O Guia Definitivo para Sobreviver aos Sistemas Operacionais

O Guia Definitivo para Sobreviver aos Sistemas Operacionais

Opala Corp.

Mr Punk da Silva

Table of Contents

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 PDF é estruturado para proporcionar uma compreensão acessível e prática dos sistemas operacionais, abrangendo desde a teoria básica até exercícios práticos que reforçam o aprendizado. Se você é estudante, entusiasta ou profissional na área de tecnologia, este material foi feito para você! Prepare-se para se aprofundar neste universo fascinante, desenvolver novas habilidades e, quem sabe, até se divertir no processo.

Introdução

Um **Sistema Operacional** é uma interface que interliga o **hardware** e o **software**, fazendo o gerenciamento dos dois mundos do computador.

> A inexistência, de um **SO** resulta em um impedimento de uma interação natural com o computador.

Como todo grande projeto complexo caímos em uma operação em que deve ser bem definida em como o **SO** irá trabalhar, quais são seus requisitos.

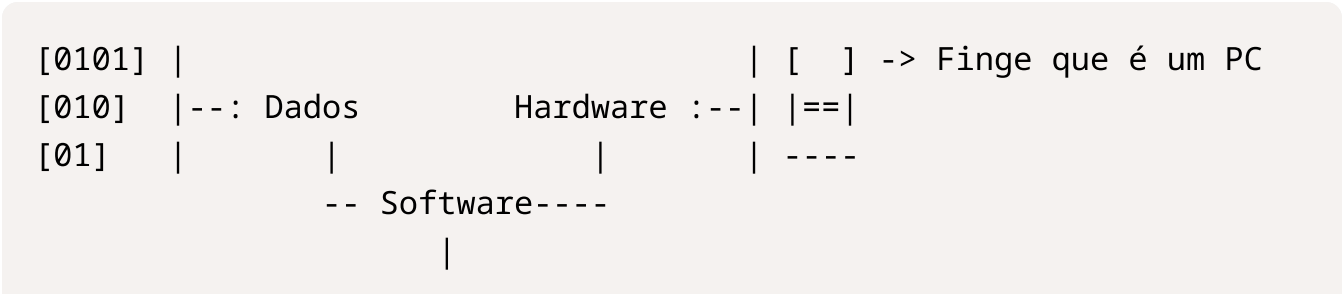
E o **SO** tem algumas funções básicas que é:

- dar partida no sistema
- gerenciar memoria
- gerenciar dispositivos E/S

O que os Sistemas Operacionais fazem

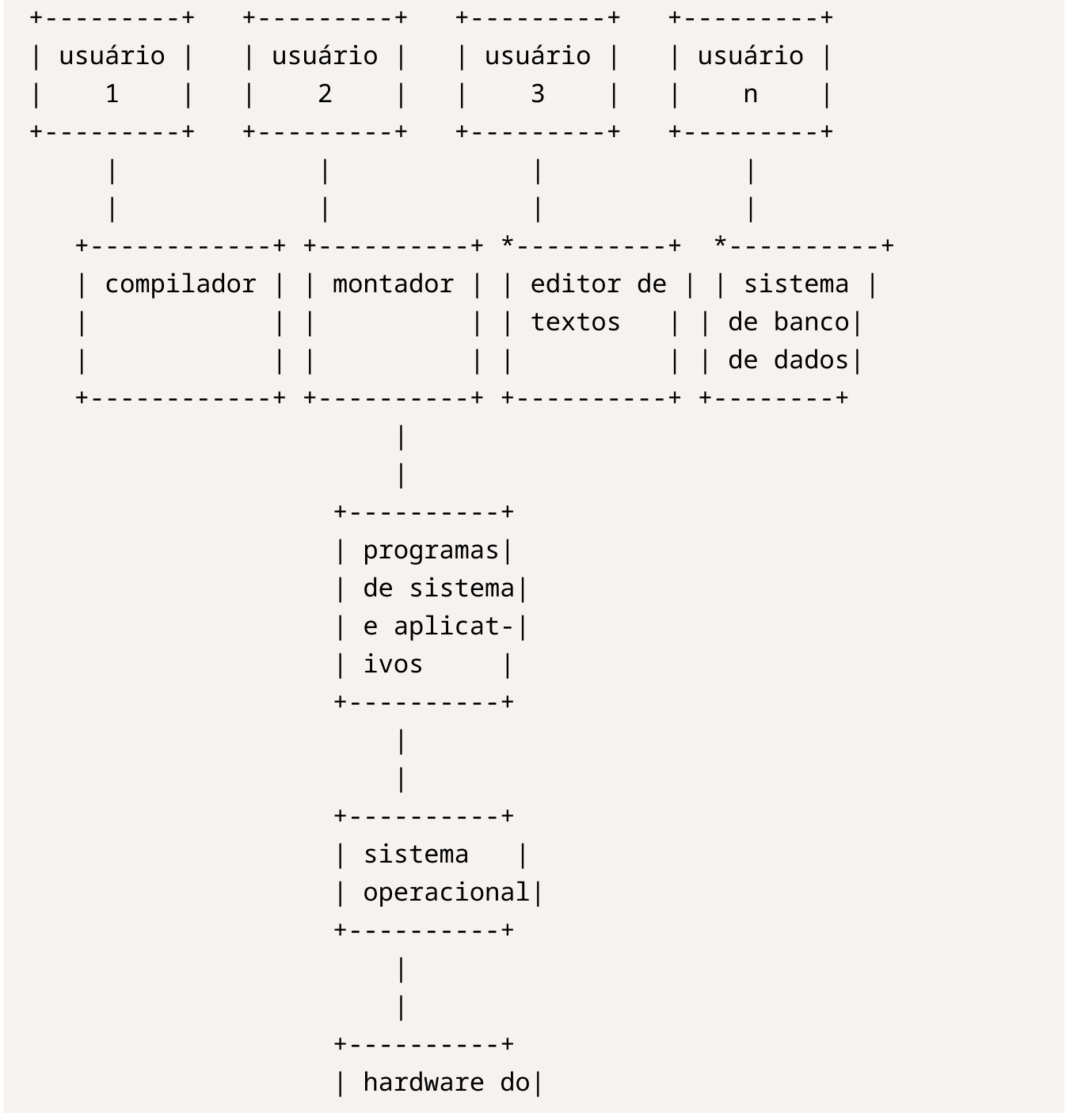
Um **sistema computadorizado** ou só computador, pode ser *dividido em quatro partes*:

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

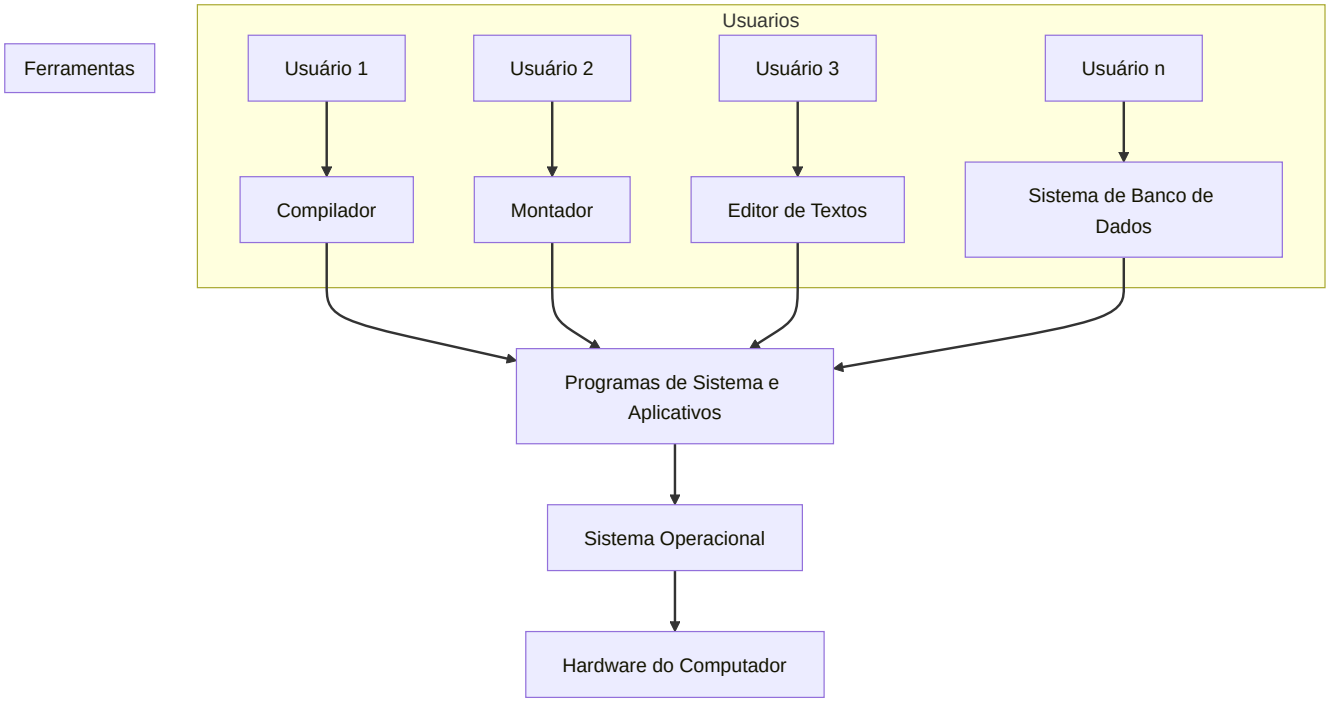


```
                    |
                 |--------|
                 |  WIN95 |
                 |--------|
```

- **Exemplo de Funcionamento de um Sistema Operacional**



```
                    | computador |
                    +-----------+
```


Ferramentas
Usuarios
Usuário 1
Usuário 2
Usuário 3
Usuário n
Compilador
Montador
Editor de Textos
Sistema de Banco de Dados
Programas de Sistema e Aplicativos
Sistema Operacional
Hardware do Computador

Operação do computador

Ao desligar o computador e ligar, o que será que acontece? Como ele "chama" o Sistema Operacional.

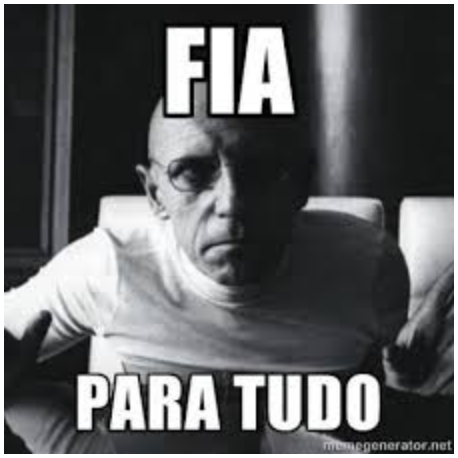
Para o computador começar a funcionar ele chama um programa básico, chamado de **bootstrap** normalmente está alocado na memoria apenas de leitura (**ROM**) ou então é salva na memoria de somente leitura apagável programavelmente **EEPROM**.

Esse programa é conhecido como **Firmware** porque está instalado diretamente no hardware, assim ele inicializa todos os aspectos do sistema que vão dos registradores da CPU até a dispositivos e conteúdo na memoria.

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

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

Assim que a CPU recebe alguma interrupção ela para o que está fazendo:



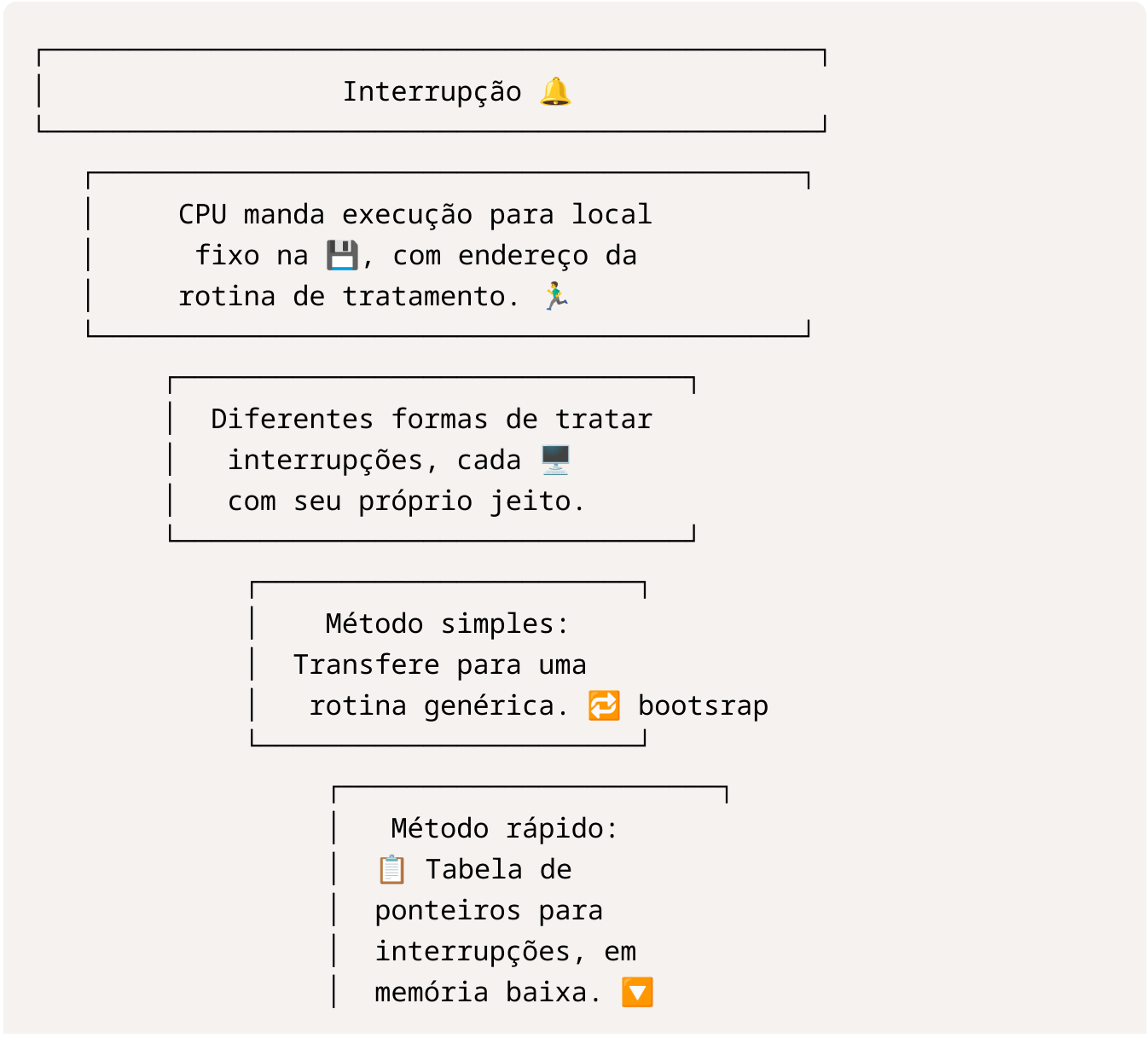
Img

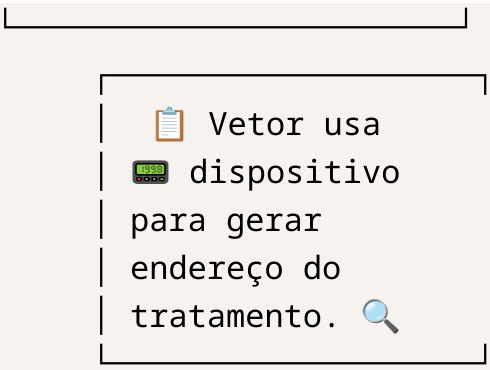
E a CPU manda a execução para uma **locação fixa de memoria**, tal locação contem o **endereço inicial** que está localizada a rotina para **atender a essa interrupção.**

Essas **interrupções** podem ser tratadas de diferentes maneiras e cada computador possui seu próprio mecanismo. Um método simples para isso, seria tratar a transferência chamando uma rotina generica.

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 memoria baixa**, sendo ela a primeira parte ou locação da memoria.

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:



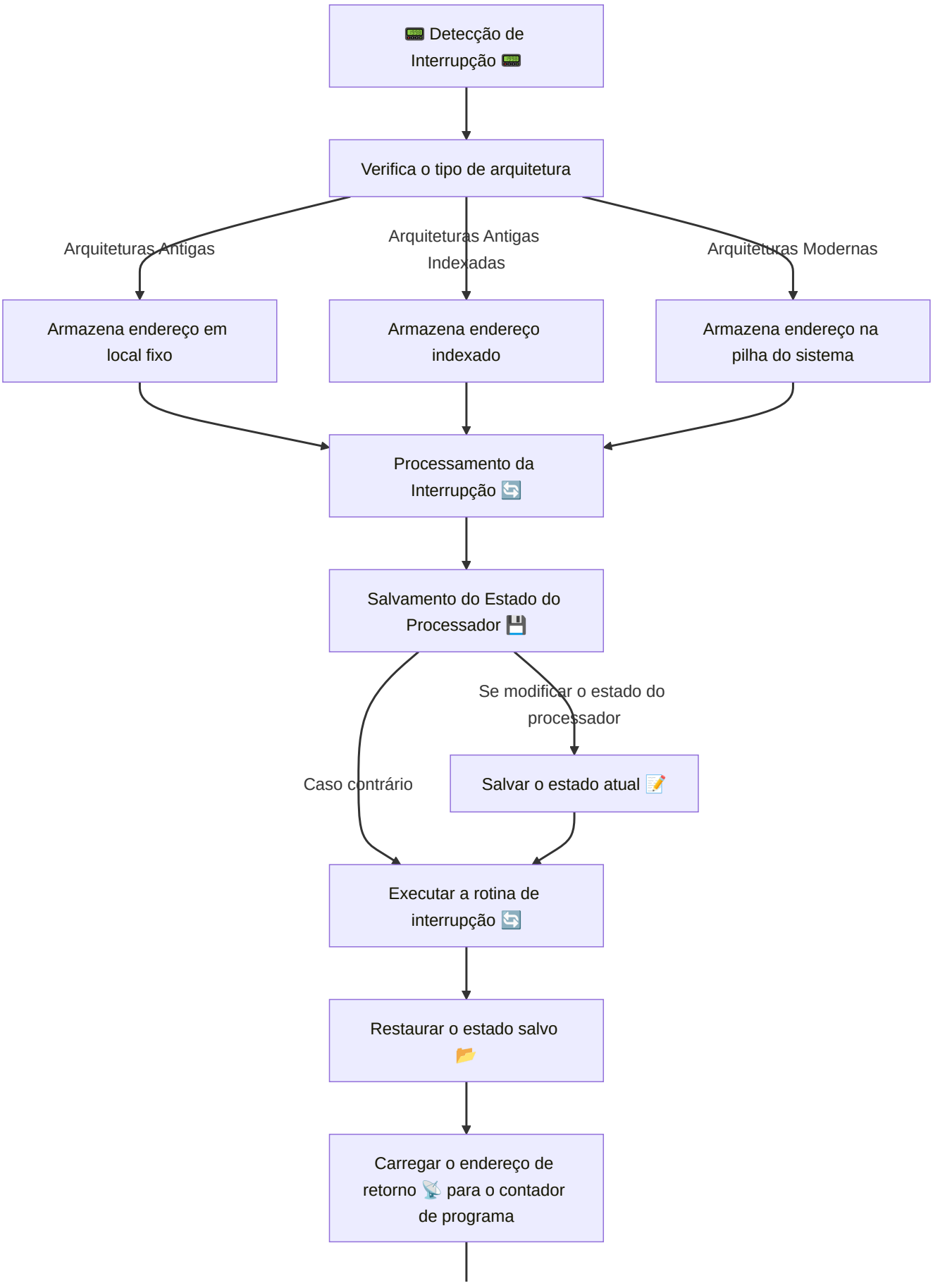


A arquitetura de interrupção **precisa salvar o endereço da instrução interrompida**, em projetos:

- 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**;

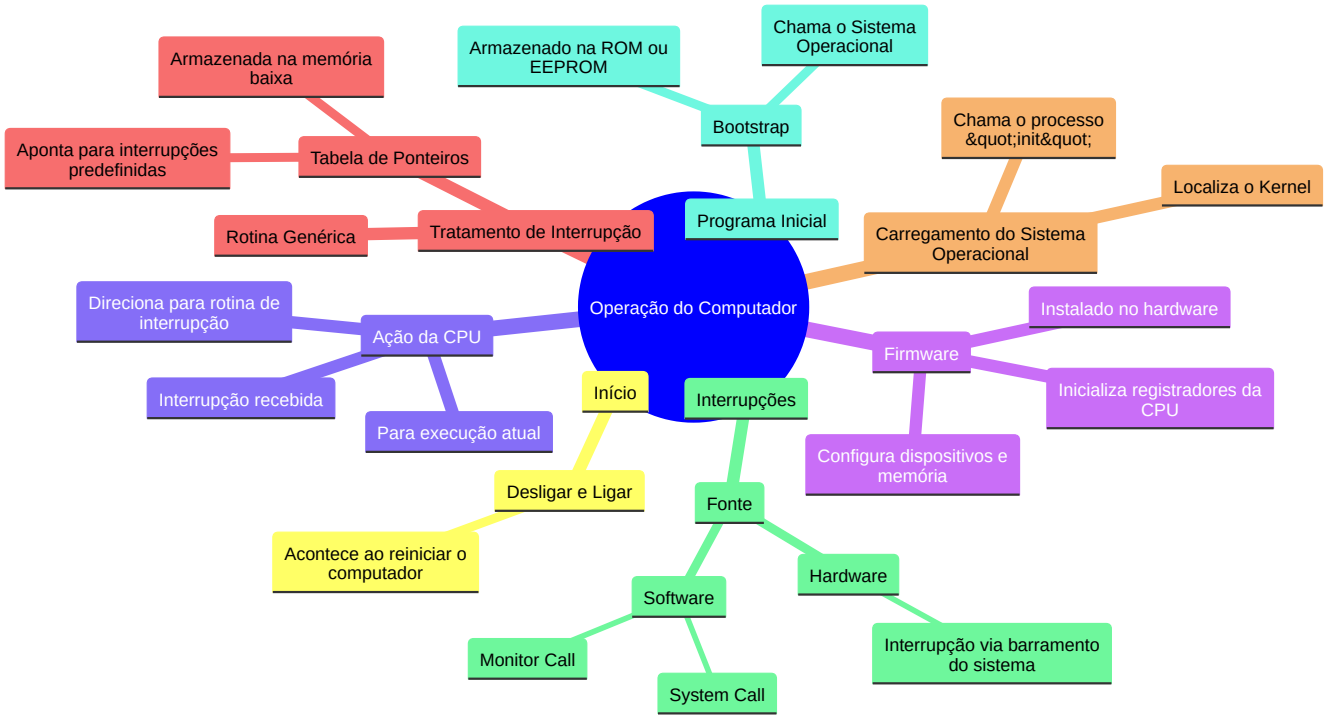
Se a rotina de interrupção precisar modificar algum estado do processador, por exemplo alterando os valores do **registrador**:

- 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:


📟 Detecção de Interrupção 📟
Verifica o tipo de arquitetura
Arquiteturas Antigas
Arquiteturas Antigas Indexadas
Arquiteturas Modernas
Armazena endereço em local fixo
Armazena endereço indexado
Armazena endereço na pilha do sistema
Processamento da Interrupção 🔄
Salvamento do Estado do Processador 💾
Se modificar o estado do processador
Caso contrário
Salvar o estado atual 📝
Executar a rotina de interrupção 🔄
Restaurar o estado salvo 📂
Carregar o endereço de retorno 📡 para o contador de programa

Processador continua a execução 🚀

Diagrama

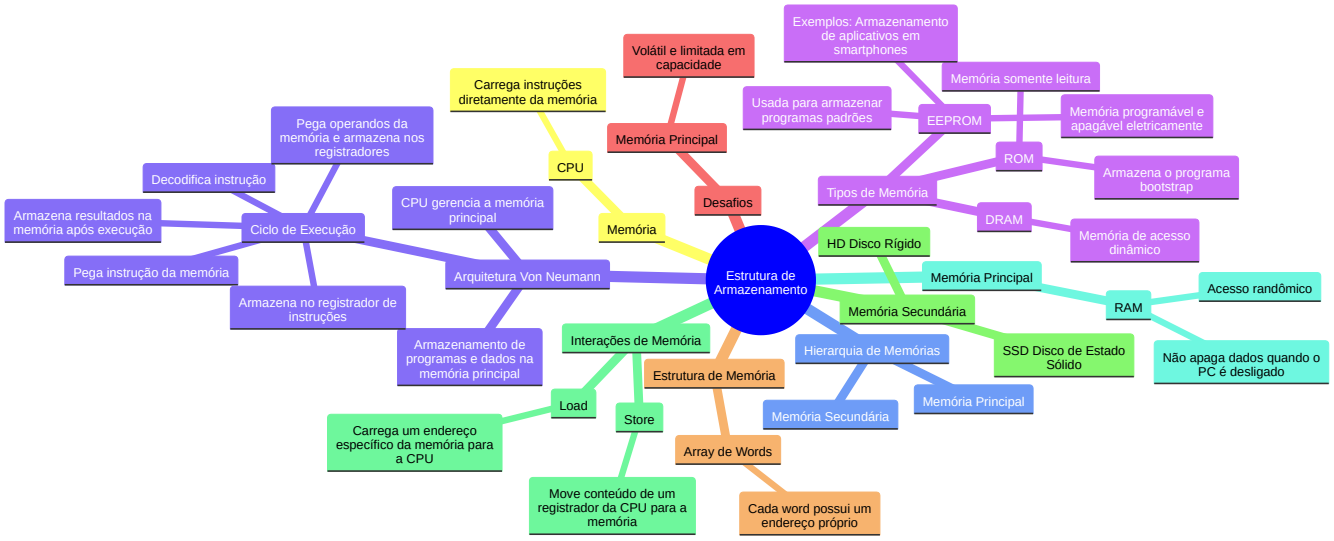


Estrutura de Armazenamento

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**.

Diagrama



> 🔗 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 modificado 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**.

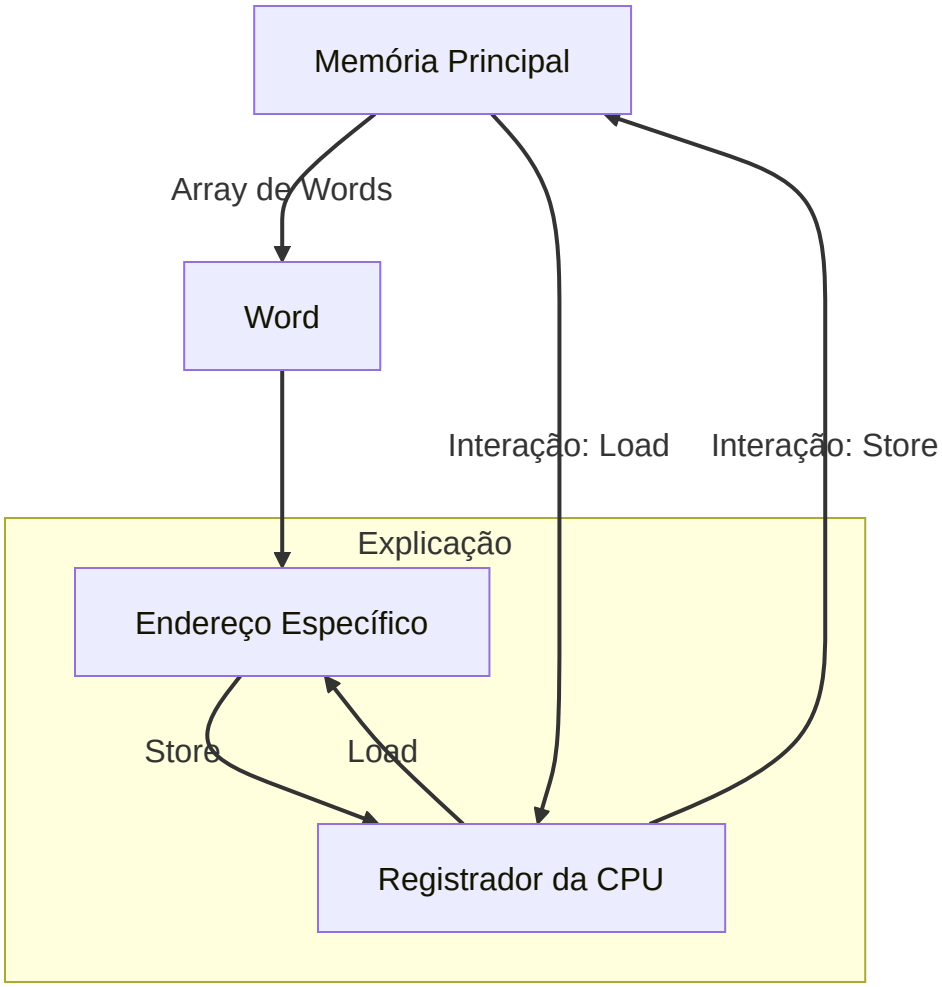


Ilustração de um esquema sobre instruções da CPU (`load` e `store`)

> 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.

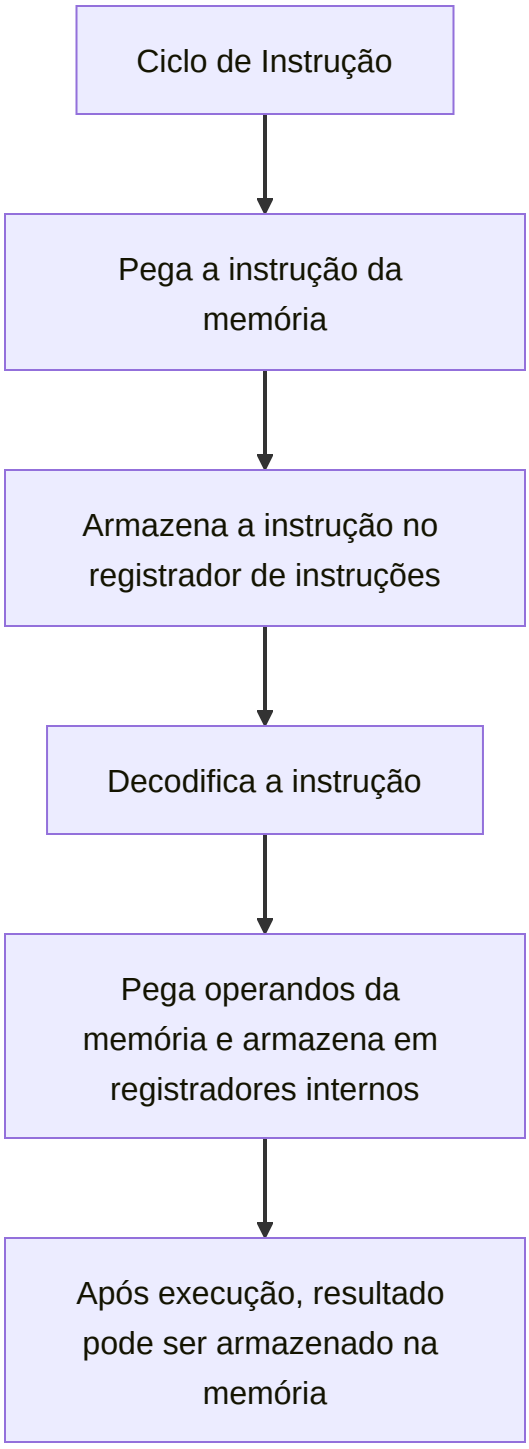
A arquitetura mais usada nos computadores modernos é a de **Von Neumann**. Essa arquitetura funciona da seguinte forma:

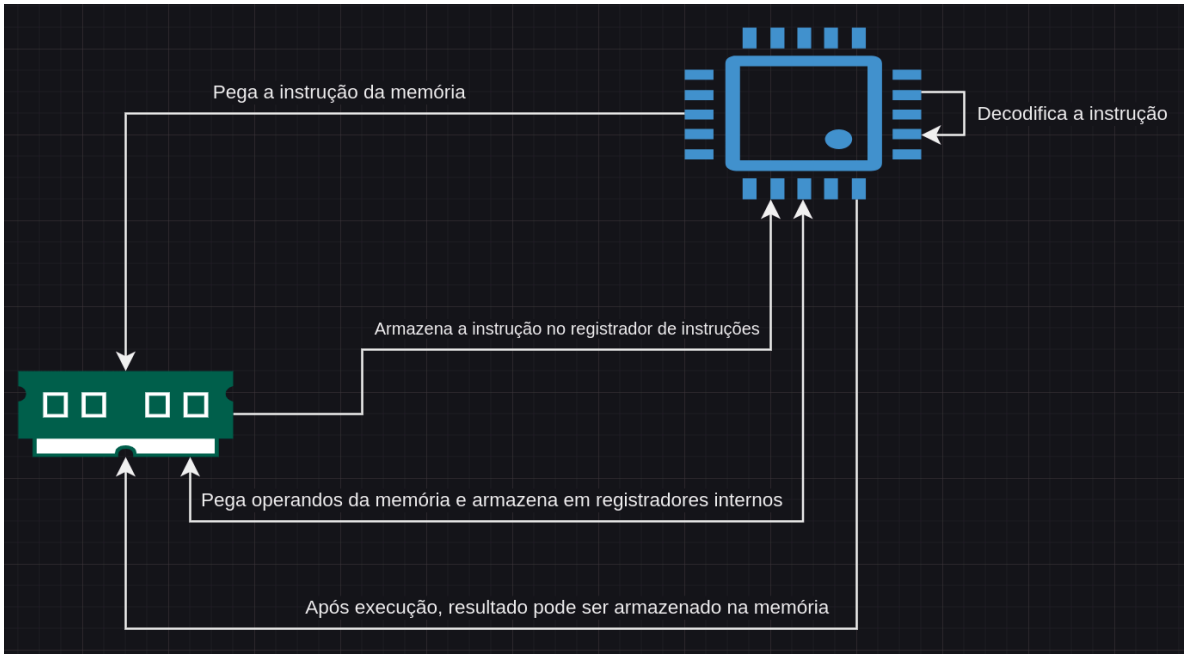
- Programas e dados são armazenados na memória principal.
- A CPU gerencia a memória principal.

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

1. Pega a instrução da memória.
2. Armazena essa instrução no **registrador 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


Ciclo de Instrução
Pega a instrução da memória
Armazena a instrução no registrador de instruções
Decodifica a instrução
Pega operandos da memória e armazena em registradores internos
Após execução, resultado pode ser armazenado na memória



003 - Estrutura de Armazenamento

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:

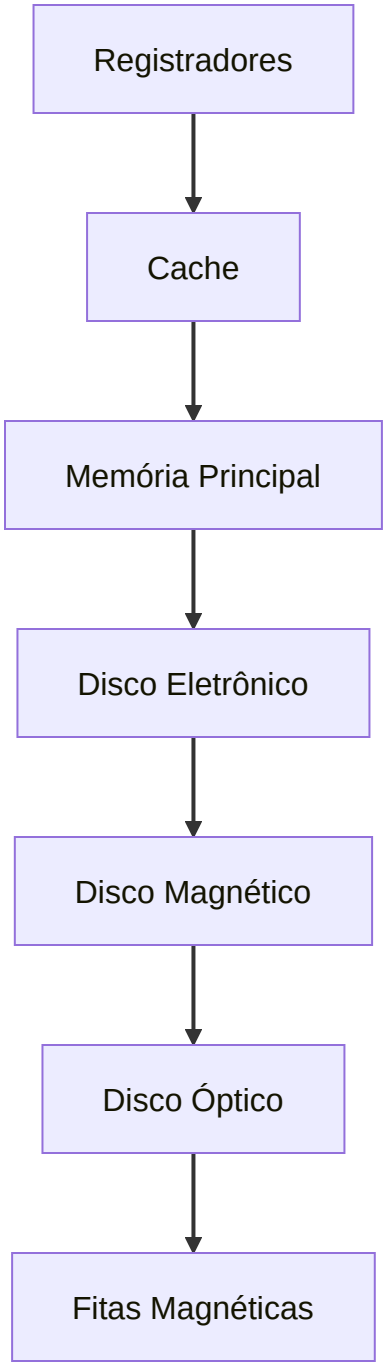
- **A memória principal é volátil**, ela perde os dados assim que a máquina é desligada.
- A memória principal possui um **armazenamento irrisoriamente pequeno** para armazenar todos os programas e dados.

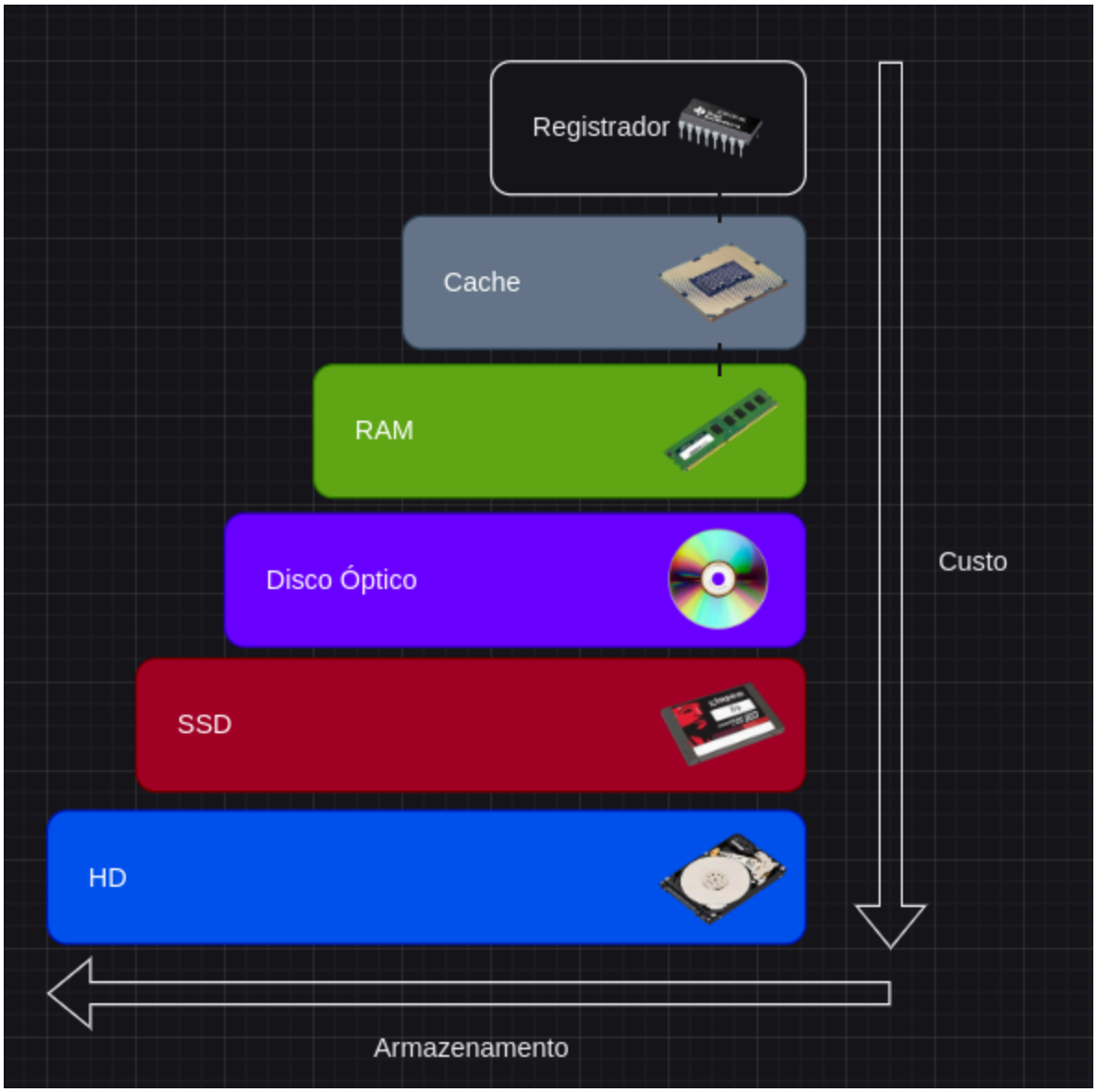
Assim, precisamos de outro tipo de memória chamado **memória secundária**, que tem o propósito de armazenar dados e programas de maneira permanente.

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

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**.

> 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.

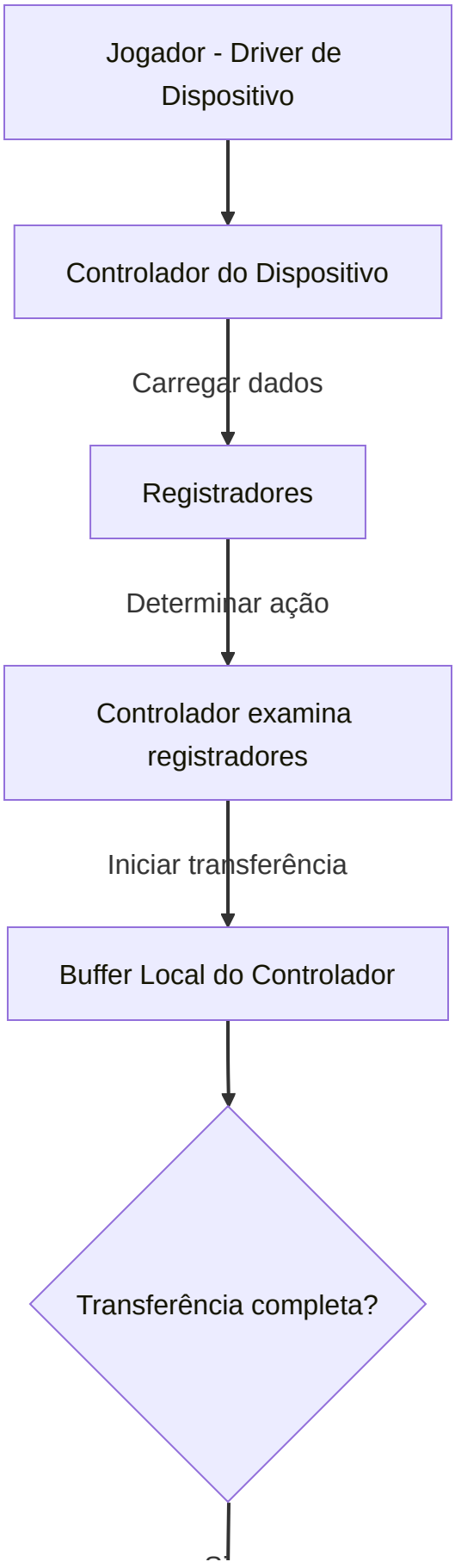
O funcionamento de uma operação de E/S:

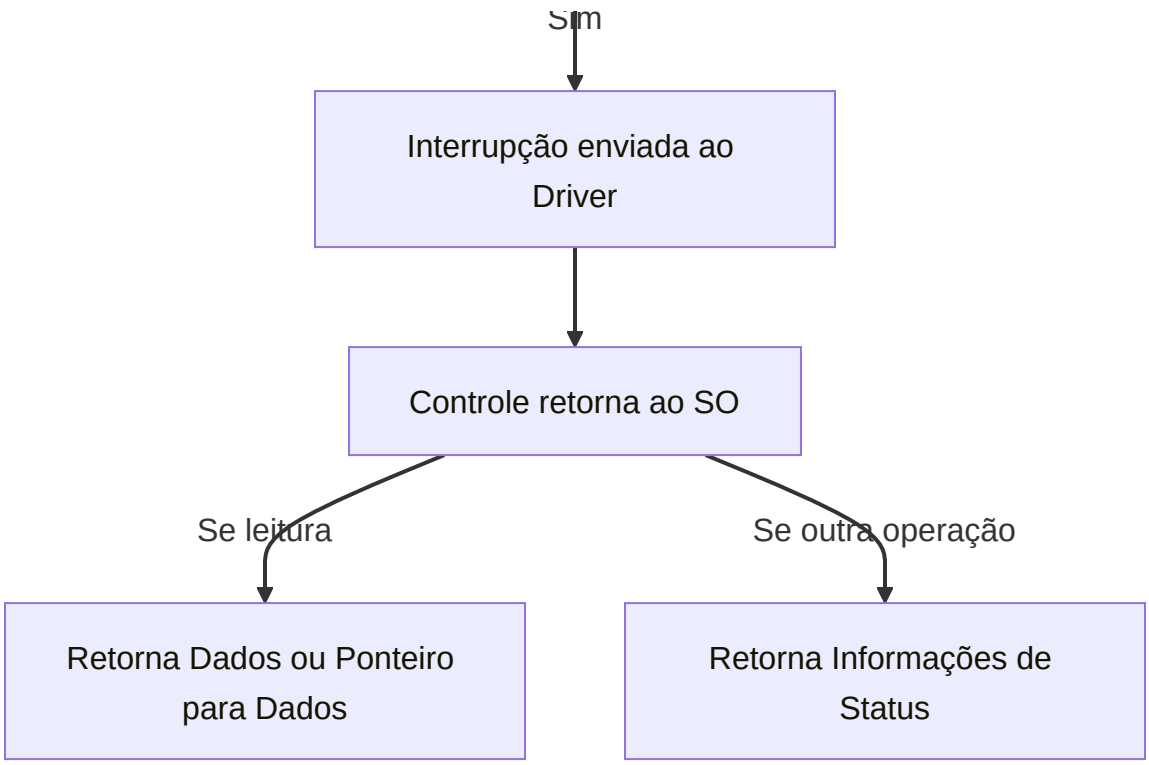
- O **driver de dispositivo** carrega os **registradores** apropriados para dentro do

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:


Jogador - Driver de Dispositivo
Controlador do Dispositivo
Carregar dados
Registradores
Determinar ação
Controlador examina registradores
Iniciar transferência
Buffer Local do Controlador
Transferência completa?



> 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**.

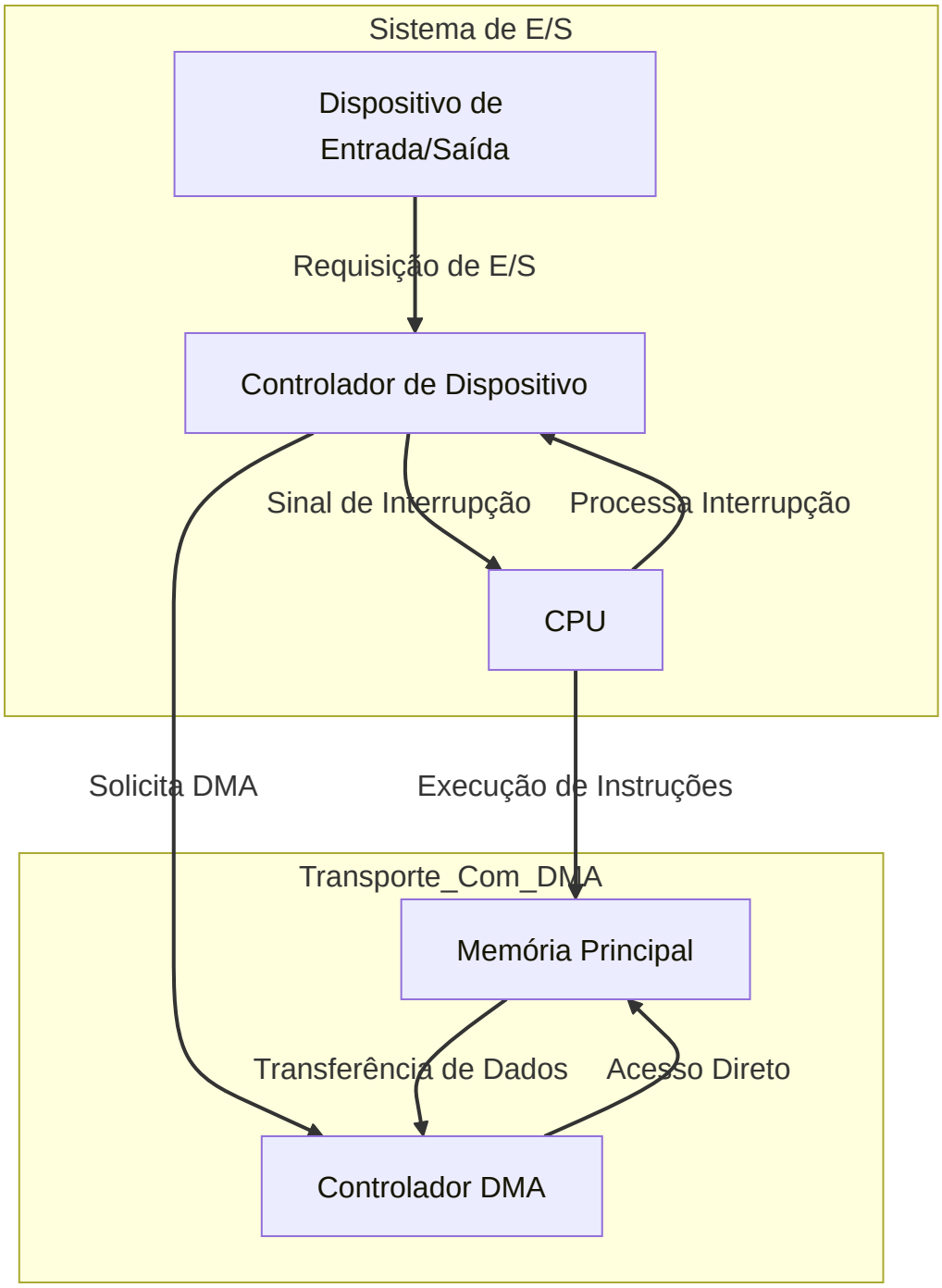
> Nesta etapa de transferência direta não ocorre intervenção da CPU, assim

apenas o controlador de dispositivo cuida dessa tarefa.

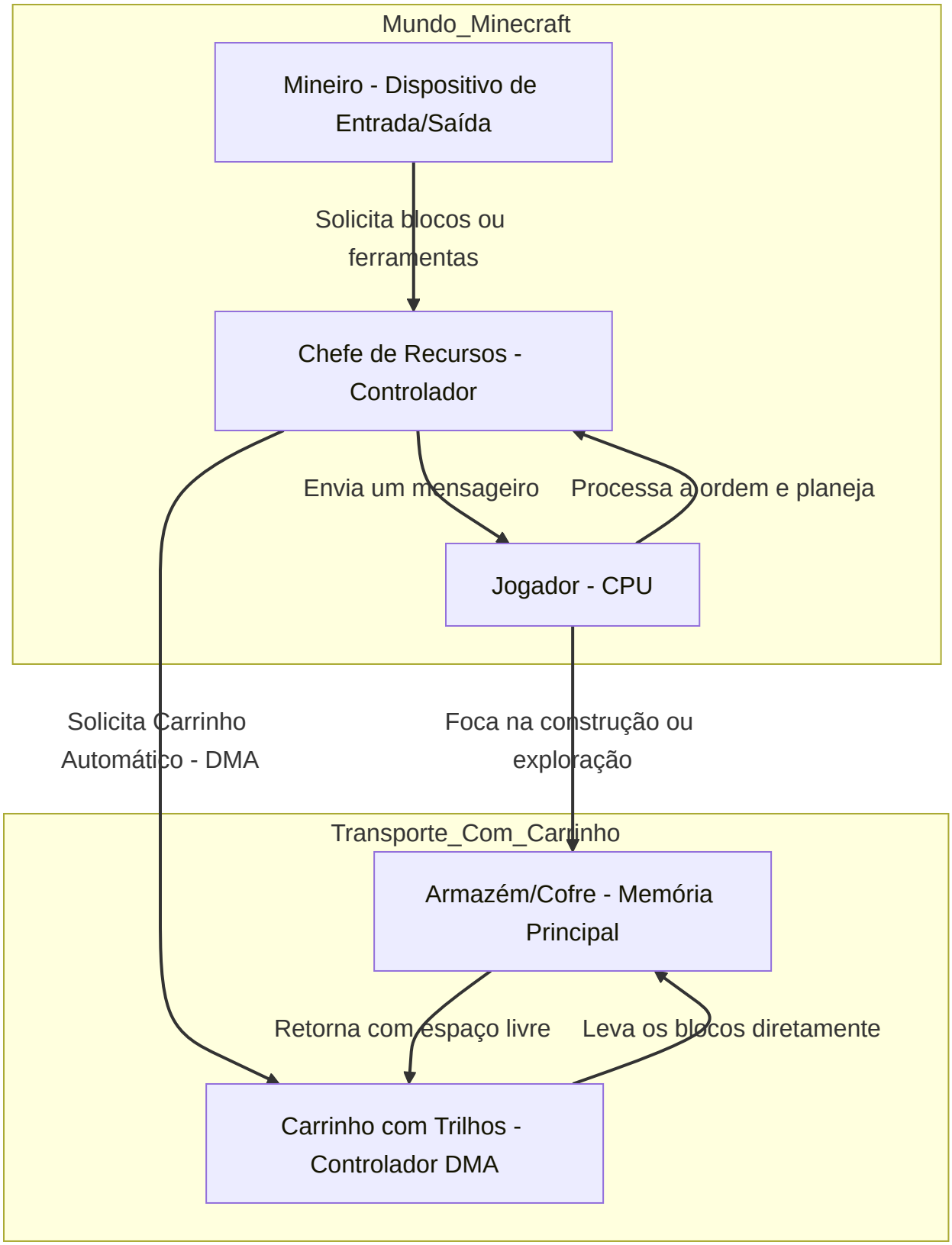
Para alguns sistemas não é utilizado essa arquitetura de barramento e sim de switch:

- 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:


Sistema de E/S
Dispositivo de Entrada/Saída
Requisição de E/S
Controlador de Dispositivo
Sinal de Interrupção
Processa Interrupção
CPU
Solicita DMA
Execução de Instruções
Transporte_Com_DMA
Memória Principal
Transferência de Dados
Acesso Direto
Controlador DMA

- ***Com Mineiro:***


Mundo_Minecraft
Mineiro - Dispositivo de Entrada/Saída
Solicita blocos ou ferramentas
Chefe de Recursos - Controlador
Envia um mensageiro
Processa a ordem e planeja
Jogador - CPU
Solicita Carrinho Automático - DMA
Foca na construção ou exploração
Transporte_Com_Carrinho
Armazém/Cofre - Memória Principal
Retorna com espaço livre
Leva os blocos diretamente
Carrinho com Trilhos - Controlador DMA

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.

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.

> ℹ 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.

> ℹ 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

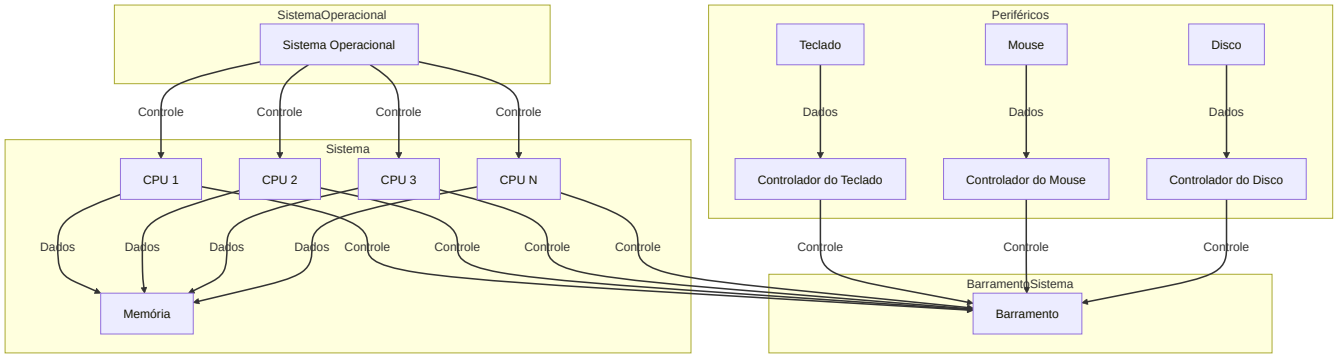
possuem um microprocessador responsável por converter os toques nas teclas em códigos que serão enviados para a CPU principal.

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



Sistema Multiprocessador

Esse tipo de sistema, no qual há mais de um processador de uso geral dentro de um mesmo sistema computadorizado, tem ganhado cada vez mais espaço por diversas razões, substituindo os sistemas monoprocessadores.

Os sistemas multiprocessados, também conhecidos como **sistemas paralelos** (*parallel system*) ou **sistemas fortemente acoplados** (*tightly coupled system*), compartilham periféricos, relógio do computador e barramento entre vários processadores, garantindo uma comunicação eficiente entre eles.

Podemos destacar **três grandes vantagens** desse tipo de arquitetura:

- **Maior vazão**
- **Economia de escala**

- **Maior confiabilidade**

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 sistema 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** a cerca desse tipo de arquitetura para sistemas:

- Maior vazão:
- Economia de escala:
- Maior confiabilidade:

Caching

O entendimento de **caching** é fundamental para compreender como os sistemas computadorizados funcionam.

Primeiro, pensemos que as **informações são armazenadas** em algum **dispositivo de armazenamento**, como a memória principal. Conforme são utilizadas, essas informações **são copiadas para uma memória mais rápida de modo temporário** (até mesmo mais rápida que a memória principal). Essa memória é o ***cache***.

Como funciona?

Primeiro, ao precisarmos de alguma informação, **o sistema busca no cache**:

- Se a informação estiver disponível, **usamos os dados dali mesmo**.
- Caso não esteja, **o sistema irá carregá-la de uma memória mais lenta** (principal ou até mesmo de massa, secundária) e, em seguida, **fará a cópia temporária para o cache**.
    - Dessa forma, em uma **nova tentativa de acesso ao dado, ele poderá ser encontrado no cache**, reduzindo significativamente as consultas lentas que seriam feitas à memória principal.

Indo além, registradores responsáveis pela comunicação com a memória principal, como registradores de índice, **são gerenciados por um algoritmo de alocação e substituição de registradores, implementado pelo programador ou compilador**.

Esses **algoritmos** definem **quais informações serão mantidas nos registradores** e quais **serão enviadas para a memória principal**.

Também existem hardwares projetados para **serem implementados inteiramente no hardware**.

A maioria dos sistemas possui um **cache de instruções**, que serve para **armazenar as próximas instruções a serem executadas**.

Sem isso, a CPU levaria **vários ciclos** para buscar na memória principal a **próxima instrução a ser executada**.

Por essa e outras razões, a maioria dos sistemas possui vários caches de dados de alta velocidade, que estão no **topo da Hierarquia de Memórias**.

Mas, como os caches possuem um **tamanho reduzido**, o **gerenciamento de cache** se torna **fundamental**. Esse gerenciamento envolve alguns aspectos importantes, como:

- Definir o **tamanho do cache**.
- Estabelecer a **política de substituição**.

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.

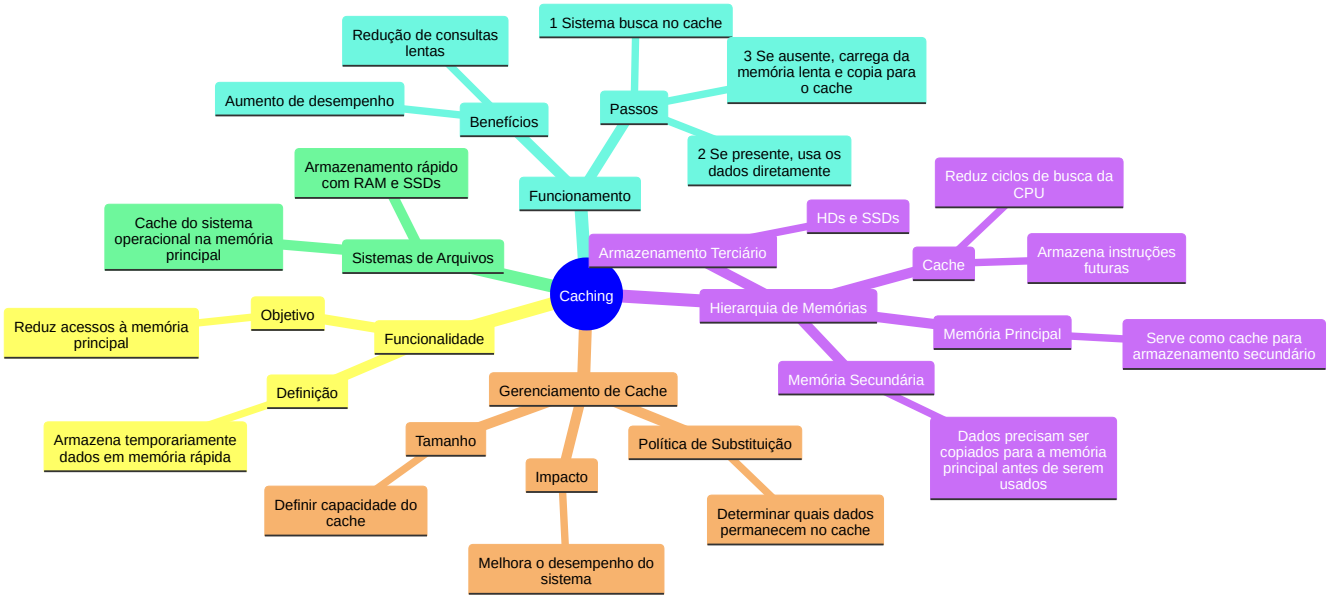
O sistema de arquivos vê os dados permanentemente gravados no armazenamento secundário de forma hierárquica, existindo *diversos níveis na hierarquia*:

- 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**.

Diagrama



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:

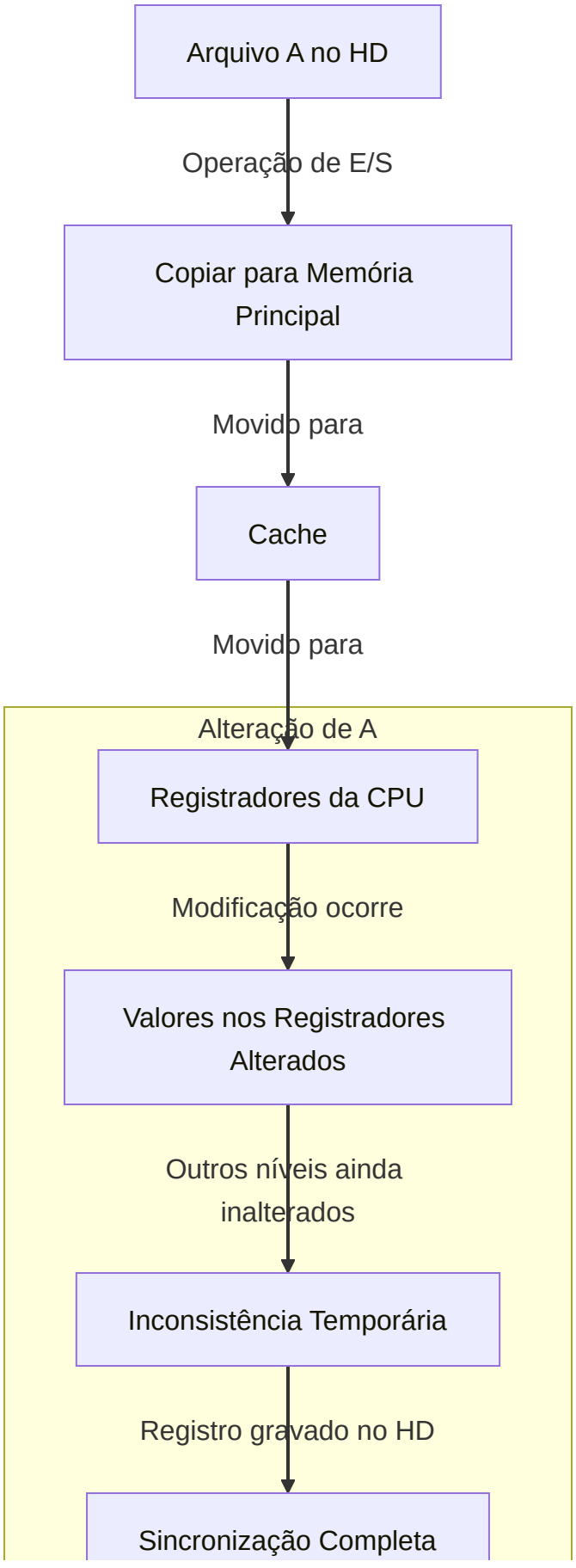
- Suponha que um texto no arquivo `A` precise ser alterado para um outro valor no arquivo `B`, que reside no HD.
- Antes da alteração, **o sistema precisa emitir uma operação de E/S para copiar o bloco de disco contendo `A` para a memória principal**.
- Em seguida, o arquivo `A` será **copiado para o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

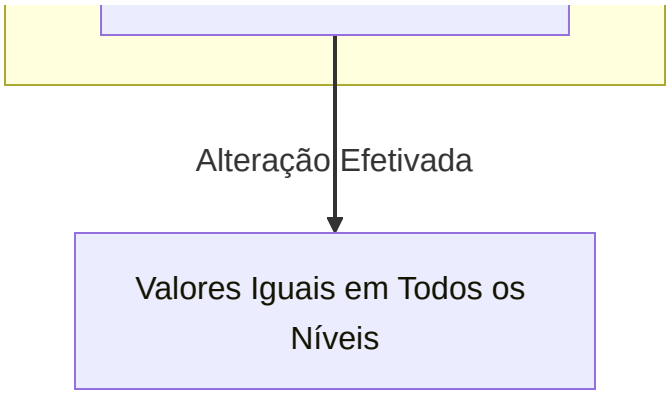
- Assim, a **cópia de `A` estará presente em vários níveis**, conforme mostrado abaixo:

```
Registradores
|
Cache
|
Memória Principal
|
Memória Secundária
```

- 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.

Diagrama


Arquivo A no HD
Operação de E/S
Copiar para Memória Principal
Movido para
Cache
Movido para
Alteração de A
Registradores da CPU
Modificação ocorre
Valores nos Registradores Alterados
Outros níveis ainda inalterados
Inconsistência Temporária
Registro gravado no HD
Sincronização Completa



Threads, Cores e Sistemas Distribuídos

Em um ambiente com **apenas uma thread** (executando uma única tarefa por vez), esse esquema hierárquico funciona perfeitamente. Quando um valor é alterado nos registradores ou acessado no disco, o fluxo ocorre de forma linear, do nível mais alto ao mais baixo, sem conflitos de dados, pois todos os níveis são atualizados sequencialmente.

Entretanto, em **sistemas multicore**, onde múltiplas threads acessam simultaneamente o mesmo arquivo `A`, é necessário um **mecanismo de controle** para garantir que todos os núcleos acessem os valores atualizados de `A`. Caso contrário, podem ocorrer inconsistências, onde diferentes threads terão versões diferentes do mesmo dado, causando falhas no sistema.

Esse problema se torna ainda mais crítico em **sistemas multiprocessadores**, pois, além de registradores internos, cada processador pode possuir caches locais distintos.

Assim, `A` pode existir em diversos caches simultaneamente, e é essencial que a **versão mais recente de `A` seja refletida em todos os núcleos**.

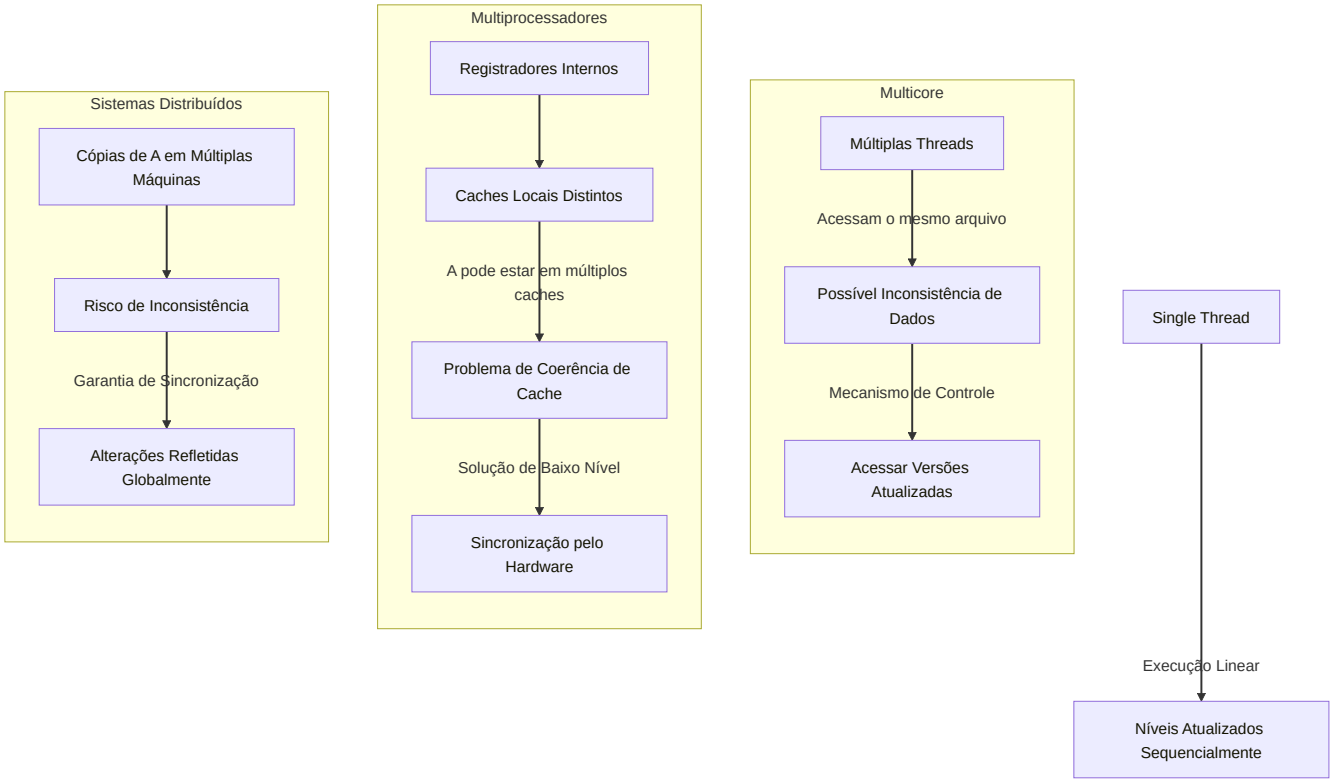
- Esse problema é conhecido como **Coerência de Cache**, e sua solução ocorre em **baixo nível**, diretamente no hardware.

A complexidade aumenta ainda mais em **sistemas distribuídos**, onde diversas cópias de `A` podem estar espalhadas por vários computadores conectados em rede.

Para evitar inconsistências, o sistema precisa garantir que **as alterações no arquivo sejam refletidas nas demais máquinas o mais rapidamente possível**.

- Existem diversas estratégias para resolver esse problema de sincronização em sistemas distribuídos.

Diagrama



Processos

O processo em sí é **um programa em execução**. Seria assim:

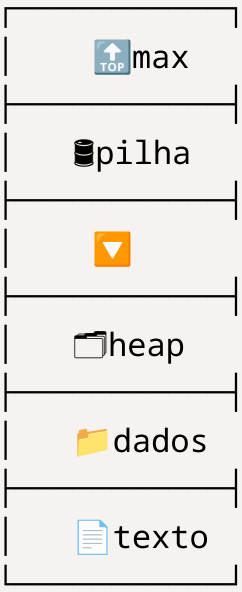
- Ao ligar o PC você **abriu** o Chrome para assistir um video, logo podemos dizer que o **Chrome é um processo**, já que é um programa que está em execução.

Diagrama

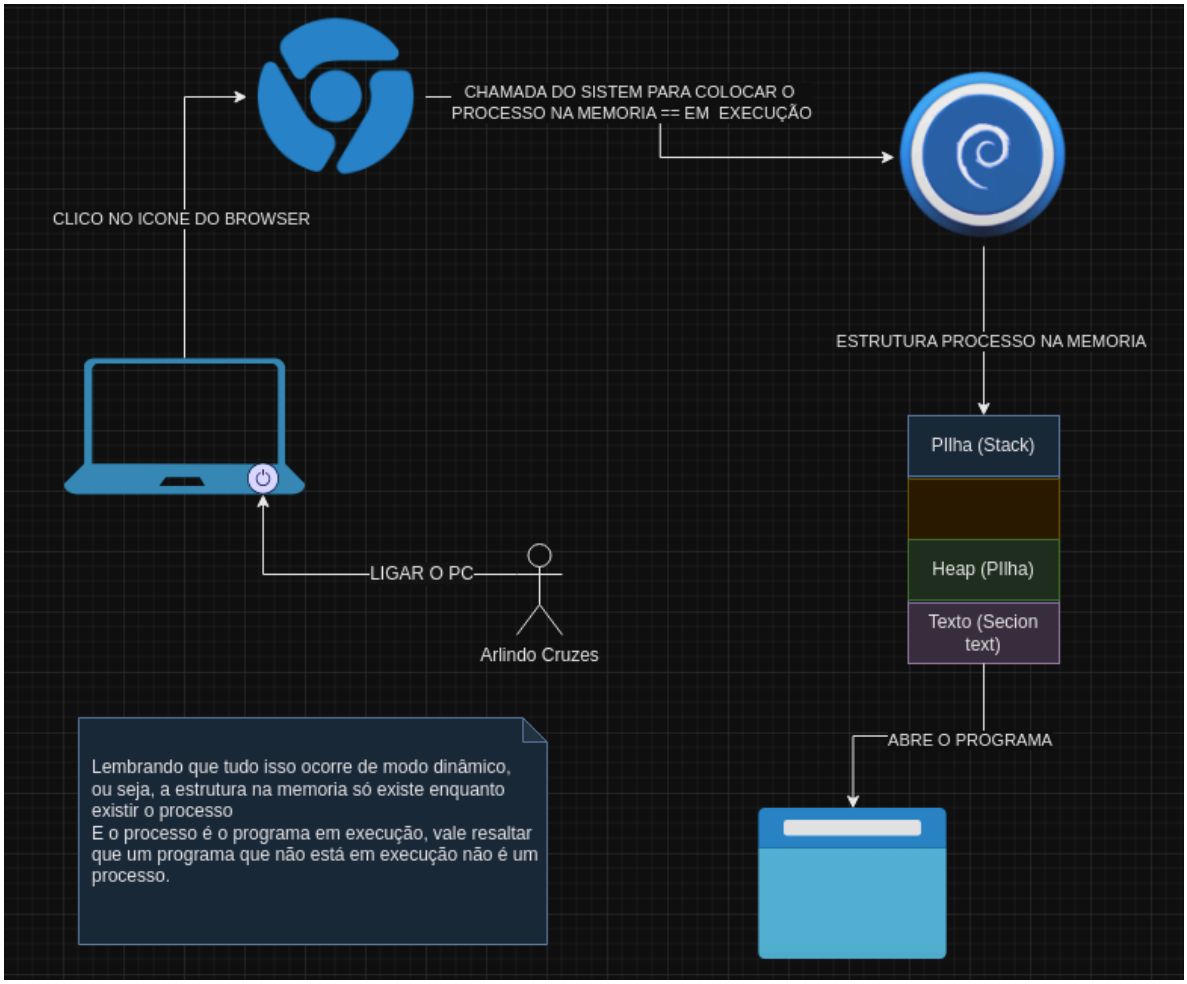
Um **processo** vai além de apenas **códigos em execução** está parte é conhecida como **section text** (seção de texto) isto é um processo possui também:

- **Atividade atual** → que é representada **por um valor do Contador de Programas**
- **Pilha (Stack) de processo** → que contém os **dados temporaries** do processo:
  - Endereços de retorno
  - Variáveis locais
  - Parametros de metodos
- **Uma pilha heap**-> uma **memória alocada dinamicamente**, seria o armazenamento que o processo vai precisar enquanto está sendo executado

Representação da estrutura de um processo na memoria:



Representação de como funcionaria uma chamada do sistema até a alocação do processo na memoria:



Estrutura do processo na memoria

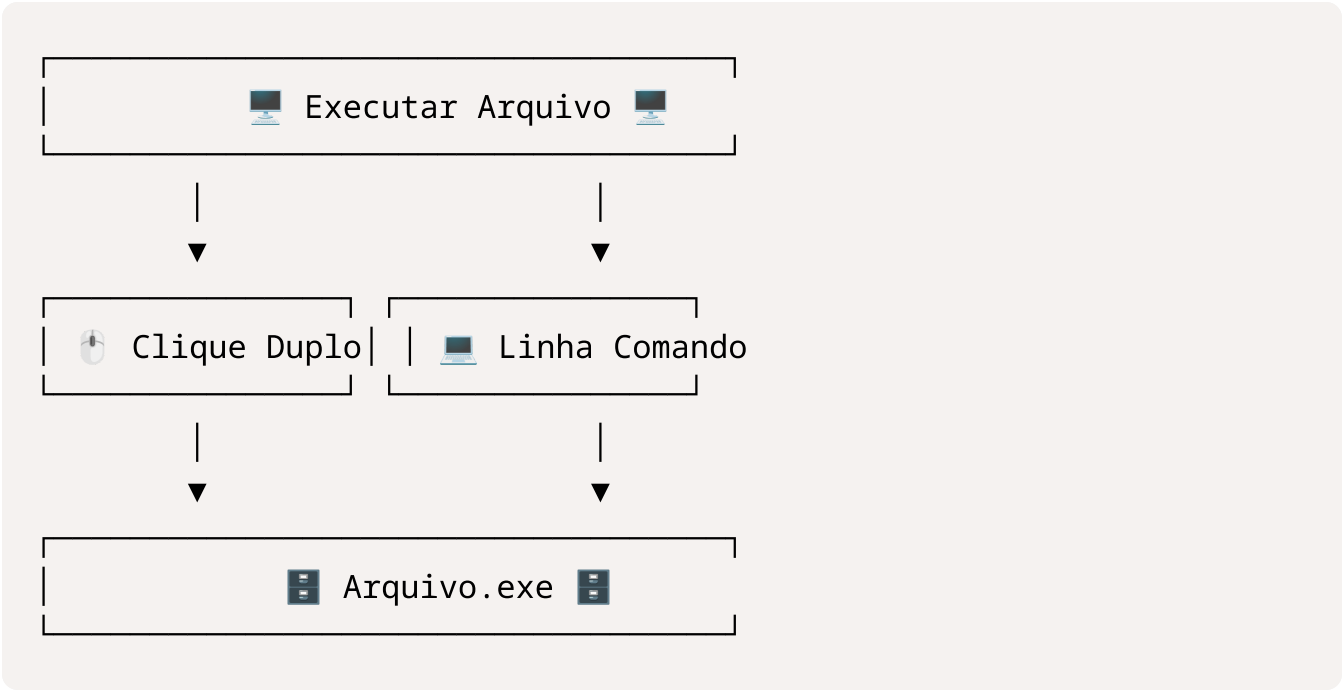
Assim temos que um programa por si só é uma **entidade passiva:

- Um arquivo com algumas instruções Já um processo é uma **entidade ativa**:
- Um arquivo com instruções (código) e, além disso, um **contador de programa** que vai dizer qual a proxima instrução a ser executada Sempre em mente que um processo é um arquivo executavel colocado na memoria

> Só um atendo, essa memória que está sendo falada é a memoria RAM, já que estamos falando de alocação dinâmica, mas não só ela acaba trabalhando temos também nesse meio os Registadores e os Caches

Podemos dizer que existem duas formas de subir um processo ou melhor executar um arquivo executavel:

- **Clique duplo** no icone do arquivo
- **Chamada do nome** do arquivo por linha de comando



Mesmo que um usuario ou varios usuarios executem **o mesmo programa** em dois processos, ou seja, você abriu duas vezes o Chrome serão criados **dois processos distintos**, mesmo que os **componentes abaixo** sejam o mesmo:

- Seção de texto
- A pilha
- Heap

Processos (Processos)

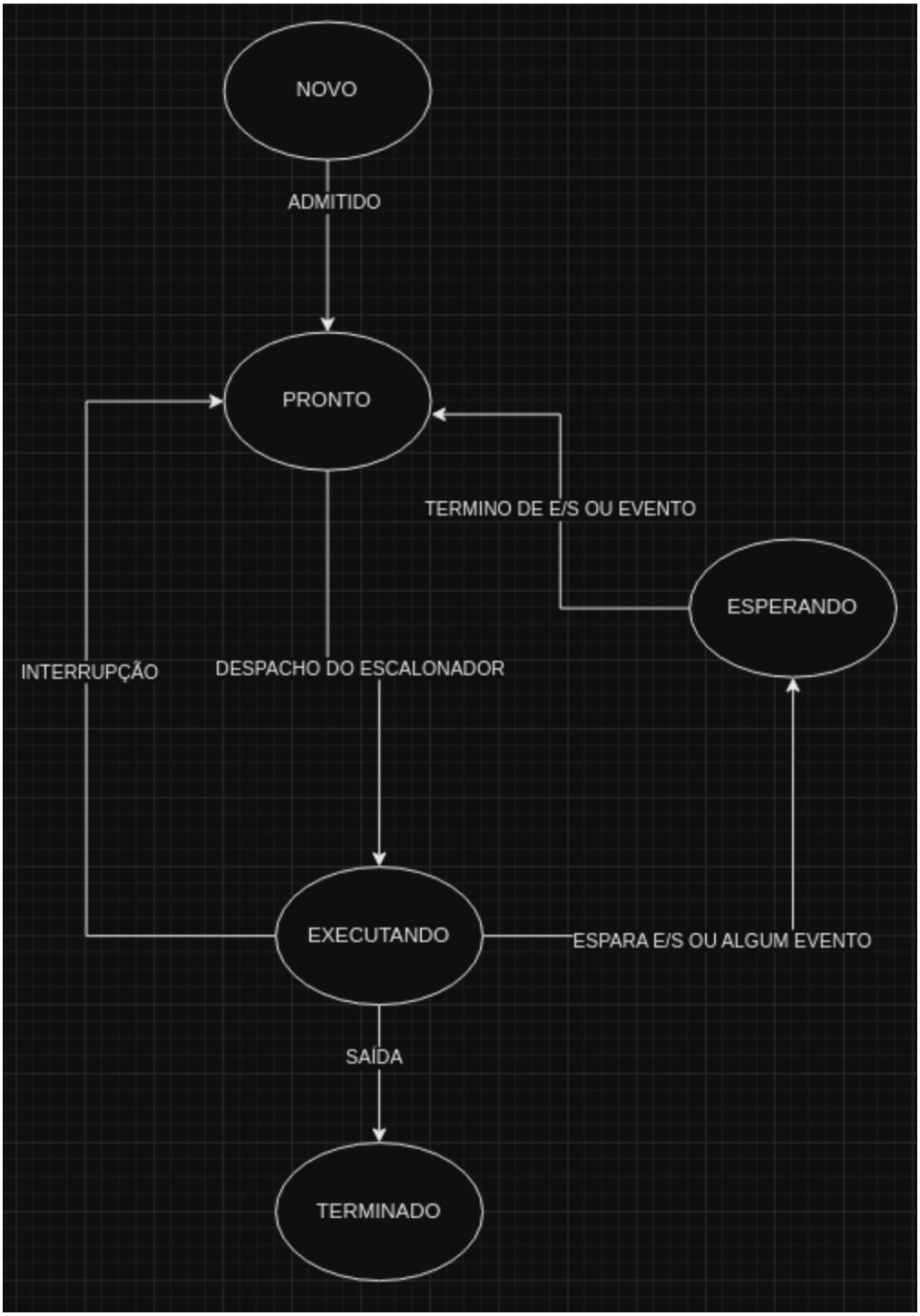
Estados dos processos

Da chamada de execução do processo até o fechamento do processo, ele **passa por alguns estados** que são eles:

- **Novo (new)**-> processo sendo **criado**, ou seja, quando você clica no browser

- **Executando (running)**-> o processo está tendo a seção de texto sendo executada, ou seja, o sistema está **fazendo** as instruções do texto
- **Esperado (Waiting)**-> o processo **está esperando algum evento** (como uma entrada ou saida do sistema ou então recebimento de algum sinal)
- **Pronto (ready)**-> o processo está **esperando ser atribuído** para algum processador
- **Terminado (Terminated)**-> o processo **finalizou** sua execução

Representação dos estados do processo:



Estados do processo

⚠ Esses nomes são arbitrarios, ou seja, podem ou não ser usados em algum sistema operacional, pórem os estados ou seja o que eles significam existem em todos os sistemas operacionais.

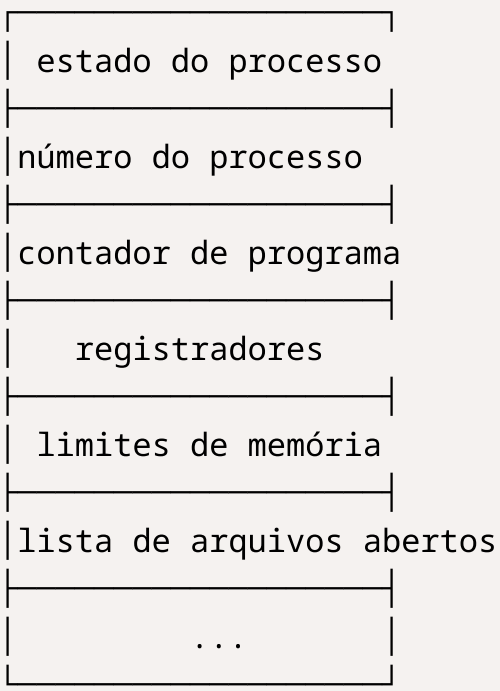
⚠ Vale ressaltar que apenas **um processo** pode estar em **running** (executando) em determinado instante **em qualquer processador**

Bloco de Controle de Processo

Esse **PCB (Process Control Block)** é uma tabela onde estão as informações associadas ao processo.

- Ele pode ser conhecido também como: **bloco de controle da tarefa**

Representação do PCB:



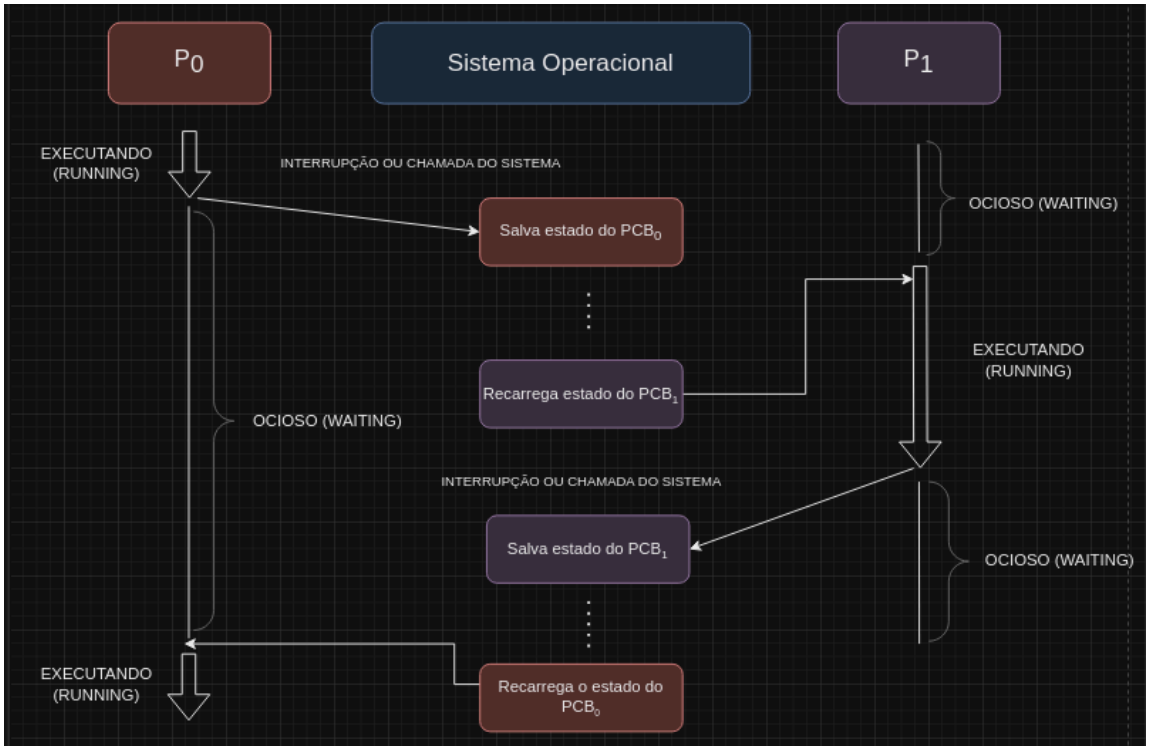
- **Estado do processo**: --> o estado atual do processo vá em Estados do Processo ("Diagrama" in "Processos")
- **Contador de programa (program counter)**-> o contador é aquele que **vai indicar qual o endereço da próxima instrução** a ser executada pelo processo
- **Registradores da CPU**--> os registradores são de vaios tipos e funções, eles possuem:

- Acumuladores
- Registradores de indices
- Ponteiros de pilha
- Registradores de uso geral
- Além de qualquer outra informação

Junto com o contador de programa a informação do estado precisa ser armazenado quando ocorre alguma interrupção, para que ao ser iniciado de novo possar ser executado corretamente.

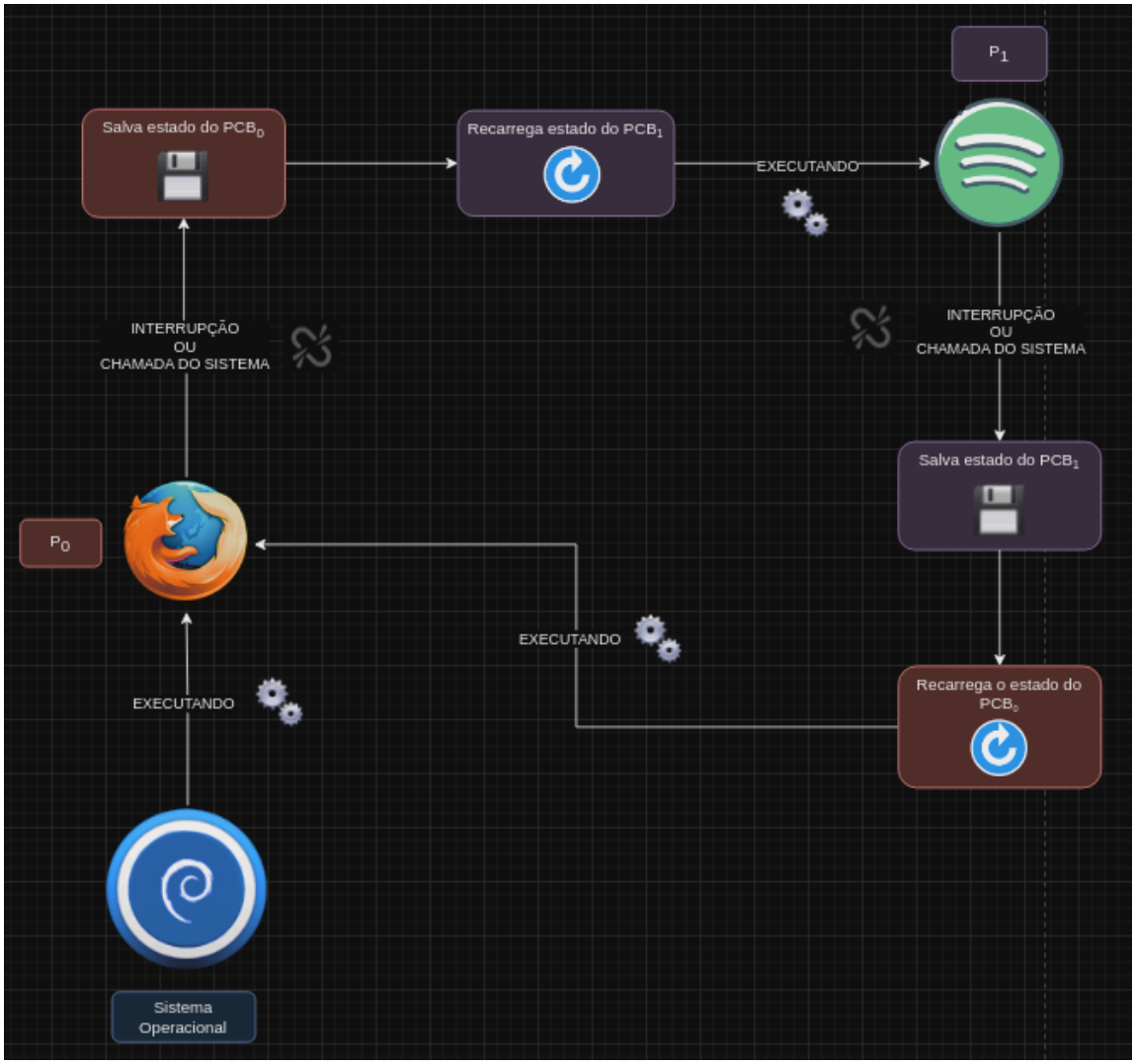
Representação da troca de um processo para outro na CPU:

- Essa é uma outra forma:



Troca cpu processo2

- Essa é uma forma mais *gostosinha*:



Troca cpu processo1

Informações que o PCB possui:

- **Informação de Escalonamento de CPU**--> são usados para o trabalho do **Escalonador (Processos)**:
  - Prioridade de escalonamento, define qual processo vai usar mais a CPU, ou melhor, o *valor processo tem prioridade de uso da CPU*
- **Informação de gêrencia de memoria**-> define as informações de quanta memoria e para qual se destinada o processo, possuindo:
  - O valor dos *registradores de base e limite*

    - As *Tabelas de páginas ou tabelas de segmento* (vai depender do sistema que está usando)

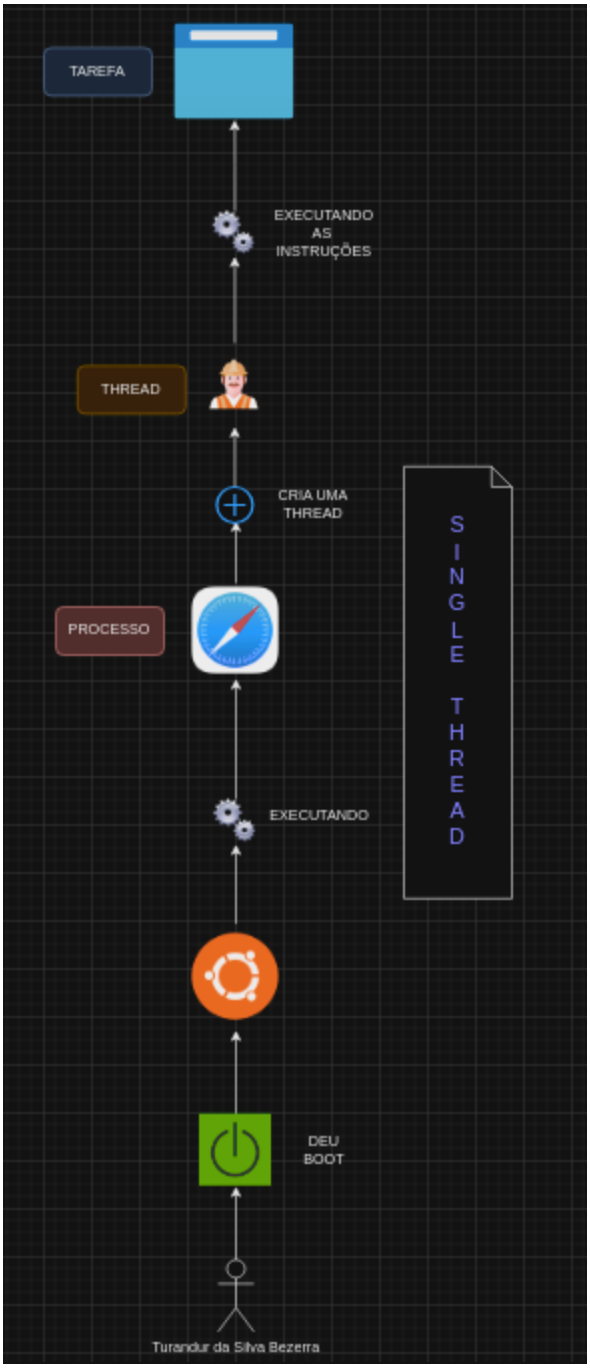
- **Informação contábil**--> define os dados sobre:
    - Uso de CPU como: *quantidade de CPU* e o *tempo de leitura* a ser utilizado
    - *Limites* de tempo
    - Número da *conta*
    - Números de *processos* ou *tarefas*
- **Informações de status de E/S**--> define as informações sobre os dispositivos de entrada e saida e sobre arquivos alocados para aquele processo, entre outros:
    - *Dispositivos E/S* alocados ao processo
    - Uma *lista de arquivos abertos*

Threads

Com o que foi discutido agora, pensemos assim: o modelo empregado até agora foi um processo que trabalha apenas com **uma thread**, ou seja, isso implica que o processo consegue executar apenas uma unica tarefa por vez, jpa que tem apenas uma unica **thread** é como se tivesse apenas um único trabalhador.

Assim ,se aplicarmos isso para entender melhor no contexto de um browser, ele só pode abrir **uma única aba**. já que dentro desse processo só existe **uma única thread (trabalhador)**.

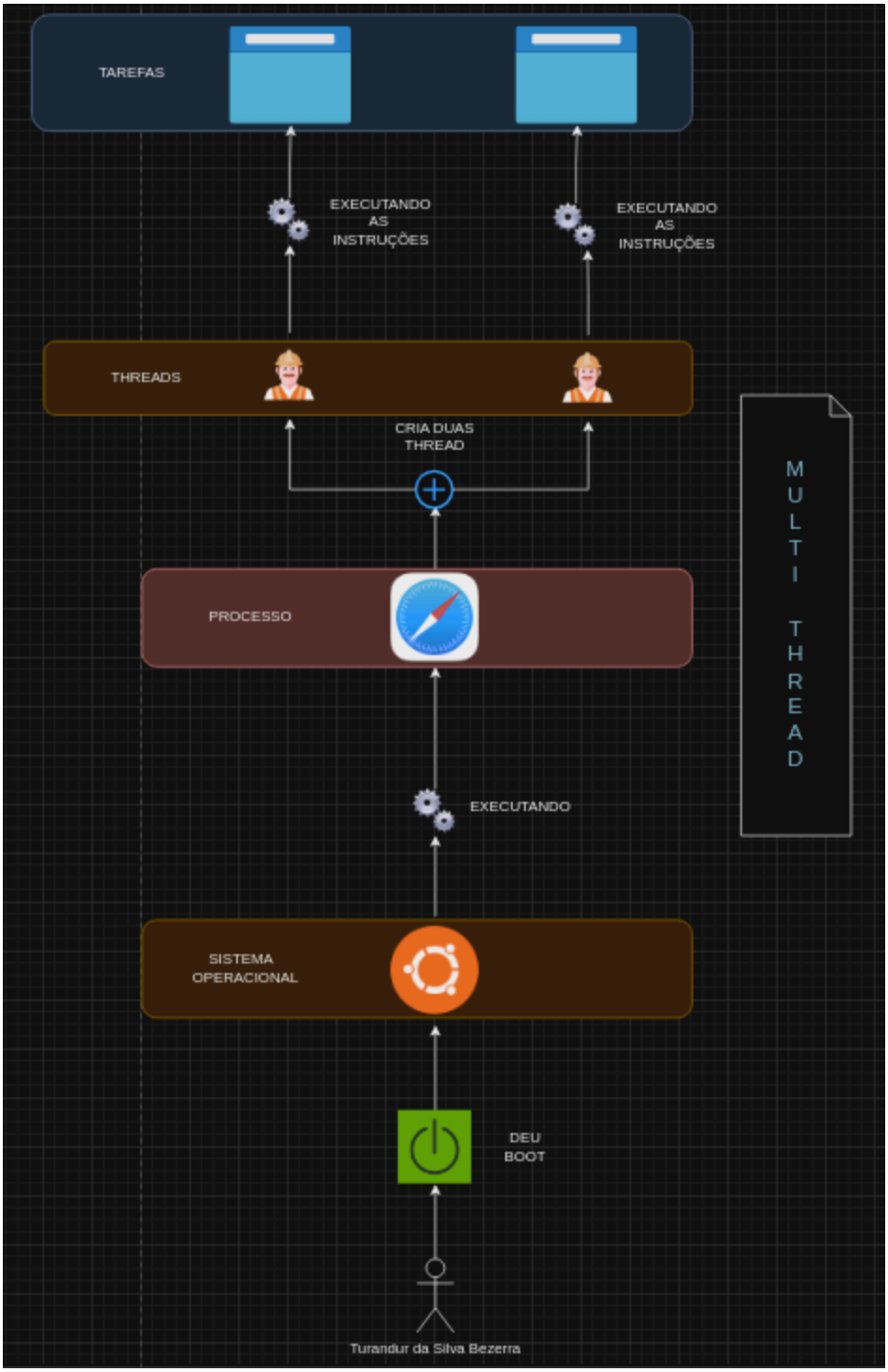
Representação de um modelo single thread:



Single thread

No cenário atual, os **sistemas operacionais** utiliza uma **arquitetura de threads diferente**: como existia essa limitação a cerca dos processos então foi criado essa arquitetura voltada a **multi-threads** que é um processo possuir **mais de uma thread**. Ou seja, ele consegue fazer **mais de uma tarefa por vez**.

Representação de um modelo multi thread:



Multi thread

Porem, para isso ocorrer deve haver mudanças estruturais na forma como os processos e outros componentes são organizados:

- Para o PCB há uma mudança de que ele é **expandido para possuir informações dessas outras threads.**
- Outras partes do sistemas se alteram

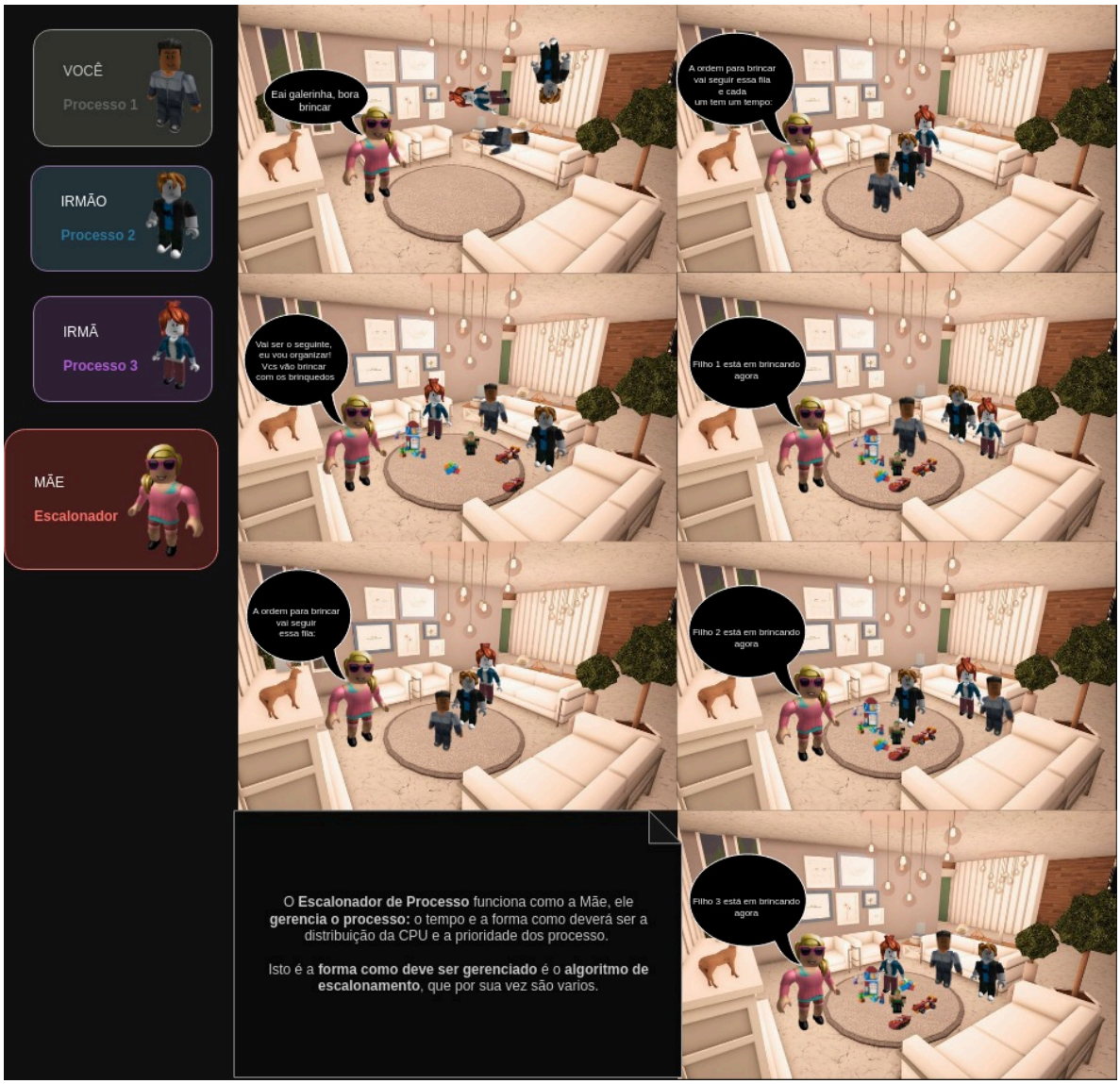
Escalonamento de processos

Objetivo primordial da multiprogramação é ter a capacidade das mulheres de fazer mais de um processo, assim ele parte do principio que **deve ter um processo rodando o tempo todo para melhorar o uso da CPU**.

Então se tem um objetivo que é o **compartilhamento do tempo de uso da CPU** entre os processos isto é feito por um outro processo que no qual é um programa que fica **rodando o tempo todo assim os programas ficam alternando** com uma frequência tão alta que os **usuários nem notem essa alternância e consigam interagir com cada programa**.

Tal que, para atender a essas demandas precisa que o **Escalonador de Processos** (Process Scheduler) **selecione um programa disponivel** (dentro da sua lista de possíveis processos disponíveis), **para que ocorra a execução do programa na CPU**.

Representação do Escalonador de Processos:



Escalonamento de processos1

> Se olharmos para um processador único em um sistema isso não será possivel e o que vai acontecer é que um programa entrara em execução enquanto os outros estarão na fila de espera de execução, até que a CPU esteja disponível para atender a chamada

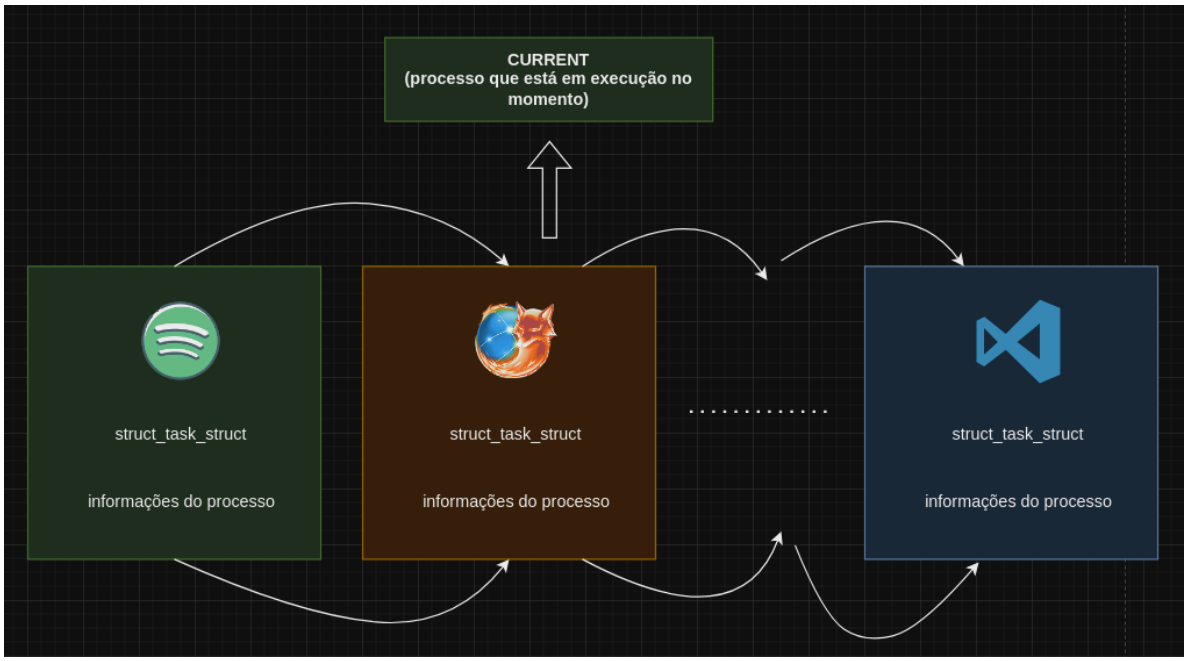
Exemplo de API Padrão: O bloco de controle de processo no sistema operacional Linux é representado pela estrutura `task_struct` que consegue representar todas as informações dos processos:

- Estado do processo
- Informações de escalonamento
- Gerência de memória
- Lista de arquivos abertos e dispositivos E/S alocados para o processo
- Ponteiros para **o pai do processo**:
  - O pai do processo é aquele que o criou
- E para qualquer um de **seus filhos**:
  - São outros processos que são criados pelo processo pai (ou seja, outro processo)

Alguns campos da estrutura:

```
pid_t pid; // identificador de processo
long state; // estado do processo
unsigned int time_slice; // informação de escalonamento
struct task_struct *parent; // processo pai
struct list_head children; // lista de processos filhos
struct files_struct *files; // informações de arquivos abertos
struct mm_struct *mm; // informações de espaço de endereços
```

Assim temos que no exemplo, o estado do processo é representado por `state`, vemos também que nessa estrutura há uma lista duplamente interligada de `task_struct` E o **Kernel** mantem **um ponteiro para processo ativo no momento** (current) para o processo que está sendo executado no momento:



Processo atualmente em execucao

Então se caso mudar o estado do processo o Kernel faria o seguinte: `current->state = new_state`

- Sendo que `curren**t` é um ponteiro para o processo em execução e alteraria um unico processo que está sendo apontado por `current**`, lembrando que ele seria uma estrutura do tipo : `task_struct` por isso poderia ser manipulada desse jeito.

Filas de Escalonamento

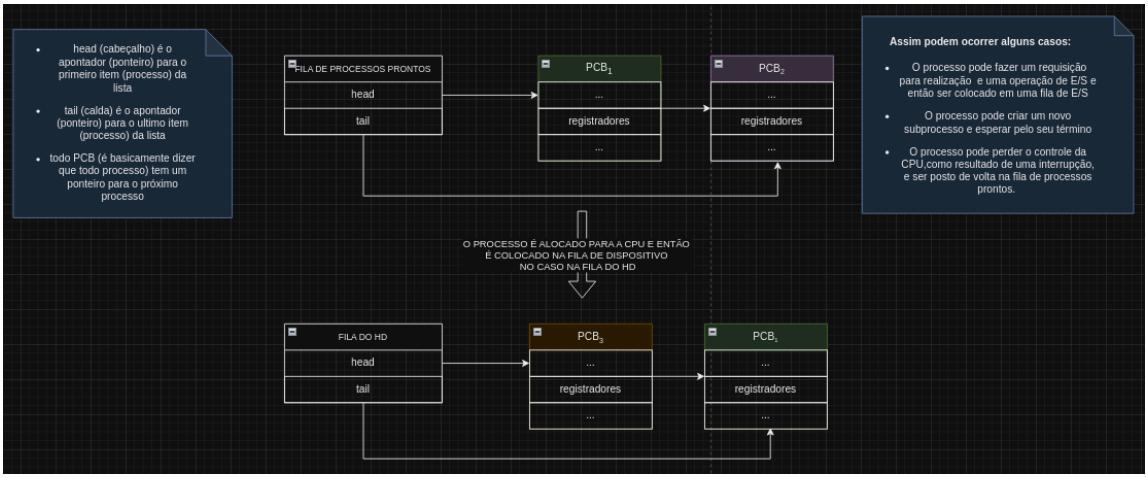
Ao programa ser executado e se tornar um programa ele **entra para a fila de tarefas (job queue)** sendo ela a fila **que contem todos os processos do sistema**.

Os processos que **estão**:

- Na memoria principal (RAM)
- Prontos
- E esperando serem chamados para a execução São colocados na **fila de prontos (ready queue)**.

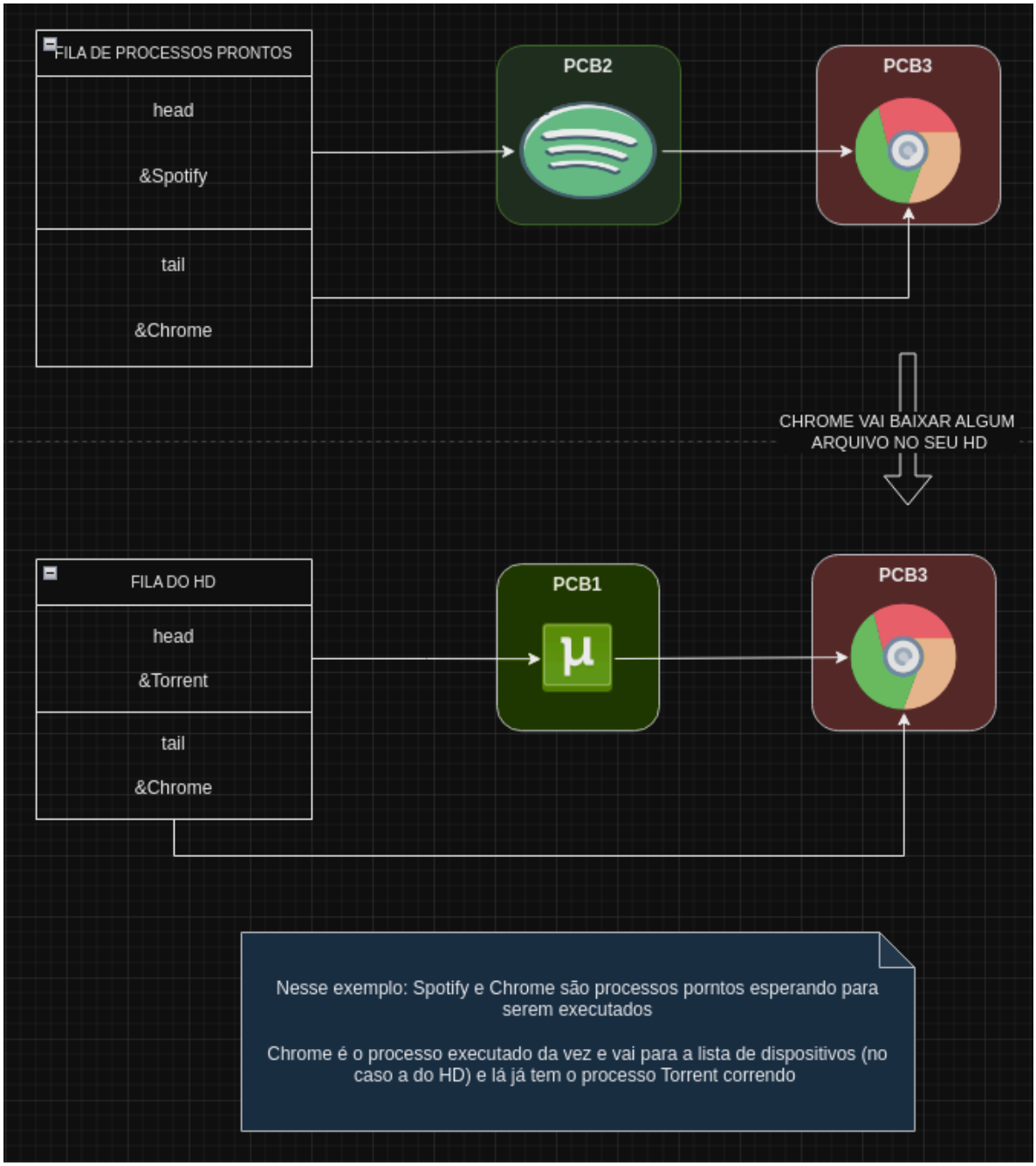
Esta sendo em geral **uma lista interligada** que possui no cabeçalho **ponteiros para o primeiro e ultimo PCB da lista**.

Cada PCB possui **um ponteiro que indica para o próximo PCB** na fila de prontos:



- Filas de prontos e filas de dispositivos

Segunda representação:



Fila de pronto e de dispositivos2

A lista de processos esperando por determinado dispositivo de E/S é chamada de: **fila de dispositivos** ela sendo a fila que vai guardar os **processos que já receberam alocação da CPU**, mas precisa usar um dispositivo:

Um **diagrama de filas**, ajuda a entender como o escalonador de processos trabalha com as listas:

Um **processo criado inicialmente é colocado na fila de pronto**. Ele espera até que vá para a execução (ou seja, **até que seja despachado**). Quando o processo já recebeu o tempo

de CPU, está alocado nela e está executando. Logo então *podem ocorrer um desses eventos*:

- O processo pode fazer uma **requisição de um dispositivo de E/S** e então ser **alocado para a fila de dispositivo** (sendo para a fila respectiva a do dispositivo que se requisitou).
- O processo **pode criar um subprocesso** e **esperar que ele termine**
- O processo pode ser **removido a força da CPU por uma interrupção** e acabar sendo **movido de novo para a fila de pronto**

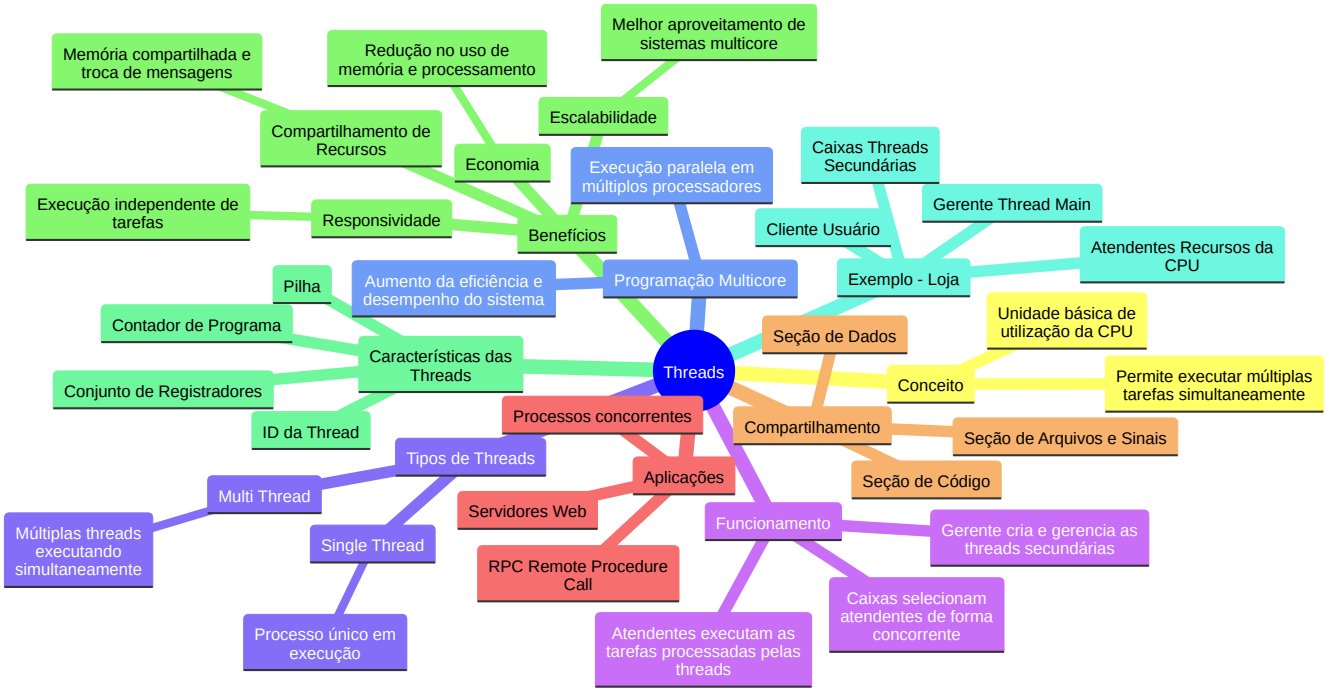
Nos dois primeiros casos, os processos passam para **o estado de espera para o estado de pronto** e depois **é colocado de vota na fila de pronto**. Esse ciclo se repete até que o processo termine **ele então saí de todas as filas que está e a alocação do PCB e seus recursos são removidos.**

Threads

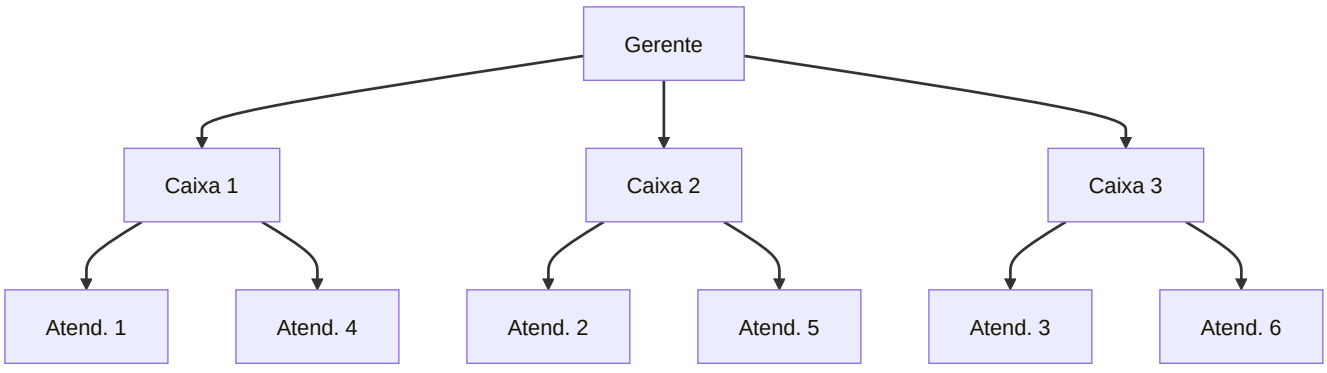
Threads são as fatias de processos do sistema, são "fios" criados para resolver um processo, assim é possível fazer mais de uma tarefa.

> Uma thread é uma unidade básica de utilização de CPU

Diagrama



Vamos imaginar um cenário, de uma loja:

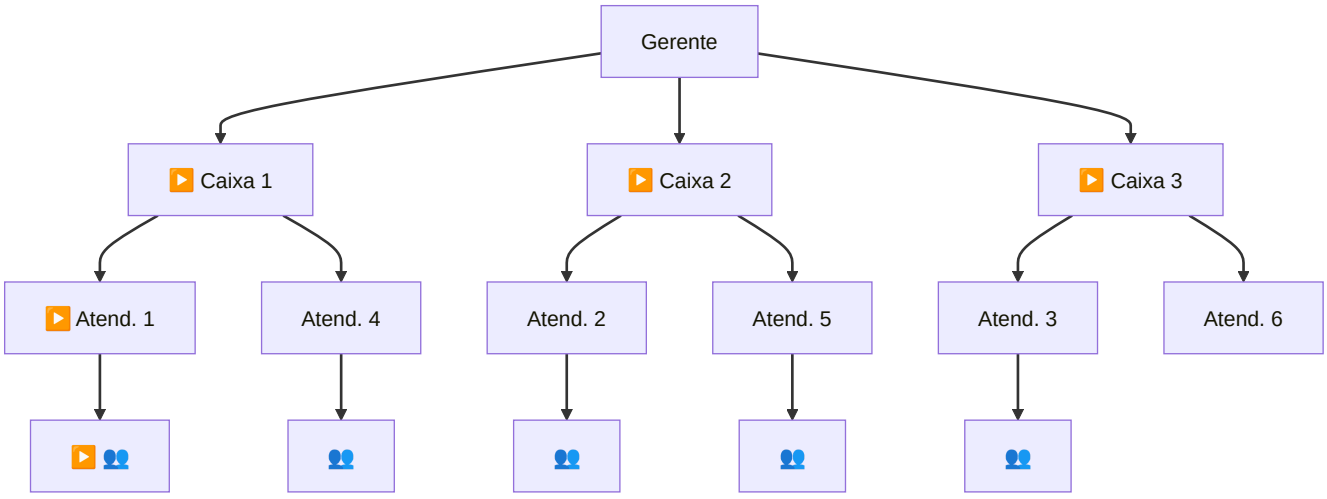


Pense assim:

- O Gerente é a thread main (seria o "fio" principal);
- O Caixa 1, 2 e 3 são respectivamente os lugares de entradas de dados, onde os atendentes fazer o atendimento dos clientes elas sendo as threads secundarias;
- Atendente 1, 2, 3... eles são os recursos da CPU, ou mesmo podemos chamar de CPU onde ela irá servir para atender o cliente e alocar ou não determinados recursos que foram pedidos;
- Cliente seria o usuario que entrou com algum dado que é passado para o sistema operacional que faz alguma chamada nas threads secundarias (atendentes) thread main (gerente);

Funcionamento:

- O Gerente **cria e gerencia** as threads secundarias (caixas).
- As caixas são as responsáveis por **escolher um dos atendentes para atender** os clientes, de modo que elas são **independentes** e veja que logo cada uma está trabalhando de modo **concorrente**, como se estivessem "disputando".
- Os atendentes são os recursos usados pela CPU, ou mesmo pode se dizer a CPU, para resolver a thread, ou seja, executar determinada tarefa.



Explicação:

- **Gerente** cria as threads de Caixa (Caixa 1, Caixa 2, Caixa 3).

- Cada **Caixa** tem atendentes (Atend. 1, 2, 3, 4, 5).
- Quando o cliente chega, ele é direcionado ao caixa específico (seta indicando o fluxo).
- **Atendente 1** é utilizado para processar a compra do cliente.

> ℹ️ Tudo isso é gerenciado e orquestrado pelo Sistema Operacional.

Assim , as threads compartilham de algumas coisas em comum:

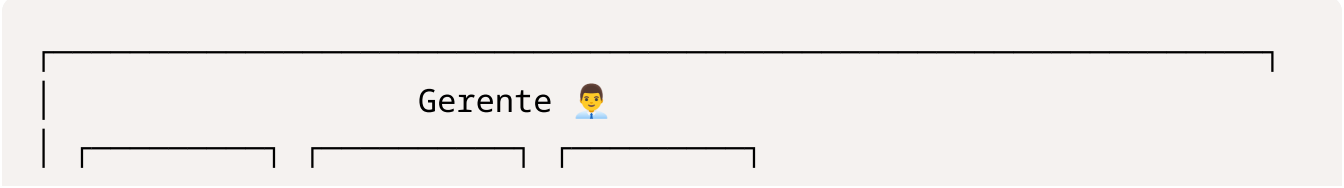
- Seção de **código**;
- seção de **dados**;
- Seção de outras coisas como **arquivos** e **sinais**;

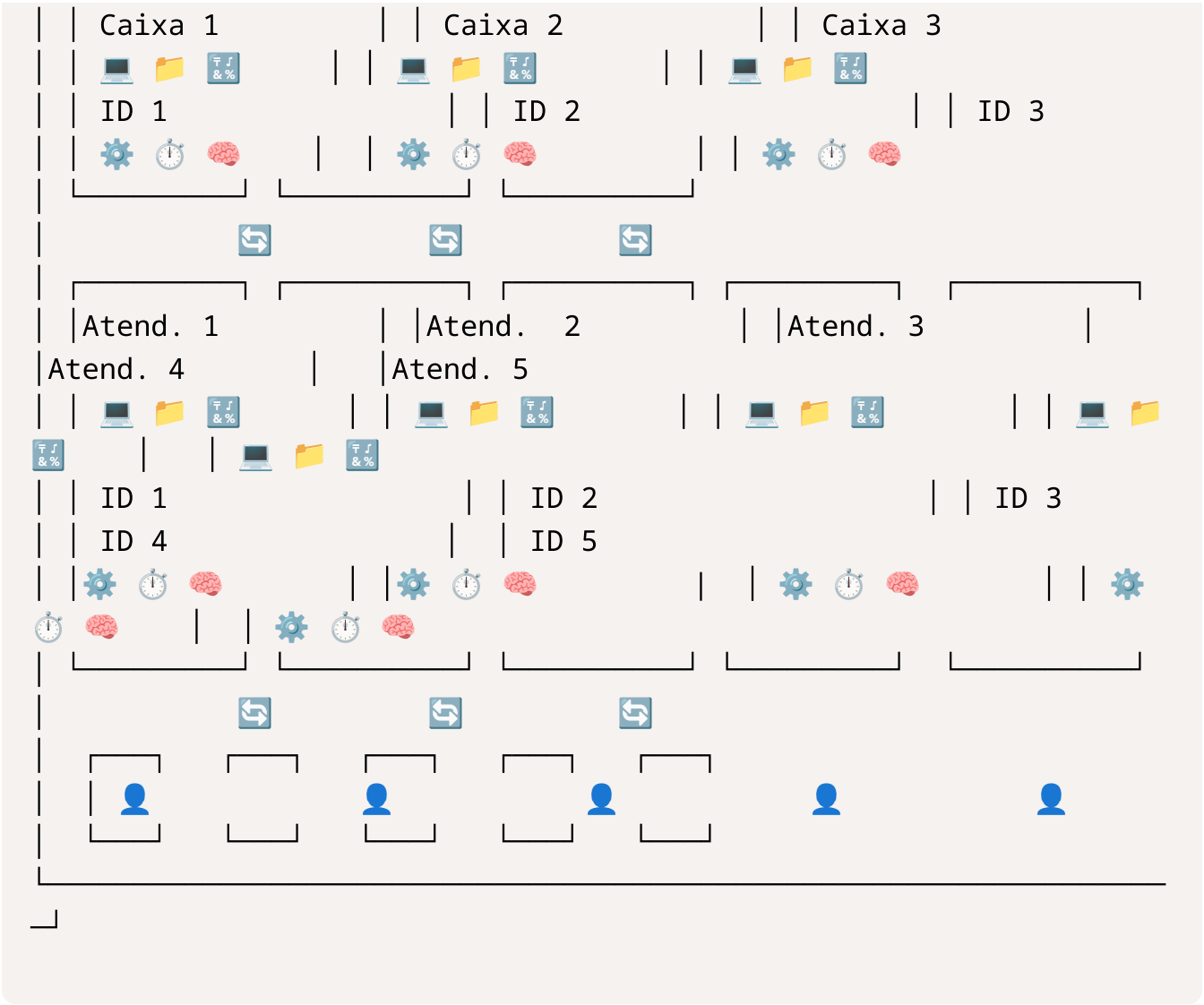
Trazendo para o exemplo acima temos que o Gerente, caixa e atendente compartilham de:

- Seção de códigos de conduta, o código que define o que eles devem fazer e como deve ser feito;
- Seção de dados, tanto da loja como deles mesmos ou de clientes ou tarefas
- Seção de arquivos ou mesmo utensílios da loja

Porem, não é só isso, **as threads possuem** basicamente:

- ID da thread
- Conjunto de registradores;
- Uma pilha;
- Contador de programa;



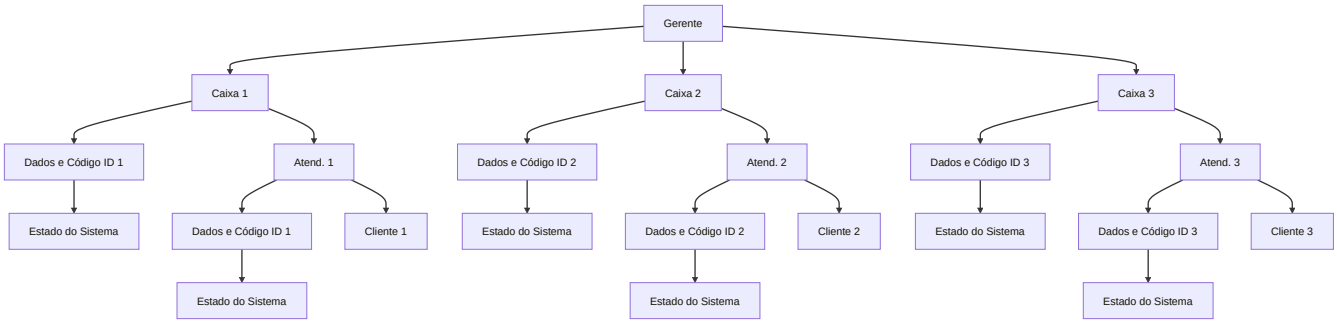


Nesta representação, utilizei os seguintes elementos:

- 👨‍💼: Símbolo do Gerente
- 💻: Código (seção de código compartilhada)
- 📁: Dados (seção de dados compartilhada)
- 🔣: Arquivos e Sinais (seção de outros recursos compartilhados)
- ⚙️: Registradores
- ⏱️: Contador de programa
- 🧠: Pilha

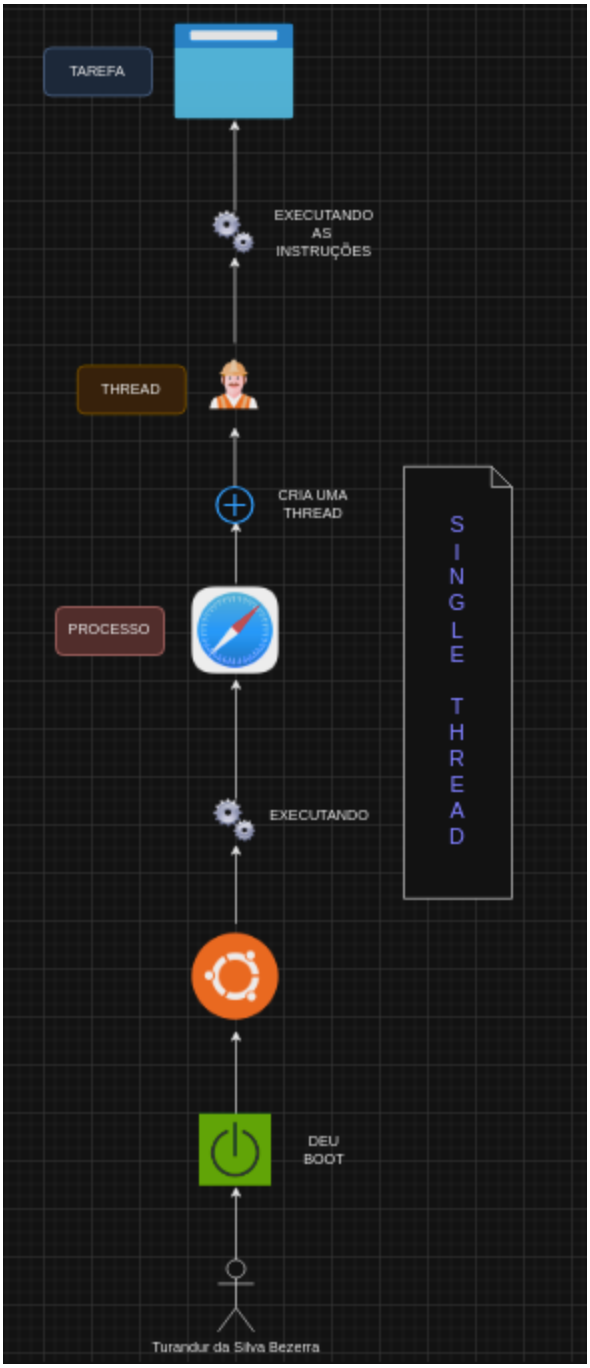
- 👤: Clientes
- 🔄: Fluxo de execução (threads)
- ID: Identificador único de cada thread

1. O Gerente 👨‍💼 cria as threads de Caixa (Caixa 1, Caixa 2, Caixa 3), cada uma com seu próprio ID, conjunto de registradores, pilha, contador de programa, e compartilhando a seção de código, dados, arquivos e sinais.
2. Os Atendentes (Atend. 1, Atend. 2, Atend. 3, Atend. 4, Atend. 5) também são criados como threads, com as mesmas características de ID, registradores, pilha e contador, além de compartilharem a seção de código, dados, arquivos e sinais com o Gerente e as Caixas.
3. Os Clientes 👤 chegam e são atendidos pelas threads de Caixa e Atendentes, que compartilham os recursos necessários para processar as compras.



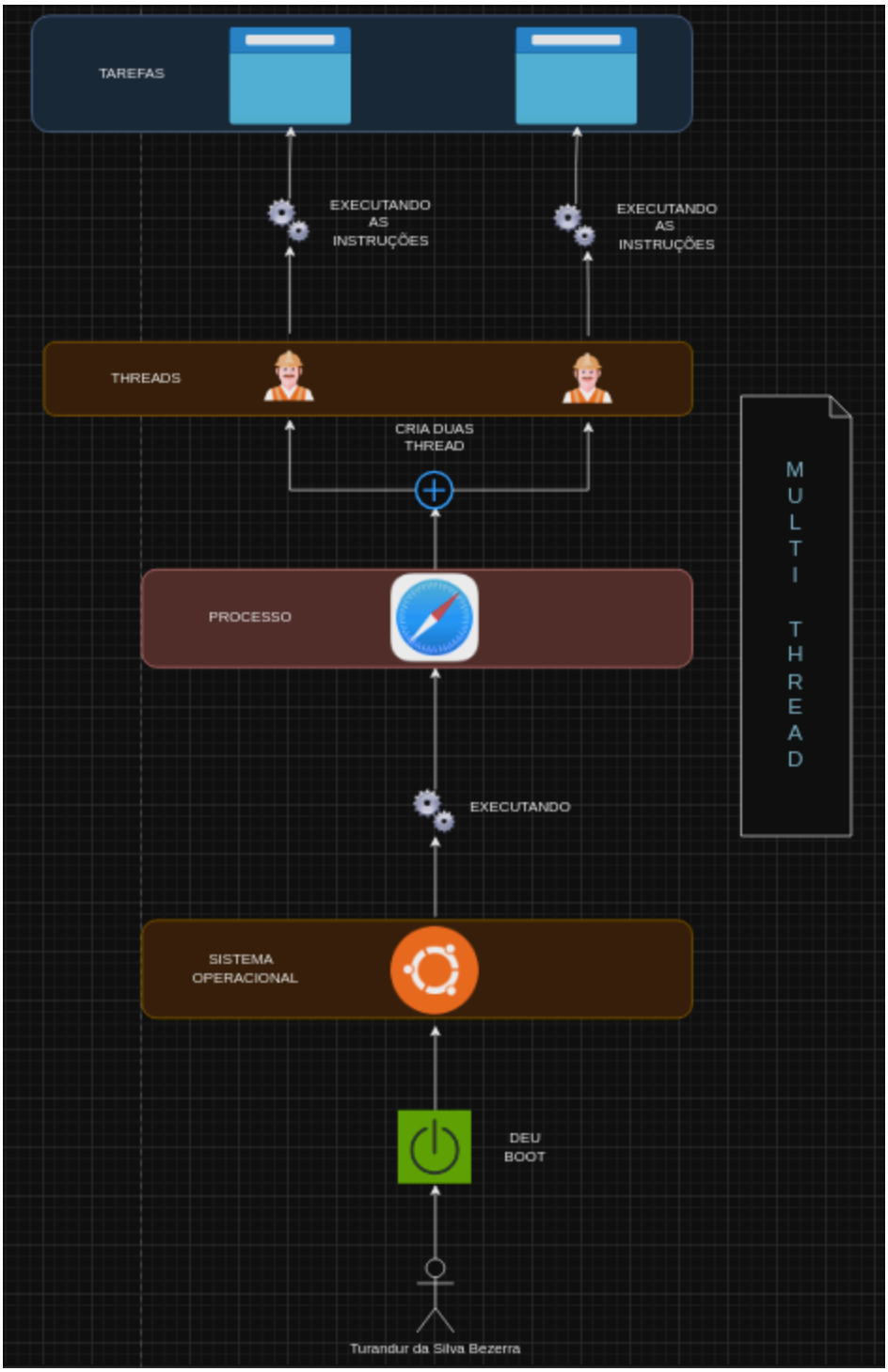
As threads em seu uso, ou seja a forma como os processos são construídos, podem ser divididos em **dois tipos threads:**

- **Singlethread**: é uma **única thread** em uso;



Single thread

- **Multithread**: são **varias threads** que funcionam simultaneamente, de modo **paralelas**;



Multi thread

Usos

Vários softwares que são executados nos computadores modernos são dotados de múltiplas threads.

Sendo que se olharmos para uma **aplicação** ela é em geral **um processo principal** sendo executado **de forma separada** e com **varias threads de controle**. Quando olhamos para **algumas aplicações** elas tendem a fazer **varias tarefas semelhantes,** como no caso de um **servidor Web**.

Tomemos como exemplo, um servidor web, o que ele faz? Bem ele recebe requisições de um cliente (um outro computador) tal servidor que é um computador pode ter diversos senão centenas de outros clientes fazendo **requisições** ao mesmo tempo de modo **concorrente** já que as requisições estão sendo executadas no mesmo momento (é o mesmo que dizer que elas estão competindo, são concorrentes);

Caso pensarmos que o servidor fosse um sistema **com uma unica thread** temos que a cada **requisição** ao **servidor ele só atenderia um cliente por ver.**

Assim temos que a **solução** para este problema é justamente fazer com que existam multiplas threads assim podemos fazer com que **o servidor** p**ossa criar uma thread para cada requisição** e assim **essa thread possa atender a requisição**.

Threads tem uma função muito importante nas **RPC** (*remote procedure call*-> **fazem a comunicação entre os processos**, algo parecido com chamadas comuns de função)

Os servidores de RPC atuam de modo multithreads: ele espera **receber** uma **requisição (mensagem)** então ele **cria uma thread especifica para resolver aquela mensagem**, assim o sistema consegue atuar com **varias requisições de modo simultâneo**.

Benefícios

Tais benefícios podem ser divididos em quatro categorias:

1. **Responsividade**--> *Capacidade de dar uma resposta não importando a condição de outras tarefas*.

- A execução de varias tarefas de modo independente faz com que mesmo se uma tarefa estiver demorando muito ou então foi interrompida não faz com que as outras acabem caindo (sejam mortas ou interrompidas). Assim temos

2. **Compartilhamento de Recursos**--> *Ter varias threads no mesmo endereço de memoria compartilhando dados*.

- Como as threads conseguem compartilhar os códigos e dados de duas formas: memoria compartilhada e trocas de mensagens (tais técnicas são feita pelos desenvolvedores), as threads conseguem executar diversas atividades e estarem no mesmo espaço de memoria e compartilharem recursos entre si

3. **Economia**--> *A principal economia que se tem ao se usar threads é o baixo processamento e uso de memoria para cria-las e gerenciar*.
    - Ao criarmos um processo temos que usar mais processamento e memoria do que criar uma thread, além de que as threads compartilham recursos do seu processo pai.
    - De modo que temos não só uma economia na criação mais também no uso de threads já que os recursos que uma usa as outras caso precisem conseguem usar, sem ter que fazer um outro processo
4. **Escalabilidade**-> *O uso e multithreads em um sistema multicore (múltiplas CPUS) faz com que se possa ter o uso do paralelismo elevado ao máximo, assim aumentamos o poder e velocidade de processamentos*.
    - Ao usar múltiplas threads em um processo em que o sistema é apenas de uma CPU acaba que temos que uma única thread só pode ser executada em um único processador o que diminui a eficiência
    - E por vez a escalabilidade, mas em sistemas com arquiteturas multicore temos varias threads sendo executadas em vários processadores, o que resulta em um maior uso do paralelismo

Programação multicore

Intro - Gerencia de Memoria

Os sistemas computadorizados possuem uma **finalidade principal de executar programas**. Logo, esses programas vão precisar dá memoria, necessariamente estar na memoria, pelo menos em parte para ser executado.

Assim, a importância se dá em que para a execução e armazenamento é preciso não apenas a memoria para armazenamento é preciso de um sistema para gerenciar as questões da memoria.

Próximo ->

Conceitos Básicos

Memoria Principal

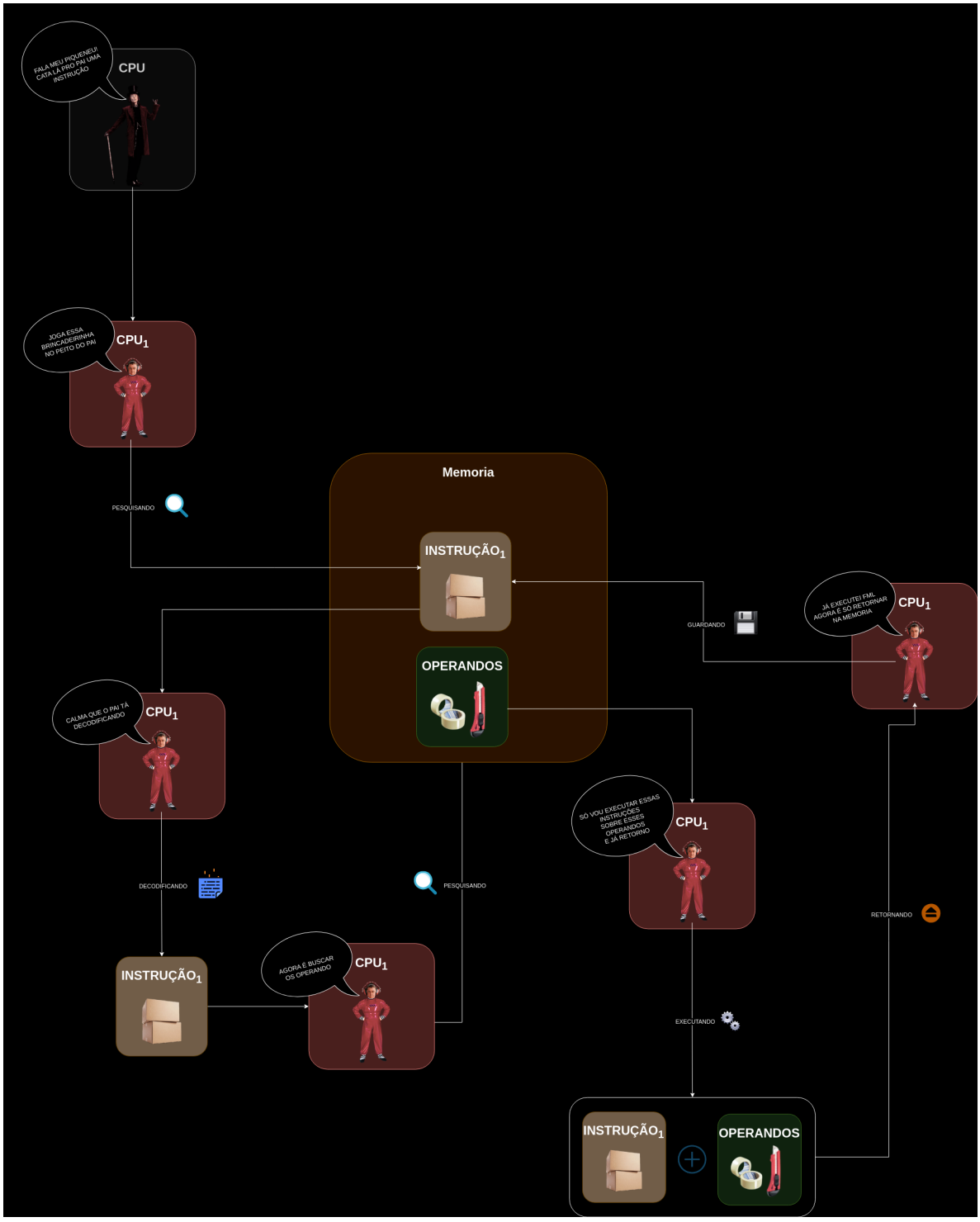
A memoria, sendo ela uma parte essencial para os sistemas computadorizados. Sendo que a essência mais básica de uma memoria é ser formada por uma grande seguem cia de **words** e **bytes**, cada uma das partes com seu próprio endereço. A **CPU pega as instruções da memoria a partir do valor o contador do programa**.

Tais instruções podem fazer com que:

- Carregamento adicional
- Alocação em endereços específicos da memoria

Um ciclo comum de execução de instrução, seria:

- **Pega** uma instrução da memoria
- **Decodifica** a instrução e faz os operandos serem buscados na memoria
- Depois de **executar** a instrução **sobre** os operandos
- Resultado é **guardado** de volta na memoria



Ciclo comum de execucao de instrucao na memoria drawio 1

> ⚠ A memoria, ou melhor, a unidade de memoria **vê apenas um fluxo de memoria** e não como são gerados pelo contador de programas, etc

Hardware básico

A **memoria principal** e os **registradores embutidos** no proprio processador são as **unicas unidades de armazenamento ligadas diretamente a CPU**, ou seja, somente essas unidades (memorias) conseguem ter acesso diretamente a CPU.

Algumas instruções pegam o **endereço de memoria** como argumentos, mas elas não podem acessar **endereços de disco**. Assim os dados que serão usados pelas instruções em execução, **precisam estar em um dos dispositivos**: memoria principal ou registradores embutidos no processador eles **estando ligados diretamente com a CPU**.

Portanto, caso um dado não esteja lá na memoria, **eles precisaram ser movidos para ela antes da CPU possa fazer algo** com eles.

Registradores internos da CPU **são acessiveis normalmente** por um **ciclo de clock (rélogio)** da CPU. Maioria das CPUs podem **decodificar e executar as instruções na velocidade de uma ou mais clock tick**. Para a **memoria principal** isso não acontece já que nela a comunicação (acesso) é **feito pelo barramento de memoria**. O acesso à memoria pode exigir varios ciclos de relogio (clocks) da CPU para completar.

Assim o **processador precisa adiar** (stall ou adiar) suas **operações** já que **ele não tem os dados** necessarios para completar a **instrução** que está **sendo executada**. Logo, essa situação se escala em um nível intolerável, pois a memoria acaba **sendo usada com muita frequência**.

Para tal a solução é criar **uma memoria de acesso rápido** que fique **entre o processador e a memoria principal** esse buffer **seria o intermediário** entre esses dois componentes -> este buffer é chamado de **[[08 - Caching]]**

Além da questão da velocidade devemos levar em consideração a questão de proteção, ou seja, proteger o sistema operacional tanto de programas do usuário como programas do usuário de outros programas.

O que pode ser um problema já que devemos mantar alguns princípios de segurança como confiabilidade. Essa proteção é feita e precisa ser assim pelo hardware, assim temos mais "segurança" já que o hardware é o mais baixo nível que conseguimos manipular.

Garantindo segurança

Primeiro temos que separar um espaço na memoria (um endereço fisico) para cada processo. Assim podemos atribuir um intervalo legal (possível) de endereços físicos na

memoria que podem ser acessados pelo processo.

Com isso vamos precisar de duas medidas, o começo (base) e o fim (limite) do intervalo de memoria que podem ser acessados:

- Base -> contem o valor do menor endereço físico da memoria
- Limite -> contem o valor do maior endereço físico

> ⚠ Essas "coisas" => base e limite possuem nome eles são registradores, estes recebem nomes específicos que são bem óbvios: **registrador de base** e **registrador de limite**

<- Anterior | Próximo ->

Associação de endereços

<- Anterior | Próximo ->

Sistemas Distribuidos

Hardware

Start typing here...

Software

Start typing here...