Language-level
Concurrency Support:
Go

Chris Rossbach
Outline for Today

• Questions?

• Administrivia
  • Lab 3 looms large: Go go go!

• Agenda
  • Message Passing background
  • Concurrency in Go
  • Thoughts and guidance on Lab 3

• Acknowledgements: Rob Pike’s 2012 Go presentation is excellent, and I borrowed from it: https://talks.golang.org/2012/concurrency.slide
Faux Quiz questions

• How are promises and futures different or the same as goroutines
• What is the difference between a goroutine and a thread?
• What is the difference between a channel and a lock?
• How is a channel different from a concurrent FIFO?
• What is the CSP model?
• What are the tradeoffs between explicit vs implicit naming in message passing?
• What are the tradeoffs between blocking vs. non-blocking send/receive in a shared memory environment? In a distributed one?
Event-based Programming: Motivation

• Threads have a *lot* of down-sides:
  • Tuning parallelism for different environments
  • Load balancing/assignment brittle
  • Shared state requires locks →
    • Priority inversion
    • Deadlock
    • Incorrect synchronization
  • …

• Events: *restructure programming model to have no threads!*
Event-based Programming: Motivation

• Threads have a *lot* of down-sides:
  • Tuning parallelism for different environments
  • Load balancing/assignment brittle
  • Shared state requires locks →
    • Priority inversion
    • Deadlock
    • Incorrect synchronization
  • ...

• Events: *restructure programming model to have no threads!*
Message Passing: Motivation
Message Passing: Motivation

• Threads have a *lot* of down-sides:
  • Tuning parallelism for different environments
  • Load balancing/assignment brittle
  • Shared state requires locks →
    • Priority inversion
    • Deadlock
    • Incorrect synchronization
  • ...

Message Passing: Motivation

- Threads have a *lot* of down-sides:
  - Tuning parallelism for different environments
  - Load balancing/assignment brittle
  - Shared state requires locks →
    - Priority inversion
    - Deadlock
    - Incorrect synchronization
  - ...

Message Passing: Motivation

• Threads have a *lot* of down-sides:
  • Tuning parallelism for different environments
  • Load balancing/assignment brittle
  • Shared state requires locks →
    • Priority inversion
    • Deadlock
    • Incorrect synchronization
  • …

• Message passing:
  • *Threads aren’t the problem, shared memory is*
  • *restructure programming model to avoid communication through shared memory (and therefore locks)*
Message Passing
Message Passing

• Threads/Processes send/receive messages
Message Passing

• Threads/Processes send/receive messages
• Three design dimensions
  • Naming/Addressing: *how do processes refer to each other?*
  • Synchronization: *how to wait for messages (block/poll/notify)?*
  • Buffering/Capacity: *can messages wait in some intermediate structure?*
Naming: Explicit vs Implicit
Also: Direct vs Indirect
Naming: Explicit vs Implicit
Also: Direct vs Indirect

- Explicit Naming
  - Each process must explicitly name the other party
  - Primitives:
    - send(receiver, message)
    - receive(sender, message)
**Naming: Explicit vs Implicit**

Also: Direct vs Indirect

- **Explicit Naming**
  - Each process must explicitly name the other party
  - Primitives:
    - send(receiver, message)
    - receive(sender, message)

- **Implicit Naming**
  - Messages sent/received to/from mailboxes
  - Mailboxes may be named/shared
  - Primitives:
    - send(mailbox, message)
    - receive(mailbox, message)
Synchronization
Synchronization

• Synchronous vs. Asynchronous
  • Blocking send: sender blocks until received
  • Nonblocking send: send resumes before message received
  • Blocking receive: receiver blocks until message available
  • Non-blocking receive: receiver gets a message or null
Synchronization

• Synchronous vs. Asynchronous
  • Blocking send: sender blocks until received
  • Nonblocking send: send resumes before message received
  • Blocking receive: receiver blocks until message available
  • Non-blocking receive: receiver gets a message or null

Blocking:
+ simple
+ avoids wasteful spinning
- inflexible
- Can hide concurrency

Non-blocking:
+ maximal flexibility
- error handling/detection tricky
- interleaving useful work non-trivial
Synchronization

• Synchronous vs. Asynchronous
  • Blocking send: sender blocks until received
  • Nonblocking send: send resumes before message received
  • Blocking receive: receiver blocks until message available
  • Non-blocking receive: receiver gets a message or null

• If both send and receive block
  • “Rendezvous”
  • Operation acts as an ordering primitive
  • Sender knows receiver succeeded
  • Receiver knows sender succeeded
  • Particularly appealing in distributed environment

Blocking:
+ simple
+ avoids wasteful spinning
- inflexible
- Can hide concurrency

Non-blocking:
+ maximal flexibility
- error handling/detection tricky
- interleaving useful work non-trivial
Communicating Sequential Processes
Hoare 1978

CSP: language for multi-processor machines
• Non-buffered **message passing**
  • No shared memory
  • **Send/recv** are blocking
• **Explicit naming** of src/dest processes
  • Also called direct naming
  • Receiver **specifies source** process
  • Alternatives: **indirect**
    • Port, mailbox, queue, socket
• **Guarded** commands to let processes wait
Communicating Sequential Processes
Hoare 1978

CSP: language for multi-processor machines
• Non-buffered **message passing**
  • No shared memory
  • **Send/recv are blocking**
• **Explicit naming** of src/dest processes
  • Also called direct naming
  • Receiver **specifies source** process
  • Alternatives: **indirect**
    • Port, mailbox, queue, socket
• **Guarded** commands to let processes wait

← Transputer!
An important problem in the CSP model:
An important problem in the CSP model:

• Processes need to receive messages from different senders
An important problem in the CSP model:

- Processes need to receive messages from different senders
- Only primitive: blocking receive(<name>, message)
An important problem in the CSP model:

- Processes need to receive messages from different senders
- Only primitive: blocking receive(<name>, message)
An important problem in the CSP model:

- Processes need to receive messages from different senders
- Only primitive: blocking receive(<name>, message)

```plaintext
recv_multi(Q) {
    receive(Q, message)
    receive(R, message)
    receive(S, message)
}
```
An important problem in the CSP model:

- Processes need to receive messages from different senders
- Only primitive: blocking receive(<name>, message)

```
recv_multi(Q) {
  receive(Q, message)
  receive(R, message)
  receive(S, message)
}
```

Is there a problem with this?
An important problem in the CSP model:

- Processes need to receive messages from different senders
- Only primitive: blocking receive(<name>, message)

```
recv_multi(Q) {
    receive(Q, message)
    receive(R, message)
    receive(S, message)
}
```

Is there a problem with this?
Blocking with Indirect Naming

• Processes need to receive messages from different senders
  • *blocking receive* with *indirect naming*
    • Process waits on port, gets first message as first message arriving at that port
Blocking with Indirect Naming

- Processes need to receive messages from different senders
- *blocking receive* with *indirect naming*
  - Process waits on port, gets first message first message arriving at that port

receive(port, message)
Blocking with Indirect Naming

• Processes need to receive messages from different senders
• *blocking receive* with *indirect naming*
  • Process waits on port, gets first message first message arriving at that port

receive(port, message)

OK to block (good)
Requires indirection (less good)
Non-blocking with Direct Naming

- Processes need to receive messages from different senders
- *Non-blocking receive* with *direct naming*
  - Requires receiver to poll senders
Non-blocking with Direct Naming

• Processes need to receive messages from different senders

• *Non-blocking receive* with *direct naming*
  • Requires receiver to poll senders
Non-blocking with Direct Naming

• Processes need to receive messages from different senders
• *Non-blocking receive* with *direct naming*
  • Requires receiver to poll senders

```c
while(...) {
    try_receive(Q, message)
    try_receive(R, message)
    try_receive(S, message)
}
```
Non-blocking with Direct Naming

• Processes need to receive messages from different senders

• **Non-blocking receive** with **direct naming**
  • Requires receiver to poll senders

```c
while(...) {
    try_receive(Q, message)
    try_receive(R, message)
    try_receive(S, message)
}
```

Polling (bad)
No indirection (good)
Blocking and Direct Naming
Blocking and Direct Naming

- How to achieve it?
Blocking and Direct Naming

• How to achieve it?
• CSP provides abstractions/primitives for it
Alternative / Guarded Commands

Guarded command is **delayed** until either
- **guard succeeds** → cmd executes or
- **guard fails** → command aborts

Alternative command:
- list of one or more guarded commands
- separated by ”| |”
- surrounded by square brackets

\[
[x \geq y \rightarrow \text{max:= } x | | y \geq x \rightarrow \text{max:= } y]
\]

**Examples**

\[
n < 10 \rightarrow \text{A!index}(n); n := n + 1;
n < 10; \text{A?index}(n)\rightarrow \text{next = MyArray}(n);
\]
Alternative / Guarded Commands

Guarded command is **delayed** until either

- **guard succeeds** → cmd executes *or*
- **guard fails** → command aborts

Guarded Commands

\[
<\text{guard}> \rightarrow <\text{command list}>
\]

- boolean expression
- at most one `?`, must be at end of guard, considered true iff message pending

Examples

\[
n < 10 \rightarrow \text{A!index}(n); \ n := n + 1; \ \\
n < 10; \ \text{A?index}(n) \rightarrow \text{next = MyArray}(n);
\]

Alternative command:

- list of one or more guarded commands
- separated by "| |
- surrounded by square brackets

\[
[ \ x \geq y \rightarrow \text{max} := x \ | \ | \ y \geq x \rightarrow \text{max} := y \ ]
\]

- Enable *choice* preserving concurrency
- *Hugely influential*
- goroutines, channels, select, defer:
  - *Trying to achieve the same thing*
Go Concurrency

• CSP: the root of many languages
  • Occam, Erlang, Newsqueak, Concurrent ML, Alef, Limbo

• Go is a Newsqueak-Alef-Limbo derivative
  • Distinguished by *first class channel support*
  • Program: *goroutines* communicating through *channels*
  • Guarded and alternative-like constructs in *select* and *defer*
A boring function

```go
func boring(msg string) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}

func main() {
    boring("boring!")
}
```
A boring function

```go
func boring(msg string) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}

func main() {
    boring("boring!")
}
```
Ignoring a boring function

- Go statement runs the function
- Doesn’t make the caller wait
- Launches a goroutine
- Analogous to & on shell command

```go
package main

import (  
    "fmt"  
    "math/rand"  
    "time"  
)

func main() {  
    go boring("boring!")  
}
```
Ignoring a boring function

- Go statement runs the function
- Doesn’t make the caller wait
- Launches a goroutine
- Analagous to & on shell command

- Keep main() around a while
- See goroutine actually running
Ignoring a boring function

- Go statement runs the function
- Doesn’t make the caller wait
- Launches a goroutine
- Analagous to & on shell command

- Keep main() around a while
- See goroutine actually running
Goroutines
Goroutines

• Independently executing function launched by go statement
Goroutines

• Independently executing function launched by go statement
• Has own call stack
Goroutines

• Independently executing function launched by go statement
• Has own call stack
• Cheap: Ok to have 1000s...100,000s of them
Goroutines

• Independently executing function launched by go statement
• Has own call stack
• Cheap: Ok to have 1000s...100,000s of them
• Not a thread
  • One thread may have 1000s of go routines!
Goroutines

- Independently executing function launched by go statement
- Has own call stack
- Cheap: Ok to have 1000s...100,000s of them
- Not a thread
  - One thread may have **1000s** of go routines!
- Multiplexed onto threads as needed to ensure forward progress
  - Deadlock detection built in
Channels

• Connect goroutines allowing them to communicate

```go
// Declaring and initializing.
var c chan int
c = make(chan int)
// or
c := make(chan int)

// Sending on a channel.
c <- 1

// Receiving from a channel.
// The "arrow" indicates the direction of data flow.
value = <-c
```
Channels

• Connect goroutines allowing them to communicate
Channels

• Connect goroutines allowing them to communicate

```go
func main() {
    c := make(chan string)
    go boring("boring!", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: \%q\n", <-c) // Receive expression is just a value.
    }
    fmt.Println("You're boring; I'm leaving.")
}

func boring(msg string, c chan string) {
    for i := 0; ; i++ {
        c <- fmt.Sprintf("%s %d", msg, i) // Expression to be sent can be any suitable value.
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```
Channels

- Connect goroutines allowing them to communicate

```go
func main() {
    c := make(chan string)
    go boring("boring!", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <-c) // Receive expression is just a value.
    }
    fmt.Println("You're boring; I'm leaving.")
}

func boring(msg string, c chan string) {
    for i := 0; ; i++ {
        c <- fmt.Sprintf("%s %d", msg, i) // Expression to be sent can be any sorted type, here string, integer
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

You say: "boring! 0"
You say: "boring! 1"
You say: "boring! 2"
You say: "boring! 3"
You say: "boring! 4"
You're boring; I'm leaving.

Program exited.
Channels

• Connect goroutines allowing them to communicate

```go
func main() {
    c := make(chan string)
    go boring("boring!", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <-c) // Receive expression is just a value.
    }
    fmt.Println("You're boring; I'm leaving.")
}
```

```go
func boring(msg string, c chan string) {
    for i := 0; i++ {
        c <- fmt.Sprintf("%s %d", msg, i) // Expression to be sent can be any
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

• When main executes <-c, it blocks
• When boring executes c <- value it blocks
• Channels communicate and synchronize
Select: Handling Multiple Channels

• All channels are evaluated
• Select blocks until one communication can proceed
  • Cf. Linux select system call, Windows WaitForMultipleObjectsEx
  • Cf. Alternatives and guards in CPS
• If multiple can proceed select chooses randomly
• Default clause executes immediately if no ready channel

```go
select {
    case v1 := <-c1:
        fmt.Printf("received %v from c1\n", v1)
    case v2 := <-c2:
        fmt.Printf("received %v from c2\n", v1)
    case c3 <- 23:
        fmt.Printf("sent %v to c3\n", 23)
    default:
        fmt.Printf("no one was ready to communicate\n")
}
Select: Handling Multiple Channels

- All channels are evaluated
- Select blocks until one communication can proceed
  - Cf. Linux select system call, Windows WaitForMultipleObjectsEx
  - Cf. Alternatives and guards in CPS
- If multiple can proceed select chooses randomly
- Default clause executes immediately if no ready channel

```go
select {
    case v1 := <-c1:
        fmt.Printf("received %v from c1\n", v1)
    case v2 := <-c2:
        fmt.Printf("received %v from c2\n", v1)
    case c3 <- 23:
        fmt.Printf("sent %v to c3\n", 23)
    default:
        fmt.Printf("no one was ready to communicate\n")
}
```

Without default clause becomes rendezvous!
Google Search

• Workload:
• Accept query
• Return page of results (with ugh, ads)
• Get search results by sending query to
  • Web Search
  • Image Search
  • YouTube
  • Maps
  • News, etc
• How to implement this?
Search 1.0

• Google function takes query and returns a slice of results (strings)
• Invokes Web, Image, Video search serially

```go
func Google(query string) (results []Result) {
    results = append(results, Web(query))
    results = append(results, Image(query))
    results = append(results, Video(query))
    return
}
```
Search 2.0

• Run Web, Image, Video searches concurrently, wait for results
• No locks, conditions, callbacks

```go
func Google(query string) (results []Result) {
    c := make(chan Result)
    go func() { c <- Web(query) } ()
    go func() { c <- Image(query) } ()
    go func() { c <- Video(query) } ()

    for i := 0; i < 3; i++ {
        result := <-c
        results = append(results, result)
    }
    return
}
```
Search 2.1

- Don’t wait for slow servers: No locks, conditions, callbacks!

```go
c := make(chan Result)
go func() { c <- Web(query) } ()
go func() { c <- Image(query) } ()
go func() { c <- Video(query) } ()

timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
    select {
        case result := <-c:
            results = append(results, result)
        case <-timeout:
            fmt.Println("timed out")
            return
    }
}
return
```
Search 3.0

- Reduce tail latency with replication. No locks, conditions, callbacks!

c := make(chan Result)
go func() { c <- First(query, Web1, Web2) } ()
go func() { c <- First(query, Image1, Image2) } ()
go func() { c <- First(query, Video1, Video2) } ()
timeout := time.After(80 * time.Millisecond)
for i := 0; i < 3; i++ {
  select {
    case result := <-c:
      results = append(results, result)
    case <-timeout:
      fmt.Println("timed out")
      return
  }
}
return
Other tools in Go

• Goroutines and channels are the main primitives
• Sometimes you just need a reference counter or lock
  • “sync” and “sync/atomic” packages
  • Mutex, condition, atomic operations
• Sometimes you need to wait for a go routine to finish
  • Didn’t happen in any of the examples in the slides
  • WaitGroups are key
func testQ() {
    var wg sync.WaitGroup
    wg.Add(4)
    ch := make(chan int)
    for i := 0; i < 4; i++ {
        go func(id int) {
            aval, amore := <- ch
            if amore {
                fmt.Printf("reader #%d got %d value\n", id, aval)
            } else {
                fmt.Printf("channel reader #%d terminated with nothing.\n", id)
            }
            wg.Done()
        }(i)
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
}
WaitGroups

```go
func testQ() {
    var wg sync.WaitGroup
    wg.Add(4)
    ch := make(chan int)
    for i := 0; i < 4; i++ {
        go func(id int) {
            aval, amore := <- ch
            if amore {
                fmt.Printf("reader #%d got %d value\n", id, aval)
            } else {
                fmt.Printf("channel reader #%d terminated with nothing.\n", id)
            }
        }(i)
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
}
```
Go: magic or threadpools and concurrent Qs?

- We’ve seen several abstractions for
  - Control flow/exection
  - Communication
- Lots of discussion of pros and cons
- Ultimately still CPUs + instructions
- Go: just sweeping issues under the language interface?
  - Why is it OK to have 100,000s of goroutines?
  - Why isn’t composition an issue?
Go implementation details
Go implementation details

• $M = \text{“machine”} \rightarrow \text{OS thread}$
Go implementation details

- M = “machine” → OS thread
- P = (processing) context
Go implementation details

- \( M = \text{“machine”} \rightarrow \text{OS thread} \)
- \( P = \text{(processing) context} \)
- \( G = \text{goroutines} \)
Go implementation details

- \( M = \text{“machine”} \) → OS thread
- \( P = \text{(processing) context} \)
- \( G = \text{goroutines} \)
- Each ‘M’ has a queue of goroutines
Go implementation details

- M = “machine” → OS thread
- P = (processing) context
- G = goroutines
- Each ‘M’ has a queue of goroutines
- Goroutine scheduling is cooperative
  - Switch out on complete or block
  - Very light weight (fibers!)
  - Scheduler does work-stealing
Go implementation details

- M = “machine” → OS thread
- P = (processing) context
- G = goroutines
Go implementation details

- M = “machine” → OS thread
- P = (processing) context
- G = goroutines
- Each ‘M’ has a queue of goroutines
- Goroutine scheduling is cooperative
  - Switch out on complete or block
  - Very light weight (fibers!)
  - Scheduler does work-stealing
Go implementation details

- M = “machine” → OS thread
- P = (processing) context
- G = goroutines
  - Each M has a queue of goroutines
  - Goroutine scheduling is cooperative
    - Switch out on complete or block
    - Very lightweight (fibers!)
    - Scheduler does work - stealing

```go
struct M {
    G* curg;       // current running goroutine
    int32 id;      // unique id
    int32 locks;   // locks held by this M
    MCache *mcache; // cache for this thread
    G* lockedg;    // used for locking M’s and G’s
    uintptr createstack [32]; // Stack that created this thread
    M* nextwaitm;  // next M waiting for lock
};
```
Go implementation details

- $M$ = “machine” $\rightarrow$ OS thread
- $P$ = (processing) context
- $G$ = goroutines
- Each ‘$M$’ has a queue of goroutines
- Goroutine scheduling is cooperative
  - Switch out on complete or block
  - Very lightweight (fibers!)
  - Scheduler does work - stealing

```
struct Sched {
    Lock; // global sched lock.
    // must be held to edit G or M queues
    G *gfree; // available g’s (status == Gdead)
    G *ghead; // g’s waiting to run queue
    G *gtail; // tail of g’s waiting to run queue
    int32 gwait; // number of g’s waiting to run
    int32 gcount; // number of g’s that are alive
    int32 grunning; // number of g’s running on cpu
    // or in syscall
    M *mhead; // m’s waiting for work
    int32 mwait; // number of m’s waiting for work
    int32 mcount; // number of m’s that have been created
    ...
};
```
Go implementation details

- $M$ = “machine” $\rightarrow$ OS thread
- $P$ = (processing) context
- $G$ = goroutines
- Each $M$ has a queue of goroutines
- Goroutine scheduling is cooperative
  - Switch out on complete or block
  - Very light weight (fibers!)
- Scheduler does work
  - stealing

```go
def Struct Sched {
    Lock; // global sched lock.
    // must be held to edit G or M queues
    G *gfree; // available g’s (status == Gdead)
    G *ghead; // g’s waiting to run queue
    G *gtail; // tail of g’s waiting to run queue
    int32 gwait; // number of g’s waiting to run
    int32 gcount; // number of g’s that are alive
    int32 grunning; // number of g’s running on cpu
    // or in syscall
    M *mhead; // m’s waiting for work
    int32 mwait; // number of m’s waiting for work
    int32 mcount; // number of m’s that have been created
    ...
}
```
Go implementation details

- **M** = “machine” → OS thread
- **P** = (processing) context
- **G** = goroutines
- Each ‘M’ has a queue of goroutines
- Goroutine scheduling is cooperative
  - Switch out on complete or block
  - Very light weight (fibers!)
  - Scheduler does work
- Struct Sched {
  Lock; // global sched lock.
  // must be held to edit G or M queues
  G *gfree; // available g’s (status == Gdead)
  G *ghead; // g’s waiting to run queue
  G *gtail; // tail of g’s waiting to run queue
  int32 gwait; // number of g’s waiting to run
  int32 gcount; // number of g’s that are alive
  int32 grunning; // number of g’s running on cpu
  // or in syscall
  M *mhead; // m’s waiting for work
  int32 mwait; // number of m’s waiting for work
  int32 mcount; // number of m’s that have been created
  ...
}
func testQ(consumers int) {
    startTimes["testQ"] = time.Now()
    var wg sync.WaitGroup
    wg.Add(consumers)
    ch := make(chan int)
    for i := 0; i < consumers; i++ {
        go func(id int) {
            aval, amore := <- ch
            if amore {
                info("reader %d got %d value\n", id, aval)
            } else {
                info("channel reader %d terminated with nothing.\n", id)
            }
            wg.Done()
        }(i)
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
    stopTimes["testQ"] = time.Now()
}
func testQ(consumers int) {
    startTimes["testQ"] = time.Now()
    var wg sync.WaitGroup
    wg.Add(consumers)
    ch := make(chan int)
    for i := 0; i < consumers; i++ {
        go func(id int) {
            aval, amore := <- ch
            if amore {
                info("reader #%d got %d value\n", id, aval)
            } else {
                info("channel reader #%d terminated with nothing.\n", id)
            }
            wg.Done()
        }(i)
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
    stopTimes["testQ"] = time.Now()
}
1000s of go routines?

- Creates a channel
- Creates “consumers” goroutines
- Each of them tries to read from the channel
- Main either:
  - Sleeps for 1 second, closes the channel
  - Sends “consumers” values
Channel implementation

• You can just read it:
  • [https://golang.org/src/runtime/chan.go](https://golang.org/src/runtime/chan.go)
  • Some highlights
Channel implementation

```go
func chansend(c *chan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    if c == nil {
        if !block {
            return false
        }
        gopark(nil, nil, "chan send (nil chan)", traceEvGoStop, 2)
        throw("unreachable")
    }

    if debugChan {
        print("chansend: chan=", c, "\n")
    }

    if raceenabled {
        racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
    }
}
```
Channel implementation

You can just read it:

• https://golang.org/src/runtime/chan.go

Some highlights

Race detection! Cool!

if raceenabled {
    racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
}
Channel implementation

- You can just read it: [https://golang.org/src/runtime/chan.go](https://golang.org/src/runtime/chan.go)
- Some highlights
Channel implementation

- You can just read it:
  - https://golang.org/src/runtime/chan.go
- Some highlights:

```go
if sg := c.recvq.dequeue(); sg != nil {
    // Found a waiting receiver. We pass the value we want to send
    // directly to the receiver, bypassing the channel buffer (if any).
    send(c, sg, ep, func() { unlock(&c.lock ), 3)
    return true
```
Channel implementation

- You can just read it:
  - [https://golang.org/src/runtime/chan.go](https://golang.org/src/runtime/chan.go)
- Some highlights
Channel implementation

- You can just read it: https://golang.org/src/runtime/chan.go
- Some highlights
Channel implementation

- You can just read it: 
  - https://golang.org/src/runtime/chan.go
- Some highlights
Channel implementation

- You can just read it:
  - [https://golang.org/src/runtime/chan.go](https://golang.org/src/runtime/chan.go)
- Some highlights

```
// Sends and receives on unbuffered or empty-buffered channels are the
// only operations where one running goroutine writes to the stack of
// another running goroutine. The GC assumes that stack writes only
// happen when the goroutine is running and are only done by that
// goroutine. Using a write barrier is sufficient to make up for
// violating that assumption, but the write barrier has to work.
// typedmemmove will call bulkBarrierPreWrite, but the target bytes
// are not in the heap, so that will not help. We arrange to call
// memmove and typeBitsBulkBarrier instead.

func sendDirect(t *type, sg *sudog, src unsafe.Pointer) {
    // src is on our stack, dst is a slot on another stack.

    // Once we read sg.elem out of sg, it will no longer
    // be updated if the destination's stack gets copied (shrunk).
    // So make sure that no preemption points can happen between read & use.
    dst := sg.elem
    typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
    memmove(dst, src, t.size)
```

per-goroutine stacks

G1 writes to G2’s stack!
Channel implementation

- You can just read it:
  - https://golang.org/src/runtime/chan.go
- Some highlights

Transputers did this in hardware in the 90s btw.
Channel implementation

- You can just read it:
  - https://golang.org/src/runtime/chan.go

- Some highlights:
  - Race detection built in
  - Fast path just write to receiver stack
  - Often has no capacity → scheduler hint!
  - Buffered channel implementation fairly standard
Go: Sliced Bread 2.0?
Go: Sliced Bread 2.0?

• Lacks compile-time generics
Go: Sliced Bread 2.0?

• Lacks compile-time generics
  • Results in code duplication
Go: Sliced Bread 2.0?

- Lacks compile-time generics
  - Results in code duplication
  - Metaprogramming cannot be statically checked
Go: Sliced Bread 2.0?

• Lacks compile-time generics
  • Results in code duplication
  • Metaprogramming cannot be statically checked
  • Standard library cannot offer generic algorithms
Go: Sliced Bread 2.0?

• Lacks compile-time generics
  • Results in code duplication
  • Metaprogramming cannot be statically checked
  • Standard library cannot offer generic algorithms

• Lack of language extensibility makes certain tasks more verbose
Go: Sliced Bread 2.0?

• Lacks compile-time generics
  • Results in code duplication
  • Metaprogramming cannot be statically checked
  • Standard library cannot offer generic algorithms

• Lack of language extensibility makes certain tasks more verbose
  • Lacks operator overloading (Java)
Go: Sliced Bread 2.0?

• Lacks compile-time generics
  • Results in code duplication
  • Metaprogramming cannot be statically checked
  • Standard library cannot offer generic algorithms

• Lack of language extensibility makes certain tasks more verbose
  • Lacks operator overloading (Java)

• Pauses and overhead of garbage collection
Go: Sliced Bread 2.0?

• Lacks compile-time generics
  • Results in code duplication
  • Metaprogramming cannot be statically checked
  • Standard library cannot offer generic algorithms

• Lack of language extensibility makes certain tasks more verbose
  • Lacks operator overloading (Java)

• Pauses and overhead of garbage collection
  • Limit Go’s use in systems programming compared to languages with manual memory management
Go: Sliced Bread 2.0?

• Lacks compile-time generics
  • Results in code duplication
  • Metaprogramming cannot be statically checked
  • Standard library cannot offer generic algorithms

• Lack of language extensibility makes certain tasks more verbose
  • Lacks operator overloading (Java)

• Pauses and overhead of garbage collection
  • Limit Go’s use in systems programming compared to languages with manual memory management

• Right tradeoffs? None of these problems have to do with concurrency!
Questions?