

Escalonamento de Processos em Sistemas de Tempo Compartilhado

Condições de Corrida

Pedroso

9 de setembro de 2025

O Problema: Condição de Corrida

- **Definição:** Uma condição de corrida ocorre quando o resultado de uma operação depende da ordem de execução de threads ou processos concorrentes.
- Isso acontece quando múltiplas threads acessam e modificam dados compartilhados simultaneamente.
- Em sistemas com **escalonamento por tempo compartilhado (time-sharing)**, a ordem de execução é não determinística, tornando o resultado imprevisível.

Exemplo em Linguagem C

- **Problema:** Duas threads inserindo itens em um vetor compartilhado.

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

int shared_vector[10000];
int index_shared = 0;

void* add_to_vector(void* arg) {
    for (int i = 0; i < 5000; i++) {
        int local_index = index_shared;
        shared_vector[local_index] = (int)pthread_self();
        index_shared++;
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, add_to_vector, NULL);
    pthread_create(&t2, NULL, add_to_vector, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Elementos: %d\n", index_shared);
    return 0;
}
```

- **Resultado Esperado:** 1000
- **Resultado Real:** Valor imprevisível e menor que 1000.
- Perda de atualizações devido à concorrência na escrita em `index_shared`.

Solução 1: Desabilitar Interrupções

- **Conceito:** O processo desabilita interrupções antes de entrar em sua seção crítica e as habilita ao sair.
- Isso impede que o escalonador mude o contexto da thread (impede a preempção).

Vantagens

- ✓ Simples de entender.
- ✓ Simples de implementar.

- As primeiras versões do Windows usavam este método.
- Na prática o processo que desabilitou interrupções impede que o SO volte a tomar o controle caso ele não habilite interrupções novamente.

Desvantagens

- ✓ **Perigoso:** Desabilitar interrupções por muito tempo pode causar problemas no sistema.
- ✓ Não funciona em sistemas multi-core.

Solução 2: Espera Ocupada (Busy Waiting)

- **Conceito:** A thread espera em um loop até que o recurso esteja disponível.
- Utiliza uma variável de bloqueio (lock).

```
volatile int lock = 0;
void lock_acquire() {
    while (lock == 1) {} // Espera ocupada
    lock = 1;
}
void lock_release() {
    lock = 0;
}
void* add_to_vector_busy_wait(void* arg) {
    for (int i = 0; i < 500; i++) {
        lock_acquire();
        int local_index = index_shared;
        shared_vector[local_index] = (int)pthread_self();
        index_shared++;
        lock_release();
    }
    return NULL;
}
```

- **Problema:** Alto consumo de CPU e pode levar a *deadlock* em sistemas com uma única CPU.

Solução 3: Semáforos

- Uma ferramenta de sincronização para controlar o acesso de múltiplos processos ou threads a um recurso compartilhado.
- Um semáforo é, em sua essência, uma **variável inteira** que, além de seu valor, possui uma fila de espera.
- É a principal alternativa para evitar a "espera ocupada" (busy waiting).

Semáforos: Operações Fundamentais

- As operações em semáforos são **atômicas**, ou seja, não podem ser interrompidas.
- Existem duas operações básicas:
 - ❶ `wait()` (ou `DOWN()`)
 - ❷ `signal()` (ou `UP()`)
- Elas garantem que apenas uma thread por vez possa modificar o semáforo.

O pai da ideia

- O conceito de semáforos foi proposto por Edsger W. Dijkstra em 1965.
- Artigo *Co-operating sequential processes*, EW Dijkstra, 1968, Citado por 3452, não foi publicado em nenhum journal ou conferência, é um relatório interno da Technische Hogeschool Eindhoven (Universidade de Tecnologia de Eindhoven), na Holanda, onde Dijkstra era professor.
- Dijkstra, um dos pioneiros da ciência da computação, utilizou a analogia de um controlador de tráfego (semáforo) para gerenciar o acesso a uma seção crítica.
- <https://pure.tue.nl/ws/files/4279816/344354178746665.pdf>
- As operações que ele definiu foram P (do holandês *PROBEREN TE VERLAGEN*, que significa *tentar diminuir*) e V (de *VERHOOG, aumentar*), que são as bases das operações `wait()` e `signal()` usadas atualmente em sistemas Unix.

A Operação DOWN() (wait()) - V

- A thread *tenta decrementar* o semáforo.
- O que acontece quando wait() é chamado?
 - ❶ O valor do semáforo é decrementado.
 - ❷ **Se o valor for positivo** (≥ 0), a thread continua sua execução.
 - ❸ **Se o valor for negativo** (< 0), a thread é **bloqueada** e colocada em uma fila de espera associada ao semáforo.
- **Importante:** A operação wait() é atômica:
 - ✓ Como Dijkstra definiu, *PROBEREN TE VERLAGEN* ou *tentar diminuir*, se o valor for já zero o semáforo fica em zero (e não negativo -1).
 - ✓ Foi desta forma que ele transferiu o problema da condição de corrida para dentro da operação wait(), como é o SO que implementa a função, o código interno da wait() nunca será interrompido dentro desta função!!!

A Operação DOWN() (wait()) - V

► Pseudocódigo:

```
wait(semáforo S) {  
    // Diminui o valor do semáforo  
    S.valor = S.valor - 1;  
    // Se o valor for negativo, bloquear em S  
    se (S.valor < 0) {  
        Bloquear esta thread no semarofa S;  
    }  
}
```

A Operação UP() (signal()) P

- A thread sinaliza que terminou de usar o recurso.
- O que acontece quando `signal()` é chamado?
 - ➊ O valor do semáforo é incrementado.
 - ➋ **Se houver threads bloqueadas** neste semáforo, elas serão **desbloqueadas**.
 - ➌ **Se não houver threads bloqueadas**, o valor é simplesmente incrementado.
- **Importante:** A operação `signal()` é atômica: nunca será interrompido também dentro da função.

A Operação UP() (signal()) P

► Pseudocódigo para compreensão:

```
post(semáforo S) {  
    // Aumenta o valor do semáforo  
    S.valor = S.valor + 1;  
    // Se o valor for menor ou igual a zero, há threads esperando  
    se (S.valor > 0) {  
        Desbloquear todas as threads bloqueadas em S;  
    }  
}
```

► Pseudocódigo usado na prática¹:

```
post(semáforo S) {  
    // Aumenta o valor do semáforo  
    S.valor = S.valor + 1;  
    // Se o valor for menor ou igual a zero, há threads esperando  
    se (S.valor > 0) {  
        Desbloquear a thread esperando a mais tempo (FIFO) em S;  
    }  
}
```

¹No primeiro código, pode haver o problema conhecido como *thundering herd* (manada estrondosa). Todas as threads acordadas tentariam imediatamente adquirir o mesmo recurso, competindo por ele. Apenas uma conseguiria, e todas as outras voltariam a se bloquear, desperdiçando ciclos de CPU e tempo do escalonador. Dijkstra propôs já a segunda versão, o que nos faz pensar no grau de compreensão do problema que ele tinha já naquela época.

Exemplo Conceitual

- Protegendo uma seção crítica com um semáforo binário (mutex).
- O semáforo é inicializado com o valor 1.

```
sem_t meu_semaforo;
```

```
void secao_critica_protegida() {  
    sem_wait(&meu_semaforo);  // UP  
  
    //  Início da Seção Crítica  
    //  Apenas uma thread pode estar aqui por vez.  
    //  Acessa e modifica o recurso compartilhado.  
    //  Fim da Seção Crítica  
  
    sem_post(&meu_semaforo);  // DOWN  
}
```

Solução 3: Usando Semáforos

- **Conceito:** Variável inteira com duas operações atômicas: `wait()` (`P()`) e `signal()` (`V()`).
- Quando o recurso não está disponível, a thread é **bloqueada**, liberando a CPU.

```
#include <semaphore.h>
sem_t semaphore;
void* add_to_vector_semaphore(void* arg) {
    for (int i = 0; i < 500; i++) {
        sem_wait(&semaphore); // Aguarda (DOWN)
        int local_index = index_shared;
        shared_vector[local_index] = (int)pthread_self();
        index_shared++;
        sem_post(&semaphore); // Libera (UP)
    }
    return NULL;
}
int main() {
    sem_init(&semaphore, 0, 1);
    // ... criação e junção de threads ...
    sem_destroy(&semaphore);
    return 0;
}
```

- **Vantagens:** Eficiente, evita consumo de CPU e é a solução mais comum em sistemas modernos.