NÃO PODE FALTAR

PROCESSOS E COMUNICAÇÃO EM SISTEMAS DISTRIBUÍDOS

Marluce Rodrigues Pereira

O QUE SÃO PROCESSOS E *THREADS*?

Processo é definido como um programa em execução, e threads que são fluxos ou linhas de execução dentro de um processo (TANENBAUM; BOS, 2016).

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

PRATICAR PARA APRENDER

Caro aluno, nesta seção, você aprenderá conceitos importantes, os quais permitirão a construção de sistemas distribuídos eficientes. O avanço da tecnologia utilizada em processadores tem permitido a geração de processadores para dispositivos cada vez menores, como *smartwatches*. Os smartphones possuem uma grande capacidade de processamento, permitem o acesso à internet e a execução de jogos e de vários outros aplicativos que demandam uso de processamento. Os notebooks e desktops também possuem processadores com múltiplos núcleos de processamento. Os sistemas operacionais são adaptados e atualizados periodicamente, para adquirir a capacidade de gerenciar essas

alterações nos processadores. E quanto aos sistemas web, você sabe como devem ser implementados para utilizar essa capacidade de processamento? O processamento sequencial é suficiente para utilizar mais de um núcleo de processamento?

Esta seção nos leva a refletir sobre como os sistemas distribuídos estão inseridos neste contexto de avanço das tecnologias e como os sistemas web são desenvolvidos. Os conceitos de processos e *threads* são utilizados para permitir a implementação de sistemas que melhor utilizem o processamento dos núcleos de processamento, que podem estar em uma mesma máquina ou em máquinas remotas. Quando os processos são executados em máquinas remotas, a comunicação torna-se necessária para realizar a troca de informação entre eles. Essa comunicação pode ocorrer somente entre dois processados ou entre todos os processos de um grupo.

Sistemas web, como comércio eletrônico, correio eletrônico, máquinas de busca, entre outros, são exemplos de sistemas distribuídos. Eles possuem o lado do cliente (o usuário), que requisita um serviço, e o lado do servidor, que processa a requisição e devolve a resposta para o cliente. Mas, você sabe como ocorre essa "conversa" entre cliente e servidor em termos de sistema web? Será que cliente e servidor estão executando na mesma máquina ou em máquinas diferentes? O cliente é um processo, e o servidor, outro processo. Eles podem executar em uma mesma máquina ou em máquinas remotas.

Considerando o exemplo de um sistema de máquina de busca, por exemplo, o buscador Google, você pode se interessar pela busca na internet por páginas com ocorrências do termo "pulseira inteligente". Ao abrir um navegador na sua máquina (smartphone, notebook, desktop, servidor, entre outros), iniciará a execução de um processo. Ao digitar o termo e pedir para buscar, esse processo comunica-se via rede com um processo servidor, que estará em uma máquina remota e fará a busca de todos os endereços de páginas na internet onde o termo está ocorrendo. Em seguida, a resposta da busca retornará para o cliente (sua máquina). Para que todo o processamento seja realizado de forma rápida, várias máquinas são consultadas, as quais guardam informações e possuem processos executando para recuperar a informação.

A comunicação entre dois processos (cliente e servidor) via rede ocorre através de uma interface de software denominada *socket*, que é uma interface entre a camada de aplicação e a de transporte. Um processo envia uma mensagem, a qual é empurrada via socket pela rede e chega ao outro processo, que a lê e realiza alguma ação. Para saber o caminho a ser percorrido na rede, o *socket* leva ao endereço do processo destino. Após a troca de várias mensagens, os sistemas web conseguem atender o usuário, gerando a informação solicitada, permitindo concluir uma compra on-line, realizar uma conversa entre duas ou mais pessoas, entre outras ações.

Você foi contratado como *trainee* de uma empresa de transportes de mercadorias que atua em todo o Brasil. Essa empresa possui mais de 200 colaboradores, sendo 100 deles motoristas de caminhão. Foi atribuída a você a tarefa de auxiliar a equipe de desenvolvimento na elaboração de uma solução utilizando os conceitos de processos, *threads* e *sockets*, a fim de implementar um chat (aplicação de conversação) que permita que os motoristas troquem mensagens de texto. Neste contexto, elabore um relatório que responda a todas as questões a seguir e, após isso, crie uma apresentação da solução do problema:

a. Como você implementaria o chat no modelo cliente-servidor?

b. Para qual tipo de dispositivo deve ser implementada a parte da aplicação que tem o processo cliente? E para o processo servidor?

c. Como poderá ser implementada a comunicação realizada entre os colaboradores? Pense na linguagem de programação, na biblioteca ou API e como coordenar a troca das mensagens.

Como o servidor poderá ser implementado para atender a um número maior de conexões?

O entendimento sobre o funcionamento dos sistemas web, como os processos se comunicam e como implementar sistemas mais eficientes utilizando esses conhecimentos é de fundamental importância na formação do analista de sistemas. Portanto, vamos nos aprofundar um pouco mais nesses conceitos. Bons estudos!

CONCEITO-CHAVE

Caro aluno, os processadores atuais possuem mais de um núcleo de processamento e estão presentes em smartphones, notebooks, desktops, servidores, pulseiras inteligentes, entre outros. A utilização dos núcleos de processamento depende de como o sistema operacional escalona os programas para utilizá-los. Desta forma, a elaboração dos algoritmos precisa ser adaptada de forma que o sistema operacional consiga escalonar parte das computações para mais de um núcleo de processamento ao mesmo tempo e use os recursos de forma mais eficiente. Portanto, os conceitos de processos e *threads* são importantes para permitir a descrição de algoritmos que funcionem de forma eficiente.

PROCESSOS E *THREADS*

Em sistemas operacionais, um conceito muito importante é o de processo, o qual é definido como um programa em execução, e *threads* que são fluxos ou linhas de execução dentro de um processo (TANENBAUM; BOS, 2016). Como exemplo de processo, podemos considerar um editor de texto executando em um desktop ou um aplicativo de calendário executado em um smartphone.

Um processo possui um espaço de endereçamento de memória que armazena o código executável, os dados referentes ao programa, a pilha de execução, o valor do contador de programa, o valor do apontador de pilha, os valores dos demais registradores do hardware e outros recursos, como informações sobre processos filhos, arquivos abertos, alarmes pendentes, tratadores de sinais, entre outras informações necessárias para a execução do programa.

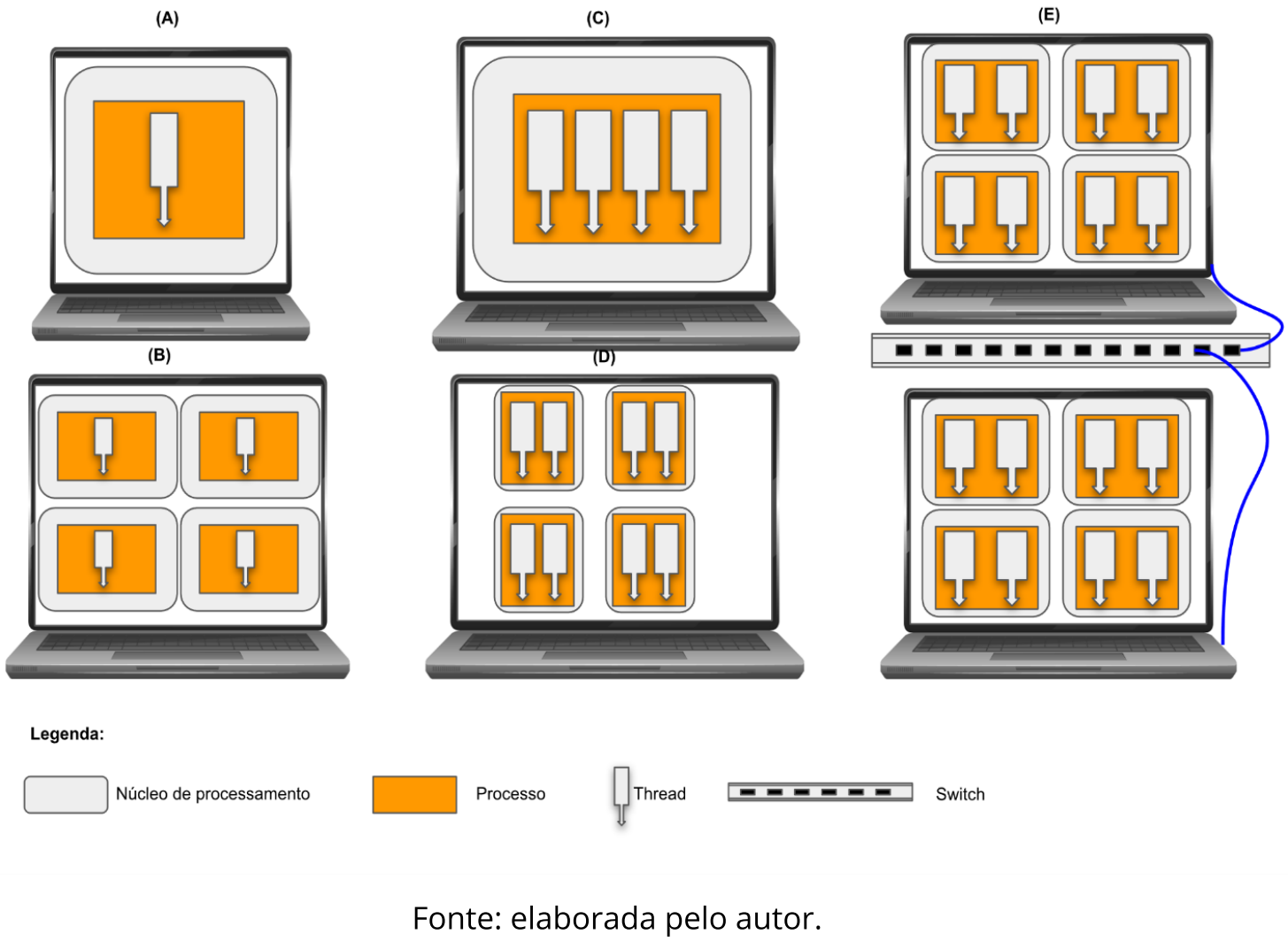
Muitas vezes, a construção de software usando um único processo pode não ser eficiente na utilização dos núcleos de processamento disponíveis nos processadores atuais, pois apenas um núcleo de processamento é utilizado. Por isso, existe a necessidade de construir aplicações paralelas e distribuídas, que permitam a execução simultânea de processos e/ou *threads*, deixando o software com execução mais rápida. O ato de gerenciar e executar múltiplas computações ao mesmo tempo, seja por troca de contexto no uso do núcleo de processamento ou pelo uso de diferentes núcleos de processamento é denominado concorrência. Por exemplo, uma máquina com processador contendo quatro núcleos de

processamento executando dez programas diferentes necessita que o sistema operacional escalone fatias de tempo para que todos os programas consigam utilizar os núcleos de processamento por meio de revezamento. Estes programas estarão executando de forma concorrente, dando ao usuário a impressão de que cada programa está utilizando os recursos do computador de forma dedicada. Quando há execução de computações simultaneamente em diferentes núcleos de processamento, tem-se paralelismo. Assim, processos e *threads* podem ser executados de forma concorrente e paralela. Por exemplo, um navegador com duas abas abertas, cada aba sendo um *thread* executado em um núcleo de processamento diferente, representa uma situação de execução paralela.

Um processo pode possuir um ou mais *threads*, permitindo que um programa execute mais de um trecho de código simultaneamente. Cada *thread* é um fluxo de controle sequencial isolado dentro de um programa capaz de utilizar CPU (*Central Processing Unit*). É composto de identificador (ID) de *thread*, contador de programa, conjunto de registradores e uma pilha de execução. Cada *thread* pode compartilhar com outros threads pertencentes ao mesmo processo a sua seção de código, seção de dados e outros recursos do sistema operacional, arquivos abertos e sinais.

A Figura 3.20 ilustra cinco possíveis situações em que existem processos e *threads* sendo executados em um notebook.

Figura 3.20 | (A) Notebook executando um processo com um *thread*; (B) Notebook executando quatro processos com um *thread* cada; (C) Notebook executando um processo com quatro *threads*; (D) Notebook executando quatro processos com dois *threads* por processo; (E) Sistema distribuído com dois notebooks executando vários processos e threads



Fonte: elaborada pelo autor.

Na imagem (A), há um processo com um único *thread* executado. O conjunto de instruções desse processo é executado de forma sequencial. Por exemplo, um programa que calcula a soma das notas inseridas pelo professor.

A imagem (B) apresenta quatro processos, cada um com um único *thread*, e os processos podem executar simultaneamente em núcleos de processamento, de acordo com escalonamento do sistema operacional.

Já na imagem (C), há um processo com quatro *threads*, e cada *thread* possui seu fluxo de controle, que pode ser executado concorrentemente a outros *threads*, pois há um único núcleo de processamento. Por exemplo, um programa que realiza a soma de um conjunto de dados, distribuindo os dados entre cada *thread* para que haja a soma parcial simultânea da porção de dados recebido e, por fim, realiza a soma total.

Vários processos com vários *threads* por processo são ilustrados na imagem (D), gerando a execução de vários fluxos de controle ao mesmo tempo e executando em núcleos de processamento diferentes. Por exemplo, vários programas, como navegador e editor de texto, , executando em um computador com quatro núcleos de processamento.

Por último, na imagem (E), tem-se um sistema distribuído composto por máquinas interligadas via um *switch*. Cada máquina pode ter mais de um processo executando mais de um *thread*. Os sistemas distribuídos são construídos sobre sistemas operacionais e utilizam processos e threads para realizar a execução de um programa. A comunicação entre os processos e/ou *threads* precisa ser realizada de forma especial, usando bibliotecas específicas. Na paralelização de um programa utilizando *threads* que executam em uma mesma máquina, podem ser utilizadas bibliotecas, como OpenMP (OPENMP, 2021) ou Pthreads (Posix Threads) (BARNEY, 2021). Os programas podem ser desenvolvidos para serem executados utilizando mais de um processo e em máquinas remotas. A comunicação entre os processos deve utilizar bibliotecas de troca de mensagens, como *Message Passing Interface* (MPI) (OPENMPI, 2021) e de *sockets*, assunto que será abordado posteriormente.

Você sabia que os programas que coloca para executar no seu notebook, no seu desktop ou mesmo no seu smartphone são processos? Mas, como pode-se criar um novo processo e finalizar (término de um processo)?

A **criação de processos** pode ocorrer em uma das seguintes situações:

- Pelo usuário, para iniciar a execução de um programa (via linha de comando ou duplo clique no mouse).
- Por funções específicas que independem do usuário (chamadas de serviços (*daemons*) ou executando em *background* sem intervenção do usuário), por exemplo, na recepção e no envio de e-mails, serviços de impressão, etc.
- Usando chamadas de sistema dentro de um programa.

Nos sistemas operacionais UNIX e Unix-like (como o Linux), os processos são criados utilizando a chamada de sistema fork(). No sistema operacional Windows, a criação ocorre pela chamada CreateProcess() (TANENBAUM; STEEN, 2007).

O **término de processos** pode ocorrer de algumas formas. Uma delas é quando o programa termina sua execução pela chamada *exit* no sistema operacional Unix-like, e ExitProcess no sistema operacional Windows; por algum erro ocorrido que não permitiu que o processo pudesse ser finalizado ou por um erro fatal; causado

por algum erro no programa (bug), como divisão por 0 (zero), referência à memória inexistente ou não pertencente ao processo e execução de uma instrução ilegal (TANENBAUM; STEEN, 2007).

Dentro de um processo, poderá existir tarefas que podem ser executadas de forma concorrente utilizando melhor os núcleos de processamento. A execução dessas tarefas pode ser realizada utilizando *threads*, que são escalonadas pelo sistema operacional para utilizarem os núcleos de processamento.

A decisão de quando criar *threads* dentro de um processo depende de quais características o problema que está sendo resolvido possui, ou seja, suas subtarefas. Assim, podem ser criadas quando há subtarefas que possuem potencial de ficarem bloqueadas por um longo tempo; usam muitos ciclos de CPU; devem responder a eventos assíncronos; tenham importância maior ou menor do que outras tarefas e podem ser executadas em paralelo com outras tarefas.

EXEMPLIFICANDO

A criação de processo nas linguagens C/C++ é realizada no Linux com a **chamada de sistema fork**, utilizando duas bibliotecas: *sys/types.h* e *unistd.h*. Essa chamada de sistema deve ser inserida no código implementado pelo usuário, que, ao ser compilado e executado, gera uma chamada ao sistema operacional, o qual cria um processo filho com um número identificador único. O processo que gerou a execução do programa é denominado **processo pai**, e o processo criado com a chamada fork() é denominado **processo filho**.

No momento da criação, o processo filho é uma cópia do pai e possui os mesmos atributos dele (variáveis, descritores de arquivos, dados, etc.). Após a criação do processo filho, ele é executado, e o que acontece em um processo não ocorre no outro, são processos distintos agora, cada um seguindo seu rumo, tornando-se possível mudar o valor de uma variável em um e isso não alterará o valor desta variável no outro processo.

A função fork() retorna um número identificador do processo (PID – *Process Identification*). No trecho de código executado pelo processo filho, PID tem valor igual a 0 (zero). Dentro do processo pai, PID tem valor igual ao

identificador do processo filho, retornado pelo sistema operacional. A função fork() retorna valor menor do que 0, caso ocorra algum erro na criação do processo.

A função getpid() retorna o valor do identificador do processo em execução.

A seguir, apresenta-se um código, no qual é criado um processo filho com a chamada fork():

```
#include <iostream>
#include <sys/types.h>
#include <unistd.h>
using namespace std;

int main(){
    pid_t pid;
    pid = fork();
    if (pid < 0){
        cerr << "Erro ao criar processo" << endl;
        exit(1);
    }
    if (pid ==0){
        // Código executado somente no processo filho
        cout << "Identificador do processo filho: " << getpid()
<< endl;
    } else {
        // Código executado somente no processo pai
        cout << "Identificador do processo pai: " << getpid() <<
endl;
    }
    return 0;
}
```


PROCESSOS CLIENTE-SERVIDOR

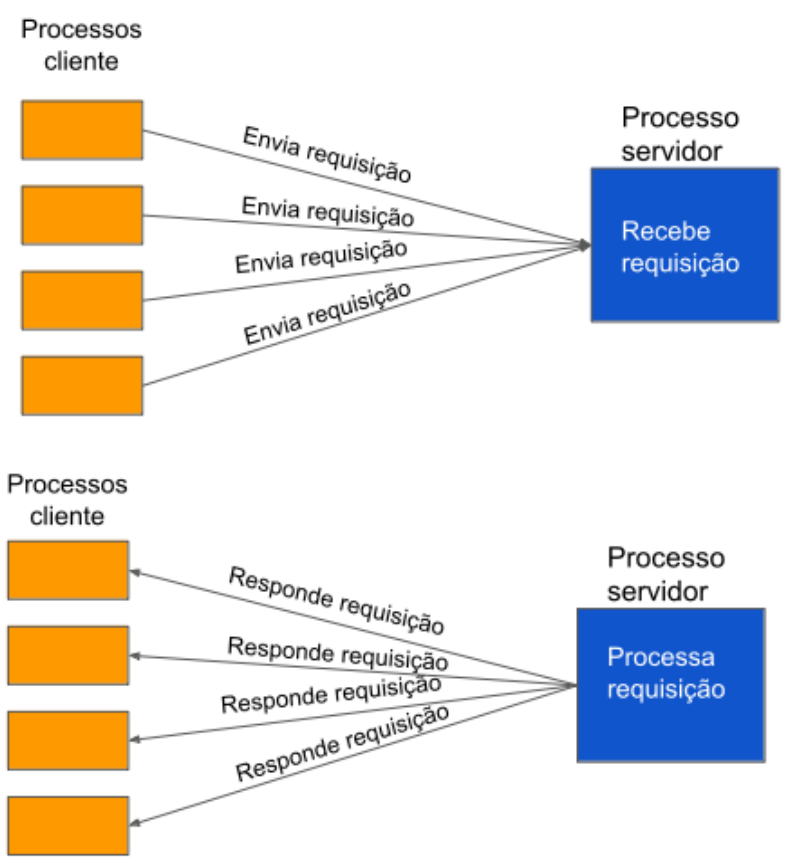
O modelo cliente-servidor é o modelo de computação distribuída em que um programa é dividido em duas partes: servidor e cliente (COULOURIS *et al.*, 2013.; KUROSE; ROSS, 2013). O servidor é executado em um processo, e o cliente, em outro processo diferente, que pode estar na mesma máquina ou em uma máquina remota. O servidor tem acesso direto ao recurso de hardware ou software que o cliente quer usar. Na maioria dos casos, o cliente está localizado em diferentes máquinas clientes.

Ver anotações

Tipicamente, há um relacionamento muitos-para-um entre o servidor e os clientes, isto é, há, geralmente, um servidor atendendo às requisições de muitos clientes. O servidor é o mediador do acesso a grandes bases de dados, a um recurso de hardware caro ou a uma coleção importante de aplicações. O cliente faz requisições para ter acesso a dados e obter o resultado de um cálculo e outros tipos de processamento. A Figura 3.21 ilustra o funcionamento de um sistema cliente-servidor, no qual há um conjunto de processos cliente enviando requisições para o processo servidor, que é o responsável por realizar as computações necessárias e responder aos clientes.

Supondo que o servidor implementa a soma de dois valores, um cliente envia dois valores, e o servidor realiza a soma e retorna o valor resultante.

Figura 3.21 | Modelo cliente-servidor



Fonte: elaborada pela autora.

Outro exemplo é uma máquina de busca no modelo cliente-servidor, que é usada para localizar informação na internet ou em intranets corporativas e organizacionais. Neste caso, o processo cliente é executado no navegador para obter a palavra-chave ou frase sobre o tema que o usuário está interessado. O processo cliente envia a requisição para o servidor. O processo servidor, por sua vez, realiza uma ampla busca pela palavra-chave ou frase do usuário, pois este tem acesso direto à informação ou a outros servidores que têm acesso à informação. Posteriormente, o servidor retorna as informações ao cliente. Embora os processos cliente e servidor sejam programas separados em computadores diferentes, eles trabalham em uma única aplicação. Quando o processo pode funcionar como cliente e servidor ao mesmo tempo, temos um sistema Peer-to-Peer (P2P).

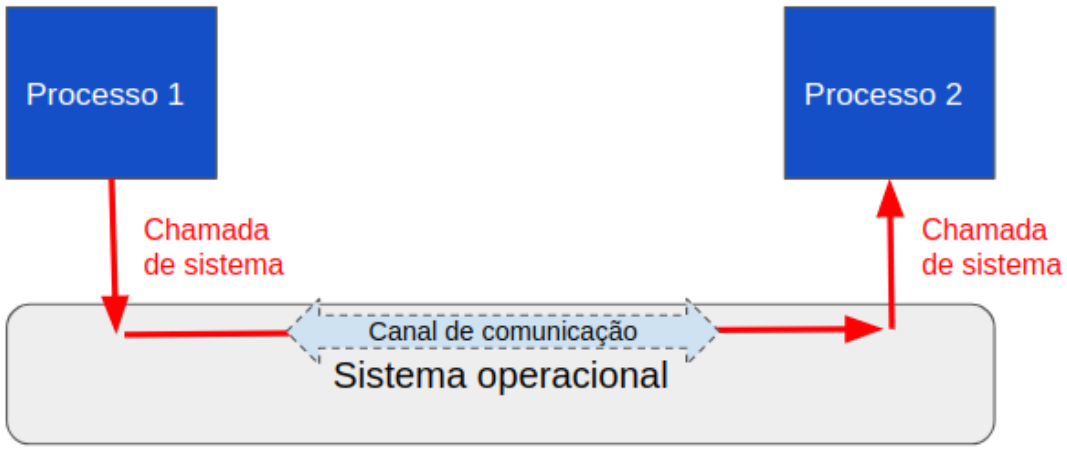
> **ASSIMILE**
>
> No contexto de comunicação entre um par de processos, o processo que inicia a comunicação é o cliente, e o processo que espera pela requisição é o servidor. Na web, o navegador é o processo cliente, e o servidor web é o processo servidor (KUROSE; ROSS, 2013).

COMUNICAÇÃO ENTRE PROCESSOS E EM GRUPO

Após a criação dos processos, eles são executados de forma independente e possuem regiões de memória independentes. Entretanto, há aplicações que necessitam trocar dados (comunicação) entre processos ou entre *threads*. Para haver comunicação entre processos, é necessária a utilização de mecanismos específicos. O sistema operacional implementa “canais” de comunicação entre processos em um mesmo computador (tais canais podem ser implícitos ou explícitos) ou em computadores diferentes.

A Figura 3.22 ilustra a comunicação entre dois processos, que estão em uma mesma máquina, gerenciados pelo mesmo sistema operacional. A comunicação entre processos através de chamadas de sistema pode ocorrer com o uso de *pipes* ou alocação de memória compartilhada.

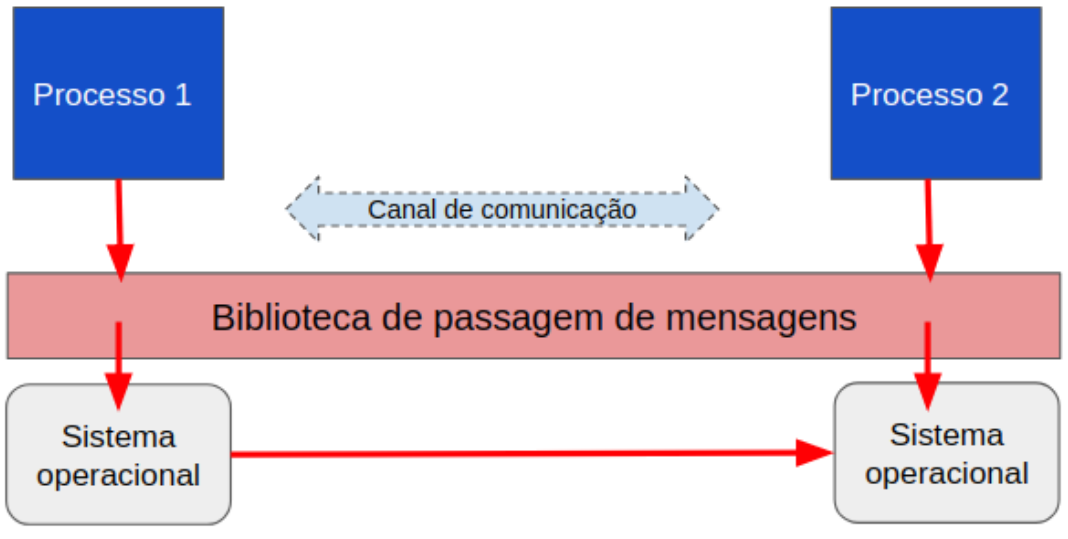
Figura 3.22 | Canal de comunicação entre dois processos



Fonte: elaborada pela autora.

A Figura 3.23 ilustra o processo 1 na máquina 1 se comunicando com o processo 2 na máquina 2 através de chamadas a funções de uma biblioteca de passagem de mensagens, como MPI.

Figura 3.23 | Processos em máquinas diferentes se comunicando por passagem de mensagens

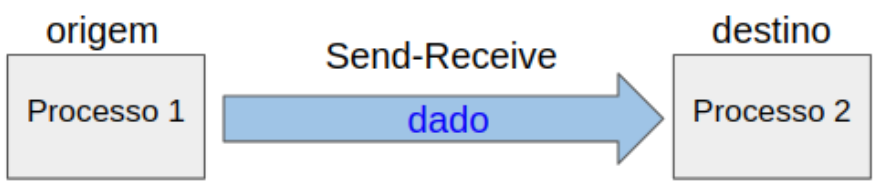


Fonte: elaborada pela autora.

O modelo de passagem de mensagens entre processos possui a operação básica *send-receive*. Outras operações de comunicação, como *broadcast*, *reduction*, *scatter* e *gather*, são derivadas da básica.

A operação *send-receive* ocorre somente entre dois processos (Figura 3.24). Um processo envia o dado e o outro processo recebe o dado.

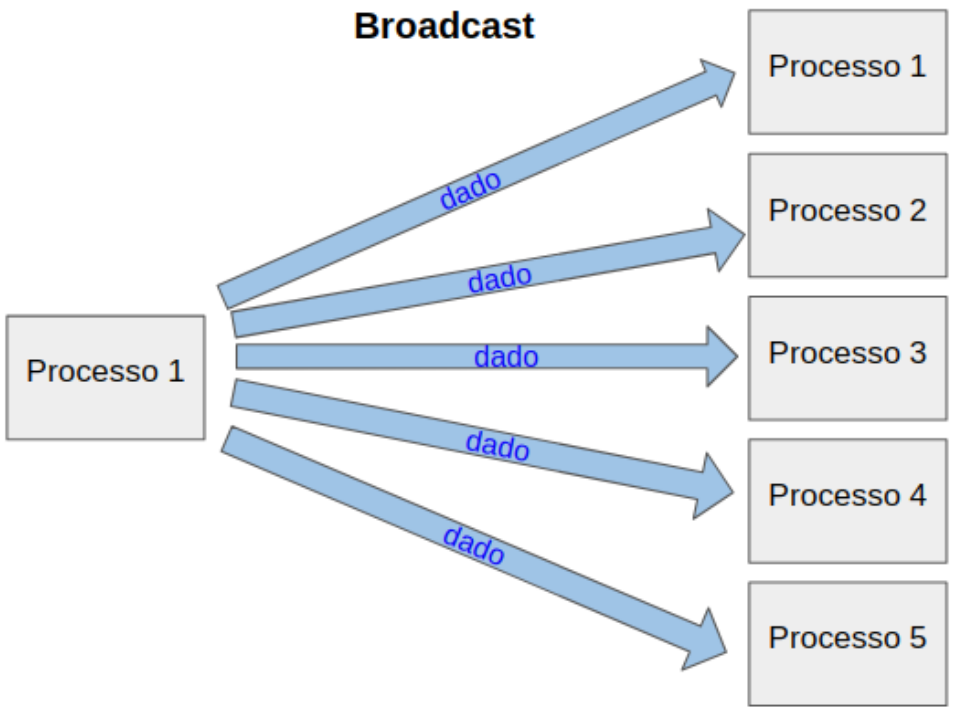
Figura 3.24 | Operação *send-receive*



Fonte: elaborada pela autora.

A operação de *broadcast* é uma operação entre um grupo de processos conhecidos. Consiste no envio de um dado presente em um processo para os demais processos do mesmo grupo.

Figura 3.25 | Operação *broadcast*



Fonte: elaborada pela autora.

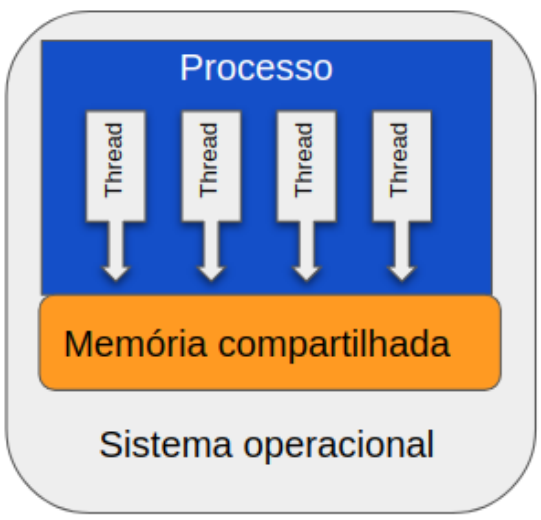
Na Figura 3.25, o processo 1 envia o dado para os demais processos do mesmo grupo de processos. Ao final da comunicação, todos os processos do grupo conhecem o dado.

Para implementar aplicações que utilizem a troca de mensagens entre processos, existem bibliotecas, como *sockets*, sobre as quais trataremos posteriormente, e MPI (*Message Passing Interface*) (OPENMPI, 2021; TANENBAUM; STEEN, 2007).

MPI é um padrão de comunicação que permite a troca de mensagens entre processos pertencentes a um grupo de processos criado no início da execução do programa. Cada processo recebe um identificador único dentro deste grupo, facilitando a comunicação entre pares de processos ou grupo de processos. É possível identificar qual é o processo emitente e o receptor da mensagem. Dentro de um mesmo processo, é possível criar *threads*, que podem se comunicar através de troca de mensagens ou acesso à memória compartilhada.

Na Figura 3.26, há quatro *threads* executando em um mesmo processo. Esses *threads* compartilham memória e trocam informação através do acesso à memória compartilhada. Para gerenciar o acesso à memória compartilhada, são utilizadas bibliotecas, como OpenMP (OPENMP, 2021) e PosixThreads (BARNEY, 2021).

Figura 3.26 | Threads de um mesmo processo acessando a memória compartilhada



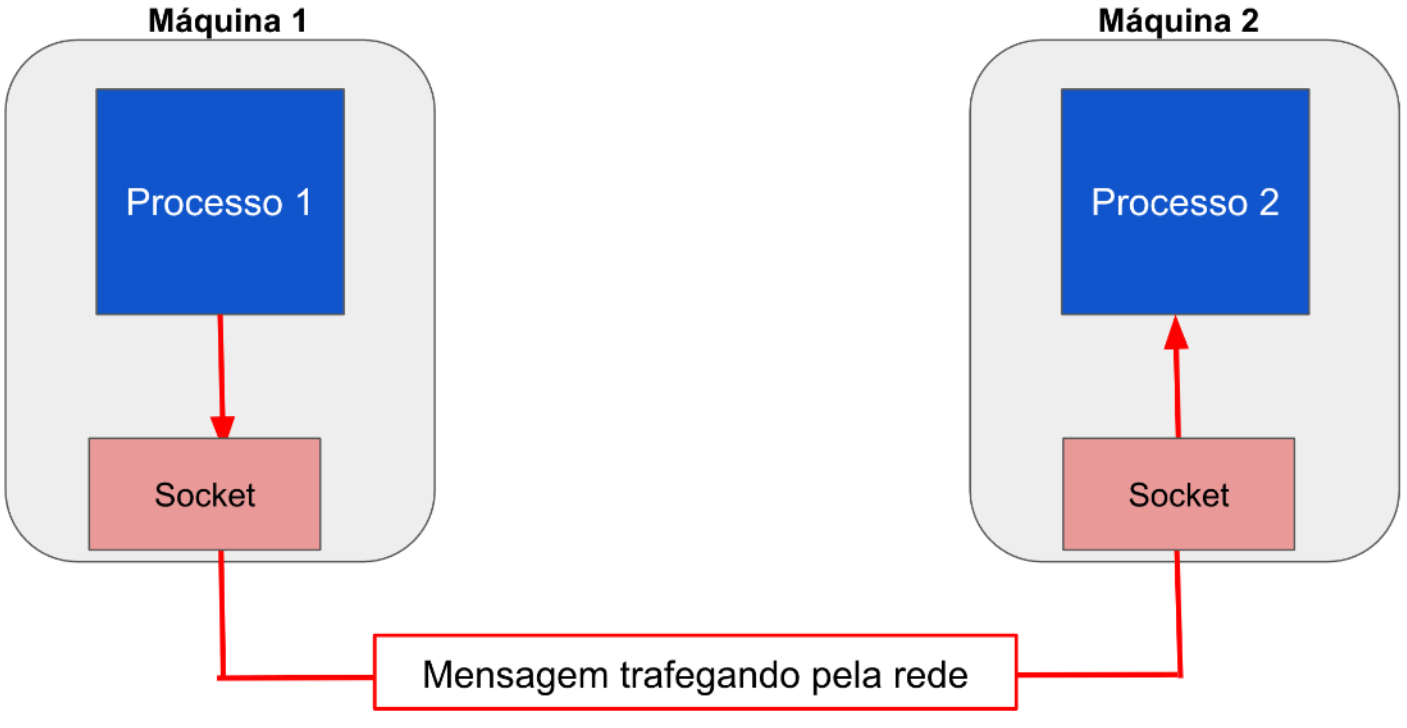
Fonte: elaborada pela autora.

CONCEITOS DE *SOCKETS*

A padronização da interface da camada de transporte permitiu que os protocolos de troca de mensagens sejam utilizados pelos programadores no desenvolvimento de aplicações que envolvam mais de uma máquina. Uma mensagem enviada de um processo para outro deve passar pela rede, e uma forma de realizar a implementação dessa comunicação é utilizando a interface de software denominada *socket*. É uma interface entre a camada de aplicação e o protocolo da camada de transporte.

> “Socket é um terminal de comunicação para o qual uma aplicação pode escrever dados que devem ser enviados pela rede subjacente e do qual pode ler dados que chegam.” (TANENBAUM; STEEN, 2007, p. 85).

Figura 3.27 | Comunicação entre dois processos via *socket*



Fonte: elaborada pela autora.

A Figura 3.27 ilustra a comunicação utilizando *socket* entre dois processos de máquinas diferentes. A seta vermelha indica a origem e o destino da mensagem. Para que o Processo 2 receba uma mensagem do Processo 1, é necessário que a mensagem trafegue pela rede. Como o Processo 1 é o processo emissor, é necessário que, quando a mensagem for enviada por ele, contenha o dado, o endereço e o identificador do processo receptor da mensagem. Na camada de rede da internet, o endereço é fornecido pelo IP (*Internet Protocol*), que identifica a máquina destino de forma única. O identificador do processo receptor é um número de porta. Aplicações mais utilizadas possuem um número de porta padrão, como é o caso do servidor web, que é identificado pela porta 80, e o servidor de e-mail, pela porta 25.

A interface *socket* pode utilizar os protocolos de transporte TCP (*Transmission Control Protocol*) ou UDP (*User Datagrama Protocol*). *Sockets* utilizando TCP/IP possuem as primitivas sintetizadas no Quadro 3.1.

Quadro 3.1 | Primitivas da interface *socket* para TCP/IP

Primitiva	**Significado**
socket	Criar um novo terminal de comunicação.

Primitiva	Significado
bind	Anexar um endereço local a um *socket*.
listen	Anunciar a disposição de aceitar conexões.
accept	Bloquear o chamador até chegar a uma requisição de comunicação.
connect	Tentar estabelecer uma conexão ativamente.
send	Enviar alguns dados pela conexão.
receive	Receber alguns dados pela conexão.
close	Encerrar a conexão.

Fonte: adaptado de Tanenbaum e Steen (2007, p. 85).

- Uma chamada *socket* retorna um descritor de arquivo que será utilizado em outras chamadas.
- A chamada *bind* designa um endereço para o *socket* no servidor.
- A chamada *listen* aloca espaço para a fila de chamadas recebidas ao mesmo tempo.
- *Accept* é utilizada para bloquear a espera por uma conexão de entrada.
- *Connect* bloqueia o responsável pela chamada e inicia o processo de conexão.
- *Send* e *receive* são usadas para transmitir e receber dados em uma conexão *full-duplex*.
- Quando ambos os lados executarem *close*, a conexão será encerrada.

A implementação de programas no modelo cliente-servidor envolve a implementação de um processo cliente e um processo servidor, que utilizam *sockets* para comunicação. Os serviços de *ssh* (*secure shell*) e ftp (*file transfer protocol*) são exemplos de implementações que utilizam sockets. O programa ssh é usado para realizar conexão entre duas máquinas e possui um processo cliente denominado ssh e um processo servidor denominado sshd (ssh deamon). O

programa ftp é usado para transferência de arquivo entre duas máquinas e possui um processo cliente denominado ftp e um processo servidor denominado ftpd (*ftp deamon*).

Os *sockets* TCP são orientados à conexão e têm um canal exclusivo de comunicação entre cliente e servidor. Uma aplicação que faz transações bancárias é um exemplo de aplicação que pode usar sockets TCP.

Várias linguagens de programação, como Java, PHP e Python, fornecem *socket* ao desenvolvedor API, para facilitar o desenvolvimento das aplicações.

REFLITA

Atualmente, existem muitos jogos que você pode jogar com seus amigos, por exemplo, Minecraft, League of Legends e Among Us. Você já pesquisou como são implementados esses jogos? Como um jogador é capaz de visualizar os movimentos do outro jogador? É necessária a criação de processos e *threads*? Há comunicação entre processos ou compartilhamento de dados entre *threads*?

PESQUISE MAIS

O conteúdo desta seção pode ser complementado pela leitura de livros sobre sistemas distribuídos e redes de computadores disponíveis na Biblioteca Virtual, indicados a seguir:

KUROSE, J. F.; ROSS, K. W. **Redes de computadores e a internet**: uma abordagem top-down. 6. ed. São Paulo: Pearson, 2013.

TANENBAUM, A. S.; STEEN, M. V. **Sistemas Distribuídos**: princípios e paradigmas. 2. ed. São Paulo: Pearson Prentice Hall, 2007.

TANENBAUM, A. S.; WETHERALL, D. **Redes de Computadores**. 5. ed. São Paulo: Pearson Prentice Hall, 2011.

Nesta seção, você aprendeu sobre a utilização de processos e *threads* no contexto de sistemas distribuídos, como ocorre a comunicação entre processos, processos cliente e servidor e *sockets*. Agora, você é capaz de entender melhor as aplicações

existentes, identificar quais são cliente-servidor e aplicar estes conceitos em aplicações em desenvolvimento de forma a ter melhor desempenho. Bons estudos!

FAÇA VALER A PENA

Questão 1

De uma forma geral, os sistemas operacionais implementam mecanismos para garantir a independência entre processos, com suporte de hardware para proteção de memória. No entanto, quando há necessidade de troca de informações entre processos, existem mecanismos apropriados para isso.

Assinale a alternativa correta:

a. A comunicação entre processos em computadores diferentes pode ser feita utilizando sockets.

b. A comunicação entre processos em um mesmo computador pode ser feita através de acesso direto à memória, sem uso de bibliotecas específicas.

c. A comunicação entre threads de um mesmo processo não pode ocorrer, pois um thread interrompe a execução do outro.

d. A comunicação entre *threads* em máquinas diferentes pode ocorrer através do uso de uma biblioteca específica para troca de mensagens entre *threads*.

e. A troca de informação entre processos pode gerar resultados incorretos, pois altera a região de memória do processo remoto.

Questão 2

O processamento digital de imagens demanda grande capacidade de processamento. Por isso, os softwares precisam ser implementados de forma que haja distribuição do processamento em vários núcleos de processamento em uma mesma máquina ou em máquinas remotas em uma mesma rede de computadores.

Assinale a alternativa correta:

a. Para que o processamento ocorra em uma máquina remota, é necessário que seja criado um *thread* remoto e esse se comunique com os demais por memória compartilhada.

b. A distribuição do processamento em núcleos de processamento de uma mesma máquina pode ser feita pela criação de *threads* que se comunicam por memória compartilhada.

c. Para que o processamento ocorra em uma mesma máquina, é necessário que seja utilizado somente um processo e um único núcleo de processamento.

d. A distribuição do processamento entre máquinas remotas é feita pelo sistema operacional.

e. A distribuição do processamento entre máquinas remotas pode gerar resultados incorretos, pois altera a região de memória do processo remoto.

Questão 3

Mapas são amplamente utilizados em várias aplicações, e o compartilhamento de coordenada/localização em tempo real é de fundamental importância. A atualização da coordenada pode ser realizada usando *socket*.

Analise as afirmativas sobre *sockets* a seguir:

I. *Socket* é uma interface entre a camada física e a de rede.

II. Cada ponto final na interface *socket* para TCP/IP é identificado por uma tupla contendo porta TCP e endereço IP.

III. Através da conexão entre *sockets* é possível conectar pontos finais e fazer operações de E/S.

IV. *Sockets* não podem ser utilizados para realizar a comunicação entre processos cliente e servidor.

É correto o que se afirma em:

a. I e IV, apenas.

b. II e IV, apenas.

c. I e II, apenas.

d. III e IV, apenas.

e. II e III, apenas.

Ver anotações 0

REFERÊNCIAS

BARNEY, B. POSIX Threads Programming. **Computing**, 2021. Disponível em: https://computing.llnl.gov/tutorials/pthreads/. Acesso em: 13 fev. 2021.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T.; BLAIR, G. **Sistemas Distribuídos**: conceitos e projetos. Porto Alegre, RS: Bookman, 2013.

KUROSE, J. F.; ROSS, K. W. **Redes de computadores e a internet**: uma abordagem top-down. 6. ed. São Paulo: Pearson, 2013.

OPENMP. Disponível em: https://www.openmp.org. Acesso em: 13 fev. 2021.

OPENMPI. Disponível em: https://www.open-mpi.org/. Acesso em: 13 fev. 2021.

TANENBAUM, A. S.; BOS, H. **Modern operating systems**. Pearson, 2015.

TANENBAUM, A. S.; STEEN, M. V. **Sistemas Distribuídos**: princípios e paradigmas. 2. ed. São Paulo: Pearson Prentice Hall, 2007.