



Universidade Tecnológica Federal do Paraná – UTFPR
Coordenação de Ciência da Computação - COCIC
Ciência da Computação

BCC34G – Sistemas Operacionais

Prof. Rogério A. Gonçalves
rogerioag@utfpr.edu.br

Aula 008

- Conceitos de *Threads*

Threads

- Qual é a ideia?
 - Processos leves (LWP).
- Tarefas de uma mesma aplicação
- Facilidades de compartilhamento de espaço de endereçamento e dados
- Fácil criação e destruição (cerca de 100 x)
- Maior paralelismo
- Nível Usuário x Nível SO
 - Devem ser gerenciados pelo sistema operacional ou pela aplicação de usuário.

Threads

- Itens compartilhados por todos os *threads* em um processo
- Itens privativos de cada *thread*

Itens por processo	Itens por thread
Espaço de endereçamento	Contador de programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos filhos	Estado
Alarmes pendentes	
Sinais e tratadores de sinais	
Informação de contabilidade	

Threads

- Operações com *Threads* incluem:
 - Criação
 - Terminação
 - Sincronização (*joins, blocking*)
 - Escalonamento (*scheduling*)
 - Gerenciamento de dados e interação com o processo.
- Uma *thread* não mantém uma lista de *threads* criadas, nem sabe qual *thread* a criou.

Threads

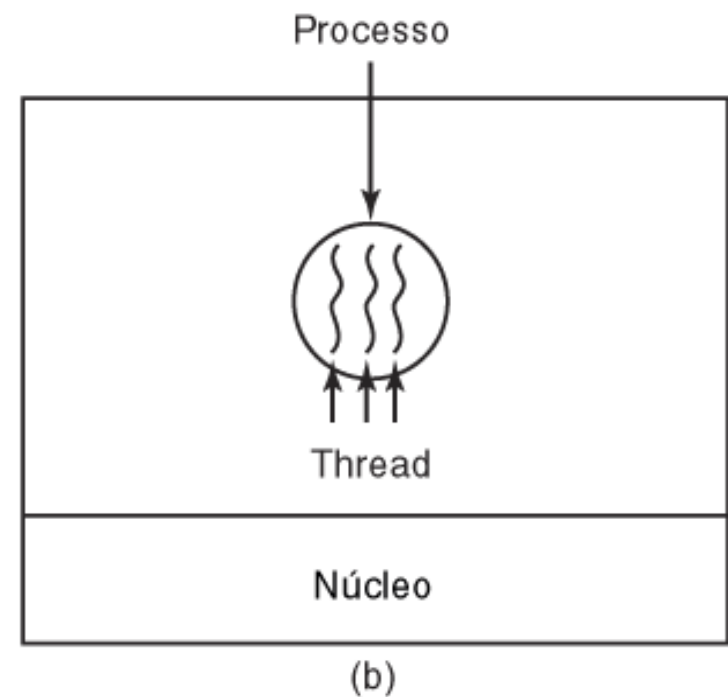
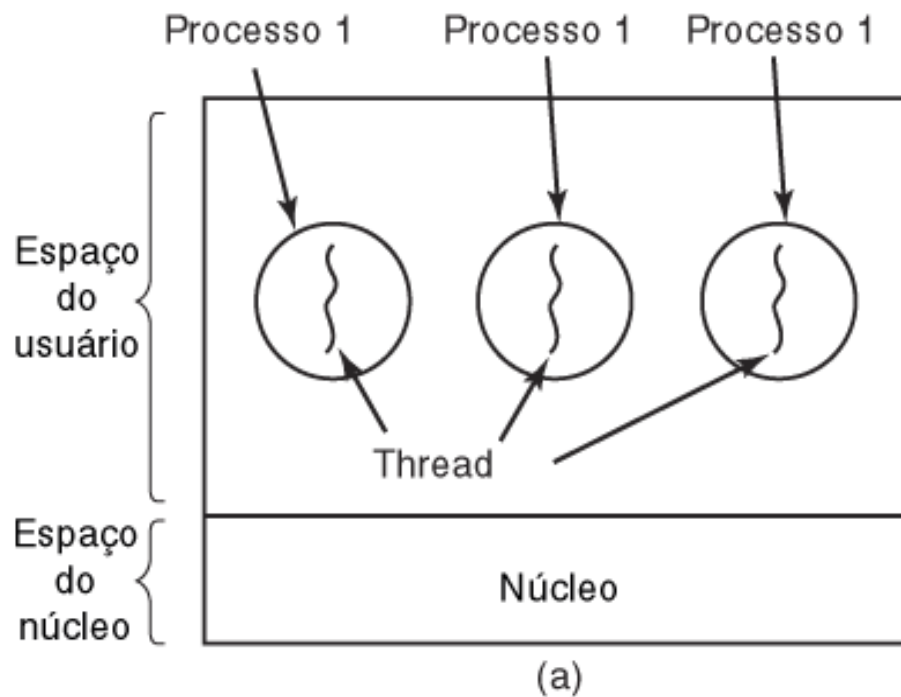
- Todas as *threads* dentro de um processo compartilham o mesmo espaço de endereçamento.
- *Threads* no mesmo processo compartilham:
 - Instruções do processo.
 - Dados
 - Descritores de arquivos abertos
 - Sinais e tratadores de sinais
 - Diretório de trabalho
 - Id do usuário e do grupo

Threads

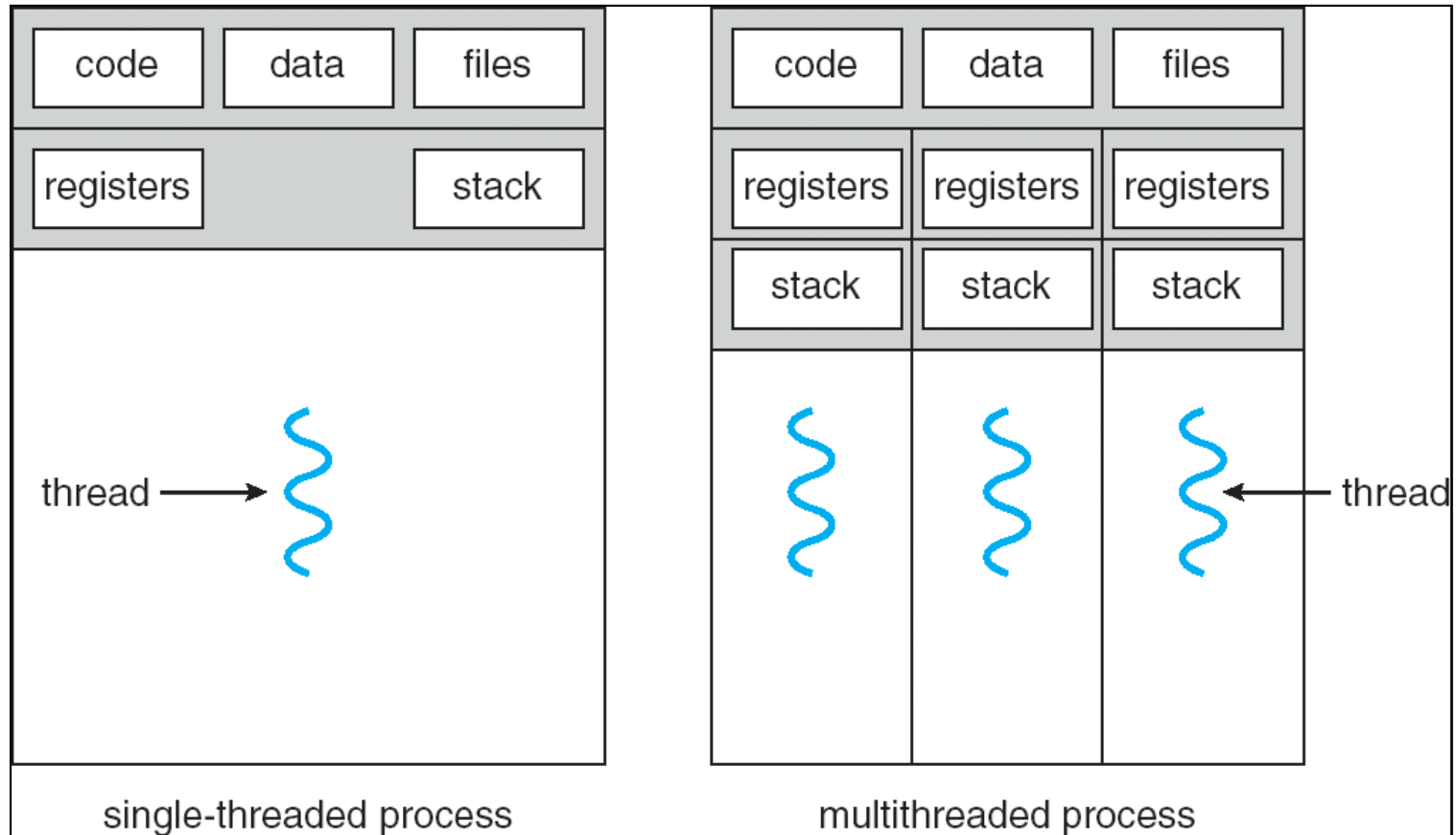
- Cada *thread* tem um único:
 - *Thread* ID
 - Conjunto de registradores, *stack pointer*
 - Pilha para variáveis locais e endereços de retorno
 - Máscara de sinais
 - Prioridade
 - Valor de retorno: **errno**
 - As funções da biblioteca *pthread* retornam "0" se OK.

Processos de único e múltiplos *threads*

- (a) Três processos cada um com um *thread*
- (b) Um processo com três *threads*



Processos de único e múltiplos *threads*

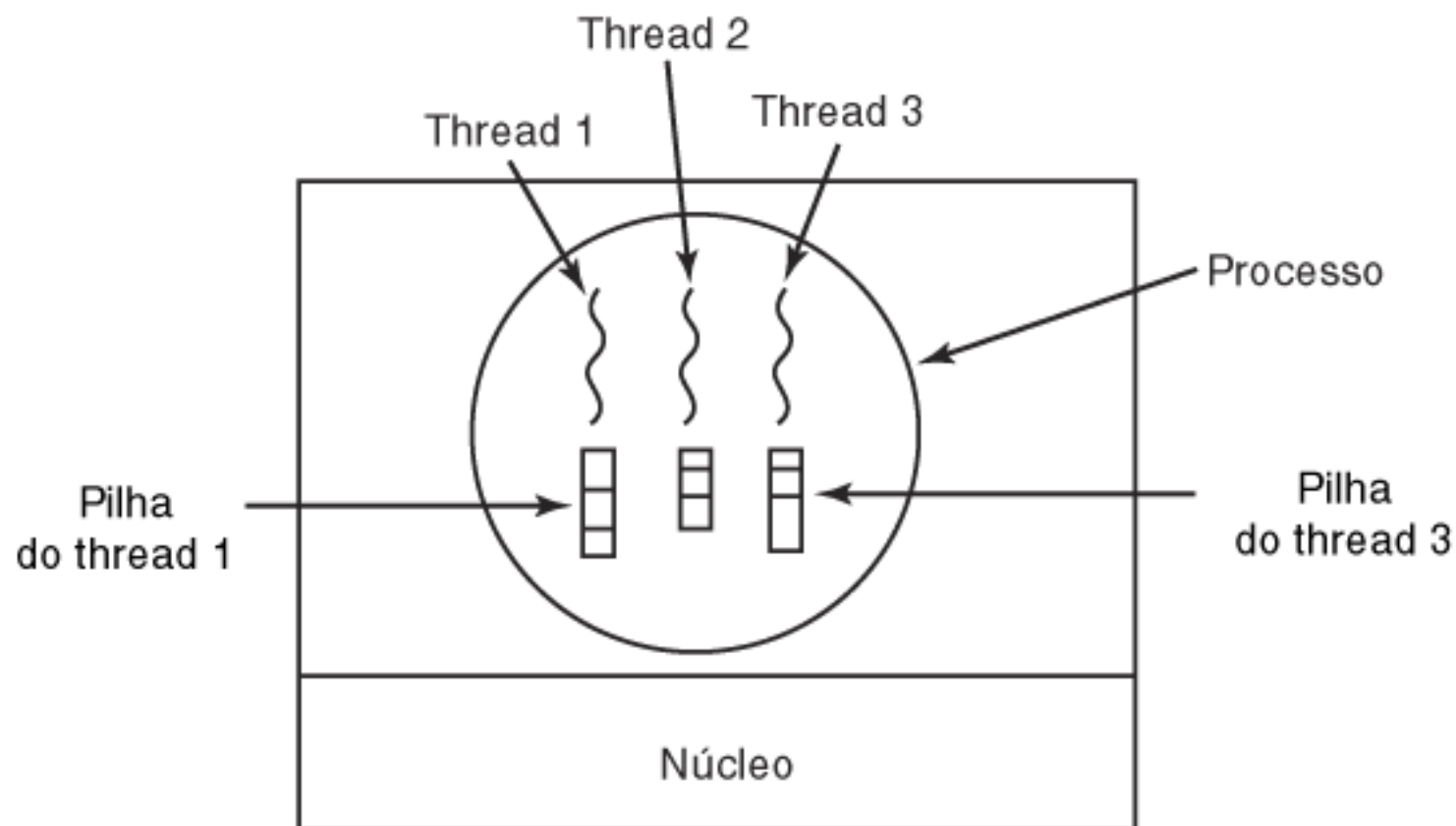


Cada processo tem uma *thread* principal

Se for um processo *multithreaded* terá a *thread* principal e as *threads* que criar

Controle e estrutura dos *threads*

- Cada *thread* tem sua própria pilha

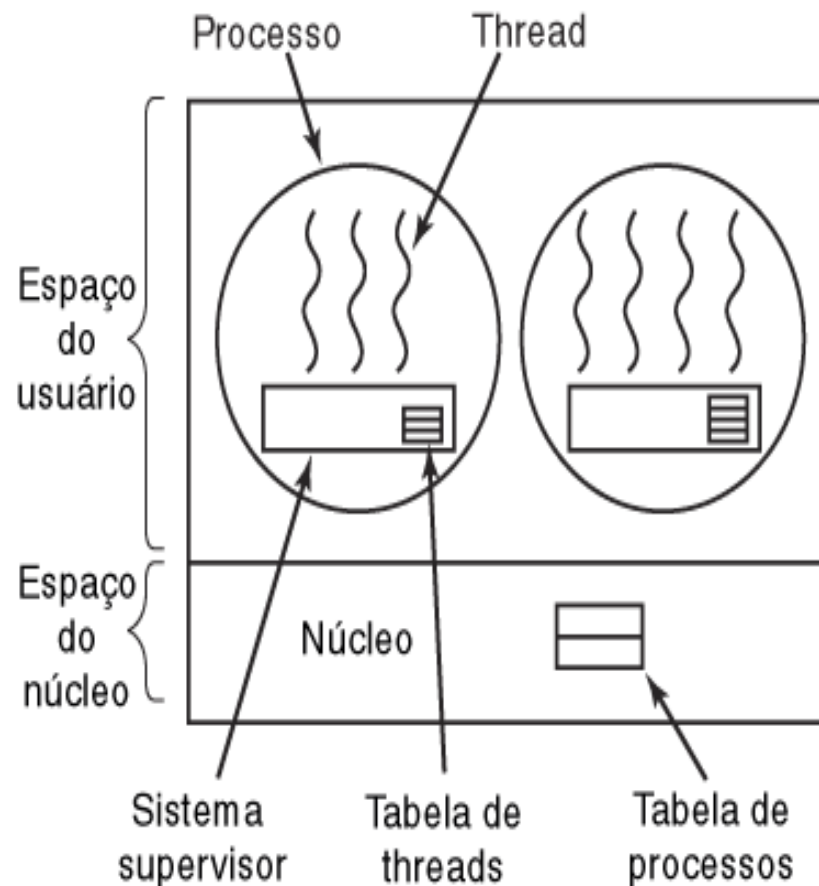
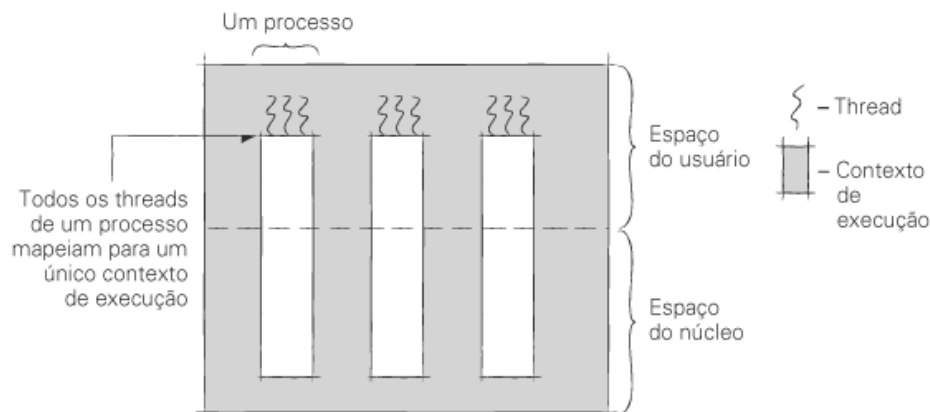


Threads* de usuário e de *kernel

- ***Threads* do usuário** – Gerenciamento de *thread* feito pela biblioteca de *threads* em nível de usuário.
- ***Threads* do kernel** - *Threads* admitidos diretamente pelo *kernel*.

Threads no nível de usuário

- Implementado e utilizado através de pacotes ou bibliotecas



Threads no nível de usuário

- Os *threads* de usuário executam operações de suporte a *threads* no espaço do usuário.
 - Isso significa que os *threads* são criados por bibliotecas em tempo de execução que não podem executar instruções privilegiadas nem acessar as primitivas do núcleo diretamente.
- Implementação de *thread* de usuário
 - Mapeamentos de *thread* muitos-para-um
 - O sistema operacional mapeia todos os *threads* de um processo *multithread* para um único contexto de execução.

Threads no nível de usuário

– Vantagens

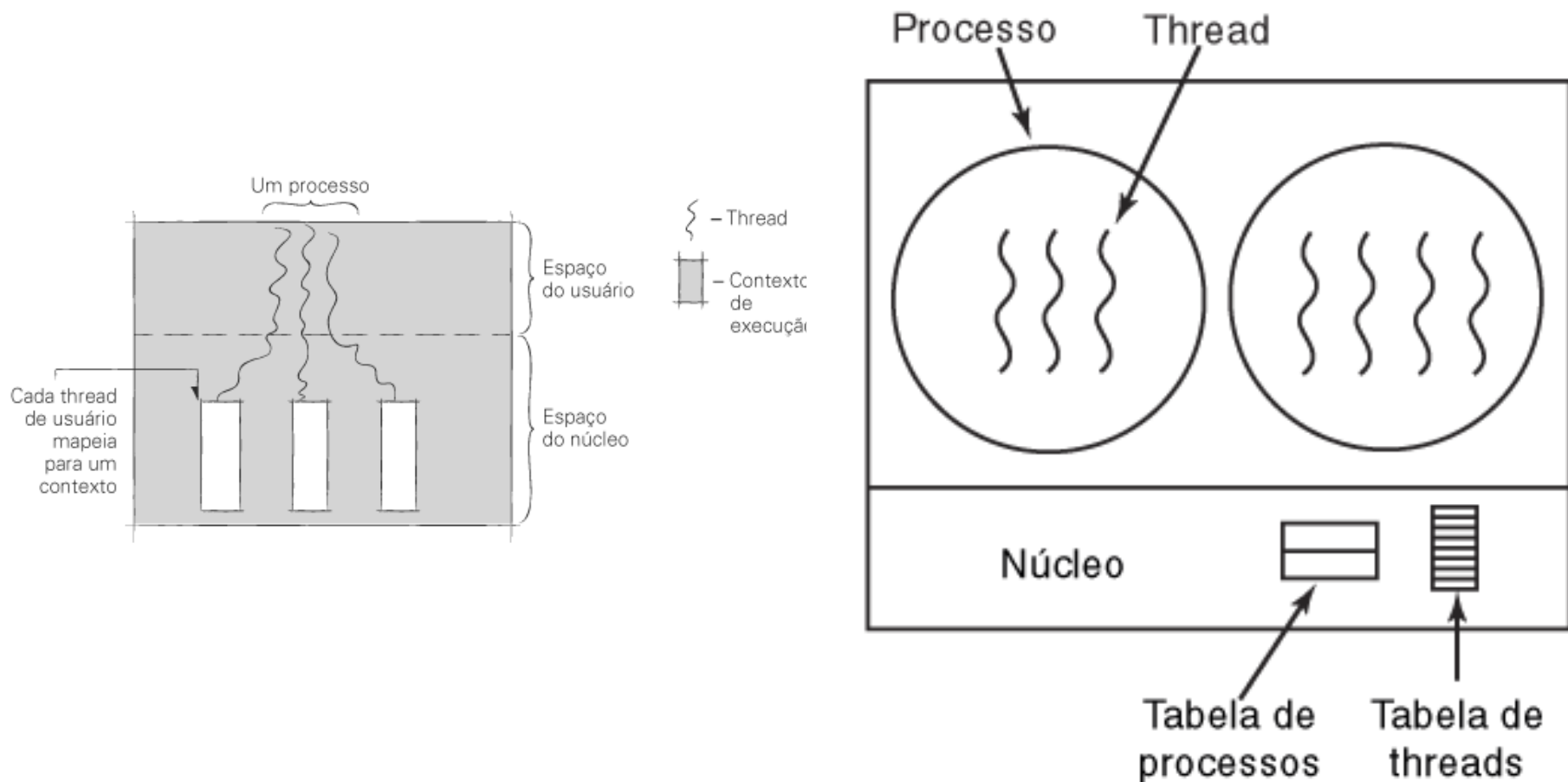
- As bibliotecas de usuário podem escalonar seus *threads* para otimizar o desempenho.
- A sincronização é realizada fora do núcleo, e isso evita chaveamento de contexto.
- É mais portátil.

– Desvantagens

- O núcleo considera o processo *multithread* como um único *thread* de controle.
- Isso pode fazer com que o desempenho fique abaixo do ideal se um *thread* requisitar uma operação E/S.
- Não pode ser escalonado para executar em múltiplos processadores ao mesmo tempo.

Threads: nível do SO (kernel)

Um pacote de *threads* gerenciado pelo núcleo

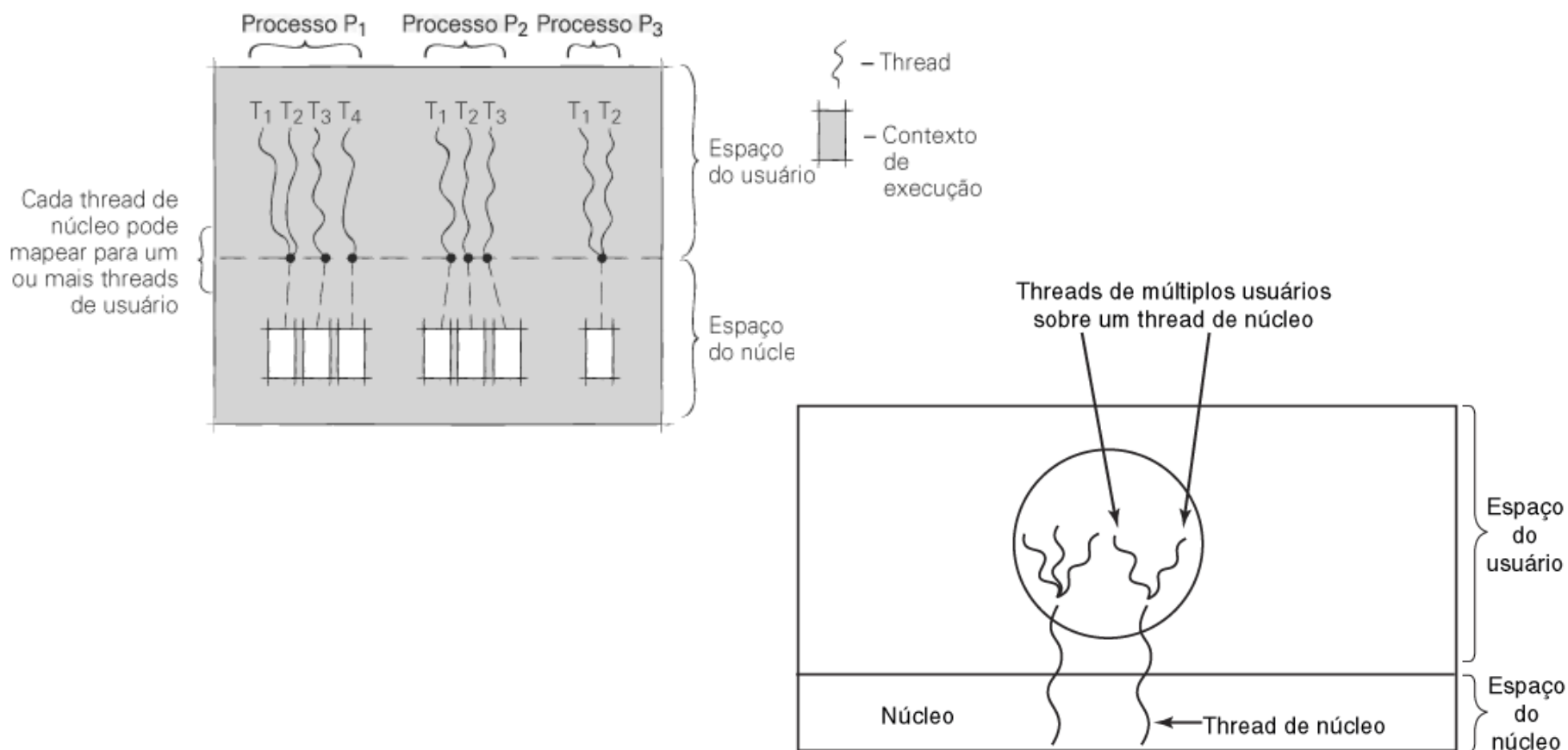


Threads: nível do SO (kernel)

- Os *threads* de núcleo tentam resolver as limitações dos *threads* de usuário mapeando cada *thread* para o seu próprio contexto de execução.
 - O *thread* de núcleo oferece mapeamento de *thread* um-para-um.
 - **Vantagens:** maior escalabilidade, interatividade e rendimento.
 - **Desvantagens:** sobrecarga decorrente do chaveamento de contexto e menor portabilidade em virtude de as APIs serem específicas ao sistema operacional.
- Os *threads* de núcleo nem sempre são a solução ideal para as aplicações.

Implementações Híbridas

- Multiplexação de *threads* de usuário sobre *threads* de núcleo



Implementações Híbridas

- Implementação da combinação de *threads* de usuário e de núcleo
 - Mapeamento de *threads* muitos-para-muitos (mapeamento de *threads* m-to-n)
 - O número de *threads* de usuário e de núcleo tem de ser o mesmo.
 - Em comparação com os mapeamentos de threads um-para-um, esse mapeamento consegue reduzir a sobrecarga implementando o reservatório de *threads*.
- *Threads* operários
 - *Threads* de núcleo persistentes que ocupam o reservatório de *threads*.
 - Os *threads* operários melhoram o desempenho em ambientes em que os *threads* são criados e destruídos com frequência.
 - Cada novo *thread* é executado por um *thread* operário.

Implementações Híbridas

- Ativação de escalonador
 - Técnica que permite que uma biblioteca de usuário escalone seus *threads*.
 - Ocorre quando o SO chama uma biblioteca de *threads* de usuário para determinar se algum de seus *threads* precisam ser reescalonados.

Threads de kernel

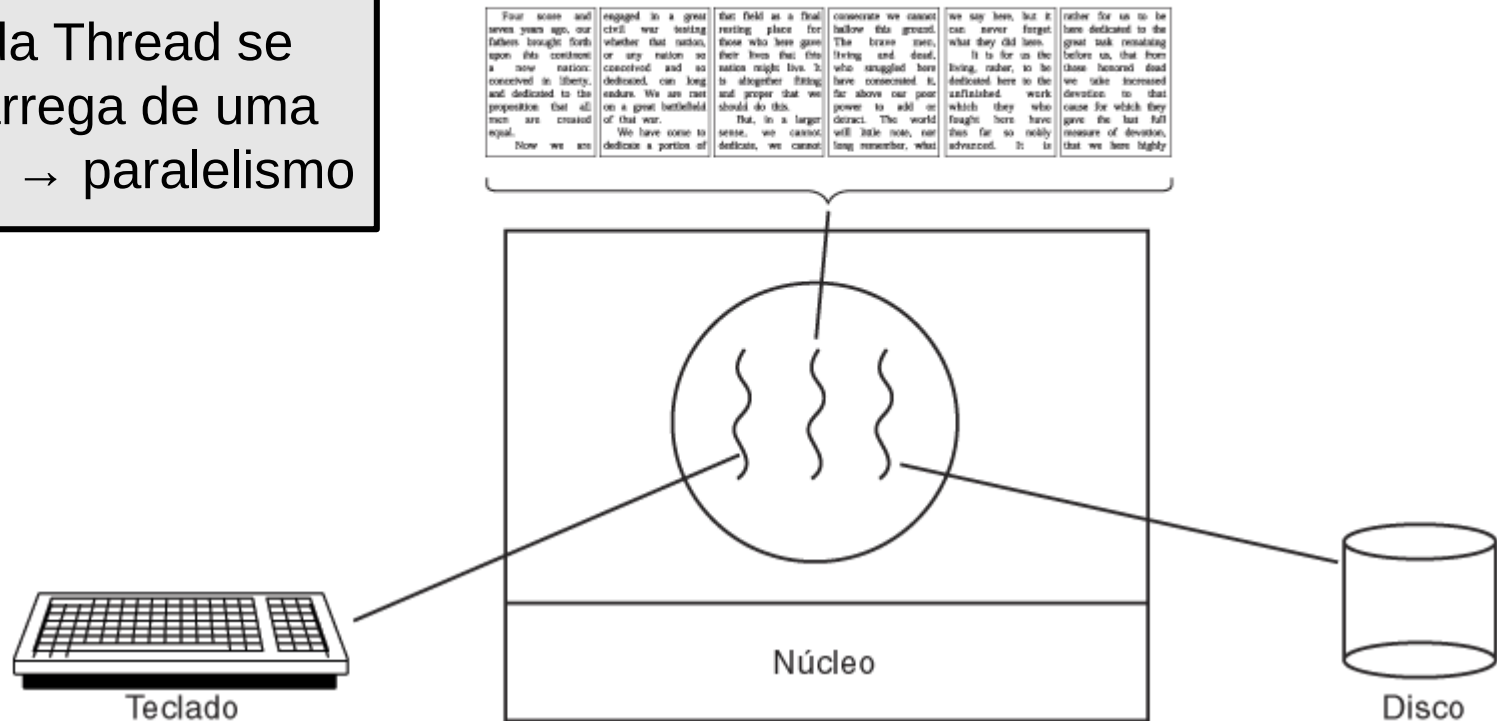
- Exemplos
 - Windows XP/2000
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS Xc

Benefícios

- Responsividade
- Compartilhamento de recursos
- Economia
- Utilização de arquiteturas de MP

Uso de *Threads*

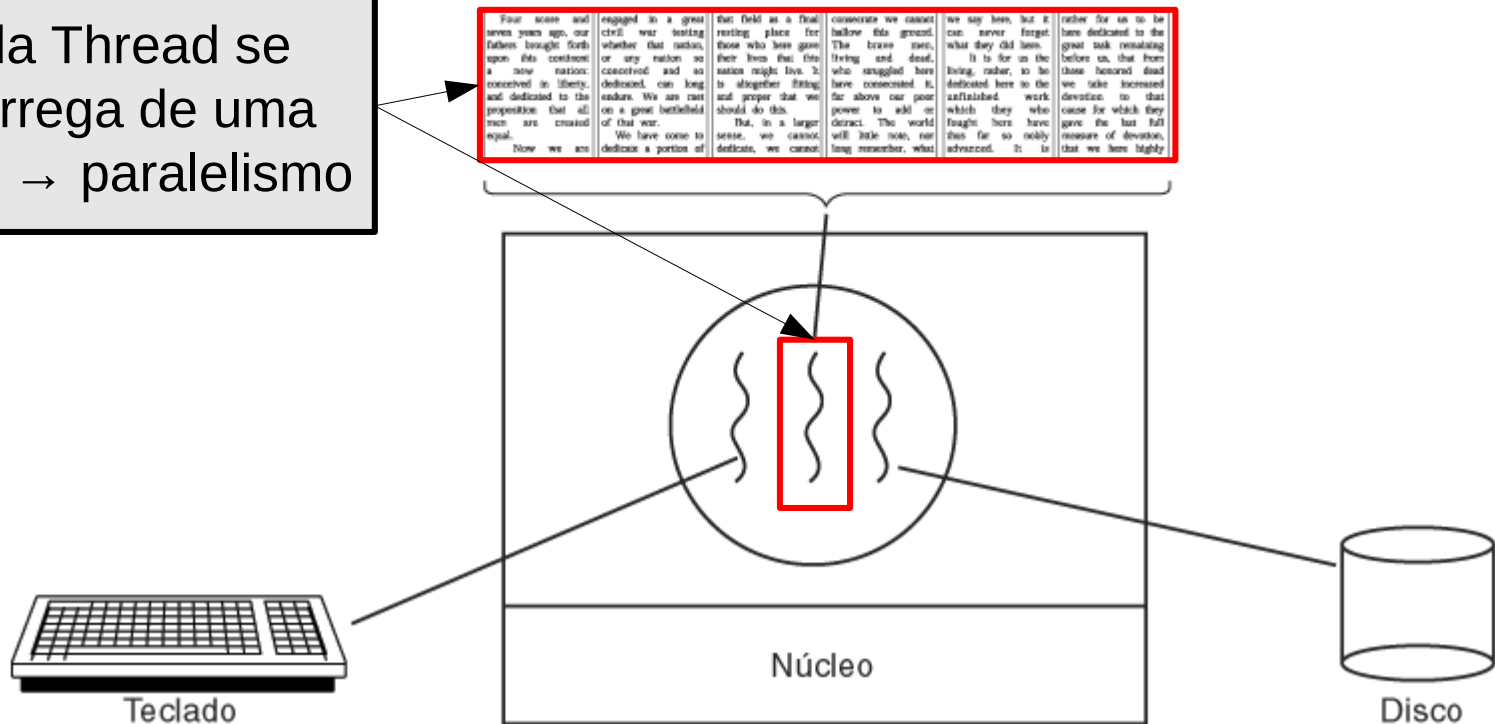
Cada Thread se encarrega de uma tarefa → paralelismo



Um processador de texto com três *threads*

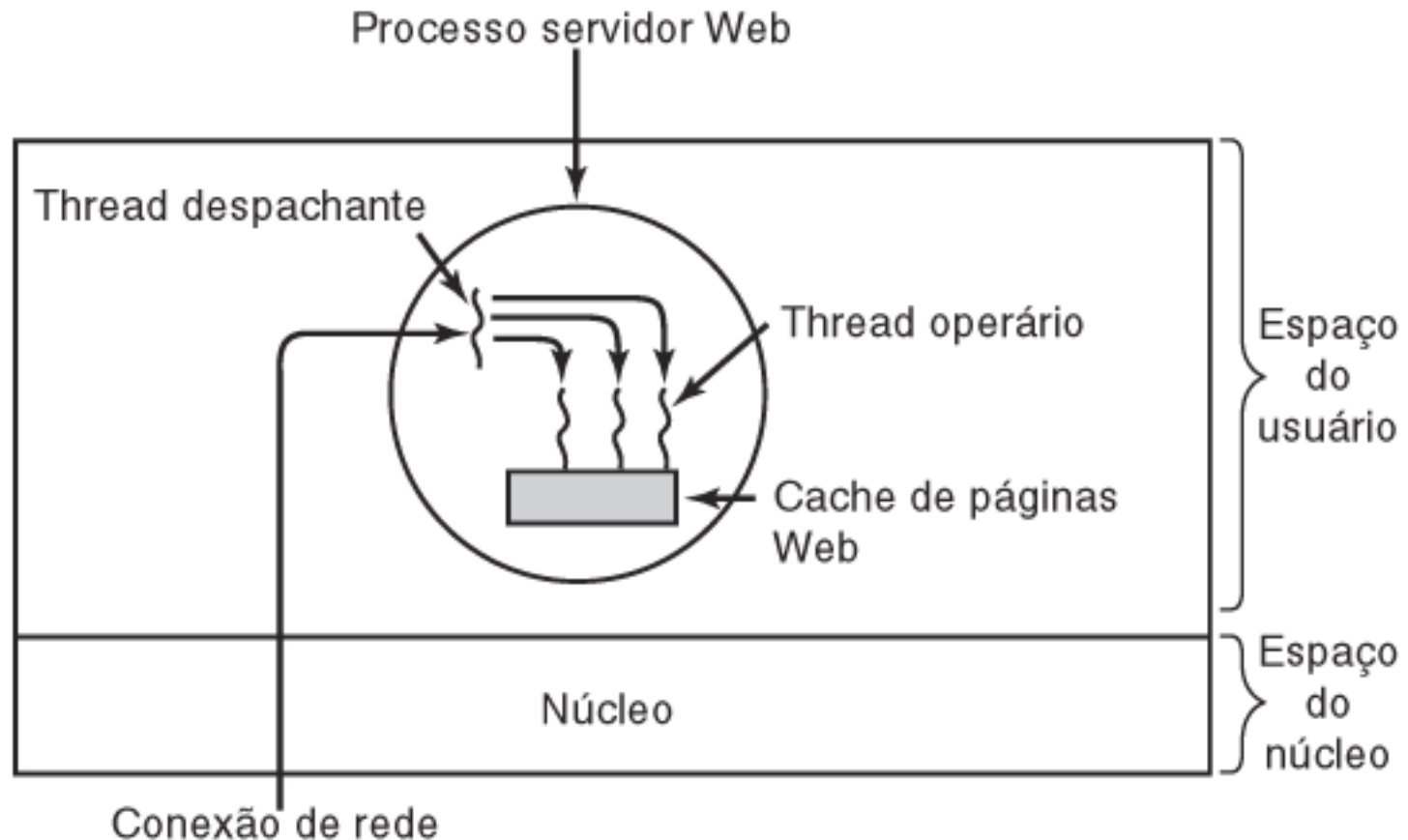
Uso de *Threads*

Cada Thread se encarrega de uma tarefa → paralelismo



Um processador de texto com três *threads*

Uso de *Threads*



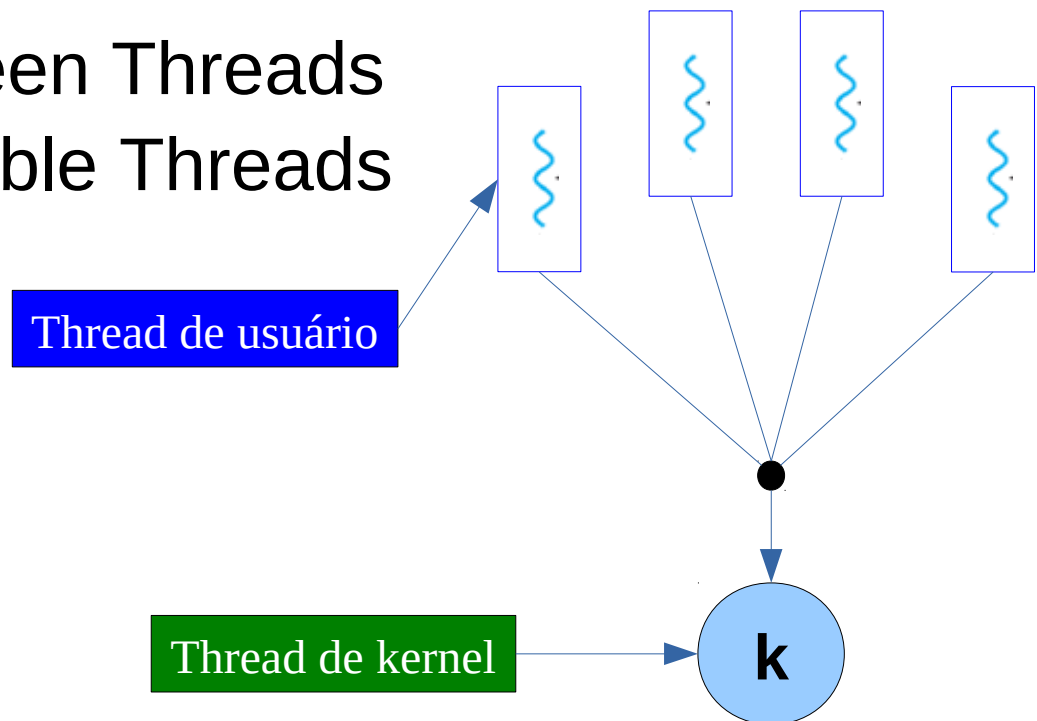
Um servidor web com múltiplos threads

Modelos *Multithreading*

- Mapeamento entre *threads* de usuário e *threads* de *kernel*:
 - Muitos-para-um
 - Um-para-um
 - Muitos-para-muitos

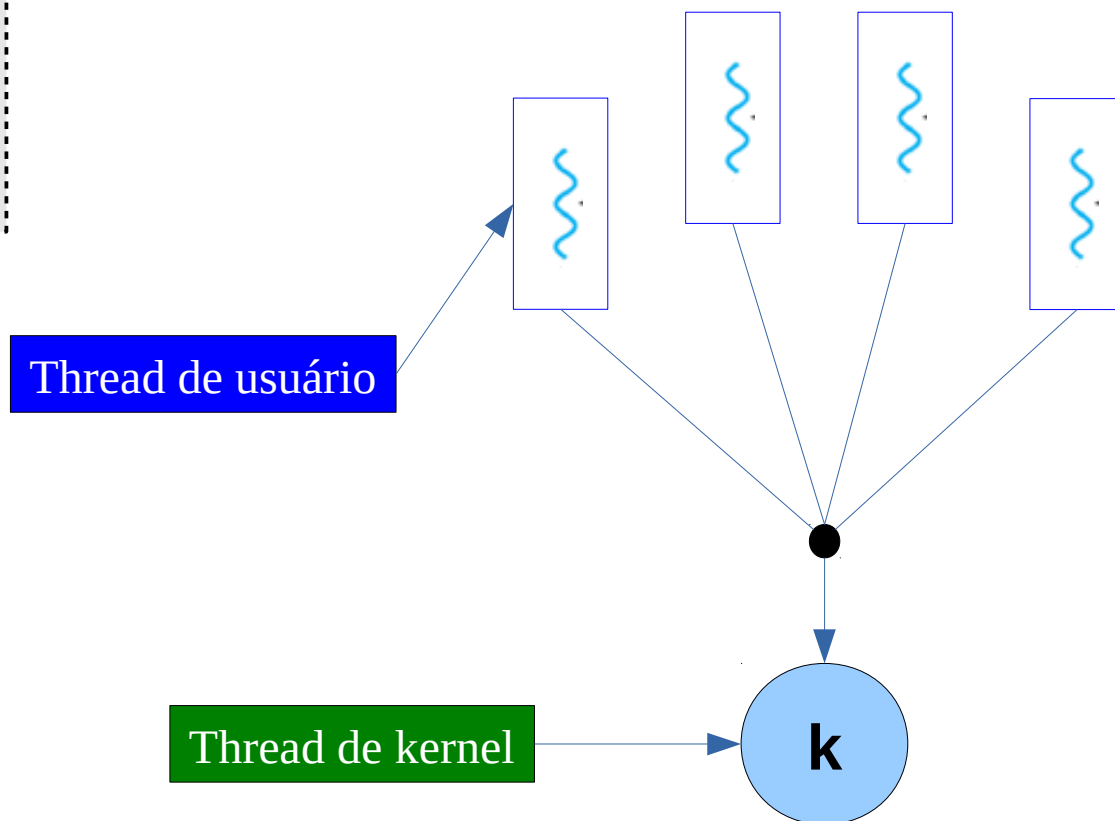
Modelo muitos-para-um

- Muitos *threads* em nível de usuário mapeados para único *thread* do *kernel*
- Exemplos:
 - Solaris Green Threads
 - GNU Portable Threads



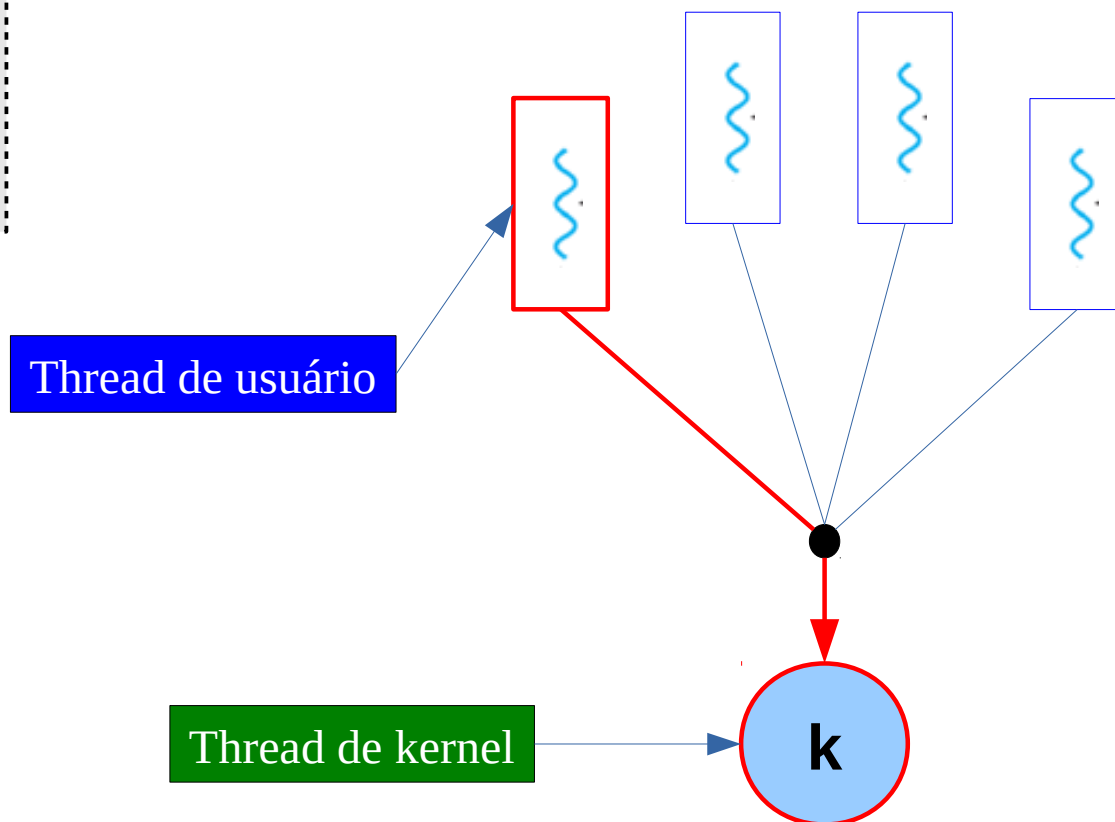
Modelo muitos-para-um

Threads do usuário mapeadas em uma *thread* do kernel



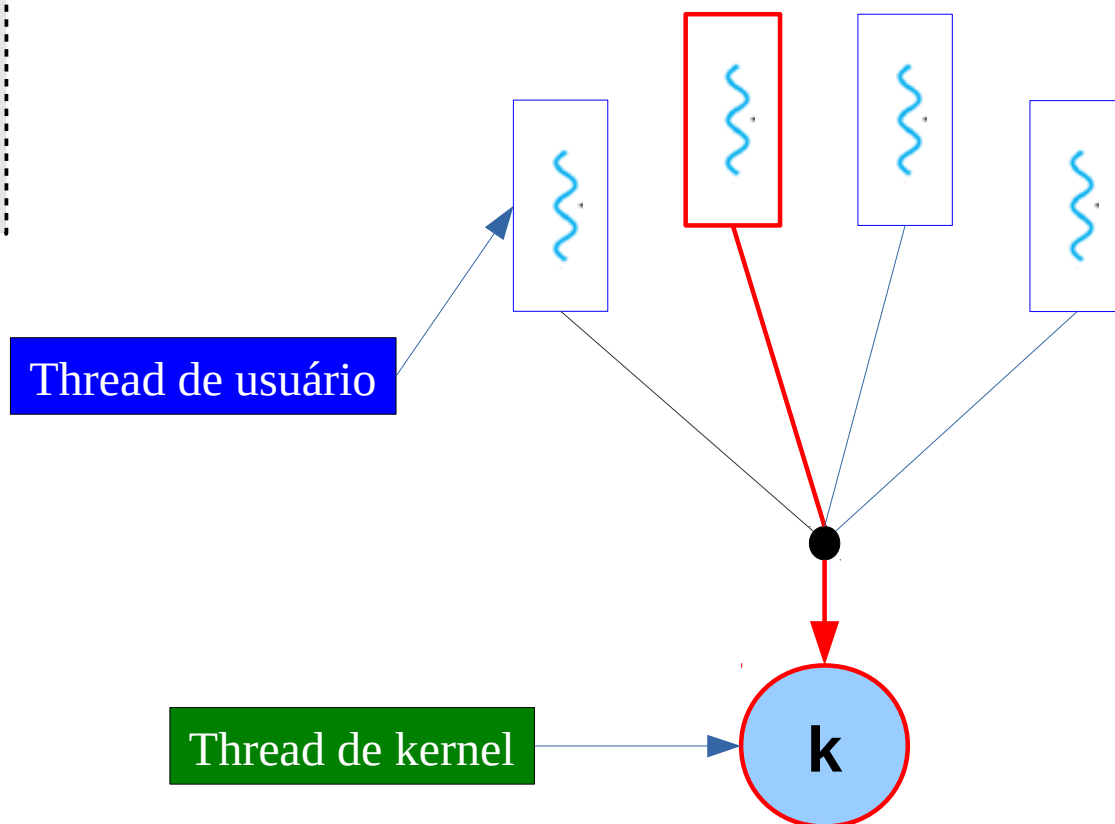
Modelo muitos-para-um

Threads do usuário mapeadas em uma *thread* do kernel



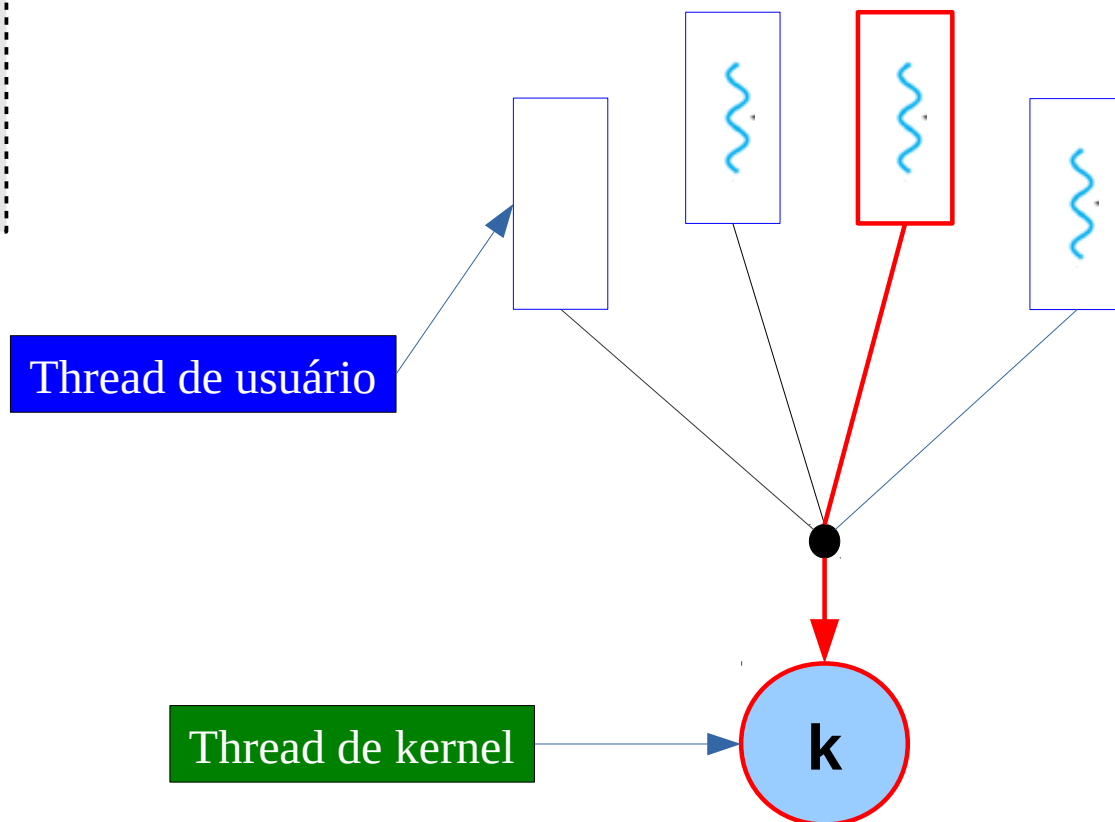
Modelo muitos-para-um

Threads do usuário mapeadas em uma *thread* do kernel



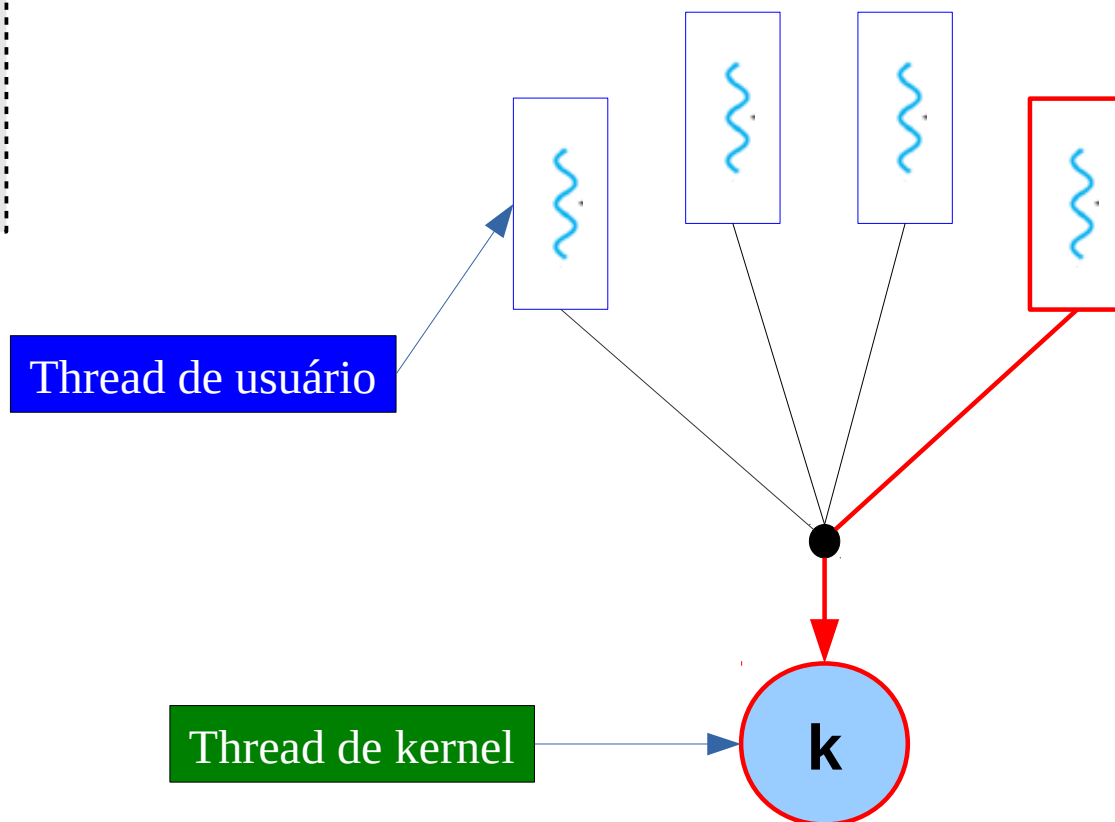
Modelo muitos-para-um

Threads do usuário mapeadas em uma *thread* do kernel



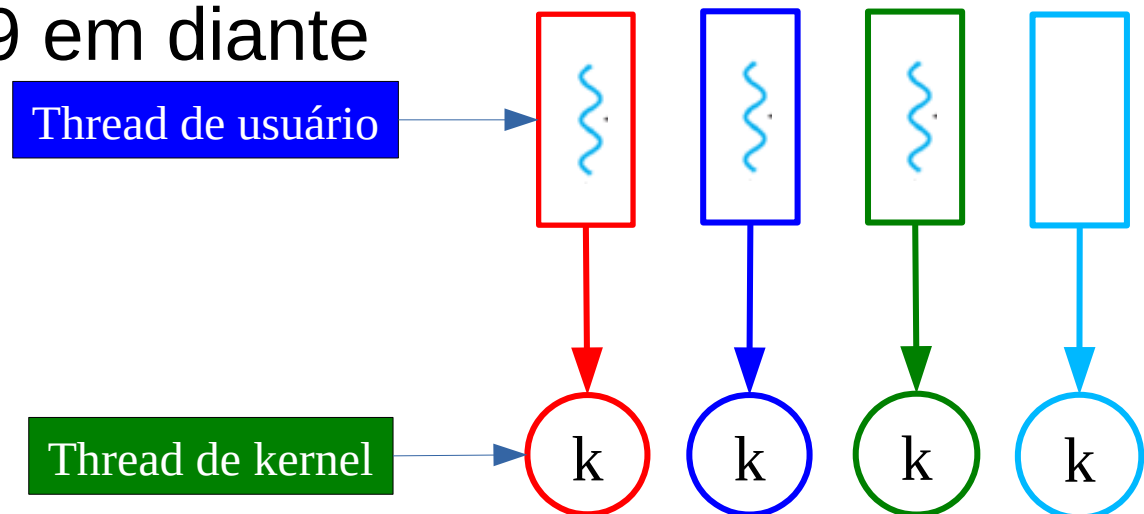
Modelo muitos-para-um

Threads do usuário mapeadas em uma *thread* do kernel



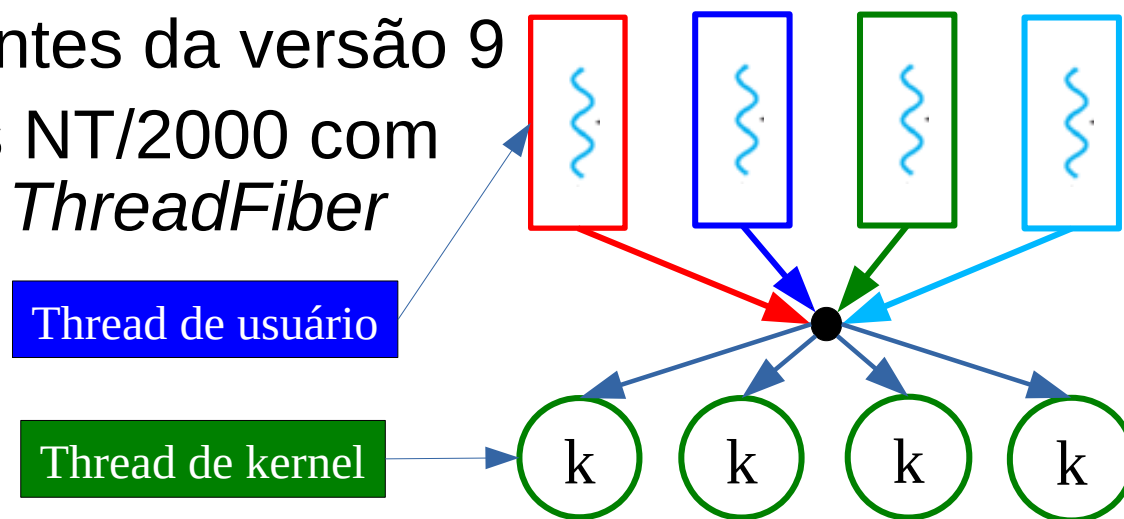
Modelo um-para-um

- Cada thread em nível de usuário é mapeado para thread do kernel
- Exemplos
 - Windows NT/XP/2000
 - Linux
 - Solaris 9 em diante



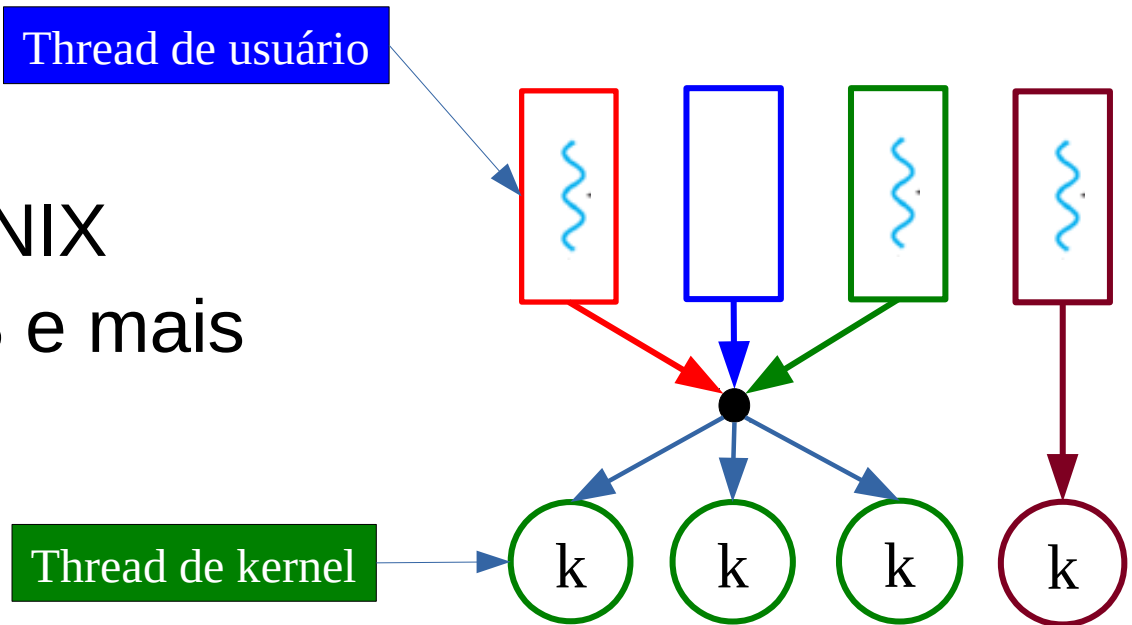
Modelo muitos-para-muitos

- Permite que muitos *threads* em nível de usuário sejam mapeados para muitos *threads* do kernel
- Permite que o sistema operacional crie um número suficiente de *threads* do kernel
 - Solaris antes da versão 9
 - Windows NT/2000 com o pacote *ThreadFiber*



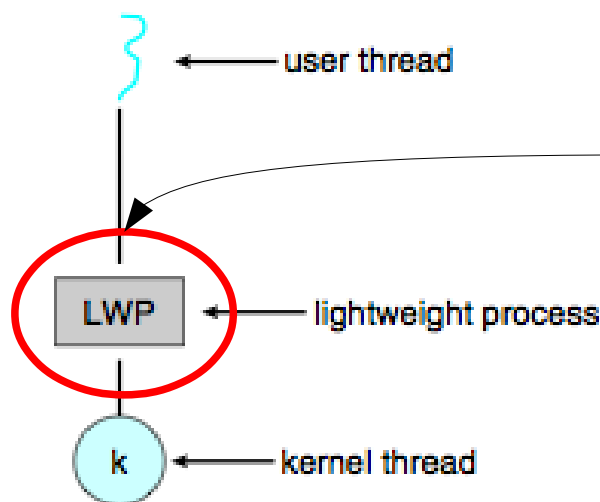
Modelo de dois níveis

- Semelhante a M:M, exceto por permitir que um *thread* do usuário seja ligado ao *thread* do kernel
- Exemplos
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8 e mais antigos



Ativações do escalonador

- Os modelos M:M e de dois níveis exigem comunicação para manter o número apropriado de *threads* de kernel alocados à aplicação.
- Ativações do escalonador oferece **upcalls** – um mecanismo de comunicação do *kernel* para a biblioteca de *threads*
- Essa comunicação permite que uma aplicação mantenha o número correto de *threads* do *kernel*



Estrutura de dados intermediária, processo leve, funciona como um processador virtual

Tratamento de Sinal

- Sinais são usados em sistemas UNIX para notificar um processo de que ocorreu um evento em particular
- Um **manipulador de sinal** é usado para processar sinais
 - Sinal é gerado por evento em particular
 - Sinal é entregue a um processo
 - Sinal é tratado
- Opções de projeto:
 - Entregar o sinal ao *thread* ao qual o sinal se aplica
 - Entregar o sinal a cada *thread* no processo
 - Entregar o sinal a certos *threads* no processo
 - Atribuir uma área específica para receber todos os sinais para o processo

Entrega do Sinal

- **Dois tipos de sinal**
 - **Síncrono:**
 - Resulta diretamente da execução de um programa.
 - Pode ser emitido (entregue) para um *thread* que esteja sendo executado no momento.
 - **Assíncrono:**
 - Resulta de um evento em geral não relacionado com a instrução corrente.
 - A biblioteca de *threads* precisa identificar todo receptor de sinal para que os sinais assíncronos sejam emitidos (entregues) devidamente.
- Todo *thread* normalmente está associado a um conjunto de sinais pendentes que são emitidos (entregues) quando ele é executado.
- O *thread* pode mascarar todos os sinais, exceto aqueles que deseja receber.

POSIX

- Uma API padrão POSIX (IEEE 1003.1c) para criação e sincronismo de *thread*
- A API especifica o comportamento da biblioteca de *threads*, a implementação fica para o desenvolvimento da biblioteca
- Comum em sistemas operacionais UNIX (Solaris, Linux, Mac OS X)

POSIX e Pthreads

- Os *threads* que usam a API de *thread* POSIX são chamados de Pthreads.
- A especificação POSIX determina que os registradores do processador, a pilha e a máscara de sinal sejam mantidos individualmente para cada *thread*.
- A especificação POSIX especifica como os sistemas operacionais devem emitir sinais a Pthreads, além de especificar diversos modos de cancelamento de *thread*.

Threads no Linux

- Linux se refere a eles como *tarefas* ao invés de *threads*.
 - O Linux aloca o mesmo tipo de descritor para processos e tarefas.
 - Para criar tarefas-filhas, o Linux usa a chamada ***fork***, baseada no Unix.
 - Para habilitar os *threads*, o Linux oferece uma versão modificada, denominada ***clone***.

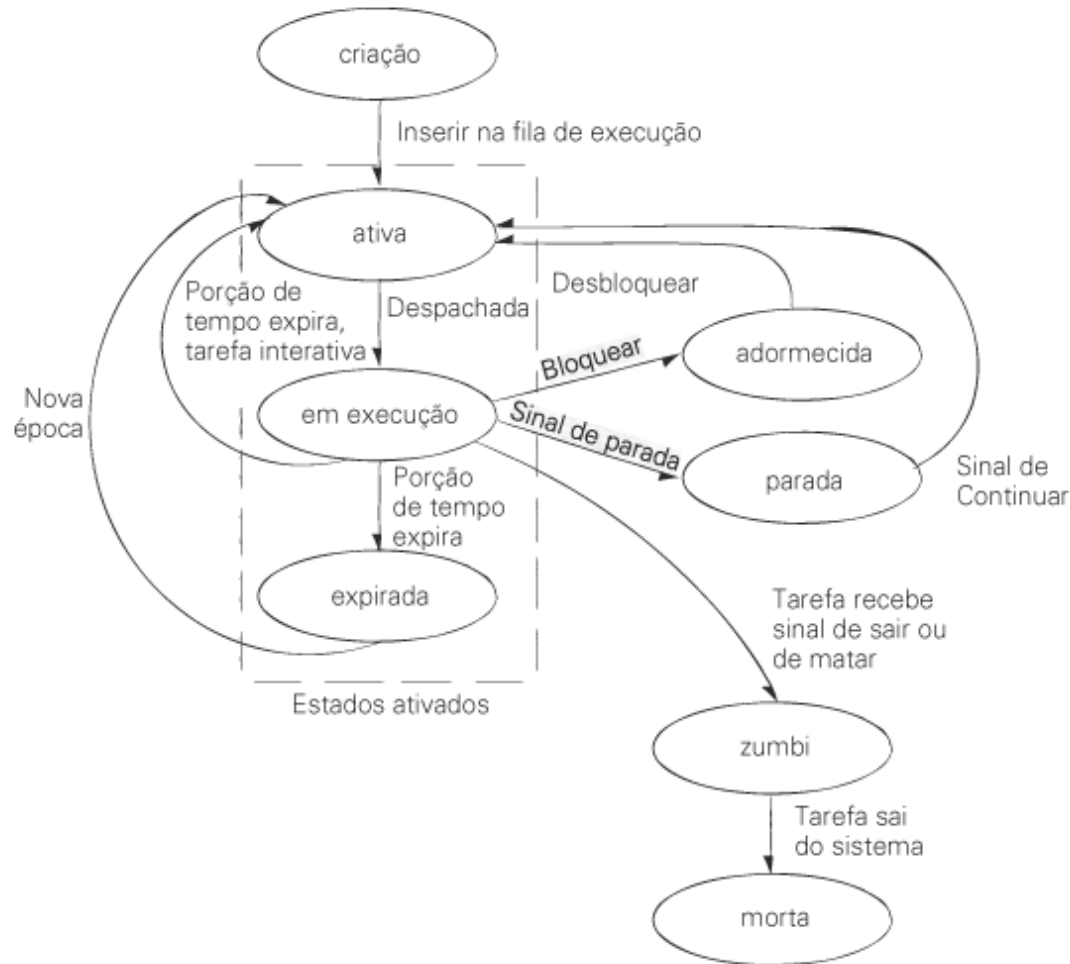
Threads no Linux

- A criação de *thread* é feita por meio da chamada do sistema **clone()**
 - Clone aceita argumentos que determinam os recursos que devem ser compartilhados com a tarefa-filha.
 - **clone()** permite que uma tarefa filha compartilhe o espaço de endereços da tarefa pai (processo)

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Threads no Linux

Diagrama de transição de estado de tarefa do Linux.



Threads no Linux

- ***pthread_create***: Cria uma nova *thread*.

```
int pthread_create(pthread_t * thread,  
                  const pthread_attr_t * attr,  
                  void * (*start_routine)(void *),  
                  void *arg);
```

- **Retorno:**
 - 0 se conseguiu criar a *thread*.
 - Ou um código de erro.
 - Veja: http://linux.die.net/man/3/pthread_create

Threads no Linux

- ***pthread_join***: Aguarda pelo término de outra *thread*.

```
int pthread_join(pthread_t th, void **thread_return);
```

- A *thread* principal que cria outras *threads* deve aguardar o término das *threads* filhas.
- Pois pode acontecer da *thread* principal terminar antes.
 - Veja: http://linux.die.net/man/3/pthread_join

Threads no Linux

- ***pthread_exit***: Termina a *thread*.

```
void pthread_exit(void *retval);
```

- Veja: http://linux.die.net/man/3/pthread_exit

Tutorial:

<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>

Threads no Linux

- ***pthread_cancel***: Cancela a execução de uma *thread*.

```
int pthread_cancel(pthread_t thread);
```

- Veja: http://linux.die.net/man/3/pthread_cancel

Tutorial:

<https://computing.llnl.gov/tutorials/pthreads>

Threads no Linux

- ***pthread_kill***: Envia um sinal para a *thread*.

```
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);
```

- Veja: http://linux.die.net/man/3/pthread_kill

Tutorial:

<https://computing.llnl.gov/tutorials/pthreads>

Threads no Linux

exemplo-thread.c

```
gcc -lpthread exemplo-pthread.c -o exemplo-pthread.exe
```

```
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <pthread.h>
8
9  void *print_message_function( void *ptr );
10
11 int main() {
12     /* Declara duas threads */
13     pthread_t thread1, thread2;
14     char *message1 = "Olá eu sou a Thread 1";
15     char *message2 = "Olá eu sou a Thread 2";
16     int iret1, iret2;
17
18     /* Cria duas threads independentes, cada uma irá executar a função */
19     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
20     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
21
22     /* Aguarda até que todas as threads completem antes de continuar. */
23     /* Pode acontecer de executar algo que termine o processo/thread principal antes das threads terminarem */
24     pthread_join(thread1, NULL);
25     pthread_join(thread2, NULL);
26
27     printf("Thread 1 retornou: %d\n", iret1);
28     printf("Thread 2 retornou: %d\n", iret2);
29
30     exit(0);
31 }
32
33 void *print_message_function( void *ptr )
34 {
35     char *message;
36     message = (char *) ptr;
37     printf("%s \n", message);
38 }
39
```


Threads no Linux

```
gcc -lpthread exemplo-pthread.c -o exemplo-pthread.exe
```

```
exemplo-pthread.c x
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <pthread.h>
8
9 /* Assinatura da função que a thread irá executar. */
10 void *print_message_function(void *ptr);
11
12 int main(int argc, char *argv[]) {
13     /* Declara duas threads */
14     pthread_t thread1, thread2;
15
16     /* Declara as mensagens que as threads irão imprimir. */
17     char *message1 = "Olá! Eu sou a Thread 1.";
18     char *message2 = "Olá! Eu sou a Thread 2.";
19
20     /* Variáveis para armazenar o retorno. */
21     int iret1, iret2;
22
23     printf("Thread[%lu]: Criando as threads...\n", (long int) pthread_self());
24     /* Cria duas threads independentes, cada uma irá executar a função */
25     iret1 = pthread_create(&thread1, NULL, print_message_function, (void *)message1);
26
27     /* Se conseguiu criar a thread pthread_create retorna 0. */
28     printf("Thread[%lu]: Criação da Thread 1 [%s]\n", (long int) pthread_self(), ((iret1 == 0)? "OK" : "Erro"));
29
30     iret2 = pthread_create(&thread2, NULL, print_message_function, (void *)message2);
31
32     /* Se conseguiu criar a thread pthread_create retorna 0. */
33     printf("Thread[%lu]: Criação da Thread 2 [%s]\n", (long int) pthread_self(), ((iret2 == 0)? "OK" : "Erro"));
34
35     /* Aguarda até que todas as threads completem antes de continuar. */
36     /* Pode acontecer de executar algo que termine o processo/thread
37     principal antes das threads terminarem */
38     printf("Thread[%lu]: Aguardando o término das threads...\n", (long int) pthread_self());
39     pthread_join(thread1, NULL);
40     pthread_join(thread2, NULL);
41
42     printf("Thread[%lu]: Todas as threads terminaram...\n", (long int) pthread_self());
43     printf("Thread[%lu]: Fui, Tchau!\n", (long int) pthread_self());
44
45     return 0;
46 }
47
48 /* Função que as threads irão executar. */
49 void *print_message_function(void *ptr) {
50     char *message;
51     message = (char *)ptr;
52     printf(" Thread[%lu]: Executando...\n", (long int) pthread_self());
53     printf(" Thread[%lu]: %s\n", (long int) pthread_self(), message);
54     printf(" Thread[%lu]: Terminando...\n", (long int) pthread_self());
55     pthread_exit(0);
56 }
57
```

Git branch: master, index: 21?, working: 6≠ 8× 21?, Line 5, Column 19

Spaces: 2

C

Threads do Windows XP

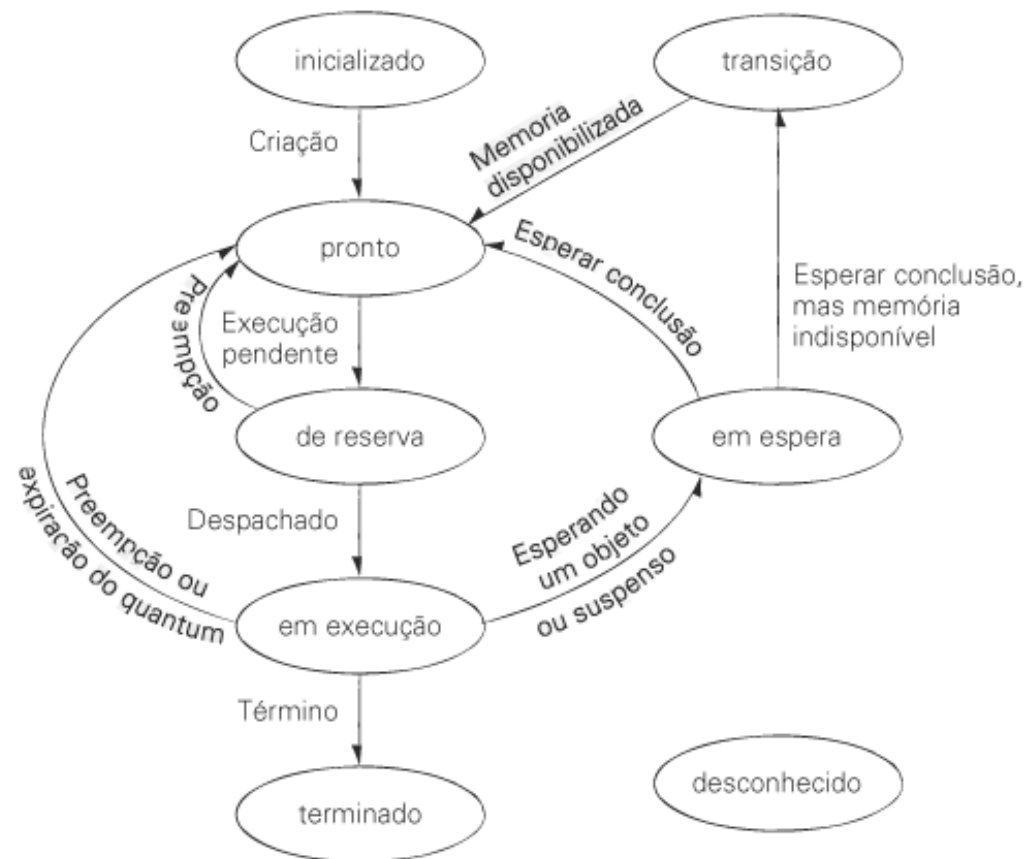
- Implementa o mapeamento um-para-um
- Cada *thread* contém
 - Uma id de *thread*
 - Conjunto de registradores
 - Pilhas de usuário e *kernel* separadas
 - Área privativa de armazenamento de dados
- O conjunto de registradores, pilhas e área de armazenamento privativa são conhecidos como o contexto dos *threads*

Threads do Windows XP

- Os *threads* do Windows XP podem criar fibras.
 - A execução da fibra é escalonada pelo *thread* que a cria, e não pelo escalonador.
- O Windows XP fornece a cada processo um reservatório de *threads* que consiste em inúmeros *threads* operários, que são *threads* de núcleo que executam funções especificadas pelos *threads* de usuário.

Threads do Windows XP

Diagrama de transição de estado de thread do Windows XP.



Threads em Python

```
from threading import  
Thread  
from time import sleep  
  
class Hello(Thread):  
    def run(self):  
        sleep(3);  
        print("Hello World")  
  
thread = Hello()  
thread.start()
```

Por herança

```
from threading import Thread  
from time import sleep  
  
class Hello:  
    def foo(self):  
        sleep(3);  
        print("Hello World")  
  
hello = Hello()  
thread =  
Thread(target=hello.foo)  
thread.start()
```

Por alvo

Atividade

- Implementar o exemplo soma de vetores utilizando *threads* com a biblioteca *pthread*.
- Pesquisar sobre Sinais e seu uso.

Threads em Java

Criando uma subclasse de *java.lang.Thread* e rescrevendo o método *run()*

```
public class ThreadSimples extends Thread {
    public ThreadSimples(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("Feito: " + getName());
    }
}
```

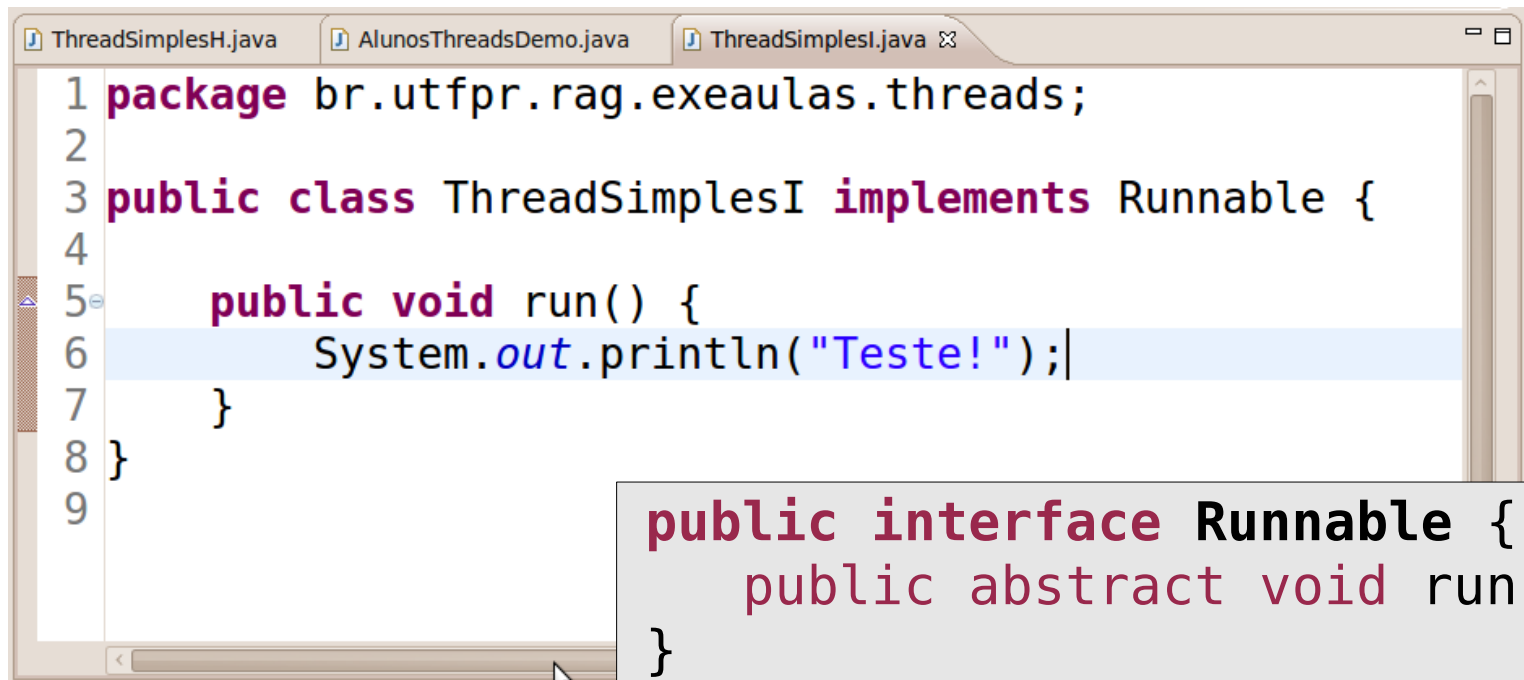
Threads em Java

Uso da classe *ThreadSimples* que foi definida.

```
public class AlunosThreadsDemo {  
    public static void main (String[] args) {  
        new ThreadSimples("Huguinho").start();  
        new ThreadSimples("Zezinho").start();  
        new ThreadSimples("Luizinho").start();  
        new ThreadSimples("Donald").start();  
        new ThreadSimples("Tio Patinhas").start();  
        new ThreadSimples("Capitão Boing").start();  
    }  
}
```


Threads em Java

- *Threads* Java são gerenciados pela JVM
- Como *Threads* podem ser criadas em Java:
 - 1) Implementando a ***interface Runnable***



The screenshot shows a Java IDE with three tabs: ThreadSimpleH.java, AlunosThreadsDemo.java, and ThreadSimpleI.java. The ThreadSimpleI.java tab is active, displaying the following code:

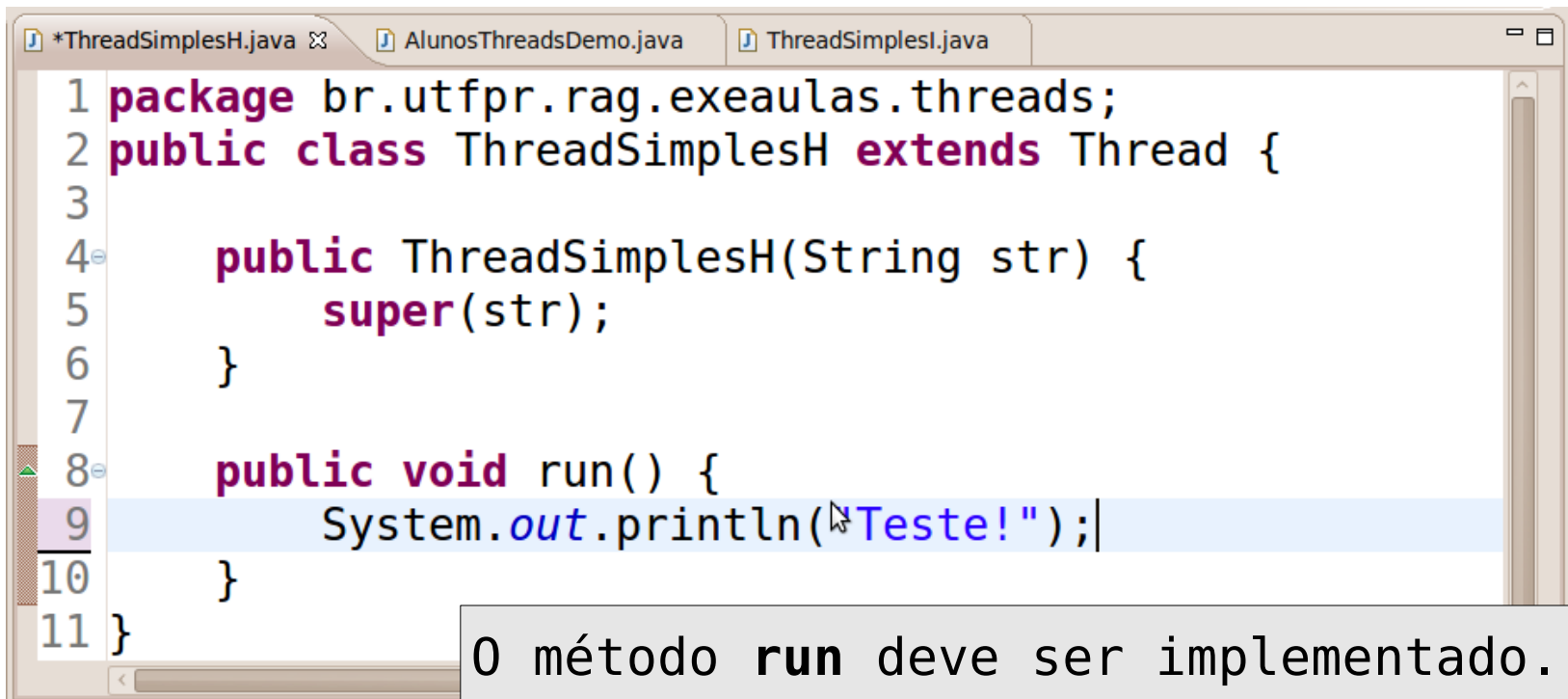
```
1 package br.utfpr.rag.exeaulas.threads;
2
3 public class ThreadSimpleI implements Runnable {
4
5     public void run() {
6         System.out.println("Teste!");
7     }
8 }
9
```

A callout box in the bottom right corner shows the definition of the Runnable interface:

```
public interface Runnable {
    public abstract void run();
}
```

Threads em Java

- Como *Threads* podem ser criadas em Java:
2) Estendendo a **classe Thread** (herança)

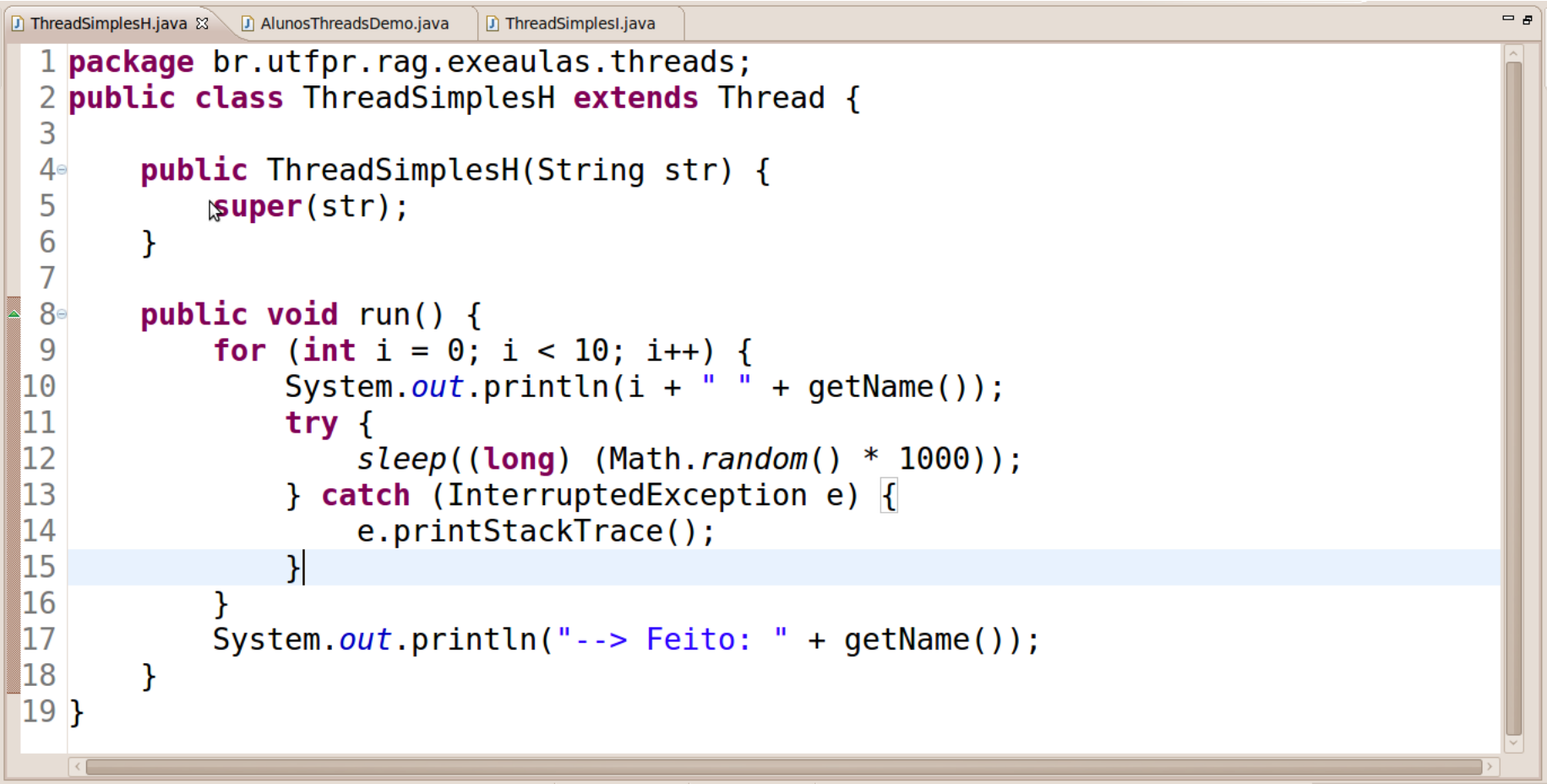


```
1 package br.utfpr.rag.exeaulas.threads;
2 public class ThreadSimplesH extends Thread {
3
4     public ThreadSimplesH(String str) {
5         super(str);
6     }
7
8     public void run() {
9         System.out.println("Teste!");
10    }
11 }
```

O método **run** deve ser implementado.

Threads Java - Programa de exemplo

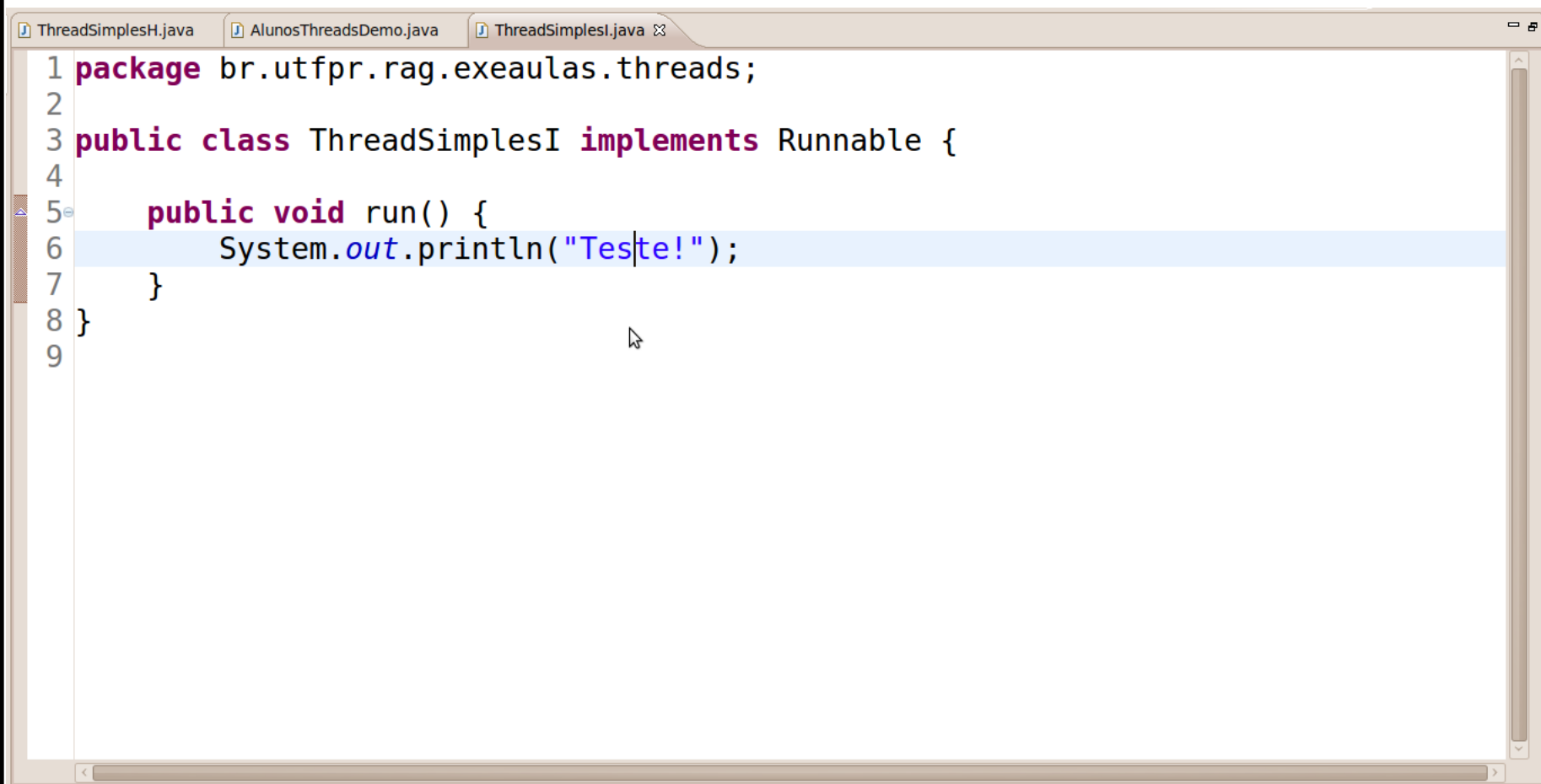
- Estendendo a **classe Thread** (herança)



```
1 package br.utfpr.rag.exeaulas.threads;
2 public class ThreadSimpleH extends Thread {
3
4     public ThreadSimpleH(String str) {
5         super(str);
6     }
7
8     public void run() {
9         for (int i = 0; i < 10; i++) {
10             System.out.println(i + " " + getName());
11             try {
12                 sleep((long) (Math.random() * 1000));
13             } catch (InterruptedException e) {
14                 e.printStackTrace();
15             }
16         }
17         System.out.println("--> Feito: " + getName());
18     }
19 }
```

Threads Java - Programa de exemplo

- Implementando a interface *Runnable*



The screenshot shows a Java IDE with three tabs: ThreadSimplesH.java, AlunosThreadsDemo.java, and ThreadSimplesI.java. The ThreadSimplesI.java tab is active, displaying the following code:

```
1 package br.utfpr.rag.exeaulas.threads;
2
3 public class ThreadSimplesI implements Runnable {
4
5     public void run() {
6         System.out.println("Teste!");
7     }
8 }
9
```

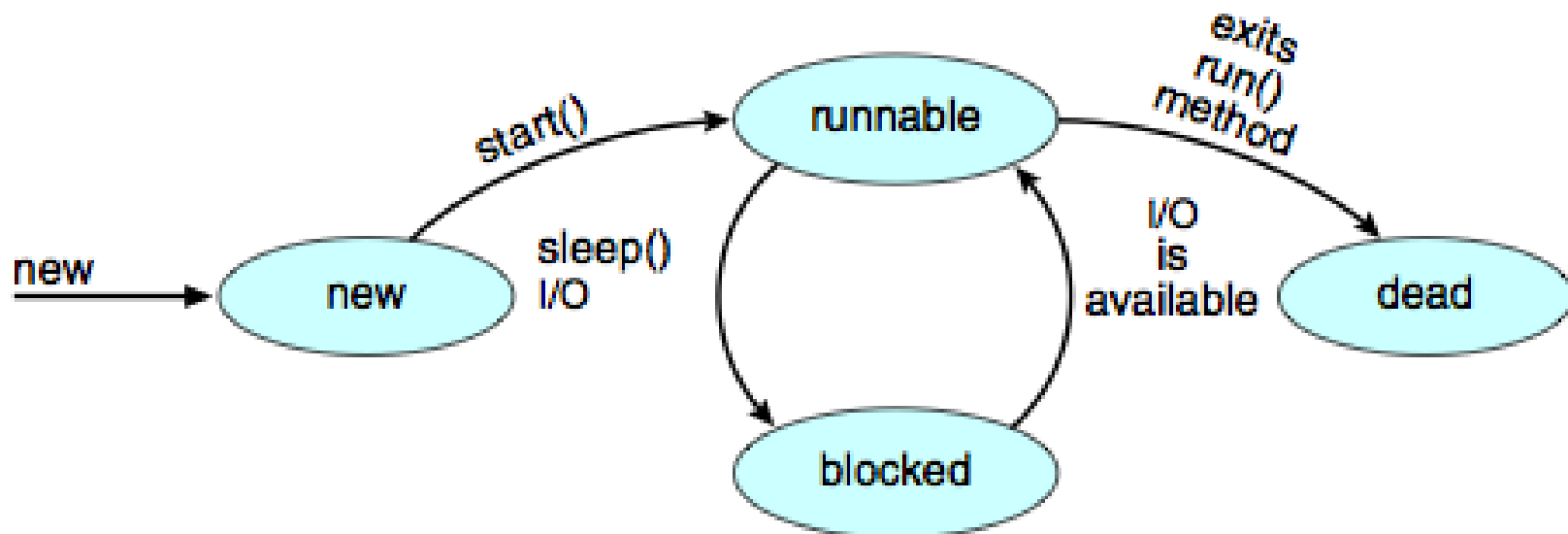
Threads Java - Programa de exemplo

- Usando os exemplos:

A screenshot of a Java IDE window with three tabs: ThreadSimpleH.java, *AlunosThreadsDemo.java, and ThreadSimpleI.java. The code in the active tab is as follows:

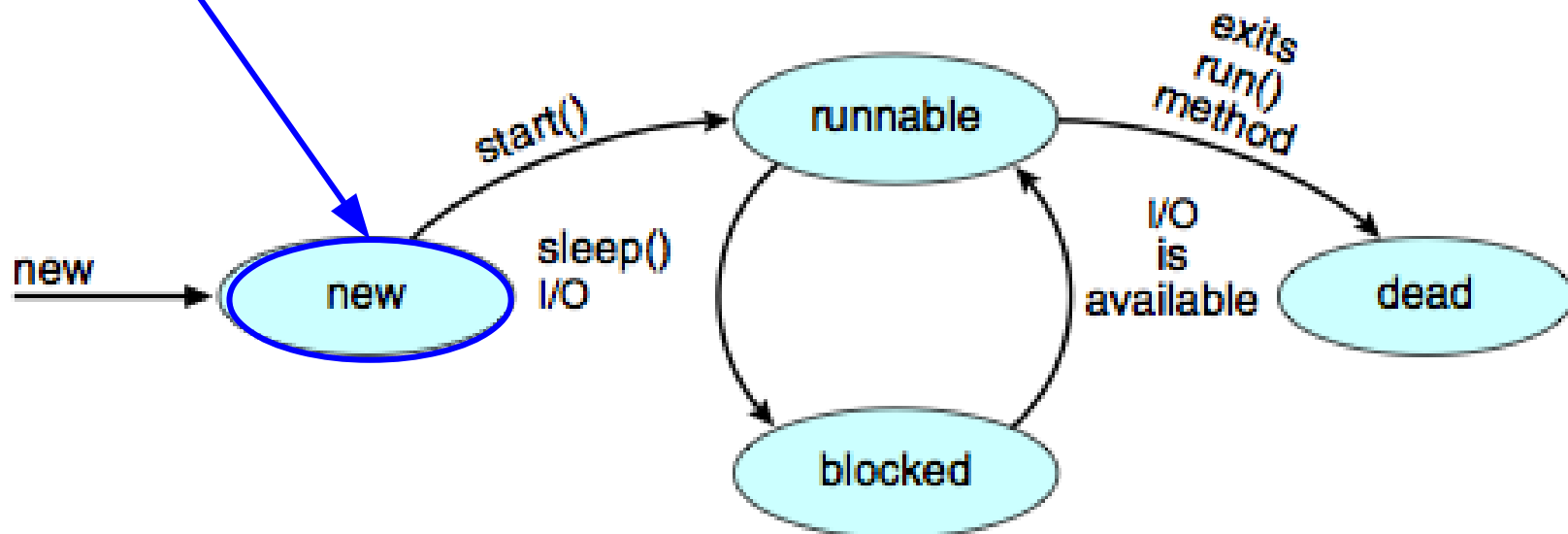
```
1 package br.utfpr.rag.exeaulas.threads;
2
3 public class AlunosThreadsDemo {
4     public static void main(String[] args) {
5
6         ThreadSimpleH ts1 = new ThreadSimpleH("Huguinho");
7         ts1.setPriority(Thread.MIN_PRIORITY);
8         ts1.start();
9
10        ThreadSimpleI ts2 = new ThreadSimpleI();
11        ts2.run();
12
13    }
14 }
```

Estados de *Threads* em Java

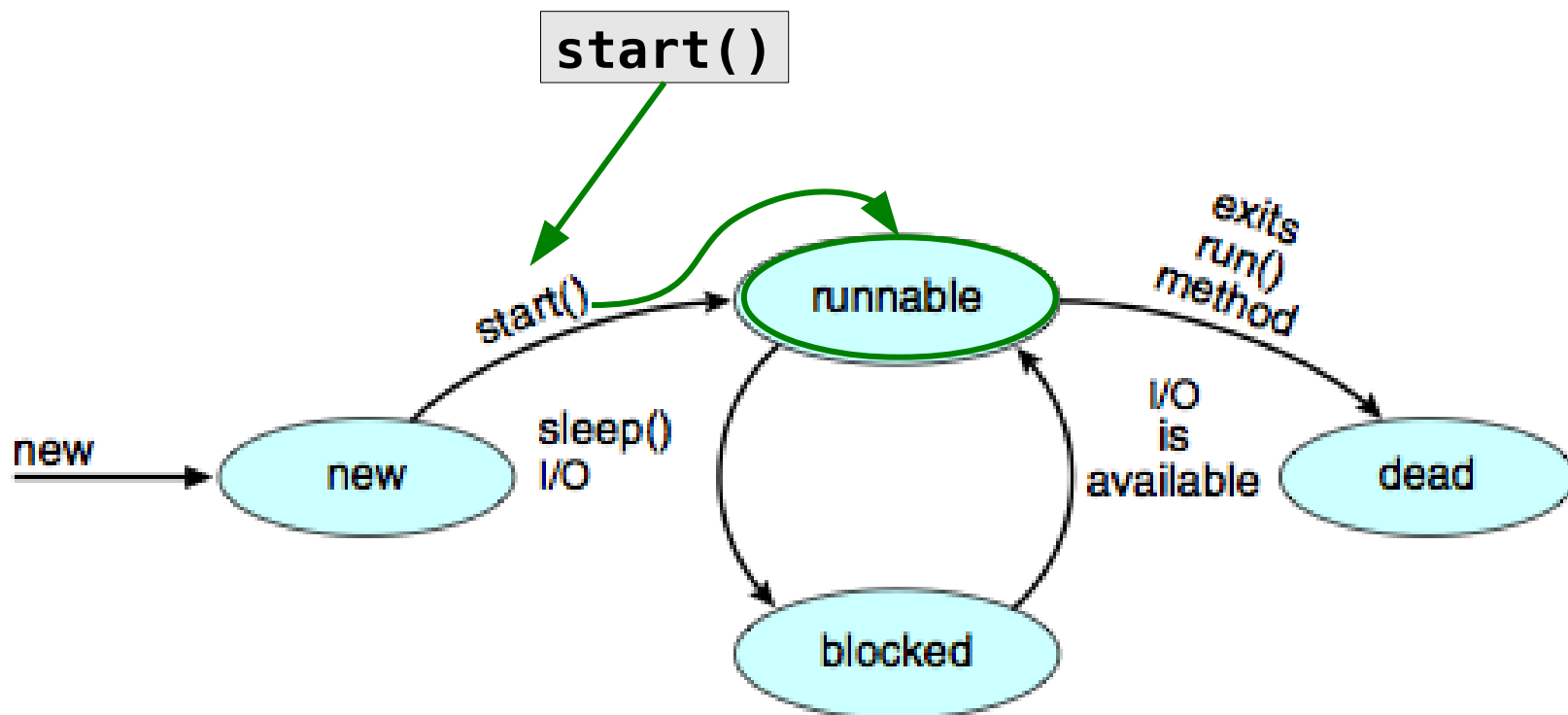


Estados de *Threads* em Java

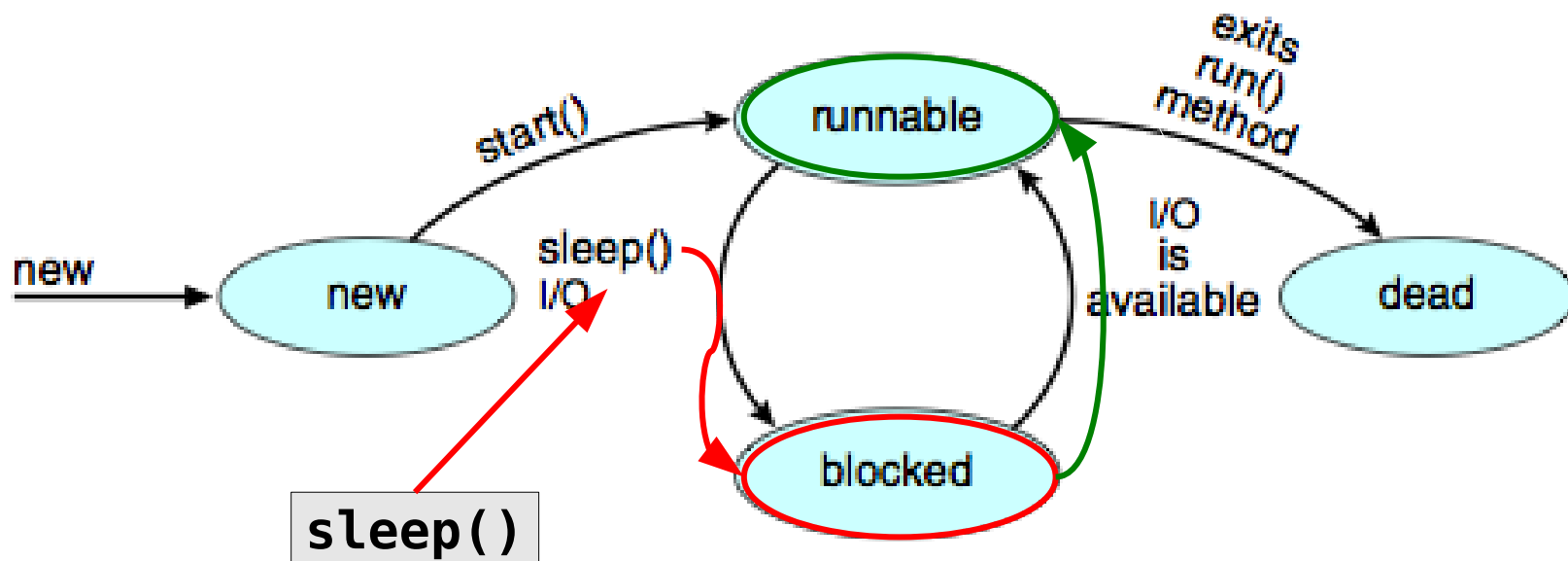
Cria a Thread



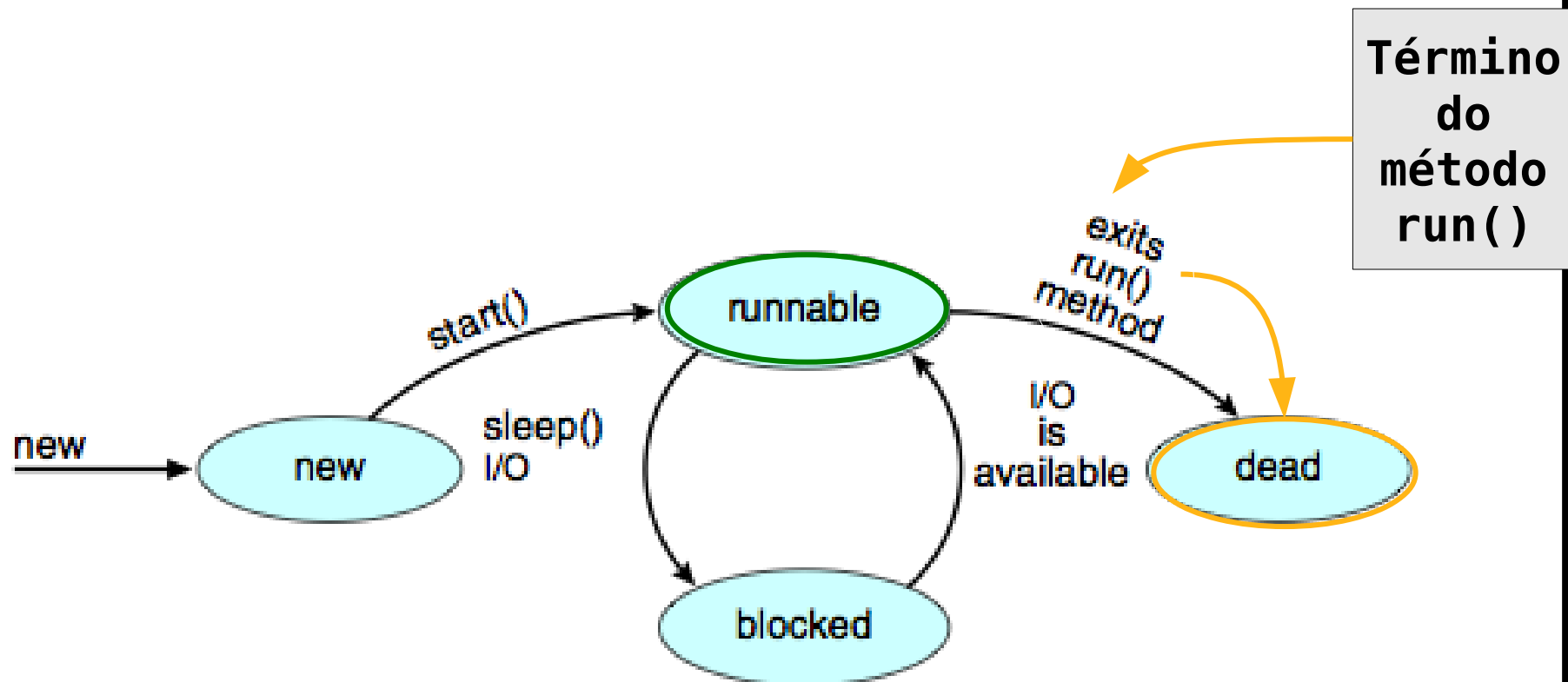
Estados de *Threads* em Java



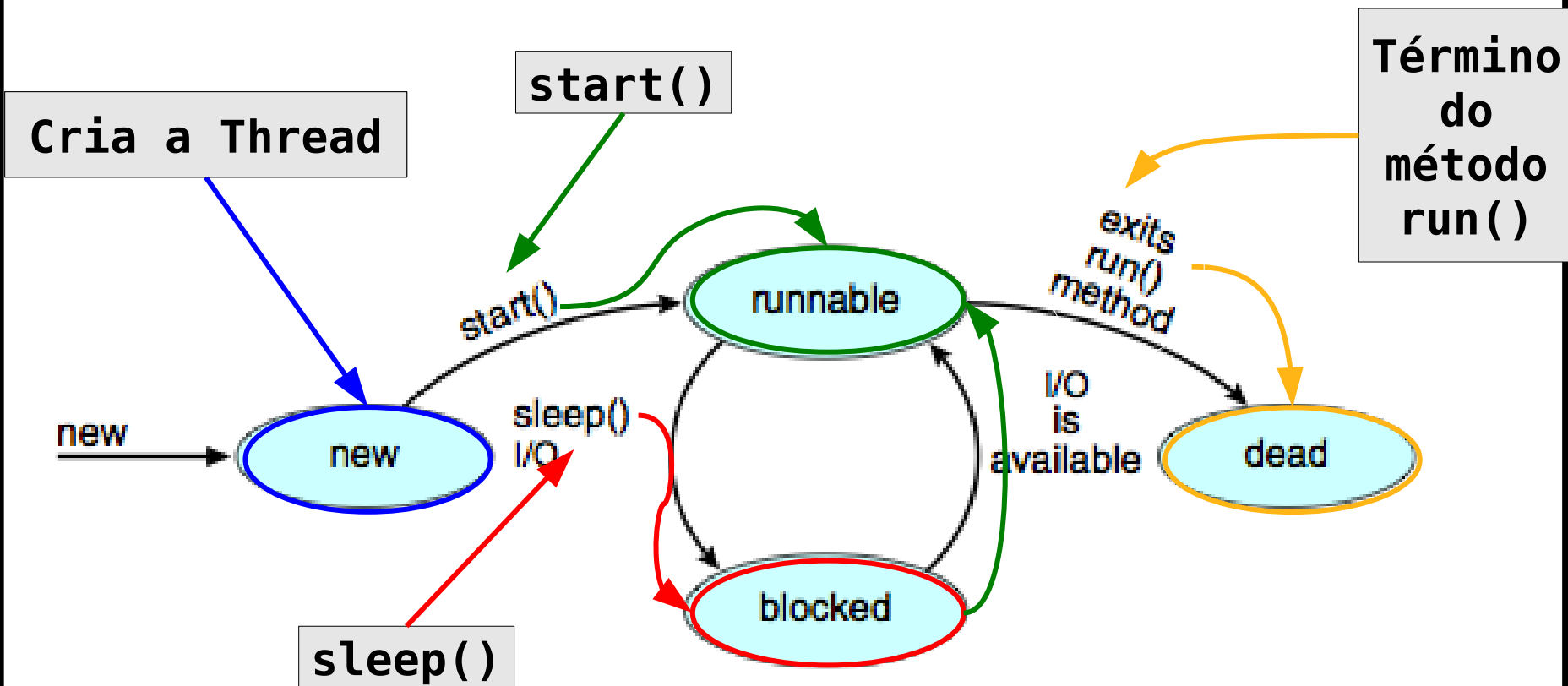
Estados de *Threads* em Java



Estados de *Threads* em Java



Estados de *Threads* em Java



Cancelamento de *Thread*

- Terminando um *thread* antes que ele tenha terminado
- Duas técnicas gerais:
 - **Cancelamento assíncrono** termina o thread de destino imediatamente
 - **Cancelamento adiado** permite que o thread de destino verifique periodicamente se ele deve ser cancelado

Cancelamento de *Thread*

Cancelamento adiado em Java
Interrompendo um Thread

```
Thread thrd = new Thread(new InterruptibleThread());  
thrd.start();  
...  
thrd.interrupt();
```

Cancelamento de *Thread*

Cancelamento adiado em Java Verificando status da interrupção

```
class InterruptibleThread implements Runnable
{
    /**
     * This thread will continue to run as long
     * as it is not interrupted.
     */
    public void run() {
        while (true) {
            /**
             * do some work for awhile
             * . . .
             */

            if (Thread.currentThread().isInterrupted()) {
                System.out.println("I'm interrupted!");
                break;
            }
        }
        // clean up and terminate
    }
}
```

Pools de threads

- Criam uma série de *threads* em um ***pool*** onde esperam trabalho.
- Vantagens:
 - Em geral é mais rápido para atender a solicitação com um *thread* existente do que criar um novo thread → **Tempo de Criação**
 - Permite que uma série de *threads* nas aplicações sejam vinculadas ao tamanho do ***pool***.

Pools de threads em Java

- Executor de único thread
 - `static ExecutorService
newSingleThreadExecutor()`
- Executor de thread fixo
 - `static ExecutorService
newFixedThreadPool(int n)`
- Executor de thread fixo
 - `static ExecutorService
newCachedThreadPool()`

Pools de Threads

Uma tarefa a ser atendida em um pool de threads

```
public class Task implements Runnable {  
    public void run() {  
        System.out.println("Uma tarefa");  
    }  
}
```

Pools de Threads

Criando um pool de Threads

```
import java.util.concurrent.*;

public class TPEXample
{
    public static void main(String[] args) {
        int numTasks = Integer.parseInt(args[0].trim());

        // create the thread pool
        ExecutorService pool = Executors.newCachedThreadPool();

        // run each task using a thread in the pool
        for (int i = 0; i < numTasks; i++)
            pool.execute(new Task());

        // Shut down the pool. This shuts down the pool only
        // after all threads have completed.
        pool.shutdown();
    }
}
```

Dados específicos do *Thread*

- As *threads* pertencentes a um mesmo processo compartilham os dados do processo.
- Isso é um dos benefícios da programação *multithreading*.
- Mas em certos casos, talvez cada *thread* necessite ter sua própria cópia dos dados.

Dados específicos do *Thread*

- A classe *ThreadLocal* para declarar dados específicos.
- Permitindo que cada *thread* tenha sua própria cópia dos dados
- Útil quando você não tem controle sobre o processo de criação de *thread* (ou seja, ao usar um pool de threads)
- Métodos *set()* e *get()*: inicializam/setam e recuperam os dados de *ThreadLocal*.

Dados específicos do *Thread*

Dados específicos do thread em Java

```
class Service
{
    private static ThreadLocal errorCode =
        new ThreadLocal();

    public static void transaction() {
        try {
            /**
             * some operation where an error may occur
             * . . .
             */
        }
        catch (Exception e) {
            errorCode.set(e);
        }
    }

    /**
     * get the error code for this transaction
     */
    public static Object getErrorCode() {
        return errorCode.get();
    }
}
```

Dados específicos do *Thread*

Dados específicos do thread em Java

Declara `errorCode` como sendo um atributo local para a Thread.

```
class Service
{
    private static ThreadLocal errorCode =
        new ThreadLocal();

    public static void transaction() {
        try {
            /**
             * some operation where an error may occur
             * . . .
             */
        }
        catch (Exception e) {
            errorCode.set(e);
        }
    }

    /**
     * get the error code for this transaction
     */
    public static Object getErrorCode() {
        return errorCode.get();
    }
}
```

Qualquer quantidade de Threads podem chamar o método *transaction()*

Caso aconteça um erro ele é armazenado

Se alguma exceção ocorrer o `errorCode` será salvo para cada uma.

Podendo ser recuperado através de um `get()`

Manipulando *Threads* em Java

- ***Threads.sleep(t)***
 - Faz com que o *thread* suspenda a execução por um tempo *t*
- ***Thread.interrupted()***
 - Interrompe a execução corrente do *thread*
- ***t.join()***
 - Faz com que o *thread* que esteja executando suspenda para que o *thread t* execute até terminar

Pools de threads em Java - Exemplo

```
public class Task implements Runnable {  
    public void run() {  
        System.out.println("Uma tarefa");  
    }  
}
```


Pools de threads em Java - Exemplo

```
import java.util.concurrent.*

public class Exemplo {

    public static void main(String[] args) {
        int nt = Integer.parseInt(args[0].trim());
        ExecutorService pool = Executors.newCachedThreadPool();

        for (int i=0; i<nt; i++)
            pool.execute(new Task());

        pool.shutdown();
    }
}
```

Próxima Aula
