



GOPHERCON 2018

# Rethinking Classical Concurrency Patterns



Bryan C. Mills

[bcmills@google.com](mailto:bcmills@google.com)

GH: @bcmills

(A recording of this talk is available at <https://youtu.be/5zXAHh5tJqQ>.)

Hi, I'm Bryan Mills. I work on the Go Open Source Project at Google.

In this talk we're going to rethink some classical concurrency patterns you may have encountered in Go programs.

—

“You cannot flip a brain from *zero* to *one* simply by praising the *one*. You must start at the *zero*, extoll its virtues, explore its faults, exhort your listeners to look beyond it. To weigh the *zero* against the *one*, the listener must have both in mind together. Only when they have freely chosen the *one* will they abandon the *zero*.”

— [The Codeless Code, Case 196: Fee](#)

A static version of this presentation is available to users outside of Google:  
<https://drive.google.com/file/d/1nPdvhB0PutEJzdCq5ms6UI58dp50fcAN/view>



## INTRODUCTION



This talk covers two principles that you will hopefully find familiar.

We're going to apply them to some concurrency patterns that are hopefully also familiar.

The two principles relate to the Go concurrency primitives: goroutines and channels.



Start goroutines  
when you have  
concurrent work.



The first principle is, “Start goroutines when you have concurrent work.”



# Share by communicating.



The second principle is, “Share by communicating.”

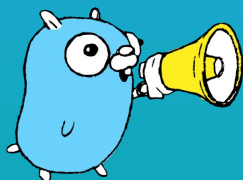


## INTRODUCTION



If you understand the implications of these two principles, I have nothing more to teach you in this talk.

But I've written about a hundred slides trying to understand them myself, and I'd appreciate your feedback.



## Rethinking Classical Concurrency Patterns

---

Introduction

---

Asynchronous APIs: Futures & Queues

---

Condition Variables

---

Worker Pools

---

Recap

---

First we'll examine the basic “asynchronous” patterns — futures and queues — which function as the concurrency primitives in some other languages.

Then we'll do a deep dive on condition variables. If you've been scuba diving, you know that the deeper you dive, the less time you can spend at depth: the slides will have a lot of code, but we won't spend much time on the details. The slides will be available after the talk and there are Playground links in the notes.

In the third section, we'll apply what we've learned to analyze the worker-pool pattern.



---

# Asynchronous APIs

---



First, let's talk about asynchronous APIs.

You've heard from Rob Pike that "Concurrency is not Parallelism". Concurrency is not Asynchronicity either.



DEFINITION

An **asynchronous API**  
returns to the caller  
before its result is ready.

ASYNCHRONOUS APIS



For the purpose of this talk, an asynchronous API is one that returns to the calling function early.

An asynchronous program is not *necessarily* concurrent: a program could call an asynchronous function and then sit idle waiting for the results.

Some poorly-written asynchronous programs do exactly that: a sequential chain of calls that each return early, wait for a result, and then start the next call.



## ASYNCHRONOUS CALLBACK: API

```
// Fetch immediately returns, then fetches the item and
// invokes f in a goroutine when the item is available.
// If the item does not exist,
// Fetch invokes f on the zero Item.
func Fetch(name string, f func(Item)) {
    go func() {
        [...]
        f(item)
    }()
}
```

This is not how we write Go. (You likely know that already.)

ASYNCHRONOUS APIS



Programmers coming to Go from certain languages, such as JavaScript, sometimes start with asynchronous callbacks, because that's the way they're used to structuring code.

The problems with asynchronous callbacks are well-described already. You may be thinking, "Why are we talking about callbacks? This is Go, and Go programmers know to use channels and goroutines instead!"

I agree: please don't use asynchronous callbacks, and we won't discuss them further. But that brings us to two other asynchronous patterns: Futures, and Producer–Consumer Queues.

—

<https://play.golang.org/p/jlrZshjnJ9z>

## FUTURE: API

```
// Fetch immediately returns a channel, then fetches
// the requested item and sends it on the channel.
// If the item does not exist,
// Fetch closes the channel without sending.
func Fetch(name string) <-chan Item {
    c := make(chan Item, 1)
    go func() {
        [...]
        c <- item
    }()
    return c
}
```

The Go analogue to a Future is a single-element buffered channel.

In the Future pattern, instead of returning the *result*, the function returns a *proxy object* that allows the caller to *wait* for the result at some later point.

You may also know “futures” by the name “async and await” in languages that have built-in support for the pattern.

The usual Go analogue to a Future is a single-element buffered channel that receives a single value, and it often starts a goroutine to compute that value. It's not exactly the conventional Future pattern, since we can only receive the value from the channel once, but the channel pattern seems to be more common than the function-based alternative.<sup>1</sup>

—

<sup>1</sup> See [FUTURE \(THE PRECISE WAY\)](#) in the backup slides.

[https://play.golang.org/p/v\\_IGf8tU3UT](https://play.golang.org/p/v_IGf8tU3UT)

## FUTURE: CALL SITE

### Yes:

```
a := Fetch("a")  
b := Fetch("b")  
consume(<-a, <-b)
```

### No:

```
a := <-Fetch("a")  
b := <-Fetch("b")  
consume(a, b)
```

To use Futures for concurrency, the caller must set up concurrent work **before** retrieving results.

ASYNCHRONOUS APIS



Callers of a Future-based API set up the work, *then* retrieve the results.

If they retrieve the results too early, the program executes sequentially instead of concurrently.

—

<https://play.golang.org/p/wVzp2Cou54I>

## PRODUCER-CONSUMER QUEUE: API

```
// Glob finds all items with names matching pattern
// and sends them on the returned channel.
// It closes the channel when all items have been sent.
func Glob(pattern string) <-chan Item {
    c := make(chan Item)
    go func() {
        defer close(c)
        for [...] {
            [...]
            c <- item
        }
    }()
    return c
}
```

A channel fed by one goroutine and read by another acts as a queue.

ASYNCHRONOUS APIS



A producer-consumer queue also returns a channel, but the channel receives any number of results and is typically unbuffered.

—

<https://play.golang.org/p/GpnC3KgwIT0>

## PRODUCER-CONSUMER QUEUE: CALL SITE

---


```
for item := range Glob("[ab]*") {  
    [...]  
}
```

---

The consumer of a producer-consumer queue is usually a range-loop.

The call site is a range-loop rather than a single receive operation.

Responsiveness.



Avoid blocking  
UI and network  
threads.

Now that we know what an asynchronous API looks like, let's examine the reasons we might want to use them.

Most other languages don't multiplex across OS threads, and kernel schedulers can be unpredictable. So some popular languages and frameworks keep all of the UI or network logic on a single thread. If that thread makes a call that blocks for too long, the UI becomes choppy, or network latency spikes.

Since calls to asynchronous APIs by definition don't block, they help to keep single-threaded programs responsive.

Efficiency.



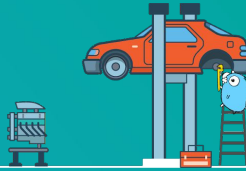
Reduce idle  
threads.

On some platforms, OS threads are — or historically have been — expensive.

Languages that don't multiplex over threads can use asynchronous APIs to keep threads busy, reducing the total number of threads — and context-switches — needed to run the program.

“

EFFECTIVE GO



**A goroutine [...] is lightweight**, costing little more than the allocation of stack space. And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required.

These first two benefits don't apply in Go.

The runtime manages threads for us, so there is no single “UI” or “network” thread to block, and we don't have to touch the kernel to switch goroutines. Kavya gave a lot more detail about that in her excellent talk this morning.

The runtime also resizes and relocates thread stacks as needed, so goroutine stacks can be very small — and don't need to fragment the address space with guard pages. Today, a goroutine stack starts around two kilobytes — half the size of the smallest amd64 pages.

—

[https://golang.org/doc/effective\\_go.html#goroutines](https://golang.org/doc/effective_go.html#goroutines)



Efficiency.



Reclaim  
stack frames.

An asynchronous call *may* allow the caller to return from arbitrarily many frames of the stack.

That frees up the memory *containing* those stack frames for other uses,<sup>1</sup> and allows the Go runtime to collect any other allocations that are only reachable from those frames.

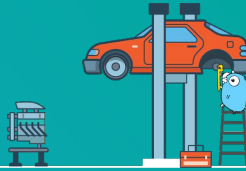
---

<sup>1</sup> From “Why Events Are A Bad Idea (for high-concurrency servers)”:

“Threaded systems typically face a tradeoff between risking stack overflow and wasting virtual address space on large stacks. Since event systems typically use few threads and unwind the thread stack after each event handler, they avoid this problem. To solve this problem in threaded servers, we propose a mechanism that will enable dynamic stack growth; we will discuss this solution in Section 4.”

“

THE GO F.A.Q.



Each variable in Go exists as long as there are references to it. **The storage location chosen by the implementation is irrelevant** to the semantics of the language.

Sometimes reclaiming stack frames is an optimization, but sometimes it isn't. Any reference that escapes its frame must be allocated in the heap, and *heap* allocations are more expensive in terms of CPU, memory, *and* cache.

Furthermore, the compiler can *already* prune out any stack allocations that it knows are unreachable: it can move large allocations to the heap, and the garbage collector can ignore dead references.

Finally, the benefit of this optimization depends on the *specific* call site: if the caller doesn't have a lot of data on the stack in the *first* place, then making the call asynchronous won't help much.

When we take all that into account, asynchronicity as an *optimization* is *very* subtle: it requires careful benchmarks for the impact *on specific callers*, and the impact may change or even reverse from one version of the runtime to the next. It's *not* the sort of optimization we want to build a stable API around!

—

[https://golang.org/doc/faq#stack\\_or\\_heap](https://golang.org/doc/faq#stack_or_heap)

Concurrency.



Initiate  
concurrent  
work.

A final benefit of asynchronous APIs *really does* apply in Go.

When an asynchronous function returns, the caller can immediately make *further* calls to start other concurrent work.

Concurrency can be especially important for network RPCs, where the CPU cost of a call is very low compared to its latency.

# Caller-side ambiguity

Unfortunately, that benefit comes at the cost of making the caller side of the API much less clear.



# Pop quiz!

ASYNCHRONOUS APIS



Let's look at some examples. Suppose that we come across an asynchronous call while we're debugging or doing a code review.

What can we infer about it from the call site?

## QUIZ TIME!



```
a := Fetch("a")
b := Fetch("b")
if err := [...] {
    return err
}
consume(<-a, <-b)
```

What happens if we return early?

ASYNCHRONOUS APIS



If we return without waiting for the futures to complete, how long will they continue using resources?

Might we start Fetches faster than we can retire them, and run out of memory?

## QUIZ TIME!



```
a := Fetch(ctx, "a")
b := Fetch(ctx, "b")
[...]
consume(<-a, <-b)
```

What happens in case of cancellation or error?

ASYNCHRONOUS APIS



Will Fetch keep using the passed-in context after it has returned?  
If so, what happens if we cancel it and then try to read from the channel?  
Will we receive a zero-value, some other sentinel value, or block?

## QUIZ TIME!



```
for result := range Glob("[ab]*") {  
    if err := [...] {  
        return err  
    }  
}
```

What happens if we return early?

If we return without draining the channel from Glob, will we leak a goroutine?



## QUIZ TIME!



```
for result := range Glob(ctx, "[ab]*") {  
    [...]  
}
```

What happens in case of cancellation or error?

ASYNCHRONOUS APIS



Will Glob keep using the passed-in Context as we iterate over the results?

If so, what happens if we cancel it? Will we still get results?

When, if ever, will the channel be closed?



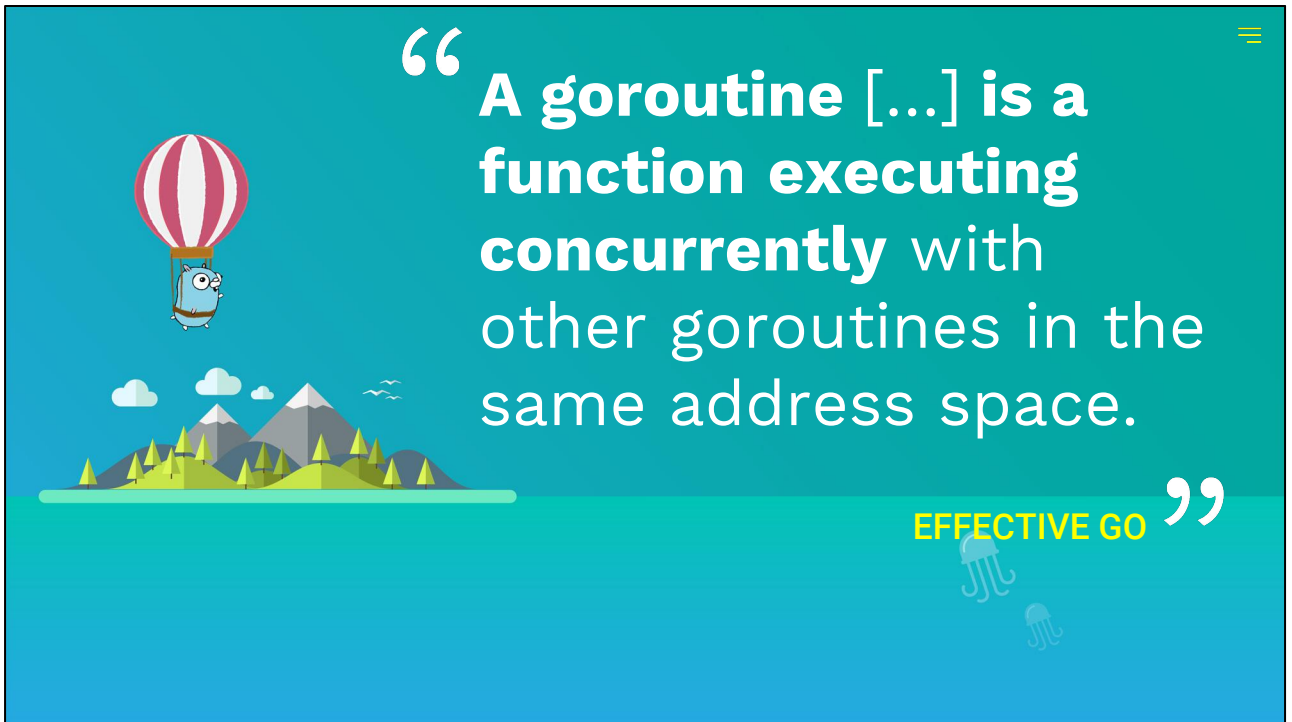
## ASYNCHRONOUS APIS



These asynchronous APIs raise a lot of questions, and to answer those questions we would have to go digging around in the documentation — if the answers are even there.

So let's rethink this pattern: how can we get the benefits of asynchronicity without this ambiguity?

Let's go back to the drawing board.



**Way** back to the drawing board. We're using goroutines to implement these asynchronous APIs, but what *is* a goroutine, anyway?

A goroutine is the execution of a function.

If we don't have another function to execute, a goroutine adds complexity without benefit.

—

[https://golang.org/doc/effective\\_go.html#goroutines](https://golang.org/doc/effective_go.html#goroutines)

A person is silhouetted against a hazy, mountainous landscape, standing on a rocky peak. The text is overlaid on this image.

# Start goroutines when **you** have concurrent work.



The benefit of asynchronicity is that it allows the caller to initiate other work. But how do we know that the caller even *has* any other work?

Functions like `Fetch` and `Glob` shouldn't need to know what other work their *callers* may be doing. That's not *their* job.

# ASYNCHRONOUS $\equiv$ SYNCHRONOUS

```
func Async(x In) (<-chan Out) {  
    c := make(chan Out, 1)  
    go func() {  
        c <- Synchronous(x)  
    }()  
    return c  
}
```

```
func Synchronous(x In) Out {  
    c := Async(x)  
    return <-c  
}
```

In Go, synchronous and asynchronous APIs are interchangeable.

ASYNCHRONOUS APIS



In languages without threads or coroutines, asynchronous APIs are viral: if we can't execute *function calls* concurrently, any function that *may* be concurrent *must* be asynchronous.<sup>1</sup>

In contrast, in Go it's easy to wrap an asynchronous API to make it synchronous, or vice-versa. We can write the clearer API, and adapt it as needed at the call site.

—

<https://play.golang.org/p/FxSsaTTolHe>

<sup>1</sup> See <http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/>.



# Add concurrency on the **caller side** of the API.



If we keep the API synchronous, we may need to add concurrency at the call site.

## CALLER-SIDE CONCURRENCY: SYNCHRONOUS API



```
// Fetch returns the requested item.  
func Fetch(context.Context, name string) (Item, error) {  
    [...]  
}
```

Synchronous APIs usually require much less documentation.

ASYNCHRONOUS APIS



Consider a synchronous version of our Fetch function.  
The cancellation and error behavior is obvious from the function signature: we don't need extra documentation for it now.

## CALLER-SIDE CONCURRENCY: VARIABLES

```
var a, b Item
g, ctx := errgroup.WithContext(ctx)
g.Go(func() (err error) {
    a, err = Fetch(ctx, "a")
    return err
})
g.Go(func() (err error) {
    b, err = Fetch(ctx, "b")
    return err
})
err := g.Wait()
[...]
```

The caller can invoke synchronous functions concurrently, and often won't need to use channels at all.

ASYNCHRONOUS APIS



The caller can use whatever pattern they like to add concurrency. In many cases, they won't even need to go through channels, so the questions about channel usage won't arise. Here, we're using the [golang.org/x/sync/errgroup](https://golang.org/x/sync/errgroup) package and writing the results directly into local variables.

—

[https://play.golang.org/p/zbP4cko\\_rTI](https://play.golang.org/p/zbP4cko_rTI)





# Make concurrency an **internal detail**.



As long as we present a simple, synchronous API to the caller, they don't need to care *how* many concurrent calls its implementation makes.

## INTERNAL CONCURRENCY: SYNCHRONOUS API

---



```
// Glob finds all items with names matching pattern.  
func Glob(ctx context.Context, pattern string) ([]Item, error) {  
    [...]  
}
```

---

A synchronous API can have a concurrent implementation.

ASYNCHRONOUS APIS



For example, consider a synchronous version of our Glob function.

# INTERNAL, CALLER-SIDE CONCURRENCY: CHANNELS

```
func Glob([...] []Item, error) {
    [...] // Find matching names.
    c := make(chan Item)
    g, ctx :=
        errgroup.WithContext(ctx)
    for _, name := range names {
        name := name
        g.Go(func() error {
            item, err :=
                Fetch(ctx, name)
            if err == nil {
                c <- item
            }
            return err
        })
    }
}
```

```
go func() {
    err = g.Wait()
    close(c)
}()

var items []Item
for item := range c {
    items = append(items, item)
}
if err != nil {
    return nil, err
}
return items, nil
}
```

Keeping the channel local to the caller function makes its usage much easier to see.

ASYNCHRONOUS APIS



Internally it can Fetch all of its items concurrently and stream them to a channel, but the caller doesn't need to know that.

And because the channel is local to the function, we can see both the sender and receiver locally.

That makes the answers to our channel questions obvious:

- Since the send is unconditional, the receive loop must drain the channel.
- In case of error, the err variable is set and the channel is still closed.

—

<https://play.golang.org/p/XU1kBmfkKI3>

# Concurrency is not Asynchronicity.



In Go, synchronous and asynchronous APIs are equally expressive.

We can call synchronous APIs concurrently, and they're *clearer* at the call site.

We don't need to pay the *cost* of *asynchronicity* to get the *benefits* of *concurrency*.

# Condition Variables

Condition variables — our next classical pattern — are part of a larger concurrency pattern called *Monitors*, but the phrase “condition variable” appears in the Go standard library, whereas “monitor” (in this sense) does not, so that's what this section is called.

The concept of monitors dates to 1973,<sup>1</sup> and condition variables to 1974,<sup>2</sup> so this is a fairly old pattern.

---

<sup>1</sup> Hansen, *Shared Classes* (1973)

<sup>2</sup> Hoare, *Monitors: an operating system structuring concept* (1974)

## CONDITION VARIABLE: SETUP

```
type Queue struct {  
    mu sync.Mutex  
    items []Item  
    itemAdded sync.Cond  
}  
  
func NewQueue() *Queue {  
    q := new(Queue)  
    q.itemAdded.L = &q.mu  
    return q  
}
```

A condition variable is associated with a sync.Locker (e.g., a Mutex).

CONDITION VARIABLES



First, a refresher on condition variables. Let's look at a simple example: an unbounded queue of items.

A Go condition variable must be associated with a mutex or other Locker.

—

<https://play.golang.org/p/8m1i4lgealw>

## CONDITION VARIABLE: WAIT AND SIGNAL

```
func (q *Queue) Get() Item {
    q.mu.Lock()
    defer q.mu.Unlock()
    for len(q.items) == 0 {
        q.itemAdded.Wait()
    }
    item := q.items[0]
    q.items = q.items[1:]
    return item
}
```

```
func (q *Queue) Put(item Item) {
    q.mu.Lock()
    defer q.mu.Unlock()
    q.items = append(q.items, item)
    q.itemAdded.Signal()
}
```

Wait atomically unlocks the mutex and suspends the goroutine.  
Signal locks the mutex and wakes up the goroutine.

CONDITION VARIABLES



The two basic operations on condition variables are Wait, and Signal.

Wait atomically unlocks the mutex and suspends the calling goroutine.

Signal wakes up a waiting goroutine, which relocks the mutex before proceeding.

In our *queue*, we can use Wait to block on the availability of enqueued items, and Signal to indicate when another item has been added.

## CONDITION VARIABLE: BROADCAST

```
type Queue struct {  
    [...]   
    closed bool  
}  
  
func (q *Queue) Close() {  
    q.mu.Lock()  
    defer q.mu.Unlock()  
    q.closed = true  
    q.cond.Broadcast()  
}
```

Broadcast usually communicates events that affect all waiters.

Broadcast wakes up *all* of the waiting goroutines instead of just one.

Broadcast is usually for events that affect all waiters, such as marking the end of the queue...



## CONDITION VARIABLE: BROADCAST

```
func (q [...]) GetMany(n int) []Item {
    q.mu.Lock()
    defer q.mu.Unlock()
    for len(q.items) < n {
        q.itemAdded.Wait()
    }
    items := q.items[:n:n]
    q.items = q.items[n:]
    return items
}
```

```
func (q *Queue) Put(item Item) {
    q.mu.Lock()
    defer q.mu.Unlock()
    q.items = append(q.items, item)
    q.itemAdded.Broadcast()
}
```

Since we don't know **which** of the GetMany calls may be ready after a Put, we can wake them **all** up and let them decide.

CONDITION VARIABLES



...however, it is sometimes<sup>1</sup> used to wake up *some* waiter or waiters when we don't *know* exactly which are eligible.

Here, we've changed Get to GetMany. After a Put, one of the waiting GetMany calls may be ready to complete, but Put has no way of knowing which one to wake, so it must wake all of them.

—

<sup>1</sup> These sloppy wakeups, as it turns out, are pretty much the *original* use-case for the Broadcast operation.

See Lamson & Redell, *Experience with Processes and Monitors in Mesa*, 1980.



That's the basic operations.

Condition variables have a lot of different *use-cases* that we'll want to focus on one at a time, but the downsides are similar for all of them, so we'll start there.

## CONDITION VARIABLES

---

# Spurious wakeups

For events that aren't really global, Broadcast may wake up too many waiters. For example, one call to Put wakes up *all* of the GetMany callers, even though at most *one* of them will actually be able to complete.

Even Signal can result in spurious wakeups: if Put used Signal instead of Broadcast, it could wake up a caller that is not yet ready *instead of* one that is. If it does that repeatedly, it could strand items in the queue without corresponding wakeups.

If we're very careful, we can minimize or avoid spurious wakeups — but that generally adds even *more* complexity and subtlety to the code.

## CONDITION VARIABLES

---

# Forgotten signals

If we prune out spurious signals *too* aggressively, we risk going too far and dropping some that are actually *necessary*.

And since the condition variable decouples the signal from the data, it's easy to add some *new* code to update the *data* and forget to signal the *condition*.

## CONDITION VARIABLES

---

# Starvation

Even if we don't forget a signal, if the waiters are not *uniform*, the pickier ones can *starve*.

Suppose that we have one call to `GetMany(3000)`, and one caller executing `GetMany(3)` in a tight loop. The two waiters will be about equally likely to wake up, but the `GetMany(3)` call will be able to consume three items, whereas `GetMany(3000)` won't have enough ready. The queue will remain drained and the larger call will block forever.

If we happened to notice the starvation problem ahead of time, we could add an explicit wait-queue to avoid starvation, but that again makes the code more complex.

# Unresponsive cancellation

The whole point of condition variables is to put a goroutine to sleep while we wait for something to happen.


But while we're waiting for the condition, we may miss some *other* event that we ought to notice too. For example, the caller might decide they don't wait to wait that long and cancel a passed-in Context, expecting us to notice and return more-or-less immediately.

Unfortunately, condition variables *only* let us wait for events associated with their *own* mutex, so we can't select on a *condition* and a *cancellation* at the same time.<sup>1</sup>

Even if the caller cancels, *our call* will block until the next time the condition is signalled.

—

<sup>1</sup> See proposal [#16620](#).



“Concurrent programming [...] is made difficult by the subtleties required to implement correct access to shared variables. Go encourages a different approach in which **shared values are passed around on channels** [...].”

EFFECTIVE GO ”

Fundamentally, condition variables rely on communicating by sharing memory: they signal that a change has occurred, but leave it up to the signalled goroutine to check *other* shared variables to figure out what.

On the other hand, the Go approach is to *share by communicating*.



# Share by communicating.

Share by communicating. What does that even mean here?

Let's look at the use-cases for condition variables and rethink them in terms of *communication*. Perhaps we'll spot a pattern.



Sharing.



Release  
resources.

A signal or broadcast on a condition variable tells the waiters “something has changed”.

Often, that something is the availability of a shared resource, such as a connection or buffer in a pool.

## CONDITION VARIABLE: RESOURCE POOL

```
type Pool struct {  
    mu      sync.Mutex  
    cond    sync.Cond  
    numConns, limit int  
    idle    []net.Conn  
}  
  
func NewPool(limit int) *Pool {  
    p := &Pool{limit: limit}  
    p.cond.L = &p.mu  
    return p  
}  
  
func (p *Pool) Release(c net.Conn) {  
    p.mu.Lock()  
    defer p.mu.Unlock()  
    p.idle = append(p.idle, c)  
    p.cond.Signal()  
}  
  
func (p *Pool) Hijack(c net.Conn) {  
    p.mu.Lock()  
    defer p.mu.Unlock()  
    p.numConns--  
    p.cond.Signal()  
}
```

Signal when the set of resources changes.

Let's look at a concrete example.

Our resources for this example will be net.Conns in a pool, and we'll start with the condition-variable version for reference.

We've got a limit on the total number of connections, plus a pool of idle connections, and a condition variable that tells us when the set of connections changes.

When we're done with a connection, we can either release it back into the idle pool, or hijack it so that it no longer counts against the limit.

—

<https://play.golang.org/p/cpDHxvzTMSp>

## CONDITION VARIABLE: RESOURCE POOL

```
func (p *Pool) Acquire() (
    net.Conn, error) {

    p.mu.Lock()
    defer p.mu.Unlock()
    for len(p.idle) == 0 &&
        p.numConns >= p.limit {
        p.cond.Wait()
    }

    if len(p.idle) > 0 {
        c := p.idle[len(p.idle)-1]
        p.idle =
            p.idle[:len(p.idle)-1]
        return c, nil
    }

    c, err := dial()
    if err == nil {
        p.numConns++
    }
    return c, err
}
```

Loop until a resource is available, then extract it from the shared state.

To acquire a connection, we wait until we have an idle connection to reuse or are under the limit.



Share **resources**  
by communicating  
the **resources**.



Now let's rethink. Let's share *resources* by communicating the *resources* themselves.



# Resource limits are **resources** too!



And the *limit* is a resource too: in particular, an available slot *toward* the limit is a thing we can *consume*.

“

**A buffered channel can be used like a semaphore [...].**

The capacity of the channel buffer limits the number of simultaneous calls to process.

EFFECTIVE GO ”

Effective Go has a hint for that. It mentions *another* classical concurrency pattern: the **semaphore**, which was described by Dijkstra in the early 1960s.

## COMMUNICATION: RESOURCE POOL

```
type Pool struct {
    sem chan token
    idle chan net.Conn
}
type token struct{}

func NewPool(limit int) *Pool {
    sem := make(chan token, limit)
    idle :=
        make(chan net.Conn, limit)
    return &Pool{sem, idle}
}

func (p *Pool) Release(c net.Conn) {
    p.idle <- c
}

func (p *Pool) Hijack(c net.Conn) {
    <-p.sem
}
```

Channel operations combine synchronization, signalling, and data transfer.

CONDITION VARIABLES



So we'll have a channel for the *limit tokens*, and one for the idle-connection resources. A send on the semaphore channel will communicate that we have consumed a slot toward the limit, and the idle channel will communicate the actual *connections* as they are idled.

Now Release and Hijack have become trivial: Release literally puts the connection back in the pool, and Hijack releases a token from the semaphore. They've dropped from four-line bodies to one line each: instead of locking, storing the resource, signalling, and unlocking, they simply communicate the resource.

If we really wanted to, we *could* use a *single* channel for this instead of two: we could use a nil net.Conn to represent "permission to create a new connection". Personally, I think the code is clearer with separate channels.

—

[https://play.golang.org/p/j\\_OmiKuyWo8](https://play.golang.org/p/j_OmiKuyWo8)

## COMMUNICATION: RESOURCE POOL



```
func (p *Pool) Acquire(ctx context.Context) (net.Conn, error) {  
    select {  
        case conn := <-p.idle:  
            return conn, nil  
        case p.sem <- token{}:  
            conn, err := dial()  
            if err != nil {  
                <-p.sem  
            }  
            return conn, err  
        case <-ctx.Done():  
            return nil, ctx.Err()  
    }  
}
```

When we block on communicating, others can **also** communicate with us:  
for example, to cancel the call.

CONDITION VARIABLES



Acquire ends up a lot simpler too — and even cancellation is just one more case in the select.



Synchronization.



Indicate the  
existence of  
new data.

Conditions can *also* indicate the existence of new *data* for processing.

## CONDITION VARIABLE: ONE ITEM PER SIGNAL

```
func (q *Queue) Get() Item {
    q.mu.Lock()
    defer q.mu.Unlock()
    for len(q.items) == 0 {
        q.itemAdded.Wait()
    }
    item := q.items[0]
    q.items = q.items[1:]
    return item
}
```

```
func (q *Queue) Put(item Item) {
    q.mu.Lock()
    defer q.mu.Unlock()
    q.items = append(q.items, item)
    q.itemAdded.Signal()
}
```

Each Put wakes up at most one Get.

Let's go back to our queue example. For the single-item Get and Put, a signal indicates the availability of an *item* of data...

## CONDITION VARIABLE: ZERO OR ONE ITEMS?

```
func ([...] GetMany(n int) []Item {  
    q.mu.Lock()  
    defer q.mu.Unlock()  
    for len(q.items) < n {  
        q.itemAdded.Wait()  
    }  
    items := q.items[:n:n]  
    q.items = q.items[n:]  
    return items  
}
```

```
func (q *Queue) Put(item Item) {  
    q.mu.Lock()  
    defer q.mu.Unlock()  
    q.items = append(q.items, item)  
    q.itemAdded.Broadcast()  
}
```

Each Put wakes up all GetMany, but at most one will consume the item.

...while in the *GetMany* version it indicates *potential* availability of an item *that some other goroutine may have already consumed*.

That *imprecise targeting* is the cause of both *spurious wakeups* and starvation.

# Share **data** by communicating the **data**.

To avoid spurious wakeups, we should signal only the goroutine that will actually *consume* the data. But if we know which goroutine will consume the data, we may as well send the data along too.

Sending the data makes it much easier to see whether the signal is spurious: if we resend the exact *same* data to the same receiver, or if the caller explicitly *ignores* the channel-receive — for example, by executing a continue in a range loop — then we probably didn't need to send it in the first place.

Sending the data also makes signals harder to forget: we'll very likely notice if we compute data and then don't send it anywhere, although we do *still* have to be careful to send it to *all* interested receivers.



# Metadata are **data** too!



So how do we identify the right receivers?

The information about “who needs which data” is *a/so* data. We can communicate that too!

# COMMUNICATION: QUEUE



```
type Queue struct {  
    items chan []Item // non-empty slices only  
    empty chan bool    // holds true if the queue is empty  
}  
  
func NewQueue() *Queue {  
    items := make(chan []Item, 1)  
    empty := make(chan bool, 1)  
    empty <- true  
    return &Queue{items, empty}  
}
```

The empty channel conveys metadata about the items channel:  
empty indicates that no goroutine is sending to items.

CONDITION VARIABLES



We'll start with the single-item Get.

We need two channels: one to communicate the *items*, and another to communicate whether any items even *exist*.

Both will have a buffer size of one: the items channel functions like a mutex, while the empty channel is like a one-token semaphore.

—

<https://play.golang.org/p/uvx8vFSQ2f0>

## COMMUNICATION: QUEUE

```
func (q *Queue) Get() Item {  
    items := <-q.items  
  
    item := items[0]  
    items = items[1:]  
    if len(items) == 0 {  
        q.empty <- true  
    } else {  
        q.items <- items  
    }  
    return item  
}
```

```
func (q *Queue) Put(item Item) {  
    var items []Item  
    select {  
        case items = <-q.items:  
        case <-q.empty:  
        }  
    items = append(items, item)  
    q.items <- items  
}
```

Each waiter consumes the data they need,  
and communicates any remaining data back to the channels.

CONDITION VARIABLES



This time, we really *do* need the two separate channels: Put needs to know when there are *no* items so that it can start a new slice, but Get wants only *non-empty* items.

## COMMUNICATION: QUEUE CANCELLATION

```
func (q *Queue) Get(ctx context.Context) (Item, error) {
    var items []Item
    select {
        case <-ctx.Done():
            return 0, ctx.Err()
        case items = <-q.items:
    }

    item := items[0]
    if len(items) == 1 {
        q.empty <- true
    } else {
        q.items <- items[1:]
    }
    return item, nil
}
```

To support cancellation, select on operations that block indefinitely.  
Operations that we know will not block do not need to select.

CONDITION VARIABLES



To support *cancellation* in `Get`, all we have to do is move the initial channel-receive into a select statement.

We don't need to select on the sends at the end because we *know* they won't block: when we received the *items*, we also received the information that *our* goroutine *owns* those items.



## SPECIFIC COMMUNICATION: QUEUE

```
type waiter struct {  
    n    int  
    c    chan []Item  
}
```

```
type state struct {  
    items []Item  
    wait  []waiter  
}
```

```
type Queue struct {  
    s chan state  
}
```

```
func NewQueue() *Queue {  
    s := make(chan state, 1)  
    s <- state{}  
    return &Queue{s}  
}
```

To wait for specific data, the waiters communicate **their needs**.

CONDITION VARIABLES



That's Get with one item. What about *GetMany*?

To figure out whether we should wake a *GetMany* caller, we need to know how many items it *wants*. Then we need a channel on which we can *send* those items to that particular caller.

We'll put the items and the metadata together in one "queue state" struct and, just for good measure, we'll share that state by communicating it *too*. A channel with a one-element buffer functions much like a selectable mutex.

—

[https://play.golang.org/p/rzSXpophC\\_p](https://play.golang.org/p/rzSXpophC_p)

## SPECIFIC COMMUNICATION: QUEUE

```
func ([...] GetMany(n int) []Item {
    s := <-q.s
    if len(s.wait) == 0 &&
        len(s.items) >= n {
        items := s.items[:n:n]
        s.items = s.items[n:]
        q.s <- s
        return items
    }
    c := make(chan []Item)
    s.wait =
        append(s.wait, waiter{n, c})
    q.s <- s

    return <-c
}
```

```
func (q *Queue) Put(item Item) {
    s := <-q.s
    s.items = append(s.items, item)
    for len(s.wait) > 0 {
        w := s.wait[0]
        if len(s.items) < w.n {
            break
        }
        w.c <- s.items[w.n:w.n]
        s.items = s.items[w.n:]
        s.wait = s.wait[1:]
    }
    q.s <- s
}
```

Put sends to the next waiter if — and only if — it has enough items for **that** waiter.

CONDITION VARIABLES



To get a *run* of items, we first check the current state for sufficient items. If there aren't enough, we add an entry to the metadata.

To put an item to the queue, we append it to the current state and check the metadata to see whether that makes *enough* items to send to the next waiter. When we don't have enough items *left*, we'll stop sending items and send back the updated state.

Since *all* of the communication occurs on channels, it is *possible* to plumb in cancellation here too — but that's too much code to fit into this talk, so we'll leave it as an exercise.

Coordination.



Mark  
transitions.

Broadcast on a condition may signal a transition from one state to another: for example, it may indicate that the program has *finished* loading its initial configuration, or that a communication stream has been *terminated*.

## CONDITION VARIABLE: REPEATING TRANSITION

```
type Idler struct {
    mu    sync.Mutex
    idle  sync.Cond
    busy  bool
    idles int64
}

func (i *Idler) AwaitIdle() {
    i.mu.Lock()
    defer i.mu.Unlock()
    idles := i.idles
    for i.busy && idles == i.idles {
        i.idle.Wait()
    }
}

func (i *Idler) SetBusy(b bool) {
    i.mu.Lock()
    defer i.mu.Unlock()
    wasBusy := i.busy
    i.busy = b
    if wasBusy && !i.busy {
        i.idles++
        i.idle.Broadcast()
    }
}

func NewIdler() *Idler {
    i := new(Idler)
    i.idle.L = &i.mu
    return i
}
```

Since awakened goroutines must wait to reacquire the mutex, the waiter must be robust to subsequent changes.

CONDITION VARIABLES



One simple state transition is the transition from “busy” to “idle”.

Using *condition variables*, we need to store the state *explicitly*.

You might think we would only need to store the current state — the “busy” boolean — but that turns out to be a very *subtle* decision. If `AwaitIdle` looped only until it saw a *non-busy* state, it would be possible to transition from busy to idle and back before `AwaitIdle` got the chance to check, and we would miss short idle events.

Go's condition variables — unlike *pthread* condition variables — don't have spurious wakeups, so in theory we could return from `AwaitIdle` unconditionally after the first `Wait` call.

However, it's common for condition-based code to *intentionally* over-signal — for example, as a workaround for an undiagnosed deadlock — so to avoid introducing subtle problems later it's best to keep the code robust to spurious wakeups.

Instead, we can track the cumulative count of events, and wait until *either* we catch the idle event in progress or observe its effect on the counter.

—

<https://play.golang.org/p/HYiRtJcyaX9>

# Share **completion** by **completing** communication.

We can avoid the double-state-transition race entirely by *communicating* the transition instead of *signalling* it — *and* we can plumb in cancellation to boot.

We can broadcast a state transition by closing a channel.

There's a nice symmetry to that: a state transition marks the *completion* of the previous state, and closing a channel marks the completion of *communication* on the channel.

## COMMUNICATION: REPEATING TRANSITION

```
type Idler struct {
    next chan chan struct{}
}
func (i *Idler) AwaitIdle(
    ctx context.Context) error {
    idle := <-i.next
    i.next <- idle
    if idle != nil {
        select {
            case <-ctx.Done():
                return ctx.Err()
            case <-idle:
        }
    }
    return nil
}
```

```
func (i *Idler) SetBusy(b bool) {
    idle := <-i.next
    if b && (idle == nil) {
        idle = make(chan struct{})
    } else if !b && (idle != nil) {
        close(idle) // Idle now.
        idle = nil
    }
    i.next <- idle
}

func NewIdler() *Idler {
    next := make(chan chan struct{}, 1)
    next <- nil
    return &Idler{next}
}
```

Allocate the channel to be closed when the event starts,  
or when the first waiter appear.

CONDITION VARIABLES



Here is an example showing how that fits together.

SetBusy allocates a new channel on the idle-to-busy transition, and closes the previous channel — if any — on the busy-to-idle transition.

—

<https://play.golang.org/p/4WG59Juxjch>

Fanout.



Broadcast  
events.

Broadcast may also signal *ephemeral* events, such as configuration reload requests.



# Events can be data.



We can treat broadcast events like data updates, and send them individually to each interested subscriber.<sup>1</sup>

The os/signal package in the standard library takes that approach, so that waiters can receive multiple events on the same channel.

—

<sup>1</sup> See [COMMUNICATION: EVENT FANOUT \(CHANNELS\)](#) in the backup slides.





# Events can be completions.



Alternately, we can treat the event as the completion of the “*hasn't happened yet*” state — and indicate it by closing a channel.<sup>1</sup>

That *typically* results in fewer channel allocations, but when we have closed the channel we can't communicate any additional *data* about the event.

—

<sup>1</sup> See [COMMUNICATION: EVENT FANOUT \(CLOSURE\)](#).



Share a **thing**  
by communicating  
**the thing.**

GO

Did you spot the *pattern* in this section?

When we “share by communicating”, we should communicate *the things that we want to share*, not just messages *about* them.

# Worker Pools



We started with asynchronous patterns, which deal with *goroutines*. Then, we looked at condition variables, which sometimes deal with *resources*.

Now, let's put them together. The Worker Pool is a pattern that treats a set of *goroutines* as *resources*.<sup>1</sup>

Just a note on terminology: in other languages the pattern is usually called a “thread pool”,<sup>2</sup> but in Go we're working with goroutines, so we just call them workers.

<sup>1</sup> Specific Go examples in the wild include:

- [https://golang.org/doc/effective\\_go.html#channels](https://golang.org/doc/effective_go.html#channels) (2009)
- <https://gobyexample.com/worker-pools> (2012)
- <http://marcio.io/2015/07/handling-1-million-requests-per-minute-with-golang/> (2015)
- <https://brandur.org/go-worker-pool> (2016)
- <https://medium.freecodecamp.org/million-websockets-and-go-cc58418460bb#d801> (2017)
- <https://rodaine.com/2018/08/x-files-sync-golang/#controlling-concurrent-access-with-semaphore> (2018)

<sup>2</sup> I've had some trouble tracking down the origin of this term: it's attested by 1999, in Doug Lea's *Concurrent Programming in Java, second edition*, but I suspect it's older

than that. (The phrase “thread pool” appears much earlier — the first reference I could find is in *The 7th International Conference on Distributed Computing Systems, Berlin, West Germany, September 21-25, 1987* — but it's not clear to me when it began to refer to this specific pattern.)

## WORKER POOL

Start the workers:

```
work := make(chan Task)
for n := limit; n > 0; n-- {
    go func() {
        for task := range work {
            perform(task)
        }
    }()
}
```

Send the work:

```
for _, task := range hugeSlice {
    work <- task
}
```

A fixed pool of goroutines receive and perform tasks from a channel.  
Another goroutine sends the work to the channel.

WORKER POOLS



In the Worker Pool pattern, we start up a fixed number of “worker” goroutines that each read and perform tasks from a channel.

*Another* goroutine — often the same one that *started* the workers — sends the tasks to the workers. The sender blocks until a worker is available to receive the next task.

Efficiency.



Distribute work  
across threads

In languages with heavyweight threads, the worker pool pattern allows us to reuse threads for multiple tasks, avoiding the overhead of creating and destroying threads for small amounts of work.

“

EFFECTIVE GO



## Goroutines are multiplexed onto multiple OS threads [...].

Their design hides many of the complexities of thread creation and management.

That benefit doesn't apply in Go.

Remember Kavya's talk? The Go runtime *already* distributes goroutines across threads. It *knows* how many system threads we have available and can reschedule goroutines efficiently when they block.

Flow control.



Limit  
work in flight.

The benefit that worker pools *do* provide in Go is to limit the amount of concurrent work in flight.

If each task needs to use some *limited* resource — such as file handles, network bandwidth, or even a nontrivial amount of RAM — a worker pool can bound the peak resource usage of the program.



## WORKER POOLS: COST

---

# Worker lifetimes

The simple worker pool I showed earlier has a problem. It leaks the workers forever.

If the API we're implementing is synchronous — and remember what we said *before* about asynchronous APIs? — or, if we want to be able to *reset* the worker state for a unit-test, then we need to be able to shut down the workers *and* know when they've finished.

## WORKER POOL: CLEANING UP

Start the workers:

```
work := make(chan Task)
var wg sync.WaitGroup
for n := limit; n > 0; n-- {
    wg.Add(1)
    go func() {
        for task := range work {
            perform(task)
        }
        wg.Done()
    }()
}
```

A `sync.WaitGroup` tracks the worker goroutines.

Cleaning up the workers adds a bit more boilerplate to the pattern.

First we'll add a `WaitGroup` to track the goroutines...

—

<https://play.golang.org/p/zz7LkM6WqX0>

## WORKER POOL: CLEANING UP

Send the work:      `for _, task := range hugeSlice {  
                          work <- task  
                      }`

Signal end of work:      `close(work)`

Wait for completion:      `wg.Wait()`

---

After we've sent all the work,  
we close the work channel to trigger the workers to return.

...then, after we send the work, we can close the channel and wait for the workers to exit.

# Idle workers



But we may have another problem: even if we remember to clean up the workers when we're done, we may leave them idle for a long time — especially toward the end of work — and “the end of work” may be *forever* if we've accidentally deadlocked something.

Assuming we've remembered to clean up, if we have a deadlock our tests will *hang* instead of *passing*. So at least we can get a goroutine dump to help debug, right?

[illegible]

Can you spot the cause of this deadlock?



That will make the *interesting* goroutines a lot harder to find — especially if our program happens to be a *large* service implemented with several *different* pools. It will also be a problem if we want to use a goroutine dump to debug other issues, such as crashes or memory leaks.

*This* is an actual goroutine dump from a test failure involving a deadlock between a worker pool and the goroutine that feeds it. This one is just a toy: there's only one pool, and it only has a hundred workers. Even so, one of the goroutines involved in the deadlock ended up all the way at the bottom of page four — and these are long pages!

---

<https://play.golang.org/p/zz7LkM6WqX0>

## QUIZ TIME!



```
2 kilobytes  
x N workers  
= ?
```

Goroutines are lightweight, **not** free.



And remember, goroutines are lightweight, *not* free: those idle workers still have a resource cost, too, and for large pools that cost may not be completely negligible.



WORKER POOLS



So let's rethink this pattern: how can we get the same *benefits* as worker pools without the complexity of *workers* and their lifetimes?

A person is silhouetted against a hazy, mountainous landscape, standing on a rocky peak. The text is overlaid on the left side of the image.

# Start goroutines when you have concurrent work **to do now.**



We want to start the goroutines only when we're *actually* ready to do the work, and let them exit as *soon* as the work is done.

Let's do just that part and see where we end up.



## WAITGROUP: DISTRIBUTING (UNLIMITED) WORK

Start the work:

```
var wg sync.WaitGroup
for _, task := range hugeSlice {
    wg.Add(1)
    go func(task Task) {
        perform(task)
        wg.Done()
    }(task)
}
```

Wait for completion:

```
wg.Wait()
```

The Go runtime distributes tasks across OS threads.  
A sync.WaitGroup tracks the worker goroutines.

If we *only* need to distribute work across threads, we can omit the worker pool *and* its channel and use only the WaitGroup.

This code is a *lot* simpler, but now we need to figure out how to *limit* the in-flight work again.



Share **resources**  
by communicating  
the **resources.**



We already *have* a pattern for that. Remember, *limits* are *resources*!

## SEMAPHORE CHANNEL: LIMITING CONCURRENCY

Start the work:

```
sem := make(chan token, limit)
for _, task := range hugeSlice {
    sem <- token{}
    go func(task Task) {
        perform(task)
        <-sem
    }(task)
}
```

Wait for completion:

```
for n := limit; n > 0; n-- {
    sem <- token{}
}
```

A semaphore channel takes the place of the WaitGroup.

WORKER POOLS



So let's use the semaphore-channel pattern from the *last* section.

The semaphore example in Effective Go acquires a token *inside* the goroutine, but we'll acquire it *earlier* — right where we had the WaitGroup Add call. We don't want a lot of goroutines sitting around doing nothing, and this way we have only *one* idle goroutine instead of many. Recall that we acquire this semaphore by sending a token, and we release it by discarding a token.

This semaphore fits pretty nicely in place of the WaitGroup, and that's no accident: sync.WaitGroup is very similar to a semaphore.<sup>1</sup> The only *major* difference is that the WaitGroup allows further Add calls during Wait, whereas the wait loop on our semaphore channel does not. Fortunately, that doesn't usually matter, as in this case.

---

<sup>1</sup> [#20687](#) notwithstanding.

## SEMAPHORE CHANNEL: INVERTED WORKER POOL?

```
sem := make(chan token, limit)
for _, task := range hugeSlice {
    sem <- token{}
    go func(task Task) {
        perform(task)
        <-sem
    }(task)
}

for n := limit; n > 0; n-- {
    sem <- token{}
}
```

```
work := make(chan Task)
for n := limit; n > 0; n-- {
    go func() {
        for task := range work {
            perform(task)
        }
    }()
}

for _, task := range hugeSlice {
    work <- task
}
```

The semaphore pattern has the same number of lines as the worker pool, but no leaked workers!

WORKER POOLS



Remember our first worker pool with the two loops, and how we leaked all those idle workers forever?

If you look *carefully*, these are the *same* two loops swapped around – we've eliminated the leak *without* adding any *net* lines of code.



---

# Recap



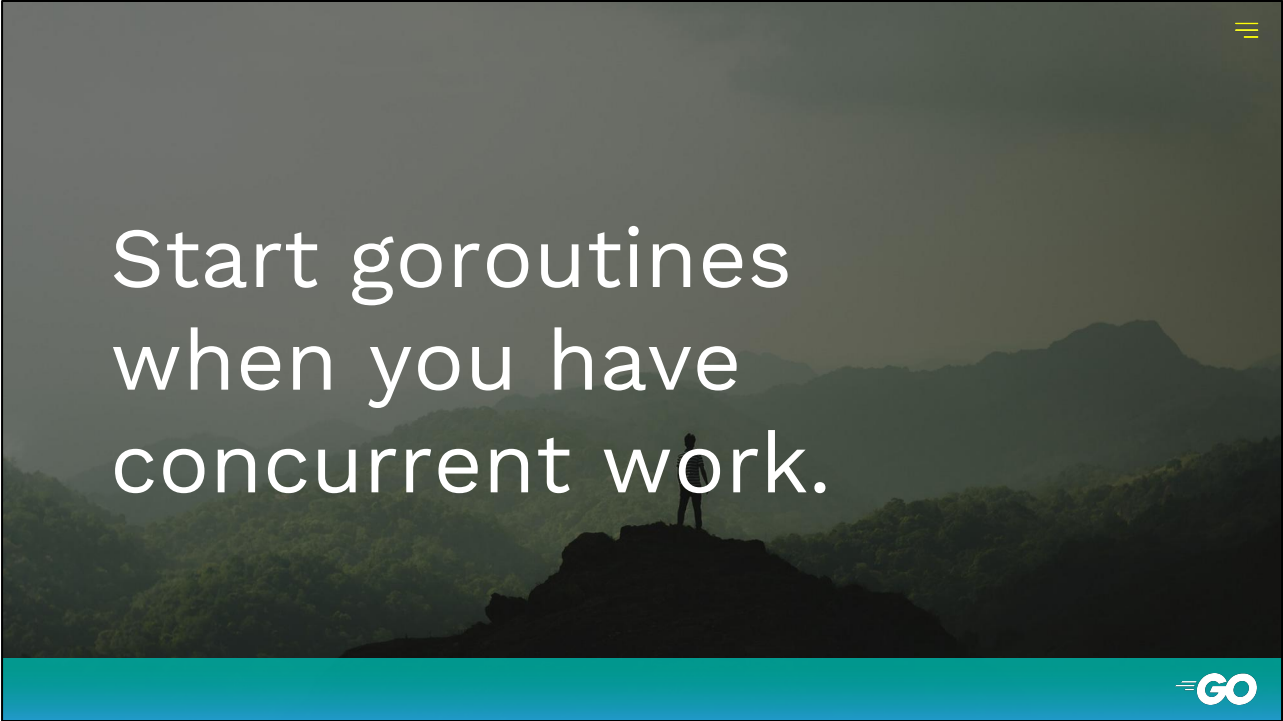
So let's recap.



But before that, I have one last note to add: in this talk I've focused on making the code *clear* and *robust*.

The patterns I'm recommending here *should* all be *reasonably* efficient — generally the right asymptotic complexity and reasonable constant factors — but I *don't* promise that they provide optimal performance. I haven't *benchmarked* them.

If *you* have, you may find that performance is better with one of the patterns I've cautioned against. You may take the *downsides* of those patterns into account and decide to use the patterns anyway. If you *do*, please remember to *document* your reasoning, and *check in* the benchmarks that support it. The Go language itself doesn't change much at the moment, but the implementation *certainly* does.



# Start goroutines when you have concurrent work.



So here's what we've learned.

Start goroutines when *you* have concurrent work to do immediately.

Don't contort the API to avoid blocking the caller, and don't spool up idle workers that will just fill up your goroutine dumps.

It's easy to start goroutines when you *need* them, and easy to *block* when you don't.



# Share by communicating.

Share *things* by communicating *those things* directly.

*Opaque* signals about *shared memory* make it entirely too easy to send the signals to the wrong place, or miss them entirely.

Instead, communicate *where* things need to go, and then communicate to *send* them there.





GOPHERCON 2018

# Rethinking Classical Concurrency Patterns



Bryan C. Mills

[bcmills@google.com](mailto:bcmills@google.com)

GH: [@bcmills](https://github.com/bcmills)

Thank you for your time and attention!

I'd appreciate any feedback you may have, in person here at GopherCon or by email to the [golang-nuts](https://groups.google.com/forum/#!forum/golang-nuts) list or to [bcmills@google.com](mailto:bcmills@google.com).

- Go gopher designed by Renee French.  
Licensed under the Creative Commons 3.0 Attribution license (CC BY 3.0).
- Go presentation theme (with edits)  
Licensed under the Creative Commons 3.0 Attribution license (CC BY 3.0).
- Background photos courtesy unsplash.com,  
under the Unsplash license.



⚠ ENTER AT OWN RISK

---

# Backup Slides

## FUTURE (THE PRECISE WAY)

API:

```
func Fetch(name string) (func() Item) {  
    item := new(Item)  
    ready := make(chan struct{})  
    [...]  
    return func() Item {  
        <-ready  
        return *item  
    }  
}
```

If we want to make the Future idempotent,  
we can close the channel instead of sending on it.

For completeness, here's the alternative. It doesn't fit on a slide very well, but trust me that it has similar problems.

And you can't select on it without breaking the abstraction.

## CONDITION VARIABLE: EVENT FANOUT

```
type Notifier struct {
    mu    sync.Mutex
    changed sync.Cond
    seq int64
}

func (n *Notifier) AwaitChange(
    seq int) (newSeq int) {
    n.mu.Lock()
    defer n.mu.Unlock()
    for n.seq == seq {
        n.changed.Wait()
    }
    return n.seq
}

func (n *Notifier) NotifyChange() {
    n.mu.Lock()
    defer n.mu.Unlock()
    n.seq++
    n.changed.Broadcast()
}

func NewNotifier() *Notifier {
    r := new(Notifier)
    r.changed.L = &r.mu
    return r
}
```

Here's an example. Each broadcast unblocks zero or more waiters.

—

<https://play.golang.org/p/Fw04Ceylfx>

## COMMUNICATION: EVENT FANOUT (CHANNELS)

```
type Notifier struct {  
    st chan state  
}  
  
type state struct {  
    seq int64  
    wait []chan<- int64  
}  
  
func NewNotifier() *Notifier {  
    st := make(chan state, 1)  
    st <- state{seq: 0}  
    return &Notifier{st}  
}
```

CONDITION VARIABLES



We can send a description of the event on multiple channels, one per waiter.

—

[https://play.golang.org/p/uRwV\\_i0v13T](https://play.golang.org/p/uRwV_i0v13T)

## COMMUNICATION: EVENT FANOUT



```
func (n *Notifier) AwaitChange(
    ctx context.Context,
    seq int64) (newSeq int64) {

    c := make(chan int64, 1)
    st := <-n.st
    if st.seq == seq {
        st.wait = append(st.wait, c)
    } else {
        c <- st.seq
    }
    n.st <- st
}
```

```
select {
    case <-ctx.Done():
        return seq
    case newSeq = <-c:
        return newSeq
}

func (n *Notifier) NotifyChange() {
    st := <-n.st
    for _, c := range st.wait {
        c <- st.seq + 1
    }
    n.st <- state{st.seq + 1, nil}
}
```

With multiple channels in place of one condition variable,  
the code is more verbose — but we can support cancellation.

Because we're using multiple channels in place of a single condition variable, the code is a bit more verbose — but now we can support cancellation.

## COMMUNICATION: EVENT FANOUT (CLOSURE)

```
type Notifier struct {
    st chan state
}

type state struct {
    seq      int64
    changed chan struct{} // Closed when the state is replaced.
}

func NewNotifier() *Notifier {
    st := make(chan state, 1)
    st <- state{seq: 0, changed: make(chan struct{})}
    return &Notifier{st}
}
```

CONDITION VARIABLES



We can allocate the channels upfront, or wait for the first waiter. Here, we allocate eagerly.

—

<https://play.golang.org/p/tWVvXOs87HX>



## COMMUNICATION: EVENT FANOUT



```
func (n *Notifier) AwaitChange(  
    ctx context.Context,  
    seq int64) (newSeq int64) {  
  
    st := <-n.st  
    n.st <- st  
    if st.seq != seq {  
        return st.seq  
    }  
    select {  
    case <-ctx.Done():  
        return seq  
    case <-st.changed:  
        return seq + 1  
    }  
}
```

```
func (n *Notifier) NotifyChange() {  
    st := <-n.st  
    close(st.changed)  
    n.st <- state{  
        st.seq + 1,  
        make(chan struct{}),  
    }  
}
```

If we close a channel to broadcast, we can select on cancellation but must convey any additional data out-of-band.

CONDITION VARIABLES



This approach doesn't need as many channel allocations if we have multiple waiters, but requires a fresh channel for each event — and prevents us from sending any additional data along with the signal.

# CHANNEL SEMAPHORE: CACHING RESOURCES

Set up the  
semaphores:

```
sem := make(chan token, limit)
idle := make(chan Conn, limit)
```

Each resource has a corresponding token in the semaphore.

CONDITION VARIABLES



—  
For examples of this pattern, see:

- <http://www.ryanday.net/2012/09/12/golang-using-channels-for-a-connection-pool/>
- <https://github.com/fatih/pool> (circa 2013)
- <http://dustin.sallings.org/2014/04/25/chan-pool.html>

## CHANNEL SEMAPHORE: CACHING RESOURCES

Acquire a resource:

```
select {  
  case conn = <-idle:  
default:  
  select {  
    case conn = <-idle:  
    case sem <- token{}:  
      conn = newConn()  
  }  
}  
  
defer func() { idle <- conn }()
```

If we can add a token, we can add a resource.

## CHANNEL SEMAPHORE: CACHING RESOURCES

Drain the pool:

```
for n := limit; n > 0; n-- {  
    select {  
        case conn := <-idle:  
            conn.Close()  
        case sem <- token{}:  
    }  
}
```

To acquire all of the tokens, acquire all of the resources.

## CHANNEL SEMAPHORE: CACHING RESOURCES

Prune idle resources:

```
empty := false
for !empty {
    select {
        case conn := <-idle:
            conn.Close()
            <-sem // Remove conn's token.
        default:
            empty = true
    }
}
```

We can safely prune resources at any time.

## CHANNEL SEMAPHORE: LIMITING RESOURCES

Use the pool:

```
var wg sync.WaitGroup
for _, task := range hugeSlice {
    wg.Add(1)
    go func(task Task) {
        prepare(task)
        conn := <-idle
        perform(task, conn)
        idle <- conn
        wg.Done()
    }(task Task)
}
wg.Wait()
```

If only part of the work requires the resource,  
we can limit only that part (and use a WaitGroup for the rest).

## CHANNEL SEMAPHORE: LIMITING RESOURCES

Create the tokens:

```
idle := make(chan Conn, limit)
for len(idle) < cap(idle) {
    idle <- newConn()
}
```

Since we're swapping the direction of the channel,  
we need to fill it before use.

CONDITION VARIABLES



Since we're swapping the direction of the semaphore channel, we need to pre-populate it with resource-tokens.

If our “tokens” also need to be closed when no longer in use, we may need to close and drain the channel when we're done too.

## WORKER POOL: TASK-BASED SYNCHRONIZATION

Start the workers:

```
work := make(chan Task)
done := make(map[Task]chan struct{})
for n := limit; n > 0; n-- {
    go func() {
        for task := range work {
            perform(task)
            close(done[task])
        }
    }()
}
```

A fixed pool of goroutines read and perform tasks from a channel.

Synchronizing on the tasks instead of using a WaitGroup doesn't help much. It's still about the same amount of boilerplate, and more allocations to boot.



## WORKER POOL: TASK-BASED SYNCHRONIZATION

Send the work:

```
for _, task := range hugeSlice {  
    done[task] = make(chan struct{})  
    work <- task  
}
```

Clean up:

```
close(work)  
for _, task := range hugeSlice {  
    <-done[task]  
}
```

Another goroutine feeds work to the channel.

# SEMAPHORE CHANNEL: DYNAMIC TASK SPLITTING

```
sem := make(chan token, limit)
var children sync.WaitGroup
walk = func(n *Node) {
    [...]
    for _, child := range n.Children {
        select {
            case sem <- token{}:
                children.Add(1)
                go func() {
                    walk(child)
                    <-sem
                    children.Done()
                } ()
            default:
                walk(child)
        }
    }
}
```

```
sem <- token{}
walk(root)
<-sem
children.Wait()
```

Since we don't need to close a channel to signal the end of work, we can add new tasks on the fly.

WORKER POOLS



When we separate resources from concurrency, we can make decisions about concurrency based on resource availability.

For example, we can try to acquire another token without blocking, and add concurrency as we go:

This pattern would be [much more complicated](#) with a conventional worker pool: because we can't do any of the work before we send the task, we need to add the tasks to the WaitGroup *before* we try to send them on the channel.

Even with semaphores we have to be careful, though: we can't acquire the semaphore tokens to wait for completion, because the remaining workers might want them again. We'll need a separate WaitGroup for that.

# WORKER POOL: DYNAMIC TASK SPLITTING

```
work := make(chan *Node)
var workers sync.WaitGroup
var nodes sync.WaitGroup
walk = func(n *Node) {
    [...]
    for _, child := range n.Children {
        nodes.Add(1)
        select {
        case work <- child:
        default:
            walk(child)
            nodes.Done()
        }
    }
}
```

```
for n := limit; n > 0; n-- {
    go func() {
        for n := range work {
            walk(n)
            nodes.Done()
        }
        workers.Done()
    }()
}

nodes.Add(1)
work <- root
nodes.Wait()
close(work)
workers.Wait()
```

Since we don't need to close a channel to signal the end of work, we can add new tasks on the fly.

Because we can't check whether we have an available worker without sending the work to it, we have to add every node to the WaitGroup.

## A NEW API: WORKER.POOL?



```
package worker

// NewPool starts a pool of n workers.
func NewPool(n int) *Pool

// Add sends f to a worker goroutine, which then executes it.
// If Finish has been called, Add returns ErrFinished.
// If ctx is done and no worker is available, Add returns ctx.Err().
// f must not call Add.
func (p *Pool) Add(ctx context.Context, f func()) error

// Finish causes pending and future calls to Add to return ErrFinished,
// then blocks until all workers have returned.
func (p *Pool) Finish()

// ErrFinished indicates that Finish has been called on a Pool.
var ErrFinished error = [...]
```

Yet another worker pool package?

WORKER POOLS



By now we've got a fair amount of boilerplate. It's mostly straightforward, but at least a little bit subtle.

So maybe we decide to add a library instead.

There are tons of these in godoc.org. Here's my version: it seems to cover the basics, it supports Context cancellation, and the implementation — including TryAdd and MustAdd variants — fits in a hundred lines of code.

Unfortunately, this API isn't very much simpler than the code it's replacing: the caller still has to remember to call Finish, even if they wait for each individual task to complete (for example, by reading from a channel of errors). Otherwise, they'll leak worker goroutines.

But if they call Finish too soon, Add will fail.

—

<https://play.golang.org/p/i81YDGrhK0C>

## A NEW API: WORKER.POOL?

Start the workers: **work := worker.NewPool(limit)**

Send the work: 

```
for _, task := range hugeSlice {  
    task := task  
    work.Add(ctx, func() {  
        perform(task)  
    })  
}
```

Signal end of work & wait for completion: **work.Finish()**

It does clean up the code a bit, though. It's an improvement, but is it enough of one?

## A “NEW” API: SEMAPHORE?



```
package semaphore

// New returns a new semaphore with n tokens.
func New(n int) *Semaphore

// Add calls f in a goroutine while holding a semaphore token.
// If ctx is done and no token is available, Add returns ctx.Err().
// f must not call Add.
func (s *Semaphore) Add(ctx context.Context, f func()) error

// Wait blocks until all tokens are released.
func (s *Semaphore) Wait()
```

Remember the worker-pool package we considered?

We can write nearly the same API for semaphores, but without that messy “finished” error condition.

—

<https://play.golang.org/p/OJZsRoaFLtN>

## A “NEW” API: SEMAPHORE?

Start the work:

```
work := semaphore.New(limit)  
for _, task := range hugeSlice {  
    task := task  
    work.Add(ctx, func() {  
        perform(task)  
    })  
}
```

Wait for completion:

```
work.Wait()
```

## A NEW API: ERRGROUP ADDITIONS?

---



```
package errgroup

func (*Group) SetLimit(n int)
func (*Group) TryGo(f func() error) bool
```

---



## A NEW API: ERRGROUP ADDITIONS?



```
work, ctx := errgroup.WithContext(ctx)
work.SetLimit(limit)
for _, task := range hugeSlice {
    if ctx.Err() != nil {
        work.Go(ctx.Err)
        break
    }
    task := task
    work.Go(func() error {
        defer func() {
            return perform(ctx, task)
        }()
    })
}
err := work.Wait()
```