

**操作系统课程设计报告**

班级：2012级软件工程8班

团队成员：

杨环

张俊

吴佩瑶

王飞

王梅瑞

目录

[目录 2](#_Toc27407)

[一、 前期工作 3](#_Toc18752)

[1.1平台搭建 3](#_Toc30785)

[二、 代码分工 3](#_Toc10162)

[三、设计及实现 4](#_Toc22647)

[3.1 Task1.1 KThread.join() 4](#_Toc27533)

[3.1.1 要求分析 4](#_Toc23052)

[3.1.2 设计方案 4](#_Toc3964)

[3.1.3 实现代码 4](#_Toc17360)

[3.1.4 测试代码及结果 5](#_Toc25244)

[3.2 Task1.2 condition2类 7](#_Toc26064)

[3.2.1 要求分析 7](#_Toc17564)

[3.2.2 设计方案 8](#_Toc6512)

[3.2.3 实现代码 8](#_Toc2043)

[3.2.4 测试代码及结果 8](#_Toc29493)

[3.3 Task1.3 alram类 11](#_Toc21286)

[3.3.1 要求分析 11](#_Toc15340)

[3.3.2 设计方案 11](#_Toc24442)

[3.3.3 实现代码 12](#_Toc13341)

[3.3.4 测试代码及结果 12](#_Toc11648)

[3.4 Task1.4 communicator类 14](#_Toc31710)

[3.4.1要求分析 14](#_Toc16349)

[3.4.2 设计方案 14](#_Toc2698)

[3.4.3 实现代码 14](#_Toc15036)

[3.4.4 测试代码及结果 15](#_Toc15590)

[3.5 Task1.5 priority scheduler类 17](#_Toc29651)

[3.5.1 要求分析 17](#_Toc30140)

[3.5.2 设计方案 17](#_Toc7973)

[3.5.3 实现代码 18](#_Toc27631)

[3.5.4 测试代码及结果 18](#_Toc26412)

[3.6 Task1.6 boat类 23](#_Toc23193)

[3.6.1 要求分析 23](#_Toc29464)

[3.6.2 设计方案 23](#_Toc10932)

[3.6.3 实现代码 23](#_Toc13382)

[3.6.4 测试代码及结果 30](#_Toc11090)

[3.7 Task2.1 系统调用creat, open, read, write, close, unlink 30](#_Toc23574)

[3.7.1 要求分析 30](#_Toc21634)

[3.7.2 设计方案 31](#_Toc9672)

[3.7.3 实现代码 32](#_Toc12868)

[3.7.4 测试代码及结果 34](#_Toc15382)

[3.8 Task2.2 多进程内存分配及访问 35](#_Toc16447)

[3.8.1 要求分析 35](#_Toc147)

[3.8.2 设计方案 35](#_Toc1052)

[3.8.3 实现代码 36](#_Toc27424)

[3.8.4 测试代码及结果 39](#_Toc25515)

[3.9 Task2.3 系统调用exec, join, exit 41](#_Toc10742)

[3.9.1 要求分析 41](#_Toc23871)

[3.9.2 设计方案 41](#_Toc12055)

[3.9.3 实现代码 42](#_Toc15334)

[3.9.4 测试代码及结果 43](#_Toc32289)

[3.10 Task2.4 Lottery Schedule类 44](#_Toc28265)

[3.10.1 要求分析 44](#_Toc26071)

[3.10.2 设计方案 45](#_Toc7987)

[3.10.3 实现代码 45](#_Toc27357)

[3.10.4 测试代码及结果 45](#_Toc23699)

[总结 48](#_Toc5504)

# 前期工作

## 1.1平台搭建

Nachos For Java

phrase1部分：IDE环境可采用Eclipse。

Phase 2及以后阶段：需要把C程序编译成MIPS二进制文件COFF，需要MIPS的C编译器 mips-x86.linux-xgcc(ubuntu12.04平台下)。

# 代码分工

# 三、设计及实现

## 3.1 Task1.1 KThread.join()

### 3.1.1 要求分析

Join()方法的含义：当前线程a在运行，执行b.join(),则a阻塞，直到线程b结束，a继续执行。

Join函数的作用即为等待某线程运行完毕。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。注意在一个KThread对象上只能调用一次join，且当前线程不能对自身调用join。

### 3.1.2 设计方案

为每个线程创建一个线程队列joinqueue，joinqueue由对本线程调用join方法的其他线程对象构成，可“捐赠”优先级 ，以及布尔形变量joined，判断本线程是否被其他线程调用过join方法，以及一个判断线程状态方法IsAlive()。

假设当前线程A调用就绪线程B的join函数. 在join函数中线程A被添加到线程B的线程队列joinqueue中，将线程A“休眠”, 线程B得到执行，在线程B结束时(此时线程B成为当前线程, 发生在KThead.finish函数中) 选择B的joinqueue的nextThread()执行, A唤醒并继续.

### 3.1.3 实现代码

在KThread.java 中:

**private** ThreadQueue joinQueue = **null**;

**private** **boolean** Joined = **false**;

**public** **boolean** IsAlive(){

**if**(**this**.status==*statusNew*||status==*statusReady*||

status==*statusRunning*||status==*statusBlocked*)

**return** **true**;

**else** **return** **false**;

}

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

//

**if** (!**this**.IsAlive()) **return**;

**while** (IsAlive()) {

//

**this**.joinQueue.acquire(**this**);

//

}

**this**.joinQueue.waitForAccess(KThread.*currentThread*());

KThread.*sleep*();

//

}

}

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

///

**if** (*currentThread*.Joined) {

*currentThread*.joinQueue.nextThread().ready();

}

///

}

### 3.1.4 测试代码及结果

创建AThread和Bthread两个线程，AThread循环打印“AThread循环第..次”语句，Bthread开始打印“B\_thread 就绪”语句，中间执行AThread。Join（）方法，最后打印“B\_thread执行结束”语句。

测试代码:

**package** nachos.threads;

**import** nachos.machine.\*;

**public** **class** KThreadTest {

**public** KThreadTest() {

}

**public** **static** **void** simpleJoinTest() {

KThread A\_thread = **new** KThread(**new** KThreadTest.A\_thread(5));

KThread B\_thread = **new** KThread(**new** KThreadTest.B\_thread(A\_thread));

B\_thread.fork();

B\_thread.join();

}

**public** **static** **class** B\_thread **implements** Runnable {

B\_thread(KThread joinee) {

**this**.joinee = joinee;

}

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

System.*out*.println("B\_thread 就绪");

System.*out*.println("Forking and joining A\_thread...");

**this**.joinee.fork();

**this**.joinee.join();

System.*out*.println("B\_thread 执行结束");

}

**private** KThread joinee;

}

**public** **static** **class** A\_thread **implements** Runnable {

A\_thread(**int** num) {

**this**.num = num;

}

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

System.*out*.println("A\_thread 就绪");

System.*out*.println("A\_thread开始执行");

// This should just kill some cycles

**for** (**int** i = 0; i < **this**.num; ++i) {

System.*out*.println("A\_thread 循环 第" + i + " 次");

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

}

System.*out*.println("A\_thread 执行结束");

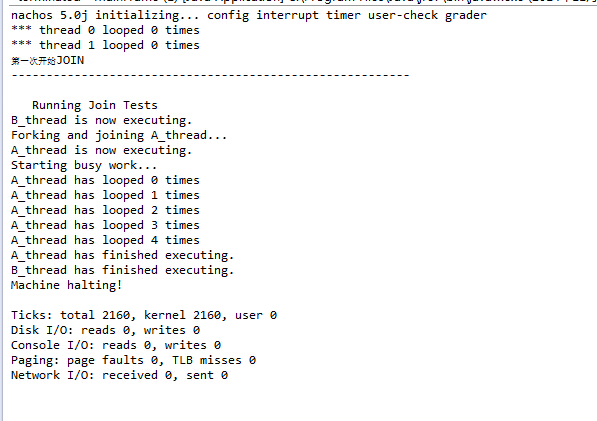
}

**private** **int** num;

}

}

测试结果：



## 3.2 Task1.2 condition2类

### 3.2.1 要求分析

threads.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threads.Condition实现, 它是用来保证同步的工具. 每一个条件变量拥有一个锁变量 (该锁变量亦可被执行acquire和release操作, 多个条件变量可共享同一个锁变量). 当处于临界区内的拥有某锁Lock的当前线程对与锁Lock联系的条件变量执行sleep操作时, 该线程失去锁L并被挂起. 下一个等待锁L的线程获得锁L (这个过程由调度程序完成) 并进入临界区. 当拥有锁L的临界区内的当前线程对与锁L联系的条件变量执行wake操作时 (通常调用wake之后紧接着就是Lock.release), 等待在该条件变量上的至多一个被挂起的线程 (由调用sleep引起) 被重新唤醒并设置为就绪状态. 若执行wakeall操作, 则等待在该条件变量上的所有被挂起的线程都被唤醒. threads.condition已经实现了一个条件变量 (采用信号量实现), 题目要求用屏蔽/禁止中断的方法再实现一下条件变量 (写在threads.condition2中)

### 3.2.2 设计方案

为每个条件变量添加一个锁conditionLock和一个Kthread对象组成的等待队列waitqueue，condition.sleep 将调用sleep（）的线程加入到等待队列，释放锁，并阻塞，以便其他线程唤醒它，一旦’唤醒’立即要求锁，并在sleep函数开始/结尾处屏蔽/允许中断以保证原子性;condition.wake中从等待队列中取出线程进行ready()实现唤醒,

(同样要在wake函数开始/结尾处屏蔽/允许中断), wakeall函数的实现依赖于wake. 只需不断地wake 直到队列为空为止.

### 3.2.3 实现代码

见threads/condition2.java

**private** Lock conditionLock;

**private** LinkedList<KThread> waitQueue;

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

**//**

waitQueue.add(KThread.*currentThread*());

conditionLock.release();

KThread.*sleep*();

conditionLock.acquire();

//

}

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

**//**

waitQueue.remove().ready();

//

}

### 3.2.4 测试代码及结果

创建共有条件变量c2test，一个临界资源count和三个线程，thread1，thread2，thread3，初始化三个线程后thread1，thread2调用c2test.sleep()后开始“睡眠”，thread3调用c2test.wakeAll()唤醒所有线程，thread1，thead2开始交替访问临界资源count并更改其“余额”数值。

测试代码:

**package** nachos.threads;

**import** nachos.machine.\*;

**public** **class** Condition2Test {

Lock condlock = **new** Lock();

Condition2 c2test = **new** Condition2(condlock);

**public** Condition2Test(){

}

**public** **void** simpleCondition2Test(){

System.*out*.println("\n \*\*\*\*Condition2Test is now executing.\*\*\*\*");

System.*out*.println("初始化账户余额为10000");

**final** MyCount myCount = **new** MyCount("0000001", 10000); //创建并发访问的账户

KThread thread1 = **new** KThread(**new** Runnable() {

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

**new** SaveThread("张三", myCount, 2000);

System.*out*.println("张三 goes to sleep");

condlock.acquire();

c2test.sleep();

System.*out*.println("张三 reacquires lock when woken.");

condlock.release();

System.*out*.println("张三 is awake!!!");

myCount.save(2000," 张三");

}

});

KThread thread2 = **new** KThread(**new** Runnable() {

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

**new** SaveThread("李四", myCount, 2000);

System.*out*.println("李四 goes to sleep");

condlock.acquire();

c2test.sleep();

System.*out*.println("李四 reacquires lock when woken.");

condlock.release();

System.*out*.println("李四 is awake!!!");

myCount.save(2000," 李四");

}

});

KThread thread3 = **new** KThread(**new** Runnable() {

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

System.*out*.println("thread 3 Waking up the thread");

condlock.acquire();

c2test.wakeAll();

condlock.release();

System.*out*.println("张三 and 李四 woke up by wakeAll");

}

});

thread1.fork();

thread2.fork();

thread3.fork();

thread1.join();

thread2.join();

thread3.join();

System.*out*.println("\*\*\*\*Condition2Test finished.\*\*\*\*\n");

}

}

**class** SaveThread **extends** KThread {

**private** String name; //操作人

**private** MyCount myCount; //账户

**private** **int** x; //存款金额

SaveThread(String name, MyCount myCount, **int** x) {

**this**.name = name;

**this**.myCount = myCount;

**this**.x = x;

}

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

myCount.save(x, name);

}

}

**class** MyCount {

**private** String id; //账号

**private** **int** cash; //账户余额

Lock lock = **new** Lock(); //账户锁

Condition2 c2test = **new** Condition2(lock);

MyCount(String id, **int** cash) {

**this**.id = id;

**this**.cash = cash;

}

**public** **void** save(**int** x, String name) {

lock.acquire(); //获取锁

**if** (x > 0) {

cash += x; //存款

System.*out*.println(name + "存款" + x + "，当前余额为" + cash);

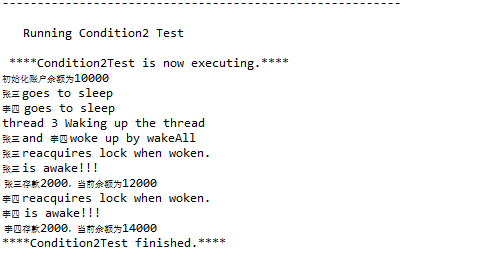
}

lock.release(); //释放锁

}

}

测试结果：



## 3.3 Task1.3 alram类

### 3.3.1 要求分析

一个线程调用waitUntil(x) 后, 它即被挂起. 在当前时钟走过x个滴答后, 该线程被重新唤醒. 线程被唤醒后并不一定立即进入运行态, 只要将其放入就绪队列中即可. 另外, 多个线程可以分别调用waitUntil函数。

### 3.3.2 设计方案

于Alarm类有关的是machine.Timer类. 它在大约每500个时钟滴答使调用回调函数 (由Timer.setInterruptHandler函数设置). 因此, Alarm类的构造函数中首先要设置该回调函数Alarm.timerInterrupt().

为了实现waitUntil, 需要在Alarm类中实现一个队列 Alarm.alarmQueue. 队列中的每个项目是 (线程, 唤醒时间, 相应条件变量) 的三元组. 在调用waitUntil(x) 函数时, 首先得到关于

以上三元组的信息: (线程: 当前线程, 唤醒时间: 当前时间+x, 新分配的条件变量), 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.

### 3.3.3 实现代码

**public** **void** timerInterrupt() {

**//**

**while**(!alarmqueue.isEmpty() && (alarmqueue.peek().wakeTime <= Machine.*timer*().getTime())){

AlarmThread thread = alarmqueue.remove();

thread.waitThread.ready();

}

//

}

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

//

AlarmThread thread = **new** AlarmThread(KThread.*currentThread*(), wakeTime,conditionlock);

alarmqueue.add(thread);

KThread.*sleep*();

//

}

**private** KThread waitThread;

**private** Condition timecondition;

**private** **long** wakeTime;

### 3.3.4 测试代码及结果

创建5个线程，依次“睡眠”20000ticks，线程醒来后各自打印出睡眠开始时间，要求睡眠时间，醒来时间，以及实际睡眠时间。

测试代码：

**public** **static** **void** selfTest(**int** numOfTest){

Runnable a = **new** Runnable() {

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

*AlarmTestThread*();

}

};

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

**int** numOfThreads = 5;

*print*("Creating " + numOfThreads + " num of threads");

**for**(**int** j=0; j<numOfThreads; j++){

KThread thread = **new** KThread(a);

thread.setName("thread");

thread.fork(); }

ThreadedKernel.*alarm*.waitUntil(30000000);

}

}

**static** **void** AlarmTestThread(){

**long** sleepTime =20000;

**long** timeBeforeSleep = Machine.*timer*().getTime();

ThreadedKernel.*alarm*.waitUntil(sleepTime);

**long** timeAfterSleep = Machine.*timer*().getTime();

**long** actualSleepTime = timeAfterSleep - timeBeforeSleep;

*print*(KThread.*currentThread*().toString() + " 被要求在： " + timeBeforeSleep + " 睡眠 " + sleepTime + " ticks");

*print*(KThread.*currentThread*().toString() + " 被唤醒时间： " + Machine.*timer*().getTime() + "要求睡眠时间：: " + sleepTime + " ticks, 实际睡眠时间: " + actualSleepTime);

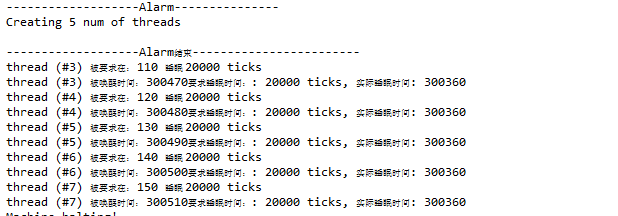
}

**public** **static** **void** print(String Message) {

System.*out*.println(Message);

}

测试结果：



## 3.4 Task1.4 communicator类

### 3.4.1要求分析

用条件变量实现Communicator类中的speaker()和listen()函数一个锁lock

两个条件变量speaker(lock)，listener(lockw)用来用来控制听说动作。

### 3.4.2 设计方案

每个Communicator有一个锁 lock(保证操作的原子性) 和与该锁联系的两个条件变量用于保证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(message变量) 或无listener等待, 则挂起. 否则设置message变量, 准备数据并唤醒一个listener. 在listen函数中, 增加一个listener后, 若speaker数目非零且listener数目为1（及恰好可以配对）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/消费者问题。

### 3.4.3 实现代码

**Communicator.java**

**int** speakersize,listenersize;

**int** Message;

Condition speaker,listener;

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

//

**while**(speakersize!=0)

{speaker.sleep();}

**this**.Message=word;

speakersize++;

**while**(listenersize==0)

{speaker.sleep();}

listener.wake();

speakersize--;

speaker.wake();

//

}

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

//

listenersize++;

**if**(listenersize==1&&speakersize!=0)

speaker.wake();

listener.sleep();

**int** myMessage = **this**.Message;

listenersize--;

//

**return** myMessage;

}

### 3.4.4 测试代码及结果

测试代码：

package nachos.threads;



import nachos.machine.\*;

import java.util.ArrayList;

import java.util.Random;

public class CommunicatorTest {

public CommunicatorTest(){

Message = 1;

numOfSpeakers = 5;

numOfListeners = 5;

communicator = new Communicator();

}

public void commTest(int num){

System.out.println("\nCommunicatorTest开始");

for (int i=0; i<num; i++){

createSpeakers(numOfSpeakers);

createListeners(numOfListeners);

print("\n 演讲者: " + numOfSpeakers);

print(" 收听者: " + numOfListeners + "\n");

sleep(numOfSpeakers+numOfListeners);

print("\n 演讲者与收听者创建完毕");

}

print(" Communicator Test结束 ");

}

public void sleep(int numThreadsCreated){

ThreadedKernel.alarm.waitUntil(numThreadsCreated\*100);

}

public class Listener implements Runnable{

public void run(){

int messageToRecieve = communicator.listen();

System.out.print(Message);

print(" " + KThread.currentThread().getName() + " 收到信息 " + messageToRecieve);

}

}

public class Speaker implements Runnable{

public void run(){

communicator.speak(Message++);

System.out.print(Message);

print(" " + KThread.currentThread().getName() + " 发出信息 " + Message);

}

}

public void createSpeakers(int speakers){

int j;

for(j=1; j<=speakers; j++){

KThread speakerThread = new KThread(new Speaker());

speakerThread.setName("Speaker" + j);

speakerThread.fork();

};

}

public void createListeners(int listeners){

int k;

for(k=1; k<=listeners; k++){

KThread listenerThread = new KThread(new Listener());

listenerThread.setName("Listener" + k);

listenerThread.fork();

}

}

public static void print(String Message) {

System.out.println(Message);

}

public static int MAX\_THREADS = 245;

private int Message;

private Communicator communicator;

private int numOfSpeakers;

private int numOfListeners;

}

测试结果：



## 3.5 Task1.5 priority scheduler类

### 3.5.1 要求分析

实现优先级调度。优先级调度的传统算法如下: 每个线程拥有一个优先级 (在Nachos中, 优先级是一个0到7之间的整数, 默认为1， 在线程调度时, 调度程序选择一个拥有最高优先级的处于就绪状态的线程运行. 这种算法的问题是可能出现 “饥饿” 现象: 设想有一个低优先级的线程处于临界区中运行而高优先级的线程在临界区外等待. 由于前者优先级较低, 它可能不会被调度器选中从而高优先级的线程也不得不浪费时间等待. 需要实现一种 “让出” 优先级的机制 (Priority Donation) : 提高拥有锁的低优先级线程的优先级以使它迅速完成临界区, 不使其它较高优先级的线程等待太久. 重新计算后的优先级称为有效优先级, 它可以不断变化. 以有效优先级为评判标准。

### 3.5.2 设计方案

要完善的的PriorityScheduler继承自Scheduler。

每个KThread类的线程都有一个ThreadState类型的 “线程状态” 对象. 它保存了线程的优先级priority, 有效优先级effectivepriority以及正在等待该线程的线程队列IHave，该线程正在等待的线程队列Iwant。

不同的调度器有不同类型的线程队列 (均继承自ThreadQueue)， 线程队列由threadstate对象组成，是否可提高拥有锁线程优先级取决于创建队列时transferPriority的值，它有两个重要方法: waitForAccess (KThread thread) 和 acquire (KThread thread). 前者将一个线程加入等待队列(实际上加入的是该线程的线程状态); 后者将一个线程从等待队列中取出, 表示该线程当前处于运行状态. 这两个方法中, 首先找出相应线程的线程状态, 再对线程状态执行相应的waitForAccess和acquire操作 (对队列的真正操作是在ThreadState.waitForAccess和ThreadState.acquire中完成的).

设置, 返回, 增加, 减少优先级的函数在Scheduler中. ThreadQueue.nextThread 返回当前队列中有效优先级最高的线程以运行之. 而对某线程有效优先级的计算在ThreadState.getEffectivePriority中. 当该线程没有正在等待该线程的线程队列时，其有效优先级等于其原始优先级，不然其有效优先级增加到不小于其等待队列IHave中各线程的有效优先级的最大值。它也有两个重要方法: waitForAccess (PriorityQueue waitQueue) 和 acquire (PriorityQueue waitQueue). 前者将waitqueue从IHave中移出（如果有的话），加入到Iwant中；后者将waitqueue从Iwant中移出（如果有的话），加入到IHave中。

### 3.5.3 实现代码

实现代码见PriorityScheduler.java

### 3.5.4 测试代码及结果

测试方法：创建两个可捐赠优先级q1，q2及一个不可捐赠优先级队列q3，三个优先级分别高中低线程l，m，h。第一次将l，m加到q1队列，l，m加到q2队列，测试q1的nextthread是m。分别测试优先级是否“捐赠”，低优先级线程提告优先级后是否被选中执行。

测试代码：

package nachos.threads;

import nachos.machine.\*;

import nachos.threads.\*;

import java.util.\*;

public final class PrioritySchedulerTest {

/\*\*

\* A very simple test of priority donation.

\*/

public static void simplePrioritySchedulerTest() {

final boolean intStatus = Machine.interrupt().disable();

final Scheduler s = new PriorityScheduler();

// Enable priority donation.

final ThreadQueue q1 = s.newThreadQueue(false);

final ThreadQueue q2 = s.newThreadQueue(false);

final ThreadQueue q3 = s.newThreadQueue(true);

final Runnable dummyRunnable = new Runnable() {

public void run() {

// do nothing

}

};

// Create dummy Threads

final KThread lowPriorityThread = new KThread(dummyRunnable);

final KThread mediumPriorityThread = new KThread(dummyRunnable);

final KThread highPriorityThread = new KThread(dummyRunnable);

// Link dummy Threads with dummy ThreadState

s.setPriority(lowPriorityThread, PriorityScheduler.priorityMinimum);

s.setPriority(mediumPriorityThread, PriorityScheduler.priorityDefault);

s.setPriority(highPriorityThread, PriorityScheduler.priorityMaximum);

// Test that effectivePriorities start as the initial priorities

Lib.assertTrue(s.getEffectivePriority(lowPriorityThread) == PriorityScheduler.priorityMinimum);

Lib.assertTrue(s.getEffectivePriority(mediumPriorityThread) == PriorityScheduler.priorityDefault);

Lib.assertTrue(s.getEffectivePriority(highPriorityThread) == PriorityScheduler.priorityMaximum);

// Low priority and Medium priority thread are on the ready queue

q1.waitForAccess(lowPriorityThread);

q1.waitForAccess(mediumPriorityThread);

q2.waitForAccess(lowPriorityThread);

q2.waitForAccess(mediumPriorityThread);

// Now, medium priority thread should be the next Thread

Lib.assertTrue(q1.nextThread().equals(mediumPriorityThread),

"nextThread 应该是 mediumpriority thread");

// Low priority thread acquires the lock

q3.acquire(lowPriorityThread);

// High priority thread waits for the lock

q3.waitForAccess(highPriorityThread);

int a=s.getEffectivePriority(lowPriorityThread);

int b=s.getEffectivePriority(highPriorityThread);

// System.out.print(a+" "+b);

// Check if priority is donated

Lib.assertTrue(s.getEffectivePriority(lowPriorityThread) == PriorityScheduler.priorityMaximum,

"高优先级应该被捐赠给低优先级线程");

// Make sure Medium priority thread still has same priority

Lib.assertTrue(s.getEffectivePriority(mediumPriorityThread) == PriorityScheduler.priorityDefault,

"Medium priority thread有效优先级应该不变");

// Now, low priority thread should be the next Thread

Lib.assertTrue(q2.nextThread().equals(lowPriorityThread),

"Low priority thread 优先级提高后应该成为nextthread");

// This line will only be reached if all these assertions pass.

System.out.println("simplePrioritySchedulerTest 测试通过");

Machine.interrupt().restore(intStatus);

}

/\*\* Forks a KThread of each priority and makes sure that it runs in the correct order. This test will

\* only pass if PriorityScheduler is set as the Machine's scheduler in nachos.conf

public static void prioritySchedulerReadyQueueTest() {

final List<Integer> resultList = new LinkedList<Integer>();

final List<KThread> threads = new LinkedList<KThread>();

final boolean intStatus = Machine.interrupt().disable();

for (int curPriority = PriorityScheduler.priorityMinimum + 1; curPriority <= PriorityScheduler.priorityMaximum; ++curPriority) {

final KThread curThread = new KThread(new AppenderTestThread(curPriority, resultList));

ThreadedKernel.scheduler.setPriority(curThread, curPriority);

threads.add(curThread);

}

Machine.interrupt().restore(intStatus);

for (final KThread curThread : threads)

curThread.fork();

for (final KThread curThread : threads)

curThread.join();

// Lib.assertTrue(resultList.equals(Arrays.asList(7, 6, 5, 4, 3, 2, 1)));

System.out.println("prioritySchedulerReadyQueueTest has passed successfully!");

}

\*/

/\*\* A simple test thread that appends its id to a list. \*/

private static class AppenderTestThread implements Runnable {

AppenderTestThread(int id, List<Integer> list) {

this.id = id;

this.appendList = list;

}

public void run() {

appendList.add(id);

}

private int id;

private final List<Integer> appendList;

}

public static void priorityDonationWithMutablePriorityTest() {

final Scheduler s = new PriorityScheduler();

final Runnable dummyRunnable = new Runnable() {

public void run() {

// do nothing

}

};

final KThread resourceHolder = new KThread(dummyRunnable);

final KThread prioritySwitcher = new KThread(dummyRunnable);

// disable interrupts as we will be modifying priorities

final boolean intStatus = Machine.interrupt().disable();

s.setPriority(resourceHolder, 0);

s.setPriority(prioritySwitcher, 4);

// sanity checks

Lib.assertTrue(s.getEffectivePriority(resourceHolder) == 0);

Lib.assertTrue(s.getEffectivePriority(prioritySwitcher) == 4);

// The resource, backed by a thread queue that donates priority

final ThreadQueue resource = s.newThreadQueue(true);

// Give the resource to resourceHolder

resource.acquire(resourceHolder);

// Priority should be unchanged

Lib.assertTrue(s.getEffectivePriority(resourceHolder) == 0);

// prioritySwitcher now waits for the resource

resource.waitForAccess(prioritySwitcher);

// Check that priority was donated

Lib.assertTrue(s.getEffectivePriority(resourceHolder) == 4);

// Now we increase the priority of prioritySwitcher

s.setPriority(prioritySwitcher, 5);

// Check to see that prioritySwitcher's priority was increased successfully

Lib.assertTrue(s.getEffectivePriority(prioritySwitcher) == 5);

// Check to see that priority donation is still correct

Lib.assertTrue(s.getEffectivePriority(resourceHolder) == 5);

// Change prioritySwitcher's priority once more

s.setPriority(prioritySwitcher, 3);

// Check to see if that worked

Lib.assertTrue(s.getEffectivePriority(prioritySwitcher) == 3);

Lib.assertTrue(s.getEffectivePriority(resourceHolder) == 3);

// re-enable interrupts

Machine.interrupt().restore(intStatus);

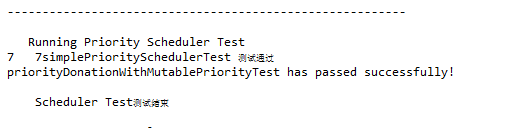
// If this line is reached, the test has passed

System.out.println("priorityDonationWithMutablePriorityTest has passed successfully!");

}

}

测试结果：



## 3.6 Task1.6 boat类

### 3.6.1 要求分析

A岛=molokai B岛=oahu

解决boat问题的主要思想：我们的解决方法是：首先让两个孩子划船从A岛到B岛，放下一个孩子在B岛，另一个孩子划船回A岛再接一个孩子到B岛，这样循环直到A岛上没有剩下的孩子，这时B岛上一个孩子划船到A岛上唤醒一个大人，由大人划船回到B岛，唤醒一个孩子，这个被唤醒的孩子划船到A岛将上一次留下的孩子带回B岛。如果A岛上有大人需要走，那么一个孩子划船回到A岛重复这个过程直到A岛上没有剩下的大人，孩子划船回到B岛，然后结束。

### 3.6.2 设计方案

我们的解决方法是：首先让两个孩子划船从A岛到B岛，放下一个孩子在B岛，另一个孩子划船回A岛再接一个孩子到B岛，这样循环直到A岛上没有剩下的孩子，这时B岛上一个孩子划船到A岛上唤醒一个大人，由大人划船回到B岛，唤醒一个孩子，这个被唤醒的孩子划船到A岛将上一次留下的孩子带回B岛。如果A岛上有大人需要走，那么一个孩子划船回到A岛重复这个过程直到A岛上没有剩下的大人，孩子划船回到B岛，然后结束。

### 3.6.3 实现代码

public class Boat {

static BoatGrader bg;

static String boatLocation;

static boolean isThereAtLeast1childOnMolokai;

static boolean done;

static boolean amIpilot;

static boolean adultHasBoat;

static int numOfChildrenOnOahu;

static int numOfChildrenOnMolokai;

static int numOfChildrenWaitingForBoat;

static int boatPasses;

static int numOfAdultsOnOahu;

static int numOfAdultsOnMolokai;

static int message;

static Lock lock;

static Condition childOnOahu;

static Condition childOnMolokai;

static Condition adultOnOahu;

static Condition boat;

static Alarm alarm;

static Communicator walkyTalky;

// Testing & Debuging Varibles

//static boolean prints = false;

static boolean prints = true;

static boolean bgPrints = true;

public static void selfTest(int numTest, boolean testingPrints) {

BoatGrader b = new BoatGrader();

bgPrints = false;

int ch, ad;

int minNumOfChidren = 2;

int maxNumOfThreads = 24;

Random random = new Random();

random.setSeed(System.currentTimeMillis());

for (int i = 1; i < numTest; i++) {

ch = randomIntInRange(minNumOfChidren, maxNumOfThreads, random);

ad = random.nextInt(maxNumOfThreads - ch + 1);

if (testingPrints) {

print(" Testing Boats with " + ch + " children, " + ad + " adults");

}

begin(ad, ch, b);

}

print(" \*-\*-\*-\* All Boat Test Passed \*-\*-\*-\*");

}

public static int randomIntInRange(int lowBound, int upBound, Random rand) {

long range = (long) upBound - (long) lowBound + 1;

long fraction = (long) (range \* rand.nextDouble());

int randomNumber = (int) (fraction + lowBound);

return randomNumber;

}

public static void print(String aMessage) {

System.out.println(aMessage);

}

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;

// Instantiate global variables here

boatLocation = "Oahu";

isThereAtLeast1childOnMolokai = false;

done = false;

amIpilot = false;

adultHasBoat = false;

numOfChildrenOnOahu = 0;

numOfChildrenOnMolokai = 0;

numOfChildrenWaitingForBoat = 0;

boatPasses = 0;

numOfAdultsOnOahu = 0;

numOfAdultsOnMolokai = 0;

message = 0;

lock = new Lock();

childOnOahu = new Condition(lock);

childOnMolokai = new Condition(lock);

adultOnOahu = new Condition(lock);

boat = new Condition(lock);

alarm = new Alarm();

walkyTalky = new Communicator();

if (prints) {

print("\*\*\*Created Instance varibles\*\*\*");

}

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

// Walkthrough linked from the projects page.

Runnable a = new Runnable() {

public void run() {

AdultItinerary();

}

};

if (prints) {

print("\*\*\*Created Adult Itinerary\*\*\*");

}

Runnable c = new Runnable() {

public void run() {

ChildItinerary();

}

};

if (prints) {

print("\*\*\*Created Child Itinerary\*\*\*");

}

for (int i = 1; i <= children; i++) {

KThread child = new KThread(c);

child.setName("Child: " + i);

if (prints) {

print("\*\*\*Child " + i + " created\*\*\*");

}

child.fork();

}

for (int j = 1; j <= adults; j++) {

KThread adult = new KThread(a);

adult.setName("Adult: " + j);

if (prints) {

print("\*\*\*Adult " + j + " created\*\*\*");

}

adult.fork();

}

int threadsCraeated = children + adults;

while (threadsCraeated != message) {

if (prints) {

print("\*\*\*Parent Thread went to sleep\*\*\*");

}

message = walkyTalky.listen();

}

if (prints) {

print("\*\*\*Everybody got across\*\*\*");

}

done = true;

alarm.waitUntil(threadsCraeated \* 150);

if (prints) {

print("\*\*\*All Threads joined\*\*\*");

}

}

static void AdultItinerary() {

/\* 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

\*/

lock.acquire();

numOfAdultsOnOahu++;

if (prints) {

print("\*\*\*Adult went to sleep on Oahu\*\*\*");

}

adultOnOahu.sleep();

if (prints) {

print("\*\*\*Adult woke up\*\*\*");

}

if (bgPrints) {

bg.AdultRowToMolokai();

}

if (prints) {

print("\*\*\*Adult rowed to Molokai\*\*\*");

}

numOfAdultsOnOahu--;

boatLocation = "Molokai";

numOfAdultsOnMolokai++;

if (prints) {

print("\*\*\*Adult tried to wake up a child on Molokai\*\*\*");

}

childOnMolokai.wake();

adultHasBoat = false;

if (prints) {

print("\*\*\*Adult Thead terminated\*\*\*");

}

lock.release();

}

static void ChildItinerary() {

lock.acquire();

while (!done) {

numOfChildrenOnOahu++;

boatPasses++;

if (prints) {

print("\*\*\*Child arived on Oahu\*\*\*");

}

while (boatLocation.equals("Molokai") || boatPasses > 2 || adultHasBoat) {

boatPasses--;

if (prints) {

print("\*\*\*Child went to sleep on Oahu\*\*\*");

}

childOnOahu.sleep();

if (prints) {

print("\*\*\*Child woke up on Oahu\*\*\*");

}

boatPasses++;

}

childOnOahu.wake();

numOfChildrenWaitingForBoat++;

if (numOfChildrenWaitingForBoat < 2 | boatLocation.equals("Molokai")) {

if (numOfChildrenOnOahu == 1 && isThereAtLeast1childOnMolokai) {

adultOnOahu.wake();

if (prints) {

print("\*\*\*Child woke up Adult\*\*\*");

}

adultHasBoat = true;

}

if (prints) {

print("\*\*\*Child Pilot went to sleep on boat\*\*\*");

}

boat.sleep();

boatPasses -= 2;

if (bgPrints) {

bg.ChildRideToMolokai();

}

if (prints) {

print("\*\*\*Child Pilot woke up in boat & went to Molokai\*\*\*");

}

numOfChildrenOnOahu -= 2;

numOfChildrenOnMolokai += 2;

numOfChildrenWaitingForBoat--;

}

boat.wake();

if (!amIpilot) {

amIpilot = true;

numOfChildrenWaitingForBoat--;

if (bgPrints) {

bg.ChildRowToMolokai();

}

if (prints) {

print("\*\*\*Child passenger rode to Molokai\*\*\*");

}

boatLocation = "Molokai";

if (prints) {

print("\*\*\*Child Passenger went to sleep on Molokai\*\*\*");

}

childOnMolokai.sleep();

if (prints) {

print("\*\*\*Child old passenger woke up on Molokai\*\*\*");

}

} else if (numOfChildrenOnOahu + numOfAdultsOnOahu == 0) {

walkyTalky.speak(numOfChildrenOnMolokai + numOfAdultsOnMolokai);

if (prints) {

print("\*\*\*Child Pilot signaled Parent that he's done\*\*\*");

}

alarm.waitUntil(50);

}

if (!done) {

amIpilot = false;

if (prints) {

print("\*\*\*Child rowed back to Oahu\*\*\*");

}

if (bgPrints) {

bg.ChildRowToOahu();

}

boatLocation = "Oahu";

numOfChildrenOnMolokai--;

isThereAtLeast1childOnMolokai = (numOfChildrenOnMolokai > 0);

if (prints) {

print("\*\*\*there is at least 1 child on Molokai? " + isThereAtLeast1childOnMolokai + " \*\*\*");

}

}

}

childOnMolokai.wakeAll();

if (prints) {

print("\*\*\*Child Thead terminated\*\*\*");

}

lock.release();

}

}

### 3.6.4 测试代码及结果

## 3.7 Task2.1 系统调用creat, open, read, write, close, unlink

### 3.7.1 要求分析

系统共提供了七个系统调用: halt (关机, 已经提供), creat (创建并打开文件), open (打开文件), read (读IO), write (写IO), close (关闭IO), unlink (删除文件). 当发生系统调用时, UserProcess.handleSyscall被调用, 我们在其中编写系统调用处理函数. 要求是：(1) 稳定性, 不能因为一个进程的非法系统调用就使操作系统崩溃, 而应该返回错误代码. (2) halt 调用只能由第一个进程 (root process) 执行. (3) 系统调用需要读写内存时, 通过UserProcess.readVirtualMemory和UserProcess.writeVirtualMemory进行. (4) 文件名以null结尾, 不超过256字符. (5) 如果系统调用出错, 应返回 -1. (6) 为每个打开的IO文件分配一个 “文件描述符”, 用整数表示. 每个进程最多可以拥有16个. 其中0和1应分配给标准输入和标准输出 (即屏幕), 这由SynchConsole类管理. 不同进程可以用相同的文件描述符处理不同的文件 (7) Nachos已经提供了一个简单的文件系统Machine.FileSystem, 可以通过ThreadedKernel.fileSystem访问. (8) 系统不需要考虑文件访问的互斥等问题.

### 3.7.2 设计方案

Halt系统调用

为每个进程增加一个ID，root ProcessID为0，在系统调用前, 检查调用的进程是否为rootProcess.是rootProcess则停机，否则返回-1.

Creat系统调用

注:打开现存文件或新建文件时，内核会返回一个文件描述符。读写文件也需要使用文件描述符来指定待读写的文件。

若进程已打开相同文件描述符文件，则不创建，返回该文件的文件描述符，

为了实现 “文件描述符”, 为每个进程开一张大小为16的数组 (本地描述符表), 下标为描述符编号, 内容为文件对象 (OpenFile类型, 描述符未使用为null). 1.用readVirtualMemoyString读取文件名（第一个参数为文件名的内存地址) 2.判断文件名是否已有效，为文件创建一个新的文件描述符3.通过UserKernel.fileSystem.open打开文件 (第二个参数为true 表示创建新文件) 维护本地描述符表 (返回一个内容为null的项目的下标作为描述符, 将文件对象填入) 4.返回文件描述符.

Open系统调用

与Create调用很类似, 唯一的一个参数为文件名的内存地址. 在Open系统调用中进行如下操作: 1.从内存读取文件名2.通过UserKernal.fileSystem.open打开文件 (第二个参数为false) 3.维护本地描述符表 4.返回文件描述符.

Read系统调用

Read系统调用的三个参数依次为: 文件描述符, 写入的内存地址, 读取的字节数. 在Read系统调用中进行如下操作: 1.从本地描述符表中得到文件对象2.通过OpenFile.read读取文件内容3.将文件内容写入内存4.返回写入内存的字节数.

Write系统调用

Write系统调用的三个参数依次为: 文件描述符, 读内存的地址, 写入文件的字节数. 在Write系统调用中进行如下操作: 1。从本地描述符表中得到文件对象2.访问内存, 3.得到要写入文件的内容4.通过OpenFile.write写文件5.返回写入文件的字节数.

Close系统调用

Close系统调用的唯一一个参数为文件描述符. 在Close系统调用中进行如下操作: 1.从本地描述符表中得到文件对象2.通过OpenFile.close关闭文件3.从本地描述符表中移出文件对象4.返回SUCCESS 0.

Unlink系统调用

Unlink系统调用的唯一一个参数为文件名的内存地址. 操作如下：1.1.从内存读取文件名2.

从本地描述符表中得到文件对象3.从本地描述符表中删除该文件名.

### 3.7.3 实现代码

**UserProcess.java中**

**private** **int** **handleHalt**() {

**if** (**this**.processID != 0)

**return** -1;

Machine.*halt*();

//

**return** 0;

}

**private** **int** **getUnusedFileDescriptor**() **throws** NachosInternalException {

**for** (**int** i = 0; i < **this**.openFiles.length; ++i) {

**if** (**this**.openFiles[i] == **null**) {

**return** i;

}

}

**throw** **new** NachosOutOfFileDescriptorsException("No file descriptors.");

}

**private** **int** **handleCreat**(**final** **int** pName) **throws** NachosInternalException {

**final** String fileName = readVirtualMemoryString(pName, *MAX\_FILENAME\_LENGTH*);

//

**for** (**int** i = 0; i < **this**.openFiles.length; ++i) {

**if** (**this**.openFiles[i] != **null** && **this**.openFiles[i].getName().equals(fileName)) {

**return** i;

}

}

**final** **int** fileDescriptor = **this**.getUnusedFileDescriptor();

**final** OpenFile file = ThreadedKernel.*fileSystem*.open(fileName, **true**);

//

**this**.openFiles[fileDescriptor] = file;

**return** fileDescriptor;

}

**private** **int** **handleOpen**(**final** **int** pName) **throws** NachosInternalException {

**final** **int** fileDescriptor = **this**.getUnusedFileDescriptor();

**final** String fileName = readVirtualMemoryString(pName, *MAX\_FILENAME\_LENGTH*);

//

**final** OpenFile file = ThreadedKernel.*fileSystem*.open(fileName, **false**);

//

**this**.openFiles[fileDescriptor] = file;

**return** fileDescriptor;

}

**private** **int** **handleRead**(**final** **int** fileDescriptor, **final** **int** pBuffer, **final** **int** count)

**throws** NachosInternalException {

//

**final** OpenFile file = **this**.openFiles[fileDescriptor];

**final** **byte**[] tmp = **new** **byte**[count];

**final** **int** numBytesRead = file.read(tmp, 0, count);

**final** **int** numBytesWritten = writeVirtualMemory(pBuffer, tmp, 0, numBytesRead);

//

);

}

**return** numBytesRead;

}

**private** **int** **handleWrite**(**final** **int** fileDescriptor, **final** **int** pBuffer, **final** **int** count)

**throws** NachosInternalException {

//

**final** OpenFile file = **this**.openFiles[fileDescriptor];

**final** **byte**[] tmp = **new** **byte**[count];

**final** **int** numBytesToWrite = readVirtualMemory(pBuffer, tmp);

//

**return** file.write(tmp, 0, numBytesToWrite);

}

**private** **int** **handleClose**(**final** **int** fileDescriptor) **throws** NachosInternalException {

//

OpenFile file = openFiles[fileDescriptor];

//

openFiles[fileDescriptor] = **null**;

file.close();

**return** *SUCCESS*;

}

**private** **int** **handleUnlink**(**final** **int** pName) **throws** NachosInternalException {

String fileName = readVirtualMemoryString(pName, *MAX\_FILENAME\_LENGTH*);

//

**for** (**int** i = 0; i < openFiles.length; i++) {

**if** ((openFiles[i] != **null**) && (openFiles[i].getName().equals(fileName))) {

openFiles[i] = **null**;

**break**;

}

}

//

**return** *SUCCESS*;

}

### 3.7.4 测试代码及结果

2.1与2.2测试合并,见3.8.4的测试代码及报告.

## 3.8 Task2.2 多进程内存分配及访问

### 3.8.1 要求分析

由于 Nachos 支持多道程序设计, 所以不同进程应分配完全不同的物理内存。Nachos不支持动态内存分配, 故在装入程序时就要为每个进程一次性分配固定的物理内存, 在进程结束时收回它们。 还需要实现简单的虚拟内存方案: 每个进程的地址空间是连续的虚拟内存 (以页面为单位, 1页=1024字节)，但这些连续的虚拟页面在物理内存中却不一定是连续的。这个方案很简单，虚拟空间的总容量和物理空间的总容量相等，映射机制只是从虚拟内存到物理内存的一一映射。

### 3.8.2 设计方案

为了实现上面的内存分配方案, 需要用到

数据结构:系统提供了页表pageTable, 它以虚拟页号为下标, 其中的每个项目是一个machine.TranslationEntry类型的对象.

还需要一个全局队列freePages, 用于存放当前空闲的物理页面号，还有一个保护全局队列的锁pageLock。开始时, freePages包括所有的物理页面。

需要装入的进程是一个 machine.Coff 类型的对象,它包括若干个段。每个段是一个machine.CoffSection类型的对象,每个CoffSection又包括若干个页。

系统载入进程对象时已经得到进程所需页数numPages，则从freePages中出队相应数量的页面 （物理页面）。

在UserKernel中编写两个方法 malloc和free，实现简单的分配内存和释放内存操作。

当启动新进程时, UserProcess.load 负责从磁盘装入进程。需要装入的进程是一个 machine.Coff 类型的对象,它包括若干个段。每个段是一个machine.CoffSection类型的对象,它又包括若干个页。 当启动新进程时, UserProcess.load 负责从磁盘装入进程。其中需要我们完善的是UserProcess.loadSections。

UserProcess.readVirtualMemory 读虚拟内存,它可以读取任何虚拟地址开始的任意数目的字节 (在进程可访问的内存范围内)。

参数：

int vaddr 要读的虚拟内存的初始地址

byte[] data 存放数据的数组

int offset 往data中写入数据的初始地址

int length 要读取的数据长度

变量：

numBytesRead 已经读取的字节数 初始值为0

freeBytes 数组data中还可以写入的字节数 初始为：

它可以读取任何虚拟地址开始的任意数目的字节 (在进程可访问的内存范围内). 首先, 得到起始和终止的虚拟内存页面号和偏移量 (通过参数计算得). 然后得到起始和终止的物理内存的页面号 (通过pageTable得到). 最后读3部分: 第一页的部分, 中间的若干整页, 最后一页的部分.

类似地, UserProcess.writeVirtualMemory写虚拟内存, 其方法与readVirtualMemory类似. 只是注意取页面时要检查页面是否为只读.

### 3.8.3 实现代码

**UserKernel.java中**

**public** List<Integer> malloc(**int** numPages) {

//

List<Integer> toRet = **new** ArrayList<Integer>();

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

toRet.add(freePages.remove());

}

//

**return** toRet;

}

**public** **void** free(List<Integer> allocatedPages) {

//

**for**(Integer page: allocatedPages) {

freePages.add(page);

}

//

}

**UserProcess.java中**

**protected** **boolean** loadSections() {

//

//利用malloc方法从freePages中出队相应数量的页面。

allocatedPages = ((UserKernel) Kernel.*kernel*).malloc(numPages);

//

//每个虚拟页在 pageTable 中创建一个新的 TranslationEntry 对象

pageTable = **new** TranslationEntry[numPages];

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

pageTable[i] = **new** TranslationEntry( i, allocatedPages.get(i), **true**, **false**, **false**, **false** );

}

//利用循环嵌套对每一页调用CoffSection.loadPage方法将页的内容读入

**for** (**int** s = 0; s < coff.getNumSections(); ++s) {

CoffSection section = coff.getSection(s);

//

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

**int** vpn = section.getFirstVPN() + i;

pageTable[vpn].readOnly = section.isReadOnly();

section.loadPage(i, pageTable[vpn].ppn);

}

}

**return** **true**;

}

**protected** **void** unloadSections() {

((UserKernel) Kernel.*kernel*).free(allocatedPages);

coff.close();

}

**public** **int** readVirtualMemory(**int** vaddr, **byte**[] data, **int** offset, **int** length) {

**int** numBytesRead = 0;

**if** (inVaddressSpace(vaddr)) {

**int** freeBytes = data.length-offset; //data中可用byte数

**if** ( (data.length>0) &&(length>0)&&(data.length>=length)) {

**int** vpn = vaddr / *pageSize*; //页面号

**int** pageOffset = vaddr % *pageSize*; //页偏移量

**final** **int** startPhysMem = (pageTable[vpn].ppn \* *pageSize* + pageOffset); //物理地址起始处

**if** (inPhysAddressSpace(startPhysMem)) {

**int** srcOffset, destOffset, amountToRead;

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

**do**{

amountToRead = Math.*min*(freeBytes, Math.*min*(*pageSize* - pageOffset, length - numBytesRead));

destOffset = offset + numBytesRead; //数组data中要存放数据的初始地址

srcOffset = (pageTable[vpn].ppn\**pageSize*)+pageOffset; //原数组要复制的初始地址 即物理内存的地址 System.*arraycopy*(memory, srcOffset, data, destOffset, amountToRead);

numBytesRead += amountToRead; freeBytes -= amountToRead;

pageOffset = 0;

vpn++;

}**while** ( numBytesRead<length &&

//

);

}

}

}

**return** numBytesRead;

}

**public** **int** writeVirtualMemory(**int** vaddr, **byte**[] data, **int** offset,

**int** length) **throws** NachosInternalException {

**int** numBytesWritten = 0;

**if** (inVaddressSpace(vaddr)){

**int** writeBytes = data.length-offset;

**if** ( (length>0)&&(data.length>0)&&

(data.length>=length) && (writeBytes>0) ) {

**int** vpn = vaddr / *pageSize*;

**int** pageOffset = vaddr % *pageSize*;

**final** **int** startPhysMem = pageTable[vpn].ppn \* *pageSize*+pageOffset;

**if**(inPhysAddressSpace(startPhysMem)&&

!pageTable[vpn].readOnly) {

**int** amountToWrite, srcOffset, destOffset;

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

**do**{

amountToWrite = Math.*min*(writeBytes, Math.*min*(*pageSize* - pageOffset, length - numBytesWritten));

srcOffset = offset + numBytesWritten;

destOffset=(pageTable[vpn].ppn\**pageSize*)+pageOffset;

System.*arraycopy*(data, srcOffset, memory,

destOffset, amountToWrite);

numBytesWritten += amountToWrite;

writeBytes -= numBytesWritten;

pageOffset = 0;

pageTable[vpn].dirty = **true**; // 标记dirty

pageTable[vpn].used = **true**; // 标记used

vpn++;

} **while** ( (numBytesWritten < length) && (vpn <= pageTable.length)&&(pageTable[vpn].valid)&&(!pageTable[vpn].readOnly) && inPhysAddressSpace(destOffset)&&(writeBytes>0) );

}

}

}

**return** numBytesWritten;

}

### 3.8.4 测试代码及结果

设计一个应用程序，包含这5组函数调用：

1.首先创建一个文件 test;

2.打开这个文件test,获得它的返还数值 fd；

3.通过fd, 在这个文件里写入一行字符，比如："hello,world.I am YePeng.How are ou?\n" 这37个字节，然后关闭文件；

4.再次打开这个文件test, 获得新的fd;

5.调用read()函数，从这个打开的文件里读取40个字节的文件（实际上，它只能读取37个字节）；

6关闭这个文件。

这样，通过这一条龙的操作，把所有想测试的系统调用函数都包括进去了，没调用一个函数成功后，通过printf()和printfNum()这两个小函数检验以下运行的过程。

添加更改的代码：

1.编写一个叫homework.c的文件：

2.在makefile TARGETS后添加homewok

#include "syscall.h"

#include "stdio.h"

#include "stdlib.h"

int mystrlen(char \*buffer) //similar like strleng() ,get the length of the char arry

{

int i;

for(i=0;i<500;i++)

{

if(buffer[i]==0)

return i;

}

return -1;

}

void main()

{

int fd=0;

char \*filename = "aa.txt";

int ByteNum;

char \*buffer = "Hello! This is the test for Task2.1.\n";

char buffersize = mystrlen(buffer);

char buf[40]; //testing for Create(char \*)

creat(filename);

printf("Calling 'creat(filename)'... ");

printf(" done!\n"); //testing for Open()

fd = open(filename);

printf("Calling 'fd = open(filename)'... done!\n");

printf("Return value fd = ");

printf("\n"); //testing for Write()

write(fd, buffer, buffersize);

close(fd);

printf("Calling 'Write(buffer, buffersize, fd)' ...done!\n"); //testing for Read()

fd = open(filename);

int i;

ByteNum=read(fd, buf, 40);

printf("Calling 'Read(buf, 40, fd)' ... Done\n");

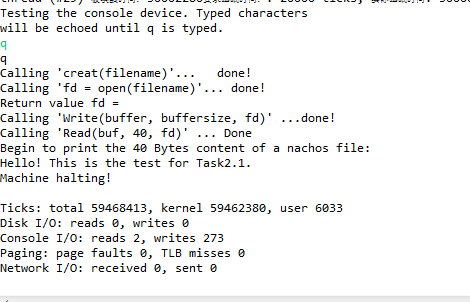
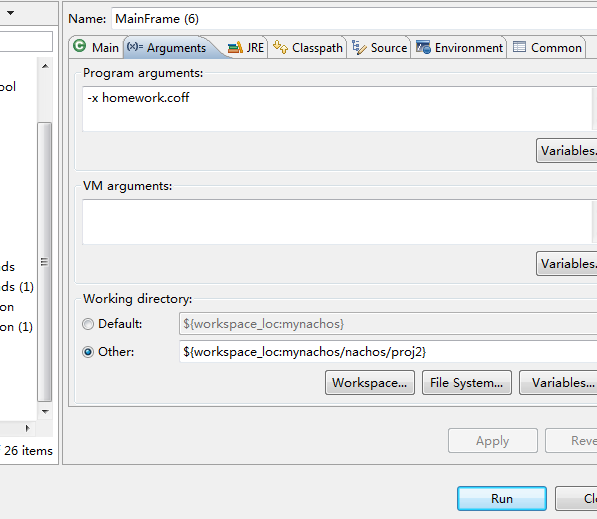
printf("Begin to print the 40 Bytes content of a nachos file:\n");

printf(buf);

close(fd);

halt();

}



## 3.9 Task2.3 系统调用exec, join, exit

### 3.9.1 要求分析

这三个系统调用与进程调度有关. 其中: exec启动一个新的进程; join与线程中的join操作类似, 等待某进程结束后当前线程继续; exit退出当前进程. 需要注意的是: (1) 父进程和子进程不共享任何的内存, 文件或其它状态. (2) 只有父进程能对子进程进行join操作. 例如A执行B, B执行C, 则A不允许join C, 而B允许join C. (3) 需要为每个进程分配一个唯一的进程编号 (4) exit操作将使当前进程立即结束, 如果父进程对其进行join操作, 返回代码应返回. (5) 最后一个Root Process调用 exit 的进程将使系统停机.

### 3.9.2 设计方案

在进程对象的构造函数中将计数器加1作为ProcessId.

此外, 需要为每个进程维护一个HashSet childcreated, 存放所有子进程的编号

还要维护一个全局的Hashmap pidThreadMap, 存放 (进程号, 进程对象) 元组, 在每个进

Exit系统调用

该系统调用的唯一参数为返回值. 需要做的工作：1.关闭所有打开的文件2.调用unloadsections释放内存3.从pidThreadMap中移出当前进程编号如果是最后一个进程(pidThreadMap为空) 调用Kernel.kernel.terminate() 停机4.调用KThread.finish方法结束当前线程 (在Nachos中每个用户进程只有一个线程) .

Join系统调用

该系统调用中, 第一个参数为子进程编号, 第二个参数一个地址, 用于保存子进程的返回值. 需要做的工作是: 1.检查childProcesses, 如果子进程编号不在其中则出错返回 2.Uthread继承Kthread，直接调用已实现的join方法 3.等子进程返回时, 得到返回代码调用Lib.bytesFromInt()，将返回值转换为byte型数组，将其写入第二个参数表示的地址中.

Exec系统调用

在该系统调用中, 第一个参数为文件名地址, 第二个参数为参数个数, 第三个参数为参数表指针. 1.得到文件名 2.用第三个参数作为虚拟内存地址得到参数表数组的首址

3.创建子进程, 将子进程加入到父进程私有的一个HashSet中 4.用UserProcess.execute方法执行子进程 5.返回子进程编号.

### 3.9.3 实现代码

**UserProcess.java**中

**private** **int** handleJoin(**int** processID, **int** pStatus) {

//

UThread uThread = *pidThreadMap*.get(processID);

uThread.join();

**int** Status = uThread.process.processStatus;

//

**byte**[] statusBytes = **new** **byte**[*sizeOfInt*];

Lib.*bytesFromInt*(statusBytes, 0, Status);

**int** statusBytesWritten = writeVirtualMemory(pStatus, statusBytes);

**//**

**return** 0; }

**private** **int** handleExec(**int** pFile, **int** argc, **int** pArgv) {

//

**byte**[] argvBytes = **new** **byte**[argc \* *sizeOfInt*];

**if** (readVirtualMemory(pArgv, argvBytes, 0, argvBytes.length) == argvBytes.length) {

**int**[] argvAddrs = **new** **int**[argc];

**for** (**int** i = 0; i < (argc \* *sizeOfInt*); i += *sizeOfInt*){

argvAddrs[i / *sizeOfInt*] = Lib.*bytesToInt*(argvBytes, i);

}

String[] argvStrings = **new** String[argc];

//

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

argvStrings[i] = readVirtualMemoryString(argvAddrs[i], Math.*min*(remainingBytes, 256));

**//**

remainingBytes -= argvStrings[i].length();

}

UserProcess childProcess = UserProcess.*newUserProcess*();

**if**(childProcess.execute(fileName,argvStrings)){

childrenCreated.add(childProcess.processID);

**return** childProcess.processID;

}

}

}

**return** *ERROR*;

}

**private** **void** handleExit(**int** status) {

**for** (OpenFile file : openFiles) {

**if** (file != **null**)

file.close();

}

**this**.unloadSections();

**if** (*pidThreadMap*.size() == 1) {

Kernel.*kernel*.terminate();

}

*pidThreadMap*.remove(**this**.processID);

processStatus = status;

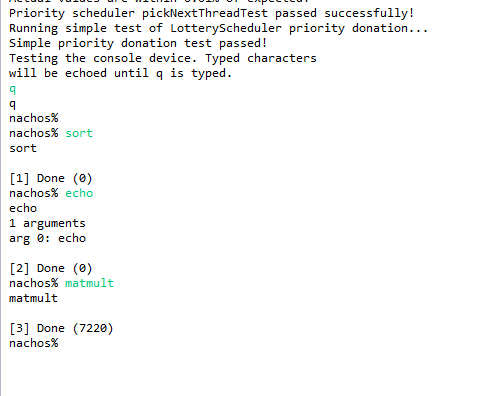
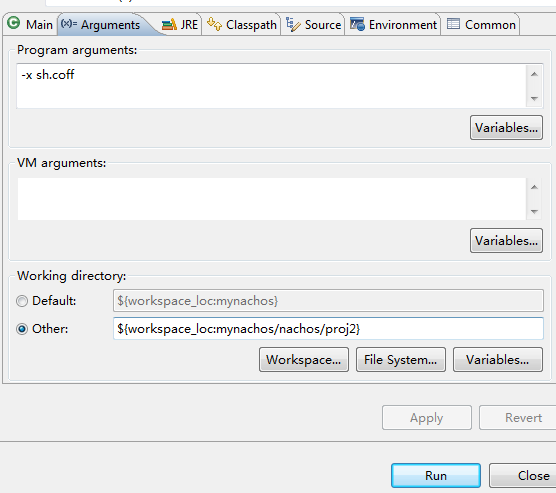
exitSuccess = **true**;

UThread.*finish*();

}

### 3.9.4 测试代码及结果

测试方法：执行sh.coff，相当于控制台程序，可执行多个程序。



## 3.10 Task2.4 Lottery Schedule类

### 3.10.1 要求分析

彩票调度 (Lottery Schedule) 和Task 1.5的优先级调度非常类似, 只是下面两点作了改变: (1) “优先级” 的概念变成了 “彩票”, 即表示该线程下次被运行的几率 (2) 在调度过程中, 并不是选择彩票数最多的线程运行, 而是随机抽取一张彩票, 让彩票的主人运行. 这样, 彩票越多, 下次得到的运行机会就越大.

### 3.10.2 设计方案

我们只要修改PrioritySchedueler.java中的两处即可实现彩票调度. 首先在getEffectivePriority

中将选取最大优先级的过程改成将所有等待线程的优先级 (彩票数) 加到owner上 (即等待者把自己拥有的彩票给获得锁的线程) , 其次将nextThread修改如下: 首先遍历一遍队列, 计算出当前队列中的所有队列的彩票总数 (考虑过donation后的) S, 其次生成一个 0到S-1之间的随机数R表示抽取的彩票, 最后再次遍历一遍队列, 用累加器T计算彩票总数, 当T的当前值大于R时说明当前循环到的线程正好落在目标区间内, 即选出该线程.

由于彩票调度和优先级调度差异很小, 可以把彩票调度的实现继承PriorityScheduler中. 重写PriorityQueue.getEffectivePriority(),

PriorityQueue.pickNextThread()，ThreadState.getEffectivePriority()方法即可.

### 3.10.3 实现代码

见threads/PriorityScheduler.java

threads/LotteryScheduler.java

### 3.10.4 测试代码及结果

在测试时需要修改配置文件 nachos.conf中的ThreadedKernel.scheduler项目为nachos.threads.LotteryScheduler

测试代码如下：

**public** **final** **class** LotterySchedulerTest {

**public** **static** **void** testPickNextThread() {

**final** **int** NUM\_ITERATIONS = 10000000;

**final** **double** TOLERANCE = 0.01;

**final** **boolean** intStatus = Machine.*interrupt*().disable();

**final** Scheduler s = **new** LotteryScheduler();

// fake thread Queue

**final** ThreadQueue queue = s.newThreadQueue(**true**);

**final** KThread waiter20 = **new** KThread(**null**);

**final** KThread waiter25 = **new** KThread(**null**);

**final** KThread waiter50 = **new** KThread(**null**);

**final** KThread waiter5 = **new** KThread(**null**);

s.setPriority(waiter20, 20);

s.setPriority(waiter25, 25);

s.setPriority(waiter50, 50);

s.setPriority(waiter5 , 5);

queue.waitForAccess(waiter20);

queue.waitForAccess(waiter25);

queue.waitForAccess(waiter50);

queue.waitForAccess(waiter5);

**final** Map<KThread, Integer> pickFrequency = **new** HashMap<KThread, Integer>();

pickFrequency.put(waiter20, 0);

pickFrequency.put(waiter25, 0);

pickFrequency.put(waiter50, 0);

pickFrequency.put(waiter5, 0);

System.*out*.println("Running priority scheduler pickNextThreadTest for " + NUM\_ITERATIONS + " iterations.");

System.*out*.println("This will take a second or two....");

**for** (**int** i = 0; i < NUM\_ITERATIONS; ++i) {

**final** KThread wasPicked = queue.nextThread();

Lib.*assertTrue*(wasPicked != **null**);

pickFrequency.put(wasPicked, pickFrequency.get(wasPicked) + 1);

queue.waitForAccess(wasPicked);

}

**try** {

// Testing nondeterministic code is FUN!!!

**for** (**final** Map.Entry<KThread, Integer> e : pickFrequency.entrySet()) {

**final** **int** expectedPicks = (**int**) (((**double**) s.getPriority(e.getKey()) / 100.0) \* NUM\_ITERATIONS);

**final** **int** actualPicks = e.getValue();

Lib.*assertTrue*(Math.*abs*(actualPicks - expectedPicks) < (TOLERANCE \* NUM\_ITERATIONS));

}

} **catch** (**final** Exception e) {

System.*err*.println("Priority scheduler pickNextThreadTest failed!");

System.*exit*(-1);

}

System.*out*.println("Actual values are within " + TOLERANCE + "% of expected!");

System.*out*.println("Priority scheduler pickNextThreadTest passed successfully!");

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

}

**public** **static** **void** simplePriorityDonationTest() {

**final** Scheduler s = **new** LotteryScheduler();

**final** **boolean** intStatus = Machine.*interrupt*().disable();

System.*out*.println("Running simple test of LotteryScheduler priority donation...");

**int** totalPriority = 0;

List<KThread> threadList = **new** ArrayList<KThread>();

**for** (**int** i = 10; i < 100; i+=10) {

totalPriority += i;

KThread cur = **new** KThread(**null**);

s.setPriority(cur, i);

threadList.add(cur);

}

**final** ThreadQueue queue = s.newThreadQueue(**true**);

**for** (**final** KThread thread : threadList) {

queue.waitForAccess(thread);

}

**final** KThread resourceHolder = **new** KThread(**null**);

**final** **int** initialPriority = s.getEffectivePriority(resourceHolder);

queue.acquire(resourceHolder);

**try** {

Lib.*assertTrue*(s.getEffectivePriority(resourceHolder) - initialPriority == totalPriority);

} **catch** (Exception e) {

System.*err*.println("Simple priority donation test failed!");

System.*exit*(-1);

}

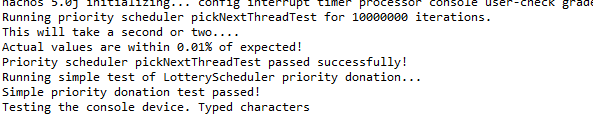
System.*out*.println("Simple priority donation test passed!");

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

}

}

测试结果:



### 总结

没有太多的参考资料。虽然通过google,在网上也能找到一些前届学生的作业，但他们全是c的，由于设计思路并不合我及组员的风格（比如说虚拟内存相关的试验项目），所以只能通过理解书本知识自己设计和实现。所以，费时费力，进度缓慢。经常是连续的熬夜编程和调式，才刚刚获得一点点的收获（在做第二部分试验的时候，曾经经历了好几轮的奋战1天的情况！）

收获及感想

TASK2是本门课程的第二大部分的试验项目。个人感觉，TASK2跟之前的TASK1的难度基本上是难不少的。 通过自己与小组将近2个多星期的努力，终于完成里面的内容。感觉收获不小，至少对多线程运行，页表，内存地址转化机制，虚拟内存实现机制这4方面，都有了深刻的理解和认识。  
 1.系统调用这部分的试验是相对较简单的，也是比较容易理解的。实现它们并不是很难的事情，主要是弄懂虚拟寄存器的在系统调用过程中的使用，而文件描述符让我确实看到了它们对实现文件系统的简易性。  
 2.虚拟到物理地址的内存转化机制，这个在nachos里实现起来真是有些让人头疼，因为需要计算偏移量，还需要跟虚拟机的内存下标，还有 MIPS COFF format格式文件的数据结构都有着密切的联系，把它们之间的关系准确的搞清楚真的不是一件轻松容易的事情。  
 此外，调试技能在完成作业上扮演了很重要的角色。很多代码，在设计之初我都不敢想想会最终得以实现，虽然最终只写了个简单的homework.c程序，却成功验证了6个系统调用及文件载入等方法.....因为我之前没有类似领域的编程开发经历，也没有可以阅读的demo代码，完全是通过反复的理解文档和书本的基本理论，然后运用”调式技能”这一把得力的工具，才把试验坚持下来的。俗话说的好，搞软件研发，3分写，7分调。很多代码是通过反复的调试才成为产品的。尤其是在不是很熟练的领域上，调试就扮演了决定性的作用。给我印象最深的就是在刚开始这个实验的时候，线程总是莫名的中断。没办法，只好通过debug进行变量的跟踪，开始的跟踪还算比较容易，但是到了最后跟踪中断那部分的时候，简直可以用“噩梦”来形容：那么多的循环，一个一个的跟踪，到后来不知不觉的就跟丢了。。。。。。所以，nachos的入门时很难得，要搞清它的初始化过程及其的复杂。虽然没有完全弄清，但是通过这样的疯狂跟踪，至少让我体验到了那些搞底层开发或者是嵌入式开发的程序员们的炼狱般的工作深度和工作激情。如果有机会，我也会尝试一下从事这类性质的开发工作（尽管我目前已经是20岁的programmer 了，^\_^）  
  
对课程的意见和建议  
 OS课程设计进度还是合理，我们很满意。如果不是因为学校的考试，我相信还是能够在规定的时间内完成更多的试验的。