Christian Carreras

12/12/2017

CSC 510

Dr. Parson

The Readers Writers Problem

A common problem found within the realm of computer science is synchronization. A subset of that problem which I chose to model is the Readers Writers Problem. This problem specifically deals with the synchronization of writer and reader processes and how they interact with the critical section problem. Under normal circumstances the critical section is mutually exclusive in that only one process should be allowed inside at a time. However, this is known to be entirely suboptimal when considering there are some processes that do not change anything at all in the critical section. Readers are just that, they only read. They do not mutate, only inspect. Therefore, the Readers Writers Problem suggests an alternate approach: concurrency among readers in the critical section. By letting multiple readers in the critical section, synchronization is optimized and performance is increased. Concurrency among readers and variations of the Readers Writers Problem is what I sought to model in this project.

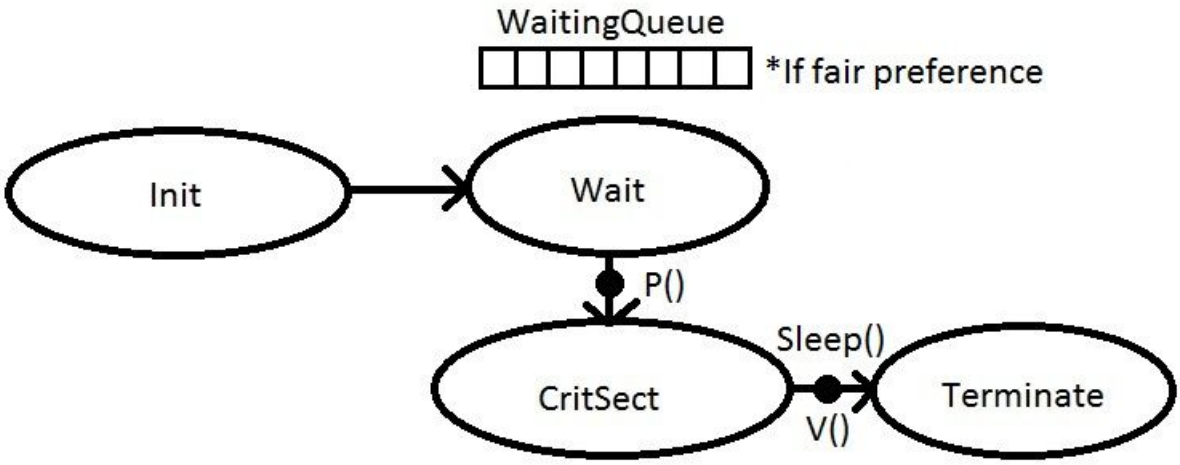
I was inspired to pick the Readers Writers Problem as a topic after taking a course with Dr. Spiegel. In that course he talked of the problem and its several incarnations. Now while taking Advanced OpSys I have the opportunity to explore that topic further and learn hands-on. At first I was reading through my old notes to brush up on the topic. Fortunately for me Dr. Spiegel also keeps a handful of examples on his webpage for students to access. From there I made the decision to model the variations that he calls, Fair Preference, Weak Reader Preference, Strong Reader Preference, Weak Writer Preference, and Strong Reader

Preference. I know that there is no computer science standard or convention with these names but I decided to use Spiegel's names since that is what I was exposed to first.

Before I coded any variations of the Readers Writers Problem, I first decided to make the object to run it. I choose C++ for my implementation since it was the language I was most comfortable with. I also chose C++ because it has, along with C, a plethora of libraries dealing with threads, timing, mutexes, atomic variables, condition variables, and so on. I eventually settled with pthreads to do most of the state machine's heavy lifting. It had threads, mutexes, and condition variables which is the majority of what the state machine needed. Having found which libraries I wanted to use in order, I made a header file for the state machine object and a source code file that implemented the object's functions. An object-oriented design for the state machine kept everything modular, readable, and easily understandable. The state machine itself was modeled after the state machines used in previous Operating System course projects. I wanted the execution of the state machine object to run the simulation with multiple threads and iterations and print out a log with enough information to debug errors and get a peek at the inner-workings of the state machine. Threads needed to use the sleep command to both avoid race conditions/deadlock and simulate a process in the critical section. Threads also required an assignment of a reader or writer role pseudo-randomly since the problem cannot be modeled with readers and writers! The simulation itself is timed for data analyzing purposes and to be able to compare different models with each other. After the foundation was laid out with the state machine, I could finally start modeling the five variations to the Readers Writers Problem.

For my first model, I decided to make the fair preference solution since it seemed the easiest to implement. The basis for my state machine that I used for every model was a switch statement. The switch statement represents the states similarly as to what was given in the C++ template but with some changes to better fit the problem at hand. To start, there are four states

inside the state machine. The first state is the initial state call “init”; this is where all threads start. From here the thread will go to the “wait” state. Threads will remain in the wait state until they can access the critical section. A thread can only enter the critical section by acquiring the lock for the critical section mutex or being allowed in concurrently if the thread is a reader. Once in the critical section the thread will sleep for a predefined amount of time and unlock the mutex for the next in line. After leaving the critical section the thread will go to the terminate state. The terminate state is an accept state i.e. the thread has completed its task and reached conclusion. If the thread has iterations left, it will assign a new role and go through the state machine again. Threads will continue to loop through all the states in the STM until it has no iterations left. Below is a picture depicting the state machine I just described with a queue and mutex added.



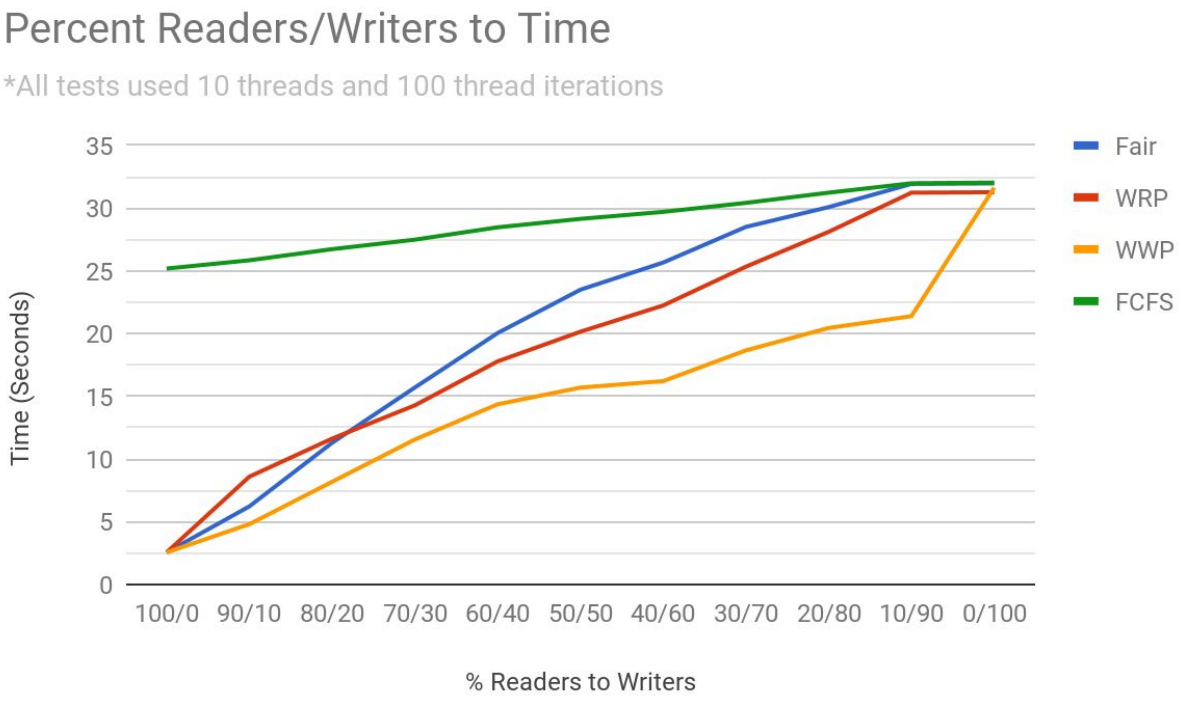
From there I added a the necessary parts such as the critical section mutex, condition variables, and of course the waiting queue. For the fair model, all readers and writers are inserted into a queue and must wait their turn. Once it is their turn they may enter the critical section. But it was not until after this point that I was exposed to the true hero of the reader/writer dilemma:

concurrency. By allowing multiple readers to enter the critical section simultaneously, there is a significant performance boost. Concurrency led to an almost 30% time deduction in my tests versus a non-concurrent solution. As a control, I made a non-concurrency FCFS model to test against just to see the improvements. I needed to know how far concurrency can take the state machine's performance so I finished what was left with the fair preference implementation and continued on with the other variations.

Models that were next on my list to create were Weak Reader Preference and Weak Writer Preference. Weak Reader Preference was quite easy to make as it was almost exactly the same as Fair Preference. The only difference is that Weak Reader Preference does not have a waiting queue. Instead whoever reaches the mutex first acquires it. As far as the critical section goes, the same rules apply. Multiple readers can enter the critical section if they are waiting and there is a reader in the critical section already. Writers must wait until all readers exit the critical section to enter by itself. Since writers edit the shared resource, it has to have mutually exclusive access to the critical section. With two models and a control down, I moved on to Weak Writer Preference. WWP seems to add onto WRP by adding similar bit of code for the writers. Like how once a reader is in the critical section, it stays in the hands of the readers until they all exit, the same now applies to the writers in WWP. If a writer is in the critical section, writers will go next until all writers have been to the critical section. This creates a "flip-flopping" effect where writers will control the critical section for a while, then the readers, and then back to the writers again. The log I set up for the state machine helped immensely in guiding my discovery of the inner workings of these models. The log also showed me that for the "weak" models there is some starvation at play. Writers will be waiting a while in WRP and both readers and writers will be waiting in WWP. I doubt starvation would have any noticeable negative impact in these models except in less than likely scenarios but it is certainly something to be

aware of and prepared for. At some point I was attempting to model Strong Reader Preference and Strong Writer Preference but their implementations were a little more in-depth than time would allow. I really wanted to prove if starvation really became factor in these models but for now I would have to wait on seeing if that is truly the case.

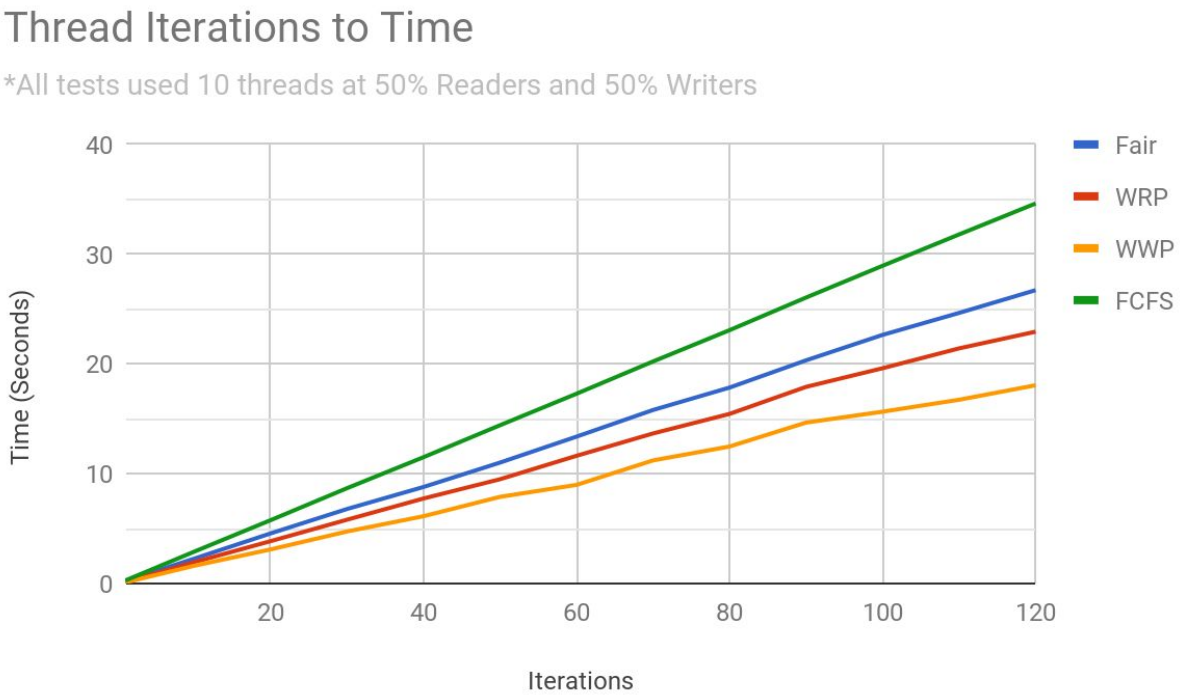
Even with the three out of the five models I originally sought out to make, it was more than enough to test and gather results to show the benefits of concurrency among readers. From here I tested the three models along with the control at varying ratios of readers to writers and varying thread iterations. The results I assembled proved how concurrency was indeed a powerful solution to the Readers Writers Problem. Shown below is the performance of each model plus the control (FCFS) at varying ratios of readers to writers. Weak Writer Preference stood ahead of the pack by having the best performance at every ratio (excluding the 100%'s).



I believe this is due to how it groups all the readers and writers together and lets them in the critical section as a collective. Since all the readers will be together and will have time to

accumulate multiple readers waiting for the writers, all these readers will enter the critical section simultaneously and cut time down by a large margin.

At 100% readers the models with concurrency are much faster than the non-concurrent control by a factor of ten. Now 100% readers is not realistic but to potentially be 10x faster shows how much use concurrency can be in a critical section or synchronization problem. Without concurrency there would be no solution to the Readers Writers problem. While some models perform better than others, they all perform better than the control and by a pretty decent margin. Starvation may be a risk but some models find a nice balance between performance and said risk. Namely in my results, Weak Writer Preference shines the brightest by having the best performance in every single test I conducted. I believe if I ever need to implement a solution to the Readers Writers Problem in the future, I know my go to model. Below I show the performance of each model based on number of thread iterations: *



*I decided to add this extra picture at the end for clarification not page count.