**作业as-6-22**

刘睿 SG13225047

1. chapter8-96

(1)使用锁和条件变量实现类

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class Bathroom

{

private Lock lock;

private Condition maleCond;

private Condition femaleCond;

private int maleAcq;

private int maleRel;

private int femaleAcq;

private int femaleRel;

private int maleWait;

private int femaleWait;

public static final int MALE = 1;

public static final int FEMALE = 2;

public Bathroom()

{

lock = new ReentrantLock();

maleCond = lock.newCondition();

femaleCond = lock.newCondition();

}

public void enterMale()

{

lock.lock();

try

{

maleWait ++;

while((femaleWait != femaleAcq && maleAcq != maleRel)

|| (femaleAcq != femaleRel))

{

maleCond.await();

}

maleAcq ++;

}catch(Exception e)

{

e.printStackTrace();

maleWait --;

maleCond.signalAll();

femaleCond.signalAll();

}finally

{

lock.unlock();

}

}

public void leaveMale()

{

lock.lock();

maleRel ++;

if(maleRel == maleAcq)

{

femaleCond.signalAll();

}

lock.unlock();

}

public void enterFemale()

{

lock.lock();

try

{

femaleWait ++;

while((maleWait != maleAcq && femaleAcq != femaleRel)

|| (maleAcq != maleRel))

{

femaleCond.await();

}

femaleAcq ++;

}catch(Exception e)

{

e.printStackTrace();

femaleWait --;

maleCond.signalAll();

femaleCond.signalAll();

}finally

{

lock.unlock();

}

}

public void leaveFemale()

{

lock.lock();

femaleRel ++;

if(femaleRel == femaleAcq)

{

maleCond.signalAll();

}

lock.unlock();

}

}

通过锁lock()和unlock()来保证互斥，根绝男女性的等待数和释放数作为判断条件，通过signalAll()来避免线程出现饥饿。

(2) 通过synchronized、wait()、notify()和notifyAll()实现类

import java.util.Random;

class Bath

{

int males;

boolean male;

int females;

boolean female;

public Bath()

{

this.male=false;

this.males=0;

this.female=false;

this.females=0;

}

public synchronized void enterMale()

{

System.out.println("Male wait");

while(female==true)

{

try

{

this.wait();

} catch (InterruptedException e)

{

e.printStackTrace();

}

}

System.out.println("Male in");

this.male=true;

this.males++;

}

public synchronized void leaveMale()

{

this.males--;

System.out.println("Male out");

if(this.males==0)

{

this.male=false;

this.notifyAll();

}

}

public synchronized void enterFemale()

{

System.out.println("Female wait");

while(male==true)

{

try

{

this.wait();

} catch (InterruptedException e)

{

e.printStackTrace();

}

}

System.out.println("Female in");

this.female=true;

this.females++;

}

public synchronized void leaveFemale()

{

this.females--;

System.out.println("Female out");

if(this.females==0)

{

this.female=false;

this.notifyAll();

}

}

public void takeABath()

{

if(this.female==true)

{

System.out.println(this.females+" females"+" bath");

}

else if(this.male==true)

{

System.out.println(this.males+" males"+" bath");

}

}

}

class ThreadM extends Thread

{

Bath bath;

public ThreadM(Bath bath)

{

this.bath=bath;

}

public void run()

{

int number = new Random().nextInt(100) + 1;

bath.enterMale();

bath.takeABath();

try

{

sleep(number);

} catch (InterruptedException e)

{

e.printStackTrace();

}

bath.leaveMale();

}

}

class ThreadF extends Thread

{

Bath bath;

public ThreadF(Bath bath)

{

this.bath=bath;

}

public void run()

{

int number = new Random().nextInt(200) + 1;

bath.enterFemale();

bath.takeABath();

try

{

sleep(number);

} catch (InterruptedException e)

{

e.printStackTrace();

}

bath.leaveFemale();

}

}

public class PublicBath

{

public static void main(String[] args)

{

Bath bath = new Bath();

for(int i=0;i<10;i++)

{

ThreadM tp1=new ThreadM(bath);

ThreadF tp2=new ThreadF(bath);

tp1.start();

tp2.start();

}

}

}

通过synchronized、wait()和notify()保证线程间的互斥访问，因为每个线程之间的竞争是公平的，同时再加上notifyAll()可以唤醒全部线程，避免了线程出现饥饿。

1. chapter8-97

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class Rooms

{

public interface Handler

{

void onEmpty();

}

private Lock lock;

private Condition[] conds;

private int waiting[];

private int acquire[];

private int release[];

private Handler[] handlers;

private final int m;

private int currRoom;

public Rooms(int m)

{

this.m = m;

lock = new ReentrantLock();

conds = new Condition[m + 1];

for(int i = 0; i < conds.length; i ++)

{

conds[i] = lock.newCondition();

}

waiting = new int[m + 1];

acquire = new int[m + 1];

release = new int[m + 1];

handlers = new Handler[m + 1];

currRoom = -1;

}

private boolean toWait(int index)

{

if(currRoom == -1)

{

return false;

}

if(currRoom != index)

{

return true;

}

for(int i = 1; i <= m; i ++)

{

int other = (i + index) % (m + 1);

if(waiting[other] != acquire[other])

{

return true;

}

}

return false;

}

public void enter(int i)

{

if(i < 0 || i > m)

{

return;

}

lock.lock();

try

{

waiting[i] ++;

while(toWait(i))

{

conds[i].await();

}

acquire[i] ++;

currRoom = i;

}catch(Exception e)

{

e.printStackTrace();

}finally

{

lock.unlock();

}

}

private int notifyWho()

{

for(int i = 1; i <= m; i ++)

{

int other = (i + currRoom) % (m + 1);

if(waiting[other] != acquire[other])

{

return other;

}

}

return -1;

}

public boolean exit()

{

lock.lock();

release[currRoom] ++;

if(release[currRoom] == acquire[currRoom])

{

if(handlers[currRoom] != null)

{

handlers[currRoom].onEmpty();

}

int other = notifyWho();

if(other >= 0)

{

conds[other].signalAll();

}

currRoom = -1;

}

lock.unlock();

return true;

}

public void setExitHandler(int i, Rooms.Handler handler)

{

if(i < 0 || i > m)

{

return;

}

lock.lock();

handlers[i] = handler;

lock.unlock();

}

}

1. Chapter9-101所有的锁都按照升序获得，不会有循环，所以无死锁。

Chapter9-108 常规的锁法之所以需要2个锁，是因为要确定pred->curr的关系，即确保pred->next在add/remove操作过程中不会改变。因为add操作只涉及到pred->next和一个将成为pred->next的新节点，所以add只需要锁pred。具体来说，考虑五种情况：

1. ThreadA.add && ThreadB.add

ThreadA.pred.lock ==> ThreadB waiting for pred.lock ==> ThreadA.pred.next = new, new->next = ThreadA.curr ==> ThreadB.pred.lock() ==> ThreadB.valiation ==> Thread.B.pred.next != ThreadB.curr

1. ThreadA.add && ThreadB.remove(pred)

ThreadA.pred.lock ==> ThreadB.pred.pred.lock(), ThreadB.pred waiting for lock ==> ThreadA.pred.next = new, Thread.new.next = curr ==> ThreadB.pred.lock() ==> ThreadB.pred.pred.next = new

1. ThreadA.add && ThreadB.remove(pred)

ThreadB.pred.pred.lock(), ThreadB.pred.lock() ==> ThreadA.pred waiting for lock ==> ThreadB.pred.pred.next = ThreadB.cur; ==> ThreadA.pred.lock() ==> ThreadA.validate ==> pred is not reachable

1. ThreadA.add && ThreadB.remove(curr)

ThreadA.pred.lock ==> ThreadB.pred waiting for lock ==> ThreadA.pred.next = new, ThreadA.pred.next.next = ThreadA.cur; ==> ThreadB.pred.lock() ==> ThreadB.validate ==> ThreadB.pred.next != ThreadB.cur

1. ThreadA.add && ThreadB.remove(curr)

ThreadB.pred.lock(), ThreadB.cur.lock() ==> ThreadA.pred waiting for lock ==> ThreadB.pred.next = ThreadB.cur.next ==> ThreadA.pred.lock() ==> ThreadA.validate ==> ThreadA.pred.next != ThreadA.curr