Sistemas Operacionais

Multitarefa: Processos e Threads

Pontifícia Universidade Católica de Campinas

Prof. Dr. Denis M. L. Martins

Objetivos de Aprendizado

- Explicar conceitos e relações entre programa, processo, thread e multitarefa.
- Compreender a diferenças entre threads e processos.
- Entender os estados de processos/threads.

Disclaimer

Parte do material apresentado a seguir foi adaptado de:

- IT Systems – Open Educational Resource, produzido por Jens~Lechtenböger; e
- Open Education Hub - Operating Systems

Imagens decorativas retiradas de Unsplash

Multitarefa

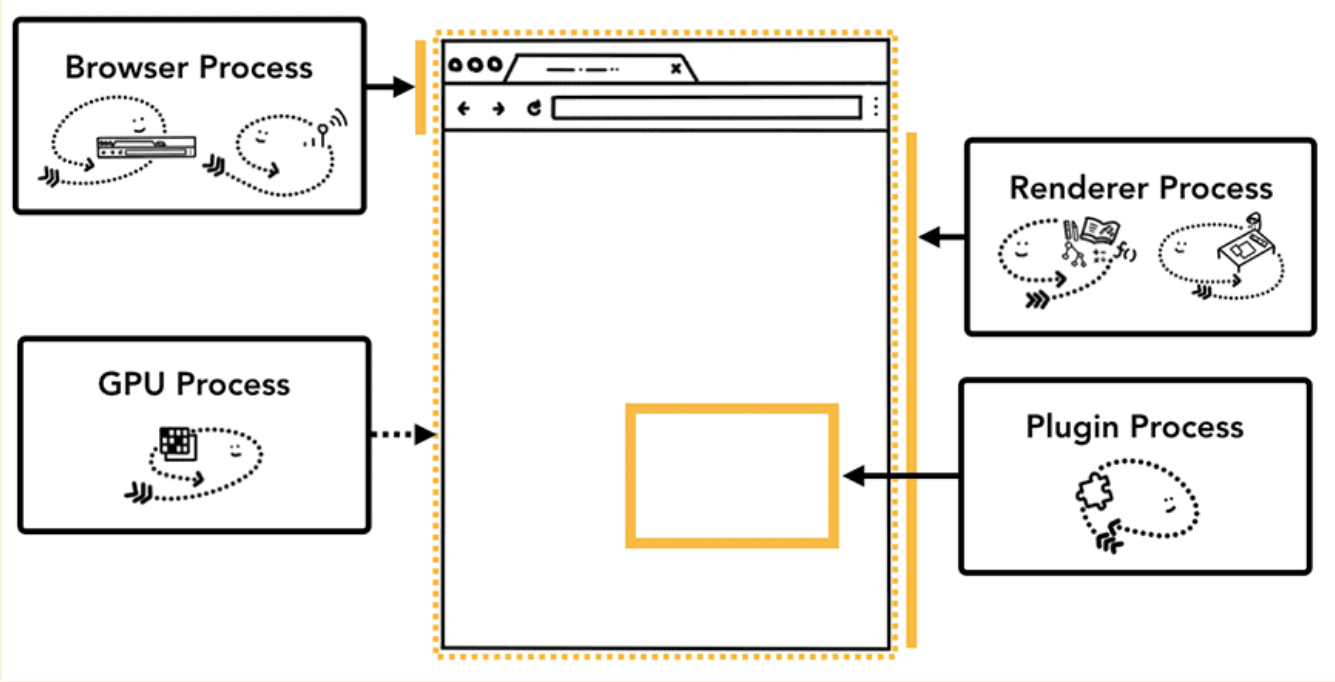
- Os sistemas operacionais permitem que múltiplas computações ocorram **concorrentemente** em um único sistema computacional.
  - Um exemplo de **multitarefa** ocorre quando você está **ouvindo música no Spotify** enquanto **navega na internet e clica em links**.
  - O **sistema operacional** gerencia 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
- 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.

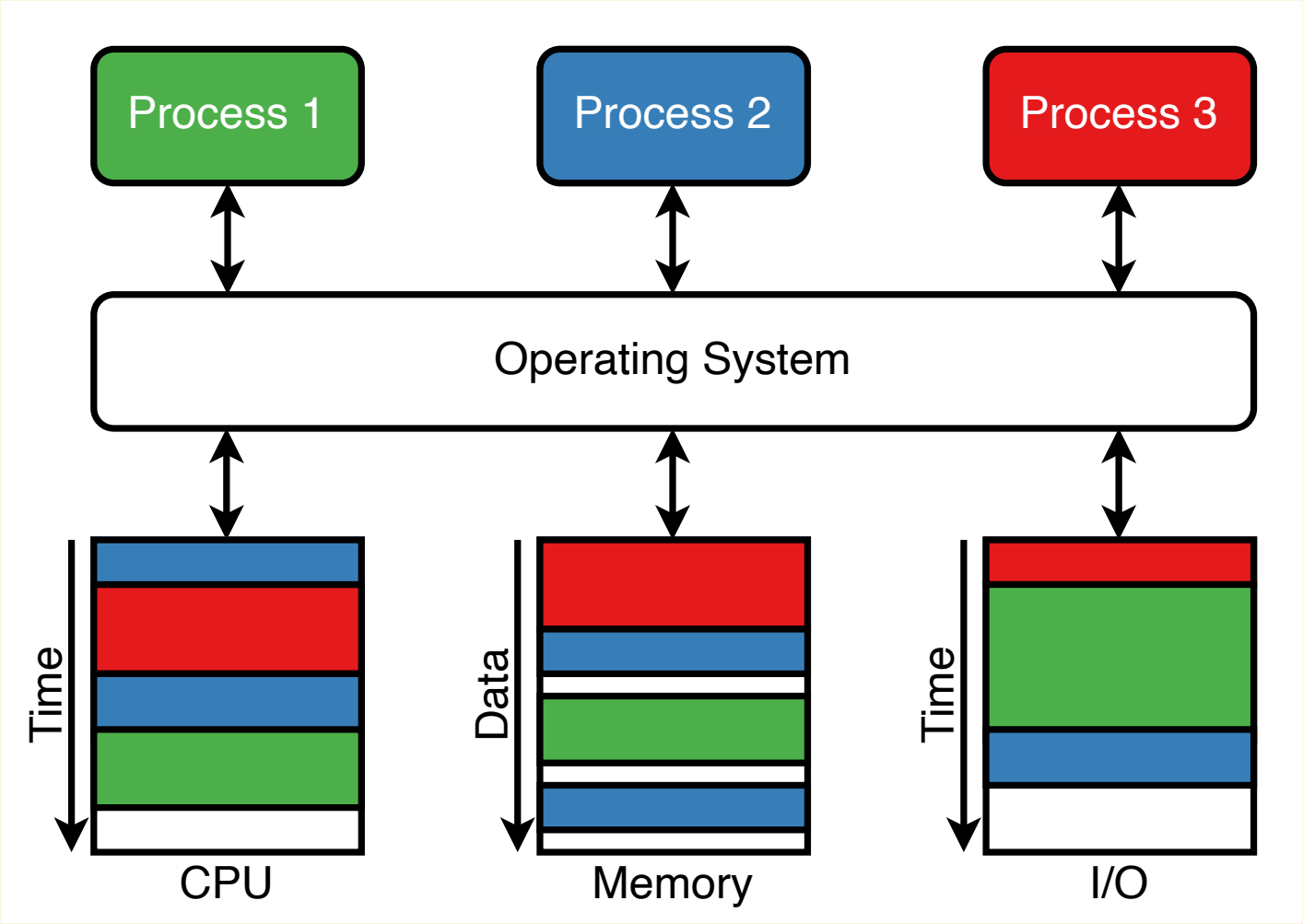
Processos em um SO

- Processo ≈ programa em execução.
  - **Programa**: entidade **passiva** guardada no disco (**arquivo executável**).
- Processo como unidade de **gerenciamento** e **proteção**.
  - Um programa pode criar múltiplos processos.



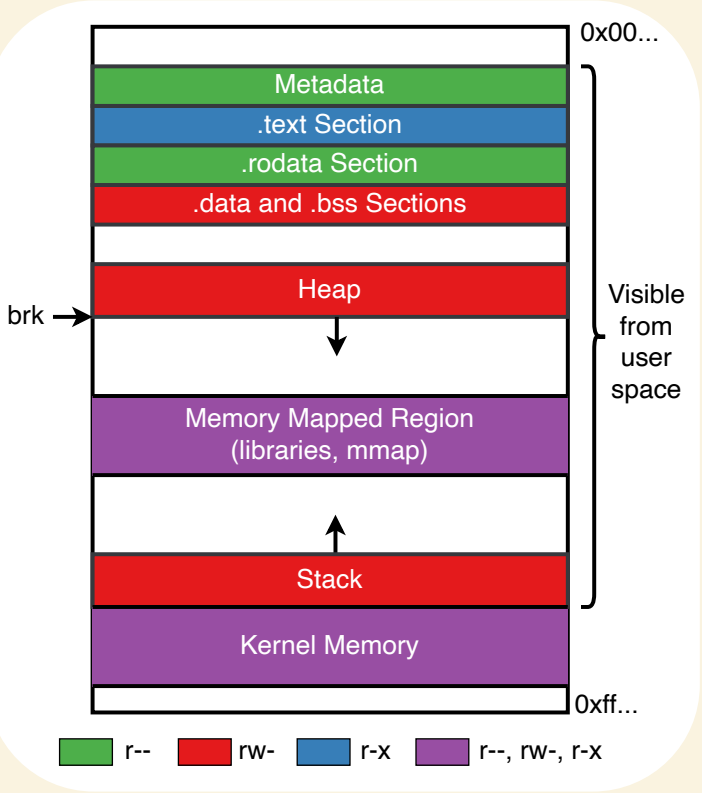
Na imagem: Processos apontando para diferentes partes da interface do usuário (UI) do navegado. Fonte: Google Developers

SO gerencia recursos para processos



Na Imagem: SO gerenciando memória, CPU e I/O para três processos diferentes. Fonte: OER OS

Bloco de Controle de Processo

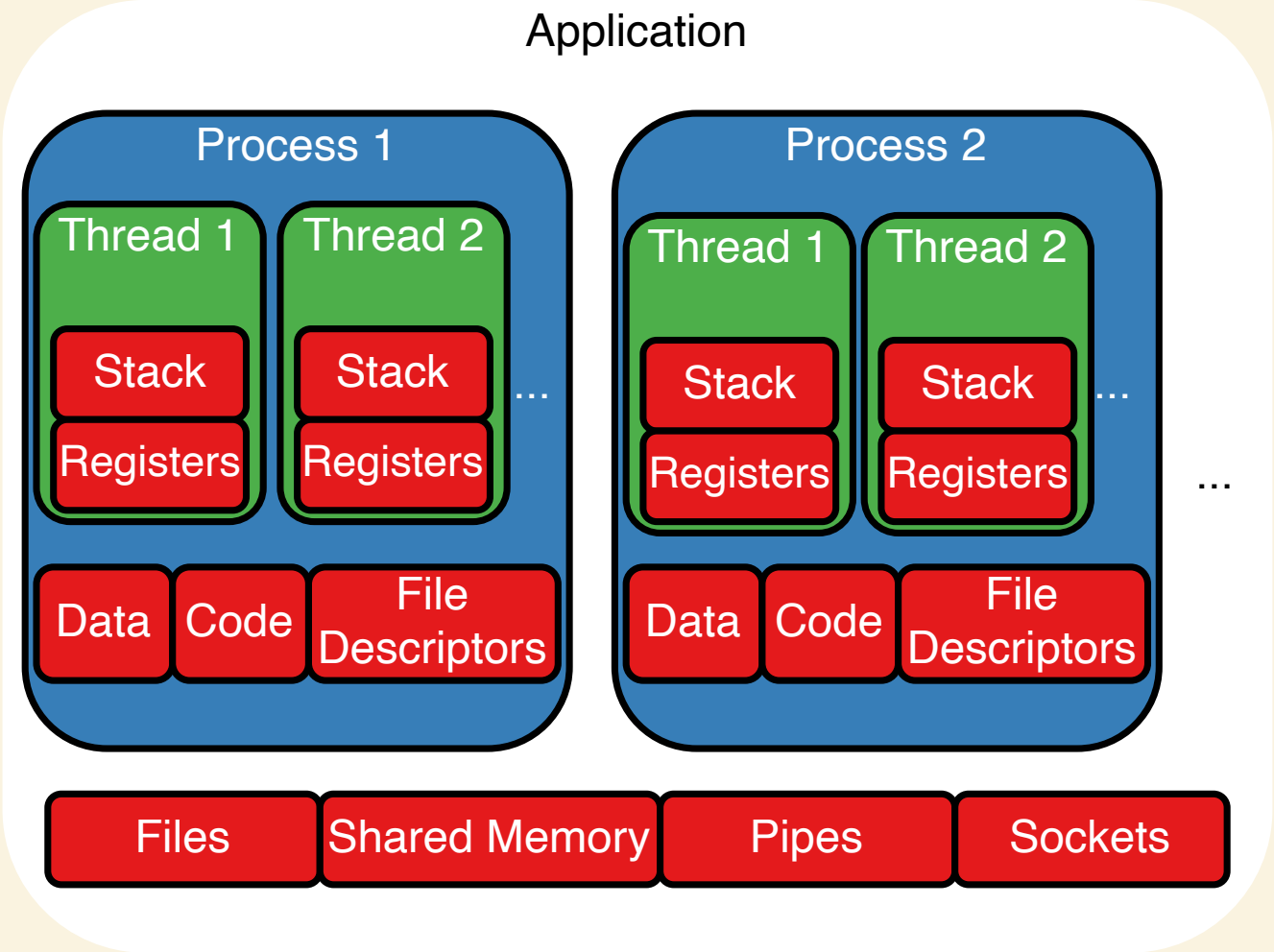


- **Estado do processo**: em execução, em espera, etc.
- **Contador de programa** (PC): endereço da próxima instrução.
- **Registradores da CPU**: conteúdo de todos os registradores utilizados por processos.
- **Informação de gerenciamento de memória**: memória alocada para o processo.
- **Estatísticas**: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

Threads

- Thread: unidade de escalonamento do SO. Sequência independente de computações. São mais leves e mais fáceis de criar e destruir do que processos.
  - Um mesmo programa (processo) pode realizar várias tarefas concorrentemente.
  - Processador de texto (e.g. MS Word): processa texto do teclado, verifica ortografia, salva continuamente o documento → 3 threads num mesmo processo.
- Alto grau de independência: pense em funções diferentes no código.
- Núcleos individuais não estão se tornando significativamente mais rápidos. Relembre a Lei de Moore. Threads permitesm aproveitar o hardware atual.

Threads



Na Imagem: Applicação com dois processos, cada processo com múltiplas threads. Fonte: OER OS

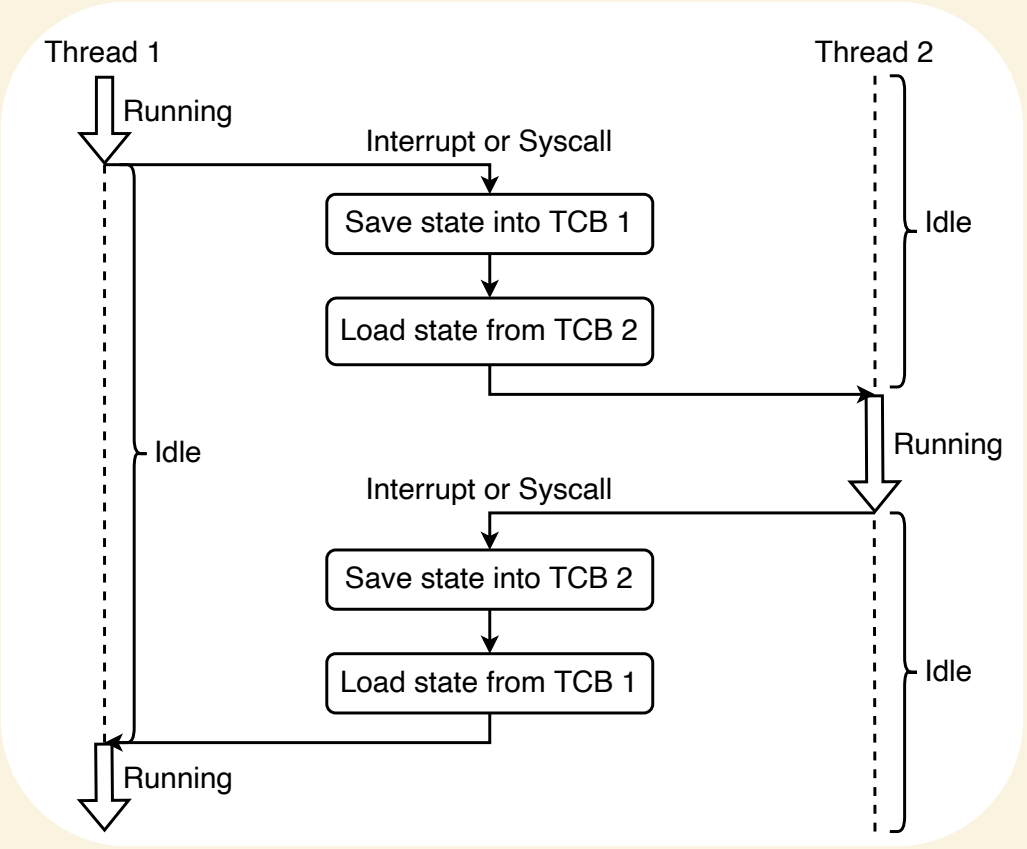
Criação de Thread

A thread é criada com `pthread_create` e esperada com `pthread_join`.

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

void *thread_function(void *arg) {
    printf("Olá do thread!\n");
    pthread_exit(0);
}

int main() {
    pthread_t thread1;
    pthread_create(&thread1, NULL, thread_function, NULL);
    pthread_join(thread1, NULL);
    return 0;
}
```



Mudança de Contexto

- Interrupções fazem com que o sistema operacional alterne a execução da CPU para uma rotina do kernel.
- Quando uma interrupção ocorre, o sistema deve salvar o contexto do processo/thread atual para restaurá-lo depois.
- O contexto de um processo é armazenado no PCB/TCB.
- Alternar a CPU entre processos requer salvar o estado do processo atual e restaurar o estado do novo processo.
- Esse procedimento é chamado de mudança de contexto.
- O tempo de mudança de contexto é considerado sobrecarga, pois não realiza trabalho útil
- Na imagem: mudança de contexo durante a execução intercalada de duas threads. Fonte: OER OS

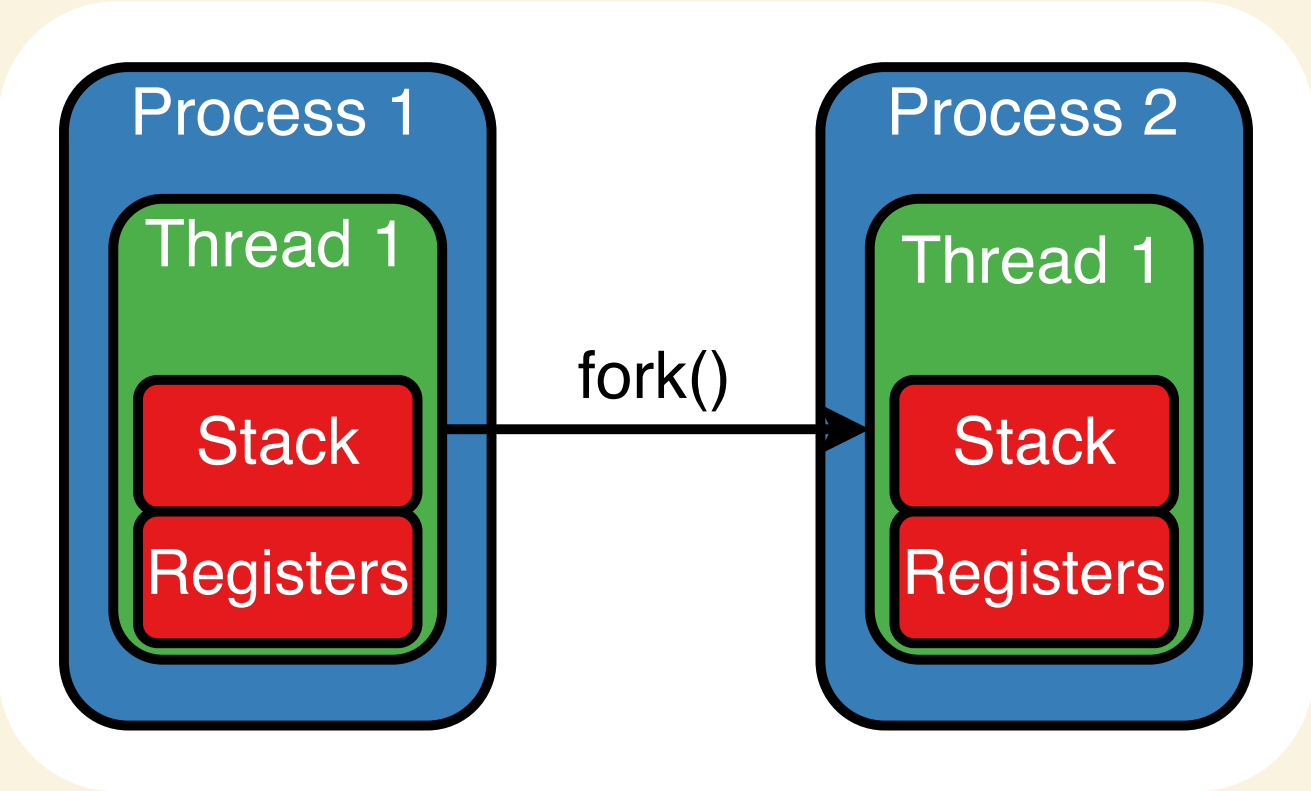
Criando Processos - `fork()`

- Processo-pai cria filhos, que podem criar outros processos, formando uma árvore de processos
- Um processo é identificado por um *process identifier* (pid)
- Usando `fork()`, o processo-filho é uma cópia do processo-pai.
  - O processo-filho executa a próxima instrução do programa.
  - O processo-pai continua executando a próxima instrução do seu programa.

```c
//pidouzero.c
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pid = fork();
    printf("fork retornou %d\n", pid);
    return 0;
}
```

```
$ ./pidouzero
Hello, world!
fork retornou 1111
fork retornou 0
```

Criando Processos - `fork()`



Cópia do processo-pai pela função `fork()`. Fonte: OER OS

Criando processos - `fork()` (cont.)

- `fork()` é chamada uma vez, mas retorna duas vezes:
  - No processo-pai: retorna o **pid** do processo-filho.
  - No processo-filho: retorna **zero**.

```c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    pid_t pid = fork(); //Cria processo-filho
    if (pid < 0){
        printf("Erro ao criar processo.\n")
    }
    else if (pid == 0){
        printf("Eu sou o filho.\n");
        exit(1); //Encerra o processo
    }
    else {
        printf("Eu sou o pai.\n");
        wait(NULL); //Espera o processo-filho encerrar
    }
    return 0;
}
```

Debug seu Conhecimento

Quantos processos são criados, incluindo o processo-pai?

```c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    fork();
    fork();
    fork();
    return 0;
}
```

> Note que não podemos assumir **a ordem de execução dos processos**. O SO decide a ordem de execução com base em seu algoritmo de escalonamento (aula futura).

Função `exec()`

A função `exec()` é uma syscall usada depois do `fork()` para carregar um novo programa na memória do processo-filho.

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    pid t pid;
    /* fork a child process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```

Encerramento de um Processo

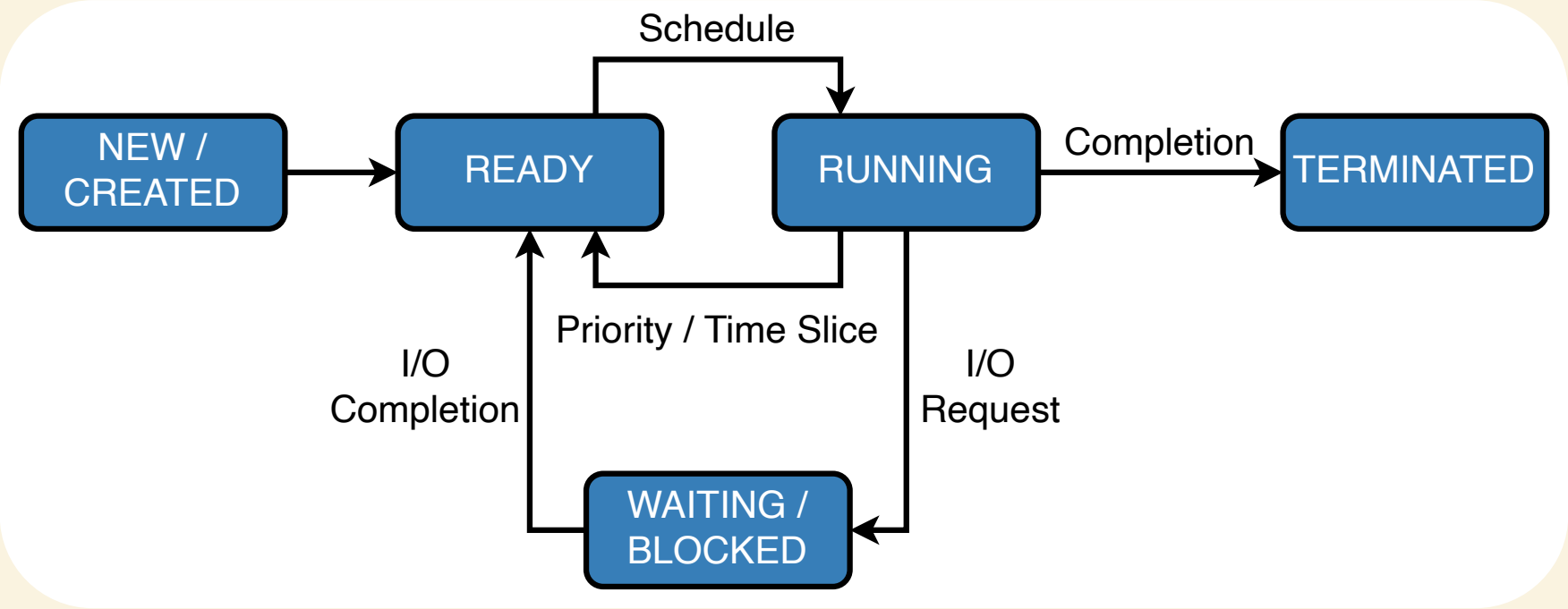
- Um processo é encerrado quando termina a execução de seu último comando e solicita ao sistema operacional que o exclua, usando a chamada de sistema `exit()`.
  - O processo pode retornar um valor de status (normalmente, um inteiro) para seu processo-pai. Exemplo: `exit(1);`
  - No encerramento normal, `exit()` pode ser chamada diretamente (como mostrado acima) ou indiretamente por um comando `return` em `main()`.
- Um processo-pai pode esperar o encerramento de um processo-filho usando a chamada de sistema `wait()`.
  - `wait()` recebe um parâmetro que permite que o pai obtenha o status de saída do filho. Exemplo: `int status; pid = wait(&status);`

Encerramento de um Processo (cont.)

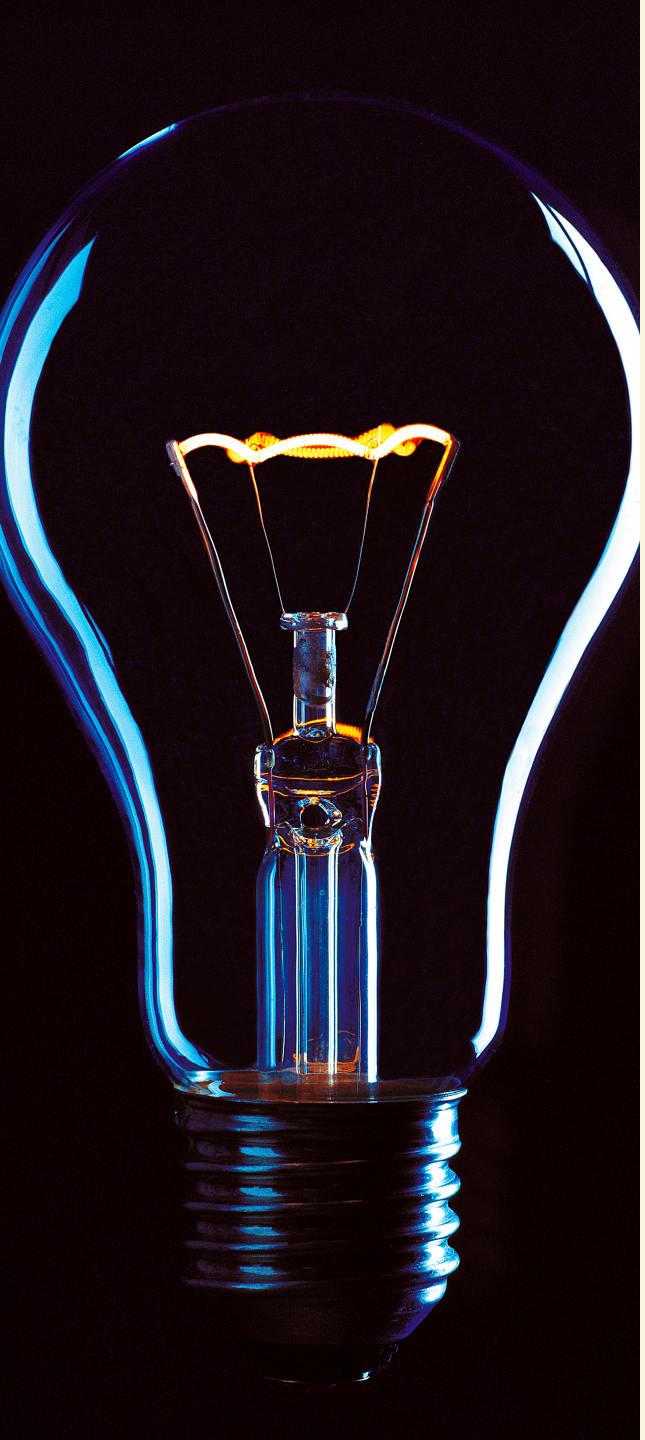
- Quando um processo termina, seus recursos são desalocados pelo sistema operacional. No entanto, **sua entrada na tabela de processos** deve permanecer **até que o pai chame** `wait()`, porque a tabela de processos contém o status de saída do processo.
  - Um processo que foi encerrado, mas cujo pai ainda não chamou `wait()`, é conhecido como processo **zumbi**.
  - Quando o pai chama `wait()`, o identificador do processo zumbi e sua entrada na tabela de processos são liberados.
- Um processo cujo pai não invocou `wait()` e, em vez disso, foi encerrado é chamado de proceso **órfão**.

Diagrama de estados de threads/processos

Conforme threads e processos executam, eles mudam de estado:



Fonte da Imagem: OER OS

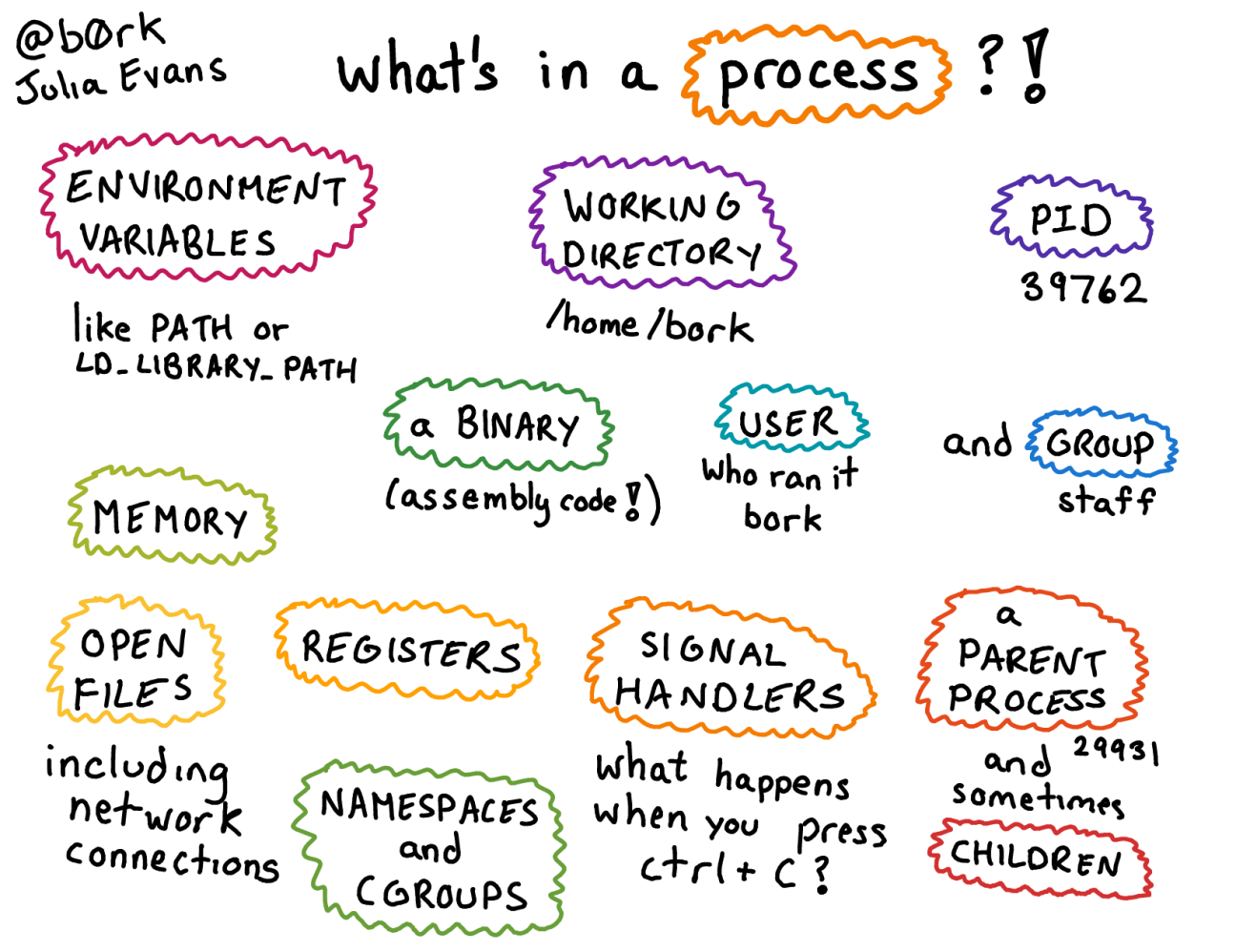


Conclusão

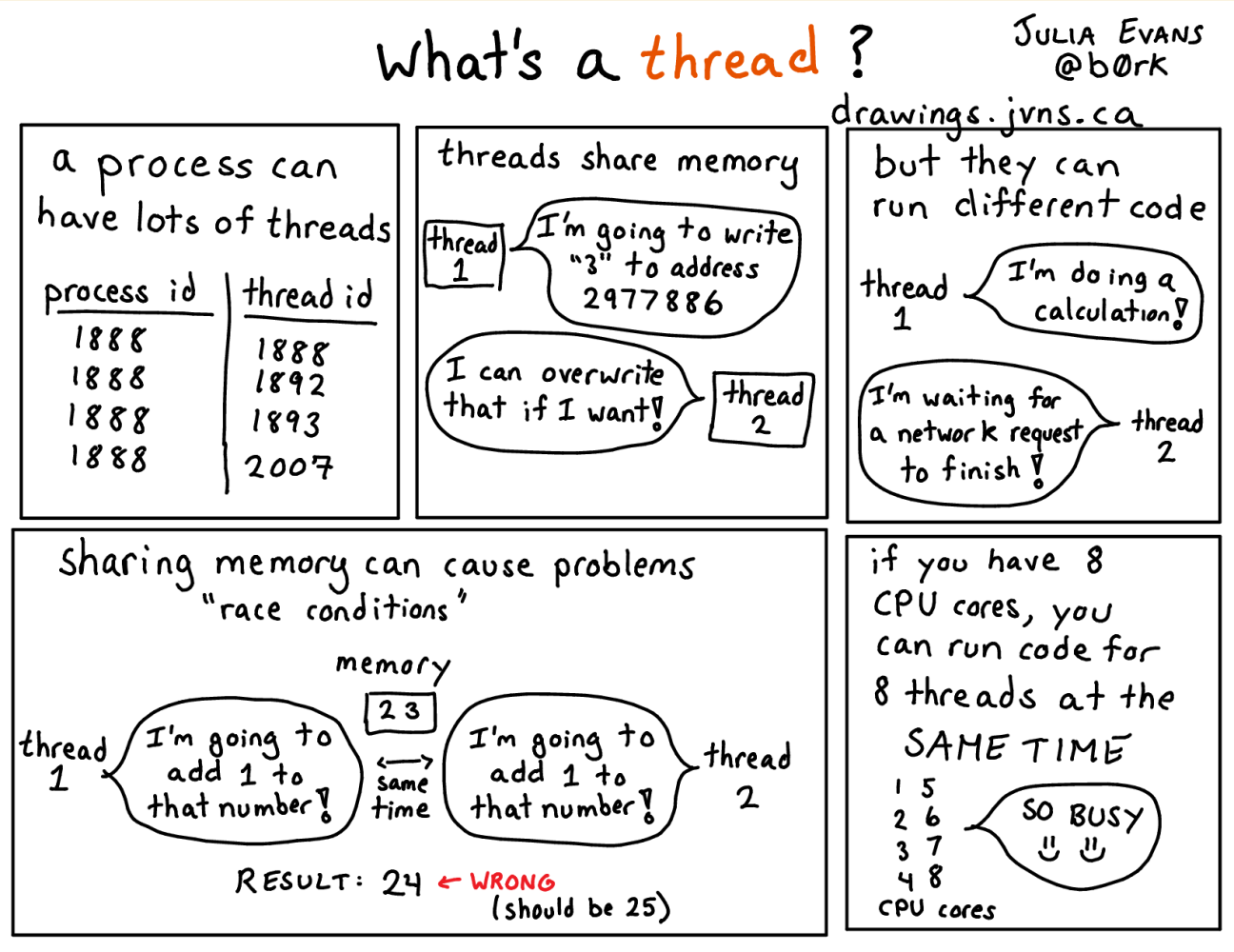
- **Sistemas Operacionais:** Gerenciam a execução de múltiplos processos e threads simultaneamente.
- **Multitarefa:** Permite melhor **utilização dos recursos** da CPU. A **multitarefa** permite que vários processos compartilhem a CPU de forma eficiente.
  - Melhora a **responsividade** do sistema.
  - Permite melhor **utilização dos recursos** da CPU.
- Um **processo** é uma instância em execução de um programa, enquanto uma **thread** é a menor unidade de execução dentro de um processo.
- **Execução Concorrente vs. Paralela:** Concorrência alterna a execução entre threads, enquanto o paralelismo ocorre simultaneamente em múltiplos núcleos.

Leitura Adicional

- Capítulo 2 do livro **Sistemas Operacionais Modernos**, de A. TANENBAUM
- Capítulo 3 do livro: **Operating Systems Concepts**, de A. SILBERCHATZ *et. al*.



Fonte da Imagem: Julia Evans.



Fonte da Imagem: Julia Evans.

Material Adicional

Threads em Linguagem C

Passagem de Argumentos

O nome "Ada" é passado como um argumento da função `thread_function`. É importante o cast do `void *` para `char *` para acessar o argumento corretamente.

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

void *thread_function(void *arg) {
    char *name = (char *)arg; // Cast do argumento para char*
    printf("Olá, %s!\n", name);
    pthread_exit(0);
}

int main() {
    pthread_t thread1;
    pthread_create(&thread1, NULL, thread_function, "Ada"); // Passa o nome como argumento
    pthread_join(thread1, NULL);
    return 0;
}
```

Múltiplas Threads

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h> // Para sleep()

void *print_message(void *arg) {
    int thread_id = (int)(long)arg; /* Cast para obter o ID da thread */
    printf("Thread %d: Olá do thread!\n", thread_id);
    sleep(1); /* Pausa a execução da thread por 1 segundo */
    printf("Thread %d: Thread terminando...\n", thread_id);
    pthread_exit(0);
}

int main() {
    pthread_t threads[5]; /* Declara um array de threads (pthread_t) para armazenar os IDs das threads criadas */
    int thread_ids[5];

    /* Cria 5 threads */
    for (int i = 0; i < 5; i++) {
        thread_ids[i] = i + 1; /* Atribui um ID único para cada thread */
        pthread_create(&threads[i], NULL, print_message, (void *)thread_ids[i]);
    }
    /* Faz com que o programa principal espere até que as threads sejam finalizadas antes de continuar a execução */
    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Programa principal terminado.\n");
    return 0;
}
```

Execução do Programa

Compilando o Código e Executando o Programa

```
gcc -c multiple_threads.c
gcc -o multiple_threads multiple_threads.o
./multiple_threads
```

Saída possível (a ordem em que as mensagens são impressas pode variar devido à natureza concorrente das threads)

```
Thread 1: Olá do thread!
Thread 2: Olá do thread!
Thread 3: Olá do thread!
Thread 4: Olá do thread!
Thread 5: Olá do thread!
Thread 1: Thread terminando...
Thread 2: Thread terminando...
Thread 3: Thread terminando...
Thread 4: Thread terminando...
Thread 5: Thread terminando...
Programa principal terminado.
```

Função de Thread com retorno

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *increment_counter(void *arg) {
    int *iptr = (int *)malloc(sizeof(int));
    *iptr = 0;
    for (int i = 0; i <= 10; i++){
        (*iptr)++;
    }
    return iptr;
}

int main() {
    pthread_t thread1;
    int *resultado;
    pthread_create(&thread1, NULL, increment_counter, NULL);

    //pthread_join permite acessar o retorno da função da thread
    pthread_join(thread1, (void *)&resultado);
    printf("Thread retornou o valor %d\n", *resultado);
    return 0;
}
```

Dúvidas e Discussão