INTRODUÇÃO AO SOFTWARE

Prof. Antônio Eugênio

Sumário

1. Introdução

O conceito de software transcende a mera noção de código e instruções; ele representa a espinha dorsal da operacionalidade dos sistemas computacionais. Segundo Donald D. Knuth, o renomado matemático e cientista da computação, o software é o conjunto de algoritmos que dá vida à máquina, permitindo a execução de um conjunto de operações complexas e variadas. Sob uma perspectiva mais abrangente, Theodore O. Ellis, um dos pioneiros no campo da ciência da computação, expande essa definição, enfatizando que o software abarca tanto os algoritmos quanto os dados que estes manipulam, criando uma simbiose intricada que impulsiona o funcionamento do sistema.

O termo software engloba todos os elementos intangíveis de um sistema computacional, compreendendo um conjunto de instruções, códigos e dados que permitem o funcionamento, a interação e a execução de tarefas em um computador ou dispositivo eletrônico. O software é uma entidade abstrata que coordena e controla os recursos físicos (como processadores, memória e dispositivos de armazenamento) para realizar operações complexas e variadas.

De forma simplificada podemos definir software como:

- “Um ou mais programas que definem uma aplicação específica para o computador".

- “Parte lógica que dota o equipamento físico de capacidade para realizar todo tipo de trabalho.

Exemplos de software:

- Um programa que represente um pequeno jogo pode ser considerado um software.

- Um conjunto de 500 programas que juntos realizam a administração e controle de uma caderneta de poupança também é um software.

- Sistemas operacionais.

As distintas categorias de software, delineadas por autores como David S. Cargo, englobam o software de sistema, que é o alicerce primordial, um ecossistema de programas essenciais que gerenciam os recursos do hardware e fornecem um ambiente propício à execução de outros programas. Ao mesmo tempo, temos o software aplicativo, um conjunto de programas destinados a cumprir funções específicas para os usuários finais, uma grande quantidade de ferramentas que abrange desde o processamento de texto até a manipulação de multimídia e jogos de alta complexidade. O software de programação, delineado por John Guttag, é a trilha de desenvolvimento que empodera os programadores a criar novos programas, fornecendo as ferramentas, como compiladores e ambientes de desenvolvimento integrado, para escrever e compilar código com precisão.

De forma simplificada podemos definir as categorias de software como:

- Software de Sistema ou básico: Este componente essencial é encarregado de gerenciar os recursos do hardware e proporcionar uma plataforma para a execução de outros programas. Isso inclui o sistema operacional, responsável por tarefas fundamentais como a

alocação de memória e o gerenciamento de processos.

- Software Aplicativo: São os programas que desempenham tarefas específicas para os usuários finais. Estes abrangem uma ampla gama de categorias, desde processadores de texto e planilhas até navegadores web, jogos e aplicativos de design gráfico.

- Software de Programação: Ferramentas e ambientes que facilitam o desenvolvimento de novos programas. Isso inclui compiladores, editores de código e ambientes de desenvolvimento integrado (IDEs), que fornecem as ferramentas necessárias para escrever e compilar código.

Um software aplicativo muito utilizado atualmente é Software cliente/servidor. Equivale a um tipo de software de aplicação desenvolvido para redes ou Internet. São aplicações constituídas de dois lados: o cliente e o servidor.

O ambiente cliente /servidor surgiu para substituir os grandes computadores, integrando os microcomputadores em rede, permitindo a distribuição de recursos através das redes corporativas. Algumas características são:

- Pode ser utilizado em diversos tipos de redes;

- Pode ser utilizado em vários tipos de meios de comunicação;

- Possui tráfego de rede mais reduzidos;

- Segurança e desempenho são questões importantes para essa arquitetura de rede;

- A rede pode crescer adicionando-lhe servidores.

- Uma parte da aplicação roda na estação de trabalho (Cliente) e outra parte roda no servidor, dividindo a carga de processamento.

1.1 Analogia entre hardware e software

A interface física de comunicação entre usuário e o computador é o hardware. A interface lógica de comunicação entre usuário e computador é o software. A figura 1 representa a visão geral desta analogia.

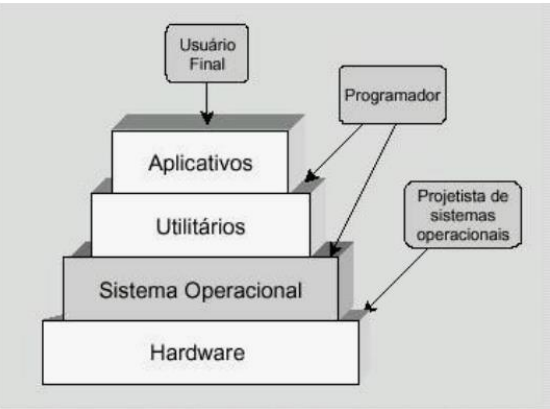


Figura 1 – Visão geral

A figura 2 apresenta a organização do Software em um computador

Figura 2 – Relação Hardware e software

1.2 Programa

O programa, a expressão concreta do software, é um conjunto delineado de instruções meticulosamente elaboradas em uma linguagem de programação específica, como exposto por Maurice H. Halstead.

Um programa é um conjunto específico de instruções e códigos escritos em uma linguagem de programação que serve para realizar uma tarefa ou conjunto de tarefas específicas em um sistema computacional. Ele representa uma manifestação concreta de software, sendo um arquivo ou conjunto de arquivos que podem ser executados por um sistema operacional.

A distinção crucial entre o código fonte e o código objeto, delineada por autores como Peter J. Ashenden, ressoa na essência dos programas. O código fonte é a linguagem humana, a expressão primordial do algoritmo, enquanto o código objeto é sua transformação, uma representação passível de ser processada pelo hardware. Consequentemente, o programa executável, formulado a partir da união do código objeto e outros recursos essenciais, torna-se o catalisador da ação, o embrião do desempenho algorítmico.

1.3 Elementos de um Programa

- Código Fonte: É o texto escrito na linguagem de programação escolhida pelo desenvolvedor. Este código é posteriormente traduzido ou compilado em uma forma que o computador pode entender e executar.

- Código Objeto ou Binário: É a versão traduzida do código fonte, representada em uma forma que o hardware pode processar diretamente.
- Executável: É o arquivo final que contém o código objeto e outros recursos necessários para a execução do programa.

1.4 Geração de Software

As gerações de software referem-se a diferentes estágios de desenvolvimento e evolução na história do software de computador. Vamos explorar as principais gerações de software, começando com a primeira:

- 1ª Geração de Software (1940s - 1950s)
Características:

- Linguagem de Máquina: Os primeiros programas eram escritos diretamente em linguagem de máquina, que é a linguagem de baixo nível compreendida pelo hardware do computador.
- Foco em Hardware Específico: Os programas eram altamente dependentes do hardware específico da máquina para a qual foram escritos.
- Programação de Baixo Nível: Programadores precisavam ter um profundo entendimento da arquitetura do hardware e codificar diretamente em binário ou em linguagem de montagem.
- Exemplo de Software: O ENIAC (Electronic Numerical Integrator and Computer), o primeiro computador eletrônico de grande escala, era programado diretamente em linguagem de máquina.

- 2ª Geração de Software (Finais dos anos 1950 - Início dos anos 1960)
Características:

- Linguagem Assembly: Introdução da linguagem Assembly, uma linguagem simbólica mais legível pelos humanos, mas ainda próxima do código de máquina.
- Sistemas Operacionais: Surgiram os primeiros sistemas operacionais, como o UNIVAC I e o IBM OS/360, que forneciam abstrações de hardware para facilitar a programação.

- Uso de Transistores: A transição dos tubos de vácuo para transistores permitiu a miniaturização dos computadores e o aumento da confiabilidade.
- Exemplo de Software: COBOL (Common Business-Oriented Language) foi uma das primeiras linguagens de programação de alto nível desenvolvidas nessa época.

- 3ª Geração de Software (Década de 1960 - Década de 1970)
Características:

- Linguagens de Alto Nível: A popularização de linguagens de alto nível como FORTRAN, COBOL, ALGOL, e mais tarde, C, fez com que a programação se tornasse mais acessível e menos dependente do hardware.
- Multiprogramação: Sistemas operacionais permitiram a execução simultânea de vários programas, melhorando a eficiência dos sistemas.
- Introdução da Noção de Arquivos: Surgiu a ideia de arquivos, permitindo que os dados fossem armazenados de forma estruturada.
- Exemplo de Software: O sistema operacional UNIX, desenvolvido na década de 1970, é um marco da terceira geração de software.

- 4ª Geração de Software (Década de 1980 - Década de 1990)
Características:

- Linguagens de Programação de Alto Nível Avançadas: Surgiram linguagens como C++, Java e Python, que oferecem recursos mais avançados e abstrações poderosas.
- Sistemas Distribuídos: Desenvolvimento de sistemas que podem operar em várias máquinas conectadas em rede.
- Ambientes Gráficos: Introdução de ambientes gráficos de usuário (GUI), facilitando a interação dos usuários com os sistemas.
- Exemplo de Software: O Microsoft Windows, lançado na década de 1980, é um exemplo proeminente da quarta geração de software.

- 5ª Geração de Software (Década de 1990 - Presente)
Características:

- Computação em Nuvem: Avanços na tecnologia da internet permitiram o desenvolvimento de serviços em nuvem, proporcionando armazenamento e processamento distribuído.
- Inteligência Artificial e Machine Learning: Surgimento de tecnologias avançadas de aprendizado de máquina, redes neurais e algoritmos de inteligência artificial.
- Big Data e Análise Avançada: Enormes conjuntos de dados podem ser processados e analisados para extrair informações valiosas.

- Exemplo de Software: Plataformas como Google Cloud, AWS, e Microsoft Azure são exemplos da infraestrutura de nuvem que caracteriza essa geração.

Cada geração de software trouxe avanços significativos, desde a programação em linguagem de máquina até as aplicações de aprendizado de máquina e computação em nuvem da atualidade. Essas evoluções moldaram profundamente a forma como os computadores são utilizados e continuam a impulsionar a inovação tecnológica.

2. Sistemas operacionais - SO

Um sistema operacional é um conjunto complexo de software que atua como uma interface intermediária entre o hardware de um computador e os programas/aplicativos que são executados nele. É responsável por gerenciar e coordenar recursos como processadores, memória, dispositivos de entrada e saída, entre outros. Além disso, fornece serviços essenciais como a criação de arquivos, comunicação com periféricos e controle de processos.

Os sistemas operacionais são um conjunto de programas e serviços que atuam como uma interface entre os componentes de hardware do computador e os aplicativos que os usuários desejam executar. Eles têm várias funções essenciais, incluindo:

- Gestão de Recursos: Alocação e gerenciamento de recursos de hardware, como CPU, memória, dispositivos de armazenamento e periféricos, de maneira eficiente e justa.

- Fornecimento de Interfaces: Oferecimento de uma interface de usuário ou linha de comando para interação com o sistema e execução de aplicativos.

- Execução de Programas: Carregamento e execução de programas e aplicativos, garantindo que eles funcionem corretamente no hardware subjacente.

- Gerenciamento de Arquivos: Organização, criação, leitura, gravação, exclusão e gerenciamento de arquivos e diretórios no sistema de armazenamento.

- Segurança: Implementação de mecanismos de segurança para proteger os dados e garantir o acesso autorizado aos recursos do sistema.

Os SO podem ser vistos com uma Visão top-down, uma camada de software que apresenta para o usuário uma visão modificada do hardware ou uma visão bottom-up, um gerenciador de recursos.

Resumidamente os SO têm as seguintes características:

- Definição de interface com o usuário;
- Compartilhamento do hardware entre os usuários;
- Permite aos usuários compartilhar dados entre si;
- Escalonamento de recursos entre usuários;
- Facilidades de Entrada e Saída;
- Recuperação de erros.

2.1 Classificação de Sistemas Operacionais

Os sistemas operacionais podem ser classificados de diversas maneiras, levando em consideração suas características, funcionalidades e aplicações específicas:

- Monotarefa e Multitarefa:

- Monotarefa: São sistemas operacionais que permitem a execução de apenas um programa por vez. Eles são mais simples e geralmente encontrados em sistemas embarcados ou dispositivos com funções específicas.

- Multitarefa: Permitem a execução simultânea de múltiplos programas, dividindo o tempo do processador entre eles. É o tipo mais comum em computadores pessoais e servidores.

- Monousuário e Multiusuário:

- Monousuário: Projetados para atender a apenas um usuário por vez. Sistemas de uso pessoal, como em desktops comuns, são exemplos típicos.
- Multiusuário: Possibilitam que vários usuários utilizem o sistema simultaneamente, cada um com seu ambiente de trabalho independente. São comuns em servidores e sistemas compartilhados.

- Monoprogramáveis e Multiprogramáveis:

- Monoprogramáveis: Permitem a execução de um único programa por vez e não aceitam a concorrência de processos.
- Multiprogramáveis: São capazes de executar múltiplos programas concorrentemente, gerenciando eficientemente o compartilhamento de recursos.

- Sistemas de Tempo Real:

- Requerem respostas imediatas e precisas a eventos específicos,como em sistemas de controle de tráfego aéreo.

- Sistemas Distribuídos:

Permitem que um conjunto de computadores trabalhe em conjunto para fornecer um único ambiente coerente para o usuário, geralmente conectados por uma rede.

- Sistemas Embarcados:

Projetados para operar em dispositivos específicos, muitas vezes com recursos limitados, como microcontroladores em eletrônicos de consumo.

- Sistemas de Tempo Compartilhado:

Permitem que vários usuários utilizem um sistema simultaneamente, interagindo através de terminais remotamente conectados.

- Sistemas de Batch:

Processam tarefas em lotes, sem a interação direta do usuário, sendo comuns em ambientes de processamento de dados.

Estas classificações fornecem uma visão abrangente dos diversos tipos de sistemas operacionais, cada um projetado para atender às necessidades específicas de diferentes ambientes e cenários de computação. Cada categoria apresenta suas próprias características e desafios, demonstrando a diversidade e a adaptabilidade dos sistemas operacionais no universo da computação.

2.2 Arquitetura dos SO

A arquitetura de um sistema operacional é uma estrutura complexa que organiza e coordena os diversos componentes do sistema computacional. Essa arquitetura é composta principalmente pelo kernel e pelo shell.

2.2.1 Kernel

O kernel é o núcleo do sistema operacional, e é responsável por fornecer serviços essenciais para o funcionamento do sistema e a interação com o hardware. Ele opera no modo privilegiado, o que significa que tem acesso total aos recursos do sistema. Algumas de suas principais funções são:

- Gestão de Memória: Alocação e desalocação de memória para programas em execução, garantindo que diferentes processos não interfiram uns com os outros.
- Gestão de Processos: Controle e coordenação dos processos em execução, escalonando a CPU para garantir que todos tenham a oportunidade de serem executados.
- Gestão de Dispositivos: Controla a interação com os dispositivos de hardware, como discos rígidos, impressoras, teclados, etc.
- Gestão de Arquivos: Organiza e gerencia os arquivos no sistema de armazenamento, permitindo sua criação, leitura, gravação e exclusão.
- Comunicação entre Processos: Facilita a comunicação e a troca de dados entre diferentes processos em execução.

2.2.2 Shell

O shell é a interface de usuário com o sistema operacional. É um ambiente de linha de comando ou uma interface gráfica que permite que os usuários interajam com o sistema operacional e executem comandos. Algumas das funções do shell incluem:
- Interpretação de Comandos: Recebe os comandos inseridos pelo usuário e os traduz para instruções compreensíveis pelo kernel.
- Gestão de Processos: Permite a execução, pausa e término de processos, bem como a comunicação entre eles.
- Manipulação de Arquivos e Diretórios: Facilita a criação, leitura, gravação e exclusão de arquivos e diretórios.
- Personalização do Ambiente: Os usuários podem configurar e personalizar o ambiente do shell de acordo com suas preferências.
- Scripting: Permite a criação de scripts, ou seja, sequências de comandos que podem ser executadas em lote para automatizar tarefas.

2.2.3 Integração entre Kernel e Shell

O shell interage diretamente com o usuário e envia comandos para o kernel. O kernel, por sua vez, executa as operações solicitadas pelo shell e retorna os resultados.

Isso cria uma comunicação bidirecional entre o usuário e o hardware, mediada pelo sistema operacional.

Em sistemas modernos, o shell pode ser tanto uma interface de linha de comando (CLI) quanto uma interface gráfica de usuário (GUI). A escolha entre eles depende das preferências do usuário e das tarefas a serem realizadas.

A eficácia da arquitetura de um sistema operacional depende da habilidade do kernel em gerenciar os recursos de hardware de forma eficiente e da capacidade do shell em fornecer uma interface amigável e funcional para os usuários interagirem com o sistema. Essa colaboração entre kernel e shell é o que possibilita o funcionamento fluido e eficaz de um sistema operacional.

2.3 Computadores com Múltiplos Processadores: Uma Visão Abrangente

Os computadores com múltiplos processadores, também conhecidos como sistemas multiprocessados, são sistemas que possuem mais de uma unidade de processamento central (CPU). A arquitetura multiprocessada pode variar desde sistemas com dois processadores até grandes clusters de servidores com centenas de CPUs. Vamos explorar como esses sistemas funcionam, considerando tanto sistemas fortemente quanto fracamente acoplados.

2.3.1 Sistemas Fortemente Acoplados

Em sistemas fortemente acoplados, todas as CPUs têm acesso igualitário à memória física e a todos os periféricos. Elas compartilham uma memória comum, o que significa que qualquer CPU pode acessar qualquer parte da memória diretamente. Além disso, as CPUs têm a capacidade de se comunicar diretamente entre si. Esse tipo de sistema geralmente possui um sistema operacional que gerencia as tarefas e a alocação de recursos de maneira eficiente.

- Funcionamento:

- Acesso à Memória Compartilhada: Todas as CPUs têm a capacidade de ler e escrever na mesma área de memória física. Isso permite uma comunicação eficiente e rápida entre os processadores.

- Escalonamento de Processos: O sistema operacional deve ser capaz de distribuir tarefas de forma eficaz entre os processadores disponíveis. Ele decide quais threads serão executados em quais núcleos e por quanto tempo, visando a maximizar a utilização de todos os núcleos.

- Sincronização e Comunicação: Como múltiplos processadores podem compartilhar recursos, é crucial implementar mecanismos de sincronização para garantir que diferentes partes do sistema não interfiram umas com as outras.

- Alta Escalabilidade: Adicionar mais processadores pode resultar em aumentos significativos de desempenho. Isso faz dos sistemas fortemente acoplados uma escolha popular para servidores e ambientes de alto desempenho.

2.3.2 Sistemas Fracamente Acoplados

Em sistemas fracamente acoplados, as CPUs têm sua própria memória e geralmente operam de forma independente umas das outras. Elas se comunicam através de uma rede de comunicação, o que pode resultar em latências maiores em comparação com sistemas fortemente acoplados. Cada CPU pode ter seu próprio sistema operacional e recursos.

- Funcionamento:

- Recursos Isolados: Cada processador tem sua própria memória e não compartilha recursos diretamente com os outros. Isso significa que eles têm menos visibilidade e controle uns sobre os outros.

- Comunicação por Rede: Processadores se comunicam através de uma rede, o que pode resultar em latências maiores em comparação com a comunicação interna a um processador. Isso pode afetar a eficiência da comunicação entre processadores.

- Limitações de Escalabilidade: Adicionar mais processadores pode não levar a um aumento linear de desempenho devido à latência de comunicação. Exemplos: Um cluster de servidores independentes, onde cada servidor tem seu próprio sistema operacional e memória.

2.3.3 Comparação: Fortemente Acoplado vs. Fracamente Acoplado

A principal diferença está na forma como as CPUs compartilham recursos, com sistemas fortemente acoplados compartilhando memória física e sistemas fracamente acoplados geralmente tendo memória isolada.

- Aplicações: Sistemas fortemente acoplados são ideais para aplicações que requerem alta interação entre os processadores e uma alta taxa de transferência de dados.
- Sistemas fracamente acoplados são mais apropriados para cenários onde a independência entre os processadores é uma prioridade.

Ambas as arquiteturas têm suas vantagens e são escolhidas com base nas necessidades específicas de aplicativos e no tipo de operações a serem realizadas.

2.4 SO para Multicore

Os sistemas operacionais multicore são projetados para aproveitar ao máximo o poder de processamento de computadores com múltiplos núcleos de CPU. Esses

sistemas têm a capacidade de executar múltiplos threads simultaneamente, o que significa que podem executar várias tarefas independentes de forma concorrente. Para entender completamente como isso funciona, é importante entender o que são threads e como são gerenciados em um sistema operacional multicore.

2.4.1 Threads

Um(a) thread é uma unidade básica de execução em um programa. Ele representa uma sequência de instruções que podem ser executadas independentemente das outras partes do programa. Um programa pode ter vários threads, cada um realizando tarefas diferentes. Eles compartilham o mesmo espaço de memória e recursos, o que permite uma comunicação mais eficiente entre eles.

Os threads são parte fundamental da programação concorrente e paralela, permitindo que múltiplas tarefas sejam executadas simultaneamente em um único processo ou programa. Vamos explorar detalhadamente os threads.

- Características das Threads:

- Execução Independente: Threads são unidades independentes de execução que podem executar partes separadas de um programa ao mesmo tempo. Isso permite que um programa realize tarefas concorrentemente.
- Contexto de Execução Próprio: Cada thread possui seu próprio contexto de execução, incluindo registradores de CPU, contador de programa e pilha de execução. Isso permite que os threads mantenham estados separados e realizem operações independentes.
- Compartilhamento de Recursos: Threads podem compartilhar recursos do processo, como memória, arquivos abertos e descritores de arquivo. No entanto, o compartilhamento deve ser gerenciado com cuidado para evitar condições de corrida e problemas de concorrência.

- Tipos de Threads:

Existem dois tipos principais de threads:

- Threads de Usuário (User-Level Threads): São gerenciadas inteiramente por uma biblioteca de threads em nível de usuário, sem a intervenção do kernel do sistema operacional. São mais leves em termos de recursos, mas não aproveitam totalmente a multiprocessamento (multicore) do hardware.
- Threads de Kernel (Kernel-Level Threads): São gerenciadas diretamente pelo kernel do sistema operacional. São mais robustas e eficientes em termos de multiprocessamento, já que o kernel pode agendar threads em diferentes núcleos de CPU. No entanto, a criação e gerenciamento de threads de kernel tendem a ser mais pesados em termos de recursos.

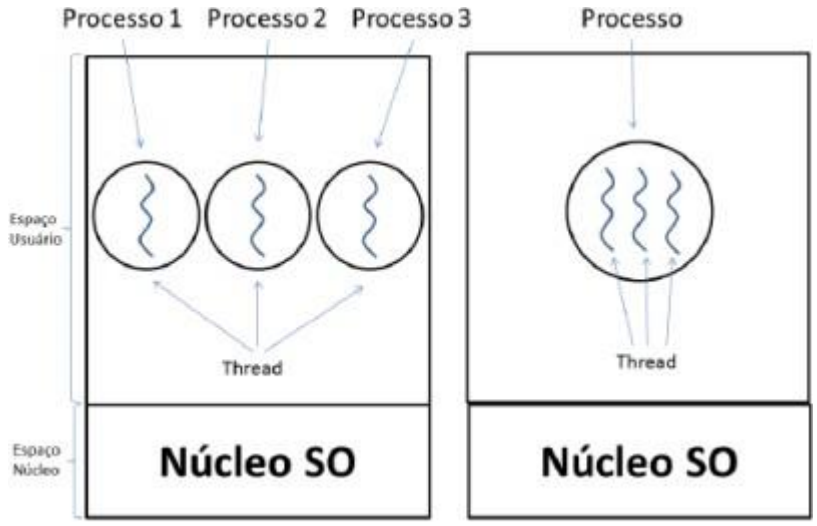


Figura 3 – Thread

- Uso de Threads:

Os threads são usados em diversas aplicações e cenários:

- Melhoria de Desempenho: Os threads são usados para dividir tarefas computacionalmente intensivas em partes menores que podem ser executadas concorrentemente, aproveitando o poder de processamento de múltiplos núcleos de CPU.
- Programação Concorrente: Em aplicativos de servidor, como servidores web, threads são usadas para lidar com múltiplas solicitações de clientes simultaneamente.
- Programação Paralela: Em computação paralela, threads são usadas para executar partes separadas de um cálculo em paralelo, o que é essencial para tarefas de alto desempenho, como simulações científicas e processamento de imagens.
- Interface de Usuário Responsiva: Em aplicativos com interfaces gráficas, threads são usadas para manter a interface do usuário responsiva enquanto tarefas de longa duração são executadas em segundo plano.'

- Estados de um Thread:

- Pronto (Ready): O thread está pronto para ser executado, mas ainda não foi selecionado pelo sistema operacional para execução.
- Executando (Running): O thread está sendo executado atualmente pela CPU.
- Bloqueado (Blocked ou Waiting): O thread está temporariamente inativo, muitas vezes devido à espera de algum recurso, como a conclusão de uma operação de leitura ou escrita em disco.

2.4.2 Sistema Operacional Multicore

Quando um sistema possui múltiplos núcleos de CPU, o sistema operacional tem a capacidade de distribuir a carga de trabalho entre esses núcleos. Isso é feito através do uso eficaz de threads.

2.4.2.1 Gerenciamento de Threads

O sistema operacional deve ser capaz de gerenciar os threads de forma eficaz para aproveitar ao máximo os recursos de hardware disponíveis. Isso inclui:
- Escalonamento: O sistema operacional decide quais threads serão executados em quais núcleos e por quanto tempo. O objetivo é maximizar a utilização de todos os núcleos, evitando sobrecarga ou inatividade.
- Sincronização: Como os threads compartilham recursos, é importante garantir que eles não interfiram uns com os outros. Mecanismos de sincronização, como semáforos são usados para controlar o acesso a recursos compartilhados.
- Comunicação entre Threads: Os threads muitas vezes precisam trocar informações. O sistema operacional fornece mecanismos de comunicação, como filas e canais, para facilitar isso.
- Gestão de Concorrência: Em um ambiente multicore, é crucial garantir que os threads não entrem em conflito ao tentar acessar os mesmos recursos. Técnicas como bloqueios e exclusões mútuas são usadas para prevenir condições de corrida.

2.4.2.2 Benefícios dos Sistemas Operacionais Multicore

- Paralelismo Real: Com múltiplos núcleos, é possível realizar tarefas de forma verdadeiramente simultânea, aumentando significativamente o desempenho.
- Melhor Resposta a Eventos: Threads podem ser usados para lidar com eventos em tempo real, como interações do usuário ou comunicações de rede, sem interromper outras operações.
- Melhor Desempenho de Aplicativos Multitarefa: Aplicativos que fazem uso intensivo de CPU, como renderização de gráficos ou simulações, podem se beneficiar significativamente de um ambiente multicore.
- Maior Capacidade de Processamento: Em geral, os sistemas operacionais multicore podem lidar com cargas de trabalho mais pesadas e complexas.

Em resumo, os sistemas operacionais multicore são projetados para otimizar a utilização de computadores com múltiplos núcleos de CPU. Eles fazem isso através do gerenciamento eficiente de threads, permitindo a execução de várias tarefas independentes simultaneamente. Isso resulta em um aumento significativo no desempenho e na capacidade de processamento de sistemas modernos.

2.5 Exemplos de SO comerciais

Existem vários sistemas operacionais disponíveis, cada um com suas características e aplicações específicas:

- Windows: Desenvolvido pela Microsoft, o Windows é um dos sistemas

operacionais mais populares, amplamente utilizado em desktops e laptops.

- macOS: Criado pela Apple, o macOS é projetado exclusivamente para hardware da Apple, conhecido pela sua interface intuitiva e design elegante.
- Linux: Uma família de sistemas operacionais de código aberto, o Linux é altamente flexível e é amplamente utilizado em servidores, supercomputadores e dispositivos embarcados.
- Unix: O Unix foi o precursor de muitos sistemas operacionais modernos e ainda é amplamente utilizado em ambientes empresariais.

3. Linguagens de programação

Linguagens naturais, como o português ou o inglês, servem à comunicação humana. Em contrapartida, os computadores usam **linguagens de programação** – linguagens artificiais projetadas para instruir máquinas. Um programa de computador é um conjunto de instruções formuladas em uma linguagem de programação que diz ao hardware o que executar, passo a passo. Essas linguagens variam em nível de abstração e sintaxe, mas todas têm como objetivo traduzir lógica e algoritmos em operações computacionais executáveis.

3.1 História das Linguagens de Programação e da IA:

- **Década de 1940–1950 (Linguagem de Máquina):** Os primeiros computadores eram programados diretamente em código de máquina (bits e bytes), específico ao hardware de cada máquina. Essa programação era extremamente trabalhosa e propensa a erros, pois os desenvolvedores precisavam controlar manualmente cada instrução de baixo nível (por exemplo, instruções da CPU em binário ou hexadecimal).

- **Final dos anos 1950 – início dos anos 1960 (Assembly):** Surgiram as linguagens **Assembly**, que introduziram mnemônicos para as instruções de máquina, tornando o código mais legível para humanos. Ainda assim, cada instrução Assembly correspondia aproximadamente a uma instrução de máquina. Ferramentas chamadas *montadores* (assemblers) traduzem código Assembly para código de máquina. Exemplo: era comum usar Assembly em programação de sistemas e controladores da época.

- **Décadas de 1960–1970 (Linguagens de Alto Nível):** Com o avanço da tecnologia, surgiram linguagens de alto nível projetadas para abstrair detalhes de hardware. Por exemplo, **Fortran** (1957) foi criada para cálculos científicos, **COBOL** (1959) para aplicações comerciais e **ALGOL** (1958) serviu como base teórica para muitas linguagens posteriores. Essas linguagens permitiram escrever código mais próximo da linguagem humana (com instruções do tipo “se”, “para cada”, etc.), facilitando o desenvolvimento e a portabilidade entre máquinas. Também nesse período foram criadas **Pascal** (1970) – voltada ao ensino de programação estruturada – e **C** (1972), desenvolvida para sistemas operacionais (com nível de abstração maior que Assembly, mas ainda próxima do hardware).

- **Década de 1970 (Linguagem Funcional e Lógica):** Nos anos 1960 e 1970 surgiram linguagens paradigmas específicos. Por exemplo, **Lisp** (1958) tornou-se popular em pesquisas de Inteligência Artificial por sua abordagem funcional, que privilegia listas e recursão. **Prolog** (1972), de paradigma lógico, ganhou espaço em sistemas especialistas e aplicações de lógica computacional. Essas linguagens influenciaram áreas específicas da computação e introduziram conceitos como processamento simbólico e inferência.

- **Década de 1980 (Orientação a Objetos e Linguagens de Sistemas):** A partir dos anos 1980 proliferaram linguagens orientadas a objetos (OO). **C++** (década de 1980) estendeu C com classes e herança, trazendo abstração de objetos para desenvolvimento de sistemas complexos. Por volta de 1983 nasceu **Objective-C**, e nas décadas seguintes, linguagens OO ganharam destaque em softwares de larga escala.

- **Década de 1990 (Internet e Multiparadigma):** Com a popularização da internet, surgiram linguagens focadas em aplicações distribuídas e web. **Java** (1995) introduziu o conceito de "write once, run anywhere" por compilar para bytecode executável em máquinas virtuais (JVM), reforçando portabilidade. **JavaScript** (1995) foi criada para programação de páginas web, trazendo interatividade ao navegador. **Python** (1991, mas disseminou-se nos anos 2000) combinou paradigmas procedural e orientado a objetos, e tornou-se popular pela simplicidade de sintaxe e pela ampla biblioteca padrão. Na mesma década surgiram **Ruby** (1995), voltada à produtividade do programador, e **PHP** (1995) para desenvolvimento de sites dinâmicos. Essas linguagens são multiparadigma (suportam vários estilos de programação) e facilitaram muito o desenvolvimento de software complexo.

- **Anos 2000 – presente (Linguagens Modernas):** No século XXI surgiram linguagens com foco em produtividade, segurança e concorrência. **C#** (2000) combinou recursos de Java e C++, **Go** (2009) introduziu concorrência nativa (goroutines) em modelo semelhante ao C, e **Kotlin** (2011) estendeu o modelo de objetos da JVM com sintaxe moderna. **Python** e **JavaScript** continuaram ganhando popularidade em áreas como ciência de dados, IA e aplicações web. Em Inteligência Artificial, **Python** tornou-se dominante devido a bibliotecas especializadas (TensorFlow, PyTorch, etc.), enquanto **R** (1993) destaca-se em estatística e análise de dados. Atualmente, cada linguagem evolui buscando atender a necessidades específicas, refletindo um cenário diversificado onde a escolha depende de contexto e requisitos.

- **Inteligência Artificial (IA):**

São criadas bases de conhecimentos, obtidas a partir de especialistas e as linguagens fazem deduções, inferências e tiram conclusões baseadas nas bases de conhecimento. A IA envolve o uso de algoritmos e técnicas para permitir que as máquinas possam perceber, raciocinar, aprender e tomar decisões com base em dados. Os sistemas de IA são projetados para simular certos aspectos do pensamento humano e têm o potencial de realizar uma variedade de tarefas complexas. Eles podem processar grandes quantidades de dados, identificar padrões, reconhecer imagens e fala, compreender linguagem natural, tomar decisões baseadas em informações disponíveis e até mesmo interagir com humanos de maneira natural. Historicamente podemos citar:

- Década de 1950: A IA nasceu como um campo de pesquisa interdisciplinar, com pioneiros como Alan Turing, John McCarthy e Marvin Minsky. Fortran, uma das primeiras linguagens de alto nível, foi usada em pesquisas iniciais de IA.
- Década de 1960 - 1970: Lisp, uma linguagem funcional, tornou-se a linguagem de escolha para a pesquisa de IA devido à sua flexibilidade.
- Década de 1980 - 1990: Prolog, uma linguagem lógica, também foiamplamente usada em sistemas de IA, especialmente em sistemas especialistas.
- Década de 2000 - Presente: Python emergiu como uma das principais linguagens para IA, devido à sua simplicidade, vasta biblioteca e comunidade ativa.

Existem várias subáreas da IA incluindo:

- Aprendizado de Máquina (Machine Learning): envolve a construção de algoritmos e modelos que permitem que as máquinas aprendam com dados e melhorem seu desempenho ao longo do tempo sem serem explicitamente programadas.
- Redes Neurais Artificiais: são modelos computacionais inspirados no funcionamento do cérebro humano. Eles são capazes de aprender e reconhecer padrões complexos em dados.
- Processamento de Linguagem Natural (Natural Language Processing - NLP): lida com ainteração entre computadores e linguagem humana, permitindo que as máquinas compreendam, interpretem e gerem linguagem natural.
- Visão Computacional: envolve o desenvolvimento de algoritmos e técnicas que permitem que as máquinas "vejam" e interpretem informações visuais, como imagense vídeos.
- Robótica: combinação da IA com a engenharia de robôs, envolvendo o desenvolvimento de sistemas robóticos capazes de realizar tarefas físicas e interagir com o ambiente.

3.2 Paradigmas de Programação

- Procedural:

- Características: Baseada em funções e procedimentos que manipulam dados.

Ênfase na estruturação de código e reutilização de código. Nesse paradigma, os programas são organizados em uma sequência de instruções, que são executadas de forma linear, passo a passo. A ênfase está na *estruturação* do fluxo de controle (laços, condicionais e chamadas de procedimento). Na programação procedural, o foco principal é na decomposição do programa em procedimentos ou sub-rotinas. Um procedimento é uma sequência de instruções que realizam uma tarefa específica e podem ser chamados em diferentes partes do programa. Essa abordagem permite a reutilizaçãode código, uma vez que procedimentos podem ser chamados várias vezes em diferentes partes do programa. Exemplo: C, Pascal.

- Orientado a Objetos:

- Organiza o software em torno de **objetos**, instâncias de *classes* que encapsulam dados (atributos) e operações (métodos). Os princípios principais são **encapsulamento** (dados e métodos juntos), **herança** (reuso de propriedades entre classes) e **polimorfismo** (mesma interface com implementações diferentes). Esse paradigma modela conceitos do mundo real, facilitando a representação de entidades complexas e promovendo reutilização de código. Linguagens populares com POO são Java, C++ e Python (que suporta objetos). Em POO, programas são construídos definindo classes e criando objetos que interagem; isso traz organização hierárquica ao código e facilita a abstração de problemas. As informações são encapsuladas e, ao mesmo tempo, dados e funções circulam juntos. Os programas são organizados em torno de objetos, que são entidades que combinam dados (atributos) e comportamentos (métodos) relacionados. Como exemplo:

- Objeto: Pessoa;
- Atributos: Nome, data de nascimento, cor;
- Métodos: Acordar, comer, beber, dormir.

Resumidamente, temos:

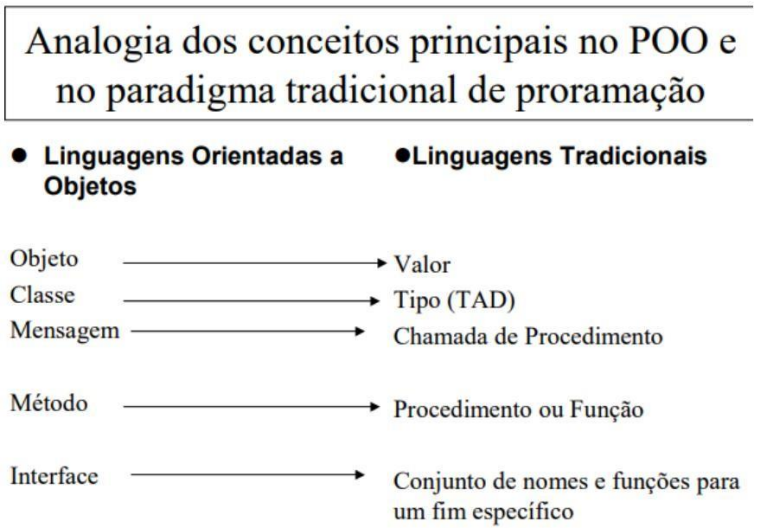
- A Programação Orientada a Objetos (POO) é um paradigma de desenvolvimento de software que organiza o código em torno de objetos, que são instâncias de classes. Cada objeto representa uma entidade do mundo real ou conceitual e possui atributos (características) e métodos (ações ou comportamentos).
Os quatro principais princípios da POO são:

- Objetos e Classes: Imagine uma **classe** como um molde (por exemplo, para fazer um carro) e os **objetos** são os carros feitos com esse molde. Cada carro pode ter características próprias (cor, modelo), mas todos seguem o mesmo molde.
- Encapsulamento: Pense em uma cápsula onde as partes importantes do carro (motor, eletrônicos) ficam protegidas. Você só pode interagir com o que a cápsula permite, como ligar o carro com uma chave. O que está dentro da cápsula não é visível diretamente. Portanto, esconde os detalhes internos do funcionamento do objeto, expondo apenas o necessário por meio de interfaces. Isso protege os dados e facilita a manutenção.
- **Herança**: É como quando você herda uma característica de sua família. Se você tem uma classe "Carro" e uma classe "CarroElétrico", a "CarroElétrico" herda tudo que a "Carro" tem, mas pode ter algo a mais, como uma bateria. Permite que uma classe herde atributos e métodos de outra classe, promovendo o reaproveitamento de código e a criação de hierarquias.
- **Polimorfismo**: Isso significa que um mesmo comando pode funcionar de maneiras diferentes. Por exemplo, tanto um carro de corrida quanto um carro elétrico têm a função "acelerar", mas eles podem fazer isso de maneiras diferentes. Permite que objetos de classes diferentes sejam tratados de forma semelhante, desde que compartilhem uma interface ou herança. Isso torna o código mais flexível e extensível.
- **Abstração**: Pense em dirigir um carro. Você só precisa saber dirigir, sem se preocupar com o que acontece dentro do motor. A abstração esconde detalhes complicados e só te mostra o que você precisa usar. Foca nos aspectos essenciais de um objeto, ignorando os detalhes desnecessários. Isso permite modelar sistemas complexos de forma mais simples e compreensível.

🌍 Resumo Visual da POO

Conceito	Definição	Exemplo Simples
Classe	Molde para objetos	`Carro`
Objeto	Instância de uma classe	`meu_carro = Carro()`
Atributo	Característica do objeto	`cor`, `modelo`
Método	Ação do objeto	`ligar()`, `desligar()`
Encapsulamento	Protege dados e oculta complexidade	Uso de métodos públicos
Herança	Herda características de outra classe	`Moto` herda de `Veiculo`
Polimorfismo	Mesmo método com comportamentos diferentes	`emitir_som()`
Abstração	Foca no essencial, oculta detalhes	Botão "ligar" do carro

No quadro abaixo apresentamos uma comparação dos conceitos principais no POO e no paradigma procedural.



Aspecto	Programação Orientada a Objetos (POO)	Programação Procedural
Organização	Em objetos (dados + métodos juntos)	Em funções e estruturas de dados
Reuso de Código	Usa herança e polimorfismo	Usa funções e bibliotecas
Facilidade de Manutenção	Alta (encapsulamento e abstração)	Menor, código mais espalhado
Escalabilidade	Melhor para sistemas grandes	Funciona bem para programas pequenos
Dados	Dados e comportamentos encapsulados	Dados e funções são separados
Exemplo da Vida Real	Um carro com ações próprias	Uma lista de dados manipulada por funções

3.3 Compilação vs. Interpretação

Compiladores e interpretadores são duas abordagens diferentes para processar e executar código em linguagens de programação.

- **Linguagens compiladas:** O código-fonte (texto do programa escrito pelo desenvolvedor) é transformado por um *compilador* em código de máquina nativo (binário) **antes** da execução. Esse código binário pode ser executado diretamente pelo hardware. Em geral, linguagens compiladas (como C e C++) geram executáveis que tendem a ter alta eficiência de execução, já que não dependem de um programa intermediário em tempo de execução. A desvantagem é que é necessário um processo de compilação a cada alteração no código e o programa compilado fica ligado à arquitetura alvo (por exemplo, um executável Windows não roda no Linux sem recompilação). Além do compilador, existem também *montadores* (para Assembly) e *ligadores* (linkers) que unem diversos módulos de código em um só executável.
- **Linguagens interpretadas:** O código-fonte é lido e executado **linha a linha** (ou bloco a bloco) por um programa chamado *interpretador*. Não há arquivo executável separado; o próprio interpretador analisa e executa cada instrução em tempo de execução. Esse modelo facilita a portabilidade – o programa-fonte pode rodar em qualquer plataforma que tenha o interpretador adequado –, além de permitir desenvolvimento mais interativo e testes rápidos. Exemplos clássicos de linguagens interpretadas incluem Python e JavaScript. A desvantagem tradicional é desempenho ligeiramente menor em relação a código nativo, já que há sobrecarga de interpretação em tempo de execução.
- **Abordagens híbridas e JIT:** Hoje, muitos ambientes de execução combinam compilação e interpretação para otimizar desempenho. Por exemplo, **Java** é compilada para um *bytecode* intermediário, que é executado pela Máquina Virtual Java (JVM). Durante a execução, o bytecode pode ser compilado *just-in-time* (JIT) em código nativo, melhorando a velocidade. Motores modernos de JavaScript (como V8 do Chrome) realizam técnicas JIT para compilar trechos do código durante a execução. Ainda assim, a distinção conceitual persiste: linguagens rotuladas como “compiladas” tendem a gerar executáveis binários, enquanto linguagens “interpretadas” são normalmente executadas dentro de um ambiente virtual.

  Resumo: Compiladores traduzem previamente para código de máquina, resultando em programas eficientes em tempo de execução, mas requerem etapas de compilação e são menos flexíveis a mudanças rápidas. Interpretadores executam o código diretamente, oferecendo portabilidade e rapidez no ciclo de desenvolvimento, às custas de desempenho bruto. A escolha entre compilação e interpretação depende do contexto: linguagens de sistemas críticos geralmente são compiladas, enquanto linguagens de script são interpretadas.

3.4 Nível de Abstração

- **Baixo Nível**: Aproxima-se do hardware do computador. Linguagens de baixo nível trabalham diretamente com registradores, endereços de memória e instruções nativas da CPU, permitindo controle fino do hardware. Isso oferece alto desempenho e eficiência (útil em sistemas operacionais, drivers e aplicações de tempo real), mas exige esforço maior do programador e reduz portabilidade. Exemplos típicos são **Assembly** (onde cada instrução reflete diretamente uma instrução de máquina) e, em

certa medida, **C** (muitas vezes considerado "médio-baixo nível" por permitir manipulação de ponteiros e operações bit a bit). Nessas linguagens, o desenvolvedor gerencia manualmente recursos como alocação de memória e controle de hardware, o que aumenta a complexidade do código.

.

- **Alto Nível**: Oferece múltiplas camadas de abstração acima do hardware. Linguagens de alto nível providenciam construções avançadas – como estruturas de dados complexas, gerenciamento automático de memória, bibliotecas padronizadas abrangentes e sintaxe próxima à linguagem natural – que tornam o desenvolvimento mais rápido e o código mais legível. Exemplos são **Python**, **Java**, **JavaScript** e **Ruby**. Esses ambientes priorizam a produtividade do programador e a portabilidade entre plataformas, ao custo de um overhead de abstração (menor desempenho e menos controle direto sobre o hardware).
- **Nível Médio / Crossover:** Algumas linguagens (por exemplo, C++) são consideradas de nível médio, pois combinam aspectos dos dois extremos: permitem abstrações de alto nível (classes, bibliotecas padrão ricas) e ao mesmo tempo dão acesso a recursos de baixo nível (como ponteiros e alocação manual de memória). Isso confere flexibilidade, permitindo tanto programação de sistemas quanto de aplicações de maior abstração.

Resumo: Quanto menor o nível de abstração, mais próximo da "linguagem da máquina" o programador deve trabalhar, obtendo desempenho máximo e controle preciso. Quanto maior o nível, mais o software é "independente" do hardware, favorecendo facilidade de escrita e manutenção. A escolha depende da aplicação: sistemas embarcados ou de alto desempenho podem exigir linguagens de baixo nível (como C/Assembly), enquanto desenvolvimento de aplicações web, empresariais e de pesquisa tende a usar linguagens de alto nível (Python, Java, etc.).

3.5 Aplicações

As linguagens de programação são usadas em diversas áreas, cada uma com necessidades específicas. A seguir, destacam-se algumas aplicações típicas e exemplos de linguagens comuns em cada caso:

- **Desenvolvimento Web:** Utiliza linguagens tanto para o *front-end* (lado do cliente) quanto para o *back-end* (lado do servidor). *Front-end* é dominado por **JavaScript** (executado em navegadores, com frameworks como React e Angular) e tecnologias associadas (HTML/CSS[1] não são linguagens de programação, mas markup/estilo). *Back-end* frequentemente usa **Python** (frameworks Django, Flask), **Java** (Spring), **Ruby** (Ruby on Rails) e **JavaScript** (Node.js), entre outras. Essas linguagens suportam servidores web e serviços de aplicação, além de integrarem com bancos de dados e APIs.
- **Desenvolvimento de Aplicativos Móveis:** Envolve linguagens específicas de plataforma e multiplataforma. Para *iOS*, a linguagem oficial atual é **Swift** (antecessora: Objective-C);

[1] CSS (Cascading Style Sheets) é uma **linguagem de estilização**, responsável pela apresentação visual do conteúdo HTML, definindo cores, fontes, tamanhos, posicionamento e outros aspectos visuais.

para *Android*, é **Kotlin** (compatível com Java) ou Java puro. Além disso, existem frameworks híbridos que usam **JavaScript** (por exemplo, React Native, Ionic) ou outras linguagens para desenvolver apps que rodam em ambas as plataformas. Esses ambientes exigem linguagens que suportem interfaces gráficas ricas e acesso a recursos do dispositivo (câmera, GPS etc.).

- **Ciência de Dados e Análise:** Engloba estatística, aprendizado de máquina e processamento de dados. Linguagens populares nessa área incluem **Python** (com bibliotecas como NumPy, Pandas, TensorFlow, Scikit-Learn), **R** (focada em estatística e visualização de dados, com vasta coleção de pacotes no CRAN[2]) e **SQL** (para consultas a bancos de dados relacionais). Essas linguagens fornecem bibliotecas científicas, ambientes interativos e ferramentas de análise de grandes volumes de dados, sendo essenciais em pesquisa acadêmica, negócios e inteligência artificial.

- **Sistemas Embarcados e Internet das Coisas (IoT):** Envolve dispositivos com recursos limitados (microcontroladores, sensores, atuadores). As linguagens de escolha são geralmente **C** e **C++** pela eficiência e controle de hardware, além de permitir aproveitar bibliotecas de baixo nível. Recentemente, linguagens como **Rust** também têm ganhado espaço, oferecendo segurança de memória sem sacrificar desempenho. Esses sistemas usam compiladores que geram código otimizado para o hardware embarcado.

- **Jogos e Computação Gráfica:** O desenvolvimento de jogos envolve gráficos 2D/3D e física em tempo real. **C++** é amplamente usado em motores de jogos profissionais, devido ao alto desempenho e controle sobre recursos gráficos. **C#** é popular com o motor Unity. **Python** é empregado em scripts de automação e ferramentas internas, embora não seja usado diretamente na lógica de jogos de alta performance. Essas linguagens interagem com APIs gráficas (OpenGL, DirectX) e bibliotecas de game engines.

- **Inteligência Artificial e Machine Learning:** As linguagens dominantes são **Python** (devido a frameworks como TensorFlow e PyTorch) e **R** (para estatística). **Java** também é usado em algumas aplicações corporativas de IA (por exemplo, bibliotecas Weka, Deeplearning4j). Essas linguagens oferecem ambiente de prototipagem ágil, suporte a cálculos matemáticos e bibliotecas especializadas em algoritmos de aprendizado.

- **Automação, Scripts e DevOps[3]:** Tarefas rotineiras de automação (administrativas, test automation, devops) frequentemente utilizam **shell scripts** (bash, PowerShell) ou linguagens de script como **Python**. Essas linguagens permitem escrever scripts simples para manipular arquivos, executar programas, configurar ambientes de servidor e realizar tarefas repetitivas de forma automatizada.

Em suma, cada área emprega as linguagens que melhor atendem a seus requisitos: desempenho, portabilidade, riqueza de bibliotecas e ferramentas de suporte. A escolha de linguagem leva em conta o domínio da aplicação, a plataforma-

[2] O **CRAN** (Comprehensive R Archive Network) é um **repositório oficial de pacotes para a linguagem de programação R**. Ele funciona como uma grande coleção de softwares complementares, onde os usuários podem encontrar, instalar e compartilhar pacotes que estendem as funcionalidades do R.

[3] **DevOps** é uma metodologia que integra as equipes de **Desenvolvimento (Dev)** e **Operações (Ops)**, com o objetivo de automatizar, integrar e acelerar os processos de desenvolvimento de software, testes, implantação e manutenção em produção.

alvo e a preferência da equipe de desenvolvimento.

3.6 Caracteristicas de algumas linguagens comercias

A seguir, apresenta-se uma breve descrição de algumas linguagens amplamente usadas na indústria, destacando seu ano de criação, paradigmas principais e características marcantes:

- **JavaScript:** Criado em 1995 (ECMAScript) para páginas web. *Paradigma:* Multiparadigma – suporta programação **procedural**, **orientada a objetos** (baseada em protótipos) e aspectos **funcionais** (funções de primeira classe, programação reativa). *Características:* É interpretado/executado pelos navegadores e em servidores (Node.js). Suporta tipagem dinâmica e possui uma sintaxe flexível. Permite manipular o DOM[4] para criar interfaces web interativas e, atualmente, é usado também em aplicativos móveis (React Native) e servidor. Possui vasto ecossistema de bibliotecas e frameworks modernos (React, Angular, Vue).
- **Ruby:** Criado em 1995 por Yukihiro Matsumoto. *Paradigma:* Orientado a objetos puro (tudo é objeto). *Características:* Focado na produtividade do desenvolvedor e legibilidade do código, seguindo a filosofia "otimizar para a felicidade do programador". Ruby tem sintaxe expressiva e dinâmica, gerencia memória automaticamente via *garbage collector*[5] e é interpretado pela máquina virtual Ruby (MRI). É conhecido pelo framework web Ruby on Rails, que facilitou o desenvolvimento rápido de aplicações web.
- **Go (Golang):** Desenvolvido pelo Google em 2009. *Paradigma:* Imperativo e concorrente. *Características:* É compilada e estaticamente tipada, com sintaxe simples inspirada em C. Destaca-se pelo suporte nativo à **concorrência** (por meio de *goroutines* e *channels*), facilitando o desenvolvimento de aplicações distribuídas e de alto desempenho em rede. Oferece gerenciamento automático de memória, ferramentas de criação e teste integradas, e tem ênfase em desempenho e simplicidade. É muito usado em sistemas de servidores web, containers (Docker é escrito em Go) e infraestrutura de rede.
- **R:** Lançada em 1993 (versão inicial do S/R). *Paradigma:* Funcional / orientado a objetos (paradigma misto). *Características:* Projetada para estatística e análise de dados. R possui um grande conjunto de pacotes (*CRAN*) para estatística, modelagem e visualização gráfica. É interpretada, com tipagem dinâmica e alta capacidade de manipulação de vetores e matrizes. É amplamente usada em pesquisa científica, análise de dados e estatística, oferecendo recursos avançados de gráficos e modelagem estatística.

[4] O **DOM** (Document Object Model) é uma **interface de programação que representa a estrutura de um documento HTML ou XML na forma de uma árvore de objetos**. Cada elemento, atributo e conteúdo dentro de uma página web é convertido em um objeto que pode ser manipulado dinamicamente por linguagens de programação, como JavaScript.

[5] O Garbage Collector é um processo que **identifica e libera automaticamente blocos de memória que não estão mais sendo utilizados** pelo programa, prevenindo vazamentos de memória e melhorando a eficiência na utilização dos recursos.

- **Kotlin:** Desenvolvida pela JetBrains em 2011. *Paradigma:* Multiparadigma – orientado a objetos e funcional. *Características:* É compilada para bytecode Java e executa na JVM. Tem sintaxe concisa e moderna, inferência de tipos, segurança contra *null* (nullable/non-nullable)[6] e interoperabilidade total com Java. Tornou-se linguagem oficial para desenvolvimento Android (ao lado de Java) devido à sua simplicidade e melhorias sobre Java. Além de Android, é usada em aplicações web (Ktor), sistemas de servidor e multiplataforma (Kotlin Multiplatform).

Cada linguagem acima tem pontos fortes próprios. A seleção de uma linguagem para um projeto leva em conta seu paradigma, ecossistema de bibliotecas, desempenho e adequação ao domínio de aplicação. Assim, linguagens como JavaScript, Python e Java continuam dominantes em muitas áreas, mas novas linguagens (como Go e Kotlin) vêm ganhando espaço em nichos específicos, sempre visando melhorar eficiência, segurança e produtividade no desenvolvimento de software.

[6] Os termos **nullable** e **non-nullable** estão relacionados ao conceito de **valores nulos** em linguagens de programação e banco de dados, definindo se uma variável, campo ou objeto pode ou não assumir o valor **nulo (null)**.