Введение

Живая блокировка (Live Lock) — это одна из классических проблем многопоточного программирования. В отличие от Deadlock, где все потоки зависают и ничего не делают, в случае Live Lock потоки продолжают работать, но не могут продвинуться к выполнению своей задачи из-за постоянной реакции на действия друг друга.

Эта проблема особенно актуальна при разработке систем с высокой степенью параллелизма, таких как серверы, распределённые системы, обработчики событий и конкурентные алгоритмы [1].

В данной работе будет рассмотрена проблема Live Lock , её причины, пример реализации на языке Go , а также методы предотвращения. Также будут предложены две дополнительные задачи, связанные с этой темой.

1. Что такое Live Lock

Live Lock — это состояние системы, при котором два или более процессов активно реагируют на действия друг друга, освобождая ресурсы и повторяя попытку захвата, вместо того чтобы использовать их. Таким образом, прогресс не достигается, несмотря на активность всех участников.

Такое поведение часто возникает, когда процессы используют неблокирующиеся операции , такие как TryLock, и реагируют на конфликты , уступая друг другу без ожидания [2].

Пример из жизни:

Два человека пытаются пройти через узкий проход одновременно. Каждый уступает другому, и в результате никто не может пройти первым.

Примером может быть простой алгоритм консенсуса. Предположим, что есть некоторый алгоритм, который согласует текущее значение какого-то регистра. Если в момент выработки консенсуса приходит сообщение об обновлении данных, алгоритм сбрасывается и начинает сначала. Если такие сообщения приходят слишком часто, получается тот самый live lock - система постоянно работает, но никак не может согласовать значение. Упрощенным примером (это не алгоритм консенсуса) будет подсчет всех записей, которыми управляет приложение - если в момент подсчета прилетает сообщение о том, что была изменена или добавлена запись, алгоритм должен начаться сначала. Если сообщения добавляются слишком часто, система попадает в live lock.

2. Как возникает Live Lock

Live Lock обычно возникает в следующих случаях:

При использовании неблокирующихся примитивов синхронизации , например, TryLock

Когда несколько горутин активно уступают ресурс друг другу , вместо ожидания

В системах, где используется повторная попытка после отказа , без случайного элемента или таймаута

Код, в котором может произойти Live Lock:

```
package main

import (
```

```
    "fmt"
    "sync"
    "time"
)

type Chopstick struct {
    mutex sync.Mutex
}

func (c *Chopstick) PickUp(wg *sync.WaitGroup) {
    defer wg.Done()
    for {
        if c.mutex.TryLock() {
            fmt.Println("Chopstick picked up")
            time.Sleep(500 * time.Millisecond)
            c.mutex.Unlock()
            break
        } else {
            fmt.Println("Chopstick busy, retrying...")
            time.Sleep(100 * time.Millisecond)
        }
    }
}

func main() {
    var wg sync.WaitGroup
    cs := &Chopstick{}

    wg.Add(2)
    go cs.PickUp(&wg)
    go cs.PickUp(&wg)

    wg.Wait()
}
```

Этот код демонстрирует ситуацию, где две горутины постоянно конкурируют за один ресурс , вызывая Live Lock.

3. Разница между Deadlock, Starvation и Live Lock

Тип проблемы	Описание	Реакция потоков
Deadlock	Все потоки заблокированы и ждут друг друга	Никто не работает
Starvation	Один или несколько потоков никогда не получают ресурс	Остальные работают, один голодает
Live Lock	Потоки работают, но не продвигаются	Активное "уступание"

4. Где встречается Live Lock

Примеры:

Сетевые протоколы : Ethernet, Wi-Fi, Bluetooth — устройства могут "переговариваться", откладывая передачу данных [3]

Распределённые системы : нода пытается выполнить задачу, но постоянно перепланируется из-за изменений состояния других нод

Базы данных : транзакции, которые откатываются из-за конкуренции и повторяются снова

Графические движки : объекты постоянно перемещаются, чтобы не пересекаться

5. Решение проблемы в коде

Чтобы решить проблему Live Lock, можно применить следующие подходы:

1. **Добавить случайную задержку перед повторной попыткой**

```
func (c *Chopstick) PickUpFixed(wg *sync.WaitGroup) {
    defer wg.Done()
    for {
        if c.mutex.TryLock() {
            fmt.Println("Chopstick picked up")
            time.Sleep(500 * time.Millisecond)
            c.mutex.Unlock()
            break
        } else {
            delay := time.Duration(rand.Intn(300)+100) * time.Millisecond
            fmt.Printf("Chopstick busy, backing off for %v\n", delay)
            time.Sleep(delay)
        }
    }
}
```

2. Использовать ограничения на количество попыток

Можно добавить счётчик попыток и принудительно завершать работу, если ресурс недоступен слишком долго.

3. Использовать контекст с таймером

Для долгих операций полезно использовать **context.WithTimeout**.

6. Пример тестирования с параллелизмом

```
func TestLivelock(t *testing.T) {
    var wg sync.WaitGroup
    cs := &Chopstick{mutex: sync.Mutex{}}

    runTest := func(id int, t *testing.T) {
        for i := 0; i < 100; i++ {
            if cs.mutex.TryLock() {
                time.Sleep(time.Duration(rand.Intn(10)) * time.Millisecond)
                cs.mutex.Unlock()
                break
            } else {
                time.Sleep(time.Duration(rand.Intn(300)+100) * time.Millisecond)
            }
        }
        t.Logf("Goroutine %d finished\n", id)
    }

    for i := 0; i < 10; i++ {
```

```
        wg.Add(1)
        go func(id int) {
            defer wg.Done()
            runTest(id, t)
        }(i)
    }

    wg.Wait()
}
```

Такой тест эмулирует случайное поведение и помогает проверить стабильность решения.

7. Дополнительные задачи по concurrency

Задача 1:

Следует реализовать систему обработки задач, где воркеры постоянно переходят к следующей задаче, если текущая занята, без ожидания ("перепрыгивают"). Это может привести к Live Lock.

```
package main

import (
	"fmt"
	"math/rand"
	"sync"
	"time"
)

type Task struct {
	ID        int
	mu        sync.Mutex
	processed bool
}

func (t *Task) Process(workerID int) bool {
	locked := t.mu.TryLock()
	if !locked {
		return false
	}
	defer t.mu.Unlock()

	if t.processed {
		return false
	}

	fmt.Printf("Worker %d processing task %d\n", workerID, t.ID)
	time.Sleep(time.Duration(rand.Intn(100)+50) * time.Millisecond)
	t.processed = true

	return true
}
```

```
func worker(workerID int, tasks <-chan *Task, wg *sync.WaitGroup) {
	defer wg.Done()

	for task := range tasks {
		if task.Process(workerID) {
			continue
		}
		go func(t *Task) { tasks <- t }(task)
	}
}

func main() {
	const numWorkers = 5
	const numTasks = 20

	rand.Seed(time.Now().UnixNano())

	taskList := make([]*Task, numTasks)
	for i := 0; i < numTasks; i++ {
		taskList[i] = &Task{ID: i + 1}
	}

	tasks := make(chan *Task, numTasks)
	for _, task := range taskList {
		tasks <- task
	}

	var wg sync.WaitGroup
	wg.Add(numWorkers)

	for i := 0; i < numWorkers; i++ {
		go worker(i+1, tasks, &wg)
	}

	for {
		allProcessed := true
		for _, task := range taskList {
			task.mu.Lock()
			if !task.processed {
				allProcessed = false
				task.mu.Unlock()
				break
			}
			task.mu.Unlock()
		}

		if allProcessed {
			close(tasks)
			break
		}
		time.Sleep(100 * time.Millisecond)
	}
```

```
    wg.Wait()
    fmt.Println("All tasks processed")
}
```

Ключевые моменты решения:

1. **Использование TryLock**: Воркеры пытаются захватить задачу без блокировки. Если не получается, они переходят к следующей.
2. **Отметка о выполнении**: Каждая задача имеет флаг processed, чтобы избежать повторной обработки.
3. **Возврат задач в очередь**: Если задача не была обработана (занята или уже выполнена), она возвращается в канал для повторной обработки.
4. **Контроль завершения**: Главная горутина проверяет, все ли задачи выполнены, перед закрытием канала.
5. **Буферизированный канал**: Позволяет избежать блокировки при возврате задач в очередь. Это решение предотвращает live lock, гарантируя, что:

- Каждая задача будет обработана ровно один раз
- Воркеры не будут бесконечно "перепрыгивать" между задачами
- Система корректно завершится после обработки всех задач

Задача 2:

У нас есть очередь задач и несколько воркеров, которые берут задачи из очереди. Если очередь быстро обновляется, возможен случай, когда воркеры никогда не находят задачи, хотя они есть.

```
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

func main() {
    const numWorkers = 5
    const queueSize = 100
    const totalTasks = 1000

    var wg sync.WaitGroup
    taskQueue := make(chan int, queueSize)
    rand.Seed(time.Now().UnixNano())

    wg.Add(numWorkers)
    for i := 0; i < numWorkers; i++ {
        go func(id int) {
            defer wg.Done()
            for {
                select {
                case task, ok := <-taskQueue:
```

```
                                        if !ok {
                                                return
                                        }
                                        fmt.Printf("Worker %d got task %d\n", id, task)
                                        time.Sleep(time.Duration(rand.Intn(10))
                                default:
                                        time.Sleep(time.Duration(rand.Intn(10)))
                                }
                        }
                }(i)
        }

        go func() {
                for i := 0; i < totalTasks; i++ {
                        taskQueue <- i
                        time.Sleep(time.Duration(rand.Intn(5)))
                }
                close(taskQueue)
        }()

        wg.Wait()
        fmt.Println("All workers done")
}

package main

import (
        "fmt"
        "math/rand"
        "sync"
        "time"
)

func main() {
        const numWorkers = 5
        const queueSize = 100
        const totalTasks = 1000

        var wg sync.WaitGroup
        taskQueue := make(chan int, queueSize)
        rand.Seed(time.Now().UnixNano())

        wg.Add(numWorkers)
        for i := 0; i < numWorkers; i++ {
                go func(id int) {
                        defer wg.Done()
                        backoff := time.Duration(0)
                        for {
                                select {
                                case task, ok := <-taskQueue:
                                        if !ok {
                                                return
```

```
                                        }
                                        backoff = 0
                                        fmt.Printf("Worker %d got task %d\n", id, task)
                                        time.Sleep(time.Duration(rand.Intn(10)))
                                default:
                                        if backoff < time.Millisecond*100 {
                                                backoff += time.Millisecond * 10
                                        }
                                        time.Sleep(backoff)
                                }
                        }
                }(i)
        }

        go func() {
                for i := 0; i < totalTasks; i++ {
                        taskQueue <- i
                        time.Sleep(time.Duration(rand.Intn(5)))
                }
                close(taskQueue)
        }()

        wg.Wait()
        fmt.Println("All workers done")
}
```

Ключевые моменты решения проблемы live lock:

1. **Экспоненциальная задержка (backoff)**
   - Воркеры увеличивают время ожидания при пустой очереди
   - Начинают с малой задержки (10ms), постепенно увеличивают до максимума (100ms)
   - Сбрасывают задержку при успешном получении задачи
2. **Приоритет обработки над проверкой очереди**
   - Используется select с приоритетным чтением из канала (case task)
   - Только при отсутствии задач переходит к default с задержкой
3. **Контроль скорости потребления**
   - Воркеры искусственно замедляют обработку (rand.Intn(10))
   - Позволяет производителю добавлять новые задачи
4. **Гибкое управление потоком**
   - Разная скорость производства (5ms) и потребления (10ms) задач
   - Буферизированный канал (size=100) сглаживает пики нагрузки
5. **Гарантия завершения**
   - Явное закрытие канала после всех задач
   - sync.WaitGroup для корректного завершения воркеров

Стратегия эффективна потому что:

- Уменьшает конкуренцию воркеров при малом количестве задач
- Сохраняет быструю реакцию при появлении новых задач
- Автоматически адаптируется к нагрузке без ручной настройки

9. Заключение

Live Lock — это сложная и трудно диагностируемая проблема в параллельном программировании. Она возникает, когда потоки или горутины постоянно реагируют на действия друг друга, не позволяя себе выполниться .
Особенно важна борьба с этим явлением в высоконагруженных системах, где применяется неблокирующаяся синхронизация и алгоритмы типа try-lock-release-retry [4].

Используя рандомные задержки , ограничения на попытки , и правильные стратегии ожидания , можно эффективно избежать Live Lock.
Представленная практическая реализация и тестирование показывают, как эта проблема может быть выявлена и исправлена в реальных условиях. Эти знания применимы в разработке высоконагруженных систем, микросервисов и распределённых архитектур [5].

10. Список источников

1. Goetz, B. et al. *Java Concurrency in Practice* . Addison-Wesley, 2006.
2. Herlihy, M., Shavit, N. *The Art of Multiprocessor Programming* . Morgan Kaufmann, 2008.
3. Birrell, A. *An Introduction to Programming with Threads* . Digital Equipment Corporation, 2003.
4. Burns, J.E., Peterson, G.L. *The Best Algorithm for Mutual Exclusion* . ACM Transactions on Programming Languages and Systems, 1983.
5. Ben-Ari, M. *Principles of Concurrent and Distributed Programming* . Addison-Wesley, 2006.
6. https://habr.com/ru/articles/345144/
7. https://www.geeksforgeeks.org/deadlock-starvation-and-livelock/
8. https://learn.microsoft.com/en-us/dotnet/csharp/async
9. https://pkg.go.dev/sync#Mutex
10. https://medium.com/@tyler.treat/advanced-concurrency-in-go-84f2fe1e35b0
11. https://gobyexample.com/mutexes
12. https://go.dev/tour/concurrency/1
13. https://github.com/golang/go/wiki/MutexOrChannel