

Sistemas Distribuídos

Aula 16

Aula passada

- Relógio de vetores
- Propriedades
- Garantindo ordenação total

Aula de hoje

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

Exemplo Bancário (aula 4)

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

```
retirada(conta, valor) {  
    saldo = get_saldo(conta);  
    saldo = saldo - valor;  
    put_saldo(conta, saldo);  
    retorna saldo;  
}
```

- O que pode acontecer?

Condição de Corrida!

- saldo pode não estar correto



Condição de Corrida

- Como resolvemos o problema antes?

Exclusão Mútua!

- Usando mutex ou semáforos



- Podemos reusar a ideia?

Não! Por que?

- Mutex/semáforos assumem acesso a memória compartilhada

- imediato com threads, bem difícil com processos

Exclusão Mútua

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

```
...  
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 necessita
 - **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

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

Algoritmo Centralizado



- **Ideias** para um algoritmo centralizado?
- Coordenador: processo responsável por coordenar acesso a região crítica
 - comunicação com mensagens
 - utiliza fila para armazenar pedidos
- Processos: solicitam entrada na região crítica ao coordenador; liberam região crítica avisando coordenador

Algoritmo Centralizado

Processo i

```
...  
send(Coordinator, Request, i)  
receive(Coordinator, Grant)  
  
//  
// executa região crítica  
//  
  
send(Coordinator, Release)  
...
```

Coordinator

```
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?

■ Multi-threaded?

Propriedades do Algoritmo Centralizado

■ Demandas básicas

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

■ Desempenho

- Quantas mensagens por acesso a região crítica?
- Três: Request, Grant, Release

■ Limitações

- Ponto único de falha (o que ocorre se coordenador falhar, ou rebootar)

Exclusão Mútua de Lamport

- Algoritmo distribuído de 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 de pedidos de entrada na RC
- Fila ordenada por valor do relógio lógico associado ao pedido de entrada
 - relógio lógico de Lamport, adicionar identificador do processo (não há empates no valor de relógio)
- Para entrar na RC:
 - envia pedido com *timestamp* (relógio local) 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

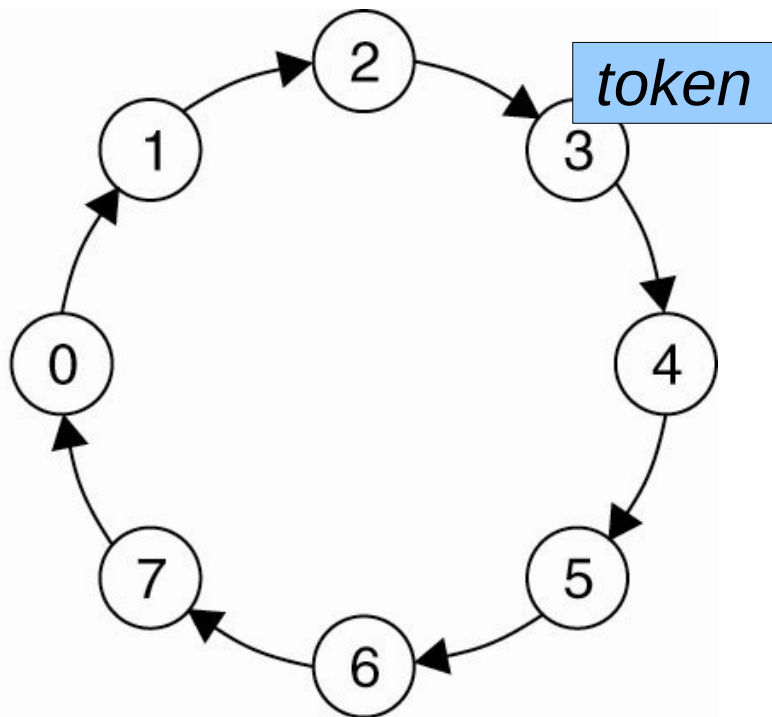
- Ao sair da RC:
 - Remover pedido da fila, enviar mensagem de *release* a todos os processos
- Outros processos:
 - Ao receber um pedido, adicionar na fila em ordenado pelo *timestamp* do pedido, enviar confirmação
 - Ao receber mensagem *release*, remover o pedido da cabeça da fila
 - Se próprio pedido 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 na fila deste processo terão tempos menores que T2
- Quantas mensagens por acesso a região crítica?
 - $3*(n-1)$: pedido, confirmação, release
- Limitações
 - se um processo falhar?
 - se mensagens trocarem de ordem na rede?

Algoritmo de *Token Ring*

- **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* acessa RC, se necessário
- envia *token* para próximo processo
- *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!
 - Se todos: 1 mensagem por acesso, se apenas 1: n mensagens por acesso
- Limitações
 - se um processo falhar?
 - se o *token* se perder?