[20241217] INFOB3CC - Concurrency - 1 - USP

Course: BETA-INFOB3CC Concurrency (INFOB3CC)

Duration:	2 hours
Number of questions:	6
Generated on:	Nov 5, 2025

Contents: Pages:

[20241217] INFOB3CC - Concurrency - 1 - USP

Course: Concurrency (INFOB3CC)

- The exam is a closed book exam.
- The exam must be made alone. No communication with others is allowed.
- Provide brief and concise answers. Overly verbose responses 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.
- You may assume that threads do not crash.
- You have two hours to complete the exam. You can go back to previous questions.
- *Good luck! (:*

Number of questions: 6

You can score a total of 36 points for this exam, you need 21.6 points to pass the exam.

1 Complete the program below to create an example which may end in a deadlock.
Non-deterministically, it should either execute properly, or result in a deadlock. Assume the program has two threads, thread 1 and thread 2, and two locks, *lockA* and *lockB*.

Function *acquire(lock)* takes the given lock and *release(lock)* releases the given lock.

Thread 1

- *acquire(lockA)*
- **(a)** **A.** acquire(lockA) **B.** acquire(lockB) **C.** release(lockA) **D.** release(lockB) (0.3 pt.)
- **(b)** **A.** acquire(lockA) **B.** acquire(lockB) **C.** release(lockB) (0.25 pt.)
- **(c)** **A.** acquire(lockA) **B.** acquire(lockB) **C.** release(lockA) **D.** release(lockB) (0.3 pt.)

Thread 2

- **(d)** **A.** acquire(lockA) **B.** acquire(lockB) **C.** release(lockA) **D.** release(lockB) (0.3 pt.)
- **(e)** **A.** acquire(lockA) **B.** acquire(lockB) **C.** release(lockA) **D.** release(lockB) (0.3 pt.)
- **(f)** **A.** acquire(lockA) **B.** acquire(lockB) **C.** release(lockB) (0.25 pt.)
- **(g)** **A.** acquire(lockA) **B.** acquire(lockB) **C.** release(lockA) **D.** release(lockB) (0.3 pt.)

3 pt. **h.** Which of the following approaches can be used to prevent deadlocks?

Deadlocks can be prevented by

a. using a fair scheduler.

b. replacing locks with MVars.

c. taking locks in a fixed order.

d. using STM for concurrent code.

e. using a lock based on compare-and-swap instead of Peterson's algorithm.

f. only taking a single lock at a time.

2
3 pt.

When designing a concurrent data structure with locks, one must decide where locks are used in the structure, and thus how many locks are used. For instance, for a concurrent table or matrix we may choose to add a single lock to the entire table, to have locks per row or to use one lock per field of the table. This choice influences several aspects of the program. In this question, we consider the implications of this trade-off.

You may assume that threads don't crash in a critical section.

For each potential problem, select whether it is more likely to occur in a situation with few, or with many locks.

If a problem is not related to the granularity of locks, select 'neither'. You may select multiple answers per column.

		Few locks	Many locks	Neither
		A	**B**	**C**
Lock contention	**1**	○	○	○
Less concurrency available	**2**	○	○	○
Bugs causing deadlocks	**3**	○	○	○

3 This question considers the design of a concurrent stack. First, we consider a stack implemented using an IORef. The stack and the *push* operation (which adds a value to the stack) are given by:

```
type Stack a = IORef [a]

push :: Stack a -> a -> IO ()
push ref value = do
  ticket <- readForCAS ref
  let currentList = peekTicket ticket
  (success, _) <- casIORef ref ticket (value : currentList)
  if success then return ()
             else push ref value  -- if the CAS failed, try again
```

Similar to *push*, we can also implement *tryPop*, which removes a value from the stack if the stack is not empty (and returns Nothing if the stack is currently empty).

```
tryPop :: Stack a -> IO (Maybe a)
```

If no other threads are running, *push* and *tryPop* operate in constant time.

2 pt. **a.** What happens when two threads call *push* at the same time?

2 pt. **b.** The implementation of *push* uses *(:)* to add an item to a list. Since lists are linked lists, this takes *O(1)* time.

Imagine that the stack uses arrays or vectors instead of lists. The operation *(:)* then takes *O(n)* time, where *n* is the length of the array. What are the concurrency-related implications of this change?

Besides *push* and *tryPop*, we also want to have a *pop* function. This function is similar to *tryPop*, but it blocks if the stack is currently empty and waits until another threads pushes a value to the stack. The function *tryPop* would have returned *Nothing* in that scenario.

It is not possible to implement this using the IORef-based lock, hence we switch to STM. The queue and its operations now have the following types:

```
type Stack a = TVar [a]
push :: Stack a -> a -> STM ()
tryPop :: Stack a -> STM (Maybe a)
pop :: Stack a -> STM a             -- the new operation
```

3 pt. **c.** Implement a blocking version of *pop :: Stack a -> STM a*.

2 pt. **d.** Explain how STM internally blocks and restarts your implementation of *pop*.

4 6 pt.

Atomic fetch-and-add, *atomic_fetch_add(var, increment)*, is an atomic instruction that atomically reads a number at address *var*, adds *increment* to it, and writes it back to memory. Finally it returns the value from before the change. It is the atomic equivalent of the following procedure:

```
int fetch_add(int *var, int increment) {
  int old = *var;
  *var = old + increment;
  return old;
}
```

Suppose we have a processor that supports both the atomic compare-and-swap instruction and the atomic fetch-and-add instruction.

To also support processors that do not support the atomic fetch-and-add instruction, a colleague suggests an alternative implementation in terms of atomic compare-and-swap. They have written the following loop:

```
int x = *var;
while (true) {
  Result cas = atomic_compare_exchange(var, x, x + increment);
  if (cas.success) return;
  x = cas.original;
}
```

In this question, we compare this atomic compare-and-swap loop with the real atomic fetch-and-add instruction.

Here are some statements about atomic fetch-and-add and the atomic compare-and-swap (CAS) loop shown above. Which of them are true?

a. Atomic fetch-and-add is wait-free.

b. The CAS loop is wait-free.

c. Atomic fetch-and-add and the CAS loop have the same (strongest) progress guarantee.

d. The CAS loop is non-blocking.

e. The CAS loop does not behave as an atomic function as it consists of multiple instructions.

f. In terms of their effect on memory, these functions behave identically.

g. Atomic fetch-and-add and the CAS loop are both lock-free.

h. Whereas atomic fetch-and-add provides (‘fetches’) the old value, you cannot modify the CAS loop to also provide the old value.

5 In this question, we consider a system that uses three threads: A, B and C. Thread A is responsible for the audio of the system, B for the video and C for the user interface. Since the user interface influences which audio should be played, these threads should synchronise and communicate.

1 pt. **a.** Is it possible to execute this system on a single logical core?

- **a.** No, with one logical core thread B can only start after thread A is finished, whereas they should be executing at the same time.
- **b.** Yes, via simultaneous multi-threading (SMT).
- **c.** Yes, multiple threads take turns on this core.

1 pt. **b.** Is this scenario, with one logical core, an example of concurrency and/or parallelism?

Zero, one or two options are possible.

- **a.** Concurrency
- **b.** Parallelism

1 pt. **c.** Now assume we run these three threads on a processor with three logical cores. Is that an example of concurrency and/or parallelism?

Zero, one or two options are possible.

- **a.** Concurrency
- **b.** Parallelism

To prevent hiccups in the music, the audio thread should run every 10 milliseconds for roughly 2 milliseconds to load the next sound data.

In the next questions, we consider which progress guarantees are desired for synchronisation between the three threads.

2 pt. **d.** Explain why using a blocking algorithm for synchronisation between threads is not appropriate in this scenario.

2 pt. **e.** Is lock-free sufficient in this case? Explain why this is sufficient, or explain why it is not sufficient and what progress guarantee you would require.

6

2 pt. **a.** What are possible uses of a single MVar?

- **a.** A single MVar can be used as a one-slot communication channel.
- **b.** A single MVar can be used as a lock-free counter.
- **c.** A single MVar can be used as a lock.
- **d.** A single MVar can be used as a fast concurrent queue.

The following questions regard the behavior of *putMVar* depending on the state of an MVar called *mvar*.

1 pt. **b.** If *mvar* is currently **empty** and no threads are currently trying to read or take the value from the MVar, what is the behavior of *putMVar mvar 42*?

- **a.** It stores the value in the MVar and returns immediately.
- **b.** It does not alter the MVar and returns immediately.
- **c.** The current thread blocks until another thread reads the value from the mvar with *readMVar mvar*.
- **d.** The current thread blocks until another thread takes the value from the mvar with *takeMVar mvar*.
- **e.** The current thread blocks until another thread reads or takes the value from the mvar with *readMVar mvar* or *takeMVar mvar*.
- **f.** It throws an exception.

1 pt. **c.** If *mvar* is currently **full** and no threads are currently trying to read or take the value from the MVar, what is the behavior of *putMVar mvar 42*?

- **a.** It stores the value in the MVar and returns immediately.
- **b.** It does not alter the MVar and returns immediately.
- **c.** The current thread blocks until another thread reads the value from the mvar with *readMVar mvar*.
- **d.** The current thread blocks until another thread takes the value from the mvar with *takeMVar mvar*.
- **e.** The current thread blocks until another thread reads or takes the value from the mvar with *readMVar mvar* or *takeMVar mvar*.
- **f.** It throws an exception.

1 pt. **d.** Assume two threads, thread 1 and 2, are currently trying to **read** the value from the MVar with *readMVar*. The MVar is currently **empty**. Thread 1 called *readMVar* before thread 2.

What is the behavior of *putMVar mvar 42*?

a. It unblocks both threads performing *readMVar*.

b. It unblocks thread 1.

c. It unblocks thread 2.

d. It unblocks one of the two threads. Which thread is unblocked is non-deterministic.

e. It throws an exception.

1 pt. **e.** Assume two threads, thread 1 and 2, are currently trying to **take** the value from the MVar with *takeMVar*. The MVar is currently **empty**. Thread 1 called *takeMVar* before thread 2.

What is the behavior of *putMVar mvar 42*?

a. It unblocks both threads performing *takeMVar*.

b. It unblocks thread 1.

c. It unblocks thread 2.

d. It unblocks one of the two threads. Which thread is unblocked is non-deterministic.

e. It throws an exception.

Name:			Signature:
Date:	/ /	Date of birth:	/ /

Course: BETA-INFOB3CC Concurrency (INFOB3CC) - Questions: [20241217] INFOB3CC - Concurrency - 1 - USP

- The exam is a closed book exam.
- The exam must be made alone. No communication with others is allowed.
- Provide brief and concise answers. Overly verbose responses 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.
- You may assume that threads do not crash.
- You have two hours to complete the exam. You can go back to previous questions.
- *Good luck! (:*

1
5 pt.

a. ○ A ○ B ○ C ○ D

b. ○ A ○ B ○ C

c. ○ A ○ B ○ C ○ D

d. ○ A ○ B ○ C ○ D

e. ○ A ○ B ○ C ○ D

f. ○ A ○ B ○ C

g. ○ A ○ B ○ C ○ D

h. ☐ A ☐ B ☐ C ☐ D ☐ E ☐ F

There are multiple answers possible

2
3 pt.

1: ○ A ○ B ○ C

2: ○ ○ ○

3: ○ ○ ○

Choose one option per row

3
9 pt.

a.

Answer:

b.

Answer:

c.

Answer:

d.

Answer:

4
6 pt.

A ☐ B ☐ C ☐ D ☐ E ☐ F ☐ G ☐ H ☐

There are multiple answers possible

5
7 pt.

a. A ○ B ○ C ○

b. A ☐ B ☐

There are multiple answers possible

c. A ☐ B ☐

There are multiple answers possible

d.

Answer:

2

4

6

8

e.

Answer:

2

4

6

8

6
6 pt.

a. A ☐ B ☐ C ☐ D ☐

There are multiple answers possible

b. A ○ B ○ C ○ D ○ E ○ F ○

c. A ○ B ○ C ○ D ○ E ○ F ○

d. A ○ B ○ C ○ D ○ E ○

e. A ○ B ○ C ○ D ○ E ○

Elaboration of the answer

1.
5 pt.

a. B

b. C

c. C

d. B

e. A

f. C

g. C

h. A

B

1 pt. C

1 pt. D

E

1 pt. F

Bonus: 0 pt.

2.
3 pt.

1. 1 pt. A
2. 1 pt. A
3. 1 pt. B

3.
9 pt.

a.

Correction criterion	Points
CAS fails on one of the two threads	0 to 1 points
That thread does another iteration of the loop, and then adds its value to the stack	0 to 1 points
Total points:	*2 points*

b.

Correction criterion	Points
The program does more redundant work: it does linear work, that is thrown away if the CAS fails OR Time between read and CAS is longer, so the CAS has a higher chance of failure, leading to redundant work.	0 to 2 points
Total points:	*2 points*

c.

Correction criterion	Points
list <- readTVar var OR m <- tryPop var	0 to 0.5 points
case list of [] -> (A) ; x:xs -> (do writeTVar var xs; (B)) OR case m of Nothing -> (A); Just x -> (B)	0 to 1 points
(A) retry	0 to 1 points
(B) return x	0 to 0.5 points
Total points:	*3 points*

d.

Correction criterion	Points
retry stops the transaction	0 to 0.6 points
STM builds a list of read variables	0 to 0.7 points
If any of those variables changes, this transaction is restarted	0 to 0.7 points
Total points:	*2 points*

4. 6 pt.

1 pt. A
B
C
1 pt. D
E
1 pt. F
1 pt. G
H
Bonus: 0 pt.

5. 7 pt.

a. 1 pt. C

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

c. 1 pt. A
1 pt. B
Bonus: 0 pt.

d.

Correction criterion	**Points**
The audio thread has a small time budget	0 to 1 points
It might not meet its deadline if it needs to wait on a lock, that another thread holds	0 to 1 points
Total points:	*2 points*

e.

Correction criterion	**Points**
No: lock-free guarantees system-wide progress, but it does not guarantee the progress of a single thread. OR: It may have starvation. Or yes: lock-free does not guarantee the progress of a single thread, but the expected waiting time is low	0 to 2 points
Total points:	*2 points*

6. 6 pt.

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

b. 1 pt. A

c. 1 pt. D

d. 1 pt. A

e. 1 pt. B

Caesura

Applied guessing score: 7.193 pt

Points scored	Grade
36	10
35	9.69
34	9.38
33	9.06
32	8.75
31	8.44
30	8.13
29	7.81
28	7.50
27	7.19
26	6.88
25	6.56
24	6.25
23	5.94
22	5.63
21	5.31
20	5.00
19	4.69
18	4.38
17	4.06
16	3.75
15	3.44
14	3.13
13	2.81
12	2.50
11	2.19
10	1.88
9	1.56
8	1.25
7	1.00
6	1.00
5	1.00
4	1.00
3	1.00
2	1.00
1	1.00
0	1.00