Министерство образования и науки Российской Федерации

**Саратовский государственный технический университет**

**имени Гагарина Ю.А.**

**Data Race**

**Практическая работа**

по курсу «Параллельное программирование»

для студентов направления

«Информационные системы и технологии»

Выполнили:

студенты м2-ифст-11

Решетников М.С.

Алексеев В.В.  
Проверил:

старший преподаватель кафедры ПИТ

Акутин А.С.

Саратов 2025

1. **Введение**

**Состояние гонки (data race)** возникает в многопоточной программе, когда несколько потоков одновременно обращаются к одному и тому же разделяемому участку памяти, где хотя бы одно обращение – запись [1].

Обращения могут выполняться в любом порядке, что приводит к непредсказуемому состоянию программы. Интуитивно понятно, что при одновременном чтении с записью может быть возвращено старое локальное или недавно записанное значение; конечным значением, сохраняемым в местоположении при двух записях, может быть любое из вновь записанных значений. Скачки данных, как известно, приводят к возникновению проблемных и запутанных ошибок. Незаметное, непредсказуемое разрешение скачков данных ухудшает воспроизводимость при тестировании и отладке и может приводить к другим, более сложным ошибкам, связанным с общей памятью, таким как нарушения атомарности или детерминизма [2].

Состояния гонки слабо локализованы и их сложно обнаружить вручную, поэтому исследования в области автоматического поиска гонок ведутся уже более 20 лет[1].

Дополнительная трудность с обнаружением гонок состоит в том, что они, как правило, не приводят к немедленному сбою и отказу программы. Напротив, приложение продолжает работать с повреждёнными глобальными структурами данных, что приводит к труднообъяснимым эффектам впоследствии [1].

Более того, в то время как модели согласованности данных в памяти современных основных языков программирования гарантируют, что программы, свободные от скачков данных, выполняются с интуитивно понятной семантикой последовательной согласованности, они не дают никаких гарантий относительно выполнения программ с скачками данных. Оптимизации в современных компиляторах и многопроцессорных архитектурах могут взаимодействовать с скачками данных, что приводит к не интуитивному выполнению программ, когда чередования операций в нескольких потоках по порядку недостаточно для того, чтобы оценить все возможные варианты выполнения [2].

1. **Определение гонки данных в Go (data race)**

Гонка данных – это состояние, при котором несколько потоков исполнения (в случае Go – горутин) одновременно обращаются к одному и тому же участку памяти без должной синхронизации, причём хотя бы одно из этих обращений является записью. Если все одновременные обращения лишь читают общие данные, гонка не возникает [3]. Концепция гонки данных относится к классу состояний гонки (race conditions), означающих, что результат работы программы зависит от непредсказуемого чередования операций нескольких потоков.

Причиной гонки данных является неправильное проектирование многопоточных участков программы, приводящее к конфликтующим изменениям общих данных. Современные аппаратные платформы обеспечивают независимость выполнения потоков, что создает условия для непредсказуемого результата. Следовательно, ошибка появляется только тогда, когда конкретный порядок выполнения нарушает целостность состояния программы.

Важно отметить, что сложность обнаружения гонок данных связана с их вероятностным характером проявления. Ошибка может возникать нерегулярно, что затрудняет повторяемость наблюдаемого сбоя и выявление истинной причины. Как следствие, устранение таких дефектов требует тщательной проверки логики многопоточного взаимодействия и глубокого понимания правил конкуренции в программе.

Вероятность возникновения гонок данных возрастает с увеличением количества потоков и частотой изменений общего ресурса. Это ставит дополнительные требования к архитектуре программы и повышает важность грамотного проектирования конкурентных секций.

**3.Теоретическая природа и поведение гонок**

Гонки данных принадлежат к числу самых сложных и непредсказуемых ошибок многопоточного программирования [4]. При наличии гонки порядок выполнения конфликтующих операций не определён, что ведёт к минимальным гарантиям на поведение программы. Как отмечается в литературе, компиляторы и аппаратное обеспечение в таких условиях могут переупорядочивать выполнение инструкций ради оптимизации, из-за чего во время гонки данных «вы можете увидеть почти всё что угодно» [5].

Проще говоря, программа с гонками данных теряет свойства последовательной согласованности: она может давать неожиданные результаты, сбои или нарушать целостность данных. Например, функция с гонкой может вывести неправильное значение (вместо задуманного результата «2» напечатать «1») вследствие неопределённости выполнения [5]. Официальная документация Go прямо предупреждает, что гонки данных способны приводить к сбоям (panic) и повреждению памяти [6].

С теоретической точки зрения, поведение программ без гонок данных значительно более предсказуемо. Согласно памятной модели Go (определяющей правила видимости памяти при Concurrency), если программа не содержит гонок, то она эквивалентна некоторому последовательному выполнению всех горутин – свойство, известное как DRF-SC (data race free – sequential consistency) [7]. Это означает, что при отсутствии гонок параллельные операции имеют строгий порядок, как если бы выполнялись на одном процессоре, что упрощает рассуждения о корректности.

Однако при наличии хотя бы одной гонки данных такие гарантии теряются. Строгих ограничений на поведение уже нет: язык Go рассматривает гонки как ошибку, и выполнение программы с гонкой может иметь неоднозначные результаты.

В отличие от языков вроде C/C++, где любая гонка делает поведение полностью неопределённым, и компилятор волен совершать произвольные преобразования, модель памяти Go накладывает некоторый минимум ограничений на программу с гонкой. В частности, гарантируется, что каждое чтение видит некоторое действительно записанное значение (а не произвольный «невозможный» результат), и реализация Go может сразу завершить программу, обнаружив гонку. Такой подход делает «испорченные» гонкой программы чуть более предсказуемыми (большинство гонок приводят лишь к ограниченному числу возможных исходов), но гонка данных остаётся логической ошибкой, против которой включены диагностические инструменты [7]. Эксперты подчёркивают, что из-за контринтуитивных эффектов гонок разработчикам настоятельно рекомендуется их выявлять и устранять [8].

**4. Проявления гонок данных в Go и особенности языка**

Язык Go изначально спроектирован для удобной конкурентности: в нём есть лёгкие потоки (горутины) и средства синхронизации через обмен сообщениями (каналы), а также возможность разделять память между горутинами [9]. Однако даже продуманный механизм конкурентности не устраняет саму природу гонок – без грамотной синхронизации общие данные остаются уязвимыми для одновременного доступа [4].

Go-разработчики широко используют параллелизм, и исследования показывают, что сочетание многочисленных горутин с некоторыми идиоматическими приёмами языка приводит к высокой подверженности программ на Go гонкам данных. Так, в работе PLDI   2022 года отмечено, что «обилие конкурентности наряду с идиомами и нюансами языка делает программы на Go крайне склонными к гонкам данных». В крупномасштабном коде эти проблемы проявляются на практике: при анализе 46 миллионов строк Go-кода в компании Uber было обнаружено свыше 2000 случаев гонок данных за полгода, требующих исправления [9]. Такой опыт подчёркивает, что гонки данных – не абстрактная угроза, а распространённая проблема даже в индустриальных Go-проектах.

Гонки данных в Go обычно возникают, когда разработчики прямо или косвенно разделяют изменяемое состояние между горутинами. Примером может служить ситуация, когда две горутины обновляют общую переменную без какой-либо синхронизации – итоговое значение зависит от несовместимых по порядку операций и может оказаться неправильным. В официальном примере из документации Go две горутины одновременно записывают в общую структуру (словарь map), что приводит к конфликтующим доступам [6].

func main() {

c := make(chan bool)

m := make(map[string]string)

go func() {

m["1"] = "a" // First conflicting access.

c <- true

}()

m["2"] = "b" // Second conflicting access.

<-c

for k, v := range m {

fmt.Println(k, v)

}

}

Без синхронизации подобные конфликты чреваты некорректным состоянием памяти и даже аварийным завершением программы. Go придерживается принципа: «не общайтесь, деля память; делитесь памятью, общаясь», призывая отправлять данные через каналы вместо конкурентного доступа к разделяемым переменным [5]. Тем не менее, если этот принцип нарушается и горутины одновременно обращаются к одним данным, возникает гонка. Статическая и динамическая проверка таких условий – важная часть обеспечения корректности программ на Go.

**5. Типичные примеры ошибок, при которых возникает гонка данных**

Вот некоторые типичные данные гонки. Все они могут быть обнаружены с помощью детектора гонки [10].

**1.Гонка на счетчике цикла**

func main() {

var wg sync.WaitGroup

wg.Add(5)

for i := 0; i < 5; i++ {

go func() {

// Не та 'i' которую вы ожидаете.

fmt.Println(i)

wg.Done()

}()

}

wg.Wait()

}

Здесь запускается пять горутин внутри цикла for. Проблема возникает потому, что каждая горутина захватывает значение переменной i, которое изменяется во внешнем цикле. Поскольку цикл быстро увеличивается до значения 5, большинство запущенных горутин получают последнее значение (i=5), а не значения от 0 до 4, которые ожидались бы. Это приводит к тому, что вывод программы становится непредсказуемым — вместо чисел от 0 до 4 выводятся числа типа 55555. Программу можно исправить, сделав копию переменной:

func main() {

var wg sync.WaitGroup

wg.Add(5)

for i := 0; i < 5; i++ {

go func(j int) {

// Считывает локальную копию счетчика цикла.

fmt.Println(j)

wg.Done()

}(i)

}

wg.Wait()

}

**2. Случайно разделяемая переменная**

// ParallelWrite записывает данные в file1 и file2,

// возвращает ошибки.

func ParallelWrite(data []byte) chan error {

res := make(chan error, 2)

f1, err := os.Create("file1")

if err != nil {

res <- err

} else {

go func() {

// Эта err разделяемая с main goroutine,

// поэтому выполнение записи вызывает

// гонку с выполнением записи ниже.

\_, err = f1.Write(data)

res <- err

f1.Close()

}()

}

// Вторая конфликтующая запись в err.

f2, err := os.Create("file2")

if err != nil {

res <- err

} else {

go func() {

\_, err = f2.Write(data)

res <- err

f2.Close()

}()

}

return res

}

В данной реализации две разные горутины используют общую переменную err для хранения ошибок, возникающих при выполнении операций ввода-вывода. Это создаёт условия для гонки данных, поскольку обе горутины могут пытаться записать в одну и ту же переменную одновременно.

Исправление заключается в том, что каждая горутина должна иметь свою собственную переменную для сохранения результата операции. Для этого достаточно объявлять локальные переменные прямо перед операциями чтения-записи:

...

\_, err := f1.Write(data)

...

\_, err := f2.Write(data)

...

**3.Незащищенная глобальная переменная**

var service map[string]net.Addr

func RegisterService(name string, addr net.Addr) {

service[name] = addr

}

func LookupService(name string) net.Addr {

return service[name]

}

Глобальная service map доступен сразу нескольким горутинам без какой-либо защиты от одновременного доступа. Операции добавления элемента (RegisterService) и извлечения элемента (LookupService) происходят параллельно, что может вызвать конфликты доступа (гонку данных). Чтобы сделать код безопасным, необходимо защитить доступ с помощью механизма синхронизации мьютекса  (sync.Mutex):

var (

service map[string]net.Addr

serviceMu sync.Mutex

)

func RegisterService(name string, addr net.Addr) {

serviceMu.Lock()

defer serviceMu.Unlock()

service[name] = addr

}

func LookupService(name string) net.Addr {

serviceMu.Lock()

defer serviceMu.Unlock()

return service[name]

}

**4.Примитивная незащищенная переменная**

Гонки данных могут происходить и с переменными примитивных типов (bool, int, int64 и т. д.), как в этом примере [10]:

type Watchdog struct{ last int64 }

func (w \*Watchdog) KeepAlive() {

// **Первый конфликтующий доступ**.

w.last = time.Now().UnixNano()

}

func (w \*Watchdog) Start() {

go func() {

for {

time.Sleep(time.Second)

// **Второй конфликтующий доступ**.

if w.last < time.Now().Add(-10\*time.Second).UnixNano() {

fmt.Println("No keepalives for 10 seconds. Dying.")

os.Exit(1)

}

}

}()

}

Даже такие "невинные" гонки данных могут привести к трудным для отладки проблемам, вызванным неатомарностью доступа к памяти, вмешательством в оптимизацию компилятора или проблемами переупорядочения доступа к памяти процессора [10].

type Watchdog struct{ last int64 }

func (w \*Watchdog) KeepAlive() {

atomic.StoreInt64(&w.last, time.Now().UnixNano())

}

func (w \*Watchdog) Start() {

go func() {

for {

time.Sleep(time.Second)

if atomic.LoadInt64(&w.last) < time.Now().Add(-10\*time.Second).UnixNano() {

fmt.Println("No keepalives for 10 seconds. Dying.")

os.Exit(1)

}

}

}()

}

Типичным решением этой гонки является использование канала или мьютекса. Чтобы сохранить поведение без блокировки, можно также использовать пакет sync/atomic [10].

**Список использованных источников**

1. Цителов Д., Трифонов В. Динамический поиск гонок в Java-программах на основе синхронизационных контрактов // Devexperts LLC, СПбГУ, конференция «Инструменты и методы анализа программ (TMPA-2013), 2013.
2. Wood B.P. Software and Hardware Support for Data-Race Exceptions : PhD diss. / Benjamin P. Wood ; University of Washington. — Seattle, WA, USA, 2014.
3. Цукалос Михалис. Golang для профи: работа с сетью, многопоточность, структуры данных и машинное обучение с Go. — СПб.: Питер, 2020. — 720 с.: ил. — (Серия «Для профессионалов»).
4. К чему приводит обилие конкурентности в кодовой базе Go? [Электронный ресурс] URL: https://proglib.io/p/k-chemu-privodit-obilie-konkurentnosti-v-kodovoy-baze-go-2022-05-22 (дата обращения: 23.05.2025)
5. Golang: Гонки данных в Golang простыми словами [Электронный ресурс] URL: https://golang-blog.blogspot.com/2020/03/data-races.html (дата обращения: 23.05.2025)
6. Data Race Detector - The Go Programming Language [Электронный ресурс] URL: https://go.dev/doc/articles/race\_detector (дата обращения: 23.05.2025)
7. The Go Memory Model - The Go Programming Language [Электронный ресурс] URL: https://go.dev/ref/mem (дата обращения: 23.05.2025)
8. Fava D.S., Steffen M. Ready, Set, Go! Data-Race Detection and the Go Language. Science of Computer Programming, 2020.
9. Chabbi M., Ramanathan M. A Study of Real-World Data Races in Golang. PLDI 2022.
10. Golang: Работа с гонками данных (data race) в Golang [Электронный ресурс] URL: https://golang-blog.blogspot.com/2020/01/data-race-golang.html?ysclid=mb0z4bahll930219716 (дата обращения 23.05.2025)