Programação Concorrente (IC/CCMN/UFRJ)

Sincronização por exclusão mútua e condicional usando semáforos

Profa. Silvana Rossetto

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*

Definição de semáforo

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

1. **inicialização** (**sem_init()**): inicia o semáforo com valor não negativo
2. **decremento** (**sem_wait()**): "pode resultar no bloqueio da thread"
3. **incremento** (**sem_post()**): "pode resultar no desbloqueio de outra thread"
4. **finalização** (**sem_destroy()**): desaloca e finaliza o semáforo

Operações básicas sobre semáforos

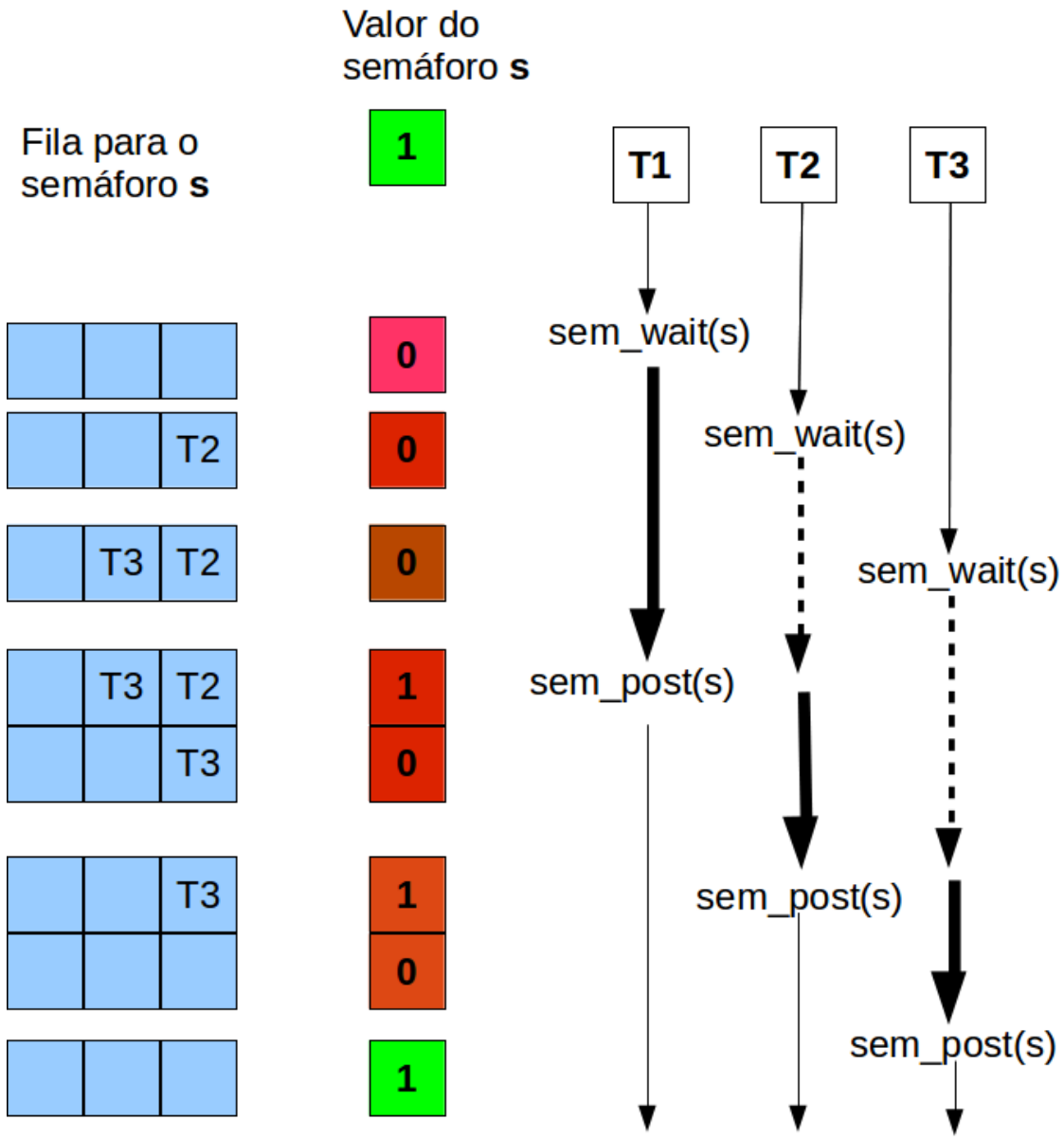
A operação **sem_wait()**:

- 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
}
```

Locks versus semáforos binários

- 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

Locks versus semáforos binários

- 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)

Exemplo: problema produtor/consumidor

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;
```

Sincronização coletiva (barreira) com semáforos

- Projetar uma função **void barreira(int nthreads)** para implementar sincronização coletiva
- O parâmetro `nthreads` informa o número total de threads que devem participar da barreira
- Todas as threads deverão chamar essa função no ponto do código onde a sincronização por barreira é requerida
- Usar **semáforos** para sincronização por condição e por exclusão mútua

Essa solução está correta?

```
sem_t mutex; //exclusao mutua (iniciado com 1)
sem_t cond; //condicional (iniciado com 0)
int chegaram=0; //variavel de estado global
void barreira(int numThreads) {
    sem_wait(&mutex);
    chegaram++;
    if (chegaram < numThreads) {
      sem_post(&mutex);
      sem_wait(&cond);
    } else {
      for(int i=1; i<numThreads; i++)
         { sem_post(&cond); }
      chegaram = 0;
      sem_post(&mutex);
    }
}
```

Problema leitor/escritor

Condições básicas do problema:

- Os leitores podem ler simultaneamente uma região de dados compartilhada
- Apenas um escritor pode escrever a cada instante em uma região de dados compartilhada
- Se um escritor está escrevendo, nenhum leitor pode ler a mesma região de dados compartilhada

Leitores

```
int leitores=0; //leitores lendo
sem_t mutex, escrita; //inicializados com 1
void leitor(){
   sem_wait(&mutex);
   leitores++;
   if(leitores==1) //primeiro leitor
      { sem_wait(&escrita); }
   sem_post(&mutex);
   /* faz a leitura */
   sem_wait(&mutex);
   leitores--;
   if(leitores==0) //ultimo leitor
      { sem_post(&escrita); }
   sem_post(&mutex);
}
```

Escritores

```
void escritor(){
    sem_wait(&escrita);
    /* faz a escrita */
    sem_post(&escrita);
}
```

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