Relatório do trabalho "Implementação em Go ou Rust"

Aluno: Jonathas Augusto de Oliverira Conceição
Linguagem escolhida: Go

Problema Escolhido: Thread-Ring

Thread-Ring
Use OS threads or the language implementation pre-emptive lightweight threads. As a practical matter, continuations & coroutines & cooperative threading will not be accepted.
Please don't implement your own custom "custom scheduler" or use "continuations" or "coroutines" or "cooperative threading" - they will not be accepted.

How to implement
We ask that contributed programs not only give the correct result, but also use the same algorithm to calculate that result.
Each program should:
- create 503 linked pre-emptive threads (named 1 to 503)
- thread 503 should be linked to thread 1, forming an unbroken ring
- pass a token to thread 1
- pass the token from thread to thread N times
- print the name of the last thread (1 to 503) to take the token

Recursos utilizados

- **sync.Mutex**: mecanismo bloqueante de exclusão múltua [1].
- **go**: é uma palavra reservada que dispara uma thread concorrente independente (*goroutine*) [2].
- **chan**: é um canal utilizado para comunicação concorrente entre threads enviando e recebendo dados [3].
- **runtime.GoSched**: libera a CPU permitindo que outra thread possa executar [4].
- **runtime.GOMAXPROCS**: seta o número máximo de CPUs a ser útilizado simultaneamente [5].

Solução

No programa uma *struct* é utilizado para representar uma thread do problema. Cada thread é associado a um id e um mutex que é adquirido na inicialização da thread, bem como uma referência para a próxima thread no ciclo. Após a inicialização uma nova *goroutine* é disparada com a tarefa de execução da thread.

Quando iniciadas as *goroutines*, a rotina principal envia o token para a primeira thread para que o ciclo começe. O número de operações a ser executado é utilizado como token que deve ser passado de thread em thread. A rotina principal é então bloqueada enquanto aguarda até que algum valor seja posto no canal (**chan**) de comunicação indicando que a tarefa terminou. Após receber o ID da última thread pelo canal a thread principal o imprime e termina a execução do programa.

Quando a thread 1 recebe o token inicialmente seu lock é liberado. Ela então readiquire o seu lock, decrementa o token e o passa para a thread seguinte liberando o lock da mesma. Logo as threads que recebem o token tem sempre seu lock liberado para poder processa-lo, se o token contem 0 a thread envia seu ID para o canal de comunicação, indicado o fim das tarefas; Caso contrário o processo de passagem de token é repetido.

O problema pede que o escalonador da linguagem seja totalmente preemptivo, entretando em alguns pontos, como laços de repetição, outras *goroutines* não podem obetar a CPU por preempção [6]. Logo no programa faz-se uso da função **runtime.GoSched** para permitir que outras rotinas possam ser executadas antes do reinício do laço de repetiação onde o lock da thread é adquirido.

Resultados e Desempenho

Os experimentos foram executados numa máquina com processador Intel Core i7, frequência de 3.40GHz, 4 cores físicos e 4 lógicos, 8GiB de memória RAM. O sistema operacional foi um Ubuntu 16.04, com a versão 1.6 do Go Language. Para as medições de tempo o comando **time** do GNU/Linux foi utilizado.

A a tabela abaixo apresenta a média de 30 execuções parada cada um dos números de operações.

Operações	1000	50000	500000	5000000	50000000
Média	0,0029667	0,009800	0,1131667	0,7135667	6,53210

Referências

[1]: https://golang.org/pkg/sync/#Mutex
[2]: https://golang.org/ref/spec#Go_statements
[3]: https://golang.org/ref/spec#Channel_types
[4]: https://golang.org/pkg/runtime/#Gosched
[5]: https://golang.org/pkg/runtime/#GOMAXPROCS
[6]: https://github.com/golang/go/issues/11462