GoLang. Планировщик

#Потоки #Планировщик #Goroutine #go

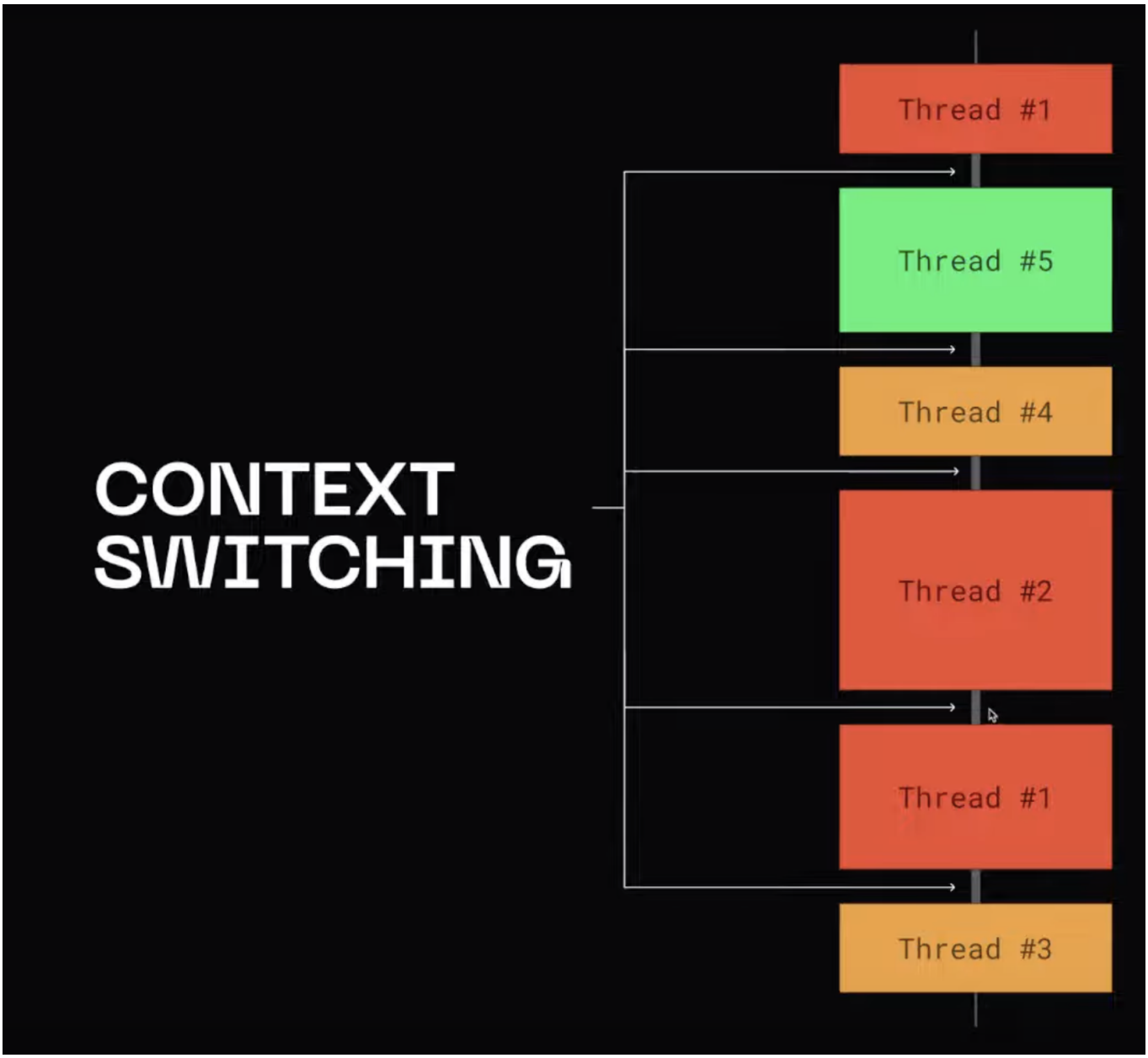
Виды многозадачности

---

1. **Вытесняющая**: Все программы равны. Выделю каждой строго одинаковое количество времени. Так обеспечу справедливость и равенство.
2. Кооперативная: пусть программы выполняются столько, сколько им нужно, и сами уступают друг другу место. Так обеспечу гармонию и эффективность. go

Как это работает нативно

Изначально в ОС Линукс, создание процессов происходит медленно, так как создание происходит на уровне ядра, с передачи контекста (стека, копирования регистров и т.д.)



Данный подход является не эффективным для нашей задачи (разработка сервера).

Как сделать эффективнее

Создадим свой *context switching*, сделав его легковесным. Будем использовать ассемблер для снятия регистров и пуша в стек.

Ядро не знает про смену контекста, поэтому в наш код будет работать быстрее (мы работаем в user space, а не kernel space).

Стек потока в среднем 8 мб (нам не нужно столько)
Стек гоу-рутины от 2х кб

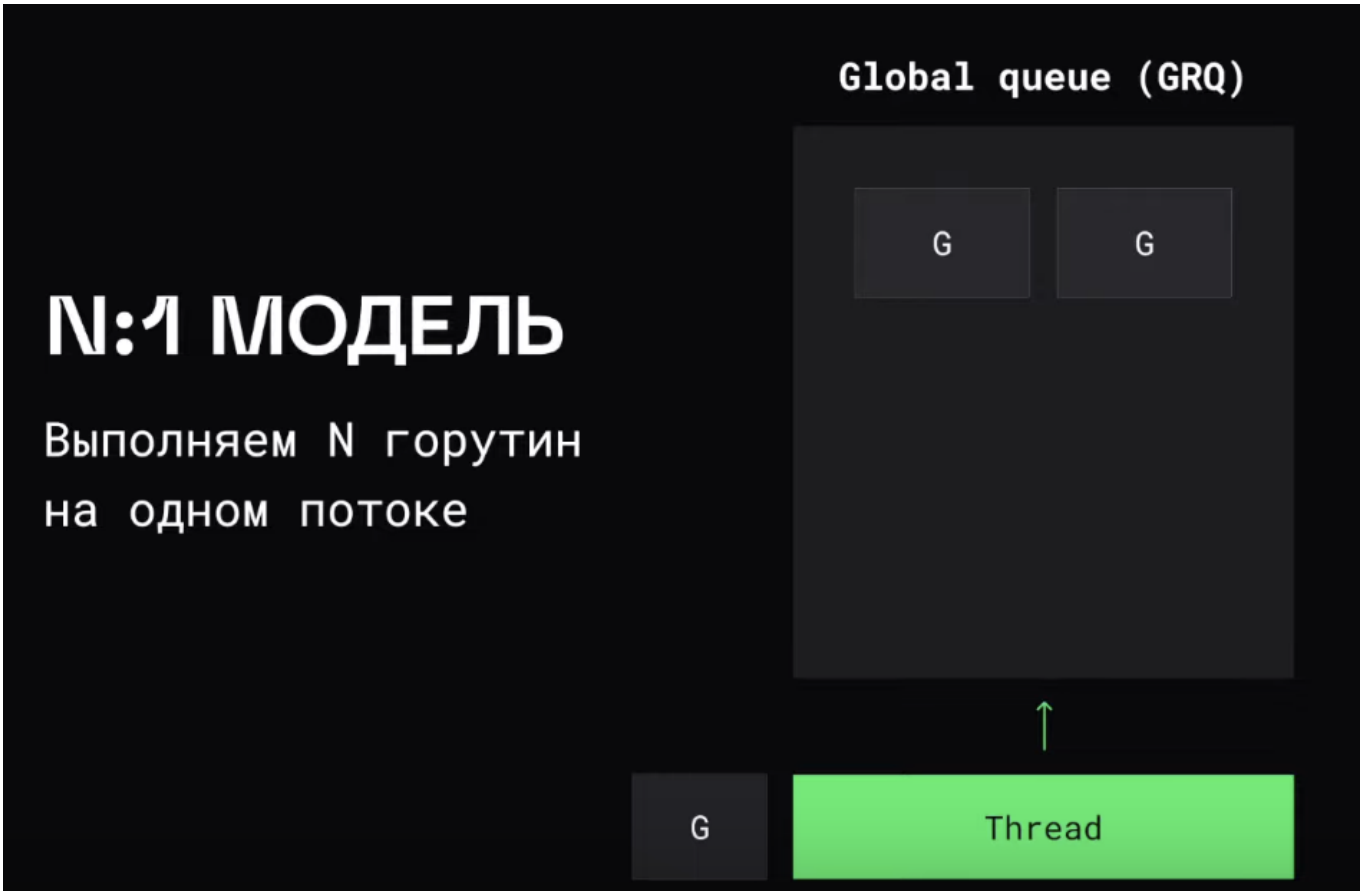
И теперь на 10000 соединений нам надо 20 мб , а не 80 гб. Мы также выигрываем в скорости из-за user space'a.=

Ключевая идея go-routine

1. Не даем в руки программисту поток.
2. Делаем вид, что есть только горутины.
3. Всю сложность распределения горутин абстрагируем.

N к 1 (N:1) модель

Выполняем N горутин на одном потоке. Горутины выполняются в очереди.

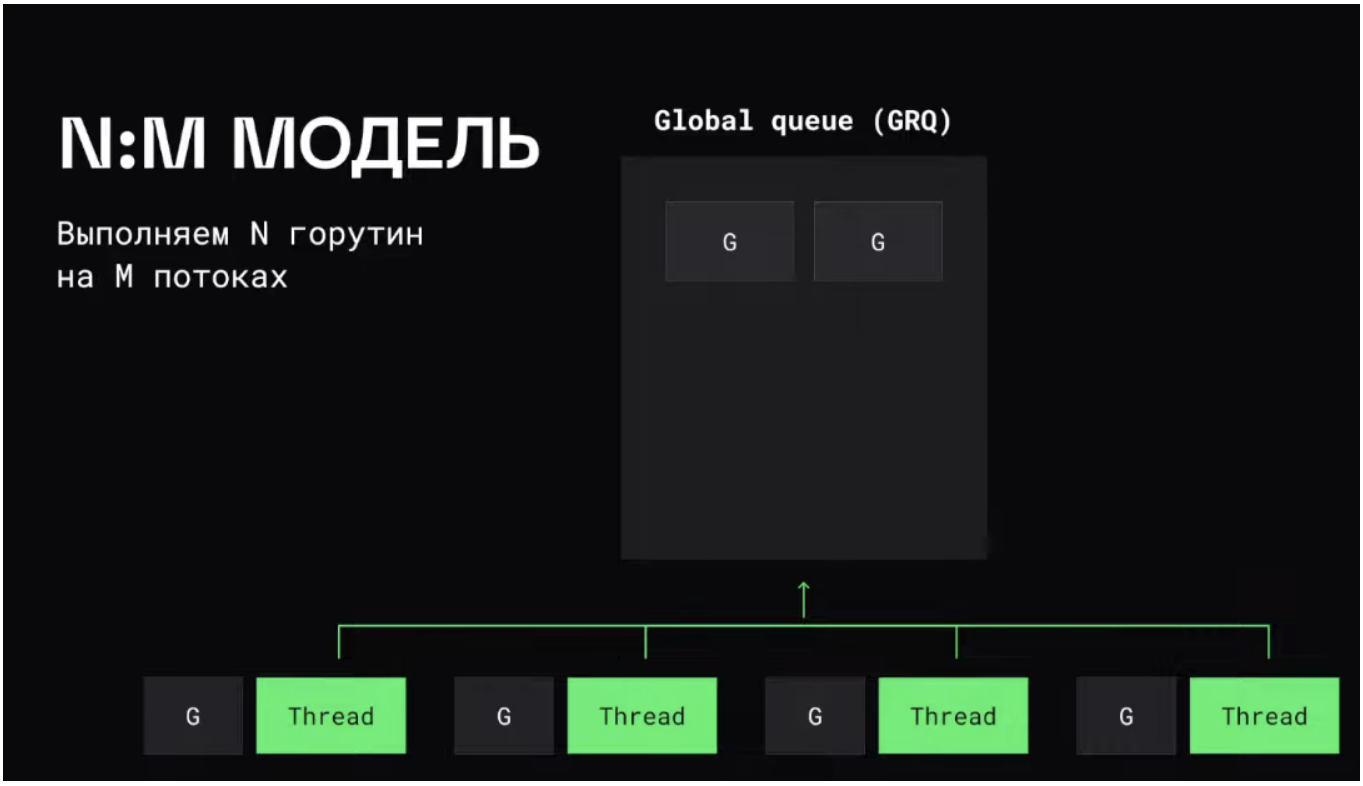


Как и когда вытеснять горутины?

Компилятор добавит места в исходный код для переключения контекста. (Например в момент проверки необходимого пространства для стека гоурутины (кооперативная многозадачность)

Как быть с многоядерными процессорами?

Тут приходит на помощь модель N:M. Выполняем N горутин на M потоках.



Однако подход все равно не масштабируемый. Ведь если мы добавим Мутексы, то все сломается.

Проблему можно решить делением очереди на локальные очереди. (шрадируем очереди). Каждый поток обращается к своей локальной очереди и проблема избыточной синхронизации решена (мутекса).
И мы приходим к GMP модели.

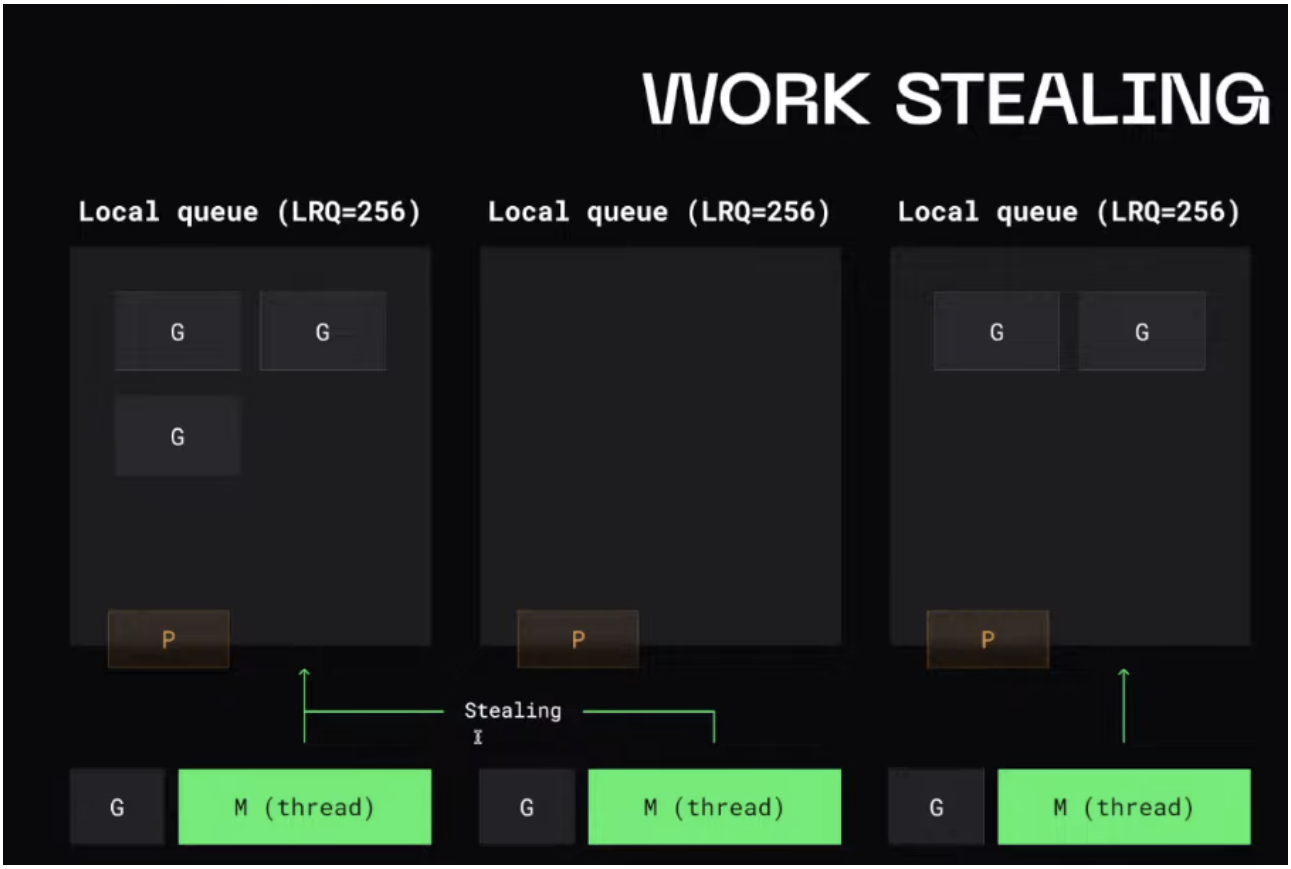
GMP модель

```
G – Goroutine (что исполняем) Код
M – Machine (где исполняем) Поток ОС
P – Processor (права и ресурсы для исполнения)  Очередь
```

В какую из очередей добавлять новую рутину?

- В локальную очередь
  А что делать, если в очереди закончатся рутины?

- Work Stealing



  Но нам нужна синхронизация , в контексте go используется lock-free.
  А у какой именно очереди красть?
- Рандомно
  Что делать, если в другой очереди тоже нет горутин?
- Пробуем 4 раза
  Сколько красть из другой локальной очереди ?
- Крадем половину горутин

Syscalls

Создается отдельный пулл с потоками для системных вызовов.

Планировщик Go сразу открепит поток (M) от процессора (P), если понимает, что поток будет заблокирован на системном вызове в течении долгого времени

• В других случаях (short-lived syscalls) он позволит потоку быть заблокированным и не будет откреплять от процессора

Но опять проблема, если системный планировщик не будет долгое время выполнять syscall , тогда на помощь придет отдельный thread - sysmon (system monitoring), который

переодически проверяет потоки, если какой-то поток встрял, то мы его открепляем.

После выполнения syscall'а рутина кладется в свою локальную очередь (в который она была) , но если capacity этой очереди превышен, то кладем ее в новую ГЛОБАЛЬНУЮ очередь. Для синхронизации глобальной очереди используем Mutex.

Очередь

Проблема глобальной очереди решается за счет runtime.schedule() , он раз в 1/61 проверяет глобальную очередь.

Ограничения планировщика Go

```
Очередь FIFO
Отсутствие гарантий времени выполнения
Горутины могут перемещаться между тредами, что снижает эффективность кэшей
(но может сделать runtime.LockOSThread())
```

Что делает слово go?

```go
package main

import (
        "fmt"
)

func main() {
        go fmt.Println("Hello")
}
```

Может вывести что-то, а может и нет. Зависит от того, закончится быстрее main или горутина.

А теперь поехали под капот.

Функция go вызывает другую функцию из пакета runtime

```
go func == runtime.newproc(func)
```

Что делает runtime.newproc(func)?
Она вызывает newproc1(fn...):

```
// Create a new g running fn
// g is put on the queue
func newproc1(fn...) {
        _g_ := getg()
        _p_ := _g_.m.p.ptr()
        ...
        newg := gfget(_p_)
        newg.startpc = fn
        newg.return_to = goexit
        runqput(_p_, newg)
}
```

То есть мы не запускаем горутину сразу, мы просто кладем ее в очередь.

Все главное происходит в функции schedule().

```
// One round of scheduler: find a runnable goroutine and execute it
// Never returns
func schedule() {
        g := get_gc_worker()
        if g == nil {
                g = runqget()
        }
        if g == nil {
                g = steal_or_wait()
        }
        execute(g)
}
```

Так что происходит при запуске *go fmt.Println("Hello")*?

Пролог

```
PUSHL $runtime mainPC() // Добовляем функцию на стек
CALL runtime newproc
CALL runtime mstart // Она вызывает schedule()
```

Акт 1

```
func mstart() {
        ...
        schedule()
}
```

Акт 2

```
func main(){
        go start_gc()
        ...
        main_main()
}
```

Добавляет служебную функцию main, которая запускает сборщик мусора и другие подготовительные действия и запускает наш main().

Акт 3

Запускается main, после в go fmt.Println() запускается newproc1, однако заметим

```
// Create a new g running fn
// g is put on the queue
func newproc1(fn...) {
        _g_ := getg()
        _p_ := _g_.m.p.ptr()
        ...
        newg := gfget(_p_)
        newg.startpc = fn
        newg.return_to = goexit // тут происходит магия
        runqput(_p_, newg)
}
```

Через определенные стековые операция, он добивается того, что при ассемблерном ret, возвращалась функция goexit1.

```
// Finishes execution of the current goroutine
func goexit1() {
        switch_to_main_g()
        gfput()
```

```
        schedule() // опять кладем новую рутину
}
```

Sum up

1. При старте программы код на ассемблере создает тред и запускает в нем первый раунд планировщика
2. Планировщик запускает runtime/proc.go:main()
3. runtime/proc.go:main() запускает служебные задачи (GC), а затем — пользовательский main()
4. При завершении горутины вызывается новый раунд планировщика