Sincronização de Processos e Threads

Sistemas Operacionais

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.

Conceitos Fundamentais

- Acesso **concorrente** a dados compartilhados pode resultar em **inconsistências**
- Exemplo: problemas de transações concorrentes em Bancos de Dados. Propriedades ACID.
  - **Lost update**: surge quando duas ou mais transações tentam modificar a mesma informação simultaneamente e uma das modificações é perdida.

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

- **Sincronização**: Manter consistência de dados utilizando **mecanismos** para garantir a execução ordenada de tarefas.

Condição de Corrida - *Race Conditions*

- **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.
- 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.

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.

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 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;`

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)
- 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.
  - **Semáforo de contagem**: Valor inteiro não é restrito.
  - **Semáforo binário**: Mesmo que um mutex.
- Acessado por duas operações atômicas: `wait()` e `signal()`.

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

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

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.

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.

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