



1

Profs. Martin Odersky and Sanidhya Kashyap
CS-206 Parallelism and Concurrency
27.04.2022 from 14h15 to 15h45
Duration : 90 minutes













SCIPER: 1000001

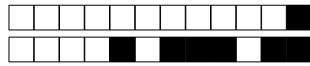
ROOM: CO1

Ada Lovelace

Wait for the start of the exam before turning to the next page. This document is printed double sided, 16 pages. Do not unstaple.

- This is a closed book exam. No electronic devices allowed.
- Place on your desk: your student ID, writing utensils place all other personal items below your desk or on the side.
- You each have a different exam. For technical reasons, **do use black or blue pens for the MCQ part, no pencils!** Use white corrector if necessary.
- **Your Time:** All points are not equal: we do not think that all exercises have the same difficulty, even if they have the same number of points.
- **Your Attention:** The exam problems are precisely and carefully formulated, some details can be subtle. Pay attention, because if you do not understand a problem, you cannot obtain full points.
- The two last pages of this exam contains an appendix. Do not detach this sheet.

Respectez les consignes suivantes Observe this guidelines Beachten Sie bitte die unten stehenden Richtlinien		
choisir une réponse select an answer Antwort auswählen	ne PAS choisir une réponse NOT select an answer NICHT Antwort auswählen	Corriger une réponse Correct an answer Antwort korrigieren
  		 
ce qu'il ne faut PAS faire what should NOT be done was man NICHT tun sollte		
     		



First part: single choice questions

Each question has **exactly one** correct answer. Marking only the box corresponding to the correct answer will get you 4 points. Otherwise, you will get 0 points for the question.

Function `parallel3` is a variation of the `parallel` construct seen in class:

```
1 def parallel3[A, B, C](op1: => A, op2: => B, op3: => C): (A, B, C) =  
2   val res1 = task { op1 }  
3   val res2 = task { op2 }  
4   val res3 = op3  
5   (res1.join(), res2.join(), res3)
```

Consider the following code that uses the `parallel3` function:

```
1 def find(arr: Array[Int], value: Int, threshold: Int): Option[Int] =  
2   def findHelper(start: Int, end: Int): Option[Int] =  
3     if end - start <= threshold then  
4       var i = start  
5       while i < end do  
6         if arr(i) == value then  
7           return Some(value)  
8         i += 1  
9       None  
10    else  
11      val inc = (end - start) / 3  
12      val (res1, res2, res3) = parallel3(  
13        findHelper(start, start + inc),  
14        findHelper(start + inc, start + 2 * inc),  
15        findHelper(start + 2 * inc, end)  
16      )  
17      res1.getOrElse(res2).getOrElse(res3)  
18      findHelper(0, arr.length)
```

Question 1 How many time will `task` be called when running `find` with the following arguments?

```
1 find(Array(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19), 18, 3)
```

- ☐ 9
- ☐ 3
- ☐ 15
- ☐ 6
- ☐ 0
- ☐ 10
- ☐ 5
- ☐ 16



Question 2 Define `f` and `g` such that `find_aggregated` computes the same result as the `find` method:

```
1 def findAggregated(arr: Array[Int], value: Int): Option[Int] =  
2   val no: Option[Int] = None  
3   val yes: Option[Int] = Some(value)  
4   arr.par.aggregate(no)(f, g)
```

- ☐ `def f = (x1: Int, x2: Option[Int]) => if (x1 == value) yes else Some(x1)`
`def g = (x1: Int, x2: Int) => if (x1 != value) x2 else x1`
- ☐ `def f = (x1: Option[Int], x2: Int) => if (x2 == value) Some(x2) else x1`
`def g = (x1: Option[Int], x2: Option[Int]) => if (x1 != None) x1 else x2`
- ☐ `def f = (x1: Option[Int], x2: Option[Int]) => if (x1 == yes) x2 else x1`
`def g = (x1: Option[Int], x2: Option[Int]) => if (x1 == None) x2 else x1`
- ☐ `def f = (arg1: Option[Int], x2: Int) => if (x2 == value) value else arg1`
`def g = (x1: Option[Int], x2: Option[Int]) => if (x2 != None) x1 else x2`
- ☐ `def f = (arg1: Int, arg2: Int) => if (arg1 == value) Some(arg1) else arg2`
`def g = (x1: Option[Int], x2: Option[Int]) => if (x2 != None) x1 else x2`
- ☐ `def f = (x: Option[Int], y: Option[Int]) => if (x != yes) None else yes`
`def g = (x: Option[Int], y: Option[Int]) => if (x == None) y else x`
- ☐ `def f = (x1: Option[Int], x_2: Int) => if (x1 == yes) Some(x_2) else x1`
`def g = (x1: Option[Int], x_2: Option[Int]) => if (x1 == None) x_2 else x1`

Question 3 What is the range of possible number of calls to `g` when running `find_aggregated` with the following arguments:

```
1 find_aggregated(Array(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19),9)
```

- ☐ [1, 10]
- ☐ [0, 18]
- ☐ [1, 20]
- ☐ [0, 9]
- ☐ [0, 19]
- ☐ [0, 20]
- ☐ [1, 18]
- ☐ [1, 19]



Consider the following contains function defined on Iterable (in particular it accepts both Vector and List).

```
1 def contains[A](l: Iterable[A], elem: A): Boolean =
2   val n = l.size
3   if n <= 5 then
4     for i <- l do
5       if i == elem then
6         return true
7     false
8   else
9     val (p0, p1) = parallel(
10      contains(l.take(n / 2), elem),
11      contains(l.drop(n / 2), elem)
12    )
13    p0 || p1
```

Let n be the size of l . Assume that drop and take run in $\Theta(1)$ on Vector, and in $\Theta(n)$ on List.

Question 4 What is the asymptotic work of contains if it is called on a Vector?

- ☐ $W(n) = \Theta(\log(n))$
- ☐ $W(n) = \Theta(1)$
- ☐ $W(n) = \Theta(n \cdot \log(n))$
- ☐ $W(n) = \Theta(n)$

Question 5 What is the asymptotic depth of contains if it is called on a Vector?

- ☐ $D(n) = \Theta(n)$
- ☐ $D(n) = \Theta(\log(n))$
- ☐ $D(n) = \Theta(n \cdot \log(n))$
- ☐ $D(n) = \Theta(1)$

Question 6 What is the asymptotic work of contains if it is called on a List?

- ☐ $W(n) = \Theta(n \cdot \log(n))$
- ☐ $W(n) = \Theta(\log(n))$
- ☐ $W(n) = \Theta(n)$
- ☐ $W(n) = \Theta(1)$

Question 7 What is the asymptotic depth of contains if it is called on a List?

- ☐ $D(n) = \Theta(n \cdot \log(n))$
- ☐ $D(n) = \Theta(\log(n))$
- ☐ $D(n) = \Theta(1)$
- ☐ $D(n) = \Theta(n)$

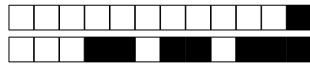


Question 8 Read the code below and select which is the correct output of the code.

```
1 def thread(b: => Unit) =  
2   val t = new Thread:  
3     override def run() = b  
4   t  
5 val t = thread { println(s"Hello World") }  
6 t.join()  
7 println(s"Hello")
```

- ☐ Hello World
Hello
- ☐ Hello World
- ☐ Hello
Hello World
- ☐ Hello

DRAFT



Second part: yes/no questions

The answer of each question is **either “Yes”, either “No”**. Marking only the box corresponding to the correct answer will get you 2 points. Marking only the wrong answer will get you -1 point. Otherwise, you will get 0 point for the question.

Read the code below and answer the following questions.

```
1 class Node(  
2     // Globally unique identifier. Different for each instance.  
3     val guid: Int  
4 )  
5  
6 // sum and total variables are shared by all the threads.  
7 var sum: Int = 0  
8 var total = new AtomicInteger(0)  
9  
10 def increment(e: Int) = sum += e  
11  
12 // This function might be called concurrently.  
13 def lockFun(nodes: List[Node], fn: (e: Int) => Unit): Unit =  
14     if nodes.size > 0 then  
15         nodes.head.synchronized {  
16             fn(nodes(0).guid)  
17             lockFun(nodes.tail, fn)  
18         }  
19     else  
20         println(sum + " " + total.get)  
21  
22 // List of nodes used by the current thread. This list is a subset of  
23 // a global list of nodes shared by all threads.  
24 var nodes: List[Node] = getFromInput()  
25 nodes = ???  
26 lockFun(nodes, increment)
```

Question 9 Replace line 25 by the line below. Select Yes if the code can result in a deadlock.

```
1 nodes = nodes
```

☐ Yes ☐ No

Question 10 Replace line 25 by the line below. Select Yes if the code can result in a deadlock.

```
1 nodes = nodes.sortWith((x,y) => x.guid > y.guid)
```

☐ Yes ☐ No

Question 11 Replace line 25 by the line below. Select Yes if the code can result in a deadlock.

```
1 nodes = nodes.sortWith((x,y) => x.guid < y.guid)
```

☐ Yes ☐ No



Question 12 Replace line 25 by the line below. Select Yes if the code can result in a deadlock.

```
1 nodes = nodes.tail.appended(nodes(0))
```

☐ Yes ☐ No

For the 3 next questions, assume that line 25 is replaced such that no deadlock can happen.

Question 13 Replace line 10 by the line below. Select Yes if the code will compute the correct value of sum.

```
1 def increment(e: Int) = sum += e
```

☐ Yes ☐ No

Question 14 Replace line 10 by the line below. Select Yes if the code will compute the correct value of sum.

```
1 def increment(e: Int) = synchronized { sum += e }
```

☐ Yes ☐ No

Question 15 Replace line 10 by the line below. Select Yes if the code will compute the correct value of total.

```
1 def increment(e: Int) = total.addAndGet(e)
```

☐ Yes ☐ No



Consider the following function call on a possibly parallelized sequence:

```
1 seq.fold(z: B) (f: (B, B) -> B)
```

For which of the following type and arguments is the result of that call deterministic (i.e. it always produce the same result than if it was executed sequentially)?

Question 16 B=Int, z=1, f= (a,b) -> a+b

☐ Yes ☐ No

Question 17 B=List[C], z=Nil, f= (a,b) -> a++b

☐ Yes ☐ No

Question 18 B=String, z="", f= (a,b) -> **if** (a.size > b.size) a **else** b

☐ Yes ☐ No

Question 19 B=Boolean, z=True, f= (a,b) -> (a && b) || (!b && !a)

☐ Yes ☐ No

Question 20 B=Double, z=0, f= (a,b) -> a+b

☐ Yes ☐ No

Question 21 Consider the following code:

```
1 class TicketsManager(totalTickets: Int):  
2   var remainingTickets = totalTickets  
3  
4   // This method might be called concurrently  
5   def getTicket(): Boolean =  
6     if remainingTickets > 0 then  
7       this.synchronized {  
8         remainingTickets -= 1  
9       }  
10    true  
11    else false
```

If getTicket() is called concurrently, can there be a race condition on remainingTickets?

☐ Yes ☐ No



We implemented a function as shown below to assign Network Interface Cards (NIC) to concurrent threads. Each thread requires two NICs for receiving and sending data. When threads finish their jobs, they will release their corresponding NICs.

```
1 class NIC(val index: Int, var assigned: Boolean)
2
3 class NICManager(n: Int):
4     // Creates a list with n NICs
5     val nics = (for i <- 0 until n yield NIC(i, false)).toList
6
7     // This method might be called concurrently
8     def assignNICs(): (Int, Int) =
9         var recvNIC: Int = 0
10        var sendNIC: Int = 0
11        var gotRecvNIC: Boolean = false
12        var gotSendNIC: Boolean = false
13
14        /// Obtaining receiving NIC...
15        while !gotRecvNIC do
16            nics(recvNIC).synchronized {
17                if !nics(recvNIC).assigned then
18                    nics(recvNIC).assigned = true
19                    gotRecvNIC = true
20                else
21                    recvNIC = (recvNIC + 1) % n
22            }
23        // Successfully obtained receiving NIC
24
25        // Obtaining sending NIC...
26        while !gotSendNIC do
27            nics(sendNIC).synchronized {
28                if !nics(sendNIC).assigned then
29                    nics(sendNIC).assigned = true
30                    gotSendNIC = true
31                else
32                    sendNIC = (sendNIC + 1) % n
33            }
34        // Successfully obtained sending NIC
35
36        return (recvNIC, sendNIC)
```

Question 22 If the number of concurrent threads is less than n , is it possible to have deadlocks?

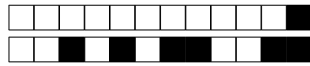
☐ Yes ☐ No

Question 23 Is there definitely a deadlock if the number of concurrent threads is more than n ?

☐ Yes ☐ No

Question 24 The number of available NICs is n and we limit the number of available receiving NICs to $n - 1$ (i.e. we replace line 21 by $\text{recvNIC} = (\text{recvNIC} + 1) \% (n - 1)$). Can there be any deadlock?

☐ Yes ☐ No



Third part, open questions

Question 25: *This question is worth 36 points.*

<input type="checkbox"/>	0	<input type="checkbox"/>	1	<input type="checkbox"/>	2	<input type="checkbox"/>	3	<input type="checkbox"/>	4	<input type="checkbox"/>	5	<input type="checkbox"/>	6	<input type="checkbox"/>	7	<input type="checkbox"/>	8	<input type="checkbox"/>	9	<input type="checkbox"/>	10
<input type="checkbox"/>	11	<input type="checkbox"/>	12	<input type="checkbox"/>	13	<input type="checkbox"/>	14	<input type="checkbox"/>	15	<input type="checkbox"/>	16	<input type="checkbox"/>	17	<input type="checkbox"/>	18	<input type="checkbox"/>	19	<input type="checkbox"/>	20		
<input type="checkbox"/>	21	<input type="checkbox"/>	22	<input type="checkbox"/>	23	<input type="checkbox"/>	24	<input type="checkbox"/>	25	<input type="checkbox"/>	26	<input type="checkbox"/>	27	<input type="checkbox"/>	28	<input type="checkbox"/>	29	<input type="checkbox"/>	30		
<input type="checkbox"/>	31	<input type="checkbox"/>	32	<input type="checkbox"/>	33	<input type="checkbox"/>	34	<input type="checkbox"/>	35	<input type="checkbox"/>	36										

In this exercise, you will implement a data structure to handle the "following" relation in a social network (who is following who). This data structures should contain 4 methods `add`, `follow`, `unfollow` and `delete` that are documented below.

Internally, your data structure should use a `TrieMap` to store the relation and an atomic counter to store the maximum id given to a user. You can find the documentation for `TrieMap` and `AtomicInteger` in the appendix at the end of the exam.

Here is the interface of what you should implement:

```
1 import scala.collection.concurrent.TrieMap
2 import java.util.concurrent.atomic.AtomicInteger
3
4 // Represent a social network where user can follow each other. Each user is
5 // represented by an id that is an 'Int'.
6 abstract class AbstractInstagram:
7   // The map storing the "following" relation of our social network.
8   // 'graph(a)' contains the list of user ids that user 'a' follows.
9   val graph = new TrieMap[Int, List[Int]]()
10  // The maximum user id allocated until now. This value should be incremented
11  // by one each time a new user is added.
12  val maxId = new AtomicInteger(0)
13  // Allocates a new user and returns its unique id. Internally, this should
14  // also create an empty list at the corresponding id in 'graph'. The
15  // implementation must be thread-safe.
16  def add(): Int
17  // Make 'a' follow 'b'. The implementation must be thread-safe.
18  def follow(a: Int, b: Int): Unit
19  // Makes 'a' unfollow 'b'. The implementation must be thread-safe.
20  def unfollow(a: Int, b: Int): Unit
21  // Removes user with id 'a'. This should also remove all references to 'a'
22  // in 'graph'. The implementation must be thread-safe.
23  def remove(a: Int): Unit
```

Example successful sequential run:

```
1 val insta = Instagram()
2 assertEquals(1, insta.add())
3 assertEquals(2, insta.add())
4 insta.follow(1, 2)
5 assertEquals(insta.graph, Map(1 -> List(2), 2 -> List()))
6 insta.follow(2, 1)
7 insta.unfollow(1, 2)
8 assertEquals(insta.graph, Map(1 -> List(), 2 -> List(1)))
9 insta.follow(3, 1) // fails silently
10 assertEquals(insta.graph, Map(1 -> List(), 2 -> List(1)))
11 insta.remove(1)
12 assertEquals(insta.graph, Map(2 -> List()))
13 insta.unfollow(1, 2) // fails silently
```



Your implementation **must** follow these guidelines:

- Your methods must be thread-safe and lock-free.
- You can *not* use `synchronized`.
- At the end of each operation, your data structure must be in a valid state: there must be no user following an inexistent or removed user. Also you must make sure that no `follow` or `unfollow` call is ignored when it shouldn't.

```
class Instagram extends AbstractInstagram:
```



DRAFT



DRAFT



DRAFT



Appendix: Scala and Java Standard Library Methods

Here are the prototypes of some Scala and Java classes that you might find useful:

```
// Represents optional values. Instances of Option are either an instance of
// scala.Some or the object None.
sealed abstract class Option[+A]:
  // Returns the option's value if the option is an instance of scala.Some, or
  // throws an exception if the option is None.
  def get: A
  // Returns true if the option is an instance of scala.Some, false otherwise.
  // This is equivalent to:
  //   option match
  //     case Some(v) => true
  //     case None   => false
  def isDefined: Boolean
  // Returns this scala.Option if it is nonempty, otherwise return the result of
  // evaluating alternative.
  def orElse[B >: A](alternative: => Option[B]): Option[B]

abstract class Iterable[+A]:
  // Selects all elements except first n ones.
  def drop(n: Int): Iterable[A]
  // The size of this collection.
  def size: Int
  // Selects the first n elements.
  def take(n: Int): Iterable[A]

abstract class List[+A] extends Iterable[A]:
  // Adds an element at the beginning of this list.
  def ::[B >: A](elem: B): List[B]
  // A copy of this sequence with an element appended.
  def appended[B >: A](elem: B): List[B]
  // Get the element at the specified index.
  def apply(n: Int): A
  // Selects all elements of this list which satisfy a predicate.
  def filter(pred: (A) => Boolean): List[A]
  // Selects the first element of this list.
  def head: A
  // Sorts this sequence according to a comparison function.
  def sortWith(lt: (A, A) => Boolean): List[A]
  // Selects all elements except the first.
  def tail: List[A]

abstract class Array[+A] extends Iterable[A]:
  // Get the element at the specified index.
  def apply(n: Int): A

abstract class Thread:
  // Subclasses should override this method.
  def run(): Unit
  // Causes this thread to begin execution; the Java Virtual Machine calls the
  // run method of this thread.
  def start(): Unit
  // Waits for this thread to die.
  def join(): Unit

// Creates and starts a new task ran concurrently.
def task[T](body: => T): ForkJoinTask[T]
```



```
abstract class ForkJoinTask[T]:
  // Returns the result of the computation when it is done.
  def join(): T

// A concurrent hash-trie or TrieMap is a concurrent thread-safe lock-free
// implementation of a hash array mapped trie.
abstract class TrieMap[K, V]:
  // Retrieves the value which is associated with the given key. Throws an exception
  // if there is no mapping from the given key to a value.
  def apply(key: K): V
  // Tests whether this map contains a binding for a key.
  def contains(key: K): Boolean
  // Applies a function f to all elements of this concurrent map. This function
  // iterates over a snapshot of the map.
  def foreach[U](f: ((K, V)) => U): Unit
  // Optionally returns the value associated with a key.
  def get(key: K): Option[V]
  // Collects all key of this map in an iterable collection. The result is a
  // snapshot of the values at a specific point in time.
  def keys: Iterator[K]
  // Transforms this map by applying a function to every retrieved value. This
  // returns a new map.
  def mapValues[W](f: V => W): TrieMap[K, W]
  // Associates the given key with a given value, unless the key was already
  // associated with some other value. This is an atomic operation.
  def putIfAbsent(k: K, v: V): Option[V]
  // Removes a key from this map, returning the value associated previously with
  // that key as an option.
  def remove(k: K): Option[V]
  // Removes the entry for the specified key if it's currently mapped to the
  // specified value. This is an atomic operation.
  def remove(k: K, v: V): Boolean
  // Replaces the entry for the given key only if it was previously mapped to a
  // given value. Returns true if the change is successful, or false otherwise.
  // This is an atomic operation.
  def replace(k: K, oldvalue: V, newvalue: V): Boolean
  // Adds a new key/value pair to this map.
  def update(k: K, v: V): Unit
  // Collects all values of this map in an iterable collection. The result is a
  // snapshot of the values at a specific point in time.
  def values: Iterator[V]

// An int value that may be updated atomically.
// The constructor takes the initial value at its only argument. For example,
// this create an 'AtomicInteger' with an initial value of '42':
//     val myAtomicInteger = new AtomicInteger(42)
abstract class AtomicInteger:
  // Atomically adds the given value to the current value and returns the
  // updated value.
  def addAndGet(delta: Int): Int
  // Atomically sets the value to the given updated value if the current value
  // == the expected value. Returns true if the change is successful, or false
  // otherwise. This is an atomic operation.
  def compareAndSet(oldvalue: Int, newvalue: Int): Boolean
  // Gets the current value. This is an atomic operation.
  def get(): Int
  // Atomically increments by one the current value. This is an atomic operation.
  def incrementAndGet(): Int
```