Alex Fattouche
V00828296

Assignment 1: Classical Synchronization Problems

Introduction

Before diving into the details of each individual problem, this introduction gives some information and specifics that all problems have in common.

Repository link: https://github.com/Fattouche/Classical_Concurrency
Specs of System:

- OS: Windows 10, 64 bit.
- Processor: Intel core I7 CPU @ 2.2 GHZ
- Ram: 8 GB

Languages and versions:

- Java: 1.8
- Go: 1.11

In order to measure performance, a profiler was used with each respective language:

- Java: Hprof - Java profiling tool for cpu and heap analysis [1].
- Go: Pprof/Graphviz - Pprof is a profiler native to golang and graphviz is graph visualization tool that integrates with pprof [2].

In order to ensure most accurate results for comparison between implementations in java and golang, the same number of sleeps and I/O operations were used in both programs, in the same locations. The comparisons made between Golang and Java for each problem were broken up into three criteria; correctness, comprehensibility and performance. Correctness will be measured by verifying the expected output of ten test executions and a line by line code walkthrough. The idea is that several test results should produce similar output and none of them should cause unexpected behavior. Comprehensibility will be measured by evaluating the lines of code to function ratio and evaluating the idiomacy of each respective solution. Lastly, performance will be measured by cpu/thread information from pprof and hprof for go and java respectively as well as general execution time.

✔️ = Best
🚧 = Medium
🔺 = Worst

** All java programs with the exception of the thread safe cache were implemented using the solutions presented in the little book of semaphores, each program makes direct reference to the book [3].*

Problem 1: Readers/Writers

Introduction

Readers/Writers is a classic concurrency problem that continues to cause issues today. This problem is so relevant to modern code that languages like Golang and Java have created built in data structures to handle this problem. Golang's sync library provides the RWMutex structure which has operations such as readLock(), readUnlock(), Lock() and Unlock() for handling the readers/writers problem [4]. Java uses the *ReentrantReadWriteLock* class to provide exposure to functions like readLock() and writeLock(), both of which return locks which can then be locked/unlocked as shown myRWLock.readlock().lock(). An example of where this problem arises today is in microservice architecture where several services need to read from one source, but need to have the latest up to date data.

Results

Both solutions were tested using 10 readers and 5 writers, in both situations no priority was given to the writer.

Go (source)

The go implementation of readers/writers was completed through the use of the built in RWMutex provided in golang. This allows the reader to call RWLock() and the writer to call Lock() which automatically synchronizes to this problem. Due to the lack of priority given to either readers or writers, the reader will generally have priority because of the nature of the read/write lock. The output of a test run is pasted below:

Read: 0
Read: 0
Write changed from 0 to 1
Read: 1
Read: 1
Read: 1
Read: 1
Read: 1
Read: 1
Read: 1
Read: 1
Write changed from 1 to 2
Write changed from 2 to 3
Write changed from 3 to 4
Write changed from 4 to 5

It is clear that the readers have priority because they can overlap lock holds to prevent any situation where the writer might grab the lock. Out of 10 test runs, the average time taken to

finish execution was roughly 730 ms. Figure 1 below is sample output produced from pprof combined with graphviz.

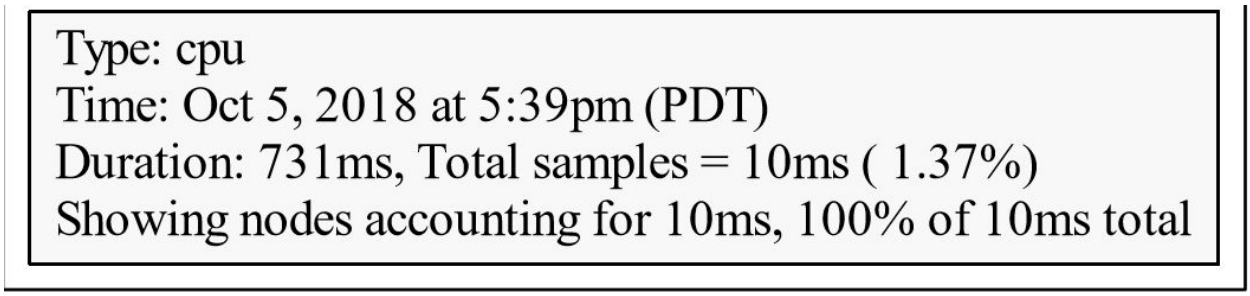

Figure 1: Pprof output for readers/writers

This figure shows an execution time of 731 ms and displays that the summation of samples gathered during the execution accounts for 10ms or 1.37% of the total runtime.

Java **(source)**
The java version of readers/writers was implemented using only semaphores as described in the little book of semaphores [3]. The general idea is similar to that of the lightswitch implementation in that several readers can enter a room at the same time (room signifies critical section) and the last reader must turn off the light in the room(light signifies semaphore) to notify the writer that they can enter. The text below is an output of a sample run:

Write changed from 0 to 1
Read: 1
Read: 1
Read: 1
Read: 1
Read: 1
Read: 1
Read: 1
Read: 1
Read: 1
Read: 1
Write changed from 1 to 2
Write changed from 2 to 3
Write changed from 3 to 4
Write changed from 4 to 5

Similar to the golang implementation, no priority was placed on either readers or writers which ultimately leads to an unavoidable reader priority. Hprof was used to analyze the cpu usage, the text below is the cpu usage breakdown:

rank self accum count trace method
1 14.29% 14.29% 1 300090 java.lang.Thread.currentThread
2 14.29% 28.57% 1 300067 java.lang.Thread.start0
3 14.29% 42.86% 1 300064 java.io.FileInputStream.open0
4 14.29% 57.14% 1 300049 sun.net.www.protocol.file.Handler.createFileURLConnection
5 14.29% 71.43% 1 300089 java.io.FileOutputStream.writeBytes
6 14.29% 85.71% 1 300040 java.io.FileInputStream.open0
7 14.29% 100.00% 1 300026 java.io.FileOutputStream.close0

We can see that the cpu breakdown is evenly 14.29% for all operations which is likely due to the fact that the number of readers/writers used was small. Due to this result, future problem solutions have had their test numbers increased. The average time for execution was 1.10 seconds over ten test results.

Comparison

As previously described, the solutions have been compared using the criteria of correctness, comprehensibility and performance.

Correctness

Golang

The solution to readers/writers implemented in golang is fairly straightforward which makes the evaluation for correctness reasonable. In order to ensure correctness of the code, idiomatic go code was written to utilize the built in RWMutex which is part of the sync package. The is mutex is almost an exact mold for this problem and allows the programmer to lock and unlock the required lock when needed. The code snippet below of the read() function is taken directly from the solution:

```
func read() {
    sleepThread()
    mutex.RLock()
    fmt.Println("Read: ", myGlobal)
    sleepThread()
    mutex.RUnlock()
    wg.Done()
}
```

The code for the read() function shows the use of mutex which of type RWMutex. Since this is the read function, it means that only the RLock() function needs to be called on the mutex variable. This style of code and the 10 successful test runs has determined this to be labeled as correct.

Java

The java code is significantly more complex than the golang code and is therefore harder to evaluate for correctness. This solution was implemented using semaphores in such a way that a mutex (semaphore of value 1) and a lightswitch semaphore were used to synchronize critical data as well as number of concurrent readers. The code snippet below shows the read functionality for java using semaphores.

```
mutex.acquire();
numReaders++;
if (numReaders == 1) {
  roomEmpty.acquire();
}
mutex.release();
System.out.println("Read: " + myGlobal);
mutex.acquire();
numReaders--;
if (numReaders == 0) {
  roomEmpty.release();
}
mutex.release();
```

The idea is that the mutex must be acquired before reading the numReaders variable and the roomEmpty semaphore must be acquired before any readers can begin to read(enter the room). This solution prevents race conditions by ensuring that only the authorized threads can enter their critical section at a time and that the decision to enter their critical section is guarded by a general data mutex. After 10 test runs, this solution has been labeled as correct.

Comprehensibility

Golang

As described in the previous section, this idiomatic code follows all golang practices. The code snippet below shows that for every new goroutine, we are adding one of those readers to the waitgroup which means our main program will block until they have finished working.

```
wg.Add(totalReaders)
for i := 0; i < totalReaders; i++ {
    go read()
}
```

This is the common style in go to ensure that all threads of execution have been properly joined before finishing. In addition to best practices in golang, this solution only uses 50 lines of code, evenly spread across functions, with no single function having more than 10 lines. These results have led this solution to be labeled as comprehensible.

Java

This java code has been structured to implement all best practices used in java development. The general solution is relatively straightforward once the idea of entering and exiting a room is understood properly. The area where the java code falls short in terms of comprehensibility is when attempting to understand the notation from the point of view of a non java developer. Java chooses to replace words like Signal() and wait() with function calls like acquire() and release() for semaphores which can be generally confusing. In addition, the programmer is required to handle exceptions that might arise from calling the semaphore functions. The code snippet below shows this:

```
try {
//my code
} catch (InterruptedException e) {
        System.out.println(e.getMessage());
}
```

Although the overall solution is comprensible, the idiomatic java code requires that the developer understand details and potential problems that might arise from multithreaded programming which is why this has been labeled as medium comprehensible.

Performance

Golang

Golang is well known for having blazing fast speeds and the readers/writers problem is no exception to this. Written on top of C, this solution uses user level concurrency known as goroutines to give the illusion of multithreaded programming. Clocking in at an average of 730ms, this program is roughly 300ms faster when written in golang than java. As the number of readers/writers increased, this number would grow larger. In addition, although the I/O operations in this program account for some of the latency, figure 2 shows that only 10m/s or roughly 1.37% of the time was spent doing I/O operations.

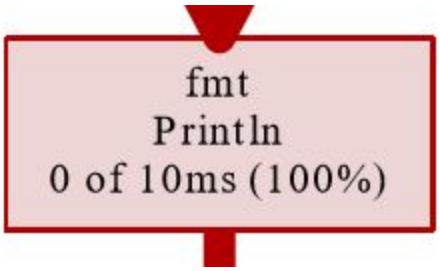

Figure 2: I/O operation usage in golang for readers/writers.

The fast speeds are most likely due to the fact that the RWMutex is specifically optimized for this problem, making it faster than an alternative solution using semaphores. This leads to the conclusion that this readers/writers solution in golang should be labeled with best performance.

Java

Unlike the golang solution, the java solution makes use of two different semaphores to ensure correctness. This sacrifice in correctness leads to the loss of performance because it introduces another bottleneck in the code that requires the programmer to check variables like *numReaders* before reading which are protected by a mutex semaphore. Clocking in at an average of 1.10 seconds over ten test results, this solution was roughly 100% slower than the golang solution. Therefore this has been labeled with worst performance.

Table 1 shows the final results of each category.

Language	Correctness	Comprehensibility	Performance	Overall
Golang	✔	✔	✔	✔
Java	✔	🚧	🔺	🚧

Table 1: Readers/Writers Overall Criteria

Problem 2: Barbershop

Introduction

The barbershop problem is based off the concept that several processes need the services that a single process provides, and cannot continue until the service has been received from that process. Although the name itself is a telling indication of how this problem might arise in everyday (non software related) life, it also appears in software more than one might think. An example of the barbershop problem could be seen with operating system scheduling where several processes need system level interrupts in order to proceed, but must wait on a queue before the kernel can assist them.

Results

Both solutions were tested using 100 customers, 5 chairs and 1 barber.

Go (source)

The golang implementation of barbershop was implemented using one buffered channel of length numChairs that simulates people waiting on chairs and one non buffered channel that simulates the availability of the barber. The snippet below is some sample output where each customer is its own goroutine.

Customer 35: Waiting in chair
Customer 30: Getting haircut
Customer 36: Arriving to barbershop
Customer 36: Waiting in chair
Customer 37: Arriving to barbershop
Customer 37: All chairs taken, leaving
Customer 31: Getting haircut

The average run time for this solution was 12.25 seconds over 10 runs. Figure 3 shows the output from graphviz and pprof.

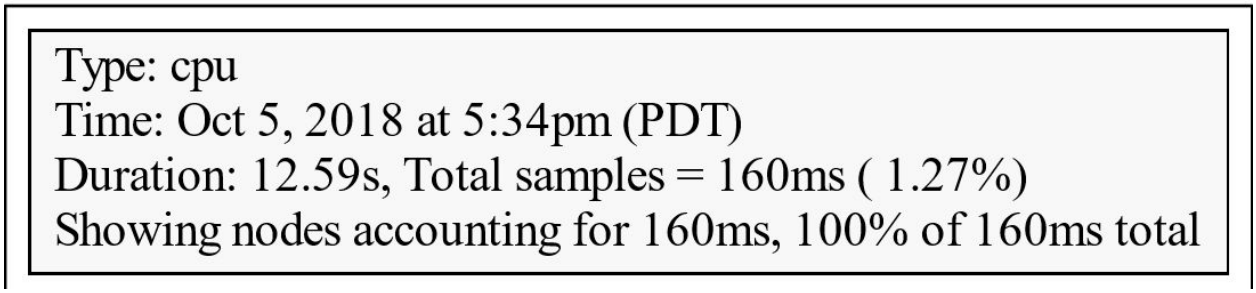

Figure 3: Pprof output for barbershop

Java (source)

Once again, this java solution to barbershop was using only semaphores and is based entirely off the solution given in the little book of semaphores [3]. The solution utilizes 5 different semaphores to achieve the goal of ensuring that the barber only serves one customer at a time while at most 5 customers can wait for the barber to be available. The snippet below shows the output from a sample test run.

Customer 44: Getting hair cut
Customer 49: Arriving to barbershop
Customer 49: Waiting in chair
Customer 50: Arriving to barbershop
Customer 50: All chairs taken, leaving
Customer 45: Getting hair cut
Customer 46: Getting hair cut

The average time for the barbershop problem with the java solution is 13.10 seconds. A snippet of the cpu usage breakdown is shown below:

2.70% java.util.concurrent.locks.LockSupport.unpark
2.70% java.io.FileOutputStream.writeBytes
2.70% java.lang.Thread.setPriority0
2.70% java.io.FileOutputStream.writeBytes
2.70% java.lang.Object.getClass
2.70% java.lang.System.arraycopy

2.70% java.io.FileOutputStream.close0
2.70% java.lang.Integer.stringSize
2.70% java.io.FileOutputStream.writeBytes
2.70% sun.misc.Unsafe.compareAndSwapObject

Some important things to notice is the cpu usage of *compareAndSwap* as well as *LockSupport.unpark* both of which are directly related to the semaphores.

Comparison

As previously described, the solutions have been compared using the criteria of correctness, comprehensibility and performance.

Correctness

Golang

Unlike the readers/writers solution, the barbershop problem is more complex and is implemented using the idiomatic style of channels. The solution simplifies the barbers code for cutting hair down to two lines, making it very clear what the goal is and how it is being achieved.

```
func cutHair(chairs, barberAvailable chan int) {
    for {
        //Wait till someone enters
        barberAvailable <- 1

        //Free up a chair
        <-chairs

        //cut hair
    }
}
```

The code for the customer getting a haircut is very similar in terms of correctness and the 10 successful test runs verify that the solution is indeed correct.

Java

Once again, the java code for the barbershop problem is significantly more complex than the golang implementation which is a result of using semaphores to implement the functionality. The correctness is especially difficult to determine when looking at the customer function because several semaphores are being handled at the same time which could lead to possible deadlock. The code snippet below is an example of this:

```
mutex.acquire();
if (currentCustomers == chairs) {
```

```
            mutex.release();
            return;
        }
        currentCustomers++;
        mutex.release();
        customerMutex.release();
        barberMutex.acquire();
```

In these 8 lines of code, 3 different semaphores and 2 variables are being used. Fortunately, the ten test runs show that this solution indeed works and deadlock or data races were never observed. Therefore this solution has been labeled as correct.

Comprehensibility

Golang

The golang implementation does a fantastic job at minimizing variables and using descriptive variable names as shown below.

```
chairs := make(chan int, numChairs)
barberAvailable := make(chan int)
```

Here it is clear to the developer that the chairs channel is buffered to have a size equal to the number of chairs. In addition, the barberAvailable channel is unbuffered and tells the programmer it signifies the barbers availability which is clearly a binary value. The combination of these two channels leads to the simple solution that when a customer arrives they check to see if the barber is available, if it is not available, they will attempt to wait in one of the chairs. Since these channels can potentially block, the select statement shown below is used to avoid blocking in wrong locations.

```
//Check if barber is available
    select {
    case <-barberAvailable:
        fmt.Printf("Customer %d: Getting haircut\n", id)
        return
```

Here the customer will attempt to write to the barberAvailable channel, but will not block if the barber is unavailable. The customer will then attempt to do the same thing with the chairs channel and ultimately return if all chairs are taken. As a result of the code obeying idiomatic go guidelines and the longest function being 15 lines long, this implementation has been labeled as comprehensible.

Java

The java implementation of the barbershop problem utilizes 5 semaphores as mentioned in the correctness comparison. Not only does this large number of semaphores impact the

programmers ability to determine the correctness of the solution, it also reduces the comprehensibility of the code.

```
static Semaphore mutex = new Semaphore(1);
static Semaphore customerMutex = new Semaphore(0);
static Semaphore barberMutex = new Semaphore(0);
static Semaphore customerDoneMutex = new Semaphore(0);
static Semaphore barberDoneMutex = new Semaphore(0);
```

Each of these semaphores is responsible for controlling some sort of critical section and it is not entirely clear what each semaphore is responsible for. In addition to the overkill on semaphore usage, the idioms used by java to handle errors and initialize new threads give this a comprehensibility score of bad.

Performance

Golang

The golang implementation of the barbershop problem reduces the opportunity for multiple locks/unlocks or waits/signals by reducing the number of synchronization structures to two. The chairs channel specifically does an excellent job of tackling this problem by providing functionality for both synchronization but also tracking the remaining available chairs in one native data structure. The average test run lasted 12.25 seconds and figure 4 shows the acquire call, responsible for channel manipulation lasting 10ms.

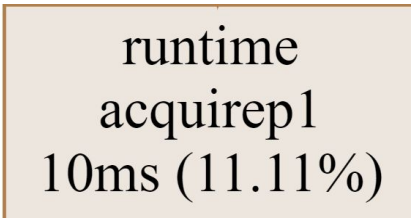

Figure 4: Golang breakdown using graphviz

Due to the minimal blocking done in this solution and the 12.25 average run time, this has been labeled as performant.

Java

The increase in locking mechanisms used throughout this implementation in comparison to the golang solution, leads to slower performance. For example, in order for the barber to successfully check whether someone needs their hair cut, the barber thread needs to acquire 2 different semaphores. Additionally, once the barber is finished with the semaphores, they must be released. This falls short in comparison to golang because each time the barber performs this check, 4 operations must be done. Alternatively in golang, the act of interacting with a

channel incorporates locking and unlocking in one operation. In the end, the java solution is only 1 second slower on average, clocking in at 13.10 seconds, making its performance medium.

Table 2 shows the final results of each category.

Language	Correctness	Comprehensibility	Performance	Overall
Golang	✔	✔	✔	✔
Java	✔	🔺	🚧	🚧

Table 2: Barbershop Overall Criteria

Problem 3: Dining Savages

Introduction

The dining savages problem is similar to the Barbershop problem in that it can be applied to system level operating systems. The idea behind the barbershop problem is that several processes can consume resources until there is none left, which then requires the cook to step in and produce more. This can be mapped to the Linux out of memory(OOM) killer that is responsible for killing processes once the system has ran out of memory which is usually a result of several processes "eating" up the memory.

Results

Both solutions were tested using 100 savages and a serving size of 10.

Go (source)

The golang implementation of the dining savages problem utilizes 3 channels, all of which are buffered. The first channel *potEmpty* is used by the savages to notify the cook of when the pot is empty, it is buffered to a size of 1. The second channel, *potFull* is used by the cook to notify the savages of when the pot is full and ready to be eaten from again, it is also buffered to a size of 1. The last channel *pot* is used to simulate a pot with *n* servings which can be actively consumed and refilled by both the cook and the savages, it is buffered to fit the size of servings which is 10. The snippet below is an example of the golang output from this problem.

eating
eating
eating
eating
eating
eating
eating

eating
Replenishing pot

The snippet shows that after the 10th savage has finished eating, the cook is notified to replenish the pot. The average run time for this solution was 205ms over 10 runs. Figure 4 shows the output from graphviz and pprof.

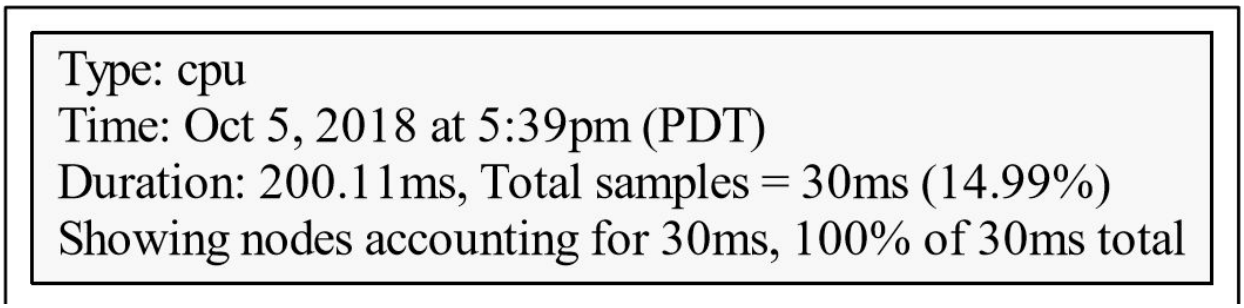

Figure 4: Pprof output for dining savages

Java **(source)**

Similar to the golang implementation, the java implementation to dining savages uses three semaphores to handle synchronization. The solution uses the general mutex style semaphore and two semaphores to handle when the pot is full and when the pot is empty. The snippet below is the example output in java.

eating
eating
eating
eating
eating
eating
eating
eating
Replenishing pot

The average time for the barbershop problem with the java solution is 400ms, a breakdown of the cpu usage is provided below

rank	*self*	*accum*	*count*	*trace*	*method*
1	*11.11%*	*11.11%*	*1*	*300176*	*java.lang.System.arraycopy*
2	*11.11%*	*22.22%*	*1*	*300114*	*java.io.FileOutputStream.writeBytes*
3	*11.11%*	*33.33%*	*1*	*300113*	*java.lang.Object.<init>*
4	*11.11%*	*44.44%*	*1*	*300175*	*java.io.FileOutputStream.writeBytes*
5	*11.11%*	*55.56%*	*1*	*300066*	*java.io.FileInputStream.open0*

Comparison

As previously described, the solutions have been compared using the criteria of correctness, comprehensibility and performance.

Correctness

Golang

The golang implementation for dining savages is very similar to the barbershop problem in the fact that they both use a buffered channel to handle both synchronization and information tracking. Instead of using chairs however, the *pot* channel is used to keep track of how many servings remain within the pot. The code for the chef is quite self explanatory and relies on the fact that the *potEmpty* channel will not be written to unless the savages are in need of more food.

```
<-potEmpty
//Fill pot
fmt.Println("Replenishing pot")
for i := 0; i < servingSize; i++ {
    pot <- 1
}
//Notify pot is full
potFull <- 1
```

Likewise, the code for the savages is almost a direct opposite of the chef in the fact that they wait until the pot is full, and eat from the pot when they are able to. This idea of how the channels play into the solution leave little room for error. In addition, the 10 successful test runs label this as correct.

Java

The java solution is actually very similar to the golang solution for the dining savages problem. The accesses to *potEmpty* and *potFull* channels found in the chef code are replaced with *emptyPot.aquire()* and *fullPot.release()* respectively as shown in the code snippet below.

```
// wait for pot to be empty
emptyPot.acquire();

// Refill pot
System.out.println("Replenishing pot");

// Notify pot is full
fullPot.release();
```

Additionally the savage code is very similar to the golang code except for the fact that an additional *servings* variable needs to be kept track of in java. We can then say that based off the logic defined in the code and the 10 successful test runs that this solution is correct.

Comprehensibility

Golang
In terms of comprehensibility, this golang solution appears to be slightly more complex than the previous two problems because it introduces an addition channel. After viewing the code however, it appears to make more sense as to why three channels were needed instead of 2. Each of the *potFull* and *potEmpty* channels describe the communication between chef and savage and vice versa, making the code more readable while sacrificing overall lines of code. One area where the code is not as straightforward is when the savage needs to eat as shown in the code snippet below.

```
<-potFull
select {
case <-pot:
    //Eat from pot
    fmt.Printf("eating\n")
    potFull <- 1
    return
default:
    //Pot is empty, need to notify cook
    potEmpty <- 1
}
```

Since channels in go cannot be simply peeked like one might do with a stack or queue, the savage is forced to write back into *potFull* if the pot ended up not being full. This prevents the next savage from being blocked on *<-potFull* which would cause a deadlock because the chef is not in the process of filling the pot. This slight nuisance in the code justifies that it be labeled as medium in comprehensibility.

Java
In the java implementation, there is no catch or specific problems that when solved, may make the code confusing. Apart from the requirements of java that make the software heavy and nearly 15 lines longer than the golang implementation, the code is quite self explanatory and does a good job of relating the problem to a real world situation giving it an overall medium in comprehensibility.

Performance

Golang
The golang performance was very clear during this problem because of the efficient use of channels. Averaging 205 ms per 100 savages, the code for golang was almost twice as fast as the code written in java. Unlike the other problems where golang was faster, the reason for the speedup is not immediately clear for the dining savages problem. One potential reason could be that in golang, channels transfer the data one byte at a time which not only synchronizes but also shares memory. On the other hand, semaphores require an atomic operation to be completed that does not do any memory sharing. Each of these atomic operations may add a small amount of latency, however added up could be a significant amount. Due to the average speed of the program, this has been labeled as best performance.

Java
The java performance for dining savages is significantly slower than the golang implementation, clocking in at an average of 400ms over 10 test runs. The profile shows that the operation *sun.misc.Unsafe.compareAndSwapInt* is responsible for 11% of the cpu usage throughout the execution of the program. This in addition to the tracking of global variables causes a decrease in performance. One alternative to speed up the program would be to try and force the compiler to store variables in registers to speed up access times, however, Java does not provide this option. Therefore, this has been labeled as poor performance.

Table 3 shows the final results of each category.

Language	Correctness	Comprehensibility	Performance	Overall
Golang	✔	🚧	✔	✔
Java	✔	🚧	▲	🚧

Table 3: Dining Savages Overall Criteria

Problem 4: Santa Claus

Introduction
The santa claus problem attempts to handle several different concurrency situations in one problem. The problem is broken down such that santa is in charge of helping the elves until the time has come that all 9 reindeer return to him. The catch to the problem is that he can only help the elves if they come to him in groups of three. This is known as a grouping turnstile where one elf is blocked until all three elves have arrived. While this is occurring, santa must also be aware of how many reindeer have arrived. This problem could show up in any concurrent program that

requires all threads to be working on the same task in a way that they cannot move on until all threads have finished their pieces of the task. Similar to the readers/writers problem, Go and Java provide built in mechanisms such as waitGroups and join() function calls to handle this problem.

Results

Both solutions were tested using 100 elves, 9 reindeer and one santa.

Go (source)

The golang implementation of the santa claus problem once again makes use of the golang channels. Due to the complexity of the program, five channels were needed to create a correct solution. Two buffered channels *reindeerRemaining* and *elvesWaiting* keep track of how many reindeer still need to arrive and how many elves are waiting to get help from santa respectively. In addition, three other non buffered channels *reindeerFinished, santaAvailable* and *threeElves* are used to communicate between goroutines about information related to the state of each entity in the problem. The snippet below is output from a sample test run.

Helping three elves
Helping three elves
Helping three elves
New reindeer arrived
Helping three elves
New reindeer arrived
Last reindeer arrived, notifying santa
Reindeers finished, hitching sled!

The output is from the point of view of santa and shows the different options for reindeer arriving and helping elves. Once the last reindeer has arrived, no more elves should be helped. The average runtime across 10 runs was clocked at 2.30 seconds. Figure 4 shows the output from graphviz and pprof.

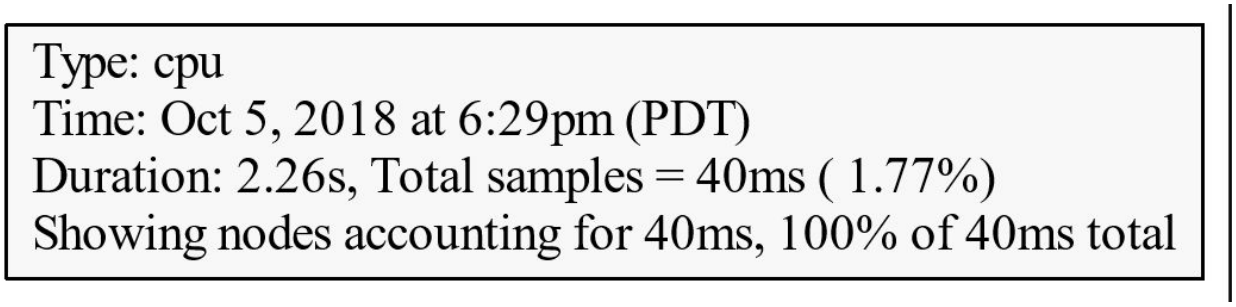

Figure 5: Pprof output for santaclaus

Java (source)

The java solution to santa claus once again utilizes semaphores as described in the little book of semaphores [3]. This solution is the only one of the six in which the synchronization structures that are needed for the java implementation is less than the golang implementation. This implementation uses four semaphores, three of which are responsible for the elves, reindeer and santa respectively and one that is used as a general mutex when altering the number of elves that are currently waiting. The snippet below is example output from the java solution.

New reindeer arrived
New reindeer arrived
Helping three elves
Helping three elves
New reindeer arrived
Helping three elves
New reindeer arrived
Last reindeer arrived, notifying santa
Reindeers finished, hitching sled!

The average time for the santa claus problem with the java solution is 2.55s, a breakdown of the cpu usage is provided below.

rank	*self*	*accum*	*count*	*trace*	*method*
1	*9.09%*	*9.09%*	*1*	*300188*	*sun.misc.Unsafe.compareAndSwapInt*
2	*9.09%*	*18.18%*	*1*	*300186*	*java.nio.Buffer.position*
3	*9.09%*	*27.27%*	*1*	*300185*	*sun.misc.Unsafe.compareAndSwapInt*
4	*9.09%*	*36.36%*	*1*	*300184*	*java.io.FileOutputStream.writeBytes*
5	*9.09%*	*45.45%*	*1*	*300183*	*sun.misc.Unsafe.putObject*
6	*9.09%*	*54.55%*	*1*	*300182*	*sun.misc.Unsafe.compareAndSwapInt*
7	*9.09%*	*63.64%*	*1*	*300123*	*java.lang.Thread.start0*
8	*9.09%*	*72.73%*	*1*	*300187*	*sun.misc.Unsafe.unpark*
9	*9.09%*	*81.82%*	*1*	*300060*	*java.util.concurrent.Semaphore.<init>*

Comparison

As previously described, the solutions have been compared using the criteria of correctness, comprehensibility and performance.

Correctness

Golang

The golang implementation of the santa claus problem is quite complex due to the number of channels that are being used at the same time. It is not immediately clear based off just logical analysis whether this solution will work in every case. The code shown below could be a point of error at some point if really stress tested, however no bugs have appeared yet.

```
func waitForReindeer(reindeersFinished chan int) {
    //Unbufferred channel, blocks till reindeers finished
    <-reindeersFinished
    fmt.Println("Reindeers finished, hitching sled!")
    wg.Done()
    return
}
```

This could potentially be a problem because the program expects that once the last reindeer has arrived, no other elves should be helped, however if this function calls wg.Done() while at the same time an elf is attempting to get help, santa may still help. This is because santa was split up into two separate goroutines, one that watches for reindeer arriving and one that watches for elves that need help. Although this could be a problem, out of the 10 runs, none of them behaved abnormally. Therefore this is given a medium correctness.

Java

Although the java solution is quite complex to understand, once each line has been walked through and understood, the implementation removes any doubt about its correctness. The code shown below does a good job of showing how every critical section has been carefully thought out, ensuring proper locks are acquired and released appropriately.

```
elfTex.acquire();
mutex.acquire();
elves++;
// If max elves, notify santa otherwise move on
if (elves == 3) {
    santaSem.release();
} else {
    elfTex.release();
}
mutex.release();
```

In addition to the well examined code, which includes edge case coverage, the test run showed 10 successful results, giving this the best correctness score.

Comprehensibility

Golang

The golang implementation of santa claus is not entirely clear cut in comparison to the java solution. Unlike in the java solution, the core functionality of the santa is split up across several functions and threads. This attempt to break down specific tasks into unique functions makes it difficult to understand what is happening at what times in the code. For example, the santa figure in this problem is required to not only help elves but also monitor the reindeer count so that christmas can begin. This is split into two separate goroutines that when the problem is really understood, makes sense why, however, at first glance the code below might be confusing.

```
func waitForReindeer(reindeersFinished chan int) {
    //Unbufferred channel, blocks till reindeers finished
    <-reindeersFinished
    wg.Done()
    return
}

func helpElves(santaAvailable, threeElves chan int) {
    for {
        <-threeElves
        fmt.Println("Helping three elves")
        santaAvailable <- 1

    }
}
```

The alternative for this problem could be to separate each of the different figures into their own files (santa.go, elf.go, reindeer.go) so that the code could be more accurately modularized. Apart from this confusing code structure, the LOC/function ratio is quite low and each function can be clearly understood on its own. Therefore this is given a medium comprehensibility.

Java

The java implementation of santa claus is quite well structured. Each figure in the problem is given its own class and the functionality is well contained within a single function. Apart from the need for keeping track of the four semaphores shown below, this code is quite comprehensible.

```
static Semaphore mutex = new Semaphore(1);
static Semaphore elfTex = new Semaphore(1);
static Semaphore santaSem = new Semaphore(0);
static Semaphore reindeerSem = new Semaphore(0);
```

This will be given the best rating in comprehensibility.

Performance

Golang
The golang performance for this problem was underwhelming when compared to other problems. Although still faster than the java solution, it failed to maintain the blazing fast speeds that previous problems have shown. Clocking in at an average time of 2.30 seconds, this was only 0.20 seconds faster than the corresponding java solution. This is most likely a result of the several branching factors within the program as shown in figure 6.

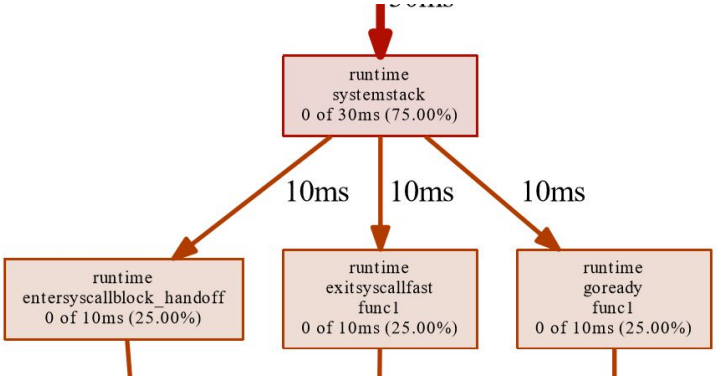

Figure 6: Santa claus profile output

This is most likely a result of several goroutines executing at the same time all needing access to the same shared memory. This will be given a performance rating of medium.

Java
The java performance for santa claus exceeded expectations given its history with the problems previously described. Having said that, this speed is still slower than the golang program, even when the golang program had been especially slow for normal. Clocking in at an average time of 2.5 seconds, the main delay in the code was probably a result of the *sun.misc.Unsafe.compareAndSwapInt* operation, which accounted for roughly 30% of the CPU usage. Overall this will be given a performance rating of medium.

Table 4 shows the final results of each category.

Language	Correctness	Comprehensibility	Performance	Overall
Golang	🚧	🚧	🚧	🚧
Java	✔️	✔️	🚧	✔️

Table 4: Santa Claus Overall Criteria

Problem 5: Search Insert Delete

Introduction

The search insert delete problem is similar to the readers/writers as described above because of the necessity to block concurrent updates but allow for concurrent reads. This problem arises in caching and queuing systems that require items to be constantly read/updated/consumed by several different processes. A common industry example of this might be the use of redis as a key/value store but also as a job queue.

Results

Both solutions were tested using 50 searchers, 20 inserters and 10 deleters.

Go (source)

The golang implementation for search, insert, delete is fairly straightforward and very similar to the readers writers problem. The idea with search, insert, delete is that any number of searchers and a single inserter can work concurrently, however a single deleter must block everyone else. Two RWMutexes were used two solve this problem. The first *mutexDelete* is readlocked by searchers and inserters and write locked by the deleter. This allows multiple concurrent inserters/searchers but only one deleter. Similarly, the *mutexInsert* must be locked by both inserters and deleters to ensure only one inserter can insert at a time. The snippet below is an example of the golang output.

Inserting: 74
Searched: 74
Searched: 74
Searched: 74
Searched: 74
Searched: 74
Inserting: 29
Searched: 29
Deleting: 3

The golang implementation has the readers reading from the front of the list, however the problem makes no mention of where the readers are supposed to be reading from. Figure 7 shows the output from the pprof and graphviz for this problem. The average runtime over 10 runs was 4.17 s.

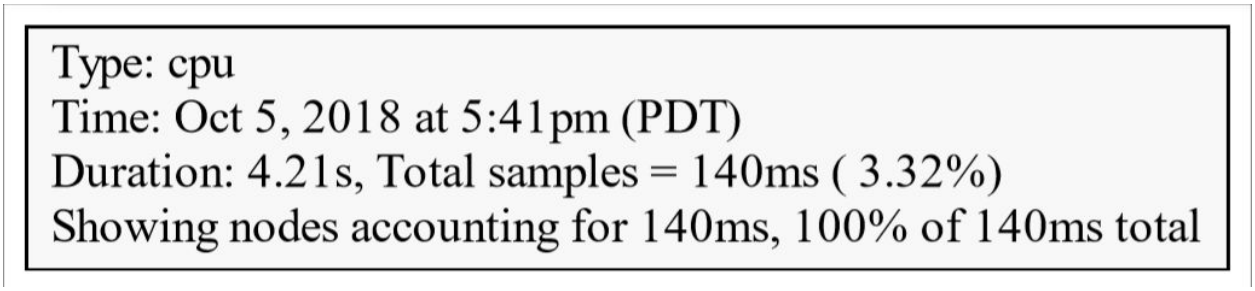

Figure 7: Pprof output for search insert delete

Java **(source)**

The java solution to santa claus once again utilizes semaphores as described in the little book of semaphores [3]. The solution as described in the book utilizes another custom class called *LightSwitch* that controls the number of threads attempting to access a shared variable. The *LightSwitch* class is also commonly used in the "lightswitch" solution to readers writers. Overall this solution uses three basic semaphores and 2 *LightSwitch* objects. The sample below is output from the java solution.

Searched: 1
Searched: 1
Searched: 1
Searched: 1
Inserting: 85
Inserting: 45
Inserting: 18

It is clear that unlike the golang solution, there is very strict priority in this implementation. First most of the search, then the inserters insert and lastly the deleters delete. The average runtime over 10 test runs was 4.5 seconds and the snippet below shows the most applicable cpu usages.

6.67% sun.misc.Unsafe.compareAndSwapInt
6.67% java.io.FileOutputStream.writeBytes
6.67% java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly
6.67% java.security.AccessController.getStackAccessControlContext
6.67% java.io.FileOutputStream.writeBytes

Comparison

As previously described, the solutions have been compared using the criteria of correctness, comprehensibility and performance.

Correctness

Golang

The golang implementation of search, insert, delete makes good use of the RWMutex structure provided by the golang sync package. The breakdown of mutexes into hierarchies related to their applicability makes the objective quite clear and unambigious. The code snippet below shows the search and insert functions and how they both utilize mutexes.

```
func search() {
    mutexDelete.RLock()
    fmt.Println("Searched: ", myList.Front().Value)
    mutexDelete.RUnlock()
}
func insert() {
    mutexInsert.Lock()
    mutexDelete.RLock()
    fmt.Printf("Inserting: %d\n", val)
    mutexInsert.Unlock()
    mutexDelete.RUnlock()
}
```

The code shows that both search and insert can be performed at the same time, however, once a deleter attempts to delete, the thread must wait for both the searcher and the inserter to finish. After 10 successful test runs, this is given the best correctness score.

Java

The use of the *LightSwitch* class in this solution allows for more precise control on the solution which removes any ambiguity about whether or not this solution is correct. The code snippet below shows that the searcher does not need to acquire the *noSearcher* sempaphore unless they are the first thread. This allows several searchers to continue concurrently, however, prevents any other thread from accessing the list.

```
static class Searcher implements Runnable {
       public void run() {
```

```
                searchSwitch.wait(noSearcher);
                System.out.println("Searched: " + myList.get(0));
                searchSwitch.signal(noSearcher);
            }
        }


    public void wait(Semaphore semaphore) {
              mutex.acquire();
              counter++;
              if (counter == 1) {
                  semaphore.acquire();
              }
              mutex.release();
    }
```

Although the output for the java solution is quite different than the golang solution in terms of priorities, out of 10 test runs, all outputs obeyed the problem definition. Therefore this will be labeled as fully correct.

Comprehensibility

Golang
The golang implementation of search,insert,delete does a fantastic job of breaking down specific pieces of code into well structured components. Each figure(searcher, inserter, deleter) in this problem is given their own function and has well defined requirements that must be met in order to perform their given task. For example, it is very clear that in order for the deleter to delete something, it must first acquire the *mutexInsert* lock to prevent any inserters from concurrent access and then acquire the *mutexDelete* lock to prevent readers/inserters. In addition, each function averages to be only 8 lines of code. This will be given a comprehensibility score of best.

Java
The java implementation of search,insert,delete using semaphores and *LightSwitch* is far too complex for what the problem describes. Not only must the programmer fully understand the basic problem, they also must understand the purpose of the *LightSwitch* class and why it is needed within the solution. An alternative and possibly easier to understand approach would be to use Java's built in *ReentrantReadWriteLock* so that the solution could be broken down into two read/write mutexes as shown in the golang solution. This will be given a comprehensibility score of poor.

Performance

Golang

The golang performance for this problem was once again underwhelming when compared to other problems. This is most likely a result of the excessive branching that occurs throughout the program and leads to unnecessary cpu usage as shown in figure 8.

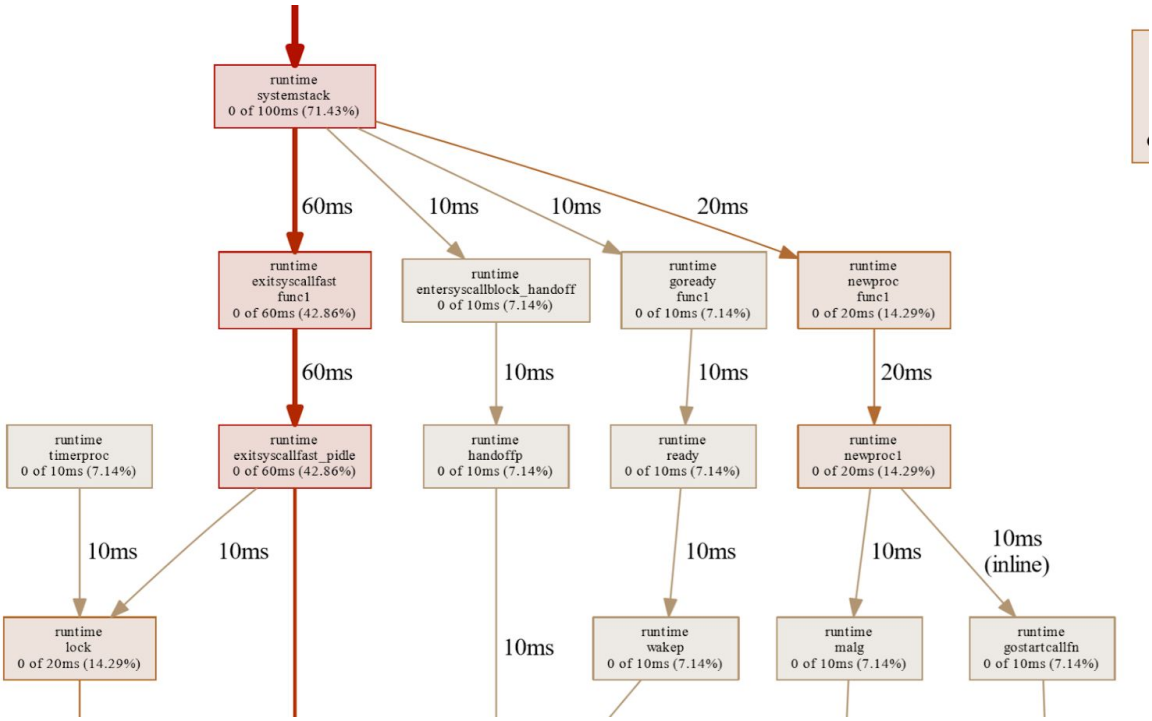

Figure 8: Search, Insert, Delete PProf output

The figure shows that several *runtime wakeup*'s are occurring throughout the execution of the program which means that contexts are constantly being switched. This differs from the java program where priority is clearly given to the searchers, which ultimately reduces the need for context switching. Clocking in at an average of 4.17 s over an average of 10 runs, this will be given a performance rating of medium.

Java

The java performance for search,insert,delete was comparable to the golang solution and this was to be expected given the priority given to each figure. This meant that all the searchers could execute at the very start, then the inserters, then the deleters. This could be compared to a situation where a deleter attempts to delete something every X seconds, forcing all other threads to wait until finished. Clocking in at an average time of 4.5 seconds, this will be given a performance of medium.

Table 5 shows the final results of each category.

Language	Correctness	Comprehensibility	Performance	Overall
Golang	✔	✔	🚧	✔
Java	✔	🔺	🚧	🚧

Table 5: Search, Insert, Delete Overall Criteria

Problem 6 (Custom): Thread Safe Expiring Cache

Introduction

The sixth and extra problem that was chosen is a thread safe cache that allows for expirations to be set on objects within the cache so they are automatically removed once expired. This is very applicable to things related to networking because it allows for very precise, up to date information to be stored and used on the fly. An example of a system that might use a thread safe expiring cache is a proxy server that acts as distributed denial of service(DDOS) protection by caching IP addresses that have recently been causing lots of traffic. The thread safe part of the cache allows for the server to be multithreaded so that any thread can have access to which IP's are the most impactful. The expiration part of the cache allows for only recent information to be used to block IP addresses which can prevent permanently banning IP address that might have been used from hacked devices.

Results

This solution was tested by inserting ten things into the cache while simultaneously attempting to search and delete from the cache. The print statements have been specifically shown in a way that shows how one could intend to perform an action, but the state of the cache could be changed by the time the action was execution.

Go (source)

The golang implementation for the thread safe cache allows for multiple concurrent gets, while only allowing a single insert or delete operation at a time. This functionality was introduced through the use of a single RWMutex that is read locked within the *get* and *exists* function and is write locked within the *insert* and *remove* functions. In addition to the basic thread safe cache functionality, the cache was implemented to allow for an expiration to be set on the items in the cache so that they are removed once they have expired. This expiration functionality appears to be some form of concurrency, however, it is actually simulated by checking if the item has expired before returning from the respective function. This "call time" check on the expiration reduces the overhead of having a long running goroutine that removes expired items from the cache. The snippet below is a sample run.

Inserting 1-10

Deleting 1
deleted 1 from cache
Searching 1
Failed to find 1
Searching 2
Found 2 with value 2
Deleting 2
deleted 2 from cache
Searching 5

The average runtime for the tests performed on the cache was 290 ms. Figure 8 shows the output from pprof and graphviz for the cache problem.

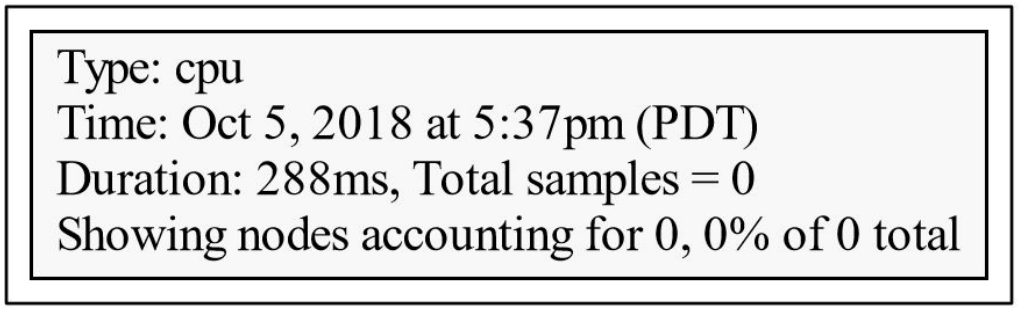

Figure 8: Pprof output for expiring cache.

Java (source)

Unlike all other java solutions described in this report, the java solution to the thread safe cache uses a similar solution to the golang solution described above through the use of *ReentrantReadWriteLock* class in Java. This allows for a more direct approach than the other solutions and does not use any ideas from the little book of semaphores. Below is sample output from the a test run.

Deleting 2
deleted 2 from cache
Deleting 7
deleted 7 from cache
Deleting 9
deleted 9 from cache
Searching 2
Failed to find 2
Searching 5

Due to the similarities in implementation details between the golang and java solutions, the output is very similar to the golang output. This java solution averaged 285 ms across ten test runs. The text below describes the cpu output given by Hprof.

```
 rank   self  accum   count trace method
1 12.50% 12.50%       1 300085 java.lang.System.arraycopy
2 12.50% 25.00%       1 300074 java.io.WinNTFileSystem.getLength
3 12.50% 37.50%       1 300071 java.io.FileInputStream.open0
4 12.50% 50.00%       1 300062 java.io.FileInputStream.open0
5 12.50% 62.50%       1 300083 java.io.FileInputStream.open0
6 12.50% 75.00%       1 300041 java.io.FileInputStream.open0
7 12.50% 87.50%       1 300039 java.net.URL.<init>
```

Note that the cpu usage does not give any indication of the use of compare and swap that was formerly very apparent in semaphore solutions.

Comparison

As previously described, the solutions have been compared using the criteria of correctness, comprehensibility and performance.

Correctness

Golang

The golang implementation of the threadsafe cache is broken down into two different pieces of functionality. The cache struct implements 4 functions; *get(), insert(), remove(), exists()* and wraps around the built in golang map of type *map[string]object*. Each cache has a single read write mutex that is used by the 4 functions previously described. The values that enter into the cache are structs called *objects* and each *object* type has an integer value and corresponding expiration time. The *object* struct implements two functions; one that returns the integer value and one that checks if the object is expired. This layout of object oriented code means that the synchronization within the 4 cache functions is straightforward and reasonable to prove is correct. In addition, from a statistics point of view this implementation was successful on all ten test runs, giving it the best score for correctness.

Java

As previously described in the results section, the java solution is an almost identical copy of the golang solution, however it uses the java class object system instead of golang structs. This is the only difference in implementation meaning that it is also given the best correctness score.

Comprehensibility

Golang
The argument for comprehensibility in this case is very similar to the argument for correctness because both of them are directly impacted by the structure of the code and how easy it is to understand. In terms of comprehensibility specifically, this code follows idiomatic go patterns such as the use of defer as shown in the code segment below.

```
func (c *cache) get(key string) (interface{}, bool) {
    c.mutex.RLock()
    defer c.mutex.RUnlock()
    obj, ok := c.objects[key]
    if isExpired(obj) {
        delete(c.objects, key)
        return nil, false
    }
    return obj, ok
}
```

The use of the *defer* keyword in the *get* function shown above notifies the programmer that the mutex will be read unlocked no matter what happens. In addition, the signature of the get function tells the programmer that the function is implemented on the type cache and returns two parameters, one of which is the value from the map of type *interface* and the other is a boolean signifying whether the item existed or not. In addition, each function is relatively short, averaging 10 lines of code per function. Due to these reasons this will be given the best comprehensibility score.

Java
The comprehensibility aspect of the thread safe cache is the criteria that differs the most between the golang and java solutions. What could be done in 170 lines in golang takes an addition 60 lines in java, coming to a total of 230 lines of code. In addition, the use of two nested classes contained within the overall public class seems to clutter the code. If this code was to be exposed as a public package, the better alternative would be to split up each class into its own file which could then be shared across the package. The excessive lines of code and general cluttering gives this a comprehensibility score of medium.

Performance

Golang
The golang performance for the cache tester was difficult to test because it would most likely thrive in an environment such that the expirations were used and compared to a cache that

implements expirations using goroutines. This comparison would likely show that the current implementation of call time expiration checking results in less cpu usage but more memory usage and could ultimately result in an out of memory error if the cache wasn't cleared every so often. Due to the lack of extensive, load testing on this cache, the pprof output that was produced was minimal and did not show any cpu level details. Overall the runtime averaged 290 ms over 10 tests which is reasonable considering the number of concurrent actions that were being performed. In a production situation there would need to be a more low level analysis of whether a program should be concurrent in order to use this or if it should only be used by previously concurrent programs. Overall the performance will be given a medium score.

Java

The java performance for the thread safe cache is similarly difficult to measure on its own, although the direct comparison with golang shows that they produce essentially the same performance. This is most likely due to the fact that both solutions were implemented in the same way. In order for a noticeable difference in performance to be observed, these solutions would need to be compared in a different environment, for example a networking cache as described in the introduction. Averaging 285ms over 10 test runs, this will also be given a medium performance score.

Table 6 shows the final results of each category.

Language	Correctness	Comprehensibility	Performance	Overall
Golang	✔	✔	🚧	✔
Java	✔	🚧	🚧	🚧

Table 6: Search, Insert, Delete Overall Criteria

References

[1] "HPROF: A Heap/CPU Profiling Tool", *Docs.oracle.com*, 2018. [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html. [Accessed: 07- Oct- 2018].

[2]"Getting started with Go CPU and memory profiling", *Flaviocopes.com*, 2018. [Online]. Available: https://flaviocopes.com/golang-profiling/. [Accessed: 07- Oct- 2018].

[3] A. Downey, *The Little Book of Semaphores*. Needham MA: Green Tea Press, 2016.