Sistemas Operacionais

Threads e Modelos
Multithreading

Baseado nos slides do Prof. Felipe Lobo

Prof. Herbert Rocha
herbert.rocha@ufrr.br

*Threads (*Linha ou Encadeamento de execução*)*

- Razões para existência de *threads*:
  - Em múltiplas aplicações ocorrem múltiplas atividades “ao mesmo tempo”, e algumas dessas atividades podem bloquear de tempos em tempos.
  - As *threads* são mais fáceis de gerenciar do que processos, pois elas não possuem recursos próprios;
  - **Desempenho**: quando há grande quantidade de E/S, as threads permitem que essas atividades se sobreponham, acelerando a aplicação.
  - Paralelismo Real em sistemas com múltiplas CPUs.;

Threads

- Considere um servidor de arquivos:
  - Recebe diversas requisições de leitura e escrita em arquivos e envia respostas a essas requisições.
  - Para melhorar o desempenho, o servidor mantém uma *cache* dos arquivos mais recentes, lendo da *cache* e escrevendo na *cache* quando possível.
  - Quando uma requisição é feita, uma *thread* é alocada para seu processamento. Suponha que essa *thread* seja bloqueada esperando uma transferência de arquivos. Nesse caso, outras *threads* podem continuar atendendo a outras requisições.

Threads

- **Considere um navegador WEB:**
  - Muitas páginas WEB contêm muitas figuras que devem ser mostradas assim que a página é carregada.
  - Para cada figura, o navegador deve estabelecer uma conexão separada com o servidor da página.
  - Com múltiplas *threads*, muitas imagens podem ser requisitadas ao mesmo tempo melhorando o desempenho.

Threads

- **Benefícios:**
  - **Capacidade de resposta:** aplicações interativas Ex.: servidor WEB.
  - **Compartilhamento de recursos:** mesmo endereçamento, memória, recursos.
  - **Economia:** criar e realizar chaveamento de *threads* é mais barato.
  - **Utilização de arquiteturas multiprocessador:** processamento paralelo.

Threads

- Modo Usuário x Modo núcleo
  - Threads no modo núcleo, são mais lentas que no modo usuário, porém tem suporte ao multiprocessamento;
  - Threads no modo usuário, são mais rápidas porque dispensam o acesso ao núcleo;
- Implementações Híbridas:
  - São implementações que tentam combinar as vantagens dos threads de usuários com os threads de núcleo;

Threads

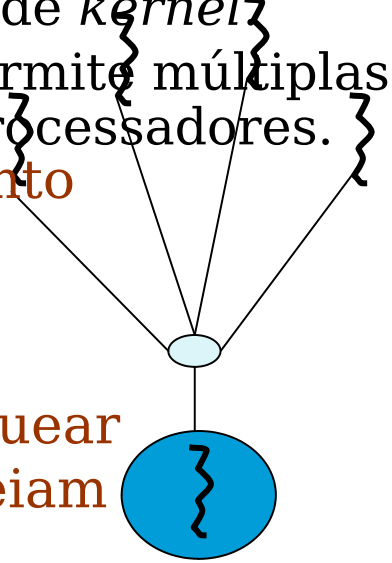
- Escalonamento:
  - Da mesma forma que os processos sofrem escalonamento, as threads também têm a mesma necessidade.
  - Com as threads ocorre o mesmo, elas esperam até serem executadas. Como esta alternância é muito rápida, há impressão de que todas as threads são executadas paralelamente.

Threads

- Ativações do Escalonador:
  - Tem a função de imitar threads de núcleo, porém com melhor desempenho;
  - Consegue ser eficiente por evitar transições desnecessárias entre o espaço usuário e núcleo;

Threads

- **Modelos *Multithreading***
  - **Muitos-para-um**: (*Green Threads e GNU Portable Threads*)
    - Mapeia muitas *threads* de usuário em apenas uma *thread* de *kernel*.
    - Não permite múltiplas *threads* em paralelo em multiprocessadores.



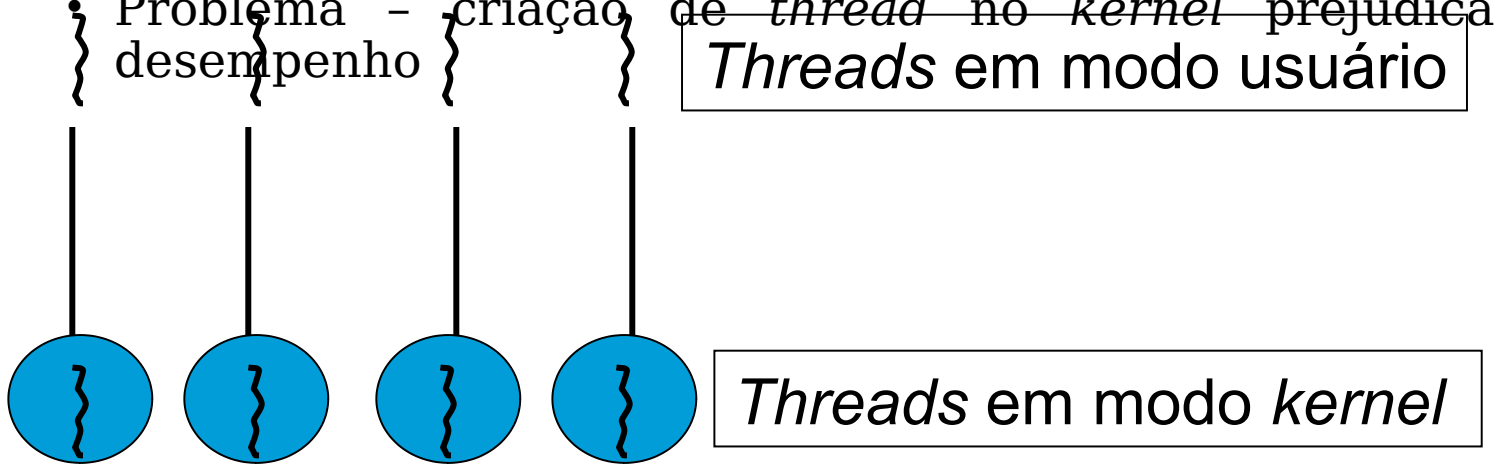
Threads em modo usuário

Thread em modo *kernel*

- Gerenciamento Eficiente
- Se uma bloquear todas bloqueiam

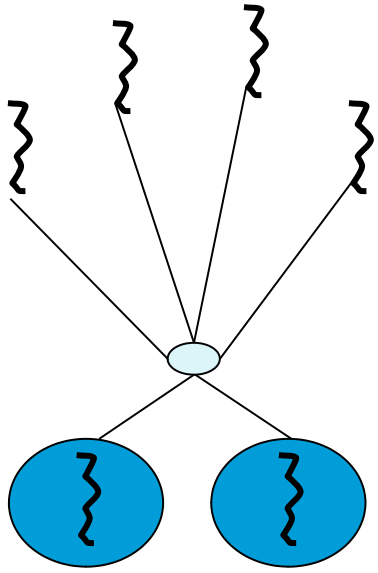
Threads

- **Modelos *Multithreading***
  - **Um-para-um:** (Linux, Família Windows, OS/2, Solaris 9)
    - Mapeia para cada *thread* de usuário uma *thread* de *kernel*;
    - Permite múltiplas *threads* em paralelo;
    - Problema – criação de *thread* no *kernel* prejudica o desempenho



Threads

- **Modelos *Multithreading***
  - **Muitos-para-muitos**: (Solaris até versão 8, HP-UX, Tru64 Unix, IRIX)
    - Mapeia para múltiplos *threads* de usuário um número menor ou igual de *threads* de *kernel*.
    - Permite múltiplas *threads* em paralelo.



Threads em modo usuário

Threads em modo *kernel*

Threads

- **Estados:** executando, pronta, bloqueada.

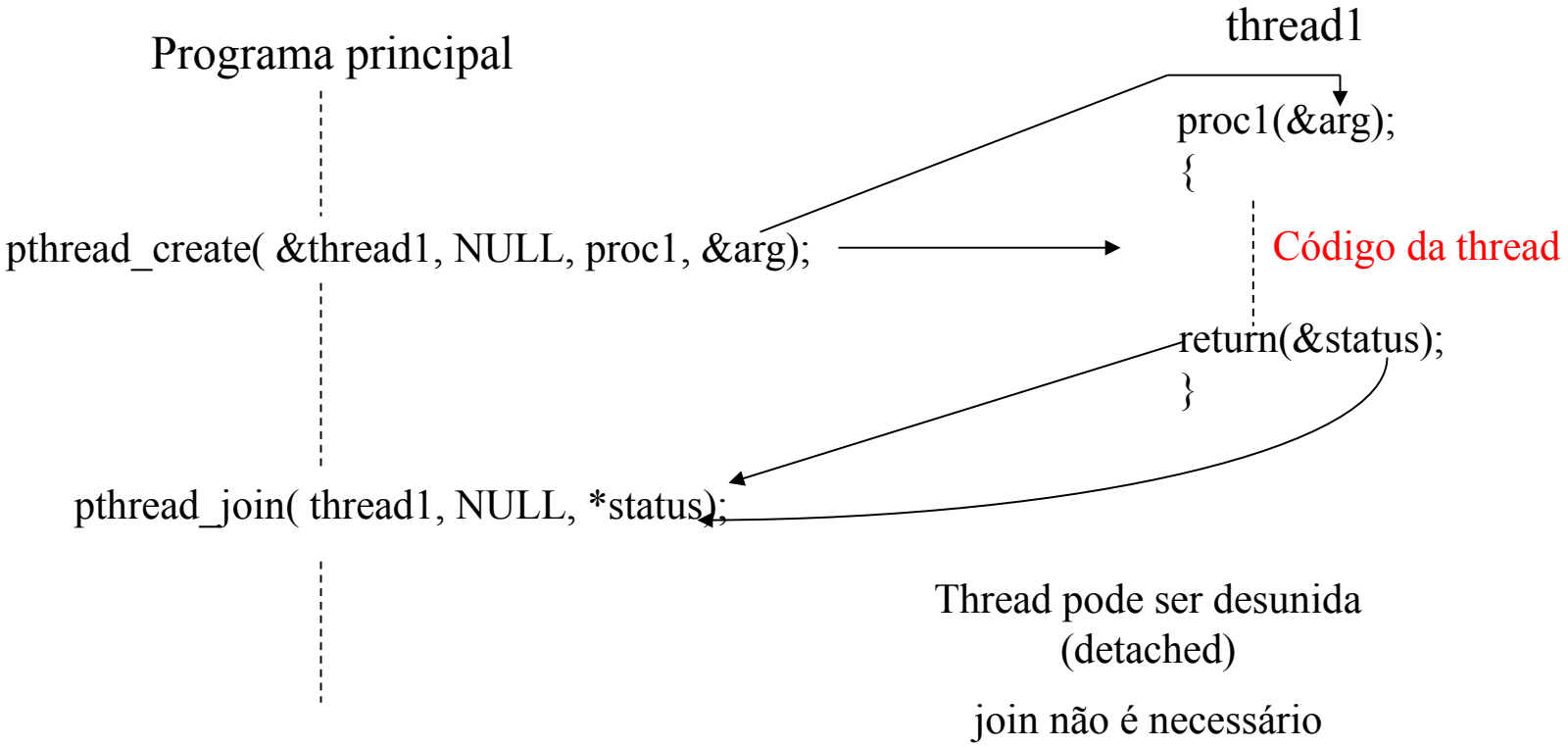
- Comandos para manipular threads:
  - *Thread_create;*
  - *Thread_exit;*
  - *Thread_wait;*
  - *Thread_yield* (permite que uma *thread* desista voluntariamente da CPU).

Implementação

- **Java**
  - Classe *Threads*
  - A própria linguagem fornece suporte para a criação e o gerenciamento das *threads*, as quais são gerenciadas pela JVM e não por uma biblioteca do usuário ou do kernel.

- **C**
  - Biblioteca *Pthreads*
  - Padrão POSIX (IEEE 1003.1c) que define uma API para a criação e sincronismo de *threads.*
  - Modo usuário

Threads em C
PThreads



Threads em C *PThreads*

- pthread_create (thread,attr,start_routine,arg)
  - **thread**: identificador único para a nova *thread* retornada pela função.
  - **attr**: Um objeto que pode ser usado para definir os atributos (como por exemplo, prioridade de escalonamento) da *thread*. Quando não há atributos, define-se como NULL.
  - **start_routine**: A rotina em C que a *thread* irá executar quando for criada.
  - **arg**: Um argumento que pode ser passado para a *start_routine*. Deve ser passado por referência com um *casting* para um ponteiro do tipo void. Pode ser usado NULL se nenhum argumento for passado.

Threads em C *PThreads*

- PThread Join
  - A rotina *pthread_join*() espera pelo término de uma thread específica:

```
for (i = 0; i < n; i++)
    pthread_create(&thread[i], NULL, (void *) slave, (void
   *) &arg);
// código thread mestre
// código thread mestre
for (i = 0; i < n; i++)
    pthread_join(thread[i], NULL);
```

Threads em C
PThreads

```
/******************************************************************************
* FILE: hello.c
* DESCRIPTION:
*   A "hello world" Pthreads program.  Demonstrates thread creation and
*   termination.
* AUTHOR: Blaise Barney
* LAST REVISED: 01/29/09
******************************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}
```

Threads em C
PThreads

```
int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
int rc;
long t;
for(t=0;t<NUM_THREADS;t++){
  printf("In main: creating thread %ld\n", t);
  rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
  if (rc){
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
    }
  }
pthread_exit(NULL);
}
```

Threads em Java

```
Runnable r1 = new Runnable() {
    public void run() {
      try {
        while (true) {
          System.out.println("Olá, mundo!");
          Thread.sleep(1000L);
        }
      } catch (InterruptedException iex) {}
    }
  };
  Runnable r2 = new Runnable() {
    public void run() {
      try {
        while (true) {
          System.out.println("Adeus, " + "mundo cruel!");
          Thread.sleep(2000L);
        }
      } catch (InterruptedException iex) {}
    }
  };
```

Um objeto *Runnable* define uma tarefa que será executada.
É uma interface, com um único método run().

http://www.treinaweb.com.br/ler-artigo/26/java-e-threads

Kernel Threads em C

- A maior diferença é que eles existem no espaço do kernel e são executados em um modo privilegiado e têm acesso total às estruturas de dados do kernel
- A tarefa pode ser o tratamento de eventos assíncronos ou aguardar a ocorrência de um evento.
- Os drivers de dispositivo utilizam os serviços de threads do kernel para lidar com tais tarefas.
- O thread de kernel khubd monitora os hubs usb e ajuda a configurar dispositivos usb durante o hot-plugging.

https://sysplay.in/blog/linux-kernel-internals/2015/04/kernel-threads/

Kernel Threads em C

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/delay.h>

static struct task_struct *thread_st;
```

```
#include <kthread.h>
kthread_create(int (*function)(void *data), void *data, const char name[], ...)
```

Kernel Threads em C

```
static int thread_fn(void *unused)
{
    while (1)
    {
        printk(KERN_INFO "Thread Running\n");
        ssleep(5);
    }
    printk(KERN_INFO "Thread Stopping\n");
    do_exit(0);
    return 0;
}
// Module Initialization
static int __init init_thread(void)
{
    printk(KERN_INFO "Creating Thread\n");
    //Create the kernel thread with name 'mythread'
    thread_st = kthread_create(thread_fn, NULL, "mythread");
    if (thread_st)
        printk("Thread Created successfully\n");
    else
        printk(KERN_INFO "Thread creation failed\n");
    return 0;
}
```

Kernel Threads em C

```
// Module Exit
static void __exit cleanup_thread(void)
{
    printk("Cleaning Up\n");
}
```

Rust Threads

https://www.rust-lang.org/pt-BR

Por que Rust?

Desempenho

Rust é extremamente rápido e gerencia memória eficientemente: sem *runtime* ou *garbage collector*, podendo potencializar a performance de serviços críticos, rodar em sistemas embarcados, e facilmente integrar-se a outras linguagens.

Confiabilidade

O rico sistema de tipos de Rust e seu modelo de *ownership* garantem segurança de memória e segurança de concorrência — e permite que você elimine muitas categorias de erros durante a compilação.

Produtividade

Rust possui uma ótima documentação, um compilador amigável com mensagens de erros úteis, e ferramental de primeira qualidade — uma ferramenta integrada de compilação e gerenciamento de pacotes, suporte inteligente para múltiplos editores com autocompleção e inspeções de tipos, um formatador automático, e muito mais.

Rust Threads

https://www.rust-lang.org/pt-BR

Construa com Rust

Em 2018, a comunidade de Rust decidiu melhorar a experiência de programar em alguns domínios distintos (veja o planejamento para 2018). Para estes, você pode encontrar várias *crates* de alta qualidade e alguns guias para lhe ajudar a começar.

Linha de Comando

Monte uma ferramenta de linha de comando rapidamente com o ecossistema robusto de Rust. Rust te ajuda a manter sua aplicação com confiança e a distribuí-la com facilidade.

CONSTRUINDO FERRAMENTAS

WebAssembly

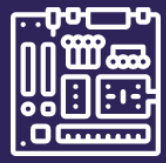
Use Rust para tornar seu JavaScript mais poderoso, um modulo por vez. Publique no npm, empacote com webpack, e você está pronto.

ESCREVENDO APLICAÇÕES WEB

Redes

Desempenho previsível. Pouco uso de recursos. Alta confiabilidade. Rust é ótimo para serviços de rede.

TRABALHANDO EM SERVIDORES



Embarcados

Planejando usar aparelhos com poucos recursos? Precisa de controle baixo nível sem desistir de conveniências de alto nível? Rust tem a solução.

COMEÇANDO COM EMBARCADOS

Rust Threads

https://www.rust-lang.org/pt-BR

Rust em produção

Centenas de empresas ao redor do mundo já estão usando Rust em produção para criar solução multiplataforma rápidas e eficientes. Software que você conhece e ama, como Firefox, Dropbox, e Cloudflare usam Rust. **De startups a grandes corporações, de sistemas embarcados a serviços web escaláveis, Rust é uma ótima escolha.**

> Meu maior elogio à Rust é que ela é entediante, e isso é um elogio incrível.
>
> – Chris Dickinson, Engenheiro na npm, Inc

Hello Rust

Cargo é o sistema de construção e gerenciador de pacotes do Rust.

```
$ cargo -version

$ cargo new hello_w

$ cd hello_w

$ cargo build

$ cargo run

$ cargo build --release
```

```rust
fn main() {
    println!("Hello, world!");
}
```

Rust Threads

- Programação Concorrente: onde diferentes partes de um programa são executadas de forma independente.
- Aproveitando a verificação de tipo, muitos erros de concorrência são erros em tempo de compilação no Rust, em vez de erros em tempo de execução.
- A biblioteca padrão Rust fornece apenas uma implementação de threading 1:1.

Rust Threads

Filename: src/main.rs

thread::spawn - closure

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Ao contrário das funções, as closures podem capturar valores do escopo em que são definidos.

Rust Threads

Filename: src/main.rs

thread::spawn - closure

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```