PROGRAMMING EXERCISES IN GO AND ADA

MAURO CARLIN & MATTIA BOTTARO

CONCURRENCY IN ADA

TASK AND PROTECTED TYPES

- Task
  - Rendezvous
  - This is NOT message passing
- Protected types
  - They allow to exploit communication by sharing memory
  - It is easier to share memory than usual PL (no use of lock)

```
task type T is
    entry A;
    entry B(X : Integer);
end T;
```

```
task body T is
    Y : Natural := 0;
    Z : Positive := 1;
begin
    stm_1;
    accept A do
        stm_2;
    end A;
    stm_3;
    accept B(X : Integer) do
        stm_4;
    end B;
    stm_5;
end T;
```

```
task body T is
begin
    select
        accept A do
            stm_A;
        end A;
    OR
        accept B do
            stm_B;
        end B;
    end select;
end T;
```

```ada
protected type Buffer_T is
      entry Put (Item: Integer);
      entry Get (Item: out Integer);
  private
      A_Container: Container;
      Size: Natural := 0;
  end Buffer_T;
```

```ada
protected body Buffer_T is
   entry Put (Item : Integer) when Size < LIMIT is
   begin
      A_Container(Size) := Item;
      Size := Size + 1;
   end Put;

   entry Get(Item : out Integer) when Size > 0 is
   begin
      Item := A_Container(Size - 1);
      Size := Size - 1;
   end Get;
end Buffer_T;
```

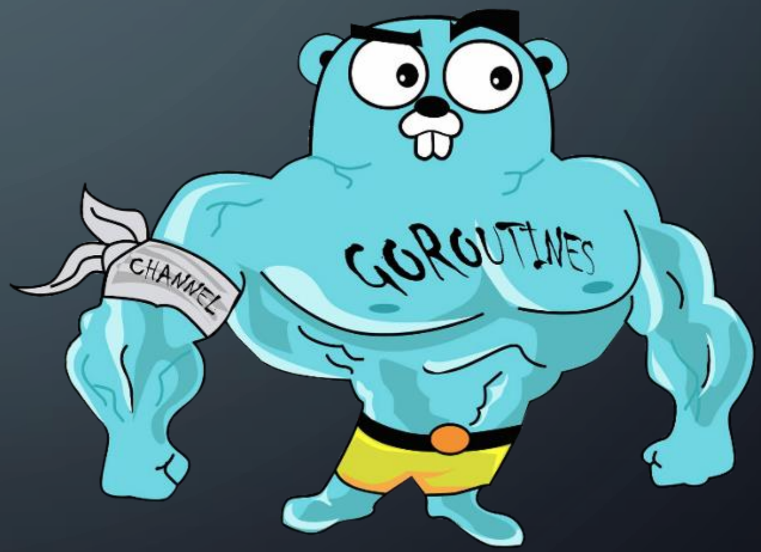
CONCURRENCY IN GO

"Do not communicate by sharing memory; instead, share memory by communicating"

\- Go team

SHARE MEMORY BY COMMUNICATING

- Concurrency is a fundamental part of the language, not a library
- Shared values are passed around on **channels** and, in fact, never actively shared by separate threads of execution
- Data races cannot occur by design



CHANNELS

- By default communication is **synchronous**
- Send and receive are **blocking**
- **Send** on a channel blocks until a receiver is available for the same channel
- **Receive** on a channel blocks until a sender is available for the same channel
- **Sends** to a buffered channel block only when the buffer is full. **Receives** block when the buffer is empty (asynchronous)

EXERCISES

Intel i3-7100

3,9 GHz

2 CPU

2 <u>cores</u> per CPU

FACTORIAL

- Given a **number x** and a **number of processes k**, compute the **factorial of x** by splitting the products in k groups. “Slave“ multipliers can fail, and in this case are restarted.
- A **supervisor** process then collect the partial products and computes the overall product.

- Assumption:
  - Max x = 20 for overflow issue

Es. n = 10, k = 3, fail prob = 40%

FIRST SOLUTION

- Create k "balanced" lists (i.e. [10,7,4], [9,6,3], [8,5,2])
- Main creates multipliers and assigns a list to each of them, if one fails, main creates another multiplier.
- Supervisor collects partial products and computes the final result.

SECOND SOLUTION

- Main creates multipliers, if one fails, main creates another slave.
- This time multipliers request new list of numbers when they finish their computation.
- List are made of 2 numbers (i.e. [10,2], [9,3]…)
- Supervisor collects partial products and computes the final result.

FACTORIAL - BENCHMARK

20!

1° SOLUTION

WORKERS	ADA	GO
1	0,3 ms	1,6 µs
2	0,53 ms	2,5 µs
5	1,4 ms	4,8 µs

2° SOLUTION

WORKERS	ADA	GO
1	1,04 ms	1,5 ms
2	1,15 ms	0,83 ms
5	1,6 ms	0,37 ms

SEARCH

- **Search** a given **string**, in a **set of files**, concurrently.
- Find all occurrences.

- **Assumption**:
  - We search only words, not phrases.
  - Files are 3, two of 2M words and one with 10k words.

FIRST SOLUTION

- Files are splitted in chunks of 200 words.
- Main creates a fixed numbers of “workers”.
- Worker checks for occurrences, printing results, and then asks for another chunk.

SECOND SOLUTION

- Main creates an analyzer for each file.
- Analyzer splits the file in chunks of 200 words, assigning each of them to a “worker” that search for occurrences.
- Workers print occurrences with position and name of the file.

SEARCH - BENCHMARK

1° SOLUTION

WORKERS	ADA	GO
1	2,1 s	0,70 s
6	2,08 s	0,67 s
12	2,1 s	0,62 s
24	2,01 s	0,61 s

2° SOLUTION

ADA	GO
1,3 s	0,44 s

PRIME SIEVE – V1

- Compute **the first N primes** using their recursive definition (0,1 not prime and a number n is prime if it is not a multiple of the primes m < n)
- **Concurrent solution**: Chain of processes Pk, each associated with a prime k
- The first process P2 fed by a generator
- Each Pk checks if k divides x, if yes, x is discarded else it is passed to the next process. If the process is the last one in the chain, x is prime hence it is given in output and a process Px is created

PRIME SIEVE V1 - BENCHMARK

NUMBERS OF PRIMES	ADA	GO
100	20,02 ms	1,01 ms
1000	1,64 s	73,7 ms
10000	3 min 30 s	6,2 s
50000	7 min 35 s	4 min 21 s

PRIME SIEVE – V2

- The previous solution could create **too many processes.**

- Develop a different solution where a **bound** to the number of processes is fixed (hence each process takes care of many primes).

SOLUTION

- **K** subprocesses form a closed chain (last one communicates with the first).
- Every subprocess has a list of primes, updated by main (round robin).
- Every number **N** to be checked comes with a level, that represent the position in the list of primes.
- Every subprocess:
  - controls if N > $\sqrt{list[level]}$, if true N is a **new prime** (this saves a lot of controls)
  - Else, computes N % list[level], if it is equal to zero discards N, otherwise sends it to next subprocess (if it is the last subprocess, sends N to the first subprocess increasing level by 1)
- Only K-1 numbers can be computed in parallel in the chain in order to avoid deadlock

PRIME SIEVE V2 - BENCHMARK

NUMBERS OF PRIMES	ADA	GO
100 – 4 workers	5,3 ms	0,47 ms
100 – 8 workers	4,9 ms	0,47 ms
1000 – 4 workers	112,7 ms	9,9 ms
1000 – 8 workers	98,75 ms	9,57 ms
10000 – 4 workers	2,4 s	219,2 ms
10000 – 8 workers	2,13 s	215,71 ms
50000 – 4 workers	21,3 s	2,07 s
50000 – 8 workers	19,1 s	1,93 s
50000 – 16 workers	18,1 s	1,7 s
50000 – 32 workers	19,9 s	1,25 s