 children_condition_O;// 孩子在 O 岛上的条件变量
    private static Condition2 children_condition_M;// 孩子在 M 岛上的条件变量
    private static Condition2 adult_condition_O;// 成人在 O 岛上的条件变量

    private static Lock lock;
    private static boolean is_adult_go;// 判断是否该成人走：O 上还有大人、船上有位置、M 岛有小孩
    private static boolean boat_in_O;// 判断船是否在 O
    private static boolean is_pilot;// 判断现在的孩子 是不是驾驶员
    private static boolean finished;// 判断运送是否结束

    public static void selfTest()
    {
    BoatGrader b = new BoatGrader();

    System.out.println("\n ***Testing Boats with 3 children,4 adults***");
    begin(4, 3, b);
    }

    public static void begin( int adults, int children, BoatGrader b )
    {
    bg = b;
    parentThread = KThread.currentThread();

    for (int i = 0; i < adults; i++)
        {
        new KThread(new Runnable() {
```

```
            public void run() {
                AdultItinerary();
            }
        }).setName((i+1)+"号大人").fork();//分别将大人线程加入等待队列，还未开始执行 target 里面的参数，只有当唤醒的时候才。。
        //bg.initializeAdult();
        }

    for (int i = 0; i < children; i++)
        {new KThread(new Runnable() {
            public void run() {
                ChildItinerary();
            }
        }).setName((i+1)+"号小孩").fork();
        //bg.initializeChild();
        }
    children_number_O = children;
    adult_number_O = adults;
    children_number_M = 0;
    adult_number_M = 0;
    lock = new Lock();
    children_condition_O = new Condition2(lock);
    children_condition_M = new Condition2(lock);
    adult_condition_O = new Condition2(lock);
    is_pilot = true;
    is_adult_go = false;
    finished = false;
    boat_in_O = true;
     }

     static void AdultItinerary()
     {
    bg.initializeAdult(); //Required for autograder interface. Must be the first thing called.
    lock.acquire();
    if (!(is_adult_go && boat_in_O)) {//只要有一个条件不满足就不能走

    System.out.println(KThread.currentThread().getName()+"sleep ~");
        adult_condition_O.sleep();
    }
    System.out.println(KThread.currentThread().getName()+"被船载到 M 岛");
```

```
    adult_number_O--;
    adult_number_M++;
    is_adult_go = false;// 这一次是成人走下一次就必须是孩子走
    boat_in_O = false;

    children_condition_M.wake();//唤醒一个在 Molokai 的孩子进程将船
开回来

    lock.release();
        }

     static void ChildItinerary()
    {
    bg.initializeChild(); //Required for autograder interface.
Must be the first thing called.
    boolean is_on_O = true;//每个小孩的状态
    lock.acquire();
    while (!finished) {//没有结束
       //考虑 O 岛上的小孩
       if (is_on_O) {
          if (!boat_in_O || is_adult_go)
             {  children_condition_O.sleep();   }//小孩不能走
          if(is_pilot)//如果是开船者
          {
           System.out.print(KThread.currentThread().getName()+"
上船发动船，等待乘客中。。。。");
           is_on_O=false;//要往 M 上划过去
           children_number_O--;
           children_number_M++;
           is_pilot = false;
           children_condition_O.wake();
           //继续唤醒 O 上的一个小孩，唤醒后去往 M，M 上多两个小孩，一
个下船 sleep，一个乘客开回去。
           children_condition_M.sleep();  //实际上不是同时过去，一
前一后
          }
          else//乘客
          {
             if (adult_number_O == 0 && children_number_O == 1)
                finished = true;
             if(adult_number_O != 0)
                is_adult_go=true;

System.out.println(KThread.currentThread().getName()+"被船载着
```

```
向 M 过去了。    ");
            is_on_O=false;
            boat_in_O = false;
            children_number_O--;
            children_number_M++;
            is_pilot = true;


            if(!finished)//划到 M 后检查是否结束了。未结束则唤醒 M 的一
个小孩划回去。
            {
               // System.out.println("要唤醒 M 岛上的:
"+KThread.currentThread().getName());
               children_condition_M.wake();//加入就绪对列
            }

            children_condition_M.sleep();

         }
      }
      //考虑 M 上的小孩
      else {


   System.out.println(KThread.currentThread().getName()+"行驶回
去到 O");
         is_on_O=true;
         boat_in_O = true;
         children_number_M--;
         children_number_O++;

         if (adult_number_O == 0) {
            is_adult_go = false;
            children_condition_O.wake();
         } else {
            if(is_adult_go)
            adult_condition_O.wake();
            else
            children_condition_O.wake();
         }

         children_condition_O.sleep();//在 O 岛上待命

      }
```

```
}
    }
```

C、测试代码：

测试代码就是直接测试本程序，但要放在 KThread 的 selftest 中，

d、测试结果及分析：

下面是测试结果：

```
 ***Testing Boats with 3 children,4 adults***
An adult as forked.
1号大人sleep~
An adult as forked.
2号大人sleep~
An adult as forked.
3号大人sleep~
An adult as forked.
4号大人sleep~
A child has forked.
1号小孩上船发动船，等待乘客中。。。。A child has forked.
2号小孩被船载着向M过去了。
要唤醒线程的名字： 1号小孩
A child has forked.
1号小孩行驶回去到O
要唤醒线程的名字： 1号大人
1号大人被船载到M岛
要唤醒线程的名字： 2号小孩
2号小孩行驶回去到O
要唤醒线程的名字： 3号小孩
3号小孩上船发动船，等待乘客中。。。。要唤醒线程的名字： 1号小孩
1号小孩被船载着向M过去了。
要唤醒线程的名字： 3号小孩
3号小孩行驶回去到O
要唤醒线程的名字： 2号大人
```

2号大人被船载到M岛
要唤醒线程的名字： 1号小孩
1号小孩行驶回去到O
要唤醒线程的名字： 2号小孩
2号小孩上船发动船，等待乘客中。。。。要唤醒线程的名字： 3号小孩
3号小孩被船载着向M过去了。
要唤醒线程的名字： 2号小孩
2号小孩行驶回去到O
要唤醒线程的名字： 3号大人
3号大人被船载到M岛
要唤醒线程的名字： 3号小孩
3号小孩行驶回去到O
要唤醒线程的名字： 1号小孩
1号小孩上船发动船，等待乘客中。。。。要唤醒线程的名字： 2号小孩
2号小孩被船载着向M过去了。
要唤醒线程的名字： 1号小孩
1号小孩行驶回去到O
要唤醒线程的名字： 4号大人
4号大人被船载到M岛
要唤醒线程的名字： 2号小孩
2号小孩行驶回去到O
要唤醒线程的名字： 3号小孩
3号小孩上船发动船，等待乘客中。。。。要唤醒线程的名字： 1号小孩
1号小孩被船载着向M过去了。
要唤醒线程的名字： 3号小孩
3号小孩行驶回去到O
要唤醒线程的名字： 2号小孩
2号小孩上船发动船，等待乘客中。。。。要唤醒线程的名字： 3号小孩

最后三号小孩被载到 M 岛上去了。

e.收获与感想：

主要算法思想如下：

需要记录的信息：O 岛上大人/小孩的人数、M 岛上大人/小孩的人数、船的位置、船的状态（0 人/1 小孩/两人）。

初始状态：大人和小孩都在 O 岛上，船在 O 岛，船为空

分为大人和小孩两种情况：

A.大人：

1)如果船的位置在 O 岛这边且 M 岛上没有孩子，这意味着如果大人上了船到达了 M 岛的话，除非自己开着船回来否则问题得不到解决，但是大人开船回来是一件没有意义的事情，故在这种情况下大人放弃对船的占用，在 O 岛睡眠等待并且释放锁。

2)如果 M 岛上有孩子，船在 O 岛这边且船上剩余容量可以容下一个大人，那么大人线程获取船并驶向 M 岛，修改这个大人线程以及船的位置信息，在到达之后唤醒在 M 岛上休眠等待的孩子线程并且释放锁。

3)如果船的位置在 O 岛这边且 M 岛上有孩子，但是船上已经没有容得下一个大人的位置了，此时大人线程放弃这次对船的占有在 O 岛继续睡眠。

简言之，判断三个条件，**船的位置、船的状态、M 岛上小孩数**

B.小孩：

1)如果有孩子和船都在 O 岛，如果船上没有人，则作为开船者获取船：

如果此时 O 岛上还有其他孩子，则线程开始等待孩子乘客进程，并且在到达 M 岛后睡眠；如果此时 O 岛上没有其他孩子则自己过去了（不出现这样的情况）。

2)如果孩子线程和船都在 O 岛，且船上已经有一个孩子作为开船者，则该线程作为乘客获取船驶向 M 岛，并且做问题解决检查，如果 O 岛上在自己离开后没有人了的话则问题解决，finished 置为真。如果没有结束，开船者将船驶回 O 岛。

3)如果孩子位置和船都在 O 岛，但船上已经满载，则放弃船。

4)如果孩子线程和船都在 M 岛，则乘船驶回 O 岛唤醒 O 岛上的孩子大人线程进行船的资源争夺。

三、Project2

前期准备

• linux 虚拟机

虚拟机只是作为电脑的一个软件来运行，管理方便，并且采用的是 virtualbox+ubun 镜像，具体安装跳过。

• 交叉编译工具的安装

交叉编译工具用于将我们编写的测试程序 C 程序，编译为 nachos 系统可以执行的 coff 可执行文件，然后使用 nachos 命令运行。

在介绍交叉编译工具的安装和使用之前，先看一下 nachos 命令是否能够运行进入到 nachos 目录，我的是在这个 nachos-java/nachos,然后编译 proj1，使用命令 gmake,。

将压缩包解压到 nachos 所在目录，我的是在 /root/nachos-java/nachos，nachos/test/Makefile 要求，设置 ARCHDIR 环境变量，export ARCHDIR=../mips-x86.linux-xgcc,要切换到 nachos/test 下，然后执行，将编译路径加入到 path 路径中。

一、Task 1：

a. 实验题目、实验要求：

在这个任务中，我们参照 syscall.h 中的代码信息，需要在 userprocess.java 中实现 7 个系统调用，包括

halt,create,read,write,open,close 和 unlink。在实现过程中，需要注意的是我们不需要真正的实现一个文件系统，而只是简单的给用户进程访问文件系统的能力。

b. **问题分析及代码实现：**

第一个系统调用——halt，根据分析可知，要注意的是只能第一个进程可以执行，所以我设置了一个 static 静态变量 maxID，每当创建一个进程时，先将 maxID 赋值给 pid，maxID 会自动增加 1。直接判断 pid 是否为 0 即可判断是否是第一个进程，若是第一个进程则调用 Machine.*halt*()方法。

```
 private int handleHalt() {

       //MOD
       if(pid!=0)
          return -1;
       //MOD

   Machine.halt();

   Lib.assertNotReached("Machine.halt() did not halt machine!");
   return 0;
    }
```

第二个系统调用—create，首先我增加了一个变量

```
  private OpenFile[] openFiles;
```

负责管理所有打开的文件，当打开一个文件时就在这个变量数组里增加一个值。先根据参数读取虚拟内存相应的文件名，再根据这个文件名打开相应的文件，如果文件存在就直接打开，如果文件不存在就先创建,相应的文件会在 openFiles 中注册一下，返回的该文件在 openFiles 中的索引置。

```
  private int handleCreate(int fileNameAddr){
       String fileName = readVirtualMemoryString(fileNameAddr,
256);
       if(fileName==null)
          return -1;
       int fileDes=-1;
       for(int i=0;i<openFiles.length;i++){
          if(openFiles[i]==null){
             fileDes=i;

   openFiles[i]=ThreadedKernel.fileSystem.open(fileName,
true);//打开名为 filename 的文件，如果不存在就创建一个
             /*
              * Atomically open a file, optionally creating it if
it does not already exist.
              * If the file does not already exist and create is
false, returns null.
```

```
             * If the file does not already exist and create is
true, creates the file with zero length.
             * If the file already exists, opens the file without
changing it in any way
             */
            if(openFiles[i]==null)
               fileDes=-1;
            else
               break;
         }
      }
      return fileDes;
   }
```

系统调用——open，与 create 相差无几，只是在 ThreadedKernel.*fileSystem*.open(fileName, false);第二个参数设置的是 false，即如果文件不存在不会去创建：

```
private int handleOpen(int fileNameAddr){
      String fileName = readVirtualMemoryString(fileNameAddr,
256);//限定文件名长度

      if(fileName==null)
         return -1;
      int fileDes=-1;
      for(int i=0;i<openFiles.length;i++){
         if(openFiles[i]==null){
            fileDes=i;

   openFiles[i]=ThreadedKernel.fileSystem.open(fileName,
false);
            if(openFiles[i]==null)
               fileDes=-1;
            break;
         }
      }
      return fileDes;
   }
```

第四、五个系统调用——read、write：将文件中的内容与虚拟内容进行交互（写、读），直接使用了 writeVirtualMemory、readVirtualMemory 方法。而关于这两个方法的具体实现在 task2 中再具体介绍。

```
private int handleRead(int fileDes,int buffer,int size){

   if(fileDes<0||fileDes>=openFiles.length||openFiles[fileDes]
==null)
         return -1;
```

```
        byte[] data = new byte[size];
        int len = openFiles[fileDes].read(data, 0, size);//len:
实际上成功读到的数据
        if(len>=0){
            return writeVirtualMemory(buffer, data, 0, len);//写的
buffer 开始的虚拟内存中
        }
        return -1;
    }

    private int handleWrite(int fileDes,int buffer,int size){

   if(fileDes<0||fileDes>=openFiles.length||openFiles[fileDes]
==null)
            return -1;
        byte[] data = new byte[size];
        int len = readVirtualMemory(buffer, data);
        if(len>=0){
            int writeLen = openFiles[fileDes].write(data, 0, len);
            if(writeLen>=len)
                return writeLen;
        }
        return -1;
    }
```

第六个系统调用——close，就是关闭相应的文件，对文件会进行判断是否存在。并将 openFiles 中删除该文件，实际上没有删除该文件。

```
  private int handleClose(int fileDes){

   if(fileDes<0||fileDes>=openFiles.length||openFiles[fileDes]
==null)
            return -1;
        openFiles[fileDes].close();
        openFiles[fileDes]=null;// 将文件从进程打开的文件表中去除
        return 0;
    }
```

第七个系统调用——unlink，删从系统的文件系统中彻底移除一个文件：

```
private int handleUnlink(int fileNameAddr){
        String fileName = readVirtualMemoryString(fileNameAddr,
256);
        if(fileName==null)
            return -1;
        if(ThreadedKernel.fileSystem.remove(fileName))
            return 0;
```

```
        else
            return -1;
    }
```

测试程序是将 task1~3 一起测试的，在之后一起介绍。

c. 收获与感想：

在需要实现的七个系统调用中，需要注意几个细节：(1)halt 只能被第一个进程也就是根进程执行。(2)任务中，每个打开的文件都必须要分配一个文件描述符，用于标注，文件描述符用整形表示。(3)在 Nachos 系统中，在 machine 包中，我们已经提供了 FileSystem 文件系统，由于用户内核类 UserKernal 继承与类 ThreadedKernel，所以我们可以通过 ThreadedKernel.fileSystem 访问这个文件系统。

在 machine.FileSystem 中，这个给定的文件系统允许用户创建，打开和删除文件，并且当打开文件的时候，文件不存在，那么会返回 false，并且会创建此文件，长度为空，但如果文件存在，则返回 true，并且直接打开这个文件，不需要修改。

在对打开文件进行操作时，需要用到打开文件表 OpenFile[],相当于一个存储众多打开文件的列表。

二、Task 2：

a. 实验题目、实验要求：

Nachos 是不支持动态内存分配的，因此在装入程序时，为每个进程分配固定的不可改变的物理内存。然后在进程结束时回收这些内存。

首先我们应该分析 coff 程序的程序结构，一个 coff 程序由许多段组成，每个段是一个 machine.CoffSection 类型的对象，每个段它又由若干个页组成。

我们需要关注每个进程逻辑页(虚拟页)到物理页的映射方案。实现实现一个简单的映射机制，使虚拟内存能够与物理内存一一对应。

b. 问题分析及代码实现：

• **load 方法：**将运行的文件加载到虚拟内存中，首先打开文件，一个 coff 文件是由多个 section 组成，而一个 section 又是由多个页组成还要确保文件在虚拟内存页中从 0 开始并且连续。之后还要多一也页是存放所有的参数的，要把所有的参数也写到相应的虚拟页中。

```
 private boolean load(String name, String[] args) {
    Lib.debug(dbgProcess, "UserProcess.load(\"" + name + "\")");

    OpenFile executable = ThreadedKernel.fileSystem.open(name,
false);
    if (executable == null) {
```

```
            Lib.debug(dbgProcess, "\topen failed");
            return false;
        }

        try {
            // 加载通用文件并导入虚拟内存
            coff = new Coff(executable);
        }
        catch (EOFException e) {
            executable.close();
            Lib.debug(dbgProcess, "\tcoff load failed");
            return false;
        }

        // make sure the sections are contiguous and start at page 0
        numPages = 0;
        for (int s=0; s<coff.getNumSections(); s++) {
            CoffSection section = coff.getSection(s);
            if (section.getFirstVPN() != numPages) {
            coff.close();
            Lib.debug(dbgProcess, "\tfragmented executable");
            return false;
            }
            numPages += section.getLength();
        }

        // make sure the argv array will fit in one page
        byte[][] argv = new byte[args.length][];
        int argsSize = 0;
        for (int i=0; i<args.length; i++) {
            argv[i] = args[i].getBytes();
            // 4 bytes for argv[] pointer; then string plus one for null
byte
            argsSize += 4 + argv[i].length + 1;
        }
        if (argsSize > pageSize) {
            coff.close();
            Lib.debug(dbgProcess, "\targuments too long");
            return false;
        }

        // program counter initially points at the program entry point
        initialPC = coff.getEntryPoint();
```

```
    // next comes the stack; stack pointer initially points to top
of it
    numPages += stackPages;
    initialSP = numPages*pageSize;

    // and finally reserve 1 page for arguments
    numPages++;

    if (!loadSections())
        return false;

    // store arguments in last page
    int entryOffset = (numPages-1)*pageSize;
    int stringOffset = entryOffset + args.length*4;

    this.argc = args.length;
    this.argv = entryOffset;

    for (int i=0; i<argv.length; i++) {
        byte[] stringOffsetBytes =
Lib.bytesFromInt(stringOffset);

Lib.assertTrue(writeVirtualMemory(entryOffset,stringOffsetByte
s) == 4);
        entryOffset += 4;
        Lib.assertTrue(writeVirtualMemory(stringOffset, argv[i])
==
              argv[i].length);
        stringOffset += argv[i].length;
        Lib.assertTrue(writeVirtualMemory(stringOffset,new byte[]
{ 0 }) == 1);
        stringOffset += 1;
    }

    return true;
    }
```

• **loadSection**：这一方法是具体实现将每个段加载到虚拟页中的，根据 load 中对 coff 文件的分析得到总共需要的页数，创建页表，获取相应的页，把 section 依次装入。

```
 protected boolean loadSections() {

       UserKernel.allocLock.acquire();
    if (numPages > UserKernel.physicalPages.size()) {//页数大于真
正的物理页数
```

```
            coff.close();
            Lib.debug(dbgProcess, "\tinsufficient physical memory");


            UserKernel.allocLock.release();
            return false;
        }

        //创建页表
        pageTable = new TranslationEntry[numPages];
        for(int i = 0;i<numPages;i++){
            int nextPage = UserKernel.physicalPages.poll();
            pageTable[i]=new TranslationEntry(i, nextPage, true, false,
false, false);//虚拟和物理转换
        }
        UserKernel.allocLock.release();

        // load sections
        for (int s=0; s<coff.getNumSections(); s++) {
            CoffSection section = coff.getSection(s);

            Lib.debug(dbgProcess, "\tinitializing " +
section.getName()
                    + " section (" + section.getLength() + " pages)");

            for (int i=0; i<section.getLength(); i++) {
            int vpn = section.getFirstVPN()+i;

            // for now, just assume virtual addresses=physical addresses
            //section.loadPage(i, vpn);
            pageTable[vpn].readOnly=section.isReadOnly();
            section.loadPage(i, pageTable[vpn].ppn);
            }
        }
        return true;
            }
```

unloadSections 从物理页中删除相应的段，也就是删除段所拥有的页们。

```
    protected void unloadSections() {
            UserKernel.allocLock.acquire();
            for(int i=0;i<numPages;i++){
                UserKernel.physicalPages.offer(pageTable[i].ppn);
            }
            UserKernel.allocLock.release();
            }
```

• **writeVirtualMemory** 将内容写入相应的虚拟内存中去。

```
public int writeVirtualMemory(int vaddr, byte[] data, int offset,
                int length) {
    Lib.assertTrue(offset >= 0 && length >= 0 && offset+length <=
data.length);

    byte[] memory = Machine.processor().getMemory();

    // for now, just assume that virtual addresses equal physical
addresses

    int amount = Math.min(Math.min(length,
numPages*pageSize-vaddr), data.length-offset);
    int trans = 0;
    do{
        int vpn = Processor.pageFromAddress(vaddr + trans);
        if (vpn < 0 || vpn >= numPages)
            return 0;
        int ppn = pageTable[vpn].ppn;
        int pOffset = Processor.offsetFromAddress(vaddr + trans);
        int paddr = Processor.makeAddress(ppn, pOffset);
        int len = Math.min(amount-trans, pageSize-pOffset);
        System.arraycopy(data, offset+trans, memory, paddr, len);
        trans+=len;
    }while(trans<amount);

    return trans;
    //MOD
     }
```

• **readVirtualMemory** 将内容从相应的虚拟内存中读出放入 **data** 中。

```
public int readVirtualMemory(int vaddr, byte[] data, int offset,
                int length) {
    Lib.assertTrue(offset >= 0 && length >= 0 && offset+length <=
data.length);

    byte[] memory = Machine.processor().getMemory();

    // for now, just assume that virtual addresses equal physical
addresses

    //读有限

    int amount = Math.min(Math.min(length,
numPages*pageSize-vaddr), data.length-offset);
```

```
int trans = 0;
do{
    int vpn = Processor.pageFromAddress(vaddr + trans);
    if (vpn < 0 || vpn >= numPages)
        return 0;
    int ppn = pageTable[vpn].ppn;
    int pOffset = Processor.offsetFromAddress(vaddr + trans);
    int paddr = Processor.makeAddress(ppn, pOffset);
    int len = Math.min(amount-trans, pageSize-pOffset);
    System.arraycopy(memory, paddr, data, offset+trans, len);
    trans+=len;
}while(trans<amount);

return trans;
 }
```

实验测试结果在 task3 后。

c. **收获与感想：**

• 要求我们实现 loadSection 和 unloadSection 方法，这要求我们对 Coff 结构有所了解。

• 明确修改部分：readVirtualMemory()、writeVirtualMemory()、loadSections()、unloadSections()。

• 对于 readVirtualMemory：先将虚拟地址转换为物理地址，再从物理地址读取数据到 buffer 中。

• 对于 writeVirtualMemory：先将虚拟地址转换为物理地址，然后再把 buffer 中的数据写入对应的物理地址中。

• loadSections：在进行 Coff 文件导入之前首先要为进程分配页表，在本次 nachos 设计中，当且仅当进程需要装入程序执行时才被分配以页表，该页表用于在虚拟内存和物理内存之间进行地址转换，然后再把 coff 文件中的程序导入到对应页中。

• unloadSections：在进程结束时调用释放掉占有的内存。

三、Task 3：

a. **实验题目、实验要求：**

参照 syscall.h 中的代码信息，需要实现 exec，join 和 exit 三个系统调用。根据之前的几个任务，我们分析出着三个系统调用与进程调度有关，其中 exec 是启动一个新的进程，join 与线程中的 join 调用类似，等待某进程结束后，继续当前进程，而 exit 是退出当前

进程。

b. 问题分析及代码实现：

• handleExec　在创建一个新的进程时，会将其的父进程设置为当前进程，在其父进程的子进程中加入新创建的这个进程。

```
private int handleExec(int nameAddr, int argc, int argvAddr){
      String fileName = readVirtualMemoryString(nameAddr,
256);// 从内存中读出这个子进程使用的文件的名字
      if(fileName==null||!fileName.endsWith(".coff")||argc<0)
         return -1;
      String[] argv = new String[argc];//设置子进程的参数表
      for(int i=0;i<argc;i++){
         byte[] strAddr = new byte[4];
         readVirtualMemory(argvAddr+i*4, strAddr);//把虚拟地址转
化为物理地址

   argv[i]=readVirtualMemoryString(Lib.bytesToInt(strAddr, 0),
256);
      }
      UserProcess process=UserProcess.newUserProcess();
      if(!process.execute(fileName, argv)){
         return -1;
      }
      children.put(process.pid, process);
      process.parent=this;
      return process.pid;
   }
```

•handleJoin

与进程退出进行配合，当调用 join 方法的时候，先判断调用这个方法的进程是否是当前进程的父进程，如果是挂起当前进程，如果不是，返回-1.

```
 private int handleJoin(int pid, int statusAddr){
      UserProcess process = children.get(pid);
      if(process==null){
         return -1;
      }
      joinSem=new Semaphore(0);
      joinSem.P();
      byte[] ret = Lib.bytesFromInt(process.status);
      if(writeVirtualMemory(statusAddr,
ret)==4&&process.isExitNormally){
         return 1;
      }
      return 0;
```

```
    }
```

- **handleExit**

在每个进程执行完毕退出的时候会自动调用这个方法，此处对本线程的父线程进行判断，如果存在当前进程的父线程，就将其唤醒，同时其父线程所拥有的子线程 map 数据结构中删去当前进程。最后判断当前进程是否是最后一个进程，如果是，则整个操作系统终止。在用户进程中设置了一个 runningProc 变量用来保存所有的用户线程数，每当执行完一个线程，该变量数减一。

```
private int handleExit(int status){
        coff.close();
        for(int i=0;i<openFiles.length;i++){
            if(openFiles[i]!=null){
                openFiles[i].close();
                openFiles[i]=null;
            }
        }
        this.status=status;
        isExitNormally=true;
        if(parent!=null){
            parent.joinSem.V();
            parent.children.remove(pid);
            parent=null;
        }
        unloadSections();
        runningProc--;
        //最后一个进程
        if(runningProc<=0){
            Kernel.kernel.terminate();
        }
        UThread.finish();//用户进程结束
        return 0;
            }
```

c. 收获与感想：

· 这次任务中的要求需要添加一些变量才能够完成。Pid 是在之前就已经实现了的东西。对于子进程，每个进程都可能有一定数目的子进程，所以利用一个哈希集来保存一个进程的所有子进程的 pid，该哈希集被命名为 children。对于子进程是否执行成功，利用 processStatus 来描述，进程退出是否成功利用 exitSuccess 来描述。同时还设置一个 parent 变量，表示当前进程的父进程。

• 对于 handleExit：进程结束时调用该函数，操作包括：关闭所有打开的文件，释放掉所有占有的资源。尝试唤醒其他正在等待的进程，检查自己是不是最后一个结束的进程，如果是最后一个结束的进程则需要直接调用 kernel.kernel.terminate 函数关闭 nachos 系统。

• 对于 handleExec：从内存中读出文件名以及参数，创建用户进程运行相关程序。

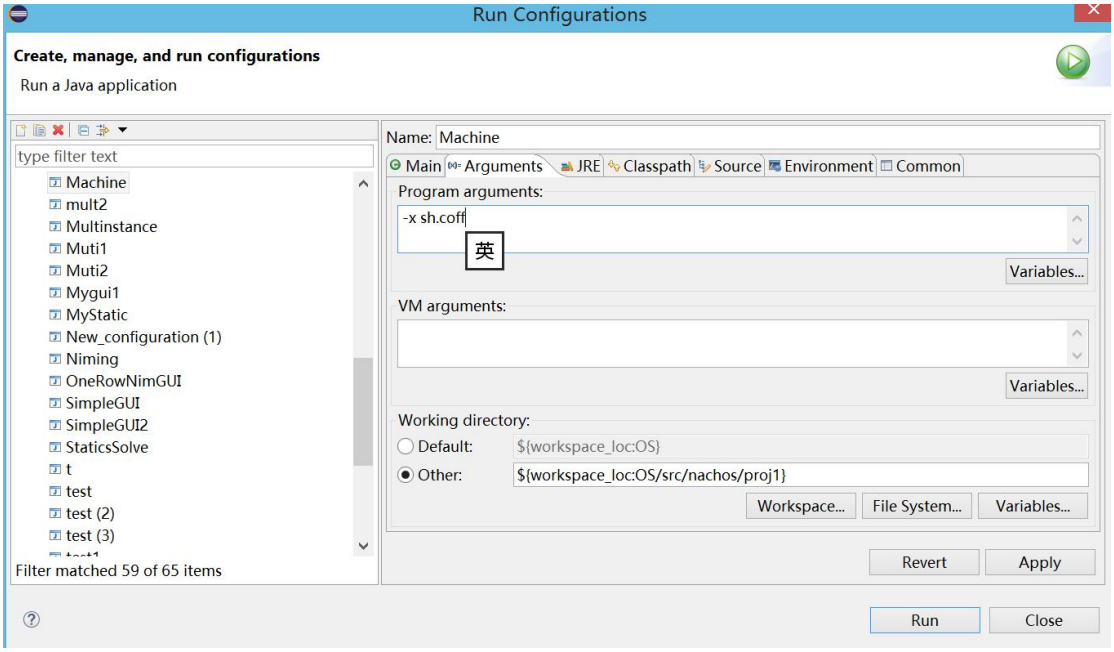
• 对于 handleJoin：进程只能 Join 自己的子进程，被 Join 的子进程陷入休眠，在结束任务后唤醒子进程将子进程的状态存入父进程中。

d、前三个的测试代码、测试结果及分析：

前三个实验都是加载相应的 coff 文件执行，根据题目要求是自己编写.c 文件，用交叉编译器编译成相应的.coff 文件，之后运行有两种方法，一种是在 conf 文件中更改设置：

```
Kernel.shellProgram = sh.coff #sh.coff
```

另一种是在 run configration 中加上-x 要执行的文件名.coff

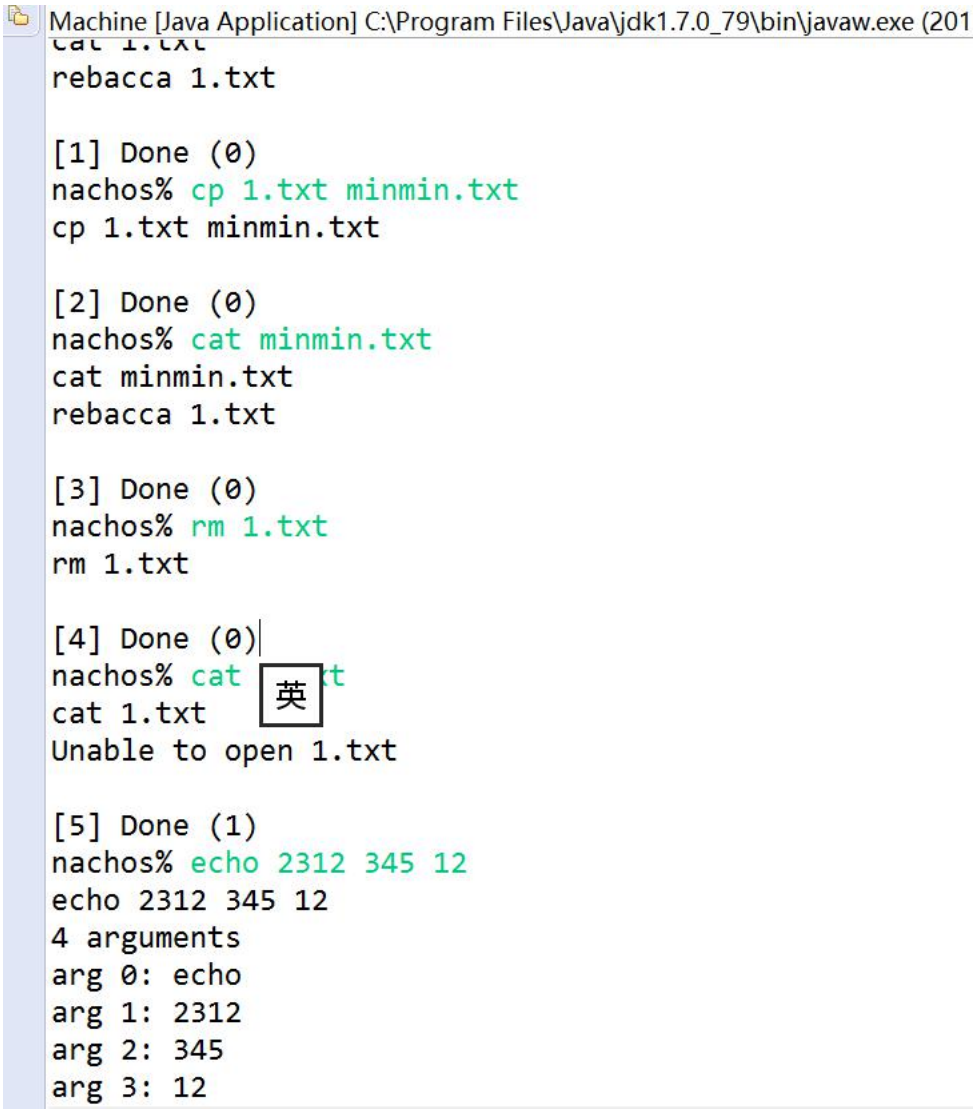


在给定的网址上下了相应的交叉编译器，在 ubuntu 底下编译相应的.c 文件。但在一开始下载的 nachos 中给定了相应的测试 coff 文件，其中 cat 文件是打开某个文件并输出其中的内容，rm 文件是删除参数传入的文件，cp 是将一个文件打开并复制到另一个文件中去。而存在一个 sh 文件，它可以根据用户在控制台上输入的文件名，进而执行上述所介绍的具体文件，所以我直接设置加载 sh.coff 文件。仔细分析 coff 文件，其中不仅设计到了读文件、删除文件等，还涉及到 join、创建 exec、finish（进程执行完自动调用）等方法：

```
int pid, background, status;

 char args[BUFFERSIZE], prog[BUFFERSIZE];
 char *argv[MAXARGS];

 int argc = tokenizeCommand(line, MAXARGS, argv, args);
 if (argc <= 0)
return;

 if (argc > 0 && strcmp(argv[argc-1], "&") == 0) {
argc--;
background = 1;
 }
 else {
background = 0;
 }

 if (argc > 0) {
if (strcmp(argv[0], "exit")==0) {
    if (argc == 1) {
   exit(0);
    }
    else if (argc == 2) {
   exit(atoi(argv[1]));
    }
    else {
   printf("exit: Expression Syntax.\n");
   return;
    }
}
else if (strcmp(argv[0], "halt")==0) {
    if (argc == 1) {
   halt();
   printf("Not the root process!\n");
    }
    else {
   printf("halt: Expression Syntax.\n");
```

```
        }
        return;
    }
    else if (strcmp(argv[0], "join")==0) {
        if (argc == 2) {
        pid = atoi(argv[1]);
        }
        else {
        printf("join: Expression Syntax.\n");
        return;
        }
    }
    else {
        strcpy(prog, argv[0]);
        strcat(prog, ".coff");

        pid = exec(prog, argc, argv);
        if (pid == -1) {
        printf("%s: exec failed.\n", argv[0]);
        return;
        }
    }

    if (!background) {
        switch (join(pid, &status)) {
        case -1:
        printf("join: Invalid process ID.\n");
        break;
        case 0:
        printf("\n[%d] Unhandled exception\n", pid);
        break;
        case 1:
        printf("\n[%d] Done (%d)\n", pid, status);
        break;
        }
```

可以看到，当输入正确的参数，background 置为 1，然后调用 join 方法，就是去执行一个新的线程（新的通用文件）。



四、Task 4：

4.1 实验要求

实施彩票调度程序（将其放置在线程/ LotteryScheduler.java 中）。 注意这个类扩展了 PriorityScheduler，你应该能够重用那个类的大部分功能；彩票调度器不应该有大量的附加代码。 唯一的主要区别是用于从队列中选择线程的机制：持有彩票，而不是仅选择具有最高优先级的线程。

4.2 实验分析

- 明确 LotteryScheduler 与 PriorityScheduler 之间的异同点，明确代码需要做的改动。这里主要是 getEffectivePriority 和 pickNextThread 之间存在不同需要小作修改。
- 最高优先级 LotteryScheduler 是 Integer 类型的最大值。
- 优先级转置获取的是所有等待队列中线程优先级的加和
- 在选择下一个执行的线程的过程中，调度器随机抽取一张彩票，彩票数量多的线程有更高可能性得到执行。

4.3 具体实现基本过程介绍

与 project 1 很多方法都是类似的，主要需要修改的地方在于增加了一个表示彩票数的变量，在 runNext 函数处不再是需要选择优先级最大的线程，而是先得到所有的彩排数，调用 random 函数在该数的范围内进行一个随机选择，根据概率的知识，某个线程的彩票数越大，随机选择的数落在该数段的概率较大。

此处是：

```
public KThread nextThread() {
        // TODO Auto-generated method stub
        Lib.assertTrue(Machine.interrupt().disabled());
        int lottery=0;
        KThread thread=null;
        for(int i=0;i<link.size();i++)
        {
            lottery+=link.get(i).getEffectivePriority();

        }

        int run=Lib.random(lottery+1);//随机选取运行线程的彩票数
        int rank=0;
        for(int i=0;i<link.size();i++)
        {
            rank+=link.get(i).getEffectivePriority();
            if(rank>=run)
            {
                thread=link.get(i).thread;
                break;
            }
        }
        if(thread!=null)
        {
            link.remove(thread.schedulingState);
            return thread;
        }
        else
        {
            return null;
        }
        }
```

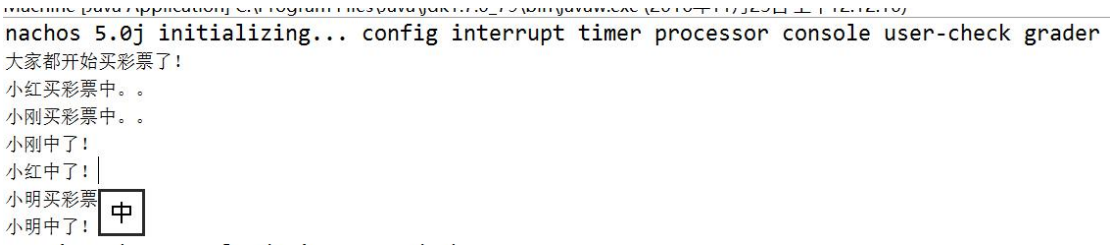
测试代码：

```
public static void selfTest_LotteryScheduler()
    {
       Lib.debug(dbgThread, "Enter KThread.selfTest_join");
```

```
System.out.println("大家都开始买彩票了！ ");
final String[] things=new String[]{"买彩票中。。","中了！ "};
KThread thread1=new KThread(new Runnable()
{
    public void run()
    {
        for(int i=0;i<2;i++)
        {
            currentThread.yield();
            System.out.println("小明"+things[i]);
        }
    }
});
KThread thread2=new KThread(new Runnable()
{
    public void run()
    {
        for(int i=0;i<2;i++)
        {
            currentThread.yield();
            System.out.println("小红"+things[i]);
        }
    }
});
KThread thread3=new KThread(new Runnable()
{
    public void run()
    {

        for(int i=0;i<2;i++)
        {
            currentThread.yield();
            System.out.println("小刚"+things[i]);
        }
    }
});
thread1.setName("thread1");
thread2.setName("thread2");
thread3.setName("thread3");
boolean intStatus=Machine.interrupt().disable();
ThreadedKernel.scheduler.setPriority(thread1,1);
ThreadedKernel.scheduler.setPriority(thread2,5);
ThreadedKernel.scheduler.setPriority(thread3,7);
Machine.interrupt().setStatus(intStatus);
```

```
        thread1.fork();
        thread2.fork();
        thread3.fork();
        }
```

测试结果：



可以看出给三个线程赋予的彩票数是递增的，但实际上执行的时候并不是每次都是彩票数大的最先执行，可以看出彩票调度的顺序不是人为设置的，执行顺序也存在很多的不可确定性。

四、Nachos 实验总结（主要是 Project2 的实验总结）：

做完了两个 project，总算是松了口气，回头来看，整个 nachos 系统并不是特别的复杂。在做实验的过程中，深深的感受到 api 和 debug 的用处，还有注释，读源代码的时候英文注释极大的帮助了我。

在 project 1 的时候，一开始做了大量的了解工作，但是实际操作时遇到了很多问题，印象比较深刻的是 project 1 的 task 5，是在整个 project 1 中遇到最难的问题，，因为不仅涉及到优先级的调度，还设计到优先级反转问题，要结合 lock 类一起，因为要判断拥有锁的线程，要对其进行优先级反转。Task 6 也并不简单，主要是很麻烦，有多种情况都需要考虑。

在 project 2 中，在编译遇到了问题，因为自己写出的 c 文件后想要用编译器编译，但是那个交叉编译器在 ubuntu 底下编译的时候，首先需要配置环境变量。官方给的下载界面上做了些许的介绍但是并不具体，所以遇到了很多问题，最后问了一下同学顺利的解决了。

此外虚拟内存和物理内存之间的转换也是一个重要的问题，需要对页表深入理解，进程拥有的页表是虚拟内存与物理内存之间的映射关系的列表。进程占有

一定的内存，这体现在其页表的数量上，而页表的每个下标对应的是一块内存，在本 nachos 系统中 pageSize 设置为 8bytes，通过某个下标对应的页的 ppn 属性就知道其对应的物理内存页号，由此完成了由虚拟内存到物理内存的对应关系。在对虚拟内存读写方法中虚拟内存与物理内存之间的转换关系比较复杂，需要保持头脑清醒仔细安排好每个步骤进行编写。

对于 task 2 的多道程序，简单的总结为：在导入 coff 之前应该创建一个页表，进行物理地址和逻辑地址的关联，然后把程序的每一块按照顺序对应于物理地址导入到内存中。 读内存时，要先利用页表将逻辑地址转换为物理地址然后再将内存数据复制到数组中。写内存时，要先利用页表将逻辑地址转换为物理地址然后再将内存数据复制到数组中。

通过此次的操作系统实验，我对上学期学的操作系统的纯理论知识有了进一步的了解，动手操作使我更好的学习到了关于操作系统的很多知识：页表、虚拟内存、子进程父进程、条件变量等很多知识。