SO: Multitarefa e Threads

Projetos de Sistemas Operacionais

Prof. Dr. Denis M. L. Martins

Engenharia de Computação: 5º Semestre

Introdução

Objetivos de Aprendizado

- Definir o que é uma thread e como ela difere de um processo.
- Explicar as vantagens do uso de threads em sistemas operacionais.
- Compreender o ciclo de vida de uma thread e como ela é gerenciada pelo sistema operacional.

Disclaimer

Parte do material apresentado a seguir foi adaptado de *IT Systems – Open Educational Resource*, disponível em https://oer.gitlab.io/oer-courses/it-systems/, produzido por Jens Lechtenböger, e distribuído sob a licença CC BY-SA 4.0.

Threads

Threads

- **Thread** = unidade de escalonamento do SO. Sequência independente de computações.
- Thread são mais leves e mais fáceis de criar e destruir do que processos.
- Permitem um novo nível de multitarefa:
  - 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 $\rightarrow$ **3 threads num mesmo processo**.
- Alto grau de *independência*: pense em funções diferentes no código.
- Concorrência e divisão de tempo de execução: em uma CPU de único núcleo, 3 threads pareceriam rodar em 3 núcleos fictícios com 1/3 da velocidade da CPU real.
- Núcleos individuais não estão se tornando significativamente mais rápidos (relembre a Lei de Moore). $\rightarrow$ É necessário utilizar programação paralela para aproveitar o hardware atual.

Threads

- **Thread** = unidade de escalonamento do SO. Sequência independente de computações.
- Thread são mais leves e mais fáceis de criar e destruir do que processos.
- Permitem um novo nível de multitarefa:
  - 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 $\rightarrow$ **3 threads num mesmo processo**.
- Alto grau de *independência*: pense em funções diferentes no código.
- Concorrência e divisão de tempo de execução: em uma CPU de único núcleo, 3 threads pareceriam rodar em 3 núcleos fictícios com 1/3 da velocidade da CPU real.
- Núcleos individuais não estão se tornando significativamente mais rápidos (relembre a Lei de Moore). $\rightarrow$ É necessário utilizar programação paralela para aproveitar o hardware atual.

Threads

- **Thread** = unidade de escalonamento do SO. Sequência independente de computações.
- Thread são mais leves e mais fáceis de criar e destruir do que processos.
- Permitem um novo nível de multitarefa:
  - 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 $\rightarrow$ **3 threads num mesmo processo**.
- Alto grau de *independência*: pense em funções diferentes no código.
- Concorrência e divisão de tempo de execução: em uma CPU de único núcleo, 3 threads pareceriam rodar em 3 núcleos fictícios com 1/3 da velocidade da CPU real.
- Núcleos individuais não estão se tornando significativamente mais rápidos (relembre a Lei de Moore). $\rightarrow$ É necessário utilizar programação paralela para aproveitar o hardware atual.

Threads

- O programador decide quantas threads são criadas.
  - API de Java para threads.
  - PThreads em C.
- Cada thread executa seu próprio código.
- Thread de um mesmo processo têm exatamente o mesmo espaço de endereçamento: compartilham as mesma variáveis globais e outros recursos do sistema operacional, como arquivos abertos e sinais.

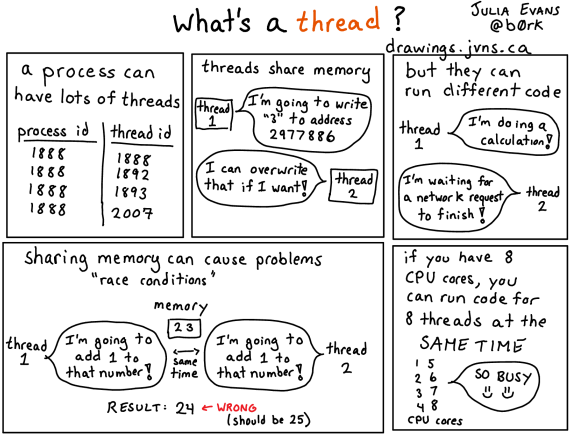


Figura 1: O que é uma thread? Créditos: Julia Evans.

Threads

- O programador decide quantas threads são criadas.
  - API de Java para threads.
  - PThreads em C.
- Cada thread executa seu próprio código.
- Thread de um mesmo processo têm exatamente o mesmo espaço de endereçamento: compartilham as mesma variáveis globais e outros recursos do sistema operacional, como arquivos abertos e sinais.

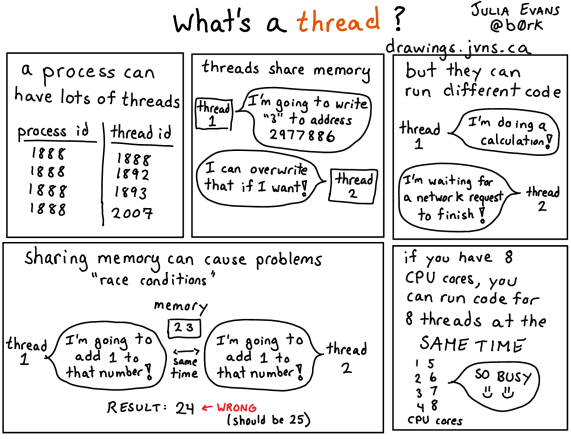
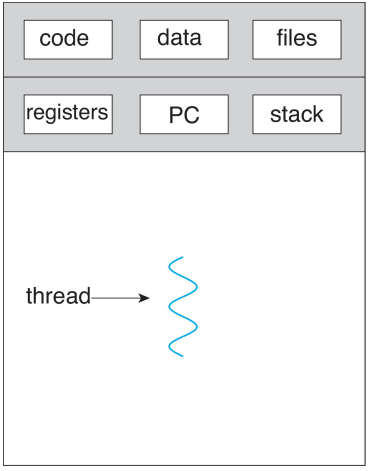
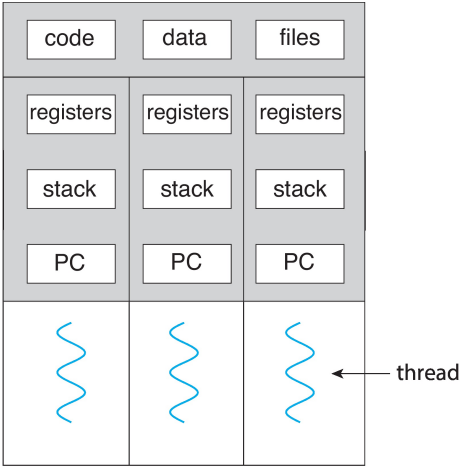


Figura 1: O que é uma thread? Créditos: Julia Evans.

Threads e Processos



single-threaded process



multithreaded process

Razões para usar Threads

- **Utilização de Recursos:**
  - A programação multithreaded mantém o hardware ocupado, evitando desperdício de recursos.
  - Quando uma thread está bloqueada (ex: esperando I/O ou eventos externos), outras podem continuar executando.
  - Distribuir tarefas entre múltiplas threads e núcleos de CPU aumenta a eficiência e acelera a execução.
- **Melhoria na Responsividade:**
  - Threads dedicadas a eventos externos garantem respostas rápidas.
  - Essencial para sistemas em tempo real, como jogos e servidores.

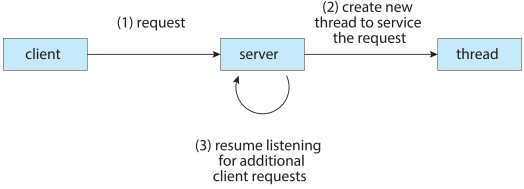


Figura 2: Aplicação de threads em um servidor web. Créditos: Silberschatz, Galvin and Gagne, 2018.

Execução Intercalada de Threads

Execução de Threads

- Relembre a diferença entre **Concorrência** e **Paralelismo** → paradigmas de execução.
- Cada thread tem sua própria *stack*.
- Uma thread pode ler, escrever ou até mesmo apagar a *stack* de um outro thread.
- A execução de múltiplas threads torna-se mais complexa quando elas precisam interagir entre si (problema de sincronização), e.g., thread escreve dados na memória e outra os lê.

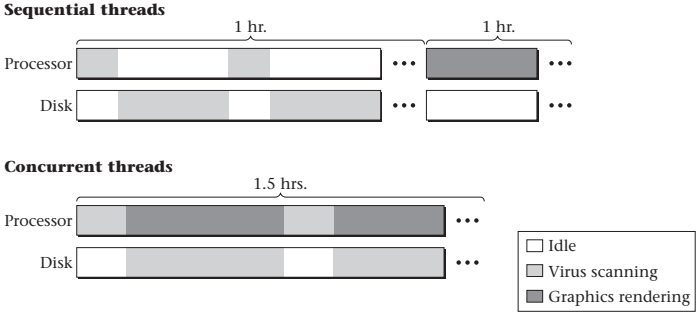


Figura 3: Execução Intercalada de Threads para otimizar a utilização de recursos. Créditos: Max Hailperin.

Mudança de Contexto

- SO escalona threads para execução (usando, e.g., técnicas de time-slicing $\rightarrow$ aula futura).
- Semelhante ao que ocorre com processos, escalonamento de threads involve **mudança de contexto**.
  - Remover a thread atual da CPU: salvar o estado da computação $\rightarrow$ Thread Control Block (TCB), semelhante ao PCB (aula anterior).
  - Despachar a próxima thread para a CPU: recuperar o estado da computação.
- Mais leve que escalonar processos.
  - Todas as threads compartilham os mesmos registradores da CPU.
  - Cada thread tem o seu *Program Counter* (PC) e um *Stack Pointer* (SP).

Mudança de Contexto

- SO escalona threads para execução (usando, e.g., técnicas de time-slicing $\rightarrow$ aula futura).
- Semelhante ao que ocorre com processos, escalonamento de threads involve **mudança de contexto**.
  - Remover a thread atual da CPU: salvar o estado da computação $\rightarrow$ Thread Control Block (TCB), semelhante ao PCB (aula anterior).
  - Despachar a próxima thread para a CPU: recuperar o estado da computação.
- Mais leve que escalonar processos.
  - Todas as threads compartilham os mesmos registradores da CPU.
  - Cada thread tem o seu *Program Counter* (PC) e um *Stack Pointer* (SP).

Mudança de Contexto

- SO escalona threads para execução (usando, e.g., técnicas de time-slicing $\rightarrow$ aula futura).
- Semelhante ao que ocorre com processos, escalonamento de threads involve **mudança de contexto**.
  - Remover a thread atual da CPU: salvar o estado da computação $\rightarrow$ Thread Control Block (TCB), semelhante ao PCB (aula anterior).
  - Despachar a próxima thread para a CPU: recuperar o estado da computação.
- Mais leve que escalonar processos.
  - Todas as threads compartilham os mesmos registradores da CPU.
  - Cada thread tem o seu *Program Counter* (PC) e um *Stack Pointer* (SP).

Mudança de Contexto

- SO escalona threads para execução (usando, e.g., técnicas de time-slicing $\rightarrow$ aula futura).
- Semelhante ao que ocorre com processos, escalonamento de threads involve **mudança de contexto**.
  - Remover a thread atual da CPU: salvar o estado da computação $\rightarrow$ Thread Control Block (TCB), semelhante ao PCB (aula anterior).
  - Despachar a próxima thread para a CPU: recuperar o estado da computação.
- Mais leve que escalonar processos.
  - Todas as threads compartilham os mesmos registradores da CPU.
  - Cada thread tem o seu *Program Counter* (PC) e um *Stack Pointer* (SP).

Pool de Threads

- Objetivo: melhorar a eficiência reduzindo a sobrecarga da mudança de contexto.
- Cria um conjunto de threads *workers* na inicialização do sistema (alocação de memória realizada).
- Ao receber uma requisição de tarefa, uma thread *dispatcher* "acorda" um dos workers para processar a requisição.
  - se algum worker estiver disponível: requisição é processada;
  - senão, a requisição é armazenada numa fila de tarefas até uma thread worker se tornar disponível.
- Completada a tarefa, a thread worker retorna para a pool, onde aguarda por uma nova tarefa.
- Vantagens: Reusa a estrutura de dados das threads. Limita a quantidade de threads criadas.
- Problema: dispatcher pode se tornar um gargalo do sistema ou delays longos se #requests $\gg$ #workers.

Pool de Threads

- Objetivo: melhorar a eficiência reduzindo a sobrecarga da mudança de contexto.
- Cria um conjunto de threads *workers* na inicialização do sistema (alocação de memória realizada).
- Ao receber uma requisição de tarefa, uma thread *dispatcher* "acorda" um dos workers para processar a requisição.
  - se algum worker estiver disponível: requisição é processada;
  - senão, a requisição é armazenada numa fila de tarefas até uma thread worker se tornar disponível.
- Completada a tarefa, a thread worker retorna para a pool, onde aguarda por uma nova tarefa.
- Vantagens: Reusa a estrutura de dados das threads. Limita a quantidade de threads criadas.
- Problema: dispatcher pode se tornar um gargalo do sistema ou delays longos se #requests $\gg$ #workers.

Pool de Threads

- Objetivo: melhorar a eficiência reduzindo a sobrecarga da mudança de contexto.
- Cria um conjunto de threads *workers* na inicialização do sistema (alocação de memória realizada).
- Ao receber uma requisição de tarefa, uma thread *dispatcher* "acorda" um dos workers para processar a requisição.
  - se algum worker estiver disponível: requisição é processada;
  - senão, a requisição é armazenada numa fila de tarefas até uma thread worker se tornar disponível.
- Completada a tarefa, a thread worker retorna para a pool, onde aguarda por uma nova tarefa.
- Vantagens: Reusa a estrutura de dados das threads. Limita a quantidade de threads criadas.
- Problema: dispatcher pode se tornar um gargalo do sistema ou delays longos se #requests $\gg$ #workers.

Pool de Threads

- Objetivo: melhorar a eficiência reduzindo a sobrecarga da mudança de contexto.
- Cria um conjunto de threads *workers* na inicialização do sistema (alocação de memória realizada).
- Ao receber uma requisição de tarefa, uma thread *dispatcher* "acorda" um dos workers para processar a requisição.
  - se algum worker estiver disponível: requisição é processada;
  - senão, a requisição é armazenada numa fila de tarefas até uma thread worker se tornar disponível.
- Completada a tarefa, a thread worker retorna para a pool, onde aguarda por uma nova tarefa.
- Vantagens: Reusa a estrutura de dados das threads. Limita a quantidade de threads criadas.
- Problema: dispatcher pode se tornar um gargalo do sistema ou delays longos se #requests $\gg$ #workers.

Pool de Threads

- Objetivo: melhorar a eficiência reduzindo a sobrecarga da mudança de contexto.
- Cria um conjunto de threads *workers* na inicialização do sistema (alocação de memória realizada).
- Ao receber uma requisição de tarefa, uma thread *dispatcher* "acorda" um dos workers para processar a requisição.
  - se algum worker estiver disponível: requisição é processada;
  - senão, a requisição é armazenada numa fila de tarefas até uma thread worker se tornar disponível.
- Completada a tarefa, a thread worker retorna para a pool, onde aguarda por uma nova tarefa.
- Vantagens: Reusa a estrutura de dados das threads. Limita a quantidade de threads criadas.
- Problema: dispatcher pode se tornar um gargalo do sistema ou delays longos se #requests $\gg$ #workers.

Pool de Threads

- Objetivo: melhorar a eficiência reduzindo a sobrecarga da mudança de contexto.
- Cria um conjunto de threads *workers* na inicialização do sistema (alocação de memória realizada).
- Ao receber uma requisição de tarefa, uma thread *dispatcher* "acorda" um dos workers para processar a requisição.
  - se algum worker estiver disponível: requisição é processada;
  - senão, a requisição é armazenada numa fila de tarefas até uma thread worker se tornar disponível.
- Completada a tarefa, a thread worker retorna para a pool, onde aguarda por uma nova tarefa.
- Vantagens: Reusa a estrutura de dados das threads. Limita a quantidade de threads criadas.
- Problema: dispatcher pode se tornar um gargalo do sistema ou delays longos se #requests $\gg$ #workers.

Pool de Threads

- Objetivo: melhorar a eficiência reduzindo a sobrecarga da mudança de contexto.
- Cria um conjunto de threads *workers* na inicialização do sistema (alocação de memória realizada).
- Ao receber uma requisição de tarefa, uma thread *dispatcher* "acorda" um dos workers para processar a requisição.
  - se algum worker estiver disponível: requisição é processada;
  - senão, a requisição é armazenada numa fila de tarefas até uma thread worker se tornar disponível.
- Completada a tarefa, a thread worker retorna para a pool, onde aguarda por uma nova tarefa.
- Vantagens: Reusa a estrutura de dados das threads. Limita a quantidade de threads criadas.
- Problema: dispatcher pode se tornar um gargalo do sistema ou delays longos se #requests $\gg$ #workers.

Pool de Threads

- Objetivo: melhorar a eficiência reduzindo a sobrecarga da mudança de contexto.
- Cria um conjunto de threads *workers* na inicialização do sistema (alocação de memória realizada).
- Ao receber uma requisição de tarefa, uma thread *dispatcher* "acorda" um dos workers para processar a requisição.
  - se algum worker estiver disponível: requisição é processada;
  - senão, a requisição é armazenada numa fila de tarefas até uma thread worker se tornar disponível.
- Completada a tarefa, a thread worker retorna para a pool, onde aguarda por uma nova tarefa.
- Vantagens: Reusa a estrutura de dados das threads. Limita a quantidade de threads criadas.
- Problema: dispatcher pode se tornar um gargalo do sistema ou delays longos se #requests $\gg$ #workers.

Threads de Usuário e Threads de *Kernel*

- **Threads de usuário**: gerenciamento por biblioteca de threads a nível (espaço) de usuário.
  - POSIX Pthreads
  - Windows threads
  - Java threads
- **Threads de *kernel***: suportadas pelo kernel.
  - Virtualmente todos os SOs: Windows, Linux, macOS, iOS, Android.
- Vários modelos de implementação:
  - Many-to-One: Várias threads do usuário mapeadas para uma única thread de kernel.
  - One-to-One: Cada thread do usuário mapeada para uma thread de kernel.
  - Many-to-Many: Múltiplas threads do usuário podem ser mapeadas em uma ou mais threads de kernel.

Classificação de Threads

- **Threads I/O Bound:**
  - Passam a maior parte do tempo esperando operações de entrada/saída (I/O).
  - Executam por curtos períodos antes de ficarem bloqueadas para a próxima operação de I/O.
  - Exemplos: Scanners de vírus, servidores de rede.
- **Threads CPU Bound:**
  - Passam a maior parte do tempo executando cálculos intensivos.
  - Utilizam completamente os ciclos de CPU antes de serem interrompidas.
  - Exemplos: Renderização gráfica, compilação de código, treinamento de modelos de deep learning.

Classificação de Threads

- **Threads I/O Bound:**
  - Passam a maior parte do tempo esperando operações de entrada/saída (I/O).
  - Executam por curtos períodos antes de ficarem bloqueadas para a próxima operação de I/O.
  - Exemplos: Scanners de vírus, servidores de rede.
- **Threads CPU Bound:**
  - Passam a maior parte do tempo executando cálculos intensivos.
  - Utilizam completamente os ciclos de CPU antes de serem interrompidas.
  - Exemplos: Renderização gráfica, compilação de código, treinamento de modelos de deep learning.

Classificação de Threads

- **Threads I/O Bound:**
  - Passam a maior parte do tempo esperando operações de entrada/saída (I/O).
  - Executam por curtos períodos antes de ficarem bloqueadas para a próxima operação de I/O.
  - Exemplos: Scanners de vírus, servidores de rede.
- **Threads CPU Bound:**
  - Passam a maior parte do tempo executando cálculos intensivos.
  - Utilizam completamente os ciclos de CPU antes de serem interrompidas.
  - Exemplos: Renderização gráfica, compilação de código, treinamento de modelos de deep learning.

Classificação de Threads

- **Threads I/O Bound:**
  - Passam a maior parte do tempo esperando operações de entrada/saída (I/O).
  - Executam por curtos períodos antes de ficarem bloqueadas para a próxima operação de I/O.
  - Exemplos: Scanners de vírus, servidores de rede.
- **Threads CPU Bound:**
  - Passam a maior parte do tempo executando cálculos intensivos.
  - Utilizam completamente os ciclos de CPU antes de serem interrompidas.
  - Exemplos: Renderização gráfica, compilação de código, treinamento de modelos de deep learning.

Classificação de Threads

- **Threads I/O Bound:**
  - Passam a maior parte do tempo esperando operações de entrada/saída (I/O).
  - Executam por curtos períodos antes de ficarem bloqueadas para a próxima operação de I/O.
  - Exemplos: Scanners de vírus, servidores de rede.
- **Threads CPU Bound:**
  - Passam a maior parte do tempo executando cálculos intensivos.
  - Utilizam completamente os ciclos de CPU antes de serem interrompidas.
  - Exemplos: Renderização gráfica, compilação de código, treinamento de modelos de deep learning.

Classificação de Threads

- **Threads I/O Bound:**
  - Passam a maior parte do tempo esperando operações de entrada/saída (I/O).
  - Executam por curtos períodos antes de ficarem bloqueadas para a próxima operação de I/O.
  - Exemplos: Scanners de vírus, servidores de rede.
- **Threads CPU Bound:**
  - Passam a maior parte do tempo executando cálculos intensivos.
  - Utilizam completamente os ciclos de CPU antes de serem interrompidas.
  - Exemplos: Renderização gráfica, compilação de código, treinamento de modelos de deep learning.

Biblioteca de Threads

Biblioteca de Threads

- Objetivo: prover ao programador uma API para criação e gerenciamento de threads.
- Biblioteca pode ser implementada inteiramente em espaço do usuário ou pode estar a nível de kernel.
- Exemplo de biblioteca: PThreads em C.
  - Segue o padrão da API POSIX (IEEE 1003.1c). Comum em UNIX (Linux e macOS)
  - API especifica o comportamento da biblioteca de threads, mas a implementação é de decisão do desenvolvedor da biblioteca.

Função da API	Descrição
`pthread_create`	Cria uma nova thread e a inicia na função especificada.
`pthread_exit`	Termina a execução de uma thread e libera seus recursos
`pthread_join`	Aguarda a conclusão de uma thread específica.
`pthread_attr_init`	Cria e inicializa a estrutura de atributos da thread.

Exemplo de threads em PThreads

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

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}
```

Exemplo de threads em PThreads (cont.)

```
/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Exemplo de threads em PThreads (cont.)

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Exemplo de Thread em Java

```java
public class Simpler2Threads { // Based on Fig. 2.3 of [Hai17]
      // "Simplified" by removing anonymous class.
      public static void main(String args[]){
        Thread childThread = new Thread(new MyThread());
        childThread.start();
        sleep(5000);
        System.out.println("Parent is done sleeping 5 seconds.");}

      static void sleep(int milliseconds){
        // Sleep milliseconds (blocked/removed from CPU).
        try{ Thread.sleep(milliseconds); } catch(InterruptedException e){
          // ignore this exception; it won't happen anyhow
        }}}

    class MyThread implements Runnable {
      public void run(){
        Simpler2Threads.sleep(3000);
        System.out.println("Child is done sleeping 3 seconds.");
      }}
```

Problemas de Threads

Problemas

Semântica de chamadas

- Em multithreading, a semântica das chamadas `fork()` e `exec()` pode ser confusa.
- Chamar `fork()` em uma thread duplicaria **todas** as threads ou o novo processo seria ***single-thread***?
- Alguns sistemas UNIX usam duas versões de `fork()`.
- `exec()` funciona da mesma forma: o programa especificado substituirá o processo e suas threads.

Problemas (cont.)

Manipulação de Sinais

- UNIX sua sinais para notificar processos sobre a ocorrência de eventos.
  1. Sinal gerado por evento.
  2. Sinal passado ao processo.
  3. Sinal deve ser processado.
- Sinais síncronos: acesso ilegal de memória, divisão por zero, etc. $\rightarrow$ Enviados ao processo que causou o evento.
- Sinais assíncronos: evento externo, `CONTROL+C` $\rightarrow$ Kernel processa os sinais ou usuário define um processador personalizado.
- Em processo single-thread: sinais são sempre passados para o processo.
- Em multithread: depende do tipo de sinal, e.g., manda para todas as threads ou para uma thread em específico.
- Em UNIX: `kill(pid_t pid, int signal)` para enviar sinal.

Fechamento e Perspectivas

Palavras Finais

- **Resumo:**
  - Uma thread é a menor unidade de execução dentro de um processo.
  - Diferentes threads de um mesmo processo compartilham memória e recursos.
- **Importância de Threads:**
  - Melhora a **responsividade** do sistema.
  - Compartilhamento de recursos, reduzindo o custo de criação e troca de contexto.
  - É essencial para aplicações modernas como servidores web, sistemas em tempo real e aplicações gráficas.
- **Próximos Passos:**
  - Explorar a API POSIX Threads (pthreads) em sistemas UNIX para criação de threads.
  - Compreender algoritmos de escalonamento no SO.

Palavras Finais

- **Resumo:**
  - Uma thread é a menor unidade de execução dentro de um processo.
  - Diferentes threads de um mesmo processo compartilham memória e recursos.
- **Importância de Threads:**
  - Melhora a **responsividade** do sistema.
  - Compartilhamento de recursos, reduzindo o custo de criação e troca de contexto.
  - É essencial para aplicações modernas como servidores web, sistemas em tempo real e aplicações gráficas.
- **Próximos Passos:**
  - Explorar a API POSIX Threads (pthreads) em sistemas UNIX para criação de threads.
  - Compreender algoritmos de escalonamento no SO.

Palavras Finais

- **Resumo:**
  - Uma thread é a menor unidade de execução dentro de um processo.
  - Diferentes threads de um mesmo processo compartilham memória e recursos.
- **Importância de Threads:**
  - Melhora a **responsividade** do sistema.
  - Compartilhamento de recursos, reduzindo o custo de criação e troca de contexto.
  - É essencial para aplicações modernas como servidores web, sistemas em tempo real e aplicações gráficas.
- **Próximos Passos:**
  - Explorar a API POSIX Threads (pthreads) em sistemas UNIX para criação de threads.
  - Compreender algoritmos de escalonamento no SO.

Palavras Finais

- **Resumo:**
  - Uma thread é a menor unidade de execução dentro de um processo.
  - Diferentes threads de um mesmo processo compartilham memória e recursos.
- **Importância de Threads:**
  - Melhora a **responsividade** do sistema.
  - Compartilhamento de recursos, reduzindo o custo de criação e troca de contexto.
  - É essencial para aplicações modernas como servidores web, sistemas em tempo real e aplicações gráficas.
- **Próximos Passos:**
  - Explorar a API POSIX Threads (pthreads) em sistemas UNIX para criação de threads.
  - Compreender algoritmos de escalonamento no SO.

Palavras Finais

- **Resumo:**
  - Uma thread é a menor unidade de execução dentro de um processo.
  - Diferentes threads de um mesmo processo compartilham memória e recursos.
- **Importância de Threads:**
  - Melhora a **responsividade** do sistema.
  - Compartilhamento de recursos, reduzindo o custo de criação e troca de contexto.
  - É essencial para aplicações modernas como servidores web, sistemas em tempo real e aplicações gráficas.
- **Próximos Passos:**
  - Explorar a API POSIX Threads (pthreads) em sistemas UNIX para criação de threads.
  - Compreender algoritmos de escalonamento no SO.

Palavras Finais

- **Resumo:**
  - Uma thread é a menor unidade de execução dentro de um processo.
  - Diferentes threads de um mesmo processo compartilham memória e recursos.
- **Importância de Threads:**
  - Melhora a **responsividade** do sistema.
  - Compartilhamento de recursos, reduzindo o custo de criação e troca de contexto.
  - É essencial para aplicações modernas como servidores web, sistemas em tempo real e aplicações gráficas.
- **Próximos Passos:**
  - Explorar a API POSIX Threads (pthreads) em sistemas UNIX para criação de threads.
  - Compreender algoritmos de escalonamento no SO.

Palavras Finais

- **Resumo:**
  - Uma thread é a menor unidade de execução dentro de um processo.
  - Diferentes threads de um mesmo processo compartilham memória e recursos.
- **Importância de Threads:**
  - Melhora a **responsividade** do sistema.
  - Compartilhamento de recursos, reduzindo o custo de criação e troca de contexto.
  - É essencial para aplicações modernas como servidores web, sistemas em tempo real e aplicações gráficas.
- **Próximos Passos:**
  - Explorar a API POSIX Threads (pthreads) em sistemas UNIX para criação de threads.
  - Compreender algoritmos de escalonamento no SO.

Dúvidas e Discussão

Prof. Dr. Denis M. L. Martins

denis.mayr@puc-campinas.edu.br