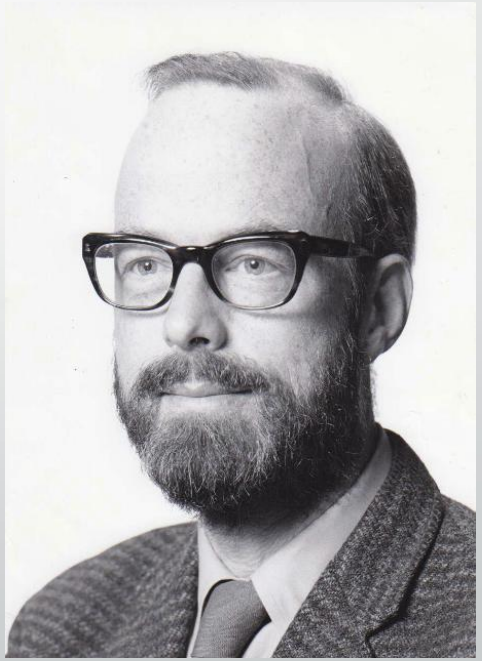
Concurrency in Go

Where did Go's concurrency model come from?

Communicating Sequential Processes (CSP)

- Seminal paper by C.A.R. Hoare (1978)
- Parallel composition of communicating sequential processes
- Communication synchronizes
- No sharing of memory



CSP

C.A.R. HOARE

SQUEAK

LUCA CARDELLI
ROB PIKE

NEWSQUEAK

ROB PIKE

GO

ROB PIKE
KEN THOMPSON
ROBERT GRIESEMER

Concurrency fundamentals in Go
GOROUTINE

- Growable stacks
  - Goroutines are very lightweight
  - Might have millions
- Multiplexed on few threads
  - The Go runtime schedules goroutines on OS threads
  - No full context switch
- No identity
- Same address space
  - Access to shared memory must be synchronized

A goroutine is an *independently executing function*

It is not a thread or a coroutine!

— Call a goroutine

```
func f(x int) {
    // do something
}
go f(4)
```

Concurrency fundamentals in Go
CHANNEL

- First-class object
  - *"ability to communicate the ability to communicate"*
- Senders don't have to know about receivers, and vice versa
  - 2nd generation CSP focuses on the channels over which messages are sent
  - Unlike actors
- *"Don't communicate by sharing memory; share memory by communicating"*
  - *Pass by value*

A channel is a *typed communication mechanism*

Sending and receiving

```
c := make(chan int)

go func() {          go func() {
  c <- 1               value <- c
}()                  }()
```

Concurrency fundamentals in Go

CHANNEL BUFFERING

Unbuffered channel

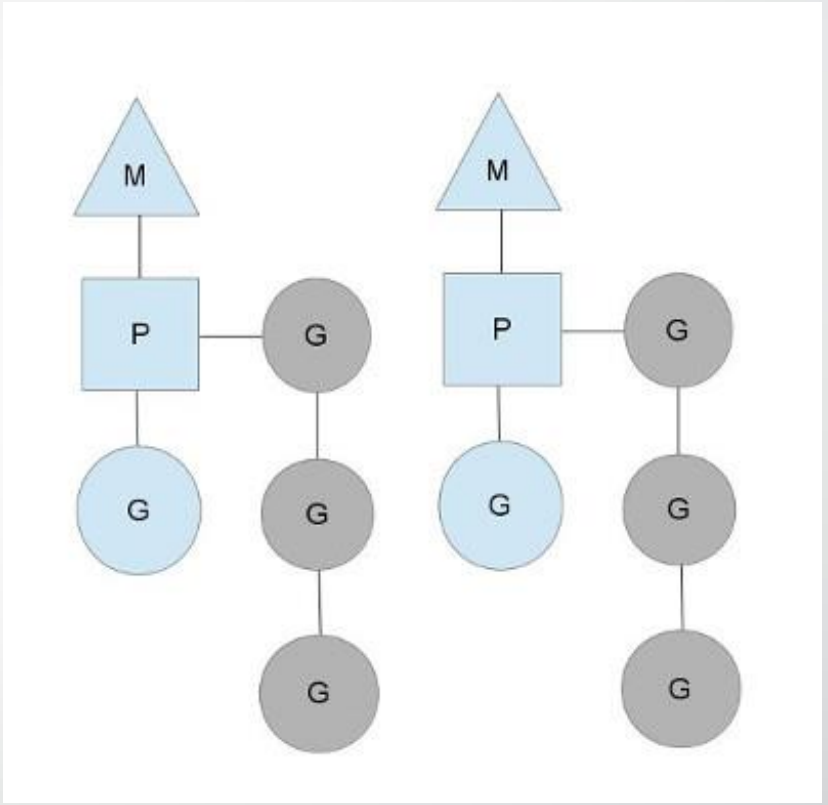
- Default option (2nd generation CSP feature)
- The sending and receiving goroutines communicates and synchronize in a single operation
  - A receiving goroutine waits for a value to be sent on the channel
  - A sending goroutine waits for a receiver to be ready

Buffered channel

- Buffering removes synchronization
- Queue of messages, with a maximum size
  - As long as there's space available, writing to a buffered channel completes immediately
  - If the channel is full, the send operation blocks its goroutine until space is made available by another goroutine's receive
  - Receives block when the buffer is empty

The Go scheduler

- Schedules goroutines on available resources
- Picks up a new goroutine when the current one…
  - Finishes
  - Makes a blocking system call
    e.g. reading a file
  - Makes a blocking Go runtime call
    e.g. reading from a channel
  - Invokes a new goroutine



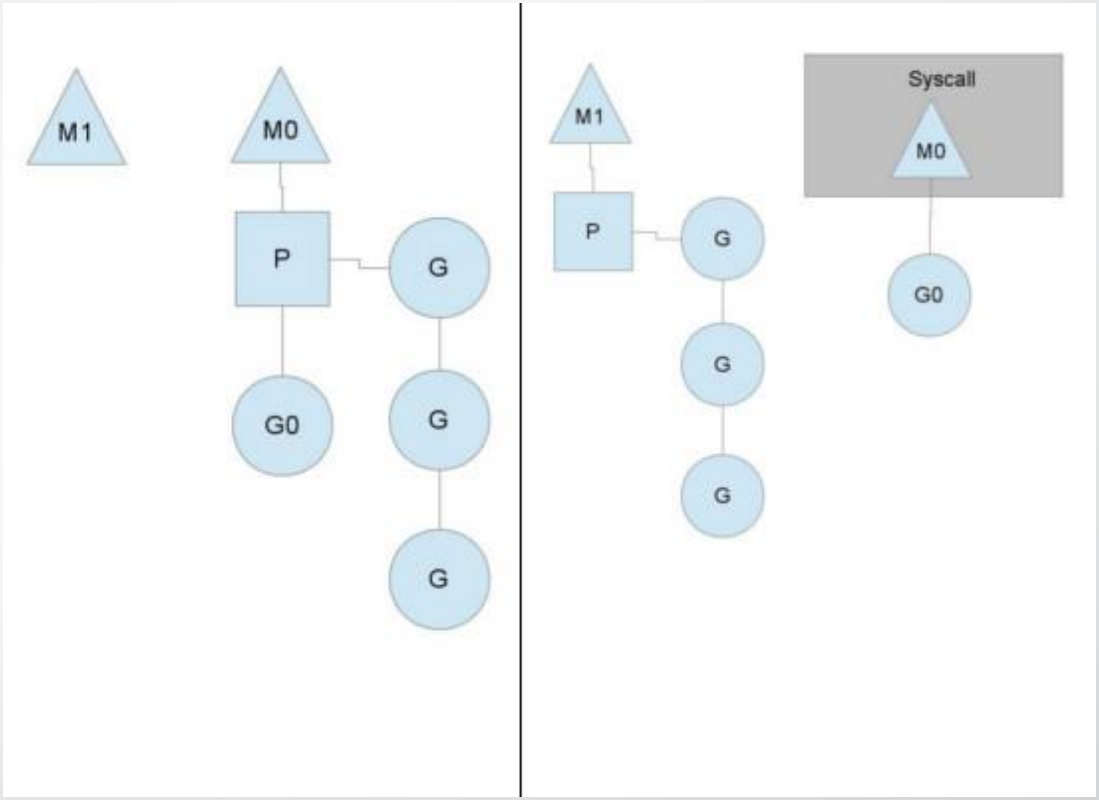
M OS thread

P Context

G Goroutine (with stack, instruction pointer…)

Goroutine blocked on a system call

- The OS thread processing the goroutine idles waiting for the system call to return
- The context that was scheduling goroutines on the blocked thread is moved to a new thread (or a new one is created if none are available)



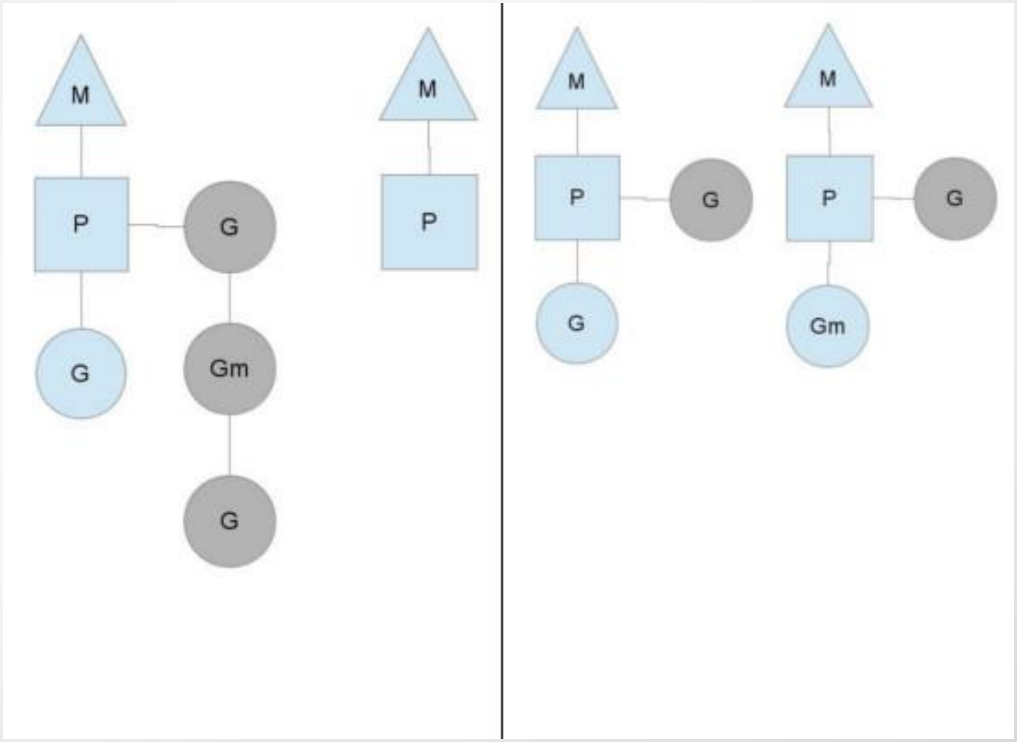
M OS thread

P Context

G Goroutine (with stack, instruction pointer…)

Stealing work

- If the amount of work on the contexts' runqueues is unbalanced, a context could run out of goroutines
- A context can take goroutines out of the global runqueue or from other contexts
- There is always work to do on each of the contexts



M OS thread

P Context

G Goroutine (with stack, instruction pointer…)