

Faculdade de Informática - PUCRS

---

## ALGORITMOS DISTRIBUÍDOS

### Exclusão mútua

Sistemas Distribuídos

184

### Algoritmos Distribuídos (exclusão mútua)

- **Problema:** alguns recursos não podem ser usados simultaneamente por diversos processos (ex.: arquivos)
- Exclusividade de acesso deve ser garantida pelo sistema
  - esta exclusividade é conhecida como *exclusão mútua*
- Um algoritmo que implementa exclusão mútua deve satisfazer os seguintes critérios:
  - *exclusão mútua*: dado um recurso compartilhado que pode ser requisitado por diversos processos ao mesmo tempo, **somente um processo pode acessar aquele recurso** a qualquer momento
  - *no-starvation*: cada processo que requisita o recurso deve recebê-lo em algum momento
- **Seção crítica:** parte do código dos processos em que é realizado o acesso a algum recurso compartilhado que deve garantir acesso exclusivo.

Sistemas Distribuídos

185

### Algoritmos Distribuídos (exclusão mútua)

- **Algoritmo centralizado**
  - Neste algoritmo, um processo do sistema é eleito como o coordenador e coordena as entradas na seção crítica (SC)
  - Cada processo que deseja entrar em uma SC deve antes pedir autorização para o coordenador
  - Se não existe processo acessando a SC, então o coordenador pode imediatamente garantir acesso ao processo que fez a requisição
  - Se mais de um pede acesso à SC, então só um ganha acesso
  - Após término do uso, processo informa coordenador
  - Coordenador pode então liberar SC para outro processo (se existir)

Sistemas Distribuídos

186

### Algoritmos Distribuídos (exclusão mútua)

- **Algoritmo Distribuído**
  - Se processo deseja acessar SC, então ele envia mensagem para todos os outros processos
  - Mensagem contém:
    - identificador do processo
    - nome da SC que ele deseja acessar
    - um *timestamp* único gerado pelo processo que enviou a mensagem

Sistemas Distribuídos

187

### Algoritmos Distribuídos (exclusão mútua)

- Ao receber mensagem, processo:
  - **responde** ao processo que enviou msg e garante acesso à SC se:
    - não quer acessar SC
    - quer acessar SC mas seu *timestamp* é maior que o *timestamp* do processo que enviou a mensagem
  - **não responde** se:
    - processo que recebeu mensagem está executando na SC
    - processo está esperando para acessar SC e seu *timestamp* é menor que o *timestamp* do processo que enviou a mensagem

Sistemas Distribuídos

188

### Algoritmos Distribuídos (exclusão mútua)

The diagram shows four processes: P1 (yellow), P2 (green), P3 (blue), and P4 (red). Each process has a timestamp (TS) and a request for the critical section (SC). P1 has TS=6, P2 has TS=4, P3 has TS=4, and P4 has TS=4. P1 and P2 both want to enter the SC. P4 is already in the SC. P1 sends 'OK' to P2, and P2 sends 'OK' to P1. P1 enters the SC. P2 sends 'OK' to P3, and P3 sends 'OK' to P2. P2 enters the SC. P2 sends 'OK' to P4, and P4 sends 'OK' to P2. P2 exits the SC. P4 sends 'OK' to P1, and P1 sends 'OK' to P4. P4 exits the SC. P1 sends 'OK' to P3, and P3 sends 'OK' to P1. P1 exits the SC. P3 enters the SC.

Sistemas Distribuídos

189

## Enunciado do 1ro Trabalho Prático

- Implementar o algoritmo de exclusão mútua distribuída baseado na garantia de acesso a partir da autorização de todos
  - Supor relógios das máquinas sincronizados
  - Trabalho deve ser feito na linguagem C, utilizando RPC
  - O recurso compartilhado pelos diversos processos é um arquivo que está em uma área comum das diversas máquinas onde os processos vão ser executados. No início este arquivo tem um valor aleatório. Cada processo deve ler o último valor que se encontra neste arquivo, realizar uma operação sobre o valor, e armazenar o resultado na próxima posição (append). A seqüência de operações (o conteúdo do arquivo) deve constar no documento final a ser entregue. Cada vez que o processo acessa o recurso (arquivo) ele deve imprimir o valor que lá se encontrava, a operação que foi realizada, e o novo valor que lá foi armazenado. Cada processo deve realizar no mínimo 5 operações sobre o valor do arquivo. Cada processo deve ter uma operação diferente dos outros, de tal forma que o resultado de dois processos com base na mesma entrada é sempre diferente.
  - Devem existir no mínimo 5 processos executando

## Algoritmos Distribuídos (exclusão mútua)

### ➤ Algoritmo baseado na passagem de token

- Neste método, exclusão mútua é conseguida pelo uso de um token único que circula entre os processos do sistema
- Um token é uma mensagem especial que dá ao detentor da mensagem direito de acesso à SC
- Para que o algoritmo seja justo, os processos são organizados em um anel
- O token circula entre os processos no anel sempre na mesma direção

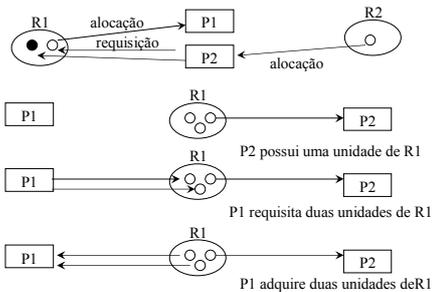
## ALGORITMOS DISTRIBUÍDOS *Deadlock*

## Algoritmos Distribuídos (*deadlock*)

- Um *deadlock* é causado pela situação onde um conjunto de processos está bloqueado permanentemente, i.e., não conseguem prosseguir a execução, esperando um evento que somente outro processo do conjunto pode causar.
- Situação de *deadlock*:  
alocação de recursos formando ciclo
- Outra Situação de *deadlock*:
  - vários processos tentam alocar recursos para realizar suas tarefas, alocando o total de recursos de uma máquina.
  - Não podem acabar as tarefas por não terem recursos suficientes, não liberam recursos por não terem acabado as tarefas.
  - sinalizar a execução dos processos, não permitindo concorrência de processos que utilizem na soma mais que a quantidade de recursos disponíveis

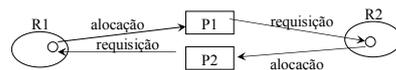
## Algoritmos Distribuídos (*deadlock*)

### ➤ Representação da relação entre processos e recursos



## Algoritmos Distribuídos (*deadlock*)

### ➤ Deadlock



## Algoritmos Distribuídos (*deadlock*)

- **Condições necessárias para *deadlock*** [Coffman, et.al., 1971]
  - *exclusão mútua*: se recurso bloqueado por processo P, então outros processos tem que esperar processo P para usar o recurso;
  - *segura e espera*: processos podem requerer uso de novos recursos sem liberar recursos em uso;
  - *não preempção*: recurso se torna disponível somente pela liberação do recurso pelo processo;
  - *espera circular*: 2 ou mais processos formam uma cadeia circular na qual cada processo está a espera de um recurso bloqueado pelo próximo membro da cadeia.
- As quatro condições devem ser válidas ao mesmo tempo
- Se uma delas for quebrada, então *deadlock* não acontece no sistema

## Algoritmos Distribuídos (*deadlock*)

### ➤ Estratégias de tratamento de *deadlock*

- evitar
  - evita *deadlock* alocando recursos cuidadosamente
- prevenir
  - estaticamente faz com que *deadlocks* não ocorram
- detectar
  - permite que o *deadlock* ocorra, detecta e tenta recuperar

## Algoritmos Distribuídos (*deadlock*)

### ➤ Métodos para evitar *deadlocks*

- usam conhecimento sobre quantidades de recursos que processos irão usar, prevendo estado futuro do sistema que possa ocasionar *deadlock*
- **noção de *safe-state***: sistema está em um *safe-state* se não está em *deadlock* e existe uma ordem de execução dos processos do sistema que garanta que eles possam alocar os recursos necessários para prosseguir e terminar
- **formação de uma *safe-state sequence***:
  - para qualquer processo  $P_i$  em de uma *safe sequence*, os recursos que  $P_i$  pode ainda requisitar podem ser satisfeitos pelos recursos disponíveis no momento mais os recursos bloqueados por todos processos antes de  $P_i$  na *safe sequence*
  - se  $P_i$  não pode executar imediatamente, isto significa que  $P_i$  pode esperar que os processos antes dele na *safe-sequence* acabem, para então prosseguir

## Algoritmos Distribuídos (*deadlock*)

### ➤ Métodos para evitar *deadlocks*

- usados raramente
- assume-se que existe conhecimento prévio sobre o comportamento dos processos - difícil na prática
- assume-se que número de unidades de um recurso é fixa e conhecida *a priori*
- restringe muitos pedidos de alocação que não necessariamente levariam a *deadlock*
- em sistemas distribuídos, dificuldade para:
  - coletar informação sobre necessidade/oferta de recursos é mais difícil

## Algoritmos Distribuídos (*deadlock*)

### ➤ Métodos para prevenir *deadlocks*

- projetar o sistema de tal maneira que *deadlocks* sejam impossíveis
- não necessita teste durante *run-time*
- condições necessárias e suficientes:
  - exclusão mútua
  - segura e espera
  - não preempção
  - espera circular
- técnicas: garantir que ao menos uma das condições não é nunca satisfeita
- métodos: *collective requests*, *ordered requests*, preempção

## Algoritmos Distribuídos (*deadlock*)

### ➤ Métodos para prevenir *deadlocks*

- *collective requests*
  - evita que condição segura e espera possa ser satisfeita
  - se um processo tem um recurso ele não pode ter outros
  - políticas
    - a) fazer *request* de todos recursos antes de sua execução: se todos recursos estão disponíveis, o processo pode executar senão, nenhum recurso é alocado e o processo espera
    - b) processo pode requerer recursos durante execução se ele liberar outros - tem que liberar todos e alocar recursos necessários até poder novamente liberar para depois alocar
  - b melhor que a pois processo pode não saber quantos recursos são necessários antes de começar a execução;

## Algoritmos Distribuídos (*deadlock*)

### ➤ Métodos para prevenir deadlocks

- *collective requests*
  - má utilização de recursos: processo pode ter vários recursos alocados e não usar alguns por longos períodos
  - *starvation*: se processo precisa de muitos recursos, cada vez que faz pedido pode encontrar um ou mais já alocados, tem que esperar e voltar a pedir.

## Algoritmos Distribuídos (*deadlock*)

### ➤ Métodos para prevenir deadlocks

- *ordered requests*
  - evita que espera circular aconteça
  - associa número a recurso
  - processo só pode alocar recursos em uma determinada ordem

## Algoritmos Distribuídos (*deadlock*)

### ➤ Métodos para prevenir deadlocks

- *ordered requests*

$r = \{ r_1, r_2, r_3, \dots, r_n \}$  // recursos

função  $f = r \rightarrow N$  onde  $N$  é o conjunto dos números naturais

$f(\text{disco}) = 1$

$f(\text{fita}) = 2$

$f(\text{impressora}) = 3$

Processo que requisita  $R_i$  somente pode requisitar  $R_j$  se e somente se  $f(R_j) > f(R_i)$ , caso contrário somente requisita  $R_j$  após liberar  $R_i$

## Algoritmos Distribuídos (*deadlock*)

### ➤ Métodos para prevenir deadlocks

- preempção
  - possibilita preempção de recurso
  - recurso preemptável: estado pode ser facilmente salvo para ser restaurado depois (ex.: CPU e memória)
  - se processo precisa de recurso, faz *request*, se outro processo que detém recurso está bloqueado a espera de outro recurso, então recurso é preemptado e passado ao primeiro processo, caso contrário primeiro processo espera
  - durante espera recursos podem ser preemptados deste processo para que outros progridam
  - processo é desbloqueado quando recurso requerido, e qualquer outro recurso preemptado dele, possam ser bloqueados para o processo

## Algoritmos Distribuídos (*deadlock*)

### ➤ Métodos para detecção de deadlocks

- não evita nem previne, deixa que aconteçam e depois detecta para corrigir
- algoritmo examina estado do sistema para determinar se existe *deadlock*
- se existe - toma ação corretiva
- técnica equivalente a sistema centralizado
- mantém informação sobre alocação de recursos, formando grafo de alocação de recursos, e procurando ciclos neste grafo (grafo WFG - wait for graph)

## Algoritmos Distribuídos (*deadlock*)

### ➤ Algoritmo centralizado de detecção

- Em cada máquina se mantém um grafo de alocação de recursos pelos processos
- Um coordenador centralizado mantém um grafo completo do sistema (a união dos grafos locais)
- Quando o coordenador detecta um ciclo, ele mata um dos processos e acaba com o *deadlock*

## Algoritmos Distribuídos (*deadlock*)

- Diferente de sistemas centralizados, onde as máquinas estão disponíveis automaticamente, em um sistema distribuído estas informações devem ser enviadas ao coordenador explicitamente
- Como cada máquina possui um grafo local, quando um arco é incluído ou excluído do grafo local, uma mensagem deve ser enviada para o coordenador
- Problema: quando uma mensagem demora para chegar, pode causar um **falso deadlock**

## Algoritmos Distribuídos (*deadlock*)

### ➤ Algoritmo distribuído de detecção

- Proposto por Chandy-Misra-Haas
- Funciona da seguinte forma:
  - iniciado quando um processo tem que esperar um recurso alocado por outro processo
  - processo envia msg (*probe message*) para processo (ou processos) que está(ão) utilizando recurso
  - msg contém 3 informações:
    - número do processo que está bloqueado
    - número do processo que enviou a msg
    - número do processo que está recebendo a msg
  - quando a msg chega a um processo, ele verifica se está esperando por recurso
    - se sim a msg é atualizada e enviada para o processo que está usando o recurso
  - se a msg dá toda a volta e chega ao processo que iniciou a msg, um ciclo existe e o sistema está em *deadlock*

## ALGORITMOS DISTRIBUÍDOS

### Algoritmos de eleição

## Algoritmos Distribuídos (eleição)

- Em sistemas distribuídos, diversos algoritmos necessitam que um processo funcione como coordenador, inicializador, sequenciador, enfim, ter um papel especial
- exemplos.:
  - coordenador de exclusão mútua com controle centralizado
  - coordenador para detecção de deadlock distribuído
  - sequenciador de eventos para ordenação consistente centralizada
  - etc.
- falha do coordenador compromete serviço para vários processos
- novo coordenador deve assumir - **eleição!**
  - **Objetivo: eleger um processo, entre os ativos, para desempenhar função especial**

## Algoritmos Distribuídos (eleição)

- Existem algoritmos especiais para escolha de um processo que assumirá o papel de coordenador
- Veremos dois *algoritmos de eleição*:
  - algoritmo do valentão (*bully algorithm*)
  - algoritmo em anel (*ring algorithm*)

## Algoritmos Distribuídos (eleição)

- Para ambos os algoritmos, assume-se que: (*general assumptions*)
  - 1) Todo processo no sistema tem uma *prioridade única*
  - 2) Quando eleição acontece, o processo com maior prioridade *entre os processos ativos* é eleito como coordenador
  - 3) Na recuperação (volta à atividade), um processo falho pode tomar ações para juntar-se ao grupo de processos ativos

## Algoritmos Distribuídos (eleição)

### Bully algorithm [García-Molina em 1982]

- Assume que um processo sabe a prioridade de todos outros processos no sistema

### Algoritmo

- Quando um processo  $P_i$  detecta que o coordenador não está respondendo a um pedido de serviço, ele inicia uma eleição da seguinte forma:
  - $P_i$  envia uma msg ELEIÇÃO para todos processos com prioridade maior que a sua
  - se nenhum processo responde
    - $P_i$  vence a eleição e torna-se coordenador
    - // significa que não há processos ativos com maior prioridade que a sua
    - $P_i$  manda uma msg COORDENADOR para os processos de menor prioridade informando que é o coordenador deste momento em diante.
- se processo  $P_j$  com prioridade maior que  $P_i$  responde (msg ALIVE)
  - $P_i$  não faz mais nada,  $P_j$  assume o controle
  - $P_j$  age como  $P_i$  nos passos a) e b)

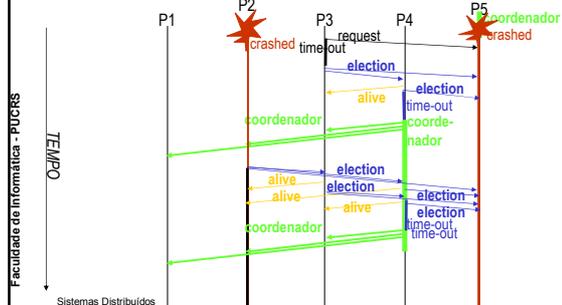
Sistemas Distribuídos

214

## Algoritmos Distribuídos (eleição)

### Bully algorithm - exemplo [Pradeep, pg 334]

- suponha processos P1 a P5 com prioridade conforme seu identificador



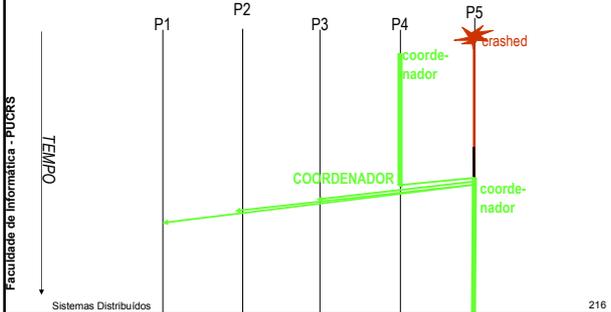
Sistemas Distribuídos

215

## Algoritmos Distribuídos (eleição)

### Bully algorithm - exemplo [Pradeep, pg 334]

- suponha processos P1 a P5 com prioridade conforme seu identificador



Sistemas Distribuídos

216

## Algoritmos Distribuídos (eleição)

### Ring algorithm

- Baseado no uso de um anel lógico, sem uso de token
- Cada processo conhece o anel inteiro, mas manda mensagens somente para o próximo processo ativo na direção do anel
- Quando processo detecta que o coord. não está ativo, ele constrói uma msg ELEIÇÃO contendo seu id., e manda para o próximo do anel
- A cada passo, o processo que recebe a msg inclui seu id na msg e envia para o próximo do anel
- No final o processo que iniciou a eleição recebe a msg e escolhe aquele que tem maior id.
- Nova msg é enviada novamente através do anel para todos contendo o novo coordenador
- Uma vez que a msg passou por todos processos e chegou no originador, ela é retirada do sistema

Sistemas Distribuídos

217

## Algoritmos Distribuídos (eleição)

### Análise

- algoritmo do valentão (bully)
  - se processo com prioridade mais baixa detecta a falha do coordenador, em um sistema com  $n$  processos, então  $n-1$  eleições acontecem
  - cada eleição tem mensagens conforme número de processos -  $O(n^2)$  mensagens - *no pior caso*
  - se processo que detecta falha é o ativo de maior prioridade precisa só de  $n-2$  mensagens - *melhor caso*
- algoritmo do anel
  - eleição sempre precisa de  $2n-1$  mensagens
  - $n-1$  mensagens para rotação da mensagem de eleição
  - $n-1$  mensagens para rotação da mensagem de coordenador

Sistemas Distribuídos

218