Deadlocks

Let's get familiar with deadlocks and learn how to avoid them in concurrent programming.

WE'LL COVER THE FOLLOWING

What is a deadlock?

> A deadlock occurs when all processes are blocked while waiting for each other and the program cannot proceed further.

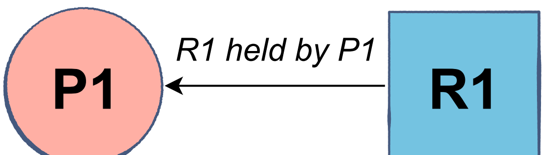
Let's have a look at the conditions for a deadlock to occur:

Coffman Conditions

There are four conditions, known as the **Coffman Conditions**, that must be present simultaneously for a deadlock to occur:

Mutual Exclusion

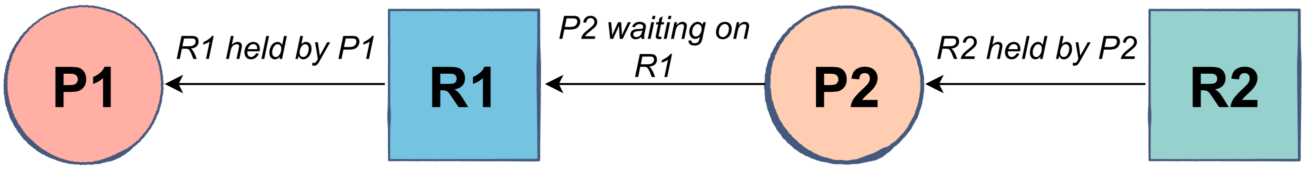
A concurrent process holds at least one resource at any one time making it non-sharable.



Mutual Exclusion

Hold And Wait

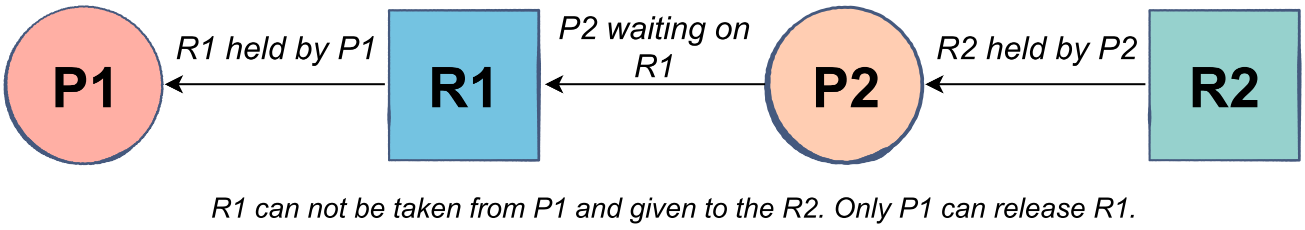
A concurrent process holds a resource and is waiting for an additional resource.



Hold And Wait

No Preemption

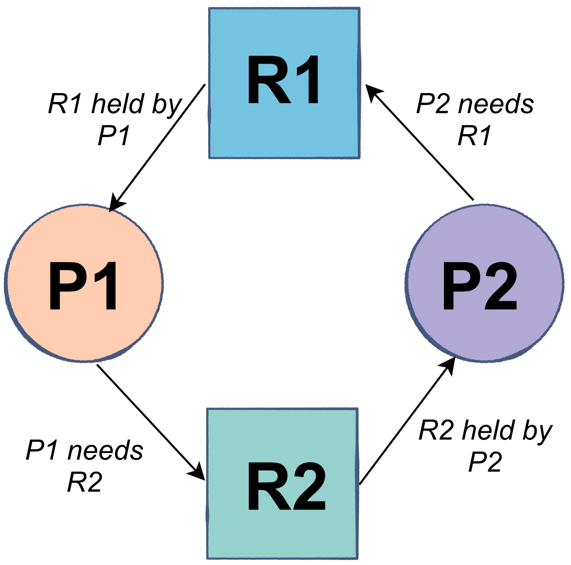
A resource held by a concurrent process cannot be taken away by the system. It can only be freed by the process holding it.


P1
R1 held by P1
R1
P2 waiting on R1
P2
R2 held by P2
R2
R1 can not be taken from P1 and given to the R2. Only P1 can release R1.

No Preemption

Circular Wait

A concurrent process must be waiting on a chain of other concurrent processes such that P1 is waiting on P2, P2 on P3, and so on, and there exists a Pn which is waiting on P1. This forms a circular loop.



Circular Wait

In order to prevent deadlocks, we need to make sure that at least one of the conditions stated above should not hold. The good news is that Go is able to detect deadlocks at runtime.

Here is an example of how Go detects deadlocks:

```go
package main
import "fmt"

func main() {
        mychannel := make(chan int)
        mychannel <- 10
        fmt.Println(<-mychannel)
}
```

Example of Go detecting deadlock

If you run the code above, you will get the following error:

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
    /usercode/main.go:6 +0x62
exit status 2
```

This is because the program is stuck on sending a value to the channel: `mychannel <- 10`. The sending operation is a blocking operation and requires the receive channel to be ready before sending data to the channel. We'll learn more about this when we'll study channels in the next chapter!

Note that we can avoid this deadlock if we put the send operation in a goroutine such that both the send/receive operations are ready for each other simultaneously.

```go
package main
import "fmt"

func main() {
        mychannel := make(chan int)
        go func(){
                mychannel <- 10
        }()

        fmt.Println(<-mychannel)
}
```

Example of avoiding deadlock

Bear in mind that Go only detects deadlocks when the program is stuck as a whole and not when a few of the goroutines are blocked. Also, goroutines are mostly blocked while waiting for a channel operation or for the locks which belong to the `sync` package. Have a look at an example below:

```
package main
import "fmt"

func main() {
        mychannel1 := make(chan int)
        mychannel2 := make(chan int)
        mychannel3 := make(chan int)
        go func(){
                <-mychannel1
        }()

        go func(){
                mychannel2 <- 20
        }()

        go func(){
                <-mychannel3
        }()


        fmt.Println(<-mychannel2)
}
```

As you can see from the code above, the first and the third goroutine are blocked because there is no send operation that sends to these channels. Still, the program executes successfully and we don't get any deadlock error as it only occurs when the program is stuck as a whole. Now that you have a sense of what deadlocks are, we'll learn about starvation in the next lesson.