Background Report: Go Exercise

JI YOON OK, jyo1031@gmail.com
BENJAMIN GEE, bmgee27@gmail.com
CHRISTINE SAM, xtinesam@gmail.com

---

1 CONCURRENCY

1.1 Background

Concurrent programs are programs that contain more than one active thread of control, in contrast to sequential programs which only have a single active thread of control [1]. Each stream itself can run as a single sequential program, but have the additional ability to communicate with other threads. While the origin of concurrency dates back to the 1960s with Edsger Dijkstra's paper on mutual exclusion, it was the availability of multi-core and multiprocessor systems which fueled an interest in concurrency [2].

Writing concurrent programs, however, is not a simple task. The main challenge with concurrency is that the execution of the program becomes non-deterministic, due to the unpredictable order of operations from multiple threads [3]. Because threads share access to variables and objects, this introduces the possibility of two types of errors: thread interference and memory inconsistency [4]. Thread interference is when two different threads interleave when operating on the same data [4]. On the other hand, memory inconsistency arises when different threads have inconsistent results despite viewing the same data [4]. Both of these problems, however, can be solved through synchronization.

Synchronization, specifically process synchronization, is when the execution of threads are coordinated to achieve a certain outcome [4]. There are many different synchronization techniques, including the use of locks, mutexes and semaphores. However, although synchronization can help with thread interference and memory inconsistency issues, it can in turn introduce new problems. One of the major issues is deadlocks, which occurs when a set of processes are permanently blocked because each are waiting for a resource that another process has locked [4]. Other problems that can arise are starvation, which is when a thread can’t make any progress because it is unable to obtain access to resources it needs, and livelock, which is when threads are constantly changing their states in response to each other and hence cannot make progress [4].

Finally, the last major difficulty with writing concurrent programs is to keep resource consumption low. Threads are generally considered to be expensive because of the large amount of memory needed to allocate a thread as well as overhead costs. Also, to switch from one thread to another usually requires a context switch, which is also very computationally expensive.

1.2 Importance

Concurrency plays a very important role in modern day programming. Concurrency arises in programming for three main reasons. First, with many modern day programs, particularly in respects to servers and graphical applications, there are multiple, independent tasks running that must be monitored at the same time [1]. The best way to structure the program is to represent each task as it's own running thread [1]. Second, with many programs, the runtime of one task is almost never continuous and is often interrupted by something [1]. In an event of an interruption, the program needs a thread to represent what it was doing before the interrupt, and another to handle the interruption [1]. Moreover, systems for real-time control usually include many processors; each connected to separate physical devices. Each processor has its own thread and task and must communicate with one another in order to accomplish one overall task for the system [1]. Lastly, in connection to the last point, having processes run over multiple processors can greatly increase performance of the program [1].

In particular, concurrency is used to improve performance and reduce latency when faced with delay or slow responses. Concurrency becomes useful when applications are accessing slow I/O devices. Instead of waiting for the device to respond, the CPU can exploit concurrency to use that time to perform other important processes [6]. At a larger scale, concurrency can be used by servers to service multiple clients concurrently and disallow one, slow client from blocking service to other clients [6].

2 HOW CONCURRENCY IS IMPLEMENTED IN GO

Concurrency in Go is implemented with goroutines [7]. Goroutine is a function that is executed independent of the main function with the call to the go keyword [7]. Each goroutine maintains it's own call stack, but is lightweight and cheap enough to have multiple instances created at a time [7]. However, a goroutine is not a thread in itself. It is multiplexed onto a single thread, and a single thread can have multiple goroutines running on it simultaneously [7].

With multiple goroutines, it is useful for them to be able to communicate and pass information between each other. This can be accomplished with Go channels [7]. Channels are created by executing the make function and values are passed into the channel using an arrow pointed to the channel name, like "c <- value" [7]. A value is received from a channel with an arrow pointer out of the channel name, like "<-c" [7]. When communicating between two goroutines, one must be the sender and the other the receiver. Both the sender and receiver must be ready in order for the communication to be successful. Thus, channels not only help goroutines communicate with each other, but also synchronize with each other [7].

Contrary to mainstream programming languages like Java, C++ and Python, communication between threads does not require the careful access to shared memory [8]. In the above languages, typically shared data structures are protected with locks and threads contend for their access [8]. However, with goroutines and channels, the communication follows the Communication Sequential Process (CSP) model [8]. Instead of using locks when accessing shared data, channels will pass references to shared

data to the goroutines. Therefore, whichever goroutine is ready to receive the reference at that time will gain access to the shared data and no other goroutine may access it simultaneously [8].

The main advantage of concurrency in Go is that it is lightweight, which means many goroutines can run without affecting performance [9]. It is easy to implement within the code by calling the go keyword before designated concurrent method. This method does away with extraneous processes such as thread schedulers in Java [9]. Executing goroutines are self-sufficient, meaning that they keep track of stack and heap variable references and does not need to be referred to themselves to avoid garbage collection. Upon finishing a goroutine, it will release all resources automatically [9].

3 CONCURRENCY IN OTHER LANGUAGES

3.1 Concurrency in Java

Concurrency is managed by 4 main tools: processes, threads, green threads and goroutines [5]. The two that interest us the most are threads in Java and goroutines in Go.

Java uses threads to manage concurrency, which means that an address space is shared among threads running in the same process [5]. A thread, like a goroutine, is a lightweight process that possesses it's own call stack, but can access shared data of other threads in the same process [14]. Within a Java application, there is by default one process and running several threads will help achieve an asynchronous behavior [14].

An important point to keep in mind when talking about concurrency is synchronization or how two concurrent processes communicate with each other [5]. Java uses objects to communicate about tasks between concurrent processes. But this process involves overhead of obtaining locks on the shared data [5]. Because threads have their own call stack, but can also access shared data, two problems arise: visibility and access problems. Visibility problems occur when thread A reads data that is later changed by thread B, but is unaware of this change and access problems occur when both thread A and B try to change the same shared data at the same time. Both of these problems lead to deadlocks and incorrect return of data [14]. A deadlock occurs when if all processes are waiting for an event that another process has to cause. This creates a cyclic dependency of processes which halts a program [14]. To combat these problems, Java provides locks to protect parts of the code from being executed by different threads at the same time. The simplest way to incorporate this into the code is by the synchronized keyword ahead of methods or classes [14]. By including this keyword it ensures that only a single thread executes a block of code at a time and that each thread sees the effects of the previous thread that executed that part of the code [14]. On the other hand, Go uses channels that are shared between units of work to achieve synchronization. Channels are essentially a pipe that a unit of work can read or write to. So instead of having to synchronize access, units of work can easily communicate over shared channels [5].

When designing a concurrent program in Java, it creates a concern about performance issues such as allocation of thread pools. In Go, however, there is no need to worry about performance since

goroutines can be used to track concurrent units of work, and what is determined as most efficient gets multiplexed onto OS threads [5]. This allows developers to focus mainly on optimal concurrency design.

3.2 Actor Model for Concurrency

Another popular model for concurrency is the actor model, which is utilized by languages such as Erlang and Scala. In this model, there are multiple lightweight threads which are referred to as actors [10]. Each actor has its own buffered mailbox which is used to communicate with other actors [10]. Actors send messages asynchronously to each others mailboxes where they are stored until they are received and processed by the owner of the mailbox [11]. Upon receiving a message, there are three possible actions an actor can take: (1) create more actors, (2) send messages to other actors, and (3) assume a new state for the next message it will receive [11].

What makes this model unique is that while actors can have mutable state, no other actors can have access to it [10]. Actors can only obtain information about another actor's state through messages. Because state is not shared, there is no need for explicit locking of data structures. Furthermore, messages passed between actors must be immutable and they are queued and dequeued atomically, which eliminates the problems of race conditions [15].

There are a few advantages of the actor model over other models of concurrency. As mentioned above, one of the major problems the actor model eliminates is the need for synchronization of shared state. This avoids the complications of creating and coordinating locks. Another advantage is that actors are lightweight threads. A single operating system thread can map to one actor as well as multiple inactive actors [12]. The actor model is also well suited for cloud based environments because it is easy to scale up in areas of high activity by increasing the number of actors [16]. Finally, this model is resilient to failures that may occur in a single actor due to the fact that actors are fairly isolated from other actors [16].

The actor model, however, also has a few shortcomings. First of all, although the model eliminates the need of locks, this, however, doesn’t completely prevent deadlocks or starvation. For example, a deadlock can be introduced by having two actors both waiting to receive a message from each other (and therefore causing a receive block) [15]. Secondly, because of the lack of global state since actors must be isolated threads, this model may not be suitable for problems that cannot be broken up into individual tasks [16].

One of the main differences between the actor model and the CSP model used by Go is that message passing is asynchronous in the actor model and synchronous in the CSP model. Having a fully synchronous message system is easier to reason about since a channel can only ever be blocked on a single message. Another difference between the two models is in the actor based model, messages are directly addressed to process with the channels of communication being anonymous, whereas in the CSP model, messages are addressed to channels with the processes being anonymous [17]. In terms of similarities, both models have very lightweight threads, with multiple process threads being able to run on a single operating system thread. Also, both methods do not require the explicit use of locks to synchronize state.

4 GO EXERCISE TIMER

4.1 Project Overview

To demonstrate the use of concurrency in Go, the project we will be creating is an exercise timer. In our application, the user has the ability to choose a number of circuits. For each circuit, they can set the amount of time to exercise, as well as the amount of short breaks in between each exercise. Each circuit will essentially consist of a number of exercise and rest intervals, with the number of intervals being set by the user. In between circuits, the user can set a time for a longer break. Additionally, if the user chooses, they can include a warmup and cool down time at the beginning and end respectively. Finally, the application will allow the user to pause and resume the the timer at any point. Figure 1 below shows an example of what the timer could look like.

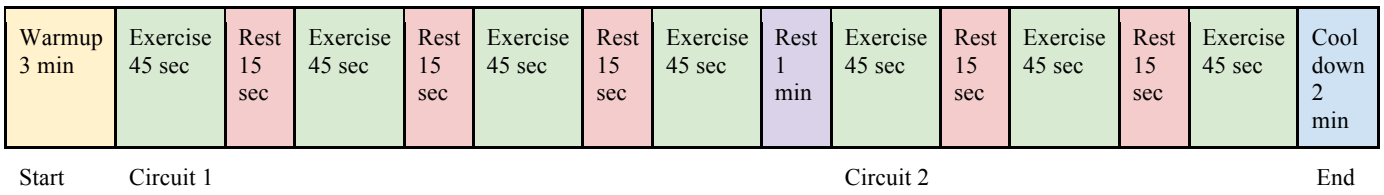

Figure 1. Sample timer program. This is an example of a timer program that can be created in our application. In this specific timer, a user has created two different circuits with a 1minute rest in between, as well as a warmup and cool down time.

4.2 Technical Elements

The technical elements of our projects that utilize the concurrency features of Go are the use of goroutines to create threads for specific timers and channels to communicate between timers. More specifically, the two will be used as follows:

Goroutines

We will have a goroutine for each of the following “timer”: main program, warm up, cool down, exercise, short break, long break, pause/resume, and service manager. The main program goroutine is implicitly launched at the start of the entire program. If the user chooses to have a warm up, the warm up go routine is then launched. Otherwise, the exercise goroutine is launched when user starts the exercise. It will run for the specified time. When exercise goroutine is finished, it will message the service manager goroutine, which will then launch either the short break or long break goroutine based on user’s settings. When the break timer is finished, it again messages the service manager, which launches the exercise goroutine again. At the end of specified number of exercise circuits, the user can choose to start the cool down, which will again be then launched by service manager. Otherwise, control returns to the main goroutine where it will wait the user to end the program or start again. During any time of the execution of the timer, the user can also pause the timer. During the pause, control goes to the pause/resume goroutine. When the user is ready to start again, control transfers back to the previous goroutine (where the user left off). The service manager goroutine will be a special goroutine that will act as a coordinator to handle messages from the goroutines.

Channels

Once we have our goroutines, we then need channels for the separate goroutines to communicate with each other. We will be using unbuffered channels, which means the goroutines will be blocked until the message is received. Other than the main and the service manager goroutine, every other goroutine will have its own channel. Go allows different types of channels (boolean, string, int, etc.). For our purposes, having the channels as boolean values would be sufficient as we only need to signify whether the goroutine is "done" its turn.

4.3 Plan to Achieve 100% Milestone

To reach the 100% milestone, we aim to complete a fully functional timer program. We will do this by first implementing the main goroutine function. Then we will slowly build on this by added more goroutines, starting with the essential ones such as the exercise goroutine and the break goroutine. After that, we can start adding basic communication channels. Initially, we will have our program have fixed times and circuits. By the end, we will modify it to accept user input, allowing the user to fully customize their timer. Once all program functionalities are in place, we can create project description that illustrates what we had accomplished in completing the project. This project description will also highlight the main technical details used to implement concurrency in our program as well as the overall significance of our project.

REFERENCES

[1] Michael L. Scott. 2000. Chapter 12 Concurrency [pdf]. Retrieved from https://www.cs.rochester.edu/~scott/456/local_only/chap12.pdf

[2] Leslie Lamport. 2015. Turing Lecture: The Computer Science of Concurrency: The Early Years. *Commun. ACM* 58, 6 (Jun. 2015), 58–78. DOI: http://dx.doi.org/10.1145/2771951

[3] Corky Cartwright. 2000. What is Concurrent Programming? Retrieved from https://www.cs.rice.edu/~cork/book/node96.html

[4] Oracle. 2017. Synchronization. Retrieved from https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html

[5] Ben Schwartz. 2014. Concurrency in Java and Go. Retrieved from http://txt.fliglio.com/2014/04/concurrency-in-java-and-go/

[6] Randal E. Bryant and David R. O'Hallaron. 2010. Computer Systems: A Programmer's Perspective (2nd Edition). Pearson Education, Boston, Chapter 12. Retrieved from http://csapp.cs.cmu.edu/2e/ch12-preview.pdf

[7] Rob Pike. 2012. Go Concurrency Patterns. Retrieved from https://talks.golang.org/2012/concurrency.slide#55

[8] Andrew Gerrand. 2010. Share Memory by Communicating. The Go Blog. Retrieved from https://blog.golang.org/share-memory-by-communicating

[9] Matthew Barlocker. 2017. Golang Pros and Cons for DevOps: Goroutines, Panics, and Errors. Retrieved from https://blog.bluematador.com/posts/golang-pros-cons-for-devops-part-1-goroutines-panics-errors/

[10] Ruben Vermeersch. 2009. Concurrency In Erlang & Scala: The Actor Model. Retrieved from https://rocketeer.be/articles/concurrency-in-erlang-scala/

[11] Brian Storti. 2015. The Actor Model in 10 Minutes. Retrieved from http://www.brianstorti.com/the-actor-model/

[12] Sander Sonajalg 2009. Actor-based Model for Concurrent Programming. Retrieved from http://ds.cs.ut.ee/courses/previous/seminar-materials/SanderS6najalg-slides.pdf

[13] Paul Mackay. Why has the actor model no succeeded?. Retrieved from https://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol2/pjm2/

[14] Lars Vogel. 2016. Java concurrent (multi-threading) - Tutorial. Retrieved from http://www.vogella.com/tutorials/JavaConcurrency/article.html#concurrency

[15] Benjamin Erb. 2012. *Concurrent Programming for Scalable Web Architectures.* Diploma Thesis. Institute of Distributed Systems, Ulm University, Baden-Württemberg, Germany.

[16] Alex Mackey. 2016. What is the actor model? Retrieved from https://gooroo.io/GoorooTHINK/Article/16608/What-is-the-actor-model/21113

[17] Texlution. 2015. Elixir concepts for Go developers. Retrieved from https://texlution.com/post/elixir-concepts-for-golang-developers/#processes