

1º LAB de CSC-27 / CE-288

CTA - ITA - IEC

Prof Hirata e Prof Juliana

Objetivo: Trabalhar com algoritmo de exclusão mútua para sistemas distribuídos.

Entregar (através do Google Classroom):

- Códigos dos exercícios (arquivos .go): cada aluno desenvolve os seus códigos e os entrega.
Obs: Se possível, favor NÃO compactar os arquivos.
 - Relatório: deve ser realizado em dupla e somente um aluno da dupla faz a entrega.
Obs: O relatório deve explicar particulares do código da tarefa solicitada, apresentar casos de testes realizados e comentar os resultados encontrados (comparando com os resultados esperados). Caprichem!
-

Preparação: Processos usando relógio lógico escalar

Você vai precisar de realizar essa etapa como parte do lab propriamente dito. Então a sugestão é atacar isso logo para garantir o correto funcionamento dessa parcela do trabalho.

- Utilize o código `Process.go`.
 - Considere o último código da Atividade Dirigida 1, pois ele já envia e recebe mensagens, ouve o teclado e trabalha com mensagens estruturadas.
- Considere que vamos rodar a simulação assim:
 - Exemplo com dois processos:
 - ✓ Terminal A: `Process 1 :10002 :10003`
 - ✓ Terminal B: `Process 2 :10002 :10003`
 - Exemplo com três processos:
 - ✓ Terminal A: `Process 1 :10002 :10003 :10004`
 - ✓ Terminal B: `Process 2 :10002 :10003 :10004`
 - ✓ Terminal C: `Process 3 :10002 :10003 :10004`
 - Nesse caso, temos sempre a mesma sequência de portas. Cada processo tem seu *id*. De acordo com o *id*, o processo sabe sua porta e a dos colegas. Nesse exemplo o processo 1 ‘escuta’ na porta 10002, o processo 2 ‘escuta’ na porta 10003, e o processo 3 ‘escuta’ na porta 10004.
 - Considere que *id* começa em 1.
 - ✓ Os ids são sempre valores consecutivos (adicionando 1).
 - Não precisa tratar erros relacionados a chamadas erradas (por exemplo, ids errados, e portas fora de sequência).
- Cada processo terá seu *clock* que inicia em 0.
 - Vamos usar relógio lógico escalar!
 - Acrescente um inteiro *clock* para representar o relógio do processo.
 - ✓ Por enquanto não vamos alterar o valor desse relógio. Vamos aguardar as instruções relativas ao algoritmo (que vamos implementar no lab) daí saberemos como mudar o relógio.
 - Acrescente na `struct Message` um inteiro `MsgClock` para representar o relógio lógico que vai na mensagem.

Tarefa: Algoritmo de Ricart-Agrawala para exclusão mútua.

Atenção: Esse algoritmo usa **relógio lógico ESCALAR**.

Obs: CS = *critical section*

Requisitos:

1) Vamos rodar o sistema como abaixo descrito. Neste exemplo, temos 3 processos (`Process.go`) e um recurso compartilhado (`SharedResource.go`):

- Terminal A: `SharedResource`
- Terminal B: `Process 1 :10002 :10003 :10004`
- Terminal C: `Process 2 :10002 :10003 :10004`
- Terminal D: `Process 3 :10002 :10003 :10004`

Obs: O algoritmo de Ricart-Agrawala é implementado no processo (`Process.go`).

2) O recurso compartilhado será representado por um outro processo, chamado `SharedResource.go`.

- Basicamente o `SharedResource` fica esperando alguém mandar uma mensagem para ele, através de uma porta fixa (no nosso caso, 10001).
- A mensagem recebida contém: o processo (*id*) que a enviou, o relógio lógico de quem a enviou, e um texto qualquer (ex: “Oi”).
- O código do `SharedResource` é simples como indicado abaixo.

```
func main() {
    Address, err := net.ResolveUDPAddr("udp", ":10001")
    CheckError(err)
    Connection, err := net.ListenUDP("udp", Address)
    CheckError(err)
    defer Connection.Close()
    for {
        //Loop infinito para receber mensagem e escrever todo
        //conteúdo (processo que enviou, relógio recebido e texto)
        //na tela
        //FALTA FAZER
    }
}
```

3) O processo (`Process.go`) deve “escutar” o teclado (terminal).

3.1) Caso receba do teclado um “x”, o processo deve solicitar acesso à CS, em seguida usar a CS e liberar a CS.

- Caso o processo receba “x” indevido, ele deve somente imprimir algo na tela (ex: “x ignorado”).
 - O “x” é indevido quando o processo já está na CS ou esperando para obter a CS.
- O controle para acessar a CS deve justamente ser feito com o Algoritmo de Ricart-Agrawala!

3.2) Caso receba do teclado o seu *id*, o processo executa uma ação interna (apenas um incremento do seu relógio lógico).

- Depois disso vai servir para testarmos o algoritmo com relógios distintos para os processos.

- 4) “Usar a CS” (em `Process.go`) significa que, após conseguir o acesso a CS, o processo deve:
- Escrever na sua própria tela “Entrei na CS”.
 - Mandar uma mensagem para o `SharedResource`.
 - Dormir um pouco. Isso simula que o processo segura a CS por um tempo.
 - Escrever na sua própria tela “Sai da CS”.
 - Só então, liberar a CS.
- 5) O processo deve sempre ser capaz de receber mensagens dos outros processos.
- Assim o processo pode estar esperando a CS, mas receber a mensagem de outro processo solicitando a CS.
 - Por isso precisamos das *goroutines*!

Convenção sobre como atualizar o clock:

O algoritmo de Ricart-Agrawala usa relógio lógico escalar, mas não indica onde/como atualizá-lo. Existem diferentes formas de atualizar o *clock*. Abaixo segue uma proposta. A ideia é não aumentar o *clock* sem necessidade, e garantir que o processo esteja atualizado em relação aos demais.

- a) Quando houver uma ação interna, incrementar o *clock*.
- b) Quando for enviar os *requests*, incrementar o *clock* uma vez apenas e mandar esse valor (T no algoritmo) numa mensagem para todos os demais processos. Obs: Todos amigos vão receber o mesmo valor T!
- c) Quando receber um *request*, atualizar o *clock* para ficar coerente com quem enviou: $I + \max(my\ clock, received\ clock)$.
- d) Quando enviar um *reply*, não incrementar o *clock*. Mande o seu *clock* atual mesmo.
- e) Quando receber um *reply*, atualizar o *clock* para ficar coerente com quem enviou: $I + \max(my\ clock, received\ clock)$.

Atenção:

- Provavelmente você terá diferentes *goroutines* alterando o *clock*, logo convém utilizar um **mutex** para garantir a correta manipulação desta variável.
 - Referências sobre mutex: <https://golangbot.com/mutex/> e <https://tour.golang.org/concurrency/9>
 - Outras variáveis de seu código podem precisar de mutex. Fique atento!

Casos de teste (para o relatório):

- **Caso 1:** Elabore um caso trivial com um processo solicitando a CS e, depois que ele liberar, outro processo solicita a CS.
- **Caso 2:** Elabore um caso em que um processo solicita a CS enquanto outro processo está usando a CS.
- **Caso 3:** Elabore um caso que caia especificamente na condição “`state=WANTED` and $(T, p_j) < (T_i, p_i)$ ”. A ideia é mostrar que um processo que pediu a CS antes, terá a preferência.

Para cada caso:

- Use três ou mais processos.

- Mostre o esquema (figura) do resultado esperado. A figura 1 é um bom exemplo disso.
- Pode ser que você precise ‘simular/forçar comportamentos’ dos processos com `Sleep`.
- Apresente o resultado obtido (*print* de suas telas).
- Explique/comente.

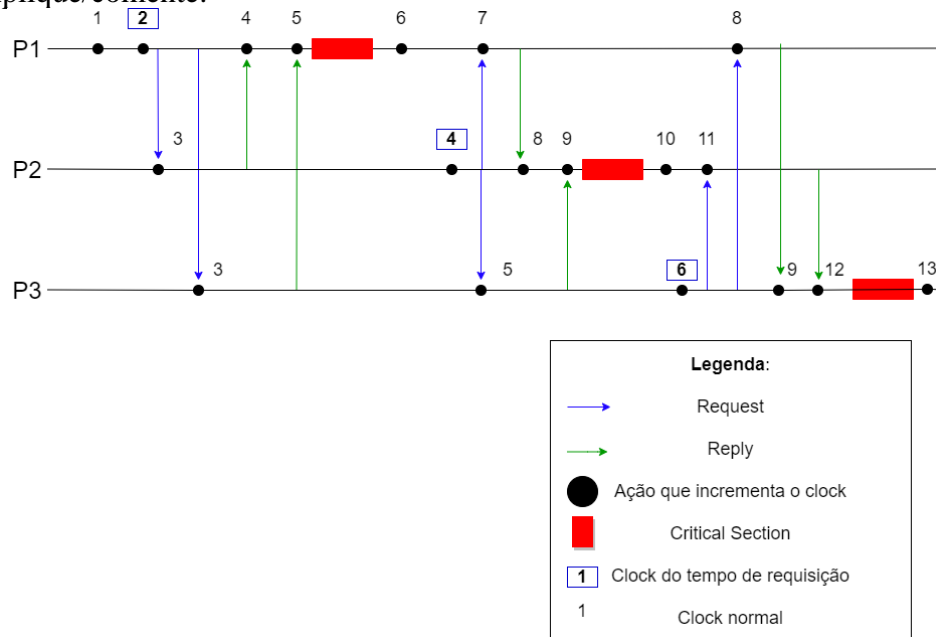


Figura 1: Exemplo de um esquema testado (seguindo nossa convenção de *clock*)

Dica: Relação entre algoritmo e código

