



# Sistemas Operacionais 1

## Threads

Robson Siscoutto  
robson@unoeste.br

# Sistemas Operacionais

## Gerenciamento de Processos

---

- Introdução;
- Modelos do processo;
- Estados e mudanças do processo;
- Tipos de processos;
- Comunicação entre processos;
- Escalonamento de Processos;
- Concorrência entre Processos
  - Problemas de sincronização;
  - Soluções de hardware e software;
- Deadlock
- **Threads**
  - Conceitos
  - Sincronização entre Threads (em Java)

# Gerenciamento de Processos

## Threads

---

- Em sistemas operacionais tradicionais, cada processo têm:
  - um espaço de endereçamento e
  - uma única Thread de controle.
- Modelo de processo é baseado em dois conceitos básicos:
  - agrupamento de recursos:
    - Código do programa, dados, arquivos abertos, processo filhos, registradores, sinais, identificador etc.
  - Execução:

# Gerenciamento de Processos

## Threads

---

- Uma Thread é uma unidade de execução da CPU;
- É constituída de :
  - Contador de programa – Registrador PC;
  - Conjunto de registradores;
  - Espaço de pilha;
- Uma Thread compartilha com outras Threads parceiras:
  - Seção de código;
  - Seção de dados;
  - Recursos dos sistema operacional;

# Gerenciamento de Processos

## Threads

- Itens compartilhado por todas as Threads em um processo;
- Itens privados de cada Thread;

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

### Per process items

Address space  
Global variables  
Open files  
Child processes  
Pending alarms  
Signals and signal handlers  
Accounting information

### Per thread items

Program counter  
Registers  
Stack  
State

# Gerenciamento de Processos

## Threads

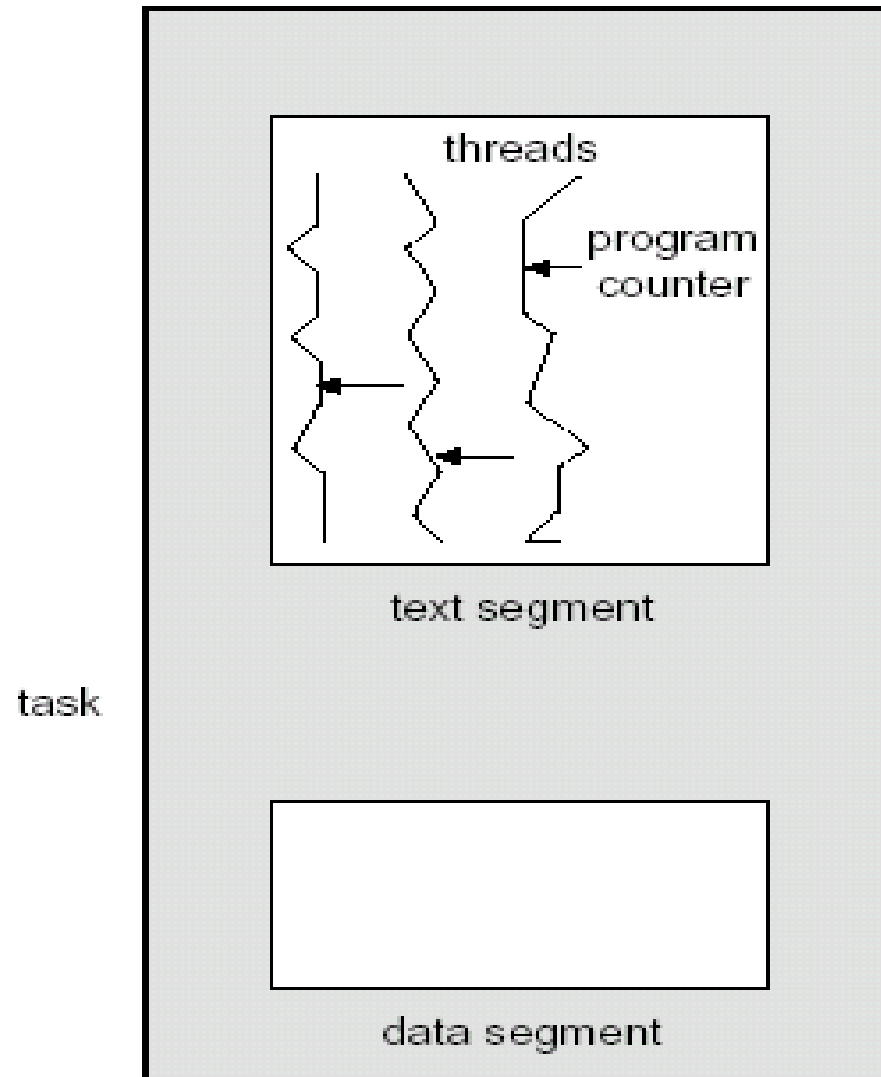
---

- Por que Threads:
  - Threads permite múltiplas execuções no mesmo ambiente do processo, com um alto grau de independência uma das outras.
  - múltiplas Threads executando em paralelo sobre um processo é análogo a ter múltiplos processos executando em paralelo em um computador.
- Conceito de Multi-Threading
  - Múltiplas Threads executando sobre um processo;

# Gerenciamento de Processos

## Threads

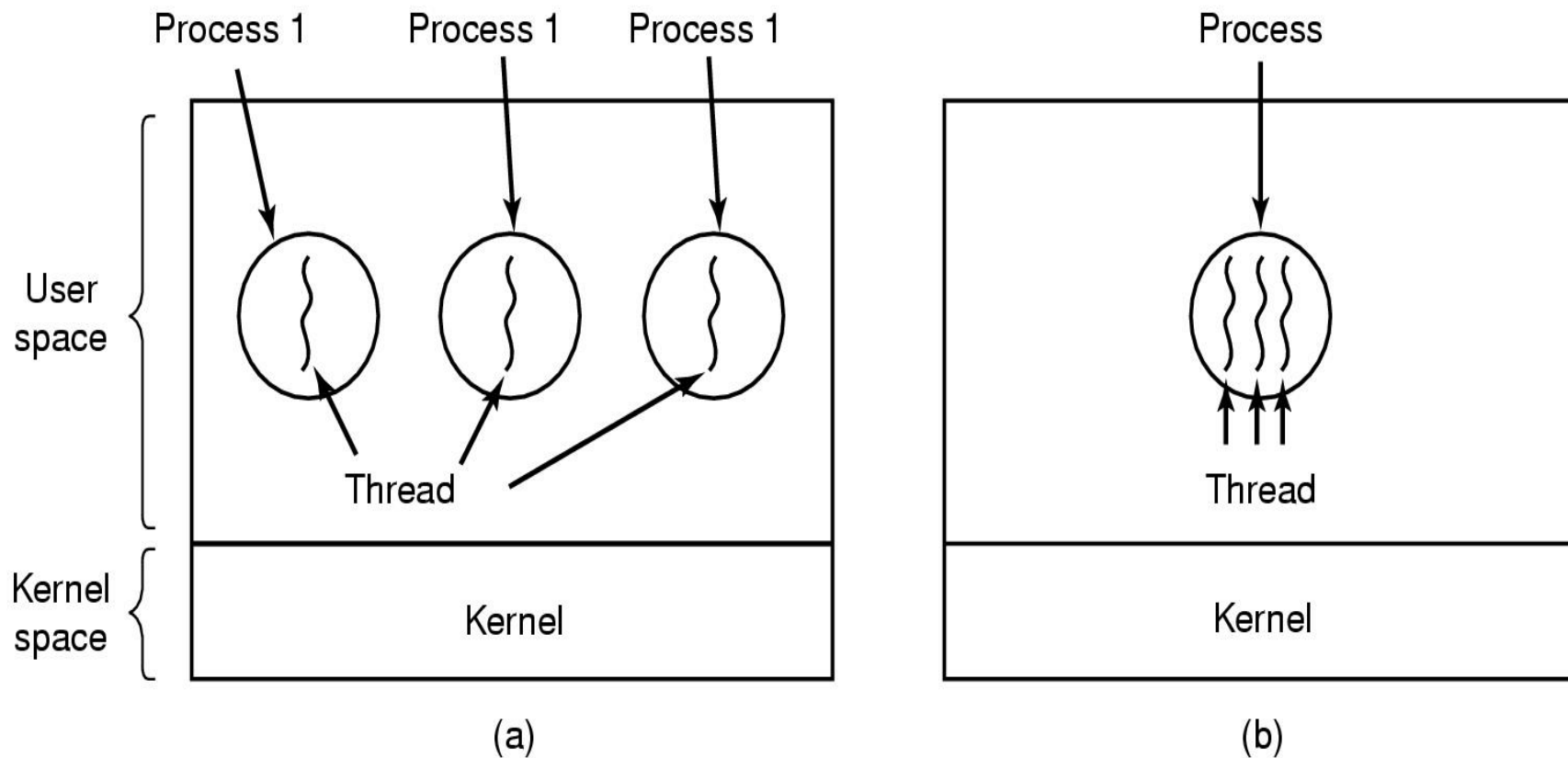
- Processo com várias Threads



# Gerenciamento de Processos

## Threads

- Comparação entre Threads e Processos





# Gerenciamento de Processos

## Threads

---

- Execução das Threads:
  - Similar aos processos (troca de contexto);
  - A CPU é trocada rapidamente entre as Threads dando a ilusão de que as Threads estarão rodando em paralelo.
  - Velocidade real da CPU é dividida igualmente pelo numero de Threads.
- As Threads em um processo não são totalmente independentes:
  - Compartilham espaço endereçamento, logo variáveis globais;

# Gerenciamento de Processos

## Threads

---

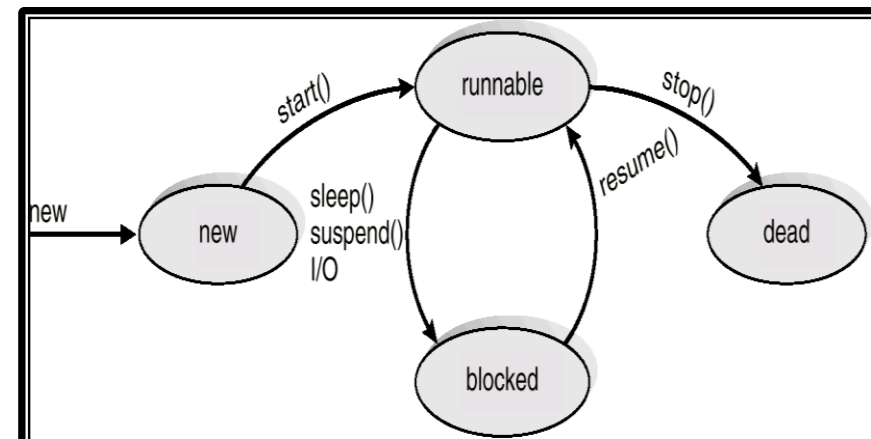
- Uma Thread pode:
  - acessar qualquer endereço de memória dentro do “espaço de endereçamento” do processo,
  - Pode ler, escreve ou limpar completamente a pilha de uma outra Thread.
- Não há proteção entre Threads porque:
  - (1) é impossível
    - Só possível proteger processo e não Threads
  - (2) não deve ser necessário.
    - Pertence o um só usuário

# Gerenciamento de Processos

## Threads

### Estado das Threads:

- Executando
  - tem a CPU e está ativa
- Bloqueado
  - Esperando algum evento para se desbloquear;
  - Pode esperar que outra Threads a desbloqueei;
- Pronto
  - Esperando a CPU
- Finalizado;



# Gerenciamento de Processos

## Threads

---

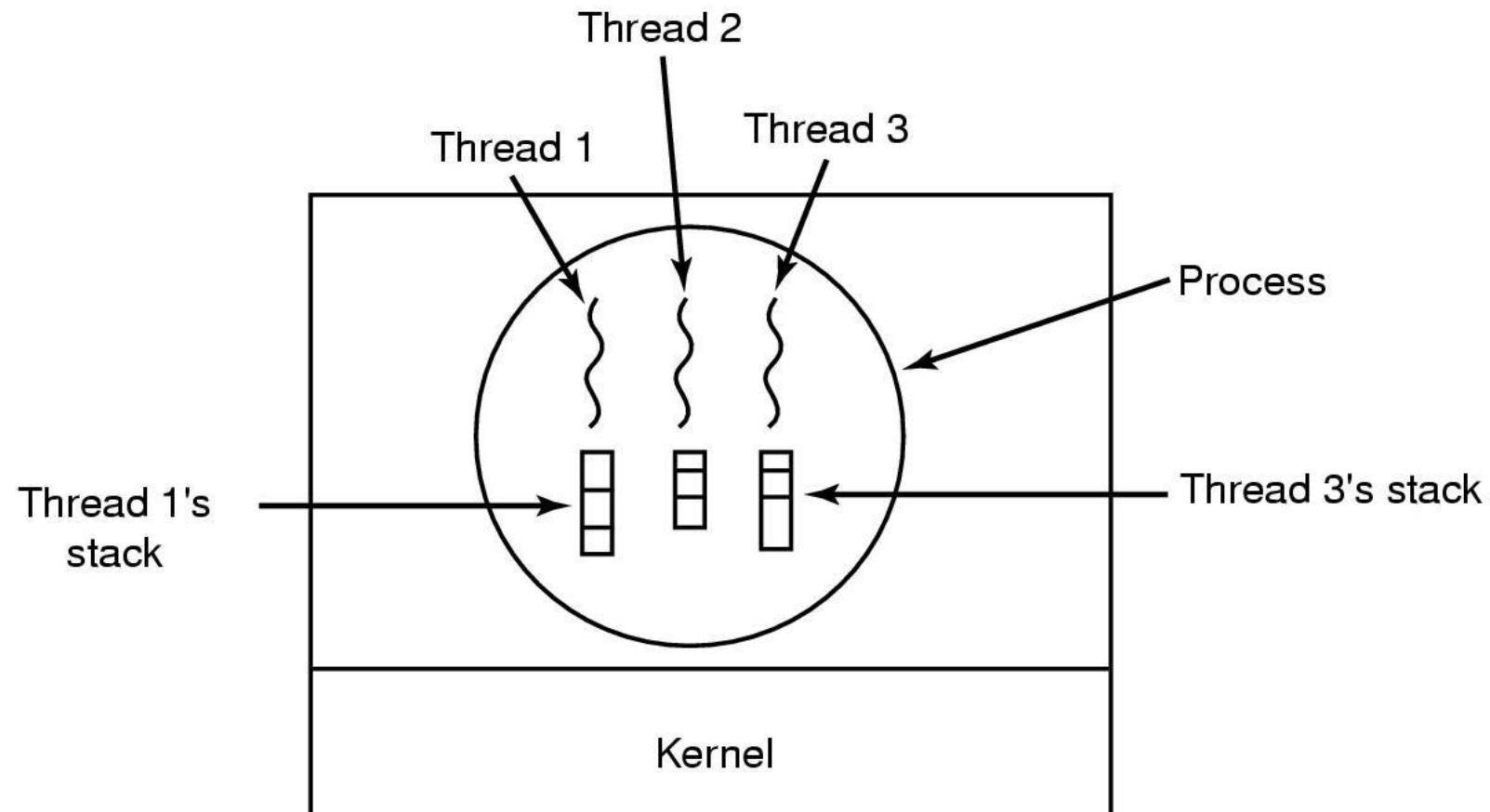
### Cada Thread tem sua própria PILHA:

- Cada Thread durante sua execução têm uma história diferente;
- É na pilha onde as chamadas de procedimentos de armazenadas;
- Uma entrada para cada chamada;
- Contem as variáveis locais e o endereço do retorno do procedimento para ser usado quando a chamada ao procedimento terminar;

# Gerenciamento de Processos

## Threads

- Cada Thread com sua Pilha



# Gerenciamento de Processos

## Threads

---

### Comandos para manipulação de Threads:

- Thread\_create
  - Criar novas Threads
  - Não existe relacionamento Pai-Filho como processos;
  - Todas as Threads são iguais;
  - Cada Thread possui seu próprio identificador;
- Thread\_exit
  - Finalizar um Thread;

# Gerenciamento de Processos

## Threads

---

- Comandos para manipulação de Threads:
  - Thread\_wait
    - uma Thread pode esperar por uma Thread (específica) para finalizar;
  - Thread\_yield
    - permite que uma Thread dê voluntariamente a CPU para uma outra Thread executar;
    - Importante pois não Fatia de Tempo para Threads;

# Vantagens das Threads sobre Processos <sup>(1)</sup>

---

- A **criação e terminação de uma *thread* é mais rápida** do que a criação e terminação de um processo pois elas não têm quaisquer recursos alocados a elas.
  - (S.O. Solaris) Criação = 30:1, Troca de contexto = 5:1
- A **mudança de contexto entre *threads* é mais rápida** do que entre dois processos, pois elas compartilham os recursos do processo.
  - (S.O. Solaris) Troca de contexto = 5:1
- A **comunicação entre *threads* é mais rápida** do que a comunicação entre processos, já que elas compartilham o espaço de endereçamento do processo.
  - O uso de variáveis globais compartilhadas pode ser controlado através de primitivas de sincronização (monitores, semáforos, etc).



# Vantagens das Threads sobre Processos <sup>(2)</sup>

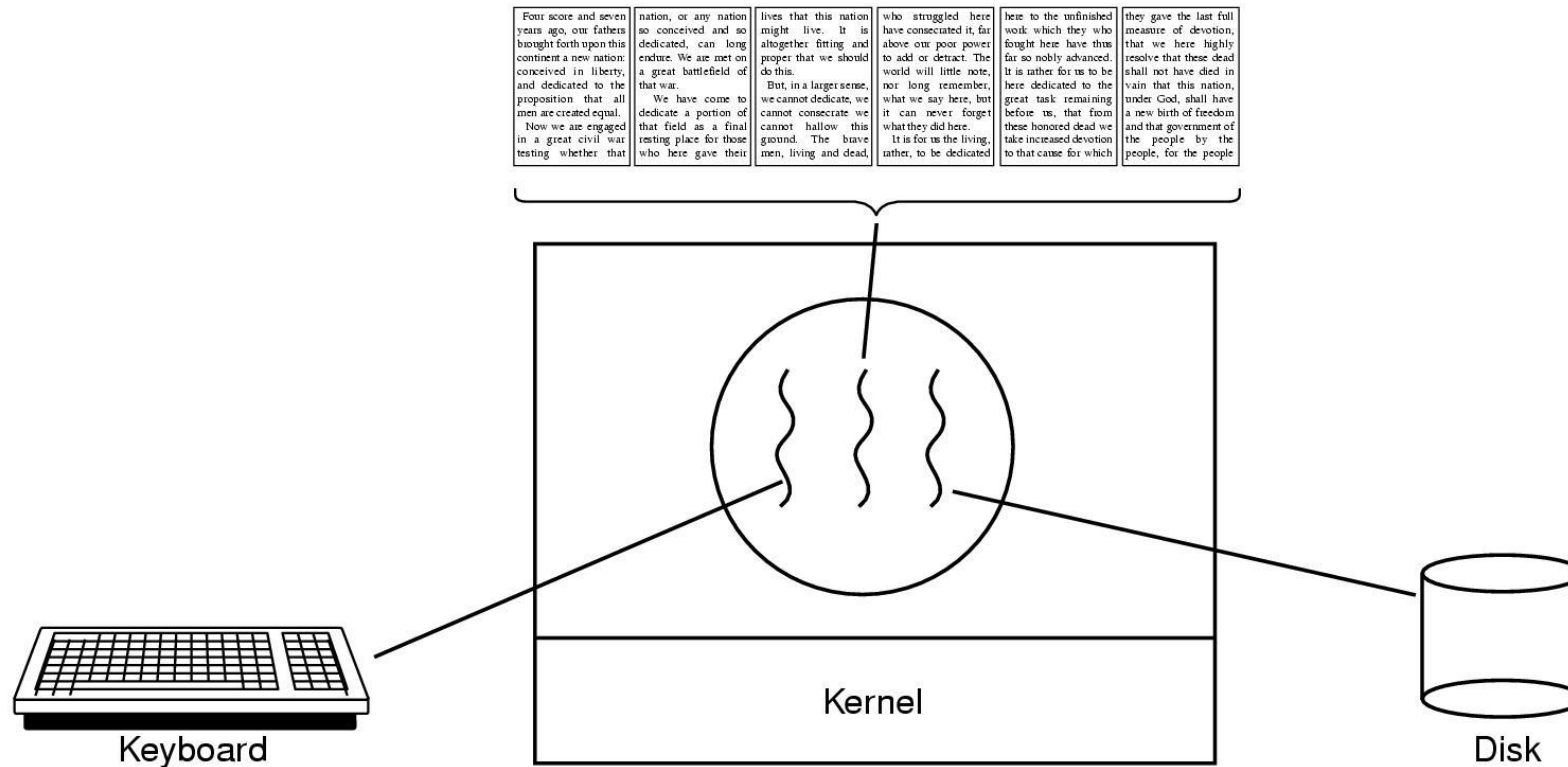
---

- É possível **executar em paralelo** cada uma das *threads* criadas para um mesmo processo usando diferentes CPUs.
- **Primitivas de sinalização** de fim de utilização de recurso compartilhado também existem. Estas primitivas permitem “acordar” uma ou mais *threads* que estavam bloqueadas.
- São **mais fáceis de criar e destruir** do que processos;
  - 100 vezes mais rapidamente do que criar um processo;

# Gerenciamento de Processos

## Threads

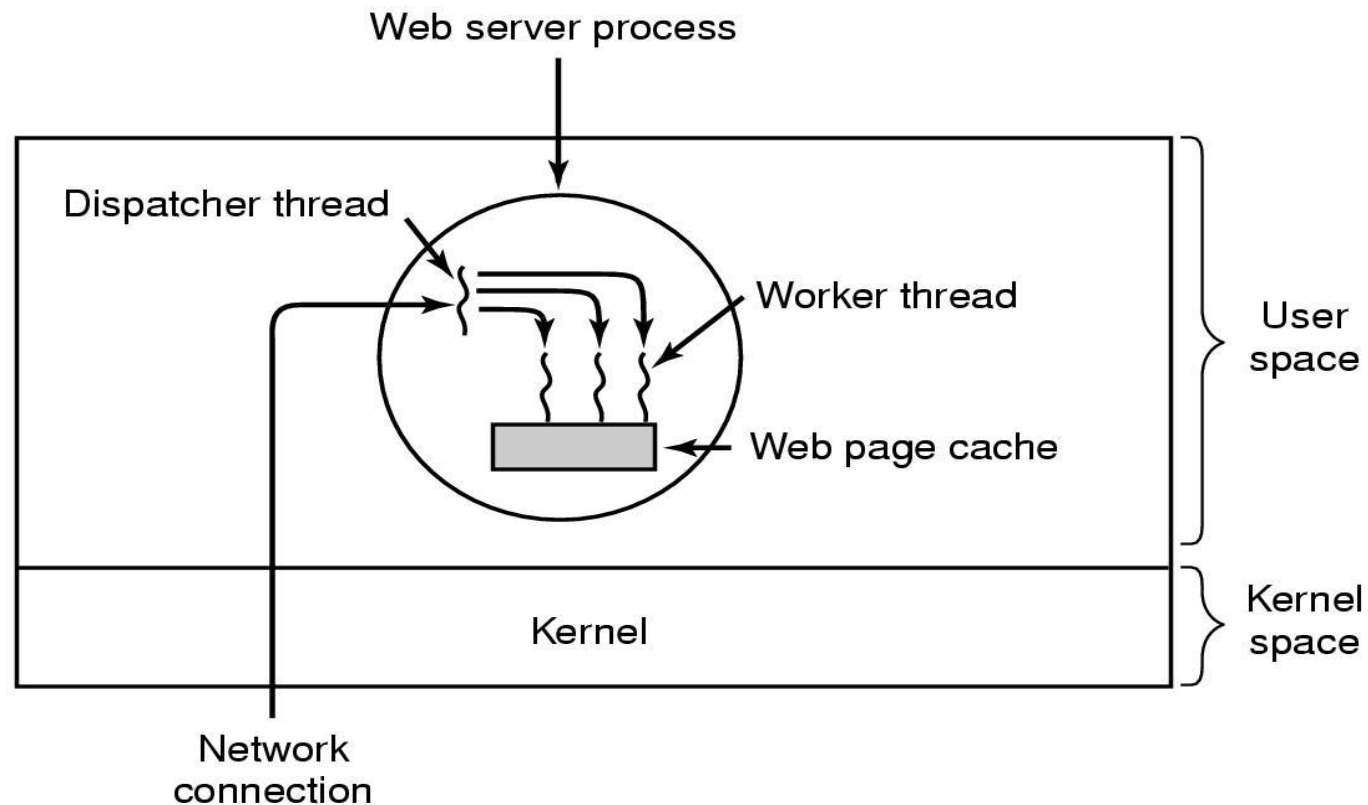
- Exemplos do uso de Threads
  - Um processador de texto com três Threads



# Gerenciamento de Processos

## Threads

- Exemplo do uso de Threads
  - Servidor Web multi-threaded



# Gerenciamento de Processos

## Threads

- Exemplo do uso de Threads
  - Servidor Web multi-threaded

```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

(a)

```
while (TRUE) {  
    wait_for_work(&buf)  
    look_for_page_in_cache(&buf, &page);  
    if (page_not_in_cache(&page)  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

(b)

- (b) Thread Trabalhador

# Gerenciamento de Processos

## User-level threads (ULT) – nível de usuário

---

### Implementação de Threads no Espaço do Usuário

- Implementadas através de bibliotecas;
  - Vantagem para S.O que não suportam Threads;
- O kernel não sabe sobre as Threads;
- As Threads são gerenciadas através de chamadas a procedimento:
  - `thread_create`, `thread_exit`, `thread_wait`, e `thread_yield`

# Gerenciamento de Processos

## Threads

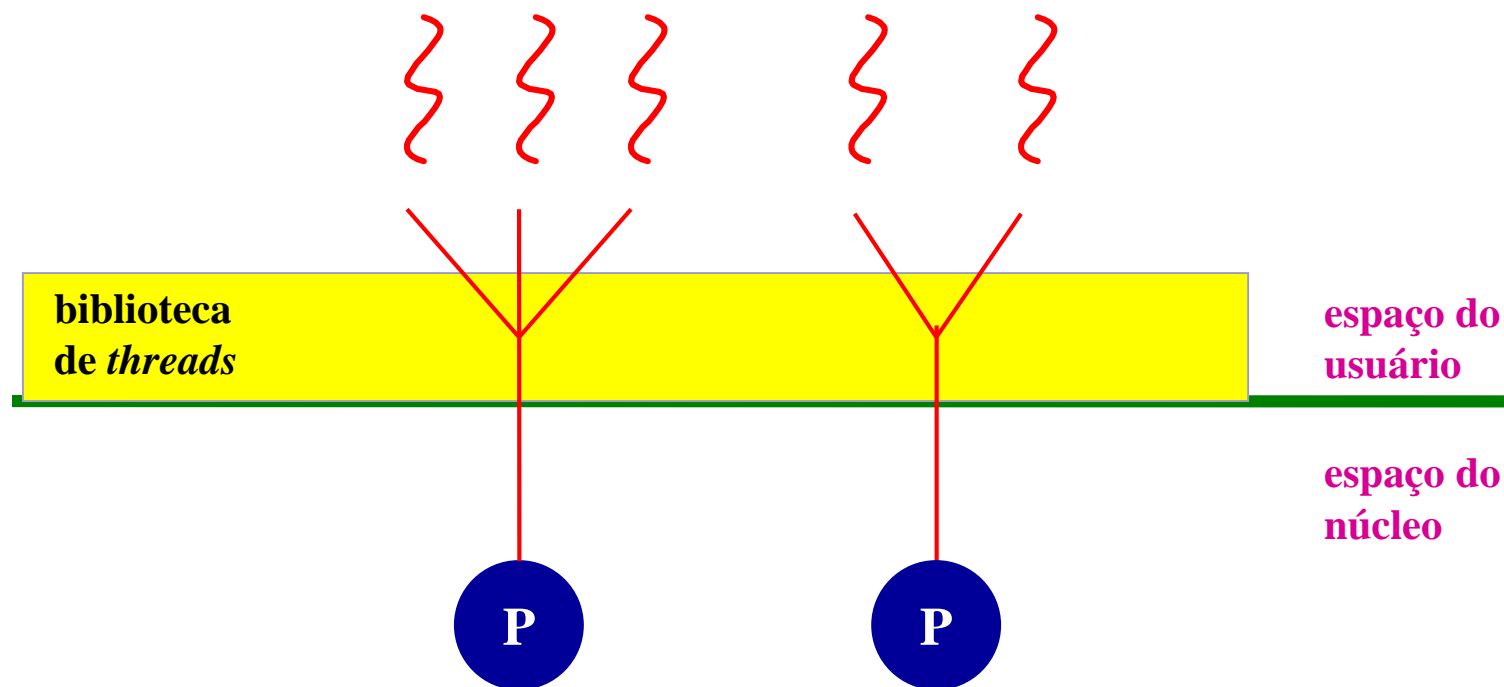
---

- Implementação de Threads no Espaço do Usuário:
  - Cada processo têm sua própria tabela de Threads;
    - Guarda informações referentes a cada Thread: contador de programa da cada Thread, o ponteiro de pilha, os registros, o estado, etc..
  - Esta tabela é gerenciada pelo sistema em tempo de execução;
  - É utilizada para salvar as informações da Thread quando sai da CPU ou quando reinicia na CPU.

# Gerenciamento de Processos

## Threads

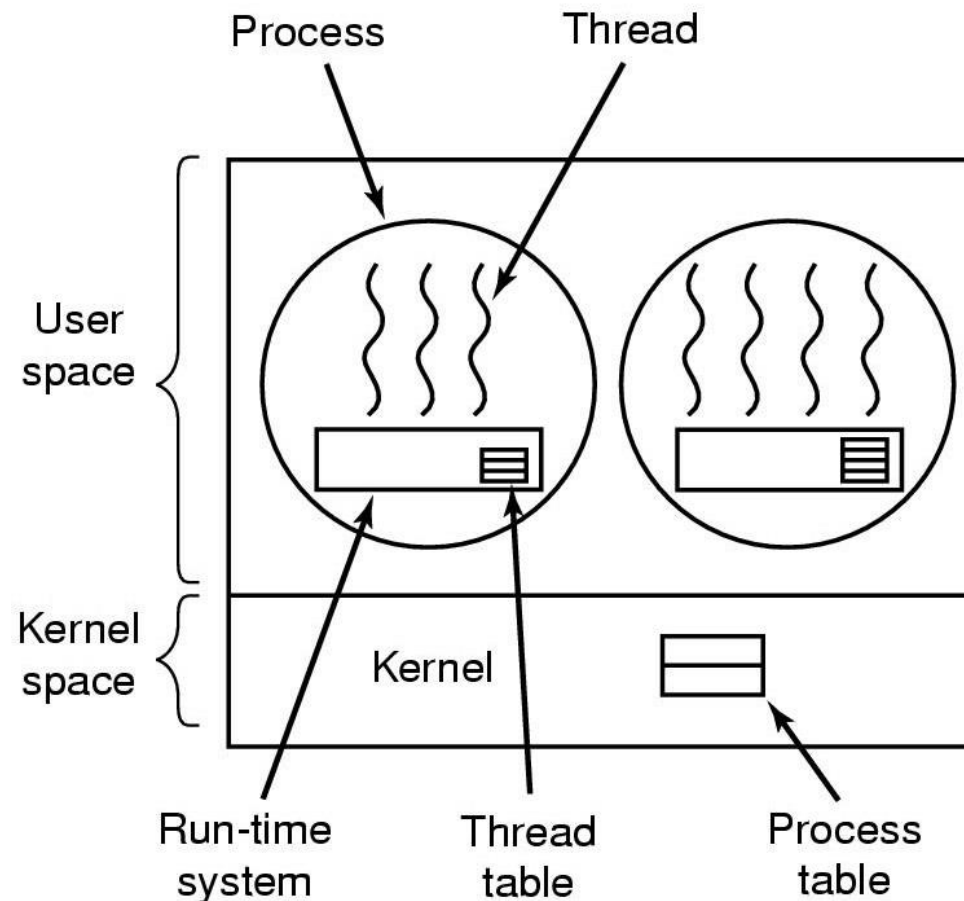
- Implementação de Threads no Espaço do Usuário:



# Gerenciamento de Processos

## Threads

- Implementação de Threads no Espaço do Usuário:





# Gerenciamento de Processos

## Threads

---

- Implementação de Threads no Espaço do Usuário:
  - A troca de Threads é similar a troca de processos na CPU;
  - Quando uma Thread sai da CPU, apenas as Threads do processo podem entrar para executar (de outro não);
  - Quando as inf. da tabela de Threads é carregada e o ponteiro de pilha e o contador de programa forem trocados a Thread passa a executar;
  - O processo de troca é mais rápido que no modo normal
    - Chamada local e não via Kernel.

# Benefícios das User Level Threads

---

- O **chaveamento das *threads*** não requer privilégios de *kernel* porque todo o gerenciamento das estruturas de dados das *threads* é feito **dentro do espaço de endereçamento de um único processo de usuário**.
  - Economia de duas trocas de contexto: user-to- kernel e kernel-to-user.
- O **escalonamento** pode ser específico **da aplicação**.
  - Uma aplicação pode se beneficiar mais de um escalonador Round Robin, enquanto outra de um escalonador baseado em prioridades.
- **ULTs** podem executar em **qualquer S.O.** As bibliotecas de código são portáteis.

# Gerenciamento de Processos

## Threads

---

### Alguns Problemas

- Implementação de **chamadas sistema Bloqueante**:
  - Evitar interferência entre Threads:
  - Exemplo: Page Fault – bloqueia todo o processo (Kernel não sabe da existencia de Threads)
- Se uma Thread inicia executando , nenhuma outra Thread executará se a primeira não **liberar a CPU voluntariamente**;
  - Solução : interrupção por relógio
- Implementação de Threads feita por **programadores inexperientes**;



---

# **Gerenciamento de Processos**

## **Threads - Kernel-level Threads - KLT**

# Gerenciamento de Processos

## Threads - Kernel-level Threads - KLT

---

### Implementação de Threads no Kernel:

- O Kernel sabe da existência e gerência as Threads;
  - Sistema em tempo de execução não é necessário;
- Não existe tabela de Threads em cada processo;
- A tabela de Threads é no Kernel e possui todas as Threads do sistema;
- Para criação e destruição de Threads é necessário uma chamada de sistema ao Kernel para atualização da Tab. Threads;

# Gerenciamento de Processos

## Threads

---

- Implementação de Threads no Kernel:
  - A tabela de Threads mantém os registradores, estado e outras informações de cada Thread;
  - No Kernel também é mantida a Tabela de processos normalmente;
  - Quando uma Thread sai da CPU (mesmo bloqueada) uma outra Thread do mesmo processo ou de outro processo pode executar;

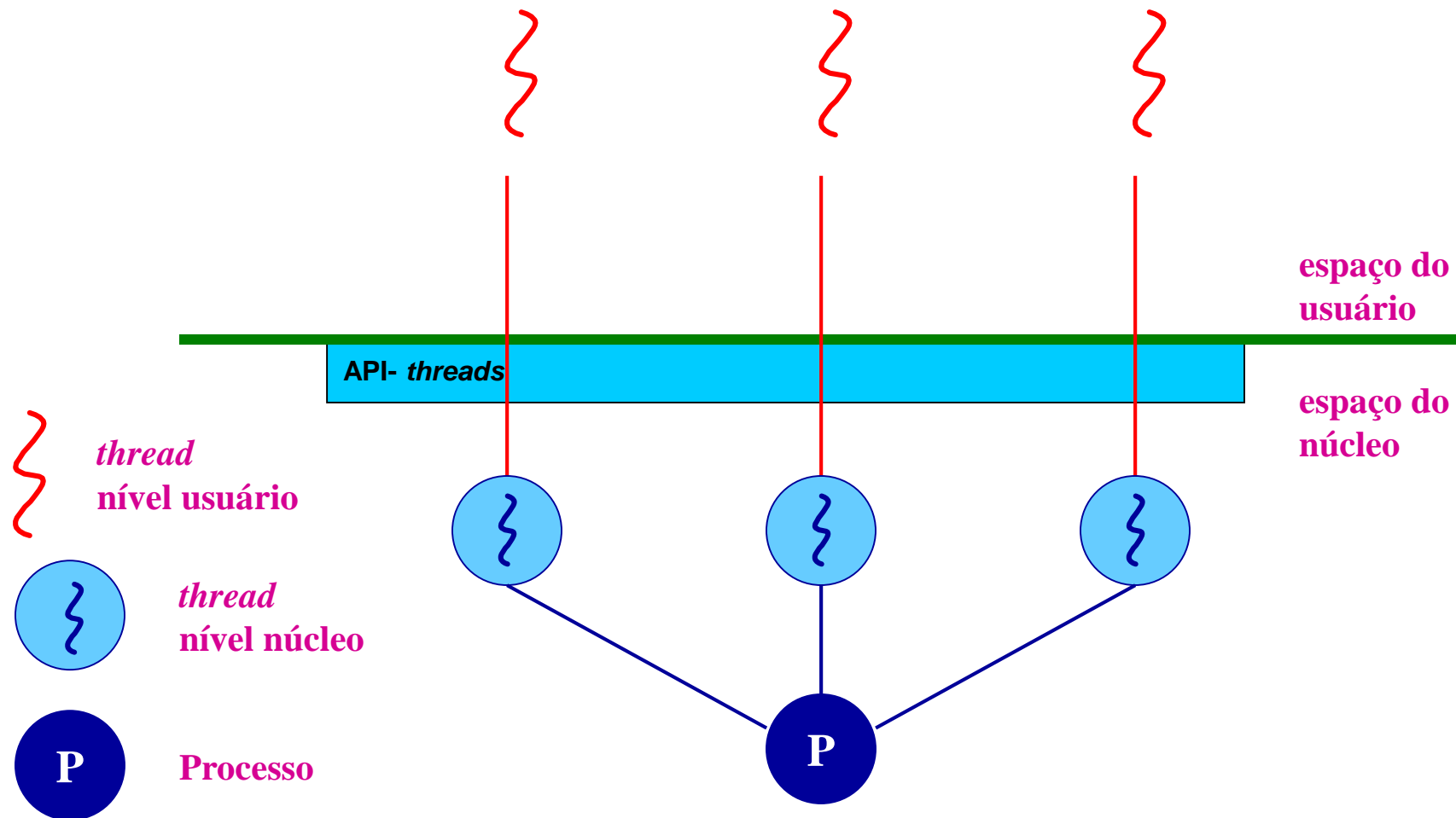
# Gerenciamento de Processos

## Threads

---

- Implementação de Threads no Kernel:
  - Devido ao alto custo de criação e destruição da Threads no Kernel, normalmente as Threads são recicladas;
    - Marcadas como não executáveis.
  - Não há necessidade de **chamadas de sistemas**
- **Bloqueante:**
  - O Kernel sabe da existência da Thread e simplesmente coloca a Thread no estado Bloqueado e passa a executar outra Thread.

# Kernel-level Threads - KLT (1)

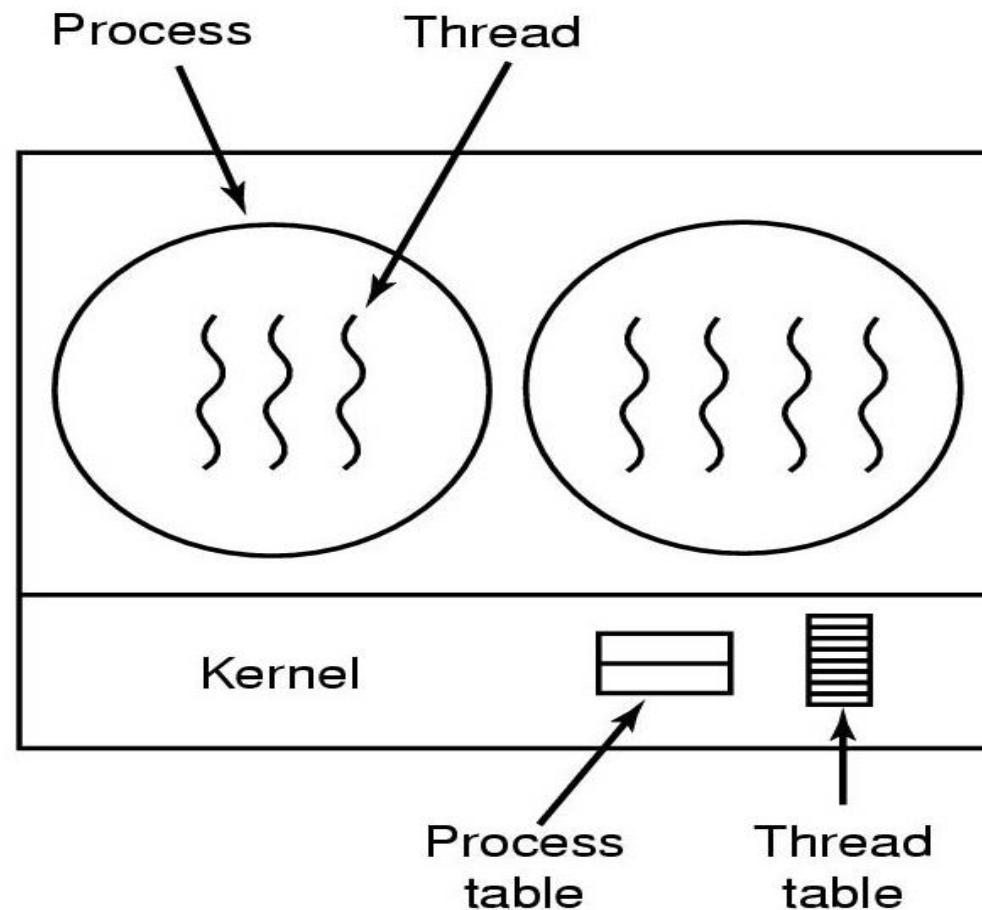




# Gerenciamento de Processos

## Threads

- Implementação de Threads no Kernel:

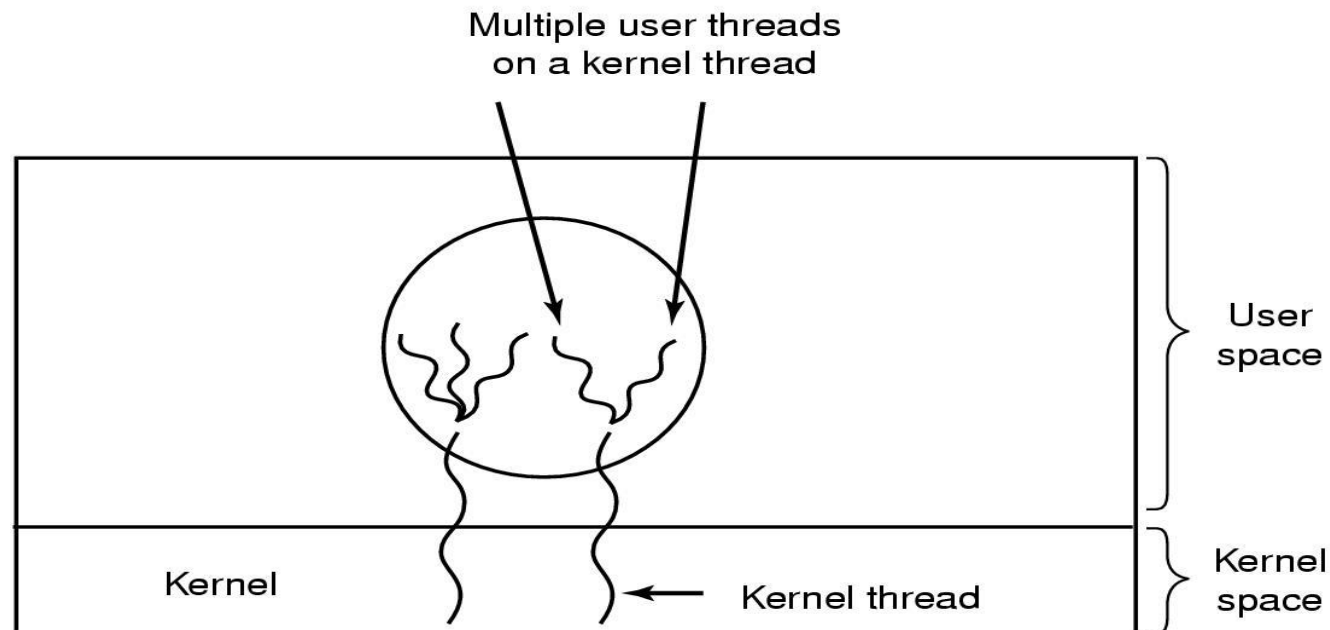


# Gerenciamento de Processos

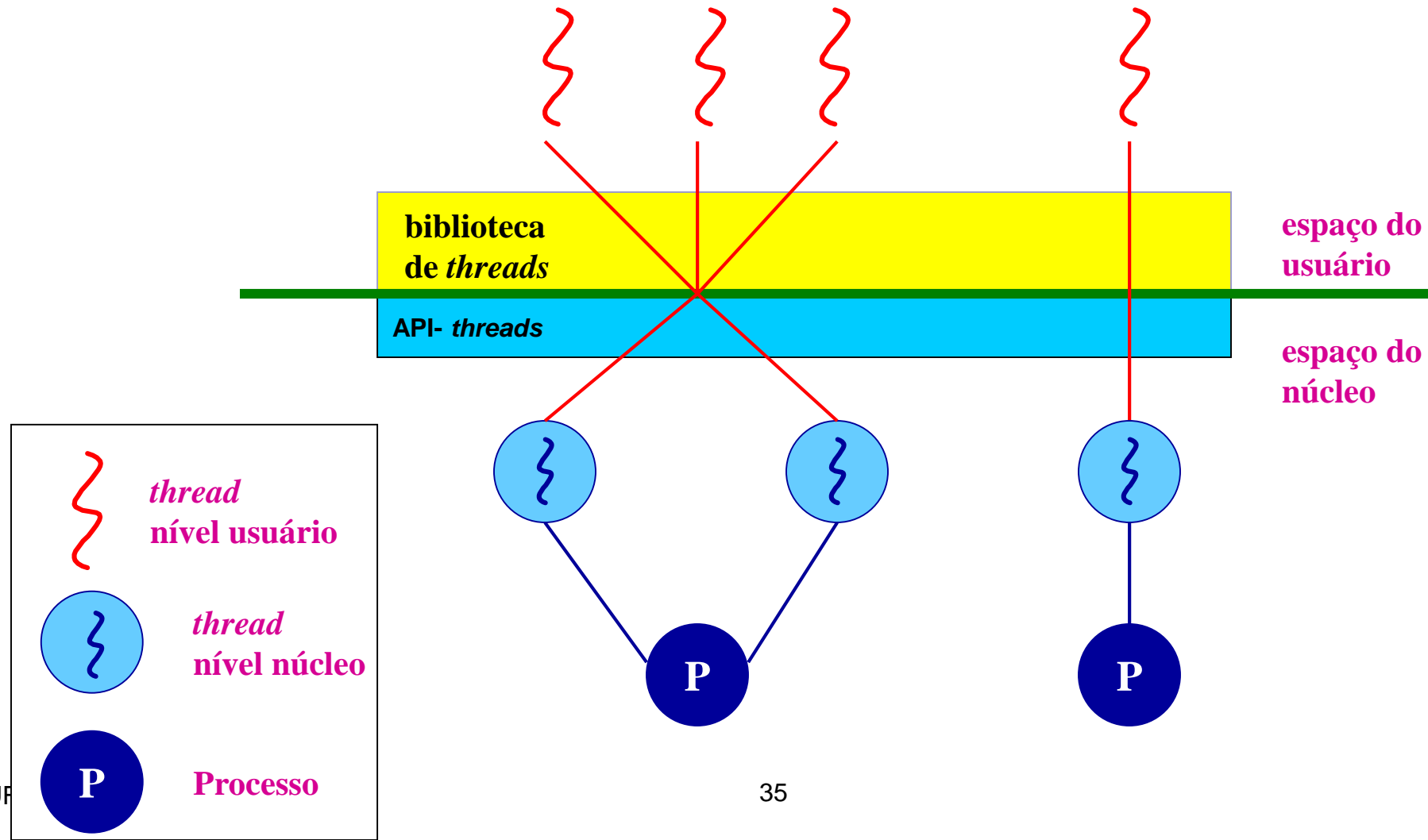
## Threads – Híbrida

### Implementação Híbrida:

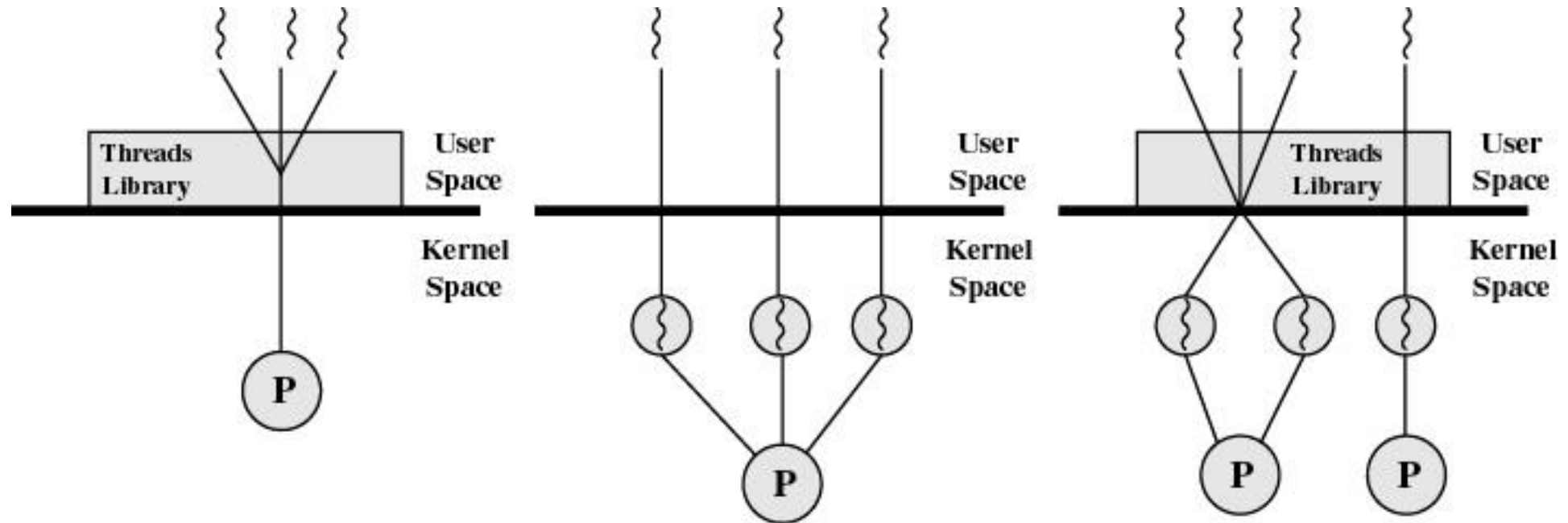
- Multiplexar Threads a nível do Usuário sobre algumas ou todas as Threads do Kernel;
- O kernel está ciente somente das Threads a nível de Kernel e escalona elas.



# Combinando Modos



# Resumindo ...



(a) Pure user-level

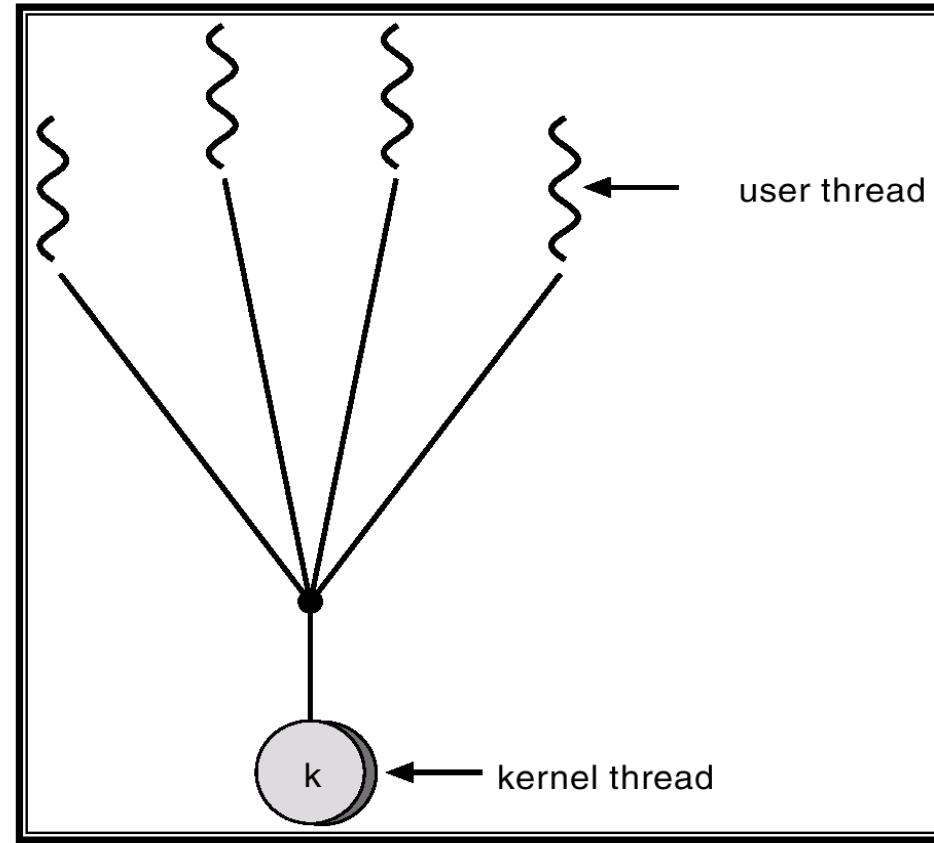
(b) Pure kernel-level

(c) Combined

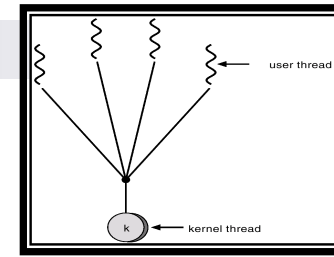


# Resumindo... Modelo M:1 - ULT

- Muitas *user-level threads* mapeadas em uma única *kernel thread*.
- Modelo usado em sistemas que não suportam *kernel threads*.



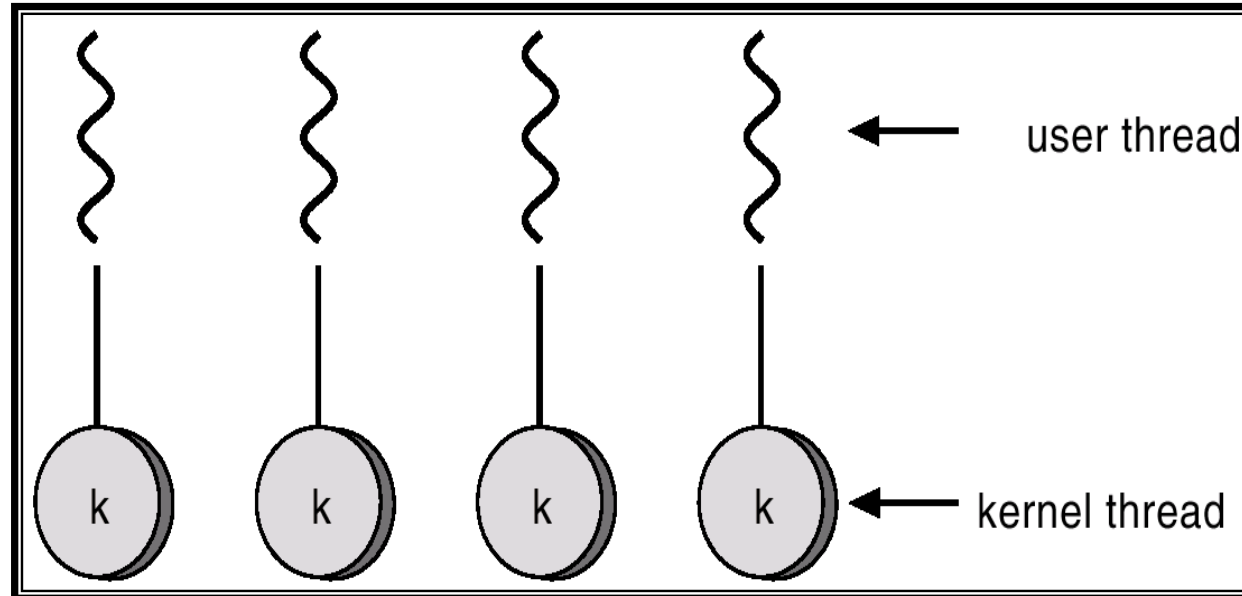
# Resumindo... Modelo M:1



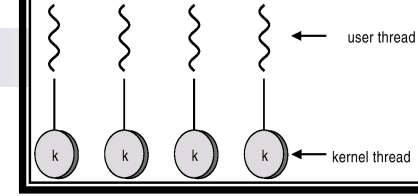
- Mapeia muitas threads em uma thread no kernel;
- A gerencia das threads é no espaço do usuário (+eficiente, + bloqueio)
- Apenas uma thread por vez pode acessar o kernel, **não é possível executar multiplas threads em multiprocessadores;**
- Bibliotecas de threads que não suportam threads no kernel são muitos para um.

# Resumindo ... Modelo 1:1 - KLT

- Cada *user-level thread* é mapeada em uma única *kernel thread*.
- Exemplos: Windows 95/98/NT/2000 e OS/2



# Resumindo ... Modelo 1:1

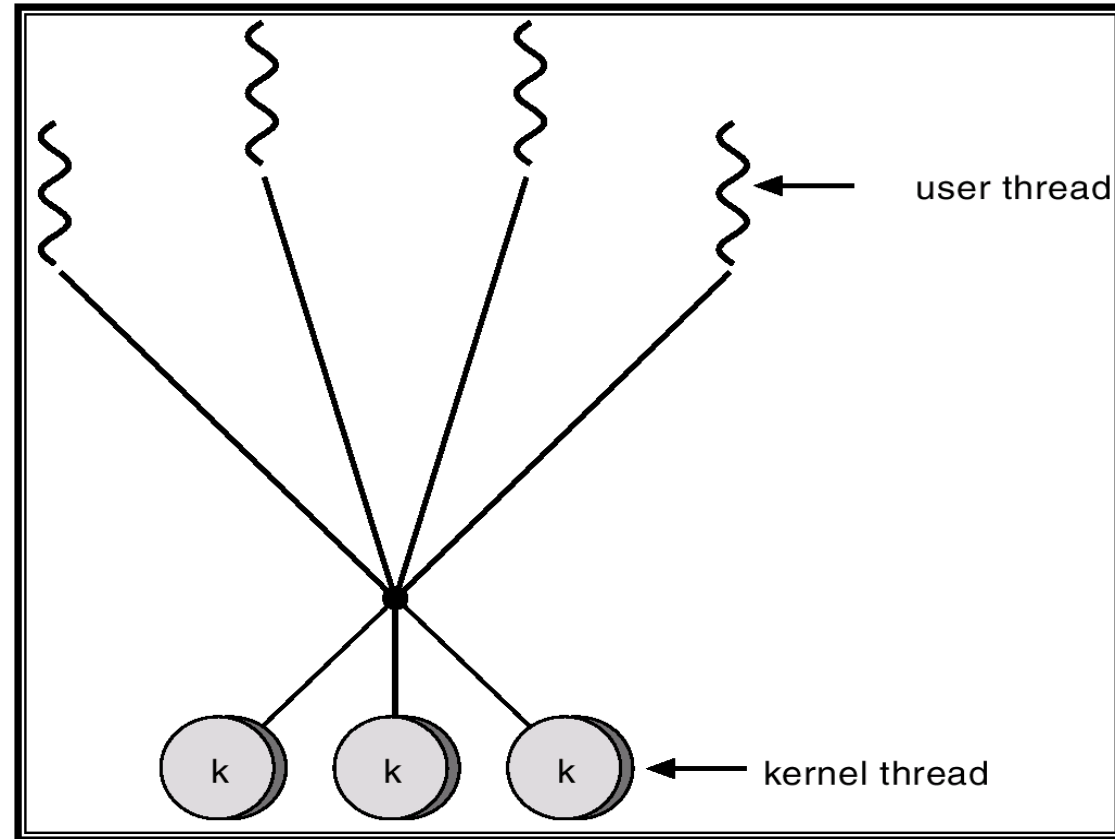


- Mapeia cada thread de usuário em uma thread do kernel;
- Fornece mais concorrência (se uma bloqueia a outra pode continuar);
- É possível executar em multiprocessadores;
- Pode ter o desempenho prejudicado – criar threads no kernel;
- Usado no NT e OS/2

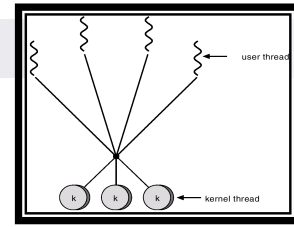


# Resumindo... Modelo M:n

- Permite que diferentes *user-level threads* de um processo possam ser mapeadas em *kernel threads* distintas.
- Permite ao S.O. criar um número suficiente de *kernel threads*.
- Exemplos: Solaris 2, Tru64 UNIX's, Windows com o *ThreadFiber* package.




# Resumindo... Modelo M:n



- Multiplexa muitas threads de usuário em um numero menor ou igual de threads de kernel;
- Fornece mais concorrência, sem restringir o numero de threads (um-para-um ocorre) e com concorrência real (muitos-para-um);
- Os desenvolvedores podem criar tantas threads quanto forem necessárias e as threads de kernel correspondentes podem executar em paralelo em um multiprocessador;
- Chamadas bloqueantes não bloqueiam os processos, o kernel escalona outra thread para execução.



# Sincronização entre Threads - JAVA



# Roteiro Geral

## Sincronização entre Threads

---

- **expressão da concorrência em Java**
  - conceitos de threads e Java threads
  - criação de threads: por herança e por interface
  - exemplo
- **sincronização em Java**
  - conceitos de monitores
  - exclusão mútua em Java threads
  - sinalização em Java threads
  - exemplo
  - outros recursos de Java threads

# Sincronização entre Threads

## Concorrência

---

### ■ Conceitos de threads Java:

#### □ uma thread Java é

- um fluxo de controle, um processo leve, um objeto do tipo Thread equivale a um fluxo de controle de um processo Java (programa em execução) contendo:

- dados;
- prioridades;
- métodos do pacote Thread;
- métodos da aplicação;

- uma thread Java permite a um programa Java ter mais de um fluxo de execução;

# Sincronização entre Threads

## Concorrência

---

### ■ Conceitos de threads Java

#### □ Uma thread Java possui métodos como:

- run()
  - código (principal) do fluxo da thread
- setPriority()
  - altera a prioridade de uma thread
- start()
  - inicia a execução da thread
- join()
  - uma thread (fluxo) espera pelo fim da execução de outra thread
- vários outros métodos

# Sincronização entre Threads

## Concorrência

---

### ■ Conceitos de threads Java

#### □ Vantagens das Threads (sobre processos)

- mais eficientes (rápidas), por exemplo, na criação
- consomem menos recursos (dados -> memória)
- compartilham dados (objetos em Java)

#### □ Restrições das Threads

- contexto limitado a um processo ou programa (usual para objetos Java)
- a ação de um método é limitado às threads do programa
- exemplo:
  - método wait(): a outra thread deve ser do mesmo programa

# Sincronização entre Threads

## Concorrência

---

### ■ Escalonamento

#### □ Conceito:

- função para determinar quando e por quanto tempo uma thread é executada (a cpu é alocada à thread)

#### □ Implementado por:

- nas 1as versões pela máquina virtual de Java (JVM)
- atualmente por 2 opções
  - como nas 1as versões
  - de forma híbrida pela JVM e por um pacote de threads (SO, ..)

#### □ conforme esquema de escalonamento preemptivo, baseado em prioridades



# Sincronização entre Threads

## Concorrência

---

### ■ Aplicações de Threads

- ☐ entrada e saída assíncrona;
- ☐ tratamento assíncrono de requisições em sistemas;
- ☐ cliente/servidor;
- ☐ processos de serviço em retaguarda (background); por exemplo, coletores de lixo (garbage collectors);
- ☐ sistemas reativos que precisam tratar certos eventos;
- ☐ rapidamente usando escalonamento baseado em prioridades;
- ☐ processamento paralelo: uso do poder de computação das máquinas multiprocessadoras;
- ☐ jogos, simuladores, etc.

# Sincronização entre Threads

## Prioridades e Nomes

---

### ■ Prioridades de threads Java

#### □ Constantes:

- máxima: MAX\_PRIORITY = 10
- mínima: MIN\_PRIORITY = 1
- normal: NORM\_PRIORITY = 5

#### □ Nomes de Threads Java:

- Toda Thread tem um Nome;
- Dado pelo Usuário:
  - construtores com arg nome
- ou Dado pelo Sistema:
  - construtores sem arg nome
- 2 ou mais threads podem ter o mesmo nome;

# Sincronização entre Threads

## Prioridades e Nomes

### ■ Prioridades de threads Java

```
class BaixaPrioridade extends Thread
{
    public void run() {
        setPriority(Thread.MIN_PRIORITY);
        for(;;) {
            System.out.println("Thread de baixa prioridade
            executando -> 1");
        }
    }
}

class AltaPrioridade extends Thread {
    public void run() {
        setPriority(Thread.MAX_PRIORITY);
        for(;;) {
            for(int i=0; i<5; i++)
                System.out.println("Thread de alta prioridade
                executando -> 10");
            try {
                sleep(100);
            } catch (InterruptedException e) {
                System.exit(0);
            }
        }
    }
}
```

# Sincronização entre Threads

## Prioridades e Nomes

---

### ■ Prioridades de threads Java

```
class Lançador {  
    public static void main(String args[]) {  
        AltaPrioridade a = new AltaPrioridade();  
        BaixaPrioridade b = new BaixaPrioridade();  
        System.out.println("Iniciando threads...");  
        b.start();  
        a.start();  
        // deixa as outras threads iniciar a execução.  
        Thread.currentThread().yield();  
        System.out.println("Main feito");  
    }  
}
```

# Sincronização entre Threads

## Criação de Java threads

---

### ■ Criação de threads Java


#### □ Dois Métodos:

##### ■ A: por herança

- lembrar que herança é simples em Java
- classe da thread deve ser “nova”

##### ■ B: por interface

- permite tornar uma classe existente em thread



# Sincronização entre Threads

## Criação de Java threads

---

### ■ Criação de threads

#### □ método A:

- **Estender a classe Thread** definindo uma nova subclasse;
- **Reescrever o método run();**
- Criar um objeto da nova subclasse;

# Sincronização entre Threads

## Criação de Java threads

---

### ■ Criação de threads

#### □ método B:

- definir uma classe C que **implementa a interface Runnable**;
- **criar um objeto O** da classe C;
- **criar um objeto do tipo Thread** (classe) passando o objeto O como parâmetro;
- importante: objeto da classe C não é Thread;

# Sincronização entre Threads

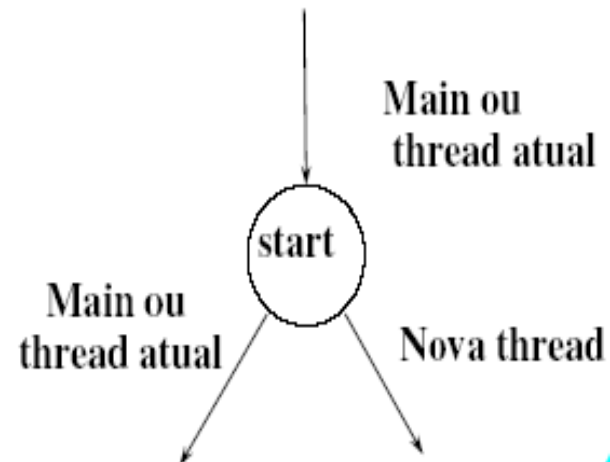
## Criação de Java threads

---

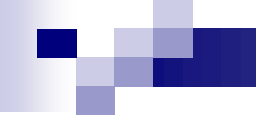
### ■ Criação de threads

#### □ em ambos os métodos:

- objeto thread foi somente criado;
- **ativação** da execução (fluxo) da thread:
  - pelo método **start()**
    - semântica assíncrona
    - similar à do fork do Unix







# Sincronização entre Threads

## Criação de Java threads

---

### ■ Exemplo de criação pelo método A

#### □ Aplicação

- criação de 3 threads pelo main
- cada thread
  - dorme um certo tempo específico;
  - imprime seu nome;
  - tempo e nome: dados pelo programador;

# Sincronização entre Threads

## Criação de Java threads

---

### ■ Exemplo de criação pelo método A

```
public class MyThread extends Thread {
    private String whoami;
    private int delay;

    public MyThread(String name, int d) {
        whoami = name;
        delay = d;
    }

    public void run() {
        try {
            sleep(delay);
        } catch (InterruptedException e) {}
        System.out.println("Hello, this is
        "+whoami+" ! ");
    }
}
```

# Sincronização entre Threads

## Criação de Java threads

---

### ■ Exemplo de criação pelo método A (cont.)

```
public class TestThreads {  
    public static void main(String[] args) {  
        MyThread t1, t2, t3;  
        t1 = new MyThread("First", 1000);  
        t2 = new MyThread("Second", 500);  
        t3 = new MyThread("Third", 2000);  
  
        t1.start();  
        t2.start();  
        t3.start();  
    }  
}
```

execução provável:

- Hello, this is Second!
- Hello, this is First!
- Hello, this is Third!

# Sincronização entre Threads

## Criação de Java threads

---

### ■ Criação thread com interface Runnable

- a classe Thread também é implementada como Runnable
- Runnable é uma interface do ambiente Java
- código da classe:

```
class ThreadBody implements Runnable {  
    ...  
    public void run() {  
        // código do corpo da thread  
    }  
}
```

# Sincronização entre Threads

## Criação de Java threads

---

- **Exemplo de criação thread usando Runnable**


- **código do uso:**

- passa objeto Runnable (ThreadBody) à criação de objeto Thread

```
Thread t = new Thread (new ThreadBody());  
t.start();
```

- **vantagem: classe thread (ThreadBody) pode estender uma outra superclasse**

- diferente da classe Thread



# Sincronização entre Threads

## Criação de Java threads

---

### ■ Outras características das threads Java

- uma thread Java pode ter métodos do usuário;
- os métodos do usuário (e da classe Thread) podem ser chamados:
  - antes, durante e depois da execução do run();
  - se durante: execução concorrente ao run();

# Sincronização entre Threads

## Criação de Java threads

---

### ■ Método join()

#### □ semântica similar à da primitiva join do conceito fork/join

##### ■ “waitpid” no Unix

##### ■ 3 formas

##### □ join(): simples

##### □ join(long milisegundos)

- espera até milisegundos
- se zero: espera infinita

##### □ join(long milisegundos, long nanosegundos)

- espera milisegundos mais nanosegundos

• FORK – Inicia a execução de outro programa concorrentemente

• JOIN – O programa chamador espera o outro programa terminar para continuar o processamento

# Sincronização entre Threads

## Criação de Java threads

---

### ■ Método join()

- thread main

- `t1 = new MyThread();`  
`t1.start();`  
`...`  
`t1.join();`  
`...`



# Sincronização entre Threads

## Criação de Java threads

---

### ■ Método join()

```
public class MeuThread extends Thread
{
    public MeuThread(String nome)
    {
        super(nome);
    }
    public void run() // o metodo que vai ser executado no thread tem sempre nome run
    {
        for (int i=0; i<5; i++)
        {
            System.out.println(getName()+ " na etapa:"+i);
            try
            {
                sleep((int) (Math.random() * 2000)); //milisegundos
            } catch (InterruptedException e) {}
        }
        System.out.println("Corrida de threads terminada:" + getName());
    }
}
```

# Sincronização entre Threads

## Criação de Java threads

---

### ■ Método join()

```
class CorridaThreads
{
    public static void main (String args[])
    {
        MeuThread a,b;
        a=new MeuThread("Leonardo Xavier Rossi");
        a.start();
        b=new MeuThread("Andre Augusto Cesta");
        b.start();
        try {a.join(); } catch (InterruptedException ignorada) {}
        try {b.join(); } catch (InterruptedException ignorada) {}
    }
}
```

# Sincronização entre Threads

## Quantidades de Threads

---

### ■ Quantidades de threads

- ☐ por JVM
- ☐ criação de threads até a falha da JVM
- ☐ cada thread executou um comando `sleep(999999999999999999)`

# Sincronização entre Threads

## Quantidades de Threads

---

### ■ Quantidades de threads

#### □ Resultados

- Máquina TINI (chip com JVM):
  - 64 threads
  - informação da documentação
- Máquina Sun UltraSparc 10 com 128 MB e 100 MB de heap na JVM:
  - 954.412 threads
  - SO: Solaris
- Máquina Sun UltraSparc 10 com 512 MB e 400 MB de heap na JVM:
  - 3.939.782 threads
  - SO: Solaris

# Sincronização entre Threads

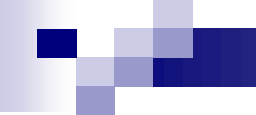
## Quantidades de Threads

---

### ■ Quantidades de Threads

#### □ Considerações

- de acordo com os experimentos e não havendo limitação explícita na especificação de Java nem na documentação do JDK, é possível:
  - que o número máximo de threads dependa principalmente da memória disponível;
  - e, eventualmente, de limites impostos pela implementação da JVM (principalmente em sistemas tempo real e embarcados).



# Sincronização entre Threads

## **Considerações sobre o uso de threads**

---

- **Considerações sobre o uso de threads**
  - **o uso de threads tem um custo específico**
    - criação, memória, trocas de contexto, sincronização, ...
  - **antes de usá-las analisar e comparar benefícios x custos:**
    - benefícios
      - eventual paralelismo (várias cpus);
      - facilidade de expressão de tarefas concorrentes
        - simulações, jogos, ...
      - tratamento assíncrono de I/O e troca de mensagens

# Sincronização entre Threads

## Sincronização

---

### ■ Conceito Geral de Monitores:

- Um Monitor Consiste de:
  - estruturas de dados;
  - coleção de *procedures* que operam sobre as estruturas do monitor;
  - conceito de dados protegidos:
    - as estruturas só podem ser acessadas pelas *procedures* do monitor

# Sincronização entre Threads

## **sincronização**

---

### ■ **Conceito Geral de Monitores**

#### □ Exclusão Mútua:

- Somente uma thread cliente pode executar uma procedure do monitor em um momento;

#### ■ Gera fila de entrada no monitor:

- threads bloqueadas esperando sua vez



# Sincronização entre Threads

## **sincronização**

---

### ■ **Conceito Geral de Monitores**

#### □ variáveis de condição

- variáveis especiais sobre as quais podem ser chamadas as operações wait e signal
- apenas dentro do monitor

# Sincronização entre Threads

## **sincronização**

---

### ■ conceito geral de monitores

#### □ operação wait:

- coloca a própria thread em estado bloqueado (dormindo) numa lista de espera do monitor
- isto permite a execução de outra thread
- lista de espera: associada à variável de condição
- `vc1.wait()`

# Sincronização entre Threads

## **sincronização**

---

### ■ conceito geral de monitores

#### □ operação signal:

- acorda uma outra thread que está na lista de espera associada à variável de condição
- `vc1.signal();`
- questão: qual thread continua sua execução imediatamente?
  - a que acorda?: mais eficiente
  - a acordada?: mais seguro

# Sincronização entre Threads

## **sincronização**

---

- **Conceito Geral de Sincronização de Threads em Java:**

- **Monitores em Java:**

- um objeto qualquer, compartilhado por várias threads (“servidor”), pode ser visto como um monitor;
    - diversas diferenças com relação ao conceito clássico de monitor;

# Sincronização entre Threads

## sincronização

---

- Sincronização: **modificadores** e métodos básicos
  - Synchronized em Método de Instância (não estático)
    - modifica método declarando que o mesmo faz parte de monitor;
    - o monitor é o (um) objeto;
    - impõe exclusão mútua (lock/unlock) na execução de todos os métodos com **synchronized** do objeto:
      - métodos sem synchronized não verificam o lock
        - são executados concorrentemente entre si e com relação aos synchronized;
      - existe um lock associado a cada objeto da classe;

# Sincronização entre Threads

## **sincronização**

---

- Sincronização: **modificadores** e métodos básicos
  - `synchronized` em método de instância (não estáticos)
    - importante:
      - o normal é execução concorrente dos métodos;
      - para ter exclusão mútua, é preciso usar o `synchronized`;

# Sincronização entre Threads

## **sincronização**

---

- Sincronização: **modificadores** e métodos básicos
  - **synchronized em métodos estáticos**
    - o lock é um objeto associado à classe
    - independente dos locks por objetos
    - métodos estáticos sem synchronized
      - idem métodos de instância

# Sincronização entre Threads

## sincronização

---

### ■ Sincronização: **modificadores** e métodos básicos

#### □ **Deadlock**

- ocorre se (por exemplo)
  - thread A tem lock 1 (associado a objeto x)
  - thread B tem lock 2 (associado a objeto y)
  - thread A pede lock 2
    - chama método de y
  - thread B pede lock 1
    - chama método de x
- não ocorre se
  - thread tem lock 1
  - chama outro método associado ao mesmo lock
    - método do mesmo objeto



# Sincronização entre Threads

## sincronização

---

- Sincronização: **modificadores** e métodos básicos

- **synchronized em bloco de comandos**

- esqueleto de código em método:

```
...  
synchronized (objetoQualquer) {  
    ...  
    "bloco de comandos"  
    ...  
}.  
..
```

- objetoQualquer: funciona como um lock
      - todos os blocos synchronized com o mesmo objetoQualquer são executados de forma atômica

# Sincronização entre Threads

## **sincronização**

---

- Sincronização: **modificadores** e métodos básicos

- **synchronized** em bloco de comandos

- o objeto lock (objetoQualquer) pode ser um objeto externo ao objeto onde está inserido o código sincronizado
    - nesse caso, pode-se expressar blocos “independentes”, em diferentes objetos e classes, mas compartilhando o mesmo lock
      - **condição**: passar o objeto lock a todos os objetos com o mesmo bloco
        - no construtor, por exemplo;

# Sincronização entre Threads

## sincronização

---

### ■ Sincronização: **modificadores** e métodos básicos

#### □ **synchronized** em bloco de comandos:

##### ■ Métodos Estáticos:

- com variável de classe
  - obrigatoriamente: método estático não tem acesso direto a variáveis de instâncias
  - ou objeto passado como argumento

##### ■ Métodos de Instância:

- com variável de classe
  - opcional: controla acesso a variáveis estáticas
- com variável de instância
  - para controlar variáveis de classe: usar objeto lock único em todos os objetos

# Sincronização entre Threads

## **sincronização**

---

- Sincronização: **modificadores** e métodos básicos

- **synchronized** em bloco de comandos

- Observação:

- o lock está sempre associado a um objeto e não a uma variável (um nome simbólico em um único contexto - objeto);
      - um objeto pode estar sendo referenciado em n variáveis (usual em OO).

# Sincronização entre Threads

## **sincronização**

---

### ■ Sincronização: modificadores e **métodos básicos:**

#### ☐ **wait()**

- inclui a thread em execução na lista de espera do monitor e permite a execução de outra thread

#### ☐ **notify()**

- Acorda uma thread da lista de espera do monitor
  - ☐ a condição de somente uma thread no monitor continua válida
  - ☐ acorda qual thread?: a que espera há mais tempo (escalonamento justo)?

# Sincronização entre Threads

## **sincronização**

---

### ■ Sincronização: modificadores e **métodos básicos:**

#### □ **notify()**

- caso exista, escolhe-se uma thread de forma arbitrária, remove-se a thread da fila associada (lock) e coloca-se a thread em fila de ready
- a thread deve obter o lock, que (certamente) está com a thread que executou o notify, para continuar sua execução
  - uma outra thread pode obter o lock entretanto
- a thread corrente não perde o lock (estado running)

# Sincronização entre Threads

## **sincronização**

---

### ■ Sincronização: modificadores e **métodos básicos:**

#### ☐ **notifyall**

- diferença: acorda todas as threads bloqueadas na fila do lock associado ao notifyall

#### ☐ **wait**

- duas versões com time-out
- passado o tempo, a thread (ou threads) é acordada automaticamente

#### ☐ **wait e notify**

- runtime verifica se notify e wait estão dentro do bloco(ou método inteiro) sincronizado
  - ☐ não verifica se o lock é externo

# Sincronização entre Threads

## sincronização

---

### ■ Sincronização: modificadores e métodos básicos:

#### □ wait/notify em métodos estáticos

- wait/notify são métodos não estáticos e finais
  - não podem ser reescritos
  - não podem ser chamados pelo nome da classe
- alternativa em bloco synchronized em método estático
  - usar um objeto comum a toda a classe
    - um objeto do tipo Class, inicializado com o
    - objeto da classe específica
    - `Class c = AlgumaClasse.class`
  - um objeto único passado em todas as chamadas dos métodos estáticos (argumento)



# Sincronização entre Threads

## **sincronização**

---

### ■ **Considerações sobre o uso de Sincronização**

#### □ **Desenvolvimento Conservativo**

- Inicialmente coloque synchronized em todos os métodos das classes com objetos compartilhados;
- Teste e verifique a correção do programa;
- Teste e analise o desempenho do programa;
- Se desempenho for “ruim”:
  - Reexamine o código procurando;
    - métodos que não precisem de synchronized;
    - seções críticas mais finas que um método, isto é, uso de bloco synchronized;

# Sincronização entre Threads

## **sincronização**

---

- **Exemplo: classe que implementa uma fila**
  - **conceito de fila: ordem FIFO: First In, First Out**
    - inclusão (append): no fim (tail) da fila
    - retirada (get): do início (head) da fila
  - **fila monothread**
    - se a fila está vazia em retirada: retorna um aviso
    - se a fila está cheia em inclusão: idem

# Sincronização entre Threads

## **sincronização**

---

- **Exemplo: classe que implementa uma fila**

- **fila multithread**

- usuário ou cliente (processo, thread): espera em caso de exceção
      - fila vazia em retirada: até haver inclusão
      - fila cheia: considera-se buffer ilimitado
    - concorrência possível nos acessos a fila: head, tail, ...
      - várias threads inserindo e excluindo elementos

# Sincronização entre Threads

## **sincronização**

---

- **Exemplo: classe que implementa uma fila (cont.)**

```
classe Queue {  
    // first and last element of the queue  
    // structure: value of element and pointer to next elem.  
    Element head, tail;  
  
    // inserts element at the end of the queue  
    public synchronized void append(Element p) {  
  
        if (tail == null) // if queue is empty: simple case  
            head = p;    // first element is the new element  
        else  
            tail.next = p; // last elem. points to new element  
  
        p.next = null;    // next of last element is null  
        tail = p;        // last element is the new element  
        notify();        // notify that 1 element arrived  
    }  
}
```

# Sincronização entre Threads

## sincronização

---

- Exemplo: classe que implementa uma fila (cont.)

```
// gets an element of queue
public synchronized Element get() {

    try {
        while (head == null) // if queue is empty?
            wait();          // wait for an element
    } catch (InterruptedException e) {
        return null;
    }

    Element p = head;        // save first element
    head = head.next;        // take out of queue
    if (head == null)        // queue is empty?
        tail = null;        // last element null
    return p;
}
```

# Sincronização entre Threads

## **sincronização**

---

### ■ **Controle de threads:** outros métodos nativos

#### □ **observações**

- esses métodos são em geral chamados por outra thread (objeto) fazendo referência a uma segunda thread (objeto)
  - exemplo: em thread **a** sobre thread **b**
  - `b.stop();`

#### □ **start()**

- inicia execução da thread principal (run) do objeto
- JVM chama o método `run()`

# Sincronização entre Threads

## **sincronização**

---

- **Controle de threads:** outros métodos nativos

- **stop()**

- força a própria thread a terminar sua execução

- **isAlive()**

- testa se a thread foi iniciada e ainda não terminou

- **suspend()**

- suspende a thread até ser reativada (*resume()*)

# Sincronização entre Threads

## **sincronização**

---

- **Controle de threads:** outros métodos nativos

- **resume()**

- reativa thread suspensa anteriormente

- **sleep(long ml)**

- a thread é colocada em espera (“dormindo”) por ml
      - milisegundos
    - a thread não abandona o monitor (synchronized)



# Sincronização entre Threads

## **sincronização**

---

- **Controle de threads:** outros métodos nativos
  - os métodos abaixo foram *deprecated*
    - *stop()*, *suspend()*, *resume()*
  - podem provocar erros de programação de difícil depuração;
  - por exemplo:
    - variáveis com estado inconsistente;

# Sincronização entre Threads

## **sincronização**

---

### ■ **Threads e o coletor de lixo (garbage collector)**

#### ☐ **Objeto Normal**

- coletado quando não houver nenhuma referência ao mesmo

#### ☐ **Objeto Thread**

- coletado quando:
  - ☐ não houver nenhuma referência ao mesmo
  - ☐ **E** execução da thread tiver terminado

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

---

### ■ Novos recursos na versão 5 do SDK da Sun

- <http://java.sun.com/javase/6/docs/api/java/util/concurrent/package-summary.html>
- pacote `java.util.concurrent`
- diversas novas classes, interfaces, exceções, ..., para programação concorrente
  
- **Alguns Recursos Novos:**
  - classe `semáforo`;
  - locks (mais flexíveis que `synchronized`);
  - barreiras;

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

---

- **JSR (Java Specification Requests) 166**
  - conjunto de utilitários de nível médio que fornecem funcionalidade necessária em programas concorrentes
  - proposto por Doug Lea e incorporado no JDK 5.0
  - **Principais Contribuições**
    - construtores de threads de alto nível, incluindo executores (thread task framework);
    - filas seguras de threads;
    - Timers;
    - locks (incluindo atômicos);
    - e outras primitivas de sincronização (como semáforos).
  - **Pacote `java.util.concurrent.*`**

# Sincronização entre Threads

## **Novos Recursos a partir do SDK 5 da SUN**

### ■ **Variáveis Atômicas:**

- manipulação atômica de variáveis (tipos primitivos ou objetos), fornecendo aritmética atômica de alto desempenho e métodos compare-and-set
- pacote `java.util.concurrent.atomic`

### ■ **Sincronizadores**

- **mecanismos de propósito geral para sincronização:**
  - semáforos – classe `Semaphores` (incluindo exclusão mútua);
  - barreiras – classe `CyclicBarrier`
  - travas – classe `CountDownLatch`
  - trocadores – classe `Exchanger<V>`.

# Sincronização entre Threads

## **Novos Recursos a partir do SDK 5 da SUN**

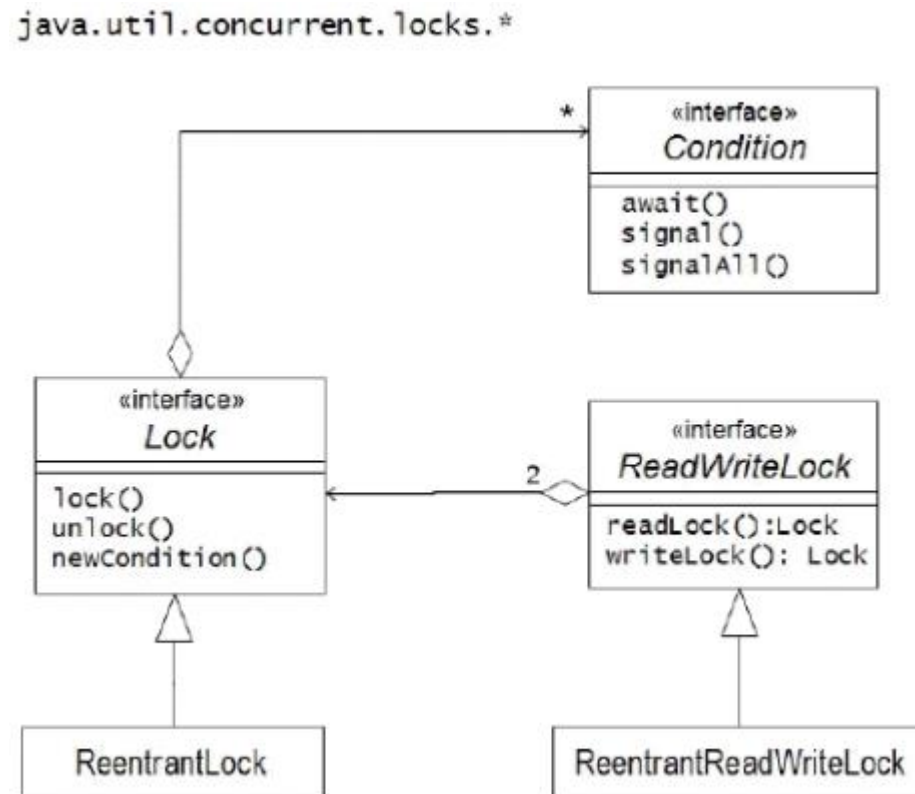
### ■ **Locks**

- resolve os inconvenientes da palavra-reservada `synchronized`;
- implementa lock de alto desempenho com a mesma semântica de memória do que sincronização;
- suportando timeout, múltiplas variáveis de condição por lock e interrompimento de threads que esperam por lock;
- Pacote `java.util.concurrent.locks`;

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

- **Locks:** Pacote `java.util.concurrent.locks`;



# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

### ■ Lock

- implementação de exclusão mútua para múltiplas threads
- substitui o uso de métodos e blocos Synchronized
  - Permite chain lock: adquiere A, depois B, libera A, adquiere C, ...
- exemplo de Uso de Lock

```
Lock l = new Lock();  
l.lock();  
try {  
    //acesso protegido pelo lock  
} finally {  
    l.unlock();  
}
```



# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

### ■ Locks (Cont.)

#### □ ReentrantLock

- pode ser justo: usar fila FIFO
- define os métodos `isLocked` e `getLockQueueLength`

```
Lock trava = new ReentrantLock();
if (trava.tryLock(30, TimeUnit.SECONDS)) {
    try {
        // acessar recurso protegido pela trava
    } finally {
        trava.unlock();
    }
} else {
    // tentar outra alternativa
}
```

**Trava condicional:** O método `tryLock()` “tenta” obter uma trava em vez de ficar esperando por uma

– Se a tentativa falhar, o fluxo do programa segue (não bloqueia o thread) e o método retorna `false`

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

### ■ Locks (Cont.)

#### □ ReadWriteLock

- pode permitir várias threads acessarem o mesmo objeto para leitura ou somente uma para escrita

```
class RWDictionary {  
    private final Map<String, Data> m = new TreeMap<String, Data>();  
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();  
  
    private final Lock r = rwl.readLock();  
    private final Lock w = rwl.writeLock();  
  
    public Data get(String key) {  
        r.lock();  
        try { return m.get(key); }  
        finally { r.unlock(); }  
    }  
    public Data put(String key, Data value) {  
        w.lock();  
        try { return m.put(key, value); }  
        finally { w.unlock(); }  
    }  
}
```

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

---

### ■ Locks (Cont.)

#### □ ReadWriteLock

- pode permitir várias threads acessarem o mesmo objeto para leitura ou somente uma para escrita

#### □ Condition

- adiciona variáveis condicionais
- uma thread suspende a execução até ser notificada por outra de que uma condição ocorreu
- `Condition cheio = lock.newCondition();`

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

---

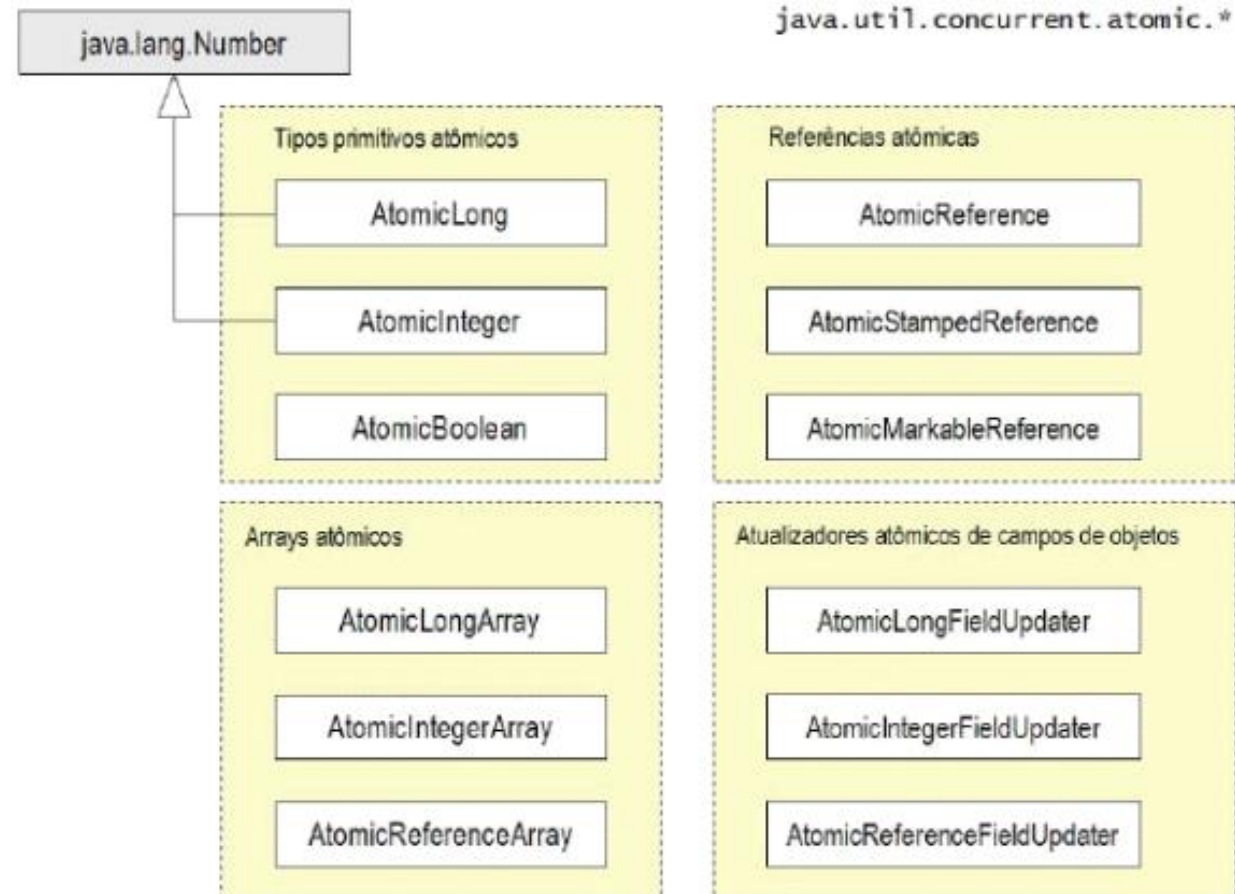
### ■ Variáveis Atômicas

- permite a utilização de uma única variável sem a necessidade de locks por múltiplas threads
- `boolean compareAndSet(expectedValue, updateValue);`
  - altera atômicamente uma variável para `updateValue` se ela atualmente tem `expectedValue`, retornando `true` se sucesso
- **principais Classes:**
  - `AtomicBoolean`
  - `AtomicInteger`
  - `AtomicLong`
  - `AtomicReference`

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

### ■ Variáveis Atômicas



# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

### ■ Variáveis Atômicas

Produtor e consumidor em que uma variável atômica é utilizada entre os dois processos

```
static class AtomicCounter
{
    private AtomicInteger contador = new AtomicInteger(0);
    public void increment() {
        System.out.println("Incrementando " + contador.incrementAndGet());
    }

    public void decrement() {
        System.out.println("Decrementando " + contador.decrementAndGet());
    }

    public int value() {
        return contador.get();
    }
}
```

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

### ■ Variáveis Atômicas

```
static class Produtor implements Runnable
{
    private AtomicCounter atomicCounter;

    public Produtor(AtomicCounter contador) {
        this.atomicCounter = contador;
    }
    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
            atomicCounter.increment();
        }
    }
}
```

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

### ■ Variáveis Atômicas

```
static class Consumidor implements Runnable
{
    private AtomicCounter atomicCounter;

    public Consumidor(AtomicCounter contador) {
        this.atomicCounter = contador;
    }
    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep(4000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
            atomicCounter.decrement();
        }
    }
}
```



# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

### ■ Variáveis Atômicas

```
public static void main(String[] args)
{
    AtomicCounter atomicCounter = new AtomicCounter();
    Thread consumidor = new Thread(new Consumidor(atomicCounter));
    Thread produtor = new Thread(new Produtor(atomicCounter));
    consumidor.start();
    produtor.start();
    while (true) {
        }
    }
}
```

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

---

### ■ Sincronizadores

- Implementam algoritmos populares de sincronização
- Facilita o uso da sincronização: evita a necessidade de usar mecanismos de baixo nível como métodos de Thread, wait() e notify()
- Estruturas disponíveis
  - Barreira cíclica: CyclicBarrier
  - Barreira de contagem regressiva: CountdownLatch - Trancas
  - Permutador: Exchanger
  - Semáforo contador: Semaphore

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

- Interface essencial

`java.util.concurrent.*`

Semaphore
<code>acquire()</code> <code>release()</code>

CountDownLatch
<code>await()</code> <code>countdown()</code>

CyclicBarrier
<code>await()</code>

Exchanger
<code>exchange(V):V</code>

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

### ■ Barreiras

- Uma barreira é um mecanismo de sincronização que determina um ponto na execução de uma aplicação onde **vários threads esperam os outros**
- Quando o thread chega no **ponto de barreira**, chama uma operação para indicar sua chegada e entra em estado inativo
- Depois que um **certo número** de threads atinge a barreira, ou certa contagem zero, ela é **vencida** e os threads acordam
- Opcionalmente, uma operação pode **sincronamente** executada na abertura da barreira antes dos threads acordarem
- Como implementar
  - Em baixo nível, usando o método **join()** para sincronizar com threads que terminam, ou usar **wait()** no ponto de encontro e **notify()** pelo último thread para vencer a barreira
  - Em java.util.concurrent: **CyclicBarrier** e **CountDownLatch**

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

### ■ Barreira Cíclica

#### □ Como criar

- `CyclicBarrier b = newCyclicBarrier (n)`
- `CyclicBarrier b = newCyclicBarrier (n, ação)`

#### ■ A barreira é criada especificando-se

- Número de *threads* *n* necessários para vencê-la
- Uma *ação (barrier action)* para ser executada sincronamente assim que a barreira for quebrada e antes dos *threads* retomarem o controle: implementada como um objeto Runnable

#### ■ Cada *thread* tem uma referência para uma instância *b* da barreira e chamar *b.await()* quando chegar no ponto de barreira

- O *thread* vai esperar até que todos os *threads* que a barreira está esperando chamem seu método *b.await()*

#### ■ Depois que os *threads* são liberados, a barreira pode ser reutilizada (por isto é chamada de cíclica)

- Para reutilizar, chame o método *b.reset()*

# Sincronização entre Threads


## Novos Recursos a partir do SDK 5 da SUN

### ■ Barreira Cíclica – Exemplo Parte 1

```
public class BarrierDemo {
    volatile boolean done = false;
    final Double[][] despesas = ...;
    volatile List<Double> parciais =
        Collections.synchronizedList(new ArrayList<Double>());

    public synchronized double somar(Double[] valores) { ... }

    class SomadorDeLinha implements Runnable {
        volatile Double[] dados; // dados de uma linha
        CyclicBarrier barreira;
        SomadorDeLinha(Double[] dados, CyclicBarrier barreira) {...}
        public void run() {
            while(!done) { // try-catch omitido
                double resultado = somar(dados);
                parciais.add(resultado); // guarda em lista
                System.out.printf("Parcial R$%.2f\n", resultado);
                barreira.await();
            }
        }
    }
    ...
}
```



Ponto de barreira

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

### ■ Barreira Cíclica – Exemplo Parte 2

...

```
class SomadorTotal implements Runnable {
    public void run() {
        Double[] array = parciais.toArray(new Double[5]);
        System.out.printf("Total R$%.2f\n", somar(array));
        done = true;
    }
}

public void executar() {
    ExecutorService es =
        Executors.newFixedThreadPool(despesas.length);
    CyclicBarrier barrier =
        new CyclicBarrier(despesas.length, new SomadorTotal());
    for(Double[] linha: despesas)
        es.execute(new SomadorDeLinha(linha, barrier));
    es.shutdown();
}
```

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

---

### ■ **CountDownLatch (Trancas)**

- similar à barreiras, a diferença é a condição para liberação:
  - não é o número de thread que estão esperando, mas sim quando um contador específico chega a zero
- threads que executarem o wait após o contador já ter atingido o zero são liberadas automaticamente
- o contador não é resetado automaticamente



# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

### ■ CountdownLatch (**Trancas**)

- Um **CountDownLatch** é um tipo de barreira
  - É inicializado com valor inicial para um contador regressivo
  - *Threads* chamam **await()** e esperam a contagem chegar a zero
  - Outros *threads* podem chamar **countDown()** para reduzir a contagem
  - Quando a contagem finalmente chegar a zero, a barreira é vencida (o trinco é aberto)
- Pode ser usado no lugar de CyclicBarrier
  - Quando liberação depender de outro(s) fator(es) que não seja(m) a simples contagem de *threads*

# Sincronização entre Threads

## Novos Recursos a partir do SDK 5 da SUN

### ■ **Exchanger (Trocadores)**

- servem para trocar dados entre *threads* de forma segura;
- o método *exchange* é chamado com o objeto de dado a ser trocado com outra *thread*
- se uma *thread* já estiver esperando, o método *exchange* retorna o objeto de dado da outra *thread*
- se nenhuma *thread* estiver esperando, o método *exchange* ficará esperando por uma

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

- Construindo um `java.util.concurrent.Semaphore`:
  - `new Semaphore(int permits)` ou
  - `new Semaphore(int permits, boolean fair)`
  - “Permits” define o valor inicial de semáforo
    - esse valor pode ser negativo:
      - nesse caso releases (V) devem ser feitos antes que uma thread possa receber uma permissão através de um acquire (P)

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

- **Construindo um `java.util.concurrent.Semaphore`:**
  - **“Fair” define se o semáforo é justo ou não:**
    - **true** - garante que as threads que invocaram `acquire(P)` recebem permissão na ordem de chamada do método `acquire` (FIFO)
    - **false** - nenhuma garantia sobre a ordem de recebimento de permissões é feita.
    - um valor `true` deve ser usado no caso de controle a recursos compartilhados, para impedir que ocorra starvation de uma thread
    - em outros casos um valor de `false` pode ser mais desejável, devido ao ganho de desempenho que ocorre neste modo.

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

- Método **acquire()**
  - o equivalente do P conforme conceitos de semáforos se o número de "permissões" de um semáforo for maior que 0
    - o contador é diminuído e o método retorna imediatamente
  - se não existir nenhuma permissão disponível no semáforo a thread fica bloqueada até que:
    - outra thread chame `release()` neste semáforo e a thread atual seja a próxima na lista para receber a permissão.
    - outra thread interrompa a thread atual

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

### ■ Versões do método acquire:

- **void acquire()**

- **void acquire(int permits)**

- ao invés de adquirir apenas uma permissão adquiere o número de permissões do parâmetro permits
- a thread é bloqueada até conseguir todas as permissões solicitadas

- **void acquireUninterruptibly()**

- similar ao método acima, mas neste método a thread não pode ser interrompida;

- **void acquireUninterruptibly(int permits)**

- similar ao método acima, mas neste método a thread não pode ser interrompida;

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

- Métodos tryAcquire
  - os métodos tryAcquire permitem
    - que uma thread tente adquirir uma permissão e continue a execução se não conseguir depois de um certo tempo;
    - todos eles retornam um valor booleano indicando se foi possível ou não adquirir a permissão.

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

### ■ versões:

- `boolean tryAcquire()`
  - adquire uma permissão do semáforo, se uma estiver disponível
- `boolean tryAcquire(int permits)`
  - adquire o número de permissões passado, se todas estiverem disponíveis
- `boolean tryAcquire(int permits, long timeout, TimeUnit unit)`
  - adquire o número de permissões dado, se todas ficarem disponíveis dentro do tempo passado
- `boolean tryAcquire(long timeout, TimeUnit unit)`
  - adquire uma permissão, se uma ficar disponível dentro do tempo passado



# Sincronização entre Threads

## Semáforos J2SE 5.0

---

### ■ Método `release()`

- o equivalente de V conforme conceitos de semáforos
- o método `release` libera uma permissão do semáforo (aumenta o contador em um)
  - se alguma thread estiver tentando realizar um `acquire`, então uma destas threads é selecionada e recebe a permissão
- uma thread não precisa ter realizado um `acquire()` antes de realizar um `release()`

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

- **Método release()**

- **versões de release() :**

- **void release()**

- libera uma permissão

- **void release(int permits)**

- libera o número de permissões passado como parâmetro

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

### ■ Métodos de verificação de estado do semáforo

#### □ `int availablePermits()`

- retorna o número de permissões disponíveis no semáforo

#### □ `protected Collection<Thread> getQueuedThreads()`

- retorna uma coleção com as Threads que podem estar tentando adquirir uma permissão neste semáforo

#### □ `int getQueueLength()`

- retorna uma estimativa do número de threads tentando adquirir uma permissão neste semáforo

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

### ■ Métodos de verificação de estado do semáforo

#### ☐ **boolean hasQueuedThreads()**

- verifica se alguma thread está tentando adquirir uma permissão neste semáforo.

#### ☐ **boolean isFair()**

- retorna true se o semáforo for justo.

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

### ■ Métodos auxiliares

#### □ **int drainPermits()**

- adquire todas as permissões disponíveis imediatamente e retorna quantas permissões foram adquiridas;

#### □ **protected void reducePermits(int reduction)**

- reduz o número de permissões disponíveis pelo número indicado pelo parâmetro reduction, que deve ser um número positivo

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

- **Exemplo 1: Um objeto que controla o acesso a uma seção crítica**

```
class Exclusao {  
    private final Semaphore livre =  
        new Semaphore(1, true);  
  
    public Object acessaSecaoCritica()  
        throws InterruptedException {  
        livre.acquire();  
        //faz acesso a seção crítica;  
        livre.release();    }  
}
```

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

### ■ Exemplo 2: Produtores Consumidores (Sun)

```
class Pool {  
    private static final MAX_AVAILABLE = 100;  
    protected Object[] items = new Object[10];  
    protected boolean[] used = new  
        boolean[MAX_AVAILABLE];  
    private final Semaphore available =  
        new Semaphore(MAX_AVAILABLE, true);  
  
    public Object getItem() throws InterruptedException {  
        available.acquire();  
        return getNextAvailableItem();  
    }  
  
    public void putItem(Object x) {  
        if (markAsUnused(x))  
            available.release();  
    }  
}
```

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

### ■ Exemplo 2: Produtores Consumidores (Sun) cont.

```
protected synchronized Object getNextAvailableItem() {  
    for (int i = 0; i < MAX_AVAILABLE; ++i) {  
        if (!used[i]) {  
            used[i] = true;  
            return items[i];  
        }  
    }  
    return null; // not reached  
}
```



# Sincronização entre Threads

## Semáforos J2SE 5.0

---

### ■ Exemplo 2: Produtores Consumidores (Sun) cont.

```
protected synchronized boolean markAsUnused(Object
item) {
    for (int i = 0; i < MAX_AVAILABLE; ++i) {
        if (item == items[i]) {
            if (used[i]) {
                used[i] = false;
                return true;
            } else
                return false;
        }
    }
    return false;
}
```

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

### ■ Exemplo 2: Produtores Consumidores (Sun)

#### □ o mais interessante de se notar neste exemplo

- é que a thread não está em um método sincronizado quando ela chama acquire e release
- isto é importante para permitir que várias threads façam acquire
- e que enquanto uma thread está trancada no acquire outra ainda assim possa colocar um item e realizar um release

#### □ o semáforo controla a sincronização necessária para restringir o acesso à fila

- independentemente da sincronização necessária para manter a consistência da fila

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

### ■ Comparação entre Semaphore e a definição clássica de semáforos

#### □ Semaphore

- é basicamente uma versão orientada a objetos da definição clássica de semáforos
- onde P e V foram substituídos pelos métodos do próprio objeto semáforo `acquire()` e `release()`
- entretanto, possui alguns métodos que não têm equivalente nos semáforos clássicos
  - o principal sendo `tryAcquire()`

# Sincronização entre Threads

## Semáforos J2SE 5.0

---

- **Comparação entre Semaphore e a definição clássica de semáforos**

- **nos Semaphores**

- o contador também não pode ser simplesmente colocado em algum valor arbitrário depois da criação;
    - mas é possível manipular este contador através de `reducePermits()` e de `release()`

# Gerenciamento de Processos

## Parte 5

---

- **Referências Utilizadas:**
  - Livro do Tanenbaum
    - Sistemas Operacionais Modernos
    - [www.cs.vu.nl/~ast](http://www.cs.vu.nl/~ast)
  - Livro do Silberschatz
    - Operating System Concepts
    - [www.bell-labs.com/topic/books/aos-book/](http://www.bell-labs.com/topic/books/aos-book/)
  - Livro do Machado e Maia
    - Arquitetura de Sistemas Operacionais.