2019-06-21 - Programming Paradigms - Concurrent Programming Test

Study: B-CS-Mod008-2B B-CS Programming Paradigms 201400537

Number of questions: 5

Generated on: Jun 27, 2019

Contents: Pages:

2019-06-21 - Programming Paradigms - Concurrent Programming Test

Study: B-CS-Mod008-2B B-CS Programming Paradigms 201400537

In this exam, all questions are related to different parts of a system for digital testing, and to how this can be set up in a thread safe way.

The exam will be open book. You can use the slides, course manual, the papers provided for Block 5 and 6 (all available with the test), and the book Java Concurrency in Practice, Goetz, Peierls, Block, Bowbeer, Holmes and Lea (2006). In addition, you are also allowed to use the Java API documentation: https://docs.oracle.com/javase/10/docs/api/overview-summary.html. The concurrency API java.util.concurrent is part of java.base.

Here you can find the slides for the Concurrent Programming lectures:
See attachment: Show "CP1.pdf" lecture 1: Basics of Concurrency

See attachment: Show "CP2.pdf"lecture 2: Synchronisation

See attachment: Show "CP3.pdf"lecture 3: Liveness, performance and fairness

See attachment: Show "CP4.pdf"lecture 4: Homogeneous threading

See attachment: Show "CP5.pdf"lecture 5: Safe concurrency

See attachment: Show "CP6.pdf"lecture 6: Fine-grained concurrency, memory models

Here you can find the course manual: See attachment: Show "pp-student-all.pdf"

Here you can find the papers for Block 5:
See attachment: Show "Portable_Shared_Memory_Parallel_Programming.pdf"Using OpenMP - The Book

See attachment: Show "DrDobb's.pdf"Dr.Dobb's introduction to OpenCL <to be added>

Here you can find the Rust documentation for Block 6:
https://doc.rust-lang.org/beta/book/

Supplied by the Chromebook

- Test environment (https://utwente.remindotoets.nl)

- Codiva.io online C and Java environment (https://www.codiva.io/java)

- Rust Playground (https://play.rust-lang.org)

- Java 10 Docs (https://docs.oracle.com/javase/10/docs/api/overview-summary.html)

- The Rust Book (https://doc.rust-lang.org/beta/book)

These should all be opened when you start your chromebook. Only these exact urls will work! Any other urls (such as utwente.nl, but also codiva.io, which is only available by putting /java after it and https:// in front of it) are not available during the test.

Native text editor

Chromebook has a simple notepad-like text editor included. You can find it in the “circle” menu in the bottom left. This can be useful if you need to store your text somewhere when restarting a tab.

How to use Codiva.io

Open https://www.codiva.io/java and click on “Try without login”. After selecting a language this will show a text editor in the center and a file manager on the left. Which language you choose is only important for the directory structure you start with; choosing Java is recommended since this will create the appropriate directory structure for you. The Main.java/main.c file will be opened by default in the editor in the center of the screen. On the left there is the “Files” view, displaying all files available in the current repo. New files can be added by clicking the plus icon in the tab section of the text editor. Doubleclicking on a file in the Files view will open it in the center. The “Run” button at the top of the screen parses, typechecks, and compiles the current file and runs the main method/function in the current file. Any error or program output will be displayed in the terminal on the bottom.

General tips for Codiva.io

- Syntax highlighting in Codiva.io will only be enabled if your file ends in .java/.c.
- If there is a syntax error in one of the files this will prevent any file from running. So if you cannot find a syntax error in your current file, make sure the other files do not have syntax errors as well.
- Don't forget imports for AtomicInteger, Reentrantlock, #include <stdio.h>, etc. Otherwise there'll be errors.
- Don’t forget the java package if you create a java file from scratch. When creating a new file codiva.io should do this automatically.
- OpenCL kernels can be written in .c files for useful syntax highlighting. However, codiva.io does not understand OpenCL syntax (i.e. __kernel, __global, etc.). So while your OpenCL kernel might be correct, it cannot be compiled or run in codiva.io.
- Codiva.io allows “#pragma openmp statements” in C, but it does not check them for correctness. So do not expect an error when typing "#pragma openmp foo" instead of "#pragma openmp for": it’ll be ignored.
- Furthermore, OpenMP pragma’s in general will not have a runtime effect. So if you have a “#pragma openmp reduction” above your for loop do not expect an actual reduction to take place. While your annotation might be correct, it will have no runtime effect.
- If it seems the command prompt has stopped responding, or an error has occurred while saving, create a new codiva.io screen and copy over the code you were working on. Reloading the page in such a state might reset your file back to a previous state, causing data loss. Furthermore, if you get docker errors when running c programs, creating a new codiva.io screen should resolve this.

How to use the Rust Playground

Using the editor should be pretty straightforward: enter the code in the center textbox, and hit “RUN” when you want to run it. Next to the “RUN” button you can choose which debug level the code has to be compiled at and which compiler version you want. The defaults should be adequate for the test. Errors will be displayed at the bottom. Refreshing the page should retain your code; if you want to be extra safe copy over your code to the text editor. Be careful: multiple tabs of rust code will overwrite the temporary storage. Therefore only the most recent tab will be saved.

Number of questions: 5

You can score a total of 100 points for this exam, you need 55 points to pass the exam.

1 To set up an environment for digital testing, the first thing that we need is a database of questions. Teachers are allowed to add new questions, and to update existing questions. In parallel, students should be able to read all the available questions. One would expect that database modifications (adding a new question, updating an existing question) would be much less frequent then reads of existing questions (in particular, as there are many more students than teachers).

Notice that you would like to allow an update of a question during an exam. If a teacher realises there is a mistake in the question, he or she can update the questions, and the students should be able to read the corrected version immediately afterwards.

You may assume that you have a right format to store questions, modelled by a class Question (see attachment).
In addition, you may assume that you have a class Exam, which is defined as follows. The query getQuestionID() returns the index where the question is stored in the question database.

Dummy class Question: See attachment:

Simple Exam class: See attachment: (Note, the ... notation is used for a function with a flexible number of arguments)

5 pt. **a.** Why is class Exam.java threadsafe?

5 pt. **b.** Discuss which data structure you would like to use to start the database questions. The database would have to be threadsafe, and it should provide a good performance. You may assume that for a given exam, you have a valid instance of exam, thus you always know the indices to lookup the question. You may choose to define your own data structure, or reuse an existing data structure from the Java API. Motivate your answer! If you choose to develop your own data structure, please provide a sketch what it would look like.

5 pt. **c.** While a digital test is going on, another teacher might be updating his or her questions, or adding new questions to the database. Your database is threadsafe, thus this should be fine.However, suppose that you are not 100 % sure that your database is fully threadsafe for multiple updates happening simultaneously. Discuss under which conditions this would still be fine.

2 For every question and for every exam, some functionality should be available to do some statistics and manipulations on the grades of a question. For this exercise, we assume that we have the following information available:
- a matrix 'score' that stores for every student the scores that he or she obtained for a question, i.e. score[i][j] denotes the score that student i obtained for question j. All elements in this area are initialised to -1, i.e. if a student never answered the question, the default value is -1.
- for every exam, we have an array that simply lists all scores obtained for this question.

6 pt. **a.** Write a function that computes for every student the total number of answered questions, i.e. where the score is different from -1. The function should write to an array totalRight, such that totalRight[i] indicates how many questions have been answered by student i. Then use OpenMP to parallellise this function, and explain why you chose this particular parallellisation, and why you believe this gives the best improvement in performance.You may assume the following variables have been declared:#define studentnr 2 /* total number of students */
#define questionnr 30 /* total number of questions */
int totalRight[studentnr] = {0,0}; /* totalRight array: all values initialised at 0 */

4 pt. **b.** Write an OpenCL kernel that for a given array with scores, computes the total number of scores above 6, i.e. how many students passed the exam.The kernel header should look as follows:__kernel (const __global int* scores, const int numOfScores, volatile __global int* pass)

5 pt. **c.** Write an OpenCL kernel that for a given exam score, computes a so-called smoothened score. I.e., for score[i], it looks at the scores of its neighbours, and it replaces score[i] by the average of this one, and the scores of the neighbours (score[i - 1] and score[i + 1]). Hint: make sure to consider the array bounds.The kernel header should look as follows:__kernel (__global int* scores, const int numOfScores)NB You may assume that there are at least two scores.

7 pt. **d.** Some of the statistics functionality is implemented in Rust.Suppose that we have a Rust-array that contains all grades (as whole numbers from 1 to 10) scored for an exam with N = 634 students.use std::thread;
use std::sync::Arc;
use std::sync::Mutex;const N:usize = 634;Describe a function compute_totals(grades : [usize; N]) that computes the total number of grades 'N', and stores this in a vector total[i32, 10], i.e. total[4] denotes the total number of 4's that were obtained for the exam. The computation for each number should be done in parallel.

5 pt. **e.** Suppose that the number of students increases further. For further speedup, the thread to compute the total number of grades g is split up into 2 disjoint threads. The first thread will count the number of grades g in the grades-array from 0 upto N/2, the second thread will count the number of grades g in the grades-array from N/2 upto N. Adjust your function accordingly.

3 pt. **f.** Suppose that the grades would not be stored in an array, but in a vector, discuss what would have to be changed in the code.

3 An important feature of a digital test system is that the students all get access to the questions at precisely the same moment. In this question you are asked to implement the synchronization mechanism for this, using locks. Your start-synchronizer should provide the following methods:
- login(String name): which allows student to login to the digital test system, to indicate that they are ready to start working on the exam. When the exam has not started yet, they will be blocked in this state.
- start(): which is invoked by the teacher, to indicate the exam has started. After start() is called, all students that have called login() will get access to the questions. Students that arrive late will immediately get access to the questions when they login.

10 pt. **a.** Provide an implementation of this Start-synchronizer, using reentrant locks as provided in Java. You can choose yourself whether you will use Java's intrinsic or explicit locks. Use a 'guardedby' annotation to indicate which data is protected by your lock.

4 pt. **b.** Suppose that the login-method is rewritten, such that it does not take a String name as argument, but instead it opens a login dialog box, which asks the user to proivde the login name and password. Discuss how you would have to make sure that your Start-synchronizer implementation stays reactive, i.e. when many students try to login at the same time, how do you ensure that will not have to wait a long time before the login dialog box pops up. Explain why this is possible.

3 pt. **c.** A quite similar mechanism is the CountDownLatch. Discuss what is the main difference between your Start-synchronizer and Java's CountDownLatch.

8 pt. **d.** Adjust your Start-synchronizer to become lock free. Discuss the main difference in behaviour of the student-threads between the two implementations (besides that one uses locks, and the other one doesn't).

4 Another important component of the digital testing system is the grading component. Here a high level of thread safe concurrency is needed: it should support multiple correctors, and it should be possible that correctors simultaneously correct different answers from the student, as well as answers to the same question from different students.

In the following questions, you may use the following declarations (an empty class Answer is provided here). <to be added>

```
private int nrStudents;
private int nrQuestions;


/*@ invariant (\forall int i, j; 0 <= i < nrStudents && 0 <= j && j < nrQuestions; answers[i][j] != null);
    invariant (\forall int i, j; 0 <= i < nrStudents && 0 <= j && j < nrQuestions; isCorrected[i][j] <==>
scores[i][j] >= 0); */

private Answer[][] answers = new Answer[nrStudents][nrQuestions];
private int[][] scores = new int[nrStudents][nrQuestions];
private boolean[][] isCorrected = new boolean[nrStudents][nrQuestions];
```

Thus answers[i][j] is the answer of student i to question j, scores[i][j] is the number of points obtained for by student i for question j, isCorrected is a boolean flag to indicate whether an answer by a student has been corrected. All elements of scores are initialised to -1.

The invariants express that all students have given answers to all questions, and that the isCorrected-flag is only set if a score is given.

10 pt. **a.** Provide a thread safe, blocking implementation of a class Correction, containing the declarations above. The class should provide the following methods:- a constructor which receives the number of Students, the number of questions and the matrix with answers as an argument.
- giveGrade: store a grade g for student i and question j, and flag that it has been corrected. If the student already had received a grade, the method will fail.
- updateGrade: replace a grade for student i and question j. This method can only succeed if the question has been corrected before.
- claimQuestion: a corrector can make a claim to obtain exclusive access to all entries for a given a question q.
- releaseClaimQuestion: the corrector indicates that he or she is done with correcting all answers to this question q.Use a guardedby annotation to indicate how your shared data is protected by a lock.

4 pt. **b.** Give a suitable (JML) postcondition for methods giveGrade and updateGrade. Explain your answer. (In this case, you are able to specify more than just ensures true).

6 pt. **c.** Assume that we have an implementation of Software Transactional Memory (STM) that would be used instead of a lock-based solution. Discuss what would change in the implementation, and under which circumstances this would work well.

5 pt. **d.** Suppose that we would implement a functionpublic int compareGrade(int s_id1, int s_id2, int q_id1, int q_id2)which would compare the grade of student s_id1 for question q_id1 with the grade of student s_id2 for question q_id2. If the grade for student s_id1 is less than the grades for s_id2, then it would return -1, if they are equal, it would return 0, and otherwise 1.This function compareGrade(int s_id1, int s_id2, int q_id1, int q_id2) would consist of the following steps:
- obtain access of the grade of student s_id1 for question q_id1
- obtain access of the grade of student s_id2 for question q_id2
- compare the two grades and return the resultsDiscuss:
- what is the main concurrency-related risk when implementing this method?
- what would be the general solution to avoid this risk?
Explain your answer.

5 Finally, to test the system, a collection of random questions is prepared. Please answer these questions.

3 pt. **a.** Lock-free data structures are more challenging than lock-based data structures. During the lecture we discussed an unbounded lock-free queue. Explain what is the major challenge when implementing a lock-free bounded queue (which can be used by multiple enqueuers and dequeuers)

2 pt. **b.** Explain why the wait-free Fifo queue on slide 20 of lecture 6 works correctly.

Correction model

1.
15 pt.

a.

Correction criterion	Points
It is immutable: class is final, thus cannot be extended, thus no possibility to add any methods (3 points), only variable is final and private (2 point).	5 points
Total points:	*5 points*

b.

Correction criterion	Points
I would use a CopyOnWriteArrayList. Good performance, suitable for infrequent updates, and many simultaneous readers.	5 points
Total points:	*5 points*

c.

Correction criterion	Points
If we assume that each teacher works on his or her own collection of questions, then this is fine. Only if multiple teachers would be updating the same questions, this could create an issue.	5 points
Total points:	*5 points*

2.
30 pt.

a.

Correction criterion	Points

4 points for the correct parallellisation pragma: Outer loop completely independent. (4 points)
 	
Inner loop all iterations access same variable total, thus cannot be parallellised (1 point)
 	
You could use a critical annotation for the inner loop, but that will create a large number of separate threads, that each do a very small amount of work, and thus the performance will probably decrease (1 point).
 	
void computeTotals(int scores[studentnr][questionnr])
{
#pragma omp parallel for shared (scores) private (i,j)
for (int i = 0; i < studentnr; i++)
{ int total = 0;
for (int j = 0; j < questionnr; j ++)
{
if (scores[i][j] >= 0) {
total++;
}
}
totalRight[i] = total;
}
}
	6 points

Total points:	*6 points*

b.

Correction criterion	Points

No points if no atomic_add is used.
 	
__kernel (const __global int* scores, const int numOfScores, volatile __global int* pass) {
const int id = get_global_id(0);
if (scores[id] >= 6) {
atomic_add(pass, 1);
}
}
	4 points

Total points:	*4 points*

c.

Correction criterion	Points

-2 points if the array bounds are not treated as a special case
-3 points if there is no barrier
 	
__kernel (__global int* scores, const int numOfScores) {
const int id = get_global_id(0);
int low = 5;
int high = 5;
if (id > 0) {
low = scores[tid - 1];
}
if (id < numOfScores - 1) {
high = scores[tid + 1];
}
barrier(CLK_GLOBAL_MEM_FENCE);
scores[i] = (low + high + scores[tid]/3);
}
	5 points

Total points:	*5 points*

d.

Correction criterion	Points

fn compute_totals(grades : [usize; N]) {
let total = Arc::new(Mutex::new(vec![0;10]));
for i in 1 .. 11 {
let total_copy = total.clone();
thread::spawn(move || {
let mut count = 0;
for j in 0 .. N {
if grades[j] == i {
count = count + 1;
}
}
total_copy.lock().unwrap()[i] = count;
});
}
}
	7 points

Total points:	*7 points*

e.

Correction criterion	Points

Note that the result should now look at what was already stored in the total array, and add the current count to that. Otherwise, changes are minimal.
 	
fn compute_totals_split(grades : [usize; N]) {
let total = Arc::new(Mutex::new(vec![0;10]));
for i in 1 .. 11 {
let total_copy = total.clone();
thread::spawn(move || {
let mut count = 0;
for j in 0 .. N/2 {
if grades[j] == i {
count = count + 1;
}
}
let mut x = total_copy.lock().unwrap();
x[i] = x[i] + count;
});
let total_copy2 = total.clone();
thread::spawn(move || {
let mut count = 0;
for j in N/2 .. N {
if grades[j] == i {
count = count + 1;
}
}
let mut x = total_copy2.lock().unwrap();
x[i] = x[i] + count;
});
}
}
	5 points

Total points:	*5 points*

f.

Correction criterion	Points
As array, grades is copyable, thus no lock-protection needed. If grade changed to vector, then it becomes non-copyable, and type would have to be adapted to be Arc and Mutex as well.	3 points
Total points:	*3 points*

3. 25 pt. a.

Correction criterion	Points

No wait/notify-mechanism used: -5
notify instead of notifyAll: -4
notifyAll before started set to true: -4
wait not inside a loop: -4
incorrect guardedby: -2
if explicit locks are used:
-2 if lock not declared final,
-2 if finally clauses forgotten for unlock
 	
public class Start {
 	
private boolean started = false; //@ guardedby this;
 	
public synchronized void login() {
while (!started) {
try {
wait();
} catch (InterruptedException e) {
}
}
}
 	
public synchronized void start() {
started = true;
notifyAll();
}
}
 	
or
 	
import java.util.concurrent.locks.*;
 	
public class StartLock {
 	
private boolean started = false; //@ guardedby l;
 	
public final Lock l = new ReentrantLock();
public final Condition c = l.newCondition();
 	
public void login() {
l.lock();
try {
while (!started) {
try {
c.await();
	10 points

Correction criterion	Points

} catch (InterruptedException e) {
}
}
}
finally {
l.unlock();
}
}
 	
public void start() {
l.lock();
try {
started = true;
c.signalAll();
}
finally {
l.unlock();
}
}
}	

Total points:	*10 points*

b.

Correction criterion	Points
Make sure that this process with the dialog box and entering name and password is done outside the critical section, i.e. while not holding the lock. This is possible because the data is local, and not shared with other threads.	4 points
Total points:	*4 points*

c.

Correction criterion	Points
The CountDownLatch waits for a particular number of students to arrive, and only when n threads have arrived, all threads can pass the latch. In the Start-synchronizer there is one special thread that decides when all threads get access.	3 points
Total points:	*3 points*

d.

Correction criterion	Points

See StartLockFree.java
5 points for the correct implementation. Be quite tough for unnecessarily complex implementations.
3 points: main difference is that in the lock-based version, the waiting threads are suspended, while in the lock-free version they are spinning.
 	
import java.util.concurrent.atomic.*;
 	
public class StartLockFree {
 	
AtomicBoolean started = new AtomicBoolean(false);
 	
public void login() {
while (! started.get()) {
}
}
 	
public synchronized void start() {
started.set(true);
}
}
	8 points

Total points:	*8 points*

4. 25 pt. a.

Correction criterion	Points

- to implement claimQuestion and releaseClaimQuestion, explicit locks should be used. Non-working solutions using synchronized: -6.
- answer also protected by the lock: -3 points
- too coarse-grained locking (not per individual element): -5 points
- no finally: -2 points
- checking isCorrected outside the lock (only): -4 points
- locks not initialised in constructor: -2 points
- no guardedby: -3
 	
public Correction(int nrStudents, int nrQuestions, Answer[][] a) {
this.nrStudents = nrStudents;
this.nrQuestions = nrQuestions;
for (int i = 0; i < nrStudents; i++) {
for (int j = 0; j < nrQuestions; j++) {
answers[i][j] = a[i][j];
scores[i][j] = -1;
isCorrected[i][j] = false;
locks[i][j] = new ReentrantLock();
}
}
}
 	
public boolean giveGrade(int s_id, int q_id, int grade) {
locks[s_id][q_id].lock();
try {
if (isCorrected[s_id][q_id]) {
return false;
}
isCorrected[s_id][q_id] = true;
scores[s_id][q_id] = grade;
}
finally {
locks[s_id][q_id].unlock();
}
return true;
}
 	
public boolean updateGrade(int s_id, int q_id, int grade) {
locks[s_id][q_id].lock();
try {
if (! isCorrected[s_id][q_id]) {
return false;
}
scores[s_id][q_id] = grade;
}
	10 points

Correction criterion	Points

finally {
locks[s_id][q_id].unlock();
}
return true;
}
 	
public void claimQuestion(int q_id) {
for (int i = 0; i < nrStudents; i++) {
locks[i][q_id].lock();
}
}
 	
public void releaseClaimQuestions(int q_id) {
for (int i = 0; i < nrStudents; i++) {
locks[i][q_id].unlock();
}
}
}	

Total points:	*10 points*

b.

Correction criterion	Points
In both cases: ensures scores[s_id][q_id] >= 0 && isCorrected[s_id][q_id} would be an acceptable postcondition. This states that the question has been corrected, and thus (by the invariant) the scores is >= 0. You cannot specify anything about the particular grade stored, as another thread might invalidate this. Thus: scores[s_id][q_id] == grade would give 0 points.	4 points
Total points:	*4 points*

c.

Correction criterion	Points
The methods would not acquire locks anymore, but instead the method body of giveGrade and updateGrade would be executed inside a transaction (2 points). The need to claim and release all the locks for a question would disappear (2 points). This would work well if you have good agreemeents on who corrects what. If the workload is well-divided, you would expect a very low number of retries (2 points).	6 points
Total points:	*6 points*

d.

Correction criterion	Points
Criterion 1Deadlock. If thread1 acquires lock (s_id1, q_id1) and then tries to acquire lock for (s_id2, q_id2), and other thread does it in inverse order. General solution: impose static lock order, ie. always acquire lock with smallest s_id first, and if s_id1 == s_id2, acquire lock with smallest question_id first.	5 points
Total points:	*5 points*

5.
5 pt.

a.

Correction criterion	Points
You have to maintain the list data structure, pointers to the head (and tail) of the list, and at the same time maintain the length of the list.	3 points
Total points:	*3 points*

b.

Correction criterion	Points
Single enqueuer, single deqeueur. (1 point) Enqueuer and dequeuer always work on disjoint parts of the memory. (1 point)	2 points
Total points:	*2 points*