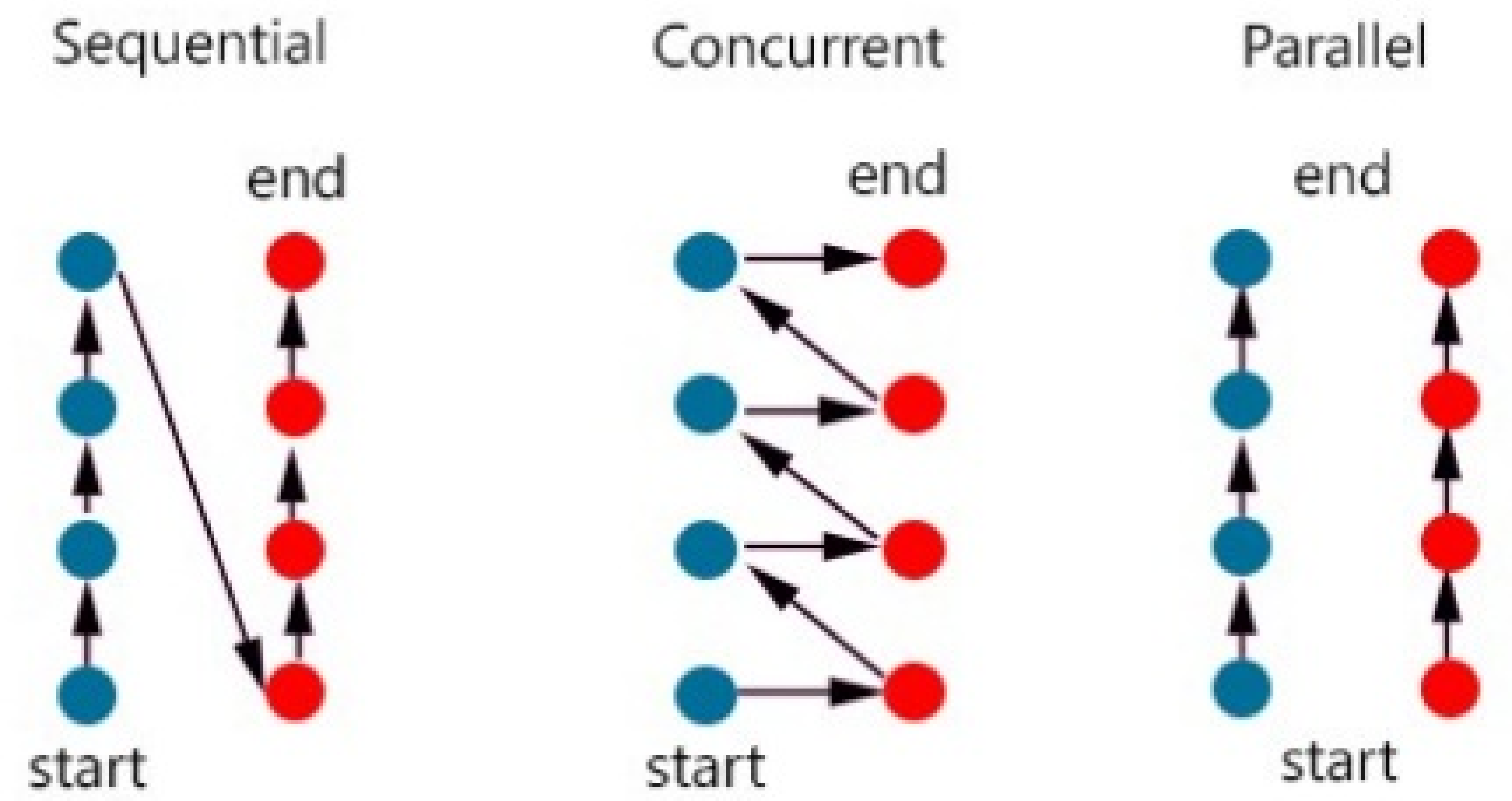
Concurrent Haskell

"Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once."
— Rob Pike



sequential vs concurrent vs parallel

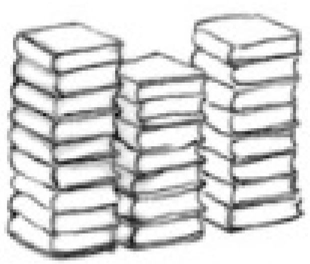
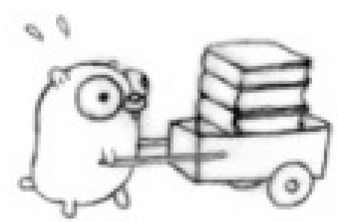
“Concurrency is about structure, parallelism is about execution.”

Credits: https://kwahome.medium.com/concurrency-is-not-parallelism-a5451d1cde8d

Analogy

Our problem

Move a pile of obsolete language manuals to the incinerator.

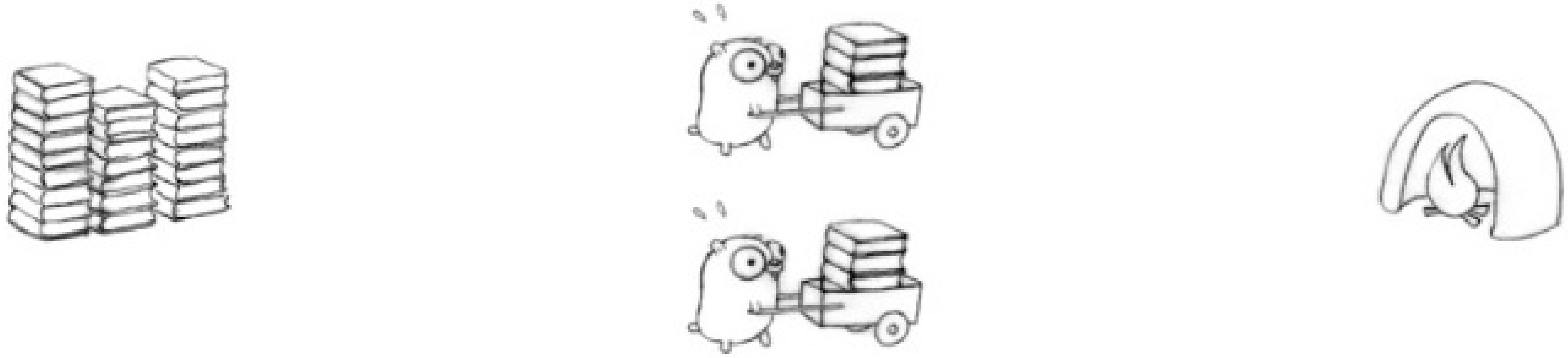




With only one gopher this will take too long.

Credits: https://go.dev/blog/waza-talk

Analogy

More gophers and more carts



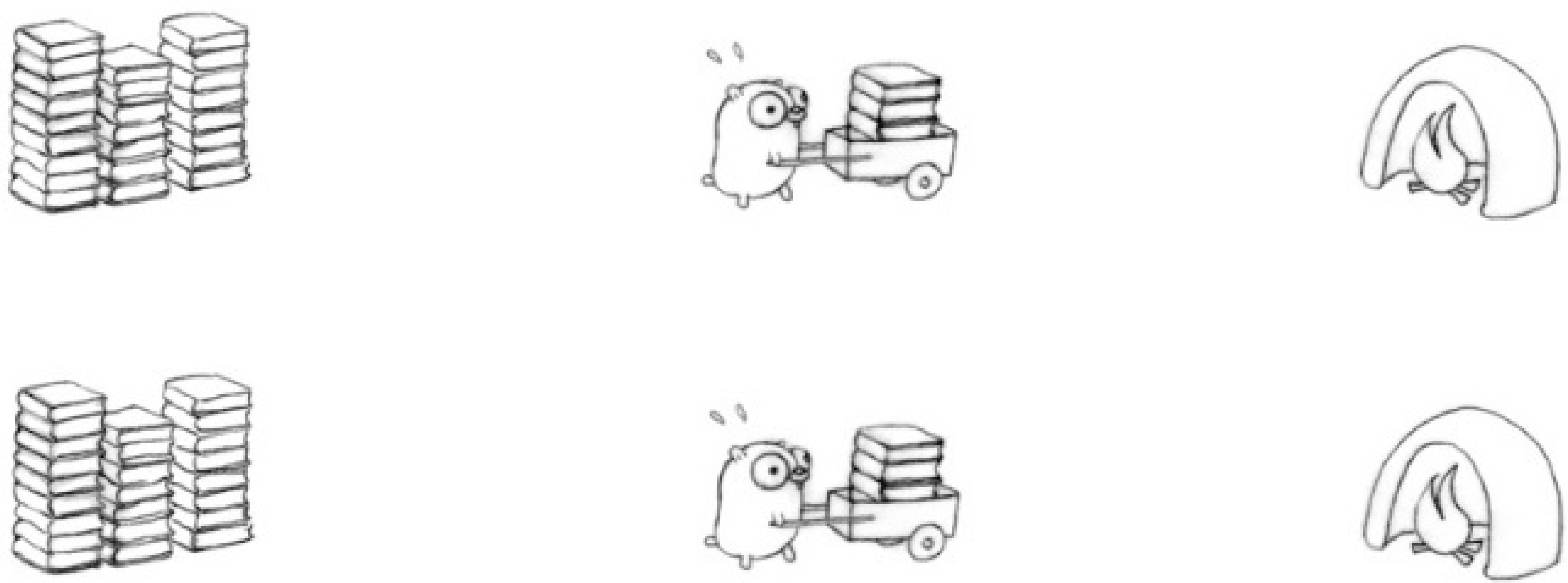
This will go faster, but there will be bottlenecks at the pile and incinerator.
Also need to synchronize the gophers.
A message (that is, a communication between the gophers) will do.

Credits: https://go.dev/blog/waza-talk

This is parallelism

Double everything

Remove the bottleneck; make them really independent.

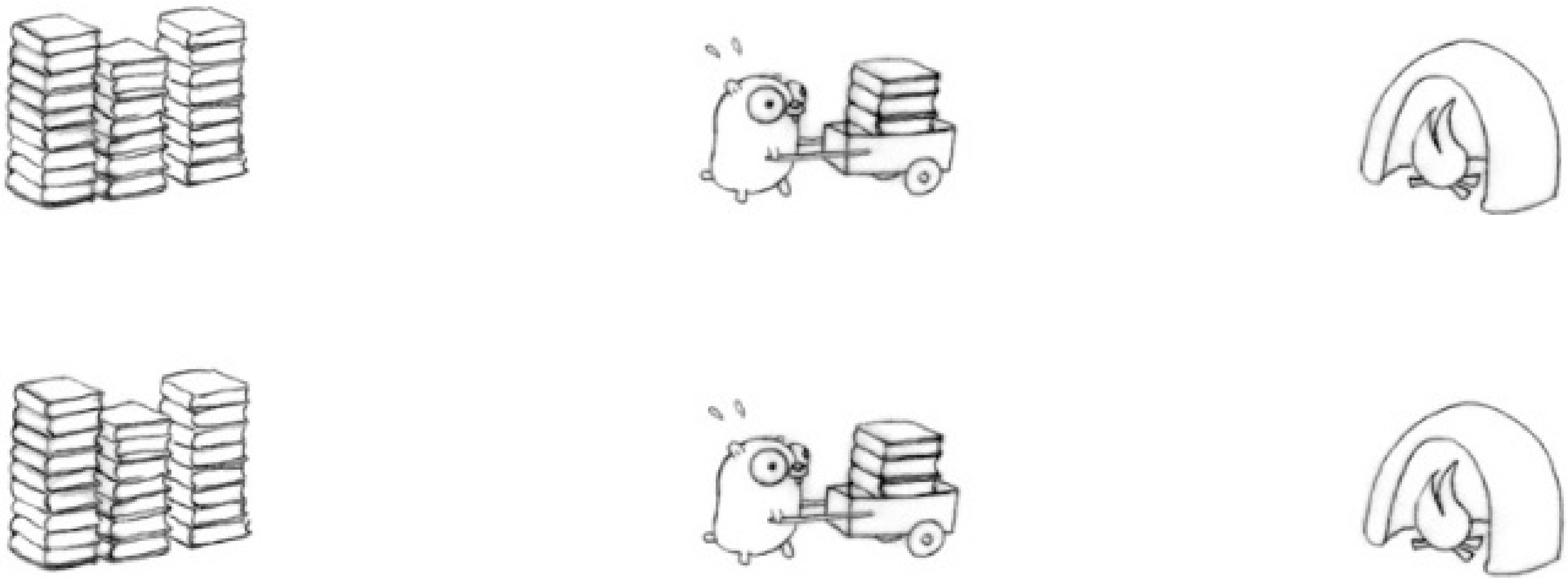


This will consume input twice as fast.

Credits: https://go.dev/blog/waza-talk

A Better Model

Concurrent composition

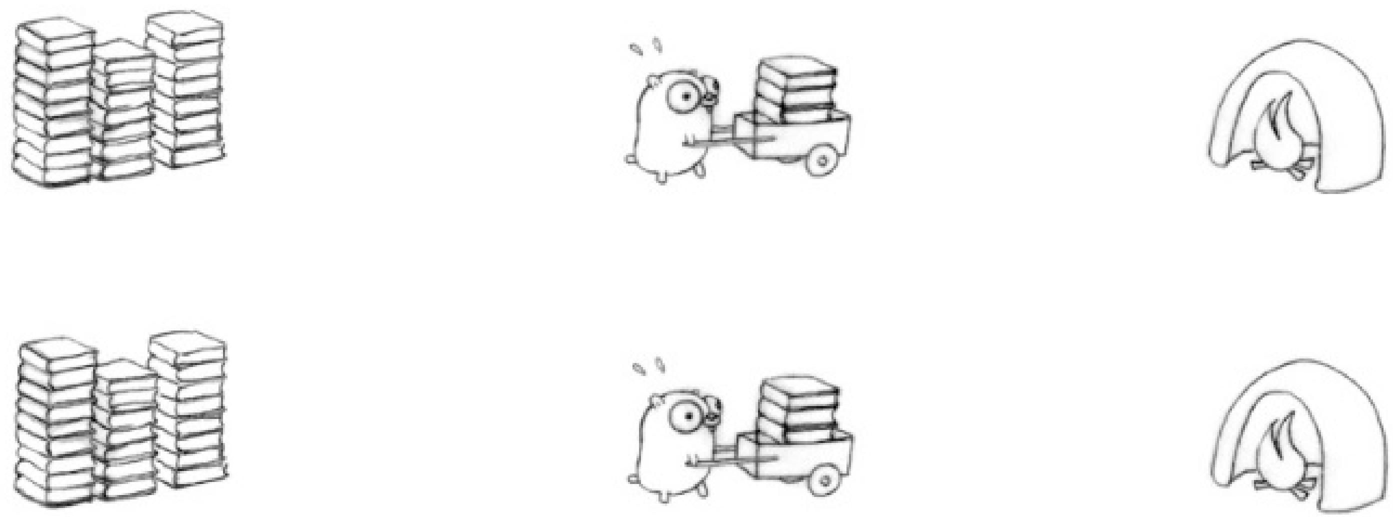


The concurrent composition of two gopher procedures.

Credits: https://go.dev/blog/waza-talk

A Better Model

Concurrent composition



The concurrent composition of two gopher procedures.

Concurrent composition

This design is not automatically parallel!

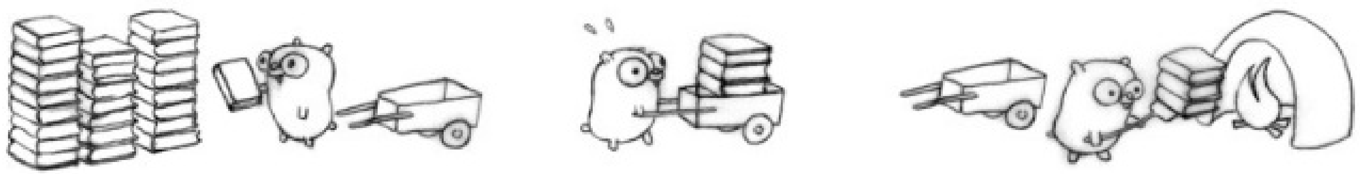
What if only one gopher is moving at a time?
Then it's still concurrent (that's in the design), just not parallel.

However, it's automatically parallelizable!

Moreover the concurrent composition suggests other models.

Credits: https://go.dev/blog/waza-talk

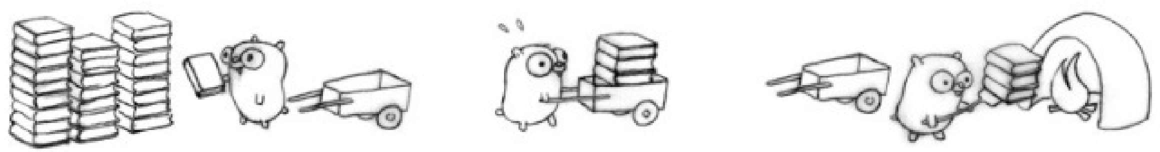
Another design



Three gophers in action, but with likely delays.
Each gopher is an independently executing procedure,
plus coordination (communication).

Credits: https://go.dev/blog/waza-talk

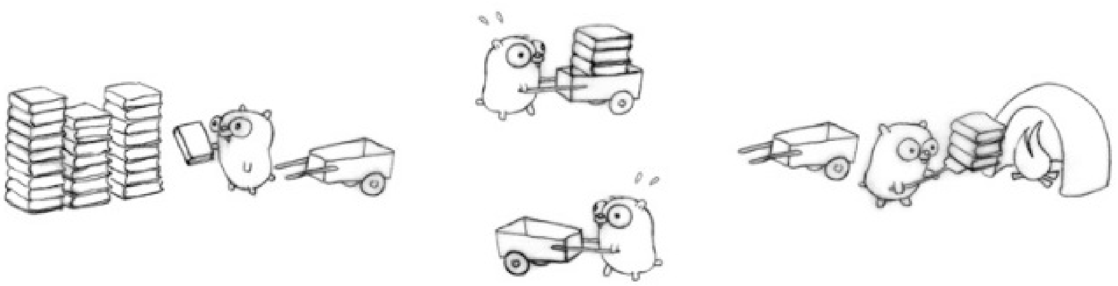
Another design



Three gophers in action, but with likely delays.
Each gopher is an independently executing procedure,
plus coordination (communication).

Finer-grained concurrency

Add another gopher procedure to return the empty carts.

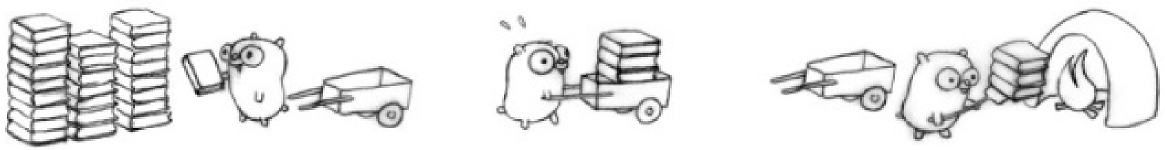


Four gophers in action for better flow, each doing one simple task.

If we arrange everything right (implausible but not impossible), that's four times faster than our original one-gopher design.

Credits: https://go.dev/blog/waza-talk

Another design

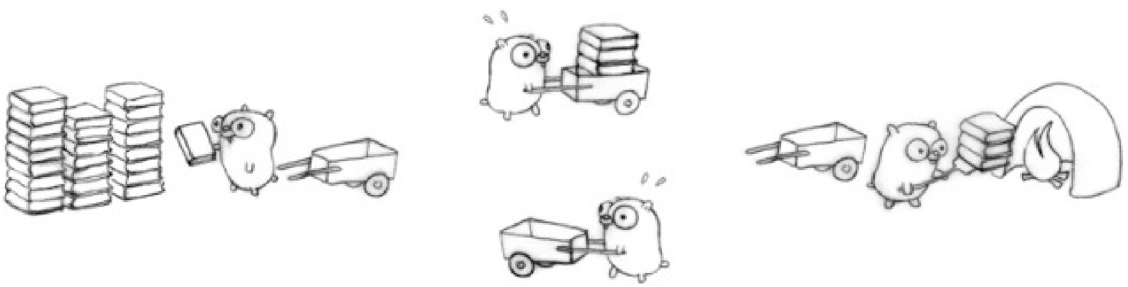


Three gophers in action, but with likely delays.
Each gopher is an independently executing procedure,
plus coordination (communication).

Credits: https://go.dev/blog/waza-talk

Finer-grained concurrency

Add another gopher procedure to return the empty carts.



Four gophers in action for better flow, each doing one simple task.

If we arrange everything right (implausible but not impossible), that's four times faster than our original one-gopher design.

Concurrent procedures

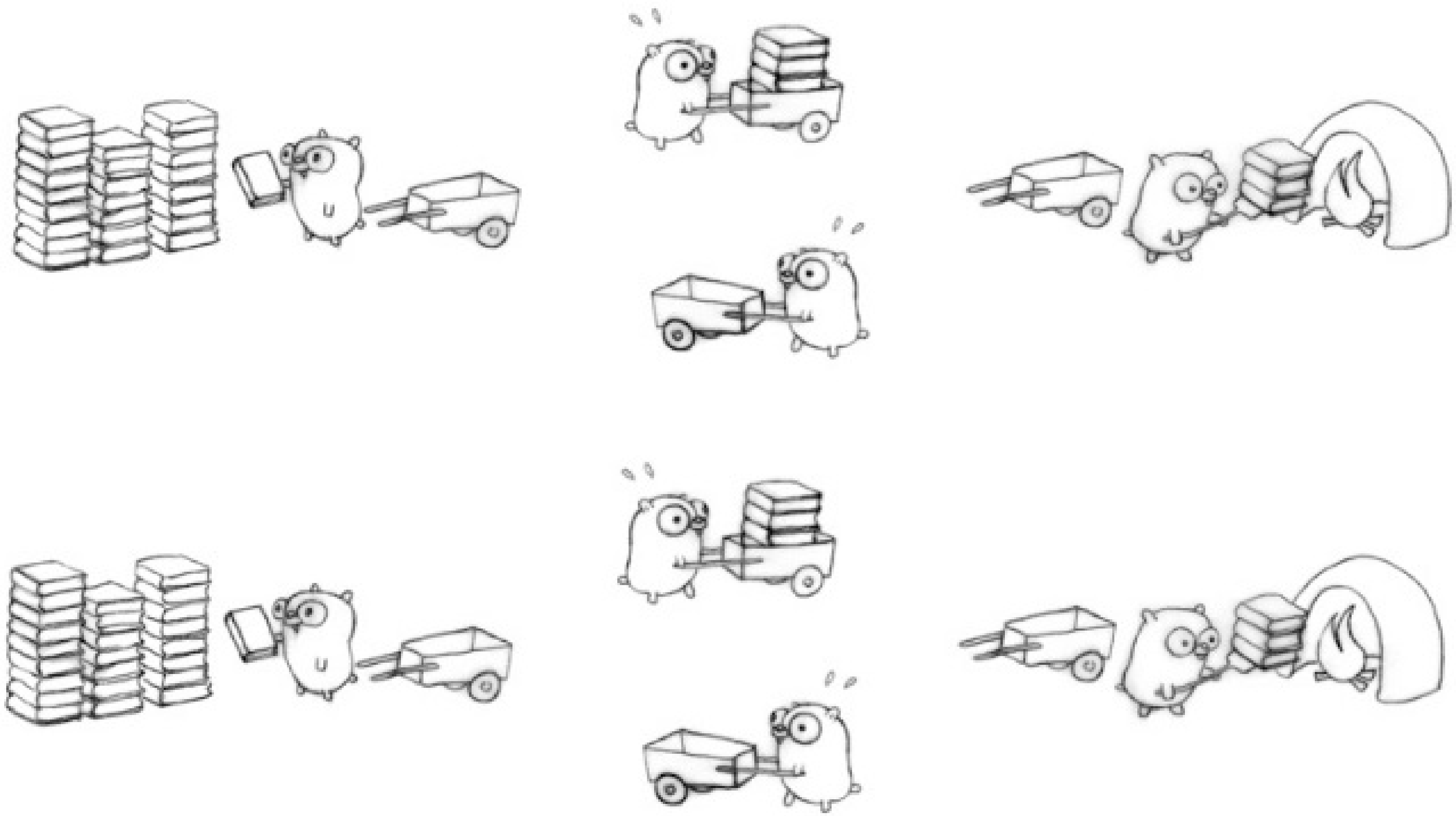
Four distinct gopher procedures:

- load books onto cart
- move cart to incinerator
- unload cart into incinerator
- return empty cart

Different concurrent designs enable different ways to parallelize.

More parallelization!

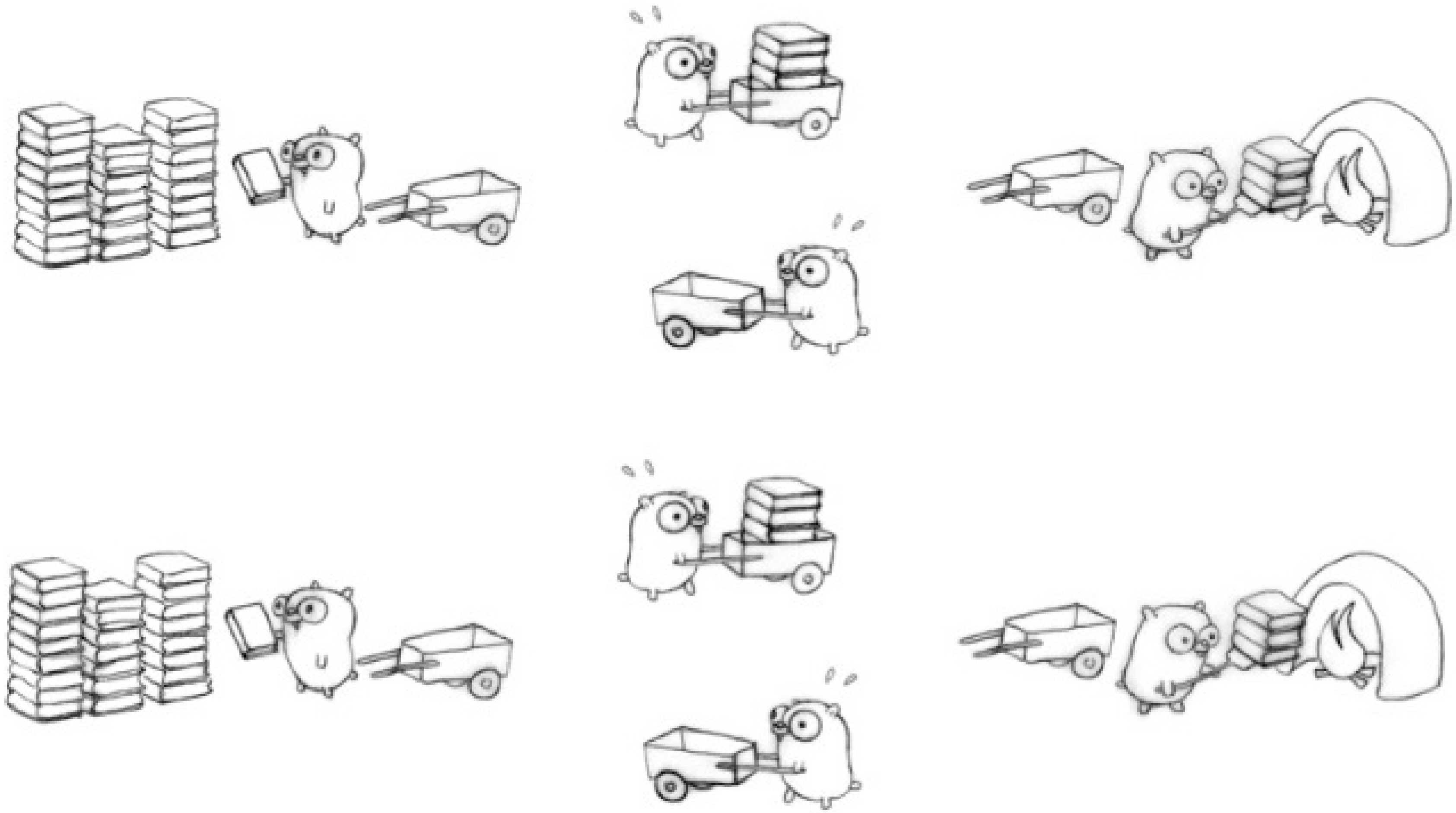
We can now parallelize on the other axis; the concurrent design makes it easy. Eight gophers, all busy.



Credits: https://go.dev/blog/waza-talk

Or maybe no parallelization at all

Keep in mind, even if only one gopher is active at a time (zero parallelism), it's still a correct and concurrent solution.



Credits: https://go.dev/blog/waza-talk

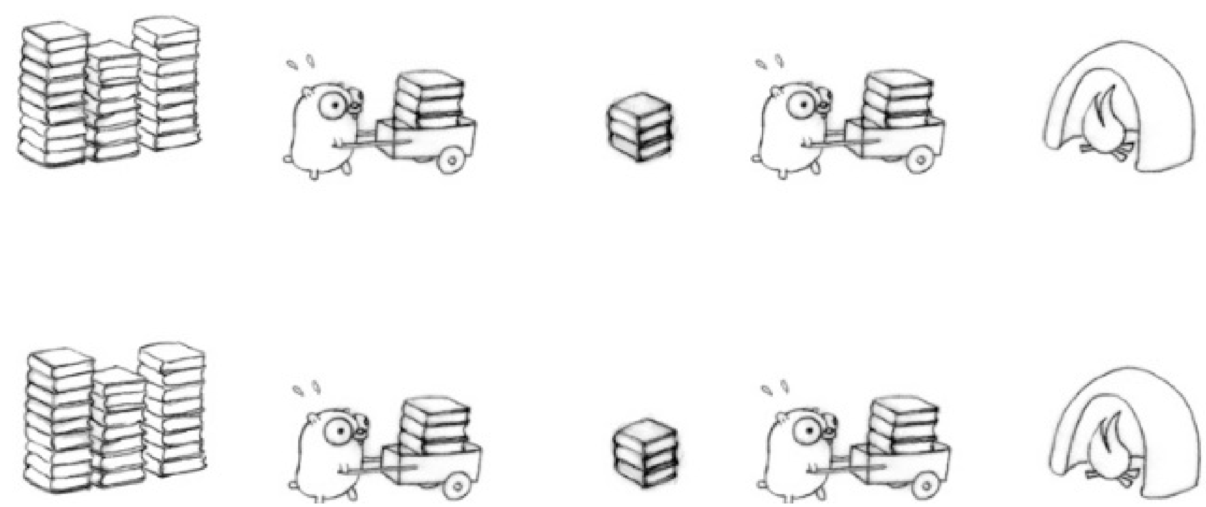
Another design

Here's another way to structure the problem as the concurrent composition of gopher procedures.

Two gopher procedures, plus a staging pile.

Parallelize the usual way

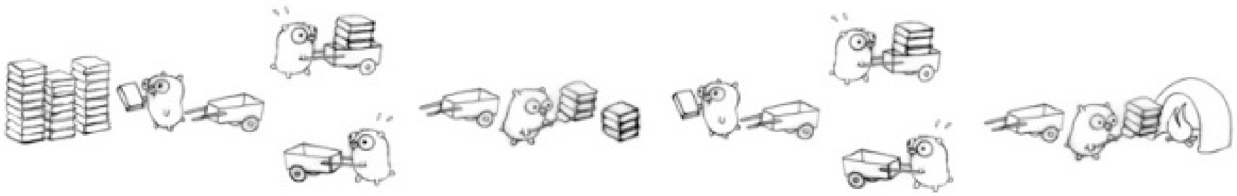
Run more concurrent procedures to get more throughput.



Credits: https://go.dev/blog/waza-talk

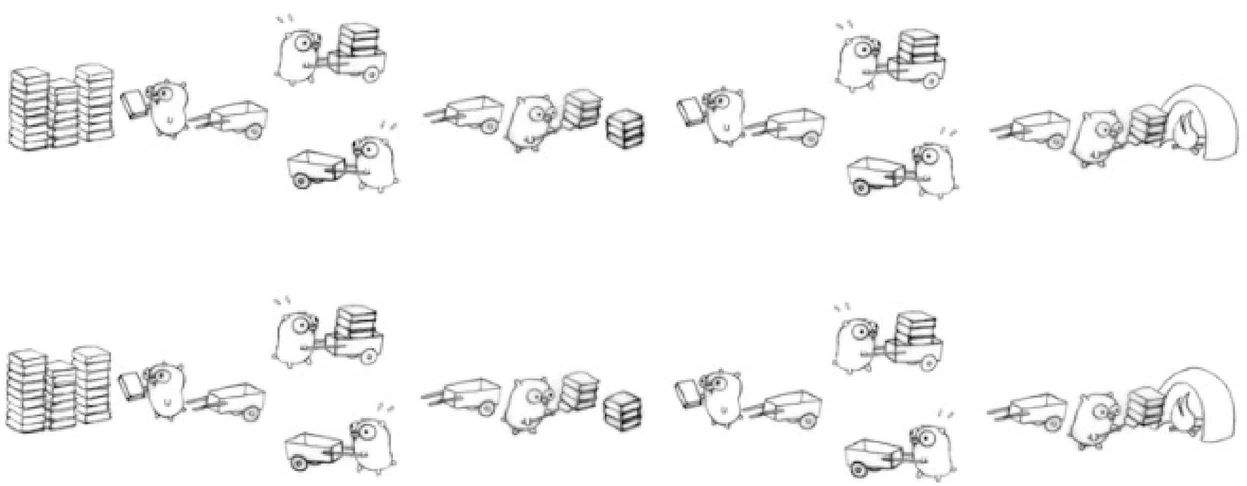
Or a different way

Bring the staging pile to the multi-gopher concurrent model:



Full on optimization

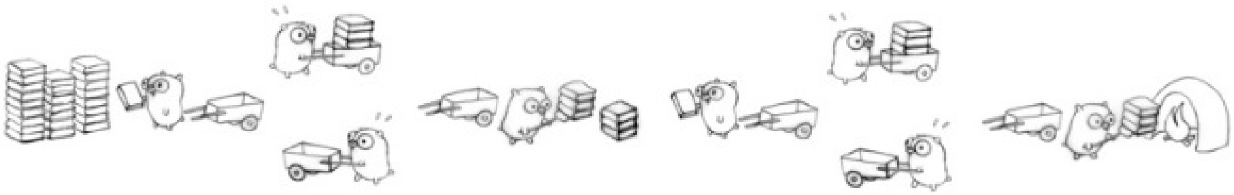
Use all our techniques. Sixteen gophers hard at work!



Credits: https://go.dev/blog/waza-talk

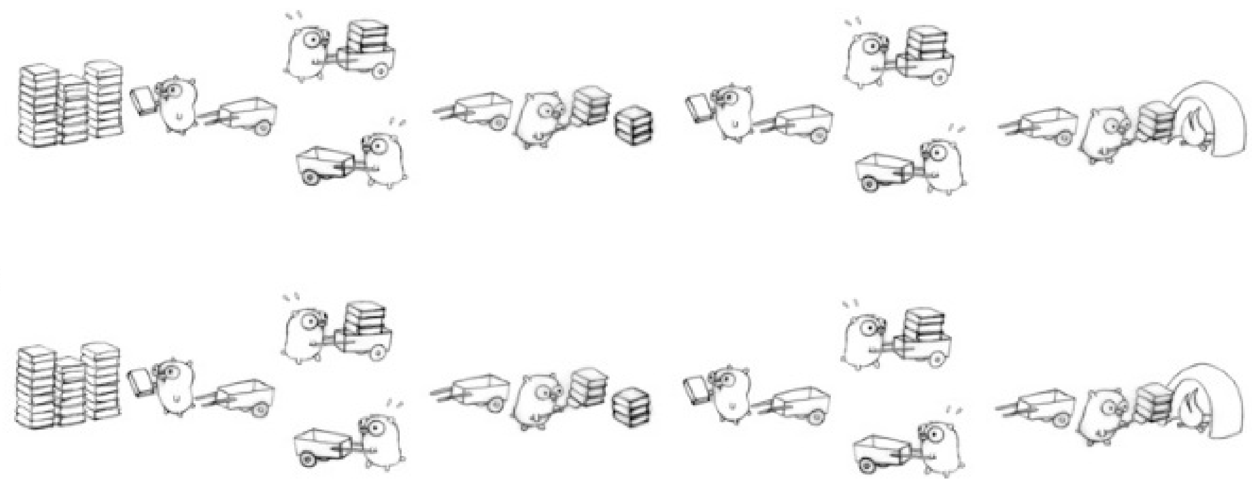
Or a different way

Bring the staging pile to the multi-gopher concurrent model:



Full on optimization

Use all our techniques. Sixteen gophers hard at work!



Lesson

There are many ways to break the processing down.

That's concurrent design.

Once we have the breakdown, parallelization can fall out and correctness is easy.

Credits: https://go.dev/blog/waza-talk

Not Just an Analogy

Back to Computing

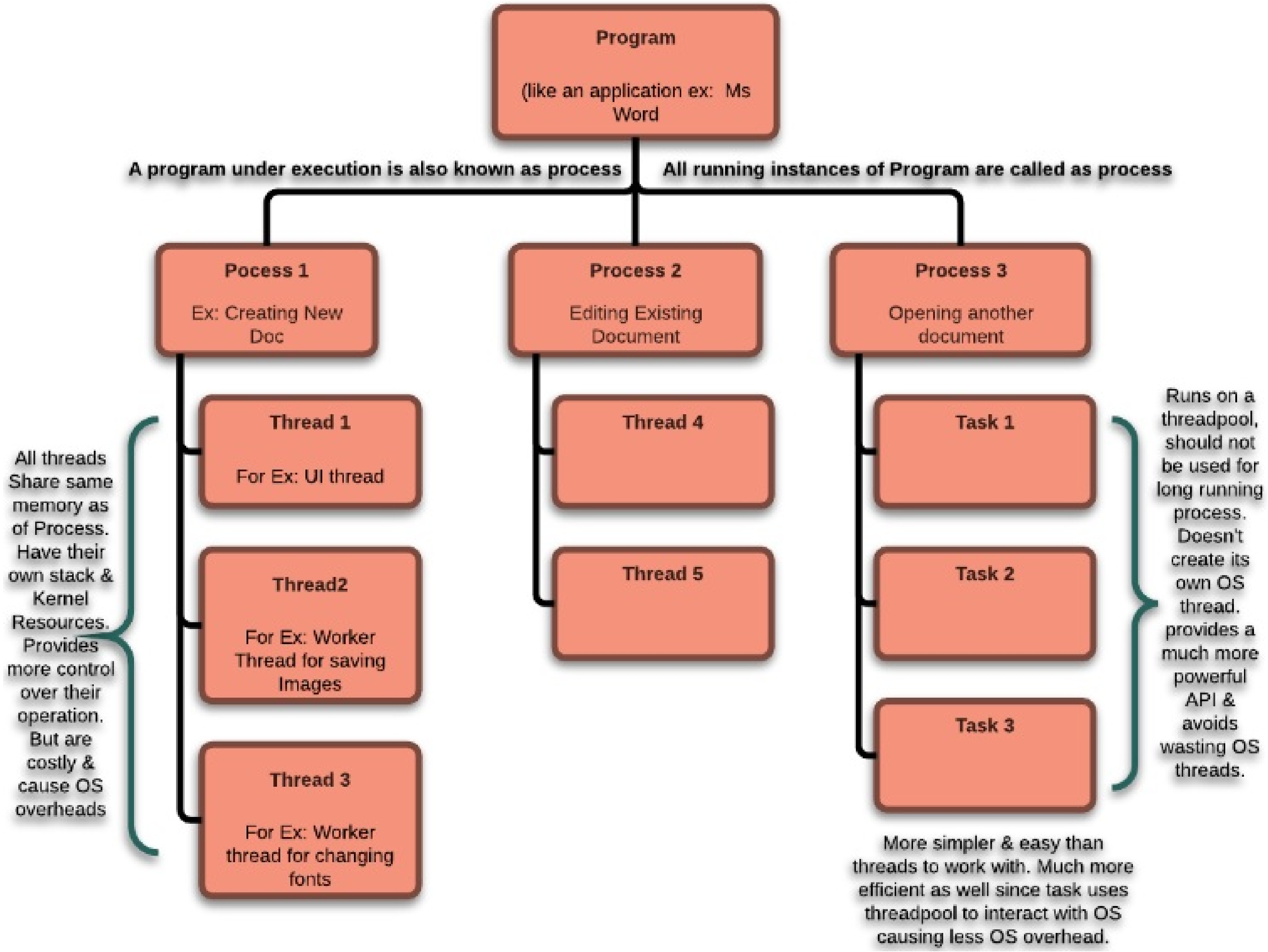
In our book transport problem, substitute:

- book pile => web content
- gopher => CPU
- cart => marshaling, rendering, or networking
- incinerator => proxy, browser, or other consumer

It becomes a concurrent design for a scalable web service.

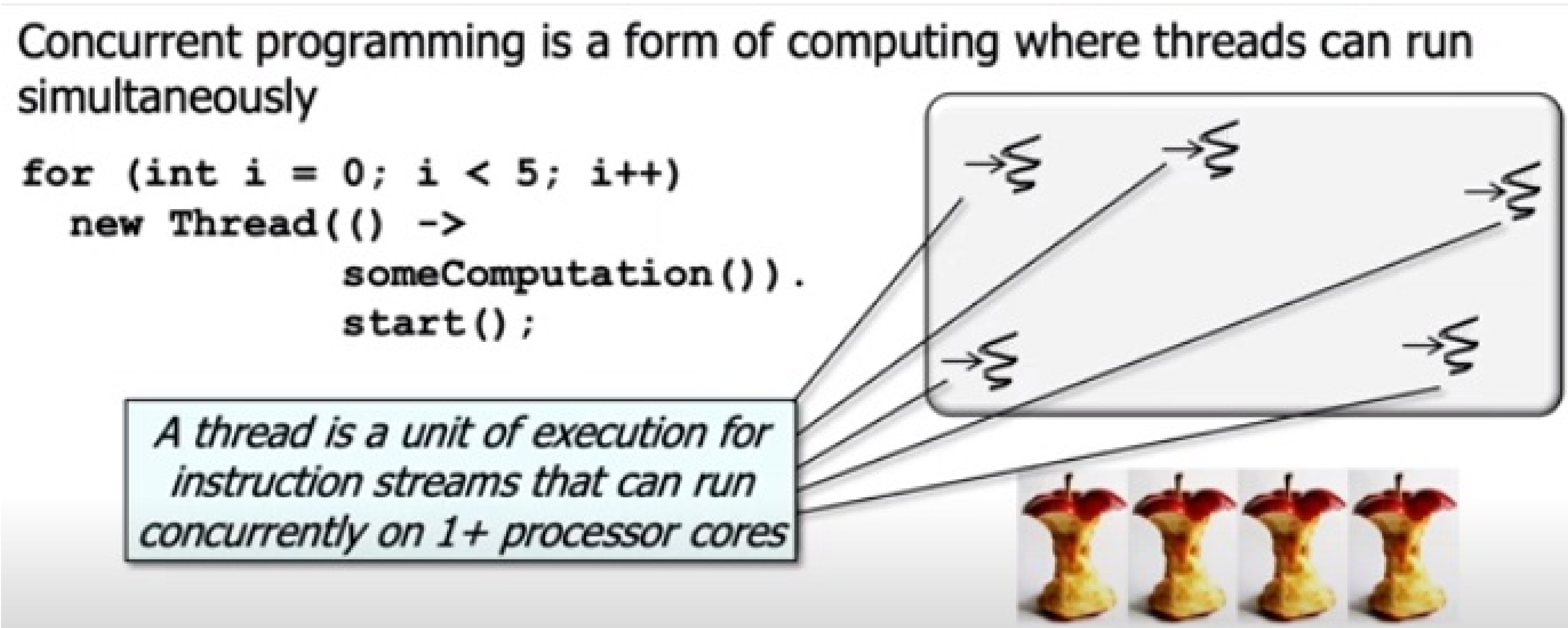
Credits: https://go.dev/blog/waza-talk

Program vs Processes vs Threads



Credits: Jogendra@net Wordpress.com

Threads



Credits: https://www.youtube.com/watch?v=pntLpR6qNnU

Back to Concurrent Haskell

The fundamental action in concurrency is forking a new thread of control. In Concurrent Haskell, this is achieved with the `forkIO` operation:

```
forkIO :: IO () -> IO ThreadId
```

The `forkIO` operation takes a computation of type `IO ()` as its argument; that is, a computation in the `IO` monad that eventually delivers a value of type `()`. The computation passed to `forkIO` is executed in a new thread that runs concurrently with the other threads in the system. If the thread has effects, those effects will be interleaved in an indeterminate fashion with the effects from other threads.

Credits: Parallel and Concurrent Programming in Haskell - Simon Marlow

Interleaving of Threads

fork.hs

```
import Control.Concurrent
import Control.Monad
import System.IO

main = do
  hSetBuffering stdout NoBuffering            -- ❶
  forkIO (replicateM_ 100000 (putChar 'A'))   -- ❷
  replicateM_ 100000 (putChar 'B')            -- ❸
```

❶ Put the output `Handle` into nonbuffered mode, so that we can see the interleaving more clearly.

❷ Create a thread to print the character `A` 100,000 times.

❸ In the main thread, print `B` 100,000 times.

reminders.hs

```haskell
import Control.Concurrent
import Text.Printf
import Control.Monad

main =
  forever $ do
    s <- getLine           -- ❶
    forkIO $ setReminder s -- ❷

setReminder :: String -> IO ()
setReminder s  = do
  let t = read s :: Int
  printf "Ok, I'll remind you in %d seconds\n" t
  threadDelay (10^6 * t)                   -- ❸
  printf "%d seconds is up! BING!\BEL\n" t -- ❹
```

We'll need an operation that waits for some time to elapse:

```haskell
threadDelay :: Int -> IO ()
```

The function `threadDelay` takes an argument representing a number of microseconds and waits for that amount of time before returning.

The program works by creating a thread for each new request for a reminder:

❶ Waits for input from the user.

❷ Creates a new thread to handle this reminder.

❸ The new thread, after printing a confirmation message, waits for the specified number of seconds using `threadDelay`.

❹ Finally, when `threadDelay` returns, the reminder message is printed.

reminders.hs

```haskell
import Control.Concurrent
import Text.Printf
import Control.Monad

main =
  forever $ do
    s <- getLine           -- ❶
    forkIO $ setReminder s -- ❷

setReminder :: String -> IO ()
setReminder s  = do
  let t = read s :: Int
  printf "Ok, I'll remind you in %d seconds\n" t
  threadDelay (10^6 * t)                      -- ❸
  printf "%d seconds is up! BING!\BEL\n" t -- ❹
```

reminders2.hs

```haskell
main = loop
 where
  loop = do
    s <- getLine
    if s == "exit"
       then return ()
       else do forkIO $ setReminder s
               loop
```

MVars - Basic Communication Mechanism

The API for `MVar` is as follows:

```haskell
data MVar a  -- abstract

newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
```

mvar1.hs

```haskell
main = do
  m <- newEmptyMVar
  forkIO $ putMVar m 'x'

  r <- takeMVar m
  print r
```

mvar2.hs

```haskell
main = do
  m <- newEmptyMVar
  forkIO $ do putMVar m 'x'; putMVar m 'y'
  r <- takeMVar m
  print r
  r <- takeMVar m
  print r
```

mvar3.hs

```haskell
main = do
  m <- newEmptyMVar
  takeMVar m
```

MVar as a simple channel - Logging Service

```
data Logger

initLogger :: IO Logger
logMessage :: Logger -> String -> IO ()
logStop    :: Logger -> IO ()
```

MVar as a simple channel - Logging Service

```haskell
data Logger

initLogger :: IO Logger
logMessage :: Logger -> String -> IO ()
logStop    :: Logger -> IO ()
```

logger.hs

```haskell
main :: IO ()
main = do
  l <- initLogger
  logMessage l "hello"
  logMessage l "bye"
  logStop l
```

MVar as a simple channel - Logging Service

```haskell
data Logger

initLogger :: IO Logger
logMessage :: Logger -> String -> IO ()
logStop    :: Logger -> IO ()
```

logger.hs

```haskell
main :: IO ()
main = do
  l <- initLogger
  logMessage l "hello"
  logMessage l "bye"
  logStop l
```

logger.hs

```haskell
data Logger = Logger (MVar LogCommand)

data LogCommand = Message String | Stop (MVar ())
```

The Details

```haskell
initLogger :: IO Logger
initLogger = do
  m <- newEmptyMVar
  let l = Logger m
  forkIO (logger l)
  return l
```

The Details

```haskell
initLogger :: IO Logger
initLogger = do
  m <- newEmptyMVar
  let l = Logger m
  forkIO (logger l)
  return l
```

```haskell
logger :: Logger -> IO ()
logger (Logger m) = loop
 where
  loop = do
    cmd <- takeMVar m
    case cmd of
      Message msg -> do
        putStrLn msg
        loop
      Stop s -> do
      putStrLn "logger: stop"
      putMVar s ()
```

MVar as a simple channel - Logging Service

```haskell
data Logger

initLogger :: IO Logger
logMessage :: Logger -> String -> IO ()
logStop    :: Logger -> IO ()
```

logger.hs

```haskell
main :: IO ()
main = do
  l <- initLogger
  logMessage l "hello"
  logMessage l "bye"
  logStop l
```

logger.hs

```haskell
data Logger = Logger (MVar LogCommand)

data LogCommand = Message String | Stop (MVar ())
```

The Details

```
logMessage :: Logger -> String -> IO ()
logMessage (Logger m) s = putMVar m (Message s)
```

The Details

```
logMessage :: Logger -> String -> IO ()
logMessage (Logger m) s = putMVar m (Message s)
```

```
logStop :: Logger -> IO ()
logStop (Logger m) = do
  s <- newEmptyMVar
  putMVar m (Stop s)
  takeMVar s
```

The Details

```haskell
logMessage :: Logger -> String -> IO ()
logMessage (Logger m) s = putMVar m (Message s)
```

```haskell
logStop :: Logger -> IO ()
logStop (Logger m) = do
  s <- newEmptyMVar
  putMVar m (Stop s)
  takeMVar s
```

MVar as a container for shared state

```
type Name        = String
type PhoneNumber = String
type PhoneBook   = Map Name PhoneNumber

newtype PhoneBookState = PhoneBookState (MVar PhoneBook)
```

MVar as a container for shared state

```
type Name        = String
type PhoneNumber = String
type PhoneBook   = Map Name PhoneNumber

newtype PhoneBookState = PhoneBookState (MVar PhoneBook)
```

```
new :: IO PhoneBookState
new = do
  m <- newMVar Map.empty
  return (PhoneBookState m)
```

MVar as a container for shared state

```haskell
type Name        = String
type PhoneNumber = String
type PhoneBook   = Map Name PhoneNumber

newtype PhoneBookState = PhoneBookState (MVar PhoneBook)
```

```haskell
new :: IO PhoneBookState
new = do
  m <- newMVar Map.empty
  return (PhoneBookState m)
```

```haskell
insert :: PhoneBookState -> Name -> PhoneNumber -> IO ()
insert (PhoneBookState m) name number = do
  book <- takeMVar m
  putMVar m (Map.insert name number book)
```

MVar as a container for shared state

```haskell
type Name        = String
type PhoneNumber = String
type PhoneBook   = Map Name PhoneNumber

newtype PhoneBookState = PhoneBookState (MVar PhoneBook)
```

```haskell
new :: IO PhoneBookState
new = do
  m <- newMVar Map.empty
  return (PhoneBookState m)
```

```haskell
insert :: PhoneBookState -> Name -> PhoneNumber -> IO ()
insert (PhoneBookState m) name number = do
  book <- takeMVar m
  putMVar m (Map.insert name number book)
```

```haskell
lookup :: PhoneBookState -> Name -> IO (Maybe PhoneNumber)
lookup (PhoneBookState m) name = do
  book <- takeMVar m
  putMVar m book
  return (Map.lookup name book)
```

MVar as a container for shared state

```haskell
type Name        = String
type PhoneNumber = String
type PhoneBook   = Map Name PhoneNumber
```

```haskell
main = do
  s <- new
  sequence_ [ insert s ("name" ++ show n) (show n) | n <- [1..10000] ]
  lookup s "name999" >>= print
  lookup s "unknown" >>= print
```

```haskell
newtype PhoneBookState = PhoneBookState (MVar PhoneBook)
```

```haskell
new :: IO PhoneBookState
new = do
  m <- newMVar Map.empty
  return (PhoneBookState m)
```

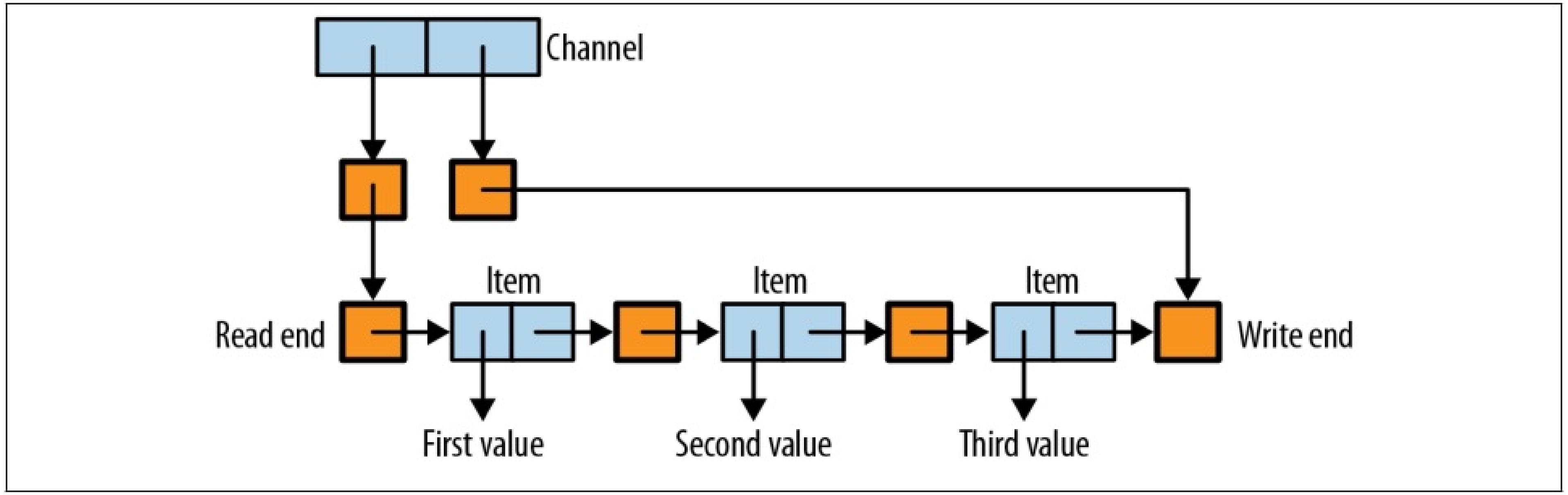
```haskell
insert :: PhoneBookState -> Name -> PhoneNumber -> IO ()
insert (PhoneBookState m) name number = do
  book <- takeMVar m
  putMVar m (Map.insert name number book)
```

```haskell
lookup :: PhoneBookState -> Name -> IO (Maybe PhoneNumber)
lookup (PhoneBookState m) name = do
  book <- takeMVar m
  putMVar m book
  return (Map.lookup name book)
```

MVar as a building block

```
data Chan a

newChan   :: IO (Chan a)
readChan  :: Chan a -> IO a
writeChan :: Chan a -> a -> IO ()
```

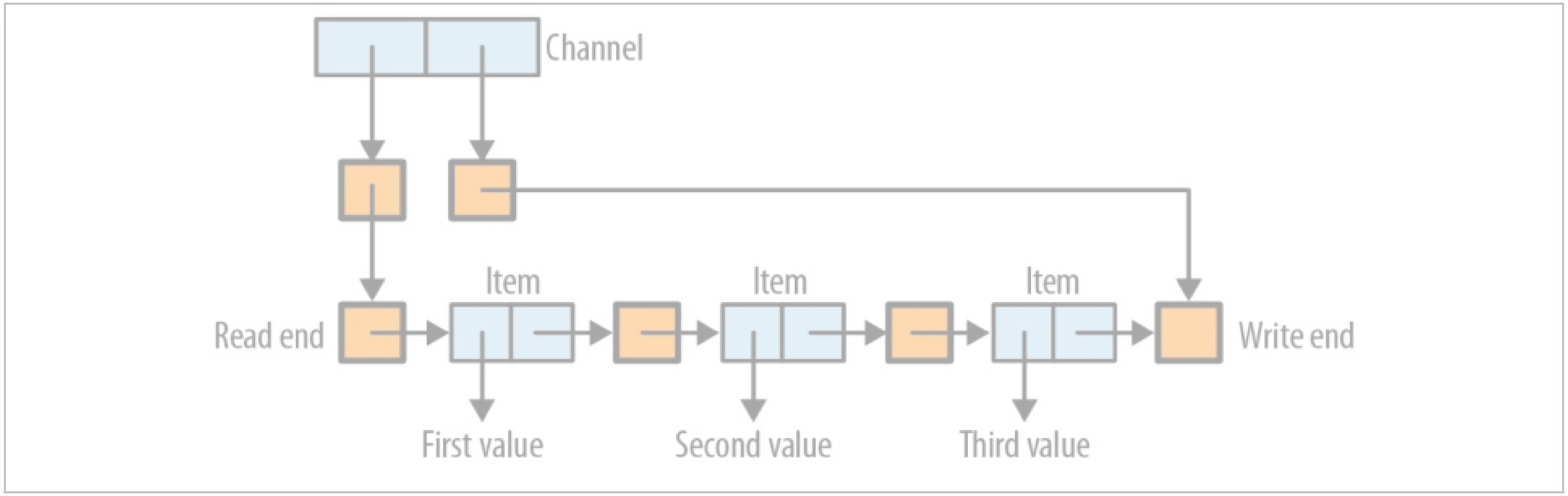


MVar as a building block

```
data Chan a

newChan   :: IO (Chan a)
readChan  :: Chan a -> IO a
writeChan :: Chan a -> a -> IO ()
```

```
type Stream a = MVar (Item a)
data Item a   = Item a (Stream a)
```



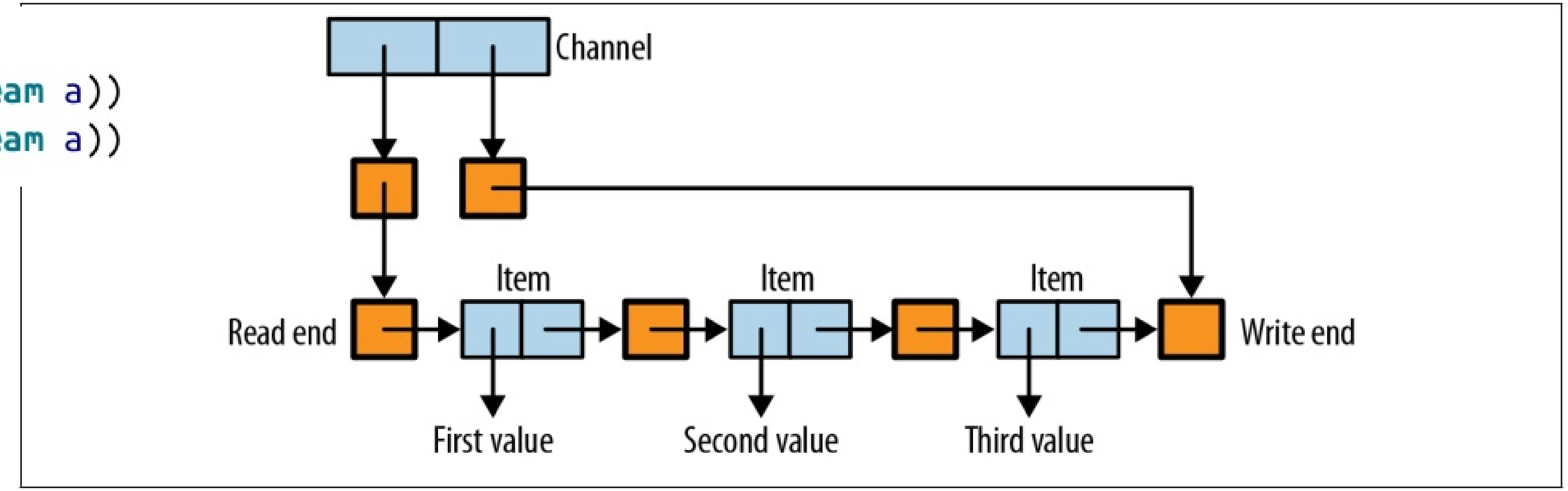
MVar as a building block

```haskell
data Chan a
```

```haskell
type Stream a = MVar (Item a)
data Item a   = Item a (Stream a)
```

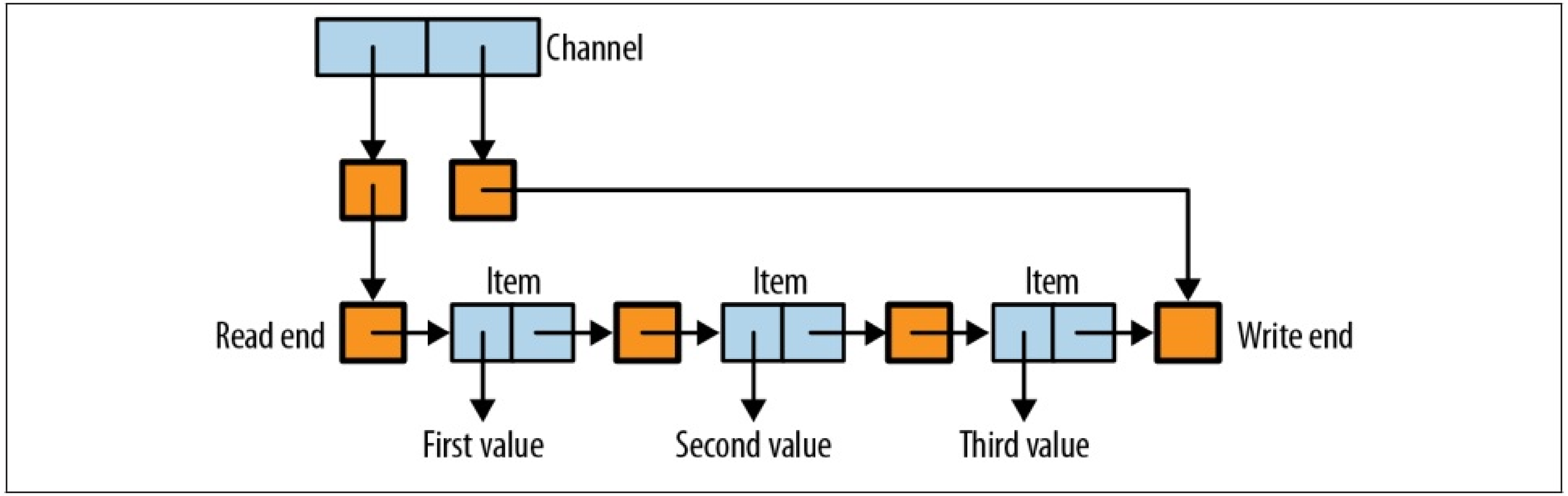
```haskell
newChan   :: IO (Chan a)
readChan  :: Chan a -> IO a
writeChan :: Chan a -> a -> IO ()
```

```haskell
data Chan a
 = Chan (MVar (Stream a))
        (MVar (Stream a))
```



MVar as a building block

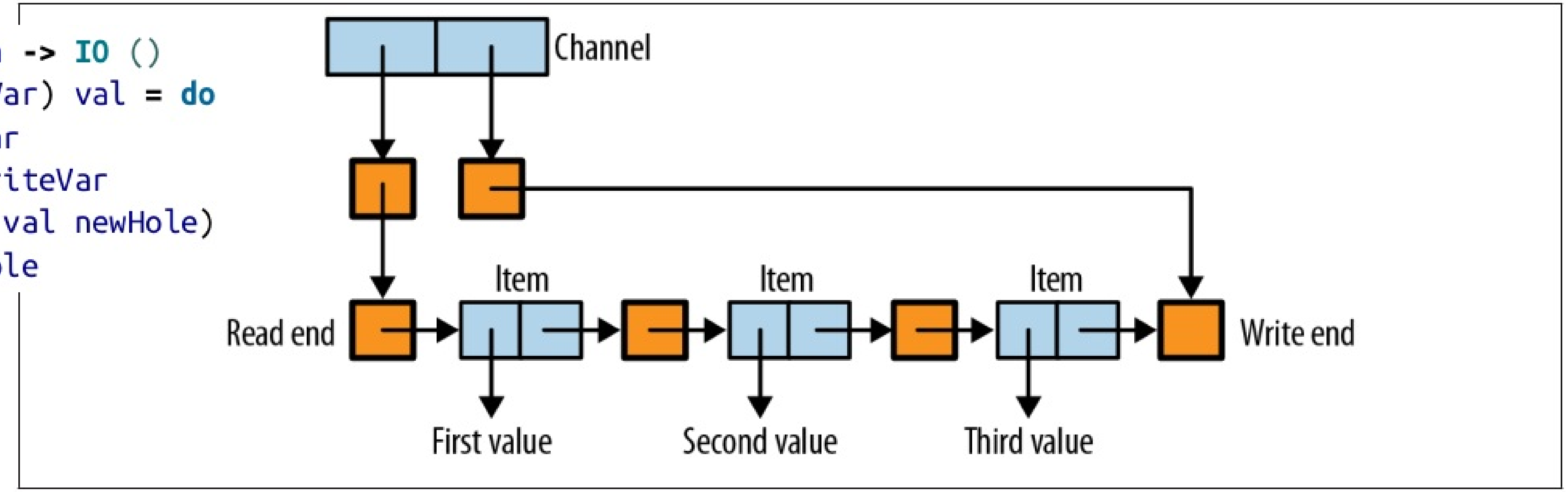
```
newChan :: IO (Chan a)
newChan = do
  hole   <- newEmptyMVar
  readVar  <- newMVar hole
  writeVar <- newMVar hole
  return (Chan readVar writeVar)
```



MVar as a building block

```haskell
newChan :: IO (Chan a)
newChan = do
  hole  <- newEmptyMVar
  readVar  <- newMVar hole
  writeVar <- newMVar hole
  return (Chan readVar writeVar)
```

```haskell
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
  newHole <- newEmptyMVar
  oldHole <- takeMVar writeVar
  putMVar oldHole (Item val newHole)
  putMVar writeVar newHole
```



MVar as a building block

```haskell
newChan :: IO (Chan a)
newChan = do
  hole   <- newEmptyMVar
  readVar  <- newMVar hole
  writeVar <- newMVar hole
  return (Chan readVar writeVar)
```

```haskell
writeChan :: Chan a -> a -> IO ()
writeChan (Chan _ writeVar) val = do
  newHole <- newEmptyMVar
  oldHole <- takeMVar writeVar
  putMVar oldHole (Item val newHole)
  putMVar writeVar newHole
```

```haskell
readChan :: Chan a -> IO a
readChan (Chan readVar _) = do
  stream <- takeMVar readVar              -- ❶
  Item val tail <- takeMVar stream        -- ❷
  putMVar readVar tail                    -- ❸
  return val
```

Consider what happens if the channel is empty. The first `takeMVar` (❶) will succeed, but the second `takeMVar` (❷) will find an empty hole, and so will block. When another thread calls `writeChan`, it will fill the hole, allowing the first thread to complete its `takeMVar`, update the read end (❸) and finally return.