Module 6: Thread Scheduling, Concurrent Designs & Patterns in Go

MPCS 52060: Parallel Programming

University of Chicago

Thread Scheduling

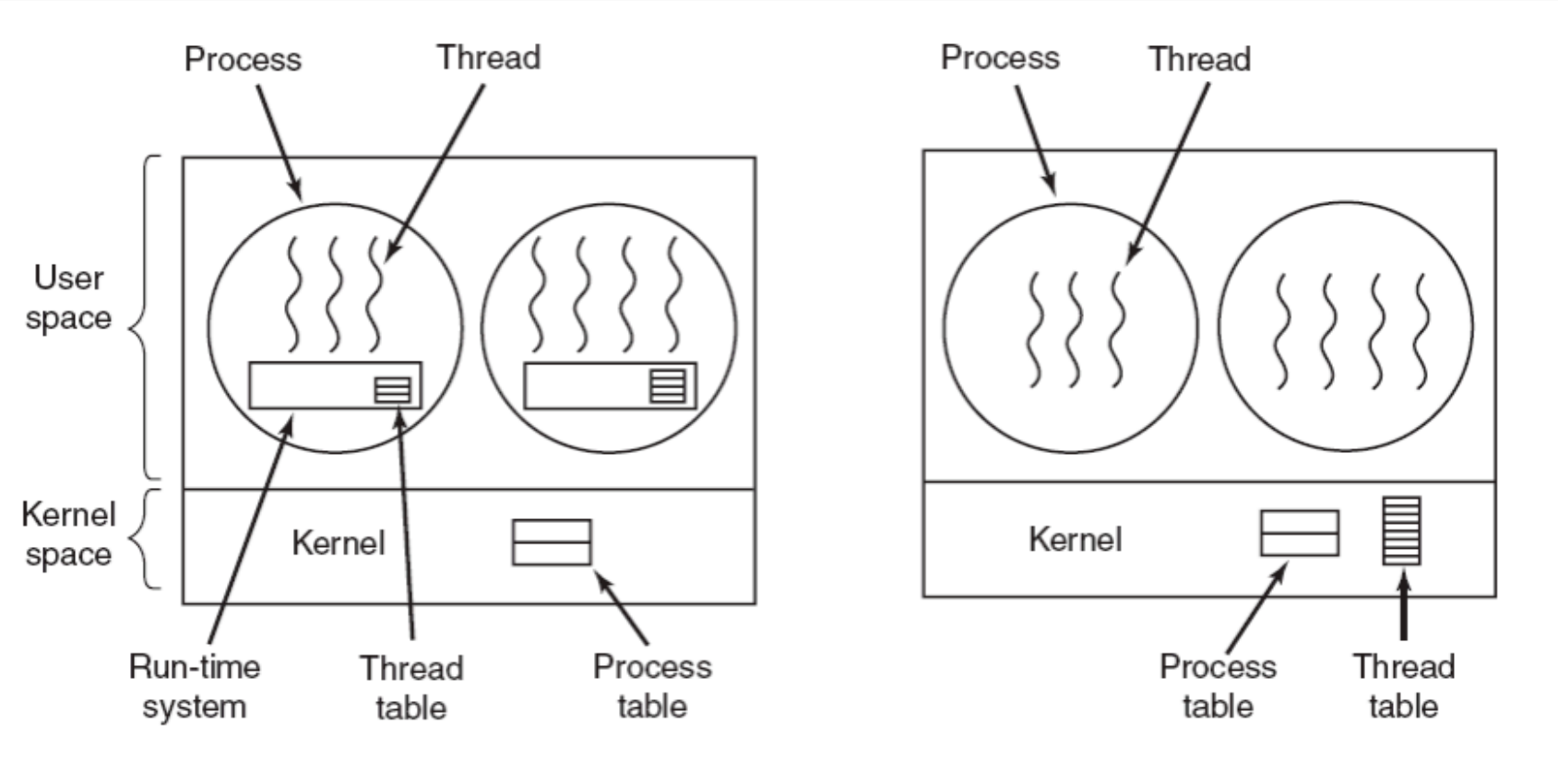
Types of Threads

There are several approaches to implementing threads in the OS, with varying degrees of functionality provided by the kernel and user space.

- User-space: System memory allocated to running applications.
- Kernel-space: System memory allocated to the kernel (i.e., the component that controls everything in the OS) and the operating system.

- User-level threads
  - Managed by a thread library or runtime system of the language (e.g., Golang)
  - Typically the OS has no knowledge of these threads.
- Kernel threads
  - Provided and managed by the OS
  - Most OSes support threads at the kernel level. .

User-level Threads vs. Kernel Threads



[1]

[1] Cite: Tanenbaum, Modern Operating Systems 3e,

Pros/Cons: User-level Threads vs. Kernel Threads

- User-level threads
  - **Pros**: Speed, since the switching of threads can be done without a system call to the operating system (OS), thread creation is fast
  - **Cons**: If an OS only provides user-level threads then,
    - OS cannot map process threads to multiple CPUs.
    - OS cannot suspend a process if a process thread is performing an I/O operation.
- Kernel threads
  - **Pros**: The disadvantages of user-level threads can be avoided since the OS knows of their existence and can react appropriately (i.e., schedule another thread if one blocks)
  - **Cons**: Slow, kernel does creation, scheduling, context-switching, etc.

Multithreading models

A thread library must map user threads to kernel threads in order to run code on the logical cores.

There are different mappings that exist that an operating system can support, each with its own tradeoffs:

- Many-to-One (N:1): many user threads map to one kernel thread.
- One-to-One (1:1): one user thread maps to a one kernel thread.
- Many-to-Many (N:M): many user threads map to many kernel threads.

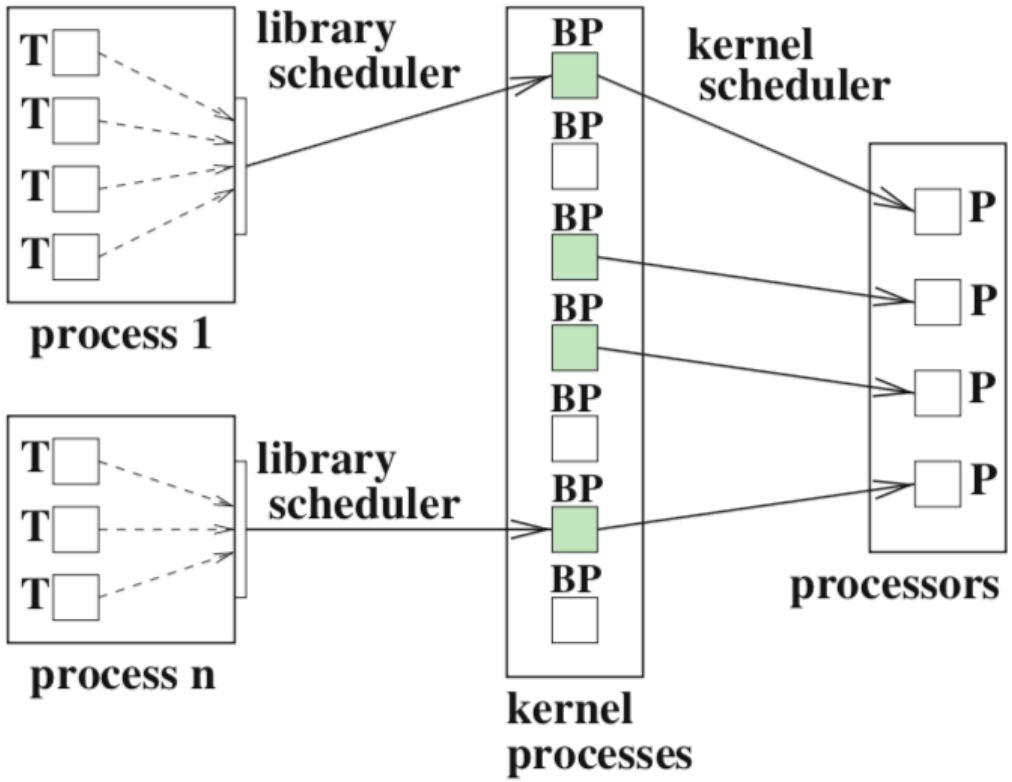
Lets take a look at each.

Many-to-One (N:1)

- This is also known as *user-level threading*
- All user-level threads of a user process are mapped to one kernel thread of the operating system.
- The kernel sees a single process.
- Thread library scheduler maps determines which user-level thread will be executing for the process (only one thread at a time).
- OS is in charge of mapping a process to a CPU.

N:1 Mapping Illustration

Fig. 3.15 Illustration of a N:1 mapping for thread management without kernel threads. The scheduler of the thread library selects the next thread T of the user process for execution. Each user process is assigned to exactly one process BP of the operating system. The scheduler of the operating system selects the processes to be executed at a certain time and maps them to the execution resources P.



[2]

[2] Source: Parallel Programming, Thomas Rauber, Gudula Runger

Pros/Cons of N:1

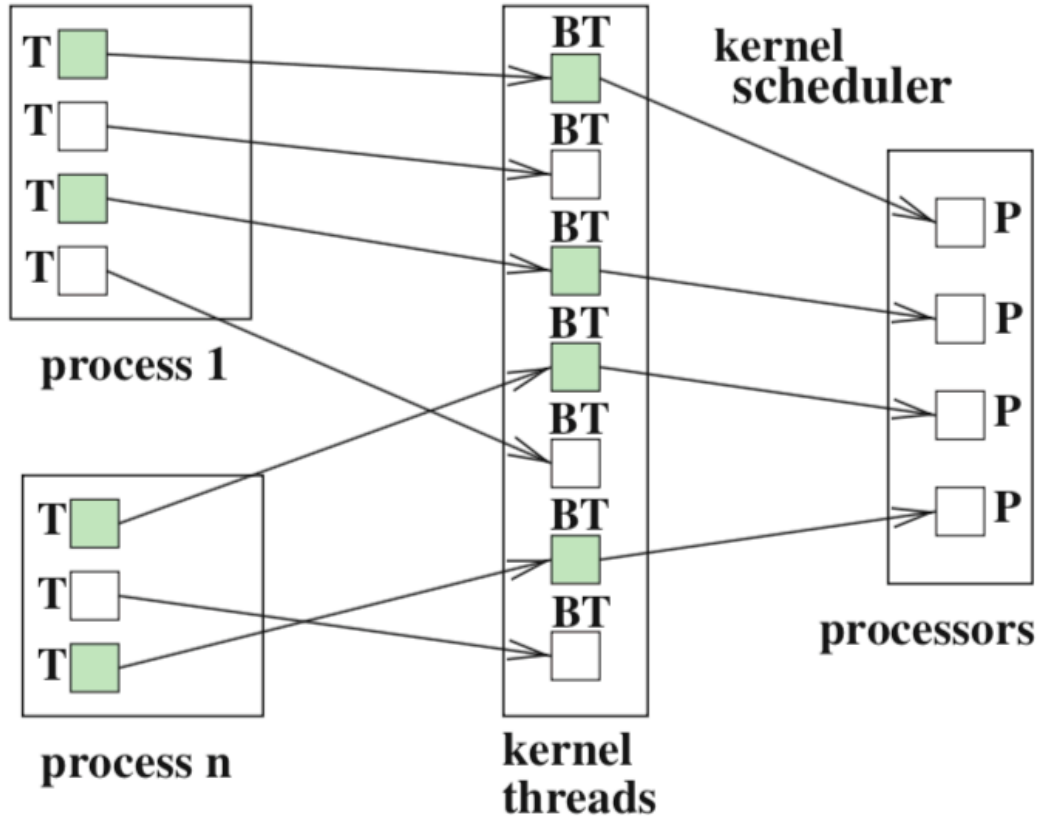
- Pros
  - Creation and context switching is fast because it requires no system calls (i.e., communicating with the kernel, which is expensive).
  - Portability: few system dependencies.
- Cons
  - There is no parallel execution of threads. There's only a single kernel entity backing the N threads; therefore, this model cannot utilize multiple processors.

One-to-One: (1:1)

- This is also known as *kernel-level threading*
- Generates a kernel thread for each user-level thread.
- The scheduler of the operating system selects which kernel threads are executed at which point in time.
- OS is in charge of mapping threads to a CPU(s)
- No, need for a library scheduler since since each user-level threads is assigned to exactly one kernel thread.

1:1 Mapping Illustration

Fig. 3.16 Illustration of a 1:1 mapping for thread management with kernel threads. Each user-level thread T is assigned to one kernel thread BT. The kernel threads BT are mapped to execution resources P by the scheduler of the operating system.



[3]

[3]Source: Parallel Programming, Thomas Rauber, Gudula Runger

Pros/Cons of 1:1

- Pros
  - Allows for more concurrency. When one thread blocks, other threads can be scheduled.
- Cons
  - Depending on the operating system this model can be expensive when it comes to creation, context switching, and deletion of threads.
  - All thread operations involve the kernel

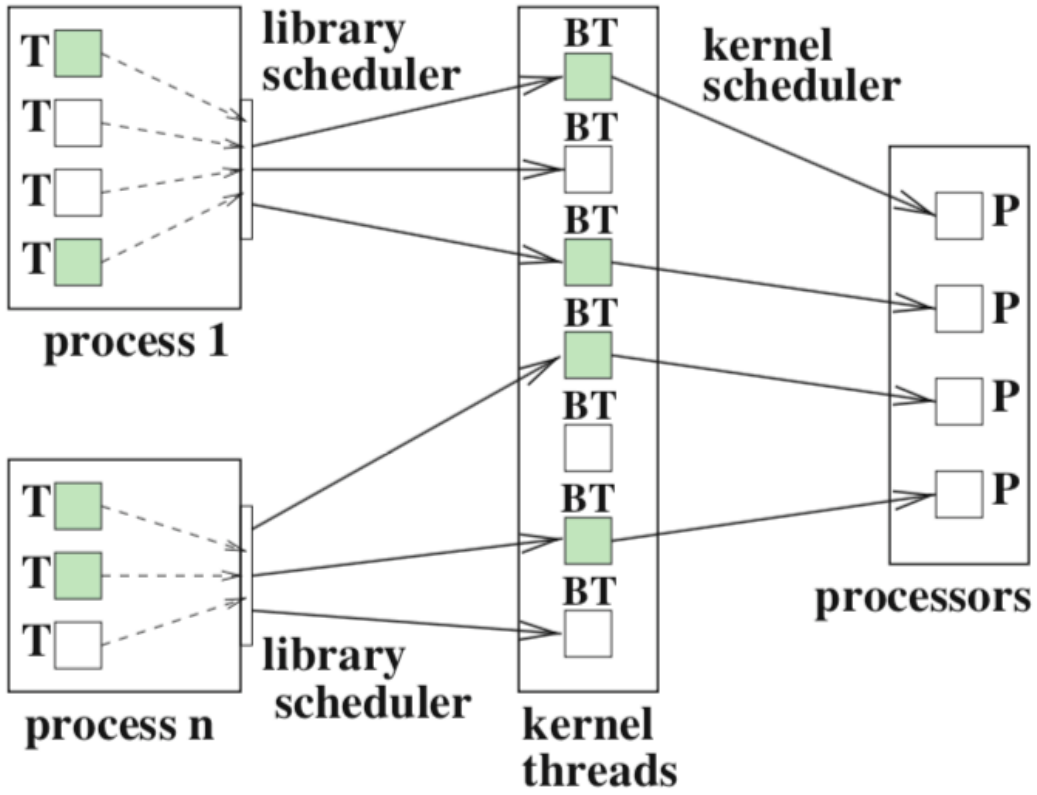
Many-to-Many: N:M

N = User threads, M = Kernel Threads

- This is also known as *Hybrid threading*
- Two-level scheduling where the thread library scheduler assigns user-level threads to a given set of kernel threads.
- At any point in time, a user threads can be mapped to a different kernel thread (no fixed mapping)
- N is greater than M.
- Correspondingly, a kernel thread can execute different user threads at any point in time.
- **The Go runtime uses this mapping to manage goroutines**.

N:M Mapping Illustration

Fig. 3.17 Illustration of a N:M mapping for thread management with kernel threads using a two-level scheduling. User-level threads T of different processes are assigned to a set of kernel threads BT (N:M mapping) which are then mapped by the scheduler of the operating system to execution resources P.



[4]

[4]Source: Parallel Programming, Thomas Rauber, Gudula Runger

Pros/Cons of N:M

- Pros
  - This model is flexible because the OS creates kernel threads for physical concurrency and the Applications creates user threads for application concurrency.
- Cons
  - It's quite a complex model to implement. Most systems use 1:1 mapping.

Coroutines

As I have mentioned before, Goroutines are not technically threads by really based off the concept of a coroutine:

- A unit of execution even lighter in weight than a thread.
- They are like user-level threads but have little to no user-space support for their scheduling and execution.
- They are cooperatively scheduled, requiring an explicitly yield to move to another coroutine (e.g., runtime schedule call, i/o call, etc.)
- They enable differing programming paradigms and I/O models such as CSP, which we will talk about next.

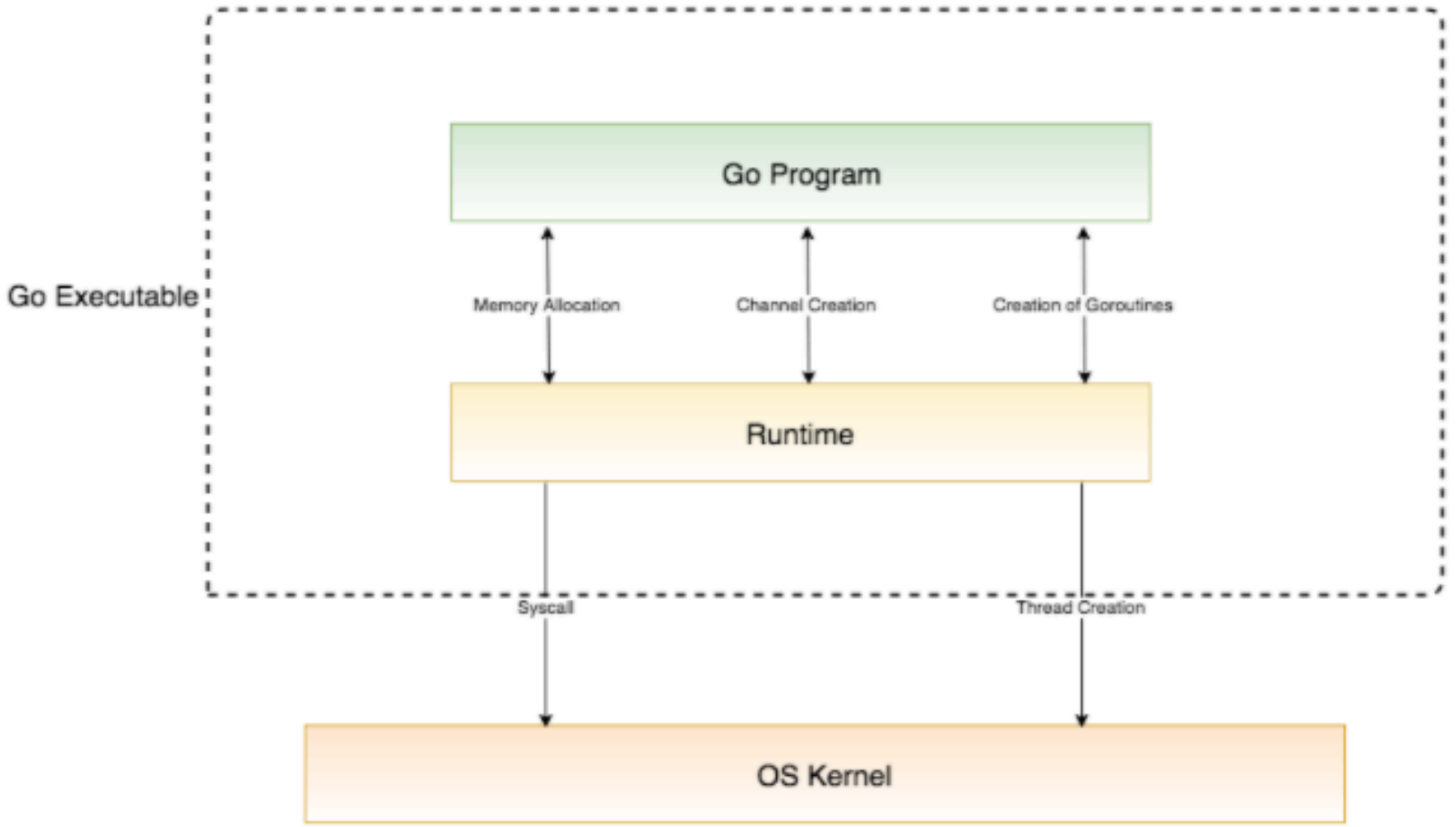
Goroutines as Coroutines

Having Goroutines be coroutines have a few benefits over using kernel threads:

- Memory consumption: Threads require 1MB of memory vs 2Kb memory to store the stack information.
- Switch Cost: When a context switch occurs, threads have to save a large amount of state information (general purpose registers, PC (Program Counter), SP (Stack Pointer), segment registers etc), whereas gorountines usually have to save just the program counter and stack pointer, and a few registers.

Goroutines as Coroutines (cont.)

- Setup and Teardown cost: Threads need to request resources from the kernel (expensive), whereas Goroutines are created and destroyed by the runtime (cheap).



[5] Cite: Analysis of the Go runtime scheduler, Deshpande, Sponsler, Weiss

Scheduling of Goroutines

Since goroutines are scheduled *cooperatively*, goroutines yield the control periodically when they are idle or logically blocked in order to run other Goroutines concurrently. This switch is is done at well defined points:

- Channels send and receive operations, if those operations would block.
- The Go statement, although there is no guarantee that the new Goroutine will be scheduled immediately.
- Blocking syscalls like file and network operations.
- After being stopped for a garbage collection cycle.

Scheduling of Goroutines Internally

Go uses three entities to explain the scheduler,

- Processor (P)
- Kernel Thread (M)
- Goroutines (G)

Go application the number of threads available for Goroutines to run is equal to the *GOMAXPROCS*:

- `func GOMAXPROCS(n int) int`: sets the maximum number of CPUs that can be executing simultaneously and returns the previous setting
- If $n < 1$, it does not change the current setting
- Defaults to the number of logical cores for the system (i.e., `func NumCPU() int`)

Scheduling of Goroutines Internally (cont.)

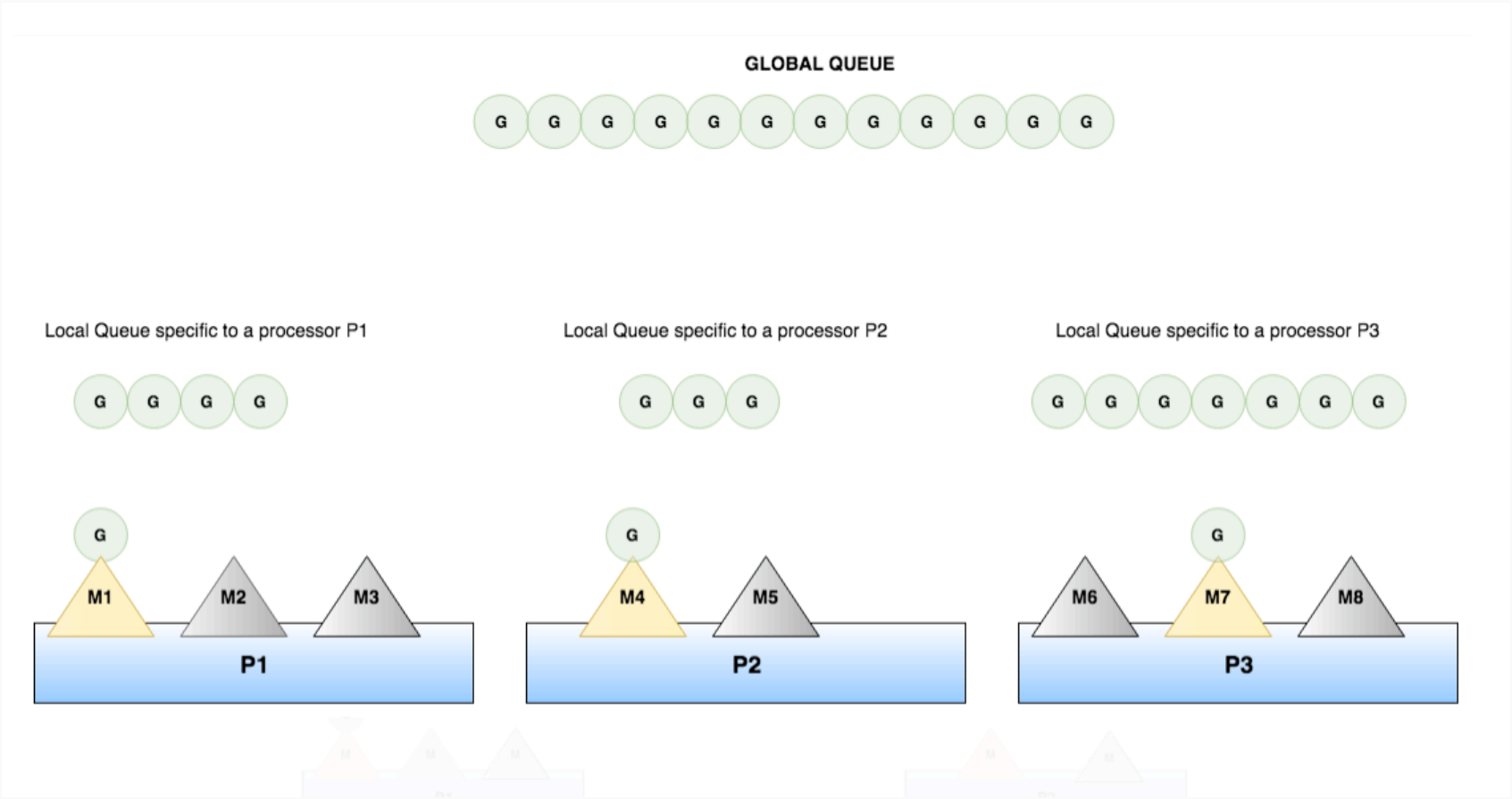
Golang uses an $M : N$ scheduler which means that M goroutines need to be scheduled on N Kernel threads that runs at most GOMAXPROCS number of processors ($N <$ GOMAXPROCS)

How it works:

"Every P has a local Goroutine queue. There's also a global Goroutine queue which contains runnable Goroutines. Each M should be assigned to a P. Ps may have no Ms if they are blocked or in a system call. At any time, there are at most GOMAXPROCS number of P and only one M can run per P. More Ms can be created by the scheduler if required." [6]

[6] Cite: https://medium.com/@riteeksrivastava/a-complete-journey-with-goroutines-8472630c7f5c

Scheduling of Goroutines Internally (cont.)

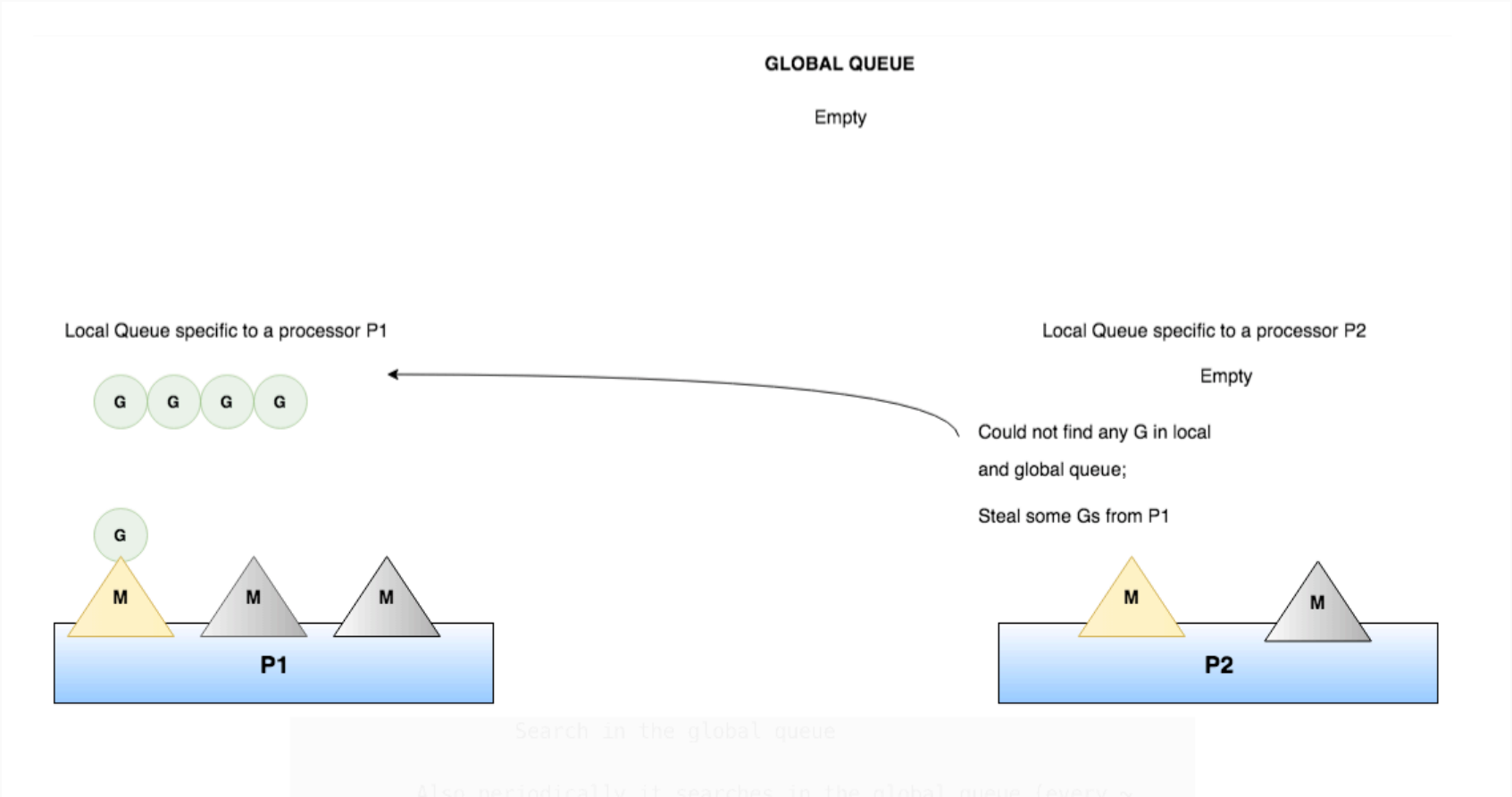


7

[7] Cite: https://medium.com/@riteeksrivastava/a-complete-journey-with-goroutines-8472630c7f5c

Scheduling of Goroutines Internally (cont.)

Each round of scheduling, the scheduler finds a runnable Goroutine and executes. If a Processor's queue is empty then it will try to *steal* Goroutines from other processor local queues.



[8]

[8] Cite: https://medium.com/@riteeksrivastava/a-complete-journey-with-goroutines-8472630c7f5c

Go’s Concurrency Model

Go's Concurrency Model

The concurrency model GO when it comes to sharing resources between goroutines is "Do not communicate by sharing memory; instead, share memory by communicating".

To do this, Go uses a powerful notion of channels:

- A channel in Go provides a connection between two goroutines, allowing them to communicate.

```
// Declaring and initializing.
var c chan int
c = make(chan int)
// or
c := make(chan int)
```

Channel Examples

Demos- See examples inside the upstream repository:

- m6/discussion/simple/simple.go
- m6/discussion/channel-buffering/channel-buffering.go
- m6/discussion/channel-directions/channel-directions.go
- m6/discussion/channel-synchronization/channel-synchronization.go
- m6/discussion/closing-channels/channel-synchronization.go

Go Patterns

Generator

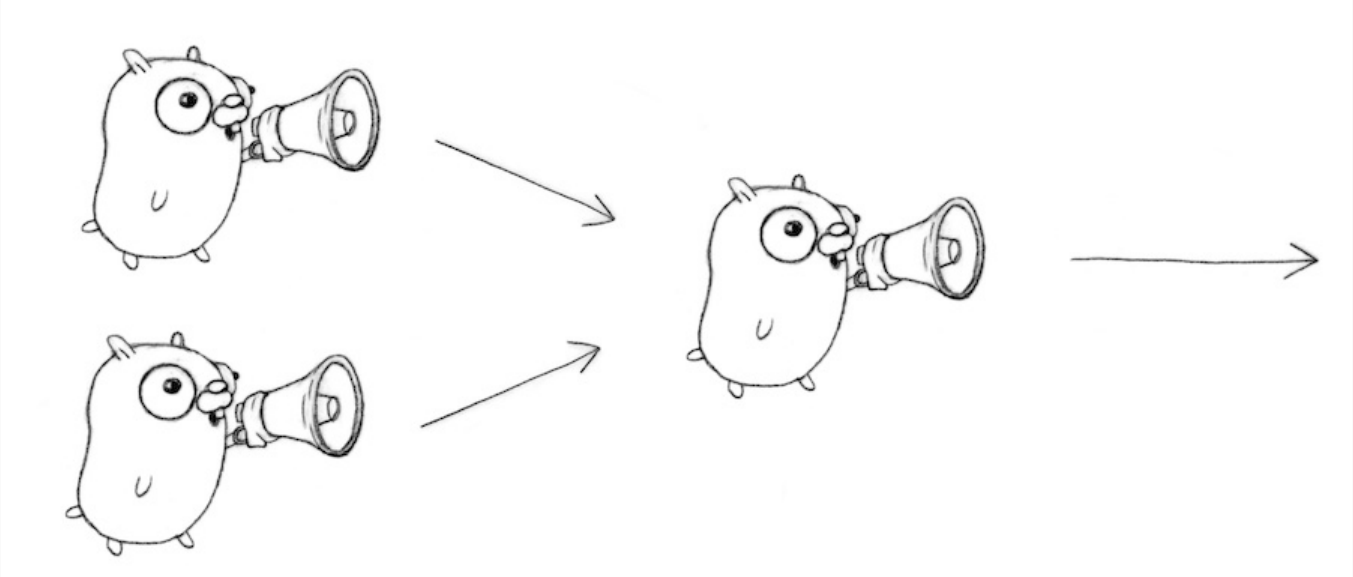
A generator is function that returns a channel

- As with other primitive types (integers, strings, etc.), channels are also first class values.
- Generators allow us to have more instances of some service.

Demo: See example in m6/discussion/generator/generator.go

Multiplexing

These programs make Bob and Sally count in lockstep. We can instead use a fan-in function to let whosoever is ready talk.



Demo: See example in m6/discussion/fanin/fanin.go

Select

The select statement lets you wait on multiple channel operations.

It's like a switch, but each case is a communication:

- When a select statement is reached, all cases are evaluated.
- The select blocks until one communication can proceed, which then performs the code inside its case block
- If multiple channels receive a value then select chooses pseudo-randomly a case to execute.
- A default clause, if present, executes immediately if no channel is ready.

```
select {
    case v1 := <-c1:
        fmt.Printf("received %v from c1\n", v1)
    case c3 <- 23:
        fmt.Printf("sent %v to c3\n", 23)
    default:
        fmt.Printf("no one was ready to communicate\n") }
}
```

Signaling Completion/Timeout

Timeouts and the closing of a channels are two ways to signal to a goroutine that a task has completed successfully or time has ran out:

- Timeouts: m6/discussion/timeouts/timeouts.go
- Closing channels: m6/discussion/closing-channels

More Patterns...

More channel patterns can be found here:

- confinement: m6/discussion/patterns/confinement
- error-handling: m6/discussion/patterns/error-handling
- fan-out-fan-int: m6/discussion/patterns/fan-out-fan-in
- pipelines: m6/discussion/patterns/pipelines
- preventing-leaks: m6/discussion/patterns/preventing-goroutine-leaks