

# Engenharia de Computação



## Arquitetura de Sistemas Operacionais

### Comunicação entre Processos

**Prof. Anderson Luiz Fernandes Perez**  
**Prof. Martín Vigil**

# 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

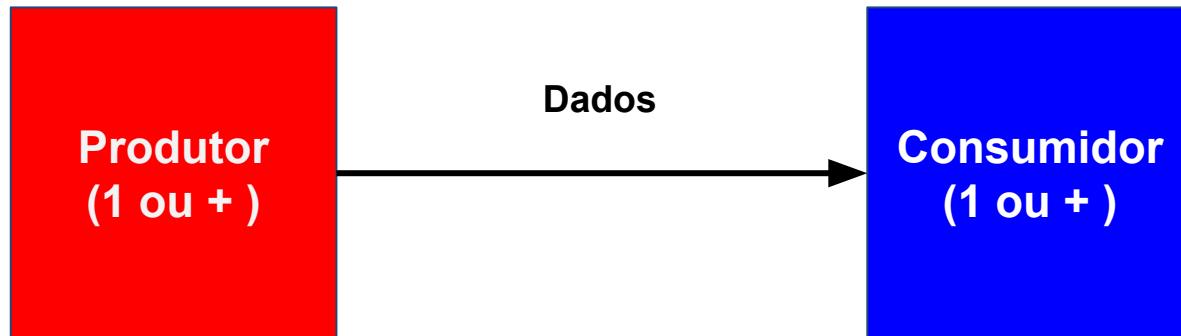
- A cooperação entre processos oferece as seguintes facilidades:
  - **Compartilhamento de Informações:** acesso concorrente a dados, por exemplo, mais de um processo acessando um arquivo em disco.
  - **Velocidade do Processamento:** a divisão de uma tarefa em subtarefas permite minimizar o tempo de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

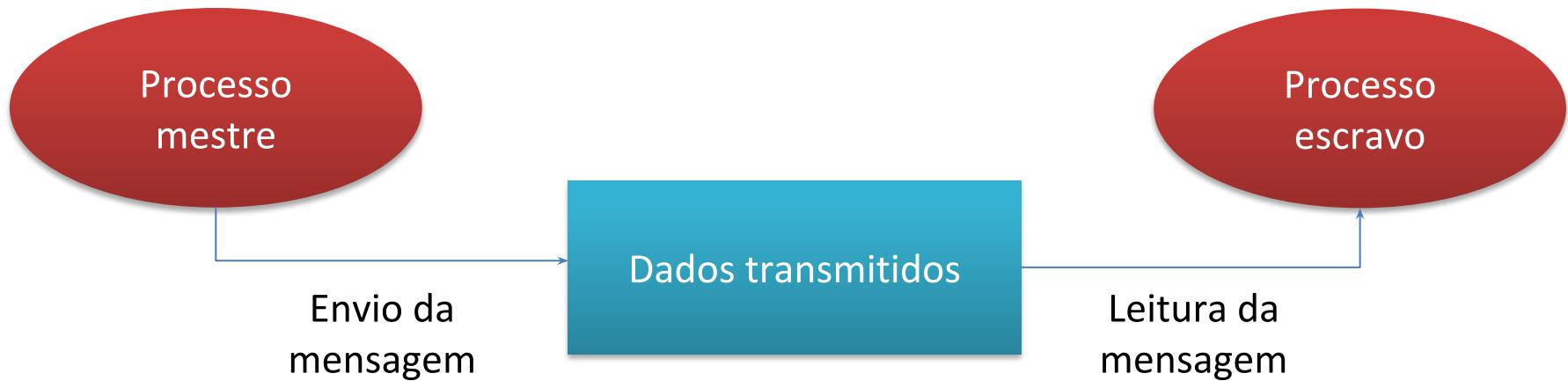


- Modelo de Interação Um para Um (mestre-escravo)
  - Baseia-se na **associação estrita entre dois processos**, sendo estabelecida uma ligação entre ambos.
  - O processo **escravo** (leitor) tem a sua atividade totalmente controlada pelo processo mestre (**produtor**).
  - O canal de comunicação é fixo 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

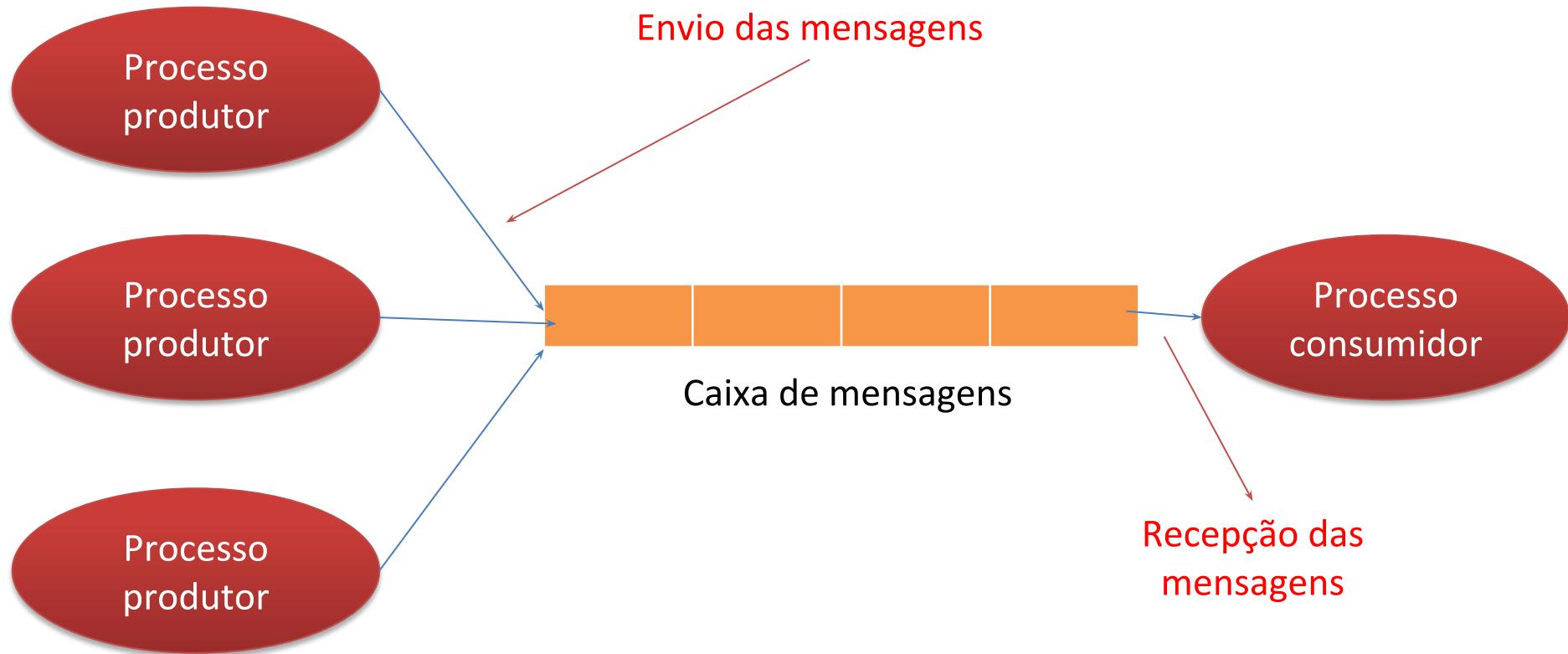


- Modelo de Interação Muitos para Um
  - O modelo **Muitos para Um**, também conhecido como **correio**, baseia-se na possibilidade de transferência de dados em modo **assíncrono** sob a forma de mensagem.
  - As **mensagens são enviadas individualmente** por um conjunto de processos produtores a um 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

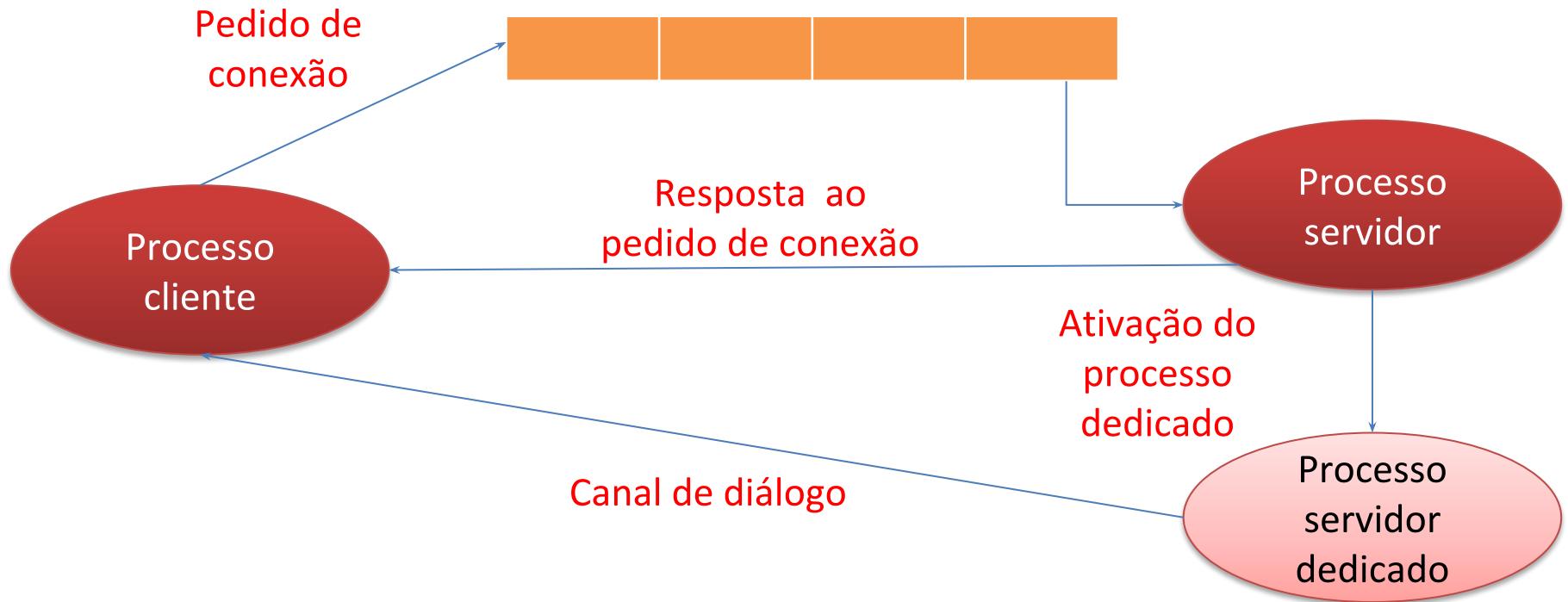


- Modelo de Interação Um para Um de Vários
  - Também conhecido como **Diálogo**, é um **modelo híbrido entre os modelos Um para Um e Muitos para Um**.
  - Neste modelo é estabelecido **um canal fixo entre dois processos**, criado de forma dinâmica.
  - Um processo, normalmente um **cliente**, **deve 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

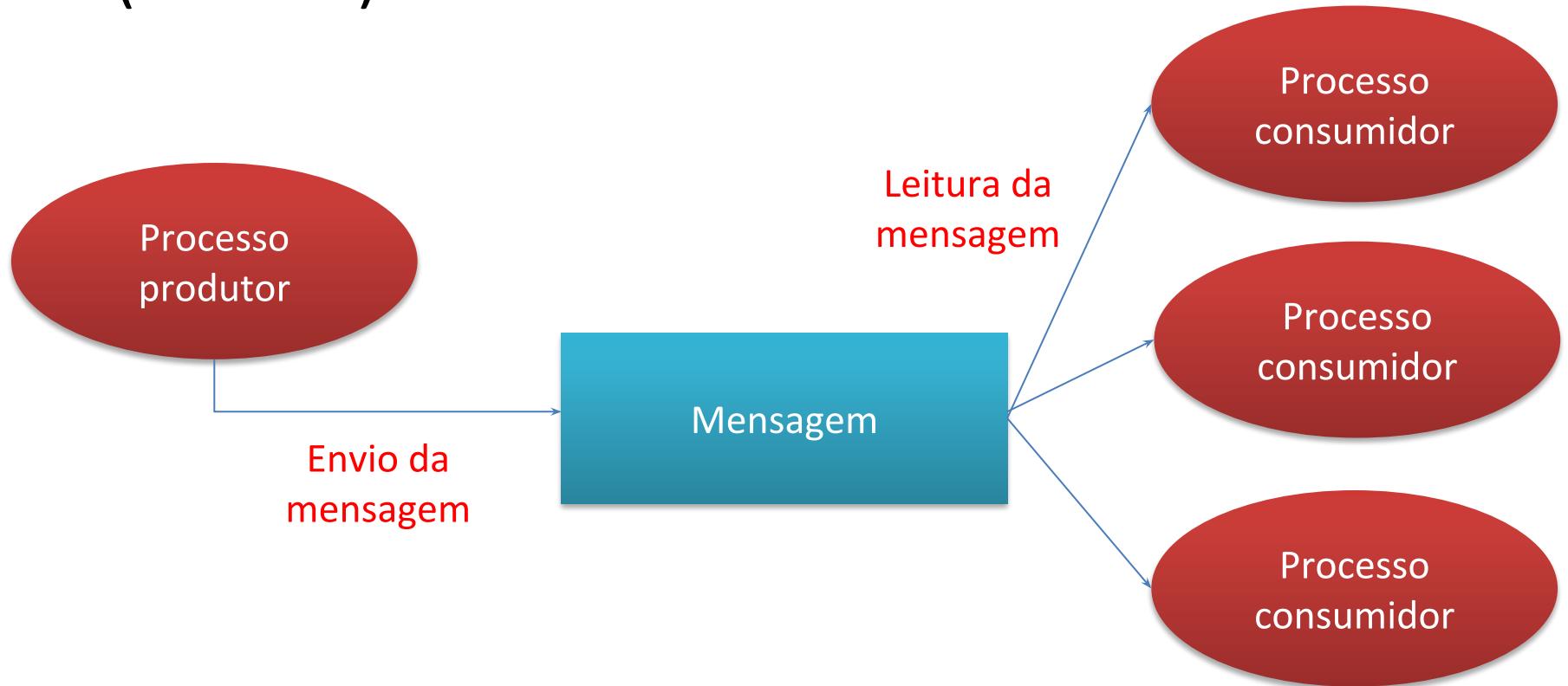


- Modelo de Interação Um para Muitos (difusão)
  - Neste modelo de interação um processo produtor envia uma mesma informação para vários processos consumidores ou para um grupo de consumidores previamente identificado.
  - Este tipo de comunicação é muito utilizado para notificações entre processos.
  - 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 modulo 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 quanto consumir informações.

# Modelos de Interação entre Produtores e Consumidores



- Resumo

Modelo de interação	Situação no mundo real
Um-para-um	Comunicação com par de walkie-talkies.
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

# 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



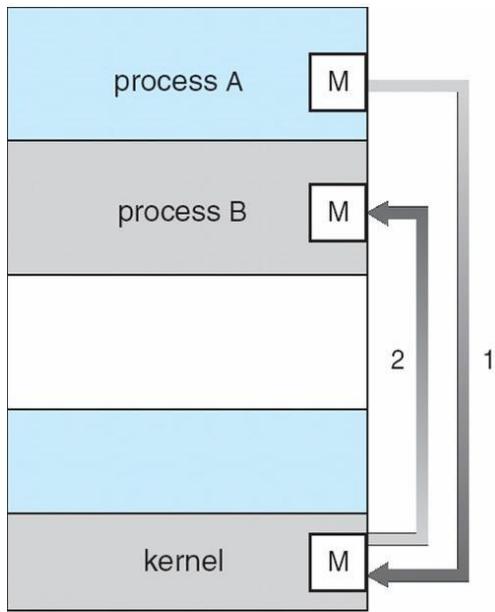
- A comunicação entre processos não somente determina um meio de troca de informações entre processos, mas também um mecanismo para sincronizar as ações dos processos comunicantes.
- A **semântica na comunicação de processos determina o comportamento do processo ao receber e enviar um mensagem**, e podem ser:
  - **Síncrona**: o produtor fica bloqueado até que o consumidor receba a mensagem e acesse o seu conteúdo.
  - **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.
  - **Cliente-Servidor**: o processo produtor (cliente) fica bloqueado até que o consumidor (servidor) tenha recebido a mensagem e enviado uma mensagem de resposta.

# Comunicação no Modelo Computacional

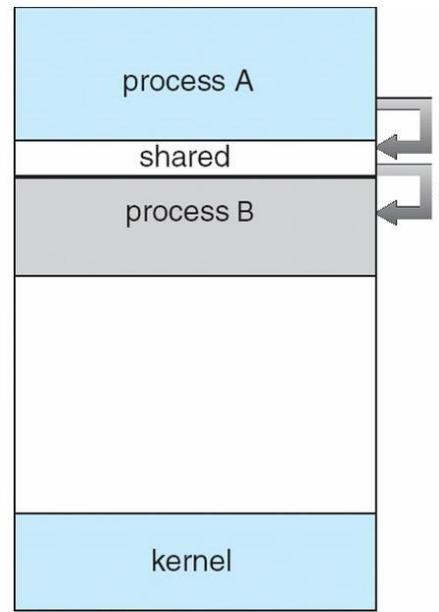
- Um canal de comunicação pode ser implementado de duas formas distintas:

## Via kernel

Evita conflitos  
Fácil implementar  
Sist. distribuídos



(a)



(b)

## Via memória compartilhada

Transmissão mais rápida

# Comunicação no Modelo Computacional



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

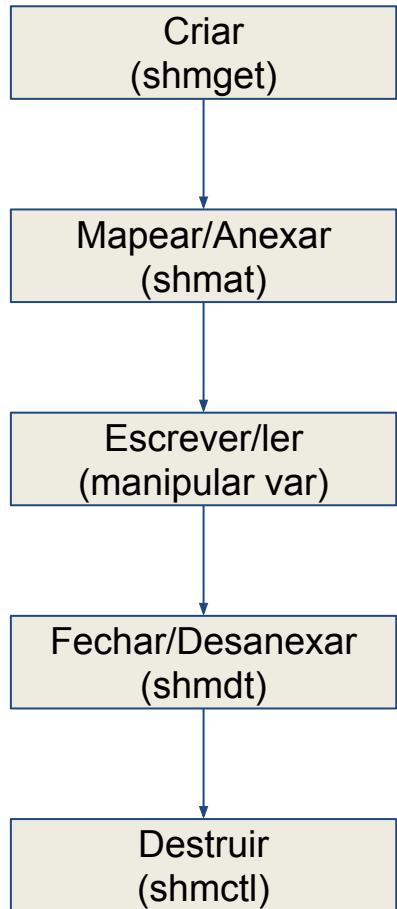
# Memória Compartilhada

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

- A criação de uma área de memória compartilhada é realizada através da função *shmget()*.
  - Sintaxe:
    - *int shmget(key\_t key, size\_t size, int shmflg)*
  - Onde:
    - **key** identificador da 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** atribui uma permissão a área compartilhada.

# 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. Geralmente usar-se NULL para que o kernel escolha o endereço.
    - **shmflg** atribui uma permissão a área compartilhada.
  - 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.
8. int main()
9. {
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.     }
17. }
```

# Memória Compartilhada

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

```
15. ptr_mem = (int*)shmat(mem_id, (char*)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. return 0;
24. }
```

# 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.
8. int main()
9. {
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.     }
17. }
```

# Memória Compartilhada

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

```
16. ptr_mem = (int*)shmat(mem_id, (char*)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.
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 possuírem um nome.
- Um PIPE anônimo tem que ser usado entre processos que possuam relação hierárquica (pai e filho).
- Cada PIPE é representando por um arquivo especial no sistema de arquivos.

# 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 ponteiro para um vetor que irá guardar os descritos do PIPE criado.
    - O descritor *filedes[0]* é utilizado para **leitura** de dados e o *filedes[1]* é utilizado para **escrita** de dados.

# PIPEs

- O envio de dados através de um PIPE é realizado a partir da função *write()*.
  - Sintaxe:
    - *int write(int descriptor, const void\* buffer, size\_t size)*
  - Onde:
    - **descriptor** representa o identificado 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

- Para ler dados de um PIPE é utilizada função *read()*.
  - Sintaxe:
    - *int read(int descriptor, void\* buffer, size\_t size)*
  - Onde:
    - **descriptor** representa o identificado 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

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

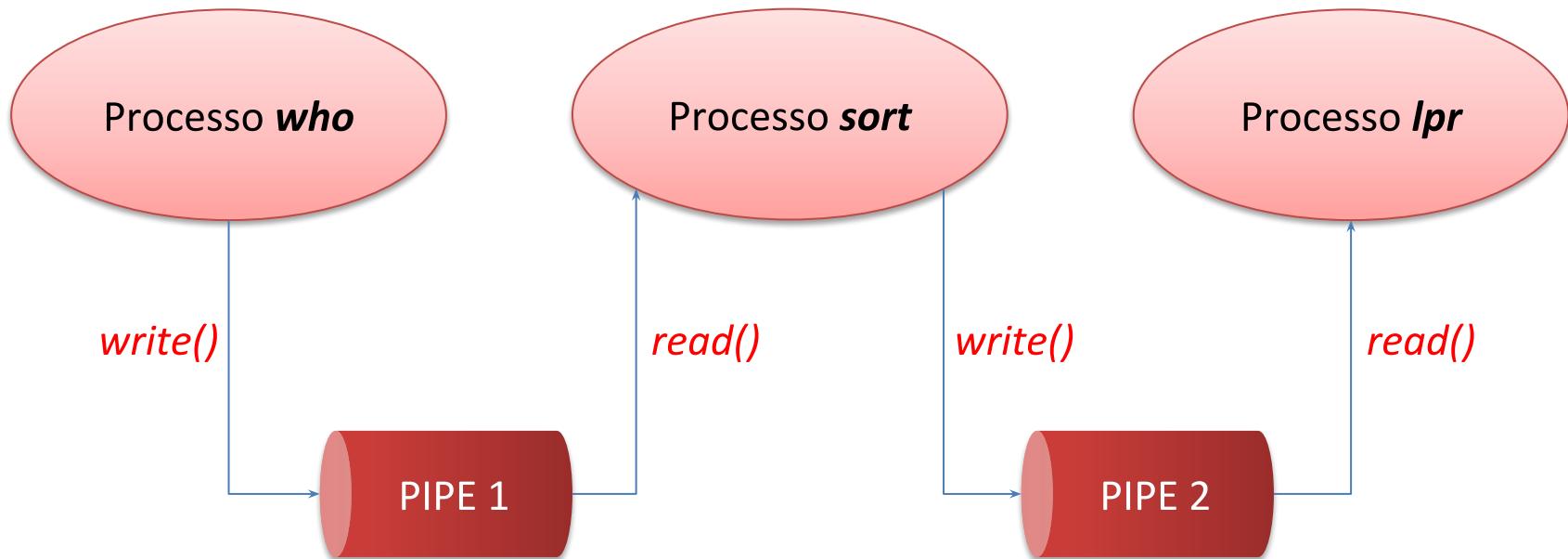
# PIPEs



- O interpretador de comandos (shell) 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



- Exemplo 2:
  - Programa em C para troca de mensagens entre dois processos via PIPE.
  - Cabe ressaltar que os processos possuem uma hierarquia, ou seja, processo pai e processo filho utilizando o PIPE.

# PIPEs



- Exemplo 2:

```
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
18.        filho...") + 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",
24.               msg);
25.    }
26.    return 0;
27. }
```

# PIPEs



- PIPEs nomeados (*named pipes*)
  - Um PIPE 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

# PIPEs

- PIPEs nomeados (*named pipes*)
  - O envio de dados através de um PIPE é realizado a partir da função *write()*.
    - Sintaxe:
      - *int write(int descriptor, const void\* buffer, size\_t size)*
    - Onde:
      - **descriptor** representa o identificado 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

- PIPEs nomeados (*named pipes*)
  - Para ler dados de um PIPE é utilizada função *read()*.
    - Sintaxe:
      - *int read(int descriptor, void\* buffer, size\_t size)*
    - Onde:
      - **descriptor** representa o identificado 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

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

# PIPEs

- PIPEs nomeados (*named pipes*)
  - Exemplo (*código do servidor*) (1/2):

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <sys/types.h>
5. #include <sys/stat.h>
6. #include <fcntl.h>

7. int main()
8. {
9.     int fd_server, fd_client;
10.    char msg[100];

11.    unlink("pipe.servidor");
12.    unlink("pipe.cliente");

13.    if (mkfifo("pipe.servidor", 0777) < 0) {
14.        printf("Erro ao criar pipe do
15. servidor...\n");
16.        exit(0);
17.    }

18.    if (mkfifo("pipe.cliente", 0777) < 0) {
19.        printf("Erro ao criar pipe do
20. cliente...\n");
21.        exit(0);
22.    }
```

# PIPEs

- PIPEs nomeados (*named pipes*)
  - Exemplo (*código do servidor* ) (2/2):

```
21. fd_server = open("pipe.servidor", O_RDONLY);      31. read(fd_server, msg, sizeof(msg));  
22. fd_client = open("pipe.cliente", O_WRONLY);       32. printf("Mensagem recebida do  
23. if (fd_server < 0) {                                cliente...: %s\n", msg);  
24.     printf("Erro ao abrir pipe do servidor...\n");  
25.     exit(0);  
26. }  
27. if (fd_client < 0) {  
28.     printf("Erro ao abrir pipe do cliente...\n");  
29.     exit(0);  
30. }
```

```
33. write(fd_client, "Ola cliente",  
        strlen("Ola cliente") + 1);  
34. return 0;
```

# PIPEs

- PIPEs nomeados (*named pipes*)
  - Exemplo (*código do cliente*) (1/2):

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <sys/types.h>
5. #include <sys/stat.h>
6. #include <fcntl.h>

7. int main()
8. {
9.     int fd_server, fd_client;
10.    char msg[100];
11.    fd_server = open("pipe.servidor",
12.                      O_WRONLY);
13.    fd_client = open("pipe.cliente", O_RDONLY);
14.    if (fd_server < 0) {
15.        printf("Erro ao abrir pipe do servidor...\n");
16.        exit(0);
17.    }
18.    if (fd_client < 0) {
19.        printf("Erro ao abrir pipe do cliente...\n");
20.        exit(0);
21.    }
```

# PIPEs

- PIPEs nomeados (*named pipes*)
  - Exemplo (*código do cliente*) (2/2):

```
21. write(fd_server, "Ola servidor", strlen("Ola servidor")+1);
22. read(fd_client, msg, sizeof(msg));
23. printf("Mensagem recebida do servidor...: %s\n", msg);
24.
25. close(fd_server);
26. close(fd_client);

27. return 0;
28. }
```

- Observação:

A função *unlink("nome do pipe")* serve para excluir um PIPE anteriormente criado. Recomenda-se que antes de se criar um PIPE o processo execute a função *unlink("nome do pipe")* para se certificar que não haverá um erro na execução do programa.

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

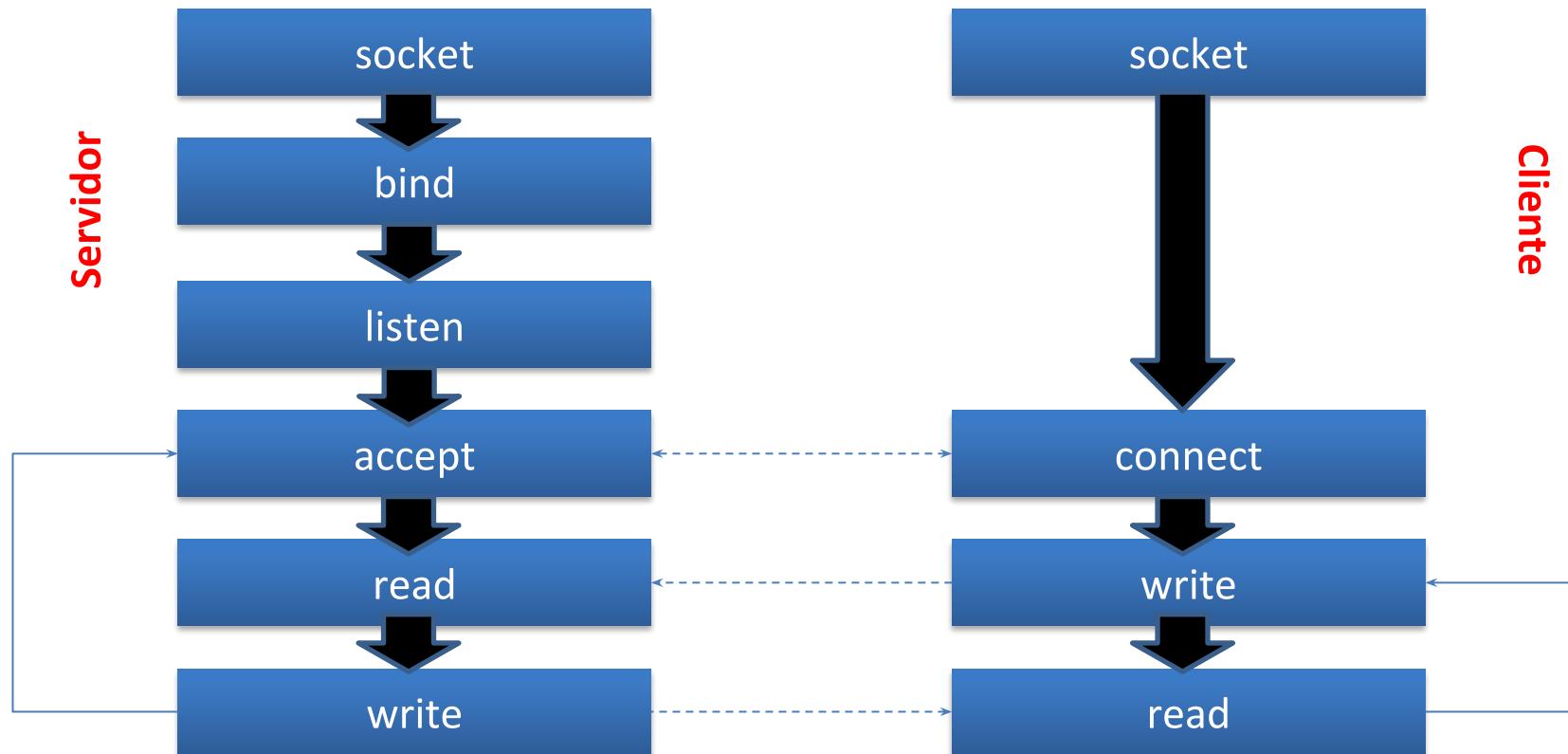
- Tipos de Socket
  - O tipo do socket estabelece a semântica de funcionamento da comunicação.
  - Cada tipo segue uma abordagem específica que dizem respeito a:
    - Garantia da sequencialidade;
    - Mensagens duplicadas;
    - Confiabilidade na comunicação;
    - Preservação das fronteiras das mensagens.

# Comunicação Cliente-Servidor

- Tipos de Socket
  - Os tipos de socket mais significativos são:
    - **Stream**: comunicação bidirecional, confiável e sequencial.
    - **Datagram**: comunicação bidirecional, sem confiabilidade, sem sequencialidade e sem eliminação de mensagens duplicadas.
    - **Sequenced packet**: comunicação bidirecional, confiável, sequencial e com preservação de fronteiras entre mensagens.
    - **Reliable datagram**: comunicação bidirecional, sem sequencialidade e com confiabilidade.
    - **Raw**: permite acesso direto aos protocolos de transporte das redes que suportam os sockets.

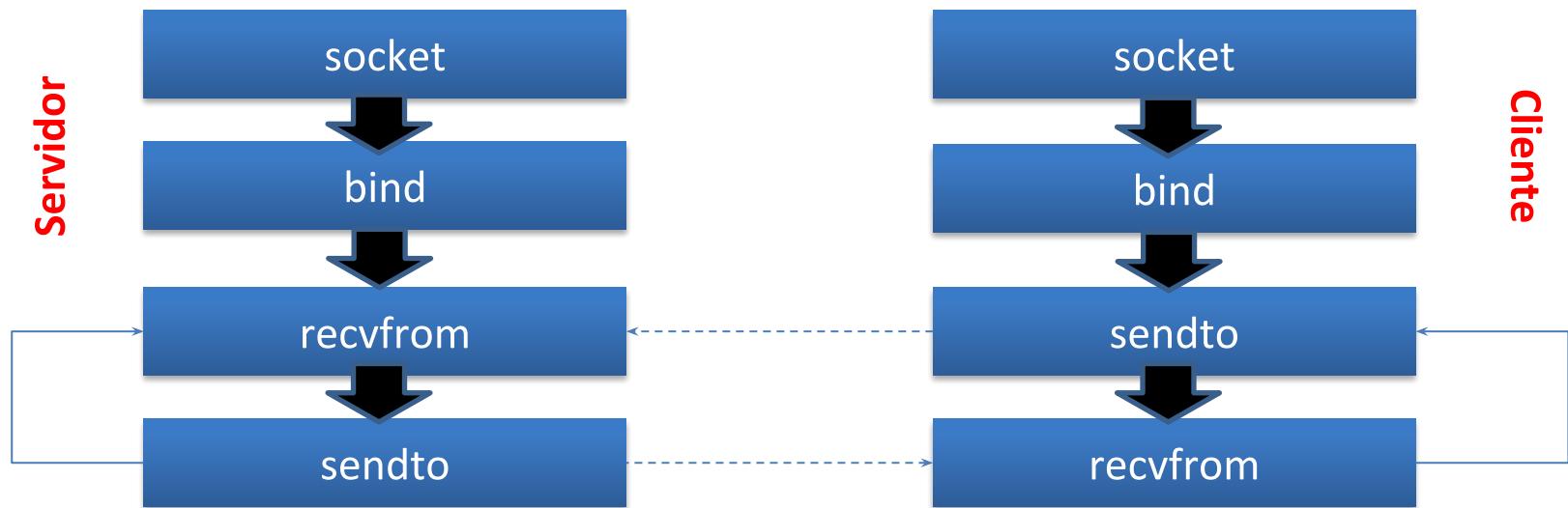
# Comunicação Cliente-Servidor

- Comunicação baseada em *Socket Stream*



# Comunicação Cliente-Servidor

- Comunicação baseada em *Socket Datagram*



# 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, descriptor, 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** – endereço da 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, criado a partir da função `socket()`. Descritor do socket do servidor.
    - **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 a identificação, descritor, do socket entre o servidor e o 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** – 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 a identificação, descritor, do socket entre o servidor e o cliente.

# Comunicação Cliente-Servidor

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

# Comunicação Cliente-Servidor

- Exemplo 1:
  - Servidor - *socket domínio AF\_UNIX (1/7)*

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <sys/types.h>
5. #include <sys/socket.h>
6. #include <sys/un.h>

7. int main()
8. {
9.     int sock_fd, sock_len, sock_novo, sock_novo_len, num;
10.    struct sockaddr_un sock_ser, sock_cli;
11.    char msg[100];
```

# Comunicação Cliente-Servidor

- Exemplo 1:
  - Servidor - *socket domínio AF\_UNIX (2/7)*

```
12.    sock_fd = socket(AF_UNIX, SOCK_STREAM, 0);
13.    if (sock_fd < 0) {
14.        printf("Erro ao criar o socket...\n");
15.        exit(0);
16.    }

17.    unlink("socket_unix.teste");

18.    bzero((char*)&sock_ser, sizeof(sock_ser));
19.    sock_ser.sun_family = AF_UNIX;
20.    strcpy(sock_ser.sun_path, "socket_unix.teste");
21.    sock_len = strlen(sock_ser.sun_path) + sizeof(sock_ser.sun_family);
```

# Comunicação Cliente-Servidor

- Exemplo 1:
  - Servidor - *socket domínio AF\_UNIX (3/7)*

```
22. if (bind(sock_fd, (struct sockaddr*)&sock_ser, sock_len) < 0) {  
23.     printf("Erro ao associar nome ao socket...\n");  
24.     exit(0);  
25. }  
  
26. listen(sock_fd, 5);
```

# Comunicação Cliente-Servidor

- Exemplo 1:
  - Servidor - *socket domínio AF\_UNIX (4/7)*

```
27. for (;;) {  
28.     sock_novo_len = sizeof(sock_cli);  
29.     sock_novo = accept(sock_fd, (struct sockaddr*)&sock_cli, &sock_novo_len);  
30.     if (sock_novo < 0) {  
31.         printf("Erro ao tentar estabeler conexao com o cliente...\n");  
32.         exit(0);  
33.     }
```

# Comunicação Cliente-Servidor

- Exemplo 1:
  - Servidor - *socket domínio AF\_UNIX (5/7)*

```
34.    if (fork() == 0) {  
35.        close(sock_fd);  
36.        if (read(sock_novo, msg, sizeof(msg)) < 0) {  
37.            printf("Erro de leitura do socket\n");  
38.            exit(0);  
39.        }  
40.        num = atoi(msg);  
41.        printf("Número recebido...: %d\n", num);  
42.        if (verifica_par(num)) {  
43.            write(sock_novo, "PAR", strlen("PAR")+1);  
44.        }
```

# Comunicação Cliente-Servidor

- Exemplo 1:
  - Servidor - *socket domínio AF\_UNIX (6/7)*

```
45. else {  
46.     write(sock_novo, "IMPAR", strlen("IMPAR")+1);  
47. }  
48. exit(0);  
49. }  
50. close(sock_novo);  
51. }  
52. return 0;  
53. }
```

# Comunicação Cliente-Servidor

- Exemplo 1:
  - Servidor - *socket domínio AF\_UNIX (7/7)*

```
54. int verifica_par(int num)
55. {
56.     if (num % 2 == 0)
57.         return 1;
58.     return 0;
59. }
```

# Comunicação Cliente-Servidor

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

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <sys/types.h>
5. #include <sys/socket.h>
6. #include <sys/un.h>

7. int main(int argc, char** argv)
8. {
9.     int sock_fd, sock_len, num;
10.    struct sockaddr_un sock_cli;
11.    char msg[100];
```

# Comunicação Cliente-Servidor

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

```
12. if (argc != 2) {  
13.     printf("Uso: %s numero\n", argv[0]);  
14.     exit(0);  
15. }  
16.  
17. sock_fd = socket(AF_UNIX, SOCK_STREAM, 0);  
18. if (sock_fd < 0) {  
19.     printf("Erro ao criar o socket...\n");  
20.     exit(0);  
21. }
```

# Comunicação Cliente-Servidor

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

```
22. bzero((char*)&sock_cli, sizeof(sock_cli));  
23. sock_cli.sun_family = AF_UNIX;  
24. strcpy(sock_cli.sun_path, "socket_unix.teste");  
25. sock_len = strlen(sock_cli.sun_path) + sizeof(sock_cli.sun_family);  
  
26. if (connect(sock_fd, (struct sockaddr*)&sock_cli, sock_len) < 0) {  
27.     printf("Erro ao tentar conectar com o servidor...\n");  
28.     exit(0);  
29. }
```

# Comunicação Cliente-Servidor

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

```
30.  write(sock_fd, argv[1], strlen(argv[1]));
31.  if (read(sock_fd, msg, sizeof(msg)) < 0) {
32.      printf("Erro na leitura da resposta do servidor...\n");
33.      exit(0);
34.  }
35.  printf("O numero %s eh %s\n", argv[1], msg);

36.  close(sock_fd);
37.
38.  return 0;
39. }
```

# Comunicação Cliente-Servidor

- Exemplo 2:
  - Servidor - *socket domínio AF\_INET (1/5)*

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <sys/types.h>
4. #include <sys/socket.h>
5. #include <arpa/inet.h>
6. #include <string.h>
7. #include <time.h>

8. int main(int argc, char** argv)
9. {
10.     int socket_des, fromlen, sock_des_cli;
11.     struct sockaddr_in servidor, sock_cli;
12.     char msg_buffer[80];
```

# Comunicação Cliente-Servidor

- Exemplo 2:
  - Servidor - *socket domínio AF\_INET (2/5)*

```
13. if (argc < 2) {  
14.     fprintf(stderr,"USO: %s <porta>\n",argv[0]);  
15.     exit(0);  
16. }  
  
17. if ((socket_des = socket(AF_INET, SOCK_STREAM, 0)) < 0) {  
18.     fprintf(stderr,"Erro ao criar o socket do servidor\n");  
19.     exit(0);  
20. }  
21. servidor.sin_family = AF_INET;  
22. servidor.sin_port = htons(atoi(argv[1]));  
23. servidor.sin_addr.s_addr = htonl(INADDR_ANY);  
24. bzero(&(servidor.sin_zero), 8);
```

# Comunicação Cliente-Servidor

- Exemplo 2:
  - Servidor - *socket domínio AF\_INET (3/5)*

```
25. if (bind(socket_des,(struct sockaddr*)&servidor,sizeof(servidor)) < 0) {  
26.     fprintf(stderr,"Erro de bind\n");  
27.     close(socket_des);  
28.     exit(0);  
29. }  
  
30. if (listen(socket_des,3) < 0) {  
31.     fprintf(stderr,"Erro de listen\n");  
32.     exit(0);  
33. }  
34. else {  
35.     printf("Servidor pronto para conexoes...\n\n");  
36. }
```

# Comunicação Cliente-Servidor

- Exemplo 2:
  - Servidor - *socket domínio AF\_INET (4/5)*

```
37. for ( ; ; ) {  
38.     fromlen = sizeof(sock_cli);  
39.     if ((sock_des_cli = accept(socket_des,(struct sockaddr*)&sock_cli,&fromlen)) < 0) {  
40.         fprintf(stderr,"Erro de conexao...\n");  
41.         exit(0);  
42.     }  
43.     else {  
44.         fprintf(stderr,"SERVIDOR::Cliente conectou...\n");  
45.         strcpy(msg_buffer,"<html> Bem vindo ao servidor MicroWeb. Voce conectou em: ");  
46.         strcat(msg_buffer,hora_data());  
47.         strcat(msg_buffer,"</html>\n");  
48.         write(sock_des_cli,msg_buffer,strlen(msg_buffer));  
49.         close(sock_des_cli);  
50.     }  
51. }
```

# Comunicação Cliente-Servidor

- Exemplo 2:
  - Servidor - *socket domínio AF\_INET (5/5)*

```
53. char* hora_data()
54. {
55.     struct tm *tempo;
56.     time_t lt;
57.     lt = time(NULL);
58.     tempo = localtime(&lt);
59.     return asctime(tempo);
60. }
```

# Comunicação Cliente-Servidor

- Exemplo 2:
  - Servidor - *socket domínio AF\_INET*
    - Para testar o servidor basta compilá-lo e então executá-lo passando como parâmetro o número de uma porta.
    - Exemplo:
      - **./servidor 8000**
    - O cliente deste servidor poderá ser o browser, para tanto basta digitar o endereço **localhost:porta**, onde porta se refere ao número passada como parâmetro ao servidor.
    - O resultado da comunicação entre o cliente e o servidor será a mensagem:
      - **Bem vindo ao servidor MicroWeb. Voce conectou em: Tue May 20 22:39:16 2014**