Goroutines and Go Scheduler: Comprehensive Guide

A Detailed Exploration of Concurrency in Go

Introduction to Goroutines

- Goroutines are lightweight threads managed by the Go runtime.
- They are multiplexed over a small number of OS threads.
- Created using the `go` keyword and managed by Go's scheduler.

Example:

```
package main
import (
    "fmt"
    "time"
)

func printMessage(msg string) {
    fmt.Println(msg)
}

func main() {
    go printMessage("Hello from Goroutine")
    time.Sleep(time.Second)
}
```

Unbuffered Channels

- Channels allow safe communication between goroutines.
- Unbuffered channels block the sender until the receiver is ready.

Example:

```
package main
import "fmt"

func main() {
    ch := make(chan string)

    go func() { ch <- "Hello, Channel!" }()

    msg := <-ch
    fmt.Println(msg)
}
```

Buffered Channels

- Buffered channels allow multiple values to be sent without immediate receiving.
- The capacity is defined during channel creation.

Example:

```
package main
import "fmt"

func main() {
   ch := make(chan int, 2)

   ch <- 1
   ch <- 2

   fmt.Println(<-ch)
   fmt.Println(<-ch)
}
```

Wait Groups (sync.WaitGroup)

- `sync.WaitGroup` helps synchronize multiple goroutines.
- It ensures all goroutines complete before the program exits.

Example:

```
package main
import (
    "fmt"
    "sync"
    "time"
)

func worker(id int, wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Printf("Worker %d started\n", id)
    time.Sleep(time.Second)
    fmt.Printf("Worker %d finished\n", id)
}

func main() {
    var wg sync.WaitGroup
```

Mutexes (sync.Mutex)

- `sync.Mutex` prevents race conditions when multiple goroutines access shared resources.
- Provides `Lock()` and `Unlock()` methods for safe access.

Example:

```
package main
import (
    "fmt"
    "sync"
)

var mu sync.Mutex
var counter int

func increment(wg *sync.WaitGroup) {
    defer wg.Done()
    mu.Lock()
    counter++
    mu.Unlock()
}
```

Go Scheduler & M:N Scheduling

- The Go scheduler uses an M:N model (M goroutines mapped to N OS threads).
- Uses a work-stealing algorithm for efficient scheduling.
- Key components:
  - G (Goroutine): Represents a single goroutine.
  - M (Machine): Represents an OS thread.
  - P (Processor): Handles scheduling goroutines on threads.
- `GOMAXPROCS` controls the number of OS threads available.

Example:

```
package main
import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Println("Default GOMAXPROCS:", runtime.GOMAXPROCS(0))
    runtime.GOMAXPROCS(2)
    fmt.Println("Updated GOMAXPROCS:", runtime.GOMAXPROCS(0))
}
```

sync/atomic for Lock-Free Operations

- `sync/atomic` provides lock-free atomic operations for concurrency.
- It ensures thread safety without explicit mutexes.

Example:

```
package main
import (
    "fmt"
    "sync/atomic"
)

var counter int64

func increment() {
    atomic.AddInt64(&counter, 1)
}

func main() {
    for i := 0; i < 1000; i++ {
        go increment()
    }
```

Using `select` for Channel Operations

- `select` allows waiting on multiple channel operations.
- It picks the first available channel.

Example:

```
package main
import (
    "fmt"
    "time"
)

func main() {
    ch1, ch2 := make(chan string), make(chan string)

    go func() { time.Sleep(2 * time.Second); ch1 <- "Hello" }()
    go func() { time.Sleep(1 * time.Second); ch2 <- "World" }()

    select {
    case msg := <-ch1:
        fmt.Println("Received:", msg)
    case msg := <-ch2:
```

Conclusion

- Goroutines enable efficient concurrency with low overhead.
- Channels facilitate safe communication between goroutines.
- `sync.WaitGroup` and `sync.Mutex` provide synchronization mechanisms.
- The Go scheduler efficiently schedules goroutines across OS threads using an M:N model.
- `GOMAXPROCS` allows CPU parallelism tuning.

Thank you for learning Goroutines with us! 🚀