CSC464 assignment 1 - Jonathan Healy V00845624

The GitHub repository for this assignment can be found at:

https://github.com/noonespecial009/conc1

The Go code used for all of these problems was based off of code found at: github.com/soniakeys/LittleBookOfSemaphores. Some of these problems use a semaphore library based on Go Channels primarily that can be installed on your system by using this command:

```
go get -u github.com/soniakeys/LittleBookOfSemaphores/sem
```

The Java code used in these problems comes from various different sources.

All of the testing was done on a MacBook Pro running MacOS 10.12.3 with the terminal.

```
go version go1.8.1 darwin/amd64
```

```
java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

Problem 1: Readers - Writers

The Readers - Writers problem comes from the Little Book of Semaphores. When data structures are being modified by concurrent Users, serious issues can arise. In this problem, Writers must have exclusive access to the critical section and cannot enter to write data until all of the Readers have left. Readers, on the other hand, must wait until a Writer has left the critical section before they can read the data there. This problem mimics many real world situations involving databases and file systems where system security relies on having reads and writes accessing critical information at separate times.

Approach 1 (RW_go): Go implementation: source: github.com/soniakeys/LittleBookOfSemaphores

Approach 2 (RW_java): Java implementation

The two approaches used here are almost identical. Both approaches follow the pseudo code from the Little Book of Semaphores almost word for word:

```
#Variables:
int readers = 0
mutex = Semaphore(1)
roomEmpty = Semaphore(1)

#Writers:
roomEmpty.wait()
critical section
roomEmpty.signal()

#Readers:
mutex.wait()
readers++
if readers == 1
```

```
        roomEmpty.wait()
mutex.signal()
critical section
mutex.wait()
readers—
if readers == 0
        roomEmpty.signal()
mutex.signal()
```

Readers - Writers	Approach 1: Go	Approach 2: Java
Correctness	Approach 1 is correct and behaves as expected without any major anomalies over numerous test runs. One observation is that readers and writers will often go one after another in groups. For example most of the readers or writers will sometimes go first in a block but not always. With 1000 readers and 1000 writers display sentences will be occasionally cut off by the next print statement.	Approach 2 is also correct but Readers tend to hog the critical section in this approach more so versus the 1st. one. Like the Go implementation readers and writers will usually run in major blocks together. Unlike the Go implementation, with 1000 readers and 1000 writers there are no errors or anomalies seen in the print statements for the display output.
Comprehensibility	Uses WaitGroups and implementation for Semaphores programmed with Go Channels. Even though the code is almost identical to the Java version it is still easier to understand for someone like myself who actually has more programming experience with Java.	Uses Java semaphores which make for almost identical code to the Little Book of Semaphores and the Go implementation. The Java approach uses try/ catch statements which makes for a few more lines of code.
Performance	10 Readers/ 10 Writers: Avg. 0.9 ms. (A 100 ms delay was added at the end of the input as shown in the output below) 100 Readers/ 100 Writers: Avg. 1.2 ms. 1000 Readers/ 1000 Writers: Avg. 12 ms.	10 Readers/ 10 Writers: Avg. 6.0 ms. (A 100 ms delay was added at the end of the input as shown in the output below) 100 Readers/ 100 Writers: Avg. 28 ms. 1000 Readers/ 1000 Writers: Avg. 90 ms.

Readers - Writers	Approach 1: Go	Approach 2: Java
Output: 10 Readers/ 10 Writers	Physics$ ./RW_go writer 2 writes writer 7 writes writer 3 writes writer 4 writes writer 5 writes writer 6 writes reader 10 sees 6 bytes reader 4 sees 6 bytes reader 5 sees 6 bytes reader 6 sees 6 bytes reader 1 sees 6 bytes reader 7 sees 6 bytes reader 2 sees 6 bytes reader 8 sees 6 bytes reader 3 sees 6 bytes reader 9 sees 6 bytes writer 8 writes writer 1 writes writer 9 writes writer 10 writes 101.181786ms	Physics$ javac RW_java.java Physics$ java RW_java writer 0 writes reader 0 sees 1 bytes reader 3 sees 1 bytes reader 1 sees 1 bytes reader 2 sees 1 bytes writer 1 writes writer 2 writes writer 9 writes writer 3 writes writer 4 writes writer 5 writes writer 6 writes writer 7 writes reader 4 sees 9 bytes reader 7 sees 9 bytes reader 6 sees 9 bytes reader 5 sees 9 bytes reader 9 sees 9 bytes reader 8 sees 9 bytes writer 8 writes 106ms

Problem 2: Dining Philosophers

The Dining Philosophers problem comes from Dijkstra and is also covered in the Little Book of Semaphores. In this problem, there are five forks and five philosophers. For a philosopher to eat, the philosopher needs a left and a right fork. Forks are like resources that threads need to hold exclusively. The goal is to allow multiple philosophers to eat at the same time while avoiding starvation and deadlock. Having only one philosopher eating at once is a solution but not a good use of resources. This is a problem that highlights synchronization issues in numerous types of applications and forces one to think about ways to solve deadlock and resource starvation.

Approach 1 (DP_go): Go implementation: source: github.com/soniakeys/LittleBookOfSemaphores

Approach 2 (DP_java): Java implementation: source: github.com/eugenp/tutorials/blob/master/core-java-concurrency/src/main/java/com/baeldung/concurrent/diningphilosophers

Dining Philosophers	**Approach 1: Go**	**Approach 2: Java**
Correctness	More than one philosopher will often eat at the same time. With a 1000 threads initialized, there will be a couple of display errors in the output where a print statement from one thread gets muddled up with another thread's output. The Go version runs significantly faster.	Two philosophers will only be eating at the same time occasionally. At one point, while testing with 1000 threads, Philosopher 3 just ate over and over about 50 times in a row. This happens more frequently then it should obviously and the reason for this has not been identified yet.
Comprehensibility	The Go code is clear and makes intuitive sense using WaitGroups and an implementation for Semaphores programmed with Go Channels. The logic used in this implementation is almost exactly the same as what is given as pseudocode in The Little Book of Semaphores and this adds to the understandability of it all.	This Java approach uses synchronized blocks which are very intuitive. A 100 ms. delay was added at the end of the program because I did not know how to implement something that was equivalent to WaitGroups that are found in Go. Comprehensibility generally does not score as high in Java versus what it does in Go.
Performance	5 Meals: Avg. 1.2 ms. (A 100 ms delay was added at the end of the input as shown in the output below) 50 Meals: Avg. 3.1 ms. 1000 Meals: Avg. 18 ms.	5 Meals: Avg. 2.9 ms. (A 100 ms delay was added at the end of the input as shown in the output below) 50 Meals: Avg. 14 ms. 1000 Meals: Avg. 234 ms.

Output:
5 Meals
	`phil 4 thinking`
	`phil 1 thinking`
	`phil 1 picked up right fork`
	`phil 4 picked up right fork`
	`phil 1 picked up left fork`
	`phil 4 picked up left fork`
	`phil 1 eats bite # 1`
	`phil 4 eats bite # 1`
	`phil 1 full, returns forks`
	`phil 4 full, returns forks`
	`phil 0 thinking`
	`phil 2 thinking`
	`phil 0 picked up right fork`
	`phil 2 picked up right fork`
	`phil 0 picked up left fork`
	`phil 2 picked up left fork`
	`phil 0 eats bite # 1`
	`phil 2 eats bite # 1`
	`phil 0 full, returns forks`
	`phil 2 full, returns forks`
	`phil 3 thinking`
	`phil 3 picked up right fork`
	`phil 3 picked up left fork`
	`phil 3 eats bite # 1`
	`phil 3 full, returns forks`
	`101.179849ms`
		`phil 1 thinking`
		`phil 1 picked up left fork`
		`phil 1 picked up right fork - eating`
		`phil 1 put down right fork`
		`phil 5 thinking`
		`phil 3 thinking`
		`phil 3 picked up left fork`
		`phil 3 picked up right fork - eating`
		`phil 3 put down right fork`
		`phil 3 put down left fork. Full`
		`phil 2 thinking`
		`phil 2 picked up left fork`
		`phil 2 picked up right fork - eating`
		`phil 2 put down right fork`
		`phil 2 put down left fork. Full`
		`phil 1 put down left fork. Full`
		`phil 4 thinking`
		`phil 4 picked up left fork`
		`phil 4 picked up right fork - eating`
		`phil 4 put down right fork`
		`phil 4 put down left fork. Full`
		`phil 5 picked up left fork`
		`phil 5 picked up right fork - eating`
		`phil 5 put down right fork`
		`phil 5 put down left fork. Full`
		`103ms`

Problem 3: Cigarette Smokers

In this problem there is an Agent who has the three ingredients needed to construct a cigarette - tobacco, paper, and matches. There are also three smokers who each only have one of these three ingredients. The agent chooses two ingredients at random and makes them available to the smokers. The agent here represents an Operating System and the smokers are like applications who need certain resources. Applications are waiting and when the resources they need become available they should be woken up. Applications that are not needed should remain sleeping.

Approach 1 (CS_go): Go implementation: source: github.com/soniakeys/LittleBookOfSemaphores

Approach 2 (CS_java): Java implementation: source: paulamarcu.wordpress.com/2014/08/16/cigarette-smokers-problem

Cigarette Smokers	**Approach 1: Go**	**Approach 2: Java**
Correctness	The Go implementation is correct and shows no weird display artifacts when using multiple runs, unlike what was seen with Problems 1 and 2. Interestingly there is not a huge performance difference between the two implementations as there was with Problems 1 and 2, comparing Go to Java.	Like the Go implementation, the Java implementation is correct and shows no anomalies over multiple runs using 1000 rounds. Performance of this implementation in terms of execution time is comparable to the Go implementation up to 1000 rounds. In terms of performance the Java code starts out slow but then passes the Go implementation with more rounds.
Comprehensibility	The Go code uses WaitGroups and implementation for Semaphores programmed with Go Channels. This code follows the pseudo code from The Little Book of Semaphores. Atomic counters are used which are a little confusing at first but offer an interesting way to solve concurrency problems in Go.	This implementation uses synchronized blocks, semaphores and a Countdown Latch. The code is a little more complicated than the Go version but mainly because it deviates from the approach used in The Little Book of Semaphores and introduces alternative ways to do things.
Performance	5 Rounds: Avg. 1.2 ms. (A 100 ms delay was added at the end of the input as shown in the output below) 100 Rounds: Avg. 2.2 ms. 1000 Rounds: Avg. 13 ms.	5 Rounds: Avg. 3.8 ms. (A 100 ms delay was added at the end of the input as shown in the output below) 100 Rounds: Avg. 5.1 ms. 1000 Rounds: Avg. 7.2 ms.

Cigarette Smokers	Approach 1: Go	Approach 2: Java
Output: 5 Rounds	agent provides paper and a match smoker with tobacco makes cigarette smoker with tobacco smokes agent provides tobacco and a match smoker with paper makes cigarette smoker with paper smokes agent provides tobacco and paper smoker with matches makes cigarette smoker with matches smokes agent provides paper and a match smoker with tobacco makes cigarette smoker with tobacco smokes agent provides tobacco and a match smoker with paper makes cigarette smoker with paper smokes 101.183594ms	agent provides paper and a match smoker with tobacco makes cigarette smoker with tobacco smokes agent provides paper and tobacco smoker with matches makes cigarette smoker with matches smokes agent provides paper and a match smoker with tobacco makes cigarette smoker with tobacco smokes agent provides paper and tobacco smoker with matches makes cigarette smoker with matches smokes agent provides paper and tobacco smoker with matches makes cigarette smoker with matches smokes Agent not available. Agent not available. Agent not available. 104ms

Problem 4: Dining Savages

Savages eat from a pot that contains stewed missionary. If the pot is empty, the savages have to wait for the cook to fill it again. The logic here is similar to processes waiting on some data to become available. It doesn't matter which processes get the data so long as it gets taken care of. Every so often, another process - the cook - fills the pot with a batch of data and the savages race to try to process the individual portions. This seems to be a problem that is faced by operating systems, for one. Data could be streamed into a computer and the OS could have processes that are ready to process a piece of that data on a first come - first serve basis.

Approach 1 (DS_go): Go Semaphores implementation: source: github.com/soniakeys/LittleBookOfSemaphores

Approach 2: (DS_atomic): Go Atomic implementation: source: github.com/soniakeys/LittleBookOfSemaphores

Dining Savages	Approach 1: Go Semaphores	Approach 2: Go Atomic
Correctness	This implementation is based on the pseudo code found in The Little Book of Semaphores and shows error free output for up to the 3000 pot fillings used for testing. Four different savages will, more often than not, get to eat from the same pot filling.	This implementation is basically correct, however often the same one or two savages will eat all of the 4 servings that have been prepared by the cook. This behaviour doesn't disqualify this solution from being correct but it definitely doesn't look right. Only rarely will more than 2 savages eat from the same pot.

Dining Savages	Approach 1: Go Semaphores	Approach 2: Go Atomic
Comprehensibility	Code is clear and almost exactly matches the solution found in The Little Book of Semaphores. Semaphores are implemented in a similar way to the Go solutions for problems 1-3. Code is easy to understand - a definite benefit to using Go. There are only a few lines of code needed for this implementation.	The atomic solution is not using the Semaphore library like the other Go solutions to the previous problems presented before. The sync/atomic library is used here for atomic counters which can be accessed by multiple GoRoutines. A WaitGroup is used with this solution as well as 3 different channels for passing information between threads. This solution is somewhat more complicated to understand.
Performance	3 Pot Fillings: Avg. 0.31 ms. 30 Pot Fillings: Avg. 1.4 ms. 3000 Pot Fillings: Avg. 138 ms.	3 Pot Fillings: Avg. 0.72 ms. 30 Pot Fillings: Avg. 6.7 ms. 3000 Pot Fillings: Avg. 582 ms.

Output:
3 Pot Fillings
	`3 pot fillings:`
	`cook puts 4 servings in pot`
	`savage 6 gets serving from pot`
	`savage 6 eats`
	`savage 2 gets serving from pot`
	`savage 2 eats`
	`savage 1 gets serving from pot`
	`savage 1 eats`
	`savage 4 gets serving from pot`
	`savage 4 eats`
	`cook puts 4 servings in pot`
	`savage 3 gets serving from pot`
	`savage 3 eats`
	`savage 5 gets serving from pot`
	`savage 5 eats`
	`savage 6 gets serving from pot`
	`savage 6 eats`
	`savage 2 gets serving from pot`
	`savage 2 eats`
	`cook puts 4 servings in pot`
	`savage 1 gets serving from pot`
	`savage 1 eats`
	`savage 4 gets serving from pot`
	`savage 4 eats`
	`savage 3 gets serving from pot`
	`savage 3 eats`
	`savage 5 gets serving from pot`
	`savage 5 eats`
	`311.017µs`
		`cook awake, starts cooking`
		`cook puts 4 servings in pot`
		`savage 3 eats ( 3 servings in pot)`
		`savage 3 eats ( 1 servings in pot)`
		`savage 3 eats ( 0 servings in pot)`
		`cook awake, starts cooking`
		`savage 4 eats ( 2 servings in pot)`
		`cook puts 4 servings in pot`
		`savage 6 eats ( 3 servings in pot)`
		`savage 6 eats ( 1 servings in pot)`
		`savage 6 eats ( 0 servings in pot)`
		`savage 3 eats ( 2 servings in pot)`
		`cook awake, starts cooking`
		`cook puts 4 servings in pot`
		`cook leaves`
		`savage 6 eats ( 3 servings in pot)`
		`savage 1 eats ( 2 servings in pot)`
		`savage 6 eats ( 1 servings in pot)`
		`savage 1 eats ( 0 servings in pot)`
		`simulation ends`
		`604.695µs`

Problem 5: Barbershop

This is another problem originally proposed by Dijkstra. There is a barbershop with a number of chairs for waiting. If there are no customers the barber goes to sleep. If a customer enters and there are no chairs available to wait, the customer leaves. There is one barber chair. If a customer comes in and the barber is asleep, the customer wakes up the barber. This problem emulates inter process communication and synchronization between multiple operating system processes. There is no point in a host process being awake if there are no processes waiting to being served. There is also only enough room for a certain number of processes to queue and wait for a turn.

Approach 1 (BS_go): Go Semaphores implementation: source: github.com/soniakeys/LittleBookOfSemaphores

Approach 2 (BS_m2): Go Mutex 2 implementation: source: github.com/soniakeys/LittleBookOfSemaphores

Barbershop	**Approach 1: Go Semaphores**	**Approach 2: Go Mutex 2**
Correctness	Output gets jumbled sometimes and occasionally looks incoherent. With running 4000 waiting chairs and 6000 customers, the behaviour of this implementation seems mostly fine. Occasionally there are numerous customers arriving quickly at the same time and leaving because all of the chairs are full. This happens when the barber is sleeping although he eventually does wake up.	This implementation seems more correct than Approach 1. The barber stays busy and there aren't large numbers of customers leaving because the waiting area is full. There is no garbled output or other errors that are easily discernible. In a way, the output from this implementation almost looks like it was programmed without threads as it churns along almost perfectly except for the odd thread being out of order.
Comprehensibility	This approach uses Sonia Keys semaphore implementation and is almost exactly like the solution presented in The Little Book of Semaphores. A WaitGroup is used to ensure that all of the GoRoutines have a chance to finish. This implementation is more comprehensible given that is follows a familiar pattern. This solution also uses only a few lines of code.	Approach 2 is a non-semaphore version that uses mutexes and channels to communicate. An unbuffered channel called barberRoom is used so that a customer can send his customer number to the Barber. Another channel called cutDone is used so that the Barber can send a message saying that the haircut has finished. After getting used to the way Channels work this solution becomes very easy to read.
Performance	4 chairs/ 6 customers Avg. 0.27 ms. 40 chairs/ 60 customers Avg. 1.8 ms. 4000 chairs/ 6000 customers Avg. 165 ms.	4 chairs/ 6 customers Avg. 5.1 ms. 40 chairs/ 60 customers Avg. 49 ms. 4000 chairs/ 6000 customers Avg. 5000 ms.

Barbershop	Approach 1: Go Semaphores	Approach 2: Go Mutex 2
Output: 4 haircuts/ 6 customers	barber sleeping customer 1 arrives, 0 customers customer 1 waits barber cutting hair customer 1 gets hair cut barber sleeping customer 6 arrives, 1 customers customer 6 waits barber cutting hair customer 6 gets hair cut barber sleeping customer 2 arrives, 2 customers customer 2 waits barber cutting hair customer 2 gets hair cut barber sleeping customer 3 arrives, 3 customers customer 3 waits barber cutting hair customer 3 gets hair cut barber sleeping customer 4 arrives, 4 customers customer 4 shop full, leaves customer 5 arrives, 4 customers customer 5 shop full, leaves customer 1 leaves with hair cut customer 6 leaves with hair cut customer 2 leaves with hair cut customer 3 leaves with hair cut 268.312µs	barber sleeping customer 2 arrives, 0 customers customer 2 waits customer 6 arrives, 1 customers barber wakes, cuts customer 2 hair customer 6 waits customer 3 arrives, 2 customers customer 3 waits customer 4 arrives, 3 customers customer 4 waits customer 5 arrives, 4 customers customer 5 ,shop full, leaves customer 1 arrives, 4 customers customer 1 ,shop full, leaves customer 2 getting hair cut customer 2 leaves with hair cut barber sleeping barber wakes, cuts customer 6 hair customer 6 getting hair cut customer 6 leaves with hair cut barber sleeping barber wakes, cuts customer 3 hair customer 3 getting hair cut customer 3 leaves with hair cut barber sleeping barber wakes, cuts customer 4 hair customer 4 getting hair cut customer 4 leaves with hair cut barber sleeping 5.457417ms

Problem 6: Rollercoaster

A roller coaster can only hold a certain number people. Once it is fully loaded, it can race around the track. All passengers must then unload before new passengers can embark for another run. This is similar to a program needing to access a number of resources to perform a task for a certain period of time and then needing to do the same thing again with a new set of resources. It seems similar to accessing data during cloud computing using map/ reduce although I could be completely wrong.

Approach 1 (RoCo_go): Go Semaphores implementation: source: github.com/soniakeys/LittleBookOfSemaphores

Approach 2 (RoCoChan_go): Go Channels implementation: source: github.com/soniakeys/LittleBookOfSemaphores

Rollercoaster	Approach 1: Go Semaphores	Approach 2: Go Channels
Correctness	Approach 1 using Go Semaphores is completely correct even after examining numerous executions with the 3000 roller coaster runs used for testing. The performance of both these implementations is almost identical considering the running time as seen in the performance section.	The output from Approach 2 is identical to that from Approach 1, even during the 3000 runs that were used for performance testing. It would be interesting to test these implementations against a sequential implementation as the output is so predictable.
Comprehensibility	Comprehensibility is extremely high with Approach 1 as it follows the pseudo code as presented in The Little Book of Semaphores. The Go code is very compact especially as the semaphore implementation based on Go Channels is abstracted away. This approach was chosen because it closely follows The Little Book of Semaphore and this makes Approach 1 considerably more comprehensible compared to Approach 2.	Approach 2 is less comprehensible when compared to Approach 1 but only because Approach 2 doesn't follow the Semaphore approach and it can take some time to wrap one's head around Go Channels if one is not familiar with them. Having said that, Approach 2 is a very elegant solution and Channels in Go are almost too easy to use. Both of these approaches also use WaitGroups to ensure that all of the Go routines finish before the main thread exits.
Performance	3 Runs Avg. 0.21 ms. 30 Runs Avg. 1.0 ms. 3000 Runs Avg. 144 ms.	3 Runs Avg. 0.24 ms 30 Runs Avg. 1.2 ms. 3000 Runs Avg. 153 ms.

Rollercoaster	Approach 1: Go Semaphores	Approach 2: Go Channels
Output 3 Runs	car ready to load passenger boards passenger boards passenger boards passenger boards car runs car ready to unload passenger unboards passenger unboards passenger unboards passenger unboards car ready to load passenger boards passenger boards passenger boards passenger boards car runs car ready to unload passenger unboards passenger unboards passenger unboards passenger unboards car ready to load passenger boards passenger boards passenger boards passenger boards car runs car ready to unload passenger unboards passenger unboards passenger unboards passenger unboards car ready to load 216.823µs	car ready to load passenger boards passenger boards passenger boards passenger boards car runs car ready to unload passenger unboards passenger unboards passenger unboards passenger unboards car ready to load passenger boards passenger boards passenger boards passenger boards car runs car ready to unload passenger unboards passenger unboards passenger unboards passenger unboards car ready to load passenger boards passenger boards passenger boards passenger boards car runs car ready to unload passenger unboards passenger unboards passenger unboards passenger unboards car ready to load 204.1µs