CS 505 Exercise 3 - Evaluation of Different Linked-list Based Set Implementations

Due: 02/10/2016 Wed. 11:59 PM

1 Overview

The goal of this exercise is for you to get familiar with different more or less fine-grained locking schemes using the example of a sorted linked-list based set. You will need to implement the following different schemes:

- Coarse-grained locking
- Hand-over-hand locking
- Optimistic with validation

These schemes have all been discussed in class. You will also need to write a tiny benchmark to evaluate and compare the performance of different implementations.

Due to time limit, this exercise could be done as a group exercise, of size no more than 2. If doing in groups, only one submission is needed for each group. Do remember to write all the names in the submitted PDF report.

2 Requirements

We will be using Java in this exercise. Java is very easy to learn, and we will not be using many of its advanced features. For those who are not very familiar with Java yet, here is the official documentation: `https://docs.oracle.com/javase/tutorial/`. In particular are the sections about locks and threads, you will be using them in this exercise.

If you are not very familiar with Java yet, it might not be a bad idea to start with some IDE support, e.g., Eclipse, IntelliJ. Just make sure your code could compile on XINU machines (This is generally true due to the guarantee provided by Java).

We will be using "Java SE 8", but for those code used in this exercise, some older version such as 1.7 is not very different.

2.1 Different Linked-list Based Set Implementations

Below is the interface you are going to implement for this linked-list based set, for simplicity, we just assume the elements in the set are of type *int*:

```
public interface IntSet {
  // These methods need to be thread-safe (i.e. well-synchronized).
  public boolean insert(int x);
  public boolean remove(int x);
  public boolean contain(int x);
```

}

The return values of each function are following those in the official document:

insert() : Return true if this set did not already contain the specified element.
remove() : Return true if this set contained the specified element.
contain() : Return true if this set contains the specified element.

It is suggested to first implement a sequential version, (and of course, ensure its correctness). And only after that adapt it to using different concurrent schemes. After all, these variations will share most of the codes.

2.2 Benchmark for Evaluation

To evaluate and compare performance of different schemes, you will write a simple benchmark to spawn a bunch of threads, and compute the throughput of each scenario. Collect the statistics, describe and analyze them in your report.

Ideally, the benchmark should be configurable:

-t Number of threads: e.g., 1, 2, 4, 6, 8
-u Update ratios: what percentage, ranging from 0 to 100, of the operations are updates (i.e., *insert()/remove()*). E.g., 0, 10, 100. For simplicity, we just let there be half *insert()* and half *remove()*.
-i Initial list size: e.g., 100, 1k, 10k. It means how many elements are already in the set when experiment begins. We also set the range of every element to be twice the size of initial-list-size. This range also applies to function parameters invoked later on in the benchmark. For example, if initially there are 100 elements, then each of these 100 elements is randomly chosen from $[0, 200)$, so is the parameters to invoke any function later on.
-d Duration of the executing time, in the unit of milliseconds.
-b Which scheme to test, could be "coarse", "hoh", or "optimistic".

Hence, some command like

```
> java ex3.Main -t 8 -d 3000 -u 100 -i 10000 -b coarse
```

means to run the benchmark against Coarse-grained Locking implementation about 3 seconds of 8 threads, with all operations being *insert()* or *remove()*, initially the set has 10000 elements, all values are randomly chosen from $[0, 20000)$.

So here is a basic outline of what a benchmark does in this exercise: (you don't have to exactly follow this, though)

1. Set up the environment according to configuration.
   (a) Prepare the Threads in Java to simulate random accesses to the shared set.
   (b) Populate the set with specified amount of random elements.
2. Run the benchmark against one particular implementation for the specified duration. Each thread is just repeatedly randomly choosing (according to update ratio) next method to call, from *insert()*, *remove()*, and *contain()*. The values to pass in are also randomly chosen from the range.
3. Print out the collected statistics.

2.3 Collecting Statistics

During benchmark execution, you shall collect statistics for the set implementation currently being tested.

The throughput is computed as Num-of-finished-calls/elapsed-time op/s. You could also record more precisely about

- how many calls were success (returned true) for each method;
- how many calls were failed (returned false).

Collect the statistics of running the benchmark against each implementation for the following scenarios:

- Number of threads fixed to 8.
- Duration fixed to 5 seconds.
- Update ratio choseon from $10\%, 100\%$.
- Initial list size chosen from $100, 1000, 10000$.

Thus there are at least 3 (implementations) $\times$ 2 (ratios) $\times$ 3 (initial size) runs to test.

Collect the throughput of each scenario above in your report **ex3.pdf** (which you shall submit as well). Describe in the report about your observation, explain the throughput under different settings, and the performance of different implementations under multiple identical settings.

3 Submit Instructions

Due to time limit, this exercise could be done as a group exercise, of size no more than 2. If doing in groups, only one submission is needed for each group. Do remember to write all the names in the submitted PDF report.

ONLY submit electronically using *turnin* command. Please submit your code as well as the report.

Electronic turn-in instructions: (If you are not familiar with these commands/apps yet, do remember to save enough time for submitting.)

1. To use *turnin*, you'll need to log in to any CS machines (e.g. those in XINU lab) using your purdue account. You can use *ssh* command (Mac/Linux) or *PuTTY* (Windows) to do this remotely at home. To transfer files to/from CS machines, you could use *scp* command (Mac/Linux) or *WinSCP* (Windows).
2. Now assume the file to submit is located at "~/ex3/". Go to the same directory of it, in this case it's your home directory "~/".
3. Clean the generated binary files, debug files before submission.
4. Type the following command: **turnin -c cs505 -p ex3 ./ex3/**
5. After submission, if you want to validate it's indeed submitted, you could use command: **turnin -v -c cs505 -p ex3** to do so.

Submission to "ex3" will be disabled after deadline. For late submission, please submit to "ex3late". Thus the command is: **turnin -c cs505 -p ex3late ./ex3/** and also **turnin -v -c cs505 -p ex3late** for validation.