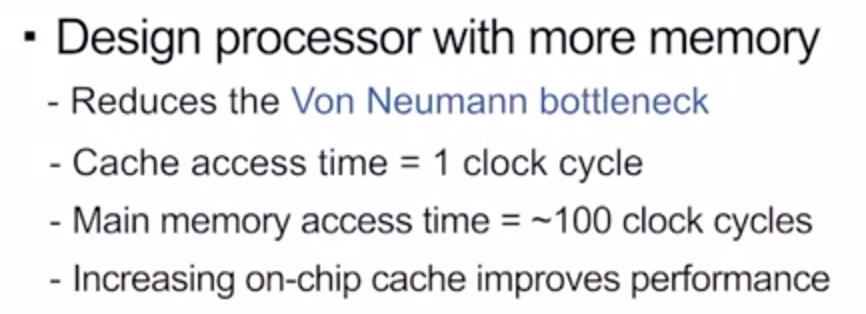
**Course 3 Concurrency**

# Week 1

## M1.1.1-3v3 Parallel Execution

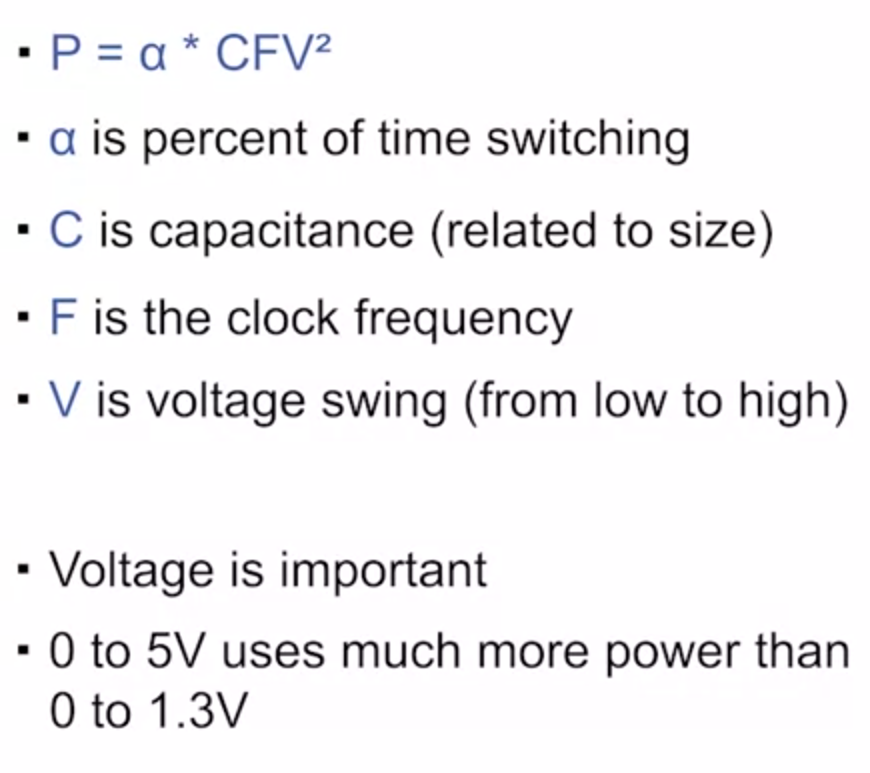
* Two Programs execute in parallel if they execute at exactly the same time
* Generally, one core runs one instruction at a time
* In order to get parallel execution, you need replicated hardware

## Topic 1.2 Von Neumann Bottleneck



* Moore’s Law is fizzling out

## Topic 1.3 Power Wall

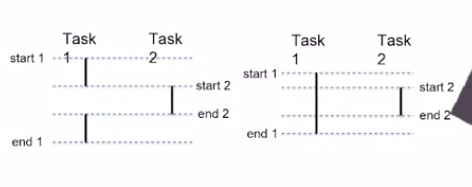
* Transistors consume power when they switch
* More power consumed – increased heat 🡪 ECE 304
* 

Dennard Scaling:

* Voltage should scale with transistor size
* Voltage can’t go too low though
  + Can’t drop below threshold voltage
  + Must be robust to noise

## M1.2.1-1v3 – Concurrent vs Parallel

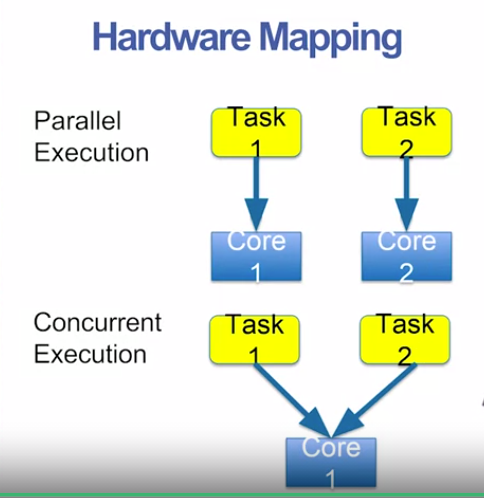
Concurrent Execution

* Not necessarily the same as parallel execution
* **Concurrent:** start and end times overlap
  + Times overlap, but they’re not actually executing at exactly the same time
* **Parallel:** execute at exactly the same time
* Concurrent (left), parallel (right)
* Concurrent tasks may be executed on the same hardware, but they don’t have to be

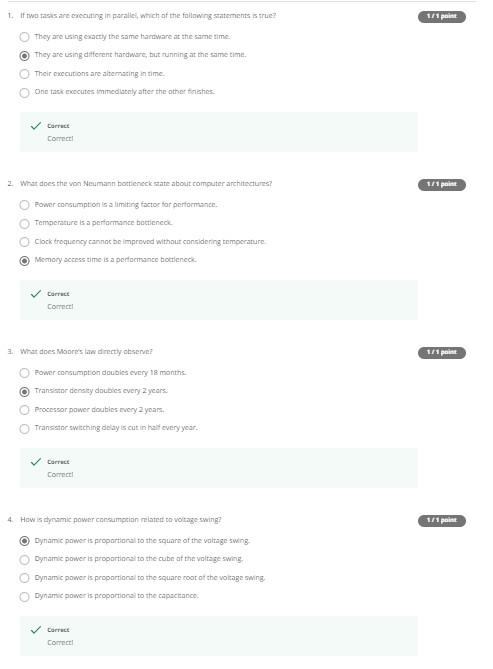
NOTE: Programmer determines which task **can** be executed in parallel, not **which will** be executed in parallel

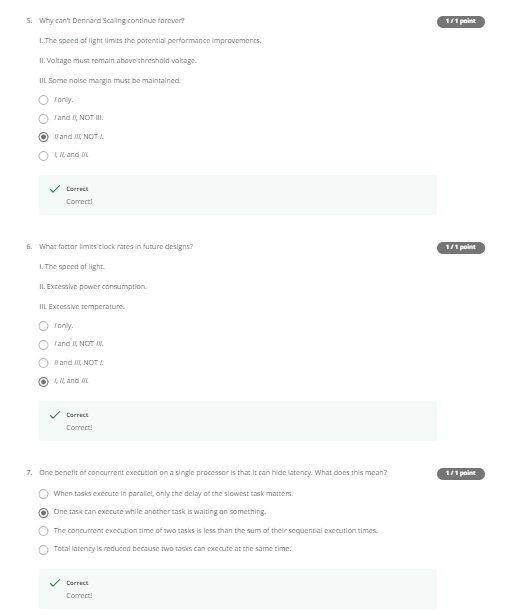
Concurrency improves performance without parallelism

* Tasks must periodically wait for event
  + Think talking to some external memory, send data on the network, io activities/accesses



## Module 1 Quiz





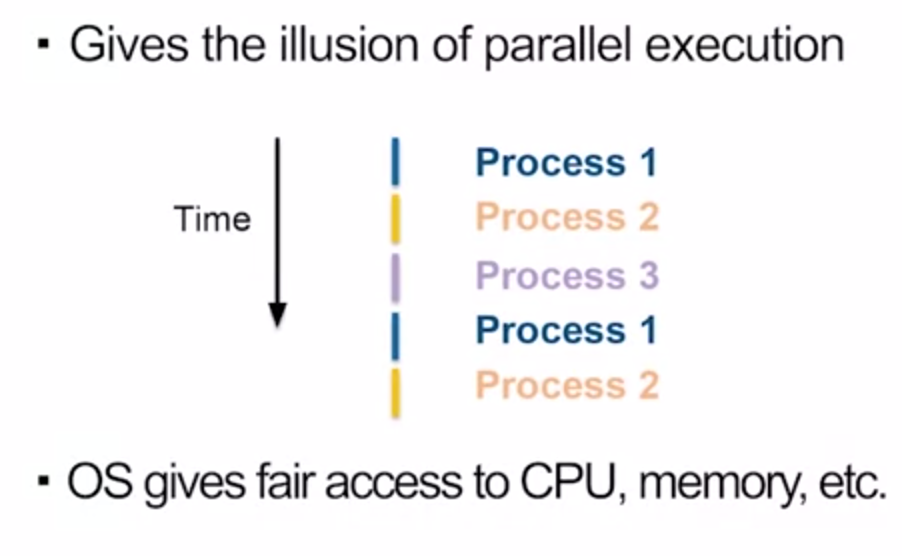
# Week 2: Concurrency Basics

## Topic 2.1 Processes

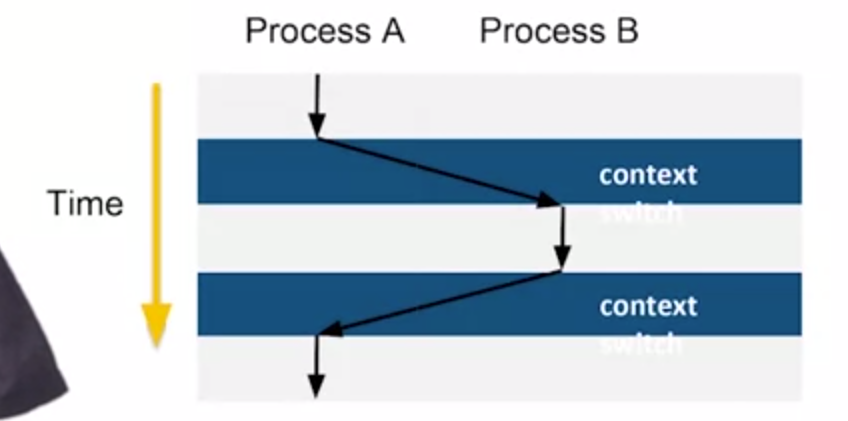
Processes:

* Instance of a running program
* Has memory
  + Virtual address space
  + Code, stack, heap, shared libraries
* Registers
  + Program counter, data regs, stack ptr

## Topic 2.2 Scheduling



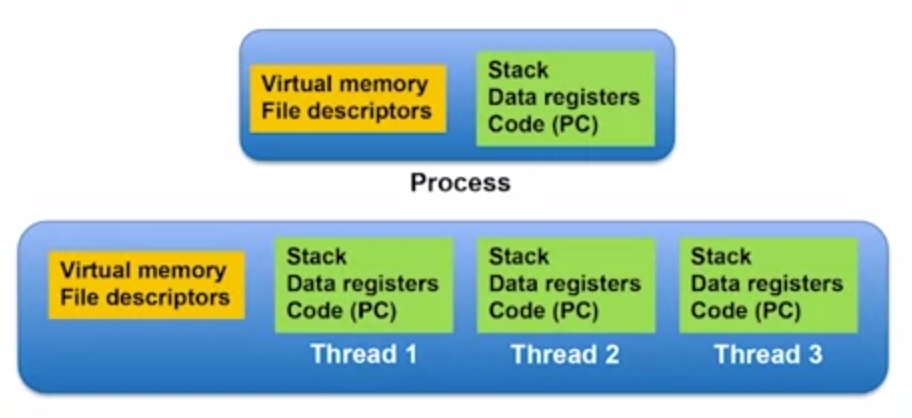
When it moves to another process, it’s called **context switch**



Before a context switch, you need to save the state of the current process

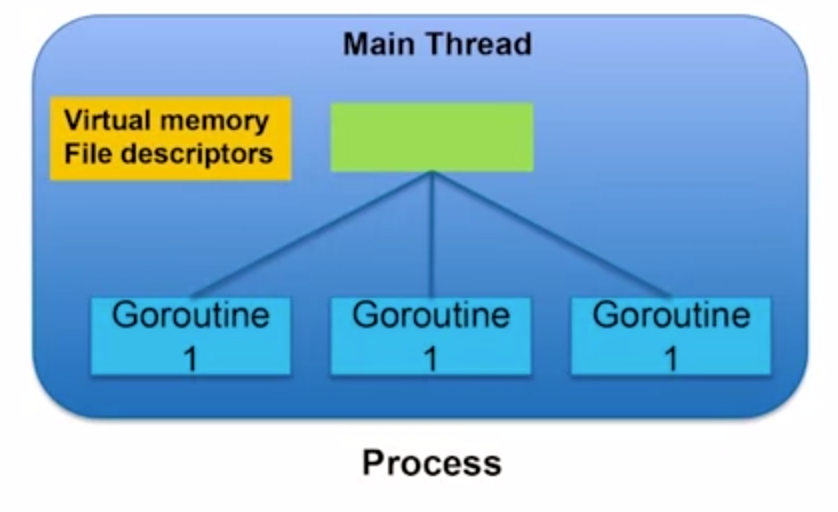
## Topic 2.3 Threads and Goroutines

* Context switch can be slow (move something into memory, pull it back out later)
* Threads can share some context
* Many threads can exist in one process

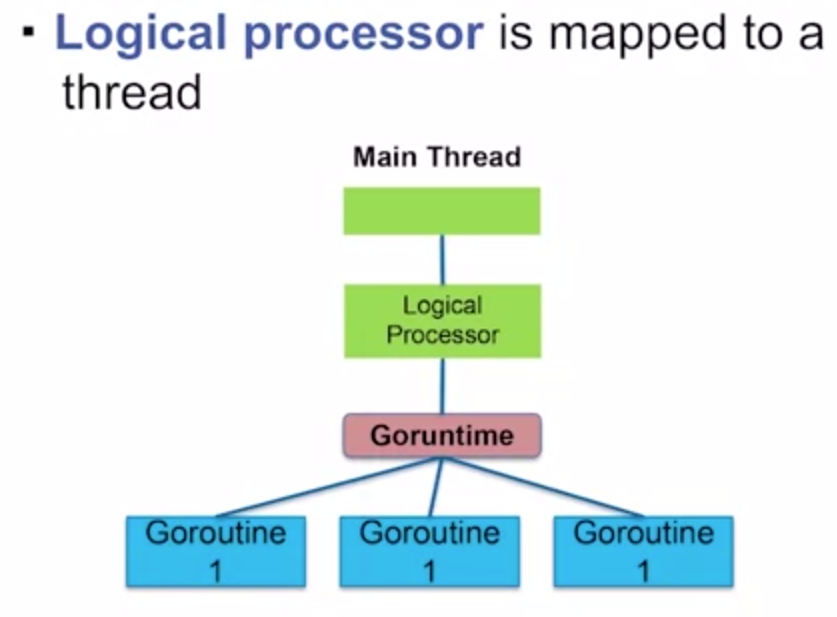


Shared quantities in yellow box

**Goroutines**

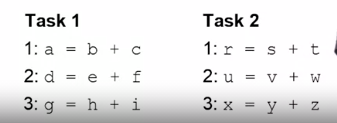
* Like a thread in Go
* Many goroutines execute within a single OS thread
* 
* OS schedules the main thread, GO handles the scheduling for the goroutines

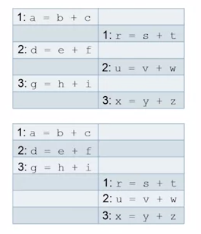
Switching is handled by the GO runtime scheduler

* Schedules goroutines inside an OS thread
* Like a little OS inside a single OS thread
* 

## Topic 2.4 Interleaving

* Concurrent code can make debugging far more challenging
* Order of execution between concurrent tasks is unknown
* Interleavings between tasks is unknown



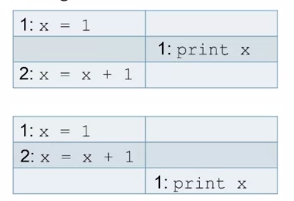


* Interleavings occurring at the machine code level
* Ordering is non-deterministic

## Topic 2.5 Race Conditions

Problem where the outcome of the program depends on the interleaving

* You almost always want deterministic outcomes



Races occur due to communication

Communication Between Tasks:

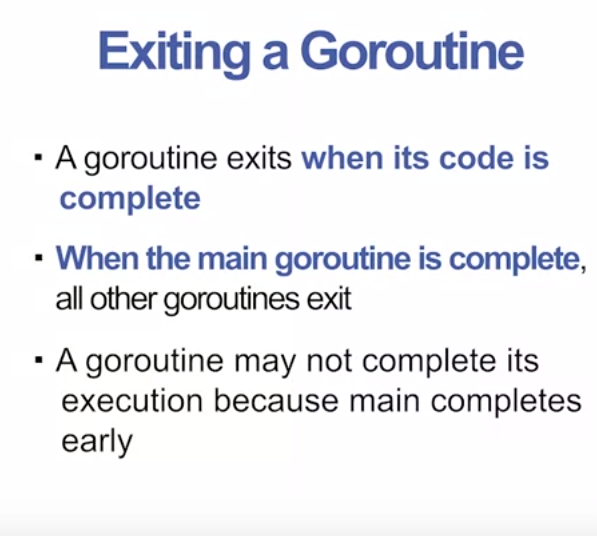
* Threads are largely independent but not completely independent
* Threads share common data

## Module 2 Quiz

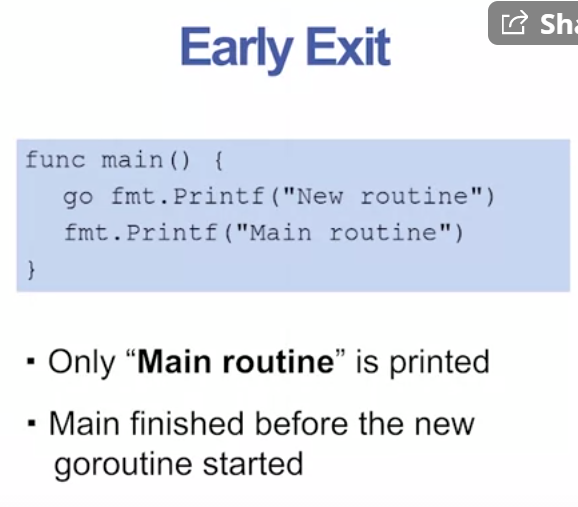
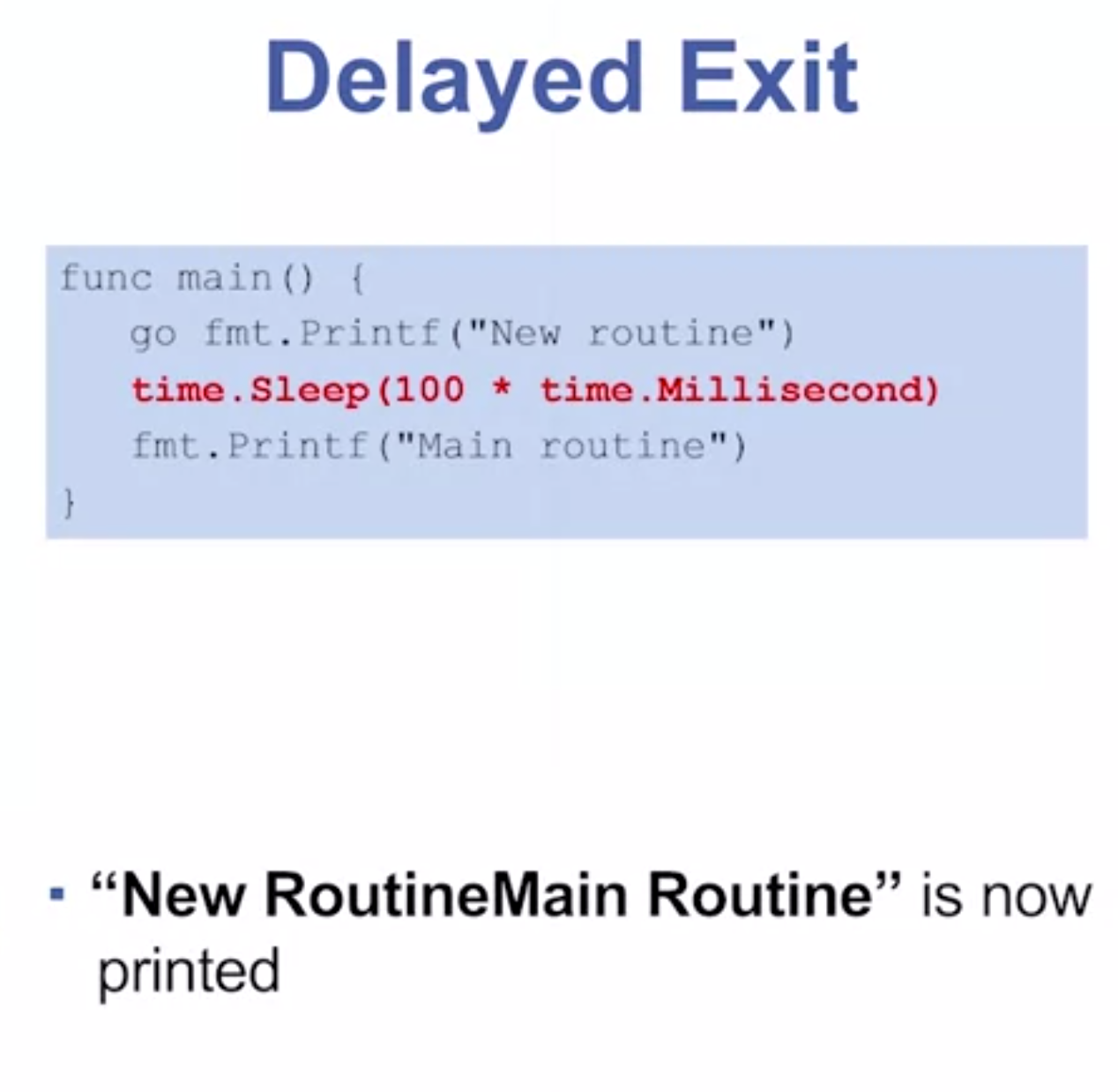
# Week Three: Threads in GO

## Topic 3.1 Goroutines

Your main is technically a goroutine

* Create other goroutines using the **go** keyword
* 
* 

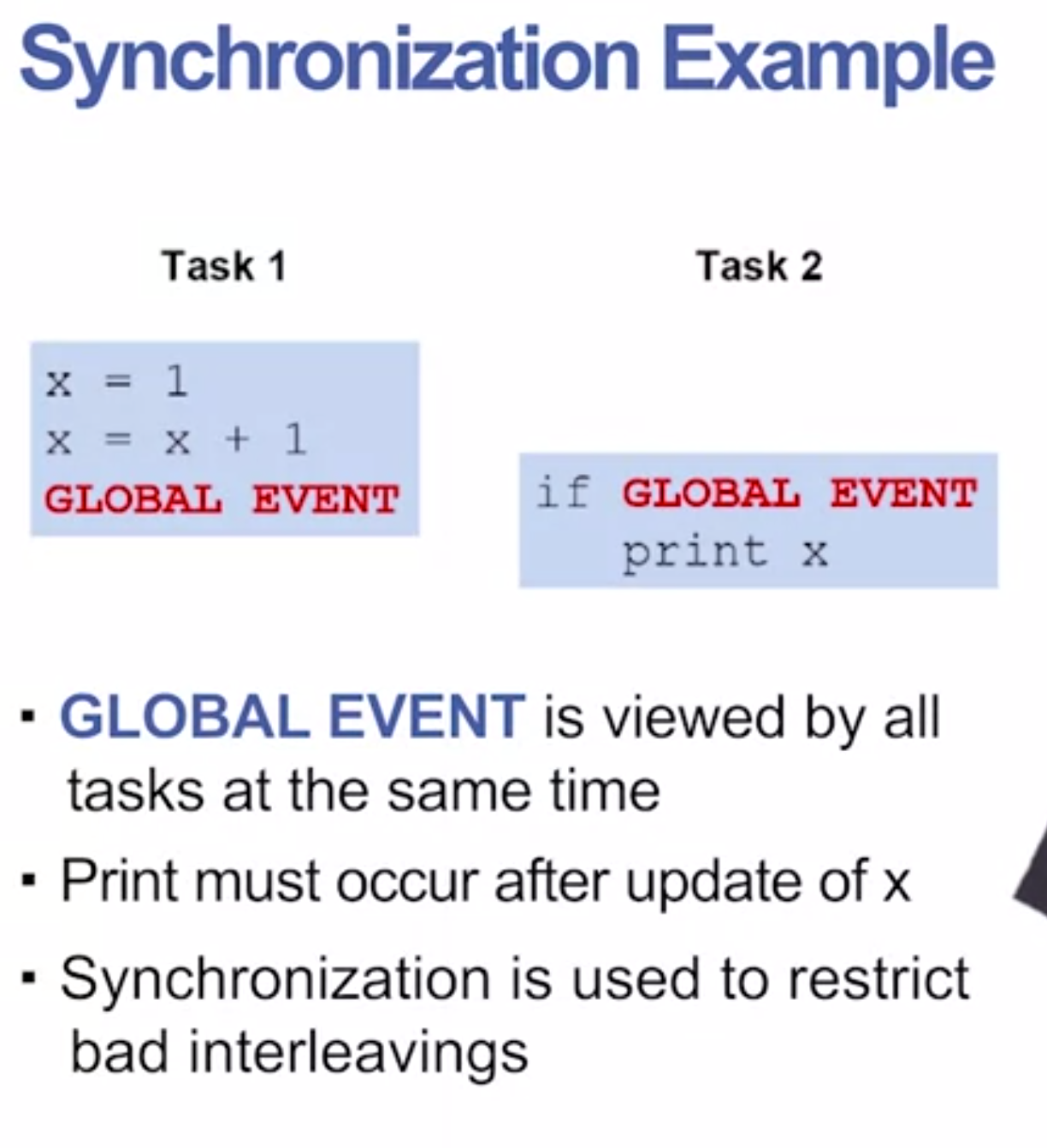
## Topic 3.2 Exiting Goroutines

 - Yucky

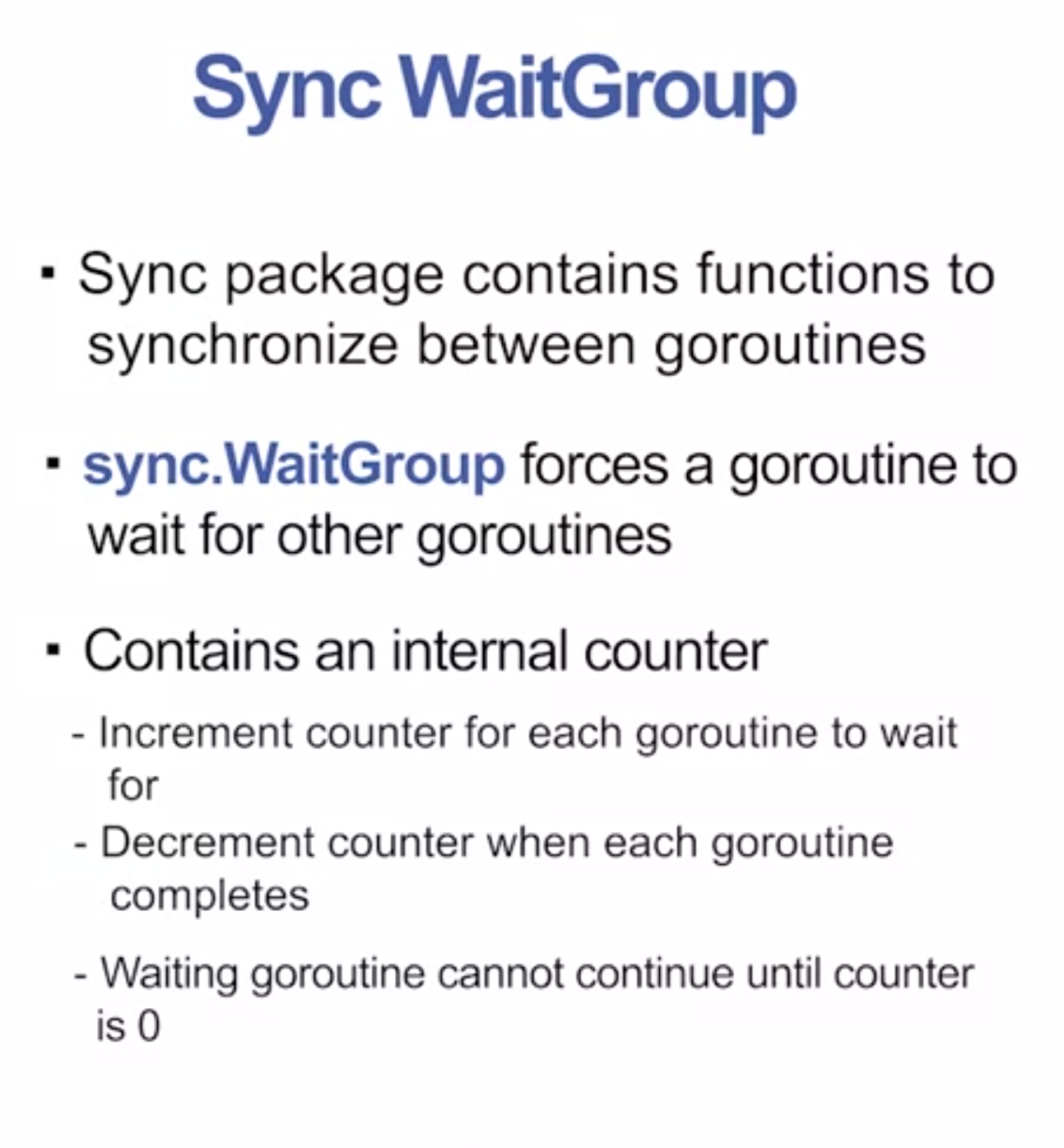
- Making timing assumptions is wrong

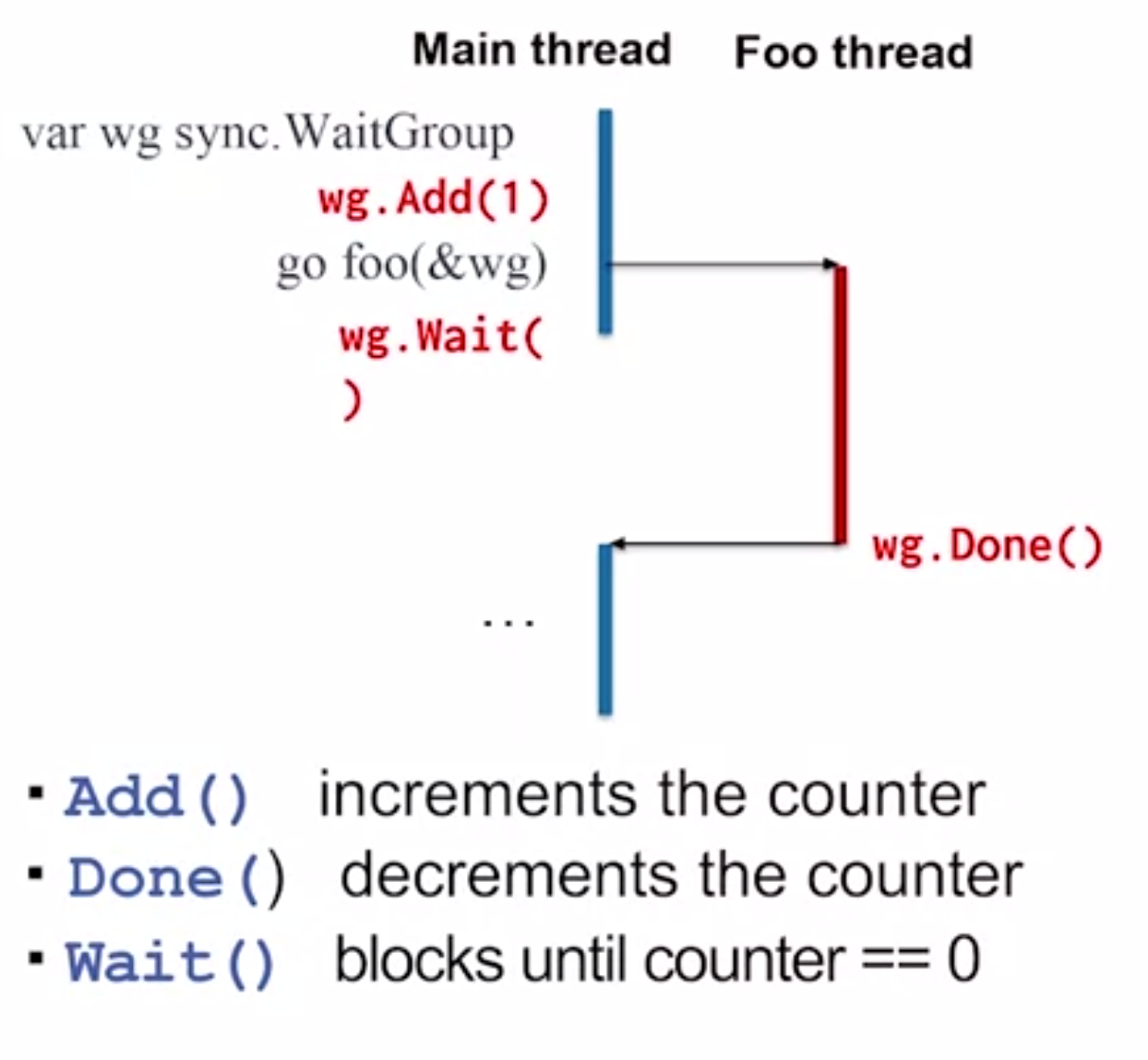


Topic 3.3 Basic Synchronization

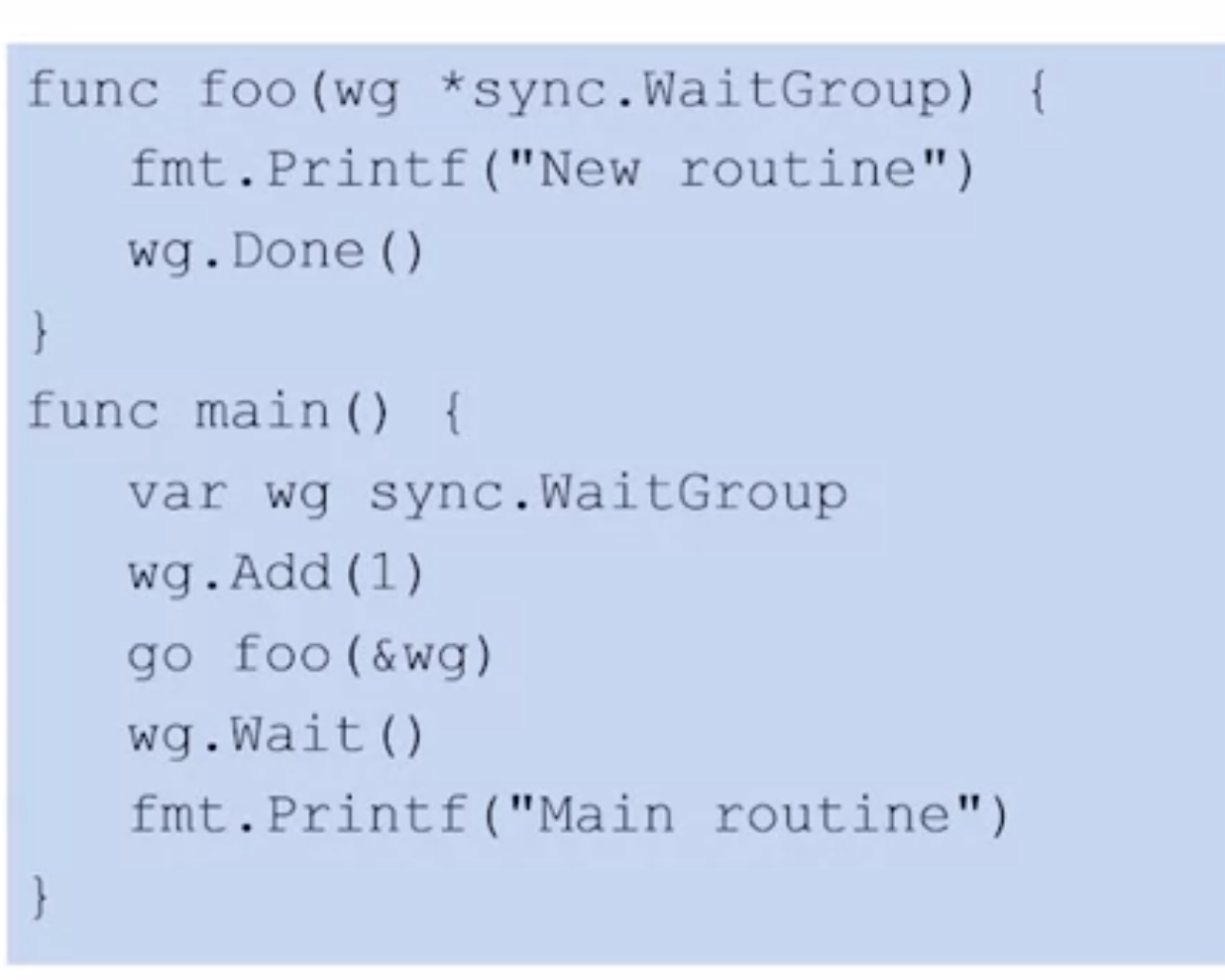


## Topic 3.3 Wait Groups

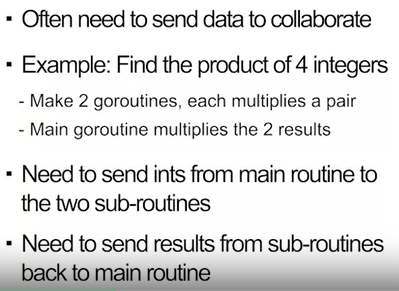




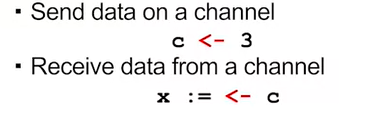
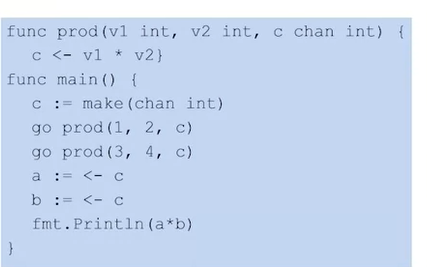
A lot of people will call defer on wg.Done() to be safe



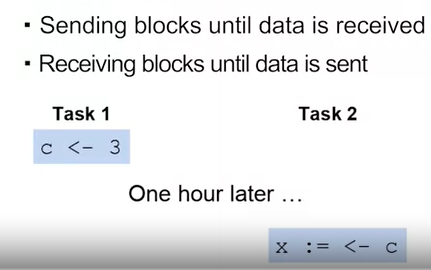
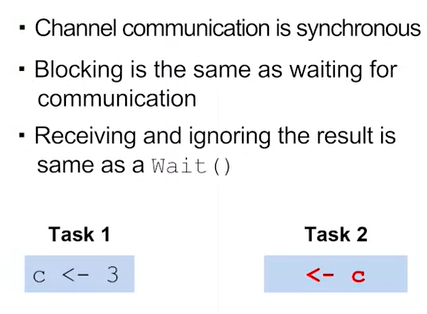
## Topic 3.4: Threads in Go & Communication

* Goroutines usually work together to perform a bigger task
* 

**Channels**

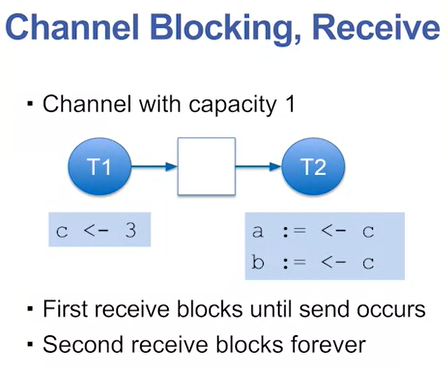
* Transfer data between goroutines
* Channels are typed
* Use make() to create a channel
* 
* Send and receive data using the <- operator
* 
* 

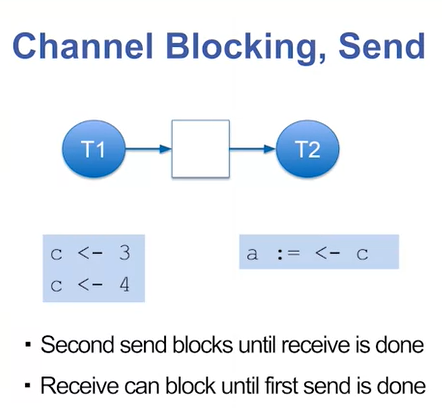
## Topic 3.5: Blocking in Channels

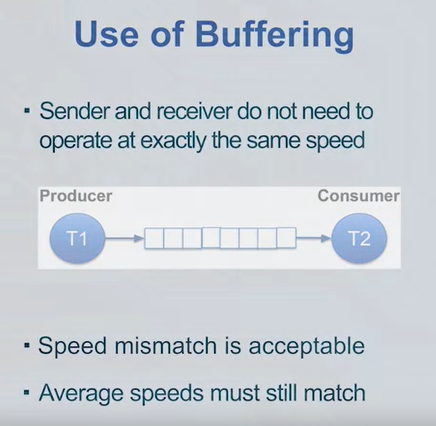
* Channels are by default unbuffered
* Cannot hold data in transit
* 
* Same thing applies to the reverse, if you’re waiting for receive, you block instructions
* Because the channel is unbuffered, the send will block until the receive happens, and the receive will block until the send happens
* 

Topic 3.6 Buffered Channels

* Channels can contain a limited number of objects
  + Default number is 0 (unbuffered)
* Capacity is the number of objects it can hold in transit
* Optional argument to make() defines channel capacity
  + 
* Sending only blocks if **buffer is full**
* Receiving only blocks if **buffer is empty**







## Module 3 Quiz

