Если вы видите что-то необычное, просто сообщите мне.

Go: горутины , потоки ОС и управление ЦПУ

Создание потоков ОС или переключение из одного в другой может стоить вашей программе память и производительность. Целью go является получение преимущества от ядра настолько, насколько это возможно. Он был создан уже с конкурентностью с самого начала.

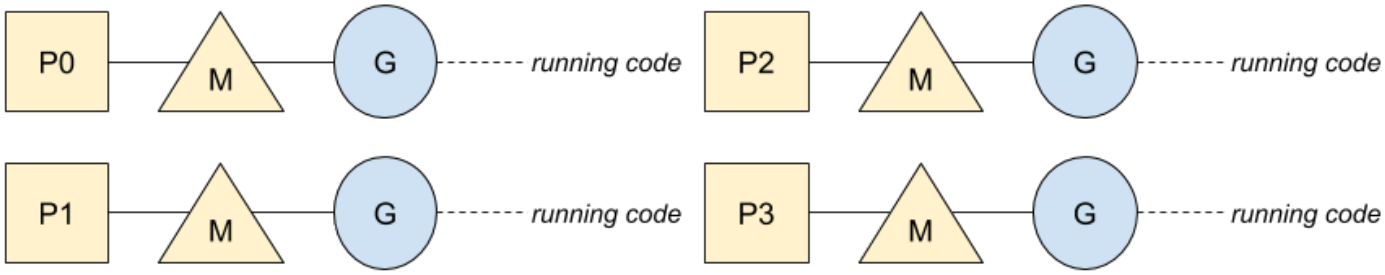
M, P, G оркестрация

Чтобы решить эту проблему, Go имеет своё расписание для распределения горутин через различные потоки. Это расписание определяет три главных идеи, как сказано в самом коде:

Главные идеи:

- G - горутины
- M - рабочий поток или машина
- P - процессор, ресурс для выполнения Go кода
  M должен иметь связанный P, чтобы выполнять код Go.

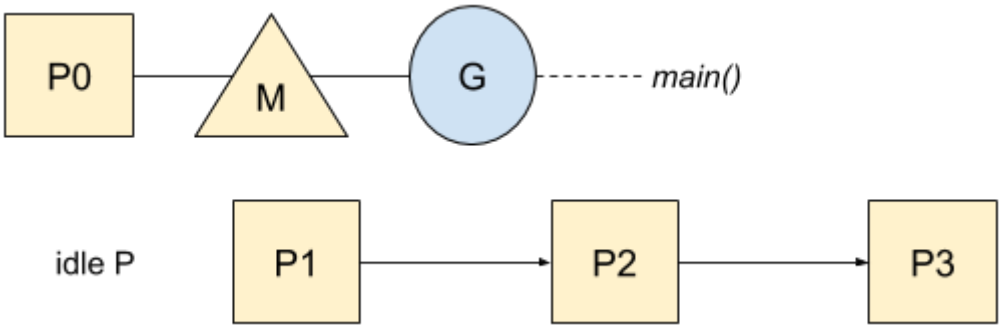
Вот диаграмма этих трех моделей.



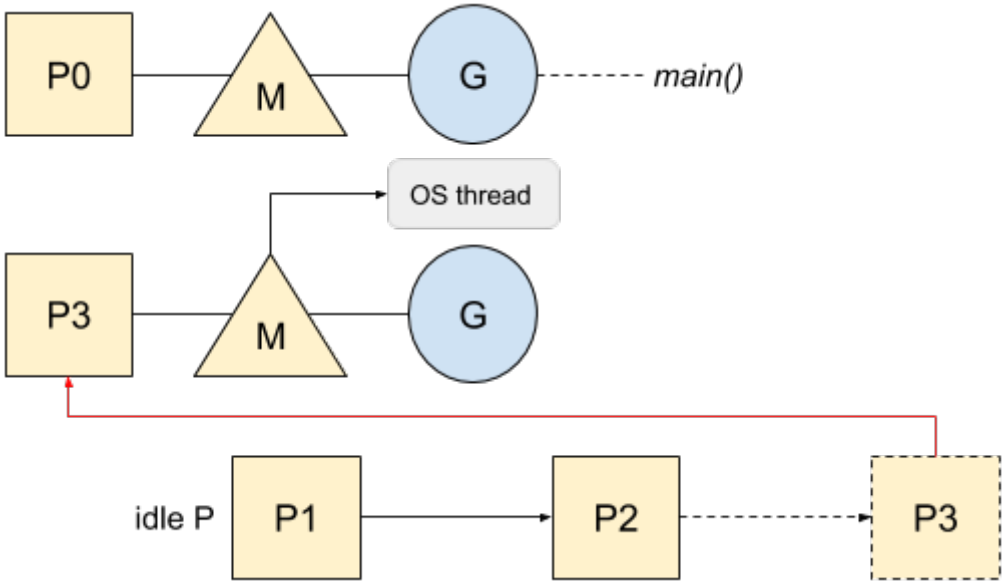
Каждая горутина (G) запускается в потоке ОС (M) которая назначена логическому ЦПУ (P). Давайте возьмем простой пример, чтобы посмотреть как Go управляет ими:

```
func main() {
   var wg sync.WaitGroup
   wg.Add(2)

   go func() {
      println(`hello`)
      wg.Done()
   }()

   go func() {
      println(`world`)
      wg.Done()
   }()

   wg.Wait()
}
```

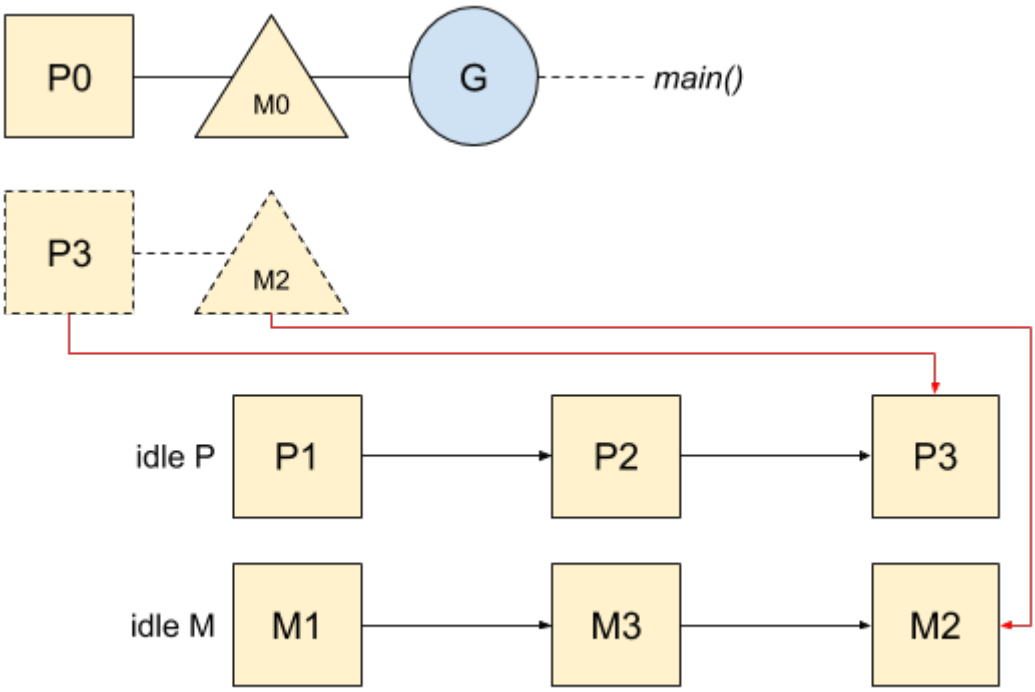
Сначала Go создаст различные P с номерами логических ЦПУ присутствующих в машине и сохранит их в качестве списка "холостых" P:



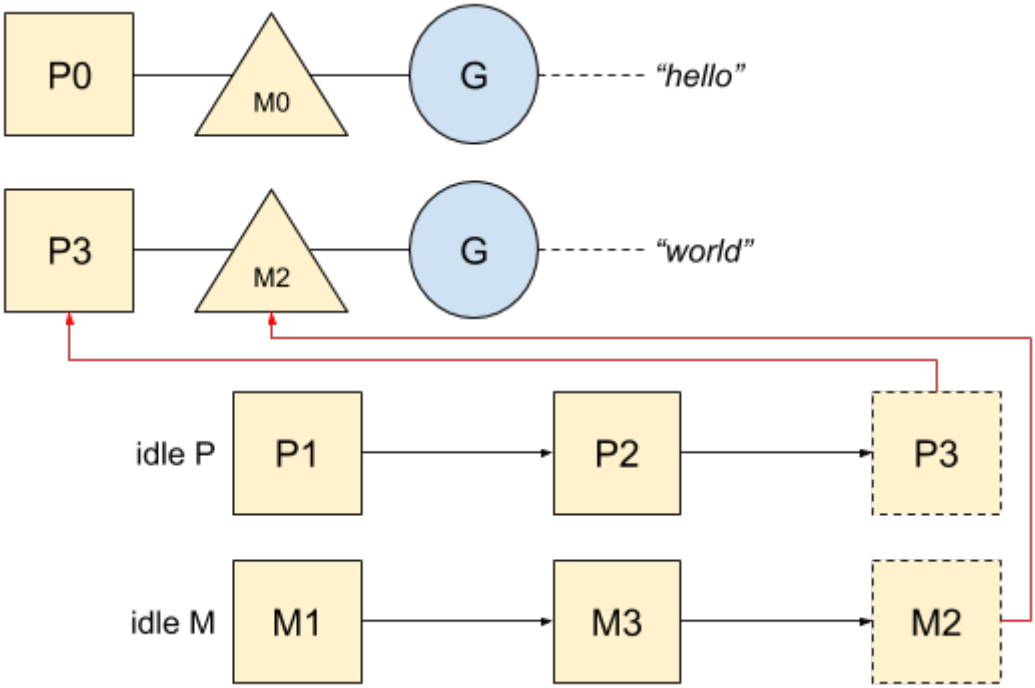
Затем, новая горутина или горутины готовые к запуску будут будить P для назначения на них работ. В этом случае P создаст M и свяжет их с ОС потоками:



Однако, как и P, M без работы - то есть не имеющая работающей горутины ожидающей запуска, возвращается из `syscall` или принудительно завершенная сборщиком мусора, попадет в "холостой" список:



Во время загрузки программы, Go уже создает потоки ОС и связывает их с M. Для нашего примера, первая горутина которая выводит "привет" будет использовать главную горутину, в то время как вторая получит M и P из "холостого" списка:



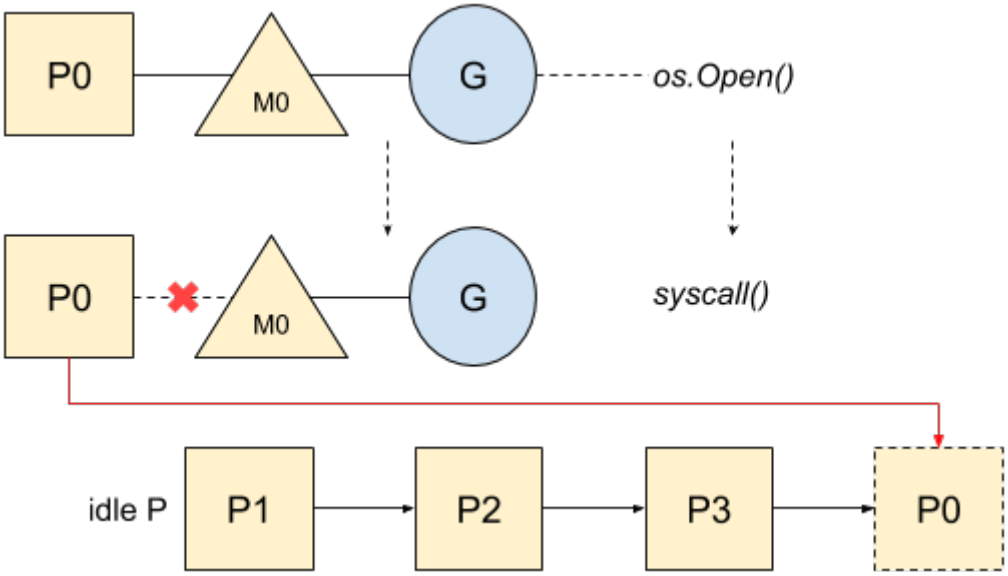
Теперь у нас есть общая картина упрвления горутинами и потоками, давайте посмотрим в какомслучае Go станет использовать M чаще чем P и как горутины управляются в случае системых вызовов.

Системные вызовы

Go оптимизирует системные вызовы, вне зависимости от блокировки, с помощью оборачивания их во время исполнения. Эта обертка будет автоматически отделять P от треда M и позволять другим тредам запускать его. Давайте посмотрим на пример чтения файла:

```
func main() {
   buf := make([]byte, 0, 2)

   fd, _ := os.Open("number.txt")
   fd.Read(buf)
   fd.Close()

   println(string(buf)) // 42
}
```

Вот рабочий процесс открытия файла:



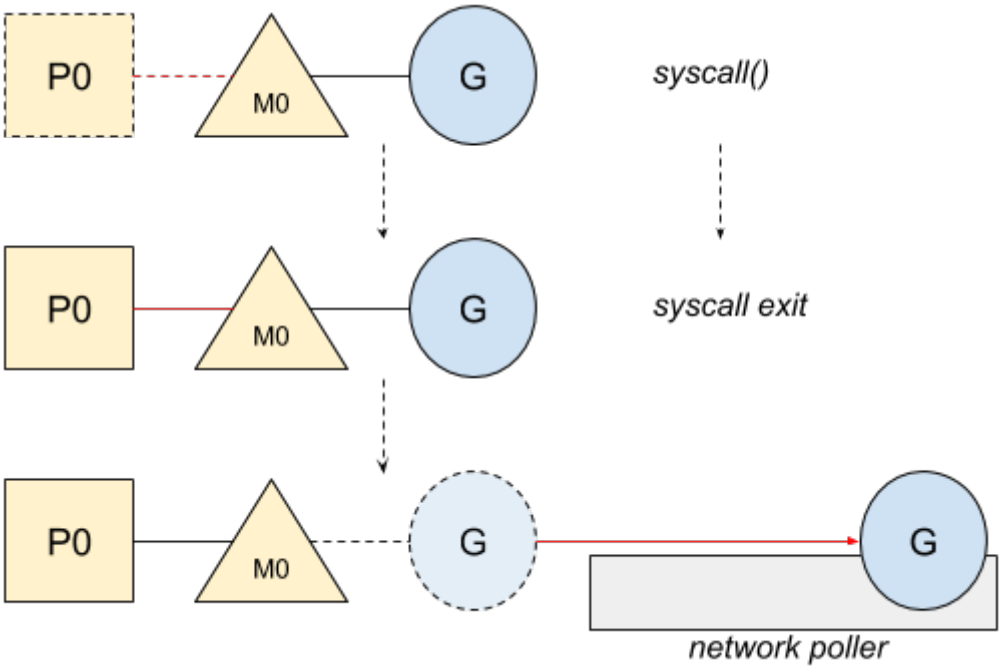
P0 в данный момент простаивает, и потенциально доступен. Затем, как только системный вызов завершен, Go применяет следующий набор правил пока одно из правил не будет удовлетворено:

- Попытатся завладеть тем же P, в нашем случае это P0, и вернутся к выполнению.
- Попытаться получить P из списка "холостых" и вернуться к выполнению
- Поместить горутину в общую очередь, а связанный с ним M вернуть обратно в "холостую" очередь.

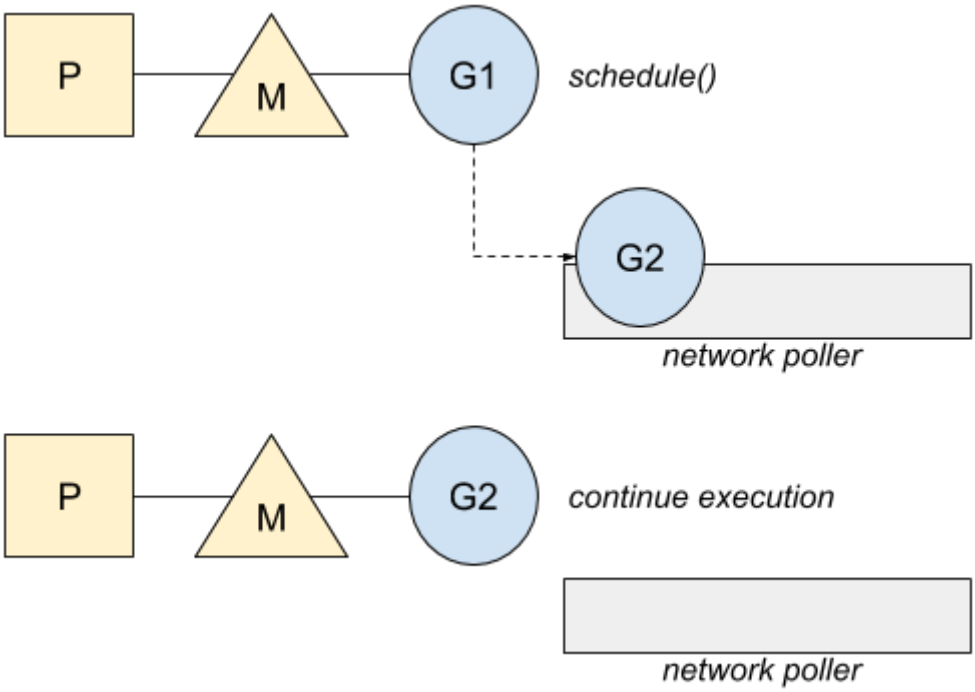
Однакой, Go, так же обрабатывает ситуации когда ресурсы еще не готовы, на случай не блокируемых I/O, например http вызовы. Тогда, первый системый вызов, который следуюет представленному выше рабочему процессу, упадет, так как нет готовых ресурсов, заставляет Go использовать сетевой опросник и остановливает горутину. Вот пример:

```
func main() {
   http.Get(`https://httpstat.us/200`)
}
```

Как первый системный вызов отработает и явно скажет, что ресуср не готов, горутина остановится до тех пор пока сетевой опросник не скажет, что ресурс готов. В этом случае тред M будет разблокирован:



Горутина заново запустится когда Go планировщик начнет искать работу. Планировщик затем будет передавать сетевому опроснику, что горутина ожидает запуска в случае успешного получения информации которая ожидается:



Если больше, чем одна горутина готова, то дополнительная будет отправлена в запускаемую глобальную очреедь и будет запущена планировщиком позже.

Ограничения в рамках потоков ОС

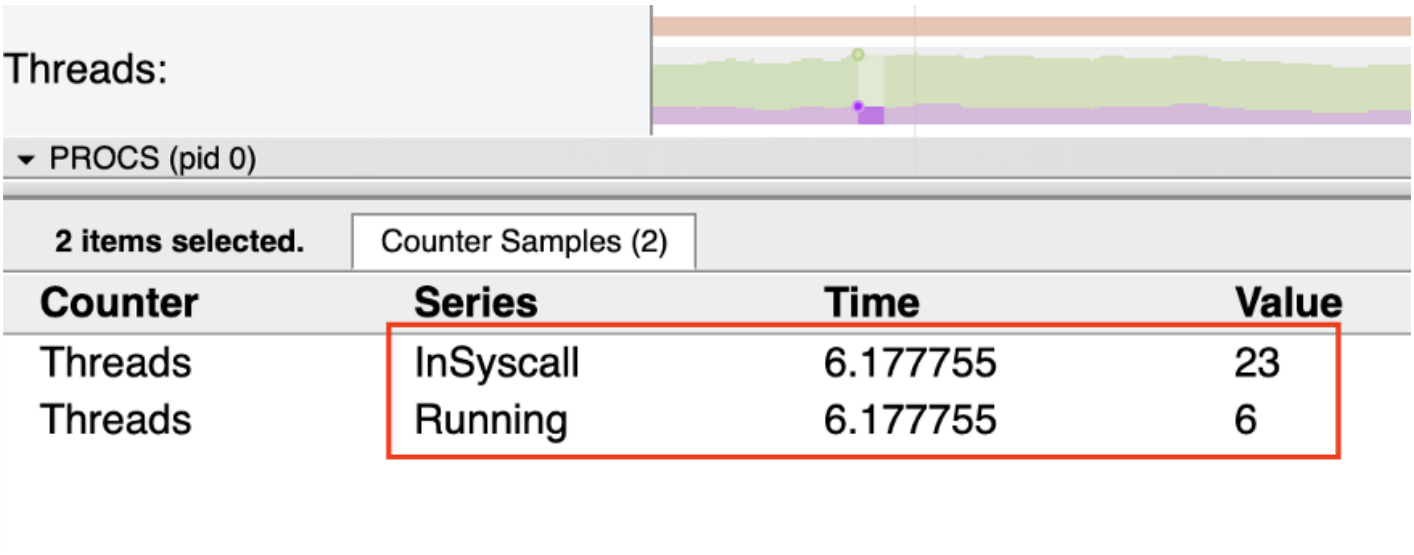
Когда системные вызовы используются, Go не ограничивает количество потоков ОС, которые могут быть заблокированны, как указано в коде:

```
GOMAXPROCS переменная ограничивает количество рабочих системных потоков, которые могут быть запущены пользвателем одновременно. Нет ограничений по количеству потоков которые могут быть заблокированны системными вызовами при выполнении от имени Go кода, отсюда, количество GOMAXPROCS ограниченно. Эта функция пакета GOMAXPROCS опрашивает и меняет количество.
```

Вот пример ситуации:

```
func main() {
   var wg sync.WaitGroup

   for i := 0;i < 100 ;i++  {
      wg.Add(1)

      go func() {
         http.Get(`https://httpstat.us/200?sleep=10000`)

         wg.Done()
      }()
   }

   wg.Wait()
}
```

Вот количество потоков созданных с помощью инструментов слежения:



Так как Go оптимизирован для использования потоков, он может быть многократно использован в то время, как горутины заблокированны. Это объясняет почему это число не совпадает с числом циклов.

Revision #2
Created 2022-02-13 19:50:19 UTC by gasick
Updated 2023-04-16 19:30:04 UTC by gasick