Conceitos básicos

- Acesso concorrente a dados compartilhados podem resultar em inconsistência de dados.
- Manutenção de consistência de dados requer mecanismos para garantir a execução ordenada de processos cooperantes.

- Problema do produtor-consumidor, visto anteriormente, permite n-1 items no buffer ao mesmo tempo. Uma solução onde todos os N buffer são usados não é simples.

  - Considere-se o problema do produtor-consumidor modificado, adicionando-se uma variável chamada counter, inicializada em 0 e incrementada cada vez que um item é incluído no buffer.

Bounded-Buffer

- Dados compartilhados

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer

- Processo produtor

```
item                                        nextProduced;

while (1) {
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Bounded-Buffer

- Processo consumidor

```
item                                                    nextConsumed;

while (1) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

Bounded Buffer

- As instruções

  counter++;
  counter--;

  precisam ser executadas *atomicamente*.

- Uma operação é dita atômica se ela é indivisível em relação à execução, ou seja, ela é completamente executada sem interrupção.

Bounded Buffer

- A instrução "**counter++**" pode ser implementada em linguagem Assembly por:

```
mov reg1, counter
add reg1, 1
mov counter, reg1
```

- A instrução "**counter--**" pode ser implementada em linguagem Assembly por:

```
mov reg1, counter
sub reg1, 1
mov counter, reg1
```

Bounded Buffer

- Se ambos processos produtor e consumidor tentam alterar o buffer concorrentemente, pode ocorrer intercalação das instruções Assembly dos processos.

- Intercalação depende de como os processos produtor e consumidor são escalonados.

Bounded Buffer

- Assuma que **counter** é inicialmente 5. Uma intercalação de instruções poderia ser:

produtor:	**mov reg1, counter** (*reg1 = 5*)
produtor :	**add reg1,1** (*reg1 = 6*)
consumidor:	**mov reg2, counter** (*reg2 = 5*)
consumidor :	**sub reg1,1** (*reg2 = 4*)
produtor :	**mov counter, reg1** (*counter = 6*)
consumidor :	**mov counter, reg2** (*counter = 4*)

- O valor de **counter** pode ser ou 4 ou 6, quando o valor correto deveria ser 5.

Condição de Disputa(Race Condition)

- **Race condition**: situação onde vários processos acessam e manipulam dados comportilhados concorrentemente. O valor final dos dados compartilhados depende do processo que termina por último.

- Para prevenir condições de disputa, processos precisam ser sincronizados.

Problema da seção crítica

- *n* processos competindo para usar dados compartilhados
- Cada processo tem um segmento de código, chamado seção crítica, onde dados compartilhados são acessados.
- **Problema**: garantir que, quando um processo está executando a seção crítica, nenhum outro processo pode ser pertmitido executar sua seção crítica.

Solução para o problema da seção crítica

1. **Exclusão mútua**. Se um processo P_i está executando sua seção crítica, nenhum outro processo pode executar sua seção crítica.
2. **Progresso**. Se senhum processo está executando sua seção crítica e existem alguns processos que queiram executar suas seções críticas, então a seleção dos processos que irão entrar na seção crítica não pode ser postecipada indefinidamente.
3. **Espera limitada**. Uma limitação precisa existir quanto ao número de vezes que outros processos são permitidos entrar em suas seções críticas depois que um processo requisitou entrada na sua região crítica e antes que a requisição seja garantida.
   - Assumir que cada processo executa a uma velocidade diferente de zero .
   - Nenhuma suposição é feita a respeito da velocidade relativa dos n processos.

Tentativas iniciais para resolver o problema

- Somente 2 processos, P_0 and P_1
- Estrutura geral do processo P_i

do {

entrada da seção

seção crítica

saída da seção

resto do programa

} **while (1)**;

- Processos podem compartilhar algumas variáveis comuns para sincronizar sua ações.

Algoritmo 1

- Variáveis compartilhadas:
  - **int turn**;
  - inicialmente **turn = 0**
  - **Turn=i** $\Rightarrow$ **P_i** pode entrar na sua seção crítica
- Process **P_i**

```
do {
        while (turn != i) ;
                seção crítica
        turn = j;
                restante do programa
} while (1);
```

- Satisfaz exclusão mútua, mas não satisfaz progresso.

Algoritmo 2

- Variáveis compatilhadas
  - **boolean flag[2]**;
  - inicialmente **flag [0] = flag [1] = false.**
  - **flag [i] = true** $\Rightarrow$ P_i pronto para entrar em sua seção crítica
- Process P_i

```
do {
        flag[i]:=true;
        while   (flag[j])   ;
        seção crítica
        flag [i] = false;
                restante do programa
} while (1);
```

- Satisfaz exclusão mútua, mas não satisfaz progresso.

Algoritmo 3

- Combinação das variáveis compartilhadas dos algoritmos 1 e 2.
- Processo P_i

```
do {
        flag[i]:=true;
        turn=j;
        while (flag [j] and turn = j) ;
                seção crítica
        flag [i] = false;
                restante do programa
} while (1);
```

- Satisfaz todos os três requisitos; resolve o problema da região crítica para dois processos.

Algoritmo de Bakery

Seção crítica para n processos

- Antes de entrar na sua seção crítica, o processo recebe um número. Quem possuir o menor número, entrada na seção crítica.
- Se processos P_i e P_j recebem o mesmo número, se $i < j$, então P_i é atendido primeiro; senão, P_j é atendido primeiro.
- Um esquema de numeração sempre garante que sempre são gerados números em ordem crescente: 1,2,3,3,3,3,4,5...

Algoritmo de Bakery

- Notação: $<\equiv$ Ordem lexicográfica
  - $(a,b) <\equiv (c,d)$ se $a < c$ ou se $a = c$ e $b < d$
  - max $(a_0,\ldots, a_{n-1})$ é um número, k, tal que $k \geq a_i$ para $i = 0, \ldots, n - 1$
- Dados compartilhados

  boolean choosing[n];

  int number[n];

  Estruturas de dados são inicializadas em **false** e **0** respectivamente

Bakery Algorithm

```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
                    while (choosing[j]) ;
                    while ((number[j] != 0) && (number[ j ] < number[
    i ])) ;
    }
        seção crítica
    number[i] = 0;
        restante do programa
} while (1);
```

Hardware para sincronização

- Testa e modifica uma palavra atomicamente

.

```
boolean TestAndSet(boolean &target) {
        boolean rv = target;
        tqrget = true;

        return rv;
}
```

Exlusão mútua com Test-and-Set

- Dados compartilhados:

  boolean lock = false;

- Processo P_i

```
do {
        while (TestAndSet(lock)) ;
                seção crítica
        lock = false;
        restante do programa
}
```

Hardware para sincronização

- Troca atômica de duas variáveis.

```
void Swap(boolean &a, boolean &b) {
        boolean temp = a;
        a = b;
        b = temp;
}
```

Exlusão mútua com Swap

- Dado compartilhado (inicializado com **false**):

```
boolean lock;
```

- Processo P_i

```
repeat {
        key = true; //key – variável local
        repeat
                Swap(lock,key);
        until key=false
        seção crítica
        lock = false;
        restante do programa
}until false;
```

Semáforos

- Ferramenta de sincronização que não requer esperar realizando processamento (busy waiting).

- Semáforo *S* – variável inteira
- Só podem ser acessados via duas operações atômica:

wait (*S*):

```
while S≤ 0 ;
S--;
```

signal (*S*):

```
S++;
```

Seção crítica de *n* processos

- Dados compartilhados:

  semaphore mutex; *//*initially *mutex* = 1

- Process *Pi:*

```
do{
        wait(mutex);
        seção crítica
        signal(mutex);
        restante do programa
} while (1);
```

Implementação de semáforos

- Semáforo pode ser um registro

```
typedef struct {
  int value; // Valor do semáfor
      struct process *L; // Fila do semáforo
}semaphore;
```

- Adicionalmente, considere-se duas operações básicas:
  - **block** : suspende o processo que invocou o semáforo.
  - **wakeup(*P*)** reativa a execução de um processo bloqueado P.

Implementação

- Agora, as operações wait e signal do semáforto podem ser definidas:

wait(*S*):

S.value--;
if (S.value < 0) {
Inclui o processo em **S.L;**
block;
}

signal(*S*):

S.value++;
if (S.value <= 0) {
Remover um processo **P** de **S.L;**
wakeup(P);
}

Semáforo como uma ferramenta geral de sincronização

- Executa *B* em P_j somente após *A* ter sido executado em P_i
- Usa semáforo *flag* inicializado em 0
- Código:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

Deadlock e Starvation

- **Deadlock** – dois ou mais processos estão esperando indefinidamente por um evento que pode ser causado por algum dos processos em espera.
- Sejam *S* e *Q* be dois semáforos inicializados em 1:

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- **Starvation** – bloqueio indefinido. Um processo nunca foi removido da fila associada ao semáforo sob o qual o processo foi suspenso.

Dois tipos de semáforos

- Semáforo de contagem – valor inteiro pode variar sobre um domínio irrestrito.
- Semáforo binário – valor inteiro pode variar somente entre 0 e 1; pode ser mais simples de implementar.
- Podemos inplementar um semáforo de contagem S como um semáforo binário.

Implementando *S* como um semáforo binário

- Estruturas de dados:

  binary-semaphore S1, S2;

  int C:

- Initialization:

  S1 = 1

  S2 = 0

  C = valor inicial do semáforo S

Implementando *S*

- Operação *wait*

```
wait(S1);
C--;
if (C < 0) {
                    signal(S1);
                    wait(S2);
}
signal(S1);
```

- Operação *signal*

```
wait(S1);
C ++;
if (C <= 0)
        signal(S2);
else
        signal(S1);
```

Problemas clássicos de sincronização

- Problema do buffer-limitado (Bounded-Buffer)
- Problema dos Leitores e Escritores(Readers and Writers)
- Problema dos filósofos(Dining-Philosophers)

Problema Bounded-Buffer

- Dados compartilhados

semaphore full, empty, mutex;

Inicialmente:

full = 0, empty = n, mutex = 1

Processo Produtor

```
do {
                    ...
        produz um item em nextp
                    ...
        wait(empty);
        wait(mutex);
                    ...
        insere nextp no buffer
                    ...
        signal(mutex);
        signal(full);
} while (1);
```

Processo consumidor

```
do {
        wait(full)
        wait(mutex);
                ...
        Remove um item do buffer para nextc
                ...
        signal(mutex);
        signal(empty);
                ...
        consome o item de  nextc
                ...
} while (1);
```

Problema dos Leitores e Escritores

- Dados compartilhados:

  semaphore mutex, wrt;

  Inicialmente:

  mutex = 1, wrt = 1, readcount = 0

Processo Produtor

wait(wrt);

...

Escrita é realizada

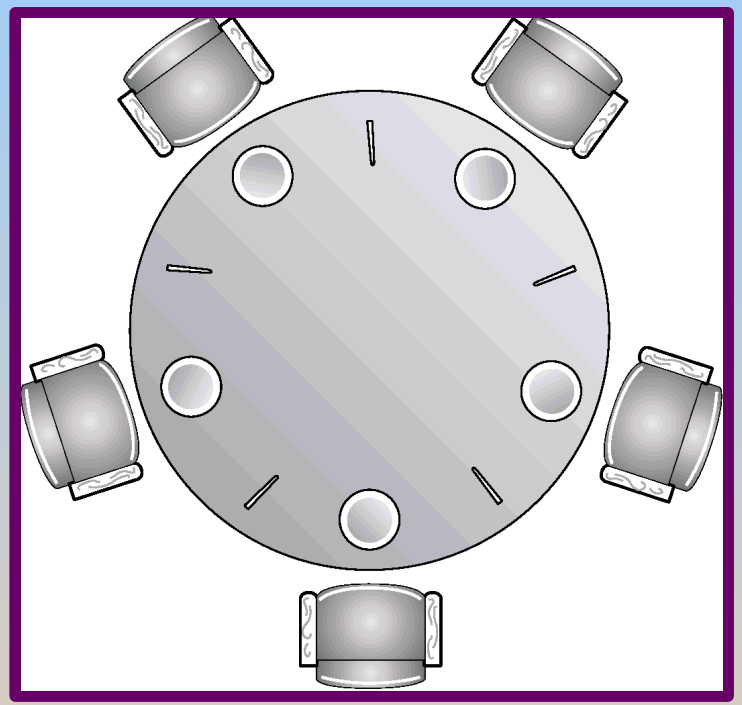
...

signal(wrt);

Processo consumidor

```
wait(mutex);
readcount++;
if (readcount == 1)
                    wait(rt);
signal(mutex);
                    …
        Leitura é realizada
                    …
wait(mutex);
readcount--;
if (readcount == 0)
        signal(wrt);
signal(mutex):
```

Problema dos Filósofos



- Dados compartilhados

semaphore garfos[5];

Inicialmente todos os valores são 1.

Problema dos Filósofos

- Filósofo *i*:

```
do {
        wait(garfos[i])
        wait(garfos[(i+1) % 5])
            ...
            comer
            ...
        signal(garfos[i]);
        signal(garfos[(i+1) % 5]);
            ...
            pensar
            ...
} while (1);
```

Regiões críticas

- Construção de sincronização de alto nível
- Uma variável v de tipo T é declarada como:

  v: shared T
- Variável ***v*** acessada somente dentro de bloco do tipo

  region v when B do S

  onde ***B*** é uma expressão booleana.

- Enquanto o bloco **S** está sendo executado, nenhum outro peocesso pode acessar a variável v.

Monitores

- Construção de alto nível que permite um compartilhamento seguro de tipos abstratos de dados entre processos concorrentes.

```
monitor nome
{
        declarações de variáveis compartilhadas
        procedure P1 (...) {
                . . .
        }
        procedure P2 (...) {
                . . .
        }
        procedure Pn (...) {
                . . .
        }
        {
                código de inicialização
        }
}
```

Monitores

- Para permitir que um processo espere dentro do monitor, uma variável do tipo condition precisa ser definida como:

  condition x, y;

- Variáveis do tipo condition podem somente ser usadas com as operações **wait** and **signal**.

  - A operação

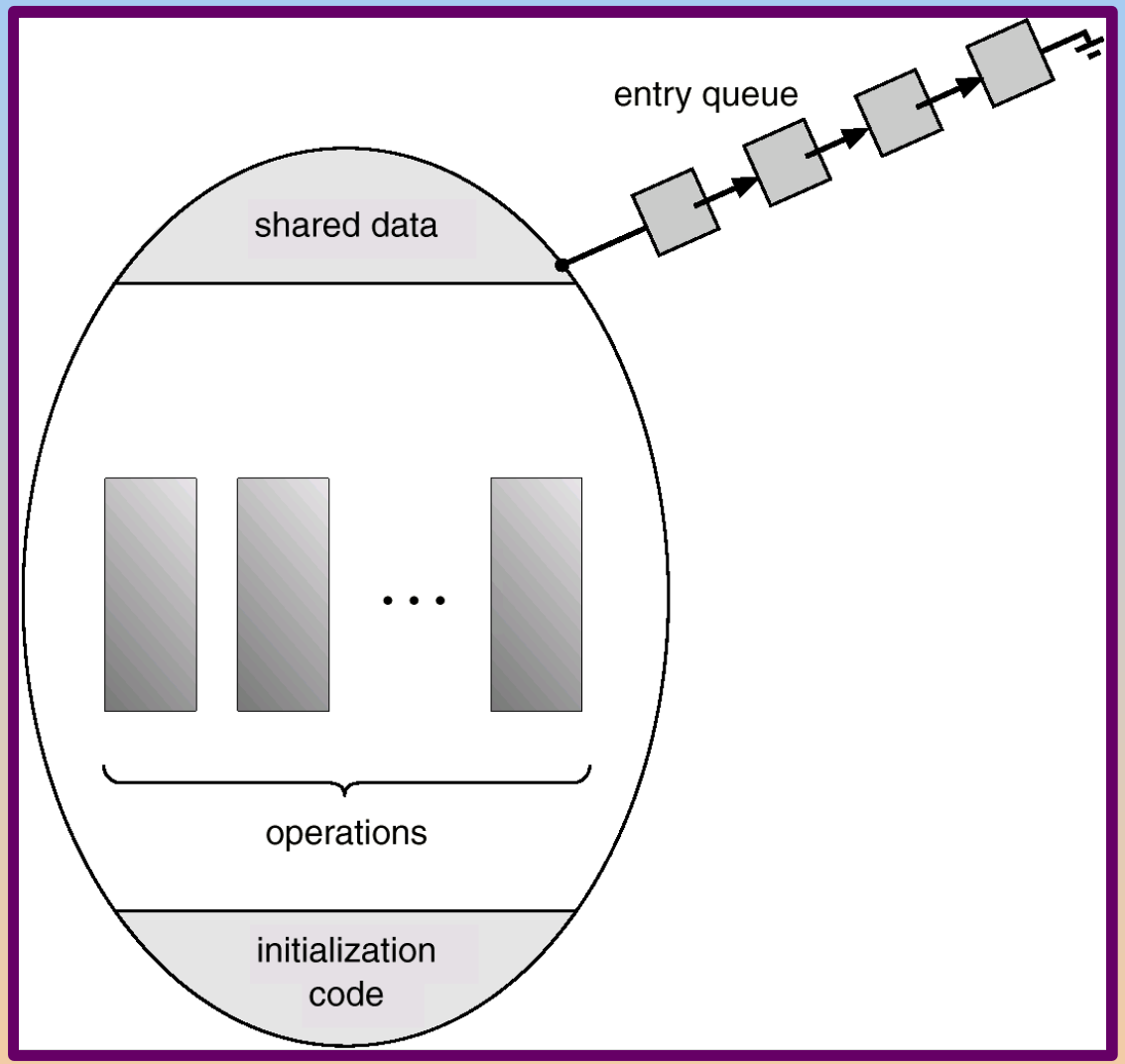
    x.wait();

    significa que o processo que invoca esta operação é suspenso até que outro processo invoque

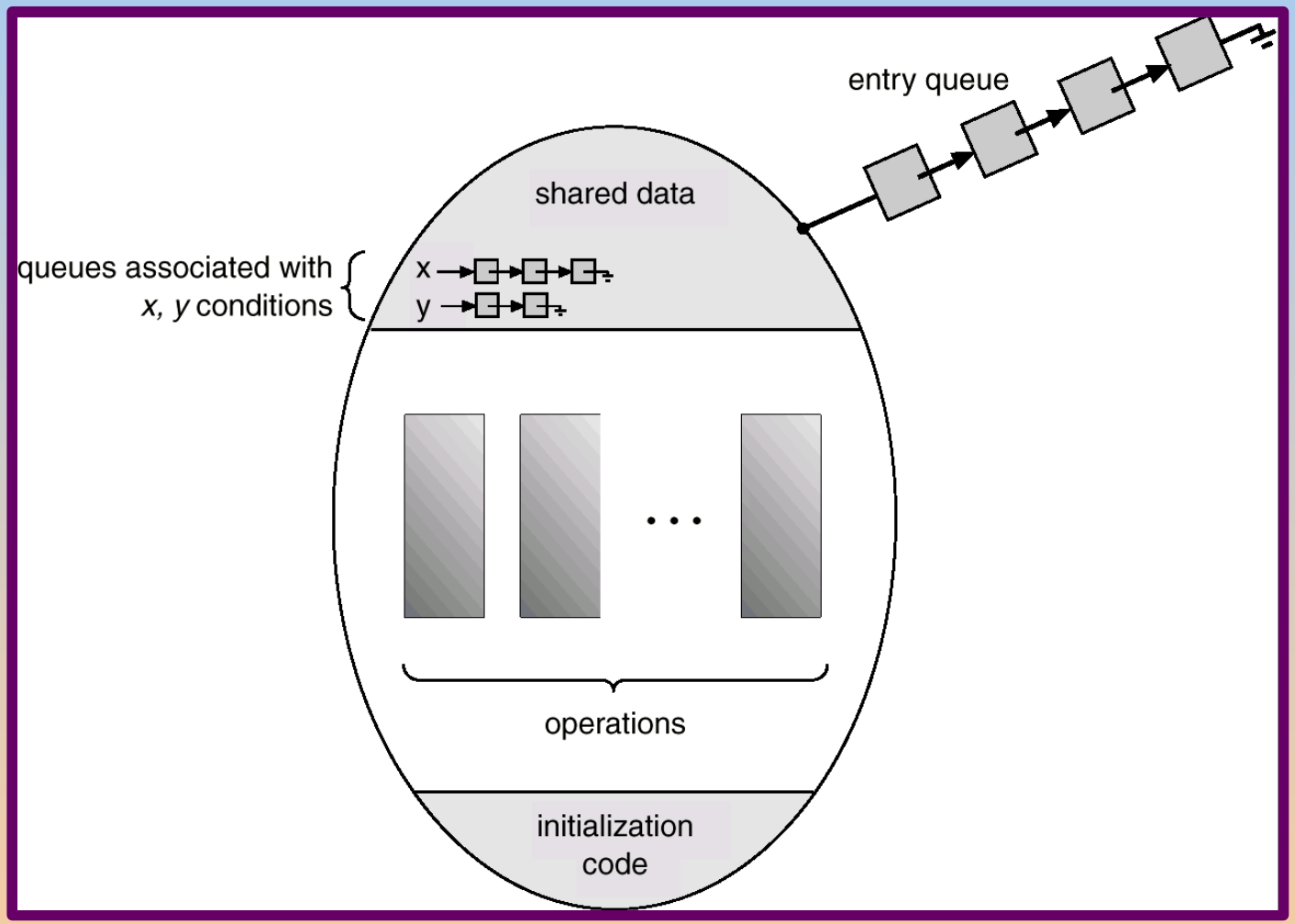
    x.signal();

  - A operação **x.signal** reativa somente um processo suspenso. Se não há processos suspensos, então a operação signal não tem efeito.

Vista esquemática de um monitor



Monitor com variáveis condition



Problema dos filósofos

```
monitor dp
{
  enum {pensando, faminto, comendo} state[5];
  condition self[5];
  void pickup(int i)
  void putdown(int i)
  void test(int i)
  void init() {
        for (int i = 0; i < 5; i++)
                state[i] = pensando;
  }
}
```

Problema dos filósofos

```
void pickup(int i) {
        state[i] = faminto;
        tes(i);
        if (state[i] != comendo)
                self[i].wait();
}

void putdown(int i) {
        state[i] = pensando;
        // testa vizinhos da esquerda e direita
test((i+4) % 5);
        test((i+1) % 5);
}
```

Problema dos filósofos

```
void test(int i) {
        if ( (state[(I + 4) % 5] != comendo) &&
          (state[i] == faminto) &&
          (state[(i + 1) % 5] != comendo)) {
                state[i] = comendo;
                self[i].signal();
        }
}
```

Implementação de monitores usando semáforos

- Variáveis

```
semaphore mutex;  // (inicialmente  = 1)
semaphore next;   // (inicialmente  = 0)
int next-count = 0;
```

- Cada procedimento externo F será substituído por:

```
wait(mutex);
    …
  corpo de F
    …
if (next-count > 0)
        signal(next)
else
        signal(mutex);
```

- Exclusão mútua dentro do monitor é garantida

Implementação de monitores usando semáforos(cont)

- Para cada variável de condução x, nós temos:

```
semaphore    x-sem; // (inicialmente  = 0)
int x-count = 0;
```

- A operação **x.wait** pode ser implementada como:

```
x-count++;
if (next-count > 0)
        signal(next);
else
        signal(mutex);
wait(x-sem);
x-count--;
```

Implementação de monitores usando semáforos(cont.)

- A operação **x.signal** pode ser implementada como:

```
if (x-count > 0) {
        next-count++;
        signal(x-sem);
        wait(next);
        next-count--;
}
```

Sincronização no Solaris

- Implementa uma variedade de locks para suportar multitarefa, multithreading (incluindo threads de tempo real) e multiprocessamento.
- Usa mutexes adaptativos visando eficiência quando está protegendo dados de segmentos de código pequenos.
- Usa variáveis do tipo condition e locks do tipo leitores-escritores quando grandes seções de código precisam acessar dados.

Sincronização no Windows XP

- Usa máscaras de interrupção para proteger acesso a recursos globais em sistemas monoprocessamento.
- Usa *spinlocks* em sistemas com multiprocessamento.
- Também disponibiliza objetos despachantes(dispatcher objects) que podem agir como mutexes e semáforos.
- Objetos despachantes também podem disponibilizar eventos. Um evento age como uma variável do tipo condition.