

Computação Concorrente (DCC/UFRJ)

Módulo 3 - Semana 1: Sincronização por exclusão mútua e condicional usando semáforos

Prof. Silvana Rossetto

Dezembro 2020

Conceito de semáforo

- Proposto por Dijkstra, em 1965, como um mecanismo para suporte à cooperação entre processos dentro de um sistema operacional
- Baseia-se no princípio de **troca de sinais**: *uma thread bloqueia a sua execução em um ponto específico do código até que ela receba um sinal que a desbloqueie*



Um **semáforo** é uma variável inteira com quatro operações básicas, todas elas executadas de forma **atômica**:

- ❶ **inicialização** (**sem_init()**): inicia o semáforo com valor não negativo
- ❷ **decremento** (**sem_wait()**): “pode resultar no bloqueio da thread”
- ❸ **incremento** (**sem_post()**): “pode resultar no desbloqueio de outra thread”
- ❹ **finalização** (**sem_destroy()**): desaloca e finaliza o semáforo

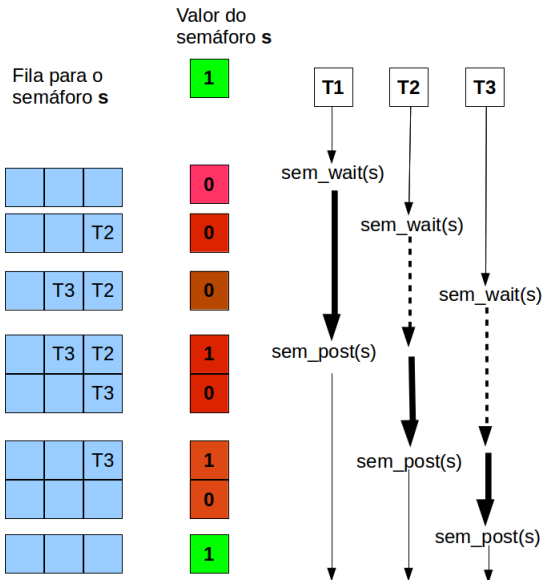
A operação **sem_wait()**:

- se o valor corrente do semáforo é **maior que 0**, decrementa esse valor de 1 e retorna
- se o valor corrente do semáforo é **0**, então a chamada bloqueia até o valor do semáforo ser maior que 0

A operação **sem_post()**:

- incrementa o valor do semáforo de 1
- se há alguma thread esperando (bloqueada em `sem_wait`), desbloqueia uma thread

Exemplo de execução com semáforos



Semáforos binários (para exclusão mútua)

- O semáforo **sem** é inicializado com valor 1 (**semáforo binário!**)
- A entrada na seção crítica é implementada executando a operação **sem_wait(&sem)**
- A saída da seção crítica é implementada executando a operação **sem_post(&sem)**

```
while (true) {  
    sem_wait(&sem);  
    //executa a secao critica  
    sem_post(&sem);  
    //executa fora da secao critica  
}
```

- Para um **semáforo binário**, se duas chamadas `sem_wait()` são feitas sem uma chamada `sem_post()` intermediária, a segunda chamada irá bloquear a thread
- Para o caso de **locks recursivos**, se a thread que está de posse do lock o requisita novamente, essa thread não é bloqueada

- Para o caso de **locks**, chamadas sucessivas das operações de **lock()** e **unlock()** devem ser feitas pela thread proprietária do lock
- Para o caso de **semáforos binários**, chamadas sucessivas de **sem_wait()** e **sem_post()** podem ser feitas por diferentes threads

Semáforos binários versus semáforos contadores

Semáforo binário

Semáforo iniciado com **valor 1** (normalmente usado para implementar **sincronização por exclusão mútua**)

Semáforo contador

Semáforo iniciado com **valor N** (normalmente usado para implementar **sincronização por condição**)

Condições da aplicação

- Os produtores não podem inserir novos elementos quando a área de dados já está cheia
- Os consumidores não podem retirar elementos quando a área de dados já está vazia
- Os elementos devem ser retirados na mesma ordem em que foram inseridos
- Os elementos inseridos não podem ser perdidos (sobreescritos por novos elementos)
- Um elemento só pode ser retirado por um consumidor

Exemplo: problema produtor/consumidor

```
void *produtor(void * arg) {
    int elemento;
    while(1) {
        //produz um elemento....
        Insere(elemento);
    }
    pthread_exit(NULL);
}

void *consumidor(void * arg) {
    int elemento;
    while(1) {
        elemento = Retira();
        //consome o elemento....
    }
    pthread_exit(NULL);
}
```

Exemplo: UM produtor e UM consumidor

```
// Variaveis globais
sem_t slotCheio, slotVazio;//condicao
int Buffer[N];
...
sem_init(&slotCheio, 0, 0);
sem_init(&slotVazio, 0, N);
```

Exemplo: UM produtor e UM consumidor

```
void Insere (int item) {  
    static int in=0;  
    //aguarda slot vazio  
    sem_wait(&slotVazio);  
    Buffer[in] = item;  
    in = (in + 1) % N;  
    //sinaliza um slot cheio  
    sem_post(&slotCheio);  
}
```

Exemplo: UM produtor e UM consumidor

```
int Retira (void) {  
    int item;  
    static int out=0;  
    //aguarda slot cheio  
    sem_wait(&slotCheio);  
    item = Buffer[out];  
    out = (out + 1) % N;  
    //sinaliza um slot vazio  
    sem_post(&slotVazio);  
    return item;  
}
```

Exemplo: vários produtores e consumidores

```
// Variaveis globais
sem_t slotCheio, slotVazio;//condicao
sem_t mutexProd, mutexCons;//exclusao mutua
int Buffer[N];
...
sem_init(&mutexCons, 0, 1);
sem_init(&mutexProd, 0, 1);
sem_init(&slotCheio, 0, 0);
sem_init(&slotVazio, 0, N);
```

Exemplo: vários produtores e consumidores

```
void Insere (int item) {  
    static int in=0;  
    //aguarda slot vazio  
    sem_wait(&slotVazio);  
    //exclusao mutua entre produtores  
    sem_wait(&mutexProd);  
    Buffer[in] = item;  
    in = (in + 1) % N;  
    sem_post(&mutexProd);  
    //sinaliza um slot cheio  
    sem_post(&slotCheio);  
}
```


Exemplo: vários produtores e consumidores

```
int Retira (void) {  
    int item;  
    static int out=0;  
    //aguarda slot cheio  
    sem_wait(&slotCheio);  
    //exclusao mutua entre consumidores  
    sem_wait(&mutexCons);  
    item = Buffer[out];  
    out = (out + 1) % N;  
    sem_post(&mutexCons);  
    //sinaliza um slot vazio  
    sem_post(&slotVazio);  
    return item;  
}
```

Referências bibliográficas

- ① *Concurrent Programming — Principles and Practice*, Andrews, Addison-Wesley, 1991
- ② *Modern Multithreading*, Carver e Tai, Wiley, 2006
- ③ *Arquitetura e Organização de Computadores*, Stallings, Pearson, ed. 8, 2010