Patterns and Hints for Concurrency in Go

Russ Cox

MIT 6.5840 / “Spring” 2026

Concurrency is not Parallelism

Concurrency: composition of independently executing processes.
Parallelism: simultaneous execution of (possibly related) computations.

Concurrency is about ***dealing with*** lots of things at once.
Parallelism is about ***doing*** lots of things at once.

Prologue:
Goroutines for State

```
/"([^"\\]|\\.)*"/
```

```
state := 0
for {
    c := read()
    switch state {
    case 0:
        if c != '"' {
            return false
        }
        state = 1
    case 1:
        if c == '"' {
            return true
        }
        if c == '\\' {
            state = 2
        } else {
            state = 1
        }
    case 2:
        state = 1
    }
}
```

```
state := 0
for {
    c := read()
    switch state {
    case 0:

        if c != '"' {
            return false
        }
        state = 1
    case 1:

        if c == '"' {
            return true
        }
        if c == '\\' {
            state = 2
        } else {
            state = 1
        }
    case 2:

        state = 1
    }
}
```

```
state := 0
for {

    switch state {
    case 0:
        c := read()
        if c != '"' {
            return false
        }
        state = 1
    case 1:
        c := read()
        if c == '"' {
            return true
        }
        if c == '\\' {
            state = 2
        } else {
            state = 1
        }
    case 2:
        read()
        state = 1
    }
}
```

```
state := 0
for {

    switch state {
    case 0:
        c := read()
        if c != '"' {
            return false
        }
        state = 1
    case 1:
        c := read()
        if c == '"' {
            return true
        }
        if c == '\\' {
            state = 2
        } else {
            state = 1
        }
    case 2:
        read()
        state = 1
    }
}
```

```
state0:
    c := read()
    if c != '"' {
        return false
    }
    goto state1
state1:
    c := read()
    if c == '"' {
        return true
    }
    if c == '\\' {
        goto state2
    } else {
        goto state1
    }
state2:
    read()
    goto state1
```

```
state0:
    c := read()
    if c != '"' {
        return false
    }
    goto state1
state1:
    c := read()
    if c == '"' {
        return true
    }
    if c == '\\' {
        goto state2
    } else {
        goto state1
    }
state2:
    read()
    goto state1
```

⇨

```
state0:
    c := read()
    if c != '"' {
        return false
    }

state1:
    c := read()
    if c == '"' {
        return true
    }
    if c == '\\' {
        goto state2
    } else {
        goto state1
    }
state2:
    read()
    goto state1
```

```
state0:
    c := read()
    if c != '"' {
        return false
    }

state1:
    c := read()
    if c == '"' {
        return true
    }
    if c == '\\' {
        goto state2
    } else {
        goto state1
    }
state2:
    read()
    goto state1
```

```
state0:
    c := read()
    if c != '"' {
        return false
    }

state1:
    c := read()
    if c == '"' {
        return true
    }
    if c == '\\' {
        read()
        goto state1
    } else {
        goto state1
    }
```

```
state0:
    c := read()
    if c != '"' {
        return false
    }

state1:
    c := read()
    if c == '"' {
        return true
    }
    if c == '\\' {
        read()
        goto state1
    } else {
        goto state1
    }
```

```
state0:
    c := read()
    if c != '"' {
        return false
    }

state1:
    c := read()
    if c == '"' {
        return true
    }
    if c == '\\' {
        read()
    }
    goto state1
```

```
state0:
    c := read()
    if c != '"' {
        return false
    }

state1:
    c := read()
    if c == '"' {
        return true
    }
    if c == '\\' {
        read()
    }
    goto state1
```

```
c := read()
if c != '"' {
    return false
}

for {
    c := read()
    if c == '"' {
        return true
    }
    if c == '\\' {
        read()
    }
}
```

```go
c := read()
if c != '"' {
	return false
}

for {
	c := read()
	if c == '"' {
		return true
	}
	if c == '\\' {
		read()
	}
}
```

⇨

```go
if read() != '"' {
	return false
}

var c rune
for c != '"' {
	c = read()
	if c == '\\' {
		read()
	}
}
return true
```

```
if read() != '"' {
    return false
}
inEscape := false
for {
    c := read()
    if inEscape {
        inEscape = false
        continue
    }
    if c == '"' {
        return true
    }
    if c == '\\' {
        inEscape = true
    }
}
```

⇨

```
if read() != '"' {
    return false
}

var c rune
for c != '"' {
    c = read()
    if c == '\\' {
        read()
    }
}
return true
```

```
           func parse(read func() rune) bool {
state==0 →     if read() != '"' {
                   return false
               }

               var c rune
               for c != '"' {
state==1 →         c = read()             ← inEscape==false
                   if c == '\\' {
state==2 →             read()             ← inEscape==true
                   }
               }
               return true
           }
```

Hint: Convert data state into code state when it makes programs clearer.

```go
type quoter struct {
    state int
}

func (q *quoter) Init() {
    r.state = 0
}

func (q *quoter) Write(c rune) Status {
    switch q.state {
    case 0:
        if c != '"' {
            return BadInput
        }
        q.state = 1
    case 1:
        if c == '"' {
            return Success
        }
        if c == '\\' {
            q.state = 2
        } else {
            q.state = 1
        }
    case 2:
        q.state = 1
    }
    return NeedMoreInput
}
```

```
type quoter struct {
    char   chan rune
    status chan Status
}

func (q *quoter) Init() {
    q.char = make(chan rune)
    q.status = make(chan Status)
    go q.parse()
    <-q.status // always NeedMoreInput
}

func (q *quoter) Write(c rune) Status {
    q.char <- c
    return <-q.status
}
```

```
func (q *quoteReader) parse() {
    if q.read() != '"' {
        q.status <- SyntaxError
        return
    }

    var c rune
    for c != '"' {
        c = q.read()
        if c == '\\' {
            q.read()
        }
    }
    q.status <- Done
}

func (q *quoter) read() int {
    q.status <- NeedMoreInput
    return <-q.char
}
```

Hint: Use additional goroutines to hold additional code state.

```go
package main

import (
    "net/http"
    _ "net/http/pprof"
)

var c = make(chan int)

func main() {
    for i := range 100 {
        go f(0x10*i)
    }
    http.ListenAndServe("localhost:8080", nil)
}

func f(x int) {
    g(x+1)
}

func g(x int) {
    h(x+1)
}

func h(x int) {
    c <- 1
    f(x+1)
}
```

Hint: Know why and when each goroutine will exit.

```
$ go run x.go
^\

SIGQUIT: quit
PC=0x105a17b m=0 sigcode=0

...

goroutine 18 [chan send]:
main.h(0x12)
    /tmp/x.go:26 +0x45
main.g(0x11)
    /tmp/x.go:22 +0x20
main.f(0x10)
    /tmp/x.go:18 +0x20
created by main.main
    /tmp/x.go:12 +0x42

goroutine 19 [chan send]:
main.h(0x22)
    /tmp/x.go:26 +0x45
main.g(0x21)
    /tmp/x.go:22 +0x20
main.f(0x20)
    /tmp/x.go:18 +0x20
created by main.main
    /tmp/x.go:12 +0x42

...
```

Hint: Type Ctrl-\ to kill a program and dump all its goroutine stacks.

```
goroutine profile: total 106
100 @ 0x12d8715 0x12d86c0 0x12d8690 0x1058d61
#   0x12d8714   main.h+0x44 /tmp/x.go:26
#   0x12d86bf   main.g+0x1f /tmp/x.go:22
#   0x12d868f   main.f+0x1f /tmp/x.go:18

2 @ 0x11ddfcf 0x11dddcf 0x1248265 0x124f513 0x1253636 0x1058d61
#   0x11ddfce   net/textproto.(*Reader).readLineSlice+0x5e  go/src/net/textproto/reader.go:55
#   0x11dddce   net/textproto.(*Reader).ReadLine+0x2e       go/src/net/textproto/reader.go:36
#   0x1248264   net/http.readRequest+0xa4                   go/src/net/http/request.go:926
#   0x124f512   net/http.(*conn).readRequest+0x1b2          go/src/net/http/server.go:934
#   0x1253635   net/http.(*conn).serve+0x495                go/src/net/http/server.go:1763

1 @ 0x115a102 0x116b1cd 0x124dc92 0x1058d61
#   0x115a101   net.(*netFD).Read+0x51                      go/src/net/fd_unix.go:207
#   0x116b1cc   net.(*conn).Read+0x6c                       go/src/net/net.go:182
#   0x124dc91   net/http.(*connReader).backgroundRead+0x61  go/src/net/http/server.go:656

1 @ 0x12cfe22 0x12cfc20 0x12cc6e5 0x12d8051 0x12d8365 0x1254b84 0x1255fa0 0x1257312 0x1253845
0x1058d61
#   0x12cfe21   runtime/pprof.writeRuntimeProfile+0xa1      go/src/runtime/pprof/pprof.go:634
#   0x12cfc1f   runtime/pprof.writeGoroutine+0x9f           go/src/runtime/pprof/pprof.go:596
#   0x12cc6e4   runtime/pprof.(*Profile).WriteTo+0x3b4      go/src/runtime/pprof/pprof.go:310
#   0x12d8050   net/http/pprof.handler.ServeHTTP+0x1d0      go/src/net/http/pprof/pprof.go:232
#   0x12d8364   net/http/pprof.Index+0x1e4                  go/src/net/http/pprof/pprof.go:244
#   0x1254b83   net/http.HandlerFunc.ServeHTTP+0x43         go/src/net/http/server.go:1942
#   0x1255f9f   net/http.(*ServeMux).Serv
#   0x1257311   net/http.serverHandler.Se
#   0x1253844   net/http.(*conn).serve+0x
```

Hint: Use the HTTP server's /debug/pprof/goroutine to inspect live goroutine stacks.

Pattern #1

Publish/subscribe server

```go
type PubSub interface {
    // Publish publishes the event e to
    // all current subscriptions.
    Publish(e Event)

    // Subscribe registers c to receive future events.
    // All subscribers receive events in the same order,
    // and that order respects program order:
    // if Publish(e1) happens before Publish(e2),
    // subscribers receive e1 before e2.
    Subscribe(c chan<- Event)

    // Cancel cancels the prior subscription of channel c.
    // After any pending already-published events
    // have been sent on c, the server will signal that the
    // subscription is cancelled by closing c.
    Cancel(c chan<- Event)
}
```

```go
type PubSub interface {
    // Publish publishes the event e to
    // all current subscriptions.
    Publish(e Event)

    // Subscribe registers c to receive future events.
    // All subscribers receive events in the same order,
    // and that order respects program order:
    // if Publish(e1) happens before Publish(e2),
    // subscribers receive e1 before e2.
    Subscribe(c chan<- Event)

    // Cancel cancels the prior subscription of channel c.
    // After any pending already-published events
    // have been sent on c, the server will signal that the
    // subscription is cancelled by closing c.
    Cancel(c chan<- Event)
}
```

Hint: Close a channel to signal that no more values will be sent.

```
type Server struct {
    mu sync.Mutex
    sub map[chan<- Event]bool
}

func (s *Server) Init() {
    s.sub = make(map[chan<- Event]bool)
}

func (s *Server) Publish(e Event) {
    s.mu.Lock()
    defer s.mu.Unlock()

    for c := range s.sub {
        c <- e
    }
}

func (s *Server) Subscribe(c chan<- Event) {
    s.mu.Lock()
    defer s.mu.Unlock()

    if s.sub[c] {
        panic("pubsub: already subscribed")
    }
    s.sub[c] = true
}

func (s *Server) Cancel(c chan<- Event) {
    s.mu.Lock()
    defer s.mu.Unlock()

    if !s.sub[c] {
        panic("pubsub: not subscribed")
    }
    close(c)
    delete(s.sub, c)
}
```

```go
type Server struct {
    mu sync.Mutex
    sub map[chan<- Event]bool
}

func (s *Server) Init() {
    s.sub = make(map[chan<- Event]bool)
}

func (s *Server) Publish(e Event) {
    s.mu.Lock()
    defer s.mu.Unlock()

    for c := range s.sub {
        c <- e
    }
}

func (s *Server) Subscribe(c chan<- Event) {
    s.mu.Lock()
    defer s.mu.Unlock()

    if s.sub[c] {
        panic("pubsub: already subscribed")
    }
    s.sub[c] = true
}

func (s *Server) Cancel(c chan<- Event) {
    s.mu.Lock()
    defer s.mu.Unlock()

    if !s.sub[c] {
        panic("pubsub: not subscribed")
    }
    close(c)
    delete(s.sub, c)
}
```

Hint: Prefer defer for unlocking mutexes.

```go
type Server struct {
    mu sync.Mutex
    sub map[chan<- Event]bool
}

func (s *Server) Init() {
    s.sub = make(map[chan<- Event]bool)
}

func (s *Server) Publish(e Event) {
    s.mu.Lock()
    defer s.mu.Unlock()

    for c := range s.sub {
        c <- e
    }
}

func (s *Server) Subscribe(c chan<- Event) {
    s.mu.Lock()
    defer s.mu.Unlock()

    if s.sub[c] {
        panic("pubsub: already subscribed")
    }
    s.sub[c] = true
}

func (s *Server) Cancel(c chan<- Event) {
    s.mu.Lock()
    defer s.mu.Unlock()

    if !s.sub[c] {
        panic("pubsub: not subscribed")
    }
    close(c)
    delete(s.sub, c)
}
```

Hint: Consider the effect of slow goroutines.

Options for slow goroutines

- Slow down event generation.
- Drop events.
  Examples: os/signal, runtime/pprof
- Queue an arbitrary number of events.

Hint: Think carefully before introducing unbounded queuing.

```go
type Server struct {
    mu sync.Mutex
    sub map[chan<- Event]bool
}

func (s *Server) Init() {
    s.sub = make(map[chan<- Event]bool)
}
```

```go
type Server struct {
    publish   chan Event
    subscribe chan subReq
    cancel    chan subReq
}

type subReq struct {
    c  chan<- Event
    ok chan bool
}

func (s *Server) Init() {
    s.publish = make(chan Event)
    s.subscribe = make(chan subReq)
    s.cancel = make(chan subReq)
    go s.loop()
}
```

```
func (s *Server) Publish(e Event) {
    s.mu.Lock()
    defer s.mu.Unlock()

    for c := range s.sub {
        c <- e
    }
}

func (s *Server) Subscribe(c chan<- Event) {
    s.mu.Lock()
    defer s.mu.Unlock()

    if s.sub[c] {
        panic("pubsub: already subscribed")
    }
    s.sub[c] = true
}

func (s *Server) Cancel(c chan<- Event) {
    s.mu.Lock()
    defer s.mu.Unlock()

    if !s.sub[c] {
        panic("pubsub: not subscribed")
    }
    close(c)
    delete(s.sub, c)
}
```

```
func (s *Server) loop() {
    sub := make(map[chan<- Event]bool)
    for {
        select {
        case e := <-s.publish:
            for c := range sub {
                c <- e
            }

        case r := <-s.subscribe:
            if sub[r.c] {
                r.ok <- false
                break
            }
            sub[r.c] = true
            r.ok <- true

        case c := <-s.cancel:
            if !sub[r.c] {
                r.ok <- false
                break
            }
            close(r.c)
            delete(sub, r.c)
            r.ok <- true
        }
    }
}
```

```go
func (s *Server) Publish(e Event) {
	s.mu.Lock()
	defer s.mu.Unlock()

	for c := range s.sub {
		c <- e
	}
}

func (s *Server) Subscribe(c chan<- Event) {
	s.mu.Lock()
	defer s.mu.Unlock()

	if s.sub[c] {
		panic("pubsub: already subscribed")
	}
	s.sub[c] = true
}

func (s *Server) Cancel(c chan<- Event) {
	s.mu.Lock()
	defer s.mu.Unlock()

	if !s.sub[c] {
		panic("pubsub: not subscribed")
	}
	close(c)
	delete(s.sub, c)
}
```

```go
func (s *Server) Publish(e Event) {
	s.publish <- e
}

func (s *Server) Subscribe(c chan<- Event) {
	r := subReq{c: c, ok: make(chan bool)}
	s.subscribe <- r
	if !<-r.ok {
		panic("pubsub: already subscribed")
	}
}

func (s *Server) Cancel(c chan<- Event) {
	r := subReq{c: c, ok: make(chan bool)}
	s.cancel <- r
	if !<-r.ok {
		panic("pubsub: not subscribed")
	}
}
```

```
type Server struct {
    publish chan Event
    subscribe chan subReq
    cancel chan subReq
}

type subReq struct {
    c chan<- Event
    ok chan bool
}

func (s *Server) Init() {
    s.publish = make(chan Event)
    s.subscribe = make(chan subReq)
    s.cancel = make(chan subReq)
    go s.loop()
}

func (s *Server) Publish(e Event) {
    s.publish <- e
}

func (s *Server) Subscribe(c chan<- Event) {
    r := subReq{c: c, ok: make(chan bool)}
    s.subscribe <- r
    if !<-r.ok {
        panic("pubsub: already subscribed")
    }
}

func (s *Server) Cancel(c chan<- Event) {
    r := subReq{c: c, ok: make(chan bool)}
    s.cancel <- r
    if !<-r.ok {
        panic("pubsub: not subscribed")
    }
}
```

```
func (s *Server) loop() {
    sub := make(map[chan<- Event]bool)
    for {
        select {
        case e := <-s.publish:
            for c := range sub {
                c <- e
            }

        case r := <-s.subscribe:
            if sub[r.c] {
                r.ok <- false
                break
            }
            sub[r.c] = true
            r.ok <- true

        case c := <-s.cancel:
            if !sub[r.c] {
                r.ok <- false
                break
            }
            close(r.c)
            delete(sub, r.c)
            r.ok <- true
        }
    }
}
```

Hint: Convert mutexes into goroutines when it makes programs clearer

```go
func helper(in <-chan Event,
            out chan<- Event) {
    var q []Event
    for {
        select {
        case e := <-in:
            q = append(q, e)
        case out <- q[0]:
            q = q[1:]
        }
    }
}
```

```go
func helper(in <-chan Event,
            out chan<- Event) {
    var q []Event
    for {
        select {
        case e := <-in:
            q = append(q, e)
        case out <- q[0]:
            q = q[1:]
        }
    }
}
```

```go
func helper(in <-chan Event,
            out chan<- Event) {
    var q []Event
    for {
        // Decide whether and what to send.
        var sendOut chan<- Event
        var next Event
        if len(q) > 0 {
            sendOut = out
            next = q[0]
        }

        select {
        case e := <-in:
            q = append(q, e)
        case sendOut <- next:
            q = q[1:]
        }
    }
}
```

```
func helper(in <-chan Event,
            out chan<- Event) {
    var q []Event
    for {
        // Decide whether and what to send.
        var sendOut chan<- Event
        var next Event
        if len(q) > 0 {
            sendOut = out
            next = q[0]
        }

        select {
        case e := <-in:
            q = append(q, e)
        case sendOut <- next:
            q = q[1:]
        }
    }
}
```

⇒

```
func helper(in <-chan Event,
            out chan<- Event) {
    var q []Event
    for in != nil || len(q) > 0 {
        // Decide whether and what to send.
        var sendOut chan<- Event
        var next Event
        if len(q) > 0 {
            sendOut = out
            next = q[0]
        }

        select {
        case e, ok := <-in:
            if !ok {
                in = nil // stop receiving from in
                break
            }
            q = append(q, e)
        case sendOut <- next:
            q = q[1:]
        }
    }
    close(out)
}
```

```go
func (s *Server) loop() {
    sub := make(map[chan<- Event]bool)
    for {
        select {
        case e := <-s.publish:
            for c := range sub {
                c <- e
            }

        case r := <-s.subscribe:
            if sub[r.c] {
                r.ok <- false
                break
            }
            sub[r.c] = true
            r.ok <- true

        case c := <-s.cancel:
            if !sub[r.c] {
                r.ok <- false
                break
            }
            close(r.c)
            delete(sub, r.c)
            r.ok <- true
        }
    }
}
```

```
func (s *Server) loop() {
    sub := make(map[chan<- Event]bool)
    for {
        select {
        case e := <-s.publish:
            for c := range sub {
                c <- e
            }

        case r := <-s.subscribe:
            if sub[r.c] {
                r.ok <- false
                break
            }
            sub[r.c] = true
            r.ok <- true

        case c := <-s.cancel:
            if !sub[r.c] {
                r.ok <- false
                break
            }
            close(r.c)
            delete(sub, r.c)
            r.ok <- true
        }
    }
}
```

```
func (s *Server) loop() {
    sub := make(map[chan<- Event]chan<- Event)
    for {
        select {
        case e := <-s.publish:
            for _, h := range sub {
                h <- e
            }

        case r := <-s.subscribe:
            if sub[r.c] != nil {
                r.ok <- false
                break
            }
            h = make(chan Event)
            go helper(h, r.c)
            sub[r.c] = h
            r.ok <- true

        case c := <-s.cancel:
            if sub[r.c] == nil {
                r.ok <- false
                break
            }
            close(sub[r.c])
            delete(sub, r.c)
            r.ok <- true
        }
    }
}
```

Hint: Use goroutines to let independent concerns run independently.

Pattern #2

Work scheduler

```
func Schedule(servers []string, numTask int,
              call func(srv string, task int))
```

```
func Schedule(servers []string, numTask int,
              call func(srv string, task int)) {

    idle := make(chan string, len(servers))
    for _, srv := range servers {
        idle <- srv
    }
}
```

Hint: Use a buffered channel as a concurrent blocking queue.

```
func Schedule(servers []string, numTask int,
              call func(srv string, task int)) {

    idle := make(chan string, len(servers))
    for _, srv := range servers {
        idle <- srv
    }

    for task := range numTask {
        go func() {
            srv := <-idle
            call(srv, task)
            idle <- srv
        }()
    }
}
```

Hint: Use goroutines to let independent concerns run independently.

```
func Schedule(servers []string, numTask int,
              call func(srv string, task int)) {

    idle := make(chan string, len(servers))
    for _, srv := range servers {
        idle <- srv
    }

    for task := range numTask {
        go func() {
            srv := <-idle
            call(srv, task)
            idle <- srv
        }()
    }
}
```

Hint: Think carefully before introducing unbounded queuing.

```
func Schedule(servers []string, numTask int,
              call func(srv string, task int)) {

    idle := make(chan string, len(servers))
    for _, srv := range servers {
        idle <- srv
    }

    for task := range numTask {
       go func() {
            srv := <-idle
            call(srv, task)
            idle <- srv
        }()
    }
}
```

```
for task := range numTask {
    srv := <-idle
    go func() {
        call(srv, task)
        idle <- srv
    }()
}
```

```
func Schedule(servers []string, numTask int,
              call func(srv string, task int)) {

    idle := make(chan string, len(servers))
    for _, srv := range servers {
        idle <- srv
    }

    for task := range numTask {
        srv := <-idle
        go func() {
            call(srv, task)
            idle <- srv
        }()
    }

    for range servers {
        <-idle
    }
}
```

```go
func Schedule(servers []string, numTask int,
              call func(srv string, task int)) {

    idle := make(chan string, len(servers))
    for _, srv := range servers {
        idle <- srv
    }

    for task := range numTask {
        srv := <-idle
        go func() {
            call(srv, task)
            idle <- srv
        }()
    }

    for range servers {
        <-idle
    }
}
```

```
func Schedule(servers []string, numTask int,
              call func(srv string, task int)) {

    work := make(chan int)
    done := 0

    runTasks := func(srv string) {
        for task := range work {
            call(srv, task)
        }
        done++
    }

    for _, srv := range servers {
        go runTasks(srv)
    }

    for task := range numTask {
        work <- task
    }
    close(work)

    for done < len(servers) {
        runtime.Gosched()
    }
}
```

Hint: Think carefully before introducing unbounded queuing.

Hint: Close a channel to signal that no more values will be sent.

```
func Schedule(servers []string, numTask int,
              call func(srv string, task int)) {

    work := make(chan int)
    done := 0

    runTasks := func(srv string) {
        for task := range work {
            call(srv, task)
        }
        done++
    }

    for _, srv := range servers {
        go runTasks(srv)
    }

    for task := range numTask {
        work <- task
    }
    close(work)

    for done < len(servers) {
        runtime.Gosched()
    }
}
```

```
$ go run -race /tmp/x.go
==================
WARNING: DATA RACE
Write at 0x00c0000121d8 by goroutine 6:
  main.Schedule.func1()
      /tmp/x.go:19 +0x80
  main.Schedule.gowrap1()
      /tmp/x.go:23 +0x48

Previous read at 0x00c0000121d8 by main goro
  main.Schedule()
      /tmp/x.go:31 +0x27c
  main.main()
      /tmp/x.go:6 +0x90

Goroutine 6 (running) created at:
  main.Schedule()
      /tmp/x.go:23 +0x13c
  main.main()
      /tmp/x.go:6 +0x90
==================
```

Hint: Use the race detector,
for development and even production.

```
func Schedule(servers []string, numTask int,
              call func(srv string, task int)) {

    work := make(chan int)
    done := 0

    runTasks := func(srv string) {
        for task := range work {
            call(srv, task)
        }
        done++
    }

    for _, srv := range servers {
        go runTasks(srv)
    }

    for task := range numTask {
        work <- task
    }
    close(work)

    for done < len(servers) {
        runtime.Gosched()
    }
}
```

```
$ go run -race /tmp/x.go
==================
WARNING: DATA RACE
Write at 0x00c00019c008 by goroutine 7:
  main.Schedule.func1()
      /tmp/x.go:19 +0x80
  main.Schedule.gowrap1()
      /tmp/x.go:23 +0x48

Previous write at 0x00c00019c008 by goroutine
  main.Schedule.func1()
      /tmp/x.go:19 +0x80
  main.Schedule.gowrap1()
      /tmp/x.go:23 +0x48|

Goroutine 7 (running) created at:
  main.Schedule()
      /tmp/x.go:23 +0x13c
  main.main()
      /tmp/x.go:6 +0x90

Goroutine 6 (finished) created at:
  main.Schedule()
      /tmp/x.go:23 +0x13c
  main.main()
      /tmp/x.go:6 +0x90
==================
```

```go
func Schedule(servers []string, numTask int,
              call func(srv string, task int)) {

    work := make(chan int)
    done := 0

    runTasks := func(srv string) {
        for task := range work {
            call(srv, task)
        }
        done++
    }

    for _, srv := range servers {
        go runTasks(srv)
    }

    for task := range numTask {
        work <- task
    }
    close(work)

    for done < len(servers) {
        runtime.Gosched()
    }
}
```

```go
work := make(chan int)
done := make(chan bool)

runTasks := func(srv string) {
    for task := range work {
        call(srv, task)
    }
    done <- true
}

...

for range servers {
    <-done
}
```

Hint:

Don't communicate by sharing memory.

Share memory by communicating.

```go
func Schedule(servers []string, numTask int,
              call func(srv string, task int)) {

    work := make(chan int)
    done := make(chan bool)

    runTasks := func(srv string) {
        for task := range work {
            call(srv, task)
        }
        done <- true
    }

    for _, srv := range servers {
        go runTasks(srv)
    }

    for task := range numTask {
        work <- task
    }
    close(work)

    for range servers {
        <-done
    }
}
```

```go
func Schedule(servers chan string, numTask int,
              call func(srv string, task int)) {
```

```go
go func() {
    for srv := range servers {
        go runTasks(srv)
    }
}()
```

Hint: Use goroutines to let independent concerns run independently.

```go
func Schedule(servers chan string, numTask int,
              call func(srv string, task int)) {

    work := make(chan int)
    done := make(chan bool)

    runTasks := func(srv string) {
        for task := range work {
            call(srv, task)
        }
        done <- true
    }

    go func() {
        for _, srv := range servers {
            go runTasks(srv)
        }
    }()

    for range numTask {
        work <- task
    }
    close(work)

    for range servers {
        <-done
    }
}
```

```go
runTasks := func(srv string) {
   for task := range work {
        call(srv, task)
        done <- true
    }
}
```

```go
for range numTask {
    <-done
}
```

```
func Schedule(servers chan string, numTask int,
              call func(srv string, task int)) {

    work := make(chan int)
    done := make(chan bool)

    runTasks := func(srv string) {
        for task := range work {
            call(srv, task)
            done <- true
        }
    }

    go func() {
        for _, srv := range servers {
            go runTasks(srv)
        }
    }()

    for range numTask {
        work <- task
    }
    close(work)

    for range numTask {
        <-done
    }
}
```

```
$ go run /tmp/x.go
fatal error: all goroutines are as

goroutine 1 [chan send]:
main.Schedule(0xc4200120c0, 0x3, 0
    /tmp/x.go:26 +0x150
main.main()
    /tmp/x.go:4 +0x96

goroutine 5 [chan send]:
main.Schedule.func1(0x1066bc0, 0x1
    /tmp/x.go:15 +0xba
created by main.Schedule.func2
    /tmp/x.go:21 +0x5f
```

Hint: Know why and when each communication will proceed.

```go
func Schedule(servers chan string, numTask int,
              call func(srv string, task int)) {

    work := make(chan int)
    done := make(chan bool)

    runTasks := func(srv string) {
        for task := range work {
            call(srv, task)
            done <- true
        }
    }

    go func() {
        for _, srv := range servers {
            go runTasks(srv)
        }
    }()

    for task := range numTask {
        work <- task
    }
    close(work)

    for i := range numTask {
        <-done
    }
}
```

```go
    i := 0
WorkLoop:
    for task := range numTask {
        for {
            select {
            case work <- task:
                continue WorkLoop
            case <-done:
                i++
            }
        }
    }
    close(work)

    for ; i < numTask; i++ {
        <-done
    }
```

```
func Schedule(servers chan string, numTask int,
              call func(srv string, task int)) {

    work := make(chan int)
    done := make(chan bool)

    runTasks := func(srv string) {
        for task := range work {
            call(srv, task)
            done <- true
        }
    }

    go func() {
        for _, srv := range servers {
            go runTasks(srv)
        }
    }()

    for task := range numTask {
        work <- task
    }
    close(work)

    for range numTask {
        <-done
    }
}
```

```
go func() {
    for task := range numTask {
        work <- task
    }
    close(work)
}()
```

Hint: Use goroutines to let independent concerns run independently.

```go
func Schedule(servers chan string, numTask int,
              call func(srv string, task int)) {

    work := make(chan int)
    done := make(chan bool)

    runTasks := func(srv string) {
        for task := range work {
            call(srv, task)
            done <- true
        }
    }

    go func() {
        for _, srv := range servers {
            go runTasks(srv)
        }
    }()

    for task := range numTask {
        work <- task
    }
    close(work)

    for range numTask {
        <-done
    }
}
```

```go
work := make(chan int, numTask)
```

Hint: Think carefully before introducing unbounded queuing.

```
func Schedule(servers chan string, numTask int,
              call func(srv string, task int)) {

    work := make(chan int, numTask)
    done := make(chan bool)

    runTasks := func(srv string) {
        for task := range work {
            call(srv, task)
            done <- true
        }
    }

    go func() {
        for _, srv := range servers {
            go runTasks(srv)
        }
    }()

    for task := range numTask {
        work <- task
    }
    close(work)

    for range numTask {
        <-done
    }
}
```

```go
func Schedule(servers chan string, numTask int,
              call func(srv string, task int) bool) {

    work := make(chan int, numTask)
    done := make(chan bool)

    runTasks := func(srv string) {
        for task := range work {
            call(srv, task)
            done <- true
        }
    }

    go func() {
        for _, srv := range servers {
            go runTasks(srv)
        }
    }()

    for task := range numTask {
        work <- task
    }
    close(work)

    for range numTask {
        <-done
    }
}
```

```go
runTasks := func(srv string) {
    for task := range work {
        if call(srv, task) {
           done <- true
        } else {
           work <- task
        }
    }
}

...

for task := range numTask {
    work <- task
}

for range numTask {
    <-done
}
close(work)
```

Hint: Know why and when each communication will proceed.

```go
func Schedule(servers chan string, numTask int,
              call func(srv string, task int) bool) {

    work := make(chan int, numTask)
    done := make(chan bool)

    runTasks := func(srv string) {
        for task := range work {
            call(srv, task)
            done <- true
        }
    }

    go func() {
        for _, srv := range servers {
            go runTasks(srv)
        }
    }()

    for task := range numTask {
        work <- task
    }
    close(work)

    for range numTask {
        <-done
    }
}
```

```go
runTasks := func(srv string) {
    for task := range work {
        if call(srv, task) {
           done <- true
        } else {
           work <- task
        }
    }
}

...

for task := range numTask {
    work <- task
}

for range numTask {
    <-done
}
close(work)
```

Hint: Close a channel to signal that no more values will be sent.

```
func Schedule(servers chan string, numTask int,
              call func(srv string, task int) bool) {

    work := make(chan int, numTask)
    done := make(chan bool)

    runTasks := func(srv string) {
        for task := range work {
            if call(srv, task) {
                done <- true
            } else {
                work <- task
            }
        }
    }

    go func() {
        for _, srv := range servers {
            go runTasks(srv)
        }
    }()

    for task := range numTask {
        work <- task
    }

    for range numTask {
        <-done
    }
    close(work)
}
```

Hint: Make sure you know why and when each goroutine will exit.

```
func Schedule(servers chan string, numTask int,
              call func(srv string, task int) bool) {

    work := make                                                  t, numTask)
    done := make                                                  ol)
                                                                  ol)

    runTasks := func(srv string) {
        for task := range work {
            if call(srv, task) {
                done <- true
            } else {
                work <- task
            }
        }
    }

    go func() {
        for _, srv := range servers {
            go runTasks(srv)
        }
    }()

    for task := range numTask {
        work <- task
    }

    for range numTask {
        <-done
    }
    close(work)
}
```

```
...

go func() {
    for {
        select {
        case srv := <-servers:
            go runTasks(srv)
        case <-exit:
            return
        }
    }
}()

...

for range numTask {
    <-done
}
close(work)
exit <- true
```

Pattern #3

Replicated service client

```
type ReplicatedClient interface {
    // Init initializes the client to use the given servers.
    // To make a particular request later,
    // the client can use callOne(srv, args), where srv
    // is one of the servers from the list.
    Init(servers []string, callOne func(string, Args) Reply)

    // Call makes a request on any available server.
    // Multiple goroutines may call Call concurrently.
    Call(args Args) Reply
}
```

```
type Client struct {
    servers []string
    callOne func(string, Args) Reply

    mu     sync.Mutex
    prefer int
}

func (c *Client) Init(servers []string, callOne func(string, Args) Reply) {
    c.servers = servers
    c.callOne = callOne
}
```

Hint: Use a mutex if that is the clearest way to write the code.

```go
type Client struct {
    servers []string
    callOne func(string, Args) Reply

    mu     sync.Mutex
    prefer int
}

func (c *Client) Init(servers []string, callOne func(string, Args) Reply) {
    c.servers = servers
    c.callOne = callOne
}

func (c *Client) Call(args Args) Reply {
    type result struct {
        serverID int
        reply Reply
    }

    done := make(chan result, 1)

    id := ...

    go func() {
        done <- result{id, c.callOne(c.servers[id], args)}
    }()
}
```

Hint: Use goroutines to let independent concerns run independently.

```
func (c *Client) Call(args Args) Reply {
    type result struct {
        serverID int
        reply Reply
    }

    const timeout = 1 * time.Second
    t := time.NewTimer(timeout)
    defer t.Stop()

    done := make(chan result, 1)

    id := ...

    go func() {
        done <- result{id, c.callOne(c.servers[id], args)}
    }()

    select {
    case r := <-done:
        return r.reply
    case <-t.C:
        // timeout
    }
}
```

~~Hint: Stop timers you don't need.~~
(fixed in Go 1.23)

Hint: Know why and when each goroutine will exit.

Hint: Know why and when each communication will proceed.

```
func (c *Client) Call(args Args) Reply {
    type result struct {
        serverID int
        reply Reply
    }

    const timeout = 1 * time.Second
    t := time.NewTimer(timeout)
    defer t.Stop()

    done := make(chan result, len(c.servers))

    for id := range c.servers {
        go func() {
            done <- result{id, c.callOne(c.servers[id], args)}
        }()

        select {
        case r := <-done:
            return r.reply
        case <-t.C:
            // timeout
            t.Reset(timeout)
        }
    }

    r := <-done
    return r.reply
}
```

```
    c.mu.Lock()
    prefer := c.prefer
    c.mu.Unlock()

    var r result
    for off := range c.servers {
        id := (prefer + off) % len(c.servers)
        go func() {
            done <- result{id, c.callOne(c.servers[id], args)}
        }()

        select {
        case r = <-done:
            goto Done
        case <-t.C:
            // timeout
            t.Reset(timeout)
        }
    }

    r = <-done
Done:
    c.mu.Lock()
    c.prefer = r.serverID
    c.mu.Unlock()
    return r.reply
```

Hint: Use a goto if that is the clearest way to write the code.

Pattern #4

Protocol multiplexer

```go
type ProtocolMux interface {
    // Init initializes the mux to manage messages to the given service.
    Init(Service)

    // Call makes a request with the given message and returns the reply.
    // Multiple goroutines may call Call concurrently.
    Call(Msg) Msg
}

type Service interface {
    // ReadTag returns the muxing identifier in the request or reply message.
    // Multiple goroutines may call ReadTag concurrently.
    ReadTag(Msg) int64

    // Send sends a request message to the remote service.
    // Send must not be called concurrently with itself.
    Send(Msg)

    // Recv waits for and returns a reply message from the remote service.
    // Recv must not be called concurrently with itself.
    Recv() Msg
}
```

```
type Mux struct {
    srv  Service
    send chan Msg

    mu sync.Mutex
    pending map[int64]chan<- Msg
}

func (m *Mux) Init(srv Service) {
    m.srv = srv
    m.pending = make(map[int64]chan Msg)
    go m.sendLoop()
    go m.recvLoop()
}
```

```go
type Mux struct {
    srv  Service
    send chan Msg

    mu sync.Mutex
    pending map[int64]chan<- Msg
}

func (m *Mux) Init(srv Service) {
    m.srv = srv
    m.pending = make(map[int64]chan Msg)
    go m.sendLoop()
    go m.recvLoop()
}

func (m *Mux) sendLoop() {
    for args := range m.send {
        m.srv.Send(args)
    }
}
```

```go
func (m *Mux) sendLoop() {
    for args := range m.send {
        m.srv.Send(args)
    }
}

func (m *Mux) recvLoop() {
    for {
        reply := m.srv.Recv()
        tag := m.srv.ReadTag(reply)

        m.mu.Lock()
        done := m.pending[tag]
        delete(m.pending, tag)
        m.mu.Unlock()

        if done == nil {
            panic("unexpected reply")
        }
        done <- reply
    }
}
```

```go
func (m *Mux) sendLoop() {
    for args := range m.send {
        m.srv.Send(args)
    }
}

func (m *Mux) recvLoop() {
    for {
        reply := m.srv.Recv()
        tag := m.srv.Tag(reply)

        m.mu.Lock()
        done := m.pending[tag]
        delete(m.pending, tag)
        m.mu.Unlock()

        if done == nil {
            panic("unexpected reply")
        }
        done <- reply
    }
}
```

```go
func (m *Mux) Call(args Msg) (reply Msg) {
    tag := m.srv.ReadTag(args)
    done := make(chan Msg, 1)

    m.mu.Lock()
    if m.pending[tag] != nil {
        m.mu.Unlock()
        panic("mux: duplicate call tag")
    }
    m.pending[tag] = done
    m.mu.Unlock()

    m.send <- args
    return <-done
}
```

Hint: Use goroutines, channels, and mutexes together if that is the clearest way to write the code.

Hints

Use the race detector, for development and even production.
Don’t communicate by sharing memory. Share memory by communicating.

Convert data state into code state when it makes programs clearer.
Convert mutexes into goroutines when it makes programs clearer.
Use additional goroutines to hold additional code state.
Use goroutines to let independent concerns run independently.

Consider the effect of slow goroutines.
Know why and when each communication will proceed.
Know why and when each goroutine will exit.
Type Ctrl-\ to kill a program and dump all its goroutine stacks.
Use the HTTP server’s /debug/pprof/goroutine to inspect live goroutine stacks.

Use a buffered channel as a concurrent blocking queue.
Think carefully before introducing unbounded queuing.
Close a channel to signal that no more values will be sent.

Use a mutex if that is the clearest way to write the code.
Prefer defer for unlocking mutexes.
Use a goto if that is the clearest way to write the code.
Use goroutines, channels, and mutexes together
if that is the clearest way to write the code.