[20201217] INFOB3CC - Concurrency - 1 - at home

Course: BETA-INFOB3CC Concurrency (INFOB3CC)

Duration:	2 hours
Number of questions:	10
Generated on:	May 24, 2021

Contents: Pages:

[20201217] INFOB3CC - Concurrency - 1 - at home

Course: Concurrency (INFOB3CC)

- The exam is an open book exam. You can consult the material from the book, the webpage or other sources.
- The exam must be made alone. No communication with others is allowed.
- The first question contains further information regarding the rules and regulations of examination at home. Please read these regulations and acknowledge that you have read and accept these rules.
- Provide brief and concise answers. Overly verbose reseponses or nonsense added to otherwise good answers can deduct from your grade.
- When asked to explain your choice on a multiple choice question, your reasoning should explain why your chosen answer is correct and why the others are not correct.
- If a question does not give you all the details you need, you may make reasonable assumptions. Your assumptions must be clearly stated. If your solution only works under certain conditions, state them.
- If you think that the exam contains a mistake, send a private message via Teams to Ivo Gabe de Wolff. We have a separate channel in Teams to make any announcements about the exam, if needed.
- You have two hours to complete the exam. You can go back to previous questions.
- *Good luck! (:*

Number of questions: 10

You can score a total of 38.01 points for this exam, you need 19.01 points to pass the exam.

1 Dear student,

0.01 pt.

This test takes place under special circumstances in which we, even more than usual, rely on your professionalism and integrity.
To ensure that everyone has the same expectations, we ask you to **sign the statement below by typing "agree" in the answer box**.

If you do not agree, the test is in principle invalid. This can only be undone by a decision of the Examination Board, which can be obtained by sending the committee a written statement by email.

- I take this test under my own name / logged in with my own account.
- I take this test by myself, without contact or help from others.
- I will not copy, screen dump or otherwise record or distribute questions and answers during or after the test.
- I will only use the permitted resources that are indicated in the introduction text of the test.

I know that:

- Violation of the aforementioned agreements is regarded as Fraud (see OER art 5.14).
- Answers can be checked for plagiarism.
- The results of this test are preliminary, the examiner can invite me at a later stage for an additional oral examination (for example via Skype or Teams).

Sign by typing the word AGREE in this box

.....

After saving and submitting the answer, the test will be started.

2 2 pt.

You have been asked to implement a Map data structure to be used by multiple threads concurrently. If a key is not found in the Map, rather than returning Nothing, the thread should instead block until the desired value becomes available. Geert comes up with the following implementation:

```
import qualified Data.Map as Map

data IMap key val = IMap
  { imap_content :: IORef (Map.Map key val)
  , imap_signal  :: MVar ()
  }

lookup :: Ord key => key -> IMap key val -> IO val
lookup key imap = do
  map <- readIORef (imap_content imap)
  case Map.lookup key map of
    Just val -> return val
    Nothing  -> do
      takeMVar (imap_signal imap) -- wait until new values are available
      lookup key imap             -- try again

insert :: Ord key => key -> val -> IMap key val -> IO ()
insert key val imap = do
  atomicModifyIORef (imap_content imap) (\map -> (Map.insert key val map,
()))
  putMVar (imap_signal imap) ()   -- wake up waiters
```

Does this implementation work correctly?

Explain why this does or does not work.

3 3 pt.

Maurice needs to mirror an array on the GPU, i.e., the first element should become the last element, the second the one but last, and so on. He uses this pseudo-code to do that, where *threadIdx* is the thread index, *n* the array size and *array* the actual array.

```
if (threadIdx < n) {
  value = array[n - 1 - threadIdx]
  array[threadIdx] = value
}
```

Note that this would not work in a sequential execution, as we would read values from the array which are already overwritten in an earlier iteration of the loop. Maurice claims that this does work on a GPU, as all threads first read the old value before writing it to a different location, because of lockstep execution.

Explain why Maurice's statement is incorrect. Are there any special cases for which it would work? If so give an example and explain why.

4 How can memory operations for a (one-dimensional) stencil be optimized in the SIMD and SIMT
2 pt. models?

5 The CAS lock does not guarantee absense of starvation. To overcome this limitation, we consider an attempt on creating a lock consisting of a boolean array *flags*, denoting which threads want to acquire the lock, and a variable *lock*, denoting the index of the thread which currently has the lock or -1 if no thread has the lock. Each thread has a unique positive thread index in the range *[0 ... thread_count - 1]*.

To acquire the lock, we write to *flags* and then try to take the lock using a compare-and-swap instruction:

```
flags[thread_index] = true
while (true) {
  old = atomic_cas(lock, -1, thread_index)
  if (old == -1 || old == thread_index) { return }
}
```

To release the lock, we look for a next thread which is waiting to acquire the lock, and assign the lock to that thread, or write -1 if no other thread needs the lock:

```
flags[thread_index] = false
for (i = thread_index + 1; i < thread_index + thread_count; i++) {
  if (flags[i % thread_count]) {
    lock = i % thread_count
    return
  }
}
lock = -1
```

1 pt. **a.** Does this attempt of creating a lock guarantee mutual exclusion?

- **a.** Yes, for any number of threads
- **b.** Yes, when the program has at most two threads
- **c.** No

1 pt. **b.** Recall that deadlock freedom of a lock means that if threads want to get the lock, and it is free, they can acquire it. Does this attempt of creating a lock guarantee deadlock freedom?

- **a.** Yes, for any number of threads
- **b.** Yes, when the program has at most two threads
- **c.** No

2.5 pt. **c.** Explain why this lock does, does not, or does only for two threads guarantee deadlock freedom.

1 pt. **d.** Does this attempt of creating a lock guarantee the absense of starvation?

- **a.** Yes, for any number of threads
- **b.** Yes, when the program has at most two threads
- **c.** No

2.5 pt. **e.** Explain why this lock does, does not, or does only for two threads guarantee the absense of starvation.

6
3 pt. The type system of Haskell ensures that no side effects can be performed in an atomically-block in STM. To overcome this limitation, someone proposes to add the ability to perform IO operations, by giving both the IO operation itself and an IO operation which will undo the effect of this operation:

```
performIO :: IO a -> IO () -> STM a
performIO action undo = ...
```

You may assume that the undo operation will correctly revert all the side effects which the first action did and you may ignore type safety concerns.

Can this functionality be added to STM while still guaranteeing atomic behaviour?

Explain why this can, or can not, be implemented.

7
2 pt. Which of those scenarios is or are suitable to use a spinlock over an MVar lock?

- **a.** The lock is only taken for very short periods of time
- **b.** The lock should not waste resources while trying to acquire the lock
- **c.** The lock should provide fairness
- **d.** The lock is acquired infrequently

8 Consider we have an application where we need to take a lock on a large lists of locks. The locks are built using TMVars from STM. Tessa wants to acquire the locks in multiple *atomically*-blocks. She uses *atomically* for each variable separately:

```
acquireLocks :: [TMVar ()] -> IO ()
acquireLocks locks = mapM_ (\lock -> atomically (takeMVar lock)) locks
```

Mark says it is better to use a single *atomically*-block to acquire all the locks:

```
acquireLocks' :: [TMVar ()] -> IO ()
acquireLocks' locks = atomically (mapM_ takeTMVar locks)
```

Both approaches have problem(s) with correctness and/or performance.

1 pt. **a.** What is (are) the problem(s) with the approach by Tessa?

- **a.** Performance
- **b.** Correctness
- **c.** Both performance and correctness

2 pt. **b.** Explain how the problem(s) with Tessa's approach can occur.

1 pt. **c.** What is (are) the problem(s) with the approach by Mark?

- **a.** Performance
- **b.** Correctness
- **c.** Both performance and correctness

2 pt. **d.** Explain how the problem(s) with Mark's approach can occur.

9 The Collatz conjecture says that each positive number will eventually end up at one by iterating these rules:

- If the number is even, divide the number by two.
- If the number is odd, multiply the number by three and add one.

For this question, we want to compute how many iterations it takes for a number to end up at one. For instance, the number 4 takes two iterations, and the number 3 takes seven iterations:

```
4 -> 2 -> 1
3 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1
```

We want to know the number of iterations are required for each number in some large range.

2 pt. **a.** Explain why this can be implemented on a GPU.

2 pt. **b.** What problem can you get with this algorithm on a GPU with lockstep execution?

10 In this question, we consider a concurrent binary search tree. Nodes contain a value and (mutable) pointers to the left and right subtrees. It is a search tree, so all values in the left subtree are smaller than the value in this node, and all values in the right subtree are larger. A leaf is represented as *null*.

We use atomic *compare and swap*, which returns the old value, for the implementation of this data structure. We have an insert method with the following pseudo code:

```
void insert(Node tree, int value) {
  insertNode(tree, new Node(value, null, null))
}

void insertNode(Node tree, Node newNode) {
  if (newNode.value < tree.value) {
    let child = atomic_cas(tree.left, null, newNode)
    if (child != null) {
      insertNode(child, newNode)
    }
  } else if (newNode.value > tree.value) {
    let child = atomic_cas(tree.right, null, newNode)
    if (child != null) {
      insertNode(child, newNode)
    }
  } else {
    // Value was already present
  }
}
```

In the application, we have many threads concurrently adding nodes to the tree by calling *insert*.

2 pt. **a.** Why does this implementation of *insert* behave atomically?

2 pt. **b.** What guarantee(s) does the insert operation have?

- **a.** It guarantees starvation-freedom.
- **b.** It guarantees obstruction-freedom.
- **c.** It guarantees wait-freedom.
- **d.** It guarantees lock-freedom.

3 pt. **c.** Explain why those guarantees hold or do not hold for the insert operation.

Name:				Signature:
Date:	/ /	Birth date:	/ /	

Course: BETA-INFOB3CC Concurrency (INFOB3CC) - Questions: [20201217] INFOB3CC - Concurrency - 1 - at home

- The exam is an open book exam. You can consult the material from the book, the webpage or other sources.
- The exam must be made alone. No communication with others is allowed.
- The first question contains further information regarding the rules and regulations of examination at home. Please read these regulations and acknowledge that you have read and accept these rules.
- Provide brief and concise answers. Overly verbose reseponses or nonsense added to otherwise good answers can deduct from your grade.
- When asked to explain your choice on a multiple choice question, your reasoning should explain why your chosen answer is correct and why the others are not correct.
- If a question does not give you all the details you need, you may make reasonable assumptions. Your assumptions must be clearly stated. If your solution only works under certain conditions, state them.
- If you think that the exam contains a mistake, send a private message via Teams to Ivo Gabe de Wolff. We have a separate channel in Teams to make any announcements about the exam, if needed.
- You have two hours to complete the exam. You can go back to previous questions.
- *Good luck! (:*

1
0.01 pt.

Write your answer on the dots:

.....

2
2 pt.

Answer:

3
3 pt.

Answer:

4
2 pt.

Answer:

5
8 pt.

a. A ○ B ○ C ○

b. A ○ B ○ C ○

c.

Answer:

2

4

6

8

d. A ○ B ○ C ○

e.

Answer:

2

4

6

8

6
3 pt.

Answer:

2

4

6

8

7
2 pt.

A ☐ B ☐ C ☐ D ☐

There are multiple answers possible

8
6 pt.

a. A ○ B ○ C ○

b. Answer:

2

4

6

8

c. A ○ B ○ C ○

d. Answer:

2

4

6

8

9
4 pt.

a.

Answer:

b.

Answer:

10
8 pt.

a.

Answer:

2

4

6

8

b.

A ☐ B ☐ C ☐ D ☐

There are multiple answers possible

c.

Answer:

2

4

6

8

Correction model

1. 0.01 pt. agree
0.01 pt.

2.
2 pt.

Correction criterion	Points
It does not work	1 point
Requires threads to alternate insert and lookup	1 point
Total points:	*2 points*

3.
3 pt.

Correction criterion	Points
It does not synchronise all threads	1 point
Lockstep only operates in a warp	1 point
It does work if the array size is smaller than or equal to the warp size in lockstep execution	1 point
Total points:	*3 points*

4.
2 pt.

Correction criterion	Points
Neighbouring threads / simd lanes will read the same memory	1 point
They can share those values to reduce the number of reads from global memory	1 point
Total points:	*2 points*

5. 8 pt.

a. 1 pt. A

b. 1 pt. A

c.

Correction criterion	Points
If variable lock is -1, then a thread can take the lock.	1 point
If the variable lock is not -1, and no thread has entered the critical section, then one of the locks was assigned the index of a thread who requested the lock. That thread can thus proceed in the cas-loop of the acquire procedure.	1.5 points
Total points:	*2.5 points*

d. 1 pt. A

e.

Correction criterion	Points
Threads request the lock with the flags array	1 point
When a thread releases the lock, it finds the next thread who requested the lock. Any thread requesting the lock will thus eventually get the lock.	1.5 points
Total points:	*2.5 points*

6. 3 pt.

Correction criterion	Points
Student says that this can not be implemented.	1 point
Student explains that another thread might perform IO between the moment where the action was performed, and the moment where the action was reverted. It could thus observe an intermediate state.	2 points
Total points:	*3 points*

7. 2 pt. A

8. 6 pt.

a. 1 pt. B

b.

Correction criterion	Points
Student sees that it is a correctness problem causing deadlocks	1 point
Student describes how the program can end up in a deadlock, for instance when two threads try to acquire the same locks, but in a different order.	1 point
Total points:	*2 points*

c. 1 pt. A

d.

Correction criterion	Points
Student sees that it is a performance problem with retries in STM	1 point
Student describes that the transaction is re-run when a variable changes, which may cause many pointless tries.	1 point
Total points:	*2 points*

9. 4 pt.

a.

Correction criterion	Points
Each number can be computed independently / Data parallelism: map pattern / Embarrassingly parallel	2 points
Total points:	*2 points*

b.

Correction criterion	Points
Some numbers may need much more iterations, causing warp divergence	2 points
Total points:	*2 points*

10.
8 pt.

a.

Correction criterion	Points
atomic_cas does both the read to check for a child node, and the insert, in case of a leaf, making that atomic	1 point
Concurrent inserts targeting the same leaf will be placed under each other.	0.5 point
BST will place the same values in the same subtree, and thus prevent that values are inserted twice	0.5 point
Total points:	*2 points*

b. A
1 pt. B
C
1 pt. D
Bonus: 0 pt.

c.

Correction criterion	Points
Threads can still proceed when other threads are also inserting	1 point
If a thread had bad luck and another thread just inserted, the atomic cas fails. However, the other thread did make progress so we do have lock freedom	1 point
A thread could repeatedly have bad luck, if other threads are also inserting. It is thus not wait free nor starvation free.	1 point
Total points:	*3 points*

Caesura

Points scored	Grade
38	10
37	9.8
36	9.5
35	9.3
34	9.1
33	8.8
32	8.6
31	8.3
30	8.1
29	7.9
28	7.6
27	7.4
26	7.2
25	6.9
24	6.7
23	6.4
22	6.2
21	6.0
20	5.7
19	5.5
18	5.3
17	5.0
16	4.8
15	4.6
14	4.3
13	4.1
12	3.8
11	3.6
10	3.4
9	3.1
8	2.9
7	2.7
6	2.4
5	2.2
4	1.9
3	1.7
2	1.5
1	1.2
0	1.0