Comunicação entre Processos

Ricardo de la Rocha Ladeira

Introdução

A aula passada abordou o tema *Processos*, apresentou conceitos, tipificações, métricas, algoritmos de escalonamento e outras informações. A proposta desse texto é, partindo da base já obtida pela aula passada, abordar o tema *Comunicação entre Processos*.

Grande parte do texto é baseada no capítulo 3 (*Processos*) de [1], cuja leitura foi solicitada nos exercícios da aula anterior.

Comunicação entre Processos

Processos executando concorrentemente no sistema operacional podem ser *independentes* ou *cooperativos*. Um processo é dito *independente* se não compartilha dados com nenhum outro processo em execução no sistema. Um processo é dito *cooperativo* se pode afetar ou ser afetado por outros processos em execução no sistema. Todo processo que compartilha dados com outros processos é um processo cooperativo.

Exemplos:

- Um processo de reprodução de música e outro de envio de e-mail são independentes.
- Processos de um editor de texto, tais como o processo da interface de usuário e outro de manipulação de arquivos, são cooperativos, pois trabalham em conjunto para fornecer uma funcionalidade completa e satisfatória para o usuário. O processo da interface do usuário depende dos dados fornecidos pelo processo de manipulação de arquivos para exibir o conteúdo do texto. Ao mesmo tempo, o processo de manipulação de arquivos depende da interface do usuário para receber comandos do usuário, como abrir, editar e salvar arquivos.

Existem várias razões para fornecer um ambiente que permita a cooperação de processos:

- **Compartilhamento de informações**. Uma vez que várias aplicações podem estar interessadas na mesma informação (por exemplo, copiar e colar), devemos fornecer um ambiente que permita o acesso simultâneo a essas informações.

- **Aceleração de computação**. Se quisermos que uma determinada tarefa seja executada mais rapidamente, devemos dividi-la em subtarefas, cada uma das quais será executada em paralelo com as outras. Observe que essa aceleração só pode ser alcançada se o computador tiver vários núcleos de processamento.
- **Modularidade**. Podemos querer construir o sistema de forma modular, dividindo as funções do sistema em processos ou *threads* separadas.

Os processos cooperativos requerem um mecanismo de comunicação entre processos (*interprocess communication* — IPC) que lhes permitirá trocar dados - isto é, enviar dados e receber dados um do outro. Existem dois modelos fundamentais de comunicação entre processos: *memória compartilhada* e *troca de mensagens*. No modelo de memória compartilhada, é estabelecida uma região de memória que é compartilhada pelos processos cooperantes. Os processos podem então trocar informações lendo e gravando dados na região compartilhada. No modelo de troca de mensagens, a comunicação ocorre por meio de mensagens trocadas entre os processos cooperantes. Os dois modelos de comunicação são contrastados na Figura 1.

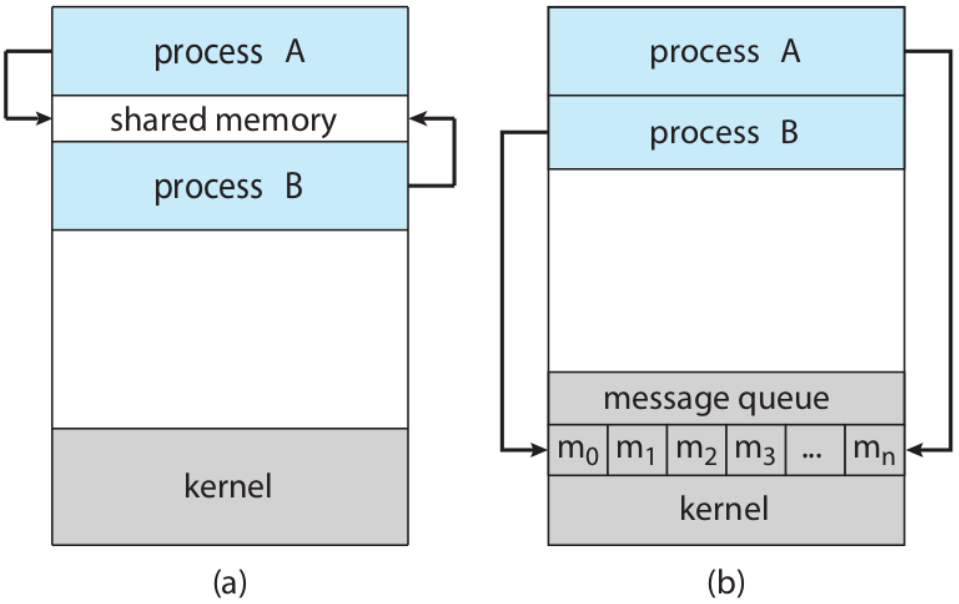

Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

Figura 1. Modelos-de-comunicação.png [1]. Disponível em https://drive.google.com/file/d/1z9QKn9Uar2wp300ziV8uppMg8gkhkuEK/view?usp=drive_link.

Ambos os modelos mencionados são comuns em sistemas operacionais, e muitos sistemas implementam ambos. A troca de mensagens é útil para troca de quantidades menores de dados, porque nenhum conflito precisa ser evitado. Troca de mensagens também é mais

fácil de implementar em um sistema distribuído do que a memória compartilhada (embora existam sistemas que fornecem memória compartilhada distribuída).

A memória compartilhada pode ser mais rápida do que a troca de mensagens, uma vez que os sistemas de troca de mensagens geralmente são implementados usando chamadas de sistema e, portanto, exigem a tarefa mais demorada de intervenção do *kernel*. Em sistemas de memória compartilhada, as chamadas de sistema são necessárias apenas para estabelecer regiões de memória compartilhadas. Uma vez estabelecida a memória compartilhada, todos os acessos são tratados como acessos de memória rotineiros e nenhuma assistência do *kernel* é necessária.

Sistemas de Memória Compartilhada

A comunicação entre processos usando memória compartilhada requer que os processos se comuniquem para estabelecer uma região de memória compartilhada. Normalmente, uma região de memória compartilhada reside no espaço de endereço do processo que cria o segmento de memória compartilhada. Outros processos que desejam se comunicar usando esse segmento de memória compartilhada devem anexá-lo ao seu espaço de endereço. Normalmente, o sistema operacional tenta impedir que um processo acesse a memória de outro processo.

A memória compartilhada requer que dois ou mais processos concordem em remover essa restrição. Eles podem então trocar informações lendo e escrevendo dados nas áreas compartilhadas. A forma dos dados e a localização são determinadas por esses processos e não estão sob o controle do sistema operacional. Os processos também são responsáveis por garantir que não estejam gravando no mesmo local simultaneamente.

Um exemplo comumente usado e solucionado com memória compartilhada (entre outras formas) é o *problema do produtor-consumidor*. Neste problema, um *processo produtor* produz uma informação que é consumida por um *processo consumidor*. O par produtor-consumidor pode ser representado em diversas aplicações por um servidor (produtor) e um cliente (consumidor), um compilador (produtor) e um montador (consumidor) entre outros.

Para permitir que os processos produtor e consumidor sejam executados simultaneamente, é necessário um *buffer*[1] que possa ser preenchido pelo produtor e esvaziado pelo consumidor. Esse *buffer* residirá em uma região de memória compartilhada pelos processos produtor e consumidor. Um produtor pode produzir um item enquanto o consumidor está consumindo outro item. O produtor e o consumidor devem estar sincronizados, para que o consumidor não tente consumir um item que ainda não foi produzido.

Além disso, o *buffer* pode ser

- **Ilimitado**, caso em que o produtor nunca espera (pode seguir produzindo sempre) e o consumidor espera quando o *buffer* está vazio (não há o que ser consumido); ou
- **Limitado**, caso em que o *buffer* tem tamanho fixo. Com esse tipo de *buffer* o produtor espera quando o *buffer* está cheio e o consumidor espera quando o buffer está vazio.

Código (incompleto) do processo *produtor* presente em [1]:

```
#define BUFFER SIZE 10

typedef struct {
(...)
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

(...)

item next_produced;
while (true) {
     /* produz um item em next_produced */
     while (((in + 1) % BUFFER_SIZE) == out); /* faz nada */
     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

[1] Um *buffer* é uma área de armazenamento de dados temporários, utilizada para acomodar informações que estão sendo transferidas de um lugar para outro.

Código (incompleto) do processo consumidor presente em [1]:

```
item next_consumed;
while (true) {
     while (in == out)
     ; /* não faz nada */
     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consome o item em next_consumed */
}
```

- O *buffer* compartilhado é implementado como uma matriz circular com dois ponteiros lógicos: `in` e `out`.
- A variável `in` aponta para a próxima posição livre no *buffer*.
- A variável `out` aponta para a primeira posição completa no *buffer*.
- O *buffer* está vazio quando `in` **==** `out`.
- O *buffer* está cheio quando `((in + 1) % BUFFER SIZE) == out`.
- O processo produtor tem uma variável local `next_produced` na qual o novo item a ser produzido é armazenado.
- O processo consumidor possui uma variável local `next_consumed` na qual o item a ser consumido é armazenado.
- Este esquema permite no máximo `BUFFER SIZE − 1` itens no *buffer* ao mesmo tempo.

Observe que pelos trechos de código acima não há qualquer tratamento de sincronização, o que ainda será visto na disciplina.

Por que é necessário sincronizar? Imagine o caso em que um processo produtor e um processo consumidor tentam acessar o *buffer* ao mesmo tempo. O que acontecerá?

Pergunta: que tipo de *buffer* foi usado no exemplo de [1]?

Sistemas de Troca de Mensagens

O modelo recém visto requer que os processos compartilhem uma região de memória e que o código para acessar e manipular a memória compartilhada seja escrito explicitamente

pelo programador. Outra maneira de obter o mesmo efeito é o sistema operacional fornecer os meios para que os processos cooperativos se comuniquem entre si por meio de um recurso de troca de mensagens.

A troca de mensagens fornece um mecanismo para permitir que os processos se comuniquem e sincronizem suas ações sem compartilhar o mesmo espaço de endereço. É particularmente útil em um ambiente distribuído, onde os processos de comunicação podem residir em diferentes computadores conectados por uma rede. Por exemplo, um programa de *chat* na Internet pode ser projetado para que os participantes se comuniquem trocando mensagens.

Um recurso de troca de mensagens fornece pelo menos duas operações: `send(msg)` e `receive(msg)`.

As mensagens enviadas por um processo podem ser de tamanho fixo ou variável. Se apenas mensagens de tamanho fixo puderem ser enviadas, a implementação no nível do sistema será direta. Essa restrição, entretanto, torna a tarefa de programação mais difícil. Por outro lado, as mensagens de tamanho variável exigem uma implementação mais complexa no nível do sistema, mas a tarefa de programação se torna mais simples

Se os processos P e Q quiserem se comunicar, eles devem enviar mensagens e receber mensagens um do outro: um link de comunicação deve existir entre eles. Esse link pode ser implementado de várias maneiras. Não estamos preocupados aqui com a implementação física do link (como memória compartilhada, barramento de hardware ou rede), mas sim com sua implementação lógica. Aqui estão vários métodos para implementar logicamente um link e as operações `send()` / `receive()`:

- Comunicação direta ou indireta
- Comunicação síncrona ou assíncrona
- *Buffering* automático ou explícito

Os processos que desejam se comunicar devem ter uma forma de se referir uns aos outros.

Na comunicação *direta*, cada processo que deseja se comunicar deve nomear explicitamente o destinatário ou remetente da comunicação. Nesse esquema, as primitivas `send()` e `receive()` são definidas como:

- **`send(P, msg)`**: envia uma mensagem para o processo `P`.

- **`receive(Q, msg)`**: recebe uma mensagem do processo `Q`.

Este esquema apresenta simetria no endereçamento; ou seja, tanto o processo emissor quanto o processo receptor devem nomear o outro para se comunicar. Uma variante desse esquema emprega assimetria no endereçamento. Aqui, apenas o remetente nomeia o destinatário; o destinatário não é obrigado a nomear o remetente. Neste outro esquema, as primitivas `send()` e `receive()` são definidas da seguinte forma:

- **`send(P, msg)`**: envia uma mensagem para o processo `P`.
- **`receive(id, msg)`**: recebe uma mensagem de qualquer processo. A variável `id` é definida como o nome do processo com o qual a comunicação ocorreu.

Na comunicação *indireta*, as mensagens são enviadas e recebidas de *caixas de correio* (*mailbox*) ou *portas*. Uma *caixa de correio* pode ser vista abstratamente como um objeto no qual as mensagens podem ser colocadas por processos e do qual as mensagens podem ser removidas. Cada caixa de correio tem uma identificação única, que pode ser um valor inteiro (exemplo: as filas de mensagens POSIX usam um valor inteiro para identificar uma caixa de correio).

Um processo pode se comunicar com outro processo por meio de várias caixas de correio diferentes, mas dois processos só podem se comunicar se tiverem uma caixa de correio compartilhada. As primitivas `send()` e `receive()` são definidas da seguinte forma:

- **`send(A, msg)`**: envia uma mensagem para a caixa de correio `A`.
- **`receive(A, msg)`**: recebe uma mensagem da caixa de correio `A`.

A comunicação entre os processos ocorre por meio de chamadas às primitivas `send()` e `receive()`. Existem diferentes opções para implementar cada primitiva. A troca de mensagens pode ser *bloqueante* ou não - também conhecida como *síncrona* e *assíncrona*.

- **Envio bloqueante**: O processo de envio é bloqueado até que a mensagem seja recebida pelo processo de recebimento ou pela caixa de correio.
- **Envio sem bloqueio**: O processo de envio envia a mensagem e retoma a operação.
- **Recepção bloqueante**: O receptor bloqueia até que uma mensagem esteja disponível.
- **Recepção sem bloqueio**: O receptor recupera uma mensagem válida ou nula.

A solução para o problema produtor-consumidor torna-se trivial quando usamos instruções de bloqueio `send()` e `receive()`. O produtor simplesmente invoca a chamada `send()`

de bloqueio e espera até que a mensagem seja entregue ao destinatário ou à caixa de correio. Da mesma forma, quando o consumidor invoca `receive()`, ele bloqueia até que uma mensagem esteja disponível. Isso é ilustrado nos códigos (incompletos) dos processos produtor e consumidor presentes em [1]:

Processo produtor:

```
message next_produced;
while (true) {
     /* produz um item em next_produced */
     send(next_produced);
}
```

Processo consumidor:

```
message next_consumed;
while (true) {
     receive(next_consumed);
     /* consome um item em next_consumed */
}
```

Seja a comunicação direta ou indireta, as mensagens trocadas pelos processos de comunicação residem em uma fila temporária. Basicamente, essas filas podem ser implementadas de três maneiras:

1. **Capacidade zero**: A fila tem um comprimento máximo de zero; assim, o link não pode ter nenhuma mensagem esperando nele. Nesse caso, o remetente deve bloquear até que o destinatário receba a mensagem.
2. **Capacidade limitada**: A fila tem comprimento finito *n*; assim, no máximo *n* mensagens podem residir nele. Se a fila não estiver cheia quando uma nova mensagem for enviada, a mensagem será colocada na fila (a mensagem será copiada ou um ponteiro para a mensagem será mantido) e o remetente poderá continuar a execução sem esperar. A capacidade do link é finita, no entanto. Se o link estiver cheio, o remetente deve bloquear até que haja espaço disponível na fila.
3. **Capacidade ilimitada**: O comprimento da fila é potencialmente infinito; assim, qualquer número de mensagens pode esperar nele. O remetente nunca bloqueia.

O caso de capacidade zero às vezes é referido como um sistema de mensagens sem *buffer*. Os outros casos são referidos como sistemas com *buffer* automático.

Pipes

Um *pipe* atua como um canal permitindo que dois processos se comuniquem. Os *pipes* foram um dos primeiros mecanismos IPC nos primeiros sistemas Unix. Eles normalmente fornecem uma das maneiras mais simples para os processos se comunicarem uns com os outros, embora também tenham algumas limitações.

Ao implementar um *pipe*, quatro questões devem ser consideradas:

1. O *pipe* permite comunicação unidirecional ou bidirecional?
2. Se a comunicação bidirecional for permitida, ela é *half duplex* (os dados podem trafegar apenas em uma direção por vez) ou *full duplex* (os dados podem trafegar em ambas as direções ao mesmo tempo)?
3. Deve existir um relacionamento (como pai-filho) entre os processos de comunicação?
4. Os *pipes* podem se comunicar em uma rede ou os processos de comunicação devem residir na mesma máquina?

Os pipes usam troca de mensagens ou memória compartilhada? Embora possa ser implementado com memória compartilhada, o padrão de comunicação é o de troca de mensagens, pois requerem a participação de ambos os lados.

Pipes comuns

Pipes comuns são uma forma simples de enxergar a comunicação entre produtor e consumidor. O produtor escreve em uma extremidade do *pipe* (a extremidade de gravação) e o consumidor lê na outra extremidade (a extremidade de leitura).

Exemplo:

```
> echo "Sou um texto" | wc -c
13
```

O exemplo acima é de um *pipe comum* (também conhecido como *pipe anônimo* ou *pipe sem nome*). Os *pipes* comuns são criados implicitamente pelo *shell* para permitir a comunicação entre os processos em uma única direção, normalmente da saída de um comando para a entrada de outro.

Neste exemplo específico, o comando `echo` é executado primeiro. Ele simplesmente imprime a expressão na saída padrão (`stdout`). Em seguida, o *pipe* (utilizado com o símbolo `|`) é usado para redirecionar a saída do comando `echo` para a entrada do próximo comando, que é `wc`. O comando `wc -c` conta o número de caracteres que recebe na entrada padrão (`stdin`) e imprime o resultado na saída padrão (`stdout`).

Portanto, o comando usa um *pipe* comum para redirecionar a saída do comando `echo` para a entrada do comando `wc`, permitindo que o segundo comando conte o número de caracteres da saída do primeiro.

Nos sistemas Unix, os *pipes* comuns são criados usando a chamada do sistema `pipe()`, como mostra a Figura 2 [1]. Eles são *unidirecionais*, ou seja, a comunicação ocorre em apenas uma direção: do processo ancestral para o processo filho. Além disso, podem ser utilizados apenas para comunicação entre processos relacionados (processo ancestral e processo filho ou processos criados por um mesmo ancestral).

Os *pipes* comuns têm tamanho limitado[2] e são representados por um descritor de arquivo do tipo *array*, onde `pipe_fd[0]` é usado para leitura e `pipe_fd[1]` é usado para escrita.

Em sistemas Unix, *pipes* comuns são construídos usando a função `pipe(int fd[])`. Essa função cria um *pipe* que é acessado por meio dos descritores de arquivo `fd[]`: `fd[0]` é o final de leitura do *pipe* e `fd[1]` é o final de gravação. O Unix trata um *pipe* como um tipo especial de arquivo. Assim, os *pipes* podem ser acessados usando chamadas de sistema `read()` e `write()` comuns.

Um *pipe* comum não pode ser acessado de fora do processo que o criou. Normalmente, um processo ancestral cria um canal e o usa para se comunicar com um processo filho que ele cria via `fork()`. Como um *pipe* é um tipo especial de arquivo, o filho herda o *pipe* de seu processo ancestral.

---

[2] Nas versões modernas de sistemas Linux, a capacidade padrão do *pipe* é de 16 páginas (64 KiB, se uma página do sistema tiver 4 KiB), mas ela pode ser alterada, conforme descreve o manual pipe(7) do Linux [2].

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

Figura 2. Windows-e-Unix-chamadas-de-sistema.png [1]. Disponível em https://drive.google.com/file/d/1bN8Sq2BRAZZw7ex9HmLYj-qQW8Fy37d4/view?usp=drive_link.

A Figura 3 ilustra o relacionamento dos descritores de arquivo no *array* `fd` com os processos ancestral e filho. Qualquer gravação do ancestral em sua extremidade de gravação do canal — `fd[1]` — pode ser lida pelo filho a partir de sua extremidade de leitura — `fd[0]` — do canal.

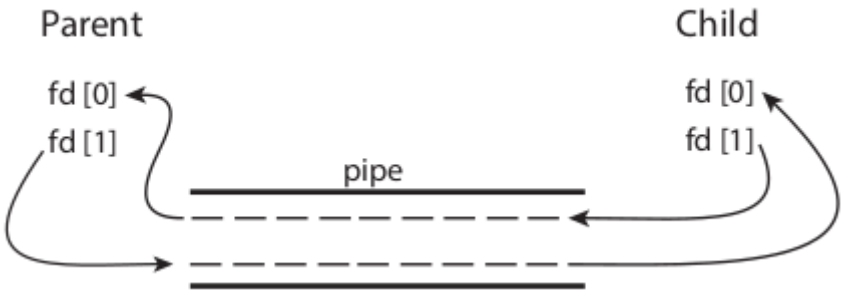

Figura 3. Descritor-de-arquivo-pipes-comuns.png [1]. Disponível em https://drive.google.com/file/d/1-_J5-fT2C8sdn8AU0_VqFFcwdUAql3Dn/view?usp=drive_link.

Um exemplo de criação dos processos e comunicação com *pipes* utilizando a linguagem C está no arquivo `pipe-comum.c`. O exemplo realiza o mesmo que o comando *shell*

```
> echo 'Sou um texto' | wc -c
```

O exemplo `ordinary-pipes.c`, adaptado de [1], traz um código em linguagem C em que um processo (ancestral) escreve a mensagem "`Saudações!`" para o *pipe* e o outro processo (filho) lê essa mensagem do *pipe*.

Pipes Nomeados

Os *pipes* nomeados, também conhecidos como FIFOs (*First In, First Out*), são criados usando a chamada do sistema `mkfifo()` em sistemas Unix ou `CreateNamedPipe()` em Windows. Eles são bidirecionais, permitindo comunicação em ambas as direções entre processos independentes.

Além disso, os *pipes* nomeados

- Podem ser usados para comunicação entre quaisquer processos, mesmo que não estejam relacionados.
- Representam um arquivo especial no sistema de arquivos e podem ser acessados por diferentes processos por meio do nome do *pipe*.
- Permitem comunicação entre processos em diferentes momentos, sendo os dados armazenados temporariamente no pipe até que o processo receptor os leia.
- Têm tamanho limitado, mas a quantidade de dados transmitidos depende do espaço disponível no *buffer* do *pipe*.

Exemplo:

1. Crie um *pipe* no terminal digitando `mkfifo meu_pipe`.
2. Crie um código que produz mensagens (`produtor-named-pipe.sh`) e outro que consome mensagens (`consumidor-named-pipe.sh`).
    2.1. No exemplo, o produtor apenas produz uma mensagem com a data atual a cada segundo e a envia para o *pipe*.
    2.2. O consumidor lê essa mensagem do *pipe* e a imprime.
3. Forneça permissão de execução para os códigos (na forma `chmod +x arquivo`).
4. Abra dois terminais e execute no primeiro o *script* do produtor e, ao mesmo tempo, execute o *script* do consumidor. Observe o resultado.
5. Idealmente, exclua o *pipe* com `rm meu_pipe`.

Comunicação em Sistemas Cliente-Servidor

Existem diversas estratégias de comunicação voltadas especificamente para o modelo cliente-servidor. Entre elas estão *sockets* e RPCs (*Remote Procedure Calls – Chamadas de Procedimentos Remotos*), abordados na sequência.

Sockets

Sockets são a base para muitas comunicações cliente-servidor. Eles fornecem uma API para a comunicação entre processos, permitindo que aplicativos em diferentes sistemas (ou mesmo na mesma máquina) troquem dados por meio de fluxos de *bytes*. Os *sockets* podem ser usados tanto para comunicação síncrona quanto assíncrona e estão disponíveis em muitas linguagens de programação, tais como C, C++, C#, Java, Ruby, PHP e Python.

- *Sockets* são considerados *endpoints* de comunicação, ou *nós* em uma rede de comunicação.
- É necessário um *socket* para cada processo que precisa se comunicar na rede.
- Um *socket* é identificado conjuntamente por um endereço IP e uma porta.
- O servidor aguarda a(s) solicitação(ões) recebida(s) do cliente ouvindo uma porta especificada.
- Depois que uma solicitação é recebida, o servidor aceita uma conexão do *socket* do cliente para concluir a conexão.
- Servidores que implementam serviços específicos escutam *portas conhecidas* (*well-known ports*[3]). Exemplos:

[3] *Well-known ports* são as portas menores que 1024.

    - um servidor SSH escuta a porta 22;
    - um servidor FTP escuta a porta 21;
    - um servidor web ou HTTP escuta a porta 80
- Quando um processo cliente inicia uma solicitação de conexão, uma porta é atribuída a ele por seu dispositivo *host*. Esta porta tem um número maior que 1023. Exemplo:
    - Se um cliente no *host* X (endereço IP 146.86.5.20) deseja estabelecer uma conexão com um servidor web (que está escutando na porta 80) no endereço 161.25.19.8, o *host* X pode ter atribuída a porta 1625. A conexão consistirá em um par de *sockets*: (146.86.5.20:1625) no *host* X e (161.25.19.8:80) no servidor web. Esta situação é ilustrada na Figura 4. Os pacotes que viajam entre os *hosts* são entregues ao processo apropriado com base no número da porta de destino.

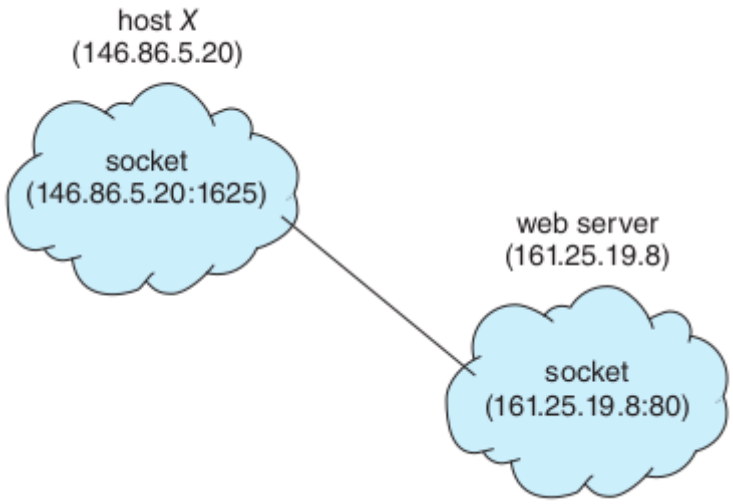

Figura 4. Comunicação-sockets.png [1]. Disponível em https://drive.google.com/file/d/1c7K6Tc-UjvWwvrxT2CCeddVDOF97cdqT/view?usp=sharing.

    - Se o mesmo *host* estabelecer outra conexão, seria atribuído a ele um número de porta maior que 1023 e não igual a 1625. Isso garante que todas as conexões consistam em um único par de *sockets*.
- A comunicação usando *sockets* – embora comum e eficiente – é considerada uma forma de comunicação de baixo nível entre processos distribuídos. Um motivo é que os *sockets* permitem que apenas um fluxo não estruturado de *bytes* seja trocado entre as *threads* de comunicação. É responsabilidade do aplicativo cliente ou servidor impor uma estrutura aos dados. Já o RPC (visto na sequência) é um método de comunicação de nível superior.

Passos para uso de sockets em C

1. **Criação**: Cria-se um *socket* usando a função `socket()`, que retorna um descritor de arquivo representando o *socket*. O tipo de *socket* (TCP ou UDP) e a família de endereços (IPv4 ou IPv6) são especificados durante a criação do *socket*.
2. **Configuração**: O *socket* pode precisar de algumas configurações, como reutilização de endereços ou portas. Isso é feito usando a função `setsockopt()`, que define as opções do *socket*.
3. **Conexão ou Bind**: Para o servidor, é necessário associar o *socket* a um endereço IP e número de porta usando a função `bind()`. Em seguida, o servidor fica pronto para escutar por conexões de clientes usando a função `listen()`. Para o cliente, é necessário conectar-se ao servidor usando a função `connect()`, fornecendo o endereço IP e número de porta do servidor.
4. **Comunicação**: Uma vez estabelecida a conexão, os processos cliente e servidor podem se comunicar usando as funções `send()` e `recv()`. O cliente envia dados ao servidor usando `send()`, e o servidor lê os dados recebidos usando `recv()`. O servidor também pode enviar dados de volta ao cliente usando a mesma abordagem.
5. **Encerramento**: Quando a comunicação é concluída, os *sockets* devem ser fechados para liberar o(s) recurso(s) alocado(s). Isso é feito com a função `close()`.

Exemplo de uso de *sockets* em C

Baixe os arquivos abaixo:

- `cliente.c`
- `cliente-2.c`
- `servidor.c`
- `servidor-2.c`

1. Compile os arquivos `cliente.c` e `servidor.c`:

```
> gcc -o cliente cliente.c
> gcc -o servidor servidor.c
```

2. Abra dois terminais e, no primeiro, execute o servidor:

```
> ./servidor
```

3. No segundo terminal, execute o cliente:

```
> ./cliente
```

O arquivo `servidor-2.c` faz a mesma ação que o `servidor.c`, mas fica ativo (em *loop*) esperando novas conexões.

4. Compile o arquivo `servidor-2.c` e execute-o em conjunto com o arquivo `cliente`.

Agora o servidor fica ativo e os clientes podem ser executados diversas vezes.
O arquivo `cliente-2.c` envia mensagens personalizadas, definidas pelo cliente, e fica ativo até que o usuário deseje sair (digitando "`sair`").

5. Compile o arquivo `cliente-2.c` e execute-o em conjunto com o executável gerado a partir do arquivo `servidor-2.c`.

Agora o servidor fica ativo e os clientes podem enviar diversas mensagens até que a mensagem informada seja "`sair`".

Porém, a solução ainda tem um problema: ao tentar executar dois ou mais clientes ao mesmo tempo, o servidor só atenderá o primeiro cliente. O segundo cliente fica esperando para poder conectar, e só conseguirá quando o primeiro cliente sair.

O problema ocorre porque o servidor está projetado para lidar com apenas um cliente por vez. Quando um cliente se conecta, o servidor entra em um *loop* para receber e enviar mensagens com esse cliente. Enquanto esse *loop* está em execução, o servidor não pode aceitar novas conexões de outros clientes.

Para permitir que o servidor aceite várias conexões simultaneamente e atenda a múltiplos clientes, é necessário usar *threads*, assunto que será visto futuramente.

RPC (*Remote Procedure Call – Chamadas de Procedimento Remoto*)

O RPC é uma técnica que permite que um programa em um *host* chame uma função ou procedimento em outro *host*, como se fosse uma chamada local. Isso torna a comunicação entre cliente e servidor mais transparente e abstrai a complexidade da comunicação de rede. É semelhante em muitos aspectos ao mecanismo IPC, mas como aqui os processos são executados em sistemas separados, usa-se um esquema de comunicação baseado em mensagens para fornecer serviço remoto.

Leia mais sobre RPC e como esta técnica é utilizada no Sistema Operacional Android em [1].

Resumo

Nesta aula, foi abordada a comunicação entre processos em sistemas operacionais, especialmente focando na comunicação cooperativa entre processos independentes e a importância de fornecer um ambiente que permita a cooperação de processos. Foram apresentados dois modelos fundamentais de comunicação entre processos: *memória compartilhada* e *troca de mensagens*. A memória compartilhada permite que os processos compartilhem uma região de memória e troquem informações escrevendo e lendo dados na região compartilhada. Já na troca de mensagens, os processos se comunicam por meio de mensagens trocadas entre eles, podendo ser implementada com comunicação direta ou indireta, síncrona ou assíncrona, e com ou sem *buffering* automático. Além disso, foi explorado o uso de *pipes* (canais) como um mecanismo de comunicação unidirecional ou bidirecional entre processos. Também foram explicadas técnicas de comunicação em sistemas cliente-servidor, como *sockets* e RPCs.

Exercícios

1) (IDECAN 2018 — CRF/SP — Analista de Suporte) No desenvolvimento de aplicações TCP/IP, as interfaces utilizadas são conhecidas como APIs *socket*. “O socket estabelece um conjunto de interfaces para uma aplicação acessar os protocolos do modelo de referência TCP/IP”. Uma API *socket* é constituída basicamente por constantes, estruturas e funções C, que são chamadas em uma sequência adequada, na definição de algoritmos genéricos para aplicações cliente-servidor. Entre as funções *socket*, uma delas atribui o número de porta e o endereço IP para um *socket* recém-criado pela função `socket()`. Assinale a alternativa referente a essa função.

a) `bind()`.
b) `accept()`.
c) `connect()`.
d) `recvfrom()`.

2) Crie exemplos de códigos (bash, C ou na linguagem de sua preferência) que utilizem *pipes* anônimos (comuns) e nomeados.

3) Crie exemplos de códigos (bash, C ou na linguagem de sua preferência) que utilizem *sockets*.

4) Em conjunto, conclua o resumo do capítulo 3 (*Processos*) de [1], a partir da seção 3.4 (*Comunicação entre Processos*). Escreva o resumo colaborativamente.

5) Escolha duas questões do capítulo 3 de [1] para resolver.

Gabarito

1) a
2) Livre.
3) Livre.
4) Livre.
5) Livre.

Referências

[1] SILBERSCHATZ, A., GALVIN, P. B., & GAGNE, G. (2018). Operating System Concepts (10ª ed.). Wiley.
[2] DIE.NET. pipe(7) - Linux man page. Disponível em: https://linux.die.net/man/7/pipe. Acesso em: 09 ago. 2024.