# Sistemas Operacionais

# Processos

## SEMÁFOROS

- Mecanismo criado para solucionar o problema de armazenar múltiplos WAKEUPS (E.W.Dijkstra)

- O mecanismo envolve a utilização de uma variável compartilhada chamada **semáforo**, e de duas operações primitivas indivisíveis que atuam sobre ela.

- A variável compartilhada pelos processos, poderá assumir valores inteiros não negativos e sua manipulação será restrita às operações **P** e **V** (ou **Down** e **Up**, ou **Wait** e **Signal**, respectivamente).

# Processos

## SEMÁFOROS

As operações que atuam em um semáforo denominado **s**, incluindo seus efeitos são definidos a seguir:

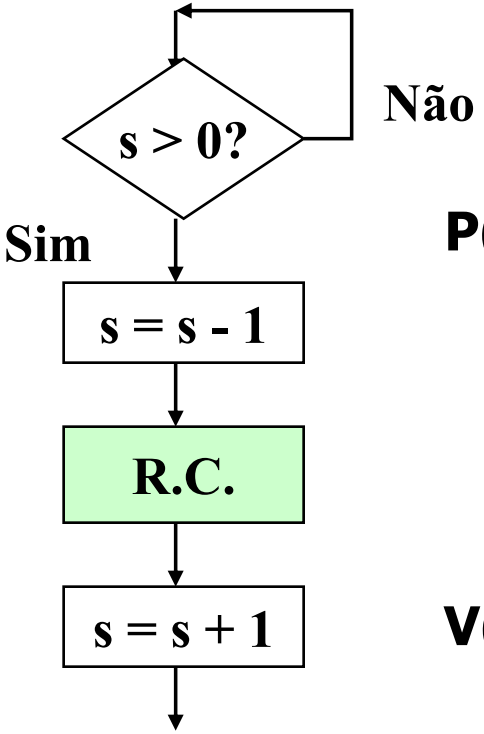
- **P(s):** Espera até que s>0 e então decrementa s;
- **V(s):** Incrementa s;

# Processos

## SEMÁFOROS

### 1ª. Implementação – Espera ocupada

Esta implementação é através da espera ocupada: não é a melhor, apesar de ser fiel à definição original.




**P(s):** Espera até que s>0 e então decrementa s;

**V(s):** Incrementa s;

# Processos

## SEMÁFOROS

### 2ª. Implementação – Associando uma fila Qi a cada semáforo si

Quando se utiliza este tipo de implementação, o que é muito comum, as primitivas P e V apresentam o seguinte significado:

**P(si):** se si>0 e então decrementa si (e o processo continua)
senão bloqueia o processo, colocando-o na fila Qi;

**V(si):** se a fila Qi está vazia então incrementa si
senão acorda processo da fila Qi;

# Processos

## SEMÁFOROS

O semáforo é um mecanismo bastante geral para resolver problemas de sincronismo e exclusão mútua.

### Tipos de Semáforos

- **Semáforo geral:** se o semáforo puder tomar qualquer valor inteiro não negativo;

- **Semáforo binário (booleano):** só pode tomar os valores 0 e 1.

# Processos

## Problema da Exclusão Mútua com Semáforos

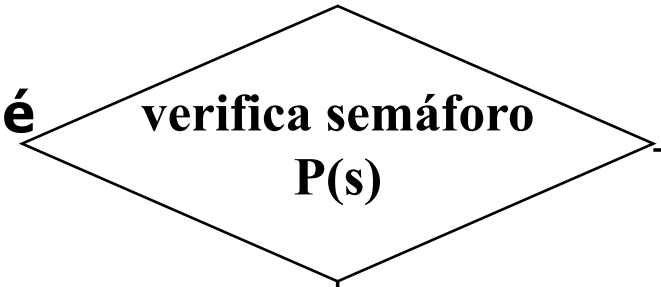
```
Program exclusao_mutua;
Var Mutex: semaphore;
Begin                    /* inicio do programa principal */
        Mutex:=1;   /* condição inicial */
        Cobegin     /* inicio dos processos concorrentes */
                Begin /* Processo 1 */
                        Repeat
                          ...
                          P(Mutex);
                                  Seção_crítica_1;
                          V(Mutex);
                          ...
                        until false;
                End;

                /* outros processos*/
        Coend
End.
```

# Processos

## SEMÁFOROS e a Sincronização Baseada em Condições

**Sincronização de processos baseada em condições** → **Cada condição é representada por um semáforo**

**Se um processo precisa verificar se uma condição é verdadeira antes de prosseguir**



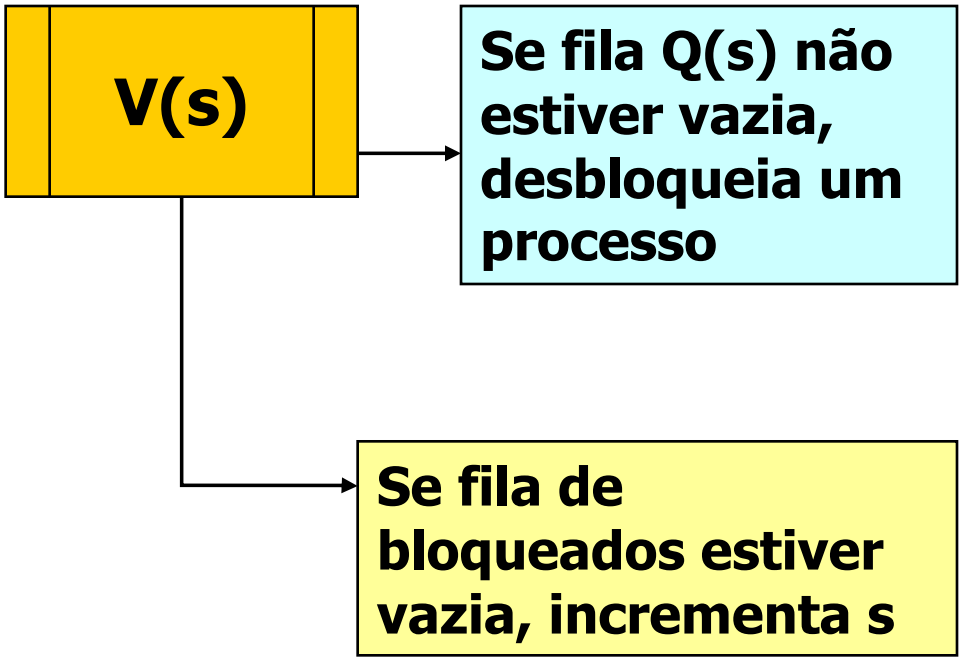

**Condição falsa = (s=0) bloqueia processo**

**Condição verdadeira = (s>0) processo continua**

# Processos

- **SEMÁFOROS e a Sincronização Baseada em Condições**

**Se um processo torna uma condição verdadeira, deve sinalizar isto através do semáforo, usando V(s)**



# Processos

## Problema do Produtor/ Consumidor usando Semáforos

```
Program Produtor_consumidor;
Const max = ...;
Type msg=...;
Var mensagem: msg;
Var buffer:...;
Var p, c: 0..max –1; /* ponteiros do buffer*/
Var cheio, vazio, mutex: semaphore;

Procedure Depositar(m:msg)
Begin ... End;

Procedure Retirar (var m:msg)
Begin ... End;
/* inicio do programa principal */
Begin
        cheio:=0;
        vazio:=max;
        mutex:=1;
        p:=0;
        c:=0;
```

```
Cobegin  /* inicio dos processos concorrentes */
  Begin  /* processo Produtor */
     Repeat
            ...
            produção da mensagem;
            P(vazio);  /* sincronização */
            P(mutex);  /* exclusão mútua */
              Depositar(mensagem);
            V(mutex); /*liberação da exclusão mútua */
            V(cheio); /*sincronização para o consumidor */
    Until false;
  End;

  Begin  /* processo Consumidor */
    Repeat
            P(cheio);  /* sincronização */
            P(mutex);  /* exclusão mútua */
              Retirar(mensagem);
            V(mutex); /*liberação da exclusão mútua */
            V(vazio); /*sincronização para o produtor */
            consumo da mensagem;
            ...
    Until false;
  End
 Coend
End.
```

# Processos

- **Principais vantagens dos semáforos**
- A ordem de execução das operações é irrelevante. Uma primitiva V disparada antes de uma P, "guarda" sua ocorrência no valor do semáforo.
- Também podem ser usados para controlar a alocação de recursos N-plicados no sistema:
  - Se um recurso é N-plicado no sistema, então N processos poderão alocá-lo, antes que seja necessário suspender um processo.
  - A alocação deve ser feita usando-se P(s) e a liberação usando-se V(s).
  - O semáforo deve ser iniciado com o número de recursos (N).

# Processos

## REGIÕES CRÍTICAS

- As primitivas **P** e **V** podem levar a situações irreproduzíveis, se usadas erroneamente (ex., um **P** sem um **V** correspondente).
- Para minimizar este problema, criou-se o comando de linguagem chamado **Região Crítica.**
- Com este comando é possível a um **compilador** verificar se determinada variável compartilhada está sendo manipulada no interior de uma seção crítica de código.

# Processos

## REGIÕES CRÍTICAS

- A varíavel compartilhada pode ser declarada do seguinte modo:

  **VAR v: shared T;** /* onde T é o tipo da variável */

- O comando região crítica tem o seguinte formato:

  **Region v do S;** /* onde S é a seção do programa onde a variável **v** será manipulada */

- As regiões críticas associadas a uma mesma variável compartilhada são executadas com exclusão mútua no tempo.

# Processos

## Exemplo de Exclusão Mútua usando Regiões Críticas

```
Program exclusao_mutua;
Var v: shared T;
Begin                          /* inicio do programa principal */
 Cobegin     /* inicio dos processos concorrentes */
  Begin /* Processo 1 */
   Repeat
    ...
      Region v do
            Seção_crítica_1;
             ...
   until false;
  End;

  Begin /* Processo 2 */
   Repeat
   ...
      Region v do
            Seção_crítica_2;
             ...
   until false;
  End;
 /* outros processos*/
 Coend
End.
```

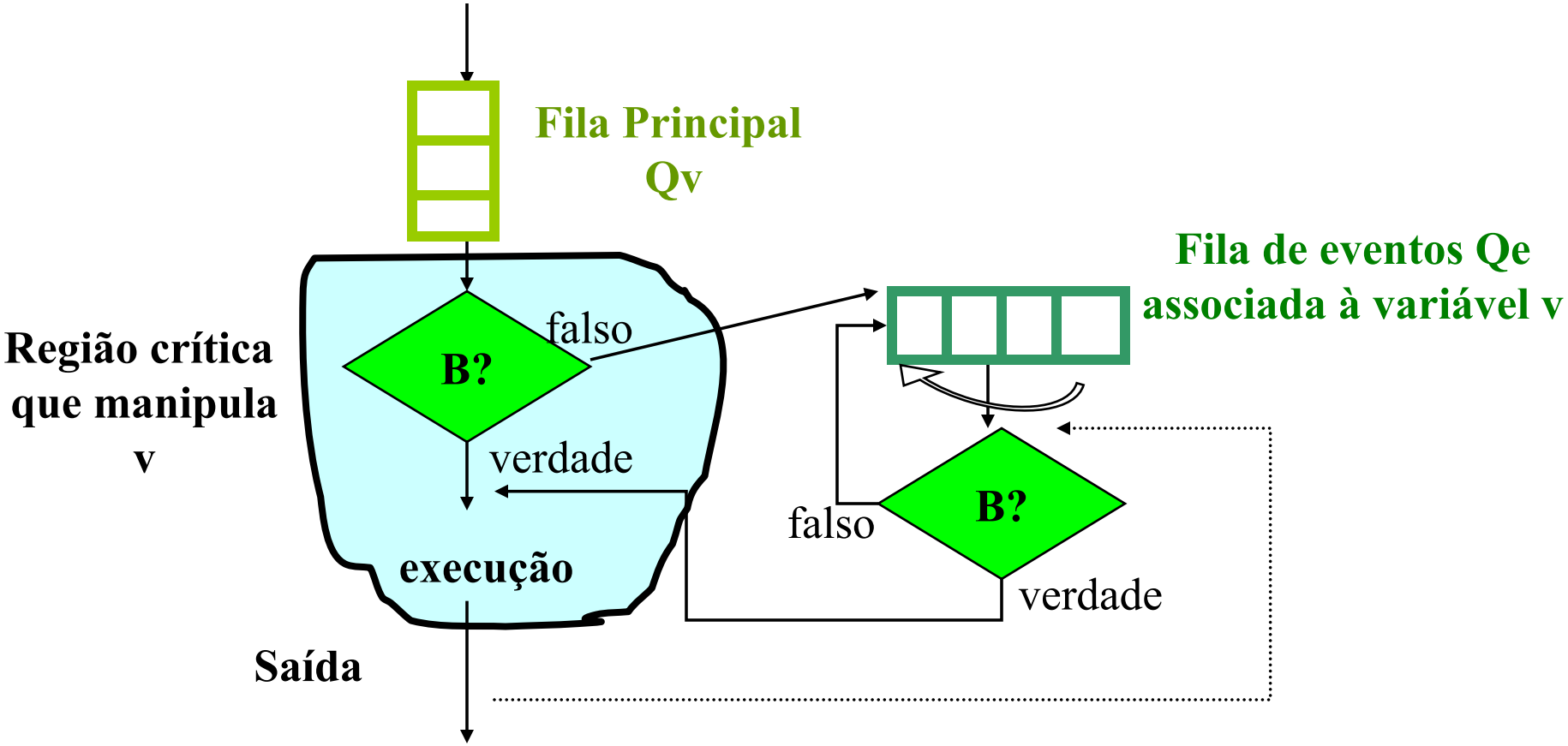
# Processos

- **REGIÕES CRÍTICAS CONDICIONAIS**
- Criadas para suprir a necessidade de sincronização diferente da exclusão mútua.
- **Notação:**

```
VAR v: shared T; /* onde T é o tipo da variável */
Region v  do
        begin
         await (B) /* onde B é uma expressão booleana */
         S; /* onde S é a seção do programa
                 onde a variável v será manipulada */
        end;
```

# Processos

- **REGIÕES CRÍTICAS CONDICIONAIS**



# Processos

## REGIÕES CRÍTICAS CONDICIONAIS

## Descrição do Funcionamento

- Um processo primeiramente deverá entrar na R.C., cujo acesso é controlado por uma fila Qv, associada ao recurso compartilhado, e chamada fila principal; em seguida a continuação do processo dentro da R.C. ficará condicionada à avaliação de uma expressão booleana.

- Se o resultado for verdadeiro, o processo poderá continuar sua execução, mas se for falso, ele será bloqueado, indo para a fila Qe existente fora da R.C., e denominada fila de eventos, que também será associada à variável compartilhada. Depois disso, o processo abandonará a R.C., permitindo a entrada de outro processo.

# Processos

**Exemplo do Problema do Produtor/Consumidor usado Regiões Críticas Condicionais**

```
Program produtor_consumidor;
Const max = ...;
Type msg = ...;

Type B = shared record
            buffer: array[0..max-1] of msg;
            p, c: 0..max;
            cont_buf:0..max; /* contador do buffer */
         end;
Var b:B;
Var mensagem : msg;

Procedure depositar(m:msg)
Begin ... End;
```

# Processos

```
Procedure retirar (var m:msg)
Begin ... End;

Begin  /* inicio do programa principal */
      region b do
            Begin
                  p:=0;
                  c:=0;
                  cont_buf:=0;
            end;
```

# Processos

```
Cobegin  /* inicio dos processos concorrentes */
    Begin  /* processo Produtor */
       Repeat
       ...
       produzir_mensagem();
       region b do
            Begin
             await(cont_buf < max);
             depositar(mensagem);
             cont_buf:=cont_buf+1;
            end
       until false;
    End;
```

# Processos

```
Begin  /* processo Consumidor */
   Repeat
   region b do
        Begin
         await(cont_buf > 0);
         retirar(mensagem);
         cont_buf:=cont_buf-1;
        end;
     consumir_mensagem();
     until false;
   End;
  Coend
End.
```

# Monitores

- **Monitor** – unidade de sincronização de alto nível proposta na década de 70
- Consiste de uma coleção de procedimentos, variáveis e estruturas de dados, agrupada em um tipo especial de módulo ou pacote
- Os processos podem chamar os procedimentos em um monitor, mas não podem ter acesso direto às estruturas internas de dados do monitor a partir de procedimentos declarados fora do monitor
- Somente um processo pode estar ativo dentro de um monitor em um dado momento
- Cabe ao compilador implementar a exclusão mútua

# Monitores

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Exemplo de um monitor

# Monitores

- Para a sincronização baseada em condições: uso de variáveis condicionais, com duas operações sobre elas:

  *wait* e *signal*.
- Quando um procedimento do monitor descobre que não pode prosseguir, emite um *wait* sobre uma variável condicional (por exemplo, *full* )
- Isto bloqueia o processo que está chamando e permite que outro processo, proibido de entrar anteriormente no monitor agora entre.
- Este outro processo pode acordar o processo adormecido, a partir da emissão do *signal* para a variável condicional que o mesmo está esperando.

# Monitores

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

- Delineamento do problema do produtor-consumidor com monitores
  - somente um procedimento está ativo por vez no monitor
  - o buffer tem N lugares