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)

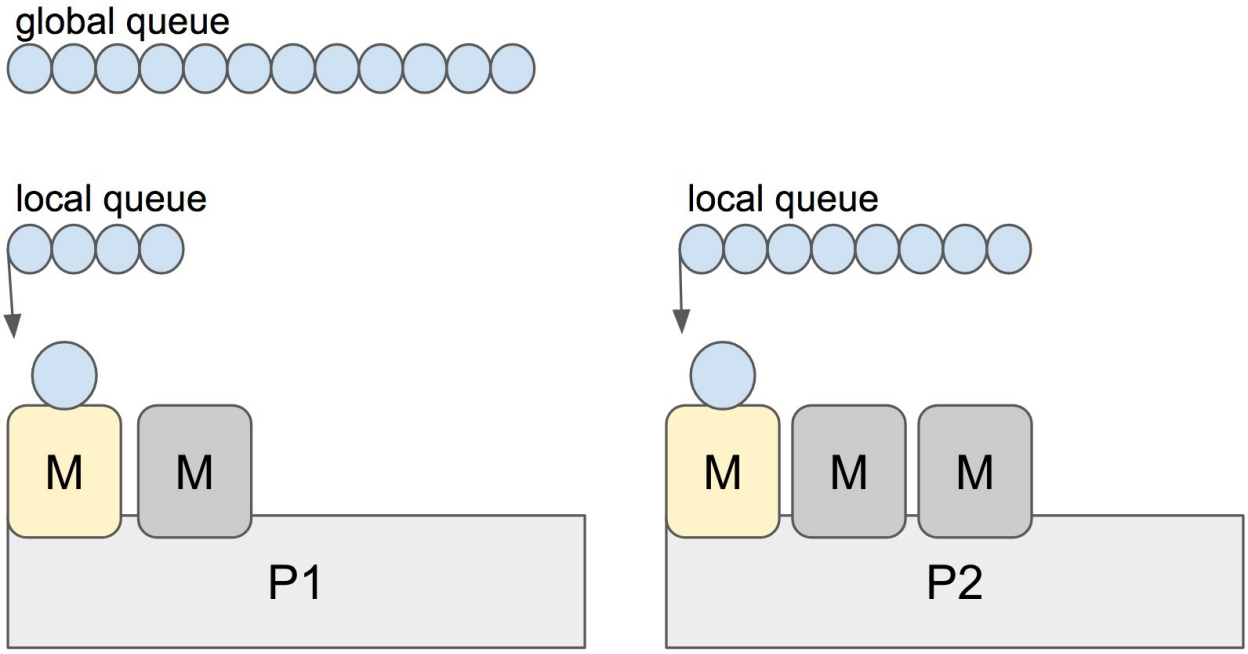
P "processor" – то, что выполняет наш Go-код

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

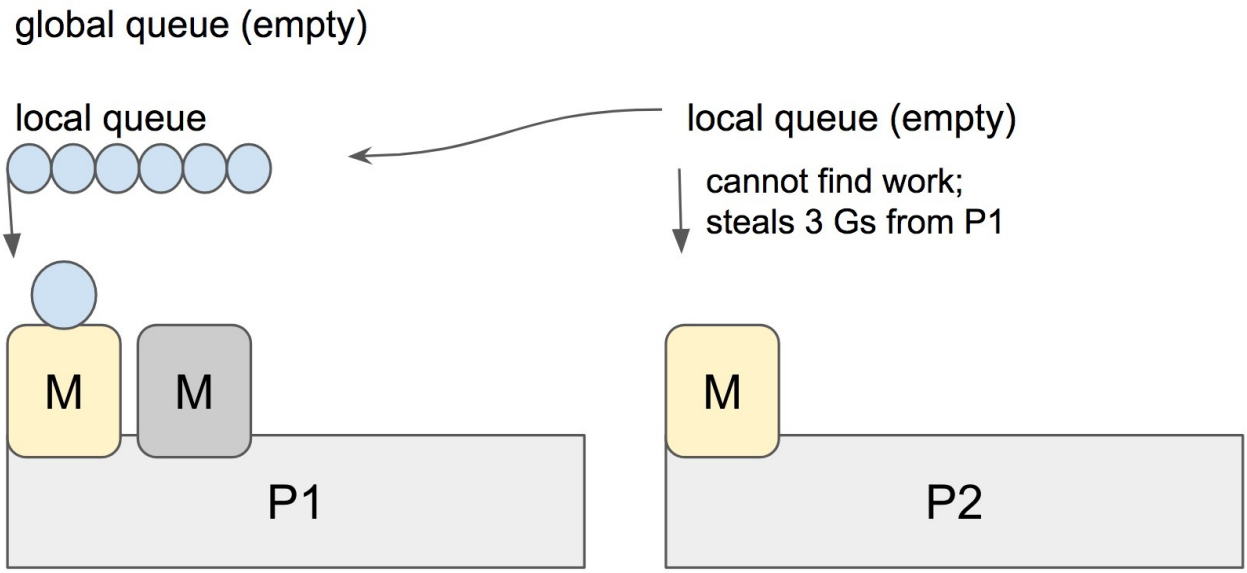
```
runtime.schedule() {
       // только 1/61 всего времени проверяется глобальная очередь для
G
       // если ничего не было найдено, то проверить локальную очередь
       // если ничего не нашлось, то:
       //     попытаться украсть у других Ps
       //     если не получилось, то проверить глобальную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

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()

2)операции с сетью(чтение/запись)

3)системные вызовы

4)блокировки(мьютексы, каналы)

Когда в небуферизированный канал отправляется сообщение, горутина блокируется, попадает в очередь заблокированных. Потом когда происходит чтение, горутина разблокируется.

5)некоторые аллокации morestack()

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