# Concurrency (1) - Fundamentals, Goroutines, Channels

nprenet@bsu.edu

CS431, Spring 2022

## Contents

## Discussion: Data Race in a Bank Account

Let us a classic example to illustrate the most important issue that comes with concurrency: a **race condition**.

```go
// package bank implements a bank with only one account
package bank

var balance int

func Deposit(amount int){ balance = balance + amount }

func Balance() int { return balance }
```

Any *sequential* execution of calls to `Deposit` and `Balance` will yield the right answer, where `Balance` report the sum of all amounts previously deposited. However, the program is not **concurrency-safe**. Consider the following code bit, where the functions are run into concurrent routines, i.e. routines that can be active *at the same time*.

```
// Alice
go func() {
    bank.Deposit(200)                      // A1
    fmt.Println("=", bank.balance())       // A2
}()

// Bob
go bank.Deposit(100)                     // B
```

Intuitively, there are only 3 possible ordering of actions A1, A2, and B, with 3 possible outcomes:

```
 Alice first            Bob first             Alice/Bob/Alice
           0                      0                       0
 A1      200            B       100           A1        200
 A2    "=200"           A1      300           B         300
 B       300            A2    "=300"          A2      "=300"
```

Note in all cases, the final balance is correct and each customer is satisfied. However there is a *fourth* possible outcome, in which Bob's deposit occurs in the middle of Alice's deposit:

```
                 0
    A1r          0            ... = balance + amount
    B          100
    A1w        200            balance = ...
    A2       "=200"
```

In this case, Bob's transaction disappears! This is what we call a **data race**: two goroutines access the same variable concurrently and at least one of the accesses is a write.

Considerations:

- Even the notion that a concurrent program is an interleaving of sequential programs is a false intuition.
- Can you fix the problem by inserting a sleep into Bob's `Deposit` code, as shown below?

  ```
  func Deposit(amount int) {
    time.Sleep(1 * time.Second)
    balance = balance + amount
  }
  ```

  This is, of course, a terrible idea. Not to mention the fact that it defeats the purpose of concurrency (better performance), it does make incorrect outcomes less likely, but still possible, because the logic is flawed.

- *There is no such thing as a benign data race.* Your confidence should come from the fact that your program is logically correct, not from the absence of issues on your platform.

## 1. Definitions and issues

If we cannot confidently say that one event $x$ *happens* before another event $y$, then events $x$ and $y$ are said to be concurrent.

- Order of events within a single goroutine *is* sequential.
- With two goroutines `g1` and `g2` running, the succession of an event $x$ in `g1` and an event $y$ in `g2` is never certain.

### 1.1 Race conditions

When two or more operations must execute in the correct order, but the program has not been written so that this order is guaranteed to be maintained. E.g.

```
var data int
go func(){
    data++
}()
if data == 0 {
    fmt.Printf("the value is %v.\n", data)
}
```

The output may be either one of the following:

- Nothing is printed (line 3 executes before 5)
- "the value is 0" is printed (lines 5 and 6 execute before line 3)
- "the value is 1" is printed (line 5 executes before line 3, but line 3 executes before line 6)

In itself, a race condition just describes the fact that the behavior of a program is not deterministic.

- When involving only read operations, it may be confusing to the observer, but it does not involve data corruption.
- When invoving at least one write operation to shared data, we have a **data race**, illustrated by our initial discussion.

## 1.2 What are the properties of concurrency-safe operations and data?

### Atomicity

An operation is **atomic** if it happens *in a given context* in its entirety without anything happening in the same context simultaneously. E.g.

```
i += 1
```

This incrementation statement combines 3 operations (read, add, assign). Whether it is atomic or not depends on the context:

- in a sequential program, the statement may be considered atomic—no other process interferes with `i`'s incrementation during its execution;
- in a concurrent goroutine that does *not* expose `i` to other goroutines, the code is atomic.

→ to make concurrent programs safe, we have to force atomicity for those parts of our program that require it.

### Memory Access Synchronization

Various techniques in Go allow for making the *critical section* of a program atomic in a concurrent context:

- Using locks (Go `sync` package), in order to guard access to critical section of code
- Confining data to single goroutines, and then using channels so that goroutines communicate to each other the variables they need to operate on, instead of sharing them.

We'll cover them more in depth, but just know that they all implement the same goal: *synchronizing* access to the memory. They may have unintended effects, however, that are briefly described in the next section.

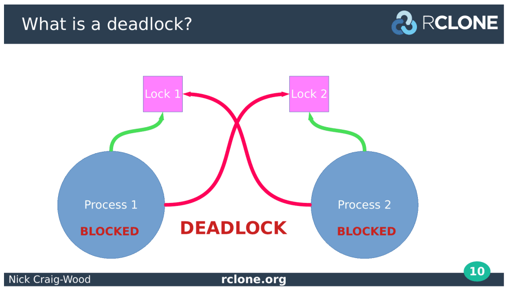
## 1.3 Why is concurrency hard?

### Deadlocks

A deadlocked program is one in which all concurrent processes are waiting on one another. In this state, the program cannot recover without outside intervention. For instance:

```
var (
              lock1 sync.Mutex
              lock2 sync.Mutex
```

```
	)
	go func() {
		for {
			lock1.Lock()
			lock2.Lock()
			lock2.Unlock()
			lock1.Unlock()
		}
	}()
	go func() {
		for {
			lock2.Lock()
			lock1.Lock()
			lock1.Unlock()
			lock2.Unlock()
		}
	}()
	select {}
}
```



Output:

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

- if all routines in the program deadlock, Go detects it at runtime
- unfortunately, this is rarely the case: a long as there something else running, the detector does not kick in

### Livelocks

Programs that are actively performing concurrent operations, but these operations do nothing to move the state of the program forward: indeed, they are two or more concurrent processes attempting to get a lock, failing, and trying again and again, without coordination.

E.g. (an image) trying to avoid walking into another person in a hallway: you move to one side to let her pass, but she's done just the same. When this is going forever, you have a livelock.

- difficult to detect, because there is some activity going on;
- it is a special case of resource starvation, where two processes are starved equally of a shared lock.

### Starvation

Any situation where a concurrent process cannot get all the resources it needs to perform work, because one or more greedy routines are hogging a lock.

- Usually caused by locks that are held unnecessarily beyond the section of program that needs exclusive access to a resource (*critical section*).
- May be identified by metrics and logging.

# 2. Go approach to concurrency: goroutines and channels

## 2.1 Concurrency is an abstraction

Concurrency is a property of the code; parallelism is a property of the running program.

Explanation: if you write your code *as though* those routines actually execute in parallel, that makes it concurrent. On most platforms (your laptop, or personal computer), the reality is that your routines are executing in sequential manner faster than is distinguishable. Time share is the rule, in every context:

- program's runtime
- operating system
- container or virtual machine
- (ultimately) CPU

Concurrent code is an *abstraction* layer over a lower-level context that may be parallel (in a parallel hardware architecture, f.i.) or not. Historically, concurrent logic has not been well decoupled from OS threads (see Java threads). By contrast

- Go **goroutines** allow the programmer to think at a higher level: they are OS-independent and built in the language (vs. thread library)
- **channels** allow goroutine to communicate and synchronize with each other, offering an alternative to lock-based memory access.

## 2.2 Goroutines

A goroutine is a function that runs concurrently alongside other code:

- Neither OS threads, nor green threads[1], they are higher-level *coroutines*, i.e. routines that cannot be interrupted.
- Goroutines are very lightweight.
- All goroutines operate in the same address space and simply hosts functions: *any function or method* can be run as a goroutine.

Example:

```go
func main(){
    go sayHello()
    // continue doing other things
}

func sayHello(){ fmt.Println("hello") }
```

Every program has at least one goroutine: the *main* goroutine, which runs (implicitly) the `main` function. When a child goroutine forks out of the main routine, both routines run to completion. When the parent goroutines terminates, child goroutines that are still executing are abruptly terminated, unless you make the child routines rejoints the parent at a *joint point*, as illustrated by Figure 1. This explains why the program above is not likely to show anything on the console.

The example below illustrates a simple application of goroutines:

- The main routine is an conditional loop that listens to connections requests on the 8000 port.
- The `handleConn` functin (which serves the current time to the client) runs in a goroutines, allowing for simultaneous, non-blocking connections.

```go
func main(){
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
```

[1] "Green" threads designates those processes that are managed by a language's runtime. In the backends, goroutines are scheduled to run on N green threads, that are themselves spread on M operating system threads (M:N threap mapping).

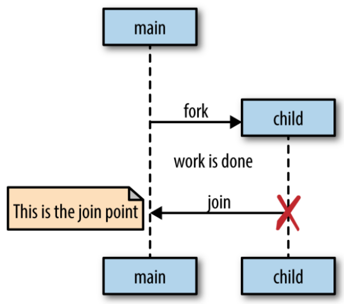


Figure 1: Figure 2.1: Katherine Cox-Buday, *Concurrency in Go*, 2017, p. 39

```
        log.Fatal(err)
    }
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Print(err) // connection aborted
            continue
        }
        go handleConn( conn )  // more than one client
    }                              // can connect at a time
}
func handleConn(c net.Conn){
    defer c.Close()
    for {
        _, err := io.WriteString(c, time.Now().Format("15:04:05\n"))
        if err != nil {
            return       // client disconnected
        }
        time.Sleep(1 * time.Second )
    }
}
```

Because the parent function runs an endless loop, having the two routines join at some point is not a concern here: all calls to the child goroutine return before the parent routine terminates.

## 2.3 Channels

### Creation

Channels are the connections between goroutines:

- a channel lets a goroutine send values to another goroutine
- each channel is a conduit for values of particular type (*element type*)

```
ch := make(chan int) // ch has type 'chan int'
```

Note: `ch` is a reference (a pointer): when passed around from caller to callee, both routines refer to the same channel

**Operations**:

```
ch <- x       // send value x to channel ch
x = <-ch      // receives from channel ch, and assign to x
```

```
<-ch            // receives, but discard the result
close(ch)       // close a channel
```

### Unbuffered channels

Creation:

```
ch = make(chan int)
ch = make(chan int, 0) // capacity = 0
```

A send operation on an unbuffered channel blocks the sending goroutine until *another goroutine* executes a corresponding receive on the same channel. The reverse is true too: a receive blocks until a send operation happens on the same channel.

→ a channel allows for goroutines to *synchronize*. As an example, let us it to fix our `hello.go` program:

```
func main(){

    done := make(chan struct{})

    go func(){
        fmt.Println("hello")
        done <- struct{}{}
    }()
    <-done      // blocks until child goroutine sends signal
}                       // on channel
```

Comments:

- use a anonymous function (fairly typical)
- `main` goroutine blocks until termination signal is sent by child routine
- in this case, we don't care about the message's value, but on the fact of communication: to stress this aspect, we use an empty `struct` type for such message *events*.

## An example: bank account

Goroutines and channels allow us to solve the data race problem described in our introductory discussion. The technique is called **data confinement** because it ensures that at any time a single goroutine has access to the variable: in order to access the variable (for reads or updates), other goroutines must send a request:

```
var balances = make(chan int)
var deposits = make(chan int)

func Deposit(amount int) { deposits <- amount }

func Balance() int { return <-balances }

func teller() {                 // the monitor goroutine
        var balance int         // brokers access to the balance
        for {
                select {
                case amount := <-deposits:
                        balance += amount
                case balances <- balance:
                }
        }
}
func init() {
```

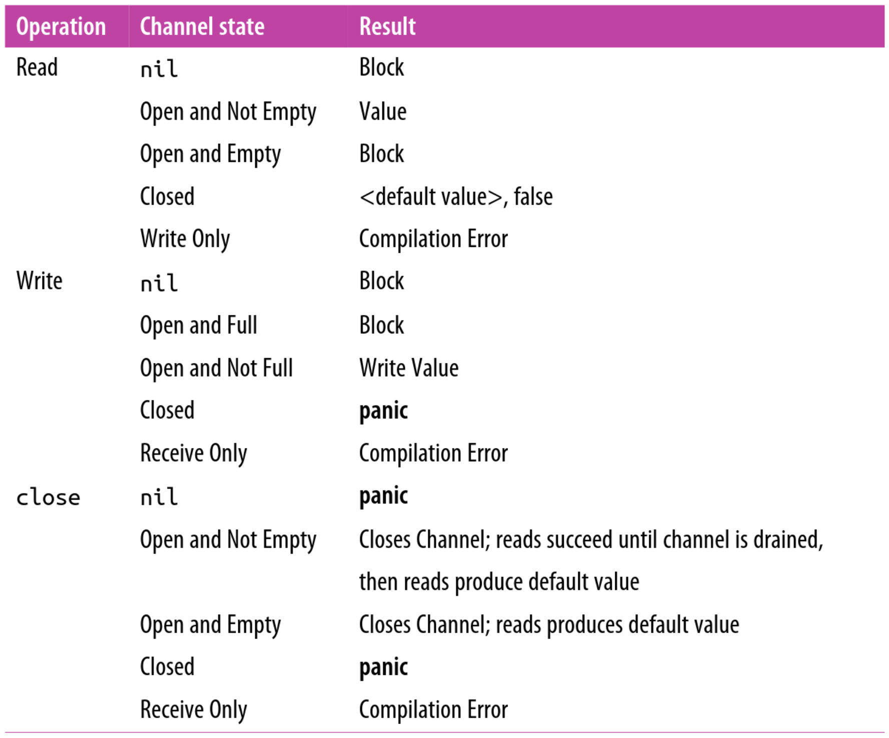
```go
        go teller()
}
```

Summary (the Go mantra):

> Do not communicate by sharing memory; instead, share memory by communicating.

## Appendix: channel operations

*Table 3-2. Result of channel operations given a channel's state*



| Operation | Channel state | Result |
|---|---|---|
| Read | `nil` | Block |
| | Open and Not Empty | Value |
| | Open and Empty | Block |
| | Closed | <default value>, false |
| | Write Only | Compilation Error |
| Write | `nil` | Block |
| | Open and Full | Block |
| | Open and Not Full | Write Value |
| | Closed | **panic** |
| | Receive Only | Compilation Error |
| `close` | `nil` | **panic** |
| | Open and Not Empty | Closes Channel; reads succeed until channel is drained, then reads produce default value |
| | Open and Empty | Closes Channel; reads produces default value |
| | Closed | **panic** |
| | Receive Only | Compilation Error |

Katherine Cox-Buday, *Concurrency in Go*, 2017, p. 75.