

# Computação Concorrente (DCC/UFRJ)

## Aula 6: Padrões de uso de semáforos em problemas clássicos de concorrência

Prof. Silvana Rossetto

10 de abril de 2012

- 1 Padrões de uso de semáforos
  - Semáforos para seção crítica
  - Semáforos para alocação de recursos
  - Semáforos e variável contadora para alocação de recursos
  
- 2 Outros problemas clássicos de concorrência
  - Problema dos leitores e escritores
  - Problema do jantar dos filósofos

# Mecanismos de sincronização

## Locks e semáforos binários

- Tipos simples de **mecanismo de escalonamento**:
  - o recurso escalonado é a memória compartilhada (**instruções de uma seção crítica**)
  - a política de escalonamento é uma thread de cada vez usando o recurso

## Semáforos contadores

- Necessidade de expressar outros requisitos/políticas de escalonamento
- As threads devem BLOQUEAR **até que um evento ocorra** ou **até que um recurso fique disponível**

# Semáforos binários versus semáforos contadores

## Semáforo binário

Semáforo iniciado com **valor 1** é conhecido como **semáforo binário** (normalmente usado para implementar **sincronização por exclusão mútua**)

## Semáforo contador

Semáforo iniciado com **valor N** é conhecido como **semáforo contador** (normalmente usado para implementar **sincronização por condição**)

# Semáforos para seção crítica

- 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  
}
```

# Semáforos para alocação de recursos

- Coordenar a alocação de um recurso, **ou de várias réplicas de um mesmo recurso**, para threads concorrentes
- Quando nenhuma réplica está disponível, a **thread deve esperar até que uma das réplicas seja liberada por outra thread**
- Um **semáforo contador** pode ser usado para resolver esse problema

# Exemplo de semáforos para alocação de recursos

```
sem_t s;  
sem_init(&s, 0, M) // M igual a número de réplicas do recurso
```

**T1:**

```
sem_wait(&s);  
//seção crítica  
sem_post(&s);
```

**T2:**

```
sem_wait(&s);  
//seção crítica  
sem_post(&s);
```

**T3:**

```
sem_wait(&s);  
//seção crítica  
sem_post(&s);
```

# Problema dos produtores e escritores



Threads produzem/consomem dados usando uma área de armazenamento comum

- As **threads produtoras** geram/depositam elementos
- As **threads consumidoras** consomem/processam elementos



# Problema clássico: produtor/consumidor

```
const int tam_buffer = N;
T_item buffer[N];
sem_t EM=1;
sem_t CHEIO=0, VAZIO=N;

void Produtor() {
    while(true) {
        produz_item();
        sem_wait(VAZIO);
        sem_wait(EM);
        insere_item(buffer);
        sem_post(EM);
        sem_post(CHEIO);
    }
}
```

```
void Consumidor() {
    sem_wait(CHEIO);
    sem_wait(EM);
    retira_item(buffer);
    sem_post(EM);
    sem_post(VAZIO);
    consome_item();
}

void main() {
    //inicia threads prod/consum
}
```

# Semáforos e variável contadora para alocação de recursos

- Alternativa para alocação de recursos é usar uma **variável contadora** (*count*) para gerir o **as réplicas do recurso**
- Quando o **valor da variável é maior que zero**, o recurso está disponível, e quando é **igual ou menor que zero**, o recurso não está disponível


A variável é decrementada quando um recurso é alocado para uma thread e incrementada quando o recurso é devolvido

# Semáforos e variável contadora para alocação de recursos


- Outra variável compartilhada (*waiting*) é usada para contabilizar o **número de threads aguardando pelo recurso**
  - Quando uma thread deseja usar o recurso ela checa o valor de **count**
- 
- Se  $count \leq 0$ , a thread incrementa o valor de **waiting** e bloqueia em um semáforo binário
  - Quando uma thread libera o recurso, ela checa o valor de **waiting**:
  - Se  $waiting > 0$ , uma thread bloqueada é sinalizada, senão a variável **count** é incrementada

# Exemplo de semáforos e variável contadora para alocação de recursos

```
int count = N, waiting = 0;  
sem_t em, recDisp; sem_init(&em, 0, 1); sem_init(&recDisp, 0, 0);
```



```
1: sem_wait(&em);  
2: if(count > 0) {  
3:     count--;  
4:     sem_post(&em);  
5: } else {  
6:     waiting++;  
7:     sem_post(&em);  
8:     sem_wait(&recDisp);  
9: }  
10: //seção crítica (usa o recurso)
```



```
11: sem_wait(&em);  
12: if(waiting > 0) {  
13:     waiting--;  
14:     sem_post(&recDisp);  
15: } else count++;  
16: sem_post(&em);
```

# Problema dos leitores e escritores

- Uma área de dados (ex., arquivo, bloco da memória, tabela de um banco de dados) é compartilhada entre diferentes threads
- As **threads leitoras** apenas lêem o conteúdo da área de dados
- As **threads escritoras** apenas escrevem dados nessa área

## Exemplo de problema real

Em um sistema de reservas de passagens aéreas, vários usuários podem inspecionar concorrentemente os assentos disponíveis, mas um usuário que está reservando um assento deve ter acesso exclusivo à base de dados

# Problema dos leitores e escritores

## Condições para o problema dos leitores/escritores:

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

# Exemplo de solução para leitores e escritores



```
int contLeit = 0; sem_t em, escrita;  
sem_init(&em, 0, 1); sem_init(&escrita, 0, 1);
```

```
void leitor() {  
    while(1) {  
        sem_wait(&em);  
        contLeit++;  
        if(contLeit == 1)  
            sem_wait(&escrita);  
        sem_post(&em);  
        // seção crítica: leituras...  
        sem_wait(&em);  
        contLeit--;  
        if(contLeit == 0)  
            sem_post(&escrita);  
        sem_post(&em);  
    }  
}
```

```
void escritor() {  
    while(1) {  
        sem_wait(&escrita);  
        //seção crítica: escritas...  
        sem_post(&escrita);  
    }  
}
```

# Problema do jantar dos filósofos





# Definição do problema

- 5 filósofos estão sentados em uma mesa circular
- Cada filósofo tem um prato de *spaghetti* e ao lado de cada prato dois garfos que são necessários para comer o *spaghetti*
- A rotina de cada filósofo consiste em alternar períodos em que ele **pensa** e outro que ele **come**
- Quando o filósofo sente fome ele **tenta pegar os garfos à esquerda e à direita, um de cada vez, sempre nessa ordem**
- Se ele consegue os dois garfos, ele come e depois devolve os garfos, e volta a pensar (**no máximo dois filósofos conseguem comer ao mesmo tempo**)

# Problema do jantar dos filósofos

- Exemplo de problema que precisa **lidar com a coordenação de recursos compartilhados**
- A solução deve satisfazer:
  - 1 **exclusão mútua**: dois filósofos não podem usar os mesmos garfos (recursos) ao mesmo tempo
  - 2 **justiça**: todos os filósofos devem conseguir comer em algum momento

# Primeira solução para o problema

```
void filosofo (int i) {  
    while(true) {  
        pensa();  
        pega_garfo(i);  
        pega_garfo((i+1)%N);  
        come();  
        devolve_garfo(i);  
        devolve_garfo((i+1)%N);  
    }  
}
```

1

<sup>1</sup>Fonte: Tanenbaum/Hoodhull, 2006.

# Problema dessa solução: *deadlock*

Suponha que todos os filósofos peguem o garfo esquerdo simultaneamente, nenhum deles será capaz de pegar o garfo direito e o programa entrará em **deadlock**

**Deadlock** é um estado no qual nenhuma thread consegue avançar porque está bloqueada esperando por um recurso alocado a outra thread que também encontra-se bloqueada

# Solução alternativa

Uma alternativa para resolver esse problema seria:


- **após conseguir o garfo à esquerda o programa checa se o garfo à direita está liberado, se não estiver ele libera o garfo à esquerda**

# Problema da solução alternativa: *starvation*

Essa estratégia pode levar a outro problema, chamado **starvation** (inanição):

- suponha que todos os filósofos peguem o garfo à esquerda ao mesmo tempo, eles verão o garfo à direita ocupado, devolverão o garfo à esquerda e a mesma situação poderá se repetir
- **as threads não progridem, embora nenhuma delas esteja bloqueada**

# Solução para o problema usando semáforos

```
sem em = 1;   
sem estado[N] = [0..0]; //N=num. de filósofos  
  
void filosofo(int i) {  
    while (true) {  
        pensa();  
        pega_garfos(i);  
        come();  
        devolve_garfos(i);  
    }  
}
```

2

<sup>2</sup>Fonte: Tanenbaum/Hoodhull, 2006.

# Solução para o problema usando semáforos

```
void pega_garfos(int i) {  
    semWait (em);  
    estado[i] = FAMINTO;  
    teste(i);  
    semSignal (em);  
    semWait (estado[i]);  
}
```



```
void devolve_garfos(int i) {  
    semWait (em);  
    estado[i] = PENSANDO;  
    teste(ESQ);  
    teste(DIR);  
    semSignal (em);  
}
```

```
void teste (int I) {  
    if (estado[i] == FAMINTO && estado[ESQ] != COMENDO &&  
        estado[DIR] != COMENDO) {  
        estado[i] = COMENDO;  
        semSignal(estado[i]);  
    }  
}
```

3

<sup>3</sup>Fonte: Tanenbaum/Hoodhull, 2006.



# Referências bibliográficas

- ❶ *Concurrent Programming — Principles and Practice*, Andrews, Addison-Wesley, 1991
- ❷ *Programming Language Pragmatics*, Scott, Morgan-Kaufmann, ed. 2, 2006
- ❸ *Operating Systems – Internals and Design Principles*, Stallings, Pearson, ed. 6, 2009
- ❹ *Modern Multithreading*, Carver e Tai, Wiley, 2006