

Table of Contents

Sistemas Operacionais	3
1.1 O que os Sistemas Operacionais fazem	4
1.1.1 Hardware	6
1.1.2 Software	8
1.1.3 Visões do Sistema	10
1.2 Operação do Computador	13
1.3 Estrutura de Armazenamento	18
1.4 Estrutura de Entrada e Saída	25
1.5 Arquitetura do Sistema	31
1.6 Estrutura do sistema operacional	37
1.7 Operações do Sistema Operacional	38
1.8 Gerência de processos	40
1.9 Gerência de memória	42
1.10 Gerência de armazenamento	44
1.11 Proteção e Segurança	51
1.12 Sistemas de uso específico	53
1.13 Ambientes de Computação	56
1.14 Sistemas Operacionais de Código Aberto	59
Exercícios Práticos Resolvidos - 1	62
Domus 2	66
2.2 Interface usuário-sistema operacional	68
2.3 Chamadas de sistema	71
2.4 Tipos de chamadas de sistema	75
2.5 Programas do sistema	78
2.6 Projeto e implementação do sistema operacional	79
2.7 Máquinas virtuais	82
2.8 Geração do sistema operacional	85
2.9 Boot do sistema	88
Exercícios Práticos Resolvidos - 2	91
Questões 1	96
Prática 1	102
Respostas - Questões 1	148
Threads	151
4.1. Usos	153
4.2 Benefícios da Programação Multithread	159
4.3 Programação multicore	163
4.4 Modelos de múltiplas threads (multithreading)	166

4.5 Bibliotecas de threads	170
4.6 Threads em Java	186
4.7 Aspectos do Uso de Threads.....	193
4.8 Exemplos em Sistemas Operacionais	201
Exercícios Práticos - 4	206
Escalonamento de CPU	211
5.1 Conceitos básicos	215
5.2 Critérios de Escalonamento	222
5.3 Algoritmos de Escalonamento	228
5.4 Escalonamento de Threads	234
5.5 Escalonamento em Múltiplos Processadores	240
5.6 Exemplos de Sistema Operacional	243
5.8 Avaliação de Algoritmos de Escalonamento	247
Exercícios Práticos	253
6.1 Introdução - Gerenciamento de Memória	258
6.2 Conceitos Básicos	261
Associação de Endereços	265
Espaço de Endereços Lógicos e Físicos	267
Carregamento dinâmico	269
Bibliotecas de vínculo dinâmico e compartilhadas.....	271
6.3 Swapping	273
6.4 Alocação de memória contígua	276
Proteção e Mapeamento da Memória	278
Alocação de Memória	280
Fragmentação	283
6.5 Páginas	285
Método Básico da Paginação	288
Suporte do Hardware para Paginação	293
Proteção em Sistemas Paginados	298
Páginas Compartilhadas	301
6.6 Estrutura da Tabela de Página	305
Paginação Hierárquica	311
Tabelas de Página Invertidas	316
6.7 Segmentação	321
6.8 Visão Geral do Gerenciamento de Memória no Pentium	326
Exercícios Práticos 6	333
Bibliografia	339

Sistemas Operacionais

Bem-vindo ao nosso guia sobre Sistemas Operacionais! Nesta obra, exploraremos os conceitos fundamentais que regem o funcionamento dos sistemas que permitem que nossos dispositivos funcionem de maneira eficaz. Os sistemas operacionais são uma peça crucial da tecnologia moderna, servindo como intermediários entre o hardware e o software, gerenciando recursos, permitindo a execução de aplicativos e garantindo uma experiência de usuário fluida.

Este guia é estruturado para proporcionar uma compreensão acessível e prática dos sistemas operacionais, abrangendo desde a teoria básica até exercícios práticos que reforçam o aprendizado.

Como utilizar este material

Para aproveitar ao máximo este guia, recomendamos a seguinte abordagem:

- 1. Estude os conceitos:** Leia atentamente cada seção, focando na compreensão dos conceitos fundamentais. Não se preocupe se não entender tudo de imediato; os sistemas operacionais são um tema complexo que se torna mais claro com o tempo e a prática.
- 2. Pratique regularmente:** Utilize os exercícios práticos fornecidos para reforçar seu aprendizado. A experiência prática é essencial para solidificar o conhecimento teórico.
- 3. Resolva as questões:** Tente responder às questões propostas ao final de cada seção. Isso ajudará a avaliar sua compreensão e identificar áreas que podem precisar de revisão.
- 4. Explore além do material:** Encorajamos você a pesquisar tópicos adicionais que despertem seu interesse. A área de sistemas operacionais é vasta e está em constante evolução.
- 5. Aplique o conhecimento:** Sempre que possível, relate o que você aprendeu com situações do dia a dia ou problemas reais de computação. Isso ajudará a contextualizar o conhecimento adquirido.

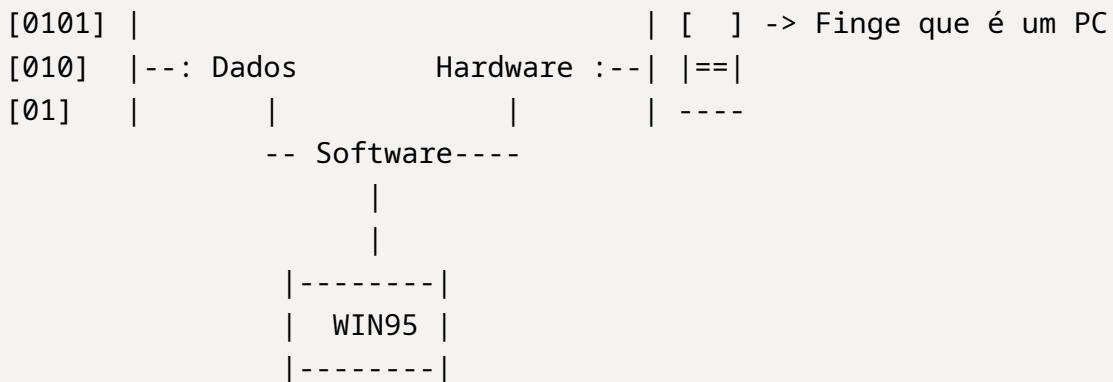
 Livro usado:

sistemas-operacionais-com-javascript.pdf

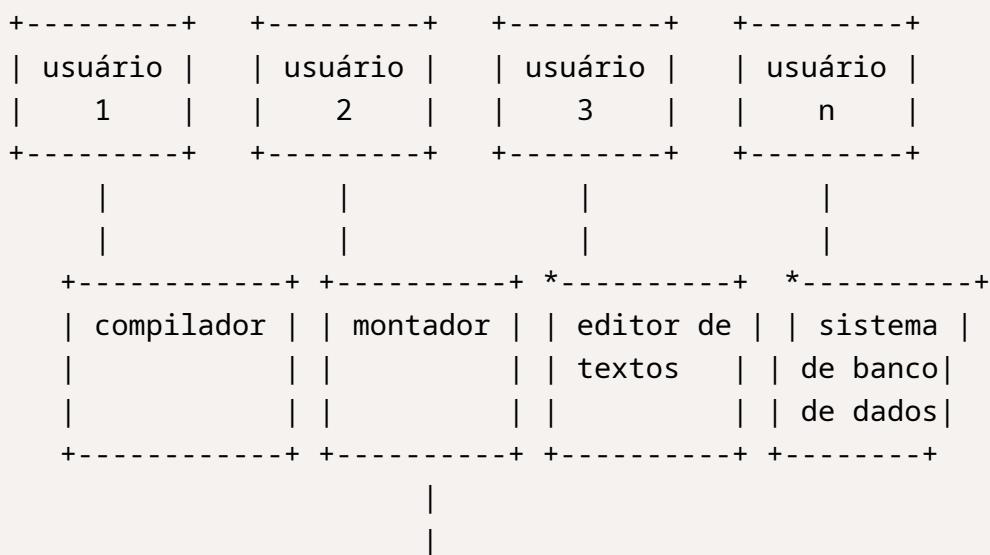
1.1 O que os Sistemas Operacionais fazem

Um sistema computadorizado ou só computador, pode ser *dividido em quatro partes*:

- Hardware
 - Sistema Operacional
 - Software
 - Usuários
- Também podemos considerar que um sistema computadorizado é composto por:



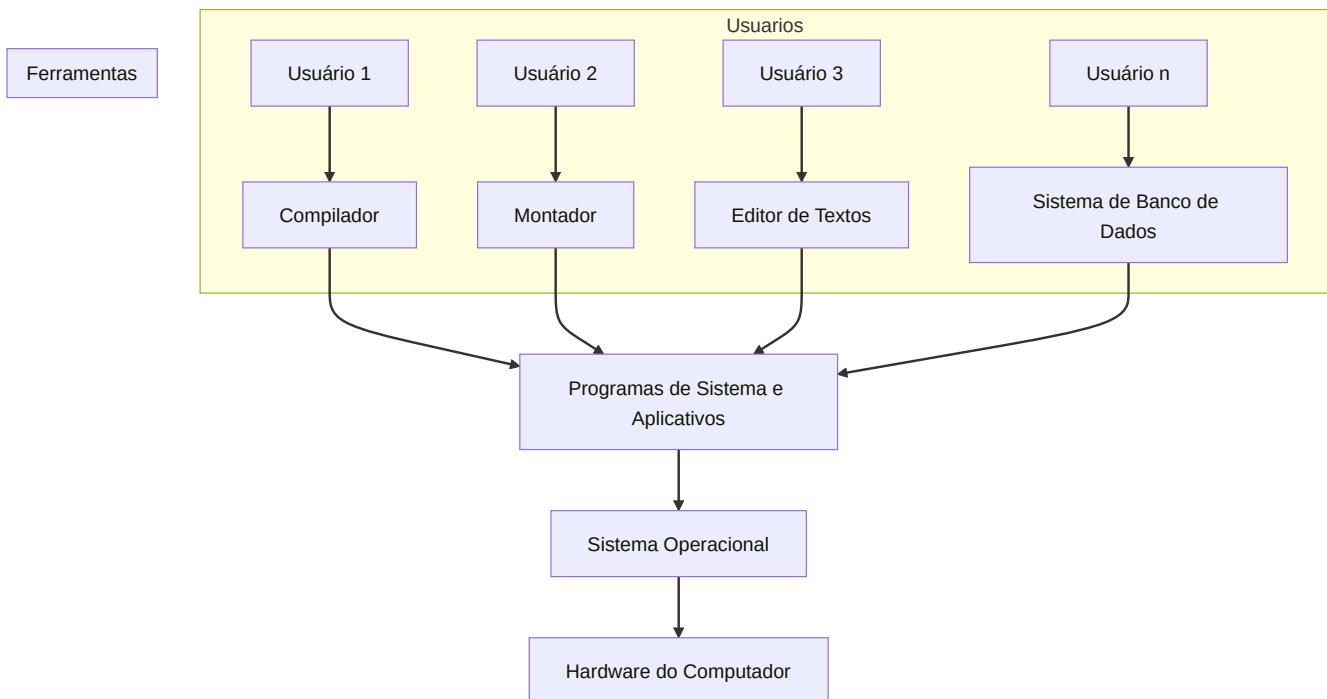
- Exemplo de Funcionamento de um Sistema Operacional



```

+-----+
| programas |
| de sistema |
| e aplicat- |
| ivos       |
+-----+
|
|
+-----+
| sistema   |
| operacional|
+-----+
|
|
+-----+
| hardware do |
| computador  |
+-----+

```



1.1.1 Hardware

O hardware de um computador é como os blocos fundamentais de Minecraft que compõem o mundo do seu computador. Assim como você precisa de diferentes tipos de blocos para construir estruturas complexas em Minecraft, um computador precisa de vários componentes de hardware para funcionar.

Componentes Principais

Processador (CPU)

Pense no processador como o jogador em Minecraft. Assim como o jogador executa ações e toma decisões, a CPU processa instruções e realiza cálculos. É o cérebro do computador.

Memória RAM

A RAM é como o inventário do jogador em Minecraft. Ela armazena temporariamente informações que o processador precisa acessar rapidamente, assim como você mantém itens importantes no seu inventário para uso imediato.

Armazenamento (HDD/SSD)

O armazenamento é semelhante aos baús em Minecraft. HDDs e SSDs guardam dados a longo prazo, como programas e arquivos, assim como os baús armazenam itens que você não precisa carregar o tempo todo.

Placa-mãe

A placa-mãe é como o terreno em Minecraft onde você constrói. Ela conecta todos os outros componentes, permitindo que eles se comuniquem entre si.

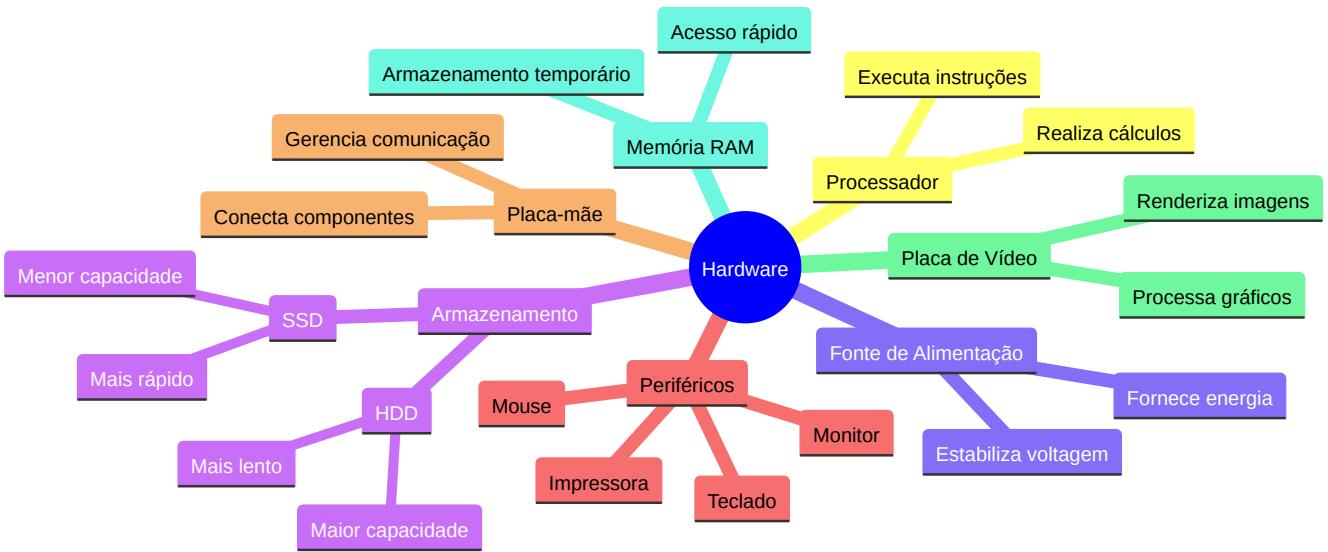
Placa de Vídeo (GPU)

A GPU é como o mecanismo de renderização em Minecraft. Ela processa gráficos e imagens, tornando possível ver o mundo digital na sua tela.

Fonte de Alimentação

A fonte de alimentação é como a energia redstone em Minecraft. Ela fornece energia para todos os componentes, mantendo tudo funcionando.

Mindmap do Hardware



Este mindmap ilustra os principais componentes de hardware de um computador, mostrando como eles se relacionam entre si, assim como diferentes estruturas em Minecraft se conectam para formar um mundo funcional.

Entender o hardware é essencial para compreender como os sistemas operacionais interagem com os componentes físicos do computador, gerenciando recursos e otimizando o desempenho, assim como um bom jogador de Minecraft gerencia seus recursos para construir e explorar eficientemente.

1.1.2 Software

Software é como o conjunto de regras e mecânicas que fazem o mundo de Minecraft funcionar. Assim como Minecraft tem diferentes tipos de mecânicas (como física, geração de mundo, interações de itens), um computador tem diferentes tipos de software que trabalham juntos para criar uma experiência funcional e interativa.

Tipos de Software

Sistema Operacional

O sistema operacional é como o modo de jogo em Minecraft (Sobrevivência, Criativo, etc.). Ele define as regras básicas de como o computador funciona e como os outros programas podem interagir com o hardware.

Aplicativos

Aplicativos são como os mods em Minecraft. Eles adicionam funcionalidades específicas ao sistema, permitindo que você realize tarefas como escrever documentos, navegar na internet ou editar imagens.

Drivers

Drivers são semelhantes aos comandos de bloco em Minecraft. Eles permitem que o sistema operacional se comunique com o hardware específico, assim como os comandos de bloco permitem interações complexas com o mundo do jogo.

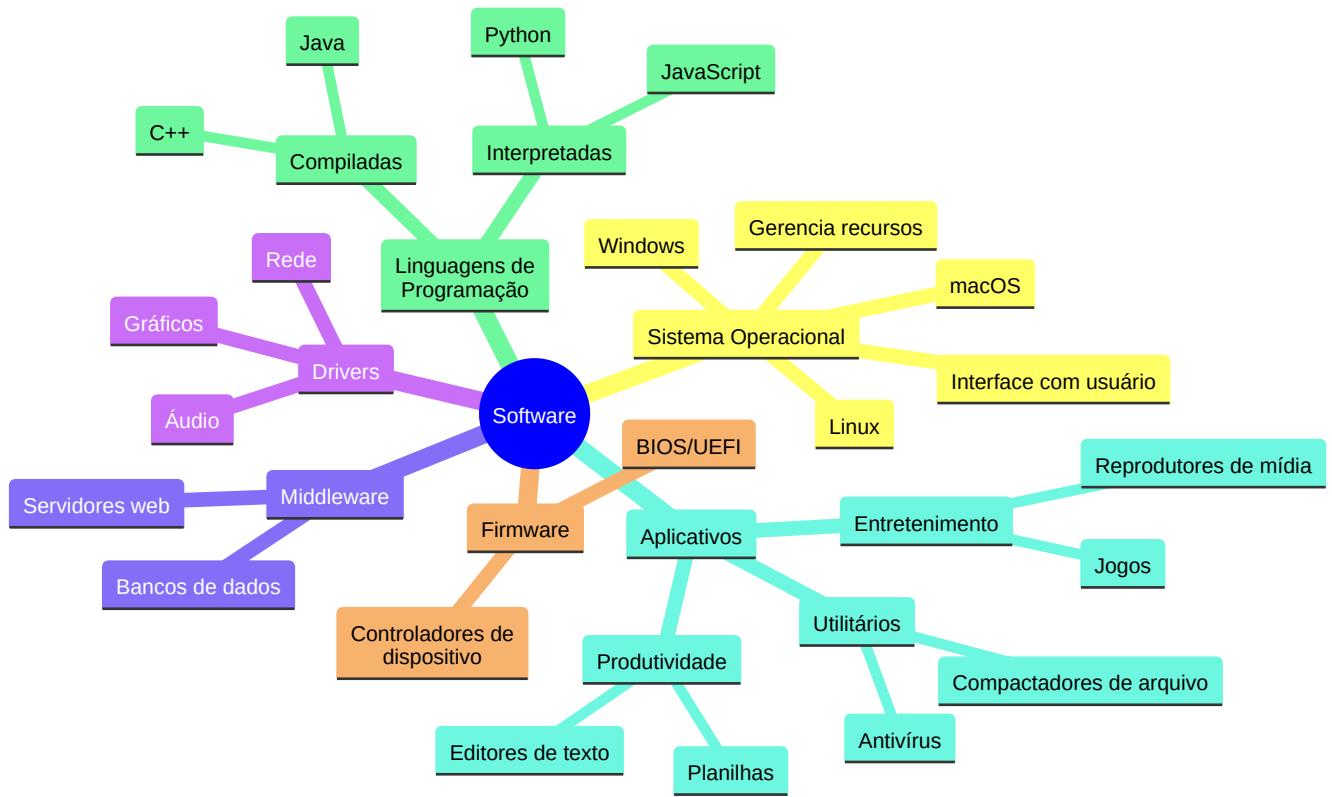
Firmware

O firmware é como as configurações internas dos blocos em Minecraft. É um software embutido no hardware que fornece instruções básicas para o funcionamento do dispositivo.

Linguagens de Programação

As linguagens de programação são como a linguagem de comandos em Minecraft. Elas permitem que os desenvolvedores criem software, assim como os comandos permitem aos jogadores criar comportamentos complexos no jogo.

Mindmap do Software



Este mindmap ilustra os principais tipos e categorias de software, mostrando como eles se relacionam e se organizam no ecossistema digital, assim como diferentes elementos se combinam para criar a experiência completa de Minecraft.

1.1.3 Visões do Sistema

Visão do Sistema: O Administrador do Servidor

Do ponto de vista do computador, o sistema operacional é como o administrador de um servidor Minecraft. Assim como um admin controla todos os aspectos do jogo, o sistema operacional gerencia intimamente o hardware do computador.

O Sistema Operacional como Alocador de Recursos

Imagine o sistema operacional como o sistema de plugins de um servidor Minecraft, responsável por gerenciar:

1. **Tempo de CPU:** Como o dia e a noite no Minecraft, distribuindo tempo para cada processo.
2. **Espaço de Memória:** Similar ao inventário dos jogadores, alocando espaço para programas.
3. **Armazenamento de Arquivos:** Como baús no Minecraft, organizando e armazenando dados.
4. **Dispositivos de E/S:** Portais para outros mundos, gerenciando a comunicação com dispositivos externos.

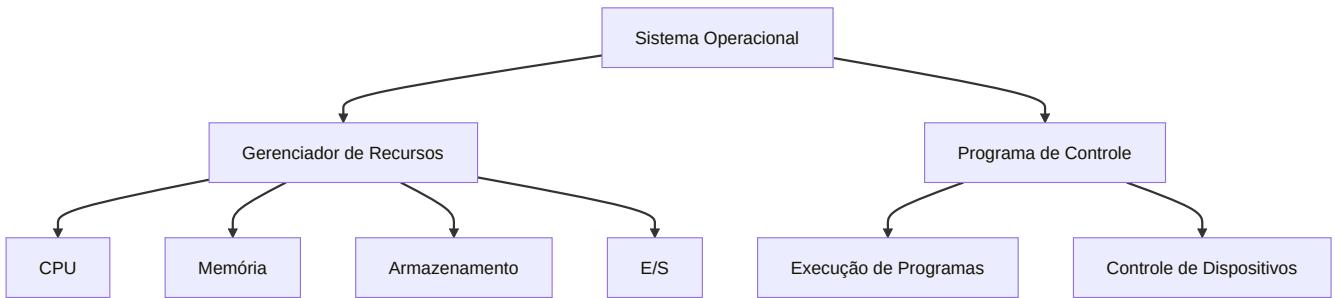
O sistema operacional deve alocar esses recursos de forma eficiente e justa, assim como um bom admin de Minecraft garante que todos os jogadores tenham acesso justo aos recursos do servidor.

Unidades de Armazenamento: Os Blocos do Mundo Digital

- **Bit:** O bloco mais básico, como um grão de areia no Minecraft.
- **Byte:** 8 bits, como um bloco completo no Minecraft.
- **Word:** A unidade nativa do computador, como um chunk no Minecraft.
- **Kilobyte (KB):** 1.024 bytes, como uma pequena construção.
- **Megabyte (MB):** 1.024^2 bytes, como uma vila inteira.
- **Gigabyte (GB):** 1.024^3 bytes, como um reino completo no Minecraft.

O Sistema Operacional como Programa de Controle

Assim como as regras e configurações de um servidor Minecraft, o sistema operacional controla a execução de programas e o uso de dispositivos para prevenir erros e uso indevido.



Este diagrama mostra como o Sistema Operacional, assim como o core de um servidor Minecraft, gerencia recursos e controla a execução de programas e dispositivos, mantendo todo o sistema funcionando harmoniosamente.

Visão do Usuário

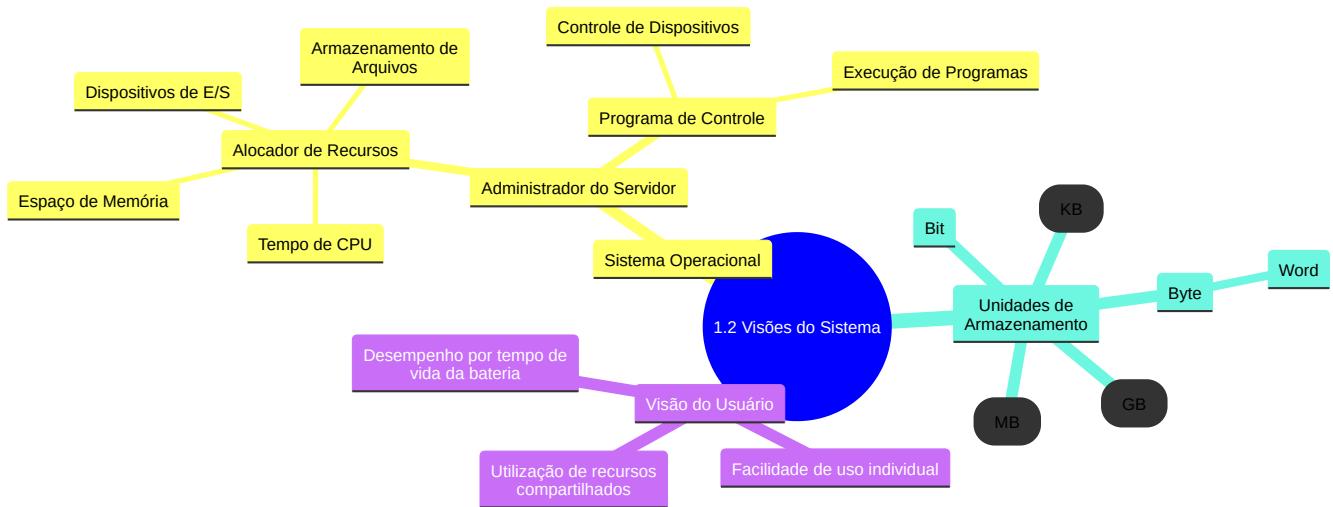
A visão do computador pelo usuário varia de acordo com a interface utilizada. Na maioria dos casos, os jogadores de Minecraft se sentam à frente de um computador, com um monitor, teclado, mouse e processador. Esse sistema foi projetado para que o jogador monopolize os recursos do computador.

O objetivo é proporcionar uma experiência mais rápida e imersiva no jogo. Nesse caso, o sistema operacional foi projetado principalmente para a facilidade de uso, com alguma atenção ao desempenho e pouca consideração à utilização de recursos – como a competição por espaço de memória e processamento.

É natural que o desempenho seja importante para o jogador; mas esses sistemas são otimizados para a experiência individual do jogador.

Em alguns casos, o jogador pode se conectar a um servidor remoto, permitindo que vários jogadores accessem o mesmo computador. Nesse caso, o sistema operacional foi projetado para um equilíbrio entre a facilidade de uso individual e a utilização de recursos compartilhados.

Alguns computadores podem ter pouca ou nenhuma visão do usuário. Por exemplo, os computadores embutidos nos consoles de jogos podem ter teclados numéricos e botões de luz indicadora, para mostrar o status do jogo, mas, em sua maioria, eles e seus sistemas operacionais são projetados para serem executados sem a intervenção do jogador.



1.2 Operação do Computador

Ao desligar o computador e ligá-lo, o que acontece? Como ele "chama" o Sistema Operacional.

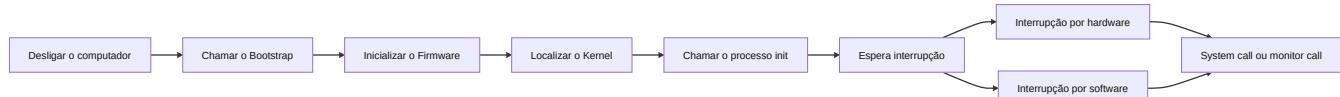
Para o computador começar a funcionar, ele chama um programa básico, chamado de **bootstrap**.

Normalmente, este programa está alocado na memória apenas de leitura (**ROM**) ou é salvo na memória de somente leitura apagável programavelmente (**EEPROM**).

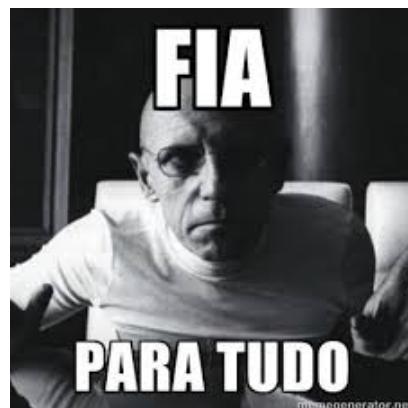
Este programa é conhecido como **Firmware**, pois está instalado diretamente no hardware, assim, ele inicializa todos os aspectos do sistema, desde os registradores da CPU até os dispositivos e o conteúdo na memória.

Para carregar o SO, ele precisa localizar o **Kernel**, que é o núcleo do sistema operacional. Assim que o Kernel é carregado na memória do computador, ele chama um processo chamado **init**, que espera uma interrupção do sistema ou do hardware. Os dois casos são:

- Se for pelo hardware, ele envia uma interrupção por sinal para a CPU, via normalmente o barramento do sistema;
- Se for por software, ele pode fazer de duas maneiras: chamando uma **system call** (chamada do sistema) ou usando um **monitor call** (monitor de chamada). Essas são operações especiais executadas para disparar uma interrupção, enviando um sinal para a CPU.



Quando a CPU recebe uma interrupção, ela para o que está fazendo e executa a rotina de tratamento correspondente:



Meme fia para tudo

A CPU então manda a execução para uma **localização fixa na memória**, onde essa localização contém o **endereço inicial** da rotina para **atender a essa interrupção**.

Essas **interrupções** podem ser tratadas de diferentes maneiras, e cada computador possui seu próprio mecanismo. Um método simples para isso é tratar a transferência chamando uma rotina genérica. Para dar mais enfoque em velocidade pode ser usada uma **tabela de ponteiros a pontando para as interrupções**, já que elas devem ser predefinidas. **Essa tabela é armazenada em memória baixa**, sendo ela a primeira parte ou locação da memória.

Esse **vetor de interrupção** vai ser indexado exclusivamente pelo número do dispositivo, fornecido com a requisição da interrupção para gerar o endereço do tratamento da interrupção:

Interrupção

CPU manda execução para local fixo na , com endereço da rotina de tratamento. 

Diferentes formas de tratar interrupções, cada  com seu próprio jeito.

Método simples:
Transfere para uma rotina genérica.  bootstrap

Método rápido:
 Tabela de ponteiros para interrupções, em memória baixa. 

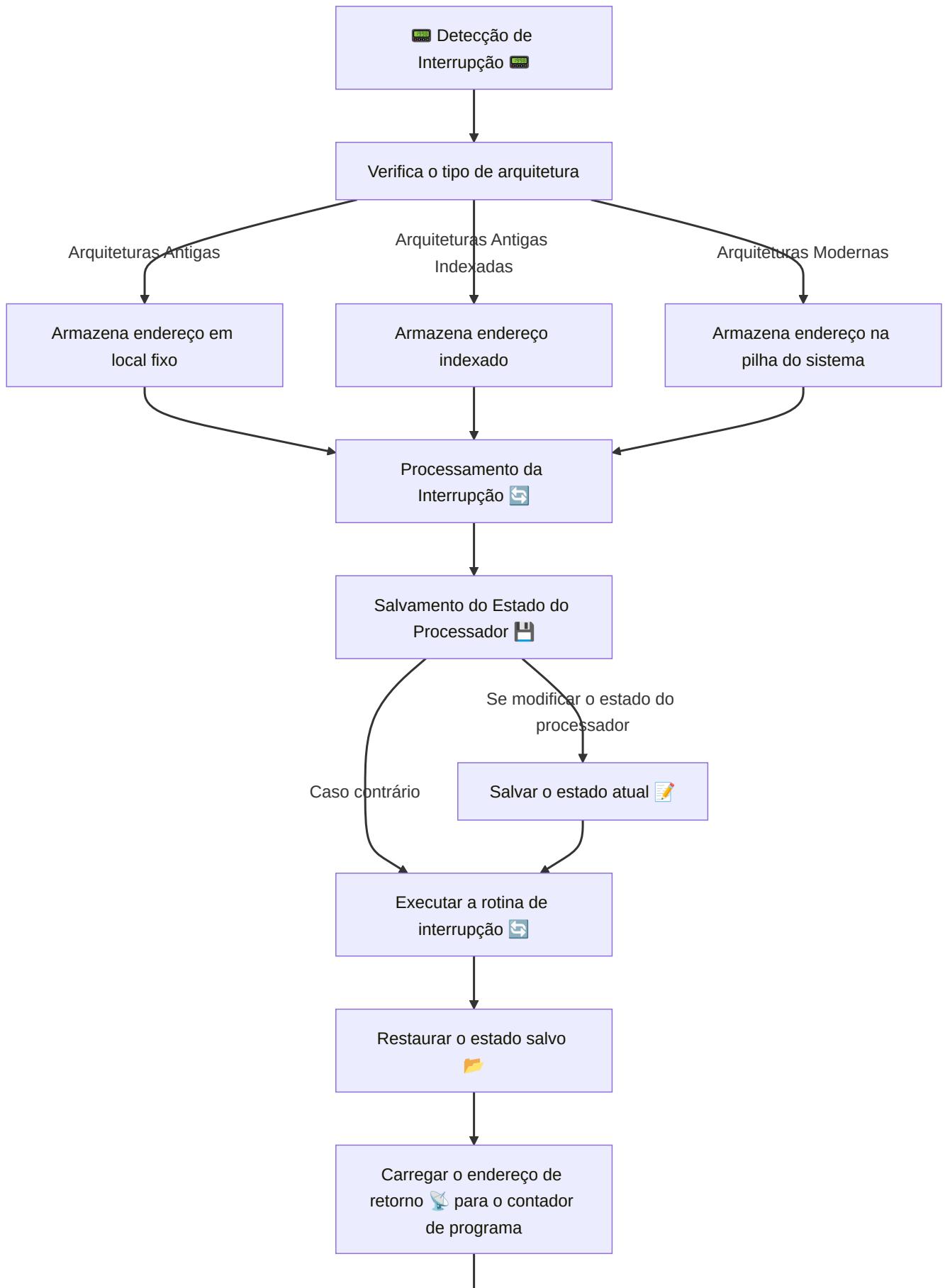
 Vetor usa  dispositivo para gerar endereço do tratamento. 

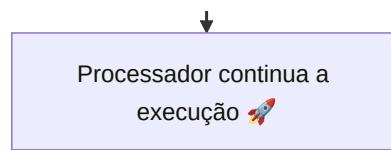
A arquitetura de interrupção **precisa salvar o endereço da instrução interrompida**, em projetos:

- Em alguns antigos armazenam o endereço da interrupção de maneira **fixa ou local indexado** por um numero do dispositivo;
- Em arquiteturas modernas, eles armazenam em **pilhas do sistema**;

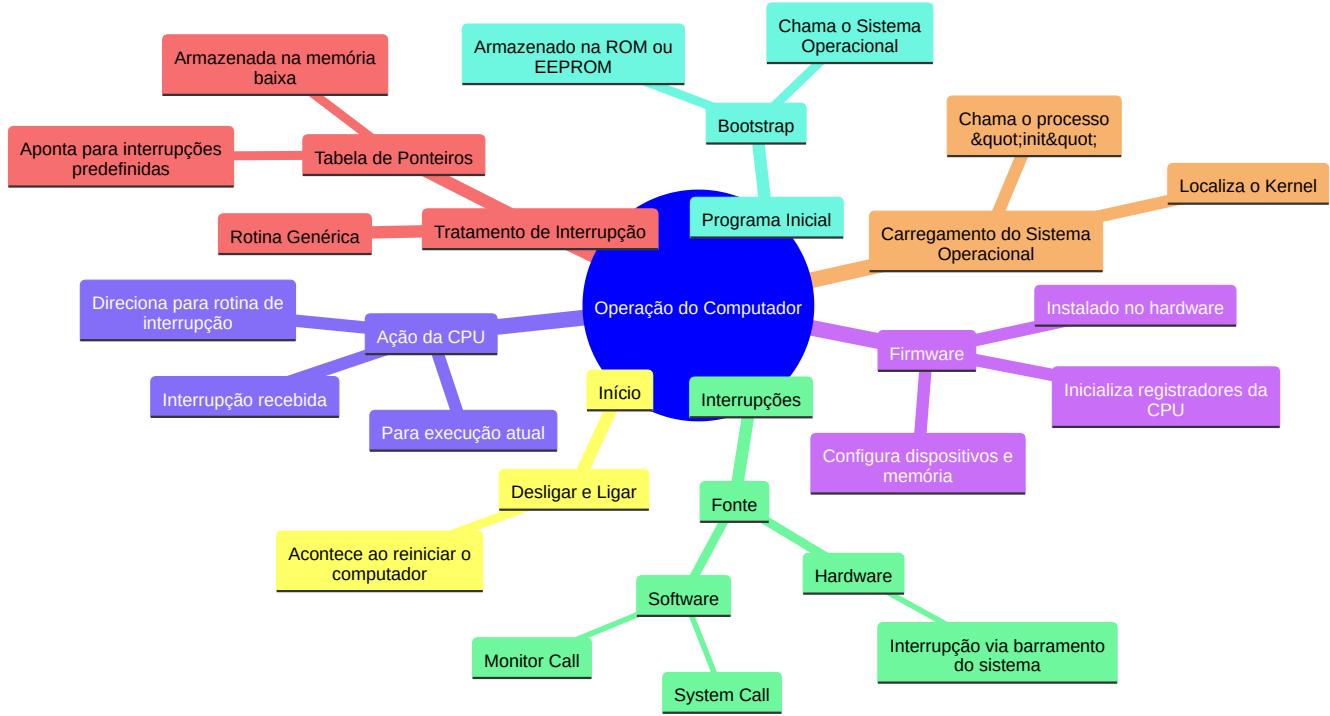
Se a rotina de interrupção precisar modificar algum estado do processador, por exemplo alterando os valores do **registrador**:

- Ela vai **salvar** o estado atual, explicitamente;
- Depois **carregar e restaurar** esse estado para depois **retornar**;
- Em seguida será carregado para o **contador de programa o endereço do retorno e o processador** que foi **interrompido** continua como se nada tivesse acontecido:





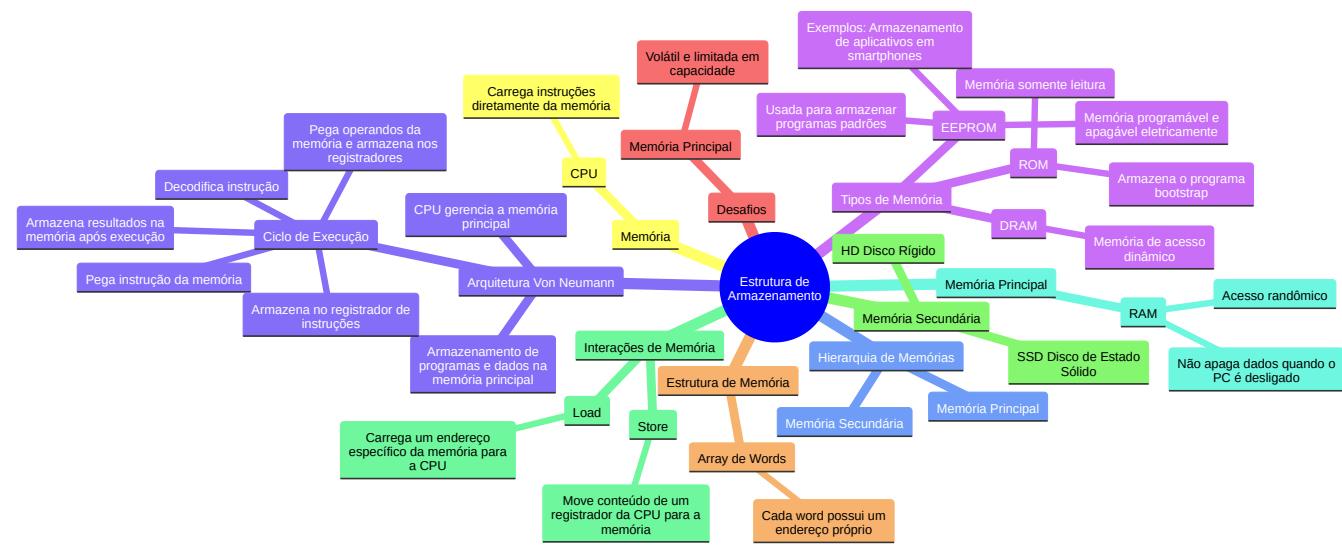
Diagrama



1.3 Estrutura de Armazenamento

Para os computadores que temos a **CPU** só consegue carregar instruções que vêm diretamente da memória.

- A memória não sendo nada, mas a **Memória Principal** – aquela cujo acesso é randômico, ou seja, desligar o PC não apaga os dados armazenados, que é a memória **RAM**.



i Veja mais sobre tipos de memória em:

A memória RAM é comumente feita numa arquitetura de semicondutores chamada de **Dynamic Random Access Memory** (DRAM) ou, em português, **memória de acesso dinâmica**.

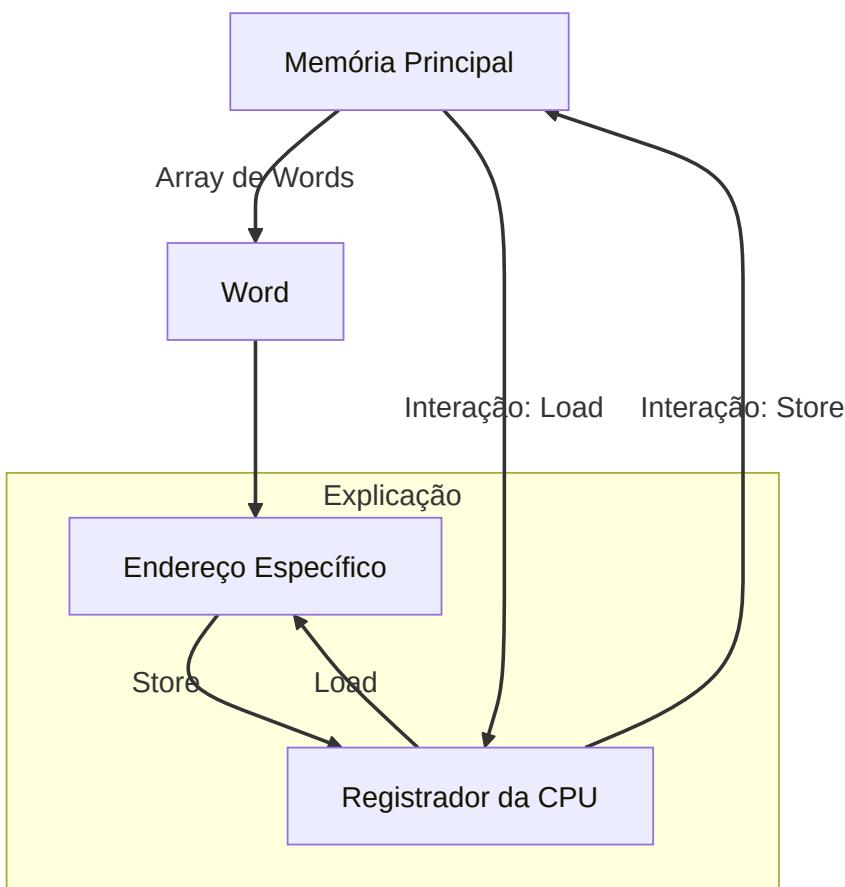
Um outro tipo de memória é aquela que só serve para leitura, assim como a mulher do seu amigo, apenas olhe. As conhecidas são:

- **ROM** (Read Only Memory) ==> normalmente vem nos computadores e é usada para armazenar o programa bootstrap.
 - Além disso, é usada por empresas de jogos para guardar os jogos, já que ela possui essa natureza imutável.
- **EEPROM** (Electrically Erasable Programmable Read Only Memory)
 - Por não ser modificada com frequência, essa memória costuma ser usada para armazenar programas padrões de modo estático.

- Smartphones, por exemplo, utilizam a EEPROM de modo que as fabricantes armazenam nele os aplicativos de fábrica.

Quaisquer destas memórias utilizam **um array de words** ou uma unidade de armazenamento.

- Cada *word* possui seu próprio endereço.
- As interações se dão por instruções:
 - **load** - carrega um endereço específico da **memória principal** para um dos **registradores da CPU**.
 - **store** - move um conteúdo de um **registrador da CPU** para a **memória principal**.



*Ilustração de um esquema sobre instruções da CPU (**load** e **store**)*

- A CPU carrega e armazena essas instruções tanto explicitamente (dizer para ela fazer) como de maneira automática - ela faz sozinha o carregamento da memória principal para serem executadas.

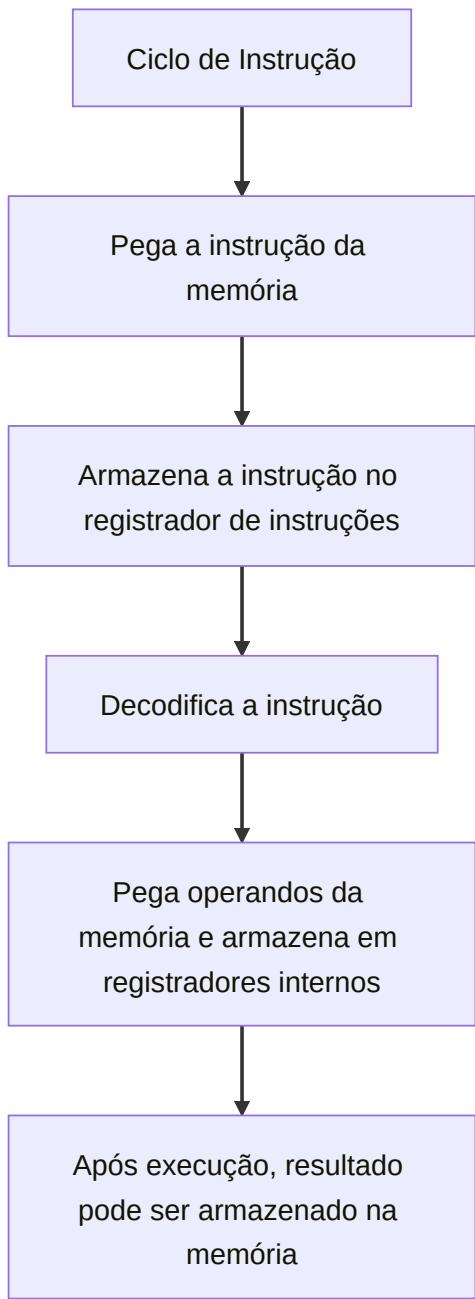
A arquitetura mais usada nos computadores modernos é a de **Von Neumann**. Essa arquitetura funciona da seguinte forma:

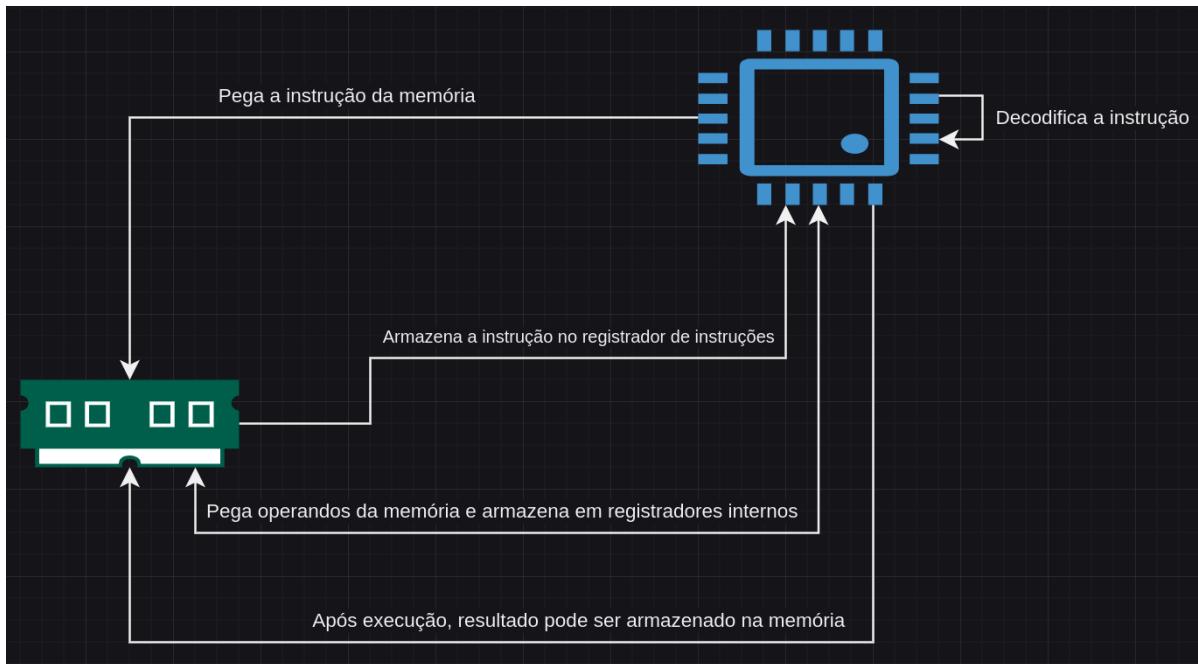
- Programas e dados são armazenados na memória principal.
- A CPU gerencia a memória principal.

Vamos para um ciclo de execução - quando uma instrução é dada:

1. Pega a instrução da memória.
2. Armazena essa instrução no **registraror de instruções**.
3. Essa instrução é então decodificada.
 1. Pode pegar operandos da memória e armazená-los em registradores internos.
4. Após a execução dos operandos, o resultado pode ser armazenado na memória.

Diagramas de Execução de Instrução





003 - Estrutura de Armazenamento

- i** A unidade de memória só consegue ver um fluxo de endereços de memória. Ela não sabe:
- Como são gerados (Gerados por contador de instruções, indexação, endereços literais e etc)
 - Para que servem
 - Se são instruções ou dados.

Seria bom, mas a vida não é um morango, a memória principal não consegue armazenar todos os dados e programas. Entretanto, não temos isso, já que:

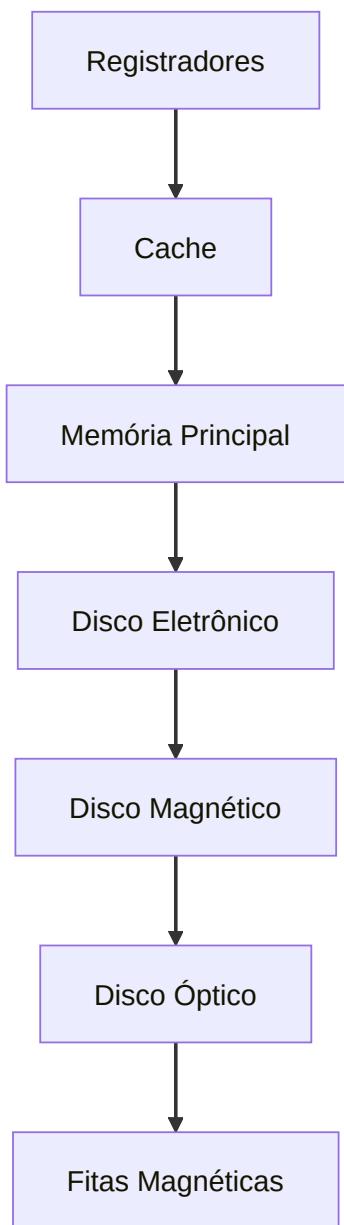
- A **memória principal** é **volátil**, ela perde os dados assim que a máquina é desligada.
- A memória principal possui um **armazenamento irrisoriamente pequeno** para armazenar todos os programas e dados.

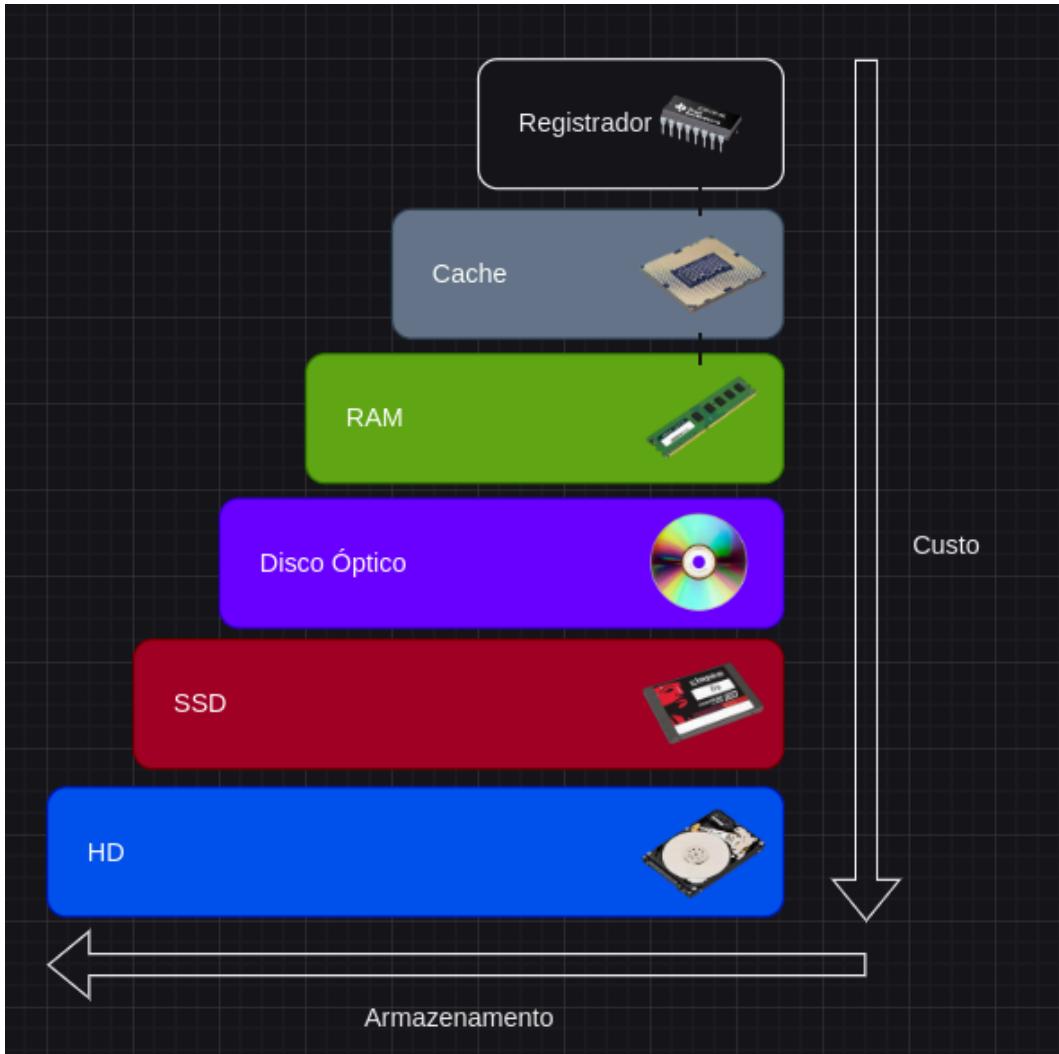
Assim, precisamos de outro tipo de memória chamado **memória secundária**, que tem o propósito de armazenar dados e programas de maneira permanente.

Um bom exemplo de memória secundária é o HD (Disco Rígido) e também temos outro tipo que está se tornando mais popular no mercado, o SSD (Disco de Estado Sólido).

No entanto, não há apenas dispositivos de armazenamento nessa hierarquia. Também podemos fazer uma hierarquia desses dispositivos, que é assim:

Diagramas de Dispositivos de Armazenamento:





003 - Estrutura de Armazenamento Hierarquia Dispositivos De Armazenamento

1.4 Estrutura de Entrada e Saída

Os dispositivos de Entrada e Saída (ou E/S), são um dos grandes pontos importantes para um Sistema Operacional, como podemos notar no armazenamento que possui grande importância para ser um dispositivo de E/S.

- Um outro ponto importante é que grande parte do código do SO é pensado para E/S;
 - Tanto por causa da **confiabilidade** como **desempenho**.

i Um sistema computadorizado para uso geral, consiste em:

- CPU
- Diversos tipos de controladores de dispositivos conectados por um barramento comum
- Cada controlador possui um tipo específico de dispositivo

Por exemplo, para o controlador SCSI (Small Computer-System Interface) podemos ter sete ou até mais dispositivos conectados ao mesmo controlador.

Cada controlador armazena **buffer local** e um **conjunto de registradores de uso especial**.

Os controladores tem duas funções básicas, que se baseiam:

- **Move** os dados para os dispositivos periféricos que controla.
- **Gerencia** o uso do buffer local.

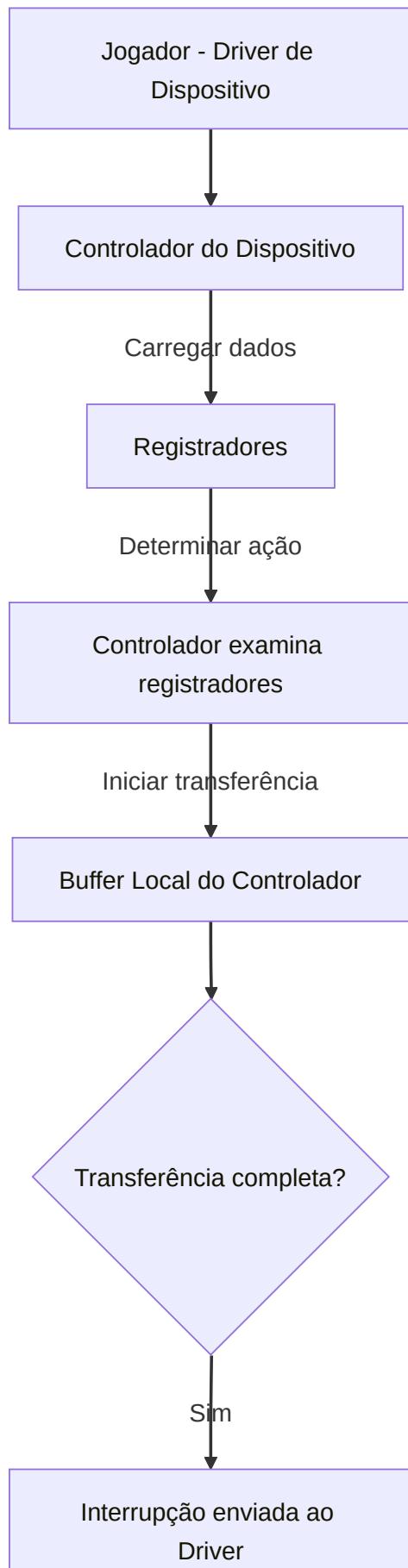
Tais sistemas possuem um **driver de dispositivo** (driver de dispositivo) que serve como ponte entre o dispositivo e o sistema, permitindo que a **entrada dos dispositivos** tenha uma **saída uniforme** para o restante do sistema.

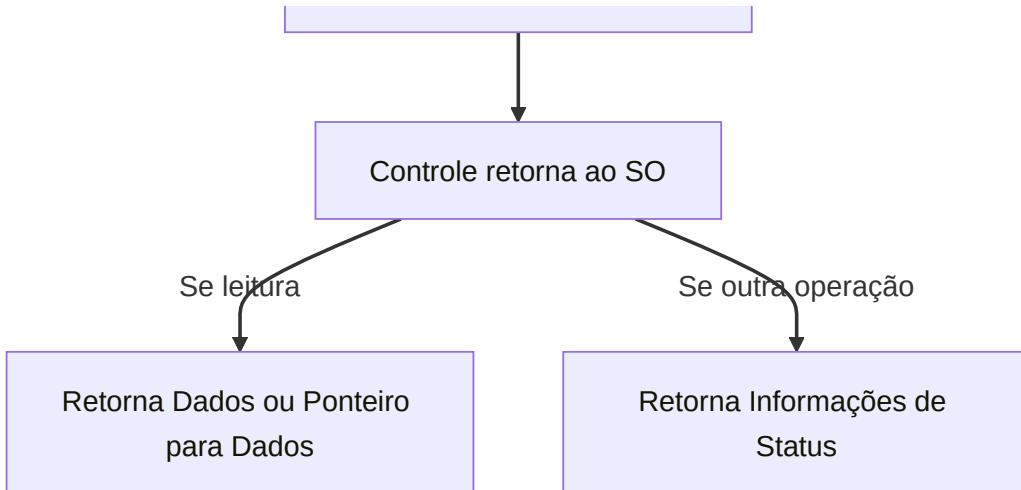
O funcionamento de uma operação de E/S:

- O **driver de dispositivo** carrega os **registradores** apropriados para dentro do **controlador do dispositivo**.
- O **controlador** examina o **conteúdo** que tem nos **registradores**, para determinar que ação deve ser tomada.

- O controlador começa a transferir os dados do dispositivo para o seu buffer local.
- Assim que a transferência está concluída, o **controlador de dispositivo** envia uma **interrupção** para o **driver de dispositivo** informando que a transferência foi concluída.
- O driver de dispositivo então retorna o controle diretamente para o SO, retornando os dados ou um ponteiro para esses dados, possivelmente, caso a operação seja de leitura.
 - Para outras operações, o driver retorna informações de status.

Representação:





- Para pequenas porções de dados, essa arquitetura de E/S por interrupção funciona bem, mas não funciona somente com isso há muito tempo, por isso, se usarmos essa forma para grandes volumes de dados como E/S de disco causa um **overhead** (que é uma sobrecarga).

Com esse grande problema, precisamos então de um outro dispositivo, um que armazene esses dados para que o acesso seja mais rápido, para isso usamos a **DAM** (Direct Access Memory ou Memória de Acesso Direto).

Logo o ciclo se torna assim:

- Depois de configurar buffers, ponteiros e contadores, o dispositivo de E/S, o controlador de dispositivo **move um bloco inteiro de dados** diretamente para ou do seu próprio buffer local para a memória.
 - Somente **uma interrupção é feita por bloco**, para que seja avisado ao driver de dispositivo que a **transferência foi concluída**.

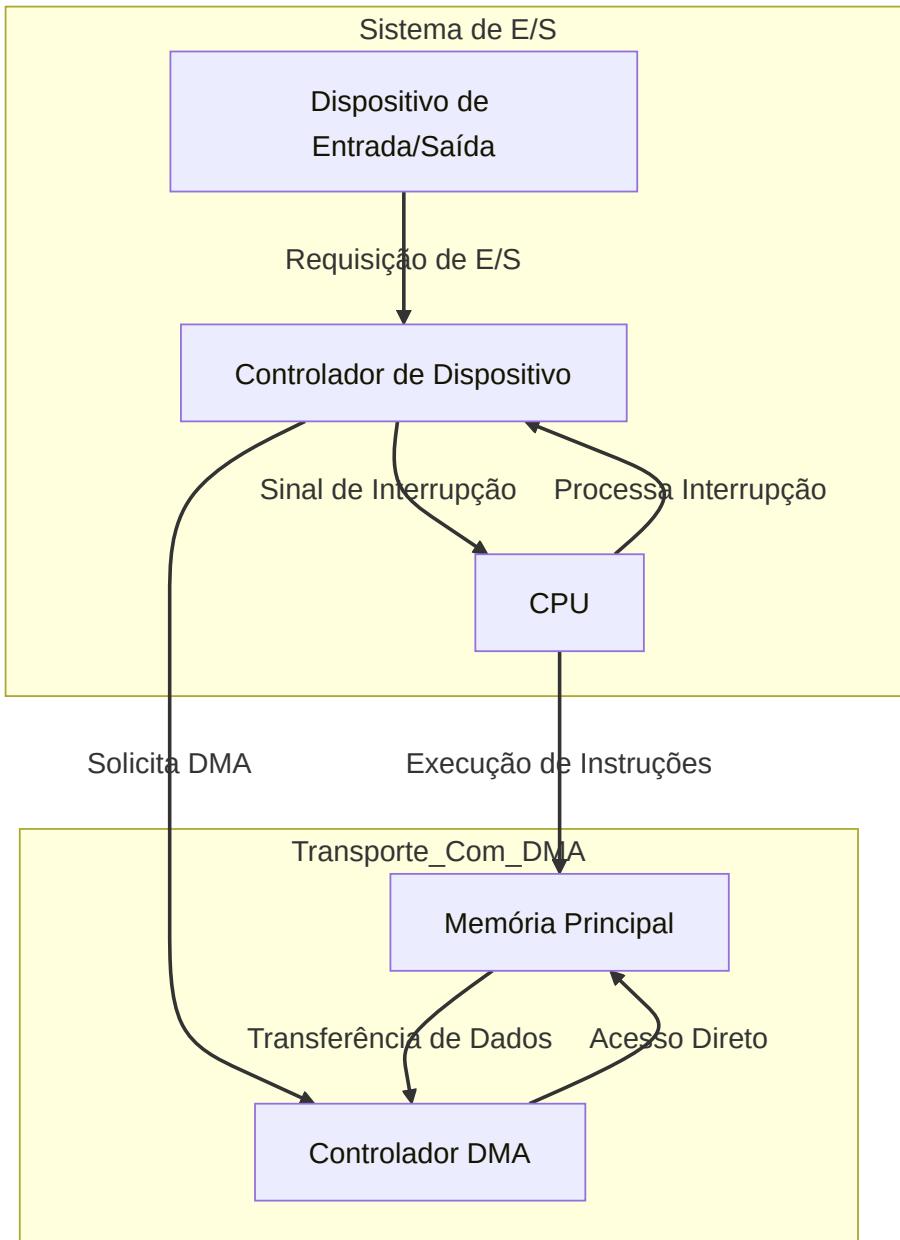
- Nesta etapa de transferência direta não ocorre intervenção da CPU, assim apenas o controlador de dispositivo cuida dessa tarefa.

Para alguns sistemas não é utilizado essa arquitetura de barramento e sim de switch:

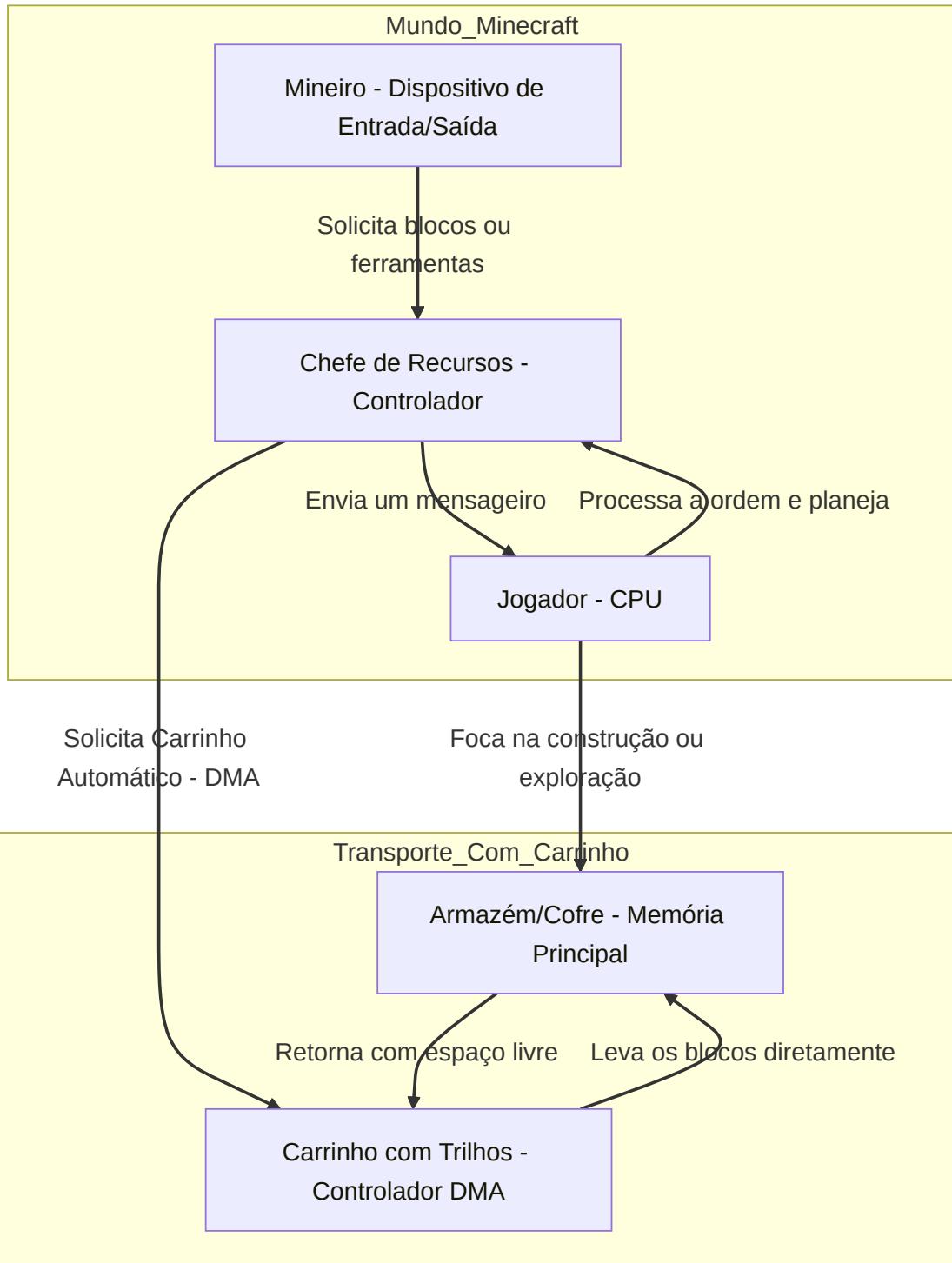
- Nesse tipo de sistema, os vários componentes do sistema podem interagir entre si ao mesmo tempo.
- Ao invés de competir por ciclos de um barramento compartilhado.

- Assim o DMA consegue ser ainda mais eficiente.

Representação da interação dos componentes num sistema:



- Com Mineiro:*



1.5 Arquitetura do Sistema

Agora falaremos sobre a categorização dos sistemas computadorizados, que é feita com base no número de processadores que ele possui, ou seja, estamos nos referindo a computadores de uso geral.

1.5.1 Sistema Monoprocessador

Esses sistemas, como o nome diz, possuem um único processador e foram muito utilizados, desde PDAs até mainframes. Assim, esses sistemas contêm uma única CPU que pode realizar diversas instruções de uso geral, assim como os processos do usuário.

- A maioria dos sistemas utiliza um processador de uso específico, como, por exemplo, para processamento gráfico, com os controladores gráficos, ou nos mainframes, com os processadores de E/S.

Esses processadores específicos não executam processos do usuário e somente realizam instruções limitadas e especializadas.

- Em alguns casos, o sistema operacional controla esse componente, pois o sistema envia informações sobre sua próxima tarefa e monitora seu status.

Exemplo:

- Um processador controlador de disco recebe uma sequência de requisições da CPU principal.
- Implementa sua própria fila de disco e algoritmo de escalonamento.

- Com isso, há um alívio na carga de processamento do escalonamento de disco, que, de outra forma, seria delegado à CPU principal.

O sistema operacional não pode se comunicar diretamente com esses processadores, pois eles operam em um nível mais baixo. Um exemplo disso são os teclados, que possuem um microprocessador responsável por converter os toques nas teclas em códigos que serão enviados para a CPU principal.

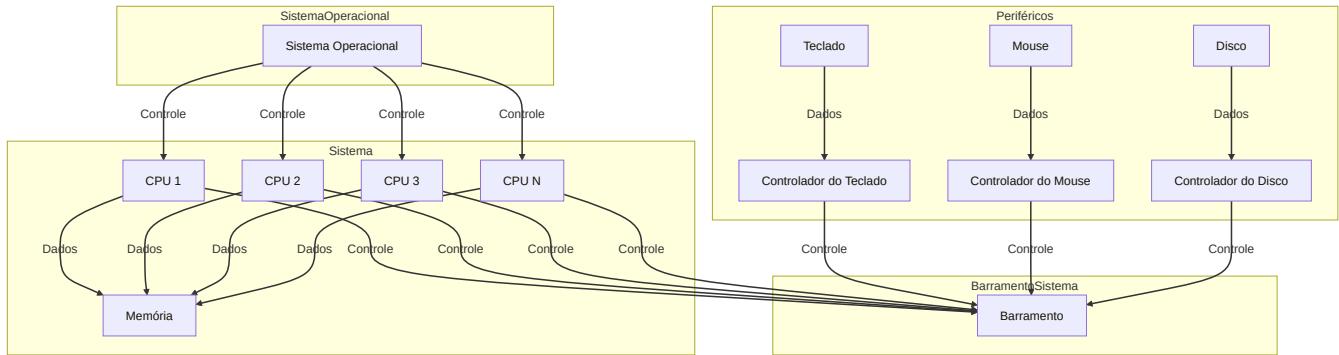
Assim, esses processadores realizam suas tarefas de forma anônima, pois não interagem diretamente com o sistema operacional.

Mesmo com o uso desses processadores específicos, o sistema ainda não é considerado

multiprocessado.

Para que um sistema seja classificado como monoprocessador, ele deve possuir uma única CPU de uso geral. Os processadores mencionados anteriormente são de uso específico.

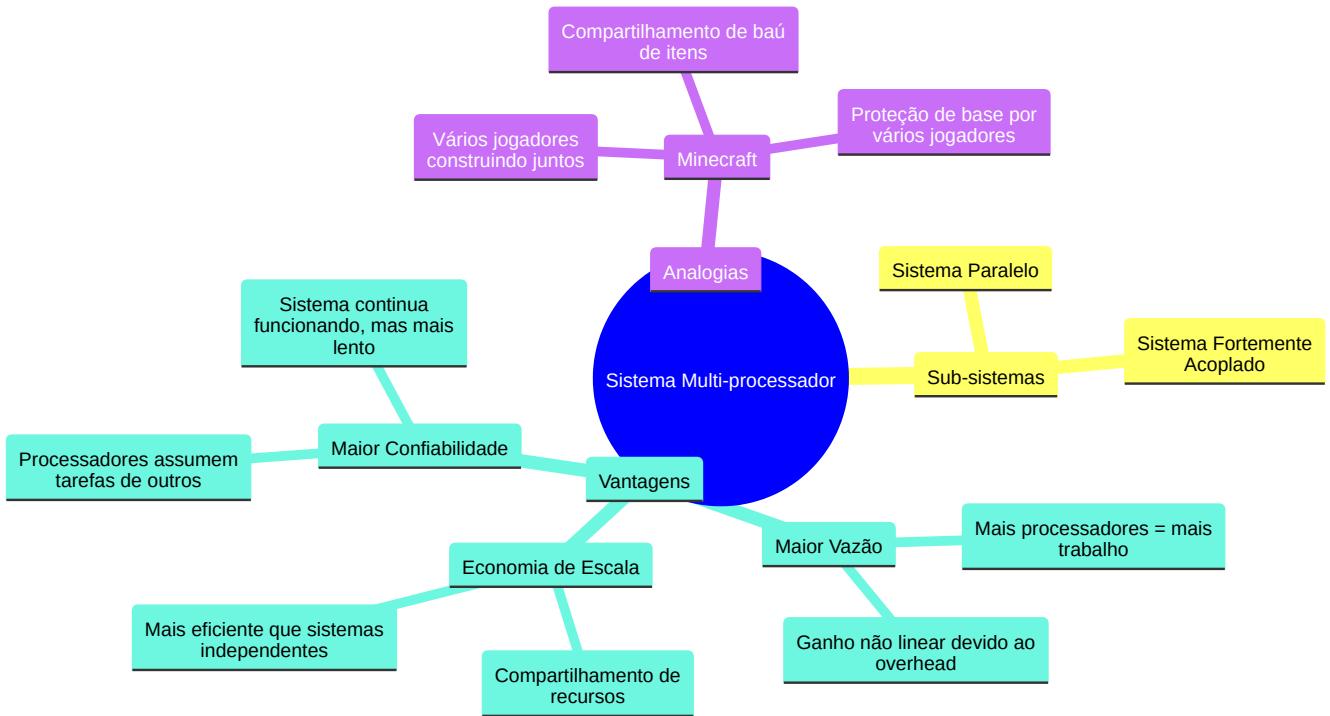
Diagrama



1.5.2 Sistema multi-processador

Esse tipo de sistema em que temos mais de um processador, de uso geral, dentro de um mesmo sistema computadorizado tem ganhado cada vez mais espaço por diversas razões o lugar dos sistemas mono processador.

Os sistemas multiprocessados, ou também conhecidos como: **sistemas paralelos** (parallel system) ou **sistema fortemente acoplado** (tightly coupled system) fazem um compartilhamento perfeito de periféricos, relógio do computador, barramento do computador para vários processadores de modo que a comunicação entre eles é perfeita.



Podemos escalar **três grandes vantagens** acerca desse tipo de arquitetura para sistemas:

1. Maior vazão:

- Como ter vários jogadores trabalhando juntos em uma construção no Minecraft.
- Mais processadores = mais trabalho realizado em menos tempo.
- Porém, o ganho não é linear devido ao overhead de coordenação.

2. Economia de escala:

- Semelhante a compartilhar um baú de itens entre vários jogadores no Minecraft.
- Sistemas multiprocessados compartilham recursos (periféricos, armazenamento, energia).
- Mais eficiente que ter vários sistemas independentes.

3. Maior confiabilidade:

- Como ter vários jogadores protegendo uma base no Minecraft.
- Se um processador falha, os outros podem assumir suas tarefas.
- O sistema continua funcionando, apenas mais lento, em vez de travar completamente.

Estas vantagens tornam os sistemas multiprocessados cada vez mais populares, assim como servidores de Minecraft com vários jogadores oferecem uma experiência mais robusta e dinâmica.

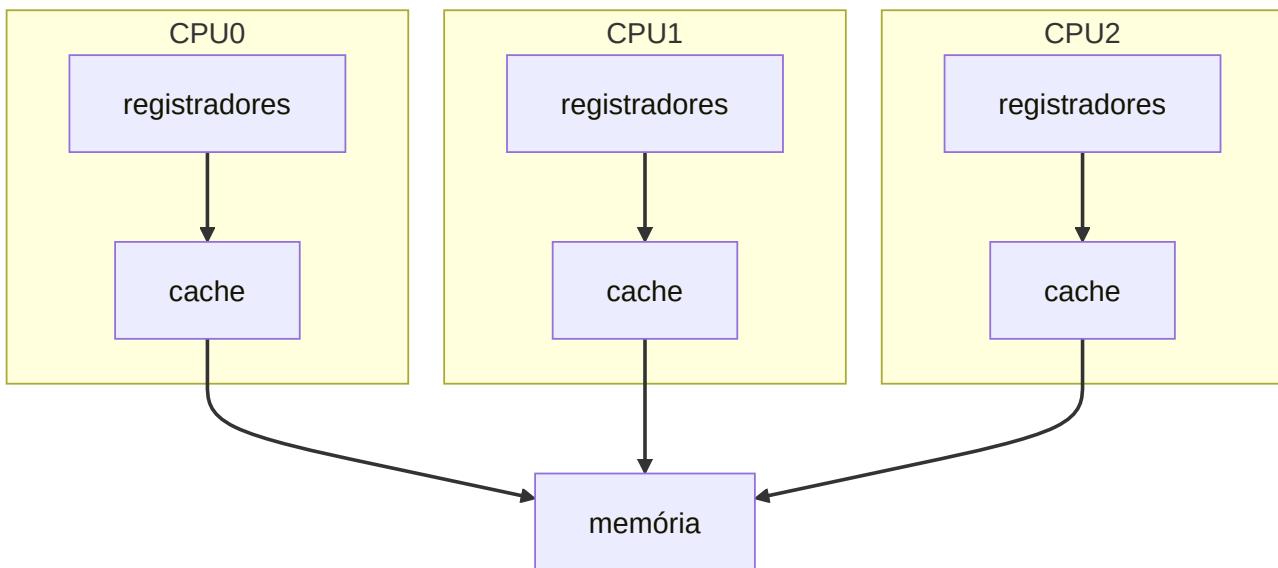
Imagine construir um mundo no Minecraft. A **confiabilidade** do sistema é como a estabilidade do mundo: se algo der errado (bloco sumir, mob bugar), ele continua funcionando, mesmo que limitado. Isso é **degradação controlada** — como minerar com uma ferramenta pior se a melhor quebrar.

Sistemas **tolerantes a falhas** vão além: mesmo com falhas, funcionam sem interrupções. No Minecraft, seria um backup automático que restaura blocos destruídos por creepers sem você sair do jogo.

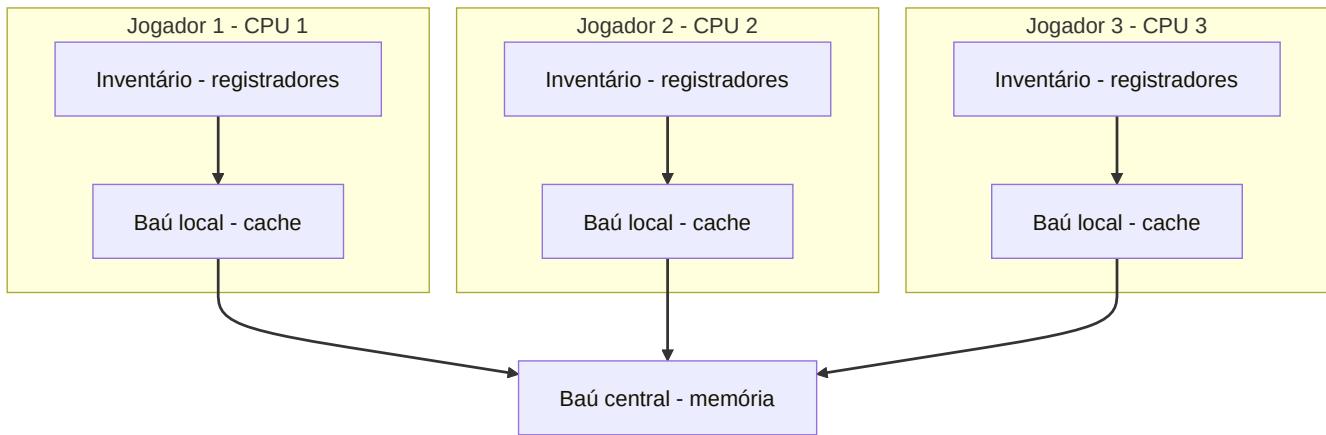
O **HP NonStop** é como um servidor com duplicação: dois jogadores (CPUs) constroem a mesma coisa ao mesmo tempo. Se um errar, o sistema corrige e transfere a tarefa para outro par, garantindo continuidade, mas com custo maior.

Já os **sistemas multiprocessados** são como vários jogadores trabalhando juntos:

1. **Assimétrico**: Um jogador mestre comanda os outros. Se ele sair, tudo pode parar.
2. **Simétrico (SMP)**: Todos são iguais, compartilham recursos (baú/memória) e trabalham juntos sem perder desempenho. Sistemas como **Solaris**, Windows e Linux usam isso.



- Com Minecraft:



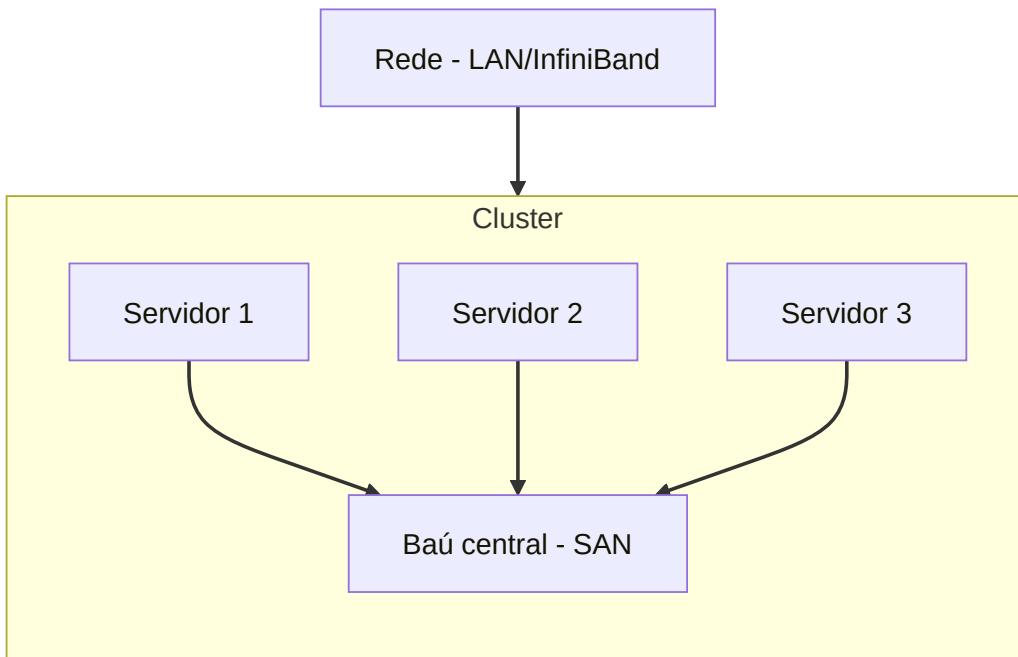
1.5.3 Sistemas em Clusters

Resumo com analogias ao Minecraft:

Um **sistema em cluster** é como um grupo de servidores de Minecraft trabalhando juntos. Cada servidor (nó) é independente, mas eles estão conectados por uma rede (LAN ou conexão rápida) e compartilham armazenamento (como um baú central). O objetivo é garantir **alta disponibilidade e alto desempenho**.

- **Alta disponibilidade:** Se um servidor falhar (explodir como um creeper), outro assume seu lugar, mantendo o mundo (serviço) funcionando com pouca interrupção.
- **Modo assimétrico:** Um servidor fica de olho (hot-standby) enquanto o outro roda o jogo. Se o ativo falhar, o standby assume.
- **Modo simétrico:** Vários servidores rodam o jogo e se monitoram, usando todo o hardware de forma eficiente.
- **Alto desempenho:** Vários servidores podem trabalhar juntos para resolver tarefas complexas, como gerar chunks ou processar comandos em paralelo. Isso exige que o jogo (aplicação) seja dividido em partes que rodam simultaneamente em diferentes servidores.
- **Clusters paralelos:** Vários servidores acessam os mesmos dados (como um banco de dados compartilhado). Para evitar conflitos, um sistema de "trava" (DLM) garante que apenas um servidor modifique os dados por vez.
- **SANs (Storage-Area Networks):** É como um baú gigante conectado a todos os servidores. Se um servidor cair, outro pode pegar os itens (dados) e continuar o jogo.

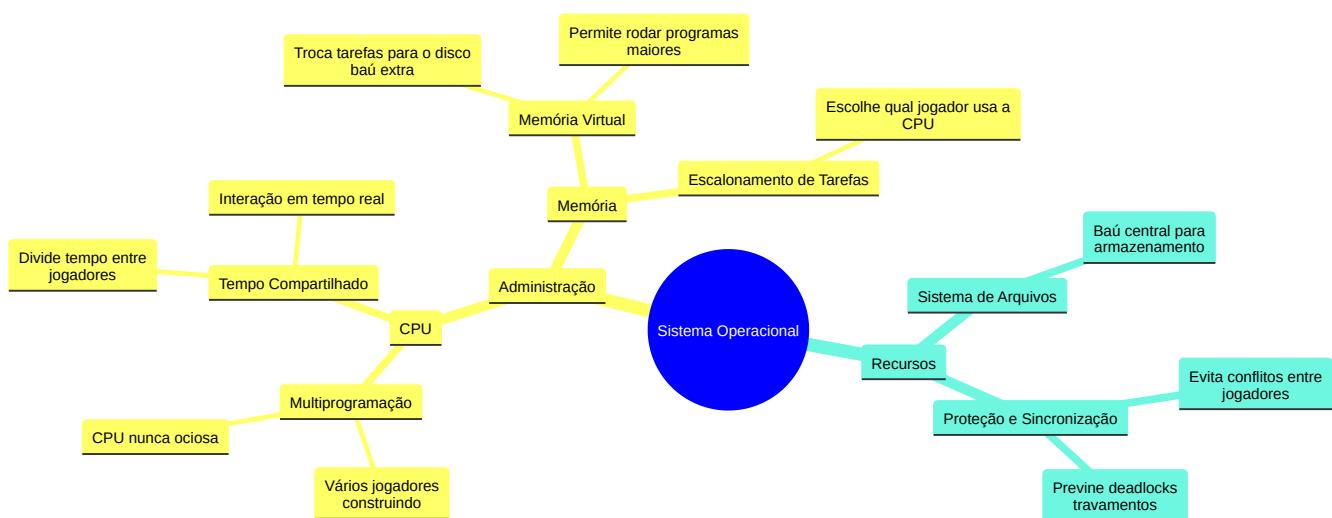
Resumo visual:



1.6 Estrutura do sistema operacional

Um **sistema operacional** é como o "administrador" de um servidor de Minecraft. Ele gerencia recursos (CPU, memória, dispositivos) e permite que vários programas (ou jogadores) funcionem ao mesmo tempo.

- **Multiprogramação:** É como ter vários jogadores construindo no mesmo mundo. Se um jogador precisa esperar (por exemplo, para minerar), o sistema passa para outro, mantendo a CPU sempre ocupada. Isso evita que o servidor fique ocioso.
- **Tempo compartilhado (time sharing):** É como dividir o tempo do servidor entre vários jogadores. Cada um recebe um pouco de atenção do servidor, mas tão rápido que parece que todos estão jogando ao mesmo tempo. Isso permite interação em tempo real, como digitar comandos e ver resultados imediatos.
- **Escalonamento de tarefas:** O sistema escolhe qual jogador (tarefa) deve usar o servidor (CPU) a seguir, garantindo que todos tenham uma chance justa.
- **Memória virtual:** Se o servidor não tem espaço para todos os jogadores (tarefas) na memória, ele "troca" alguns para o disco (como um baú extra) e os traz de volta quando necessário. Isso permite rodar programas maiores do que a memória física.
- **Sistema de arquivos:** É como o baú central do servidor, onde todos os itens (arquivos) são armazenados e organizados.
- **Proteção e sincronização:** O sistema garante que os jogadores (tarefas) não interfiram uns com os outros, evitando conflitos e travamentos (deadlocks).



1.7 Operações do Sistema Operacional

Resumo com analogias ao Minecraft:

O sistema operacional é como o "administrador" de um servidor de Minecraft, controlando tudo que acontece no mundo (sistema). Ele usa **interrupções** e **traps** para lidar com eventos, como um jogador tentando fazer algo que não deveria (erro) ou pedindo ajuda (chamada de sistema).

1. Modo Dual (Usuário e Kernel):

- **Modo Usuário:** Onde os jogadores (programas de usuário) operam. Eles têm permissão limitada, como construir ou minerar, mas não podem alterar o servidor diretamente.
- **Modo Kernel:** Onde o administrador (sistema operacional) opera. Ele tem controle total sobre o servidor, como gerenciar recursos, corrigir erros ou expulsar jogadores problemáticos.
- **Transição:** Quando um jogador precisa de algo que só o administrador pode fazer (como abrir um portal), ele faz uma **chamada de sistema**, e o servidor muda para o modo kernel temporariamente.

2. Proteção:

- O sistema operacional protege o servidor de jogadores mal-intencionados ou erros. Por exemplo, se um jogador tentar destruir o servidor (executar uma instrução privilegiada no modo usuário), o sistema bloqueia a ação e notifica o administrador.

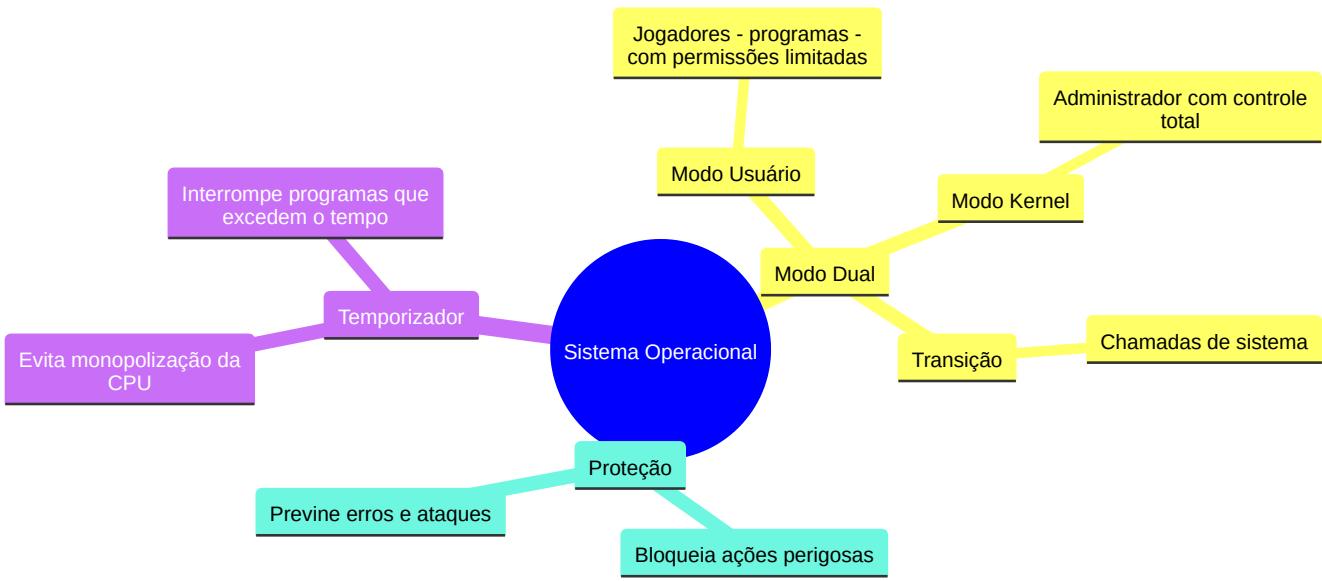
3. Temporizador:

- Para evitar que um jogador monopolize o servidor (loop infinito), o sistema usa um **temporizador**. Se um jogador ficar muito tempo sem ceder a vez, o sistema interrompe e passa o controle para outro jogador ou para o administrador.

4. Ciclo de Execução:

- O sistema operacional começa no modo kernel (administrador) ao ligar o servidor. Ele carrega os jogadores (programas) no modo usuário e alterna entre os modos conforme necessário, garantindo que tudo funcione sem problemas.

Resumo visual:



Em resumo, o sistema operacional é como um administrador de servidor de Minecraft, alternando entre modos para garantir que os jogadores (programas) possam jogar sem causar problemas, enquanto mantém o controle total sobre o sistema.

1.8 Gerência de processos

Um **processo** é como um jogador em um servidor de Minecraft. Um programa (arquivo no disco) é só um conjunto de instruções, mas quando ele é executado, vira um processo (jogador ativo). Cada processo precisa de recursos como tempo de CPU (atenção do servidor), memória (espaço no inventário), e dispositivos de E/S (ferramentas e blocos).

1. Processo vs. Programa:

- **Programa:** É como um livro de instruções para construir algo no Minecraft (passivo).
- **Processo:** É um jogador seguindo essas instruções e construindoativamente (ativo).

2. Recursos do Processo:

- Cada processo (jogador) recebe recursos do sistema operacional (administrador do servidor), como tempo de CPU, memória e acesso a arquivos ou dispositivos.
- Quando o processo termina (jogador sai), os recursos são devolvidos ao sistema.

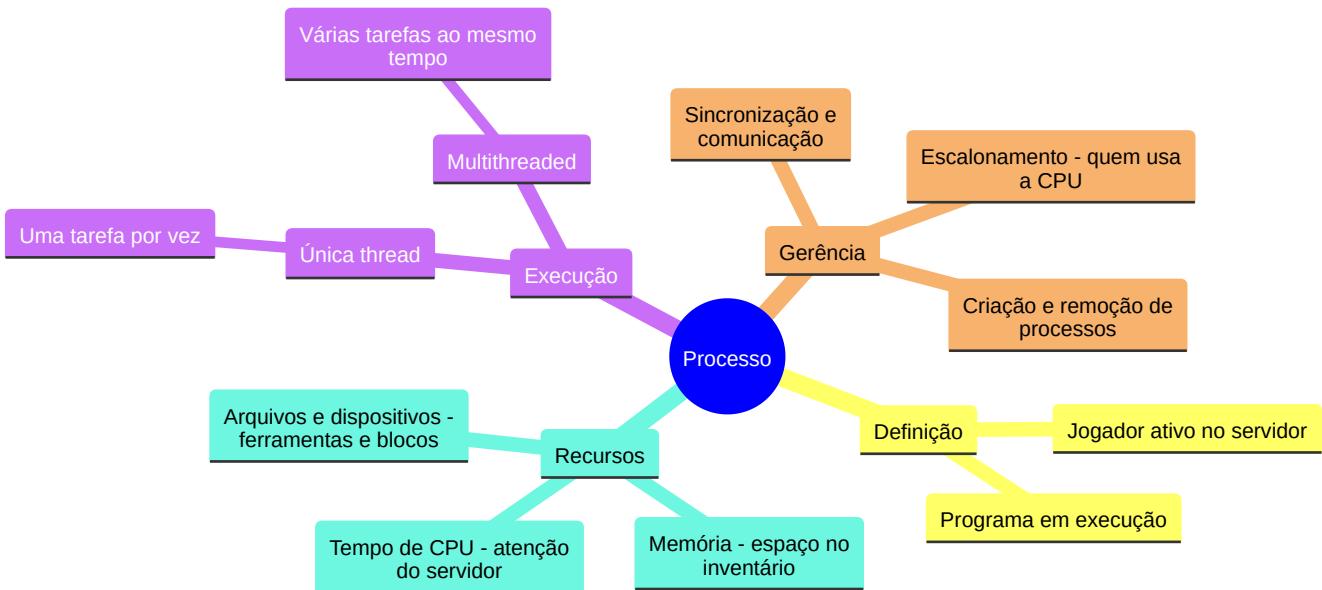
3. Execução de Processos:

- Um processo de **única thread** é como um jogador com uma única tarefa, seguindo uma sequência de instruções (contador de programa).
- Um processo **multithreaded** é como um jogador com várias tarefas ao mesmo tempo (vários contadores de programa).

4. Gerência de Processos:

- O sistema operacional (administrador) gerencia os processos (jogadores), decidindo quem usa a CPU (escalonamento), criando ou removendo processos, e garantindo que eles não interfiram uns com os outros (sincronização e comunicação).

Resumo visual:



Em resumo, um processo é como um jogador ativo no servidor de Minecraft, usando recursos e seguindo instruções. O sistema operacional é o administrador que gerencia todos os jogadores, garantindo que tudo funcione sem problemas.

1.9 Gerência de memória

Resumo com analogias ao Minecraft:

A **memória principal** é como o inventário do jogador no Minecraft. Ela armazena dados e instruções que a CPU (jogador) precisa para executar tarefas rapidamente. Assim como o inventário tem espaço limitado, a memória principal também tem um tamanho finito e precisa ser gerenciada com cuidado.

1. Função da Memória Principal:

- É o "inventário" do computador, onde a CPU busca instruções e dados para executar programas.
- Para que um programa rode, ele precisa ser carregado na memória, como colocar itens no inventário.

2. Acesso Direto:

- A CPU só pode acessar diretamente a memória principal. Dados de dispositivos como discos (baús externos) precisam ser transferidos para a memória antes de serem usados.

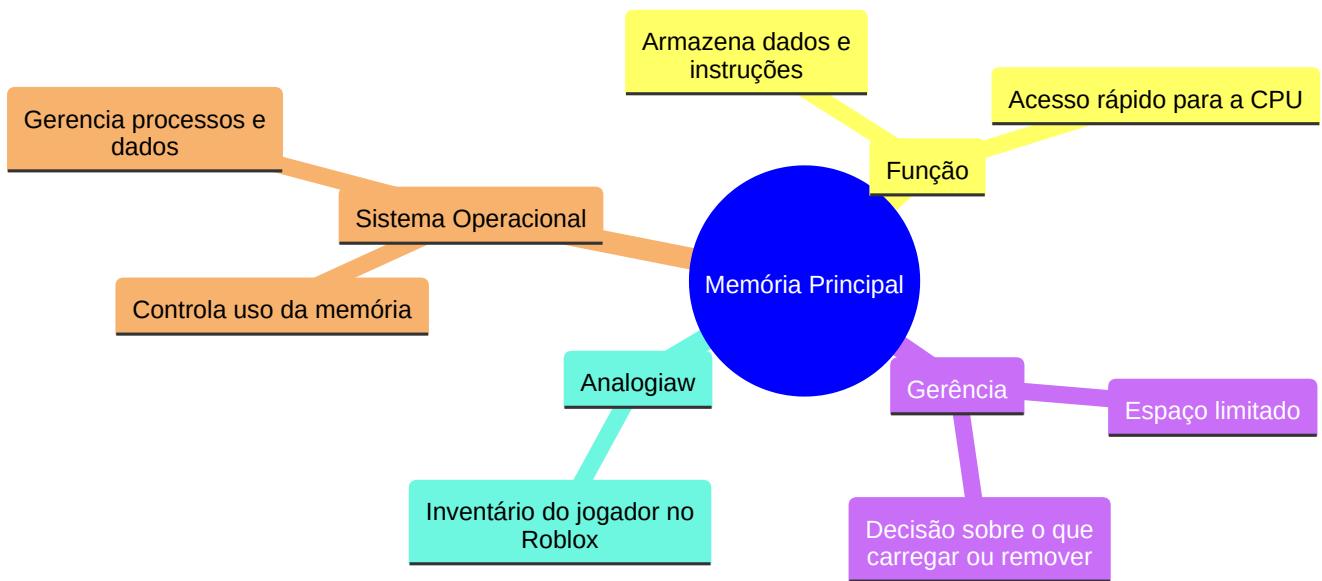
3. Gerência de Memória:

- O sistema operacional (administrador) gerencia o espaço na memória, decidindo quais programas (itens) ficam na memória e quais são removidos quando o espaço acaba.
- Isso é crucial para manter vários programas rodando ao mesmo tempo, como ter vários itens no inventário para diferentes tarefas.

4. Atividades do Sistema Operacional:

- Controlar quais partes da memória estão em uso e por quem.
- Decidir quais processos (tarefas) e dados devem ser carregados ou removidos da memória.

Resumo visual:



1.10 Gerência de armazenamento



O sistema operacional fornece uma visão lógica e uniforme do armazenamento de informações, abstraindo as propriedades físicas dos dispositivos de armazenamento. Ele define uma unidade de armazenamento lógica chamada **arquivo**, que é mapeada no meio físico e acessada por dispositivos de armazenamento.

Analogia com Roblox: Imagine o Roblox como um sistema operacional. Ele gerencia todos os itens, skins, mapas e scripts que você usa nos jogos. Esses itens são como "arquivos" que o Roblox organiza e torna acessíveis para você, independentemente de onde eles estejam armazenados fisicamente (servidores, nuvem, etc.).

1.10.1 Gerência de Sistema de Arquivos

A gerência de arquivos é uma parte visível do sistema operacional, responsável por organizar e controlar o acesso a arquivos e diretórios. Os arquivos podem ser de vários tipos (texto, binários, etc.) e são armazenados em diferentes mídias (discos magnéticos, ópticos, fitas). O sistema operacional gerencia a criação, remoção, organização e acesso a esses arquivos.

Analogia com Roblox: No Roblox, você tem uma "pasta" de inventário onde todos os seus itens (arquivos) são organizados. Alguns itens são raros (como arquivos importantes), outros são comuns (como arquivos de texto). O Roblox também controla quem pode acessar seus itens (leitura, escrita, remoção), assim como um sistema operacional faz com arquivos.

1.10.2 Gerência de Armazenamento em Massa

Como a memória principal é limitada e volátil, o armazenamento secundário (como discos) é essencial para guardar programas e dados. O sistema operacional gerencia o espaço livre, a

alocação de armazenamento e o escalonamento do disco para garantir eficiência. Além disso, há o armazenamento terciário (como fitas e CDs), usado para backups e dados raramente acessados.

Analogia com Roblox: Pense no armazenamento secundário como o seu "inventário principal" no Roblox, onde você guarda os itens que usa com frequência. Já o armazenamento terciário seria como um "baú de tesouro" onde você guarda itens raros ou que não usa muito (como skins antigas ou itens de eventos passados). O Roblox gerencia esses espaços para que você possa acessá-los quando precisar.

1.10.3 Caching

O **caching** é um conceito essencial para entender como os sistemas computadorizados otimizam o acesso a informações. Ele funciona como uma camada intermediária de armazenamento rápido, reduzindo o tempo de acesso a dados frequentemente utilizados.

Como funciona:

1. Armazenamento de Informações:

- As informações são armazenadas em dispositivos como a memória principal.
- Quando acessadas, são copiadas temporariamente para uma memória mais rápida, chamada **cache**.

2. Busca de Dados:

- Ao buscar uma informação, o sistema primeiro verifica se ela está no cache.
 - Se estiver (**cache hit**), os dados são usados diretamente do cache.
 - Se não estiver (**cache miss**), o sistema busca a informação na memória principal (ou secundária) e a copia para o cache, acelerando futuros acessos.

3. Registradores e Algoritmos:

- Registradores (como os de índice) são gerenciados por algoritmos que decidem quais dados manter no cache e quais enviar para a memória principal.
- Esses algoritmos são implementados por programadores, compiladores ou diretamente no hardware.

4. Cache de Instruções:

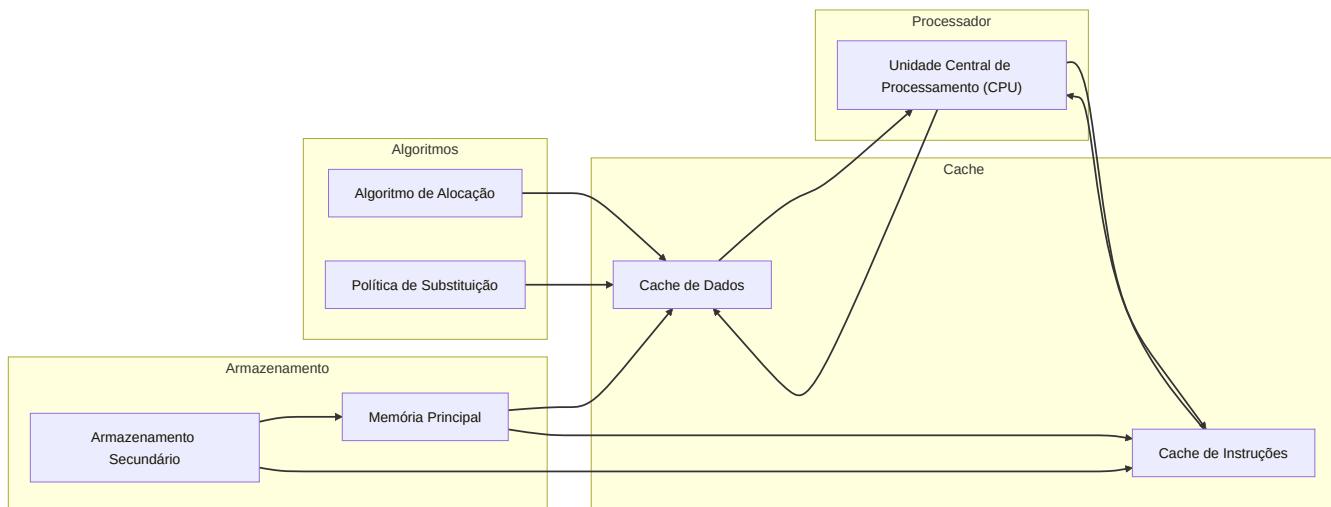
- Muitos sistemas possuem um **cache de instruções**, que armazena as próximas instruções a serem executadas pela CPU.
- Isso evita que a CPU perca ciclos buscando instruções na memória principal.

5. Hierarquia de Memórias:

- O cache está no topo da hierarquia de memórias, sendo a mais rápida, porém com capacidade limitada.
- Abaixo dele estão a memória principal e o armazenamento secundário (discos, SSDs).

6. Gerenciamento de Cache:

- Como o cache tem tamanho reduzido, seu gerenciamento é crucial. Isso inclui:
 - Definir o **tamanho do cache**.
 - Estabelecer a **política de substituição** (ex.: LRU - Least Recently Used) para decidir quais dados remover quando o cache estiver cheio.



Esses fatores podem **melhorar o desempenho da memória cache**.

A **memória principal** pode ser vista como um **cache rápido para o armazenamento secundário**, pois os dados precisam ser copiados da memória secundária para a principal antes de serem utilizados.

De forma recíproca, para serem **movidos para a memória secundária**, os dados **precisam estar primeiro na memória principal**, garantindo proteção e integridade.

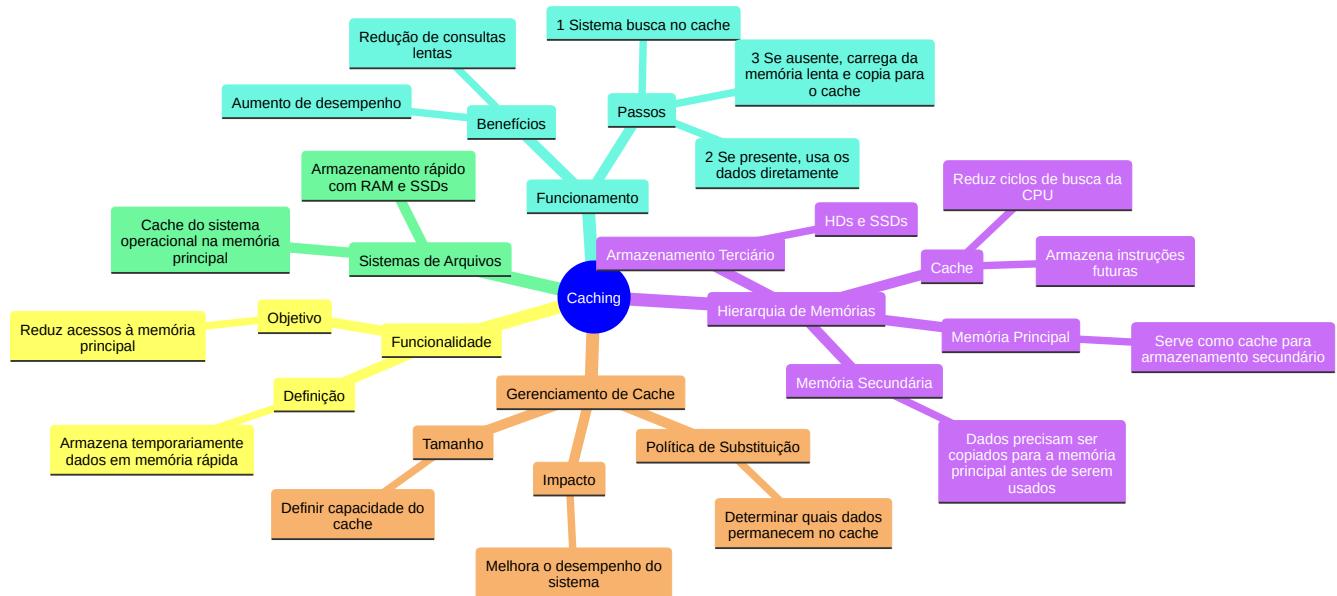
O sistema de arquivos vê os dados permanentemente gravados no armazenamento secundário de forma hierárquica, existindo *diversos níveis na hierarquia*:

- No nível mais alto -> o **sistema operacional** pode **manter um cache do sistema de arquivos na memória principal**.

Também é **possível que memórias RAM, como discos de estado sólido** (ou então discos eletrônicos de RAM), sejam usadas para **armazenamento de alta velocidade**, acessados pela **interface do**

sistema de arquivos. Isso significa que a comunicação deve ser feita diretamente com o sistema de arquivos.

Atualmente, a maior parte do **armazenamento terciário** consiste em **HDs ou SSDs**.



Níveis e o Cache

Os **movimentos** de informações entre os **níveis da hierarquia de memórias** podem ser de dois tipos: **explícitos** e **implícitos**. Isso depende da arquitetura do **hardware** e do **software** que controla o sistema operacional.

Podemos exemplificar essa questão:

- A **transferência de dados entre a cache e a CPU e seus registradores**-> ocorre diretamente no **hardware**, sem intervenção do sistema operacional.
- A **transferência de dados do disco para a memória RAM**-> normalmente é controlada pelo **sistema operacional**.

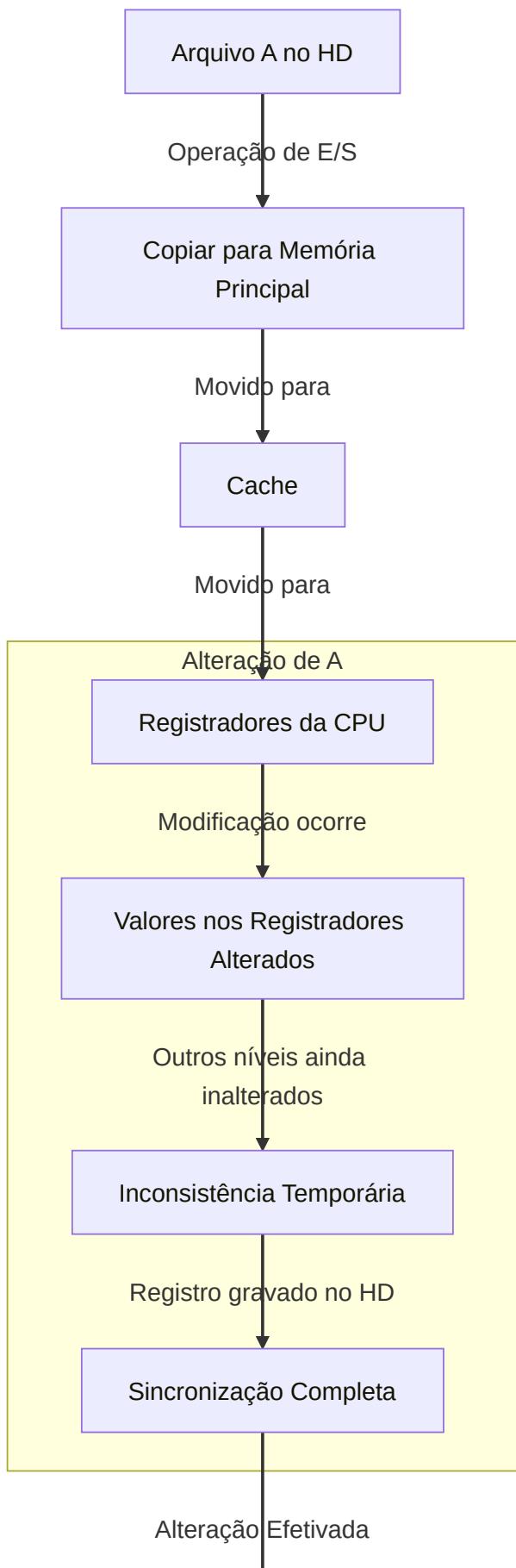
Como, nessa estrutura hierárquica, os mesmos dados podem aparecer em diferentes níveis de armazenamento, vejamos um exemplo:

- Suponha que um texto no arquivo A precise ser alterado para um outro valor no arquivo B, que reside no HD.
- Antes da alteração, o sistema precisa emitir uma operação de E/S para copiar o bloco de disco contendo A para a memória principal.
- Em seguida, o arquivo A será copiado para o cache e para os registradores internos da CPU.

- Assim, a cópia de A estará presente em vários níveis, conforme mostrado abaixo:



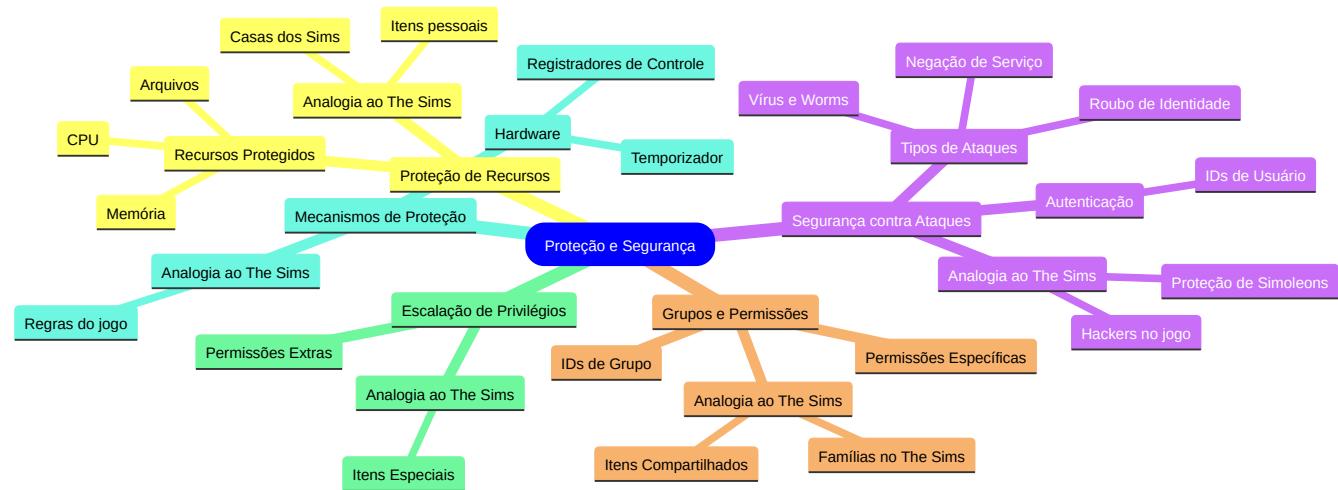
- Quando a alteração for feita nos registradores internos da CPU, os valores de A serão diferentes nos outros níveis de armazenamento, que permanecerão inalterados.
- Somente quando o registrador gravar a mudança no disco rígido (memória secundária), os valores nos diferentes níveis estarão sincronizados, tornando a alteração efetiva.



Valores Iguais em Todos os
Níveis



1.11 Proteção e Segurança



1. Proteção de Recursos:

- Em um sistema com múltiplos usuários e processos, o acesso aos recursos (arquivos, memória, CPU) precisa ser controlado. O sistema operacional garante que apenas processos autorizados possam acessar esses recursos.
- Analogia ao The Sims:** Imagine que cada Sim (usuário) tem sua própria casa (espaço de memória) e itens (recursos). O jogo impede que um Sim entre na casa de outro ou use seus itens sem permissão.

2. Mecanismos de Proteção:

- O hardware e o sistema operacional trabalham juntos para proteger recursos. Por exemplo, o temporizador impede que um processo monopolize a CPU, e os registradores de controle de dispositivo protegem periféricos.
- Analogia ao The Sims:** No jogo, há regras que impedem que um Sim fique indefinidamente em uma atividade (como cozinhar ou dormir), garantindo que outros Sims também tenham acesso aos recursos.

3. Segurança contra Ataques:

- A segurança protege o sistema contra ataques externos e internos, como vírus, roubo de identidade e negação de serviço. A autenticação (IDs de usuário) é usada para garantir que apenas usuários autorizados acessem o sistema.
- Analogia ao The Sims:** Imagine que um "hacker" tenta invadir o jogo e roubar os Simoleons (moeda do jogo) de um Sim. O sistema de segurança do jogo (como senhas ou autenticação em dois fatores) impede isso.

4. Grupos e Permissões:

- Os sistemas operacionais usam IDs de usuário e grupo para controlar permissões. Um usuário pode pertencer a um ou mais grupos, e cada grupo tem permissões específicas.
- **Analogia ao The Sims:** No jogo, você pode criar famílias (grupos) e definir quais Sims têm permissão para usar certos itens ou áreas da casa.

5. Escalação de Privilégios:

- Às vezes, um usuário precisa de permissões extras para realizar tarefas específicas. O sistema operacional permite essa escalação de forma controlada.
- **Analogia ao The Sims:** Se um Sim precisa usar um item especial (como um objeto de magia), ele pode ganhar permissões temporárias para acessá-lo.

1.12 Sistemas de uso específico

1.11 Sistemas de Uso Específico

Os sistemas computadorizados de uso específico são projetados para tarefas especializadas e limitadas, diferindo dos sistemas de uso geral que estamos acostumados a utilizar. Eles são amplamente empregados em dispositivos embutidos, como eletrodomésticos inteligentes, carros autônomos, drones e dispositivos IoT (Internet das Coisas).

1.11.1 Sistemas de Tempo Real Embutidos

- **O que são?** Sistemas embutidos são computadores dedicados a tarefas específicas, como controlar motores de carros, robôs industriais, drones ou até mesmo dispositivos domésticos inteligentes, como assistentes virtuais (Alexa, Google Home) e termostatos (Nest). Eles operam em **tempo real**, o que significa que precisam responder a eventos dentro de um tempo definido, ou o sistema falha.
- **Analogia ao Minecraft:** Imagine um **redstone circuit** no Minecraft. Ele é projetado para realizar uma tarefa específica, como abrir uma porta automaticamente quando um jogador se aproxima. Se o circuito não responder imediatamente, a funcionalidade falha. Assim como um sistema de tempo real, o circuito de redstone tem "restrições de tempo" para funcionar corretamente.
- **Exemplos Modernos:**
 - Carros autônomos (como um **redstone contraption** que controla um veículo automático no Minecraft).
 - Drones (como um **dispenser** que lança foguetes no tempo exato).
 - Dispositivos IoT (como um **sensor de movimento** no Minecraft que acende luzes automaticamente).

1.11.2 Sistemas Multimídia

- **O que são?** Sistemas multimídia lidam com dados como áudio, vídeo e realidade aumentada (AR), que precisam ser entregues em "streaming" com restrições de tempo (ex.: 60 frames por segundo para jogos ou vídeos 4K). Eles são usados em aplicações como videoconferências (Zoom, Teams), streaming (Netflix, YouTube) e realidade virtual (VR).
- **Analogia ao Minecraft:** Pense em um **mapa de aventura** no Minecraft com cutscenes (cenas pré-gravadas). Para que a experiência seja imersiva, as cenas precisam ser exibidas sem atrasos,

assim como um vídeo precisa ser reproduzido sem travamentos. Se o sistema não conseguir entregar os frames no tempo certo, a experiência é prejudicada.

- **Exemplos Modernos:**

- Streaming de jogos (como o **Minecraft RTX**, que exige alta performance gráfica).
- Realidade virtual (como um **mundo VR** no Minecraft).
- Videoconferências (como um **evento ao vivo** no servidor de Minecraft com transmissão em tempo real).

1.11.3 Sistemas Portáteis

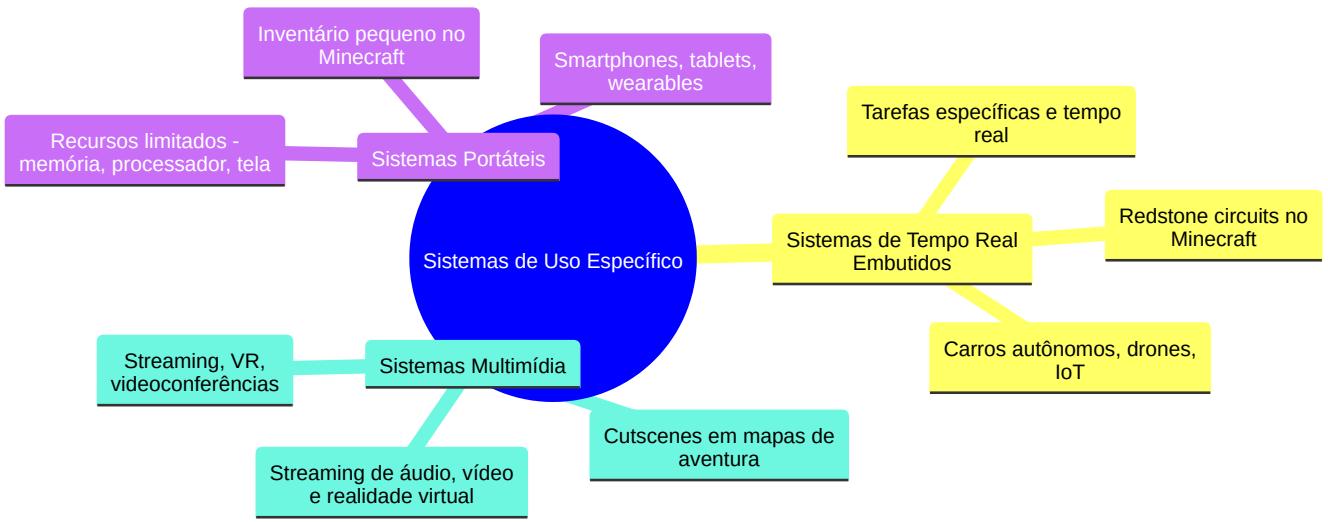
- **O que são?** Sistemas portáteis, como smartphones, tablets e wearables (smartwatches), têm recursos limitados devido ao seu tamanho reduzido. Eles possuem pouca memória, processadores eficientes (mas não tão potentes quanto desktops) e telas pequenas, mas são altamente convenientes e portáteis.
- **Analogia ao Minecraft:** Imagine um **inventário de Minecraft**. Ele tem espaço limitado, então você precisa gerenciar os itens com cuidado, priorizando o que é mais importante. Assim como um smartphone, o inventário é pequeno, mas essencial para a jogabilidade.

- **Desafios Modernos:**

- Memória limitada (como um **baú pequeno** no Minecraft, mas com otimizações para armazenar mais itens).
- Processadores eficientes (como jogar Minecraft no **celular** com gráficos reduzidos para evitar lag).
- Telas pequenas (como a interface compacta do **Minecraft Pocket Edition**).

- **Tecnologias Atuais:**

- Smartphones com 5G (como um **servidor de Minecraft** com conexão ultrarrápida).
- Wearables (como um **smartwatch** que monitora sua saúde enquanto você joga).
- Tablets (como jogar Minecraft em um **iPad** com tela maior e portabilidade).



1.13 Ambientes de Computação

1.12.1 Computação Tradicional

- **O que é?** A computação tradicional refere-se ao uso de PCs, servidores e mainframes em ambientes como escritórios e residências. Antigamente, os sistemas eram centralizados, com terminais conectados a mainframes ou PCs ligados a redes locais. Hoje, a computação tradicional se expandiu com o uso de tecnologias web, dispositivos portáteis e conexões de alta velocidade.
- **Evolução:**
 - **Antes:** Sistemas em lote (batch) e interativos, com tempo compartilhado para otimizar recursos.
 - **Hoje:** PCs potentes, laptops, tablets e smartphones com acesso remoto e portabilidade.
 - **Tendências:** Portais web, sincronização de dispositivos e redes domésticas inteligentes.
- **Exemplos Modernos:**
 - **Escritórios:** Uso de laptops, desktops e servidores em nuvem (como Google Workspace ou Microsoft 365).
 - **Residências:** Redes domésticas com dispositivos IoT (smart TVs, assistentes virtuais) e conexões de alta velocidade (fibra óptica, 5G).

1.12.2 Sistemas Cliente-Servidor

- **O que é?** Neste modelo, os sistemas são divididos em dois papéis:
 - **Cliente:** Solicita serviços (ex.: navegador web).
 - **Servidor:** Fornece serviços (ex.: servidor de arquivos ou banco de dados).
- **Tipos de Servidores:**
 - **Servidor de Processamento (Compute-Server):** Executa ações e retorna resultados (ex.: servidor de banco de dados).
 - **Servidor de Arquivos (File-Server):** Gerencia arquivos e os disponibiliza para clientes (ex.: servidor web).

- **Vantagens:**
 - Centralização de recursos e dados.
 - Facilidade de gerenciamento e segurança.
- **Exemplos Modernos:**
 - Serviços em nuvem (AWS, Google Cloud).
 - Aplicações web (Netflix, Spotify).

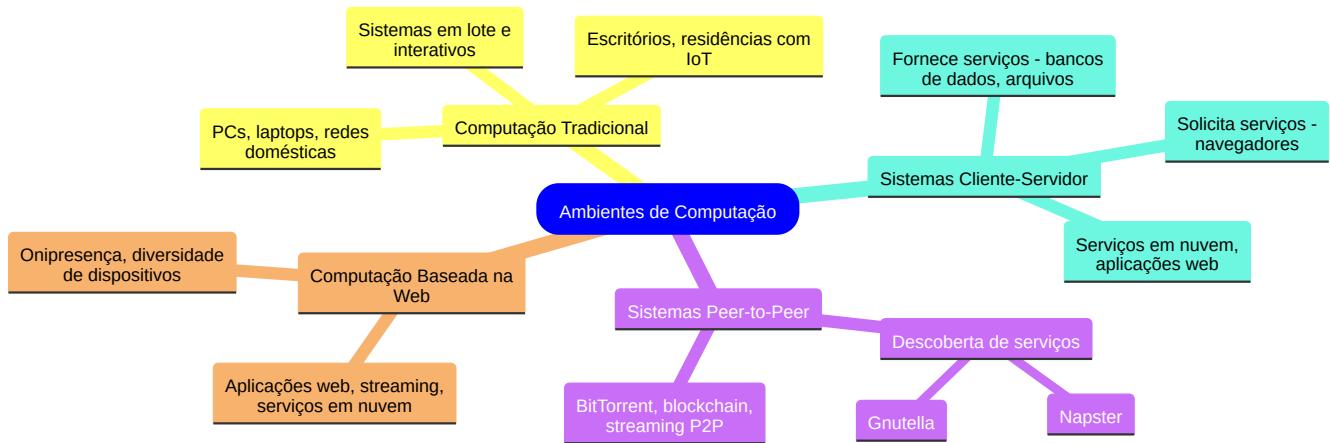
1.12.3 Sistemas Peer-to-Peer (P2P)

- **O que é?** No modelo P2P, todos os nós (dispositivos) na rede são iguais, podendo atuar como clientes e servidores. Não há centralização, e os serviços são distribuídos entre os nós.
- **Funcionamento:**
 - **Descoberta de Serviços:**
 - **Centralizada:** Um servidor central mantém um índice de serviços (ex.: Napster).
 - **Descentralizada:** Os nós enviam requisições por broadcast (ex.: Gnutella).
- **Vantagens:**
 - Eliminação de gargalos (não há um único servidor).
 - Escalabilidade e resiliência.
- **Exemplos Modernos:**
 - Compartilhamento de arquivos (BitTorrent).
 - Criptomoedas (blockchain, Bitcoin).
 - Streaming P2P (ex.: plataformas de vídeo descentralizadas).

1.12.4 Computação Baseada na Web

- **O que é?** A computação baseada na web transformou a forma como acessamos e utilizamos recursos computacionais. Ela permite o acesso a serviços e dados por meio de navegadores e dispositivos conectados à internet.
- **Características:**

- **Onipresença:** Acesso de qualquer lugar, a qualquer hora.
- **Diversidade de Dispositivos:** PCs, smartphones, tablets, IoT.
- **Conectividade:** Redes sem fio (Wi-Fi, 5G) e balanceadores de carga para distribuição de tráfego.
- **Exemplos Modernos:**
 - Aplicações web (Google Docs, Figma).
 - Plataformas de streaming (YouTube, Twitch).
 - Serviços em nuvem (Dropbox, iCloud).



1.14 Sistemas Operacionais de Código Aberto

1.13.1 Benefícios dos Sistemas de Código Aberto

- **Transparência e Flexibilidade:** O acesso ao código-fonte permite que programadores e estudantes entendam como o sistema funciona, modifiquem o código e criem versões personalizadas.
- **Aprendizado Prático:** Estudantes podem modificar o código, compilar e testar suas alterações, o que é uma excelente ferramenta educacional.
- **Comunidade Ativa:** Uma grande comunidade de desenvolvedores contribui para o código, ajudando a identificar e corrigir bugs rapidamente. Isso torna o software mais seguro e confiável.
- **Modelos de Negócios:** Empresas como Red Hat e SUSE mostram que é possível gerar receita com software de código aberto, oferecendo suporte técnico, serviços personalizados e hardware compatível.

1.13.2 História dos Sistemas de Código Aberto

- **Origens:** Nos anos 1950 e 1960, o software era frequentemente compartilhado livremente entre entusiastas e grupos de usuários. A cultura de compartilhamento de código era comum.
- **Restrições Comerciais:** Com o crescimento da indústria de software, empresas começaram a proteger seus códigos-fonte, distribuindo apenas binários compilados para evitar cópias não autorizadas.
- **Movimento de Software Livre:** Em 1983, Richard Stallman iniciou o projeto **GNU** para criar um sistema operacional livre e compatível com UNIX. Ele fundou a **Free Software Foundation (FSF)** e criou a **Licença Pública Geral (GPL)**, que exige que o código-fonte seja compartilhado junto com qualquer distribuição do software.

1.13.3 Linux

- **Origem:** Criado em 1991 por Linus Torvalds, o Linux é um kernel de código aberto que, combinado com ferramentas GNU, forma o sistema operacional **GNU/Linux**.

- **Distribuições:** Existem centenas de distribuições Linux, como **Ubuntu**, **Fedora**, **Debian** e **Red Hat**, cada uma com foco em diferentes usuários (desktop, servidores, gamers, etc.).
- **Acesso ao Código-Fonte:** O código-fonte do Linux pode ser baixado e modificado por qualquer pessoa. Ferramentas como o **VMware Player** permitem testar distribuições Linux em máquinas virtuais.

1.13.4 BSD UNIX

- **História:** Derivado do UNIX da AT&T, o **BSD UNIX** foi desenvolvido na Universidade da Califórnia em Berkeley. Em 1994, uma versão totalmente funcional e de código aberto, a **4.4BSD-lite**, foi lançada.
- **Distribuições:** Incluem **FreeBSD**, **NetBSD**, **OpenBSD** e **DragonFly BSD**, cada uma com foco em diferentes aspectos, como segurança, portabilidade e desempenho.
- **Influência no macOS:** O kernel do macOS, chamado **Darwin**, é baseado no BSD e também é de código aberto.

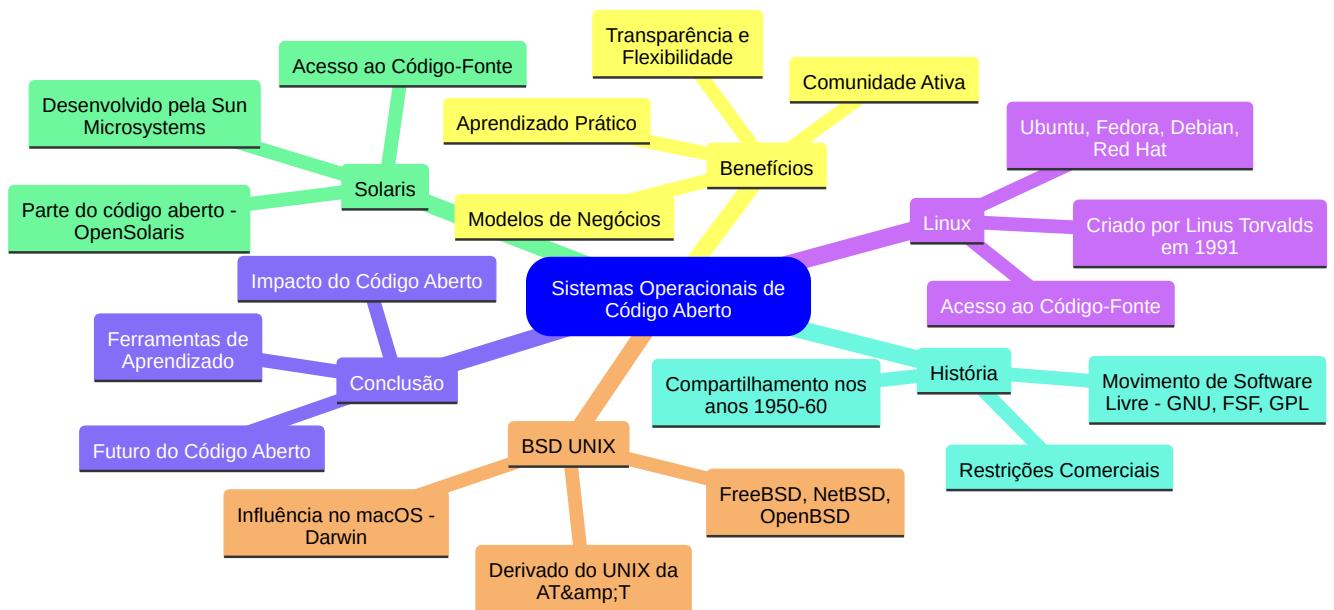
1.13.5 Solaris

- **Origem:** Desenvolvido pela Sun Microsystems, o **Solaris** é um sistema operacional baseado no UNIX. Em 2005, a Sun abriu parte do código-fonte do Solaris, criando o projeto **OpenSolaris**.
- **Características:** Embora nem todo o Solaris seja de código aberto (devido a componentes proprietários), grande parte do sistema pode ser explorada e modificada.
- **Acesso ao Código-Fonte:** O código-fonte está disponível no site opensolaris.org, onde também é possível explorar o código online.

1.13.6 Conclusão

- **Impacto do Código Aberto:** O movimento de software livre e código aberto tem impulsionado a inovação, permitindo que milhares de desenvolvedores colaborem em projetos como Linux, BSD e Solaris.
- **Ferramentas de Aprendizado:** O acesso ao código-fonte de sistemas operacionais maduros, como Linux e BSD, é uma ferramenta valiosa para estudantes e profissionais que desejam entender e contribuir para o desenvolvimento de software.
- **Futuro:** A tendência é que mais empresas e indivíduos adotem projetos de código aberto,

impulsionados pela transparência, segurança e colaboração que esse modelo oferece.



Exercícios Práticos Resolvidos - 1

1.1. Quais são as três principais finalidades de um sistema operacional?

- 1. Gerenciamento de recursos:** Controlar e alocar hardware (CPU, memória, dispositivos de E/S) para programas.
- 2. Facilitar a execução de programas:** Fornecer um ambiente para que os programas sejam executados de forma eficiente.
- 3. Proteger o sistema:** Garantir que programas e usuários não interfiram uns com os outros ou com o sistema.

1.2. Quais são as principais diferenças entre os sistemas operacionais para computadores mainframe e computadores pessoais?

- **Mainframe:**
 - Focado em alta confiabilidade, disponibilidade e processamento de grandes volumes de dados.
 - Suporta milhares de usuários simultaneamente.
 - Exemplos: IBM z/OS, Linux on IBM Z.
- **Computadores pessoais:**
 - Focado em interatividade e usabilidade para um único usuário.
 - Suporta aplicações como navegadores, editores de texto e jogos.
 - Exemplos: Windows, macOS, Linux.

1.3. Relacione as quatro etapas que são necessárias para executar um programa em uma máquina completamente dedicada – um computador que esteja executando apenas esse programa.

- 1. Carregar o programa na memória:** Transferir o código do programa do disco para a memória

RAM.

2. **Configurar o contador de programa:** Definir o endereço inicial do programa para a CPU começar a executá-lo.
3. **Executar o programa:** A CPU executa as instruções do programa.
4. **Finalizar o programa:** Encerrar a execução e liberar os recursos usados.

1.4. Quando é apropriado que o sistema operacional abra mão da eficiência e “desperdice” recursos?

- **Resposta:** Em sistemas interativos ou de tempo real, onde a experiência do usuário é prioridade (ex.: animações suaves, respostas rápidas).
- **Por que não é desperdício?:** O "desperdício" de recursos pode melhorar a usabilidade e a satisfação do usuário, o que é valioso em muitos contextos.

1.5. Qual é a principal dificuldade que um programador deverá contornar na escrita de um sistema operacional para um ambiente de tempo real?

- **Resposta:** Garantir que o sistema atenda a prazos rígidos (deadlines) para execução de tarefas, sem atrasos.
- **Explicação:** Em sistemas de tempo real, a previsibilidade e a resposta rápida são essenciais, o que exige algoritmos de escalonamento e gerenciamento de recursos altamente otimizados.

1.6. O sistema operacional deverá incluir aplicações como navegadores Web e programas de e-mail?

- **Argumento a favor:**
 - Facilita a usabilidade, pois o usuário já tem ferramentas essenciais instaladas.
 - Integração mais profunda com o sistema operacional.
- **Argumento contra:**
 - Aumenta o tamanho e a complexidade do sistema operacional.
 - Limita a escolha do usuário, que pode preferir outras aplicações.

1.7. Como a distinção entre o modo kernel e o modo usuário pode funcionar como uma forma rudimentar de sistema de proteção (segurança)?

- **Resposta:** O modo kernel tem acesso total ao hardware, enquanto o modo usuário tem acesso restrito. Isso impede que programas de usuário realizem operações perigosas, como acessar diretamente o hardware ou modificar áreas críticas do sistema.

1.8. Quais das seguintes instruções deverão ser privilegiadas?

- **Privilegiadas:**
 - a. Definir o valor do temporizador.
 - c. Apagar a memória.
 - e. Desativar interrupções.
 - f. Modificar entradas na tabela de status de dispositivo.
 - g. Passar do modo usuário para o modo kernel.
 - h. Acessar dispositivo de E/S.
- **Não privilegiadas:**
 - b. Ler o valor do relógio.
 - d. Emitir uma instrução de trap.

1.9. Duas dificuldades de proteger o sistema operacional em uma partição de memória imutável

1. **Falta de flexibilidade:** Dificulta atualizações e correções no sistema operacional.
2. **Ineficiência:** Pode limitar o uso de técnicas avançadas de gerenciamento de memória, como memória virtual.

1.10. Dois usos possíveis para múltiplos modos de operação em CPUs

1. **Virtualização:** Um modo adicional para executar máquinas virtuais.
2. **Segurança:** Modos intermediários para controle de acesso a recursos específicos.

1.11. Como temporizadores poderiam ser usados para calcular a hora atual?

- **Resposta:** Um temporizador pode ser configurado para gerar interrupções em intervalos regulares (ex.: 1 segundo). Cada interrupção incrementa um contador que representa a hora atual.
- **Explicação:** O sistema operacional usa o contador para manter o relógio do sistema atualizado.

1.12. A Internet é uma LAN ou uma WAN?

- **Resposta:** A Internet é uma **WAN (Wide Area Network)**, pois conecta redes e dispositivos em escala global, ao contrário de uma **LAN (Local Area Network)**, que é limitada a uma área geográfica pequena, como uma casa ou escritório.

Domus 2

2.1 Serviços do sistema operacional

Os serviços do sistema operacional usando analogias simples do Minecraft:

1. Interface do Usuário (UI)

- No Minecraft, você pode jogar de várias formas: no modo criativo (GUI, com menus e cliques), no modo sobrevivência (linha de comando, digitando comandos) ou com mods pré-configurados (interface batch, arquivos de comandos).
- O sistema operacional também oferece diferentes interfaces para você interagir com ele, seja por cliques, comandos ou scripts.

2. Execução de Programas

- No Minecraft, você coloca blocos e cria estruturas (programas) para fazer coisas acontecerem. O sistema operacional é como o mundo do Minecraft: ele carrega e executa os programas, permitindo que eles funcionem e, se necessário, os interrompe se algo der errado.

3. Operações de Entrada/Saída (E/S)

- No Minecraft, você interage com o mundo usando ferramentas (teclado, mouse) e dispositivos como portais ou baús (arquivos e periféricos). O sistema operacional gerencia isso, garantindo que você não "quebre" o jogo ao tentar acessar algo diretamente.

4. Manipulação de Arquivos

- No Minecraft, você organiza seus itens em baús (arquivos) e pastas (diretórios). O sistema operacional faz o mesmo, permitindo criar, ler, escrever e excluir arquivos, além de controlar quem pode acessá-los.

5. Comunicação

- No Minecraft, você pode jogar com amigos no mesmo mundo (memória compartilhada) ou em servidores diferentes (rede). O sistema operacional facilita a comunicação entre programas, seja no mesmo computador ou em redes.

6. Detecção de Erros

- No Minecraft, se você tentar colocar um bloco onde não pode, o jogo avisa. O sistema operacional faz o mesmo, detectando erros de hardware, software ou permissões e corrigindo ou alertando sobre eles.

7. Alocação de Recursos

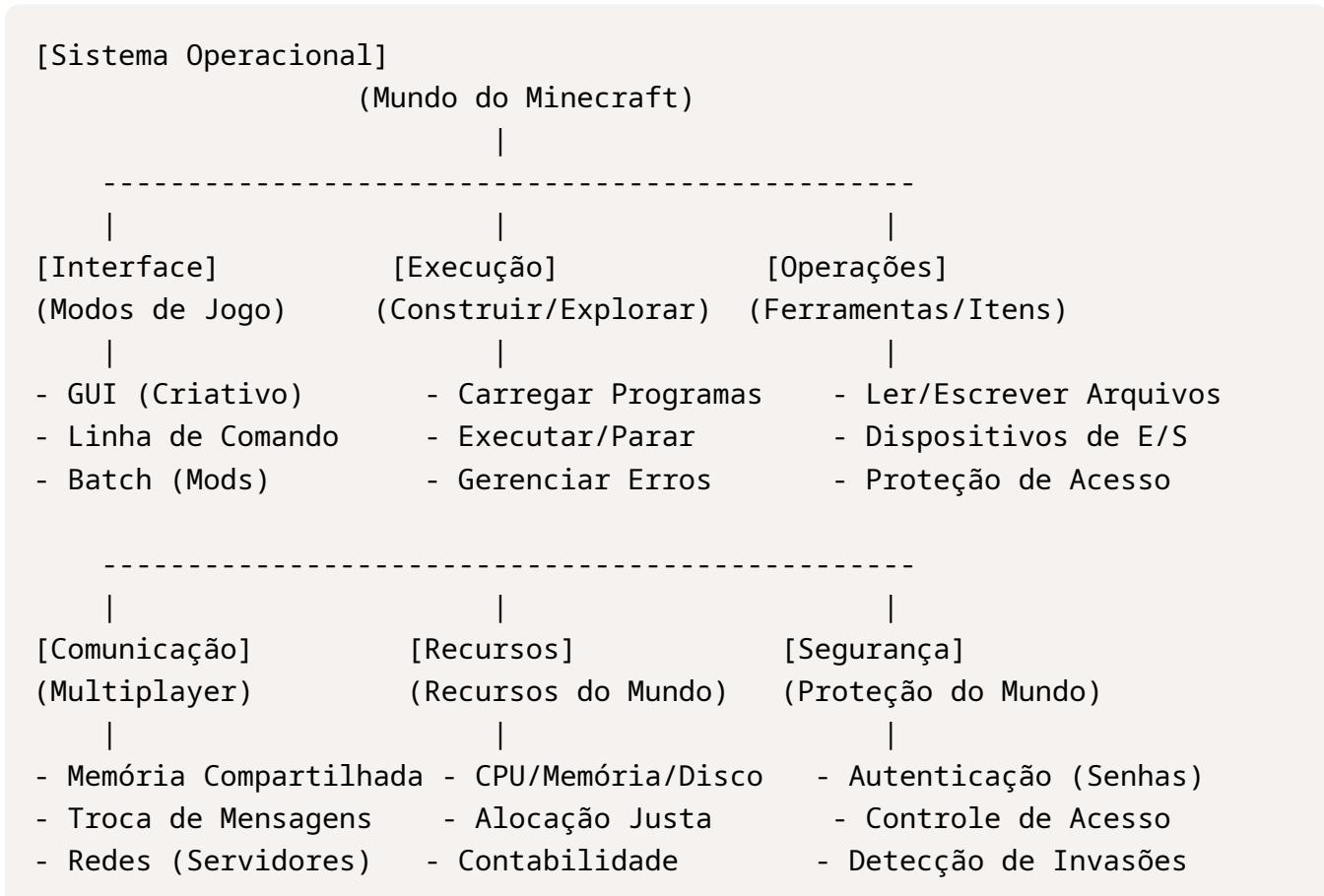
- No Minecraft, recursos como madeira, minérios e tempo são limitados. O sistema operacional gerencia recursos como memória, CPU e dispositivos, distribuindo-os de forma justa entre os programas.

8. Contabilidade

- No Minecraft, você pode ver quanto de cada recurso coletou. O sistema operacional registra o uso de recursos para cobrança ou análise, como um "log" de atividades.

9. Proteção e Segurança

- No Minecraft, você protege seu mundo com senhas ou modos de jogo. O sistema operacional faz o mesmo, garantindo que apenas usuários autorizados acessem recursos e protegendo o sistema contra invasões.



2.2 Interface usuário-sistema operacional

1. Interpretador de Comandos (CLI - Command Line Interface)

- **O que é:** Uma interface baseada em texto onde o usuário digita comandos diretamente.
- **Funcionamento:**
 - O interpretador (ou *shell*) captura e executa os comandos.
 - Exemplos: Bourne shell, C shell, Bash (Linux/UNIX), Prompt de Comando (Windows).
- **Implementação:**
 - **Método 1:** O próprio interpretador contém o código para executar os comandos (ex.: comandos internos).
 - **Método 2:** Comandos são programas externos (ex.: `rm` no UNIX), onde o interpretador apenas localiza e executa o arquivo correspondente.
- **Vantagens:**
 - Poderoso e flexível para tarefas avançadas.
 - Permite automação via scripts.

2. Interface Gráfica com o Usuário (GUI - Graphical User Interface)

- **O que é:** Uma interface visual com janelas, ícones, menus e mouse.
- **Funcionamento:**
 - O usuário interage clicando em ícones, arrastando arquivos ou selecionando opções em menus.
 - Exemplos: Windows Explorer, Aqua (Mac OS X), GNOME/KDE (Linux).
- **Histórico:**
 - Surgiu na década de 1970 (Xerox PARC).

- Popularizada pelo Macintosh (1980) e Windows (1990).

- **Vantagens:**

- Mais intuitiva e acessível para usuários comuns.
- Facilita a organização de arquivos e execução de programas.

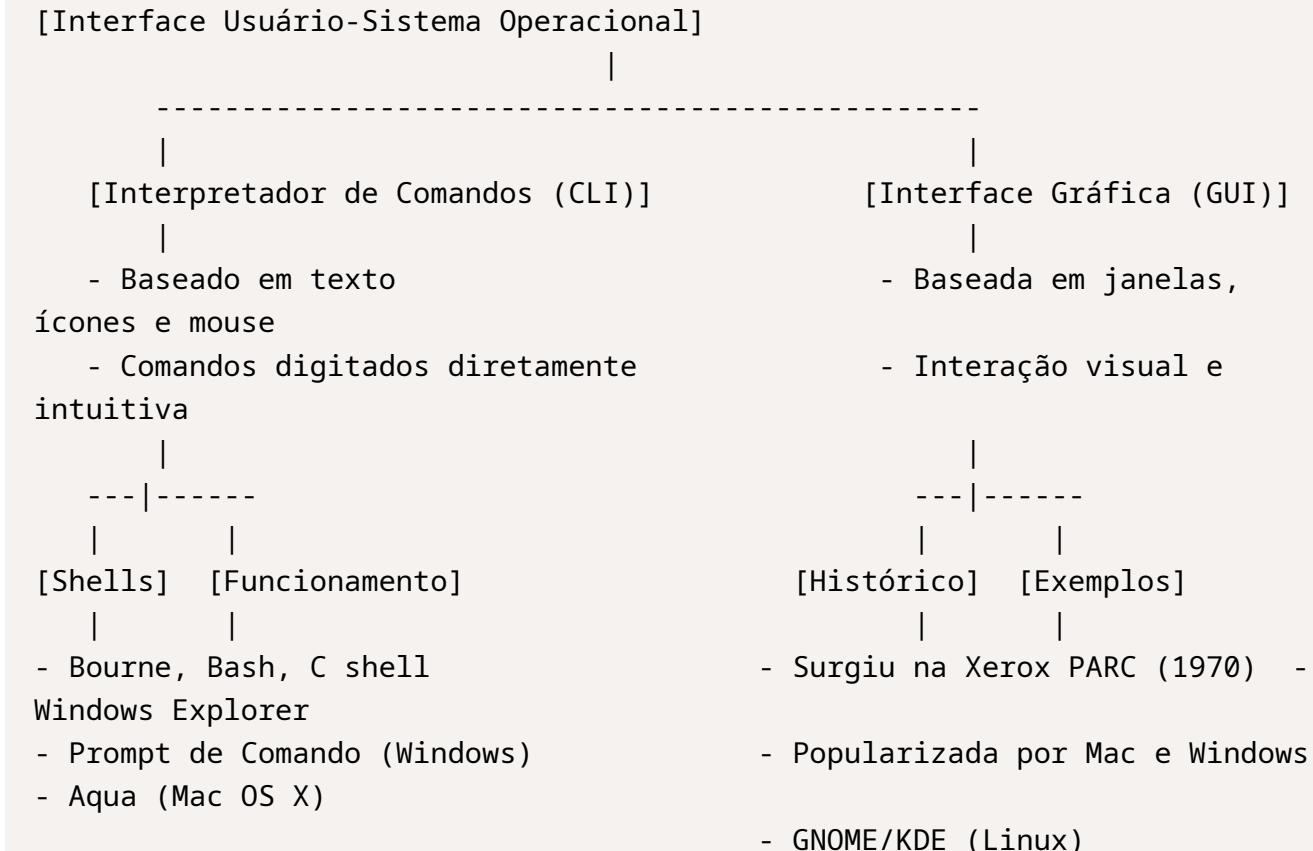
Comparação e Preferências

- **CLI vs GUI:**

- **CLI:** Preferido por usuários avançados (ex.: programadores, administradores de sistemas) por sua eficiência e controle.
- **GUI:** Preferido pela maioria dos usuários por ser mais amigável e visual.

- **Exemplos:**

- UNIX/Linux: Tradicionalmente CLI, mas oferece GUIs como GNOME e KDE.
- Windows e Mac: Focam em GUIs, mas também possuem CLIs (Prompt de Comando no Windows, Terminal no Mac).



[Vantagens]	[Implementação]	[Vantagens]	[Preferências]
- Poderoso e flexível		- Mais acessível e intuitiva	-
Usuários comuns			- Facilita organização e
- Permite automação (scripts)	- Menos técnica		- Foco em usabilidade
execução			
- Ideal para tarefas avançadas			

[Comparação CLI vs GUI]			
<ul style="list-style-type: none"> - CLI: Preferido por técnicos e programadores - GUI: Preferido pela maioria dos usuários - Ambos coexistem para atender diferentes necessidades 			

2.3 Chamadas de sistema

As **chamadas de sistema** são a interface entre os programas e os serviços oferecidos pelo sistema operacional. Elas permitem que programas solicitem operações como leitura/escrita de arquivos, gerenciamento de memória e comunicação com dispositivos. Aqui está um resumo organizado:

1. O que são Chamadas de Sistema?

- **Definição:** São rotinas que permitem que programas solicitem serviços do sistema operacional.
- **Implementação:** Escritas em linguagens como C/C++ ou assembly (para tarefas de baixo nível).
- **Exemplo:** Um programa que lê dados de um arquivo e os copia para outro usa várias chamadas de sistema:
 - Solicitar nomes dos arquivos (E/S).
 - Abrir arquivos.
 - Ler e escrever dados.
 - Tratar erros (arquivo inexistente, falta de espaço no disco, etc.).
 - Fechar arquivos e finalizar o programa.

2. Como Funcionam?

- **Sequência de Chamadas:**
 1. Solicitar nomes dos arquivos (E/S interativa ou via GUI).
 2. Abrir arquivo de entrada e criar arquivo de saída.
 3. Ler dados do arquivo de entrada e escrever no arquivo de saída.
 4. Tratar erros durante a leitura/escrita.
 5. Fechar arquivos e finalizar o programa.
- **Exemplo de Chamadas:**
 - `open()`: Abrir um arquivo.
 - `read()`: Ler dados de um arquivo.

- `write()`: Escrever dados em um arquivo.
- `close()`: Fechar um arquivo.

3. APIs e Chamadas de Sistema

- **API (Interface de Programação de Aplicação):**
 - Conjunto de funções que simplificam o uso de chamadas de sistema.
 - Exemplos: API Win32 (Windows), API POSIX (UNIX/Linux/Mac), API Java.
- **Vantagens:**
 - Portabilidade: Programas podem rodar em sistemas com a mesma API.
 - Facilidade: APIs são mais simples de usar do que chamadas de sistema diretas.
- **Relacionamento:**
 - Funções da API (ex.: `CreateProcess()` no Windows) chamam funções do sistema operacional (ex.: `NTCreateProcess()`).
 - O sistema operacional executa a operação e retorna o resultado.

4. Passagem de Parâmetros

- **Métodos:**
 1. **Registradores:** Parâmetros são passados diretamente nos registradores da CPU.
 2. **Bloco/Tabela:** Parâmetros são armazenados em memória, e o endereço do bloco é passado em um registrador.
 3. **Pilha:** Parâmetros são empilhados (push) e desempilhados (pop) pela CPU.
- **Exemplo:** No Linux, parâmetros são passados como uma tabela na memória.

5. Chamadas de Sistema em Java

- **Java Native Interface (JNI):**
 - Permite que métodos Java chamem funções nativas escritas em C/C++.
 - Essas funções podem invocar chamadas de sistema específicas do sistema operacional.

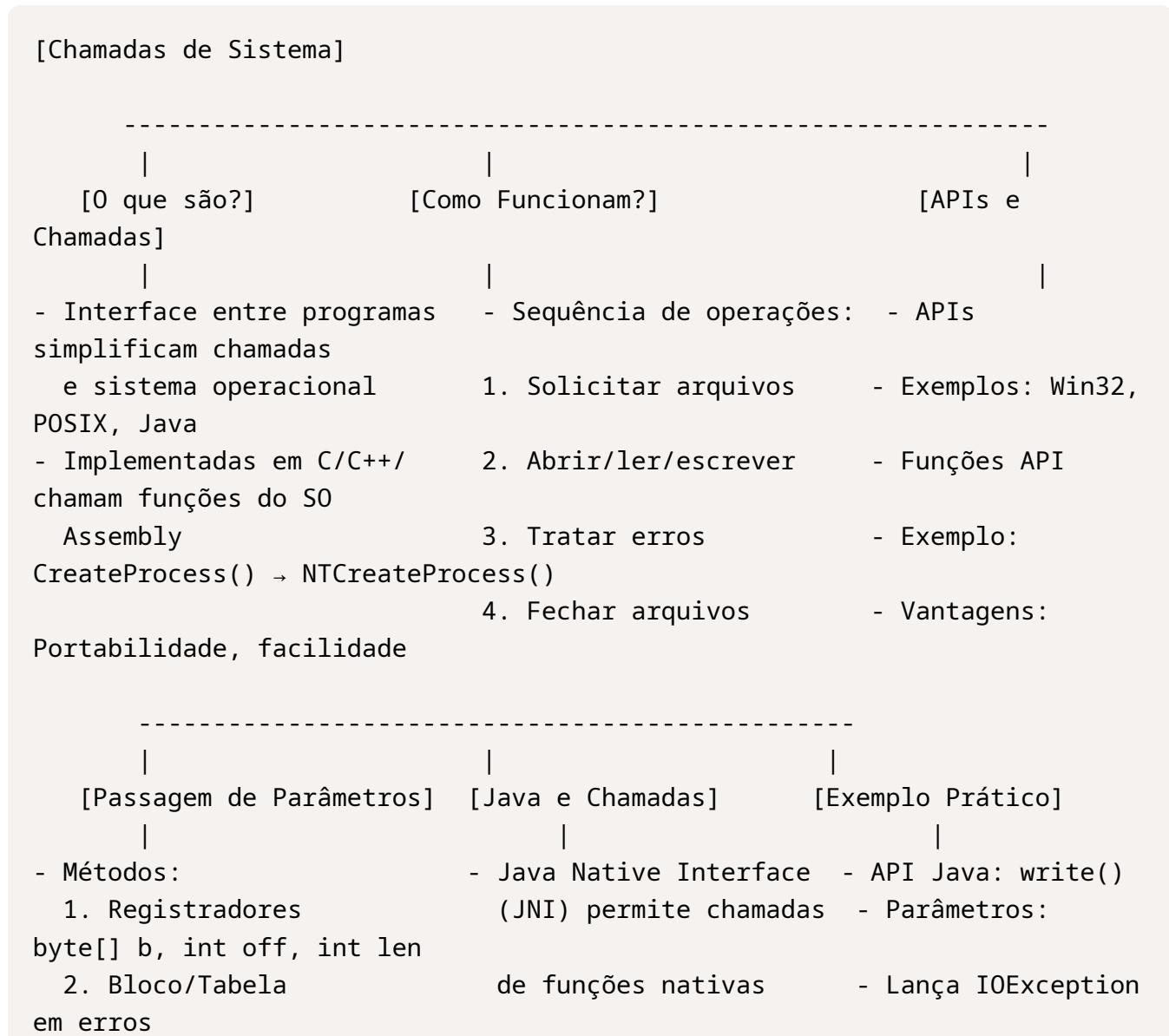
- **Limitação:** Programas que usam JNI perdem portabilidade entre plataformas.

6. Exemplo Prático

- **API Java:**

- Método `write()` da classe `java.io.OutputStream`:
 - Escreve dados em um arquivo ou conexão de rede.
 - Parâmetros: `byte[] b` (dados), `int off` (offset), `int len` (número de bytes).
 - Lança `IOException` em caso de erro.

Diagrama



3. Pilha

(C/C++) para chamadas
de sistema
- Perde portabilidade

2.4 Tipos de chamadas de sistema

1. Controle de Processos

- **Função:** Gerenciar a execução de programas (processos).
- **Exemplos de Chamadas:**
 - **Criação/Término:** fork(), create process(), exit(), abort().
 - **Controle:** wait(), signal(), get/set process attributes().
 - **Sincronização:** acquire lock(), release lock().
- **Casos de Uso:**
 - Iniciar, pausar ou finalizar processos.
 - Esperar por eventos ou processos filhos.
 - Gerenciar concorrência e compartilhamento de recursos.

2. Manipulação de Arquivos

- **Função:** Criar, ler, escrever e gerenciar arquivos e diretórios.
- **Exemplos de Chamadas:**
 - **Abertura/Fechamento:** open(), close().
 - **Leitura/Escrita:** read(), write().
 - **Atributos:** get file attributes(), set file attributes().
- **Casos de Uso:**
 - Criar, excluir ou renomear arquivos.
 - Ler e escrever dados em arquivos.
 - Gerenciar permissões e atributos de arquivos.

3. Manipulação de Dispositivos

- **Função:** Gerenciar dispositivos de hardware (físicos ou virtuais).
- **Exemplos de Chamadas:**
 - **Acesso:** `read()`, `write()`, `ioctl()`.
 - **Alocação:** `request device()`, `release device()`.
- **Casos de Uso:**
 - Ler/escrever em dispositivos como impressoras ou discos.
 - Controlar dispositivos com operações específicas (ex.: ajustar resolução de tela).

4. Manutenção de Informações

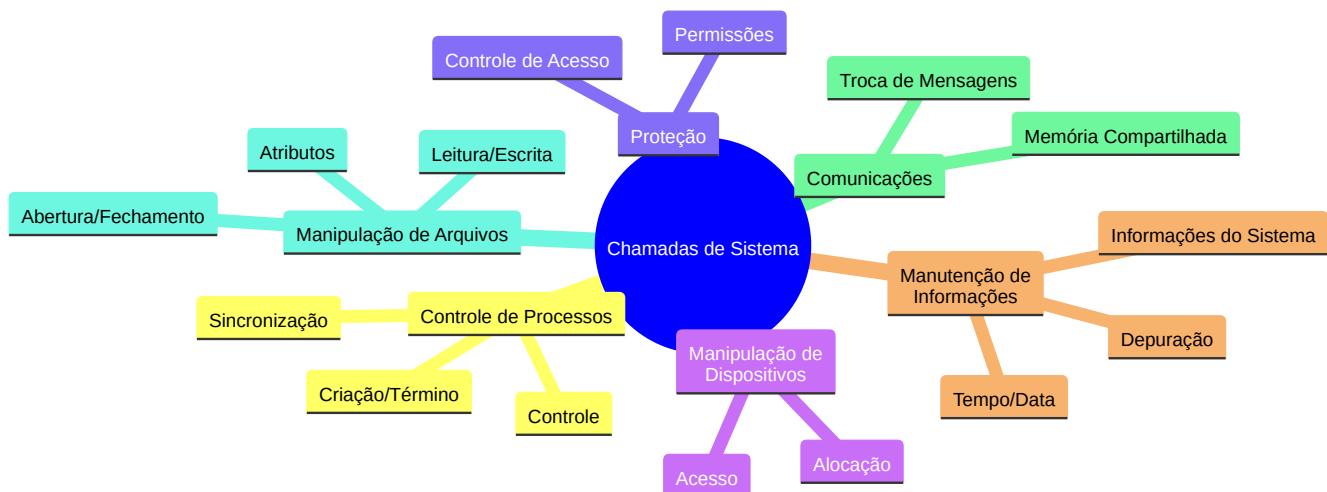
- **Função:** Obter e definir informações do sistema e do usuário.
- **Exemplos de Chamadas:**
 - **Tempo/Data:** `get time()`, `set time()`.
 - **Informações do Sistema:** `get system info()`, `get process info()`.
 - **Depuração:** `dump memory()`, `trace()`.
- **Casos de Uso:**
 - Obter informações como uso de memória, número de usuários ou versão do sistema.
 - Depurar programas com ferramentas como dump de memória ou perfil de tempo.

5. Comunicações

- **Função:** Facilitar a comunicação entre processos (no mesmo computador ou em rede).
- **Modelos:**
 - **Troca de Mensagens:** `send message()`, `receive message()`.
 - **Memória Compartilhada:** `shared memory create()`, `shared memory attach()`.
- **Casos de Uso:**
 - Trocar mensagens entre processos (ex.: cliente-servidor).
 - Compartilhar memória para comunicação rápida entre processos.

6. Proteção

- **Função:** Controlar o acesso a recursos do sistema.
- **Exemplos de Chamadas:**
 - **Permissões:** set permission(), get permission().
 - **Controle de Acesso:** allow user(), deny user().
- **Casos de Uso:**
 - Definir permissões de acesso a arquivos, dispositivos ou processos.
 - Proteger o sistema contra acessos não autorizados.



2.5 Programas do sistema

Resumo:

Os **programas do sistema** (ou utilitários) são ferramentas incluídas no sistema operacional para facilitar o desenvolvimento, execução e gerenciamento de programas. Eles se dividem em categorias como:

1. **Gerência de Arquivos:** Criar, remover, copiar, renomear e manipular arquivos/diretórios.
2. **Informações de Status:** Obter dados como hora, uso de memória, espaço em disco e logs de desempenho.
3. **Modificação de Arquivos:** Editores de texto e ferramentas para buscar/transformar conteúdo.
4. **Suporte para Linguagem de Programação:** Compiladores, interpretadores e depuradores.
5. **Carga e Execução de Programas:** Carregadores e sistemas de depuração para executar programas.
6. **Comunicações:** Ferramentas para conexões remotas, transferência de arquivos e mensagens.
7. **Programas de Aplicação:** Navegadores, editores de texto, planilhas, jogos, etc.

Além disso, a experiência do usuário é definida pelos programas de aplicação e interfaces (GUI ou CLI), que podem variar mesmo no mesmo hardware (ex.: dual-booting entre Mac OS X e Windows).



2.6 Projeto e implementação do sistema operacional

O projeto e implementação de sistemas operacionais envolvem desafios complexos, como definir objetivos, separar políticas de mecanismos, escolher linguagens de programação e estruturar o sistema de forma eficiente. Aqui estão os principais pontos:

1. Objetivos de Projeto

- **Objetivos do Usuário:** Conveniência, facilidade de uso, confiabilidade, segurança e velocidade.
- **Objetivos do Sistema:** Facilidade de projeto, implementação, manutenção, flexibilidade e eficiência.
- **Desafio:** Não há uma solução única; os requisitos variam conforme o tipo de sistema (batch, tempo real, multiusuário, etc.).

2. Mecanismos e Políticas

- **Mecanismo:** Como algo é feito (ex.: temporizador para proteção da CPU).
- **Política:** O que deve ser feito (ex.: tempo alocado para cada usuário).
- **Separação:** Mantém o sistema flexível, permitindo mudanças de políticas sem alterar mecanismos.

3. Implementação

- **Linguagens:** Sistemas operacionais modernos são escritos em linguagens de alto nível (ex.: C, C++), com trechos em assembly para otimização.
- **Vantagens:** Código mais rápido de escrever, compacto, portável e fácil de depurar.
- **Desvantagens:** Potencial redução de desempenho, mas compensada por otimizações de compiladores modernos.

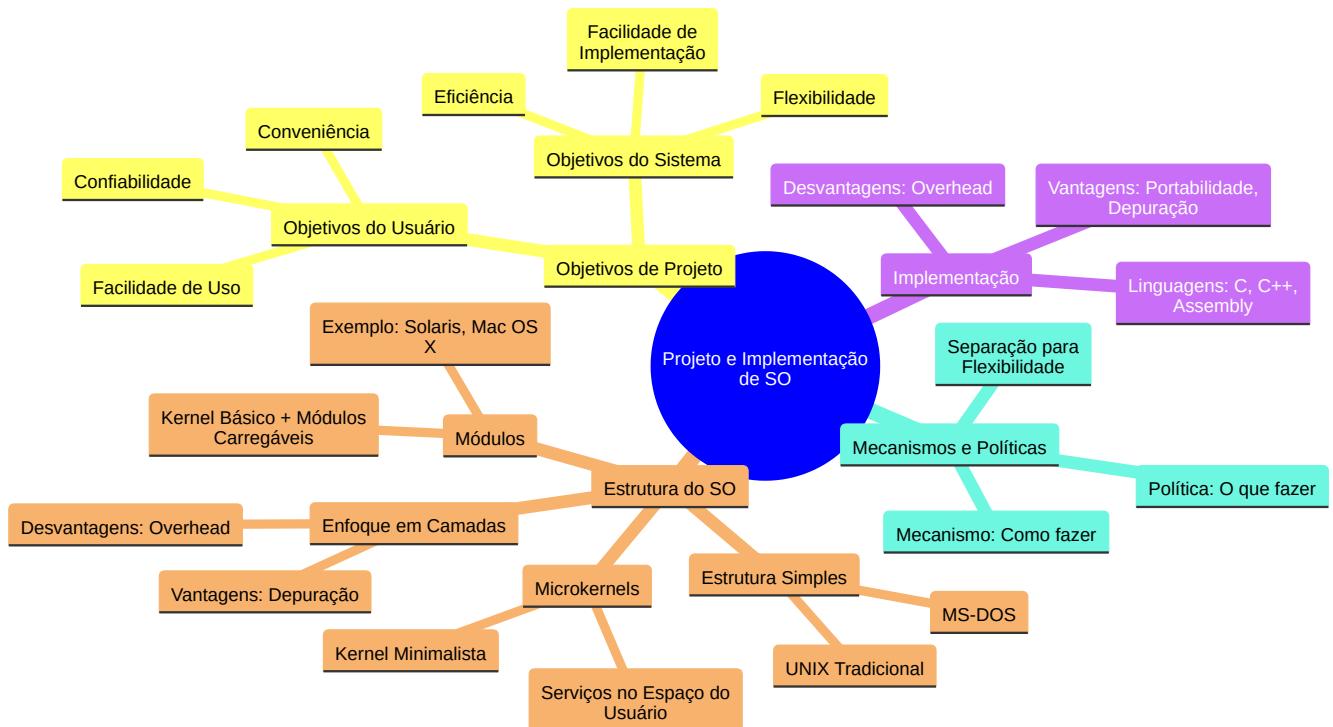
4. Estrutura do Sistema Operacional

- **Estrutura Simples:** Sistemas como MS-DOS e UNIX inicial tinham designs monolíticos, com pouca separação de componentes.
- **Enfoque em Camadas:** Divide o sistema em níveis, facilitando depuração e manutenção, mas pode adicionar overhead.
- **Microkernels:** Kernel minimalista, com serviços essenciais (gerência de processos, memória e comunicação). Serviços adicionais rodam no espaço do usuário, aumentando segurança e modularidade.
- **Módulos:** Combina vantagens de camadas e microkernels. O kernel básico carrega módulos dinamicamente (ex.: drivers, sistemas de arquivos), oferecendo flexibilidade e eficiência.

5. Exemplos de Estruturas

- **MS-DOS:** Monolítico, sem proteção de hardware.
- **UNIX Tradicional:** Kernel grande e monolítico, difícil de manter.
- **Solaris:** Usa módulos carregáveis para sistemas de arquivos, drivers e escalonamento.
- **Mac OS X:** Híbrido, com microkernel Mach e componentes BSD para redes, sistemas de arquivos e threads.

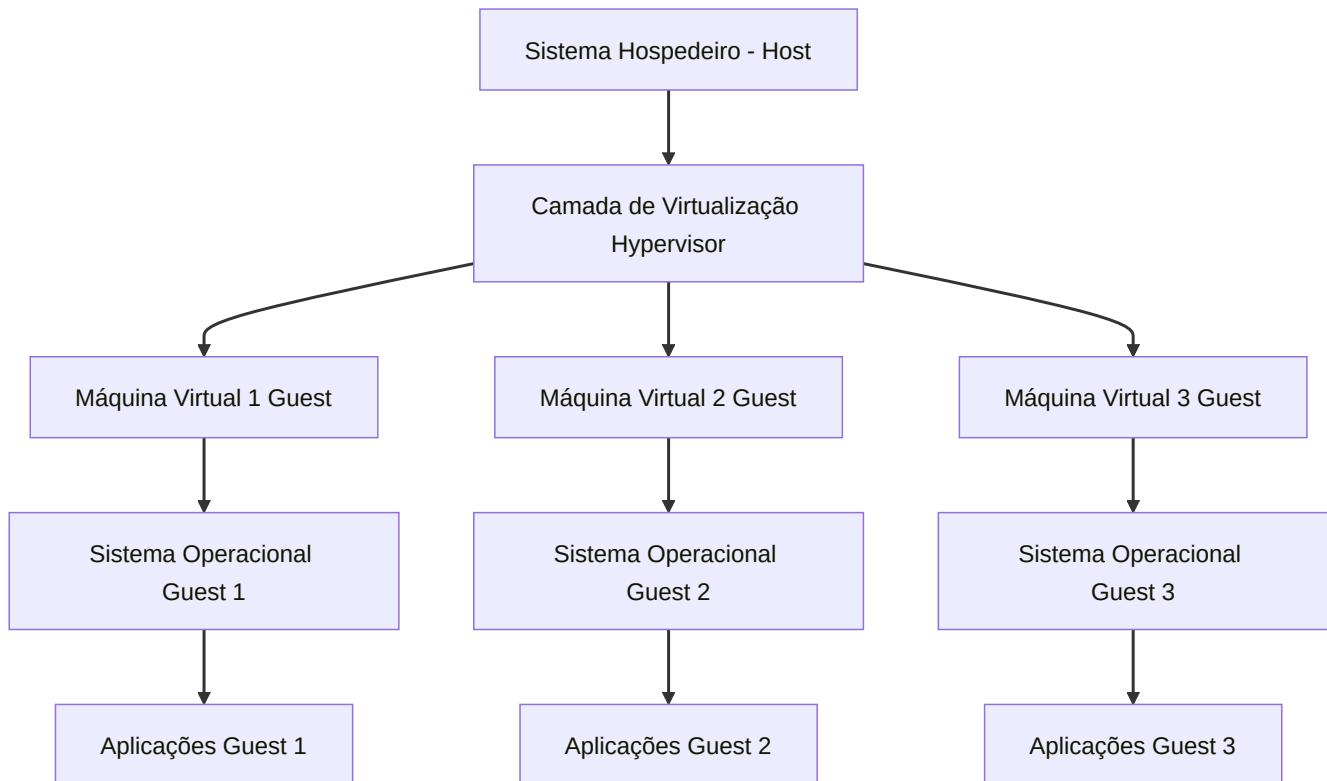
Mindmap em Mermaid:



2.7 Máquinas virtuais

Resumo:

As **máquinas virtuais (VMs)** são ambientes isolados que simulam um computador completo, permitindo a execução de múltiplos sistemas operacionais simultaneamente em um único hardware. Aqui estão os principais pontos:



1. Sistema Hospedeiro (Host):

- É o sistema físico que contém o hardware real (CPU, memória, disco, etc.).
- Roda o sistema operacional principal (ex.: Linux, Windows).

2. Camada de Virtualização (Hypervisor):

- É o software que gerencia as máquinas virtuais.
- Pode ser do Tipo 1 (executa diretamente no hardware) ou Tipo 2 (executa como uma aplicação no sistema hospedeiro).

3. Máquinas Virtuais (Guests):

- São ambientes isolados que simulam um computador completo.

- Cada máquina virtual tem seu próprio sistema operacional e aplicações.

4. Sistemas Operacionais Guests:

- Sistemas operacionais rodando dentro das máquinas virtuais (ex.: Windows, Linux, macOS).

5. Aplicações Guests:

- Programas que rodam dentro dos sistemas operacionais guests.

1. Conceito de Máquinas Virtuais

- **Definição:** Separação do hardware em múltiplos ambientes de execução, cada um com seu próprio sistema operacional.
- **Funcionamento:** Usa técnicas de escalonamento de CPU e memória virtual para criar a ilusão de um computador dedicado para cada VM.
- **Exemplo:** Um sistema físico pode rodar Windows, Linux e macOS simultaneamente como VMs.

2. Benefícios das Máquinas Virtuais

- **Isolamento:** Protege o sistema hospedeiro e outras VMs de falhas ou vírus.
- **Desenvolvimento e Testes:** Permite testar sistemas operacionais e aplicações em ambientes isolados sem afetar o sistema principal.
- **Consolidação de Sistemas:** Reduz custos ao executar múltiplos sistemas em um único hardware.
- **Portabilidade:** Facilita a migração de aplicações entre sistemas.

3. Implementação de Máquinas Virtuais

- **Desafios:** Simular o hardware completo, incluindo modos de operação (usuário e kernel).
- **Técnicas:**
 - **Modo Usuário Virtual:** Simula o modo usuário dentro do modo usuário físico.
 - **Modo Kernel Virtual:** Simula o modo kernel dentro do modo usuário físico.
- **Supporte de Hardware:** CPUs modernas (ex.: Intel VT-x, AMD-V) facilitam a virtualização com modos hospedeiro e guest.

4. VMware

- **Funcionamento:** Executa como uma aplicação no sistema hospedeiro, criando VMs independentes.
- **Exemplo:** Um sistema Linux pode rodar FreeBSD, Windows NT e Windows XP como VMs.
- **Vantagens:** Facilita a cópia, movimentação e gerenciamento de sistemas guest.

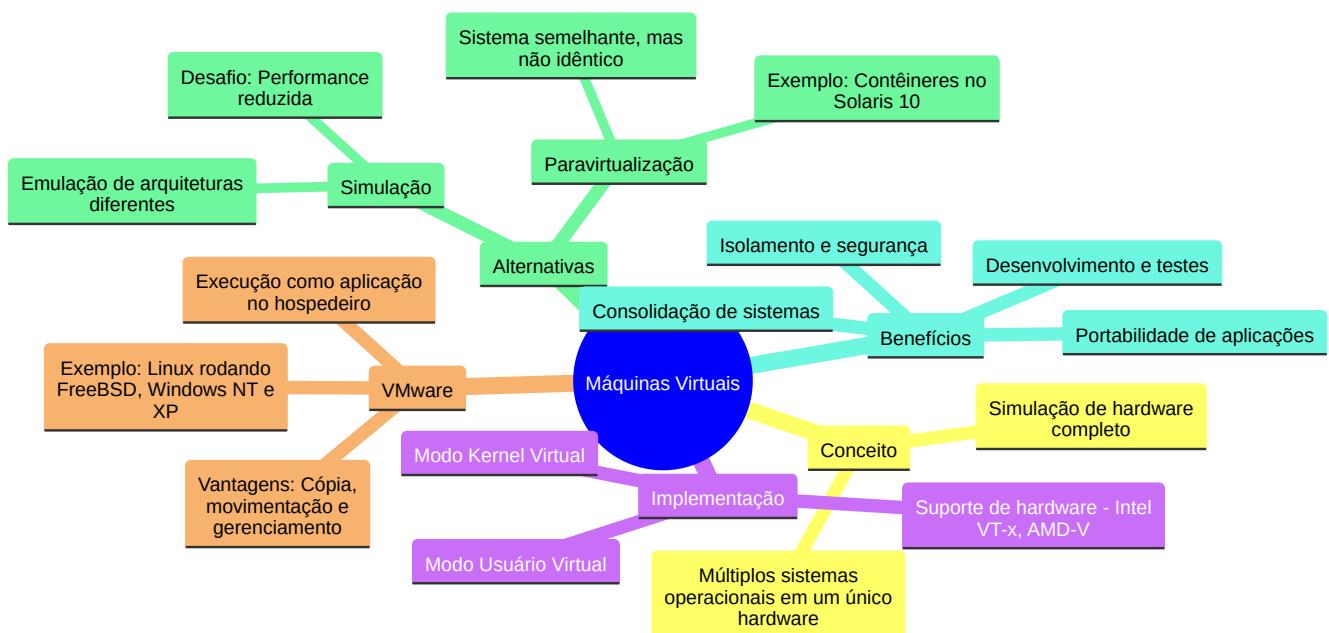
5. Alternativas à Virtualização

- **Simulação:**

- **Definição:** Emula uma arquitetura de hardware diferente da do sistema hospedeiro.
- **Uso:** Executar programas antigos em hardware moderno.
- **Desafio:** Performance reduzida, pois cada instrução é traduzida.

- **Paravirtualização:**

- **Definição:** Apresenta um sistema semelhante, mas não idêntico, ao hardware real.
- **Uso:** Requer modificações no sistema operacional guest, mas oferece melhor desempenho.
- **Exemplo:** Contêineres no Solaris 10, que virtualizam o sistema operacional, não o hardware.



2.8 Geração do sistema operacional

A **geração do sistema operacional (SYSGEN)** é o processo de configurar um sistema operacional para uma máquina específica, considerando seu hardware, periféricos e necessidades do usuário. Esse processo garante que o sistema operacional funcione de forma otimizada para a configuração do computador. Aqui estão os principais pontos:

1. Objetivo da Geração do Sistema

- **Personalização:** Adaptar o sistema operacional para uma máquina específica.
- **Configuração:** Definir parâmetros como CPU, memória, dispositivos de E/S e opções do sistema.

2. Informações Necessárias para SYSGEN

- **CPU:**
 - Tipo de processador e opções instaladas (ex.: aritmética de ponto flutuante).
 - Número de CPUs em sistemas multiprocessados.
- **Memória:**
 - Quantidade de memória RAM disponível.
- **Dispositivos de E/S:**
 - Tipos de dispositivos (ex.: discos, impressoras, placas de rede).
 - Endereços de hardware, interrupções e características específicas.
- **Opções do Sistema:**
 - Tamanho de buffers, algoritmo de escalonamento, número máximo de processos, etc.

3. Métodos de Geração do Sistema

1. Compilação Personalizada:

- Modifica o código-fonte do sistema operacional com base nas informações coletadas.
- Compila o sistema operacional para gerar uma versão específica para a máquina.

- **Vantagem:** Altamente personalizado.
- **Desvantagem:** Processo lento e complexo.

2. Seleção de Módulos Pré-Compilados:

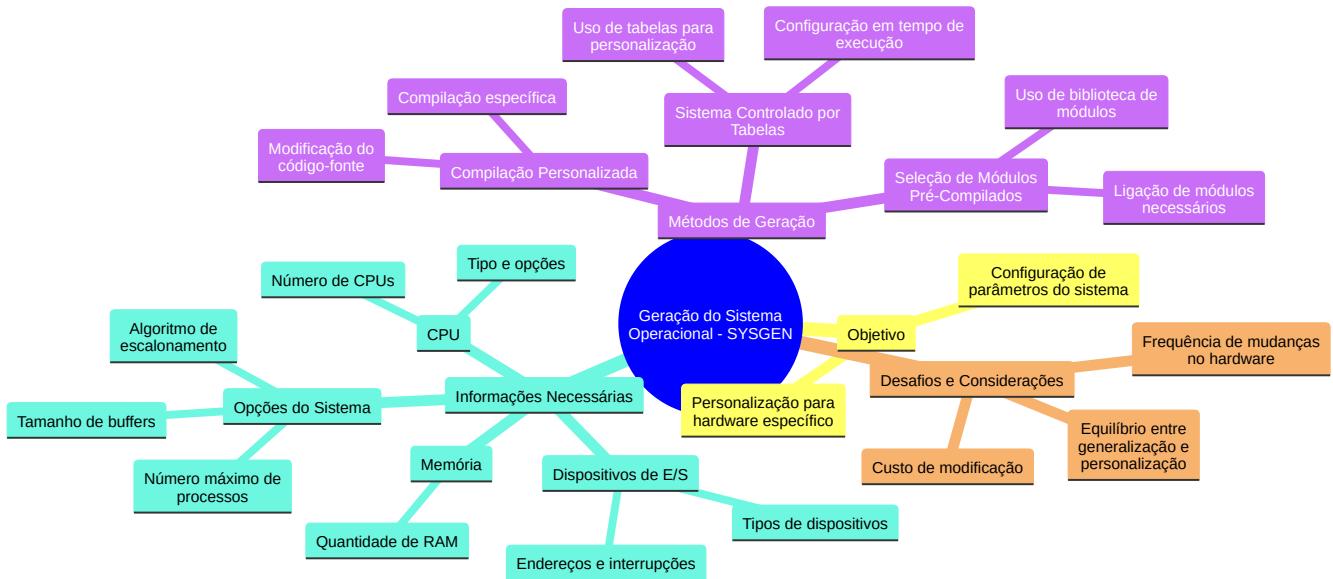
- Usa uma biblioteca de módulos pré-compilados.
- Seleciona e liga apenas os módulos necessários para a configuração.
- **Vantagem:** Mais rápido que a compilação personalizada.
- **Desvantagem:** Menos personalizado.

3. Sistema Controlado por Tabelas:

- Todo o código do sistema operacional está presente.
- A configuração é feita em tempo de execução, usando tabelas.
- **Vantagem:** Flexível e fácil de modificar.
- **Desvantagem:** Pode ser menos eficiente.

4. Desafios e Considerações

- **Frequência de Mudanças:**
 - A necessidade de reconfiguração depende da frequência com que o hardware muda.
- **Custo de Modificação:**
 - Alterar o sistema para suportar novos dispositivos pode ser caro e demorado.
- **Equilíbrio entre Generalização e Personalização:**
 - Sistemas muito genéricos podem ser menos eficientes.
 - Sistemas muito personalizados podem ser difíceis de manter.



2.9 Boot do sistema

O **boot do sistema** é o processo de inicialização do computador, que carrega o sistema operacional na memória e o prepara para execução. Com avanços tecnológicos, o processo de boot evoluiu, mas mantém os princípios básicos. Aqui estão os principais pontos atualizados:

1. Programa de Boot (Bootstrap Loader)

- **Função:** Localiza o kernel do sistema operacional, carrega-o na memória e inicia sua execução.
- **Localização:** Armazenado em firmware (UEFI/BIOS) ou em memória não volátil (como chips SPI Flash).
- **Processo:**
 - A CPU começa a execução em um endereço predefinido após o reset.
 - O programa de boot realiza diagnósticos (POST - Power-On Self-Test) e inicializa o hardware.
 - Carrega o kernel do sistema operacional na memória.

2. Tipos de Boot

- **Sistemas com Sistema Operacional em Memória Não Volátil:**
 - Usado em dispositivos embarcados, como smartphones, IoT e consoles modernos.
 - Vantagem: Simplicidade e operação reforçada.
 - Desvantagem: Dificuldade de atualização (requer reflash do firmware).
- **Sistemas com Sistema Operacional em Armazenamento (SSD/NVMe/HDD):**
 - Usado em PCs, servidores e dispositivos modernos.
 - O programa de boot (armazenado em firmware UEFI) carrega o sistema operacional do armazenamento para a memória.
 - Vantagem: Fácil atualização (basta modificar o sistema operacional no armazenamento).

3. Etapas do Boot Moderno

1. Reset da CPU: A CPU começa a execução em um endereço predefinido (definido pelo firmware UEFI/BIOS).

2. Execução do Firmware (UEFI/BIOS):

- Realiza diagnósticos do hardware (POST).
- Inicializa dispositivos básicos (memória, controladores de armazenamento, etc.).
- Localiza e executa o **bootloader** (ex.: GRUB, Windows Boot Manager).

3. Carregamento do Kernel:

- O bootloader carrega o kernel do sistema operacional na memória.
- Inicia a execução do kernel, que inicializa o sistema operacional.

4. Firmware Moderno (UEFI vs BIOS)

• **BIOS (Legacy):**

- Mais antigo, com limitações (ex.: suporte a discos de até 2 TB).
- Usa o MBR (Master Boot Record) para gerenciar o boot.

• **UEFI (Unified Extensible Firmware Interface):**

- Substituiu o BIOS na maioria dos sistemas modernos.
- Oferece suporte a discos maiores (GPT - GUID Partition Table).
- Permite boot mais rápido e seguro (Secure Boot).
- Suporta drivers e aplicativos UEFI.

5. Armazenamento de Boot

• **Disco de Boot (SSD/NVMe/HDD):**

- Contém o sistema operacional e o bootloader.
- Partição de boot (ex.: EFI System Partition no UEFI).

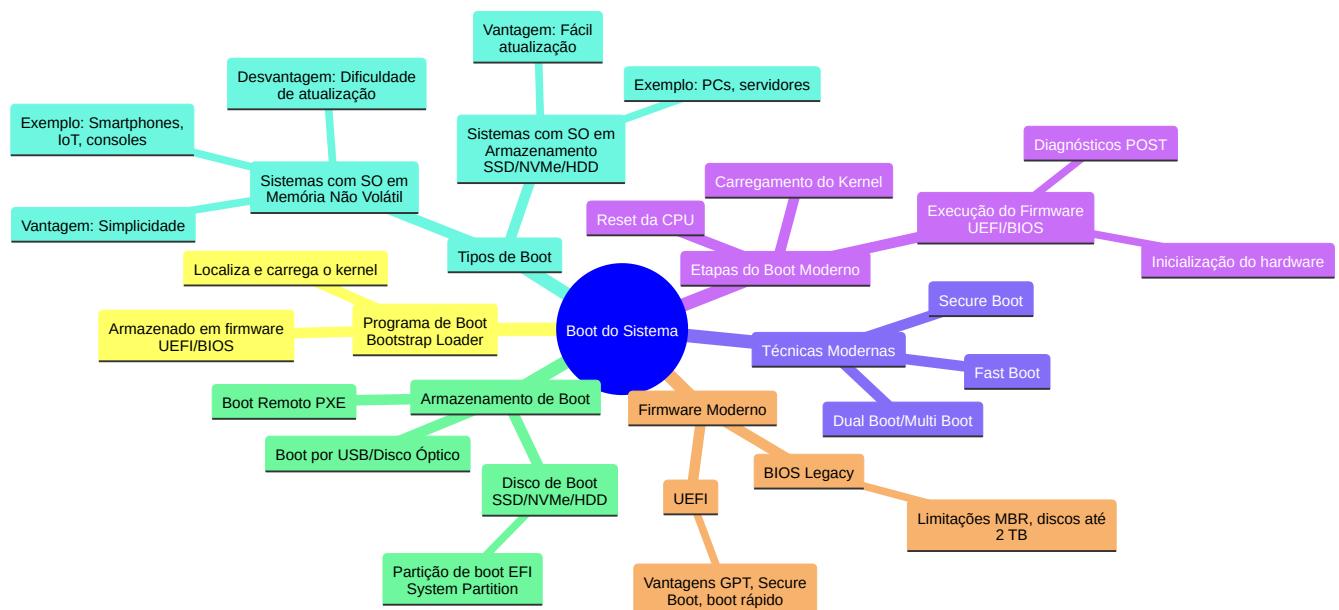
• **Boot Remoto (PXE):**

- Usado em servidores e sistemas corporativos.

- O sistema operacional é carregado pela rede.
- **Boot por USB/Disco Óptico:**
 - Usado para instalação ou recuperação de sistemas operacionais.

6. Técnicas Modernas de Boot

- **Fast Boot:**
 - Reduz o tempo de boot ao pular verificações desnecessárias.
- **Secure Boot:**
 - Verifica a integridade do bootloader e do kernel para evitar malware.
- **Dual Boot/Multi Boot:**
 - Permite a escolha entre múltiplos sistemas operacionais no boot.



Exercícios Práticos Resolvidos - 2

2.1. Qual é o propósito das chamadas do sistema?

- **Resposta:** As chamadas do sistema (system calls) são interfaces que permitem que programas de usuário solicitem serviços ao sistema operacional. Elas atuam como uma ponte entre o software de aplicação e o hardware, permitindo que os programas realizem operações como leitura/escrita de arquivos, criação de processos, comunicação entre processos e acesso a dispositivos de hardware.
- **Explicação:** Imagine que você está escrevendo um programa e precisa ler um arquivo do disco. Em vez de acessar o disco diretamente (o que seria complexo e inseguro), você usa uma chamada de sistema como `read()`. O sistema operacional cuida de todos os detalhes de baixo nível, como acessar o hardware e garantir que o arquivo seja lido corretamente.

2.2. Quais são as cinco principais atividades de um sistema operacional em relação ao gerenciamento de processos?

- **Resposta:**
 1. **Criação e término de processos:** Criar novos processos (ex.: ao abrir um programa) e encerrá-los quando não são mais necessários.
 2. **Escalonamento de processos:** Decidir qual processo deve ser executado pela CPU em um determinado momento.
 3. **Sincronização de processos:** Garantir que processos que compartilham recursos não interfiram uns com os outros.
 4. **Comunicação entre processos:** Permitir que processos troquem informações (ex.: mensagens ou memória compartilhada).
 5. **Gerenciamento de deadlocks:** Evitar ou resolver situações em que processos ficam bloqueados esperando por recursos que nunca serão liberados.
- **Explicação:** O sistema operacional age como um "gerente" dos processos, garantindo que todos tenham acesso justo aos recursos e que o sistema funcione de forma eficiente e segura.

2.3. Quais são as três principais atividades de um sistema operacional em relação ao gerenciamento de memória?

- **Resposta:**
 1. **Alocação de memória:** Distribuir a memória disponível para os processos que precisam dela.
 2. **Proteção de memória:** Garantir que um processo não acesse a memória de outro processo sem permissão.
 3. **Gerenciamento de memória virtual:** Usar técnicas como paginação e segmentação para expandir a memória disponível e otimizar o uso da memória física.
- **Explicação:** O sistema operacional gerencia a memória para evitar conflitos e garantir que cada processo tenha o espaço necessário para executar suas tarefas.

WW

2.4. Quais são as três principais atividades de um sistema operacional em relação ao gerenciamento de armazenamento secundário?

- **Resposta:**
 1. **Gerenciamento de espaço livre:** Controlar quais áreas do disco estão disponíveis para armazenar novos dados.
 2. **Alocação de espaço:** Atribuir espaço no disco para arquivos e diretórios.
 3. **Gerenciamento de disco:** Otimizar o acesso aos dados no disco (ex.: agendamento de operações de leitura/escrita).
- **Explicação:** O sistema operacional organiza o armazenamento secundário (como discos rígidos ou SSDs) para garantir que os dados sejam armazenados e recuperados de forma eficiente.

2.5. Qual é a finalidade do interpretador de comandos? Por que, normalmente, ele é separado do kernel?

- **Resposta:** O interpretador de comandos (ou shell) é um programa que permite aos usuários interagir com o sistema operacional, executando comandos e scripts. Ele é separado do kernel para:
 1. **Flexibilidade:** Diferentes interpretadores de comandos (ex.: Bash, PowerShell) podem ser usados sem modificar o kernel.
 2. **Segurança:** Se o interpretador de comandos falhar, o kernel não é afetado.

- 3. **Facilidade de desenvolvimento:** Novos interpretadores podem ser criados sem alterar o núcleo do sistema.
- **Explicação:** Imagine o shell como um "tradutor" entre o usuário e o sistema operacional. Ele recebe comandos do usuário, traduz para chamadas de sistema e envia ao kernel para execução.

2.6. Quais chamadas do sistema precisam ser executadas por um interpretador de comandos ou shell a fim de iniciar um novo processo?

- **Resposta:**
 1. **fork()**: Cria uma cópia do processo atual (o processo filho).
 2. **exec()**: Substitui o código do processo filho pelo código de um novo programa.
 3. **wait()**: Espera que o processo filho termine (opcional).
- **Explicação:** Quando você digita um comando no shell, ele usa **fork()** para criar um novo processo e **exec()** para carregar o programa que você quer executar. O **wait()** é usado se o shell precisar esperar o término do processo.

2.7. Qual é a finalidade dos programas do sistema?

- **Resposta:** Os programas do sistema (ou utilitários) fornecem ferramentas para gerenciar e interagir com o sistema operacional. Eles incluem editores de texto, compiladores, gerenciadores de arquivos e ferramentas de rede.
- **Explicação:** Esses programas facilitam tarefas como editar arquivos, compilar código, gerenciar arquivos e configurar redes, sem que o usuário precise escrever código complexo.

2.8. Qual é a principal vantagem da técnica de camadas para o projeto do sistema? Quais são as desvantagens do uso da técnica de camadas?

- **Resposta:**
 - **Vantagem:** Facilita a depuração e manutenção, pois cada camada pode ser testada e modificada independentemente.
 - **Desvantagens:**

1. **Overhead:** A comunicação entre camadas pode adicionar custos de desempenho.
 2. **Complexidade:** Definir as camadas de forma adequada pode ser difícil.
- **Explicação:** Imagine o sistema operacional como um prédio com vários andares (camadas). Cada andar tem uma função específica, mas subir e descer entre eles pode ser lento.

2.9. Relacione cinco serviços fornecidos por um sistema operacional e explique como cada um cria conveniência para os usuários. Em que casos seria impossível que os programas no nível do usuário provessem esses serviços?

- **Resposta:**
 1. **Gerenciamento de arquivos:** Permite criar, ler e organizar arquivos. Programas de usuário não poderiam acessar o disco diretamente sem o sistema operacional.
 2. **Gerenciamento de memória:** Aloca memória para programas. Sem o sistema operacional, os programas poderiam colidir e corromper a memória.
 3. **Escalonamento de processos:** Decide qual programa roda na CPU. Programas de usuário não têm visão global do sistema para tomar essa decisão.
 4. **Proteção e segurança:** Impede que programas maliciosos acessem recursos indevidos. Programas de usuário não têm controle sobre o hardware.
 5. **Comunicação entre processos:** Permite que programas troquem dados. Programas de usuário não poderiam coordenar isso sem o sistema operacional.
- **Explicação:** O sistema operacional age como um "guardião" que gerencia recursos e garante que tudo funcione de forma segura e eficiente.

2.10. Por que alguns sistemas armazenam o sistema operacional no firmware, enquanto outros o armazenam no disco?

- **Resposta:**
 - **Firmware:** Usado em dispositivos embarcados (ex.: smartphones, IoT) para simplicidade e operação reforçada. O sistema operacional é carregado diretamente da memória não volátil.
 - **Disco:** Usado em PCs e servidores para flexibilidade e facilidade de atualização. O sistema operacional é carregado do armazenamento secundário (SSD/HDD).

- **Explicação:** Dispositivos pequenos e especializados usam firmware para economizar espaço e garantir operação confiável, enquanto sistemas maiores usam disco para permitir atualizações e personalização.

2.11. Como um sistema poderia ser projetado para permitir uma escolha de sistemas operacionais para o boot do sistema? O que o programa de boot precisaria fazer?

- **Resposta:**
 - **Dual Boot/Multi Boot:** O programa de boot (ex.: GRUB) permite escolher entre vários sistemas operacionais instalados no disco.
 - **Funcionamento:**
 1. O programa de boot carrega uma lista de sistemas operacionais disponíveis.
 2. O usuário seleciona o sistema desejado.
 3. O programa de boot carrega o kernel do sistema operacional escolhido na memória.
- **Explicação:** Imagine o programa de boot como um "menu" que permite escolher entre Windows, Linux ou outro sistema operacional instalado no computador.

Questões 1

Esta seção contém perguntas relacionadas ao assunto em questão, que podem ser usadas como referência para aprender ou revisar conhecimentos.

- i** Sugerimos que resolva estas questões para auxiliar em um melhor entendimento do conteúdo e assim melhor resolução dos quizzes

Pergunta 1

Dispositivos que utilizam Bluetooth e redes Wi-Fi se comunicam sem fio dentro de uma determinada área, formando uma:

- a) Rede de área metropolitana (MAN)
- b) Rede de área local (LAN)
- c) Rede de pequena área (SAN)
- d) Rede de longa distância (WAN)

Pergunta 2

O que fornece serviços adicionais para desenvolvedores ao intermediar a comunicação entre aplicativos e o sistema operacional?

- a) Virtualização
- b) Middleware
- c) Computação em nuvem
- d) Software de sistema

Pergunta 3

Os sistemas operacionais geralmente operam em dois modos distintos. Quais são eles?

- a) Modo físico e modo lógico
- b) Modo usuário e modo kernel
- c) Modo supervisor e modo secundário
- d) Modo protegido e modo comum

Pergunta 4

No contexto do Linux, uma versão personalizada do sistema operacional é conhecida como:

- a) Instalação (Installation)
- b) Distribuição (Distribution)
- c) LiveCD
- d) Virtual Machine

Pergunta 5

Qual das seguintes afirmações sobre dispositivos móveis é falsa?

- a) Eles geralmente possuem menos núcleos de processamento do que desktops.
- b) O consumo de energia é um fator crítico para dispositivos móveis.
- c) Dispositivos móveis sempre possuem mais capacidade de armazenamento que laptops.
- d) Algumas funcionalidades dos dispositivos móveis não estão presentes em desktops.

Pergunta 6

Sistemas embarcados geralmente executam um sistema operacional de:

- a) Tempo real
- b) Rede
- c) Clusterizado
- d) Multiprogramação

Pergunta 7

Quais são dois fatores importantes no design da memória cache?

- a) Consumo de energia e reutilização
- b) Política de tamanho e substituição
- c) Privilégios de acesso e controle
- d) Velocidade e volatilidade

Pergunta 8

De que forma um sistema operacional pode ser comparado a um governo?

- a) Ele executa todas as funções sozinho.
- b) Ele cria um ambiente para que outros programas possam operar.

- c) Ele prioriza as necessidades individuais dos usuários.
- d) Ele raramente funciona corretamente.

Pergunta 9

Sobre instruções privilegiadas, qual das afirmações a seguir é incorreta?

- a) Elas não podem ser executadas no modo usuário.
- b) Apenas podem ser executadas no modo kernel.
- c) São usadas para gerenciamento de interrupções.
- d) Nunca representam riscos ao sistema.

Pergunta 10

A menor unidade de execução dentro de um sistema operacional é chamada de:

- a) Sistema operacional
- b) Timer
- c) Bit de modo
- d) Processo

Pergunta 11

Qual das opções abaixo é um exemplo de um programa de sistema?

- a) Navegador Web
- b) Interpretador de comandos
- c) Planilha eletrônica
- d) Software de edição de imagem

Pergunta 12

Qual chamada de sistema do Windows é equivalente à `close()` do UNIX?

- a) CloseHandle()
- b) close()
- c) Exit()
- d) CloseFile()

Pergunta 13

As chamadas de sistema são responsáveis por:

- a) Fornecer uma interface para os serviços do sistema operacional
- b) Gerenciar apenas memória virtual
- c) Proteger exclusivamente processos críticos
- d) Impedir a execução de programas de terceiros

Pergunta 14

O que define o que será feito dentro de um sistema operacional?

- a) Política
- b) Mecanismo
- c) Estratégia
- d) Interface

Pergunta 15

No Windows, a chamada de sistema `CreateFile()` é utilizada para criar arquivos. Qual a chamada equivalente no UNIX?

- a) `fork()`
- b) `open()`
- c) `createfile()`
- d) `ioctl()`

Pergunta 16

Qual das opções **não** é uma categoria principal de chamadas de sistema?

- a) Segurança
- b) Proteção
- c) Controle de processos
- d) Comunicação

Pergunta 17

Microkernels utilizam ____ para comunicação interna.

- a) Chamadas de sistema
- b) Memória compartilhada
- c) Virtualização
- d) Passagem de mensagens

Pergunta 18

Um bloco de inicialização (bootstrap loader):

- a) É composto por vários blocos de disco
- b) Pode conter múltiplos cilindros de disco
- c) Normalmente é suficiente para carregar e iniciar o sistema operacional
- d) Apenas aponta para a localização do restante do sistema de boot

Pergunta 19

Qual é o sistema operacional utilizado em dispositivos iPhone e iPad?

- a) iOS
- b) UNIX
- c) Android
- d) Mac OS X

Pergunta 20

Para um programa SYSGEN de um sistema operacional, qual das informações abaixo é **menos útil**?

- a) Quantidade de memória disponível
- b) Configurações como tamanho de buffer e algoritmo de escalonamento de CPU
- c) Lista de aplicativos a serem instalados
- d) Arquitetura da CPU

Pergunta 21

Um sistema operacional pode ser classificado como um:

- a) Gerenciador de recursos
- b) Programa de usuário
- c) Firmware de inicialização

- d) Simulador de processos

Pergunta 22

O que é multitarefa em um sistema operacional?

- a) A execução de um único processo por vez
- b) A capacidade de executar múltiplos processos simultaneamente
- c) A execução de tarefas em tempo real sem atraso
- d) A execução de comandos administrativos pelo usuário

Pergunta 23

Qual das seguintes opções descreve um **hipervisor**?

- a) Um software responsável pela virtualização de hardware
- b) Um protocolo de comunicação em redes
- c) Um sistema operacional para servidores
- d) Um método de gerenciamento de arquivos

Pergunta 24

O que acontece quando um processo em execução tenta acessar uma página de memória que não está carregada?

- a) Ele é imediatamente encerrado pelo sistema operacional
- b) O sistema operacional gera uma interrupção e carrega a página necessária
- c) O sistema operacional ignora a solicitação
- d) O processo entra em um estado de loop infinito

Pergunta 25

O que é um deadlock em um sistema operacional?

- a) Um erro crítico no kernel
- b) Um estado onde dois ou mais processos ficam bloqueados indefinidamente
- c) Um método de gerenciamento de arquivos
- d) Uma forma de escalonamento de processos

Prática 1

Nesta prática, vamos criar e configurar máquinas virtuais.

Conhecendo ferramentas

Para fazer a virtualização de sistemas operacionais, temos alguns programas disponíveis:

- Windows:
 - VirtualBox
 - VMWare
 - Parallels Desktop
- Linux:
 - VirtualBox
 - GNOME Boxes
 - Virtual Machine Manager

i Há casos em que o VirtualBox falha por razões desconhecidas, então é melhor ter mais de uma opção disponível.

Vamos começar

Requisitos dos sistemas

Confira os requisitos dos sistemas operacionais que serão usados:

- Windows 10 (<https://support.microsoft.com/pt-br/windows/requisitos-do-sistema-do-windows-10-6d4e9a79-66bf-7950-467c-795cf0386715>)
- Ubuntu Desktop (<https://ubuntu.com/server/docs/system-requirements>)

Instalando a ferramenta

Vamos usar o VirtualBox.

- Para Windows (<https://download.virtualbox.org/virtualbox/7.1.6/VirtualBox-7.1.6-167084-Win.exe>)

- Para Linux (https://www.virtualbox.org/wiki/Linux_Downloads)

Instalar ISOs (Windows e Ubuntu)

- Windows (<https://www.microsoft.com/pt-br/software-download/windows10ISO>)

i Na ISO oficial do Windows, o link acima, vêm todas as versões.

- Ubuntu Desktop (<https://ubuntu.com/download/desktop>)

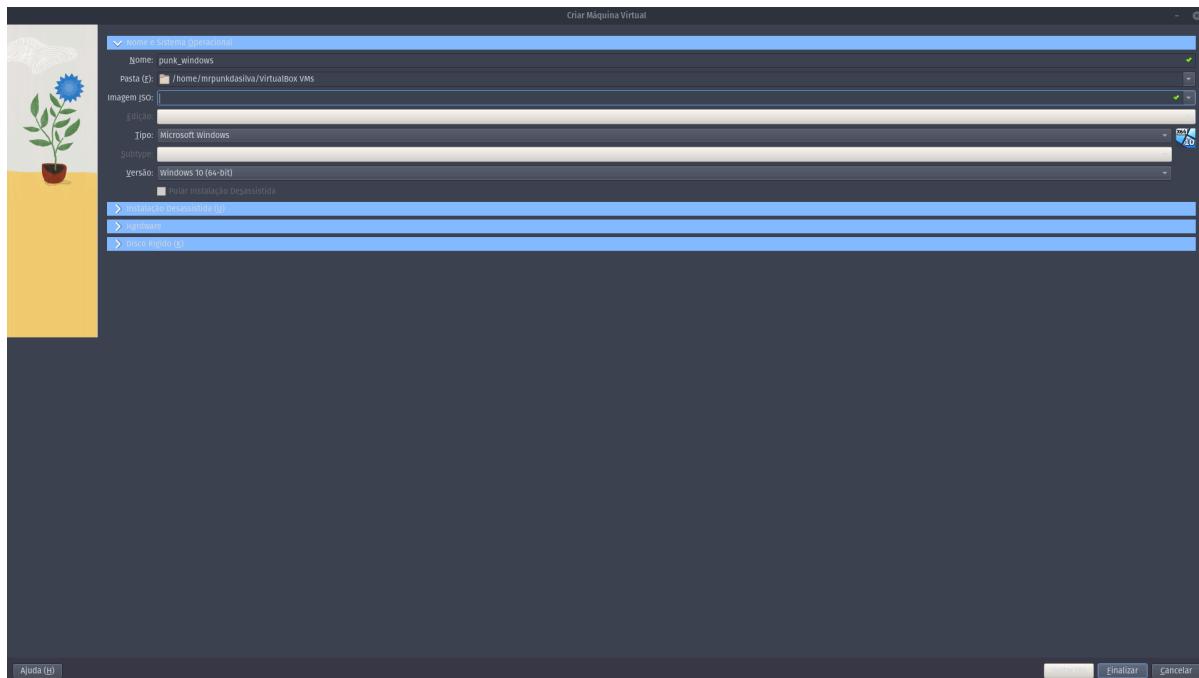
Criando Máquinas Virtuais

Windows

1. Com o VirtualBox aberto, pressione: **CTRL** + **N**

i Você pode acessar a opção também por: Machine > Add

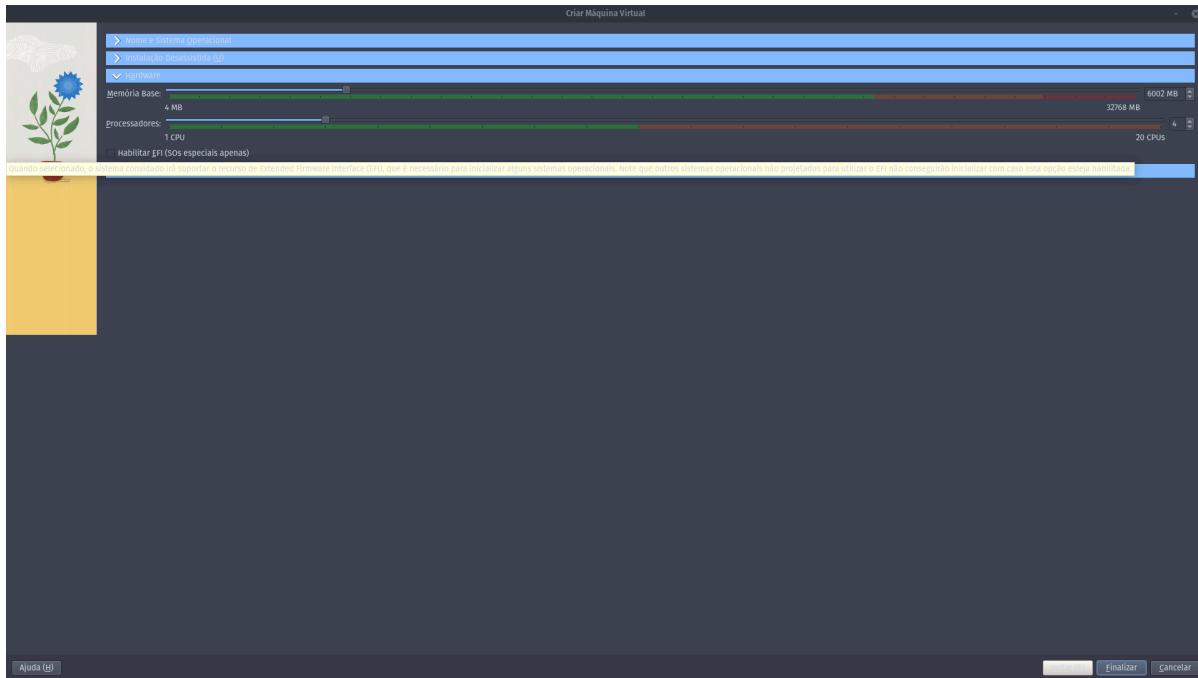
2. Definir nome, sistema e versão:



Tela de criação de máquina virtual no VirtualBox

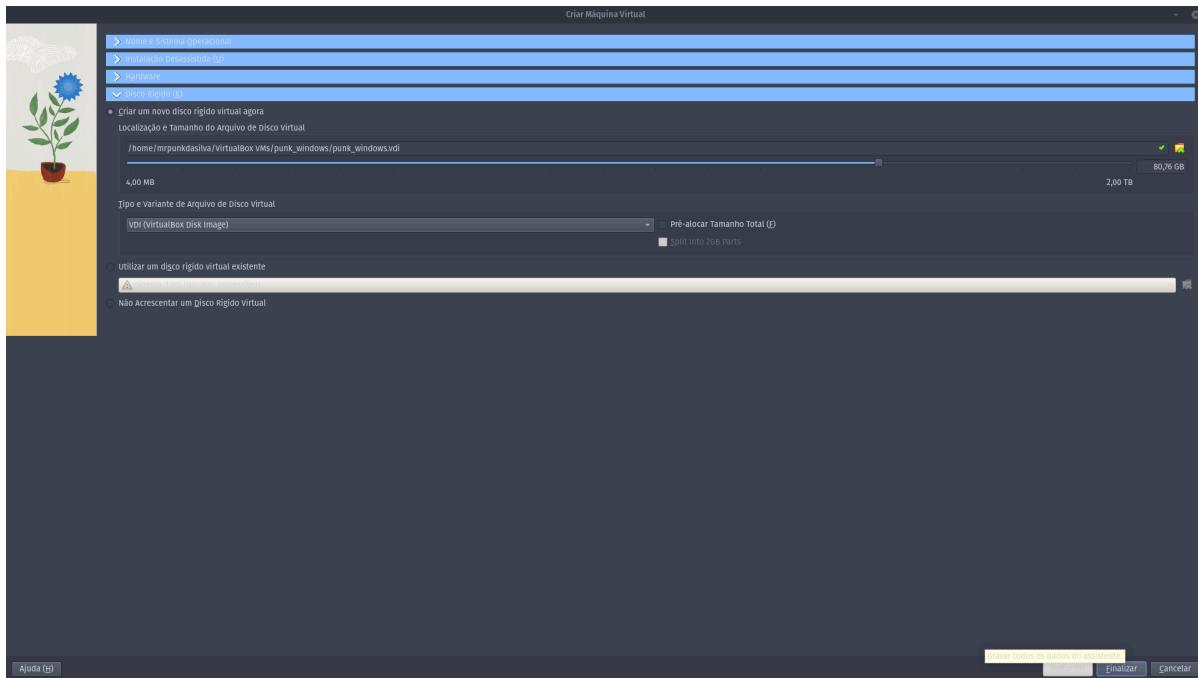
3. Agora vamos definir a memória RAM. É bom deixarmos no mínimo 4GB

- i** Atente-se que computadores não lidam bem com números ímpares.



Configuração de memória RAM para a máquina virtual

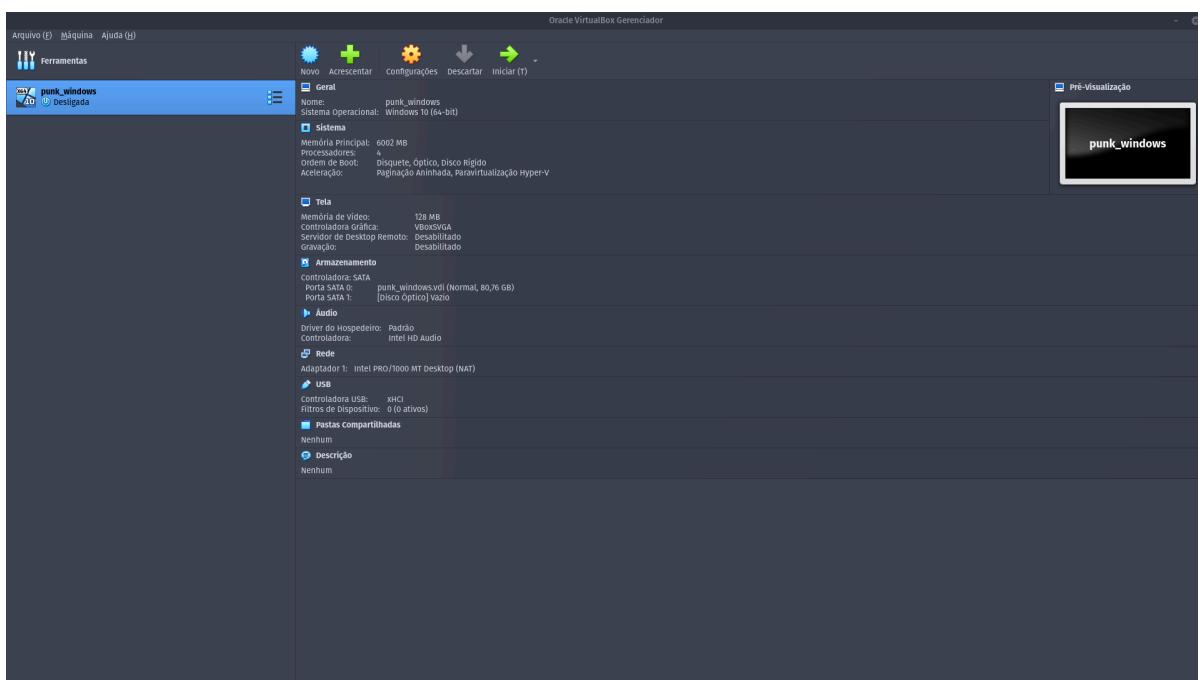
4. Agora definimos o espaço de armazenamento, disco rígido:



Configuração de disco rígido para a máquina virtual

5. Com tudo criado, basta ir em **Finish**:

6. Temos então nossa primeira máquina virtual criada:

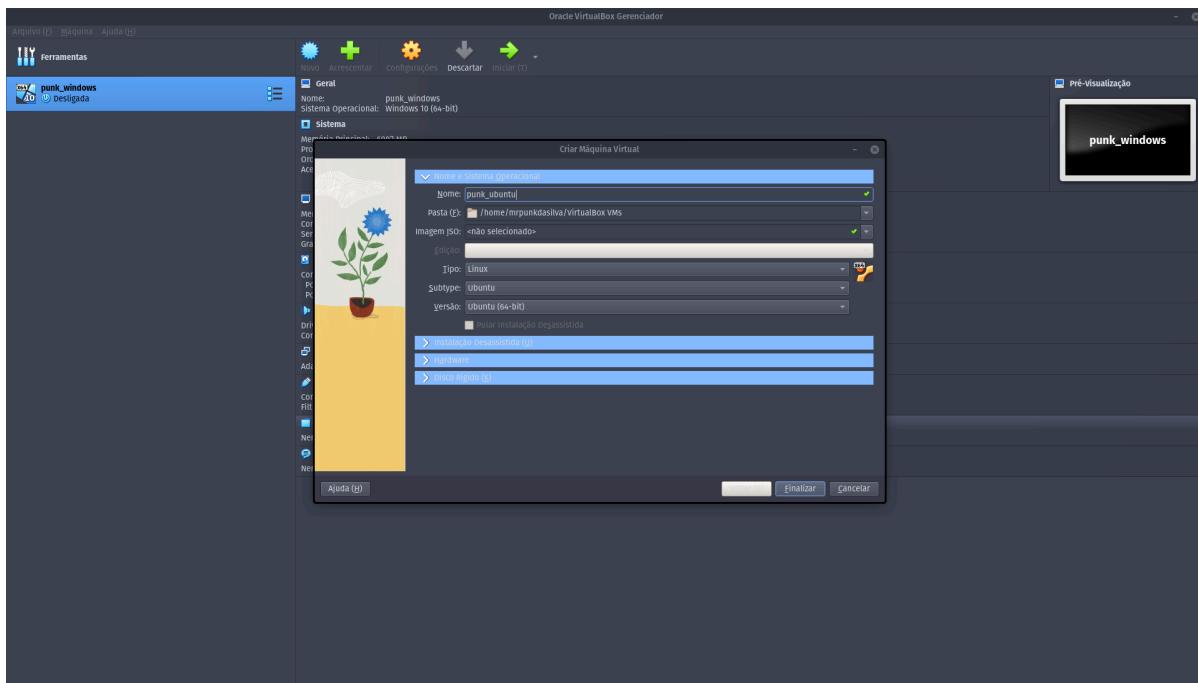


Máquina virtual Windows criada no VirtualBox

1. Com o VirtualBox aberto, pressione: **CTRL** + **N**

i Você pode acessar a opção também por: Machine > Add

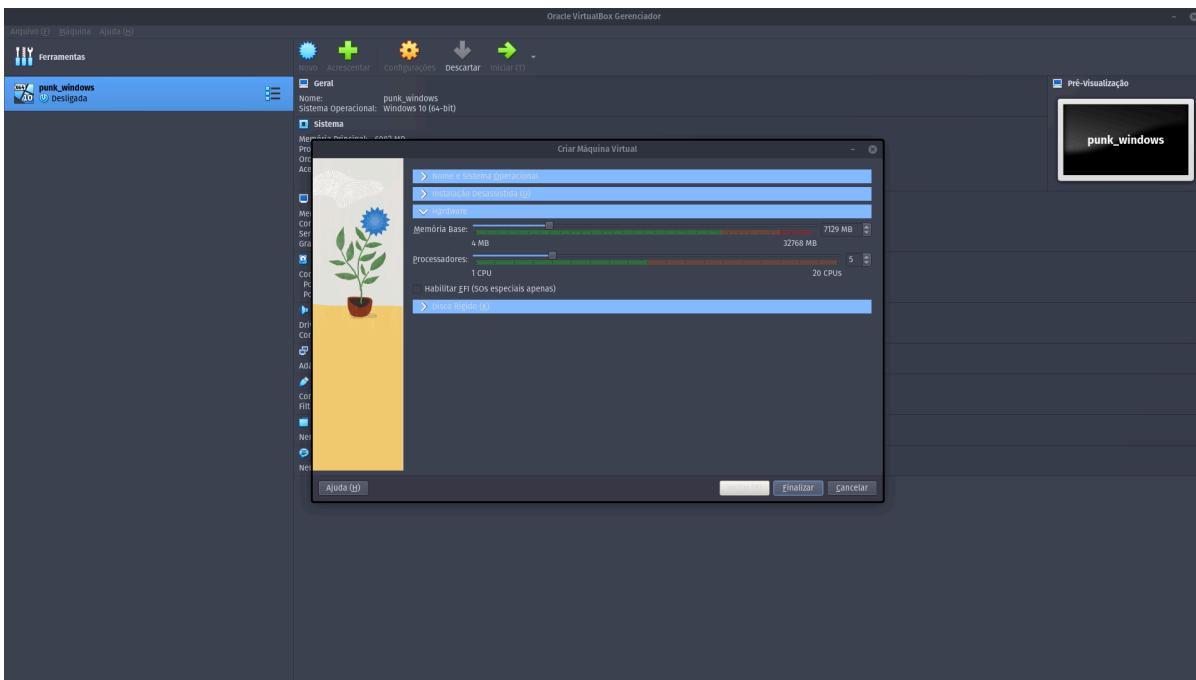
2. Definir nome, sistema e versão:



Tela de criação de máquina virtual Ubuntu no VirtualBox

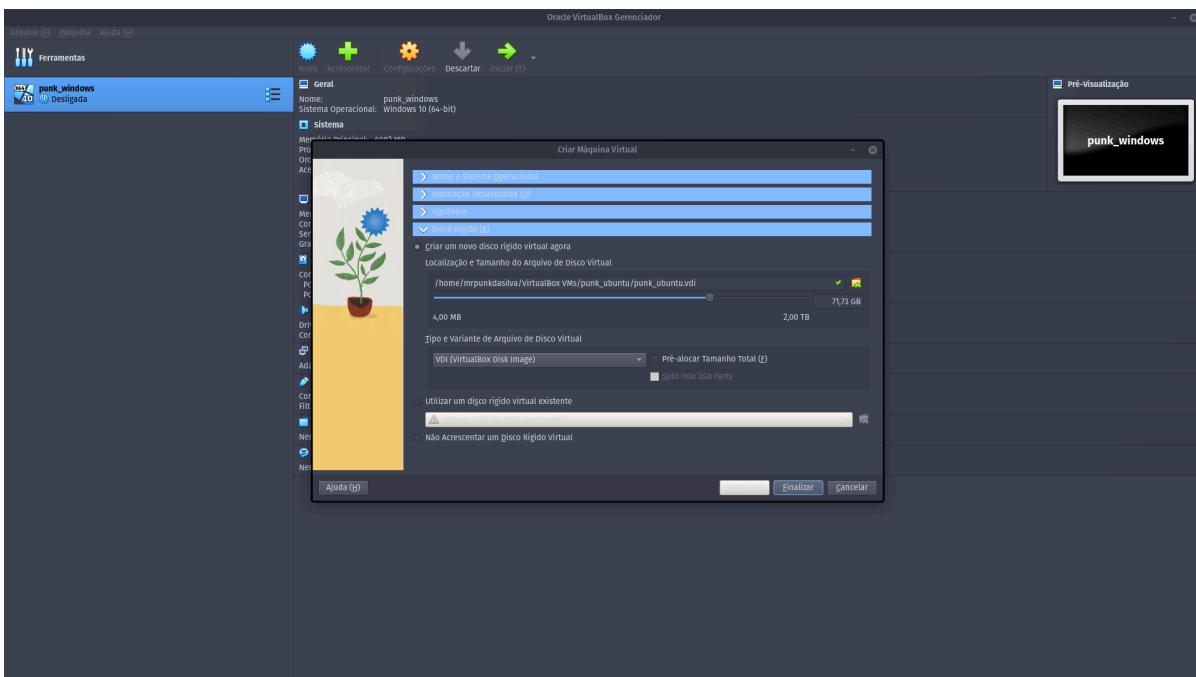
3. Agora vamos definir a memória RAM. É bom deixarmos no mínimo 4GB

i Atente-se que computadores não lidam bem com números ímpares.



Configuração de memória RAM para a máquina virtual Ubuntu

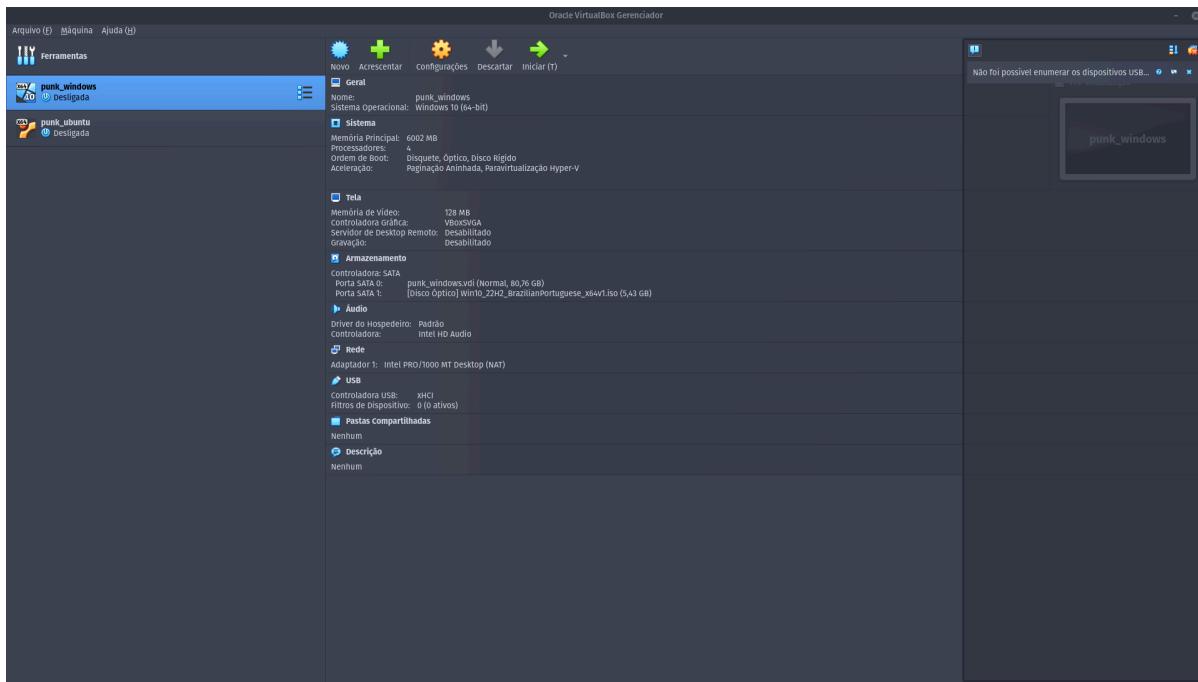
4. Agora definimos o espaço de armazenamento, disco rígido:



Configuração de disco rígido para a máquina virtual Ubuntu

5. Com tudo criado, basta ir em **Finish**:

6. Temos então nossa primeira máquina virtual criada:

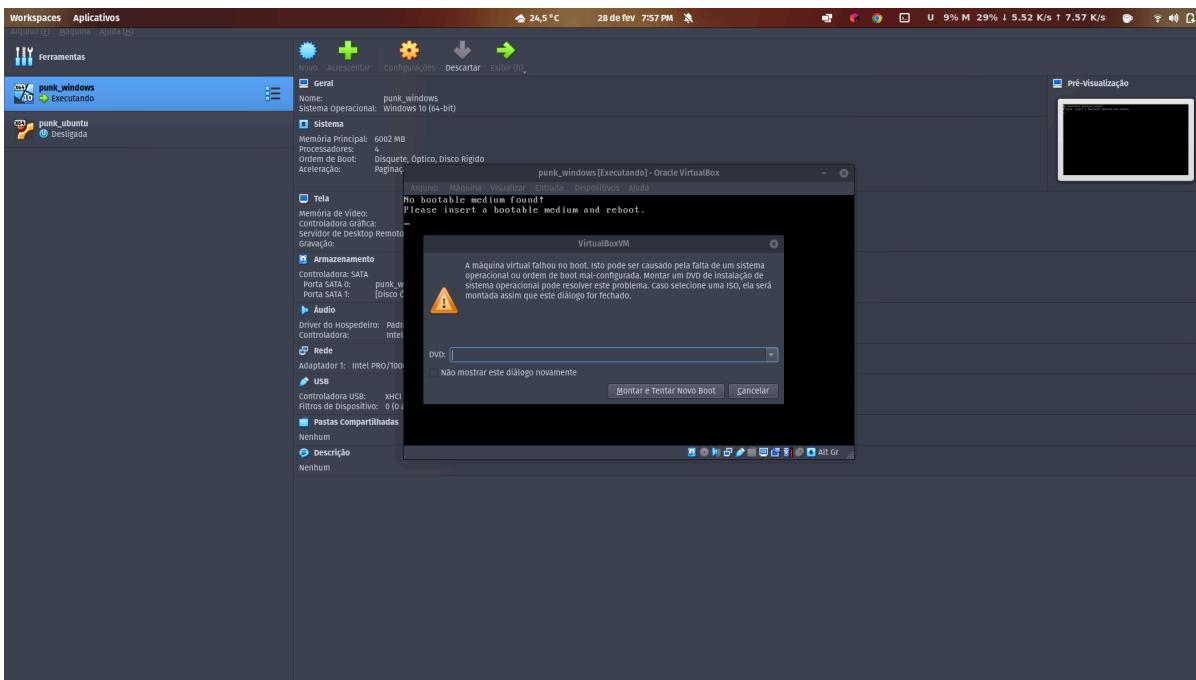


Máquina virtual Ubuntu criada no VirtualBox

Logar nas VMs recém-criadas

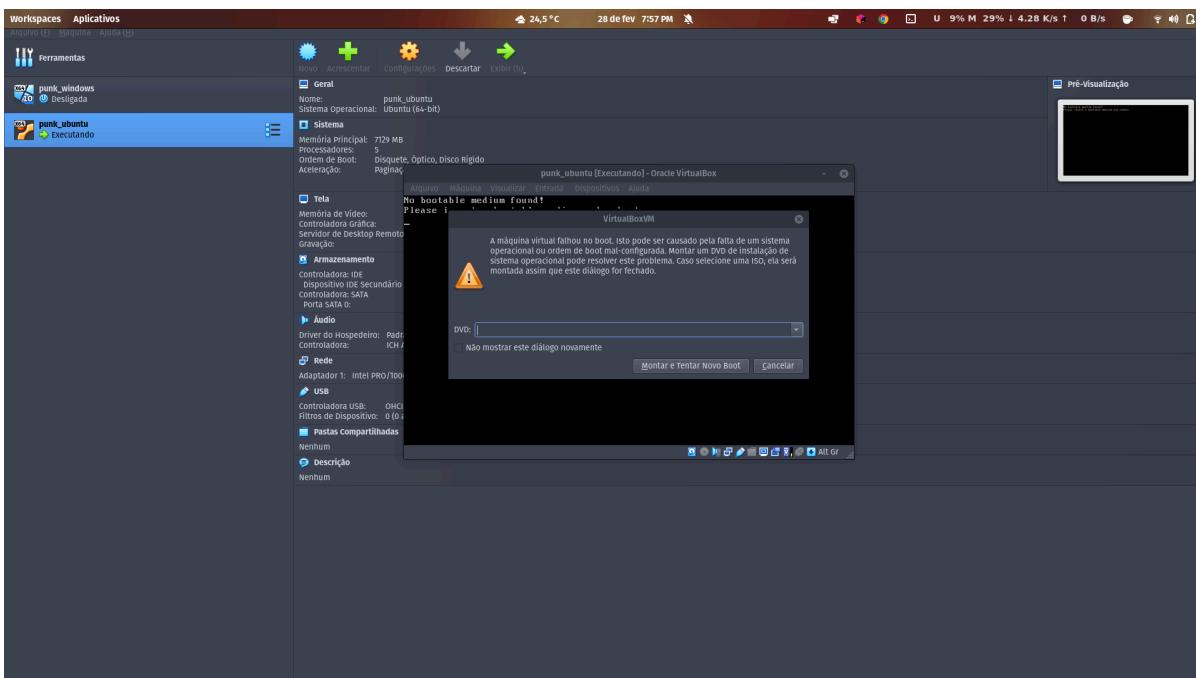
Ao executar as máquinas, nenhum sistema será inicializado, já que não foi definida nenhuma ISO (Imagem de um Sistema Operacional). Assim, as máquinas ficam em seu estado puro, sem nenhum sistema operacional, e são inutilizáveis.

Windows



Tela inicial da máquina virtual Windows sem sistema operacional

Ubuntu



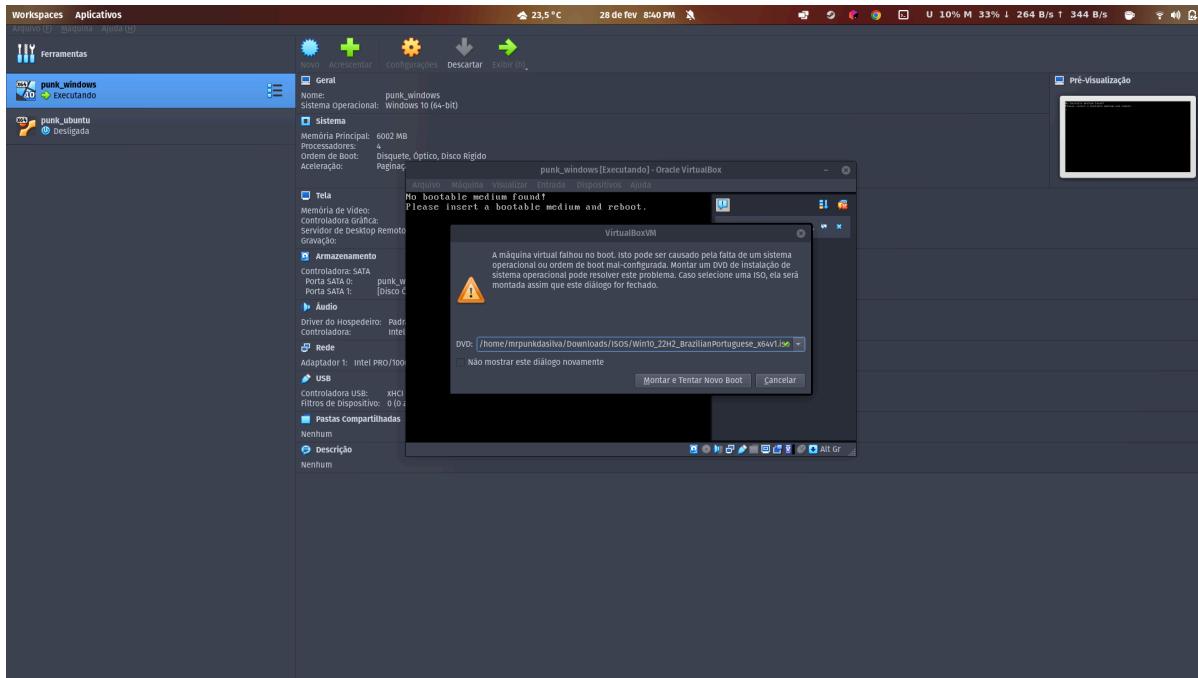
Tela inicial da máquina virtual Ubuntu sem sistema operacional

Configurando VMs para os SOs

Para tornar as VMs utilizáveis, precisamos definir as ISOs que serão as imagens do sistema usadas para instalar o sistema operacional.

Windows

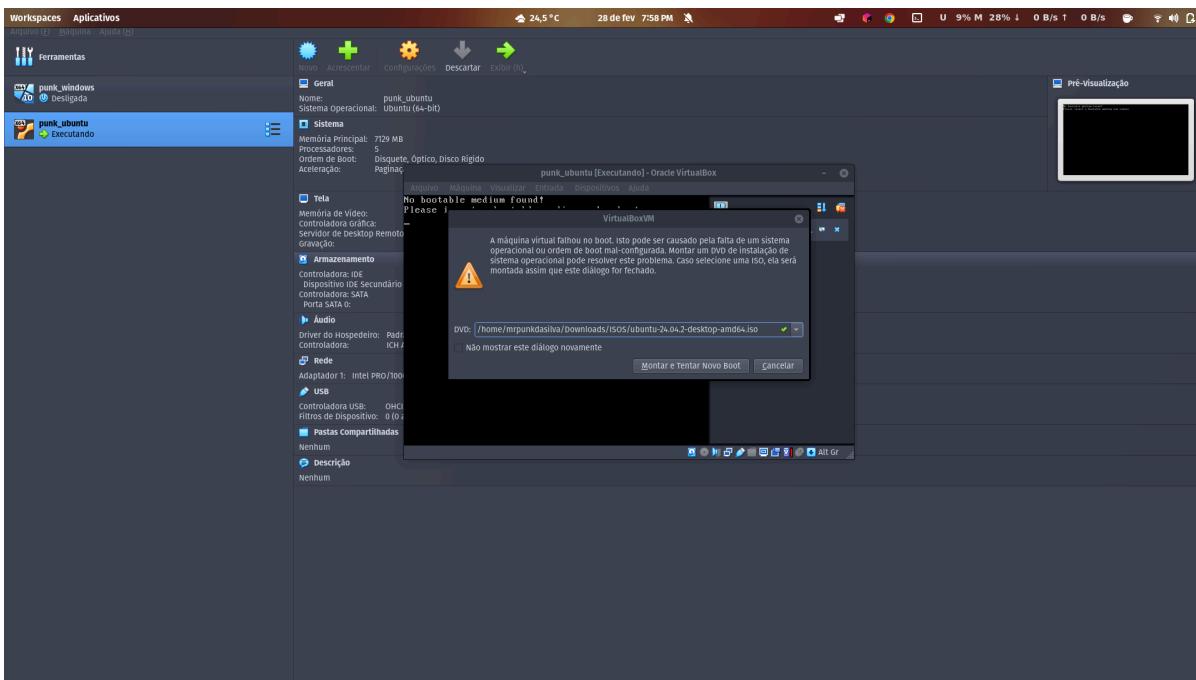
Com a máquina em execução e este pop-up aparecendo, selecionamos onde está a ISO do Windows que foi baixada nos passos anteriores:



Seleção da ISO do Windows para instalação

Ubuntu

Com a máquina em execução e este pop-up aparecendo, selecionamos onde está a ISO do Ubuntu que foi baixada nos passos anteriores:

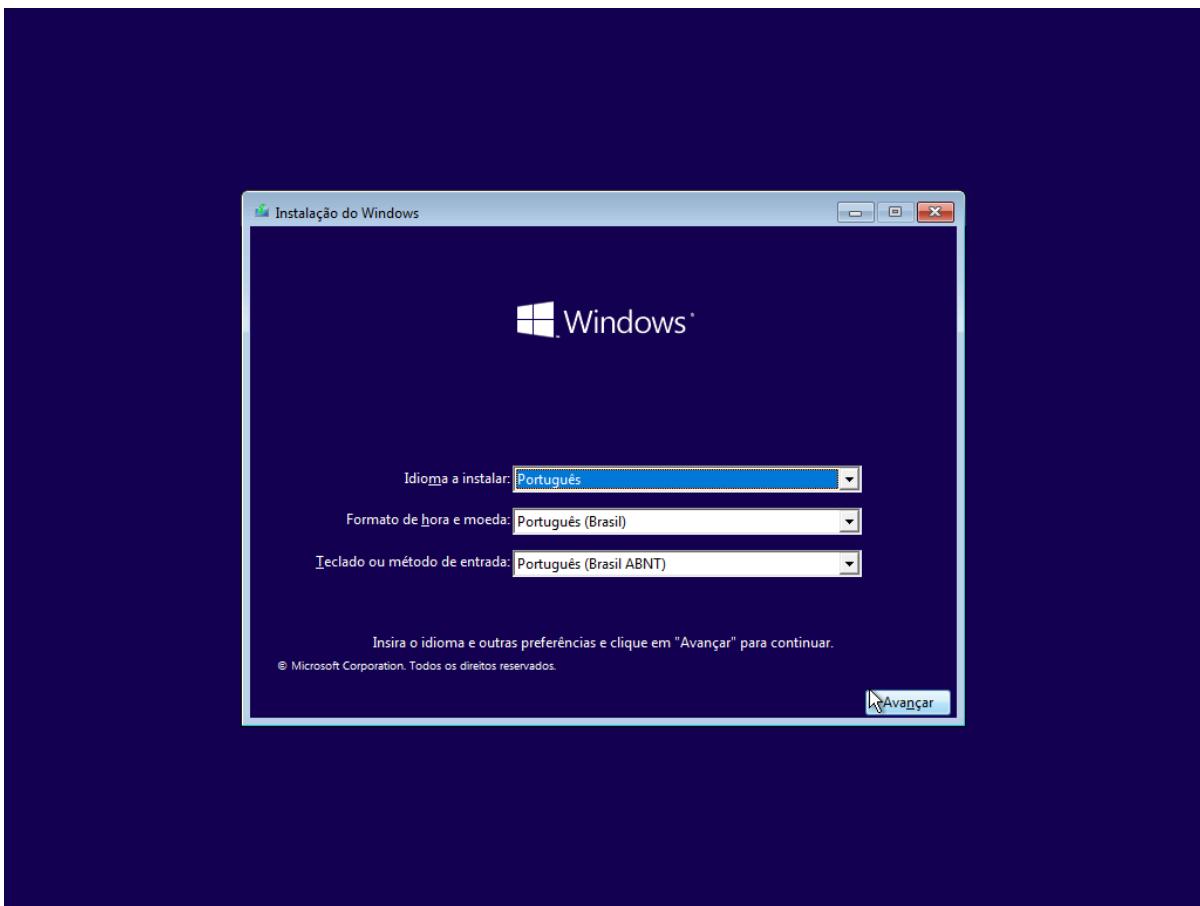


Seleção da ISO do Ubuntu para instalação

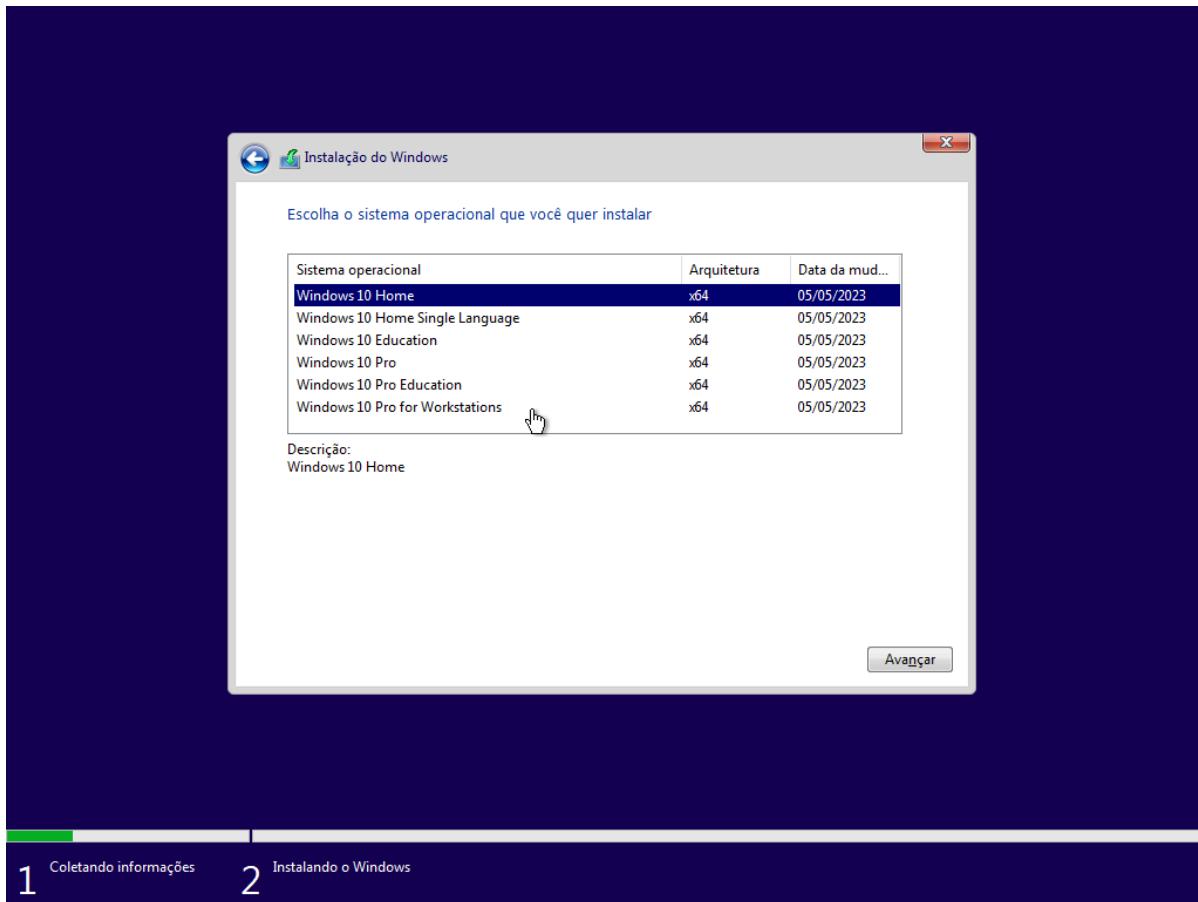
Logar nas VMs

Agora, com tudo o que vimos, podemos fazer a instalação do sistema. Para isso, devemos logar ou entrar nas máquinas que criamos e então fazer as etapas de instalação do Windows e Ubuntu.

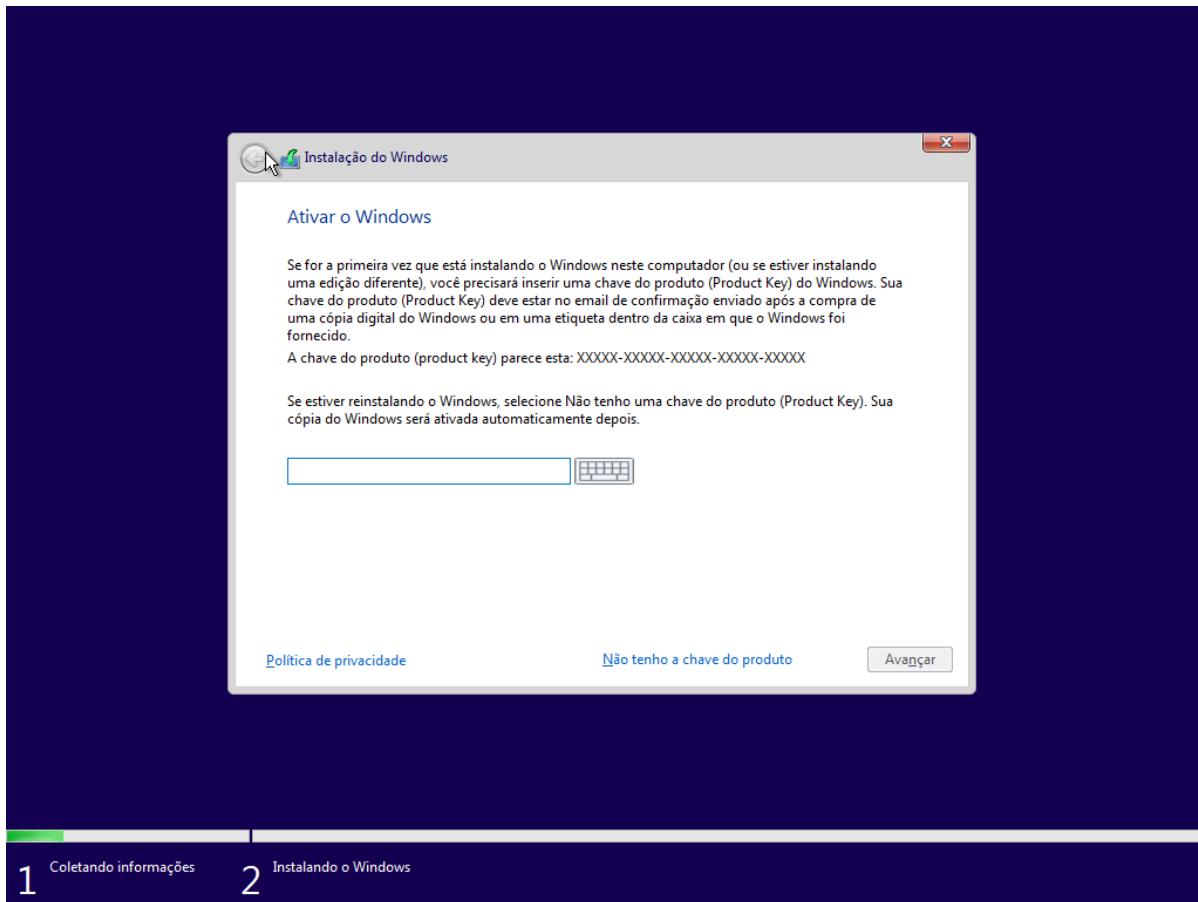
Windows



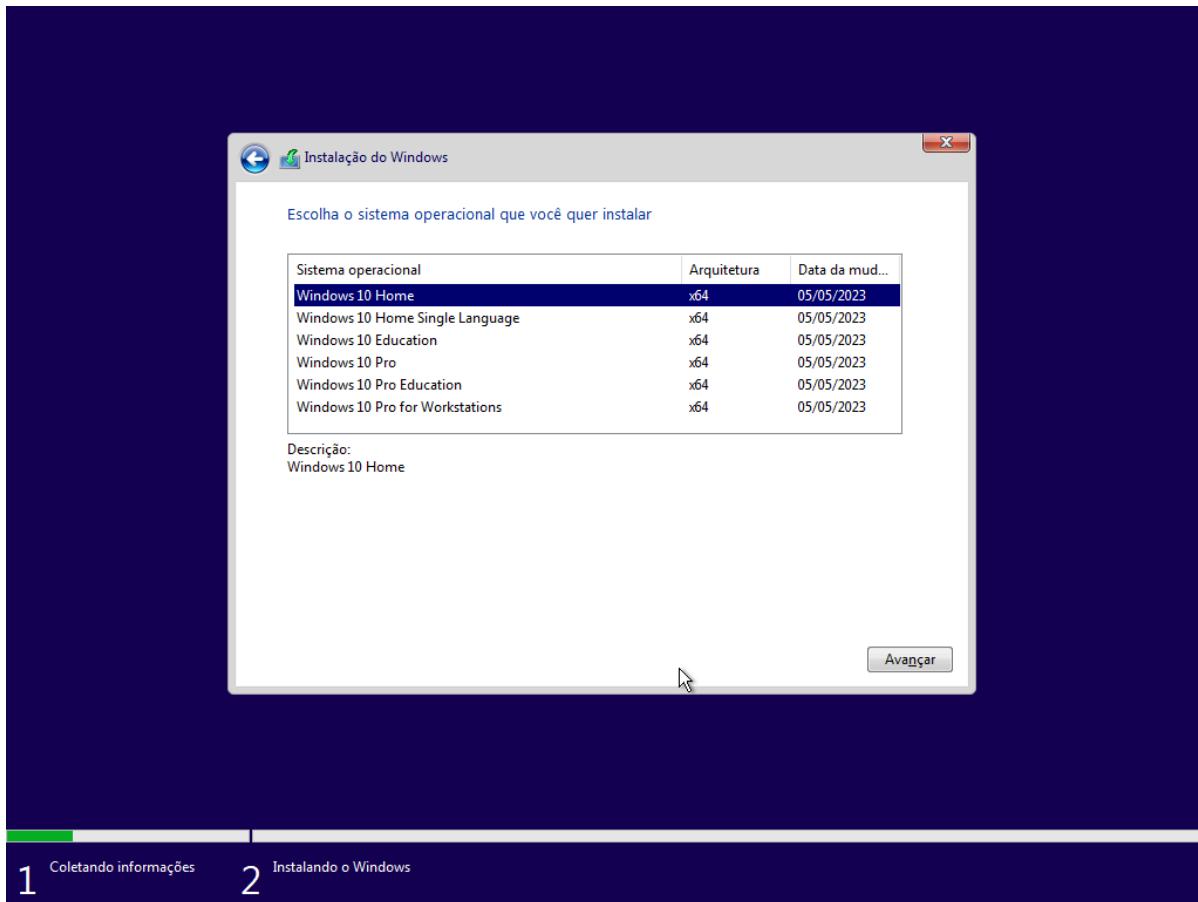
Tela inicial de instalação do Windows



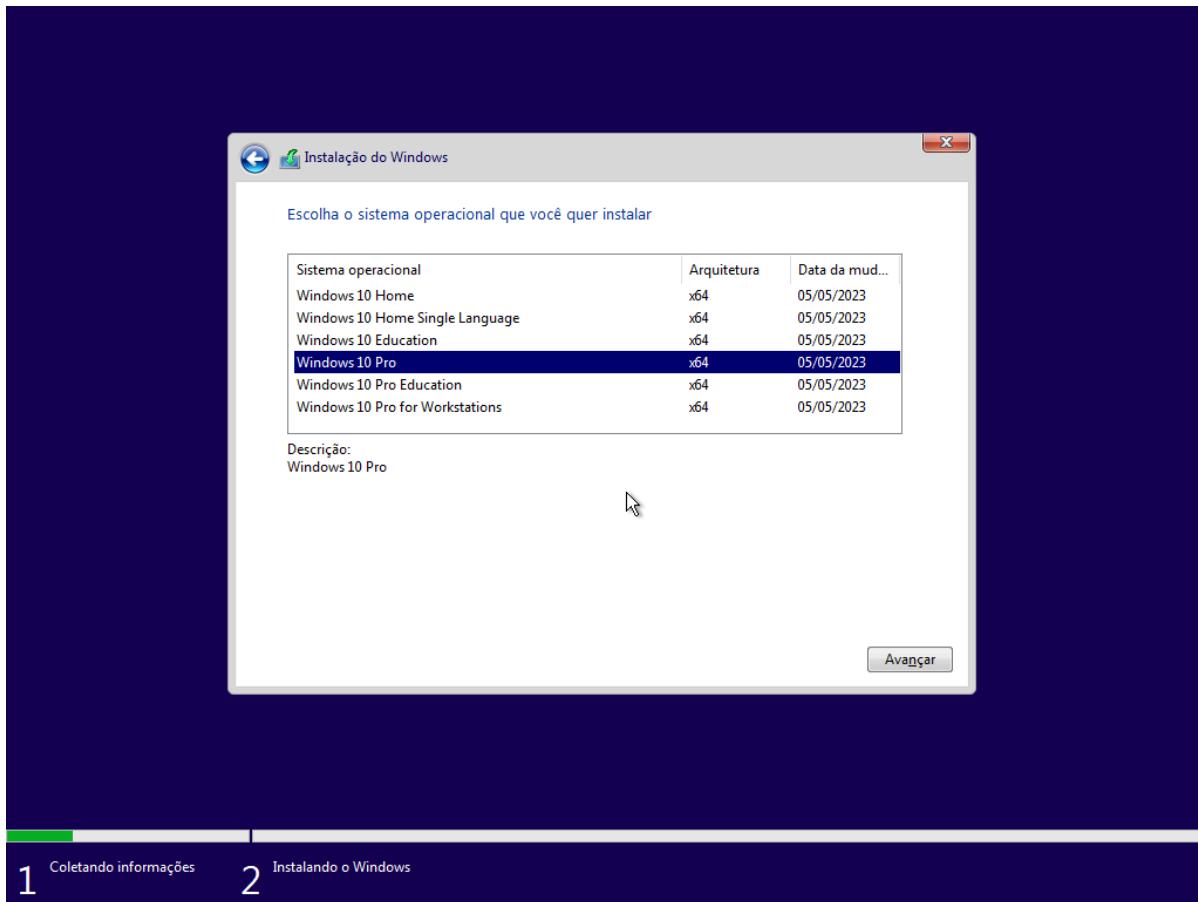
Seleção de idioma, formato de hora e moeda, e layout de teclado



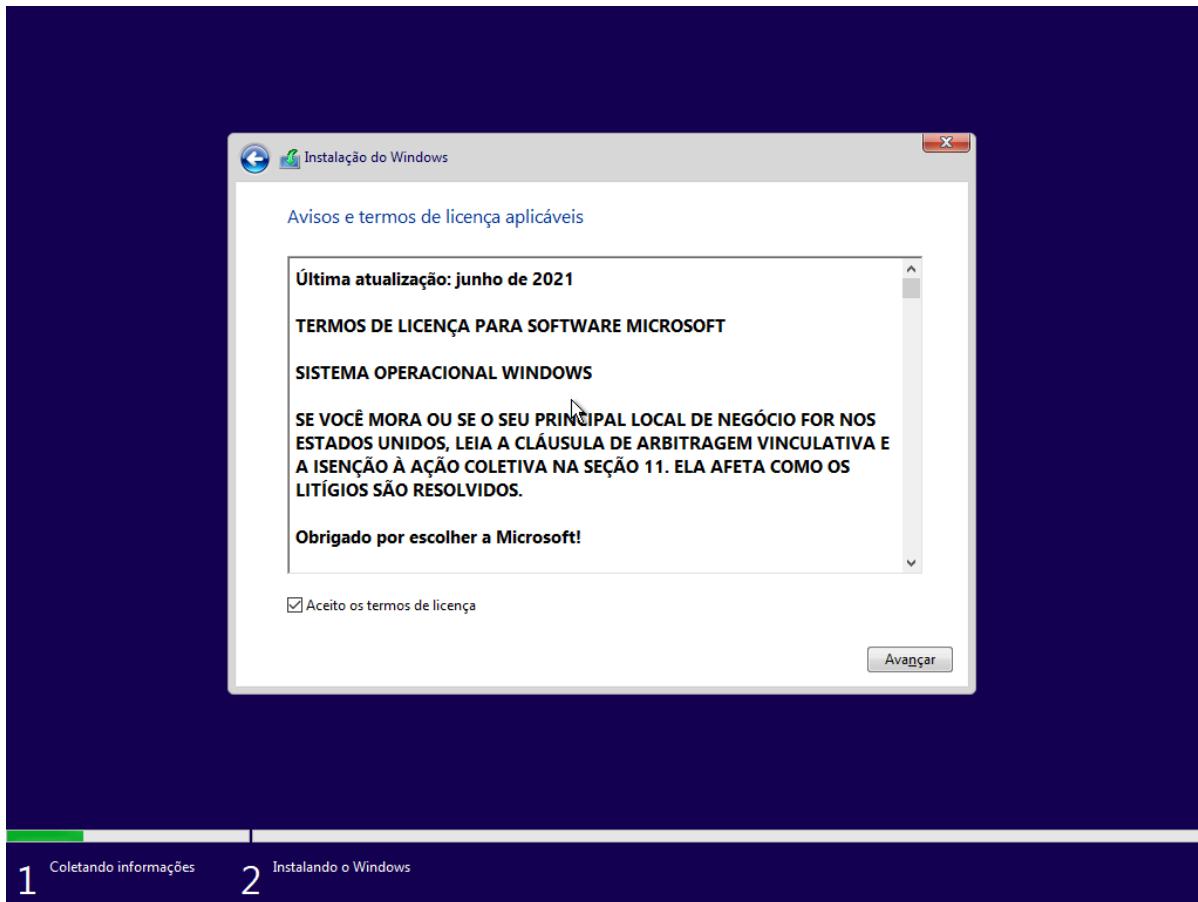
Botão "Instalar agora" para iniciar a instalação do Windows



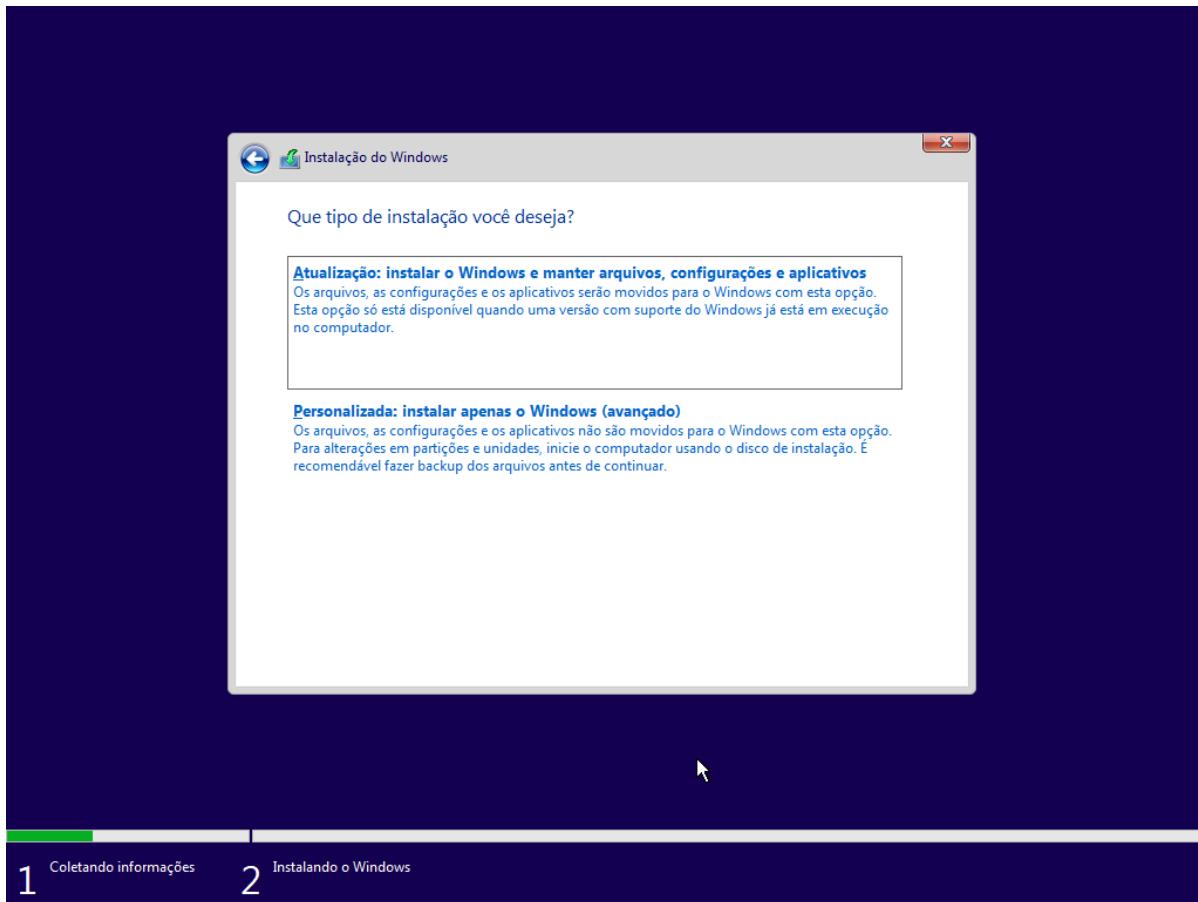
Inserção da chave do produto Windows



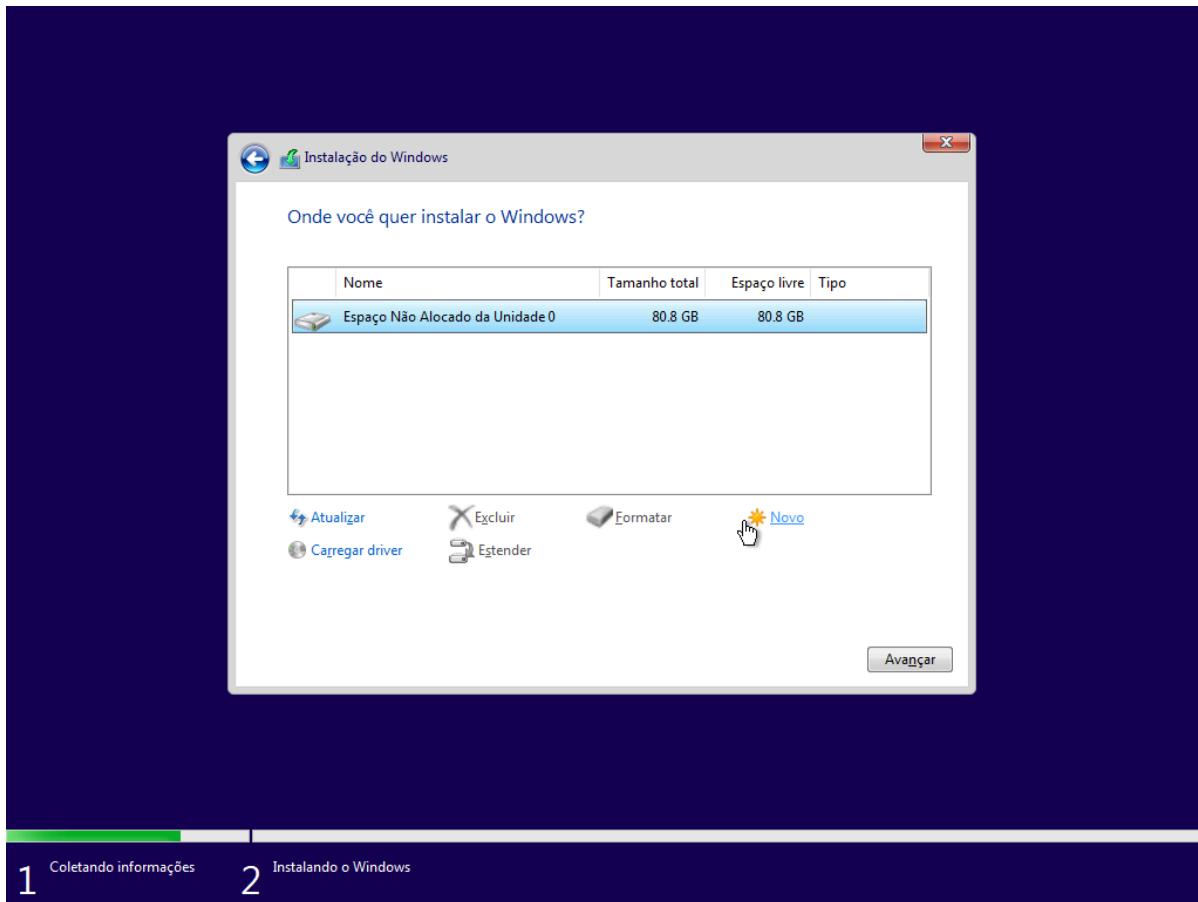
Seleção da versão do Windows a ser instalada



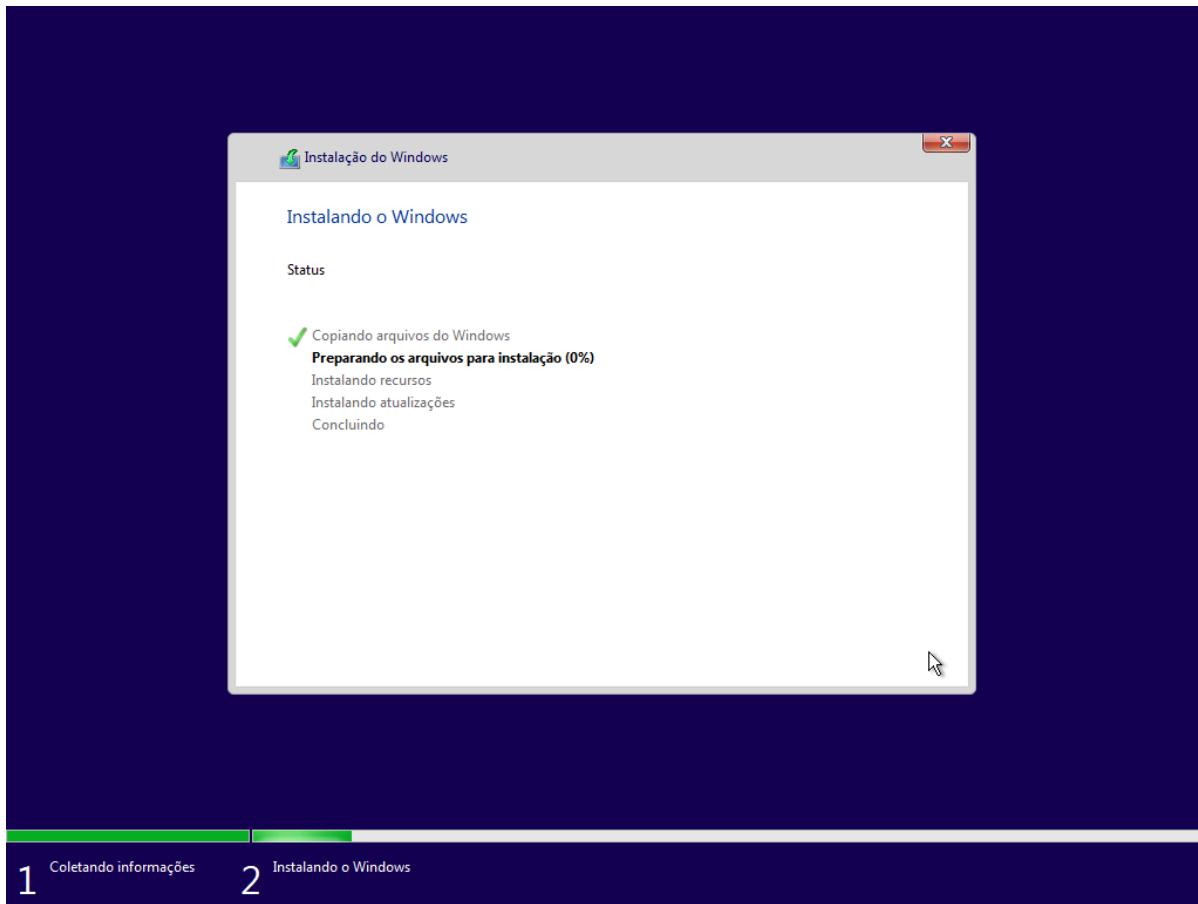
Aceitação dos termos de licença do Windows



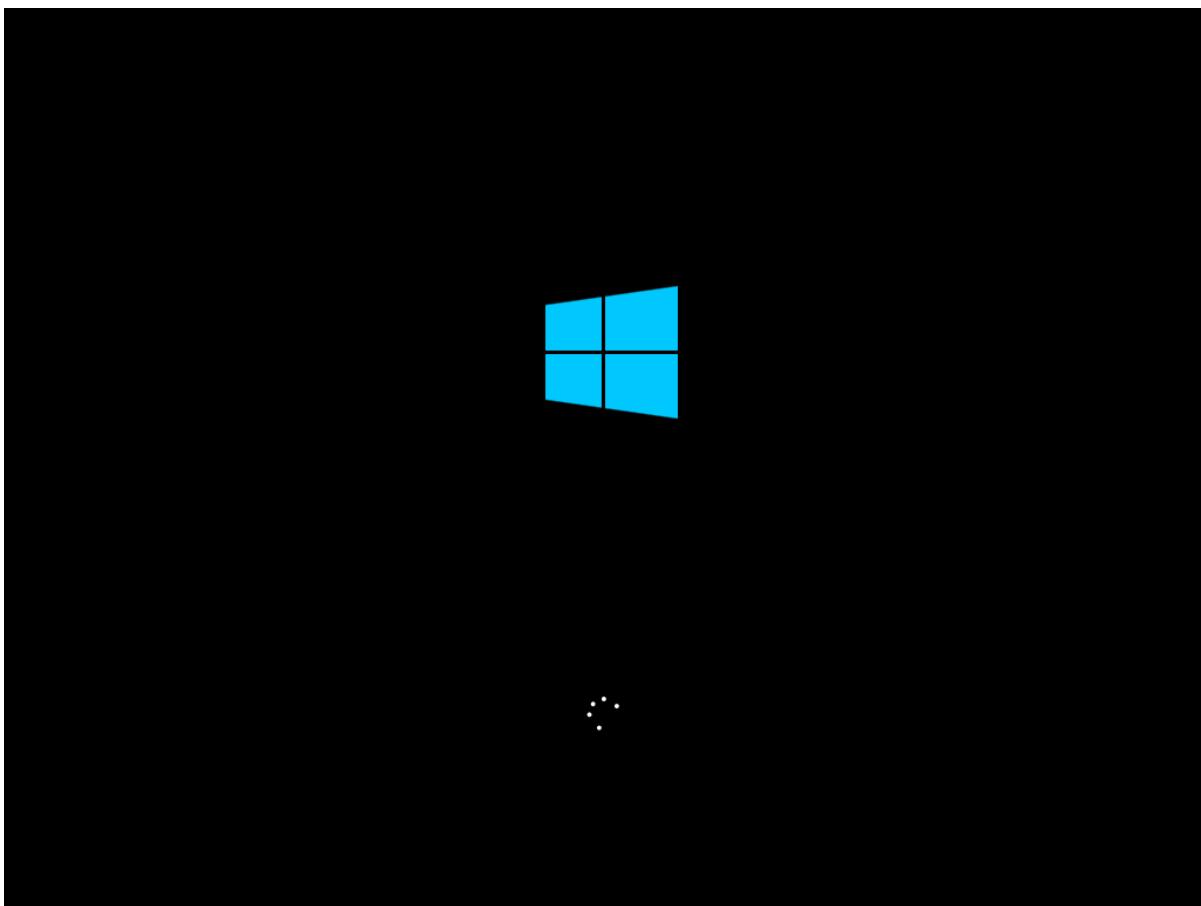
Escolha do tipo de instalação: Atualização ou Personalizada



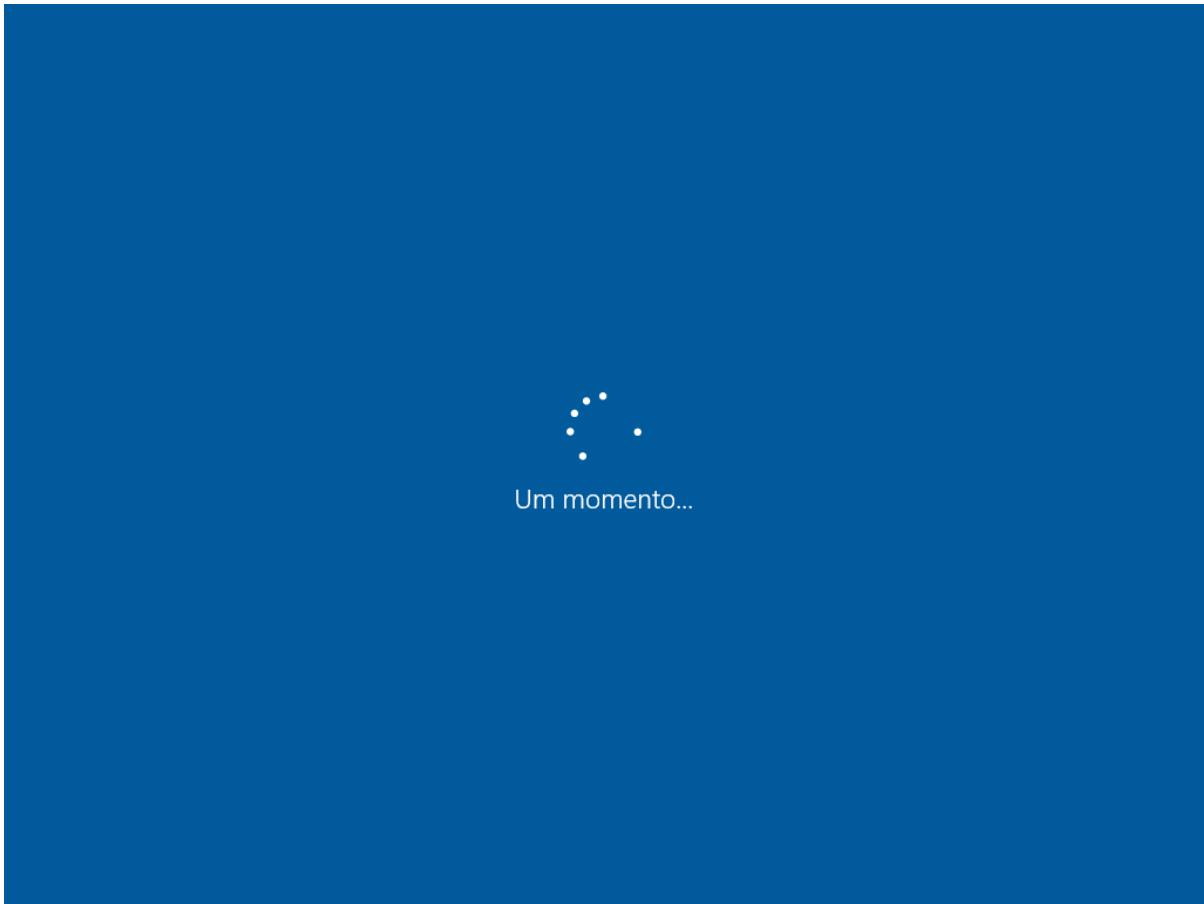
Seleção do disco onde o Windows será instalado



Progresso da instalação do Windows



Reinicialização do sistema após a instalação inicial



Configuração inicial: Seleção de região

Vamos começar com a região. Isto está certo?

Benin

Bermuda

Bolívia

Bonaire, Santo Eustáquio e Saba

Bósnia e Herzegovina

Botsuana

Brasil

Sim



Configuração inicial: Confirmação do layout de teclado

Este é o layout de teclado correto?

Se você também usa outro layout de teclado, pode adicioná-lo em seguida.

Português (Brasil ABNT)

Portugal

Português (Brasil ABNT2)

Albanês

Alemanha

Alemanha (IBM)

América Latina

Sim



Configuração inicial: Opção de adicionar um segundo layout de teclado

Como você gostaria de configurar?



Configurar para uso pessoal

Iremos ajudá-lo a configurar com uma conta pessoal da Microsoft. Você terá controle total sobre este dispositivo.



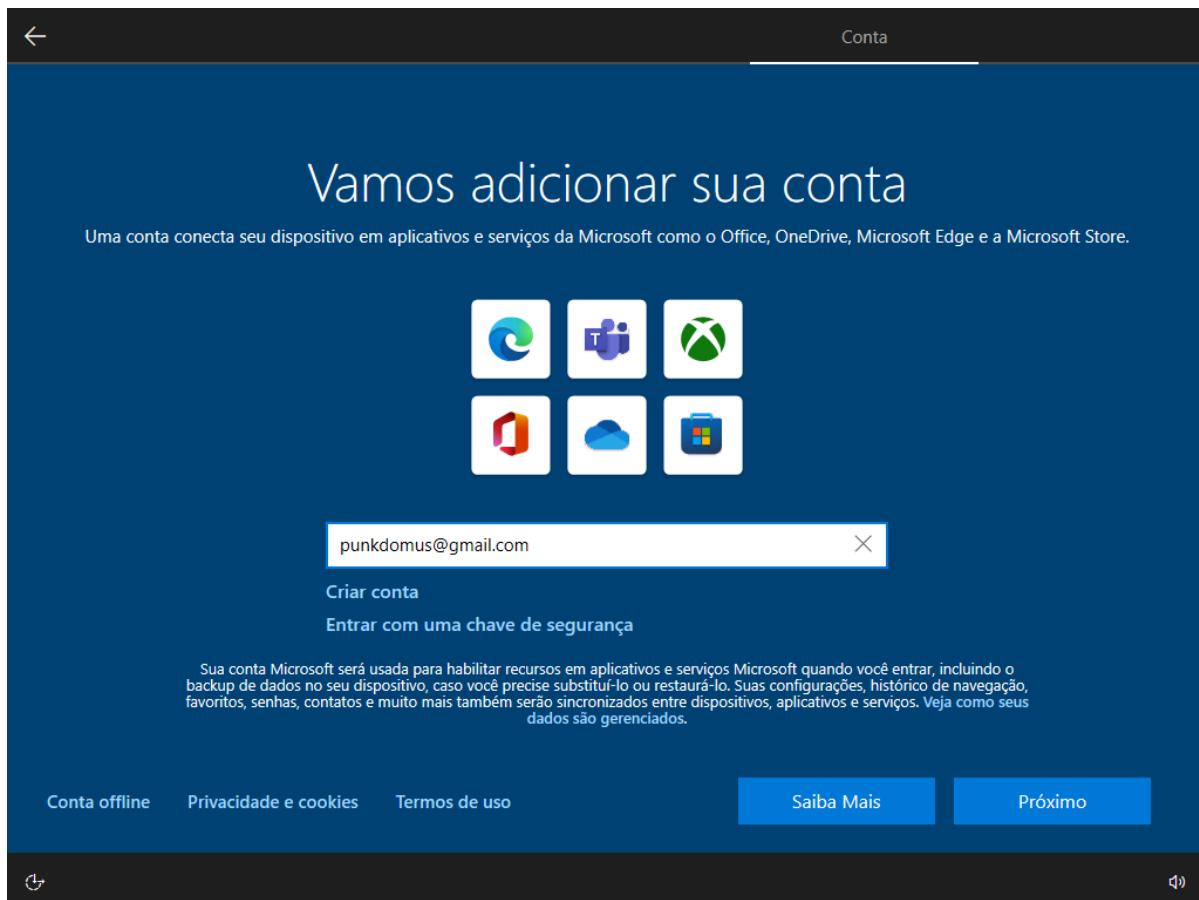
Configurar para uma organização

Você terá acesso aos recursos de sua organização como email, rede, aplicativos e serviços. Sua organização terá controle total sobre este dispositivo.

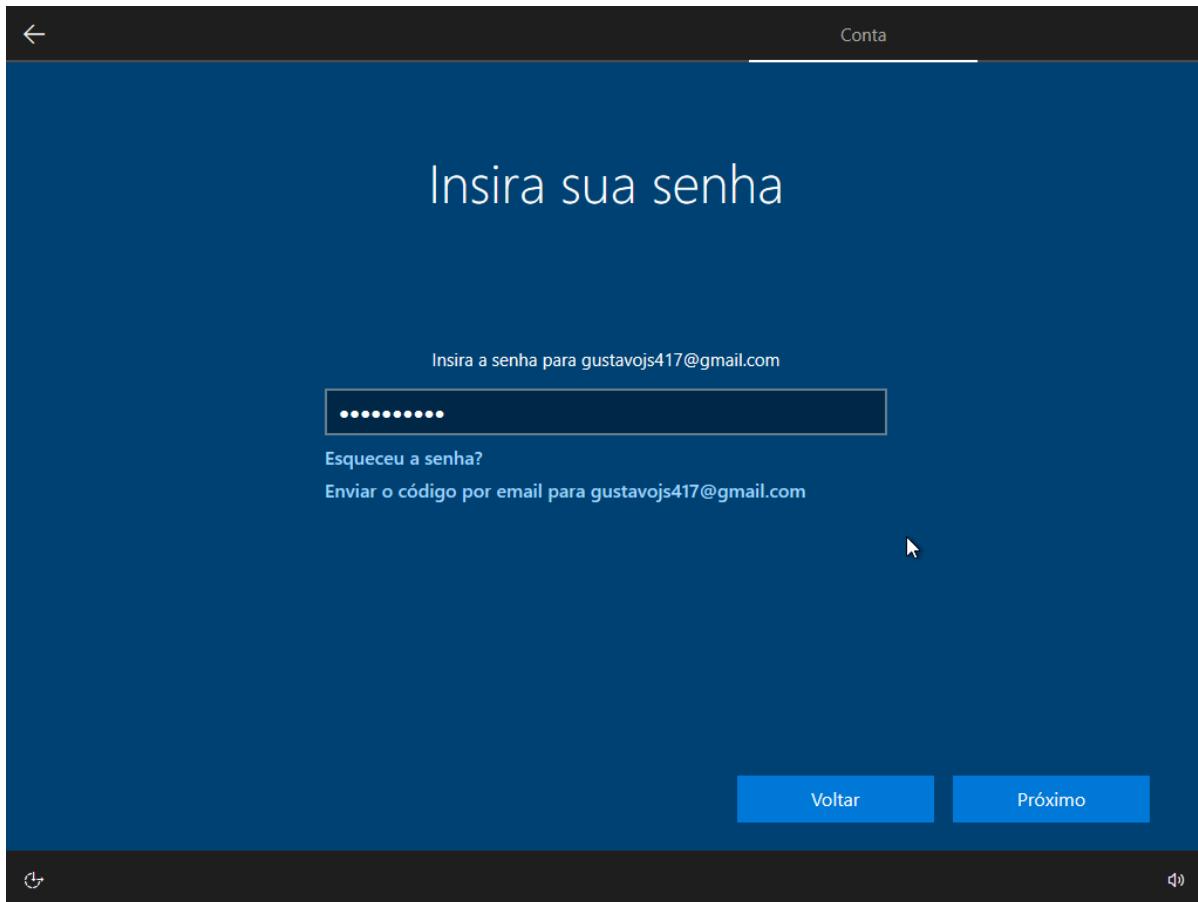
Próximo



Configuração de rede: Conexão à internet



Configuração de conta: Opções de login



Definição de senha para a conta local

Quem usará este computador?

Que nome você deseja usar?



Punk da Silva

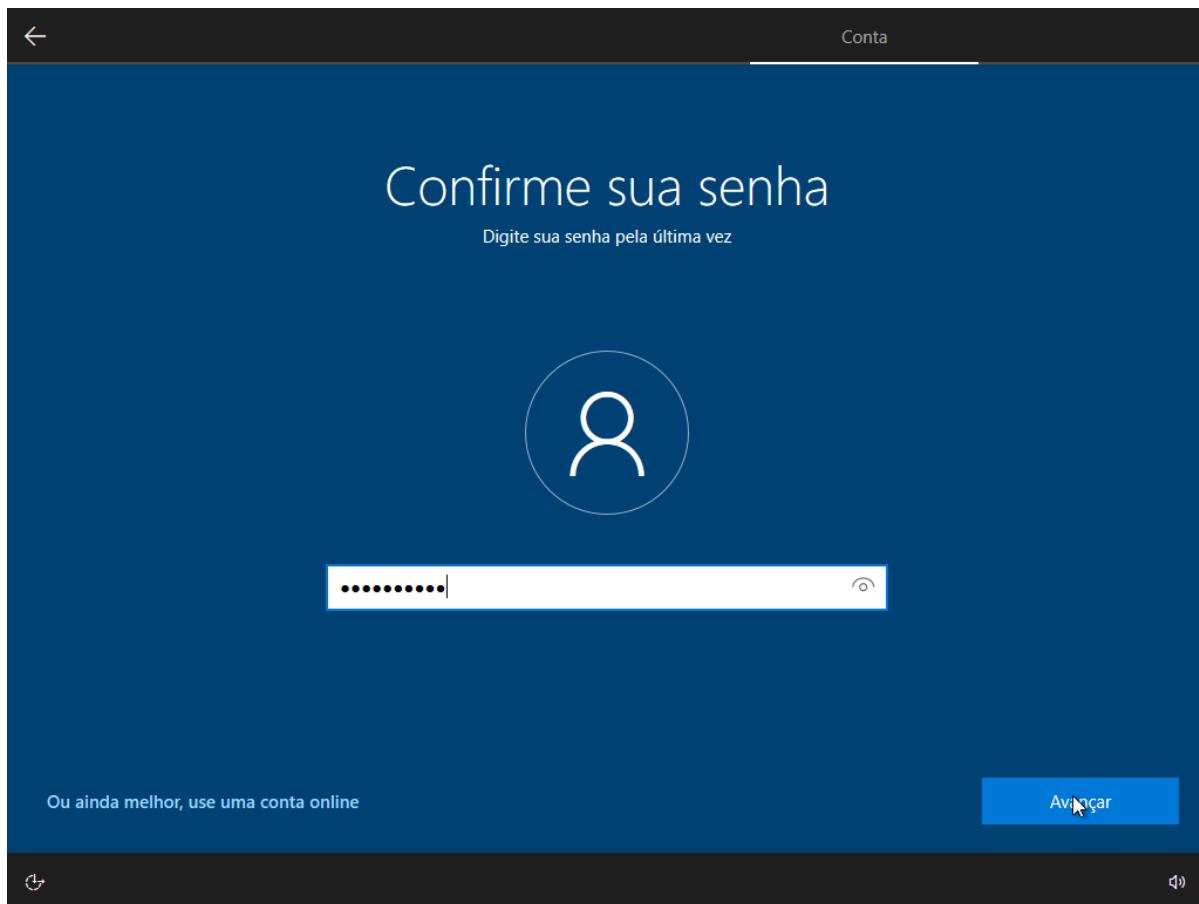
X

Ou ainda melhor, use uma conta online

Avançar



Configuração de perguntas de segurança



Configurações de privacidade: Escolha de permissões



Conta

Criar perguntas de segurança para esta conta

Para o caso de você esquecer sua senha, escolha 3 perguntas de segurança e certifique-se de que as respostas sejam inesquecíveis.



Qual é o nome da cidade onde você nasceu?

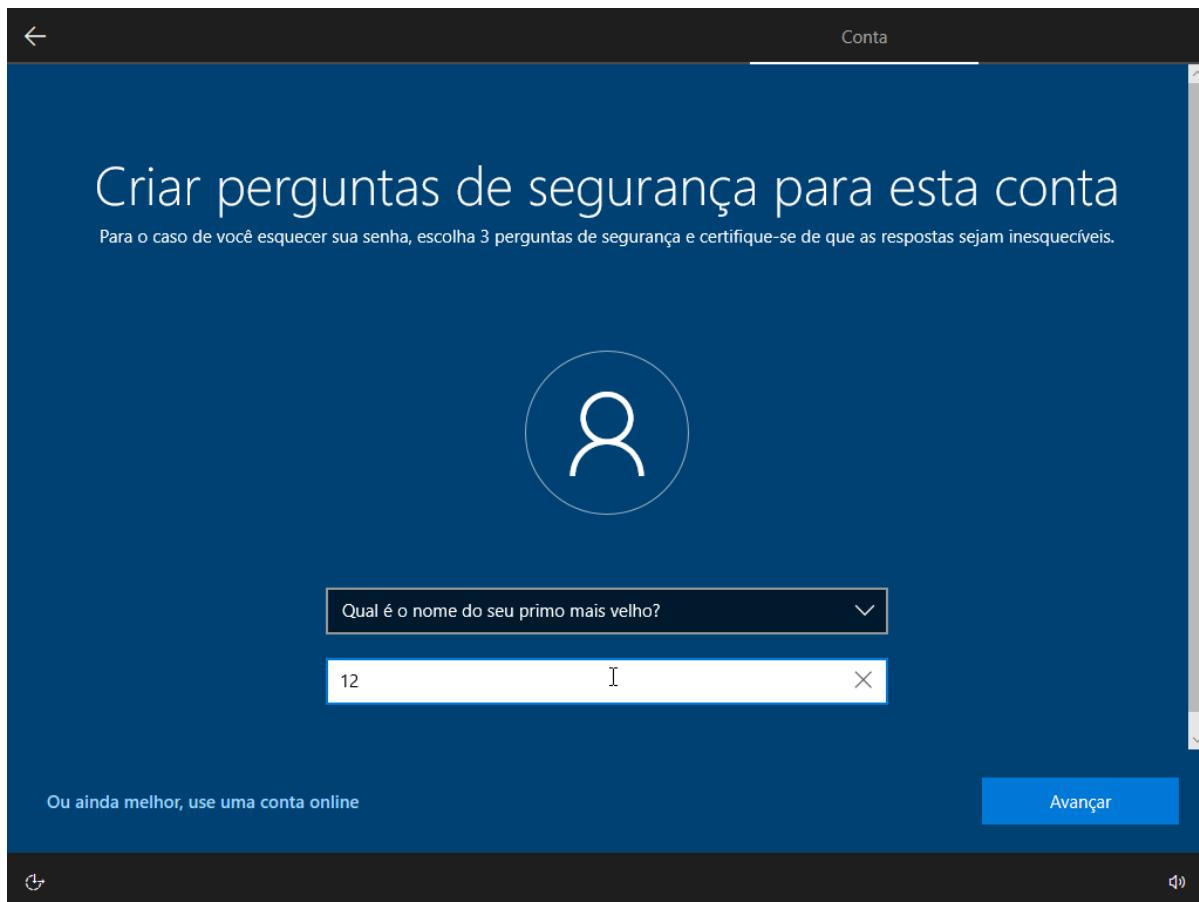
12

Ou ainda melhor, use uma conta online

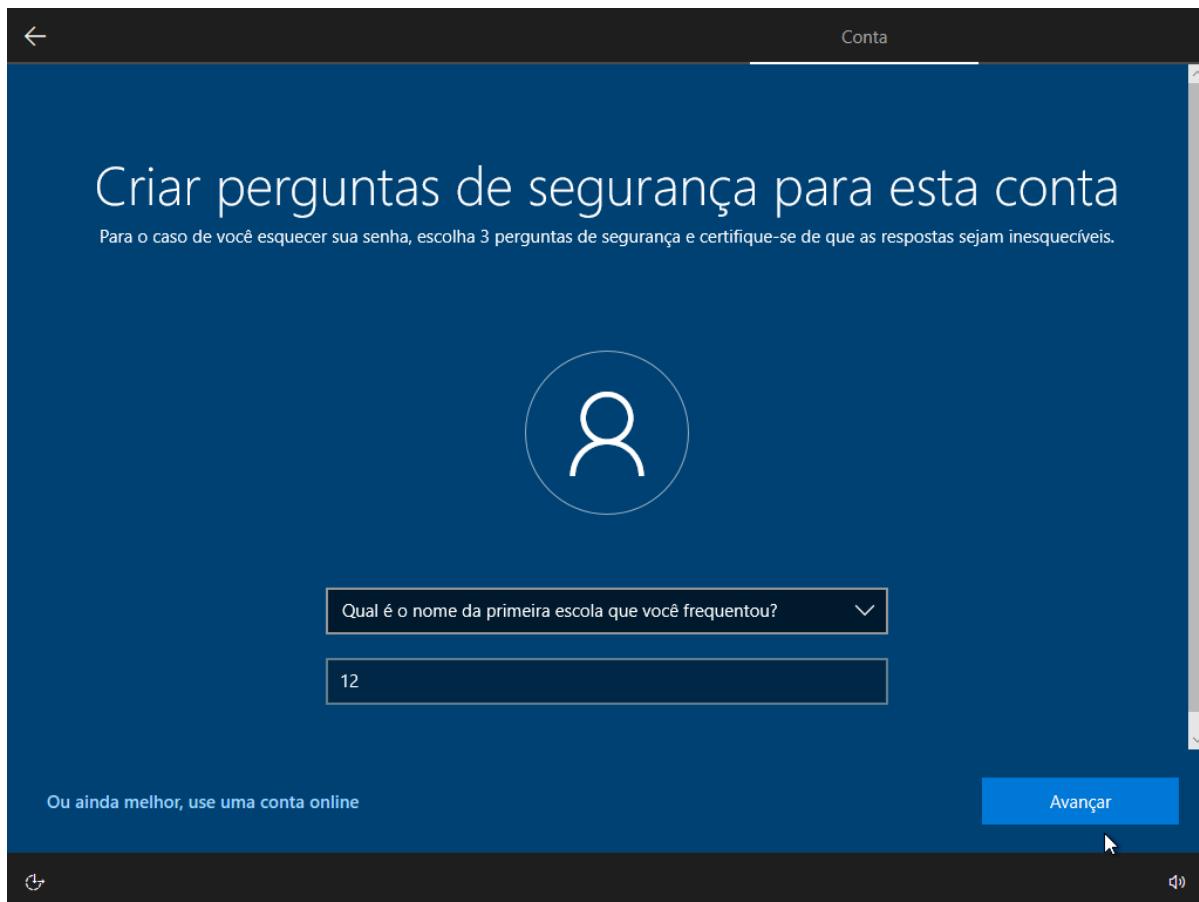
Avançar



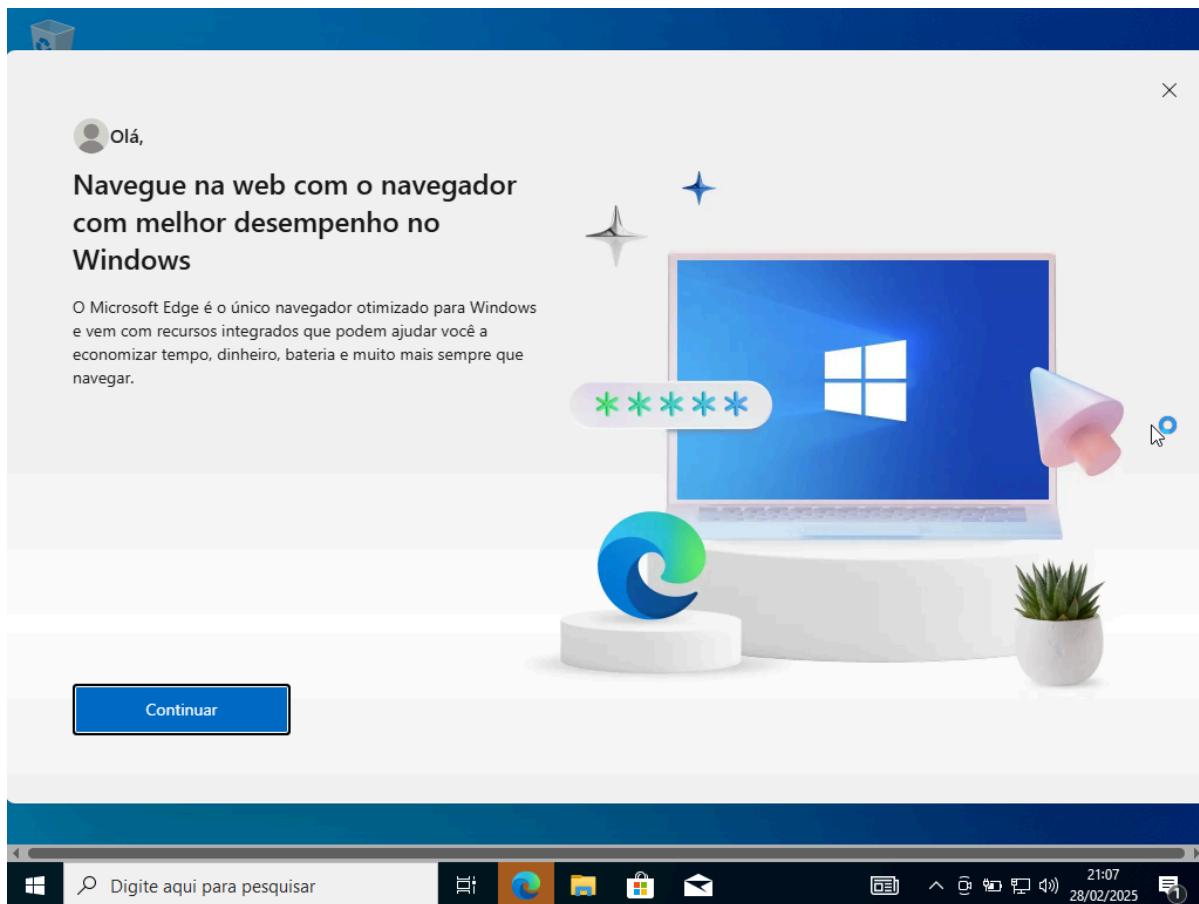
Configuração de experiência personalizada



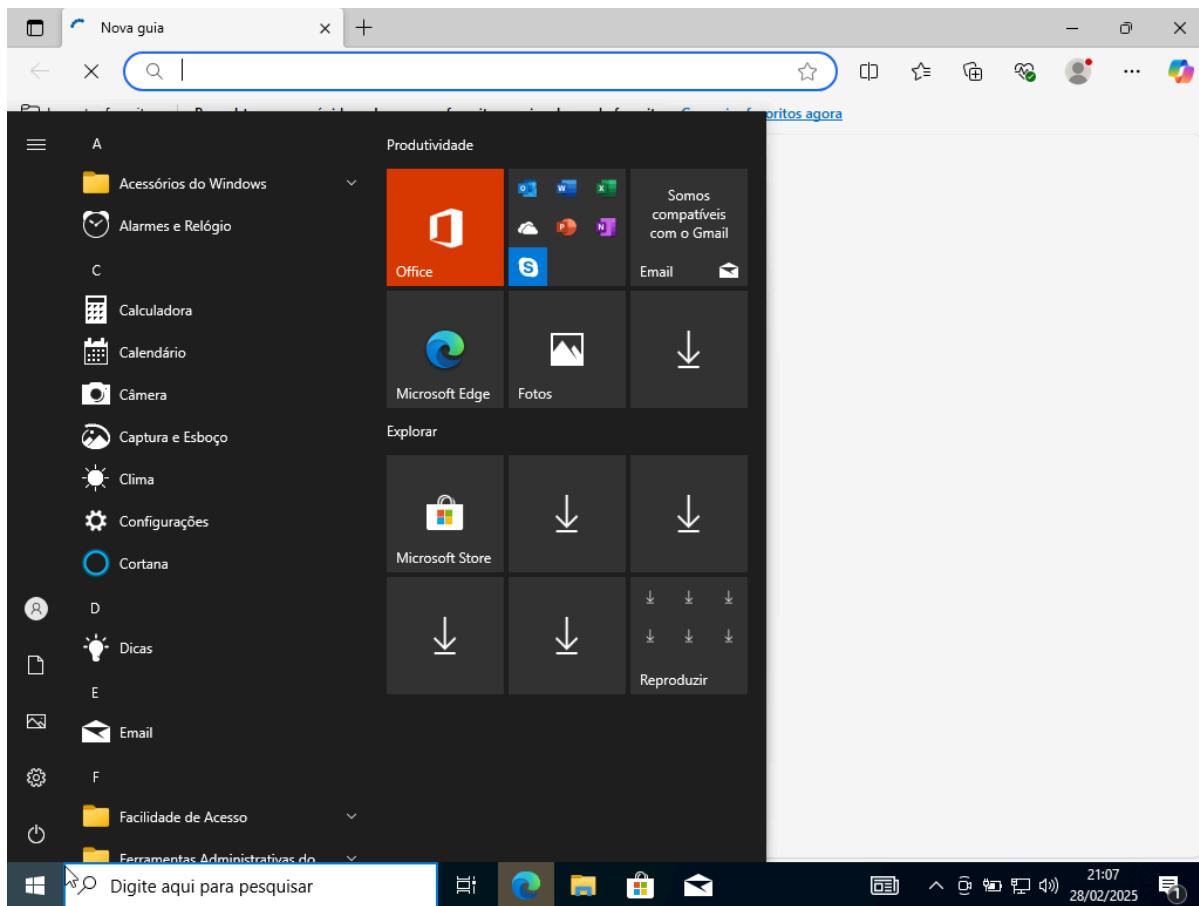
Configuração do assistente digital Cortana



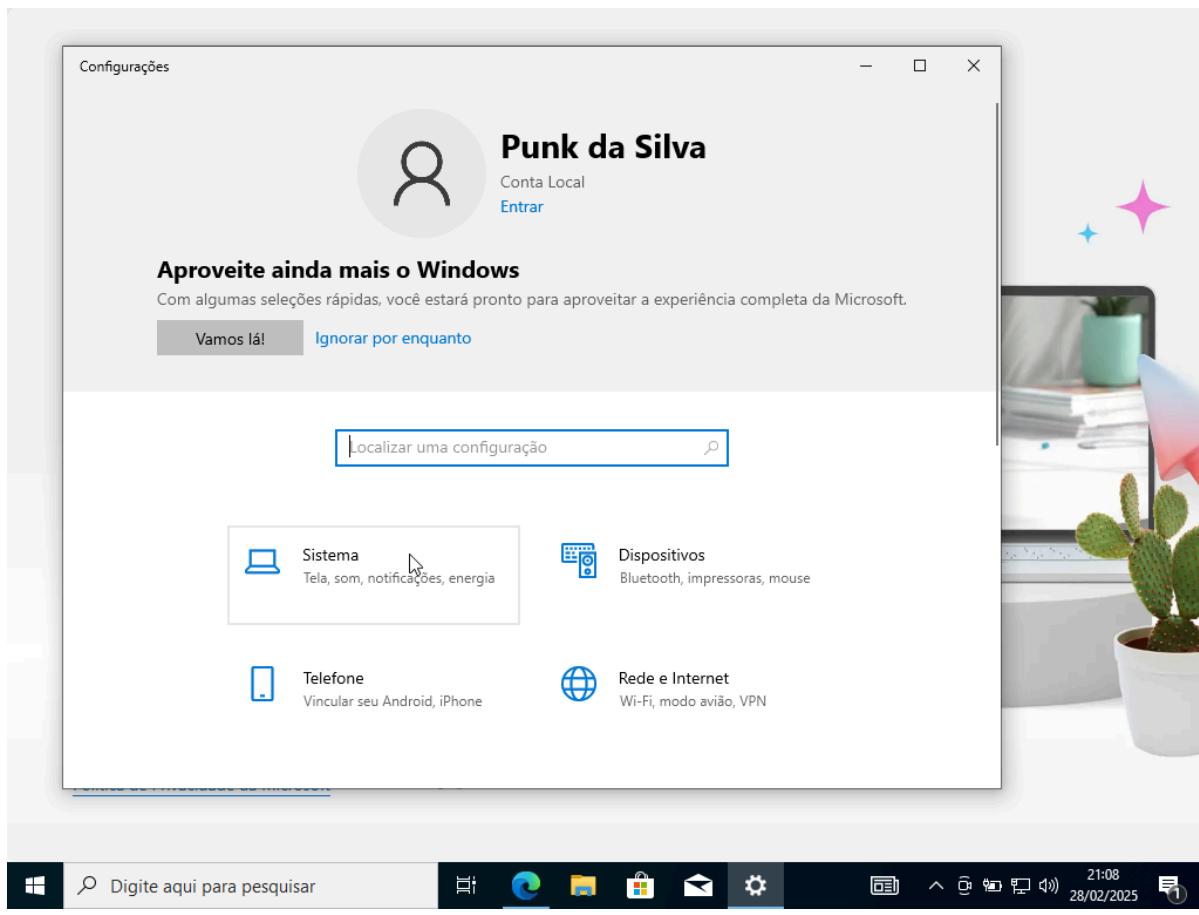
Finalização da configuração do Windows



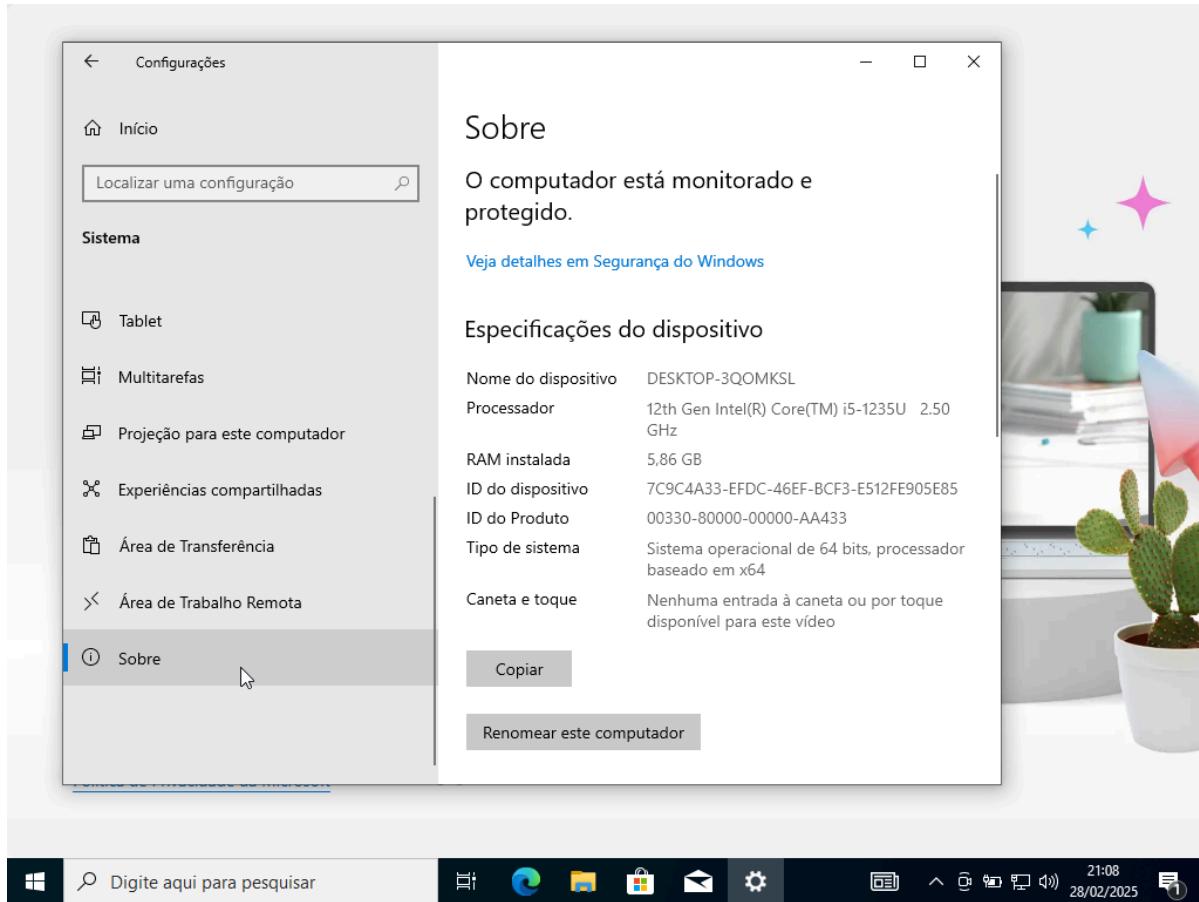
Área de trabalho do Windows após a instalação completa



Menu Iniciar do Windows recém-instalado e digite "Configurações"



Vá em "Sistema"



Na seção "Sobre" do sistema podemos ver as configurações da máquina que foi instalada

Ubuntu

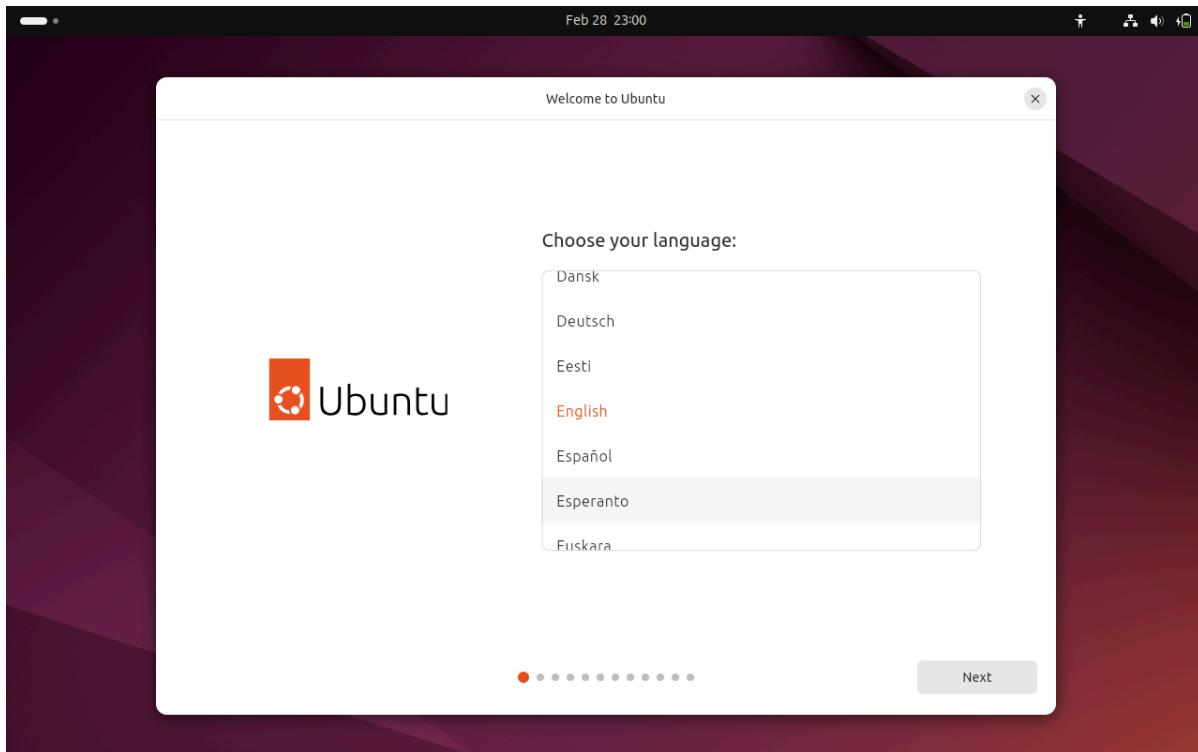
- Configurações de instalação:



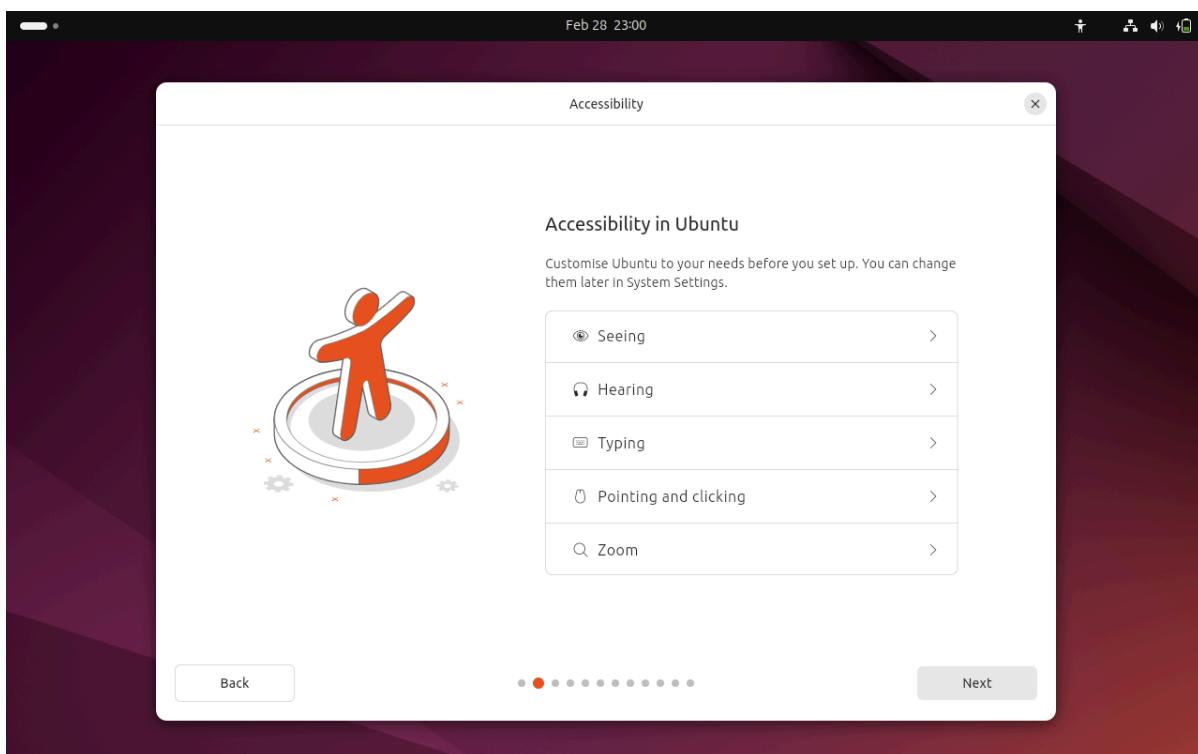
Assim que rodarmos a máquina, vai aparecer a tela do GRUB e então selecionamos a primeira opção

- i** O GRUB (Grand Unified Bootloader) é o menu de inicialização que aparece quando você inicia o sistema Ubuntu. Ele permite que você escolha entre diferentes opções de inicialização.
- A primeira opção geralmente é "Ubuntu", que inicia o sistema normalmente.
 - Outras opções podem incluir modos de recuperação ou versões anteriores do kernel.

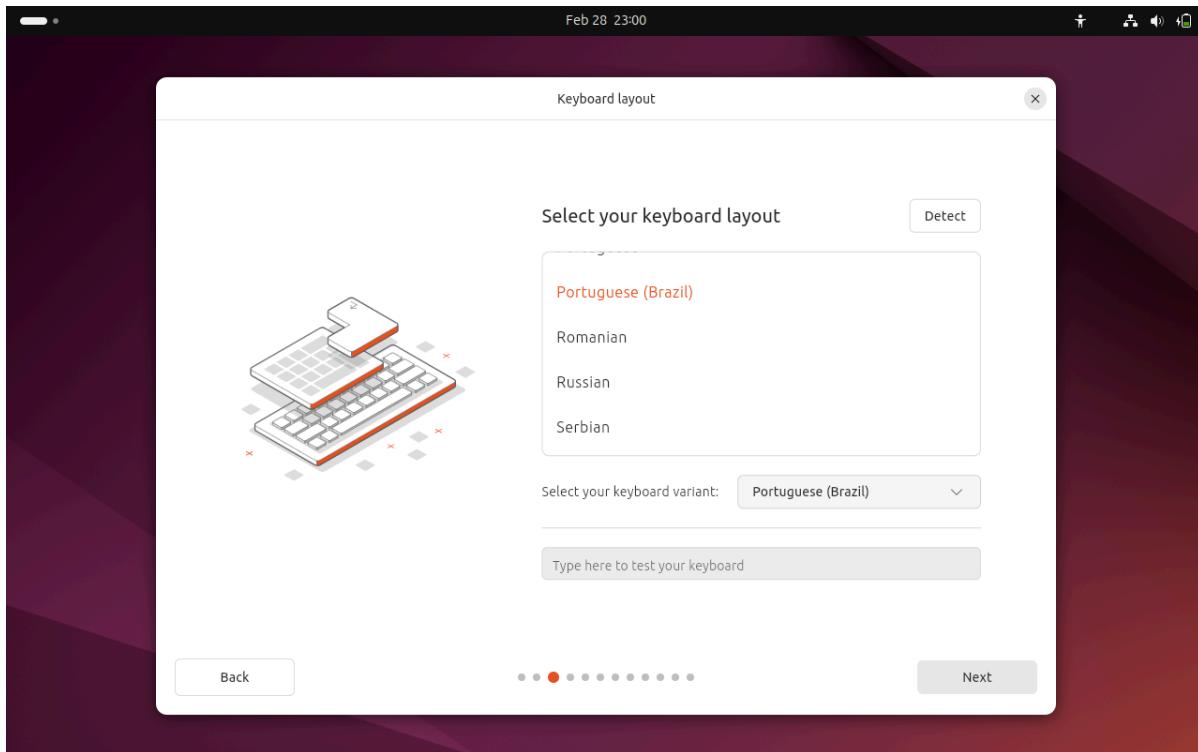
Para a instalação padrão, selecione a primeira opção "Ubuntu" e pressione Enter para continuar.



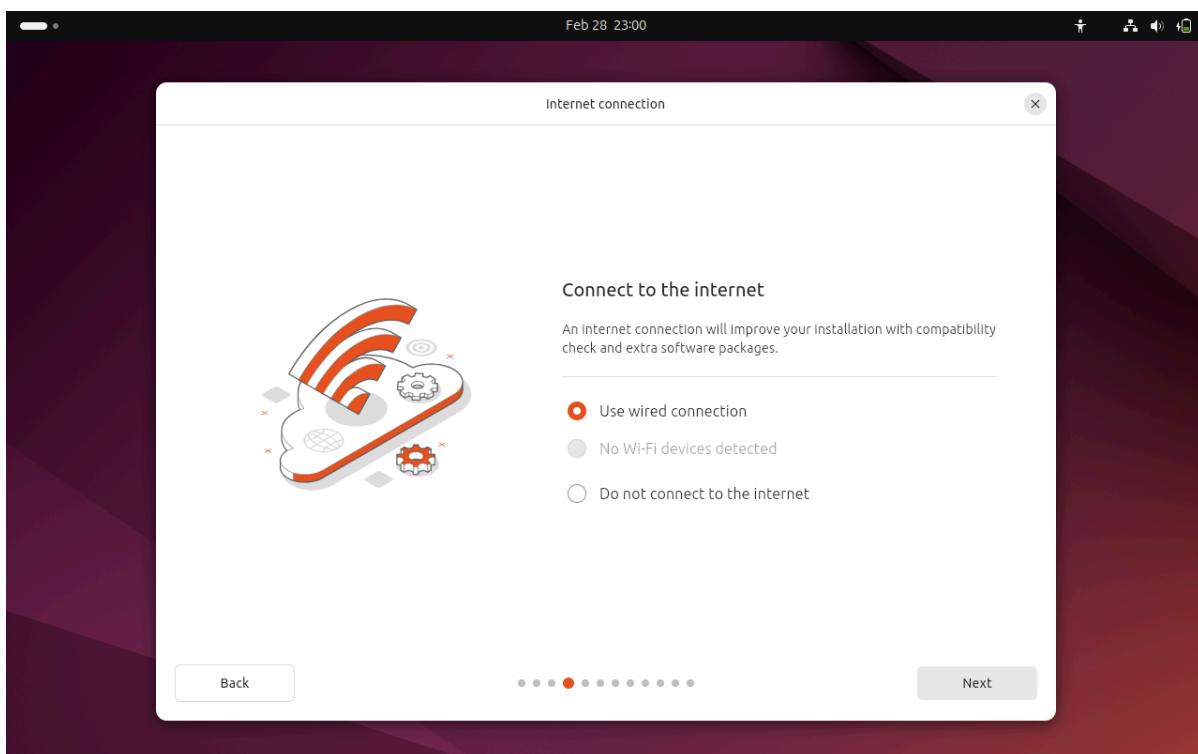
Seleccionamos o idioma



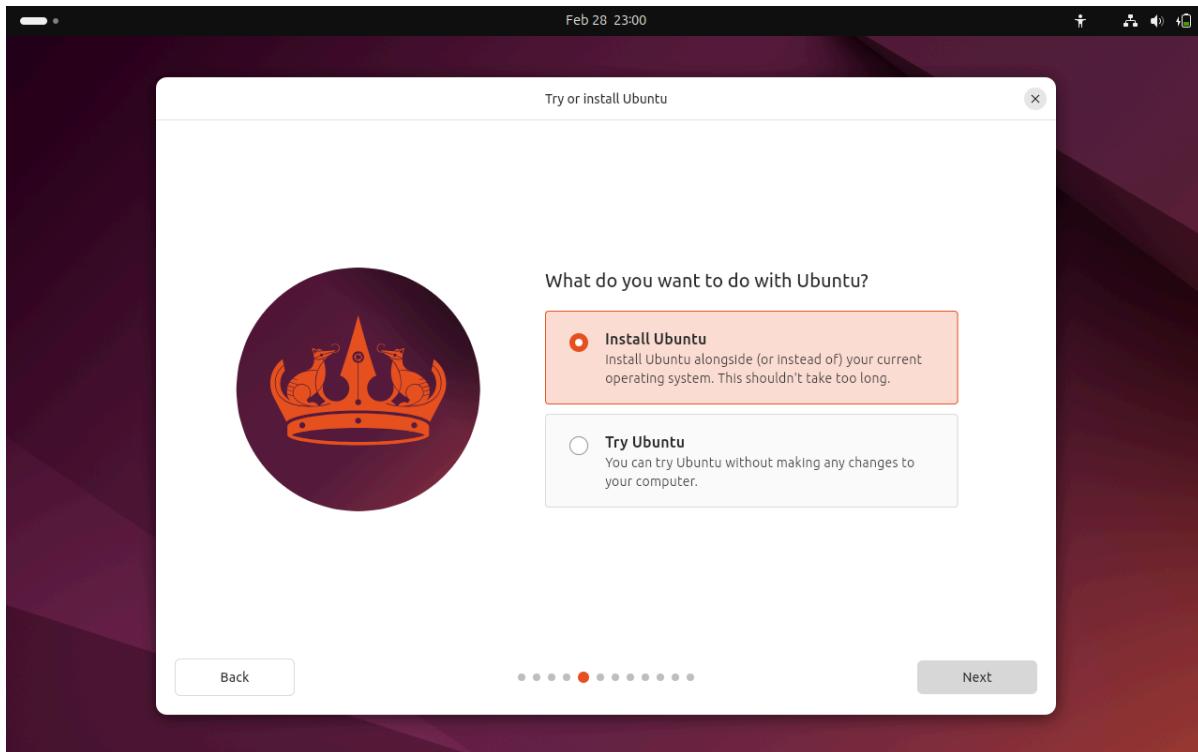
Na parte de Acessibilidade, é só se for necessário



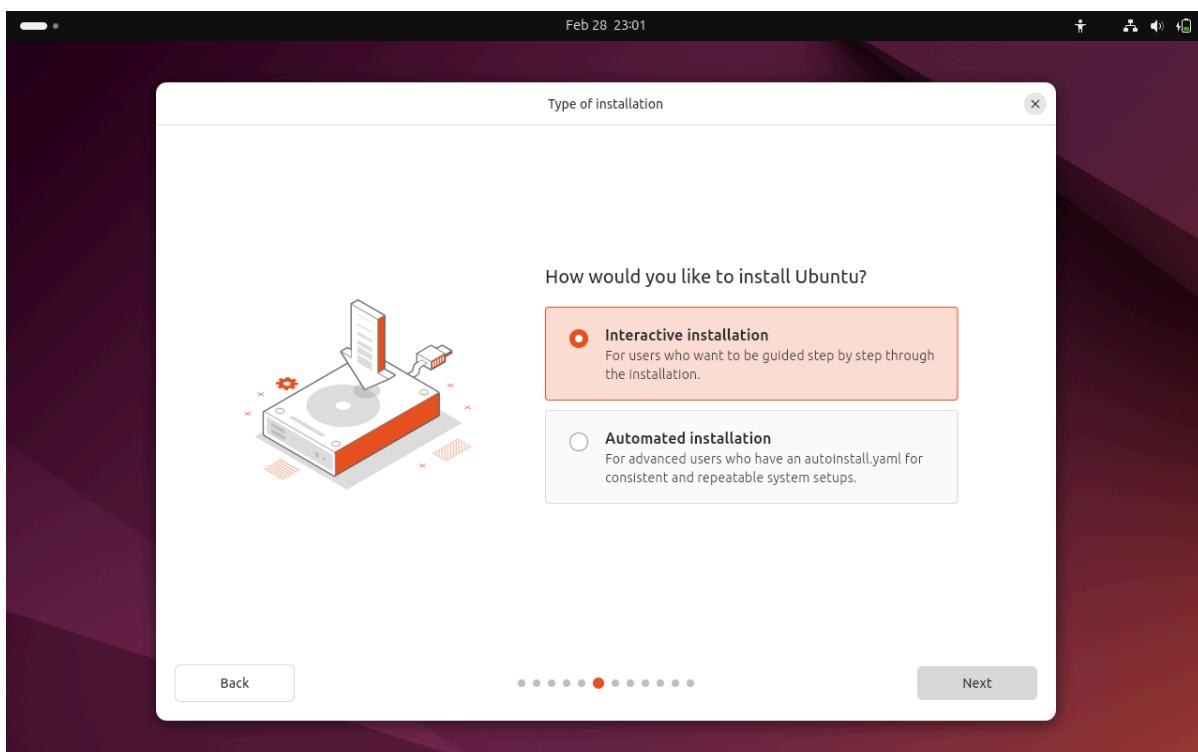
Selecionaremos agora o layout do teclado



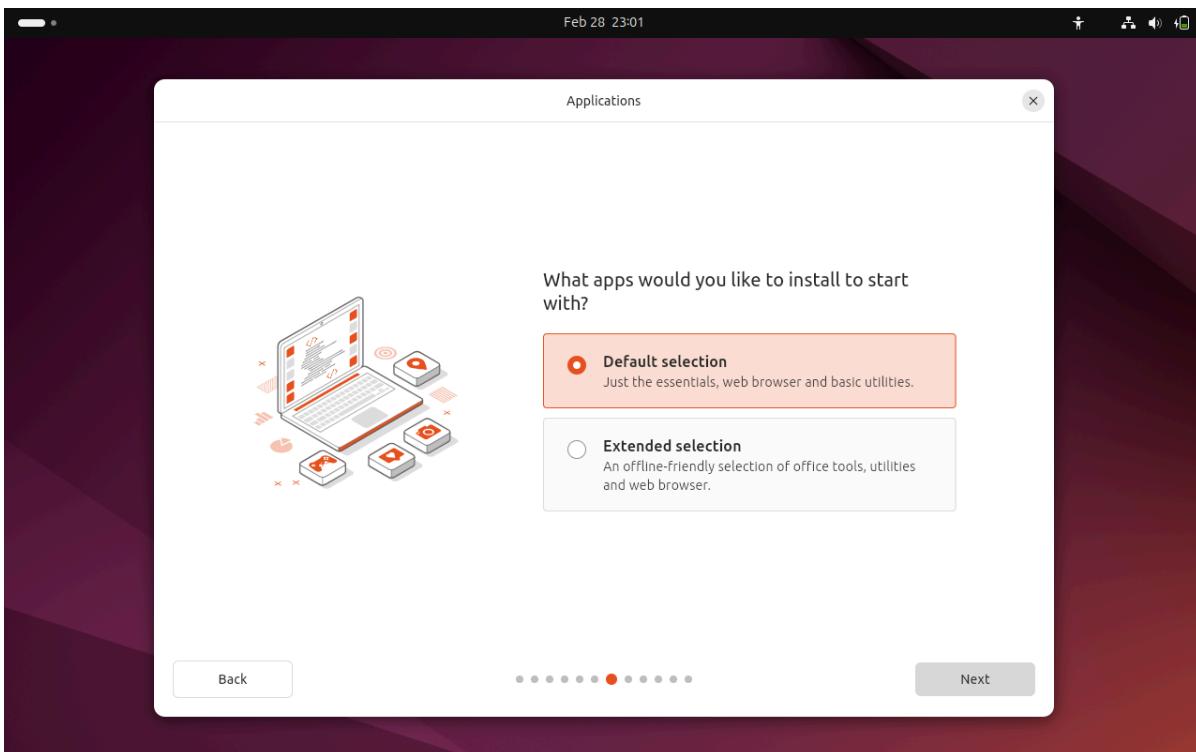
Deixe na forma padrão de conexão



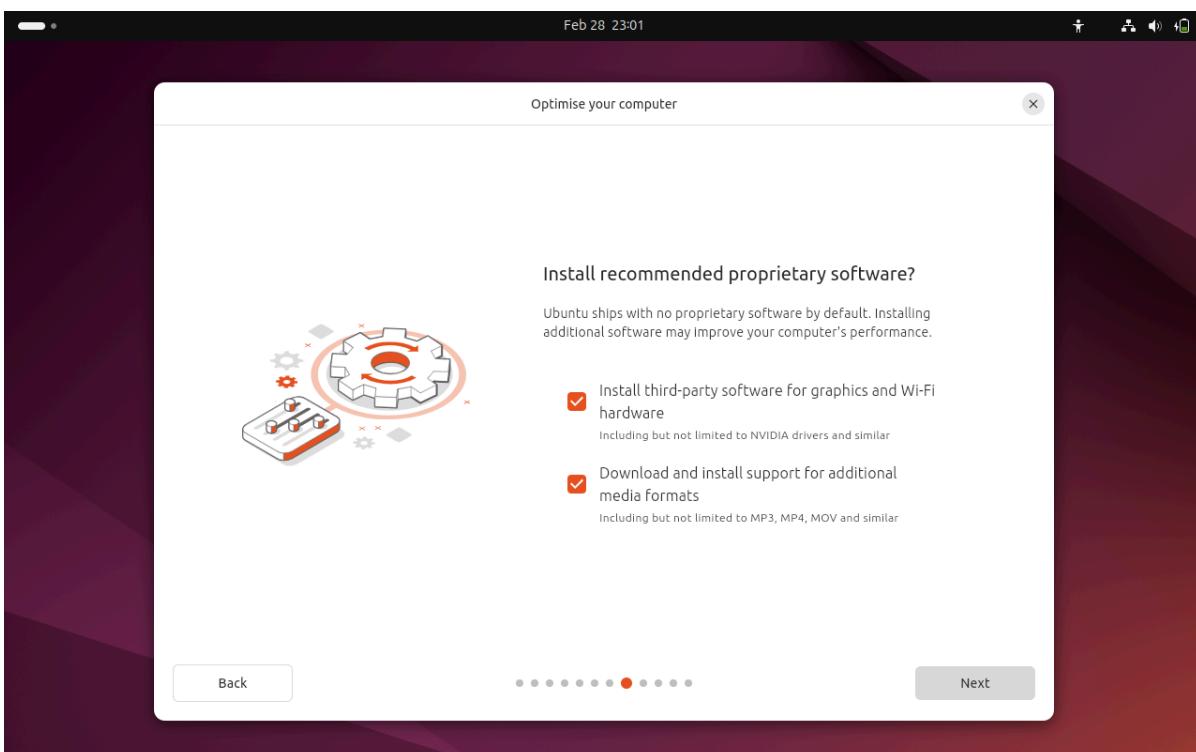
Aperte em "Next" ou "Próximo", deixe selecionada a forma padrão de instalação



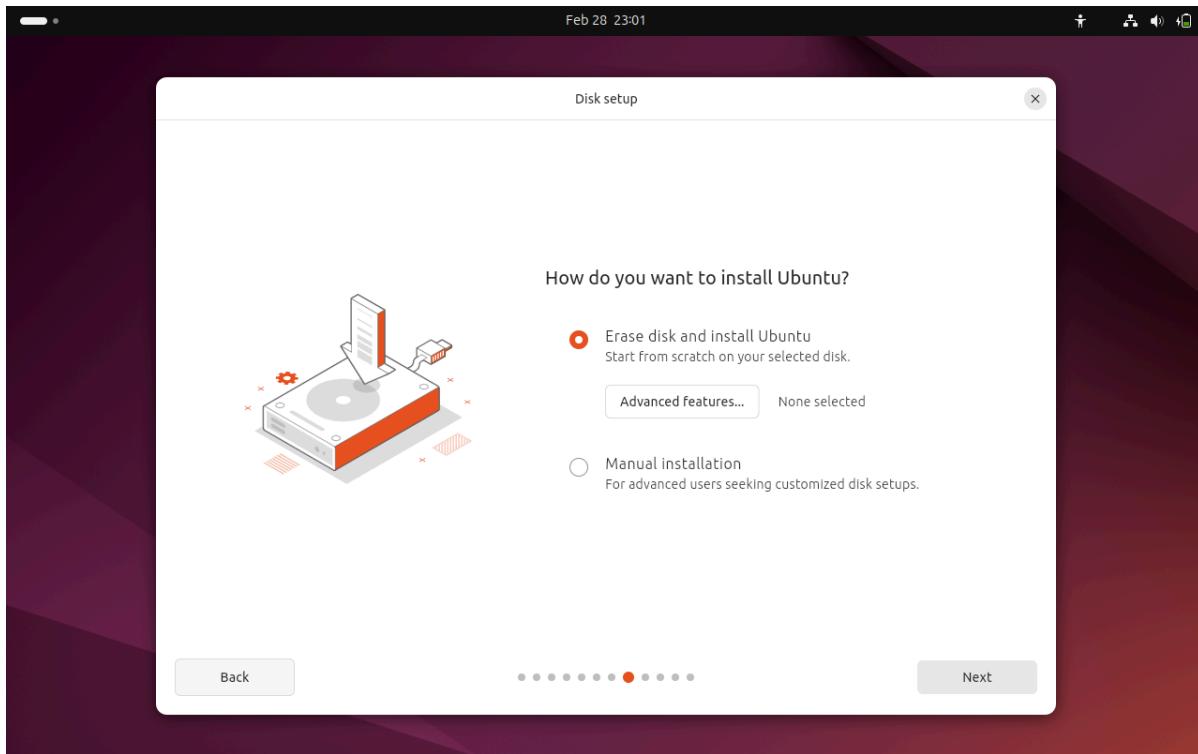
Deixe na forma interativa de instalação, ou seja, primeira opção



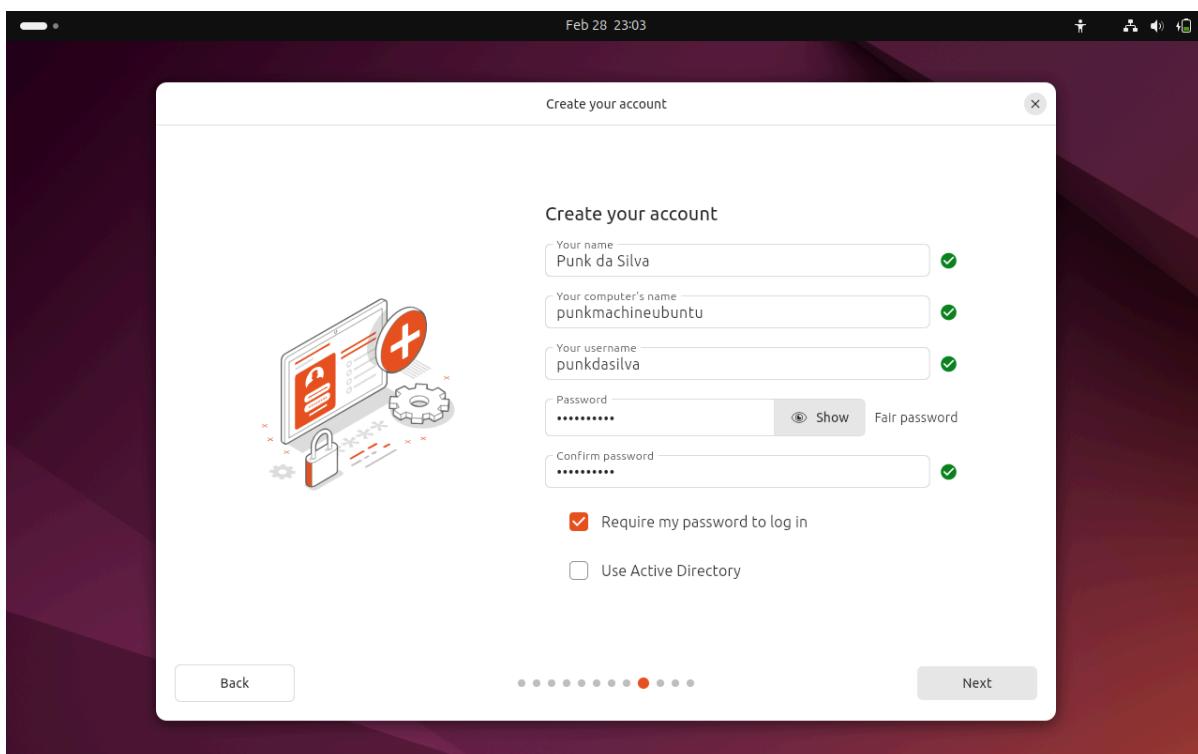
Para a próxima parte, é onde definimos se queremos que ao instalar o Ubuntu sejam instalados aplicativos adicionais, além dos básicos como navegador e outros utilitários. Neste caso, deixe na opção padrão



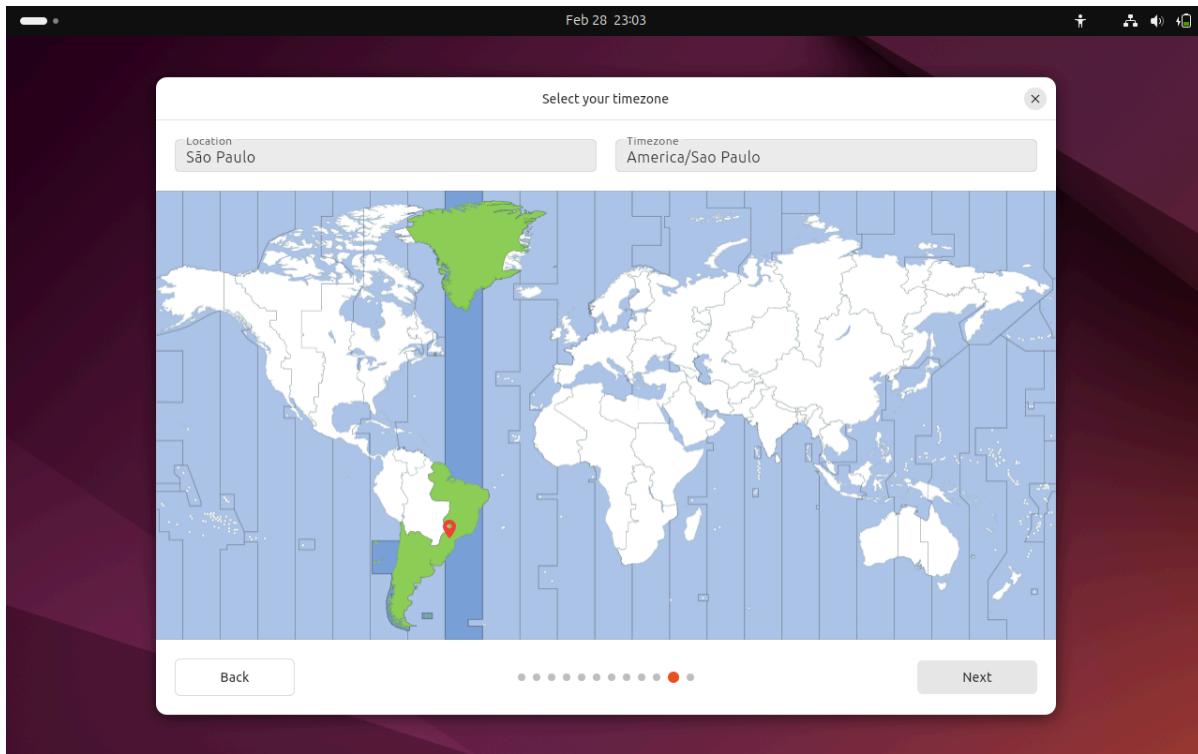
Nesta etapa, selecione todas as opções, que são para instalar softwares de terceiros e download de formatos de mídia adicionais



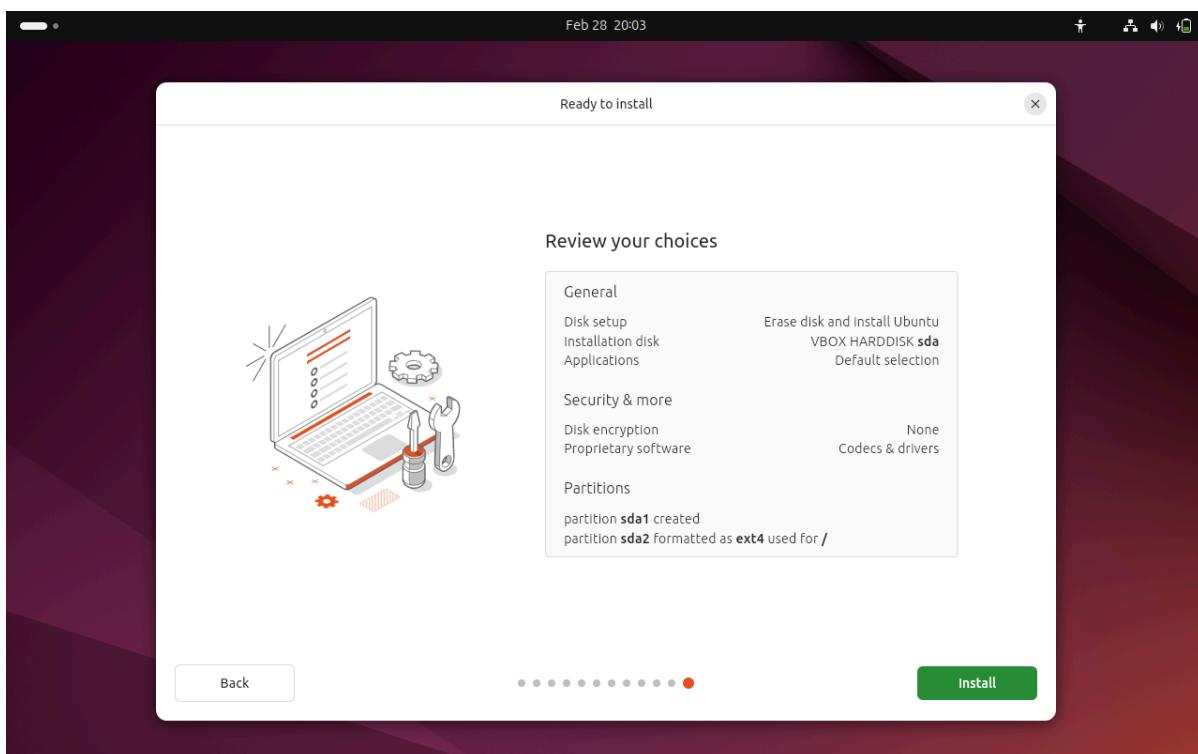
Nesta parte é a definição de se iremos instalar limpando o disco ou se faremos o particionamento do disco. Deixe a opção como padrão e aperte em next



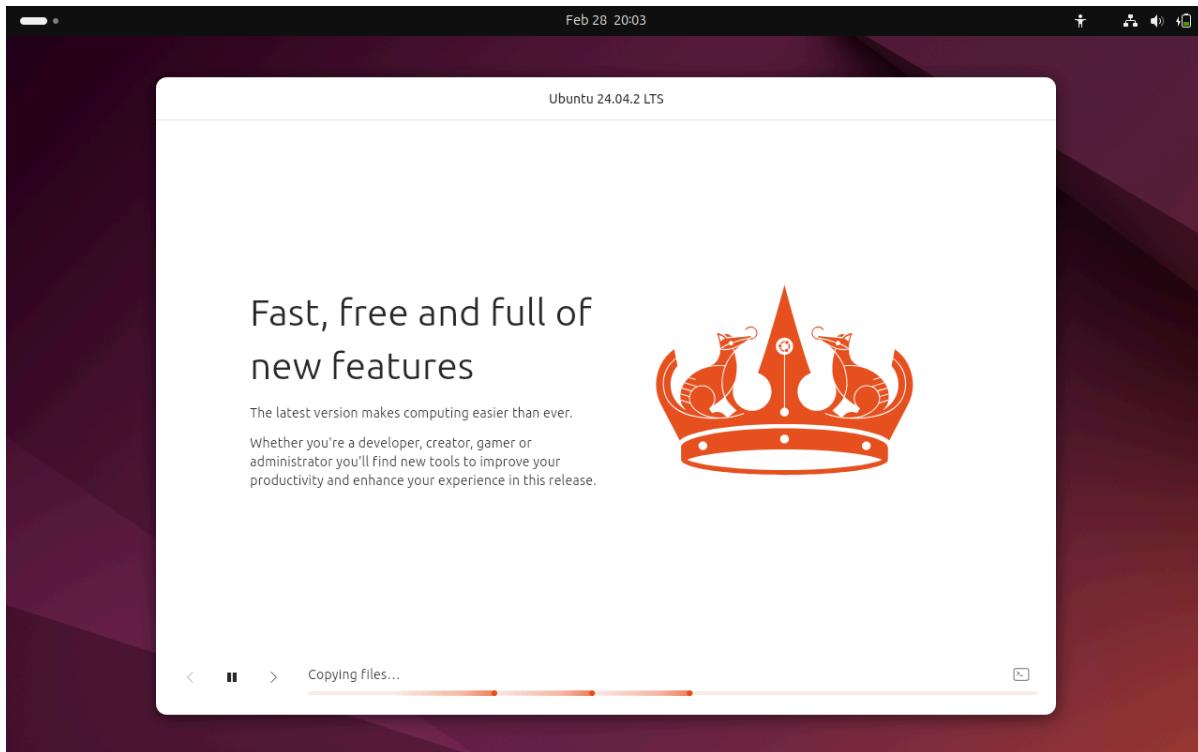
Agora na parte de criação de conta, defina suas credenciais



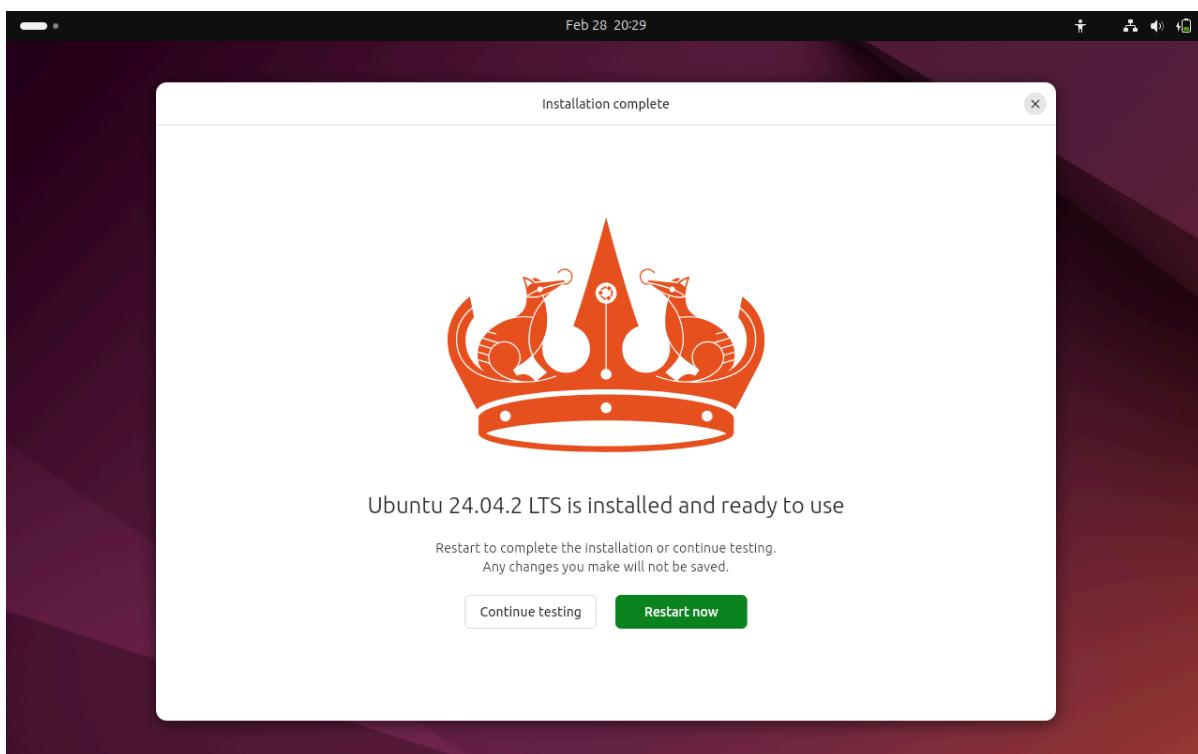
Agora é só fazermos a configuração do fuso horário, ou seja, do tempo que o computador irá usar



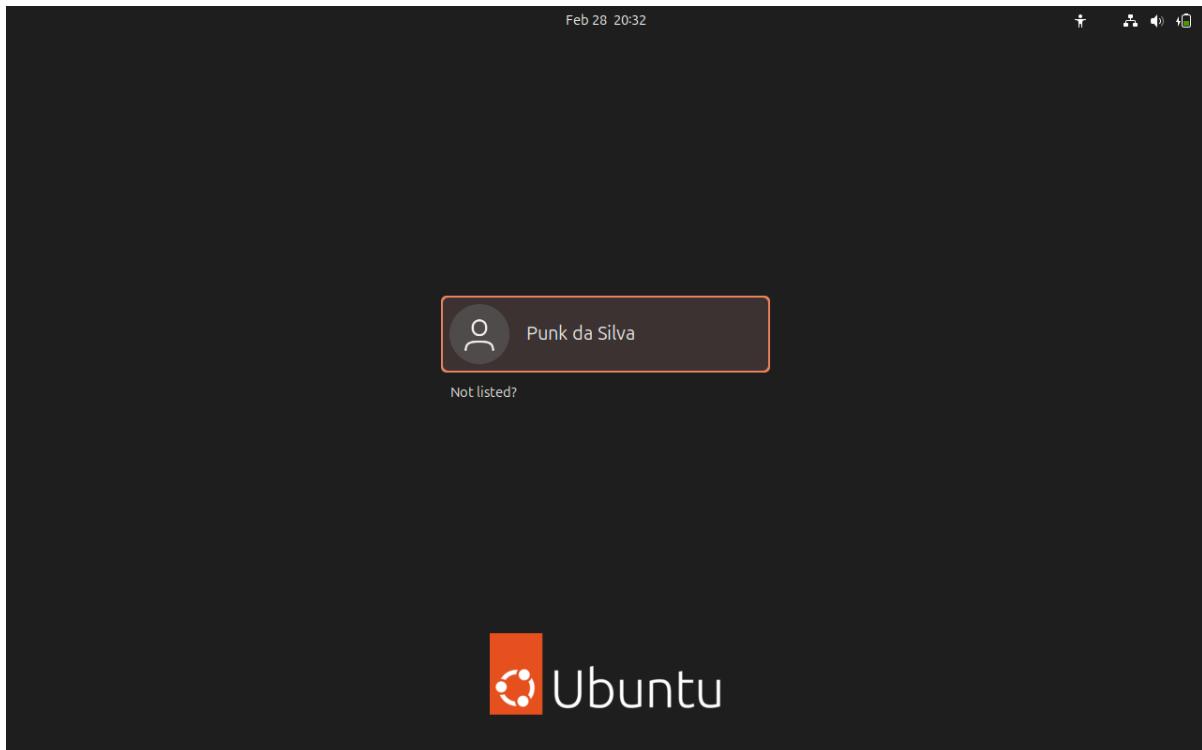
Nesta página será apenas para mostrar o resumo da instalação, uma visão geral das configurações para instalação



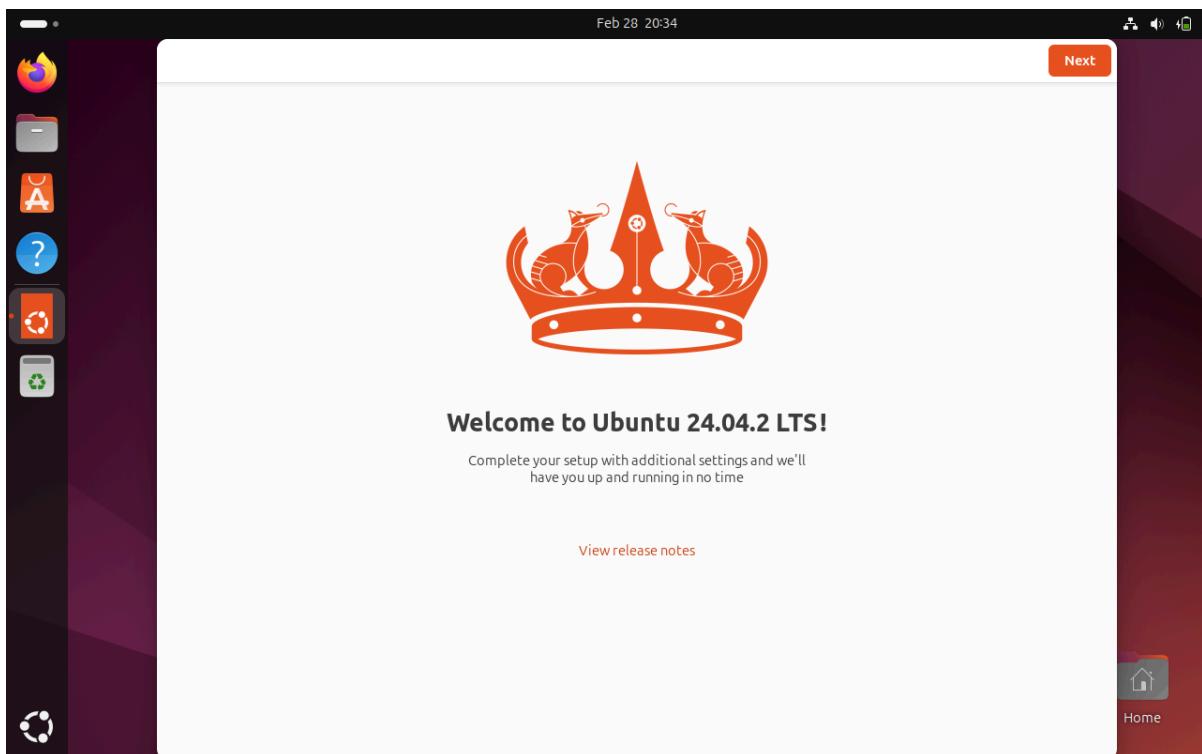
Aqui é a etapa em que o sistema será instalado, pode demorar uns 30 minutos



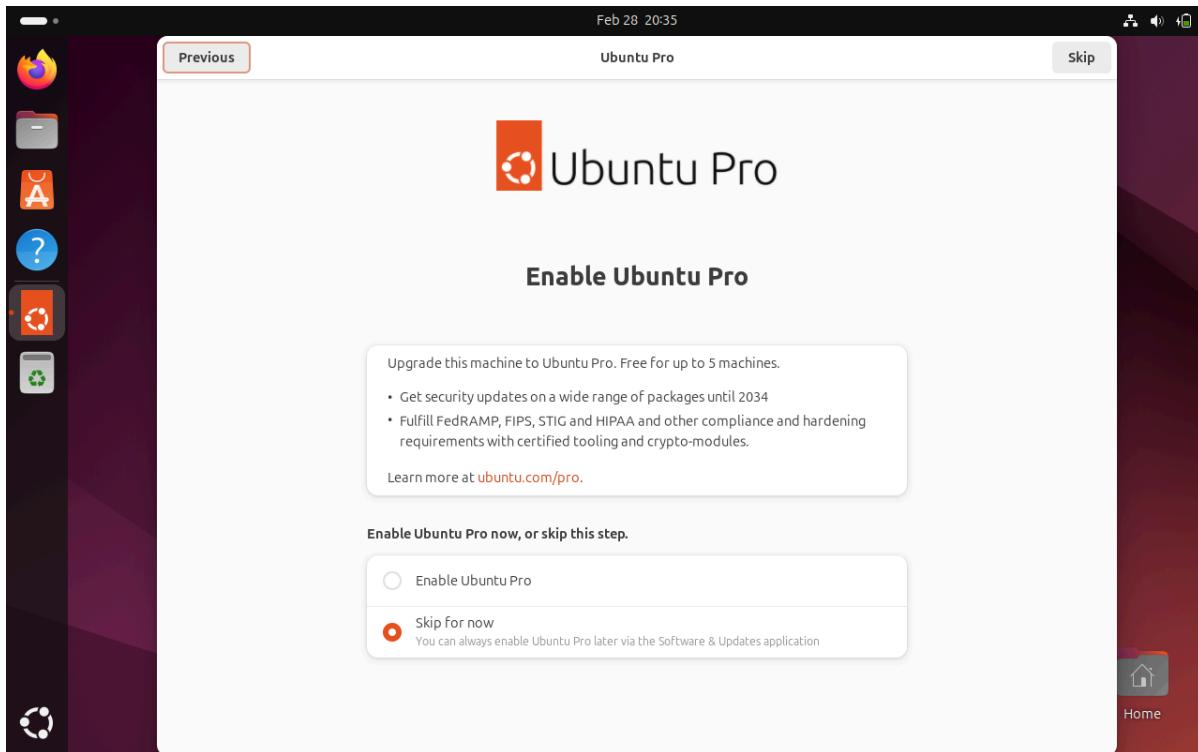
Com a etapa anterior concluída, o sistema já foi instalado e já podemos reiniciar a máquina



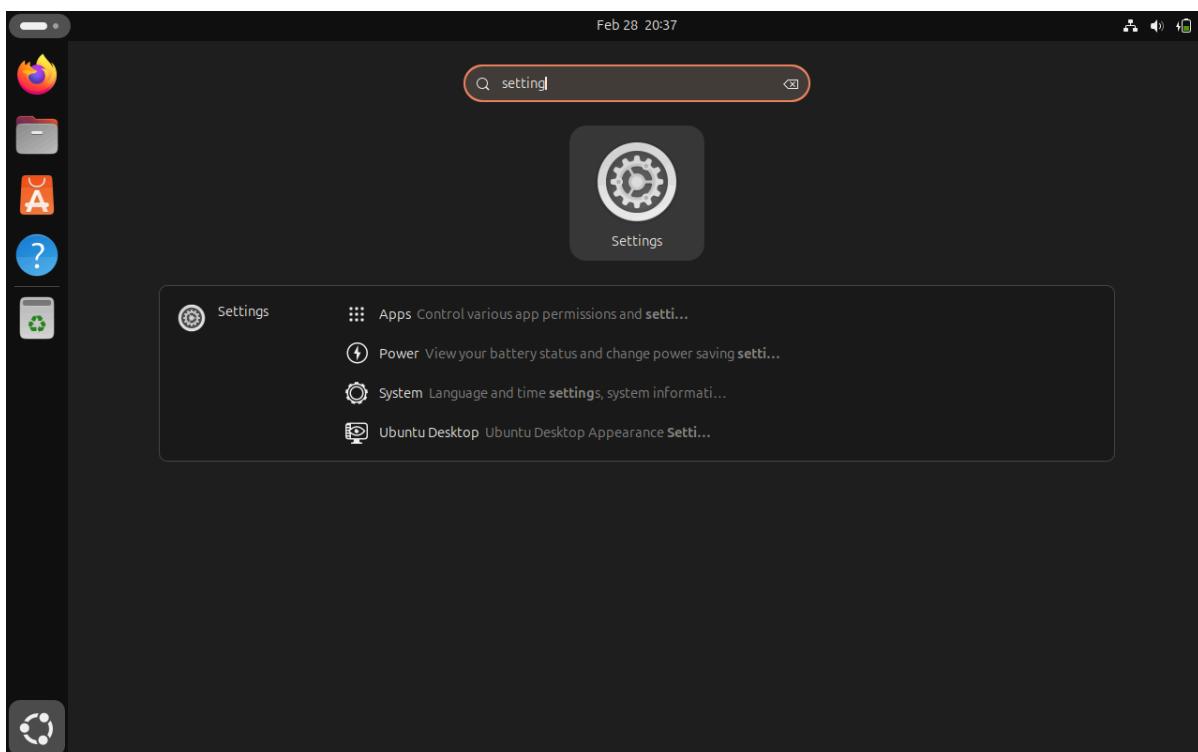
Ao reiniciarmos a máquina, aparece então a tela de login do usuário que foi criado



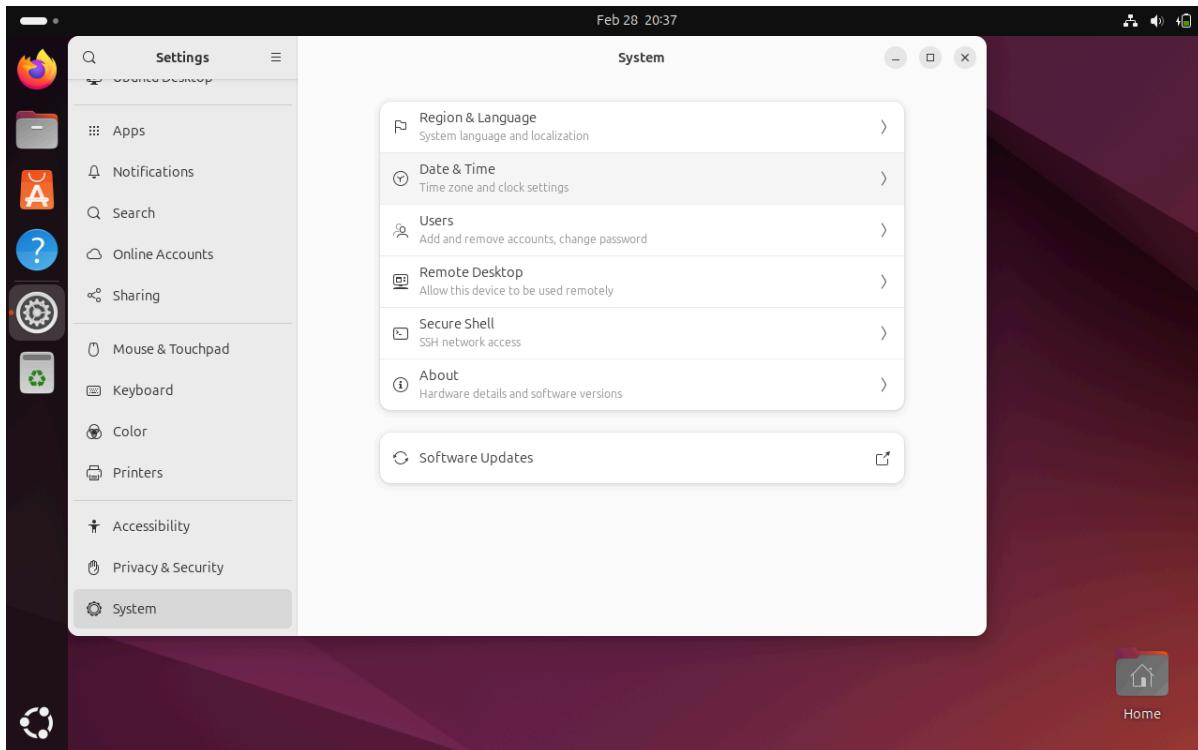
Ao logarmos, temos a visão desta tela



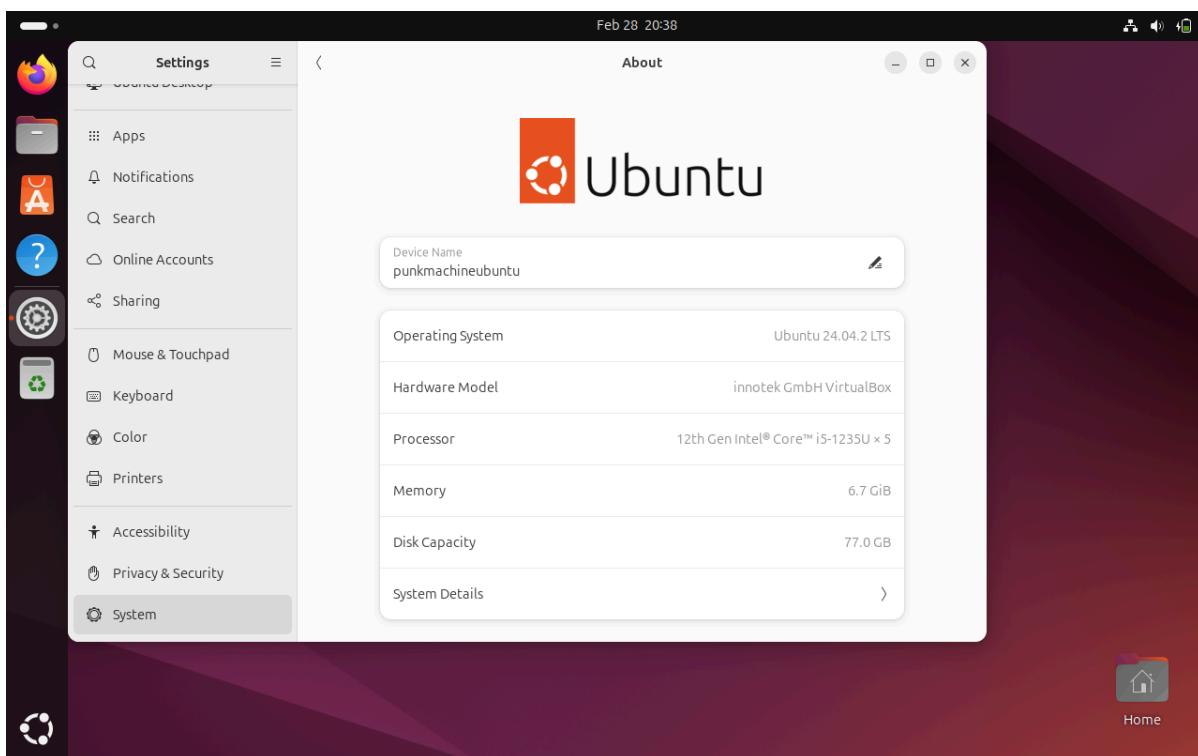
Nessa parte, basta passarmos adiante - clique em Skip ou Prosseguir



Aperte a tecla Super do seu teclado e digite "settings" ou "configurações" e aperte no primeiro item



Vá na parte inferior do menu lateral esquerdo e aperte em "System" ou "Sistema" e em seguida aperte em "About" ou "Sobre"



Assim podemos visualizar as informações do sistema que está instalado

Respostas - Questões 1

#	Resposta Correta
1	b
2	b
3	b
4	b
5	c
6	b
7	d
8	b
9	d
10	d
11	d
12	a
13	d
14	d
15	b
16	a
17	d

18	d
19	a
20	c
21	a
22	b
23	a
24	b
25	b

Threads

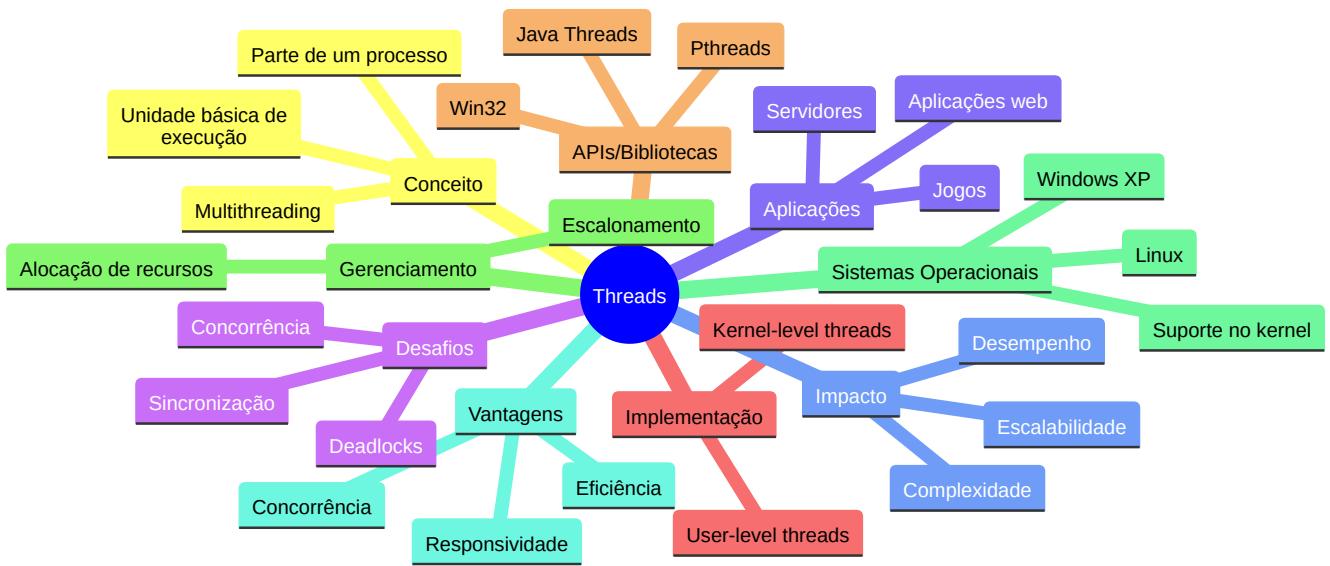
No contexto da computação moderna, o conceito de processos foi tradicionalmente associado à execução de um programa com uma única linha de execução, ou *thread*. No entanto, com o avanço das tecnologias e a necessidade de maior eficiência e desempenho, os sistemas operacionais evoluíram para suportar processos com múltiplas threads de controle. Este capítulo explora o conceito de *threads*, que são unidades fundamentais de execução dentro de um processo, permitindo que tarefas sejam realizadas de forma concorrente e paralela.

A introdução de threads trouxe uma nova dimensão ao design de sistemas operacionais e à programação de aplicações. Ao permitir que um processo contenha várias threads, os sistemas podem executar múltiplas tarefas simultaneamente, melhorando a utilização de recursos e a responsividade das aplicações. Este capítulo aborda os principais conceitos relacionados a sistemas *multithreaded*, incluindo as APIs mais comuns para manipulação de threads, como Pthreads, Win32 e as bibliotecas de threads em Java.

Além disso, serão examinadas as questões e desafios associados à programação multithread, como sincronização, concorrência e escalonamento, e como esses aspectos influenciam o design dos sistemas operacionais. Por fim, será explorado o suporte a threads no nível do kernel em sistemas operacionais modernos, como Windows XP e Linux, destacando como esses sistemas gerenciam e otimizam a execução de múltiplas threads.

Objetivos do Capítulo

- Introduzir o conceito de thread como uma unidade fundamental de execução.
- Explorar as APIs e bibliotecas para manipulação de threads em diferentes ambientes.
- Discutir os desafios e técnicas de programação multithread.
- Analisar o impacto das threads no design dos sistemas operacionais.
- Examinar o suporte a threads no nível do kernel em sistemas operacionais modernos.



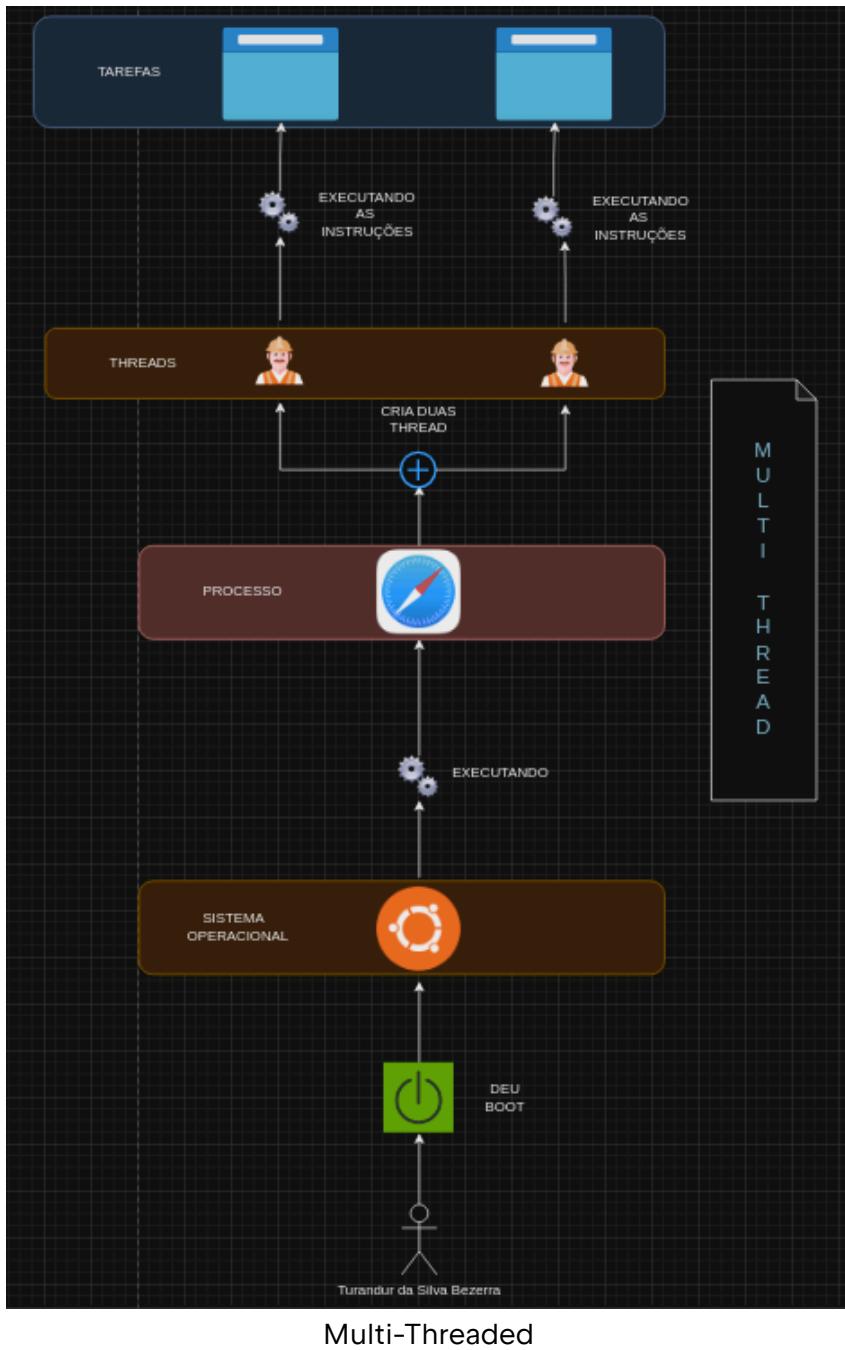
4.1. Usos

Contextualização: O que são Threads e Por Que São Importantes?

Em computação, um **processo** é um programa em execução, como um navegador Web, um jogo ou um servidor. Tradicionalmente, um processo tinha apenas uma **thread de controle**, ou seja, uma única sequência de execução de instruções. Isso significa que, em um processo single-threaded, todas as tarefas são executadas de forma sequencial, uma após a outra. Por exemplo, se você estivesse rodando um navegador Web single-threaded, ele não poderia carregar uma página enquanto responde aos cliques do mouse ou verifica a ortografia de um texto.



Single Thread



Multi-Threaded

No entanto, com o avanço da tecnologia e a necessidade de maior eficiência e desempenho, os sistemas operacionais modernos passaram a suportar **processos multithreaded**, ou seja, processos que contêm múltiplas threads de controle. Uma **thread** é uma unidade básica de execução dentro de um processo, capaz de realizar tarefas de forma independente. Isso permite que um processo execute várias operações simultaneamente, melhorando a utilização de recursos e a responsividade das aplicações.

Por Que Threads São Importantes?

- 1. Concorrência:** Threads permitem que várias tarefas sejam executadas ao mesmo tempo, como carregar uma página Web enquanto o usuário digita ou ouve música.
- 2. Eficiência:** Threads são mais leves que processos, pois compartilham recursos como memória e arquivos abertos. Isso reduz a sobrecarga do sistema.
- 3. Responsividade:** Aplicações multithreaded são mais ágeis, pois tarefas demoradas podem ser executadas em segundo plano sem travar a interface do usuário.
- 4. Escalabilidade:** Servidores e sistemas operacionais podem atender a milhares de requisições simultaneamente, criando uma thread para cada tarefa.

Agora que entendemos o que são threads e por que elas são importantes, vamos explorar exemplos práticos usando **Minecraft** como analogia para ilustrar como as threads são usadas em diferentes contextos.

1. Navegador Web (Minecraft como analogia)



Explicação Detalhada:

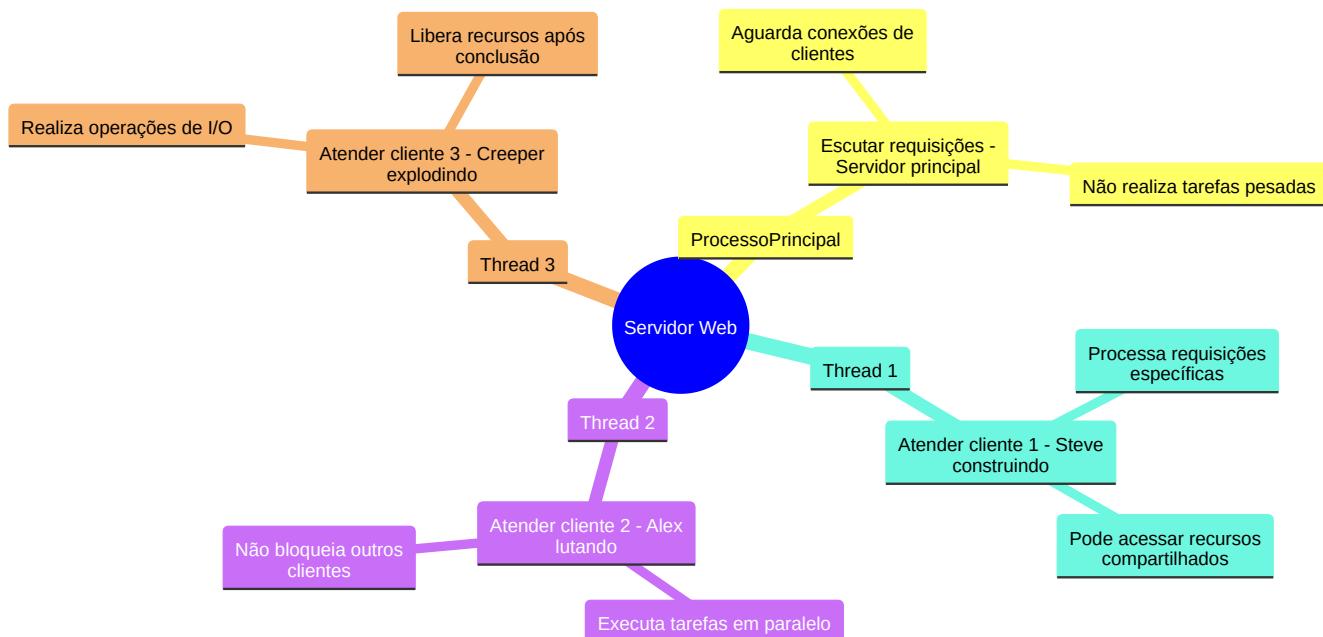
- Um navegador Web moderno é como um **Minecraft com múltiplos personagens**. Cada thread (personagem) tem uma função específica:
 - Thread 1 (Steve minerando):** Responsável por exibir imagens e texto na tela. Precisa ser rápido para garantir que a interface do usuário não trave.
 - Thread 2 (Alex explorando):** Busca dados da rede, como páginas, imagens e vídeos. Trabalha em segundo plano para que o usuário possa continuar interagindo com a interface.
 - Thread 3 (Verificação ortográfica):** Executa tarefas em background. Não interfere na experiência principal do usuário.

- **Thread 3 (Creeper esperando):** Realiza tarefas em background, como verificação ortográfica. Não interfere na experiência principal do usuário.

Benefícios:

- **Concorrência:** As threads permitem que o navegador execute várias tarefas ao mesmo tempo, como carregar uma página enquanto o usuário digita.
- **Responsividade:** A interface do usuário não trava, pois as tarefas demoradas são executadas em segundo plano.
- **Eficiência:** Recursos do sistema são utilizados de forma otimizada.

2. Servidor Web (Minecraft Servidor)



Explicação Detalhada:

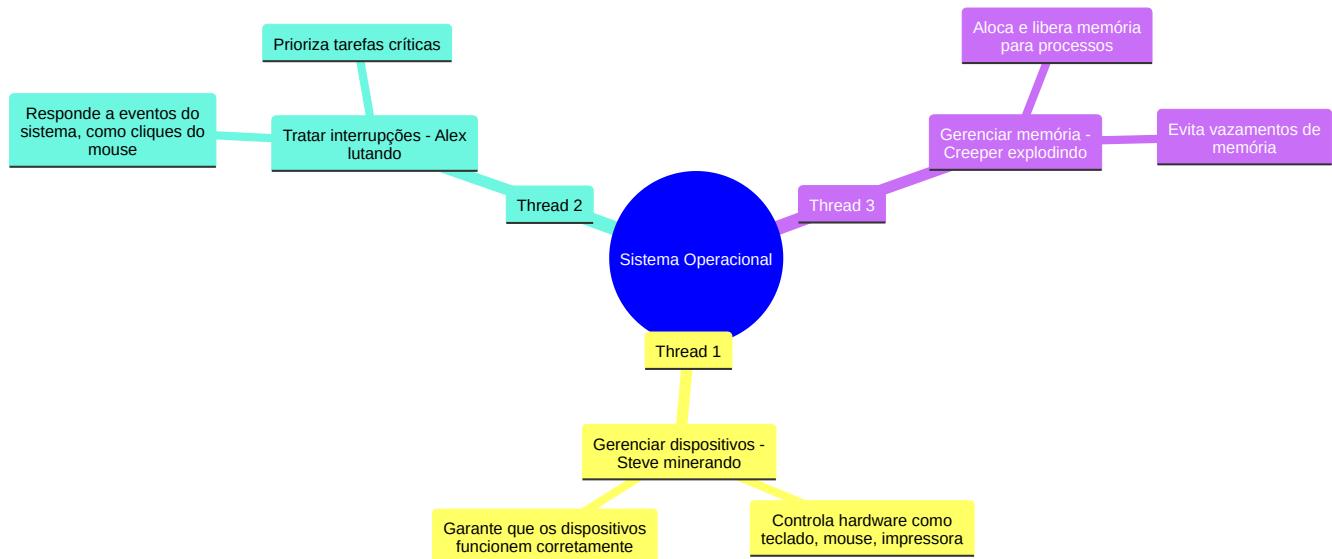
- Um servidor Web é como um **servidor de Minecraft** que precisa atender a vários jogadores (clientes) ao mesmo tempo:
 - **Processo Principal:** Escuta requisições de clientes, mas não realiza tarefas pesadas. É como o servidor principal que aguarda conexões.
 - **Thread 1 (Steve construindo):** Atende a um cliente específico, processando suas requisições. Pode acessar recursos compartilhados, como bancos de dados.
 - **Thread 2 (Alex lutando):** Atende outro cliente em paralelo, sem bloquear os demais.

- **Thread 3 (Creeper explodindo):** Realiza operações de I/O, como leitura/escrita de arquivos, e libera recursos após concluir a tarefa.

Benefícios:

- **Escalabilidade:** O servidor pode atender a milhares de clientes simultaneamente, criando uma thread para cada requisição.
- **Eficiência:** Threads são mais leves que processos, economizando recursos do sistema.
- **Concorrência:** Várias requisições são processadas ao mesmo tempo, sem que os clientes precisem esperar.

3. Sistema Operacional Multithread



Explicação Detalhada:

- O sistema operacional é como um **Minecraft com mods**, onde cada thread (personagem) tem uma função específica:
 - **Thread 1 (Steve minerando):** Gerencia dispositivos de hardware, como teclado, mouse e impressora. Garante que todos os dispositivos funcionem corretamente.
 - **Thread 2 (Alex lutando):** Trata interrupções do sistema, como cliques do mouse ou pressionamentos de tecla. Prioriza tarefas críticas para manter o sistema responsivo.
 - **Thread 3 (Creeper explodindo):** Gerencia a memória do sistema, aloçando e liberando memória para processos. Evita vazamentos de memória, que podem travar o sistema.

Benefícios:

- **Modularidade:** Cada thread é responsável por uma tarefa específica, facilitando a manutenção e o desenvolvimento do sistema operacional.
- **Eficiência:** Tarefas críticas, como o gerenciamento de memória, são executadas de forma independente, sem interferir no funcionamento geral do sistema.
- **Concorrência:** Várias tarefas do sistema são executadas simultaneamente, garantindo que o computador funcione de forma suave e responsiva.

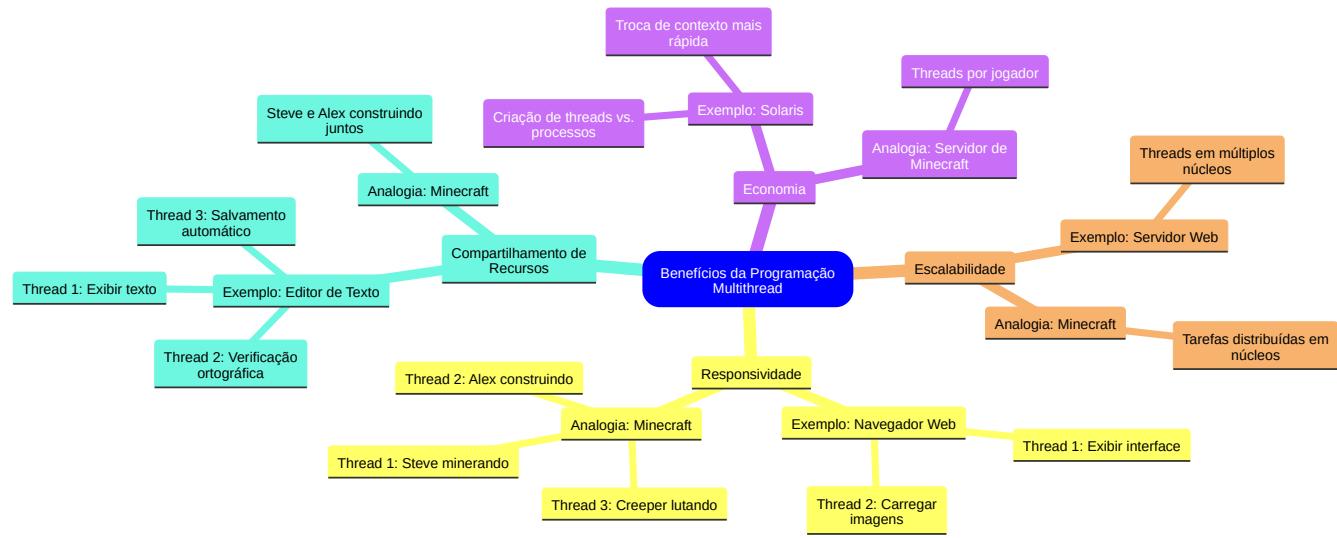
Conclusão Geral

Threads são como **personagens em Minecraft**: cada um pode realizar tarefas independentes, tornando o sistema mais eficiente, responsivo e escalável. Sem threads, seria como jogar Minecraft com apenas um personagem fazendo tudo de forma lenta e sequencial. Aqui estão os principais pontos:

1. **Concorrência:** Threads permitem que várias tarefas sejam executadas ao mesmo tempo, como minerar, construir e lutar em Minecraft.
2. **Eficiência:** Threads são mais leves que processos, economizando recursos do sistema.
3. **Responsividade:** A interface do usuário não trava, pois tarefas demoradas são executadas em segundo plano.
4. **Escalabilidade:** Sistemas multithread podem atender a milhares de requisições simultaneamente, como um servidor Web ou um servidor de Minecraft.

4.2 Benefícios da Programação Multithread

A programação multithread oferece vantagens significativas em relação ao uso de processos single-threaded. Esses benefícios podem ser categorizados em quatro áreas principais: **responsividade, compartilhamento de recursos, economia e escalabilidade**. Vamos explorar cada uma delas em detalhes, utilizando exemplos práticos e analogias para facilitar o entendimento.



4.2.1 Responsividade

A **responsividade** é um dos benefícios mais perceptíveis da programação multithread. Em aplicações interativas, como navegadores Web ou editores de texto, o uso de múltiplas threads permite que o programa continue funcionando de forma ágil, mesmo que parte dele esteja ocupada com operações demoradas.

Exemplo Prático: Navegador Web

Imagine um navegador Web que utiliza uma única thread para todas as tarefas. Se você estiver carregando uma página com muitas imagens, a interface do navegador pode travar até que todas as imagens sejam carregadas. Isso resultaria em uma experiência frustrante para o usuário.

Com o uso de múltiplas threads, o navegador pode:

- **Thread 1:** Exibir a interface e responder aos cliques do usuário.
- **Thread 2:** Carregar imagens e outros recursos em segundo plano.

Dessa forma, o usuário pode continuar interagindo com a interface enquanto as imagens são carregadas, aumentando a **responsividade** do sistema.

Analogia com Minecraft

Pense em um jogador de Minecraft que precisa minerar recursos, construir estruturas e lutar contra mobs ao mesmo tempo. Se ele tivesse que fazer tudo de forma sequencial, a experiência seria lenta e frustrante. Com múltiplas threads (ou "personagens"), ele pode:

- **Thread 1 (Steve):** Minerar recursos.
- **Thread 2 (Alex):** Construir uma casa.
- **Thread 3 (Creeper):** Lutar contra mobs.

Isso torna o jogo mais dinâmico e responsivo.

4.2.2 Compartilhamento de Recursos

As threads compartilham naturalmente a memória e os recursos do processo ao qual pertencem, o que facilita a comunicação e a coordenação entre elas. Em contraste, os processos precisam usar técnicas como **memória compartilhada** ou **troca de mensagens** para compartilhar recursos, o que exige mais esforço do programador.

Exemplo Prático: Aplicações Multithreaded

Em um editor de texto multithreaded, várias threads podem acessar o mesmo documento simultaneamente:

- **Thread 1:** Exibe o texto na tela.
- **Thread 2:** Realiza a verificação ortográfica.
- **Thread 3:** Salva o documento automaticamente.

Como as threads compartilham o mesmo espaço de memória, elas podem acessar e modificar o documento sem a necessidade de mecanismos complexos de comunicação.

Analogia com Minecraft

Imagine que Steve e Alex estão construindo uma casa juntos. Como eles compartilham o mesmo mundo (espaço de memória), podem trabalhar em diferentes partes da construção sem precisar se comunicar constantemente. Isso torna o processo mais eficiente.

4.2.3 Economia

Criar e gerenciar processos é uma operação custosa em termos de recursos do sistema. Cada processo requer sua própria alocação de memória, espaço de endereçamento e recursos do

sistema operacional. Já as threads, por compartilharem os recursos do processo ao qual pertencem, são muito mais leves e econômicas.

Exemplo Prático: Criação de Threads vs. Processos

No sistema operacional **Solaris**, por exemplo:

- A criação de um processo é cerca de **30 vezes mais lenta** do que a criação de uma thread.
- A troca de contexto entre processos é cerca de **5 vezes mais lenta** do que a troca de contexto entre threads.

Isso significa que, em aplicações que exigem a criação frequente de tarefas (como servidores Web), o uso de threads é muito mais eficiente.

Analogia com Minecraft

Pense em um servidor de Minecraft que precisa atender a vários jogadores. Se cada jogador exigisse a criação de um novo processo, o servidor ficaria sobrecarregado rapidamente. Em vez disso, o servidor cria uma thread para cada jogador, compartilhando recursos como memória e arquivos, o que é muito mais econômico.

4.2.4 Escalabilidade

A **escalabilidade** é um benefício crucial em sistemas multithreaded, especialmente em arquiteturas multiprocessadas (com múltiplos núcleos de CPU). Enquanto um processo single-threaded só pode ser executado em um único processador, um processo multithreaded pode distribuir suas threads entre vários processadores, aumentando o paralelismo e o desempenho.

Exemplo Prático: Aplicações em Máquinas Multiprocessadas

Em um servidor Web multithreaded rodando em uma máquina com 8 núcleos de CPU:

- Cada thread pode ser executada em um núcleo diferente.
- Isso permite que o servidor atenda a múltiplas requisições simultaneamente, aumentando a capacidade de processamento.

Imagine que você está jogando Minecraft em um computador com 8 núcleos de CPU. Com múltiplas threads, o jogo pode distribuir tarefas como renderização, física e IA de mobs entre os núcleos, resultando em um desempenho muito melhor do que se tudo fosse executado em um único núcleo.

4.2.5 Resumo dos Benefícios

Benefício	Descrição	Exemplo Prático	Analogia com Minecraft
Responsividade	Permite que aplicações continuem funcionando durante operações demoradas.	Navegador Web carregando imagens em segundo plano.	Steve minerando enquanto Alex constrói.
Compartilhamento de Recursos	Threads compartilham memória e recursos, facilitando a comunicação.	Editor de texto com verificação ortográfica.	Steve e Alex construindo a mesma casa.
Economia	Threads são mais leves e rápidas de criar e gerenciar do que processos.	Servidor Web atendendo múltiplos clientes.	Servidor de Minecraft com threads por jogador.
Escalabilidade	Aumenta o paralelismo em sistemas multiprocessados.	Servidor Web rodando em múltiplos núcleos.	Minecraft usando todos os núcleos da CPU.

4.2.6 Conclusão

A programação multithread traz benefícios significativos para o desenvolvimento de aplicações modernas, desde a melhoria da **responsividade** até a **escalabilidade** em sistemas multiprocessados. Ao permitir que tarefas sejam executadas de forma concorrente e paralela, as threads tornam os sistemas mais eficientes, econômicos e capazes de lidar com demandas crescentes. Usar threads é como adicionar **mods ao Minecraft**: cada um traz novas funcionalidades e melhora a experiência geral.

4.3 Programação multicore

Imagine que você está jogando Minecraft em um computador com **um único núcleo** (single-core) e outro com **múltiplos núcleos** (multicore). Vamos usar o jogo para entender como a programação multithreaded funciona em cada cenário.

4.3.1 Tipos de Sistemas

Sistema de Único Núcleo (Single-Core)

Em um computador com apenas um núcleo, todas as tarefas do Minecraft precisam ser executadas de forma **concorrente**, ou seja, uma de cada vez, intercaladas no tempo. Por exemplo:

- **Thread 1:** Renderizar o mundo (gráficos).
- **Thread 2:** Calcular a física (queda de blocos, água, etc.).
- **Thread 3:** Executar a inteligência artificial dos mobs (zumbis, creepers, etc.).

Como há apenas um núcleo, o sistema operacional precisa alternar rapidamente entre essas threads, dando a impressão de que tudo está acontecendo ao mesmo tempo. No entanto, isso pode causar lentidão, especialmente se uma das tarefas for muito pesada.

Sistema de Múltiplos Núcleos (Multicore)

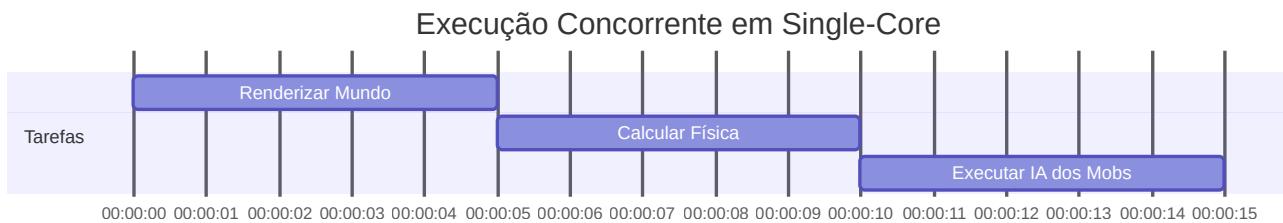
Em um computador com múltiplos núcleos, as threads podem ser executadas em **paralelo**, ou seja, cada núcleo pode processar uma thread simultaneamente. Por exemplo:

- **Núcleo 1:** Renderizar o mundo.
- **Núcleo 2:** Calcular a física.
- **Núcleo 3:** Executar a IA dos mobs.

Isso permite que o jogo funcione de forma muito mais rápida e eficiente, pois as tarefas são distribuídas entre os núcleos, sem precisar alternar entre elas.

4.3.2 Visualizando

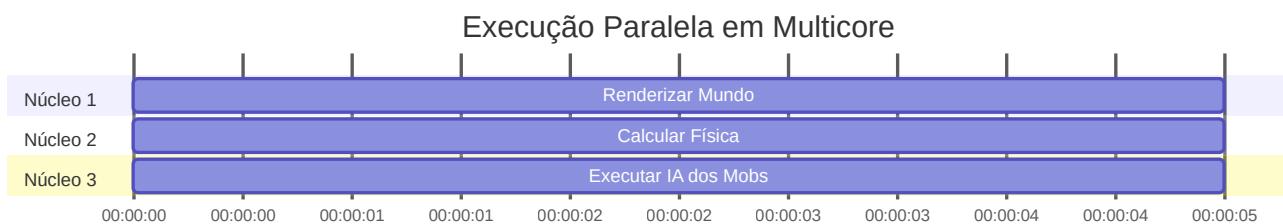
1. Execução Concorrente em Single-Core



Explicação:

- Em um sistema single-core, as tarefas são executadas uma de cada vez, intercaladas no tempo.
- O núcleo alterna entre renderizar o mundo, calcular a física e executar a IA dos mobs.

2. Execução Paralela em Multicore



Explicação:

- Em um sistema multicore, cada núcleo pode executar uma tarefa simultaneamente.
- O Núcleo 1 renderiza o mundo, o Núcleo 2 calcula a física e o Núcleo 3 executa a IA dos mobs ao mesmo tempo.

Desafios da Programação Multicore

1. Divisão de Atividades:

- No Minecraft, você precisa dividir as tarefas do jogo (renderização, física, IA) em threads separadas para aproveitar os múltiplos núcleos.
- Exemplo: Se você não separar a renderização da física, o jogo pode ficar lento.

2. Equilíbrio:

- As tarefas devem ter um valor igual. Por exemplo, se a renderização for muito mais pesada que a física, um núcleo pode ficar sobrecarregado enquanto outros ficam ociosos.

3. Separação de Dados:

- Os dados do jogo (como a posição dos blocos e mobs) precisam ser divididos entre os núcleos. Se dois núcleos tentarem modificar o mesmo bloco ao mesmo tempo, pode ocorrer um conflito.

4. Dependência de Dados:

- Se a física depende da posição dos mobs (por exemplo, um creeper explodindo um bloco), você precisa garantir que a thread da física espere a thread da IA terminar de calcular a posição.

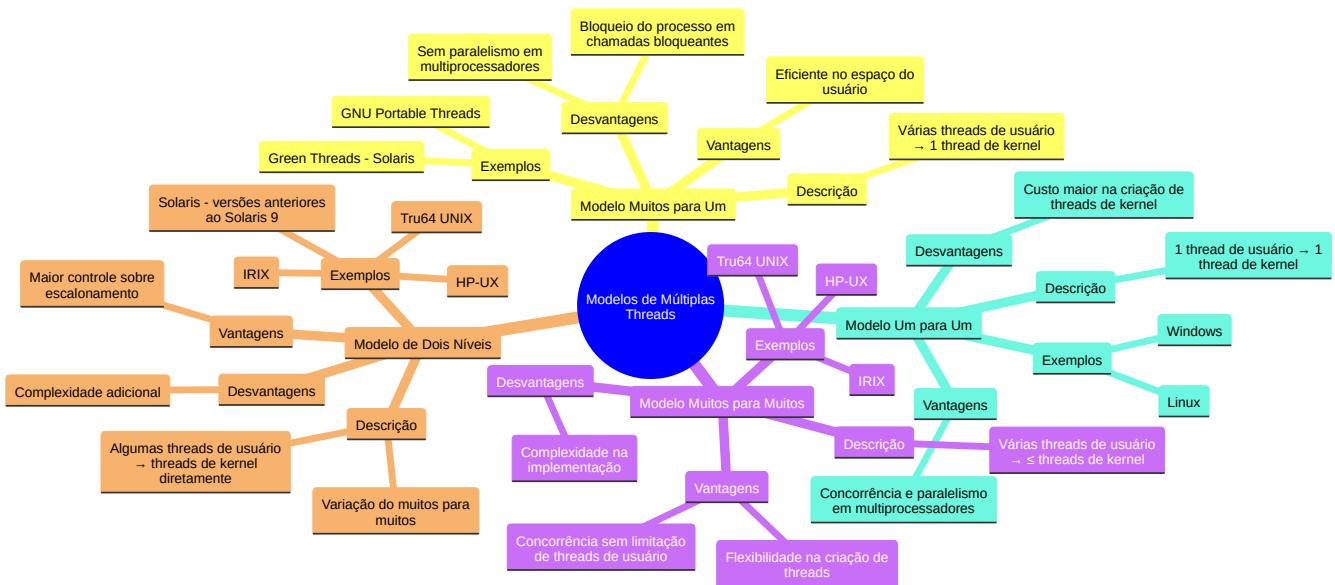
5. Teste e Depuração:

- Em um jogo multithreaded, bugs podem ser difíceis de reproduzir, pois dependem da ordem de execução das threads. Por exemplo, um creeper pode explodir antes de ser renderizado, causando um bug visual.

4.3.3 Resumo dos Desafios

Desafio	Descrição	Exemplo no Minecraft
Divisão de Atividades	Dividir o jogo em tarefas concorrentes.	Separar renderização, física e IA em threads distintas.
Equilíbrio	Garantir que as tarefas tenham valor igual.	Evitar que a renderização sobrecarregue um núcleo enquanto outros ficam ociosos.
Separação de Dados	Dividir os dados do jogo entre os núcleos.	Garantir que cada núcleo acesse blocos e mobs diferentes.
Dependência de Dados	Sincronizar tarefas que dependem de dados compartilhados.	Garantir que a física espere a IA terminar de calcular a posição dos mobs.
Teste e Depuração	Testar e depurar programas com múltiplos caminhos de execução.	Reproduzir bugs que ocorrem apenas quando um creeper explode durante a renderização.

4.4 Modelos de múltiplas threads (multithreading)



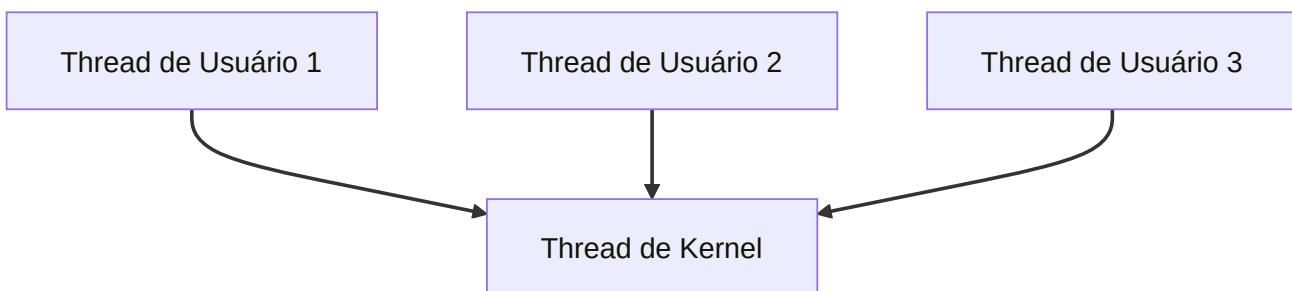
4.4.1 Modelos de Múltiplas Threads

Os sistemas operacionais modernos suportam threads de duas formas: **threads de usuário** (gerenciadas no espaço do usuário) e **threads de kernel** (gerenciadas diretamente pelo sistema operacional). A relação entre essas threads pode ser estabelecida de três maneiras principais: **muitos para um**, **um para um** e **muitos para muitos**.

1. Modelo Muitos para Um

No modelo **muitos para um**, várias threads de usuário são mapeadas para uma única thread de kernel. O gerenciamento das threads é feito por uma biblioteca no espaço do usuário, o que torna o processo eficiente. No entanto, se uma thread fizer uma chamada de sistema bloqueante, todo o processo será bloqueado. Além disso, como apenas uma thread pode acessar o kernel por vez, não é possível executar threads em paralelo em sistemas multiprocessadores.

Diagrama Mermaid:



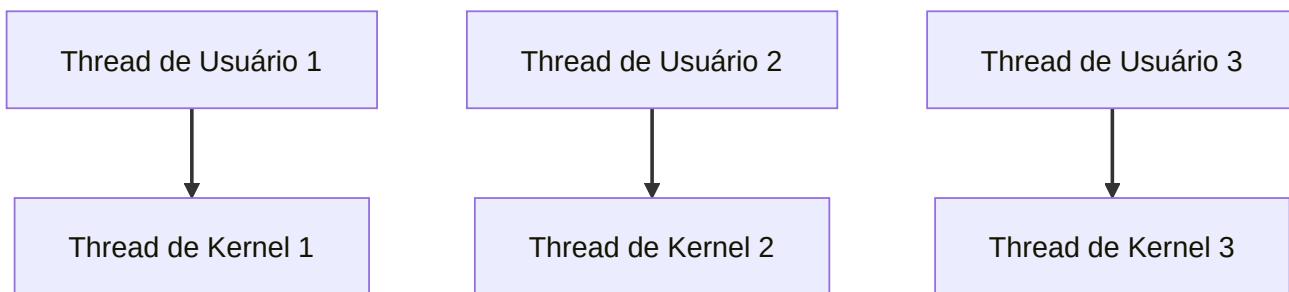
Exemplo:

- **Green Threads** (biblioteca do Solaris) e **GNU Portable Threads** usam esse modelo.
- **Vantagem:** Eficiência no gerenciamento de threads no espaço do usuário.
- **Desvantagem:** Bloqueio do processo inteiro em chamadas bloqueantes e falta de paralelismo em multiprocessadores.

2. Modelo Um para Um

No modelo **um para um**, cada thread de usuário é mapeada para uma thread de kernel. Isso permite maior concorrência, pois o kernel pode escalar threads independentemente. Se uma thread fizer uma chamada bloqueante, outras threads podem continuar executando. Além disso, threads podem ser executadas em paralelo em sistemas multiprocessadores. A principal desvantagem é que a criação de threads de kernel é mais custosa, o que pode limitar o número de threads que uma aplicação pode criar.

Diagrama Mermaid:



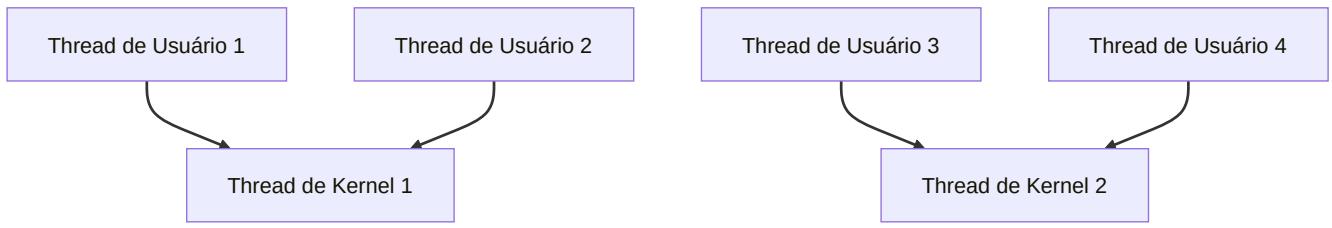
Exemplo:

- Sistemas operacionais como **Linux** e **Windows** usam esse modelo.
- **Vantagem:** Maior concorrência e paralelismo em multiprocessadores.
- **Desvantagem:** Custo maior na criação de threads de kernel.

3. Modelo Muitos para Muitos

No modelo **muitos para muitos**, várias threads de usuário são mapeadas para um número menor ou igual de threads de kernel. Isso permite que os desenvolvedores criem quantas threads de usuário forem necessárias, enquanto o kernel gerencia um número menor de threads de kernel. Esse modelo combina as vantagens dos modelos anteriores: concorrência, paralelismo e eficiência no gerenciamento de threads.

Diagrama Mermaid:



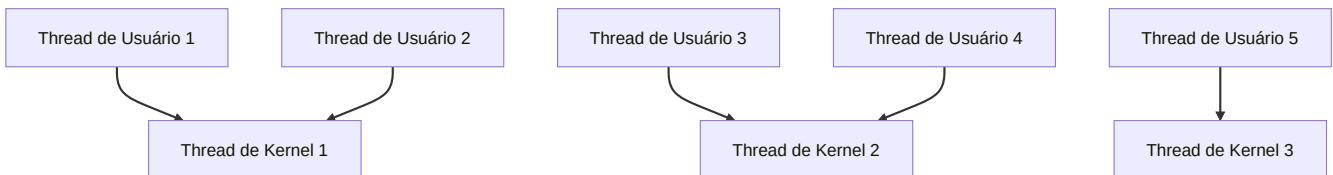
Exemplo:

- Sistemas como **IRIX**, **HP-UX** e **Tru64 UNIX** usam esse modelo.
- **Vantagem:** Flexibilidade para criar muitas threads de usuário e executar threads de kernel em paralelo.
- **Desvantagem:** Complexidade na implementação.

4. Modelo de Dois Níveis (Variação do Muitos para Muitos)

O modelo de **dois níveis** é uma variação do modelo muitos para muitos, onde algumas threads de usuário são mapeadas diretamente para threads de kernel, enquanto outras são multiplexadas. Isso oferece maior controle sobre o escalonamento de threads.

Diagrama Mermaid:



Exemplo:

- Sistemas como **IRIX**, **HP-UX** e **Tru64 UNIX** usam esse modelo.
- **Vantagem:** Combina a flexibilidade do modelo muitos para muitos com a eficiência do modelo um para um.
- **Desvantagem:** Complexidade adicional na implementação.

4.4.2 Comparação dos Modelos

Modelo	Descrição	Vantagens	Desvantagens
Muitos para Um	Várias threads de usuário mapeadas para uma thread de kernel.	Eficiente no espaço do usuário.	Bloqueio do processo em chamadas bloqueantes; sem paralelismo em multiprocessadores.
Um para Um	Cada thread de usuário mapeada para uma thread de kernel.	Concorrência e paralelismo em multiprocessadores.	Custo maior na criação de threads de kernel.
Muitos para Muitos	Várias threads de usuário mapeadas para um número menor de threads de kernel.	Flexibilidade e concorrência sem limitação no número de threads de usuário.	Complexidade na implementação.
Dois Níveis	Combina muitos para muitos com mapeamento direto de algumas threads.	Maior controle sobre o escalonamento de threads.	Complexidade adicional.

4.4.3 Conclusão

Os modelos de múltiplas threads (**muitos para um**, **um para um**, **muitos para muitos** e **dois níveis**) oferecem diferentes abordagens para gerenciar a concorrência e o paralelismo em sistemas operacionais. Cada modelo tem suas vantagens e desvantagens, e a escolha do modelo adequado depende das necessidades da aplicação e do ambiente de execução. Enquanto o modelo **um para um** é amplamente utilizado em sistemas modernos como Linux e Windows, o modelo **muitos para muitos** e sua variação **dois níveis** oferecem flexibilidade para aplicações que exigem um grande número de threads.

4.5 Bibliotecas de threads

Imagine que você está construindo uma cidade gigante no Minecraft. Para acelerar o processo, você decide chamar amigos (threads) para ajudar. Cada amigo pode trabalhar em uma tarefa específica, como construir casas, minerar recursos ou plantar árvores. Aqui está como as bibliotecas de threads se encaixam nessa analogia:

1. Threads no Espaço do Usuário

Como Funciona

- As threads são gerenciadas **inteiramente pela aplicação**, sem intervenção direta do sistema operacional (SO).
- A biblioteca de threads (como Pthreads em modo usuário) é responsável por criar, escalar e gerenciar as threads.
- Quando uma thread é criada, a biblioteca aloca uma estrutura de dados no espaço de memória do processo para armazenar informações sobre a thread (como estado, pilha, etc.).
- O **escalonamento** (decidir qual thread roda a seguir) é feito pela biblioteca, não pelo SO.

Vantagens

1. Menos overhead:

- Como não há chamadas ao kernel, a criação e troca de threads são mais rápidas.
- A troca de contexto entre threads é feita no espaço do usuário, sem a necessidade de mudar para o modo kernel.

2. Portabilidade:

- A aplicação pode ser portada para diferentes sistemas operacionais sem alterações significativas, desde que a biblioteca de threads seja suportada.

3. Controle total:

- O programador tem controle completo sobre o comportamento das threads, como políticas de escalonamento personalizadas.

Desvantagens

1. Falta de isolamento:

- Se uma thread falhar (por exemplo, causar um acesso inválido à memória), todo o processo pode ser afetado, já que todas as threads compartilham o mesmo espaço de memória.

2. Escalonamento limitado:

- O SO não está ciente das threads, então ele escalona o processo como um todo. Se uma thread faz uma operação bloqueante (como I/O), todo o processo é bloqueado, mesmo que outras threads estejam prontas para executar.

3. Menos suporte a multiprocessamento:

- Como o SO não conhece as threads, ele não pode distribuir as threads entre múltiplos núcleos de CPU de forma eficiente.

Exemplo Prático

Imagine que você está jogando Minecraft em um servidor privado com seus amigos. Vocês decidem quem faz o quê e como, sem precisar pedir permissão ao administrador do servidor. Isso é rápido e eficiente, mas se alguém cometer um erro (como derrubar um bloco errado), pode afetar todo o grupo.

2. Threads no Nível do Kernel

Como Funciona

- As threads são gerenciadas diretamente pelo sistema operacional.
- Quando uma thread é criada, o kernel aloca uma estrutura de dados no espaço do kernel para armazenar informações sobre a thread.
- O **escalonamento** é feito pelo SO, que decide qual thread deve ser executada em qual núcleo de CPU.
- Cada chamada à biblioteca de threads (como `pthread_create` ou `CreateThread`) resulta em uma **chamada de sistema** ao kernel.

Vantagens

1. Isolamento e segurança:

- O kernel garante que uma thread não interfira no funcionamento de outras threads ou do sistema como um todo.

- Se uma thread falhar, o SO pode encerrá-la sem afetar o restante do processo.

2. Escalonamento eficiente:

- O SO pode distribuir as threads entre múltiplos núcleos de CPU, aproveitando ao máximo o hardware disponível.
- Se uma thread é bloqueada (por exemplo, esperando I/O), o SO pode escalarar outra thread para executar.

3. Suporte a operações bloqueantes:

- Como o SO conhece as threads, ele pode gerenciar operações bloqueantes de forma eficiente, sem parar todo o processo.

Desvantagens

1. Overhead maior:

- Cada operação relacionada a threads (criação, troca de contexto, etc.) envolve uma chamada de sistema ao kernel, o que é mais lento do que operações no espaço do usuário.

2. Menos portabilidade:

- As APIs de threads no nível do kernel (como Win32) são específicas para cada sistema operacional, o que pode dificultar a portabilidade do código.

3. Complexidade:

- O programador tem menos controle sobre o comportamento das threads, pois o SO gerencia tudo.

Exemplo Prático

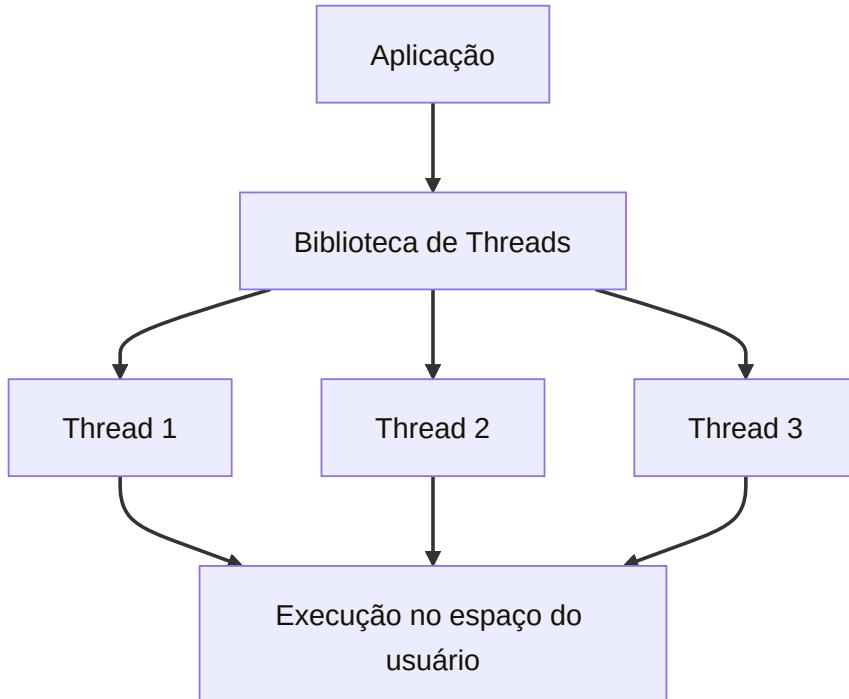
Agora, imagine que vocês estão jogando Minecraft em um servidor público. Tudo o que vocês fazem precisa ser aprovado pelo administrador do servidor. Isso é mais seguro e organizado, mas pode ser um pouco mais lento, pois vocês precisam esperar a aprovação do admin para cada ação.

Comparação Detalhada

Característica	Threads no Espaço do Usuário	Threads no Nível do Kernel
Gerenciamento	Pela aplicação (biblioteca de threads)	Pelo sistema operacional
Chamadas de sistema	Não usa	Usa (chamadas ao kernel)
Velocidade	Mais rápido	Mais lento (devido ao overhead)
Isolamento	Menos seguro (threads compartilham memória)	Mais seguro (isolamento pelo SO)
Escalonamento	Limitado (feito pela aplicação)	Eficiente (feito pelo SO)
Suporte a multiprocessamento	Limitado	Completo (SO distribui threads entre núcleos)
Portabilidade	Alta (depende da biblioteca)	Baixa (depende do SO)

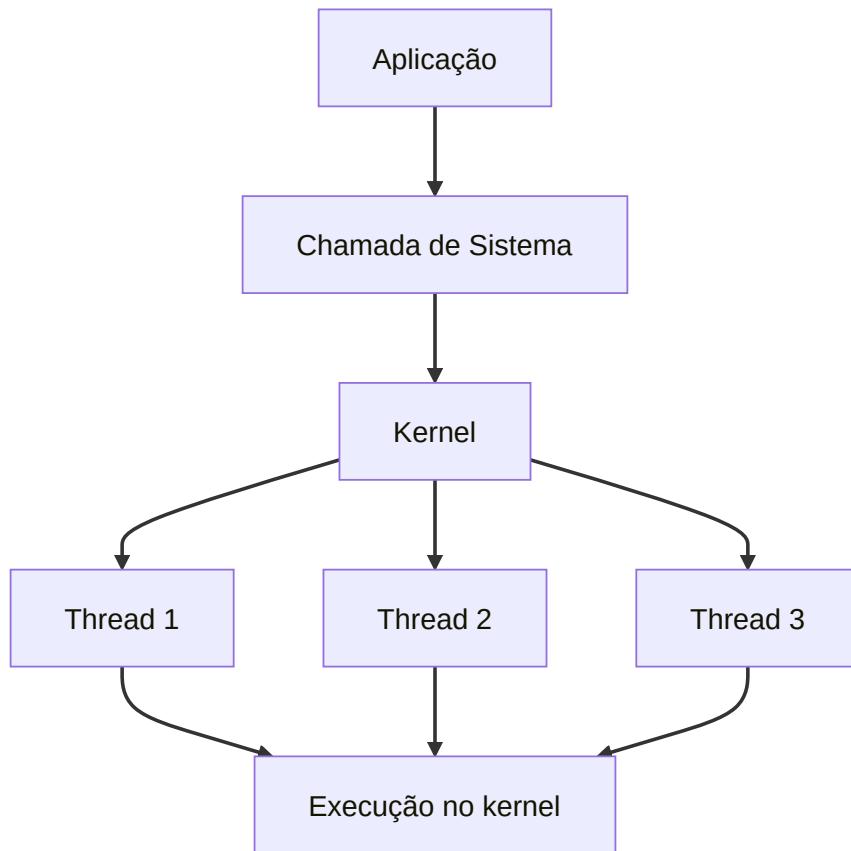
Diagrama de Funcionamento

Threads no Espaço do Usuário



- A aplicação gerencia as threads diretamente, sem interação com o kernel.

Threads no Nível do Kernel



- A aplicação faz chamadas ao kernel para criar e gerenciar threads.

Quando Usar Cada Abordagem

1. Threads no Espaço do Usuário:

- Quando a aplicação precisa de **alto desempenho** e baixo overhead.
- Quando o sistema operacional não suporta threads no nível do kernel.
- Quando o programador precisa de **controle total** sobre o comportamento das threads.

2. Threads no Nível do Kernel:

- Quando a aplicação precisa de **segurança e isolamento**.
- Quando o sistema operacional suporta multiprocessamento e você quer aproveitar ao máximo o hardware.

- Quando a aplicação precisa lidar com operações bloqueantes (como I/O) de forma eficiente.

3. Bibliotecas de Threads no Espaço do Usuário:

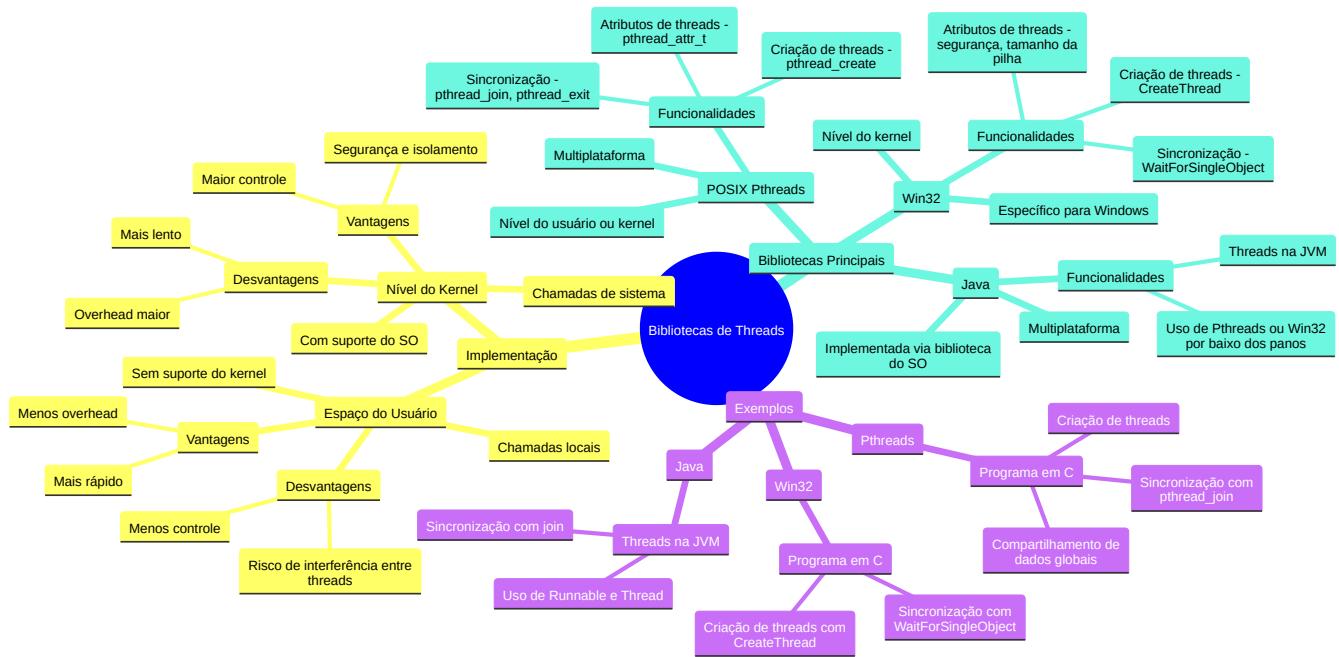
- É como se você e seus amigos estivessem trabalhando em um servidor privado (espaço do usuário). Tudo o que vocês fazem é gerenciado por vocês mesmos, sem precisar pedir permissão ao administrador do servidor (kernel). Isso é rápido e eficiente, mas se alguém cometer um erro (como derrubar um bloco errado), pode afetar todo o grupo. Além disso, vocês têm recursos limitados, pois o servidor privado não tem o poder total do servidor público.

2. Bibliotecas de Threads no Nível do Kernel:

- Agora, imagine que vocês estão em um servidor público (espaço do kernel). Tudo o que vocês fazem precisa ser aprovado pelo administrador do servidor. Isso é mais seguro e organizado, pois o administrador garante que ninguém vai interferir no trabalho dos outros. No entanto, pode ser um pouco mais lento, pois vocês precisam esperar a aprovação do admin para cada ação.

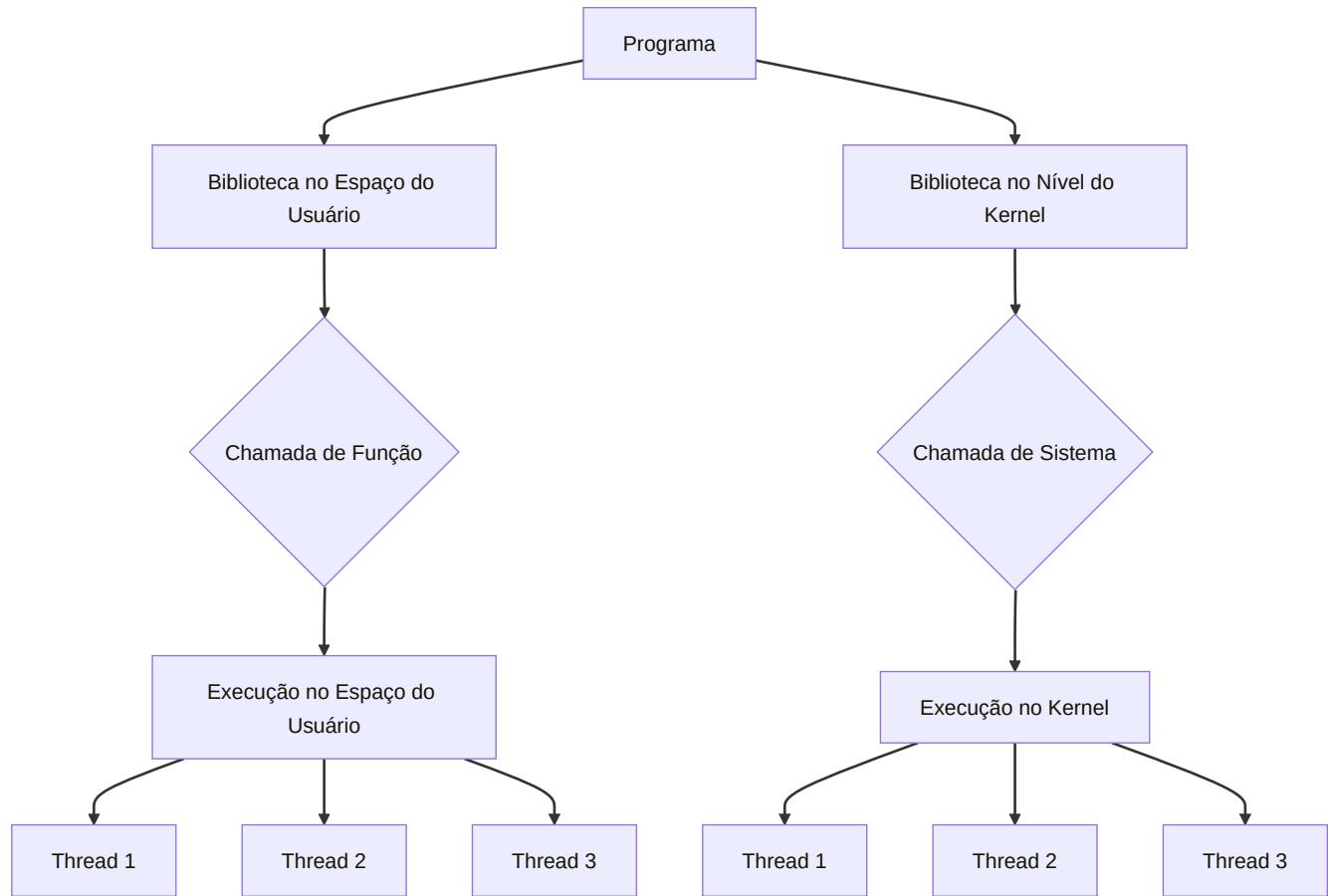
3. Pthreads, Win32 e Java:

- **Pthreads:** É como um manual de instruções universal para construir vilas, que funciona em diferentes servidores (sistemas operacionais). Você pode usá-lo em servidores privados ou públicos. Ele é flexível e amplamente suportado.
- **Win32:** É um manual específico para servidores Windows. Ele é muito eficiente, mas só funciona nesse tipo de servidor. É como ter um guia detalhado para construir no Minecraft, mas que só funciona em um tipo específico de servidor.
- **Java:** É como um manual que funciona em qualquer servidor, mas por baixo dos panos, ele usa o manual específico do servidor (Pthreads no Linux ou Win32 no Windows). É como se você tivesse um tradutor automático que converte suas instruções para o manual do servidor em que você está jogando.



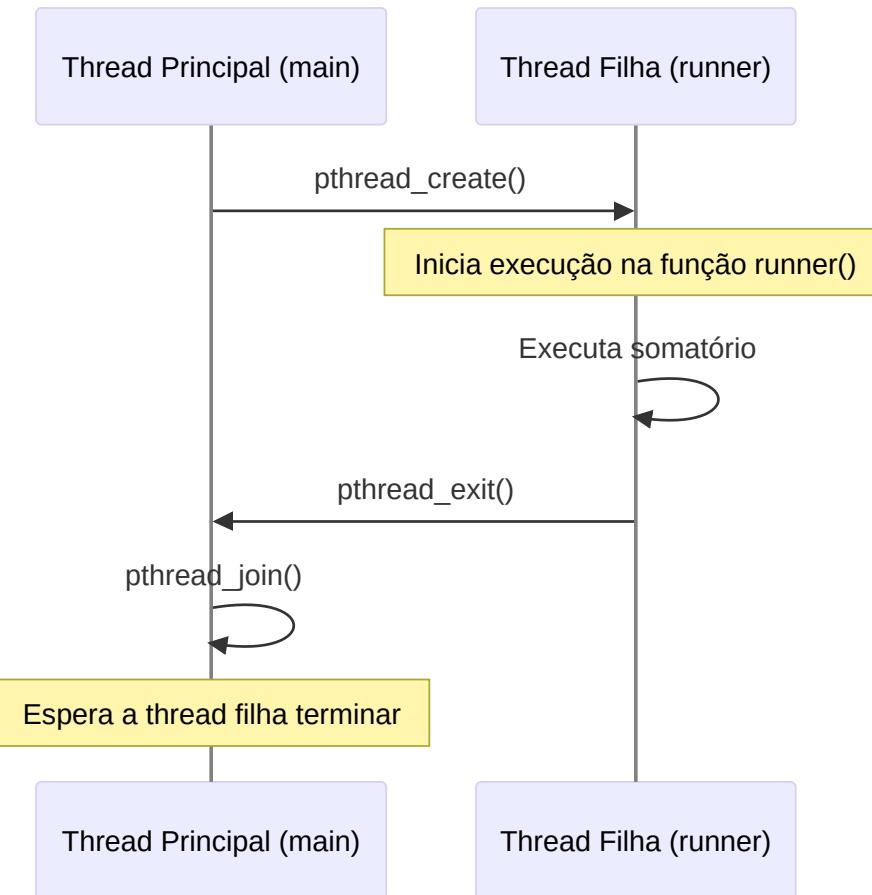
Diagrams Específicos Detalhados

1. Funcionamento de Threads no Espaço do Usuário vs. Nível do Kernel



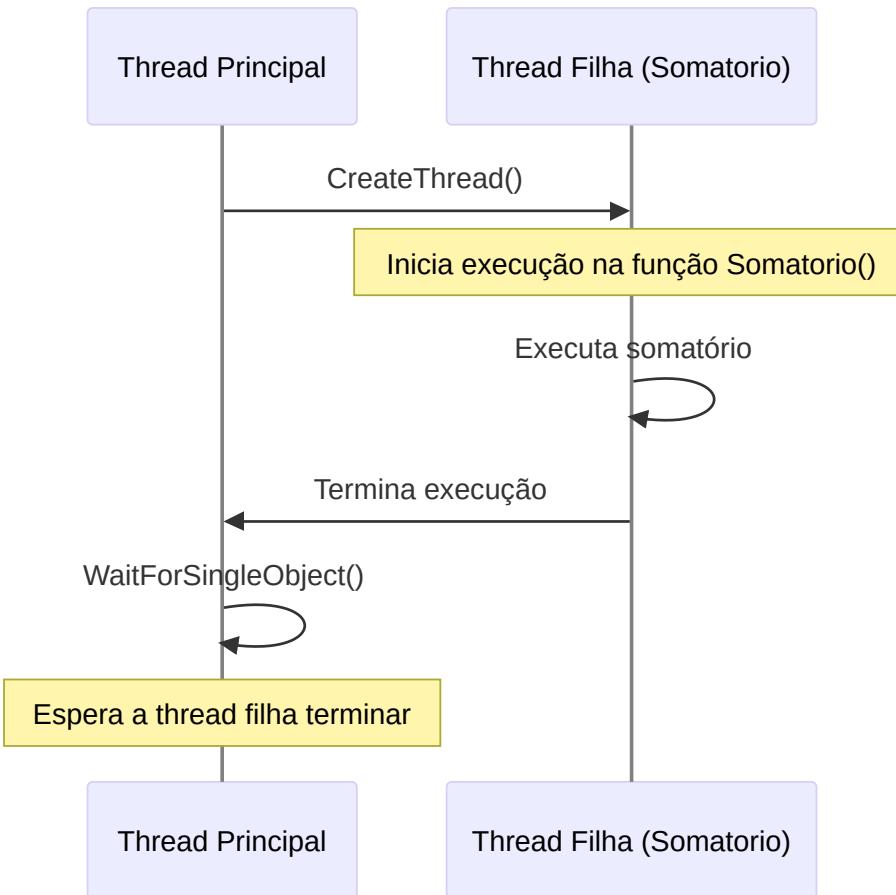
- **Espaço do Usuário:** As threads são gerenciadas pelo próprio programa, sem interação direta com o sistema operacional. Isso é mais rápido, mas menos seguro.
- **Nível do Kernel:** O sistema operacional gerencia as threads, garantindo maior controle e segurança, mas com um overhead maior.

2. Fluxo de Execução com Pthreads



- A thread principal cria uma nova thread com `pthread_create`.
 - A thread filha executa o somatório na função `runner`.
 - A thread filha termina com `pthread_exit`.
 - A thread principal espera a thread filha terminar com `pthread_join`.

3. Fluxo de Execução com Win32



- A thread principal cria uma nova thread com `CreateThread`.
 - A thread filha executa o somatório na função `Somatorio`.
 - A thread filha termina sua execução.
 - A thread principal espera a thread filha terminar com `WaitForSingleObject`.

Explicação Detalhada dos Conceitos

1. Pthreads:

- **Criação de Threads:** Usa `pthread_create` para criar uma nova thread, passando a função que a thread executará (`runner` no exemplo).
- **Sincronização:** Usa `pthread_join` para fazer a thread principal esperar a thread filha terminar.
- **Atributos de Threads:** Podem ser configurados com `pthread_attr_t`, mas no exemplo, usamos os atributos padrão.

2. Win32:

- **Criação de Threads:** Usa `CreateThread`, passando a função `Somatorio` e os atributos da thread.
- **Sincronização:** Usa `WaitForSingleObject` para fazer a thread principal esperar a thread filha terminar.
- **Atributos de Threads:** Incluem segurança, tamanho da pilha e flags de inicialização.

3. Java:

- **Threads na JVM:** A JVM usa a biblioteca de threads do sistema operacional subjacente (Pthreads no Linux, Win32 no Windows).
- **Sincronização:** Usa métodos como `join()` para esperar que uma thread termine.

Exemplos de código na prática

1. Exemplo de Threads no Espaço do Usuário (Pthreads)

Neste exemplo, usamos a biblioteca **Pthreads** para criar e gerenciar threads no espaço do usuário. O programa calcula o somatório de um número inteiro não negativo em uma thread separada.

Código em C

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Variável global para armazenar o resultado do somatório
int sum = 0;

// Função que a thread executará
void* runner(void* param) {
    int upper = atoi(param); // Converte o parâmetro para inteiro
    for (int i = 1; i <= upper; i++) {
        sum += i; // Calcula o somatório
    }
    pthread_exit(0); // Termina a thread
}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <valor>\n", argv[0]);
        return 1;
    }
}
```

```

pthread_t tid; // Identificador da thread
pthread_attr_t attr; // Atributos da thread

// Inicializa os atributos da thread com os valores padrão
pthread_attr_init(&attr);

// Cria a thread
pthread_create(&tid, &attr, runner, argv[1]);

// Espera a thread terminar
pthread_join(tid, NULL);

// Exibe o resultado
printf("Somatório = %d\n", sum);

return 0;
}

```

Explicação do Código

1. Variável Global sum:

- Armazena o resultado do somatório. Como é global, é compartilhada entre a thread principal e a thread filha.

2. Função runner:

- É a função que a thread filha executa. Ela calcula o somatório de 1 até o valor passado como argumento.

3. Criação da Thread:

- `pthread_create` cria uma nova thread que executa a função `runner`.
- O argumento `argv[1]` (valor passado na linha de comando) é passado para a thread.

4. Sincronização:

- `pthread_join` faz a thread principal esperar a thread filha terminar.

5. Saída:

- O resultado do somatório é exibido após a thread filha terminar.

Como Executar

Compile o programa com:

```
gcc -o somatorio somatorio.c -lpthread
```

Execute passando um valor:

```
./somatorio 5
```

Saída esperada:

```
Somatório = 15
```

2. Exemplo de Threads no Nível do Kernel (Win32)

Neste exemplo, usamos a API Win32 para criar e gerenciar threads no nível do kernel. O programa também calcula o somatório de um número inteiro não negativo, mas usando a API específica do Windows.

Código em C

```
#include <windows.h>
#include <stdio.h>

// Variável global para armazenar o resultado do somatório
DWORD sum = 0;

// Função que a thread executará
DWORD WINAPI Somatorio(LPVOID param) {
    int upper = *(int*)param; // Converte o parâmetro para inteiro
    for (int i = 1; i <= upper; i++) {
        sum += i; // Calcula o somatório
    }
    return 0; // Termina a thread
}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <valor>\n", argv[0]);
        return 1;
    }
```

```

int upper = atoi(argv[1]); // Converte o argumento para inteiro
HANDLE hThread; // Handle para a thread
DWORD threadID; // ID da thread

// Cria a thread
hThread = CreateThread(
    NULL, // Atributos de segurança padrão
    0, // Tamanho da pilha padrão
    Somatorio, // Função que a thread executará
    &upper, // Argumento para a função
    0, // Flags de criação (0 = execução imediata)
    &threadID // ID da thread
);

if (hThread == NULL) {
    fprintf(stderr, "Erro ao criar a thread.\n");
    return 1;
}

// Espera a thread terminar
WaitForSingleObject(hThread, INFINITE);

// Fecha o handle da thread
CloseHandle(hThread);

// Exibe o resultado
printf("Somatório = %lu\n", sum);

return 0;
}

```

Explicação do Código

1. Variável Global sum:

- Armazena o resultado do somatório. É compartilhada entre a thread principal e a thread filha.

2. Função Somatorio:

- É a função que a thread filha executa. Ela calcula o somatório de 1 até o valor passado como argumento.

3. Criação da Thread:

- `CreateThread` cria uma nova thread que executa a função `Somatorio`.
- O argumento `upper` (valor passado na linha de comando) é passado para a thread.

4. Sincronização:

- `WaitForSingleObject` faz a thread principal esperar a thread filha terminar.

5. Saída:

- O resultado do somatório é exibido após a thread filha terminar.

Como Executar

Compile o programa com um compilador compatível com Windows (como o MinGW ou Visual Studio):

```
gcc -o somatorio_win32 somatorio_win32.c -lws2_32
```

Execute passando um valor:

```
somatorio_win32 5
```

Saída esperada:

```
Somatório = 15
```

Comparação entre os Exemplos

Característica	Pthreads (Espaço do Usuário)	Win32 (Nível do Kernel)
Biblioteca	Pthreads	Win32 API
Chamadas de sistema	Não usa	Usa (CreateThread, WaitForSingleObject)
Portabilidade	Multiplataforma (Linux, macOS, etc.)	Específico para Windows
Overhead	Menor	Maior (devido a chamadas de sistema)
Controle	Total (programador gerencia threads)	Limitado (SO gerencia threads)

4.6 Threads em Java

Explicação Detalhada

1. Threads em Java: Visão Geral

Em Java, as threads são fundamentais para a execução de programas concorrentes. Todo programa Java começa com pelo menos uma thread, chamada de **thread principal**, que executa o método `main()`. A partir daí, outras threads podem ser criadas para realizar tarefas em paralelo.

2. Criando Threads em Java

Existem duas maneiras principais de criar threads em Java:

1. Estendendo a classe Thread:

- Cria-se uma nova classe que herda de `Thread` e sobrescreve o método `run()`.
- Exemplo:

```
class MinhaThread extends Thread {  
    public void run() {  
        System.out.println("Thread em execução!");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        MinhaThread thread = new MinhaThread();  
        thread.start(); // Inicia a thread  
    }  
}
```

2. Implementando a interface Runnable:

- Cria-se uma classe que implementa `Runnable` e define o método `run()`.
- Essa abordagem é mais flexível, pois permite que a classe herde de outra classe.
- Exemplo:

```
class MeuRunnable implements Runnable {  
    public void run() {  
        System.out.println("Thread em execução!");  
    }  
}
```

```

        }
    }

public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread(new MeuRunnable());
        thread.start(); // Inicia a thread
    }
}

```

3. Exemplo Completo: Somatório com Threads

Vamos implementar o exemplo do somatório de um número inteiro não negativo usando threads em Java.

Código Java

```

class Somatorio implements Runnable {
    private int upper; // Limite superior do somatório
    private int sum = 0; // Resultado do somatório

    // Construtor
    public Somatorio(int upper) {
        this.upper = upper;
    }

    // Método run (executado pela thread)
    public void run() {
        for (int i = 1; i <= upper; i++) {
            sum += i;
        }
        System.out.println("Somatório até " + upper + " = " + sum);
    }

    // Método para obter o resultado do somatório
    public int getSum() {
        return sum;
    }
}

public class Main {
    public static void main(String[] args) {
        if (args.length != 1) {

```

```

        System.out.println("Uso: java Main <valor>");
        return;
    }

    int upper = Integer.parseInt(args[0]); // Converte o argumento
    para inteiro
    Somatorio task = new Somatorio(upper); // Cria a tarefa
    Thread thread = new Thread(task); // Cria a thread
    thread.start(); // Inicia a thread

    try {
        thread.join(); // Espera a thread terminar
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("Resultado final: " + task.getSum());
}
}

```

Explicação do Código

1. Classe Somatorio:

- Implementa `Runnable` e define o método `run()`, que calcula o somatório.
- O resultado é armazenado na variável `sum`.

2. Classe Main:

- Cria uma instância de `Somatorio` e uma `thread` associada a ela.
- Inicia a `thread` com `start()` e espera seu término com `join()`.
- Exibe o resultado final.

Como Executar

Compile e execute o programa:

```

javac Main.java
java Main 5

```

Saída esperada:

Somatório até 5 = 15

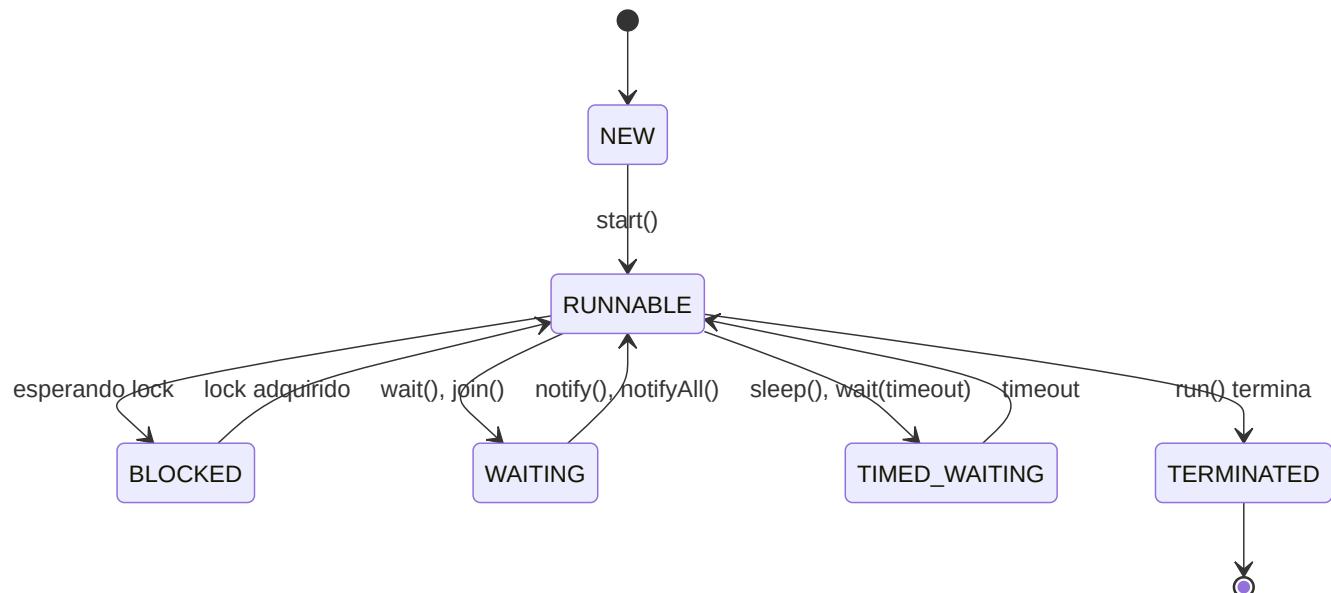
Resultado final: 15

4. Estados de uma Thread em Java

Uma thread em Java pode estar em um dos seguintes estados:

1. **NEW**: A thread foi criada, mas ainda não foi iniciada.
2. **RUNNABLE**: A thread está em execução ou pronta para executar.
3. **BLOCKED**: A thread está bloqueada, esperando por um lock.
4. **WAITING**: A thread está esperando indefinidamente por outra thread.
5. **TIMED_WAITING**: A thread está esperando por um tempo específico.
6. **TERMINATED**: A thread terminou sua execução.

Diagrama de Estados



5. Threads Daemon vs. Não Daemon

- **Threads Daemon:**
 - São threads de baixa prioridade que rodam em segundo plano.
 - A JVM termina quando todas as threads **não daemon** terminam.
 - Exemplo: Garbage Collector.

- Definida com `thread.setDaemon(true)`.
- **Threads Não Daemon:**
 - São threads comuns.
 - A JVM espera que todas terminem antes de encerrar.

6. JVM e o Sistema Operacional Hospedeiro

A JVM pode mapear threads Java para threads do sistema operacional de diferentes formas:

- **Modelo 1:1:** Cada thread Java é associada a uma thread do kernel (usado no Windows).
- **Modelo M:N:** Várias threads Java são mapeadas para um número menor de threads do kernel (usado em alguns sistemas UNIX).
- **Modelo M:1:** Várias threads Java são mapeadas para uma única thread do kernel (antigo modelo "green threads").

Exemplo Completo: Produtor-Consumidor

Vamos implementar uma solução para o problema clássico do produtor-consumidor usando threads em Java.

Código Java

```
import java.util.LinkedList;
import java.util.Queue;

class MessageQueue {
    private Queue<String> queue = new LinkedList<>();
    private int capacity;

    public MessageQueue(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void send(String message) throws
InterruptedException {
        while (queue.size() == capacity) {
            wait(); // Espera se a fila estiver cheia
        }
        queue.add(message);
        notifyAll(); // Notifica os consumidores
    }
}
```

```

}

public synchronized String receive() throws InterruptedException {
    while (queue.isEmpty()) {
        wait(); // Espera se a fila estiver vazia
    }
    String message = queue.poll();
    notifyAll(); // Notifica os produtores
    return message;
}

class Produtor implements Runnable {
    private MessageQueue queue;

    public Produtor(MessageQueue queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                String message = "Mensagem " + i;
                queue.send(message);
                System.out.println("Produzido: " + message);
                Thread.sleep(500); // Simula tempo de produção
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Consumidor implements Runnable {
    private MessageQueue queue;

    public Consumidor(MessageQueue queue) {
        this.queue = queue;
    }

    public void run() {
        try {

```

```

        for (int i = 0; i < 10; i++) {
            String message = queue.receive();
            System.out.println("Consumido: " + message);
            Thread.sleep(1000); // Simula tempo de consumo
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public class Main {
    public static void main(String[] args) {
        MessageQueue queue = new MessageQueue(5); // Fila com capacidade 5
        Thread produtor = new Thread(new Produtor(queue));
        Thread consumidor = new Thread(new Consumidor(queue));

        produtor.start();
        consumidor.start();
    }
}

```

Explicação do Código

1. MessageQueue:

- Gerencia uma fila de mensagens com capacidade limitada.
- Usa `wait()` e `notifyAll()` para sincronização.

2. Produtor:

- Gera mensagens e as envia para a fila.

3. Consumidor:

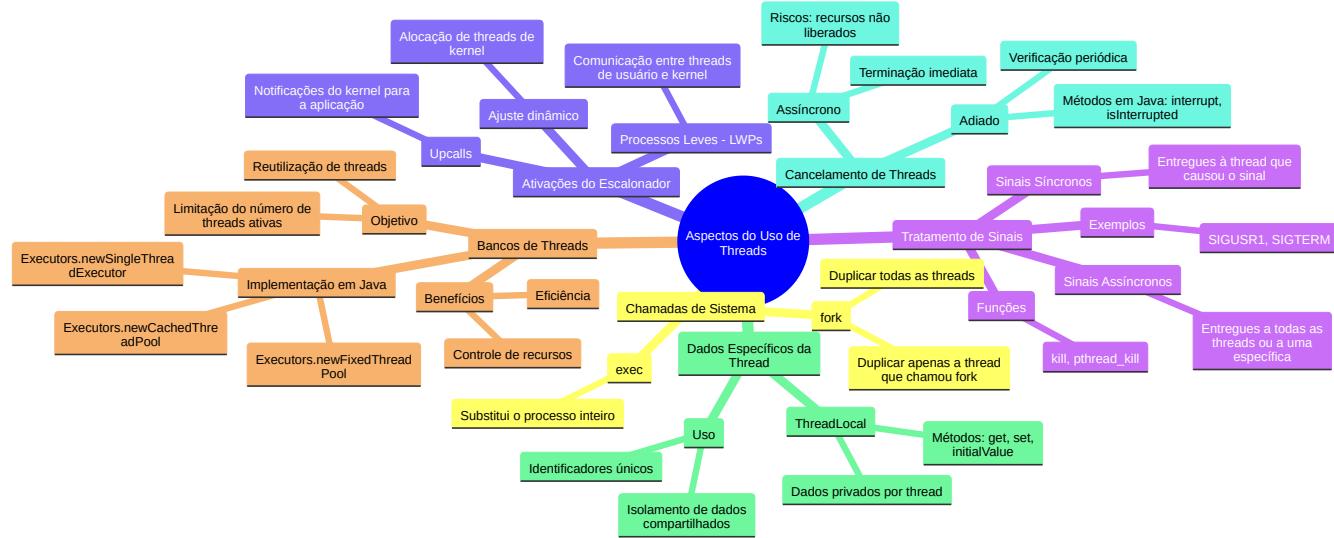
- Recebe e processa mensagens da fila.

4. Main:

- Cria a fila e inicia as threads do produtor e consumidor.

4.7 Aspectos do Uso de Threads

Vamos explorar os **aspectos do uso de threads** de forma detalhada, com exemplos práticos, analogias e diagramas para facilitar o entendimento.



1. Chamadas de Sistema:

- Aborda o comportamento de `fork()` e `exec()` em programas multithread, destacando as duas versões de `fork()` e o impacto de `exec()`.

2. Cancelamento de Threads:

- Discute as técnicas de cancelamento assíncrono e adiado, com exemplos em Java usando `interrupt()` e `isInterrupted()`.

3. Tratamento de Sinais:

- Explora como os sinais são entregues em programas multithread, diferenciando sinais síncronos e assíncronos, e como são tratados em sistemas UNIX e Windows.

4. Bancos de Threads:

- Explica a criação e uso de bancos de threads para melhorar a eficiência e o controle de recursos, com exemplos práticos em Java.

5. Dados Específicos da Thread:

- Introduz o conceito de `ThreadLocal` para armazenar dados privados por thread, útil em cenários como processamento de transações.

6. Ativações do Escalonador:

- Descreve a comunicação entre threads de usuário e kernel por meio de LWPs e upcalls, permitindo ajustes dinâmicos no escalonamento.

1. Chamadas de Sistema fork() e exec()

Problema

Quando uma thread em um programa multithread chama `fork()`, o novo processo deve duplicar todas as threads ou apenas a thread que chamou `fork()`? Além disso, como a chamada `exec()` afeta as threads?

Solução

- Duas versões de `fork()`:
 1. Duplicar todas as threads: O novo processo terá uma cópia de todas as threads do processo original.
 2. Duplicar apenas a thread que chamou `fork()`: O novo processo terá apenas uma thread.
- Escolha da versão:
 - Se `exec()` for chamado logo após `fork()`, duplicar todas as threads é desnecessário, pois o programa será substituído.
 - Se `exec()` não for chamado, o novo processo deve duplicar todas as threads para manter a funcionalidade.

Exemplo em C

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void* thread_func(void* arg) {
    printf("Thread filha em execução\n");
    sleep(2);
    printf("Thread filha terminou\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_func, NULL);
```

```

pid_t pid = fork();
if (pid == 0) { // Processo filho
    printf("Processo filho criado\n");
    execlp("ls", "ls", NULL); // Substitui o processo filho
} else if (pid > 0) { // Processo pai
    printf("Processo pai esperando\n");
    pthread_join(thread, NULL);
}

return 0;
}

```

Explicação

- O processo filho criado por `fork()` substitui seu espaço de memória com `exec()`, então apenas a thread que chamou `fork()` é duplicada.

2. Cancelamento de Threads

Problema

Cancelar uma thread antes que ela termine sua execução pode ser necessário, mas isso pode causar problemas se a thread estiver manipulando recursos compartilhados.

Solução

- **Cancelamento Assíncrono:** A thread é terminada imediatamente.
- **Cancelamento Adiado:** A thread verifica periodicamente se deve ser cancelada, permitindo uma finalização segura.

Exemplo em Java

```

class InterruptibleThread implements Runnable {
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            System.out.println("Thread em execução");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Thread interrompida");
            }
        }
    }
}

```

```

        Thread.currentThread().interrupt(); // Restaura o status
de interrupção
    }
}
System.out.println("Thread terminada");
}
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new InterruptibleThread());
        thread.start();

        Thread.sleep(3000); // Espera 3 segundos
        thread.interrupt(); // Interrompe a thread
    }
}

```

Explicação

- A thread verifica seu status de interrupção com `isInterrupted()` e termina de forma segura.

3. Tratamento de Sinais

Problema

Em programas multithread, os sinais podem ser entregues a uma thread específica ou a todas as threads, dependendo do tipo de sinal.

Solução

- **Sinais Síncronos:** Entregues à thread que causou o sinal.
- **Sinais Assíncronos:** Podem ser entregues a todas as threads ou a uma thread específica.

Exemplo em C (UNIX)

```
#include <stdio.h>
#include <signal.h>
#include <pthread.h>
#include <unistd.h>
```

```

void handle_signal(int sig) {
    printf("Sinal %d recebido pela thread %ld\n", sig,
    (long)pthread_self());
}

void* thread_func(void* arg) {
    signal(SIGUSR1, handle_signal);
    while (1) {
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, thread_func, NULL);
    pthread_create(&thread2, NULL, thread_func, NULL);

    sleep(2);
    pthread_kill(thread1, SIGUSR1); // Envia sinal para thread1
    pthread_kill(thread2, SIGUSR1); // Envia sinal para thread2

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}

```

Explicação

- O sinal `SIGUSR1` é enviado para threads específicas usando `pthread_kill()`.

4. Bancos de Threads

Problema

Criar uma nova thread para cada requisição em um servidor pode ser ineficiente e consumir muitos recursos.

Solução

- **Bancos de Threads:** Um conjunto de threads é criado no início e reutilizado para atender requisições.

Exemplo em Java

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Task implements Runnable {
    private int id;

    public Task(int id) {
        this.id = id;
    }

    public void run() {
        System.out.println("Task " + id + " executada por " +
Thread.currentThread().getName());
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3); // 
Banco com 3 threads

        for (int i = 1; i <= 10; i++) {
            executor.execute(new Task(i));
        }

        executor.shutdown();
    }
}

```

Explicação

- O banco de threads com 3 threads executa 10 tarefas, reutilizando as threads disponíveis.

5. Dados Específicos da Thread

Problema

Threads compartilham dados globais, mas às vezes cada thread precisa de sua própria cópia de dados.

Solução

- **ThreadLocal**: Permite que cada thread tenha sua própria cópia de dados.

Exemplo em Java

```
class ThreadLocalExample {  
    private static ThreadLocal<Integer> threadLocal =  
        ThreadLocal.withInitial(() -> 0);  
  
    public static void main(String[] args) {  
        Runnable task = () -> {  
            int value = threadLocal.get();  
            threadLocal.set(value + 1);  
            System.out.println(Thread.currentThread().getName() + ":" +  
                threadLocal.get());  
        };  
  
        Thread thread1 = new Thread(task);  
        Thread thread2 = new Thread(task);  
  
        thread1.start();  
        thread2.start();  
    }  
}
```

Explicação

- Cada thread mantém sua própria cópia do valor em `threadLocal`.

6. Ativações do Escalonador (Scheduler Activations)

Problema

A comunicação entre threads de usuário e threads do kernel pode ser necessária para ajustar dinamicamente o número de threads de kernel.

Solução

- **Processos Leves (LWPs):** Estruturas intermediárias que permitem a comunicação entre threads de usuário e threads do kernel.
- **Upcalls:** O kernel notifica a aplicação sobre eventos, como o bloqueio de uma thread.

Exemplo Conceitual

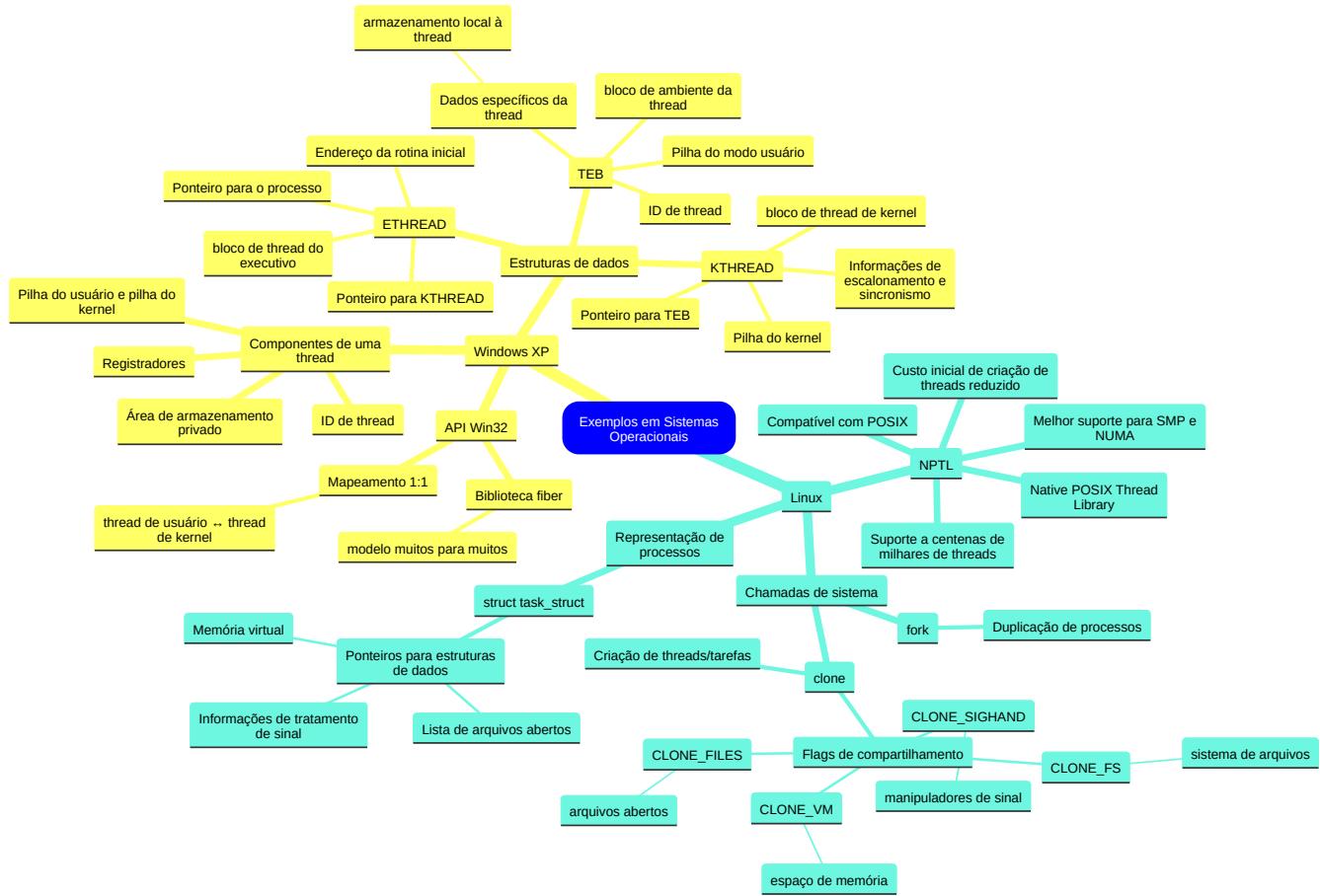
1. O kernel aloca LWPs para a aplicação.
2. Quando uma thread de usuário é bloqueada, o kernel faz um upcall para a aplicação.
3. A aplicação salva o estado da thread bloqueada e escalona outra thread no LWP disponível.

Resumo

Tópico	Descrição
fork() e exec()	Duplicação de threads e substituição de processos.
Cancelamento de Threads	Assíncrono (imediato) ou adiado (seguro).
Tratamento de Sinais	Entregues a threads específicas ou a todas as threads.
Bancos de Threads	Reutilização de threads para melhorar eficiência.
Dados Específicos	Uso de ThreadLocal para dados privados por thread.
Ativações do Escalonador	Comunicação entre threads de usuário e kernel via LWPs e upcalls.

4.8 Exemplos em Sistemas Operacionais

Nesta seção, exploramos como as **threads** são implementadas em dois sistemas operacionais populares: **Windows XP** e **Linux**. Cada sistema operacional tem sua própria abordagem para gerenciar threads, refletindo suas filosofias de design e necessidades específicas. Vamos detalhar cada um deles.



Threads no Windows XP

O Windows XP utiliza a **API Win32**, que é a principal interface para criação e gerenciamento de threads na família de sistemas operacionais da Microsoft (Windows 95, 98, NT, 2000 e XP). Aqui estão os principais pontos:

1. Mapeamento 1:1

- O Windows XP usa o **modelo de mapeamento 1:1**, onde cada thread no nível do usuário é associada a uma thread no nível do kernel.

- Isso significa que o sistema operacional gerencia diretamente cada thread, o que simplifica o escalonamento e a sincronização, mas pode limitar a escalabilidade em sistemas com muitas threads.

2. Biblioteca Fiber

- Além do modelo 1:1, o Windows XP oferece suporte à **biblioteca fiber**, que implementa o **modelo muitos para muitos**.
- Nesse modelo, várias threads de usuário são mapeadas para um número menor de threads de kernel, permitindo maior flexibilidade e eficiência em certos cenários.

3. Componentes de uma Thread

Cada thread no Windows XP é composta por:

- **ID da thread**: Identifica a thread de forma única.
- **Registradores**: Armazenam o estado atual da CPU.
- **Pilhas**: Uma pilha para o modo usuário e outra para o modo kernel.
- **Área de armazenamento privado**: Usada por bibliotecas em tempo de execução e DLLs.

4. Estruturas de Dados

O Windows XP utiliza três estruturas de dados principais para gerenciar threads:

- **ETHREAD (Executive Thread Block)**:
 - Armazena informações sobre o processo ao qual a thread pertence.
 - Contém o endereço da rotina onde a thread começa a executar.
 - Aponta para a estrutura **KTHREAD** correspondente.
- **KTHREAD (Kernel Thread Block)**:
 - Gerencia informações de escalonamento e sincronização.
 - Contém a pilha do kernel, usada quando a thread está no modo kernel.
 - Aponta para a estrutura **TEB**.
- **TEB (Thread Environment Block)**:

- Estrutura no espaço do usuário que contém dados específicos da thread, como a pilha do usuário e um array para armazenamento local à thread.

5. Conclusão sobre Windows XP

O Windows XP é projetado para oferecer um gerenciamento robusto de threads, com suporte tanto para o modelo 1:1 quanto para o modelo muitos para muitos (via fibers). Suas estruturas de dados são bem definidas, permitindo um controle eficiente das threads no nível do kernel e do usuário.

4.6.2 Threads no Linux

O Linux tem uma abordagem diferente para threads, baseada na ideia de **tarefas** (tasks), que podem ser tanto processos quanto threads. Aqui estão os principais pontos:

1. Chamadas de Sistema

- **fork()**:
 - Cria um novo processo duplicando o processo atual.
 - Não há compartilhamento de recursos entre o processo pai e o filho.
- **clone()**:
 - Permite criar threads (ou tarefas) com diferentes níveis de compartilhamento de recursos.
 - Dependendo dos **flags** passados, a nova tarefa pode compartilhar recursos como memória, arquivos abertos e manipuladores de sinais.

2. Flags do clone()

O **clone()** aceita vários flags que determinam o nível de compartilhamento entre a tarefa pai e a filha:

- **CLONE_FS**: Compartilha informações do sistema de arquivos (ex.: diretório atual).
- **CLONE_VM**: Compartilha o espaço de memória virtual.
- **CLONE_SIGHAND**: Compartilha manipuladores de sinais.
- **CLONE_FILES**: Compartilha arquivos abertos.

3. Representação de Processos

- No Linux, cada processo ou thread é representado por uma estrutura de dados chamada **struct task_struct**.

- Essa estrutura não armazena diretamente os dados do processo, mas contém **ponteiros** para outras estruturas que gerenciam recursos como:
 - Lista de arquivos abertos.
 - Informações de tratamento de sinais.
 - Memória virtual.

4. NPTL (Native POSIX Thread Library)

- O Linux moderno utiliza a **NPTL**, uma biblioteca de threads compatível com o padrão POSIX.
- A NPTL oferece:
 - Melhor suporte para sistemas **SMP (Symmetric Multiprocessing)** e **NUMA (Non-Uniform Memory Access)**.
 - Custo reduzido para criação de threads.
 - Suporte a centenas de milhares de threads, o que é essencial para sistemas multicore e servidores de alta carga.

5. Conclusão sobre Linux

O Linux trata threads e processos de forma semelhante, usando a estrutura `task_struct` e a chamada `clone()` para gerenciar o compartilhamento de recursos. A NPTL trouxe melhorias significativas, especialmente em sistemas multiprocessados, tornando o Linux uma plataforma robusta para aplicações multithread.

Comparação entre Windows XP e Linux

Característica	Windows XP	Linux
Modelo de Threads	Mapeamento 1:1 (com suporte a fibers)	Tarefas (processos/threads) via <code>clone()</code>
Chamadas de Sistema	API Win32 (<code>CreateThread</code> , etc.)	<code>fork()</code> e <code>clone()</code>
Compartilhamento	Definido pelo sistema	Configurável via flags no <code>clone()</code>
Biblioteca de Threads	Biblioteca fiber	NPTL (POSIX-compliant)
Estruturas de Dados	<code>ETHREAD</code> , <code>KTHREAD</code> , <code>TEB</code>	<code>struct task_struct</code>
Escalabilidade	Limitada pelo modelo 1:1	Alta (suporte a centenas de milhares de threads)

Conclusão Geral

Tanto o Windows XP quanto o Linux oferecem suporte robusto para threads, mas com abordagens diferentes:

- O Windows XP prioriza o controle direto sobre as threads, com estruturas de dados bem definidas e suporte a modelos de mapeamento flexíveis.
- O Linux trata threads como tarefas, com compartilhamento de recursos configurável via `clone()`, e a NPTL trouxe melhorias significativas para sistemas modernos.

Essas diferenças refletem as filosofias de design de cada sistema operacional e suas aplicações típicas. Ambos são eficientes em seus contextos, mas o Linux se destaca em cenários que exigem alta escalabilidade e suporte a sistemas multiprocessados.

Exercícios Práticos - 4

4.1. Prepare dois exemplos de programação nos quais o uso de multithreading ofereça melhor desempenho do que uma solução de única thread.

Exemplo 1: Download de Múltiplos Arquivos

- **Problema:** Baixar vários arquivos de um servidor.
- **Solução com Multithreading:**
 - Cada thread pode ser responsável por baixar um arquivo individualmente.
 - Enquanto uma thread espera por I/O (download), outras threads podem continuar trabalhando.
- **Vantagem:** O tempo total de download é reduzido, pois os downloads ocorrem em paralelo.

Exemplo 2: Processamento de Imagens

- **Problema:** Aplicar filtros (como desfoque ou detecção de bordas) em várias imagens.
- **Solução com Multithreading:**
 - Cada thread processa uma imagem independentemente.
 - O processamento é distribuído entre os núcleos da CPU.
- **Vantagem:** O tempo total de processamento é reduzido, especialmente em CPUs multicore.

4.2. Quais são as duas diferenças entre as threads em nível de usuário e as threads em nível de kernel? Sob quais circunstâncias um tipo é melhor do que o outro?

Diferenças

1. Gerenciamento:

- **Threads em nível de usuário:** Gerenciadas pela aplicação (biblioteca de threads).
- **Threads em nível de kernel:** Gerenciadas diretamente pelo sistema operacional.

2. Troca de Contexto:

- **Threads em nível de usuário:** A troca de contexto é mais rápida, pois não envolve o kernel.
- **Threads em nível de kernel:** A troca de contexto é mais lenta, pois envolve uma chamada ao sistema.

Circunstâncias

• Threads em nível de usuário:

- Melhor para aplicações que exigem muitas threads e trocas de contexto frequentes.
- Exemplo: Servidores web com alta concorrência.

• Threads em nível de kernel:

- Melhor para aplicações que exigem integração com o sistema operacional (ex.: operações de I/O bloqueantes).
- Exemplo: Aplicações de tempo real.

4.3. Descreva as ações tomadas por um kernel para a troca de contexto entre as threads em nível de kernel.

1. Salvar o estado da thread atual:

- O kernel salva os registradores da CPU, o contador de programa e a pilha da thread que está sendo interrompida.

2. Escolher a próxima thread:

- O escalonador do kernel seleciona a próxima thread a ser executada com base em políticas de escalonamento.

3. Restaurar o estado da próxima thread:

- O kernel restaura os registradores, o contador de programa e a pilha da próxima thread.

4. Retomar a execução:

- A CPU começa a executar a próxima thread a partir do ponto onde ela foi interrompida.

4.4. Quais recursos são usados quando uma thread é criada? Qual a diferença entre eles e aqueles usados quando um processo é

criado?

Recursos usados na criação de uma thread

1. **Espaço de endereçamento:** Compartilhado com outras threads do mesmo processo.
2. **Pilha:** Cada thread tem sua própria pilha.
3. **Registradores:** Cada thread tem seu próprio conjunto de registradores.
4. **Contexto de execução:** Inclui o contador de programa e o estado da CPU.

Diferença em relação à criação de um processo

1. **Espaço de endereçamento:** Um processo tem seu próprio espaço de endereçamento, enquanto threads compartilham o mesmo espaço.
2. **Recursos do sistema:** Processos exigem mais recursos, como tabelas de páginas e descritores de arquivos.
3. **Custo:** Criar uma thread é mais rápido e consome menos recursos do que criar um processo.

4.5. Suponha que um sistema operacional faça um mapeamento entre as threads em nível de usuário e o kernel, usando o modelo muitos para muitos, e que o mapeamento seja feito por meio de LWPs. Além do mais, o sistema permite que os desenvolvedores criem threads em tempo real para uso em sistemas de tempo real. É necessário vincular uma thread em tempo real a um processo leve? Explique.

Resposta

- **Não é necessário** vincular uma thread em tempo real a um LWP (Lightweight Process).
- **Motivo:** Threads em tempo real geralmente exigem controle direto sobre o hardware e o escalonamento, o que é melhor gerenciado pelo kernel sem a camada intermediária de LWPs.
- **Benefício:** Isso permite que as threads em tempo real tenham prioridade máxima e sejam escalonadas de forma preemptiva, garantindo atendimento de prazos rígidos.

4.6. Um programa Pthread que executa a função de somatório foi apresentado abaixo. Reescreva esse programa em Java.

Código Original em C (Pthreads)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int sum = 0;

void* runner(void* param) {
    int upper = atoi(param);
    for (int i = 1; i <= upper; i++) {
        sum += i;
    }
    pthread_exit(0);
}

int main(int argc, char* argv[]) {
    pthread_t tid;
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, argv[1]);
    pthread_join(tid, NULL);

    printf("Somatório = %d\n", sum);
    return 0;
}
```

Código em Java

```
class Somatorio implements Runnable {
    private int upper;
    private int sum = 0;

    public Somatorio(int upper) {
        this.upper = upper;
    }

    public void run() {
        for (int i = 1; i <= upper; i++) {
            sum += i;
        }
    }
}
```

```

    }
    System.out.println("Somatório = " + sum);
}

public static void main(String[] args) {
    if (args.length != 1) {
        System.out.println("Uso: java Somatorio <valor>");
        return;
    }

    int upper = Integer.parseInt(args[0]);
    Somatorio task = new Somatorio(upper);
    Thread thread = new Thread(task);
    thread.start();

    try {
        thread.join(); // Espera a thread terminar
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

Explicação

1. Classe Somatorio:

- Implementa a interface `Runnable` para definir a tarefa da thread.
- O método `run()` calcula o somatório.

2. Thread Principal:

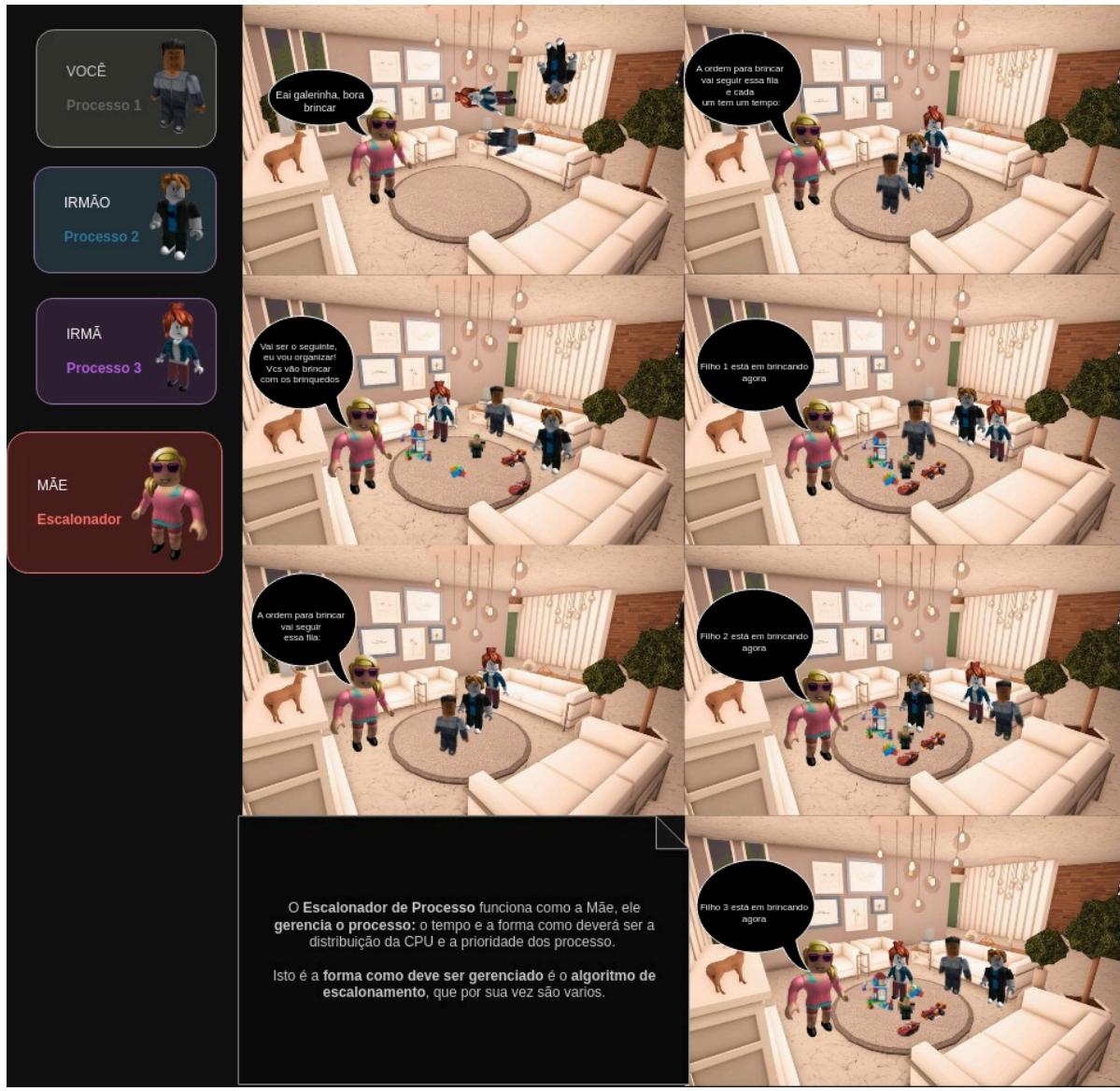
- Cria uma instância de `Somatorio` e uma thread associada.
- Inicia a thread com `start()` e espera seu término com `join()`.

3. Saída:

- O resultado do somatório é exibido após a thread terminar.

Escalonamento de CPU

O escalonamento de CPU é um dos conceitos fundamentais dos sistemas operacionais multiprogramados. Ele permite que o sistema operacional gerencie a alocação da CPU entre os processos (ou threads), tornando o computador mais produtivo e responsivo. Nesta seção, exploramos os conceitos básicos do escalonamento de CPU, os principais algoritmos utilizados e os critérios para selecionar o algoritmo mais adequado para um sistema específico.



Escalonamento de processos

Conceitos Básicos do Escalonamento de CPU

1. O que é Escalonamento de CPU?

- O escalonamento de CPU é o processo de decidir qual processo (ou thread) deve receber a CPU para execução em um determinado momento.
- Em sistemas multiprogramados, vários processos competem pela CPU, e o escalonador (scheduler) é responsável por gerenciar essa competição.

2. Objetivos do Escalonamento:

- **Maximizar a utilização da CPU:** Garantir que a CPU esteja sempre ocupada, evitando ociosidade.
- **Garantir justiça:** Todos os processos devem ter uma chance justa de usar a CPU.
- **Minimizar o tempo de resposta:** Reduzir o tempo que os processos levam para serem executados.
- **Maximizar o throughput:** Executar o maior número possível de processos em um determinado período.

3. Tipos de Escalonamento:

- **Escalonamento de Processos:** Quando o sistema operacional gerencia processos.
- **Escalonamento de Threads:** Quando o sistema operacional gerencia threads no nível do kernel.

Algoritmos de Escalonamento de CPU

Aqui estão alguns dos principais algoritmos de escalonamento de CPU:

1. First-Come, First-Served (FCFS)

- **Funcionamento:** O primeiro processo que chega é o primeiro a ser executado.
- **Vantagem:** Simples de implementar.
- **Desvantagem:** Pode causar o problema do "convoy effect", onde processos longos atrasam processos curtos.

2. Shortest-Job-First (SJF)

- **Funcionamento:** O processo com o menor tempo de execução é selecionado primeiro.
- **Vantagem:** Minimiza o tempo médio de espera.
- **Desvantagem:** Difícil de prever o tempo de execução dos processos.

3. Round-Robin (RR)

- **Funcionamento:** Cada processo recebe um "quantum" de tempo para executar. Se não terminar, é colocado no final da fila.
- **Vantagem:** Justo e adequado para sistemas interativos.
- **Desvantagem:** Pode aumentar o tempo de resposta se o quantum for muito grande ou muito pequeno.

4. Priority Scheduling

- **Funcionamento:** Cada processo tem uma prioridade, e o processo com a prioridade mais alta é executado primeiro.
- **Vantagem:** Permite priorizar processos importantes.
- **Desvantagem:** Pode causar "starvation" (processos de baixa prioridade nunca são executados).

5. Multilevel Queue Scheduling

- **Funcionamento:** Divide os processos em várias filas com prioridades diferentes. Cada fila pode usar um algoritmo de escalonamento diferente.
- **Vantagem:** Flexível e adequado para sistemas com diferentes tipos de processos.
- **Desvantagem:** Complexo de implementar.

6. Multilevel Feedback Queue

- **Funcionamento:** Similar ao multilevel queue, mas permite que processos mudem de fila com base em seu comportamento.
- **Vantagem:** Adapta-se dinamicamente ao comportamento dos processos.
- **Desvantagem:** Ainda mais complexo que o multilevel queue.

Critérios de Avaliação para Seleção de Algoritmos

Ao escolher um algoritmo de escalonamento, os seguintes critérios devem ser considerados:

1. Utilização da CPU:

- O algoritmo deve maximizar o uso da CPU, evitando ociosidade.

2. Throughput:

- O número de processos concluídos por unidade de tempo deve ser maximizado.

3. Tempo de Resposta:

- O tempo que um processo leva para começar a ser executado deve ser minimizado.

4. Tempo de Espera:

- O tempo total que um processo passa esperando na fila de prontos deve ser minimizado.

5. Tempo de Retorno:

- O tempo total que um processo leva desde sua submissão até sua conclusão deve ser minimizado.

6. Justiça:

- Todos os processos devem ter uma chance justa de usar a CPU.

7. Previsibilidade:

- O comportamento do algoritmo deve ser previsível para garantir consistência.

Escalonamento de Threads

- Em sistemas que suportam threads no nível do kernel, o escalonamento é feito no nível das threads, não dos processos.

- **Benefícios:**

- Threads são mais leves que processos, permitindo maior concorrência.
- O escalonamento de threads pode ser mais eficiente, especialmente em sistemas com múltiplos núcleos de CPU.

- **Desafios:**

- O escalonador deve garantir que threads do mesmo processo sejam tratadas de forma justa.
- A sincronização entre threads pode ser complexa.

5.1 Conceitos básicos

Nesta seção, exploramos os conceitos fundamentais do escalonamento de CPU, que é essencial para o funcionamento eficiente de sistemas operacionais multiprogramados. Vamos detalhar cada tópico para facilitar o entendimento.

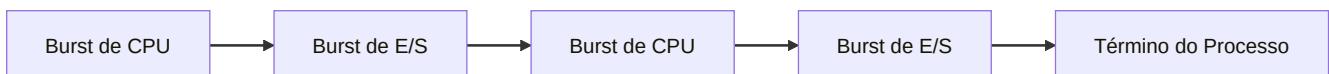
5.1.1 Ciclo de Burst CPU-E/S

O que é o Ciclo de Burst CPU-E/S?

- Os processos alternam entre dois estados principais:
 1. **Burst de CPU:** O processo está executando instruções na CPU.
 2. **Burst de E/S:** O processo está aguardando a conclusão de uma operação de entrada/saída (E/S).
- Esse ciclo se repete até que o processo termine.

Exemplo de Ciclo de Burst

1. O processo começa com um **burst de CPU**.
2. Em seguida, faz uma requisição de E/S e entra em um **burst de E/S**.
3. Após a conclusão da E/S, o processo retorna para outro **burst de CPU**.
4. Esse padrão continua até o término do processo.



Distribuição dos Tempos de Burst

- A maioria dos processos tem **bursts de CPU curtos**, enquanto uma minoria tem **bursts de CPU longos**.
- Isso é representado por uma curva exponencial ou hiperexponencial (veja a Figura 5.2).

Implicações para o Escalonamento

- Algoritmos de escalonamento devem ser escolhidos com base no comportamento dos processos (CPU-bound ou I/O-bound).

- **Processos I/O-bound:** Muitos bursts de CPU curtos.
- **Processos CPU-bound:** Poucos bursts de CPU longos.

5.1.2 Escalonador de CPU

O que é o Escalonador de CPU?

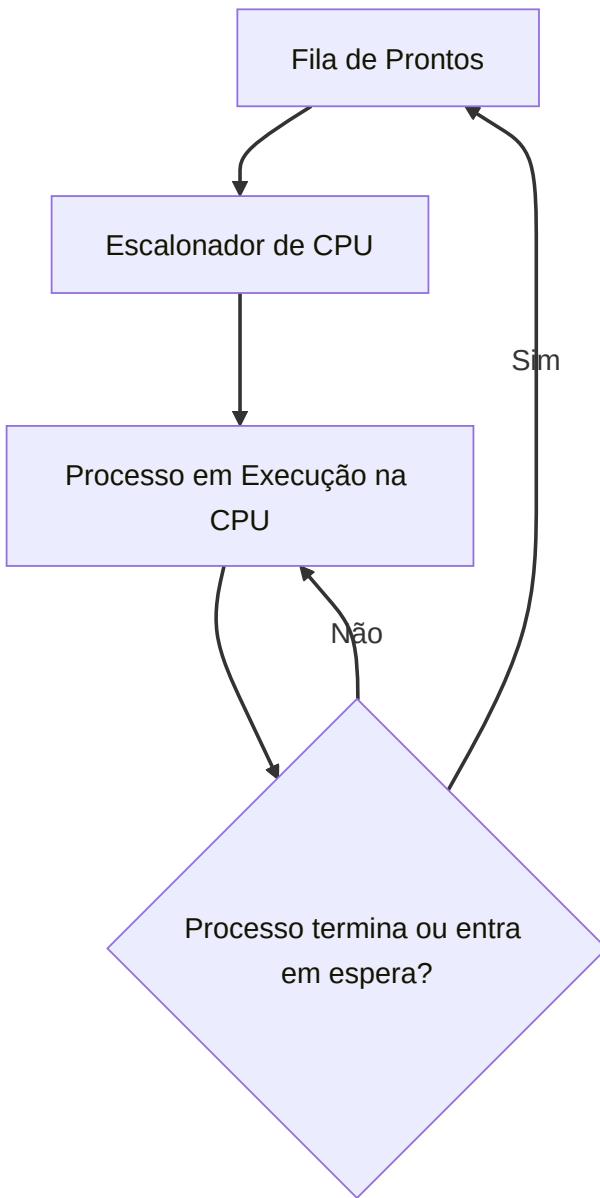
- O **escalonador de curto prazo** (ou escalonador de CPU) é responsável por selecionar qual processo na **fila de prontos** (ready queue) deve receber a CPU.

Funcionamento

1. Quando a CPU fica ociosa, o escalonador escolhe um processo da fila de prontos.
2. O processo selecionado é alocado para execução na CPU.

Estrutura da Fila de Prontos

- A fila de prontos pode ser implementada de várias formas:
 - **FIFO (First-In, First-Out):** O primeiro processo que entra é o primeiro a ser executado.
 - **Fila de Prioridade:** Processos com prioridade mais alta são executados primeiro.
 - **Lista Encadeada:** Permite flexibilidade na organização dos processos.



Registros na Fila de Prontos

- Cada entrada na fila de prontos é um **Bloco de Controle de Processo (PCB)**, que contém informações sobre o estado do processo.

5.1.3 Escalonamento Preemptivo vs. Não Preemptivo

Escalonamento Não Preemptivo

- A CPU é alocada a um processo até que ele termine ou entre em estado de espera.
- **Vantagem:** Simplicidade e menor custo de troca de contexto.

- **Desvantagem:** Pode causar atrasos para outros processos, especialmente em sistemas interativos.

Escalonamento Preemptivo

- A CPU pode ser retirada de um processo em execução e alocada a outro processo.
- **Cenários de Preempção:**
 1. Um processo passa de **executando** para **esperando** (ex.: requisição de E/S).
 2. Um processo passa de **executando** para **pronto** (ex.: interrupção).
 3. Um processo passa de **esperando** para **pronto** (ex.: término de E/S).
 4. Um processo termina.

Vantagens do Escalonamento Preemptivo

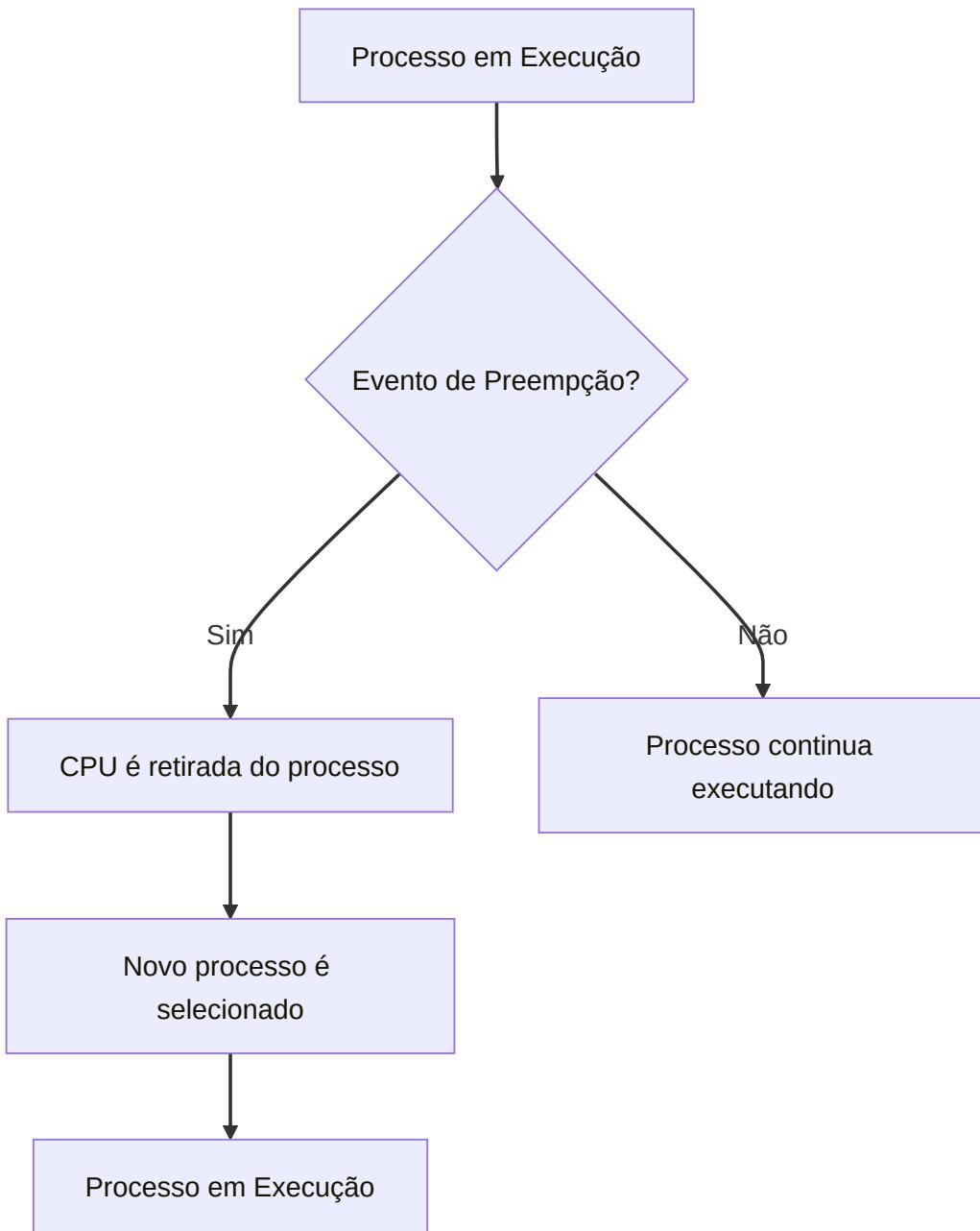
- Melhor tempo de resposta para processos interativos.
- Mais justo, pois evita que um processo monopolize a CPU.

Desafios do Escalonamento Preemptivo

- **Problemas de sincronização:** Dados compartilhados podem ficar inconsistentes se um processo for preemptado durante uma atualização.
- **Complexidade do kernel:** O kernel deve garantir que estruturas de dados internas não fiquem inconsistentes durante a preempção.

Exemplos de Sistemas

- **Windows 95 e versões posteriores:** Usam escalonamento preemptivo.
- **Mac OS X:** Também usa escalonamento preemptivo.
- **Windows 3.x e Macintosh antigos:** Usavam escalonamento cooperativo (não preemptivo).



5.1.4 Despachante

O que é o Despachante?

- O **despachante** é o módulo do sistema operacional responsável por:
 - 1. Trocar o contexto:** Salvar o estado do processo atual e restaurar o estado do próximo processo.
 - 2. Trocar para o modo usuário:** Retornar o controle ao programa do usuário.

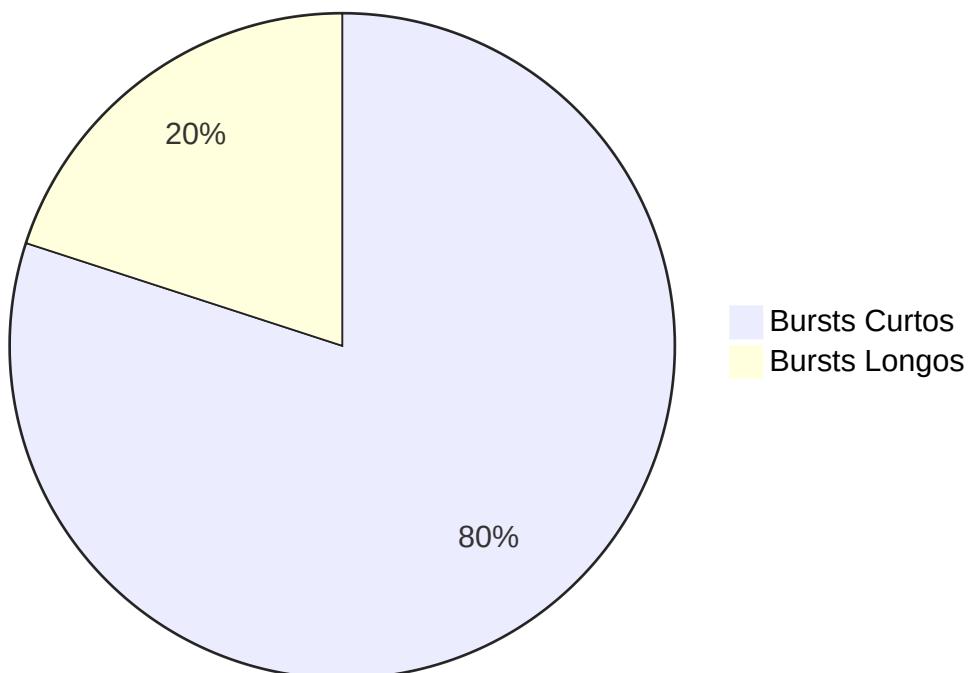
- 3. Reiniciar o programa:** Continuar a execução do processo a partir do ponto onde ele foi interrompido.



Latência de Despacho

- É o tempo que o despachante leva para:
 - Interromper um processo.
 - Iniciar a execução de outro processo.
- **Objetivo:** Minimizar a latência de despacho para melhorar a eficiência do sistema.

Distribuição dos Tempos



Importância do Despachante

- O despachante é chamado toda vez que ocorre uma troca de processo, portanto, deve ser rápido

e eficiente.

Resumo dos Conceitos

Tópico	Descrição
Ciclo de Burst CPU-E/S	Processos alternam entre execução na CPU e espera por E/S.
Escalonador de CPU	Seleciona o próximo processo a ser executado na fila de prontos.
Escalonamento Preemptivo	Permite interromper um processo em execução para alocar a CPU a outro.
Despachante	Responsável pela troca de contexto e reinício da execução do processo.

Exemplo Prático

Cenário de Escalonamento Preemptivo

1. O **Processo A** está em execução na CPU.
2. Uma **interrupção** ocorre (ex.: término de E/S do **Processo B**).
3. O escalonador decide preemptar o **Processo A** e alocar a CPU ao **Processo B**.
4. O **despachante** salva o estado do **Processo A** e restaura o estado do **Processo B**.
5. O **Processo B** começa a executar.

5.2 Critérios de Escalonamento

Nesta seção, discutimos os **critérios** usados para avaliar e comparar algoritmos de escalonamento de CPU. Esses critérios ajudam a determinar qual algoritmo é mais adequado para um determinado sistema ou cenário. Vamos detalhar cada um deles e explicar sua importância.

Critérios de Escalonamento

1. Utilização da CPU

- **Definição:** Percentual de tempo em que a CPU está ocupada executando processos.
- **Intervalo:** Varia de **0%** (CPU ociosa) a **100%** (CPU sempre ocupada).
- **Objetivo:** Maximizar a utilização da CPU.
- **Exemplo:**
 - Em um sistema pouco carregado, a utilização pode ser de **40%**.
 - Em um sistema muito utilizado, pode chegar a **90%**.

2. Throughput (Vazão)

- **Definição:** Número de processos concluídos por unidade de tempo.
- **Objetivo:** Maximizar o throughput.
- **Exemplos:**
 - Para processos longos: **1 processo por hora**.
 - Para transações curtas: **10 processos por segundo**.

3. Turnaround Time (Tempo de Retorno)

- **Definição:** Tempo total desde a submissão de um processo até o seu término.
- **Componentes:**
 1. Tempo de espera para entrar na memória.
 2. Tempo de espera na fila de prontos.
 3. Tempo de execução na CPU.

4. Tempo de E/S.

- **Objetivo:** Minimizar o turnaround time.
- **Exemplo:** Se um processo leva **10 segundos** para ser concluído, desde sua submissão até o término, seu turnaround time é **10 segundos**.

4. Tempo de Espera

- **Definição:** Tempo total que um processo passa esperando na fila de prontos.
- **Objetivo:** Minimizar o tempo de espera.
- **Observação:** O tempo de espera é influenciado apenas pelo algoritmo de escalonamento, não pelo tempo de execução ou E/S.

5. Tempo de Resposta

- **Definição:** Tempo desde a submissão de uma requisição até a primeira resposta ser produzida.
- **Objetivo:** Minimizar o tempo de resposta.
- **Importância:** Critério crucial para sistemas interativos (ex.: sistemas de tempo compartilhado).
- **Exemplo:** Em um sistema interativo, o tempo de resposta deve ser curto para garantir uma boa experiência do usuário.

Objetivos Gerais

- **Maximizar:**
 - Utilização da CPU.
 - Throughput.
- **Minimizar:**
 - Turnaround time.
 - Tempo de espera.
 - Tempo de resposta.

Otimização de Valores

- Na maioria dos casos, o foco é otimizar os **valores médios**.
- Em alguns cenários, é importante otimizar os **valores mínimo ou máximo**.
 - Exemplo: Reduzir o **tempo máximo de resposta** para garantir que todos os usuários recebam um bom atendimento.

Variância no Tempo de Resposta

- Para sistemas interativos, minimizar a **variância no tempo de resposta** pode ser mais importante do que minimizar o tempo de resposta médio.
- Um sistema com tempo de resposta **previsível** é preferível a um sistema mais rápido, porém com alta variabilidade.

Exemplo de Comparação de Algoritmos

Suponha que temos três processos com os seguintes tempos de burst de CPU:

Processo	Tempo de Burst (ms)
P1	24
P2	3
P3	3

Vamos comparar os tempos de espera médios para dois algoritmos de escalonamento: **FCFS** (First-Come, First-Served) e **SJF** (Shortest-Job-First).

FCFS

- Ordem de execução: P1 → P2 → P3.
- Tempos de espera:
 - P1: 0 ms.
 - P2: 24 ms.
 - P3: 27 ms.
- **Tempo de espera médio:** $(0 + 24 + 27) / 3 = 17$ ms.

SJF

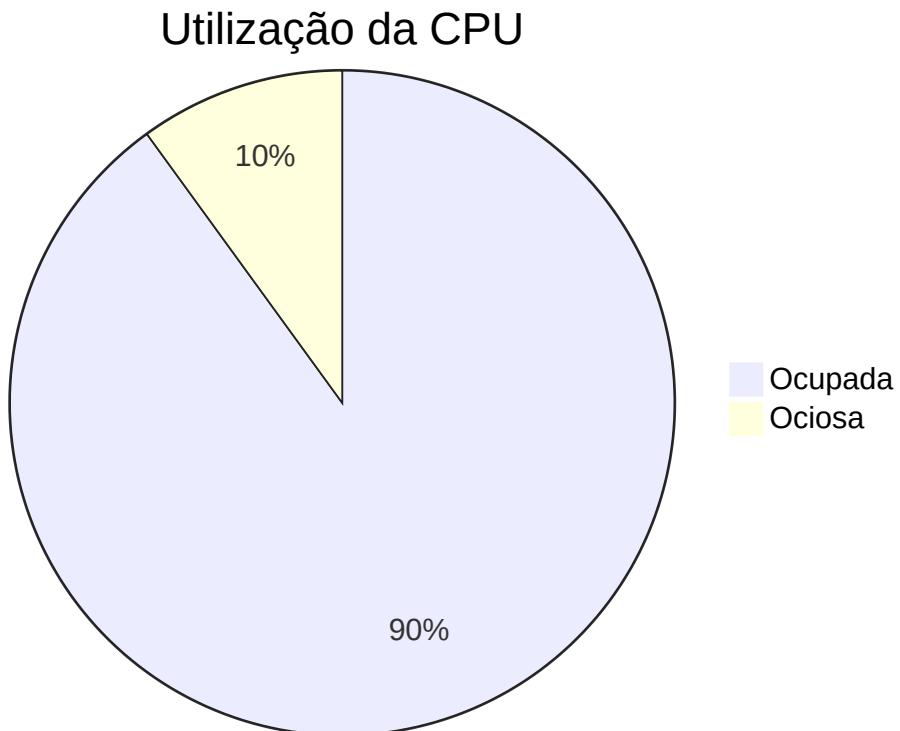
- Ordem de execução: P2 → P3 → P1.
- Tempos de espera:
 - P2: 0 ms.
 - P3: 3 ms.
 - P1: 6 ms.
- **Tempo de espera médio:** $(0 + 3 + 6) / 3 = 3 \text{ ms.}$

Conclusão

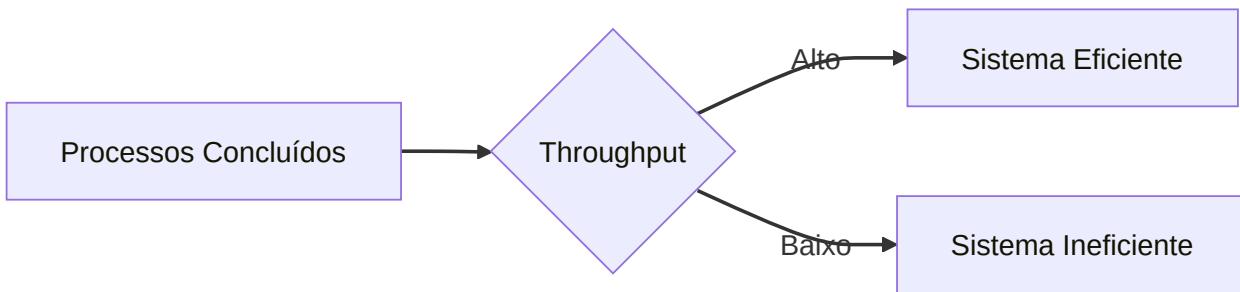
- O algoritmo SJF é melhor nesse caso, pois reduz o tempo de espera médio.

Diagramas para Ilustração

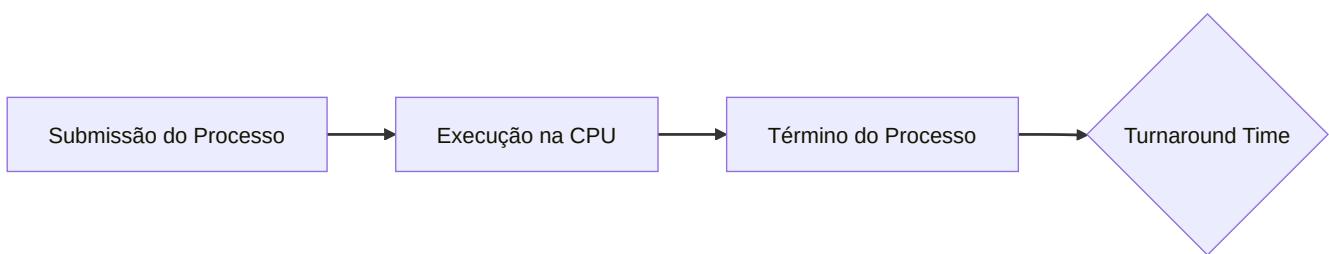
1. Diagrama de Utilização da CPU



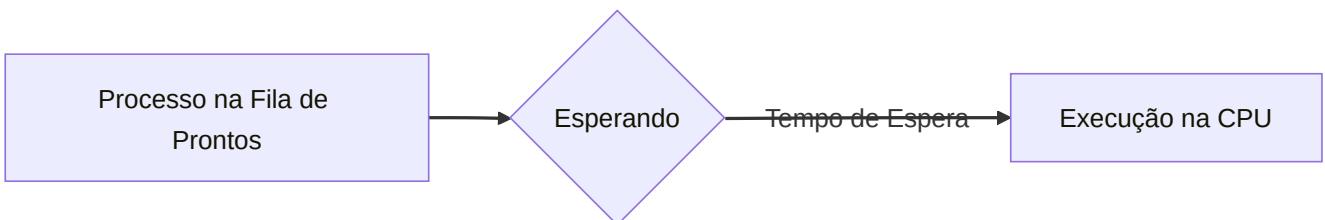
2. Diagrama de Throughput



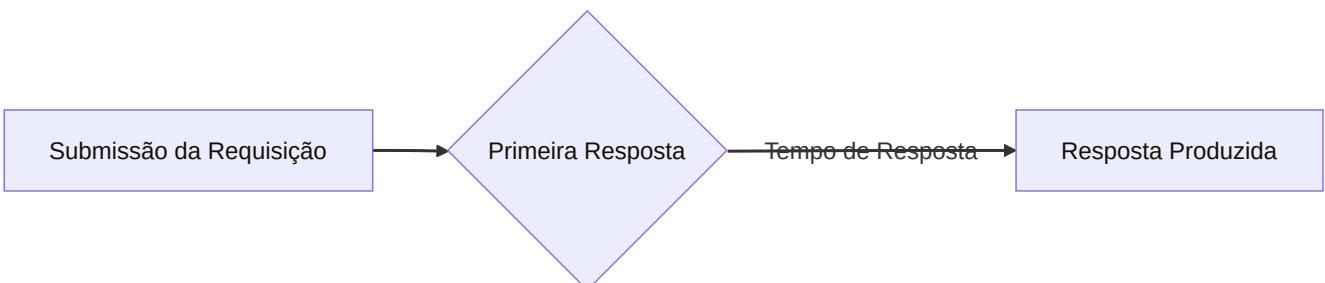
3. Diagrama de Turnaround Time



4. Diagrama de Tempo de Espera



5. Diagrama de Tempo de Resposta



Resumo dos Critérios

Critério	Definição	Objetivo
Utilização da CPU	Percentual de tempo em que a CPU está ocupada.	Maximizar
Throughput	Número de processos concluídos por unidade de tempo.	Maximizar
Turnaround Time	Tempo total desde a submissão até o término do processo.	Minimizar
Tempo de Espera	Tempo que um processo passa esperando na fila de prontos.	Minimizar
Tempo de Resposta	Tempo desde a submissão até a primeira resposta.	Minimizar

5.3 Algoritmos de Escalonamento

Nesta seção, discutimos os principais **algoritmos de escalonamento de CPU**, que são responsáveis por decidir qual processo na fila de prontos deve receber a CPU. Cada algoritmo tem suas próprias características, vantagens e desvantagens, e a escolha do algoritmo adequado depende das necessidades do sistema e dos processos.

5.3.1 Escalonamento First-Come, First-Served (FCFS)

Descrição

- O algoritmo **FCFS** (First-Come, First-Served) é o mais simples: o primeiro processo que chega à fila de prontos é o primeiro a ser executado.
- É implementado usando uma **fila FIFO** (First-In, First-Out).

Vantagens

- Simples de implementar e entender.
- Justo, pois os processos são atendidos na ordem de chegada.

Desvantagens

- **Tempo de espera médio** pode ser alto, especialmente se processos longos chegarem antes de processos curtos.
- Pode causar o **efeito comboio**: processos curtos ficam esperando por processos longos, o que reduz a eficiência do sistema.

Exemplo

- Processos: P1 (24 ms), P2 (3 ms), P3 (3 ms).
- Ordem de chegada: P1 → P2 → P3.
- **Tempo de espera médio:** $(0 + 24 + 27) / 3 = 17 \text{ ms}$.

5.3.2 Escalonamento Shortest-Job-First (SJF)

Descrição

- O algoritmo **SJF** (Shortest-Job-First) seleciona o processo com o menor **tempo de burst de CPU**.
- Pode ser **preemptivo** (chamado **SRTF** - Shortest Remaining Time First) ou **não preemptivo**.

Vantagens

- Minimiza o **tempo de espera médio**.
- Ideal para sistemas onde o tempo de burst de CPU é conhecido ou pode ser previsto.

Desvantagens

- Difícil de implementar, pois o tempo de burst de CPU nem sempre é conhecido.
- Pode causar **starvation** (processos longos podem nunca ser executados).

Exemplo

- Processos: P1 (6 ms), P2 (8 ms), P3 (7 ms), P4 (3 ms).
- Ordem de execução: P4 → P1 → P3 → P2.
- **Tempo de espera médio:** $(0 + 3 + 9 + 16) / 4 = 7 \text{ ms}$.

5.3.3 Escalonamento por Prioridade

Descrição

- Cada processo tem uma **prioridade**, e a CPU é alocada ao processo com a **maior prioridade**.
- Prioridades podem ser **internas** (baseadas em características do processo) ou **externas** (definidas pelo usuário).

Vantagens

- Permite priorizar processos importantes.
- Flexível, pois as prioridades podem ser ajustadas dinamicamente.

Desvantagens

- Pode causar **starvation** para processos de baixa prioridade.

- Requer mecanismos como **envelhecimento** (aging) para evitar starvation.

Exemplo

- Processos: P1 (10 ms, prioridade 3), P2 (1 ms, prioridade 1), P3 (2 ms, prioridade 4), P4 (1 ms, prioridade 5), P5 (5 ms, prioridade 2).
- Ordem de execução: P2 → P5 → P1 → P3 → P4.
- **Tempo de espera médio:** $(0 + 1 + 6 + 16 + 17) / 5 = 8 \text{ ms}$.

5.3.4 Escalonamento Round-Robin (RR)

Descrição

- O algoritmo **RR** (Round-Robin) aloca a CPU a cada processo por um **quantum de tempo** (ex.: 10 ms).
- Se o processo não terminar dentro do quantum, ele é preemptado e colocado no final da fila de prontos.

Vantagens

- Justo, pois todos os processos recebem uma fatia de tempo igual.
- Adequado para sistemas **interativos** e de **tempo compartilhado**.

Desvantagens

- **Tempo de espera médio** pode ser alto se o quantum for muito grande.
- **Troca de contexto** frequente pode reduzir a eficiência do sistema.

Exemplo

- Processos: P1 (24 ms), P2 (3 ms), P3 (3 ms).
- Quantum: 4 ms.
- **Tempo de espera médio:** $(6 + 4 + 7) / 3 = 5,66 \text{ ms}$.

5.3.5 Escalonamento Multilevel Queue

Descrição

- A fila de prontos é dividida em **várias filas**, cada uma com seu próprio algoritmo de escalonamento.
- Exemplo: fila de processos **interativos** (usando RR) e fila de processos **batch** (usando FCFS).

Vantagens

- Permite tratar diferentes tipos de processos de forma adequada.
- Flexível, pois cada fila pode ter um algoritmo diferente.

Desvantagens

- Complexo de implementar.
- Pode causar **starvation** se uma fila de alta prioridade monopolizar a CPU.

Exemplo

- Filas:
 1. Processos do sistema (prioridade máxima).
 2. Processos interativos (RR).
 3. Processos batch (FCFS).

5.3.6 Escalonamento Multilevel Feedback Queue

Descrição

- Similar ao **Multilevel Queue**, mas permite que processos **mudem de fila** com base em seu comportamento.
- Processos que usam muita CPU são movidos para filas de **menor prioridade**, enquanto processos que esperam muito são movidos para filas de **maior prioridade**.

Vantagens

- Combina as vantagens de vários algoritmos.

- Evita **starvation** por meio do envelhecimento.

Desvantagens

- Complexo de configurar e implementar.
- Requer ajuste cuidadoso dos parâmetros.

Exemplo

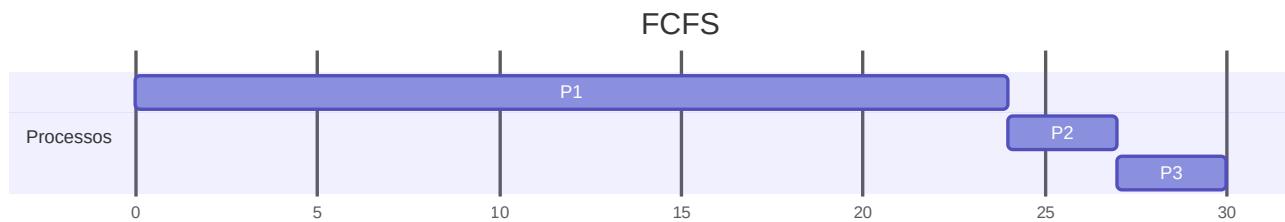
- Filas:
 1. Fila 0: Quantum de 8 ms (RR).
 2. Fila 1: Quantum de 16 ms (RR).
 3. Fila 2: FCFS.

Resumo dos Algoritmos

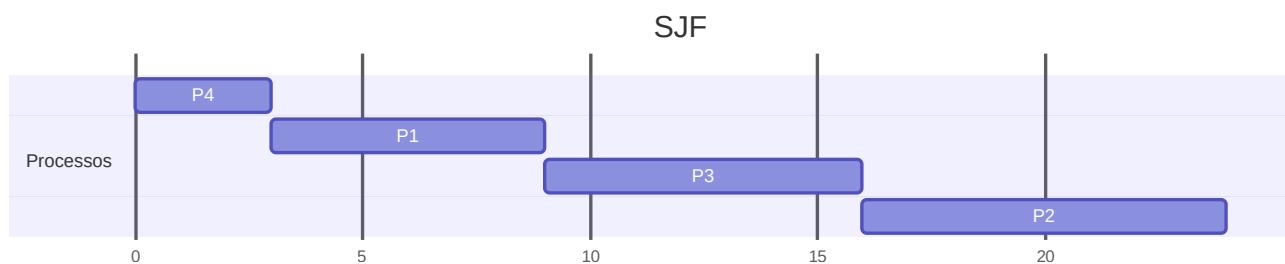
Algoritmo	Vantagens	Desvantagens	Melhor Uso
FCFS	Simples e justo	Tempo de espera médio alto	Sistemas com processos similares
SJF	Minimiza tempo de espera médio	Difícil de prever tempos de burst	Sistemas batch
Prioridade	Prioriza processos importantes	Pode causar starvation	Sistemas com prioridades definidas
Round-Robin (RR)	Justo e adequado para sistemas interativos	Troca de contexto frequente	Sistemas de tempo compilhado
Multilevel Queue	Trata diferentes tipos de processos	Complexo e pode causar starvation	Sistemas com múltiplas classes de processos
Multilevel Feedback Queue	Combina vantagens de vários algoritmos	Complexo de configurar	Sistemas que exigem flexibilidade

Diagramas para Ilustração

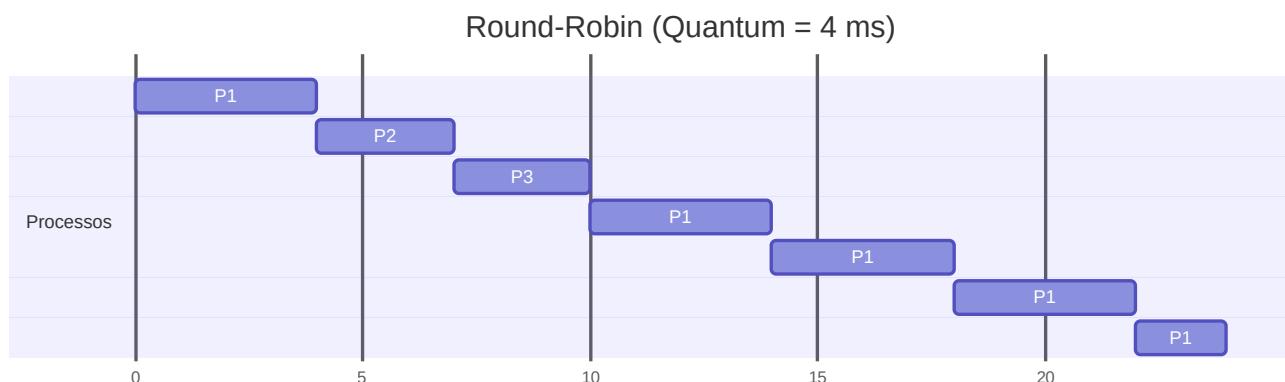
1. Diagrama de Gantt para FCFS



2. Diagrama de Gantt para SJF



3. Diagrama de Gantt para Round-Robin



5.4 Escalonamento de Threads

Nesta seção, exploramos como o **escalonamento de threads** é tratado em sistemas operacionais, com foco nas diferenças entre **threads no nível do usuário** e **threads no nível do kernel**. Também discutimos como a API **Pthreads** permite configurar o **escopo de disputa** para threads.

5.4.1 Escopo de Disputa

Threads no Nível do Usuário vs. Threads no Nível do Kernel

- **Threads no nível do usuário:**
 - Gerenciadas por uma **biblioteca de threads**.
 - O **kernel** não tem conhecimento direto dessas threads.
 - Para executar em uma CPU, as threads no nível do usuário precisam ser **mapeadas** para threads no nível do kernel, geralmente por meio de **Processos Leves (LWPs)**.
- **Threads no nível do kernel:**
 - Gerenciadas diretamente pelo **sistema operacional**.
 - São escalonadas pelo **escalonador de CPU** do sistema.

Escopo de Disputa

- **Process Contention Scope (PCS):**
 - A disputa pela CPU ocorre entre **threads do mesmo processo**.
 - Usado em sistemas que implementam os modelos **muitos para um** ou **muitos para muitos**.
 - A biblioteca de threads escalona as threads no nível do usuário para executar em **LWPs disponíveis**.
- **System Contention Scope (SCS):**
 - A disputa pela CPU ocorre entre **todas as threads do sistema**.
 - Usado em sistemas que implementam o modelo **um para um** (ex.: Windows XP, Solaris, Linux).

Prioridades no PCS

- As threads no nível do usuário são escalonadas com base em **prioridades** definidas pelo programador.
- O escalonador interrompe uma thread em execução para dar lugar a uma thread de **prioridade mais alta**.
- Não há garantia de **fatia de tempo** (time-slicing) entre threads de mesma prioridade.

5.4.2 Escalonamento Pthread

API Pthreads para Escopo de Disputa

A API Pthreads permite especificar o **escopo de disputa** durante a criação de threads. Os valores possíveis são:

- **PTHREAD_SCOPE_PROCESS**:
 - Usa o **PCS** (Process Contention Scope).
 - Threads no nível do usuário são escalonadas para **LWPs disponíveis**.
- **PTHREAD_SCOPE_SYSTEM**:
 - Usa o **SCS** (System Contention Scope).
 - Cada thread no nível do usuário é associada a um **LWP**, efetivamente mapeando threads no modelo **um para um**.

Funções Pthreads

- **pthread_attr_setscope**:
 - Define o escopo de disputa para uma thread.
 - Sintaxe:


```
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
```
 - Parâmetros:
 - **attr**: Ponteiro para os atributos da thread.
 - **scope**: Valor do escopo de disputa (**PTHREAD_SCOPE_PROCESS** ou **PTHREAD_SCOPE_SYSTEM**).
- **pthread_attr_getscope**:

- Obtém o escopo de disputa atual de uma thread.

- Sintaxe:

```
int pthread_attr_getscope(pthread_attr_t *attr, int *scope);
```

- Parâmetros:

- attr: Ponteiro para os atributos da thread.
- scope: Ponteiro para armazenar o valor do escopo de disputa.

Exemplo de Uso

Aqui está um exemplo de código que define o escopo de disputa como PCS e cria cinco threads:

```
#include <pthread.h>
#include <stdio.h>

void* thread_function(void* arg) {
    printf("Thread %ld executando\n", (long)arg);
    return NULL;
}

int main() {
    pthread_t threads[5];
    pthread_attr_t attr;
    int scope;

    // Inicializa os atributos da thread
    pthread_attr_init(&attr);

    // Define o escopo de disputa como PCS
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS);

    // Obtém o escopo de disputa atual
    pthread_attr_getscope(&attr, &scope);
    if (scope == PTHREAD_SCOPE_PROCESS)
        printf("Escopo de disputa: PCS\n");
    else
        printf("Escopo de disputa: SCS\n");

    // Cria cinco threads
    for (long i = 0; i < 5; i++) {
```

```
    pthread_create(&threads[i], &attr, thread_function, (void*)i);
}

// Aguarda as threads terminarem
for (int i = 0; i < 5; i++) {
    pthread_join(threads[i], NULL);
}

// Destroi os atributos da thread
pthread_attr_destroy(&attr);

return 0;
}
```

Explicação do Código

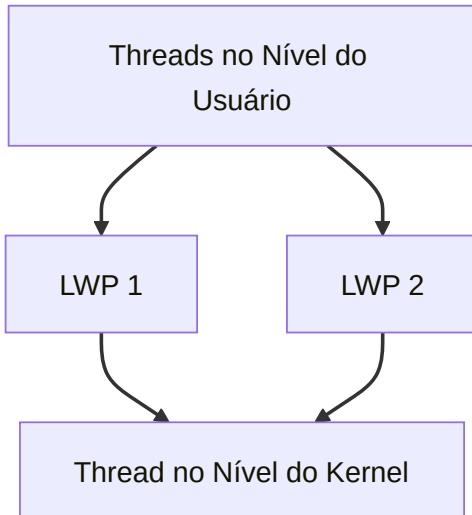
1. **pthread_attr_init**: Inicializa os atributos da thread.
2. **pthread_attr_setscope**: Define o escopo de disputa como PCS.
3. **pthread_attr_getscope**: Obtém o escopo de disputa atual para verificação.
4. **pthread_create**: Cria cinco threads que executam a função `thread_function`.
5. **pthread_join**: Aguarda todas as threads terminarem.
6. **pthread_attr_destroy**: Destroi os atributos da thread.

Resumo

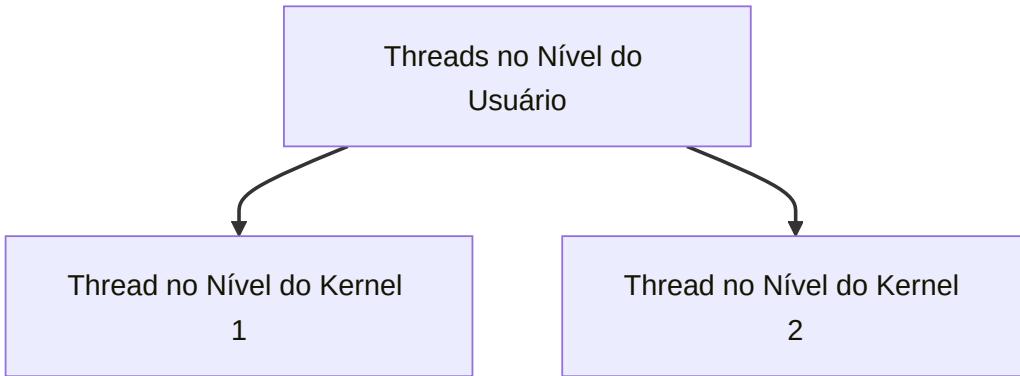
Conceito	Descrição
Threads no Nível do Usuário	Gerenciadas por bibliotecas de threads; mapeadas para LWPs.
Threads no Nível do Kernel	Gerenciadas diretamente pelo sistema operacional.
PCS (Process Contention Scope)	Disputa pela CPU entre threads do mesmo processo.
SCS (System Contention Scope)	Disputa pela CPU entre todas as threads do sistema.
PTHREAD_SCOPE_PROCESS	Usa PCS; threads no nível do usuário são escalonadas para LWPs disponíveis.
PTHREAD_SCOPE_SYSTEM	Usa SCS; cada thread no nível do usuário é associada a um LWP.

Diagramas para Ilustração

1. Modelo Muitos para Um (PCS)



2. Modelo Um para Um (SCS)



5.5 Escalonamento em Múltiplos Processadores

Imagine que você está jogando Minecraft em um servidor com vários amigos. Cada amigo é como um **processador**, e as tarefas que vocês fazem no jogo (como minerar, construir ou lutar) são os **processos**. Agora, vamos entender como o jogo (sistema operacional) decide quem faz o quê e como isso funciona quando há vários "amigos" (processadores) disponíveis.

5.5.1 Técnicas de Escalonamento com Multiprocessadores

1. Multiprocessamento Assimétrico (ASMP):

- Imagine que um dos seus amigos é o **chefe do servidor**. Ele decide quem faz o quê (escalona as tarefas), enquanto os outros só jogam (executam tarefas).
- **Vantagem:** Simples, pois só o chefe toma decisões.
- **Desvantagem:** Se o chefe ficar ocupado, todo o servidor pode ficar lento.

2. Multiprocessamento Simétrico (SMP):

- Aqui, todos os amigos são **chefes** e decidem o que fazer. Eles podem compartilhar uma lista de tarefas ou cada um ter sua própria lista.
- **Desafio:** Se dois amigos pegarem a mesma tarefa, pode dar confusão. Então, é preciso sincronização.
- **Exemplo:** Sistemas como Windows, Linux e macOS usam SMP.

5.5.2 Afinidade de Processador

- Imagine que você está minerando em uma caverna e já decorou onde estão os minérios (dados na **cache**). Se você for para outra caverna (outro processador), vai perder tempo reaprendendo onde estão os minérios.
- **Afinidade de Processador:** O sistema tenta manter você na mesma caverna (processador) para evitar perda de tempo.
 - **Afinidade Flexível:** O sistema tenta, mas não garante.
 - **Afinidade Rígida:** Você pode dizer "não quero sair daqui!".

- **NUMA (Acesso Não Uniforme à Memória):** Em servidores grandes, algumas cavernas são mais rápidas de acessar do que outras, dependendo da localização.

5.5.3 Balanceamento de Carga

- Se um amigo está sobrecarregado (minerando e construindo ao mesmo tempo), enquanto outro está só olhando a paisagem, o sistema tenta **equilibrar** as tarefas.
 - **Migração Push:** O sistema redistribui as tarefas ativamente.
 - **Migração Pull:** O amigo ocioso pega uma tarefa de quem está ocupado.
- **Problema:** Se você mudar de caverna (processador), perde o benefício de já conhecer o local (cache).

5.5.4 Processadores Multicore

- Agora imagine que cada amigo tem **várias mãos** (núcleos) para fazer tarefas ao mesmo tempo.
 - **Multithreading:** Cada mão pode fazer uma tarefa diferente.
 - **Coarse-Grained:** Troca de tarefas só quando algo demora muito (como esperar um bloco cair).
 - **Fine-Grained:** Troca de tarefas rapidamente, a cada pequena ação.
- **Exemplo:** Um processador com 8 núcleos e 4 threads por núcleo parece ter 32 "mãos" para o sistema operacional.

5.5.5 Virtualização e Escalonamento

- Imagine que você está jogando em um **servidor virtual** (como um Minecraft dentro de outro Minecraft). O servidor real tem que dividir seus recursos entre vários jogos virtuais.
 - **Problema:** Se o servidor real estiver ocupado, seu jogo virtual pode ficar lento, mesmo que você tenha configurado tudo certinho.
 - **Impacto:** Sistemas de tempo real (como mods de redstone) podem falhar porque o tempo não é preciso.

Mindmap

Escalonamento em Múltiplos Processadores

- **Técnicas de Escalonamento**
 - Assimétrico (ASMP)
 - 1 chefe (processador mestre)
 - Outros só executam tarefas
 - Simétrico (SMP)
 - Todos são chefes
 - Fila de tarefas comum ou privada
 - Sincronização necessária
- **Afinidade de Processador**
 - Manter processo no mesmo processador
 - Benefícios: aproveitar a cache
 - Tipos
 - Afinidade Flexível
 - Afinidade Rígida
 - NUMA (Acesso Não Uniforme à Memória)
- **Balanceamento de Carga**
 - Distribuir tarefas uniformemente
 - Técnicas
 - Migração Push
 - Migração Pull
 - Conflito com afinidade de processador
- **Processadores Multicore**
 - Vários núcleos em um chip
 - Multithreading
 - Coarse-Grained
 - Fine-Grained
 - Dois níveis de escalonamento
 - Escalonamento de threads de software
 - Escalonamento de threads de hardware
- **Virtualização e Escalonamento**
 - CPUs virtuais para máquinas virtuais
 - Impacto no desempenho
 - Desafios para sistemas de tempo real

5.6 Exemplos de Sistema Operacional

Vamos explorar como sistemas operacionais modernos, como **Windows 10/11**, **Linux (com foco no kernel 5.x ou superior)** e **macOS**, lidam com o escalonamento de tarefas. Para facilitar o entendimento, vamos usar **Minecraft** como analogia. Imagine que o sistema operacional é o **servidor de Minecraft**, e as tarefas (processos ou threads) são os **jogadores** que precisam realizar atividades no jogo.

5.6.1 Escalonamento no Windows 10/11

O Windows 10/11 usa um sistema de **prioridades dinâmicas** e **escalonamento preemptivo** para gerenciar tarefas. Ele é uma evolução do Windows XP, com melhorias para suportar hardware moderno, como processadores multicore e sistemas NUMA.

Características Principais:

1. Prioridades Dinâmicas:

- As tarefas são organizadas em **32 níveis de prioridade** (0 a 31).
- Tarefas de **tempo real** (16-31) têm prioridade máxima e são executadas imediatamente.
- Tarefas comuns (1-15) têm prioridades ajustadas dinamicamente:
 - Tarefas interativas (como abrir um aplicativo) ganham prioridade.
 - Tarefas que usam muita CPU (como renderização) perdem prioridade.

2. Balanceamento de Carga:

- O Windows distribui tarefas entre **núcleos de processadores** para evitar sobrecarga.
- Se um núcleo estiver ocioso, ele "puxa" tarefas de outros núcleos ocupados.

3. Suporte a NUMA:

- Em sistemas com múltiplos processadores e memória não uniforme (NUMA), o Windows tenta manter as tarefas próximas à memória que estão usando, para melhorar o desempenho.

4. Modo de Economia de Energia:

- O Windows ajusta o escalonamento para reduzir o consumo de energia em dispositivos móveis, priorizando tarefas em núcleos de baixo consumo.

Como Funciona no Minecraft:

- Se um jogador estiver construindo algo complexo (uso intenso de CPU), ele pode perder prioridade para outro jogador que está interagindo com o ambiente (abrir baús, clicar em blocos).
- O servidor (escalonador) garante que todos os núcleos do processador sejam usados de forma equilibrada.

5.6.2 Escalonamento no Linux (Kernel 5.x ou superior)

O Linux moderno usa o **escalonador CFS** (Completely Fair Scheduler), que é altamente eficiente e justo. Ele foi projetado para sistemas multicore e grandes cargas de trabalho.

Características Principais:

1. CFS (Completely Fair Scheduler):

- O CFS usa um conceito de **tempo virtual** para garantir que todas as tarefas recebam uma fatia justa da CPU.
- Tarefas com **prioridades mais altas** recebem mais tempo de CPU, mas todas são atendidas de forma equilibrada.

2. Prioridades:

- As tarefas são organizadas em dois grupos:
 - **Tempo Real (0-99):** Prioridade máxima, executadas imediatamente.
 - **Tarefas Comuns (100-139):** Prioridades ajustadas dinamicamente com base no valor **nice** (quanto maior o valor nice, menor a prioridade).

3. Balanceamento de Carga:

- O Linux distribui tarefas entre núcleos de processadores e tenta manter a **afinidade de processador** (evitar migração desnecessária de tarefas entre núcleos).
- Se um núcleo estiver ocioso, ele "puxa" tarefas de outros núcleos.

4. Suporte a NUMA:

- O Linux é altamente otimizado para sistemas NUMA, garantindo que as tarefas sejam executadas próximas à memória que estão usando.

5. Escalonamento em Tempo Real:

- O Linux suporta tarefas de tempo real com **prioridades estáticas**, garantindo que elas sejam executadas imediatamente.

Como Funciona no Minecraft:

- O servidor (escalonador) garante que todos os jogadores tenham uma fatia justa do tempo de CPU.
- Se um jogador estiver minerando (uso intenso de CPU), ele não dominará o servidor, permitindo que outros jogadores interajam com o ambiente.

5.6.3 Escalonamento no macOS

O macOS usa um sistema de escalonamento baseado em **prioridades dinâmicas e qualidade de serviço (QoS)**, projetado para oferecer uma experiência suave e responsiva.

Características Principais:

1. Qualidade de Serviço (QoS):

- As tarefas são classificadas em **níveis de QoS**, que determinam sua prioridade:
 - **User Interactive (UI)**: Prioridade máxima para tarefas interativas (como animações de interface).
 - **User Initiated**: Para tarefas iniciadas pelo usuário (como abrir um aplicativo).
 - **Utility**: Para tarefas em segundo plano (como downloads).
 - **Background**: Para tarefas de baixa prioridade (como indexação de arquivos).

2. Prioridades Dinâmicas:

- O macOS ajusta as prioridades das tarefas com base no comportamento:
 - Tarefas interativas ganham prioridade.
 - Tarefas que usam muita CPU perdem prioridade.

3. Grand Central Dispatch (GCD):

- O GCD é uma tecnologia que facilita a execução de tarefas em paralelo, distribuindo-as entre núcleos de processadores.

4. Suporte a NUMA:

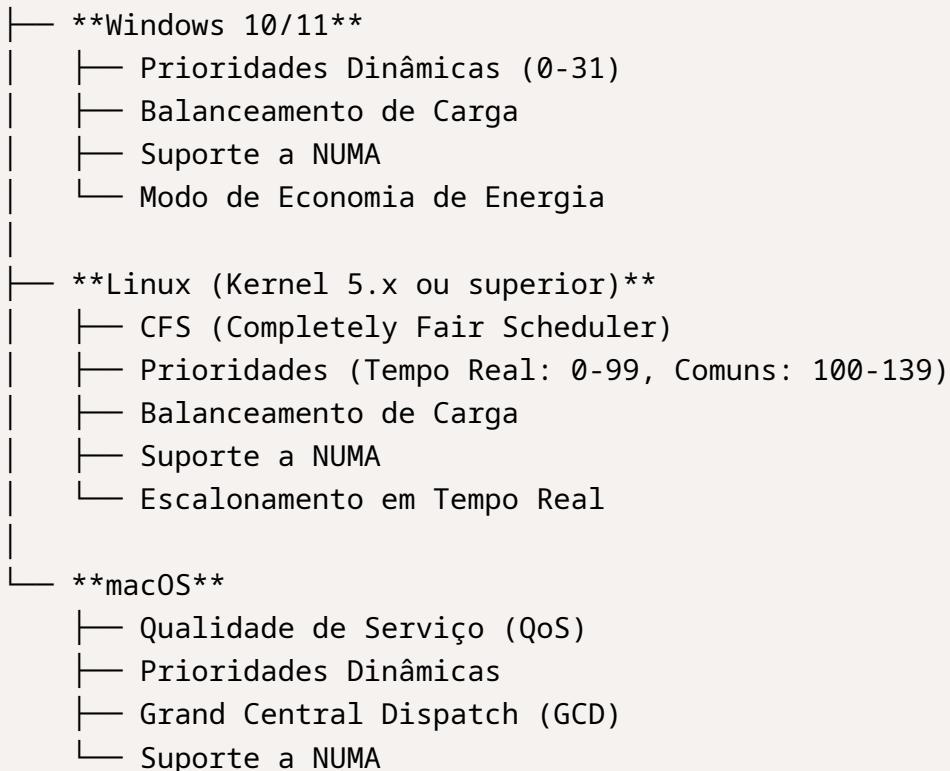
- O macOS é otimizado para sistemas com múltiplos processadores e memória não uniforme (NUMA).

Como Funciona no Minecraft:

- Se um jogador estiver interagindo com a interface do jogo (como abrir um menu), ele terá prioridade máxima.
- Tarefas em segundo plano (como carregar chunks do mundo) são executadas com prioridade mais baixa, sem afetar a experiência do jogador.

Mindmap

Exemplos de Sistemas Operacionais Modernos



5.8 Avaliação de Algoritmos de Escalonamento

Escolher o algoritmo de escalonamento de CPU ideal para um sistema específico é uma tarefa complexa, pois envolve a análise de diversos fatores, como utilização da CPU, tempo de resposta, throughput e justiça. Nesta seção, exploramos os métodos de avaliação de algoritmos de escalonamento, desde modelos determinísticos até simulações e implementações reais.

➊ Escolha do Método de Avaliação:

- Use **modelagem determinística** para análises rápidas e cenários controlados.
- Use **modelos de enfileiramento** para análises teóricas e tendências gerais.
- Use **simulações** para cenários complexos e realistas.
- A **implementação real** é a mais precisa, mas também a mais cara e complexa.

5.8.1 Modelagem Determinística

A modelagem determinística é uma técnica analítica que utiliza uma carga de trabalho específica para avaliar o desempenho de diferentes algoritmos de escalonamento. Ela é útil para comparar algoritmos em cenários controlados.

Exemplo Prático:

Considere a seguinte carga de trabalho, onde todos os processos chegam no tempo 0:

Processo	Tempo de Burst (ms)
P1	10
P2	29
P3	3
P4	7
P5	12

Avaliamos três algoritmos: **FCFS** (First-Come, First-Served), **SJF** (Shortest Job First) e **RR** (Round Robin com quantum = 10 ms).

1. FCFS:

- Ordem de execução: P1 → P2 → P3 → P4 → P5.
- Tempos de espera: P1 (0 ms), P2 (10 ms), P3 (39 ms), P4 (42 ms), P5 (49 ms).
- Tempo de espera médio: $\frac{0+10+39+42+49}{5} = 28$ ms.

2. SJF (não preemptivo):

- Ordem de execução: P3 → P4 → P1 → P5 → P2.
- Tempos de espera: P1 (10 ms), P2 (32 ms), P3 (0 ms), P4 (3 ms), P5 (20 ms).
- Tempo de espera médio: $\frac{10+32+0+3+20}{5} = 13$ ms.

3. RR (quantum = 10 ms):

- Ordem de execução: P1 → P2 → P3 → P4 → P5 → P2 → P5.
- Tempos de espera: P1 (0 ms), P2 (32 ms), P3 (20 ms), P4 (23 ms), P5 (40 ms).
- Tempo de espera médio: $\frac{0+32+20+23+40}{5} = 23$ ms.

i Trade-offs:

- Algoritmos como **SJF** minimizam o tempo de espera, mas podem causar starvation.
- Algoritmos como **RR** são justos, mas podem aumentar o tempo de resposta.

Conclusão:

- O **SJF** fornece o menor tempo de espera médio (13 ms).
- O **RR** oferece um equilíbrio entre tempo de resposta e justiça.
- O **FCFS** é o menos eficiente nesse cenário.

i

- A modelagem determinística é simples e rápida, mas só se aplica a cargas de trabalho específicas.

- Ela é útil para ilustrar tendências e comparar algoritmos em cenários controlados.

5.8.2 Modelos de Enfileiramento

Os modelos de enfileiramento são usados para analisar sistemas onde os processos chegam e são atendidos de acordo com distribuições de probabilidade. Eles são úteis para calcular métricas como utilização da CPU, tempo médio de espera e tamanho médio da fila.

Fórmula de Little:

A fórmula de Little relaciona o tamanho médio da fila (n), o tempo médio de espera (W) e a taxa de chegada de processos (λ):

$$n = \lambda \times W$$

Exemplo:

- Se $\lambda = 7$ processos/segundo e $n = 14$ processos na fila, então:

$$W = \frac{n}{\lambda} = \frac{14}{7} = 2 \text{ segundos.}$$

Limitações:

- Os modelos de enfileiramento assumem distribuições matemáticas simplificadas, que podem não refletir cenários reais.
- Eles são mais úteis para análises teóricas do que para previsões precisas.

5.8.3 Simulações

As simulações são usadas para avaliar algoritmos de escalonamento em cenários mais realistas.

Elas envolvem a criação de um modelo computacional do sistema, onde os processos são gerados de acordo com distribuições de probabilidade ou fitas de rastreamento (trace tapes).

Tipos de Simulações:

1. Simulação Controlada por Distribuição:

- Usa geradores de números aleatórios para criar processos com base em distribuições (exponencial, Poisson, etc.).
- Útil para cenários genéricos, mas pode não capturar correlações entre eventos.

2. Simulação com Fitas de Rastreamento:

- Usa dados reais coletados de um sistema em operação.
- Fornece resultados precisos para cenários específicos.

Vantagens:

- Permite a avaliação de algoritmos em cenários complexos e realistas.
- Pode ser usada para comparar múltiplos algoritmos com as mesmas entradas.

Desvantagens:

- Pode ser computacionalmente cara e demorada.
- Requer grande quantidade de dados e espaço de armazenamento.

5.8.4 Implementação

A implementação real de um algoritmo de escalonamento em um sistema operacional é a forma mais precisa de avaliar seu desempenho. No entanto, essa abordagem tem desafios significativos.

Desafios:

1. Custo:

- Modificar o sistema operacional para incluir um novo algoritmo é caro e complexo.
- Requer testes extensivos para garantir que o sistema continue estável.

2. Reação dos Usuários:

- Usuários podem ajustar seu comportamento para se beneficiar do novo algoritmo (por exemplo, dividindo processos longos em menores).
- Isso pode distorcer os resultados da avaliação.

3. Ambiente Dinâmico:

- O desempenho do algoritmo pode variar conforme o ambiente de trabalho muda.

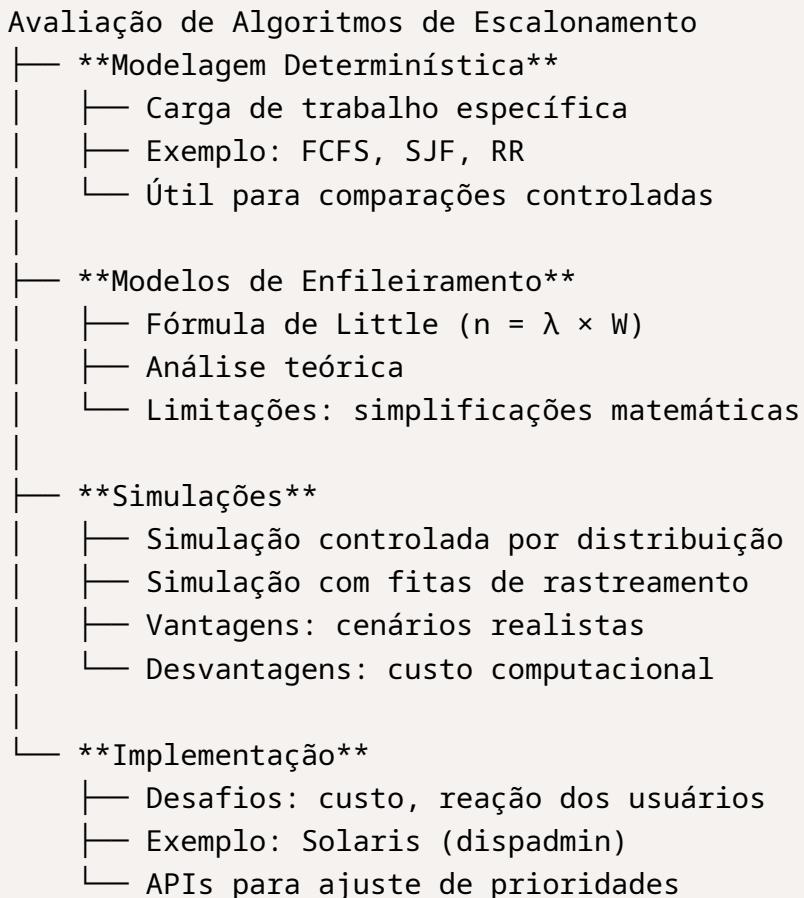
Exemplo:

- No Solaris, o comando `dispadmin` permite ajustar os parâmetros de escalonamento.
- APIs como as do Java, POSIX e Win32 permitem modificar prioridades de threads, mas isso pode não ser eficaz em cenários genéricos.

i Simulações vs. Implementação:

- Simulações são úteis para testes preliminares, mas a implementação real é necessária para validação final.

Mapa mental



Conclusão

A escolha do algoritmo de escalonamento ideal depende dos critérios de desempenho desejados (tempo de resposta, throughput, justiça) e do ambiente de trabalho. A combinação de modelagem, simulação e implementação real é essencial para tomar decisões informadas.

i Adaptação ao Ambiente:

- Algoritmos de escalonamento devem ser ajustados conforme o ambiente de trabalho

muda.

- Sistemas operacionais modernos permitem ajustes dinâmicos (por exemplo, prioridades de threads).

Exercícios Práticos

Exercício 5.1

Pergunta: Um algoritmo de escalonamento de CPU determina uma ordem para a execução de seus processos escalonados. Com n processos a serem escalonados em um processador, quantos escalonamentos diferentes são possíveis? Mostre uma fórmula em termos de n .

Resposta: O número de escalonamentos possíveis é dado pelo número de permutações dos n processos. Isso ocorre porque cada ordem de execução dos processos é uma permutação única. A fórmula para o número de permutações de n elementos é:

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 1$$

Explicação:

- Se houver 3 processos ($n = 3$), os escalonamentos possíveis são $3! = 6$: (P1, P2, P3), (P1, P3, P2), (P2, P1, P3), (P2, P3, P1), (P3, P1, P2), (P3, P2, P1).
- Esse conceito é importante porque mostra que, à medida que o número de processos aumenta, o número de possíveis escalonamentos cresce rapidamente (fatorialmente).

Exercício 5.2

Pergunta: Explique a diferença entre escalonamento preemptivo e não preemptivo.

Resposta:

- **Escalonamento preemptivo:** O sistema operacional pode interromper um processo em execução e substituí-lo por outro, mesmo que o processo atual não tenha terminado. Isso permite maior flexibilidade e melhor uso da CPU, especialmente em sistemas com múltiplos processos.
- **Escalonamento não preemptivo:** Uma vez que um processo começa a executar, ele só é interrompido quando termina ou bloqueia (por exemplo, para E/S). Isso pode levar a tempos de resposta mais longos, especialmente se processos longos estiverem em execução.

Explicação:

- O escalonamento preemptivo é comum em sistemas modernos, pois permite priorizar processos mais importantes ou curtos.
- O escalonamento não preemptivo é mais simples, mas pode causar problemas como o "efeito

"convoy", onde processos curtos ficam esperando processos longos terminarem.

Exercício 5.3

Pergunta: Suponha que os processos a seguir cheguem para execução nos tempos indicados. Cada processo será executado por um período listado. Use escalonamento não preemptivo e responda as perguntas.

Processo	Tempo de chegada	Tempo de burst
P1	0,0	8
P2	0,4	4
P3	1,0	1

- a. Qual é o tempo de turnaround médio para estes processos com o algoritmo de escalonamento FCFS? b. Qual é o tempo de turnaround médio para estes processos com o algoritmo de escalonamento SJF? c. Calcule o tempo de turnaround médio se a CPU ficar ociosa por uma unidade e depois usar SJF.

Resposta: a. FCFS (First-Come, First-Served):

- Ordem de execução: P1 (0-8), P2 (8-12), P3 (12-13).
- Turnaround: $P1 = 8$, $P2 = 12 - 0,4 = 11,6$, $P3 = 13 - 1,0 = 12$.
- Média: $(8 + 11,6 + 12)/3 = 10,53$.

b. SJF (Shortest Job First):

- Ordem de execução: P1 (0-8), P3 (8-9), P2 (9-13).
- Turnaround: $P1 = 8$, $P2 = 13 - 0,4 = 12,6$, $P3 = 9 - 1,0 = 8$.
- Média: $(8 + 12,6 + 8)/3 = 9,53$.

c. SJF com CPU ociosa:

- CPU fica ociosa até $t = 1$.
- Ordem de execução: P3 (1-2), P2 (2-6), P1 (6-14).

- Turnaround: $P1 = 14 - 0 = 14$, $P2 = 6 - 0,4 = 5,6$, $P3 = 2 - 1,0 = 1$.
- Média: $(14 + 5,6 + 1)/3 = 6,87$.

Explicação:

- O FCFS é simples, mas pode não ser eficiente.
- O SJF melhora o tempo de turnaround médio, mas depende do conhecimento prévio dos tempos de burst.
- A ociosidade inicial pode melhorar ainda mais o desempenho, mas aumenta o tempo de espera dos processos que chegam antes.

Exercício 5.4

Pergunta: Qual é a vantagem de haver diferentes tamanhos de quantum de tempo em diferentes níveis de um sistema de enfileiramento multilevel queue?

Resposta: A vantagem é permitir que processos curtos sejam executados rapidamente (com quanta menores) e processos longos recebam mais tempo de CPU (com quanta maiores). Isso melhora o tempo de resposta para processos interativos e a eficiência para processos de longa duração.

Explicação:

- Filas com quanta menores são ideais para processos interativos (como editores de texto).
- Filas com quanta maiores são ideais para processos de longa duração (como compiladores).
- Isso equilibra justiça e eficiência.

Exercício 5.5

Pergunta: Que relação existe entre os seguintes pares de conjuntos de algoritmos? a. Prioridade e SJF b. Multilevel feedback queues e FCFS c. Prioridade e FCFS d. RR e SJF

Resposta: a. **Prioridade e SJF:** O SJF pode ser visto como um caso especial de prioridade, onde a prioridade é inversamente proporcional ao tempo de burst. b. **Multilevel feedback queues e FCFS:** O FCFS pode ser uma das filas em um sistema multilevel feedback queue. c. **Prioridade e FCFS:** O FCFS pode ser implementado como um caso especial de prioridade, onde todos os processos têm a mesma prioridade. d. **RR e SJF:** Não há relação direta, pois o RR é baseado em tempo, enquanto o SJF é baseado no tempo de burst.

Explicação:

- Essas relações mostram como os algoritmos de escalonamento podem ser generalizados ou combinados.

Exercício 5.6

Pergunta: Por que um algoritmo que favorece processos que usaram menos tempo de CPU recentemente favorece programas voltados para E/S e evita starvation?

Resposta:

- Programas voltados para E/S passam a maior parte do tempo esperando por operações de E/S, usando pouco tempo de CPU. Assim, eles são frequentemente favorecidos por esse algoritmo.
- Programas voltados para CPU, embora possam esperar mais, não sofrem starvation porque, eventualmente, seu tempo de uso recente de CPU se torna baixo, e eles são escalonados novamente.

Explicação:

- Esse equilíbrio é importante para sistemas interativos, onde a responsividade é crucial.

Exercício 5.7

Pergunta: Distinção entre escalonamento PCS e SCS.

Resposta:

- **PCS (Process-Contention Scope):** Escalonamento de threads no nível do processo, onde o sistema operacional não interfere.
- **SCS (System-Contention Scope):** Escalonamento de threads no nível do sistema, onde o sistema operacional gerencia a competição por recursos.

Explicação:

- PCS é comum em threads de usuário, enquanto SCS é comum em threads de kernel.

Exercício 5.8

Pergunta: É necessário vincular uma thread em tempo real a um LWP?

Resposta: Sim, threads em tempo real precisam ser vinculadas a LWPs (Lightweight Processes) para garantir que tenham prioridade e recursos adequados, especialmente em sistemas com

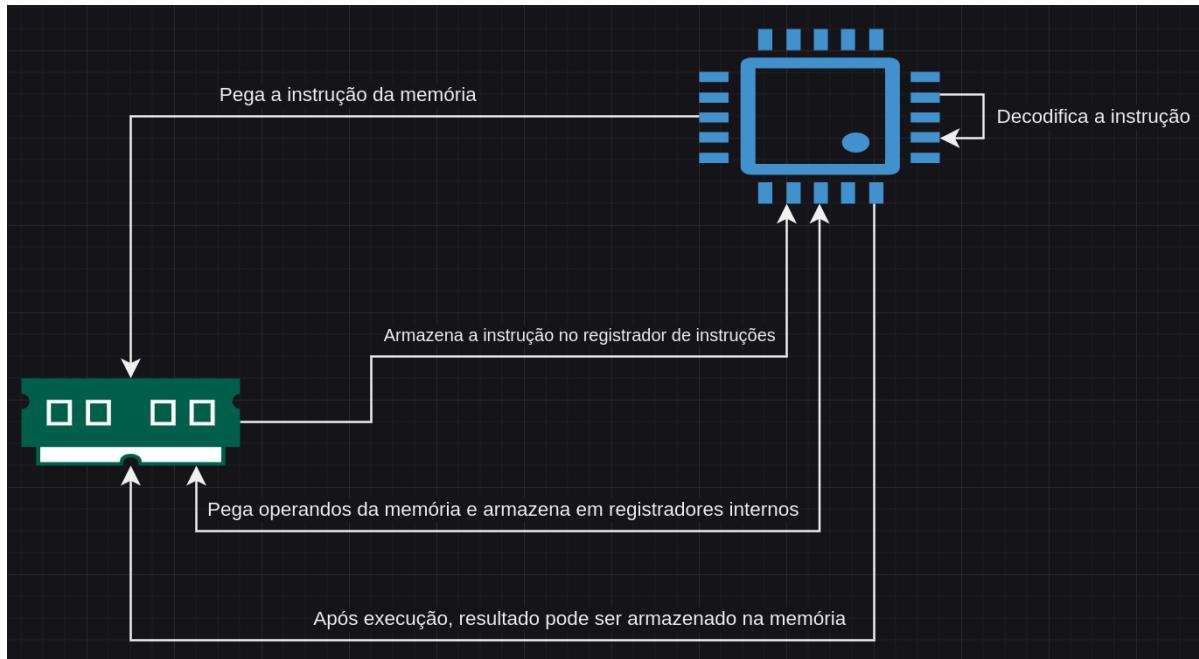
mapeamento muitos-para-muitos.

Explicação:

- LWP s atuam como intermediários entre threads de usuário e threads de kernel, garantindo que threads em tempo real sejam tratadas com a urgência necessária.

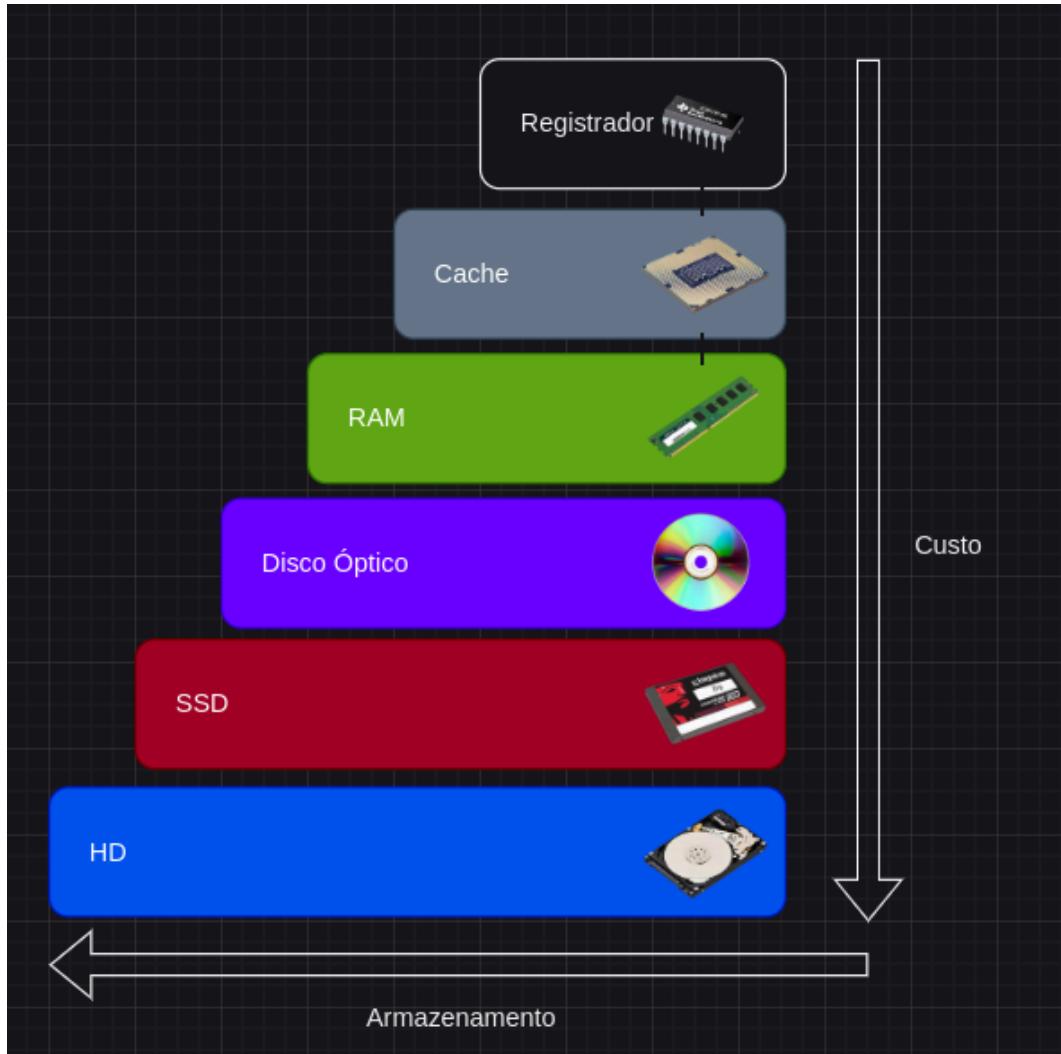
6.1 Introdução - Gerenciamento de Memória

Os sistemas computacionais têm como **principal finalidade a execução de programas**. Para que esses programas possam ser executados, é essencial que estejam armazenados na memória, pelo menos parcialmente, durante sua execução.



Estrutura de Armazenamento

Dessa forma, a importância do gerenciamento de memória reside no fato de que, além de fornecer espaço para armazenamento, é necessário um sistema eficiente para administrar as demandas relacionadas à memória. Esse sistema deve garantir que os recursos de memória sejam alocados, liberados e otimizados de maneira adequada, permitindo que múltiplos programas sejam executados de forma eficaz e sem conflitos.



Estrutura de Armazenamento Hierarquia Dispositivos De Armazenamento

Gerenciamento de Memória

- Objetivo Principal
 - Execução de Programas
 - Alocação Eficiente
- Componentes
 - Memória Principal (RAM)
 - Memória Secundária (HD/SSD)
 - Memória Cache
- Funções
 - Alocação de Memória
 - Liberação de Memória
 - Otimização de Uso
 - Proteção de Memória
- Técnicas

- └── Paginação
- └── Segmentação
- └── Memória Virtual
 - └── Alocação Contígua/Não Contígua
- └── Desafios
 - └── Fragmentação
 - └── Sobre carga de Gerenciamento
 - └── Concorrência de Processos
- └── Benefícios
 - └── Melhor Desempenho
 - └── Execução Simultânea
 - └── Uso Eficiente de Hardware

6.2 Conceitos Básicos

Memória Principal

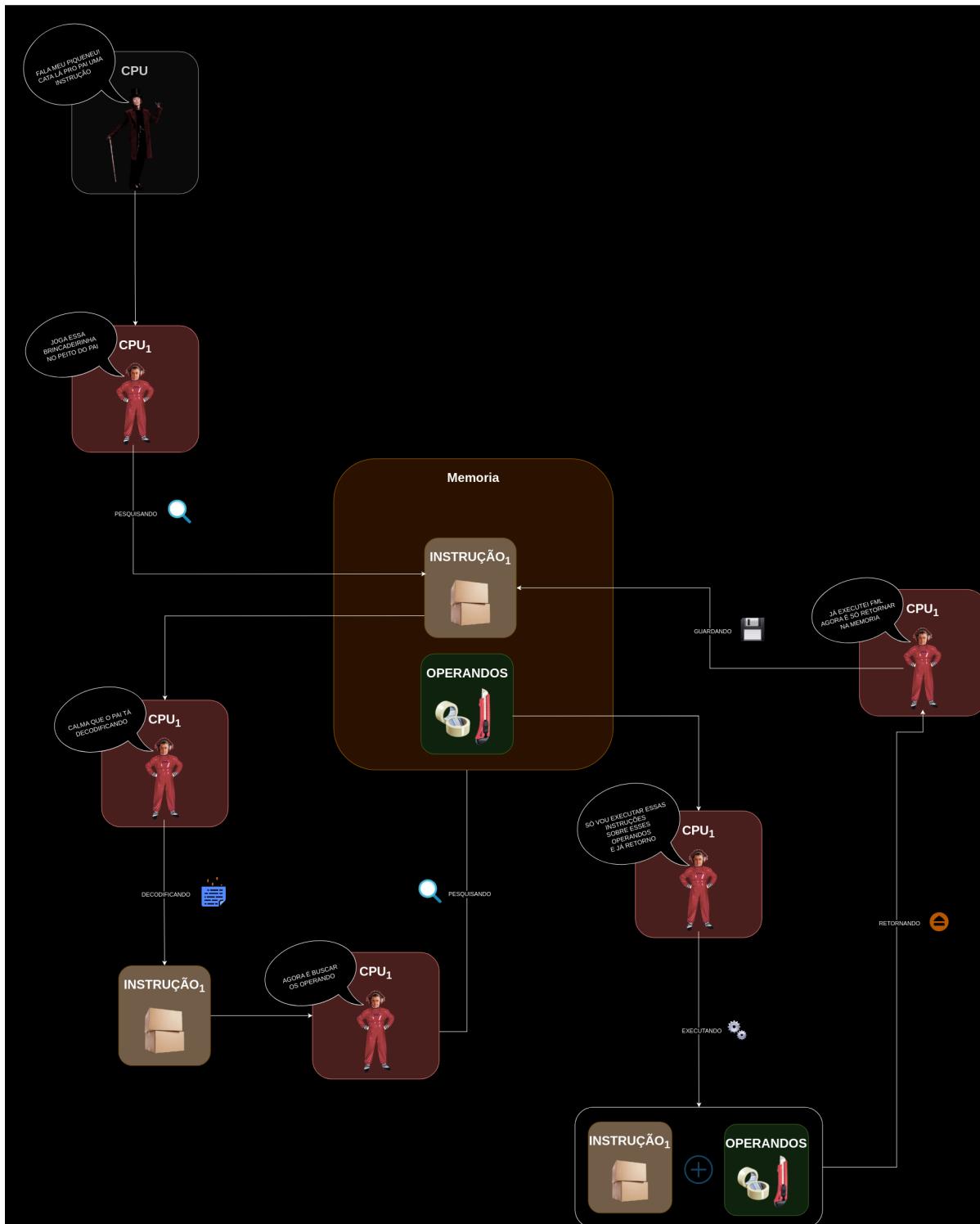
A memória é um componente essencial para os sistemas computacionais. Sua estrutura básica é composta por uma sequência de **words** (palavras) e **bytes**, cada um com seu próprio endereço único. A CPU busca as instruções da memória com base no valor do contador de programa.

Essas instruções podem realizar operações como:

- **Carregamento adicional** de dados.
- **Alocação** em endereços específicos da memória.

Um ciclo comum de execução de instrução envolve as seguintes etapas:

1. **Busca:** A CPU busca uma instrução na memória.
2. **Decodificação:** A instrução é decodificada, e os operandos são buscados na memória.
3. **Execução:** A instrução é executada sobre os operandos.
4. **Armazenamento:** O resultado é guardado de volta na memória.



Ciclo Comum De Execucao De Instrucao Na Memoria

Texto Alternativo: "Diagrama ilustrando o ciclo comum de execução de instrução na memória, composto por quatro etapas: Busca, Decodificação, Execução e Armazenamento. A CPU busca instruções da memória, decodifica e executa as operações, e armazena os resultados de volta na memória."

⚠ A unidade de memória **enxerga apenas um fluxo de endereços**, sem considerar como eles são gerados (por exemplo, pelo contador de programa).

Hardware Básico

A memória principal e os **registradores embutidos** no processador são os **únicos dispositivos de armazenamento diretamente conectados à CPU**. Isso significa que apenas esses componentes podem acessar a CPU diretamente.

Algumas instruções utilizam **endereços de memória** como argumentos, mas não podem acessar **endereços de disco**. Portanto, os dados necessários para a execução das instruções **devem estar na memória principal ou nos registradores** para que a CPU possa processá-los. Caso contrário, os dados precisam ser movidos para a memória antes do processamento.

Velocidade de Acesso

- **Registradores internos:** Acessíveis em um único ciclo de clock da CPU.
- **Memória principal:** O acesso é feito através do **barramento de memória**, podendo levar vários ciclos de clock para ser concluído.

Essa diferença de velocidade pode causar atrasos (stalls) na execução das instruções, já que a CPU pode ficar esperando pelos dados necessários. Para mitigar esse problema, é utilizado um **buffer de memória rápida**, chamado de **08 - Caching**, que fica entre a CPU e a memória principal.

Proteção e Segurança

Além da velocidade, é crucial garantir a **proteção do sistema operacional e dos processos de usuário** uns contra os outros. Essa proteção é implementada em nível de hardware para garantir confiabilidade e segurança.

Garantindo Segurança

Para proteger a memória, cada processo tem um **espaço de endereçamento reservado**. Dois registradores são usados para definir os limites desse espaço:

- **Registrador de Base:** Armazena o endereço físico inicial (menor endereço) do processo.
- **Registrador de Limite:** Armazena o endereço físico final (maior endereço) do processo.

Esses registradores garantem que um processo só acesse os endereços de memória dentro do intervalo permitido, prevenindo acessos indevidos.

Mind Map: Conceitos Básicos de Memória

Memória

- Memória Principal
 - Estrutura
 - Words e Bytes
 - Endereços Únicos
 - Ciclo de Execução
 - Busca
 - Decodificação
 - Execução
 - Armazenamento
 - Fluxo de Endereços
- Hardware Básico
 - Componentes Diretos
 - Memória Principal
 - Registradores
 - Velocidade de Acesso
 - Registradores: 1 ciclo de clock
 - Memória Principal: Vários ciclos
 - Buffer de Memória Rápida (Caching)
 - Proteção e Segurança
 - Espaço de Endereçamento por Processo
 - Registrador de Base
 - Registrador de Limite
- Objetivos
 - Execução Eficiente de Programas
 - Alocação e Liberação de Memória
 - Proteção de Dados e Processos

Associação de Endereços

Imagine que você está jogando **Minecraft**. Seu mundo é como a **memória do computador**, e os **processos** são como **construções** que você cria. Para construir algo, você precisa de **blocos** (dados e instruções) que estão armazenados no seu **inventário** (disco). Para começar a construir, você precisa **trazer os blocos do inventário para o mundo** (memória). Esse processo de mover blocos entre o inventário e o mundo é semelhante à **associação de endereços** na memória.

Diagrama 1: Processo de Construção no Minecraft

Inventário (Disco) → Mundo (Memória) → Construção (Processo)

Etapas de Associação de Endereços

1. Tempo de Compilação (Compile Time):

- É como planejar uma construção no Minecraft antes de começar. Você já sabe exatamente onde cada bloco vai ficar no mundo.
- Se o local inicial mudar, você precisa **replanejar tudo** (recompilar o código).
- **Exemplo no Minecraft:** Você decide construir uma casa em uma coordenada específica ($X=100, Y=64, Z=200$). Se mudar de ideia e quiser construir em outro lugar, terá que refazer o plano.

2. Tempo de Carga (Load Time):

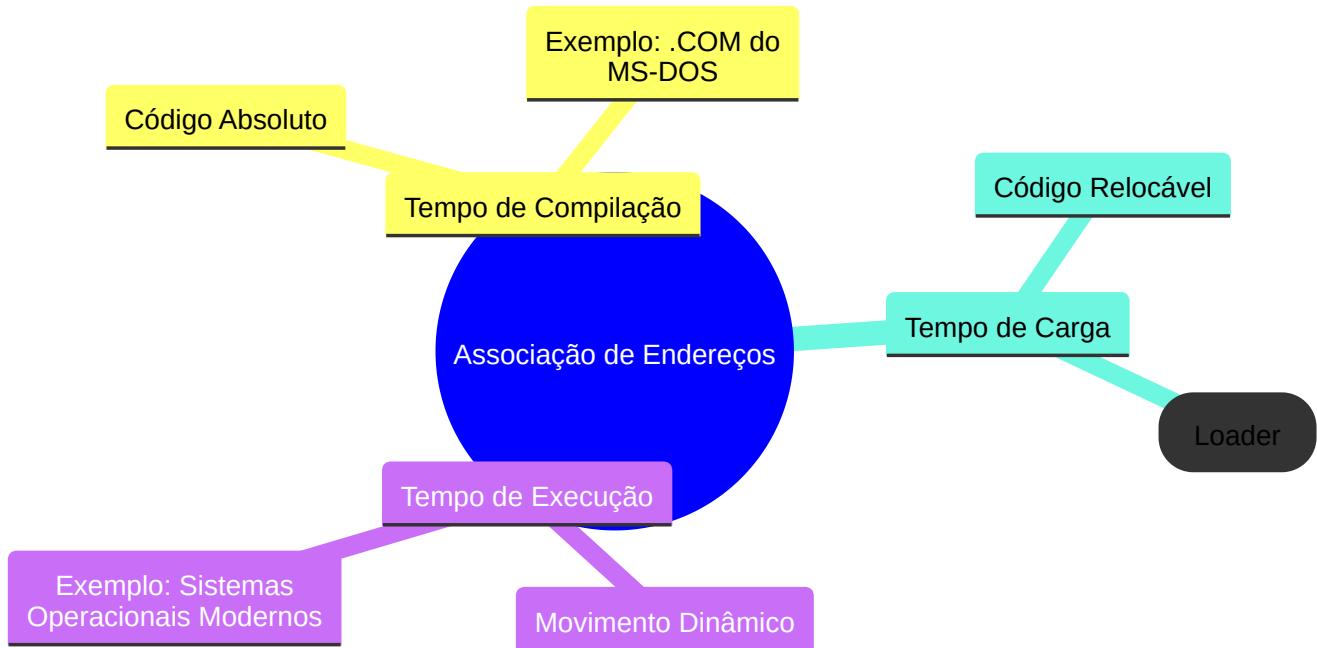
- Aqui, você sabe que vai construir algo, mas ainda não decidiu o local exato. Você só escolhe o local quando começa a colocar os blocos no mundo.
- **Exemplo no Minecraft:** Você tem um projeto de casa, mas só decide onde construí-la quando começa a jogar. Se mudar de local, basta **recarregar** o projeto no novo local.

3. Tempo de Execução (Runtime):

- Nesse caso, você pode **mover a construção** para outro lugar enquanto joga. Isso requer um "poder especial" (hardware adicional) para garantir que tudo funcione corretamente.
- **Exemplo no Minecraft:** Você constrói uma casa e, depois de um tempo, decide movê-la para outro bioma. O jogo precisa ajustar automaticamente as coordenadas dos blocos para que a casa continue intacta.

Diagrama 2: Associação de Endereços

Tempo de Compilação → Tempo de Carga → Tempo de Execução



A **associação de endereços** é como organizar e mover construções no Minecraft. Dependendo do momento em que você decide onde colocar os blocos (dados e instruções), o processo pode ser mais ou menos flexível. No **tempo de compilação**, tudo é fixo; no **tempo de carga**, você escolhe o local ao carregar; e no **tempo de execução**, você pode mover as construções livremente, mas isso requer suporte especial (hardware). Cada método tem suas vantagens e é usado em diferentes cenários, dependendo das necessidades do sistema. 🎮

Espaço de Endereços Lógicos e Físicos

Imagine que você está jogando um jogo de RPG onde o mapa do jogo é dividido em duas partes: o **mapa lógico** (que você vê na tela) e o **mapa físico** (que está armazenado no console). O **endereço lógico** é como a posição que você vê no mapa do jogo (ex: "Floresta das Sombras, coordenada X:10, Y:20"). Já o **endereço físico** é onde essa informação realmente está guardada no hardware do console (ex: "Bloco de memória 1024, setor 512").

- **Endereço Lógico:** É a "coordenada" que o jogo (ou programa) usa para acessar algo. No jogo, você só enxerga isso.
- **Endereço Físico:** É onde essa informação realmente está armazenada no hardware. Você nunca vê isso diretamente.

Mapeamento de Endereços

O **hardware** (MMU - Unidade de Gerenciamento de Memória) faz a tradução entre o endereço lógico e o físico, como um "tradutor" que converte as coordenadas do jogo para o local real no console.

- **Exemplo:** Se o jogo diz "vá para X:10", o MMU traduz isso para "Bloco de memória 1024". Isso é feito em tempo real, enquanto o jogo roda.

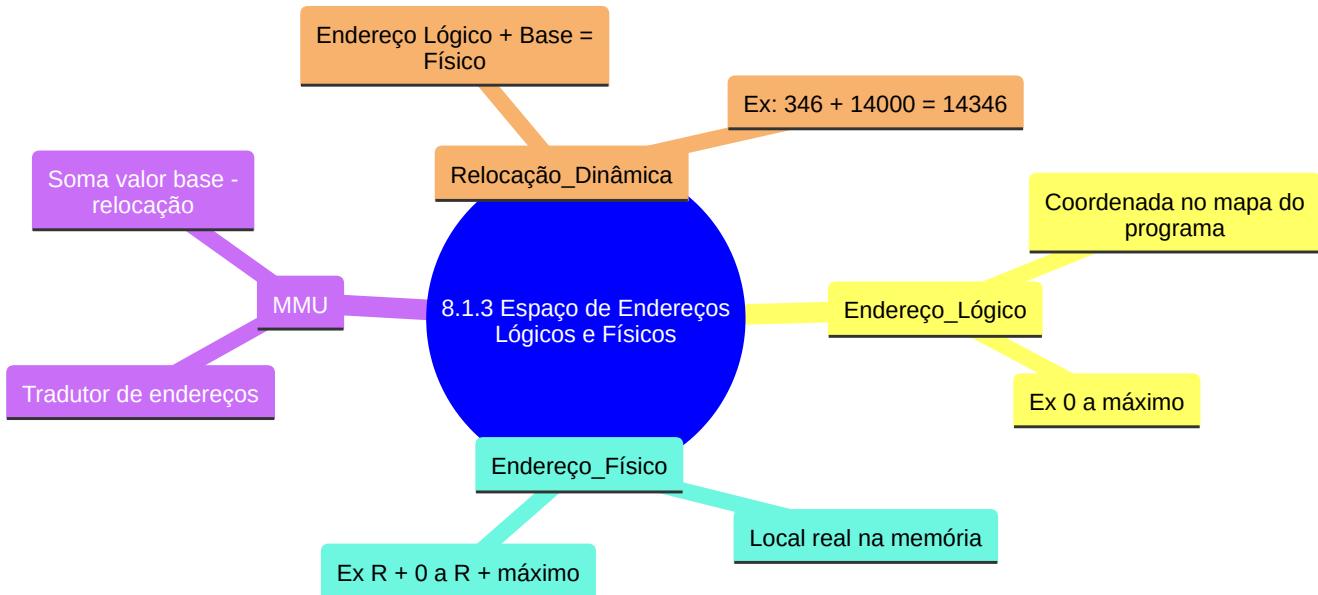
Relocação Dinâmica

Pense em um jogo onde você pode mover seu personagem para qualquer lugar do mapa. O MMU faz algo parecido: ele "reloca" os endereços lógicos para os físicos dinamicamente, somando um valor base (registrar de relocação). Por exemplo:

- Se o valor base for 14000, o endereço lógico "346" vira o físico "14346".

Espaços de Endereços

- **Espaço Lógico:** Todas as "coordenadas" que o jogo (programa) pode usar.
- **Espaço Físico:** Todos os locais reais onde os dados são armazenados.



Resumo

- Endereço Lógico:** O que o programa vê (como coordenadas no jogo).
- Endereço Físico:** Onde os dados realmente estão (como o local no hardware).
- MMU:** Faz a tradução entre os dois, em tempo real.
- Relocação Dinâmica:** Ajusta os endereços lógicos para físicos somando um valor base.

Carregamento dinâmico

Imagine que você está jogando um jogo de mundo aberto, mas o jogo só carrega as partes do mapa que você está explorando no momento. Se você não entra em uma floresta, ela não é carregada na memória. Isso é o **carregamento dinâmico**: só carregar o que é necessário, quando é necessário.

Como Funciona

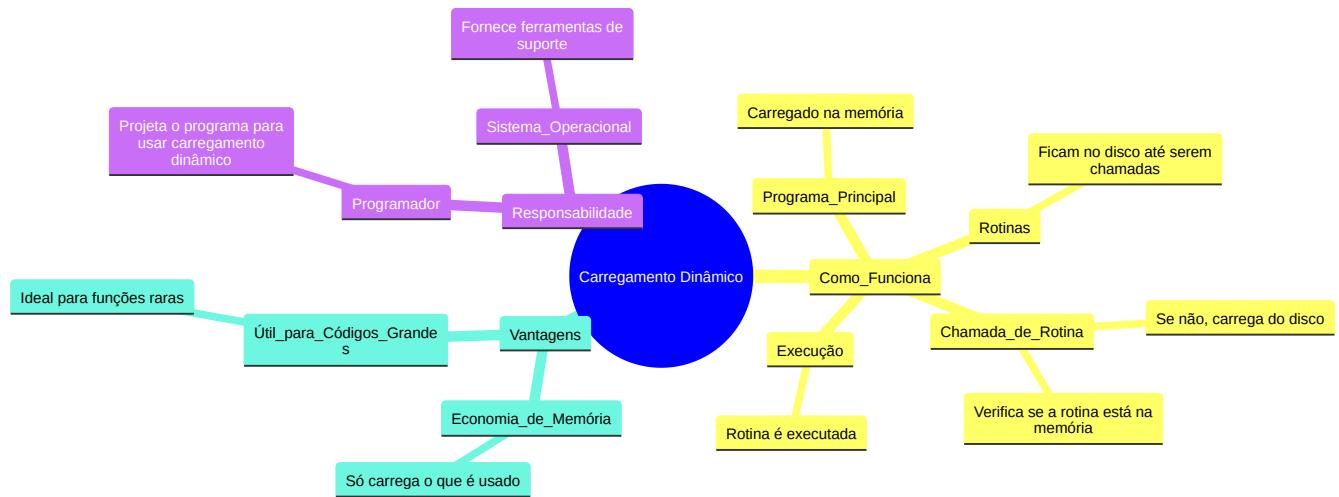
1. **Programa Principal:** O jogo principal (ou programa) é carregado na memória e começa a rodar.
2. **Rotinas (Funções):** As partes do jogo (ou funções do programa) ficam no disco, prontas para serem carregadas.
3. **Chamada de Rotina:** Quando o jogo precisa de uma função específica (ex: abrir um baú), ele verifica se essa função já está na memória.
 - Se não estiver, o **loader** (carregador) busca a função no disco e a coloca na memória.
 - Atualiza as "tabelas de endereços" para saber onde a função foi carregada.
4. **Execução:** A função é executada, e o jogo continua.

Vantagens

- **Economia de Memória:** Só carrega o que é usado. Se uma função nunca é chamada, ela nunca ocupa espaço na memória.
- **Útil para Códigos Grandes:** Ideal para programas com muitas funções, mas que usam apenas algumas delas na maior parte do tempo (ex: rotinas de erro que só rodam em situações raras).

Quem Faz Isso?

- **Programador:** Projeta o programa para usar carregamento dinâmico.
- **Sistema Operacional:** Pode ajudar fornecendo ferramentas (bibliotecas) para facilitar o carregamento dinâmico.



Resumo

- **Carregamento Dinâmico:** Só carrega na memória o que é necessário, quando é necessário.
- **Vantagens:** Economia de memória e eficiência para programas grandes.
- **Responsabilidade:** Programador projeta, sistema operacional pode ajudar.

Bibliotecas de vínculo dinâmico e compartilhadas

Imagine que você está jogando um jogo que usa várias "ferramentas" (como espadas, magias, etc.) que são compartilhadas entre diferentes jogadores. Em vez de cada jogador carregar sua própria cópia dessas ferramentas, todos usam a **mesma cópia** guardada em um local central. Isso é o conceito de **bibliotecas compartilhadas e vínculo dinâmico**.

O que é Vínculo Dinâmico?

- **Vínculo Estático:** Todas as ferramentas (bibliotecas) são copiadas para o jogo de cada jogador (programa). Isso ocupa muito espaço.
- **Vínculo Dinâmico:** As ferramentas ficam em um local central (biblioteca compartilhada). Quando um jogador precisa de uma ferramenta, ele "empresta" da biblioteca central.

Como Funciona?

1. **Stub:** No jogo (programa), há um "placeholder" (stub) que diz onde encontrar a ferramenta (rotina da biblioteca).
 - **O que é um Stub?:** É um pequeno trecho de código que age como um "atalho" para a rotina real. Ele sabe como localizar ou carregar a rotina da biblioteca.
2. **Verificação:** Quando o jogo precisa da ferramenta, o stub verifica se ela já está na memória.
 - Se não estiver, a ferramenta é carregada da biblioteca para a memória.
3. **Substituição:** O stub é substituído pelo endereço real da ferramenta, e ela é usada diretamente.
4. **Compartilhamento:** Todos os jogadores (processos) usam a mesma cópia da ferramenta, economizando memória.

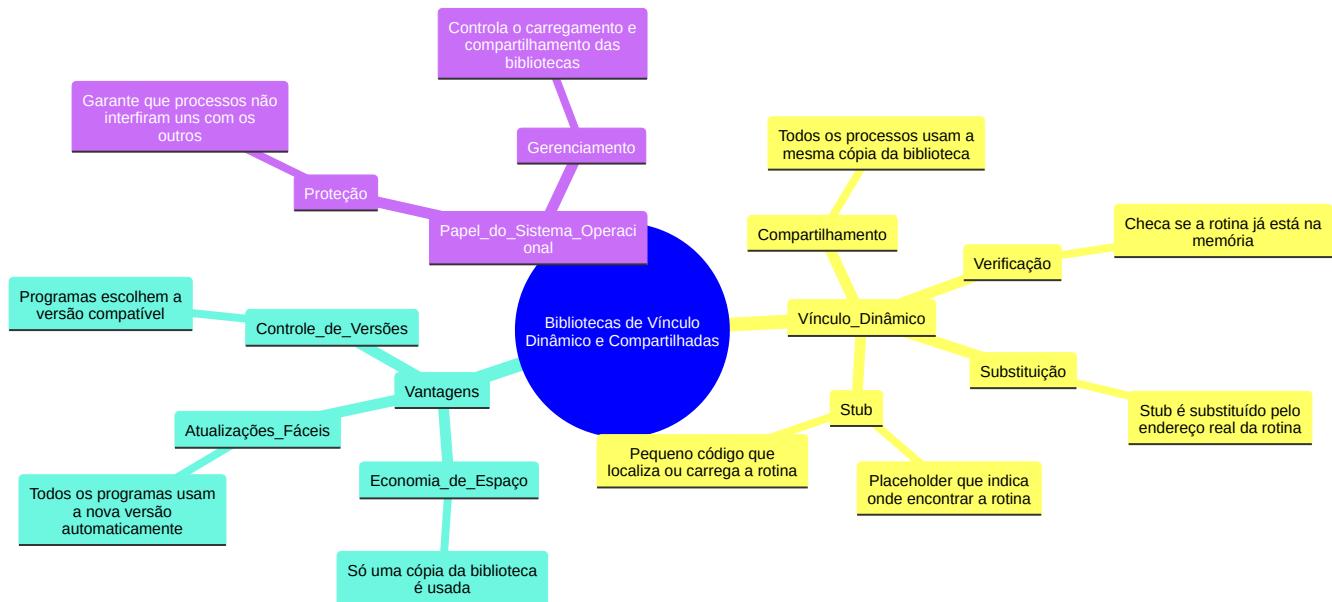
Vantagens

- **Economia de Espaço:** Só uma cópia da biblioteca é usada por todos os programas.
- **Atualizações Fáceis:** Se a biblioteca for atualizada (ex: corrigir bugs), todos os programas automaticamente usam a nova versão.

- **Controle de Versões:** Programas podem escolher qual versão da biblioteca usar, evitando incompatibilidades.

Papel do Sistema Operacional

- **Proteção:** O sistema operacional garante que os programas não interfiram uns com os outros ao acessar a mesma biblioteca.
- **Gerenciamento:** Ele controla o carregamento e o compartilhamento das bibliotecas na memória.



Resumo

- **Stub:** Um "atalho" que localiza ou carrega a rotina da biblioteca.
- **Vínculo Dinâmico:** As bibliotecas são carregadas e compartilhadas em tempo de execução.
- **Vantagens:** Economia de espaço, atualizações fáceis e controle de versões.
- **Sistema Operacional:** Gerencia o acesso e a proteção das bibliotecas compartilhadas.

6.3 Swapping

Imagine que você está jogando um jogo de estratégia onde só pode ter um número limitado de unidades (processos) no campo de batalha (memória) ao mesmo tempo. Quando uma nova unidade precisa entrar, você remove temporariamente uma unidade existente e a guarda no "banco de reservas" (backing store, como um disco). Isso é o **swapping**: mover processos entre a memória e o disco para liberar espaço.

Como Funciona?

1. **Swap Out:** Quando um processo não está ativo (ex: terminou seu tempo de execução ou tem prioridade baixa), ele é movido da memória para o disco.
2. **Swap In:** Quando o processo precisa ser executado novamente, ele é trazido de volta para a memória.
3. **Fila de Prontos:** O sistema mantém uma lista de processos prontos para executar, estejam na memória ou no disco.

Quando é Usado?

- **Escalonamento Round Robin:** Quando o tempo de execução (quantum) de um processo acaba, ele é trocado por outro.
- **Prioridade:** Se um processo de alta prioridade chega, um de baixa prioridade pode ser trocado para liberar espaço.

Desafios do Swapping

1. **Tempo de Transferência:** Mover processos entre memória e disco é lento. Por exemplo:
 - Um processo de 100 MB em um disco com taxa de transferência de 50 MB/s leva 2 segundos para ser movido.
 - Considerando a latência do disco, o tempo total pode chegar a 4 segundos (swap out + swap in).
2. **E/S Pendente:** Se um processo está aguardando uma operação de E/S (ex: leitura de dados), ele não pode ser trocado, pois a E/S pode tentar acessar memória que não está mais disponível.

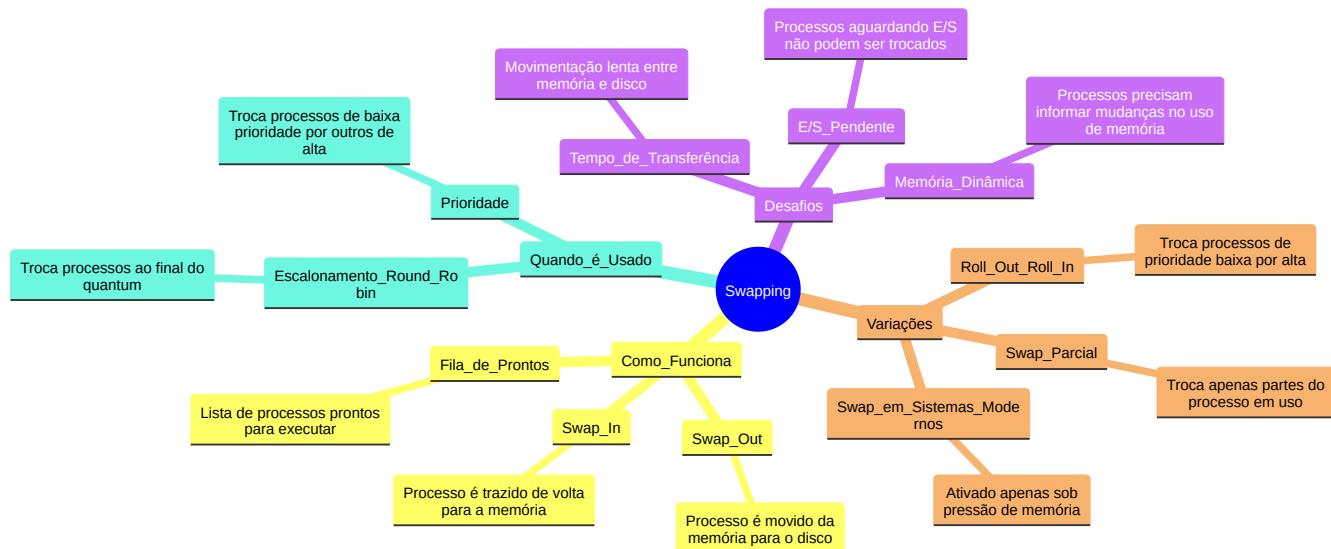
3. Memória Dinâmica: Processos que mudam seu uso de memória durante a execução precisam informar ao sistema operacional para evitar problemas.

Variações do Swapping

- **Roll Out, Roll In:** Troca processos de baixa prioridade por outros de alta prioridade.
- **Swap Parcial:** Troca apenas partes do processo que estão em uso, em vez de todo o processo.
- **Swap em Sistemas Modernos:** Em sistemas como UNIX, o swap é ativado apenas quando a memória está sob pressão.

Exemplo Prático

- **Windows 3.1:** Usava uma versão simples de swap, onde o usuário decidia manualmente quais processos trocar.
- **Sistemas Modernos:** Usam técnicas mais avançadas, como memória virtual, para evitar o swap completo.



Resumo

- **Swapping:** Move processos entre memória e disco para liberar espaço.
- **Desafios:** Tempo de transferência lento e problemas com E/S pendente.
- **Variações:** Roll out/roll in, swap parcial e uso em sistemas modernos.

- **Objetivo:** Maximizar o uso da memória e permitir a execução de múltiplos processos.

6.4 Alocação de memória contígua

Imagine que a memória do computador é como um grande armário com gavetas. Cada gaveta precisa ser usada de forma eficiente para guardar coisas (processos). A **alocação de memória contígua** é como organizar as gavetas de modo que cada processo ocupe uma seção contínua do armário, sem espaços vazios no meio.

Como Funciona?

1. Partições da Memória:

- **Memória Baixa:** Reservada para o sistema operacional.
- **Memória Alta:** Reservada para os processos do usuário.

2. Alocação Contígua:

Cada processo ocupa uma única seção contínua de memória. Por exemplo:

- Processo A: Ocupa as gavetas 1 a 3.
- Processo B: Ocupa as gavetas 4 a 6.
- Processo C: Ocupa as gavetas 7 a 10.

Desafios

1. Fragmentação:

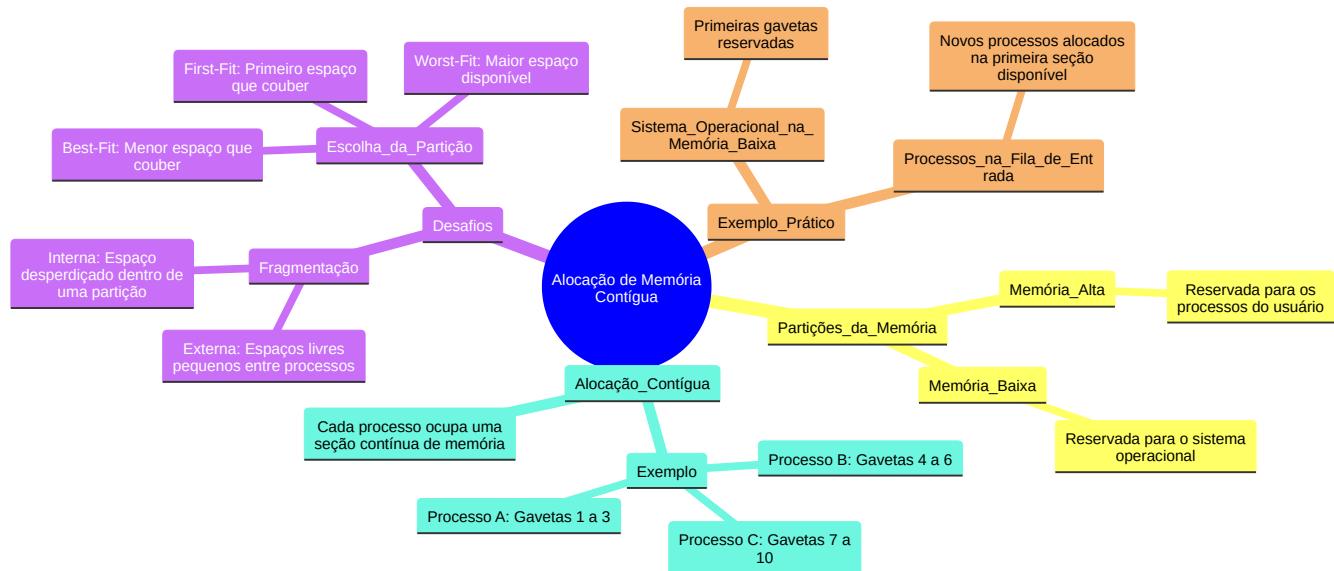
- **Fragmentação Externa:** Espaços livres entre processos alocados, que são pequenos demais para serem usados.
- **Fragmentação Interna:** Espaço desperdiçado dentro de uma partição alocada, porque o processo não usa toda a memória reservada.

2. Escolha da Partição:

- **First-Fit:** Aloca o primeiro espaço livre que couber o processo.
- **Best-Fit:** Aloca o menor espaço livre que couber o processo.
- **Worst-Fit:** Aloca o maior espaço livre disponível.

Exemplo Prático

- **Sistema Operacional na Memória Baixa:** O sistema operacional fica nas primeiras gavetas (memória baixa), e os processos do usuário ocupam o restante (memória alta).
- **Processos na Fila de Entrada:** Se um novo processo chega, ele é colocado na primeira seção contígua de memória disponível que couber.



Resumo

- **Alocação Contígua:** Cada processo ocupa uma seção contínua de memória.
- **Desafios:** Fragmentação (externa e interna) e escolha da partição (first-fit, best-fit, worst-fit).
- **Objetivo:** Usar a memória de forma eficiente, minimizando espaços desperdiçados.

Proteção e Mapeamento da Memória

Imagine que a memória do computador é como um prédio com vários apartamentos (processos). Para garantir que um morador (processo) não entre no apartamento errado ou cause problemas, precisamos de um **sistema de segurança** (proteção) e um **mapa** (mapeamento) que mostre onde cada morador pode ir. Isso é feito usando dois registradores especiais: o **registraror de relocação** e o **registraror de limite**.

Como Funciona?

1. Registrador de Relocação:

- Contém o endereço físico inicial onde o processo começa.
- Exemplo: Se o valor for 100040, o processo começa nesse endereço.

2. Registrador de Limite:

- Define o tamanho máximo que o processo pode ocupar.
- Exemplo: Se o valor for 74600, o processo não pode acessar memória além de $100040 + 74600$.

3. Proteção:

- Cada endereço gerado pelo processo é verificado:
 - Se o endereço for maior que o limite, ocorre um erro (proteção contra acesso indevido).
 - Caso contrário, o endereço é mapeado somando o valor do registrador de relocação.

4. Mapeamento:

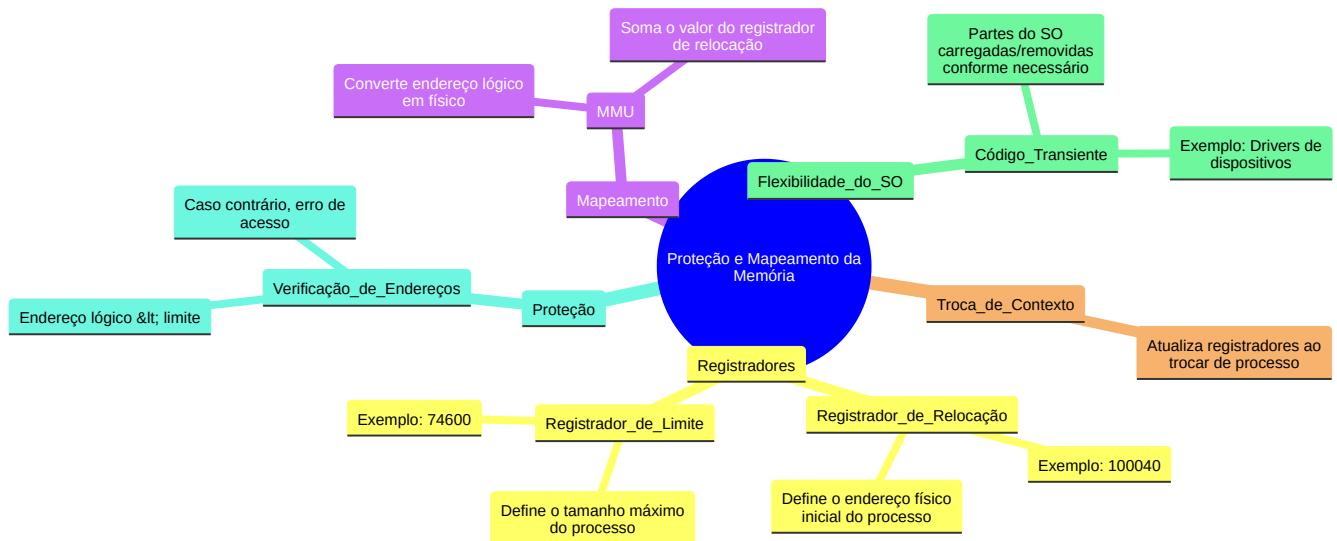
- A MMU (Unidade de Gerenciamento de Memória) converte o endereço lógico (visto pelo processo) em um endereço físico (real na memória).

Troca de Contexto

- Quando o sistema operacional troca de processo, ele atualiza os registradores de relocação e limite para os valores corretos do novo processo.
- Isso garante que cada processo só acesse sua própria área de memória.

Flexibilidade do Sistema Operacional

- **Código Transiente:** Partes do sistema operacional (ex: drivers de dispositivos) podem ser carregadas e removidas da memória conforme necessário.
- Isso permite que o sistema operacional ajuste dinamicamente seu tamanho, liberando espaço para outros processos.



Resumo

- **Registradores de Relocação e Limite:** Protegem e mapeiam a memória, garantindo que cada processo acesse apenas sua área.
- **Proteção:** Evita que processos acessem memória indevida.
- **Mapeamento:** Converte endereços lógicos em físicos.
- **Flexibilidade:** Permite que o sistema operacional ajuste seu tamanho dinamicamente.

Alocação de Memória

Imagine que a memória do computador é como um grande armário com várias gavetas. Cada gaveta pode guardar uma coisa, mas o tamanho das gavetas pode variar. A **alocação de memória** é o processo de decidir qual gaveta usar para cada coisa, de forma eficiente.

Métodos de Alocação

1. Partições de Tamanho Fixo:

- A memória é dividida em gavetas de tamanho fixo.
- Cada gaveta guarda um processo.
- Limitação: O número máximo de processos é igual ao número de gavetas.
- Exemplo: IBM OS/360 MFT.

2. Partições de Tamanho Variável:

- A memória é tratada como um grande espaço contínuo, dividido em buracos de tamanhos variados.
- Quando um processo chega, ele é colocado em um buraco que caiba.
- Se o buraco for maior que o necessário, ele é dividido: uma parte é usada pelo processo, e a outra volta para a lista de buracos.
- Quando um processo termina, seu espaço é liberado e pode ser mesclado com buracos adjacentes para formar um buraco maior.
- Exemplo: IBM OS/360 MVT.

Estratégias de Alocação

• First-Fit:

- Aloca o primeiro buraco que couber o processo.
- Rápido, mas pode deixar buracos pequenos e não contíguos.

• Best-Fit:

- Aloca o menor buraco que couber o processo.

- Pode criar muitos buracos pequenos e inúteis.
- **Worst-Fit:**
 - Aloca o maior buraco disponível.
 - Tende a deixar buracos grandes, que podem ser úteis no futuro.

Desafios

- **Fragmentação:**
 - Fragmentação Externa: Buracos pequenos e espalhados que não podem ser usados.
 - Fragmentação Interna: Espaço desperdiçado dentro de uma partição alocada.
- **Eficiência:**
 - O first-fit é geralmente mais rápido, enquanto o best-fit pode ser mais eficiente no uso da memória.

Exemplo Prático

- **Fila de Entrada:** Processos aguardando para serem alocados na memória.
- **Lista de Buracos:** Espaços livres na memória, que podem ser mesclados ou divididos conforme necessário.



Resumo

- **Partições Fixas:** Gavetas de tamanho fixo, limitando o número de processos.
- **Partições Variáveis:** Buracos de tamanho variável, permitindo alocação dinâmica.
- **Estratégias:** First-fit é rápido, best-fit pode ser mais eficiente, worst-fit deixa buracos grandes.
- **Desafios:** Fragmentação externa e interna, e eficiência na alocação.

Fragmentação

Imagine que a memória do computador é como um grande armário cheio de gavetas. Com o tempo, as gavetas vão sendo usadas e liberadas, mas de forma desorganizada, deixando pequenos espaços vazios entre elas. Esses espaços são a **fragmentação**, que pode ser de dois tipos: **externa** e **interna**.

Fragmentação Externa

- **O que é:** Espaços livres pequenos e não contíguos na memória.
- **Problema:** Mesmo que haja memória livre suficiente, ela pode estar dividida em pedaços tão pequenos que não podem ser usados.
- **Exemplo:** Se você tem 10 MB de memória livre, mas dividida em 10 pedaços de 1 MB, não é possível alocar um processo de 2 MB.
- **Causas:** Alocação e liberação repetida de processos, especialmente com estratégias como **first-fit** e **best-fit**.
- **Regra dos 50%:** Em média, um terço da memória pode ficar inutilizável devido à fragmentação.

Fragmentação Interna

- **O que é:** Espaço desperdiçado dentro de uma partição alocada.
- **Problema:** A memória alocada é maior que a necessária, deixando um pedaço inutilizado.
- **Exemplo:** Se um processo precisa de 18.462 bytes e a partição alocada tem 18.464 bytes, sobram 2 bytes que não são usados.
- **Causa:** Alocação em blocos de tamanho fixo, onde o processo não usa todo o espaço reservado.

Soluções para Fragmentação

1. Compactação:

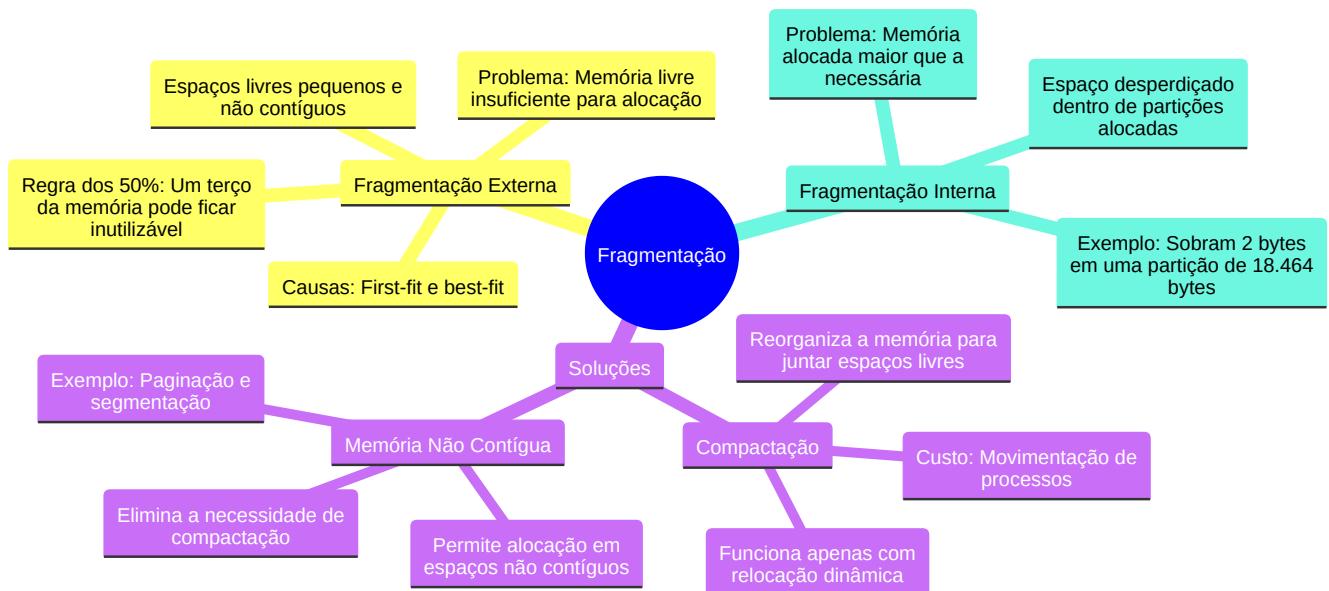
- **O que é:** Reorganizar a memória para juntar todos os espaços livres em um único bloco.
- **Desafio:** Só funciona se a relocação for dinâmica (feita em tempo de execução).

- **Custo:** Pode ser caro em termos de tempo e processamento, pois todos os processos precisam ser movidos.

2. Memória Não Contígua:

- **O que é:** Permitir que um processo use espaços de memória não contíguos.
- **Vantagem:** Elimina a necessidade de compactação, pois os processos podem ser alocados em qualquer espaço livre.
- **Exemplo:** Técnicas como **paginção** e **segmentação** (abordadas em seções posteriores).

Diagrama Mermaid



Resumo

- **Fragmentação Externa:** Espaços livres pequenos e não contíguos, causados por alocação e liberação repetida.
- **Fragmentação Interna:** Espaço desperdiçado dentro de partições alocadas, devido a blocos de tamanho fixo.
- **Soluções:** Compactação (reorganização da memória) e memória não contígua (paginação e segmentação).

6.5 Páginas

Imagine que a memória do computador é como um livro, e cada página desse livro é um pequeno bloco de memória. A **paginação** é uma técnica que divide a memória em "páginas" de tamanho fixo, permitindo que um processo use páginas não contíguas. Isso resolve problemas como a **fragmentação externa** e elimina a necessidade de **compactação**.

Como Funciona?

1. Divisão da Memória:

- A memória física é dividida em **quadros** (frames) de tamanho fixo.
- A memória lógica (vista pelo processo) é dividida em **páginas** do mesmo tamanho dos quadros.

2. Tabela de Páginas:

- Cada processo tem uma **tabela de páginas** que mapeia suas páginas lógicas para quadros físicos.
- Exemplo: A página 1 do processo pode estar no quadro 3 da memória física.

3. Endereçamento:

- O endereço lógico é dividido em duas partes:
 - **Número da página**: Identifica a página no espaço lógico.
 - **Deslocamento**: Indica a posição dentro da página.
- A tabela de páginas converte o número da página no número do quadro físico, e o deslocamento é mantido.

Vantagens

- **Elimina Fragmentação Externa**: Como as páginas são de tamanho fixo, não há buracos pequenos e inúteis.
- **Não Precisa de Compactação**: A memória é gerenciada de forma eficiente sem precisar reorganizar processos.
- **Facilita o Swapping**: Páginas podem ser movidas para o disco (backing store) e trazidas de volta sem problemas de fragmentação.

Desafios

- **Overhead da Tabela de Páginas:** A tabela de páginas pode ocupar muita memória, especialmente em sistemas com espaço de endereçamento grande.
- **Acesso à Memória:** Cada acesso à memória requer uma consulta à tabela de páginas, o que pode ser lento sem otimizações como TLB (Translation Lookaside Buffer).

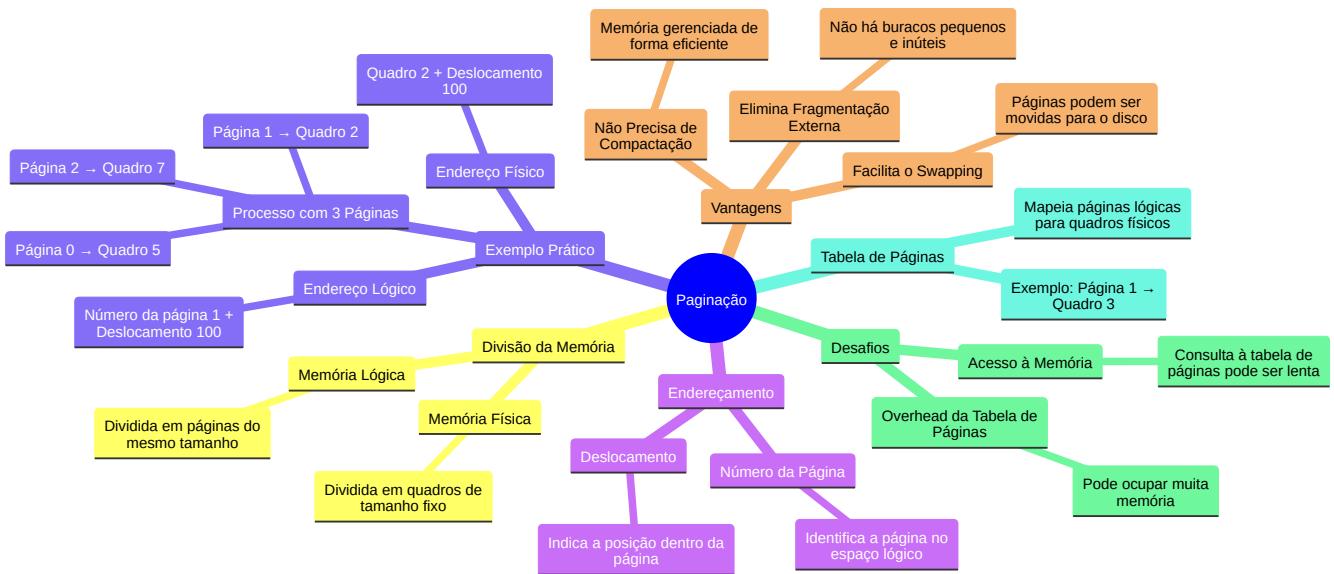
Exemplo Prático

- **Processo com 3 Páginas:**

- Página 0 → Quadro 5
- Página 1 → Quadro 2
- Página 2 → Quadro 7

- **Endereço Lógico:** Número da página 1 + Deslocamento 100.

- **Endereço Físico:** Quadro 2 + Deslocamento 100.



Resumo

- **Paginação:** Divide a memória em páginas de tamanho fixo, permitindo alocação não contígua.
- **Vantagens:** Elimina fragmentação externa, não precisa de compactação e facilita o swapping.

- **Desafios:** Overhead da tabela de páginas e acesso à memória mais lento.
- **Funcionamento:** Tabela de páginas mapeia páginas lógicas para quadros físicos.

Método Básico da Paginação

A **paginação** é uma técnica de gerenciamento de memória que divide a memória física e lógica em blocos de tamanho fixo, chamados **quadros** (na memória física) e **páginas** (na memória lógica). Esse método elimina a **fragmentação externa** e facilita o gerenciamento de memória, permitindo que um processo use páginas não contíguas na memória física.

1. Divisão da Memória

Memória Física

- Dividida em **quadros** de tamanho fixo.
- Exemplo: Se o tamanho do quadro for 4 KB, uma memória de 32 KB terá 8 quadros.

Memória Lógica

- Dividida em **páginas** do mesmo tamanho dos quadros.
- Cada processo tem seu próprio espaço de endereçamento lógico, dividido em páginas.

Armazenamento de Apoio (Disco)

- Também dividido em blocos do mesmo tamanho das páginas/quadros.
- Usado para armazenar páginas que não cabem na memória física (swapping).

2. Tabela de Páginas

Cada processo possui uma **tabela de páginas**, que mapeia suas páginas lógicas para quadros físicos. A tabela de páginas é usada pelo hardware (MMU - Unidade de Gerenciamento de Memória) para traduzir endereços lógicos em endereços físicos.

Estrutura da Tabela de Páginas

- **Número da Página (p)**: Índice na tabela de páginas.
- **Endereço do Quadro Físico**: Localização real da página na memória física.

3. Endereçamento

O endereço lógico gerado pela CPU é dividido em duas partes:

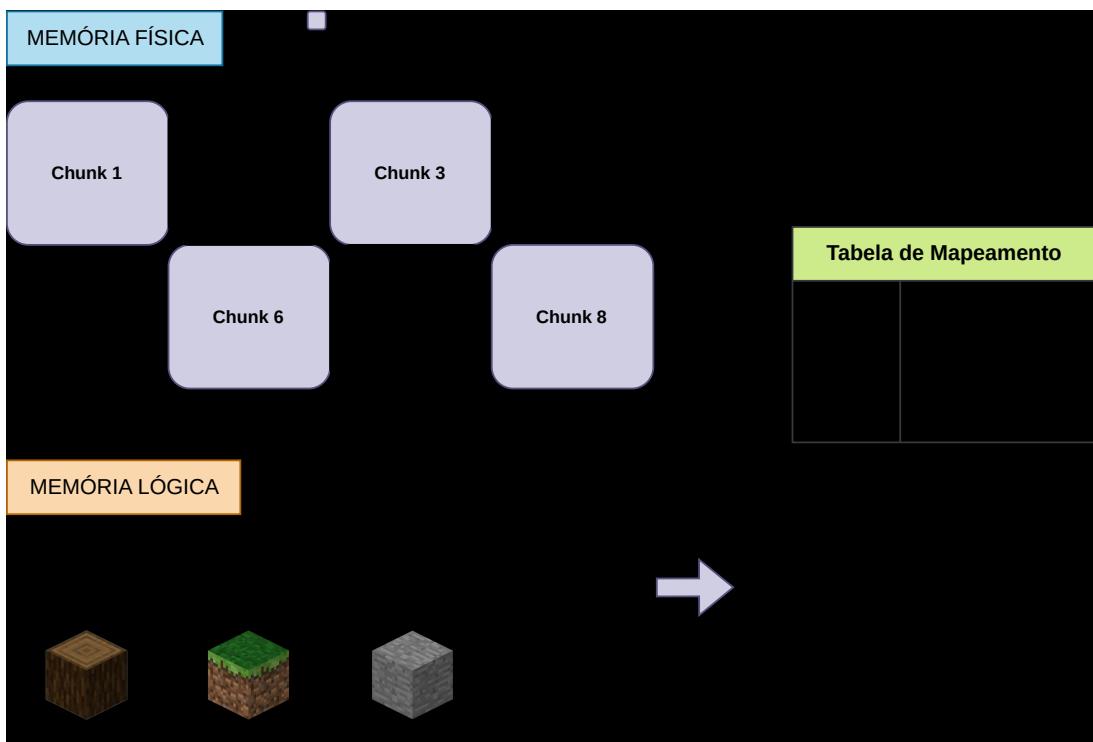
1. Número da Página (p): Identifica a página no espaço lógico.

2. Deslocamento (d): Indica a posição dentro da página.

Tradução de Endereço

1. O hardware usa o número da página para consultar a tabela de páginas e obter o endereço do quadro físico.

2. O endereço físico é formado pela combinação do endereço do quadro físico e do deslocamento.



Memória Física e Lógica.drawio.svg

4. Exemplo Prático

Memória Física

- Tamanho do quadro: 4 bytes.
- Memória física: 32 bytes (8 quadros).

Memória Lógica

- Tamanho da página: 4 bytes.

- Processo com 3 páginas:
 - Página 0 → Quadro 5
 - Página 1 → Quadro 2
 - Página 2 → Quadro 7

Tradução de Endereço

- **Endereço Lógico 0:**
 - Página 0, Deslocamento 0.
 - Endereço Físico: $(5 \times 4) + 0 = 20$.
- **Endereço Lógico 3:**
 - Página 0, Deslocamento 3.
 - Endereço Físico: $(5 \times 4) + 3 = 23$.
- **Endereço Lógico 4:**
 - Página 1, Deslocamento 0.
 - Endereço Físico: $(6 \times 4) + 0 = 24$.

5. Fragmentação

Fragmentação Externa

- **Eliminada:** Como as páginas são de tamanho fixo, não há buracos pequenos e inúteis.

Fragmentação Interna

- **Ocorre:** Se o processo não usar todo o espaço de uma página, o restante fica inutilizado.
- **Exemplo:** Um processo de 72.766 bytes com páginas de 2.048 bytes precisaria de 36 páginas, resultando em 962 bytes de fragmentação interna.

6. Tamanho da Página

- **Tamanho Fixo:** Definido pelo hardware, geralmente uma potência de 2 (ex: 4 KB, 8 KB).

- **Vantagens:**
 - Facilita a tradução de endereços.
 - Melhora a eficiência de E/S de disco.
- **Desvantagens:**
 - Fragmentação interna pode aumentar com páginas grandes.

7. Diagramas

Diagrama 1: Paginação da Memória Lógica e Física

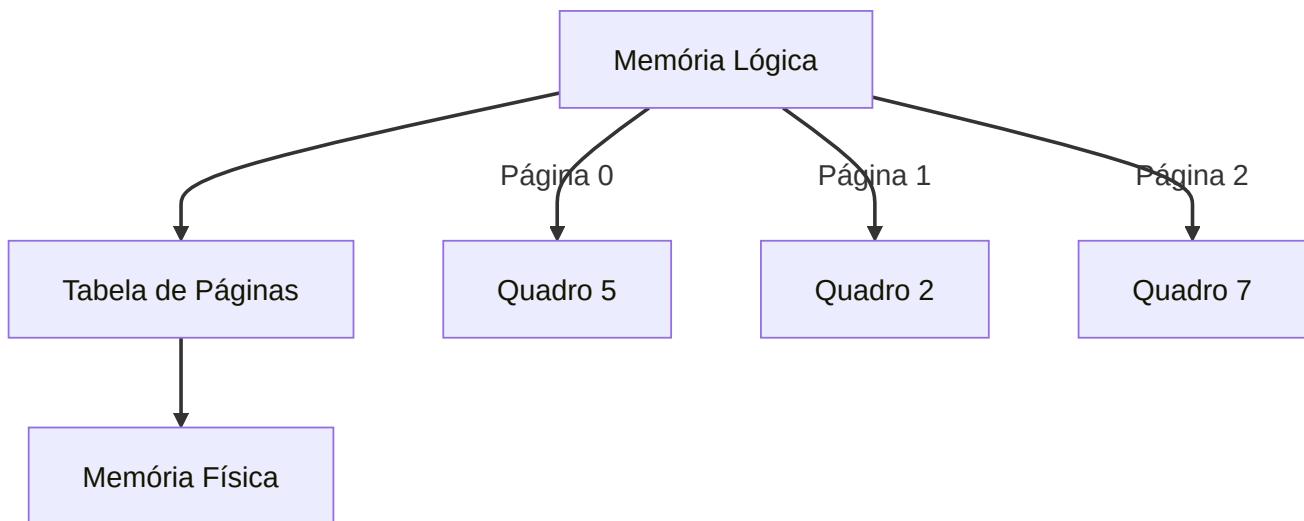
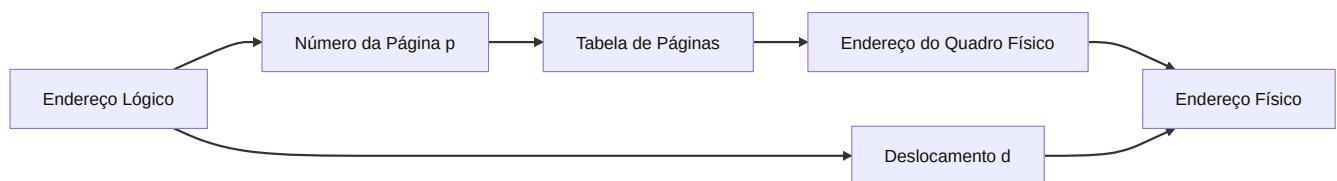


Diagrama 2: Tradução de Endereço



8. Vantagens e Desafios

Vantagens

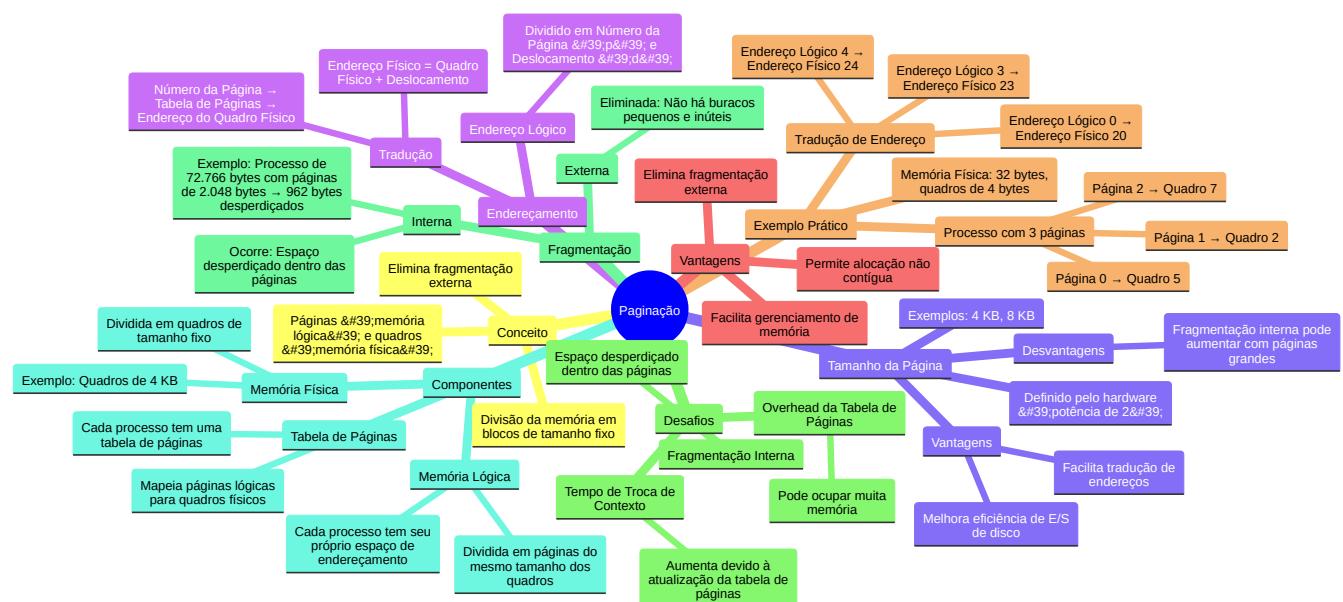
- Elimina fragmentação externa.
- Facilita o gerenciamento de memória e o swapping.
- Permite alocação não contígua de memória.

Desafios

- **Overhead da Tabela de Páginas:** Pode ocupar muita memória.
- **Fragmentação Interna:** Espaço desperdiçado dentro das páginas.
- **Tempo de Troca de Contexto:** Aumenta devido à necessidade de atualizar a tabela de páginas.

Resumo

- **Paginação:** Divide a memória em páginas e quadros de tamanho fixo.
- **Tabela de Páginas:** Mapeia páginas lógicas para quadros físicos.
- **Endereçamento:** Número da página + Deslocamento → Endereço físico.
- **Vantagens:** Elimina fragmentação externa e facilita o gerenciamento de memória.
- **Desafios:** Fragmentação interna e overhead da tabela de páginas.



Suporte do Hardware para Paginação

A paginação é uma técnica poderosa para gerenciar memória, mas seu sucesso depende do hardware que a implementa. Vamos explorar como o hardware suporta a paginação, incluindo o uso de **tabelas de páginas**, **TLB** (Translation Look-aside Buffer) e **registradores especiais**.

1. Tabelas de Páginas e Registradores

Tabela de Páginas

- Cada processo tem sua própria **tabela de páginas**, que mapeia páginas lógicas para quadros físicos.
- A tabela de páginas é armazenada na **memória principal**.
- Um **registrador especial**, chamado **PTBR (Page Table Base Register)**, aponta para o início da tabela de páginas do processo em execução.

Troca de Contexto

- Quando o sistema operacional troca de processo, ele atualiza o **PTBR** para apontar para a tabela de páginas do novo processo.
- Isso é feito pelo **despachante** (scheduler), que também recarrega outros registradores, como o **contador de instrução**.

Desafio: Acesso à Memória

- Para acessar um endereço físico, o hardware precisa:
 1. Consultar a tabela de páginas (usando o PTBR).
 2. Obter o número do quadro físico.
 3. Combinar o quadro físico com o deslocamento para formar o endereço físico.
- Isso resulta em **dois acessos à memória** (um para a tabela de páginas e outro para o dado), o que pode ser lento.

2. TLB (Translation Look-aside Buffer)

Para acelerar o processo de tradução de endereços, o hardware usa um **cache especial** chamado

TLB.

O que é a TLB?

- A TLB é uma memória associativa de alta velocidade que armazena entradas recentes da tabela de páginas.
- Cada entrada na TLB contém:
 - **Chave (Tag)**: Número da página.
 - **Valor**: Número do quadro físico correspondente.

Funcionamento da TLB

1. Quando a CPU gera um endereço lógico, o número da página é enviado à TLB.
2. Se a página estiver na TLB (**TLB hit**), o número do quadro é retornado imediatamente.
3. Se a página não estiver na TLB (**TLB miss**), o hardware consulta a tabela de páginas na memória principal e atualiza a TLB com a nova entrada.

Vantagens da TLB

- Reduz o tempo de tradução de endereços.
- A maioria dos acessos à memória é resolvida pela TLB, evitando consultas à tabela de páginas na memória principal.

Taxa de Acertos (Hit Rate)

- A **taxa de acertos** é a porcentagem de vezes que a TLB encontra o número da página.
- Exemplo:
 - Taxa de acertos de 80%: 80% dos acessos são resolvidos pela TLB.
 - Taxa de acertos de 98%: 98% dos acessos são resolvidos pela TLB.

Cálculo do Tempo Efetivo de Acesso

- **TLB Hit:** 20 ns (pesquisa na TLB) + 100 ns (acesso à memória) = 120 ns.
- **TLB Miss:** 20 ns (pesquisa na TLB) + 100 ns (acesso à tabela de páginas) + 100 ns (acesso à

memória) = 220 ns.

- **Tempo Efetivo:**

- Para taxa de acertos de 80%: $(0,80 \times 120 + 0,20 \times 220 = 140)$ ns.
- Para taxa de acertos de 98%: $(0,98 \times 120 + 0,02 \times 220 = 122)$ ns.

3. ASID (Address Space Identifier)

Para evitar conflitos entre processos, a TLB pode usar **ASIDs**.

O que é o ASID?

- Um **ASID** é um identificador único para cada processo.
- Cada entrada na TLB contém um ASID, que garante que as traduções de endereços sejam válidas apenas para o processo correto.

Vantagens do ASID

- Permite que a TLB armazene entradas de vários processos simultaneamente.
- Evita a necessidade de esvaziar a TLB a cada troca de contexto.

4. Diagramas

Diagrama 1: Funcionamento da TLB

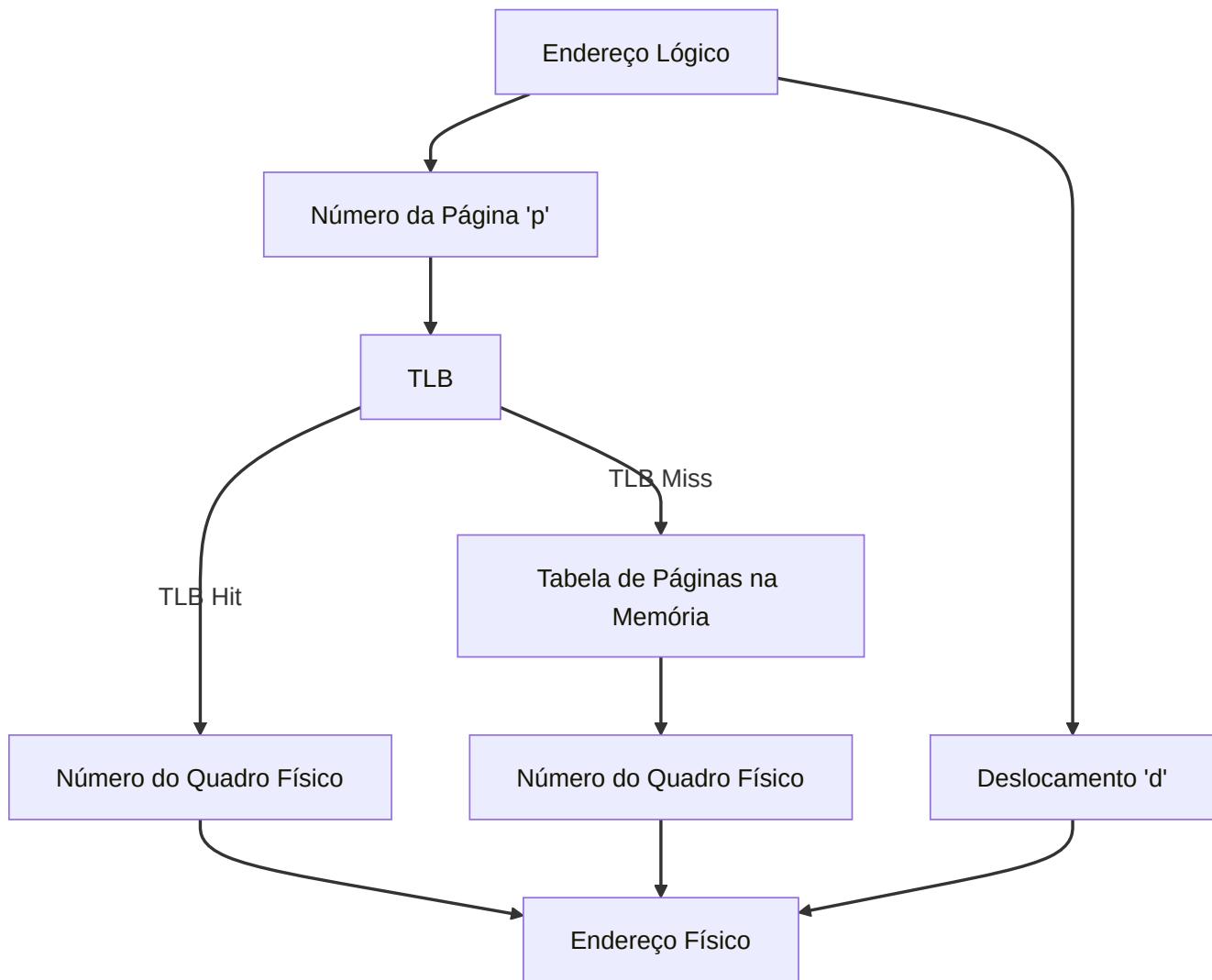
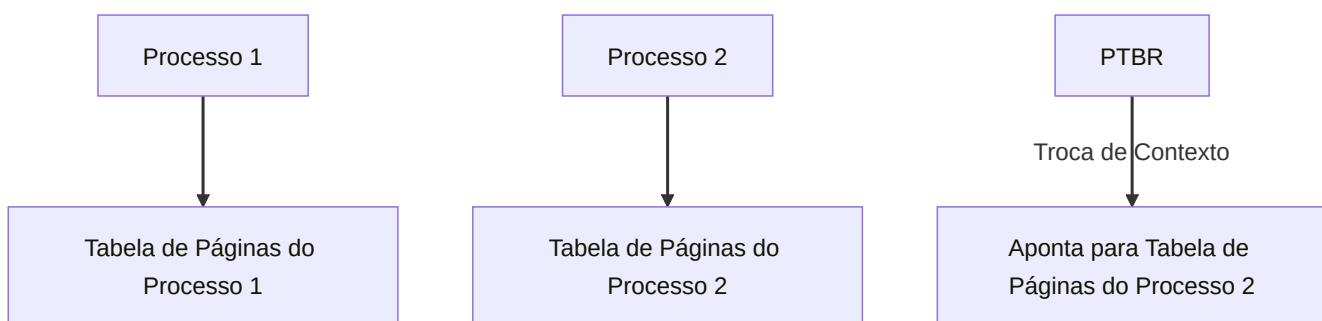


Diagrama 2: Troca de Contexto com PTBR



5. Resumo

Conceito	Descrição
Tabela de Páginas	Mapeia páginas lógicas para quadros físicos. Armazenada na memória principal.
PTBR	Registrador que aponta para a tabela de páginas do processo em execução.
TLB	Cache de alta velocidade para entradas da tabela de páginas.
TLB Hit	Número da página encontrado na TLB. Tradução rápida.
TLB Miss	Número da página não encontrado na TLB. Requer acesso à tabela de páginas.
ASID	Identificador único para cada processo, evitando conflitos na TLB.

Conclusão

O suporte de hardware para paginação, especialmente com o uso de **TLB** e **ASID**, é essencial para garantir eficiência e desempenho. A TLB reduz significativamente o tempo de tradução de endereços, enquanto o ASID permite que múltiplos processos compartilhem a TLB sem conflitos.

Proteção em Sistemas Paginados

Em sistemas que utilizam paginação, a **proteção de memória** é essencial para garantir que processos não acessem ou modifiquem áreas de memória que não lhes pertencem. Isso é feito por meio de **bits de proteção** associados a cada entrada na tabela de páginas. Vamos explorar como isso funciona.

1. Bits de Proteção

Cada entrada na tabela de páginas contém **bits de proteção** que definem permissões de acesso para a página correspondente. Esses bits podem incluir:

- **Leitura (R)**: Permite apenas leitura da página.
- **Escrita (W)**: Permite leitura e escrita na página.
- **Execução (X)**: Permite a execução de código na página.

Exemplo de Uso

- Se uma página estiver marcada como **somente leitura (R)**, qualquer tentativa de escrita nessa página causará uma **interceptação (trap)** do hardware, notificando o sistema operacional de uma violação de proteção.
- Páginas de código podem ser marcadas como **somente execução (X)**, impedindo que sejam modificadas ou lidas como dados.

2. Bit Válido-Inválido

Além dos bits de proteção, cada entrada na tabela de páginas possui um **bit válido-inválido**:

- **Válido (V)**: A página está no espaço de endereços lógicos do processo e pode ser acessada.
- **Inválido (I)**: A página não está no espaço de endereços lógicos do processo. Qualquer tentativa de acesso a uma página inválida causa uma **interceptação** (referência de página inválida).

Exemplo Prático

- Suponha um processo com espaço de endereços de 14 bits (0 a 16383) e páginas de 2 KB.
- O processo usa apenas os endereços de 0 a 10468, ocupando as páginas 0 a 5.

- As páginas 6 e 7 são marcadas como **inválidas**, pois estão fora do espaço de endereços do processo.
- Qualquer tentativa de acessar as páginas 6 ou 7 resultará em uma interceptação.

3. Fragmentação Interna e Proteção

A paginação pode levar à **fragmentação interna**, onde parte de uma página fica inutilizada. Isso também afeta a proteção:

- No exemplo anterior, o processo usa apenas até o endereço 10468, mas a página 5 (que cobre até 12287) é marcada como **válida**.
- Isso significa que os endereços de 10469 a 12287 são **válidos**, mas não são usados pelo processo, resultando em fragmentação interna.

4. Registro de Extensão da Tabela de Páginas (PTLR)

Para evitar o desperdício de memória com tabelas de páginas grandes, alguns sistemas usam um **PTLR (Page Table Length Register)**:

- O PTLR indica o tamanho válido da tabela de páginas para um processo.
- Cada endereço lógico é comparado com o valor do PTLR. Se o endereço estiver fora do intervalo válido, uma **interceptação** é gerada.

Vantagens

- Reduz o espaço ocupado pela tabela de páginas.
- Impede acessos a endereços fora do espaço de endereços do processo.

5. Diagramas

Diagrama 1: Bits de Proteção na Tabela de Páginas

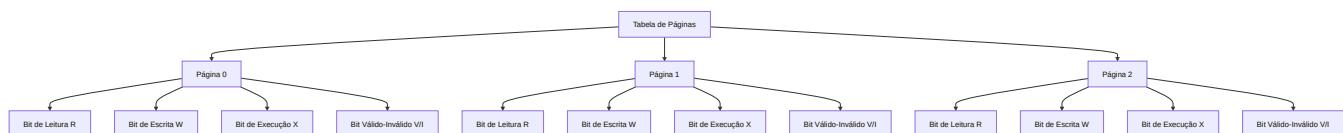
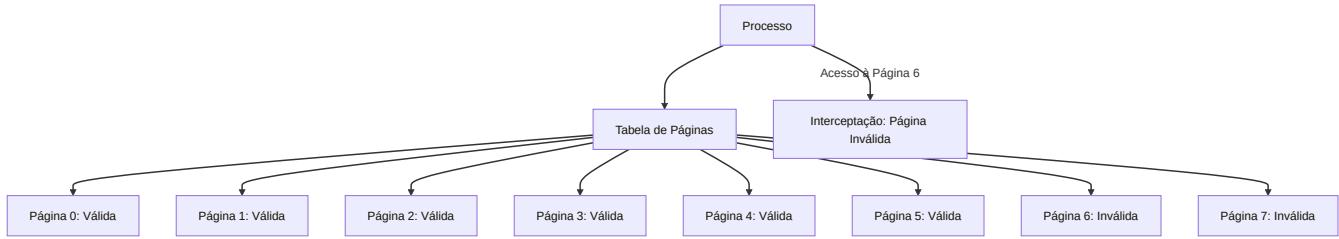


Diagrama 2: Bit Válido-Inválido em Ação



6. Resumo

Conceito	Descrição
Bits de Proteção	Controlam permissões de leitura, escrita e execução para cada página.
Bit Válido-Inválido	Indica se uma página está no espaço de endereços válido do processo.
Fragmentação Interna	Espaço não utilizado dentro de uma página válida.
PTLR	Registro que define o tamanho válido da tabela de páginas.
Interceptação	Notificação do hardware ao sistema operacional sobre violações de acesso.

A proteção em sistemas paginados é garantida por **bits de proteção** e **bits válido-inválido** na tabela de páginas. Esses mecanismos impedem acessos ilegais e garantem que processos só possam acessar suas próprias áreas de memória. O uso de **PTLR** também ajuda a otimizar o uso da memória, evitando tabelas de páginas desnecessariamente grandes.

Páginas Compartilhadas

Uma das grandes vantagens da paginação é a capacidade de **compartilhar código comum** entre processos. Isso é especialmente útil em sistemas de **tempo compartilhado**, onde vários usuários executam os mesmos programas. Vamos explorar como isso funciona e os benefícios que traz.

1. Compartilhamento de Código Reentrante

O que é Código Reentrante?

- **Código reentrante** (ou código puro) é um código que **não se modifica** durante a execução.
- Ele pode ser executado por vários processos simultaneamente, pois cada processo tem sua própria cópia dos **registradores** e **dados**, mas compartilha o mesmo código.

Exemplo: Editor de Texto

- Suponha que um editor de texto tenha:
 - **150 KB de código** (reentrante).
 - **50 KB de dados** (específicos para cada usuário).
- Em um sistema com **40 usuários**:
 - Sem compartilhamento: Cada usuário precisaria de 200 KB (150 KB de código + 50 KB de dados), totalizando **8.000 KB**.
 - Com compartilhamento: Apenas **uma cópia do código** (150 KB) é necessária, mais **40 cópias dos dados** (50 KB cada), totalizando **2.150 KB**.

Economia de Memória

- O compartilhamento de código reduz drasticamente o uso de memória.
- No exemplo acima, a economia foi de **8.000 KB** para **2.150 KB**.

2. Como Funciona o Compartilhamento de Páginas

Tabelas de Páginas

- Cada processo tem sua própria **tabela de páginas**.

- As páginas de **código compartilhado** são mapeadas para o **mesmo quadro físico** na memória.
- As páginas de **dados** são mapeadas para **quadros diferentes**, pois cada processo tem seus próprios dados.

Exemplo Visual

- **Processo 1:**
 - Página de Código 1 → Quadro 5 (compartilhado).
 - Página de Dados 1 → Quadro 10.
- **Processo 2:**
 - Página de Código 1 → Quadro 5 (compartilhado).
 - Página de Dados 1 → Quadro 15.
- **Processo 3:**
 - Página de Código 1 → Quadro 5 (compartilhado).
 - Página de Dados 1 → Quadro 20.

3. Benefícios do Compartilhamento de Páginas

1. Economia de Memória:

- Reduz a quantidade de memória necessária para executar múltiplas instâncias do mesmo programa.

2. Desempenho:

- Menos memória usada significa mais espaço para outros processos, melhorando a eficiência do sistema.

3. Facilidade de Atualização:

- Se o código compartilhado precisar ser atualizado, apenas uma cópia precisa ser modificada.

4. Aplicações Comuns de Páginas Compartilhadas

- **Editores de Texto:** Como no exemplo anterior.

- **Compiladores:** Vários usuários podem compilar programas simultaneamente usando o mesmo código do compilador.
- **Bibliotecas de Tempo de Execução:** Funções comuns, como manipulação de strings ou operações matemáticas, podem ser compartilhadas.
- **Sistemas de Banco de Dados:** Múltiplas instâncias de um banco de dados podem compartilhar o código do sistema.

5. Diagramas

Diagrama 1: Compartilhamento de Código

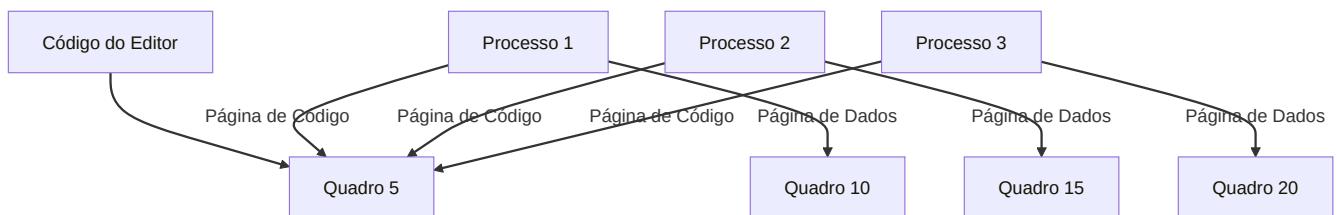
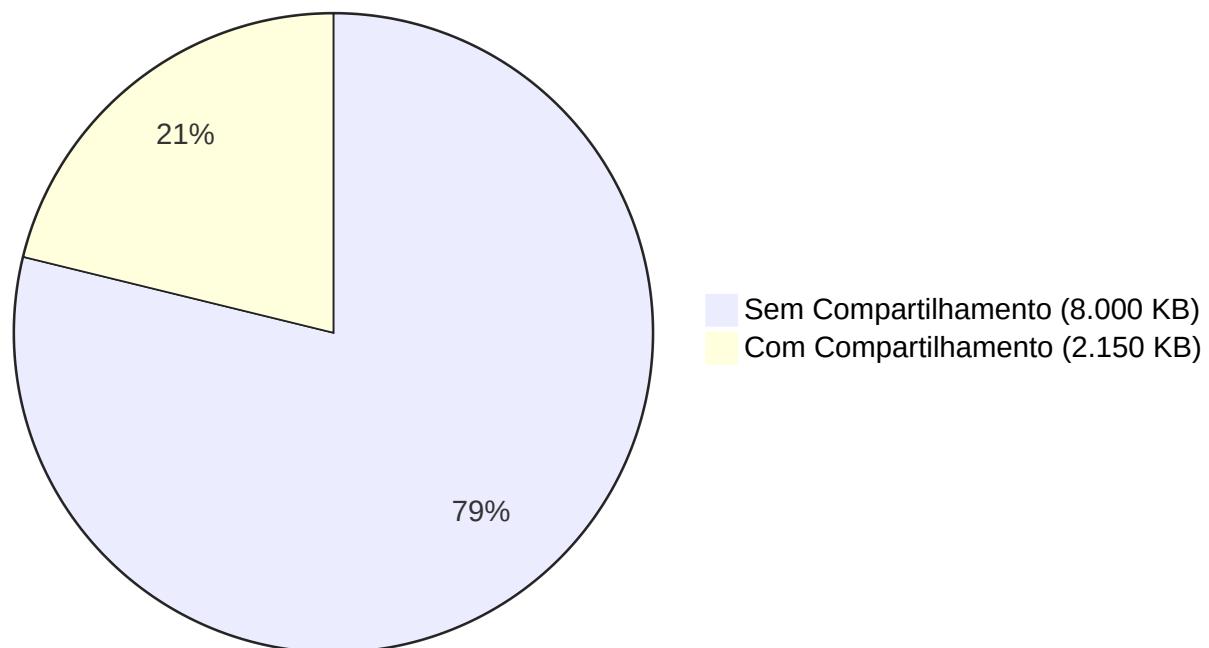


Diagrama 2: Economia de Memória

Uso de Memória



6. Resumo

Conceito	Descrição
Código Reentrante	Código que não se modifica e pode ser compartilhado entre processos.
Compartilhamento de Páginas	Múltiplos processos usam a mesma cópia física do código.
Economia de Memória	Reduz o uso de memória ao compartilhar código comum.
Aplicações Comuns	Editores, compiladores, bibliotecas, sistemas de banco de dados.

O compartilhamento de páginas é uma técnica poderosa que permite a **reutilização de código** entre processos, reduzindo o uso de memória e melhorando a eficiência do sistema. Isso é especialmente útil em ambientes de tempo compartilhado, onde múltiplos usuários executam os mesmos programas.

6.6 Estrutura da Tabela de Página

A **tabela de páginas** é uma estrutura essencial para a implementação da paginação. No entanto, dependendo do tamanho do espaço de endereçamento e da quantidade de memória disponível, diferentes técnicas são usadas para organizar e gerenciar a tabela de páginas. Vamos explorar três abordagens comuns:

1. Paginação Hierárquica.
2. Tabelas de Página com Hash.
3. Tabelas de Página Invertidas.

1. Paginação Hierárquica

O que é?

- A **paginação hierárquica** divide a tabela de páginas em **níveis** (ou camadas), criando uma estrutura em árvore.
- Cada nível da hierarquia mapeia uma parte do endereço lógico.

Como Funciona?

- O endereço lógico é dividido em várias partes, cada uma correspondendo a um nível da tabela de páginas.
- O primeiro nível aponta para uma tabela de segundo nível, que pode apontar para uma tabela de terceiro nível, e assim por diante.
- Apenas as tabelas de páginas necessárias são carregadas na memória, economizando espaço.

Exemplo: Paginação de Dois Níveis

- O endereço lógico é dividido em:
 - Número da Página de Primeiro Nível (p1).
 - Número da Página de Segundo Nível (p2).
 - Deslocamento (d).
- A tabela de primeiro nível contém ponteiros para tabelas de segundo nível.

- A tabela de segundo nível contém os números dos quadros físicos.

Vantagens

- Reduz o espaço ocupado pela tabela de páginas, pois apenas as partes necessárias são carregadas.
- Adequado para sistemas com espaços de endereçamento grandes.

Desvantagens

- Aumenta o tempo de acesso à memória, pois múltiplos níveis precisam ser consultados.

2. Tabelas de Página com Hash

O que é?

- A **tabela de páginas com hash** usa uma **função de hash** para mapear números de páginas lógicas em entradas da tabela de páginas.
- É útil para sistemas com espaços de endereçamento muito grandes, onde a tabela de páginas tradicional seria inviável.

Como Funciona?

- O número da página lógica é passado para uma função de hash, que retorna um índice na tabela de páginas.
- Cada entrada na tabela de hash contém:
 - **Número da Página.**
 - **Número do Quadro Físico.**
- Em caso de colisões (quando dois números de página são mapeados para o mesmo índice), uma lista encadeada ou outra técnica de resolução de colisões é usada.

Vantagens

- Reduz o tamanho da tabela de páginas, especialmente em sistemas com espaços de endereçamento grandes.
- Eficiente em termos de espaço.

Desvantagens

- Pode haver colisões, aumentando o tempo de acesso.
- Complexidade adicional para lidar com colisões.

3. Tabelas de Página Invertidas

O que é?

- Na **tabela de páginas invertida**, há apenas **uma tabela de páginas para todo o sistema**, em vez de uma tabela por processo.
- Cada entrada na tabela contém:
 - **Número do Quadro Físico**.
 - **Identificador do Processo (PID)**.
 - **Número da Página Lógica**.

Como Funciona?

- Quando um processo acessa uma página, o sistema usa o **PID** e o **número da página lógica** para encontrar a entrada correspondente na tabela invertida.
- A entrada contém o número do quadro físico, que é combinado com o deslocamento para formar o endereço físico.

Vantagens

- Reduz drasticamente o espaço ocupado pela tabela de páginas, pois há apenas uma tabela para todo o sistema.
- Adequado para sistemas com muitos processos e grandes espaços de endereçamento.

Desvantagens

- Acesso mais lento, pois a tabela invertida precisa ser pesquisada para cada referência à memória.
- Complexidade adicional para gerenciar a tabela.

Comparação das Técnicas

Técnica	Vantagens	Desvantagens
Paginação Hierárquica	Economiza espaço; adequada para grandes espaços de endereçamento.	Aumenta o tempo de acesso devido a múltiplos níveis.
Tabelas de Página com Hash	Eficiente em termos de espaço; útil para espaços de endereçamento muito grandes.	Colisões podem aumentar o tempo de acesso.
Tabelas de Página Invertidas	Reduz o espaço da tabela; uma única tabela para todo o sistema.	Acesso mais lento; complexidade de gerenciamento.

Diagramas

Diagrama 1: Paginação Hierárquica

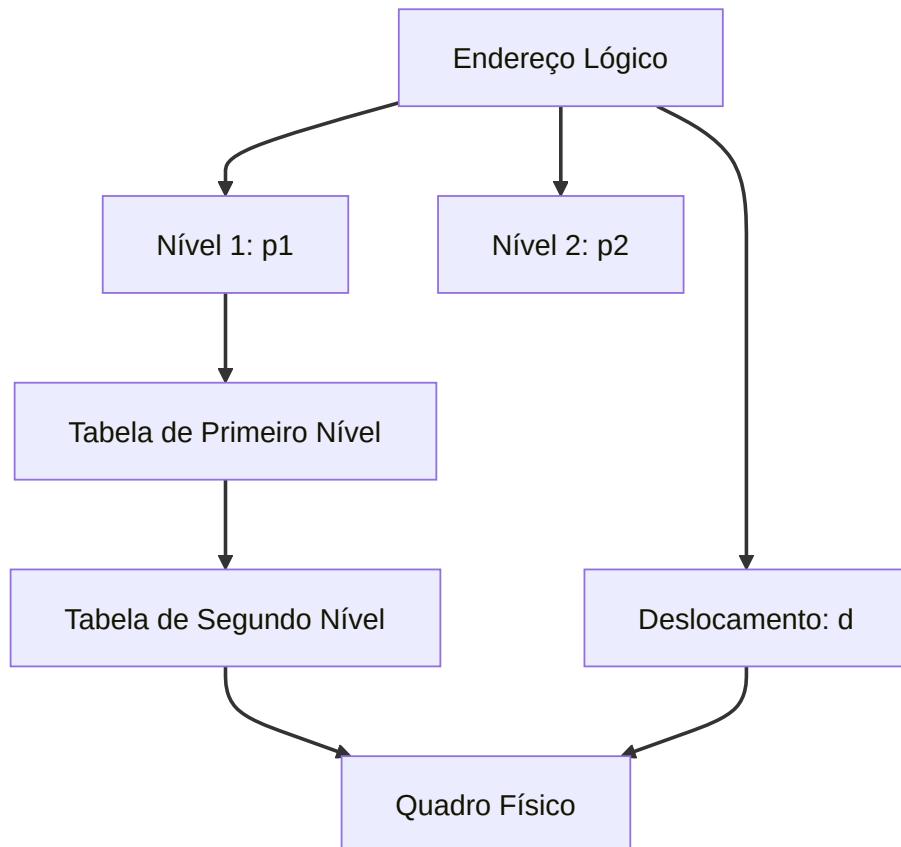


Diagrama 2: Tabela de Página com Hash

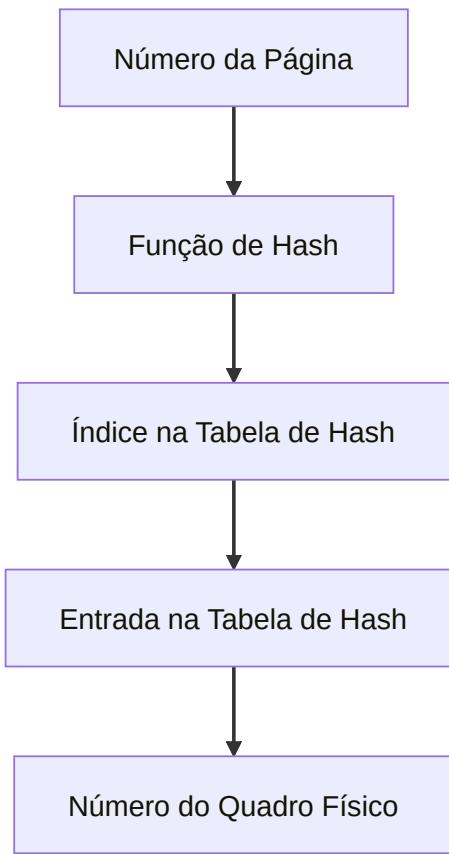
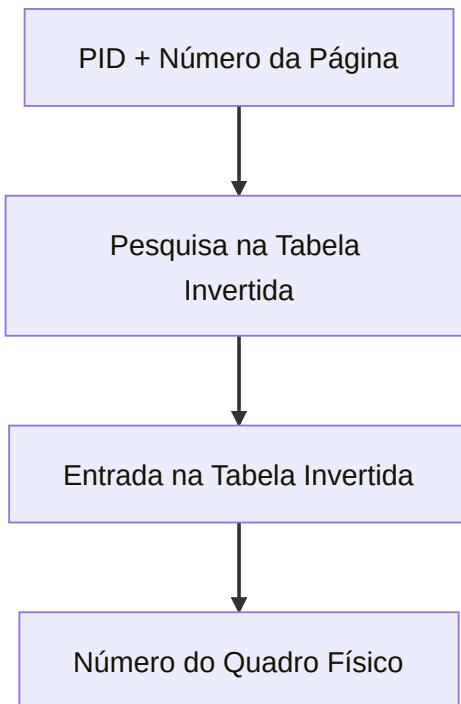


Diagrama 3: Tabela de Página Invertida



A escolha da estrutura da tabela de páginas depende das necessidades do sistema:

- **Paginação Hierárquica** é ideal para sistemas com grandes espaços de endereçamento.
- **Tabelas de Página com Hash** são eficientes em termos de espaço, mas podem sofrer com colisões.
- **Tabelas de Página Invertidas** reduzem o espaço da tabela, mas têm um custo maior em termos de tempo de acesso.

Paginação Hierárquica

A **paginação hierárquica** é uma técnica usada para gerenciar tabelas de páginas em sistemas com **grandes espaços de endereçamento** (por exemplo, 32 bits ou 64 bits). Em vez de armazenar toda a tabela de páginas de forma contígua na memória, ela é dividida em **níveis** (ou camadas), criando uma estrutura em árvore. Isso reduz o espaço ocupado pela tabela de páginas e permite que apenas as partes necessárias sejam carregadas na memória.

1. Motivação

Em sistemas com espaços de endereçamento grandes, a tabela de páginas pode se tornar **excessivamente grande**. Por exemplo:

- **Espaço de endereçamento de 32 bits com páginas de 4 KB:**
 - Número de páginas: $2^{32}/2^{12} = 2^{20}$ (1 milhão de páginas).
 - Tamanho da tabela de páginas: $2^{20} \times 4 \text{ bytes} = 4 \text{ MB}$ por processo.
- **Espaço de endereçamento de 64 bits com páginas de 4 KB:**
 - Número de páginas: $2^{64}/2^{12} = 2^{52}$ (um número enorme de páginas).
 - Tamanho da tabela de páginas: $2^{52} \times 4 \text{ bytes}$ (impraticável).

A paginação hierárquica resolve esse problema dividindo a tabela de páginas em **níveis menores**.

2. Paginação de Dois Níveis

Como Funciona?

- O endereço lógico é dividido em três partes:
 1. **Número da Página de Primeiro Nível (p1)**: Índice na tabela de páginas de primeiro nível.
 2. **Número da Página de Segundo Nível (p2)**: Índice na tabela de páginas de segundo nível.
 3. **Deslocamento (d)**: Posição dentro da página.

Exemplo: Espaço de 32 bits com páginas de 4 KB

- **p1**: 10 bits (índice na tabela de primeiro nível).
- **p2**: 10 bits (índice na tabela de segundo nível).

- **d**: 12 bits (deslocamento dentro da página).

Tradução de Endereço

1. O **p1** é usado para indexar a tabela de primeiro nível, que contém ponteiros para tabelas de segundo nível.
2. O **p2** é usado para indexar a tabela de segundo nível, que contém o número do quadro físico.
3. O **d** é combinado com o número do quadro físico para formar o endereço físico.

Vantagens

- Reduz o espaço ocupado pela tabela de páginas, pois apenas as partes necessárias são carregadas.
- Adequado para sistemas com espaços de endereçamento grandes.

Desvantagens

- Aumenta o tempo de acesso à memória, pois múltiplos níveis precisam ser consultados.

3. Paginação de Três ou Mais Níveis

Para sistemas com espaços de endereçamento ainda maiores (por exemplo, 64 bits), a paginação de dois níveis pode não ser suficiente. Nesses casos, a tabela de páginas é dividida em **três ou mais níveis**.

Exemplo: Espaço de 64 bits com páginas de 4 KB

- **p1**: 12 bits (índice na tabela de primeiro nível).
- **p2**: 12 bits (índice na tabela de segundo nível).
- **p3**: 12 bits (índice na tabela de terceiro nível).
- **d**: 12 bits (deslocamento dentro da página).

Vantagens

- Permite gerenciar espaços de endereçamento extremamente grandes.
- Apenas as partes necessárias da tabela de páginas são carregadas na memória.

Desvantagens

- Aumenta ainda mais o tempo de acesso à memória, pois mais níveis precisam ser consultados.

4. Exemplo: Arquitetura VAX

A arquitetura VAX usa uma variação da paginação hierárquica:

- O espaço de endereçamento de 32 bits é dividido em **quatro seções**:
 - Cada seção tem 2^{30} bytes.
 - Os dois primeiros bits do endereço lógico indicam a seção.
 - Os 21 bits seguintes representam o número da página.
 - Os 9 bits finais representam o deslocamento.

Tradução de Endereço

1. O número da seção (**s**) é usado para selecionar a tabela de páginas correspondente.
2. O número da página (**p**) é usado para indexar a tabela de páginas.
3. O deslocamento (**d**) é combinado com o número do quadro físico para formar o endereço físico.

5. Comparação de Níveis de Paginação

Níveis de Paginação	Espaço de Endereçamento	Tamanho da Tabela de Páginas	Tempo de Acesso
Dois Níveis	32 bits	4 MB por processo	2 acessos à memória
Três Níveis	64 bits	Reduzido, mas ainda grande	3 acessos à memória
Quatro Níveis	64 bits	Reduzido ainda mais	4 acessos à memória

6. Diagramas

Diagrama 1: Paginação de Dois Níveis

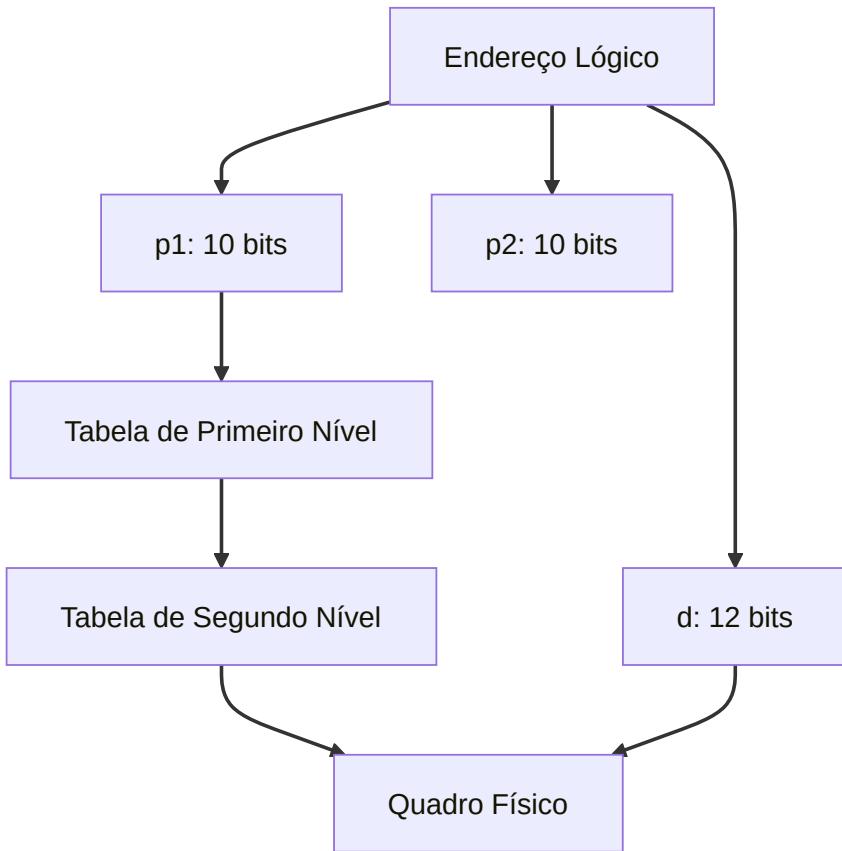
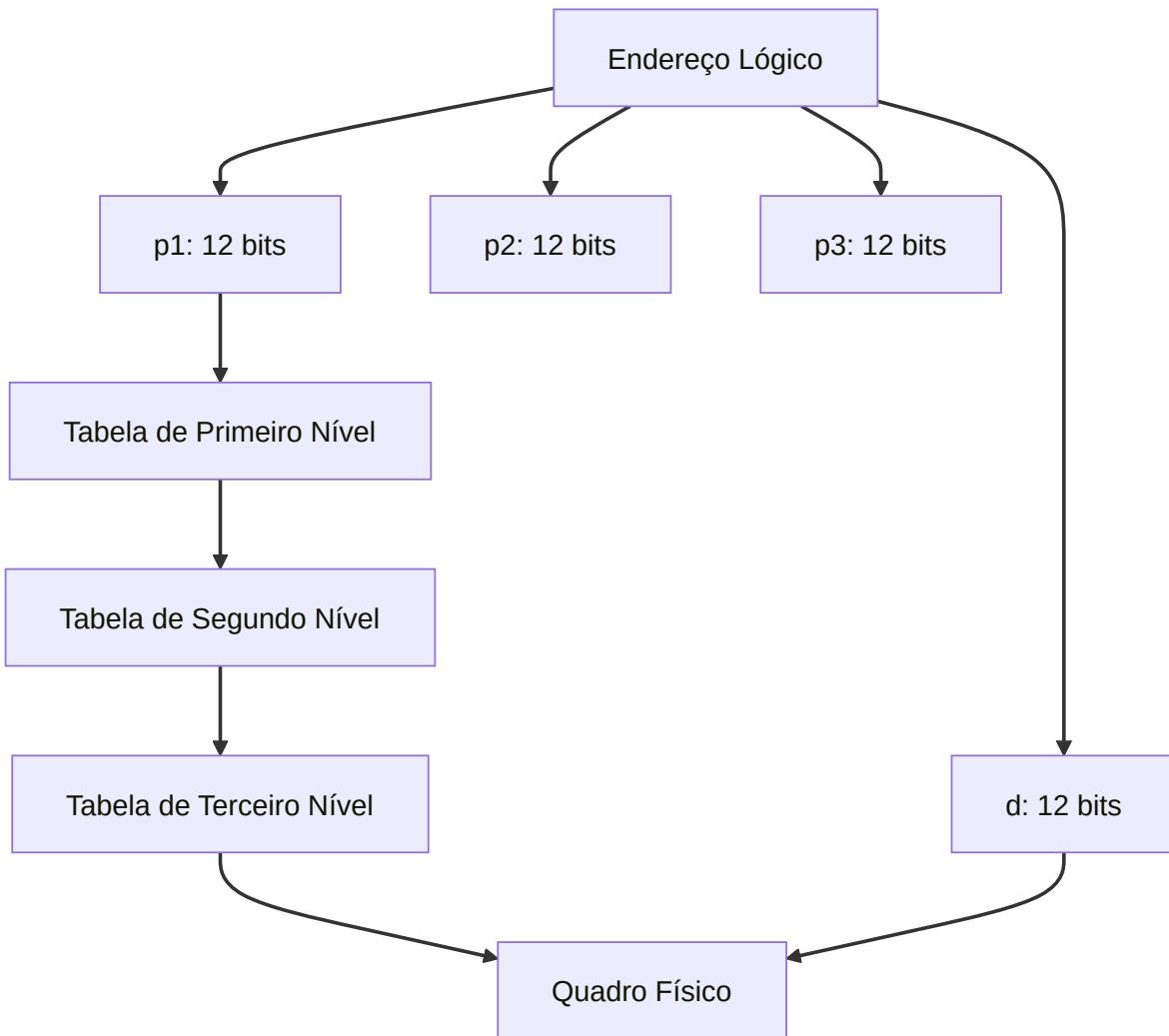


Diagrama 2: Paginação de Três Níveis



A **paginação hierárquica** é uma técnica essencial para gerenciar tabelas de páginas em sistemas com grandes espaços de endereçamento. Ela divide a tabela de páginas em **níveis**, reduzindo o espaço ocupado e permitindo que apenas as partes necessárias sejam carregadas na memória. No entanto, o aumento no número de níveis também aumenta o tempo de acesso à memória, o que pode ser um desafio em sistemas com espaços de endereçamento muito grandes (como 64 bits).

Tabelas de Página Invertidas

As **tabelas de página invertidas** são uma abordagem alternativa para gerenciar tabelas de páginas em sistemas com grandes espaços de endereçamento. Diferente das tabelas de página tradicionais, que possuem uma entrada para cada página virtual, as tabelas invertidas possuem uma entrada para cada **quadro físico** da memória. Isso reduz drasticamente o tamanho da tabela de páginas, mas introduz desafios em termos de desempenho e implementação.

1. O que é uma Tabela de Página Invertida?

- **Tabela Tradicional:** Cada processo tem sua própria tabela de páginas, com uma entrada para cada página virtual.
- **Tabela Invertida:** Há apenas **uma tabela de páginas para todo o sistema**, com uma entrada para cada quadro físico da memória.

Estrutura da Tabela Invertida

Cada entrada na tabela invertida contém:

- **Identificador do Processo (PID):** Identifica o processo que está usando a página.
- **Número da Página Virtual:** Identifica a página lógica associada ao quadro físico.
- **Outras Informações:** Bits de proteção, bits válido-inválido, etc.

2. Como Funciona?

Tradução de Endereço

1. O endereço virtual é dividido em:

- **PID:** Identificador do processo.
- **Número da Página Virtual:** Identifica a página lógica.
- **Deslocamento:** Posição dentro da página.

2. A tabela invertida é pesquisada para encontrar uma entrada que corresponda ao **<PID, Número da Página Virtual>**.

3. Se a entrada for encontrada, o **número do quadro físico** é combinado com o deslocamento para formar o endereço físico.

4. Se a entrada não for encontrada, ocorre uma **falha de página** (acesso ilegal).

Exemplo

- Endereço Virtual: <PID=1, Número da Página=5, Deslocamento=100>.
- Tabela Invertida:
 - Entrada 1: <PID=1, Número da Página=5, Quadro Físico=10>.
 - Entrada 2: <PID=2, Número da Página=3, Quadro Físico=15>.
- Resultado: Endereço Físico = <Quadro Físico=10, Deslocamento=100>.

3. Vantagens

1. Economia de Memória:

- A tabela invertida tem apenas uma entrada por quadro físico, em vez de uma entrada por página virtual.
- Reduz o espaço ocupado pela tabela de páginas, especialmente em sistemas com muitos processos.

2. Simplicidade:

- Há apenas uma tabela de páginas para todo o sistema, simplificando o gerenciamento.

4. Desafios

1. Tempo de Pesquisa:

- A tabela invertida precisa ser pesquisada para cada referência à memória.
- Isso pode ser lento, especialmente em sistemas com muita memória física.

2. Memória Compartilhada:

- Em tabelas invertidas, uma página física só pode ser mapeada para um único endereço virtual.
- Isso dificulta a implementação de **memória compartilhada**, onde múltiplos processos precisam mapear a mesma página física.

5. Soluções para Melhorar o Desempenho

Tabela Hash

- Uma **tabela hash** é usada para acelerar a pesquisa na tabela invertida.
- O endereço virtual (PID + Número da Página) é passado para uma função de hash, que retorna um índice na tabela invertida.
- Isso reduz o número de entradas que precisam ser pesquisadas.

TLB (Translation Lookaside Buffer)

- A TLB é usada para armazenar entradas recentes da tabela invertida.
- Se o endereço virtual estiver na TLB, a tradução é feita rapidamente, sem consultar a tabela invertida.

6. Exemplo de Uso

IBM RT

- O sistema IBM RT usa tabelas de página invertidas.
- Cada endereço virtual é uma tripla: <PID, Número da Página, Deslocamento>.
- A tabela invertida é pesquisada para encontrar uma correspondência com <PID, Número da Página>.

UltraSPARC e PowerPC

- Essas arquiteturas também utilizam tabelas de página invertidas.
- Elas armazenam um **identificador de espaço de endereço (ASID)** em cada entrada para garantir que as páginas de diferentes processos não entrem em conflito.

7. Comparação com Tabelas de Página Tradicionais

Característica	Tabela Tradicional	Tabela Invertida
Tamanho da Tabela	Grande (uma entrada por página virtual).	Pequeno (uma entrada por quadro físico).
Complexidade	Mais complexa (uma tabela por processo).	Mais simples (uma tabela para todo o sistema).
Desempenho	Mais rápido (acesso direto à tabela).	Mais lento (pesquisa necessária).
Memória Compartilhada	Fácil de implementar.	Difícil de implementar.

8. Diagramas

Diagrama 1: Tabela de Página Invertida

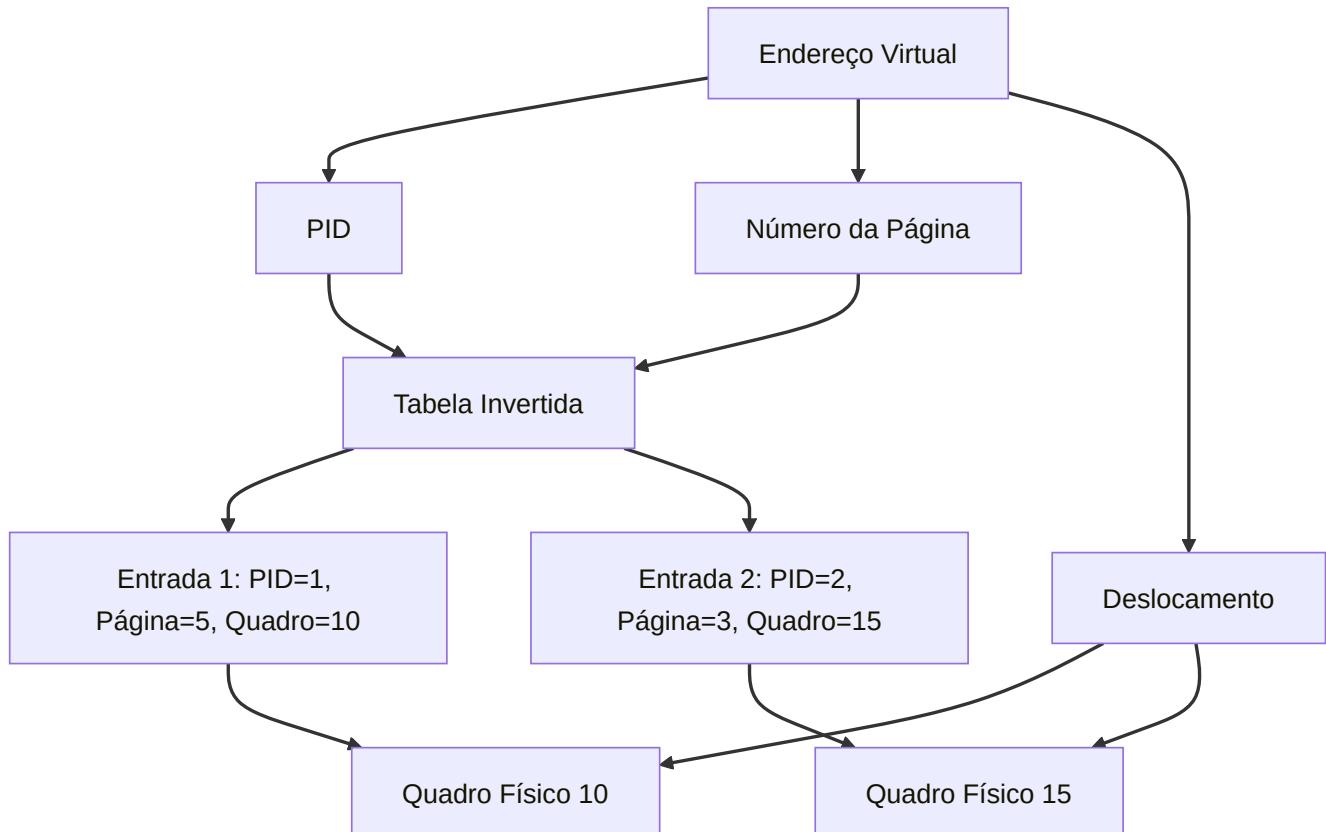
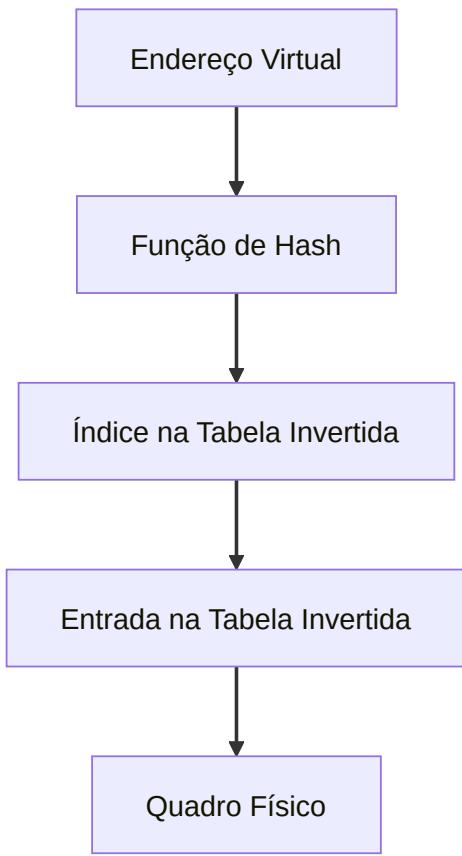


Diagrama 2: Uso de Tabela Hash



Conclusão

As **tabelas de página invertidas** são uma solução eficiente em termos de espaço para gerenciar tabelas de páginas em sistemas com grandes espaços de endereçamento. No entanto, elas introduzem desafios em termos de desempenho e implementação de memória compartilhada. O uso de **tabelas hash** e **TLB** ajuda a mitigar esses problemas, tornando as tabelas invertidas uma opção viável para sistemas modernos.

6.7 Segmentação

A **segmentação** é um esquema de gerenciamento de memória que reflete a forma como os usuários (programadores) enxergam a memória: como uma coleção de **segmentos de tamanho variável**, cada um com um propósito específico (por exemplo, código, dados, pilha, etc.). Diferente da paginação, que divide a memória em páginas de tamanho fixo, a segmentação permite que os segmentos tenham tamanhos diferentes, o que se alinha melhor com a estrutura lógica de um programa.

1. Método Básico

Visão do Usuário

- Os usuários (programadores) não pensam na memória como um array linear de bytes.
- Em vez disso, eles veem a memória como uma coleção de **segmentos**:
 - **Código**: Instruções do programa.
 - **Dados**: Variáveis globais, estruturas de dados.
 - **Pilha**: Usada para chamadas de função e armazenamento temporário.
 - **Heap**: Memória alocada dinamicamente.
- Cada segmento tem um **nome** e um **tamanho variável**.

Endereçamento Lógico

- Um endereço lógico na segmentação é representado por um **par ordenado**:
 - **Número do Segmento (s)**: Identifica o segmento.
 - **Deslocamento (d)**: Posição dentro do segmento.
- Exemplo: `<s=2, d=100>` refere-se ao byte 100 do segmento 2.

2. Hardware de Segmentação

Para implementar a segmentação, o hardware usa uma **tabela de segmentos**.

Tabela de Segmentos

- Cada entrada na tabela de segmentos contém:

- **Base:** Endereço físico inicial do segmento.
- **Limite:** Tamanho do segmento.
- O número do segmento (**s**) é usado como índice na tabela de segmentos.
- O deslocamento (**d**) deve estar dentro do limite do segmento. Caso contrário, ocorre uma **interceptação** (erro de acesso à memória).

Tradução de Endereço

1. O número do segmento (**s**) é usado para indexar a tabela de segmentos.
2. O hardware verifica se o deslocamento (**d**) é menor que o limite do segmento.
 - Se for válido, o endereço físico é calculado como: **base + d**.
 - Se for inválido, ocorre uma interceptação.

Exemplo

- **Segmento 2:**
 - Base: 4300.
 - Limite: 400.
- **Endereço Lógico:** <s=2, d=53>.
- **Endereço Físico:** \((4300 + 53 = 4353)\).

3. Vantagens da Segmentação

1. Alinhamento com a Visão do Programador:

- Reflete a estrutura lógica do programa (código, dados, pilha, etc.).
- Facilita o desenvolvimento e a depuração.

2. Proteção:

- Cada segmento pode ter permissões de acesso diferentes (leitura, escrita, execução).
- Acesso a segmentos inválidos é detectado pelo hardware.

3. Compartilhamento de Segmentos:

- Segmentos podem ser compartilhados entre processos (por exemplo, bibliotecas compartilhadas).

4. Desafios da Segmentação

1. Fragmentação Externa:

- Como os segmentos têm tamanhos variáveis, a memória pode ficar fragmentada, com pequenos espaços livres entre segmentos.
- Isso pode dificultar a alocação de novos segmentos.

2. Gerenciamento Complexo:

- Alocar e desalocar segmentos de tamanhos variáveis é mais complexo do que gerenciar páginas de tamanho fixo.

3. Desempenho:

- A tradução de endereços é mais lenta do que na paginação, pois envolve consultas à tabela de segmentos e verificações de limites.

5. Exemplo Prático

Programa em C

Um programa em C pode ser dividido nos seguintes segmentos:

1. **Código:** Instruções do programa.
2. **Variáveis Globais:** Dados compartilhados.
3. **Heap:** Memória alocada dinamicamente.
4. **Pilha:** Usada para chamadas de função.
5. **Biblioteca Padrão:** Funções da biblioteca C.

Tabela de Segmentos

Número do Segmento	Base	Limite
0 (Código)	2000	1000
1 (Variáveis Globais)	3000	500
2 (Heap)	3500	800
3 (Pilha)	4300	400
4 (Biblioteca)	4700	600

Tradução de Endereços

- **Endereço Lógico:** `<s=0, d=100>` → Endereço Físico: `\(2000 + 100 = 2100\)`.
- **Endereço Lógico:** `<s=3, d=852>` → **Erro:** O segmento 3 tem limite 400.

6. Comparação com Paginação

Característica	Paginação	Segmentação
Tamanho das Unidades	Páginas de tamanho fixo.	Segmentos de tamanho variável.
Visão do Programador	Linear (array de bytes).	Lógica (código, dados, pilha).
Fragmentação	Fragmentação interna.	Fragmentação externa.
Proteção	Por página.	Por segmento.
Desempenho	Mais rápido (tabelas simples).	Mais lento (verificação de limites).

7. Diagramas

Diagrama 1: Segmentação da Memória

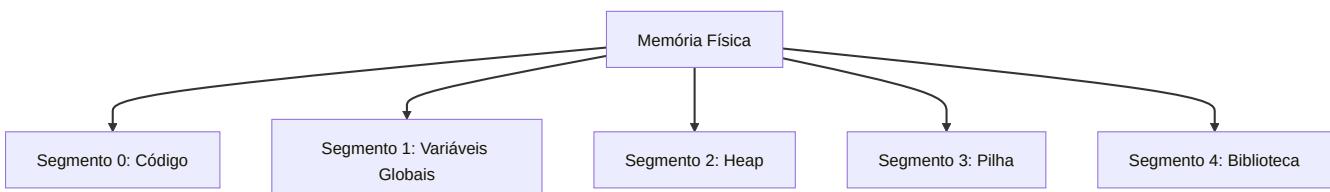
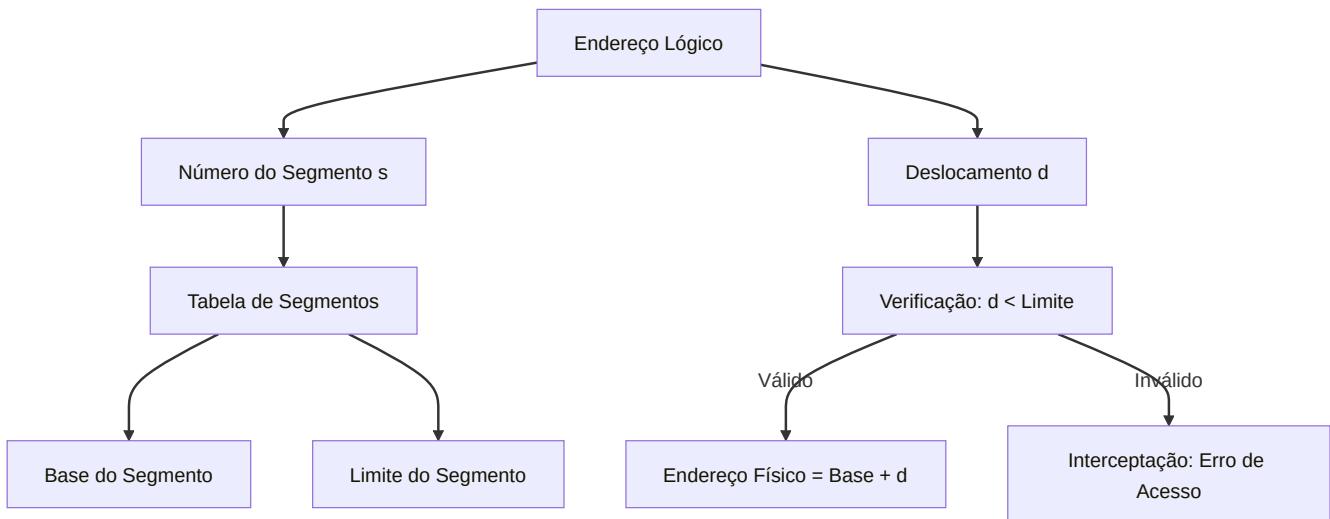


Diagrama 2: Tradução de Endereço



A **segmentação** é uma técnica de gerenciamento de memória que reflete a visão lógica do programador, dividindo a memória em segmentos de tamanho variável. Embora ofereça vantagens como alinhamento com a estrutura do programa e proteção, ela também apresenta desafios, como fragmentação externa e complexidade de gerenciamento. A escolha entre segmentação e paginação depende das necessidades do sistema e da aplicação.

6.8 Visão Geral do Gerenciamento de Memória no Pentium

O Pentium usa uma abordagem híbrida, combinando **segmentação** e **paginação**, para gerenciar a memória. Isso permite que o sistema operacional e os programas tenham uma visão lógica da memória (segmentação) enquanto mantêm o controle eficiente da memória física (paginação). Vamos detalhar cada parte.

2. Segmentação no Pentium: A Visão Lógica da Memória

A segmentação é como o programador vê a memória: dividida em **segmentos** lógicos, como código, dados, pilha, etc. No Pentium, isso é implementado da seguinte forma:

Tabelas de Segmentos

- **LDT (Local Descriptor Table)**: Armazena os descritores dos segmentos privados de um processo.
- **GDT (Global Descriptor Table)**: Armazena os descritores dos segmentos compartilhados entre processos.
- Cada **descritor de segmento** contém:
 - **Base**: O endereço físico inicial do segmento.
 - **Limite**: O tamanho do segmento.
 - **Permissões**: Proteção (leitura, escrita, execução) e tipo de acesso.

Endereço Lógico

- Um endereço lógico no Pentium é um par:
 - **Seletor (16 bits)**:
 - **Número do Segmento (13 bits)**: Índice na LDT ou GDT.
 - **Indicador de GDT/LDT (1 bit)**: Define se o segmento está na GDT ou LDT.
 - **Nível de Proteção (2 bits)**: Define o nível de privilégio (kernel, usuário, etc.).
 - **Deslocamento (32 bits)**: A posição dentro do segmento.

Tradução de Endereço

1. O **seletor** é usado para indexar a **LDT** ou **GDT** e obter o descritor do segmento.
2. O **deslocamento** é verificado contra o **limite** do segmento.
 - Se o deslocamento for menor que o limite, o endereço linear é calculado como: **base + deslocamento**.
 - Caso contrário, ocorre uma **falha de segmentação** (erro de acesso à memória).

Exemplo Prático

- Suponha que o **segmento 2** tenha:
 - Base: 4300.
 - Limite: 400.
- Um endereço lógico `<s=2, d=53>` é traduzido para:
 - Endereço linear: $\backslash(4300 + 53 = 4353\backslash)$.

3. Paginação no Pentium: A Visão Física da Memória

Após a segmentação, o endereço linear é convertido em um endereço físico usando a **paginação**. O Pentium suporta dois tamanhos de página: **4 KB** e **4 MB**.

Paginação de Dois Níveis

- O endereço linear de 32 bits é dividido em:
 - **Diretório de Página (10 bits)**: Índice na tabela de diretório de páginas.
 - **Tabela de Página (10 bits)**: Índice na tabela de páginas.
 - **Deslocamento (12 bits)**: Posição dentro da página.

Tradução de Endereço

1. O **diretório de página** é consultado para encontrar a tabela de páginas correspondente.
2. A **tabela de páginas** é consultada para encontrar o quadro físico.
3. O **deslocamento** é combinado com o quadro físico para formar o endereço físico.

Páginas de 4 MB

- Se a flag **Page Size** estiver ativada, o diretório de página aponta diretamente para um quadro de 4 MB.
- Nesse caso, os **22 bits de baixa ordem** do endereço linear são usados como deslocamento.

Exemplo Prático

- Endereço linear: 0x00402030.
 - Diretório de Página: 0x004 (índice 1 no diretório de páginas).
 - Tabela de Página: 0x020 (índice 32 na tabela de páginas).
 - Deslocamento: 0x030 (48 bytes dentro da página).
- Suponha que a tabela de páginas aponte para o quadro físico 0x1000.
- Endereço físico: \((0x1000 + 0x030 = 0x1030)\).

4. Linux no Pentium: Minimizando a Segmentação

O Linux foi projetado para ser portável entre diferentes arquiteturas, muitas das quais não suportam segmentação. Por isso, o Linux usa a segmentação de forma mínima no Pentium.

Segmentação no Linux

- O Linux usa apenas **6 segmentos**:
 - Código do Kernel:** Para executar o código do sistema operacional.
 - Dados do Kernel:** Para acessar dados do sistema operacional.
 - Código do Usuário:** Para executar código dos programas de usuário.
 - Dados do Usuário:** Para acessar dados dos programas de usuário.
 - TSS (Task State Segment):** Armazena o contexto de hardware durante trocas de contexto.
 - LDT-padrão:** Não é usado, mas pode ser substituído por uma LDT personalizada.

Paginação no Linux

- O Linux adota um modelo de **paginação de três níveis** para ser compatível com arquiteturas de 32 e 64 bits.
 - **Diretório Global:** Aponta para diretórios de páginas.

- **Diretório do Meio:** Aponta para tabelas de páginas.
- **Tabela de Página:** Aponta para quadros físicos.
- No Pentium, o **diretório do meio** é ignorado, efetivamente reduzindo o modelo para dois níveis.

Troca de Contexto

- Durante uma troca de contexto, o valor do registrador **CR3** (que aponta para o diretório de páginas) é salvo e restaurado no **TSS** da tarefa.

5. Por Que Isso Tudo Importa?

Vantagens da Segmentação

- **Visão Lógica:** Facilita o desenvolvimento, pois o programador vê a memória como segmentos (código, dados, pilha, etc.).
- **Proteção:** Cada segmento pode ter permissões diferentes (leitura, escrita, execução).

Vantagens da Paginação

- **Gerenciamento Eficiente:** Permite alocar memória física em blocos de tamanho fixo (páginas).
- **Redução de Fragmentação:** A paginação evita a fragmentação externa.

Desafios

- **Complexidade:** A combinação de segmentação e paginação aumenta a complexidade do hardware e do software.
- **Overhead:** A tradução de endereços envolve múltiplas consultas a tabelas, o que pode impactar o desempenho.

6. Diagramas para Visualizar o Processo

Diagrama 1: Tradução de Endereço no Pentium

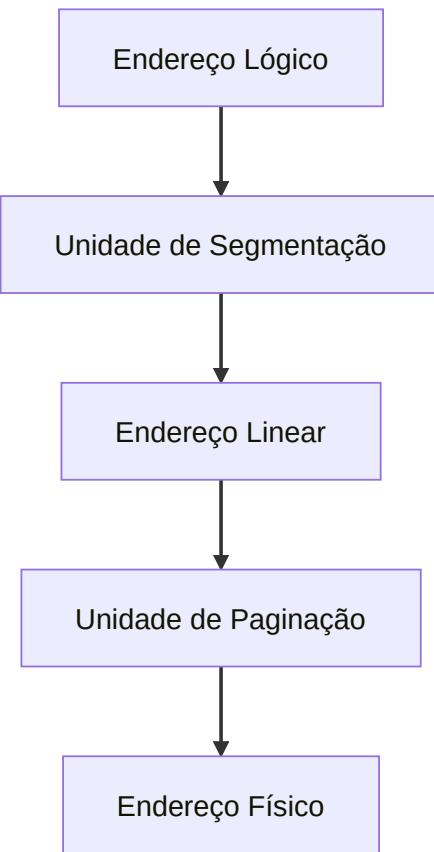


Diagrama 2: Segmentação no Pentium

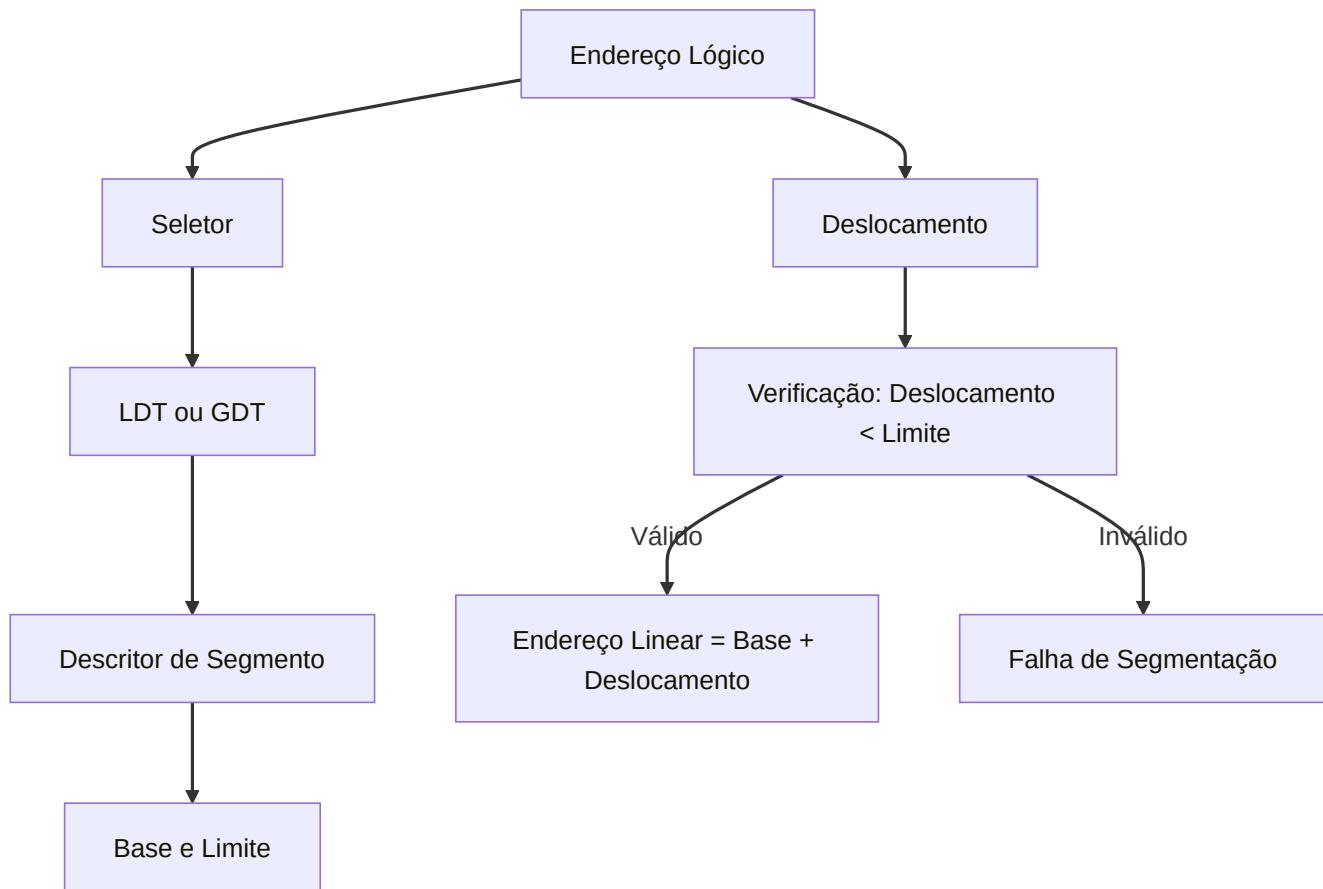
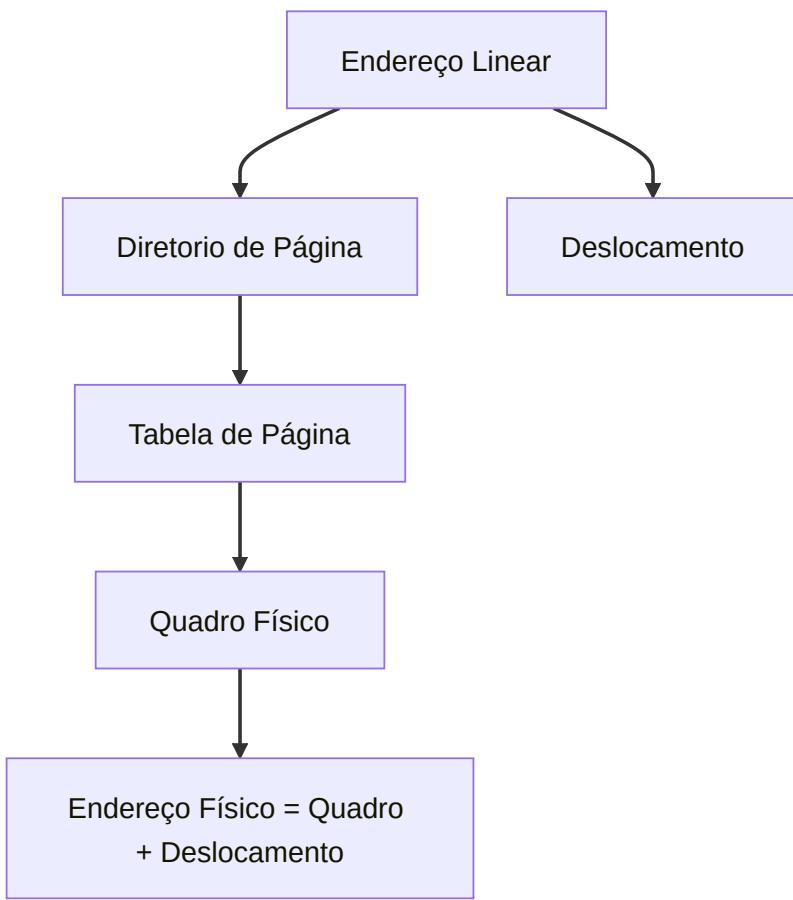


Diagrama 3: Paginação no Pentium



A arquitetura Intel Pentium combina **segmentação** e **paginação** para oferecer uma solução poderosa e flexível para o gerenciamento de memória. A segmentação fornece uma visão lógica da memória, alinhada com a forma como os programadores pensam, enquanto a paginação gerencia a memória física de forma eficiente. O **Linux** utiliza essas funcionalidades de forma mínima, priorizando a portabilidade e a simplicidade. Essa combinação permite que sistemas modernos sejam robustos, seguros e eficientes, mesmo em cenários complexos.

Exercícios Práticos 6

Exercício 6.1: Cite duas diferenças entre endereços lógicos e físicos.

Explicação:

- **Endereço Lógico:** É o endereço que o programa usa. Ele é gerado pela CPU e existe no "mundo" do programa. O programa não sabe onde ele está realmente na memória física.
- **Endereço Físico:** É o endereço real na memória RAM. Ele é o local onde os dados ou instruções estão armazenados fisicamente.

Diferenças:

1. Visibilidade:

- O **endereço lógico** é visível para o programa.
- O **endereço físico** é visível apenas para o hardware (CPU e sistema operacional).

2. Mapeamento:

- O endereço lógico é mapeado para o endereço físico pelo sistema operacional (usando tabelas de páginas ou segmentação).
- O endereço físico é fixo e não muda, enquanto o endereço lógico pode variar dependendo do processo.

Exemplo:

- Imagine que você está em um prédio com vários apartamentos (memória física). O endereço lógico é como o número do apartamento que você vê no seu contrato, enquanto o endereço físico é a localização real do apartamento no prédio.

Exercício 6.2: Discussão sobre registradores de base-limite para código e dados.

Explicação:

- O sistema tem dois pares de registradores de base-limite:

- Um para **código** (instruções).
- Outro para **dados**.
- Esses registradores são **somente leitura**, o que permite que programas sejam compartilhados entre usuários.

Vantagens:

1. Compartilhamento de Código:

- Vários usuários podem rodar o mesmo programa sem precisar de cópias separadas do código.

2. Proteção:

- O código é somente leitura, então ninguém pode alterá-lo acidentalmente ou maliciosamente.

Desvantagens:

1. Complexidade:

- O sistema precisa gerenciar dois pares de registradores, o que aumenta a complexidade do hardware.

2. Limitação de Flexibilidade:

- Se o programa precisar modificar o código (por exemplo, em linguagens que permitem auto-modificação), isso não será possível.

Exemplo:

- Imagine que você tem um livro (código) que várias pessoas podem ler, mas ninguém pode escrever nele. Isso é bom para compartilhar, mas ruim se você quiser fazer anotações.

Exercício 6.3: Por que os tamanhos de página são sempre potências de 2?

Explicação:

- Os tamanhos de página são potências de 2 (por exemplo, 4 KB, 8 KB, 16 KB) porque isso facilita o cálculo de endereços e a divisão da memória.

Motivos:

1. Facilidade de Cálculo:

- Em binário, potências de 2 são representadas por um único bit "1" seguido de zeros (ex: 4 KB = 2^{12}).
- Isso simplifica a divisão do endereço lógico em número da página e deslocamento.

2. Alinhamento de Memória:

- Potências de 2 garantem que as páginas comecem e terminem em endereços alinhados, o que melhora a eficiência do hardware.

Exemplo:

- Se o tamanho da página for 4 KB (2^{12}), os últimos 12 bits do endereço lógico são o deslocamento, e os bits restantes são o número da página. Isso é fácil de calcular em hardware.

Exercício 6.4: Espaço de endereços lógicos e físicos.

Dados:

- Páginas lógicas: 64.
- Tamanho de cada página: 1.024 words.
- Quadros físicos: 32.

a) Quantos bits existem no endereço lógico?

- Número de páginas: $64 = 2^6 \rightarrow$ 6 bits para o número da página.
- Tamanho da página: $1.024 \text{ words} = 2^{10} \rightarrow$ 10 bits para o deslocamento.
- Total: $6 + 10 = 16 \text{ bits}$.

b) Quantos bits existem no endereço físico?

- Número de quadros: $32 = 2^5 \rightarrow$ 5 bits para o número do quadro.
- Tamanho do quadro: $1.024 \text{ words} = 2^{10} \rightarrow$ 10 bits para o deslocamento.
- Total: $5 + 10 = 15 \text{ bits}$.

Exercício 6.5: Compartilhamento de páginas.

Explicação:

- Se duas entradas na tabela de páginas apontam para o mesmo quadro físico, isso significa que duas páginas lógicas compartilham a mesma página física.

Vantagens:

1. Economia de Memória:

- Reduz a quantidade de memória usada, pois a mesma página física é compartilhada.

2. Cópia Rápida:

- Para copiar uma grande quantidade de memória, basta apontar as entradas da tabela de páginas para o mesmo quadro físico, sem precisar copiar os dados.

Efeito de Atualização:

- Se um byte em uma página for atualizado, a outra página também será afetada, pois ambas compartilham o mesmo quadro físico.

Exemplo:

- Imagine que duas pessoas estão lendo o mesmo livro. Se uma pessoa escrever algo no livro, a outra pessoa verá a alteração.

Exercício 6.6: Compartilhamento de segmentos entre processos.

Explicação:

- Um segmento pode pertencer ao espaço de endereços de dois processos se ambos mapearem o mesmo segmento físico em suas tabelas de segmentos.

Mecanismo:

1. Tabela de Segmentos Compartilhada:

- Ambos os processos têm entradas em suas tabelas de segmentos que apontam para o mesmo segmento físico.

2. Proteção:

- O sistema operacional garante que os processos tenham permissão para acessar o segmento compartilhado.

Exemplo:

- Dois programas podem compartilhar uma biblioteca de funções (como uma biblioteca matemática), sem precisar de cópias separadas.

Exercício 6.7: Compartilhamento de segmentos e páginas.

a) Compartilhamento de segmentos com vínculo estático:

- O sistema pode usar um **identificador único** para cada segmento compartilhado, em vez de depender do número do segmento. Assim, processos podem compartilhar segmentos sem precisar ter os mesmos números de segmento.

b) Compartilhamento de páginas:

- O sistema pode usar uma **tabela de páginas invertida**, onde várias entradas podem apontar para o mesmo quadro físico. Isso permite que páginas sejam compartilhadas sem precisar ter os mesmos números de página.

Exercício 6.8: Proteção de memória no IBM/370.

Explicação:

- O IBM/370 usa **chaves de 4 bits** para proteger a memória. Cada bloco de 2 KB tem uma chave, e a CPU também tem uma chave. Acesso é permitido apenas se as chaves forem iguais ou se uma delas for zero.

Esquemas compatíveis:

- a) **Máquina pura:** Sim, pois a proteção é feita por hardware.
- b) **Sistema monousuário:** Sim, mas a proteção é desnecessária.
- c) **Multiprogramação com processos fixos:** Sim, cada processo pode ter uma chave única.
- d) **Multiprogramação com processos variáveis:** Sim, mas a gestão de chaves pode ser complexa.
- e) **Paginação:** Sim, as chaves podem ser usadas para proteger páginas.

- f) Segmentação: Sim, as chaves podem ser usadas para proteger segmentos.

Bibliografia

SILBERSCHATZ, Abraham; GALVIN, Peter B.; GAGNE, Greg. Sistemas Operacionais com Java. 8. ed. Rio de Janeiro: Elsevier, 2010.

TutorialsPoint - Operating System. TUTORIALSPOINT. Operating System Tutorial. Disponível em:
https://www.tutorialspoint.com/operating_system/index.htm
(https://www.tutorialspoint.com/operating_system/index.htm).

TutorialsPoint - OS Overview. TUTORIALSPOINT. Operating System - Overview. Disponível em:
https://www.tutorialspoint.com/operating_system/os_overview.htm
(https://www.tutorialspoint.com/operating_system/os_overview.htm).

GeeksforGeeks - Operating Systems. GEEKSFORGEEKS. Operating Systems. Disponível em:
<https://www.geeksforgeeks.org/operating-systems/> (<https://www.geeksforgeeks.org/operating-systems/>).

GEEKSFORGEEKS. What is an Operating System? Disponível em:
<https://www.geeksforgeeks.org/what-is-an-operating-system/>
(<https://www.geeksforgeeks.org/what-is-an-operating-system/>).

TechTarget - Operating System (OS) TECHTARGET. What is an Operating System (OS)? Disponível em: [https://www.techtarget.com/whatis/definition/operating-system-OS#:~:text=An%20operating%20system%20\(OS\)%20is,application%20program%20interface%20\(API\)](https://www.techtarget.com/whatis/definition/operating-system-OS#:~:text=An%20operating%20system%20(OS)%20is,application%20program%20interface%20(API))
([https://www.techtarget.com/whatis/definition/operating-system-OS#:~:text=An%20operating%20system%20\(OS\)%20is,application%20program%20interface%20\(API\)](https://www.techtarget.com/whatis/definition/operating-system-OS#:~:text=An%20operating%20system%20(OS)%20is,application%20program%20interface%20(API))).