

Sistemas Operacionais

Comunicação entre Processos

Prof. Otávio Gomes

otavio.gomes@unifei.edu.br

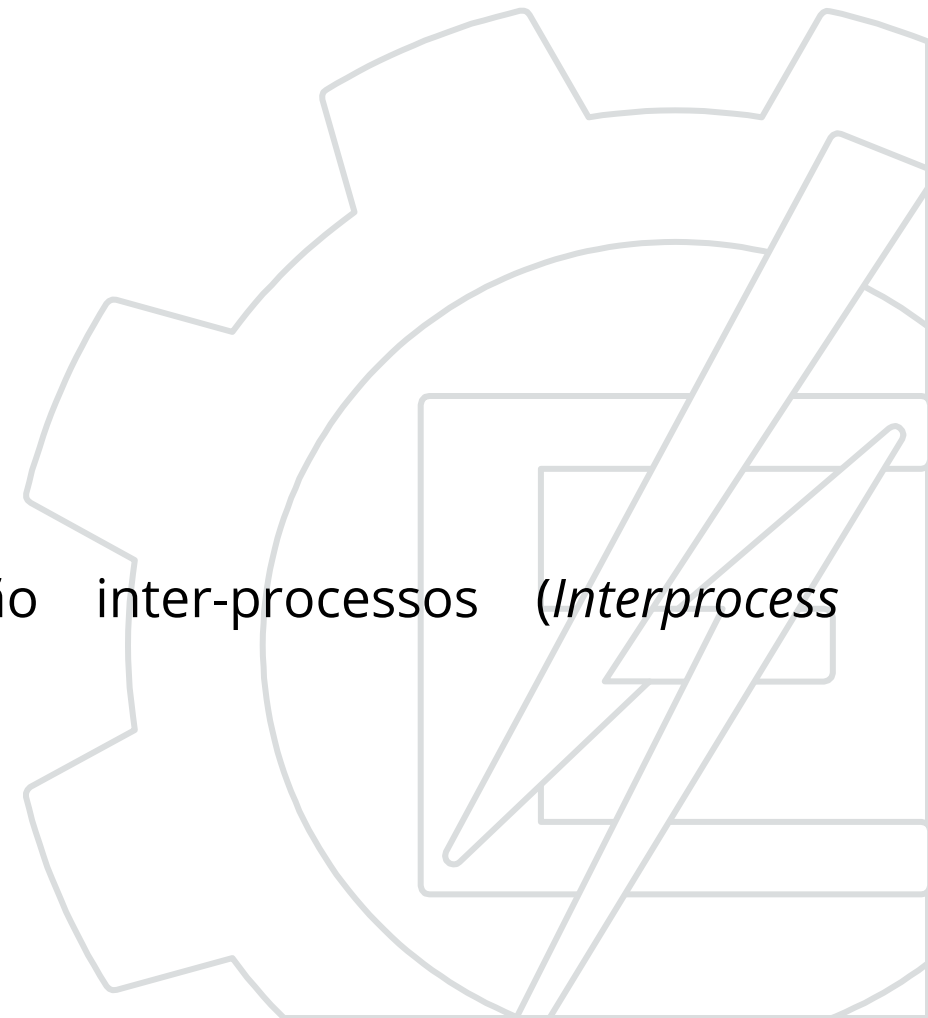


Comunicação entre Processos

Interprocess Communication – IPC

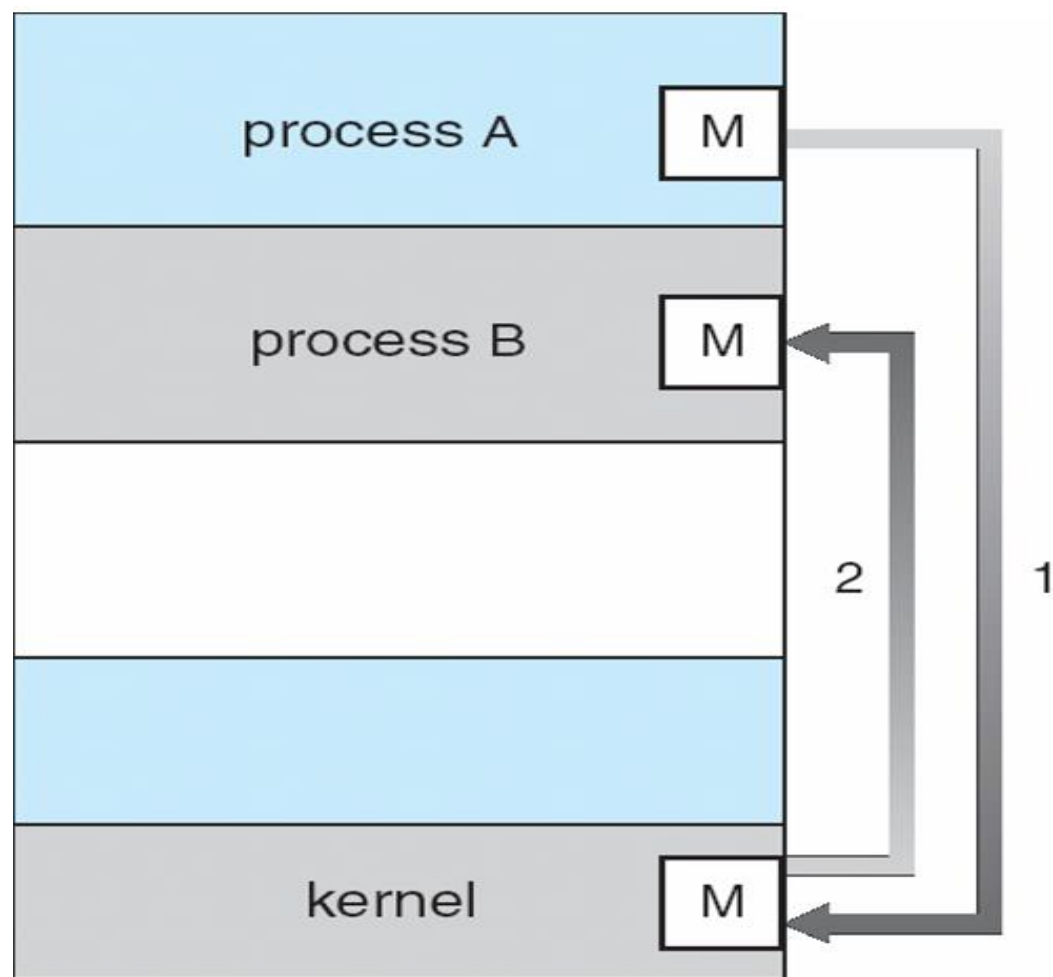


- Processos em um sistema podem ser independentes ou colaborativos;
- Razões para se ter processos colaborativos:
 - **Compartilhamento** de informação;
 - Melhoria de **desempenho**;
 - **Modularidade** (ex.: *spellchecker* invocado pelo LaTeX);
 - **Conveniência** (ex.: *web browser* + *anti-malware*).
- Processos colaborativos necessitam de comunicação inter-processos (*Interprocess Communication* - IPC) que possuem dois modelos:
 - **Memória compartilhada**
 - **Passagem/Troca de mensagens**

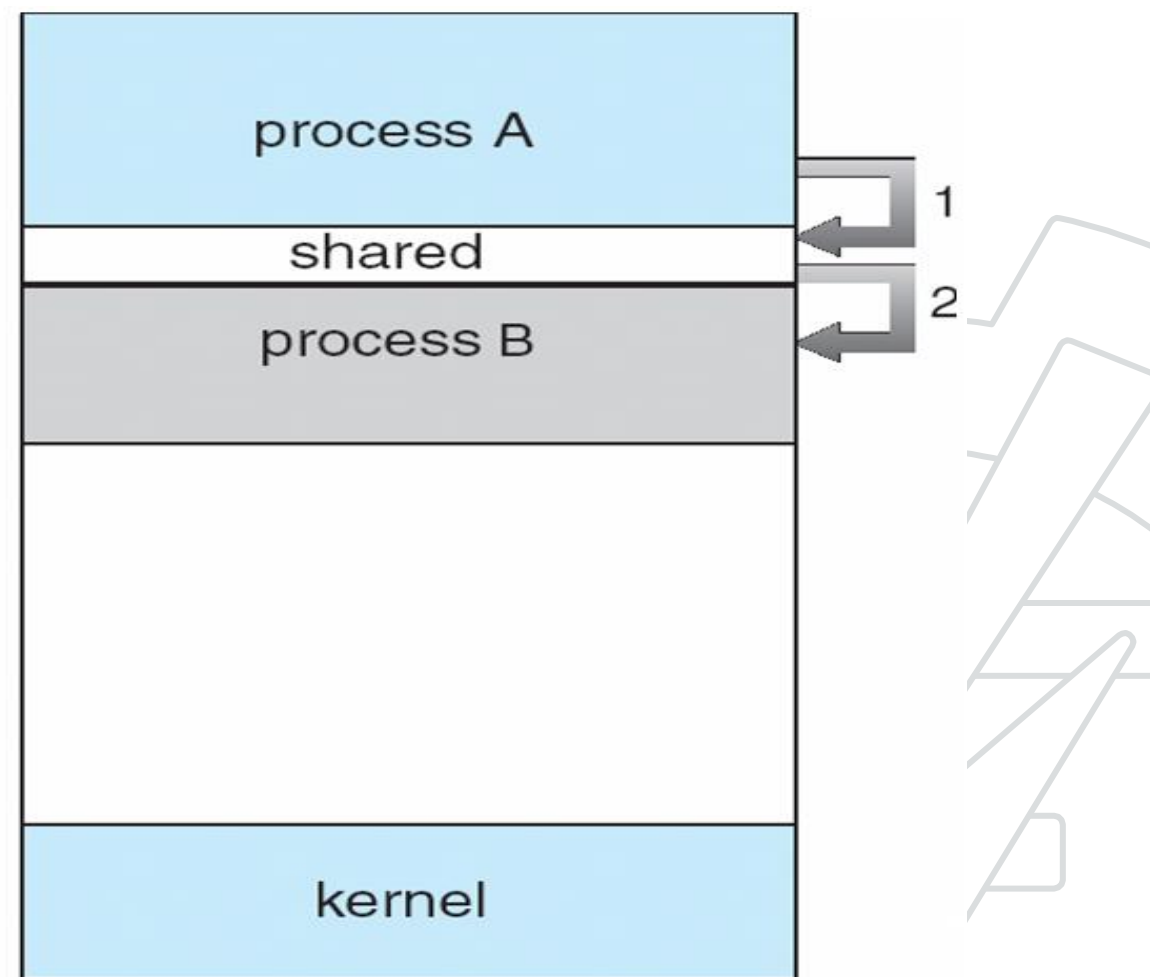


Processos

Modelos de IPC

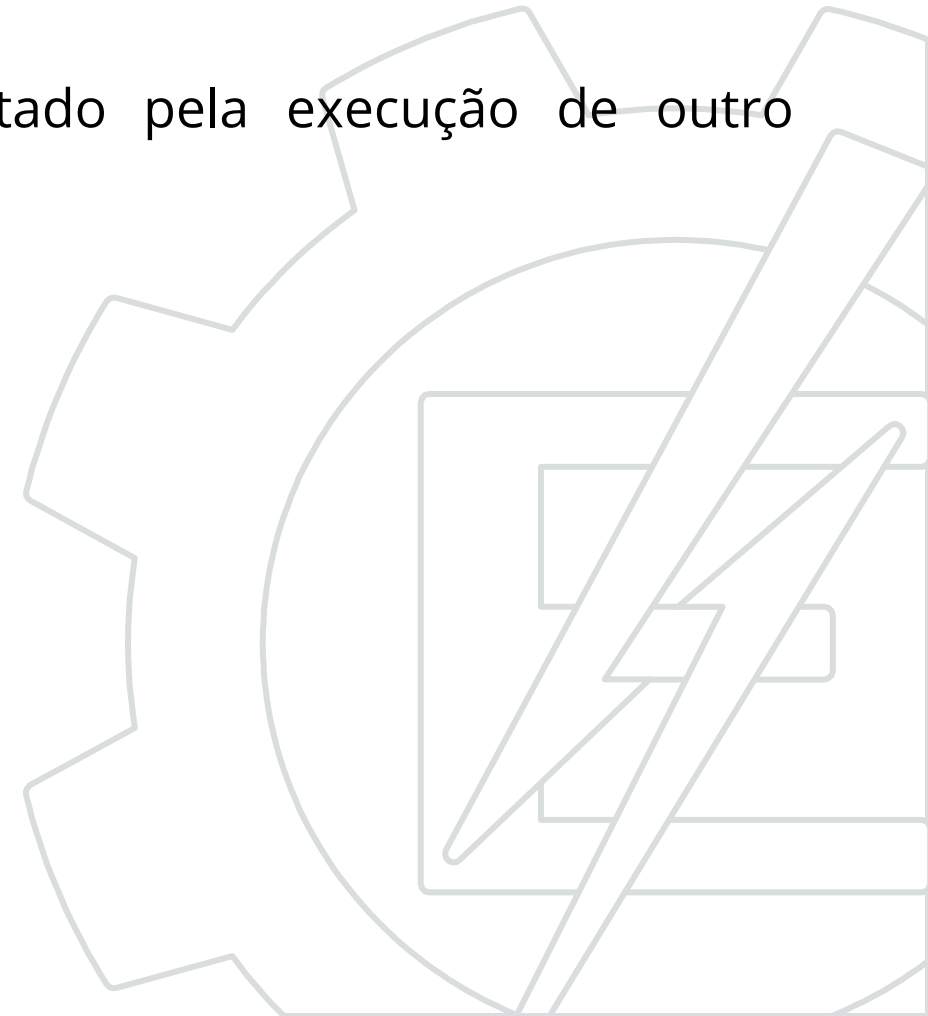


(a)

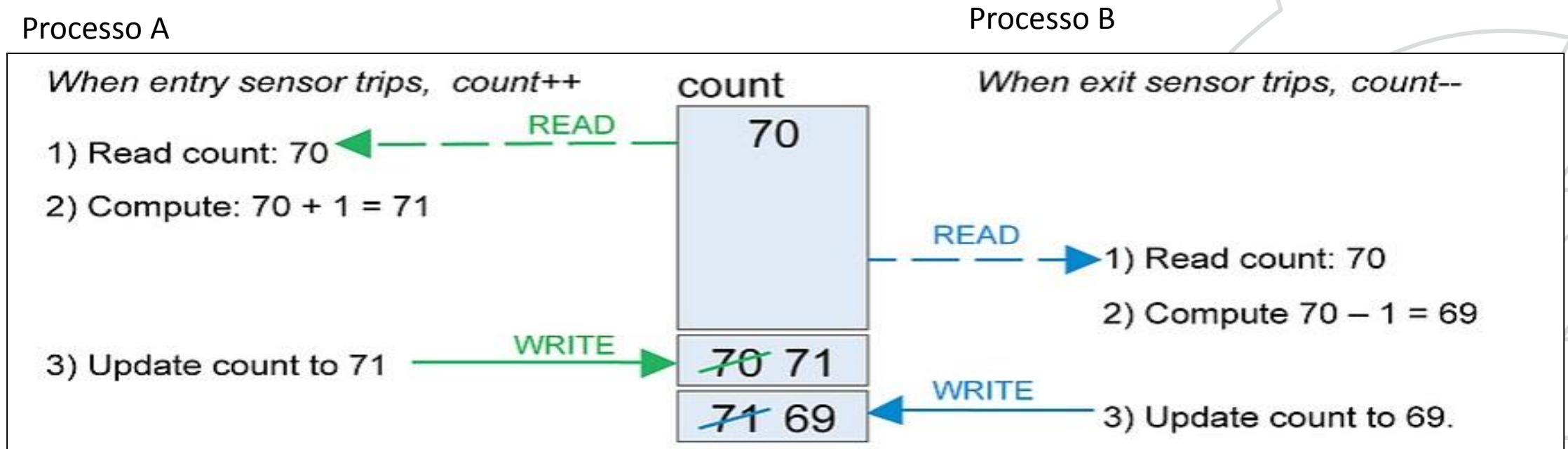


(b)

- Um **processo independente** não pode afetar ou ser afetado pela execução de um outro processo.
- Um **processo colaborativo** pode afetar ou ser afetado pela execução de outro processo.
- Vantagens da colaboração entre processos:
 - Compartilhamento de informação
 - Melhoria de desempenho
 - Modularidade
 - Conveniência



- Acesso concorrente a dados compartilhados pode resultar em **inconsistências**;
- **Condições de corrida** (*race conditions*):
 - Situação onde dois ou mais processos acessam e manipulam recursos compartilhados simultaneamente.



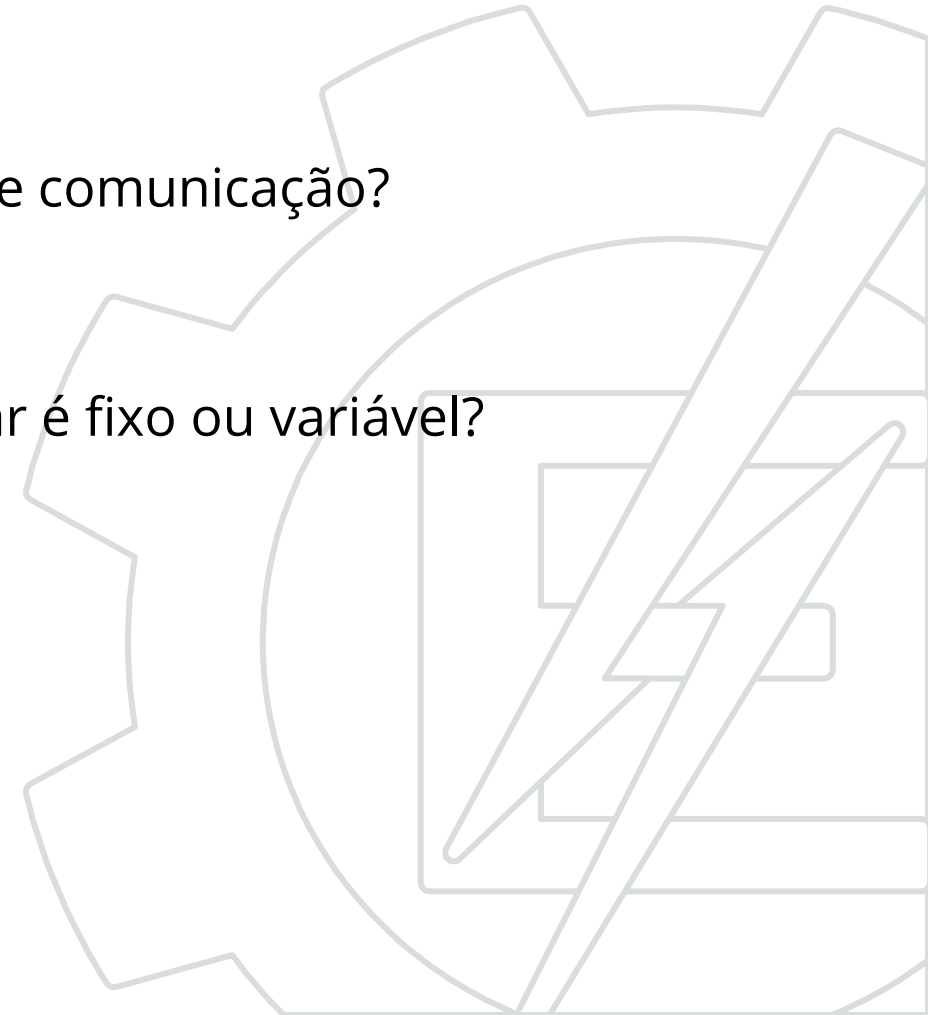
- Mecanismo para processos se comunicarem e sincronizarem suas ações
- Sistema de Mensagem – processos se comunicam sem recorrer a variáveis compartilhadas
- O mecanismo de IPC fornece duas operações:
 - **`send(mensagem)` – tamanho da mensagem fixo ou variável**
 - **`receive(mensagem)`**



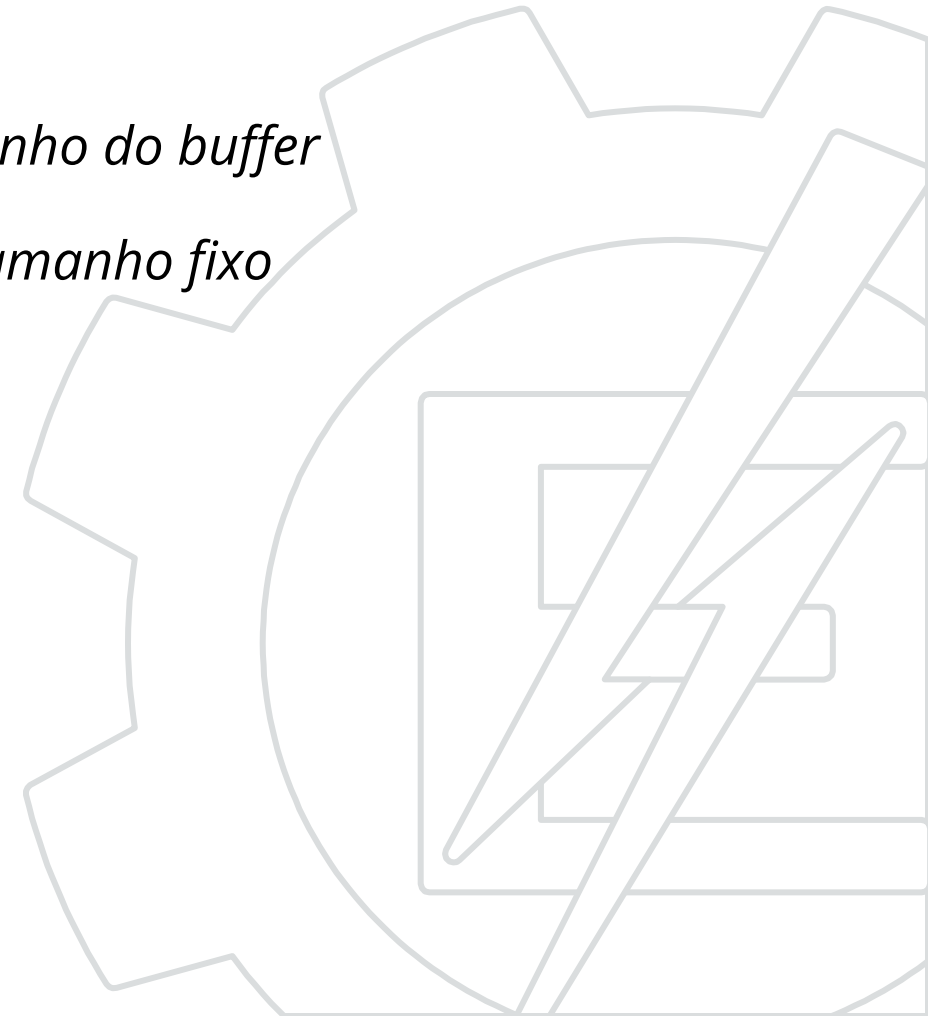
- Se P e Q desejam comunicar, eles precisam:
 - Estabelecer um ***link*** de comunicação entre eles
 - Trocar mensagens via ***send/receive***
- Implementação do *link* de comunicação
 - Física (ex:, memória compartilhada, barramento de *hardware*)
 - Lógica (ex: propriedades lógicas)



- Como os *links* são estabelecidos?
- Um *link* pode ser associado com mais de dois processos?
- Quantos *links* podem existir entre cada par de processos de comunicação?
- Qual é a capacidade de um *link*?
- O tamanho de uma mensagem que um *link* pode acomodar é fixo ou variável?
- Um *link* é unidirecional ou bi-direcional?



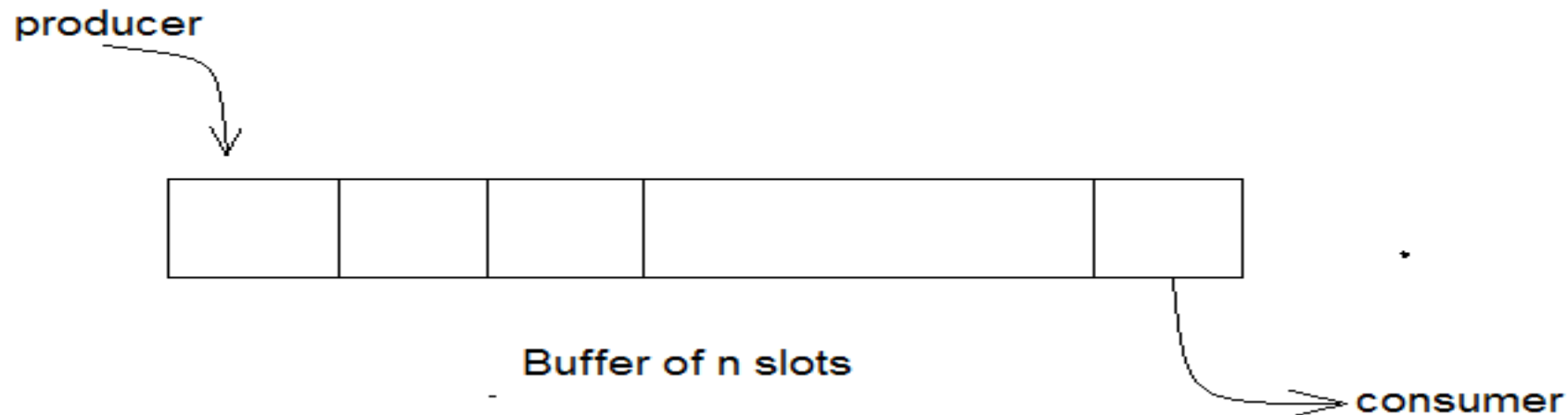
- Paradigma de processos colaborativos: o processo produtor produz informação que é consumida por um processo consumidor:
 - *unbounded-buffer* não restringe limite prático no tamanho do buffer
 - *bounded-buffer* assume a existência de um buffer de tamanho fixo



Para comunicações diretas ou indiretas a **troca de mensagens** entre os processos é **armazenada (*buffering*)** em uma fila temporária que pode ser implementada, basicamente, de três formas:

- 1) ***Zero capacity*** (Capacidade zero): A fila possui o tamanho máximo igual a zero, isto é, a conexão não permite qualquer espera de mensagem. Neste caso, o processo de envio (***sender***) deve bloquear até que o destinatário receba a mensagem.
- 2) ***Unbounded capacity*** (Capacidade ilimitada): A capacidade da fila é potencialmente infinita, isto é, qualquer número de mensagens que for recebida será armazenada. O remetente (***sender***) nunca é bloqueado.

3) **Bounded capacity** (Capacidade limitada): A fila possui uma capacidade de armazenamento finita igual a n , isto é, pelo menos n mensagens podem ser armazenadas neste buffer. Se a fila não estiver cheia durante a chegada da mensagem, a mesma é armazenada na fila e o remetente (**sender**) pode continuar o envio sem ter que aguardar. Se a fila estiver cheia, o remetente deve ser bloqueado até que seja liberado espaço na fila.



- Dado compartilhado

```
#define BUFFER_SIZE 10
```

```
typedef struct {
```

```
    ...
```

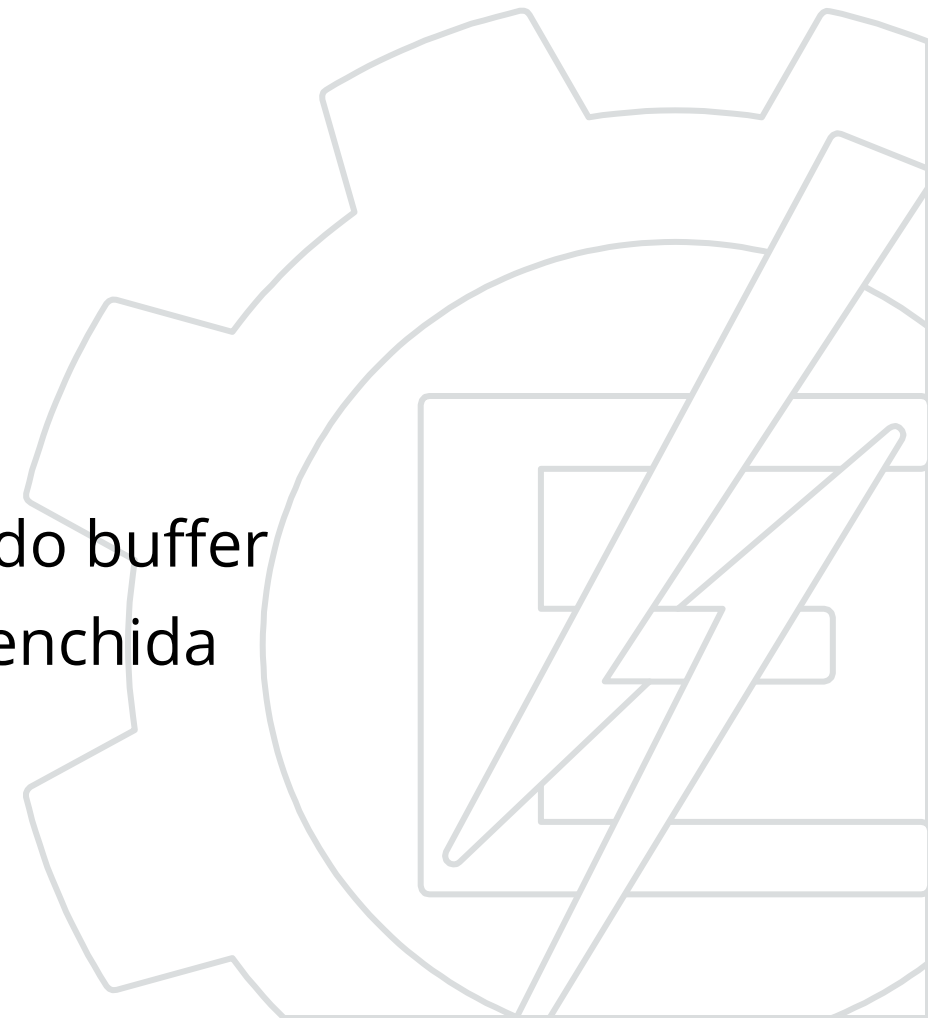
```
} item;
```

```
item buffer[BUFFER_SIZE];
```

```
int in = 0; //aponta para a próxima posição livre do buffer
```

```
int out = 0; // aponta para a primeira posição preenchida
```

OBS.: A solução está correta, mas somente pode usar `BUFFER_SIZE-1` elementos



item nextProduced;

while (true) {

/* produz um novo item e armazena em nextProduced */

while (((in + 1) % BUFFER SIZE) == out) //buffer cheio

; /* não faz nada */

// coloca o item produzido no *buffer*

buffer[in] = nextProduced;

in = (in + 1) % BUFFER SIZE;

}



item nextConsumed;

```
while (true) {  
    while (in == out) //buffer vazio  
        ; /* não faz nada */  
  
    // remove um item do buffer  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER SIZE;  
    /* consome o item contido em nextConsumed */  
}
```



1. Capacidade zero – 0 mensagem

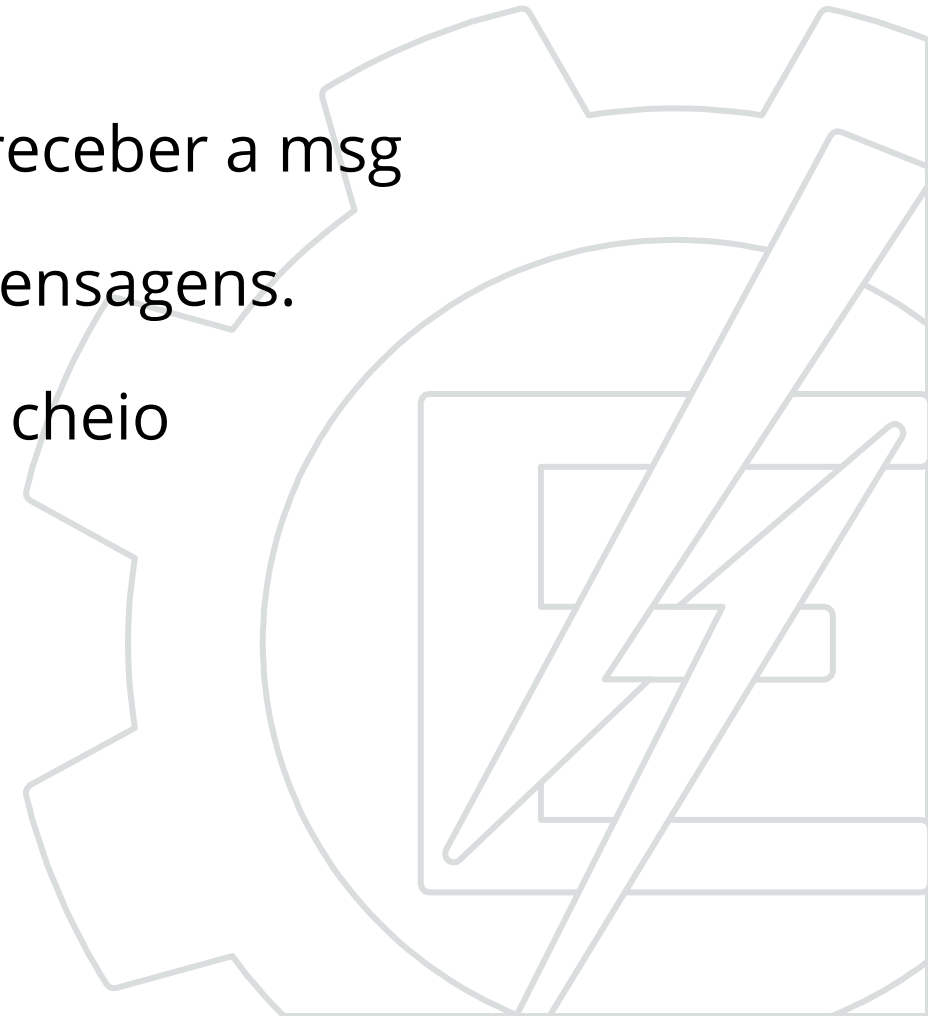
Remetente deve esperar até o destinatário receber a msg

2. Capacidade limitada – tamanho finito de n mensagens.

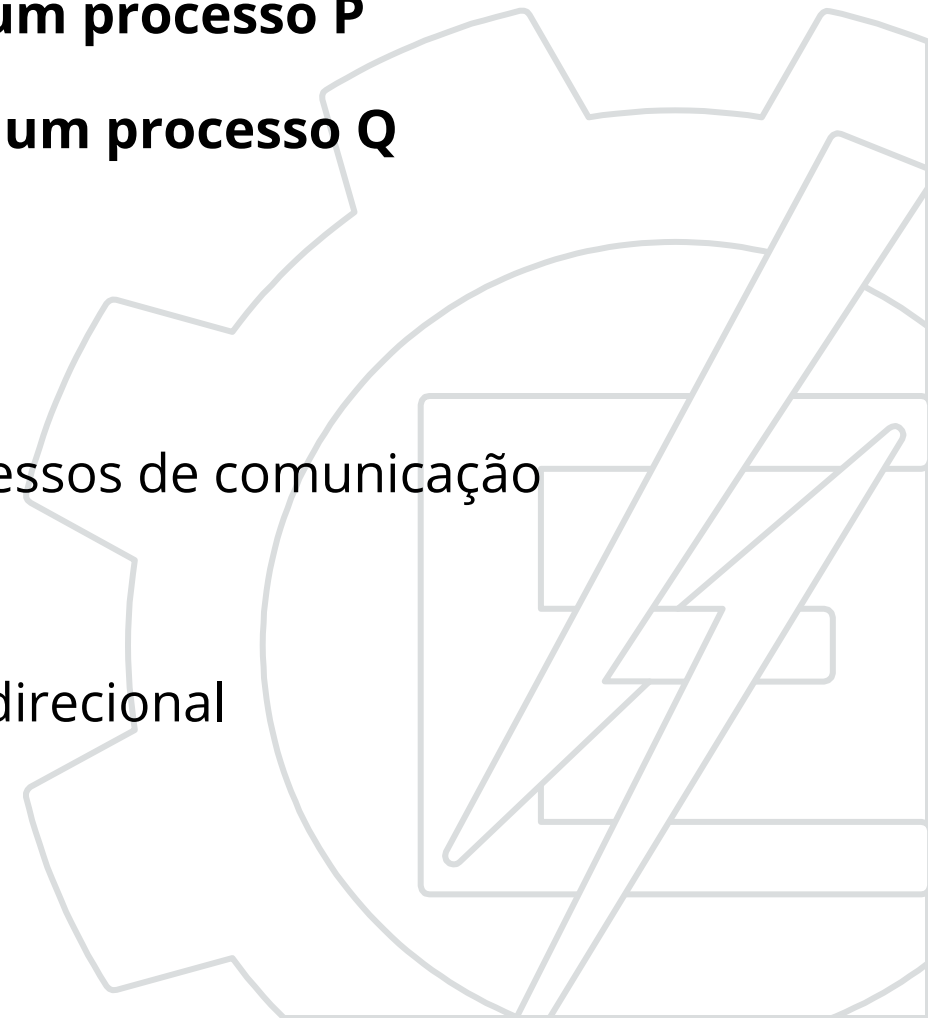
O remetente deve aguardar se o link estiver cheio

3. Capacidade ilimitada – tamanho infinito.

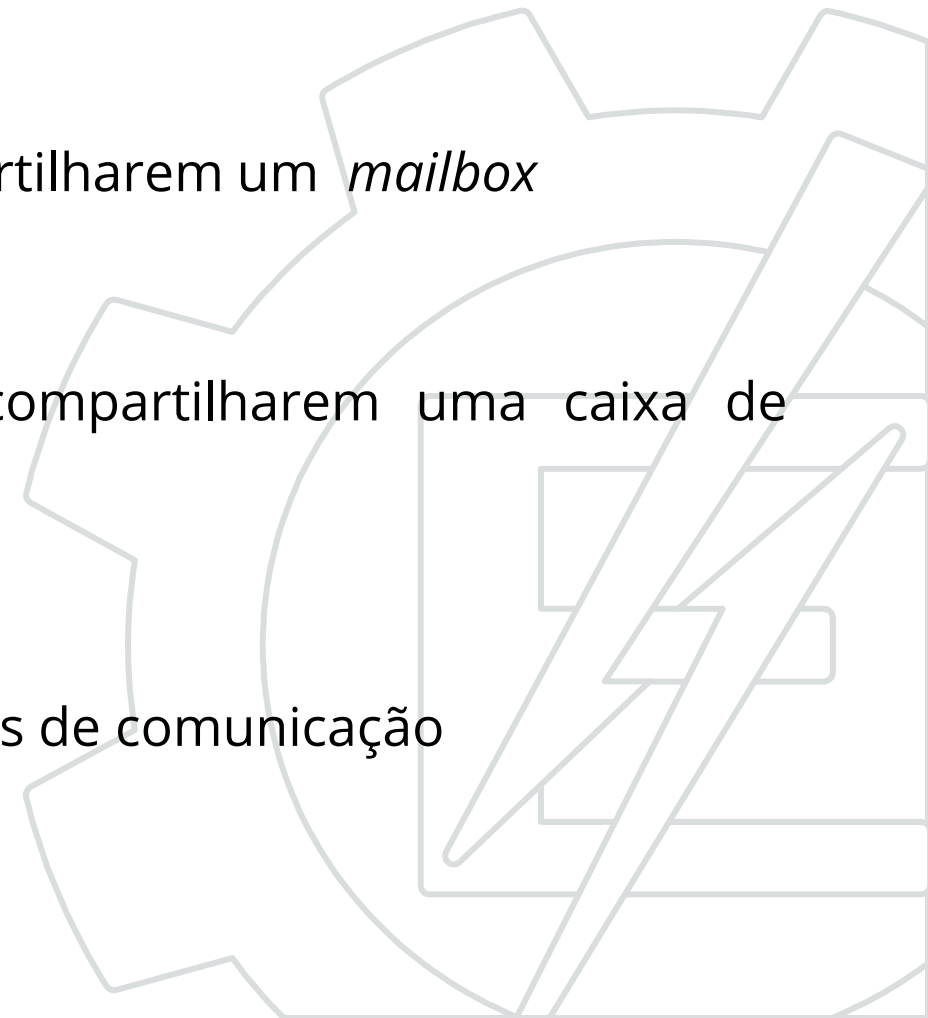
O remetente nunca espera



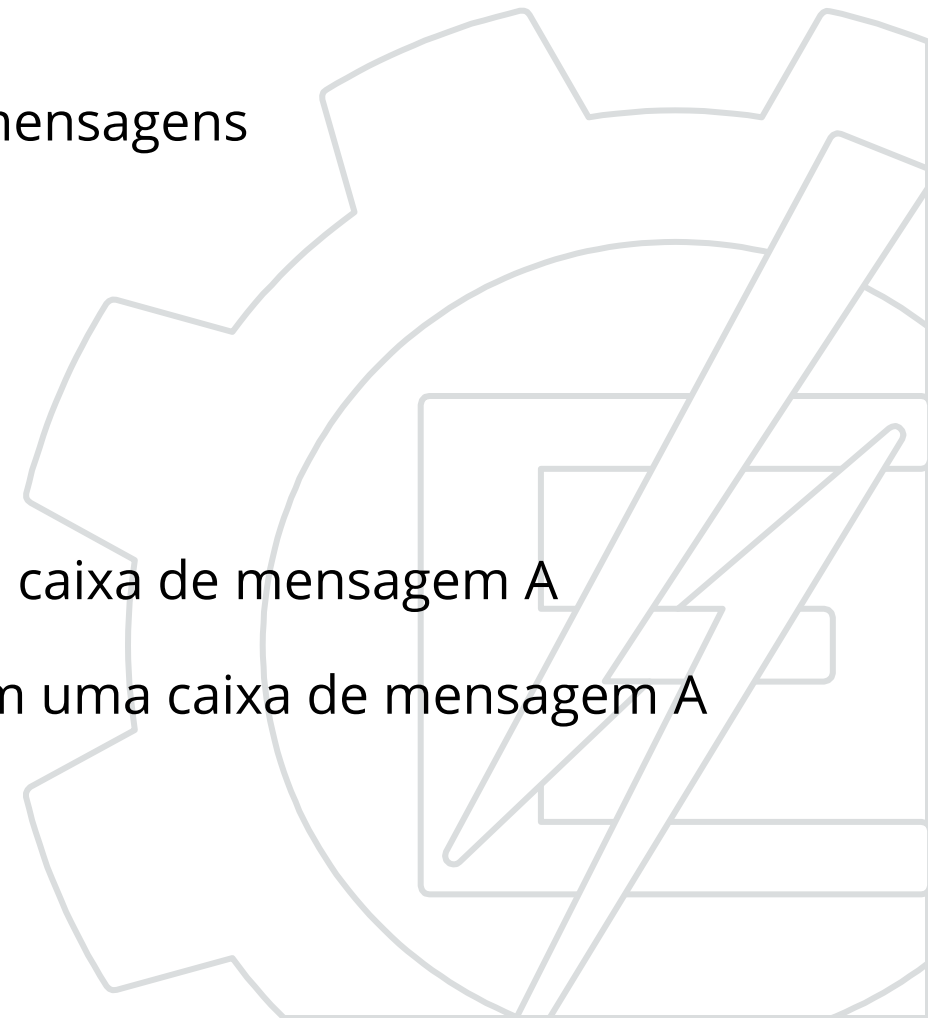
- Processos devem nomear um ao outro explicitamente:
 - **`send(P, mensagem)`** – envia uma mensagem para um processo P
 - **`receive(Q, mensagem)`** – recebe uma mensagem de um processo Q
- Propriedades de um *link* de comunicação:
 - *Links* são estabelecidos automaticamente
 - Um *link* é associado com exatamente um par de processos de comunicação
 - Entre cada par existe exatamente um *link*
 - O *link* pode ser unidirecional, mas é normalmente bi-direcional



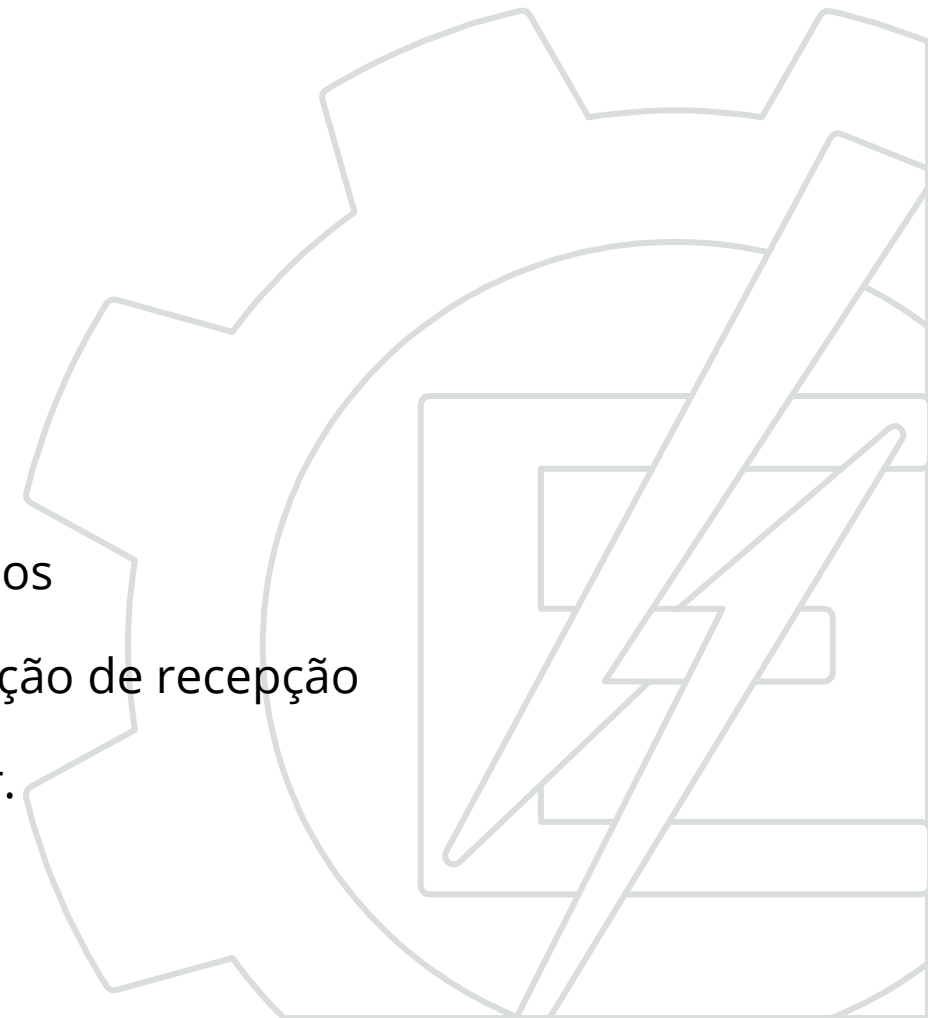
- Mensagens são direcionadas e recebidas de caixas de mensagens - **mailboxes** (também referenciadas como portas)
 - Cada *mailbox* tem um único **id**
 - Os processos pode comunicar somente se eles compartilharem um *mailbox*
- Propriedades do *link* de comunicação
 - O *link* é estabelecido somente se os processos compartilharem uma caixa de mensagem em comum
 - Um *link* pode ser associado com vários processos
 - Cada par de processos podem compartilhar vários links de comunicação
 - Um *link* pode ser uni ou bi-direcional



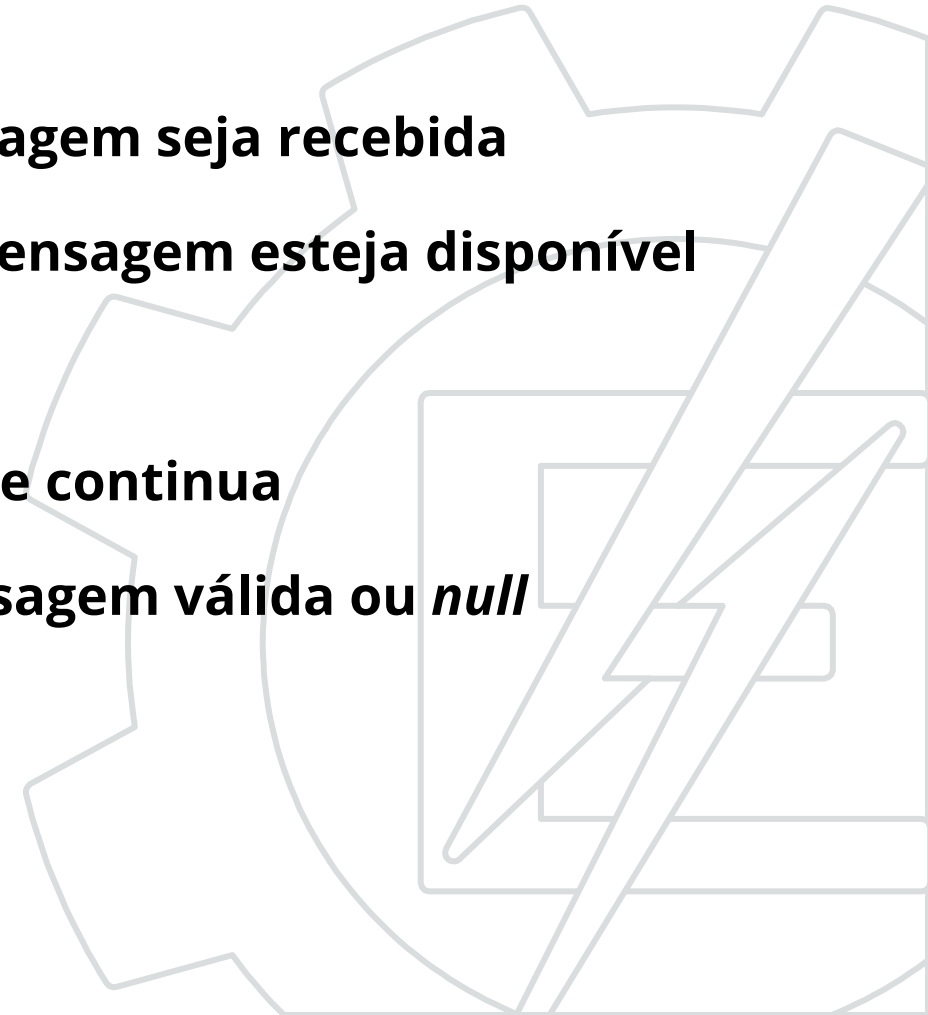
- Operações
 - Criar uma nova caixa de mensagem
 - Enviar e receber mensagens através da caixa de mensagens
 - Destruir uma caixa de mensagem
- As primitivas são definidas como:
 - send**($A, mensagem$) – envia uma mensagem para uma caixa de mensagem A
 - receive**($A, mensagem$) – recebe uma mensagem de um uma caixa de mensagem A



- Compartilhamento de uma caixa de mensagem
 - P_1 , P_2 , e P_3 compartilham uma caixa de mensagem A
 - P_1 , envia; P_2 e P_3 recebem
- Questão: Quem obtém a mensagem?
- Soluções
 - Permitir um *link* ser associado com no máximo dois processos
 - Permitir somente um processo por vez executar uma operação de recepção
 - Permitir que o sistema selecione arbitrariamente o receptor.



- A passagem de mensagem pode ser tanto *blocking* quanto *non-blocking*
- ***Blocking*** é considerada síncrona
 - ***Blocking send*** bloqueia o remetente até que a mensagem seja recebida
 - ***Blocking receive*** bloqueia o receptor até que uma mensagem esteja disponível
- ***Non-blocking*** é considerada assíncrona
 - ***Envio Non-blocking*** o remetente envia a mensagem e continua
 - ***Recepção Non-blocking*** o receptor recebe uma mensagem válida ou *null*



Exemplos de Comunicação Inter-Processos



- Memória compartilhada no **POSIX**

- O processo primeiramente **cria** um segmento de memória compartilhada

segment id = shmget(IPC PRIVATE, size, S IRUSR | S IWUSR);

- O processo que quer acesso a esta memória compartilhada deve se **anexar** a ela

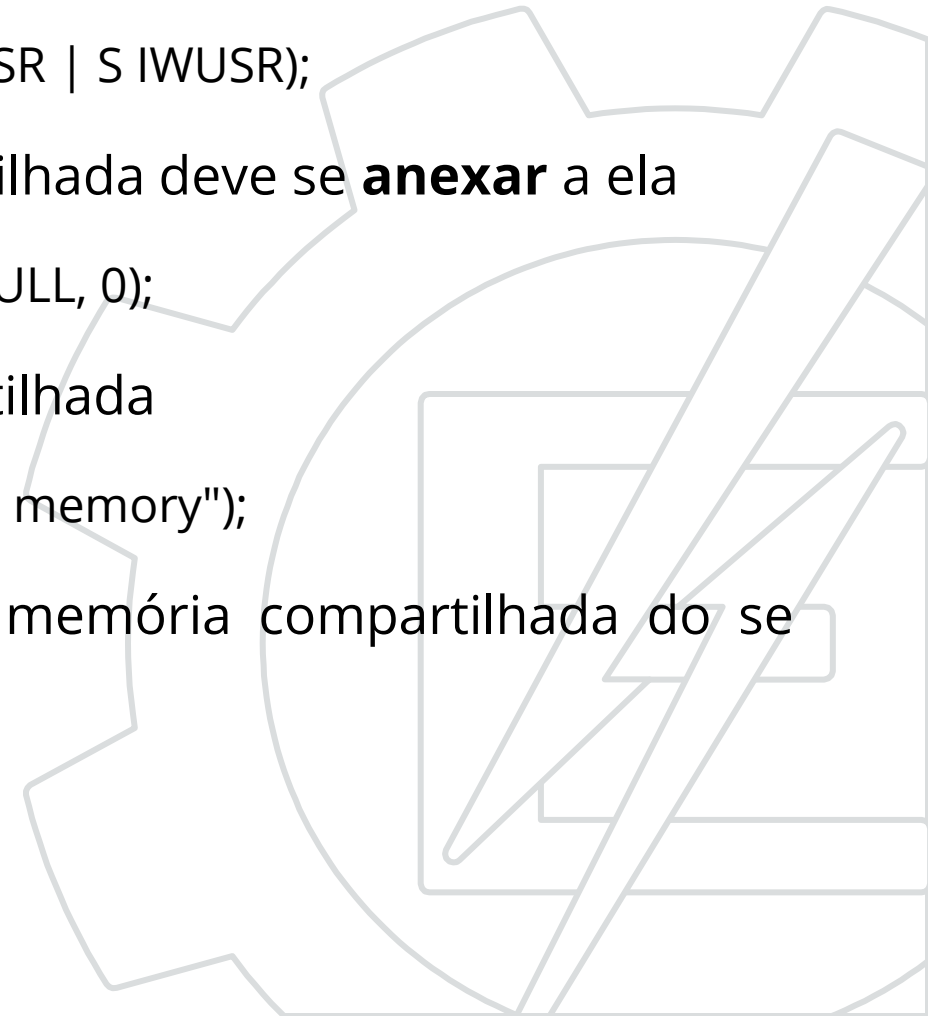
shared memory = (char *) shmat(id, NULL, 0);

- Agora o processo pode escrever na memória compartilhada

sprintf(shared memory, "Writing to shared memory");

- Quando pronto, um processo pode desconectar a memória compartilhada do seu espaço de endereçamento

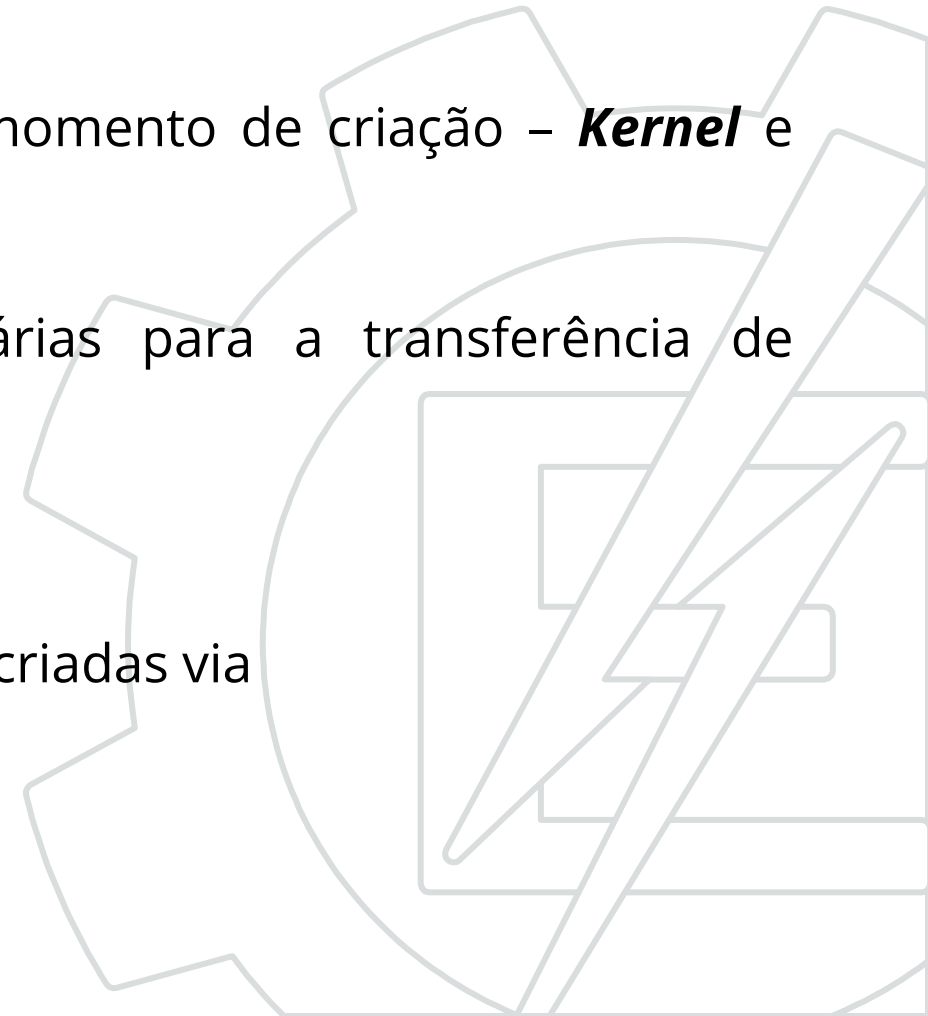
shmdt(shared memory);



- A comunicação no Mach é baseada em mensagens
 - Mesmo as chamadas de sistema são mensagens
 - Cada tarefa obtêm duas caixas de mensagens no momento de criação – **Kernel** e **Notify**.
 - Somente três chamadas de sistema são necessárias para a transferência de mensagem

`msg_send(), msg_receive(), msg_rpc()`

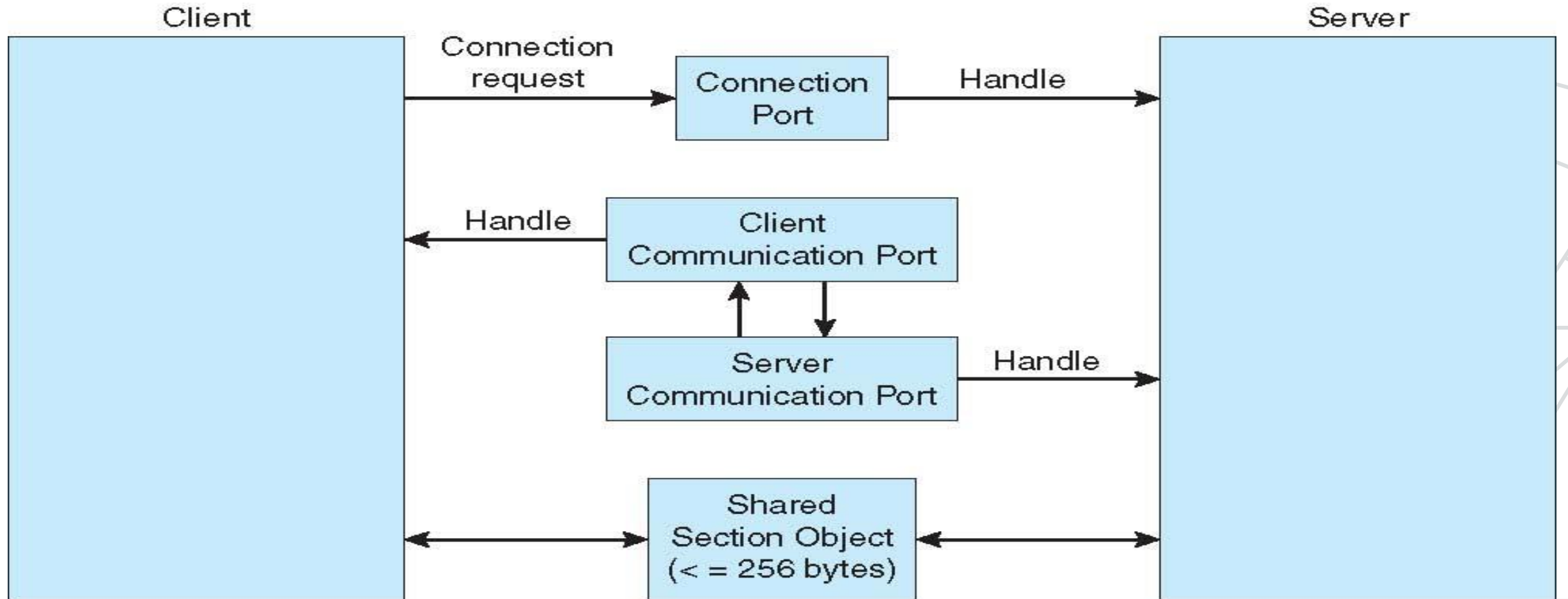
- Caixas de mensagem necessárias para comunicação, criadas via
`port_allocate()`



- Passagem de mensagem via mecanismo LPC (*Local Procedure Call*)
 - Somente funciona entre processos no mesmo sistema
 - Usa portas (da mesma forma que *mailboxes*) para estabelecer e manter os canais de comunicação
 - A comunicação funciona da seguinte maneira:
 - ✦ O cliente abre um *handle* para o *objeto da porta de conexão do subsistema*
 - ✦ O cliente envia uma requisição de conexão
 - ✦ O servidor cria duas portas de comunicação privadas e retorna o *handle* para um deles para o cliente
 - ✦ O cliente e o servidor usam o *handle* da porta correspondente para enviar mensagens e *callbacks* e escutar por respostas

Chamadas Locais de Procedimentos

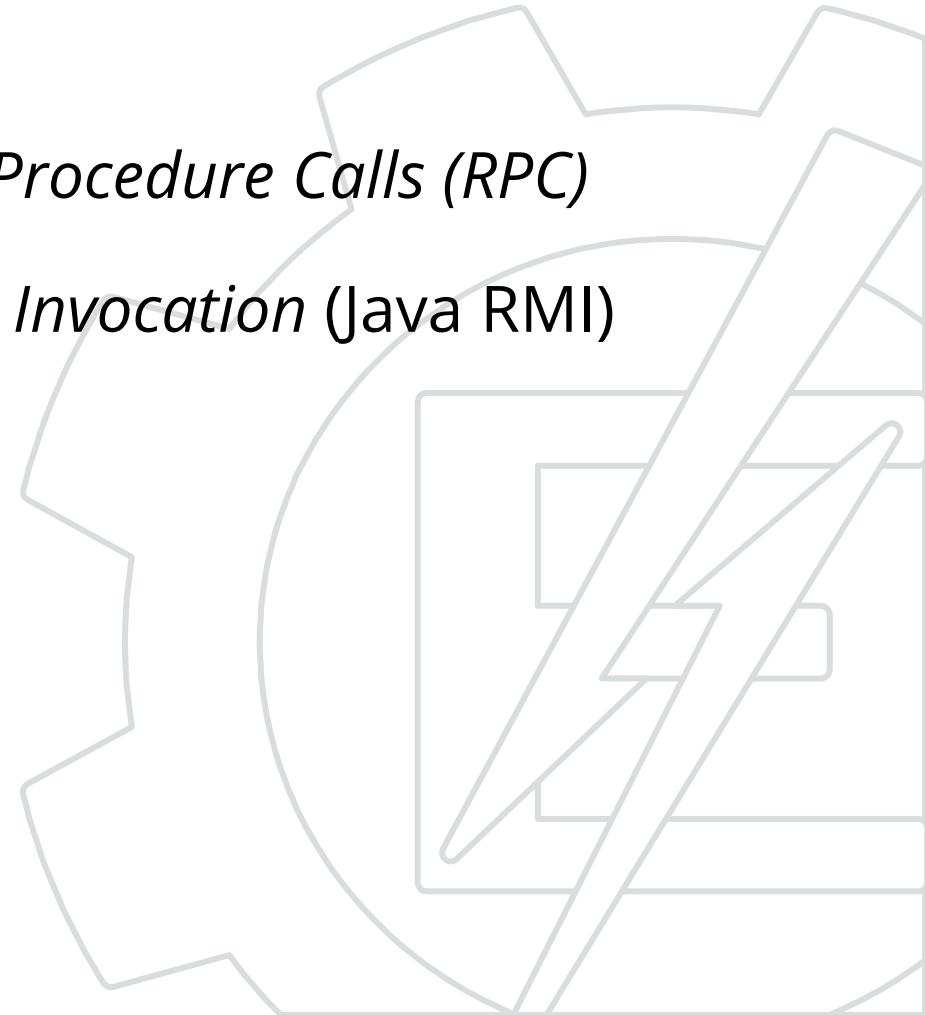
Windows XP



Comunicação Cliente-Servidor



- *Pipes*
- *Sockets*
- Chamadas Remotas de Procedimento - *Remote Procedure Calls (RPC)*
- Invocação Remota de Métodos - *Remote Method Invocation (Java RMI)*

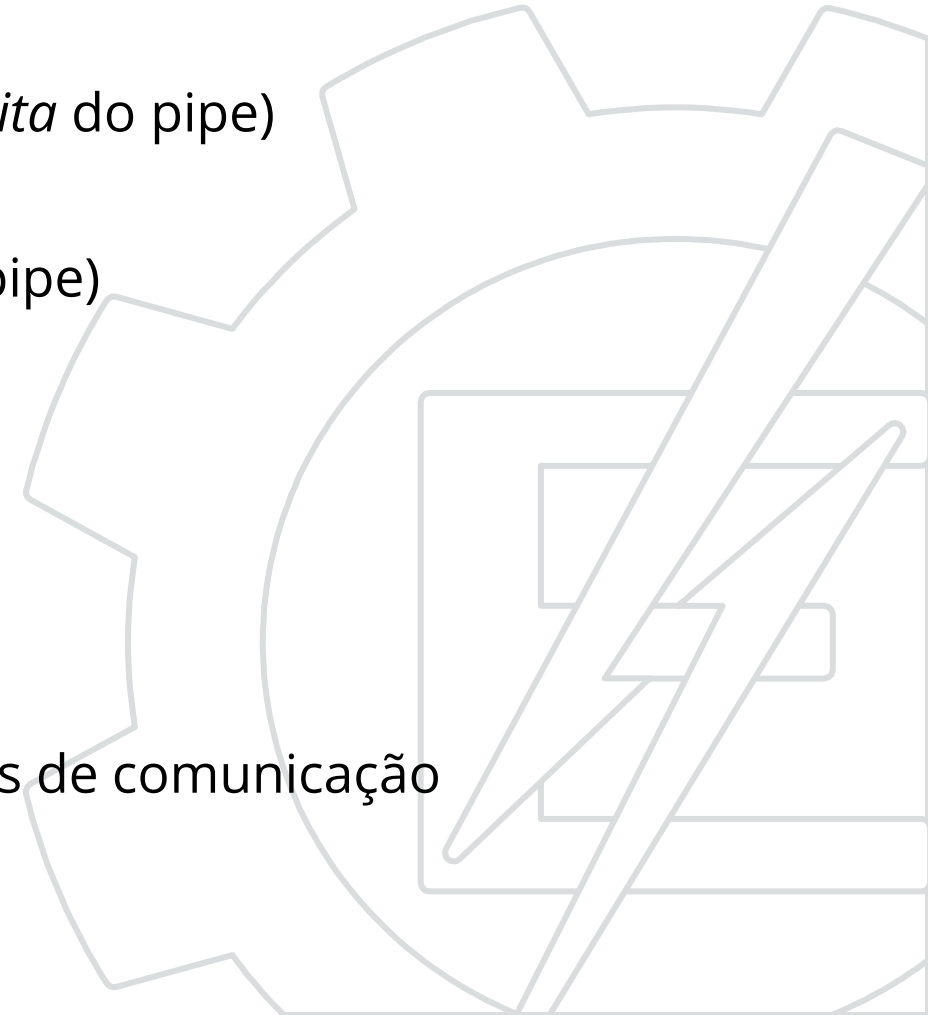


- Agem como um conduíte permitindo a comunicação entre dois processos
- Foram os primeiros mecanismos de comunicação entre processos (IPC) nos primeiros sistemas UNIX.
- Questões:
 - A comunicação é **unidirecional** ou **bi-direcional**?
 - No caso da comunicação bidirecional, ela é **half** ou **full-duplex**?
 - Deve existir um **relacionamento** (ex: pai-filho) entre os processos de comunicação?
 - Os pipes podem ser usados sobre uma **rede**?

Comunic. Cliente-Servidor

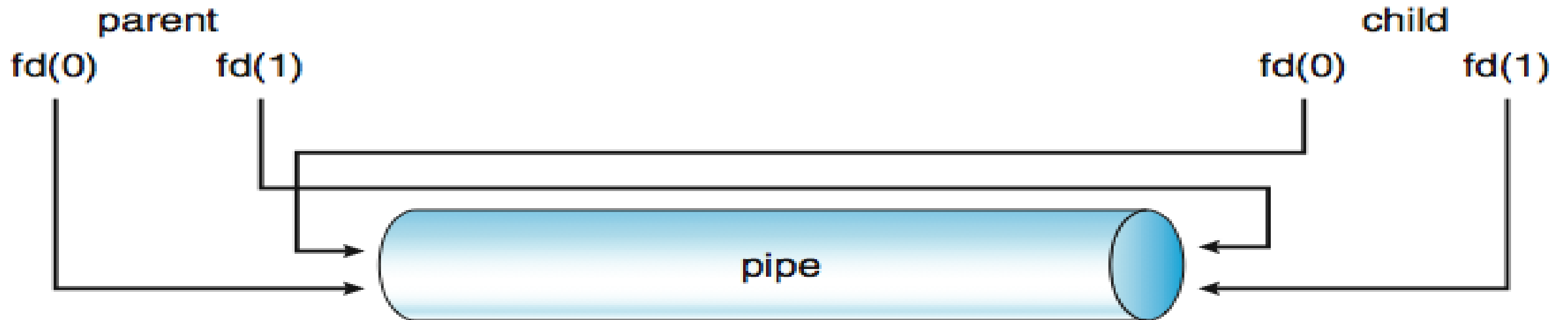
Pipes Comuns (*Ordinary pipes*)

- Os *Pipes* Comuns permitem a comunicação no estilo padrão produtor-consumidor
- O produtor escreve em um **extremo - end** (o *extremo de escrita* do pipe)
- O consumidor lê no outro extremo (o *extremo de leitura* do pipe)
- Os pipes comuns são portanto, **unidirecionais**
 - Duas vias de comunicação requerem dois *pipes*
- Requerem um **relacionamento pai-filho** entre os processos de comunicação



Comunic. Cliente-Servidor

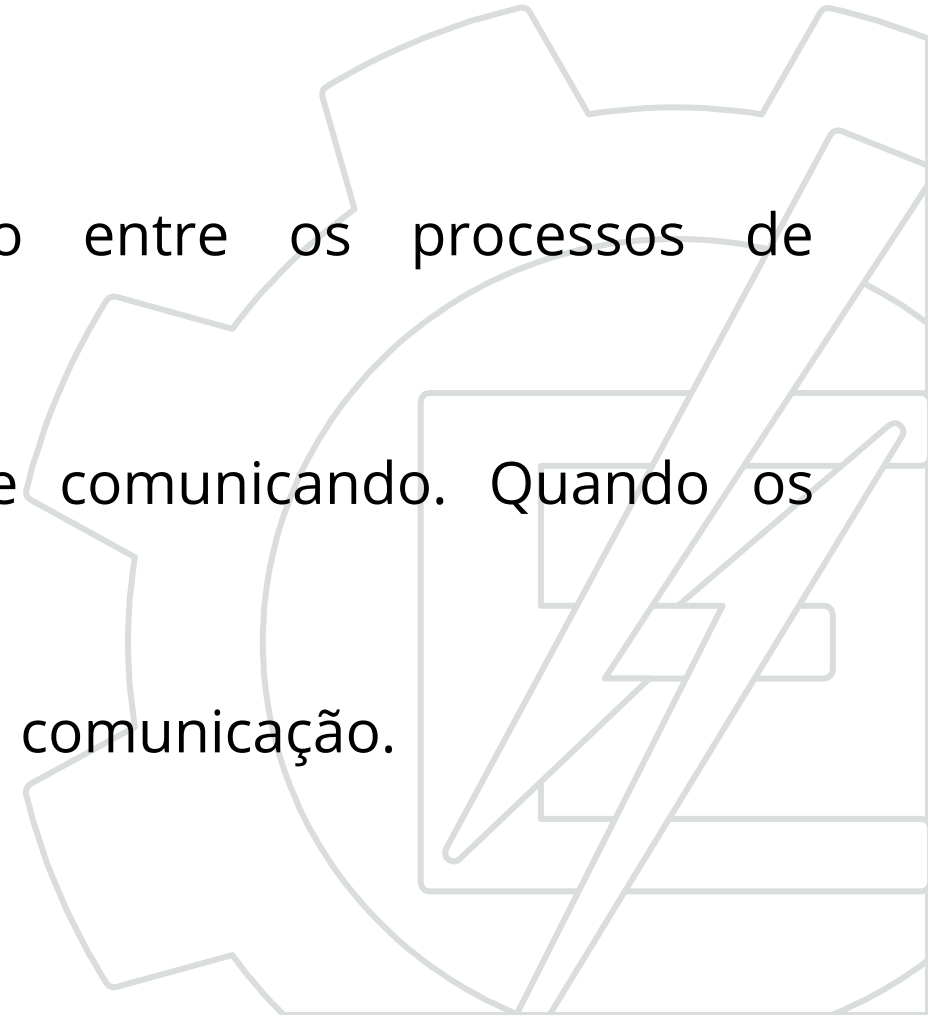
Pipes Comuns



Comunic. Cliente-Servidor

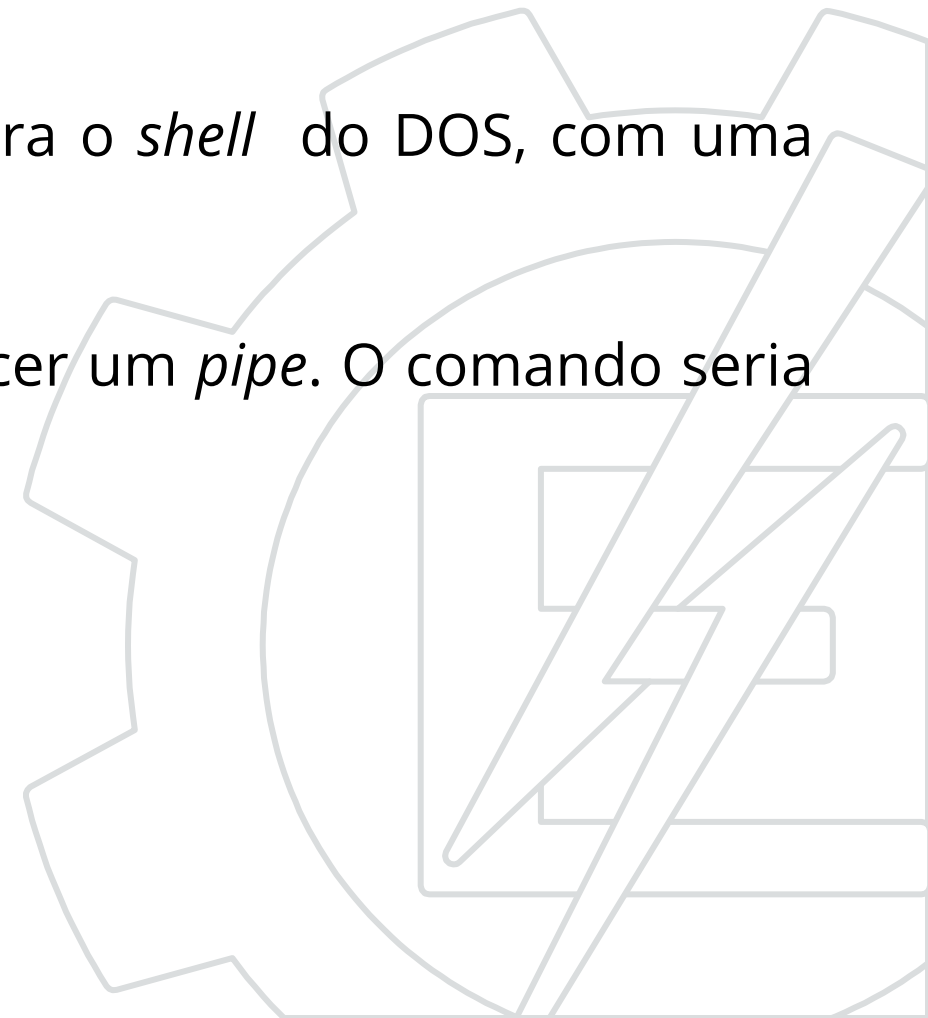
Pipes Nomeados (*Named pipes*)

- Os pipes nomeados são mais poderosos que os pipes comuns.
- A comunicação é **bidirecional**.
- Não é necessário o relacionamento pai-filho entre os processos de comunicação.
- Existe apenas enquanto os processos estão se comunicando. Quando os processos são finalizados o *pipe* deixa de existir.
- Vários processos podem usar o *pipe* nomeado para comunicação.
- Fornecido tanto em sistemas UNIX como Windows



- *Pipes* são utilizados frequentemente em ambientes de linhas de comando em UNIX.
- Por exemplo, o comando **ls** produz a lista de diretórios. Para uma lista muito grande, a saída deste comando pode ocupar muitas telas.
- O comando **more** pode administrar a saída apresentando uma página por vez.
- Criar um pipe entre estes comandos, que são executados em processos individuais, permite que a saída seja exibida em páginas, conforme o controle do usuário.
- O comando deve ser fornecido da seguinte forma: **ls | more**

- Neste cenário, o comando **ls** funciona como produtor e sua saída é consumida pelo comando **more**.
- Sistemas Windows fornecem o comando `more` para o *shell* do DOS, com uma funcionalidade similar.
- O DOS também utiliza o caractere **|** para estabelecer um *pipe*. O comando seria o seguinte: **dir | more**



Comunic. Cliente-Servidor

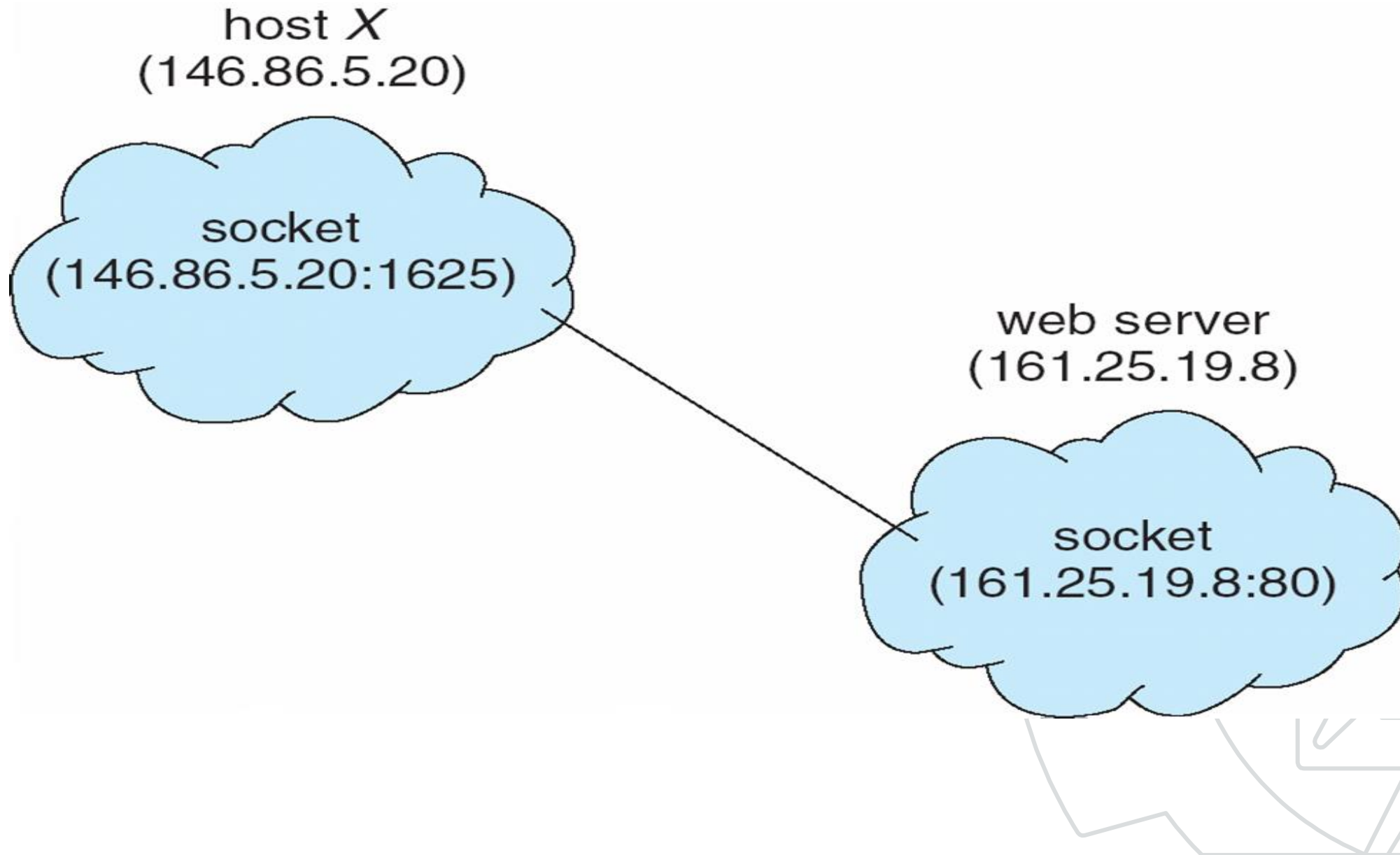
Sockets

- Um socket é definido como um *ponto final para comunicação* - *endpoint for communication*
- Concatenação do endereço IP e a porta
- O socket **161.25.19.8:1625** refere-se à porta **1625** no host **161.25.19.8**
- A comunicação consiste entre um par de *sockets*
- Java disponibiliza três tipos de *sockets*:
 - Conexão orientada (TCP);
 - Conexão não-orientada (UDP); e
 - *MulticastSocket* (uma subclasse de *DatagramSocket*).



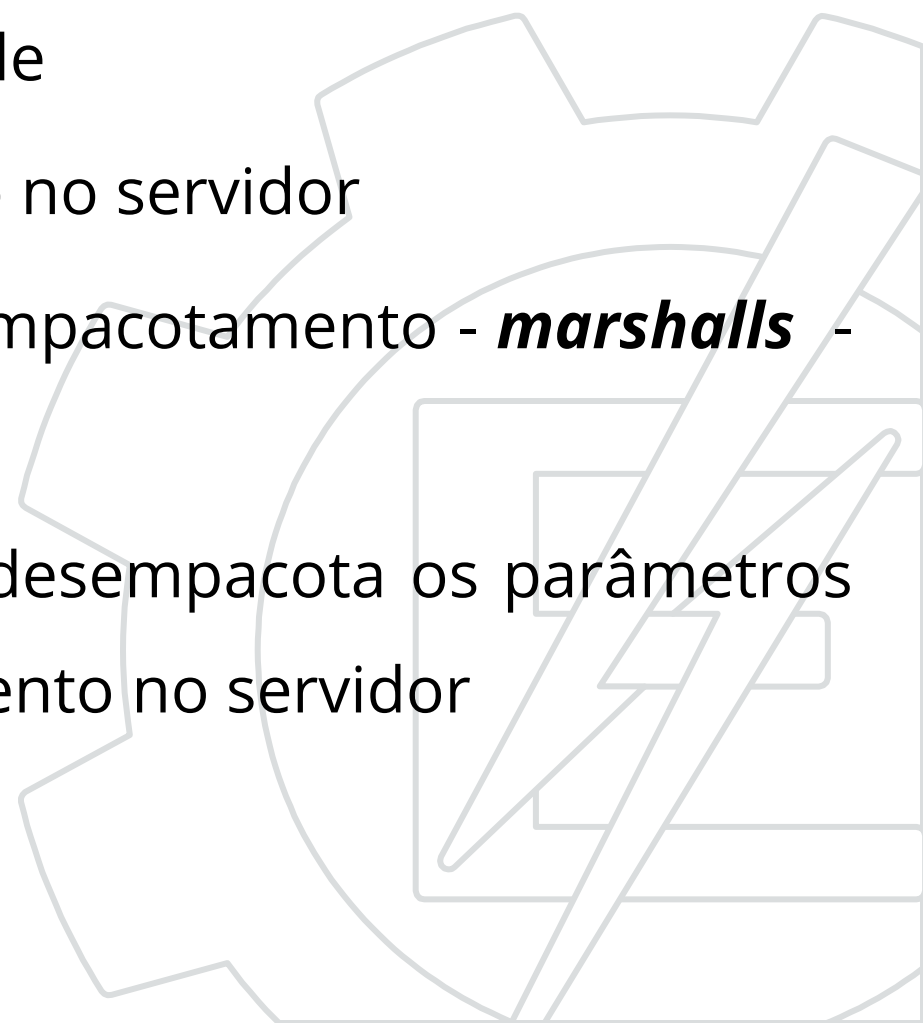
Comunic. Cliente-Servidor

Comunicação com Sockets



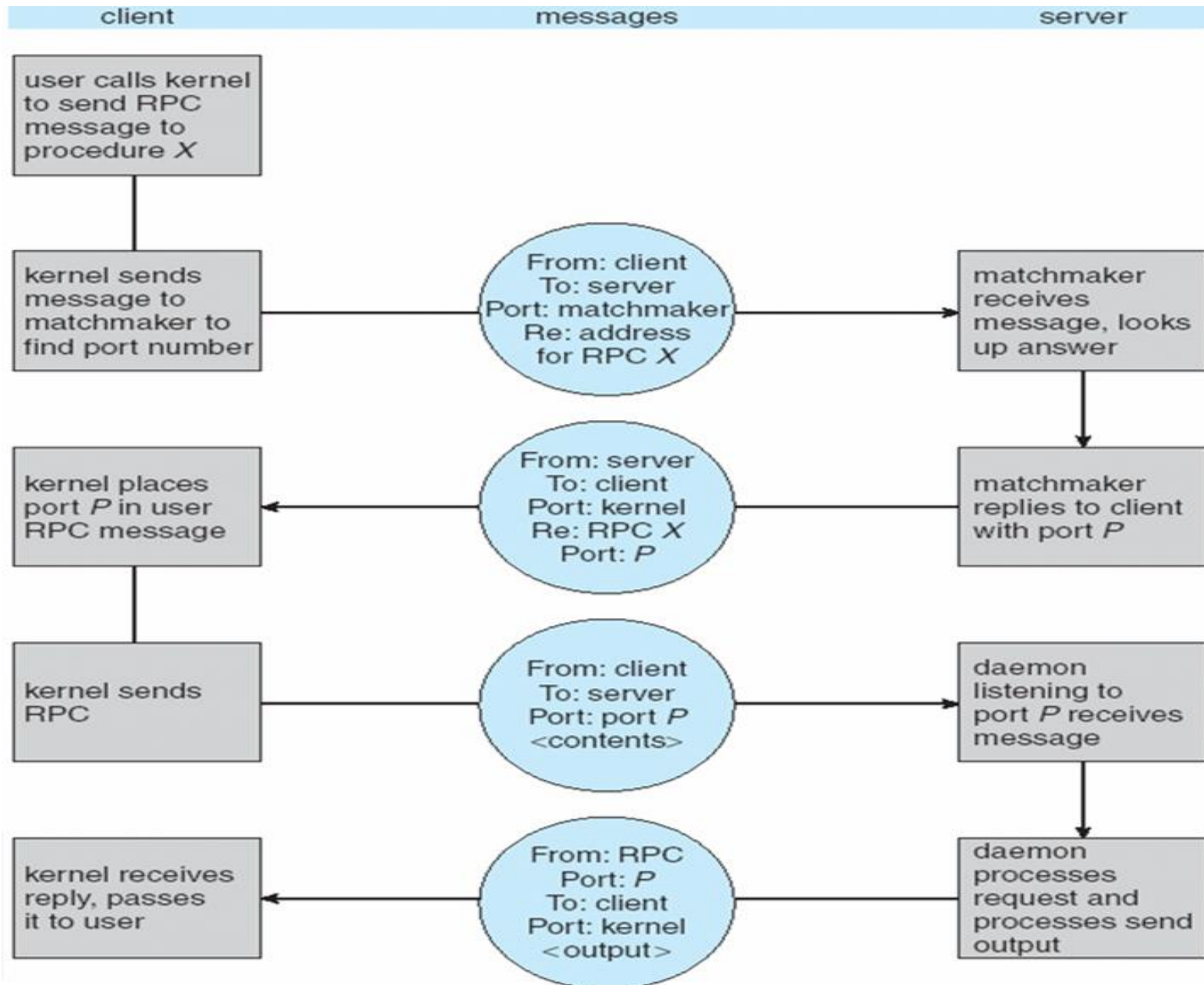
Comunic. Cliente-Servidor

Chamadas Remotas de Procedimento (RPC)

- Chamadas remotas de procedimento (RPC) abstraem as chamadas de procedimento para processos em sistemas em rede
 - **Stubs** – proxy do lado-cliente para o procedimento no servidor
 - O *stub* no lado-cliente localiza o servidor e faz o empacotamento - **marshalls** - dos parâmetros
 - O *stub* do lado servidor recebe esta mensagem, desempacota os parâmetros que estavam empacotados, e executa o procedimento no servidor
- 

Comunic. Cliente-Servidor

Execução de RPC



Bibliografia

- TANENBAUM, Andrew S; BOS, Herbert. Sistemas operacionais modernos. 4a ed. São Paulo: Pearson Education do Brasil, 2016.

Capítulo 2.

<https://plataforma.bvirtual.com.br/Acervo/Publicacao/1233>

- DEITEL, H.M; DEITEL, P.J; CHOFFNES,D.R. Sistemas Operacionais. 3a ed. São Paulo: Pearson Prentice Hall, 2005. **Capítulo 3.**

<https://plataforma.bvirtual.com.br/Acervo/Publicacao/315>



Sistemas Operacionais

Prof. Otávio Gomes

otavio.gomes@unifei.edu.br

