

# Revisão SO: Parte 1

## Implementação de Núcleo de Sistema Operacional

Prof. Dr. Denis M. L. Martins

Engenharia de Computação: 5º Semestre

# 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](#).

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.

## Software que:

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

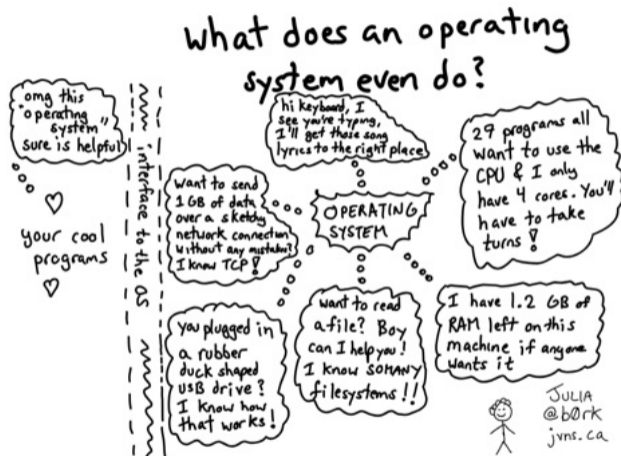


Figura 1: O que um SO faz. Créditos: [Julia Evans](#).

## Software que:

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

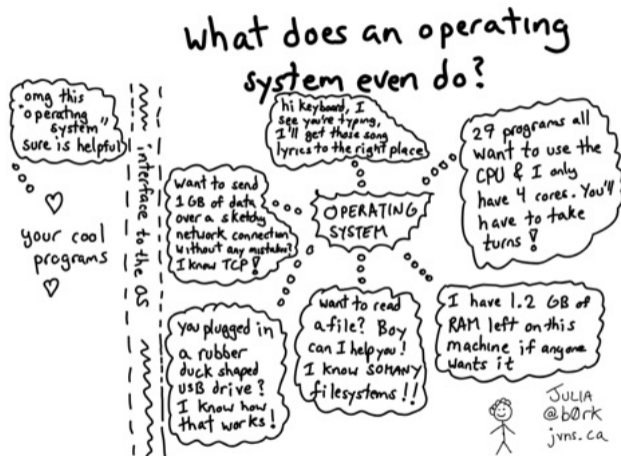



Figura 1: O que um SO faz. Créditos: [Julia Evans](#).

# Kernel/Núcleo de um SO



# Como falar com o SO

- O núcleo (*kernel*) de um SO oferece uma API (*Application Programming Interface*)
  - ▶ Expõe um conjunto de interfaces para os serviços do OS. **Chamada de sistema** (system calls) = função.
  - ▶ Veja também o vídeo [What is a Kernel?](#) do canal *Techquickie* no YouTube .
- Implementação de serviços do sistema operacional, como:
  - ▶ Execução de processos
  - ▶ Alocação de memória principal
  - ▶ Acesso a recursos de hardware (exemplo: teclado, arquivos e disco)
- Diferentes sistemas operacionais oferecem diferentes chamadas de sistema (ou seja, APIs incompatíveis)



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

Figura 2: 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).

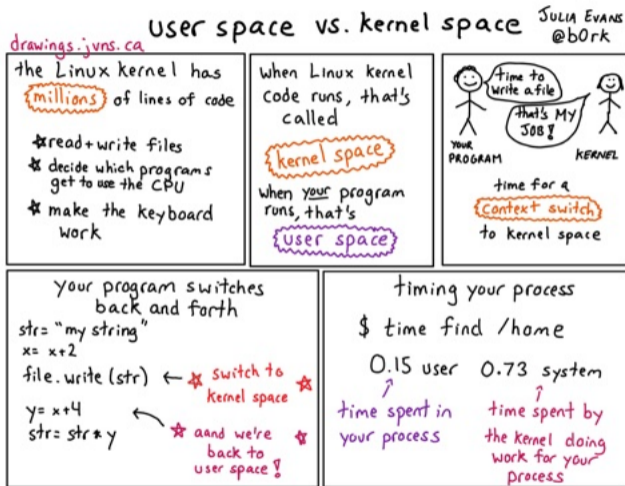


Figura 3: User space vs. kernel space. Imagem: [Julia Evans](#).

# 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).

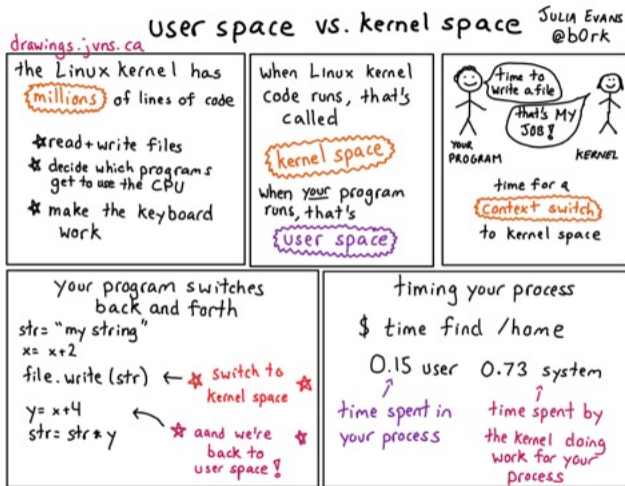


Figura 3: User space vs. kernel space. Imagem: [Julia Evans](#).

# 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).

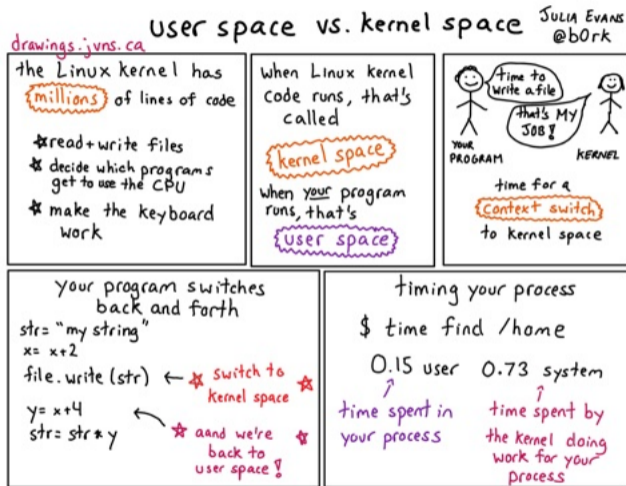


Figura 3: User space vs. kernel space. Imagem: [Julia Evans](#).

- 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.

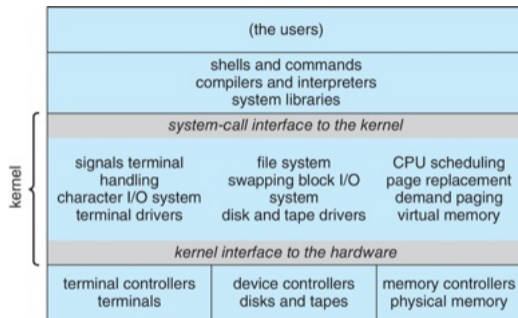


Figura 4: Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.

- 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.

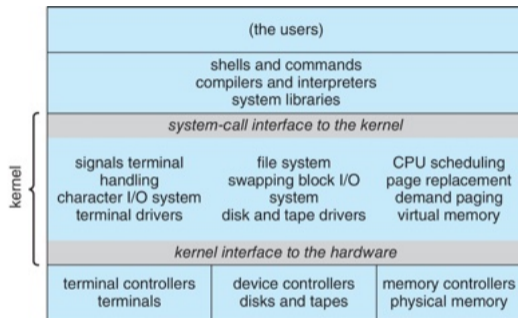
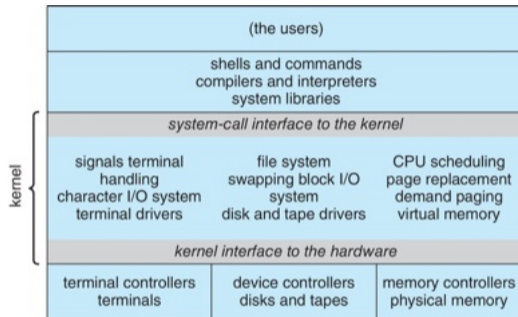


Figura 4: Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.

- 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 4:** 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.

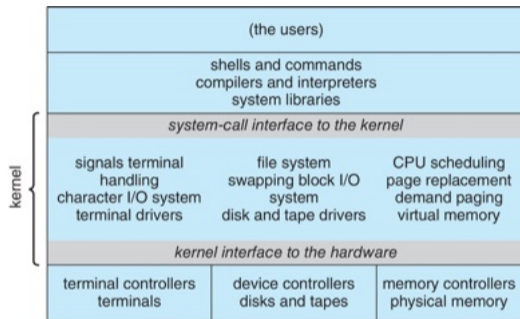


Figura 4: Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.



- 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.

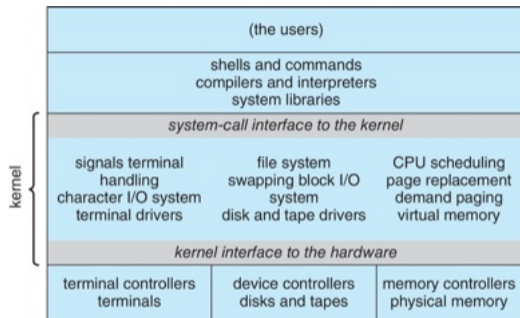


Figura 4: Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.

- 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.

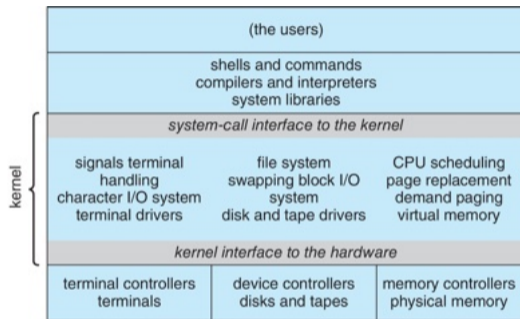


Figura 4: Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.

- 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.

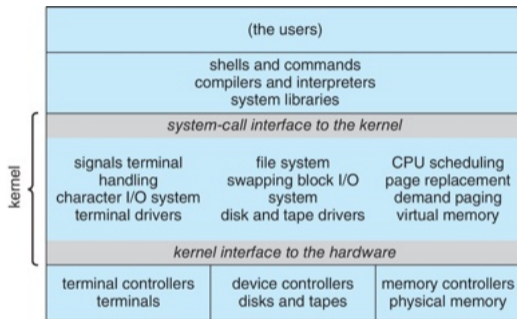


Figura 4: Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.

- 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.

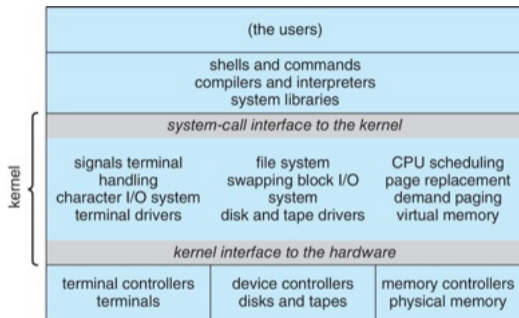


Figura 4: Estrutura tradicional de um sistema UNIX. Imagem: Figura 2.12 de Silberschatz et al. Fundamentos de Sistemas Operacionais.

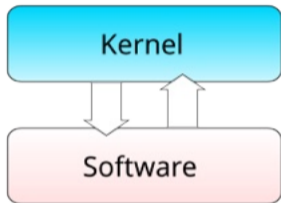


Figura 5: Núcleo monolítico.  
Imagem: [Wikipedia](#)

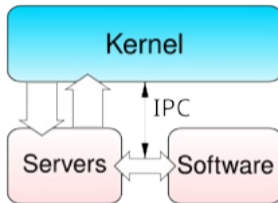


Figura 6: Micronúcleo. Imagem:  
[Wikipedia](#)

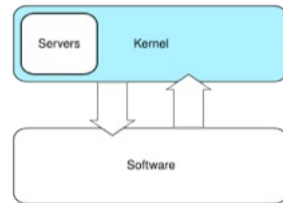




Figura 7: Núcleo híbrido. Imagem:  
[Wikipedia](#)

- 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>.


# Boot e o Kernel de um SO

- 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.
- **Paralelismo versus Concorrência** .

- 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.

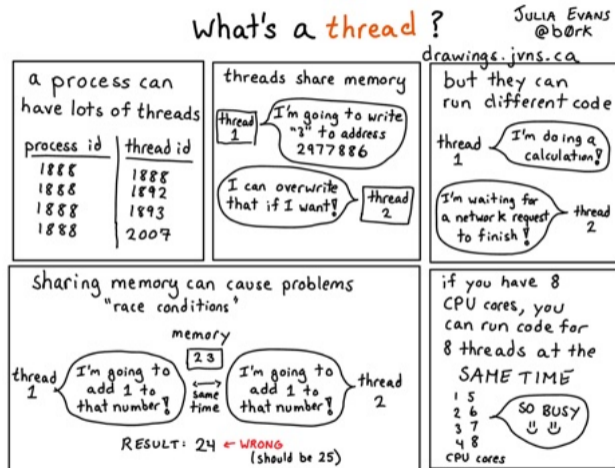


Figura 8: O que é uma thread? Créditos: Julia Evans.

- 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.

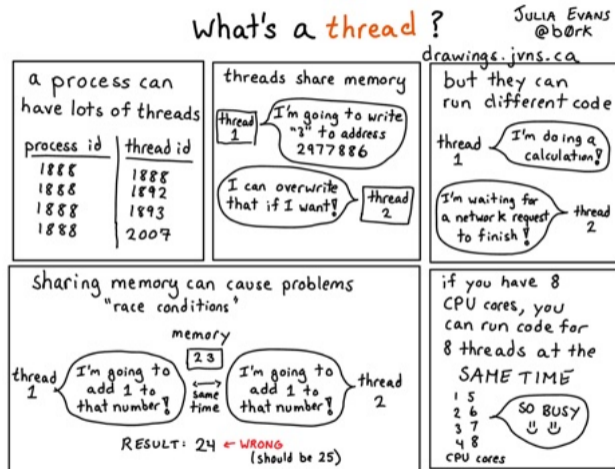


Figura 8: O que é uma thread? Créditos: Julia Evans.

- Processo  $\approx$  programa em execução.
  - ▶ **Programa:** entidade **passiva** guardada no disco (**arquivo executável**).
- Um programa pode criar múltiplos processos, e.g., um processo por tab aberta no navegador.
  - ▶ No Firefox, digite `about:processes` na barra de endereço.
- Processo como unidade de **gerenciamento e proteção**.
- O que há em um processo?
  - ▶ **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.
  - ▶ ... (ver próximo slide)



Figura 9: O que há em um processo?  
Créditos: [Julia Evans](#).

- **Estado do processo:** em execução, em espera, etc.
- **Contador de programa (PC):** endereço da próxima instrução.
- **Registadores da CPU:** conteúdo de todos os registradores utilizados por processos.
- **Informações de escalonamento:** prioridades, ponteiros para filas.
- **Informação de gerenciamento de memória:** memória alocada para o processo.
- **Estatísticas:** uso de CPU, tempo desde o início, limites de tempo.
- **Informações de I/O:** dispositivos alocados ao processo, lista de arquivos abertos.

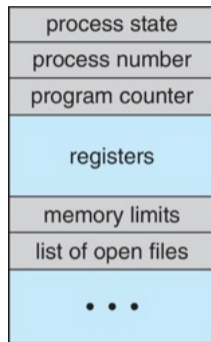


Figura 10: Bloco de Controle de Processo, do inglês *Process Control Block*. Créditos: Silberschatz, Galvin and Gagne, 2018.



# Layout de Memória de um programa em C

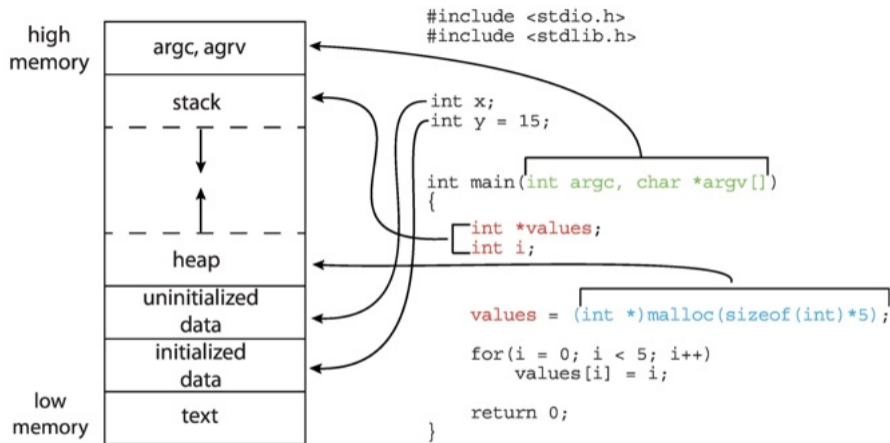



Figura 11: Layout de um programa escrito em Linguagem C na memória.

Créditos: Silberschatz, Galvin and Gagne, 2018.

Um processo na memória respeita o seguinte layout :

- Código do programa: **text section**.
- Atividade atual: **program counter** (PC), registradores.
- **Stack** (pilha) contendo dados temporários.
- Parâmetros de funções, endereços de retorno, variáveis locais.
- **Data section** contendo variáveis globais.
- **Heap** contendo memória alocada dinamicamente.

# Estado de um Processo

Conforme um processo executa, ele muda de estado:<sup>a</sup>

- **Novo:** O processo está sendo criado.
- **Em execução:** Instruções estão sendo executadas.
- **Em espera:** O processo está esperando por um evento.
- **Pronto:** O processo está esperando que seja atribuído a um processador.
- **Concluído:** O processo terminou sua execução.

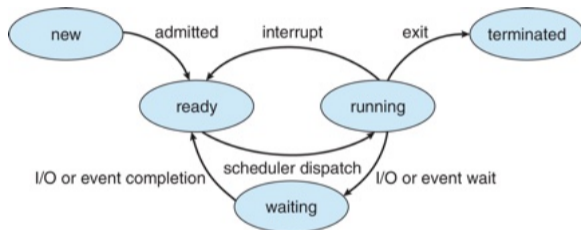



Figura 12: Diagrama de estados de um processo. Créditos: Silberschatz, Galvin and Gagne, 2018.

<sup>a</sup>No livro de Max Hailperin (2019), há uma representação alternativa do diagrama (para threads) .

# Estado de um Processo

Conforme um processo executa, ele muda de estado:<sup>a</sup>

- **Novo:** O processo está sendo criado.
- **Em execução:** Instruções estão sendo executadas.
- **Em espera:** O processo está esperando por um evento.
- **Pronto:** O processo está esperando que seja atribuído a um processador.
- **Concluído:** O processo terminou sua execução.

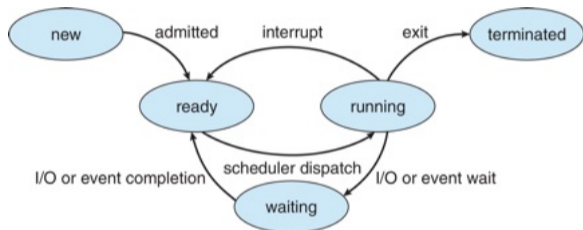



Figura 12: Diagrama de estados de um processo. Créditos: Silberschatz, Galvin and Gagne, 2018.

<sup>a</sup>No livro de Max Hailperin (2019), há uma representação alternativa do diagrama (para threads) .

# Estado de um Processo

Conforme um processo executa, ele muda de estado:<sup>a</sup>

- **Novo:** O processo está sendo criado.
- **Em execução:** Instruções estão sendo executadas.
- **Em espera:** O processo está esperando por um evento.
- **Pronto:** O processo está esperando que seja atribuído a um processador.
- **Concluído:** O processo terminou sua execução.

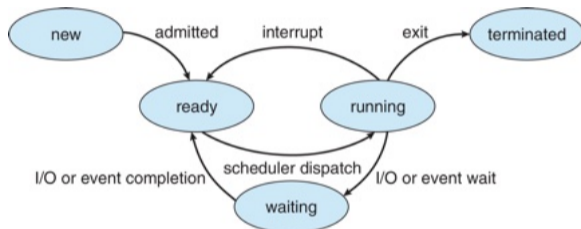



Figura 12: Diagrama de estados de um processo. Créditos: Silberschatz, Galvin and Gagne, 2018.

<sup>a</sup>No livro de Max Hailperin (2019), há uma representação alternativa do diagrama (para threads) .

# Estado de um Processo

Conforme um processo executa, ele muda de estado:<sup>a</sup>

- **Novo:** O processo está sendo criado.
- **Em execução:** Instruções estão sendo executadas.
- **Em espera:** O processo está esperando por um evento.
- **Pronto:** O processo está esperando que seja atribuído a um processador.
- **Concluído:** O processo terminou sua execução.

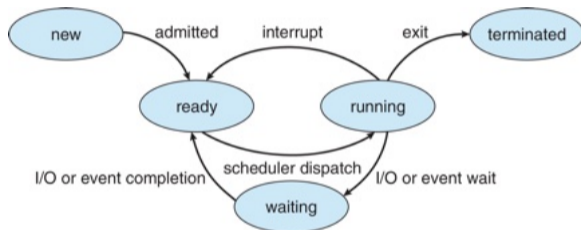



Figura 12: Diagrama de estados de um processo. Créditos: Silberschatz, Galvin and Gagne, 2018.

<sup>a</sup>No livro de Max Hailperin (2019), há uma representação alternativa do diagrama (para threads) .

# Estado de um Processo

Conforme um processo executa, ele muda de estado:<sup>a</sup>

- **Novo:** O processo está sendo criado.
- **Em execução:** Instruções estão sendo executadas.
- **Em espera:** O processo está esperando por um evento.
- **Pronto:** O processo está esperando que seja atribuído a um processador.
- **Concluído:** O processo terminou sua execução.

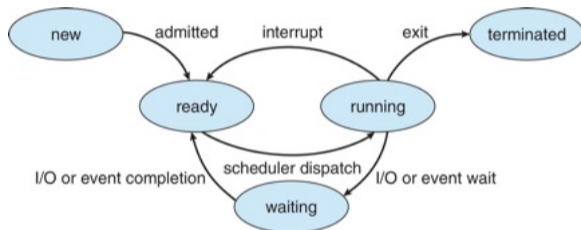



Figura 12: Diagrama de estados de um processo. Créditos: Silberschatz, Galvin and Gagne, 2018.

<sup>a</sup>No livro de Max Hailperin (2019), há uma representação alternativa do diagrama (para threads) .

# *Debug* seu Conhecimento



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.

# 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.*
  - ▶ Prática sobre processos: Utilizar linha de comando

# Dúvidas e Discussão

Prof. Dr. Denis M. L. Martins

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