Synchronous and Asynchronous Communication

This lesson describes synchronous and asynchronous communication and how channels are designed to handle both.

WE'LL COVER THE FOLLOWING

Blocking of goroutines

By default, communication is *synchronous*, and *unbuffered*, which means the send operation does not complete until there is a receiver to accept the value. One can think of an unbuffered channel as if there is no space in the channel for data. There must be a receiver ready to receive data from the channel, and then the sender can hand it over directly to the receiver. So, channel send/receive operations *block* until the other side is ready:

- *A send operation on a channel (and the goroutine or function that contains it) blocks until a receiver is available* for the same channel `ch`. If there's no recipient for the value on `ch`, no other value can be put in the channel, which means no new value can be sent in `ch` when the channel is not empty. So, the send operation will wait until `ch` becomes available again. This is the case when the channel-value is received (can be put in a variable).
- *A receive operation for a channel blocks (and the goroutine or function that contains it) until a sender is available* for the same channel. If there is no value in the channel, the receiver blocks. Although this seems a severe restriction, this offers a simple form of synchronizing and which works well in most practical situations.

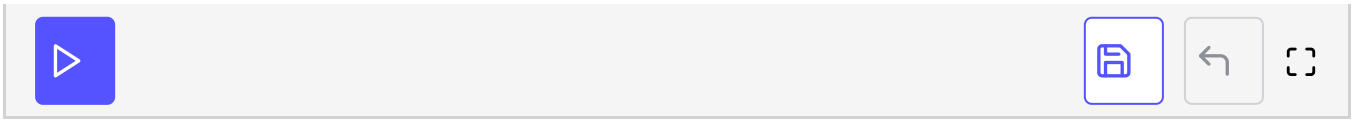
This is illustrated in the following program, where the goroutine `pump` sends

This is illustrated in the following program, where the goroutine `pump` sends integers in an infinite loop on the channel. However, because there is no receiver, the only output is the number 0.

```
package main
import (
  "fmt"
  "time"
)

func main() {
  ch1 := make(chan int)
  go pump(ch1) // pump hangs
  fmt.Println(<-ch1) // prints only 0
  time.Sleep(1e9)
}

func pump(ch chan int) {
  for i:= 0; ; i++ {
  ch <- i
  }
}
```



Channel Block

A channel for *ints* `ch1` is made at **line 8**. At **line 9**, a goroutine `pump` is started, passing `ch1` as a parameter. `Pump` is an infinite loop (from **line 15** to **line 17**), trying to push numbers on the channel, but the channel only can accept one number. This is printed out at **line 10**, after which, the program exits.

The `pump()` function, which supplies the values for the channel, is sometimes called a *generator*. To unblock the channel, define the function `suck`, which reads from the channel in an infinite loop. See the following snippet:

```
func suck(ch chan int) {
  for {
    fmt.Println(<-ch)
  }
}
```

and start this as a goroutine in `main()`:

```
go pump(ch1)
go suck(ch1)
```

```
time.Sleep(1e9)
```

Give the program 1s to run. Now tens of thousands of integers appear on output.

Synchronization between one (or more) channel(s)

Communication is, therefore, a form of synchronization. Two goroutines exchanging data through a channel synchronize at the moment of communication. Unbuffered channels make a perfect tool for synchronizing multiple goroutines. It is possible that the two sides block, waiting for each other, creating what is called a *deadlock situation*. The Go runtime will detect this and panic, stopping the program with this message: `all goroutines are asleep - deadlock!` A deadlock is always caused by bad program design. Deadlocks, like infinite loops, shouldn't happen.

We see that channel operations on unbuffered channels can *block*. The way to avoid this is to design the program such that blocking does not occur (preferably), or by using buffered channels.

Asynchronous channels

An unbuffered channel can only contain 1 item, and for that reason, it is sometimes too restrictive. We can provide a buffer in the channel, whose capacity gets set in an extended make command, like this:

```
buf := 100
ch1 := make(chan string, buf)
```

The `buf` is the number of elements (in this example: strings) the channel can hold.

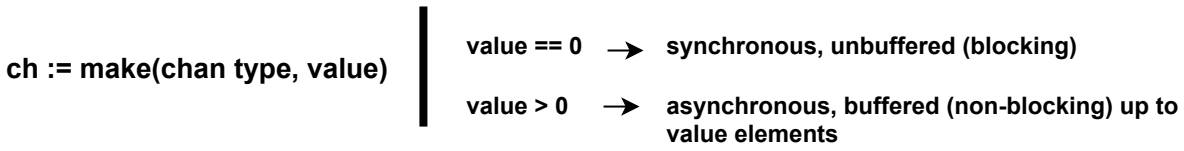
Sending to a buffered channel will not block unless the buffer is full (the capacity is completely used), and reading from a buffered channel will not block unless the buffer is empty.

The buffer capacity does not belong to the type, so it is possible to assign channels with different capacities to each other, as long as they have the same element type. The built-in `cap` function on a channel returns this buffer

capacity.

If the capacity is greater than 0, the channel is asynchronous, which means the communication operations succeed without blocking if the buffer is not full (sends) or not empty (receives). Elements are received in the order they are sent. If the capacity is zero or absent, the communication succeeds only when both a sender and receiver are ready.

To synthesize:



If you use buffers in the channels, your program will react better to sudden increases in the number of *requests*. It will react more elastically, and it will be more scalable. Despite this, design your algorithm in the first place with unbuffered channels, and only introduce buffering when the former is problematic.

Now that you are familiar with the basics of goroutines and channels, the next lesson brings you a challenge to solve.