Assignment #7

Pongsathorn Cherngchaosil

Cesar Montelongo

CECS 327

Code:

import java.util.LinkedList;  
import java.util.Queue;  
import java.util.concurrent.CopyOnWriteArrayList;  
import java.util.concurrent.locks.Condition;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
public class Main{  
  
 private static int WAITING = 0, EATING = 1, THINKING = 2;  
 private static final int NUM\_PHILS = 20;  
 private static Condition[] phil = new Condition[NUM\_PHILS];  
 private static int[] states = new int[NUM\_PHILS];  
 private static Lock lock = new ReentrantLock();  
 private static Lock lock\_que = new ReentrantLock();  
 private static int[] count\_eat = new int[NUM\_PHILS];  
 private static CopyOnWriteArrayList<Philosopher> multArrayList = new CopyOnWriteArrayList<>();  
  
  
 public static void main(String[] args){  
 init();  
  
 Philosopher[] p = new Philosopher[NUM\_PHILS];  
 Queue<Philosopher> que = new LinkedList<Philosopher>();  
 for(int i = 0; i < p.length; i++){  
 p[i] = new Philosopher(lock, phil, states, NUM\_PHILS, multArrayList, count\_eat);  
 p[i].start();  
 }  
  
 for(int i = 0; i < p.length; i++){  
 try {  
 p[i].join();  
 } catch (InterruptedException e) {  
 // TODO Auto-generated catch block  
 e.printStackTrace();  
 }  
 }  
 int minEat = 1000000000;  
 System.out.println("Thread | Execution Time | Eat Count | Wait Count | Wait Time");  
 for(int i = 0; i < p.length; i++){  
  
 System.out.printf("%7d: %10.3f %10d %10d %15.3f\n", i,(float)p[i].getExtTime(), count\_eat[(int)p[i].getThreadID()], p[i].getWaitCount(), (float)p[i].getWaitTime());  
 }  
// System.out.println("Minimum number of eats : " + minEat);  
 }  
  
 public static void init(){  
 for(int i = 0; i < NUM\_PHILS; i++){  
 phil[i] = lock.newCondition();  
 states[i] = THINKING;  
 }  
 }  
}

----------------------------------------------------------------------------------------------------------------

import java.util.concurrent.locks.\*;  
import java.util.concurrent.CopyOnWriteArrayList;  
  
public class Philosopher extends Thread {  
 private static int WAITING = 0, EATING = 1, THINKING = 2;  
 private Lock lock;  
 private Condition phil[];  
 private int states[];  
 private int timeEated[];  
 private int NUM\_PHILS;  
 private int id;  
 private final int TURNS = 20;  
 private long waitTimes = 0;  
 private long executionTimes = 0;  
 private CopyOnWriteArrayList<Philosopher> waiting;  
 private int count\_stall;  
 boolean ableToEat;  
  
 public Philosopher(Lock l, Condition p[], int s[], int num, CopyOnWriteArrayList<Philosopher> arr, int[] eat) {  
 lock = l;  
 phil = p;  
 states = s;  
 NUM\_PHILS = num;  
 waiting = arr;  
 timeEated = eat;  
 count\_stall = 0;  
 }  
  
 public void run() {  
 long time = System.nanoTime(); // Time for phil to execute  
 id = ThreadID.get();  
 for (int k = 0; k < TURNS; k++) {  
 try {  
 sleep(100);  
 } catch (Exception ex) {  
 /\* lazy \*/  
 }  
 takeSticks(id);  
 try {  
 sleep(20);  
 } catch (Exception ex) {  
  
 }  
 putSticks(id);  
 }  
 executionTimes = System.nanoTime() - time;  
  
 }// end run  
  
 public void takeSticks(int id) {  
 long time = System.nanoTime();  
 lock.lock();  
 waitTimes += System.nanoTime() - time;  
 try {  
 ableToEat = true; // condition for checking to see if waiting  
 // threads are able to eat  
 if (!waiting.isEmpty()) {// check if queue is null  
 for (int i = 0; i < waiting.size(); i++) {// waiting threads  
 int temp = waiting.get(i).getID();  
 if (id == leftof(temp) || id == rightof(temp)) {  
 // if either side is waiting, thread does not proceed  
 ableToEat = false;  
 break;  
 } // end if  
 } // end for  
 } // end if  
  
 // protocol so that threads cannot bypass other threads in the queue  
 if (states[leftof(id)] == THINKING && states[rightof(id)] == THINKING && ableToEat) {  
 states[id] = EATING;  
 timeEated[id]++;  
 } else {  
 count\_stall++;  
 states[id] = WAITING;  
 waiting.add(this); // will add thread in the queue  
 time = System.nanoTime();  
 // philosopher is waiting for status update  
 // await is a synch method to make it starvation free  
 phil[id].await(); // will set thread to wait  
 waitTimes += System.nanoTime() - time;  
 }  
 } // end try block  
 catch (InterruptedException e) {  
 System.exit(-1);  
 } finally {  
 lock.unlock();  
 }  
 }// end takeSticks  
  
 public void putSticks(int id) {  
 long time = System.nanoTime();  
 lock.lock();  
 waitTimes += System.nanoTime() - time;  
 try {  
 states[id] = THINKING;  
 // will check the queue to see which thread can start anytime and  
 // which thread stops eating goes at the front of the queue  
 // to give priority to the threads that has waited longer  
 if (!waiting.isEmpty()) {  
 for (int i = 0; i < waiting.size(); i++) {  
 Philosopher p = waiting.get(i);  
 // thread came from list = ignores if others are waiting  
 if (states[leftof(p.getID())] != EATING && states[rightof(p.getID())] != EATING) {  
 phil[p.getID()].signal();// Signal an awaiting phil  
 states[p.getID()] = EATING;  
 timeEated[p.getID()]++;  
 waiting.remove(i);  
 }  
 }  
 }  
 }  
 finally {  
 lock.unlock();  
 }  
 }  
  
 private int leftof(int id) { // clockwise  
 int retval = id - 1;  
 if (retval < 0) // not valid id  
 retval = NUM\_PHILS - 1;  
 return retval;  
 }  
  
 private int rightof(int id) {  
 int retval = id + 1;  
 if (retval == NUM\_PHILS) // not valid id  
 retval = 0;  
 return retval;  
 }  
  
 public int getID() {  
 return id;  
 }  
  
 public long getWaitTime(){  
 return waitTimes;  
 }  
  
 public long getExtTime(){  
 return executionTimes;  
 }  
  
 public int getWaitCount(){  
 return count\_stall;  
 }  
  
 public int getThreadID(){  
 return id;  
 }  
}

----------------------------------------------------------------------------------------------------------------

public class ThreadID {  
  
 private static volatile int nextID = 0;  
 private static ThreadLocalID threadID = new ThreadLocalID();  
  
 public static int get() {  
 return threadID.get();  
 }  
  
 public static void set (int index) {  
 threadID.set(index);  
 }  
  
  
 private static class ThreadLocalID extends ThreadLocal<Integer> {  
  
 protected synchronized Integer initialValue () {  
 return nextID++;  
 }  
 }  
}

Output :

Thread | Execution Time | Eat Count | Wait Count | Wait Time  
 0: 2477640448.000 20 5 25686972.000  
 1: 2453518592.000 20 0 1519885.000  
 2: 2531321600.000 20 9 84116760.000  
 3: 2477452544.000 20 1 24370224.000  
 4: 2524081664.000 20 1 64034608.000  
 5: 2477301248.000 20 1 24794884.000  
 6: 2523905536.000 20 7 61171044.000  
 7: 2487962112.000 20 1 26551258.000  
 8: 2523861504.000 20 5 60172476.000  
 9: 2458261760.000 20 0 5279381.000  
  
Thread | Execution Time | Eat Count | Wait Count | Wait Time  
 0: 2492063744.000 20 1 26287398.000  
 1: 2473070592.000 20 0 3243746.000  
 2: 2506756608.000 20 1 28231936.000  
 3: 2465769984.000 20 0 3640187.000  
 4: 2554059264.000 20 11 87830496.000  
 5: 2465707264.000 20 0 4604114.000  
 6: 2537098752.000 20 3 68339792.000  
 7: 2533493504.000 20 12 67390400.000  
 8: 2506159616.000 20 1 26192028.000  
 9: 2496451072.000 20 1 29285240.000  
 10: 2551302400.000 20 1 69971712.000  
 11: 2495831040.000 20 2 29632920.000  
 12: 2464433152.000 20 0 2176326.000  
 13: 2495267840.000 20 1 29125296.000  
 14: 2531779840.000 20 7 61166764.000  
 15: 2470344192.000 20 0 3869827.000  
 16: 2494110976.000 20 3 30333420.000  
 17: 2462799104.000 20 0 4367617.000  
 18: 2533909248.000 20 5 70689320.000  
 19: 2503127296.000 20 1 27679404.000  
 20: 2530082048.000 20 9 62167044.000  
 21: 2462006272.000 20 0 2609967.000  
 22: 2529868288.000 20 9 60347812.000  
 23: 2529517568.000 20 6 56519884.000  
 24: 2492685568.000 20 1 24058468.000  
  
Thread | Execution Time | Eat Count | Wait Count | Wait Time  
 0: 2483201536.000 20 3 31920016.000  
 1: 2522700800.000 20 11 60988440.000  
 2: 2501432064.000 20 4 29935270.000  
 3: 2458456832.000 20 0 3246309.000  
 4: 2482758912.000 20 3 29229220.000  
 5: 2458319360.000 20 0 1666991.000  
 6: 2501101824.000 20 2 29867278.000  
 7: 2511263232.000 20 7 50608872.000  
 8: 2500633600.000 20 4 31901624.000  
 9: 2468629760.000 20 0 2485951.000  
 10: 2511100928.000 20 5 55625244.000  
 11: 2521963264.000 20 8 69811344.000  
 12: 2457772800.000 20 0 1517743.000  
 13: 2510923264.000 20 4 52572224.000  
 14: 2468256768.000 20 0 3220655.000  
 15: 2482184960.000 20 1 27182902.000  
 16: 2482063104.000 20 1 27865000.000  
 17: 2521524736.000 20 11 70639704.000  
 18: 2500292352.000 20 5 34580864.000  
 19: 2467916800.000 20 0 5083938.000  
 20: 2503350016.000 20 5 48508236.000  
 21: 2532267520.000 20 4 67964312.000  
 22: 2467727872.000 20 0 2177178.000  
 23: 2499783936.000 20 4 35882648.000  
 24: 2456983040.000 20 0 3313454.000  
 25: 2499859456.000 20 2 32639324.000  
 26: 2467463424.000 20 0 3309611.000  
 27: 2502865408.000 20 2 47215876.000  
 28: 2456719104.000 20 0 3162498.000  
 29: 2509859840.000 20 6 57623668.000  
 30: 2481177600.000 20 1 27919748.000  
 31: 2535086592.000 20 8 75766832.000  
 32: 2509664256.000 20 6 52601304.000  
 33: 2480952064.000 20 1 25645916.000  
 34: 2509478400.000 20 6 54672860.000  
 35: 2466808320.000 20 0 1864995.000  
 36: 2502257920.000 20 4 45670336.000  
 37: 2477301248.000 20 0 2641179.000  
 38: 2498861056.000 20 10 33113598.000  
 39: 2466525696.000 20 0 2010397.000  
 40: 2520101632.000 20 10 59671264.000  
 41: 2509042432.000 20 3 45612172.000  
 42: 2480450048.000 20 1 22240516.000  
 43: 2505349376.000 20 3 46021012.000  
 44: 2476720128.000 20 0 2083522.000  
 45: 2508791552.000 20 7 52791184.000  
 `46: 2480162560.000 20 1 23671872.000  
 47: 2508690944.000 20 5 52319480.000  
 48: 2466018048.000 20 0 1254307.000  
 49: 2508686336.000 20 5 53197456.000  
  
Thread | Execution Time | Eat Count | Wait Count | Wait Time  
 0: 2446123264.000 20 0 3311322.000  
 1: 2517838848.000 20 2 55180480.000  
 2: 2445964544.000 20 0 1619944.000  
 3: 2477947904.000 20 4 28283676.000  
 4: 2524641024.000 20 3 54079704.000  
 5: 2488213760.000 20 1 23867740.000  
 6: 2488181504.000 20 1 27911190.000  
 7: 2513487616.000 20 7 59820512.000  
 8: 2517794048.000 20 4 55437496.000  
 9: 2487946496.000 20 1 26634652.000  
 10: 2445431808.000 20 0 3428064.000  
 11: 2472715008.000 20 0 9652982.000  
 12: 2505491712.000 20 4 35995120.000  
 13: 2501910016.000 20 5 37754052.000  
 14: 2525934848.000 20 6 56982184.000  
 15: 2473201664.000 20 0 10653260.000  
 16: 2513468672.000 20 3 51961536.000  
 17: 2535126784.000 20 5 59720016.000  
 18: 2504895232.000 20 1 32967334.000  
 19: 2472666368.000 20 0 11365305.000  
 20: 2471733760.000 20 0 8917414.000  
 21: 2455166464.000 20 0 998565.000  
 22: 2501325824.000 20 1 35993408.000  
 23: 2504785152.000 20 5 38641860.000  
 24: 2471577088.000 20 0 9079495.000  
 25: 2497633280.000 20 11 41097448.000  
 26: 2471485184.000 20 0 3807394.000  
 27: 2501057024.000 20 4 39854256.000  
 28: 2471380480.000 20 0 8112577.000  
 29: 2504413952.000 20 4 36362044.000  
 30: 2471411968.000 20 0 9067523.000  
 31: 2534479360.000 20 7 61644880.000  
 32: 2504286208.000 20 5 37439304.000  
 33: 2471073024.000 20 0 7022062.000  
 34: 2515716096.000 20 1 49517928.000  
 35: 2471532288.000 20 1 24937724.000  
 36: 2516105216.000 20 1 48114800.000  
 37: 2470867712.000 20 0 10324827.000  
 38: 2503888384.000 20 4 36431752.000  
 39: 2534081280.000 20 6 67333520.000  
 40: 2471728128.000 20 1 23456334.000  
 41: 2515412736.000 20 1 48629696.000  
 42: 2454119424.000 20 0 1227359.000  
 43: 2500172544.000 20 8 42894440.000  
 44: 2470908672.000 20 0 6444298.000  
 45: 2503951616.000 20 4 37384132.000  
 46: 2470467328.000 20 0 6897187.000  
 47: 2499970304.000 20 9 44715824.000  
 48: 2453750528.000 20 0 1375765.000  
 49: 2511771392.000 20 3 53643496.000  
 50: 2485670400.000 20 2 27424100.000  
 51: 2453587968.000 20 0 1104203.000  
 52: 2477759232.000 20 1 21475864.000  
 53: 2510856704.000 20 1 49553424.000  
 54: 2510751744.000 20 1 49141592.000  
 55: 2452588544.000 20 0 1567350.000  
 56: 2484489984.000 20 1 22692976.000  
 57: 2513756160.000 20 3 52288692.000  
 58: 2477320192.000 20 0 11026176.000  
 59: 2510224640.000 20 1 41749188.000  
 60: 2469767168.000 20 1 19421004.000  
 61: 2513478656.000 20 1 42682752.000  
 62: 2477104896.000 20 0 11192533.000  
 63: 2513517824.000 20 1 43335348.000  
 64: 2468487424.000 20 1 21374952.000  
 65: 2509347072.000 20 1 44242836.000  
 66: 2468551168.000 20 0 10329098.000  
 67: 2501428224.000 20 5 38229600.000  
 68: 2531515136.000 20 5 61288648.000  
 69: 2468740608.000 20 0 10733658.000  
 70: 2501239552.000 20 4 40150624.000  
 71: 2531362048.000 20 7 60135696.000  
 72: 2468469760.000 20 1 19978234.000  
 73: 2508742400.000 20 2 43164288.000  
 74: 2468360704.000 20 0 13338920.000  
 75: 2500985600.000 20 7 41435284.000  
 76: 2472658688.000 20 0 9554194.000  
 77: 2508773632.000 20 2 43116820.000  
 78: 2467671040.000 20 1 20374240.000  
 79: 2508628992.000 20 1 44829568.000  
 80: 2467977984.000 20 1 19691274.000  
 81: 2508188928.000 20 1 44034136.000  
 82: 2467372800.000 20 1 18341176.000  
 83: 2508506112.000 20 2 40769428.000  
 84: 2467159808.000 20 0 9123551.000  
 85: 2507954176.000 20 1 45320088.000  
 86: 2467603712.000 20 1 20310948.000  
 87: 2511871488.000 20 1 44352308.000  
 88: 2466904064.000 20 0 11014202.000  
 89: 2471807744.000 20 1 20804452.000  
 90: 2511546112.000 20 1 43549612.000  
 91: 2511724800.000 20 1 44738052.000  
 92: 2467643904.000 20 1 21518208.000  
 93: 2511326976.000 20 1 44333920.000  
 94: 2466597888.000 20 0 7344514.000  
 95: 2499706112.000 20 5 32141960.000  
 96: 2467370240.000 20 0 8304163.000  
 97: 2507363584.000 20 5 50015720.000  
 98: 2481874176.000 20 1 20972954.000  
 99: 2511121664.000 20 3 50826112.000

Report:

Our assignment does exhibit bottlenecking because there was only one lock being used in its implementation, so whenever someone tries to eat or put down their sticks they lock down the whole table. If there are no one in the waitlist, check both side for eating, then eat. Else put yourself in queue. If there are people in the waitlist while trying to eat, check your position if you are next to anyone in the list because if you start eating, they will not be able to eat when it is their turn to check if they can pick up the sticks. When you put your stick down, go through the waitlist and allow those they can eat to eat. Our solution scales well in terms of no starvation, but not when it comes to bottlenecks because again, we only use one lock.