

# SO: Introdução

## Projetos de Sistemas Operacionais

Prof. Dr. Denis M. L. Martins

Engenharia de Computação: 5º Semestre

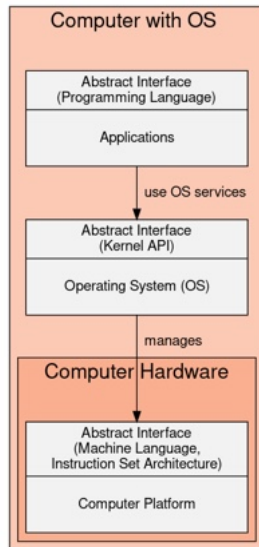


Image credits: Jens Lechtenböger

# Introdução

- Explicar o conceito de Sistema Operacional e seus serviços típicos.
- Explicar o conceito de **kernel**, incluindo a API de chamadas de sistema, modo usuário e modo kernel.
- Explicar conceitos e relações entre programa, processo, thread e multitarefa.

Parte do material apresentado a seguir foi adaptado de *IT Systems – Open Educational Resource*, disponível em <https://oer.gitlab.io/oer-courses/it-systems/>, produzido por [Jens Lechtenböger](#), e distribuído sob a licença [CC BY-SA 4.0](#).

Questão: Quais afirmações são corretas sobre conceitos de Sistemas Operacionais?

- a) O código do kernel é armazenado na **ROM**.
- b) As aplicações podem acessar mais memória RAM se precisarem.
- c) O sistema operacional gerencia as aplicações como **processos**.
- d) O sistema operacional aloca memória RAM como parte da **memória virtual**.
- e) As threads de um processo compartilham recursos.
- f) O sistema operacional protege arquivos com **criptografia**.

Questão: Quais afirmações são corretas sobre conceitos de Sistemas Operacionais?

- a) O código do kernel é armazenado na **ROM**.
- b) As aplicações podem acessar mais memória RAM se precisarem.
- c) O sistema operacional gerencia as aplicações como **processos**.
- d) O sistema operacional aloca memória RAM como parte da **memória virtual**.
- e) As threads de um processo compartilham recursos.
- f) O sistema operacional protege arquivos com **criptografia**.

**Resposta:** São verdadeiras: (c), (d), (e).

**Software** que:

- utiliza recursos de hardware de um sistema computacional, e
- provê suporte para execução de outros softwares.

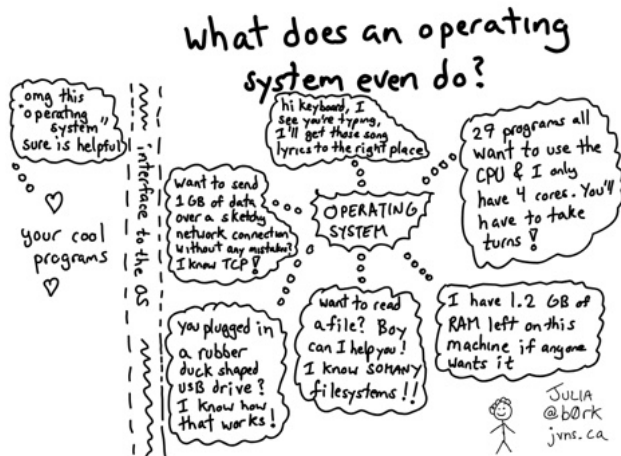


Figura 1: O que um SO faz. Créditos: [Julia Evans](#).

## Software que:

- utiliza recursos de hardware de um sistema computacional, e
- provê suporte para execução de outros softwares.

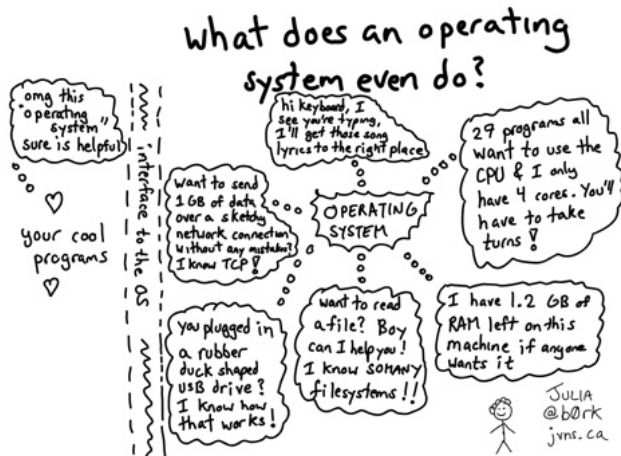


Figura 1: O que um SO faz. Créditos: [Julia Evans](#).



Diferentes sistemas para diferentes cenários:

- **Mainframes:** [BS2000/OSD](#), [GCOS](#), [z/OS](#)
- **PCs:** [MS-DOS](#), [GNU/Linux](#), [MacOS](#), [Redox](#), [Windows](#)
- **Dispositivos móveis:**
  - ▶ Variantes de outros sistemas operacionais
  - ▶ Desenvolvimentos independentes, por exemplo: BlackBerry ([BlackBerry 10](#) baseado em QNX, descontinuado), [Google Fuchsia](#), [Symbian](#) (Nokia, sistema operacional para smartphones mais popular até 2010, agora substituído)
- **Dispositivos para jogos**
- **Sistemas operacionais em tempo real (RTOS):**
  - ▶ Sistemas embarcados
  - ▶ Variantes do [L4](#), [FreeRTOS](#), [QNX](#), [VxWorks](#)

Diferentes sistemas para diferentes cenários:

- **Mainframes:** [BS2000/OSD](#), [GCOS](#), [z/OS](#)
- **PCs:** [MS-DOS](#), [GNU/Linux](#), [MacOS](#), [Redox](#), [Windows](#)
- **Dispositivos móveis:**
  - ▶ Variantes de outros sistemas operacionais
  - ▶ Desenvolvimentos independentes, por exemplo: BlackBerry ([BlackBerry 10](#) baseado em QNX, descontinuado), [Google Fuchsia](#), [Symbian](#) (Nokia, sistema operacional para smartphones mais popular até 2010, agora substituído)
- **Dispositivos para jogos**
- **Sistemas operacionais em tempo real (RTOS):**
  - ▶ Sistemas embarcados
  - ▶ Variantes do [L4](#), [FreeRTOS](#), [QNX](#), [VxWorks](#)

Diferentes sistemas para diferentes cenários:

- **Mainframes:** [BS2000/OSD](#), [GCOS](#), [z/OS](#)
- **PCs:** [MS-DOS](#), [GNU/Linux](#), [MacOS](#), [Redox](#), [Windows](#)
- **Dispositivos móveis:**
  - ▶ Variantes de outros sistemas operacionais
  - ▶ Desenvolvimentos independentes, por exemplo: BlackBerry ([BlackBerry 10](#) baseado em QNX, descontinuado), [Google Fuchsia](#), [Symbian](#) (Nokia, sistema operacional para smartphones mais popular até 2010, agora substituído)
- **Dispositivos para jogos**
- **Sistemas operacionais em tempo real (RTOS):**
  - ▶ Sistemas embarcados
  - ▶ Variantes do [L4](#), [FreeRTOS](#), [QNX](#), [VxWorks](#)

# Sistemas Operacionais

- **Gerenciamento de multitarefa:** O sistema operacional permite a execução simultânea de múltiplas computações, gerenciando a alternância entre elas e garantindo a retomada correta de cada uma.
- **Controle de concorrência:** Regula a interação entre processos concorrentes, impedindo acessos indevidos a estruturas de dados e fornecendo áreas de memória isoladas para diferentes computações.
- **Interação entre computações assíncronas:** Suporta a troca de informações entre computações que não são executadas ao mesmo tempo, por meio de sistemas de arquivos e armazenamento de longo prazo.
- **Interação via rede:** Facilita a comunicação entre computações distribuídas em diferentes sistemas computacionais através de redes, sendo um recurso essencial em sistemas operacionais modernos.


- **Gerenciamento de multitarefa:** O sistema operacional permite a execução simultânea de múltiplas computações, gerenciando a alternância entre elas e garantindo a retomada correta de cada uma.
- **Controle de concorrência:** Regula a interação entre processos concorrentes, impedindo acessos indevidos a estruturas de dados e fornecendo áreas de memória isoladas para diferentes computações.
- **Interação entre computações assíncronas:** Suporta a troca de informações entre computações que não são executadas ao mesmo tempo, por meio de sistemas de arquivos e armazenamento de longo prazo.
- **Interação via rede:** Facilita a comunicação entre computações distribuídas em diferentes sistemas computacionais através de redes, sendo um recurso essencial em sistemas operacionais modernos.

- **Gerenciamento de multitarefa:** O sistema operacional permite a execução simultânea de múltiplas computações, gerenciando a alternância entre elas e garantindo a retomada correta de cada uma.
- **Controle de concorrência:** Regula a interação entre processos concorrentes, impedindo acessos indevidos a estruturas de dados e fornecendo áreas de memória isoladas para diferentes computações.
- **Interação entre computações assíncronas:** Suporta a troca de informações entre computações que não são executadas ao mesmo tempo, por meio de sistemas de arquivos e armazenamento de longo prazo.
- **Interação via rede:** Facilita a comunicação entre computações distribuídas em diferentes sistemas computacionais através de redes, sendo um recurso essencial em sistemas operacionais modernos.

- **Gerenciamento de multitarefa:** O sistema operacional permite a execução simultânea de múltiplas computações, gerenciando a alternância entre elas e garantindo a retomada correta de cada uma.
- **Controle de concorrência:** Regula a interação entre processos concorrentes, impedindo acessos indevidos a estruturas de dados e fornecendo áreas de memória isoladas para diferentes computações.
- **Interação entre computações assíncronas:** Suporta a troca de informações entre computações que não são executadas ao mesmo tempo, por meio de sistemas de arquivos e armazenamento de longo prazo.
- **Interação via rede:** Facilita a comunicação entre computações distribuídas em diferentes sistemas computacionais através de redes, sendo um recurso essencial em sistemas operacionais modernos.



# Funções de um Sistema Operacional

- O núcleo (*kernel*) de um SO oferece uma API (*Application Programming Interface*)
  - ▶ Expõe um conjunto de interfaces para os serviços do OS (system calls).
  - ▶ Deixa transparente para o programador uma série de detalhes (operações) de baixo nível.
  - ▶ Veja também o vídeo [What is a Kernel?](#) do canal *Techquickie* no YouTube .
- Interface com o usuário (UI) é um item não obrigatório.
  - ▶ UI: processos usando funcionalidade do núcleo do SO para gerenciar entrada do usuário, iniciar programas, produzir saída, ...
  - ▶ Exemplos de UIs: linha de comando, Explorer (Windows), ambientes de desktop para GNU/Linux, [assistentes virtuais](#).

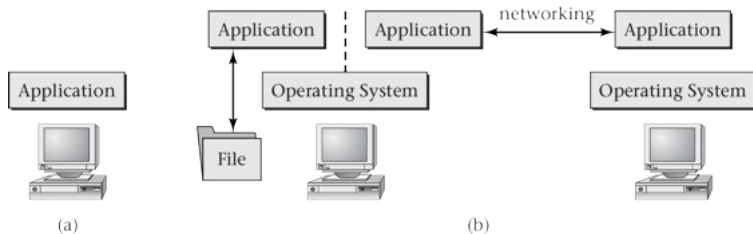



Figura 2: Em (a), baixo nível de abstração. Em (b), alto nível de abstração. Imagem: [Max Hailperin](#).

# Funções de um Sistema Operacional

- O núcleo (*kernel*) de um SO oferece uma API (*Application Programming Interface*)
  - ▶ Expõe um conjunto de interfaces para os serviços do OS (system calls).
  - ▶ Deixa transparente para o programador uma série de detalhes (operações) de baixo nível.
  - ▶ Veja também o vídeo [What is a Kernel?](#) do canal *Techquickie* no YouTube .
- Interface com o usuário (UI) é um item não obrigatório.
  - ▶ UI: processos usando funcionalidade do núcleo do SO para gerenciar entrada do usuário, iniciar programas, produzir saída, ...
  - ▶ Exemplos de UIs: linha de comando, Explorer (Windows), ambientes de desktop para GNU/Linux, [assistentes virtuais](#).

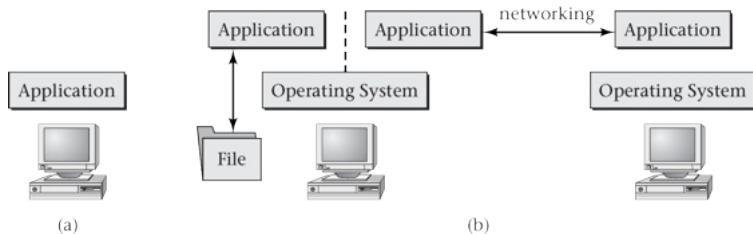
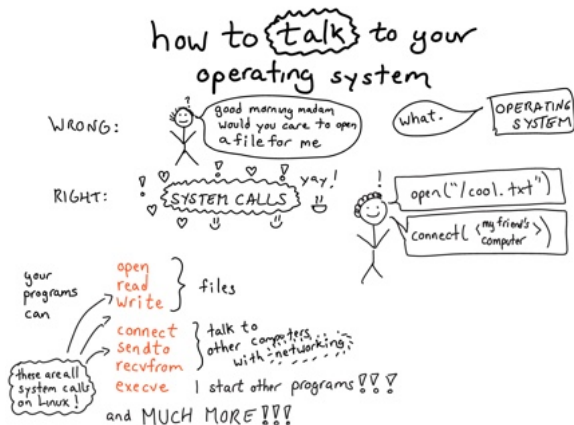


Figura 2: Em (a), baixo nível de abstração. Em (b), alto nível de abstração. Imagem: [Max Hailperin](#).

- **Chamada de sistema** = função = parte da API do kernel
- Implementação de serviços do sistema operacional, como:
  - ▶ Execução de processos
  - ▶ Alocação de memória principal
  - ▶ Acesso a recursos de hardware (exemplo: teclado, rede, arquivos e disco, placa de vídeo)
- Diferentes sistemas operacionais oferecem diferentes chamadas de sistema (ou seja, APIs incompatíveis)
  - ▶ Com diferentes implementações
  - ▶ Com diferentes convenções de chamada



<https://wizardzines.com/comics/how-to-talk-to-your-operating-system/>

Figura 3: System calls. Imagem: Julia Evans.

- **Chamada de sistema** = função = parte da API do kernel
- Implementação de serviços do sistema operacional, como:
  - ▶ Execução de processos
  - ▶ Alocação de memória principal
  - ▶ Acesso a recursos de hardware (exemplo: teclado, rede, arquivos e disco, placa de vídeo)
- Diferentes sistemas operacionais oferecem diferentes chamadas de sistema (ou seja, APIs incompatíveis)
  - ▶ Com diferentes implementações
  - ▶ Com diferentes convenções de chamada



<https://wizardzines.com/comics/how-to-talk-to-your-operating-system/>

Figura 3: System calls. Imagem: Julia Evans.

- **Chamada de sistema** = função = parte da API do kernel
- Implementação de serviços do sistema operacional, como:
  - ▶ Execução de processos
  - ▶ Alocação de memória principal
  - ▶ Acesso a recursos de hardware (exemplo: teclado, rede, arquivos e disco, placa de vídeo)
- Diferentes sistemas operacionais oferecem diferentes chamadas de sistema (ou seja, APIs incompatíveis)
  - ▶ Com diferentes implementações
  - ▶ Com diferentes convenções de chamada



<https://wizardzines.com/comics/how-to-talk-to-your-operating-system/>

Figura 3: System calls. Imagem: Julia Evans.

# Espaço de Núcleo *versus* Espaço de Usuário

- No espaço de núcleo (*kernel space*), o SO tem controle total sobre o hardware.
- Aplicações rodando em espaço do usuário precisam invocar chamadas de sistema: requisitar ao SO para realizar alguma tarefa que requer maiores *privilégios* (e.g., receber *input* de algum hardware/aparelho ou escrever um arquivo).
- *System calls* levam a **mudanças de contexto** entre diferentes contextos de execução. (Vamos explorar esse conceito mais à frente no curso).

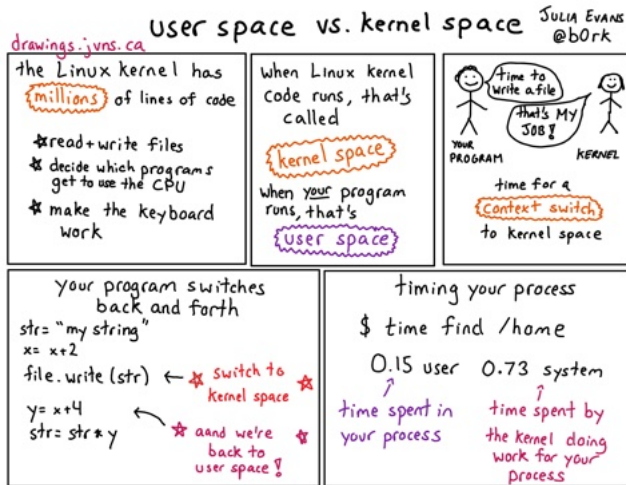


Figura 4: User space vs. kernel space. Imagem: [Julia Evans](#).

# Espaço de Núcleo *versus* Espaço de Usuário

- No espaço de núcleo (*kernel space*), o SO tem controle total sobre o hardware.
- Aplicações rodando em espaço do usuário precisam invocar chamadas de sistema: requisitar ao SO para realizar alguma tarefa que requer maiores *privilégios* (e.g., receber *input* de algum hardware/aparelho ou escrever um arquivo).
- *System calls* levam a **mudanças de contexto** entre diferentes contextos de execução. (Vamos explorar esse conceito mais à frente no curso).

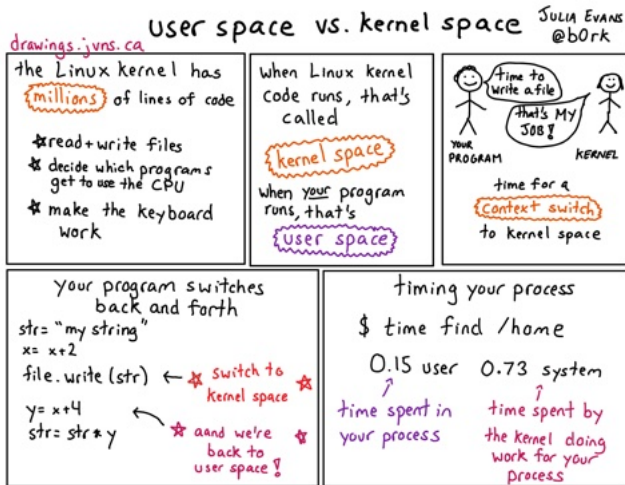


Figura 4: User space vs. kernel space. Imagem: [Julia Evans](#).

# Espaço de Núcleo *versus* Espaço de Usuário

- No espaço de núcleo (*kernel space*), o SO tem controle total sobre o hardware.
- Aplicações rodando em espaço do usuário precisam invocar chamadas de sistema: requisitar ao SO para realizar alguma tarefa que requer maiores *privilégios* (e.g., receber *input* de algum hardware/aparelho ou escrever um arquivo).
- *System calls* levam a **mudanças de contexto** entre diferentes contextos de execução. (Vamos explorar esse conceito mais à frente no curso).

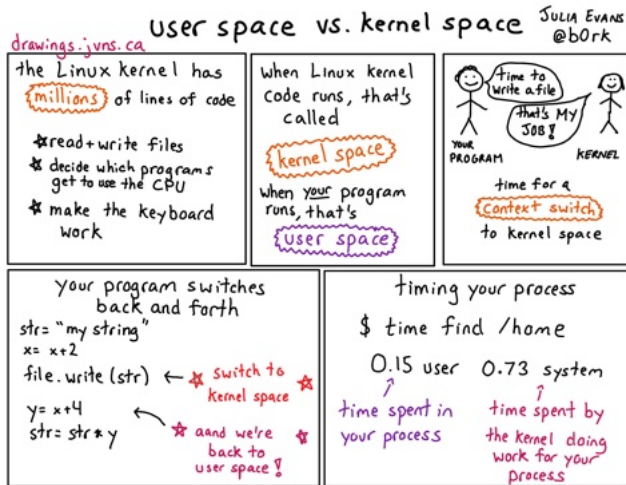
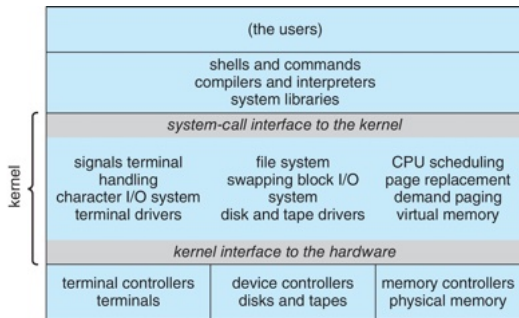


Figura 4: User space vs. kernel space. Imagem: [Julia Evans](#).



# Núcleo de SO e suas variantes

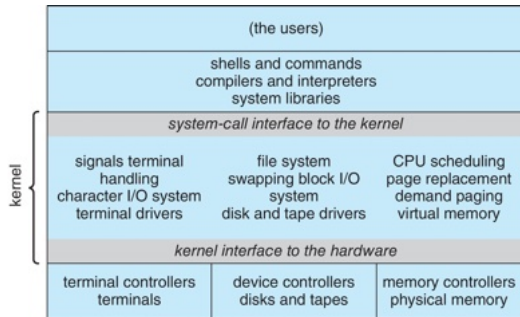
- SO roda como qualquer outro programa na CPU.
- O núcleo contém a parte mais central de um SO.
  - Código + dados do núcleo reside normalmente em memória principal.
  - As funcionalidades do núcleo rodam na CPU em *kernel mode*, reagindo a *system calls* e interrupções (tema de aula futura)
- Variantes:
  - **Monolítico**: núcleo único com todos os serviços integrados
  - **Microkernel**: núcleo mínimo, com serviços em espaço de usuário
  - **Híbrido**: mistura microkernel e monolítico para otimizar desempenho.



**Figura 5:** Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.

# Núcleo de SO e suas variantes

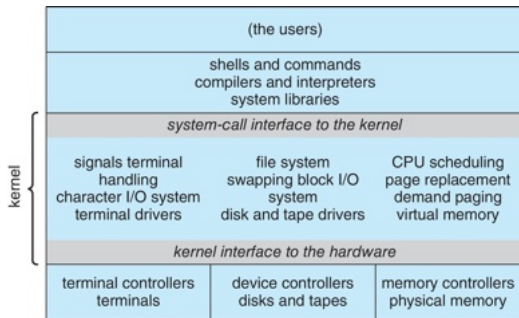
- SO roda como qualquer outro programa na CPU.
- O núcleo contém a parte mais central de um SO.
  - ▶ Código + dados do núcleo reside normalmente em memória principal.
  - ▶ As funcionalidades do núcleo rodam na CPU em *kernel mode*, reagindo a *system calls* e **interrupções** (tema de aula futura)
- Variantes:
  - ▶ **Monolítico**: núcleo único com todos os serviços integrados
  - ▶ **Microkernel**: núcleo mínimo, com serviços em espaço de usuário
  - ▶ **Híbrido**: mistura microkernel e monolítico para otimizar desempenho.



**Figura 5:** Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.

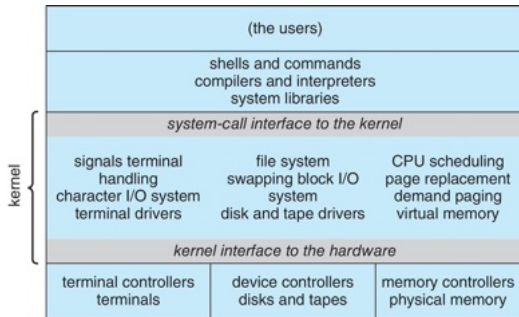
# Núcleo de SO e suas variantes

- SO roda como qualquer outro programa na CPU.
- O núcleo contém a parte mais central de um SO.
  - ▶ Código + dados do núcleo reside normalmente em memória principal.
  - ▶ As funcionalidades do núcleo rodam na CPU em *kernel mode*, reagindo a *system calls* e **interrupções** (tema de aula futura)
- Variantes:
  - ▶ **Monolítico**: núcleo único com todos os serviços integrados
  - ▶ **Microkernel**: núcleo mínimo, com serviços em espaço de usuário
  - ▶ **Híbrido**: mistura microkernel e monolítico para otimizar desempenho.



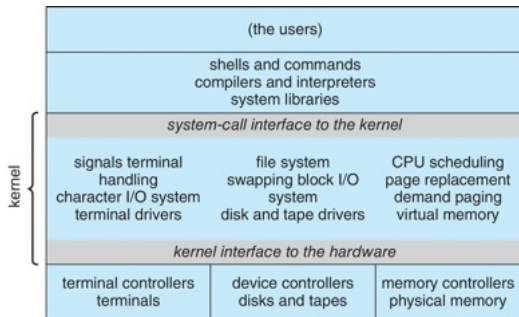
**Figura 5:** Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.

- SO roda como qualquer outro programa na CPU.
- O núcleo contém a parte mais central de um SO.
  - ▶ Código + dados do núcleo reside normalmente em memória principal.
  - ▶ As funcionalidades do núcleo rodam na CPU em *kernel mode*, reagindo a *system calls* e **interrupções** (tema de aula futura)
- Variantes:
  - ▶ **Monolítico**: núcleo único com todos os serviços integrados
  - ▶ **Microkernel**: núcleo mínimo, com serviços em espaço de usuário
  - ▶ **Híbrido**: mistura microkernel e monolítico para otimizar desempenho.



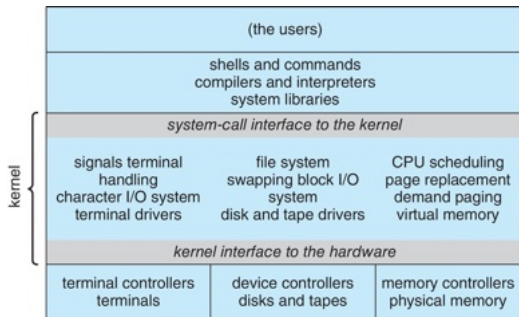
**Figura 5:** Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.

- SO roda como qualquer outro programa na CPU.
- O núcleo contém a parte mais central de um SO.
  - ▶ Código + dados do núcleo reside normalmente em memória principal.
  - ▶ As funcionalidades do núcleo rodam na CPU em *kernel mode*, reagindo a *system calls* e **interrupções** (tema de aula futura)
- Variantes:
  - ▶ **Monolítico**: núcleo único com todos os serviços integrados
  - ▶ **Microkernel**: núcleo mínimo, com serviços em espaço de usuário
  - ▶ **Híbrido**: mistura microkernel e monolítico para otimizar desempenho.



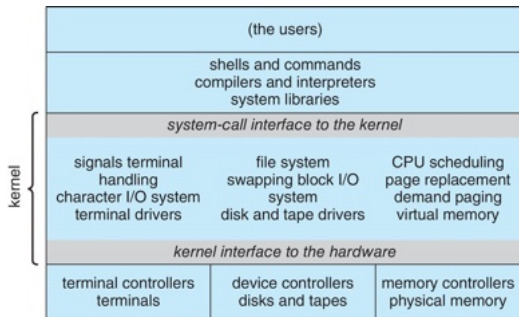
**Figura 5:** Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.

- SO roda como qualquer outro programa na CPU.
- O núcleo contém a parte mais central de um SO.
  - ▶ Código + dados do núcleo reside normalmente em memória principal.
  - ▶ As funcionalidades do núcleo rodam na CPU em *kernel mode*, reagindo a *system calls* e **interrupções** (tema de aula futura)
- Variantes:
  - ▶ **Monolítico:** núcleo único com todos os serviços integrados
  - ▶ **Microkernel:** núcleo mínimo, com serviços em espaço de usuário
  - ▶ **Híbrido:** mistura microkernel e monolítico para otimizar desempenho.



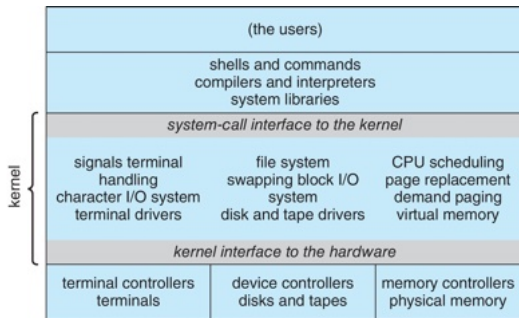
**Figura 5:** Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.

- SO roda como qualquer outro programa na CPU.
- O núcleo contém a parte mais central de um SO.
  - ▶ Código + dados do núcleo reside normalmente em memória principal.
  - ▶ As funcionalidades do núcleo rodam na CPU em *kernel mode*, reagindo a *system calls* e **interrupções** (tema de aula futura)
- Variantes:
  - ▶ **Monolítico:** núcleo único com todos os serviços integrados
  - ▶ **Microkernel:** núcleo mínimo, com serviços em espaço de usuário
  - ▶ **Híbrido:** mistura microkernel e monolítico para otimizar desempenho.



**Figura 5:** Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.

- SO roda como qualquer outro programa na CPU.
- O núcleo contém a parte mais central de um SO.
  - ▶ Código + dados do núcleo reside normalmente em memória principal.
  - ▶ As funcionalidades do núcleo rodam na CPU em *kernel mode*, reagindo a *system calls* e **interrupções** (tema de aula futura)
- Variantes:
  - ▶ **Monolítico**: núcleo único com todos os serviços integrados
  - ▶ **Microkernel**: núcleo mínimo, com serviços em espaço de usuário
  - ▶ **Híbrido**: mistura microkernel e monolítico para otimizar desempenho.



**Figura 5:** Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.



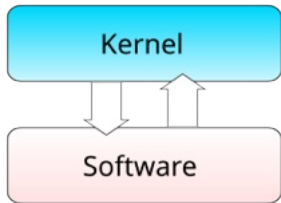


Figura 6: Núcleo monolítico.  
Imagem: [Wikipedia](#)

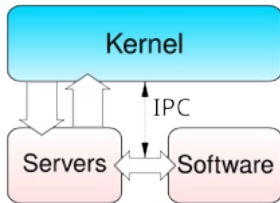


Figura 7: Micronúcleo. Imagem:  
[Wikipedia](#)

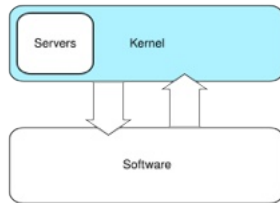


Figura 8: Núcleo híbrido. Imagem:  
[Wikipedia](#)

# Linha do tempo

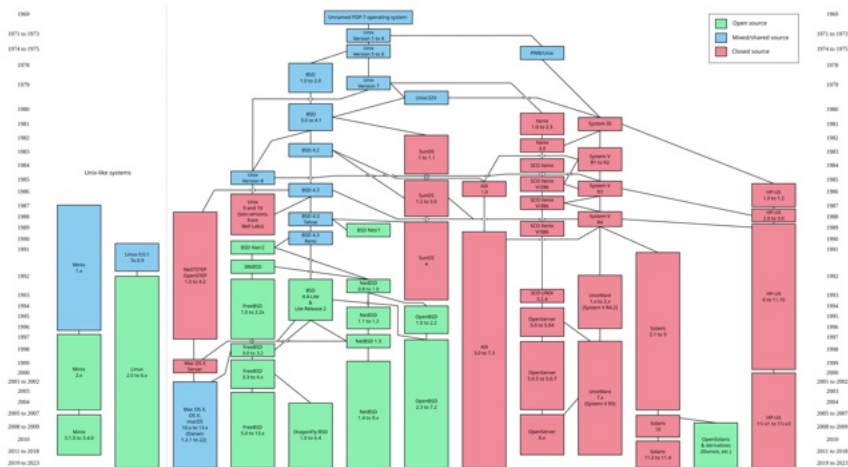


Figura 9: Linha to tempo de sistemas *Unix-like*. Imagem: [Wikipedia](#)

- 
- Linux kernel map**
- functional layers**
- user space interfaces
  - virtual
  - logical
  - device control
  - hardware interfaces
  - electronics
- human interfaces**
- HI char devices
  - HI peripherals device drivers
- system**
- interfaces core
  - Driver Model
  - system run
  - generic HW access
  - device access and bus drivers
  - user peripherals
- processing**
- processes
  - threads
  - synchronization
  - interrupts core
  - CPU specific
- memory**
- memory access
  - virtual memory
  - memory mapping
  - Page Allocator
  - physical memory operations
- storage**
- files & directories access
  - Virtual File System
  - page cache
  - swap
  - block devices
  - storage drivers
- networking**
- sockets access
  - address families
  - network storage
  - socket splice
  - protocols
  - network interfaces
  - network device drivers

16 / 30

- O código fonte do GNU/Linux tem cerca de 5 milhões de linhas.
- Já o código fonte do Windows, com seus pacotes essenciais teria cerca de 70 milhões de linhas.
- Como esse código pode ser mantido e compreendido?  
Através de abstração, modularização e organização em camadas.

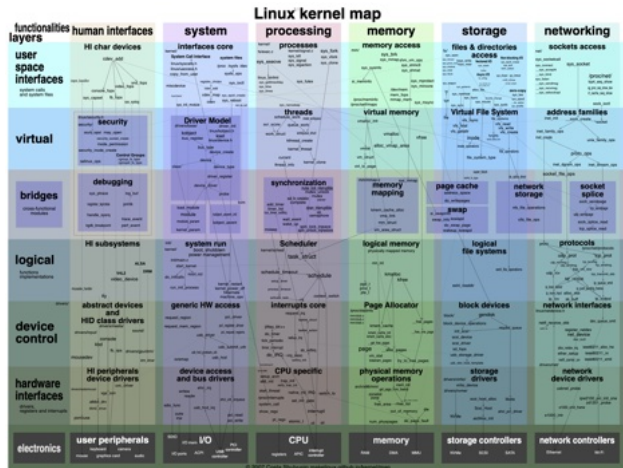


Figura 10: Mapa do núcleo do Linux. Imagem: [Costa Shulyupin](#).

- O código fonte do GNU/Linux tem cerca de 5 milhões de linhas.
- Já o código fonte do Windows, com seus pacotes essenciais teria cerca de 70 milhões de linhas.
- Como esse código pode ser mantido e compreendido?  
**Através de abstração, modularização e organização em camadas.**

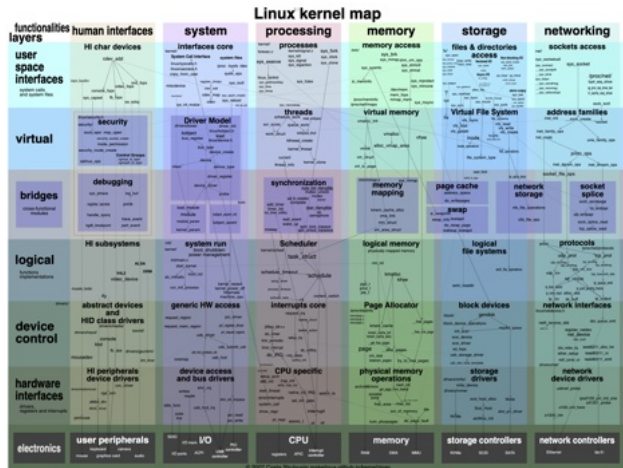




Figura 10: Mapa do núcleo do Linux. Imagem: [Costa Shulyupin](#).

- O microkernel L4 foi desenvolvido pelo alemão [Jochen Liedtke](#).
  - ▶ Hoje existe uma família de kernels baseado no L4.
  - ▶ 12 KB de código fonte (versus 918 KB do [código fonte do Linux 1.0](#), depois de comprimido).
- Em 2009, Klein et al.<sup>1</sup> apresentou uma prova formal de correção.
- Software formalmente verificado **não precisa de patches** para correção de bugs, pois não há bugs.
  - ▶ É essencial lembrar que softwares formalmente verificados existem e já atingem a complexidade de microkernels.
  - ▶ Falhas e bugs em sistemas críticos são cada vez menos aceitáveis.
- [Processadores Apple](#) (e.g., A7) incluem uma variante do L4.
- Veja também o vídeo [Microkernel operating systems: what they are and why they're so important today](#) do canal *Kaspersky* no YouTube .

---

<sup>1</sup>Klein, Gerwin, et al. "seL4: Formal verification of an OS kernel." Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009. <https://dl.acm.org/doi/10.1145/1629575.1629596>.

- O microkernel L4 foi desenvolvido pelo alemão [Jochen Liedtke](#).
  - ▶ Hoje existe uma família de kernels baseado no L4.
  - ▶ 12 KB de código fonte (versus 918 KB do [código fonte do Linux 1.0](#), depois de comprimido).
- Em 2009, Klein et al.<sup>1</sup> apresentou uma prova formal de correção.
- Software formalmente verificado **não precisa de patches** para correção de bugs, pois não há bugs.
  - ▶ É essencial lembrar que softwares formalmente verificados existem e já atingem a complexidade de microkernels.
  - ▶ Falhas e bugs em sistemas críticos são cada vez menos aceitáveis.
- [Processadores Apple](#) (e.g., A7) incluem uma variante do L4.
- Veja também o vídeo [Microkernel operating systems: what they are and why they're so important today](#) do canal *Kaspersky* no YouTube .

---


<sup>1</sup>Klein, Gerwin, et al. "seL4: Formal verification of an OS kernel." Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009. <https://dl.acm.org/doi/10.1145/1629575.1629596>.

- O microkernel L4 foi desenvolvido pelo alemão Jochen Liedtke.
  - ▶ Hoje existe uma família de kernels baseado no L4.
  - ▶ 12 KB de código fonte (versus 918 KB do código fonte do Linux 1.0, depois de comprimido).
- Em 2009, Klein et al.<sup>1</sup> apresentou uma prova formal de correção.
- Software formalmente verificado **não precisa de patches** para correção de bugs, pois não há bugs.
  - ▶ É essencial lembrar que softwares formalmente verificados existem e já atingem a complexidade de microkernels.
  - ▶ Falhas e bugs em sistemas críticos são cada vez menos aceitáveis.
- Processadores Apple (e.g., A7) incluem uma variante do L4.
- Veja também o vídeo Microkernel operating systems: what they are and why they're so important today do canal Kaspersky no YouTube ▶.

---


<sup>1</sup>Klein, Gerwin, et al. "seL4: Formal verification of an OS kernel." Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009. <https://dl.acm.org/doi/10.1145/1629575.1629596>.





- O microkernel L4 foi desenvolvido pelo alemão [Jochen Liedtke](#).
  - ▶ Hoje existe uma família de kernels baseado no L4.
  - ▶ 12 KB de código fonte (versus 918 KB do [código fonte do Linux 1.0](#), depois de comprimido).
- Em 2009, Klein et al.<sup>1</sup> apresentou uma prova formal de correção.
- Software formalmente verificado **não precisa de patches** para correção de bugs, pois não há bugs.
  - ▶ É essencial lembrar que softwares formalmente verificados existem e já atingem a complexidade de microkernels.
  - ▶ Falhas e bugs em sistemas críticos são cada vez menos aceitáveis.
- [Processadores Apple](#) (e.g., A7) incluem uma variante do L4.
- Veja também o vídeo [Microkernel operating systems: what they are and why they're so important today](#) do canal *Kaspersky* no YouTube .


---


<sup>1</sup>Klein, Gerwin, et al. "seL4: Formal verification of an OS kernel." Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009. <https://dl.acm.org/doi/10.1145/1629575.1629596>.

- Quando o sistema é ligado, a execução começa em um endereço de memória fixo.
- Um pequeno trecho de código, chamado **bootstrap loader** ou **BIOS**, armazenado em ROM ou EEPROM, localiza o kernel, carrega-o na memória e inicia sua execução.
- Algumas vezes, esse processo ocorre em duas etapas:
  - ▶ Um **bloco de boot** localizado em um endereço fixo é carregado pelo código da ROM.
  - ▶ Esse bloco carrega o **bootstrap loader** a partir do disco.
- Sistemas modernos substituem a **BIOS** pela **Unified Extensible Firmware Interface (UEFI)**.
  - ▶ **Vários usuários** reportam problemas com **dual boot** em sistemas UEFI.
- Um **bootstrap loader** comum é o **GRUB**, que permite, e.g., configurar opções para o kernel.
- O kernel é carregado e o sistema operacional entra em execução.
- Veja também o vídeo [How Does Linux Boot Process Work?](#) do canal *ByteByteGo* no YouTube .

- Quando o sistema é ligado, a execução começa em um endereço de memória fixo.
- Um pequeno trecho de código, chamado **bootstrap loader** ou **BIOS**, armazenado em ROM ou EEPROM, localiza o kernel, carrega-o na memória e inicia sua execução.
- Algumas vezes, esse processo ocorre em duas etapas:
  - ▶ Um **bloco de boot** localizado em um endereço fixo é carregado pelo código da ROM.
  - ▶ Esse bloco carrega o **bootstrap loader** a partir do disco.
- Sistemas modernos substituem a **BIOS** pela **Unified Extensible Firmware Interface (UEFI)**.
  - ▶ **Vários usuários** reportam problemas com **dual boot** em sistemas UEFI.
- Um **bootstrap loader** comum é o **GRUB**, que permite, e.g., configurar opções para o kernel.
- O kernel é carregado e o sistema operacional entra em execução.
- Veja também o vídeo [How Does Linux Boot Process Work?](#) do canal *ByteByteGo* no YouTube .

- Quando o sistema é ligado, a execução começa em um endereço de memória fixo.
- Um pequeno trecho de código, chamado **bootstrap loader** ou **BIOS**, armazenado em ROM ou EEPROM, localiza o kernel, carrega-o na memória e inicia sua execução.
- Algumas vezes, esse processo ocorre em duas etapas:
  - ▶ Um **bloco de boot** localizado em um endereço fixo é carregado pelo código da ROM.
  - ▶ Esse bloco carrega o **bootstrap loader** a partir do disco.
- Sistemas modernos substituem a **BIOS** pela **Unified Extensible Firmware Interface (UEFI)**.
  - ▶ **Vários usuários** reportam problemas com **dual boot** em sistemas UEFI.
- Um **bootstrap loader** comum é o **GRUB**, que permite, e.g., configurar opções para o kernel.
- O kernel é carregado e o sistema operacional entra em execução.
- Veja também o vídeo [How Does Linux Boot Process Work?](#) do canal *ByteByteGo* no YouTube .

- Quando o sistema é ligado, a execução começa em um endereço de memória fixo.
- Um pequeno trecho de código, chamado **bootstrap loader** ou **BIOS**, armazenado em ROM ou EEPROM, localiza o kernel, carrega-o na memória e inicia sua execução.
- Algumas vezes, esse processo ocorre em duas etapas:
  - ▶ Um **bloco de boot** localizado em um endereço fixo é carregado pelo código da ROM.
  - ▶ Esse bloco carrega o **bootstrap loader** a partir do disco.
- Sistemas modernos substituem a **BIOS** pela **Unified Extensible Firmware Interface (UEFI)**.
  - ▶ **Vários usuários** reportam problemas com **dual boot** em sistemas UEFI.
- Um **bootstrap loader** comum é o **GRUB**, que permite, e.g., configurar opções para o kernel.
- O kernel é carregado e o sistema operacional entra em execução.
- Veja também o vídeo [How Does Linux Boot Process Work?](#) do canal *ByteByteGo* no YouTube .

- Quando o sistema é ligado, a execução começa em um endereço de memória fixo.
- Um pequeno trecho de código, chamado **bootstrap loader** ou **BIOS**, armazenado em ROM ou EEPROM, localiza o kernel, carrega-o na memória e inicia sua execução.
- Algumas vezes, esse processo ocorre em duas etapas:
  - ▶ Um **bloco de boot** localizado em um endereço fixo é carregado pelo código da ROM.
  - ▶ Esse bloco carrega o **bootstrap loader** a partir do disco.
- Sistemas modernos substituem a **BIOS** pela **Unified Extensible Firmware Interface (UEFI)**.
  - ▶ [Vários usuários](#) reportam problemas com **dual boot** em sistemas UEFI.
- Um **bootstrap loader** comum é o [GRUB](#), que permite, e.g., configurar opções para o kernel.
- O kernel é carregado e o sistema operacional entra em execução.
- Veja também o vídeo [How Does Linux Boot Process Work?](#) do canal *ByteByteGo* no YouTube .

# Multitarefa

- Os sistemas operacionais permitem que múltiplas computações ocorram **concorrentemente** em um único sistema computacional.
  - ▶ Um exemplo de **multitarefa** ocorre quando você está **ouvindo música no Spotify** enquanto **navega na internet e clica em links**.
  - ▶ O **sistema operacional** gerencia a execução simultânea do player de música e do navegador.
- Para isso, o sistema:
  - ▶ Divide o tempo (*time slicing*) do hardware entre os diferentes operações em execução (via **Escalonamento**).
  - ▶ Gerencia as transições entre as operações.
  - ▶ Mantém o controle do estado de cada operação para que possam ser retomados corretamente.



- Mesmo em um único CPU core: ilusão de simultaneidade (computação "paralela").
- Essa capacidade é essencial para:
  - ▶ Garantir **eficiência** no uso dos recursos computacionais.
  - ▶ Proporcionar **responsividade**, permitindo que múltiplos programas rodem de forma contínua e sem atrasos perceptíveis.
  - ▶ Melhorar a **utilização do sistema**, possibilitando a execução simultânea de várias tarefas.

## ● Conceitos de Processos e Threads

- ▶ Diferentes termos podem ser usados para se referir a execuções ("computações") em um computador: *threads*, processos, tarefas ou trabalhos.
- ▶ Neste contexto, os termos **thread** e **processo** possuem significados distintos.

## ● Thread

- ▶ Unidade fundamental de concorrência.
- ▶ Representa uma sequência de ações programadas (executadas em um CPU core).
- ▶ Um programa pode criar múltiplas threads para executar diferentes tarefas simultaneamente.
- ▶ Mesmo que um programa crie apenas uma thread, um sistema típico executa diversas threads simultaneamente, incluindo aquelas do sistema operacional.

## ● Processo

- ▶ Criado sempre que um programa é iniciado.
- ▶ Atua como um contêiner que gerencia e protege uma ou mais threads.
- ▶ Impede que threads de processos diferentes interfiram umas nas outras.
- ▶ Exemplo: uma thread em um processo não pode sobrescrever a memória de outro processo.

## ● Conceitos de Processos e Threads

- ▶ Diferentes termos podem ser usados para se referir a execuções ("computações") em um computador: *threads*, processos, tarefas ou trabalhos.
- ▶ Neste contexto, os termos **thread** e **processo** possuem significados distintos.

## ● Thread

- ▶ Unidade fundamental de concorrência.
- ▶ Representa uma sequência de ações programadas (executadas em um CPU core).
- ▶ Um programa pode criar múltiplas threads para executar diferentes tarefas simultaneamente.
- ▶ Mesmo que um programa crie apenas uma thread, um sistema típico executa diversas threads simultaneamente, incluindo aquelas do sistema operacional.

## ● Processo

- ▶ Criado sempre que um programa é iniciado.
- ▶ Atua como um contêiner que gerencia e protege uma ou mais threads.
- ▶ Impede que threads de processos diferentes interfiram umas nas outras.
- ▶ Exemplo: uma thread em um processo não pode sobrescrever a memória de outro processo.

## ● Conceitos de Processos e Threads

- ▶ Diferentes termos podem ser usados para se referir a execuções ("computações") em um computador: *threads*, processos, tarefas ou trabalhos.
- ▶ Neste contexto, os termos **thread** e **processo** possuem significados distintos.

## ● Thread

- ▶ Unidade fundamental de concorrência.
- ▶ Representa uma sequência de ações programadas (executadas em um CPU core).
- ▶ Um programa pode criar múltiplas threads para executar diferentes tarefas simultaneamente.
- ▶ Mesmo que um programa crie apenas uma thread, um sistema típico executa diversas threads simultaneamente, incluindo aquelas do sistema operacional.

## ● Processo

- ▶ Criado sempre que um programa é iniciado.
- ▶ Atua como um contêiner que gerencia e protege uma ou mais threads.
- ▶ Impede que threads de processos diferentes interfiram umas nas outras.
- ▶ Exemplo: uma thread em um processo não pode sobrescrever a memória de outro processo.

## ● Conceitos de Processos e Threads

- ▶ Diferentes termos podem ser usados para se referir a execuções ("computações") em um computador: *threads*, processos, tarefas ou trabalhos.
- ▶ Neste contexto, os termos **thread** e **processo** possuem significados distintos.

## ● Thread

- ▶ Unidade fundamental de concorrência.
- ▶ Representa uma sequência de ações programadas (executadas em um CPU core).
- ▶ Um programa pode criar múltiplas threads para executar diferentes tarefas simultaneamente.
- ▶ Mesmo que um programa crie apenas uma thread, um sistema típico executa diversas threads simultaneamente, incluindo aquelas do sistema operacional.

## ● Processo

- ▶ Criado sempre que um programa é iniciado.
- ▶ Atua como um contêiner que gerencia e protege uma ou mais threads.
- ▶ Impede que threads de processos diferentes interfiram umas nas outras.
- ▶ Exemplo: uma thread em um processo não pode sobrescrever a memória de outro processo.

## ● Conceitos de Processos e Threads

- ▶ Diferentes termos podem ser usados para se referir a execuções ("computações") em um computador: *threads*, processos, tarefas ou trabalhos.
- ▶ Neste contexto, os termos **thread** e **processo** possuem significados distintos.

## ● Thread

- ▶ Unidade fundamental de concorrência.
- ▶ Representa uma sequência de ações programadas (executadas em um CPU core).
- ▶ Um programa pode criar múltiplas threads para executar diferentes tarefas simultaneamente.
- ▶ Mesmo que um programa crie apenas uma thread, um sistema típico executa diversas threads simultaneamente, incluindo aquelas do sistema operacional.

## ● Processo

- ▶ Criado sempre que um programa é iniciado.
- ▶ Atua como um contêiner que gerencia e protege uma ou mais threads.
- ▶ Impede que threads de processos diferentes interfiram umas nas outras.
- ▶ Exemplo: uma thread em um processo não pode sobrescrever a memória de outro processo.

## ● Conceitos de Processos e Threads

- ▶ Diferentes termos podem ser usados para se referir a execuções ("computações") em um computador: *threads*, processos, tarefas ou trabalhos.
- ▶ Neste contexto, os termos **thread** e **processo** possuem significados distintos.

## ● Thread

- ▶ Unidade fundamental de concorrência.
- ▶ Representa uma sequência de ações programadas (executadas em um CPU core).
- ▶ Um programa pode criar múltiplas threads para executar diferentes tarefas simultaneamente.
- ▶ Mesmo que um programa crie apenas uma thread, um sistema típico executa diversas threads simultaneamente, incluindo aquelas do sistema operacional.

## ● Processo

- ▶ Criado sempre que um programa é iniciado.
- ▶ Atua como um contêiner que gerencia e protege uma ou mais threads.
- ▶ Impede que threads de processos diferentes interfiram umas nas outras.
- ▶ Exemplo: uma thread em um processo não pode sobrescrever a memória de outro processo.

## ● Conceitos de Processos e Threads

- ▶ Diferentes termos podem ser usados para se referir a execuções ("computações") em um computador: *threads*, processos, tarefas ou trabalhos.
- ▶ Neste contexto, os termos **thread** e **processo** possuem significados distintos.

## ● Thread

- ▶ Unidade fundamental de concorrência.
- ▶ Representa uma sequência de ações programadas (executadas em um CPU core).
- ▶ Um programa pode criar múltiplas threads para executar diferentes tarefas simultaneamente.
- ▶ Mesmo que um programa crie apenas uma thread, um sistema típico executa diversas threads simultaneamente, incluindo aquelas do sistema operacional.

## ● Processo

- ▶ Criado sempre que um programa é iniciado.
- ▶ Atua como um contêiner que gerencia e protege uma ou mais threads.
- ▶ Impede que threads de processos diferentes interfiram umas nas outras.
- ▶ Exemplo: uma thread em um processo não pode sobrescrever a memória de outro processo.



## ● Conceitos de Processos e Threads

- ▶ Diferentes termos podem ser usados para se referir a execuções ("computações") em um computador: *threads*, processos, tarefas ou trabalhos.
- ▶ Neste contexto, os termos **thread** e **processo** possuem significados distintos.

## ● Thread

- ▶ Unidade fundamental de concorrência.
- ▶ Representa uma sequência de ações programadas (executadas em um CPU core).
- ▶ Um programa pode criar múltiplas threads para executar diferentes tarefas simultaneamente.
- ▶ Mesmo que um programa crie apenas uma thread, um sistema típico executa diversas threads simultaneamente, incluindo aquelas do sistema operacional.

## ● Processo

- ▶ Criado sempre que um programa é iniciado.
- ▶ Atua como um contêiner que gerencia e protege uma ou mais threads.
- ▶ Impede que threads de processos diferentes interfiram umas nas outras.
- ▶ Exemplo: uma thread em um processo não pode sobrescrever a memória de outro processo.

## ● Conceitos de Processos e Threads

- ▶ Diferentes termos podem ser usados para se referir a execuções ("computações") em um computador: *threads*, processos, tarefas ou trabalhos.
- ▶ Neste contexto, os termos **thread** e **processo** possuem significados distintos.

## ● Thread

- ▶ Unidade fundamental de concorrência.
- ▶ Representa uma sequência de ações programadas (executadas em um CPU core).
- ▶ Um programa pode criar múltiplas threads para executar diferentes tarefas simultaneamente.
- ▶ Mesmo que um programa crie apenas uma thread, um sistema típico executa diversas threads simultaneamente, incluindo aquelas do sistema operacional.

## ● Processo

- ▶ Criado sempre que um programa é iniciado.
- ▶ Atua como um contêiner que gerencia e protege uma ou mais threads.
- ▶ Impede que threads de processos diferentes interfiram umas nas outras.
- ▶ Exemplo: uma thread em um processo não pode sobrescrever a memória de outro processo.

## ● Conceitos de Processos e Threads

- ▶ Diferentes termos podem ser usados para se referir a execuções ("computações") em um computador: *threads*, processos, tarefas ou trabalhos.
- ▶ Neste contexto, os termos **thread** e **processo** possuem significados distintos.

## ● Thread

- ▶ Unidade fundamental de concorrência.
- ▶ Representa uma sequência de ações programadas (executadas em um CPU core).
- ▶ Um programa pode criar múltiplas threads para executar diferentes tarefas simultaneamente.
- ▶ Mesmo que um programa crie apenas uma thread, um sistema típico executa diversas threads simultaneamente, incluindo aquelas do sistema operacional.

## ● Processo

- ▶ Criado sempre que um programa é iniciado.
- ▶ Atua como um contêiner que gerencia e protege uma ou mais threads.
- ▶ Impede que threads de processos diferentes interfiram umas nas outras.
- ▶ Exemplo: uma thread em um processo não pode sobrescrever a memória de outro processo.

- Cada thread executa seu próprio código.
- A execução de múltiplas threads torna-se mais complexa quando elas precisam interagir entre si.
- **Problema da Sincronização**
  - ▶ Uma thread pode produzir dados enquanto outra os consome.
  - ▶ Se uma thread escreve dados na memória e outra os lê, é essencial garantir a sincronização correta.

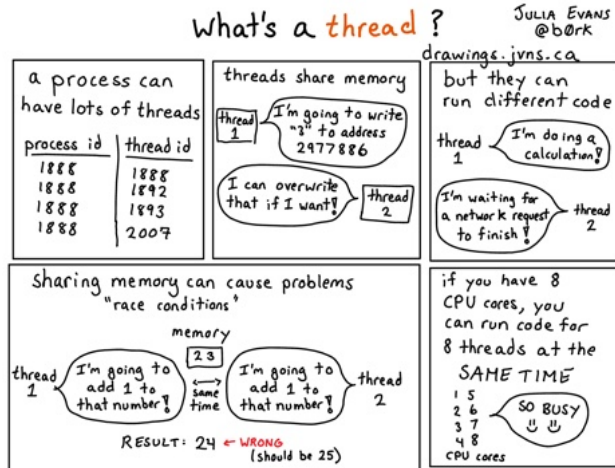


Figura 11: O que é uma thread? Créditos: [Julia Evans](#).

- Cada thread executa seu próprio código.
- A execução de múltiplas threads torna-se mais complexa quando elas precisam interagir entre si.
- **Problema da Sincronização**
  - ▶ Uma thread pode produzir dados enquanto outra os consome.
  - ▶ Se uma thread escreve dados na memória e outra os lê, é essencial garantir a sincronização correta.

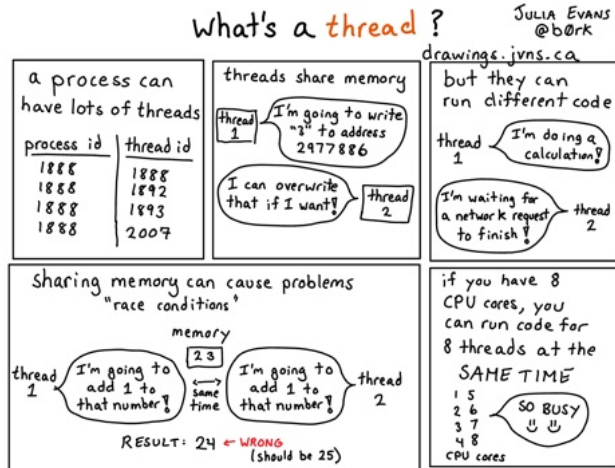


Figura 11: O que é uma thread? Créditos: [Julia Evans](#).

- Um programa pode criar múltiplos processos, e.g., um processo por tab aberta no navegador.
  - No Firefox, digite `about:processes` na barra de endereço.
- Um processo no sistema operacional contém diversas informações essenciais para sua execução.
  - PID:** Identificador único do processo.
  - Diretório de Trabalho:** O diretório no qual o processo está sendo executado.
  - Memória:** Área alocada para o processo, incluindo pilha e heap.



Figura 12: O que há em um processo? Créditos: [Julia Evans](#).

# *Debug* Seu Conhecimento

## Questão 1: Quais afirmações são corretas sobre conceitos de Sistemas Operacionais?

- a) O sistema operacional gerencia a execução de aplicações em termos de **threads**.
- b) O sistema operacional cria uma nova thread para cada chamada de sistema (*system call*).
- c) O sistema operacional agenda (*schedule*) threads para execução nos núcleos da CPU.
- d) O sistema operacional cria novas threads para utilizar todos os núcleos da CPU.
- e) O **time-slicing** cria a ilusão de paralelismo em núcleos de CPU únicos.
- f) O sistema operacional bloqueia recursos compartilhados para evitar anomalias de atualização.



Questão 1: Quais afirmações são corretas sobre conceitos de Sistemas Operacionais?

- a) O sistema operacional gerencia a execução de aplicações em termos de **threads**.
- b) O sistema operacional cria uma nova thread para cada chamada de sistema (*system call*).
- c) O sistema operacional agenda (*schedule*) threads para execução nos núcleos da CPU.
- d) O sistema operacional cria novas threads para utilizar todos os núcleos da CPU.
- e) O **time-slicing** cria a ilusão de paralelismo em núcleos de CPU únicos.
- f) O sistema operacional bloqueia recursos compartilhados para evitar anomalias de atualização.

**Resposta:** São verdadeiras: (a), (c), (e).

Questão 2: Qual das alternativas melhor descreve a diferença entre o espaço de usuário e o espaço de kernel?

- (a) O espaço de usuário é utilizado apenas por processos do sistema operacional, enquanto o espaço de kernel é usado exclusivamente por aplicativos do usuário.
- (b) No espaço de kernel, os processos executam com privilégios elevados e podem acessar diretamente o hardware, enquanto no espaço de usuário os processos possuem restrições e acessam recursos do sistema por meio de chamadas ao kernel.
- (c) O espaço de usuário contém apenas arquivos de configuração do sistema operacional, enquanto o espaço de kernel armazena aplicativos e bibliotecas do usuário.
- (d) No espaço de usuário, os processos podem acessar diretamente os dispositivos de hardware, enquanto no espaço de kernel as operações são sempre intermediadas por um gerenciador de dispositivos.

Questão 2: Qual das alternativas melhor descreve a diferença entre o espaço de usuário e o espaço de kernel?

- (a) O espaço de usuário é utilizado apenas por processos do sistema operacional, enquanto o espaço de kernel é usado exclusivamente por aplicativos do usuário.
- (b) No espaço de kernel, os processos executam com privilégios elevados e podem acessar diretamente o hardware, enquanto no espaço de usuário os processos possuem restrições e acessam recursos do sistema por meio de chamadas ao kernel.
- (c) O espaço de usuário contém apenas arquivos de configuração do sistema operacional, enquanto o espaço de kernel armazena aplicativos e bibliotecas do usuário.
- (d) No espaço de usuário, os processos podem acessar diretamente os dispositivos de hardware, enquanto no espaço de kernel as operações são sempre intermediadas por um gerenciador de dispositivos.

**Resposta:** (b).

# Fechamento e Perspectivas

- Definição e Função

- ▶ O SO atua como intermediário entre o hardware e os programas do usuário.
- ▶ Garante a execução eficiente e segura de múltiplos processos.

- Perspectiva:

- ▶ Diferentes arquiteturas e modelos influenciam o desempenho e a segurança.
- ▶ O conhecimento sobre SO é essencial para otimizar o desenvolvimento de software.

- **Próximos Passos**

- ▶ Ler as seções 1.1 (O que é um sistema operacional?) e 1.2 (História dos sistemas operacionais) do livro *TANENBAUM, A.; Sistemas Operacionais Modernos. 4a ed. Pearson Brasil, 2015.*
- ▶ Próxima aula: Explorar gerenciamento de processos e threads em detalhes.

# Dúvidas e Discussão

Prof. Dr. Denis M. L. Martins

[denis.mayr@puc-campinas.edu.br](mailto:denis.mayr@puc-campinas.edu.br)