Sistemas Operacionais

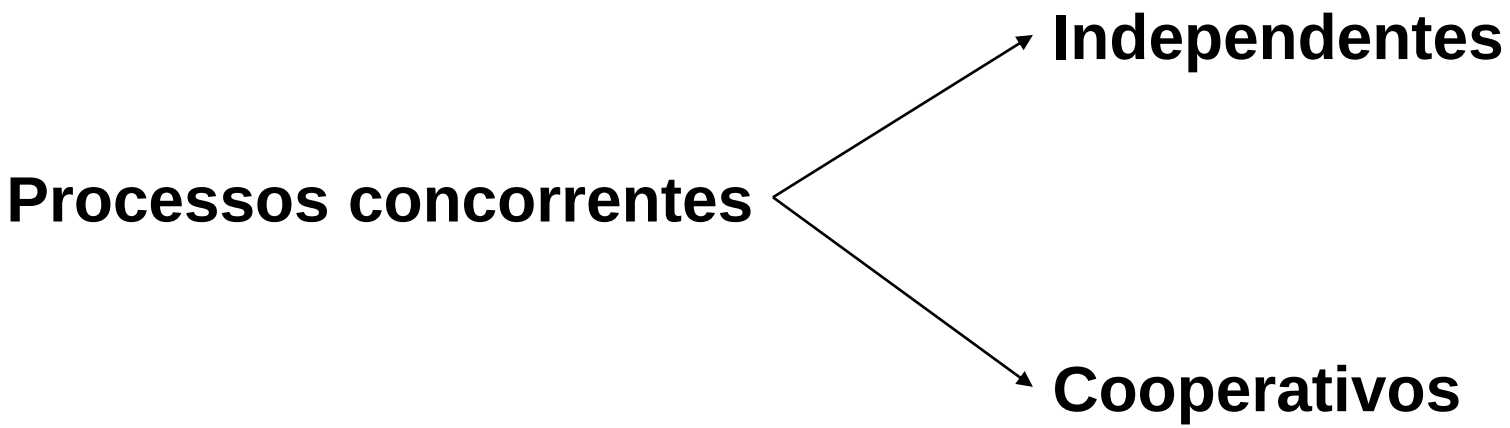
Concorrência e Comunicação entre Processos

Tarefas e Processos concorrentes

Índice

- Tarefas e processos cooperativos
- Dependência de tarefas
- Motivação
- Comunicação entre processos - IPC

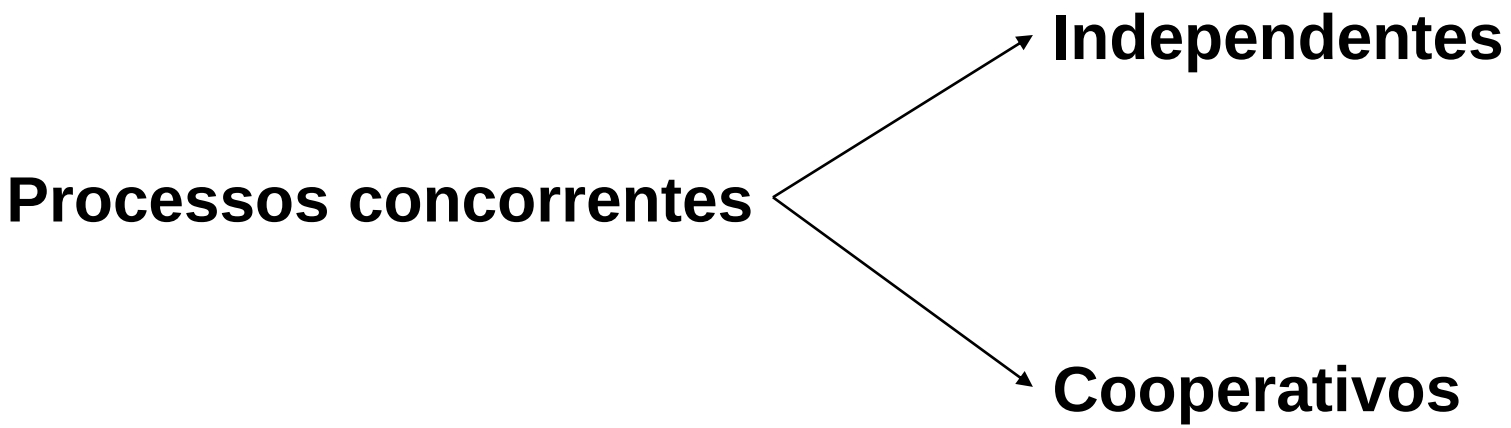
Tarefas e Processos Cooperativos



Independente – Não afeta nem é afetado por outros processos.

Inter-dependente ou Cooperativo – Pode afetar e ser afetado por outro processo executado no sistema.

Tarefas e Processos Cooperativos



Um processo que não compartilha dados com outro → **Independente**

Quaisquer processos que compartilhem dados entre si → **Inter-dependentes** ou **cooperativos**

Tarefas e Processos cooperativos

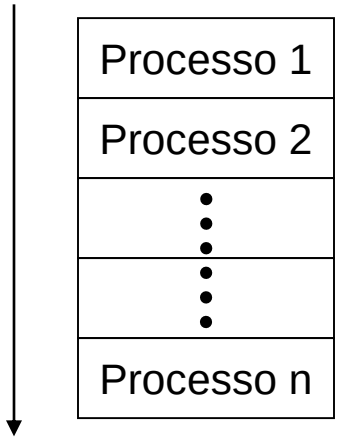
Tarefas e Processos Cooperativos

Sistemas complexos são implementados como uma estrutura com várias tarefas que cooperam entre si para atingir os objetivos da aplicação.

Processos Sequenciais e Cooperativos

Nem sempre um programa sequencial é a melhor solução

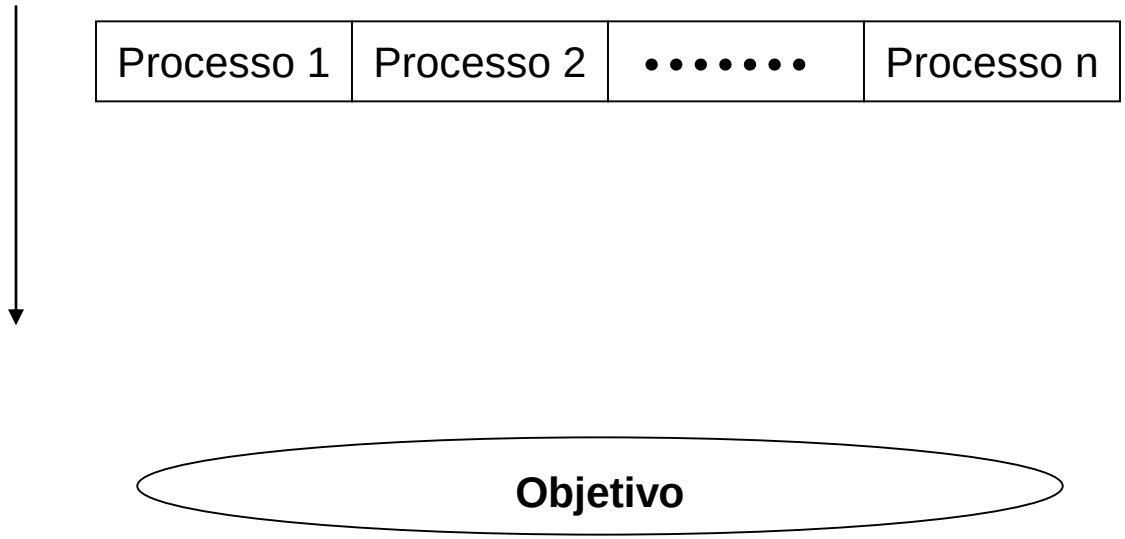
1. Sequenciais



Objetivo

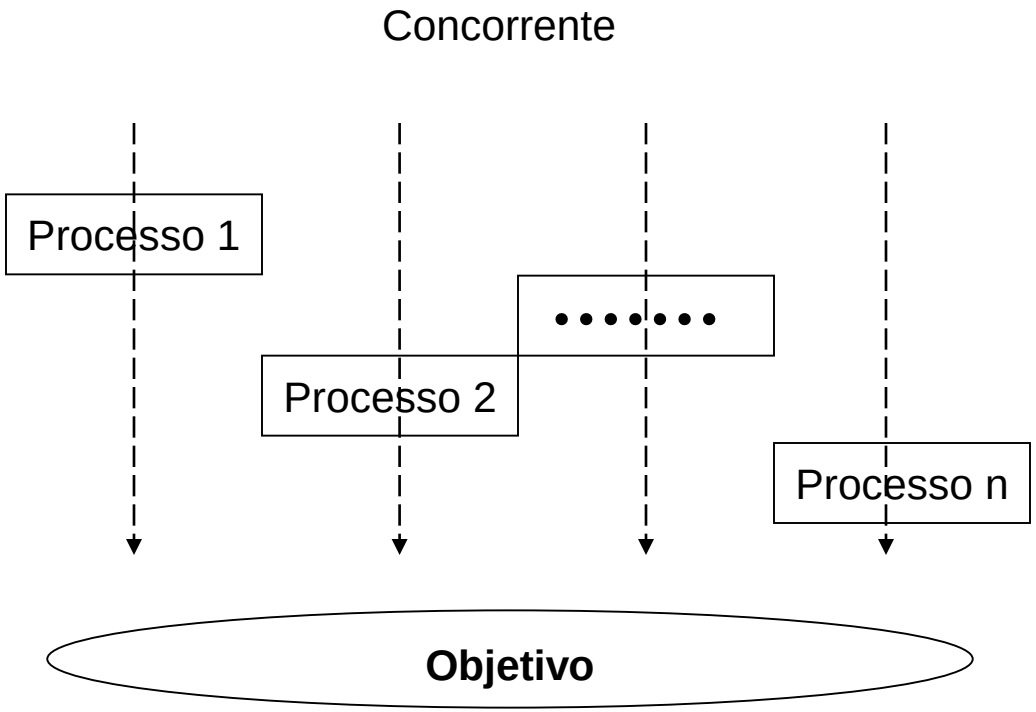
Processos Sequenciais e Cooperativos

2. Cooperativos



Processos Sequenciais e Cooperativos

2. Cooperativos

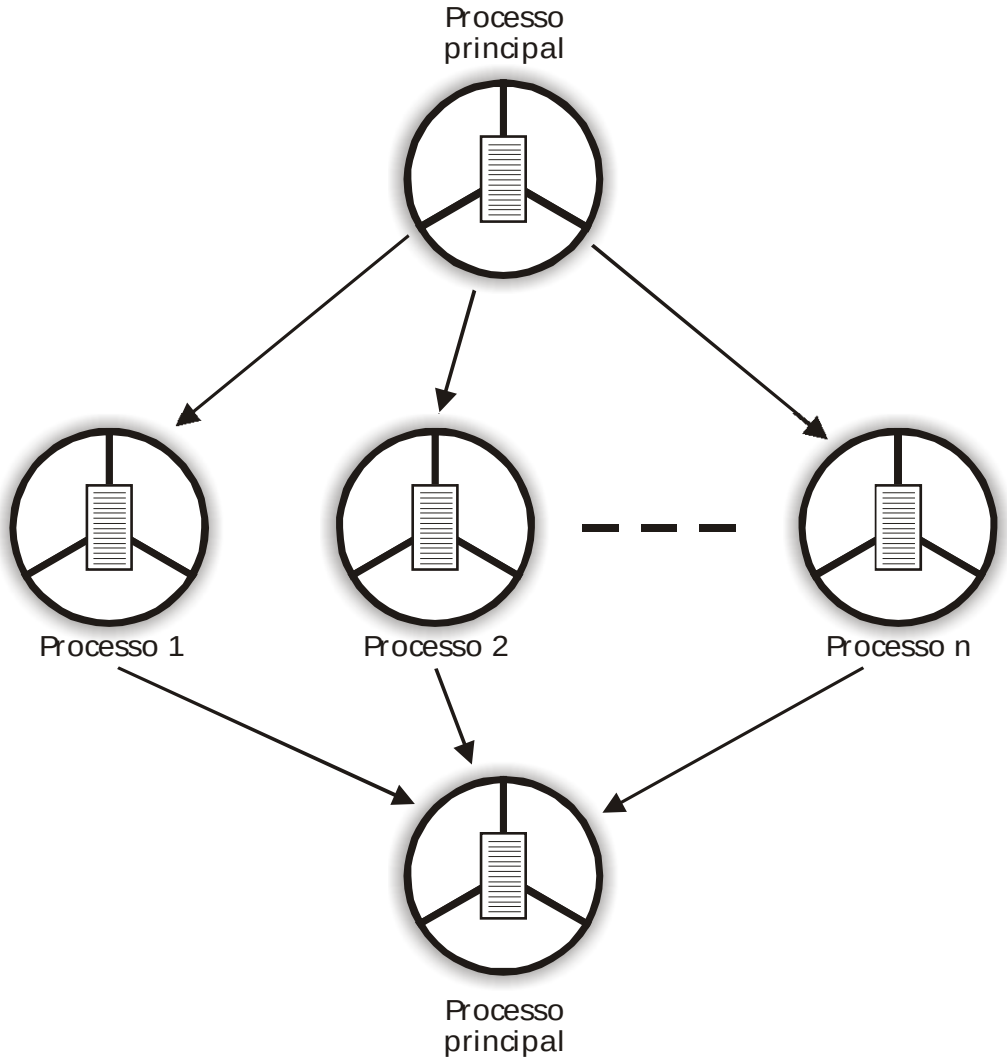


Processos Sequenciais e Cooperativos

2. Cooperativos

```
PARBEGIN
    Comando_1;
    Comando_2;
        .
        .
    Comando_n;
PAREND
```

Threads!



Por que implementar sistemas baseados em Processos Cooperativos?

-Atender vários usuários simultâneos

- Compartilhar informações (entre usuários simultâneos)

-Maior velocidade de processamento (Múltiplas CPU's)

-Modularidade

-Aplicações interativas

Por que implementar sistemas baseados em Processos Cooperativos?

-**Atender vários usuários simultâneos**

  - **Compartilhar informações (entre usuários simultâneos)**

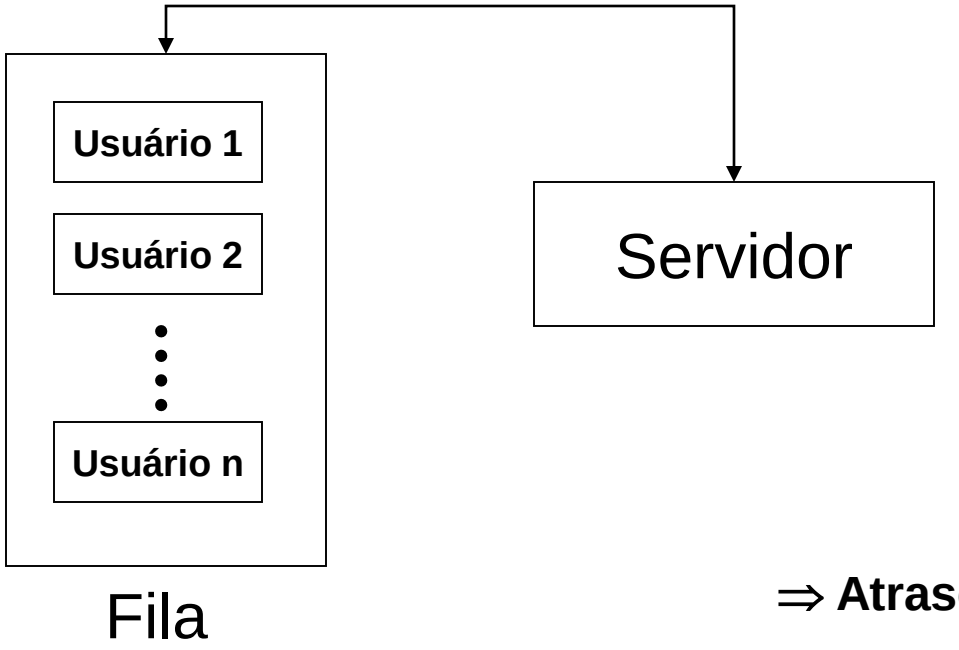
-Maior velocidade de processamento (Múltiplas CPU's)

-Modularidade

-Aplicações interativas

Múltiplos usuários (serviços)

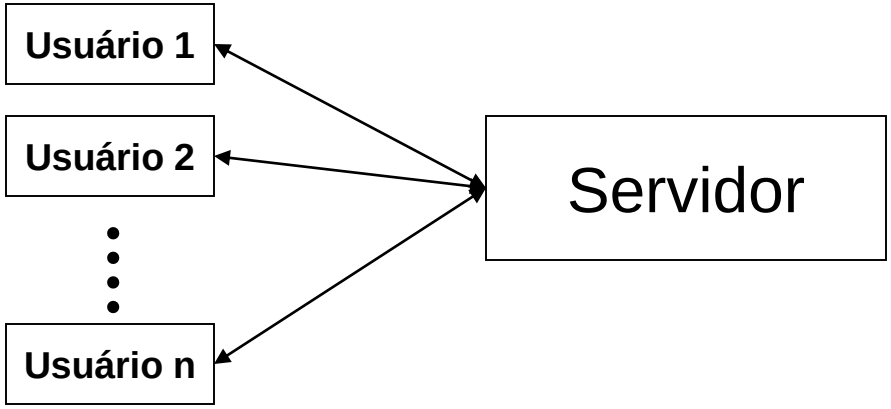
Vários Usuários / Compartilhamento de informações



⇒ **Atrasos intoleráveis**

Múltiplos usuários (serviços)

Vários Usuários / Compartilhamento de informações



⇒ **Atendimento “simultâneo”**

Por que implementar sistemas baseados em Processos Cooperativos?

-Atender vários usuários simultâneos

- Compartilhar informações (entre usuários simultâneos)

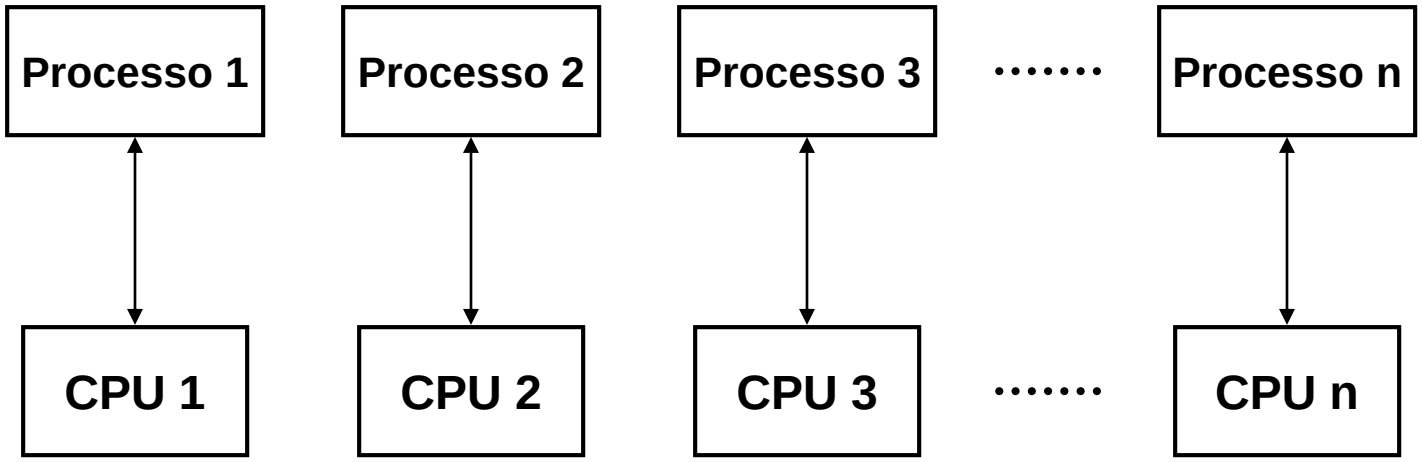
-Maior velocidade de processamento (Múltiplas CPU's)

-Modularidade

-Aplicações interativas

Múltiplos processadores

Velocidade de processamento (Múltiplas CPU's)



A aplicação é "separada" em várias tarefas paralelas

Por que implementar sistemas baseados em Processos Cooperativos?

-Atender vários usuários simultâneos

- Compartilhar informações (entre usuários simultâneos)

-Maior velocidade de processamento (Múltiplas CPU's)

-Modularidade

-Aplicações interativas

Divisão dos trabalhos

Modularidade

Sistemas muito **complexos** podem ser melhor organizados dividindo-se suas atribuições em módulos sob a responsabilidade de **tarefas interdependentes**

$\Rightarrow$ Módulos interdependentes

Por que implementar sistemas baseados em Processos Cooperativos?

-Atender vários usuários simultâneos

- Compartilhar informações (entre usuários simultâneos)

-Maior velocidade de processamento (Múltiplas CPU's)

-Modularidade

-Aplicações interativas

Sistemas interativos

Aplicações interativas

-Navegadores
-Editores de texto
-Jogos

- **Ex.: Navegador**

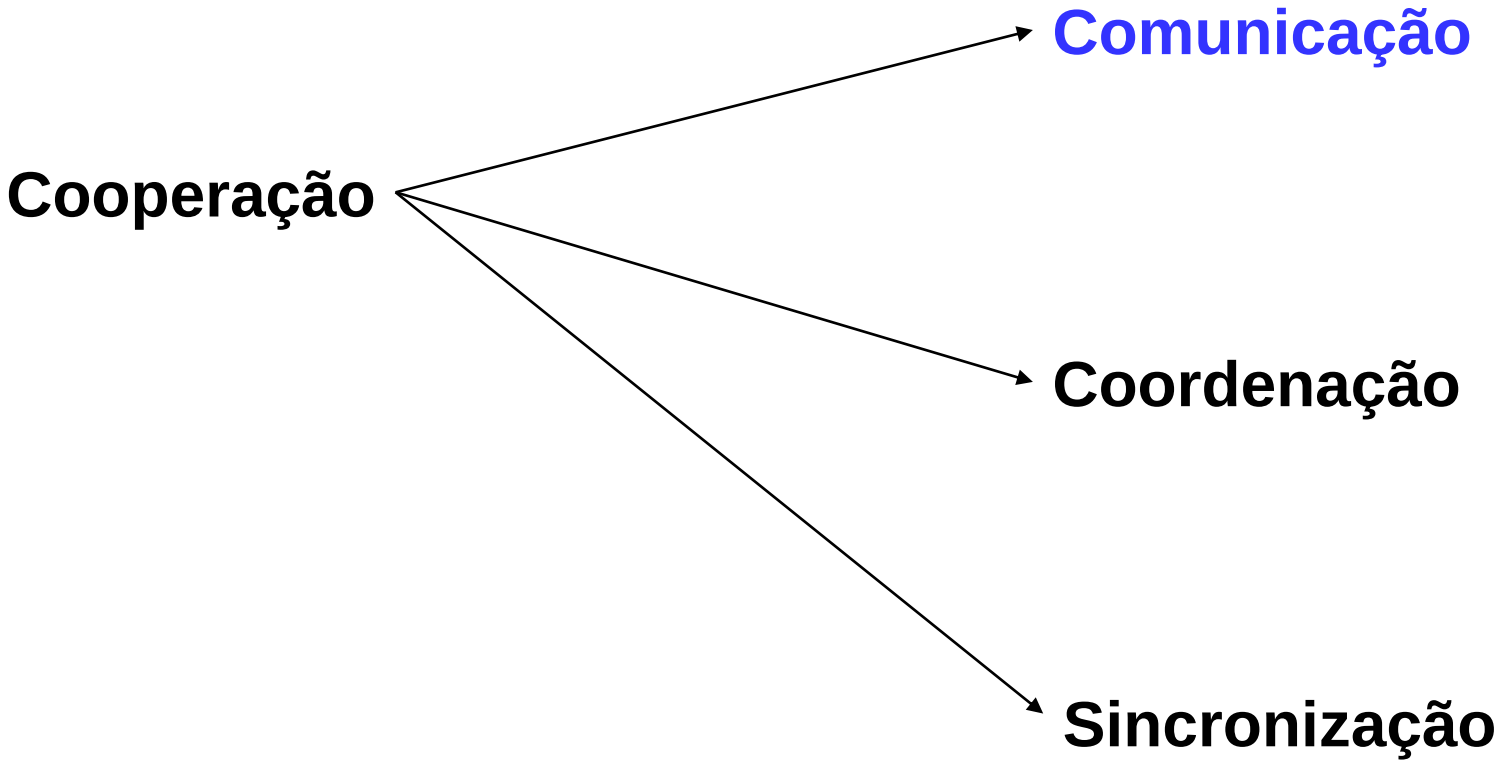
Tarefas relacionadas à interface respondem a comandos dos usuários

- **Outras tarefas:**

-Comunicação com a rede
-Revisão ortográfica
-*Renderização* de imagens na tela

Comunicação entre Tarefas

Comunicação entre Tarefas



Comunicação entre Tarefas

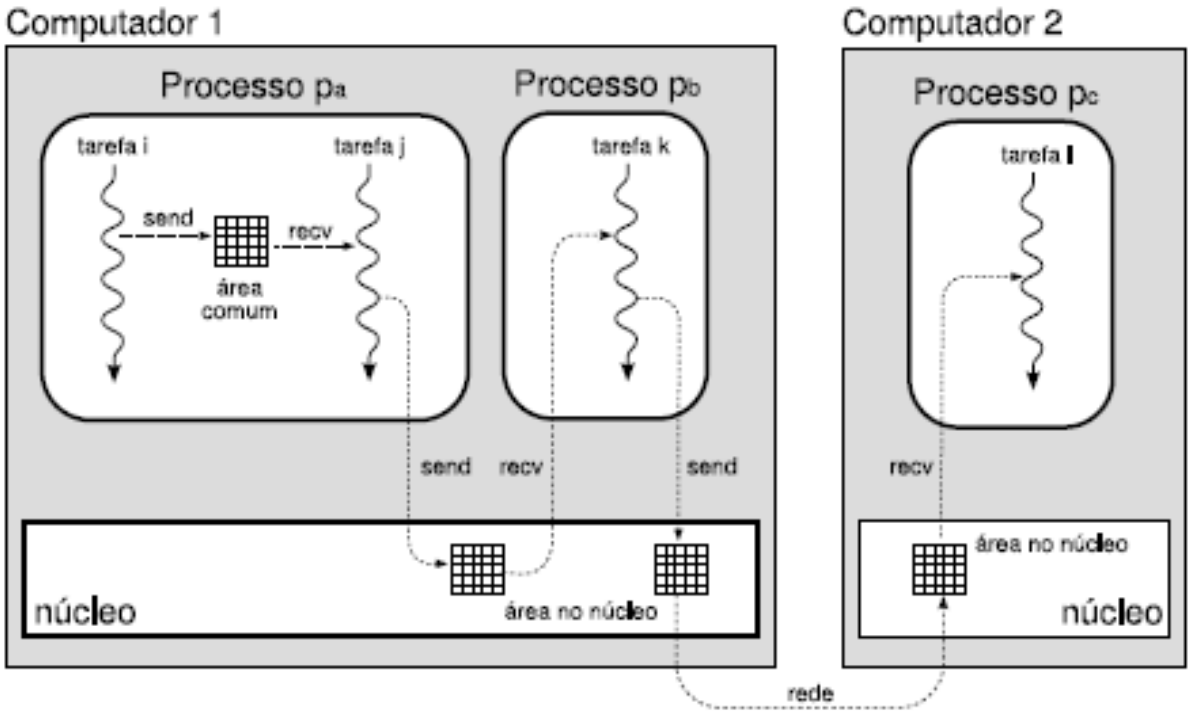
Comunicação: Compartilhamento de **informações** necessárias à execução de cada tarefa.

Coordenação: Das atividades para que os **resultados** sejam **coerentes** (sem erros).

Sincronização: O acesso ao recurso compartilhado exige a sincronização de processos vinculada a uma condição.

Escopo da Comunicação

- **Intra-processo**
- **Inter-processos**
- **Inter-sistemas**



Comunicação intra-processo (*ti* → *tj*), inter-processos (*tj* → *tk*) e inter-sistemas (*tk* → *tl*).

Formas de Comunicação

Aspectos a considerar na definição dos mecanismos de comunicação

- Conexão
- Sincronismo exigido
- Formato dos dados
- Necessidade de *Buffers*
- Quantidade de *Transmissores* e *Receptores*
- ...

Formas de Comunicação

Aspectos a considerar na definição dos mecanismos de comunicação

- Conexão
- Sincronismo exigido
- Formato dos dados
- Necessidade de *Buffers*
- Quantidade de *Transmissores* e *Receptores*
- ...

IPC – Inter-Process Communication

IPC - Inter-Process Communication

Duas abordagens:

- Transmissão de mensagens
- Memória Compartilhada

IPC - Inter-Process Communication

Duas abordagens:

- Transmissão de mensagens
- Memória Compartilhada

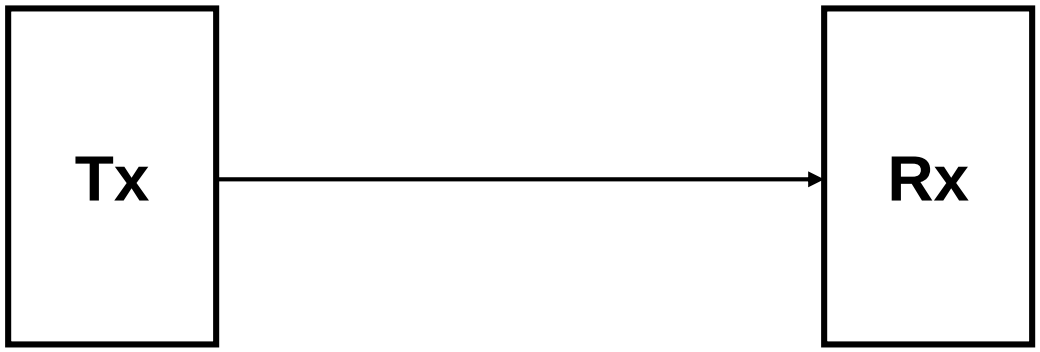
Transmissão de mensagens

- **Forma de conexão**
- Sincronismo exigido
- Formato dos dados
- Necessidade de *Buffers*
- Quantidade de *Transmissores* e *Receptores*

Comunicação Direta ou Indireta

Comunicação → Duas primitivas: ***Enviar* e *Receber***

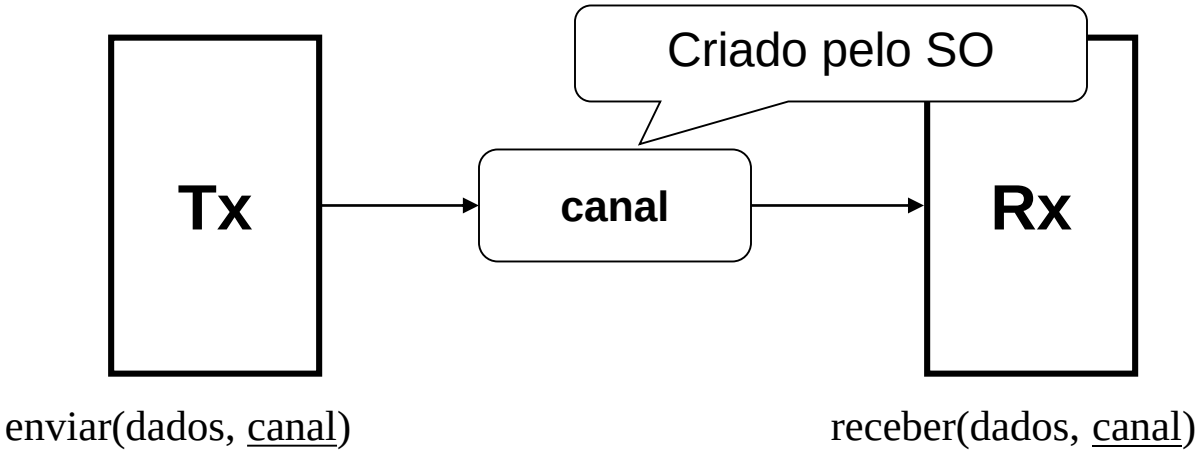
Comunicação **Direta**:



O emissor identifica claramente o receptor e vice-versa

Comunicação Direta ou Indireta

Comunicação **Indireta**:



- Emissor e receptor não precisam se conhecer
- Mais flexível $\Rightarrow$ mais utilizado

Formas de Comunicação

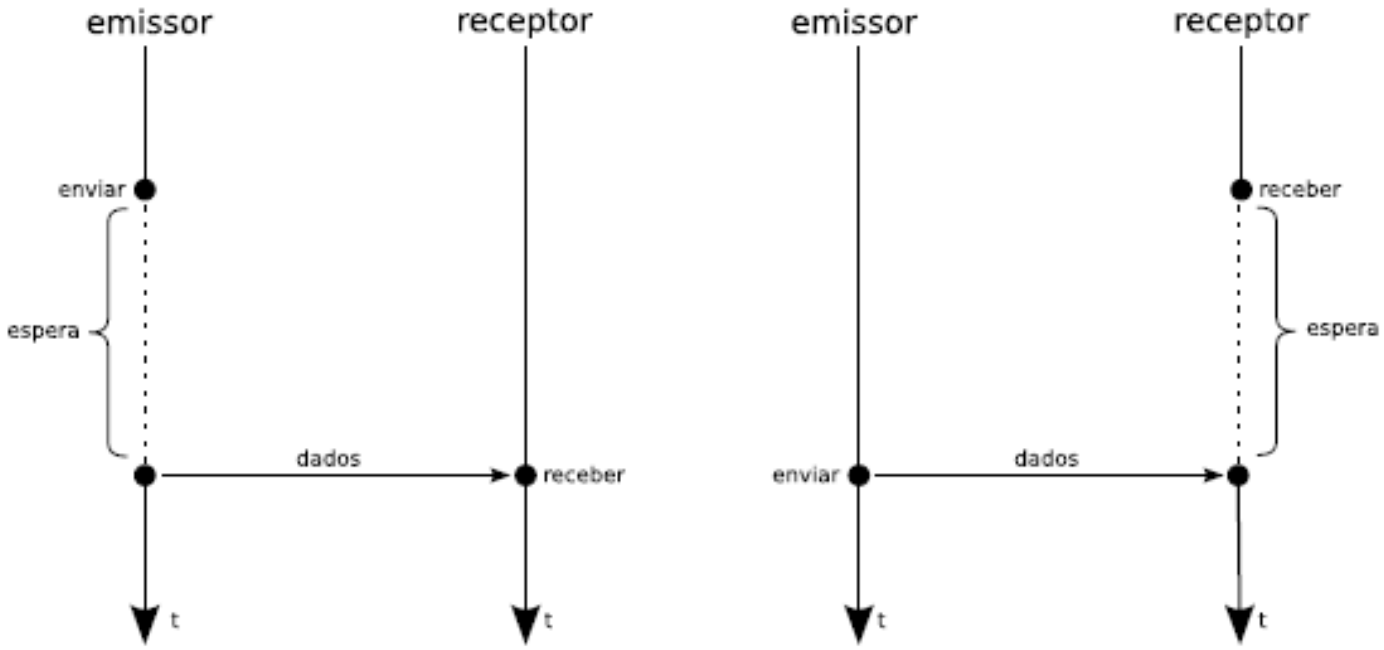
- Conexão
- **Sincronismo exigido**
- Formato dos dados
- Necessidade de *Buffers*
- Quantidade de *Transmissores* e *Receptores*

Sincronismo

A comunicação entre tarefas pode ser:

- Síncrona
- Assíncrona
- Semi-síncrona

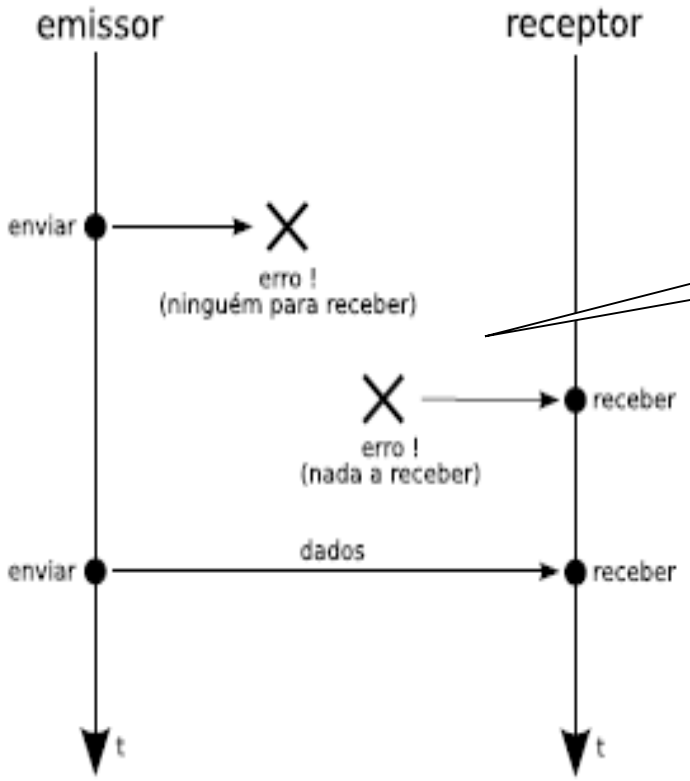
Comunicação Síncrona



Envio e recepção de dados **bloqueiam** as tarefas envolvidas até a **conclusão** da comunicação

Relembrando...

Comunicação Assíncrona

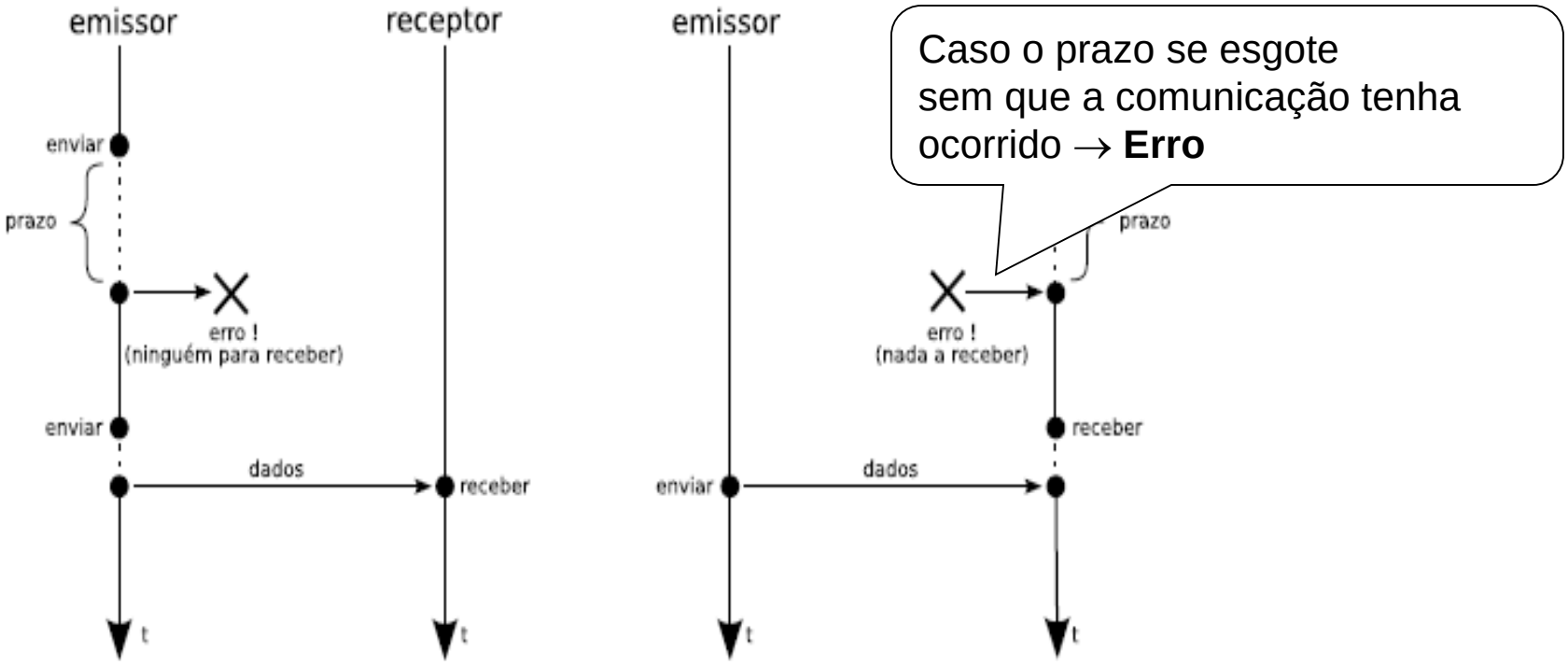


Caso a comunicação não seja possível no momento em que cada operação é invocada → **Erro**

⇒ ***Buffers*** **são imprescindíveis**

Envio e recepção não são “bloqueantes”.

Comunicação Semi-Síncrona



Comportamento síncrono (bloqueio) durante um prazo pré-definido. Primitivas recebem um parâmetro extra → enviar(dados, destino, prazo)

Formas de Comunicação

- Conexão
- Sincronismo exigido
- **Formato dos dados**
- Necessidade de *Buffers*
- Quantidade de *Transmissores* e *Receptores*

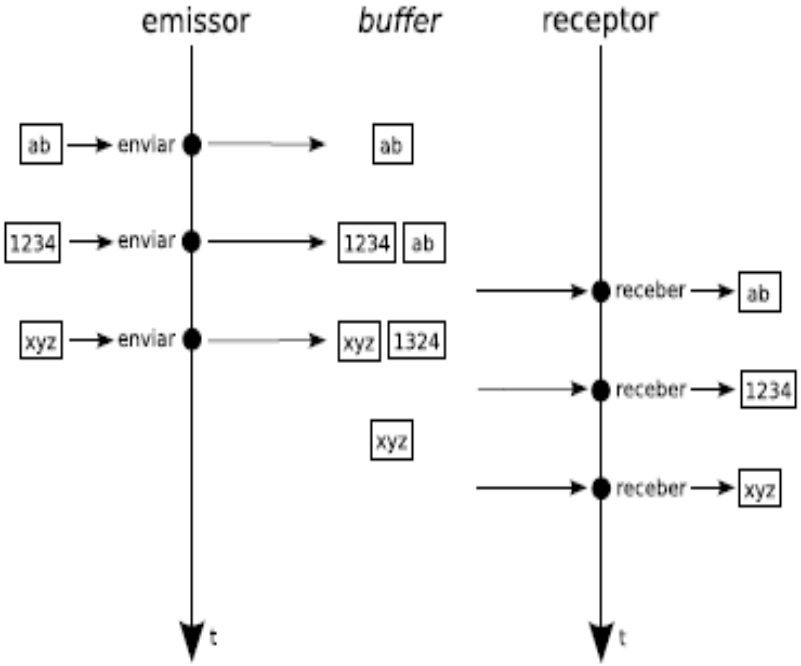
Formato de envio

Duas formas básicas:

- Sequência de mensagens
- Fluxo sequencial

Sequência de mensagens

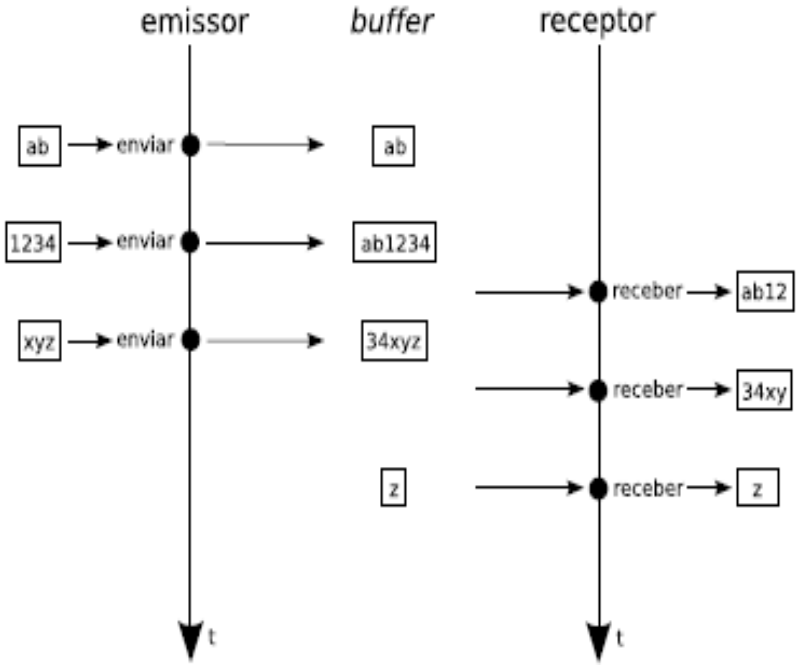
- Cada mensagem é um **pacote** de dados
- Pacote é recebido ou descartado pelo receptor em sua íntegra
- Não existe a possibilidade de receber “meia mensagem”
- Exemplos *message queues* do UNIX e os protocolos de rede UDP



Fluxo sequencial (orientado a conexão**)**

- O canal de comunicação é visto como um arquivo
- O emissor “escreve” dados nesse canal
- Dados serão “lidos” pelo receptor respeitando a ordem de envio
- Dados podem ser lidos byte a byte ou em grandes blocos

Ex: pipes do UNIX e o protocolo de rede TCP/IP

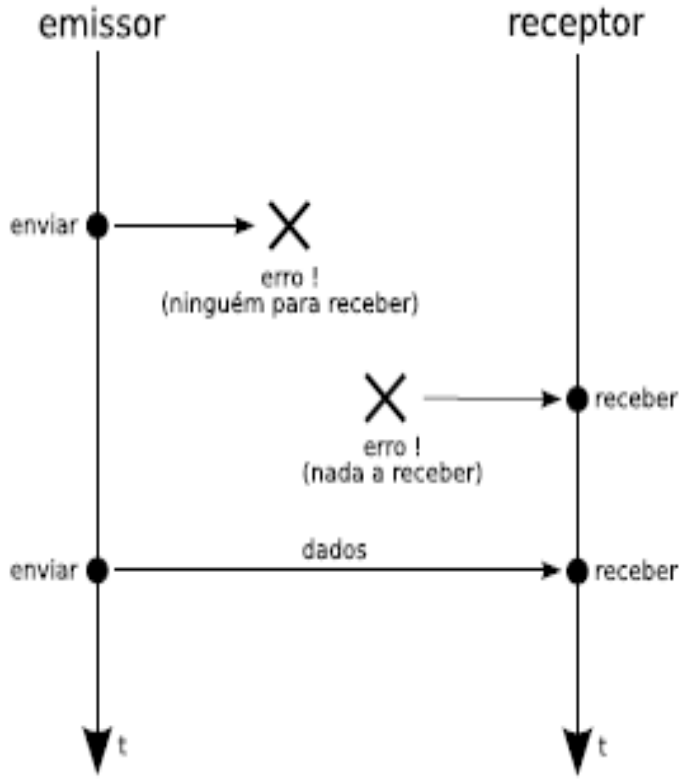


Formas de Comunicação

- Conexão
- Sincronismo exigido
- Formato dos dados
- **Necessidade de *Buffers***
- Quantidade de *Transmissores* e *Receptores*

Capacidade do Canal

Comunicacao Assíncrona $\Rightarrow$ *Buffers* são imprescindíveis



Buffers permitem armazenar temporariamente os dados em trânsito

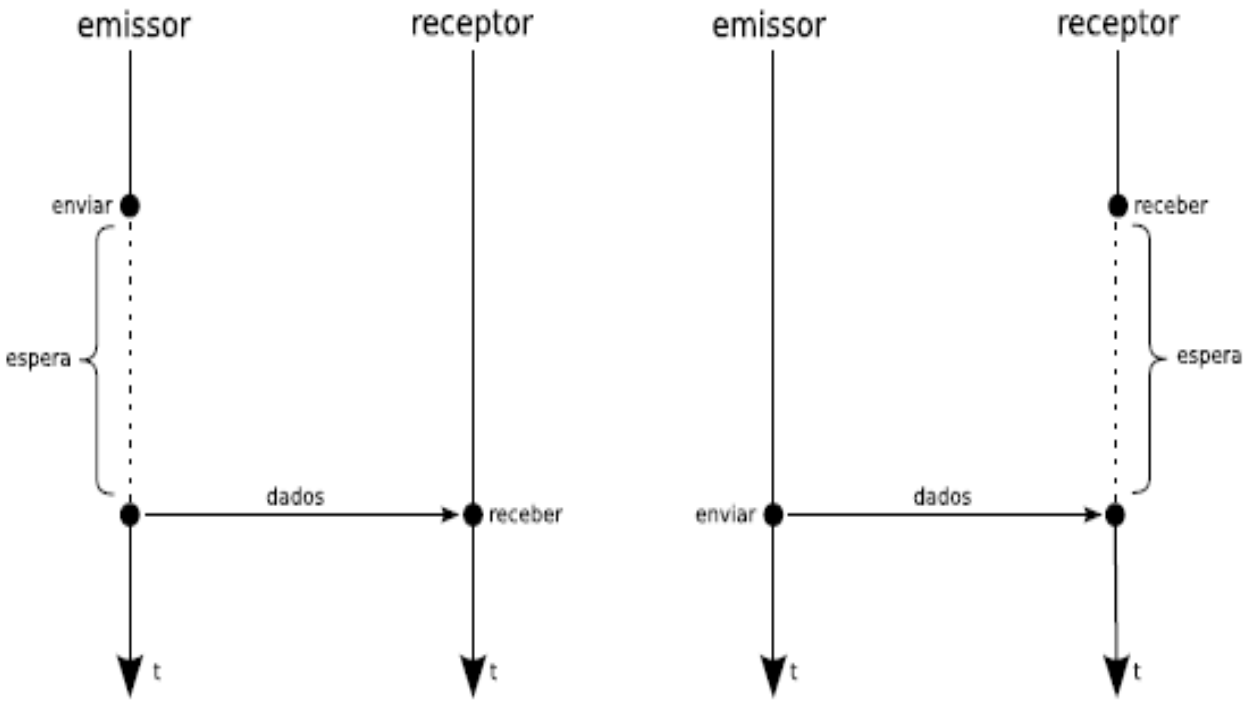
Capacidade do Canal

Capacidade nula (*n* = 0) : neste caso, o canal não pode armazenar dados.

Capacidade infinita (*n* = ∞) : o emissor sempre pode enviar dados, que serão armazenados no *buffer* do canal enquanto o receptor não os consumir.

Capacidade finita (0 < *n* < ∞**)** : uma quantidade finita de dados pode ser enviada pelo emissor sem que o receptor os consuma.

Buffer com Capacidade Nula



Comunicação síncrona → *Rendez-Vous* (Encontro)

Comunicação assíncrona → Inviável

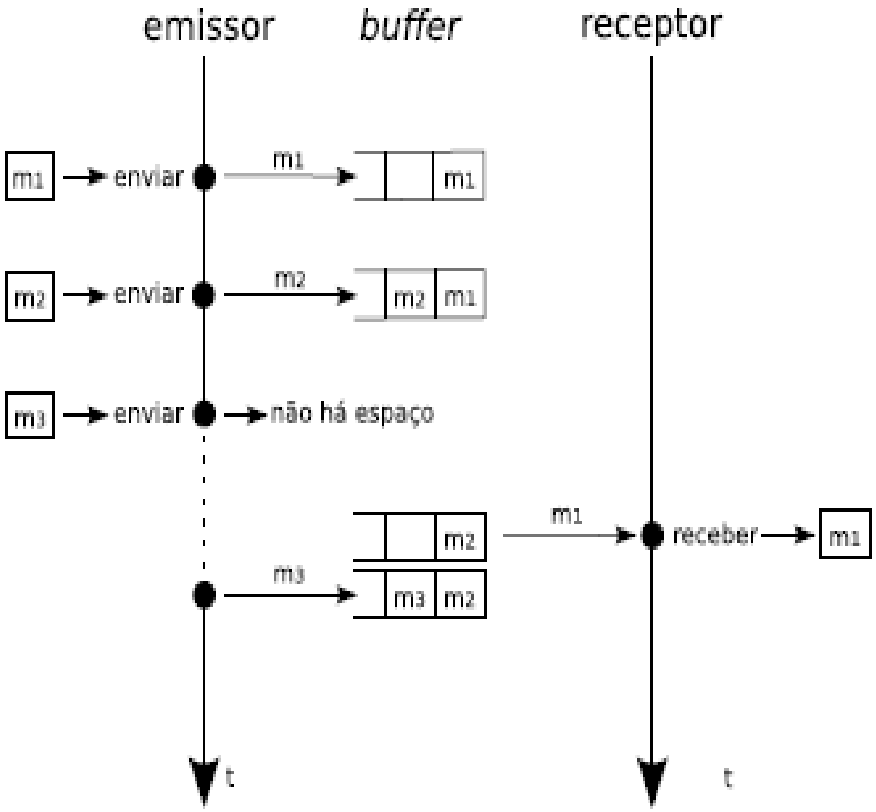
Buffer com Capacidade Infinita

•Não existe na prática, pois todos os sistemas de computação têm capacidade de memória e de armazenamento *finitas*.

•É útil no estudo dos algoritmos de comunicação e sincronização ← ***modelagem*** e ***análise*** menos complexas.

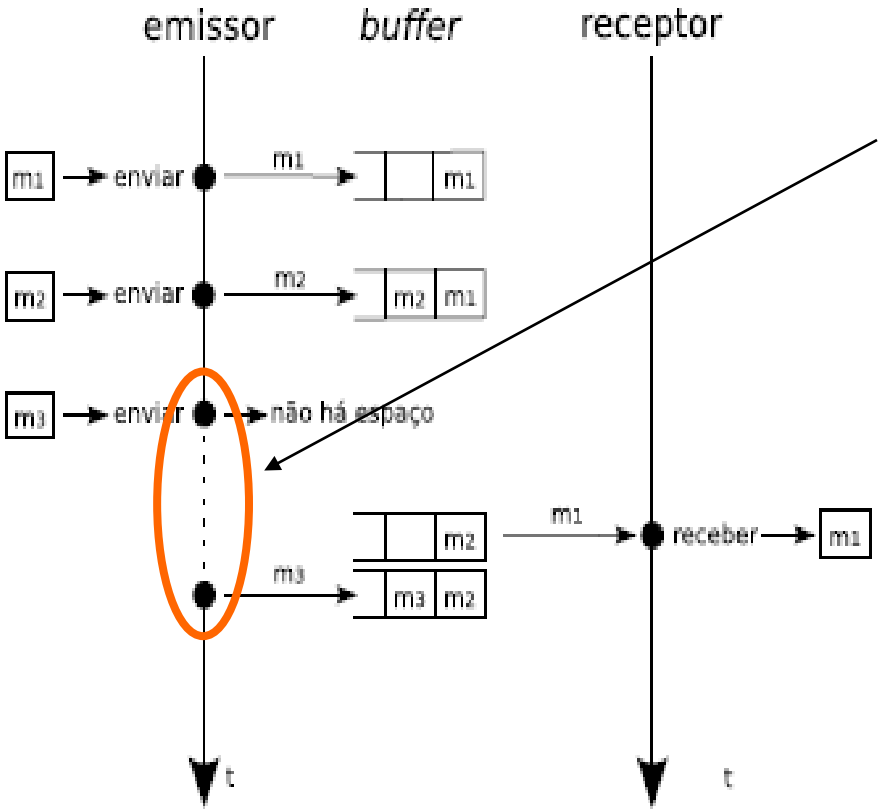
Buffer com Capacidade Finita

Ex: n = 2



Buffer com Capacidade Finita

Ex: n = 2



Canal já saturado:

1.Emissor bloqueado (comportamento síncrono) ou..

2.Retorno de erro (comportamento assíncrono)

Formas de Comunicação

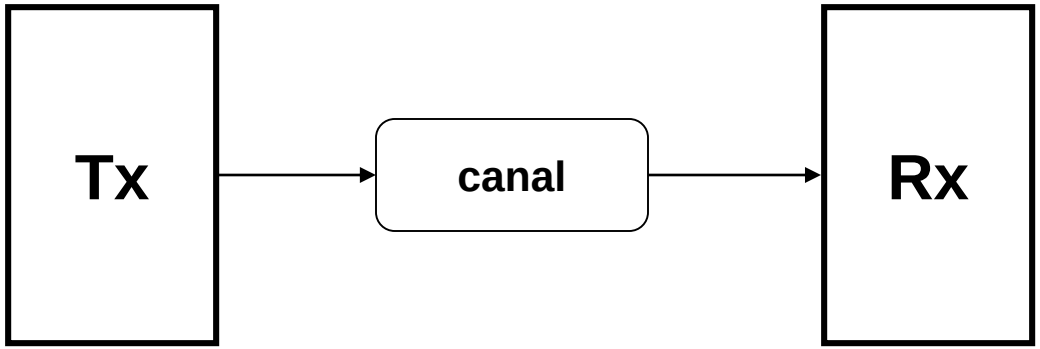
- Conexão
- Sincronismo exigido
- Formato dos dados
- Necessidade de *Buffers*
- **Quantidade de *Transmissores* e *Receptores***

Número de participantes

1:1 :

•Um emissor e um receptor interagem através do canal de comunicação

•Exemplo: *pipes Unix* e protocolo TCP.



Número de participantes

M:N :

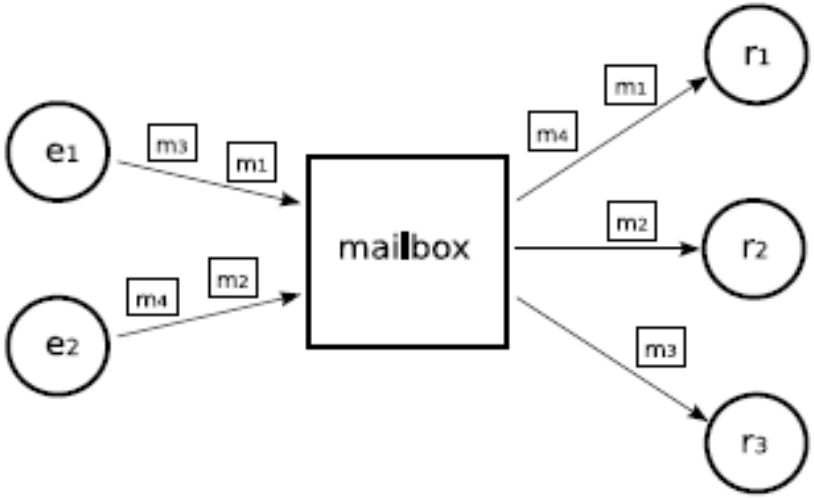
Um ou mais emissores enviam mensagens para um ou mais receptores.

Número de participantes

M:N :

Um ou mais emissores enviam mensagens para um ou mais receptores.

1 - Cada mensagem é recebida por **apenas um receptor**



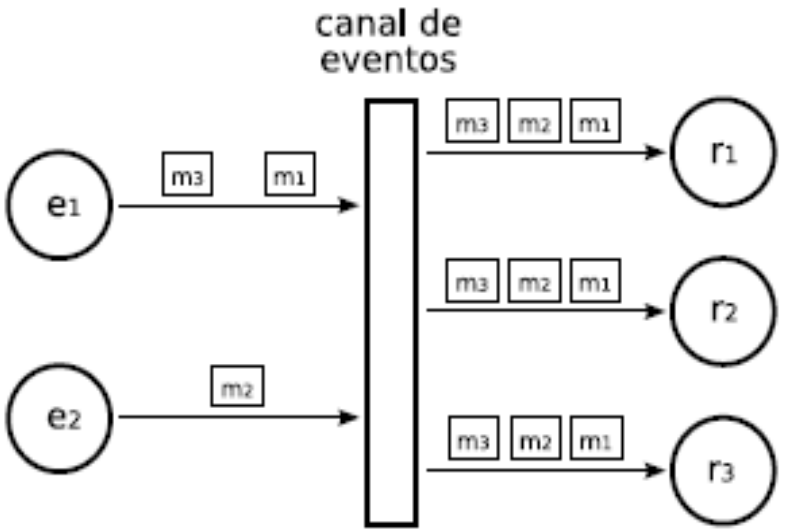
Número de participantes

M:N :

Um ou mais emissores enviam mensagens para um ou mais receptores.

2 – Cada mensagem é recebida por **todos os receptores**

Difusão (*multicast)*

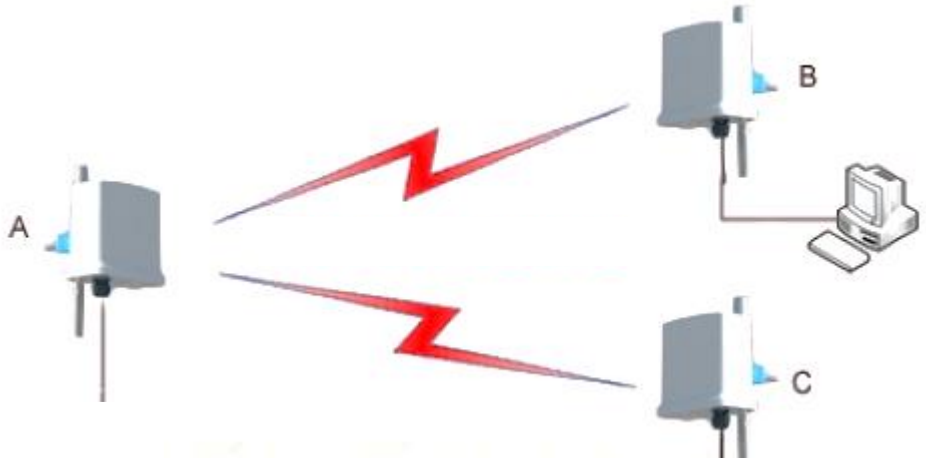


Confiabilidade do Canal

Confiabilidade do Canal

Canal confiável

O canal transporta os dados enviados para seus receptores, respeitando seus valores e a ordem em que foram enviados



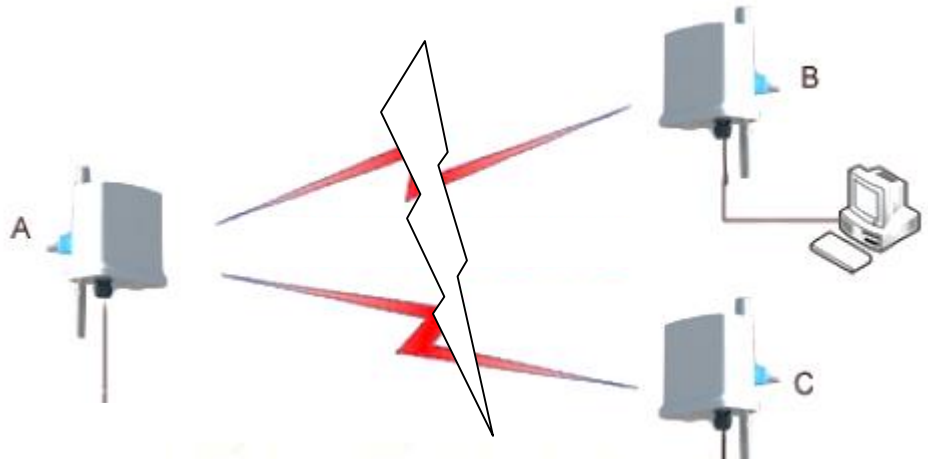
Confiabilidade do Canal

Canal confiável

O canal transporta os dados enviados para seus receptores, respeitando seus valores e a ordem em que foram enviados

Canal não-confiável

Caso contrário



Confiabilidade do Canal

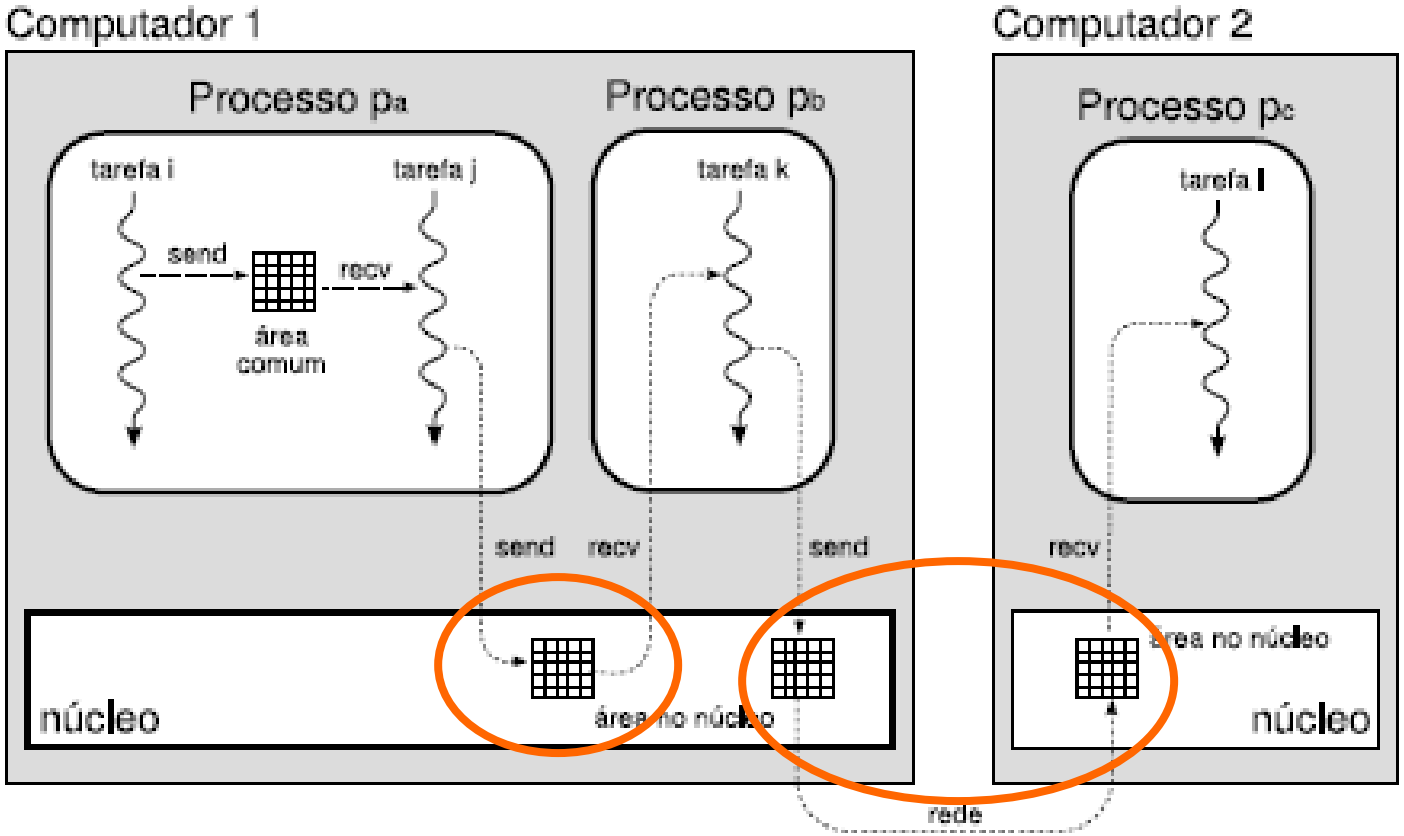
- **Perda de dados**
  - Nem todos os dados enviados chegam ao destino
  - Perdas de mensagens (comunicação orientada a mensagens)
  - Perdas de sequências de bytes (comunicação orientada a fluxo de dados)
- **Perda de integridade**
  - Os dados enviados chegam ao destino, mas...
  - Podem ocorrer modificações em seus valores
- **Perda da ordem**
  - Todos os dados enviados chegam íntegros ao destino, mas...
  - O canal não garante que eles serão entregues na ordem em que foram enviados
  - Um canal em que a ordem dos dados é garantida é denominado **canal FIFO** ou **canal ordenado**

IPC - Inter-Process Communication

Duas abordagens:

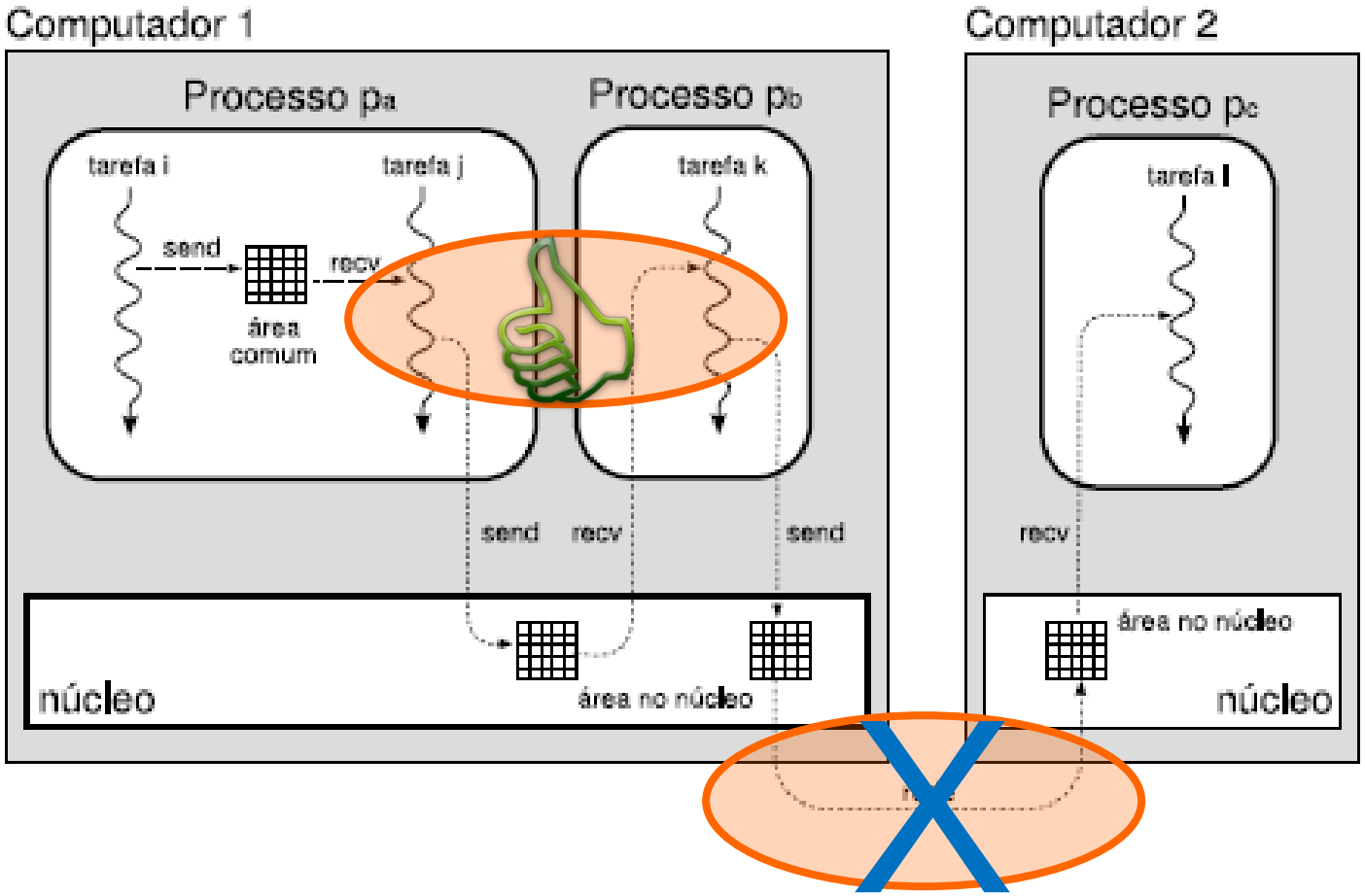
- Transmissão de mensagens
- Memória Compartilhada

Inter-processos / Inter-sistemas



Transmissão de mensagens:
Ineficiente para comunicação volumosa e frequente

Memória Compartilhada

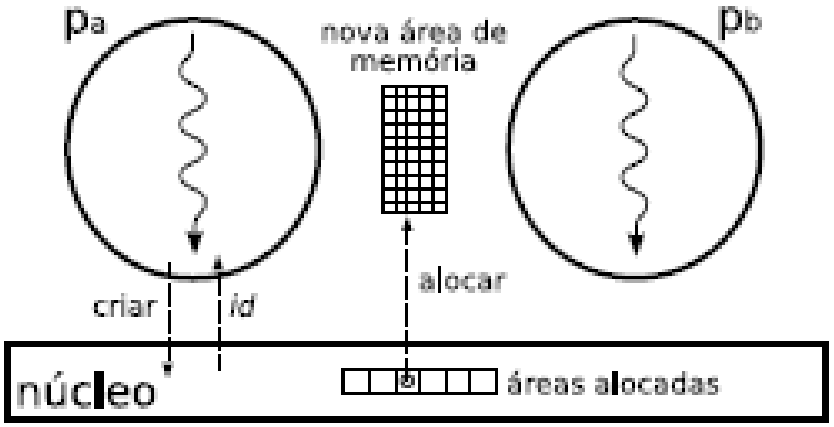


Memória Compartilhada

- Maioria dos SO`s oferece mecanismos para compartilhamento de memória
  - Núcleo **cria** a área compartilhada
  - Núcleo **gerencia** processos que a utilizam
  - **Controle de Acesso** ao conteúdo é definido pelo *processo* e implementado pelo Núcleo

Memória Compartilhada - Criação

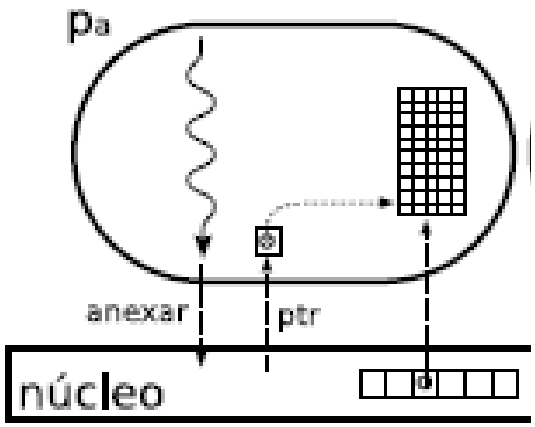
1 -



O processo *pa* solicita ao núcleo a **criação** de uma área de *memória compartilhada*, informando o **tamanho** e as **permissões** de acesso; o **retorno** dessa operação é um identificador (*id*) da área criada.

Memória Compartilhada - Criação

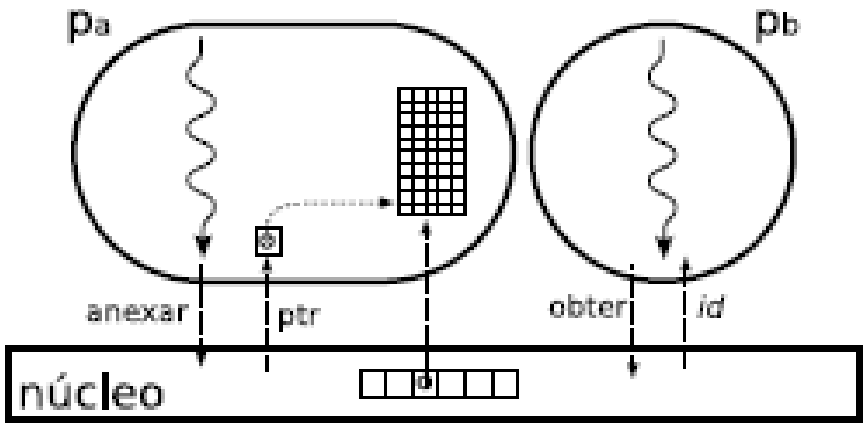
2 -



O processo *pa* **solicita** ao núcleo que a área recém-criada seja **anexada** ao seu **espaço de endereçamento**; esta operação **retorna** um **ponteiro** para a nova área de memória, que pode então ser **acessada** pelo *processo*.

Memória Compartilhada - Criação

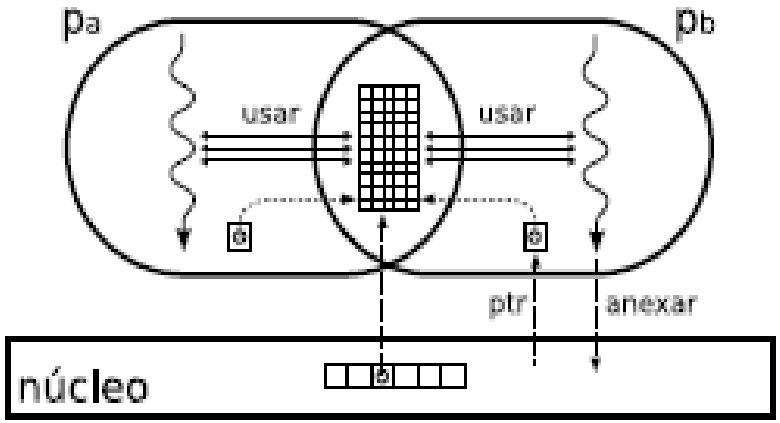
3 -



O processo *pb* **requisita** e **obtém** o identificador *id* da área de memória criada por *pa*.

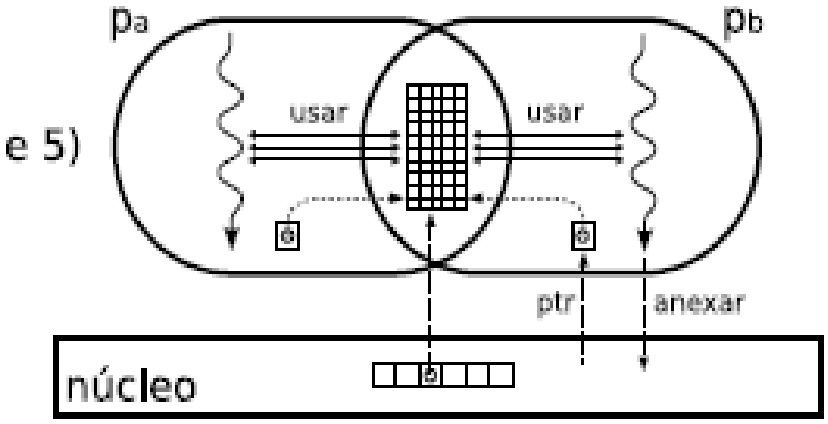
Memória Compartilhada - Criação

4 -



O processo *pb* **solicita** ao **núcleo** que a área de memória seja **anexada** ao seu **espaço de endereçamento** e recebe um **ponteiro** para o acesso à mesma.

Memória Compartilhada - Uso



Os processos *pa* e *pb* **acessam** a área de **memória compartilhada** usando os **ponteiros** informados pelo **núcleo**.

Memória Compartilhada - Uso

Nota:

Ao solicitar a **criação** da área de memória compartilhada, *pa* define as **permissões** de acesso à mesma; por isso, o pedido de **anexação** da área de memória feito por *pb* pode ser **recusado** pelo **núcleo**, se violar as **permissões** definidas por *pa.*

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main (int argc, char *argv[])
{
   int fd, value, *ptr;

   // Passos 1 a 3: abre/cria uma area de memoria compartilhada
   fd = shm_open("/sharedmem", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
   if(fd == -1) {
      perror ("shm_open");
      exit (1) ;
   }

   // Passos 1 a 3: ajusta o tamanho da area compartilhada
   if (ftruncate(fd, sizeof(value)) == -1) {
      perror ("ftruncate");
      exit (1) ;
   }

   // Passos 2 a 4: mapeia a area no espaco de enderecamento deste processo
   ptr = mmap(NULL, sizeof(value), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
   if(ptr == MAP_FAILED) {
      perror ("mmap");
      exit (1);
   }

   for (;;) {
      // Passo 5: escreve um valor aleatorio na area compartilhada
      value = random () % 1000 ;
      (*ptr) = value ;
      printf ("Wrote value %i\n", value) ;
      sleep (1);

      // Passo 5: le e imprime o conteudo da area compartilhada
      value = (*ptr) ;
      printf("Read value %i\n", value);
      sleep (1) ;
   }
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main (int argc, char *argv[])
{
  int fd, value, *ptr;

  // Passos 1 a 3: abre/cria uma area de memoria compartilhada
  fd = shm_open("/sharedmem", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
  if(fd == -1) {
    perror ("shm_open");
    exit (1) ;
  }

  // Passos 1 a 3: ajusta o tamanho da area compartilhada
  if (ftruncate(fd, sizeof(value)) == -1) {
    perror ("ftruncate");
    exit (1) ;
  }

  // Passos 2 a 4: mapeia a area no espaco de enderecamento deste processo
  ptr = mmap(NULL, sizeof(value), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
  if(ptr == MAP_FAILED) {
    perror ("mmap");
    exit (1);
  }

  for (;;) {
    // Passo 5: escreve um valor aleatorio na area compartilhada
    value = random () % 1000 ;
    (*ptr) = value ;
    printf ("Wrote value %i\n", value) ;
    sleep (1);

    // Passo 5: le e imprime o conteudo da area compartilhada
    value = (*ptr) ;
    printf("Read value %i\n", value);
    sleep (1) ;
  }
}
```

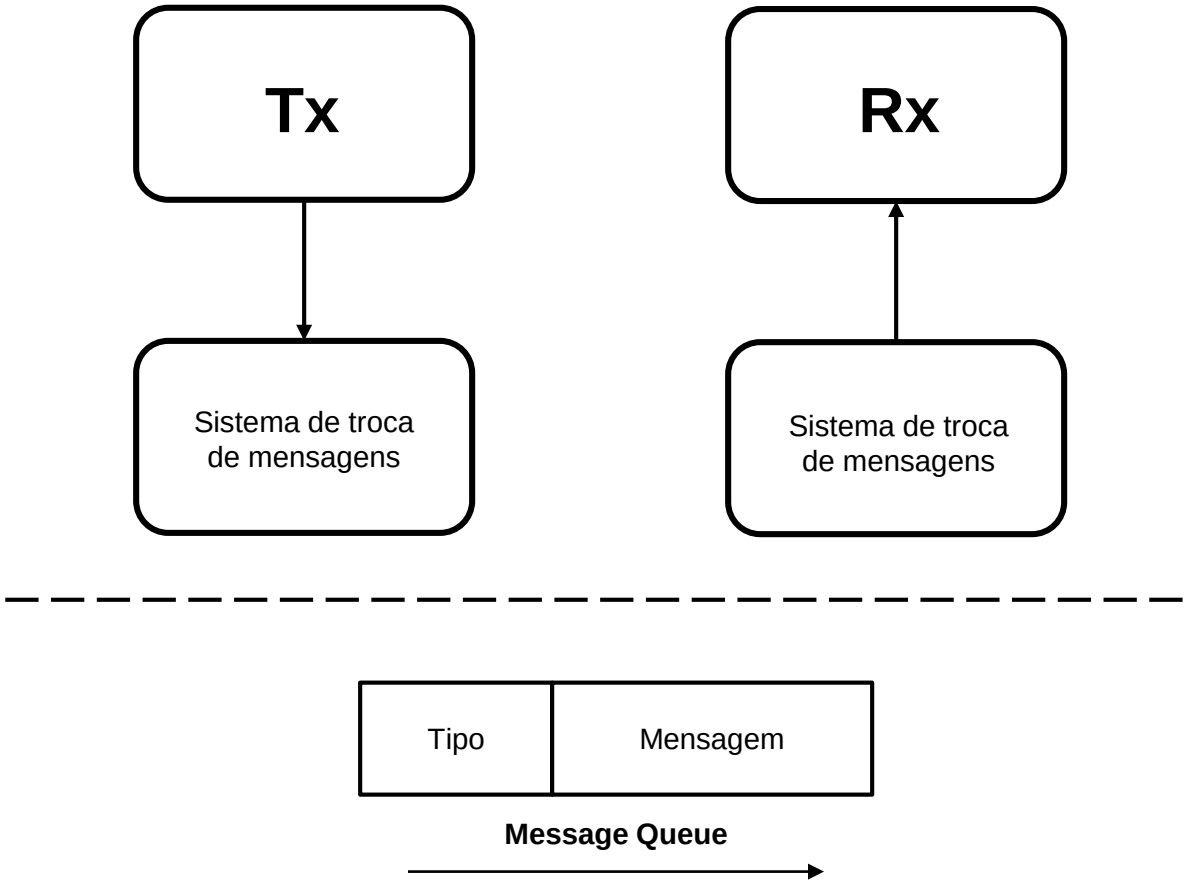
Exemplos de comunicação entre processos

- Unix Message Queues
- FreeRTOS Message Queues
- Pipes

UNIX Message Queues - POSIX

UNIX Message queues

Syscall POSIX - "Portable Operating System Interface [for Unix]" - Padrão IEEE



UNIX Message queues

Syscall POSIX - "Portable Operating System Interface [for Unix]" - Padrão IEEE

- **mq_open:** abre ou cria uma nova fila;
- **mq_setattr e mq_getattr**: ajustar ou obter atributos da fila;
- **mq_send**: envia mensagem para fila;
- **mq_timedsend:** define prazo máximo de espera;
- **mq_receive**: recebe mensagem da fila;
- **mq_timedreceive:** define prazo máximo de espera;
- **mq_close**: fecha o descritor da fila criado por mq_open;
- **mq_unlink**: remove a fila do sistema.

UNIX Message queues
- Receptor

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mqueue.h>
4 #include <sys/stat.h>
5
6 #define QUEUE "/my_queue"
7
8 int main (int argc, char *argv[])
9 {
10          mqd_t               queue; // descritor da fila de mensagens
11          Structmq_attr       attr; // atributos da fila de mensagens
12          Int                 msg ; // mensagens contendo um inteiro
13
14          // define os atributos da fila de mensagens
15          attr.mq_maxmsg      = 10 ;      // capacidade para 10 mensagens
16          attr.mq_msgsize     = sizeof(msg) ; // tamanho de cada mensagem
17          attr.mq_flags       = 0 ;
```

```
21 // caso a fila exista, remove-a para destruir seu conteudo antigo
22 mq_unlink (QUEUE) ;
23
24 // abre ou cria a fila
25 if ( (queue = mq_open (QUEUE, O_RDWR|O_CREAT, 0666, &attr)) = = -1)
26 {
27          perror ("mq_open");
28          exit (1);
29 }
30
31 // recebe cada mensagem e imprime seu conteudo
32 while (1)
33 {
34          if ( (mq_receive (queue, (void*) &msg, sizeof(msg), 0)) = = -1)
35          {
36                    perror("mq_receive:") ;
37                    exit (1) ;
38          }
39          printf ("Received msg value %d\n", msg);
40 }
41 }
```

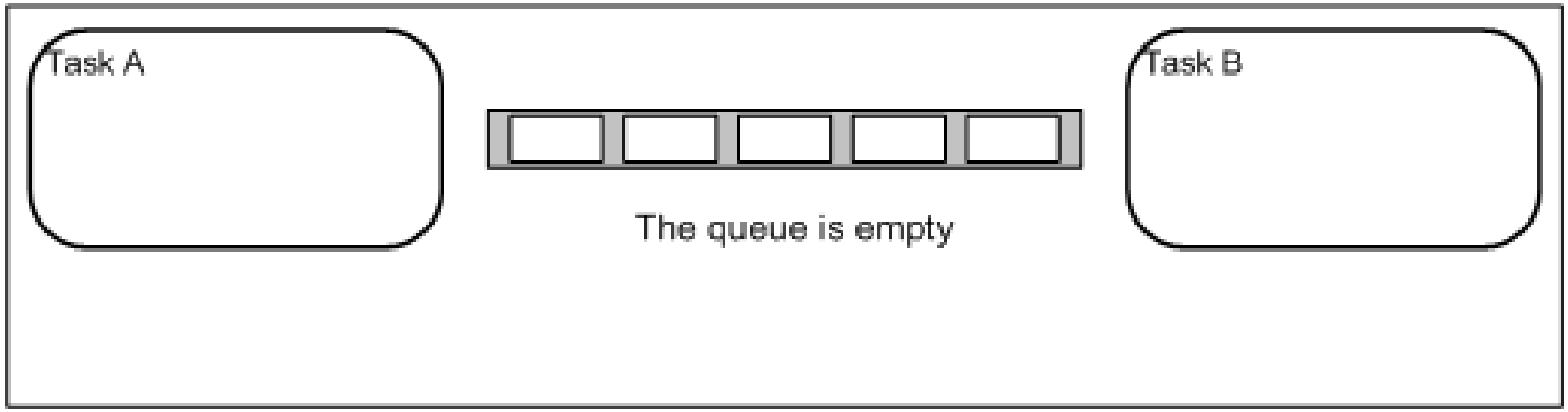
UNIX Message queues
- Transmissor

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mqueue.h>
4 #include <sys/stat.h>
5
6 #define QUEUE "/my_queue"
7
int main (int argc, char *argv[])
9 {
10          mqd_t     queue; // descritor da fila
11          Int       msg; // mensagem a enviar
12
13          // abre a fila de mensagens, se existir
14          if( (queue = mq_open (QUEUE, O_RDWR)) = = -1)
15          {
16                    perror ("mq_open");
17                    exit (1);
18          }
```

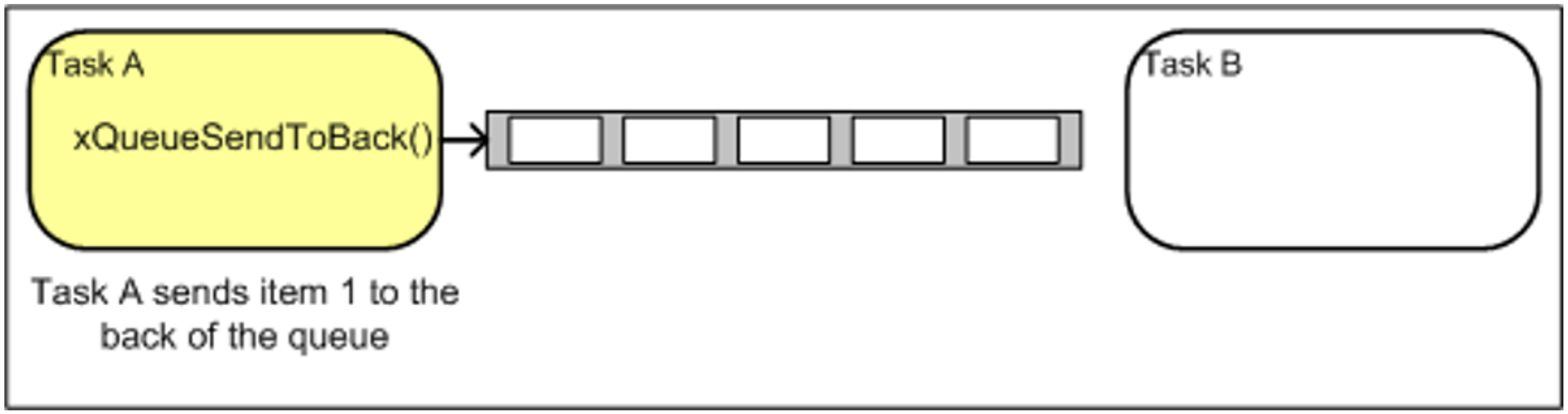
```
20 while (1)

21 {
22          msg = random() % 100 ; // valor entre 0 e 99
23
24          // envia a mensagem
25          if ( mq_send (queue, (void*) &msg, sizeof(msg), 0) = = -1)
26          {
27                    perror ("mq_send");
28                    exit (1);
29          }
30          printf ("Sent message with value %d\n", msg);
31          sleep (1) ;
32 }
33 }
```

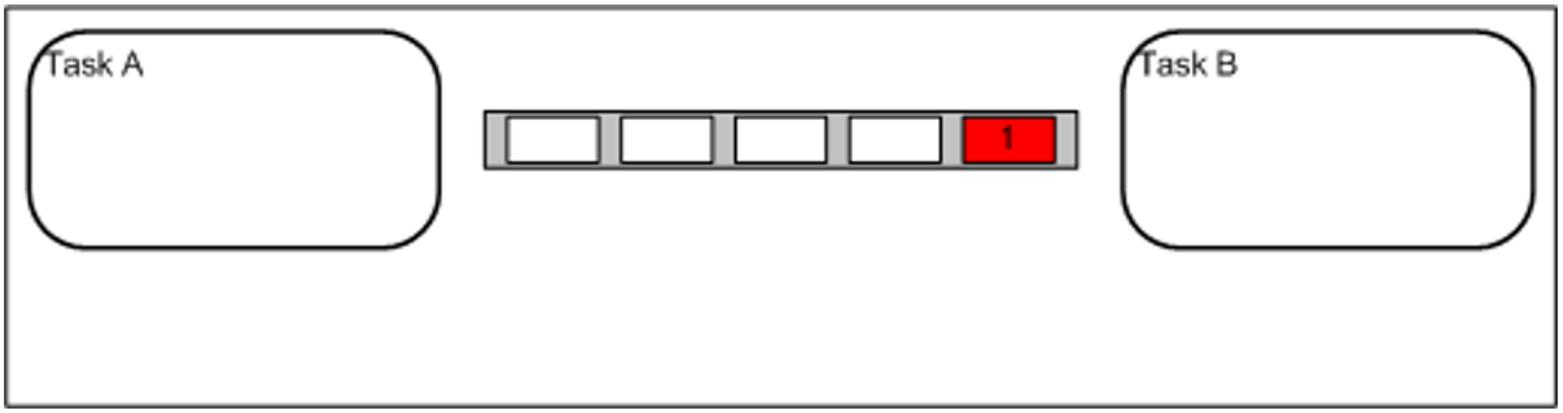
Message queues - FreeRTOS



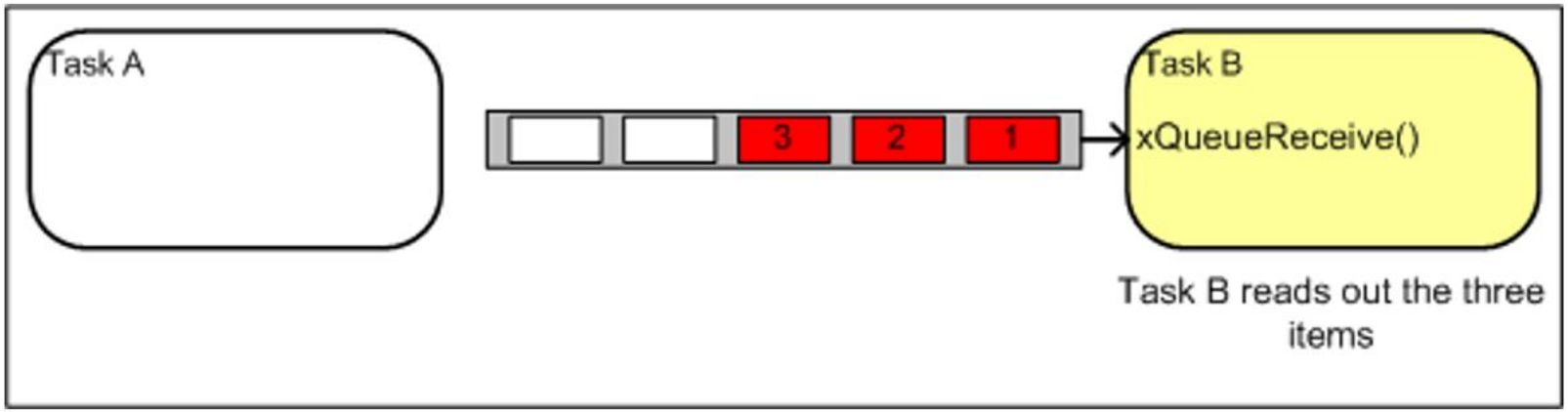
Message queues - FreeRTOS



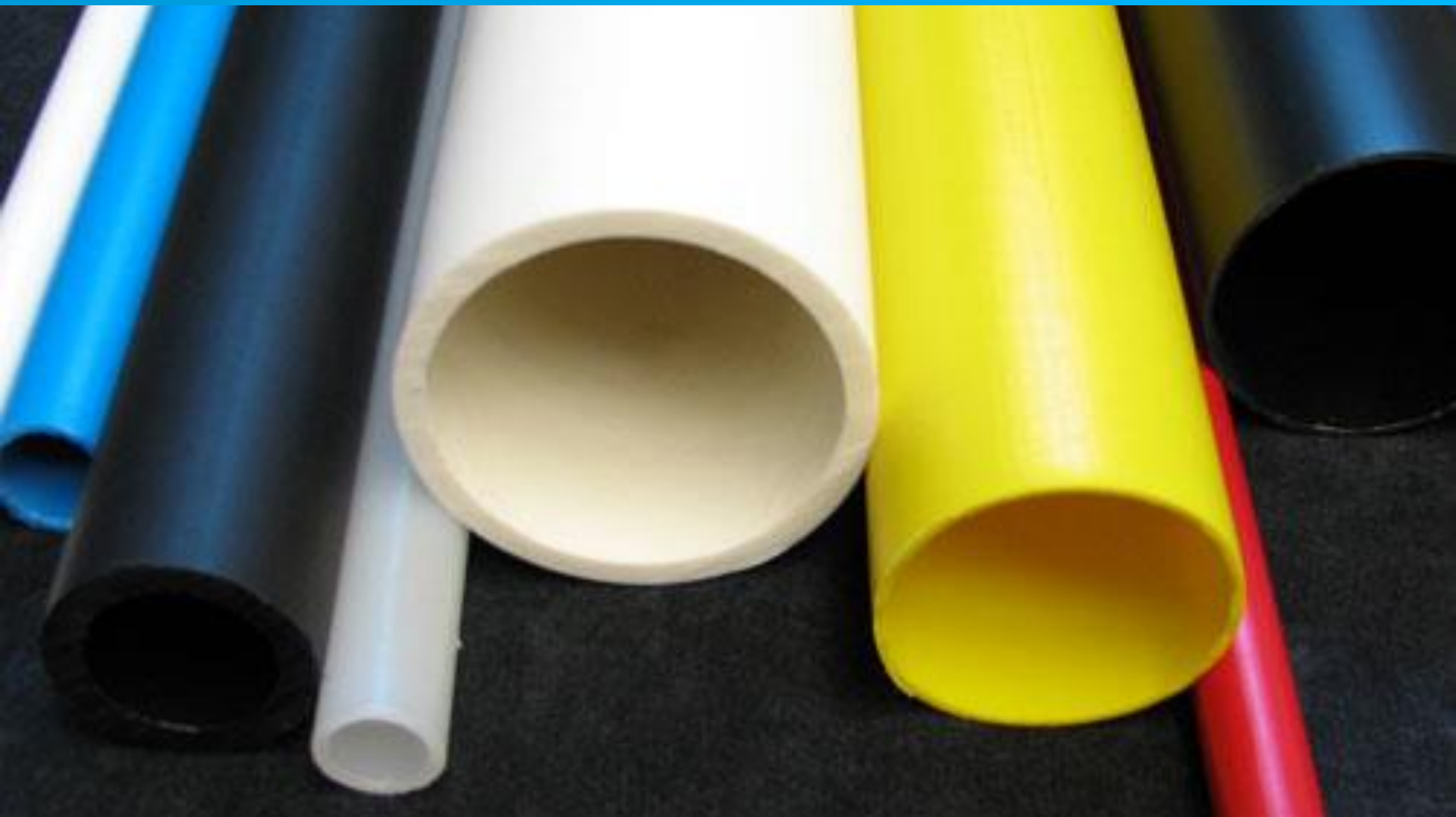
Message queues - FreeRTOS



Message queues - FreeRTOS



UNIX Pipes



Pipes

•Um dos mecanismos de comunicação entre processos mais simples de usar no ambiente UNIX.

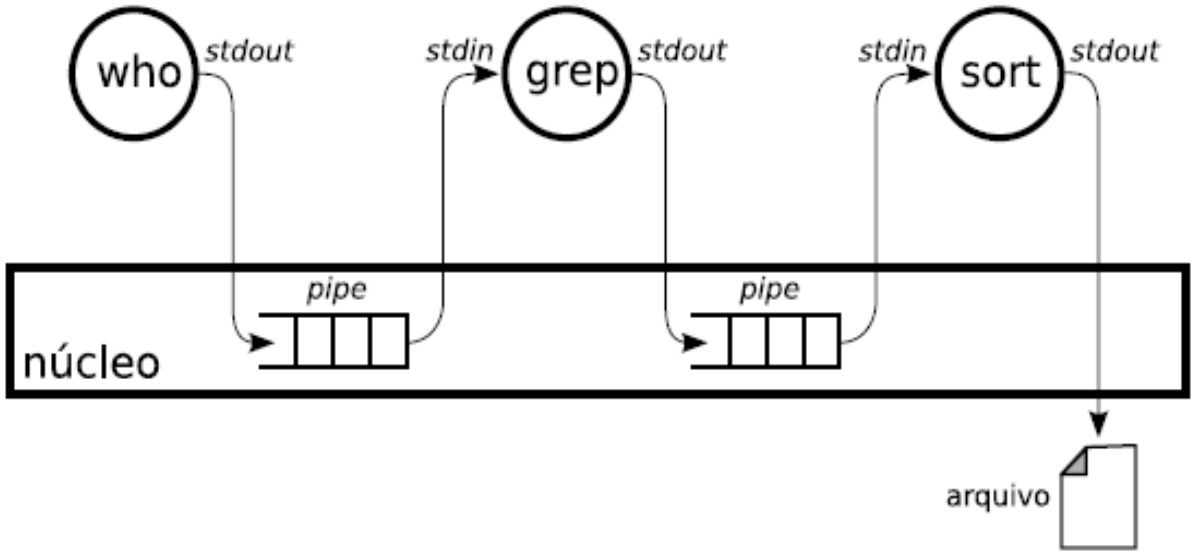
•No interpretador de comandos:

-Usado para conectar a saída padrão (*stdout*) de um comando à entrada padrão (*stdin*) de outro comando *(processo);*

-Permitindo a comunicação entre eles.

Pipes

```
# who | grep marcos | sort > login-marcos.txt
```



•Todos os processos envolvidos são lançados simultaneamente; suas ações são coordenadas pelo comportamento **síncrono** dos *pipes.*

Pipes

- O *pipe* é um canal de comunicação unidirecional entre dois processos (1:1)
- Possui capacidade finita
  - Os *pipes* do Linux armazenam 4 KBytes (default)
- Visto pelos processos como um arquivo
  - A comunicação que ele oferece é baseada em fluxo
- Envio e recepção de dados feitos pelas chamadas de sistema *write* e *read*