Universidade Federal da Bahia

Sistemas Operacionais

MATA58

Prof. Maycon Leone M. Peixoto

mayconleone@dcc.ufba.br

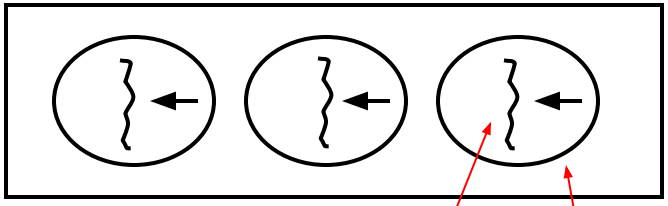
Processos

- Sistemas Operacionais tradicionais:
  - Cada processo tem um único espaço de endereçamento e um único fluxo de controle
- Existem situações onde é desejável ter múltiplos fluxos de controle compartilhando o mesmo espaço de endereçamento:
  - Solução: threads

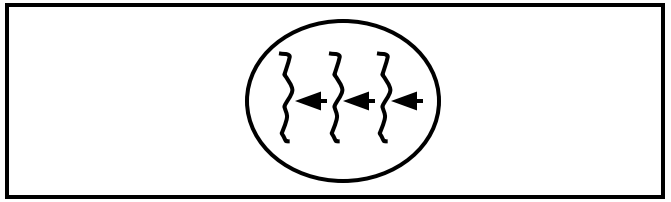
Threads

- Um processo tradicional (pesado) possui um contador de programas, um espaço de endereço e apenas uma *thread* de controle (ou fluxo de controle);
- ***Multithreading****:* Sistemas atuais suportam múltiplas *threads* de controle, ou seja, pode fazer mais de uma tarefa ao mesmo tempo, servindo ao mesmo propósito;

a) Três processos



b) Um processo com três *threads*

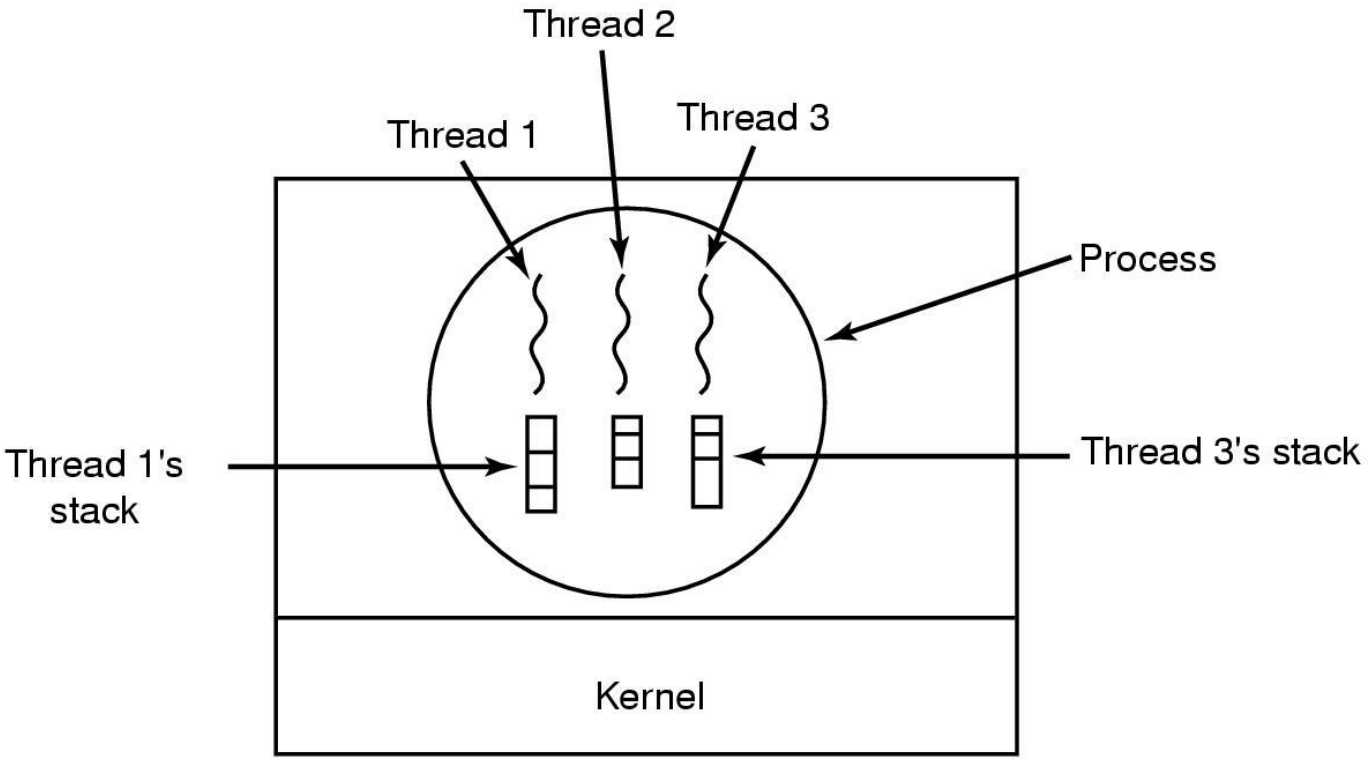


- As três *threads* utilizam o mesmo espaço de endereço

Threads

- *Thread* é uma entidade básica de utilização da CPU.
  - Também conhecidos como processos leves (*lightweight process* ou *LWP*);
- Processos com múltiplas *threads* podem realizar mais de uma tarefa de cada vez;
- Processos são usados para agrupar recursos; *threads* são as entidades escalonadas para execução na CPU
  - A CPU alterna entre as *threads* dando a impressão de que elas estão executando em paralelo;

Threads



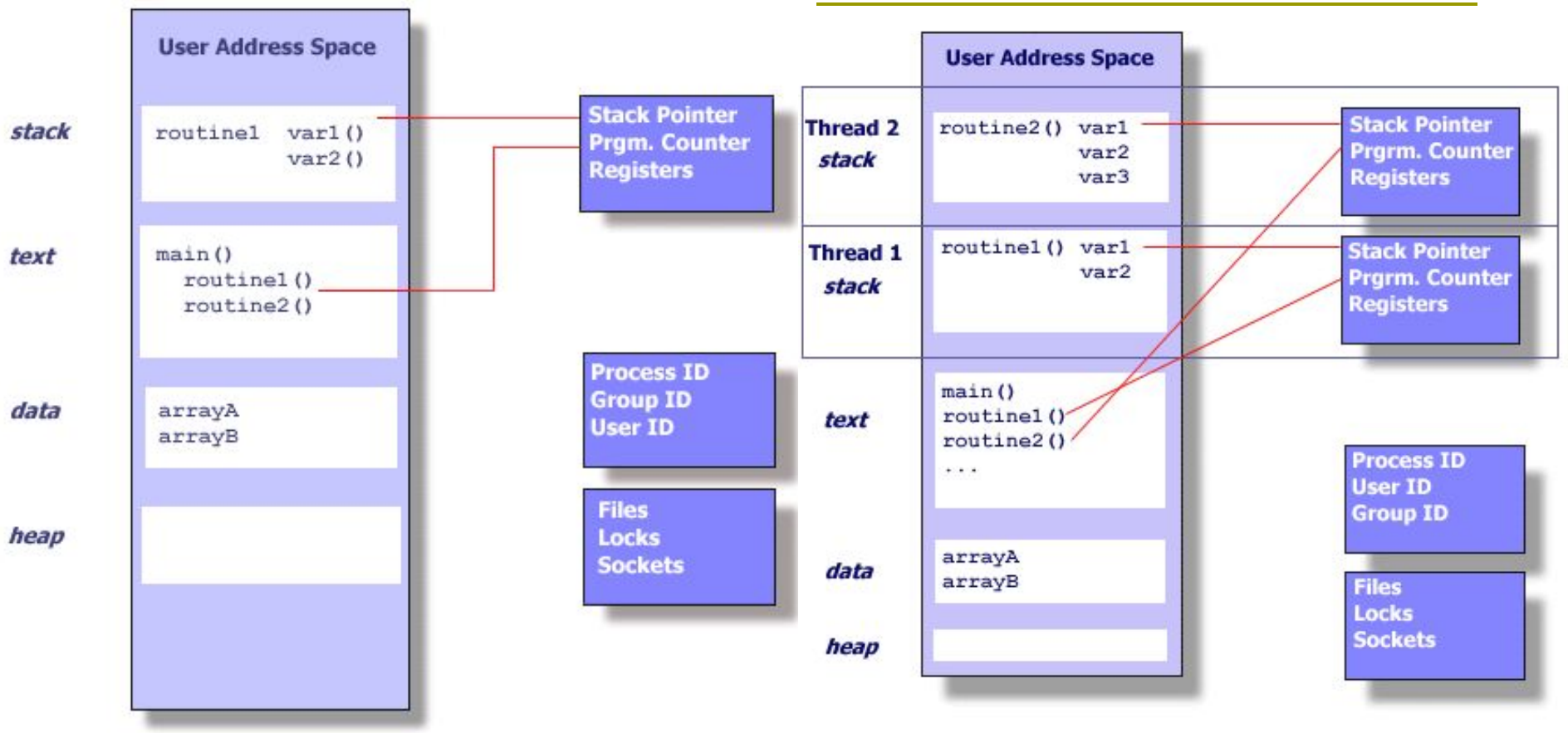
Cada *thread* tem sua pilha de execução

Threads

Itens por Processo	Itens por *Thread*
Espaço de endereçamento Variáveis globais Arquivos abertos Processos filhos Alarmes pendentes	Contador de programa Registradores (contexto) Pilha Estado

- Compartilhamento de recursos;
- Cooperação para realização de tarefas;

Threads



Processo Unix

Threads em um processo Unix

Threads

- Como cada *thread* pode ter acesso a qualquer endereço de memória dentro do espaço de endereçamento do processo, uma *thread* pode ler, escrever ou apagar a pilha de outra *thread*;
- Não existe proteção pois:
  - É impossível
  - Não é necessário pois, diferente dos processos que podem pertencer a diferentes usuários, as threads são sempre de um mesmo usuário

Threads

- 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 □ o processo é que tem!
  - 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 e requisitar a figura □ tempo;
  - Com múltiplas *threads*, muitas imagens podem ser requisitadas ao mesmo tempo melhorando o desempenho;

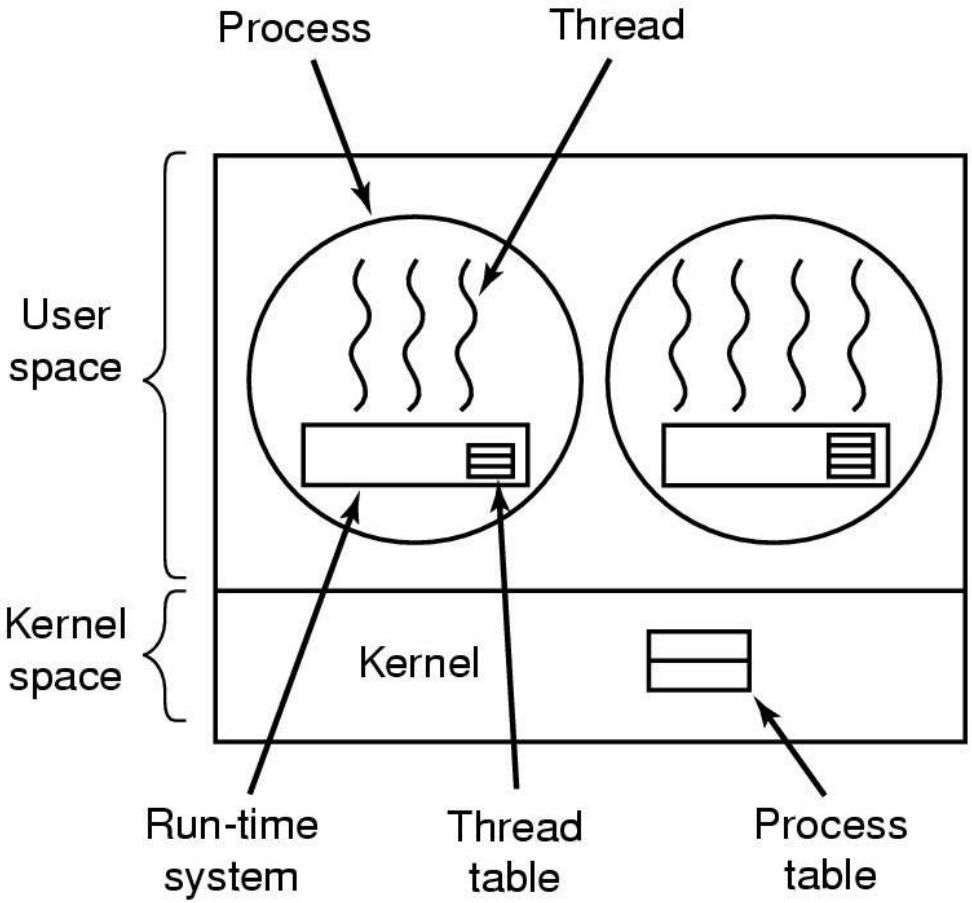
Threads

- 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

- Tipos de *threads*:
  - Em modo usuário (espaço do usuário): implementadas por bibliotecas no espaço do usuário;
    - Criação e escalonamento são realizados sem o conhecimento do *kernel*;
      - Sistema Supervisor (*run-time system*): coleção de procedimentos que gerenciam as *threads*;
      - Tabela de *threads* para cada processo;
    - Cada processo possui sua própria tabela de *threads*, que armazena todas a informações referentes à cada *thread* relacionada àquele processo;

Threads em modo usuário



Threads em modo usuário

- Tipos de *threads*: Em modo usuário
- Vantagens:
  - Alternância de *threads* no nível do usuário é mais rápida do que alternância no *kernel*;
  - Menos chamadas ao *kernel* são realizadas;
  - Permite que cada processo possa ter seu próprio algoritmo de escalonamento;
  - Podem ser implementado em Sistemas Operacionais que não têm threads
- Principal desvantagem:
  - Processo inteiro é bloqueado se uma *thread* realizar uma chamada bloqueante ao sistema;

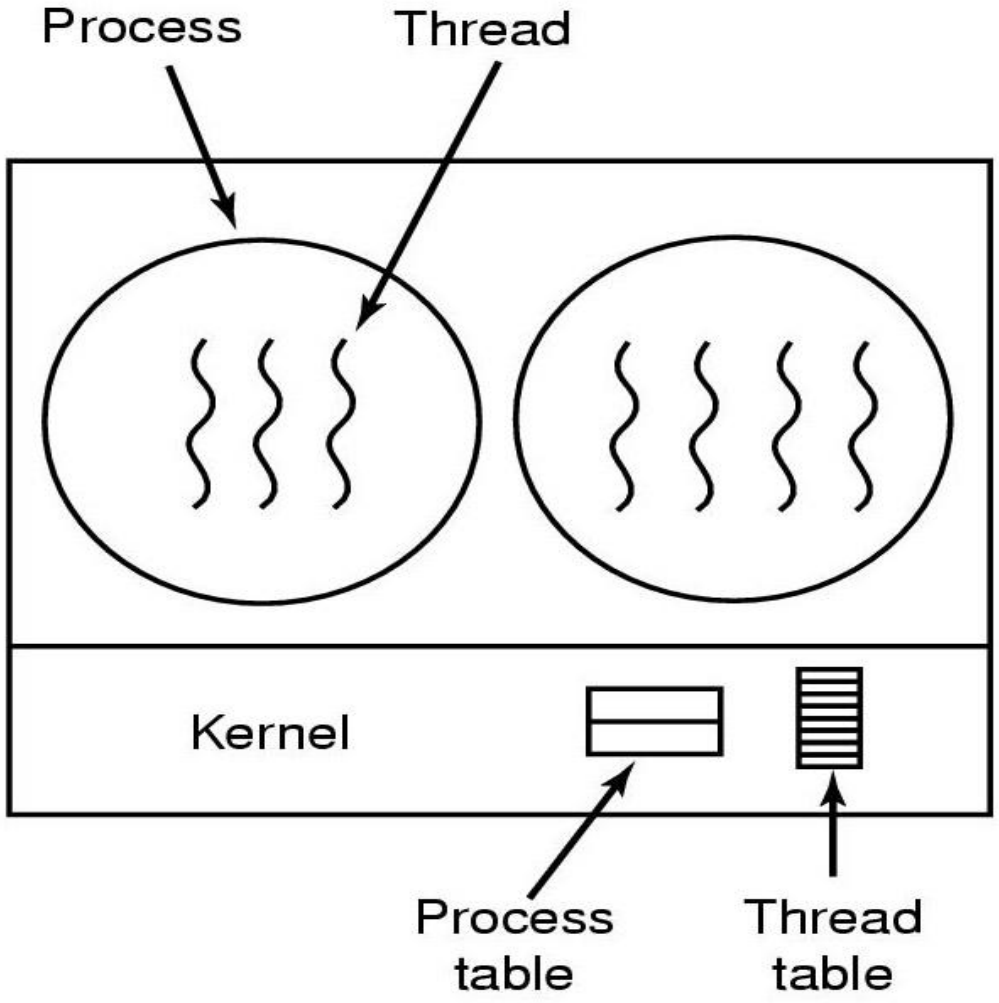
Implementação de threads

- Implementação em espaço de usuário:
  - Problemas:
    - Como permitir chamadas bloqueantes se as chamadas ao sistema são bloqueantes e essa chamada irá bloquear todas as threads?
      - Mudar a chamada ao sistema para não bloqueante, mas isso implica em alterar o SO -> não aconselhável
      - Verificar antes se uma determinada chamada irá bloquear a thread e, se for bloquear, não a executar, simplesmente mudando de thread
    - Page fault
      - Se uma thread causa uma page fault, o kernel, não sabendo da existência da thread, bloqueia o processo todo até que a página que está em falta seja buscada
    - Se uma thread não liberar a CPU voluntariamente, ela executa o quanto quiser
      - Uma thread pode não permitir que o processo escalonador do processo tenha sua vez

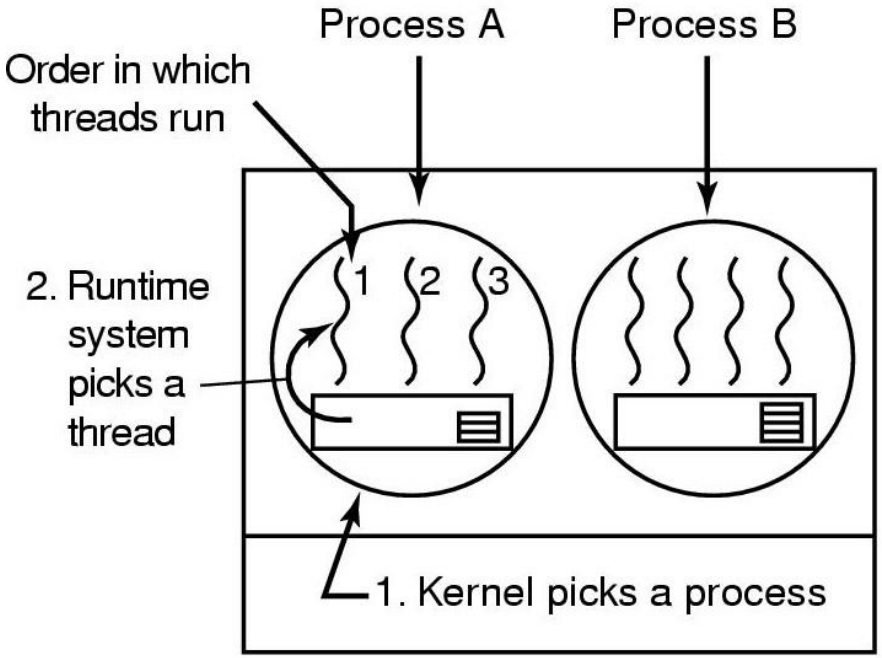
Tipos de *Threads*

- Tipos de *threads*:
  - Em modo *kernel*: suportadas diretamente pelo SO;
  - Criação, escalonamento e gerenciamento são feitos pelo *kernel*;
    - Tabela de *threads* e tabela de processos separadas;
      - as tabelas de *threads* possuem as mesmas informações que as tabelas de threads em modo usuário, só que agora estão implementadas no *kernel*;

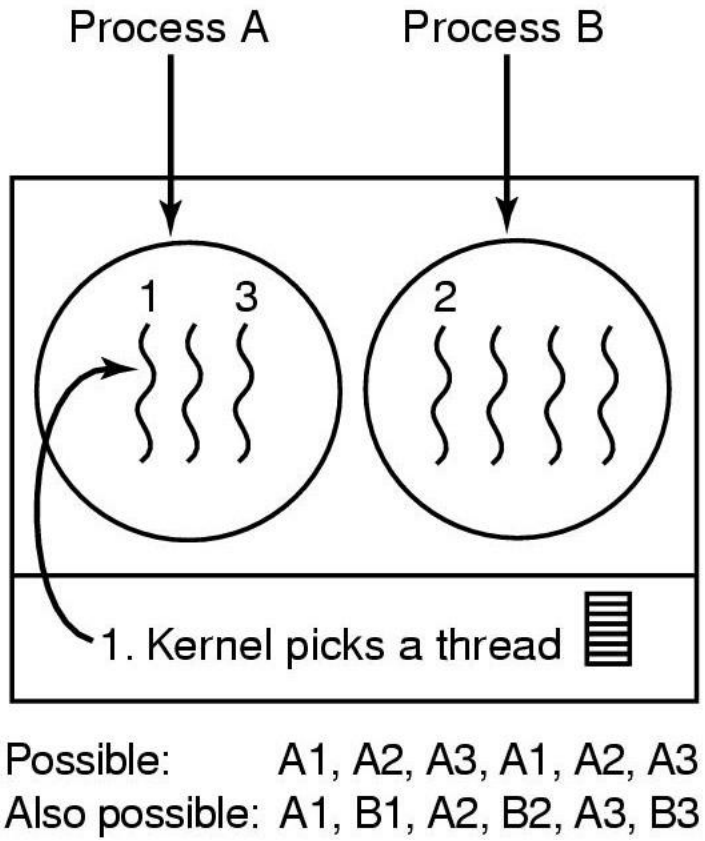
Threads em modo *kernel*



Threads em modo Usuário *x* *Threads* em modo *Kernel*



Threads em modo usuário



Threads em modo *kernel*

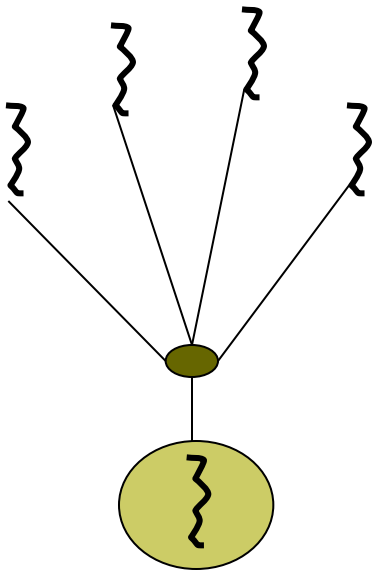
Threads em modo *kernel*

- Vantagem:
  - Processo inteiro não é bloqueado se uma *thread* realizar uma chamada bloqueante ao sistema;
- Desvantagem:
  - Gerenciar threads em modo *kernel* é mais caro devido às chamadas de sistema durante a alternância entre modo usuário e modo *kernel*;

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;

- Gerenciamento Eficiente
- Se uma bloquear todas bloqueiam

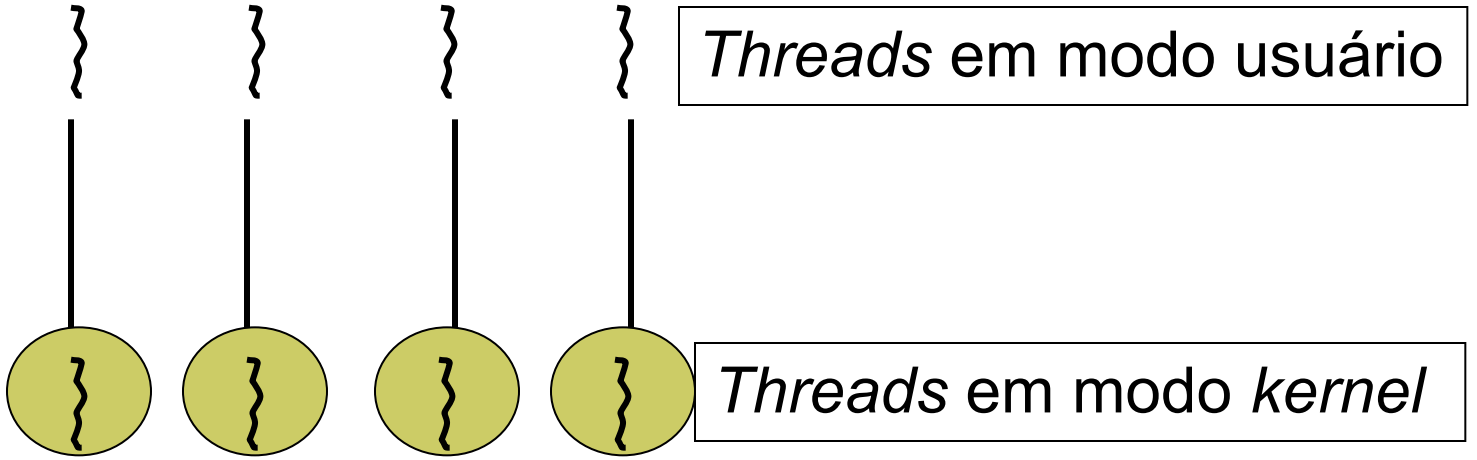


Threads em modo usuário

Thread em modo *kernel*

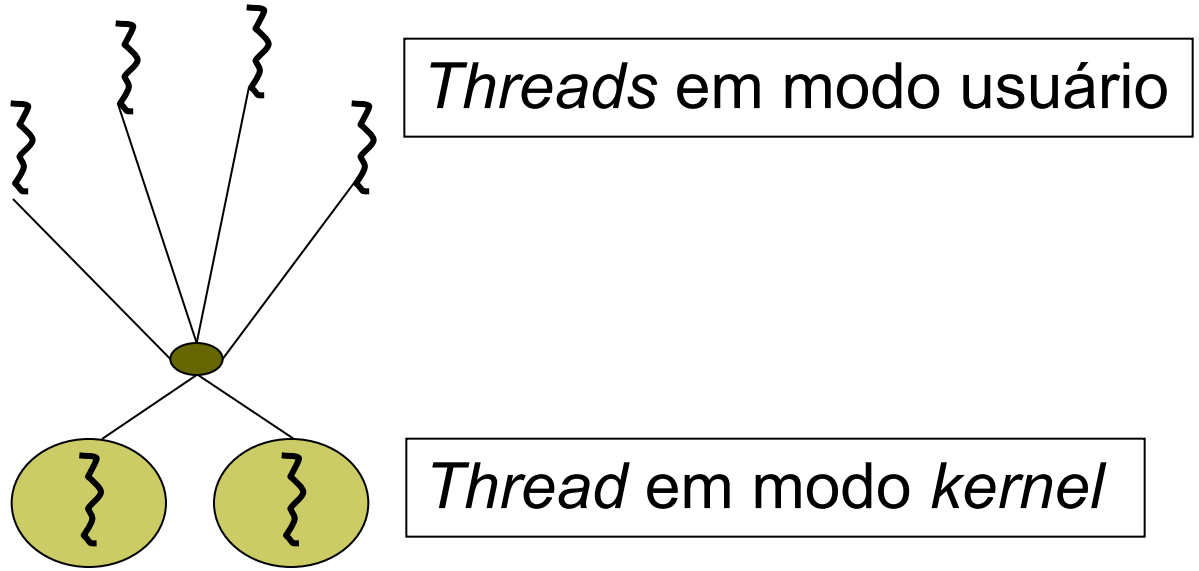
Threads

- 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

- 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;* não é uma implementação
  - Modo usuário

Implementação

- **POSIX Threads (Biblioteca Pthreads)**
  - Define uma API, implementada sobre o SO;
  - Utilizada em sistemas UNIX: Linux, Mac OS X;
  - Padrão IEEE POSIX 1003.1c ;
- **Win 32 Threads**
  - Implementação do modelo um para um no kernel;
- **Java**
  - threads são gerenciadas pela JVM, a qual é executada sobre um SO;
  - JVM especifica a interface com SO;
  - Utiliza uma biblioteca de thread do SO hospedeiro.

Implementação - Java

- Fornece um ambiente abstrato, que permite aos programas Java executar em qualquer plataforma com o JVM;
  - Threads são gerenciadas pelo JVM;
  - Pacote java.lang.Thread;
- Duas formas de manipulação:
  - Criar uma nova classe derivada da **classe Thread**;
  - Classe que implemente a **interface Runnable** (mais utilizada);
  - Nos dois casos o programador deve apresentar uma implementação para o método run( ), que é o método principal da thread.

Implementação - Java

- Exemplo (1):

```
public class Trabalhador extends Thread {
      String nome, produto;
      int tempo;

   public Trabalhador(String nome, String produto, int
    tempo) {
      this.nome = nome;
      this.produto = produto;
      this.tempo = tempo;
   }
```

Implementação - Java

- Exemplo (1):

```
public void run() {
   for (int i=0; i<50; i++) {
    try {
      Thread.sleep(tempo);
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
   System.out.println(nome+"produziu o(a)"+i+ "°"+ produto);
}
```

Implementação - Java

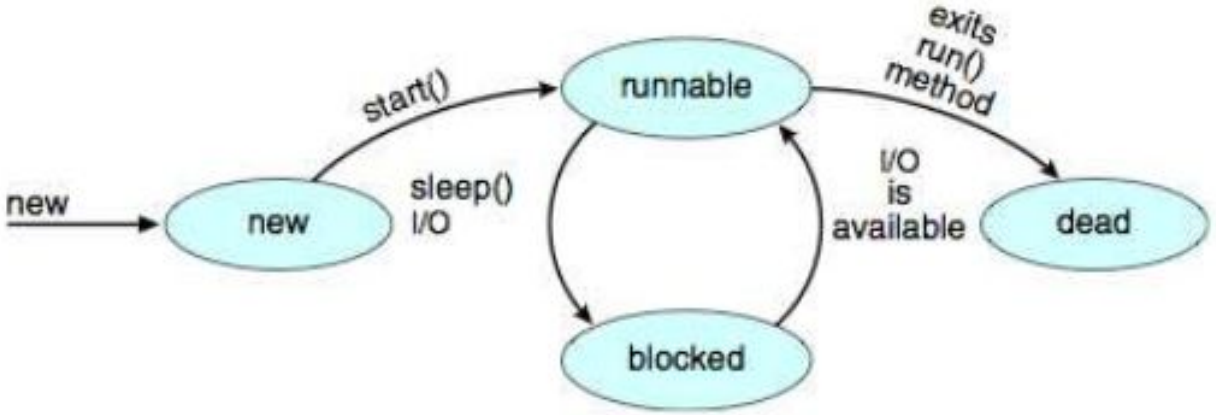
- Exemplo (1):

```
public static void main(String[] args) {
    Trabalhador t1 = new Trabalhador("Mario", "bota", 1000);
    Trabalhador t2 = new Trabalhador("Sergio", "camisa",
    2000);

    t1.start();
    t2.start();
}
```

Implementação - Java

- As threads podem estar em um dos 4 estados:
  - **Novo**: quando é criada (new);
  - **Executável**: método start( ) para alocar memória e chama o método run ( );
  - **Bloqueado**: utilizado para realizar uma instrução de bloqueo ou para invocar certos métodos como sleep( );
  - **Morto**: passa para esse estado quando termina o método run();



Cancelamento de Thread

- Corresponde à tarefa de terminar um thread antes que se complete.
- Exemplos:
  - multiplas threads pesquisando em banco de dados;
  - Navegador web acessando uma página.
- Denominada **thread alvo** e pode ocorrer em dois diferentes cenários:
  - **Cancelamento assíncrono**: um thread imediatamente termina o thread-alvo. Pode não liberar os recursos necessários a nível de sistema.
  - **Cancelamento adiado** permite o thread alvo ser periodicamente verificado se deve ser cancelada;

Tratamento de Sinais

- Sinais são usados nos sistemas UNIX para notificar um processo de que um evento específico ocorreu;
- Um sinal pode ser:
  - Síncrono: se forem liberados para o mesmo processo que provocou o sinal;
    - Exemplo: processo executa divisão por 0 e recebe sinal de notificação;
  - Assíncrono: se forem gerados por um evento externo (ou outro processo) e entregues a um processo;

Tratamento de Sinais

- Sinais para processos comuns
  - São liberados apenas para o processo específico (PID);
- Sinais para processos multithread: várias opções
  - Liberar o sinal para a thread conveniente (ex.: a que executou divisão por zero);
  - Liberar o sinal para todas as threads do processo (ex.: sinal para término do processo);
  - Liberar o sinal para determinadas threads;
  - Designar uma thread específica para receber todos os sinais.

Cadeias de Threads

- A criação/finalização de threads traz alguns problemas: overhead:
  - Caso do servidor web: recebe muitas conexões por segundo:
  - Criar e terminar inúmeras threads é um trabalho muito grande;
  - Trabalha-se com um número ilimitado de threads: término dos recursos

Cadeias de Threads

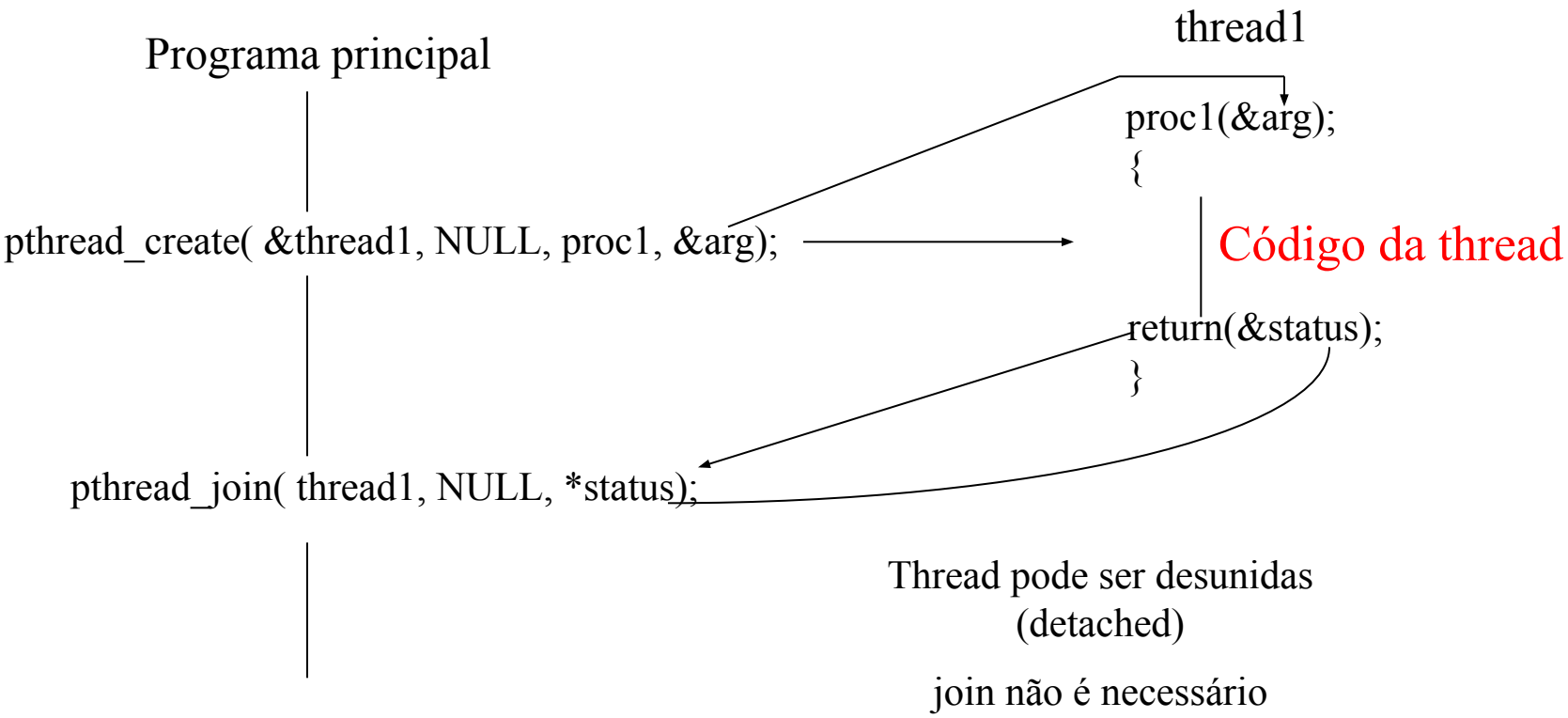
- Solução: utilizar cadeia de threads em espera
  - Na inicialização do processo, cria-se um número adequado de threads
  - As threads permanecem aguardando para entrar em funcionamento
  - Quando o servidor recebe uma solicitação, desperta uma thread da cadeia (se houver disponível, senão espera) e repassa trabalho
  - Quando thread completa serviço, volta à cadeia de espera
- Vantagens: mais rápido que criar thread, limita recursos

Threads em Linux

- O Linux refere-se a elas como tarefas, em vez de threads;
- A criação de thread é feita através da chamada de sistema clone( );

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 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.

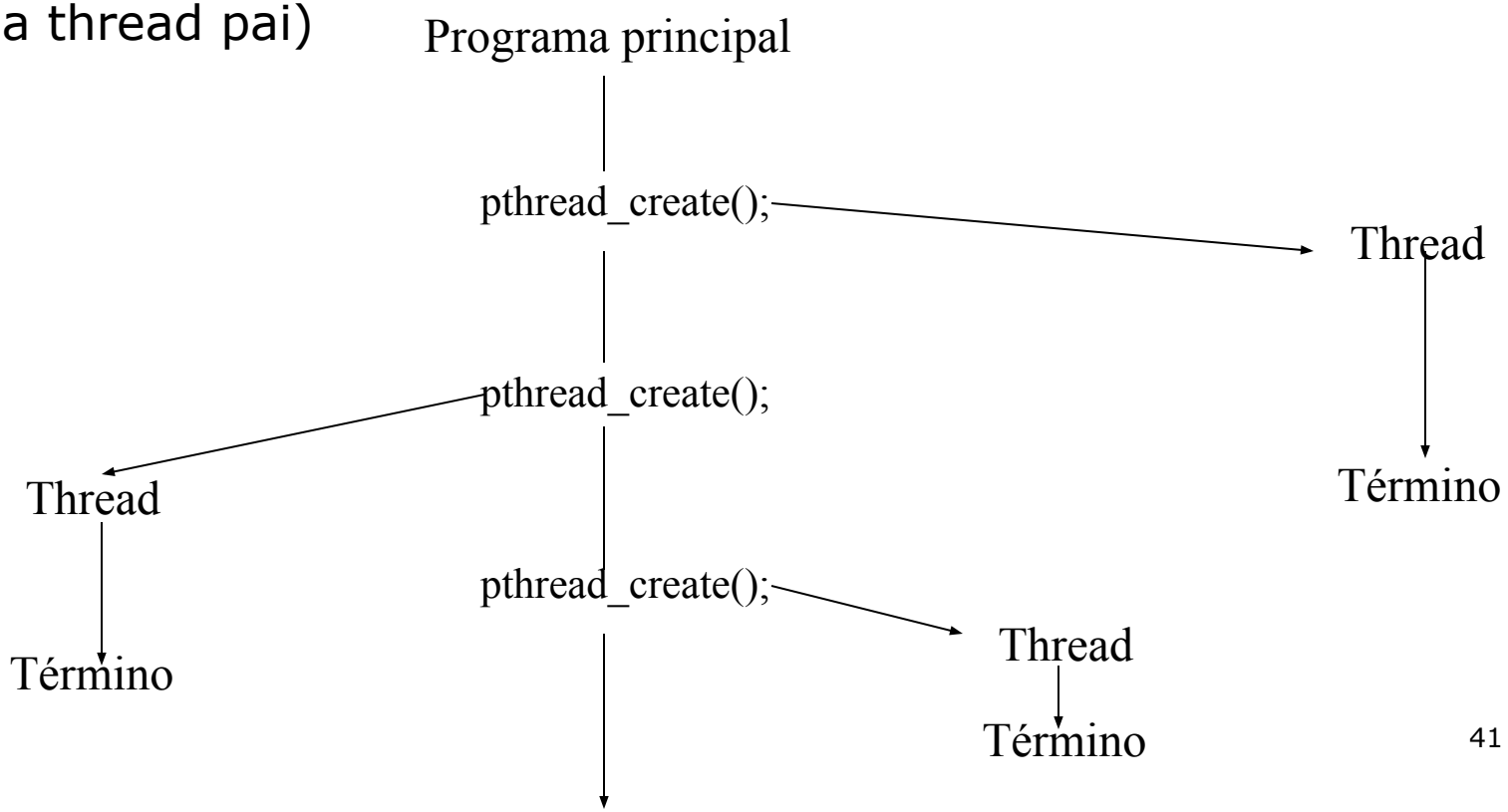
Threads em C
PThreads

- 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

- Detached Threads (desunidas)
  - Pode ser que uma thread não precisa saber do término de uma outra por ela criada, então não executará a operação de união. Neste caso diz-se que o thread criado é detached (desunido da thread pai)



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);
}
```

```
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);
}
```

Resumo sobre Threads

- Uma thread é a unidade básica utlizada pela CPU, onde um processo é composto de uma ou mais threads;
- Cada thread tem: registradores, pilha, contadores;
- As threads compartilham: código, dados, e recurso do SO, como arquiivo aberto;
- Threads de nível usuário e nível kernel;
- Modelos de mapeagemnto de threads usuário para kernel:
  - N-para -1, 1-para-1, N-para-N.
- Bibliotecas: Pthreads, Win32, Java

Referências

- PThreads
  - https://computing.llnl.gov/tutorials/pthreads/