

# Engenharia de Computação



## Arquitetura de Sistemas Operacionais

### Comunicação entre Processos

**Prof. Martín Vigil**

Adaptado de Prof. Anderson Luiz Fernandes Perez

# Conteúdo

- Introdução
- Modelos de Interação entre Produtores e Consumidores
- Comunicação no Modelo Computacional
- Memória Compartilhada
- PIPEs
- Comunicação Cliente-Servidor (*socket*)

# Introdução

- Os **processos** executando em um sistema computacional **podem ser** do tipo **independentes** ou **cooperativos**.
- Os **processos independentes** não compartilham dados com os demais processos.
- Os **processos cooperativos** compartilham algum tipo de dado com um ou mais processos.

# Introdução

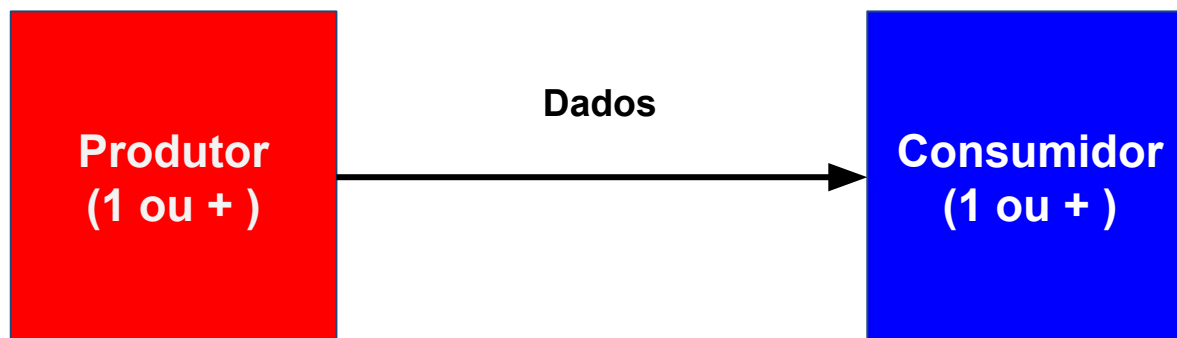
- A cooperação entre processos oferece as seguintes facilidades:
  - **Compartilhamento de Informações:** acesso concorrente a dados, por exemplo, mais de um processo acessando um arquivo em disco.
  - **Velocidade do Processamento:** a divisão de uma tarefa em subtarefas permite minimizar o tempo de processamento. O **processamento paralelo** acontece se houver mais de um processador disponível.

# Introdução

- A cooperação entre processos oferece as seguintes facilidades:
  - **Modularidade**: um sistema com muitas funcionalidades pode ser dividido em várias threads.
  - **Conveniência**: um único usuário pode usufruir das vantagens do compartilhamento de informações entre processos.

# Introdução

- A interação ou comunicação entre dois ou mais processos pode ser caracterizada por **produtores** e **consumidores**



# Modelos de Interação entre Produtores e Consumidores



- O **modelo de interação** entre processos **determina** muita **vezes qual ou quais mecanismos de comunicação** devem ser utilizados efetivamente por uma aplicação.
- Um **modelo de interação entre processos é determinado por dois aspectos**:
  - Número de processos interlocutores envolvidos na comunicação;
  - O papel desempenhado por cada um dos processos interlocutores.

# Modelos de Interação entre Produtores e Consumidores

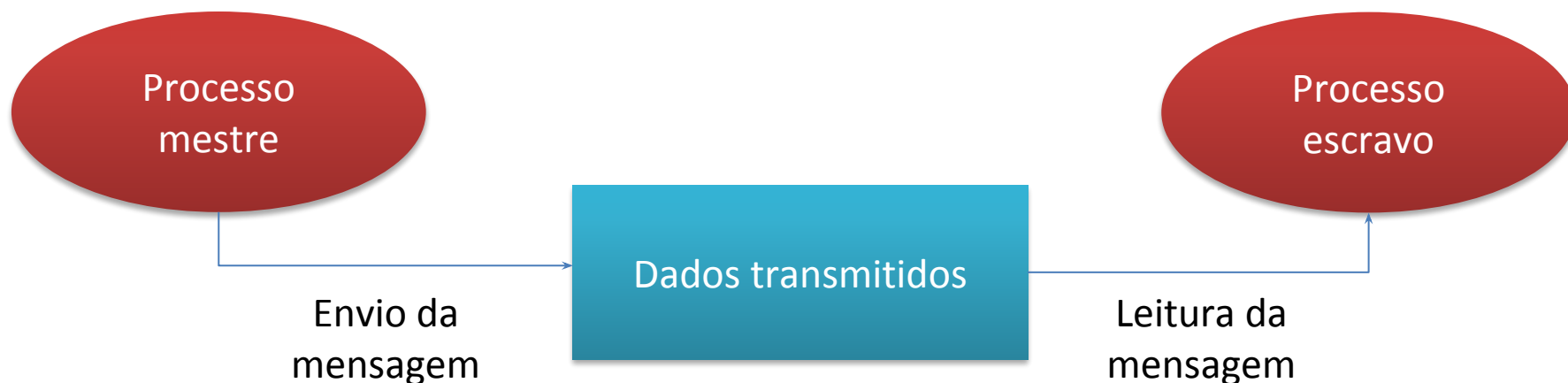


- Modelo de Interação Um para Um (mestre-escravo)
  - Baseia-se na **associação estrita entre dois processos**, sendo estabelecida uma ligação entre ambos.
  - O **processos escravo (leitor)** tem a sua atividade **totalmente controlada pelo processo mestre (produtor)**.
  - O **canal de comunicação é fixo e a associação dos processos a este é pré-estabelecida**, ou seja, cada um deve conhecer previamente a identificação do outro.



# Modelos de Interação entre Produtores e Consumidores

- Modelo de Interação Um para Um (mestre-escravo)



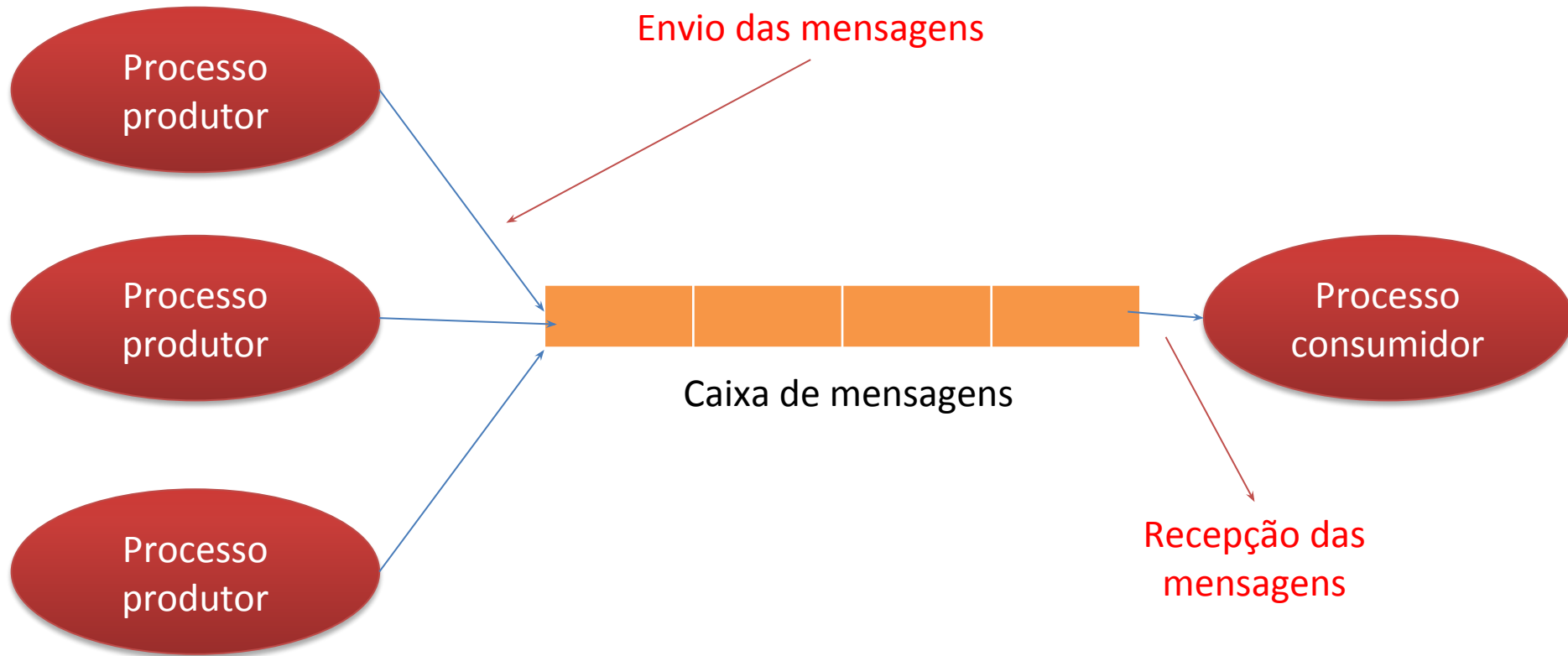
# Modelos de Interação entre Produtores e Consumidores



- Modelo de Interação Muitos para Um
  - O modelo **Muitos para Um**, também conhecido como **correio**, baseia-se na possibilidade de transferência de dados em modo **assíncrono** sob a forma de mensagem.
  - As **mensagens são enviadas individualmente** por um conjunto de processos produtores a um processo consumidor que está preparado para recebê-las.
  - O **canal de comunicação é criado previamente pelo processo consumidor** e o seu nome é conhecido pelos processos produtores.
  - O **processo consumidor é visto como um servidor que atende as solicitações de vários clientes.**

# Modelos de Interação entre Produtores e Consumidores

- Modelo de Interação Muitos para Um



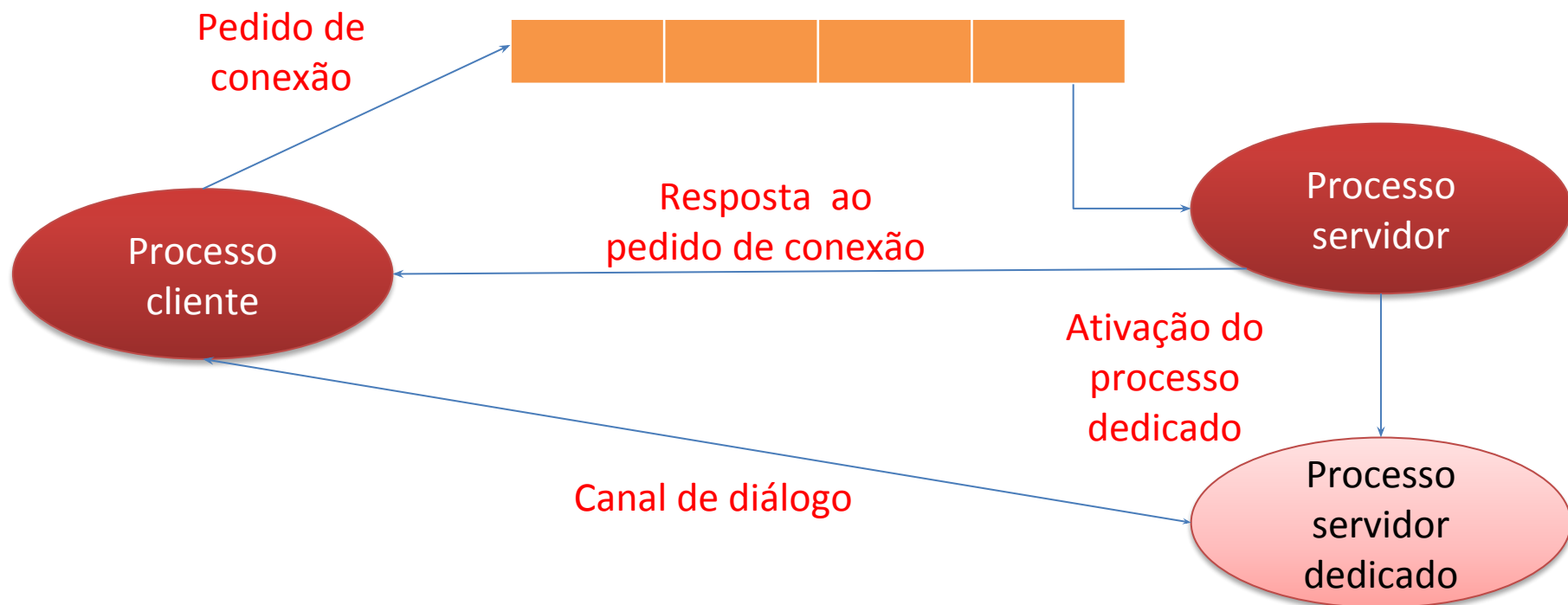
# Modelos de Interação entre Produtores e Consumidores



- Modelo de Interação Um para Um de Vários
  - Também conhecido como **Diálogo**, é um **modelo híbrido entre os modelos Um para Um e Muitos para Um**.
  - Neste modelo é estabelecido **um canal fixo entre dois processos**, criado de forma dinâmica.
  - Um processo, normalmente um **cliente**, **deve requisitar o estabelecimento da ligação** enviando uma mensagem para um canal previamente criado pelo servidor.
  - O **resultado da ligação** entre o cliente e o servidor é **um novo canal ao qual o cliente e o novo processo (processo ou thread) servidor dedicado ficam automaticamente associados**.
  - A associação é temporária e durará apenas o tempo da interação entre o processo cliente e o processo servidor.

# Modelos de Interação entre Produtores e Consumidores

- Modelo de Interação Um para Um de Vários



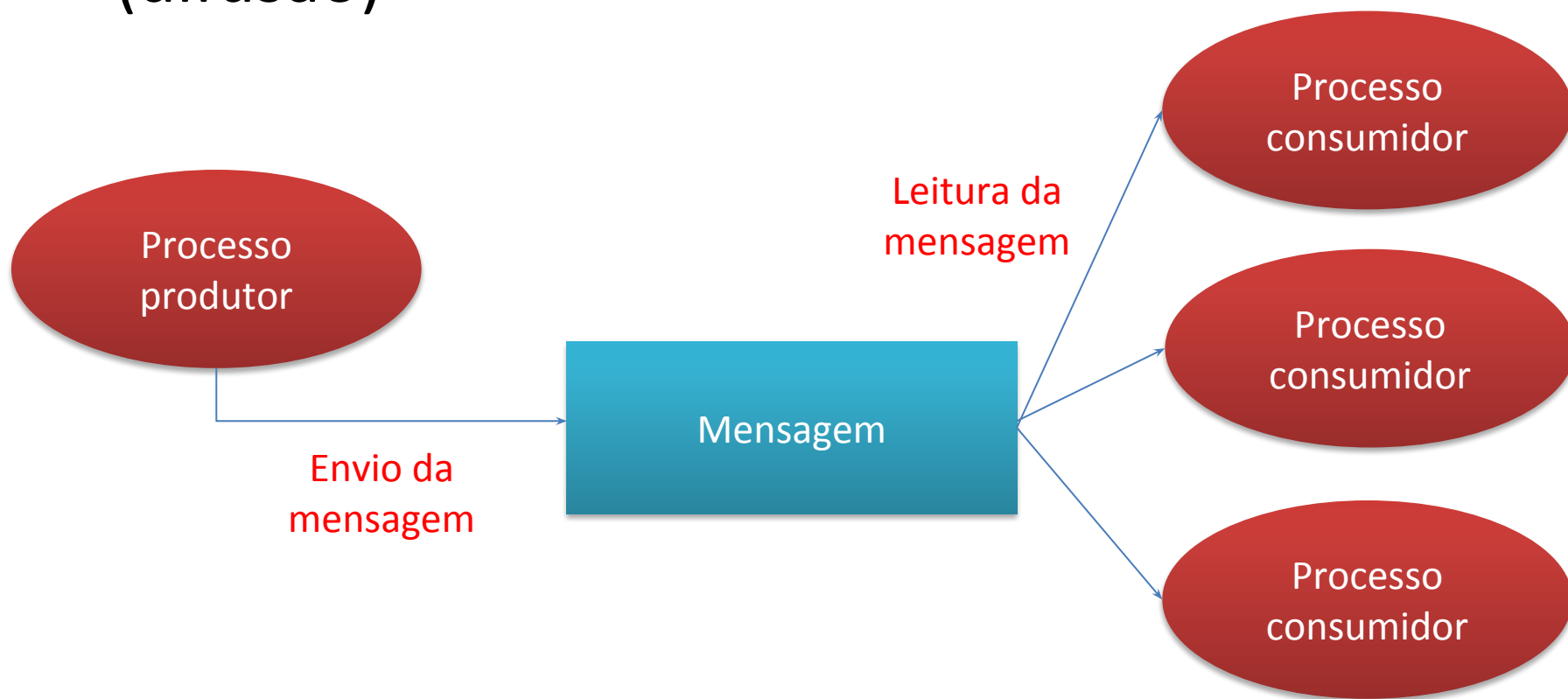
# Modelos de Interação entre Produtores e Consumidores



- Modelo de Interação Um para Muitos (difusão)
  - Neste modelo de interação **um processo produtor envia uma mesma informação para vários processos consumidores** ou para um grupo de consumidores previamente identificado.
  - Este tipo de comunicação **é muito utilizado para notificações entre processos.**
  - O gerenciador de janelas utiliza esse mecanismo quando o **usuário solicita o desligamento do computador** o gerenciador de janelas envia uma mensagem de encerramento para todas as janelas de aplicações em execução.

# Modelos de Interação entre Produtores e Consumidores

- Modelo de Interação Um para Muitos (difusão)



# Modelos de Interação entre Produtores e Consumidores



- Modelo de Interação Muitos para Muitos
  - Neste modelo de interação todos os processos podem ser produtores e consumidores de mensagens, alternando os papéis em tempo de execução.
  - Esse modelo é utilizado pelo *clipboard* do Windows, onde todas as janelas podem tanto produzir quando consumir informações.



# Modelos de Interação entre Produtores e Consumidores



- Resumo

Modelo de interação	Situação no mundo real
Um-para-um	Ligação telefônica
Muitos-para-um	Serviço postal, coleta de lixo
Um-para-muitos	Televisão, rádio, painéis publicitários.
Muitos-para-muitos	Grupos de WhatsApp, Diário Oficial da União, Blockchain Bitcoin

# Comunicação no Modelo Computacional



- A comunicação no modelo computacional é realizada por **mecanismos** disponibilizados pelo **sistema operacional** ou disponibilizados nos **ambientes de programação**.
- A comunicação entre processos é suportada por um objeto do tipo ***canal de comunicação***.
- A **transferência de informações** entre processos pode ser vista como **resultado da invocação de operações sobre um objeto canal**.

# Comunicação no Modelo Computacional

- Objeto do tipo Canal de Comunicação



# Comunicação no Modelo Computacional



- As operações realizadas em um canal de comunicação podem ser:
  - **Criar**: cria um canal de comunicação que será utilizado por dois ou mais processos.
  - **Associar**: associa o processo a um canal de comunicação.
  - **Enviar**: envia dados para o canal.
  - **Receber**: recebe dados do canal.
  - **Terminar**: fecha o canal de comunicação, sinalizando que a comunicação foi realizada.
  - **Eliminar**: elimina o canal de comunicação.

# Comunicação no Modelo Computacional



- A **comunicação entre processos pode se dar em diversos contextos**, sendo:
  - Processos executando em um mesmo computador com relação hierárquica (processo pai e processo filho);
  - Processos executando em um mesmo computador sem relação hierárquica;
  - Processos executando em computadores diferentes com o mesmo sistema operacional;
  - Processos executando em computadores diferentes com sistemas operacionais diferentes.

# Comunicação no Modelo Computacional



- As mensagens trafegadas em um canal de comunicação pode ser estruturadas de acordo com as vertentes:
  - **Interna**: determina a **codificação dos dados** trocados na comunicação.
  - Na vertente interna os canais de comunicação podem ser:
    - **Opacos**: os dados têm de ser explicitamente codificados e interpretados pelos **processos interlocutores**;
    - **Estruturados**: a comunicação impõe uma estrutura fixa para as mensagens trocadas ou então suporta a transferência de informação do tipo anexa aos dados no conteúdo das mensagens.

# Comunicação no Modelo Computacional



- As mensagens trafegadas em um canal de comunicação pode ser estruturadas de acordo com as vertentes:
  - **Externa**: foca a **delimitação** e a preservação das fronteiras entre as diferentes mensagens enviadas.
  - Na vertente externa os canais de comunicação podem ser orientados a:
    - **Mensagens-pacote**: a comunicação realiza-se através de troca de mensagens individualizadas, cuja fronteira é preservada e imposta na recepção;
    - **Streams**: a comunicação processa-se através da escrita e da leitura de sequências ordenadas de bytes.

# Comunicação no Modelo Computacional



- A comunicação entre processos não somente determina um meio de troca de informações entre processos, mas também um mecanismo para sincronizar as ações dos processos comunicantes.
- A **semântica na comunicação de processos determina o comportamento do processo ao receber e enviar um mensagem**, e podem ser:
  - **Síncrona:** o produtor fica bloqueado até que o consumidor receba a mensagem e acesse o seu conteúdo.
    - **Cliente-Servidor:** o processo produtor (cliente) fica bloqueado até que o consumidor (servidor) tenha recebido a mensagem e enviado uma mensagem de resposta.
  - **Assíncrona:** o produtor envia a mensagem e continua a execução, assim que esta tenha sido armazenada no canal de forma temporária até que seja recebida pelo consumidor.

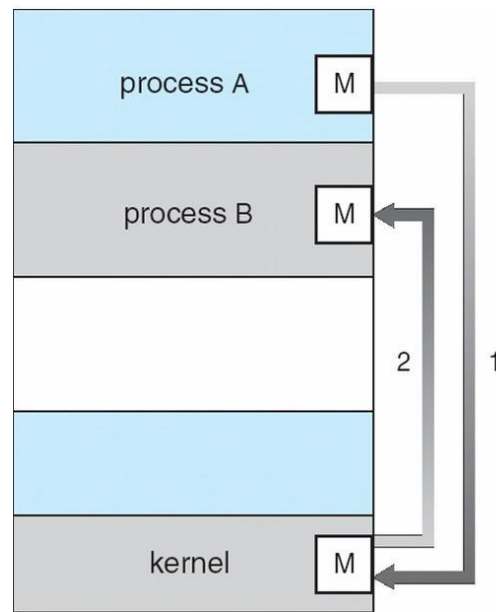


# Comunicação no Modelo Computacional

- Um canal de comunicação pode ser implementado de duas formas distintas:

## Via kernel

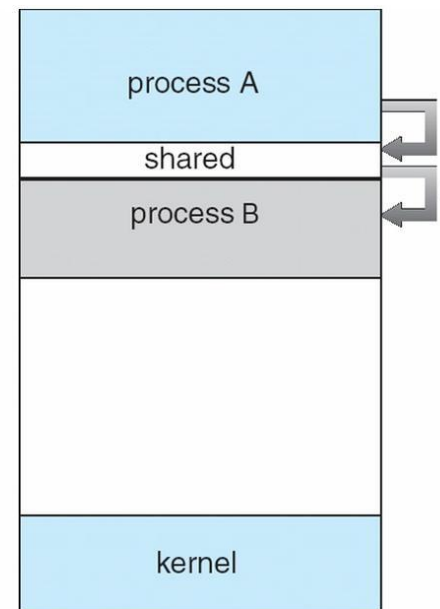
Evita conflitos  
Fácil implementar  
Sist. distribuídos



(a)

## Via memória compartilhada

Transmissão mais rápida



(b)

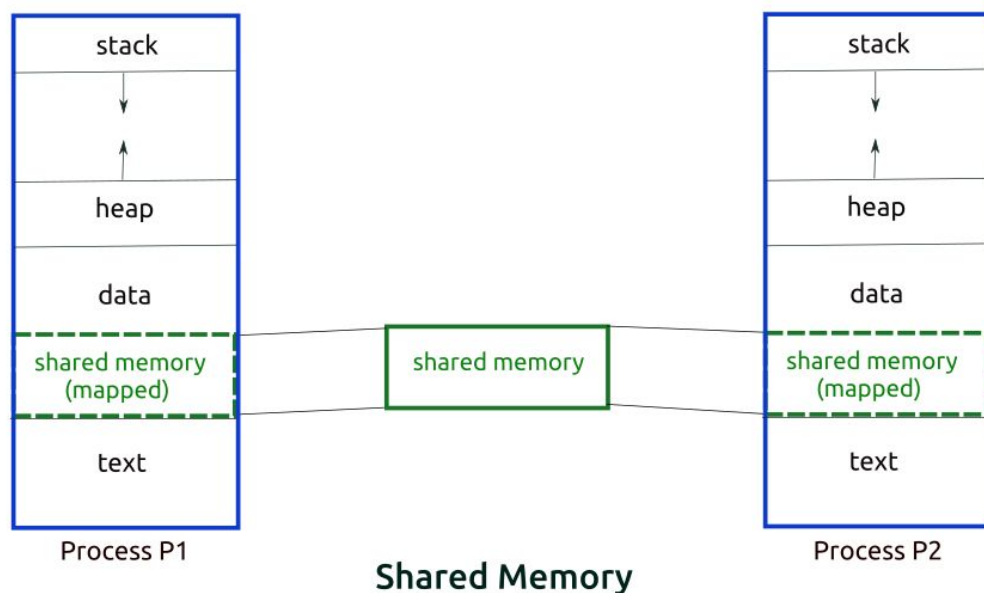
# Comunicação no Modelo Computacional



- Existem basicamente **três classes de canais de comunicação**:
  - Memória compartilhada;
  - Caixas de mensagens;
  - Conexões virtuais (stream).

# Memória Compartilhada

- Processos podem criar e associar áreas de memória compartilhada e mapeá-las em seu espaço de endereçamento.



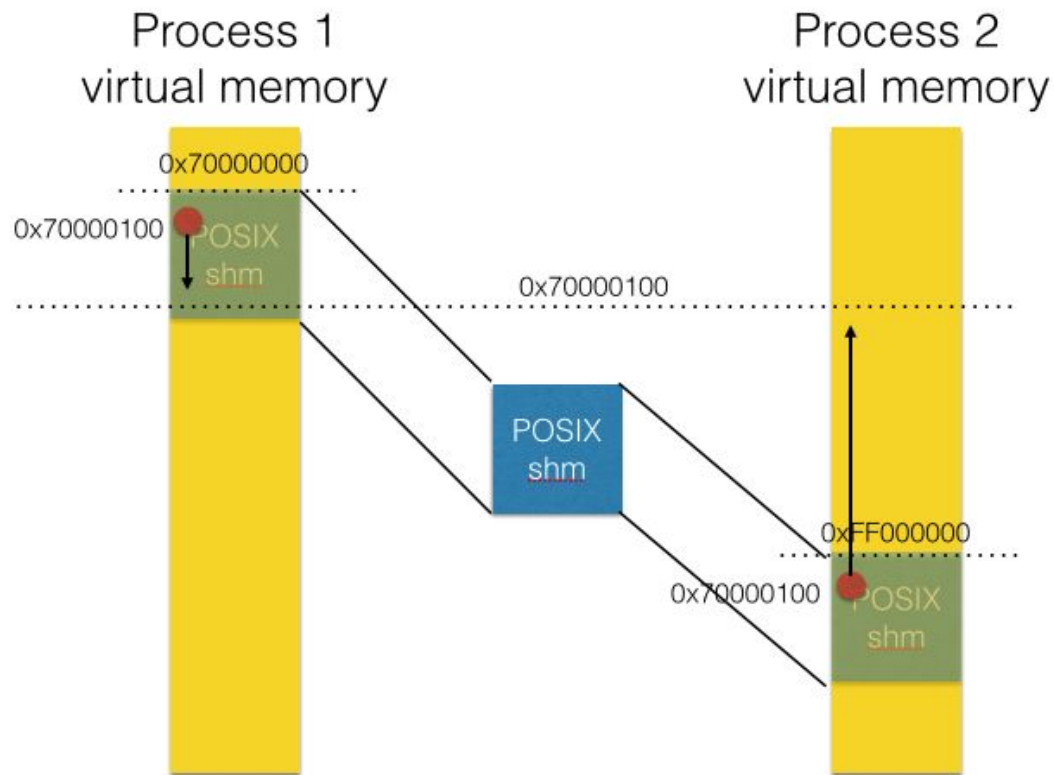
# Memória Compartilhada

- Um grupo de processos pode manipular a mesma área de memória sem que ocorram exceções devido à violação no espaço de endereçamento de cada um.
- As áreas de memória compartilhada podem ser mapeadas no contexto de cada processo naturalmente, em diferentes espaços virtuais.

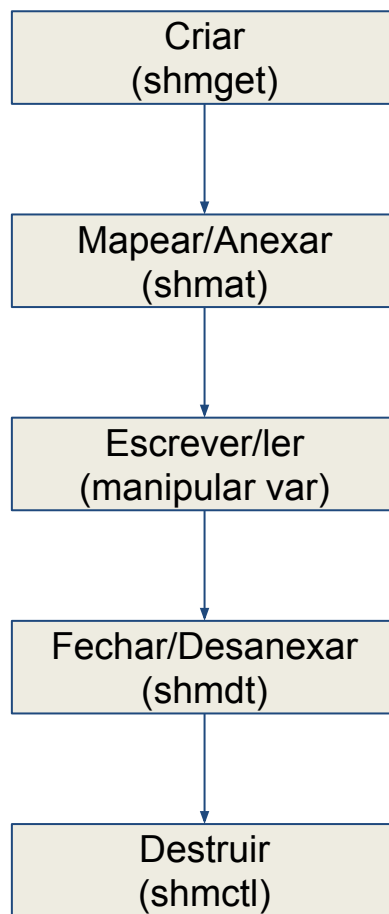
# Memória Compartilhada

- Não é possível antecipar em qual endereço virtual o processo será alocado, desta forma o uso de memória compartilhada não permite trabalhar com estruturas do tipo listas encadeadas, por exemplo.

# Memória Compartilhada



# Memória Compartilhada



# Memória Compartilhada

- A criação de uma área de memória compartilhada é realizada através da função *shmget()*.
  - *int shmget(key\_t key, size\_t size, int shmflg)*
    - **key** um identificador para área de memória compartilhada. Caso o identificador já exista, ou seja, existe uma área de memória compartilhada com esse identificador, o processo obtém acesso a área já existente.
    - **size** tamanho em bytes da área de memória compartilhada.
    - **shmflg** permissão de acesso e criação. Ex: `IPC_CREAT | 0777` cria nova área ou recupera, `0777` só recupera área existente
    - Retorno: id da memória se sucesso; -1 caso contrário



# Memória Compartilhada

- O mapeamento de uma área de memória compartilhada é realizada através da função *shmat()*.
  - Sintaxe: *int \*shmat(int shmid, const void\* shmaddr, int shmflg)*
  - Onde:
    - **shmid** identificador da área de memória compartilhada.
    - **shmaddr** endereço ao qual se quer vincular a área compartilhada. Preferencialmente se NULL para que o kernel escolha o endereço.
    - **shmflg** atribui uma permissão a área compartilhada. Geralmente usa-se 0
  - O retorno da função é o endereço da área mapeada no espaço de endereçamento do processo que solicitou acesso, também conhecido como **visão da área de memória compartilhada**.

# Memória Compartilhada

- As operações de leitura escrita em uma área de memória compartilhada são as operações usuais de manipulação de variáveis.
- Exemplos de operações sobre memória:
  - Atribuição de dados
  - Incremento
  - Decremento
  - ...

# Memória Compartilhada

- O fechamento da visão da área de memória compartilhada é realizado com a função *shmdt()*.
  - Sintaxe:
    - *int shmdt(const void\* shmaddr)*
  - Onde:
    - **shmaddr** endereço ao qual está vinculado a área compartilhada.

# Memória Compartilhada

- Para excluir uma área de memória compartilhada utiliza-se a função *shmctl()*.
  - Sintaxe: *int shmctl(int shmid, int cmd, struct shmid\_ds \*buf)*
  - Onde:
    - **shmid** identificador da área de memória compartilhada.
    - **cmd** permite especificar um conjunto de operações através de constantes predefinidas, que podem ser: `IPC_RMID` para eliminar a área de memória ou `IPC_STAT` para consultar informações sobre o processo que mapeou a área de memória por último.
    - **\*buf** indica o endereço de uma estrutura de dados que armazenará as informações do comando `IPC-STAT`, caso este tenha sido passado no segundo parâmetro.

# Memória Compartilhada

- Exemplo (Produtor/Consumidor)
  - Código do produtor (1/2)

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <sys/types.h>
4.  #include <sys/ipc.h>
5.  #include <sys/shm.h>
6.  #define CHAVE 10
7.  int main()
8.  {
9.      int mem_id, *ptr_mem, i;
10.     mem_id = shmget(CHAVE, sizeof(int)*256,
11.         0777|IPC_CREAT);
12.     if (mem_id < 0) {
13.         printf("Erro ao criar area de memoria  
compartilhada...\n");
14.         exit(0);
15.     }
```

# Memória Compartilhada

- Exemplo (Produtor/Consumidor)
  - Código do produtor (2/2)

```
15. ptr_mem = (int*)shmat(mem_id, 0, 0);
16. if (ptr_mem == NULL) {
17.     printf("Erro de mapeamento de memoria...\n");
18.     exit(0);
19. }

20. for (i = 0; i < 256; i++) {
21.     *(ptr_mem++) = i;
22. }
23. shmdt((void*)ptr_mem);
24. return 0;
25. }
```

# Memória Compartilhada

- Exemplo (Produtor/Consumidor)
  - Código do consumidor (1/2)

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <sys/types.h>
4.  #include <sys/ipc.h>
5.  #include <sys/shm.h>

6.  #define CHAVE 10

7.  int main()
8.  {
9.      int mem_id, *ptr_mem, i;

10.     mem_id = shmget(CHAVE, sizeof(int)*256,
11.                     0777|IPC_CREAT);
12.     if (mem_id < 0) {
13.         printf("Erro ao criar area de memoria
14.             compartilhada...\n");
15.         exit(0);
16.     }
```

# Memória Compartilhada

- Exemplo (Produtor/Consumidor)

- Código do consumidor (2/2)

```
16. ptr_mem = (int*)shmat(mem_id, 0, 0);
17. if (ptr_mem == NULL) {
18.     printf("Erro de mapeamento de memoria...\n");
19.     exit(0);
20. }

21. for (i = 0; i < 256; i++) {
22.     printf("Dados da memoria compartilhado...: %d\n", *(ptr_mem++));
23. }
24. shmdt((void*)ptr_mem);
25. shmctl(mem_id, 0, IPC_RMID);
26.
27. return 0;
28. }
```



# Memória Compartilhada

O programador é responsável por sincronizar os processos que usam memória compartilhada

# PIPEs

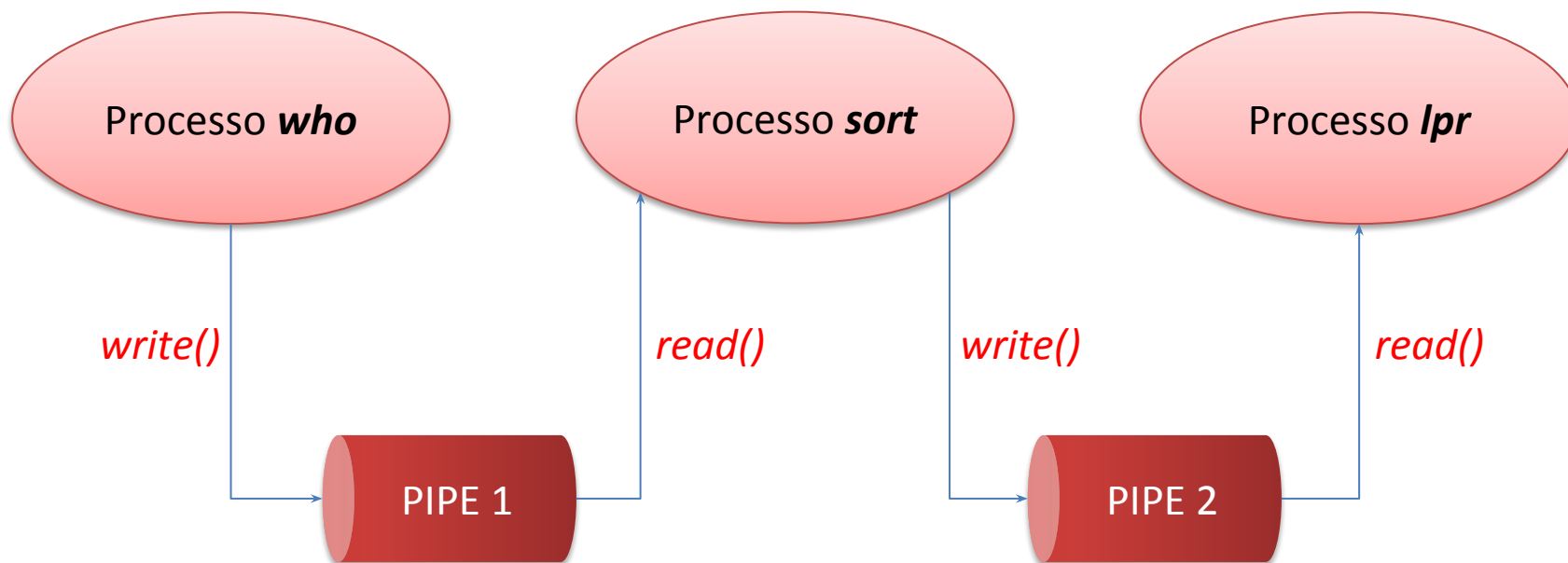
- Um PIPE pode ser enquadrado como uma versão limitada das classes de mecanismos das conexões virtuais.
- O PIPE liga dois processos o permite o fluxo de informações de forma **unidirecional**.
- São adequados para a implementação de mecanismos mestre-escravo (**um-para-um**).
- Os PIPEs podem ser **anônimos** ou **nomeados**
- Um PIPE anônimo só pode ser usado entre processos que possuam relação hierárquica (pai e filho).
- Cada PIPE é representado por um arquivo especial no sistema de arquivos.

# PIPEs

- O console utiliza PIPE para a comunicação entre processos.
- Exemplo 1:
  - *who* / *sort* / *lpr*
  - Imprimir (lpr) a listagem dos usuário logados na máquina (who) em ordem alfabética (sort).
  - A | significa PIPE para o interpretador de comandos.

# PIPEs

- Exemplo 1:
  - *who* / *sort* / *lpr*



# PIPEs

- Um PIPE em Linux é criado a partir da chamada de sistema *pipe()*.
  - Sintaxe:
    - *int pipe(int \*filedes)*
  - Onde:
    - O parâmetro *\*filedes* representa um vetor que irá guardar os descritores do PIPE criado.
    - O descritor *filedes[0]* é utilizado para **leitura** de dados e o *filedes[1]* é utilizado para **escrita** de dados.

# PIPEs

- Para ler dados de um PIPE é utilizada função *read()*.
  - Sintaxe:
    - *int read(int descritor, void\* buffer, size\_t size)*
  - Onde:
    - **descritor** representa o identificador do PIPE para leitura de dados (**filedes[0]**).
    - **buffer** é a variável onde os dados lidos serão armazenados.
    - **size** é o tamanho máximo em bytes de buffer.
  - O processo consumidor fica bloqueado somente se o PIPE estiver vazio, caso contrário ele lê o PIPE e volta a executar.

# PIPEs

- O envio de dados através de um PIPE é realizado a partir da função *write()*.
  - Sintaxe:
    - *int write(int descritor, const void\* buffer, size\_t size)*
  - Onde:
    - **descritor** representa o identificador do PIPE para escrita de dados (**filedes[1]**).
    - **buffer** é a variável onde se encontram os dados a serem transmitidos.
    - **size** representa a quantidade em bytes de dados presentes no buffer.
  - O processo produtor fica bloqueado somente se o PIPE estiver cheio, caso contrário ele escreve no PIPE e volta a executar.

# PIPEs

- Ao concluir a utilização do PIPE é necessário fechá-lo com a função *close()*.
  - Sintaxe:
    - *int close(int descritor)*
  - Onde:
    - **descritor** representa o identificador do PIPE para leitura **ou** escrita de dados (`filedes[0]` **ou** `filedes[1]`).



# PIPEs: Exemplo

- Programa em C para troca de mensagens entre dois processos via PIPE.
- Os processos possuem uma hierarquia, ou seja, processo pai e processo filho utilizando o PIPE.

# PIPEs: Exemplo

```
1. #include <unistd.h>
2. #include <stdio.h>
3. #include <string.h>
4. #include <stdlib.h>
5. #define SIZE 100

6. int main()
7. {
8.     int fd[2], pid;
9.     char msg[SIZE];

10.    if (pipe(fd) < 0) {
11.        printf("Erro ao criar o pipe...\n");
12.        exit(0);
13.    }

14.    pid = fork();

15.    if (pid > 0) { // Processo pai
16.        close(fd[0]);
17.        write(fd[1], "Fala ai filho...", strlen("Fala ai
filho...")+1);
18.        close(fd[1]);
19.    }
20.    else if (pid == 0) { // Processo filho
21.        close(fd[1]);
22.        read(fd[0], msg, sizeof(msg));
23.        printf("Mensagem recebida do pai...: %s\n",
msg);
24.        close(fd[0]);
25.    }
26.    return 0;
27. }
```

# PIPEs

- PIPEs nomeados (*named pipes*)
  - Um PIPE anônimo está restrito a processos que possuem o mesmo ancestral, ou seja, que estão na mesma hierarquia, o que limita bastante o uso de PIPEs.
  - Uma outra forma de usar PIPE sem que o processo ou processos tenham o mesmo ancestral são os PIPEs nomeados.
  - Um PIPE nomeado possui um nome lógico que pode ser manipulado por qualquer processo.

# PIPEs

- PIPEs nomeados (*named pipes*)
  - Um PIPE nomeado pode ser criado a partir da função *mkfifo()*.
    - Sintaxe:
      - *int mkfifo(const char \*path\_name, mode\_t mode)*
    - Onde:
      - ***path\_name*** indica o nome, inclusive o caminho completo, do PIPE.
      - ***mode*** representa as permissões para o uso do PIPE.

# PIPEs

- PIPEs nomeados (*named pipes*)
  - A associação de um PIPE nomeado a um processo é realizado via função *open()*.
    - Sintaxe:
      - *int open(const char \*path\_name, int options, ...)*
    - Onde:
      - ***path\_name*** indica o nome, inclusive o caminho completo, do PIPE.
      - ***options*** representa as permissões para o uso do PIPE. As permissões para se usar um PIPE são:
        - » *O\_RDONLY* – somente leitura
        - » *O\_WRONLY* – somente escrita
      - ***retorno***: descritor do pipe aberto

# PIPEs

- PIPEs nomeados (*named pipes*)
  - Para ler dados de um PIPE é utilizada função *read()*.
    - Sintaxe:
      - *int read(int descritor, void\* buffer, size\_t size)*
    - Onde:
      - **descritor** representa o identificado do PIPE para **leitura** de dados.
      - **buffer** é a variável onde os dados lidos serão armazenados.
      - **size** é o tamanho máximo em bytes de buffer.
      - **retorno**: número de bytes lidos
  - O processo consumidor fica bloqueado **se produtor não abriu o PIPE ou o PIPE estiver vazio**; caso contrário o consumidor lê o PIPE e volta a executar.

# PIPEs

- PIPEs nomeados (*named pipes*)
  - O envio de dados através de um PIPE é realizado a partir da função *write()*.
    - Sintaxe:
      - *int write(int descritor, const void\* buffer, size\_t size)*
    - Onde:
      - **descritor** representa o identificado do PIPE para **escrita** de dados
      - **buffer** é a variável onde se encontram os dados a serem transmitidos.
      - **size** representa a quantidade em bytes de dados presentes no buffer.
      - **retorno**: número de bytes escritos
  - O processo produtor fica bloqueado **se o consumidor não abriu o PIPE** ou se o PIPE estiver cheio; caso contrário ele escreve no PIPE e volta a executar.

# PIPEs

- PIPEs nomeados (*named pipes*)
  - Ao concluir a utilização do PIPE é necessário fechá-lo com a função *close()*.
    - Sintaxe:
      - *int close(int descritor)*
    - Onde:
      - **descritor** representa o identificador do PIPE que será fechado.



# PIPEs

- PIPEs nomeados (*named pipes*)
  - É recomendável destruir o pipe quando ele não é mais necessário ou antes de criá-lo (evita conflito se já existir)
    - Sintaxe:
      - *int unlink(const char \* path\_name)*
    - Onde:
      - ***path\_name*** indica o nome, inclusive o caminho completo, do PIPE.

# Exemplo PIPE nomeado:writer.c



```
1  #include <fcntl.h>
2  #include <sys/stat.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5
6  int main()
7  {
8      int fd;
9      char * myfifo = "/tmp/myfifo";
10
11      /* remove the FIFO if it already exist*/
12      unlink(myfifo);
13
14      /* create the FIFO (named pipe) */
15      mkfifo(myfifo, 0666);
16
17      /* write "Hi" to the FIFO */
18      fd = open(myfifo, O_WRONLY);
19      write(fd, "Hi", sizeof("Hi"));
20      close(fd);
21
22      /* remove the FIFO */
23      unlink(myfifo);
24
25      return 0;
26  }
```

# Exemplo PIPE nomeado:reader.c



```
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <sys/stat.h>
4  #include <unistd.h>
5
6  #define MAX_BUF 1024
7
8  int main()
9  {
10     int fd;
11     char * myfifo = "/tmp/myfifo";
12     char buf[MAX_BUF];
13
14     /* open, read, and display the message from the FIFO */
15     fd = open(myfifo, O_RDONLY);
16     read(fd, buf, MAX_BUF);
17     printf("Received: %s\n", buf);
18     close(fd);
19
20     return 0;
21 }
```

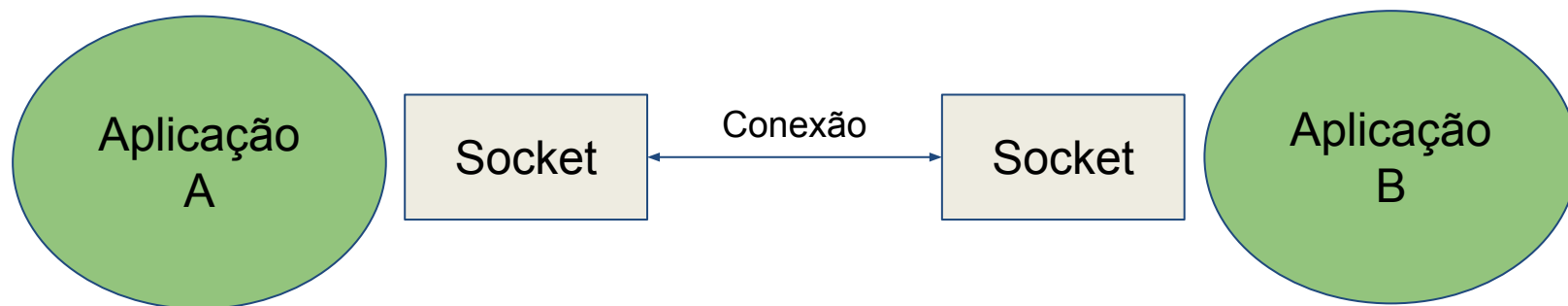
# PIPEs vs. Memória compartilhada (caso geral)



- Comunicação
  - PIPEs: um-para-um
  - Mem. compartilhada: muitos-para-muitos
- Responsabilidade pela sincronização
  - PIPEs: sistema operacional
  - Mem. compartilhada: programador

# Comunicação Cliente-Servidor

- Sockets são os pontos finais de uma conexão



# Comunicação Cliente-Servidor

- A comunicação baseada em sockets permite a adoção de um modelo de comunicação baseado em **conexões virtuais** ou em caixas de mensagens.
- A comunicação por socket tem dois objetivos:
  1. **Transparência**: a comunicação entre os processos deve ser programada de maneira uniforme, independente do contexto da comunicação.
  2. **Compatibilidade**: a comunicação por socket apresenta uma interface baseada nos descritores de arquivos, como acontece com os PIPEs.

# Comunicação Cliente-Servidor

- Os sockets são **bidirecionais** e foram concebidos para permitir a comunicação entre processos que executam em computadores diferentes.
- Os sockets são orientados a um **domínio de comunicação** que pode ser local ou remoto.
- Os domínios mais usados em sockets são: domínio local (**AF\_UNIX**) e domínio internet (**AF\_INET**).

# Comunicação Cliente-Servidor

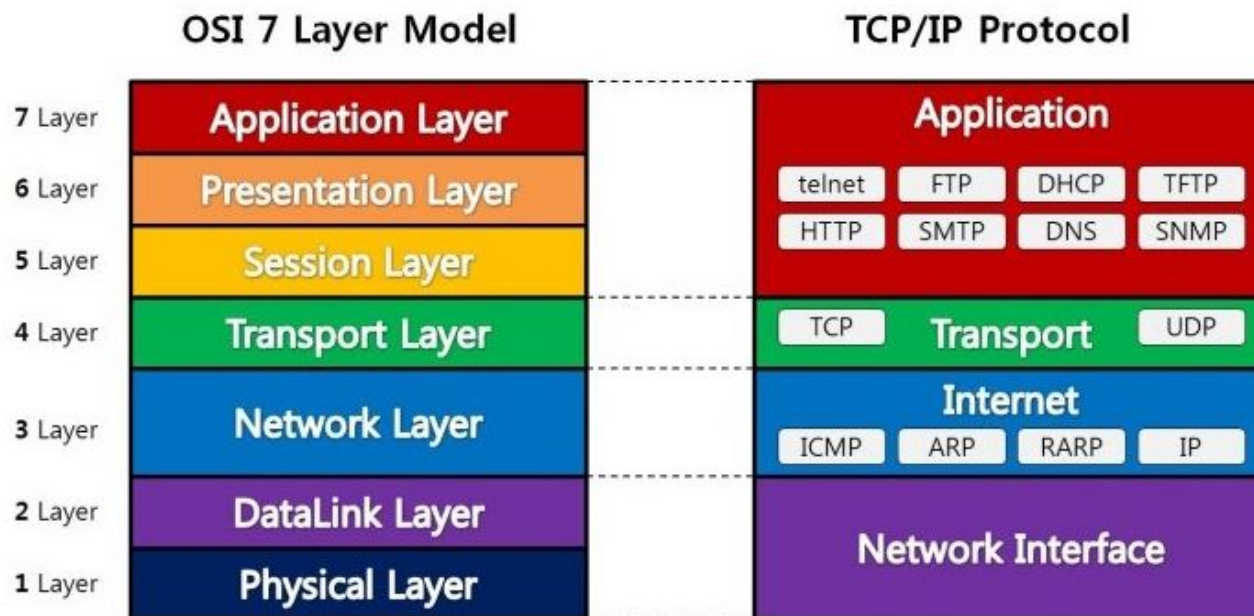


- Domínios do Socket: **AF\_UNIX (domínio local)** é **local** a uma máquina e é semelhante aos PIPEs nomeados, porém permitem bidirecionalidade dos dados. Neste domínio o socket é identificado como um **arquivo no sistema de arquivos local**.



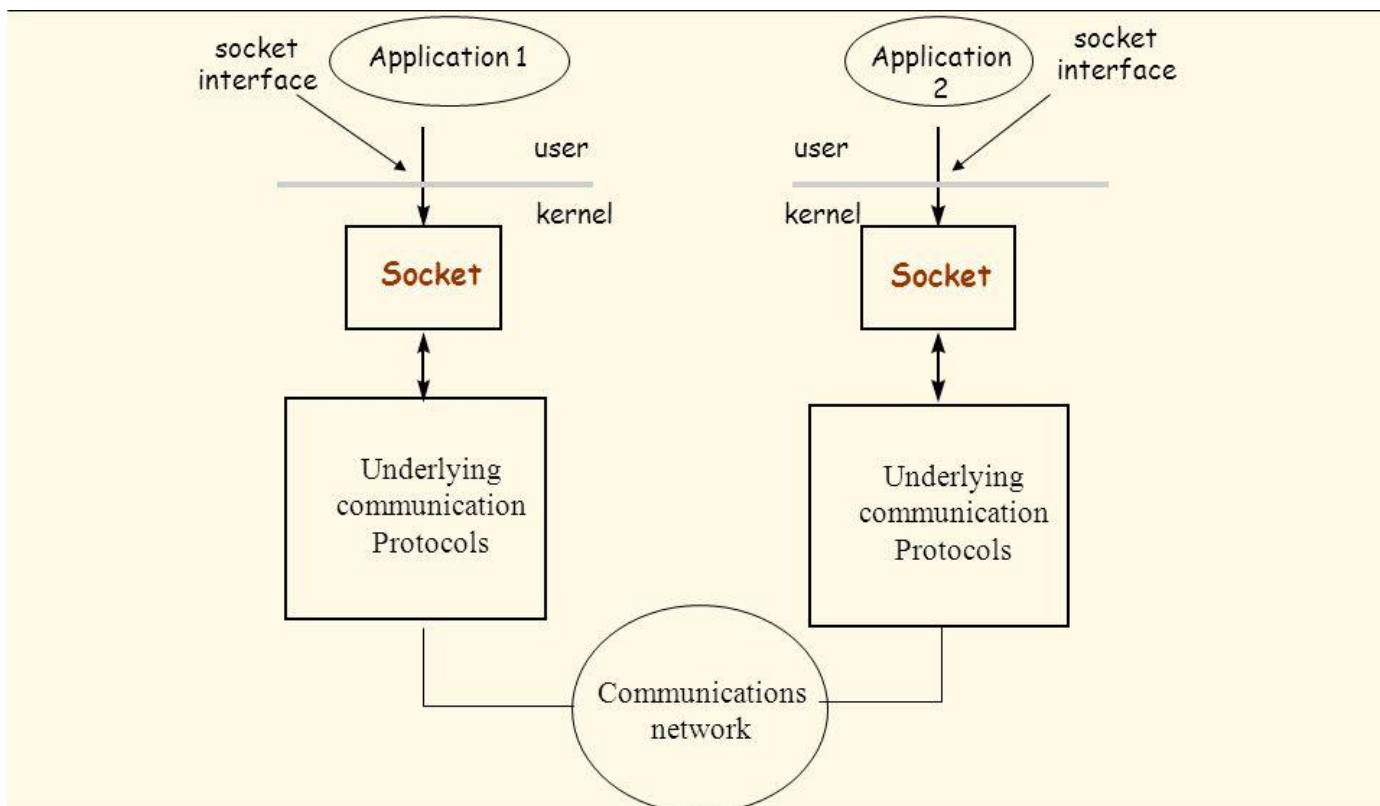
# Comunicação Cliente-Servidor

- Domínios do Socket: **AF\_INET (domínio internet)** visa comunicação entre processos executando em **máquinas diferentes**. O domínio internet baseia-se nos **protocolos de transporte UDP ou TCP**.



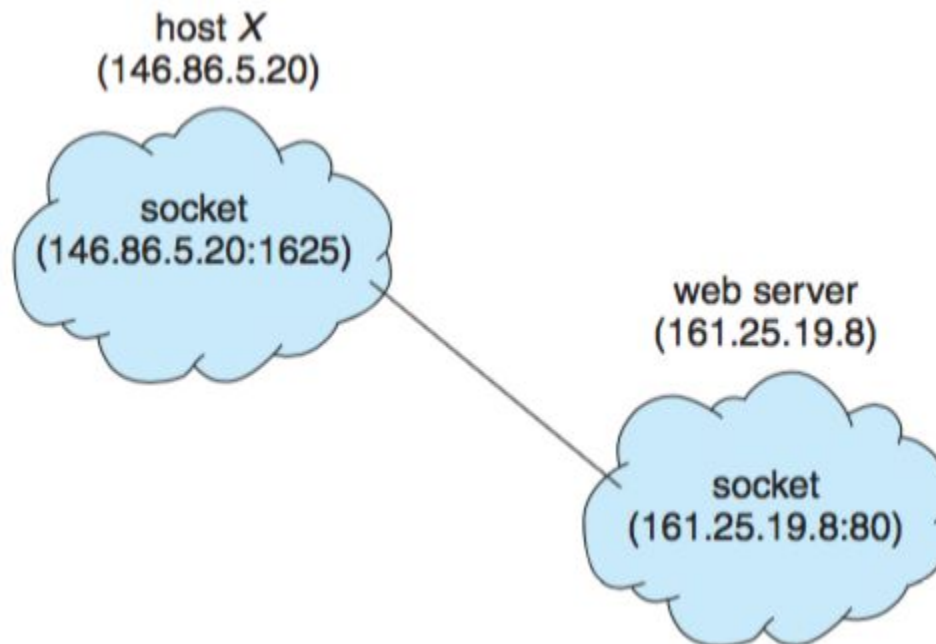
# Comunicação Cliente-Servidor

- Socket é uma interface de comunicação



# Comunicação Cliente-Servidor

- Domínio internet
  - Cada socket é identificado por *endereço:porta*
  - Par socket-socket é único na internet



# Comunicação Cliente-Servidor

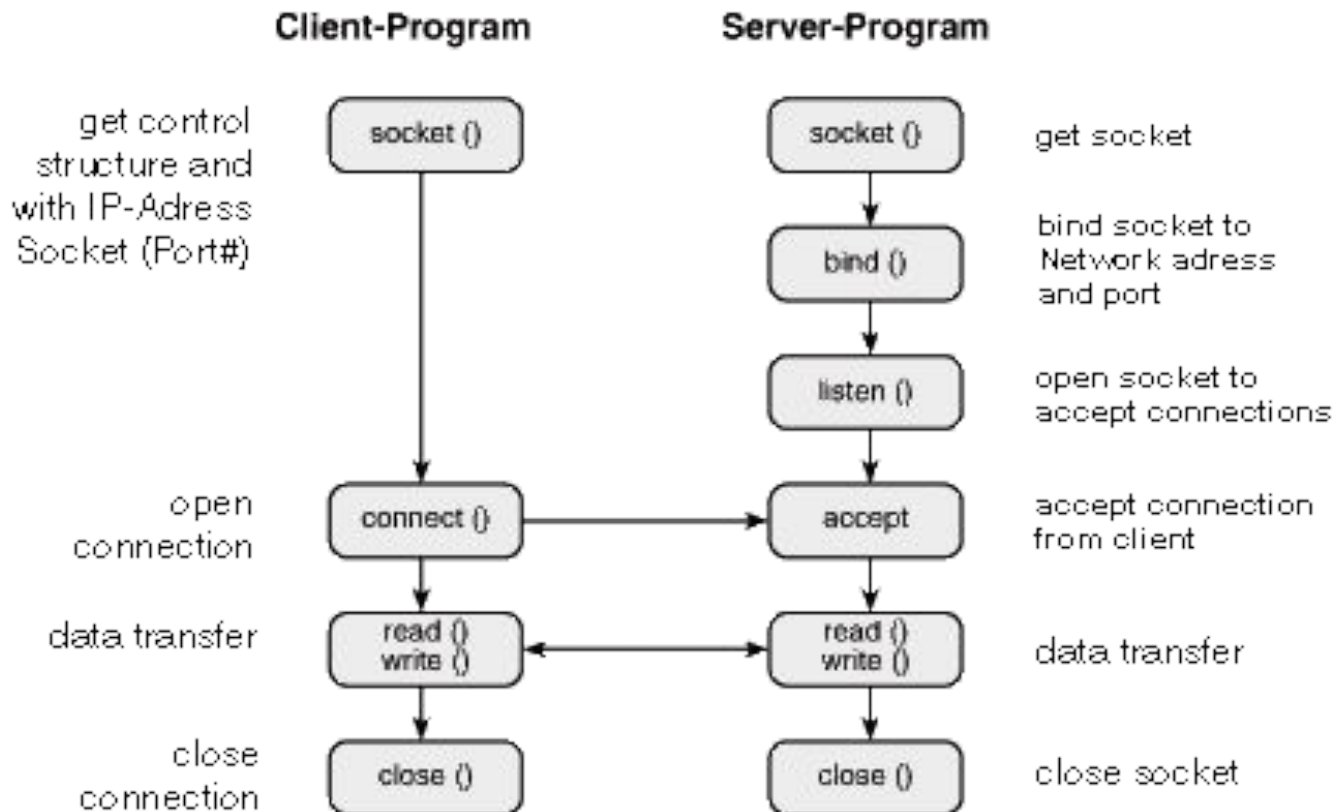
- Tipos de Socket
  - O tipo do socket estabelece a semântica de funcionamento da comunicação.
  - Cada tipo segue uma abordagem específica que dizem respeito a:
    - Garantia da sequencialidade (= ordem);
    - Mensagens duplicadas;
    - Confiabilidade na comunicação (= entrega garantida)
    - Preservação das fronteiras das mensagens.

# Comunicação Cliente-Servidor

- Tipos de Socket mais significativos
  - **Stream (TCP)**: comunicação bidirecional, fluxo de bytes sem estrutura, confiável, ordenada, com controle de duplicatas via conexão entre dois participantes. (Mais confiança, mais *overhead*).
  - **Datagram (UDP)**: comunicação bidirecional, sem confiabilidade, não ordenada, sem controle de duplicatas, sem conexão entre participantes. (Menos confiança, menos overhead)
- Outros tipos:
  - **Sequenced packet**
  - **Reliable datagram**
  - **Raw**:

# Comunicação Cliente-Servidor

- Comunicação baseada em *Socket Stream*



# Comunicação Cliente-Servidor

- Um socket é criado com a função `socket()`.
  - Sintaxe:
    - *`int socket(int domain, int type, int protocol)`*
  - Onde:
    - **domain** - refere-se ao domínio do socket (AF\_INET ou AF\_UNIX).
    - **type** – tipo do socket, ou seja, `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RDM`, `SOCK_RAW`, `SOCK_SEQPACKET` ou `SOCK_RAW`.
    - **protocol** – protocolo de comunicação, usualmente este parâmetro é zero (0).
  - O retorno da função é o identificador, descritor, do socket criado.

# Comunicação Cliente-Servidor

- A função `bind()` associa um nome ao socket.
  - Sintaxe:
    - *`int bind(int socket, const struct sockaddr *address, socklen_t address_len)`*
  - Onde:
    - **socket**- descritor do socket, criado a partir da função `socket()`.
    - **address** – estrutura de dados que contém os parâmetros do socket.
    - **address\_len** – tamanho em bytes da estrutura de dados que contém os parâmetros do socket.



# Comunicação Cliente-Servidor

- A função `listen()` habilita o servidor para receber conexões dos clientes.
  - Sintaxe:
    - *`int listen(int socket, int backlog)`*
  - Onde:
    - **socket**- descritor do socket, criado a partir da função `socket()`.
    - **backlog** – número máximo de pedidos pendentes.

# Comunicação Cliente-Servidor

- O estabelecimento da conexão entre o servidor e o cliente é realizado pela função `accept()`.
  - Sintaxe:
    - *`int accept(int socket_s, struct sockaddr *addr, int *addrlen)`*
  - Onde:
    - **socket\_s** descritor do socket do servidor, criado a partir da função `socket()`
    - **sockaddr** – endereço da estrutura de dados que conterá os dados da conexão com o cliente.
    - **addrlen** – endereço da variável que armazenará o tamanho da estrutura de dados com os dados do cliente.
  - O retorno da função é o descritor de um **novo** socket entre o servidor e **um cliente**.

# Comunicação Cliente-Servidor

- O cliente estabelece uma conexão com o servidor através da função `connect()`.
  - Sintaxe:
    - *`int connect(int s, struct sockaddr *name, int namelen)`*
  - Onde:
    - **socket\_s** descritor do socket, criado a partir da função `socket()`. Descritor do socket do servidor.
    - **sockaddr** – estrutura de dados que conterá os dados de endereço com quem se quer conectar.
    - **addrlen** – tamanho de **sockaddr**
  - O retorno da função é 0 se sucesso. Caso contrário -1

# Comunicação Cliente-Servidor

- Para ler e escrever no socket são usadas, respectivamente, as funções:
  - *int read(int descritor, void\* buffer, size\_t size)*
  - *int write(int descritor, const void\* buffer, size\_t size)*

# Comunicação Cliente-Servidor



- Exemplo 1: Servidor *socket domínio AF\_UNIX (1/4)*

```
1  #include <strings.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <sys/socket.h>
7  #include <sys/un.h>
8
9  int verifica_par(int num)
10 {
11     if (num % 2 == 0)
12         return 1;
13     return 0;
14 }
```

# Comunicação Cliente-Servidor

- Exemplo 1: Servidor *socket domínio AF\_UNIX (2/4)*

```
16 int main()
17 {
18     int sock_fd, sock_len, sock_novo, num;
19     socklen_t sock_novo_len;
20     struct sockaddr_un sock_ser, sock_cli;
21     char msg[100];
22
23     sock_fd = socket(AF_UNIX, SOCK_STREAM, 0);
24     if (sock_fd < 0) {
25         printf("Erro ao criar o socket...\n");
26         exit(0);
27     }
28
29     unlink("socket.unix.teste");
30
31     bzero((char*)&sock_ser, sizeof(sock_ser));
32     sock_ser.sun_family = AF_UNIX;
33     strcpy(sock_ser.sun_path, "socket.unix.teste2");
34     sock_len = strlen(sock_ser.sun_path) + sizeof(sock_ser.sun_family);
```

# Comunicação Cliente-Servidor



- Exemplo 1: Servidor *socket domínio AF\_UNIX (3/4)*

```
36  if (bind(sock_fd, (struct sockaddr*)&sock_ser, sock_len) < 0) {  
37      printf("Erro ao associar nome ao socket...\n");  
38      exit(0);  
39  }  
40  
41  listen(sock_fd, 5);
```



# Comunicação Cliente-Servidor

- Exemplo 1: Servidor *socket domínio AF\_UNIX (4/4)*

```
43  for (;;) {
44      sock_novo_len = sizeof(sock_cli);
45      sock_novo = accept(sock_fd, (struct sockaddr*)&sock_cli, &sock_novo_len);
46
47      if (sock_novo < 0) {
48          printf("Erro ao tentar estabelecer conexao com o cliente...\n");
49          exit(0);
50      }
51
52      if (fork() == 0) {
53          if (read(sock_novo, msg, sizeof(msg)) < 0) {
54              printf("Erro de leitura do socket\n");
55              exit(0);
56          }
57
58          num = atoi(msg);
59          printf("Numero recebido...: %d\n", num);
60
61          if (verifica_par(num)) {
62              write(sock_novo, "PAR", strlen("PAR")+1);
63          } else {
64              write(sock_novo, "IMPAR", strlen("IMPAR")+1);
65          }
66          close(sock_novo);
67          exit(0);
68      }
69  }
```



# Comunicação Cliente-Servidor

- Exemplo 1: Cliente *socket domínio AF\_UNIX (1/3)*

```
1  #include <strings.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <sys/socket.h>
7  #include <sys/un.h>
8
9  int main(int argc, char** argv)
10 {
11     int sock_fd, sock_len, num;
12     struct sockaddr_un sock_cli;
13     char msg[100];
14
15     if (argc != 2) {
16         printf("Uso: %s numero\n", argv[0]);
17         exit(0);
18     }
```

# Comunicação Cliente-Servidor

- Exemplo 1: Cliente *socket domínio AF\_UNIX (2/3)*

```
20  sock_fd = socket(AF_UNIX, SOCK_STREAM, 0);
21  if (sock_fd < 0) {
22      printf("Erro ao criar o socket...\n");
23      exit(0);
24  }
25
26  bzero((char*)&sock_cli, sizeof(sock_cli));
27  sock_cli.sun_family = AF_UNIX;
28  strcpy(sock_cli.sun_path, "socket.unix.teste2");
29  sock_len = strlen(sock_cli.sun_path) + sizeof(sock_cli.sun_family);
30
31  if (connect(sock_fd, (struct sockaddr*)&sock_cli, sock_len) < 0) {
32      printf("Erro ao tentar conectar com o servidor...\n");
33      exit(0);
34  }
```

# Comunicação Cliente-Servidor

- Exemplo 1: Cliente *socket domínio AF\_UNIX (3/3)*

```
36     write(sock_fd, argv[1], strlen(argv[1]));
37     if (read(sock_fd, msg, sizeof(msg)) < 0) {
38         printf("Erro na leitura da resposta do servidor...\n");
39         exit(0);
40     }
41     printf("O numero %s eh %s\n", argv[1], msg);
42
43     close(sock_fd);
44
45     return 0;
46 }
```

# Comunicação Cliente-Servidor

- Exemplo 2: Servidor *socket domínio AF\_INET (1/2)*

```
1  #include <sys/socket.h>
2  #include <netinet/in.h>
3  #include <arpa/inet.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7  #include <errno.h>
8  #include <string.h>
9  #include <sys/types.h>
10 #include <time.h>
11
12 int main(int argc, char *argv[])
13 {
14     ...
43 }
```

# Comunicação Cliente-Servidor

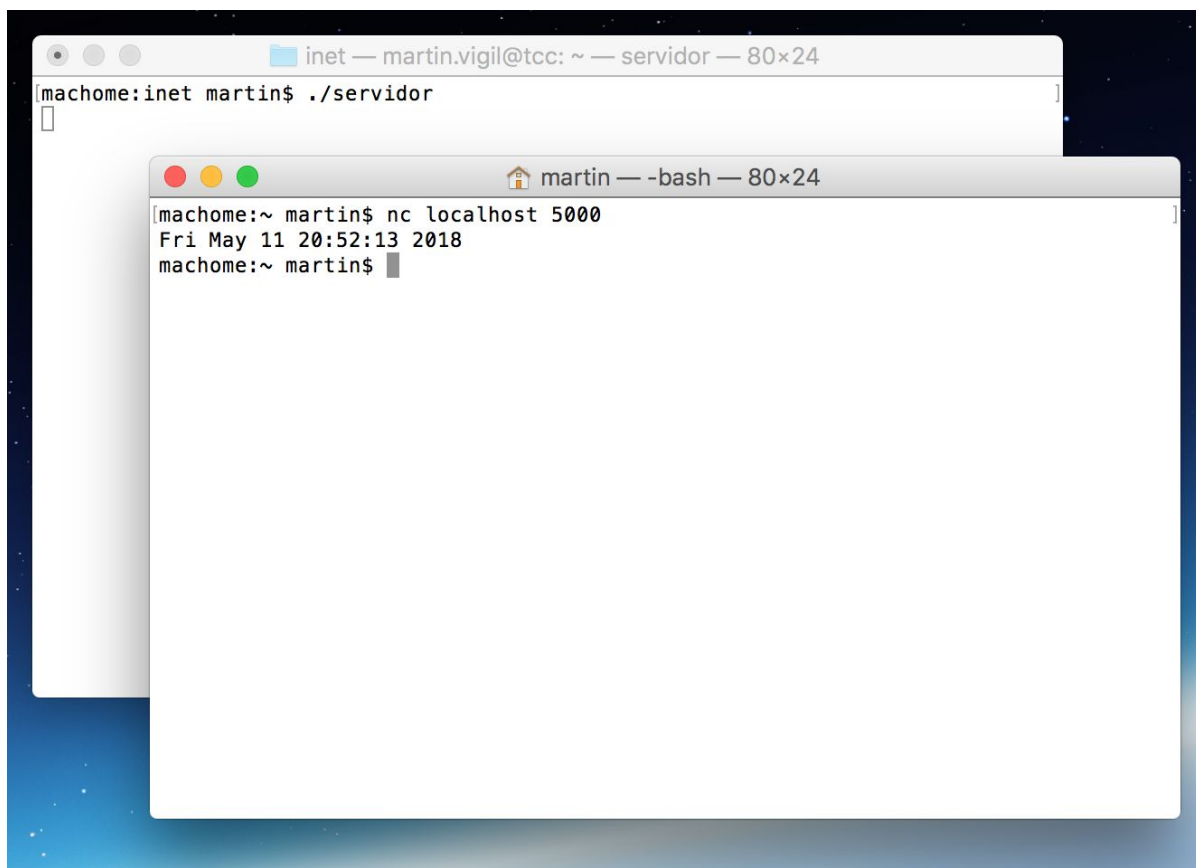
- Exemplo 2: Servidor *socket domínio AF\_INET (2/2)*

```
12 int main(int argc, char *argv[])
13 {
14     int listenfd = 0, connfd = 0;
15     struct sockaddr_in serv_addr;
16
17     char sendBuff[1025];
18     time_t ticks;
19
20     listenfd = socket(AF_INET, SOCK_STREAM, 0);
21     memset(&serv_addr, '0', sizeof(serv_addr));
22     memset(sendBuff, '0', sizeof(sendBuff));
23
24     serv_addr.sin_family = AF_INET;
25     serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
26     serv_addr.sin_port = htons(5000);
27
28     bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
29
30     listen(listenfd, 10);
31
32     while(1)
33     {
34         connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
35
36         ticks = time(NULL);
37         snprintf(sendBuff, sizeof(sendBuff), "%.24s\r\n", ctime(&ticks));
38         write(connfd, sendBuff, strlen(sendBuff));
39
40         close(connfd);
41         sleep(1);
42     }
43 }
```



# Comunicação Cliente-Servidor

- Exemplo 2: testando servidor via netcat

A screenshot of two overlapping terminal windows on a macOS desktop. The background window, titled 'inet — martin.vigil@tcc: ~ — servidor — 80x24', shows a shell prompt 'machome:inet martin\$' followed by the command './servidor' and a cursor. The foreground window, titled 'martin — -bash — 80x24', shows a shell prompt 'machome:~ martin\$' followed by the command 'nc localhost 5000'. The next line shows the connection time 'Fri May 11 20:52:13 2018', and the final line shows the prompt 'machome:~ martin\$' with a cursor.

```
inet — martin.vigil@tcc: ~ — servidor — 80x24
machome:inet martin$ ./servidor
└─
```

```
martin — -bash — 80x24
machome:~ martin$ nc localhost 5000
Fri May 11 20:52:13 2018
machome:~ martin$
```

# Comunicação Cliente-Servidor

- Comunicação baseada em *Socket Datagram*

