John Gangemi

Raj Patel

Project 4: Reader/Writer Locks Report

Problem being addressed:

The problem that is being addressed with this project is to create a reader/writer lock in which the writers do not get starved. Also making sure that multiple readers are able to be in the critical section, and that there are no readers in the critical section along with writers.

To address the problems presented above the use of semaphores is crucial, thus it is the main focus of this project.

Explanation of solution:

The code for the readers/writers problem provided in section 31.5 of the text book Operating Systems was used as a starting point to address the problems proposed in this project. The code from 31.5 while allowing concurrent readers in the critical section, will also cause writers to be starved. In order to address this fundamental issue two additional semaphores were added to the rwlock_t data structure. The first semaphore added (named write_mutex) allows mutual exclusion for the writer threads and their controlling variable. The second semaphore (named readlock) refused reader threads from accessing the critical section when a writer held the lock (i.e. writer in the critical section). Lastly a single integer value (named writers) was also instantiated to track the number of writer threads waiting to obtain the write lock.

Suppose the scheduler creates a single write and read thread where the read thread has been scheduled first. For this instance the read thread would obtain the “readlock” that would normally be unavailable if a writer held such lock , the next lock obtained by the read thread is the read_mutex lock which provides mutual exclusion to the readers variable and if necessary obtain the write lock such that no thread can write to the critical section. Upon the reader thread exiting the critical section it obtains the read mutex lock once more to alter the controlling variable and if necessary releases the write lock so threads can now write to the shared resource.

While the read thread was executing the write thread was obtaining mutual exclusion to the controlling variable (named writers) and waiting for the readlock to become available thus potentially blocking new readers from accessing the critical section. Once the read has released the writelock the write thread can obtain the writelock and enter the critical section subsequently exiting the critical section and releasing the writelock. If appropriate the writer will release the readlock to allow reading threads to enter the critical section.

Explanation of testing:

The way the code is implemented the user is able to type in how many reads and writes they want via use of command-line arguments in the terminal. One observation that was made is that when the number of readers and writers was relatively small (2 readers, 2 writers) and the code ran multiple

times the results had very few changes. However when the number of writers and readers was large by comparison there was more variability in the way the threads were scheduled by the scheduler.

For this project we went through multiple iterations of the code. The initial observation that was made is if there were two different loop structures used to create either the read or write threads then during runtime the creation of threads reflected the order of execution of code in the program. Such that the first controlling loop executed will almost always create all of its threads before beginning execution of the second loop.

We tried to resolve the unfair scheduling of threads for a very small number of readers and writers, especially when there is an equal amount amongst both types of threads. Our resolution for the problem was using a random number generator for which if the number was from 1 to 5 it would create a reader whereas if the number was from 6-10 it would create a writer. The structure of this code provided no difference in execution when compared to explicitly declaring the creation of a reader or writer before the other. To that extent, our code does explicitly declare a creation of a reader thread before the creation of a writer thread (however the threads are created inside of a singular loop to avoid the problem discussed above).

Ultimately, while we attempted to resolve the issue of unfair scheduling for a small number of threads, the creation of “pthreads” during run time is still highly dependent upon the scheduler.

Examples:

Below are print statements from various runs of the program:

We used the pthread_self function to obtain the number of each pthread which is the 14 digit values after the word Thread in the images. Following the “thread and the thread # ” two sets of messages can occur

- The messages “wants to write” and “wants to read” showcase what each thread wants to accomplish

Or

- “entered critical section and” followed by either “read” or “wrote”(depending on what the thread wants to acomplish) and the value of the shared resource

Multiple readers in the critical section

```
**Shared resource is initialized to 100**
**Writers increment shared resource by 100**
  Readers: 10 / Writers: 5

Thread 47653813716736 wants to read.
Thread 47653815817984 wants to write.
Thread 47653820020480 wants to write.
Thread 47653813716736 entered critical section and reads: 100
Thread 47653822121728 wants to read.
Thread 47653817919232 wants to read.
Thread 47653826324224 wants to read.
Thread 47653824222976 wants to write.
Thread 47653815817984 entered critical section and wrote: 200
Thread 47653830526720 wants to read.
Thread 47653832627968 wants to write.
Thread 47653834729216 wants to read.
Thread 47653836830464 wants to read.
Thread 47653838931712 wants to read.
Thread 47653828425472 wants to write.
Thread 47653841032960 wants to read.
Thread 47653820020480 entered critical section and wrote: 300
Thread 47653843134208 wants to read.
Thread 47653824222976 entered critical section and wrote: 400
Thread 47653832627968 entered critical section and wrote: 500
Thread 47653828425472 entered critical section and wrote: 600
Thread 47653822121728 entered critical section and reads: 600
Thread 47653817919232 entered critical section and reads: 600
Thread 47653826324224 entered critical section and reads: 600
Thread 47653830526720 entered critical section and reads: 600
Thread 47653834729216 entered critical section and reads: 600
Thread 47653838931712 entered critical section and reads: 600
Thread 47653836830464 entered critical section and reads: 600
Thread 47653841032960 entered critical section and reads: 600
Thread 47653843134208 entered critical section and reads: 600
```

This example shows that multiple files wanted to read (as showcased by the red highlights) and that they were all able to enter the critical section at the same time and read (as showcased by the yellow highlights).

No readers in the critical section with writer

```
**Shared resource is initialized to 100**
**Writers increment shared resource by 100**
  Readers: 4 / Writers: 2

Thread 47693603813120 wants to read.
Thread 47693603813120 entered critical section and reads: 100
Thread 47693605914368 wants to write.
Thread 47693605914368 entered critical section and wrote: 200
Thread 47693608015616 wants to read.
Thread 47693612218112 wants to read.
Thread 47693610116864 wants to write.
Thread 47693612218112 entered critical section and reads: 200
Thread 47693608015616 entered critical section and reads: 200
Thread 47693614319360 wants to read.
Thread 47693610116864 entered critical section and wrote: 300
Thread 47693614319360 entered critical section and reads: 300
```

This example shows that until the writer is done writing there are no readers in the critical section. The red highlights shows that a thread wanted to write and then entered the critical section and wrote. Once it was done (as showcased by the incrementing of the shared resource) the readers that wanted to read were able to enter the critical section and read (as showcased by the yellow highlight).

Writers do not starve

```
**Shared resource is initialized to 100**
**Writers increment shared resource by 100**
  Readers: 5 / Writers: 8

Thread 47776679372544 wants to write.
Thread 47776683575040 wants to write.
Thread 47776681473792 wants to read.
Thread 47776679372544 entered critical section and wrote: 200
Thread 47776687777536 wants to write.
Thread 47776687777536 entered critical section and wrote: 300
Thread 47776685676288 wants to read.
Thread 47776691980032 wants to write.
Thread 47776689878784 wants to read.
Thread 47776683575040 entered critical section and wrote: 400
Thread 47776698283776 wants to read.
Thread 47776694081280 wants to read.
Thread 47776700385024 wants to write.
Thread 47776702486272 wants to write.
Thread 47776696182528 wants to write.
Thread 47776691980032 entered critical section and wrote: 500
Thread 47776704587520 wants to write.
Thread 47776700385024 entered critical section and wrote: 600
Thread 47776702486272 entered critical section and wrote: 700
Thread 47776696182528 entered critical section and wrote: 800
Thread 47776704587520 entered critical section and wrote: 900
Thread 47776681473792 entered critical section and reads: 900
Thread 47776685676288 entered critical section and reads: 900
Thread 47776689878784 entered critical section and reads: 900
Thread 47776698283776 entered critical section and reads: 900
Thread 47776694081280 entered critical section and reads: 900
```

This example shows that even though writers came in at different times they were not starved as showcased by the yellow highlights.

No more than one writer in the critical section

```
**Shared resource is initialized to 100**
**Writers increment shared resource by 100**
  Readers: 4 / Writers: 2

Thread 47138771744512 wants to read.
Thread 47138773845760 wants to write.
Thread 47138771744512 entered critical section and reads: 100
Thread 47138775947008 wants to read.
Thread 47138780149504 wants to read.
Thread 47138778048256 wants to write.
Thread 47138782250752 wants to read.
Thread 47138773845760 entered critical section and wrote: 200
Thread 47138778048256 entered critical section and wrote: 300
Thread 47138775947008 entered critical section and reads: 300
Thread 47138780149504 entered critical section and reads: 300
Thread 47138782250752 entered critical section and reads: 300
```

This example shows that there are no more than one writer in the critical section since there is no occurrence where the shared resource is incremented by more than 100 for a single line.

Estimated Time Spent on Project/Roles:

The project took a total of **35 hours** to complete. We individually did research by looking online and reading the Operating Systems book. This gave us an idea on how to start to tackle the reader/writer locks problem. After completing the initial research individually, we meet at the C4 lab and collaborated on potential solutions. We came to a general consensus and John started typing the code. We went through many iterations before coming to an algorithm that we were satisfied with. Finally with John's help I (Raj) typed up the report.

References:

http://lass.cs.umass.edu/~shenoy/courses/fall08/lectures/Lec11.pdf

http://cs.nyu.edu/~lerner/spring12/Read04-ReadersWriters.pdf

http://nob.cs.ucdavis.edu/classes/ecs150-2008-02/handouts/sync/sync-1rwsem.html

http://pages.cs.wisc.edu/~remzi/OSTEP/