

Sistemas Hardware-Software

Aula 13 – Processos

Engenharia

Fabio Lubacheski

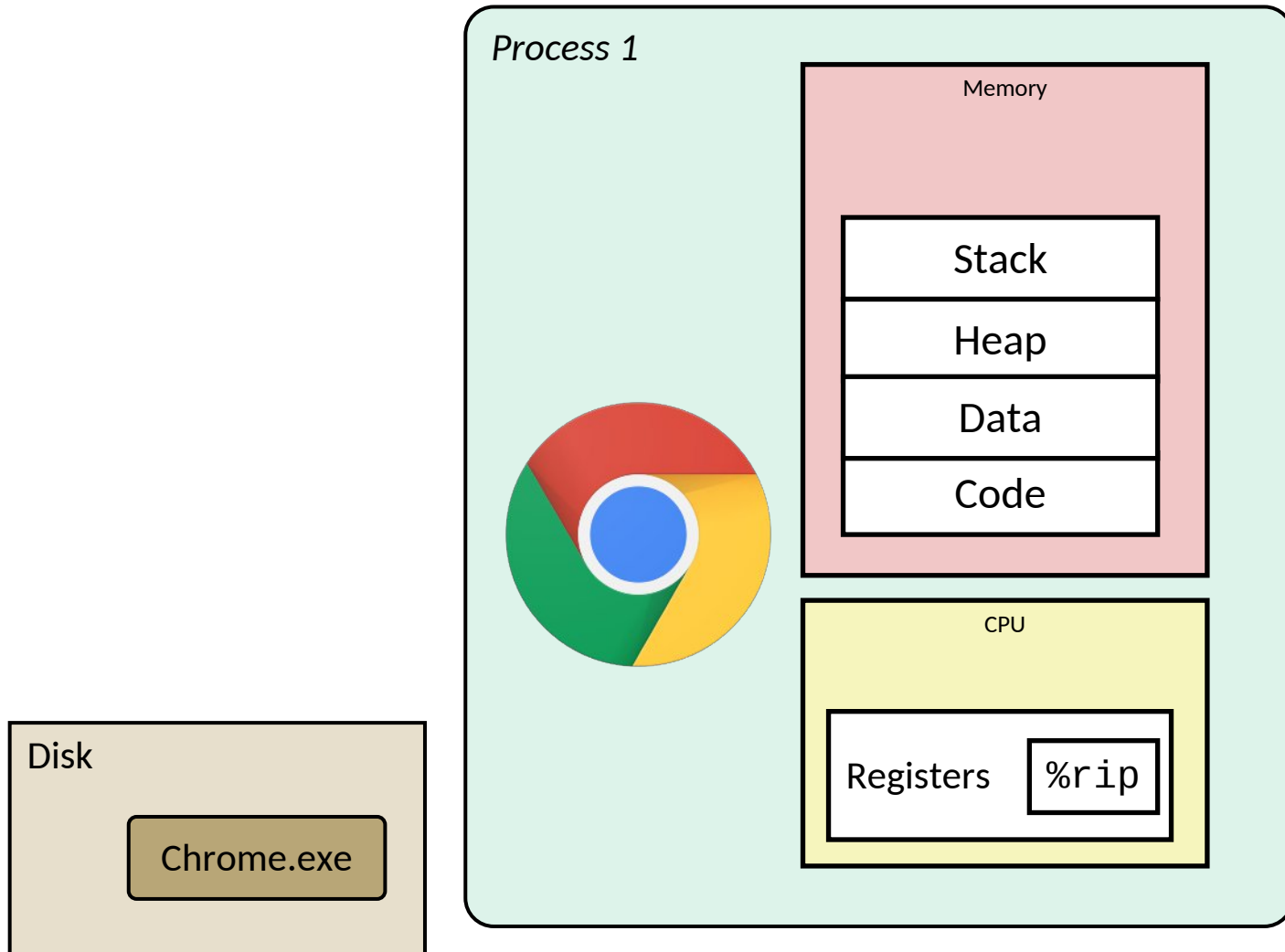
Maciel C. Vidal

Igor Montagner

Fábio Ayres

Processos

O que é um processo ?

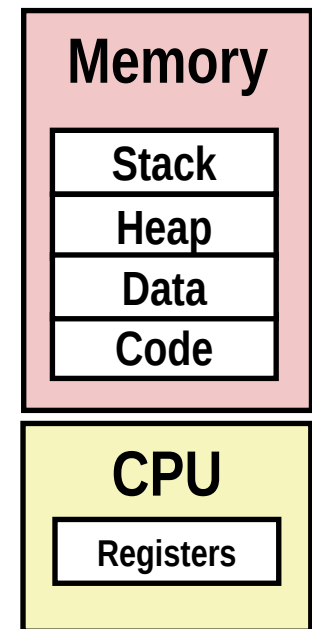


Processos

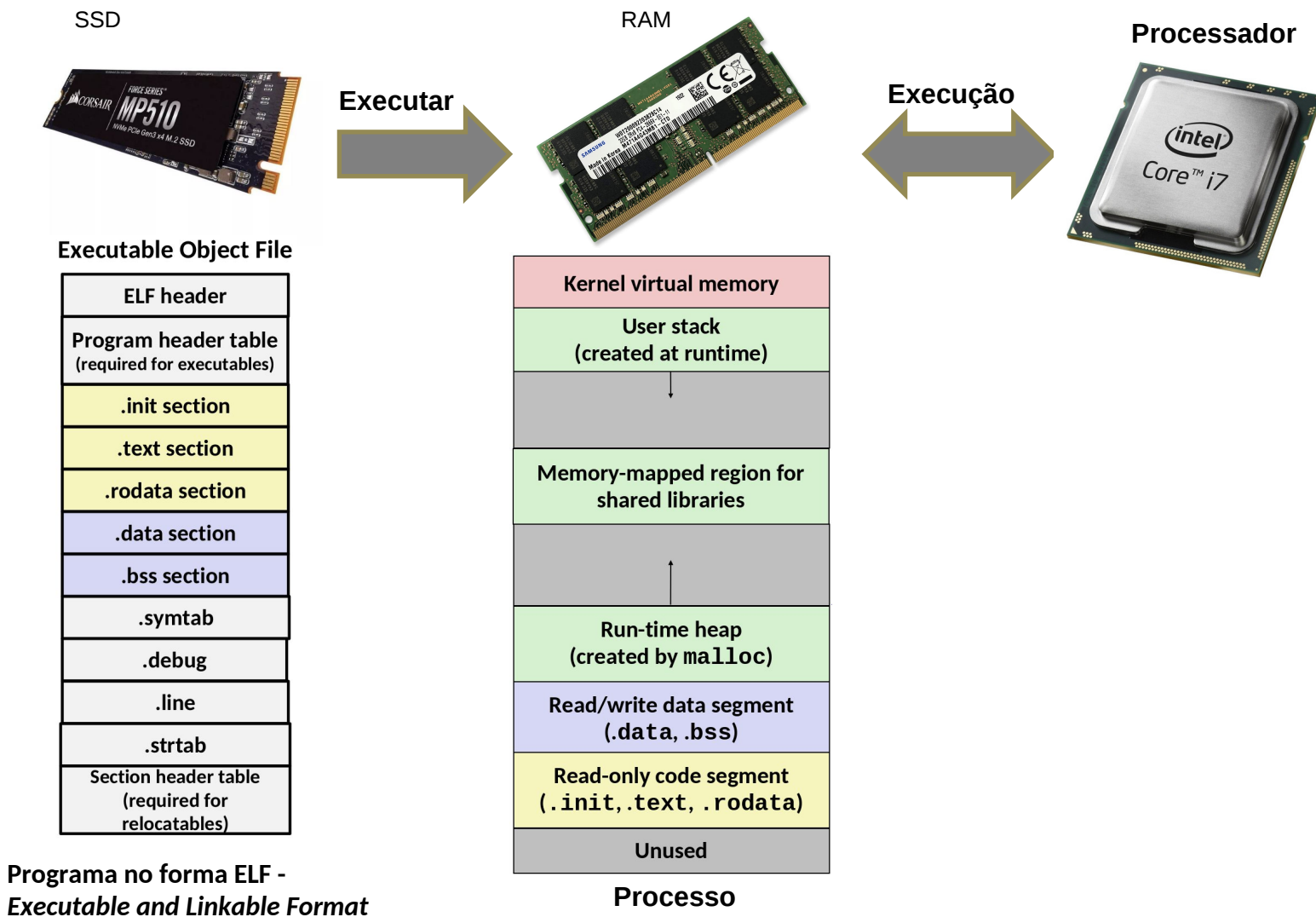
Um **processo** é uma **instância** de um **programa** em execução

O **processo** fornece a cada **programa** duas abstrações principais:

- **Fluxo de controle lógico**
 - Cada programa parece ter uso exclusivo da CPU
 - Fornecido pelo mecanismo do kernel chamado troca de contexto (**context switching**)
- **Espaço de endereço privado**
 - Cada programa parece ter uso exclusivo da memória principal
 - Fornecido pelo mecanismo do kernel chamado memória virtual
- **Qual a diferença entre:**
 - Um processador ? Um programa ? Um Processo ?



Processos

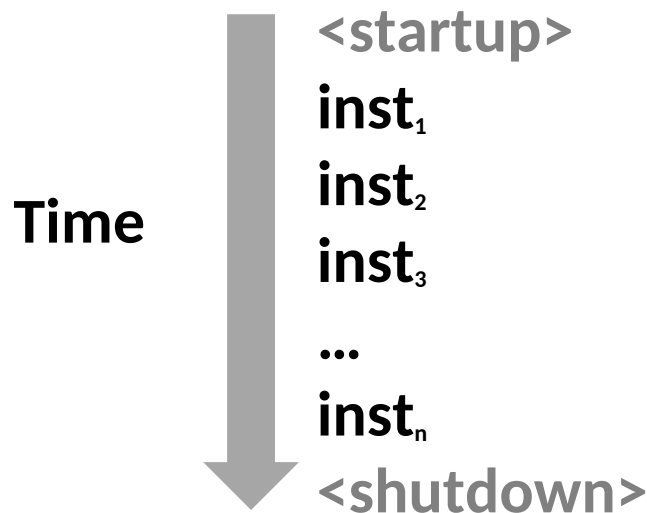


Fluxo de controle

Desde o início até o seu desligamento, a CPU apenas lê e executa uma sequência de **instruções**, uma por vez

Esta sequência é o fluxo de controle da CPU

Physical control flow



Sistemas Operacionais

Uma das principais funções do Sistema Operacional é permitir que um usuário execute diversos programas de maneira “*simultânea*” e segura.

Isso fica a cargo do **Kernel** do SO ou **núcleo do SO**.
Essa parte do SO gerencia:

- Programas
- Memória
- Recursos do hardware

O Kernel executa com **privilégios totais** no hardware.

Sistemas Operacionais

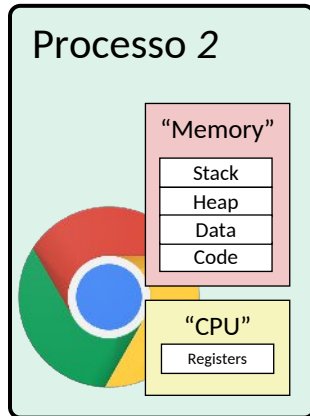
Processo de usuário: qualquer programa sendo executado no computador. **A falha de um processo não afeta os outros.**

Roda com **privilégios limitados**. Interage com o hardware por meio de **chamadas ao kernel** para obter

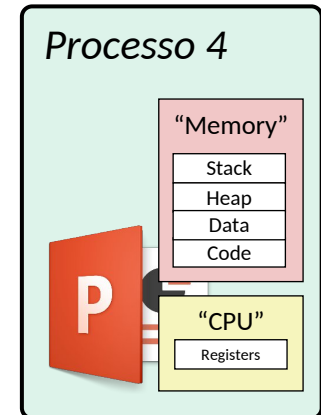
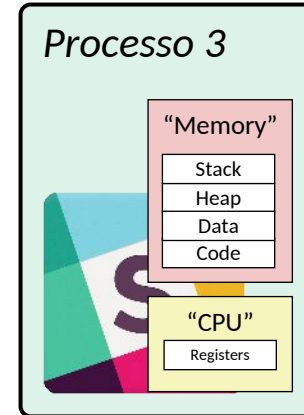
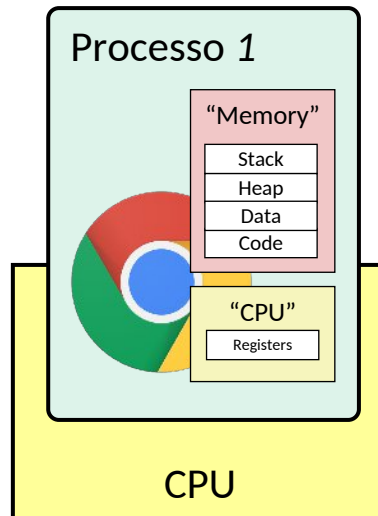
- Memória
- Acesso ao disco e outros periféricos
- Comunicar com outros processos

A ilusão do multiprocessamento - 1 CPU

Computador



Operating
System
(*Kernel*)



Disk

/Applications/

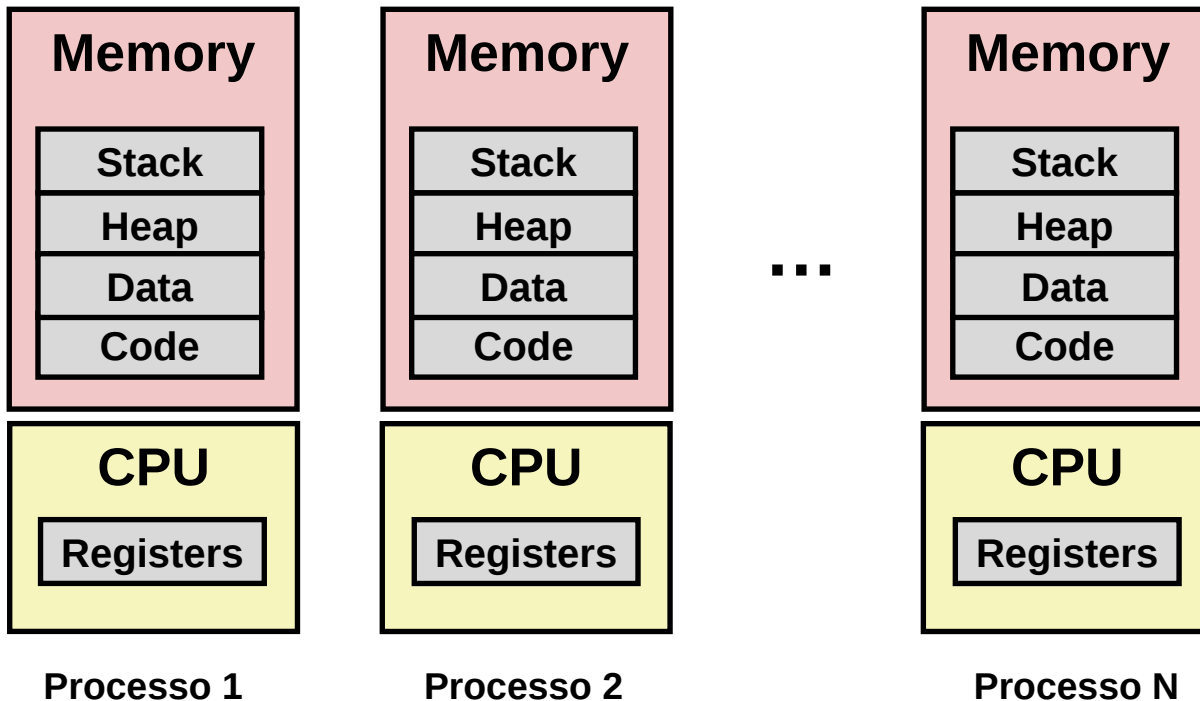


Chrome.exe

Slack.exe

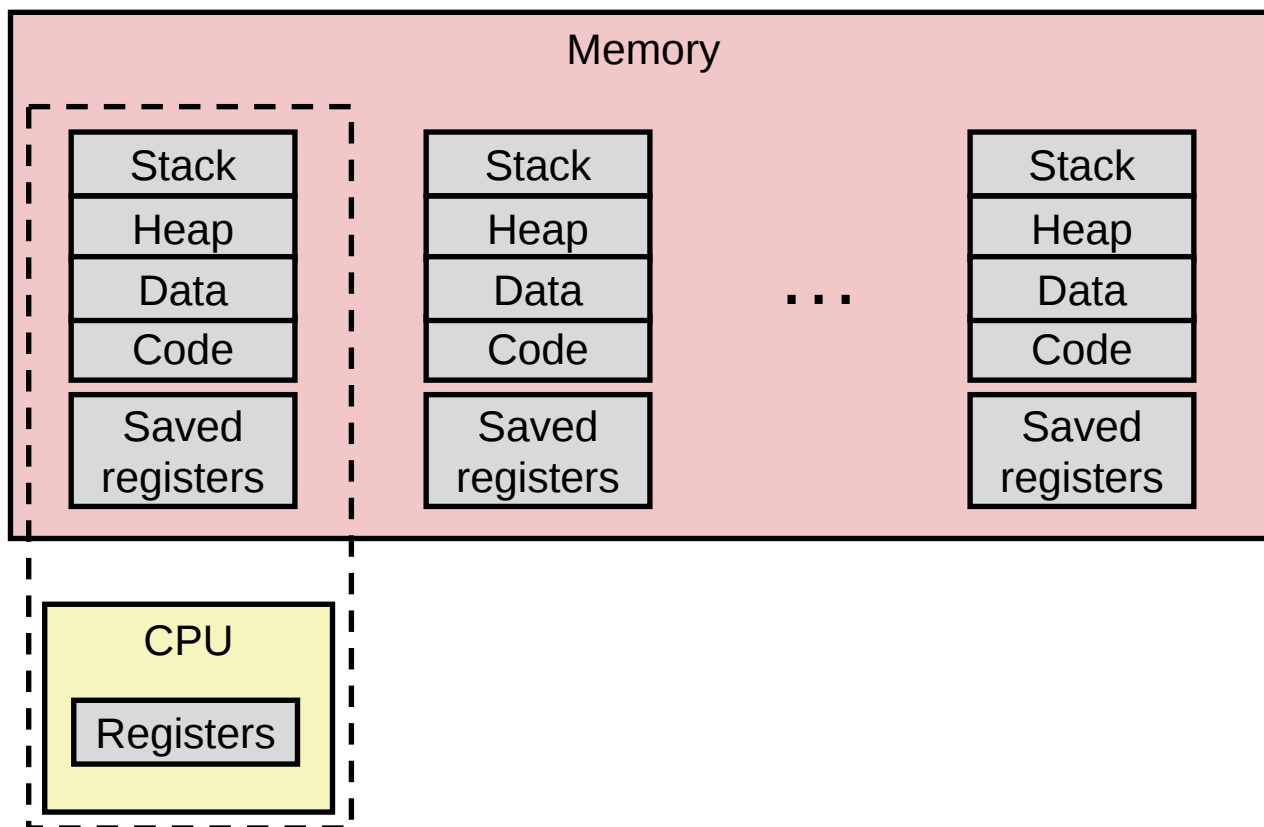
PowerPoint.exe

A ilusão do multiprocessamento – 1 CPU



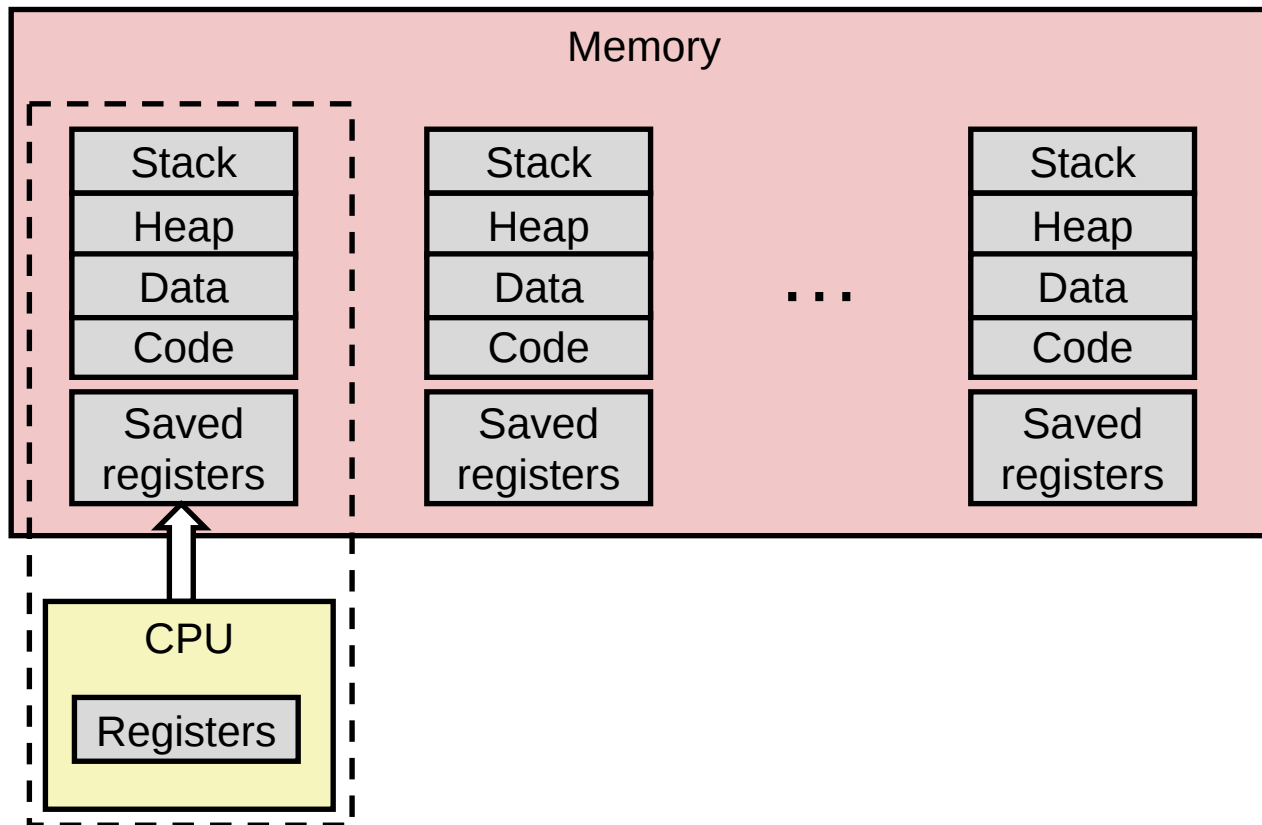
- **Um computador roda vários processos simultaneamente:**
 - Aplicações de um ou mais usuários: Web Browsers, email clients, editors,...
 - Aplicações do SO: Monitoramento de rede & I/O, drivers, ...

A realidade do multiprocessamento - 1 CPU



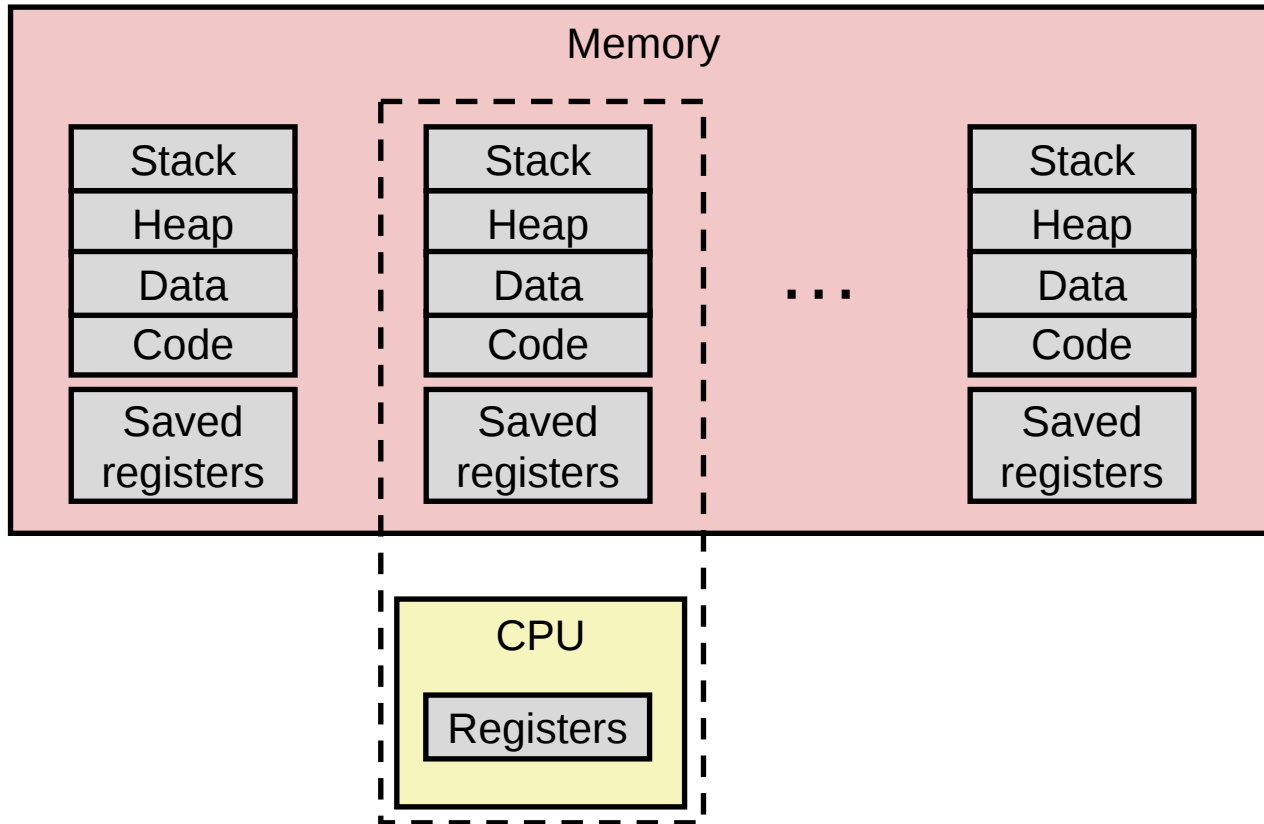
- **Um processador executa múltiplos processos concorrentes:**
- Execução de processos intercalada
- Espaços de endereçamento gerenciados pelo sistema de memória virtual
- Valores de registradores para processos em espera são gravados em memória

A realidade do multiprocessamento - 1 CPU



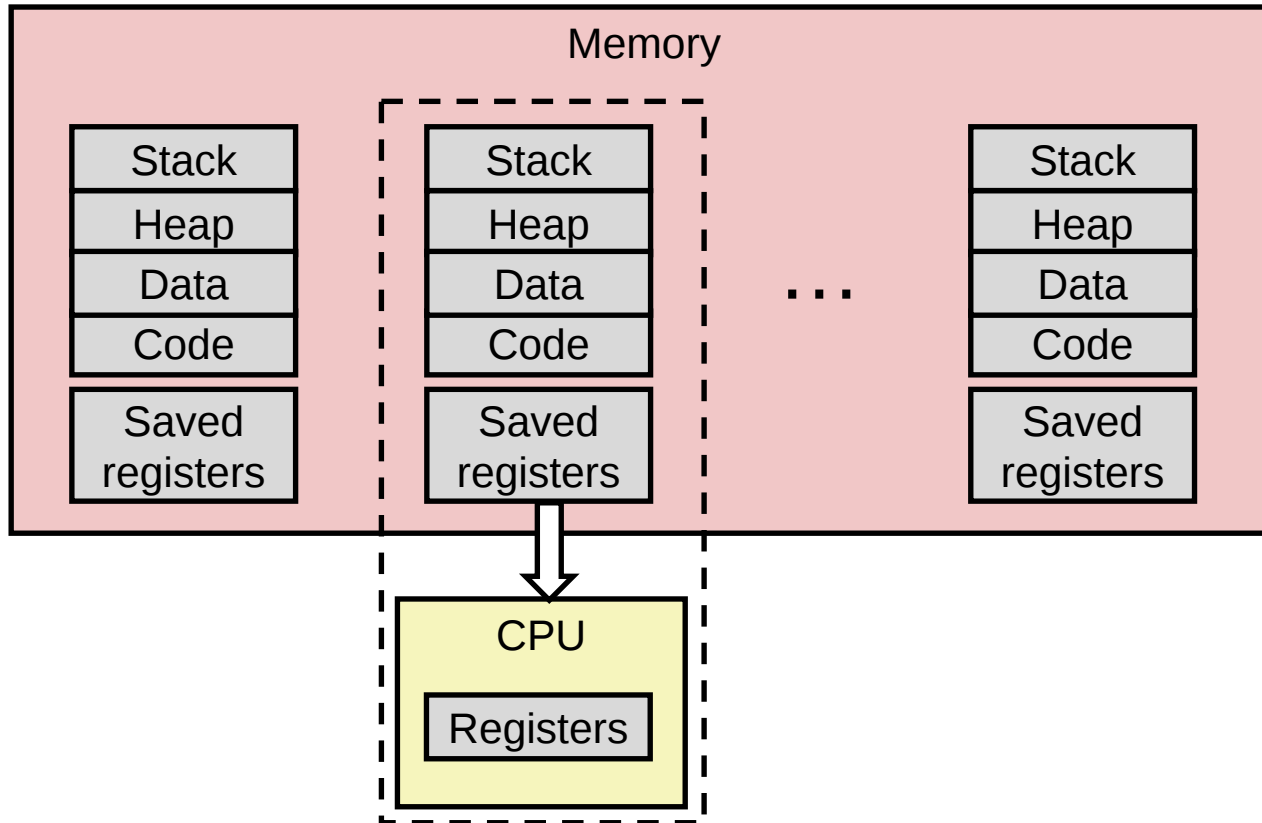
- **Troca de contexto (Context Switch)**
 - 1) Grava os registradores correntes na memória

A realidade do multiprocessamento - 1 CPU



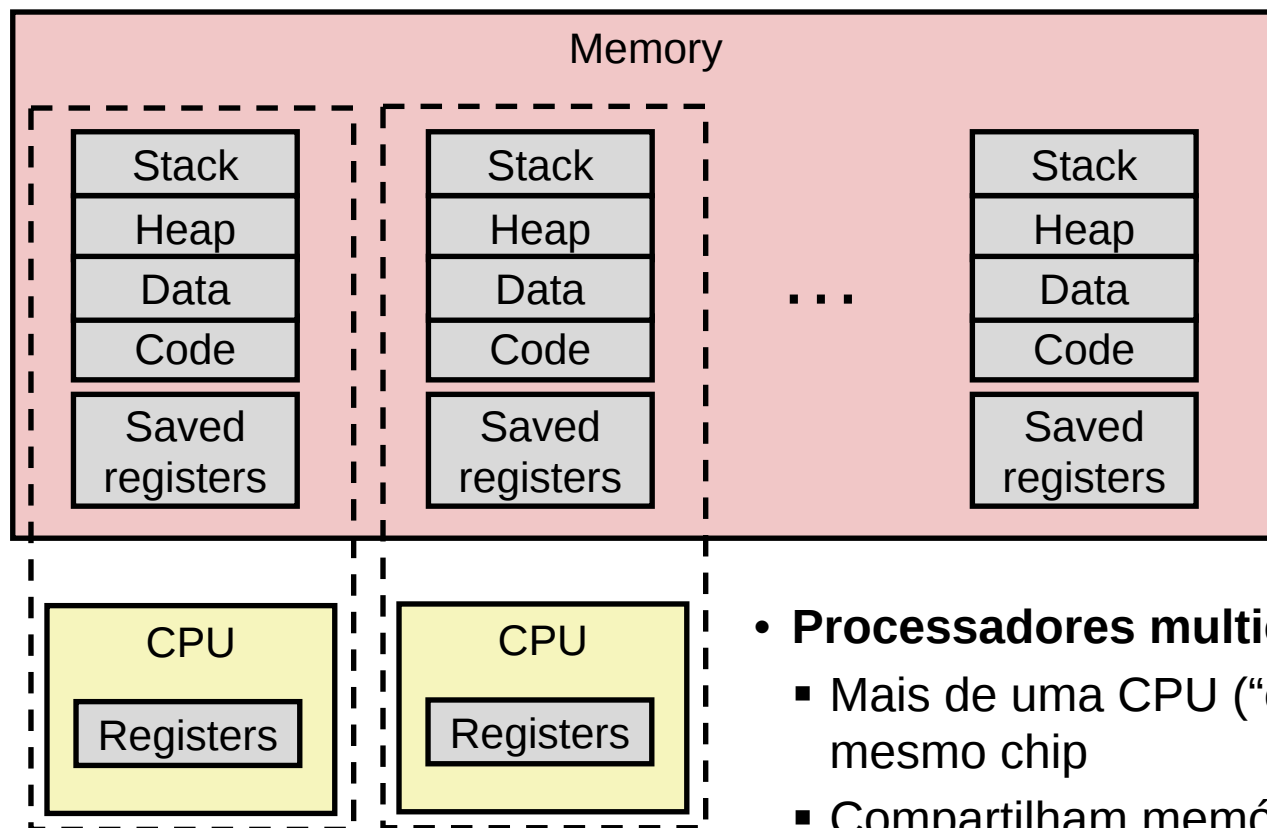
- **Troca de contexto** (Context Switch)
 - 2) Escolhe **próximo** processo a ser executado (SO decide)

A realidade do multiprocessamento - 1 CPU



- **Troca de contexto** (Context Switch)
 - 3) Carrega registradores gravados e troca de espaço de endereçamento (chaveamento de contexto)

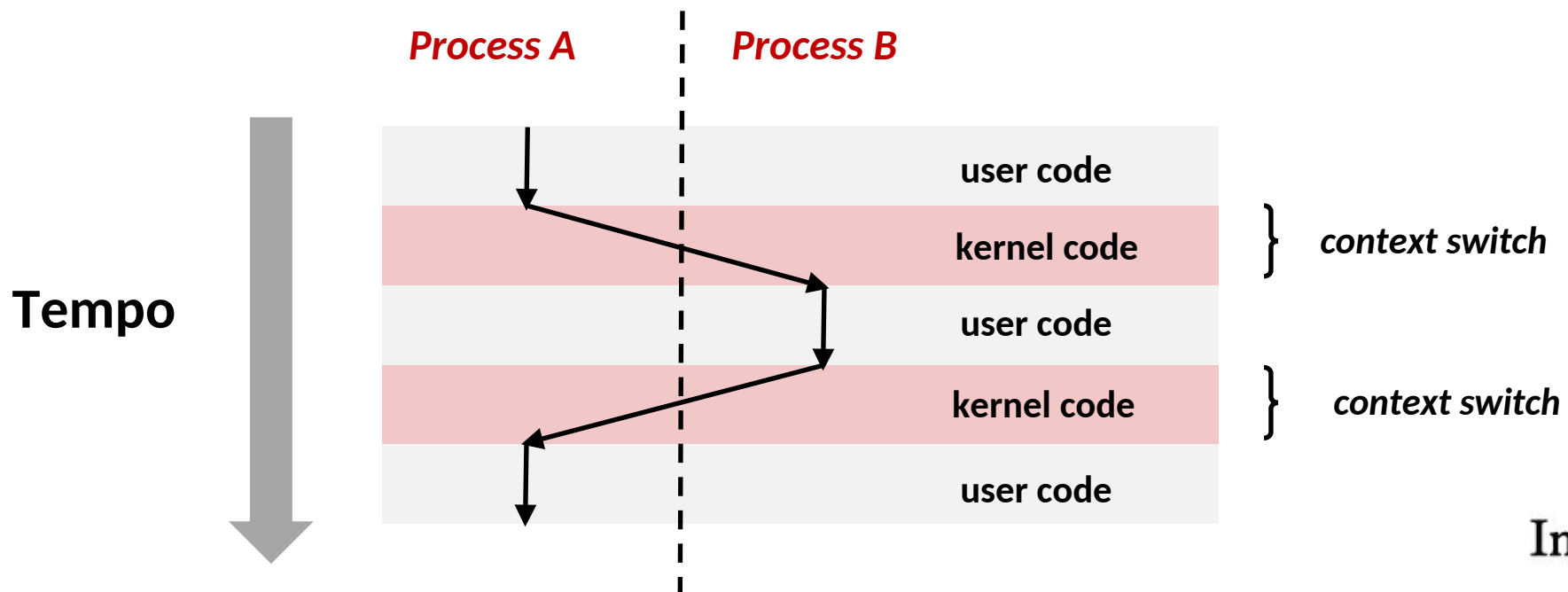
A realidade moderna do multiprocessamento



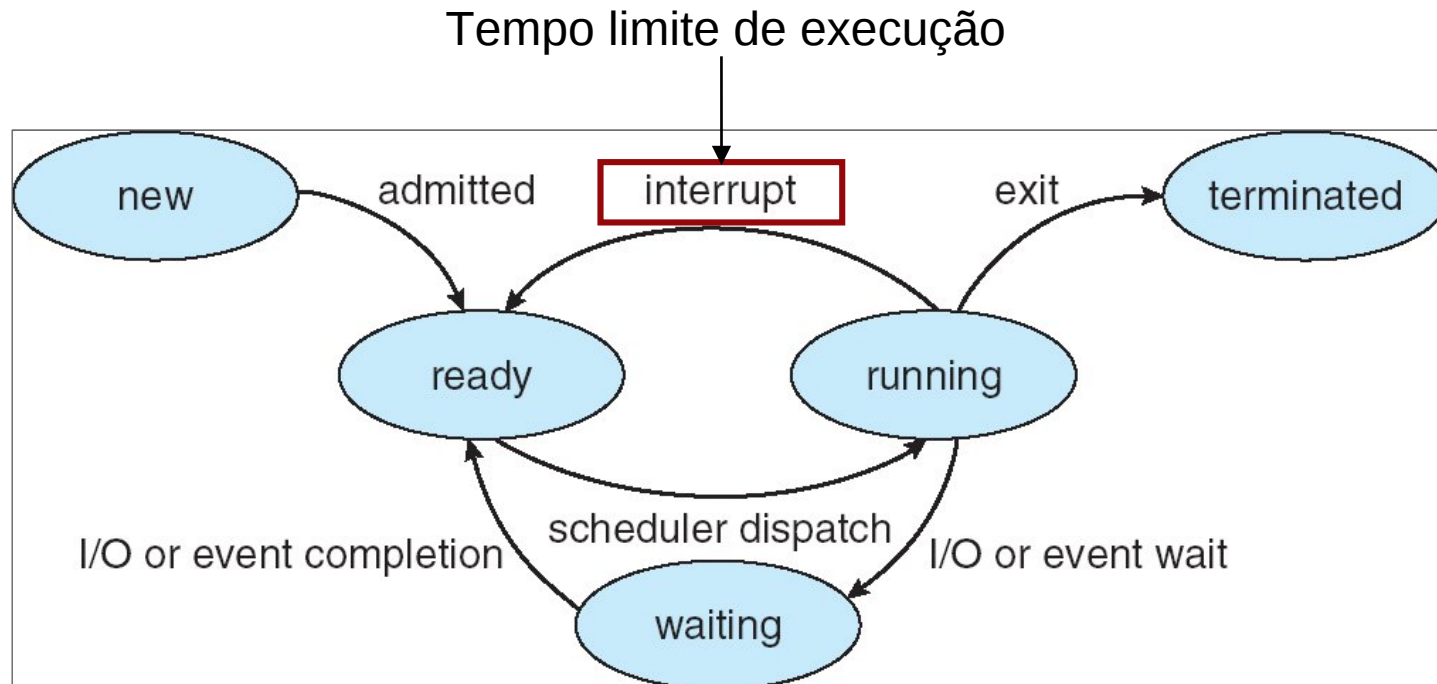
- **Processadores multicore**
 - Mais de uma CPU (“**core**”) em um mesmo chip
 - Compartilham memória principal e parte do cache (cache L3)
 - Cada core pode executar um processo separado
 - Agendamento de processos em core’s feito pelo kernel

Troca de contexto

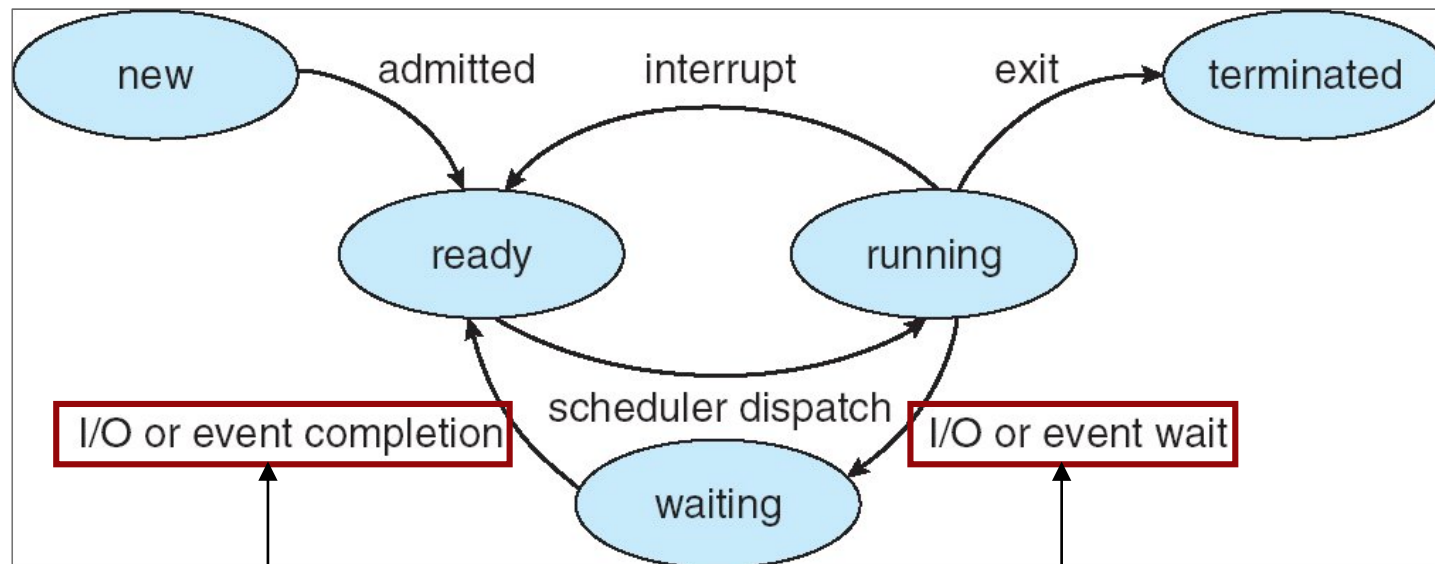
- Os processos são gerenciados por um trecho de código do sistema operacional chamado de **Kernel**;
- O **Kernel** não é um processo separado, mas é executado como parte de um processo do usuário;
- A troca de contexto passa o fluxo de controle de um processo para outro e é executada usando o **código do kernel**.



Ciclo de vida de processos



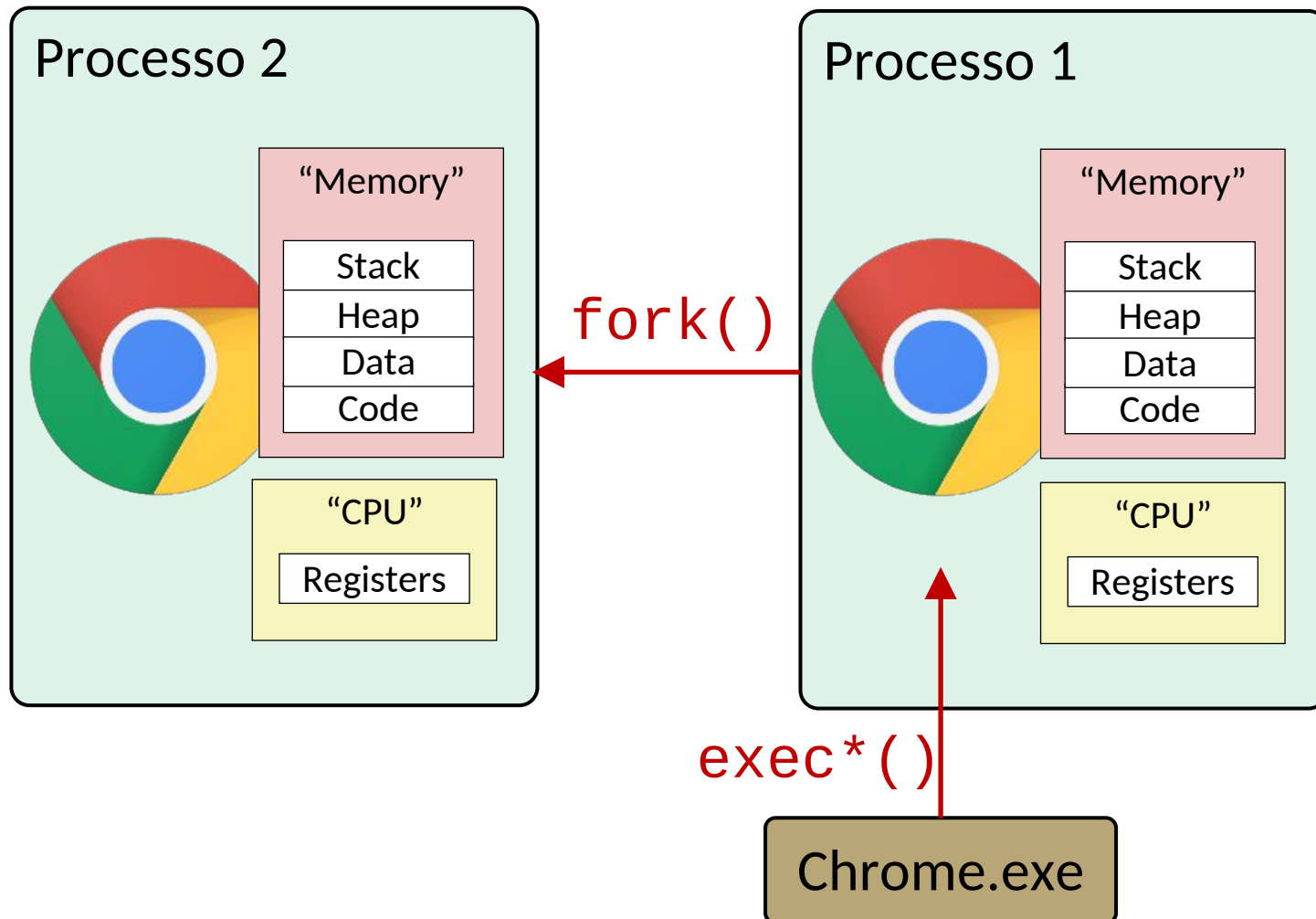
Ciclo de vida de processos



Interrupção ligada por
algum hardware (disco,
rede, usb, etc)

Chamada de sistema
Para acessar
hardware

Criação de processos



Implementando o multiprocessamento

A implementação do multiprocessamento nos sistemas operacionais modernos é realizada através de chamada de sistemas (***system calls***).

Uma chamada de sistema (***system call***) é o mecanismo usado pelo processo para requisitar um serviço do sistema operacional, ou mais especificamente, do **Kernel** do sistema operacional.

No **Linux** utilizamos a API (*Application programming interface*) implementada no padrão POSIX (*Portable Operating System Interface*).

POSIX

A **Portable Operating System Interface** (POSIX) é uma família de padrões especificados pela IEEE *Computer Society* para manter a compatibilidade entre sistemas operacionais.

POSIX define a **interface de programação de aplicativos** (API), juntamente com *shells* de linha de comando e interfaces de utilitários, para compatibilidade de software com variantes do Unix e outros sistemas operacionais

POSIX

- Gerenciamento de usuários e grupos
- Manipulação de arquivos (incluindo permissões) e diretórios
- Criação de processos e carregamento de programas
- Comunicação entre processos
- Interação direta com hardware (via drivers)

Criação de processos

Criamos processos usando a chamada de sistema *fork*

```
pid_t fork();
```

O **fork** cria um **clone** do processo atual e retorna duas vezes

No processo **original** (pai)
`fork` retorna o pid do filho

O pid do pai é obtido chamando

```
pid_t getpid();
```

No processo **filho** `fork` retorna o valor 0.
O pid do filho é obtido usando

```
pid_t getpid();
```

O pid do pai pode ser obtido usando a chamada

```
pid_t getppid();
```

Entendendo melhor o **Fork**

```
int x = 7;
if( fork() ) {
    x++;
    printf(" %d ", x);
    fork();
    x++;
    printf(" %d ", x);
} else {
    printf(" %d ", x);
}
```

Quantos processos são criados ?

Qual valor da variável x quando cada processo começa executar ?

Qual a saída do trecho de código ao lado ?

Pode ocorrer de mudar a ordem dos valores impressos na saída ?

Veja a resposta no arquivo **fork.pdf**



Atividade prática

A chamada fork

1. Criação de processos
2. Utilização do manual para dúvidas sobre as chamadas

Criação de processos – Exercício 3

O pai espera o filho ??

```
int rodando = 1;
pid_t filho;

filho = fork();

if (filho == 0) {
    printf("Acabei filho\n");
    rodando = 0;
} else {
    while (rodando) {
        printf("Esperando o filho acabar!\n");
        sleep(1);
    }
}
return 0;
```

Valor de retorno

- Um processo pode esperar pelo fim de outro processo filho usando as funções

```
pid_t wait(int *wstatus);  
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- A primeira espera qualquer um dos filhos, enquanto a segunda espera um filho (ou grupo de filhos) específico.
- Ambas bloqueiam até que um processo filho termine e retornam o pid do processo que acabou de terminar.
- O valor de retorno do processo é retornado via o ponteiro `wstatus`.

E se o processo filho deu ruim?

- É possível checar se um processo filho terminou corretamente usando o conteúdo de `wstatus` e as seguintes macros:
- `WIFEXITED(wstatus)`: `true` se o filho acabou sem erros
- `WEXITSTATUS(wstatus)`: valor retornado pelo `main`
- `WIFSIGNALED(wstatus)`: `true` se o filho foi terminado de maneira abrupta (tanto por um `ctrl+c` quanto por um erro)
- `WTERMSIG(wstatus)`: código numérico representando a razão do encerramento do filho



Atividade prática

A chamada wait

1. Criação de processos
2. Identificação de término de processos
3. Utilização do manual para dúvidas sobre as chamadas

Como executar novos programas?

- **fork** só permite a criação de **clones** de um processo!
- Família de funções **exec** permite o carregamento de um programa do disco
- É permitido setar as variáveis de ambiente do novo programa e seus argumentos.
- Funções da família **exec** nunca retornam: o programa atual é destruído durante o carregamento do novo programa

Insper

www.insper.edu.br