Kaue Soares da Silveira
171671

Trabalho Opcional 1: Estudo da Linguagem Golang

Disciplina INF01151 - Sistemas Operacionais II N

Professor: Alexandre Carissimi

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Sumário

1 Introducao

"Não comunique-se compartilhando memória. Em vez disso, compartilhe memória comunicando-se."O simples ato de comunicação garante a sincronização.

Concorrência em muitos ambientes se torna difícil pelas sutilezas envolvidas no acesso correto a variáveis compartilhadas. Go encoraja uma abordagem diferente, na qual variáveis compartilhadas são passadas através de canais e nunca são de fato ativamente compartilhadas por threads de execução separadas. Apenas uma gorotina tem acesso à variável num dado momento de tempo. Condições de corrida relativas aos dados não podem ocorrer, por construção. Modelo de concorrência inspirado no CSP [1].

2 Instalação

Passos que eu segui no Ubuntu 9.10:

```
cd ~
gedit .bashrc &
  # colar no final:
  export GOROOT=$HOME/go
  export GOARCH=386
  export GOOS=linux
# reiniciar o shell para que as alterações tenham efeito
sudo apt-get install bison gcc libc6-dev ed gawk make
sudo apt-get install python-setuptools python-dev build-essential gcc
sudo easy_install mercurial
hg clone -r release https://go.googlecode.com/hg/ $GOROOT
cd $GOROOT/src
mkdir $HOME/bin
./all.bash
cd $HOME/bin
sudo mv * /bin
```

2.1 Utilização

```
# criar o arquivo fonte num editor de sua preferência
gedit prog.go & # -> prog.go
# compilar
8g prog.go # -> prog.8
# ligar
8l -o prog prog.8 # -> prog
# executar
./prog
```

3 Mecanismos de Programação Concorrente

3.1 Channel

Representa um canal de comunicação / sincronização que pode conectar duas computações concorrentes. Na realidade são referências para um objeto que coordena a comunicação.

<- é o operador binário de comunicação (envio) <- também é o operador, desta vez unário, de recebimento. Ambos são atômicos.

Operações de leitura (respesctivamente, escrita) em canais que não têm buffer ou que estão com o buffer vazio (respectivamente, cheio) bloqueiam, e o bloqueio se mantém até que haja aconteça uma operação de escrita (respectivamente, leitura). A linguagem também permite leitura (respectivamente, escrita) não bloqueante, a qual retorna imediatamente uma flag dizendo se a operação foi realizada com sucesso ou não.

Na criação são sempre bidirecionais, mas quando são recebidos como parâmetro por uma função, além da declaração normal (chan T) podem ser declarados para apenas receber (<-chan T) e apenas enviar (chan<- T), com o objetivo garantir que serão utilizados corretamente no corpo da função.

Exemplos:

```
1  // criação de um canal sincrono para transmissão de inteiros
2  canal_sincrono := make (chan int)
3  // criação de um canal assincrono (enquanto houver espaço no buffer)
4  // com buffer de tamanho 10 (inteiros)
5  canal_assincrono := make (chan int, 10)
6  // função que utiliza um canal apenas para leitura
7  func so_leio (canal_leitura <-chan int) { ... }
8  // função que utiliza um canal apenas para escrita
9  func so_escrevo (canal_escrita chan<- int) { ... }
10 // função que utiliza um canal bidirecional
11 func leio_e_escrevo (canal chan int) { ... }
12 // tipos de leitura:
13 v := <-canal_sincrono  // sempre leitura sincrona
14 v := <-canal_assincrono  // leitura assincrona quando houver espaço no buffer, síncrona caso contrário
15 v, ok := <-canal  // sempre leitura assíncrona, ok é setado para true ou false de acordo com o sucesso
16 // tipos de escrita:
17 canal_sincrono <- v  // sempre escrita síncrona
18 canal_assincrono <- v  // escrita assíncrona quando houver espaço no buffer, síncrona caso contrário
19 ok := canal <- v  // sempre escrita assíncrona, ok é setado para true ou false de acordo com o sucesso
```

3.2 Gorotinas

Go tem seu próprio modelo de processo / thread / corotina, chamado gorotina (*goroutines*). Gorotinas são divididas de acordo com o necessário em threads (de usuário) e processos do sistema. Quando uma goroutine executa uma chamada de sistema bloqueane, nenhuma outra gorotina é bloqueada. Para configurar o número máximo de processos criados podemos modificar a variável de ambiente $GOMAXPROCS ou utilizar a função runtime.GOMAXPROCS (n int) durante a execução. Seu valor default é 1, ou seja, todas as gorotinas são threads de usuário pertencentes ao mesmo processo.

3.3 go

É o operador que inicia a execução concorrente de uma gorotina, sempre no mesmo espaço de endereçamento. Prefixe uma chamada de função ou de método com a palavra-chave go para executar a chamada numa nova gorotina. Quando a chamada termina, a gorotina também termina (silenciosamente). O efeito é similar a notação & do shell Unix para rodar um comando em background.

Exemplo:

```
...
func faz_algo () { ... }
...
func main () {
   ...
   // inicia a execução concorrente da função faz_algo ...
   go faz_algo ()
   // ... e continua executando ...
   ...
}
```

3.4 select

É um estrutura de controle análoga a um switch, mas que age sempre sobre comunicações. Escolhe qual das comunicações listadas em seus casos pode prosseguir. Se todas estão bloqueadas, ele espera até que alguma desbloqueie. Se várias podem prosseguir, ele escolhe uma aleatoriamente.

Exemplo:

```
// esta função envia zeros e uns aleatoriamente a cada iteração
func gerador_aleatorio_de_bits (canal chan<- int)
   for { // laço infinito
      select { // escolha não-determinística
         case c <- 0: // nada no corpo do case, o break é implícito (diferente de c)
         case c <- 1:
      }
   }
```

Exclusão Mútua: Criação: Uso: Destruição: Implementação: Problema produtor-consumidor com buffer limitado e taxa de produção e consumo aleatórias. Semáforos: Criação: Uso: Destruição: Implementação: Problema produtor-consumidor com buffer limitado e taxa de produção e consumo aleatórias. Monitores: Criação: Uso: Destruição: Implementação: Problema produtor-consumidor com buffer limitado e taxa de produção e consumo aleatórias. Variáveis de Condição: Criação: Uso: Destruição: Implementação: Problema produtor-consumidor com buffer limitado e taxa de produção e consumo aleatórias.

Processos, Threads ou Ambos?

Funções Úteis: time.Sleep(int) func init() http://golang.org/pkg/sync/ http://golang.org/doc/go_mem.html

Referências Bibliográficas

[1] Completar...