Insper

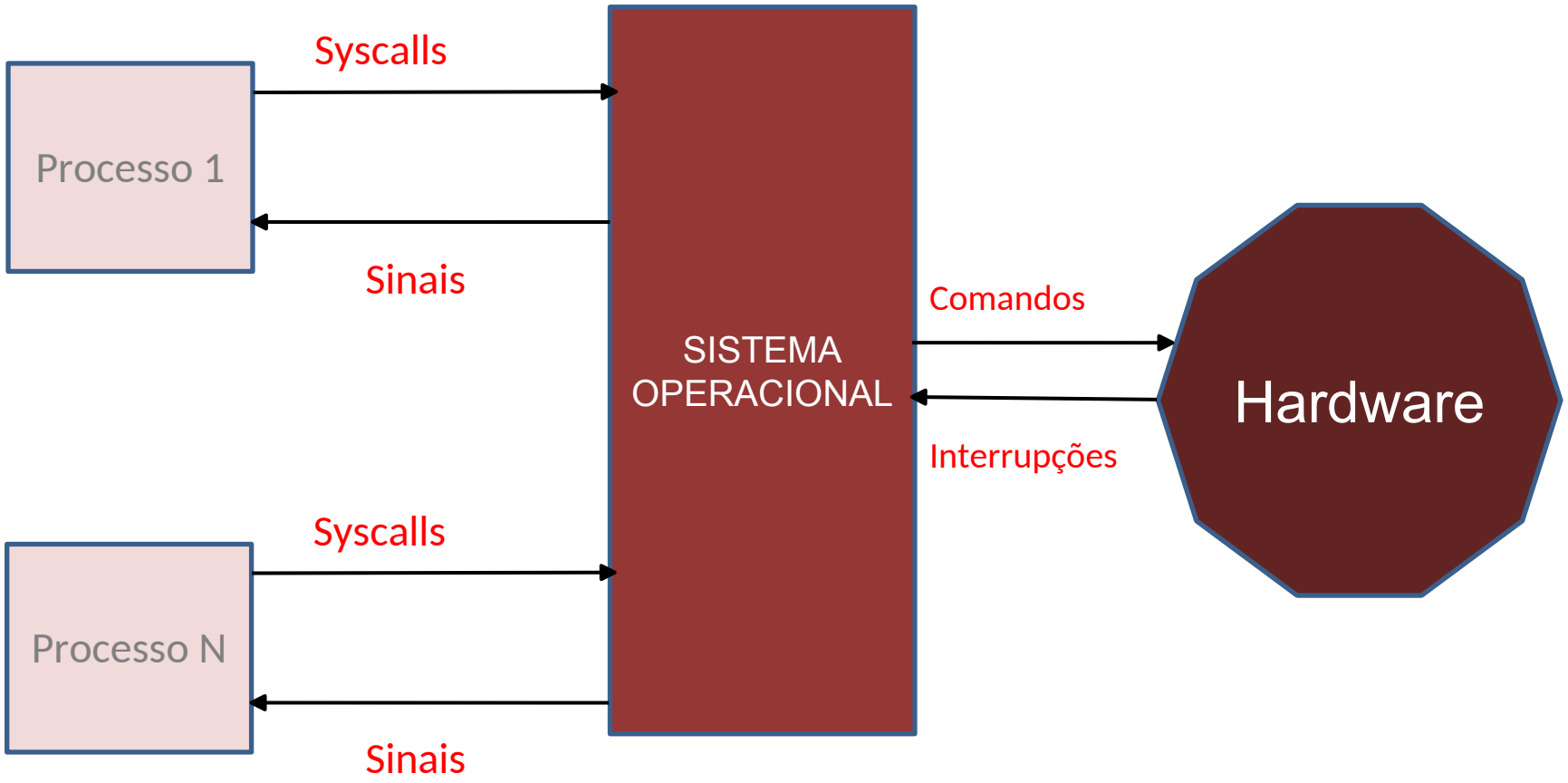
Sistemas Hardware-Software

Carregamento de Programas

Ciência da Computação

Carlos Menezes
Maciel Vidal
Igor Montagner
Fábio Ayres

Sistemas Operacionais



Insper

POSIX

The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the **application programming interface (API)**, along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems

1) Wikipedia

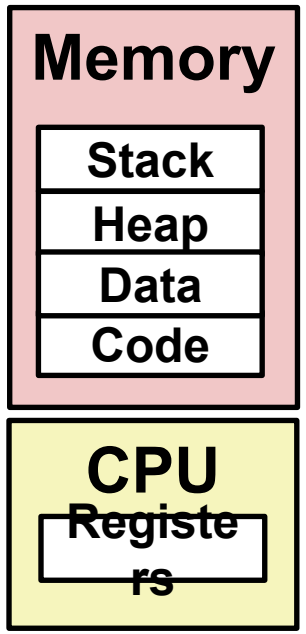
POSIX - syscalls

- 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)

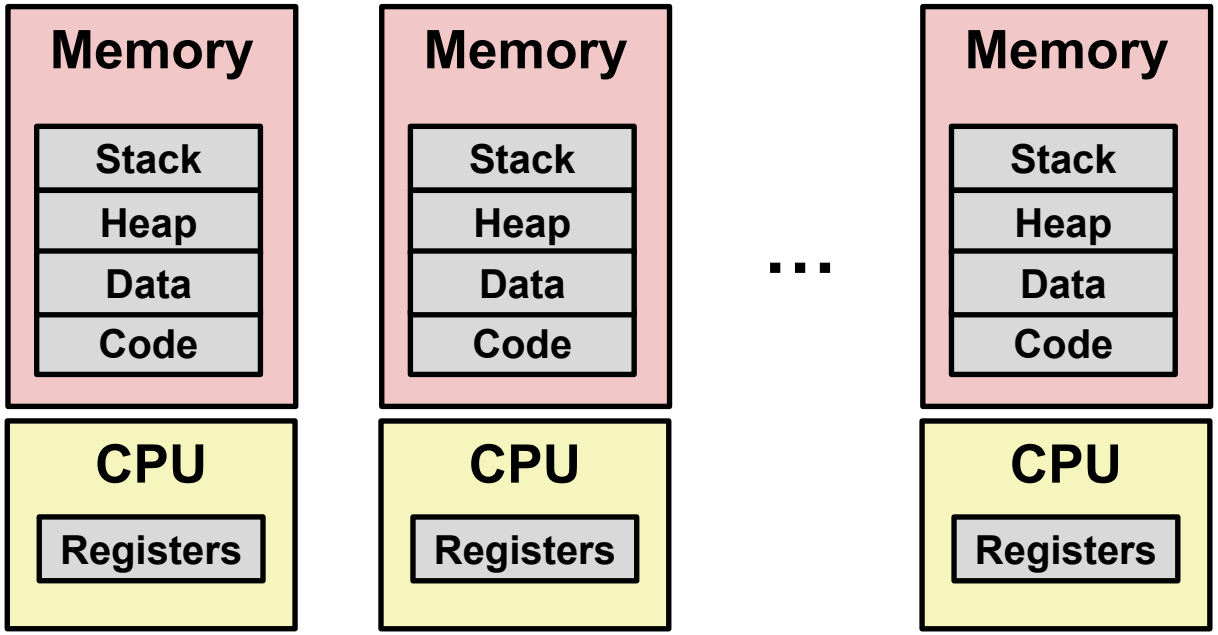
Processos

- Fluxo de controle lógico
  - Cada programa parece ter uso exclusivo da CPU
  - Provido pelo mecanismo de *chaveamento de contexto*

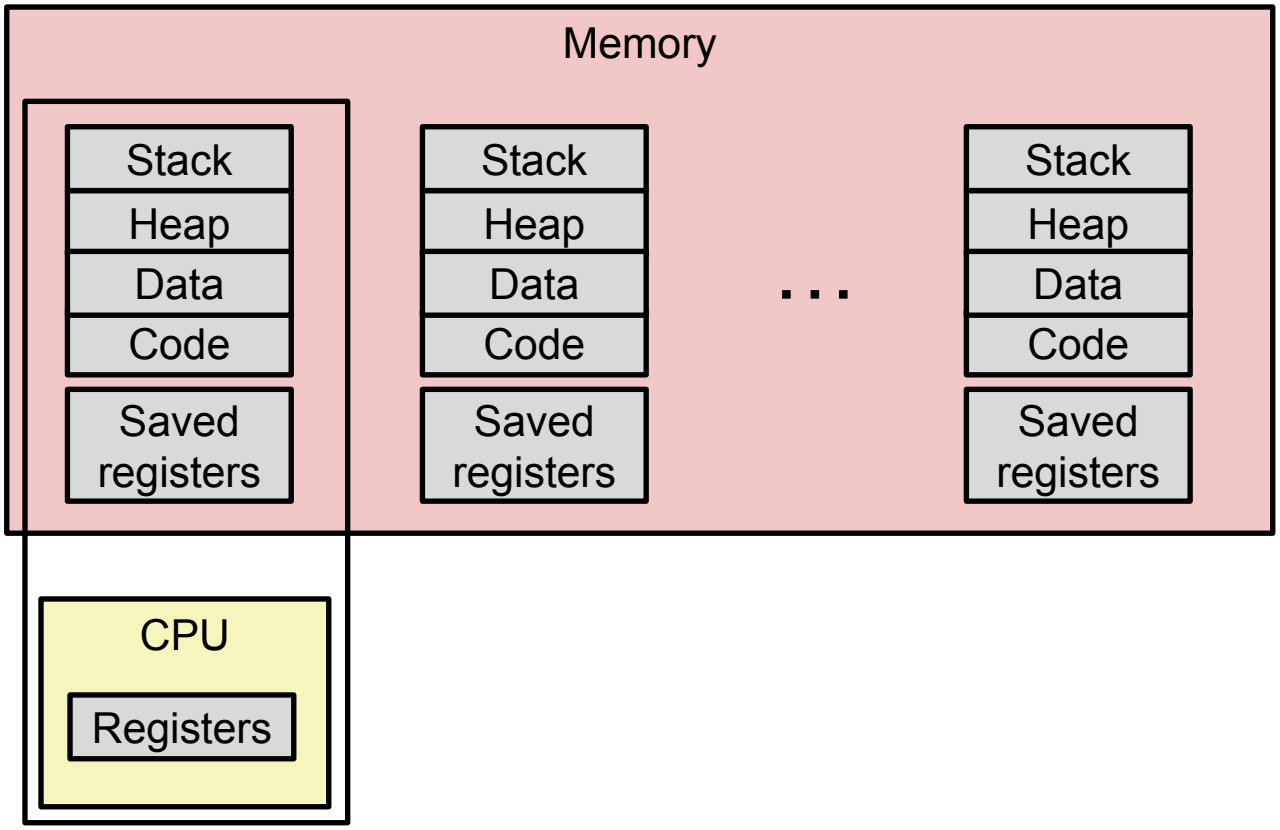
- Espaço de endereçamento privado
  - Cada programa parece ter uso exclusivo da memória principal
  - Provido pelo mecanismo de *memória virtual*



A ilusão do multiprocessamento

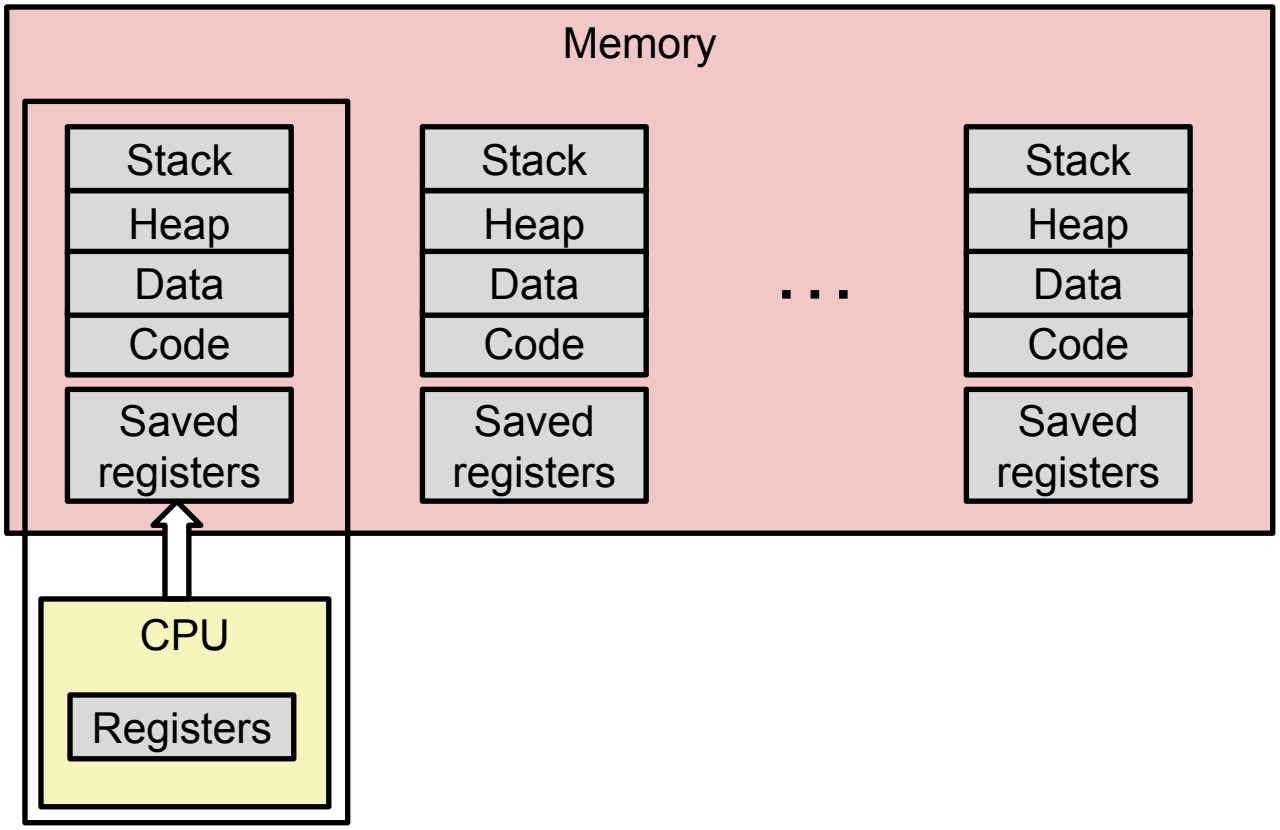


A realidade do multiprocessamento



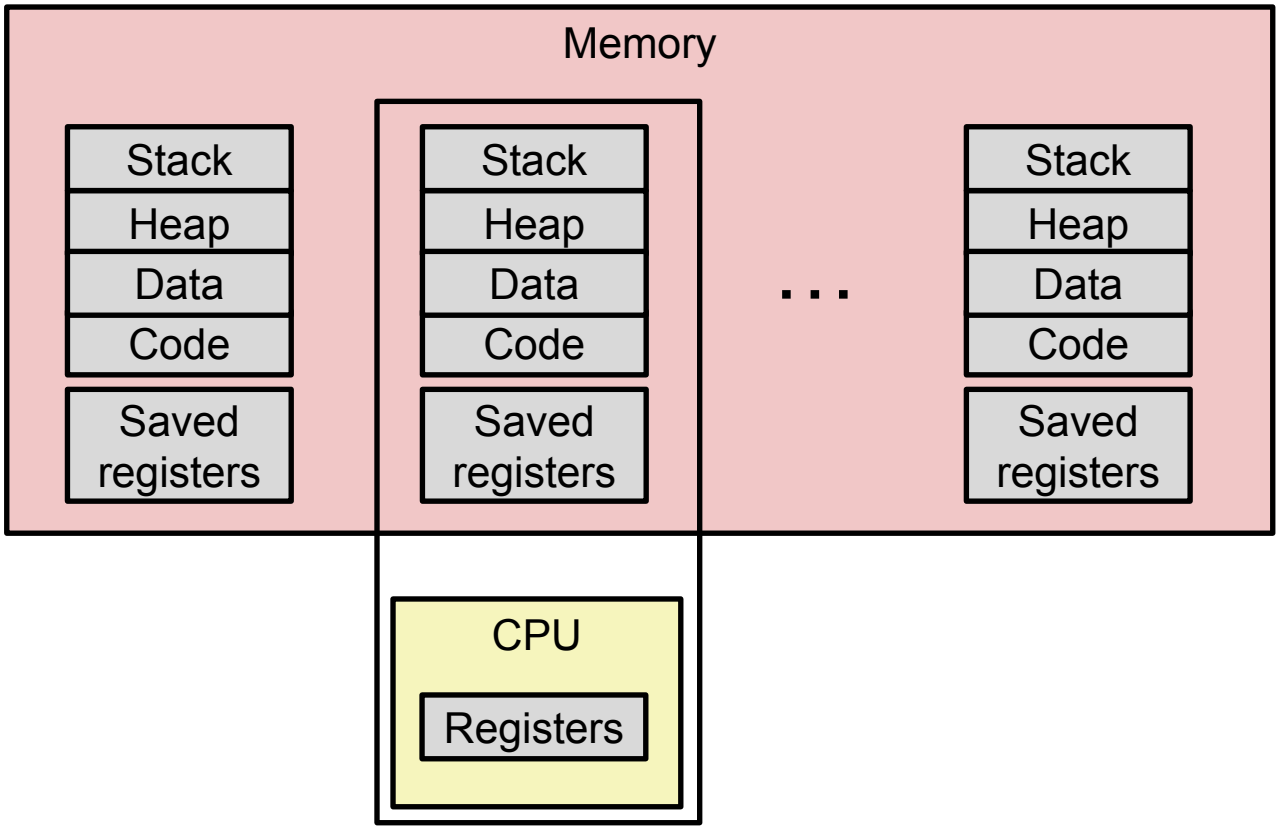
- Execução de processos intercalada
- Espaços de endereçamento gerenciados pelo sistema de memória virtual
- Valores de registradores para processos em espera são gravados em memória

A realidade do multiprocessamento



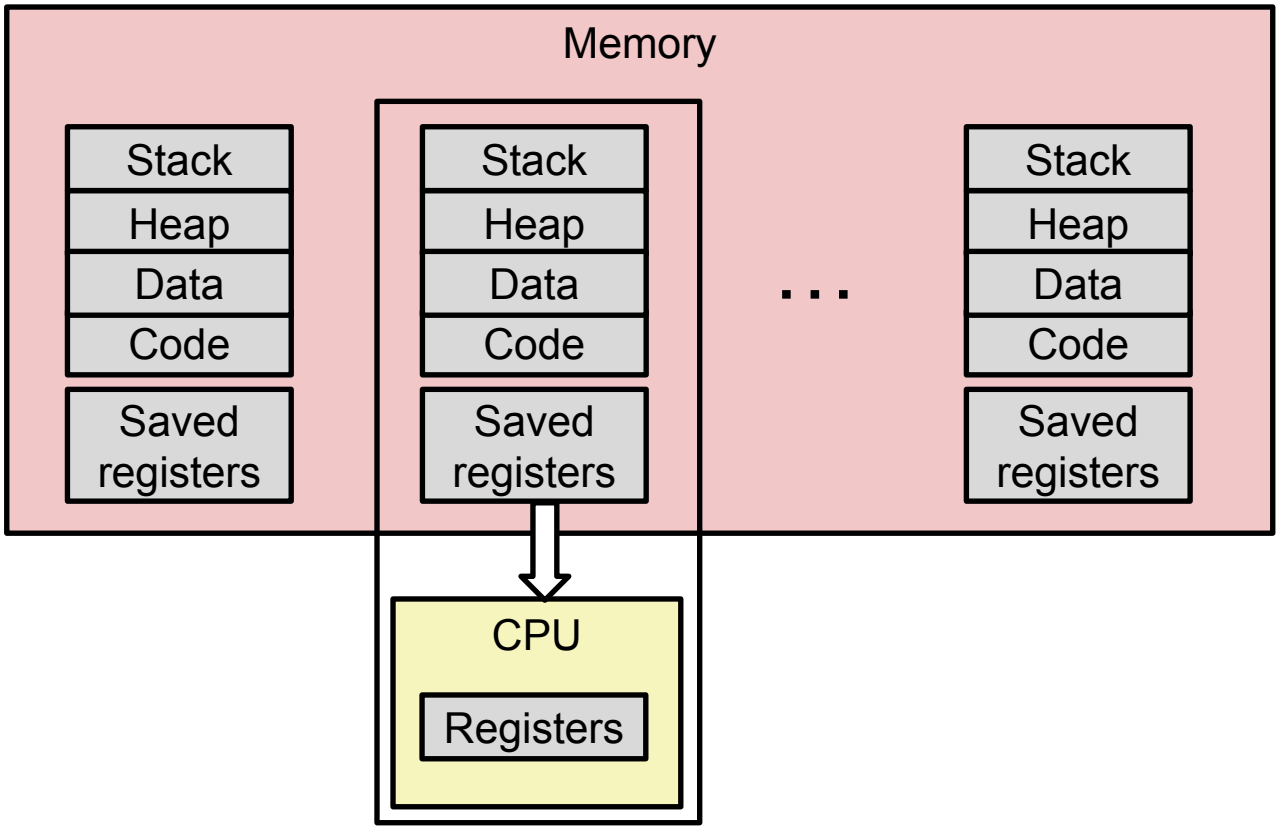
- Grava registradores na memória

A realidade do multiprocessamento



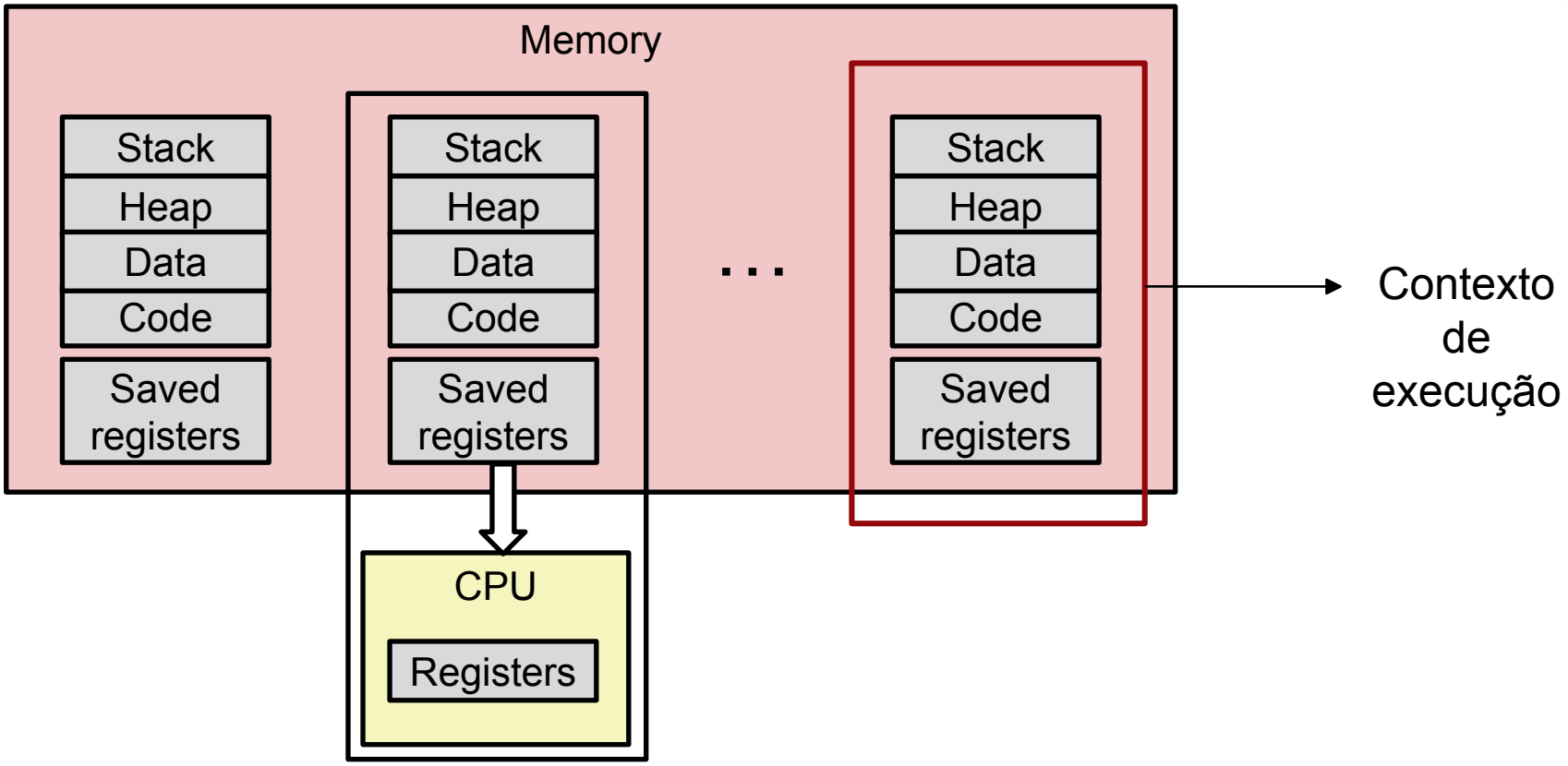
- Escolhe próximo processo a ser executado

A realidade do multiprocessamento



- Carrega registradores gravados e troca de espaço de endereçamento (*context switch* – chaveamento de contexto)

A realidade do multiprocessamento



- Carrega registradores gravados e troca de espaço de endereçamento (*context switch* – chaveamento de contexto)

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();
```

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);
```

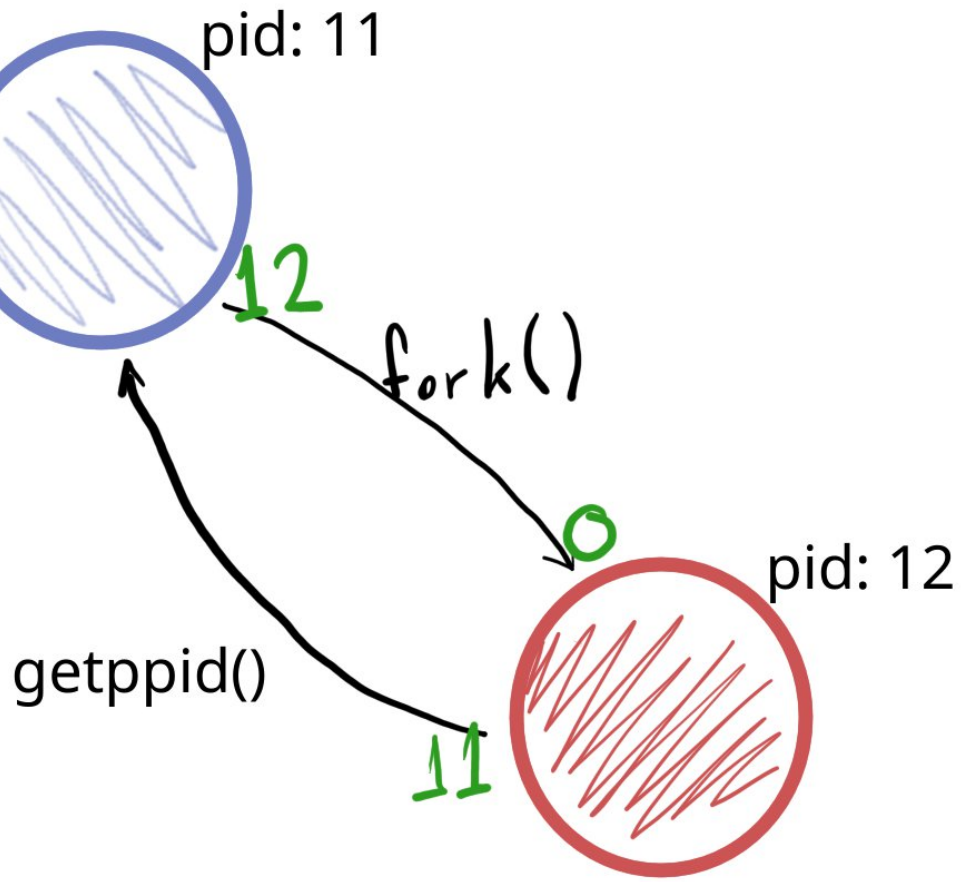
- A primeira espera qualquer um dos filhos, enquanto a segunda espera um filho (ou grupo de filhos) específico.
- Ambas bloqueiam até que um processo filho termine e retornam o pid do processo que acabou de terminar.
- O valor de retorno do processo é retornado via o ponteiro `wstatus`.

Correção

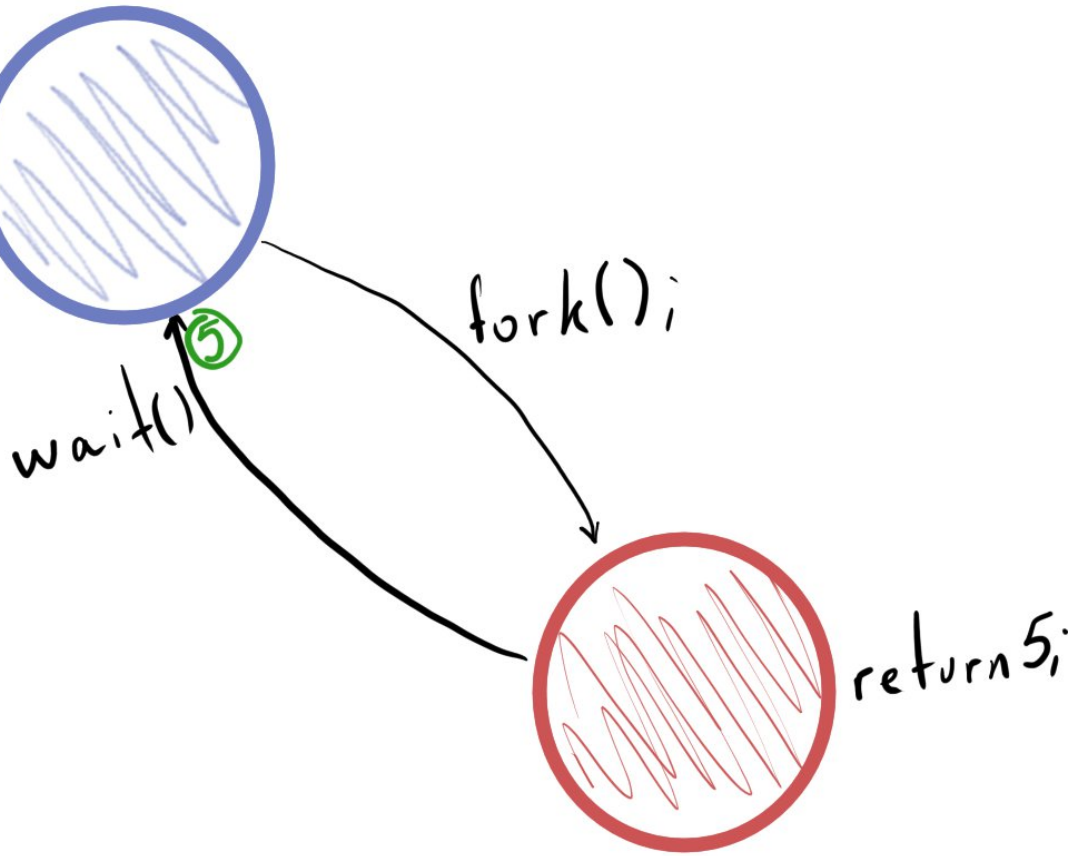
A chamada wait

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

Parentesco de processos



Parentesco de processos – II



Atividade prática

Argumentos: main(int argc, char *argv[]) (20 minutos)

1. Recepção de argumentos por programas
2. Conversão de strings para inteiros

A chamada `execvp`

```
int execvp(const char *file, char *const argv[]);
```

A chamada **`execvp`** faz duas coisas:

1. Carrega um programa na memória dentro do contexto do processo atual
2. Inicia esse programa, preenchendo os argumentos do main

O programa que estava em execução antes do `execvp` é completamente destruído.

Exemplo de uso - argumentos

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    char prog[] = "ls";
    // a lista de argumentos sempre começa com o nome do
    // programa e termina com NULL
    char *args[] = {"ls", "-l", "-a", NULL};

    execvp(prog, args);
    printf("Fim do exec!\n");

    return 0;
}
```

Exemplo de uso - argumentos

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    char prog[] = "ls";
    // a lista de argumentos sempre começa com o nome do
    // programa e termina com NULL
    char *args[] = {"ls", "-l", "-a", NULL};

    execvp(prog, args);
    printf("Fim do exec!\n");

    return 0;
}
```

Essa linha só roda se o **`execvp`** falhar!

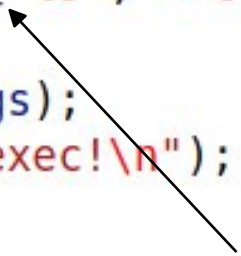
Exemplo de uso - argumentos

```c
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    char prog[] = "ls";
    // a lista de argumentos sempre começa com o nome do
    // programa e termina com NULL
    char *args[] = {"ls", "-l", "-a", NULL};

    execvp(prog, args);
    printf("Fim do exec!\n");

    return 0;
}
```



Argumento `char *argv[]` do main!
Também seta `argc = 3`, pois tem 3 strings!

Atividade prática

A chamada exec (30 minutos)

1. Carregamento de programas
2. Passagem de argumentos
3. Coleta de resultados de um programa

Insper

www.insper.edu.br