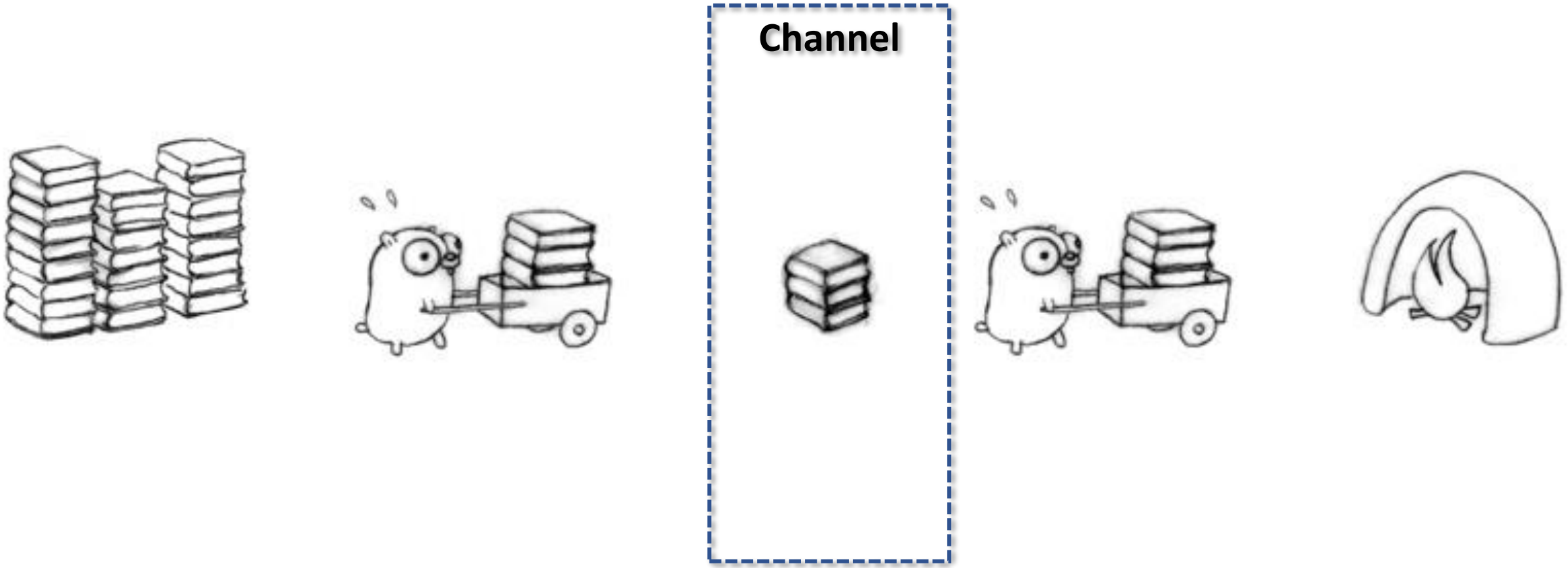
Go Programming Tutorial

CS 582 Distributed Systems, Fall 2018

Lahore University of Management Sciences (LUMS)

Delivered on Friday 14th September, 2018

Introduction to channels, 0 of 3



Introduction to channels, 1 of 3

- Syntax
  - Create a channel `c := make(chan int)`
  - Send a value `c <- 5`
  - Receive a value `a := <-c`
- Safe for concurrent use
- Blocking operation
  - Send blocks until a receiver is ready
  - Receive blocks until a sender is ready

Introduction to channels, 2 of 3

- Buffered channel
  - Create a channel `c := make(chan int, 10)`
  - Send blocks if buffer is full, i.e. `c` contains `10` integers already
  - Receive blocks if buffer is empty
- Channel can be closed to indicate no more values will be sent.
  - `close(c)`
  - Idiom used for channels that will be closed:

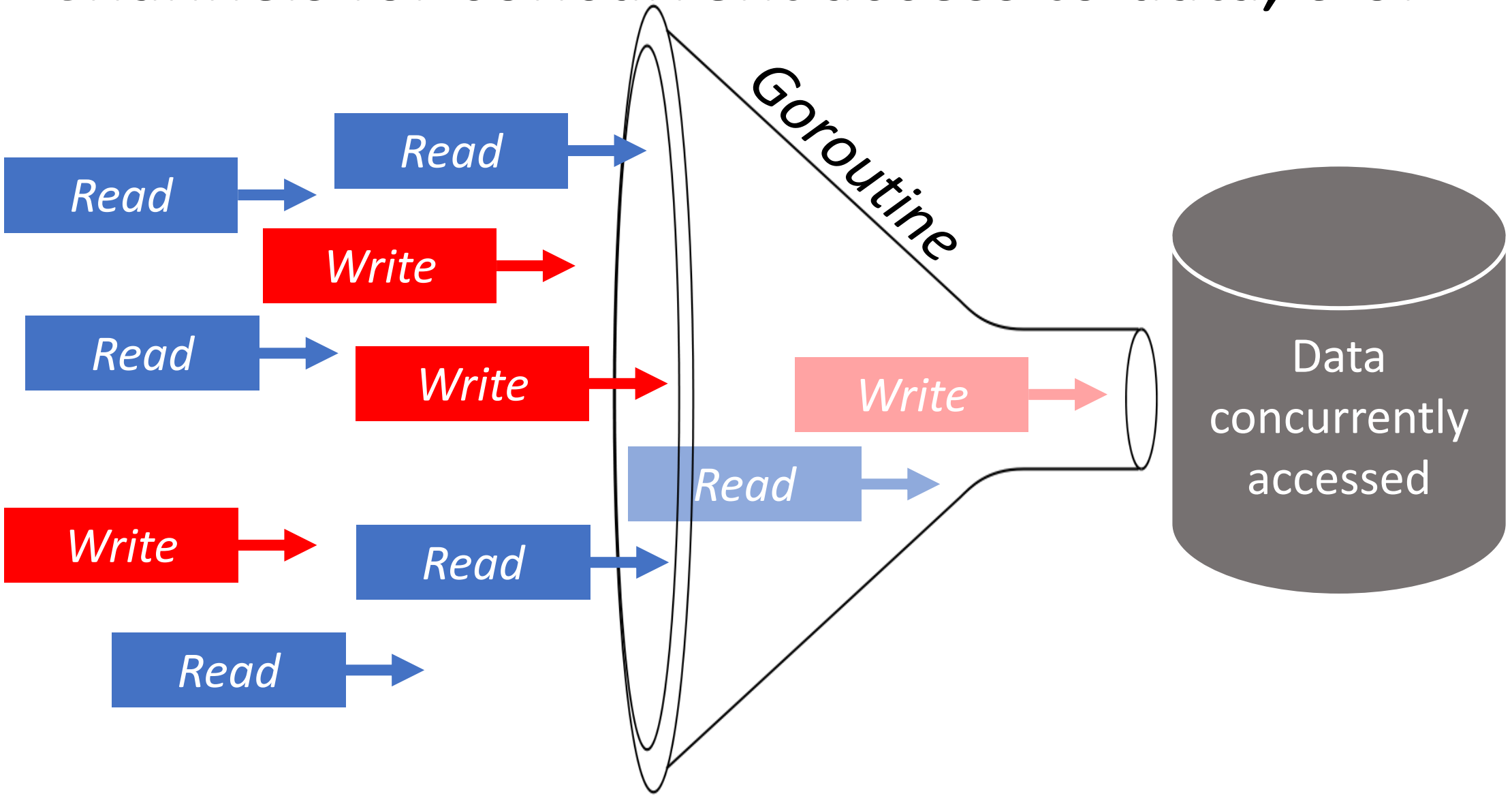
```
for val := range c {
    <code>
}
```

Introduction to channels, 3 of 3

- Use `select` clause to process multiple channels.

```
for {
        select {
        case a := <-request:
                <code>
        case <-stopSignal:
                return
        }
}
```

Channels for concurrent access to data, 0 of 2



Channels for concurrent access to data, 1 of 2

- Channel-based approach to concurrency is preferred in Go.
  - The idea is to share memory by communicating.
  - Each piece of data owned by exactly 1 goroutine.
- One channel `chan readFromDb` to send read commands:

```
type readFromDb struct {
      key         int
      …
}
```

Channels for concurrent access to data, 2 of 2

- One channel `chan writeToDb` to send write commands:

```
type writeToDb struct {
      key          int
      value        string
      …
}
```

- Exactly **ONE** goroutine will read to and write from the database.

Channels are first-class objects in Go, 1 of 2

- Channels can be passed around as other objects.
- Channels can be passed into functions as arguments.
  - Full access: `func print(request chan string) { …`
  - Receive-only: `func print(request <-chan string) { …`
  - Send-only: `func worker(task string, output chan<- int) { …`
- Channels can be returned from functions. Such functions are called generators. E.g., the following return receive-only channels:
  - `time.After(2 * time.Second)` and `time.Tick(time.Minute)`

Channels are first-class objects in Go, 2 of 2

- Channels can be part of other structs.
  - E.g. Tasks sent to a worker goroutine can include a channel. The worker can process the task and send the result on this channel directly to the client. The server coordinating clients and workers would not need to be involved.

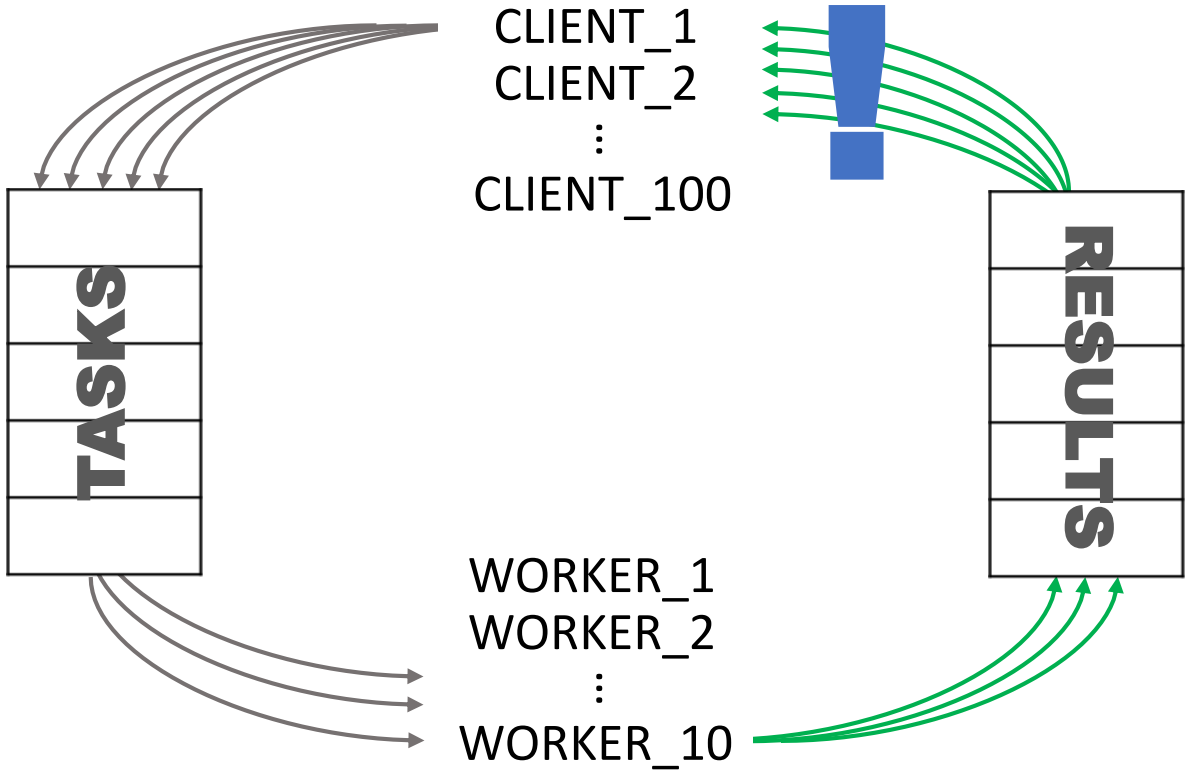
```
type squareRoot struct {
      task        int
      client      chan int
      …
}
```

Generators, 1 of 1

- Fan-in: functions can be used to combine / multiplex channels.
  - See function `combineIntChans` in *gen.go*.
- Fan-out: functions can be used to demultiplex channels.
- Broadcast: functions can be used to replicate an object on many channels.
- Note: must compare `len(c)` with `cap(c)` to see if sending into or receiving from a channel will be a blocking operation.

Advanced examples, 1 of 2

- Worker pools
  - See *workers.go*.
  - Handle clients concurrently.
  - Distribute load over workers.
  - Requires a goroutine to send back the results to the clients.



Advanced examples, 2 of 2

- Simple active load balancing
  - Keep workers in a min-heap using a measure of the load as the score.
  - When a new job comes, extract least loaded worker from the heap, give it the job, update the score and push it back in the heap.
  - When a worker finishes a job, remove worker from the heap, update its score and put it back into the heap.
- Synchronizing workers
  - Workers wait for a signal channel on a “wait” channel before sending data.
  - Each worker must have their own “wait” channel.

Avoiding common bugs, 1 of 2

- Use `:=` to declare and initialize; use `=` for a variable already declared
- Getting naming conventions wrong:
  - Use `_` for variable assigned but not used
  - Methods constants can be exported only if their names begin with a capital letter
- Type comes after variable name, i.e. `<var> <type>`
- Use parentheses when doing multiple dereferences. E.g., `*<p1>.<p2>` can mean `(*<p1>).<p2>` or `*(<p1>.<p2>)`.

Avoiding common bugs, 2 of 2

- We get a new slice with `append(<slice>, <values>)` not the same slice getting mutated. Don't forget to reassign it.
- Use `val, ok := <map>[<k>]` to get a value from `<map>` for key `<k>`.
  - `ok` will be `false` if `<k>` is not in `<map>`. We shouldn't use `val` in this case. It will be defaulted to a zero-valued literal or struct of its type.
  - `ok` will be `true` if `<k>` is in `<map>`. You may use `val` in this case.

Socket programming – client-side, 1 of 3

- To connect, use `net.Dial("tcp", "localhost:9999")`. Remember it returns a `net.Conn` and an `error` interface object.
- Use `bufio.NewReadWriter( bufio.NewReader(conn), bufio.NewWriter(conn) )` to get a reader / writer `<rw>` on the connection.
- Use `<rw>.ReadString(delim)` to read and `<rw>.WriteString(msg)` together with `<rw>.Flush()` to write.
- Do complete error-checking and close the connection when done.

Socket programming – server-side, 2 of 3

- To listen, use `net.Listen("tcp", ":9999")`. Remember it returns a `net.Listener <ln>` and an `error` interface object.
- Use `<ln>.Accept()` to accept a connection.
- Use a separate goroutine to handle that connection concurrently.
- The connection can be read from or written to in the same way as that on the client-side.
- Do complete error-checking and close the connection when done.

Socket programming – example, 3 of 3

- A simple echo server has been implemented in *echo_server.go*. It echoes back to the client their own messages. Do see how the server creates a listening socket, accepts and handles connections, does proper error-checking and closes the connection at the end.
- A simple echo client has also been implemented in *echo_client.go*. You can use it to interact with the server using command line.
  - Note: no way was implemented to cleanly close the client.
  - Note: upon exit, the client doesn't properly close the socket using `<conn>.Close()` in this program. But you should not skip this step.