Deadlock

Sistemas Operacionais

Prof. Pedro Ramos
pramos.costar@gmail.com

Pontifícia Universidade Católica de Minas Gerais
ICEI - Departamento de Ciência da Computação

ANTERIORMENTE - PROBLEMAS DE SINCRONIZAÇÃO

- Leitores e Escritores

- Vários leitores, um único escritor
- Na prática, utiliza-se locks de leitura e escrita

- Jantar com filósofos

- Precisam segurar múltiplos recursos para realizar a tarefa

SINCRONIZAÇÃO DE THREADS NO MUNDO REAL

- Produtor-consumidor
  - Reprodutor de Áudio-Vídeo: threads de rede e exibição; buffer compartilhado
  - Servidores web: thread principal e threads secundarias
- Leitor-escritor
  - Sistema bancário: leitura de saldos de contas vs. atualizações
- Jantar com filósofos
  - Processos cooperativos que precisam compartilhar recursos limitados
    - Conjunto de processos que precisam bloquear múltiplos recursos
      - *Disco e fita (backup)*
    - Reserva de viagem: bancos de dados de hotel, companhia aérea, aluguel de carro

HOJE - DEADLOCKS

O que são deadlocks?

Condições para deadlocks

Prevenção de deadlock

Detecção de deadlock

O QUE É DEADLOCK

 # Deadlock:

Uma condição em que **duas ou mais threads estão esperando por um evento que só pode ser gerado por essas mesmas threads.**

- Exemplo:

Processo A:

```
printer.Wait();
disk.Wait();

// copiar do disco
// para a impressora

printer.Signal();
disk.Signal();
```

Processo B:

```
disk.Wait();
printer.Wait();

// copiar do disco
// para a impressora

printer.Signal();
disk.Signal();
```

DEADLOCKS - TERMINOLOGIA

- **Deadlock** pode ocorrer quando várias threads competem simultaneamente por um número finito de recursos

- Algoritmos de **prevenção de deadlock** verificam solicitações de recursos e, possivelmente, a disponibilidade para evitar deadlock

- **Detecção de deadlock** encontra instâncias de deadlock quando as threads param de progredir e tenta recuperar

- **Inanição** (Starvation) ocorre quando uma thread espera indefinidamente por algum recurso, mas outras threads estão de fato utilizando-o (fazendo trabalho útil).

=> Inanição é uma condição diferente de deadlock

CONDIÇÕES NECESSÁRIAS PARA DEADLOCK

O deadlock pode ocorrer se **todas as seguintes condições forem atendidas**:

- **Exclusão Mútua:** pelo menos uma thread deve segurar um recurso em modo não-compartilhável, ou seja, o recurso só pode ser utilizado por uma thread de cada vez.
- **Espera por Recurso:** pelo menos uma thread segura um recurso e está esperando por outros recursos ficarem disponíveis. Uma thread diferente segura esses recursos.
- **Sem Preempção:** uma thread só pode liberar um recurso voluntariamente; outra thread ou o SO não pode forçar a thread a liberar o recurso.
- **Espera Circular:** um conjunto de threads em espera $\{t_1, \ldots, t_n\}$ onde t_i está esperando por t_{i+1} $(i = 1 \text{ a } n)$ e t_n está esperando por t_1.

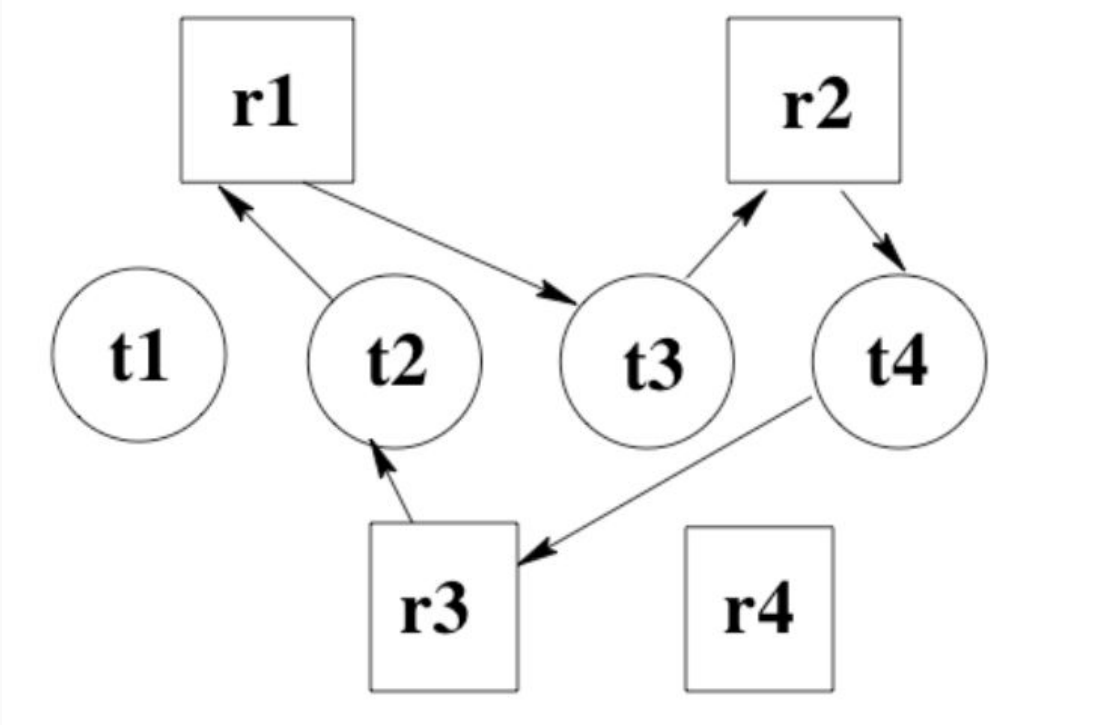
DETECÇÃO DE DEADLOCKS: GRAFO DE ALOCAÇÃO DE RECURSOS

- Definimos um grafo com vértices que representam tanto os recursos $\{r_1, \ldots, r_m\}$ quanto as threads $\{t_1, \ldots, t_n\}$.
  - Uma aresta direcionada de uma thread para um recurso $t_i \rightarrow r_j$ indica que t_i solicitou recurso r_j, mas ainda não o adquiriu *(Aresta de Solicitação)*
  - Uma aresta direcionada de um recurso para uma thread $r_j \rightarrow t_i$ indica que o SO alocou r_j para t_i *(Aresta de Atribuição)*

- Se o grafo não tiver ciclos, não existe deadlock.

- Se o grafo tiver um ciclo, ***pode existir deadlock.***

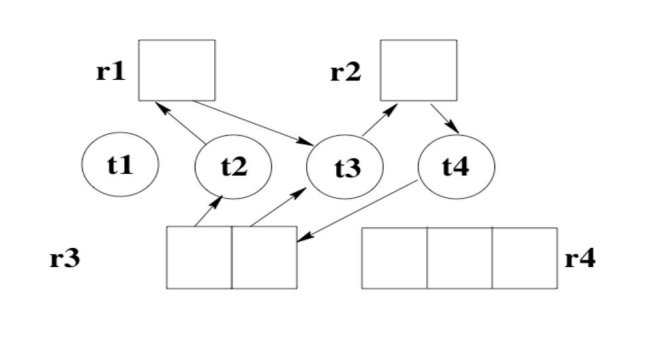
GRAFO DE ALOCAÇÃO DE RECURSOS



GRAFO DE ALOCAÇÃO DE RECURSOS: DETECTANDO DEADLOCKS

E se houver múltiplas CÓPIAS intercambiáveis de um mesmo recurso?

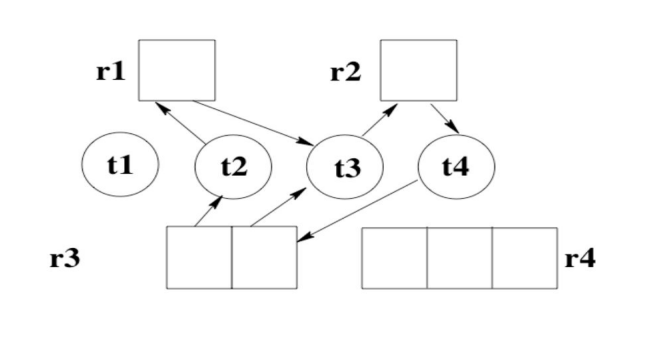
- Nesse caso, um ciclo indica apenas que **pode existir deadlock**.
- Se qualquer instância de um recurso envolvido no ciclo for **segurada por uma thread que não está no ciclo**, então podemos avançar quando esse recurso for liberado.

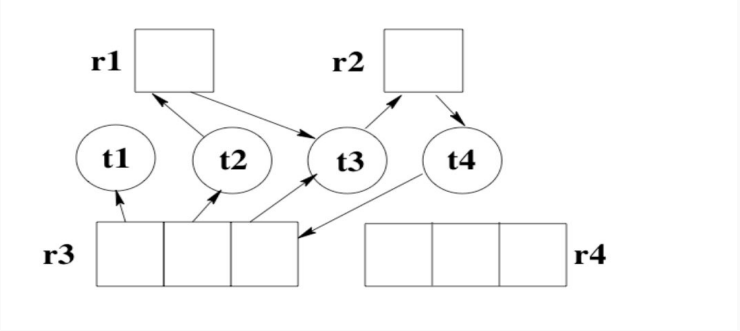


GRAFO DE ALOCAÇÃO DE RECURSOS: DETECTANDO DEADLOCKS

E se houver múltiplas instâncias intercambiáveis de um mesmo recurso?

- Nesse caso, um ciclo indica apenas que **pode existir deadlock**.
- Se qualquer instância de um recurso envolvido no ciclo for **segurada por uma thread que não está no ciclo**, então podemos avançar quando esse recurso for liberado.

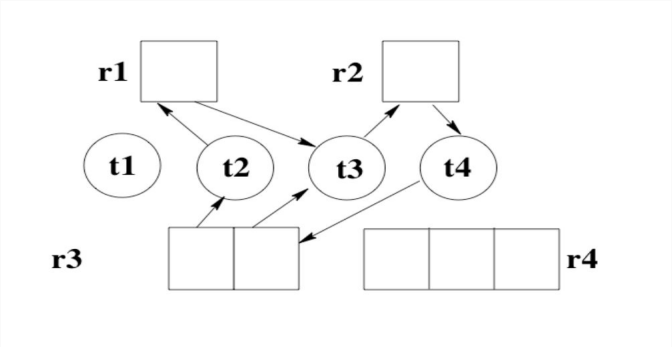




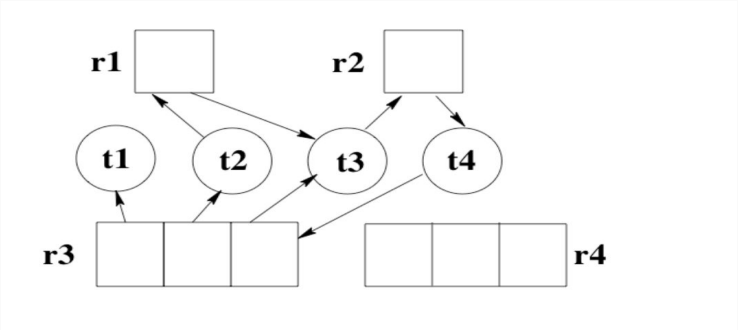
GRAFO DE ALOCAÇÃO DE RECURSOS: DETECTANDO DEADLOCKS

E se houver múltiplas instâncias intercambiáveis de um mesmo recurso?

- Nesse caso, um ciclo indica apenas que **pode existir deadlock**.
- Se qualquer instância de um recurso envolvido no ciclo for **segurada por uma thread que não está no ciclo**, então podemos avançar quando esse recurso for liberado.



DEADLOCK!



Pode haver deadlock.
(t_1 vai liberar r_3)

DEADLOCK: DETECTAR E CORRIGIR USANDO O GRAFO DE ALOCAÇÃO

- Examine o grafo de alocação de recursos em busca de ciclos e, em seguida, **quebre os ciclos**.

- Diferentes maneiras de quebrar um ciclo:
  - Matar todas as threads no ciclo.
  - Matar as threads uma a uma, forçando-as a abrir mão dos recursos.
  - Preemptar recursos um por um, revertendo o estado da thread que **segura o recurso para o estado em que estava antes de obter o recurso**. Essa técnica é comum em **transações de banco de dados. (Rollback)**

DEADLOCK: DETECTAR E CORRIGIR USANDO O GRAFO DE ALOCAÇÃO

Detectar ciclos leva O(n²) de tempo, onde n é **|T| + |R|.**

Quando devemos executar este algoritmo?

1. Apenas antes de conceder um recurso, verificar se concedê-lo levaria a um ciclo?

2. Sempre que uma solicitação de recurso não puder ser atendida?

3. Em um cronograma regular *(horário ou ...)*?

4. Quando a utilização da CPU cai abaixo de um determinado limite?

DEADLOCK: DETECTAR E CORRIGIR USANDO O GRAFO DE ALOCAÇÃO

Detectar ciclos leva $O(n^2)$ de tempo, onde n é **|T| + |R|.**

Quando devemos executar este algoritmo?

1. Apenas antes de conceder um recurso, verificar se concedê-lo levaria a um ciclo?
   (Cada solicitação é então $O(n^2)$.)
2. Sempre que uma solicitação de recurso não puder ser atendida?
   (Cada solicitação falhada é $O(n^2)$.)
3. Em um cronograma regular *(horário ou ...)*?

4. Quando a utilização da CPU cai abaixo de um determinado limite?

DEADLOCK: DETECTAR E CORRIGIR USANDO O GRAFO DE ALOCAÇÃO

Detectar ciclos leva $O(n^2)$ de tempo, onde n é **|T| + |R|.**

Quando devemos executar este algoritmo?

1. Apenas antes de conceder um recurso, verificar se concedê-lo levaria a um ciclo?
   (Cada solicitação é então $O(n^2)$.)
2. Sempre que uma solicitação de recurso não puder ser atendida?
   (Cada solicitação falhada é $O(n^2)$.)
3. Em um cronograma regular *(horário ou ...)*?
   (Pode demorar muito tempo para detectar deadlock!)
4. Quando a utilização da CPU cai abaixo de um determinado limite?
   (Pode demorar muito tempo para detectar deadlock!)

DEADLOCK: DETECTAR E CORRIGIR USANDO O GRAFO DE ALOCAÇÃO

Detectar ciclos leva $O(n^2)$ de tempo, onde n é **|T| + |R|.**

Quando devemos executar este algoritmo?

1. Apenas antes de conceder um recurso, verificar se concedê-lo levaria a um ciclo?
   (Cada solicitação é então $O(n^2)$.)
2. Sempre que uma solicitação de recurso não puder ser atendida?
   (Cada solicitação falhada é $O(n^2)$.)
3. Em um cronograma regular *(horário ou ...)*?
   (Pode demorar muito tempo para detectar deadlock!)
4. Quando a utilização da CPU cai abaixo de um determinado limite?
   (Pode demorar muito tempo para detectar deadlock!)

O que os sistemas operacionais atuais fazem?

- Deixam para o programador/aplicação.

PREVENÇÃO DE DEADLOCKS

Garantir que pelo menos uma das condições necessárias **não se mantenha**.

1. Exclusão Mútua

2. Espera por Recurso

3. Sem Preempção

4. Espera Circular

PREVENÇÃO DE DEADLOCKS

Garantir que pelo menos uma das condições necessárias **não se mantenha**.

1. Exclusão Mútua: tornar os recursos compartilháveis (mas nem todos os recursos podem ser compartilhados).
2. Espera por Recurso:
   - Garantir que uma thread não possa segurar um recurso ao solicitar outro.
   - Fazer com que as threads solicitem todos os recursos de que precisam de uma só vez e liberar todos os recursos antes de solicitar um novo conjunto.
3. Sem Preempção:
   - Se uma thread solicitar um recurso que não pode ser alocado imediatamente, o SO faz preempção (libera) todos os recursos que a thread está segurando atualmente.
   - Somente quando todos os recursos estiverem disponíveis, o SO reiniciará a thread.
   - Problema: nem todos os recursos podem ser facilmente preemptados, como impressoras.
4. Espera Circular: impor uma ordem (numeração) nos recursos e solicitá-los na ordem.

EVITAR DEADLOCK COM RESERVA DE RECURSOS

- As threads fornecem informações antecipadas sobre o número **MÁXIMO** de recursos que podem precisar durante a execução.

- Uma **sequência de threads {t_1, ..., t_n} É SEGURA** se, para cada t_i, os recursos que t_i ainda pode solicitar podem ser atendidos pelos recursos atualmente disponíveis + os recursos segurados por todas as t_j, $j < i$.

EVITAR DEADLOCK COM RESERVA DE RECURSOS

sequência de threads {t_1, ..., t_n}

- As threads estão em um ESTADO SEGURO quando existe uma sequência segura para as threads.

- Um ESTADO INSEGURO não é equivalente a deadlock; ele pode apenas levar ao deadlock, *uma vez que algumas threads podem não utilizar realmente o máximo de recursos que declararam.*

EVITAR DEADLOCK COM RESERVA DE RECURSOS

- Conceder um recurso a uma thread é **SEGURO se o novo estado for seguro.**

- **Se o novo estado for inseguro**, a thread deve **esperar**, ***mesmo que o recurso esteja atualmente disponível***.

- Este algoritmo **garante, de forma conservadora, que nunca ocorra a condição de espera circular.**

EXEMPLO

t1, t2 e t3 competem por 12 unidades de barramento.
Atualmente:

- 11 unidades estão alocadas para as threads
- Há 1 disponível.

O estado atual é seguro (existe uma sequência segura, {t1, t2, t3}, onde todas as threads podem obter seu número máximo de recursos sem esperar):
- t1 pode ser concluída com a alocação atual de recursos.
- t2 pode ser concluída com seus recursos atuais, + todos os recursos de t1 e a unidade de barramento não alocada.
- t3 pode ser concluída com todos os seus recursos atuais, todos os recursos de t1 e t2, e a unidade de barramento não alocada.

	MAX	EM USO	PODE QUERER
t1	4	3	1
t2	8	4	4
t3	12	4	8

EXEMPLO

Se t3 pede uma unidade de barramento a mais, então **ela deve esperar pois isso levaria a um estado inseguro.**

Agora existem **0 unidades de barramento disponíveis**,

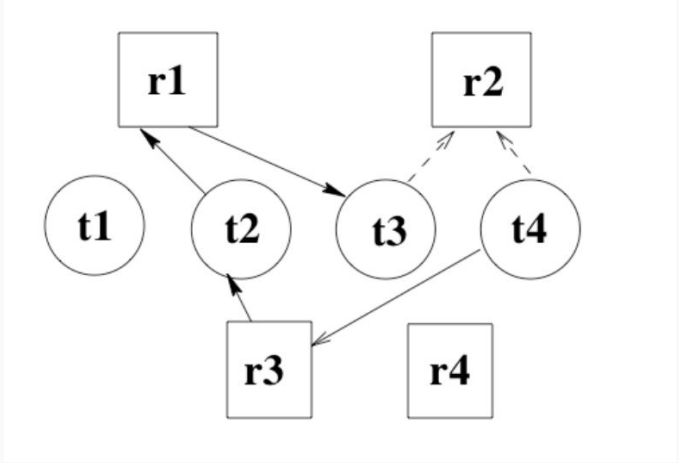
mas cada thread pode precisar de pelo menos mais uma. => ***Estado inseguro.***

	MAX	EM USO	PODE QUERER
t1	4	3	1
t2	8	4	4
t3	12	5	7

DEADLOCKS: GRAFO DE ALOCAÇÃO DE RECURSOS

Arestas de Reivindicação

- Uma aresta de reivindicação é uma **aresta de uma thread para um recurso que pode ser solicitado no futuro.**
- Satisfazer um pedido resulta na conversão de uma aresta de reivindicação em uma aresta de alocação e *muda sua direção.*
- Um **ciclo** neste gráfico estendido de alocação de recursos indica um **estado inseguro.**
- Se a alocação **resultar em um estado inseguro**, a alocação é **negada**, mesmo que o recurso esteja disponível.
  → A aresta de reivindicação **é convertida em uma aresta de solicitação** e a thread **espera.**



- Essa solução **não funciona para múltiplas instâncias do mesmo recurso.**

ALGORITMO DE BANKER

- Algoritmo para Múltiplas Instâncias do Mesmo Recurso
- Este algoritmo lida com múltiplas instâncias do mesmo recurso.
- Obriga as threads a fornecer informações antecipadas sobre quais recursos podem precisar durante a execução.
- **Os recursos solicitados não podem exceder o total disponível no sistema.**
- O algoritmo aloca recursos para uma thread que faz a solicitação se a alocação deixar o sistema em um estado seguro.
- Caso contrário, a thread deve esperar.

RESUMO

- Deadlock: situação em que um **conjunto de threads/processos não pode prosseguir porque cada um requer recursos mantidos por outro membro do conjunto.**
- Detecção e recuperação: reconhecer o deadlock após sua ocorrência e rompê-lo.
- Evitar: não alocar um recurso se isso introduzir um ciclo.
- Prevenção: projetar estratégias de alocação de recursos que **garantam que uma das condições necessárias nunca se mantenha**.
- Codificar programas concorrentes com muito cuidado. Isso ajuda apenas a prevenir deadlock sobre os recursos gerenciados pelo programa, não só os recursos do sistema operacional.
- Ignorar a possibilidade de deadlock - A maioria dos sistemas operacionais usa esta opção.

PERGUNTAS?

REFERÊNCIAS

- **TANENBAUM, Andrew.** Sistemas operacionais modernos.
- **SILBERSCHATZ, Abraham et al.** Fundamentos de sistemas operacionais: princípios básicos.
- **MACHADO, Francis; MAIA, Luiz Paulo.** Arquitetura de Sistemas Operacionais.
- **CARISSIMI, Alexandre et al.** Sistemas operacionais.