Programming Assignment #1

CS–735/835

due 12 Feb 2017 11:59 PM

1 Introduction

The theme of this assignment is to implement three variants of a simple thread-safe data structure and to evaluate them to compare their performances. The assignment also illustrates how systems with increased parallelism sometimes have to relax their semantics.[1]

Consider the problem of counting words from multiple textual sources (files, URLs) in parallel. Since counting is fast and the hard work is in I/O and the parsing of text, a possible approach is to use one thread as the counter and as many additional threads as there are sources. Each source thread handles I/O and parsing for its source and sends its words to the counting thread.[2]

Below is the output of the application with 5 sources (2 URLs and 3 files, all with the same text) and 6 threads (5 readers and 1 counter):

```
% sbt 'runMain cs735_835.WordCountApp
> http://cs.unh.edu/~cs735/Programming/usconst.txt
> http://cs.unh.edu/~cs735/Programming/usconst.txt
> src/test/resources/usconst.txt
> src/test/resources/usconst.txt
> src/test/resources/usconst.txt'
37885 words counted
10 most frequent words:
 3315: the
 2475: of
 1530: shall
 1290: and
  920: to
  895: be
  800: or
  695: in
  625: States
```

This application follows a *producer-consumer* pattern: k threads produce words and one thread consumes them. For the threads to cooperate, they need to share a *buffer* or *channel*, a data structure in which producers put words and consumers retrieve words. Note that the producing threads are likely to touch the channel less often than the consuming thread because they need to perform I/O and parsing, but that there can be several of them trying to access the channel at the same time.

In this assignment, the channel has type `cs735_835.channels.Channel`, as defined by the interface of Lis. 1. Conceptually, such a channel offers a number of input queues (`queueCount`) in which to put words (`put`) and a single output queue in which to retrieve words (`get`). The role of the channel is to merge the values inserted in the input queues into the output queue. Additionally, the channel includes a method

[1] See, for instance, the documentation for `java.util.concurrent.ConcurrentLinkedQueue.size`.

[2] If all that is needed are the final counts, it would be simpler and more efficient to have the source threads maintain partial counts and to combine these counts when the threads are finished. The approach used in the assignment is chosen to illustrate the use of a *producer-consumer* pattern. This approach would allow the application to start using the counts while sources are still being processed.

```java
package cs735_835.channels;

public interface Channel<T> {

  int queueCount ();
  T get ();
  void put (T value, int queue);
  long totalCount ();
}
```

Listing 1: The `Channel` interface.

`totalCount` that produces the total number of items added to the channel (including those that have already been retrieved).[3]

For the sake of generality, channel implementations should be thread-safe and should correctly handle cases in which the consuming thread is not the thread that created the channel or even the case of multiple consuming threads (though there won't be parallelism among them because of the single output queue).

2 `SimpleChannel` class

The first implementation to write is very simple. It uses a single queue internally, implemented as an instance of `java.util.LinkedList`. All accesses to the queue are *synchronized*, making the channel *thread-safe*. Method `totalCount` is implemented by maintaining a `long` value, also guarded by the synchronized blocks.

Because it uses a single lock to access the entire queue and the counter, `SimpleChannel` suffer from two drawbacks:

- There is no parallelism among producers: While a producer is adding to the queue, all the other producers are blocked.
- There is no parallelism between producers and the consumer: A producer blocks the consumer (and all the other producers) and the consumer blocks all the producers.

3 `MultiChannel` class

In order to alleviate these problems, a better channel implementation can use several independent input queues, all implemented as linked lists. Each queue is synchronized on a different monitor (e.g., the queue itself), which allows producing threads to add to different queues at the same time. The channel also uses a single output queue, implemented as a linked list and locked by its own monitor. The algorithms of methods `put`, `get` and `totalCount` are then as follows:

- `put(x,i)` adds a value into queue number `i`. It does not lock any other input queue, the output queue or any global data structure. It also does not impact the count returned by `totalCount`.
- `get()` removes a value from the output queue and returns it, if the output queue is not empty. In this case, the method does not touch the input queues, allowing full parallelism between producer and consumers.
- If the output queue is empty when `get` is called, the content of the input queues is dumped into the output queue and the input queues are all emptied. The number of elements in the input queues is also added to the total count at this stage. This step requires locking all the input queues (although not necessarily at the same time).

[3] A more realistic channel would offer a way to *block* the consuming thread when the channel is empty. Such synchronization strategies will be explored later. In this assignment, the consuming thread is *spinning* when the channel is empty.

- If no content is found in the input queues at the previous stage, the `get` method returns `null`. Note that, due to concurrency, there might be content in the input queues by the time the method returns `null`. However, implementations will not return `null` if there was content in the input queues at the time the `get` method was called.

This implementation allows more parallelism among threads. It also slightly changes the semantics of method `totalCount`, which now returns an approximation of the number of elements added to the channel (a lagging lower bound).

4 `NoCopyMultiChannel` class

A possible drawback of the `MultiChannel` implementation is that it requires copying the contents of the input queues, even though these queues are then cleared and their contents discarded (there is no method in Java to concatenate two linked lists). If input queues are large, this copying could negatively impact the performance of the channel.

Class `NoCopyMultiChannel` implements a variant of `MultiChannel` in which lists are never copied. The strategy is as follows:

- As long as the output queue is not empty, method `get` simply polls from it.
- If the output queue is empty when `get` is called, one of the input queues becomes the output queue (no copy) and this input queue is replaced with a fresh new list. The total count of elements added is updated at the same time (as with `MultiChannel`, it is lagging and updated in batches).
- If the chosen input queue was empty, method `get` returns `null`. Note that, as a consequence, this implementation loses the property of `MultiChannel` that method `get` called on a non-empty channel cannot return `null`.

Which input queue becomes the output queue does not matter much, as long as there is enough fairness (i.e., no input queue is ignored forever). Class `NoCopyMultiChannel` should implement a simple *Round-robin* strategy: Move to the next input queue every time method `get` is called on an empty output queue.

5 `ChannelPerf` class

The final task in the assignment is to write a class `ChannelPerf` that sets up experiments to compare the performances of `SimpleChannel`, `MultiChannel` and `NoCopyMultiChannel`. Measuring performances of parallel Java programs is tricky. While writing `ChannelPerf` keep in mind the following:

- Your development machine may not have enough parallelism to exercise channels in meaningful ways. Make sure you run your program on a true parallel machine like `c1`.
- In the case of channels, the two main measures of interest are throughput and latency:
  - *throughput*: the number of messages that go through the channel per unit of time.
  - *latency*: the amount of time it takes for a message to go through.

  Ideally, a good channel should have high throughput and low latency. Be sure to measure both in your experiments.
- Some channel implementations may behave well under some circumstances and not so well under others. Be sure to vary the parameters of your experiments, such as:
  - *The number of threads.* You should try low numbers, numbers equal or close to the number of cores and high numbers.
  - *The amount of contention.* The amount of time threads spend *not* using the channel (which you can simulate using randomized calls to `Thread.sleep` for instance) will impact the performance of a channel. Make sure to run experiments with low, moderate and high contention.

  - *The relative speed of producers and consumer(s).* In particular, if the consumer is fast, the channel will be empty or almost empty most of the time and queues will be small. Conversely, if the consumer is slow, the channel can grow and reach unreasonable sizes, potentially filling up the entire memory of the JVM.

- A Java Virtual Machine (JVM) can be a very complex piece of software. In particular, the HotSpot JVM we commonly use relies on a parallel garbage collector and a Just-In-Time compiler (JIT). These can potentially get in the way and make observed behaviors more difficult to interpret. If you suspect that this is the case, there are options to turn off the JIT. You should also give your JVM enough memory so garbage collection doesn't mess up things too much. (You can use the `-verbose:gc` option of Java to see what the GC is doing.)

6 Report

In addition to completing the code of the channel classes, students must produce a report, in pdf format, that describes design and testing strategies, test results and performance evaluation. Specifically, this report *must* discuss:

1. What tests were used to show that the channel implementations are thread-safe? How many threads were used? What hardware was used? What is the behavior of these tests when some of the `synchronized` are removed from channel code?? Were you able to make non thread-safe implementations fail?

2. What is the synchronization strategy of the thread-safe channel implementation `MultiChannel`? Why does it make the implementation thread-safe?

3. What is the synchronization strategy of the thread-safe channel implementation `NoCopyMultiChannel`? Why does it make the implementation thread-safe?

4. What is the strategy used in `ChannelPerf`? How are the parameters described above controlled (e.g., number of threads, contention, relative producer/consumer speeds)? Is `Thread.sleep` used? Is randomness used? How are throughput and latency calculated?

5. How do the three channel implementations perform in `ChannelPerf` experiments? Under which circumstances does one implementation perform better than another in throughput and/or latency? What hardware/software combinations were used for these experiments?

6. How do you explain the observed performances in `ChannelPerf`? Can you come up with a reasonable explanation for what you observed? Do some behaviors look mysterious and/or counter-intuitive?

7. The termination strategy used in `WordCountApp` does not work with `NoCopyMultiChannel`. Explain why. Propose an alternative strategy for termination and implement it. Explain why the new strategy works with all three types of channels.

Notes:

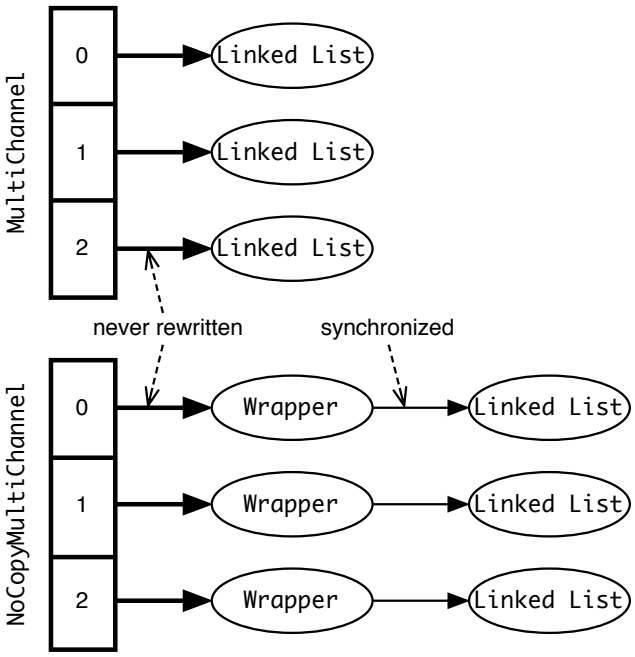
- Public classes to be implemented are described at: `http://www.cs.unh.edu/~cs735/Programming/`. Public interfaces cannot be modified (i.e., you cannot add or remove a public or protected method, constructor, field or class, nor modify the checked exceptions thrown by a public or protected method or constructor). You can add private and package-private classes, methods, fields, etc.

- Do not use `java.util.concurrent` for this assignment. This first assignment illustrates "do-it-yourself" concurrent programming in Java. Future assignments will use better code by relying on the library (e.g., blocking queues for channels, `AtomicLong` for counters, thread pools for parallelism, etc.).

- It is important that `MultiChannel` and `NoCopyMultiChannel` have no single point of contention: If all the producing threads need to go through the same lock in order to access their queue, it defeats the purpose of having multiple queues.

  In the case of `MultiChannel`, the queues can be stored in a final array (or `ArrayList`) of linked lists. As long as the values in this array (the linked lists) are never rewritten, it is safe to read them without any locking (for reasons that are subtle and will be discussed later in the course of the class).

  The case of `NoCopyMultiChannel` is trickier because the linked lists need to be replaced with fresh lists as part of the `get` algorithm. One solution is to keep a final array of wrappers that contain the linked lists: The wrappers are never rewritten and the lists inside the wrappers can be guarded by individual locks (one per wrapper), thus avoiding a central point of contention.



- Make sure the report discusses all the specified points using a clear language. Diagrams and plots can be included to help clarify things. This report *must* be in a single pdf file, submitted in *MyCourses*. It will count toward 40% of the assignment grade.