Parallelism and Concurrency

Midterm Exam

Wednesday, April 12, 2017

Your points are *precious*, don't let them go to waste!

Your Time All points are not equal. Note that 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 can not obtain full points.

Exercise	**Points**	**Points Achieved**
1	25	
2	25	
3	25	
4	25	
Total	100	

Exercise 1: Parallel Search (25 points)

Finding the index of the first occurence of an item in an array is a standard problem that can be solved naively by iterating through the array, checking at each element if it matches the one we are looking for. In this exercise, you will implement a function to perform this task in parallel.

Problem

In this exercise your task is to implement an efficient parallel version of the search algorithm. *Your implementation should use the* `parallel` *function seen in class.* The function you are implementing should have the following signature:

```
def find(arr: Array[Int], value: Int): Option[Int] = ???
```

The function takes as inputs an array of integers and a value. If the value is present in the array, it should return the *smallest index* at which the value can be found, wrapped in `Some`. Otherwise, it should return `None`.

For example:

```
find(Array(17, 5, 42, 11), 42) == Some(2)
find(Array(17, 5, 42, 11), 34) == None
find(Array(11, 5, 11, 11), 11) == Some(0)
```

Threshold

Your implementation should work sequentially when the size of the array segment considered is below or equal to the constant value `THRESHOLD`, and in parallel otherwise.

Library Functions

The appendix contains the signature of `parallel` as well as the signature of useful methods on arrays.

Please answer on *next* page.

Please answer on this page.

Exercise 2: Parallel Word Splitting (25 points)

Splitting a text into a sequence of words is a key preprocessing step in many systems that work with textual data. In this exercise, you will write a Scala function to perform this crucial task in parallel.

Definitions

- We consider a word to be a *non-empty* sequence of *non-whitespace* characters.
- Words in the text are separated by *one or more* whitespaces.
- Whitespaces can also appear at the beginning and end of the text.

Example Starting from the following text:

```
"Hello, I'm   a text! "
```

We would like to obtain the following sequence of words:

```
"Hello,", "I'm", "a", "text!"
```

Statement Your goal for this exercise is to write, in Scala, a *parallel* and *efficient* implementation of the `toWords` function. The function takes as input a parallel sequence of characters and outputs a `Vector` of words.

```
def toWords(chars: ParSeq[Char]): Vector[String] = ???
```

The output of the function should contain all words, in order, that appear in the sequence of characters `chars`.

Notes

- A reduced API for `ParSeq` is presented in appendix.
- A `Vector` is an *immutable* sequential sequence with an efficient implementation of the following methods:
  - `:+`, appends an element at the end.
  - `+:`, prepends an element at the beginning.
  - `++`, concatenates two sequences.
  - `.head`, returns the first element.
  - `.last`, returns the last element.
  - `.tail`, returns a `Vector` containing all elements but the first.
  - `.init`, returns a `Vector` containing all elements but the last.
- The following operations on `String` are considered to be efficient enough in the context of this exercise:
  - `:+`, appends a `Char` at the end of the `String`.
  - `+:`, preprends a `Char` at the beginning of the `String`.
  - `++`, concatenates two strings.
- You should use the method `isWhitespace` of `Char` to check whether a character is a whitespace or not.

Hints Think of the corner cases. What if the text is only whitespaces ? What if it contains a single word ?

Please answer on this page.

Exercise 3: Memory Models (25 points)

Assume the following two threads.

Initial state:

```
var a, b, c = 0
val lock = new AnyRef
```

Thread 1:

```
a = 1
b = 2
lock.synchronized { c = a + b }
```

Thread 2:

```
def f = a + b
println(if (b == 2) a else 2)
lock.synchronized { println(f) }
while (c == 0) {}
println(f)
lock.synchronized { println(f) }
```

Question 1

Under the sequential consistency model, what are the possible values printed by Thread 2? Use one line per print statement, and list the possible values that could have been printed on that line.

Please answer question 1 in the space below.

Question 2

Under the Java memory model, what are the possible values printed by Thread 2? Use one line per print statement, and list the possible values that could have been printed on that line.

Please answer question 2 in the space below.

Exercise 4: Lock-Free Banking (25 points)

During the course, we have seen the example of a bank account class (called `Account`) with thread-safe `getAmount` and `transfer` methods. The class has the following API:

```
class Account(initialAmount: Long = 0L) {
  def getAmount: Long = ???
  def transfer(target: Account, n: Long): Unit = ???
}
```

In this exercise, we will consider a slightly modified variant, with the following requirements:

- `getAmount` should return the current amount of money that the account holds. It should never return an amount that is inconsistent with the set of transfers performed on this account.
- `transfer` should transfer money from an account to an other, with the following requirements:
  - Attempting to transfer a negative amount of money `n` should result in an `IllegalArgumentException` being thrown and the transfer being aborted.
  - Attempting to transfer more money than available should result in an `IllegalStateException` being thrown and the transfer being aborted.
  - When a transfer is aborted, it should not modify the amount of money on either accounts.
  - If the operation is not aborted, the amount of money in `this` account should be decremented by `n`.
  - If the operation is not aborted, the amount in the `target` account should be incremented by `n`.
- If a set of transfers are performed concurrently, the results (succeeding or failing) must be consistent with one possible sequential ordering of the transfers.

For your convenience, a correct, dead-lock free, implementation of the `Account` class is shown in the appendix.

Statement

Your task is to write a revised version of `Account` with the same public API (i.e., a `getAmount` method and a `transfer` method with the same specification) but that is implemented *in a lock-free way*. An implementation that is not lock-free will not be considered valid.

To do so, you should use an `AtomicLong` to represent the `amount`, and adapt the implementation of `getAmount` and `transfer` accordingly. The API of `AtomicLong` is presented in the appendix.

To get full points, your implementation must comply to *all* of the requirements stated above and be lock-free.

Assumptions

You may assume that `Long` operations never overflow.

Desirable Properties

Your implementation should aim at minimizing the number of atomic operations performed during `transfer` and `getAmount`, i.e., the number of calls to methods of `AtomicLong`.

Please answer on next page.

Please answer on this page.

Parallel API

- `def parallel[A, B](op1: => A, op2: => B): (A, B)`: Executes the two given computations in parallel and returns the pair of results.

Option API

Relevant API for `Option[A]`:

- `def map[B](f: (A) => B): Option[B]`: Returns a `Some` containing the result of applying f to this `Option`'s value if this `Option` is nonempty. Otherwise return `None`.
- `def flatMap[B](f: (A) => Option[B]): Option[B]`: Returns the result of applying `f` to this `Option`'s value if this `Option` is nonempty.
- `def flatten[B](implicit ev: <:<[A, Option[B]]): Option[B]`: Returns flattened option.
- `def get: T`: Returns the option's value.
- `def getOrElse[B >: A](default: => B): B`: Returns the option's value if the option is nonempty, otherwise return the result of evaluating default.
- `def orElse[B >: A](alternative: => Option[B]): Option[B]`: Returns this `Option` if it is nonempty, otherwise return the result of evaluating alternative.

Array API

Relevant API for `Array[T]`:

- `def apply(index: Int): T`: Gets the element at a given index. Indexes start at 0.
- `def length: Int`: Returns the length of the array.

ParSeq API

Relevant API for `ParSeq[A]`. You can assume that the implementations of the following methods are similar to what was seen during the lectures.

- `def map[B](f: (A) => B): ParSeq[B]`: Applies, in parallel, a function on all elements of the sequence, and returns the sequence of results.
- `def reduce(op: (A, A) => A): A`: Reduces the elements of this sequence using the specified operator.
- `def aggregate[B](z: => B)(f: (B, A) => B, g: (B, B) => B): B`: Aggregates the results of applying an operator to the elements.

AtomicLong API

Relevant API for `AtomicLong`:

- `new AtomicLong(initialValue: Long)`: Creates a new `AtomicLong` with the specified initial value.
- `def get: Long`: Atomically returns the current value of the `AtomicLong`.
- `def compareAndSet(oldV: Long, newV: Long): Boolean`: Atomically sets the value of the `AtomicLong` to `newV` if the current value is `== oldV`. Returns `true` if it succeeds (the current value was `oldV`) or `false` otherwise.
- `def addAndGet(delta: Long): Long`: Atomically adds `delta` to the current value of the `AtomicLong` and returns the new value (i.e., the old value `+ delta`). This operation always succeeds.

Example Implementation of Account

Implementation of `Account` using locks:

```
class Account(initialAmount: Long = 0L) {
  private val id = generateUniqueID()
  private var amount: Long = initialAmount

  def getAmount: Long = this.synchronized {
    amount
  }

  def transfer(target: Account, n: Long): Unit = {
    if (n <= 0L)
      throw new IllegalArgumentException("n must be positive")

    val (first, second) =
      if (this.id < target.id) (this, target)
      else (target, this)

    first.synchronized {
      second.synchronized {
        if (this.amount < n)
          throw new IllegalStateException("Not enough money")
        this.amount -= n
        target.amount += n
      }
    }
  }
}
```