Планировщик Go

Серышев Денис

Оглавление

Какие бывают планировщики?

Планировщик ОС управляет тем, как работают процессы

Планировщик контейнеров управляет тем, на каких машинах создаются контейнеры(Kubernetes)

Цель планировщика

Планировщик является частью ОС и отвечает за выполнение какой-либо задачи при этом оптимально выделяя ресурсы для ее выполнения. Планировщик так же может принудительно забирать управление у потока(по таймеру или при появлении потока с большим приоритетом), либо просто ожидать пока поток сам явно или неявно отдаст управление планировщику

Планировщик Go

Суть планировщика Go заключается в распределении готовых к исполнению горутин между потоками ОС, которые могут исполняться одним или большим количеством процессоров. В многопоточных вычислениях возникли две парадигмы в планировании:

1) Work-sharing: когда процессор создает новые потоки, он пытается перенести некоторые из них на другие процессоры в надежде, что они будут использованы незадействованными/недоиспользуемыми процессорами.
2) Work-stealing (используется планировщиком Go): свободный процессор активно ищет чужие потоки других процессоров и “*ворует*” их.

При использовании парадигмы work-stealing миграции происходят реже нежели при использовании work-sharing. Когда все процессоры выполняют работу, потоки не переносятся. И как только появляется простаивающий процессор, рассматривается миграция.

Go использует M:N планировщик, который может использовать множество процессоров.

N горутин выполняется на M потоках.

Планировщик вызывается во время выполнения некоторых конструкций(блокировки горутин операциями с каналами или time.Sleep)

Так же GOMAXPROCS равен количеству **P**

Абстракции для понимания планировщика

G – горутина

M “machine” – тред (поток ОС)

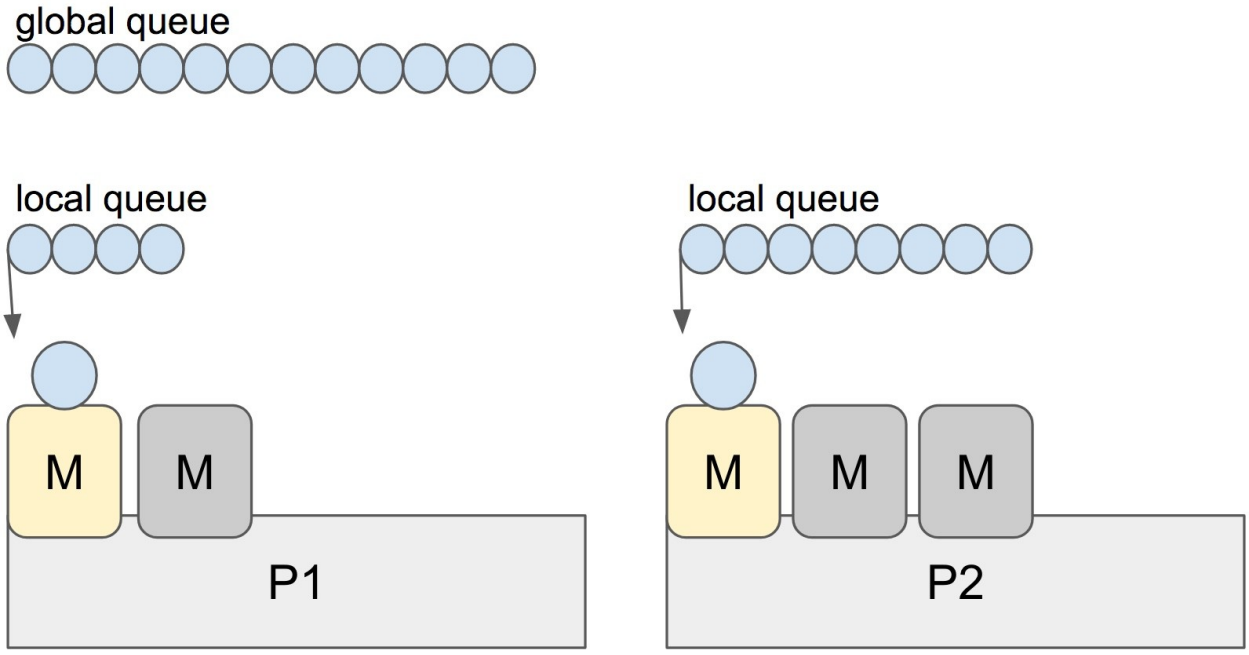
P “processor” – то, что выполняет наш Go-код

1)Каждому **M** должен быть назначен **P**

2) **P** может быть заблокирован или ожидать системный вызов, тогда у **P** не может быть **M**

3)Только один **M** может исполняться на каждом **P** в текущий момент

4)Если необходимо планировщик может создавать дополнительные **M**



Этап планирования – это просто поиск исполняемой горутины и ее выполнение. На каждом этапе поиск осуществляется по данному принципу:

```
runtime.Schedule() {
    // only 1/61 of the time, check the global runnable queue for a G.
    // if not found, check the local queue.
    // if not found,
    //     try to steal from other Ps.
    //     if not, check the global runnable queue.
    //     if not found, poll network.
}
```

Перевод

```
runtime.schedule() {
       // только 1/61 всего времени проверяется глобальная очередь для
G
       // если ничего не было найдено, то проверить локальную очередь
       // если ничего не нашлось, то:
       //     попытаться украсть у других Ps
       //     если не получилось, то проверить глобальную очередь
       //     если не вышло, то поллить(poll) сеть
}
```

То есть, время от времени (1/61) планировщик берет горутины из глобальной очереди.
Если в глобальной очереди ничего не нашлось, то планировщик проверяет из локальной очереди текущего **P**
Если в локальной очереди ничего нет, ищет и “*крадет*” горутины у других P (равномерно перераспределению очереди)
Проверка глобальной очереди важна, чтобы избежать использование горутин только из локальной очереди
Локальная очередь есть у каждого треда и хранит в себе горутины, которые готовы к выполнению.
Преимущество **локальной очереди**:

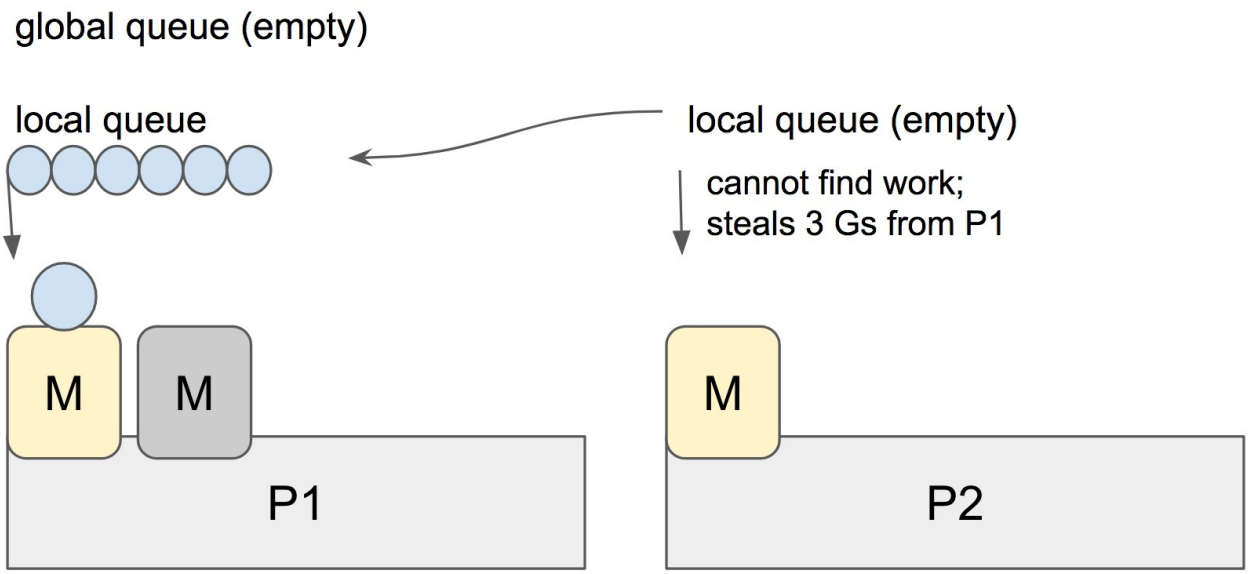
- Быстрый доступ: каждый поток имеет прямой доступ к своей локальной очереди без необходимости синхронизации с другими потоками.

Глобальная очередь является общей очередью, которая служит для распределения горутин между потоками. Когда локальная очередь потока опустеет, то поток может обратиться к глобальной очереди для получения новых горутин, которые готовы к выполнению.
Преимущество **глобальной очереди**:

- Балансировка нагрузки: глобальная очередь позволяет равномерно распределять горутины между потоками, что способствует более эффективному использованию ресурсов процессора.

Stealing

Когда новая **G** создается или существующая **G** становится готовой к выполнению, она помещается в локальную очередь готовых к исполнению горутин текущего **P**. Когда **P** завершает свою работу с **G**, он пытается извлечь **G** из своей очереди запущенных горутин. Если очередь пуста, то **P** выбирает рандомом(случайный образом) другой процессор (**P**) и пытается украсть половину горутин из его очереди.



На картинке выше показано, что **P2** не может найти запущенных горутин. Соответственно, он крадет три горутины у другого процессора **P1** и добавляет(push) их в свою локальную очередь. **P2** теперь способен запустить эти горутины и работа планировщика будет справедливо распределена между несколькими процессорами.

Spinning Threads

Планировщик всегда хочет распределить как можно больше готовых к исполнению горутин на много **M**, чтобы использовать все процессоры. Но в тоже время, мы должны уметь приостанавливать прожорливые процессы, чтобы сохранить CPU&Power.
Так же планировщик должен уметь масштабироваться для задач, которые действительно требуют много вычислительной мощности процессора и большую производительность.

Постоянное вытеснение одновременно и дорогое, и проблематичное для высокопроизводительных программ, где производительность является ключевым фактором. Горутины не должны постоянно прыгать между потоками ОС, потому что это приводит к увеличению задержки(latency).
Ко всему этому, когда вызываются системные вызовы, потоки ОС должны постоянно блокироваться и разблокироваться. Это достаточно дорого.

Чтобы горутины не прыгали между потоками ОС, планировщик Go реализует зацикленные циклы (spinning threads). Spinning threads потребляют чуть больше CPU, но уменьшают вытеснение потоков. Поток является зацикленным если:

1)**M** назначенный на **P** ищет горутину, которую бы можно было запустить

2)**M** неназначенный на **P**, ищет доступные **P**

3)Планировщик так же запускает дополнительный поток и зацикливает его, когда готовит новую горутину, и есть простаивающий **P** и нет других зацикленных потоков.

Переключение Горутин

В каких случаях происходит переключение горутин?

1) runtime.Gosched(): данная функция используется для явного предоставления возможности планировщику переключиться между горутинами. Так, а в чем смысл этой функции? Ее основная цель заключается в том, чтобы предотвратить длительную блокировку одной горутины и дать возможность другим горутинам запуститься и выполнить свою работу. Это бывает полезно, когда у вас есть вычислительно интенсивная горутина, которая может заблокировать остальные горутины, приводя к проблемам с пропускной способностью системы. Так же, когда происходит вызов runtime.Gosched(), то происходит переключение контекста между горутинами, давая им возможность начать или продолжить выполнение. Это не требует каких-либо явных синхронизаций или блокировок со стороны разработчика. Стоит отметить, что runtime.Gosched() не гарантирует немедленное переключение горутин. Это всего лишь сигнал планировщику, и фактический момент переключения зависит от внутренней реализации. В большинстве случаев планировщик все делает за нас, но в некоторых особых кейсах, например, при явной паузе в выполнении или при необходимости дать возможность горутинам запуститься, данная функция может быть полезна.

2) Операции с сетью(чтение/запись): когда горутина выполняет операцию ввода/ввывода(I/O), которая блокируется(например, чтение из сетевого соединения или запись в сетевой сокет) планировщик автоматически переключает выполнение на другую горутину. Это называется “событийно-ориентированным” или “неблокирующим” вводом/выводом. В Go I/O операции обычно выполняются асинхронно или с помощью мультиплексирования (I/O multiplexing). Это означает, что горутина инициирует I/O операцию, а затем планировщик переключает выполнение на другие горутины, вместо того чтобы ожидать завершения I/O операции. Когда I/O операция завершается, планировщик снова переключается на горутину, которая инициировала операцию, чтобы продолжить выполнение с полученными данными. Данный механизм позволяет горутинам эффективно использовать процессорное время, не блокируя его на ожидание завершения I/O операций. Таким образом, другие горутины могут продолжать работу, что способствует более эффективному использованию ресурсов.

3) Системные вызовы: когда горутина осуществляет системный вызов(ожидание завершения операции на файловом дескрипторе, сетевом соединении или ожидание события таймера), то горутина блокируется. Планировщик осуществляет переключение на другую готовую к выполнению горутину. Когда системный вызов завершается, планировщик снова переключается на горутину, которая делала вызов и продолжает выполнение с результатами системного вызова. За счет этого можно эффективно использовать процессорное время и избегать блокировки и ожидания, позволяя другим горутинам выполняться. Планировщик автоматически управляет переключением, основываясь на состоянии готовности и блокировки каждой горутины.

4) Блокировки(мьютексы, каналы): переключение может происходить, когда горутина блокируется на мьютексах, каналах или других примитивах синхронизации.

> 4.1) Блокировка на мьютексах:
>
> Когда горутина пытается захватить мьютекс с помощью операции Lock() и мьютекс уже захвачен другой горутиной, то в таком случае горутина блокируется. Планировщик переключается на другую готовую к выполнению горутину. Если мьютекс становится доступный для использования, планировщик разблокирует горутину, и она начнет выполнять свою работу с уже захваченным мьютексом.
>
> 4.2) Блокировка на каналах:

Если горутина выполняет операции с каналами, а канал не готов для чтения или записи соответственно(например, попытка чтения из пустого канала или запись в заполненный канал), горутина блокируется. Затем планировщик переключается на другую готовую к выполнению горутину. Когда состояния канала изменяется, и операция чтения или записи становится возможной, планировщик разблокирует горутину и позволит ей продолжить выполнение операций на канале.

4.3) Другие примитивы синхронизации:

Go содержит также и другие примитивы синхронизации, помимо каналов и мьютексов.

sync.Cond – условные переменные, *sync.WaitGroup* – барьеры. Когда горутина ожидает сигнала от этих примитивов или выполняет операции с ними, и условие не выполнено, горутина блокируется, и планировщик опять же переключается на другую готовую к выполнению горутину. По мере изменения состояния примитива синхронизации, планировщик разблокирует горутину и позволит ей продолжить выполнение.

5) Аллокации стека(stack growth), которая может вызвать вызов функции morestack()

В Go каждая горутина имеет свой собственный стек, который используется для хранения локальных переменных и данных выполнения функции. Стеки горутин имеют фиксированный размер, и если горутина пытается выделить больше места на стеке, чем у нее доступно, происходит аллокация дополнительного стека. Вызов morestack() приводит к переключению горутин, так как необходимо выполнить некоторые действия для создания нового стека и сохранения текущего состояния горутины. Планировщик обрабатывает это переключение, выбирает другую готовую к выполнению горутину и переключается. После успешной аллокации нового стека и переключения на него, горутина может продолжить свое выполнение с дополнительным стеком. Это позволяет горутинам динамически расширять свои стеки, чтобы избежать переполнения и сохранить нормальное выполнение программы.

Итог

Планировщик Go делает достаточно вещей для того, чтобы избежать чрезмерного вытеснения потоков ОС, распределяя их по недоиспользованным процессорам методом “*кражи*”, и также реализацией “*зацикленных*” потоков, чтобы избежать частых переходов из блокирующего в неблокирующее состояние и обратно.

Ресурсы

[1] JBD https://rakyll.org/scheduler/

[2] Dmitry Vyukov https://www.youtube.com/watch?v=-K11rY57K7k

[3] Исходники https://github.com/golang/go/blob/master/src/runtime/proc.go#L2468