

Sistemas Distribuídos

Aula 16

Roteiro

- Exclusão mútua distribuída
- Algoritmo centralizado
- Algoritmo de Lamport
- *Token Ring*

Exemplo Bancário (Aula 4)

- Diferentes processos podem atualizar o saldo
 - executando em máquinas e locais diferentes
 - ex. `get_saldo`, `put_saldo` são chamadas RPC

```
retirada(conta, valor) {  
    wait(mutex)  
    saldo = get_saldo(conta)  
    saldo = saldo - valor  
    put_saldo(conta, saldo)  
    signal(mutex)  
    retorna saldo  
}
```

- O que pode acontecer?

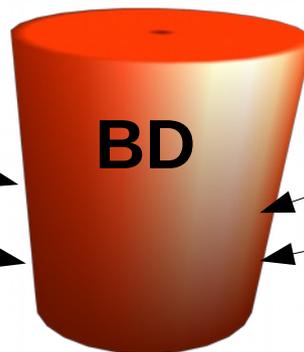
Condição de Corrida (entre processos)

- saldo pode não estar correto ao final

Exemplo Bancário

- Dois processos em máquinas diferentes, acessando BD remoto (em outra máquina)

```
P1
...
retirada(conta, valor) {
    wait(mutex)
    saldo = get_saldo(conta)
    saldo = saldo - valor
    put_saldo(conta, saldo)
    signal(mutex)
    retorna saldo
}
```



```
P2
...
retirada(conta, valor) {
    wait(mutex)
    saldo = get_saldo(conta)
    saldo = saldo - valor
    put_saldo(conta, saldo)
    signal(mutex)
    retorna saldo
}
```

- P1 e P2 podem estar dentro de suas respectivas regiões críticas, simultaneamente
- P2 pode dar `get_saldo` antes de P1 dar `put_saldo` (ou vice-versa)
 - apenas uma retirada será registrada

Condição de Corrida

- Cenário anterior: Múltiplas *threads* ou múltiplos processos no mesmo SO
- Condição de corrida resolvida com mecanismo de exclusão mútua
 - *locks* ou semáforos ou monitores
 - mecanismos assumem memória compartilhada (entre *threads* ou processos)
- Cenário atual: múltiplos processos em SO e máquinas diferentes
 - mecanismos anteriores não funcionam

Exclusão Mútua

- Condição de corrida é inerente em sistemas distribuídos
- Precisamos de exclusão mútua entre processos em máquinas diferente

```
...  
acquire(lock)  
// executa região crítica  
release(lock)  
...
```



- Como implementar exclusão mútua em sistemas distribuídos?

Trocando mensagens!

- única forma de coordenação

Demandas da Exclusão Mútua

- Algoritmo de exclusão mútua deve garantir algumas propriedades
 - **Corretude:** apenas um processo pode estar dentro da região crítica em cada instante
 - **Justiça:** qualquer processo que queira deve poder entrar na região crítica
 - implica que sistema não possui *deadlock*
 - Justiça eventual: eventualmente processo entra na região crítica

Demandas da Exclusão Mútua Distribuída

- Desejável que algoritmo de exclusão mútua ofereça
 - Baixo overhead de mensagens
 - Não possuir gargalos (e ponto único de falha)
 - Tolerar mensagens fora de ordem
 - Tolerar entrada e saída de processos
 - Tolerar perda de mensagens (pela rede)
 - Tolerar falha de processos

**Nada fácil garantir todas
essas propriedades!**

Algoritmo Centralizado



- **Ideias** para um algoritmo centralizado para exclusão mútua distribuída
- Coordenador: processo responsável por coordenar acesso a região crítica
 - comunicação e sincronização através de mensagens
 - utiliza fila para armazenar pedidos
- Processos
 - solicitam ao coordenador a entrada na região crítica
 - avisam ao coordenador ao saírem da região crítica (liberando o acesso)

Algoritmo Centralizado

Processo i

```
...
send(Coordenador, Request, i)
receive(Coordenador, Grant)

//
// executa região crítica
//

send(Coordenador, Release)
...
```

Coordenador

```
Q // fila de espera
while(1) {
    m = receive()

    if m.request
        if Q.empty
            send(m.process, Grant)
            Q.add(m.process)

    if m.release
        Q.remove()
        if !Q.empty
            process = Q.head()
            send(process, Grant)
}
```

■ Funciona?

■ Quem sincroniza chegada de pedidos no Coordenador

Propriedades do Algoritmo Centralizado

■ Demandas básicas

- **corretude**: garante acesso exclusivo, pois apenas um “Grant” por vez
- **justiça**: se política de fila for FIFO. Mas não se tivermos prioridade (ex. processo de menor índice tem preferência)

■ Desempenho

- Três mensagens por acesso a região crítica
- Request, Grant, Release
- Processos podem entrar/sair do sistema

■ Limitações

- ponto único de falha (coordenador)

Exclusão Mútua de Lamport

- Algoritmo distribuído para exclusão mútua
 - usando relógio de Lamport
- **Ideia:** todos os processos devem ter mesma ordem de entrada na RC
 - necessitamos de uma ordenação total dos pedidos de entrada na RC

Totally Ordered Multicast!

- Processos avisam quando querem entrar na RC
- Processos avisam quando saem da RC
- Acesso feito quando está na sua vez

Exclusão Mútua de Lamport

- Cada processo mantém uma fila com os pedidos de entrada na RC de todos os processos
- Fila ordenada por valor do relógio lógico associado ao pedido de entrada (*timestamp*)
 - relógio lógico de Lamport, adicionar identificador do processo (não há empates no valor de relógio)
- Para entrar na RC:
 - envia pedido com *timestamp* a todos processos (incluindo a si mesmo)
 - Aguarda confirmação de todos os processos
 - Se pedido estiver na cabeça da fila, e todas confirmações chegaram, entrar na RC!

Exclusão Mútua de Lamport

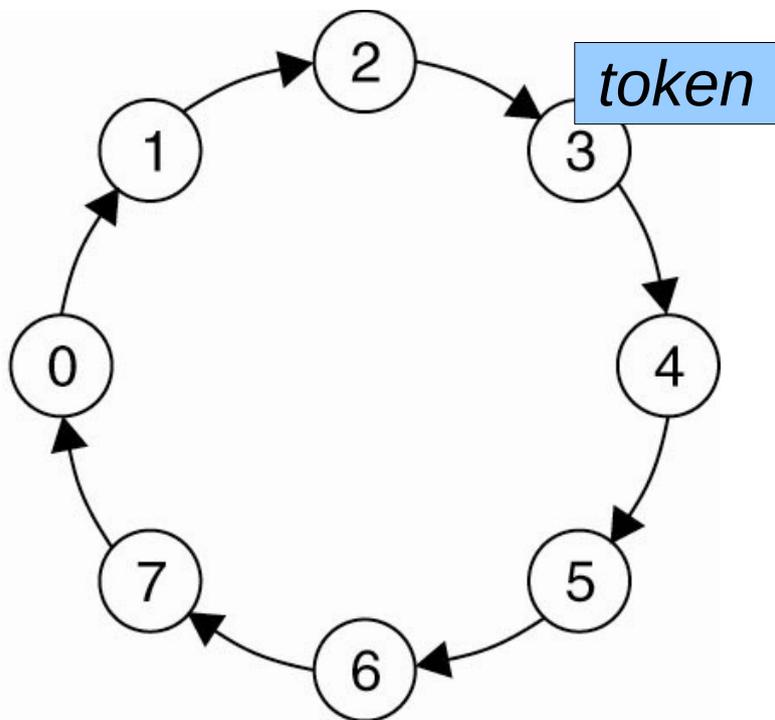
- Ao sair da RC:
 - Remover pedido da fila, enviar mensagem de *Release* a todos os processos
- Outros processos:
 - Ao receber um pedido, adicionar na fila ordenado pelo *timestamp* do pedido, enviar confirmação
 - Ao receber mensagem *release*, remover o pedido da cabeça da fila
 - Se próprio pedido estiver na cabeça da fila, e todas confirmações chegaram, entrar na RC!

Propriedades da Exclusão Mútua de Lamport

- Assumir rede FIFO e que mensagens não se perdem
- **Corretude:** Filas são ordenadas igualmente em todos os processos
- **Justiça:** ao enviar pedido em T1 e receber última confirmação em T2, todos os pedidos subsequentes que chegarem terão tempos maiores que T2
- Quantas mensagens (em *multicast*) por acesso a região crítica?
 - $1 + n + 1$: pedido, confirmação, release
- Limitações
 - se um processo falhar?
 - se mensagens trocarem de ordem na rede?

Algoritmo de *Token Ring*

- Exclusão mútua de Lamport exige comunicação entre todos os processos
- **Ideia:** Organizar processos em alguma topologia lógica, repassar mensagem (*token*) que dá acesso a região crítica
 - ex. topologia em anel (já vimos isto antes?)



- processo com *token* pode acessar RC, se necessário
- envia *token* para próximo processo ao sair da RC
- *token* circula pelos processos

Propriedades do *Token Ring*

- **Corretude:** *token* está em apenas em um processo em cada instante
- **Justiça:** acesso garantido antes de um mesmo processo acessar RC novamente (*token* circula)
- Quantas mensagens por acesso a região crítica?
 - depende do número de processos querendo acessar a RC
 - se todos: 1 mensagem por acesso, se apenas um: n mensagens por acesso
- Limitações
 - se um processo falhar?
 - se o *token* se perder?