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 fenônemo **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.

time	T_1	T_2	$amount_x$
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	pid_t child = fork();		pid_t child = fork();
T_2	request pid		request pid
T_3	$\rightarrow$	next_available_pid = 2615	$\leftarrow$
T_4	return 2615		return 2615
T_5	child = 2615		child = 2615

Exemplo

```c
#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.
- **Operção Não-Atômica:** `contador++` involve:
    i. Ler o valor atual de `contador`.
    ii. Incrementar o valor lido.
    iii. Escrever o novo valor de volta em `contador`.
- Mudança de contexto pode acontecer após qualquer uma dessas instruções de máquina, "no meio" de uma instrução de alto nível.

Exemplo: Discussão (cont.)

- **Condição de Corrida:** Se as duas threads executarem essas etapas simultaneamente, a ordem em que elas acessam e modificam `contador` é imprevisível.
- O segmento de código onde ocorre mudança de variáveis compartilhadas é chamado **seção crítica**.
- O **Problema da Seção Crítica** é projetar um protocolo para resolver isso.

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 nenhum processo estiver na seção critica 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

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

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.
  - Adiquire 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) {
  adquire 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.

Semáforos (cont.)

- Controle de seção crítica:

```
wait(mutex);
CS
signal(mutex);
```

- Sincronização de execução: Os processos `P1` e `P2` executam as tarefas `S1` e `S2` e existe um requisito que `S1` aconteça antes de `S2`.

```
// P1
S1;
signal(synch);

// P2
wait(synch);
S2;
```

Implementação de Semáforos

- Apenas um processo deve executar `wait()` ou `signal()` por vez.
- A implementação do semáforo se torna a seção crítica: `wait` e `signal` precisam ser colocados nela.
- Pode causar *busy waiting*.
- Problemas:
  - Uso incorreto: `signal(mutex); ... wait(mutex);`
  - `wait(mutex); ... wait(mutex);`
  - Omissão de `wait(mutex)` e/ou `signal(mutex)`.
- Alternativa: bloquear processo na fila de espera por um semáforo.

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.)

Deadlock

- Dois ou mais processos aguardam eventos dependentes entre si.

Exemplo: Considere `S` e `Q` dois semáforos inicializados em 1.

```
P0          P1
wait(S);    wait(Q);
wait(Q);    wait(S);
...         ...
signal(S); signal(Q);
signal(Q); signal(S);
```

- `P0` executa `wait(S)` e `P1` `wait(Q)`.
  - Quando `P0` executa `wait(Q)`, ele deve esperar até que `P1` execute `signal(Q)`.
  - Mas `P1` está esperando por `P0` executar `signal(S)`.
- Já que esses `signal()` nunca vão ser executados, `P0` e `P1` estão em deadlock.

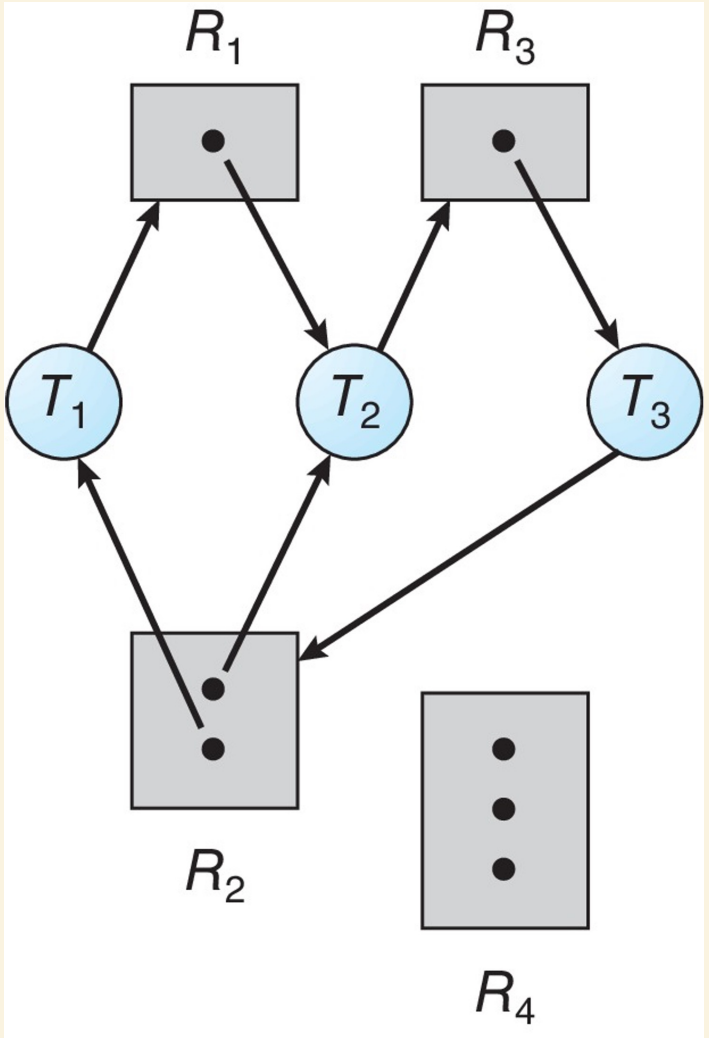
Deadlocks: Modelo de Sistema

- Recursos de tipos $R_1, R_2, \ldots, R_m$
- 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 $\{P_0, P_1, \ldots, P_n\}$ 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:
  - $P = \{P_1, P_2, \ldots, P_n\}$, o conjunto de processos no sistema.
  - $R = \{R_1, R_2, \ldots, R_m\}$, 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).

Deadlock: Prevenção

- Garantir que o sistema nunca entre em estado de deadlock, invalidando uma das quatro condições necessárias.
- Opções práticas:
  - Prevenir condição (2), "retenção e espera": Realizando o request de todos os recursos necessários de uma só vez
    - Conservative/static 2PL
    - Possível apenas em casos especiais
  - Prevenir condição (4), "espera circular": Recursos e requisições em ordenação linear.
- Problemas: baixo uso de recursos, possibilidade de inanição (**starvation**).
- **Exercício**: Explique como prevenir as condições **Exclusão Mútua e Sem Preempção**.

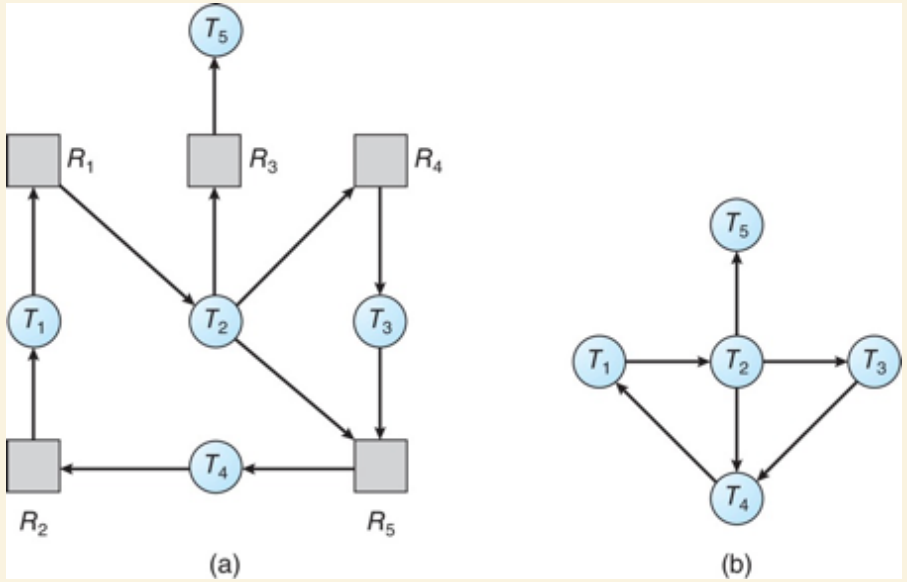
Deadlock: Impedir

- Analisar dinamicamente o estado de alocação para evitar ciclos de espera, muitas vezes requerendo que os processos declarem suas necessidades máximas de recursos a priori.
    - Se um deadlock não pode ser impedido facilmente, então o sistema pode decidir não alocar o recurso, mas bloquear o processo/thread que fez a requisição.
- Técnica clássica: Algoritmo do Banqueiro, proposto por Dijkstra.

Deadlock: Detecção e Recuperação

- Permitir que o deadlock ocorra, detectá-lo e então tomar medidas para se recuperar.
  - Manter grafo de espera `Pi → Pj` (próximo slide).
  - Detectar ciclos periodicamente.
- Abortar processos em deadlock.
- Critérios para escolher quais processos abortar: prioridade, tempo de execução, recursos usados, etc.

Exemplo de Detecção



Se todos os recursos têm apenas uma única instância, então definimos um algoritmo de detecção de deadlocks baseado em grafo de espera (*wait-for*).

(a) Grafo de Alocação de Recursos.

(b) Grafo de Espera correspondente.

Starvation

- Ocorre quando um processo nunca recebe os recursos necessários para continuar sua execução, mesmo estando pronto.
  - Outros processos continuam sendo favorecidos pelo escalonador ou pelo sistema de alocação de recursos.
  - Sistemas com políticas mal projetadas de sincronização ou acesso a recursos.
  - Escalonamento por prioridade.
- Formas de evitar:
  - Aging (envelhecimento): Aumentar gradualmente a prioridade de processos que estão esperando há muito tempo.
  - Políticas de justiça (fairness): Garantir que todos os processos eventualmente sejam atendidos.

Jantar dos Filósofos

- Proposto por Dijkstra como metáfora para os problemas de sincronização e deadlock em sistemas operacionais
- Cinco filósofos estão sentados em uma mesa redonda.
- Cada filósofo pensa e come, alternadamente.
- Entre cada par de filósofos, há um garfo (total de 5 garfos).
- Para comer, um filósofo precisa segurar os dois garfos adjacentes (esquerdo e direito)

Tópicos Adicionais

GNU/Linux: Futex

Fast user space mutex

- Provida pelo kernel Linux como variante ao mutex
- Sem system call para usuário único (*fastpath*)
- System calls para *blocking/waiting* (*slowpath*)
- Documentação: `man futex`

Inteiro com `up()` e `down()`

- Assembly com instruções atômicas de acesso a inteiros

Rust

- Thread safety via uma stack que é fortemente tipada.
- Instruções atômicas.
- Verificação em tempo de compilação.
- Leia mais em Safe systems programming in Rust.

Conclusão: Resumo

- A execução concorrente exige **cuidados com o acesso a dados compartilhados**.
- O problema da **seção crítica** requer soluções como **locks e semáforos**.
- Problemas como **espera indefinida** e **deadlocks** surgem quando há má coordenação de acesso concorrente.
- O uso de **modelos formais**, como o **grafo de alocação de recursos**, ajuda a visualizar e diagnosticar situações de risco.
- A compreensão e implementação correta desses mecanismos são fundamentais para **sistemas operacionais robustos, seguros e eficientes**.

Conclusão: Próximos Passos

- Ler os capítulos 2 e 6 do livro **Sistemas Operacionais Modernos**, de A. TANENBAUM, para mais detalhes.
- Praticar sincronização (locks e mutexes) em linguagem C.

Bibliografia Adicional (para leitores interessados)

- Lamport, L., 2019. The mutual exclusion problem: partII—statement and solutions. In Concurrency: the Works of Leslie Lamport (pp. 247-276).
- Coffman, E.G., Elphick, M. and Shoshani, A., 1971. System deadlocks. ACM Computing Surveys (CSUR), 3(2), pp.67-78.

Dúvidas e Discussão