HW6 - Answers

Answer 1:

i) 3 things learned from the talk

   i) Concurrency is not parallelism. Concurrency is meant to interact with the real world or simulate a real world
   ii) A program can still be concurrent even on only one processor, but it can’t be parallel in such a scenario
   iii) Go has no locks, no conditional variables, and no callbacks unlike a thread. This way Goroutines can be used to implement complex code much more easily than with thread

ii) 3 things agree with

   i) A goroutine executes a function but doesn’t make a caller wait. This is definitely a huge advantage. It would help developers focus on getting the functionality right rather than focusing on implementing locks, mutexes etc.,
   ii) Goroutine has a stack which increases or shrinks as required. This is a boon because the developers will not have to define anything and let Go take care of allocations
   iii) The Channels in a goroutine are designed to both communicate and synchronize. This is a very good design feature to have. This would help avoid the complexities which arise when multiple threads are used to carry out operations.

iii) 3 things disagree with

   i) The Daisy-chain functionality provided by the go-routine seems simple but it would take a lot of effort to keep track of individual channels and how they communicate with each other. It would be very easy to get it wrong and have wrong channels communicating with each other if sufficient care is not taken. The syntax in which a channel is defined could be made a little more verbose.
   ii) The concept of using Replica is very interesting in Go. However am not sure how it would help if the servers it is communicating with are always slow for one reason or the other. If it is still waiting for the first response, then Go would end up waiting for a while and not return to the calling function.
   iii) Also it was mentioned that Go has the “sync” and “sync/atomic” packages that provide mutexes, conditional variables etc., If the whole purpose of Go is to encourage concurrency and provide a cleaner solution for developers to implement these, then why provide these in the first place. Am also not sure how these would work in conjunction with the other goroutines.

iv) A GO Routine is defined as an independently executing function that is launched by a go statement. It has its own call stack which grows and shrinks as required. It is also very cheap and it’s possible to have a huge number of goroutines. A GoRoutine is different from a thread as one thread might have thousands of goroutines. Goroutines are multiplexed dynamically onto threads as needed to keep it running. A goroutine can be considered to be a very cheap thread.

v) Routine

```
func main() {
    c := make(chan string)
    go boring("boring!", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <-c)
    }
    fmt.Println("You're boring; I'm leaving.")
}
func boring(msg string, c chan string) {
    for i := 0; ; i++ {
        c <- fmt.Sprintf("%s %d", msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

A detailed explanation of the routine is provided below:

i) The main function gets executed first and it establishes a Channel in go using the command “c := make(chan string)”. The channel helps to provide a connection between two goroutines thereby enabling them to communicate with each other

ii) Once a Channel is established, the main method invokes the“func boring(msg string, c chan string)” method by making the call “go boring("boring!", c). This launches a goroutine. The go statement executes the function but ensures that the caller doesn’t have to wait

iii) Inside the “func boring(msg string, c chan string)” method, a for loop is executed. Since a channel was created between the main and the boring goroutines, they are able to communicate now.

iv) The main function on executing the command “<-c” waits for the value to be received. Similarly the boring method waits for the receiver to be ready when it executes the “c<-“ call because it has to send on that channel. It has to be noted that the arrow before or after “c” indicate the direction of data flow in the channel. “c<-“ indicates sending on a channel and “<-c” indicates that it is receiving from a channel.

v) Therefore the boring goroutine returns a formatted string on the channel which in turn is received by the main method and the value is then displayed to the user.

vi) After the for loop in the main method is executed for 5 times and the corresponding output is printed out, it will finally display the values “You're boring; I'm leaving”. The main program will then exit.

The Expected output of the sample code provided is as follows:

```
You say: "boring! 0"
You say: "boring! 1"
You say: "boring! 2"
You say: "boring! 3"
You say: "boring! 4"
You're boring; I'm leaving.
```

Answer 2:

Code to print numbers from 1 to 100 using two threads. T1 prints Odd Numbers and T2 prints Even Numbers.

Code Name: NumberIncrementer.java

https://docs.google.com/a/utexas.edu/file/d/0BwGliKJiLohnZ0JaSEtFRkZKLUk

Answer 3:

i) The reason for the non-deterministic result returned when two threads T1 and T2 increment an unguarded int variable *counter* N times can be summarized as follows. The two threads T1 and T2 try to increment the same variable *counter* at almost the same time without synchronizing with each other. As a result, the end values end up being inconsistent. During code execution, after every loop, we end up with a *counter* value which is either one or two greater than before, depending on the value observed when T2 starts its operation. Thread T1 grabs the variable "*counter*", increments it and writes it back. Now if Thread T2 reads the variable before T1 finishes writing it back, it will read the same value that T1 initially encountered. So T2 will increment the same value and when T2 writes the incremented value back, it is basically the same value that T1 had already written. The value would be incremented by just 1 though both T1 and T2 operated on it.. In cases where T1 and T2 are able to increment the *counter* value after it has being incremented by the other thread, then the value of *counter* would increase in increments of 2. This is the main reason for the inconsistency encountered.
ii) The Minimum value that the *counter* can have is N and the maximum value is 2N. Based on the explanation in the previous point, it is easy to understand that the maximum value that is possible is 2N. If the threads T1 and T2 were able to get access to the *counter* one after the other then they would be able to consistently increment the value, and therefore the maximum value thats possible is 2N. Similarly if the threads T1 and T2 end up accessing the same *counter* value at the same time, they would encounter the same value of the *counter,* therefore the minimum value that the *counter* can get incremented to is N.

Answer 4:

The benchmark results returned after implementing the ConcurrentHashMap, and without changing any parameters are:

```
Regular:Total time (ms) = 1209
Concurrent:Total time (ms) = 298
```

(or)

```
Regular:Total time (ms) = 1206
Concurrent:Total time (ms) = 240
```

The values are not always the same, during repeated executions. However it is always noticed that ConcurrentHashMapCache access is significantly faster than the RegularHashMapCache.

Code Name: ConcDS.java

https://docs.google.com/a/utexas.edu/file/d/0BwGliKJiLohncGx3NmpVY21KYnM

Answer 5:

i) The main problem noticed while executing the initial code was the lack of consistency. Sometimes the code would run fine and sometimes it would just hang. This is due to the fact that Deadlocks occur among the multiple threads that are being executed.

To elaborate further, consider this scenario. The agent randomly chooses two different ingredients as per the example. So if the agent puts out tobacco and paper, it is “expected” that a smoker with only the “match” should pick it up and make cigarettes. However, this might not always be the case. Below is a list of Agents and the ingredients they put out.

Agent A: Tobacco, Paper
Agent B: Tobacco, Matches
Agent C: Paper, Matches

There is one smoker “smokerMatches” waiting on the tobacco and another smoker “smokerTobacco” waiting on the Paper. Now since Agent A puts out the Tobacco and Paper, the smoker with matches might get unblocked. Similarly the smoker with tobacco waiting on the paper may also get unblocked. This would cause the first thread to block on paper and the second thread to block on match. This causes a deadlock and the code would hang.

ii) The strategy we can follow to solve this issue is the one proposed by Parnas to have 3 helper threads called “Pushers” which will respond to the signal from the agents. The pushers will also keep track of the ingredients and signal the appropriate smoker whenever there is a match found

Code Name: CSPSolution.java

https://docs.google.com/a/utexas.edu/file/d/0BwGliKJiLohnU1NkNHZuZXRSX1k