**Logbook Part 3**

Contents

[Week 13 2](#_Toc1472334)

[Self Assessment 4](#_Toc1472335)

[Week 14 5](#_Toc1472336)

[Additional Material. 5](#_Toc1472337)

[Self-Assessment 8](#_Toc1472338)

[Week 15 8](#_Toc1472339)

[Self Assessment 10](#_Toc1472340)

[Week 16 11](#_Toc1472341)

[Self assessment 13](#_Toc1472342)

# Week 13

This is the code added for the Run() setupRun() and main() methods respectively.

*/\*\*  
 \* Run this counter.  
 \*/*public void run() {  
 // Implement run() here.  
 // This should run the counter, by initialising the count, then, as long as

// the counter is not  
 // finished, step the counter.  
 // Use the methods defined above to implement this behaviour.  
 this.startCount();  
 while(!this.isFinished()){  
 this.stepCount();  
 }  
}

*/\*\*  
 \* Run all the threads in this thread set in parallel, and wait for them to finish.  
 \** ***@throws*** *InterruptedException if an interrupt occurs while waiting for*

*\* the threads to finish.  
 \*/*@Override  
public void runSet() throws InterruptedException {  
 // Implement runSet here.  
 // runSet() should start up all the threads in this set, and then wait for

// them to finish.  
 for (Thread thread : this){  
 thread.start();  
 }  
 for (Thread thread : this){  
 thread.join();  
 }  
}

*/\*\*  
 \* Demonstrate the behaviour of counters and ThreadSets. A thread set is populated with two counters, and  
 \* the thread set's runSet method is used to run the counters concurrently.  
 \*  
 \** ***@param*** *args not used  
 \** ***@throws*** *CounterException should not occur  
 \** ***@throws*** *InterruptedException should not occur  
 \*/* public static void main(String[] args) throws CounterException,

InterruptedException {  
 /\*  
 \* Create two counters (in a thread set), and then run them with tracing on,

\* so that their  
 \* behaviour is visible.  
 \*/  
 ThreadSet<Counter> counters = new ThreadHashSet<>();   
 counters.add(new Counter("up",0,10)); // counter "up" counts from 5 to 10  
 counters.add(new Counter("down",10,0)); // counter "down" counts from 5 to 0  
 Counter.*traceOn*(); // switch tracing on  
 Counter.*setDelay*(0.1); // set a delay from 0.0 to 0.1 seconds  
 counters.runSet(); // run the counters (concurrently)  
 }

1. Will the test always terminate? I.e. is it certain that no matter how often

you were to run the test it would always end in a finite length of time?

Based on my program. I believe it is possible, but highly unlikely that the program will not have an end to the processing time. This is because the vast majority of the time the first thread starts, the second sets the counter to the end of the first, and the first ends (E.G. Up starts and sets Count to 0. Down starts and sets Count to 10. Up checks count and terminates as it is already 10. Down then counts down to 0 and terminates.). There are times where a tug of war can continue for a few seconds at most but these are uncommon themselves. For an indefinite runtime to occur, it would be nearly impossible to replicate and just as hard to identify (if you close it now, was it going to end in another few cycles?).

Some of my executions lasted 14 lines, some lasted several thousand lines and ran for a minute or two. This difference makes this question almost impossible to answer definitively.

2. What is the shortest possible output for the test, in terms of the number

of lines output?

14 lines is the smallest possible output (0-10). These are:

*up has started: 0*

*down has started: 10*

*up has finished: 10*

*down has stepped: 9*

*down has stepped: 8*

*down has stepped: 7*

*down has stepped: 6*

*down has stepped: 5*

*down has stepped: 4*

*down has stepped: 3*

*down has stepped: 2*

*down has stepped: 1*

*down has stepped: 0*

*down has finished: 0*

or:

*down has started: 10*

*up has started: 0*

*down has finished: 0*

*up has stepped: 1*

*up has stepped: 2*

*up has stepped: 3*

*up has stepped: 4*

*up has stepped: 5*

*up has stepped: 6*

*up has stepped: 7*

*up has stepped: 8*

*up has stepped: 9*

*up has stepped: 10*

*up has finished: 10*

3. What is the largest possible value that the count can reach when the test

is run?

11. The test can reach 10 and add another sum (11) before realising it is too high and terminating.

4. What is the lowest possible value that the count can reach when the test

is run?

-1. The test can reach 0 and remove another sum (-1) before realising it is too low and terminating.

## Self Assessment

3/5

I would give myself 3 marks for this work as I have demonstrated the use of my code and believe I have demonstrated the outcome from multiple iterations of run time, giving detailed responses in relation to the running of the program.

# Week 14

I have implemented Dekker’s algorithm within the given code bundle.

For Peru:

*/\*\*  
 \* Shared basket. If 0 - Bolivia has priority. If 1 - Peru has priority.  
 \*/* public void runTrain() throws RailwaySystemError {  
 Clock clock = getRailwaySystem().getClock();  
 Railway otherRailway = getRailwaySystem().getNextRailway(this);  
 while (!clock.timeOut()) {  
 this.getBasket().putStone(this);  
 choochoo();  
 while (otherRailway.getBasket().hasStone(this)){  
 if (!*getSharedBasket*().hasStone(this)){ // If Peru has priority.  
 this.getBasket().takeStone(this);  
 while (!*getSharedBasket*().hasStone(this));  
 this.getBasket().putStone(this);  
 }  
 }  
 crossPass();  
 this.getBasket().takeStone(this);  
 *getSharedBasket*().takeStone(this);  
 }  
 }

For Bolivia:

*/\*\*  
 \* Shared basket. If 0 - Bolivia has priority. If 1 - Peru has priority.  
 \*/*public void runTrain() throws RailwaySystemError {  
 Clock clock = getRailwaySystem().getClock();  
 Railway otherRailway = getRailwaySystem().getNextRailway(this);  
 while (!clock.timeOut()) {  
 this.getBasket().putStone(this);  
 choochoo();  
 while (otherRailway.getBasket().hasStone(this)){  
 if (*getSharedBasket*().hasStone(this)){ // If Bolivia has priority.  
 this.getBasket().takeStone(this);  
 while (*getSharedBasket*().hasStone(this));  
 this.getBasket().putStone(this);  
 }  
 }  
 crossPass();  
 this.getBasket().takeStone(this);  
 *getSharedBasket*().putStone(this);  
 }  
}

## Additional Material.

As the current implementation uses a Boolean, I decided to add another railway – Chile. Chile has the same requirements as Peru and Bolivia, and the train drivers for Chile are also equally impaired as their Peruvian and Bolivian counterparts.

I was required to change the basket mechanisms to return Int values rather than Booleans, which resulted in this addition.

*/\*\*  
 \* Check if the basket contains at least one stone.  
 \** ***@return*** *true if there is at least one stone in this basket.  
 \*/*private synchronized int stoneCount() {  
 return stones;  
}  
  
*/\*\*  
 \* Check if the basket contains at least one stone.  
 \* This is the method called by a railway, and includes a delay and a trace.  
 \** ***@param*** *railway the railway checking this basket.  
 \** ***@return*** *true if there is at least one stone in this basket.  
 \** ***@throws*** *ProgrammingError if the specified railway is not registered with*

*\* a railway system.  
 \*/*public int stoneCount(Railway railway) throws ProgrammingError {  
 railway.getRailwaySystem().trace(railway.name() + ": checking " + name +

" for stones");  
 railway.delay();  
 return stoneCount();  
}

hasStone() is still within the class above as it is still necessary to check if a basket is empty or not for the individual basket for the next railway.

I implemented a new Chile class:

package ferrocarrilesDeAmericaDelSur.railways;  
  
import ferrocarrilesDeAmericaDelSur.errors.RailwaySystemError;  
import ferrocarrilesDeAmericaDelSur.errors.SetUpError;  
import ferrocarrilesDeAmericaDelSur.tools.Clock;  
import ferrocarrilesDeAmericaDelSur.tools.Delay;  
  
*/\*\*  
 \* An implementation of a railway. The runTrain method, should, in*

*\* collaboration with Peru's runTrain(), guarantee  
 \* safe joint operation of the railways.  
 \*/*public class Chile extends Railway {  
 */\*\*  
 \* Change the parameters of the Delay constructor in the call of*

*\* the superconstructor to  
 \* change the behaviour of this railway.  
 \** ***@throws*** *SetUpError if there is an error in setting up the delay.  
 \*/* public Chile() throws SetUpError {  
 super("Chile",new Delay(0.1,0.3));  
 }

*/\*\*  
 \* Shared basket. If 0 - Bolivia has priority.*

*\* If 1 - Peru has priority.*

*\* If 2- Chile has priority.  
 \*/* public void runTrain() throws RailwaySystemError {  
 Clock clock = getRailwaySystem().getClock();  
 Railway otherRailway = getRailwaySystem().getNextRailway(this);  
 while (!clock.timeOut()) {  
 this.getBasket().putStone(this);  
 choochoo();  
 while (otherRailway.getBasket().hasStone(this)){  
 if (*getSharedBasket*().stoneCount(this) == 2){ // If Chile   
 this.getBasket().takeStone(this); has priority.  
 while (*getSharedBasket*().hasStone(this));  
 this.getBasket().putStone(this);  
 }  
 }  
 crossPass();  
 this.getBasket().takeStone(this);  
 *getSharedBasket*().takeStone(this); // 2 stones need to be remove   
 *getSharedBasket*().takeStone(this); as Peru is next.  
 }  
 }  
}

This is the same inner workings as before, however the check for whether the current railway has priority has been changed to check for an integer rather than a Boolean as there are now more than 2 railways.

With this current implementation, this issue arose:

*“Bolivia: adding stone to shared basket (34 stones in the basket)”*

I removed the twin lines of getSharedBasket().takeStone(this); and replaced it with a put. I also made this change to the putStone() method.

private synchronized void putStone() {  
 stones++;  
 stones = stones % 3;  
}

This aims to make sure than there are never too many stones in the basket, that no train driver ever takes from the shared basket, although this is now more like Pythagoras’ cup / Greedy cup – a volume of water is added to a cup, if too much is added then all the water drains out. This would require a line on the cup and not fit the prompt but neither does a Chilean railway.

This alteration also improves the code’s integrity as no stone will be removed from an empty basket. If a railway has it’s turn skipped the program can deal with it without an error, rather than an exception being thrown.

This new implementation runs perfectly (actually, it runs better than it did with the put and take system from before).

*Peru: removing stone from Peru's basket (1 stone in the basket)*

*Bolivia: checking shared basket for stones*

*Chile: checking shared basket for stones*

*Bolivia: checking shared basket for stones*

*Peru: adding stone to shared basket (2 stones in the basket)*

*Chile: checking shared basket for stones*

*Peru: adding stone to Peru's basket (0 stones in the basket)*

*Bolivia: adding stone to Bolivia's basket (0 stones in the basket)*

*Peru: choo-choo*

## Self-Assessment

4/5

I would give myself 4 marks for this weeks work as I have created a correct implementation using the given code bundle, and extended and (in my opinion) improved it with the addition of a third railway and the new output appears to run better than the original did.

# Week 15

**Logbook question.**

*In the lecture it was said that in the implementation of bounded buffers using semaphores (see figure 1 on page 3)*

*the order of the criticalSection.P() and noOfElements.P(), in the Buffer class's get method, was essential, but that the order of criticalSection.V() noOfElements.V(), in the same class's put() item method, was not. Identify the corresponding piece of code in the Buffer class provided and make the change. Can you*

*produce an error situation?*

*Note:*

*You may see \error" messages about attempts to access a closed buffer. This is not the error you are looking for.*

* This is the original get() method.

public T get() throws BufferError, SemaphoreLimitError {  
 T item;  
 try {  
 noOfElements.poll(); // is there at least one data item in the buffer?  
 criticalSection.poll(); // is the buffer available?  
 item = getItem(); // add the data item  
 criticalSection.vote(); // make the buffer available again  
 noOfSpaces.vote(); // there is now one more space in the buffer  
 } catch (InterruptedException ie) {  
 throw new BufferError(

"Buffer: Data item could not be retrieved from the buffer.\n" +  
 "\t" + ie.getMessage());  
 }  
 return item;  
}

* This is the modified get() method.

public T get() throws BufferError, SemaphoreLimitError {  
 T item;  
 try {  
 criticalSection.poll(); // is the buffer available?  
 noOfElements.poll(); // is there at least one data item in the buffer?  
 item = getItem(); // add the data item  
 criticalSection.vote(); // make the buffer available again  
 noOfSpaces.vote(); // there is now one more space in the buffer  
 } catch (InterruptedException ie) {  
 throw new BufferError("Buffer: Data item could not be retrieved from

the buffer.\n" +  
 "\t" + ie.getMessage());  
 }  
 return item;  
}

* This is the error caused by this change.

\*\*\*System will run for 20.0s.

\*\*\*Buffer can hold up to 10 elements.

\*\*\*The producer will take up to 1.0s between each put() to the buffer.

\*\*\*The consumer will take up to 1.0s between each get() from the buffer.

Consumer: trying to retrieve data item.

Producer: trying to add 0 to the buffer.

\*\*\*Run time ended. Closing the buffer. Starting to terminate the system.

Consumer: has finished.

Producer: has finished.

\*\*\*System terminated.

Process finished with exit code 0

**Logbook question.**

*Why does the error situation arise when the code is changed as described in question 1? Why does it not arise in the original code?*

* Previously (pre-adjustment) the code is in the correct order. This checks that there is at least one item of data in the buffer before trying to check whether the buffer is available. Now (swapped) the semaphore that uses the get() method will take the critical section to check that it is available, but may not have any data within it to complete the process and then release the critical section. This causes Deadlock as the process is unable to finish until data is put in, but data cannot be put in by the producer as the buffer is locked by the consumer.

**Logbook question.**

*Is the order of the calls of P() in the Buffer class's put method also essential*

* I believe that with a buffer of this size that initialises empty, it is not essential to have the lines of code in one particular order. If the buffer is available it will (almost) always be guaranteed to have space within it as the consumer has just released it and took an item from the buffer. The few times where the speed of the producer adds two items in a row is counteracted by the frequency of the consumer doing the same. As the buffer starts empty, this is not a problem for this method but is for the get() method as it starts with a 50/50 chance to begin in this error state.

Although this is not essential, it is ideal as it is more consistent throughout the code, and should this issue occur, then the program is more equipped to deal with it.

## Self Assessment

3/5

I would give myself 3 marks for this weeks work as I believe I have correctly identified deadlocks and given details about them, but feel like I omitted some important information that would have been enough to warrant 4 or even 5 marks for this work.

# Week 16

Oh boy….

I made some attempt at the required exercises.

Here is the LockResourceManager that I attempted to create.

package resourceManager;  
  
  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.concurrent.locks.Condition;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
public class LockResourceManager extends BasicResourceManager {  
  
 final Lock lock = new ReentrantLock();  
 final Condition notFree = lock.newCondition();  
 private ArrayList<Integer> buffer;  
 boolean free = true;  
  
 */\*\*  
 \*  
 \* Each resource manager has a resource.  
 \*/  
  
  
 /\*\*  
 \* Set the resource and initialise the numbers of waiting processes, and*

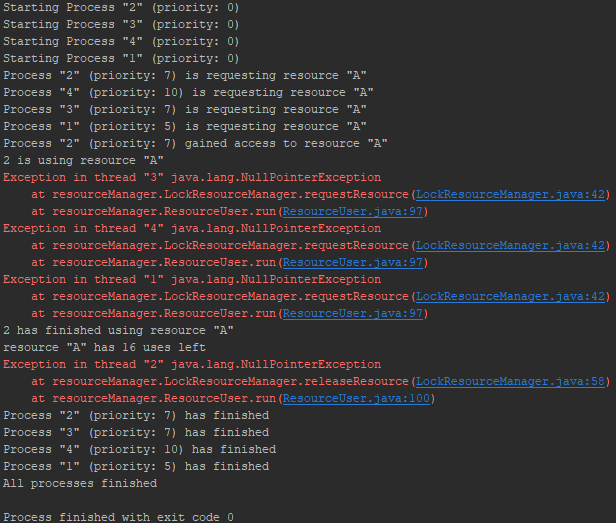
*\* the number of users, to zero.  
 \*  
 \** ***@param*** *resource the resource managed by this manager  
 \** ***@param*** *maxUses the maximum number of uses permitted for this*

*\* manager's resource.  
 \*/* public LockResourceManager(Resource resource, int maxUses) {  
 super(resource, maxUses);  
 }  
  
 @Override  
 synchronized public void requestResource(int priority) throws ResourceError {  
 if (free) {  
 lock.lock();  
 free = false;  
 }  
 else{  
 try {  
 buffer.add(priority);  
 Collections.*sort*(buffer, Collections.*reverseOrder*());  
 notFree.await();  
 } catch (InterruptedException e) {e.printStackTrace(); }  
  
 }  
 }

@Override  
 synchronized public int releaseResource() throws ResourceError {  
 if (!free){  
 free = true;  
 try{  
 int priority;  
 if(buffer.isEmpty()){priority = -1;}  
 else{  
 priority = buffer.get(0);  
 buffer.remove(0);  
 }  
 return priority;  
 }  
 finally {  
 lock.unlock();  
 }  
 }  
 else{throw new ResourceError("Attempted to release a free resource "

+ getResourceName());}  
  
 }  
}

I somewhat understand what I was supposed to do, lock the resource so that only one process can access it at a time and release one process at a time based on the value of the priority. However, when I ran the program I ended up with this output:



## Self assessment

2/5

I would give myself 2 marks for this weeks work as I have made an attempt that I believe was more than a regurgitation of the bounded buffer, though it was little more than that. I believe a combination of a lack of understanding of both the code bundle and material were to blame for this.