Concurrency in Golang For Beginners

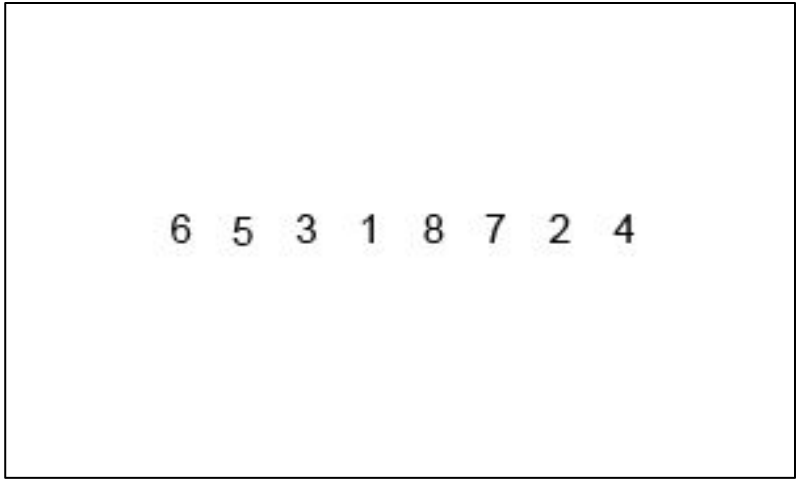
Jayaganesh Kalyanasundaram
(Software Engineer-Site Reliability Engineering)

What did I first notice in Golang?

- **`var i int`** instead of `int i`; in C++
  - **`func Job(i int) int {}`** instead of `int Job(int i) {}`
- **:=** instead of = but everything else is the same as python
- structs are same as C++, slices are similar to python, interfaces can help mimic classes, etc…
- Supports mutex and similar setup to semaphore (calls it `WaitGroup`)

tl;dr Golang == C++ in most parts including performance but with slightly easier syntax like Python

Merge sort quick overview



Picture Courtesy: Wikipedia

Merge sort in Golang

```
func Sort(arr []int) []int {
 if(len(arr) <= 1) {return arr}
 mid := len(arr)/2
 s1 := Sort(arr[:mid])
 s2 := Sort(arr[mid:])
 return merge.Merge(arr, s1, s2)
}
```

```
func Merge(arr []int, arr1 []int, arr2 []int) []int {
 size1 := len(arr1); size2 := len(arr2)
 i := 0; j := 0; index := 0
 for i < size1 && j < size2 {
   if arr1[i] < arr2[j] {
     update(arr, arr1, &index, &i) // update arr[index] with arr1[i]
   } else {
     update(arr, arr2, &index, &j) // update arr[index] with arr2[j]
   }
 }
 for i < size1 {
   update(arr, arr1, &index, &i)  // update arr[index] with arr1[i]
 }
 for j < size2 {
   update(arr, arr2, &index, &j)  // update arr[index] with arr2[j]
 }
 return arr
}
```

Wait, what else I came across?

- **go** keyword!
  Ahha! I don’t need to manually define threads anymore
- **chan**???
  What’s a channel? I’ve never heard of it? What does it do?

Well, why would anyone want to write threads and stuff? Servers would optimize them anyway!

Merge sort with **WaitGroup** and **go**

```
func Sort(arr []int) []int {
    if(len(arr) <= 1) {return arr}
    mid := len(arr)/2
    var s1, s2 []int
    var wg sync.WaitGroup
    wg.Add(2)
    go func () {
        defer wg.Done()
        s1 = Sort(arr[:mid])
    } ()
    go func () {
        defer wg.Done()
        s2 = Sort(arr[mid:])
    } ()
    // The sorting of arr[mid:] & arr[:mid] are concurrent.
    wg.Wait()
    return merge.Merge(s1, s2)
}
```

What’s slowing it down?

```
func Sort(arr []int) []int {
    if(len(arr) <= 1) {return arr}
    mid := len(arr)/2
    var s1, s2 []int
    var wg sync.WaitGroup
    wg.Add(2)
    // Concurrency established
    go func () {
        defer wg.Done()
        s1 = Sort(arr[:mid])
    } ()
    go func () {
        defer wg.Done()
        s2 = Sort(arr[mid:])
    } ()
    // The sorting of arr[mid:] & arr[:mid] occurs Concurrently now.
    wg.Wait()
    return merge.Merge(s1, s2)
}
```

What’s slowing it down?

The merger is expecting 2 fully sorted arrays (/slices).

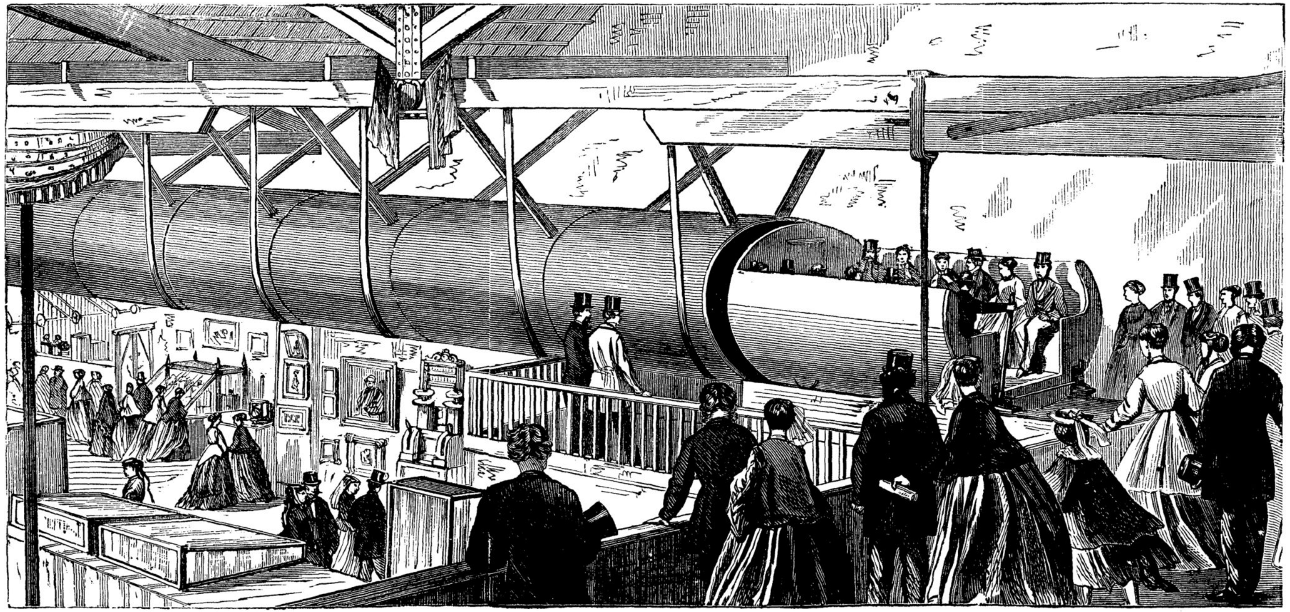
Is there any way to have a stream of data (than static data at once)?
Is there a way the merger can use them as and when the data are available?

Wait, what else I came across?

- **go** keyword!
  Ahha! I don’t need to manually define threads anymore
- **chan**
  What’s a channel? I’ve never heard of it? What does it do?

Well, why would anyone want to write threads and stuff? Servers would optimize them anyway!

So what are Channels?



Picture Courtesy: Pneumatic tube Wikipedia

So what are Channels?

- Has a start (sender) and an end (receiver)
- Has a buffer to hold (could also be 0)
- If the receiver is not receiving then the sender will be blocked
- If the sender is not sending, the receiver will wait for eternity
- The sender can close the channel to state completion

Merge sort with **Channels**

```
func Sort(arr []int, ch chan int) {
    defer close(ch)
    if(len(arr) <= 1) {
        if(len(arr)==1) {
            ch <- arr[0]
        }
        return
    }
    mid := len(arr)/2
    s1 := make(chan int, mid)
    s2 := make(chan int, len(arr) - mid)
    // Concurrency established
    go Sort(arr[:mid], s1)
    go Sort(arr[mid:], s2)
    // Merging happens simultaneously and is not blocked on individual sorting.
    merge.Merge(s1, s2, ch)
}
```

So what are Channels, again?

Journey: **From sequential sorting with blocking merge** to concurrent sorting with blocking merge **to concurrent sorting with non-blocking merge**

Transfer data directly than using shared memory with lock contention

“Don’t communicate by sharing memory, share memory by communicating”

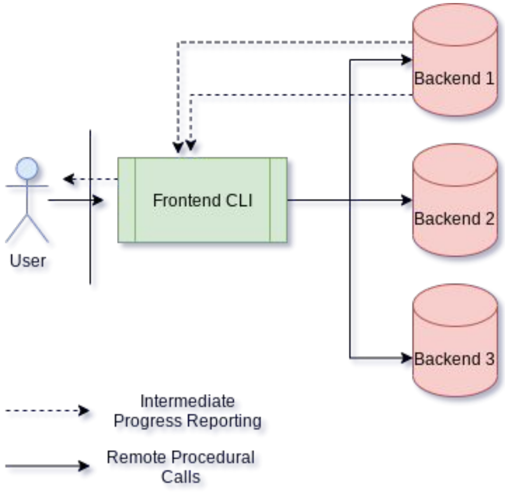
-- Courtesy Share Memory By Communicating - The Go Blog

Where did I use it practically?

One of our internal tools:

- Had a frontend which was surfacing a CLI
- Had multiple backends which communicated with the frontend
  - Each interaction took ~few minutes
- Frontend and backends were run as a single binary
- The user wanted to know the progress of the CLI

Problem:
How do we report the intermediate progress before the function call returns?

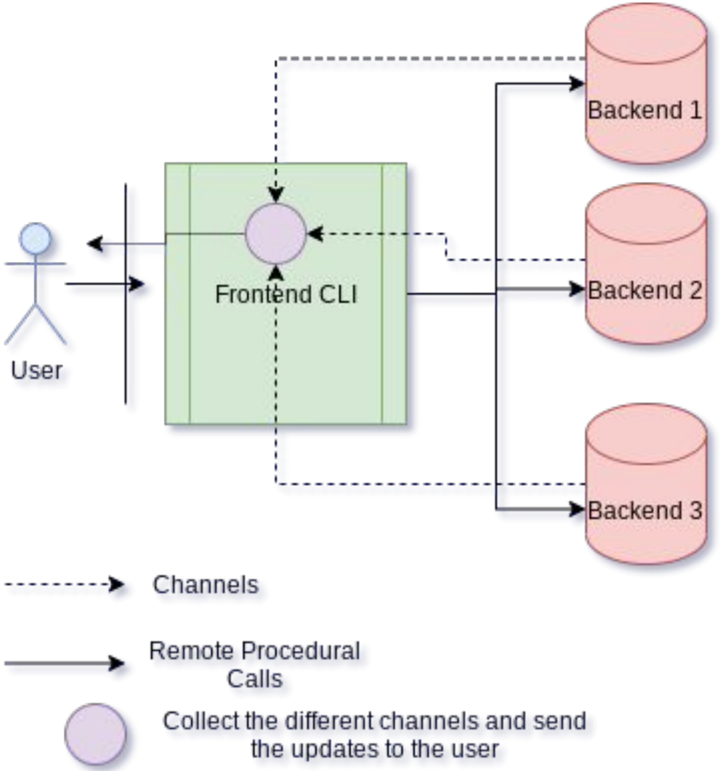


Google

Where did I use it practically?

- Every backend opens a channel with the frontend
- Populates the progress in that channel as and when some milestones are reached
- The frontend ingests the data from all channels and sends them to the user appropriately

Benefits: no extra overhead with logging, etc.



Google

Thankyou :)