

Sincronização de Processos e Threads

Engenharia de Computação

Pontifícia Universidade Católica de Campinas

Prof. Dr. Denis M. L. Martins

Objetivos de Aprendizagem

Ao final desta aula, você será capaz de:

- Reconhecer e explicar **condições de corrida**.
- Explicar o conceito de **seção crítica** e **exclusão mútua** (MX).
- Explicar o fenômeno **deadlock** e estratégias de prevenção e de detecção.
- Explicar o fenômeno **starvation** como um desafio de MX.

Conceitos Fundamentais

- Acesso **concorrente** a dados compartilhados pode resultar em **inconsistências**
- Exemplo: problemas de **transações concorrentes** em Bancos de Dados. Lembre das propriedades **ACID**.
 - **Dirty read**: surge quando transação lê dados que ainda não foram confirmados (committed) pelo banco de dados.
 - **Lost update**: surge quando duas ou mais transações tentam modificar a mesma informação simultaneamente e uma das modificações é perdida.
- **Sincronização**: Manter consistência de dados utilizando **mecanismos** para garantir a execução ordenada de tarefas.

Problemas de Concorrência: Lost Update

Atualização perdida por modificação concorrente em recurso compartilhado.

<i>time</i>	<i>T₁</i>	<i>T₂</i>	<i>amount_x</i>
t ₁		begin transaction	100
t ₂	begin transaction	read(amount _x)	100
t ₃	read(amount _x)	amount _x = amount _x + 120	100
t ₄	amount _x = amount _x - 50	write(amount _x)	220
t ₅	write(amount _x)	commit	50
t ₆	commit		50

Condição de Corrida - *Race Conditions*

Condição de corrida acontece quando atividades concorrentes acessam recursos compartilhados

Condição de corrida: situação em que dois ou mais threads/processos estão lendo, ou escrevendo dados compartilhados e o resultado depende de quem executa e quando.

- Variáveis, memória e arquivos são recursos compartilhados entre threads/processos.
- Exemplo: Dois processos `P0` e `P1` criam processos usando `fork()`.
 - Ambos precisam acessar a variável de kernel `next_available_pid` para consultar o próximo `pid` disponível para seus processos-filhos.

Condição de Corrida - *Race Conditions* (cont.)

Se não houver um mecanismo de controle ao acesso da variável `next_available_pid`, `P0` e `P1` os processos-filhos podem ter com o **mesmo** `pid`.

Tempo	P0		P1
T_1	<code>pid_t child = fork();</code>		<code>pid_t child = fork();</code>
T_2	<code>request pid</code>		<code>request pid</code>
T_3	\rightarrow	<code>next_available_pid = 2615</code>	\leftarrow
T_4	<code>return 2615</code>		<code>return 2615</code>
T_5	<code>child = 2615</code>		<code>child = 2615</code>

Operações não-atômicas

Condições de corrida acontecem geralmente em **operações não atômicas**:

- `contador++` → sequência de instruções de máquina
 - Carrega valor da variável da RAM
 - Executa `ADD` na ULA
 - Escreve o resultado na RAM
- Mudança de contexto pode acontecer após qualquer uma dessas instruções de máquina, "no meio" de uma instrução de alto nível.

Exemplo

```
#include <stdio.h>
#include <pthread.h>

int contador = 0;

void* incrementa_contador(void* thread_id) {
    int tid = (int)(long)thread_id; // Cast para obter o ID da thread
    for (int i = 0; i < 10000; i++) {
        contador++; // Acesso direto à variável global sem proteção
    }
    printf("Thread %d: Contador final = %d\n", tid, contador);
    pthread_exit(0);
}

int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, incrementa_contador, (void*)1); // Cria thread1
    pthread_create(&thread2, NULL, incrementa_contador, (void*)2); // Cria thread2
    pthread_join(thread1, NULL); // Espera a thread1 terminar
    pthread_join(thread2, NULL); // Espera a thread2 terminar

    printf("Valor final do contador: %d\n", contador); // O valor final será imprevisível
    return 0;
}
```


Exemplo: Discussão

- **Resultado:** O valor final do `contador` *não* será sempre 20000.
- **Variável Global:** A variável `contador` é uma variável global, o que significa que ela está armazenada na memória e pode ser acessada por múltiplas threads, simultaneamente, sem nenhuma proteção.
- **Incremento:** `contador++` consiste em três etapas:
 - i. Ler o valor atual de `contador` .
 - ii. Incrementar o valor lido.
 - iii. Escrever o novo valor de volta em `contador` .
- **Condição de Corrida:** Se as duas threads executarem essas etapas simultaneamente, a ordem em que elas acessam e modificam `contador` é imprevisível.

Seção Crítica

Uma seção crítica é um trecho de código onde múltiplas threads podem acessar um recurso compartilhado (como uma variável global ou um arquivo) simultaneamente.

- É a parte do código que precisa ser protegida.
- A **condição de corrida** ocorre na seção crítica.

Precisamos nos certificar que execuções concorrentes que acessam recursos compartilhados sejam **isoladas** umas das outras.

- Utilizar **exclusão mútua** para certificar que enquanto uma thread acessa um recurso compartilhado, outros(as) não poderão impedidos de modificar esse recurso.
- Cada thread deve pedir permissão para entrar na seção crítica, executar a operação, e então sair da seção crítica

Seção Crítica: Requisitos para uma Solução

1. **Exclusão Mútua:** Se um processo P_i está executando na seção crítica, então nenhum outro processo pode estar na seção crítica.
2. **Progresso:** Se um processo estiver na seção crítica e houver processos com intenção de entrar na seção crítica, então a seleção de qual processo pode entrar não deve ser adiada indefinidamente
3. **Espera Limitada:** Deve existir um limite no número de vezes que outros processos podem entrar nas suas seções críticas depois que um processo tenha pedido para entrar na sua.

Solução baseada em Interrupções

- Em um sistema de processador único, a solução mais simples é desabilitar interrupções logo após um processo entrar em sua sessão crítica e as reabilitar um momento antes de partir.
 - Problema 1: Permissão para desabilitar interrupções
 - Problema 2: Ineficiência
- Em um sistema de múltiplos processadores, a desabilitação funcionaria apenas para a CPU que executou a instrução de `disable`.

Solução de Peterson

- Proposta por [G. L. Peterson](#) em 1981. Solução para **dois** processos.

`bool flag[2] = {false, false};` → indica se um processo está pronto para entrar na seção crítica.

`int turn;` → indica de quem é a vez de entrar na seção crítica

Processo 0	Processo 1
<pre>flag[0] = true; turn = 1; while (flag[1] && turn == 1) { // busy wait } // critical section ... turn = 0; // end of critical section flag[0] = false;</pre>	<pre>flag[1] = true; turn = 0; while (flag[0] && turn == 0) { // busy wait } turn = 1; // critical section ... // end of critical section flag[1] = false;</pre>

Exercício

Mostre que a solução de Peterson atende aos requisitos de uma solução para o problema da seção crítica.

1. A exclusão mútua é preservada.
2. O requisito de progresso é atendido.
3. O requisito de espera limitada é atendido.

Lock e Mutexes

- Solução baseada em variável de trava (lock).
 - Variável booleana indicando se um lock está disponível.
 - Adquire o lock/mutex no início da seção crítica usando `acquire()`
 - Libera o no fim usando `release()`.
- As chamadas `acquire()` e `release()` precisam ser **atômicas** (via instruções atômicas de hardware)
- Espera ocupada (*busy waiting*)
- Esse tipo de lock é denominado *spinlock* (trava giratória).

```
while (true) {  
    acquire lock  
    seção crítica  
    libera lock  
    restante do código  
}
```

Semáforos

- Forma mais sofisticada que o mutex para oferecer sincronização.
- Semáforo `S` é uma variável inteira
- Acessado por duas operações atômicas: `wait()` e `signal()`.

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

```
signal(S) {  
    S++;  
}
```

- Semáforo de contagem: Valor inteiro não é restrito
- Semáforo binário: Mesmo que um mutex

Liveness

- Liveness é o conjunto de propriedades que um sistema deve ter para garantir progresso.
 - Processos podem ter que esperar **indefinidamente** por um mutex ou semáforo estar disponível
 - Esperar indefinidamente viola os requisitos de progresso e espera limitada
- Espera indefinida é uma falha de liveness.
- Desafio: **Deadlocks**
 - Dois ou mais processos estão esperando por um evento que só pode ser causado por um processo em espera.
 - Espera cíclica de recursos (locks, espaço de memória, etc.)

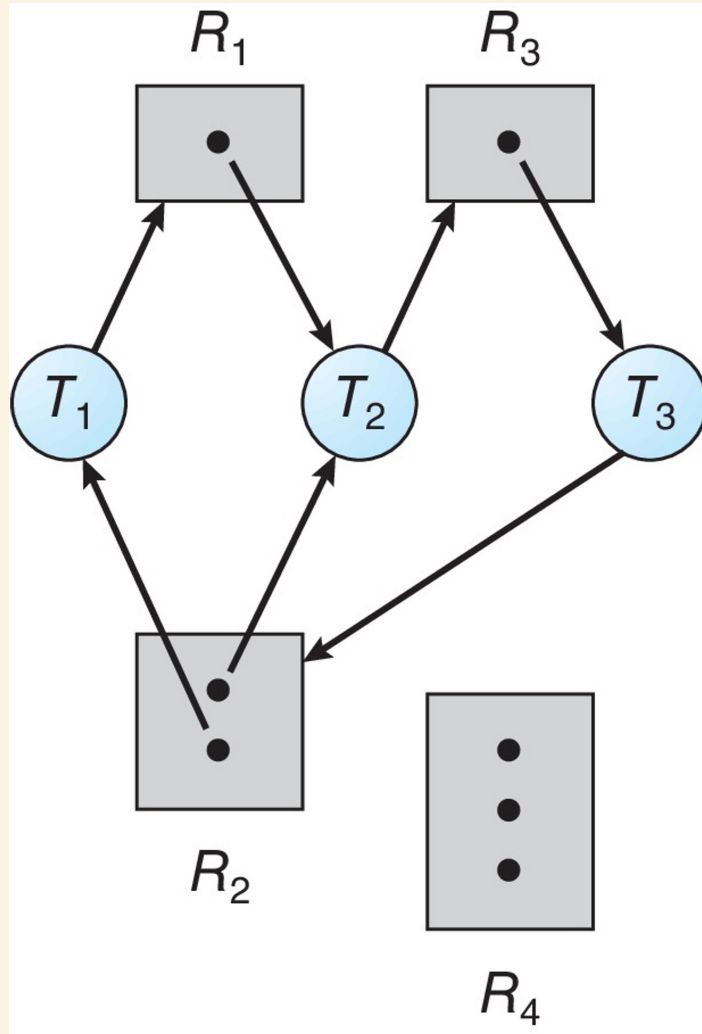
Deadlocks

- Modelo de Sistema
 - Recursos de tipos R_1, R_2, \dots, R_m
 - Cada tipo de recurso pode ter W_i instâncias
 - Cada processo usa um recurso da seguinte forma:
 - *request*: "trava" o recurso, ou espera que possa ser travado
 - *use*: operar sobre o recurso
 - *release*: destrava o recurso para outros

Condições para um Deadlock

- **Exclusão mútua:** só um processo por vez pode usar um recurso
- **Retenção e espera:** um processo que está travando um recurso fica na espera para travar outro
- **Sem preempção:** um recurso só pode ser liberado voluntariamente pelo processo que está usando
- **Espera circular:** existe um conjunto $\{P_0, P_1, \dots, P_n\}$ de processos esperando, tal que P_0 espera por um recurso travado por P_1 , P_1 espera por um recurso travado por P_2 , ..., P_{n-1} espera por um recurso travado por P_n , e P_n espera por um recurso travado por P_0 .

Grafo de Alocação de Recursos



Grafo: Conjuntos de vértices V e arestas E .

- V é particionado em dois tipos:
 - $P = \{P_1, P_2, \dots, P_n\}$, o conjunto de processos no sistema.
 - $R = \{R_1, R_2, \dots, R_m\}$, o conjunto de tipos de recursos no sistema
- Aresta de solicitação (request) – $P_i \rightarrow R_j$
- Aresta de atribuição (assignment) – $R_j \rightarrow P_i$

Fato: Sistema em deadlock se e apenas se o grafo contém um ciclo. (Se houver apenas uma instância por tipo, senão há possibilidade de deadlock).

Dúvidas e Discussão
