Concurrency and Parallel Computing

Paradigm

* Sequential Paradigm: Assume a single processor, single core architecture
* Concurrency vs Parallelism
  - A parallel program will use a multiplicity of independent processing units to accomplish tasks.
  - A concurrent program is one structured such that there are multiple threads of control, which gives the impression of simutaneous execution.
  - "Difference between dealing with and doing multiple things at the same time".

Complexity

* Increased Design Effort
  - What is the appropriate level of division of task? Decomposition across many level.
    - Blocks
    - Functions
    - Classes
    - Tasks
  - What inter-task communication is required?
    - Ideally, non, deterministic outcomes.
    - Normally, some, non-deterministic outcomes.
  - How can task correctness be maintained.
    - No longer any guarantees of order
* Increased Debugging Effort
  - Execution is potentially non-deterministic
  - No two executions will yield the same sequence of operations that lead to a bug
  - Creates another area in programs for bugs to hide and go unnoticed until some critical moment.
* Nasty Side Effects
  - Even if programs are supposedly bug-free:
    - No compilation errors.
    - No logical errors
    - No runtime errors
  - Can have concurrency issues:
    - Race conditions
    - Dead lock
    - Starvation.
  - Race condition: Two tasks that are coordinated in a sequential order is correct. Two tasks that are not coordinated and executing concurrently is wrong.
  - Dead lock: Tasks are harder to notice each other when the Tasks and Recoures are many and possibly distributed across many different systems.
  - Resource Starvation: Task A may have higher priority, or the scheduling algorithm is unfair.

Concurrency in GO

* Provides simple features to investigate concurrency with minimal cognitive overhead (i.e. no addition libraries / tools to learn).
  - Goroutines
    - Lightweight processes that can be created and execute function concurrently
    - go f()
  - Channels
    - Communication mechanism between goroutines
    - go func() { message <- "ping" }()
    - msg := <- messages

Introduction to GO

* Key Features
  - Statically typed language
  - Type inference — Fast compilation
  - Remote package management
    - Using the delightful invocation go get.
  - Garbage collection
    - Automatic memory management
    - Unrequired object's memory is reclaimed
  - Composition over inheritance
    - Less time worrying about complex type hierarchies
  - Build-in Concurrency
    - Primitives in the language, not an extra library
    - Intuitive spawning of goroutines to complete tasks.
    - Model of communication based on established theory (CSP)
  - Concurrency Management
    - Tool support to detect concurrency bugs (e.g. Race conditions)

The Inevitable Hello World

```
package main
import "fmt"
func main() {
    fmt.Println("Hello world")
}
```

Channel creation and Communication

Pinger, Ponger, Printer (PPP)

```
func main() {
    var c chan string = make(chan string)
    go pinger(c)
    go ponger(c)
    go printer(c)
    var input string
    fmt.Scanln(&input) }
```

```go
func pinger(c chan string) {
    for i := 0;  ; i++ {
        c <- "ping"
    }
}

func ponger(c chan string) {
    for i := 0; ; i++ {
        c <- "pong"
    }
}

func printer(c chan string) {
    for {
        msg := <-c
        fmt.Println("printer received:", msg)
        time.Sleep(time.Second * 1)
    }
}
```

Buffered Channels

* Unbuffered Channels:
  - Channels can be created that have no buffer
  - Sender goroutine will block, until a Receiver consumes the message sent into the channel
  - Creation: ic := make(chan int)
* Buffered Channels
  - Sender blocks only while the message has been copied into the channel
  - The channel has a given capacity (size of the buffer)
  - If buffer is full, sender waits for some receiver.
  - Creation: ic := make(chan int, 10).

Synchronisation

* Syncing goroutines
  - Use a shared channel to coordinate when both tasks have ended
  - Main(M) will wait for the Worker (W) to finish and signal over the shared channel.
* Channel Synchronisation

```go
func worker(done chan bool) {
    fmt.Print("working... ")
    time.Sleep(time.Second)
    fmt.Println("done")
    done <- true
}
func main() {
    done := make(chan bool, 1)
    go worker(done)
    <- done
}
```

Blocking & Non-blocking Operations

* Using select
  - Select is the switch statement for channels
    - Choose first available channel
    - Randomly choose if multiple channels ready
    - If none are ready, block or take default case.
  - In GO:

```
func main() {
    c1 := make(chan string)
    c2 := make(chan string)
    go func() {
        for {
            c1 <- "from 1"
            time.Sleep(time.Second * 2)
        }
    }()
    go func() {
        for {
            c2 <- "from 2"
            time.Sleep(time.Second * 3)
        }
    }()
    go func() {
        for {
            select {
            case msg1 := <-c1:
                fmt.Println(msg1):
            case msg2 := <-c2:
                fmt.Println(msg2)
            }
        }
    }()

    var input string
    fmt.Scanln(&input)
}

func main() {
    c1 := make(chan string, 1)
    go func() {
        time.Sleep(time.Second * 2)
        c1 <- "result"
    }()
    select {
    case res := <-c1:
        fmt.Println(res)
    case <-time.After(time.Second * 1):
        fmt.Println("timeout!")
    }
}
```

Using a Channel as a Queue

* Processing a queue
  - A group of goroutines share a channel and add messages, creating a queue of work
  - A single goroutine process the message in FIFO order
  - Using range: iterate over each element in queue.

Data Races

* Resolving Race Conditions
  - A data race occurs when two or more threads access the same variable (memory) concurrently and at least one of the accesses is write.
  - The solution is to synchronise access to all mutable data.
    - Using locks (to protects shared memory)
    - Using channels (to communicate activity)
  - Don't communicate by sharing memory; share memory by communicating.

Deadlock

* Go has runtime support for detecting a deadlock situation

The Dining Philosophers

* Fun question.