Concurrency in Go Notes

BY ARNAV KUMAR

Web: https://arnavcs.github.io

Concurrency in Go is a publication by O'Reilly Media Inc. written by Katherine Cox-Buday. This is a collection of notes that I make about the text as I read it and of Golang as I learn it. This is not a summary or recreation of the text, but rather a reference for anyone who has already read the text. As such, please read the text to gain a better understading of the contents.

1 Basic Concurrency Ideas

Software / Design Pattern	Definition	Note

Amdahl's Law

Amdahl's Law models the improved performance of a fixed task when the resources are improved. In parallel computing, it is used to predict the speedup of using multiple processors. The relation is given as follows:

$$S_{\text{latency}}(s) = \frac{1}{(1-p)+\frac{p}{s}}$$

S_{latency}	the theoretical speedup of the whole program
s	the speedup of the part of the task from improved resources
p	the proportion of the execution time that benefits from the improved resources

Race Conditions

A race condition is when two or more operations must execute in the correct order, but the program leaves the order of execution unspecified.

Data Race

A data race is a race condition in which two concurrent operations attempt to read the same data at an unspecified time (namely one that could potentially conflict). In the following example, the program is not given a specified evaluation order, so the code that follows may execute before, during, or even after the goroutine. As such, the output is indeterminate.

```
var data int
go func() { data++ }()
fmt.Printf("%v\n", data)
```

Atomicity

An atomic operation is indivisible or uninteruptable in the context in which it is operating. For example, the statement `i++` consists of 3 atomic operations: retriving, incrementing, and storing the value of `i`.

Critical Selection

A critical selection is a section of code that requires exclusive access to a shared resource. In the following code, the `fmt.Printf()` and the goroutine are both critical selections.

```
var data int
go func() { data++ }()
fmt.Printf("%v\n", data)
```

Memory Access Synchronization	To solve the problem of multiple critical selections, only enable one critical selection to access the same shared resource at a time. This can be achieved, for example, with a mutex.
Deadlock	A deadlock is a state in which all concurrent processes are waiting on each other. A deadlock can be identified by the Coffman Conditions.
Coffman Conditions	There are 4 Coffman Conditions that detect, prevent, and correct deadlocks. The conditions are as follows: **Mutual Exclusion**: A concurrent process must hold exclusive rights to a resource at any one time. **Wait For Condition**: A concurrent process must hold a resource and be waiting for another resource. **No Preemption**: A resource held by a concurrent process can only be released by that process. **Circular Wait**: A process must be waiting on a chain of processes which is circular (meaning that the process is directly or indirectly waiting on itself to give a result).
Livelock	A livelock is when the current concurrent processes are performing operations, but these operations do not terminate or move the program closer to termination.

Starvation	Starvation is a superset of a livelock or deadlock where, more generally, a concurrent process does not recieve access to the resources it needs. A common example is having a "greedy worker" hold on to access to the resource, while a "polite worker" does not, and thus has less access to the resource: it is starved.
"Finding a Balance"	What should the range of a memory lock be? Should it be broad and cover multiple critical selections, or should each critical selection get its own lock? It is important to strike a balance in answering this question because memory access synchronization is expensive, but you also want to avoid writing greedy processes to mitigate starvation.
OS Threads	OS threads are a primitve at the OS context that can be used to run pocesses concurrently. The operating system is responsible for creating and managing the threads. The threads all have access to a shared resource space.
Green Threads	Green threads are threads that are managed by a program's runtime.
Preemptive and Non-preemptive Scheduling	Preemptive scheduling is when a process may be interrupted during execution, whereas non-premptive scheduling involves processes which cannot be interrupted, but rather just suspended at certain points.
Coroutines	Concurrent subroutines that are non-preemptive (meaning that they can't be interrupted) are called coroutines. They feature multiple points to suspend or reenter computation.
`M:N` Scheduler	A `M:N` scheduler is the mechanism that Golang uses to host goroutines and it consists of mapping `M` green threads onto `N` OS threads.
Fork-Join Model	The model that Golang follows for concurrency, a fork-join model is one in which a child branch can fork off from parent to be run concurrently. After the termination of the child branch, it is joined back to the parent branch at a join point.
Thread Pools	Thread pools are a software design pattern that maintains a collection of threads to map incoming tasks to threads for concurrent execution.
Concurrency vs. Parallelism	Parallelism is a property of a machine to be able to run two tasks simulatiously in the considered context. On the other hand, concurrency refers to when two processes have a lifespan that overlaps. In this sense, you could have a concurrent program running on a single thread where multiple threads are simluated. It is also possible that the concurrent processes run in parallel. Concurrency is a property of the code, and parallelism is a property of the execution of the code.
Process	A process is a portion of code that requires input to run and produces an output that is consumed by another process. The input and output of a process is called communication between processes.
Communicating Sequential Processes (CSP)	CSP is the name of a paper, programming language, and the idea of a descibing programs as processes which are sequential and communicate. Used in the paper describing CSP, the CSP language supported the use of `!` and `?` to send input into and read output from a process respectively. In addition, it supported guarded commands. This is the style of concurrency programming that Golang's channels are based on.

Guarded Command	When a statement should not be executed if another statement was false or a command exited, it is a guarded command. The CSP example below denotes a process `a`, from which a character `c` is continually read (while there is something to be read), and then inputted into the process `b`.
	`*[c:character;`
	`a?c -> b!c]`

Process Calculus	Process calculus is a mathematical way to model concurrent systems and analyze their properties.
Should I use CSP style or OS threads?	The CSP style has certain advantages that it comes with, and more generally, the Golang developing team suggest to use the CSP style over primitves like `sync.Mutex`, but there are certain guidelines outlined that help determine when you should use channels or OS thread primitives. Follow the first applicable statement. 1. If your code is performance critical, use primitives 2. If you are trying to transfer ownership of data, use channels 3. If you are trying to guard the internal structure of a struct, use primitives 4. If you are coordinating multiple pieces of logic, use channels 5. Use primitives
Mutex	Mutex stands for "mutual exclusion" and enables a way to express exclusive access to a shared resource. A mutex is often used for critical selections.
Object Pool	The object pool pattern is a way to create a fixed number of objects for use, and is especially useful for objects that are computationally expensive or objects that will take a lot of memory.
Channels	The channel pattern comes from CSP and is a way to pass information. If there is nothing to be read from a channel, reading from it blocks execution; waiting for a value to be added to the channel. Additionally, channels can be closed (to stop writing to the channel), in which case reading from the channel further empties the channel and reading from an empty closed channel will indicate that the channel is closed. Channels can also have buffers to store values to be read later. As a pattern, to write robust code, seperate the ownership of the channel so that the channel utilizers only have read access to the channel, and the channel owner has the following responsibilities: 1. Instantiate the channel 2. Perform writes or pass write ownership to another goroutine 3. Close the channel 4. Expose a reader channel for the channel utilizers

2 Golang Features and Building Blocks

Type	Function / Keyword

`func`	This keyword can be used to create named functions, closures, or anonymous functions. A named function example is show below.
	
	`func helloWorld(numTimes int) {`
	`  for ; numTimes > 0; numTimes-- {`
	`  fmt.Printf("Hello World!\n")`
	`  }`
	`}`
	
	An anonymous version of the same function is also shown below.
	
	`var f := func(numTimes int) {`
	`  for ; numTimes > 0; numTimes-- {`
	`  fmt.Printf("Hello World!\n")`
	`  }`
	`}`

Loops	All loops in Golang are declared with the keyword `for`. You can supply a stepping mechanism, nothing (for an infinite loop), a condition, or a range to describe the loop like in the examples below.
	
	`for i := 0; i < 10; i++ { fmt.Println(i) }`
	`for { fmt.Println("looping forever") }`
	`j := 0; for j < 10 { fmt.Println(j) }`
	`for i, v := range []int{1, 2} { fmt.Println(i, v) }`
	
	Breaking out of a loop and continuing to the next iteration can be done with the `break` and `continue` keywords. Adding labels to loops (by preceding the loop with `labelName:`) can specify which loop to `break` or `continue` out to. For example the following code prints `0 0`.
	
	`outside:`
	`for i := 0; i < 2; i++ {`
	`  for j := 0; j < 2; j++ {`
	`  if i < j { break outside }`
	`  fmt.Println(i, j)`
	`  }`
	`}`

`range`	A range can be used to iterate over strings, arrays, slices, key/value pairs of maps, and even channels.

`type`

This keyword creates a type macro, giving the second type the name passed into the function. For example, the following creates a new type called `HouseNumber`.

```
type HouseNumber int
```

`struct`

A struct in Golang can be created with the `struct {}` syntax. Since this creates a new type, it can be saved to a type variable with `type`.

```
type Fruit struct {
    name string
}
var apple Fruit = Fruit{"Apple"}
```

`interface`

Interfaces in Golang can be declared as follows. Here again, we use `type` to assign a name to this `interface`.

```
type Plant interface {
    getHeight() float
    getSpecies() string
}
```

Additionally, the existance of the empty `interface` in Go is special, because all types satisfy the empty interface, meaning it can hold any value. It is `interface{}`.

`go`

Creates a *goroutine* that runs the function, method, or closure concurrently by multiplexing onto OS threads. Each goroutine is a special class of coroutine where you do not have to manually describe the suspension and resuming of the routine. At runtime, Golang automatically suspends goroutines when they are blocked and resumes them when they are unblocked. Goroutines use the fork-join model for concurrency and during runtime, a `M:N` scheduler is used. See the following example using goroutines modified from the textbook that uses closures to print `"go"`, `"rust"`, and `"c"` concurrently in an unspecified order.

```
var wg sync.WaitGroup
for _, lang := range []string{"go", "rust", "c"} {
    wg.Add(1)
    go func(l string) {
        defer wg.Done()
        fmt.Println(l)
    }(lang)
}
wg.Wait()
```

`defer`

Defers the execution of the statement to the end of the function. In the following example, the mutex isn't unlocked until the after the value of `data` increments.

```
var data int = 0
var mu sync.Mutex
func inc() {
    mu.Lock()
    defer mu.Unlock()
    data++
}()
```

`sync.Mutex`	A mutex type that supports the `.Lock()`, `.TryLock()`, and `.Unlock()` methods. These methods declare exclusive access to the shared resource that the mutex represents. By convention, a mutex unlock statement is in a `defer` statement to avoid `panic`ing meaning that the mutex is not unlocked.
`sync.RWMutex`	This form of mutex requires the specification of the type of access desired. An arbitrary number of readers are allowed to read the same resource granted that there are no writers. In exchange for the greater control over the memory (and potentially less opportunity for starvation), it gives lower performance than `sync.Mutex` for a small number of readers. When the number of readers is high, though, it's performance is noticible. The supported methods are those from `sync.Mutex`, and the additional `.RLock()`, `.TryRLock()`, `.RUnlock()`, and `.RLocker()`.

`sync.Cond`	A `sync.Cond` is a "rendevous point" for goroutines waiting for an event (an signal between two or more goroutines that carries no information). The instantiation of a `Cond` is done with `sync.NewCond` which takes a `sync.Locker` interface (accessible with `.L`). Additionally, the methods `.Broadcast()`, `.Signal()`, and `.Wait()` are avaliable to be used. Consider the following function from the textbook that "subscribes" a function to a `Cond`, running the function once when the `Cond` first broadcasts.
	
	`subscribe := func(c *sync.Cond, f func()) {`
	`  var goroutineRunning sync.Waitgroup`
	`  goroutineRunning.Add(1)`
	`  go func() {`
	`  goroutineRunning.Done()`
	`  c.L.Lock()`
	`  defer c.L.Unlock()`
	`  c.Wait()`
	`  f()`
	`  }()`
	`  goroutineRunning.Wait()`
	`}`

`sync.Once`	A variable, `once`, of type `sync.Once` will support the `.Do(func())` method which will only execute the passed function once regardless of what if a different function is passed.
`sync.Pool`	A pool object is an implemenation of an object pool. It can be instantiated by specifying the `New` field which is a thread safe member variable function that creates a new object in the pool. the `Pool` also supports the methods `.Get()`, and `.Put(object)`. Make no assumptions about the state of the instance you get back from `.Get()`, but objects in the `Pool` should be roughly uniform in makeup.

Channels	Channels that are read-write, read-only, and write-only that carry values of type `T` have types `chan T`, `<-chan T`, and `chan<- T` respectively. A channel can be closed if it is writable, and is done so with `close()`. To read all the values in the channel until it is closed, use `range`. Additionally, buffer size of the channel can be specified during initiation, and the default buffer size is 0. Reading from a channel instantiated with a buffer of capacity 4 can look as follows.
	`channelOwner := func() <-chan int {`
	`  intStream := make(chan int, 4)`
	`  go func() {`
	`  defer close(intStream)`
	`  for i := 0; i < 10; i++ { intStream <- i }`
	`  }()`
	`  return intStream`
	`}`
	
	`readIntStream := channelOwner()`
	`for element := range readIntStream {`
	`  fmt.Println(element)`
	`}`

`make()` vs. `new()`	`make()` creates slices, maps, and channels by taking in a type `T` followed by a list of expressions and returns a value of type `T`. On the other hand, `new()` simply returns a pointer (type `*T`) to allocated memory that is initialized with `0`s.

Type Assertions	Type assertions "reveal the concrete value" in an interface variable. If the assertion is false, and that case isn't handled, panic occurs. See the following example of the syntax of type assertion.
	`var i interface{} = 1`
	`v, ok := i.(int)`
	`if ok == false {`
	`  fmt.Println("Incorrect type")`
	`} else {`
	`  fmt.Println(v)`
	`}`

`select`	The select statement is able to bind channels together. Namely, all `case` statements are simultaneously checked to see if they are ready (for reading this is a populated or closed channel, and for writing this is a channel not at capacity). If there is no `defualt`, then the execution is blocked until one of the channels is ready. One of the cases is then chosen at random, and the associated statements run. If there is a `default`, then the execution isn't blocked. This can be used to complete other tasks while waiting for a result.
`runtime.GOMAXPROCS()`	Takes an integer parameter that specifies the number of OS threads that will host "work queues".
`runtime.NumCPU()`	Returns the number of logical CPUs that can be used by the current process.

`iota`	This keyword is used in conjuction with the `constant` keyword. It represents an non-negative integer starting from 0. It resetsits value back to 0 after every `constant`, and can be used to define enums as such. For example, the following will create an enum type, `Size`, where `xs` is `-2`, and `xl` is `2`, and there is no `Size` value `m`.
	`type Size int`
	`const (`
	`  xs = Size(iota - 2)`
	`  s`
	`  _`
	`  l`
	`  xl`
	`)`

3 Concurrency Patterns in Golang

Concurrency Pattern	Definition	Note

Safe Operations	There are a couple different safe operations in concurrent programs, including synchronization primitives for memory sharing, synchronization with communicating, immutable data, and data produced by confinement.	
Confinement	Confinement is when it is ensured that data is only accessed by a single concurrent process. There are two types of confinement: ad hoc and lexical.	
	Ad Hoc	This is confinement that adheres to a convention set, but is problematic to enforce convention when working on large projects.

	Lexical	This form of confinement uses the compiler to enforce confinement, by limiting the scope of data and concurrency primitives. This is useful for data structures that are not concurrent-safe such as `bytes.Buffer` as we can see in the example below from the text.
		`printData := func (wg *sync.WaitGroup,`
		`data []byte) {`
		`  defer wg.Done()`
		`  var buff bytes.Buffer`
		`  for _, b := range data {`
		`  fmt.Printf(&buff, "&c", b)`
		`  }`
		`  fmt.Println(buff.String())`
		`}`

When to use Confinement?	Confinement can mean code that is easier to write and keep track of, and smaller critical sections, but the techniques to implement confinement are more involved that using synchronization built-ins.	

`for-select` Loop

The pattern of sitting a `select` in a `for` as below is common. The example shows an infinite loop, but a range could be used.

```
for {
    select {
    // useful work with channels
    }
}
```

This pattern can be used to send an iteration variable on a channel or to loop infinitely until stopped. See the example below modified from the textbook which demonstrates using a select statement to complete work while waiting.

```
done := make(chan interface{})
go func() {
    defer close(done)
    time.Sleep(5 * time.Second)
}()
loop:
for {
    select {
    case <-done:
        break loop
    default:
        workCounter++
        time.Sleep(time.Second)
    }
}
fmt.Println(workCounter)
```

Goroutine Paths to Termination

There are 3 paths for a goroutine to terminate. These are

1. When the goroutine has completed its work
2. When it cannot continue its work due to an unrecoverable error
3. When it is told to stop working

The third option is one which allows programs that could possibly cause deadlock or take up unneccessary memory to be killed, and is the basis of the `done` channel concurrency pattern.

Goroutine Ownership	As a good rule of thumb, the goroutine responsible for writing to a channel and creating the channel is the one responsible for the lifetime of the channel and stopping it.

`done` Channel	This can be used to convey to a goroutine that it should stop execution. In the following example, the goroutine created is signalled to stop executing by passing a channel which will signal to stop either trying to read or write to another channel.
	`printStrings := func(done <-chan interface{}, strings <-chan string) <-chan interface{} {`
	`  ret := make(chan interface{})`
	`  go func() {`
	`  defer close(ret)`
	`  for {`
	`  select {`
	`  case s := <-strings:`
	`  fmt.Println(s)`
	`  case <-done:`
	`  return`
	`  }`
	`  }`
	`  }()`
	`  return ret`
	`}`
	
	`done := make(chan interface{})`
	`terminated := printStrings(done, nil)`
	
	`go func() {`
	`  time.Sleep(time.Second)`
	`}()`
	`<-terminated`

`or` Channel

Suppose you have to compose multiple channels into one: a channel that closes when any of the passed channels are closed or written to. While you could simply have a different `case` in the `for-select` loop for each `done` channel, you could alternatively combine the channels with the `or` channel pattern. Here's the composition function below, taken from the textbook.

```
var or func(chans ...<-chan interface{}) <-chan
interface{}
or = func(chans <-chan interface{}) <- chan interface{} {
    switch len(chans) {
    case 0:
        return nil
    case 1:
        return chans[0]
    }

    orDone := make(chan interface{})
    go func() {
        defer close(orDone)
        switch len(chans) {
        case 2:
            select {
            case <-chans[0]:
            case <-chans[1]:
            }
        default:
            select {
            case <-chans[0]:
            case <-chans[1]:
            case <-chans[2]:
            case <-or(append(chans[3:], orDone)...):
            }
        }
    }()
    return orDone
}
```

Handling Errors	Consider encapsulating errors in a struct to handle them better upstream. For example,
	`type Result struct {`
	`  Error error`
	`  Value interface{}`
	`}`

Pipelines and Pipeline stages	A pipeline is a series of *stages* which take data in, perform an operation, and pass the data out. Stages are connected by passing of data. The stage must consume and return the same type, and stages must be reified by the language so they can be passed around (like functions). Stages can be either batch processing (where whole batches of data are operated on at once) or stream processing (where stages only operate on single elements at a time).
Making a pipeline	It is advised to make a stream pipeline when possible in Go. This is done by making each stage a goroutine and returning and passing channels. Each stage `range`s over the passed channel. Additionally, a generator function is required to pass input into the pipeline. The `done` channel pattern should be used to ensure the cleanup of all goroutines, and will be passed into all stages of the pipeline as well. Two parts of a pipeline stage must be preeptable: the creation of the discrete value and sending the discrete value on the channel.

`repeat` Generator	The repeat generator outputs a stream which repeats the set of discrete values passed. See the following code modified from the textbook.
	
	`repeat := func(done <-chan interface{}, vals ...interface{}) <-chan interface{} {`
	`  ret := make(chan interface{})`
	`  go func() {`
	`  defer close(ret)`
	`  for {`
	`  for _, val := range vals {`
	`  select {`
	`  case <-done:`
	`  return`
	`  case ret <- val:`
	`  }`
	`  }`
	`  }`
	`  }()`
	`  return ret`
	`}`

`take` Stage	The take stage only takes the first `num` elements of the channel passed in. The following is modified code from the textbook.
	
	`take := func(done <-chan interface{}, values <-chan interface{}, num int) <-chan interface{} {`
	`  ret := make(chan interface{})`
	`  go func() {`
	`  defer close(ret)`
	`  for i := 0; i < num; i++ {`
	`  select {`
	`  case <-done:`
	`  return`
	`  case ret <- <-values:`
	`  }`
	`  }`
	`  }()`
	`  return ret`
	`}`

`repeatFn` Generator	Exactly like the `repeat` generator, but with a signature of: `func(done <-chan interface{}, fn func() interface{}) <-chan interface{}` Additionally, rather than a loop over the values of `vals`, a simple `select` can be used with one case as `case ret <- fn()`:.
Empty Interfaces in Pipelines	Using empty interfaces allows the library of stages and generators used in a pipeline to be common between different pipelines and at any stage of a pipeline, type assertion can be used.
Type Assertion Stage	This stage has the following type signature (for some type `T`): `func(done <-chan interface{}, vals <-chan interface{}) <-chan T` This stage applies type assertion to everything passed in the pipeline. It is similar to the `take` stage, but rather it iterates over the range of the whole channel, and performs a type assertion.

Fan-Out Fan-In

When one stage of the pipeline is slowing down the entire pipeline, you can consider using more than one goroutine to do the operations of that stage in parallel, so that more than one datum is being processed in that stage at a time.

Fan-Out	The act of splitting the input of the pipeline into multiple goroutines.
Fan-In	The act of joining multiple results or multiplexing back into one channel for the pipeline.

The pattern is applicable when the operation of the stage doesn't care about computation history (including order).

Fan-Out

One can create an array of stage goroutines as such

```
numRoutines := runtime.NumCPU()
routines := make([]<-chan interface{}, numRoutines)
for i := 0; i < numRoutines; i++ {
    routines[i] = stage(done, inStream)
}
```

Fan-In or Multiplexing

The following multiplexing code example is modified from the text and requires that the order of output does not matter.

```
fanIn := func(done <-chan interface{}, chans ...<-chan
interface{}) <-chan interface{} {
    var wg sync.WaitGroup
    multiplexedStream := make(chan interface{})
    multiplex := func(c <-chan interface{}) {
        defer wg.Done()
        for i := range c {
            select {
            case <-done:
                return
            case multiplexedStream <- i:
            }
        }
    }
    wg.Add(len(chans))
    for _, c := range chans {
        go multiplex(c)
    }
    go func() {
        wg.Wait()
        close(multiplexedStream)
    }()
    return multiplexedStream
}
```

`or-done` Channel

We seem to be using a certain pattern of wrapping our reads from a channel with a `select` so that we can safely close our goroutines with a `done`. We can abstract this, as follows:

```
orDone := func(done, c <-chan interface{}) <-chan
interface{} {
    ret := make(chan interface{})
    go func() {
        defer close(ret)
        for e := range c {
            select {
            case <-done:
                return
            case ret <- e:
            }
        }
    }()
    return ret
}
```

But if the channel `c` doesn't close, and `done` is closed, we could have our `for` waiting unneccesarily on the next element of `c`, thus stalling. Thus, we prefer the following code:

```
orDone := func(done, c <-chan interface{}) <-chan
interface{} {
    ret := make(chan interface{})
    go func() {
        defer close(ret)
        for {
            select {
            case <-done:
                return
            case v, ok := <-c:
                select {
                case valStream <- v:
                case <-done:
                }
            }
        }
    }()
    return ret
}
```

This abstraction allows us to use simpler loops.

`tee` Channel

This channel splits the incoming stream into two identical streams. This code is from the textbook.

```
tee := func(done, in <-chan interface{}) (<-chan
interface{}, <-chan interface{}) {
    out1 := make(chan interface{})
    out2 := make(chan interface{})
    go func() {
        defer close(out1)
        defer close(out2)
        for val := range orDone(done, in) {
            var out1, out2 = out1, out2 // shadowing
            for i := 0; i < 2; i++ {
                select {
                case <-done:
                case out1 <- val:
                    out1 = nil
                case out2 <- val:
                    out2 = nil
                }
            }
        }
    }()
    return out1, out2
}
```

`bridge` Channel

This channel flattens a channel of channels into a channel.

```
bridge := func(done <-chan interface{}, chans <-chan <-
chan interface{}) <-chan interface{} {
    ret = make(chan interface{})
    go func() {
        defer close(ret)
        for chan := range orDone(done, chans) {
            for elem := range orDone(done, chan) {
                select {
                case <-done:
                case ret <- elem:
                }
            }
        }
    }()
    return ret
}
```

`buffer` Stage

Creates a buffer of the given size.

```
buffer := func(done <-chan interface{}, in <-chan
interface{}, bufferSize int) <-chan interface{} {
    ret = make(chan interface{}, bufferSize)
    go func() {
        defer close(ret)
        for val := range orDone(done, in) {
            select {
            case <-done:
            case ret <- val:
            }
    }()
    return ret
}
```

Queuing

Queuing is the acceptance of work into the pipeline despite the fact that the pipeline is not ready for more. This is usually implemented with buffered channels.

Runtime Performance and Uses of Queuing

Despite accepting more work, the amount of work that must be ultimately completed is the same, and the speed of the CPU is the same. The runtime performance is not different with queuing, thus. Instead, queuing is used so that the amount of time a goroutine (specifically a stage of the pipeline) is spent blocking is reduced. Queuing, in some sense, *decouples* certain stages of the pipeline a reasonable amount. The book states that queuing should be used in the following situations:

Batching requests in a stage saves time	An example of this is the chunking of requests to a file, which is why the `bufio` package exists. Adidtionally, this is useful for database transactions, calculating checksums, and allocating continguous space.
Negative Feedback Loop	A negative feedback loop, also called a downward-spiral or death-spiral, is when if there is a delay in a stage of the pipeline, there is more input for the pipeline. Since the time the pipeline takes to clear the input is related to the amount of input, this causes a downward spiral. For example, consider servers which bounce requests instead of storing requests in a queue and processing them one by one.

As such, only implement queuing at the entrance to the pipeline (negative feedback loop) or at stages where batching is more efficient.

Stable systems, Ingress, and Egress

Ingress is the rate at which work enters the system, and egress is the rate at which it exits the system. Stable systems are those in which the egress is equal to the ingress. The two unstable systems are when the ingress is more than the egress (which is a death spiral) and when it is less than the egress.

Little's Law	Little's Law requires sufficient sampling and determines the throughput of a pipeline. This is only applicable in stable systems. $$L = \lambda W$$
	L — the average number of units in the pipeline
	λ — the average arrival rate of units
	W — the average time a unit spends in the pipeline
Persistent Queues	If a pipeline panics with a queue in it, all requests stored in the queue are lost. Thus, one may want to avoid using a queue in this case, but if this is not possible, a persistent queue is used which is persisted and can be read later.
`context`	From the package `context`, `context.Context` is a data type that carries deadlines, timeouts, cancelation signals, and "request-scoped" values between processes and across API borders. An empty context can be created with `Background()` and `contexts` cannot be mutated so that any passed contexts will not be changed by the processes they were passed to. Instead, the child process can create its own `context` with the methods `WithCancel`, `WithTimeout`, and `WithDeadline`. More information at https://pkg.go.dev/context.
`context` values	Each `context` also carries values with it in key-value pairs. New contexts with different values can be created with `.WithValue(parent Context, key val any)`. Values can be accessed from a context with `.Value`. Since the key and value are `interface{}` and can accept any type, there is a loss of type safety, which is why it is recommended to have data-hiding.
When to use `context` values?	It is recommended to only use context values for request-scoped data, and not as optional parameters to functions. The textbook provides some heuristics on what can be considered "request-scoped data". 1. Crosses processes or APIs 2. Immutable 3. Simple types 4. Is not types with methods, but rather just data 5. Doesn't "drive" operations

4 Concurrency at Scale

Definition	Note

Errors	Errors are first-class citizens in Go, and they represent when your program cannot fulfill the requested instructions. There are two main types of errors that can come up in your program: bugs, and known edge cases. It is important to convey lots of information in known edge cases. In Go, `error` is an interface with one method: `Error()` which returns a `string`.

What should errors convey?

Errors should convey the following important information:

1. What caused the error
2. When and where the error happened
3. A "friendly user-facing message"
4. Resources for getting more information

Providing this information differentiates the known edge cases from the bugs in the program. Consider the following struct which implements `error`.

```
type WrappedError struct {
    Inner      error
    Message    string
    StackTrace string
    Misc       map[string]interface{}
}

func wrapError (err error, msgf string, msgArgs
...interface{}) WrappedError {
    return WrappedError {
        Inner:      err,
        Message:    fmt.Sprintf(msgf, msgArgs...),
        StackTrace: string(debug.Stack()),
        Misc:       make(map[string]interface{}),
    }
}

func (err WrappedError) Error() string {
    return err.Message
}
```

How to handle errors between modules?

It is suggested for errors to have a special type for each module, and any error which is not of the module's error type is malformed, or a bug. This means that if a module calls another, it should wrap the potential error in its own error type, while potentially adding more information.

Why should processes support timeouts?

There are many responses to a timeout, but why should processes timeout in the first place? Timeouts can:

1. Reduce the amount of pipeline saturation (if the request won't be repeated if timed out and there's not enough resources to store it)
2. Avoid stale data (data which is no longer relevant after a certain amount of time)
3. Avoid deadlocks (at the expense of potentially turning the deadlock into a livelock)