Programação Concorrente(4)

José C. Cunha, DI/FCT/UNL

- Comunicação por memória partilhada versus comunicação por mensagens.
- Modelos de comunicação por mensagens.

Modelos de comunicação

1. **memória partilhada: partilha de espaços de endereçamento, comuns a processos distintos;**
2. **memória distribuída: cópia de valores entre espaços de endereçamento (processos) diferentes:**
   - ***PIPES* Unix: fluxos (*streams*) de bytes, sem delimitadores de fronteiras, copiados entre processos, via buffers do SO;**
   - **Mensagens: ao nível do SO, conjuntos de *bytes* são enviados encapsulados em mensagens, unidades lógicas de transmissão, por operações para enviar e receber;**
   - **Invocação de pontos de entrada de processos ou de objectos: chamadas de procedimentos ou de métodos remotos (*Remote Procedure Call* - RPC ou *Remote Method Invocation* - RMI);**

- Por **memória física partilhada:**

-- acesso a estruturas de dados comuns

-- sem envolver cópia de *bytes*

- Por **memória distribuída**: 'meio' externo aos processos:

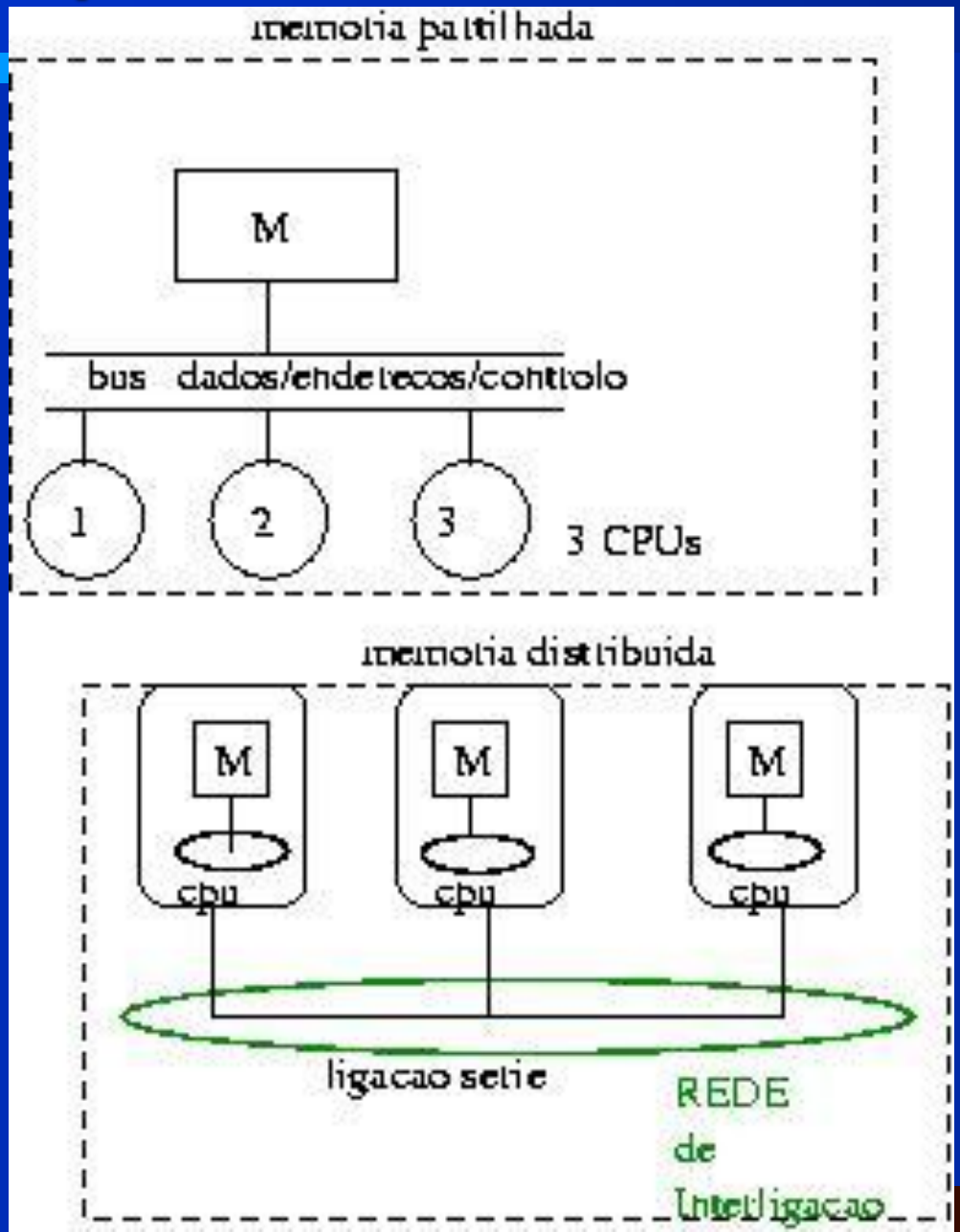
--- *pipe* Unix

--- uma caixa de correio (*mailbox*) ou fila de mensagens (*message queue Unix System V*)

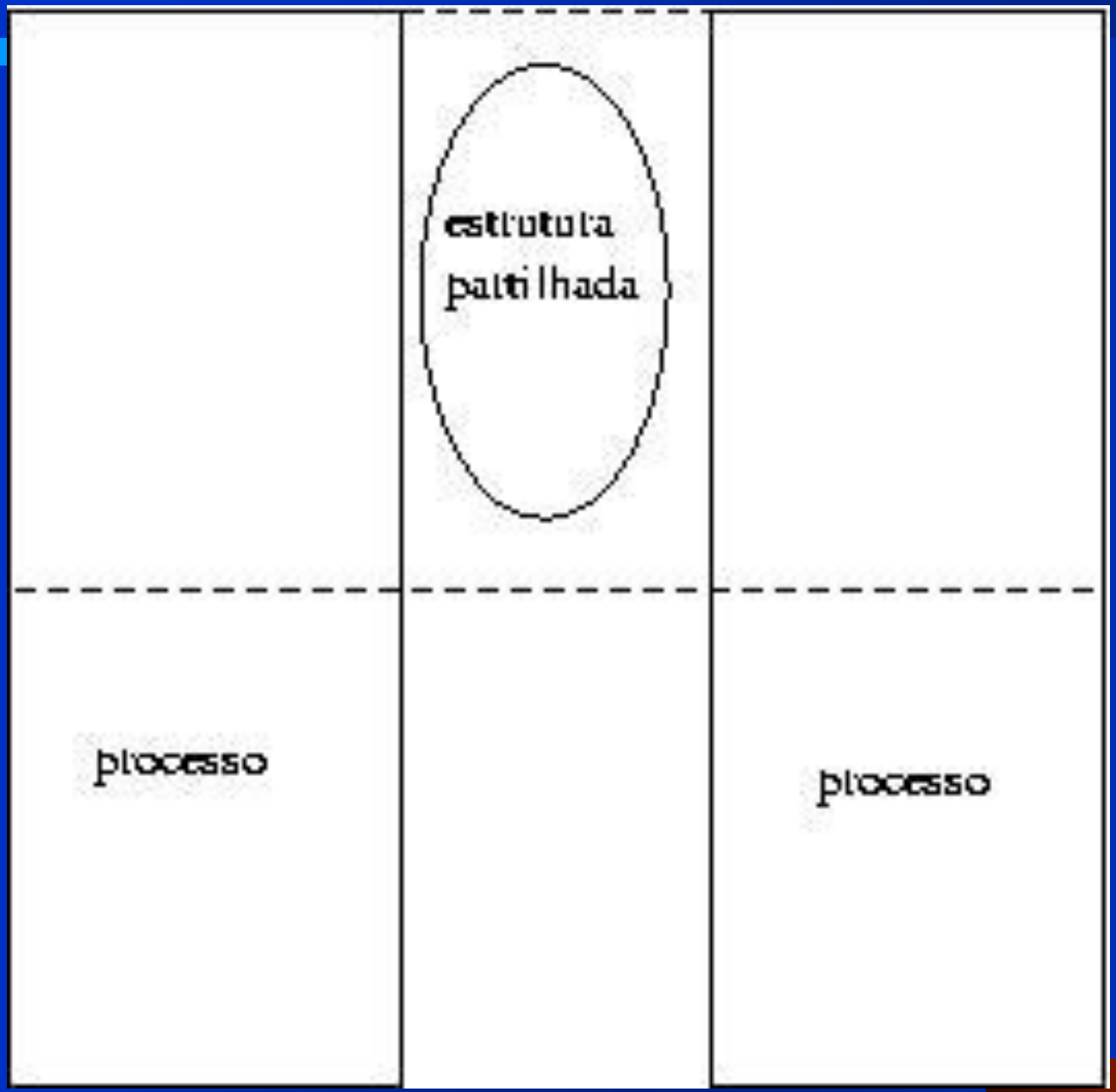
--- canais directos entre processos

→Leitura destrutiva (*bytes* ou mensagem desaparecem, após copiados para o destino)

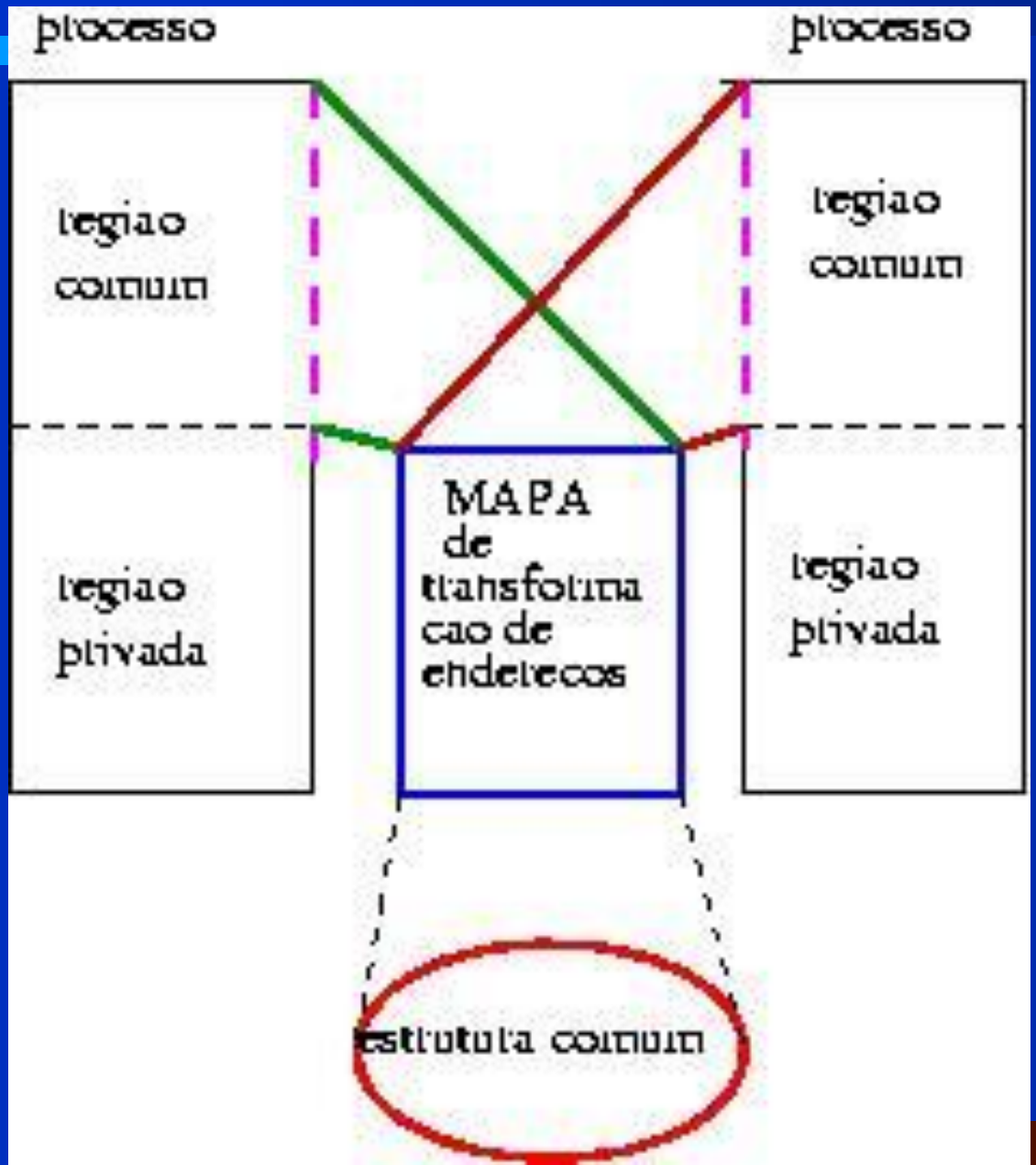
Arquitecturas Hardware



Memória Partilhada



Memória Partilhada



Memória partilhada no Unix ?

- **Integrada no conjunto de mecanismos conhecidos por IPC - Unix System V**

IPC = InterProcess Communication

Que abrange <u>três tipos de mecanismos</u>:

1-- Shared Memory : memória partilhada

2-- Semaphores : semáforos

(3-- Message Queues : filas de mensagens)

Para sistemas MONOPROCESSADORES ou MULTIPROCESSADORES de Bus Comum com Memória Partilhada

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

Para comunicar por Memória Partilhada:

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

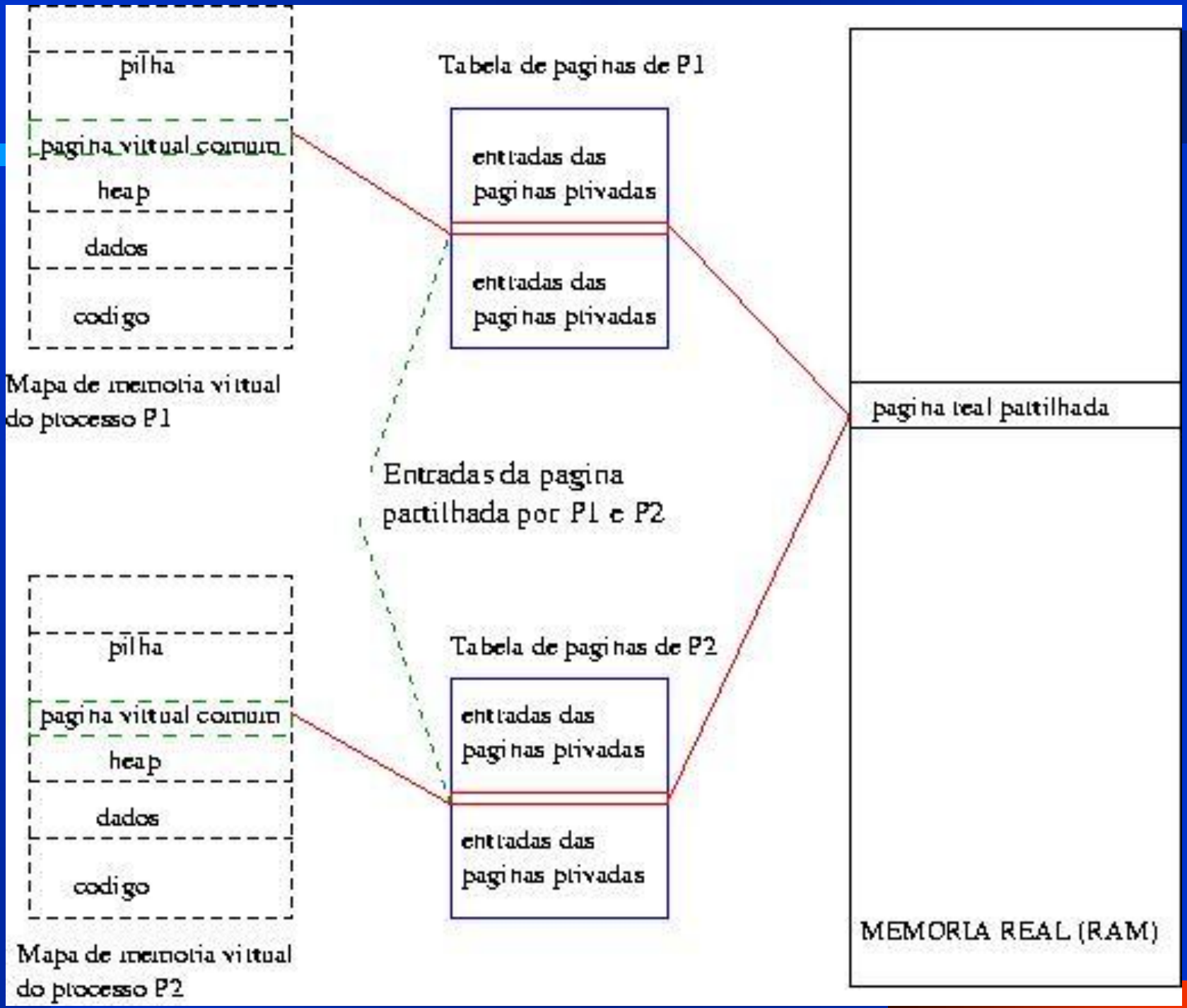
int shmget(key_t Key, int Size, int Flags);

-- devolve um valor inteiro - id único: *shmid*

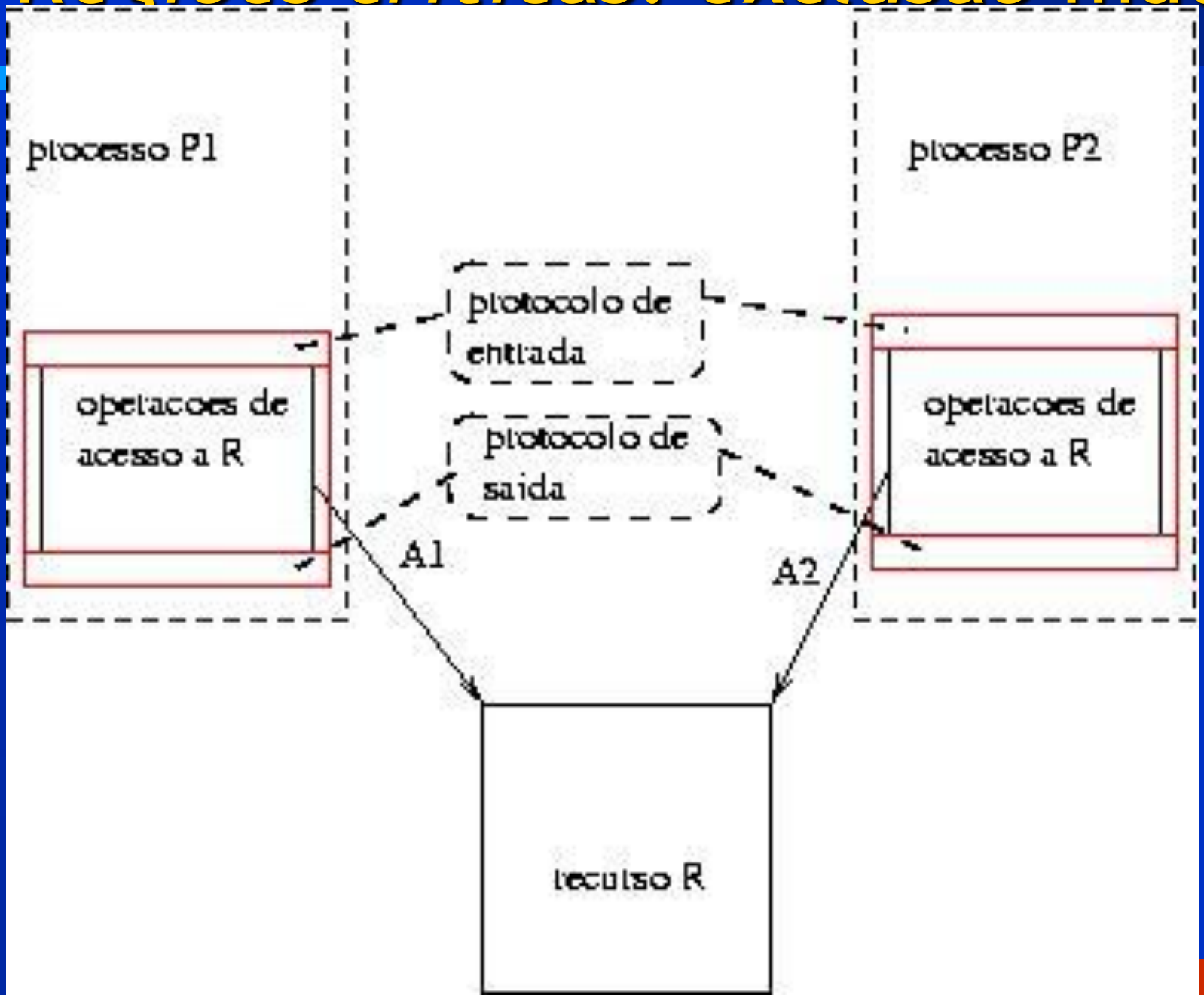
2- Cada processo deve associar (ATTACH) uma zona do seu mapa virtual àquela zona de memória física

*char *shmat(int shmid, …, …);*

-- devolve um apontador para o início da zona partilhada


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)

Regiões críticas: exclusão mútua



Processos concorrentes podem comunicar

- **Quando comunicam por Memória Partilhada:**

-- ler e escrever dados de memória

-- sem haver cópia de bytes

-- mas exige *sincronização explícita*

-- exige um bus comum de acesso a memória

- **Quando comunicam por Memória Distribuída**

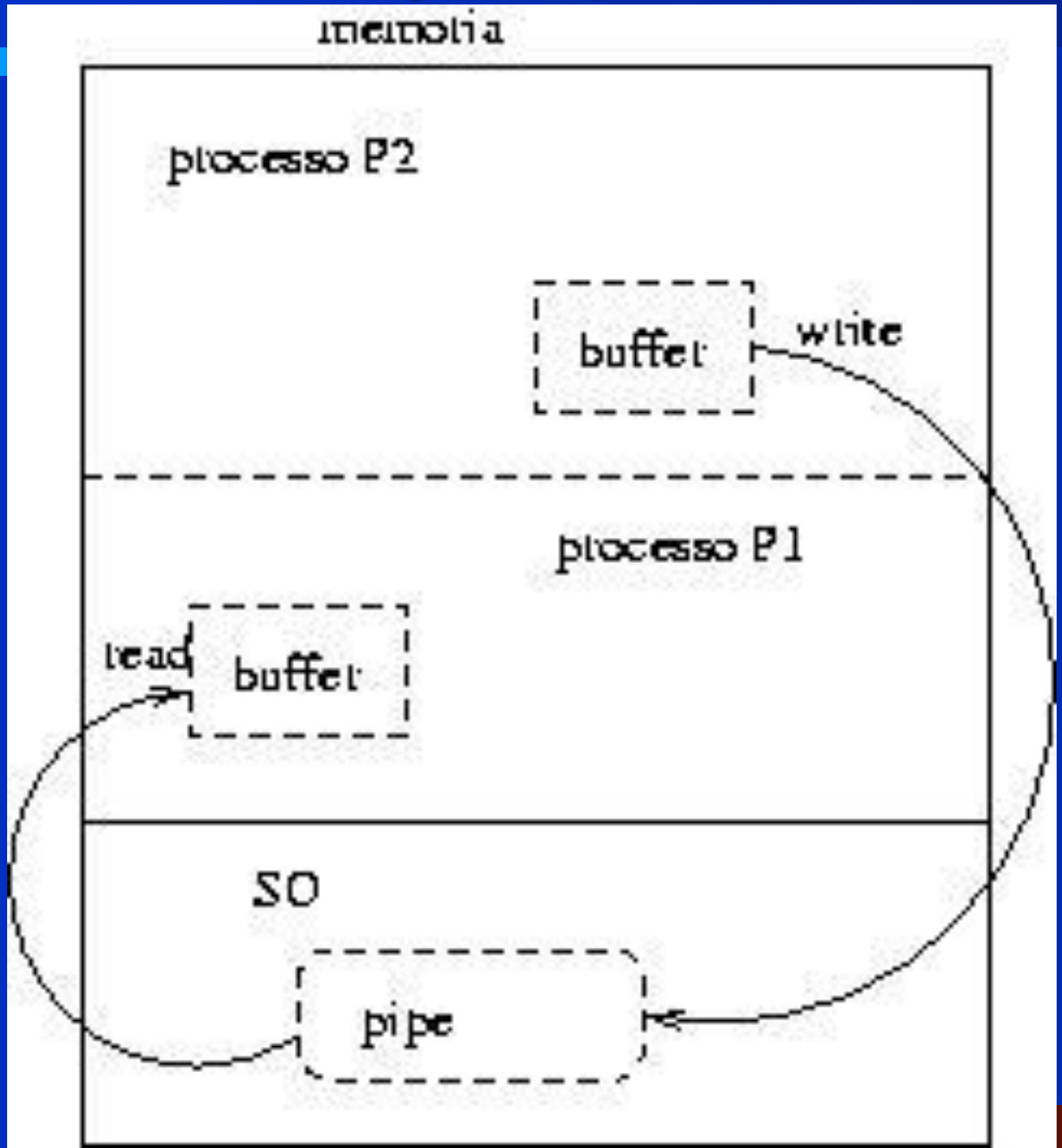
-- enviar e receber *bytes* ou mensagens

-- com cópia de bytes

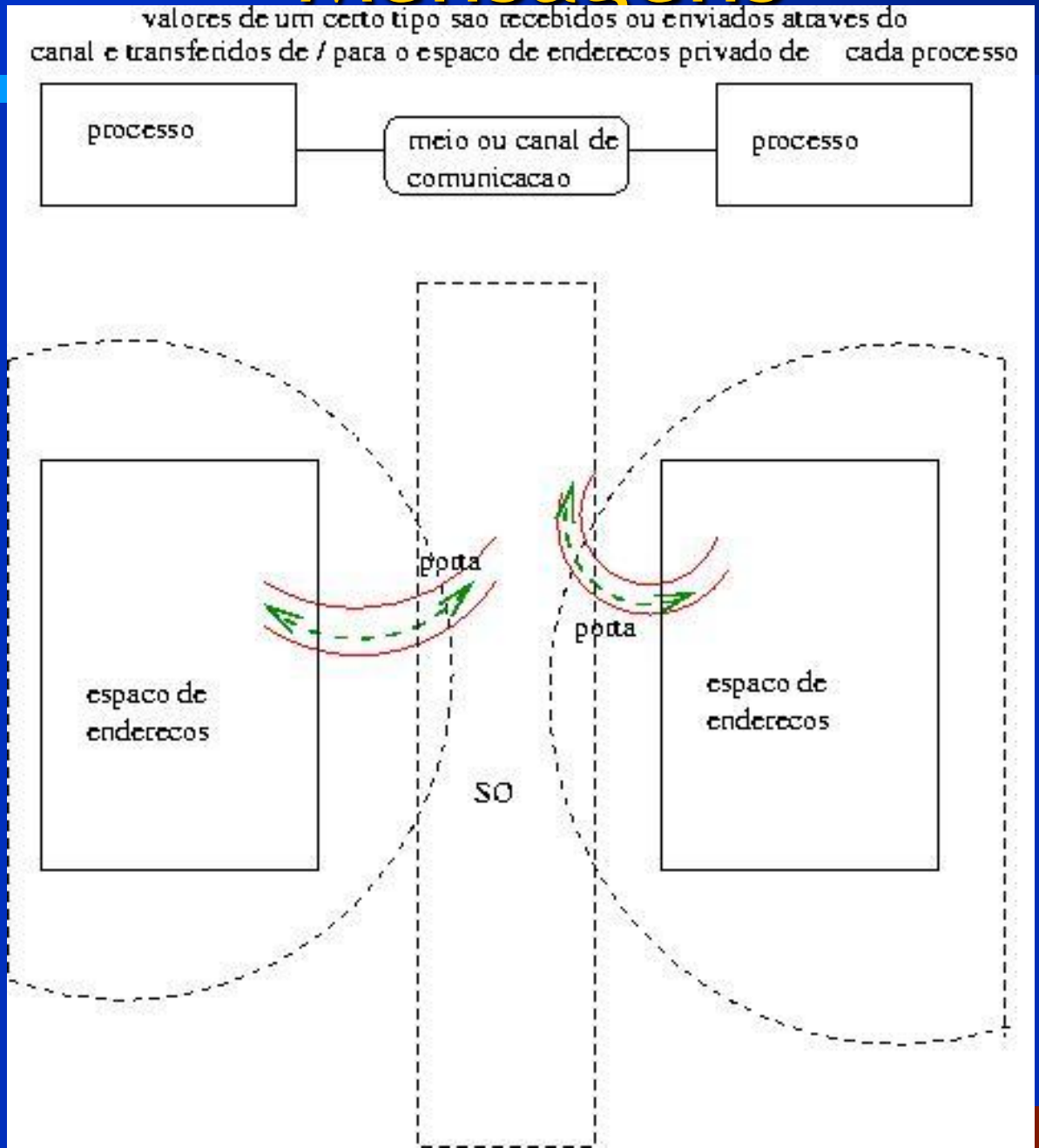
com *sincronização implícita*

-- pode ser usado num computador ou em rede

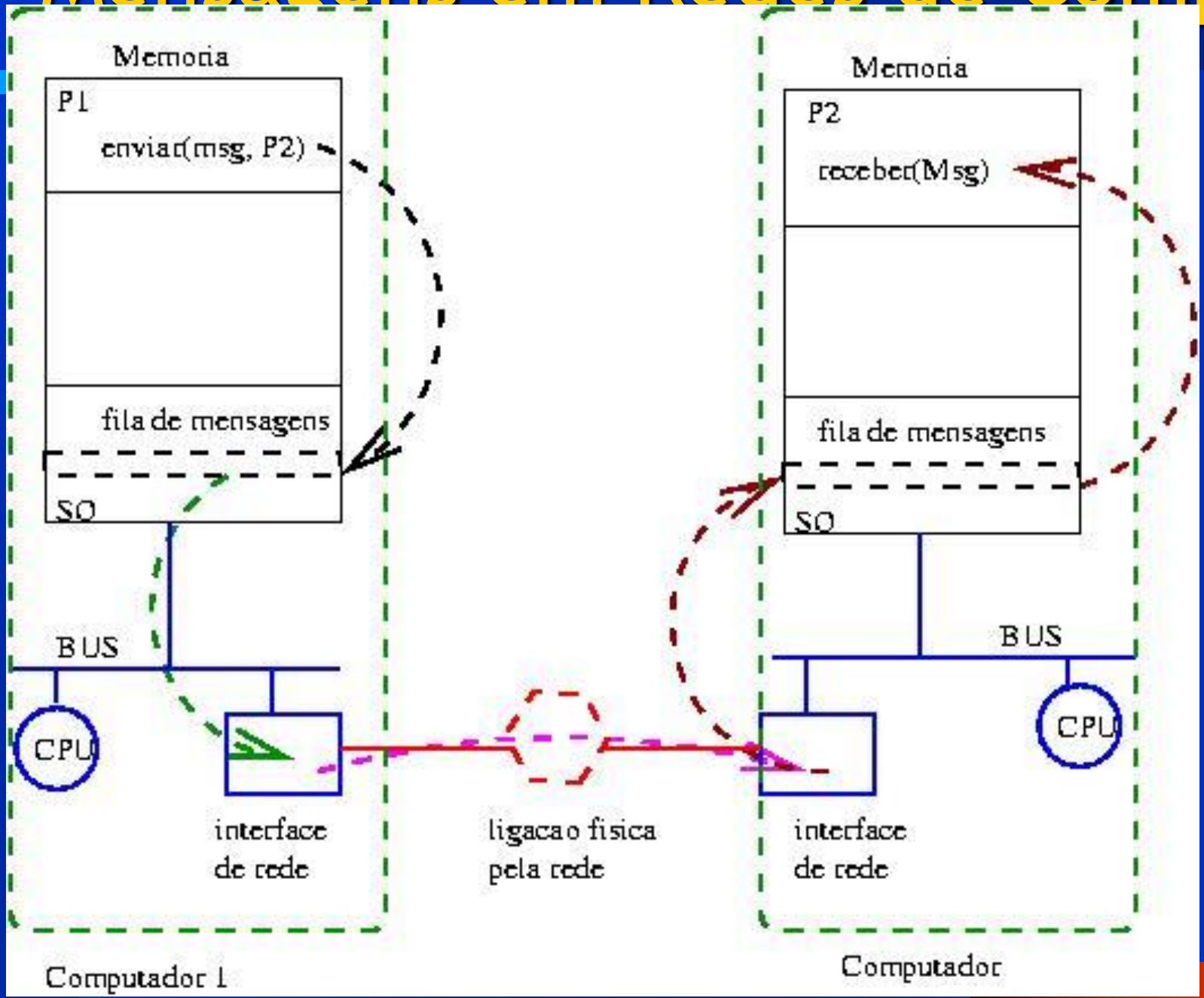
Comunicação por *Pipes Unix*



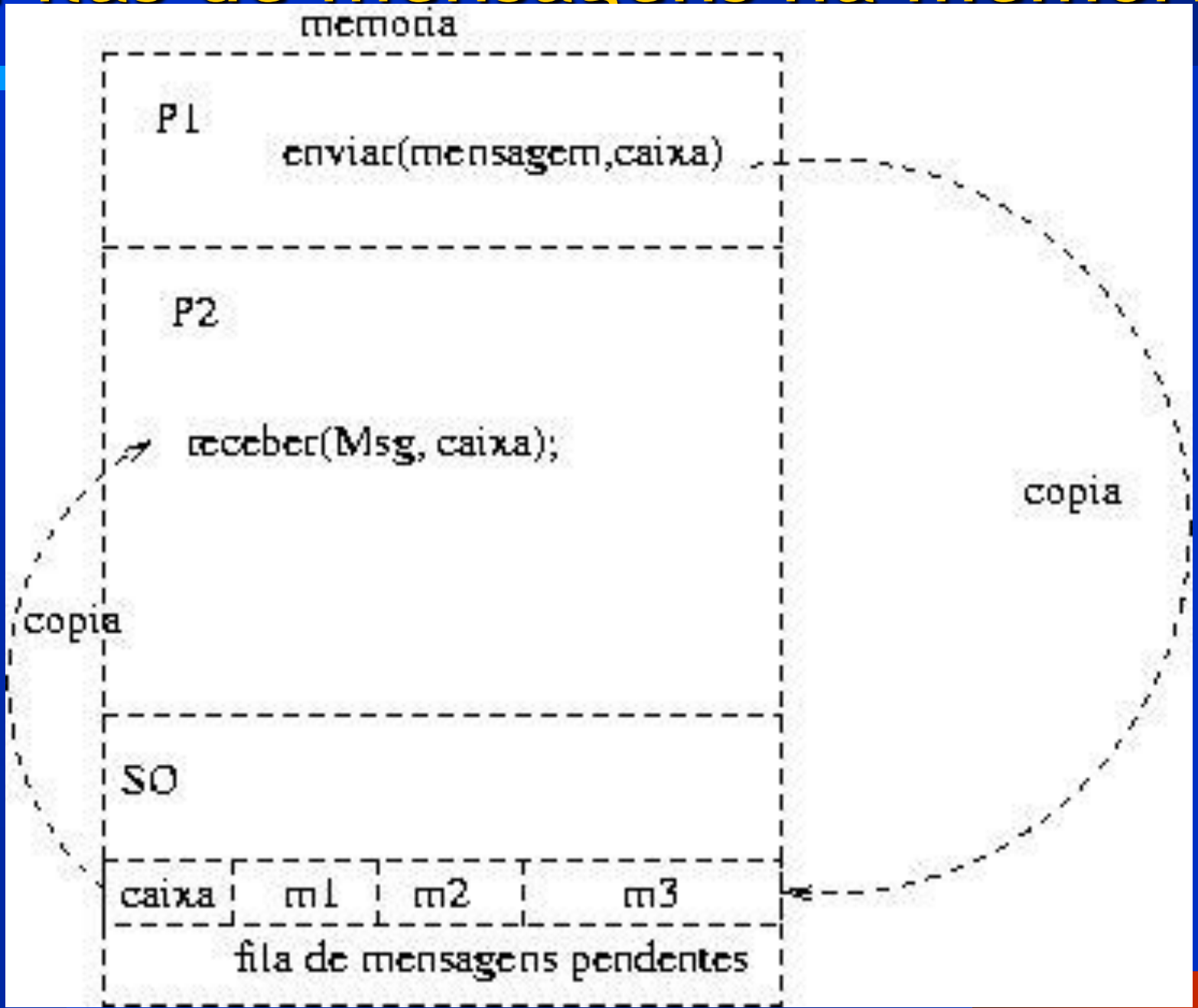
Mensagens


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

Mensagens em Redes de Comp.



Filas de mensagens na memória



Realização de comunicação por mensagens com base em memória partilhada e semáforos, num monoprocessador ou num multiprocessador com memória física partilhada.

Comunicação por mensagens

- **Envolve cópia de bytes de um espaço de endereços de um processo para outro**
- **Pode realizar-se:**

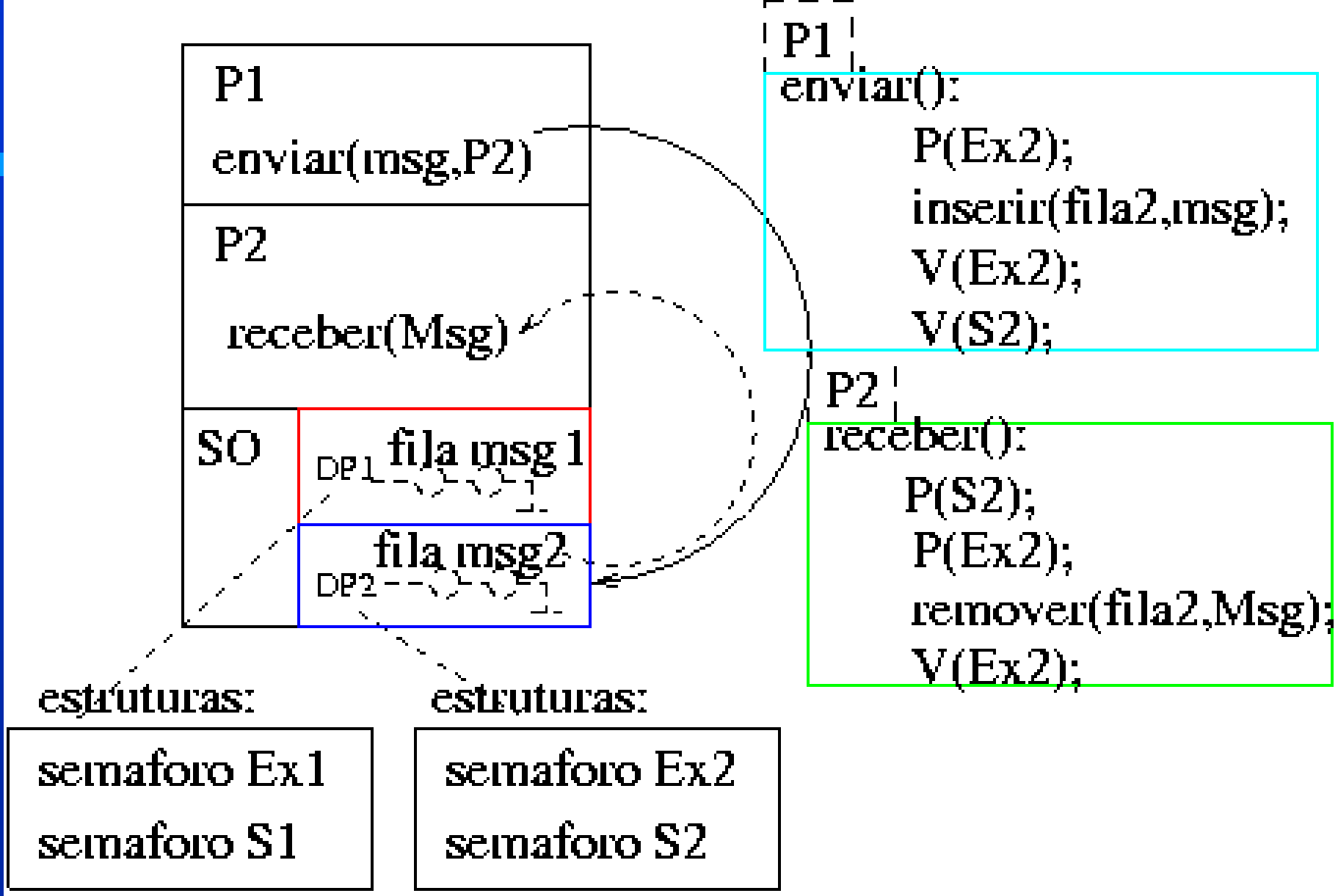
-- em mono- ou multiprocessadores com memória partilhada

-- em sistemas de memória distribuída (redes de computadores)

Em sistemas de memória partilhada? Exemplo:

-- uma fila de mensagens por cada processo

-- operações *ENVIAR(msg, processo)*
RECEBER(msg)



Ex1 e Ex2 inicial 1

S1 e S2 inicial 0

Dimensões da comunicação por mensagens

Dimensões da comunicação por mensagens

1. **Invocação explícita ou implícita**

Dimensões da comunicação por mensagens

1. **Invocação explícita ou implícita**
2. **Nomeação directa ou indirecta**

Dimensões da comunicação por mensagens

1. **Invocação explícita ou implícita**
2. **Nomeação directa ou indirecta**
3. **Interacção 1-1, 1-N, N-1, N-M**

Dimensões da comunicação por mensagens

1. **Invocação explícita ou implícita**
2. **Nomeação directa ou indirecta**
3. **Interacção 1-1, 1-N, N-1, N-M**
4. **Sincronização: operações bloqueantes/não**

Dimensões da comunicação por mensagens

1. **Invocação explícita ou implícita**
2. **Nomeação directa ou indirecta**
3. **Interacção 1-1, 1-N, N-1, N-M**
4. **Sincronização: operações bloqueantes/não**
5. **Fluxo da informação entre os processos**

Dimensões da comunicação por mensagens

1. **Invocação explícita ou implícita**
2. **Nomeação directa ou indirecta**
3. **Interacção 1-1, 1-N, N-1, N-M**
4. **Sincronização: operações bloqueantes/não**
5. **Fluxo da informação entre os processos**
6. **Não-determinismo: comunicações selectivas**

Dimensões da comunicação por mensagens

1. **Invocação explícita ou implícita**
2. **Nomeação directa ou indirecta**
3. **Interacção 1-1, 1-N, N-1, N-M**
4. **Sincronização: operações bloqueantes/não**
5. **Fluxo da informação entre os processos**
6. **Não-determinismo: comunicações selectivas**
7. **Papel dos participantes: simétrico ou assimétrico: clientes e servidores**

1- Invocação explícita ou implícita

Processo P1

```
enviar(msg,P2);
```

Processo P2

```
receber(Msg, P1);
```

- Ambos têm de invocar explicitamente, nos seus programas, as primitivas para enviar e para receber.
- Uma variante é o Processo receptor (P2) só fazer:

```
receber(Msg, Qualquer);
```

-- podendo receber de qualquer processo

Invocação implícita no receptor

Processo P1

enviar(msg,P2);

Processo P2

.....

Invocação implícita no receptor

Processo P1

enviar(msg,P2);

Processo P2

.....

- **O receptor, no início do seu programa, declara uma função (*handler*) que deve ser invocada automaticamente quando uma mensagem lhe for entregue.**

Invocação implícita no receptor

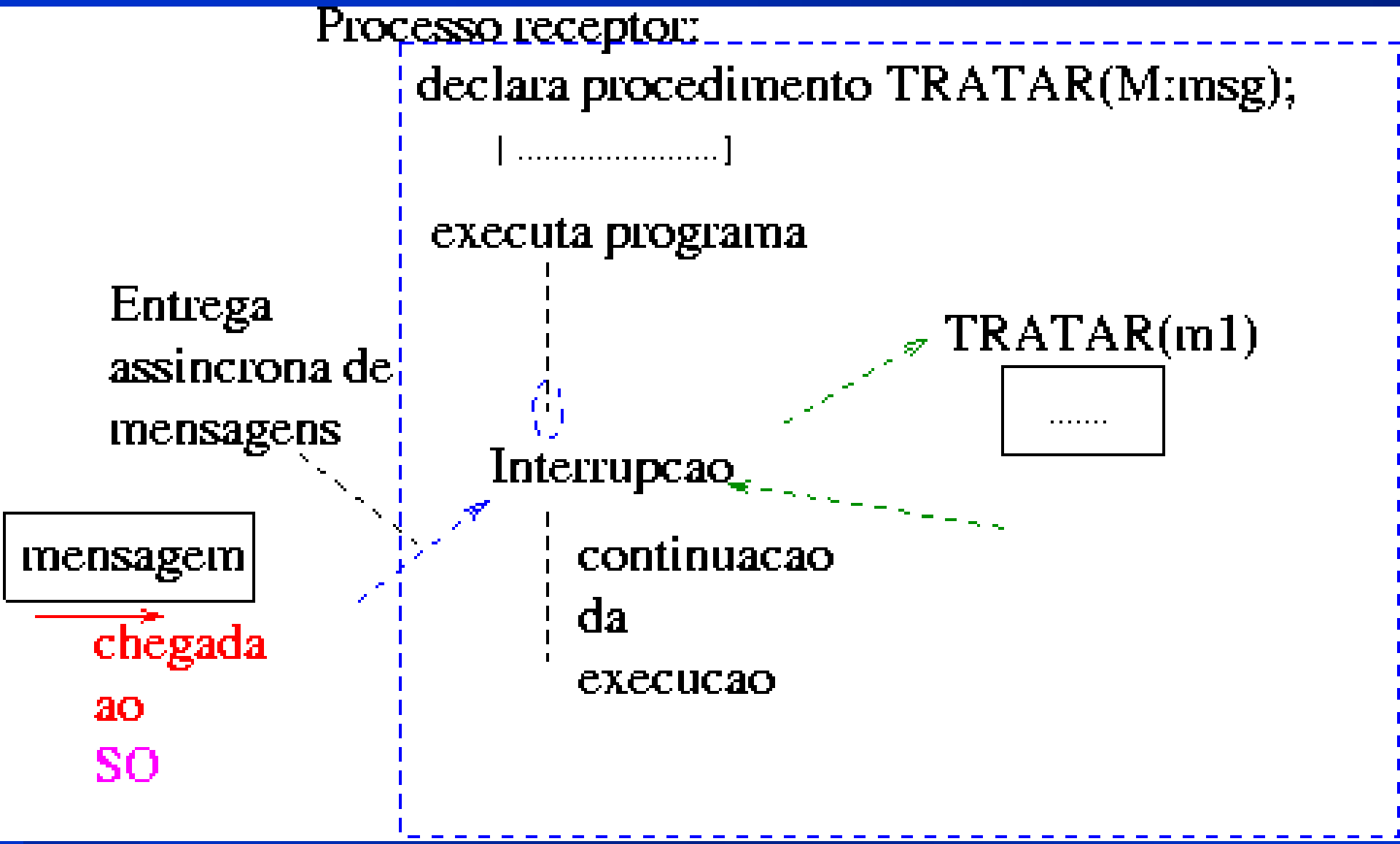
Processo P1

enviar(msg,P2);

Processo P2

.....

- **O receptor, no início do seu programa, declara uma função (*handler*) que deve ser invocada automaticamente quando uma mensagem lhe for entregue.**
- **A invocação da função *handler* é assíncrona, em relação à execução do programa do processo receptor**


Processo receptor:
declara procedimento TRATAR(M:msg);
[.....]
executa programa
Entrega
assincrona de
mensagens
TRATAR(m1)
.....
Interrupcao
mensagem
chegada
ao
SO
continuacao
da
execucao

- um modelo de programação **reactivo**:

- um modelo de programação **reactivo**:

-- um processo é levado a reagir (invocar o procedimento TRATAR), logo que uma mensagem lhe é entregue pelo SO.

- um modelo de programação **reactivo**:

-- um processo é levado a reagir (invocar o procedimento TRATAR), logo que uma mensagem lhe é entregue pelo SO.

-- como o processo não conhece o momento de chegada das mensagens, este modelo permite lidar com esse **não-determinismo**.

--

- um modelo de programação **reactivo**:

-- um processo é levado a reagir (invocar o procedimento TRATAR), logo que uma mensagem lhe é entregue pelo SO.

-- como o processo não conhece o momento de chegada das mensagens, este modelo permite lidar com esse **não-determinismo**.

-- porque, neste caso, o processo não saberia quando deveria invocar a operação de recepção de mensagens

--

- um modelo de programação **reactivo**:

-- um processo é levado a reagir (invocar o procedimento TRATAR), logo que uma mensagem lhe é entregue pelo SO.

-- como o processo não conhece o momento de chegada das mensagens, este modelo permite lidar com esse **não-determinismo**.

-- porque, neste caso, o processo não saberia quando deveria invocar a operação de recepção de mensagens

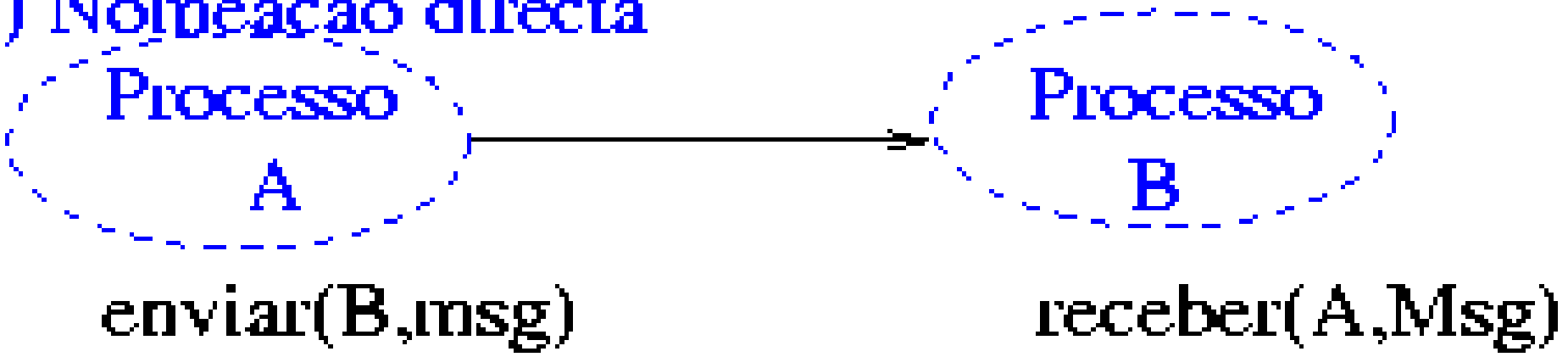
-- permite ao processo ir executando outras acções, enquanto a mensagem não chega

Caso particular

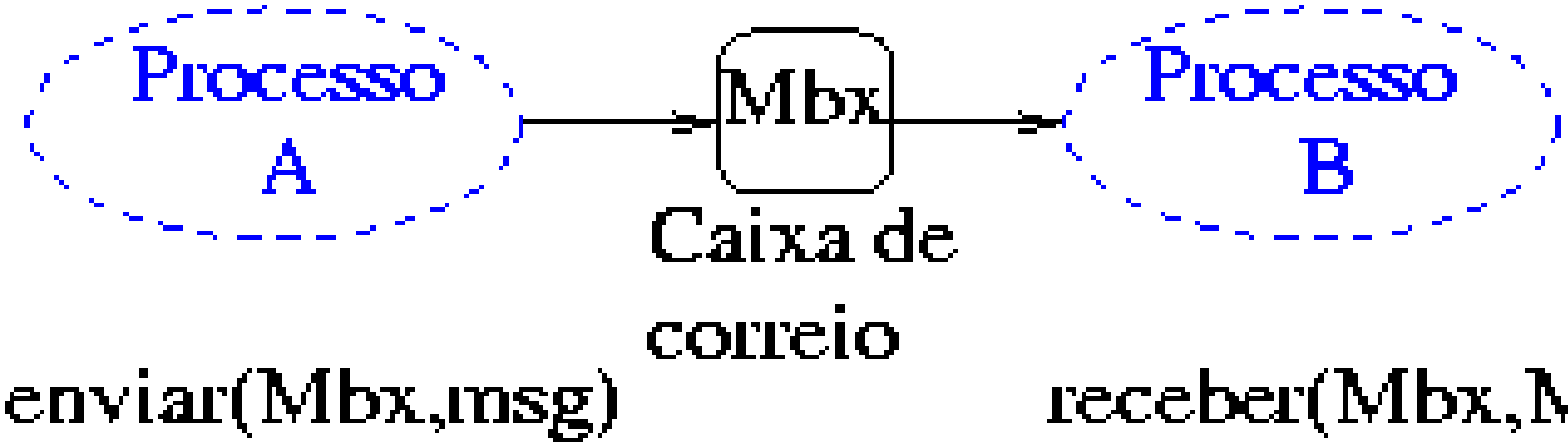
- **O mecanimo de notificação assíncrona de sinais no Unix é um exemplo (a ver nas próximas aulas)**

2 - Nomeação dos participantes

a) Nomeacao directa



b) Nomeacao indirecta



- Na **nomeação directa:**

-- os processos têm de se conhecer e co-existir simultaneamente

- Uma variante: o processo receptor (**um servidor**), não sabendo quem quer comunicar com ele, indica que pretende receber de um **qualquer emissor**:

receber(Msg, Qualquer)

Mas o processo emissor (cliente) tem de conhecer o nome do servidor (ou do 'serviço').

- **Na nomeação indirecta:**

-- existe um intermediário na comunicação entre os processos

-- cada processo dirige-se ao intermediário, para enviar ou receber mensagens

-- o intermediário, em geral, tem um papel passivo:

  - **limita-se a gerir filas de mensagens, actuando como um distribuidor indirecto e como uma espécie de *buffer activo***
  - **aceita as mensagens que lhe enviam e distribui-as pelos destinatários, quando estes lhas pedirem.**

- Na nomeação indirecta:

-- processos não precisam de co-existir simultaneamente no sistema, podem serem eventualmente executados em momentos diferentes

-- facilita a interacção entre processos, de uma forma anónima:

  qualquer processo pode enviar para a caixa

  qualquer processo pode receber qualquer mensagem da caixa sem ter de ser explicitamente nomeado e conhecido pelo emissor.

- Por exemplo, pode ser usado, para um qualquer processo no sistema, notificar todos os outros ou um subconjunto dos processos existentes, sem ter de os nomear e conhecer explicitamente.
- Qualquer novo processo que seja criado, pode começar a utilizar a caixa, desde que lhe tenha acesso.

Atributos genéricos das caixas / filas de mensagens

- um nome global, único no sistema
- permissões de acesso para protecção
- disciplinas de entrega das mensagens pendentes:
  - FIFO (first-in-first-out) ou outras
  - prioridades
  - conforme um tipo definido pelo emissor.
- operações de criação e destruição

3 - Caixas de correio globais e privadas

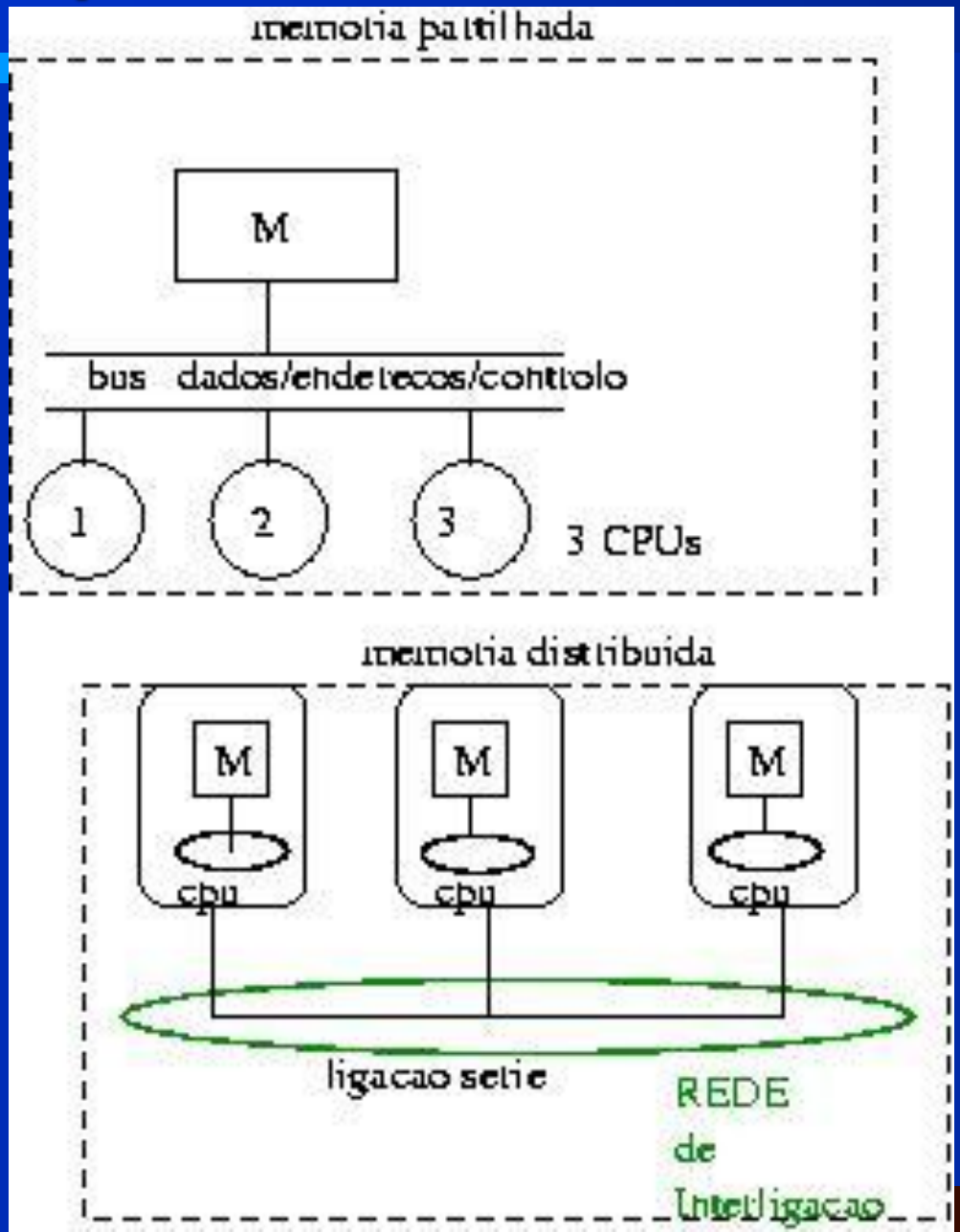
Em geral, uma fila de mensagens é suportada por estruturas internas em memória do SO

A cada momento, o SO sabe:

-- quantas e quais as mensagens pendentes

-- quais os processos que estão bloqueados a aguardar a entrega de mensagens

Arquitecturas Hardware



Modelo N*M: muitos-para-muitos

Em sistemas de memória partilhada:

- **a fila de mensagens admite um modelo de interacção N*M**
- **quaisquer N processos podem enviar mensagens**
- **quaisquer M processos podem receber mensagens, desde que tenham as devidas permissões de acesso.**

Em sistemas de memória distribuída:

- em que memória de que computador se deve reservar espaço para representar a fila de mensagens?

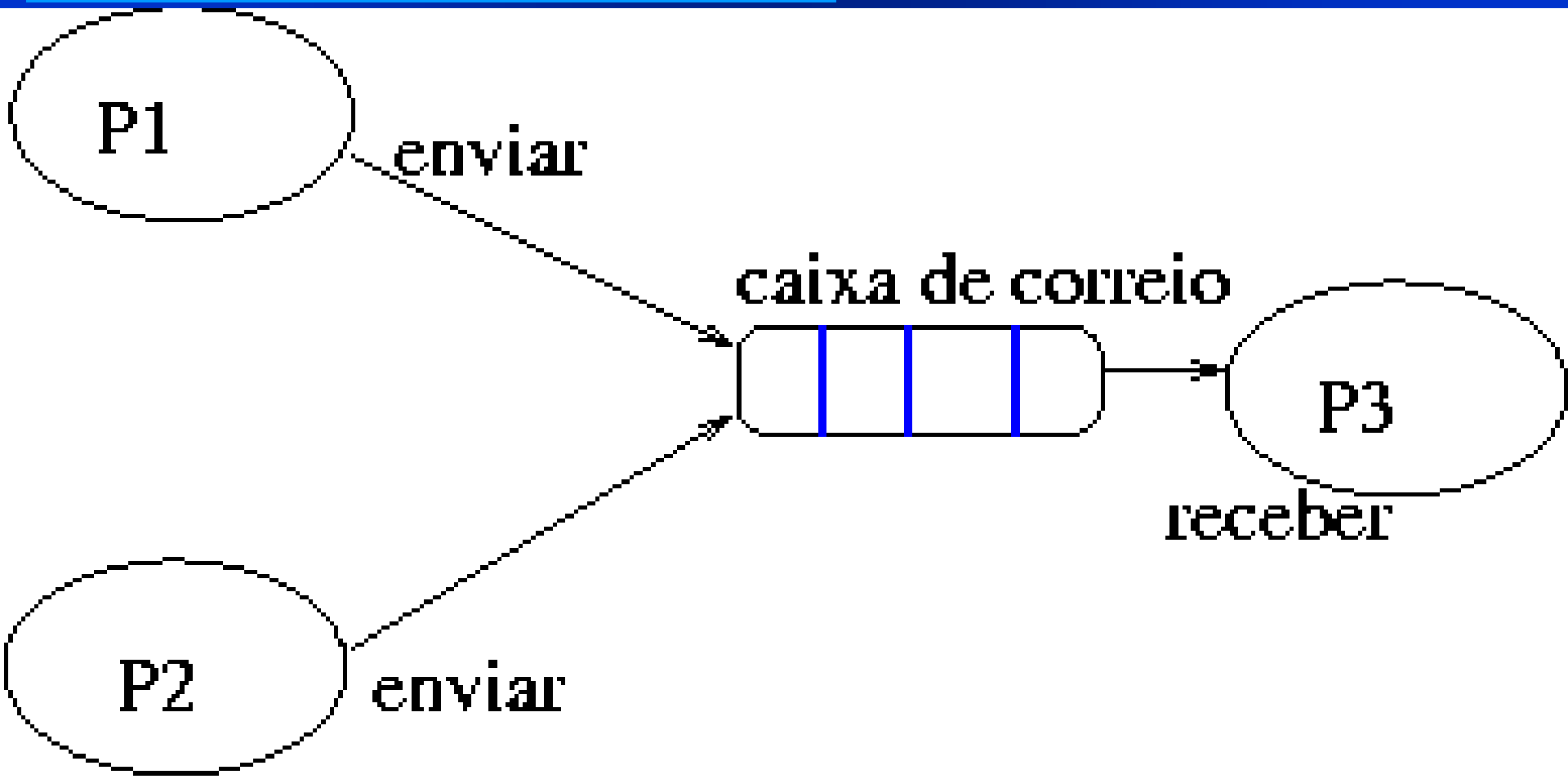
1. num computador determinado, um servidor centralizado, distribuidor de mensagens?
2. duplicada em cada computador onde haja potenciais participantes (isto é, emissores ou receptores de mensagens?

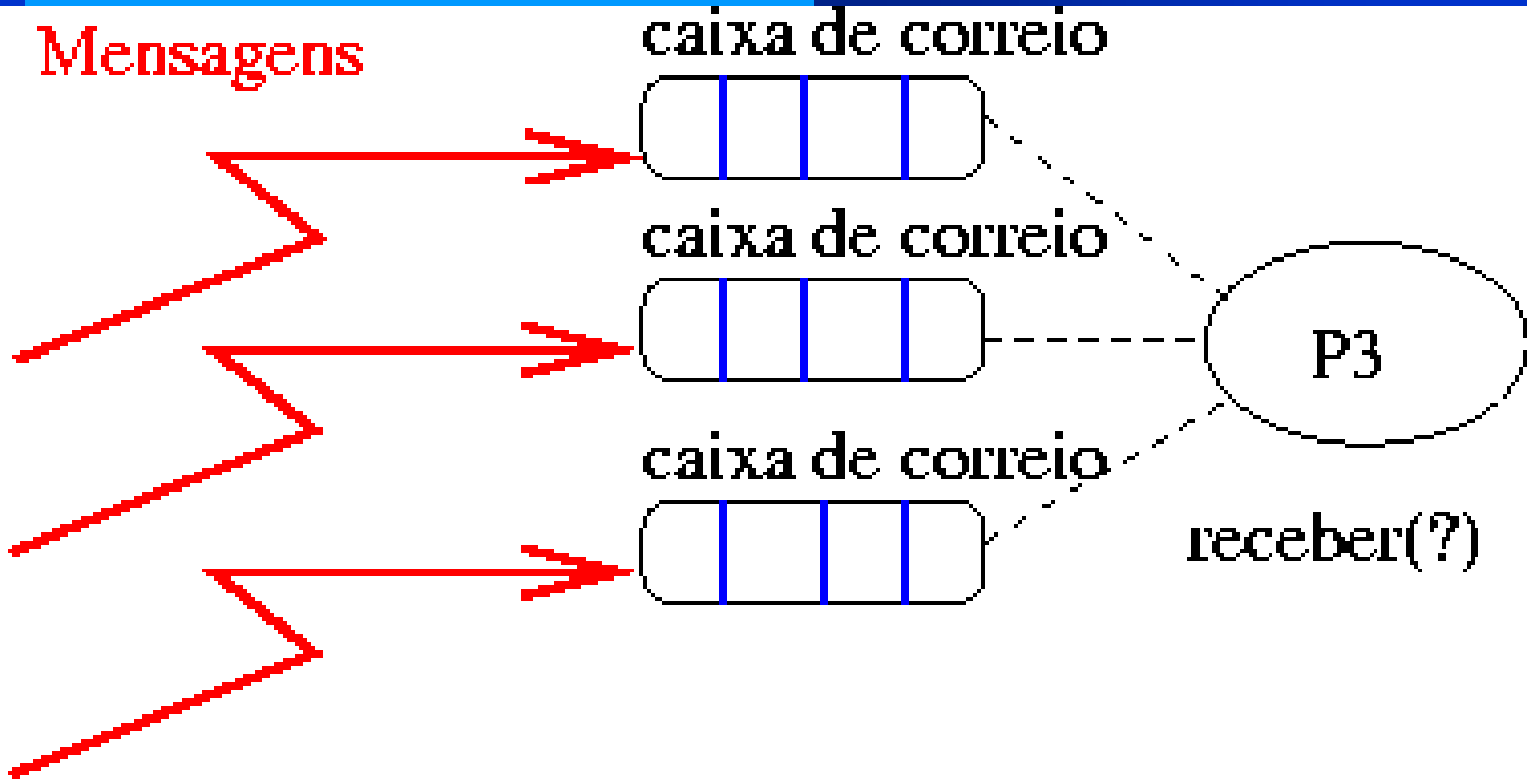
1. **Servidor centralizado: -- um ponto de 'engarrafamento' do sistema, onde afluem todos os pedidos de envio e recepção, mesmo vindo de máquina remotas (com tempos de transmissão de mensagens que podem ser apreciáveis);**
2. **Duplicação da fila em todos: -- há que manter cada cópia actualizada em cada computador, sempre que algum processo envia ou recebe (difícil, porque os tempos de transmisão de mensagens não são desprezáveis).**

Modelo N*1: muitos-para-um

- um modelo limitado, em que a fila de mensagens passa a ser 'privada‘
- quaisquer N processos podem enviar mensagens,
- mas apenas 1 pode receber: este é em geral o dono/criador da fila de mensagens
- num sistema distribuído:

-- mais fácil decidir em que computador se deve localizar a fila: no computador onde se executa o processo dono da fila.


P1
enviar
caixa de correio
P3
receber
P2
enviar


Mensagens
caixa de correio
caixa de correio
caixa de correio
P3
receber(?)

Não-determinismo na chegada de mensagens

- **o processo tem o problema de, não sabendo a ordem pela qual as mensagens chegam às suas diversas caixas, ter de decidir:**

-- de que forma / por que ordem deve aguardar pela chegada de mensagens?

(a). Aguarda, bloqueando-se primeiro à espera de mensagens na caixa C1, por exemplo, invocando receber(C1, Msg)?

Mas, nesse caso, pode ficar eternamente bloqueado, se 'nunca' chegarem mensagens a C1; e se, entretanto, forem chegando mensagens às outras caixas, estas não são nunca recebidas;

(b). Não aguarda nunca em nenhuma caixa, fazendo, testes sem bloqueio, para verificar se, em cada caixa, existem ou não mensagens pendentes para entrega:

testar-mensagens(Caixa-i,Resposta);

em que Resposta indica se *Sim* ou *Não*;

Desperdício de tempo de CPU, por espera activa:

-- não havendo mensagens em nenhuma caixa, o processo fica em ciclo de testes sucessivos, até chegar alguma mensagem…

(modelo *IPC_NOWAIT* no Unix em *msgrcv()*)

(c). Aguarda, bloqueando-se, mas sem se comprometer com nenhuma caixa específica:

receber-qualquer(Msg, Caixa);

em que *Caixa* devolve o identificador da 'primeira' caixa na qual surgiu uma mensagem, sendo esta devolvida em Msg;

-- o SO é que fica responsável por implementar esta operação

(exemplo: operação ***select()*** no Unix)

(d). Declara uma função de tratamento (*handler*) para ser invocada, de forma assíncrona, logo que uma mensagem chegue a qualquer das caixas

-- pode declarar uma função *handler* individual para cada uma das caixas

(d). Declara uma função de tratamento (*handler*) para ser invocada, de forma assíncrona, logo que uma mensagem chegue a qualquer das caixas

-- pode declarar uma função *handler* individual para cada uma das caixas

Neste caso, o processo pode ir executando um programa qualquer e ser automaticamente interrompido quando uma mensagem chegar, para a ir tratar,

e depois regressa ao ponto em que foi interrompido.

(e) Múltiplos *threads* são criados dinamicamente, isto é, à medida que as mensagens vão chegando, para executarem as funções *handlers*

→ A estudar mais adiante… como aplicação dos modelos de *Threads*

Modelo 1*N: um-para-muitos

- **Comunicação por difusão de mensagens (*broadcast*):**

-- um processo envia uma mesma mensagem para todos os processos presentes no sistema, num dado momento.

- **Difusão por *multicast*:**

-- enviar uma mensagem para os processos que pertencem a um dado grupo de processos

Modelos 1*N

Os destinatários não são nomeados individualmente mas sim indirecta e implicitamente:

-- como um todo (broadcast)

ou

-- pelo nome do grupo ao qual pertencem (multicast)

Modelos 1*N

Os destinatários não são nomeados individualmente mas sim indirecta e implicitamente:

-- como um todo (broadcast)

ou

-- pelo nome do grupo ao qual pertencem (multicast)

É o SO que sabe/gere a cada momento:

-- quais os processos no sistema ou que pertencem a cada grupo, para propagar as mensagens para todos eles.

O programador não se preocupa com esta gestão.

4 - Sincronização do envio com a recepção

- **(a) Envio *não bloqueante ou assíncrono*: o emissor não espera por qualquer confirmação de que a mensagem tenha sido recebida.**

- **(b) Envio bloqueante até que o emissor receba uma confirmação de que a mensagem foi efectivamente recebida → *Envio síncrono***

Envio assíncrono

- **No modelo assíncrono, o emissor evolui ao seu próprio ritmo, recorrendo ao SO para armazenar temporariamente, em *buffers* seus, as mensagens pendentes para serem entregues.**

Envio assíncrono

- **No modelo assíncrono, o emissor evolui ao seu próprio ritmo, recorrendo ao SO para armazenar temporariamente, em *buffers* seus, as mensagens pendentes para serem entregues.**
- **Permite maior concorrência nas acções:**

-- os emissores não têm de esperar pelos receptores.

- **Os emissores não têm confirmação de que as suas mensagens sejam recebidas.**

Envio assíncrono

- **No modelo assíncrono, o emissor evolui ao seu próprio ritmo, recorrendo ao SO para armazenar temporariamente, em *buffers* seus, as mensagens pendentes para serem entregues.**
- **Permite maior concorrência nas acções:**

-- os emissores não têm de esperar pelos receptores.

- **Os emissores não têm confirmação de que as suas mensagens sejam recebidas.**

-- pode tornar mais difícil compreender a evolução da execução dos programas

Recepção

- **(a) Recepção bloqueante: receber() bloqueia o processo até que lhe seja entregue uma mensagem (no Unix System V, é o modo pré-definido (IPC_WAIT na operação msgrcv())**

Recepção

- **(a) Recepção bloqueante: receber() bloqueia o processo até que lhe seja entregue uma mensagem (no Unix System V, é o modo pré-definido (IPC_WAIT na operação msgrcv())**
- **(b) Recepção não bloqueante: receber() devolve um indicador se não há mensagens pendentes e retorna de imediato ao invocador;**

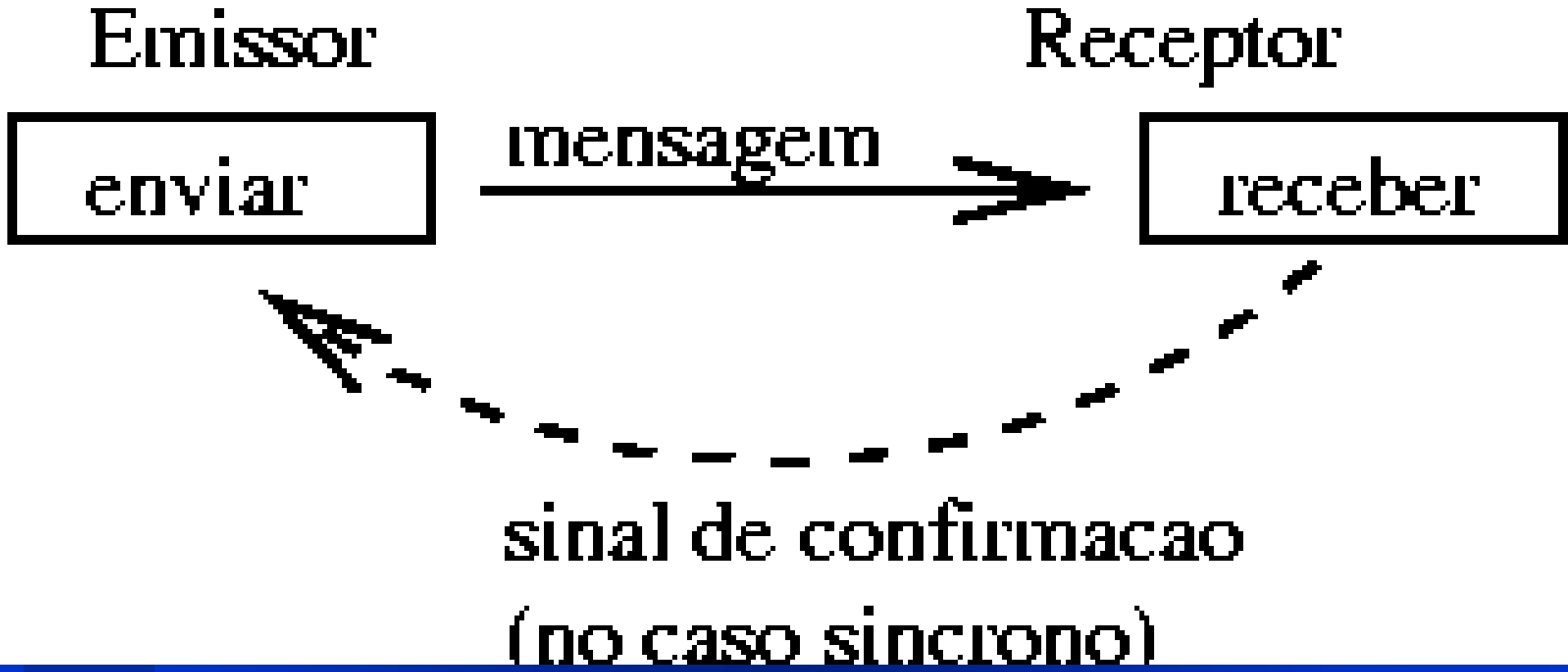
Recepção

- **(a) Recepção bloqueante: receber() bloqueia o processo até que lhe seja entregue uma mensagem (no Unix System V, é o modo pré-definido (IPC_WAIT na operação msgrcv())**
- **(b) Recepção não bloqueante: receber() devolve um indicador se não há mensagens pendentes e retorna de imediato ao invocador;**
- **(c) Recepção não bloqueante com notificação assíncrona: o processo receptor declara um handler de tratamento que será invocado quando chegar alguma mensagem.**

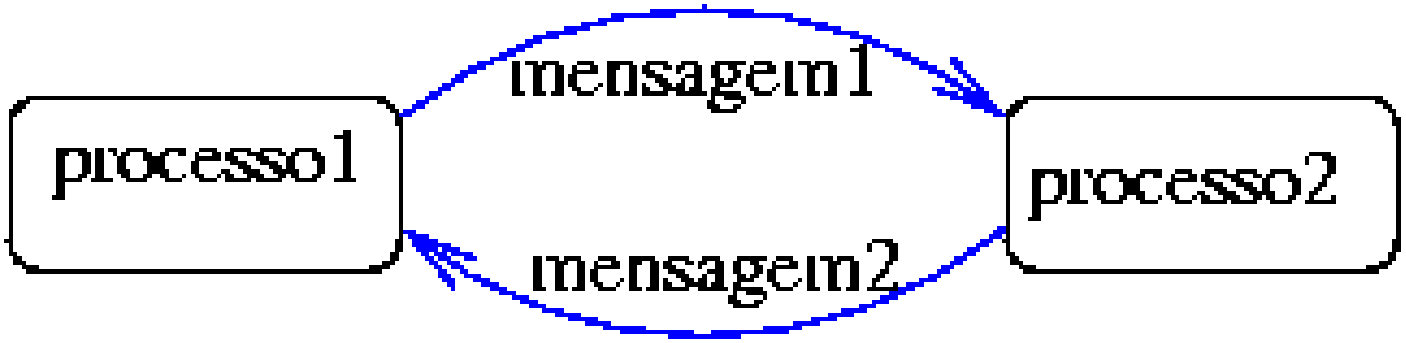
Envio e recepção síncronos

- **Tanto o emissor como o receptor, aguardam um pelo outro, até se poder transmitir a mensagem, só prosseguindo depois disso.**
- **No modelo totalmente síncrono de comunicação, os processos emissor e receptor evoluem 'passo a passo', trocando sinais de controlo (confirmação de recepção), sempre que há mensagens trocadas entre eles.**

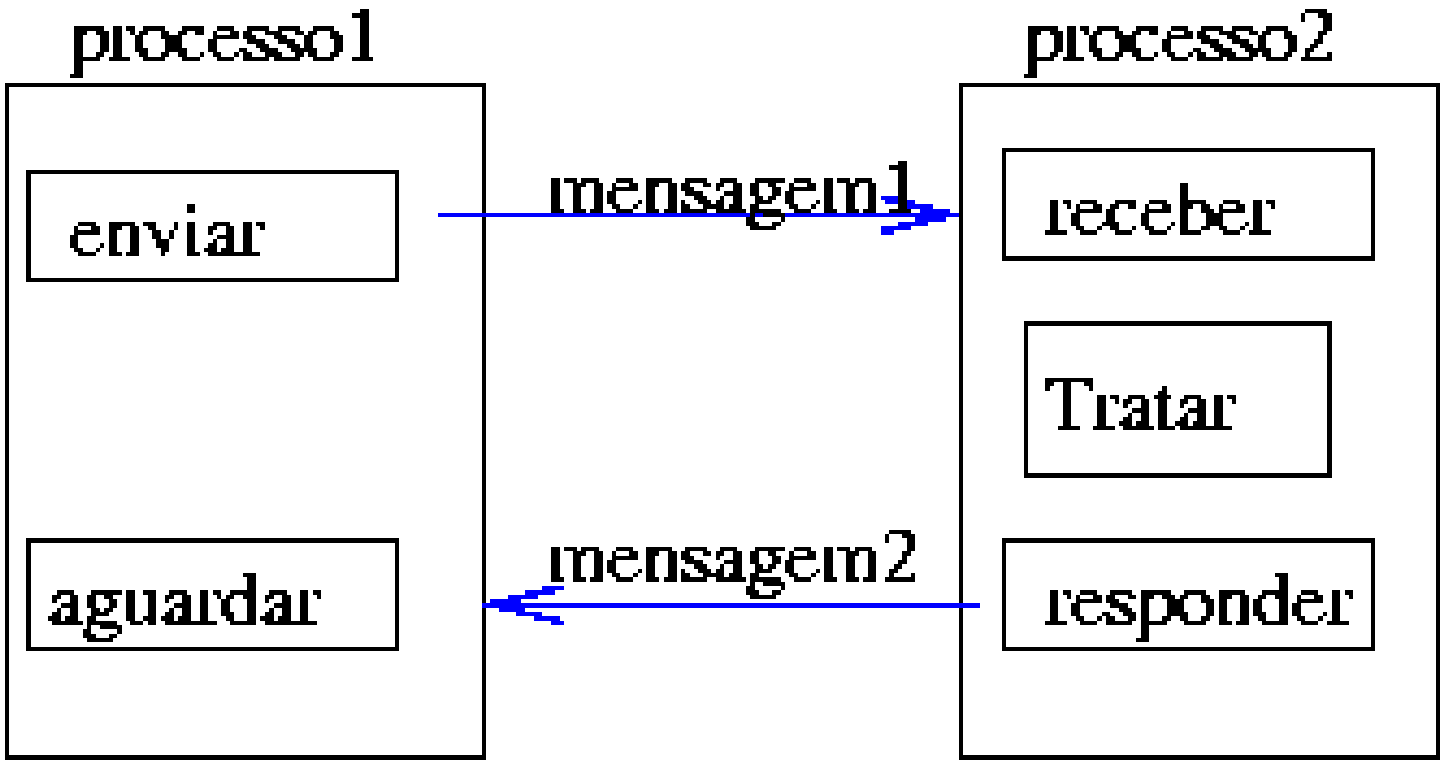
5 - Fluxo unidireccional da informação entre processos



a) Comunicaçao bidireccional simples



b) Comunicaçao bidireccional com tratamento



Comunicação bidireccional (a)

- **(a) Simples troca de duas mensagens, num mesmo acto de comunicação, em geral associado a um ponto de encontro (´´rendez-vous´´) dos dois processos envolvidos**

enviar_e_receber(m1,M2);

Comunicação bidireccional (a)

- (a) **Simples troca** de duas mensagens, num mesmo acto de comunicação, em geral associado a um ponto de encontro (´´**rendez-vous**´´) dos dois processos envolvidos

enviar_e_receber(m1,M2);

-- o primeiro a chegar bloqueia-se até que o segundo chegue e, então, trocam as mensagens. É um modelo simétrico.

- <u>Pode ser realizado com base no modelo unidireccional</u>

Comunicação bidireccional (b)

- **(b) Uma troca de duas mensagens: um cliente envia uma mensagem e um servidor, uma vez recebida a mensagem do cliente, efectua um processamento local e só depois responde, com o envio de uma mensagem ao cliente:**

Comunicação bidireccional (b)

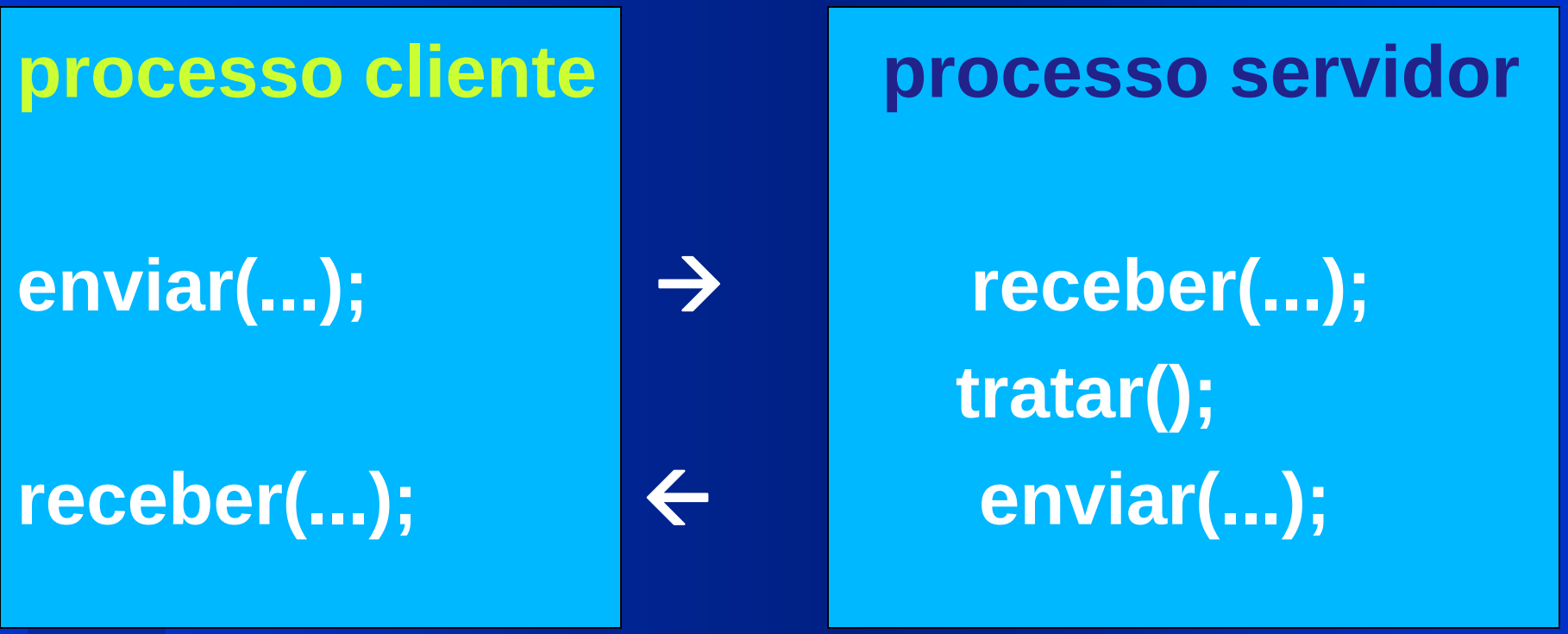
- (b) **Uma troca de duas mensagens**: um cliente envia uma mensagem e um servidor, uma vez recebida a mensagem do cliente, efectua um processamento local e só depois responde, com o envio de uma mensagem ao cliente:

-- o cliente bloqueia-se até que o servidor ´´chegue´´, receba a mensagem do primeiro, trate e responda.

-- é uma interacção assimétrica

- Pode ser realizado com base no modelo unidireccional

Interacção Cliente-Servidor



Mecanismos IPC - InterProcess Communication Unix System V (consultar: Unix System Programming secção 8.3 e manual do Unix)

- São mecanismos que não existiam na versão do Unix e foram incorporados como extensões definidas no Unix System V desenvolvido na AT&T
- Tentam unificar as operações sobre 3 tipos de entidades IPC muito diferentes:
  - regiões de memória física partilhada
  - semáforos
  - filas de mensagens (message queues)
- Só são realizados em sistemas monoprocessadores ou multiprocessadores de memória partilhada (i.e. não em redes, para as quais existem as extensões ao Unix BSD desenvolvidas na Univ. de Berkeley)

- As entidades dos 3 tipos são designadas por **identificadores, globais e únicos** em todo o sistema, que correspondem a números chamados KEY (chaves) de um tipo pré-definido "**key_t**" pelo sistema.

- São manipulados por 3 classes de operações:

-- **GET**: cria ou dá acesso a uma entidade e devolve um identificador ID inteiro, que será usado nos acessos posteriores.

-- **CTL**: controlo, obtenção de informação ou definição de parâmetros que configuram o estado da entidade.

-- **OP**: outras operações, específicas a cada tipo de entidade, para comunicação ou sincronização.

- As chaves KEY devem ser escolhidas por mútuo acordo ou convenção entre os processos concorrentes que cooperam numa aplicação.
- Para comunicarem entre si, todos os processos devem conhecer e utilizar um mesmo identificador de acesso à entidade IPC.
- As entidades IPC não têm contadores de referência (ou de utilizadores) como acontece com os 'file descriptors' dos canais Unix
- As entidades IPC têm de ser explicitamente destruídas, com operações do tipo CTL (controlo) ou com comandos do "shell" (*ipcrm* – IPC REMOVE), caso contrário só desaparecem quando da reinicialização ("*reboot*") do sistema de operação.

Comandos do Shell

- ***ipcs*: (IPC Status) informa sobre o estado corrente das entidades IPC existentes no sistema**
- ***ipcrm*: (IPC ReMove) remove entidades IPC cujos identificadores sejam especificados**

- As entidades IPC não estão integradas no modelo de canais (file descriptors) do sistema de ficheiros (como acontece com os “pipes”).
- Mas, como têm nomes globais no SO, permitem que processos que não sejam da mesma família (por operações “fork”) comuniquem entre si.

Anexo: consultar manual e livro

Entidades IPC: filas de mensagens (...)

permissões de acesso

Chamadas ao SO:

msgget: criar uma fila e/ou obter um ID

Anexo: consultar manual e livro

Entidades IPC: filas de mensagens (...)

permissões de acesso

Chamadas ao SO:

msgget: criar uma fila e/ou obter um ID

msgctl: obter info./mudar atributos/destruir

Anexo: consultar manual e livro

Entidades IPC: filas de mensagens (...)

permissões de acesso

Chamadas ao SO:

msgget: criar uma fila e/ou obter um ID

msgctl: obter info./mudar atributos/destruir

msgsnd: enviar uma mensagem

Anexo: consultar manual e livro

Entidades IPC: filas de mensagens (...)

permissões de acesso

Chamadas ao SO:

msgget: criar uma fila e/ou obter um ID

msgctl: obter info./mudar atributos/destruir

msgsnd: enviar uma mensagem

msgrcv: receber uma mensagem

Exemplos de operações GET

- **GET: cria ou dá acesso a uma entidade IPC e devolve um ID inteiro a usar em acessos posteriores:**

msg_queue_id = msgget(….);

-- para uma fila de mensagens

Exemplos de operações GET

- **GET: cria ou dá acesso a uma entidade IPC e devolve um ID inteiro a usar em acessos posteriores:**

```
msg_queue_id = msgget(….);
 -- para uma fila de mensagens
sem_id = semget(….);
 -- para um conjunto de semáforos
```

Exemplos de operações GET

- **GET: cria ou dá acesso a uma entidade IPC e devolve um ID inteiro a usar em acessos posteriores:**

```
msg_queue_id = msgget(….);
 -- para uma fila de mensagens
sem_id = semget(….);
 -- para um conjunto de semáforos
shm_id = shmget(….);
 -- para uma região de memória partilhada
```

Operações de controlo (CTL)

- **CTL: controlo, obtenção de informação ou definição de parâmetros que configuram o estado da entidade**

msgctl(…)

-- para uma fila de mensagens

semctl(…)

-- para um conjunto de semáforos

shmctl(…)

-- para uma região de memória partilhada

Operações específicas

- **OP: outras operações, específicas a cada tipo de entidade, para comunicação ou sincronização.**

msgop(…)

-- para uma fila de mensagens

semop(…)

-- para um conjunto de semáforos

shmop(…)

-- para uma região de memória partilhada

Parâmetros das funções GET

- **msg/sem/shm GET têm a forma genérica:**

…get(key_t key, int flag)

em que:

key: especifica o nome global da entidade IPC

Parâmetros das funções GET

- **msg/sem/shm GET têm a forma genérica:**

…get(key_t key, int flag)

em que:

key: especifica o nome global da entidade IPC

flag: especifica o modo de criação e as permissões de acesso (indicadas como para os ficheiros Unix) ou então é 0

criar ou obter acesso a uma fila de mensagens:

msgqid = msgget(key, options)

Sendo:

key um nome global (argumento de entrada)

msgqid um identificador único, também global, devolvido pelo SO, para acesso mais eficiente à fila.

int msgget(ket_t key, int options)

se key > 0: este é o nome global da fila

options = IPC_CREAT | IPC_EXCL

devolve um msgqid ID global e único

os processos fazem:

```
msgget(key,...); ... msgsnd / msgrcv ...
```

int msgget(ket_t key, int options)

se **key > 0**

(...)

se **key = IPC_PRIVATE**

um processo cria uma nova fila com um ID global único – gerado pelo SO

que depois passa a outros processos para fazerem msgsnd / msgrcv

int msgget(key_t key, int flag);

```
msg_id = msgget((key_t)0100, options);
```

-- se **IPC_CREAT**: cria uma fila de mensagens, se não existe nenhuma com o nome dado (0100); devolve o seu identificador global inteiro em msg_id (mesmo que a fila já exista)

int msgget(key_t key, int flag);

msg_id = msgget((key_t)0100, options);

-- se IPC_CREAT: cria uma fila de mensagens, se não existe nenhuma com o nome dado (0100); devolve o seu identificador global inteiro em msg_id (mesmo que a fila já exista)

-- se IPC_CREAT | EXCL: dá erro se já existe tal fila

int msgget(key_t key, int flag);

msg_id = msgget((key_t)0100, options);

-- se IPC_CREAT: cria uma fila de mensagens, se não existe nenhuma com o nome dado (0100); devolve o seu identificador global inteiro em msg_id (mesmo que a fila já exista)

-- se IPC_CREAT | EXCL: dá erro se já existe tal fila

-- se não usar a opção IPC_CREATE e referir um nome key já existente, msgget dá acesso à fila, devolvendo o seu identificador

int msgget(key_t key, int flag);

```
msg_id = msgget((key_t)0100, 0644  | IPC_CREATE );
```

(Owner, Group, Other)

rwx rwx rwx

-- permissões 0644 = 110 100 100

em que

o modo ´r´ permissão para receber

o modo ´w´ permissão para enviar

o modo ‘x’ não tem significado no caso das filas de mensagens.

No SO: Tabela de entidades IPC

Um número limitado de entradas

No caso da fila de mensagens:

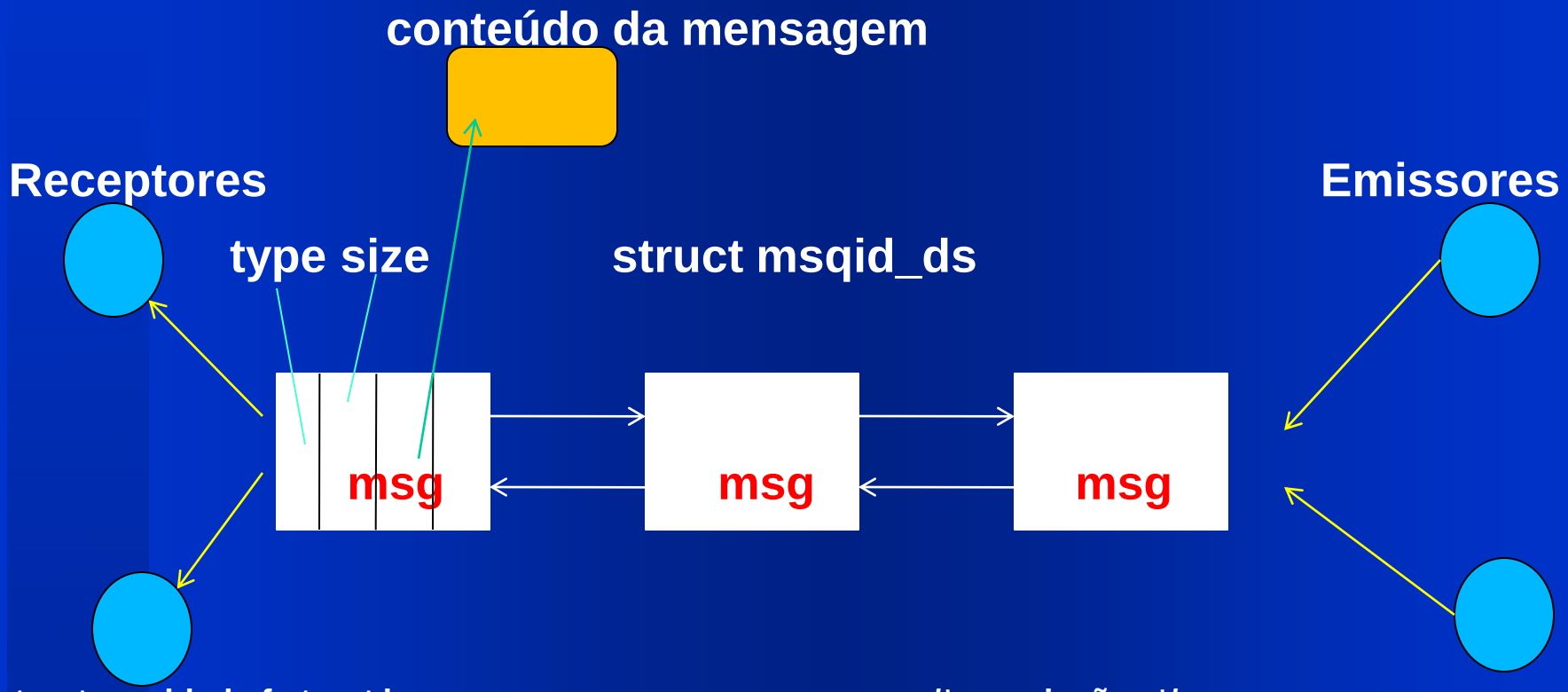
Cada entrada descreve uma fila:

-- key, criador, permissões de acesso

-- informação específica sobre a fila

O ID da fila é o índice desta entrada na tabela, devolvido por msgget() e é este valor que é utilizado nas operações de envio e recepção

Implementação




```
struct msqid_ds { struct ipc_perm      msg_perm;          /* permissões */
                  struct       msg    * msg_first,  *msg_last;
                     int        msg_cbytes;             /* número actual de bytes na fila*/
                     int        msg_qnum;               /* número actual de mensagens na fila*/
                     int        msg_qbytes;             /* max. número de bytes admitido na fila */
                     ....              /*outra informação de estado*/}
```

Representação de uma mensagem

Para o SO: internamente

```
struct msg {
      struct  msg      *msg_next;
        ...       msg_type;  /*tipo da mensagem*/
        ...       msg_tsize; /*tamanho do texto da mensagem em bytes*/
                  msg_spot;  /* referência para o endereço de início da msg*/
    }
```

Para o Programa, nas chamadas *msgsnd / msgrcv*:

```
 struct msgbuf { -- dois campos:   tipo, seguido de uma zona de dados
        long   mtype;       /*tipo*/
        char   mtext[ ];    /* um buffer de bytes */
/* argumentos de msgsnd ou msgrcv → indicam o tamanho do buffer */
} – os bytes a seguir a mtype, são copiados para a estrutura do SO
```

Para o SO, uma estrutura de mensagem válida é formada por:

um inteiro (long), seguido de uma série de 0 ou mais bytes

.... um vector de bytes ...

Para o SO: a referência para copiar o texto da mensagem é o endereço do primeiro elemento que venha logo a seguir ao campo *mtype* → permite declarar qualquer outra estrutura, desde que se siga ao 1º elemento (*mtype)*

```
struct  xmsg {
   long    mtype;
```

← *mtext[ ]*

....

Para o SO, uma estrutura de mensagem válida é formada por:

um inteiro (long), seguido de uma série de 0 ou mais bytes

.... um vector de bytes ...

Para o SO: a referência para copiar o texto da mensagem é o endereço do primeiro elemento que venha logo a seguir ao campo *mtype* → permite declarar qualquer outra estrutura, desde que se siga ao 1º elemento (*mtype)*

```
struct  xmsg {
   long    mtype;
   pid_t   m_process_id;
   char    buffer[bufsiz];
....
```

← *mtext[ ]*

EXEMPLO

enviar mensagem (message send):

msgsnd(msgqid, msg, size, options)

Sendo:

msgqid o identificador da fila

msg uma estrutura com dois campos:

- o tipo da mensagem (um inteiro>0)
- o texto (um vector de bytes ...)

o tipo é definido pelo emissor, que possibilita ao receptor seleccionar que mensagens remove da fila, conforme o seu tipo

int msgsnd(-- envia a msg apontada por *msgp*

int msqid,

const void *msgp,

size_t msize, -- um valor de 0 até um limite definido do SO

int option) ; - *default*: não bloqueia, excepto se fila cheia

- ou: se *IPC_NOWAIT* e fila cheia: retorna -1 e erro

---

int msgrcv(-- obtém1 msg → estrutura apontada por *msgp*;
devolve -1 ou o número de bytes recebidos para *mtext*

int msqid,

void *msgp,

size_t msize, -- tamanho da maior mensagem a receber

long mtype;

int option) ; - *default*: bloqueia, se fila não tem msg

- ou: se *IPC_NOWAIT* e fila vazia: retorna -1 e indica erro

Em msgrcv

Option: pode assumir os valores:

IPC_NOWAIT

|

***MSG_NOERROR*: se mensagem > msize**

trunca a mensagem, ao copiá-la

(default: -- se mensagem > msize

dá erro: errno = E2BIG)

receber mensagem (message receive):

cnt =msgrcv(msgqid, msg, size, type, options)

Sendo:

msg um apontador para uma estrutura para conter a informação sobre a mensagem recebida (campos *mtype* e *mtext*)

type um dos seguintes valores:

type = 0 a ordem de entrega é FIFO

receber mensagem (message receive):

cnt =msgrcv(msgqid, msg, size, type, options)

Sendo:

msg um apontador para uma estrutura para conter a informação sobre a mensagem recebida

type um dos seguintes valores:

- **type = 0** a ordem de entrega é FIFO
- **type > 0** deve obter a primeira mensagem, na ordem FIFO, que tenha o tipo indicado

receber mensagem (message receive):

cnt =msgrcv(msgqid, msg, size, type, options)

Sendo:

msg um apontador para uma estrutura para conter a informação sobre a mensagem recebida

type um dos seguintes valores:

- **type = 0** a ordem de entrega é FIFO
- **type > 0** deve obter a primeira mensagem, na ordem FIFO, que tenha o tipo indicado
- **type < 0** deve obter a primeira mensagem com tipo igual ou inferior ao valor absoluto **|type|** **pela ordem crescente dos valores de type**

Exemplo

1ª: M1, type = 1

2ª: M2, type = 5

3ª: M3, type = 34

Caso msgrcv() use:

-- type = 0: devolve a mensagem M1

1ª: M1, type = 1

2ª: M2, type = 5

3ª: M3, type = 34

Caso msgrcv() use:

-- type = 0: devolve a mensagem M1

-- type > 0: exemplo, type = 5, devolve M2

1ª: M1, type = 1

2ª: M2, type = 5

3ª: M3, type = 34

Caso msgrcv() use:

-- type = 0: devolve a mensagem M1

-- type > 0: exemplo, type = 5, devolve M2

-- type < 0: exemplo, type = - 34, devolve a mensagem de type <= |-34| que tenha o menor valor de *type*, ou seja M1

1ª: M1, type = 1

2ª: M2, type = 5

3ª: M3, type = 34

Caso msgrcv() use:

-- type = 0: devolve a mensagem M1

-- type > 0: exemplo, type = 5, devolve M2

-- type < 0: exemplo, type = - 34, devolve a mensagem de type <= |-34| que tenha o menor valor de *type*, ou seja M1; se repetir a chamada msgrcv() com type=-34, devolve a próxima M2

1ª: M1, type = 1

2ª: M2, type = 5

3ª: M3, type = 34

Caso msgrcv() use:

-- type = 0: devolve a mensagem M1

-- type > 0: exemplo, type = 5, devolve M2

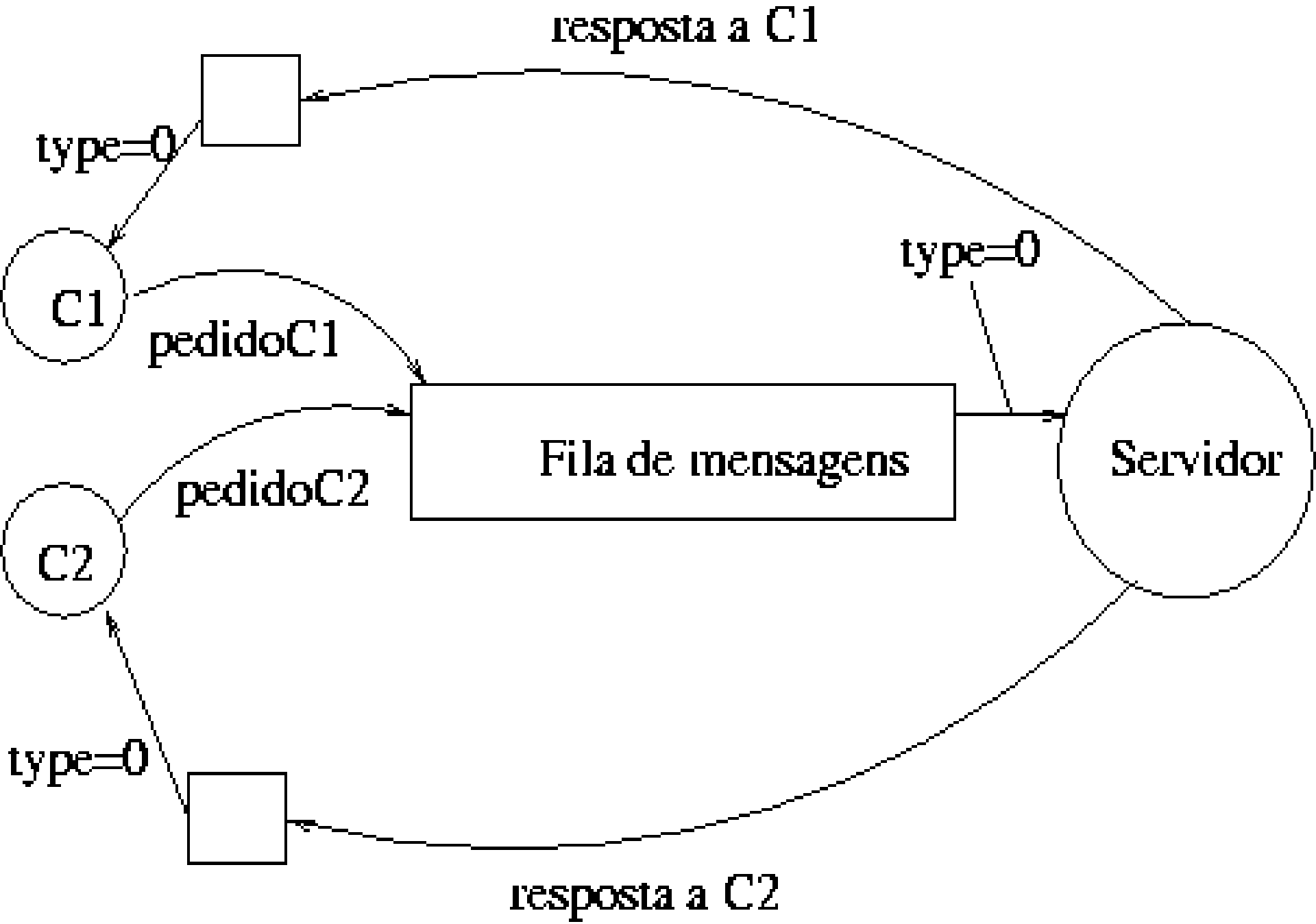
-- type < 0: exemplo, type = - 34, devolve a mensagem de type <= |-34| que tenha o menor valor de *type*, ou seja M1; se repetir a chamada msgrcv() com type=-34, devolve a próxima M2; e se repetir, devolve M3

Um exemplo: cliente-servidor

Comunicação baseada em filas de mensagens

Unix System V. **Que opções?**

a) O servidor tem uma fila única onde recebe apenas os pedidos dos clientes; cada cliente envia ao servidor o nome de uma fila privada onde fica aguardar a resposta do servidor


resposta a C1
type=0
C1
pedidoC1
type=0
Fila de mensagens
Servidor
pedidoC2
C2
type=0
resposta a C2

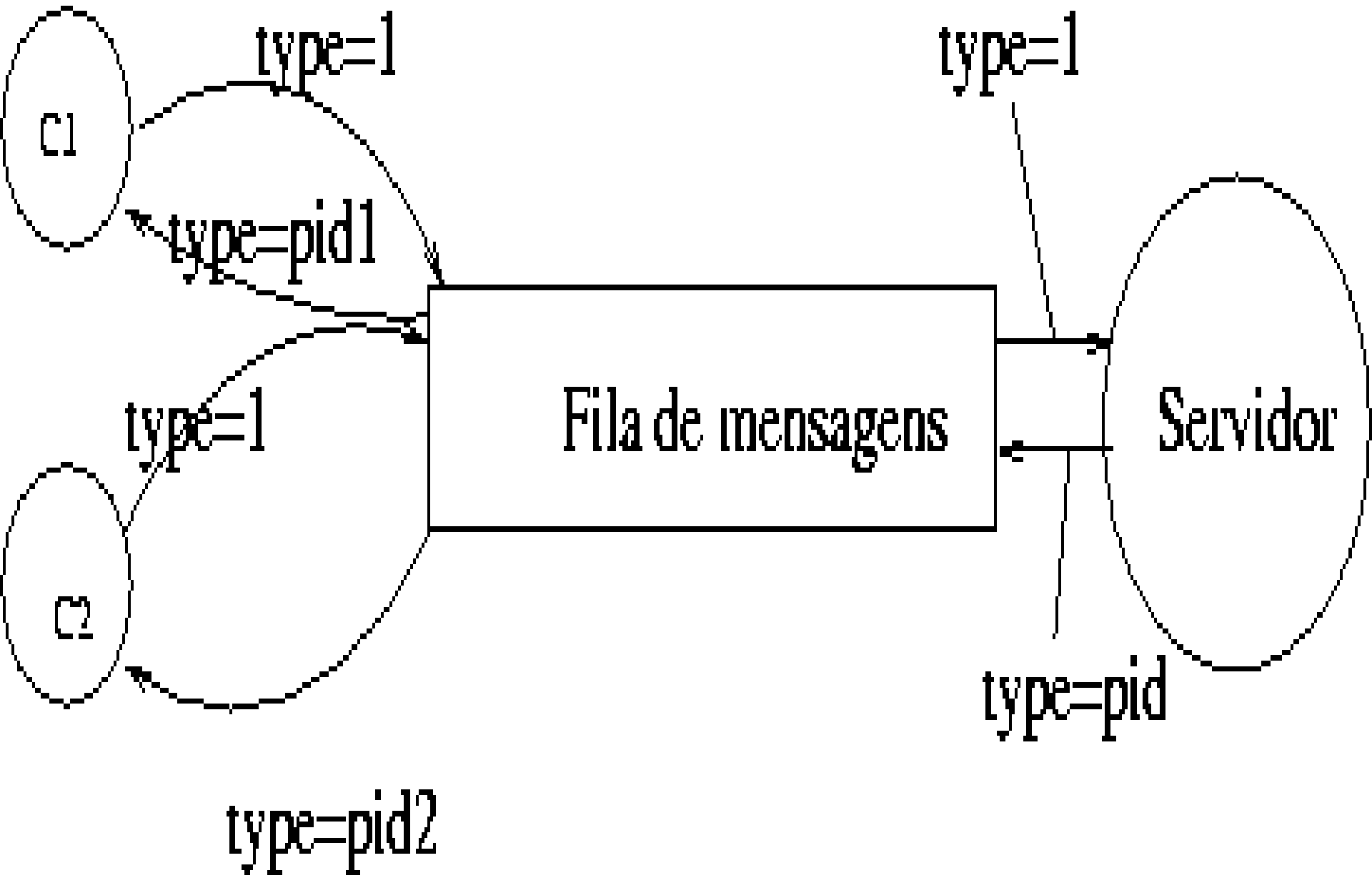
Um exemplo: cliente-servidor

Comunicação baseada em filas de mensagens

Unix System V. **Que opções?**

a) ...

b) O servidor tem uma fila única onde recebe os pedidos dos clientes e onde coloca as respostas; os clientes colocam os pedidos na fila do servidor e ficam a aguardar as respectivas respostas → tipos de mensagens


type=1
type=1
c1
type=pid1
type=1
Fila de mensagens
Servidor
c2
type=pid
type=pid2

Cliente:

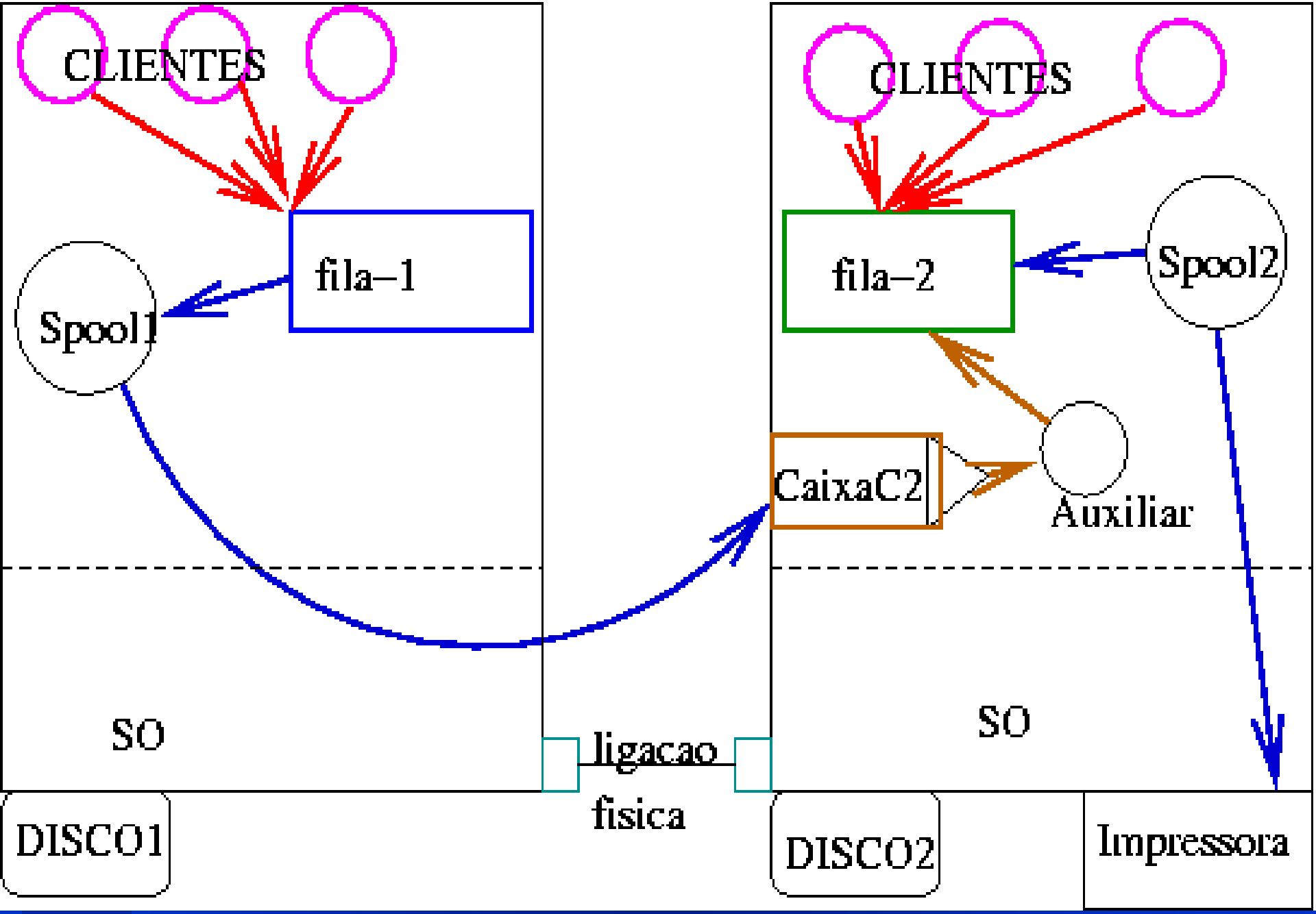
```
#define MSGKEY 75
...
msgid = msgget(MSGKEY,…);
    ...                 /*prepara mensagem: tipo = 1
    ...                  texto inclui pedido e
                                  ´´pid´´ do cliente */
msgsnd(msgid, msg1, ..., ...);
msgrcv(msgid, Msg2,.....,pid,…);
                              /*aguarda   resposta*/
….
```

Servidor:

```
#define MSGKEY 75
...
msgid = msgget(MSGKEY,…);
...
msgrcv(msgid, Msg,…,1,…); /*aguarda um pedido*/
… /*obtém texto e trata o pedido do cliente pid*/
...
/*prepara mensagem: tipo = pid  e texto inclui a
   resposta ao cliente */
msgsnd(msgid, msg2, ..., ...);
....
```

Um sistema de SPOOL distribuído

- **Um exemplo muito simplificado…**
- **Dois computadores C1 e C2 ligados entre si por um cabo… (uma rede de computadores)**
- **Cada computador, um Sistema de Operação com multiprogramação**
- **C2 é o único que tem uma impressora ligada**
- **Como permitir a qualquer processo, esteja em C1 ou em C2, possa pedir a impressão de um ficheiro, concorrentemente com outros processos?**


CLIENTES
fila–1
Spool1
SO
DISCO1
ligacao
fisica
CLIENTES
fila–2
Spool2
CaixaC2
Auxiliar
SO
DISCO2
Impressora

Em Ci (i:1..2) os Clientes invocam:

pedir-imprimir-i(Pedido):

```
{  ....                inserir(Pedido, fila-i);  .....

   ....
}
```

Em Ci (i:1..2) os Clientes invocam: **ex-fila-i = 1**

```
pedir-imprimir-i(Pedido):
 { P(ex-fila-i);  inserir(Pedido, fila-i);  V(ex-fila-i);
   .....
 }
```

Em Ci (i:1..2) os Clientes invocam:

ex-fila-i = 1

sem-i = 0

pedir-imprimir-i(Pedido):

```
{ P(ex-fila-i);  inserir(Pedido, fila-i);  V(ex-fila-i);
  V(sem-i);
}
```

Em Ci (i:1..2) os Clientes invocam:

pedir-imprimir-i(Pedido):

```
 { P(ex-fila-i);  inserir(Pedido, fila-i);  V(ex-fila-i);
   V(sem-i);
 }
```

Em C1: processo SPOOL-1:

```
while true do {
   ....
}
```

Em Ci (i:1..2) os Clientes invocam:

```
pedir-imprimir-i(Pedido):
 { P(ex-fila-i);  inserir(Pedido, fila-i);  V(ex-fila-i);
   V(sem-i);
 }
```

Em C1: processo SPOOL-1:

```
while true do {
  ....
  P(ex-fila-1);  remover(Pedido,fila-1);  V(ex-fila-1);
  ....
}
```

Em Ci (i:1..2) os Clientes invocam:

```
pedir-imprimir-i(Pedido):
 { P(ex-fila-i);  inserir(Pedido, fila-i);  V(ex-fila-i);
   V(sem-i);
 }
```

Em C1: processo SPOOL-1:

```
while true do {
  P(sem-1);
  P(ex-fila-1);  remover(Pedido,fila-1);
  ....
}
```

Em Ci (i:1..2) os Clientes invocam:

```
pedir-imprimir-i(Pedido):
 { P(ex-fila-i);  inserir(Pedido, fila-i);  V(ex-fila-i);
   V(sem-i);
 }
```

Em C1: processo SPOOL-1:

```
while true do {
  P(sem-1);
  P(ex-fila-1);  remover(Pedido,fila-1);
  enviar_msg(caixa2, Pedido);
}
```

Em C2: processo que recebe pedidos de C1

```
while true do {
    ....
}
```

Em C2: processo que recebe pedidos de C1

```
while true do {
  receber_msg(caixa2,Pedido);
  ....
}
```

Em C2: processo que recebe pedidos de C1

```
while true do {
  receber_msg(caixa2,Pedido);
  pedir-imprimir-2(Pedido);
}
```

Em C2: processo que recebe pedidos de C1

```
while true do {
  receber_msg(caixa2,Pedido);
  pedir-imprimir-2(Pedido);
}
```

Em C2: processo SPOOL-2:

```
while true do {
   ....
}
```

Em C2: processo que recebe pedidos de C1

```
while true do {
  receber_msg(caixa2,Pedido);
  pedir-imprimir-2(Pedido);
}
```

Em C2: processo SPOOL-2:

```
while true do {
  ....
 P(ex-fila-2);remover(Pedido,fila-2);V(ex-fila-2);
  ....
}
```

Em C2: processo que recebe pedidos de C1

```
while true do {
  receber_msg(caixa2,Pedido);
  pedir-imprimir-2(Pedido);
}
```

Em C2: processo SPOOL-2:

```
while true do {
  P(sem-2);
 P(ex-fila-2);remover(Pedido,fila-2);V(ex-fila-2);
  ....
}
```

Em C2: processo que recebe pedidos de C1

```
while true do {
  receber_msg(caixa2,Pedido);
  pedir-imprimir-2(Pedido);
}
```

Em C2: processo SPOOL-2:

```
while true do {
  P(sem-2);
 P(ex-fila-2);remover(Pedido,fila-2);V(ex-fila-2);
  imprimir(ficheiro-pedido);
}
```