Programação Concorrente

José C.Cunha, DI-FCT/UNL

- Encadeamento sequencial de actividades
- Processos concorrentes
- Estados de um processo
- Padrões de interacções entre processos

Encadeamento sequencial

-- Ler; Tratar; Imprimir -- SEQUENCIAL

versus

-- Ler// Tratar// Imprimir -- CONCORRENTE

Desvantagens da execução sequencial

1. Maior tempo de execução
2. Má utilização do CPU e periféricos
3. Pior ´modularidade´

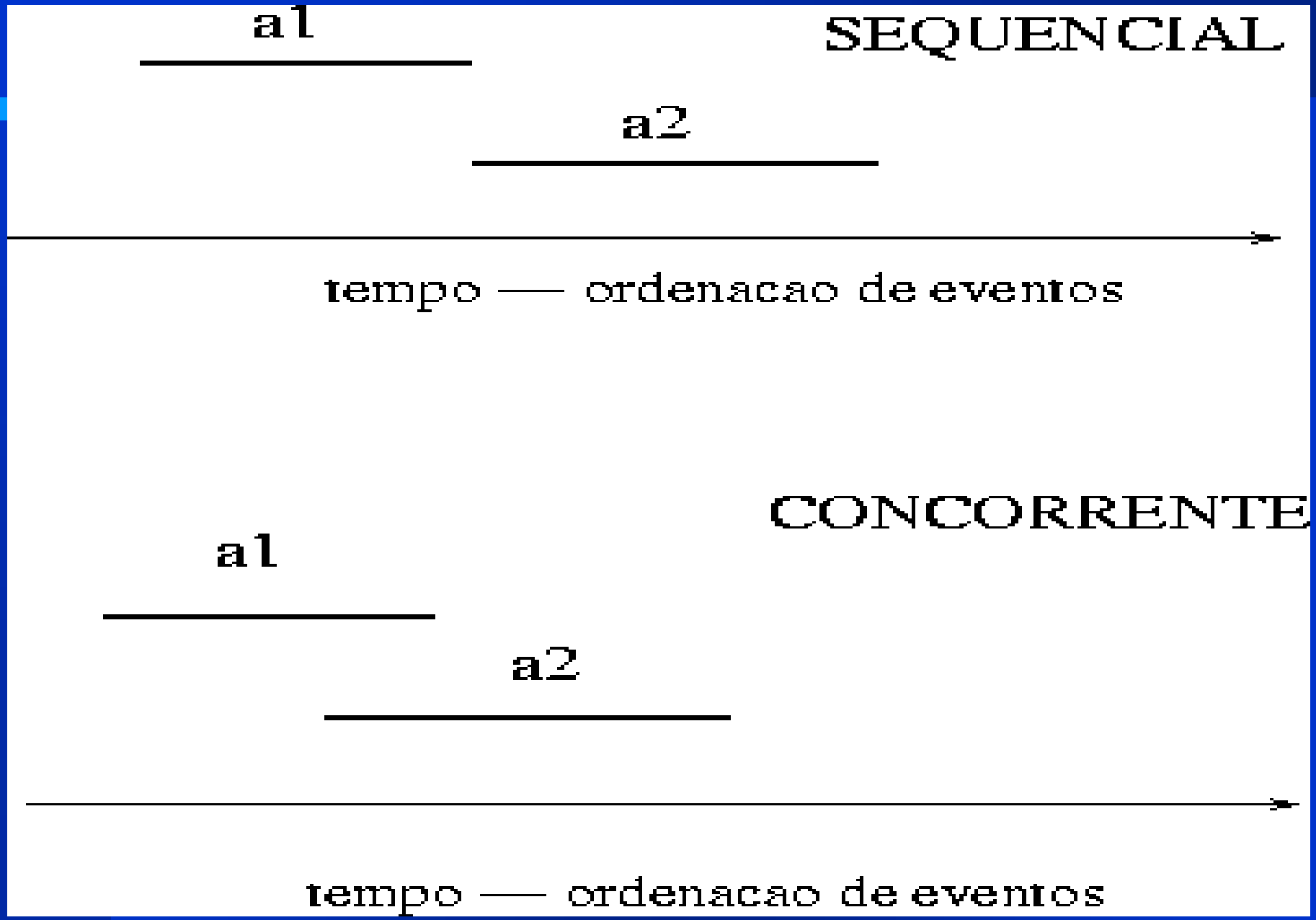
Processos concorrentes

1. **Múltiplas aplicações independentes**
2. **Múltiplos processos numa aplicação individual**

processo Ler // processo Tratar // processo Imprimir

Processo: mecanismo para encapsular a execução sequencial de um programa

Concorrência: dados 2 processos P e Q, uma vez iniciado P, poder iniciar Q, sem ter de esperar pelo fim de P, ou vice-versa.


a1
SEQUENCIAL
a2
tempo — ordenacao de eventos
CONCORRENTE
a1
a2
tempo — ordenacao de eventos

Concorrência através de multiprogramação

- **O SO controla a execução de múltiplos programas, de modo a poder ir alternando de uns para outros, em momentos oportunos:**

Concorrência através de multiprogramação

- **O SO controla a execução de múltiplos programas, de modo a poder ir alternando de uns para outros, em momentos oportunos:**

-- quando um processo se bloqueia em READ por um buffer estar vazio

Concorrência através de multiprogramação

- **O SO controla a execução de múltiplos programas, de modo a poder ir alternando de uns para outros, em momentos oportunos:**

-- quando um processo se bloqueia em READ por um buffer estar vazio

-- quando um processo faz WAIT por 1 filho que ainda não terminou

Concorrência através de multiprogramação

- **O SO controla a execução de múltiplos programas, de modo a poder ir alternando de uns para outros, em momentos oportunos:**

-- quando um processo se bloqueia em READ por um buffer estar vazio

-- quando um processo faz WAIT por 1 filho que ainda não terminou

-- quando um processo expira o tempo de CPU a que tinha direito (TIME-SLICE)

-- etc.

Concorrência através de multiprogramação

- **O SO controla a execução de múltiplos programas, de modo a poder ir alternando de uns para outros, em momentos oportunos:**

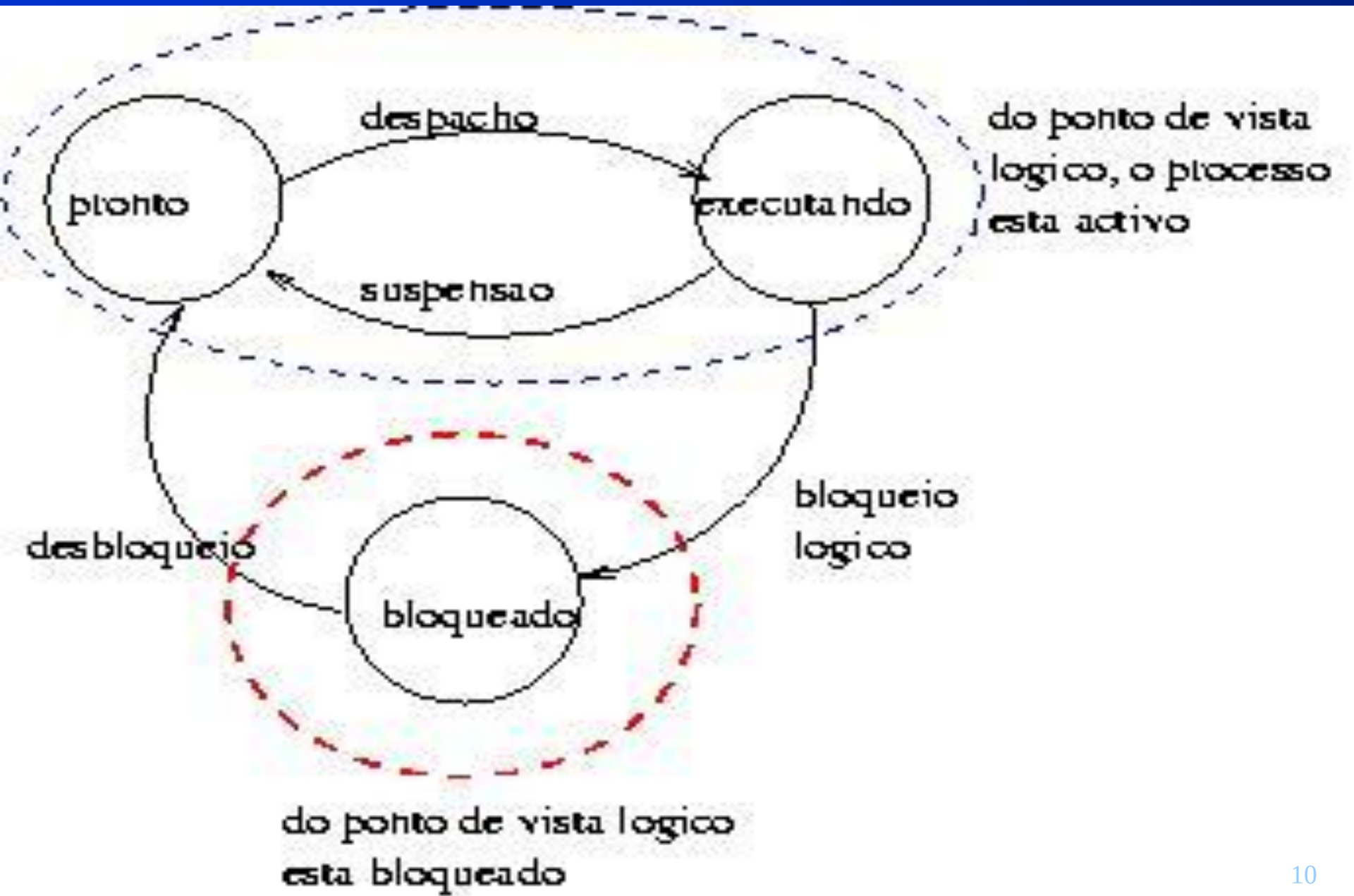
-- quando um processo se bloqueia em READ por um buffer estar vazio

-- quando um processo faz WAIT por 1 filho que ainda não terminou

-- quando um processo expira o tempo de CPU a que tinha direito (TIME-SLICE)

-- etc.

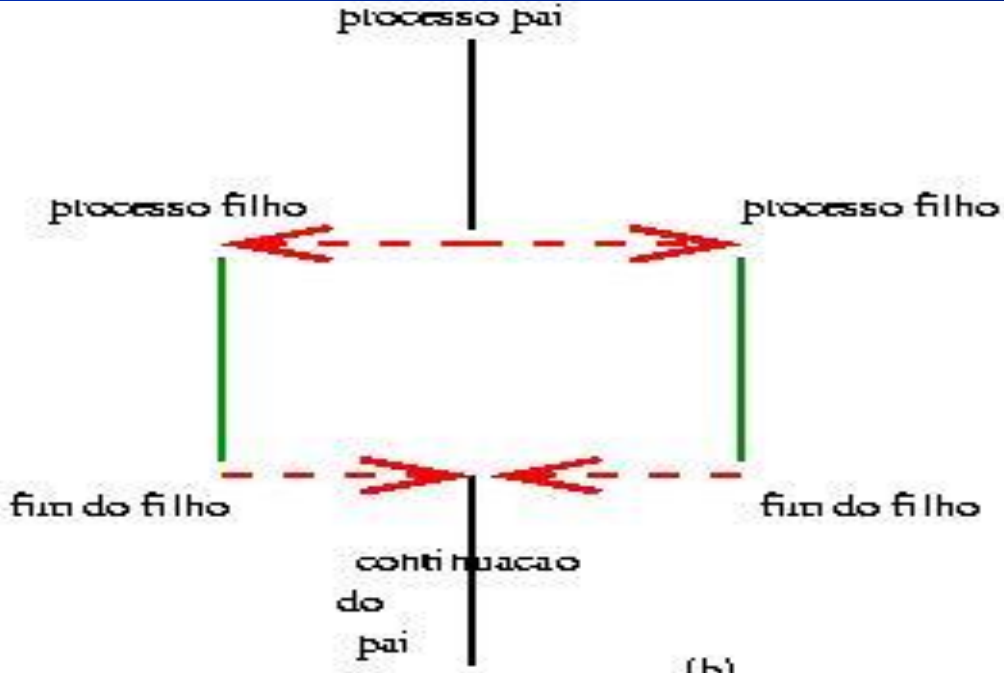
Estados de um processo no SO



Especificar processos concorrentes

(a)

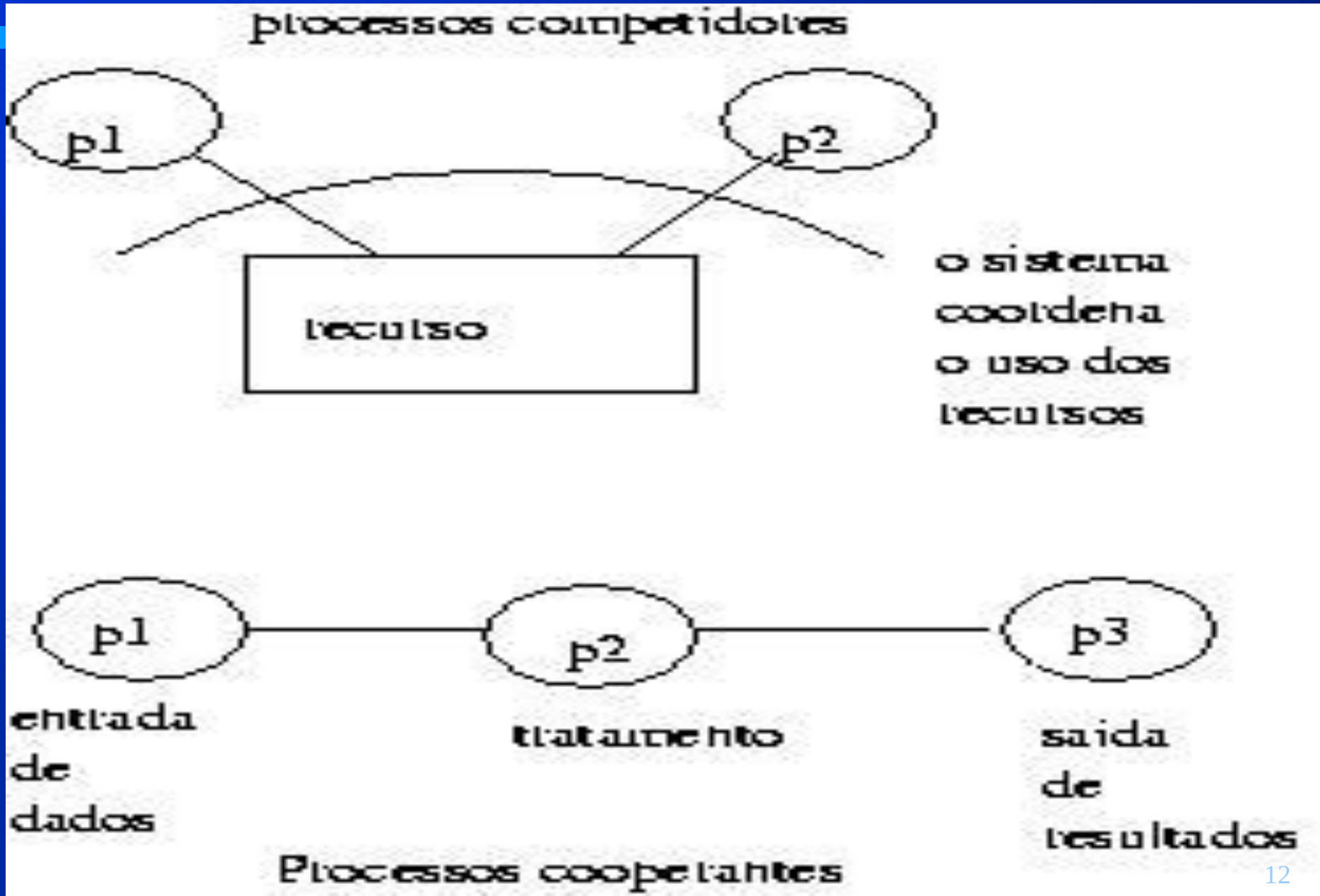
```
programa          (Unix)
.....
....

p1 = fork();

if (p1 == 0)
  | codigo do filho) ];
codigo do pai
...
...

wait();
```



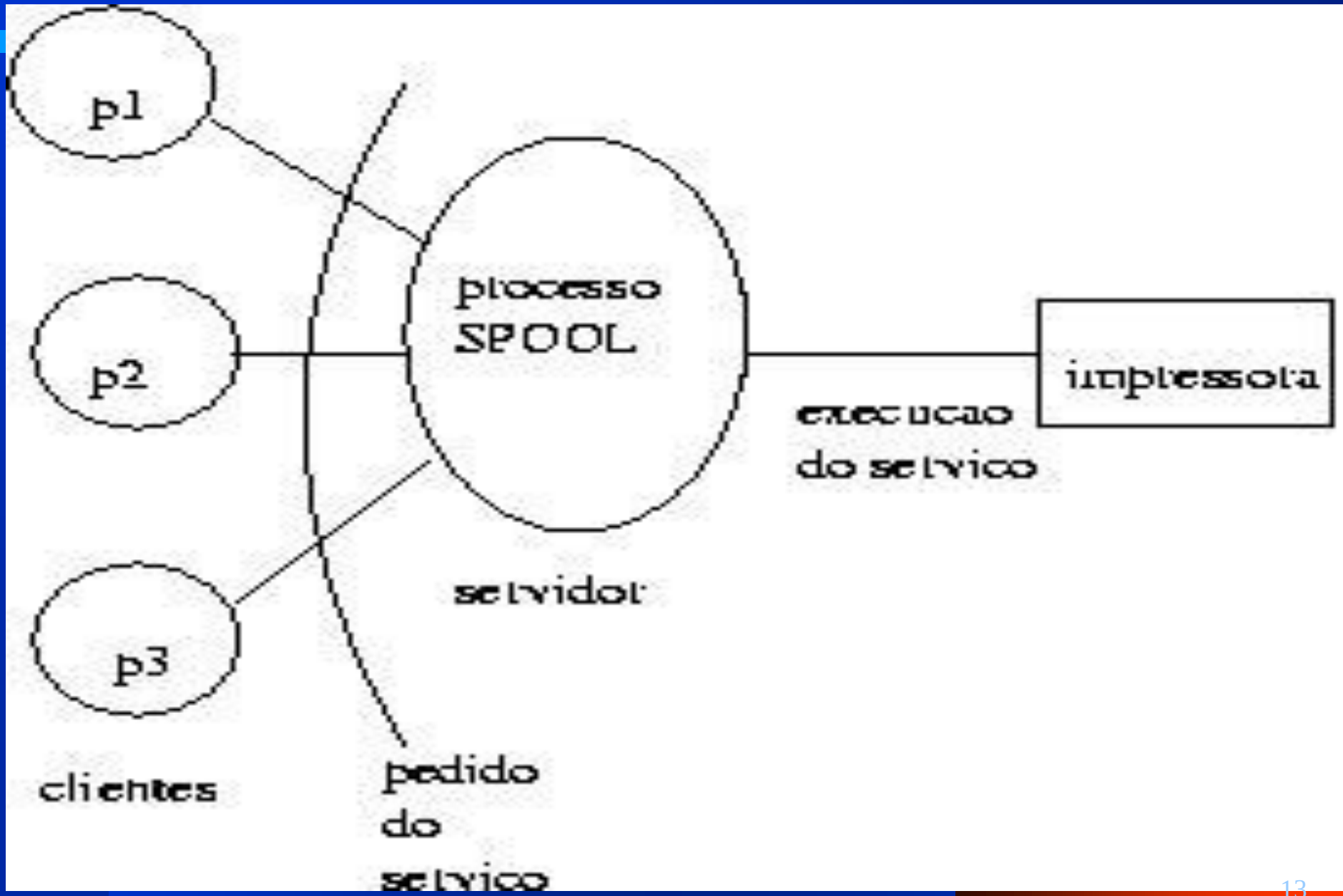
(b)

```
programa          (Linguagem
                  concorrente)
...
...
cobegin

  processo P1;
  processo P2

coend;

...
continuacao do pai;
```

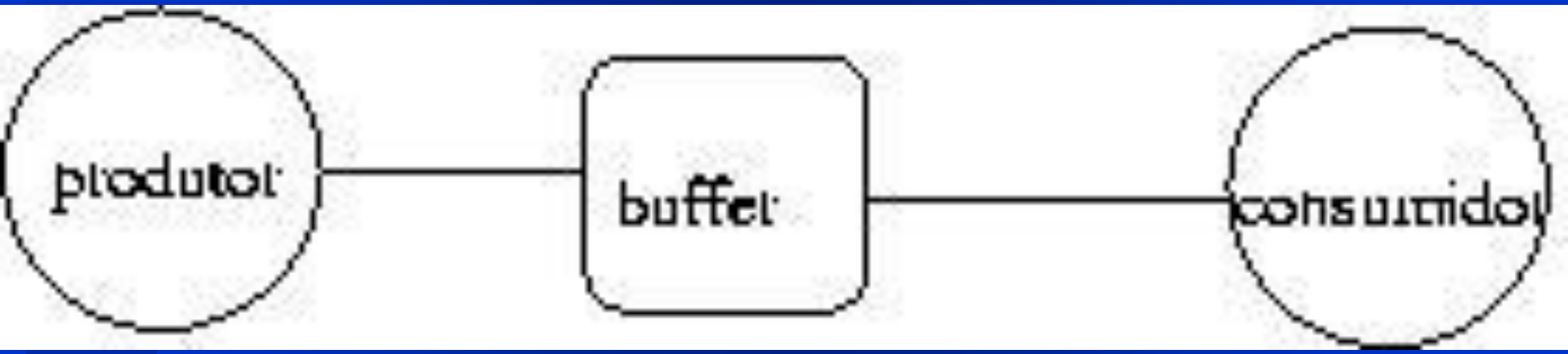
Interacções entre processos



Clientes e Servidores



Produtor-Consumidor



Modelos de comunicação

Modelos de comunicação...

Ficheiros em disco (geridos pelo SO)

comunicação ´´*a post-mortem*´´

versus

comunicação ´´*online*´´

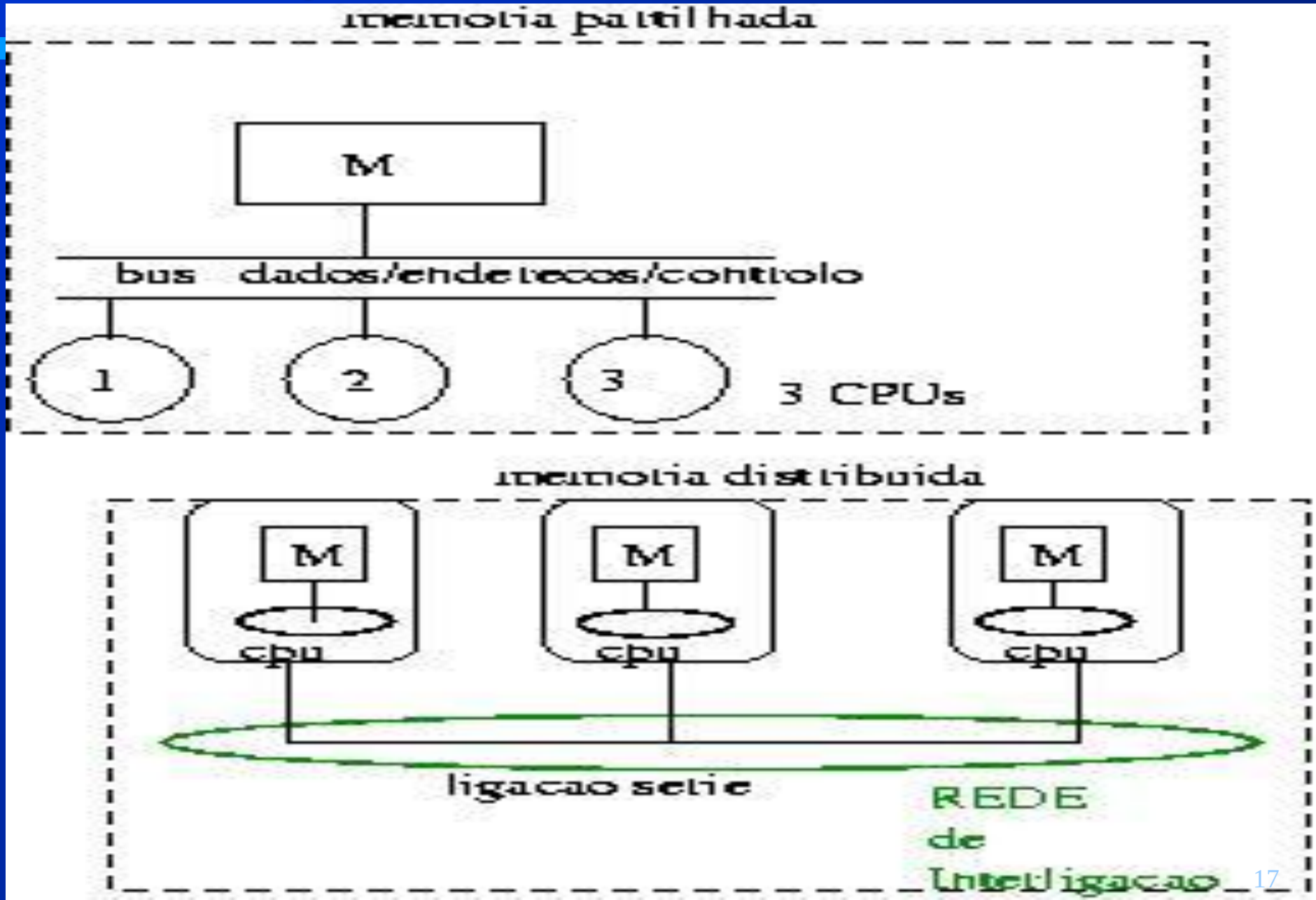
Pipes Unix *(buffers geridos pelo SO)* → ***Streaming***

Mensagens (filas geridas pelo SO) → **Mensagens**

Memória partilhada (acesso via *Bus*) → **Variáveis partilhadas**

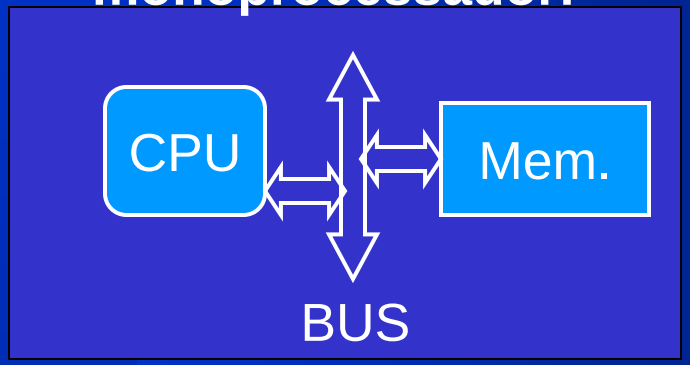
Troca de sinais de sincronização → **Eventos**

Arquitecturas Hardware

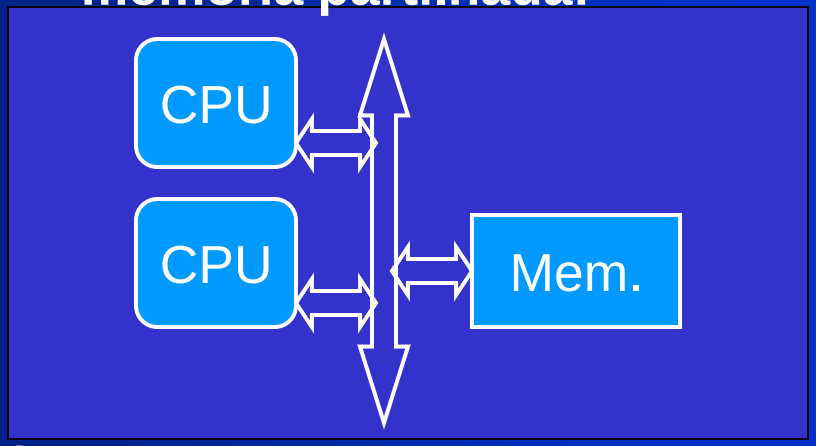


Arquitecturas *Hardware*

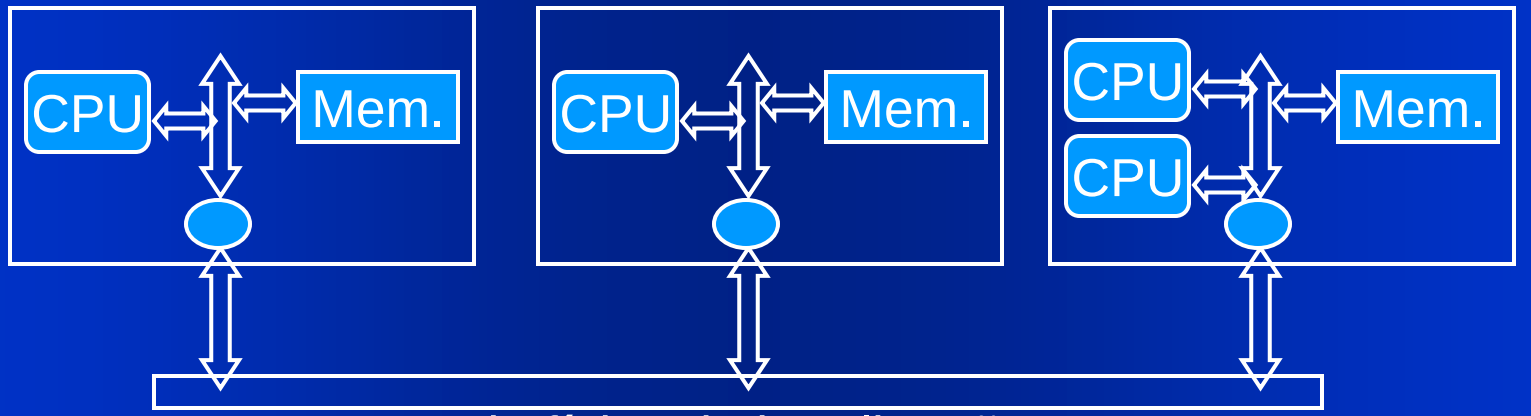
monoprocessador:



multiprocessador de memória partilhada:



multiprocessador de memória distribuída:



Modelos de comunicação

Por memória partilhada:

-- acesso a estruturas de dados comuns

-- sem envolver cópia de bytes

Por memória distribuída:

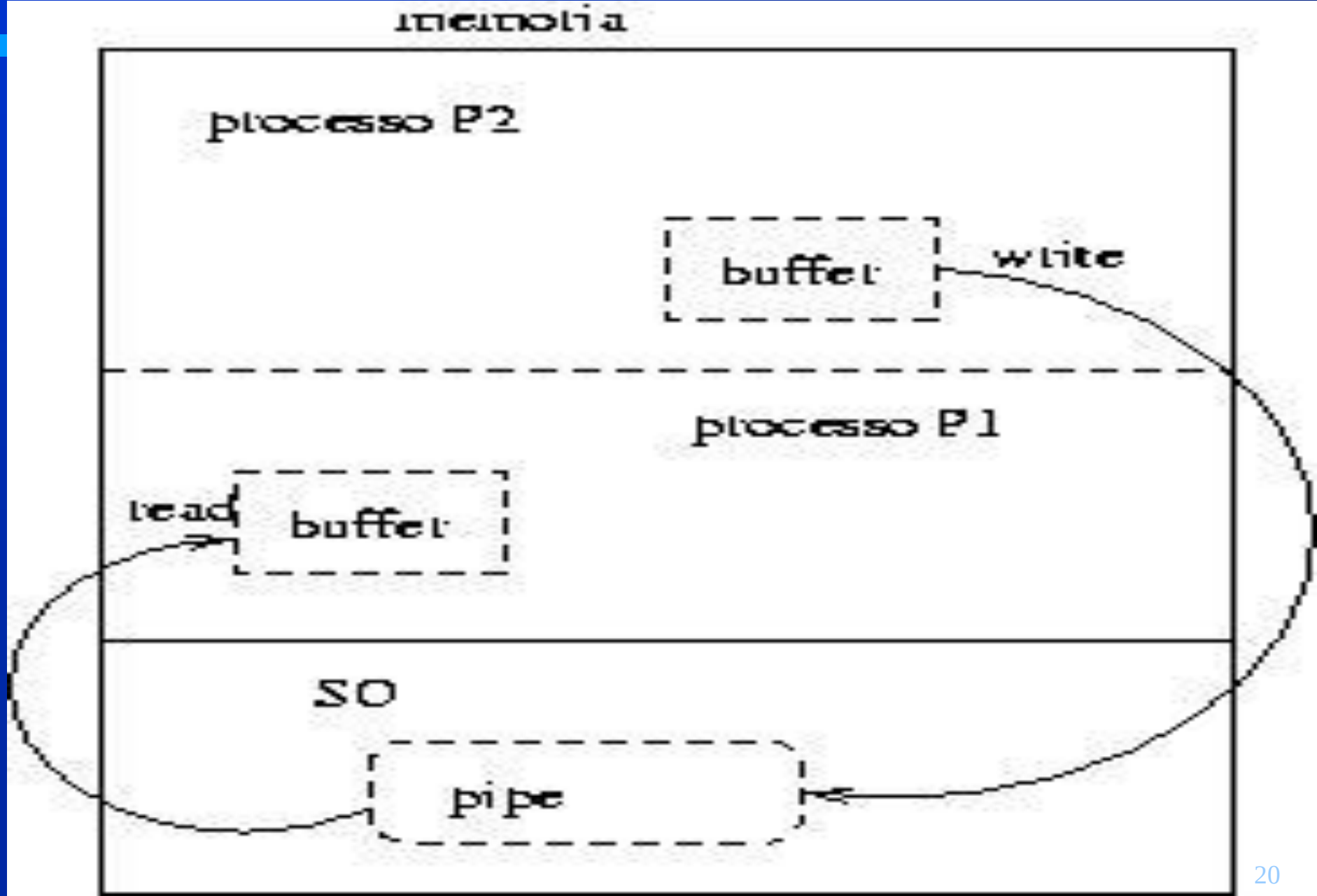
Há um meio de comunicação externo aos processos:

--- *pipe*

--- *mailbox* ou fila de mensagens

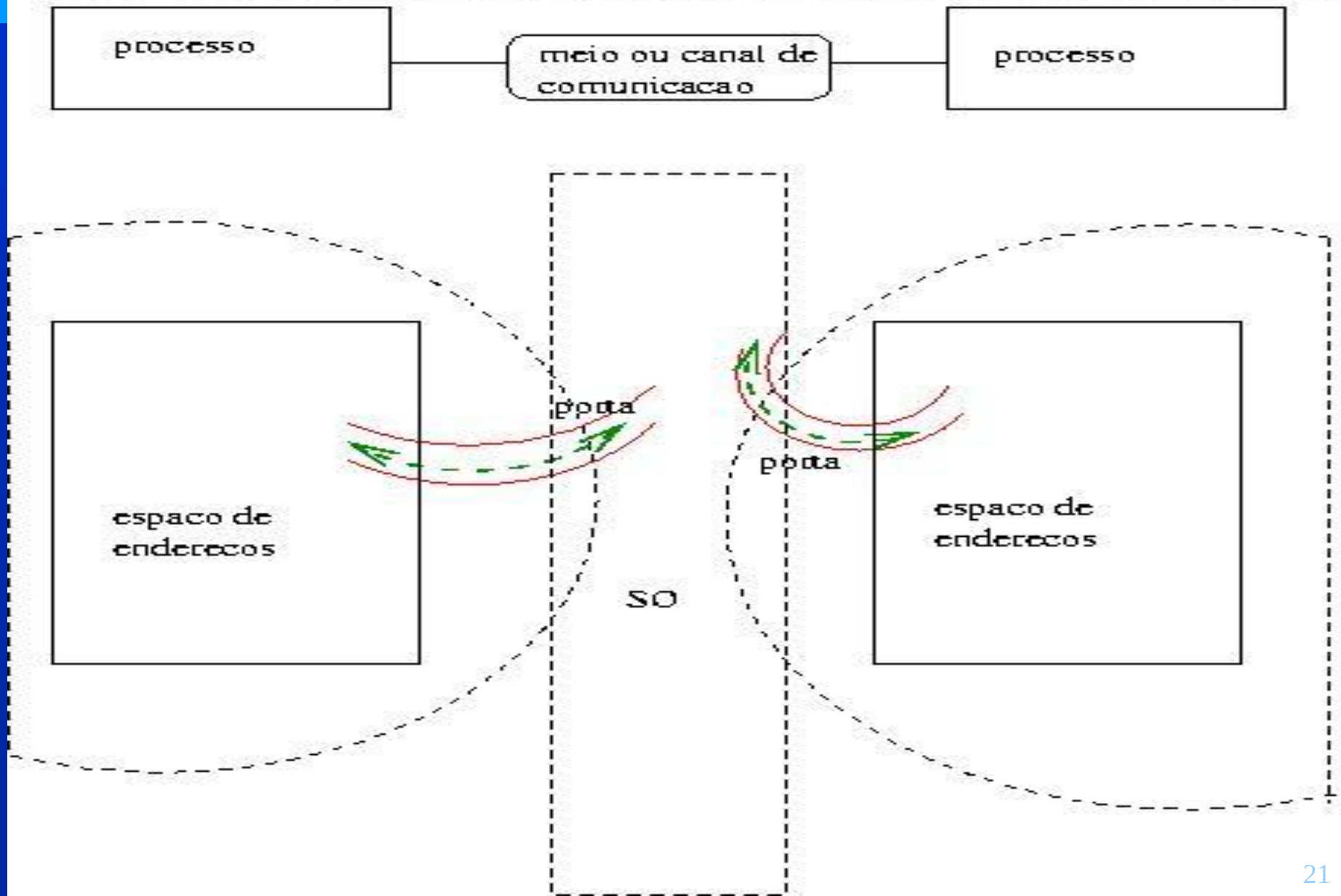
--- canais directos entre processos

Comunicação por *Pipes* Unix

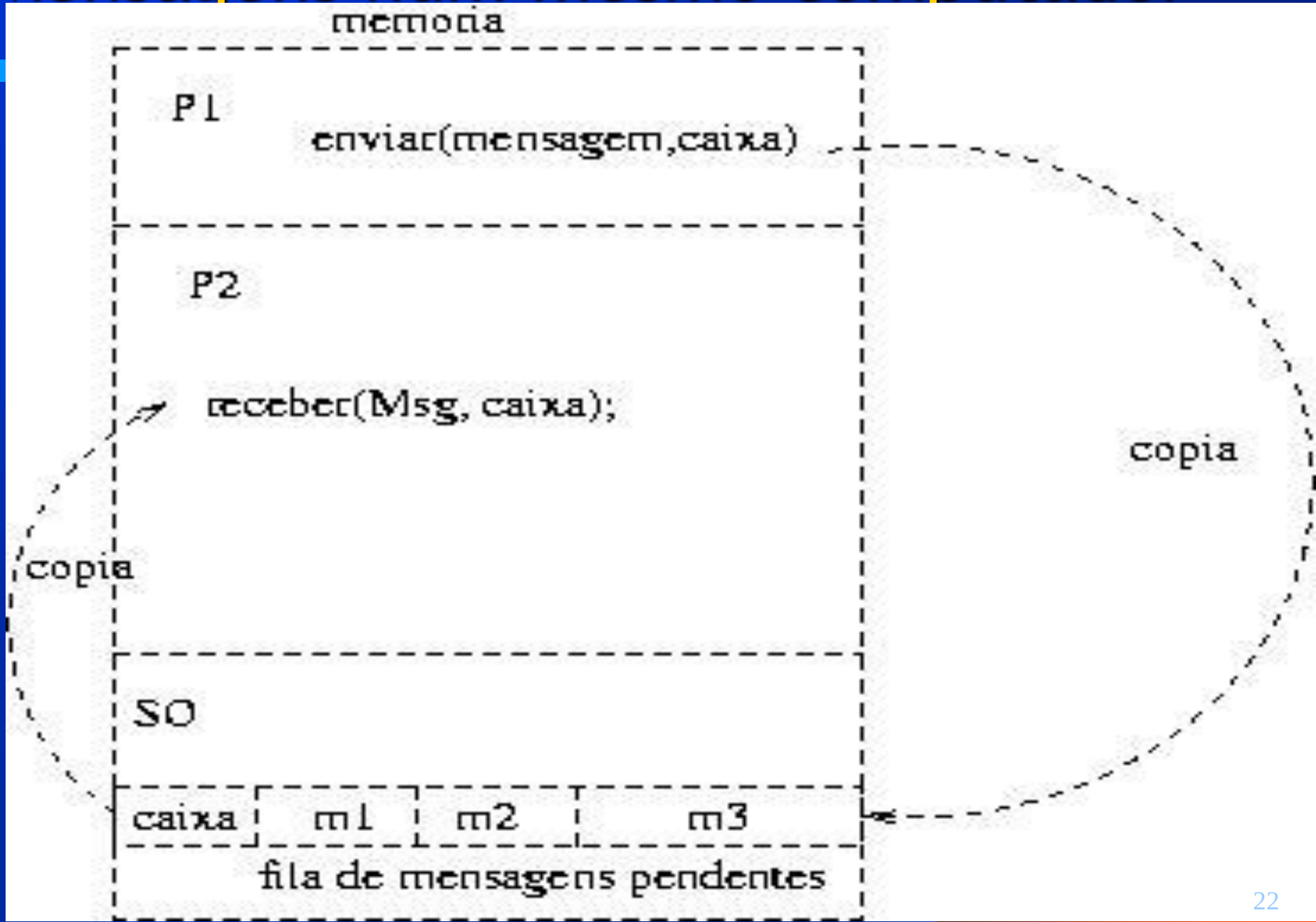


Mensagens

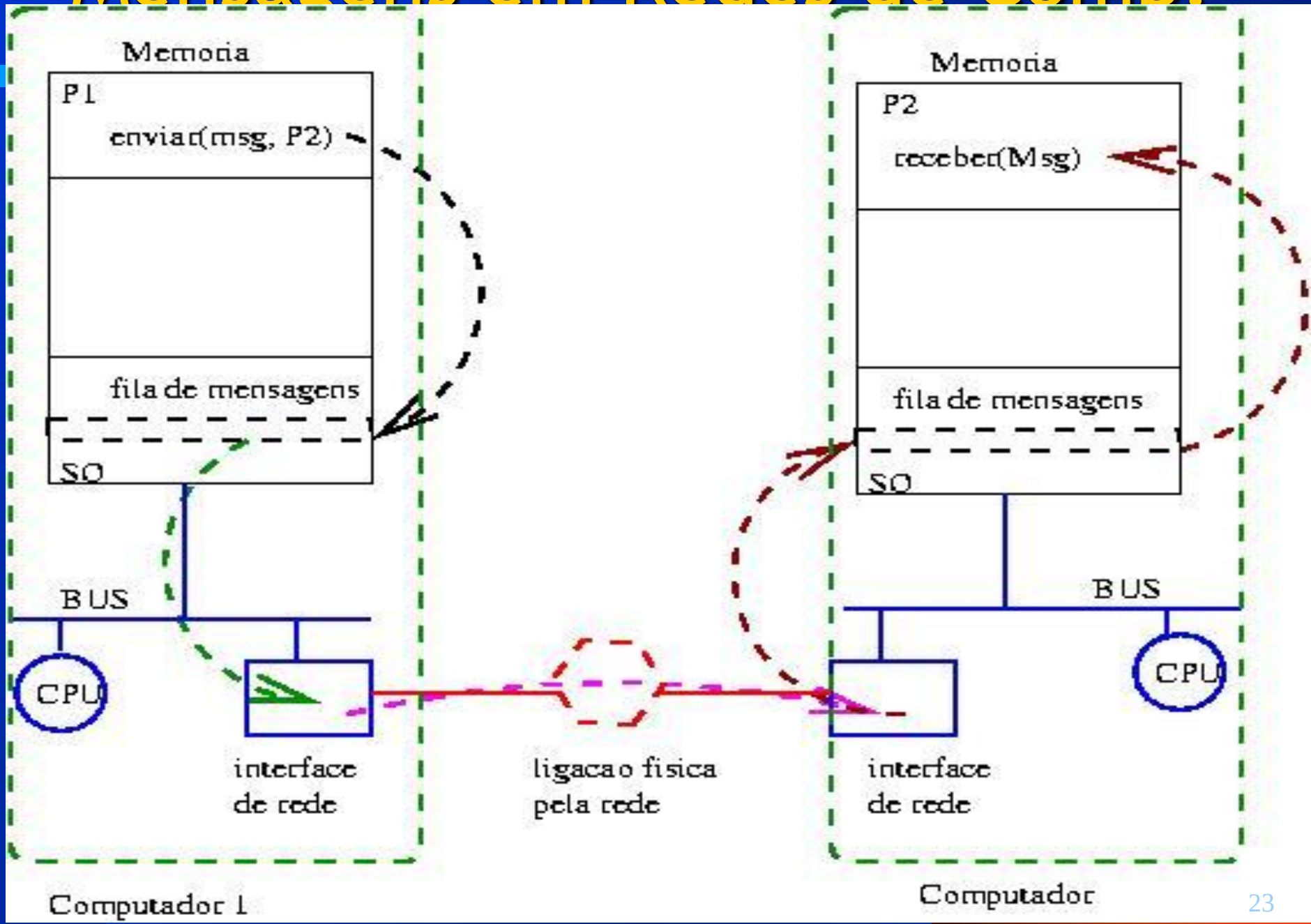
valores de um certo tipo sao recebidos ou enviados atraves do canal e transferidos de / para o espaco de enderecos privado de cada processo



Mensagens num mesmo computador



Mensagens em Redes de Comp.



Processos concorrentes

- **Quando comunicam por Memória Partilhada:**

- ler e escrever dados de memória

- sem haver cópia de *bytes* → muito rápida…

- exige um *Bus* comum de acesso a memória

- mas exige sincronização explícita pelo Programa

- **Quando comunicam por Mensagens**

- enviar e receber mensagens

- com cópia de *bytes* → mais lenta…

- com sincronização implícita pelo SO

- pode ser usado num único computador ou em rede

Modelos de comunicação

Por mensagens (memória distribuída):

- cópia de dados entre espaços de endereçamento (processos) diferentes
- um "meio" de comunicação externo
  - exemplo: *pipes*, filas de mensagens
- com sincronização implícita pelo SO
- leitura destrutiva (*bytes* do *stream* ou da mensagem desaparecem, após copiados para o destino)

Por memória partilhada… →

Memória partilhada

1- Entre múltiplos Processos distintos

- cada Processo executa um Programa num Espaço de Endereços Privado

→ Poderão ter regiões de memória comuns?

Memória partilhada

1- Entre múltiplos Processos distintos

- cada Processo executa um Programa num Espaço de Endereços Privado

→ ***Poderão ter regiões de memória comuns?***

(2- Dentro do mesmo Processo +adiante)

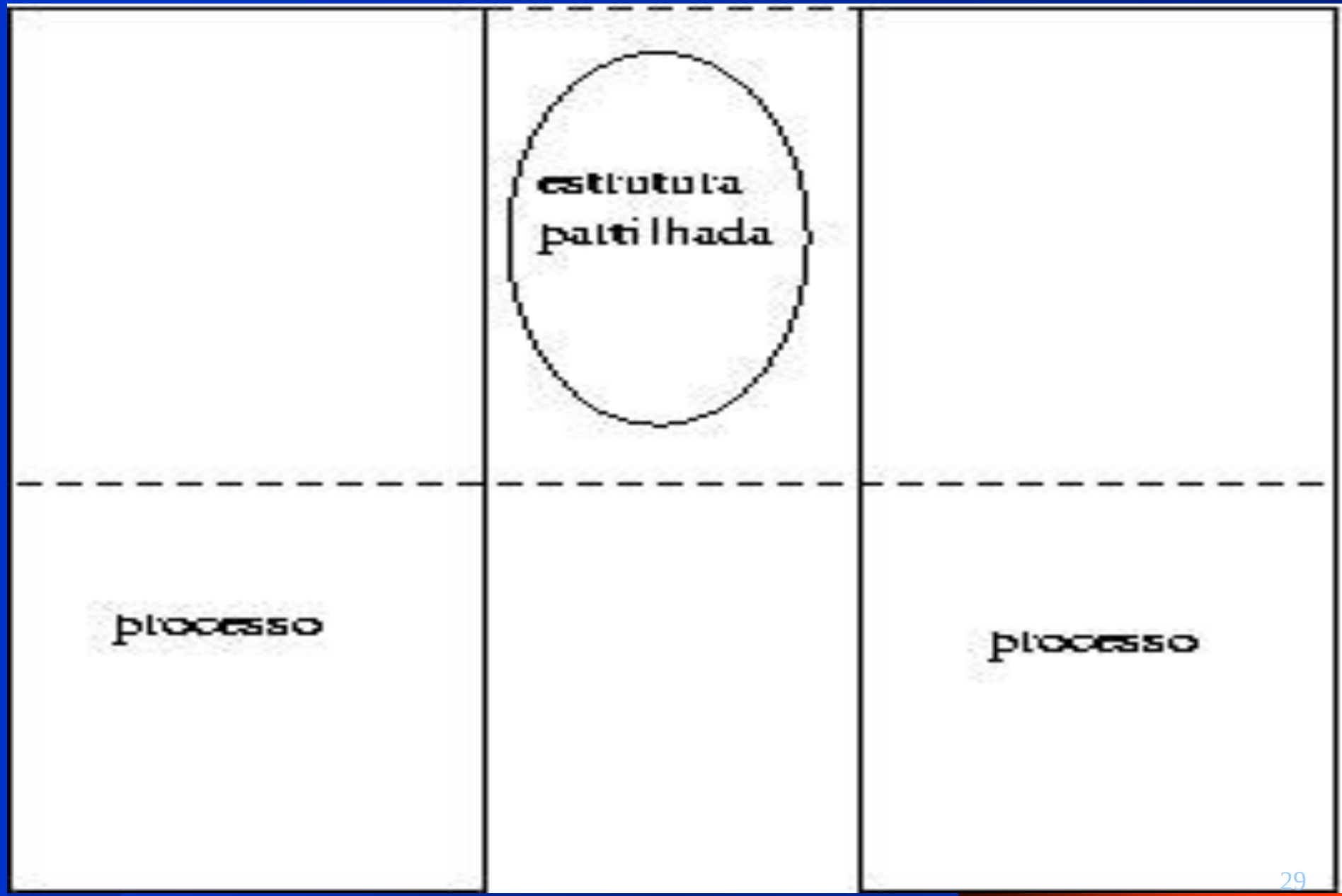
- cada *Thread* executa uma Função de um Programa, no mesmo Espaço dos outros *Threads desse Processo*

→ ***Naturalmente têm Variáveis e Dados comuns***

Por omissão, cada processo tem um MAPA de memória PRIVADO e PROTEGIDO

Como fazer partilhar a memória?

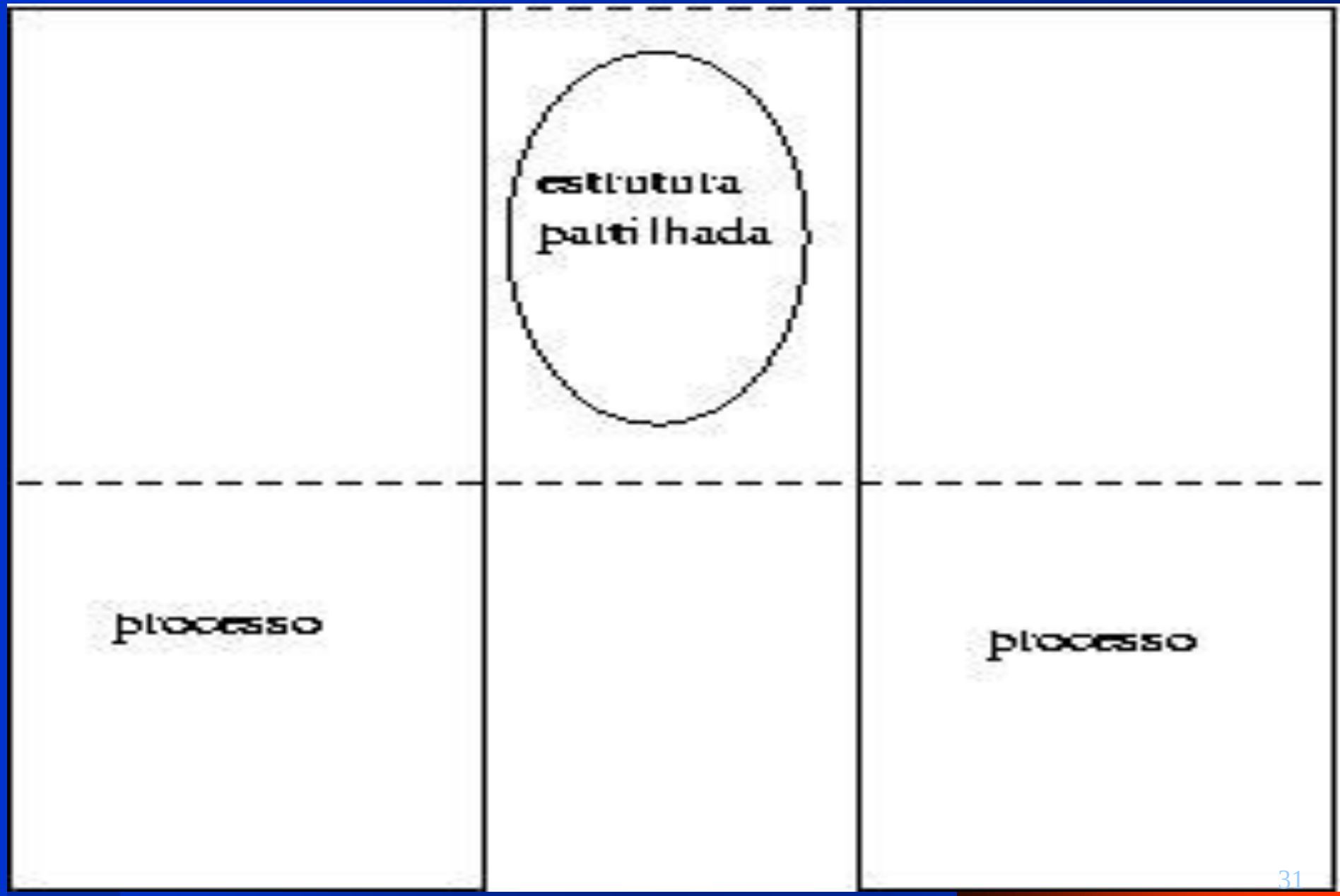
Memória Partilhada entre Processos



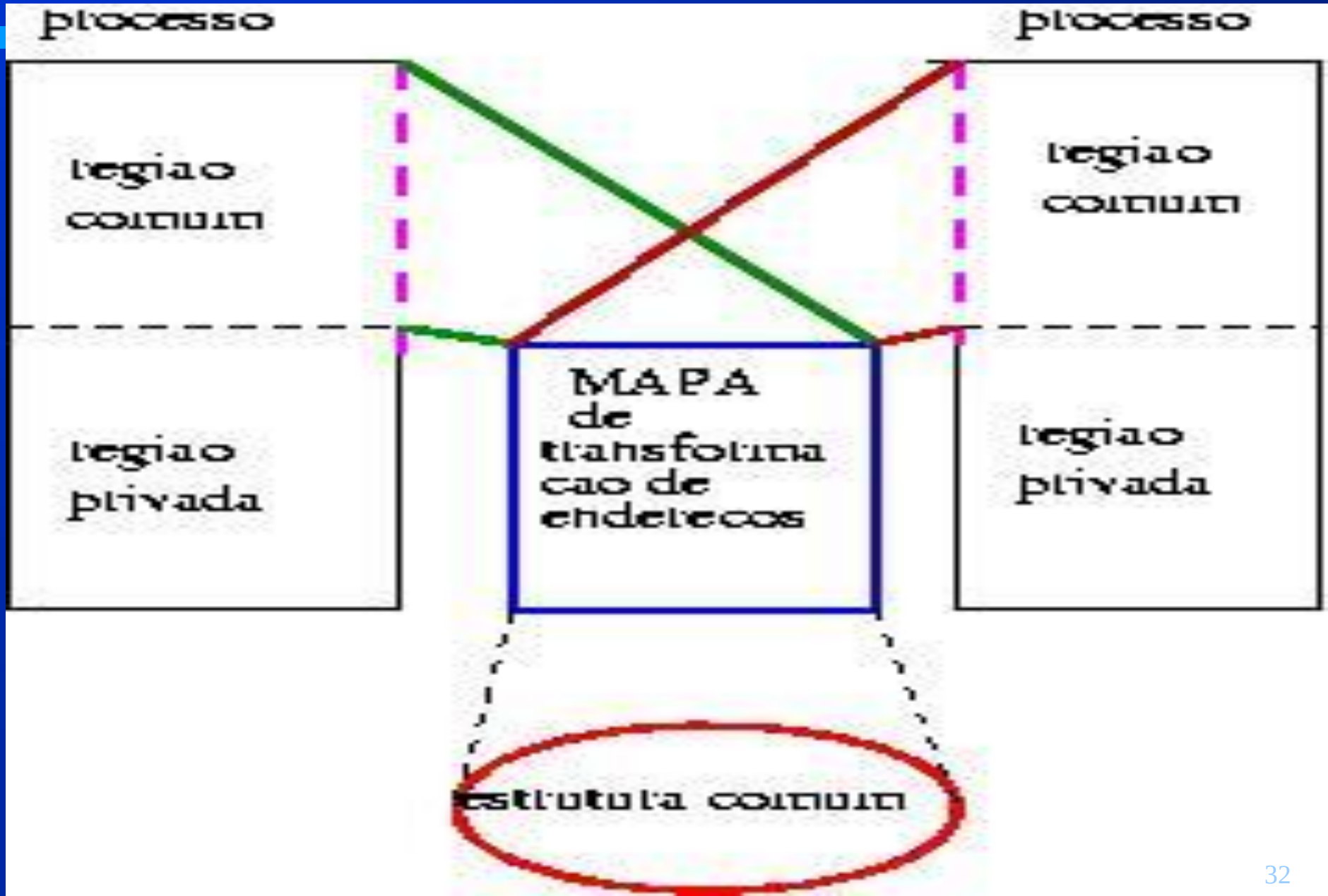
Memória partilhada entre processos

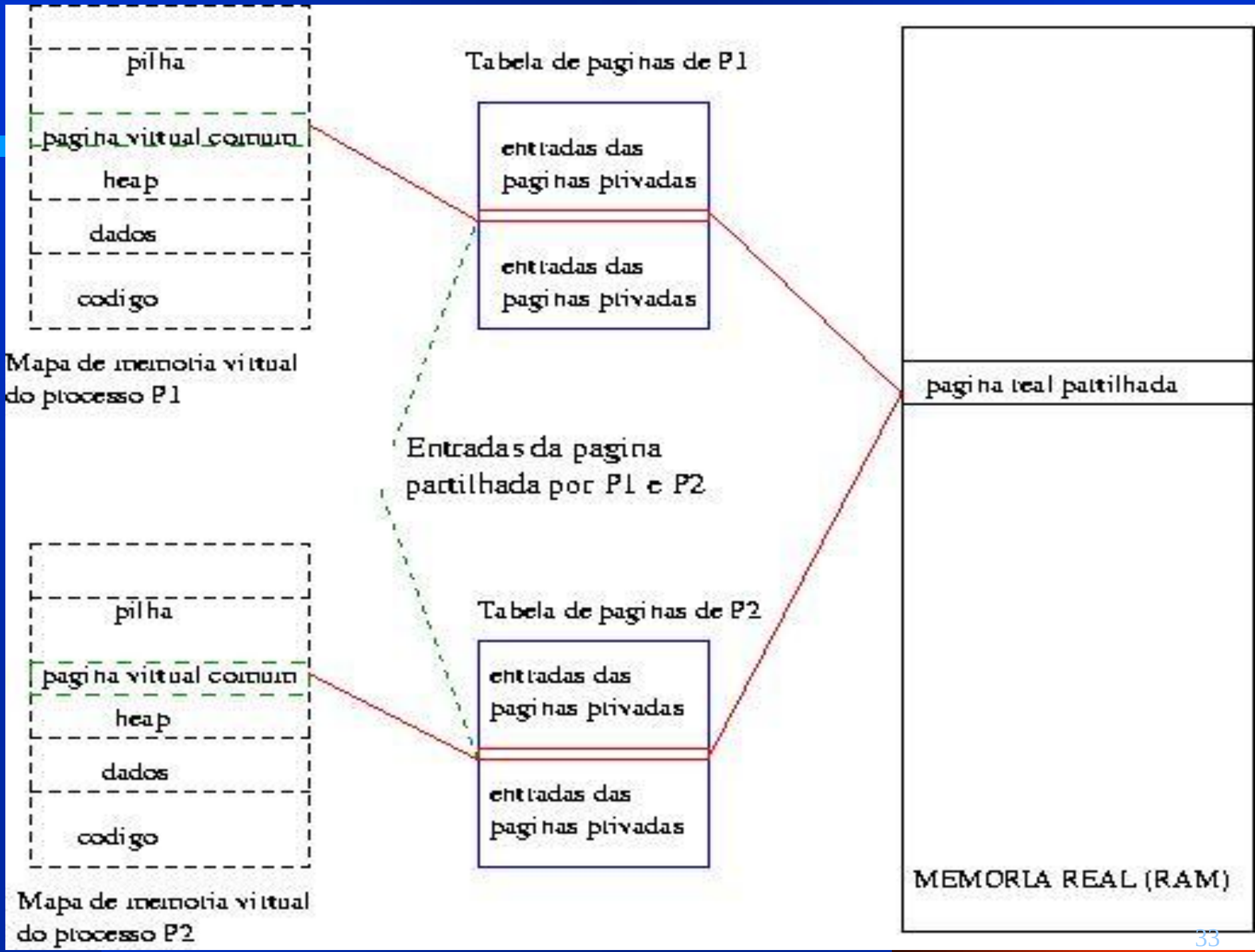
- Espaços de endereçamento de processos distintos referem a mesma memória
- Memória física partilhada:
  - acesso a estruturas de dados comuns
  - sem envolver cópia de bytes
  - leitura não destrutiva
- Mas sem sincronização garantida...
- Para ser eficiente, exige arquitecturas com memória física partilhada

Memória Partilhada entre Processos

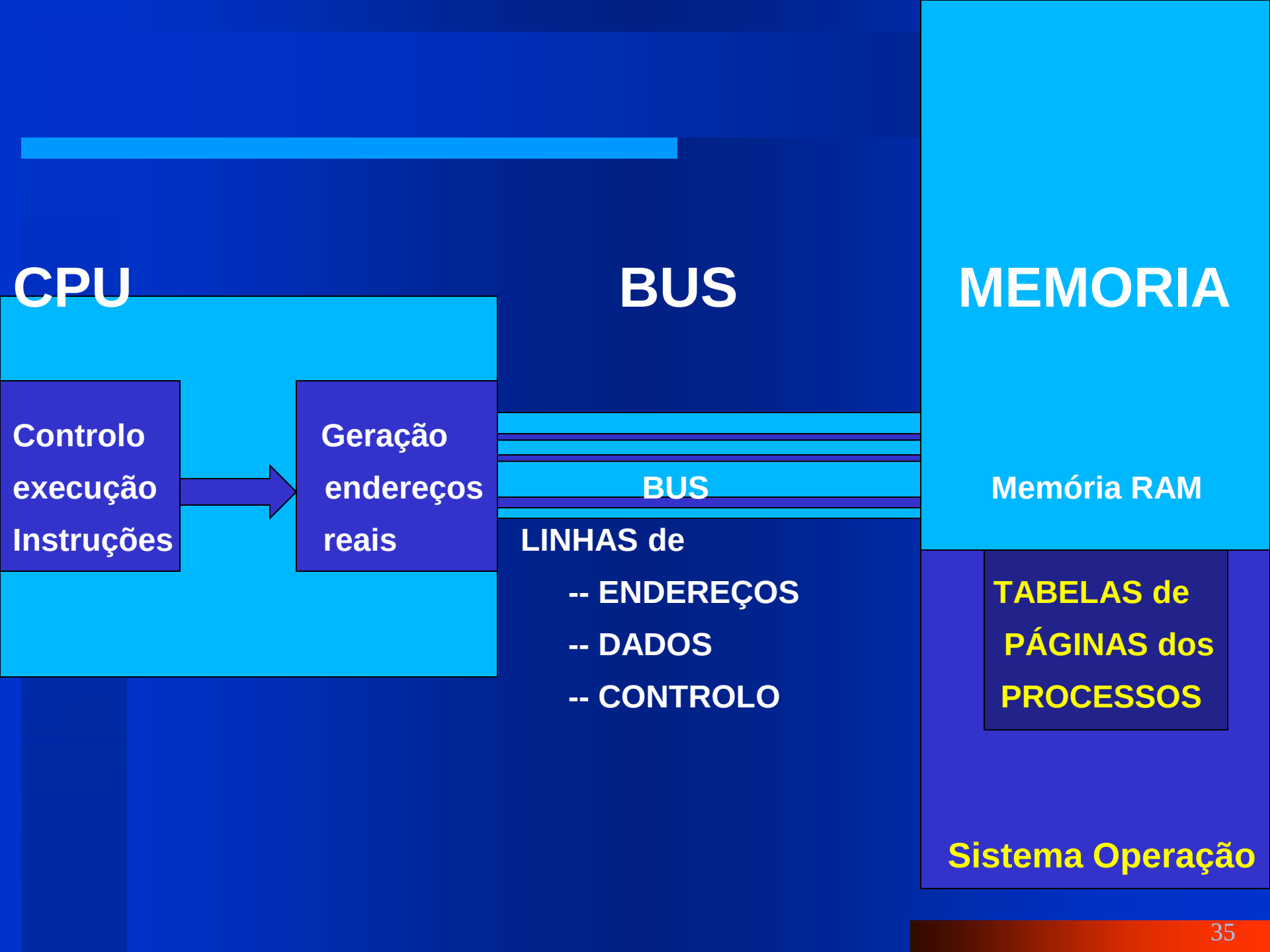


Memória Partilhada entre Processos




pilha
pagina virtual comum
heap
dados
codigo
Mapa de memoria virtual do processo P1
Tabela de paginas de P1
entradas das paginas privadas
entradas das paginas privadas
Entradas da pagina partilhada por P1 e P2
pagina real partilhada
pilha
pagina virtual comum
heap
dados
codigo
Mapa de memoria virtual do processo P2
Tabela de paginas de P2
entradas das paginas privadas
entradas das paginas privadas
MEMORIA REAL (RAM)


CPU
BUS
MEMORIA
Controlo
execução
Instruções
Geração
endereços
reais
BUS
Memória RAM
LINHAS de
-- ENDEREÇOS
-- DADOS
-- CONTROLO


CPU
BUS
MEMORIA
Controlo execução Instruções
Geração endereços reais
BUS
LINHAS de
-- ENDEREÇOS
-- DADOS
-- CONTROLO
Memória RAM
TABELAS de PÁGINAS dos PROCESSOS
Sistema Operação

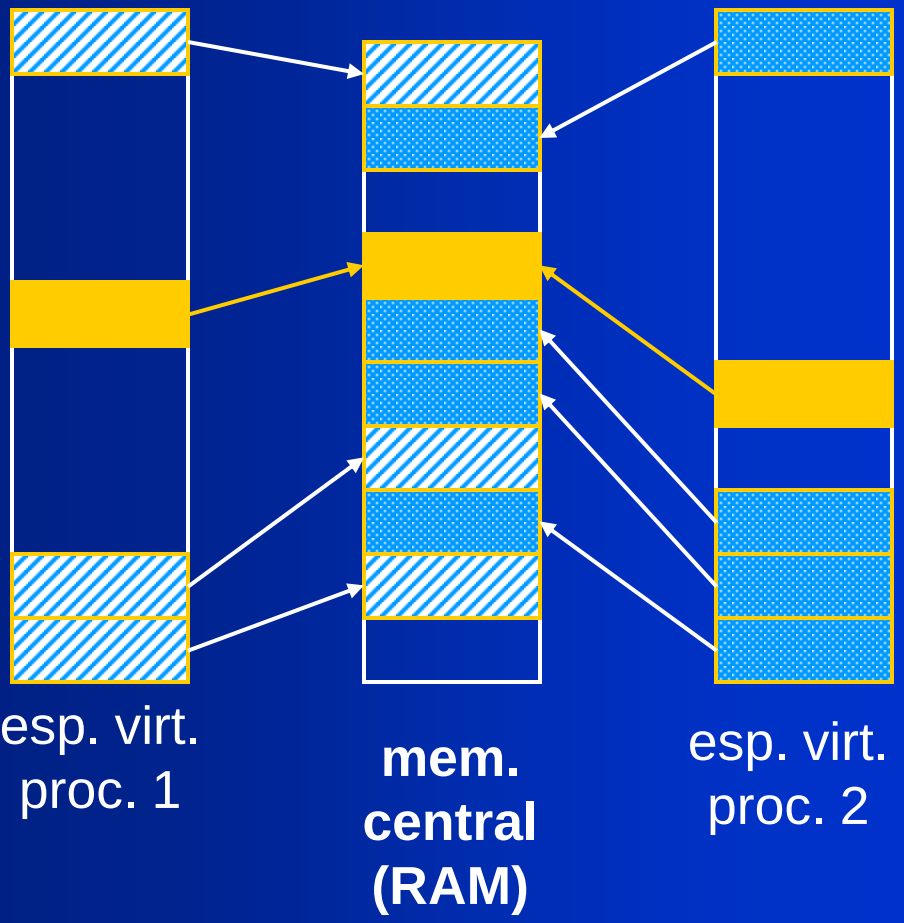
Memória Partilhada entre Processos

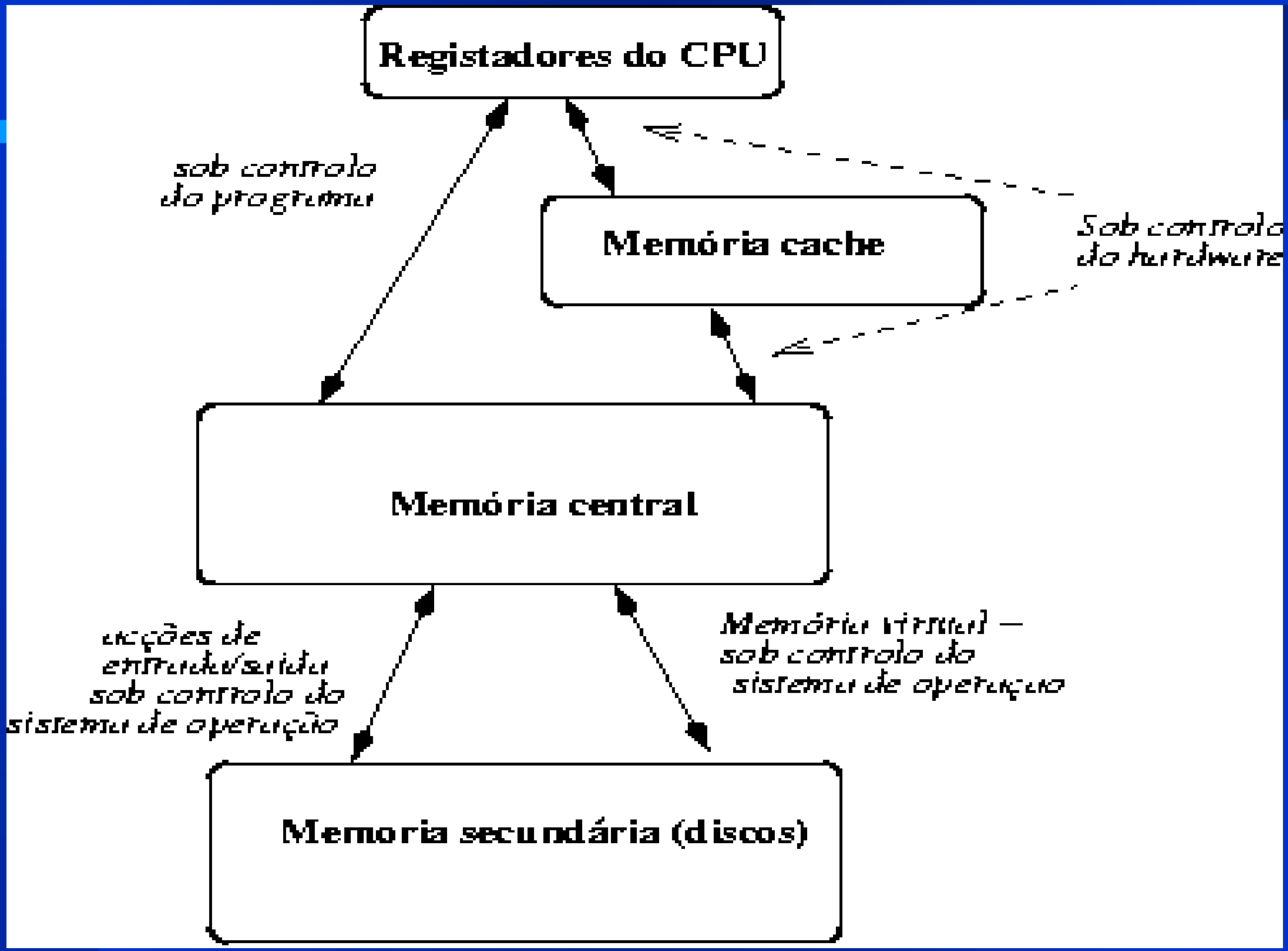
-um processo pede ao SO para reservar memória partilhável

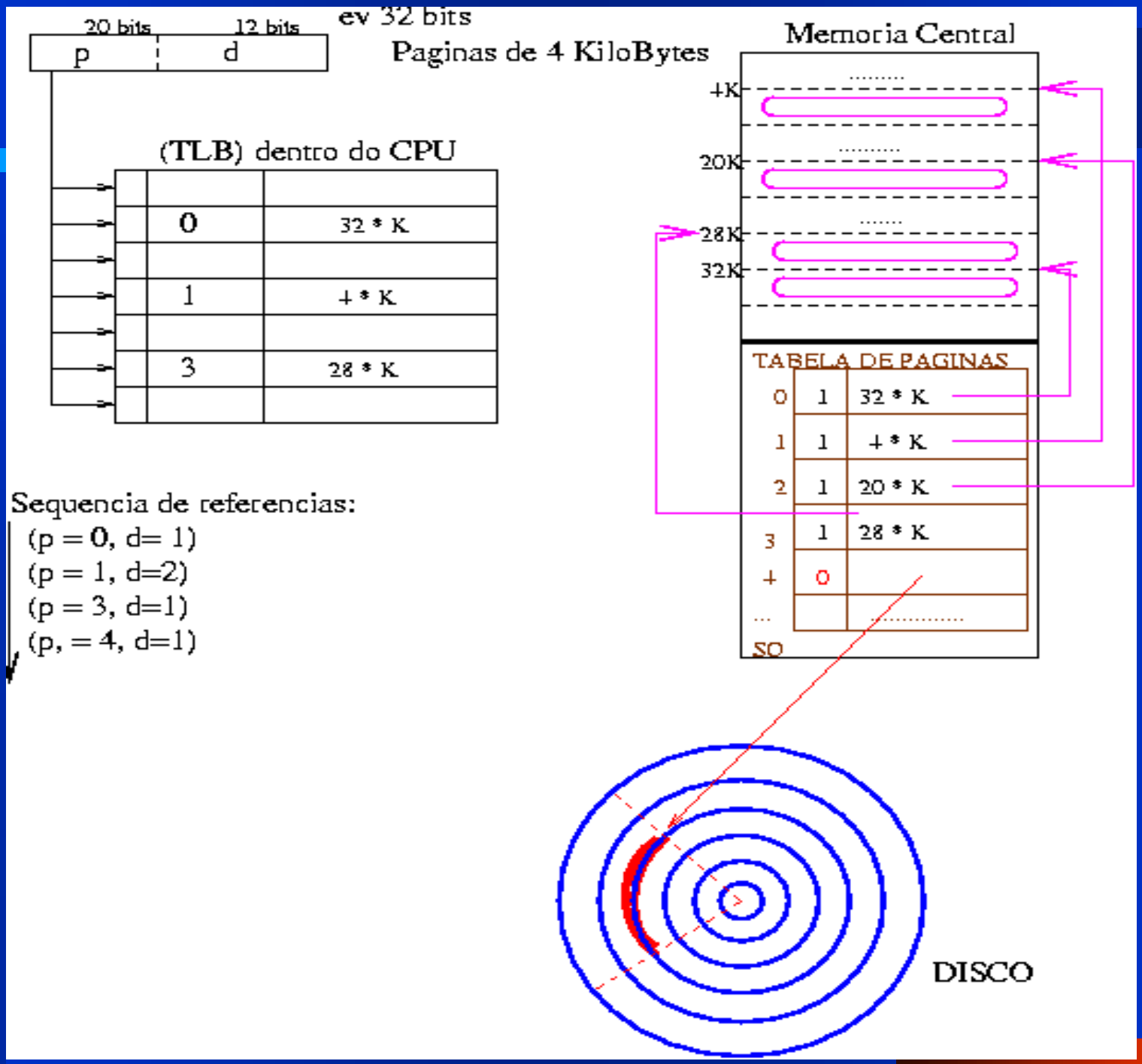
-projecta-a no espaço de endereçamento do processo

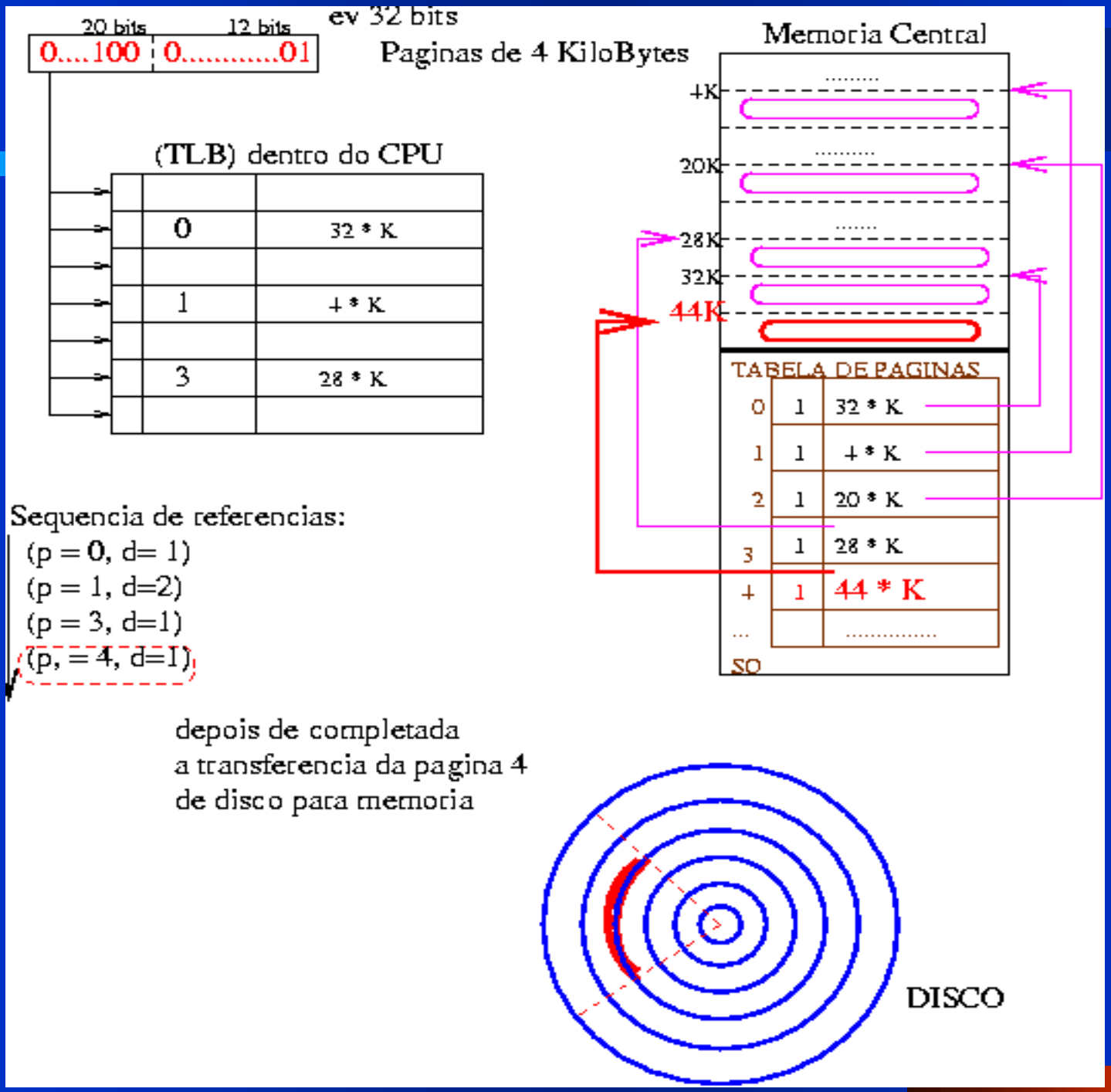
-outro processo pode projectar a mesma zona de memória no seu espaço de endereçamento

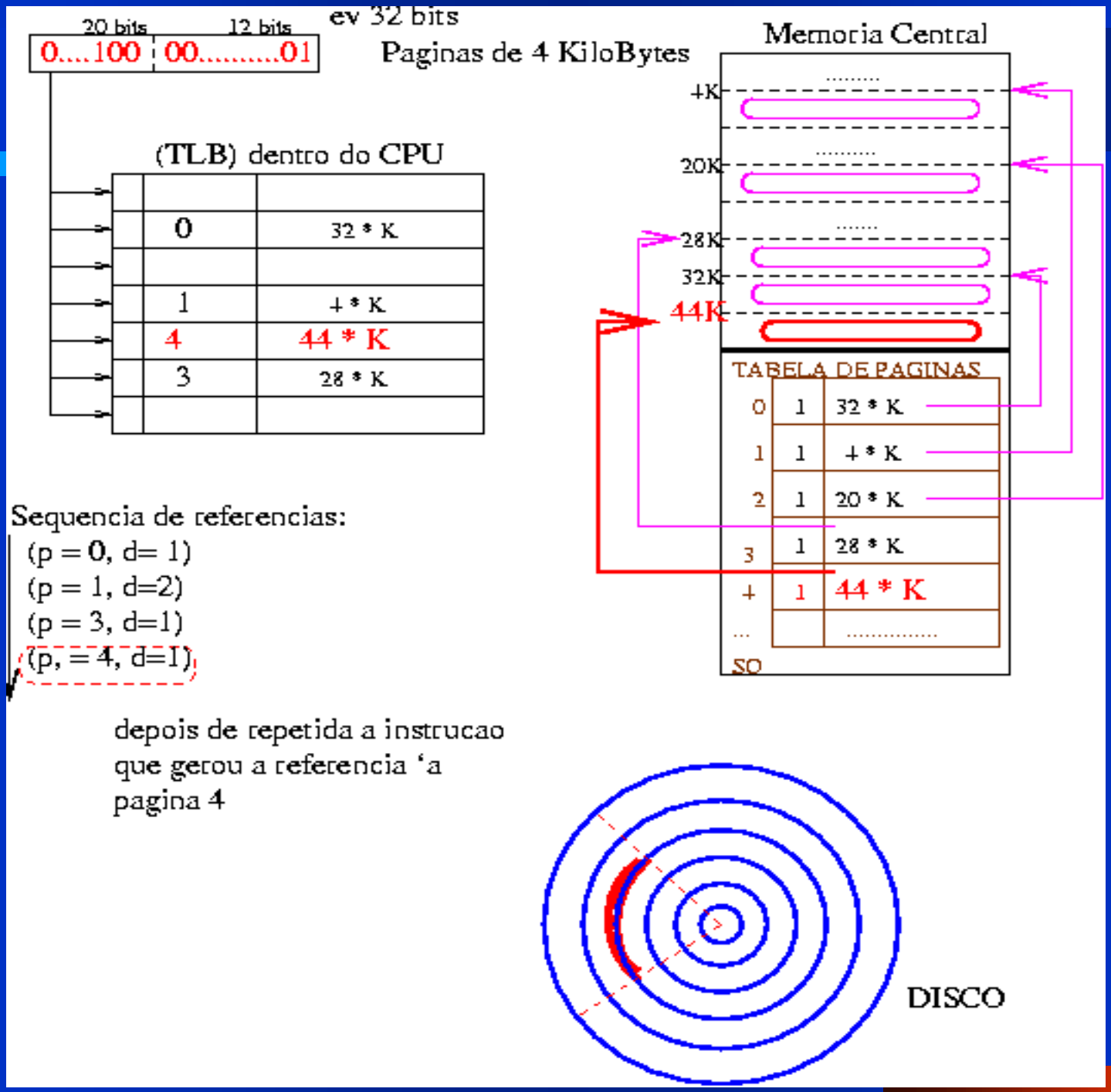
exemplo com paginação




Registadores do CPU
sob controlo do programa
Memória cache
Sob controlo do hardware
Memória central
acções de entrada/saída sob controlo do sistema de operação
Memória virtual — sob controlo do sistema de operação
Memoria secundária (discos)


20 bits
12 bits
p
d
ev 32 bits
Paginas de 4 KiloBytes
Memoria Central
(TLB) dentro do CPU
0
32 * K
1
4 * K
3
28 * K
4K
20K
28K
32K
TABELA DE PAGINAS
0 1 32 * K
1 1 4 * K
2 1 20 * K
3 1 28 * K
4 0
SO
Sequencia de referencias:
(p = 0, d= 1)
(p = 1, d=2)
(p = 3, d=1)
(p, = 4, d=1)
DISCO


20 bits
12 bits
ev 32 bits
0....100
0.....01
Paginas de 4 KiloBytes
Memoria Central
(TLB) dentro do CPU
0
32 * K
1
4 * K
3
28 * K
4K
20K
28K
32K
44K
TABELA DE PAGINAS
0 1 32 * K
1 1 4 * K
2 1 20 * K
3 1 28 * K
4 1 44 * K
SO
Sequencia de referencias:
(p = 0, d= 1)
(p = 1, d=2)
(p = 3, d=1)
(p, = 4, d=1)
depois de completada
a transferencia da pagina 4
de disco para memoria
DISCO


20 bits
12 bits
ev 32 bits
0....100
00.....01
Paginas de 4 KiloBytes
Memoria Central
(TLB) dentro do CPU
0
32 * K
1
4 * K
4
44 * K
3
28 * K
4K
20K
28K
32K
44K
TABELA DE PAGINAS
0 1 32 * K
1 1 4 * K
2 1 20 * K
3 1 28 * K
4 1 44 * K
SO
Sequencia de referencias:
(p = 0, d= 1)
(p = 1, d=2)
(p = 3, d=1)
(p, = 4, d=1)
depois de repetida a instrucao
que gerou a referencia 'a
pagina 4
DISCO

Memória partilhada no Unix

Integrada no conjunto de mecanismos IPC do Unix System V

IPC = *InterProcess Communication*

a)-- *Shared Memory* : memória partilhada → comunicação

b)-- *Semaphores* : semáforos → sincronização

(c)-- Message Queues : filas de mensagens)

Só para para arquitecturas MONOPROCESSADORES ou MULTIPROCESSADORES de *Bus* Comum com Memória Partilhada

IPC: *shared memory*

Para comunicar por Memória Partilhada:

- **Pedir ao SO que reserve uma zona de memória central física comum, para partilhar**

```
int shmget(key,size,flags)
```

devolve *identificador* `id` único da zona reservada

IPC: *shared memory*

Para comunicar por Memória Partilhada:

- **Pedir ao SO que reserve uma zona de memória central física comum, para partilhar**

```
int shmget(key,size,flags)
```

devolve *identificador* `id` único da zona reservada

- **Cada processo deve associar (*attach*) uma zona do seu mapa virtual àquela zona de memória física**

```
void *shmat(id, …, …)
```

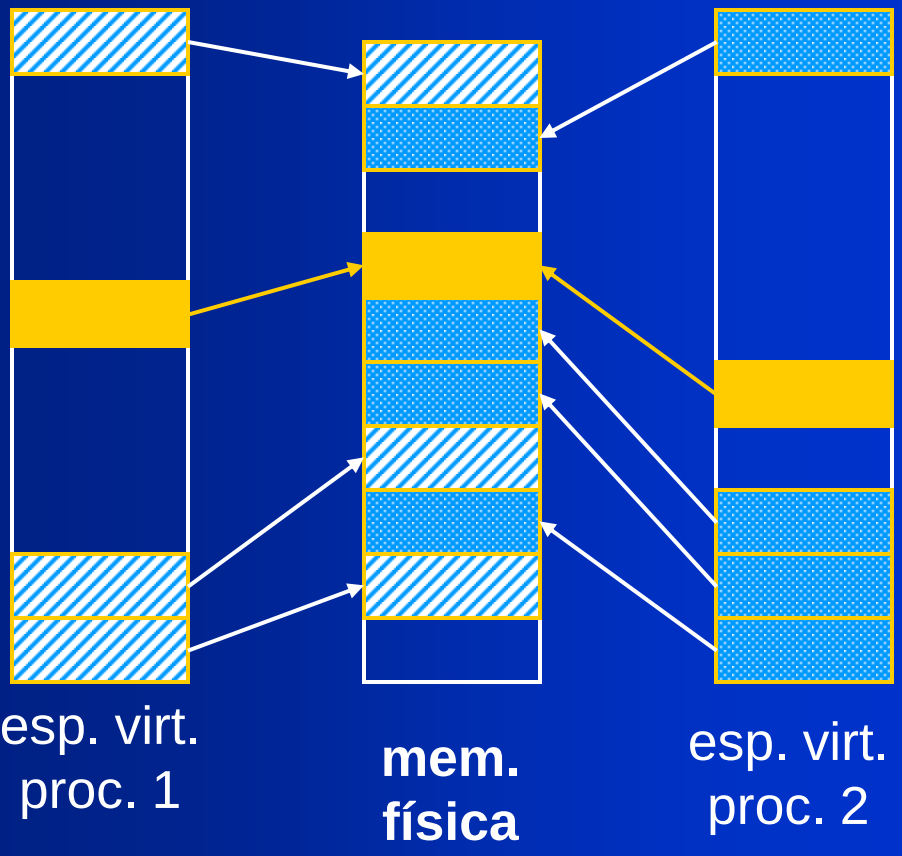
devolve um apontador para o início da zona partilhada

  - **o SO contabiliza os processos com esta memória projectada**

Memória Partilhada entre Processos

- o processo pede/cria memória física partilhável
- projecta no espaço de endereçamento do processo
- outro processo pode projectar a **mesma memória** no seu espaço de endereçamento

exemplo com paginação



Deixar de usar Mem. Partilhada

Para desligar de cada processo(´detach´):

```
int shmdt(void *ptr)
```

Para a libertar:

```
int shmctl(id,cmd,arg)
```

cmd = IPC_RMID // (*remove*)

- **a memória só será realmente libertada após todos os processos que fizeram ´´attach´´ tiverem feito ´´detach´´**

Exemplo

Process P1:

```
#define MSGKEY 80
…
char *ptr;
…
id=shmget(MSGKEY,…);
…
ptr=shmat(id,…,…);
*ptr = ´A´;
```

Process P2:

```
#define MSGKEY 80
…
char *p;
…
m=shmget(MSGKEY,…);
…
p=shmat(m,…,…);
*p = ´B´;
```

`ptr` e `p` apontam a mesma memória física, mas são provavelmente endereços virtuais diferentes, em P1 e P2

Exemplo

Process P1:

```
#define MSGKEY 80
…
struct {int a; int b;} *ptr;
…
id=shmget(MSGKEY,…);
…
ptr=shmat(id,…,…);
ptr->b = 2;
```

Process P2:

```
#define MSGKEY 80
…
struct {int a; int b;} *p;
…
m=shmget(MSGKEY,…);
…
p=shmat(m,…,…);
p->a = 1;
```

`ptr` e `p` apontam a mesma memória física, mas são provavelmente endereços virtuais diferentes, em P1 e P2

Processos concorrentes

- **Quando comunicam por Memória Partilhada:**

-- ler e escrever dados de memória

-- sem haver cópia de bytes → muito rápida…

-- mas exige sincronização explícita pelo Programa

-- exige um bus comum de acesso a memória

Problemas fundamentais da concorrência, quando processos ou *threads* partilham memória

Independentemente de ser:

-Memória partilhada entre processos

(- Memória partilhada por *threads* dentro de um processo – a estudar mais adiante)

Programação Concorrente:
Memória partilhada

Problemas…

- **Se P1 afectar um valor a uma variável na memória partilhada, como avisar P2?**
- **Como sincronizar P1 e P2 para saberem das alterações mútuas?**
- **E como impedir que ambos alterem concorrentemente a mesma zona?**

(sem destruirem as alterações um do outro)?

Um problema ...

- **Se não há controlo sobre a concorrência de processos...**

Exemplo:

P1

```
...
v1 = n;
v1 = v1+1;
n = v1;
...
```

P2

```
v2 = n;
v2 = v2+1;
n  = v2;
```

n – variável partilhada por P1 e P2

v1 – local a P1

v2 - local a P2

Problemas...

Inicialmente n = 0

P1		P2
v1 = n fica v1 == 0	COMUTAÇÃO	
----------		v2=n; v2=v2+1; n=v2
v1=v1+1 fica v1 == 1		(fica n == 1)
n = v1 fica n == 1		

O resultado deveria ser equivalente a fazer:

n=n+1 seguido de n= n+1

ou seja, no fim, ficaria n == 2

Interferências entre Processos

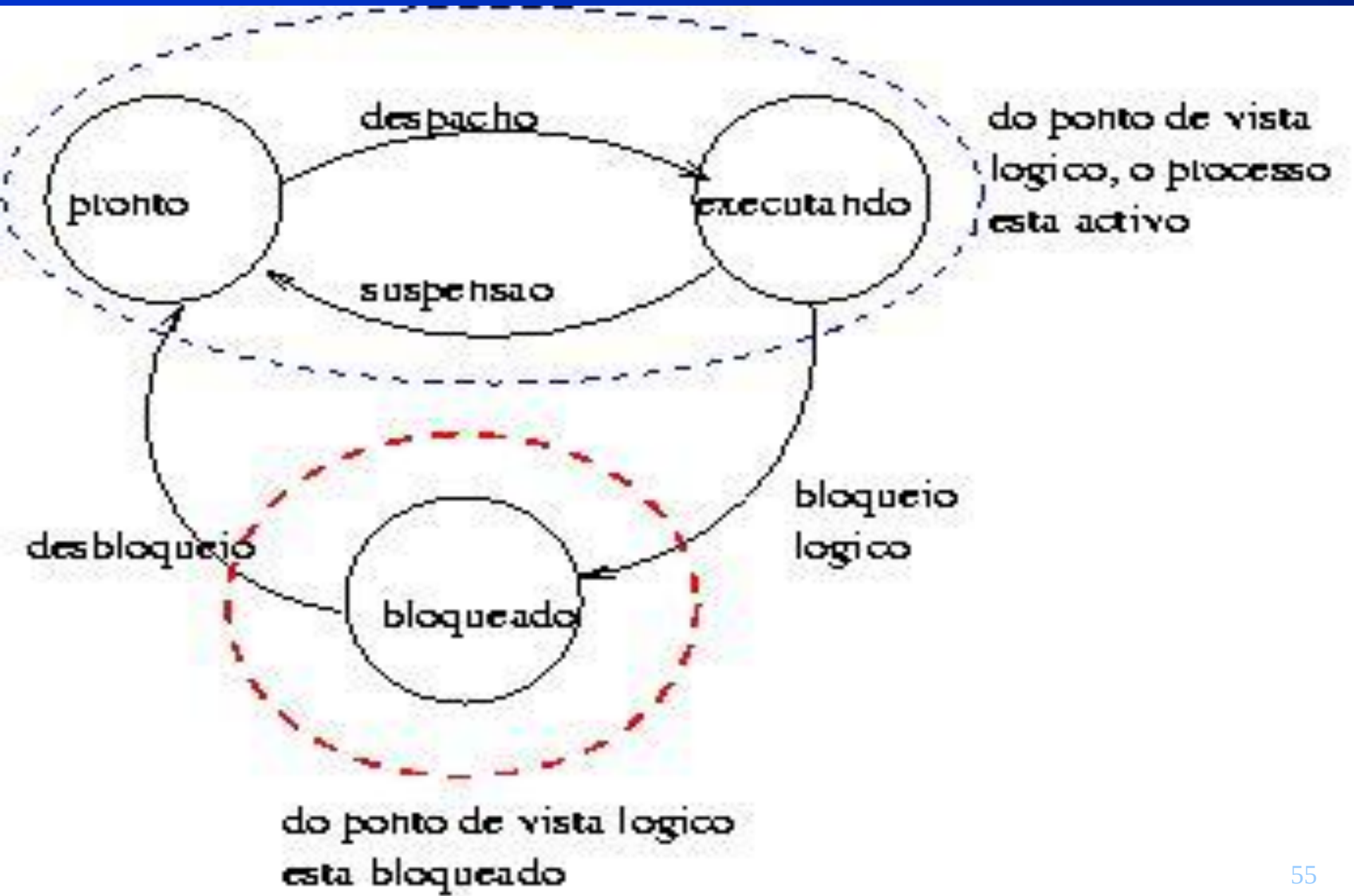
Acessos concorrentes a estruturas de dados partilhadas:

→ execução não-determinística:

A ordem de execução das instruções entre os múltiplos processos é decidida pelo SO, de uma forma imprevisível. Porquê?

1. **Multiprogramação: → *Time-slice***
2. **Interrupções pedidas e fim de operações dos periféricos: em instantes aleatórios**
3. **Conjunto de processos em execução: varia ao longo do tempo**

Estados de um processo no SO



Execução concorrente num CPU

Sejam P e Q dois processos independentes

P: { p1; p2} e Q: {q1; q2}

A execução concorrente de P com Q define o conjunto de todas as sequências possíveis:

p1; p2; q1; q2 equivale a P; Q

p1; q1; p2; q2

p1; q1; q2; p2

q1; q2; p1; p2 equivale a Q; P

q1; p1; q2; p2

q1; p1; p2; q2

----------→ Tempo

Correcção de um programa concorrente

Programa concorrente: especifica múltiplos processos concorrentes

A sua execução tem de produzir resultados correctos em TODAS as sequências de execução possíveis.

Fácil: se processos INDEPENDENTES.

Difícil: se processos DEPENDEM uns dos outros

Ao repetir a execução de um MESMO programa com os MESMOS dados, os resultados podem ser DIFERENTES!

Acção: incrementar x

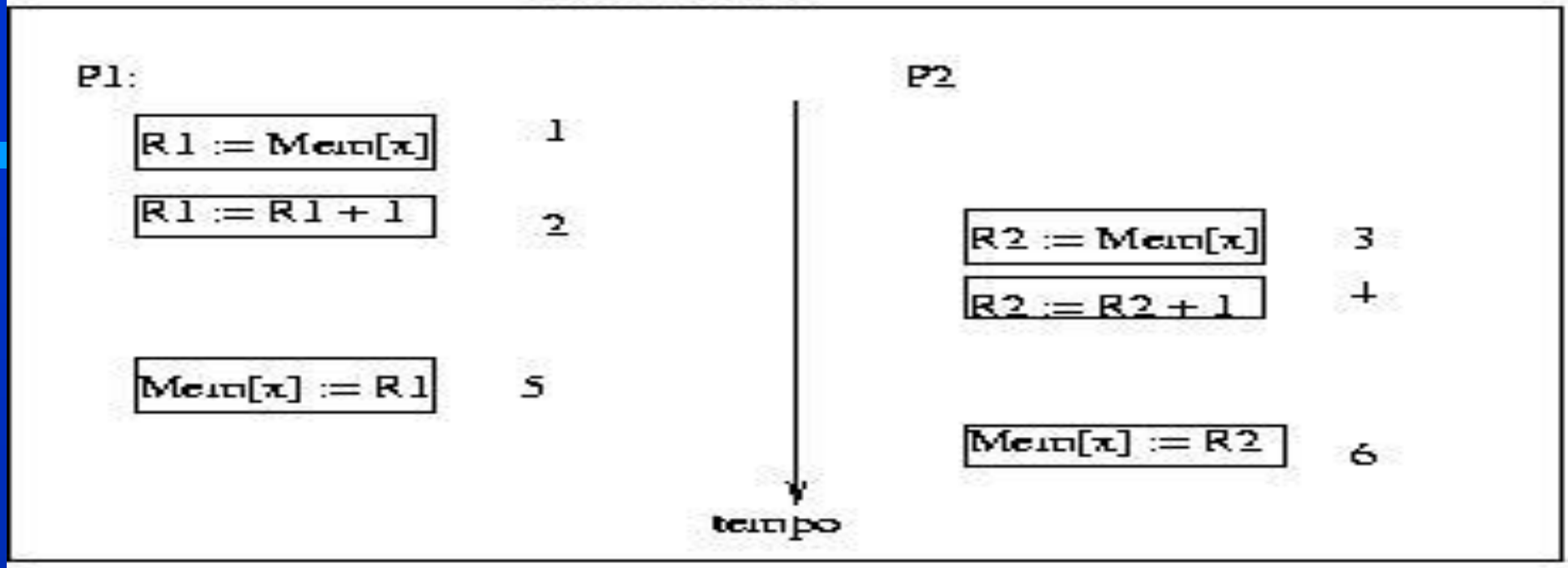
A variável x é representada em memória

Se:

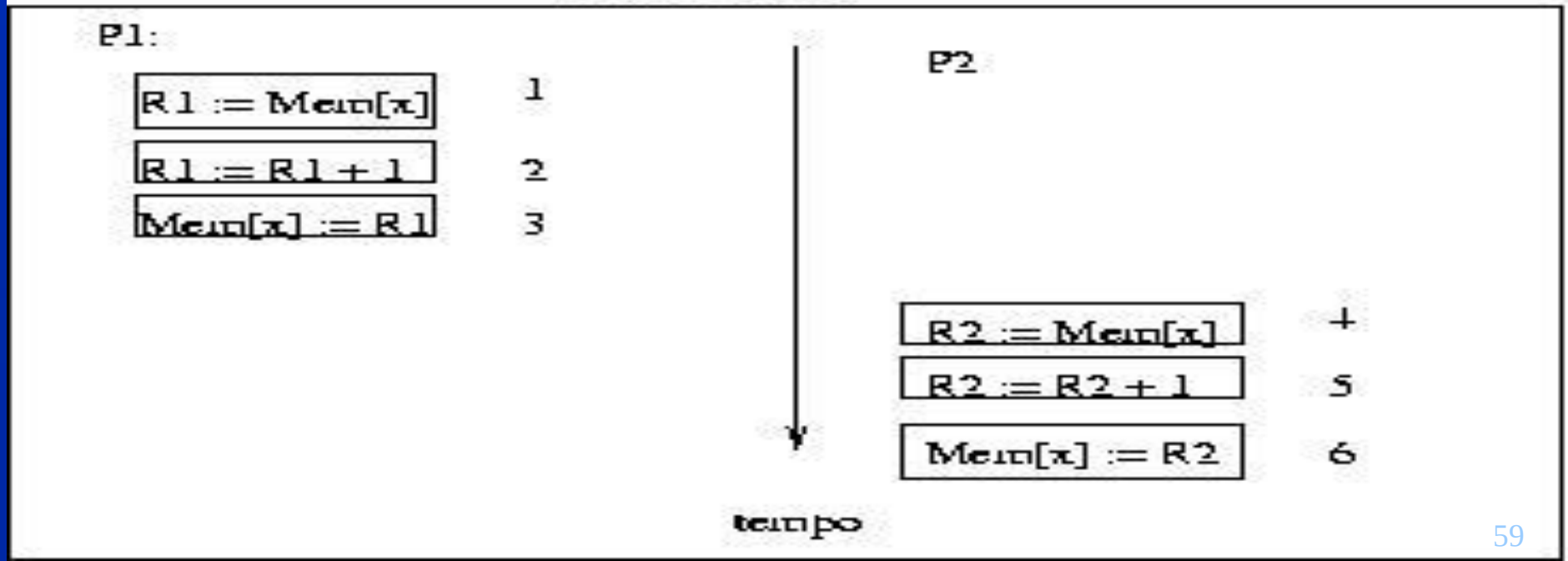
Incrementar é realizada por múltiplas instruções máquina:

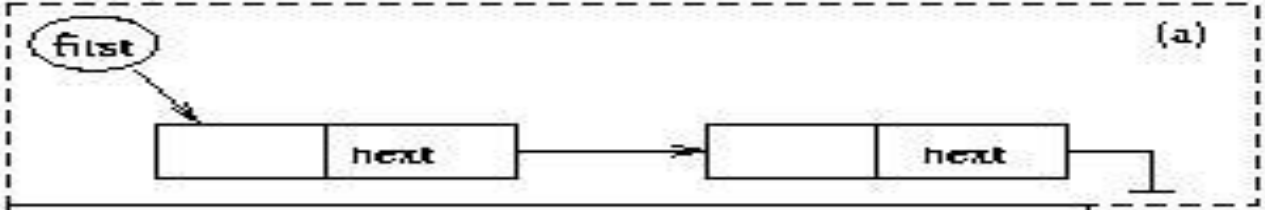
```
RegistadorCPU  = Mem[x]
RegistadorCPU  = RegistadorCPU + 1
Mem[x]  = RegistadorCPU
```

Execucao (A)



Execucao (B)





P1: inserir(E): next(E) := first;
first := E;
(b)

P1: inserir(E): next(E) := first; 1

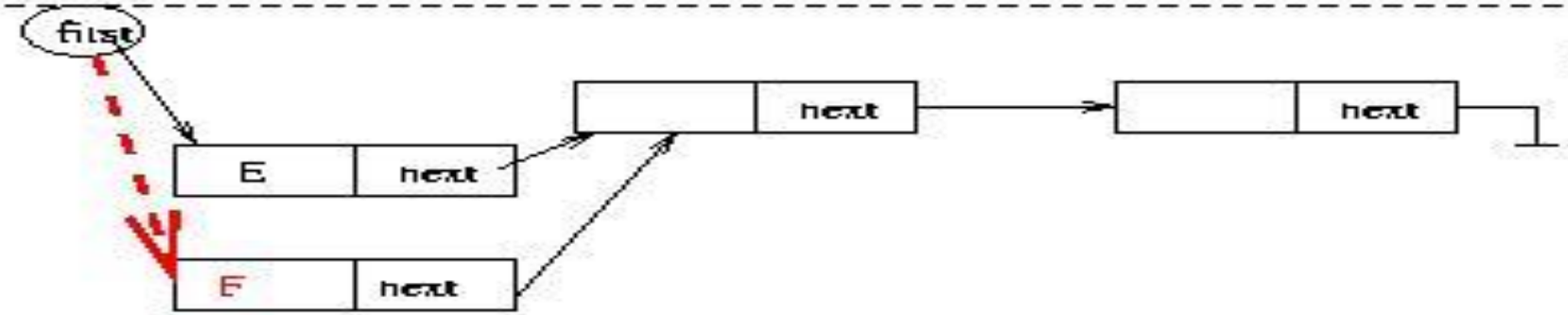
P2: inserir(F): next(F) := first; 2

first := F; 3

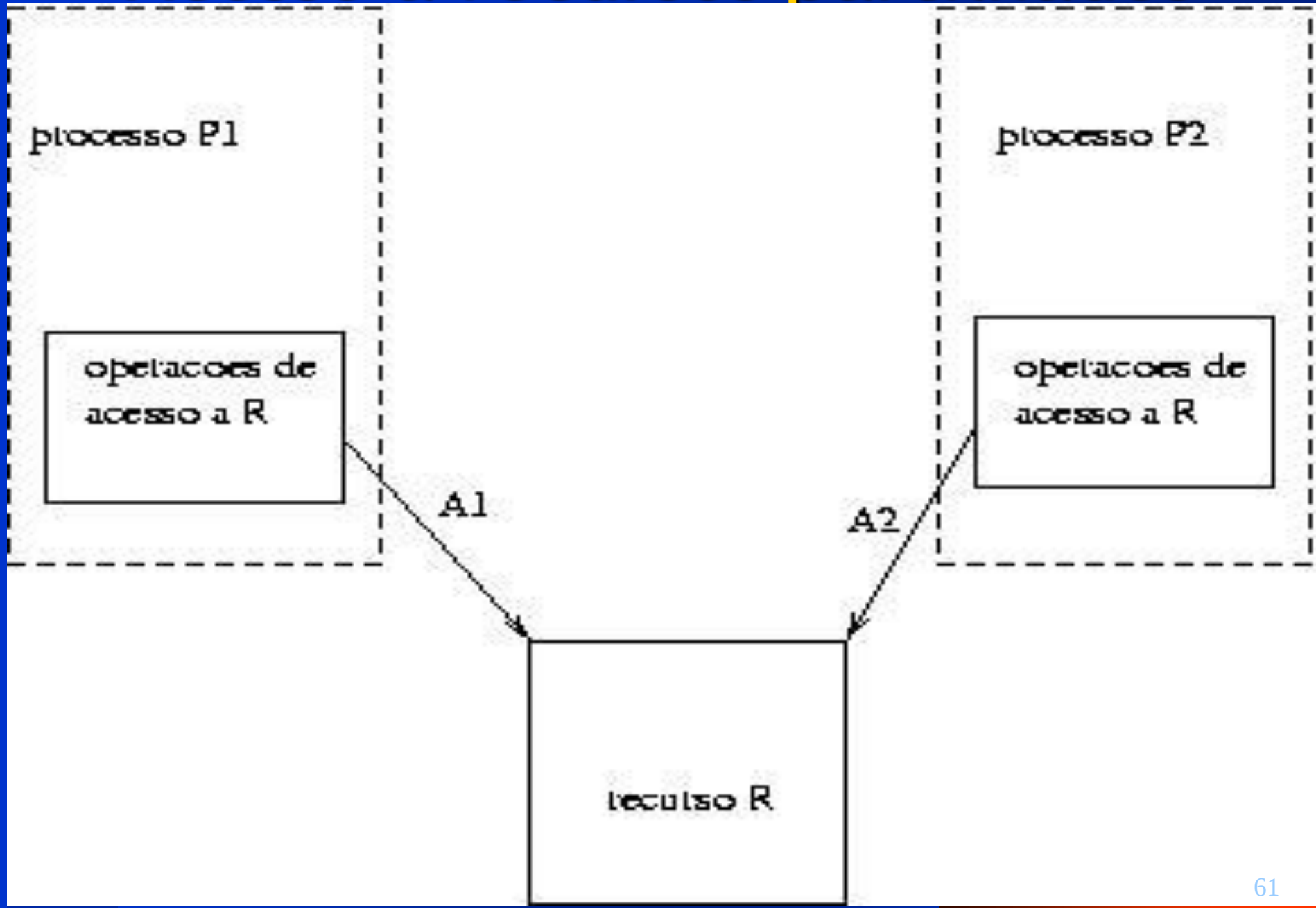
first := E; +

tempo

(d)



Acesso a recursos partilhados



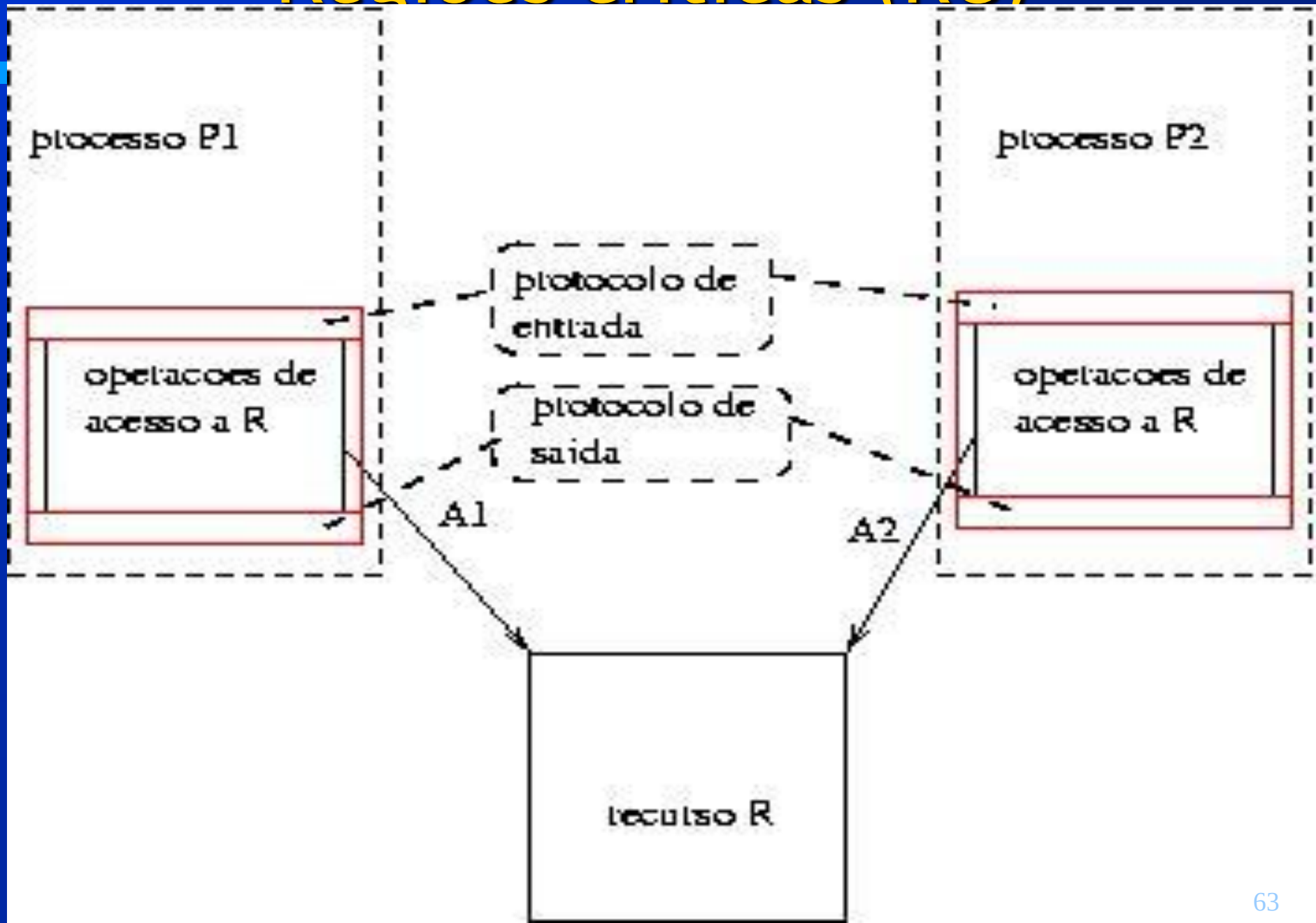
Regiões ou Secções Críticas

- **Regiões do código de um programa que acedem a recursos partilhados com outro programa concorrente...**
- **É necessário sincronizar as acções concorrentes que interferem**
  - **impor uma ordem ou impedir certas ordens:**

  →*exclusão mútua* na execução dessas acções:

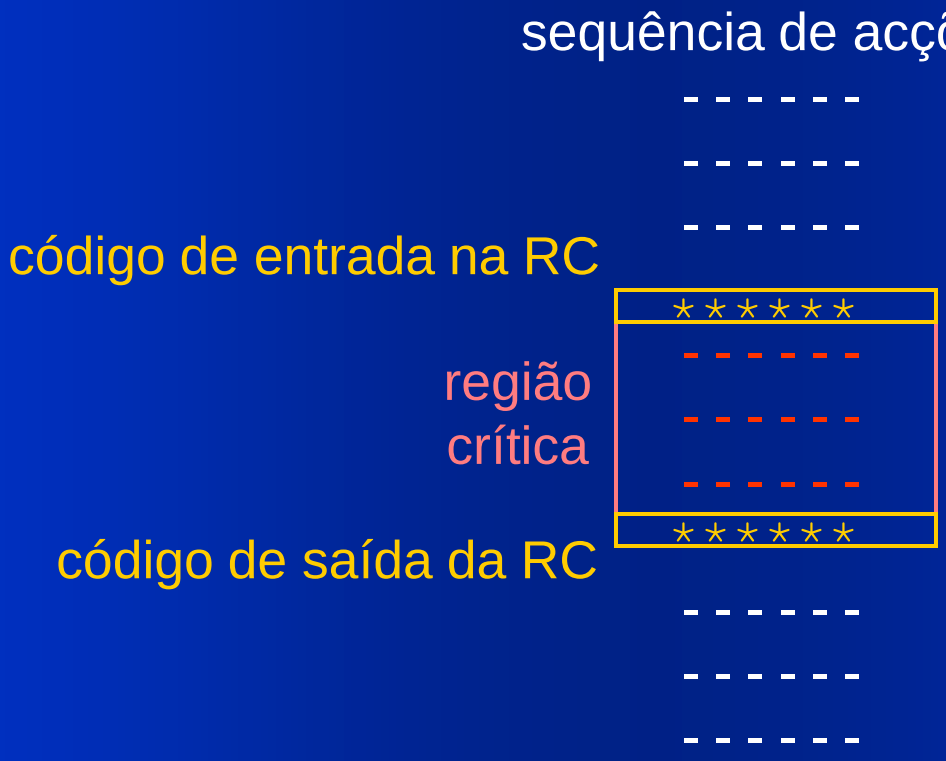
  1 acção de cada vez!
- **Precisamos de protocolos de coordenação do acesso à região crítica**

Regiões críticas (RC)



Exclusão Mútua

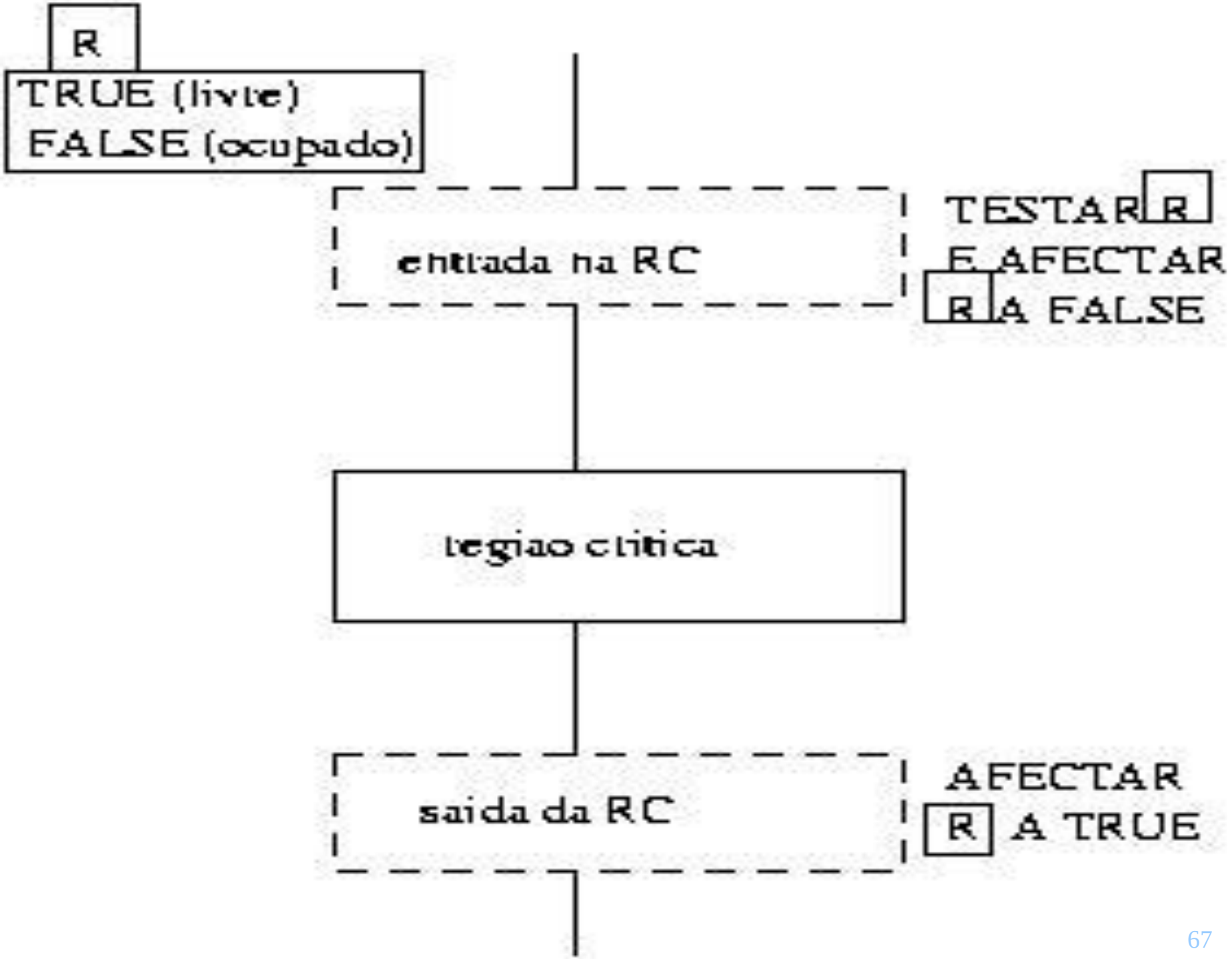
- **O código entrada / saída garante que só um processo está na RC a cada momento**



Exclusão Mútua (exemplo)

- **O objectivo é que cada região crítica seja indivisível ou atómica**

Como realizar os protocolos que coordenam o acesso às regiões críticas?


R
TRUE (livre)
FALSE (ocupado)
entrada na RC
TESTAR R
E AFECTAR
R A FALSE
regiao critica
saida da RC
AFECTAR
R A TRUE

Ou seja...

- **Usando a própria mem. partilhada:**
  - **uma variável indica que existe um processo na RC (exemplo: a escrever na memória partilhada):**

```
while ( ! livre )
   ;
livre = FALSE;
```

código entrada

```
<Região Crítica>
```

```
livre = TRUE;
```

código saída

Acções indivisíveis ou atómicas

As acções de Entrada e Saída na Região Crítica são elas próprias regiões críticas!!!

- **têm de ser indivisíveis ou atómicas:**
  - **se um processo P estiver a executar Entrada, não pode ser interrompido e o SO passar o controlo da execução para outro processo Q, que vá executar Entrada também!**

Porquê?

  - **Isto pode suceder, mesmo que o computador só tenha 1 CPU, sempre que o SO faça multiprogramação**

Porquê?

Falta de atomicidade...

- **Se só existe 1 CPU:**
  - **sabemos que cada instrução máquina não é interrompível (o CPU só atende interrupções no fim da execução de cada instrução)**

  código Saída: afectar `livre` a TRUE → está resolvido!

  código Entrada: testar `livre` e, se estiver TRUE, afectar a FALSE → não está resolvido, porque exige mais do que 1 instrução máquina!

```
ciclo: load reg, [livre]
       jumpifzero ciclo
       store [livre], 0
```

Soluções?

Memória partilhada dentro do mesmo Processo

Múltiplos Fluxos de execução - *Threads* concorrentes, partilhando o mesmo Espaço de Endereços do Processo

Um processo ou um objecto?:

-- nome global único (*process id*)

-- memória local privada / variáveis / estado

-- um fluxo de execução principal (o corpo do programa -- *main*)

-- um conjunto de funções de interface públicas que podem ser invocadas por outros objectos

Os Processos comunicam entre si, através de:

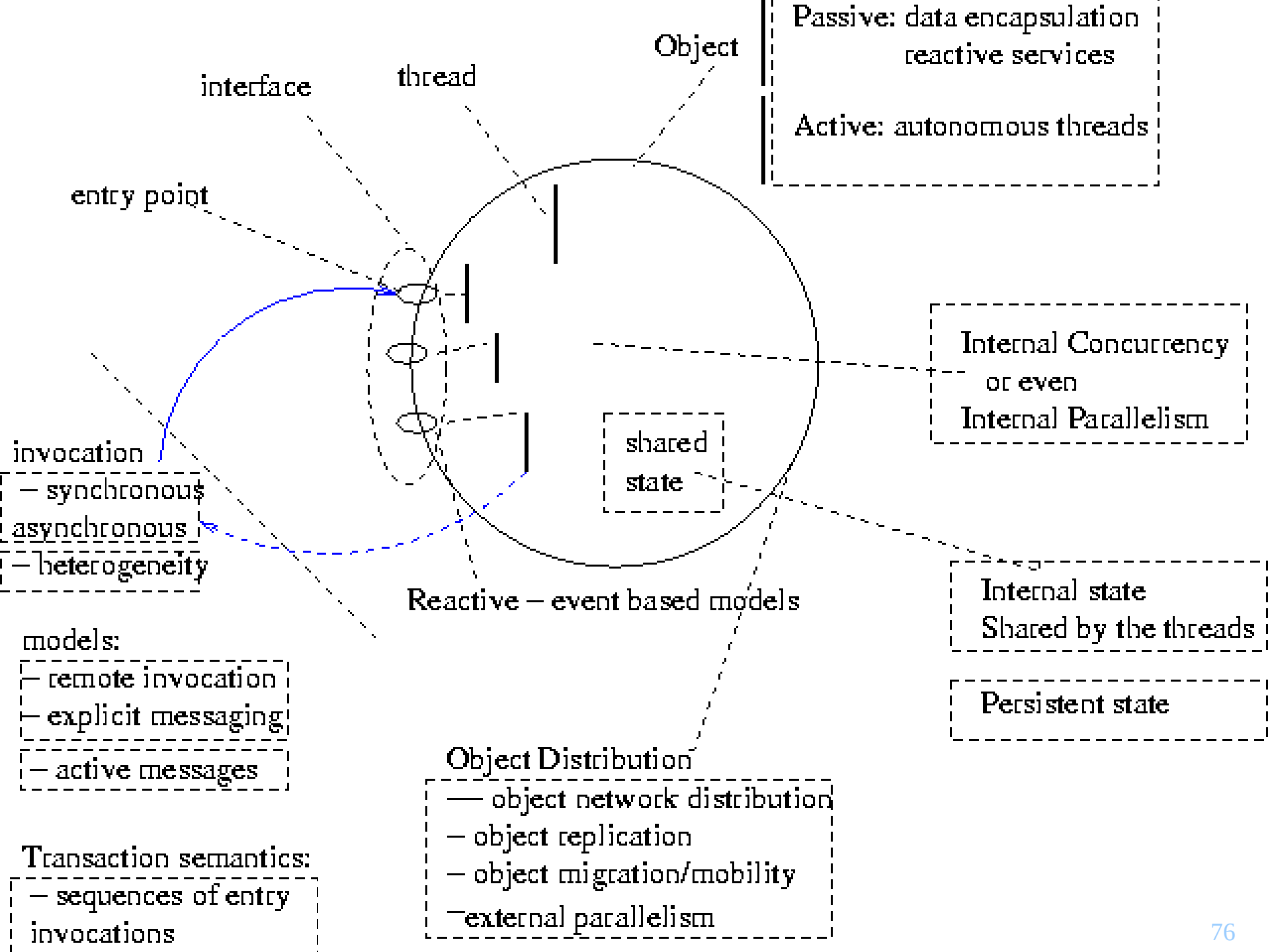
- invocações de funções de *interface (RPC/RMI)* RemoteProcedureCall / RemoteMethodInvocation
- Mensagens (cf. SEND/RECEIVE)

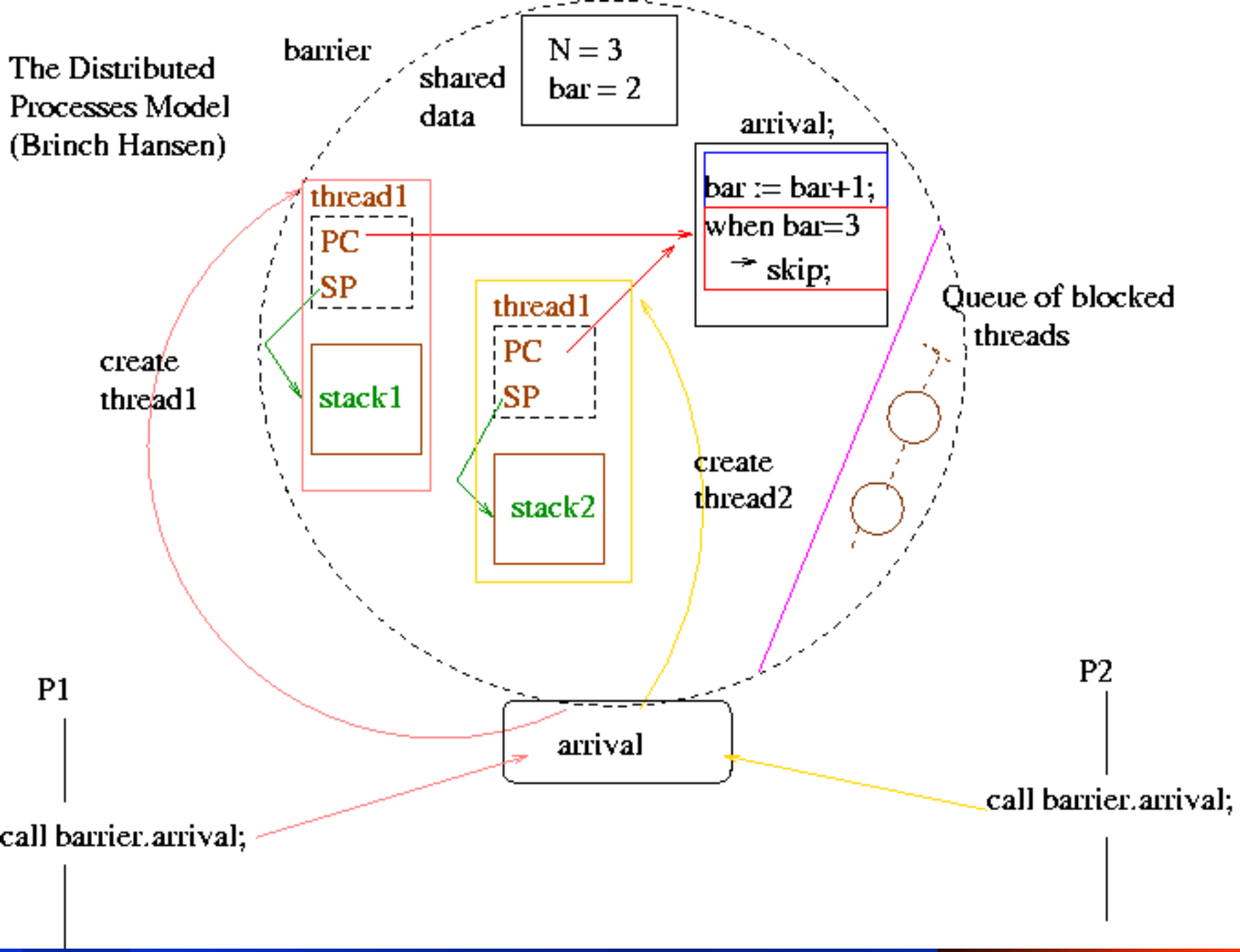
- **Um Processo/Objecto:**
  - -- uma unidade de Protecção
  - -- uma unidade de Multiprogramação
  - -- uma unidade de Distribuição espacial (possivelmente móvel…)
  - -- uma unidade para oferecer um ‘Serviço’
  - -- uma cápsula para Concorrência interna?

- **Múltiplos *Threads* dentro de um Processo?:**
  - -- partilham o espaço de endereços do processo (acesso à memória física partilhada)
  - -- cada *Thread* executa uma Função do programa
  - -- precisarão de se sincronizarem entre si

Dentro do Espaço de Endereços do Processo

- **Internos a cada processo, *threads* individuais executam-se** *(quando há invocação de funções da interface … se for como um objecto)*
- **Cada *thread* executa num contexto de execução distinto (registadores do CPU) e *stack*, mas todos os *threads* partilham o mesmo espaço de endereços (código e dados) e de I/O**
- **Num monoprocessor: um único *thread* em execução a cada instante**
- **Num multiprocessor de memória partilhada: múltiplos *threads* estão activos simultaneamente**
- **Modelos de *threads*: em linguagens como Java ou a nível de bibliotecas (norma POSIX *Pthreads*)**


Passive: data encapsulation
reactive services
Active: autonomous threads
Object
thread
interface
entry point
Internal Concurrency
or even
Internal Parallelism
shared
state
invocation
– synchronous
asynchronous
– heterogeneity
Reactive – event based models
Internal state
Shared by the threads
Persistent state
models:
– remote invocation
– explicit messaging
– active messages
Object Distribution
– object network distribution
– object replication
– object migration/mobility
–external parallelism
Transaction semantics:
– sequences of entry
invocations


The Distributed
Processes Model
(Brinch Hansen)
barrier
shared
data
N = 3
bar = 2
arrival;
bar := bar+1;
when bar=3
→ skip;
thread1
PC
SP
stack1
thread1
PC
SP
stack2
create
thread1
create
thread2
Queue of blocked
threads
P1
P2
arrival
call barrier.arrival;
call barrier.arrival;

Threads

- pthread_create
- *"attributes"*
- pthread_exit
  - ou return
- pthread_join

Processos

- fork/execve
- *atributos diferentes*
- exit
  - ou return
- waitpid