CS 511: Assignment #4
Java Concurrency Experiments

Due Sunday, November 2 at 11pm.

1 Assignment

In this assignment we will see how different approaches to concurrency in a Java program result in different performance.

The task is to analyze an XML file to determine which tokens appear most often. Test files that contain XML content for a small part of Wikipedia have been posted.

Also posted with this assignment are files `assgn4.jar` and `TokenCount.java`. JAR file `assgn4.jar` contains 5 source files: `Page.java`, `Pages.java`, `PoisonPill.java`, `WikiPage.java`, and `Words.java`. Do not change these 5 files. (Different systems have different utilities for extracting files from a JAR collection. On `linux-lab` you can use the command "`jar xvf assgn4.jar`")

File `TokenCount.java` contains a program that parses an XML file first into pages then into tokens. Next it counts how often each token appears then finally sorts the frequency counts and displays the 30 most common tokens. All these actions are done sequentially. The time to parse and count is computed and displayed. The sort operation is not part of the timing.

You must write 4 new versions of TokenCount. Each version should: display the number of processors available; time how long it takes to parse and count and display the time; and finally sort the overall token frequency map then display the 30 most common tokens.

1. The first version divides the work into two threads, a producer and a consumer. The producer thread parses the XML and pushes pages onto a shared queue. The consumer thread removes pages from the shared queue, divides each page into tokens, and keeps a running count of token frequency across all pages.

   The main class `TokenCount` should start and join both threads then sort the token frequency map and display the 30 most common tokens. Record

system time immediately before starting the first thread. Record system time again immediately after joining the second thread. That is: time the parsing and counting actions, not the sorting action.

Use an `ArrayBlockingQueue` of size 100 as the shared queue.

2. The second version operates the same way as the first version except it uses N-1 consumer threads on an N-processor system. With several consumer threads simultaneously accessing the token frequency map, you will have to synchronize access to the map.

   Use `ExecutorService` to create a "cached thread pool" of consumer threads. Record system time immediately before starting any threads. Record system time again immediately after terminating all threads.

3. The third version operates the same way as the second version except that instead of locking the entire map before each update, this version uses a `ConcurrentHashMap`. Java's `ConcurrentHashMap` class strives to improve concurrency by using per-bucket locking.

   Record system time immediately before starting any threads. Record system time again immediately after terminating all threads.

4. The fourth version operates the same way as the third version except that each consumer thread updates its own private concurrent hash map. After each consumer thread has processed its last token, it adds the contents of its private frequency map to an overall frequency map.

   Record system time immediately before starting any threads. Record system time again immediately after terminating all threads. The action of merging each thread's map into the overall map should be included in the time computation but sorting the overall map should not be included.

The producer should indicate to the consumer that there will be no more pages to process by putting a special "poison pill" item into the shared queue. When the consumer takes the poison pill off the queue, it knows to shut down. In cases where there are N consumer threads, the producer should put N poison pills onto the queue after it has put all XML pages onto the queue.

1.1 Time Analysis

Once you have a version working, run it at least 3 times and take the average of the run times. Use the same inputs to test all versions.

Write a report that includes the average time for each version as well as your explanation for why the times vary as they do; i.e., why are certain versions faster or slower than others? Your report should include:

- Name of test file
- Number of pages parsed
- Number of consumer threads (if applicable)
- Number of processors on your test system
- Average execution time for each version (the sequential version you have been given plus the 4 new versions you wrote)
- Your explanation of the results

The report should be about one page long.

1.2 Test Data

The following XML files have been posted:

- `http://www.cs.stevens.edu/ djd/CS511/few.xml`
- `http://www.cs.stevens.edu/ djd/CS511/many.xml`

File `few.xml` contains about 7000 pages; `many.xml` contains hundreds of thousands. The program will crash with an uncaught exception if you ask for more pages than exist in the file. Don't worry about this; just keep the requested number of pages under the limit for that file.

1.3 What to Submit

There are many source files plus the report to hand in. Collect all these files into a single zip or tar archive and submit that single file. Include the report plus a README file that explains which source files build which version of the program.

Do not include in your zip/tar archive any class files, backup files, or anything that's not required. If you submit extraneous files or if the TAs have trouble understanding your submission, points will be taken off. For grading, the TAs should be able to read your README file and know immediately how to build all versions of your program.

2 Submission Instructions

You must work alone on this assignment. Submit a single zip or tar file via Moodle by the indicated date and time. Moodle will shut you out at the deadline. Since your clock and Moodle's clock may differ by a few minutes, be sure to submit at least several minutes before the deadline.

Please note: *late work is not accepted; late submissions will not be graded and will receive a score of zero.*