Normalmente são diversos processos em execução e em máquinas diferentes que precisam trabalhar em conjunto para chegar a um objetivo comum. Assim, nessa aula observa-se as implementações para garantir coordenação e acordo entre processos.

---

**Introdução**

- Computadores precisam coordenar suas ações corretamente.
  É importante que seja sincronizado e coordenado, concordando sobre as situações que devem ser tomadas.
- Suposições
  - Canais confiáveis

  Não há perda de mensagens. Toda mensagem enviada é entregue. Pode-se usar o tcp.
  - Nenhum processo depende de outro para encaminhar mensagens

Cada processo envia as mensagens no momento que quiser, sem dependência.

  - Possibilidade de particionamento da rede

Uma parte da rede pode não conseguir se comunicar com a outra e mesmo assim o sistema de coordenação não deve parar.

  - Conectividade assimétrica

Se existir um processo A pode se comunicar com B, mas o inverso pode não ser verdade. **Exemplo:** O usuário se conecta ao servidor e o servidor não se conecta ao usuário.

  - Conectividade intransitiva

Comunicação onde A envia para B e B envia para C mas não necessariamente C envia para A.

  - Processos só falham por causa de defeitos

Defeitos em computador, rede, etc.

---

**Exclusão mútua distribuída**

Algoritmos de exclusão mútua

Tal como em sistemas operacionais, dois processos não podem executar simultaneamente suas regiões críticas pois podem gerar inconsistências no sistema. Assim, usa-se estratégia de exclusão mútua, mas as estratégias estudadas (semáforo, mutex, etc) antes servem apenas para uma única máquina, sendo necessário

para sistemas distribuídos usar as estratégias de coordenação.

- Suposições
  - Assíncrono

Os processos podem trabalhar de maneira independente.

  - Processos não falham
  - Envio confiável de mensagens

Usando por exemplo o TCP.

- Funções
  - enter() - entra na região crítica

Sempre que um processo quiser executar sua região crítica deve executar essa função e só deve executar quando a mesma retornar um "ok".

  - resourceAccesses() - acessa recursos

Permite acessar os recursos assim que se entra na região crítica.

  - exit() - sai da região crítica

Para sair da região crítica.

---

**Exclusão mútua distribuída**

Algoritmos de exclusão mútua

- Requisitos
  - EM1: Segurança

No máximo um único processo execute sua região crítica.

  - EM2: Subsistência

Os processos devem obter sucesso ao entrar e sair da região, sem impasses.

  - EM3: Ordenação

A Ordem de Execuções de região segue a ordem de requisições.

- Critérios para avaliar o desempenho
  - Largura de banda consumida

Quanto menor, melhor o algoritmo.

  - Atraso do cliente

Atraso causado por cada entrada e saída na região crítica.

  - Atraso de sincronização

Relacionado ao tempo de saída de um processo e entrada de outro, interferindo na vazão do sistema.

---

**Algoritmo do servidor central**

- Um servidor é responsável por controlar o acesso à RC

Um servidor controla o acesso à região crítica, tendo de posse um token. Sempre que um processo quer acessar sua região crítica, ele deve solicitar o token ao servidor e enquanto estiver usando, ele terá acesso exclusivo à região. Após usar, ele deve devolver o token ao servidor.

  - Uso de um token que é entregue ao processo que vai acessar a RC.

- Satisfaz Segurança e Subsistência, mas não satisfaz Ordenação

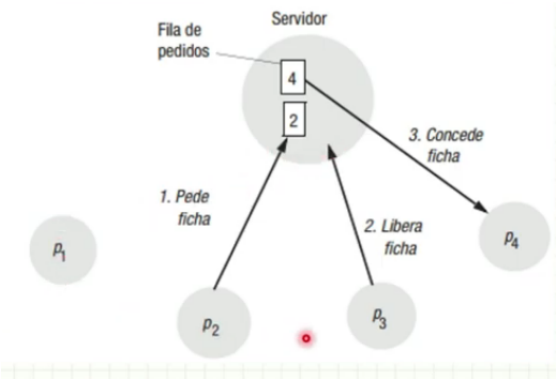
Um único processo tem acesso à região crítica por vez, portanto é seguro e sem impasses.
Não satisfaz a ordenação pois pode haver atrasos nas requisições de token ao servidor.

- Desempenho: O servidor pode se tornar um gargalo

Um único servidor controla todos os processos, podendo tornar lento o sistema.

---

**Algoritmo do servidor central**




---

**Algoritmo baseado em anel**

- Processos são organizados em um anel lógico.

Existe também o token, que contém o direito de acesso, o token viaja no anel entre os processos (ele verifica se o processo precisa executar sua região), caso algum processo precise executar ele deve esperar o token "chegar".

- O acesso é concedido através de um token passado de processo para processo em uma única direção
- Segurança e subsistência são garantidos, mas não necessariamente ordenação.

Os processos devem esperar sua vez, não garantindo ordenação.

- Consome largura de banda continuamente.

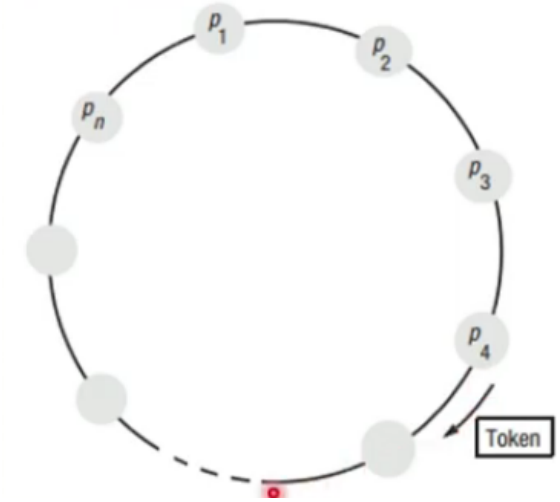
O token viaja de maneira indefinida durante o funcionamento.

- Atraso de um processo que solicitou entrar na RC varia entre 0 e N mensagens.

O atraso pode ser 0 se o processo já estiver de posse do token e N no pior caso, se o processo ainda tiver que esperar o token percorrer todo o anel.

---

**Algoritmo baseado em anel**



## Algoritmo usando multicast

Cada processo possui um id e um relógio lógico, sempre que um processo quiser entrar é feito um multicast, contudo só pode executar quando todos os que receberam a mensagem responderem que ele pode.

- Cada processo possui um identificador único e mantém um relógio lógico.
- Um processo que queira entrar na RC faz um multicast de requisição e só pode entrar quando todos responderem a esta requisição.
- Segurança, subsistência e Ordenação são garantidos.

Único que garante todos os requisitos.

- Estados dos processos:
  - RELEASED - Fora da RC
  - WANTED - Querendo entrar
  - HELD - Dentro da RC

## Algoritmo usando Multicast

*Na inicialização*
*state* := RELEASED;

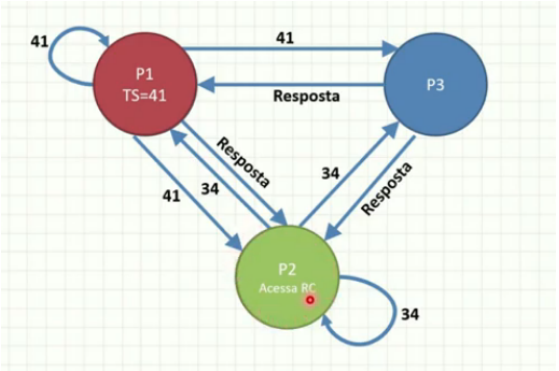
*Para entrar na seção*
*state* := WANTED;
Envia a requisição por *multicast* para todos os processos;
*T* := carimbo de tempo da requisição;
*Espera até* (número de respostas recebidas = (N – 1));
*state* := HELD;

*Processamento da requisição referido aqui*

*No recebimento de uma requisição* $<T_i, p_i>$ *em* $p_j$ $(i \neq j)$
*if* (*state* = HELD *or* (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
*then*
enfileira requisição de $p_j$ sem responder;
*else*
responde imediatamente para $p_j$;
*end if*

*Para sair da seção crítica*
*state* := RELEASED;
responde a todas as requisições enfileiradas;

## Algoritmo usando Multicast




O menor TimeStamp "vence" e executa a região crítica, depois libera e manda uma mensagem para quem "perdeu".

## Algoritmo usando multicast

- Maior dispêndio de largura de banda que os anteriores.

Pelos envios de mensagens.

- O atraso no cliente é o tempo de viagem de ida e volta da mensagem.

- O atraso de sincronização é apenas o tempo de envio de uma mensagem (Libera RC).

É muito pequeno.

## Algoritmo de votação de Maekawa

- Os processos só precisam obter permissão de subconjuntos de seus pares para entrar na RC.

Diferente do multicast, ele precisa da resposta de apenas um subconjunto de seus pares.

- Os subconjuntos usados por quaisquer dois processos devem se sobrepor.

Um mesmo processo deve pertencer a ao menos dois subconjuntos.

- Um processo candidato deve reunir votos suficientes para entrar.

A quantidade necessária é a dos elementos que pertencem ao mesmo subconjunto daquele processo.

- Os processos na intersecção de dois conjuntos de votantes garante segurança (EM1).

O processo da interseção responde apenas a um dos processos que requisita uma resposta.

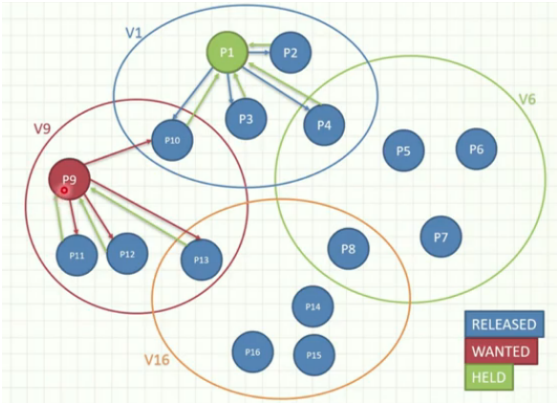
---

**Algoritmo de votação de Maekawa**

- V; - conjunto de votação de um processo p;
- V, está contido no conjunto total de processos p;
- Vj interseção com Vi não é vazio: pelo menos um membro comum de quaisquer conjuntos votantes.
- |vjl= K: Cada processo tem um conjunto votante de mesmo tamanho.
- p, não pode entrar na RC até que tenha recebido todas as K mensagens de resposta.

---

**Algoritmo Maekawa**




- P10 decide com base no timestamp?
- Pode explicar melhor a formação dos subconjuntos?

---

**Algoritmo de votação de Maekawa**

- Algoritmo é propenso a impasses.

O requisito de subsistência nem sempre é levado em consideração.

Pode haver dependência de resposta entre processos.

- Exige mais mensagens que o de multicast.

Existem vários multicasts.

- Mesmo atraso no cliente do multicast.

Deve-se aguardar a recepção de todas as respostas.

- Atraso de sincronização é pior que o de multicast.

---

**Algoritmo de votação de Maekawa**

*Na inicialização*
*state* := RELEASED;
*voted* := FALSE;

*Para $p_i$ entrar na seção crítica*
*state* := WANTED;
Envia a *requisição* por *multicast* para todos os processos em $V_i$;
*Espera até* (número de respostas recebidas = $K$);
*state* := HELD;

*No recebimento de uma requisição de $p_i$ em $p_j$*
*if* (*state* = HELD *ou voted* = TRUE)
*then*
enfileira a requisição de $p_i$ sem responder;
*else*
envia resposta para $p_i$;
*voted* := TRUE;
*end if*

*Para $p_i$ sair da seção crítica*
*state* := RELEASED;
Envia a *liberação* via *multicast* para todos os processos em $V_i$;

*No recebimento de uma liberação de $p_i$ em $p_j$*
*if* (fila de requisições não estiver vazia)
*then*
remove cabeça da fila – digamos, $p_k$;
envia *resposta* para $p_k$;
*voted* := TRUE;
*else*
*voted* := FALSE;
*end if*

---

**Tolerância a falhas**

- Questionamentos sobre os algoritmos:
  - O que acontece quando as mensagens são perdidas?
  - O que acontece quando um processo falha?
- Respostas:
  - Nenhum dos algoritmos toleraria perda de mensagens.
    Se uma mensagem é perdida, todo o algoritmo é impactado, daí surge a necessidade de confiabilidade.
  - Tolerariam algumas falhas de processos.
    Depende do caso, no Maekawa, se um processo que não está dentro do subconjunto votante falhar não afeta o funcionamento.
    No caso do servidor central, se um processo que não solicitou e nem possui o token falhar, não tem problema também.

---

**Eleições**

- Algoritmo usado para escolher um único processo para desempenhar uma função em particular.
- Exemplo: No algoritmo do servidor central ser escolhido para um processo deve desempenhar a função de servidor.

Nos algoritmos de eleição um processo **pode convocar uma eleição por vez**, porém **podem existir n eleições concorrentes**, pois se tem n processos.

Um processo **pode ou não participar de uma eleição** e cada processo tem uma **variável chamada eleito** que identifica o processo eleito e inicialmente ela é vazia.

---

**Eleições**

- Requisitos
    - E1: Segurança

Relacionada ao fato de que um processo participante de uma eleição tem sua variável eleito vazia ou p (onde p é o eleito do sistema, nó com maior identificador).

    - E2: Subsistência

Todos os processos configuram a variável eleita diferente de vazio ou os processos falham.

- Desempenho:
    - Uso de Largura de banda

Devem minimizar o uso da largura de banda.

    - Tempo do ciclo de rotação: soma dos tempos de transmissão de mensagens.

Tempo necessário para todas as mensagens de eleição chegarem.

**A escolha deve ser única, todos devem chegar ao mesmo resultado do eleito.**

---

**Algoritmo de eleição baseado em anel**

- Cada processo p; tem um canal de comunicação com o processo seguinte
- Mensagens são enviadas no sentido horário do anel
- Suposições: Não ocorrem falhas e sistema assíncrono.

Os processos nunca devem falhar e não há controles de tempo.

---

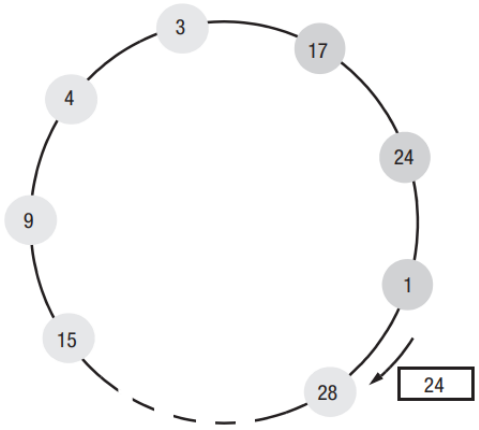
**Algoritmo de eleição baseado em anel**

- Inicialmente cada processo é marcado como não participante.
- Qualquer processo pode iniciar uma eleição.
- Quem inicia, marca a si mesmo como participante.
- Coloca seu identificador na mensagem de eleição e encaminha ao vizinho.
- O receptor compara o seu identificador com o da mensagem:
    - Se o id da mensagem que chegou é maior,

encaminha a mesma para o próximo vizinho.
    - Se o id é menor e o receptor não é participante, substitui por seu próprio id na mensagem e repassa.
    - Se já participante, não encaminha

---

**Algoritmo de eleição baseado em anel**




---

**Algoritmo de eleição baseado em anel**

- Quando o id na mensagem é igual ao do receptor este se tornará o vencedor.
- A seguir envia uma mensagem de eleito anunciando ao vizinho.
- Esta mensagem se propaga entre todos os vizinhos que mudam seus estados para não participantes.
- A condição de segurança é satisfeita (E1).

Todo mundo tem sua variável eleita configurada.

- A condição de Subsistência é satisfeita (E2).

Visto que não existem falhas então ao final todos recebem a mensagem com o processo eleito.

---

**Algoritmo do valentão**

- Permite que processos falhem

É mais robusto por isso.

- Presume confiabilidade no envio de mensagens.

Todas as mensagens são entregues.

- Sistema síncrono

Usa o tempo limite para detecção de falha dos processos.

- Cada processo sabe quais processos têm id maior e pode se comunicar com todos esses processos

Não há a restrição de se comunicar apenas com vizinhos.

---

**Algoritmo do valentão**

- Tipos de mensagens:
    - eleição: Convoca uma eleição.

Enviada quando se quer iniciar uma eleição

    - resposta: resposta a mensagem anterior.

    - coordenador: anuncia o id do eleito.
- Uma eleição é iniciada quando o tempo limite de resposta do coordenador estourou.

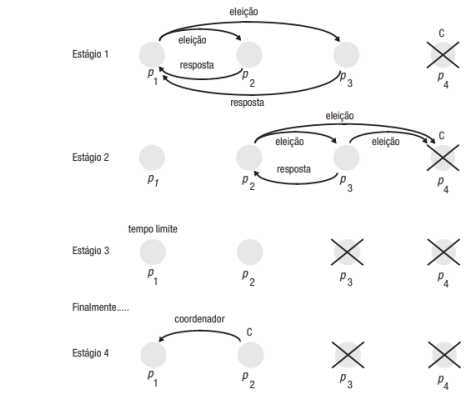
Sempre vai haver um coordenador mas se a comunicação com ele não puder ser estabelecida, inicia-se uma nova eleição.

- o processo com maior id pode eleger-se simplesmente enviando uma mensagem de coordenador a todos os outros processos

Quando um processo retorna da falha, ele automaticamente manda uma mensagem se anunciando coordenador.

---

**Algoritmo do valentão**




A eleição do coordenador $p_2$, após a falha de $p_4$ e, depois, de $p_3$.

---

**Algoritmo do valentão**

- Satisfaz E1, se nenhum processo for substituído por outro de mesmo identificador, pois é impossível dois processos decidirem simultaneamente que são o coordenador
- Satisfaz E2, pois supõe o envio confiável de mensagens.