Parallel Programming

Channels

Morgan Ericsson

Today

- Channels
- Concurrency patterns

Message passing

- Processes communicate by sending messages
- Synchronous or Asynchronous
- Buffering
- Kernel or Process

Channels

```
 1  func main() {
 2      strCh := make(chan string)
 3      go func() {
 4          strCh <- "Alice"
 5      }()
 6
 7      if res, ok := <-strCh; ok {
 8          fmt.Println("Hello", res)
 9      }
10  }
```

Channels

- Channels send and receive messages
  - `[channel] <- [value]` to send
  - `[value], [status] <-[channel]` to receive
- Channels are typed and can only send and receive messages of that type
- A form of IPC

CSP

- Go's channels are based on Communicating Sequential Processes (CSP)
- Idea that input and output need to be considered language primitives
- Communication between processes
- First-class support for modeling communication makes solving problems simpler and easier to comprehend

CSP

- `cardreader?cardimage`: read from cardreader and assign to cardimage
- `lineprinter!lineimage`: send value of lineimage to lineprinter
- `*[c:character; west?c → east!c]`: read all characters from west and pass them one by one to west. End when there are no more characters

Go

- Supports both shared-memory programming and primitives and CSP
- CSP
  - Channels models input and output
  - Communication between goroutines

Channels

Go

```
 1  func main() {
 2      strCh := make(chan string)
 3      go func() {
 4          strCh <- "Alice"
 5      }()
 6
 7      go func() {
 8          if res, ok := <-strCh; ok {
 9              fmt.Println("Hello", res)
10          }
11      }()
12      time.Sleep(time.Second)
13  }
```

Channels

- A language primitive
  - A channel can be used as any other variable
- Unbuffered or buffered
- uni- or bidirectional
- Composable

Unbuffered

```
 1  func main() {
 2      tickCh := make(chan int)
 3      go func() {
 4          tmp := time.Now()
 5          for i := 0; i < 5; i++ {
 6              tickCh <- i
 7              fmt.Println("Wrote after", time.Since(tmp))
 8          }
 9      }()
10
11      go func() {
12          for {
13              _ = <-tickCh
14              time.Sleep(time.Second)
15          }
16      }()
17      time.Sleep(8 * time.Second)
18  }
```

Unbuffered

```
Wrote after 0s
Wrote after 1s
Wrote after 2s
Wrote after 3s
Wrote after 4s
```

Unbuffered

- Reads will block if the channel is empty
- Writes will block if the channel is full
- An unbuffered channel can hold a single value
- Buffered channels can hold an arbitrary number of values
  - Specified when created

Buffered

```
 1  func main() {
 2      tickCh := make(chan int, 2)
 3      go func() {
 4          tmp := time.Now()
 5          for i := 0; i < 5; i++ {
 6              tickCh <- i
 7              fmt.Println("Wrote after", time.Since(tmp))
 8          }
 9      }()
10
11      go func() {
12          for {
13              _ = <-tickCh
14              time.Sleep(time.Second)
15          }
16      }()
17      time.Sleep(8 * time.Second)
18  }
```

Buffered

```
Wrote after 0s
Wrote after 0s
Wrote after 0s
Wrote after 1s
Wrote after 2s
```

Bidirectional

```
 1  func main() {
 2      mch := func() chan int {
 3          ch := make(chan int)
 4          go func() {
 5              _ = <-ch
 6          }()
 7          return ch
 8      }()
 9
10      mch <- 1
11  }
```

Uni- and bidirectional

- Channels are bidirectional by default (`chan`)
- Can be implicitly converted to a unidirectional channel, e.g., by parameter or return type
- Unidirectional channels are defined by adding `<-` before or after `chan`
  - Position specifies direction

Unidirectional

```
 1  func main() {
 2      mch := func() <-chan int {
 3          ch := make(chan int)
 4          go func() {
 5              _ = <-ch
 6          }()
 7          return ch
 8      }()
 9
10      mch <- 1
11  }
```

```
./prog.go:14:2: invalid operation: cannot send to receive-only channel mch (variable of type <-chan int)
```

Unidirectional

```
 1  func main() {
 2      mch := func() chan<- int {
 3          ch := make(chan int)
 4          go func() {
 5              _ = <-ch
 6          }()
 7          return ch
 8      }()
 9
10      mch <- 1
11  }
```

Closing channels

```
 1  func main() {
 2      var wg sync.WaitGroup
 3      ch := make(chan int)
 4      wg.Add(1)
 5      go func() {
 6          defer wg.Done()
 7          st := time.Now()
 8          _ = <-ch
 9          fmt.Println("Channel closed after", time.Since(st))
10      }()
11      time.Sleep(5 * time.Second)
12      close(ch)
13      wg.Wait()
14  }
```

Closing channels

- It is not possible to write to a closed channel
- Reads will return already written values or nothing
- The error return value indicates if the channel is open or closed

Ranging over channels

```
 1  gen := func () <-chan int {
 2      ch := make(chan int)
 3      go func() {
 4          defer close(ch)
 5          for i := 0;i<5;i++ {
 6              ch <- i
 7          }
 8      }()
 9      return ch
10  }
```

Ranging over channels

```
1  ch := gen()
2  for r := range ch {
3      fmt.Println("Got:", r)
4  }
```

Ranging over channels

- `for ... range [channel]` is equivalent to reading in a loop and stopping when channel is drained and closed
- Common pattern
  - Just remember to close the channel

Channels and state

- **Read**
  - `nil` will **block**
  - *open* and *not empty* will **read** the value
  - *open* and *empty* will **block**
  - *closed* will **return** zero value and false
  - *send only* will not compile

Channels and state

- **Write**
  - `nil` will **block**
  - *open* and *full* will **block**
  - *open* and *not full* will **write** the value
  - *closed* will **panic**
  - *receive only* will not compile

Channels and state

- **Close**
  - `nil` will **panic**
  - *open* and *not empty* will **close** the channel but values are available until read
  - *open* and *empty* will **close** the channel. Any reads return the zero value
  - *closed* will **panic**
  - *receive only* will not compile

Ownership

- A channel is “owned” by the creator
  - ownership can be passed
- Owner should write to and close the channel
  - again, can be passed
- Use unidirectional channels to enforce
- A goroutine does not own a read-only channel

Why?

- Writing to nil deadlocks
- Closing a closed or nil channel panics
- Writing to a closed channel panics
- Owner's responsibility to manage
- Consumer of a channel only needs to worry about
  - reading from a closed channel
  - blocking on reads

Select

```
1 var c1, c2 <-chan int
2 select {
3     case <- c1:
4         // Do something
5     case <- c2:
6         // Do something else
7 }
```

Select

- Select is a switch for channel
- Waits on multiple channels
  - Unblocks and runs a case when possible

Select

```
 1  func main() {
 2      st := time.Now()
 3      ch := make(chan interface{})
 4      go func() {
 5          time.Sleep(5 * time.Second)
 6          close(ch)
 7      }()
 8
 9      fmt.Println("Blocking")
10      select {
11      case <-ch:
12          fmt.Println("Unblocked after", time.Since(st))
13      }
14  }
```

Select

- Closing a channel will unblock any operations waiting on it
- Common way to signal goroutines that something has happened, e.g., that they should terminate

Select

```
 1  func main() {
 2      c1 := make(chan interface{})
 3      close(c1)
 4      c2 := make(chan interface{})
 5      close(c2)
 6
 7      var c1Count, c2Count int
 8      for i := 1000; i >= 0; i-- {
 9          select {
10          case <-c1:
11              c1Count++
12          case <-c2:
13              c2Count++
14          }
15      }
16
17      fmt.Println("c1Count:", c1Count, "c2Count:", c2Count)
18  }
```

Select

- Channels are not processed in order (unlike switch)
  - would make it difficult to use select to manage multiple channels
- The Go runtime randomly selects a channel if multiple are available
  - Aims for a uniform distribution

Runs

- c1Count: 534 ; c2Count: 466
- c1Count: 503 ; c2Count: 497
- c1Count: 481 ; c2Count: 519
- c1Count: 500 ; c2Count: 500
- c1Count: 510 ; c2Count: 490

Blocked

- What if there are no available channels?
  - selected blocked forever
- Sometimes what we want, but not always
- We can introduce a timeout or not block

Unblocking after some time

```
 1  func main() {
 2      st := time.Now()
 3      ch := make(chan interface{})
 4      go func() {
 5          time.Sleep(5 * time.Second)
 6          close(ch)
 7      }()
 8
 9      fmt.Println("Blocking")
10      select {
11      case <-ch:
12          fmt.Println("Unblocked after", time.Since(st))
13      case <-time.After(time.Second):
14          fmt.Println("Unblocked after", time.Since(st))
15      }
16  }
```

Unblocking after some time

- Note that `time.After` returns a channel that sends a value after some specified time
- The default clause will be executed if there are no available values
  - So, a select with default “never” blocks

Unblocking

```
 1  func main() {
 2      st := time.Now()
 3      ch := make(chan interface{})
 4      go func() {
 5          time.Sleep(5 * time.Second)
 6          close(ch)
 7      }()
 8
 9      fmt.Println("Blocking")
10      select {
11      case <-ch:
12          fmt.Println("Unblocked after", time.Since(st))
13      default:
14          fmt.Println("Unblocked after", time.Since(st))
15      }
16  }
```

Patterns

Patterns

- Channels and goroutines can be used to design interesting solutions to concurrency problems
- Can often avoid the need for shared state/memory
  - State is communicated
  - Synchronization from communication

Done and for-select

```
 1  func doSmth(done <-chan interface{}) {
 2      for {
 3          select {
 4          case <-done:
 5              return
 6          default:
 7          }
 8
 9          // Do some work
10      }
11  }
```

Done and for-select

- A done-channel can be used to tell goroutines to stop/exit
- Default means we do not block and for that we check continuously
- Note, `interface{}` (or `any`) specifies that the channel can accept any type
  - But, does not matter since we never send anything to the channel

Leaking Goroutines

```
1  func main() {
2      doSmth := func(str <-chan string) {
3          for s := range str {
4              fmt.Println(s)
5          }
6      }
7      go doSmth(nil)
8      // Do other work ...
9  }
```

Leaking Goroutines

- The for will never progress on a nil channel, so the function will never exit
  - unless the program exits
- A leaked goroutine
  - similar to memory leaks
- Can be a problem in long-running programs
  - goroutines are cheap but not free
  - resources not reclaimed until it exits

Fixed (as long as we close done)

```
1  func doSmth(str <-chan string, done <-chan interface{}) {
2      for {
3          select {
4              case res, ok := <-str:
5                  if !ok {
6                      return
7                  }
8                  // do something with res
9              case <-done:
10                 return
11         }
12     }
13 }
```

Done channels

- When composing goroutines and channels we might end up with multiple done channels
- Sufficient to wait for a single done message, so reduces to an or
- So, we need or over multiple channels

Or channel

```
 1  func or(ch1, ch2 <-chan interface{}) <-chan interface{} {
 2      orDone := make(chan interface{})
 3      go func() {
 4          defer close(orDone)
 5          select {
 6          case <-ch1:
 7          case <-ch2:
 8          }
 9      }()
10      return orDone
11  }
```

Or channel

- The or function takes two channels and returns a channel
- The returned channel will close when either of the two channels unblocks
- Can use this idea and varadic functions to create an or that waits on an arbitrary number of done channels
- Note that this pattern cannot be used to multiplex channels

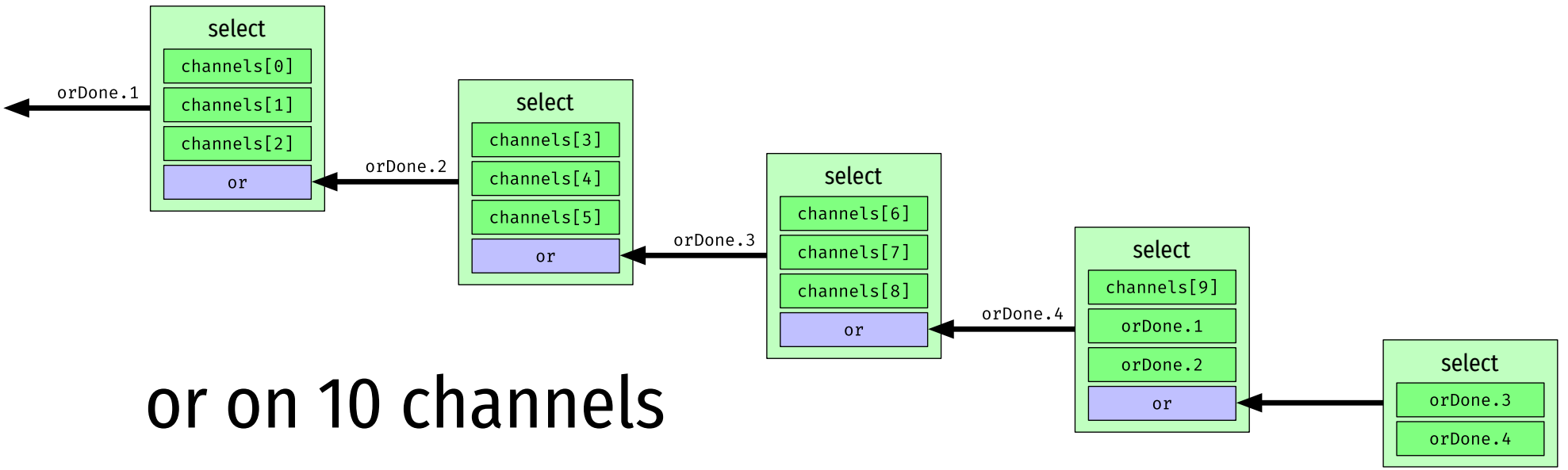
Arbitrary number of channels?

```
1  func or(channels ...<-chan interface{}) <-chan interface{} {
2      switch len(channels) {
3      case 0: return nil
4      case 1: return channels[0]
5      }
6
7      //...
```

Arbitrary number of channels?

```
    // ...
    orDone := make(chan interface{})
    go func() {
        defer close(orDone)
        switch len(channels) {
        case 2:
            select {
            case <-channels[0]:
            case <-channels[1]:
            }
        default:
            select {
            case <-channels[0]:
            case <-channels[1]:
            case <-channels[2]:
            case <-or(append(channels[3:], orDone)...):
            }
        }
    }()
    return orDone
}
```

Arbitrary number of channels?



or on 10 channels

Example

```
 1  sig := func(after time.Duration) <-chan interface{} {
 2      ch := make(chan interface{})
 3      go func() {
 4          defer close(ch)
 5          time.Sleep(after)
 6      }()
 7      return ch
 8  }
 9
10  start := time.Now()
11  <-or(
12      sig(2 * time.Hour),
13      sig(5 * time.Minute),
14      sig(time.Second),
15      sig(time.Hour),
16      sig(time.Minute),
17  )
18  fmt.Println("Done after", time.Since(start))
```

isPrime (from Wikipedia)

```go
func isPrime(num int) bool {
    if num > 1 && num <= 3 {
        return true
    }
    if num <= 1 || num%2 == 0 || num%3 == 0 {
        return false
    }

    for i := 5; i*i <= num; i += 6 {
        if num%i == 0 || num%(i+2) == 0 {
            return false
        }
    }
    return true
}
```

Find primes

```
1  func main() {
2      for i:=0;i<1000000;i++ {
3          if isPrime(i) {
4              fmt.Println("Got", i)
5          }
6          i += 1
7      }
8  }
```

Fork-join

```
 1  ch := make(chan int)
 2  for i := 0; i < 10; i++ {
 3      go func(lo, hi int) {
 4          for i := lo; i < hi; i++ {
 5              if isPrime(i) {
 6                  ch <- i
 7              }
 8          }
 9      }(i*100000, (i+1)*100000)
10  }
11
12  for p := range ch {
13      fmt.Println(p)
14  }
```

Does not work: “fatal error: all goroutines are asleep - deadlock!”

Why?

- All the goroutines will exit when they have checked their range
- The channel is still open, so the for will expect more values
- But there will be no more values, so progress is not possible!
- How can we close the channel?

Fork-join

```
 1  var wg sync.WaitGroup
 2  ch := make(chan int)
 3  for i := 0; i < 10; i++ {
 4      wg.Add(1)
 5      go func(lo, hi int) {
 6          defer wg.Done()
 7          for i := lo; i < hi; i++ {
 8              if isPrime(i) {
 9                  ch <- i
10              }
11          }
12      }(i*100000, (i+1)*100000)
13  }
14
15  for p := range ch {
16      fmt.Println(p)
17  }
```

Fork-join

- Channel is basically a thread-safe queue
- We split based on benchmarks (or gut feeling)
- Can we try a different approach?

Pipelining

```
 1  func filterPrime(ins <-chan int) <-chan int {
 2      outCh := make(chan int)
 3      go func() {
 4          for i := range ins {
 5              if isPrime(i) {
 6                  outCh <- i
 7              }
 8          }
 9      }()
10
11      return outCh
12  }
```

Pipelining

```
 1  func genInts(done <-chan interface{}) <-chan int {
 2      outCh := make(chan int)
 3      go func() {
 4          i := 1
 5          for {
 6              select {
 7              case <-done:
 8                  close(outCh)
 9                  return
10              case outCh <- i:
11              }
12              i += 1
13          }
14      }()
15      return outCh
16  }
```

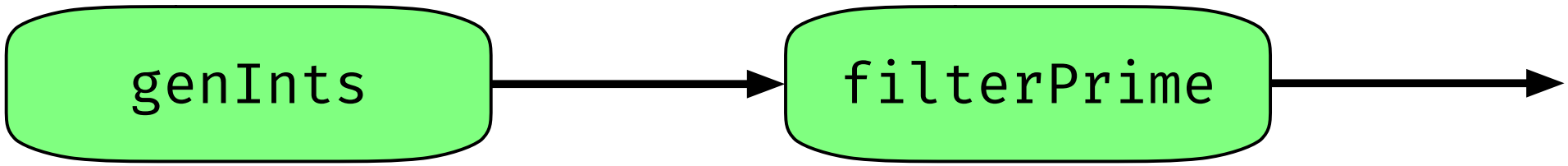
Remember

- Note that both functions own their channels
  - Returns a read-only version
- Minor issue in filterPrimes, can leak if channel is never closed
  - Should fix with the done-channel pattern

Pipelining

```
 1  func main() {
 2      var doneCh chan interface{}
 3
 4      go func() {
 5          time.Sleep(5 * time.Second)
 6          close(doneCh)
 7      }()
 8
 9      intS := genInts(doneCh)
10      divCh := filterPrime(intS)
11      for r := range divCh {
12          fmt.Println("Got", r)
13      }
14      fmt.Println("Done")
15  }
```

Pipelining



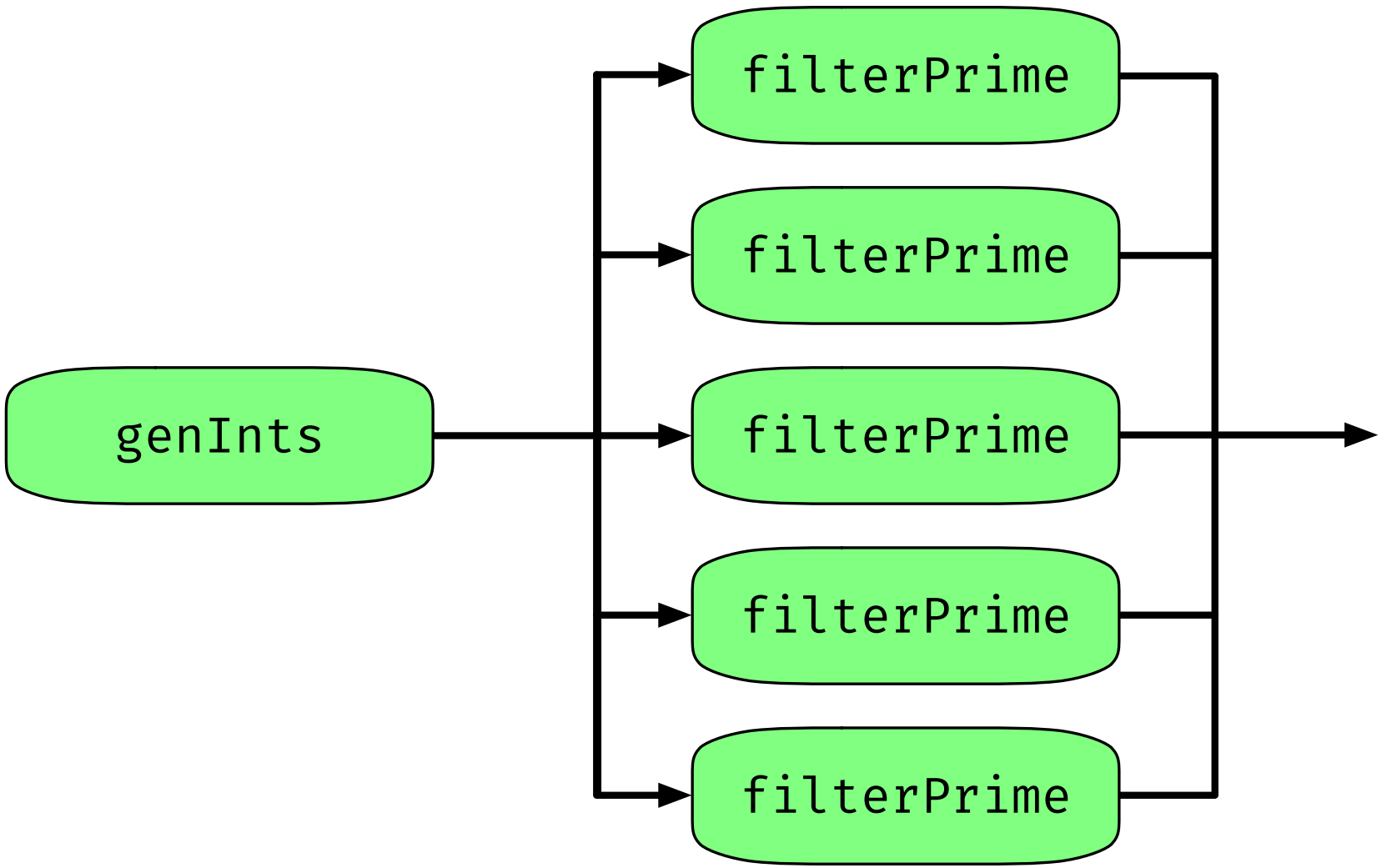
Pipelining

- Our solution drains the int channel
- We could pass the done channel to `filterPrime` and stop without necessarily draining
- Common pattern in pipelines that might not drain the channel
- We risk leaking the goroutine!

Pipelining

- In this case, no meaningful concurrency
  - Generation and checking split
- However, quite easy to add concurrency
  - By “fanning out” the pipeline

Fan-out


genInts
filterPrime
filterPrime
filterPrime
filterPrime
filterPrime

Fan-out

- If shared output channel, when and where do we close it?
- If N output channels, how do we handle these?
  - Drain each in a loop?
- Fan-in
  - Similar to or, but forwards values

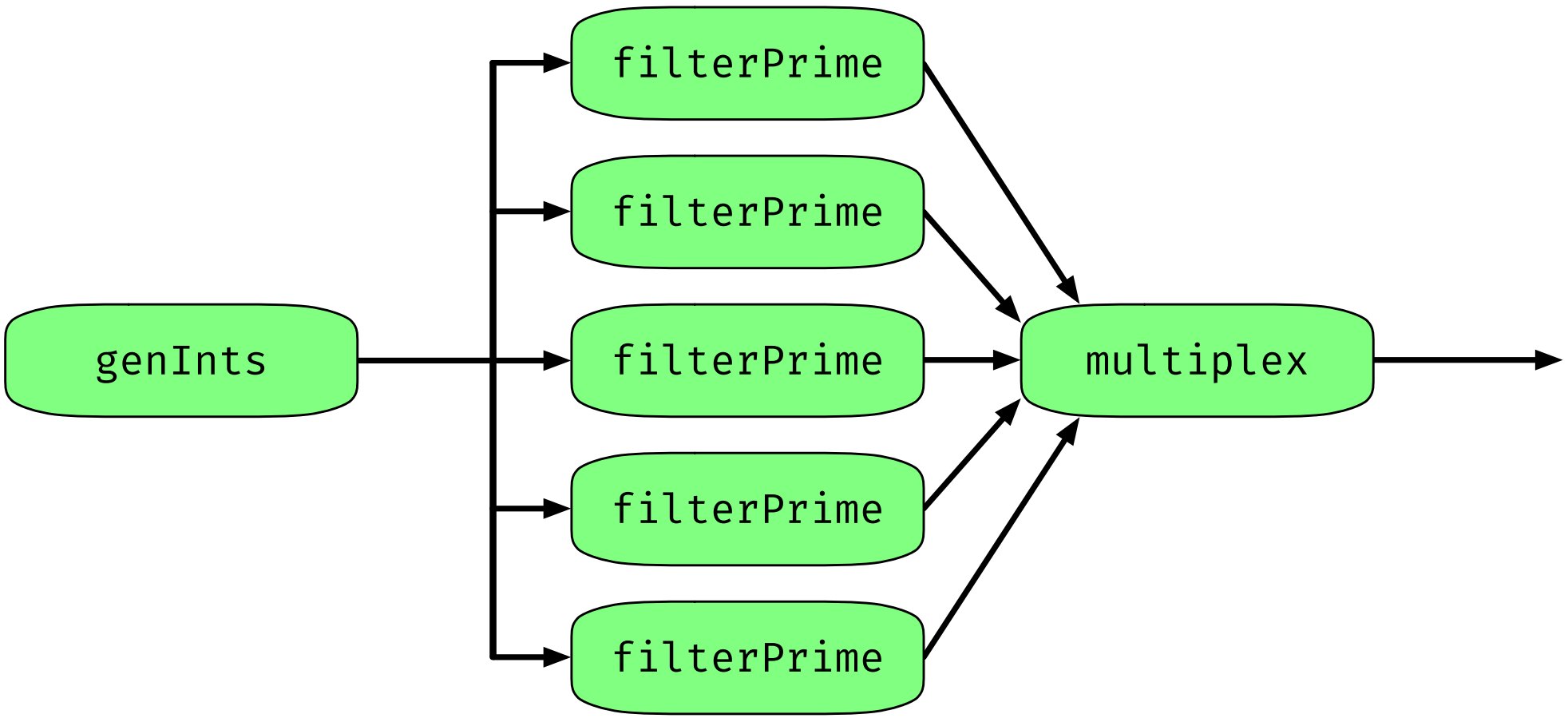
Fan-in

```
 1  func multiplexer(channels ...<-chan int) <-chan int {
 2      var wg sync.WaitGroup
 3      mpOut := make(chan int)
 4
 5      for _, ch := range channels {
 6          wg.Add(1)
 7          go func(c <-chan int) {
 8              defer wg.Done()
 9              for i := range c {
10                  mpOut <- i
11              }
12          }(ch)
13      }
14
15      go func() {
16          wg.Wait()
17          close(mpOut)
18      }()
19
20      return mpOut
21  }
```

Primes

```
 1  func main() {
 2      var doneCh chan interface{}
 3      chs := make([]<-chan int, 5)
 4
 5      go func() {
 6          time.Sleep(5 * time.Second)
 7          close(doneCh)
 8      }()
 9
10      intS := genInts(doneCh)
11      for i:=0;i<5;i++ {
12          chs[i] = filterPrime(intS)
13      }
14      pCh := multiplexer(chs...)
15      for r := range pCh {
16          fmt.Println("Got", r)
17      }
18      fmt.Println("Done")
19  }
```

Primes


filterPrime
filterPrime
genInts
filterPrime
multiplex
filterPrime
filterPrime

Fixing the potential leak

```
 1  func filterPrime(ins <-chan int, done <-chan interface{}) <-chan int {
 2      outCh := make(chan int)
 3      go func() {
 4          defer close(outCh)
 5          for {
 6              select {
 7              case <-done:
 8                  return
 9              case i, ok := <-ins:
10                  if !ok { return }
11              }
12              if isPrime(i) {
13                  outCh <- i
14              }
15          }
16      }()
17
18      return outCh
19  }
```

Annoying!

- We cannot range over the channel
- Easy to make mistakes
- Multiplex the two channels
  - done closes the channel we read from
- Allows a “simple” for range

Or-done

```
1  func orDone(done, ch <-chan interface{}) <-chan interface{} {
2      outCh := make(chan interface{})
3      go func() {
4          defer close(outCh)
5          for {
6              select {
7              case <-done:
8                  return
9              case v, ok := <-ch:
10                 if !ok { return }
11                 select {
12                 case outCh <- v:
13                 case <-done:
14                 }
15             }
16         }
17     }()
18     return outCh
19 }
```

New version

```
 1  func filterPrime(ins <-chan int, done <-chan interface{}) <-chan int {
 2      outCh := make(chan int)
 3      go func() {
 4          for i := range orDone(ins, done) {
 5              if isPrime(i) {
 6                  outCh <- i
 7              }
 8          }
 9      }()
10
11      return outCh
12  }
```

Channels

- We can do a lot more with channels
  - A tee-channel that splits for, e.g., logging
  - ...
- We can pass channels over channels
  - The multiplexer could be rewritten to take a channel that contains channels
  - And bridge these

Concurrency in Go

Remember

> Concurrency is a property of the code; parallelism is a property of the running program.

Concurrency in Go

> Do not communicate by sharing memory. Instead, share memory by communicating.

Always?

- Channels should be your first choice
- Unless you need to protect some internal state
- If channels are not performant enough, consider shared memory
  - But try to refactor first

Other languages

- Similar ideas work
- Tasks (however they are formulated) and thread-safe queues

Next time

» Deadlocks