BC1518 - Sistemas Operacionais

Aula 7: Sincronização de Processos

Tópicos desta aula

- Fundamentos
- O Problema da Seção Crítica
- Hardware de Sincronização
- Semáforos
- Problemas Clássicos de Sincronização
- Monitores
- Sincronismo em Java

Fundamentos

➢**Processo cooperativo:** aquele que pode afetar ou ser afetado por outros processos em execução

- Compartilham dados na memória principal ou utilizam um arquivo compartilhado

➢**Acesso concorrente** a **dados compartilhados** pode resultar em **inconsistência de dados e conflitos**

- Exemplos:
  - Dois processos em um sistema de reserva de passagens aéreas tentando reservar o último assento para clientes distintos (banco de dados compartilhado)
  - Comércio eletrônico, vários processos clientes para comprar um mesmo produto
  - O problema Produtor-Consumidor, que compartilham um *buffer* (produtor insere algum item e consumidor remove)

➢Manter a **consistência dos dados exige mecanismos** para garantir a correta execução dos processos ou *threads* cooperativos

Acesso concorrente à memória compartilhada

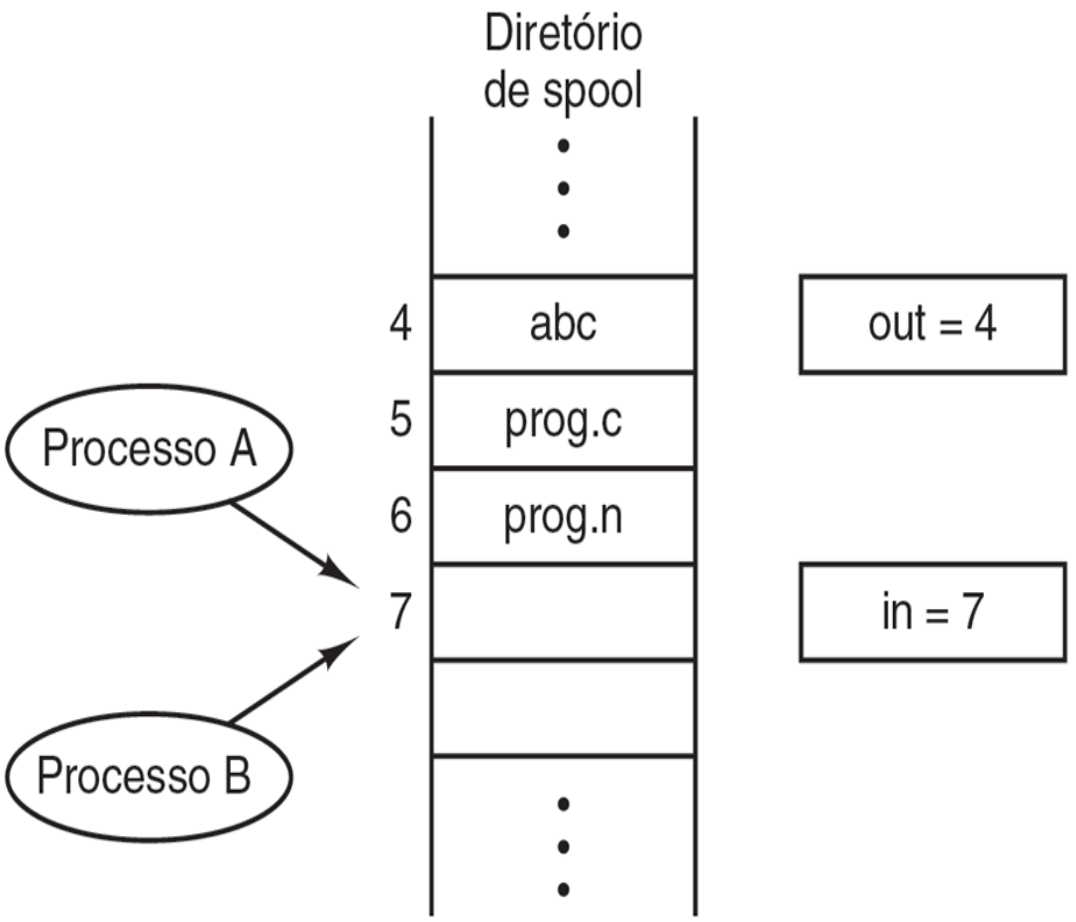


Figura 2.16 Dois processos querem acessar a memória compartilhada ao mesmo tempo. [Tanenbaum]

- Variáveis compartilhadas:
  - in: aponta para a próxima vaga no diretório
  - out: aponta para o próximo arquivo a ser impresso
- Processos A e B decidem enviar um arquivo para impressão (quase que simultaneamente)
- A obtém o valor de in, mas logo em seguida é interrompido (preempção)
- B obtém o mesmo valor de in, armazena o nome do arquivo nesta posição (7) e incrementa o valor de in (8)
- A volta a executar de onde parou, armazena o nome do arquivo nesta posição (7) e incrementa o valor de in (8), sobrepondo o arquivo de B
- B nunca receberá a saída

Problema do Produtor-Consumidor

O código para o Produtor:

```
public void insert(Object item) {
while (count == BUFFER_SIZE)
    ; // não faz nada -- buffer cheio
// acrescenta um item ao buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
}
```

Produtor e consumidor manipulam a variável **cont** concorrentemente. Essa variável contém o número de itens no buffer.
Qual problema pode ocorrer?

O código para o Consumidor:

```
public Object remove() {
Object item;
while (count == 0)
    ; // não faz nada – nada para consumir
// remove um item do buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
return item;
}
```

Produtor-Consumidor: variável compartilhada

count++ pode ser implementado (em linguagem de máquina) como:

```
register1 = count
register1 = register1 + 1
count = register1
```

count-- pode ser implementado como:

```
register2 = count
register2 = register2 - 1
count = register2
```

Considere esta sequência de execuções intercaladas (count = 5):

S0: produtor executa register1 = count	{register1 = 5}
S1: produtor executa register1 = register1 + 1	{register1 = 6}
S2: consumidor executa register2 = count	{register2 = 5}
S3: consumidor executa register2 = register2 - 1	{register2 = 4}
S4: produtor executa count = register1	{count = 6}
S5: consumidor executa count = register2	{count = 4}

(O correto seria **count = 5**, pois o produtor colocou um item (no BUFFER), que foi retirado pelo consumidor!)

Condição de corrida

➢Situações como essa, quando dois ou mais processos estão lendo ou escrevendo algum dado compartilhado e cujo resultado final depende exatamente de quem está executando e quando são chamadas de condição de corrida (*race conditions*)

- A depuração de programas que contenham condição de corrida é complicada, geralmente os testes não apresentam problemas, mas em um dado momento, o programa pode ter um comportamento estranho e inexplicável

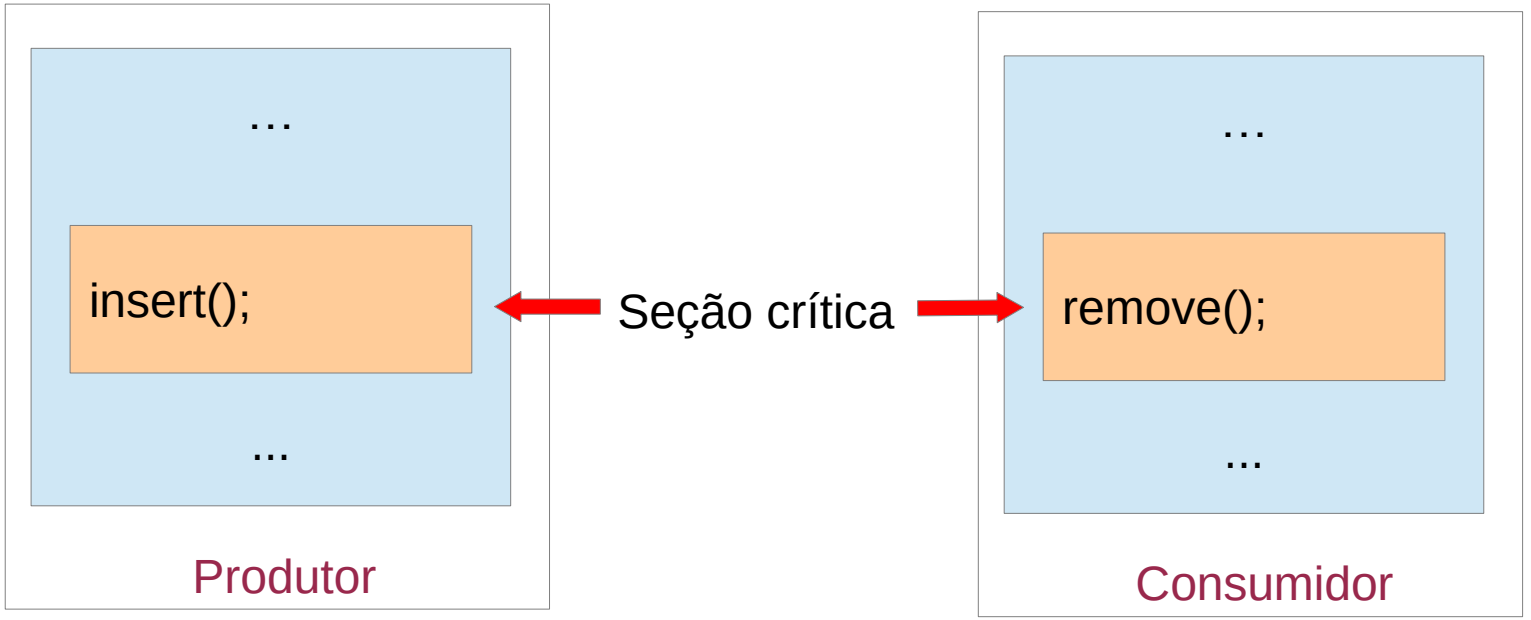
➢É preciso garantir que somente um processo poderá acessar o dado compartilhado

- Em outras palavras, é preciso de exclusão mútua (*mutual exclusion*) i.e., um modo de assegurar que outros processos sejam impedidos de usar uma variável ou arquivo compartilhado que já estiver em uso por um processo

➢A escolha das operações primitivas adequadas para realizar a

Seção crítica

➢Em cada processo, a parte de código em que há o acesso a dados compartilhados (em memória, arquivos) é chamada de região crítica (*critical region*) ou seção crítica (*critical section*)



➢O importante é garantir que quando um processo estiver executando em sua região crítica, nenhum outro processo poderá executar em sua região crítica (ou seja, dois processos nunca executarão suas regiões críticas ao mesmo tempo)

Exclusão mútua com seções críticas

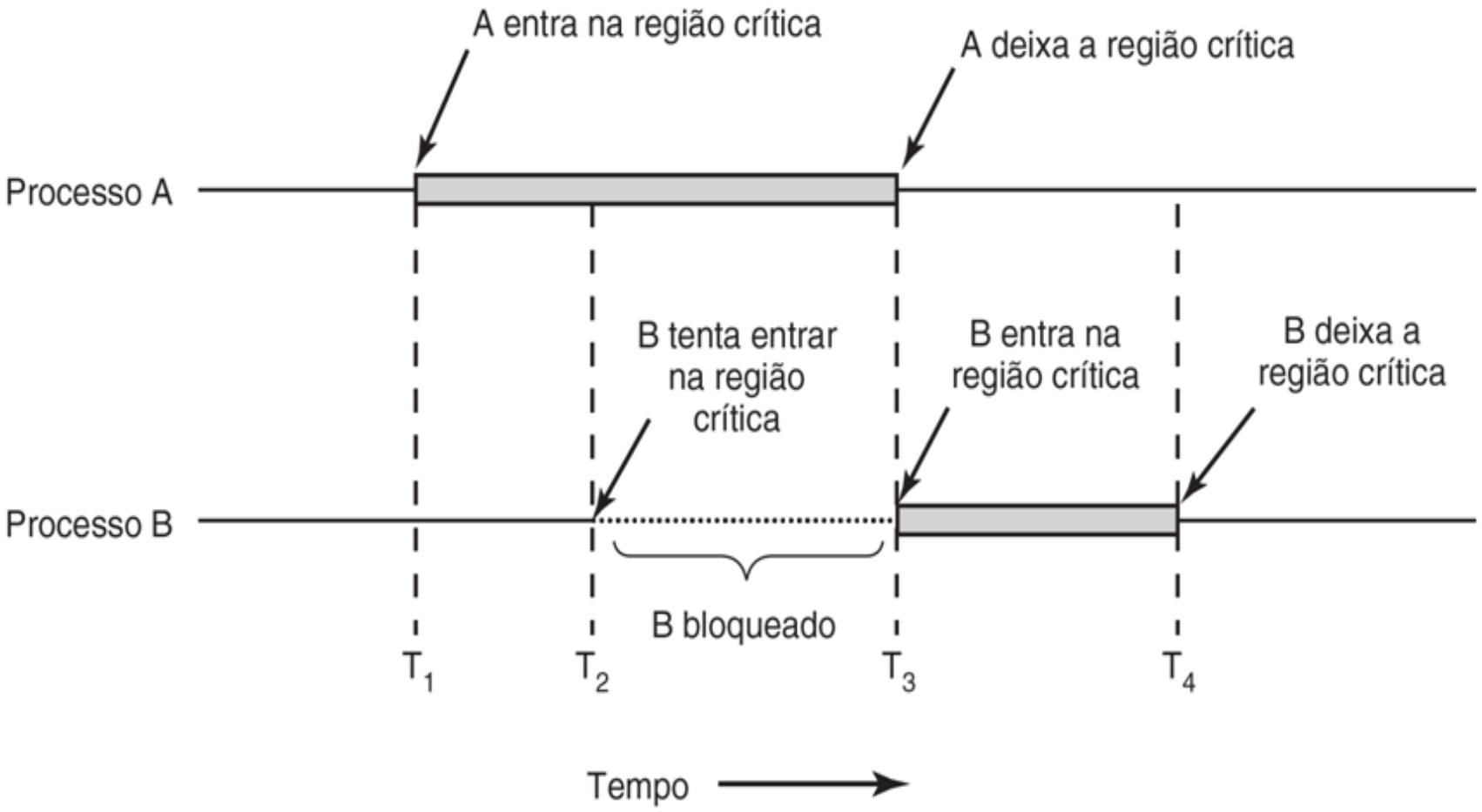
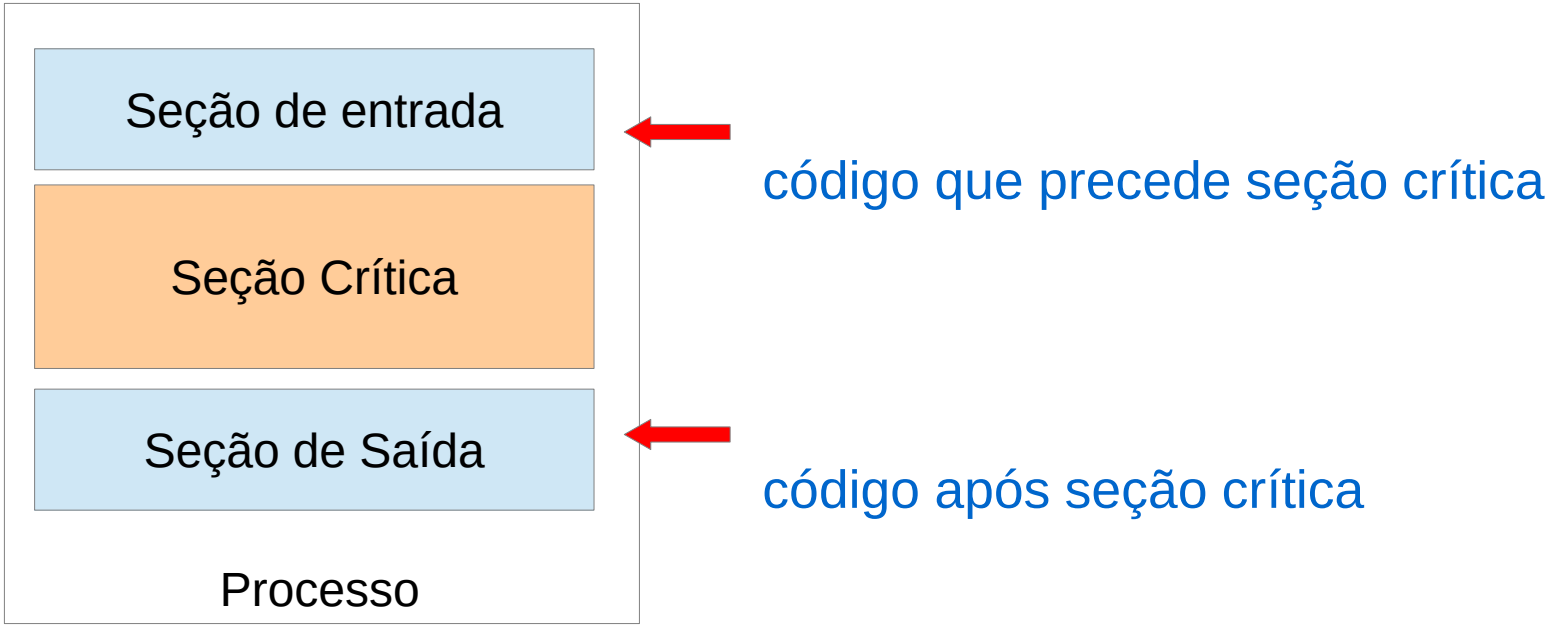


Figura 2.17 Exclusão mútua usando regiões críticas. [Tanenbaum]

Seção crítica

- Dois processos não podem executar em suas regiões críticas ao
- mesmo tempo
  - É necessário um protocolo de cooperação entre os processos
  - Cada processo precisa solicitar permissão para entrar em sua região crítica
  - Estrutura geral de um processo:



Solução para o problema da Seção Crítica

- Declaramos uma seção do código com sendo **crítica** e, em seguida, **controlamos o acesso** a essa seção, alguns requisitos precisam ser satisfeitos:

1. **Exclusão Mútua –** Se o processo P_i estiver executando em sua seção crítica, então nenhum outro processo poderá estar executando em sua respectiva seção crítica

2. **Progresso –** Se nenhum processo estiver executando em sua seção crítica e alguns processos estiverem solicitando entrada em sua respectiva seção crítica, então a seleção do processo que deverá entrar na seção crítica não poderá ser adiada indefinidamente (nenhum processo que estiver executando fora de sua região crítica pode bloquear outros processos (que queiram executar sua região crítica))

3. **Espera Limitada -** Deve existir um limite no número de vezes que outros processos podem entrar em suas seções críticas depois de um processo ter feito uma solicitação para entrar em sua seção crítica e antes que o pedido seja atendido (esse limite evita a *starvation*) (nenhum processo deve esperar

Hardware de Sincronização

➢Vários sistemas fornecem **suporte de hardware** para solucionar o problema da seção crítica

- Desativar interrupções enquanto uma variável compartilhada estiver sendo modificada
  - O código atualmente sendo executado roda sem preempção

- **Máquinas modernas** fornecem **instruções atômicas** especiais de hardware
  - **Instrução Atômica** não pode ser interrompida enquanto acessa algum recurso (como uma variável na memória)
  - Instrução *test-and-set* permite que a CPU leia um valor de memória, modifique o valor em seus registradores e atualize o valor na memória de forma atômica (sem interrupção)
  - Instrução *swap* permite a troca do conteúdo de duas variáveis diferentes de forma atômica
  - São instruções de hardware de baixo nível (*assembly*)

Suporte de hardware - Desabilitando interrupções

- Sistema monoprocessador
  - Cada processo desabilita todas as interrupções logo depois de entrar em sua região crítica e as reabilita imediatamente antes de sair dela
  - Com as interrupções desligadas, a CPU não será mais chaveada para outro processo
  - Desse modo o processo pode acessar a memória compartilhada sem temer a intervenção de outro processo
  - Não é prudente dar o poder aos processos do usuário de desligar interrupções→ se o processo não as reabilita (se entrar em *loop*) o funcionamento do sistema ficará comprometido

- Com o número crescente de chips *multicore*, a possibilidade de realizar exclusão mútua (mesmo dentro do núcleo) é cada vez menor
  - Desabilitar interrupções de uma CPU não impede que outras CPUs interfiram nas operações que a primeira CPU está executando
  - As interrupções são desabilitadas em apenas uma CPU

Algumas soluções para exclusão mútua

- Em geral, as soluções de hardware para exclusão mútua são complexas, veremos algumas soluções de software:
  - Variável trava (*lock*)
  - Chaveamento obrigatório
  - Solução de Peterson

  - Semáforos
  - Monitores

Variável de trava (lock)

- É utilizada uma variável compartilhada ***trava*** com valor inicial 0 (trava liberada)
- Antes de um processo entrar em sua região crítica, ele deve verificar antes se a trava está liberada (0=liberada; 1=ocupada)

```
// seção de entrada
while (trava == 1) ;
trava = 1;
```

```
Seção Crítica
```

```
//seção de saída
trava = 0;
```

Enquanto a trava for igual a 1, o processo ficará bloqueado. Se a trava for igual a 0, o processo a modifica para 1 e entra na seção crítica.

- Problema: na forma original, pode causar condições de corrida (mesma situação do arquivo de impressão, dois processos poderão ter acesso à seção crítica)

Chaveamento obrigatório

- Possibilita a exclusão mútua entre dois processos
  - Utiliza uma variável compartilhada ***turn*** (inicialmente 0) para controlar a vez de quem entra na região crítica
  - Um processo acessa a sua região crítica somente se for a sua vez
    - Processo 0 só consegue entrar na região crítica se ***turn*** for igual a 0 e da mesma forma, processo 1 só consegue entrar se ***turn*** igual a 1

```
// seção de entrada
while (turn != 0) ;

Seção Crítica

//seção de saída
turn = 1;
```

Processo 0

```
// seção de entrada
while (turn != 1) ;

Seção Crítica

//seção de saída
turn = 0;
```

Processo 1

Chaveamento obrigatório

- Evita o acesso concorrente aos dados compartilhados
- Mas um processo pode ser impedido de acessar sua regiao crítica mesmo que o outro processo não esteja em sua região crítica
  - Exemplo: P0 sai de sua região crítica e muda turn para 1, se desejar entrar novamente na região crítica tem que aguardar que P1 acesse e modifique turn para 0, porém, P1 pode não estar interessado em acessar a sua região crítica

- **Problema:** espera ociosa ou espera ocupada (*busy waiting*)- um processo fica testando continuamente a mudança do valor da variável, consumindo ciclos de CPU

- Uma variável de trava que usa espera ociosa é chamada de trava giratória (*spin lock*)

Solução de Peterson

- Possibilita a exclusão mútua entre dois processos
  - Além da variável compartilhada ***turn*** para controlar a vez de quem entra na região crítica, é utilizado também um array compartilhado ***interested*** que indica se um processo está interessado e pronto para executar sua região crítica

```
// seção de entrada
interested[i] = true;
turn = j;
while(interested[j] && turn==j) ;
```

```
Seção Crítica
```

```
//seção de saída
interested[i] = false;
```

Processo i

```
// seção de entrada
interested[j] = true;
turn = i;
while(interested[i] && turn==i) ;
```

```
Seção Crítica
```

```
//seção de saída
interested[j] = false;
```

Processo j

Solução de Peterson

- Antes de um processo i entrar na sua região crítica, este indica o seu interesse (interested[i]=true) e define turn como j (possibilitando que se o outro processo acesse sua região crítica, se precisar)
- Se ambos processos entrarem ao mesmo tempo, a variável turn decidirá qual processo terá permissão para acessar sua região crítica primeiro

- Possibilita exclusão mútua e não impede que outro processo entre na região crítica se o primeiro não estiver na sua região crítica (evita o problema da solução anterior)

Semáforos

➢Mecanismo para implementar exclusão mútua e sincronização de processos

➢Semáforo S – variável inteira que indica se um recurso comum já está sendo usado

- S pode ser acessada e alterada por apenas duas **operações indivisíveis (atômicas)***:

- acquire(): é chamado quando o processo deseja entrar na região crítica
- release(): é chamado quando o processo sai de sua região crítica

➢Um semáforo fica associado a um recurso compartilhado, indicando se o recurso está sendo usado

- Se o valor do semáforo é maior do que zero, então existe recurso compartilhado disponível
- Se o valor do semáforo é zero, então o recurso está sendo usado

(*Essas operações originalmente se chamavam P (do *holandês*, proberen, que significa testar) e V (*verhogen*, que

Exclusão mútua com semáforos

➢O semáforo pode ser utilizado para implementar exclusão mútua, ou seja, para proteger o acesso a uma região crítica

➢Neste caso, é utilizado um semáforo binário (o valor inteiro possui apenas dois valores possíveis: 0 ou 1), também conhecido como mutex (*mutual exclusion*)

- O semáforo é inicializado em 1 (indica que a seção crítica está disponível)
- acquire é executado quando um processo deseja entrar na sua região crítica
  - Testa se a seção crítica pode ser acessada (se semáforo = 1)
  - Decrementa o semáforo de 1 (impede que outros processos acessem a região crítica)
- release é executado quando o processo sai da sua região crítica
  - Incrementa o semáforo de 1 (libera a região crítica)

Exclusão mútua com semáforos

```
// inicial do semáforo
```

```
int value = 1;  // valor
```

```
// seção de entrada
s.acquire();
```

```
Seção Crítica
```

```
//seção de saída
s.release();
```

Processo

```
acquire() {
   while(value <= 0)
      ;
   value--;
}
```

Enquanto **value** for menor ou igual a 0, o processo ficará bloqueado (há outro processo acessando a região crítica.

```
release() {
   value++;
}
```

Semáforo

Exemplo de semáforos em Java

```
public class Worker implements Runnable {

   private Semaphore sem;
   private String name;
   // construtor
   public Worker(Semaphore sem, String name) {
      this.name = name;
      this.sem = sem;
   }

   public void run() {
      while (true) {
         sem.acquire();
         System.out.println("Worker " + name + " entering critical section");
         SleepUtilities.nap(3);
         System.out.println("Worker " + name + " leaving critical section");
         sem.release();
         SleepUtilities.nap(3);
      }
   }
}
```

Recebe a referência para o semáforo

Classe Worker [Silberschatz]

** Este código está disponível no Tidia em Repositório → Exemplos → Worker*

Exemplo de semáforos em Java

```
public class Semaphore {
   public Semaphore() {
      value = 0;
   }
   public Semaphore(int value) {
      this.value = value;
   }

   public synchronized void acquire() {
      while (value <= 0)
         ;
      value --;
   }

   public synchronized void release() {
      ++value;
   }
   private int value;
}
```

Classe Semaphore [Silberschatz]

Exemplo de semáforos em Java

```
public class SemaphoreFactory
{
   public static void main(String args[]) {
      Semaphore sem = new Semaphore(1);

      Thread[] bees = new Thread[5];

      for (int i = 0; i < 5; i++)
         bees[i] = new Thread(new Worker(sem, "Worker " +


      for (int i = 0; i < 5; i++)
         bees[i].start();
   }
}
```

Cria um semáforo com o valor inicial igual a 1

Cria 5 threads passando a referência para o semáforo

Classe SemaphoreFactory [Silberschatz]

Implementação de Semáforo

- Uma desvantagem do tipo de semáforo utilizado é que exige espera ocupada (*busy waiting*)
  - Enquanto um processo está na sua região crítica, qualquer outro processo que tentar acessar sua região crítica ficará em *loop* contínuo (desperdiça ciclos de CPU)
    - Em vez de espera ocupada, é possível fazer com que o processo se bloqueie (muda o estado para esperando) e entre numa fila de espera, liberando a CPU para escalonar outro processo para execução

```
acquire() {
   value--;
   if(value < 0) {
     acrescenta o processo a fila de espera
     block; // suspende o processo
   }
}
```

```
release() {
   value++;
   if(value = 0)
     remove processo P da fila de espera
     wakeup(P); // retoma a execução do processo bloqueado P
   }
}
```

Semáforo contador

- Semáforo contador – **o valor inteiro pode variar por um domínio irrestrito**
  - Utilizado para controlar o acesso a um recurso (com várias instâncias)
  - O semáforo é inicializado com o número de recursos disponíveis
  - acquire é executado quando um processo deseja usar um recurso
    - Decrementa o semáforo de 1
  - release é executado quando o processo libera o recurso
    - Incrementa o semáforo de 1
  - Se o valor do semáforo chegar a 0, todos os recursos estão sendo utilizados e outros processos que queiram acessar deverão esperar até que algum outro processo libere (fazendo com que o valor do semáforo fique maior que 0)
  - Exemplo: a partir do slide 31, problema do Produtor-Consumidor usando semáforos

Deadlock e *Starvation* (estagnação)

➢***Deadlock*** (Impasse) – dois ou mais **processos aguardam indefinidamente** por um evento que só poderá ser causado por um dos processos em espera

➢Exemplo: Considere S e Q dois semáforos inicializados com o valor 1

P_0	P_1
acquire(S);	acquire(Q);
acquire(Q);	acquire(S);
. . .	
. . .	
release(S);	release(Q);
release(Q);	release(S);

➢ (Inicialmente P_0 recebe um recurso (S) e P_1 recebe outro recurso (Q). Depois de alguns ciclos P_0 precisa do recurso Q e fica esperando P_1 liberá-lo. Ato contínuo, P_1 precisa do recurso S e fica esperando P_0 liberá-lo. Há uma *Espera Circular*)

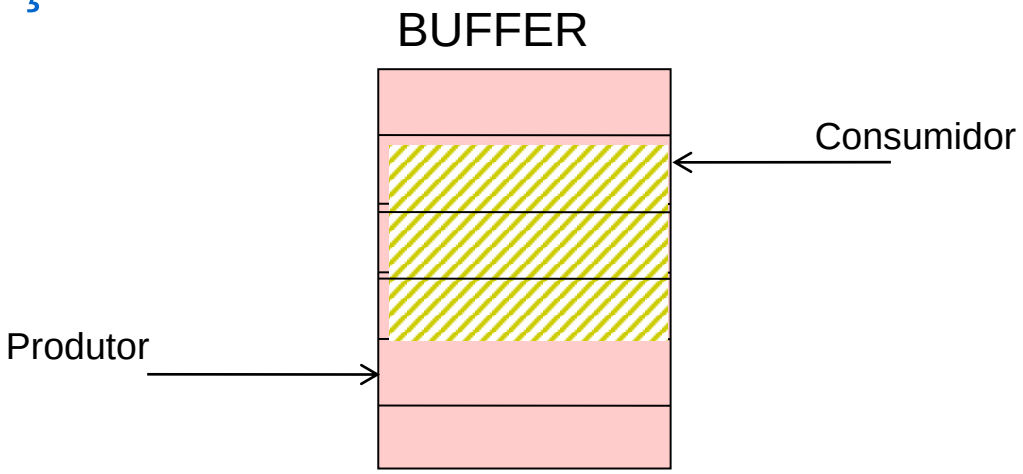
➢***Starvation*** – bloqueio indefinido. Situação em que um processo nunca consegue executar sua região crítica e acessar o recurso compartilhado

Problemas clássicos de sincronização

- O problema do ***Buffer*** **Limitado (Produtor-Consumidor)**
- O problema dos **Leitores-Escritores**
- O problema do **Jantar dos Filósofos**

O problema do buffer limitado

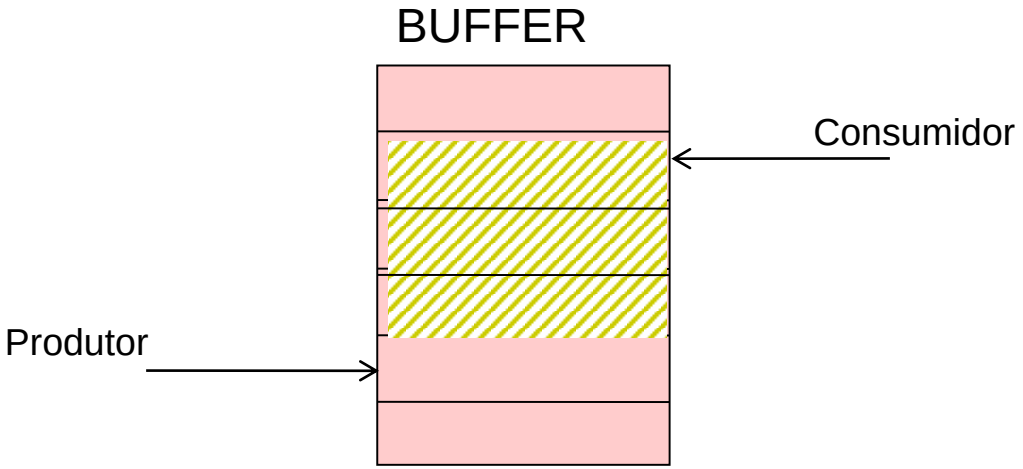
- Citamos anteriormente o problema do Produtor-Consumidor, utilizando um *buffer* compartilhado de tamanho fixo
  - Um **produtor coloca um item** no *buffer* chamando o método **insert()**
  - O **consumidor remove os itens** chamando o método **remove()**
  - Um produtor não poderia inserir em um *buffer* cheio e nem o consumidor poderia ler de um *buffer* vazio – problema de sincronismo
- Neste caso, semáforos podem ser utilizados para sincronização



O problema do buffer limitado

➢Para solucionar o problema do Produtor-Consumidor, são utilizados 3 semáforos:

- Semáforo mutex: é usado para exclusão mútua para garantir que somente um processo por vez leia ou escreva no *buffer* e outras variáveis
- Semáforos full e empty: utilizados para sincronização, garantem que o produtor pare de executar (aguarde) se o *buffer* estiver cheio e que o consumidor aguarde se o *buffer* estiver vazio



mutex: (0: buf

Na figura:
full: 3 (número de espaços ocupados no
empty: 3 (número de espaços vazios no

Classe Buffer

```
import java.util.*;
public class BoundedBuffer implements Buffer {
   private static final int BUFFER_SIZE = 2;
   private Semaphore mutex; // exclusão mutua para o buffer
   private Semaphore empty; // número de espaços vazios
   private Semaphore full;  // número de espaços preenchidos
   private int count;
   private int in, out;
   private Object[] buffer;

   public BoundedBuffer()    {
      // buffer is initially empty
      count = 0;
      in = 0;
      out = 0;
      buffer = new Object[BUFFER_SIZE];
      mutex = new Semaphore(1);
      empty = new Semaphore(BUFFER_SIZE); // todas as posições vazias
      full = new Semaphore(0);            // nenhuma posição preenchida
   }
   // producer calls this method
   public void insert(Object item) {

   }
   // consumer calls this method
   public Object remove() {

   }
}
```

Semáforos para exclusão mútua e para indicar a quantidade de recursos disponíveis

Método insert()

```
public void insert(Object item) {
   empty.acquire();
   mutex.acquire();

   // add an item to the buffer
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;

   if (count == BUFFER_SIZE)
      System.out.println("Producer Entered " + item + " Buffer FULL");
   else
      System.out.println("Producer Entered " + item + " Buffer Size = "
                                                    + count);
   mutex.release();
   full.release();
}
```

Essas duas chamadas
Verificam: se há espaço
no buffer (empty) e se pode
acessar a região crítica (mutex)

Método insert [Silberschatz]

** Este código está disponível no Tidia em Repositório → Exemplos → BoundedBuffer*

Método remove()

```
public Object remove() {
   full.acquire();
   mutex.acquire();

   // remove an item from the buffer
   --count;
   Object item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   if (count == 0)
      System.out.println("Consumer Consumed " + item + " Buffer EMPTY");
   else
      System.out.println("Consumer Consumed " + item + " Buffer Size = "
                                                          + count);
   mutex.release();
   empty.release();

   return item;
}
```

Se o buffer não está cheio e se pode acessar a região crítica

Método remove [Silberschatz]

Classe Produtor

```
import java.util.*;

public class Producer implements Runnable {
   public Producer(Buffer b) {
      buffer = b;
   }

   public void run() {
      Date message;

      while (true) {
         System.out.println("Producer napping");
            SleepUtilities.nap();

         // produce an item & enter it into the buffer
         message = new Date();
         System.out.println("Producer produced " + message);

         buffer.insert(message);
      }
   }

   private  Buffer buffer;
}
```

Classe Consumidor

```
import java.util.*;

public class Consumer implements Runnable {
   public Consumer(Buffer b) {
      buffer = b;
   }

   public void run() {
      Date message;

      while (true) {
         System.out.println("Consumer napping");
            SleepUtilities.nap();

         // consume an item from the buffer
         System.out.println("Consumer wants to consume.");

         message = (Date)buffer.remove();
      }
   }

   private  Buffer buffer;
}
```

Classe Consumidor [Silberschatz]

Classe Semaforo

```
public class Semaphore {
   public Semaphore() {
      value = 0;
   }
   public Semaphore(int value) {
      this.value = value;
   }

   public synchronized void acquire() {
      while (value <= 0)
         ;
      value --;
   }

   public synchronized void release() {
      ++value;
   }
   private int value;
}
```

Classe Semaphore [Silberschatz]

Problema dos Leitores-Escritores

Um **banco de dados** deve ser **compartilhado** entre várias *threads* concorrentes

➢**Leitores** - Algumas dessas *threads* querem apenas **ler o banco de dados**

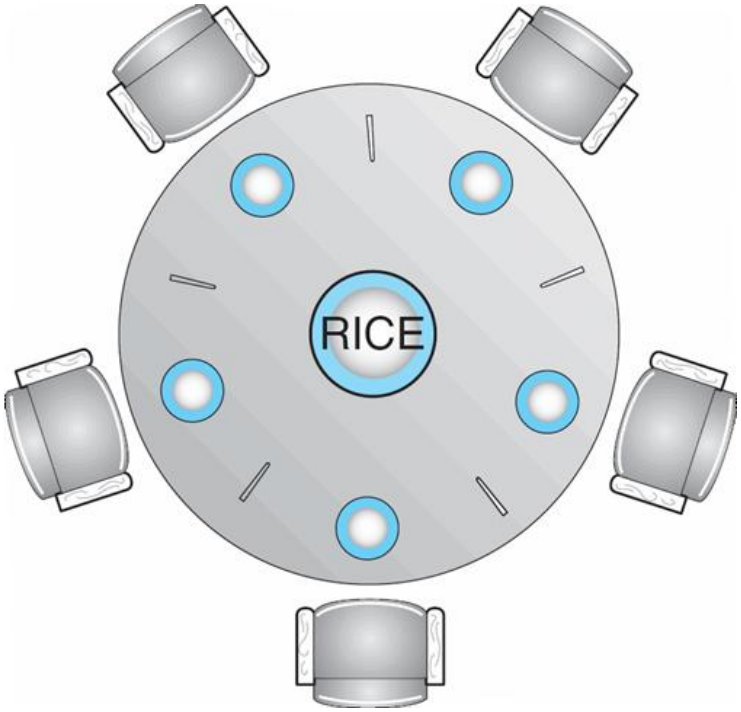
➢**Escritores** - Outras desejam atualizá-lo (**ler e gravar dados**)

➢Não há efeitos adversos se dois leitores acessarem os dados compartilhados ao mesmo tempo

➢No entanto, se um escritor e alguma outra *thread* (leitor ou escritor) acessar o banco de dados ao mesmo tempo, poderá haver confusão

➢**Solução Possível** – Os escritores devem ter acesso exclusivo ao banco de dados

Problema do Jantar dos Filósofos



Jantar dos filósofos [Silberschatz]

Cinco filósofos passam suas vidas meditando e comendo em torno de uma mesa com cinco pratos de arroz e cinco pauzinhos (*hashis*).

Se os cinco filósofos tentarem comer ao mesmo tempo, cada um ficará com apenas um *hashi*.

Quando um filósofo com fome tiver dois *hashis* ao mesmo tempo, ele poderá comer sem soltá-los.

Ao terminar de comer, o filósofo solta os *hashis* e volta a meditar.

➢O problema dos filósofos na mesa de jantar é um problema clássico de sincronismo (é um exemplo de uma grande classe de problemas de controle de concorrência)

➢É a representação simples de como alocar recursos entre vários processos de uma maneira sem *deadlock* e sem *starvation*

Problema do Jantar dos Filósofos (cont.)

Dados compartilhados (cada *hashi* é um semáforo)

```
Semaphore chopStick[]  = new Semaphore[5];
```

Filósofo *i*:

```
while (true) {
   // pega o pauzinho da esquerda
   chopStick[i].acquire();
   // pega o pauzinho da direita
   chopStick[(i + 1) % 5].acquire();
   eating();
   // devolve o pauzinho da esquerda
   chopStick[i].release();
   // devolve o pauzinho da direita
   chopStick[(i + 1) % 5].release();
   thinking();
}
```

Estrutura do filósofo i [Silberschatz]

Essa solução simples pode criar um *deadlock* !

Problema do Jantar dos Filósofos (cont.)

- Algumas soluções para o problema de deadlock:
  - Permitir no máximo 4 filósofos se sentem simultaneamente à mesa
  - Só permitir que um filósofo pegue os *hashis* se os dois estiverem disponíveis (isso deve ser feito em uma região crítica)
  - Um filósofo ímpar apanha primeiro o *hashi* da esquerda e depois o da direita, enquanto que o filósofo par apanha o *hashi* da direita e depois o da esquerda

- Porém, ainda há a necessidade de tratar o problema de um dos filósofos morrer de fome (*starvation*)
  - O exemplo de solução do slide 36 (com monitores) evita esse problema

Monitores

- Um monitor é uma estrutura de **sincronização de alto nível** proposta para facilitar o desenvolvimento de programas concorrentes

- Um monitor é uma coleção de rotinas, variáveis e estruturas de dados agrupados em um módulo

- Apenas um processo pode estar ativo em um monitor em determinado momento e a implementação interna de um monitor não pode ser acessada diretamente pelos processos

- A construção do monitor assegura que somente um processo estará ativo dentro dele por vez, assegurando a exclusão mútua (libera o programador dessa tarefa, evitando-se erros)
-

Monitores

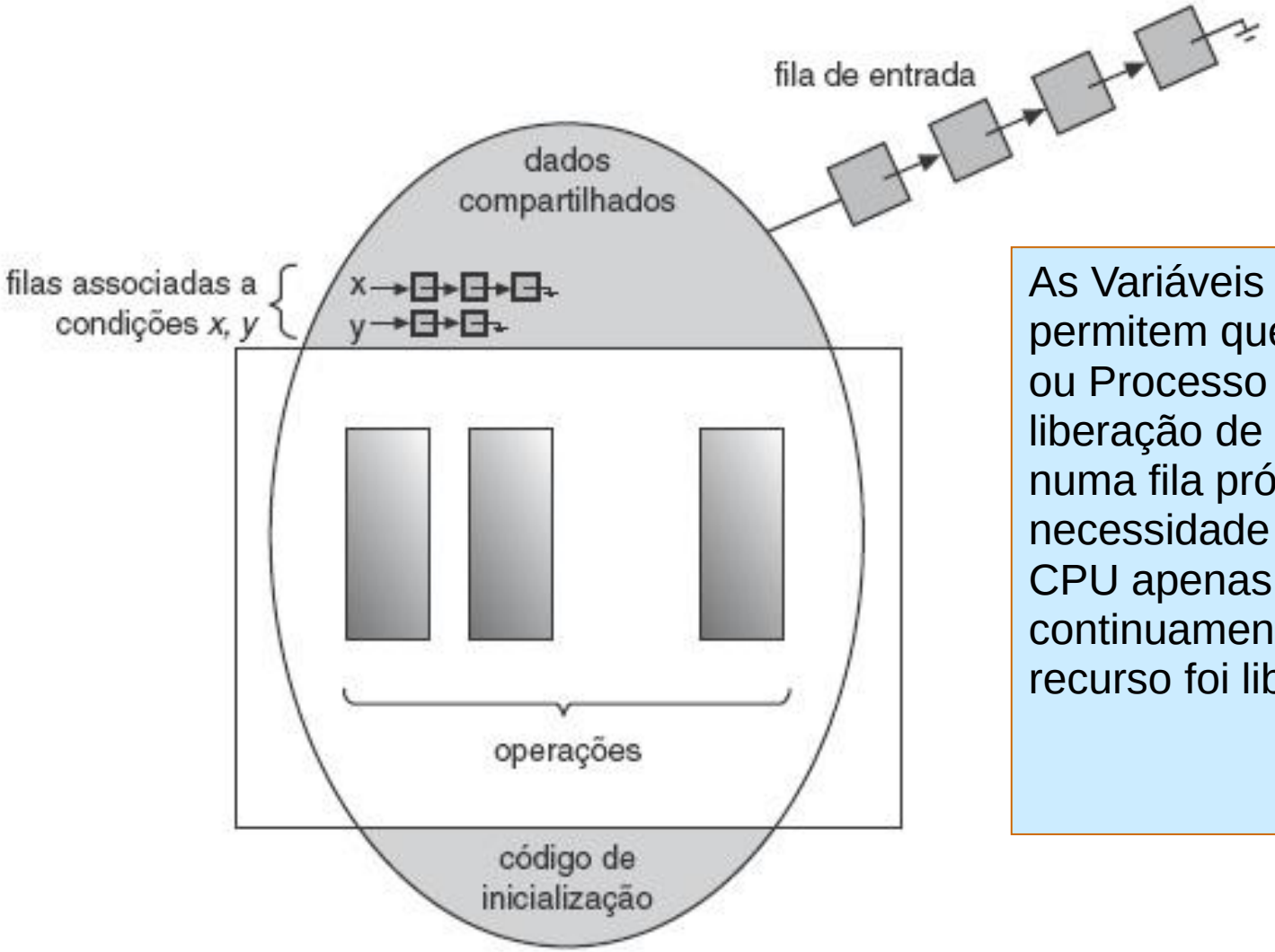
```
monitor nome-do-monitor
{
    // declaração de variáveis compartilhadas

    initialization code (…)  {
            . . .
     }
     procedure P1 (…) {
            . . .
     }
     procedure P2 (…) {
            . . .
     }
     procedure Pn (…) {
            . . .
     }
}
```

Exemplo de um monitor

Variáveis de condição

- Variáveis do tipo **condition** evitam que os monitores entrem em estado de **espera ocupada**, com a ajuda de operações especiais que podem ser chamadas: **wait** e **signal**
- Um programador que precise escrever seu próprio esquema de sincronização personalizado pode definir uma ou mais variáveis do tipo condition, não sendo necessário codificar esta sincronização explicitamente
- condition x, y;
- Um *thread* que invoca x.wait é suspenso (a *thread* ou processo é retirado da **fila de prontos**) até que outra *thread* invoque x.signal
- A operação x.signal retoma exatamente uma *thread*, isto é, libera um recurso para uma *thread*. Se nenhuma *thread* estiver suspensa, a operação signal não tem efeito

Monitor com variáveis de condição



As Variáveis de Condição permitem que uma *Thread* ou Processo aguarde a liberação de um recurso numa fila própria, sem necessidade de ocupar a CPU apenas para checar continuamente se um recurso foi liberado

Visão esquemática de um monitor [Silberschatz]

Jantar dos filósofos com monitores

```
monitor DiningPhilosophers {
    enum { THINKING; HUNGRY, EATING) state[5] ;
    condition self[5];

    void pickup(int i) {  // pega hashis
   state[i] = HUNGRY;
   test(i);
   if (state[i] != EATING) self[i].wait; // se não conseguiu pegar os hashis
    }                                          // suspende o processo

    void putdown(int i) {  // libera hashis
   state[i] = THINKING;
        // testa vizinhos da esquerda e direita
   test((i + 4) % 5);
   test((i + 1) % 5);
     }
     void test (int i) {
    if( (state[(i + 4) % 5] != EATING) &&  // se os vizinhos da direita e
        (state[i] == HUNGRY) &&            // esquerda não estiverem comendo
        (state[(i + 1) % 5] != EATING) ) { // o processo para estado EATING
            state[i] = EATING ;
            self[i].signal() ;     // retoma algum processo suspenso
    }
}

      initialization_code() {
    for(int i = 0; i < 5; i++)
       state[i] = THINKING;
}
}
```


Esqueleto de uma solução com monitores para o problema dos Filósofos na mesa de jantar [Silberschatz2]

Sincronismo em Java

- Quando uma aplicação garante que os dados permanecem coerentes mesmo quando estão sendo acessados concorrentemente por várias *threads* a aplicação é considerada segura para *thread* (*thread safe*)

- Alguns problemas que precisam ser contornados:
  - Espera ocupada
  - Condição de corrida
  - Deadlock

Sincronismo em Java – espera ocupada

- Uma forma para contornar a espera ocupada é fazer com que o processo se bloqueie caso não possa entrar em sua região crítica

- Uma forma de fazer esse bloqueio é fazer com que a *thread* chame o método Thread.yield()
  - Esse método faz com que a *thread* permaneça no estado executável mas permite que a JVM selecione outra *thread* para execução
  - O método yield() possibilita aproveitar melhor a CPU do que a espera ocupada

Sincronismo em Java – condição de corrida

- Para tratar o problema de condição de corrida no acesso concorrente aos dados compartilhados, Java utiliza um mecanismo de *lock*
  - Cada objeto Java possui associado a ele um único *lock*, que pode estar em posse de uma única *thread*
  - Normalmente, o *lock* é ignorado quando seus métodos são invocados
  - Mas se o método é declarado como synchronized, para acessar o método é preciso obter o *lock*

```
public synchronized void insert(Object item) {
   while(count == BUFFER_SIZE)
      Thread.yield();

   // add an item to the buffer
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
}
```

Sincronismo em Java – condição de corrida

- Se o *lock* já estiver em posse de outra *thread*, a *thread* que chama o método é bloqueada e é colocada em uma fila (conjunto de entrada – *entry set*) do lock do objeto
- Se o *lock* estiver disponível, a *thread* que chama o método se torna a proprietária do *lock* do objeto e pode acessá-lo
- Se o produtor chamar o insert() e obter o lock, o consumidor ficará bloqueado ao tentar invocar receive() e deverá aguardar a liberação do *lock*; dessa forma, o produtor poderá alterar os dados compartilhados de forma segura

```
public synchronized Object remove() {
   Object item;
   while(count == BUFFER_SIZE)
      Thread.yield();

   // remove an item from the buffer
   --count;
   Object item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   return item;
}
```

Sincronismo em Java – deadlock

- O synchronized evita situações de condição de corrida, garantindo que apenas uma *thread* acessará um objeto
- Porém, o synchronized pode causar um outro problema: *deadlock*
  - Considere o *buffer* cheio e produtor querendo inserir um novo item, e supondo o *lock* disponível, o produtor ficará bloqueado
  - Quando o consumidor desperta e tenta invocar remove(), também ficará bloqueado esperando que o produtor libere o *lock*

- Para solucionar esse problema, Java possibilita que uma *thread* ao ser bloqueada como no caso do produtor, libere o *lock* e fique em uma fila de espera (dessa forma, consumidor pode obter o lock e invocar remove())
  - Utiliza os métodos wait() e notify()

Sincronismo em Java – wait e notify

```
public synchronized void insert(Object item) {
   while(count == BUFFER_SIZE)
      try {
                wait();
      } catch(InterruptedException e) {}

   count++;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   notify();
}
```

```
public synchronized Object remove() {
   Object item;
   while(count == 0)
      try {
                wait();
      } catch(InterruptedException e) {}

   count--;
   Object item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   notify();
   return item;
}
```

Sincronismo em Java – wait e notify

- Quando uma *thread* faz uma chamada ao método wait():
  - A *thread* libera o *lock* do objeto
  - O estado da *thread* é definido como bloqueado
  - A thread é colocada no conjunto de espera do objeto

- Quando uma *thread* faz uma chamada ao método notify():
  - Obtém uma *thread* T na lista de threads no conjunto de espera
  - Move T para o conjunto de entrada
  - Modifica o estado de T para executável

Referências

- [Silberschatz] SILBERCHATZ, A., GALVIN, P. B. e GAGNE, G. **Sistemas Operacionais com Java**. 7ª ed., Rio de Janeiro: Elsevier, 2008.
- [Tanenbaum] TANENBAUM, A. **Sistemas Operacionais Modernos**. 3ª ed. São Paulo: Prentice Hall, 2009.
- [MACHADO] MACHADO, F. B. e MAIA, L. P. **Arquitetura de Sistemas Operacionais**. 4ª ed., Rio de Janeiro: LTC, 2007.
- [Silberschatz2] SILBERCHATZ, A., GALVIN, P. B. e GAGNE, G. **Fundamentos de Sistemas Operacionais**. 8ª ed., Rio de Janeiro: LTC, 2012.