

# Gestão de Processos

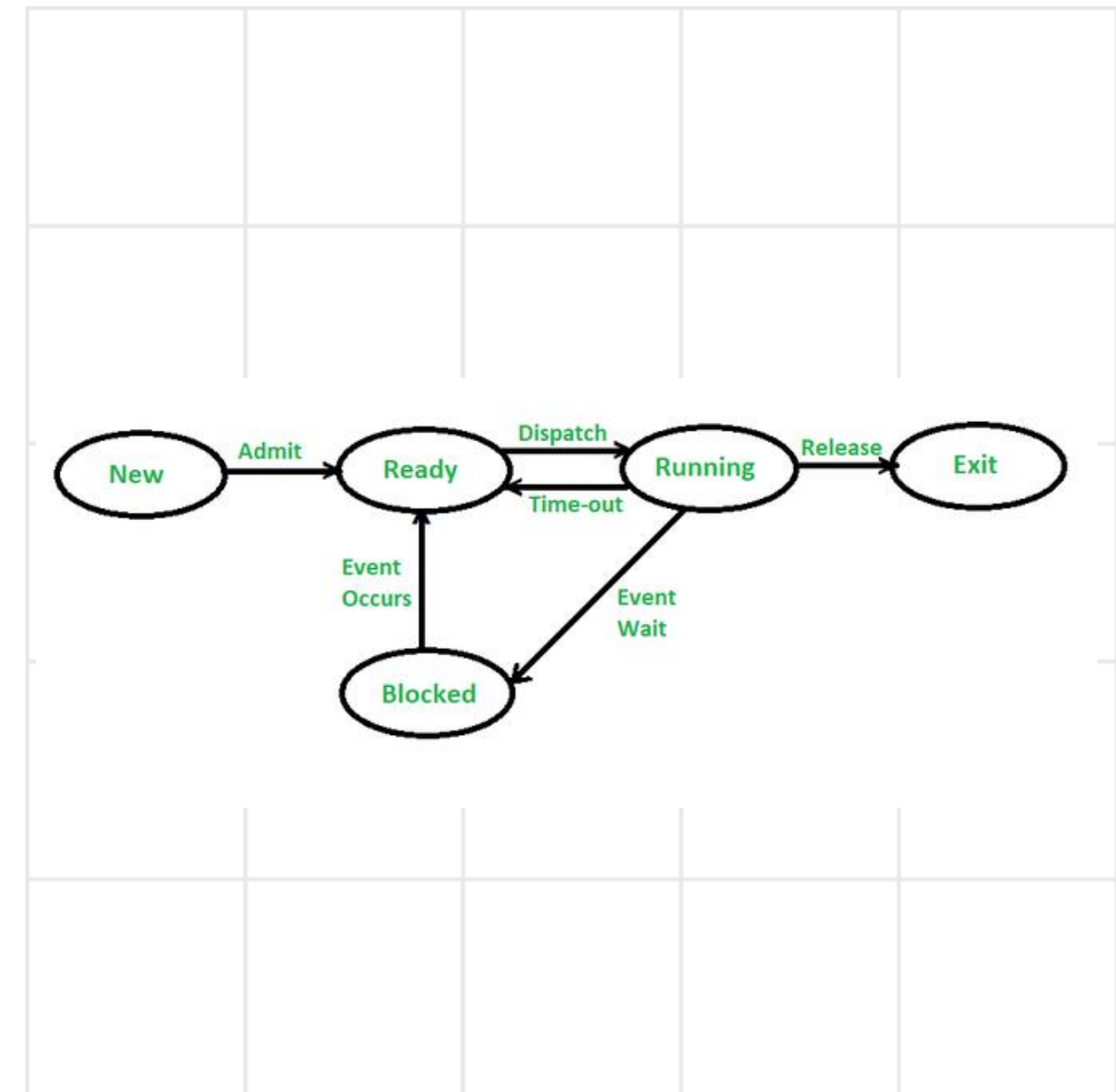
(usando a API do kernel)



# Bem-vindos à aula!

## Agenda de Hoje

- Conceitos de gestão de processos em C
- Utilizar a API do kernel em sistemas UNIX
- Resolver exercícios práticos



# Conceitos básicos



O que é um processo?

- Programa em execução.

API do Kernel

- Funções principais: `fork()`, `exec()`, `wait()`

# Process Control Block (PCB)

## Definição

- **Process Control Block (PCB):** É uma estrutura de dados no sistema operativo que contém todas as informações necessárias para gerenciar um processo. Cada processo no sistema tem um PCB associado a ele.

# Estrutura do PCB

O PCB armazena várias informações sobre o processo, incluindo:

## 1. Identificação do Processo:

**PID (Process ID):** Identificador único do processo.

**PPID (Parent Process ID):** Identificador do processo pai.

## 2. Estado do Processo:

**Estado:** Indica o estado atual do processo (pronto, em execução, bloqueado, etc.).

## 3. Contexto de CPU:

**Registradores:** Valores dos registradores da CPU quando o processo não está em execução.

**Contador de Programa (PC):** Endereço da próxima instrução a ser executada.

# Estrutura do PCB

## 4. Informações de Memória:

**Apontadores de Segmento:** Informações sobre os segmentos de código, dados e stack.

**Tabela de Páginas:** Usada em sistemas com memória virtual.

## 5. Informações de Ficheiros:

**Descritores de Ficheiros:** Lista de ficheiros abertos pelo processo.

## 6. Informações de Contabilidade:

**Tempo de CPU:** Tempo total de CPU utilizado pelo processo.

**Prioridade:** Prioridade do processo.

## 7. Informações de I/O:

**Dispositivos de I/O:** Dispositivos de entrada/saída associados ao processo.

# Funções do PCB

## **1. Gerenciamento de Processos:**

O PCB permite ao sistema operativo gerenciar e controlar os processos de forma eficiente.

## **2. Troca de Contexto:**

Durante uma troca de contexto, o estado do processo atual é salvo no PCB, e o estado do próximo processo a ser executado é carregado a partir do seu PCB.

## **3. Monitoramento e Controle:**

O PCB armazena informações que permitem ao sistema operativo monitorar o uso de recursos e controlar a execução dos processos.

# Exemplo de PCB

```
Process Control Block (PCB)
-----
| PID: 1234 |
| PPID: 5678 |
| Estado: Pronto |
| Registradores: |
| EAX: 0x00000001 |
| EBX: 0x00000002 |
| ... |
| PC: 0x00400000 |
| Segmentos de Memória: |
| Código: 0x00400000 |
| Dados: 0x00600000 |
| Stack: 0x00800000 |
| Descritores de Ficheiros: |
| File 1: /home/user/file.txt |
| File 2: /dev/null |
| Tempo de CPU: 120ms |
| Prioridade: 5 |
| Dispositivos de I/O: |
| Teclado, Monitor |
-----
```



# PCB - Resumo

- **PCB:** Estrutura de dados essencial para a gestão de processos.
- **Informações:** Contém todas as informações necessárias para gerenciar um processo.
- **Funções:** Facilita o gerenciamento de processos, troca de contexto e monitoramento de recursos.

# Processos Pai e Filho

## Processo Pai

- Definição:** Um processo pai é aquele que cria um ou mais processos filhos. Ele é responsável por iniciar a execução de um novo processo.
- Exemplo:** Quando você executa um programa no terminal, o shell (processo pai) pode criar novos processos (filhos) para executar comandos.

## Processo Filho

- Definição:** Um processo filho é um novo processo criado por um processo pai. Ele herda uma cópia do espaço de endereçamento do pai, incluindo variáveis e descritores de arquivos.
- Exemplo:** Se você usar o comando `fork()` em um programa, ele cria um processo filho que é uma cópia quase idêntica do processo pai.

# Exemplo 1 - Uso do `fork()`

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

int main(int argc, char* argv[]) {
    fork();
    fork();
    fork();
    exit(EXIT_SUCCESS);
}
```

**Quantos processos são criados? Por quê?**

# Exemplo 1 - Uso do `fork()`

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

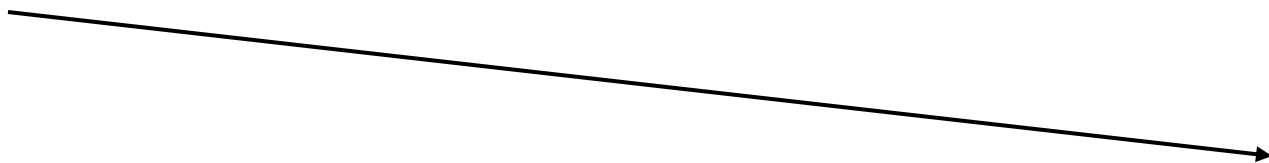
int main(int argc, char* argv[]) {
    fork();
    fork();
    fork();
    exit(EXIT_SUCCESS);
}
```

Estado Inicial:  
Processo pai

# Exemplo 1 - Uso do `fork()`

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

```
int main(int argc, char* argv[]) {
    fork();
    fork();
    fork();
    exit(EXIT_SUCCESS);
}
```



1º `fork()`:  
Processo Pai  
└─ Processo Filho 1

# Exemplo 1 - Uso do `fork()`

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

```
int main(int argc, char* argv[]) {
    fork();
    fork();
    fork();
    exit(EXIT_SUCCESS);
}
```

2º `fork()`:

Processo Pai

|— Processo Filho 2 (criado pelo Pai)

└— Processo Filho 1

    └— Processo Filho 3 (criado pelo Filho 1)

# Exemplo 1 - Uso do `fork()`

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

```
int main(int argc, char* argv[]) {
    fork();
    fork();
    fork();
    exit(EXIT_SUCCESS);
}
```

3º `fork()`:

Processo Pai

|— Processo Filho 4 (criado pelo Pai)

|— Processo Filho 2

|   |— Processo Filho 5 (criado pelo Filho 2)

|— Processo Filho 1

    |— Processo Filho 6 (criado pelo Filho 1)

    |— Processo Filho 3

        |— Processo Filho 7 (criado pelo Filho 3)

# Exemplo 1 - Uso do `fork()`

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

int main(int argc, char* argv[]) {
    fork();
    fork();
    fork();
    exit(EXIT_SUCCESS);
}
```

## Diagrama Geral para $n=3$

### 1. Pai

- |— 2. Filho 1 (criado pelo Pai)
- |— 3. Filho 2 (criado pelo Pai)
  - |— 4. Filho 3 (criado pelo Filho 2)
  - └— 5. Filho 4 (criado pelo Filho 2)
- |— 6. Filho 5 (criado pelo Filho 1)
- |— 7. Filho 6 (criado pelo Filho 1)
- └— 8. Filho 7 (criado pelo Filho 1)



# Demonstração Matemática

Seja  $P(n)$  o número total de processos após  $n$  chamadas ao `fork()`:

1. Inicialmente ( $n = 0$ ):

- Há apenas 1 processo (o processo pai original):  $P(0) = 1$

2. Após a primeira chamada ao `fork()` ( $n = 1$ ):

- O processo pai cria um processo filho:  $P(1) = 2$

3. Após a segunda chamada ao `fork()` ( $n = 2$ ):

- Cada processo (os dois existentes) cria um novo processo filho:

$$P(2) = 2 \times 2 = 4$$

4. Após a terceira chamada ao `fork()` ( $n = 3$ ):

- Cada processo (os 4 existentes) cria um novo processo filho:

$$P(3) = 2 \times 4 = 8$$

Generalizando:

$$P(n) = 2^n$$

# Exemplo 1 - Uso do `fork()`

## Resumo

- **Processo Pai:** Cria novos processos.
- **Processo Filho:** É criado pelo processo pai e herda seu espaço de endereçamento.
- **`fork()`:** Função usada para criar processos filhos, retornando diferentes valores no pai e no filho para permitir a diferenciação entre eles.
- **Valor de retorno:**
  - **Erro:** Se `fork()` falhar, ele retorna -1 no processo pai.
  - **No processo pai:** `fork()` retorna o PID (Process ID) do processo filho.
  - **No processo filho:** `fork()` retorna 0.
- **Execução Independente:** Cada processo executa de forma independente e chama `exit(EXIT_SUCCESS);` para terminar.
- **Ordem de Terminação:** A ordem de terminação dos processos não é garantida e pode variar dependendo do agendamento do sistema operativo. Pequenas variações no tempo de execução

## Exemplo 2 - Loop com fork()

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

int main(int argc, char* argv[]) {
    for (int i = 0; i < 4; i++)
        fork();
    exit(EXIT_SUCCESS);
}
```

**Quantos processos são criados? Por quê?**

## Exemplo 3 - Uso do `wait()`

A função `wait()` é usada em programação de sistemas UNIX para fazer com que um processo pai espere até que um ou mais de seus processos filhos terminem.

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

### Como Funciona

- **Espera:** Quando um processo pai chama `wait()`, ele é suspenso até que um de seus processos filhos termine.
- **Status:** O argumento `status` é um apontador para uma variável onde o status de término do processo filho será armazenado. Se `status` não for `NULL`, a função `wait()` armazena informações sobre como o processo filho terminou.
- **Valor de Retorno:** Retorna o `PID` do processo filho que terminou. Em caso de erro, retorna `-1`.

## Exemplo 3 – Uso do `wait()`

```
int main(int argc, char* argv[]) {  
    int value = 0;  
    pid_t pid = fork();  
    if (pid == 0) {  
        value = 1;  
        printf("CHILD: value = %d, addr = %p\n", value, &value);  
        exit(EXIT_SUCCESS);  
    } else {  
        wait(NULL);  
        printf("PARENT: value = %d, addr = %p\n", value, &value);  
        exit(EXIT_SUCCESS);  
    }  
}
```

**`wait(NULL)`** : Faz com que o processo pai espere até que o seu processo filho termine, sem se preocupar com o status de término.

# Exemplo 3 - Uso do `wait()`

E se não usar `wait()`:

## 1. Processos Zumbis

- **Definição:** Um processo zumbi é um processo filho que terminou sua execução, mas ainda tem uma entrada na tabela de processos do sistema porque o processo pai não chamou `wait()` para ler seu status de término.
- **Consequência:** Processos zumbis ocupam entradas na tabela de processos, o que pode eventualmente levar ao esgotamento de recursos do sistema se muitos processos zumbis se acumularem.

## 2. Recursos Não Liberados

- **Memória e Recursos:** O sistema operativo mantém informações sobre o processo filho (como o status de término) até que o processo pai chame `wait()`. Se `wait()` não for chamado, esses recursos não são liberados adequadamente.

## 3. Comportamento Indefinido

- **Sincronização:** Sem `wait()`, o processo pai pode continuar sua execução sem saber quando os processos filhos terminam. Isso pode levar a comportamentos inesperados, especialmente se o pai depende dos resultados dos filhos.

# Espaço de Endereçamento de um Processo

O espaço de endereçamento de um processo é uma organização lógica da memória que permite ao processo ter uma visão independente e isolada do sistema, graças ao mecanismo de endereços **virtuais**. Esta organização é crucial para entender operações como `fork()` e `exec()`.

# Partes do Espaço de Endereçamento

## Parte Estática (Definida pelo Binário Executável)

- **.text**: Contém o código do programa. É somente leitura para evitar modificações no código em execução.
- **.data**: Contém variáveis globais e estáticas **inicializadas**.

Exemplo: `int x = 5;`

- **.bss**: Contém variáveis globais e estáticas **não inicializadas**.

Exemplo: `int y;`

## Parte Dinâmica (Usada em Tempo de Execução)

- **Heap**: Memória dinâmica alocada durante a execução via funções como `malloc()` e `free()`. Cresce em direção aos endereços mais altos.
- **Stack**: Memória usada para chamadas de função, armazenamento de variáveis locais e retorno de funções. Cresce em direção aos endereços mais baixos.



# Partes do Espaço de Endereçamento

Endereço Mais Alto

+-----+

Kernel Space	<- Reservado para o Sistema Operativo
--------------	---------------------------------------

+-----+

Stack	<- Variáveis Locais e Chamadas de Função
-------	--

--	--

+-----+

Heap	<- Memória Dinâmica
------	---------------------

--	--

+-----+

.bss	<- Variáveis Não Inicializadas
------	--------------------------------

+-----+

.data	<- Variáveis Inicializadas
-------	----------------------------

+-----+

.text	<- Código do Programa (Somente Leitura)
-------	---

+-----+

Endereço Mais Baixo

# Endereços Virtuais e Tradução

## Endereços Virtuais:

- Os programas geram endereços **virtuais** (não reais), que são independentes do hardware físico.

## Tradução para Endereços Físicos:

- **MMU (Memory Management Unit)**: Converte endereços virtuais em endereços físicos.
- O sistema operativo utiliza **páginas de memória** para mapear partes do espaço virtual em memória física.
- Cada processo tem sua própria tabela de mapeamento, garantindo isolamento entre processos.

# Relação com `fork()`

Quando um processo chama `fork()`:

- O espaço de endereçamento do processo filho é uma **cópia exata** do espaço do processo pai (graças ao mecanismo de **Copy-On-Write**).
- Ambos compartilham as mesmas páginas de código (.text), enquanto outras regiões (heap, stack) são copiadas apenas quando modificadas.

# Relação com `exec()`

Quando o processo chama `exec()`:

- O espaço de endereçamento é **substituído** por um novo, baseado no programa especificado.
- O `.text`, `.data`, e `.bss` são carregados do novo binário, e o heap e stack são reinicializados.

## Exemplo 4 - Variável Compartilhada

```
int main(int argc, char* argv[]) {  
    int value = 0;  
    pid_t pid = fork();  
    if (pid == 0) {  
        value = 1;  
        printf("CHILD: value = %d, addr = %p\n", value, &value);  
        exit(EXIT_SUCCESS);  
    } else {  
        wait(NULL);  
        printf("PARENT: value = %d, addr = %p\n", value, &value);  
        exit(EXIT_SUCCESS);  
    }  
}
```

Como explicar o valor da variável value?

# Antes da execução do `fork()`

+-----+	
Código (text)	<- Código do programa executado
+-----+	
Dados estáticos	<- Variáveis globais e estáticas inicializadas
+-----+	
Heap	<- Memória alocada dinamicamente (malloc, etc.)
+-----+	
(Espaço vazio)	
+-----+	
Pilha (stack)	<- Funções, variáveis locais, e chamadas ativas
+-----+	

# Execução do fork ()

## Processo Pai (PID:1001)

Código (text)	
Dados estáticos	
Heap	
Pilha (stack)	
Endereço de retorno	
argc	
argv	
value = 0	
pid	



## Processo Filho (PID: 1002)

Código (text)	
Dados estáticos	
Heap	
Pilha (stack)	
Endereço de retorno	
argc	
argv	
value = 0	
pid	

Somente leitura (read-only)

# Depois da Execução do `fork()`

## Processo Pai

Código (text)
Dados estáticos
Heap
Pilha (stack)
Endereço de retorno
argc
argv
value = 0
pid

## Processo Filho

Código (text)
Dados estáticos
Heap
Pilha (stack)
Endereço de retorno
argc
argv
value = 1
pid

```
if (pid == 0)
    value = 1;
```

Copy on Write (COW)



# Conceito de Copy on Write

**Copy on Write (COW):** É uma técnica onde a cópia real dos dados na memória só é feita quando uma das cópias tenta modificar os dados.

## Funcionamento do Copy on Write

### 1. Criação do Processo Filho:

- Quando um processo chama `fork()`, o sistema operativo cria um processo filho.
- Em vez de copiar imediatamente todo o espaço de endereçamento do processo pai para o filho, o sistema marca as páginas de memória como **compartilhadas** e **somente leitura**.

### 2. Compartilhamento Inicial:

- Tanto o processo pai quanto o processo filho compartilham as mesmas páginas de memória.
- Nenhuma cópia real dos dados é feita neste momento, economizando memória e tempo.

### 3. Modificação dos Dados:

- Quando o processo pai ou o processo filho tenta modificar uma página de memória compartilhada, ocorre uma **interrupção de página** (page fault).
- O sistema operacional então cria uma **cópia** da página de memória para o processo que está tentando modificar os dados.
- A página copiada é marcada como **escrita** para o processo que fez a modificação, enquanto o outro processo continua a acessar a página original.

# Resumo

- O espaço de endereçamento de um processo é organizado em regiões distintas (estáticas e dinâmicas).
- Os endereços que vemos são virtuais, traduzidos pela MMU para endereços físicos.
- Operações como `fork()` e `exec()` interagem com essas estruturas de forma fundamental para criar e modificar processos.
- Copy on Write: Técnica que adia a cópia de dados na memória até que uma modificação seja necessária.

## Exemplo 5 - Execução de Comando

```
int main(int argc, char* argv[]) {  
    pid_t pid = fork();  
    if (pid == 0) {  
        execlp(argv[1], argv[1], NULL);  
        exit(EXIT_FAILURE);  
    } else {  
        wait(NULL);  
    } exit(EXIT_SUCCESS);  
}
```

**Como o processo filho sinaliza seu término ao processo pai?**

# Exemplo 5 - Execução de Comando

Processo Pai:

+-----+
Código (text)
+-----+
Dados estáticos
+-----+
Heap
+-----+
Pilha (stack)
+-----+

Processo Filho (depois do execvp):

+-----+
Código (text)
+-----+
Dados estáticos
+-----+
Heap
+-----+
Pilha (stack)
+-----+

<- Agora, o código foi substituído pelo código do novo programa.

<- O segmento de dados estáticos também foi substituído.

<- A alocação do heap pode ser modificada conforme o novo programa.

<- A pilha é substituída pelo novo programa.



# Tente e Aprenda

Hora da Atividade

Ficha 5: Exercícios 1 - 4



**E por hoje  
terminamos!**