

# Computação Concorrente (DCC/UFRJ)

## Aula 7: Sincronização por barreira e outros problemas de concorrência

Prof. Silvana Rossetto

17 de abril de 2012

- 1 Sincronização por barreira
- 2 Outros problemas de concorrência
  - Problema da barbearia
- 3 Exercícios

# Sincronização por barreira

- Vários problemas computacionais são resolvidos usando **algoritmos iterativos que sucessivamente computam aproximações melhores para uma resposta procurada**
  - Normalmente esses algoritmos manipulam um vetor de valores e cada iteração executa a mesma computação sobre todos os elementos do vetor
- 
- É possível **usar várias threads para computar partes disjuntas da solução de forma concorrente/paralela**
  - **Um requisito comum é que cada iteração depende da anterior, então as threads devem aguardar a próxima iteração**

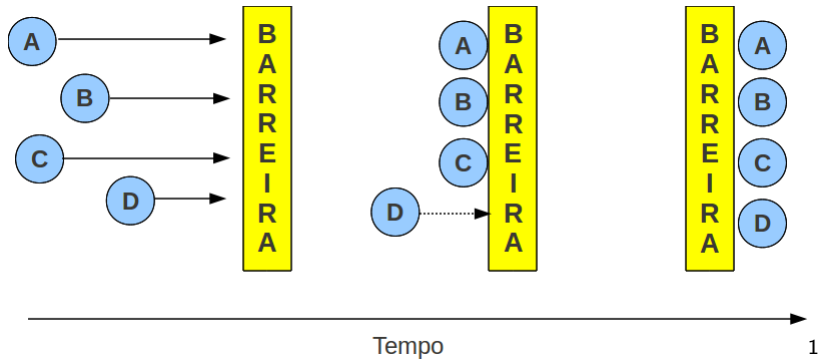
# Sincronização por barreira

Para garantir que as threads trabalhem sempre em fase (na mesma iteração) é necessário usar um tipo de sincronização chamada **sincronização por barreira**

## Barreira

Um tipo de sincronização coletiva que suspende a execução das threads de um aplicação em um dado ponto do código e somente permite que as threads prossigam quando todas elas tiverem chegado aquele ponto

## Exemplo sincronização por barreira



<sup>1</sup>Fonte:[3]

# Implementação de sincronização por barreira

- 1 Uma maneira simples de implementar uma barreira é usar um **contador que é inicializado com o número total de threads envolvidas**
- 2 Cada thread decrementa o contador após alcançar a barreira e então se bloqueia esperando o contador chegar a zero
- 3 Quando o contador chega a zero todas as threads são desbloqueadas

## Exemplo de implementação de sincronização por barreira

```
int cont = NTHREADS;  
sem_t em, sem_t continua[NTHREADS]; sem_init(&em, 0, 1);  
for(t=0; t<NTHREADS; t++) {  
    sem_init(&continua[t], 0, 0); }
```

```
for (i=0; i<NITER; i++) {  
    //executa a computacao de cada iteracao...  
    sem_wait(&em); //entrada na SC  
    cont--;  
    if (cont == 0) {  
        cont=NTHREADS;  
        for (t=0; t<NTHREADS; t++) {  
            sem_post(&continua[t]);  
        }  
    }  
    sem_post(&em); //saida da SC  
    sem_wait(&continua[tid]);  
}
```

## Exemplo de implementação de sincronização por barreira

```
int cont = 0;  
sem_t chegada, sem_t partida;  
sem_init(&chegada, 0, 1); sem_init(&partida, 0, 0);
```

```
for (i=0; i<NITER; i++) {  
    //executa a computacao de cada iteracao...  
  
    sem_wait(&chegada); //chegou na barreira  
    cont++;  
    if (cont < NTHREADS) { sem_post(&chegada); }  
    else { sem_post(&partida); }  
  
    sem_wait(&partida); //espera barreira  
    cont--;  
    if (cont > 0) { sem_post(&partida); }  
    else { sem_post(&chegada); }  
}
```



# Problema da barbearia



2

<sup>2</sup>Fonte: [www.cs.uml.edu](http://www.cs.uml.edu)

## Uma configuração do problema

- 1 barbeiro e uma área de espera que pode acomodar 5 clientes sentados
- Um cliente não pode entrar na barbearia se a sala estiver cheia
- Quando o barbeiro fica livre, o cliente há mais tempo em uma das cadeiras é atendido
- O barbeiro divide o tempo entre cortar cabelo e dormir esperando por clientes

## Solução para o problema usando semáforos

```
int esperando = 0;  
sem_t em, barbeiro, clientes;  
sem_init(&em,0,1); sem_init(&barbeiro,0,0); sem_init(&clientes,0,0);
```






```
void Barbeiro() {  
    while(1) {  
        sem_wait(&clientes);  
        sem_wait(&em);  
        esperando = esperando - 1;  
        sem_post(&barbeiro);  
        sem_post(&em);  
        //corta o cabelo do cliente  
    }  
}
```

```
void Cliente() {  
    sem_wait(&em);  
    if (esperando < CADEIRAS) {  
        esperando = esperando + 1;  
        sem_post(&clientes);  
        sem_post(&em);  
        sem_wait(&barbeiro);  
        //senta na cadeira do barbeiro  
    } else sem_post(&em);  
}
```

3

<sup>3</sup>Fonte: Stallings, 2009.

## Garantia de exclusão mútua na SC?

```
//variáveis compartilhadas por T0 e T1
boolean e0=false, e1=true; int aux=2;
T0: 
while(true) { 
  (1)   e0=true; aux=1;
  (2)   while(e1 && aux==1) {;} //SC
  (3)   e0 = false;
} 
T1:
while(true) {
  (1)   e1=true; aux=0;
  (2)   while(e0 && aux==0) {;} //SC
  (3)   e1 = false;
}
```

## Solução

**Problema:** Se T0 quer entrar na sua SC e T1 não está na sua SC e nem deseja entrar, então  $e0$  é true,  $aux$  é 1 e  $e1$  é true (valor inicial), nesse caso T0 fica impedida de executar a sua SC até que T1 execute a sua seção de entrada (fazendo  $aux$  ser 0)

Do mais o código funciona corretamente:

- se uma thread quer entrar na sua SC e a outra thread já está na SC, a primeira thread fica impedida de entrar até que a segunda execute a linha 3
- se ambas as threads tentarem entrar na SC ao mesmo tempo, a thread que setar  $aux$  primeiro conseguirá entrar e a outra ficará em espera ocupada
- *possibilidade de starvation?*

## Thread para impressão em background?

Projete uma aplicação que implemente esse algoritmo usando duas threads: uma que implementa o algoritmo de fato e outra que apenas imprime o estado atual do vetor a cada iteração do for mais externo do algoritmo

```
void bubbleSort(int v[]) {  
    for (int i = v.length; i >= 1; i--) {  
        for (int j = 1; j < i; j++) {  
            if (v[j - 1] > v[j]) {  
                int aux = v[j]; v[j]=v[j - 1]; v[j-1]=aux;  
            } } } }
```

## Solução

```
sem_t t1, t2; //ambos começam com 0
T1:
for (int i = v.length; i >= 1; i--) {
    sem_post(&t2); sem_wait(&t1);
    for (int j = 1; j < i; j++) {
        if (v[j - 1] > v[j]) {
            int aux = v[j]; v[j]=v[j - 1]; v[j-1]=aux;
        }
    }
}
T2:
while(1) {
    sem_wait(&t2);
    //imprime vetor
    sem_post(&t1);
}
```

## O problema do urso e o pote de mel

- Um urso e  $N$  abelhas compartilham um pote de mel: as abelhas colocam porções de mel no pote até ele ficar cheio
- O urso dorme enquanto o pote não fica cheio, come todo o mel e volta a dormir
- O pote é um recurso compartilhado, então no máximo uma abelha ou o urso podem acessá-lo a cada instante



# Solução

```
sem_t urso; //começa com 0
sem_t abelha; //começa com 1
Urso:
while(1) {
    sem_wait(&urso);
    comeMel();
    sem_post(&abelha);
}
Abelhas:
while(1) {
    sem_wait(&abelha);
    colocaPorçãoPote();
    if(poteCheio()) sem_post(&urso);
    else sem_post(&abelha);
}
```



## Qual é a condição lógica para executar a linha 11?

```
1: sem_t em; /*começa em 1*/ sem_t cond; /*começa em 0*/  
2: int ativo=0, esperando=0, deveEsperar=0;  
3: while(1) {  
4:     sem_wait(&em);  
5:     if(deveEsperar){  
6:         esperando++; sem_post(&em); sem_wait(&cond);  
7:         esperando--;}  
8:     ativo++; deveEsperar = (ativo==5) ? 1: 0;  
9:     if(esperando>0 && !deveEsperar)  
10:         sem_post(&cond); else sem_post(&em);  
11:     /* faz algo */  
12:     sem_wait(&em); ativo--;  
13:     if(ativo==0) deveEsperar=0;  
14:     if(esperando>0 && !deveEsperar) sem_post(&cond);  
15:     else sem_post(&em);  
}
```

# Solução

Trata-se do acesso a um recurso compartilhado com as seguintes características:

- enquanto o número de threads usando o recurso é menor que 5, novas threads podem usar o recurso imediatamente
- quando o limite de 5 threads usando o recurso é alcançado, todas elas devem liberar o recurso para que qualquer outra thread possa usá-lo
- *possibilidade de starvation?*
- *possibilidade de deadlock?*

## Referências bibliográficas

- 1 *Concurrent Programming — Principles and Practice*, Andrews, Addison-Wesley, 1991
- 2 *Modern Multithreading*, Carver e Tai, Wiley, 2006
- 3 *Synchronization Algorithms and Concurrent Programming*, G. Taubenfeld, Pearson/Prentice Hall, 2006