

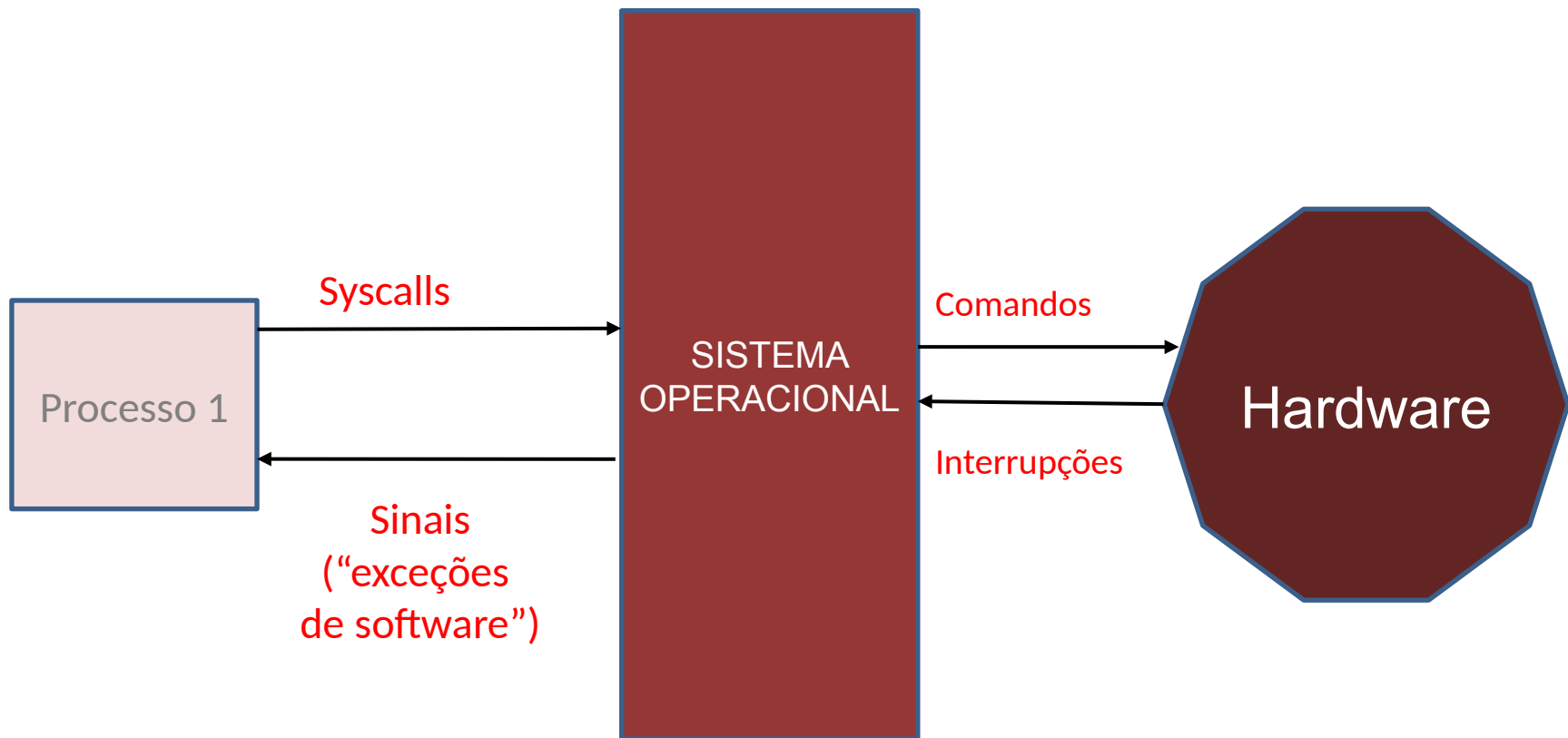
# **Sistemas Hardware-Software**

## Aula 18 – Relações entre processos

2019 – Engenharia

Igor Montagner, Fábio Ayres [<igorsm1@insper.edu.br>](mailto:igorsm1@insper.edu.br)

# Chamadas de sistema

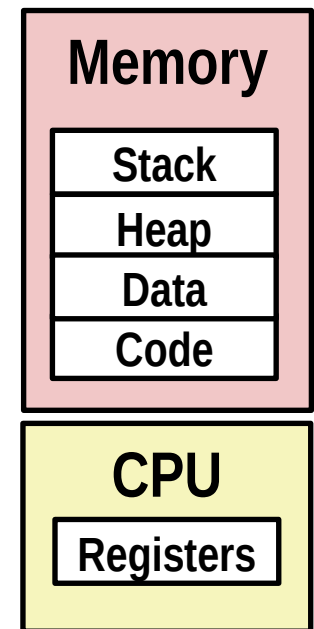


# POSIX - syscalls

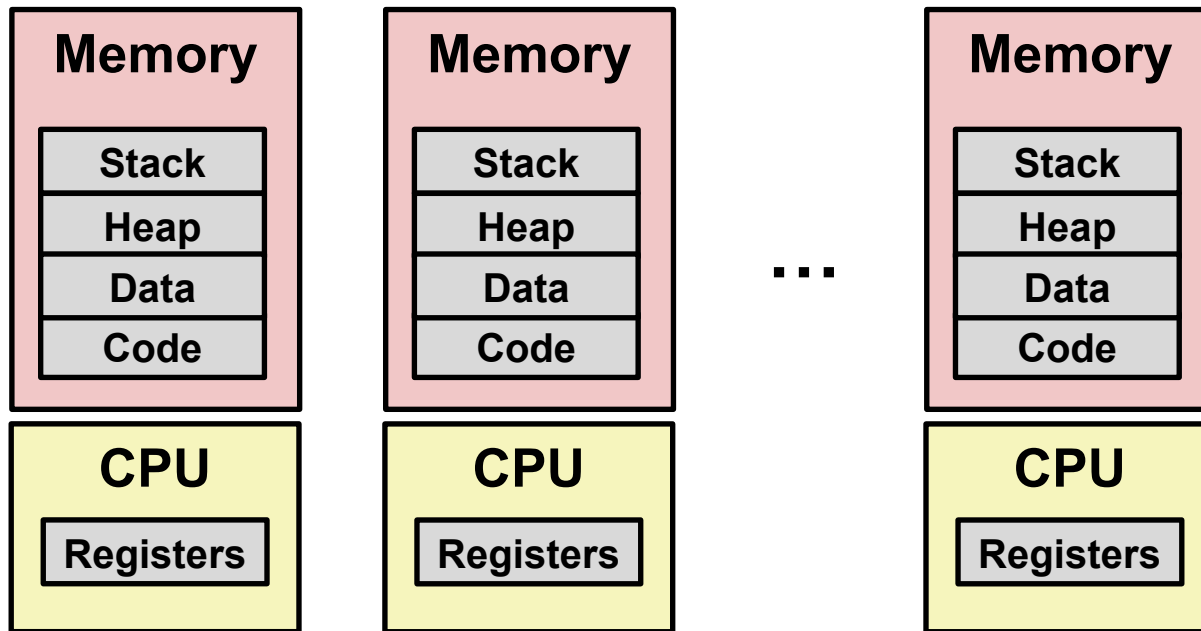
- Gerenciamento de usuários e grupos
- Manipulação de arquivos (incluindo permissões) e diretórios
- Criação de processos e carregamento de programas
- Comunicação entre processos
- Interação direta com hardware (via drivers)

# Processos

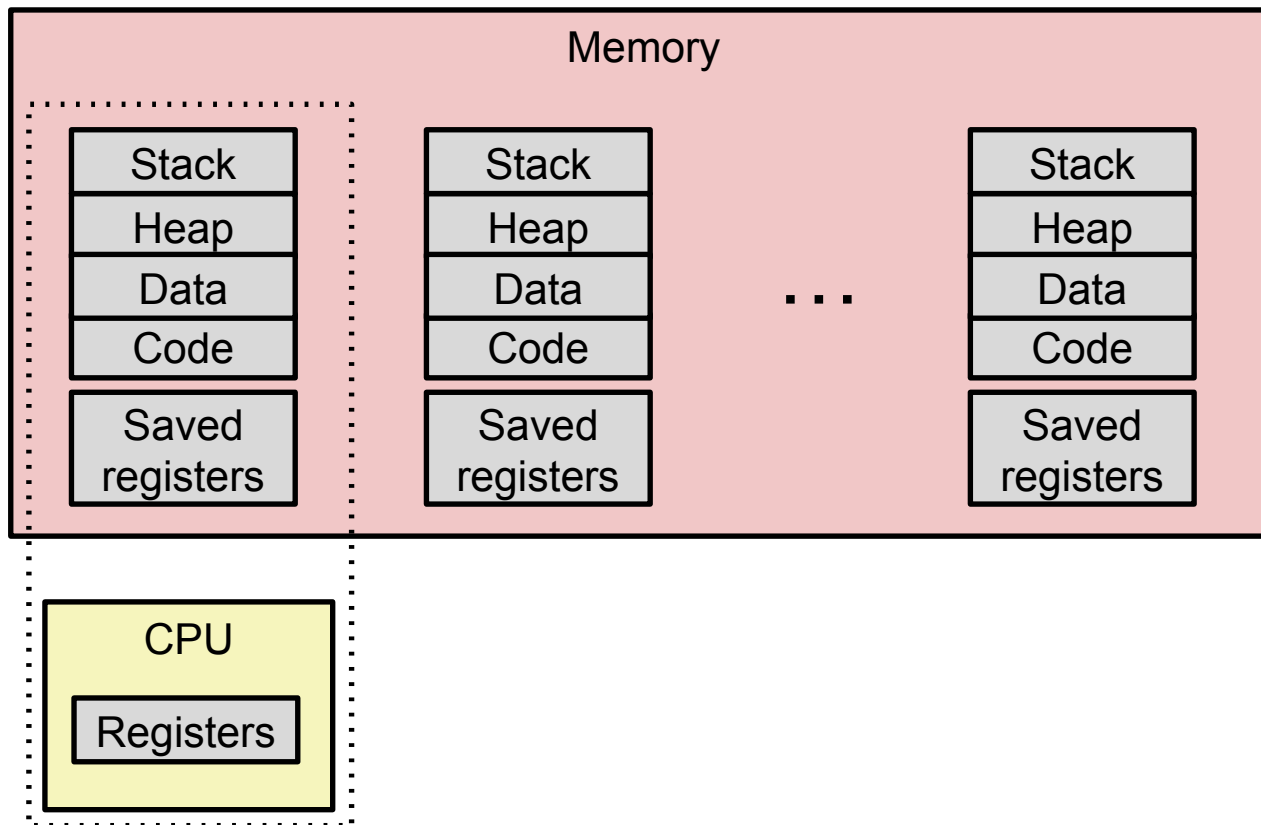
- Fluxo de controle lógico
  - Cada programa parece ter uso exclusivo da CPU
  - Provido pelo mecanismo de *chaveamento de contexto*
- Espaço de endereçamento privado
  - Cada programa parece ter uso exclusivo da memória principal
  - Provido pelo mecanismo de *memória virtual*



# A ilusão do multiprocessamento

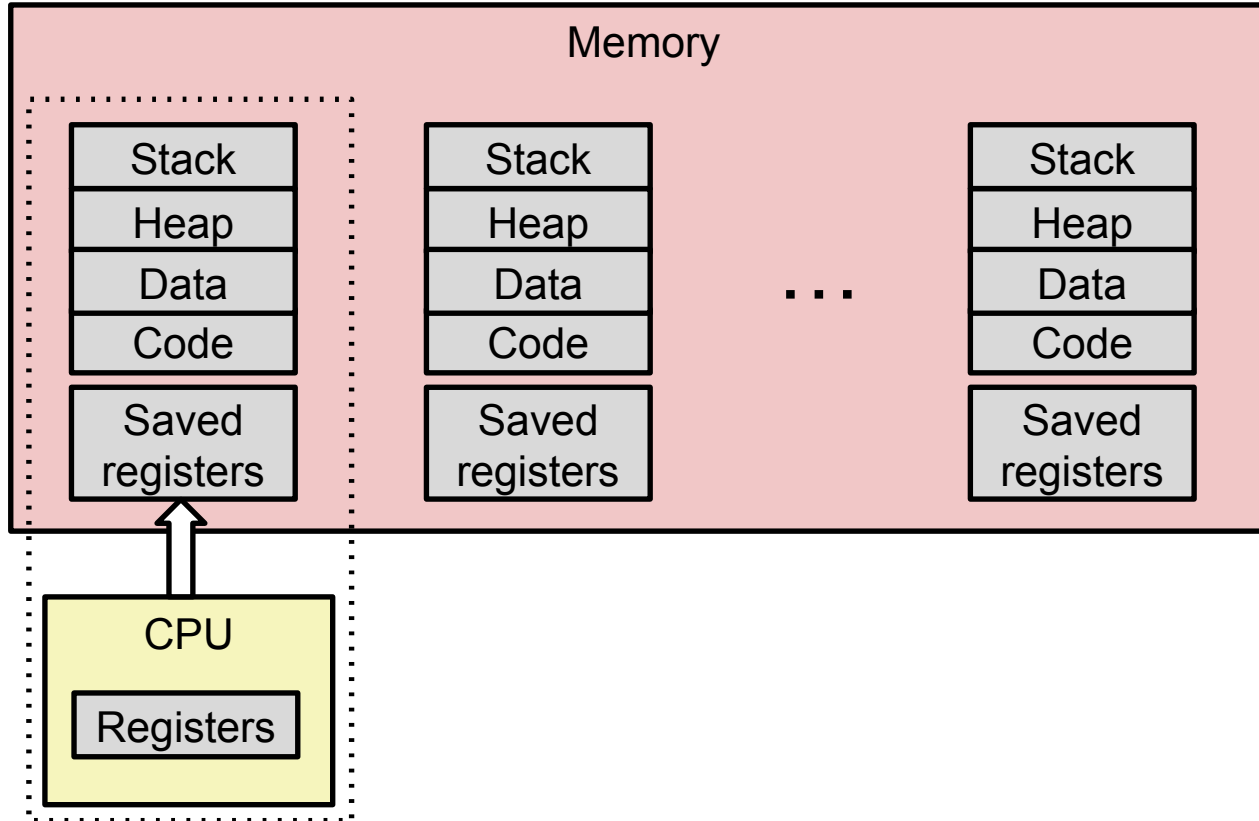


# A realidade do multiprocessamento



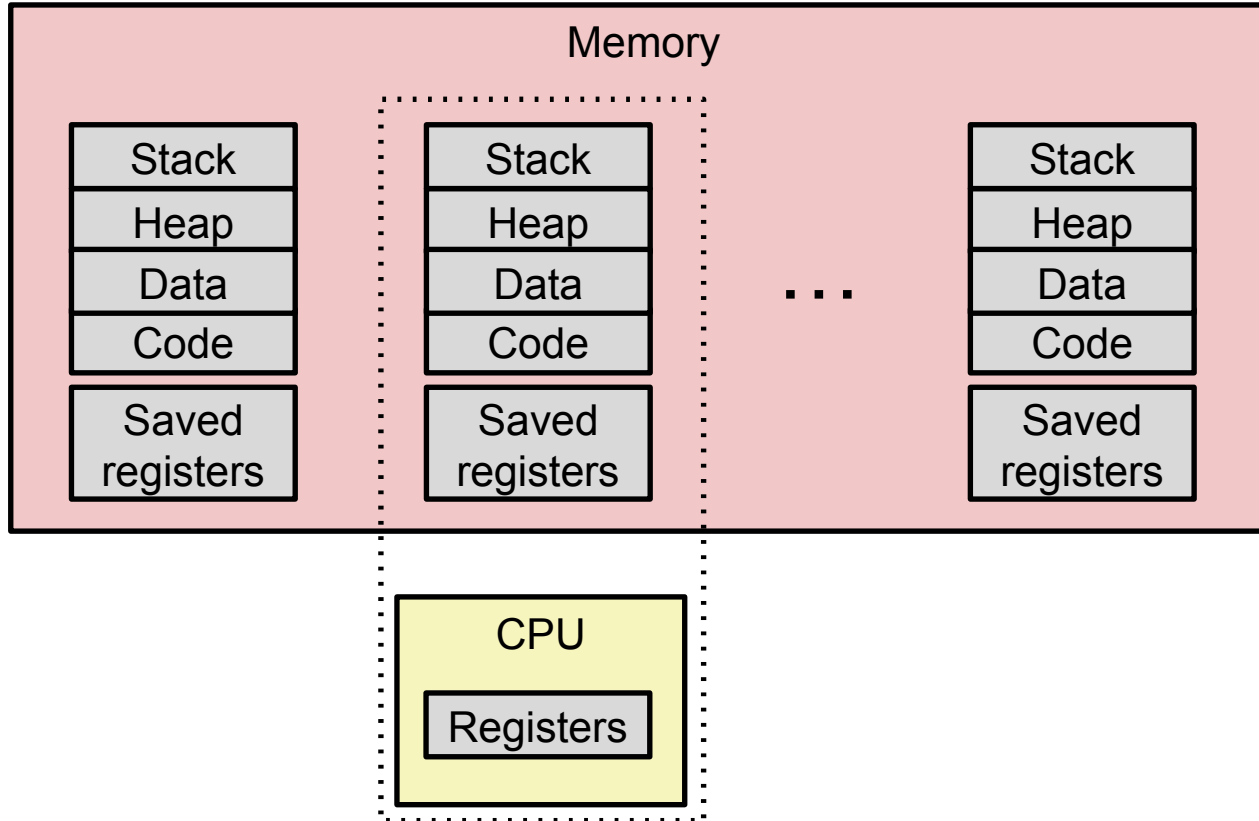
- Execução de processos intercalada
- Espaços de endereçamento gerenciados pelo sistema de memória virtual
- Valores de registradores para processos em espera são gravados em memória

# A realidade do multiprocessamento



- Grava registradores na memória

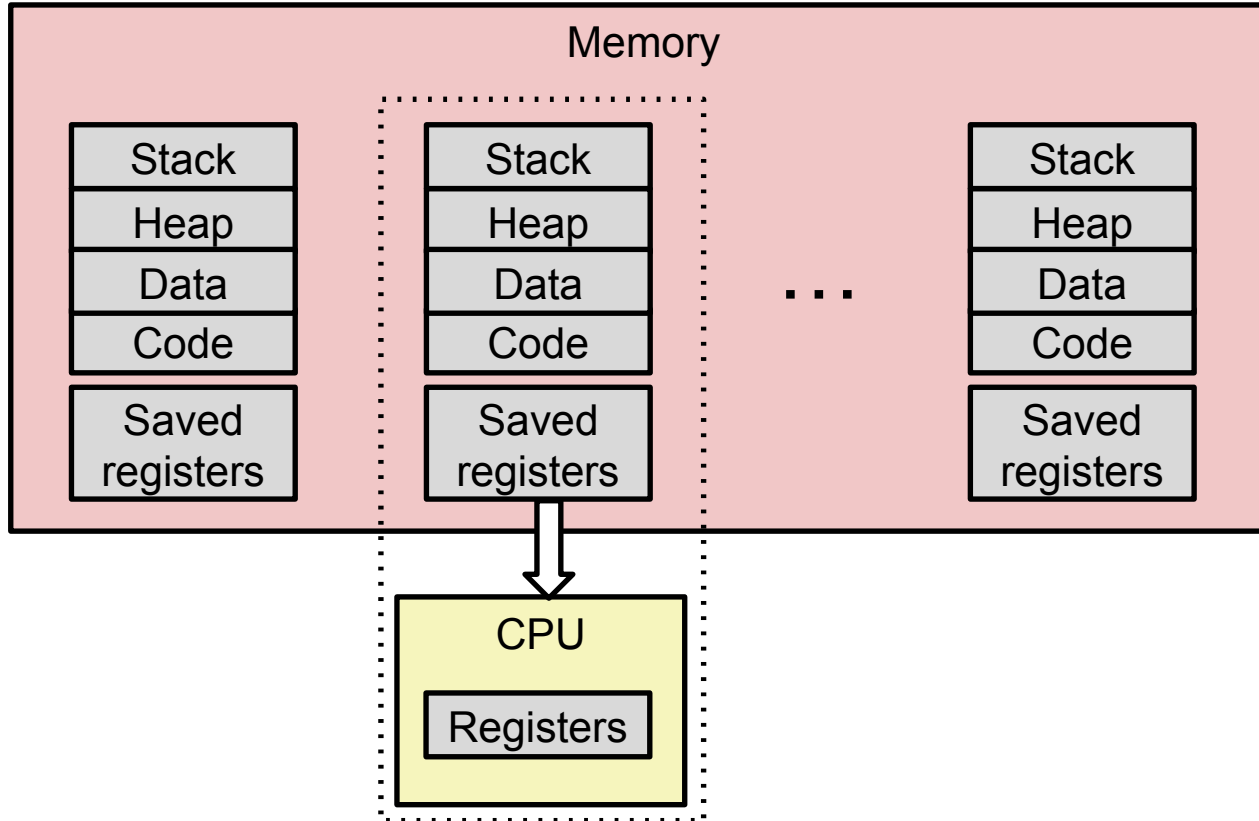
# A realidade do multiprocessamento



- Escolhe próximo processo a ser executado

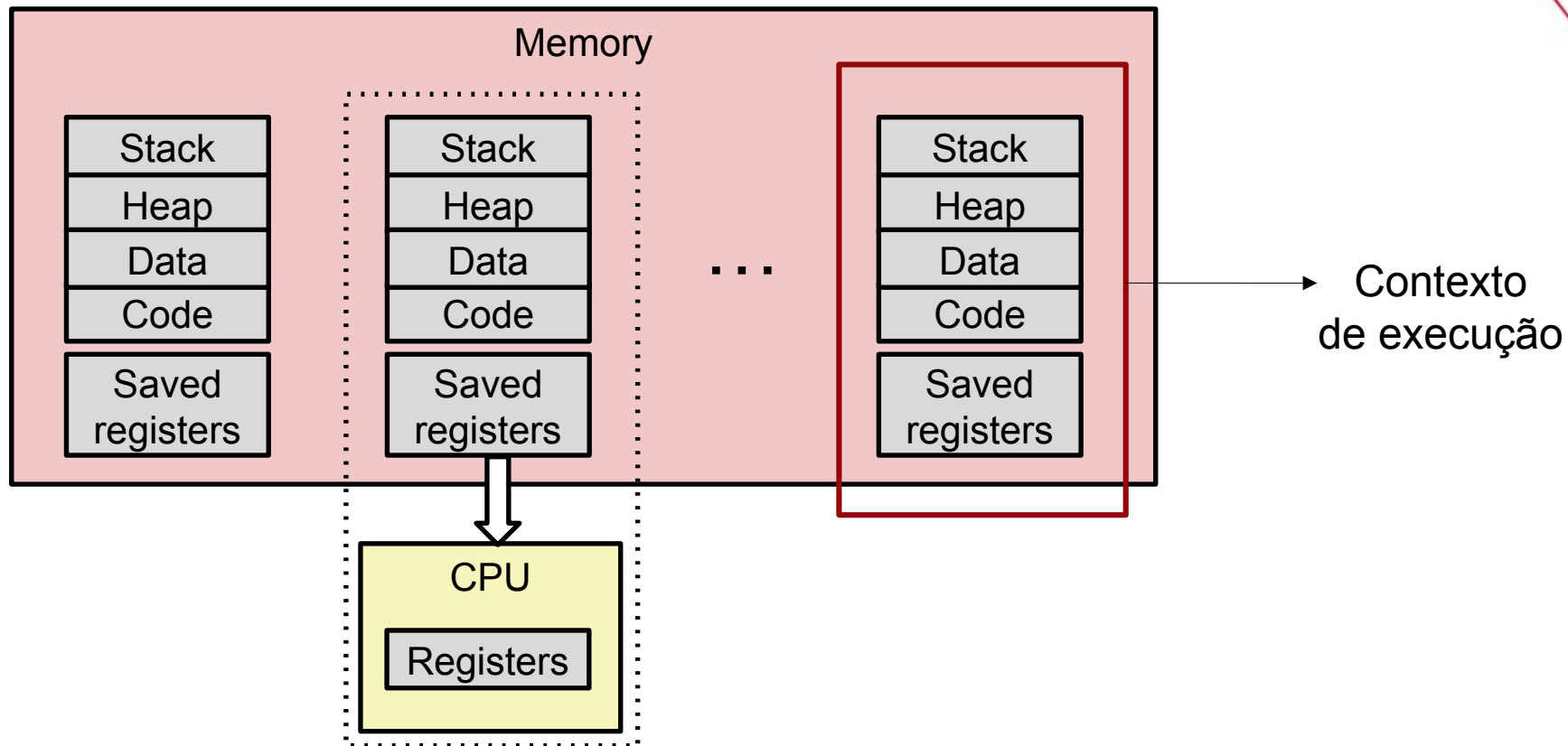


# A realidade do multiprocessamento



- 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

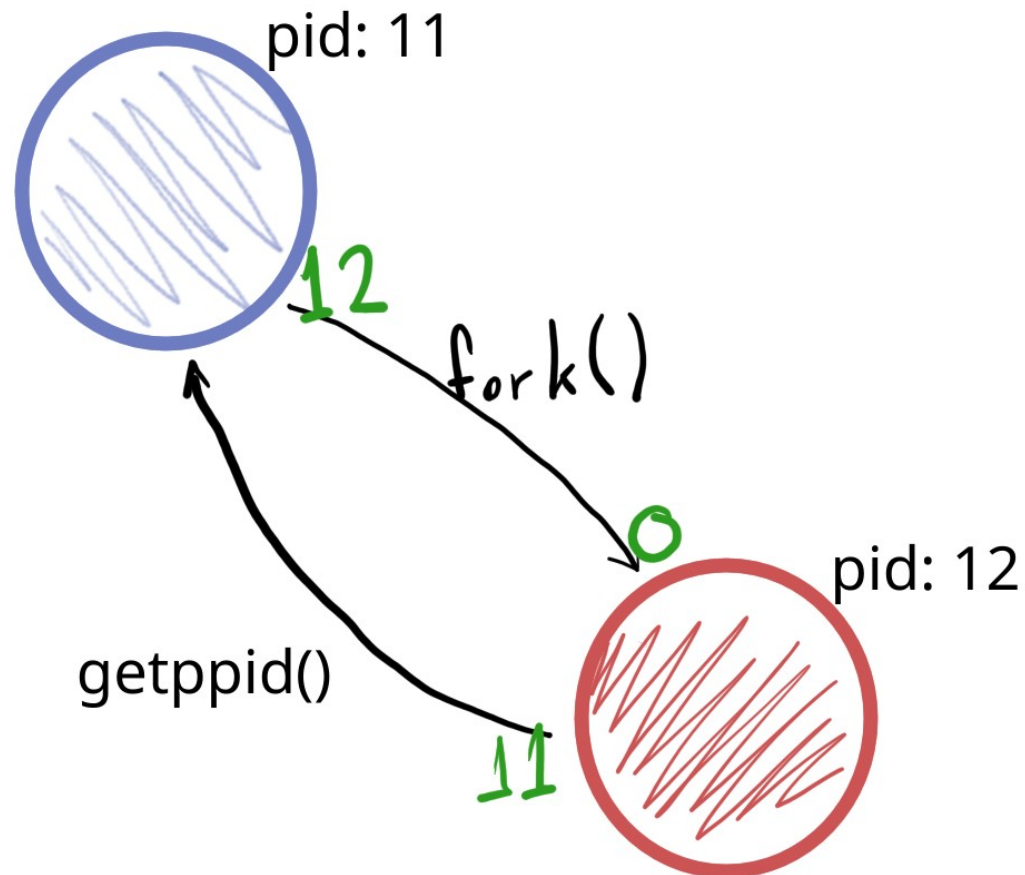
- Um processo pode esperar pelo fim de outro processo filho usando as funções

```
pid_t wait(int *wstatus);
```

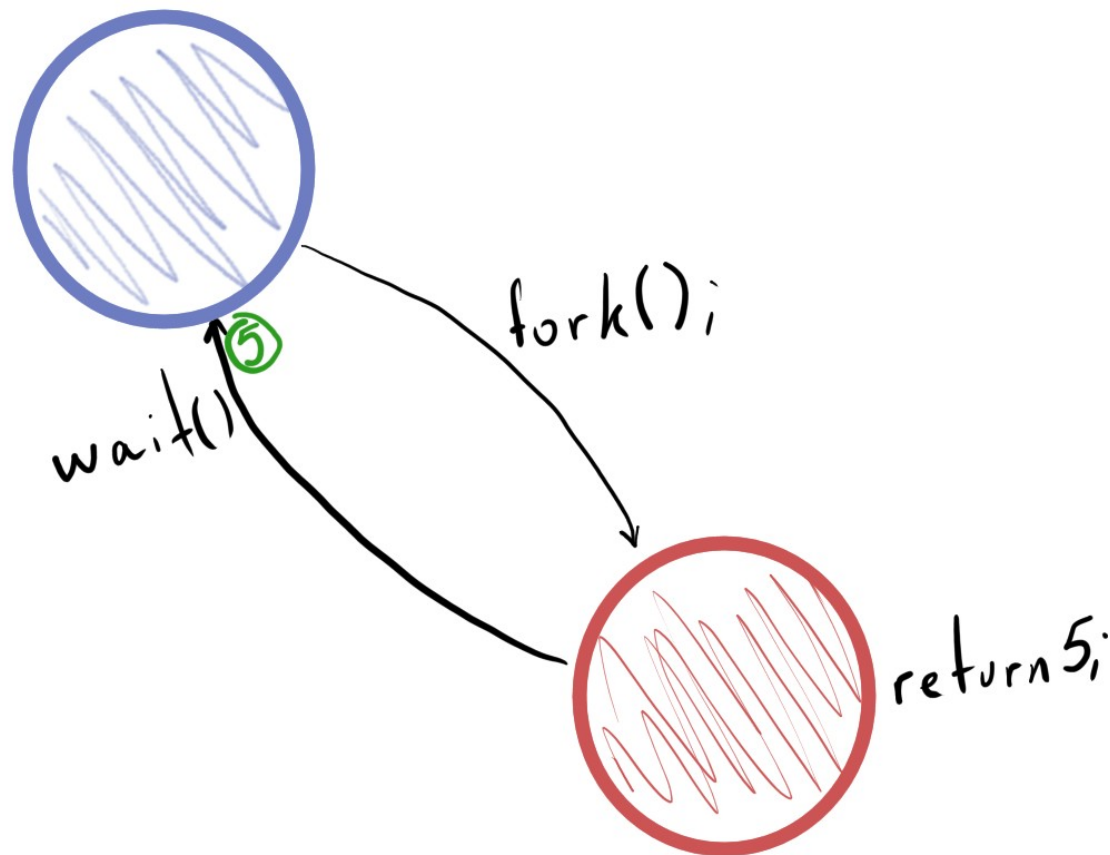
```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- 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 .

# Parentesco de processos



## Parentesco de processos – II



# Hoje

- Continuar explorando as relações de parentesco entre processos
- Criar novos processos usando exec

# Atividade

- Roteiro parte 0
- O programa `man` contém documentação sobre todas as funções usadas hoje. Use-o ;)
- O texto do manual por vezes é confuso ou muito técnico. Se você não entendeu ele me chame.



# Atividade

- Exemplo artificial usando fork e wait

# Parentesco de processos – III

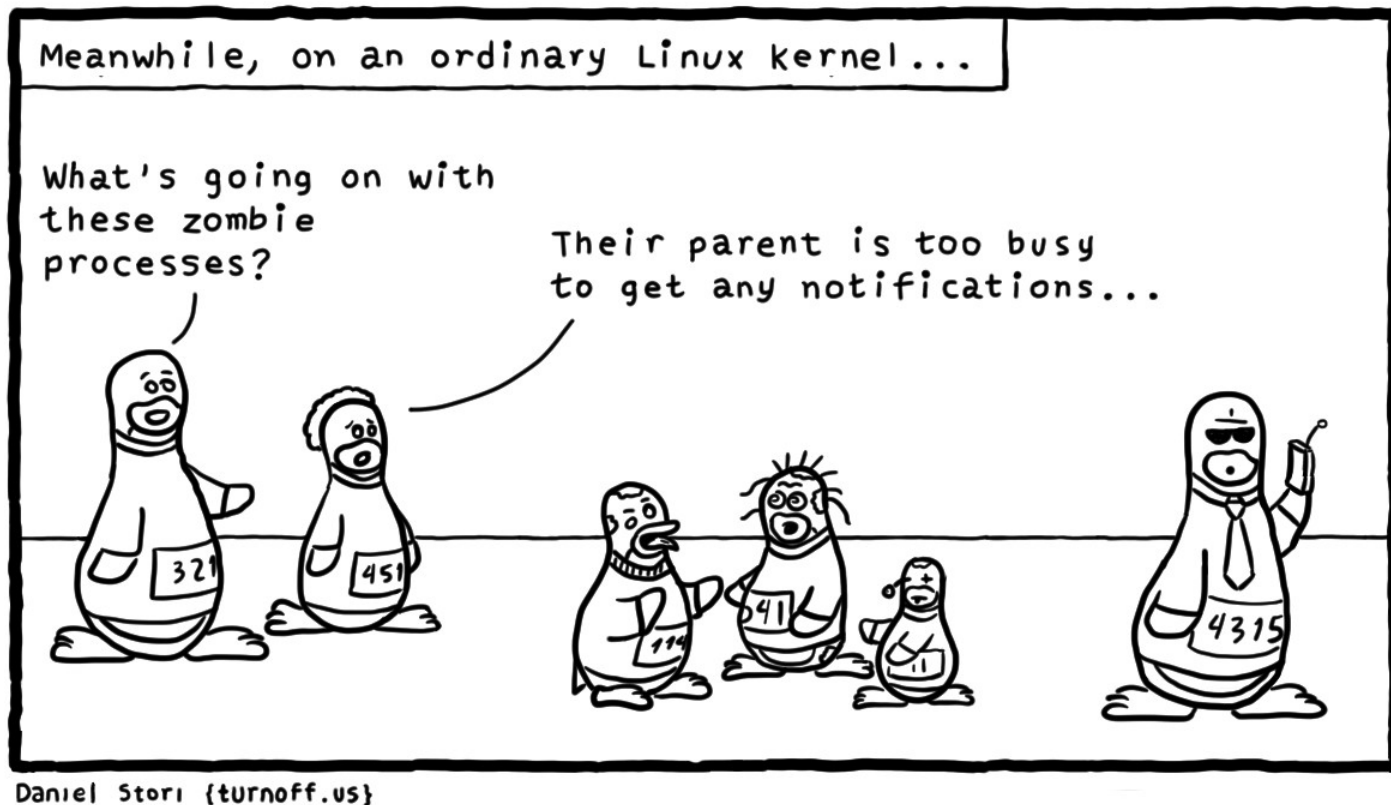
O que acontece se

1. O pai terminar antes do filho?
2. O filho terminar mas seu pai nunca chamar wait?

# Atividade

- Roteiro parte 1
- O programa `man` contém documentação sobre todas as funções usadas hoje. Use-o ;)
- O texto do manual por vezes é confuso ou muito técnico. Se você não entendeu ele me chame.

# Parentesco de processos – zumbis



<http://turnoff.us/geek/zombie-processes/>

# 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
- Funções da família exec nunca retornam: o programa atual é destruído durante o carregamento do novo programa
- Por que isso é separado?

# Exemplo de uso - argumentos

```
#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

```
#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 nunca roda!

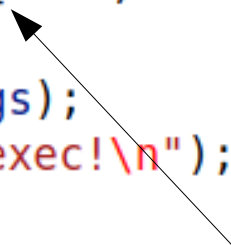
# Exemplo de uso - argumentos

```
#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!



# Exemplo de uso - argumentos

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

int main(int argc, char *argv[]) {
    for (int i = 0; i < argc; i++) {
        printf("arg: %s\n", argv[i]);
    }

    return 0;
}
```

# Atividade

- Parte 2
- O programa `man` contém documentação sobre todas as funções usadas hoje. Use-o ;)
- O texto do manual por vezes é confuso ou muito técnico. Se você não entendeu ele me chame.

# Próxima aula

Já vimos

- Arquivos e permissões
- Processos (criação e carregamento de programas)

Agora vamos juntar tudo isso em um sistema linux completo

- Compilar kernel, ferramentas de usuário
- Montar um sistema de arquivos raiz
- Configurar o processo `init` (`pid=1`)

# Insper

[www.insper.edu.br](http://www.insper.edu.br)