Applied Go Concurrency Patterns

Cengiz Can
Lead Software Engineer
hop

cengizcan

Concurrency is hard

Multiple tasks which start, run, and complete in overlapping time periods, in an uncontrolled order.

Race Condition occurs when the timing or order of events affects the correctness of the code.

Data Race race occurs when one thread accesses a mutable object while another thread is writing to it.

Why? **Sharing state**

Concurrency toolset of Go

Goroutines

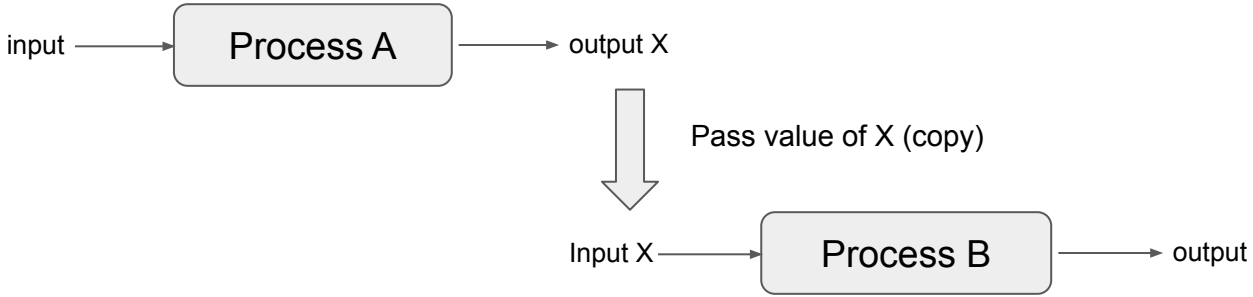
- Independently executing function
- Green thread
- Cheaper than OS thread

Channels

- Communication structure for goroutines
- A tool to designed to save us sharing state
- Inspired by Communicating sequential processes (CSP) paradigm

Communicating Sequential Processes (CSP)

- First introduced by Tony Hoare in 1978.
  - Article: https://www.cs.cmu.edu/~crary/819-f09/Hoare78.pdf
  - Book: http://www.usingcsp.com/cspbook.pdf
- Executing Sequential Processes **communicate** data, do not share state.
- I/O commands



Channels

Do not communicate by sharing memory; instead, share memory by communicating. https://go-proverbs.github.io/

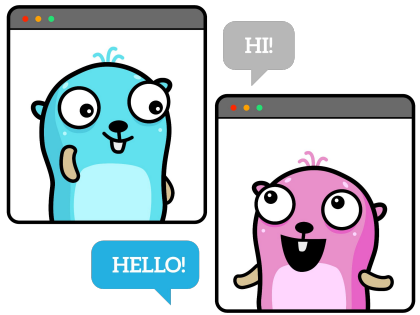
```
c := make(chan int, 3)

// Send. Sender is blocked if the buffer is full or there is no receiver

c <- 1

// Receive. Receiver is blocked if the buffer is empty or there is no sender

val := <-c // Pass the copy of the value
```



Channels are not bulletproof

- Channels may cause deadlocks:
  - goroutines are waiting for each other and none of them is able to proceed
- Closing channels may cause panic:
  - Sending to closed channel
- Channels copy data which may cause performance issues
- Passing pointers through channels may cause data races

Today's Project: Batch image processor

Requirements:

- Scan file system for image files
- Read image file
- Resize
- Create a incremental document id
- Label image with multiple AI algorithms
- Compress: Use an external lossless compression service
- Save
- Nice to have: scan multiple image sources concurrently

Step #1: Scan file system for image files

- Recursively walk through file system starting from the given path
- If encounter a image file:

Approach #1:

- Add its path to a list
- Return the list at the end

Approach #2:

- Send its path through a channel immediately

Generator pattern

```
func generator() <- chan int {
    out := make(chan int)
    go func() {
        for i := 0; i < 5; i++ {
            out <- i
        }
    }()
    return out
}
```

Benefits:

- No need to wait the whole process to finish
- Efficient in terms of memory allocation

Show me the code!

Step #2: Sequential vs. concurrent processing

For every image:

- Read
- Resize
- Generate doc id
- Save

1. Process sequentially
2. For every image start a goroutine

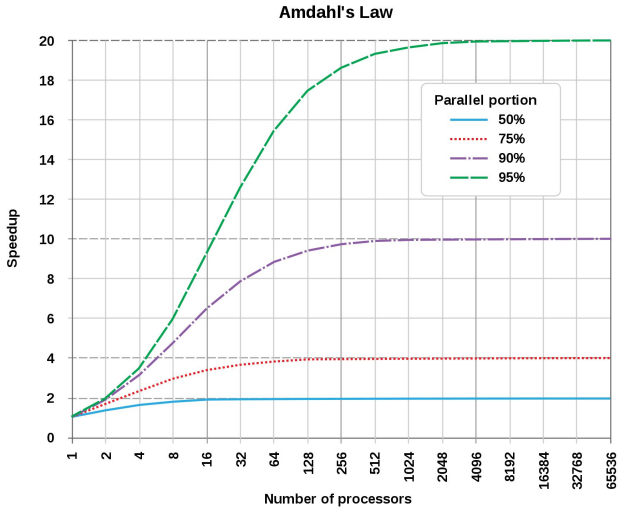
Show me the code!

Step #2: Benchmark

image #	sequential	image # goroutines	x faster
10	7.37ms	0.73ms	**10**
100	74.6ms	1.4ms	**50**
1 000	724ms	9ms	**80**
10 000	7.27s	0.09s	**80**

Amdahl’s law

$$\begin{cases} S_{\text{latency}}(s) \leq \dfrac{1}{1-p} \\ \lim\limits_{s \to \infty} S_{\text{latency}}(s) = \dfrac{1}{1-p}. \end{cases}$$



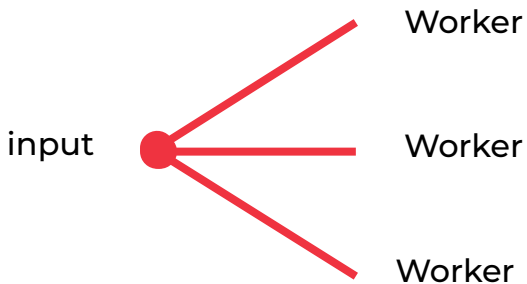
- The theoretical speedup of the latency of the execution of a program as a function of the number of processors executing it.
- The speedup is limited by the serial part of the program
- If 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times.

Source: https://en.wikipedia.org/wiki/Amdahl's_law

Step #3: Add limit to count of working goroutines

Limit the concurrently working goroutine count

Implement **fan out pattern**



Show me the code!

Step #3: Benchmark

image #	goroutines	100 worker	500 worker	1000 worker
10 000	**0.2s**	1s	0.25s	**0.2s**
500 000	11s	47.5s	10.1s	**9.0s**
750 000	28s	71.4s	16.1s	**15.5s**

Step #4: Add Lossless Compression

- Use multiple external compression service, eg: tinyPNG
- These services return:
  - Similar results by means of byte size
  - in various duration
- Pick the first response, throw away the rest
- Similar to JavaScript Promise.Any()
- Implement Rob Pike's **google3.0 pattern**
- Alternative implementation: **quit channel**

Show me the code!

Step #5: Add Some AI - Image Labeling

- Use multiple external AI API's to label objects seen on the image
- We want it all but we won't wait forever

Timeout pattern

Show me the code!

Step #6: Worker pool

Replace our fan out pattern with a worker pool

Worker pool:

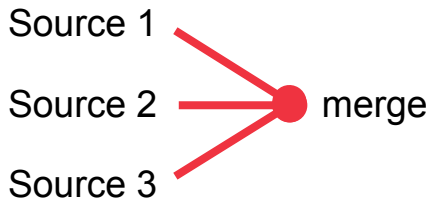
- Works like fan out
- Structured and refined
- A general purpose implementation

An alternative to WaitGroup: **done channel**

Show me the code!

Step #7: Add multiple image source

- We want to feed our work pool from various data sources
- **Fan in pattern**



Show me the code!

Step #8: Mutex vs. Atomic

Mutexes:

- Used to lock **critical section** of the code
- Multiple mutexes eventually causes **deadlocks**
- Go mutex implementation is efficient compared to other languages

Atomic:

- Mapped to thread-safe CPU instructions
- Faster than the mutex

Best Practices:

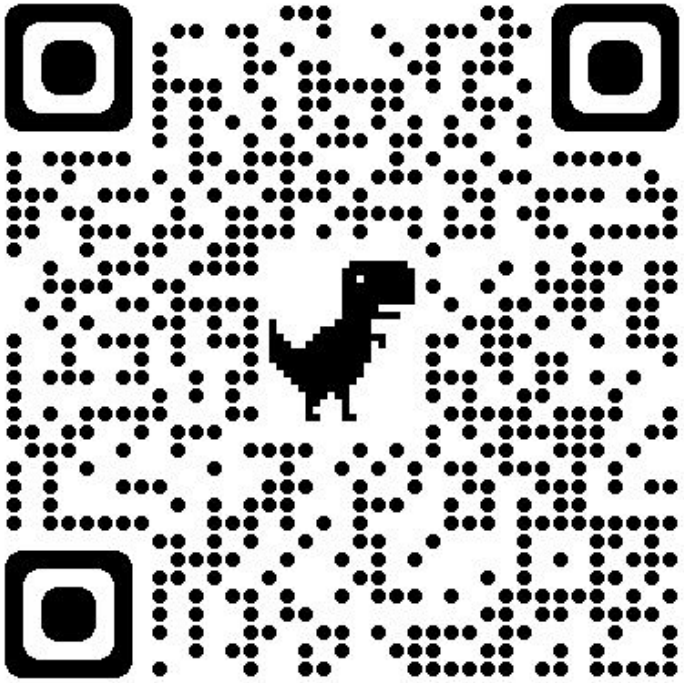
- Design to not to use locks
- Use mutex contention profiler https://go.dev/doc/diagnostics
- Try atomic

Summary

Patterns

- Generator
- Fan out
- Work pool
- Done channel
- Time out
- Fan in

THANK YOU!



github.com/cengizcan/go-concurrency-gophers-11