Sincronização em Sistemas Distribuídos

FELIPE CUNHA

Sincronização de Processos

Processos Cooperativos:

- Afetam ou são afetados por outros processos.
- Podem compartilhar um mesmo espaço de endereços lógicos, ou um mesmo arquivo.
  - Endereços lógicos: threads, ou processos leves.

Acesso concorrente a dados pode gerar inconsistência de dados.

- Devem ser desenvolvidos mecanismos para garantir a consistência de dados compartilhados por processos cooperativos.

Introdução

O caso produtor/consumidor:

- Suponha processos com memória compartilhada
- Memória consiste em um buffer de tamanho variável que abriga itens em estoque. Variável counter indica o número de itens em estoque (itens no buffer.
- Toda vez que um produto é adicionado incrementa-se o counter.
- Toda vez que um produto é retirado decrementa-se o counter.

O caso produtor/consumidor

Código do produtor:

```
do
    ...
    produce an item in nextp
    ...
    while (counter == n)
      no-op;
    buffer[in] = nextp;
    in = in++ % n;
    counter++;
while (false)
```

O caso produtor/consumidor

Código do consumidor:

```
do
    while (counter == 0)
      no-op;
    nextp = buffer[out];
    out = out++ % n;
    counter--;
    ...
    consume the item in nextp
    ...
while (false)
```

O caso produtor/consumidor

Algoritmos corretos isoladamente.

Podem causar inconsistência se rodados em paralelo.

Problema: **Sessão Crítica!!!**

- Região de código onde um processo está acessando dados compartilhados.
- Problemas de inconsistência ocorrem em sessões críticas e devem ser evitados.

O problema da sessão crítica

Soluções para o problema da sessão crítica devem satisfazer os requisitos:

- Exclusão mútua: Se um processo está executando uma sessão crítica (acessando dados compartilhados) os demais processos devem esperar.
- Progressão:Se não há processos executando em sessão crítica, somente processos que não estão executando no final da sessão podem competir pela sessão crítica.
- Espera limitada:existe um limite no número de vezes que outros processos são permitidos a entrar na sessão crítica após m processo requisitar a sessão crítica.

Soluções de implementação

Algoritmo da vez:

- Vetor de flags e variável de vez

Algoritmo do padeiro (Bakery algorithm):

- Choosing e number

Semáforos:

- *wait(S)* => **while** (S <= 0) no-op;
  S--;
- *Signal(S)* => S++;

O caso produtor/consumidor com semáforos

Código do produtor:

```
...
produce an item in nextp
...
wait(empty);
wait(mutex);
...
add nextp to buffer;
...
signal(mutex);
signal(full);
```

O caso produtor/consumidor com semáforos

Código do consumidor:

```
wait(full);
wait(mutex);
...
remove nextc from buffer;
...
signal(mutex);
signal(empty);
...
consume the item in nextc
...
```

Sincronização em Sistemas Distribuídos

Conceitos importantes no projeto de qualquer aplicação distribuída:

- Comunicação
- Sincronização

Sincronização em aplicações distribuídas:

- Como garantir exclusão mútua no acesso a seções críticas ?
- Como garantir atomicidade na execução de uma transação distribuída ?
- Como alocar recursos evitando a ocorrência de deadlocks ?

Sincronização em Sistemas Distribuídos

Sincronização em sistemas centralizados: utiliza memória compartilhada

- Exemplos: semáforos, monitores etc

Sincronização em sistemas distribuídos: utiliza troca de mensagens

- Implementada via algoritmos distribuídos
- Mesmo determinar se um evento A ocorreu antes ou depois de um evento B é mais complexo do que em um sistema centralizado

Algoritmos Distribuídos

Principais propriedades de um algoritmo distribuído:

- Informações são distribuídas pelos vários nodos do sistema
- Processos tomam decisões baseados apenas nas informações locais
- Sem um ponto único de falhas
- Não existe um relógio global a todo sistema

Ordenação de Eventos

Sistema centralizado: trivial

- Relógio comum a todos os processos

Sistema distribuído: não é nada trivial

- Não existe uma fonte de tempo global a todos os processos
- Solução: Relação Happens-Before

Relação Happens-Before

Proposta por Lamport em um paper clássico:

- Lamport, L. Time, clocks and the ordering of events in a distributed system, CACM, vol. 21, pp. 558-564, july 1978.

Idéias básicas:

- Definir ordem de eventos apenas entre processos que interagem entre si
- Relógio lógico: o que importa é a ordem dos eventos e não o tempo físico em que os mesmos ocorreram

Relação Happens-Before

Notação: A→ B

- Lê-se que "A ocorreu antes de B"
- Significado: todos os processos do sistema concordam que primeiro ocorreu o evento A e então o evento B

Eventos que não são ordenados pela relação "→" são ditos concorrentes.

- Notação: A \\ B

Relação Happens-Before

Definição:

- Se A e B são eventos de um mesmo processo e A foi executada antes de B, então $A \rightarrow B$
- Se A consiste no evento de enviar uma mensagem para um processo e B é o evento de recebimento desta mensagem, então $A \rightarrow B$.
- Se $A \rightarrow B$ e $B \rightarrow C$, então $A \rightarrow C$ (transitividade)

Implementação da Relação Happens-Before

Algoritmo de Lamport

Utiliza o conceito de relógio lógico (C):

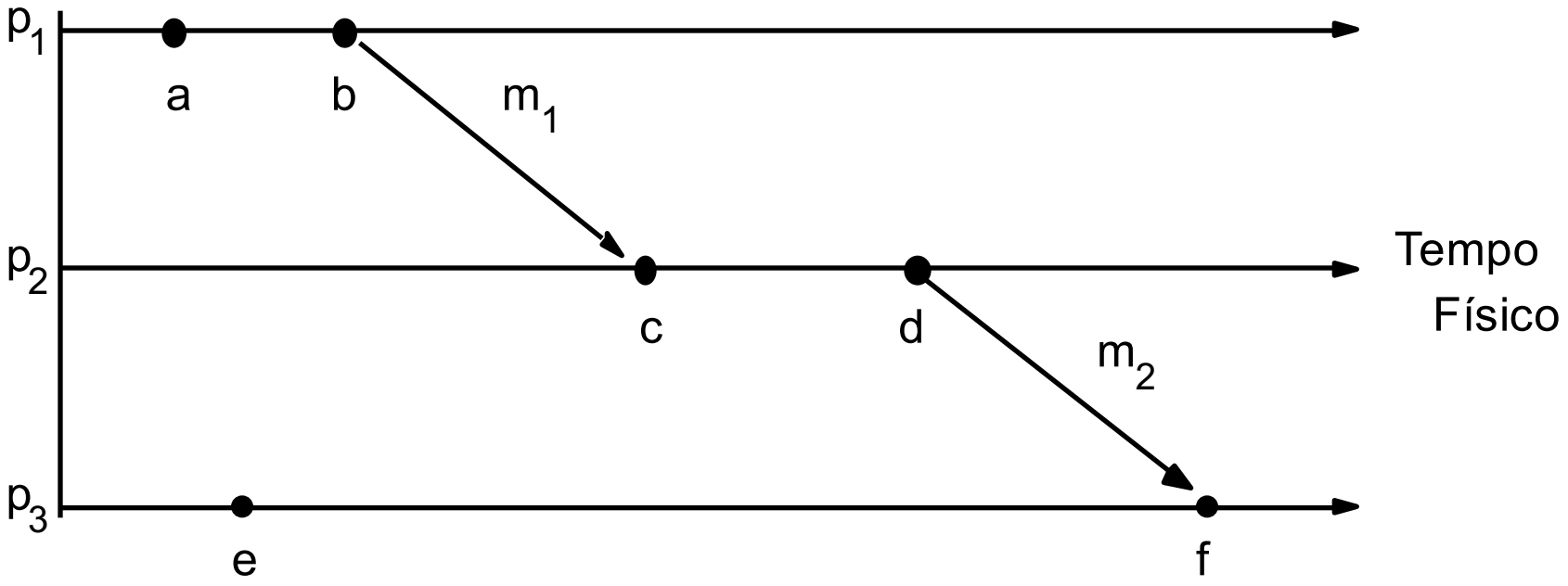
- Inteiro monotonicamente crescente armazenado em cada nodo
- Incrementado a cada evento

Implementação do conceito de relógio lógico:

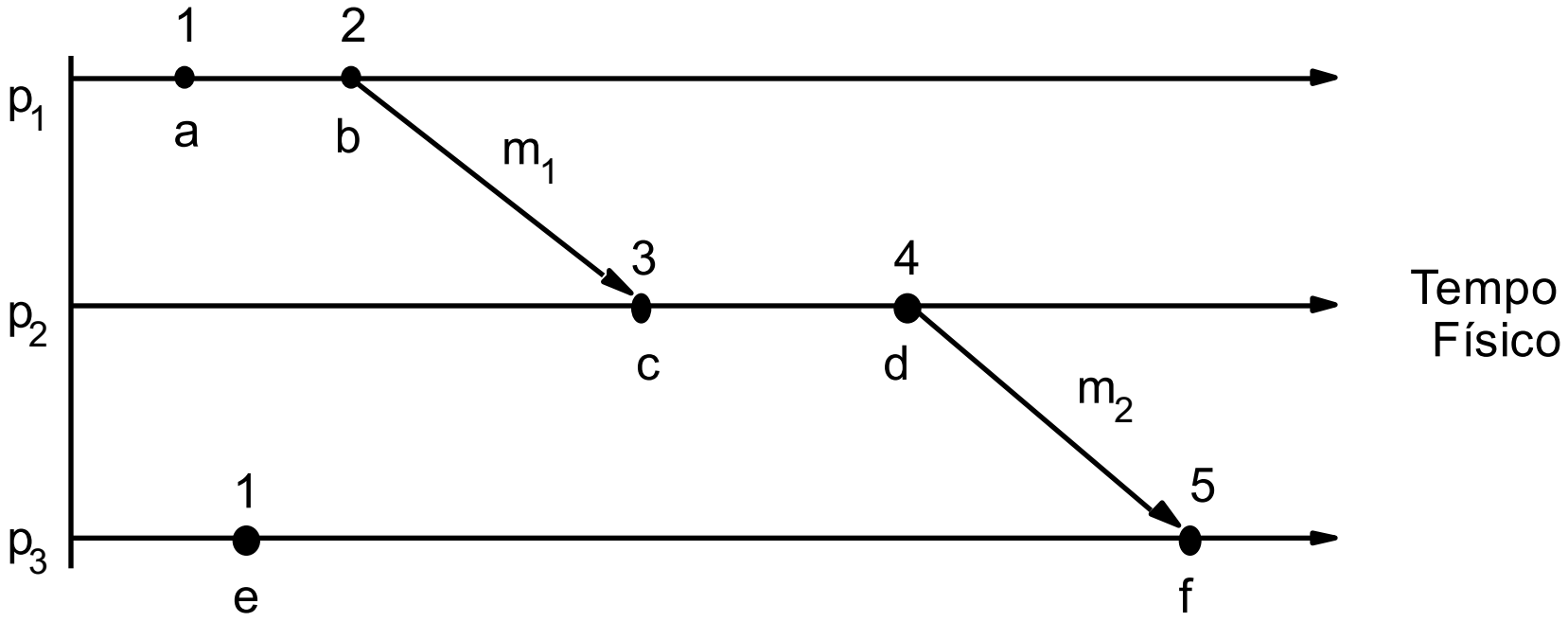
- Se $A \rightarrow B$

  então $C(A) < C(B)$

Ocorrência de eventos nos processos P1, P2 e P3



Rótulos de tempo do Algoritmo de Lamport para ordenação de eventos



Relógio Lógico

O valor do relógio lógico de um processo P, Cp, é atualizado da seguinte forma:

- Cp é incrementado antes de cada evento de P.
- Quando P envia uma mensagem M para Q, ele insere na mesma um timestamp t= Cp
- Quando Q recebe (M, t), então

  Cq:= max (Cq, t) + 1

  (garante-se assim que Cp < Cq)

Relógio Lógico

Relógio Lógico: como definido, dá origem a uma relação de ordem parcial

- Dois eventos podem ocorrer no mesmo tempo lógico

Ordenação total : adicionar o PID de cada processo no final do relógio lógico

- Exemplo:
  - Processo 1: relógio lógico = 40.1
  - Processo 2: relógio lógico = 40.2

Exclusão Mútua Distribuída

Sistemas Centralizados: memória compartilhada (semáforos, monitores etc)

Sistemas Distribuídos: via troca de mensagens

Requisitos:

- Segurança: No máximo um processo pode estar executando na seção crítica (SC)
- Liveness: Um processo que requisita acesso à SC, obtém este acesso em algum momento. Ou seja, não há deadlock ou starvation.

Primeira Solução: Algoritmo Centralizado

Existe um processo coordenado (PC)

- Gerencia o acesso à SC

Processos comuns

- Antes de entrar na SC: enviam request ao PC
- Quando recebem reply: entram na SC
- Quando saem da SC: enviam release ao PC

Processo Coordenador:

- Enfileira request quando a SC ocupada
- Desenfileira request ao receber um release

Primeira Solução: Algoritmo Centralizado (cont.)

Número de mensagens trocadas para entrar na SC: duas (um *request* e um *reply*)

Saída da SC: uma mensagem (*release*).

Problema: processo que centraliza todas as informações e decisões

Solução: distribuir a fila de pedidos

- Todos os nós recebem todos os pedidos
- Pedidos são ordenados pelos nós da mesma maneira

Algoritmos de Exclusão Mútua Distribuída

Principais Algoritmos:

- Lamport
- Ricart e Agrawala (*)
- Osvaldo e Roucariol (*)
- Maekawa
- Token Ring (*)

(*) Algoritmos que serão estudados a seguir

Algoritmo de Ricart e Agrawala

Algoritmo executado por cada processo Pi, que compartilha uma SC (i <= n):

- Variáveis:
  - estado: estado da seção crítica
  - OSN: relógio lógico do processo
  - HSN: maior valor do timestamp de qualquer mensagem enviada ou recebida
- Inicialização:
- estado:= livre;
- HSN:= 0;

Algoritmo de Ricart e Agrawala

Quando processo deseja entrar na SC:

estado:= aguardando;

OSN:= HSN + 1;

"Enviar requisição <OSN, i> para todos processos, exceto Pi"

"Espere até que o número de respostas seja igual a n-1

estado:= ocupado

Algoritmo de Ricart e Agrawala

Quando processo recebe requisição [k, j]:

HSN:= max (HSN, k)

se (estado = livre)

então "envie reply para Pj"

se (estado = ocupado)

então "enfileire requisição [k, j]"

se (estado = aguardando)

então se [OSN, i] < [k, j]

então "Enfileire requisição [k, j]

senão "Envie reply para Pj"

Algoritmo de Ricart e Agrawala

Ao sair da SC:

estado:= livre;

"Envie reply para todas requisições enfileiradas (e a retire da fila)"

Algoritmo de Ricart e Agrawala

Número de mensagens trocadas para entrar na SC: 2 (n -1)

- n-1: requests
- n-1: replys

Vantagem:

- Distribuído (sem um ponto central de falha)

Algoritmo de Carvalho e Roucariol

Algoritmo de Ricart & Agrawala:

- Número de mensagens trocadas: 2(n-1)
- "Ótimo, no sentido de que nenhum outro algoritmo distribuído e simétrico pode usar menos mensagens"

Algoritmo de Carvalho e Roucariol (1983):

- Mostraram que a afirmativa acima não é verdadeira
- Número de mensagens trocadas: entre 0 e 2(n-1)

Algoritmo de Carvalho e Roucariol

Idéia básica:

- Se Pi recebeu uma mensagem *reply* de Pj, então a autorização implícita nesta mensagem permanece válida até que Pi receba um novo *request* de Pj
- Enquanto isto não acontecer, Pi pode acessar a SC sem consultar Pj

Algoritmo de Carvalho e Roucariol

Análise:

- Melhor caso: processo que conseguir (n-1) autorizações pode acessar a SC sem enviar nenhuma mensagem (enquanto nenhum outro processo desejar acessar o recurso)
- Pior caso: entre um acesso à SC e o acesso seguinte, processo recebe (n-1) requisições de processos distintos. Logo, deverá enviar (n-1) *requests* para voltar a acessar a SC.

Algoritmo Token Ring

Supõe que os processos são organizados como um anel lógico, por onde circula uma *token*

- Não exige que a topologia da rede seja em anel (anel lógico e não físico)
- Cada processo deve armazenar a configuração completa do anel

Correção: ao processo de posse da *token* é garantido o acesso à SC

Algoritmo Token Ring

Idéia básica:

- Se um processo não deseja entrar na SC:
  - Ao receber a token, repassa a mesma para seu vizinho
- Se um processo deseja entrar na SC:
  - Aguarda a passagem da token e a retém

Número de mensagens trocadas: entre 1 e n-1

Algoritmo Token Ring

Vantagens: simplicidade

Desvantagem: tratamento de erros

- Perda da mensagem que contém a token
  - Deve-se gerar a token novamente
  - Quando gerar ? Quem deve gerar ?
- Travamento de processos
  - Deve-se reconfigurar o anel

Sincronização Física de Relógios

Relógios dos computadores atuais: podem atrasar ou adiantar com o tempo (clock drift)

- Razão: cristal de quartzo oscila a freqüências ligeiramente diferentes
- Clock drift médio: 1 seg a cada 11.6 dias

Sincronização Física de Relógios:

- Objetivo: manter os relógios físicos de um conjunto de máquinas "acertados"
  - $|Ci - Cj| < \varepsilon$, para todo i e j

Sincronização Física de Relógios

Aplicações: sistemas de tempo real ou de missão crítica

Tempo oficial (padrão internacional):

- UTC: Universal Coordinated Time
- Fornecido por relógios atômicos
- Várias fontes: estações de rádio, satélites etc

Algoritmo de Cristian

Proposto em 1989

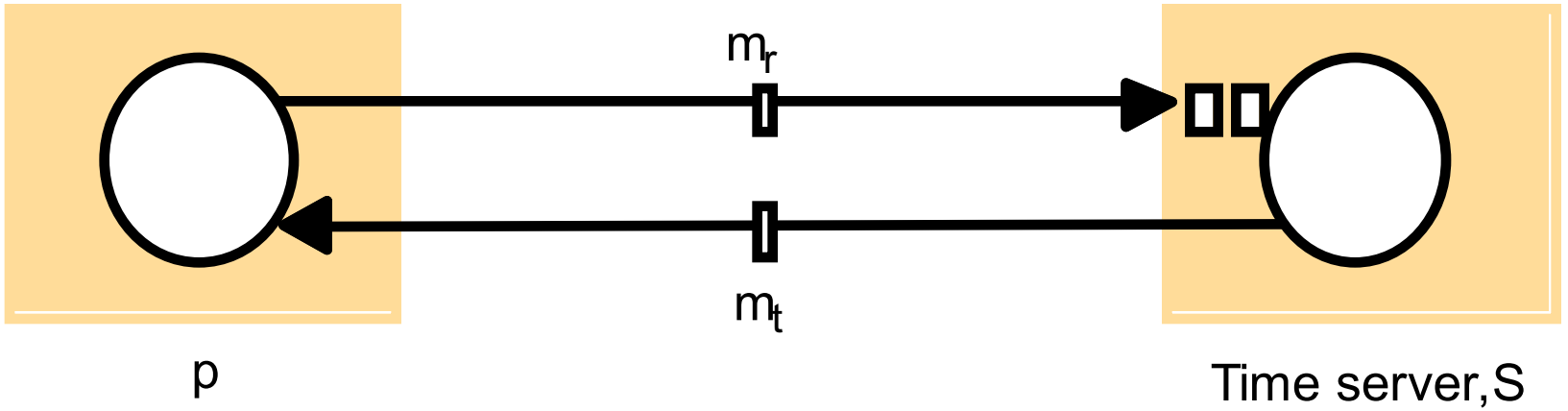
Existem dois tipos de estações:

- Um servidor de tempo (time server)
- Clientes deste servidor de tempo

Idéia básica:

- Clientes periodicamente requisitam o tempo ao servidor
- Tempo do servidor: CUTC

Sincronização usando um servidor de tempo



Algoritmo de Cristian

Problema 1: $C_{UTC} < C_{cliente}$ (ou seja, cliente está adiantado)

- Pode trazer conseqüências inesperadas para algumas aplicações
- Solução: "atrasar" o relógio gradativamente, até que a correção seja implementada

Algoritmo de Cristian

Problema 2: Como calcular o tempo de propagação da msg do servidor até o cliente ?

- T0: tempo em que o *request* foi enviado
- T1: tempo em que o *reply* foi recebido
- I: tempo médio de processamento no servidor
- TP (tempo de propagação) ≈ (T1 - T0 - I) / 2

- Novo tempo do cliente= C_{UTC} + TP

Algoritmo de Cristian

Implementação Prática:

- NTP (Network Time Protocol, 1991)
- Padrão Internet para sincronização de relógios (RFC 1129)
- Baseado em UDP
- Hierarquia de servidores (primários, secundários etc)

Algoritmo de Berkeley

Usado no Berkeley Unix (1989)

Servidor consulta clientes periodicamente (*polling*), os quais informam os valores correntes de seus relógios

Baseado nas respostas, servidor computa uma média e a envia de volta aos clientes

Adequado para sistemas que não possuem uma fonte UTC

Algoritmos de Eleição

Grupo de processos deve eleger um coordenador.

Aplicações:

- Gerenciamento de réplicas,
- Algoritmos centralizados de coordenação, sincronização, etc...

A escolha do coordenador deve ser única, mesmo que vários processos de eleição tenham começado simultaneamente.

Algoritmo Tirano (Valentão) The bully algorithm

Membros do grupo devem conhecer a identidade e o endereço dos outros membros.

Mensagens:

- Eleição: enviada para anunciar o início de uma nova eleição
- Resposta: enviada em resposta a uma mensagem de eleição.
- Coordenador: enviada para anunciar o coordenador.

Um processo inicia uma eleição quando notar uma falha no coordenador (não recebeu uma resposta a uma requisição depois de um certo número de tentativas ou após um certo timeout, por exemplo).

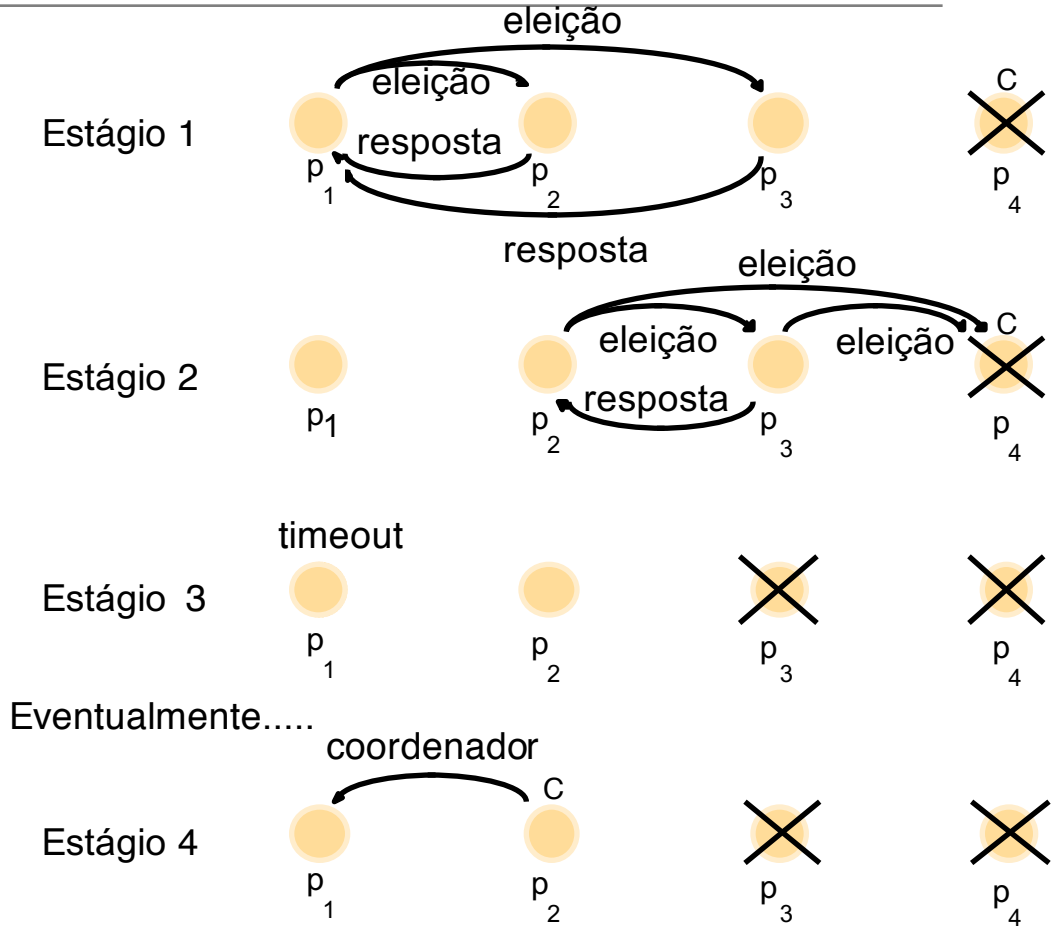
Algoritmo Tirano
The bully algorithm

Processo inicia eleição enviando mensagem de eleição para os processos de maior prioridade.

Processos respondem a mensagem e iniciam nova eleição

Processo coordenador envia mensagem de coordenador.

Se nenhuma mensagem for recebida, processo se declara coordenador e envia mensagem de coordenador para os processos de menor prioridade.



Deadlocks em sistemas distribuídos

Mais difíceis de tratar.

Estratégias para lidar com deadlocks:

- Algoritmo de Ostrich-ignore o problema (muito ineficiente)
- Detecção-deixa deadlocks ocorrerem e depois tenta recuperar estado
- Prevenção-estaticamente torne deadlocks estruturalmente impossíveis
- Evitar-evitam deadlocks alocando recursos com cuidado (nunca usado por ser extremamente complexo).

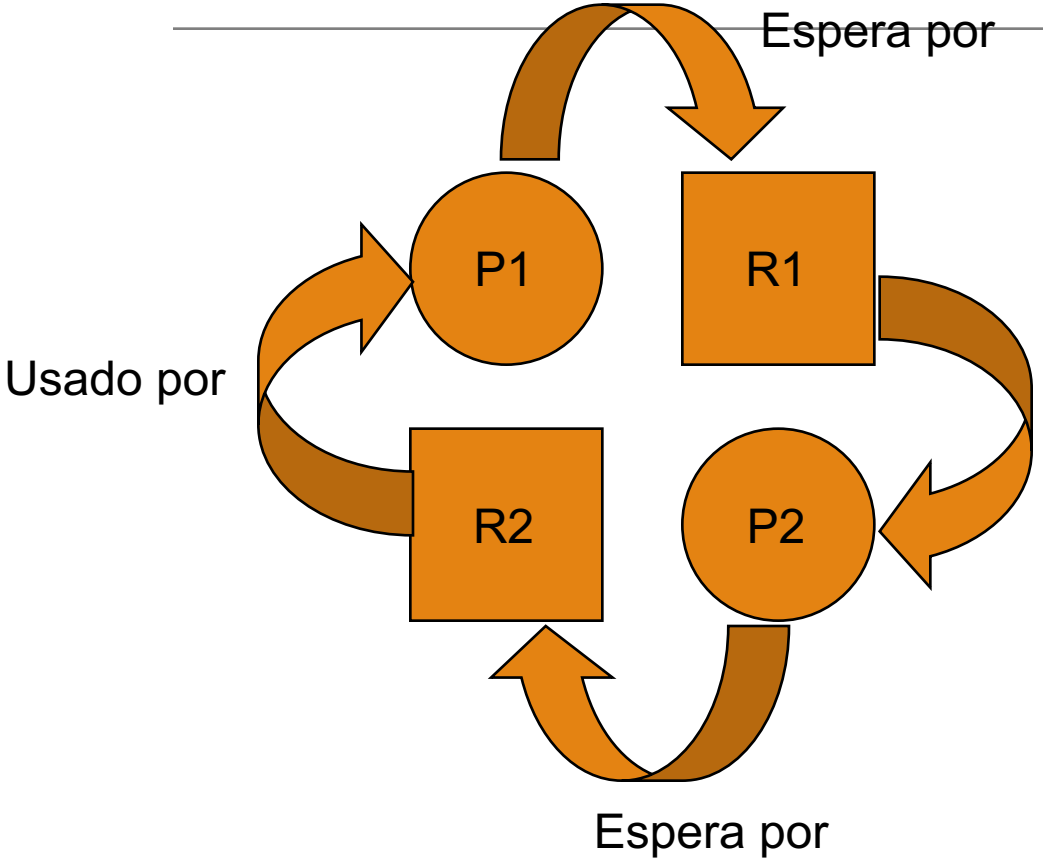
Detecção de deadlocks

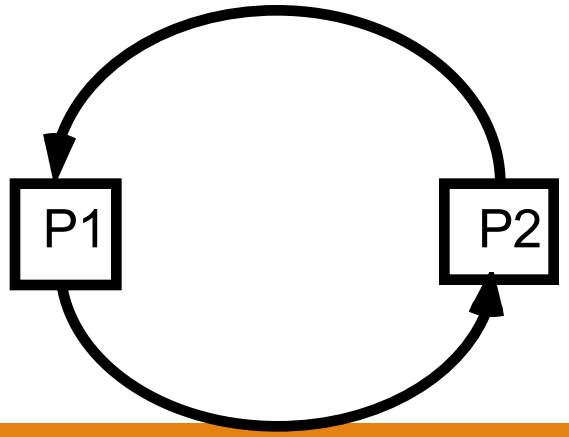
Caso um deadlock é detectado, algumas transações atômicas que ocasionaram o deadlock são destruídas.

Método Centralizado:

- Cada máquina possui o seu grafo de alocação de recursos e envia para o coordenador.
- Caso um ciclo seja detectado pelo coordenador, um processo deve cancelar a espera ou utilização do de um recurso.

Detecção de deadlocks





Detecção de deadlocks

Método Centralizado:

- Problema: Ocorrência de falsos deadlocks devido a atrasos em mensagens.
- Solução: Usar algoritmo de Lamport para tempo global. Caso ciclo seja detectado, coordenador pede por grafos atualizados.

Detecção de deadlocks

Método Distribuído:

- Algoritmo de Chandy-Misra-Haas
- Processos podem pedir múltiplos recursos.
- Processos enviam mensagens para os processos pelos quais que estão esperando. Se um loop for detectado, processos são eleitos para liberarem recursos.

Eliminando deadlocks

Processo que iniciou a prova do deadlock comete suicídio.

- Problema: se vários processos iniciam provas em paralelo podem ter morte em massa.

Incluir ID do processo na prova. Processo de maior número (mais novo) deverá cometer suicídio.

Problemas: Mandar e analisar provas quando se está bloqueado não é trivial.

Prevenção de deadlocks

Tornar dealocks impossível estruturalmente!

Método 1:

- Cada processo só pode usar um recurso de cada vez e deve liberar o recurso a fim de pedir outro.

Método 2:

- Recursos são ordenados de forma global. Processos devem pedir recursos em ordem crescente.

Prevenção de deadlocks

Método 3:

- Algoritmo wait-die;
- Utilizar tempo global.
- Processos só podem esperar por processos mais novos (tempo global maior).
- Caso um processo jovem precise de um recurso retido por um processo velho ele deverá cometer suicídio.