Новый конкурс! Нарисуйте комикс или инфографику на любую тему из области информатики. Приём работ до 17 марта.

Урок Спринт 2.1

Дедлоки и Мьютексы

Дедлок в Go происходит, когда горутина находится в состоянии вечного ожидания, так как она не может продолжить выполнение своего кода. Это обычно происходит, когда горутина ожидает получения данных из канала, но никакая другая горутина не отправляет данные в этот канал. В таких случаях Go завершает выполнение программы и выводит сообщение о дедлоке.

Пример дедлока с помощью канала:

```
package main

func main() {
    ch := make(chan int)
    ch <- 1 // это приведет к дедлоку, так как нет другой горутины, которая могла бы принять эт
}
```

В этом примере горутина, запущенная функцией `main`, пытается отправить значение в канал, но нет другой горутины, которая могла бы принять это значение. Это приводит к блокировке горутины, и так как нет других горутин, которые могли бы разблокировать её, Go завершает выполнение программы и выводит сообщение о дедлоке.

```
main() started
fatal error: all goroutines are asleep - deadlock!
goroutine 1 [chan send]:
```

```
main.main()
        program.go:10 +0×fd
exit status 2
```

В Golang, мьютексы используются для обеспечения безопасности при доступе к общим данным из нескольких горутин (goroutines). Map, как и другие структуры данных, не является потокобезопасной, что означает, что одновременный доступ к Map из нескольких горутин без синхронизации может привести к состоянию гонки (race condition), что в свою очередь может привести к непредсказуемому поведению программы.

Пример без использования мьютекса, показывающий состояние гонки:

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    data := make(map[string]int)

    go func() {
        for i := 0; i < 1000; i++ {
            data["key"] = i
        }
    }()

    go func() {
        for i := 0; i < 1000; i++ {
            fmt.Println(data["key"])
        }
    }()

    time.Sleep(time.Second) // Ожидание завершения работы горутин
}
```

В этом примере две горутины пытаются читать и записывать значение "key" в Map data. Это может привести к гонке, так как одна горутина может читать значение в то время, когда другая горутина пытается его изменить, что может привести к некорректным или неожиданным значениям.

Чтобы избежать этой проблемы, можно использовать мьютекс для синхронизации доступа к Map. Пример с мьютексом:

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    data := make(map[string]int)
    var mu sync.Mutex

    go func() {
        for i := 0; i < 1000; i++ {
            mu.Lock()
            data["key"] = i
            mu.Unlock()
        }
    }()

    go func() {
        for i := 0; i < 1000; i++ {
            mu.Lock()
            fmt.Println(data["key"])
            mu.Unlock()
        }
    }()

    time.Sleep(time.Second) // Ожидание завершения работы горутин
}
```

В этом примере `sync.Mutex` используется для блокировки доступа к `Map` `data` при его чтении и записи. Когда горутина хочет получить доступ к `Map`, она вызывает `mu.Lock()`, чтобы заблокировать его для других горутин. После завершения операции с `Map`, горутина вызывает `mu.Unlock()`, чтобы разблокировать его и разрешить доступ другим горутинам.

Использование мьютекса позволяет синхронизировать доступ к общим данным, обеспечивая безопасность и предотвращая состояния гонки. Однако, при чрезмерном использовании мьютексов или неправильной синхронизации это также может привести к проблемам с производительностью или даже к блокированию программы, если мьютексы не используются правильно.