Anna Olkhovskaia
V00832521
CSC464 Assignment 1// *All code ran in Eclipse version 4.9.0*

Readers Writers Problem

Overview/relevance:

The readers writers problem is one of the most common problems that occurs on a daily basis. There are reader and writer threads that want to access the same resource but cannot because if a reader thread is reading something and the writer changes it, it becomes old and non useful information. An example of this could be airline tickets where a customer (the reader) is looking up tickets but by the time they select one they are already sold out, that implies the reader is reading useless and out of date information. That information needs to be locked while the reader is reading it to avoid changes being made. There are three common implementations of the reader/writer problem, I have chosen to compare the reader-preference and writer-preference implementations.

Links to my code: ReadersPreference / WritersPreference

Link to references used: https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem

Results:

Both reader preference and writer preference both ran for 1 minute.

Reader: all reader threads/ 45 writer threads

- Used memory: 44,316 kbytes
- Peak Thread Count: 2009
- Total Thread Count: 2011

Writer: no reader threads/ 49 writer threads

- Used memory: 33,360 kbytes
- Peak Thread Count: 2007
- Total Thread Count: 2011

Analysis:

Correctness:

I went through many different algorithms and implementations of both the reader and writer preferences and based on those algorithms as well as debugging both implementations were correct and displayed correct results. The readers preference solution focuses on letting in all readers as they come and reading the resource all at the same time until all readers have finished. This can allow writers to starve waiting for the resource to be open to them. The second solution which is writers preference has the same idea as the readers except when a writer shows up, they are automatically let in and server first, although unlike the readers they cannot all access the resource at the same time so each writer must lock the resource. This solution benefits the writers and can also lead to starvation of the readers.

Comprehensibility:

Both the readers preference and writers preference solutions are very straightforward and easy to follow. The writers solution does require more steps because it favors the writers and lets them all in before the readers while still having to lock the resource behind each one of them which adds a few extra lines of code. Overall both solutions are fairly short with the readers preference being 73 lines and the writers having an extra few totaling 95. Both of my solutions are in java and have very similar structure to them, but if we were to go off just how easy to understand they are as well as lines of code, the readers preference solution is a little shorter and easier to understand.

Performance:

Overall the performance of both readers preference and writers preference was very similar. I ran both programs for exactly 1 minute and in that time readers preference got through every single reader thread and 45 writer threads, the writers preference got through none of the reader threads and through 49 writing threads. This is because obviously the reader threads took some time to get through in the reader preference implementation, but the upside to that implementation is that since the reader threads can all read at the same time they get through their process at a much higher rate so the possibility of starvation is less than that in writer preference.

Producer Consumer Problem

Overview/relevance:

The producer consumer problem occurs when there is something (a producer) creating data that the consumer is in need of and putting it into a buffer which the consumer takes the data out of. The key concept is that the buffer is of a fixed size and the producer cannot put data into a full buffer and likewise the consumer cannot attempt to take out data from an empty buffer. This is useful for when two systems interact at different rates. For example if a network device was putting data in a buffer and the operating system was taking the data out at its own rate.

Links to my code: Semaphore , Threads

Link to references used: https://www.geeksforgeeks.org/producer-consumer-solution-using-semaphores-java/
https://www.geeksforgeeks.org/producer-consumer-solution-using-threads-java/

Results:

Both programs were run for exactly 1 minute

Semaphore: 59 threads put and retrieved
- Used memory: 9,788 kbytes
- Thread count: 12
- Peak Thread Count: 12
- Total loaded class count: 1577

Threads: 29 threads got and retrieved
- Used memory: 16,975 kbytes
- Thread Count: 12
- Total Thread Count: 12
- Total loaded class count: 1575

Loaded Class count and CPU usage graphs look very similar in both implementations

Analysis:

Correctness:

My solutions are both written with java code, one solution uses semaphores and the other uses threads to solve the producer consumer problem. Both solutions achieve the same results and appear to not have any deadlocks or starvation happening. My second solution which uses threads, has a linked list that acts as a buffer holding at most 3 items, whether the solution using semaphores has a buffer size of 1. Both implementations are synchronized making sure the producer first puts the data into the buffer so then the consumer can take it out. Both solutions are correct in solving the producer/consumer problem, I wouldn't say that either are incorrect but the implementation which uses semaphores is more secure in providing synchronization as well as it puts a lock on any item being produced or consumed making it more reliable.

Comprehensibility:

While both implementations are simple, my implementation using semaphores is more simplistic than the one using threads. The semaphores implementation is easier to understand and has a very simple structure with producer and consumer classes as well as a distinct queue which has get() and put() functions. It is much easier to follow and is less complex. The amount of lines of code almost doesn't vary at all with the implementation using semaphores having 84 lines as opposed to the one using threads having 93. That difference really does not play a part in making one of the implementations easier to understand. In conclusion thought, the semaphore implementation is in my opinion a cleaner and more easy to follow implementation of the producer consumer problem.

Performance:

Based on the performance results the Semaphore implementation performed much better, it put and retrieved 59 threads as opposed to only 29 that the thread implementation achieved. The semaphore implementation also used a lot less memory at around 9000 kbytes rather than the thread implementation did at almost 17000 kbytes. Overall I would say that the semaphore implementation of the Producer consumer problem is more superior to that of the thread implementation of the same problem.

Dining Philosophers

Overview/ relevance:

The dining philosopher problem is a more challenging problem to understand and solve and consists of 5 philosophers sitting at a table, thinking, when they are done thinking and are ready to eat each philosopher must pick up both chopsticks on either side of him in order to start eating. The challenge is to avoid deadlock and also to not let any of the philosophers starve waiting to pick up their chopstick. Deadlock can easily occur in this situation if everyone picks up one chopstick and will indefinitely wait to pick up the second one since everyone is holding one. Even if there is a time constraint on the resources (chopsticks) there is still a possibility of livelock which can occur if every philosopher picks up the resource at the same time and also puts it down at the same time, only to repeat the same cycle indefinitely. This problem deals with a variety of common concurrency issues such as deadlock, starvation and livelock.

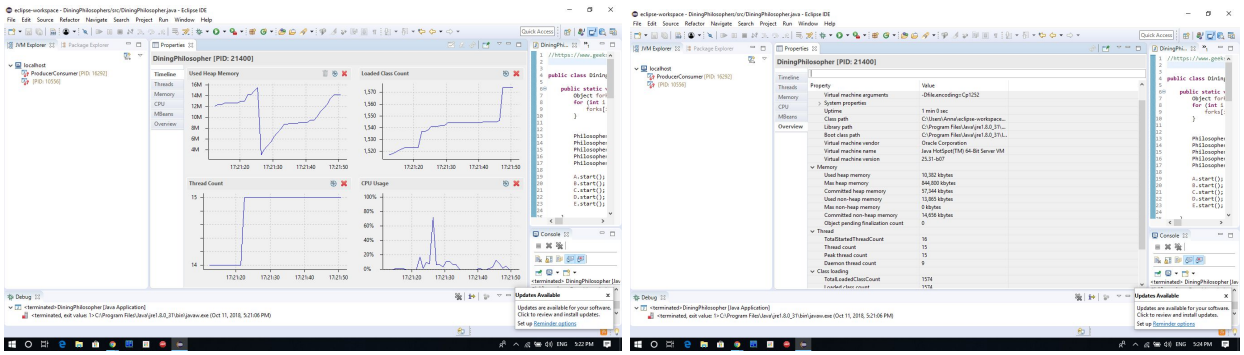
Links to my code: java , c

Links to references used:

https://www.geeksforgeeks.org/operating-system-dining-philosopher-problem-using-semaphores/#include<pthread.h>

https://github.com/topics/dining-philosophers-problem

Results:



Analysis:

Correctness:

I implemented my solutions using c and java, both solutions are fairly simple and appear to not encounter any deadlocks, livelocks or starvations. In order to test the correctness I went through the code and looked a the logic behind it and compared it to other implementations of the same problem. The java implementation also has random sleep times while he is eating. Both the java and c implementations also synchronize their fork pickups which makes sure they pick up the left fork and then the right one. In both cases the java and the c implementations are correct and appear to have no deadlocks or starvation.

Comprehensibility:

While its harder to judge the comprehensibility of both of the implementations because they are in different languages, to me it's clear that the java implementation is easier to follow. In both the c and java implementations the code is relatively easy to understand and the logic behind the code also makes sense. The number of code lines of both implementations is also very similar with the c implementation having about 30 more lines but a lot of them are whitespace so if that was to be removed it would be fairly similar. In conclusion both of the implementations were easy to understand and had easy logic behind them.

Performance:

Both of the implementations did pretty well in the performance spectrum. The java implementation had used 10,382 kbytes of heap memory and had a total thread count of 16 when ran for 1 minute straight. The C program was a bit harder for me to gather data from but it gave fairly the same results when ran. Both implementations run indefinitely with the same sleep times, and the java program yielded a more quick response time. Overall both programs are similar and neither had any performance issues to do with deadlock or starvation.

Dining Savages

Overview/ relevance:

The dining savages problem looks at the savages taking a serving out of a common resource, the pot. While the savage is looking into the pot he must have exclusive access to it so other savages cannot also take from the pot before he gets a chance to do so. The implementations i used are both written in java but one using a semaphore and another using a monitor. The dining savage problem

Links to my code: https://github.com/AnnaOlk/DiningSavages/tree/master

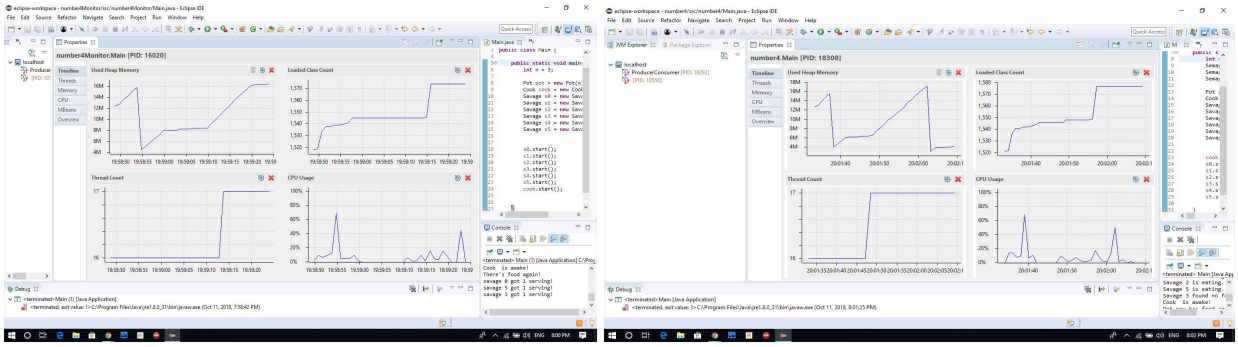
Links to references:

http://www.briantheprogrammer.com/blog/the-dining-savages-problem-in-o2/

https://github.com/natashadecoste/ConcurrencyStudies/blob/master/DiningSavages.java

Results: Both ran for exactly 1 minute

Monitor: Semaphore:



Monitor: 15,061 kbytes / Totoal Thread count: 18 / peak thread count: 17 / loaded class count: 1575

Semaphore: 14,827 kbytes / Total thread count: 18 / peak thread count 17 / loaded class count: 1577

Analysis:

Correctness:

Both implementations don't have any deadlocks or starvation as far as I could tell while debugging them. The main call functions are exactly the same for both implementations and they use the same amount of threads in each. I looked at different implementations for both the semaphore and monitor implementations and most of the logic behind every implementation is similar. In each implementation the pot gets a lock on it so that there is only one savage at a time checking to see if there is food in the pot and getting the food out for himself. Both implementations give the same results and are similar in their correctness.

Comprehensibility:

The monitor implementation is a bit simpler in terms of how easy it is to understand and in the number of lines of code used. Both the semaphore and the monitor implementations have the same number of files, one for the Main, the savage, the pot and the Cook. The monitor implementation locks the pot so only one savage at a time can be looking inside of it at a time while the semaphore implementation has the savage locked the resource. Both implementations are fairly easy to understand and are very similar in the way they are constructed. For me the monitor class is a little bit easier to understand because the pot is being locked so only one savage can look into it at a time which seems like an easier solution.

Performance:

Based on the performance both the monitor implementation as well as the semaphore implementation are almost identical. The graphs both show almost the same results for both of the implementations. Both of the implementations also had the exact same total as well as peak thread counts and just in general had very similar results. The monitor implementation had a very slightly higher kbytes total but that amount really does not make a difference in favor of the semaphore implementation. Overall both of the programs had very similar results so its hard to conclude just from that which one is superior to the other.

Barbershop Problem

Overview/ relevance:

The barbershop problem involves a barber who is waiting for customers to come into his shop, he only has one chair available so he can only tend to one customer at a time, the others are waiting in the waiting room to be helped next, there is a specified amount of chairs available in the waiting room the customers can occupy. When there are no customers the barber sleeps until a customer comes along. And if a customer walks in and that there are no chairs available in the waiting room he leaves. The problem deals with important concurrency issues of synchronization and inter-process communication.

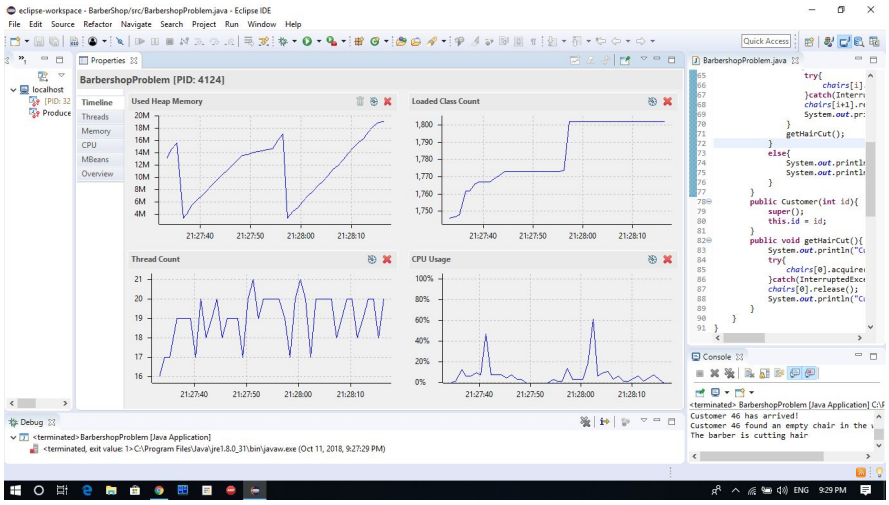
Links to my code: java , c

Links to references used:

https://github.com/kruegerstephen/BarbershopProblem

https://en.wikipedia.org/wiki/Sleeping_barber_problem

Results:



Analysis:

Correctness:

I used java to implement my first solution and C to implement the second one. Both of them have similar logic behind them and both also use semaphores. While both solutions are correct the java implementation follows more standard and easy to follow logic. The number of chairs also is the same between implementations with both of them having 3 chairs available in the waiting room.

Comprehensibility:

Both of the solutions were pretty easy to understand but the java implementation has a better layout and was easier to understand the logic behind it. The java implementation also follows logic used in the little book of semaphores more than the logic used in the c implementation. Both solutions were similar though in the sense that they used semaphores but the java implementation was overall cleaner. Personally i believe comprehensibility is subjective, and i find the java solution to be easier to understand because I am much more comfortable with using java as opposed to c. The amount of lines does not differ much between both implementations with the java implementation having 90 and the c implementation having 136 lines of code.

Performance:

The java and c implementations also differ in terms of how long they execute. The java implementation had an indefinite amount of customers always walking in while the c implementation has a limited amount of 24 customers. The java implementation was more successful in performance havin used 31,754 kbytes of heap memory, having a total of 152 threads and a peak thread count of 22. It had a better overall performance because of its indefinite amount of customers and its ability to turn them all away if no seats were available.