# Concurrent Programming Languages

Channel-based Concurrency Module
**Lecture 2:** Coordinating Goroutines

5 November 2020

**MIEI - Integrated Masters in Comp. Science and Informatics**
**Specialization Block**

**Bernardo Toninho**
(with António Ravara and Carla Ferreira)

NOVALINCS

# Last Lecture

- A quick tour of basic Go programming
- Concurrency in Go:
  - Goroutines
  - Channels (Synchronous, asynchronous)
  - Basic channel usage

# Today

- **Big picture:** how do we synchronize and coordinate many goroutines?
- Some channel-based primitives we didn't explore last time (close, select).
- First-class channels in Go.
- Various useful idioms along the way.
- Useful Go tooling.

# Goroutines

- Goroutines are a powerful and easy way to add concurrency to programs.

- But with great power…

```
func main() {
   var wg sync.WaitGroup
   wg.Add(5)
   for i := 0; i < 5; i++ {
      go func() {
         fmt.Println(i)
         wg.Done()
      }()
   }
   wg.Wait()
}
```

# Goroutines

- Goroutines are a powerful and easy way to add concurrency to programs.
- But with great power…

```
func main() {
   var wg sync.WaitGroup
   wg.Add(5)
   for i := 0; i < 5; i++ {
      go func() {
         fmt.Println(i)
         wg.Done()
      }()
   }
   wg.Wait()
}
```

Can anyone spot a **bug**?

©Renee French

# Goroutines

- Goroutines are a powerful and easy way to add concurrency to programs.
- But with great power…

```
func main() {
   var wg sync.WaitGroup
   wg.Add(5)
   for i := 0; i < 5; i++ {
      go func() {
         fmt.Println(i) //race on ‘i’!!
         wg.Done()
      }()
   }
   wg.Wait()
}
```

Can anyone spot a **bug**?

©Renee French

# Goroutines

- Goroutines are a powerful and easy way to add concurrency to programs.

- But with great power…

```
func main() {
   var wg sync.WaitGroup
   wg.Add(5)
   for i := 0; i < 5; i++ {
      go func(j int) {
         fmt.Println(j) // i is now copied, no race!
         wg.Done()
      }(i)
   }
   wg.Wait()
}
```

# Goroutines + Channels

- Goroutines are a powerful concurrency building block.
- We still have to be very careful when sharing memory…
- Channel-based concurrency helps a lot!
- Structure applications with a server/client mindset.

# Channels

- Channels are first-class typed values (can be passed as fun args, returned by functions, sent on channels)
- Channels provide communication and synchronization capabilities.
- Prime mechanism for coordinating goroutines
- Sends/receives on channels are **not** data races! (managed by go runtime)

# Channels

- Channels can be synchronous:

```
ch := make(chan int)
```

- or asynchronous (bounded buffers):

```
ch := make(chan int, N) //N known statically.
```

- can also be **ranged** over, and **closed** (New).

# Ranging over Channels

For-each idiom for channels:

```
func drain(ch chan int) {
  for v := range ch {
   fmt.Printf(“%d “,v)
  }
  fmt.Printf(“\n”)
}
```

Reads from channel until sender **closes** it.

# Closing Channels

Closing a channel signals to readers that no more data will be pushed onto channel:

```
func emit(ch chan int, n int) {
  for i := 0; i < n; i++ {
      ch <- i
  }
  close(ch)
}
```

Readers can test if a channel is closed (via **`range`**) or explicitly.

# Closing Channels

Closing a channel signals to readers that no more data will be pushed onto channel:

```
func main() {
  ch := make(chan int)
  go emit(ch,10)
  drain(ch)
}
```

```
func emit(ch chan int, n int) {
  for i := 0; i < n; i++ {
      ch <- i
  }
  close(ch)
}
```

```
0 1 2 3 4 5 6 7 8 9
```

# Closing Channels

Semantics of **`close`**:

- Reading from a closed channel **always** succeeds.
- Sending on a closed channel produces a **panic** at runtime.

```
func main() {
  c := make(chan int)
  close(c)
  fmt.Println(<-c) //prints 0
  c <- 23 //panic: send on closed channel
}
```

# Closing Channels

Semantics of **`close`**:

- Channels with no leftover data emit “default” value.
- Close is **not** necessary for garbage collection.

```
func main() {
  c := make(chan int,1)
  c <- 23
  close(c)
  n, ok := <- c
  n, ok = <- c
  fmt.Println(n,ok) //prints 23 true
  fmt.Println(n,ok) //prints 0 false
}
```

# Closing Channels

- Channels with no leftover data emit “default” value.
- Close is **not** necessary for garbage collection.
- Close is often used as a (one shot) broadcast from 1 producer to N consumers.
- Consumers test for closure explicitly or via `range` (syntactic sugar).

# Selective Communication

Go offers a switch-like construct for communication:

```
c1,c2 := make(chan int), make(chan int)
go complicatedFunction(c1,c2)
select {
  case v := <- c1:
    fmt.Printf(“received %d from c1\n”,v)
  case v := <- c2:
    fmt.Printf(“received %d from c2\n”,v)
 }
```

Each case must be a send or receive. If no comms. are available, block. If multiple are, choose randomly.

# Selective Communication

```
select {
  case v := <- ch1:
    fmt.Printf(“received”,v)
  case v := <- ch2:
    fmt.Printf(“received”,v)
  default:
    fmt.Printf(“no channel ready for input”)
 }
```

If no other **`case`** can proceed, **`default`** clause executes.
Allows us to encode **non-blocking** receive/send.

# Selective Communication

```
for {
 select {
   case v := <- ch:
     fmt.Printf(“received”,v)
   case <- time.After(10*time.Second):
     fmt.Printf(“this message took too long.”)
  }
 }
```

`After` function sends on the **returned channel** after a specified duration. Allows us to encode **timeouts**.

# Selective Communication

```
for {
 select {
   case v := <- ch:
     fmt.Printf(“received”,v)
   case <- time.After(3*time.Second):
     fmt.Printf(“this message took too long.”)
  }
 }
```

Timeouts are “per message”. Can you see how to add a **global** timeout to the conversation?

# Selective Communication

```
timeout := time.After(10*time.Second)
for {
 select {
   case v := <- ch:
     fmt.Printf(“received”,v)
   case <- timeout:
     fmt.Printf(“conversation took too long!”)
     return
  }
 }
```

**Global** (longer) timeout to the entire conversation.

# Selective Communication

```
func server(reqChannel <-chan int, quit <-chan bool) {
 for {
  select {
    case req := <- reqChannel:
      go process(req)
    case <- quit:
      graceful_cleanup()
      return
    }
   }
  }
```

By “multiplexing” with select, we can encode a server **termination** channel.

# Selective Communication

```
func server(reqChannel <-chan int, quit <-chan bool) {
 for {
  select {
    case req := <- reqChannel:
      go process(req)
    case <- quit:
      graceful_cleanup()
      return
    }
   }
  }
```

How to ensure client thread doesn’t terminate before server thread cleans up?

# Selective Communication

```
func server(reqChannel <-chan int, quit chan bool) {
 for {
  select {
    case req := <- reqChannel:
      go process(req)
    case <- quit:
      graceful_cleanup()
      quit <- true
      return
    }
   }
  }
```

Just send an acknowledgement back to the client.

# Selective Communication

- Select is a powerful tool in our concurrency arsenal.
- Can encode fan-ins, non-blocking communication, timeouts (local and global), etc…
- Concurrency is all about composition and structure!
- Safe synchronization patterns — No mutexes, no locks!

# First-class Channels

- We have already used a function returning a channel:

```
func After(d Duration) <-chan Time
```

- Useful idiom to hide complex logic:

```
func emit(ch chan int, n int) {
  for i := 0; i < n; i++ {
      ch <- i
  }
  close(ch)
}
```

# First-class Channels

- We have already used a function returning a channel:

```
func After(d Duration) <-chan Time
```

- Useful idiom to hide complex logic:

```
func emit(ch chan int, n int) {
   for i := 0; i < n; i++ {
       ch <- i
  }
  close(ch)
}
```

```
func emit(n int) (chan int) {
  ch := make(chan int)
  go func() {
   for i := 0; i < n; i++ {
         ch <- i
    }
   close(ch)
  }
  return ch
}
```

# First-class Channels

- Go's type system can enforce directionality constraints on channels:

```
func emit(n int) (<- chan int)
func drain() (chan<- int)
func main() {
  c := emit()
  d := drain()
  …
  c <- 2
  …
}
```

```
invalid operation: c <- 2 (send to receive-only type <-chan int)
```

# First-class Channels

- Go's type system can enforce directionality constraints on channels:

```
func emit(n int) (<- chan int)
func drain() (chan<- int)
func main() {
  c := emit()
  d := drain()
  …
  fmt.Println(<-d)
  …
}
```

```
invalid operation: <-d (receive from send-only type chan<- int)
```

# First-class Channels

- Channels can also be sent and received along other channels.

- Useful for multiplexing requests.

```
type Request struct {
    payload int
    replychan chan int
}
func handle(req *Request) {
    req.replychan <- some_complex_op(req.payload)
}
func server(service <-chan *Request) {
  for { req := <-service
        go handle(req) }
}
```

# First-class Channels

- Channels can also be sent and received along other channels.

- Useful for multiplexing requests.

```
type Request struct {
    payload int
    replychan chan int
}
func handle(req *Request) {
    req.replychan <- some_complex_op(req.payload)
}
func server(service <-chan *Request) {
  for { req := <-service
        go handle(req) }
}
```

# First-class Channels

- Channels can also be sent and received along other channels.

- Useful for multiplexing requests.

```
type Request struct {
    payload int
    replychan chan int
}
func handle(req *Request) {
    req.replychan <- some_complex_op(req.payload)
}
func server(service <-chan *Request) {
  for { req := <-service
        go handle(req) }
}
```

# First-class Channels

- Can further combine with “quit” channel:

```
func server(service <-chan *Request, quit <-chan bool) {
  for {
    select {
      case req := <- service:
        go handle(req)
      case: <- quit:
        return
      }
    }
}
```

# Thinking with Channels

- Goroutines are great.
- Channels are a great way of sharing data and synchronizing goroutines.
- Allow for sophisticated concurrent orchestrations without locks and mutexes (its easy to solve most of the “classical” concurrency problems).
- …but they don’t make all concurrency problems magically disappear!

# Thinking with Channels

## Introduction to Programming Redux, MSc Edition:

```
type BankAccount struct {
   balance int
}

func newAccount() *BankAccount {
   return &BankAccount{1000}
}
func (b *BankAccount) withdraw(amount int) {
   b.balance -= amount
}
func (b *BankAccount) deposit(amount int) {
   b.balance += amount
}
func (b *BankAccount) getBalance() int {
   return b.balance
}
```

# Thinking with Channels

## Introduction to Programming Redux, MSc Edition

```
func main() {
  bernardoAcct := newAccount()
  joãoAcct := newAccount()
  for i := 0; i < 1000; i++ {
    go joãoAcct.withdraw(1)
    go bernardoAcct.deposit(1)

    go bernardoAcct.withdraw(1)
    go joãoAcct.deposit(1)
  }

  time.Sleep(2*time.Second)
  fmt.Printf("Bernardo's balance is %d\n",bernardoAcct.getBalance())
  fmt.Printf("João's balance is %d\n",joãoAcct.getBalance())
}
```

Can anyone spot a **bug**?

©Renee French

# A detour: Go Tooling

- Go has a dynamic data race detector built-in.
- Can be activated with the `-race` flag in all settings (testing, running, building, etc).
- Will make your program slower (2-20x) and consume more memory (5-10x).
- Some non-determinism will be “squashed” due to timing and synchronization — don’t use it for functionality testing.
- **Dynamic** analysis — can't find races in code paths that are not executed.

# A detour: Go Tooling

## Data race detector in action:

```
$ go run -race BankAccount.go

==================
WARNING: DATA RACE
Read at 0x00c000136020 by goroutine 10:
  main.(*BankAccount).deposit()
      .../BankAccount.go:52 +0x3a

Previous write at 0x00c000136020 by goroutine 7:
  main.(*BankAccount).withdraw()
      .../BankAccount.go:49 +0x50

Goroutine 10 (running) created at:
  main.main()
      .../BankAccount.go:67 +0x15a

Goroutine 7 (finished) created at:
  main.main()
      .../BankAccount.go:63 +0xd6
==================
==================
WARNING: DATA RACE
Read at 0x00c000136028 by goroutine 9:
  main.(*BankAccount).withdraw()
      .../BankAccount.go:50 +0x3a

Previous write at 0x00c000136028 by
  goroutine 8:
  main.(*BankAccount).deposit()
      .../BankAccount.go:53 +0x55

Goroutine 9 (running) created at:
  main.main()
      .../BankAccount.go:84 +0x12f

Goroutine 8 (finished) created at:
  main.main()
      .../BankAccount.go:82 +0x104
==================
Bernardo's balance is 1000
João's balance is 1000
Found 2 data race(s)
```

# A detour: Go Tooling

## Data race detector in action:

```
$ go run -race BankAccount.go

==================
WARNING: DATA RACE
Read at 0x00c000136020 by goroutine 10:
  main.(*BankAccount).deposit()
      .../BankAccount.go:52 +0x3a

Previous write at 0x00c000136020 by goroutine 7:
  main.(*BankAccount).withdraw()
      .../BankAccount.go:49 +0x50

Goroutine 10 (running) created at:
  main.main()
      .../BankAccount.go:67 +0x15a

Goroutine 7 (finished) created at:
  main.main()
      .../BankAccount.go:63 +0xd6
==================
==================
WARNING: DATA RACE
Read at 0x00c000136028 by goroutine 9:
  main.(*BankAccount).withdraw()
      .../BankAccount.go:50 +0x3a

Previous write at 0x00c000136028 by
  goroutine 8:
  main.(*BankAccount).deposit()
      .../BankAccount.go:53 +0x55

Goroutine 9 (running) created at:
  main.main()
      .../BankAccount.go:84 +0x12f

Goroutine 8 (finished) created at:
  main.main()
      .../BankAccount.go:82 +0x104
==================
Bernardo's balance is 1000
João's balance is 1000
Found 2 data race(s)
```

# A detour: Go Tooling

## Data race detector in action:

```
$ go run -race BankAccount.go

==================
WARNING: DATA RACE
Read at 0x00c000136020 by goroutine 10:
  main.(*BankAccount).deposit()
      .../BankAccount.go:52 +0x3a

Previous write at 0x00c000136020 by goroutine 7:
  main.(*BankAccount).withdraw()
      .../BankAccount.go:49 +0x50

Goroutine 10 (running) created at:
  main.main()
      .../BankAccount.go:67 +0x15a

Goroutine 7 (finished) created at:
  main.main()
      .../BankAccount.go:63 +0xd6
==================
```

```
==================
WARNING: DATA RACE
Read at 0x00c000136028 by goroutine 9:
  main.(*BankAccount).withdraw()
      .../BankAccount.go:50 +0x3a

Previous write at 0x00c000136028 by
  goroutine 8:
  main.(*BankAccount).deposit()
      .../BankAccount.go:53 +0x55

Goroutine 9 (running) created at:
  main.main()
      .../BankAccount.go:84 +0x12f

Goroutine 8 (finished) created at:
  main.main()
      .../BankAccount.go:82 +0x104
==================
Bernardo's balance is 1000
João's balance is 1000
Found 2 data race(s)
```

# A detour: Go Tooling

- A more principled / well-engineered approach: Testing!
- Go provides a (simple) unit-testing infrastructure.
- Test files must have `_test.go` in their name.
- Tests are defined as functions with signature:

```
func TestXxx(*testing.T)
```

- Use the Error, Fail or related methods of `testing.T` to signal failure.
- Tests are run with `go test`.

# A detour: Go Tooling

Testing in action:

```
func TestAbs(t *testing.T) {
    got := int(math.Abs(-1))
    if got != 1 {
        t.Errorf("Abs(-1) = %d; want 1”, got)
    }
}
```

```
func TestAbort(t *testing.T) {
   t.Error(“This test will always fail!”)
}
```

```
$ go test -v sample_test.go
=== RUN   TestAbs
--- PASS: TestAbs (0.00s)
=== RUN   TestAbort
    sample_test.go:18: This test will always fail!
--- FAIL: TestAbort (0.00s)
```

# A detour: Go Tooling

Testing in action:

```
func TestAbs(t *testing.T) {
    got := int(math.Abs(-1))
    if got != 1 {
        t.Errorf("Abs(-1) = %d; want 1”, got)
    }
}
```

```
func TestAbort(t *testing.T) {
   t.Error(“This test will always fail!”)
}
```

```
$ go test -v sample_test.go
=== RUN   TestAbs
--- PASS: TestAbs (0.00s)
=== RUN   TestAbort
    sample_test.go:18: This test will always fail!
--- FAIL: TestAbort (0.00s)
```

# A detour: Go Tooling

Testing in action:

```
func TestAbs(t *testing.T) {
    got := int(math.Abs(-1))
    if got != 1 {
        t.Errorf("Abs(-1) = %d; want 1”, got)
    }
}

func TestAbort(t *testing.T) {
   t.Error(“This test will always fail!”)
}
```

```
$ go test -v sample_test.go
=== RUN   TestAbs
--- PASS: TestAbs (0.00s)
=== RUN   TestAbort
    sample_test.go:18: This test will always fail!
--- FAIL: TestAbort (0.00s)
```

# A detour: Go Tooling

Back to our racy bank account:

```
func TestConcurrentIdempotence(t *testing.T) {
  bernardoAcct := newAccount()
  joãoAcct := newAccount()
  for i := 0; i < 1000; i++ {
    go joãoAcct.withdraw(1)
    go bernardoAcct.deposit(1)
    go bernardoAcct.withdraw(1)
    go joãoAcct.deposit(1)
  }
  time.Sleep(2*time.Second)
  if bernardoAcct.getBalance() != joãoAcct.getBalance() {
      t.Errorf("Non-equal balances. Balance1 is %d, Balance2 is %d",
           t.Name(), bernardoAcct.getBalance(),joãoAcct.getBalance())
   }
}

$ go test -v BankAccount_test.go
=== RUN   TestConcurrentIdempotence
    BankAccount_test.go:71: Test TestConcurrentIdempotence failed.  Non-equal
balances. Balance1 is 1008, Balance2 is 1004
--- FAIL: TestConcurrentIdempotence (2.01s)
```

# A detour: Go Tooling

Back to our racy bank account:

```
func TestConcurrentIdempotence(t *testing.T) {
  bernardoAcct := newAccount()
  joãoAcct := newAccount()
  for i := 0; i < 1000; i++ {
    go joãoAcct.withdraw(1)
    go bernardoAcct.deposit(1)
    go bernardoAcct.withdraw(1)
    go joãoAcct.deposit(1)
  }
  time.Sleep(2*time.Second)
  if bernardoAcct.getBalance() != joãoAcct.getBalance() {
      t.Errorf("Non-equal balances. Balance1 is %d, Balance2 is %d",
           t.Name(), bernardoAcct.getBalance(),joãoAcct.getBalance())
   }
}

$ go test -v BankAccount_test.go
=== RUN   TestConcurrentIdempotence
    BankAccount_test.go:71: Test TestConcurrentIdempotence failed.  Non-equal
balances. Balance1 is 1008, Balance2 is 1004
--- FAIL: TestConcurrentIdempotence (2.01s)
```

# A detour: Go Tooling

Back to our racy bank account:

```
$ go test -v -race BankAccount_test.go
==================
WARNING: DATA RACE
Read at 0x00c000120098 by goroutine 11:
  command-line-arguments.(*BankAccount).deposit()
      /Users/btoninho/BankAccount_test.go:53 +0x3a

Previous write at 0x00c000120098 by goroutine 8:
  command-line-arguments.(*BankAccount).withdraw()
      /Users/btoninho/BankAccount_test.go:50 +0x50

Goroutine 11 (running) created at:
  command-line-arguments.TestConcurrentIdempotence()
      /Users/btoninho/BankAccount_test.go:67 +0x15a
  testing.tRunner()
      /usr/local/go/src/testing/testing.go:1127 +0x202

Goroutine 8 (finished) created at:
  command-line-arguments.TestConcurrentIdempotence()
      /Users/btoninho/BankAccount_test.go:63 +0xd6
  testing.tRunner()
      /usr/local/go/src/testing/testing.go:1127 +0x202
==================
    testing.go:1042: race detected during execution of test
--- FAIL: TestConcurrentIdempotence (2.28s)
```

# A detour: Go Tooling

- Go provides reasonable tooling “out of the box”.
- Runtime race detection, testing, benchmarking, debugging, etc.
- Testing includes coverage analysis and a bunch more features.
- There are user libraries that expand on the core testing facilities.
- Go read more about the various customizations and features!

# Thinking with Channels

Back to IntroProg, MSc Edition — Lets fix the race:

```
type BankAccount struct {
   balance int
   depChan chan int
   withdrawChan chan int
   balanceReq chan bool
   balanceResult chan int
}
func newAccount() *BankAccount {
   acct := &BankAccount{1000,make(chan int),…}
   go acct.handler()
   return acct
}
func (b *BankAccount) withdraw(amount int) {
   b.withdrawChan <- amount
}
...
func (b *BankAccount) getBalance() int {
   b.balanceReq <- true
   return <- b.balanceResult
}
```

# Thinking with Channels

Back to IntroProg, MSc Edition — Lets fix the race:

```
type BankAccount struct {
   balance int
   depChan chan int
   withdrawChan chan int
   balanceReq chan bool
   balanceResult chan int
}
func newAccount() *BankAccount {
   acct := &BankAccount{1000,make(chan int),…}
   go acct.handler()
   return acct
}
func (b *BankAccount) withdraw(amount int) {
   b.withdrawChan <- amount
}
...
func (b *BankAccount) getBalance() int {
   b.balanceReq <- true
   return <- b.balanceResult
}
```

# Thinking with Channels

Back to IntroProg, MSc Edition — Lets fix the race:

```
type BankAccount struct {
   balance int
   depChan chan int
   withdrawChan chan int
   balanceReq chan bool
   balanceResult chan int
}
func newAccount() *BankAccount {
   acct := &BankAccount{1000,make(chan int),…}
   go acct.handler()
   return acct
}
func (b *BankAccount) withdraw(amount int) {
   b.withdrawChan <- amount
}
...
func (b *BankAccount) getBalance() int {
   b.balanceReq <- true
   return <- b.balanceResult
}
```

# Thinking with Channels

Back to IntroProg, MSc Edition — Lets fix the race:

```
type BankAccount struct {
   balance int
   depChan chan int
   withdrawChan chan int
   balanceReq chan bool
   balanceResult chan int
}
func newAccount() *BankAccount {
   acct := &BankAccount{1000,make(chan int),…}
   go acct.handler()
   return acct
}
func (b *BankAccount) withdraw(amount int) {
   b.withdrawChan <- amount
}
...
func (b *BankAccount) getBalance() int {
   b.balanceReq <- true
   return <- b.balanceResult
}
```

# Thinking with Channels

Back to IntroProg, MSc Edition — Lets fix the race:

```
func (b *BankAccount) handler() {
   for {
     select {
       case amount := <- b.depChan:
          b.balance += amount
       case amount := <- b.withdrawChan:
          b.balance -= amount
       case <- b.balanceReq:
          b.balanceResult <- b.balance
     }
   }
}
func main() {
  ...
  time.Sleep(2*time.Second)
  fmt.Printf("Bernardo's balance is %d\n",bernardoAcct.getBalance())
  fmt.Printf("João's balance is %d\n",joãoAcct.getBalance())
}
```

# Thinking with Channels

Back to IntroProg, MSc Edition — Lets fix the race:

```
func (b *BankAccount) handler() {
   for {
     select {
       case amount := <- b.depChan:
          b.balance += amount
       case amount := <- b.withdrawChan:
          b.balance -= amount
       case <- b.balanceReq:
          b.balanceResult <- b.balance
     }
   }
}
```

©Renee French

What's the print out? Are you sure? :)

```
func main() {
  ...
  time.Sleep(2*time.Second)
  fmt.Printf("Bernardo's balance is %d\n",bernardoAcct.getBalance())
  fmt.Printf("João's balance is %d\n",joãoAcct.getBalance())
}
```

# Thinking with Channels

Back to IntroProg, MSc Edition — Lets fix the race:

```
$ go test -v -race -count 5 BankAccount_test.go
=== RUN   TestConcurrentIdempotence
--- PASS: TestConcurrentIdempotence (2.30s)
=== RUN   TestConcurrentIdempotence
--- PASS: TestConcurrentIdempotence (2.28s)
=== RUN   TestConcurrentIdempotence
--- PASS: TestConcurrentIdempotence (2.27s)
=== RUN   TestConcurrentIdempotence
--- PASS: TestConcurrentIdempotence (2.28s)
=== RUN   TestConcurrentIdempotence
--- PASS: TestConcurrentIdempotence (2.28s)
PASS
ok   command-line-arguments 11.581s
```

# Thinking with Channels

- Channels can be used as way to protect concurrent accesses to shared data structures.
- Implementation details can be hidden via methods/functions.
- Internal protocols can be arbitrarily complex.

# Thinking with Channels

```
var comm = make(chan int, 3)
var finishProd = make(chan bool)
var finishCons = make(chan bool)
func produce() {
  for i := 0; i < 1000; i++ {
     comm <- i
  }
  finishProd <- true
}

func consume() {
  for {
    select {
       case msg := <-comm:
        fmt.Println(msg)
       case <-finishProd:
        finishCons <- true
        return
     }
  }
}
```

```
func main() {
   go produce()
   go consume()
   <- finishCons
   fmt.Println(“Done!”)
}
```

# Thinking with Channels

```
var comm = make(chan int, 3)
var finishProd = make(chan bool)
var finishCons = make(chan bool)
func produce() {
  for i := 0; i < 1000; i++ {
     comm <- i
  }
  finishProd <- true
}

func consume() {
  for {
    select {
       case msg := <-comm:
        fmt.Println(msg)
       case <-finishProd:
        finishCons <- true
        return
     }
  }
}
```

```
func main() {
   go produce()
   go consume()
   <- finishCons
   fmt.Println("Done!")
}
```

# Thinking with Channels

```
var comm = make(chan int, 3)
var finishProd = make(chan bool)
var finishCons = make(chan bool)
func produce() {
  for i := 0; i < 1000; i++ {
     comm <- i
  }
  finishProd <- true
}
```

```
func consume() {
  for {
    select {
       case msg := <-comm:
        fmt.Println(msg)
       case <-finishProd:
        finishCons <- true
        return
     }
  }
}
```

```
func main() {
   go produce()
   go consume()
   <- finishCons
   fmt.Println(“Done!”)
}
```

# Thinking with Channels

```
var comm = make(chan int, 3)
var finishProd = make(chan bool)
var finishCons = make(chan bool)
func produce() {
  for i := 0; i < 1000; i++ {
     comm <- i
  }
  finishProd <- true
}

func consume() {
  for {
    select {
       case msg := <-comm:
        fmt.Println(msg)
       case <-finishProd:
        finishCons <- true
        return
     }
  }
}
```

```
func main() {
   go produce()
   go consume()
   <- finishCons
   fmt.Println("Done!")
}
```

# Thinking with Channels

```
var comm = make(chan int, 3)
var finishProd = make(chan bool)
var finishCons = make(chan bool)
func produce() {
  for i := 0; i < 1000; i++ {
     comm <- i
  }
  finishProd <- true
}

func consume() {
  for {
    select {
       case msg := <-comm:
        fmt.Println(msg)
       case <-finishProd:
        finishCons <- true
        return
     }
  }
}
```

```
func main() {
   go produce()
   go consume()
   <- finishCons
   fmt.Println("Done!")
}
```

Can anyone spot a **bug**?

©Renee French

# Thinking with Channels

- We have to be disciplined about channel communication.

```
func produce(seed int) chan int {
  ch := make(chan int)
  go func(seed int) {
     defer wg.Done()
     n := seed
     for {
       select {
          case ch <- n:
           n++
          case <-ch:
           return
       }
     }
  }(seed)
  return ch
}
```

```
func consume(ch chan int, n int) {
    defer wg.Done()
    for i := 0 ; i < n ; i++ {
        data := <-ch
        time.Sleep(…)
        //Faking a slow op
    }
}

func main() {
  wg.Add(2)
  nums := produce(23)
  go consume(nums)
  … // Complex code
  … // so complex I forgot…
  nums <- 0
  //uhoh…
  wg.Wait()
}
```

# Thinking with Channels

- We have to be disciplined about channel communication.

```
func produce(seed int) chan int {
  ch := make(chan int)
  go func(seed int) {
     defer wg.Done()
     n := seed
     for {
       select {
          case ch <- n:
           n++
          case <-ch:
           return
       }
     }
  }(seed)
  return ch
}
```

```
func consume(ch chan int, n int) {
   defer wg.Done()
   for i := 0 ; i < n ; i++ {
       data := <-ch
       time.Sleep(…)
       //Faking a slow op
   }
}
```

```
func main() {
  wg.Add(2)
  nums := produce(23)
  go consume(nums)
  … // Complex code
  … // so complex I forgot…
  nums <- 0
  //uhoh…
  wg.Wait()
}
```

# Thinking with Channels

- We have to be disciplined about channel communication.

```
func produce(seed int) chan int {
  ch := make(chan int)
  go func(seed int) {
     defer wg.Done()
     n := seed
     for {
       select {
          case ch <- n:
           n++
          case <-ch:
           return
       }
     }
  }(seed)
  return ch
}
```

```
func consume(ch chan int, n int) {
    defer wg.Done()
    for i := 0 ; i < n ; i++ {
        data := <-ch
        time.Sleep(…)
        //Faking a slow op
    }
}
```

```
func main() {
  wg.Add(2)
  nums := produce(23)
  go consume(nums)
  … // Complex code
  … // so complex I forgot…
  nums <- 0
  //uhoh…
  wg.Wait()
}
```

# Thinking with Channels

- We have to be disciplined about channel communication.

```
$ go run deadlock.go

fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0x119fa50)
   /usr/local/go/src/runtime/sema.go:56 +0x45
sync.(*WaitGroup).Wait(0x119fa48)
   /usr/local/go/src/sync/waitgroup.go:130 +0x65
main.main()
   …/deadlock.go:46 +0xa5

goroutine 7 [chan receive]:
main.consume(0xc000052060, 0x2710)
   …/deadlock.go:34 +0x6b
created by main.main
   …/deadlock.go:41 +0x77
exit status 2
```

# Thinking with Channels

- The example was a bit artificial but…
- Once code gets complex these things can happen if we are not careful!
- …and sometimes the runtime won't help!
- The deadlock detector can only catch **global** deadlocks (i.e. all threads/goroutines are stuck).

# Thinking with Channels

```
func work(data int) {
  for {
   fmt.Println("Working!")
   time.Sleep(1*time.Second)
  }
}
```

```
func recvr(ch <-chan int,
           done chan<- int) {
  val := <-ch
  go work(val)
  done <- val

}

func sender(ch chan<- int) {
  ch <- 42
}
```

```
func main() {
  ch,done := make(chan int),
             make(chan int)
  go recvr(ch,done)
  go sender(ch)
  go sender(ch)

  <-done
  <-done
}
```

# Thinking with Channels

```
func work(data int) {
  for {
   fmt.Println(“Working!”)
   time.Sleep(1*time.Second)
  }
}

func recvr(ch <-chan int,
           done chan<- int) {
  val := <-ch
  go work(val)
  done <- val

}

func sender(ch chan<- int) {
  ch <- 42
}
```

```
func main() {
  ch,done := make(chan int),
             make(chan int)
  go recvr(ch,done)
  go sender(ch)
  go sender(ch)

  <-done
  <-done
}
```

# Thinking with Channels

```
func work(data int) {
  for {
   fmt.Println(“Working!”)
   time.Sleep(1*time.Second)
  }
}

func recvr(ch <-chan int,
           done chan<- int) {
  val := <-ch
  go work(val)
  done <- val

}

func sender(ch chan<- int) {
  ch <- 42
}
```

```
func main() {
  ch,done := make(chan int),
             make(chan int)
  go recvr(ch,done)
  go sender(ch)
  go sender(ch)

  <-done
  <-done
}
```

# Thinking with Channels

```
func work(data int) {
  for {
   fmt.Println("Working!")
   time.Sleep(1*time.Second)
  }
}

func recvr(ch <-chan int,
           done chan<- int) {
  val := <-ch
  go work(val)
  done <- val

}

func sender(ch chan<- int) {
  ch <- 42
}
```

```
func main() {
  ch,done := make(chan int),
             make(chan int)
  go recvr(ch,done)
  go sender(ch)
  go sender(ch)

  <-done
  <-done
}
```

```
Working!
Working!
Working!
...
```

# Thinking with Channels

- **Partial** deadlocks — only a subset of the threads are stuck — are not detected by the runtime.
- Subtle and generally hard to debug when codebase is “real sized”.
- Long running goroutines are not stuck, but may not be doing productive work (undecidable in general):
  - Complex computations (no channel stuff required).
  - N threads ping-ponging “forever”, others waiting on them.
  - …

# Extended Examples

- We have seen a lot of small coordination patterns…
- Lets put some of these micro-patterns to use:
  - Prime Sieve
  - Concurrent binary counter
  - Hadamard Product and Matrix Multiplication

# Prime Sieve - 1st Version

- Set up a pipeline that streams prime numbers
- Successively filtering out multiples of primes.

```
func Producer(ch chan<- int) {
   for i := 2; ; i++ {
      ch <- i
   }
}
func Sieve(in <-chan int,
           out chan<- int,
           prime int) {
   for {
      n := <-in
      if n%prime != 0 {
         out <- n
      }
   }
}
```

```
func main() {
   ch := make(chan int)
   go Producer(ch)
   //Process 'count' from cmd line
   for i := 0; i < count; i++ {
      prime := <-ch
      fmt.Printf("%d ", prime)
      ch1 := make(chan int)
      go Sieve(ch, ch1, prime)
      ch = ch1
   }
   fmt.Println()
}
```

# Prime Sieve - 2nd Version

- Factoring out the spawning of sieves

```
func assembler(in <-chan int,
          result chan<- int) {
   for {
      prime := <-in
      result <- prime
      newCh := make(chan int)
      go Sieve(in,newCh,prime)
      in = newCh
   }
}
```

```
func main() {
   ch := make(chan int)
   go Producer(ch)
   //Process ‘count’ from cmd line
   resCh := make(chan int)
   go assembler(ch, resCh)
   for i := 0; i < count; i++ {
     fmt.Printf("%d ", <-resCh)
   }
   fmt.Println()
}
```

# Prime Sieve - 2nd Version

- Factoring out the spawning of sieves

```
func assembler(in <-chan int,
          result chan<- int) {
   for {
      prime := <-in
      result <- prime
      newCh := make(chan int)
      go Sieve(in,newCh,prime)
      in = newCh
   }
}
```

```
func main() {
   ch := make(chan int)
   go Producer(ch)
   //Process ‘count’ from cmd line
   resCh := make(chan int)
   go assembler(ch, resCh)
   for i := 0; i < count; i++ {
     fmt.Printf("%d ", <-resCh)
   }
   fmt.Println()
}
```

We can do a bit better still…

# Prime Sieve - 3rd Version

- Factoring out the spawning of sieves

```
func assembler(in <-chan int)
                  <-chan int {
   res := make(chan int)
   go func() {
      for {
         prime := <-in
         res <- prime
         newCh := make(chan int)
         go Sieve(in,newCh,prime)
         in = newCh
      }
   }()
   return res
}
```

```
func SieveMachine() <-chan int {
   ch := make(chan int)
   go Producer(ch)
   return assembler(ch)
}

func main() {
   //Process 'count' from cmd line
   primes := SieveMachine()
   for i := 0; i < count; i++ {
      fmt.Printf("%d ", <-primes)
   }
   fmt.Println()
}
```

# Prime Sieve - 3rd Version

- Can also test the “machine”:

```
func TestSieve(t *testing.T) {
   primes := SieveMachine()
   rand.Seed(time.Now().UnixNano())
   numTest := rand.Intn(5000)
   for i := 0; i < numTest; i++ {
      prime := <-primes
      //Baillie-PSW probabilistic primality test
      //100% accurate on inputs < 2^64
      if !big.NewInt(int64(prime)).ProbablyPrime(0) {
         t.Errorf("Number %d is not a prime!", prime)
      }
   }

}
```

# Prime Sieve - 4th Version

- As an exercise, what if we want to “terminate” the sieve?

```
func assembler(in <-chan int) chan int {
   res := make(chan int)
   go func() {
      for {
         select {
         case <-res:
            log.Println("[Primes] Assembler shutdown.")
            return
         default:
            prime := <-in
            res <- prime
            newCh := make(chan int)
            go Sieve(in, newCh, prime)
            in = newCh
         }
      }
   }()
   return res
}
```

# Prime Sieve - 4th Version

- As an exercise, what if we want to “terminate” the sieve?

```
func assembler(in <-chan int) chan int {
   res := make(chan int)
   go func() {
      for {
         select {
         case <-res:
            log.Println("[Primes] Assembler shutd
            return
         default:
            prime := <-in
            res <- prime
            newCh := make(chan int)
            go Sieve(in, newCh, prime)
            in = newCh
         }
      }
   }()
   return res
}
```

Can anyone spot a **bug**?

©Renee French

# Prime Sieve - 4th Version

- As an exercise, what if we want to “terminate” the sieve?

```
type MsgType int

const (
   Kill MsgType = iota
   NextPrime
)

type PrimeMsg struct {
   mtype MsgType
   reply chan int
}
```

```
func assembler(in <-chan int) chan *PrimeMsg {
   res := make(chan *PrimeMsg)
   go func() {
      for {
         msg := <-res
         switch msg.mtype {
         case Kill:
            return
         case NextPrime:
            prime := <-in
            msg.reply <- prime
            newCh := make(chan int)
            go Sieve(in, newCh, prime)
            in = newCh
         }
      }
   }()
   return res
}
```

# Prime Sieve - 4th Version

- As an exercise, what if we want to “terminate” the sieve?

```
func main() {
 ...
 resCh := assembler(ch)
 answerCh := make(chan int)
 for i := 0; i < count; i++ {
     resCh <- &PrimeMsg{mtype:NextPrime,reply: answerCh}
     fmt.Printf("%d ", <-answerCh)
 }
 resCh <- &PrimeMsg{mtype: Kill, reply: nil}
}
```

# Prime Sieve - 4th Version

- As an exercise, what if we want to “terminate” the sieve?

```
func main() {
 ...
 resCh := assembler(ch)
 answerCh := make(chan int)
 for i := 0; i < count; i++ {
     resCh <- &PrimeMsg{mtype:NextPrime,reply: answerCh}
     fmt.Printf("%d ", <-answerCh)
 }
 resCh <- &PrimeMsg{mtype: Kill, reply: nil}
}
```

There are still possible improvements (e.g., terminating each sieve, using arbitrary precision integers)…

# Prime Sieve - 4th Version

- A common idiom: Custom “protocol” messages.
- Messages have a “type” field, workers dispatch on received message type.
- Since Go does not have labelled sums / variants, we need messages to have fields for all possible “types”.
- Idea can be further expanded into multiple protocols within same system (e.g. worker. internal protocol; worker-server protocol).

# Binary Counter

- Encoding a bit string as a chain of nodes (one per bit).
- Nodes can be incremented or queried for the bit string decimal value (up to the node).
- “Leftmost” bit is the most significant bit.

# Binary Counter

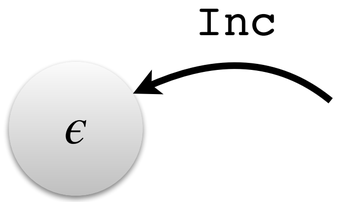
- Empty bit string ($\epsilon$):


```
type MsgType int

const (
   Inc MsgType = iota
   Val
)

type CounterMsg struct {
   mtype MsgType
   payload int
}
```

```
func Eps(ch chan *CounterMsg) {
   for {
      v := <-ch
      switch v.mtype {
      case Inc:
         newCh := make(chan *CounterMsg)
         go Bit(newCh, ch) //bits start at ‘1’
         ch = newCh
      case Val:
         ch <- &CounterMsg{Inc, 0}

      }
   }
}
```

# Binary Counter

- Empty bit string ($\epsilon$):

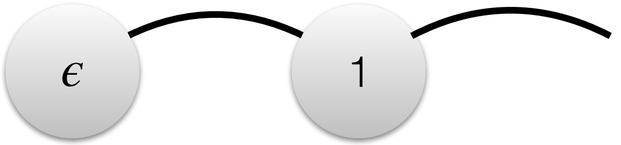



```
type MsgType int

const (
    Inc MsgType = iota
    Val
)

type CounterMsg struct {
    mtype MsgType
    payload int
}
```

```
func Eps(ch chan *CounterMsg) {
    for {
        v := <-ch
        switch v.mtype {
        case Inc:
            newCh := make(chan *CounterMsg)
            go Bit(newCh, ch) //bits start at '1'
            ch = newCh
        case Val:
            ch <- &CounterMsg{Inc, 0}

        }
    }
}
```

# Binary Counter

- Empty bit string ($\epsilon$):



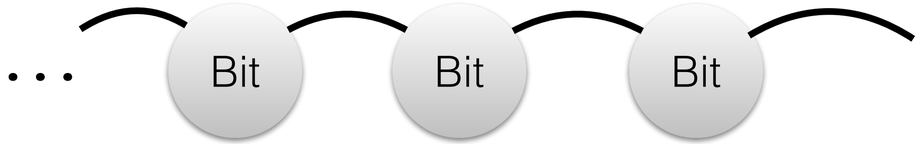

```
type MsgType int

const (
   Inc MsgType = iota
   Val
)

type CounterMsg struct {
   mtype MsgType
   payload int
}
```

```
func Eps(ch chan *CounterMsg) {
   for {
      v := <-ch
      switch v.mtype {
      case Inc:
         newCh := make(chan *CounterMsg)
         go Bit(newCh, ch) //bits start at ‘1’
         ch = newCh
      case Val:
         ch <- &CounterMsg{Inc, 0}

      }
   }
}
```

# Binary Counter

- A bit in the chain:

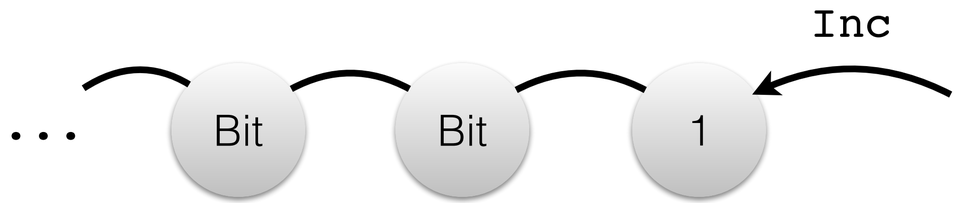
```
func Bit(left, right chan *CounterMsg) {
   bit := 1
   for {
      v := <-right
      switch v.mtype {
      case Inc:
         if bit == 1 {
            left <- &CounterMsg{mtype:Inc}
            bit = 0
         } else {
            bit = 1
         }
      case Val:
         left <- &CounterMsg{mtype:Val}
         msg := <-left
         right <- &CounterMsg{Val, 2*msg.payload + bit}
      }
   }
}
```



# Binary Counter

- A bit in the chain:

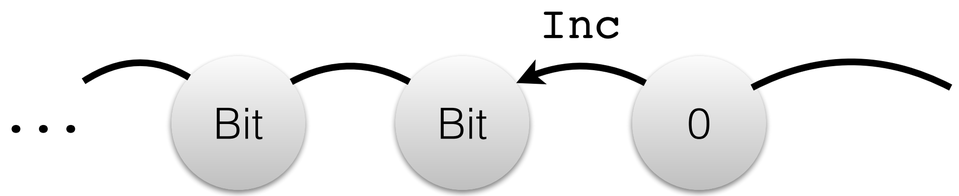
```
func Bit(left, right chan *CounterMsg) {
   bit := 1
   for {
      v := <-right
      switch v.mtype {
      case Inc:
         if bit == 1 {
            left <- &CounterMsg{mtype:Inc}
            bit = 0
         } else {
            bit = 1
         }
      case Val:
         left <- &CounterMsg{mtype:Val}
         msg := <-left
         right <- &CounterMsg{Val, 2*msg.payload + bit}
      }
   }
}
```



# Binary Counter

- A bit in the chain:

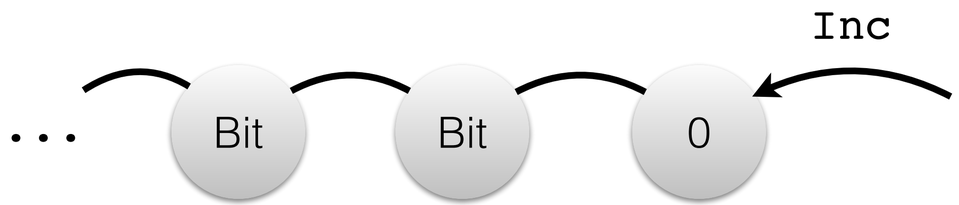
```
func Bit(left, right chan *CounterMsg) {
   bit := 1
   for {
      v := <-right
      switch v.mtype {
      case Inc:
         if bit == 1 {
            left <- &CounterMsg{mtype:Inc}
            bit = 0
         } else {
            bit = 1
         }
      case Val:
         left <- &CounterMsg{mtype:Val}
         msg := <-left
         right <- &CounterMsg{Val, 2*msg.payload + bit}
      }
   }
}
```



# Binary Counter

- A bit in the chain:

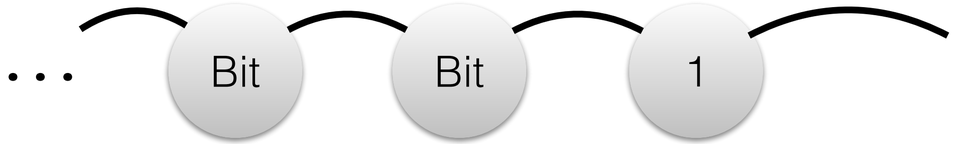
```
func Bit(left, right chan *CounterMsg) {
   bit := 1
   for {
      v := <-right
      switch v.mtype {
      case Inc:
         if bit == 1 {
            left <- &CounterMsg{mtype:Inc}
            bit = 0
         } else {
            bit = 1
         }
      case Val:
         left <- &CounterMsg{mtype:Val}
         msg := <-left
         right <- &CounterMsg{Val, 2*msg.payload + bit}
      }
   }
}
```



# Binary Counter

- A bit in the chain:

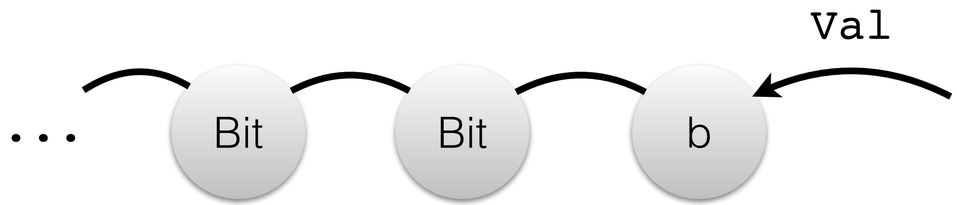
```
func Bit(left, right chan *CounterMsg) {
   bit := 1
   for {
      v := <-right
      switch v.mtype {
      case Inc:
         if bit == 1 {
            left <- &CounterMsg{mtype:Inc}
            bit = 0
         } else {
            bit = 1
         }
      case Val:
         left <- &CounterMsg{mtype:Val}
         msg := <-left
         right <- &CounterMsg{Val, 2*msg.payload + bit}
      }
   }
}
```



# Binary Counter

- A bit in the chain:

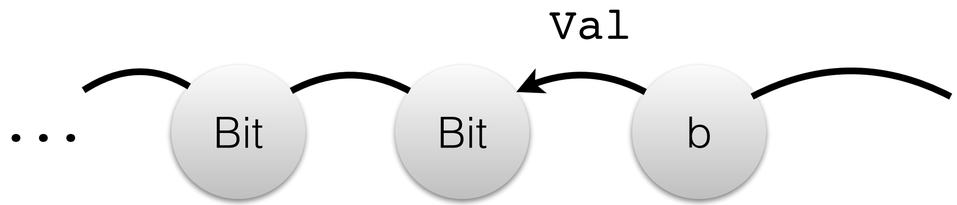
```
func Bit(left, right chan *CounterMsg) {
   bit := 1
   for {
      v := <-right
      switch v.mtype {
      case Inc:
         if bit == 1 {
            left <- &CounterMsg{mtype:Inc}
            bit = 0
         } else {
            bit = 1
         }
      case Val:
         left <- &CounterMsg{mtype:Val}
         msg := <-left
         right <- &CounterMsg{Val, 2*msg.payload + bit}
      }
   }
}
```



# Binary Counter

- A bit in the chain:

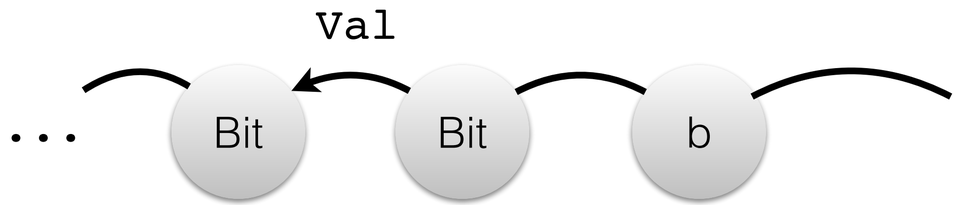
```
func Bit(left, right chan *CounterMsg) {
   bit := 1
   for {
      v := <-right
      switch v.mtype {
      case Inc:
         if bit == 1 {
            left <- &CounterMsg{mtype:Inc}
            bit = 0
         } else {
            bit = 1
         }
      case Val:
         left <- &CounterMsg{mtype:Val}
         msg := <-left
         right <- &CounterMsg{Val, 2*msg.payload + bit}
      }
   }
}
```



# Binary Counter

- A bit in the chain:

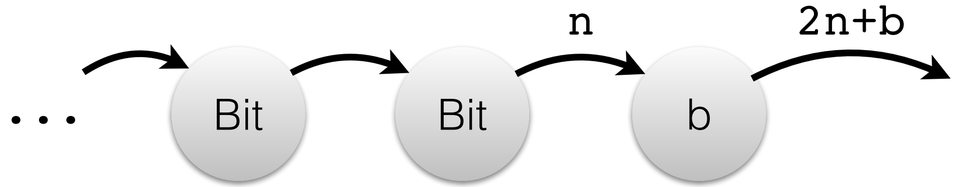
```
func Bit(left, right chan *CounterMsg) {
   bit := 1
   for {
      v := <-right
      switch v.mtype {
      case Inc:
         if bit == 1 {
            left <- &CounterMsg{mtype:Inc}
            bit = 0
         } else {
            bit = 1
         }
      case Val:
         left <- &CounterMsg{mtype:Val}
         msg := <-left
         right <- &CounterMsg{Val, 2*msg.payload + bit}
      }
   }
}
```



# Binary Counter

- A bit in the chain:

```
func Bit(left, right chan *CounterMsg) {
   bit := 1
   for {
      v := <-right
      switch v.mtype {
      case Inc:
         if bit == 1 {
            left <- &CounterMsg{mtype:Inc}
            bit = 0
         } else {
            bit = 1
         }
      case Val:
         left <- &CounterMsg{mtype:Val}
         msg := <-left
         right <- &CounterMsg{Val, 2*msg.payload + bit}
      }
   }
}
```



# Hadamard Product

Given two matrices $A$ and $B$ with the same dimensions, the Hadamard product $A \circledcirc B$ is a matrix $C$ with the same dimensions as $A$ and $B$ and elements given by:

$$c_{ij} = a_{ij} \circ b_{ij}$$

# Hadamard Product

- Generate all coord. pairs of result matrix, feed into “work” channel.
- Workers drain the “work”, computing the element product and writing in the result matrix. Reads and Writes are non-interfering.
- Closing “work” channels signals no more work. Workers signal end of computation via workgroup.

# Hadamard Product

```
type Coord struct {
   row, col int
}
```

```
func Hadamard(m1, m2 [][]float64, nWorkers int) [][]float64
{
   rows := len(m1)
   cols := len(m1[0])
   //Allocate answer matrix
   res := make([][]float64, rows)
   for i := range res {
      res[i] = make([]float64, cols)
   }
   //Wait group to notify end of computation
   var wg sync.WaitGroup
   wg.Add(nWorkers)
   work := make(chan *Coord, QSIZE)
   //Spawn workers
   for i := 0; i < nWorkers; i++ {
      go calcHadamard(work, m1, m2, res, &wg)
   }
   ...
```

# Hadamard Product

```
type Coord struct {
   row, col int
}
```

```
func Hadamard(m1, m2 [][]float64, nWorkers int) [][]float64
{
   rows := len(m1)
   cols := len(m1[0])
   //Allocate answer matrix
   res := make([][]float64, rows)
   for i := range res {
      res[i] = make([]float64, cols)
   }
   //Wait group to notify end of computation
   var wg sync.WaitGroup
   wg.Add(nWorkers)
   work := make(chan *Coord, QSIZE)
   //Spawn workers
   for i := 0; i < nWorkers; i++ {
      go calcHadamard(work, m1, m2, res, &wg)
   }
   ...
```

# Hadamard Product

```
type Coord struct {
   row, col int
}
```

```
func Hadamard(m1, m2 [][]float64, nWorkers int) [][]float64
{
   ...
   for i := 0; i < nWorkers; i++ {
      go calcHadamard(work, m1, m2, res, &wg)
   }

   for i := 0; i < rows; i++ {
      for j := 0; j < cols; j++ {
         work <- &Coord{i, j}
      }
   }
   close(work)
   wg.Wait()
   return res
}
```

# Hadamard Product

```
type Coord struct {
   row, col int
}
```

```
func Hadamard(m1, m2 [][]float64, nWorkers int) [][]float64
{
   ...
   for i := 0; i < nWorkers; i++ {
      go calcHadamard(work, m1, m2, res, &wg)
   }

   for i := 0; i < rows; i++ {
      for j := 0; j < cols; j++ {
         work <- &Coord{i, j}
      }
   }
   close(work)
   wg.Wait()
   return res
}
```

# Hadamard Product

```
type Coord struct {
   row, col int
}
```

```
func calcHadamard(queue chan *Coord, m1, m2, m3 [][]float64,
                  wg *sync.WaitGroup) {
   defer wg.Done()
   for {
      work, ok := <-queue
      if !ok {
         break
      }
      m3[work.row][work.col] = m1[work.row][work.col] *
                               m2[work.row][work.col]
   }

}
```

# Hadamard Product

```
type Coord struct {
   row, col int
}
```

```
func calcHadamard(queue chan *Coord, m1, m2, m3 [][]float64,
                  wg *sync.WaitGroup) {
   defer wg.Done()
   for {
      work, ok := <-queue
      if !ok {
         break
      }
      m3[work.row][work.col] = m1[work.row][work.col] *
                               m2[work.row][work.col]
   }

}
```

# Hadamard Product

```
type Coord struct {
   row, col int
}
```

```
func calcHadamard(queue chan *Coord, m1, m2, m3 [][]float64,
                  wg *sync.WaitGroup) {
   defer wg.Done()
   for {
      work, ok := <-queue
      if !ok {
         break
      }
      m3[work.row][work.col] = m1[work.row][work.col] *
                               m2[work.row][work.col]
   }

}
```

# Matrix Multiplication

- Same strategy as before, but now workers perform the matrix cell calculation:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

# Matrix Multiplication

```
type Coord struct {
   row, col int
}
```

```
func Product(m1, m2 [][]float64, nWorkers int) [][]float64 {
   rows := len(m1)
   cols := len(m1[0])
   //Allocate answer matrix
   res := make([][]float64, rows)
   for i := range res {
      res[i] = make([]float64, cols)
   }
   //Wait group to notify end of computation
   var wg sync.WaitGroup
   wg.Add(nWorkers)
   work := make(chan *Coord, QSIZE)
   //Spawn workers
   for i := 0; i < nWorkers; i++ {
      go calcProduct(work, m1, m2, res, &wg)
   }
   //Rest as before
   ...
```

# Matrix Multiplication

```
type Coord struct {
   row, col int
}
```

```
func calcProduct(queue chan *Coord, m1, m2, m3 [][]float64,
                 wg *sync.WaitGroup) {
   defer wg.Done()
   for {
      work, ok := <-queue
      if !ok {
         break
      }
      for i := 0; i < len(m1[0]); i++ {
         m3[work.row][work.col] += m1[work.row][i] *
                                   m2[i][work.col]
      }
   }

}
```

# Generic Matrix Products

```
type Coord struct {
   row, col int
}
```

```
func MatrixProds(m1, m2 [][]float64, nWorkers int,
   op func(chan *Coord, [][]float64, [][]float64,
          [][]float64, *sync.WaitGroup)) [][]float64 {
   rows := len(m1)
   cols := len(m1[0])
   //Allocate answer matrix
   res := make([][]float64, rows)
   for i := range res {
      res[i] = make([]float64, cols)
   }
   //Wait group to notify end of computation
   var wg sync.WaitGroup
   wg.Add(nWorkers)
   work := make(chan *Coord, QSIZE)
   //Spawn workers
   for i := 0; i < nWorkers; i++ {
      go op(work, m1, m2, res, &wg)
   }
   //Rest as before
```

# Generic Matrix Products

- Generate “work”, asynchronously, into a queue.
- Workers are spawned independently of work.
- Workers drain the queue and “do the work”.
- Generalized barber shop problem, “for free”, using channels.

# Barbershop

- Generalized barber shop problem, “for free”, using channels.
- Workers are the barbers
- Work queue is the waiting room.

# Barbershop

```
type BarberShop struct {
  capacity int
  nBarbers int

  barbersDone chan bool
  clients chan int //async of size ‘capacity’

}
```

```
func New(capacity int) BarberShop {…}
func (shop *BarberShop) AddClient(client int) {
 if shop.open {
 select {
  case shop.clients <- client:
   fmt.Printf(“Added client %d\n”,client)
  default:
   fmt.Printf(“Shop full, try again later”)
   }
 }} else { fmt.Println{“Shop closed!”) }
```

# Barbershop

```
type BarberShop struct {
  capacity int
  nBarbers int

  barbersDone chan bool
  clients chan int //async of size ‘capacity’

}

func New(capacity int) BarberShop {…}
func (shop *BarberShop) AddClient(client int) {
 if shop.open {
 select {
  case shop.clients <- client:
   fmt.Printf(“Added client %d\n”,client)
  default:
   fmt.Printf(“Shop full, try again later”)
   }
 }} else { fmt.Println{“Shop closed!”) }
```

# Barbershop

```
func (shop *BarberShop) AddBarber() {
  shop.nBarbers++;
  go func() {
    for {
      if len(shop.clients) == 0 {
        fmt.Printf(“A barber is sleeping\n")
      }
      client, shopOpen := <- shop.clients
      if shopOpen {
         //cutHair(client)
      } else {
         shop.barbersDone <- true
      }
    }
  }
}
```

```
func (shop *BarberShop) Close() {
  shop.open = false
  close(shop.clients)
  for a := 0; a < shop.nBarbers; a++ {
    <-shop.barbersDone
  }
  close(shop.barbersDone)
}
```

# Barbershop

```
func (shop *BarberShop) AddBarber() {
  shop.nBarbers++;
  go func() {
    for {
      if len(shop.clients) == 0 {
        fmt.Printf(“A barber is sleeping\n")
      }
      client, shopOpen := <- shop.clients
      if shopOpen {
         //cutHair(client)
      } else {
         shop.barbersDone <- true
      }
    }
  }
}
```

```
func (shop *BarberShop) Close() {
  shop.open = false
  close(shop.clients)
  for a := 0; a < shop.nBarbers; a++ {
    <-shop.barbersDone
  }
  close(shop.barbersDone)
}
```

# Summary

- Channels are a great tool for synchronizing and sharing data among goroutines.
- Minimize a lot of the usual pains of concurrent programming.
- But not all of them:
  - **Data races** can still happen (i.e., channel-based programs still use other types of data).
  - **Global deadlocks** if we are not careful.
  - **Partial deadlocks** if we are not really careful.
- Active research area in PL / Concurrency theory (**Lecture 4**).

# Next week

- Generalizing high-level concurrent programming patterns:
  - Propagating termination via Contexts
  - Pipelines
  - Fan-out, Fan-in
  - Worker pools
  - …