Concurrency In Go Katherine Cox-Buday 2017

* Preface

What is duck typing?

golang.org , golang.org/play , go.googlesource.com/go
groups.google.com/group/golang-nuts , github.com/golang/go/wiki
*all code examples katherine.cox-buday.com/concurrency-in-go
bit.ly/concurrency-in-go

* Ch. 1 An Intro to Concurrency

Concurrent: usually refers to a process that occurs simultaneously w one or more processes. It's implied that all of these processes are making progress at the same time

- this book covers Concurrency in the context of Go, how it's modeled, its problems & how to compose primitives w/in this model to solve problems

HISTORY

Moore's Law: # of components in an integrated circuit will 2x every year (1965-1975) then every 2 years (1975-2012)

* then ∞ core (multicore) processors were invented

Amdahl's Law: a way to model the gains from implementing a solution in a parallel manner. Gains are bounded by how much of the program must be written sequential

What are spigot algorithms?

embarrassingly parallel: technical term meaning the problem can easily be divided into parallel tasks

* Amdahl's law helps determine if parallelization will help.
* For problems that are embarrassingly parallel it's recommended that the app is written to scale horizontally

Scale horizontally: take instances of your program & run it on ↑ CPUs, or machines & runtime will improve

* cloud computing in 2000 helped us apps that should scale horizontally

Cloud computing: implied access to vast pools of resources that were provisioned into machines for workloads on-demand & provisioned specifically for the programs to be run

* w cloud computing ↑ problems, in particular how to model code concurrently & what if pieces of your program could be running on disparate machines?

Web scale: a new type of brand for software. You could expect the software to be embarrassingly parallel to enable properties like rolling upgrades, elastic ↔ scaling arch, & geographic distribution. ↑ complexity in comprehension & fault tolerance

Why is Concurrency hard?

Race Conditions

Race condition occur when ∞ operations must execute in the correct order but the code doesn't guarantee the needed order.

→ Commonly shows up as a data race where a variable is being both read from & written to at the same time by two concurrent operations

→ Commonly written b/c devs are writing or thinking about their code sequentially when order isn't guaranteed

" I sometimes find it helpful to imagine a long period of time passing b/w operations."

* introducing Sleeps() make race conditions more unlikely but ↑ inefficiency & will never 100% eliminate the issue. May help w/ debugging but won't solve the problem.

"... you should always target logical correctness."

* issues may not show up for years but are usually precipitated by a change in the environment or unprecedented occurrence.

Atomicity

atomic code is indivisible or uninterruptible in its operating contex

What makes concurrency hard?

Atomicity cont'd.

"the atomicity of an operation can change depending on the currently defined scope."

#1: define the context, or scope, the operation will be considered to be atomic in.

indivisible + uninterruptible means w/in the defined context the operation will happen in its entirety w/o anything happening in that context simultaneously

* Combining atomic actions may not create an atomic operation
* you can use different techniques to make code atomic
  → the key is given which sections of code to make atomic

Memory Access Synchronization

↳ 2 concurrent ops trying to access a resource non-atomically (for example:)

Critical Section is a section of a program that needs exclusive access to a shared resource

→ you can synchronize access to shared memory b/w crit. sections
→ will be covered more in the book
→ first attempt locks the shared resource w/ sync.mutex, enters critical section, then unlocks the resource once done.
  ↳ but this doesn't solve the race condition or logical correctness ~~Correctness~~ + ↑ maintenance + ↓ performance + requires humans to follow convention

Deadlocks, Livelocks & Starvation

"These issues all concern ensuring your program has something useful to do at all times."

Deadlock: a program where all concurrent processes are waiting on one another + can't recover w/o outside intervention

Coffman Conditions: basis for techniques to detect, prevent + correct deadlock

- mutual exclusion
- wait for condition
- No Preemption
- circular wait

→ if you can ensure even one of these conditions isn't true you can prevent deadlocks

What makes concurrency hard?

Deadlocks, Livelocks + Starvation cont'd

~~Livelock~~ Live lock is when concur. ops. are running but doing nothing to move the state of the program forward.

↳ commonly occur when ∞ concur. processes attempt to prevent a deadlock w/o coordination. Can be more difficult to spot because activity is occurring.

* Livelocks are a subset of Starvation problems

Starvation when a concurrent process can't get all the resources it needs to perform work. Usually implies that there are ⇒ >=1 *greedy concurrent processes unfairly preventing >=1 concurrent processes from accomplishing work efficiently (or at all)

ex. of greedy vs polite. Greedy locks for full duration of work. Polite will unlock + lock as it can to share resources.

* Starvation makes a good arg. for metrics. Ex. you can log when work is done + assess if your rate of work is as high as you expect

* it's much easier to expand a memory lock than restrict it.

* starvation can also apply to CPU, memory, file handles, db connections, etc. any resource that must be shared is a candidate.

Determining Concurrency Safety

It helps to know: ① who is responsible for the concurrency?
② how is the problem space mapped onto concurrency primitives?
③ who is responsible for synchronization?

↳ Comments can help cover this information or obvious func signatures

* Go has language features that help balance concurrency clarity w performance. Like:
  - concurrent, ↓ latency (blw 10 & 100 µseconds!), garbage collector
  - multiplexing ops onto OS threads
  - concurrency primitives

* Ch. 2: Modeling Your Code, Comm. sequential processes

"Concurrency is a property of the code; parallelism is a property of the running program."

① we write concurrent code that we hope is run in parallel

② it's possible to be ignorant of if concurrent code is running in parallel
↳ b/c of abstraction b/w the code & how it's run

③ parallelism is a function of time or context
↙ def. here as the bounds by which ops could be considered parallel

"...the context you define is closely related to the concept of concurrency and correctness."

- as we move ↓ stack of abstraction modeling things concurrently ↑ difficult to reason about & ↑ important & ↑ difficult, ↑ important to have access to concurrency primitives that are easy to compose
- in Go you model things in goroutines, channels & sometimes shared memory @ a level of abstraction below OS threads

"Communicating Sequential Processes" by Tony Hoare (197
↳ aka CSP & is a technique & name of a paper that Go is inspired by.
↳ it suggests that input & output are two overlooked primitives of programming (esp. w/ concurrent code)

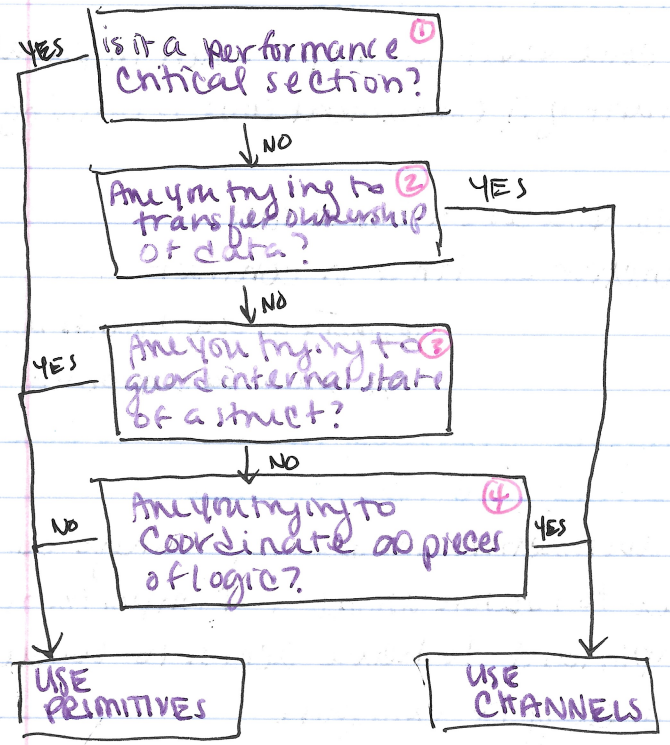
Processes: any encapsulated portion of logic that req'd input to run & produced output for other processes to consume

* two processes correspond if output from one was input for the other.

guarded command: a statement w/ (conditional or guard) left- & right-hand side split by a →, if left- didn't return true then right- didn't happen*

* Go developers handle parallelism & can improve w/o affecting the concurrent code written to match the problem statement

* Go allows you to choose b/w CSP & primitives (ex. channels) and memory access synchronizations (ex. sync library)



(1) ex. you find a bottleneck, either the program should be refactored or primitives should be used.

(2) data has an owner + ↑safety by ensuring only 1 concurrent ctx. has ownership of data at a time. Buffered channel can decouple producer + consumer + make your code composable

(3) w/ sync. primitives you can hide impl. details for ~~toting~~ locking. Try to keep locks constrained to a small lexical scope.

(4) Channels are inherently more composable "If [you're] ... struggling to understand your concurrent code, w/ a deadlock or race is occurring, & you're using primitives ... you should [probably] switch to channels."

* There are other concurrency patterns that are abstracted at the OS level. Since Go is one level below OS threading these patterns may not help + their & use case more ~~contr~~ constrained in Go.

"Stick to modeling your problem space w/ goroutines, use them to represent the concurrent parts of your workflow, & don't be afraid to be liberal when starting them."

Summary: "aim for simplicity, use channels when possible, and treat goroutines like a free resource."

* Ch 3: Go's Concurrency Building Blocks

Goroutines

↳ every Go program has one, the **main goroutine**, auto-created & started when the process begins

goroutine: a function that is running concurrently alongside other code.

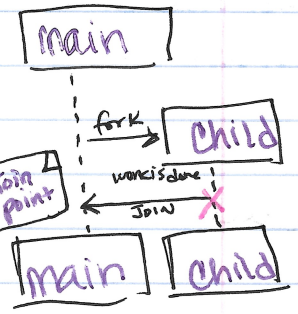
- `go funcName()` ← call function as a goroutine
- `go func() { ... }()` ← call anon. func. as a goroutine
  - ↖ invoke func right away to use go keyword

↳ goroutines are **coroutines** which are concurrent subroutines (functions, closures or methods in Go) that are **non preemptive** (can't be interrupted). They instead have ∞ points to allow suspension or reentry.

* go's runtime controls suspension/reentry for goroutines to suspend them when they're blocked & re-enter when they are unblocked.

"Coroutines, and thus goroutines, are implicitly concurrent constructs, but concurrency is not a property of a coroutine: something must host several coroutines simultaneously & give each an opportunity to ~~execu~~ execute - otherwise they wouldn't be concurrent!"

M:N scheduler is how goroutines are hosted where M green threads are mapped to N OS threads. Goroutines are then scheduled onto green threads.



fork-join model of concurrency is what Go follows.

- ↙ at any point a child branch of execution can be split & run concurrent w/ parent
- ↘ at some point in the future these branches of execution will join back together.

Join point is where the child rejoins the parent

*is possible for the main go thread to finish executing before the goroutine can ~~run~~ run.

* you can create a join point to ensure the goroutine executes. One way to do this is w/ sync.WaitGroup

Closures close around the lexical scope they are created in thereby capturing variables

"goroutines execute w/in the same address space they were created in"

* the garbage collector tracks if goroutines are accessing variables & will maintain access until the goroutine is complete.

"Since 2 goroutines can operate against the same address space, we still have to worry about synchronization."

↳ can either sync memory access or use CSP primitives to share memory by communication

* goroutines usually function w/ a few KBs of memory & grow/shrinks as needed. CPU overhead ~3 instructions / function call

* garbage collector won't collect goroutines that have been abandoned. this is a goroutine leak

Context switching is when something hosting a concurrent process must save its state to switch to running a different concurrent process

↳ cheaper w/ goroutines b/c the switching is done in software

"Creating goroutines is very cheap, and so you should only be discussing their cost if you've proven they are the root cause of a performance issue."

The sync Package

* contains concurrency primitives that are useful for ↓ level memory access synchronization

WaitGroup + you don't care about the result or you have other ways to collect the result

↳ wait for goroutines to complete + is a concurrent-safe counter. you wg.Add(#) to increment wg.Done() to decrement + wg.Wait() will trigger when the count hits zero

Sync Package Cont'd

Mutex & RWMutex

↳ "memory access synchronization"

Mutex: mutual exclusion, a way to guard critical sections of your program (sections that requires exclusive access to a shared resource).

"...whereas channels share memory by communicating, a Mutex shares memory by creating a convention developers must follow to synchronize access to the memo

↳ if you lock w/ a mutex always unlock() w/ a defer in case the code panics so you don't end up w/ a deadloc

* You can reduce the scope of a critical section w/ a RWMutex that allows you to lock for reading or writing vs both w/ Mutex. Usually advisable to use RWMutex if it logically makes sense.

Cond

"... a rendezvous point for goroutines waiting for or announcing the occurrence of an event."

↑ in this def. "event" is any arbitrary signal btw 2+ goroutines w/ no info other than it occurred.

↳ tends to be used to wait for a signal before continuing execution on a goroutine. "... [a] way for a goroutine to efficiently sleep until it was signaled to wake & check its condition"

* calling c.Wait() suspends the current goroutine which allows other goroutines to run on the OS thread.

↳ on entry to c.Wait(), the Cond's Locker has Unlock called on exiting c.Wait(), the Cond's Locker has Lock called

⚠ "It looks like we're holding this lock the entire time while we wait for the condition to occur, but that's not actually the case."

c.Signal() will signal the longest waiting goroutine

c.Broadcast() will signal all waiting goroutines

* The Cond type is more performant than channels

* works best when constrained by tight scope or encapsulated w/ a type

Sync Package Cont'd

Once

"... sync.Once is a type that utilizes some sync primitives internally to ensure that only one call to Do ever calls the function passed in — even on different goroutines."

→ only counts the # of times Do is called, not the function passed in

Pool

"concurrent-safe implementation of the object pool pattern."

→ a way to create/make available a fixed # (or pool) of things for use.

myPool.Get() get an available resource from the pool

myPool.Put() put the resource back

* helpful for memory management or warming a cache for quick operations.
* good for expensive operations

Keep In Mind:

1. give sync.Pool a New member variable that is thread-safe when called
2. when myPool.Get(), make no assumptions abt the state of the obj. received
3. call myPool.Put() when finished (usually done w/ defer)
4. objs in myPool must be roughly uniform in makeup.

Channels

→ derived from Hoare's CSP

* can be used to sync. mem. accesses
* should be used to communicate info b/w goroutines
* useful b/c channels can be composed

"... a channel serves as a conduit for a stream of [info.]; values may be passed along the channel + then read downstream."

~~Sync Package cont'd~~

Channels cont'd

* can declare a unidirectional channel w/ ← operator

Read only: `var dataStream <-chan interface{}`
`dataStream := make(<-chan interface{})`

Send only: `var dataStream chan<- interface{}`
`dataStream := make(chan<- interface{})`

Unidir. channels are often used as function parameters and return types. Go can implicitly convert bidir. chans. into unidir. chans. when needed.

* channels are typed, the empty interface `interface{}` will accept any data, but you can be more specific.
* channels in Go are **blocking**, any goroutine that attempts to write to a channel that is full will wait until it can write. Any goroutine that attempts to read from an empty channel will wait until 1≤ items is placed on the channel.
  * this blocking can cause deadlocks

`salutation, ok := <-stringStream`

salutation: what is read from the channel

ok: was the value read generated by a write someone else? or default val. gen. by a closed channel?

`close(stringStream)` sends a universal sentinel telling others that no more values will be sent.

ranging over a channel will close the loop when the channel is closed

`for integer := range intStream { ... }`

* a closed channel can be read from ∞ times
* you can use closing a channel as a way to unblock ∞ goroutines waiting on ← `begin ... close(begin`

buffered channels are given a **capacity** when instantiated

unbuffered channels are buffered channels w/ a capacity of

* channels that are at capacity, or full, can't be written to

can be a premature optimization & hide deadlocks

Channels cont'd

* reading/writing/closing nil channels, or non-instantiated channels will result in a panic.
- Channel ownership: a goroutine that instantiates, writes, & closes a channel
  - owners: write-access view of channel (chan)
  - utilizers: read-only view of channel (<-chan)

The goroutine that owns a channel should:
1. instantiate the channel
2. perform writes or pass ownership to another goroutine
3. close the channel
4. encapsulate these responsibilities & expose them via a channel

The channel utilizer/consumer worries about:
1. knowing when a channel is closed
2. responsibly handling blocking for any reason

"I can't promise that you'll never introduce deadlocks or panics, but when you do, I think you'll find that the scope of your channel ownership has either gotten too large, or ownership has become unclear"

* Channels are the glue that hold goroutines together.

Select Statement

"the glue that binds channels together"
- compose channels to form larger abstractions
- they join components & have concepts like cancellations, timeouts, waiting & default values.

Select block encompasses a series of case statements that guard a series of statements. But the cases aren't tested sequentially & there's no fall-through.
- all channel reads/writes are considered simultaneously & if none are ready the whole select blocks.
- if >1 reads are ready at the same time, Go runtime will do a pseudo-random uniform selection, each case has an equal ch

Head First Design Pattern, O'Reilly

Select cont'd

- what if no channels are ready? Block ∞ or timeout
  case <- time.After(1 * time.Second)
- what if no channel is ready & we need to do something?
  Use a default case `default:`
  ↳ usually used in a for...select loop

`select {}` will block ∞

GOMAXPROCS Lever

↳ this function relates to the # of OS threads that will host "work-queues". Automatically set to the number of logical CPUs on the host machines

* you generally leave this value alone but you could ↑ this value to be higher than your # of logical CPUs to ↑ the chance of a race condition
  ↳ you may also tweak performance but you risk stability
  check after every commit, when you use diff. hardware, or diff. versions of Go.

* the second half of this book covers how to combine these primitives & patterns that have been discovered.

* Ch. 4 Concurrency Patterns in Go

Confinement, can be difficult to achieve, ↑ simplicity

* immutable data & data protected by confinement are also safe to use in ∞ concurrent processes
  ↳ can use copies of values instead of pointers to values to keep the values immutable.

Confinement: ensure info is only ever available from one concurrent process

Ad Hoc Confinement: confinement through coding convention, difficult to maintain

Lexical Confinement: make it impossible to do the wrong thing by using lexical scope to expose data/primitives for ∞ concurrent processes

for-select Loop

```
for { // loop ∞ or over a range
  select { // do work w/ channels }
}
```

① sending iteration var over a channel
② loop ∞, waiting to be stopped

(a)
```
for {
  select {
    case <- done:
      return
    default:
  }
  // do non-preemptable work
}
```

(b)
```
for {
  select {}
  case <- done:
    return
  default:
    // do non-preemptable work
}
```

Preventing Goroutine Leaks

* remember goroutines represent units of work that may / may not run in parallel

Paths to termination:
① when work is completed
② can't complete work, unrecoverable error
③ told to stop
↳ a goroutine may be in a network of goroutines with other resources knowing if it should stop
* establish a signal b/w parent & child goroutine & then close child goroutine when ready
↳ usually done via a done channel, or built on that concept

"If a goroutine is responsible for creating a goroutine, it is also responsible for ensuring it can stop the goroutine."

The or-Channel

↳ creates a composite done channel through recursion & goroutines, helpful at the intersection of modules where ∞ conditions exist to cancel trees of goroutines

Error Handling

↳ fundamental question: who should be responsible for handling the error? Which can be difficult w/ concurrent processes.

"In general, your concurrent processes should send their errors to another part of your program that has complete information about the state of your program..."

* can couple all possible outcomes of a goroutine in its response
  ex. potential result w/ potential error & pass to code w/ wider context

① errors should be considered first-class citizens

② any produced errors should be tightly coupled w/ your result type

Pipelines

↳ another tool to form an abstraction in your system, helpful when processing streams or batches of data

pipeline: a series of things that take data in, perform an operation, & pass data out. Each part is a **stage** of the pipeline.

* stages are abstracted from each other & you can fan-out or rate-limit portions of your pipeline.

Properties of a pipeline stage:

① consumes & returns same type

② must be reified by the language to be passed around

reification: a concept in the lang. is exposed to devs to work w/ it directly, ex. in Go you can define vars of type func & pass functions around your program.

* stages could be seen as a subset of higher order functions & monads, related to functional programming.

batch processing: a function/thing that operates on chunks of data at once vs. one at a time

stream processing: stage receives & emits one element at a time

Best Practices for Constructing Pipelines

* channels are ideally suited:
  - receive & emit values
  - safe when concurrent
  - can be ranged over
  - are reified

generator a type of function that converts a discrete set of values into a stream of data on a channel

↳ frequently sum up pipelines b/c at the beginning of a pipeline you'll always have some batch of data to convert to a channel

* w/ channels each stage can execute concurrently
  ↳ the stages are interconnected by the common done channel & channels that are passed into the next stage of the pipeline
  - two points must be pre-emptable
    1. creation of discrete value that is not nearly instantaneous
    2. sending of the discrete value on its channel

Make sure "our entire pipeline is always preemptable by closing the done channel."

Handy Generators

Repeat: repeat values passed to it until it's told to stop
- efficient b/c its send will blocks on the take stage's retrieve
  ↳ can also repeat function calls

* it can be helpful to use interface{} to ↑ reusability of pipeline stages, but use of interface{} is taboo in Go
* you can have a type assertion stage if needed

"Generally, the limiting factor on your pipeline will either be your generator, or one of the stages that is computationally intensive."

Fan-Out, Fan-In

- meant to address when computationally expensive stages slow down a pipeline.
- This pattern attempts to parrallelize pulls from an upstream stage to ↑performance of a pipeline

Fan-out: the process of starting ∞ goroutines to handle pipeline input

Fan-In: the process of combining ∞ results into one channel

Consider fan-out:

① it doesn't rely on values that the stage had calc. before
② takes a long time to run
↳ b/c there's no guarantee the order concurrent copies your stage will run or return.

*how do you know which stage to fan out?

*to fan-out start ∞ version of the stage to fan out
↳ the author makes as many versions as they have CPUs
↳ in production you can determine the optimal # of CPUs

→ then you fan-in & multiplex the data streams into a single stream
↳ involves creating the multiplexed channel consumers will read from, spin up one goroutine for each incoming channel, one go routine to close the multiplexed channel when all incoming channels have ~~be~~ been closed

The or-done channel

*We need to wrap our read from the channel w a select statement that includes a done channel. You can encapsulate this added code in a goroutine for readability.
"I would encourage you to try for readability first. an avoid premature optimization
↳ if you're working w channels from disparate parts of your system & you can't make assertions about how a channel will behave when code you're working with is canceled v/a its done channel.

Tee-channel

* may want to split values coming in from a channel & send to different parts of your codebase
  - from tee command in Unix-like systems
  "pass it a channel to read from, and it will return two separate channels that will get the same value"
  → take a value & loop twice to write to one channel & then write to the other channel.
  → look at code in this chapter, it helps us using channels as the 'join points of your system.

The bridge-channel

* you may want to consume values from a sequence of channels ← chan ← chan interface{}
  ↳ a sequence of channels suggests an ordered write (although from different sources).

bridging channels can destructure the channel of channels into a simple channel
  ↳ can help consumers focus on the values vs. where they came from.

Queuing

queuing: accepting work for your pipeline even though the pipeline isn't ready for more
  ↳ once your stage has completed some work it stores the value in a temporary location to be pulled when the next stage is ready.

buffered channels: a type of queue

* introducing queuing is usually one of the last techniques you want to employ, it can hide synchronization issues & you may need ↑ or ↓ queuing

"Queuing will almost never speed up the total runtime of your program; it will only allow the program to behave differently."

* helps reduce the time the stage is in a blocking state but not necessarily ↑ performance

Queue Cont'd

- so they would experience lag but not rejections
- the true utility of queues is to decouple stages so the runtime of one stage has no impact on another stage. This can then alter the runtime behavior of the system as a whole.
- Can ↑ performance:
  1. batching requests in a stage saves time
  2. if delays in a stage produce a feedback loop in the system

chunking when units are queued in a buffer until a sufficient chunk has been accumulated & is then written out again.

"growing memory is expensive ... the less times we have to grow, the more efficient our system as a whole will perform."

- usually any time performing an operation requires an overhead, chunking may ↑ performance.

negative feedback loop: downward-spiral, because a recurrent relation exists b/w pipeline & upstream systems. The rate upstream systems submit new requests is linked to how efficient the pipeline is.

↳ ↓efficient pipeline → ↑upstream input → ↓efficient pipe which results in a death-spiral

↳ buffering ~~to~~ can prevent a death spiral unless the caller times out resulting in processing dead requests ↓efficiency of pipeline. Can ↑ lag but prevent a downward-spiral

Queuing should be implemented either:

1. at the entrance of pipeline
2. in stages where batching will lead to ↑ efficiency

* avoid queues in other locations it can have disasterous consequences.

Little's Law: $L = \lambda W$ avg units in system = avg. arrival × avg. time

only applies to stable systems (ingress = egress) & not unstable (ingress > egress) → in death-spiral

Queue Cont'd

Little's Law Cont'd

$L = \lambda W$ (L: avg units in system, λ: avg arrival, W: avg time in system)

less concerning unstable (ingress < egress) w/ underutilization

so assume a stable system (ingress = egress)

- if you want to ↓ W (avg time unit is in system), ↓ # units in system by ↑ egress
  - → can help prove queuing won't ↓ time in system
- * to consider the whole pipeline the equation should be $L = \lambda \Sigma_i W_i$, your pipeline will only be as fast as your slowest stage — optimize indiscriminately!
- * you can utilize this law to evaluate your pipeline
- * as you ↑ queue you ↑ lag, you're trading system utilization for lag.

Persistent queue: a queue that is persisted somewhere else

NOTE: failures or panics can cause the pipeline to lose all requests, take care to handle this use case depending on the impact of losing those requests.

The context Package

* standard pattern to communicate extra info. alongside the notification to cancel or message on the done channel

Context type: will flow through your system (like the done channel). Each function downstream of concurrent call takes a Context as its first argument

- Deadline func: will a goroutine be canceled after a certain ti
- Err func: return non-nil if goroutine was canceled
- Value method: request-specific info to be passed along

Context Package purposes

1. provide API for canceling branches of your call graph
2. provide data-bag for transporting request-scoped data through your call-graph

Cancellation in function: ① goroutine parent may want to cancel ② goroutine may want to cancel its children ③ any blocking ops. w/in goroutine needs to be preemptable so it can be canceled

20 | the context pkg helps w/ all three

Context Package Cont'd

- how do we affect the behavior of cancellations below a current function in a call stack?
  - ↳ WithCancel, WithDeadline, WithTimeout takes a Context + returns a new Context

"If your function needs to cancel fxns below it in the call graph ... it will call one of these functions and pass in the Context it was given, & then pass the Context returned into its children."

* this allows successive layers of the call graph can have a context that matches their needs w/o affecting their parents
* ALWAYS pass instances of the Context & not references. They may change every stack frame.

→ to start the Context chain: Background → TODO
- Background: ↑ prod, empty Context
- TODO: ↑ staging only, empty Context

* when using Value
  1. key used must satisfy comparability (equality == & != return correct & consistent results)
  2. Values returned must be safe to access from ∞ goroutines

↳ recommended rules:
1. define custom key-type in your package to prevent collisions because the type defined is unexported ↓ conflicts & you must export functions that uses the data in static, type-safe functions
   - ↳ to avoid circular dependencies this rule coerces architecture into creating packages centered on data types imported from ∞ locations
2. What should be stored in a context? Common guidance:

"use Context Values only for request-scoped data that transits processes & API boundaries, not for passing optional params. to functions."

→

Context Package Cont'd

Heuristics for what data should be stored in context packages:

① data should transit process or API boundaries
② " " be immutable
③ " " trend towards simple types
④ " " be data, not types w/ methods
⑤ " " help decorate operations, not drive them
↳ otherwise you may be in the territory of optional parameters

* if the data you're storing violates all of these heuristics then it's worth taking a look at what you're trying to do
→ think about how long data is ~~held~~ traversing before being used. Should it be a parameter instead? But this is a conversation + an opportunity for discussion vs. hard rules. It also depends on the team + project.

* Ch. 5 Concurrency at Scale

* Composing patterns into practices that enable you to write large, composable systems that scale. Scaling w/in one process + more than one process

Error Propagation

* it's easy for something to go wrong + difficult to understand why it happened So carefully consider how issues propagate + how they're represented to the user

"What follows is an opinionated framework for handling errors in concurrent systems."

- Go forces errors to be handled at every frame in the call stack

Errors indicate a system state when it can't fulfill an operation explicitly/implicitly requested. It relays:

- what happened
- when + where it occurred
  ↳ should contain full stack trace but not in the error message + context info + time in UTC + which machine error occurred on

Error Propagation cont'd

Errors should be contained:

- friendly user-facing message
  - ↳ human centric, if error is transitory, ~one line of text
- how user can get more info
  - ↳ ex. cross-referentiable ID that can lead to a log of full details: time of occurrence, stack trace, etc. hash of stack trace can help w/ aggregation in bug trackers

* ideally all errors have this info - if it doesn't then you have a bug that may need to be investigated.

Errors could be categorized

① Bugs

② known edge cases

Bugs errors not customized to your system, may be on purpose.

↳ this distinction will help when determining how to propagate errors, how your system grows over time, what to ultimately display to the user.

- when you pass an error b/w contexts it may be well-formed in one context but not in the next. Hiding data that is too low-level may also be necessary.

"Note that [converting errors to our module's error type] is only necessary to wrap errors in this fashion at your own module boundaries — public functions / methods — or when your code can add valuable context"

* wrapping errors allows for error correctness to become an emergent property of the system
  - ↳ it can help highlight malformed errors & correct them over time
* you can canvas your top-level error handling → delineate b/w bugs & well crafted errors, then ensure all created errors are considered well-crafted over time

Timeouts & Cancellation

* timeouts are crucial to creating a system w/ behavior you can understand & cancellation is one natural response to a timeout

Reasons to support a timeout

- System Saturation
  - ↳ when its ability to process requests is at capacity it's better to timeout vs. waiting for a response
  - Guidelines for timeout:
    1. request is unlikely to be repeated when timed out
    2. you don't have storage resources
    3. if the need for the request or data it's sending will go stale
  - ↳ not timing out could lead to a death spiral if these guidelines aren't followed
- Stale Data
  - ↳ sometimes data has a best by date, if the process takes too long then it would be better to timeout & cancel the concurrent process
  - ↳ if the time frame is known ahead of time context.withDeadline or context.withTimeout could help
  - ↳ if the time frame is NOT known, the parent should be able to cancel so context.withCancel would help
- Attempt to prevent Deadlocks
  - ↳ it can be difficult to predict all edge cases so it may be a helpful preventative measure to put timeouts on all processes that is longer than necessary to prevent deadlocks.
  - ↳ it could result in a livelock but ↑ probability that the issue will be fixed by a timeout.
  - ↳ "the goal should be to converge on a system w/o deadlocks where the timeouts are never triggered"

Timeouts & Cancellation Cont'd.

Why a concurrent process might be canceled
- Timeouts
- User intervention
- Parent cancellation
  - ↳ if any kind of parent of concurrent operation — human or otherwise — stops, as a child of that parent, we will be cancelled
- Replicated requests
  - ↳ may send requests to 2+ concurrent processes for the fastest response then cancel the other requests when a response is received

* how to cancel? done channel, context.Context type
  - ↳ but what does this mean for the executing algorithm & downstream consumers? what should be considered
    - ① define the period w/in which our concurrent proces is preemptable
    - ② ensure any fxnality that takes more time than this period is preemptable
      - * break up the pieces of your goroutine into ↓ pieces gain for non preemptable atomic ops to complete in ↓ time than the period deemed acceptable.
        - ↳ what happens to modified state? (ex. shared db)
          - ↳ the solution here depends on context but if modifications to shared state is w/in a tight scope and/or they are easily rolled back you should be ok.
            - ↳ if possible, build up results as you go & modify state as quickly as possible
        - ↳ what about duplicated messages?

** ① make it unlikely that a parent will cancel after receiving a result from the child, bidir. comms. req'd, see heartbeats
② accept either first or last result reported & accept dupes
③ poll parent for permission w/ bidir comms. to send msg

Heartbeats

heartbeats are a way for concurrent processes to signal life to outside parties
↳ allow insight into system + can make testing the system deterministic

① ♡'s on time interval
② ♡'s at beginning of a unit of work

- Time intervals is helpful for ops. that sit around waiting, helps to know that silence is expected.
  - defend against no one listening to ♡ w/ default clause + infin loops b/c ♡'s may be sent while processing
  - heartbeats shouldn't be interesting in production but can help avoid a deadlock + be deterministic by not relying on a longer timeout in tests.
  - ~~♡~~ at beginning of work helps w/ tests
    ↳ if I need to implement this, review the code example
  - ~~♡ at intervals~~ a bad test will fail sometimes + succeed at other
  - they aren't strictly required but can help at times

Replicated Requests

- if receiving a response quickly is important you can replicate requests + cancel after getting a response.
  ↳ if done in-memory it's not too expensive, but it's all up to if the cost is worth the benefit.
- resources must be replicated as well so that each request has a chance to complete
- this also provides fault tolerance + scalability

Rate Limiting

- You can rate limit any resource (like API connections, disk reads/writes, network packets, errors, etc.)
- can prevent a host of attack vectors + secure your system, also covers non-malicious actions
- users have a mental model of your system + that they won't be affected by other users. If you break that they will be mad

Rate Limiting cont'd

- Rate limits allow you to reason about the performance + stability by preventing it from falling outside expected boundaries
- can also help set boundaries w/ clients paying for your system access, which can protect them too from paying too much.

token bucket rate limiting algorithm, requires access token to access resource & it's stored in a bucket waiting to be retrieved w/ depth of d. When you need to access resource you grab a token from the bucket.
- Once the bucket is empty you're rejected.
- r is the rate at which tokens are added back to the bucket & that becomes the rate limit.

burstiness is how many requests can be made when the bucket is full.

* your system should be able to handle the available burst
* you can have a rate limiter on sever + client side, whatever helps protect your system.
* it can be easier to keep limiters separate for fine grained control then combine into one rate limiter.
  - need to let each rate limiter know of the work being done to decrement token bucket + allow buckets to refill. There are timing considerations in this chapter.
* you can set limits for more than just time

Healing Unhealthy Goroutines

* unhealthy goroutines are usually blocked or stuck in a bad state where it can't recover w/o ext. help
  - could say that it's not the concern of a goroutine doing work to know how to heal itself from a bad state

healing the process of restarting goroutines
- can use ♡'s to tell if process is up + doing useful work

steward: logical that monitors a goroutine's health

Healing Unhealthy Goroutines cont'd

ward - the goroutine that the steward monitors
* the steward, in order to restart, will need a reference to a fxn that can start the goroutine
* be careful w/ restarting wards if your system is sensitive to duplicate values or find a way to avoid duplicate values

* Ch 6 Goroutines & the Go Runtime

* Go runtime does many things & in particular manages goroutines for you
  - Google has also put a lot of academia to use when creating Go

Work Stealing (strategy)

fair scheduling, naive strat. for sharing work evenly across all processors
- however, in Go concurrency is modeled w/ a fork-join model & likely dependent on one another, so one processor will likely be under-utilized & ↓ cache locality b/c tasks that req. the same data are on different processors.

* so processors are likely to become blocked or idle

① work-stealing alg: a FIFO queue, tasks enter queue & are dequeued as processors have capacity or block on joins
- ↓ cache locality, ↑ enter/exit critical sections

② deque, double-ended queue & each processor has its own thread. Rules for distributed queue:
- ① at fork point, add tasks to tail of deque assoc. w/ thread
- ② if thread idle, steal from head of deque assoc. w/ other thread
- ③ at join that can't be realized, pop off tail of thread's own deque
- ④ if deque is empty either
  - ⓐ stall @ join
  - ⓑ steal work from head of rand. thread's assoc. deque.

Work Stealing Cont'd

- work sitting on deque has interesting properties:
  1. it's work most likely needed to complete the parent's join, completing joins quickly → ↑ performance & memory
  2. it's work most likely still in our processor's cache, ↓ cache miss
- we are likely enqueuing/stealing tasks (goroutines) + continuations (so after a goroutine)

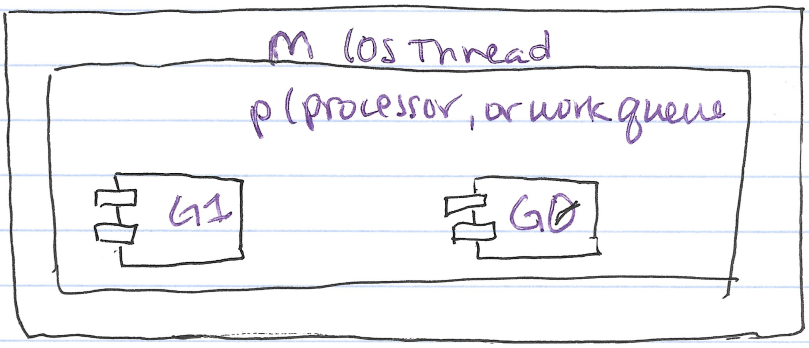
Stalling join: when an exec. thread reaches an unrealized join point + must pause + search for a task to steal
↳ there's a diff. b/w how a task + continuation stalls

"All things considered, stealing continuations are considered to be theoretically superior to stealing tasks, & therefore it is best to queue the continuation & not the goroutine."

* Continuation stealing usually req's compiler support so Go can do it but not all languages can. They'll implement "child" stealing

	Continuation (stealing)	Child
Queue size	Bounded	Unbounded
order of exec.	Serial	Out Of Order
join point	Nonstalling	Stalling

3 main concepts Go scheduler:
- G: a goroutine, current state of goroutine, its PC (program ctr.) (referenced as)
- M: an OS thread (aka machine in the source code)
- P: a context (aka ref. as a processor in the source code)



* One guarantee: "there will always be enough OS threads available to handle hosting every cont
↳ the runtime has a thread pool for not currently used threads.

* talks about optimizations for performance in this ch.

Work Stealing cont'd

"Slap the word go before a function or closure, & you've automatically scheduled a task that will be run in the most efficient way for the machine it's running on."

↳ so devs can still think in terms of primitives while using the efficiency, scaling & simplicity of Go.

* Appendix

* Tooling for writing concurrent code.

Anatomy of Goroutine Error

↳ Go >1.6 prints stack trace of the panicking goroutine but you can see the OG stack by setting GOTRACEBACK to all

Race Detection

using -race flag, this alg. will only find races in the code exercised, so it helps to run under real-world load

↳ several env vars can tweak behavior

LOG_PATH, STRIP_PATH_PREFIX, HISTORY_SIZE

* can integrate this flag into your CI process

pprof

* package in standard library for profiling. Can work while the program is running or by consuming saved runtime stats
* the present profiling tools can help & you can write custom profiles