

# Synchronization



**01**

About

**02**

Mutex

**03**

Wait Groups

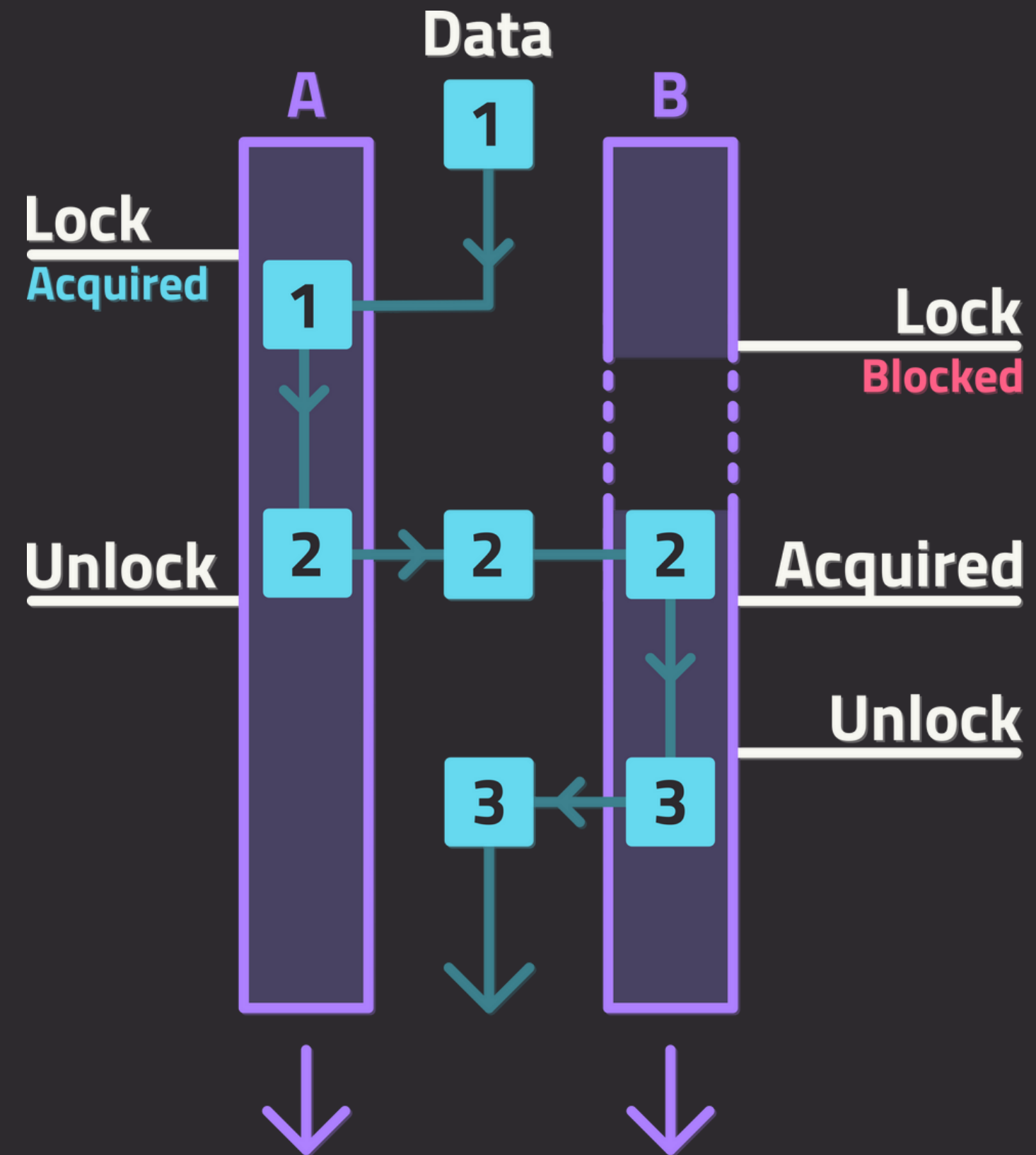
# Synchronization

- | Managing data across multiple goroutines can become problematic and hard to debug
  - | Multiple goroutines can change the same data leading to unpredictable results
  - | Using channels to communicate is not always ideal
- | **Synchronization** solves this issue, and enables:
  - | Waiting for goroutines to finish
  - | Prevents multiple goroutines from modifying data simultaneously

# Mutex

- | A **mutex** is short for mutual exclusion
- | Provides a way to **lock** and **unlock** data
  - | Locked data cannot be accessed by any other goroutine until it is unlocked
  - | While locked, all other goroutines are **blocked** until the mutex is unlocked
    - | Execution waits until lock is available, or if **select** is used
- | Helps reduce bugs when working with multiple goroutines

# Mutex





# Mutex

```
import "sync"

type SyncedData struct {
    inner map[string]int
    mutex sync.Mutex
}

func (d *SyncedData) Insert(k string, v int) {
    d.mutex.Lock()
    d.inner[k] = v
    d.mutex.Unlock()
}

func (d *SyncedData) Get(k string) int {
    d.mutex.Lock()
    data := d.inner[k]
    d.mutex.Unlock()
    return data
}
```

```
data := SyncedData{inner: make(map[string]int)}
data.Insert("sample", 5)
data.Insert("test", 2)
fmt.Println(data.Get("sample"))
fmt.Println(data.Get("test"))
```

# Deferred Unlock

| defer can be used to ensure the mutex gets unlocked

```
func (d *SyncedData) Get(k string) int {  
    d.mutex.Lock()  
    data := d.inner[k]  
    d.mutex.Unlock()  
    return data  
}
```

```
func (d *SyncedData) Get(k string) int {  
    d.mutex.Lock()  
    defer d.mutex.Unlock()  
    return d.inner[k]  
}
```

```
func (d *SyncedData) Insert(k string, v int) {  
    d.mutex.Lock()  
    d.inner[k] = v  
    d.mutex.Unlock()  
}
```

```
func (d *SyncedData) Insert(k string, v int) {  
    d.mutex.Lock()  
    defer d.mutex.Unlock()  
    d.inner[k] = v  
}
```

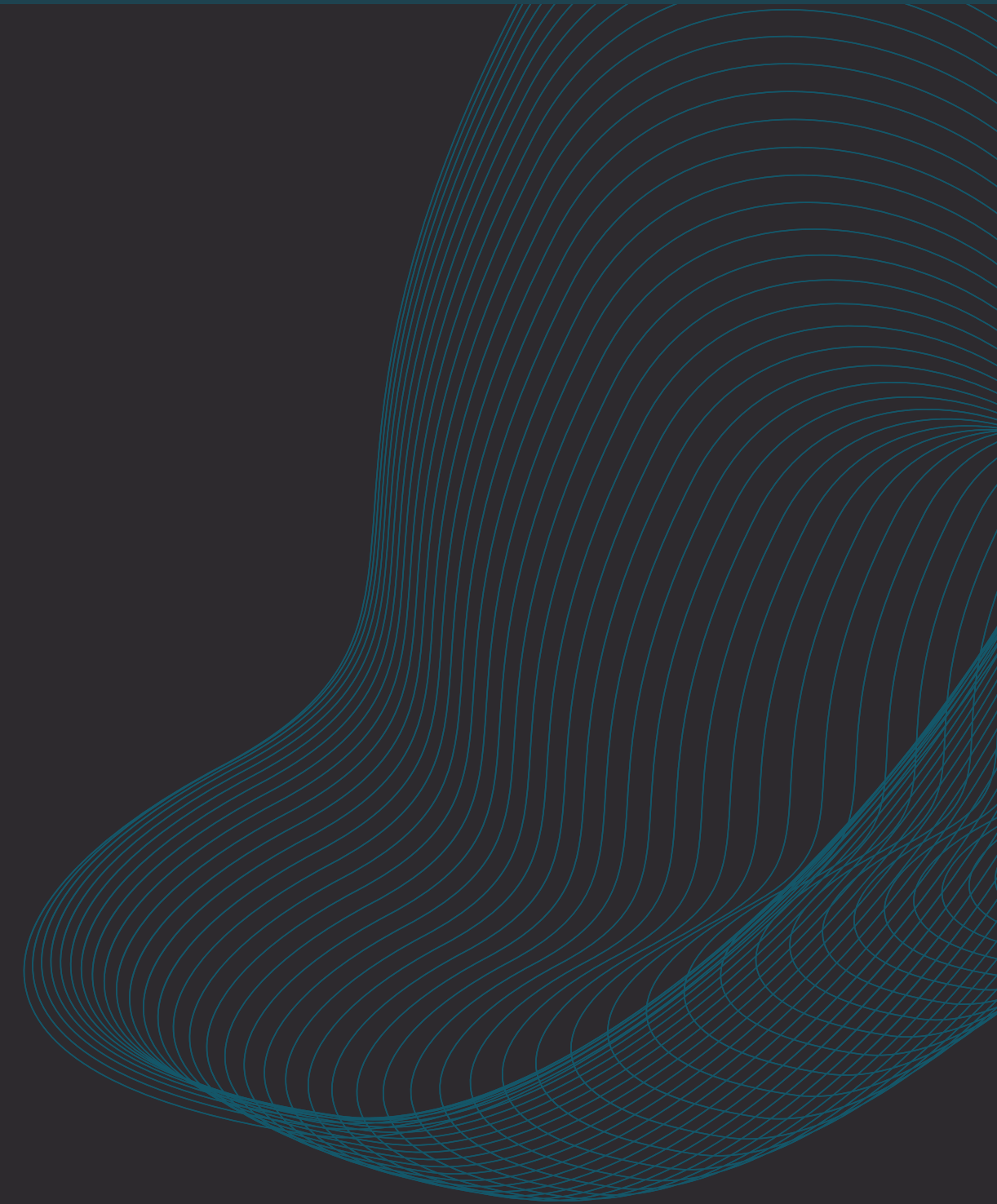
# Wait Groups

- | Wait groups enable an application to wait for goroutines to finish
- | Operates by incrementing a counter whenever a goroutine is added, and decrementing when it finishes
  - | Waiting on the group will block execution until the counter is 0



# Wait Groups

```
var wg sync.WaitGroup
sum := 0
for i := 0; i < 20; i++ {
    wg.Add(1)
    value := i
    go func() {
        defer wg.Done()
        sum += value
    }()
}
wg.Wait()
fmt.Println("sum =", sum)
```





# Recap

- | Data can be safely accessed across goroutines using a **mutex**
  - | Locking a mutex prevents other goroutines from locking it
  - | Always remember to unlock a mutex
- | It is possible to wait for goroutines to finish with a **wait group**
  - | Add 1 per goroutine to the wait group, then use **.Done()** in each goroutine to decrement the group counter
- | Using **defer** makes it simple to unlock mutexes and when working with wait groups