# ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ: MUTEX/RWMUTEX

*Выполнили студенты М2-ИФСТ-11:  
Никитин Ю.А., Сутягин Д.Д.*

Решение некоторых задач может стать более выгодным по времени, если использовать **конкурентность**. Под конкурентностью в теории вычислительных машин понимается разбиение одного процесса на несколько независимых составляющих и определение для них безопасного способа совместного использования данных [1, стр. 251]. Функция, использующая данные совместно безопасным образом, – это такая функция, которая продолжает корректно работать даже при параллельном вызове. Однако в реальности многие функции без дополнительной синхронизации не могут правильно работать при параллельном запуске или при конкурентном доступе к каким-либо общим ресурсам [2, раздел 9.1, стр. 303-304].

Одной из самых распространённых причин некорректной работы функций при параллельном вызове является **состояние гонки (состязательная ситуация)**. Данная проблема проявляется в том, что программа даёт неверный результат при чередовании операций нескольких подпрограмм [2, раздел 9.1, стр. 304]. Результат работы при состоянии гонки неопределён. Так например, если две подпрограммы, запущенные параллельно, попытаются получить доступ к одной переменной, чтобы обновить её значение, то точно сказать, каким в итоге будет новое значение - невозможно. Единственная параллельная вещь, которую можно безопасно делать с переменной – читать её значение, а запись нужно синхронизировать [3, раздел. 6].

С ростом параллелизма частота возникновения состязательных ситуаций возрастает. При этом состояния гонки очень вредны, так как могут оставаться скрытыми, проявляться крайне редко или только при некоторых условиях, например большой нагрузке, использовании некоторых компиляторов, на определённых платформах или для определённой архитектуры. То есть в реальности состояния гонки сложны для воспроизведения и диагностики [2, раздел 9.1, стр. 304; 4, раздел 2.3.1., стр. 148]. Ключевой подход к предупреждению возникновения данной проблемы заключается в определении такого способа использования общей памяти, при котором в каждый конкретный момент времени доступ к разделяемому ресурсу может получить только один процесс. Такой способ реализуется с помощью механизма **взаимных исключений**, существующего во многих языках программирования, включая Go. Этот механизм позволяет сохранить параллельный доступ к общему ресурсу, но при этом не допускает возникновение гонки [2, раздел 9.1, стр. 309; 4, раздел 2.3.2., стр. 148].

**Mutex** (Mutual Exclusion — взаимное исключение) – это шаблон взаимного исключения, накладывающий ограничение на конкурентное выполнение определённого кода или на доступ к совместно используемым данным. Та часть программы, в которой используется доступ к общей памяти, называется **критической секцией (критической областью)** [1, стр. 280; 4, раздел 2.3.6., стр. 161]. Мьютексы позволяют гарантировать, что только один поток (горутина) имеет доступ к общему ресурсу в определённый момент времени, и пока этот поток (горутина) не освободит общий ресурс, другой не сможет с ним работать [5]. С архитектурной точки зрения мьютексы являются упрощённой версией семафоров и представляют собой совместно используемую переменную (объект), которая может находиться в одном из двух состояний: заблокированном или не заблокированном [4, раздел 2.3.6., стр. 161].

В стандартном пакете sync языка go представлено две реализации мьютекса: sync.Mutex и sync.RWMutex.

Реализация Mutex предоставляет два метода Lock и Unlock. Вызов метода Lock приостанавливает выполнение текущей горутины, если критическая секция в данный момент занята другой горутиной. Если же критическая секция свободна, то текущая горутина устанавливает блокировку и выполняет код в критической секции. Вызов метода Unlock в экземпляре типа Mutex завешает использование критической секции [1, стр. 281].

Приведём пример использования sync.Mutex и покажем, как он решает проблему гонки. Во фрагменте кода ниже представлен параллельный запуск функции инкремента переменной в 1000 горутинах. Результат выполнения будет не определён: значение переменной может как оказаться равным ожидаемому значению - 1000, так и быть меньше него.

package main

import (

"fmt"

"sync"

)

var counter = 0

var wg sync.WaitGroup

func main() {

wg.Add(1000)

for i := 0; i < 1000; i++ {

go increment(&wg)

}

wg.Wait()

fmt.Println(counter)

}

func increment(wg \*sync.WaitGroup) {

counter++

wg.Done()

}

Для синхронизации работы горутин модифицируем написанную программу так, как показано ниже. После добавления мьютекса результат становится и предсказуемым, и верным.

package main

import (

"fmt"

"sync"

)

var counter = 0

var wg sync.WaitGroup

var mutex sync.Mutex

func main() {

wg.Add(1000)

for i := 0; i < 1000; i++ {

go increment(&wg)

}

wg.Wait()

fmt.Println(counter)

}

func increment(wg \*sync.WaitGroup) {

defer wg.Done()

mutex.Lock()

counter++

mutex.Unlock()

}

**RWMutex** является второй реализацией мьютекса, расширяющей его базовую функциональность. RWMutex позволяет устанавливать блокировки на чтение и запись раздельно, что делает возможным конкурентный доступ к общему ресурсу для нескольких читателей. Это означает, что RWMurtex может быть заблокирован либо одной блокировкой записи, либо одной или несколькими блокировками чтения. Когда читатель блокирует мьютекс, другие читатели смогут совместно использовать эту блокировку, что позволяет операциям чтения выполняться одновременно, без ожидания своей эксклюзивной блокировки. [6, стр. 339; 7, стр. 503]. Такой механизм может быть использован в качестве оптимизации в тех случаях, когда количество операций чтения значительно превосходит количество операций записи. Использование обычного Mutex в такой ситуации вызовет заметное снижение производительности, так как каждая горутина будет ждать исключительную блокировку, хотя в большинстве случаев в исключительной блокировке не будет необходимости. RWMutex в этой ситуации обеспечит лучшую производительность за счёт сохранения параллельного доступа для преобладающих операций чтения [8,9].

Метод RLock служит для получения блокировки чтения. Несколько горутин могут удерживать блокировку чтения одновременно, пока не удерживается блокировка записи. При том блокировка записи, накладываемая методом Lock, может быть получена лишь тогда, когда нет удерживаемых блокировок чтения. Блокировки Runlock и Unlock снимают блокировки чтения и записи соответственно [10].

Ниже приведены два фрагмента кода. В них выполняются функции записи и чтения. В первом случае используется обычный Mutex.

package main

import (

"fmt"

"sync"

"time"

)

var counter = 0

var wg sync.WaitGroup

var mutex sync.Mutex

func main() {

start := time.Now()

wg.Add(100)

for i := 0; i < 50; i++ {

go read(&wg)

go write(&wg)

}

wg.Wait()

fmt.Println(time.Now().Sub(start).Seconds())

}

func read(wg \*sync.WaitGroup) {

defer wg.Done()

mutex.Lock()

time.Sleep(time.Millisecond)

\_ = counter

mutex.Unlock()

}

func write(wg \*sync.WaitGroup) {

defer wg.Done()

mutex.Lock()

time.Sleep(time.Millisecond)

counter++

mutex.Unlock()

}

Во втором случае используется RWMutex и соответственно блокировка RLock в операции чтения. Если запустить обе программы и замерить время их выполнения, то можно убедиться, что вариант с RWMutex работает существенно быстрее: в среднем 0.12 с для Mutex и в среднем 0.6 с для RWMutex.

package main

import (

"fmt"

"sync"

"time"

)

var counter = 0

var wg sync.WaitGroup

var rwmutex sync.RWMutex

func main() {

start := time.Now()

wg.Add(100)

for i := 0; i < 50; i++ {

go read(&wg)

go write(&wg)

}

wg.Wait()

fmt.Println(time.Now().Sub(start).Seconds())

}

func read(wg \*sync.WaitGroup) {

defer wg.Done()

rwmutex.RLock()

time.Sleep(time.Millisecond)

\_ = counter

rwmutex.RUnlock()

}

func write(wg \*sync.WaitGroup) {

defer wg.Done()

rwmutex.Lock()

time.Sleep(time.Millisecond)

counter++

rwmutex.Unlock()

}

Как говорилось ранее, использование RWMutex обычно выгодно тогда, когда количество операций чтения превосходит количество операций записи, в иных случаях производительность RWMutex может оказаться равной или даже меньшей обычному Mutex. При наличии такого желания более детально ознакомиться с замерами производительности двух реализаций мьютекса можно в статье «Go Lock Performance: RwMutex vs Mutex in Various Scenarios»[11].

Стоит помнить о том, что использование RWMutex может создавать ситуацию **голодания чтения (write starvation)**. Если есть большое число блокировок чтения и несколько блокировок записи, то мьютекс будет принимать входящие дополнительные блокировки чтения после первой, вынуждая операции записи длительно ожидать пока все блокировки чтения не будут сняты. В случае использования обычного Mutex такая проблема не возникает, так как используется один тип блокировки, что означает равный приоритет операций чтения и записи [7, стр. 505-506].

Отдельно стоит упомянуть про низкоуровневый примитив синхронизации – **фьютекс (futex)** (сокращение от fast userspace mutex). На его основе реализуются другие примитивы синхронизации и механизмы: мьютексы, семафоры, условные переменные. Смысл существования фьютексов – предоставление более эффективного способа синхронизации пользовательских потоков за счёт уменьшения числа обращений к ядру ОС. Фьютекс реализует блокировку таким образом, чтобы избегать дорого обходящегося переключения в режим ядра без возникновения в этом реальной необходимости[12; 4, раздел 2.3.6., стр. 163].

До появления фьютексов для управления многопоточным доступом к общим ресурсам приходилось делать системные вызовы, что было достаточно ресурсозатратно из-за переключения контекста из пользовательского режима в режим ядра. Более того, в большинстве случаев попытка захвата заканчивается успешно, то есть не требует системного вызова. И только в менее вероятном случае, когда запрашиваемый ресурс заблокирован другим потоком, можно попросить ядро перевести текущий поток в спящий режим. В данном случае применение системного вызова может быть оправдано с точки зрения производительности, так как поток всё равно был бы заблокирован. Если же конкуренция отсутствует, ядро ОС вовлекаться в работу вообще не будет. То есть при использовании фьютексов основные операции синхронизации выполняются в пространстве пользователя, а системный вызов futex() выполняется лишь тогда, когда нужно чтобы программа вошла в режим ожидания на долгий срок. При этом использование фьютексов является лучшей альтернативой организуемым в пользовательском режиме спин-локам, так как не тратит процессорные инструкции в процессе ожидания [12; 4, раздел 2.3.6., стр. 164].

Кроме того существует такой примитив синхронизации как **timed mutex (временной мьютекс)**. Он расширяет возможности мьютекса за счёт дополнительны операций try\_lock\_for и try\_lock\_until, предоставляющих возможность попытаться захватить мьютекс с некоторым тайм-аутом. Обе функции основаны на редко используемой, но стандартной операции мьютексов try\_lock (имеется и в Go). Функция try\_lock пытается захватить мьютекс. Если мьютекс ещё не захвачен ни одним потоком, то поток, вызвавший try\_lock, становится его обладателем, а метод возвращает true, в ином случае возвращается false, при этом текущий поток не блокируется. Функция try\_lock\_for возвращает true, если удалось захватить мьютекс в течение заданного промежутка времени, иначе возвращает false. Функция try\_lock\_until возвращает true, если удалось захватить мьютекс до достижения заданной точки времени, иначе возвращает false[13,14].

Сверх того есть ещё один вид мьютекса – **recursive\_mutex (рекурсивный мьютекс)**, который соответственно своему названию позволяет рекурсивный захват мьютекса, то есть многократный вызов метода lock в потоке, который владеет мьютексом. При этом метод unlock должен быть вызван столько же раз, сколько раз был вызван lock, иначе возникнет состояние deadlock, при котором поток никогда не освободит мьютекс, а остальные потоки будут находиться в вечном ожидании. То есть рекурсивный мьютекс подсчитывает сколько раз он был заблокирован и требует выполнения того же числа операции разблокировки, прежде чем его смогут захватить другие потоки. Такой механизм может оказаться необходим при использовании рекурсивных функций, обращающихся к некоторому разделяемому ресурсу [13,15].

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Боднер Д. - Go. Идиомы и паттерны проектирования // Питер, 2022. — 416 с.
2. Алан А. А. Донован, Брайан У. Керниган.Язык программирования Go // ООО “И.Д. Вильямс”, 2016. — 432 с.
3. The Little Go Book [Электронный ресурс] URL: https://github.com/sefus/the-little-go-book/blob/master/ru/go.md.
4. Таненбаум, Бос. Современные операционные системы 4-е издание // Питер, 2015. — 1120 с
5. Мьютексы [Электронный ресурс] URL: https://metanit.com/go/tutorial/7.6.php.
6. Mihalis Tsoukalos. Go Systems Programming // Packt Publishing, 2017 – 459 с.
7. Alex Guerrieri. Hands-On Sysytem Programming with Go // Packt Publishing, 2019 – 727 с.
8. Sync.RWMutex [Электронный ресурс] URL: https://medium.com/golangspec/sync-rwmutex-ca6c6c3208a0.
9. Конкурентность в Go простым языком [Электронный ресурс] URL: https://blog.ildarkarymov.ru/posts/go-concurrency/.
10. ByteGoblin.io [Электронный ресурс] URL: https://bytegoblin.io/blog/understanding-mutex-vs-rwmutex-in-go.
11. Go Lock Performance: RwMutex vs Mutex in Various Scenarios [Электронный ресурс] URL: https://dev.to/leapcell/go-lock-performance-rwmutex-vs-mutex-in-various-scenarios-57p7.
12. Основы работы с фьютексами [Электронный ресурс] URL: https://habr.com/ru/companies/infopulse/articles/418705/.
13. Добро пожаловать в параллельный мир. Часть 1: Мир многопоточный [Электронный ресурс] URL: https://scrutator.me/post/2012/04/04/parallel-world-p1.aspx.
14. Concurrency support library.std::timed\_mutex [Электронный ресурс] URL: https://en.cppreference.com/w/cpp/thread/timed\_mutex.
15. Рекурсивный мьютекс.Многопоточное программирование [Электронный ресурс] URL: https://youtu.be/\_jZlmgDoDOM?si=igunVtsfU1Zl5pmg.