Synchronization of goroutines

This lesson shows how closing or blocking a pattern, brings effective integration between goroutines.

WE'LL COVER THE FOLLOWING

- Closing a channel
  - Signaling
  - Detection
- Blocking and the producer-consumer pattern

Channels can be closed explicitly. However, they are not like files. You don't usually need to close them. Closing a channel is only necessary when the receiver must be told that there are no more values coming. Only the sender should close a channel, never the receiver.

Closing a channel

The question is, how can we signal when `sendData()` is done with the channel, and how can `getData()` detect that the channel whether closed or blocked?

Signaling

This is done with the function `close(ch)`. This marks the channel as unable to accept more values through a send operation `<-`. Sending to or closing a closed channel causes a run-time panic. It is good practice to do this with a `defer` statement immediately after the making of the channel (when it is appropriate in the given situation):

```
ch := make(chan float64)
defer close(ch)
```

Detection

This is done with the `comma, ok` operator. This tests whether the channel is closed and then returns true, otherwise false. How can we test that we can

receive without blocking (or that channel `ch` is not closed)?

```
v, ok := <-ch // ok is true if v received a value
```

Often, this is used together with an if-statement:

```
if v, ok := <-ch; ok {
  ...
}
```

Or, when the receiving happens in a for loop, use `break` when `ch` is closed or blocked:

```
v, ok := <-ch
if !ok {
  break
}
// process(v)
```

We can trigger the behavior of a non-blocking send by writing: `_ = ch <- v` because the blank identifier takes whatever is sent on `ch`.

```
package main
import "fmt"

func main() {
  ch := make(chan string)
  go sendData(ch)
  getData(ch)
}

func sendData(ch chan string) {
  ch <- "Washington"
  ch <- "Tripoli"
  ch <- "London"
  ch <- "Beijing"
  ch <- "Tokyo"
  close(ch)
}

func getData(ch chan string) {
  for {
    input, open := <-ch
    if !open {
      break
    }
    fmt.Printf("%s ", input)
  }
}
```

Synchronization

In the code above, at **line 5**, we make a channel `ch` of strings. **Line 6** starts a goroutine with function `sendData()`, passing `ch` as a parameter. From **line 11** to **line 15**, *five* values are put on the channel, which is then closed. At **line** 7, in the `main()` routine, a function `getData()` is called, which is defined from **line 19** to **line** 27. This is, in fact, an infinite for-loop, which takes a value from the channel at **line 21** as long as it is open and prints it out. If closed, we break from the loop at **line 23**.

Here is what is changed in the code:

- Only `sendData()` is now a goroutine (see **line 6**), and `getData()` runs in the same thread as `main()` (see **line** 7).
- At the end of the function `sendData()`, the channel is closed (see **line 16**).
- In the for-loop in `getData()`, after every receive the channel is tested with `if !open` (see **line 22**).

It is even better to practice reading the channel with a for-range statement because this will automatically detect when the channel is closed:

```
for input := range ch {
  Process(input)
}
```

Blocking and the producer-consumer pattern

In this pattern, the relationship between the two goroutines is such that one is usually blocking the other. If the program runs on a multicore machine; only one processor will be employed most of the time. This can be improved by using a channel with a buffer size, greater than 0. For example, with a buffer size of 100, the iterator can produce at least 100 items from the container before blocking. If the consumer goroutine is running on a separate processor, it is possible that neither goroutine will ever block.

Since the number of items in the container is generally known, it makes sense to use a channel with enough capacity to hold all the items. This way, the iterator will never block, though the consumer goroutine still might. However, this effectively doubles the amount of memory required to iterate over any given container, so channel capacity should be limited to some maximum number. Timing or benchmarking your code will help you find the buffer capacity for minimal memory usage and optimal performance.

Lately, we have been talking about synchronization and communication. However, we still have not discussed how to switch from one goroutine to another. Study the next lesson to see how to do that!