CSc 422 Assignment 3

Play Fair

CSc 422 Assignment 3

This threding assignment is a bit of an investigation. You will find on this page an implementation of the readers/writers problem using semaphores in C++. The links are near the bottom of the page, including the full package download. The solution works, but, as noted in class, it tends to favor readers. When there are many readers, writers tend to starve.

The program contains the readers/writers solution (which is not a lot of code) and a test application (which is most of the code). The test application maintains a shared linked list of nodes containing a string and a value, and methods for manipulating the shared structure. There are two update methods: update adds or increments an item, and remove_node does just that. There are three reading methods, one that just finds the value associated with a name, one that takes an average of the values, and one that finds their range. In all cases, the values are chosen at random. Names are chosen at random from a pre-defined list.

The program contains three types of threads: a reading thread, a writing thread, and a monitor thread. The reading and writing thread each repeatedly choose at random a method of the appropriate type and execute it, printing (or maybe not printing) the result. (The types are not chosen with equal probabilities. The item lookup is a more likely reader, update is a more likely writer.) The progress monitor periodically prints a message showing how far each of the other threads has run. The program starts several reader and several writer threads, and one monitor thread. The numbers can be controlled from the command line. When run without any parameters, it runs 10 each of the reader and writer threads, and each thread performs 10 operations an exits, for a total of 200 operations on the shared data structure. It looks something like this, though I'll omit much of the lengthy output:

```
[bennet@bennet readers_writers]$ ./rw3
Running 10 read threads and 10 update threads, 10 operations each
reinitialized is not listed
international is not listed
amalgomated is not listed
embalmed is not listed
No items for range computation.
fiduciary is not listed
...
Range is 44 to 265
discombobulated given 84
consolidated could not be found
amalgomated deleted
reinitialized given 39
fiduciary could not be found
embalmed given 23
discombobulated deleted
confederated deleted
missing given 52
international deleted
embalmed deleted
reinitialized created with 22
amalgomated created with 8
amalgomated deleted
exasperated given 13
amalgomated created with 10
discombobulated could not be found
reinitialized given 84
confederated created with -6
=== 100 reads (100%), 100 writes (100%) ===
```

The read operations don't do much at first, because there is nothing in the list until the first update runs, and even then, find chooses something at random to find, and it may not be there. The last line is generated by the monitor thread. It runs periodically, and 20 threads of 10 operations is not usually take long enough to get more than the final report out of it.

The program is designed to measure progress of reading v. writing, but to do that you need to use larger numbers. It also helps to use the `-q` parameter, which basically turns off output except from the progress thread. For instance,

```
[bennet@bennet readers_writers]$ ./rw3 -q -n 30000 -r 5 -w 5
Running 5 read threads and 5 update threads, 30000 operations eac
h
=== 9984 reads (6.656%), 2816 writes (1.87733%) ===
=== 31744 reads (21.1627%), 7424 writes (4.94933%) ===
=== 55808 reads (37.2053%), 12288 writes (8.192%) ===
=== 83200 reads (55.4667%), 16128 writes (10.752%) ===
=== 104704 reads (69.8027%), 20480 writes (13.6533%) ===
=== 126720 reads (84.48%), 25600 writes (17.0667%) ===
=== 149696 reads (99.7973%), 35072 writes (23.3813%) ===
=== 150000 reads (100%), 53760 writes (35.84%) ===
=== 150000 reads (100%), 73984 writes (49.3227%) ===
=== 150000 reads (100%), 93952 writes (62.6347%) ===
=== 150000 reads (100%), 114432 writes (76.288%) ===
=== 150000 reads (100%), 134912 writes (89.9413%) ===
=== 150000 reads (100%), 150000 writes (100%) ===
```

In this run, we specified `-q`uiet output, that there should be five reader threads and five writer threads, and that each one should perform 30000 operations. The output shows the progress of reading and writing. It also shows that reading progresses faster, since the reader/writer solution favors readers. The effect is considerably less when there are fewer threads:

```
[bennet@bennet readers_writers]$ ./rw3 -q -n 150000 -r 1 -w 1
Running 1 read threads and 1 update threads, 150000 operations ea
ch
=== 9984 reads (6.656%), 8448 writes (5.632%) ===
=== 26368 reads (17.5787%), 22272 writes (14.848%) ===
=== 42496 reads (28.3307%), 36864 writes (24.576%) ===
=== 59392 reads (39.5947%), 50944 writes (33.9627%) ===
=== 76288 reads (50.8587%), 65536 writes (43.6907%) ===
=== 93184 reads (62.1227%), 80384 writes (53.5893%) ===
=== 110080 reads (73.3867%), 94976 writes (63.3173%) ===
=== 126208 reads (84.1387%), 109056 writes (72.704%) ===
=== 142080 reads (94.72%), 123136 writes (82.0907%) ===
=== 150000 reads (100%), 150000 writes (100%) ===
```

and greater when more:

```
[bennet@bennet readers_writers]$ ./rw3 -q -n 10000 -r 15 -w 15
Running 15 read threads and 15 update threads, 10000 operations e
ach
=== 4096 reads (2.73067%), 0 writes (0%) ===
=== 18176 reads (12.1173%), 0 writes (0%) ===
=== 30976 reads (20.6507%), 0 writes (0%) ===
=== 42752 reads (28.5013%), 0 writes (0%) ===
=== 55552 reads (37.0347%), 0 writes (0%) ===
=== 69120 reads (46.08%), 0 writes (0%) ===
=== 81408 reads (54.272%), 0 writes (0%) ===
=== 94976 reads (63.3173%), 0 writes (0%) ===
=== 107264 reads (71.5093%), 0 writes (0%) ===
=== 119808 reads (79.872%), 0 writes (0%) ===
=== 133888 reads (89.2587%), 0 writes (0%) ===
=== 146448 reads (97.632%), 0 writes (0%) ===
=== 150000 reads (100%), 15360 writes (10.24%) ===
=== 150000 reads (100%), 34560 writes (23.04%) ===
=== 150000 reads (100%), 53760 writes (35.84%) ===
=== 150000 reads (100%), 72960 writes (48.64%) ===
=== 150000 reads (100%), 94464 writes (62.976%) ===
=== 150000 reads (100%), 115200 writes (76.8%) ===
=== 150000 reads (100%), 134400 writes (89.6%) ===
=== 150000 reads (100%), 150000 writes (100%) ===
```

The project is to modify the program to make it fairer to writers, so the output is more along these lines:

```
[bennet@bennet readers_writers]$ ./rw3 -q -n 10000 -r 15 -w 15
=== 4096 reads (2.73067%), 4096 writes (2.73067%) ===
=== 15616 reads (10.4107%), 15360 writes (10.24%) ===
=== 23296 reads (15.5307%), 26880 writes (17.92%) ===
=== 34816 reads (23.2107%), 37120 writes (24.7467%) ===
=== 42496 reads (28.3307%), 46080 writes (30.72%) ===
=== 50176 reads (33.4507%), 57088 writes (38.0587%) ===
=== 61440 reads (40.96%), 65280 writes (43.52%) ===
=== 69376 reads (46.2507%), 76288 writes (50.8587%) ===
=== 77312 reads (51.5413%), 84992 writes (56.6613%) ===
=== 88064 reads (58.7093%), 95488 writes (63.6587%) ===
=== 96256 reads (64.1707%), 103936 writes (69.2907%) ===
=== 104192 reads (69.4613%), 114432 writes (76.288%) ===
=== 113408 reads (75.6053%), 123392 writes (82.2613%) ===
=== 123136 reads (82.0907%), 133888 writes (89.2587%) ===
=== 131072 reads (87.3813%), 143104 writes (95.4027%) ===
=== 146192 reads (97.4613%), 150000 writes (100%)
=== 150000 reads (100%), 150000 writes (100%) ===
```

Here, both operations make progress more evenly (in fact the writers actually pull ahead slightly, though that's not especially desired).

Create a version of the program in which both readers and writers will proceed at similar rates. When run with the same numbers of reader and writer thread, the percentage progress should of each side should be similar at each report. When the number of readers and writers is very different, the best result would still be similar progress (as a percentage) on each side, but that might not be possible. If there are lots of readers and just a few writers, the writers may progress faster just because they have less to do. If this is the case, a good solution should be symmetrical, so that giving either side fewer threads speeds it up by about the same amount. Most points will be just for trying something reasonable without breaking the program, but do try to create a good solution.

The file `rw3.cpp` contains the test application. You probably don't want to change this. The file `readerwriter.h` contains the procedures that readers and writers execute when starting and finishing reading. This is the place to change. The semaphore implementation is also there (`semaphore.h`/`semaphore.cpp`), and could perhaps be changed.

Building the Program

To build the program on Linux, simply unpack the zip into a directory, go there, and say `make`. The executable is `rw3`, which just means I was experimenting a bit. Macs should work the same if g++ is installed, but I don't have one, so I can't try it. Type the name to run the program. The command line flags are as discussed above. Say `rw3 -h` to get a summary of them.

The program will build on Windows with CodeBlocks. First, check the compiler flags under Settings/Compiler/Compiler Settings/Compiler Flags. Make sure to tick the box that says "Have g++ follow the C++11 ISO C++ language standard [-std=c++11]". It may already be set, but I'm not sure what the default is. Then,

1. Download and unpack the zip file.
2. Create a project. File/New/Project/Console Application, then press Go. Choose a C++ application, and name it and store it where you like. Probably makes the most sense to store it inside the folder where you unpacked the program.
3. Open each of the code files (.cpp and .h), and add them to the project. Right click its tab, and add to the current project.
4. The project will appear in the left-hand pane. Expand it.
5. Find the file main.cpp which the program has added automatically in an attempt to helpful. Right click on it and remove it from the project.
6. You should be able to build the project from the Build menu.

To run the program, you should open a command window, go do the directory where the executable is located, and run it from there by typing its name. This will give you easy control of the command-line parameters so you can experiment with the program's behavior.

The program will behave differently each time it is run because of the random numbers it uses, and because the threads will be scheduled differently. It may also behave differently on different systems (or even different C++ compilers) because of differences in the scheduling. If you find a system that exhibits some sort odd performance behavior, please let me know about it.

Submission

When your program works, and is properly commented and indented, submit it over the web using this form. You probably only want to send one file, the modified reader/writer entry methods, but you can send other files if you happened to have changed them. Note that they are all optional on the submit form.