# Programação Concorrente com Semáforos (2)

José C. Cunha, DI/FCT/UNL

Aplicações que exploram a concorrência entre processos:

a) processos independentes entre si

b) processos cooperantes: resolvem partes de uma aplicação → precisam de comunicar/sincronizar

- Se comunicam por memória partilhada:

  ***-- eficiente porque explora acesso a memória comum***

  ***-- exige sincronização explícita: Semáforos***

# 1-Regiões críticas - Exclusão mútua

- **Acesso a memória partilhada: um processo acede de cade vez**
- **Semáforo de exclusão mútua:**

**Exmut: valor inicial 1**

**Utilizado do seguinte modo:**

**--- em cada processo:**

```
???
Acede;      -- operações de acesso
???
```

# 1-Regiões críticas - Exclusão mútua

- **Acesso a memória partilhada: um processo acede de cade vez**
- **Semáforo de exclusão mútua:**

**Exmut: valor inicial 1**

**Utilizado do seguinte modo:**

**--- em cada processo:**

```
P(exmut);  -- pede autorização
Acede;     -- operações de acesso
```

# 1-Regiões críticas - Exclusão mútua

- **Acesso a memória partilhada: um processo acede de cade vez**
- **Semáforo de exclusão mútua:**

**Exmut: valor inicial 1**

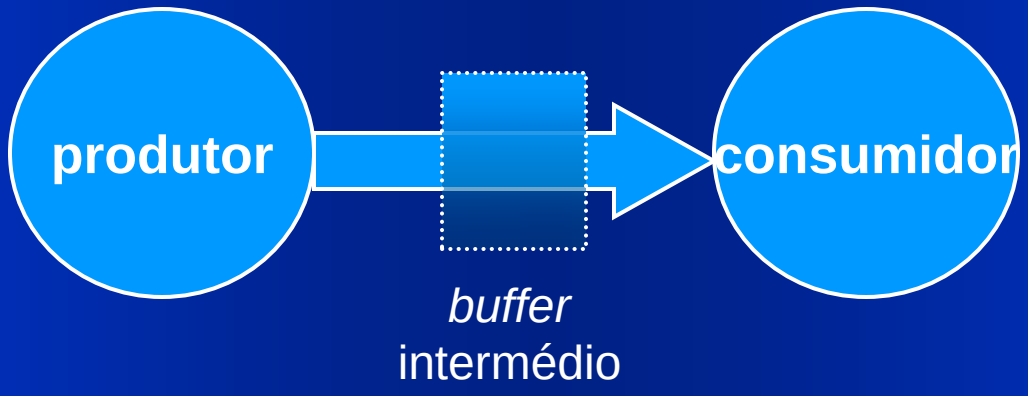
**Utilizado do seguinte modo:**

**--- em cada processo:**

```
P(exmut);  -- pede autorização
Acede;     -- operações de acesso
V(exmut);  -- indica que acabou
```

# 2 - Produtor(es) - Consumidor(es)

- **Processo Produtor:**
  - **produz resultados**
  - **põe no *buffer***
- **Processo Consumidor:**
  - **retira do *buffer***
  - **trata valores obtidos**



# 2 - Produtor - Consumidor

**Comunicam através de um Buffer em memória partilhada**

**a) Acesso exclusivo a cada célula do buffer:**

# 2 - Produtor - Consumidor

**Comunicam através de um Buffer em memória partilhada**

**a) Acesso exclusivo a cada célula do buffer:**

**-- semáforo exmut inicial 1**

# 2 - Produtor - Consumidor

**Comunicam através de um Buffer em memória partilhada**

**a) Acesso exclusivo a cada célula do buffer:**

**-- semáforo exmut inicial 1**

**b) Consumidor aguarda que Buffer Não Vazio:**

# 2 - Produtor - Consumidor

**Comunicam através de um Buffer em memória partilhada**

**a) Acesso exclusivo a cada célula do buffer:**

**-- semáforo exmut inicial 1**

**b) Consumidor aguarda que Buffer Não Vazio:**

**-- semáforo s_preenchidas inicial 0**

# 2 - Produtor - Consumidor

**Comunicam através de um Buffer em memória partilhada**

**a) Acesso exclusivo a cada célula do buffer:**

**-- semáforo exmut inicial 1**

**b) Consumidor aguarda que Buffer Não Vazio:**

**-- semáforo s_preenchidas inicial 0**

**(conta número de células preenchidas)**

**O Produtor faz V(s_preenchidas) depois de cada inserção de um novo elemento.**

# 2 - Produtor - Consumidor

**Comunicam através de um Buffer em memória partilhada**

**a) Acesso exclusivo a cada célula do buffer:**

**-- semáforo exmut inicial 1**

**b) Consumidor aguarda que Buffer Não Vazio:**

**-- semáforo s_preenchidas inicial 0**

**(conta número de células preenchidas)**

**O Produtor faz V(s_preenchidas) depois de cada inserção de um novo elemento.**

**O Consumidor faz P(s_preenchidas) antes de tentar remover um elemento**

## c) Produtor aguarda que Buffer Não Cheio:

c) **Produtor aguarda que Buffer Não Cheio:**

-- semáforo **s_vazias** inicial N (capacidade do buffer)

(conta número de células livres)

c) Produtor aguarda que Buffer Não Cheio:

-- semáforo s_vazias inicial N (capacidade do buffer)

(conta número de células livres)

O Produtor faz P(s_vazias) antes de tentar a inserção de um novo elemento.

c) **Produtor aguarda que Buffer Não Cheio:**

-- semáforo **s_vazias** inicial N (capacidade do buffer)

(conta número de células livres)

O Produtor faz **P(s_vazias)** antes de tentar a inserção de um novo elemento.

O Consumidor faz **V(s_vazias)** depois de remover um elemento

**Produtor:**

```
Produz();
P(s_vazias);
P(exmut);
Insere();
V(exmut);
```

**Consumidor:**

```
P(exmut);
Remove();
V(exmut);
V(s_vazias);
```

**s_vazias: inicialmente igual a N (capacidade do buffer)**

**exmut: inicial 1 (só um processo de cada vez)**

**Produtor:**

```
Produz();
P(s_vazias);
P(exmut);
Insere();
V(exmut);
V(s_preenchidas);
```

**Consumidor:**

```
P(s_preenchidas);
P(exmut);
Remove();
V(exmut);
V(s_vazias);
```

**s_vazias: inicialmente igual a N (capacidade do buffer)**

**s_preenchidas: inicial igual a 0**

**exmut: inicial 1 (só um processo de cada vez)**

| Produtor: | | Consumidor: |
|---|---|---|
| Produz(); | *Problema!!!* → | P(exmut); |
| P(s_vazias); | | P(s_preenchidas); |
| P(exmut); | | Remove(); |
| Insere(); | | V(exmut); |
| V(exmut); | | V(s_vazias); |
| V(s_preenchidas); | | |

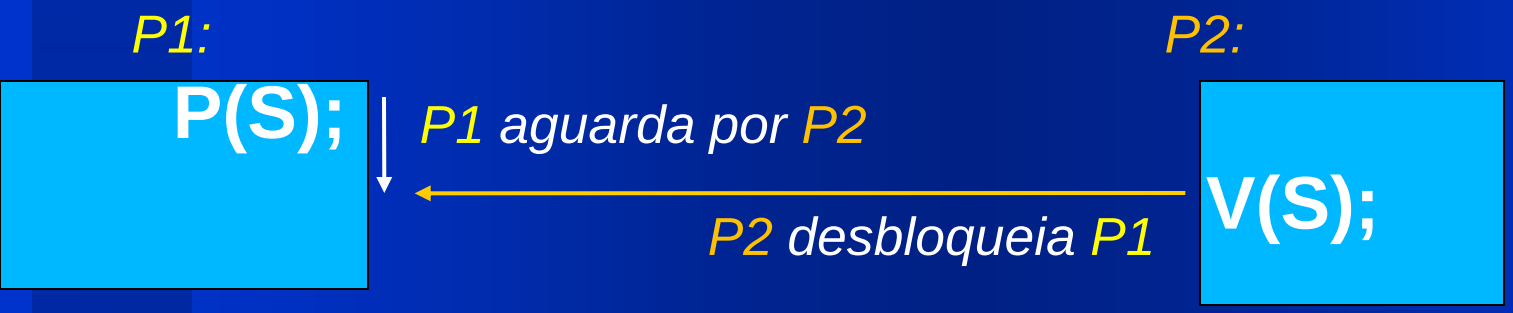
s_vazias: inicialmente igual a N (capacidade do buffer)

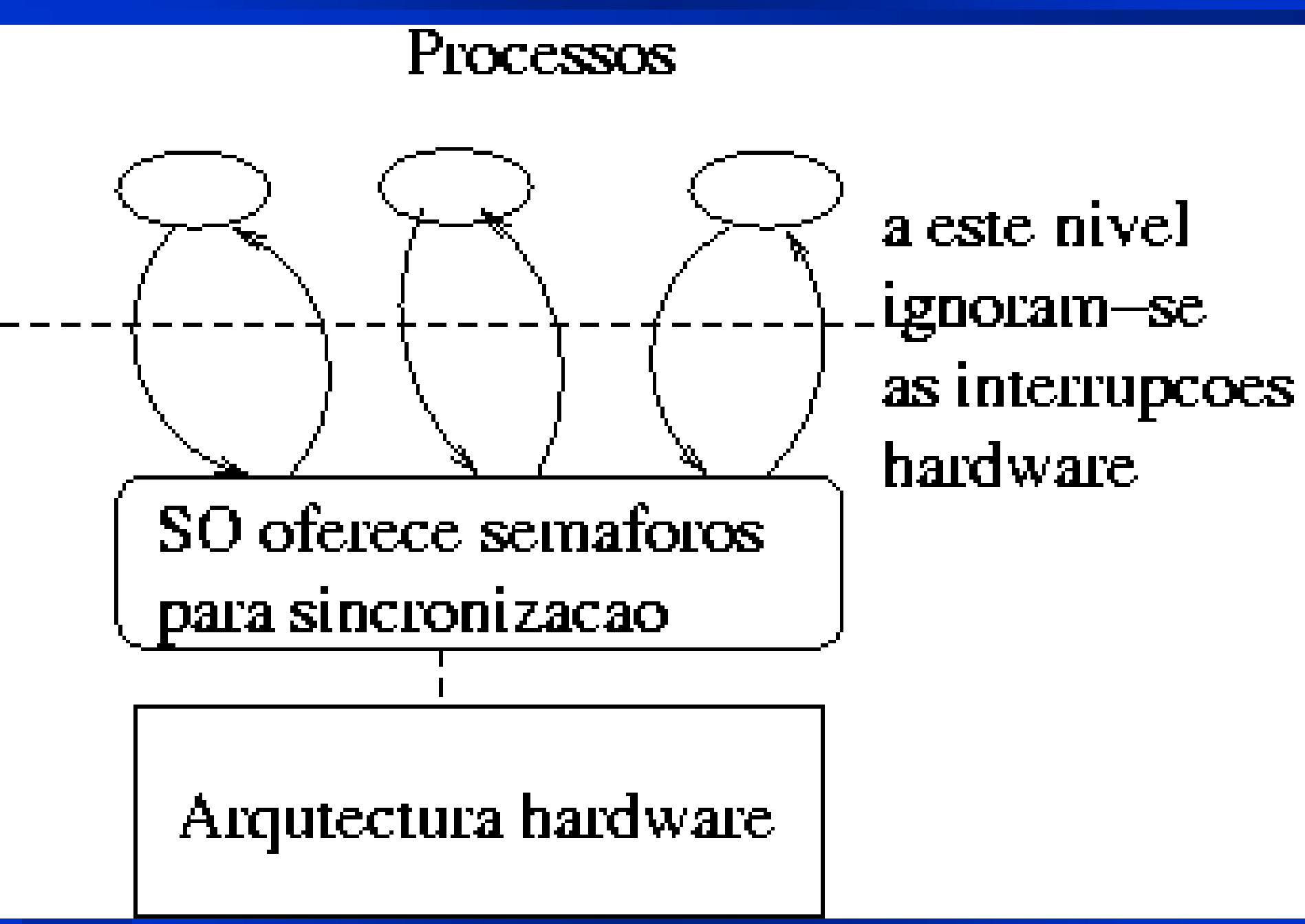
s_preenchidas: inicial igual a 0

exmut: inicial 1 (só um processo de cada vez)

# 3- Simples sincronização

- **Um processo aguarda um 'evento' vindo de outro processo**
  - **exemplo: aguarda que uma var. partilhada seja alterada**
- **Solução:**
  - **semáforo S, com valor inicial 0 (não ocorreu evento)**
  - **Um processo produz um evento:**
    **V(S) (o semáforo é incrementado)**
  - **O processo que testa a ocorrência do evento:**
    **P(S) (testa o evento, se ainda não houve evento fica a aguardar)**




Processos
a este nivel
ignoram–se
as interrupcoes
hardware
SO oferece semaforos
para sincronizacao
Arqutectura hardware

# 4 - Clientes e Servidores

- **Um Servidor é um processo que se executa concorrentemente com outros processos num SO com multiprogramação**
- **Cada Servidor é responsável por tratar pedidos de um determinado tipo -> um Serviço**
- **Exemplos:**

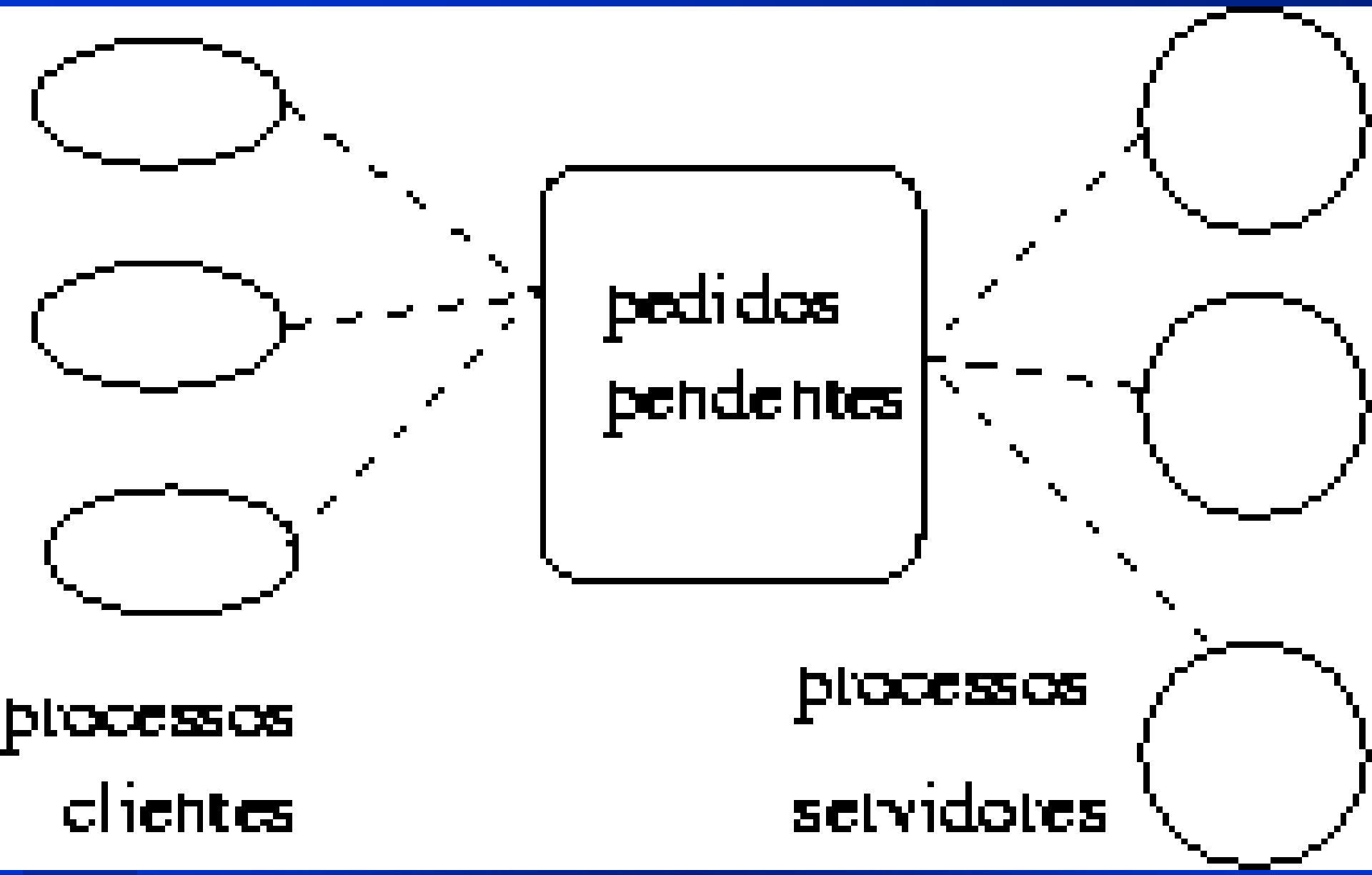
**-- serviço de impressão: SPOOLer**

**-- serviço de ficheiros;**

**-- serviço de acesso a uma base de dados;**

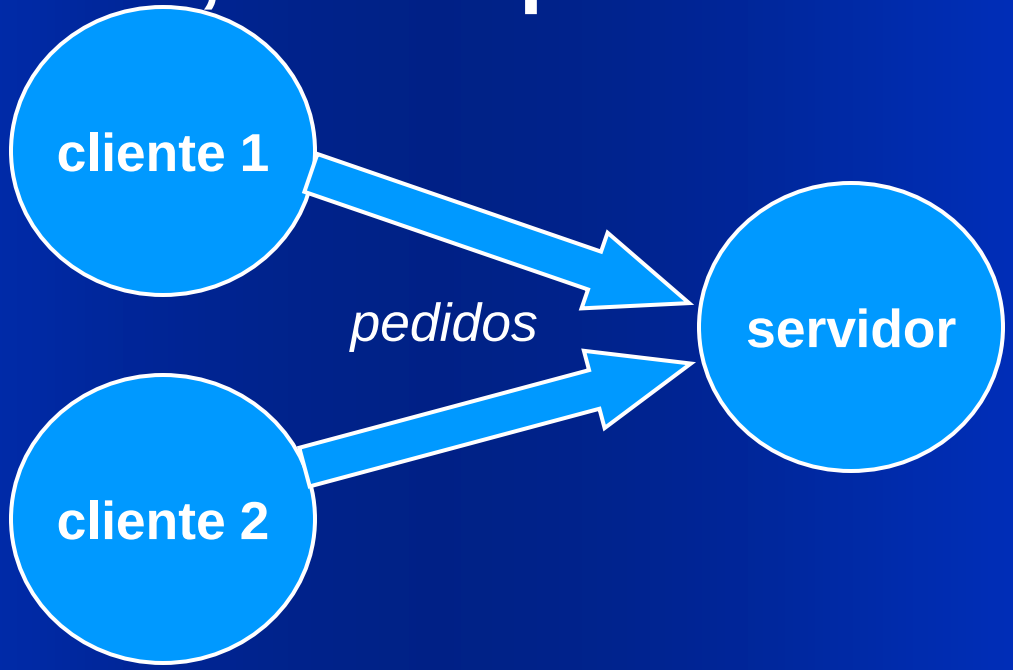
- **Um mesmo Serviço pode ser tratado por múltiplos Servidores concorrentes**

**Que vantagens?**


pedidos
pendentes
processos
clientes
processos
servidores

# Clientes e Servidor

- **Cada Servidor é responsável por tratar determinados pedidos → um Serviço**
- **Vamos considerar vários clientes concorrentes, um só processo servidor**



**Clientes:**

-- enviam pedidos aos servidores

-- aguardam resposta (ou não)

**Clientes:**

-- enviam pedidos aos servidores

-- aguardam resposta (ou não)

**Servidores:**

-- aguardam pedidos dos clientes

-- tratam cada pedido recebido

-- enviam respostas (ou não)

**Clientes:**

-- enviam pedidos aos servidores

-- aguardam resposta (ou não)

**Servidores:**

-- aguardam pedidos dos clientes

-- tratam cada pedido recebido

-- enviam respostas (ou não)

**Operação Assíncrona:**

-- os clientes vão pondo pedidos, independentes dos ritmos a que os servidores os tratam

-- os servidores podem tratar múltiplos pedidos concorrentemente

Mas, se um cliente precisa de uma resposta

então deve esperar pelo fim do tratamento

# Cliente:

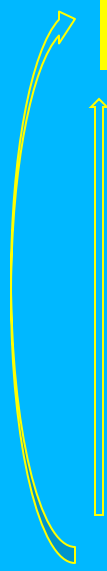
```
P(exmut);
 insere-pedido();
V(exmut);
V(pedidos);
   ……
```

ou continua

ou aguarda resposta

# Servidor:

```
|-> P(pedidos);
    P(exmut);
     remove-pedido();
    V(exmut);
    trata-pedido();
<- repete o ciclo;
```



- Uma Fila de Pedidos em Memória Partilhada
- Um semáforo contador pedidos, inicial 0
- Um semáforo exclusão mútua exmut inicial 1

- **Veremos mais tarde, outras formas de os clientes comunicarem com os servidores:**

**-- por envio e recepção de mensagens**

# 5- Operações síncronas

- **O processo aguarda até que a operação se complete: exemplo read()**
- **Implementação com semáforos:**

**-- um semáforo pedido_OK inicial 0**

**-- no programa que realiza read():**

**ver se buffer tem dados;**

**se não tiver:**

**depois de lançar a operação de leitura**

**fazer P(pedido_OK);**

**e o processo invocador bloqueia-se**

**-- quando os dados vierem:**

**algum processo interno faz V(pedido_OK);**

# 6- Operações assíncronas

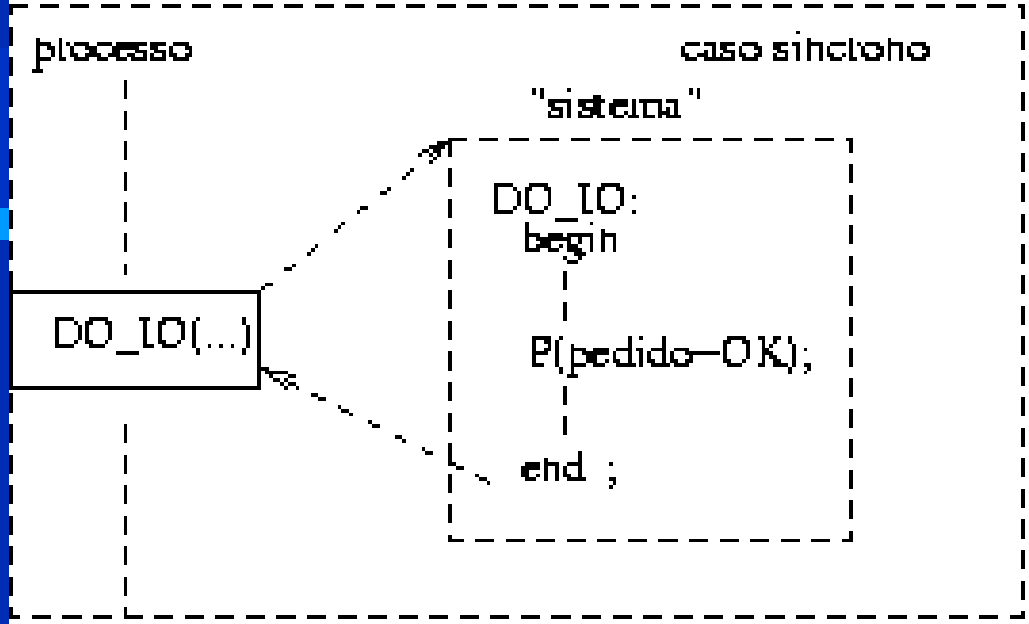
- **O processo NÃO aguarda até que a operação se complete: exemplo read_asynchronous()**
- **Implementação:**

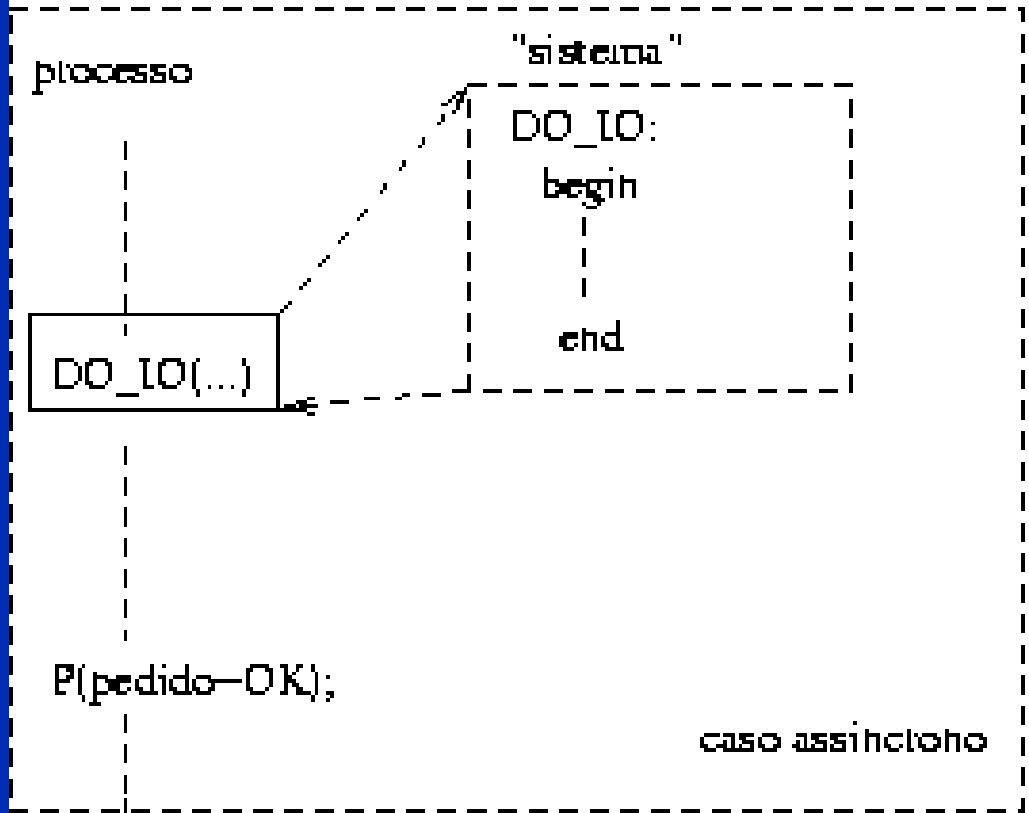
**-- no programa que realiza read_asynchronous():**

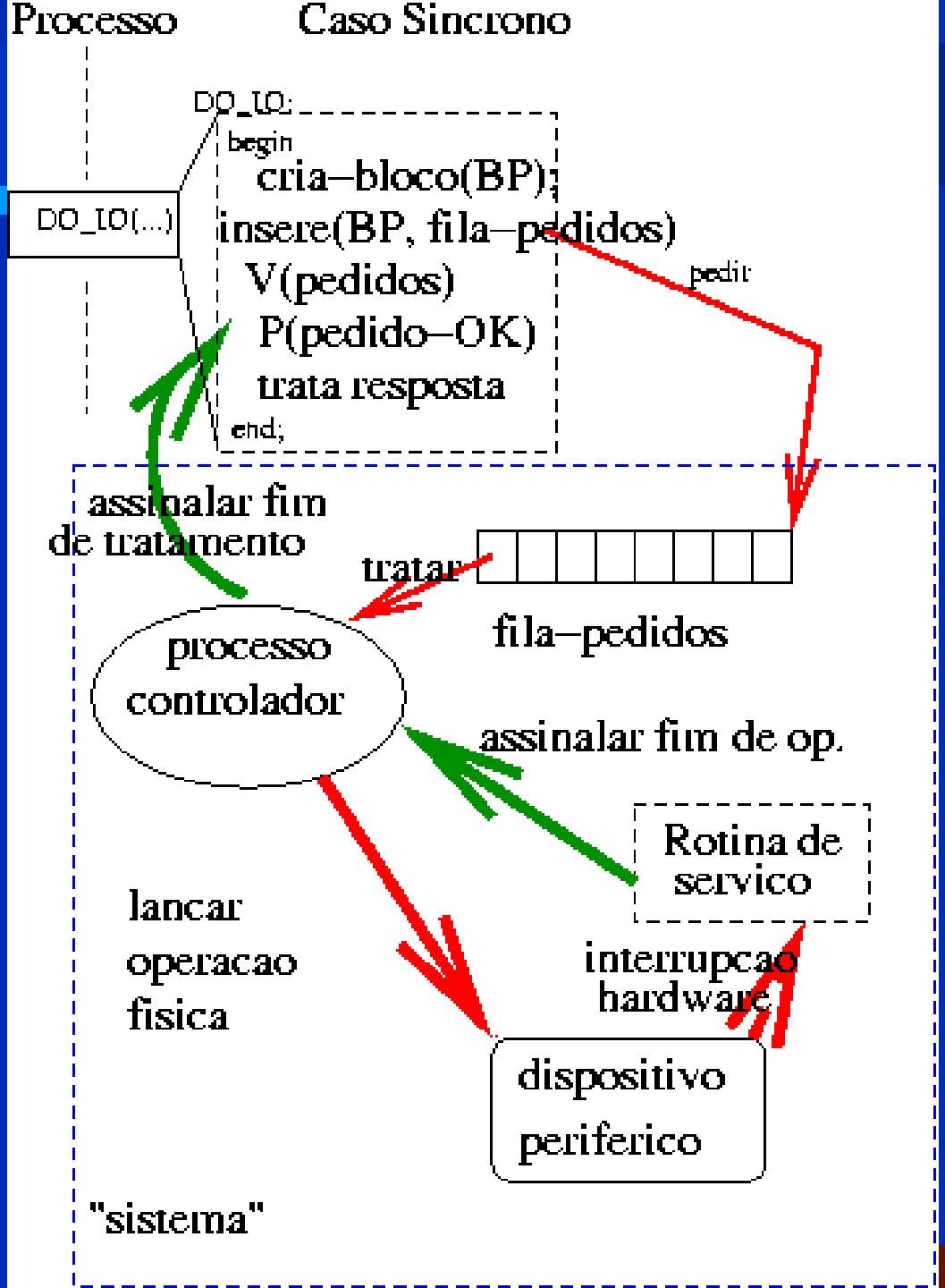
**ver se buffer tem dados;**
**se não tiver:**
**depois de lançar a operação de leitura**
**RETORNAR logo ao invocador;**
**-- e o processo invocador continua a executar**

**-- quando os dados vierem:**
**algum processo interno faz V(pedido_OK);**

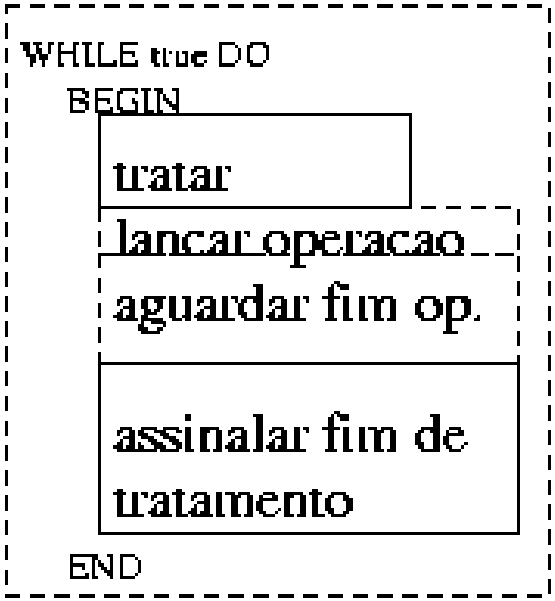
**-- quando o invocador precisar dos dados:**
**Fazer P(pedido_OK); explicitamente**


processo
caso sincrono
"sistema"
DO_IO:
begin
DO_IO(...)
P(pedido-OK);
end ;


processo
"sistema"
DO_IO:
begin
end
DO_IO(...)
P(pedido-OK);
caso assincrono


Processo
Caso Sincrono
DO_IO:
begin
cria–bloco(BP);
insere(BP, fila–pedidos)
V(pedidos)
P(pedido–OK)
trata resposta
end;
DO_IO(...)
pedir
assinalar fim
de tratamento
tratar
fila–pedidos
processo
controlador
assinalar fim de op.
Rotina de
servico
lancar
operacao
fisica
interrupcao
hardware
dispositivo
periferico
"sistema"

processo controlador:




Semaforos:

pedidos: inicial 0

fila–exmut: inicial 1

fim–op: inicial 0

pedidos–OK: inicial 0

tratar:

P(pedidos)
P(fila–exmut)
remover(BP)
V(fila–exmut)

aguardar fim op.:

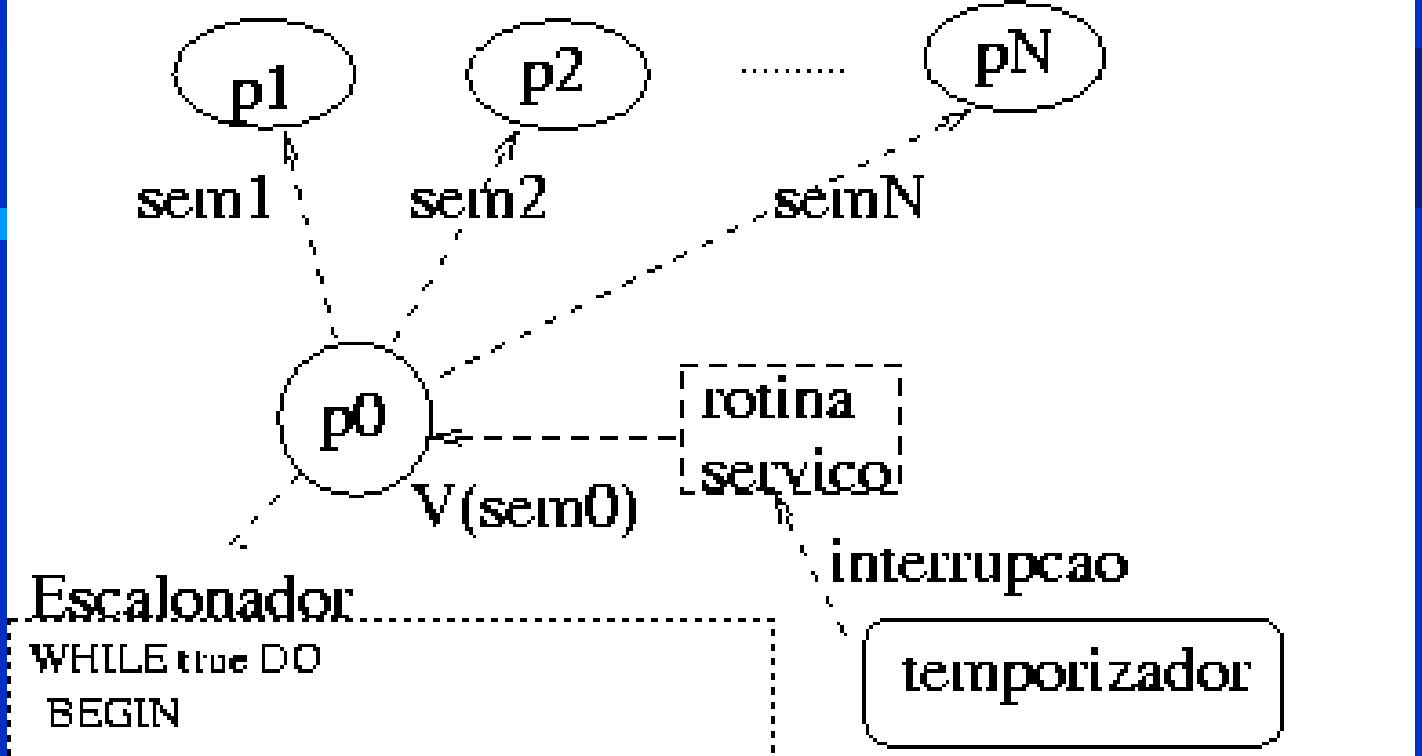
P(fim–op)

assinalar fim de tratamento:

V(pedidos–OK)

# 7 - Um Controlador de Tarefas

- **Pretende-se activar periodicamente, um conjunto fixo de N tarefas concorrentes**
- **Cada Tarefa é executada por um processo concorrente**
- **Cada Tarefa_ i deve ser activada ao fim de um intervalo de tempo Ti.**
- **Um Temporizador hardware gera uma interrupção por cada 'tick' do relógio.**
- **Deadline(i): o número de 'ticks' que faltam até ao momento de activação da Tarefa_i**
- **Assume-se que cada tarefa demora um tempo total de execução inferior ao tempo até ao próximo instante de activação.**




Escalonador

```
WHILE true DO
 BEGIN
  P(sem0);
  FOR pi(i:1..N) DO
    deadline(i):=deadline(i)–1
      IF (deadline(i) = 0 THEN
          deadline(i) := Ti;
          V(semi);

 END
```

Cada Processo pi

```
(i: 1..N)
WHILE true DO
  BEGIN
    P(semi);
    Executar
  END
```

# 8 - Leitores e Escritores

- **Processos concorrentes acedem a uma Base de dados partilhada em memória**

**Leitores: consultam a informação**

**Escritores: actualizam a informação**

- **Que restrições de sincronização?**

# 8 - Leitores e Escritores

- **Processos concorrentes acedem a uma Base de dados partilhada em memória**

**Leitores: consultam a informação**

**Escritores: actualizam a informação**

- **Que restrições de sincronização?**

**a)- 1 Escritor não pode ser concorrente com qualquer outro processo, seja Leitor ou Escritor**

# 8 - Leitores e Escritores

- **Processos concorrentes acedem a uma Base de dados partilhada em memória**

**Leitores: consultam a informação**

**Escritores: actualizam a informação**

- **Que restrições de sincronização?**

**a)- 1 Escritor não pode ser concorrente com qualquer outro processo, seja Leitor ou Escritor**

**b)- qualquer número de Leitores concorrentes entre si (só Leitores)**

# 8 - Leitores e Escritores

- **Processos concorrentes acedem a uma Base de dados partilhada em memória**

**Leitores: consultam a informação**

**Escritores: actualizam a informação**

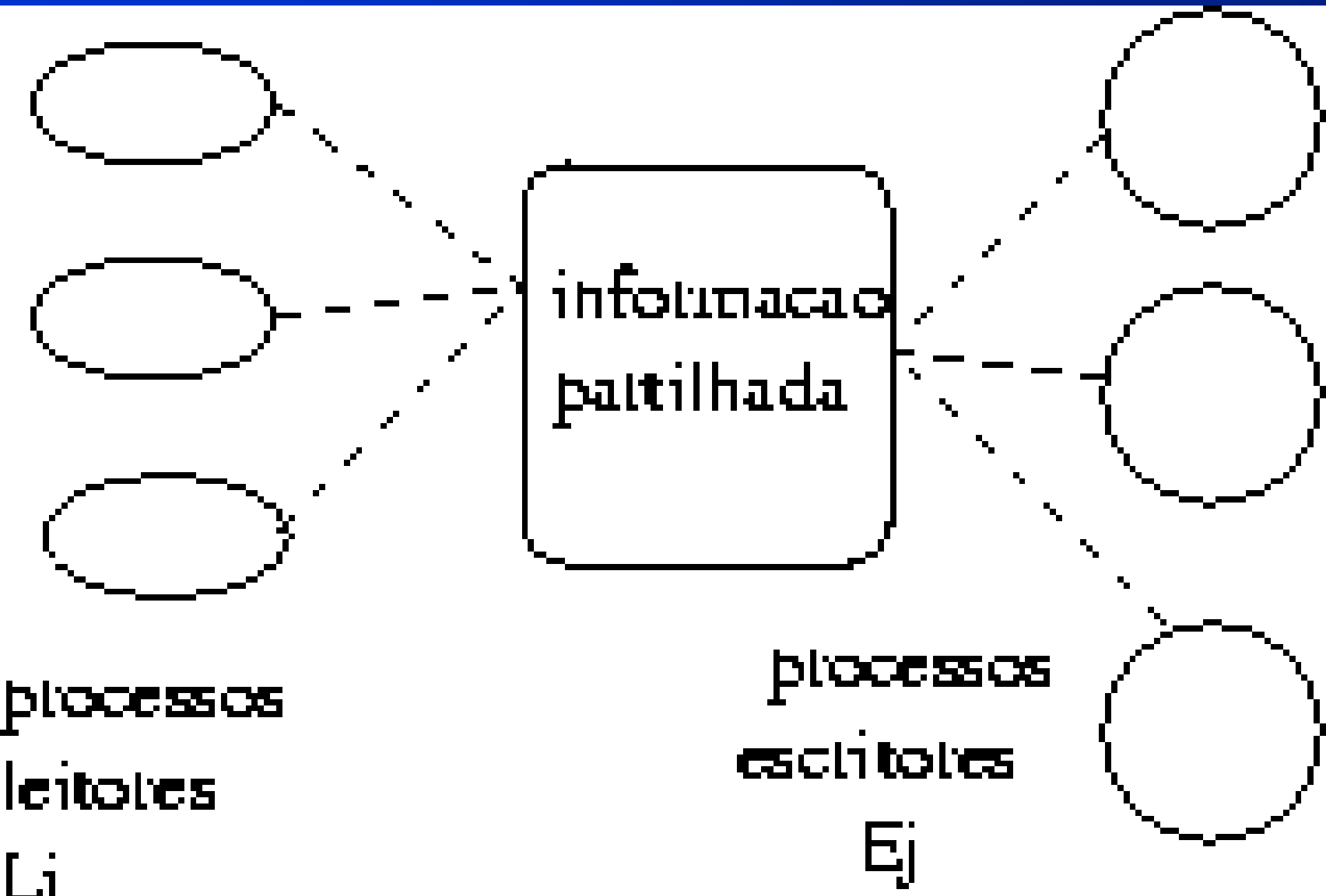
- **Que restrições de sincronização?**

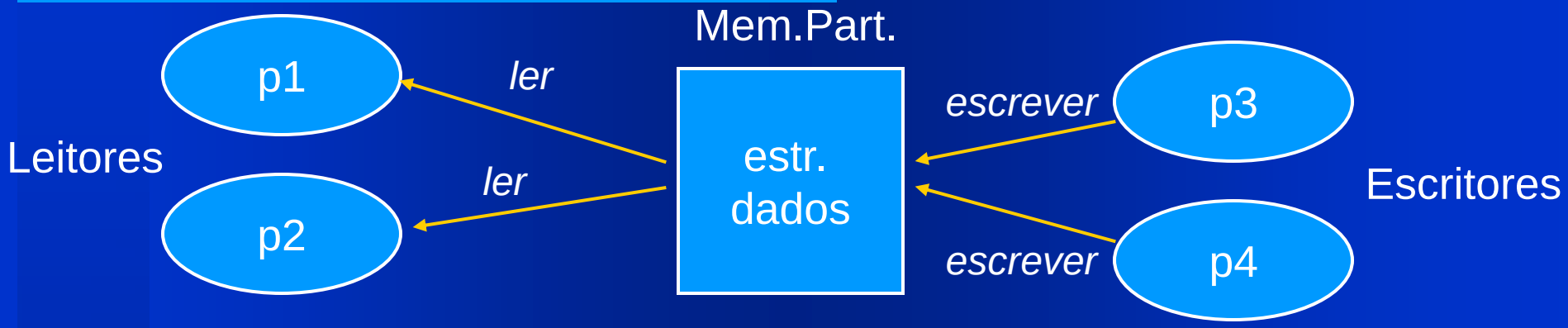
**a)- 1 Escritor não pode ser concorrente com qualquer outro processo, seja Leitor ou Escritor**

**b)- qualquer número de Leitores concorrentes entre si (só Leitores)**

**A base de dados: em Memória Partilhada**

**A sincronização: por Semáforos**


informacao
partilhada
processos
leitores
Lj
processos
escritores
Ej




*Leitor:*

```
Pede-para-Ler();  (1)
Ler();
Termina-de-Ler(); (2)
```

*Escritor:*

```
Pede-para-Escrever(); (3)
Escrever();
Termina-de-Escrever(); (4)
```

- Como programar as acções 1, 2, 3 e 4?

# Esquema de uma solução

- **Cada Escritor necessita de acesso exclusivo → semáforo exclusão mútua**
  - **nenhum outro processo, Leitor ou Escritor, pode entrar na sua região crítica**

# Esquema de uma solução

- **Cada Escritor necessita de acesso exclusivo → semáforo exclusão mútua**
  - **nenhum processo, Leitor ou Escritor, pode entrar na sua região crítica**
- **O primeiro Leitor obtém acesso para leitura**

**→ pede exclusão mútua**

  - **não pode deixar "entrar" Escritores**

# Esquema de uma solução

- Cada Escritor necessita de acesso exclusivo → semáforo exclusão mútua
  - nenhum processo, Leitor ou Escritor, pode entrar na sua região crítica
- O primeiro Leitor obtém acesso para leitura

→ pede exclusão mútua

  - não pode deixar "entrar" Escritores
- Os seguintes Leitores verificam só se já existe algum Leitor → não pedem exclusão mútua

# Esquema de uma solução

- Cada Escritor necessita de acesso exclusivo → semáforo exclusão mútua
  - nenhum processo, Leitor ou Escritor, pode entrar na sua região crítica
- O primeiro Leitor obtém acesso para leitura

→ pede exclusão mútua

  - não pode deixar "entrar" Escritores
- Os seguintes Leitores verificam só se já existe algum Leitor → não pedem exclusão mútua
- O último Leitor a sair liberta o acesso de leitura

→ liberta a exclusão mútua

  - o último Leitor deixa que "entrem" Escritores

R – contador do número de leitores (inicial: 0)
Wsem – semáforo de exclusão mútua para os escritores (incial: 1)

*Leitor:*

```
R=R+1;
if (R==1) P(Wsem);   (1)

Ler( );

R=R-1;
if (R==0) V(Wsem);   (2)
```

*Escritor:*

```
P(Wsem);   (3)

Escrever();

V(Wsem);   (4)
```

R – contador do núm. de leitores (inicial: 0)
Wsem – semáforo de exclusão mútua para os escritores (incial: 1)

*Leitor:*

```
P(exmut);
R=R+1;
if (R==1) P(Wsem);   (1)
V(exmut);
Ler( );
P(exmut);
R=R-1;
if (R==0) V(Wsem);   (2)
V(exmut);
```

*Escritor:*

```
P(Wsem);   (3)
Escrever();
V(Wsem);   (4)
```

*exmut – semáforo de exclusão mútua para os leitores porque a alteração da variável R é uma região crítica!*

variaveis globais partilhadas

```
var R: integer; inicialmente 0
    exmut, Wsem: semaphore. inicial 1
```

Processo Leitor Li

```
WHILE true DO
BEGIN
    P(exmut);
    R := R + 1;
    IF R=1 THEN P(Wsem);
    V(exmut);
    leitura;
    P(exmut);
    R := R – 1;
    IF R=0 THEN V(Wsem);
    V(exmut);
END;
```

Processo Escritor Ei

```
WHILE true DO
BEGIN
    P(Wsem);
    escrita;
    V(Wsem);
END;
```

# Problemas desta solução

- **Enquanto existirem leitores a ler, outros leitores entram sempre**
  - **os leitores passam à frente dos escritores**
  - **os escritores não têm hipótese de escrever**

- ***solução?***
  - ***quando chega um escritor, este impede que mais leitores entrem***

***Trabalho para Casa***

# 9 – Reservar/Libertar unidades de recursos

- **Certos recursos devem ser geridos dinamicamente: por exemplo, os *Buffers* de memória para as operações de entrada e saída**
- **Num momento, um processo precisa de guardar dados num *buffer*, mas depois de os tratar, pode libertar o *buffer*, para outros processos o utilizarem**
- **Conceito de Reservatório ("*Pool*") dinâmico de recursos, disponível a Processos concorrentes**

- Quando um processo precisa de um certo número (n) de unidades de um recurso:

-- invoca a operação **Reservar(n)**

- Que opções de definição de Reservar()?

- Quando um processo precisa de um certo número (n) de unidades de um recurso:

-- invoca a operação Reservar(n)

- Que opções de definição de Reservar()?

a) Se não houver n unidades disponíveis, devolve de imediato um indicador de erro e o processo invocador tenta mais tarde.

- Quando um processo precisa de um certo número (n) de unidades de um recurso:

-- invoca a operação Reservar(n)

- Que opções de definição de Reservar()?

a) Se não houver n unidades disponíveis, devolve de imediato um indicador de erro e o processo invocador tenta mais tarde.

b) Se não houver n unidades disponíveis, o processo invocador fica bloqueado até que as haja (quando algum outro Processo Libertar() unidades…)

- Quando um processo já não precisa mais das n unidades, invoca Libertar(n)

# Múltiplas unidades de recursos

- **Não se pode deixar cada processo reservar recursos um a um. Exemplo:**
  - **existem 3 unidades R disponíveis**
  - **P1 quer 2: Reservar(R);Reservar(R);**
  - **P2 quer 3: Reservar(R);Reservar(R);Reservar(R)**
  - **se P1 obtiver 1 e P2 obtiver 2, podemos ter um *deadlock***

**Cada processo: obtém todas ou aguarda que todas estejam disponíveis**

- **operação reservar(N) : deve ser atómica**

# Como realizar Reservar/Libertar?

**Reservar/Libertar podem ser chamadas concorrentemente, por diferentes processos.**

**a) Realizar Reservar/Libertar, sem um Servidor intermédio: os processos invocam directamente operações que verificam se podem reservar ou não, ou que libertam unidades.**

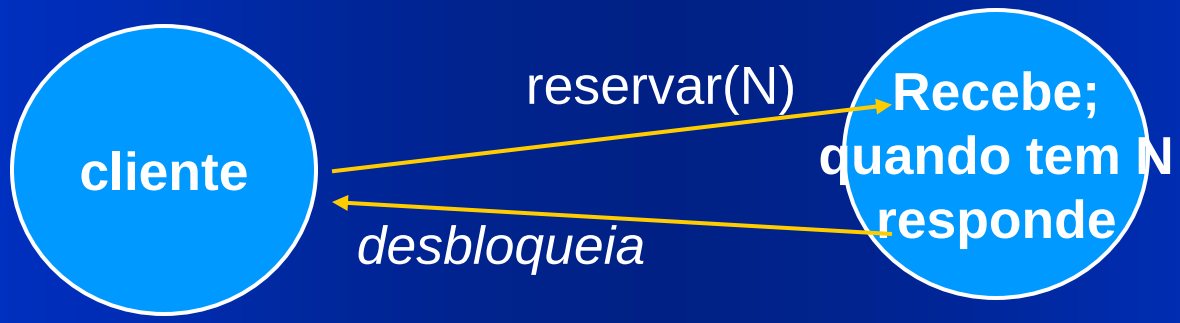
# Como realizar Reservar/Libertar?

**Reservar/Libertar podem ser chamadas concorrentemente, por diferentes processos.**

**1) Realizar Reservar/Libertar, sem um Servidor intermédio: os processos invocam directamente operações que verificam se podem reservar ou não, ou que libertam unidades.**

**2) Impôr um tratamento sequencial dos pedidos, através de um Servidor que trata os pedidos de Reservar e Libertar.**

# 2- Com um Servidor




***Trabalho para casa!***

# 1 - Sem um Servidor:

Em memória partilhada têm-se variáveis que indicam o estado corrente do reservatório:

--- um inteiro ‘total_disponivel’: número de unidades livres

--- semáforos de sincronização:

- exclusão mútua no teste de total_disponivel (exmut, inicialmente a 1)
- bloqueio de processos que invoquem Reservar: quando não há n unidades livres (bloq, inicialmente a 0)

```
Reservar(n):
{
  IF n>total_disponivel THEN
     {
        .......
     }
 total_disponivel = total_disponivel – n;
}
Libertar(n):
{
   total_disponivel = total_disponivel + n;
   ......
}
```

```
Reservar(n):
 { P(exmut);
   IF n>total_disponivel THEN
       {
           ......
       }
  total_disponivel = total_disponivel – n;
 V(exmut);
 }
Libertar(n):
 { P(exmut);
    total_disponivel = total_disponivel + n;
   V(exmut);
 }
```

```
Reservar(n):
 { P(exmut);
   IF n>total_disponivel THEN
       {
          P(bloq);
       }
  total_disponivel = total_disponivel – n;
 V(exmut);
 }
Libertar(n):
 { P(exmut);
    total_disponivel = total_disponivel + n;
    V(exmut);
 }
```

```
Reservar(n):
 { P(exmut);
   IF n>total_disponivel THEN
       {   V(exmut);
           P(bloq);
       }
  total_disponivel = total_disponivel – n;
 V(exmut);
 }
Libertar(n):
 { P(exmut);
    total_disponivel = total_disponivel + n;
    V(exmut);
 }
```

```
Reservar(n):
 { P(exmut);
   IF n>total_disponivel THEN
       {   V(exmut);
           P(bloq);
       }
  total_disponivel = total_disponivel – n;
 V(exmut);
 }
Libertar(n):
 { P(exmut);
    total_disponivel = total_disponivel + n;
    V(bloq);
    V(exmut);
 }
```

```
Reservar(n):                                  cont inicial 0
 { P(exmut);
   IF n>total_disponivel THEN
       {  cont = cont + 1;
          V(exmut);
           P(bloq);
       }
  total_disponivel = total_disponivel – n;
 V(exmut)}
Libertar(n):
 { P(exmut);
    total_disponivel = total_disponivel + n;
    WHILE cont > 0 DO { cont = cont-1; V(bloq)}
    V(exmut)}
```

```
Reservar(n):                                  cont inicial 0
 { P(exmut);
   WHILE n>total_disponivel  DO
       {  cont = cont + 1;
          V(exmut);
           P(bloq);
       }
  total_disponivel = total_disponivel – n;
 V(exmut)}
Libertar(n):
 { P(exmut);
    total_disponivel = total_disponivel + n;
    WHILE cont > 0 DO { cont = cont-1; V(bloq)}
    V(exmut)}
```

**Reservar(n):** cont inicial 0

```
{ P(exmut);
  WHILE n>total_disponivel DO
     {  cont = cont + 1;
        V(exmut);
         P(bloq);   P(exmut)
      }
 total_disponivel = total_disponivel – n;
V(exmut)}
```

**Libertar(n):**

```
{ P(exmut);
   total_disponivel = total_disponivel + n;
   WHILE cont > 0 DO { cont = cont-1; V(bloq)}
   V(exmut)}
```

# 10- Reserva de múltiplos recursos

- 2 processos P1 e P2 reservam 2 recursos R1 e R2, para acesso exclusivo

**P1**

```
P(semR1);
P(semR2);
Acede a R1 e R2;
V(semR1);
V(semR2);
```

**P2**

```
P(semR2);
P(semR1);
Acede a R1 e R2;
V(semR2);
V(semR1);
```

semR1 e semR2 inicialmente 1

- **Um cenário possível:**

**P1 faz P(semR1) e semR1 fica 0;**

**P2 faz P(semR2) e semR2 fica 0;**

**P1 tenta P(semR2) e bloqueia-se;**

**P2 tentaP(semR1) e bloqueia-se.**

**DEADLOCK**

- **Outro cenário possível:**

**P1 faz P(semR1) e semR1 fica 0;**

**P1 faz P(semR2) e semR2 fica 0;**

**P2 tenta P(semR2) e bloqueia-se;**

**Quando P1 acaba, faz V(semR1) e V(semR2), e P2 é desbloqueado e pode prosseguir**

- Qual dos cenários ocorre?

-- qualquer deles pode ocorrer…

- Qual é o problema?

-- se quero reservar dois recursos para acesso exclusivo, tenho de o fazer com uma acção atómica:

-- as sequências

P(semR1); P(semR2);

ou P(semR2); P(semR1);

não são atómicas.

# Soluções para o problema

1. **Obter acesso aos recursos de forma atómica -- uma operação P(sem1,sem2) como no Unix System V (sobre arrays de semáforos)**
2. **Forçar acesso aos recursos R1 e R2 pela mesma ordem em todos os processos**
3. **Deixar ocorrer DEADLOCKS e ter um processo que os detecte:**

**-- por construção de um GRAFO de ESPERAS**

**e detecção de um CICLO:**

**P1 espera-por P2**

**P2 espera-por P1 → e matar um dos processos**

# 11- Sincronização mútua (*rendez-vous*)

- **Processos 1 e 2 sincronizam-se mutuamente**

**o primeiro a chegar, aguarda pelo outro**

- **dupla sinalização**
- **dois semáforos: s1 e s2 (inicial: 0)**

**Proc1:** V(s2); P(s1)

**Proc2:** V(s1); P(s2)

# 12- Barreira de sincronização

- **N processos sincronizam-se num *rendez-vous* colectivo:**

**Cada processo:**

```
....
chegada()   /* espera por todos os outros*/
....        /* prossegue, após todos chegarem*/
```

# N-Barreira: baseada em semáforos

Variáveis privadas de cada instância da barreira

```
count: integer;   /* inicialmente 0*/
    exmut: semaphore; /* inicial/ 1 */
    block: semaphore; /*inicial/ 0*/
chegada()
{ P(exmut);
  count++;
  if (count<N) then
                  {V(exmut); P(block); P(exmut)};
  count--;
  if (count > 0) then V(block);
  V(exmut);
}
```

# N-Barreira: baseada em semáforos

```
var count: integer;   /* inicialmente 0*/
    exmut: semaphore; /* inicial/ 1 */
    block: semaphore; /*inicial/ 0*/
chegada() {
  int i;
  P(exmut);
   count++;
  if (count<N) then
                    {V(exmut); P(block); P(exmut)};
  else { for (i=1; i<N; i++) V(bloq);
          count = 0;}
  V(exmut);
}
```

# 13- Uma ponte com uma única faixa mas com tráfego nos dois sentidos

**Quando um carro chega a uma entrada:**

**-- só pode entrar se uma das condições for satisfeita:**

- **A ponte está desocupada**

**ou**

- **O tráfego corrente desloca-se no mesmo sentido que o do carro que acabou de chegar**

```
var contE, contD: int;                    -- inicial/ 0
    exE, exD, block: semaphore; -- inicial/ 1
```

**Um carro chega:**

**-- se é o primeiro de um lado, deve ver se a ponte está ocupada no sentido contrário:**

**-- se estiver, deve esperar**

**-- senão, deve entrar na ponte**

**-- se não é o primeiro: pode entrar, pois o tráfego está no seu sentido…**

**(-- admite-se que a ponte nunca enche...)**

E:

```
P(exE);
contE++;
if (contE==1)
    then P(block);
V(exE);
na ponte();
P(exE);
contE--;
if (contE==0)
   then V(block);
V(exE);
```

D:

```
P(exD);
contD++;
if (contD==1)
    then P(block);
V(exD);
na ponte();
P(exD);
contD--;
if (contD==0)
   then V(block);
V(exD);
```

# Conclusões sobre os semáforos

- **Mecanismo ´´universal´´: resolve quase todos os problemas de sincronização**
- **Permite resolver os principais padrões de interacção entre processos:**

**-- regiões críticas (exclusão mútua)**

**-- produtores / consumidores, via Buffers**

**-- simples troca de sinais de sincronização**

**-- controlo da activação de processos**

**-- leitores / escritores de bases de dados**

**-- clientes / servidores, via filas de pedidos**

**-- reserva / libertação de recursos**

- Não controla a ordem de execução dos processos, sob o controlo de um Sistema de Operação
- Só aplicável quando há Memória Partilhada, isto é, não aplicável em redes de computadores
- Só admite 3 operações: iniciar/P/V
- Exige operação explícita P() para um processo aguardar um sinal de sincronização de outro
- A operação P() bloqueia se S não for positivo
- O valor do semáforo não pode ser testado sem ser através de P() (Nota: o Unix System V permite isso, mas não se deve utilizar! ( Dijkstra *dixit* )

- É um mecanismo muito elementar e não estruturado:

-- um programa concorrente baseado em semáforos pode-se tornar difícil de compreender

-- a omissão de um P() ou de um V() torna o programa incorrecto e o erro pode ser difícil de detectar, pois pode haver, dispersos nos programas dos processos, múltiplos P() e V() sobre um mesmo semáforo

-- a utilização conjunta de fork/exec/semáforos torna os programas concorrentes difíceis de compreender