

Language-level Concurrency Support: Go

Chris Rossbach

cs378h

Outline for Today

- Questions?
- Administrivia
 - Lab 3 looms large: Go go go!
 - K-Means discussion → office hours
- 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!*

Remember
this slide?

~~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!*

Remember
this slide?

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
 - ...

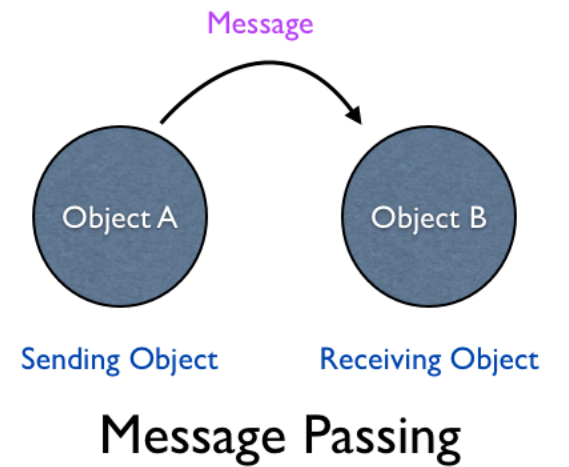
Message Passing: Motivation

- 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

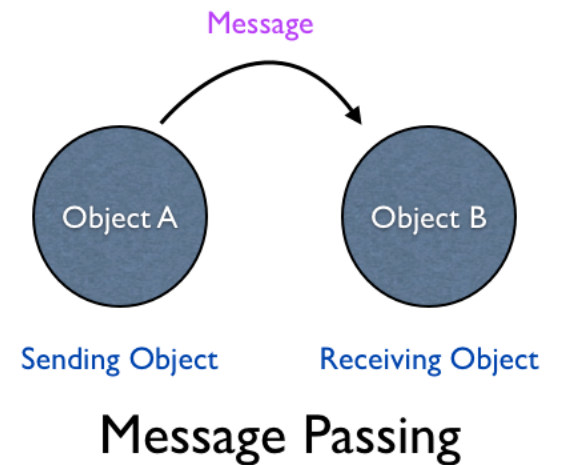
- Threads have a **lot** of down-sides:
 - Tuning parallelism for different environments
 - Load balancing/assignment brittle
 - Shared state requires locks →
 - Priority inversion
 - Deadlock
 - Incorrect synchronization
 - ...
- Message passing:
 - *Threads aren't the problem, shared memory is*
 - *restructure programming model to avoid communication through shared memory (and therefore locks)*

Message Passing



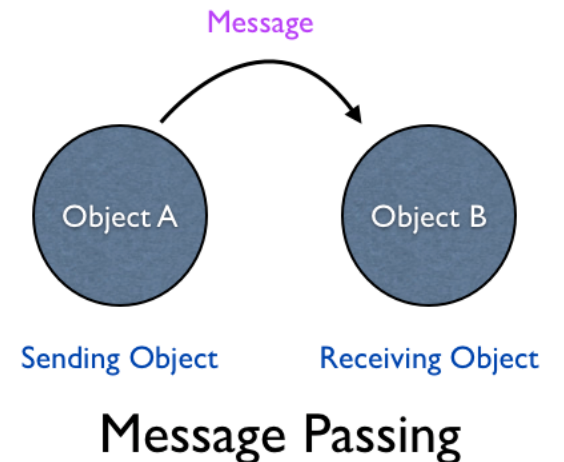
Message Passing

- 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

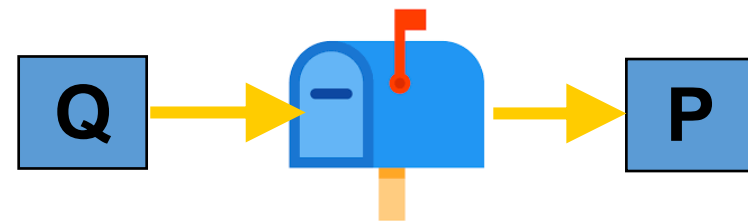
- Explicit Naming

- 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:

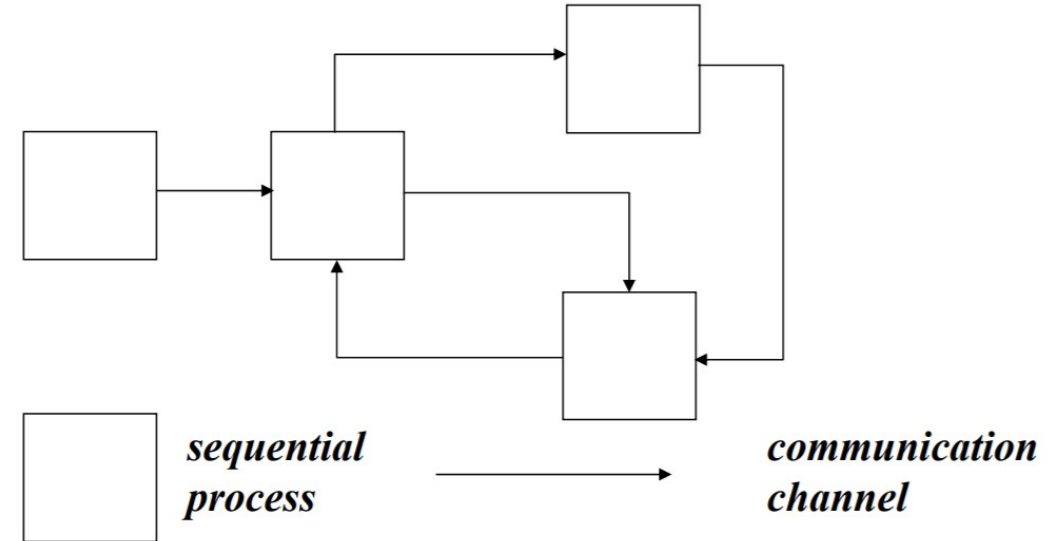
- + maximal flexibility
- error handling/detection tricky
- interleaving useful work non-trivial

Communicating Sequential Processes

Hoare 1978

CSP: language for multi-processor machines

- Non-buffered **message passing**
 - No shared memory
 - **Send/rcv 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



- single thread of control
- autonomous
- encapsulated
- named
- static

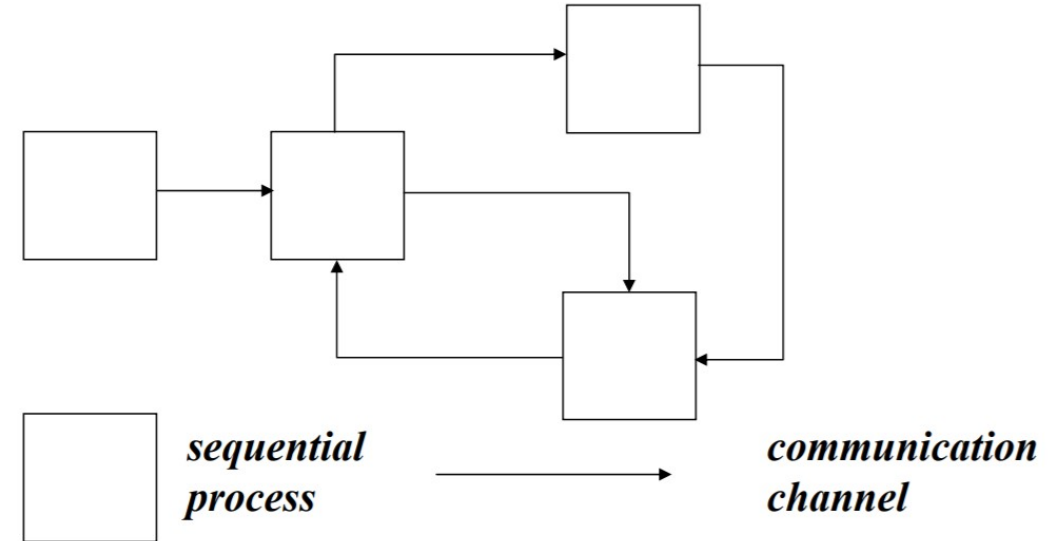
- synchronous
- reliable
- unidirectional
- point-to-point
- fixed topology

Communicating Sequential Processes

Hoare 1978

CSP: language for multi-processor machines

- Non-buffered **message passing**
 - No shared memory
 - **Send/rcv 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



- single thread of control
- autonomous
- encapsulated
- named
- static

- synchronous
- reliable
- unidirectional
- point-to-point
- fixed topology



← Transputer!

An important problem in the CSP model:

An important problem in the CSP model:

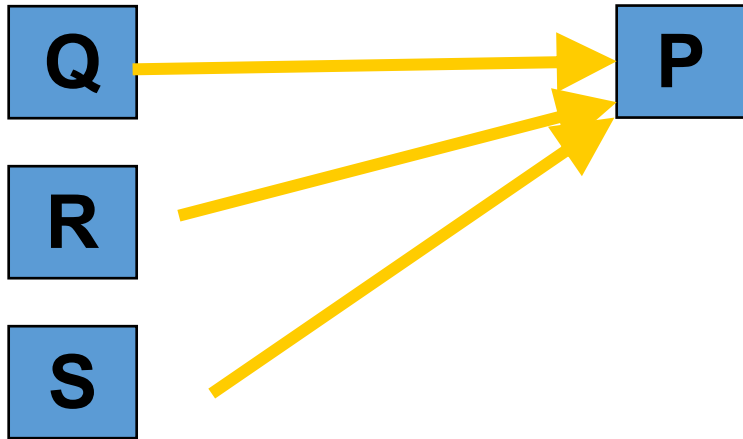
- Processes need to receive messages from different senders

An important problem in the CSP model:

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

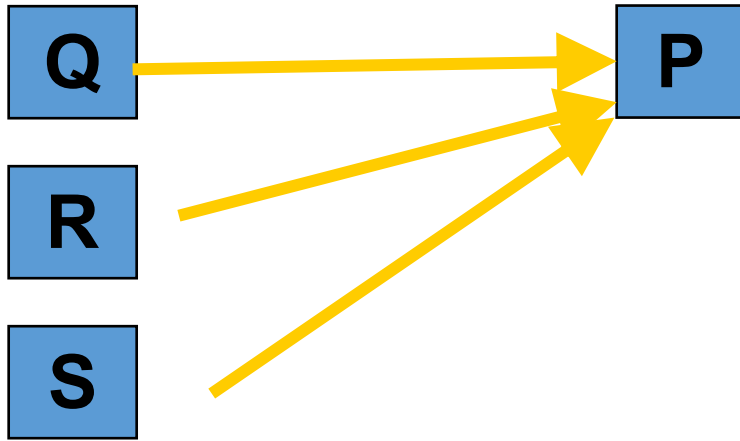
An important problem in the CSP model:

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



An important problem in the CSP model:

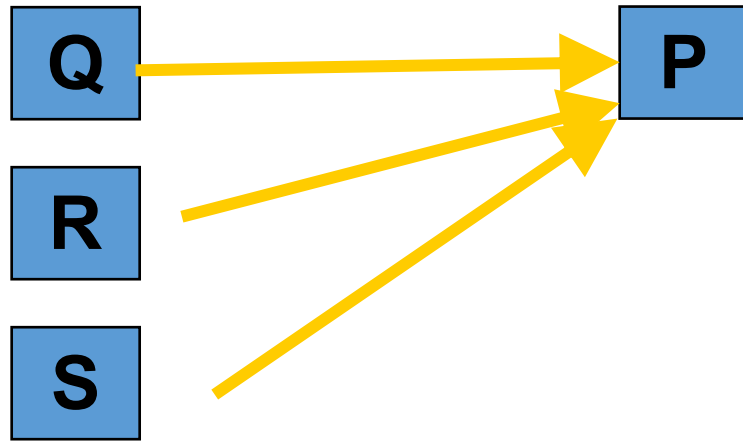
- Processes need to receive messages from different senders
- Only primitive: blocking `receive(<name>, message)`



```
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)`

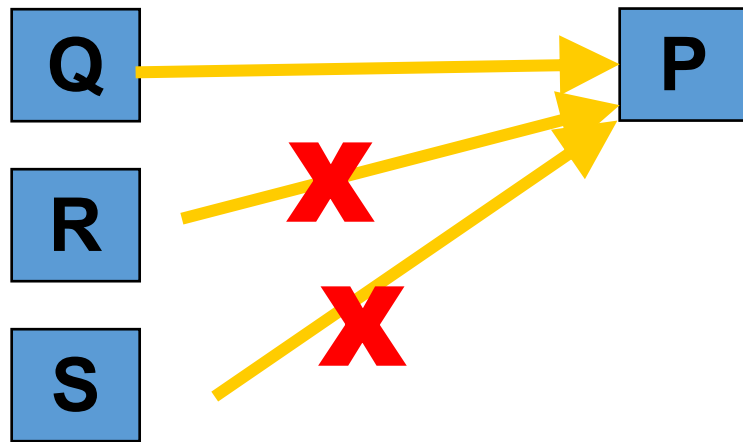


```
recv_multi(Q) {  
  receive(Q, message)  
  receive(R, message)  
  receive(S, message)  
}
```

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)  
}
```

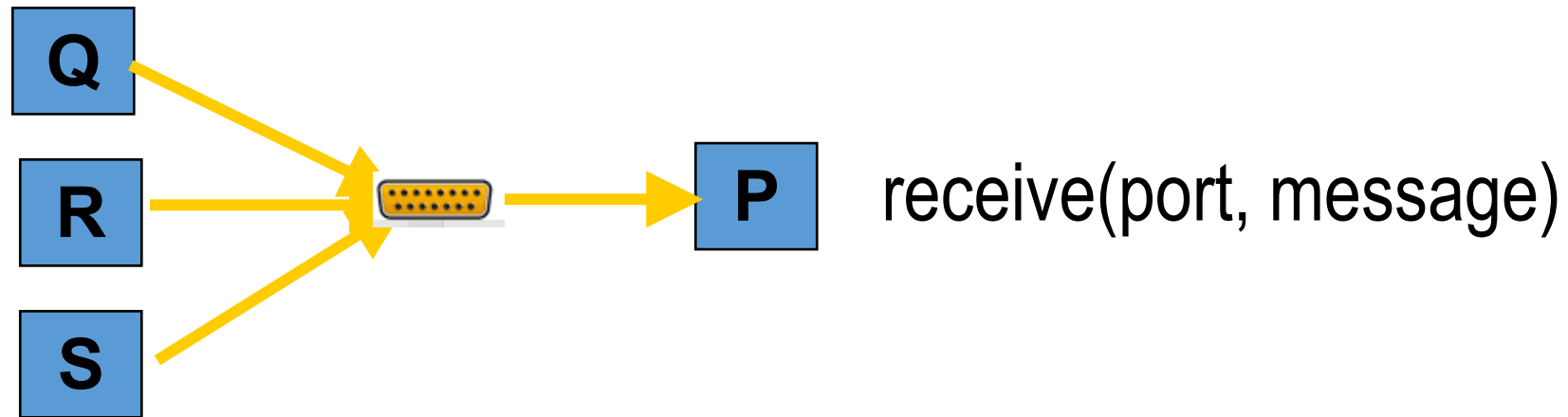
Is there a problem
with this?

Blocking with Indirect Naming

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

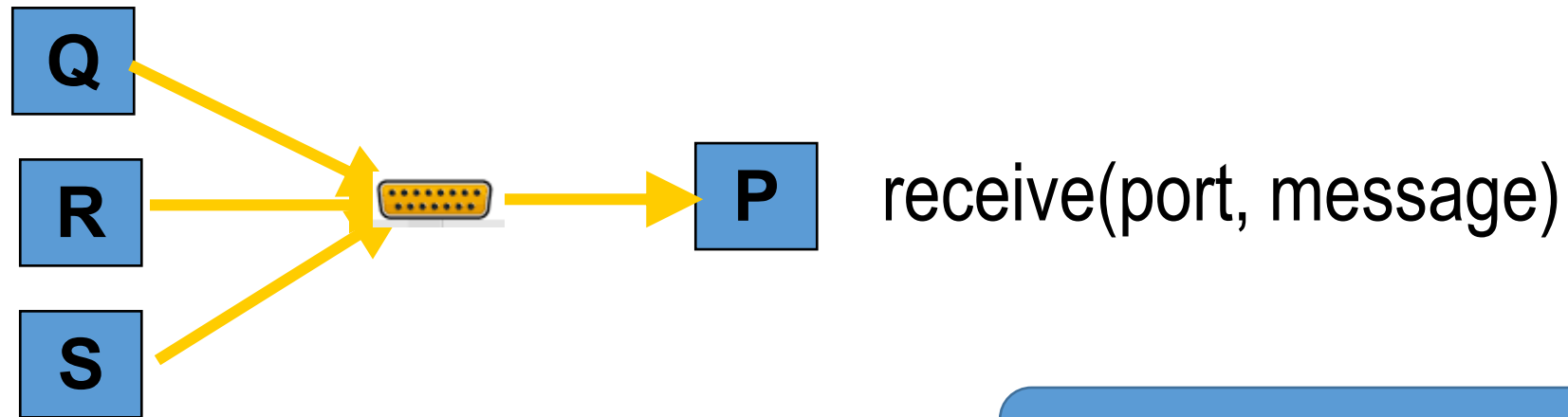
Blocking with Indirect Naming

- 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



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



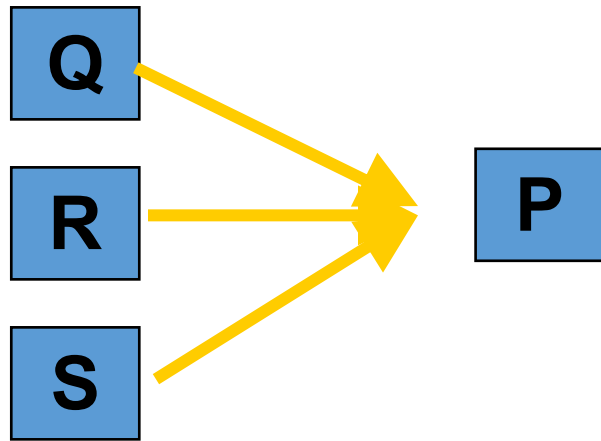
OK to block (good)
Requires indirection (less good)

Non-blocking with Direct Naming

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

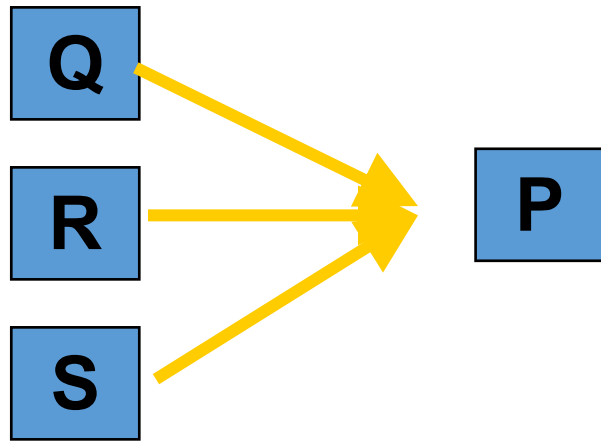
Non-blocking with Direct Naming

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



Non-blocking with Direct Naming

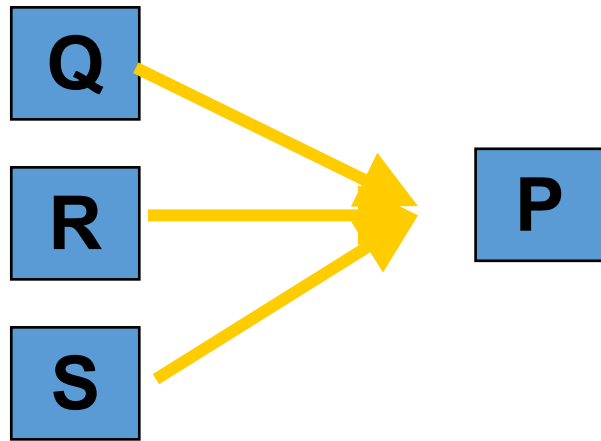
- Processes need to receive messages from different senders
- ***Non-blocking receive*** with ***direct naming***
 - Requires receiver to poll senders



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

Non-blocking with Direct Naming

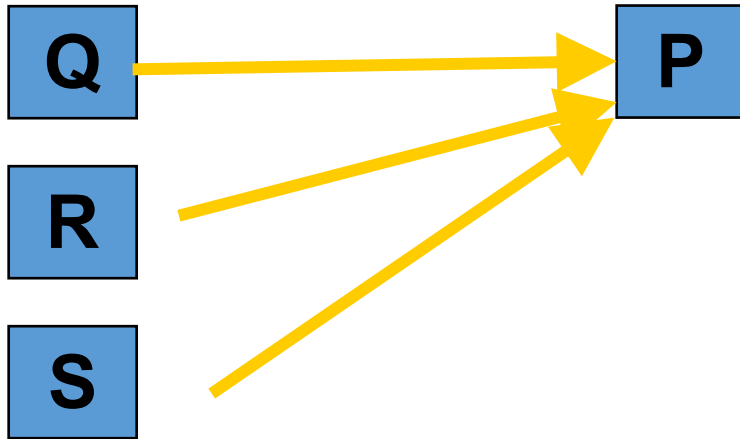
- Processes need to receive messages from different senders
- ***Non-blocking receive*** with ***direct naming***
 - Requires receiver to poll senders



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

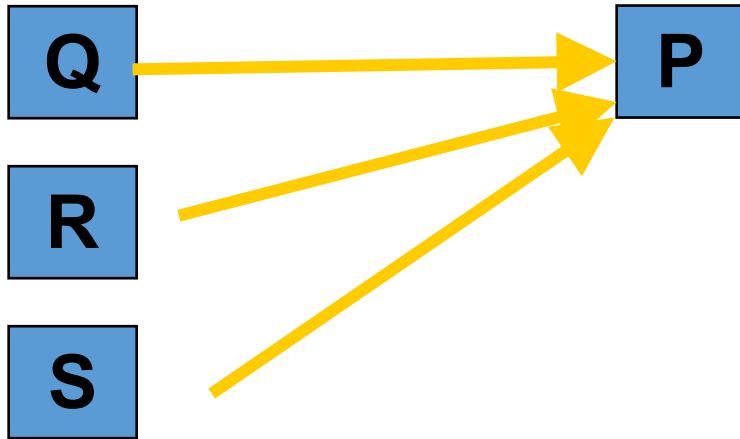
Polling (bad)
No indirection (good)

Blocking and Direct Naming



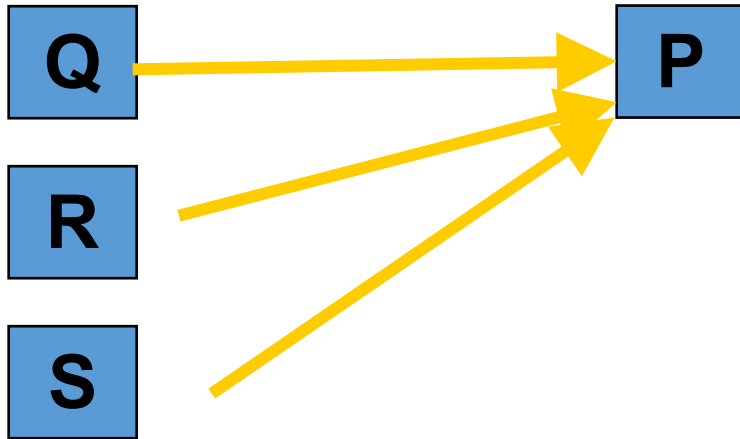
Blocking and Direct Naming

- How to achieve *it*?



Blocking and Direct Naming

- How to achieve *it*?
- ***CSP provides abstractions/primitives for it***



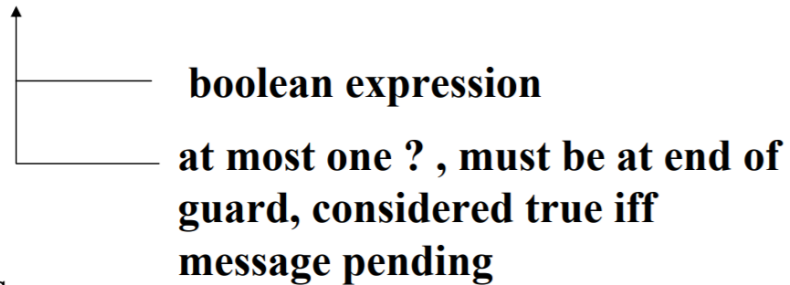
Alternative / Guarded Commands

Guarded command is **delayed** until either

- **guard succeeds** \rightarrow cmd executes *or*
- **guard fails** \rightarrow command aborts

Guarded Commands

$\langle \text{guard} \rangle \rightarrow \langle \text{command list} \rangle$



Examples

$n < 10 \rightarrow A!index(n); n := n + 1;$
 $n < 10; A?index(n) \rightarrow next = MyArray(n);$

Alternative command:

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

$[x \geq y \rightarrow max := x \ || \ y \geq x \rightarrow max := y]$

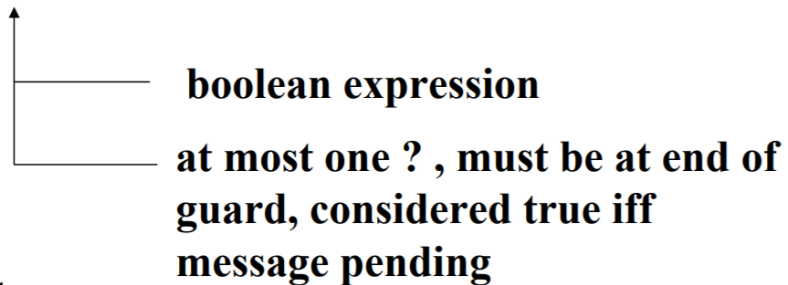
Alternative / Guarded Commands

Guarded command is **delayed** until either

- **guard succeeds** \rightarrow cmd executes *or*
- **guard fails** \rightarrow command aborts

Guarded Commands

$\langle \text{guard} \rangle \rightarrow \langle \text{command list} \rangle$



Examples

```
n < 10  $\rightarrow$  A!index(n); n := n + 1;  
n < 10; A?index(n)  $\rightarrow$  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

```
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

```
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!")  
}
```

```
boring! 0  
boring! 1  
boring! 2  
boring! 3  
boring! 4  
boring! 5
```

Ignoring a boring function

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

```
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

```
package main

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

func main() {
    go boring("boring!")
}
```

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

```
func main() {
    go boring("boring!")
    fmt.Println("I'm listening.")
    time.Sleep(2 * time.Second)
    fmt.Println("You're boring; I'm leaving.")
}
```

Ignoring a boring function

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

```
package main

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

func main() {
    go boring("bor")
}
```

```
I'm listening.
boring! 0
boring! 1
boring! 2
boring! 3
boring! 4
boring! 5
You're boring; I'm leaving.
Program exited.
```

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

```
func main() {
    go boring("boring!")
    fmt.Println("I'm listening.")
    time.Sleep(2 * time.Second)
    fmt.Println("You're boring; I'm leaving.")
}
```

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

```
// 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

```
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

```
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 s
        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

- When main executes `<-c`, it blocks
- When boring executes `c <- value` it blocks
- Channels communicate *and synchronize*

```
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 s
        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.

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

```
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")
}
```

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

```
func Google(query string) ([]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

```
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!

```
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
```

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result)
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

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

WaitGroups

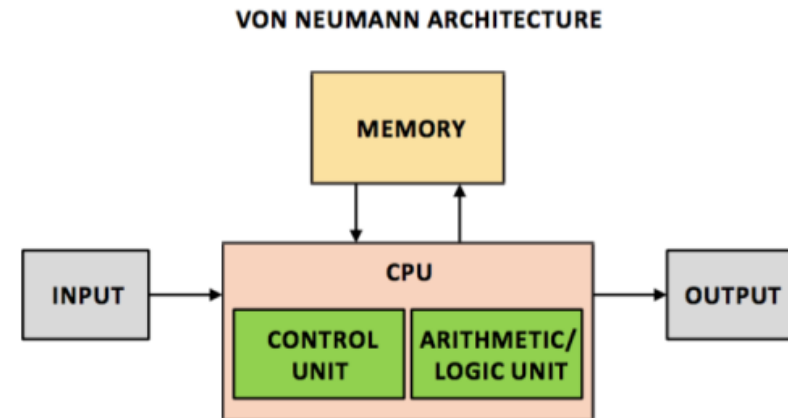
```
func testQ() {
    var wg sync.WaitGroup
    wg.Add(4)
    ch := make(chan int)
    for i:=0; i<4; i++ {
        go func(id int) {
            aval, amore := <- ch
            if(amore) {
                fmt.Printf("reader #%d got %d value\n", id, aval)
            } else {
                fmt.Printf("channel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

WaitGroups

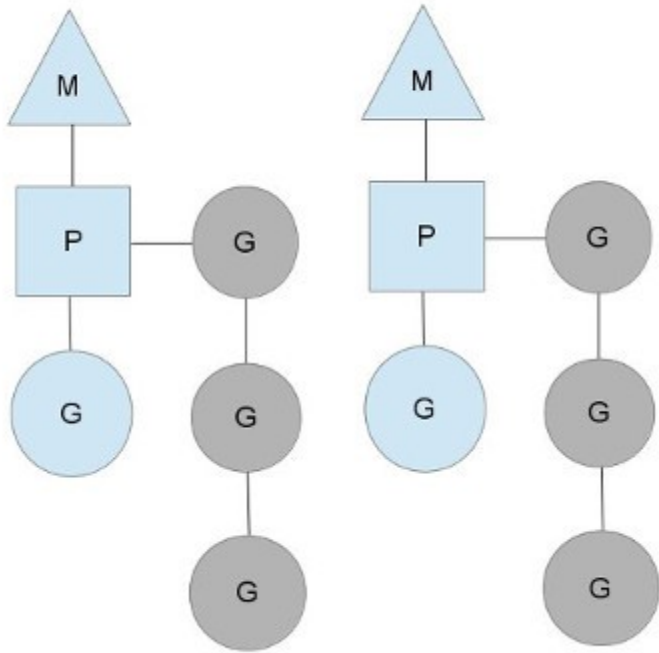
```
func testQ() {
    var wg sync.WaitGroup
    wg.Add(4)
    ch := make(chan int)
    for i:=0; i<4; i++ {
        go func(id int) {
            aval, amore := <- ch
            if(amore) {
                fmt.Printf("reader #%d got %d value\n", id, aval)
            } else {
                fmt.Printf("channel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

Go: magic or threadpools and concurrent Qs?

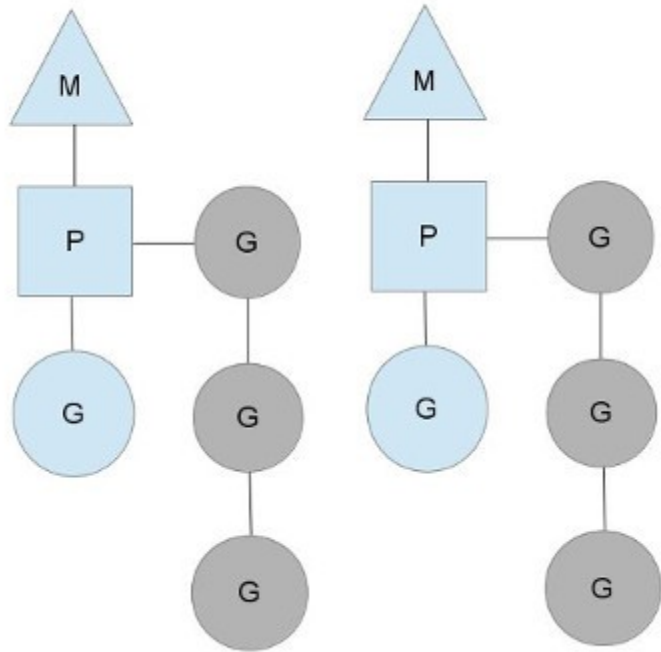
- We've seen several abstractions for
 - Control flow/execution
 - Communication
- Lots of discussion of pros and cons
- Ultimately still CPUs + instructions
- Go: just sweeping issues under the language interface?
 - Why is it OK to have 100,000s of goroutines?
 - Why isn't composition an issue?



Go implementation details

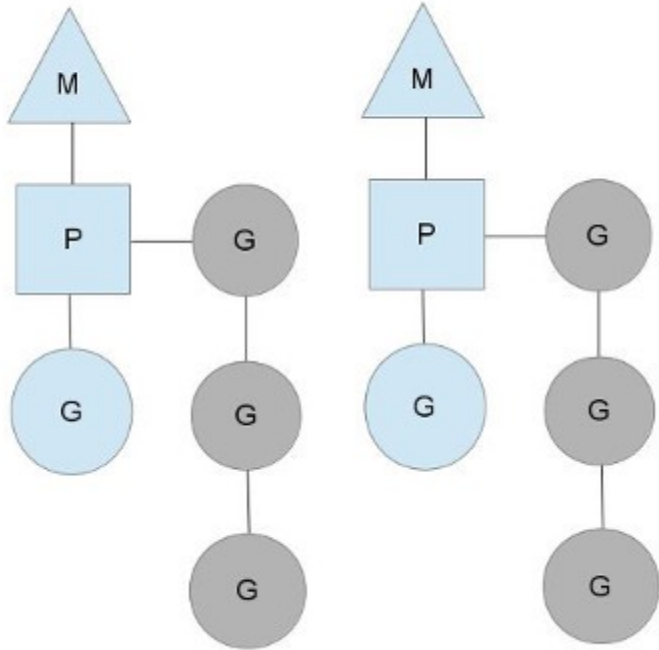


Go implementation details



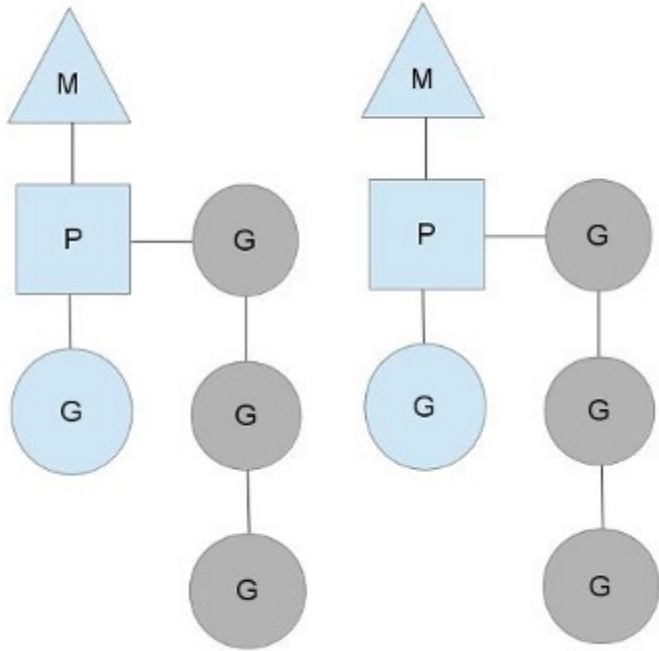
- M = “machine” → OS thread

Go implementation details



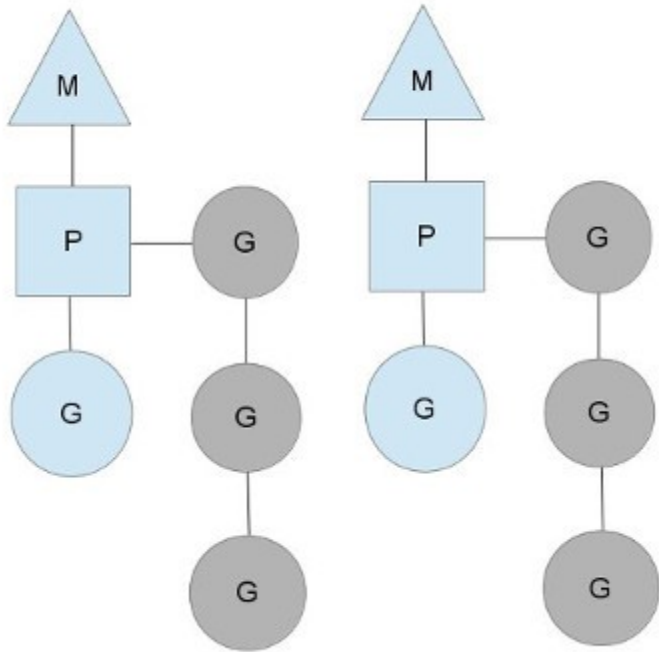
- M = “machine” → OS thread
- P = (processing) context

Go implementation details



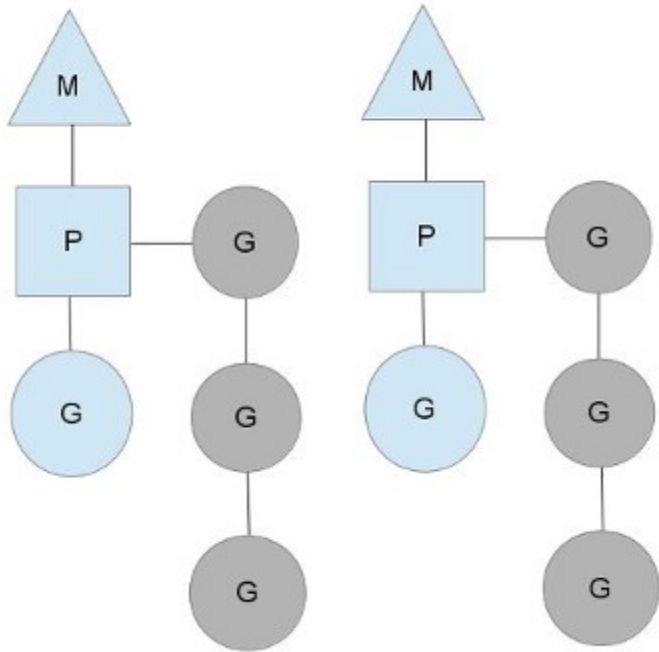
- M = “machine” → OS thread
- P = (processing) context
- G = goroutines

Go implementation details



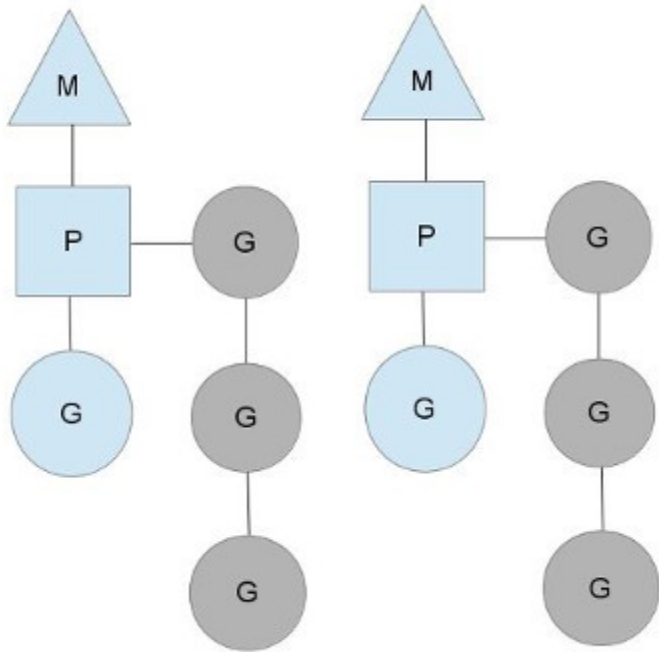
- M = “machine” → OS thread
- P = (processing) context
- G = goroutines
- Each ‘M’ has a queue of goroutines

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

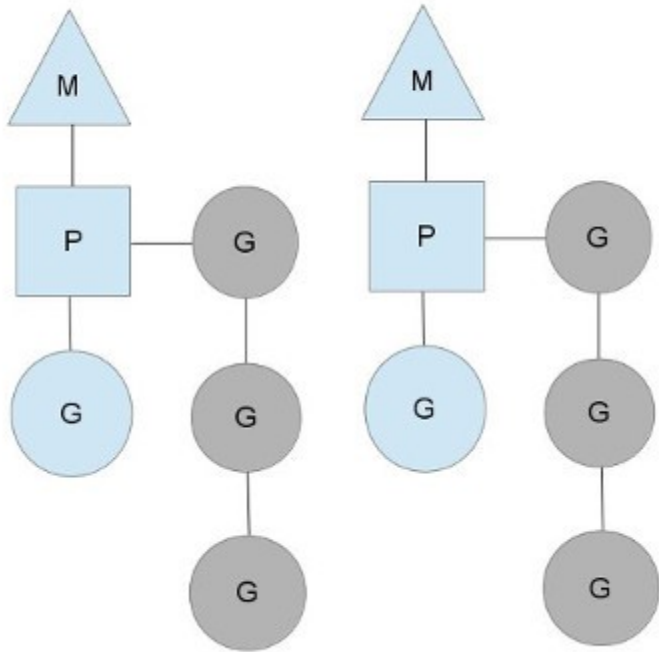
Go implementation details



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

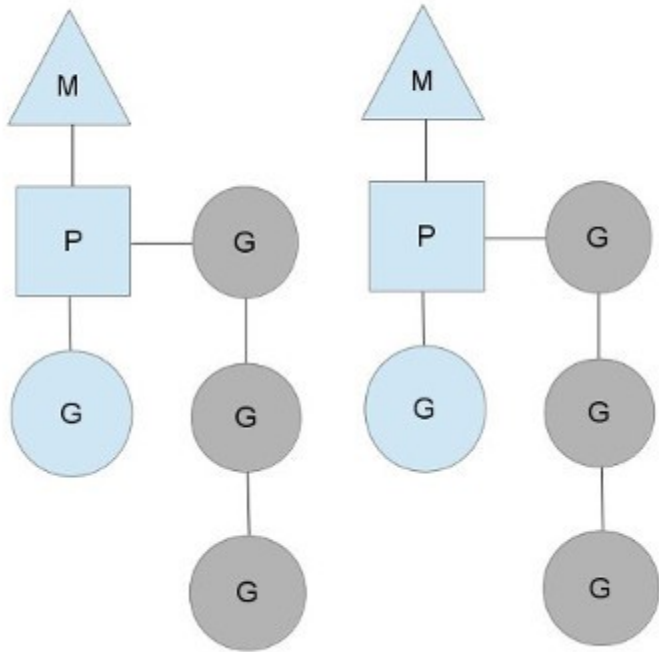
```
struct G
{
    byte*  stackguard; // stack guard information
    byte*  stackbase;  // base of stack
    byte*  stack0;     // current stack pointer
    byte*  entry;      // initial function
    void*  param;      // passed parameter on wakeup
    int16  status;     // status
    int32  goid;       // unique id
    M*     lockedm;    // used for locking M's and G's
    ...
};
```

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

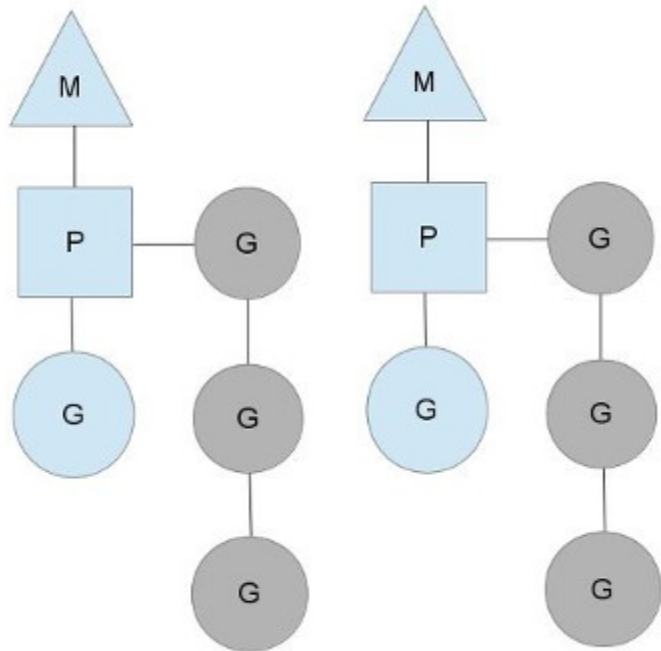
Go implementation details



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

```
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
    ...
};
```

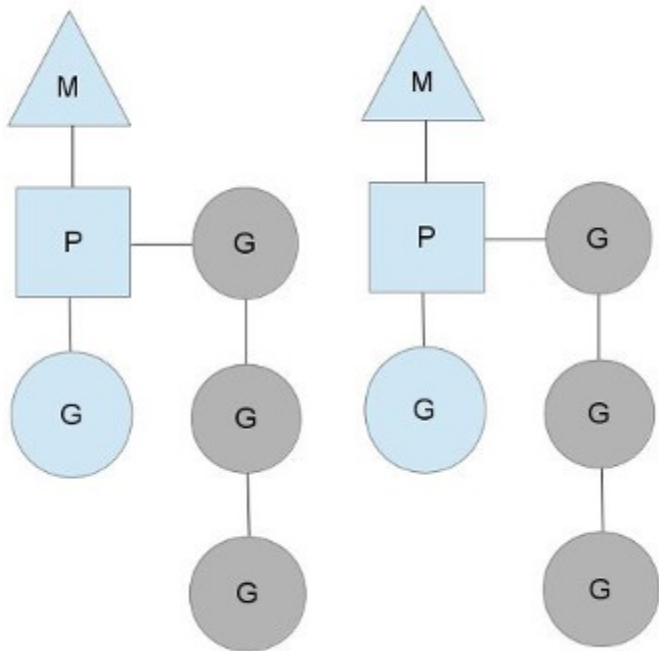
Go implementation details



- M = “machine” → OS thread

```
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

```
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  
    ...  
};
```


1000s of go routines?

```
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?

```
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()
}
```

- 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

1000s of go routines?

```
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 read")
            }
        }(i)
        wg.Done()
    }
    time.Sleep(1000 * time.Millisecond)
    close(ch)
    wg.Wait()
    stopTimes["testQ"] = time.Now()
}
```

- 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

```
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 10
testQ: 1.0016706s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 100
testQ: 1.0011655s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 1000
testQ: 1.0084796s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 10000
testQ: 1.0547925s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 100000
testQ: 1.3907835s
PS C:\Users\chris\go\src\cs378\lab3> .\lab3.exe -testq -qproducers 1000000
testQ: 4.2405814s
```

Channel implementation

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

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if !block {
143             return false
144         }
145         gopark(nil, nil, "chan send (nil chan)", traceEvGostop, 2)
146         throw("unreachable")
147     }
148
149     if debugChan {
150         print("chansend: chan=", c, "\n")
151     }
152
153     if raceenabled {
154         racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
155     }
156
157     // Fast path: check for failed non-blocking operation without acquiring the lock.
158     //
159     // After observing that the channel is not closed, we observe that the channel is
160     // not ready for sending. Each of these observations is a single word-sized read
161     // (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
162     // Because a closed channel cannot transition from 'ready for sending' to
163     // 'not ready for sending', even if the channel is closed between the two observations,
164     // they imply a moment between the two when the channel was both not yet closed
165     // and not ready for sending. We behave as if we observed the channel at that moment,
166     // and report that the send cannot proceed.
167     //
168     // It is okay if the reads are reordered here: if we observe that the channel is not
169     // ready for sending and then observe that it is not closed, that implies that the
170     // channel wasn't closed during the first observation.
171     if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
172         (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
173         return false
174     }
175
176     var t0 int64
177     if blockprofrate > 0 {
178         t0 = cputicks()
179     }
180
181     lock(&c.lock)
182
183     if c.closed != 0 {
184         unlock(&c.lock)
185         panic(plainError("send on closed channel"))
186     }
187
188     if sg := c.recvq.dequeue(); sg != nil {
189         // Found a waiting receiver. We pass the value we want to send
190         // directly to the receiver, bypassing the channel buffer (if any).
191         send(c, sg, ep, func() { unlock(&c.lock) }, 3)
192     }
```


Channel implementation

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

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if !block {
143             return false
144         }
145         gopark(nil, nil, "chan send (nil chan)", traceEvGostop, 2)
146         throw("unreachable")
147     }
148
149     if debugChan {
150         print("chansend: chan=", c, "\n")
151     }
152
153     if raceenabled {
154         racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
155     }
156
157     // Fast path: check for failed non-blocking operation without acquiring the lock.
158     //
159     // After observing that the channel is not closed, we observe that the channel is
160     // not ready for sending. Each of these observations is a single word-sized read
161     // (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
162     // Because a closed channel cannot transition from 'ready for sending' to
163     // 'not ready for sending', even if the channel is closed between the two observations,
164     // they imply a moment between the two when the channel was both not yet closed
165     // and not ready for sending. We behave as if we observed the channel at that moment,
166     // and report that the send cannot proceed.
167     //
168     // It is okay if the reads are reordered here: if we observe that the channel is not
169     // ready for sending and then observe that it is not closed, that implies that the
170     // channel wasn't closed during the first observation.
171     if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
172         (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
173         return false
174     }
175
176     var t0 int64
177     if blockprofrate > 0 {
178         t0 = cputicks()
179     }
180
181     lock(&c.lock)
182
183     if c.closed != 0 {
184         unlock(&c.lock)
185         panic(plainError("send on closed channel"))
186     }
187
188     if sg := c.recvq.dequeue(); sg != nil {
189         // Found a waiting receiver. We pass the value we want to send
190         // directly to the receiver, bypassing the channel buffer (if any).
191         send(c, sg, ep, func() { unlock(&c.lock) }, 3)
192     }
```

Channel implementation

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

```
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  
}
```

```
122 // entry point for c <- x from compiled code  
123 //go:nosplit  
124 func chansend1(c *hchan, elem unsafe.Pointer) {  
125     chansend(c, elem, true, getcallerpc())  
126 }  
127  
128 /*  
129 * generic single channel send/recv  
130 * If block is not nil,  
131 * then the protocol will not  
132 * sleep but return if it could  
133 * not complete.  
134 *  
135 * sleep can wake up with g.param == nil  
136 * when a channel involved in the sleep has  
137 * been closed. it is easiest to loop and re-run  
138 * the operation; we'll see that it's now closed.  
139 */  
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {  
141     if c == nil {  
142         if !block {  
143             return false  
144         }  
145         gopark(nil, nil, "chan send (nil chan)", traceEvGostop, 2)  
146         throw("unreachable")  
147     }  
148  
149     if debugChan {  
150         print("chansend: chan=", c, "\n")  
151     }  
152  
153     if raceenabled {  
154         racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))  
155     }  
156  
157     // Fast path: check for failed non-blocking operation without acquiring the lock.  
158     //  
159     // After observing that the channel is not closed, we observe that the channel is  
160     // not ready for sending. Each of these observations is a single word-sized read  
161     // (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).  
162     // Because a closed channel cannot transition from 'ready for sending' to  
163     // 'not ready for sending', even if the channel is closed between the two observations,  
164     // they imply a moment between the two when the channel was both not yet closed  
165     // and not ready for sending. We behave as if we observed the channel at that moment,  
166     // and report that the send cannot proceed.  
167     //  
168     // It is okay if the reads are reordered here: if we observe that the channel is not  
169     // ready for sending and then observe that it is not closed, that implies that the  
170     // channel wasn't closed during the first observation.  
171     if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||  
172         (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {  
173         return false  
174     }  
175  
176     var t0 int64  
177     if blockprofrate > 0 {  
178         t0 = cputicks()  
179     }  
180  
181     lock(&c.lock)  
182  
183     if c.closed != 0 {  
184         unlock(&c.lock)  
185         panic(plainError("send on closed channel"))  
186     }  
187  
188     if sg := c.recvq.dequeue(); sg != nil {  
189         // Found a waiting receiver. We pass the value we want to send  
190         // directly to the receiver, bypassing the channel buffer (if any).  
191         send(c, sg, ep, func() { unlock(&c.lock) }, 3)  
192     }  
193 }  
194
```

Channel implementation

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

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if !block {
143             return false
144         }
145         gopark(nil, nil, "chan send (nil chan)", traceEvGostop, 2)
146         throw("unreachable")
147     }
148
149     if debugChan {
150         print("chansend: chan=", c, "\n")
151     }
152
153     if raceenabled {
154         racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
155     }
156
157     // Fast path: check for failed non-blocking operation without acquiring the lock.
158     //
159     // After observing that the channel is not closed, we observe that the channel is
160     // not ready for sending. Each of these observations is a single word-sized read
161     // (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
162     // Because a closed channel cannot transition from 'ready for sending' to
163     // 'not ready for sending', even if the channel is closed between the two observations,
164     // they imply a moment between the two when the channel was both not yet closed
165     // and not ready for sending. We behave as if we observed the channel at that moment,
166     // and report that the send cannot proceed.
167     //
168     // It is okay if the reads are reordered here: if we observe that the channel is not
169     // ready for sending and then observe that it is not closed, that implies that the
170     // channel wasn't closed during the first observation.
171     if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
172         (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
173         return false
174     }
175
176     var t0 int64
177     if blockprofrate > 0 {
178         t0 = cputicks()
179     }
180
181     lock(&c.lock)
182
183     if c.closed != 0 {
184         unlock(&c.lock)
185         panic(plainError("send on closed channel"))
186     }
187
188     if sg := c.recvq.dequeue(); sg != nil {
189         // Found a waiting receiver. We pass the value we want to send
190         // directly to the receiver, bypassing the channel buffer (if any).
191         send(c, sg, ep, func() { unlock(&c.lock) }, 3)
192     }
```


Channel implementation

- You can just read it:
 - <https://golang.org/s>
- Some highlights

```
295 // Sends and receives on unbuffered or empty-buffered channels are the
296 // only operations where one running goroutine writes to the stack of
297 // another running goroutine. The GC assumes that stack writes only
298 // happen when the goroutine is running and are only done by that
299 // goroutine. Using a write barrier is sufficient to make up for
300 // violating that assumption, but the write barrier has to work.
301 // typedmemmove will call bulkBarrierPreWrite, but the target bytes
302 // are not in the heap, so that will not help. We arrange to call
303 // memmove and typeBitsBulkBarrier instead.
304
305 func sendDirect(t *_type, sg *sudog, src unsafe.Pointer) {
306     // src is on our stack, dst is a slot on another stack.
307
308     // Once we read sg.elem out of sg, it will no longer
309     // be updated if the destination's stack gets copied (shrunk).
310     // So make sure that no preemption points can happen between read & use.
311     dst := sg.elem
312     typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
313     memmove(dst, src, t.size)
314 }
```

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if lblock {
```

2)

)

quiring the lock.

at the channel is
: word-sized read
ig on kind of channel).
nding" to
n the two observations,
not yet closed
annel at that moment,

it the channel is not
it implies that the

: == nil) ||

```
184     unlock(&c.lock)
185     panic(plainError("send on closed channel"))
186 }
187
188 if sg := c.recvq.dequeue(); sg != nil {
189     // Found a waiting receiver. We pass the value we want to send
190     // directly to the receiver, bypassing the channel buffer (if any).
191     send(c, sg, ep, func() { unlock(&c.lock) }, 3)
```

Channel implementation

- You can just read it:
 - <https://golang.org/s>
- Some highlights

```
295 // Sends and receives on unbuffered or empty-buffered channels are the
296 // only operations where one running goroutine writes to the stack of
297 // another running goroutine. The GC assumes that stack writes only
298 // happen when the goroutine is running and are only done by that
299 // goroutine. Using a write barrier is sufficient to make up for
300 // violating that assumption, but the write barrier has to work.
301 // typedmemmove will call bulkBarrierPreWrite, but the target bytes
302 // are not in the heap, so that will not help. We arrange to call
303 // memmove and typeBitsBulkBarrier instead.
304
305 func sendDirect(t *_type, sg *sudog, src unsafe.Pointer) {
306     // src is on our stack, dst is a slot on another stack.
307
308     // Once we read sg.elem out of sg, it will no longer
309     // be updated if the destination's stack gets copied (shrunk).
310     // So make sure that no preemption points can happen between read & use.
311     dst := sg.elem
312     typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
313     memmove(dst, src, t.size)
314 }
```

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if lblock {
```

2)

)

quiring the lock.

at the channel is
: word-sized read
ig on kind of channel).
nding" to
n the two observations,
not yet closed
annel at that moment,

it the channel is not
it implies that the

: == nil) ||

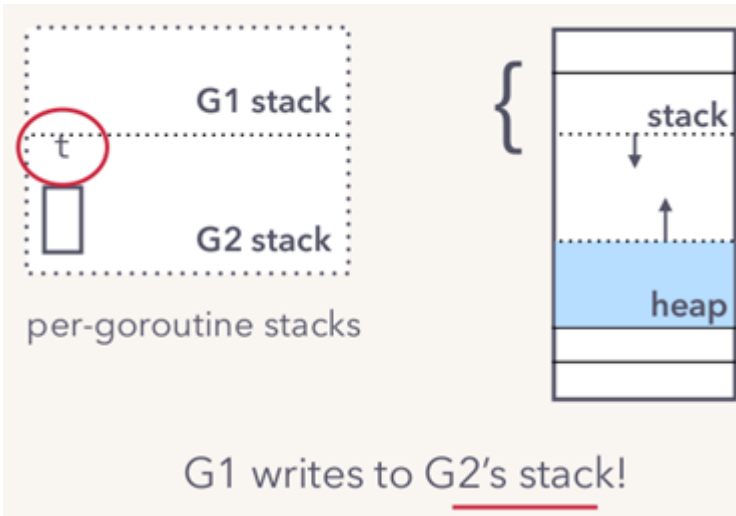
```
184     unlock(&c.lock)
185     panic(plainError("send on closed channel"))
186 }
187
188 if sg := c.recvq.dequeue(); sg != nil {
189     // Found a waiting receiver. We pass the value we want to send
190     // directly to the receiver, bypassing the channel buffer (if any).
191     send(c, sg, ep, func() { unlock(&c.lock) } 3)
```

Channel implementation

- You can just read it:
 - <https://golang.org/s>
- Some highlights

```
295 // Sends and receives on unbuffered or empty-buffered channels are the
296 // only operations where one running goroutine writes to the stack of
297 // another running goroutine. The GC assumes that stack writes only
298 // happen when the goroutine is running and are only done by that
299 // goroutine. Using a write barrier is sufficient to make up for
300 // violating that assumption, but the write barrier has to work.
301 // typedmemmove will call bulkBarrierPreWrite, but the target bytes
302 // are not in the heap, so that will not help. We arrange to call
303 // memmove and typeBitsBulkBarrier instead.
304
305 func sendDirect(t *_type, sg *sudog, src unsafe.Pointer) {
306     // src is on our stack, dst is a slot on another stack.
307
308     // Once we read sg.elem out of sg, it will no longer
309     // be updated if the destination's stack gets copied (shrunk).
310     // So make sure that no preemption points can happen between read & use.
311     dst := sg.elem
312     typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
313     memmove(dst, src, t.size)
314 }
```

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if lblock {
```



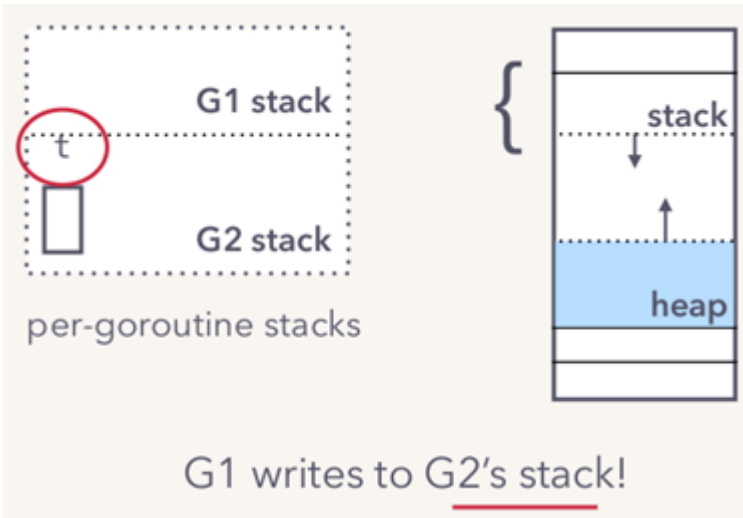
```
184     unlock(&c.lock)
185     panic(plainError("send on closed channel"))
186 }
187
188 if sg := c.recvq.dequeue(); sg != nil {
189     // Found a waiting receiver. We pass the value we want to send
190     // directly to the receiver, bypassing the channel buffer (if any).
191     send(c, sg, ep, func() { unlock(&c.lock) }, 3)
```

Channel implementation

- You can just read it:
 - <https://golang.org/s>
- Some highlights

```
295 // Sends and receives on unbuffered or empty-buffered channels are the
296 // only operations where one running goroutine writes to the stack of
297 // another running goroutine. The GC assumes that stack writes only
298 // happen when the goroutine is running and are only done by that
299 // goroutine. Using a write barrier is sufficient to make up for
300 // violating that assumption, but the write barrier has to work.
301 // typedmemmove will call bulkBarrierPreWrite, but the target bytes
302 // are not in the heap, so that will not help. We arrange to call
303 // memmove and typeBitsBulkBarrier instead.
304
305 func sendDirect(t *_type, sg *sudog, src unsafe.Pointer) {
306     // src is on our stack, dst is a slot on another stack.
307
308     // Once we read sg.elem out of sg, it will no longer
309     // be updated if the destination's stack gets copied (shrunk).
310     // So make sure that no preemption points can happen between read & use.
311     dst := sg.elem
312     typeBitsBulkBarrier(t, uintptr(dst), uintptr(src), t.size)
313     memmove(dst, src, t.size)
314 }
```

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if lblock {
```



Transputers did this in hardware in the 90s btw.

```
184     unlock(&c.lock)
185     panic(plainError("send on closed channel"))
186 }
187
188 if sg := c.recvq.dequeue(); sg != nil {
189     // Found a waiting receiver. We pass the value we want to send
190     // directly to the receiver, bypassing the channel buffer (if any).
191     send(c, sg, ep, func() { unlock(&c.lock) } 3)
```

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

```
122 // entry point for c <- x from compiled code
123 //go:nosplit
124 func chansend1(c *hchan, elem unsafe.Pointer) {
125     chansend(c, elem, true, getcallerpc())
126 }
127
128 /*
129  * generic single channel send/recv
130  * If block is not nil,
131  * then the protocol will not
132  * sleep but return if it could
133  * not complete.
134  *
135  * sleep can wake up with g.param == nil
136  * when a channel involved in the sleep has
137  * been closed. it is easiest to loop and re-run
138  * the operation; we'll see that it's now closed.
139  */
140 func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
141     if c == nil {
142         if !block {
143             return false
144         }
145         gopark(nil, nil, "chan send (nil chan)", traceEvGostop, 2)
146         throw("unreachable")
147     }
148
149     if debugChan {
150         print("chansend: chan=", c, "\n")
151     }
152
153     if raceenabled {
154         racereadpc(unsafe.Pointer(c), callerpc, funcPC(chansend))
155     }
156
157     // Fast path: check for failed non-blocking operation without acquiring the lock.
158     //
159     // After observing that the channel is not closed, we observe that the channel is
160     // not ready for sending. Each of these observations is a single word-sized read
161     // (first c.closed and second c.recvq.first or c.qcount depending on kind of channel).
162     // Because a closed channel cannot transition from 'ready for sending' to
163     // 'not ready for sending', even if the channel is closed between the two observations,
164     // they imply a moment between the two when the channel was both not yet closed
165     // and not ready for sending. We behave as if we observed the channel at that moment,
166     // and report that the send cannot proceed.
167     //
168     // It is okay if the reads are reordered here: if we observe that the channel is not
169     // ready for sending and then observe that it is not closed, that implies that the
170     // channel wasn't closed during the first observation.
171     if !block && c.closed == 0 && ((c.dataqsiz == 0 && c.recvq.first == nil) ||
172         (c.dataqsiz > 0 && c.qcount == c.dataqsiz)) {
173         return false
174     }
175
176     var t0 int64
177     if blockprofrate > 0 {
178         t0 = cputicks()
179     }
180
181     lock(&c.lock)
182
183     if c.closed != 0 {
184         unlock(&c.lock)
185         panic(plainError("send on closed channel"))
186     }
187
188     if sg := c.recvq.dequeue(); sg != nil {
189         // Found a waiting receiver. We pass the value we want to send
190         // directly to the receiver, bypassing the channel buffer (if any).
191         send(c, sg, ep, func() { unlock(&c.lock) }, 3)
192     }
```

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?

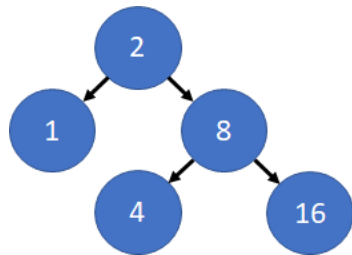
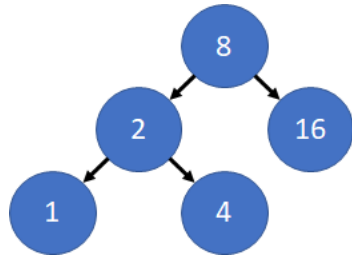
- Lacks compile-time generics
 - Results in code duplication
 - Metaprogramming cannot be statically checked
 - Standard library cannot offer generic algorithms
- Lack of language extensibility makes certain tasks more verbose
 - Lacks operator overloading (Java)
- Pauses and overhead of garbage collection
 - 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!*

Now. Let's discuss Lab 3

Binary Search Trees



- Each node has a value
- Left nodes have smaller values
- Right nodes have greater values

- Want to detect duplicate trees
 - Insertion order affects layout
- Linearize trees for comparison
 - Makes comparison expensive

Hashing BSTs

```
func initialHash() uint64 {  
    return 1  
}
```

```
func hash(uint64 hash, uint64 val) {  
    val2 = val + 2  
    prime = 4222234741  
    return (hash*val2+val2)%prime  
}
```

- Initialize hash
- Traverse tree in-order
- Incorporate values into hash

- Hash function doesn't have to be very complex
- Just make sure it handles zeros and similar numbers nicely

Processing pipeline

- Read in trees from file
 - Array / slice of BSTs
- Hash trees + insert hashes
 - Map from hash to tree indexes
- Compare trees
 - Equivalence matrix
 - num trees x num trees

Parallelizing the pipeline

Step 2

- Implement just hashing first
- Goroutines
 - 1 per tree
 - Dedicated inserter goroutine(s)
 - Communicate via channel
- Thread pool
 - hash-workers threads
 - Acquire lock(s) to insert
- Multiple data-workers optional

Step 3

- Goroutines
 - 1 per comparison
- Thread pool
 - comp-workers threads
 - Send work via channel
 - (Optional) custom implementation
 - Queue, mutex, and conditions
- Store results directly in matrix

Go: command-line flags

```
import "flag"
func main() {
    intPtr = flag.Int("num", 0, "number argument")
    flag.Parse()
    num := *intPtr
}
```

```
./my_program -num=1
```

Go: file parsing

```
import ("io/ioutil" "strconv" "strings")
func main() {
    fileData, err := ioutil.ReadFile(fileName)
    fileData = fileData[:len(fileData)-1] // remove EOF
    fileLines := strings.Split(string(fileData), "\n")
    for _, line := range fileLines {
        // parse line with strings.Split and strconv.Atoi()
    }
}
```

Go: timing

```
import "time"
func main() {
    start := time.Now()
    // do some work
    timeTakenStr:= time.Since(start)
    fmt.Printf("Doing work took %s\n", timeTakenStr)
}
```

Go: functions and return values

```
func notMain() (int, bool) { // multiple return values
    return (3, false)
}
```

```
func main() {
    i, b := notMain()
    j, _ := notMain() // throw away value
}
```


Go: synchronization

```
import "sync" // contains WaitGroups
func main() {
    var *mutex = &sync.Mutex{} // pointer to mutex
    var *cond = &sync.NewCond(mutex) // mutex condition
    mutex.Lock()
    cond.Wait() // releases lock on mutex
    cond.Signal() // wakes threads waiting on cond
    mutex.Unlock()
}
```

Go: slices

```
func main() {  
    mySlice := make([]int, 2)  
    mySlice[1] = 5 // can use like an array  
    mySlice = append(mySlice, 10) // can use like a list  
    l := len(mySlice)  
    subSlice := mySlice[0:1] // can slice like in Python  
    fromStartToTwo := mySlice[:2]  
    fromOneToEnd := mySlice[1:]  
}
```

Go: maps

```
func main() {  
    mapIntBool := make(map[int] bool) // map from ints to bools  
    mapIntBool[5] = true  
    for key, value := range mapIntBool {  
        // use key or value  
    }  
}  
  
// map value can be a slice
```

Go: misc

```
type myStruct struct {  
    mySlice []int  
    myChan chan int  
    mySliceOfSlice [][]bool  
    myPtr *myStruct  
}  
var ms myStruct // declare variable without initialization  
// use dot operator for structs, pointers, and pointers to structs  
ms.myPtr.mySlice[2]
```