

SO e Arquitetura

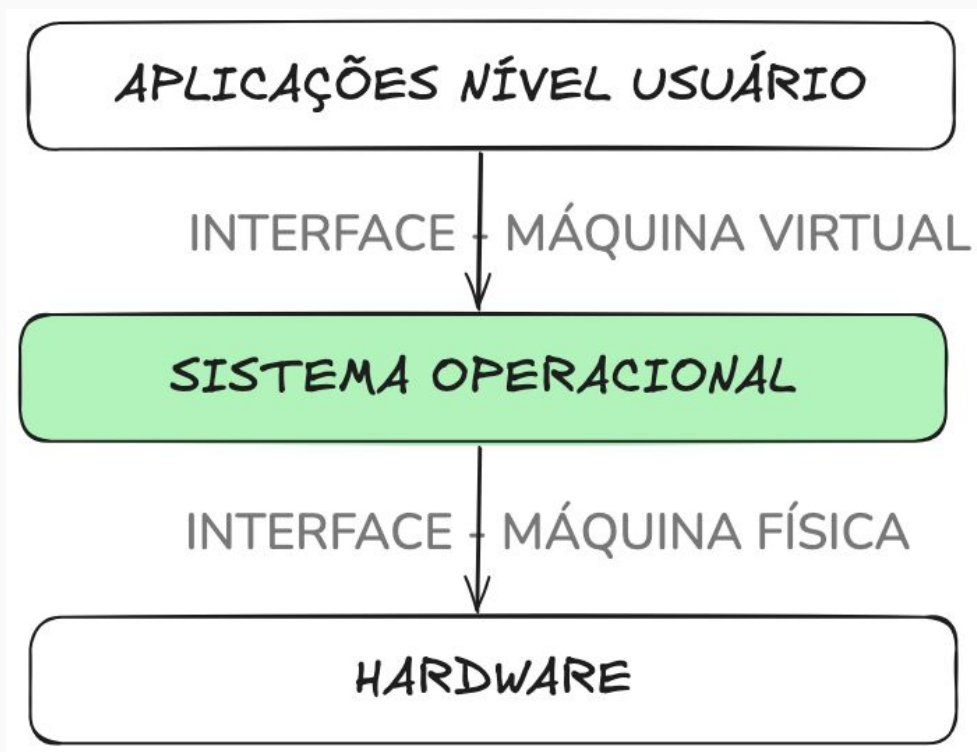
Sistemas Operacionais

Prof. Pedro Ramos
pramos.costar@gmail.com

Pontifícia Universidade Católica de Minas Gerais
ICEI - Departamento de Ciência da Computação

ANTERIORMENTE: INTRODUÇÃO

- Um sistema operacional é uma interface entre o usuário e a arquitetura.
- História: o SO reage às mudanças em hardware e também motiva novas mudanças.



HOJE: SO E ARQUITETURA

- Funcionalidades básicas de um SO
- Revisão de arquitetura
- O que um SO pode fazer é determinado pela capacidade da arquitetura
 - Dependendo da arquitetura, projetar um SO pode ser fácil ou difícil.

A arquitetura pode simplificar ou complicar muito a vida de um Engenheiro de SO.

FUNCIONALIDADES DE UM SO MODERNO

Gerenciamento de processos e threads

- *Aplicações podem disparar múltiplos processos*
- *Processos podem ter múltiplas threads*

Concorrência: fazer várias coisas ao mesmo tempo (I/O, Processamento, múltiplos programas e etc)

- *Vários usuários trabalham ao mesmo tempo como se cada um tivesse uma máquina privada.*
- *Uma thread ativa por vez na CPU, mas várias threads ativas na pool*

I/O: CPU tende a ser rápido, I/O tende a ser lento.

- *CPU deve realizar trabalho enquanto um I/O lento ocorre*

Memória: SO coordena alocação de memória e move dados entre DISCO e MEMÓRIA PRINCIPAL.

- *A máquina tem um tamanho limitado de memória física.*

FUNCIONALIDADES DE UM SO MODERNO

Gerenciamento de processos e threads

- *Aplicações podem disparar múltiplos processos*
- *Processos podem ter múltiplas threads*

Concorrência: fazer várias coisas ao mesmo tempo (I/O, Processamento, múltiplos programas e etc)

- *Vários usuários trabalham ao mesmo tempo como se cada um tivesse uma máquina privada.*
- *Uma thread ativa por vez na CPU, mas várias threads ativas na pool*

I/O: CPU tende a ser rápido, I/O tende a ser lento.

- *CPU deve realizar trabalho enquanto um I/O lento ocorre*

Memória: SO coordena alocação de memória e move dados entre DISCO e MEMÓRIA PRINCIPAL.

- *A máquina tem um tamanho limitado de memória física.*
- ***O que acontece se tenho + dados do que memória disponível?***

FUNCIONALIDADES DE UM SO MODERNO

- **Arquivos:** SO coordena como o espaço em disco é utilizado para os arquivos -> encontrar, organizar, etc.
 - *O que fazer com o espaço vazio após deletar um arquivo?*
- **Redes e sistemas distribuídos:** permite que grupos de máquinas trabalhem junto em hardware distribuído

OS 4 PRINCÍPIOS

1. MALABARISMO

Prover a **ilusão** de uma máquina dedicada com memória e CPU **infinitos**.

2. GOVERNO

Proteger usuários uns dos outros, alocar **recursos** imparcialmente, prover comunicação **segura** entre **processos**.

3. COMPLEXIDADE

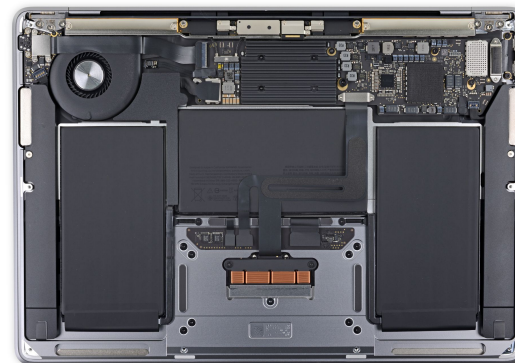
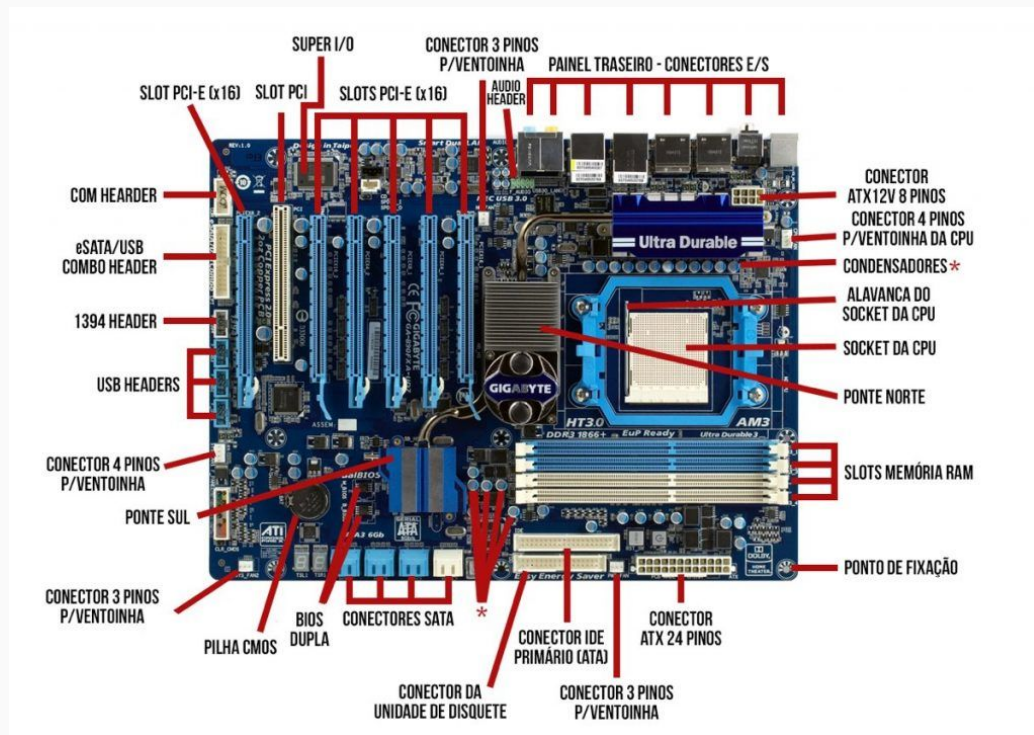
Manter a implementação de um SO o mais **simples** possível.

4. HISTÓRIA

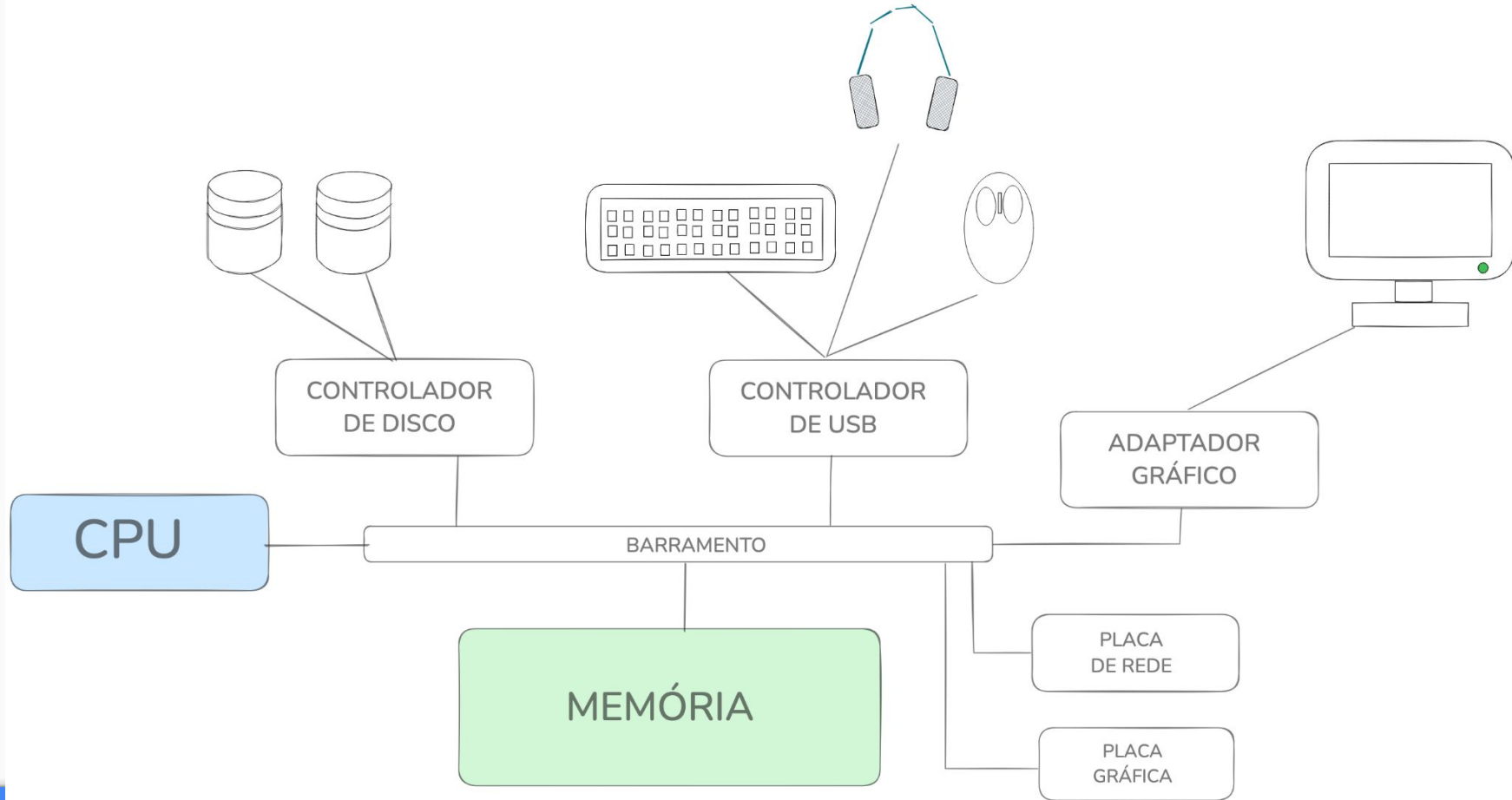
Aprender com o **passado** para prever o **futuro**, isto é: os **tradeoffs** mudam junto com a tecnologia.

REVISÃO DE ARQUITETURA

Imagem de uma placa mãe.

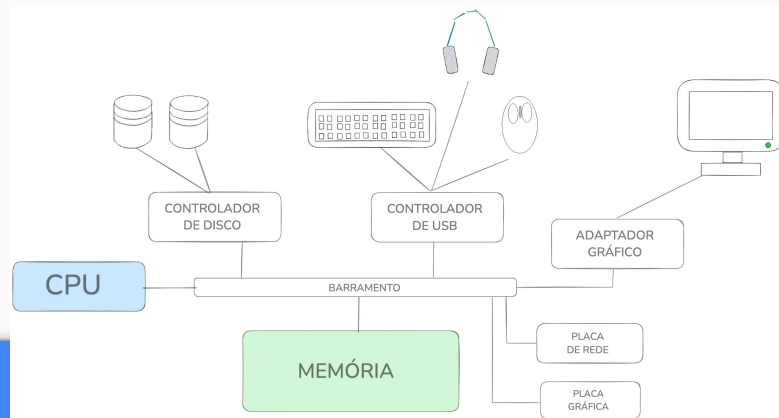


ARQUITETURA GENÉRICA DE UM COMPUTADOR



ARQUITETURA GENÉRICA DE UM COMPUTADOR

- CPU: O processador que faz a computação
- I/O: Terminal, discos, placa de vídeo, impressora, placa de rede, etc. Periféricos.
- Memória: acesso aleatório (RAM) e contém dados e programas usados pela CPU
- Barramento: Comunicação entre CPU, memória e periféricos.
 - *É possível enxergar todo o barramento a olho nu em uma placa de mãe atualmente?*



SERVIÇO x ARQUITETURA

Qual é o suporte de arquitetura que permite os serviços de um SO?

SERVIÇO

Proteção

Interrupções

Chamadas de sistema

I/O

Escalonamento, recuperação de erros

Sincronização

Memória Virtual

SUORTE NA ARQUITETURA

Modo kernel/superusuário, instruções protegidas, registradores com limite

Vetores de interrupções

Instruções de interrupção nativas

Interrupções e mapeamento de memória

Timer

Instruções atômicas

TLB (Translation lookaside buffers)

PROTEÇÃO

- A CPU suporta vários tipos de instrução assembly. Obs: NÃO PRECISAMOS DECORÁ-LAS!

PROTEÇÃO

- A CPU suporta vários tipos de instrução assembly. Obs: NÃO PRECISAMOS DECORÁ-LAS!

Exemplos de instruções:

- MOV [endereço], r1 (cópia de valor)
- ADD r1, r2 (adição)

- MOV CR1 (move o registrador de controle :scary:)
- IN, INS (lê uma string da entrada padrão)
- HLT (parada, do inglês "halt" = parar)
- LTR (carrega o registrador de tarefas)
- INT 0 (interrompe o software com sinal 0)

PROTEÇÃO

- A CPU suporta vários tipos de instrução assembly. Obs: NÃO PRECISAMOS DECORÁ-LAS!

Exemplos de instruções:

QUALQUER USUÁRIO PODE EXECUTAR:

- MOV [endereço], r1 (cópia de valor)
- ADD r1, r2 (adição)

SOMENTE USUÁRIOS PRIVILEGIADOS PODEM EXECUTAR:

- MOV CR1 (move o registrador de controle :scary:)
- IN, INS (lê uma string da entrada padrão)
- HLT (parada, do inglês "halt" = parar)
- LTR (carrega o registrador de tarefas)
- INT 0 (interrompe o software com sinal 0)

PROTEÇÃO

DE MODO GERAL, AS INSTRUÇÕES DA CPU CAEM EM 2 CATEGORIAS: INSTRUÇÕES COMUNS E INSTRUÇÕES SENSÍVEIS QUE SÓ PODEM SER **EXECUTADAS POR UM USUÁRIO COM PRIVILÉGIOS**.

NÃO QUEREMOS QUE QUALQUER USUÁRIO POSSA EMITIR UMA INSTRUÇÃO DE “HALT” (PARADA) PARA A CPU, POR EXEMPLO.

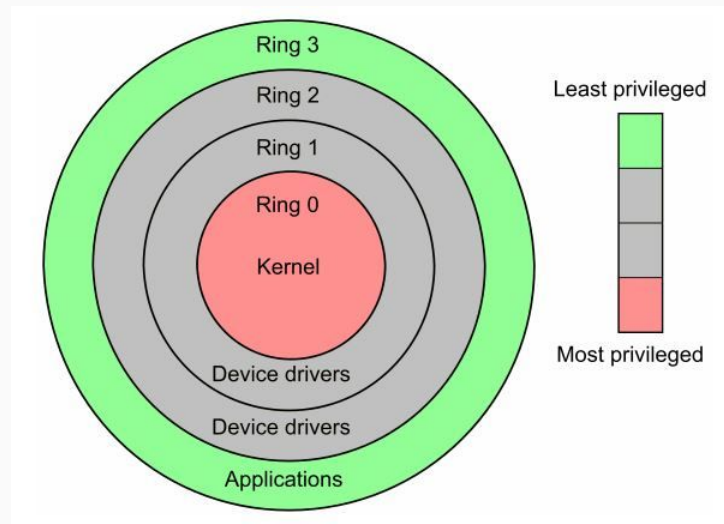
PROTEÇÃO: MODO KERNEL vs MODO USUÁRIO

- **MODO KERNEL:** Instruções sensíveis que só podem ser executadas pelo Kernel (núcleo do SO). Usuários não podem executá-las.

PROTEÇÃO: MODO KERNEL vs MODO USUÁRIO

- **MODO KERNEL:** Instruções sensíveis que só podem ser executadas pelo Kernel (núcleo do SO). Usuários não podem executá-las.
 - **Endereços de I/O** diretamente;
 - Instruções que manipulam o **estado da memória** (paginação, ponteiros etc)
 - Configurar o modo pra **kernel** ou **user**
 - Desativar ou ativar **interrupções**
 - **Parar** a máquina
- O hardware deve suportar pelo menos esses dois modos: **Kernel** e **User**

Como isso é feito na arquitetura?

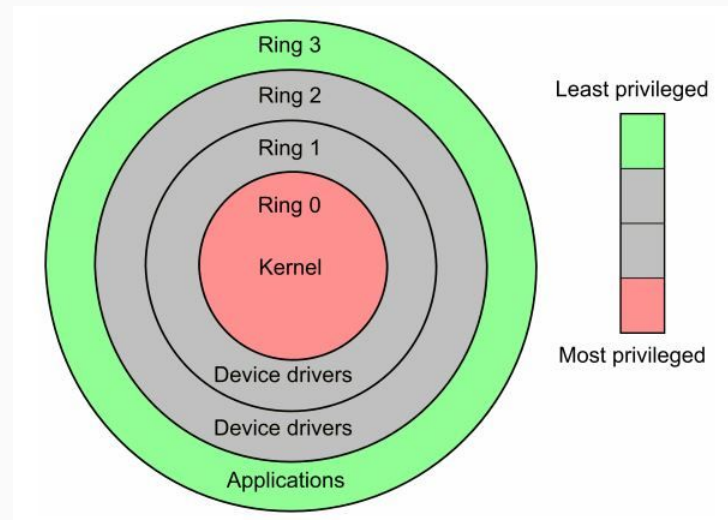


PROTEÇÃO: MODO KERNEL vs MODO USUÁRIO

- **MODO KERNEL:** Instruções sensíveis que só podem ser executadas pelo Kernel (núcleo do SO). Usuários não podem executá-las.
 - **Endereços de I/O** diretamente;
 - Instruções que manipulam o **estado da memória** (paginação, ponteiros etc)
 - Configurar o modo pra **kernel** ou **user**
 - Desativar ou ativar **interrupções**
 - **Parar** a máquina
- O hardware deve suportar pelo menos esses dois modos: **Kernel** e **User**

Como isso é feito na arquitetura?

- Um bit de status (0 ou 1) em registrador protegido na CPU indica qual é o modo
 - *Instruções protegidas só podem ser executadas no modo **kernel**, mesmo se você tentar executar um código assembly diretamente.*



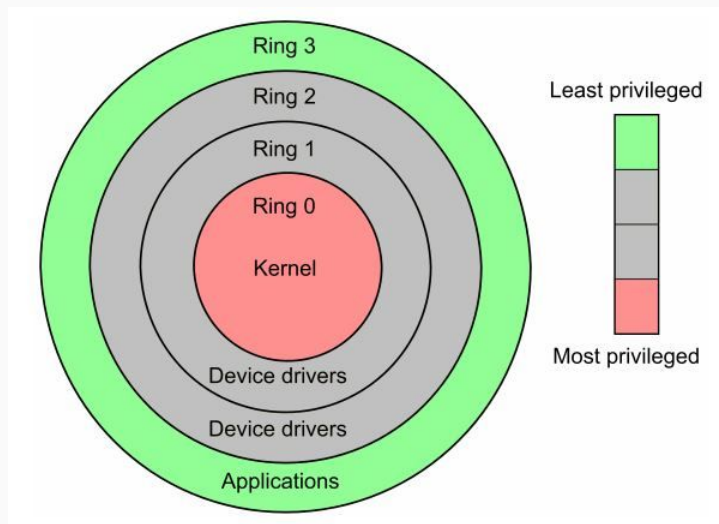
PROTEÇÃO: MODO KERNEL vs MODO USUÁRIO

Arquitetura x86

Ring 0: Modo kernel

Ring 3: Aplicações

Anéis intermediários: controle de prioridade entre dispositivos.



COMO CRUZAR A FRONTEIRA DA PROTEÇÃO?

Se um usuário comum não pode acessar endereços de I/O diretamente, como é que a gente escreve na saída padrão e lê da entrada padrão nos nossos programas?

COMO CRUZAR A FRONTEIRA DA PROTEÇÃO?

Se um usuário comum não pode acessar endereços de I/O diretamente, como é que a gente escreve na saída padrão e lê da entrada padrão nos nossos programas?

CHAMADA DE SISTEMA

Nós não fazemos I/O diretamente, mas pedimos ao SO que faça por nós.

CHAMADAS DE SISTEMA

- É um procedimento do sistema operacional que **executa comandos privilegiados** (I/O, por exemplo)

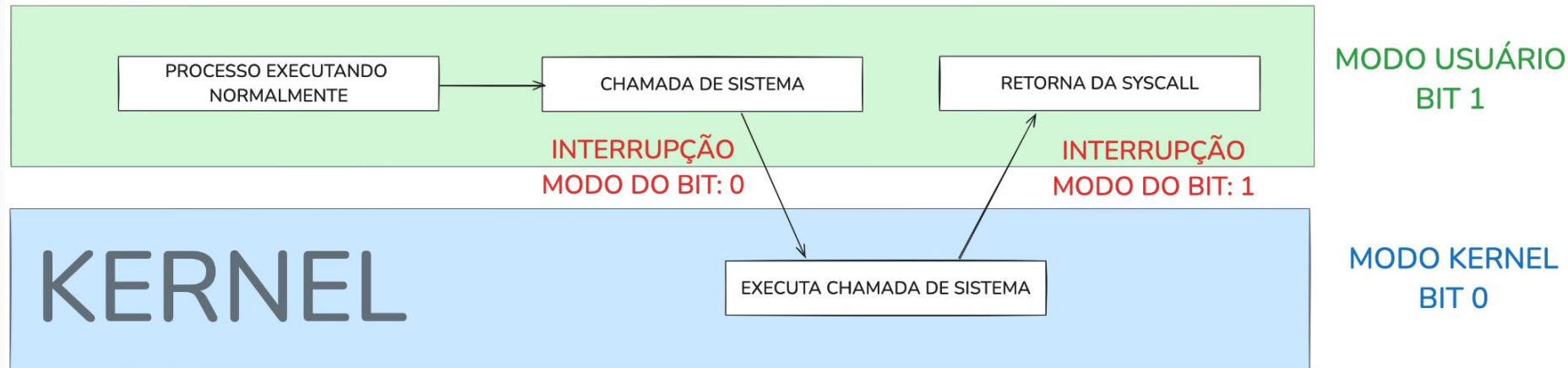
TAMBÉM É UMA **API** EXPORTADA PELO KERNEL, ou seja, podem ser consumida por usuários comuns (modo user).

- Causa uma **TRAP** (Interrupção síncrona): é um comando que mandamos ao Kernel para ele “tomar o controle da situação” e realizar trabalho em modo Kernel.
- Desafios:
 - O kernel deve salvar o estado da máquina antes da chamada (modo user) para poder restaurar o controle para o usuário após o final dos comandos em modo kernel
 - A arquitetura deve permitir que o SO realize esse processo com segurança

1. *Modo user realiza chamada de sistema*
2. *SO muda para modo kernel*
3. *executa comandos privilegiados*
4. *finaliza*
5. *restaura modo user*

CHAMADAS DE SISTEMA

PROCESSO DO USUÁRIO



CHAMADAS DE SISTEMA: EXEMPLOS

Gerenciamento de processo

<code>pid = fork()</code>	Cria um processo filho igual ao pai.
<code>pid = waitpid(pid, &endereço, opções)</code>	Espera um processo filho terminar.
<code>s = execve(nome, argumentos[], ptr ambiente)</code>	Substitui a imagem de um processo por outro.
<code>exit (status)</code>	Termina o a execução do processo

CHAMADAS DE SISTEMA: EXEMPLOS

Gerenciamento de arquivos

<code>fd = open(nome do arquivo, como, ...)</code>	Abre um arquivo para leitura, escrita ou ambos
<code>s = close(ptr arquivo)</code>	Fecha um arquivo aberto
<code>n = read(ptr arquivo, buffer, tamanho em bytes)</code>	Lê dados de um arquivo para dentro de um <i>buffer</i>
<code>n = write(ptr arquivo, buffer, tamanho em bytes)</code>	Escreve dados de um <i>buffer</i> em um arquivo.
<code>posição = lseek(ptr arquivo, offset, pra onde)</code>	Move o ponteiro do arquivo para outro endereço
<code>s = stat(nome arquivo)</code>	Pega informações sobre o status de um arquivo.

CHAMADAS DE SISTEMA

Unix vs Windows

O objetivo é permitir que o usuário faça operações sensíveis em escopo delimitado e espaço de memória controlado, sem permitir que o usuário realize qualquer operação que desejar.

UNIX	Win32	Descrição
fork	CreateProcess	Criar um novo processo
waitpid	WaitForSingleObject	Pode esperar que um processo termine
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminar a execução
open	CreateFile	Criar um arquivo ou abrir um arquivo existente
close	CloseHandle	Fechar um arquivo
read	ReadFile	Ler dados de um arquivo
write	WriteFile	Escrever dados em um arquivo
lseek	SetFilePointer	Mover o ponteiro do arquivo
stat	GetFileAttributesEx	Obter vários atributos de arquivo
mkdir	CreateDirectory	Criar um novo diretório
rmdir	RemoveDirectory	Remover um diretório vazio
link	(none)	Win32 não suporta links
unlink	DeleteFile	Destruir um arquivo existente
mount	(none)	Win32 não suporta montar
umount	(none)	Win32 não suporta desmontar
chdir	SetCurrentDirectory	Mudar o diretório de trabalho atual
chmod	(none)	Win32 não suporta segurança (embora o NT suporte)
kill	(none)	Win32 não suporta sinais
time	GetLocalTime	Obter a hora atual

PROTEÇÃO DE MEMÓRIA

A arquitetura deve prover suporte para que o S0 consiga:

- proteger os programas de usuário uns dos outros;
- proteger o S0 de programas de usuário.

A TÉCNICA MAIS SIMPLES É UTILIZAR REGISTRADORES DE **BASE** E **LIMITE**.

Antes do início do programa, o S0 determina o **espaço de memória do mesmo**.

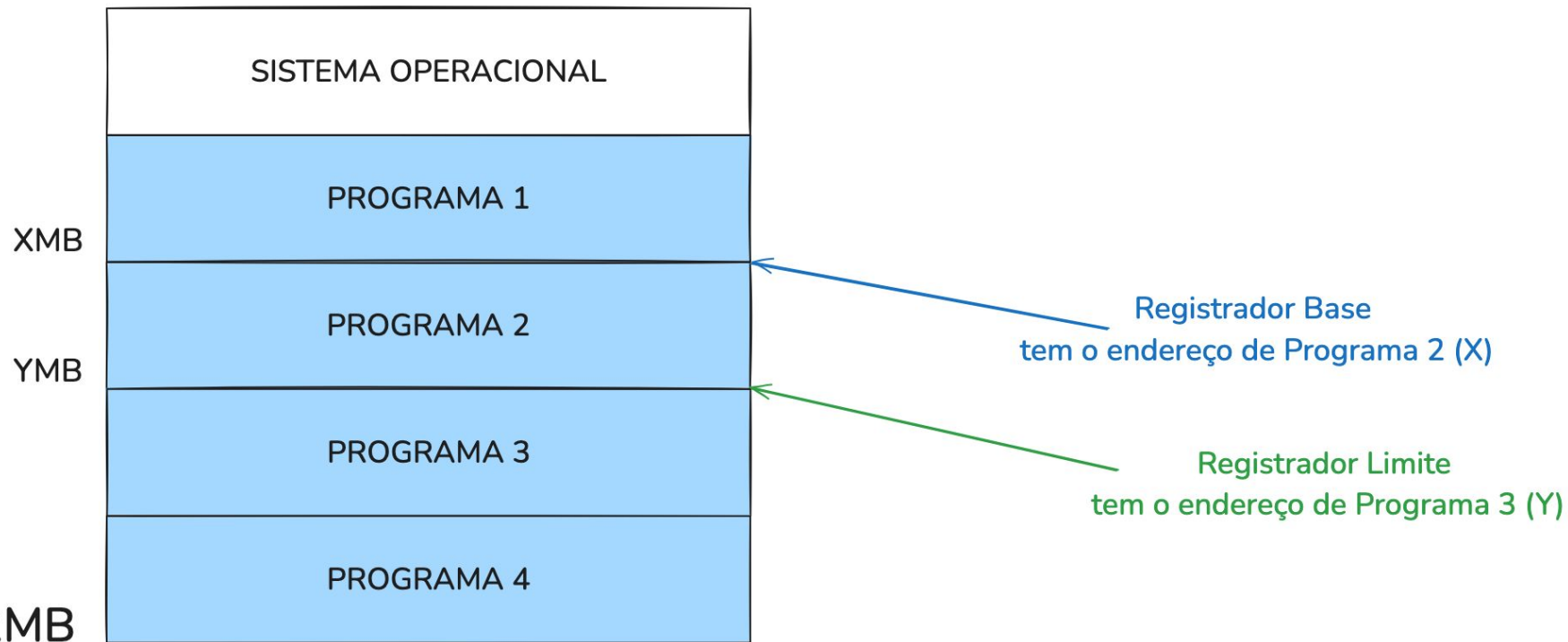
ou seja:

Os registradores de **ENDEREÇO BASE** e **ENDEREÇO LIMITE** são carregados pelo sistema operacional **antes de começar o programa**.

- A CPU verifica cada referência (instruções e dados), checando se nenhuma delas ultrapassa os limites, isto é, se existe alguma referência que está fora do intervalo *[base, limite]*

PROTEÇÃO DE MEMÓRIA

0MB



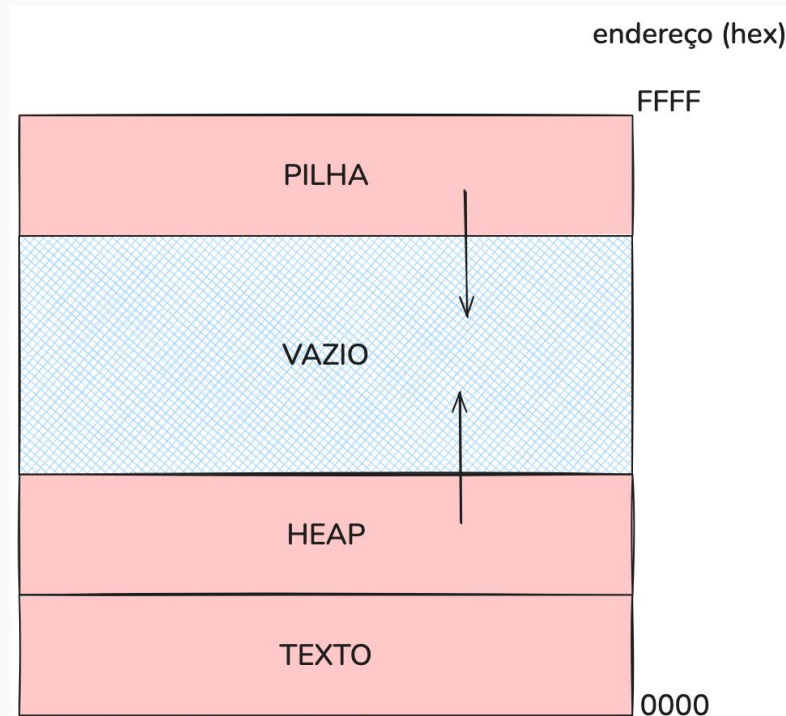
LEIAUTE (LAYOUT) DE MEMÓRIA

(simplificando), o espaço de endereços é dividido em 3 segmentos principais:

TEXT0: É o código do programa que está rodando. Cada endereço (32 bits) tem uma instrução do programa. Um apontador (*p) segue o programa na memória, executando instrução por instrução.

PILHA: Onde o programa acontece. Pode parecer nebuloso a diferença entre TEXT0 (do programa) e PILHA, mas imagine uma função recursiva sem caso base: ao executar, ela vai empilhar infinitamente o mesmo código. O seu código vive na região de TEXT0, mas a execução da função ocorre na pilha.

HEAP: Dados que vão sendo criados e destruídos pelo Processo/Programa, e vivem para além das funções (malloc/new)



REGISTRADORES

O QUE É UM REGISTRADOR?

REGISTRADORES

O QUE É UM REGISTRADOR?

Nome dedicado para um espaço de memória perto da CPU e gerenciado por ela.

REGISTRADORES

A CPU tem 32 registradores de propósito geral, mas alguns são especiais.

Alguns registradores nos permitem implementar esse layout de memória com mais facilidade -> são registradores usados pelo S0 para navegar na memória e controlar o espaço de memória de um programa.

PROPÓSITO GENÉRICO: r1, r2, r3...

PROPÓSITO ESPECIAL:

SP ou \$sp - Ponteiro para a pilha
(Stack Pointer)

FP ou \$fp - Ponteiro para a função atual.

PC ou *p - Contador do programa.

REGISTRADORES

A CPU tem 32 registradores de propósito geral, mas alguns são especiais.

Alguns registradores nos permitem implementar esse layout de memória com mais facilidade -> são registradores usados pelo SO para navegar na memória e controlar o espaço de memória de um programa.

PROPÓSITO GENÉRICO: r1, r2, r3...

PROPÓSITO ESPECIAL:

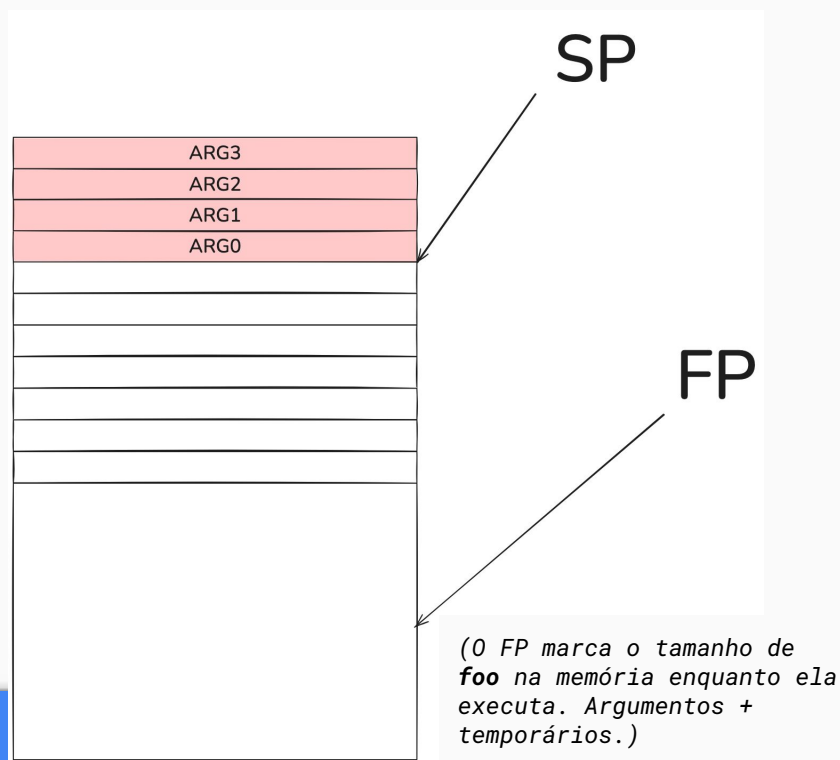
SP ou \$sp - Ponteiro para a pilha (Stack Pointer)

FP ou \$fp - Ponteiro para a função atual.

PC ou *p - Contador do programa.

(O PC vai percorrendo a região de texto e carregando as instruções do programa da memória pra CPU)

```
void main() {  
    ...  
    foo(arg0, arg1, arg2, arg3);  
    ...  
}
```



REGISTRADORES

CADA PROGRAMA VAI TER SEUS RESPECTIVOS VALORES DE SP, FP e PC. Ou seja, pra cada programa, esses registradores serão usados para controlar o espaço de memória daquele mesmo programa.

COMO UMA MESMA CPU CONSEGUE RODAR VÁRIOS PROGRAMAS?

REGISTRADORES

CADA PROGRAMA VAI TER SEUS RESPECTIVOS VALORES DE SP, FP e PC. Ou seja, pra cada programa, esses registradores serão usados para controlar o espaço de memória daquele mesmo programa.

COMO UMA MESMA CPU CONSEGUE RODAR VÁRIOS PROGRAMAS?

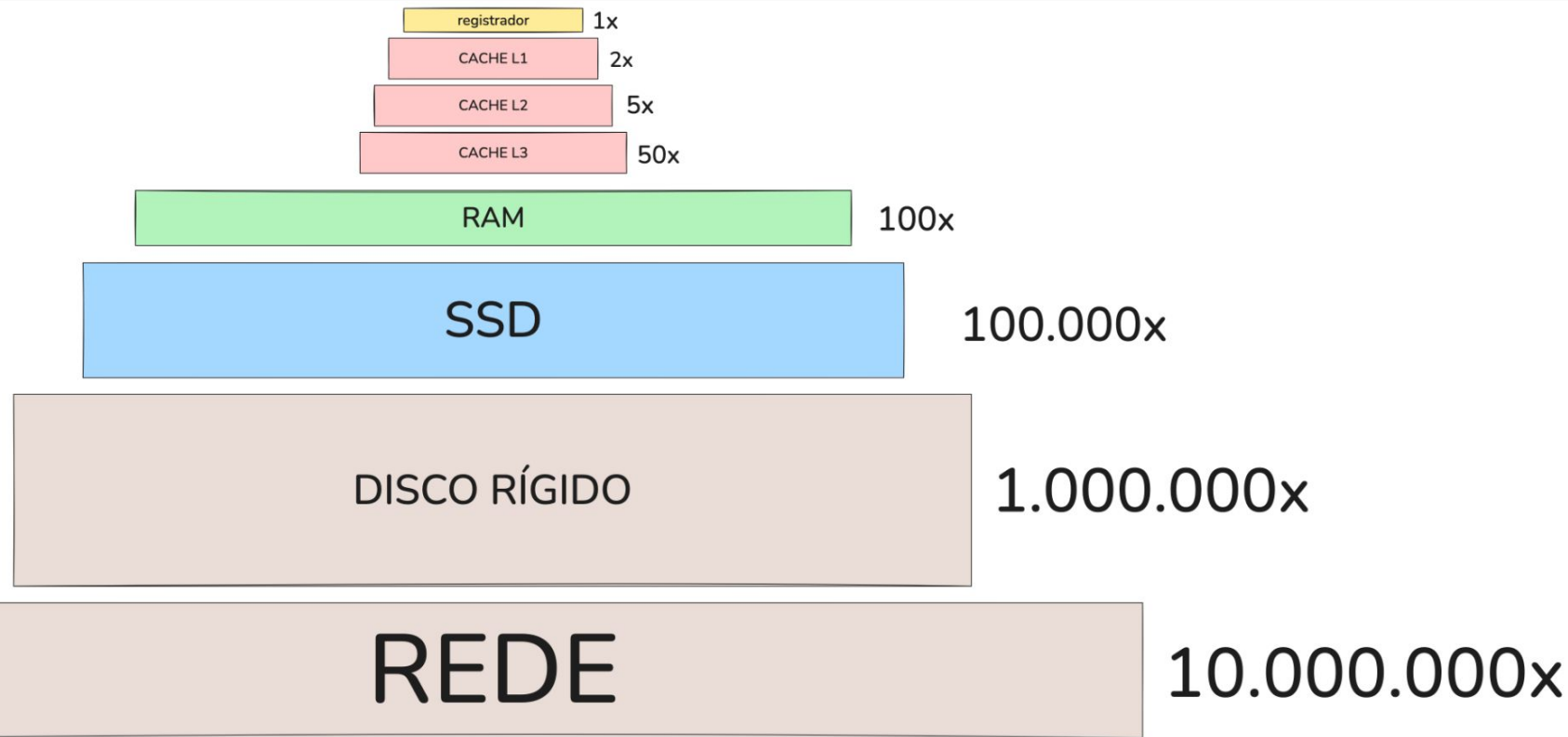
MUDANÇA DE CONTEXTO

A cpu INTERROMPE a execução do processo A para executar o processo B.

- A CPU GUARDA OS 3 VALORES (SP, FP, PC) do processo A;
(onde é o melhor local para guardar esses valores?)
- Carrega os 3 valores (SP, FP e PC) do processo B nos registradores especiais;
- Executa o processo B até precisar mudar de contexto novamente;
 - repete;

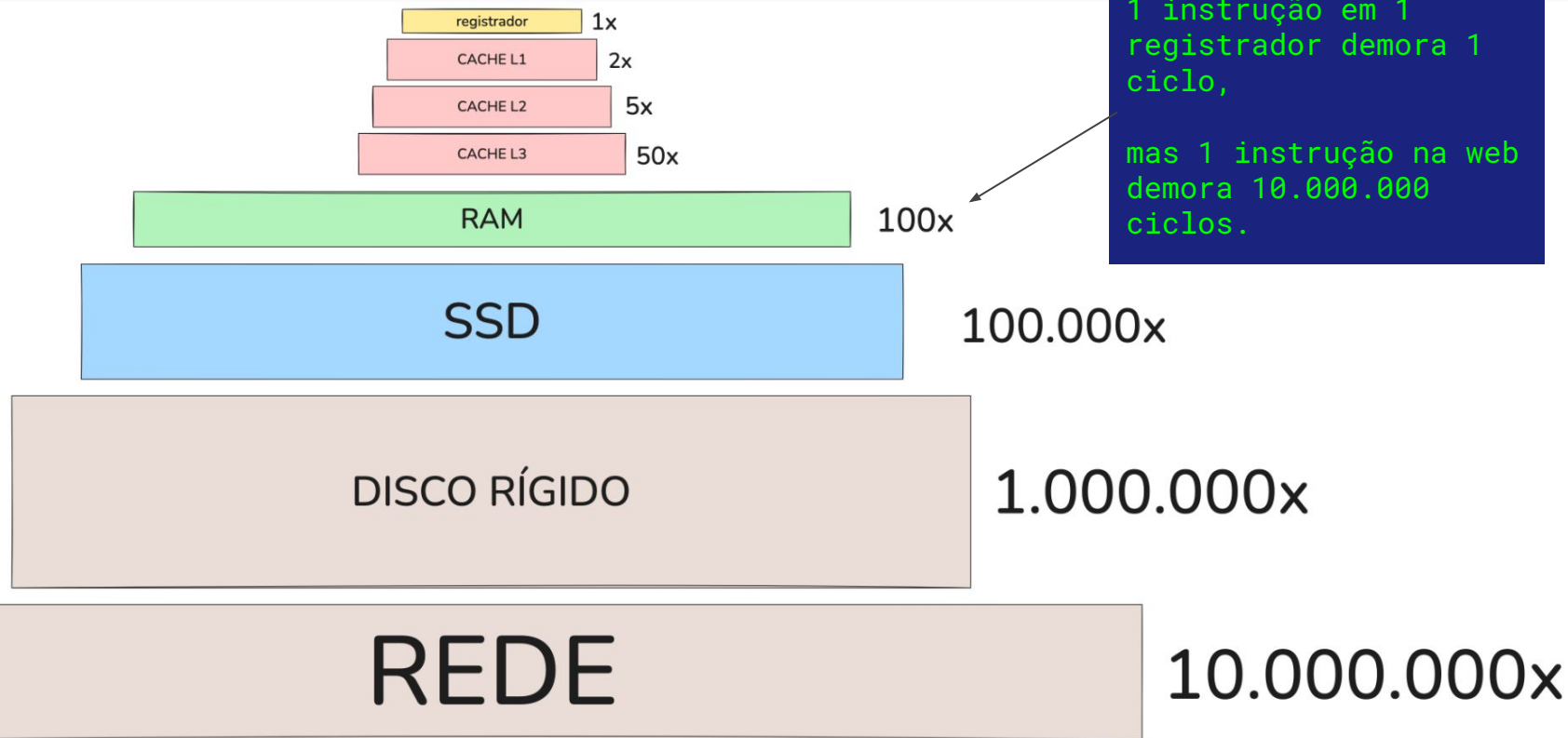
HIERARQUIA DE MEMÓRIA

- Mais alto na hierarquia: menor, mais rápido, mais caro \$
- Mais baixo na hierarquia: maior, mais lento, mais barato \$



HIERARQUIA DE MEMÓRIA

- Mais alto na hierarquia: menor, mais rápido, mais caro \$
- Mais baixo na hierarquia: maior, mais lento, mais barato \$



CACHES

- ACESSO À MEMÓRIA PRINCIPAL (RAM) É MUITO CARO.

~ 100 CICLOS (lento, mas relativamente barato em comparação com SSD, Disco e Rede)

- CACHES são pequenas, rápidas e caríssimas.

CACHES

- ACESSO À MEMÓRIA PRINCIPAL (RAM) É MUITO CARO.

~ 100 CICLOS (lento, mas relativamente barato em comparação com SSD, Disco e Rede)

- CACHES são pequenas, rápidas e caríssimas.
- Elas guardam dados ou instruções que foram recentemente acessados
- Tem tamanhos diferentes e níveis diferentes

L1

Dentro do chip (on-chip), dezenas de KB

L2

Dentro ou próximo ao chip, alguns MB

L3

Bem largo, vive no barramento, muitos MB, compartilhado por todos os núcleos de um mesmo processador.

CACHES

caches são exclusivamente gerenciadas pelo hardware
(o SO não interage com endereços da cache explicitamente)

L1

Dentro do chip (on-chip), **80KB**

L2

Dentro ou próximo ao chip, **1.25MB**

L3

Bem largo, vive no barramento, **30MB**, compartilhado por todos os núcleos de um mesmo processador.

i9-12900K

INTERRUPÇÕES / ARMADILHAS (*TRAPS*)

TRAPS SÃO COMO EXCEÇÕES!

INTERRUPÇÕES / ARMADILHAS (*TRAPS*)

TRAPS SÃO COMO EXCEÇÕES!

- TRAPS ou INTERRUPÇÕES são condições especiais detectadas pela arquitetura.
 - Exemplos:
 - page fault
 - escrever em região read-only
 - overflow
 - chamadas de sistema

INTERRUPÇÕES / ARMADILHAS (*TRAPS*)

TRAPS SÃO COMO EXCEÇÕES!

- TRAPS ou INTERRUPÇÕES são condições especiais detectadas pela arquitetura.
 - Exemplos:
 - `page fault` - Quando a memória virtual falha em encontrar uma página na RAM e deve buscar do disco
 - `escrever em região read-only` (somente leitura): quando uma aplicação de usuário tenta escrever em uma região que o SO só permite leitura
 - `overflow` quando uma computação extrapola os limites (boundaries) da memória
 - `chamadas de sistema` slides anteriores

INTERRUPÇÕES / ARMADILHAS (*TRAPS*)

TRAPS SÃO COMO EXCEÇÕES!

- TRAPS ou INTERRUPÇÕES são condições especiais detectadas pela arquitetura.
 - Exemplos:
 - *page fault* - Quando a memória virtual falha em encontrar uma página na RAM e deve buscar do disco
 - *escrever em região read-only* (somente leitura): quando uma aplicação de usuário tenta escrever em uma região que o SO só permite leitura
 - *overflow* quando uma computação extrapola os limites (boundaries) da memória
 - *chamadas de sistema* slides anteriores
- Ao detectar uma TRAP (interrupção), o hardware deve:
 1. *Salvar o estado do processo (Pilha, ponteiros SP, FP, PC)*
 2. *Transferir controle para o kernel (rotina do SO)*
 - Detalhes de implementação (veremos em aulas futuras):*
 1. *A CPU mantém um vetor de interrupções*
 2. *Quando há uma interrupção, o endereço do código a ser executado está escrito nesse vetor*
 3. *O SO muda de contexto para executar em modo kernel e depois retorna para o processo*

INTERRUPÇÕES / ARMADILHAS (*TRAPS*)

A implementação de TRAPS é uma otimização.

Há um **vetor** que contém todas as traps possíveis (endereço ilegal, violação de memória, chamada de sistema).

Quando uma aplicação dispara uma trap, esse vetor captura o fluxo de execução e encontra o código Kernel relativo à interrupção em **0(1)** pois todas as interrupções estão indexadas.

Índice do vetor	Endereço do código a ser executado pelo SO	Tipo de interrupção
0	0x00080000	Endereço ilegal
1	0x00100000	Violação de memória
2	0x00100480	Instrução ilegal
3	0x00123010	Chamada de sistema

INTERRUPÇÕES / ARMADILHAS (*TRAPS*)

A implementação de TRAPS é uma otimização.

Há um **vetor** que contém todas as traps possíveis (endereço ilegal, violação de memória, chamada de sistema).

Quando uma aplicação dispara uma trap, esse vetor captura o fluxo de execução e encontra o código Kernel relativo à interrupção em $O(1)$ pois todas as interrupções estão indexadas.


Não precisamos entender como é feito - mas precisamos entender que TRAPS tornam o sistema de interrupções + rápido e prático.

Sem as traps, o SO teria que adicionar código assembly desnecessário para lidar com todo tipo de situação (interrupção) em aplicação de usuário.

CONTROLE DE I/O (ENTRADA/SAÍDA)

TODO DISPOSITIVO I/O TEM UM MICROCHIP (PROCESSADOR) QUE PERMITE COMPUTAÇÃO BÁSICA E COMUNICAÇÃO COM A MÁQUINA HOSPEDEIRA.

CONTROLE DE I/O (ENTRADA/SAÍDA)

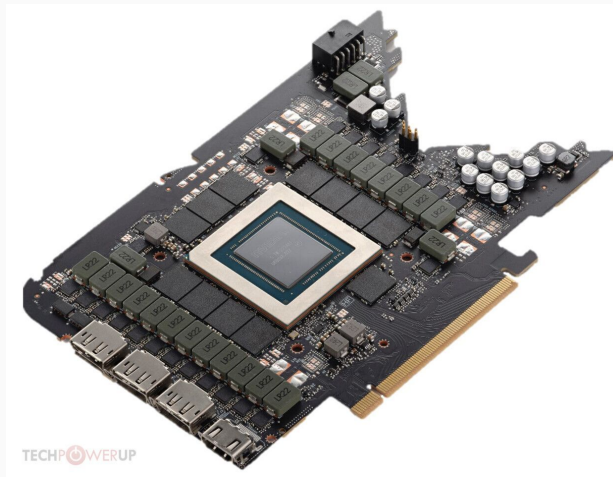
- TODO dispositivo I/O tem um pequeno processador que permite rodar de forma autônoma.
- A CPU emite um COMANDO para os dispositivos I/O, e continua.
 I/O é lento!!!
- Quando o dispositivo I/O completa o comando, ele lança uma...

CONTROLE DE I/O (ENTRADA/SAÍDA)

- TODO dispositivo I/O tem um pequeno processador que permite rodar de forma autônoma.
- A CPU emite um COMANDO para os dispositivos I/O, e continua.
- Quando o dispositivo I/O completa o comando, ele lança uma... INTERRUPÇÃO
- CPU pára o que estiver fazendo para executar a interrupção disparada pelo dispositivo I/O

CONTROLE DE I/O (ENTRADA/SAÍDA)

O melhor exemplo é a placa de vídeo - é um dispositivo autônomo, complexo, com seu próprio conjunto de instruções.



Ainda assim, para se comunicar com a máquina hospedeira, deve emitir uma interrupção para o sistema.

(CPU: dispositivo principal/controlador ==> GPU: dispositivo secundário/subordinado)

CONTROLE DE I/O (ENTRADA/SAÍDA)

PERGUNTA: Com qual frequência ocorrem interrupções no S0?

CONTROLE DE I/O (ENTRADA/SAÍDA)

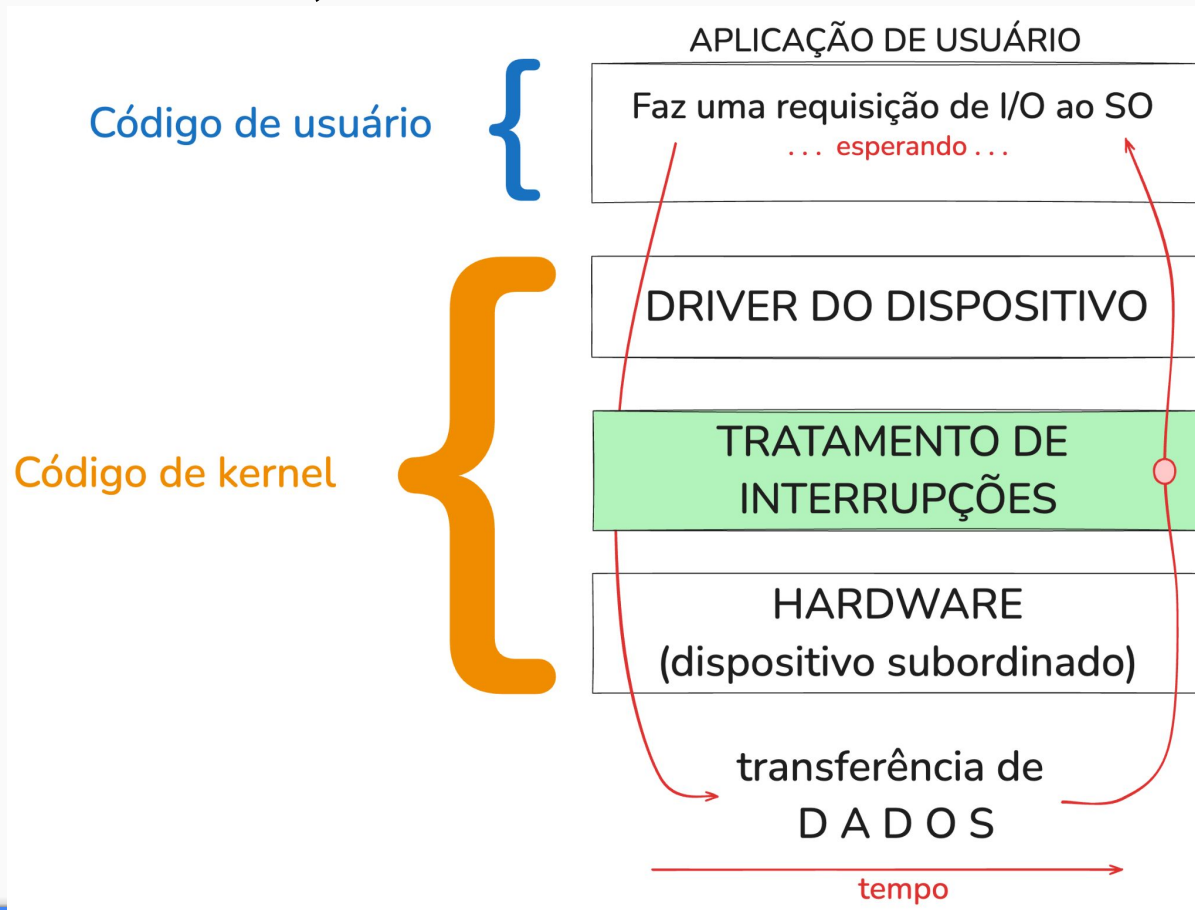
PERGUNTA: Com qual frequência ocorrem interrupções no S0?

Um número exato (%) de código executado por interrupções varia de acordo com o uso do sistema operacional.

Mas pode-se dizer que interrupções ocorrem **o tempo todo, com muita frequência.**

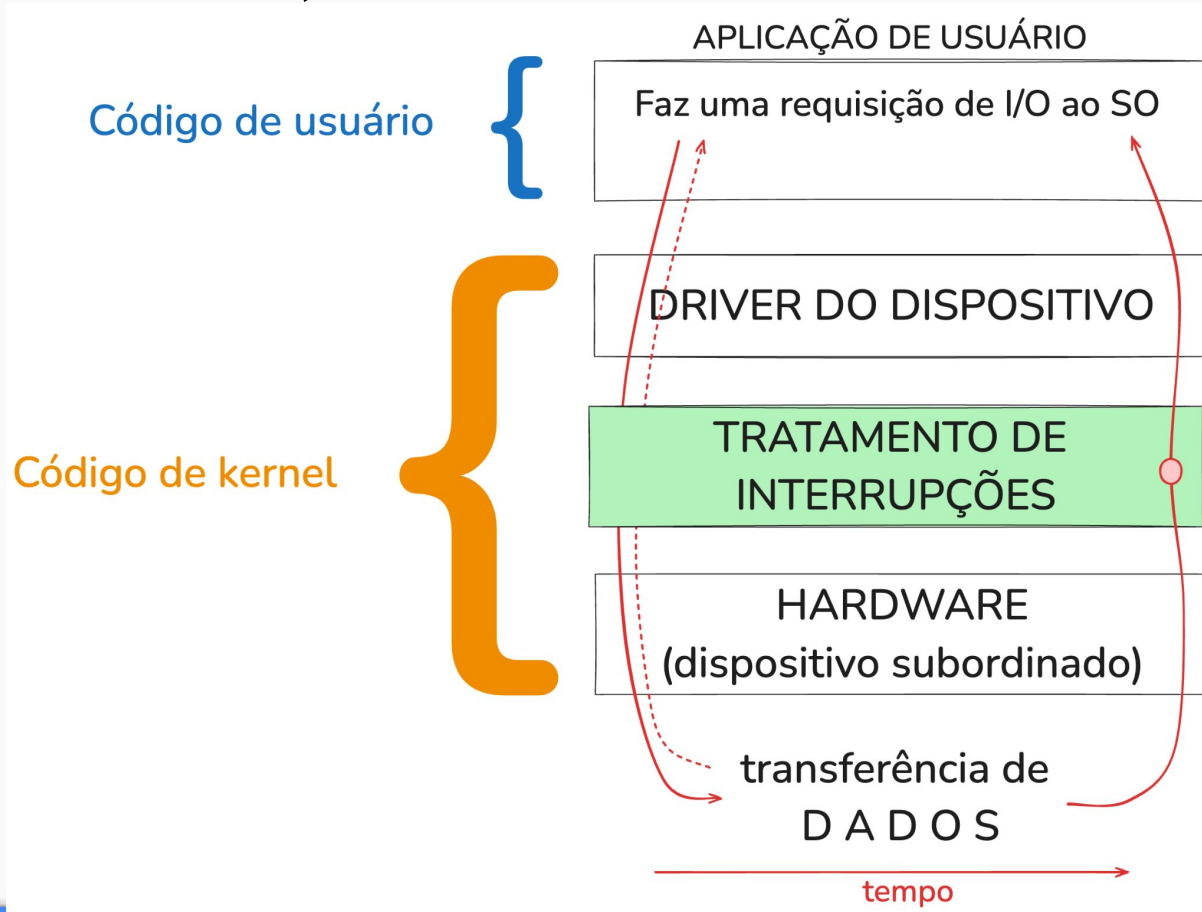
MÉTODOS DE I/O (ENTRADA/SAÍDA)

Método SÍNCRONO



MÉTODOS DE I/O (ENTRADA/SAÍDA)

Método ASSÍNCRONO



MÉTODOS DE I/O (ENTRADA/SAÍDA)

Método DE MAPEAMENTO EM MEMÓRIA

- Suponha que o usuário queira ler um grande arquivo.
 - Geralmente, o sistema de arquivos vai ter um buffer que é preenchido com os dados que o usuário quer ler, e emite uma interrupção.
 - Porém,
 - Tamanho do arquivo: 1GB
 - Tamanho do buffer 1MB

Seria ineficiente emitir 1 interrupção a cada 1MB de arquivo.

MÉTODOS DE I/O (ENTRADA/SAÍDA)

Método DE MAPEAMENTO EM MEMÓRIA

- Suponha que o usuário queira ler um grande arquivo.
 - Geralmente, o sistema de arquivos vai ter um buffer que é preenchido com os dados que o usuário quer ler, e emite uma interrupção.
 - Porém,
 - Tamanho do arquivo: 1GB
 - Tamanho do buffer 1MB

Seria ineficiente emitir 1 interrupção a cada 1MB de arquivo.

Solução: permitir que o dispositivo I/O se comunique diretamente com a memória!

MÉTODOS DE I/O (ENTRADA/SAÍDA)

Método DE MAPEAMENTO EM MEMÓRIA

- Permite acesso direto do dispositivo I/O à memória do sistema
- O Computador tem uma parte de memória reservada (física) e coloca o gerenciador do dispositivo (driver) nessa região.
 - Por exemplo, todos os bits de 1 frame (4k) para uma GPU poder escrever nessa região da memória cada frame diretamente.
- Acesso se torna rápido e tão conveniente como se fosse a CPU escrevendo em memória.

TEMPORIZADOR (TIMER)

A placa mãe tem um temporizador.

Aplicações:

TEMPORIZADOR (TIMER)

A placa mãe tem um temporizador.

Aplicações:

- Hora do dia
- contabilidade de recursos - quanto tempo cada processo está tomando da CPU
- CPU protegida de ser monopolizada - a cada 100ms, por exemplo, a CPU pode forçar mudança de contexto.

SINCRONIZAÇÃO

- Por fim, vale ressaltar que **INTERRUPÇÕES INTERFEREM NO FLUXO DE EXECUÇÃO.**
- O SO deve ser capaz de sincronizar processos que estão cooperando entre si.
- A arquitetura deve prover um meio de garantir que **algumas instruções NÃO SEJAM INTERROMPIDAS** por aplicação de usuário (ler/escrever, modificar arquivo, por ex).

SINCRONIZAÇÃO

- Por fim, vale ressaltar que **INTERRUPÇÕES INTERFEREM NO FLUXO DE EXECUÇÃO.**
- O SO deve ser capaz de sincronizar processos que estão cooperando entre si.
- A arquitetura deve prover um meio de garantir que **algumas instruções NÃO SEJAM INTERROMPIDAS** por aplicação de usuário (ler/escrever, modificar arquivo, por ex).
 - **Solução 1** - Arquitetura DESATIVA interrupções por um tempo, executa código crítico, e reativa de novo
 - **Solução 2** - Instruções atômicas

MEMÓRIA VIRTUAL

- Memória virtual permite rodar programas SEM CARREGAR ELES INTEIRAMENTE NA MEMÓRIA DE UMA SÓ VEZ.

MEMÓRIA VIRTUAL

- Memória virtual permite rodar programas SEM CARREGAR ELES INTEIRAMENTE NA MEMÓRIA DE UMA SÓ VEZ.
- Pedacos do programa são carregados na memória por demanda à medida que o usuário necessitar.

MEMÓRIA VIRTUAL

- Memória virtual permite rodar programas SEM CARREGAR ELES INTEIRAMENTE NA MEMÓRIA DE UMA SÓ VEZ.
- Pedacos do programa são carregados na memória por demanda à medida que o usuário necessitar.
- O SO deve manter um registro desses pedacos de programa que estão na memória física (RAM) e pedacos que ainda estão no disco ou SSD.

MEMÓRIA VIRTUAL

- Memória virtual permite rodar programas SEM CARREGAR ELES INTEIRAMENTE NA MEMÓRIA DE UMA SÓ VEZ.
- Pedacos do programa são carregados na memória por demanda à medida que o usuário necessitar.
- O SO deve manter um registro desses pedacos de programa que estão na memória física (RAM) e pedacos que ainda estão no disco ou SSD.
- Para que esses pedacos sejam carregados e localizados, a arquitetura provê um buffer de tradução (TLB) que traduz endereços em Memória virtual para endereços em memória física.

RESUMO

- A arquitetura deve prover detalhes de implementação que facilitam a construção do S0
- O S0 faz a interface, mas também requer da arquitetura que siga alguns requisitos.
- Essa combinação (Arquitetura + Software) resulta em **vários serviços que o S0 provê.**

PERGUNTAS?

REFERÊNCIAS

- **TANENBAUM, Andrew.** Sistemas operacionais modernos.
- **SILBERSCHATZ, Abraham et al.** Fundamentos de sistemas operacionais: princípios básicos.
- **MACHADO, Francis; MAIA, Luiz Paulo.** Arquitetura de Sistemas Operacionais.
- **CARISSIMI, Alexandre et al.** Sistemas operacionais.