**Отчет по теме:**

**Deadlock в параллельном программировании**

**Введение**Параллельное программирование является важной частью современных вычислительных систем, позволяя эффективно использовать ресурсы многопроцессорных и многоядерных систем. Однако, при работе с параллельными процессами или потоками возникают различные проблемы, одной из которых является deadlock (взаимная блокировка). Deadlock — это ситуация, при которой два или более потоков или процессов находятся в состоянии бесконечного ожидания ресурсов, занятых друг другом, что приводит к остановке выполнения программы. В данном отчете рассматриваются причины возникновения deadlock, методы его предотвращения и обнаружения, а также примеры на языке Go (Golang)[3].

**1. Понятие deadlock**Deadlock возникает, когда несколько потоков или процессов блокируют ресурсы, которые необходимы другим потокам для продолжения работы. В результате все вовлеченные потоки оказываются в состоянии ожидания, и программа не может продолжить выполнение. Для возникновения deadlock необходимо выполнение четырех условий, известных как условия Коффмана[4]:

* Условие взаимного исключения: ресурсы не могут быть разделены между потоками, и только один поток может использовать ресурс в данный момент времени.
* Условие удержания и ожидания: поток удерживает один ресурс и ожидает освобождения другого ресурса, который занят другим потоком.
* Условие отсутствия принудительного изъятия: ресурс не может быть отобран у потока, пока он сам его не освободит.
* Условие циклического ожидания: существует замкнутый цикл потоков, каждый из которых ожидает ресурса, удерживаемого следующим потоком в цикле.

Если все четыре условия выполняются одновременно, возникает deadlock.

Deadlock может возникать не только в программном коде, но и в реальных системах, таких как базы данных, операционные системы и распределенные системы. Например, в базах данных deadlock может возникнуть при одновременной блокировке строк разными транзакциями. В операционных системах deadlock может привести к зависанию системы, что требует перезагрузки и потери данных.

**2. Пример deadlock на языке Go**Классический пример deadlock на языке Go с использованием горутин и мьютексов[5]:

package main

import (

   "fmt"

   "sync"

   "time"

)

func main() {

   var mutexA, mutexB sync.Mutex

   go func() {

      mutexA.Lock()

      fmt.Println("Горутина 1 захватила мьютекс A")

      time.Sleep(100 \* time.Millisecond) // Имитация работы

      mutexB.Lock()

      fmt.Println("Горутина 1 захватила мьютекс B")

      mutexB.Unlock()

      mutexA.Unlock()

   }()

   go func() {

      mutexB.Lock()

      fmt.Println("Горутина 2 захватила мьютекс B")

      time.Sleep(100 \* time.Millisecond) // Имитация работы

      mutexA.Lock()

      fmt.Println("Горутина 2 захватила мьютекс A")

      mutexA.Unlock()

      mutexB.Unlock()

   }()

   time.Sleep(1 \* time.Second) // Ожидание завершения горутин

   fmt.Println("Программа завершена")

}

В данном примере первая горутина захватывает мьютекс A и ожидает мьютекс B, в то время как вторая горутина захватывает мьютекс B и ожидает мьютекс A. В результате обе горутины оказываются заблокированными, и программа зависает.

Этот пример иллюстрирует типичную ситуацию deadlock, которая может возникнуть при неправильном управлении ресурсами в многопоточных приложениях.

**3. Методы предотвращения deadlock**Для предотвращения deadlock используются различные стратегии[6]:

* Устранение условия взаимного исключения: если ресурсы могут быть разделены между потоками, то deadlock не возникнет. Однако это не всегда возможно, так как некоторые ресурсы по своей природе требуют исключительного доступа. В нашем примере мьютексы по своей природе требуют исключительного доступа, поэтому этот метод не применим напрямую. Вместо использования мьютексов мы применим атомарные операции для безопасного доступа.

package main

import (

   "fmt"

   "sync/atomic"

   "time"

)

var (

   resourceA int32 = 0 // Ресурс A

   resourceB int32 = 0 // Ресурс B

)

func main() {

   go func() {

      // Горутина 1 работает с ресурсами

      for {

         // Захватываем ресурс A атомарно

         if atomic.CompareAndSwapInt32(&resourceA, 0, 1) {

            fmt.Println("Горутина 1 захватила ресурс A")

            time.Sleep(100 \* time.Millisecond) // Имитация работы

            // Захватываем ресурс B атомарно

            if atomic.CompareAndSwapInt32(&resourceB, 0, 1) {

               fmt.Println("Горутина 1 захватила ресурс B")

               time.Sleep(100 \* time.Millisecond) // Имитация работы

               // Освобождаем ресурсы

               atomic.StoreInt32(&resourceB, 0)

               atomic.StoreInt32(&resourceA, 0)

               break

            } else {

               // Если ресурс B недоступен, освобождаем ресурс A и повторяем попытку

               atomic.StoreInt32(&resourceA, 0)

               fmt.Println("Горутина 1 не смогла захватить ресурс B, освобождаю ресурс A")

               time.Sleep(100 \* time.Millisecond) // Пауза перед повторной попыткой

            }

         }

      }

   }()

   go func() {

      // Горутина 2 работает с ресурсами

      for {

         // Захватываем ресурс B атомарно

         if atomic.CompareAndSwapInt32(&resourceB, 0, 1) {

            fmt.Println("Горутина 2 захватила ресурс B")

            time.Sleep(100 \* time.Millisecond) // Имитация работы

            // Захватываем ресурс A атомарно

            if atomic.CompareAndSwapInt32(&resourceA, 0, 1) {

               fmt.Println("Горутина 2 захватила ресурс A")

               time.Sleep(100 \* time.Millisecond) // Имитация работы

               // Освобождаем ресурсы

               atomic.StoreInt32(&resourceA, 0)

               atomic.StoreInt32(&resourceB, 0)

               break

            } else {

               // Если ресурс A недоступен, освобождаем ресурс B и повторяем попытку

               atomic.StoreInt32(&resourceB, 0)

               fmt.Println("Горутина 2 не смогла захватить ресурс A, освобождаю ресурс B")

               time.Sleep(100 \* time.Millisecond) // Пауза перед повторной попыткой

            }

         }

      }

   }()

   time.Sleep(1 \* time.Second) // Ожидание завершения горутин

   fmt.Println("Программа завершена")

}

* Устранение условия удержания и ожидания: можно требовать, чтобы поток запрашивал все необходимые ресурсы сразу, прежде чем начать выполнение. Если хотя бы один ресурс недоступен, поток освобождает все ранее захваченные ресурсы и повторяет попытку позже.

package main

import (

   "fmt"

   "sync"

   "time"

)

func main() {

   var mutexA, mutexB sync.Mutex

   go func() {

      for {

         mutexA.Lock()

         fmt.Println("Горутина 1 захватила мьютекс A")

         time.Sleep(100 \* time.Millisecond) // Имитация работы

         // Попытка захватить мьютекс B с таймаутом

         if mutexB.TryLock() {

            fmt.Println("Горутина 1 захватила мьютекс B")

            mutexB.Unlock()

            mutexA.Unlock()

            break

         } else {

            fmt.Println("Горутина 1 не смогла захватить мьютекс B, освобождаю мьютекс A")

            mutexA.Unlock()

            time.Sleep(100 \* time.Millisecond) // Пауза перед повторной попыткой

         }

      }

   }()

   go func() {

      for {

         mutexB.Lock()

         fmt.Println("Горутина 2 захватила мьютекс B")

         time.Sleep(100 \* time.Millisecond) // Имитация работы

         // Попытка захватить мьютекс A с таймаутом

         if mutexA.TryLock() {

            fmt.Println("Горутина 2 захватила мьютекс A")

            mutexA.Unlock()

            mutexB.Unlock()

            break

         } else {

            fmt.Println("Горутина 2 не смогла захватить мьютекс A, освобождаю мьютекс B")

            mutexB.Unlock()

            time.Sleep(100 \* time.Millisecond) // Пауза перед повторной попыткой

         }

      }

   }()

   time.Sleep(1 \* time.Second) // Ожидание завершения горутин

   fmt.Println("Программа завершена")

}

* Устранение условия отсутствия принудительного изъятия: можно реализовать механизм принудительного освобождения ресурсов у потоков, что может быть полезно в некоторых системах, но требует сложной логики восстановления.

package main

import (

   "fmt"

   "sync"

   "time"

)

func main() {

   var mutexA, mutexB sync.Mutex

   go func() {

      for {

         mutexA.Lock()

         fmt.Println("Горутина 1 захватила мьютекс A")

         if mutexB.TryLock() {

            fmt.Println("Горутина 1 захватила мьютекс B")

            mutexB.Unlock()

            mutexA.Unlock()

            break

         } else {

            fmt.Println("Горутина 1 не смогла захватить мьютекс B, освобождаю мьютекс A")

            mutexA.Unlock()

            time.Sleep(100 \* time.Millisecond) // Пауза перед повторной попыткой

         }

      }

   }()

   go func() {

      for {

         mutexB.Lock()

         fmt.Println("Горутина 2 захватила мьютекс B")

         if mutexA.TryLock() {

            fmt.Println("Горутина 2 захватила мьютекс A")

            mutexA.Unlock()

            mutexB.Unlock()

            break

         } else {

            fmt.Println("Горутина 2 не смогла захватить мьютекс A, освобождаю мьютекс B")

            mutexB.Unlock()

            time.Sleep(100 \* time.Millisecond) // Пауза перед повторной попыткой

         }

      }

   }()

   time.Sleep(1 \* time.Second) // Ожидание завершения горутин

   fmt.Println("Программа завершена")

}

* Устранение условия циклического ожидания: можно ввести порядок захвата ресурсов, чтобы потоки всегда запрашивали ресурсы в определенной последовательности. Например, если все потоки запрашивают ресурс A перед ресурсом B, то циклическое ожидание невозможно.

package main

import (

   "fmt"

   "sync"

   "time"

)

func main() {

   var mutexA, mutexB sync.Mutex

   go func() {

      mutexA.Lock()

      fmt.Println("Горутина 1 захватила мьютекс A")

      time.Sleep(100 \* time.Millisecond) // Имитация работы

      mutexB.Lock()

      fmt.Println("Горутина 1 захватила мьютекс B")

      mutexB.Unlock()

      mutexA.Unlock()

   }()

   go func() {

      mutexA.Lock() // Теперь обе горутины захватывают мьютекс A первым

      fmt.Println("Горутина 2 захватила мьютекс A")

      time.Sleep(100 \* time.Millisecond) // Имитация работы

      mutexB.Lock()

      fmt.Println("Горутина 2 захватила мьютекс B")

      mutexB.Unlock()

      mutexA.Unlock()

   }()

   time.Sleep(1 \* time.Second) // Ожидание завершения горутин

   fmt.Println("Программа завершена")

}

**4. Методы обнаружения deadlock**Если предотвращение deadlock невозможно, можно использовать методы его обнаружения:

* Граф ожидания: система строит граф, где узлы представляют потоки и ресурсы, а ребра — запросы и удержания ресурсов. Если в графе обнаруживается цикл, это указывает на наличие deadlock.
* Алгоритм банкира: этот алгоритм моделирует распределение ресурсов и проверяет, может ли система безопасно завершить выполнение всех потоков без deadlock.
* Таймауты: если поток ожидает ресурс слишком долго, система может предположить наличие deadlock и предпринять действия для его устранения, например, завершить один из потоков.

**5. Практические примеры и последствия deadlock**

Для более наглядного примера возьмем реальную практическую задачу с вычислением чисел фибоначчи рекурсивным методом. Программа будет использовать кеш для хранения уже вычисленных значений и пытаться работать с кешем из нескольких горутин параллельно:

package main

import (

"fmt"

"sync"

)

// Кэш для хранения вычисленных чисел Фибоначчи

var cache = struct {

sync.Mutex

values map[int]int

}{

values: make(map[int]int),

func (fc \*FibonacciCalculator) FibonacciWithDeadlock(n int) int {

fc.deadlockCache.Lock()

defer fc.deadlockCache.Unlock()

if val, found := fc.deadlockCache.values[n]; found {

return val

}

if n <= 1 {

return n

}

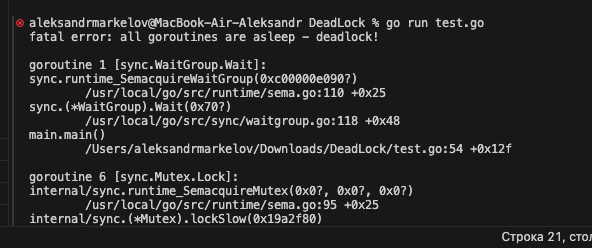
val := fc.FibonacciWithDeadlock(n-1) + fc.FibonacciWithDeadlock(n-2)

fc.deadlockCache.values[n] = val

return val

}

Горутина блокирует мьютекс для Fib(n). Рекурсивно вызывает Fib(n-1) и Fib(n-2), пытаясь повторно заблокировать тот же мьютекс. Результат: программа зависает, так как мьютекс нельзя захватить дважды в одной горутине.:



Варианты решения:

##### **Двойная проверка**. Первая проверка кэша — с блокировкой. Рекурсивные вычисления — без блокировки. Сохранение результата — с повторной блокировкой:

func (fc \*FibonacciCalculator) FibonacciWithDoubleCheck(n int) int {

fc.doubleCheckCache.Lock()

if val, found := fc.doubleCheckCache.values[n]; found {

fc.doubleCheckCache.Unlock()

return val

}

fc.doubleCheckCache.Unlock()

if n <= 1 {

return n

}

val := fc.FibonacciWithDoubleCheck(n-1) + fc.FibonacciWithDoubleCheck(n-2)

fc.doubleCheckCache.Lock()

fc.doubleCheckCache.values[n] = val

fc.doubleCheckCache.Unlock()

return val

}

### **Использование sync.Map.** Встроенная потокобезопасность. Не требует явных блокировок. Высокая производительность при частых чтениях:

func (fc \*FibonacciCalculator) FibonacciWithSyncMap(n int) int {

if val, ok := fc.syncMapCache.Load(n); ok {

return val.(int)

}

if n <= 1 {

return n

}

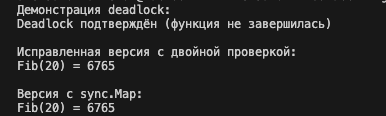
val := fc.FibonacciWithSyncMap(n-1) + fc.FibonacciWithSyncMap(n-2)

fc.syncMapCache.Store(n, val)

return val

}

При этих решениях deadlock не возникает и программа корректно обрабатывает результат:



Deadlock может возникать в различных системах, включая операционные системы, базы данных и распределенные системы. Например, в базах данных deadlock может возникнуть при одновременной блокировке строк разными транзакциями. В операционных системах deadlock может привести к зависанию системы, что требует перезагрузки и потери данных.

Для предотвращения deadlock в базах данных используются механизмы обнаружения и разрешения конфликтов, такие как откат одной из транзакций. В операционных системах применяются алгоритмы планирования, которые минимизируют вероятность возникновения deadlock.

Рассмотрим также еще один пример программы с deadlock и его исправлением. Программа моделирует банковские переводы между счетами (имитация работы транзакций в БД).

func main() {

   rand.Seed(time.Now().UnixNano())

   const numAccounts = 100

   const numTransactions = 10000

   fmt.Println("=== ДЕМОНСТРАЦИЯ DEADLOCK С 10 000 ГОРУТИН ===")

   //с deadlock

   fmt.Println("Запуск 10 000 небезопасных переводов...")

   bankDeadlock := Bank{}

   for i := 0; i < numAccounts; i++ {

      bankDeadlock.Accounts = append(bankDeadlock.Accounts, &Account{ID: i, Balance: 1000})

   }

   runTransactions(&bankDeadlock, numTransactions, true)

   //без deadlock

   fmt.Println("Запуск 10 000 безопасных переводов...")

   bankCorrect := Bank{}

   for i := 0; i < numAccounts; i++ {

      bankCorrect.Accounts = append(bankCorrect.Accounts, &Account{ID: i, Balance: 1000})

   }

   runTransactions(&bankCorrect, numTransactions, false)

   fmt.Println("Программа завершена!")

}

В данном случае программа сначала выполняет транзакции с помощью метода, который имеет вероятность получить deadlock(для того чтобы горутины точно успели заблокировать друг друга выставлена задержка)

func (b \*Bank) TransferDeadlock(fromID, toID, amount int) {

   from := b.Accounts[fromID]

   to := b.Accounts[toID]

   from.mu.Lock()

   // Искусственная задержка для увеличения вероятности deadlock

   time.Sleep(time.Microsecond \* 100)

   to.mu.Lock()

   from.Balance -= amount

   to.Balance += amount

   to.mu.Unlock()

   from.mu.Unlock()

}

При вызове данного метода программа будет последовательно ментяь счета from и to, что в конечном счете может рано или поздно привести к возникновению deadlock (Если одновременно горутина 1 блокирует А, затем пытается заблокировать Б, а горутина 2 блокирует Б и пытается заблокировать А). Все условия Коффмана выполняются, и поэтому мы можем лицезреть заветный deadlock в тесте этого метода:



Далее программа запускает вторую имитацию транзакций через метод, который не допускает образования deadlock

func (b \*Bank) TransferCorrect(fromID, toID, amount int) {

   first, second := fromID, toID

   if fromID > toID {

      first, second = second, first

   }

   b.Accounts[first].mu.Lock()

   b.Accounts[second].mu.Lock()

   b.Accounts[fromID].Balance -= amount

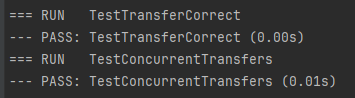
   b.Accounts[toID].Balance += amount

   b.Accounts[second].mu.Unlock()

   b.Accounts[first].mu.Unlock()

}

В этом методе используется универсальный порядок блокировки: всегда сначала блокируется счет с меньшим ID. Это гарантирует отсутствие циклических зависимостей. Также порядок блокировки не зависит от направления перевода и все горутины используют одинаковый алгоритм. Из-за того что мы исключили циклические зависимости и сделали поведение программы детерминированным (результат не зависит от случайных факторов) deadlock перестал возникать при работе программы, что наглядно отображено в тестах:



Так как в методе нарушается 4-е условие Коффмана (циклическое ожидание) deadlock перестает возникать.

**Заключение**Deadlock является серьезной проблемой в параллельном программировании, которая может привести к остановке выполнения программы и потере данных. Для предотвращения deadlock необходимо понимать его причины и использовать соответствующие стратегии, такие как упорядоченное захватывание ресурсов или алгоритмы обнаружения. В реальных системах важно проектировать программы с учетом возможных deadlock и тестировать их на наличие подобных проблем.

**Список источников**

1. The Go Programming Language Specification  
   Mutual Exclusion  
   Go Authors. (2023). The Go Programming Language Specification.  
   URL: <https://go.dev/ref/spec#Mutual_exclusion>(Официальная документация о мьютексах в Go)
2. Cox-Buday, K. (2017)  
   Concurrency in Go: Tools and Techniques for Developers  
   O'Reilly Media. Chapter 4: "Deadlocks, Starvation, and Livelocks".  
   ISBN: 978-1-491-94194-3  
   (Практическое руководство по конкурентности в Go)
3. Effective Go: Concurrency  
   Go Documentation Team. (2023).  
   URL: <https://go.dev/doc/effective_go#concurrency>(Рекомендации от создателей языка)
4. Coffman, E.G. Jr., Elphick, M.J., Shoshani, A. (1971)  
   System Deadlocks  
   ACM Computing Surveys, 3(2), pp. 67-78.  
   DOI: 10.1145/356586.356588  
   (Фундаментальная работа об условиях возникновения deadlock)
5. Go by Example: Mutexes  
   (2023) Practical examples of mutex usage.  
   URL: <https://gobyexample.com/mutexes>
6. Ousterhout, J. (2014)  
   Deadlock Prevention in Distributed Systems  
   Stanford University CS140 Lecture Notes.  
   URL: <https://web.stanford.edu/~ouster/cgi-bin/cs140-spring14/lecture.php?topic=deadlock>
7. Silberschatz, A., Galvin, P.B., Gagne, G. (2018)  
   Operating System Concepts (10th ed.)  
   Wiley. Chapter 7: "Deadlocks".  
   ISBN: 978-1-119-32091-3