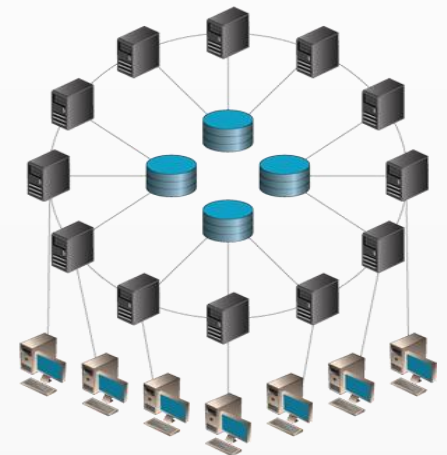


Processos e Threads

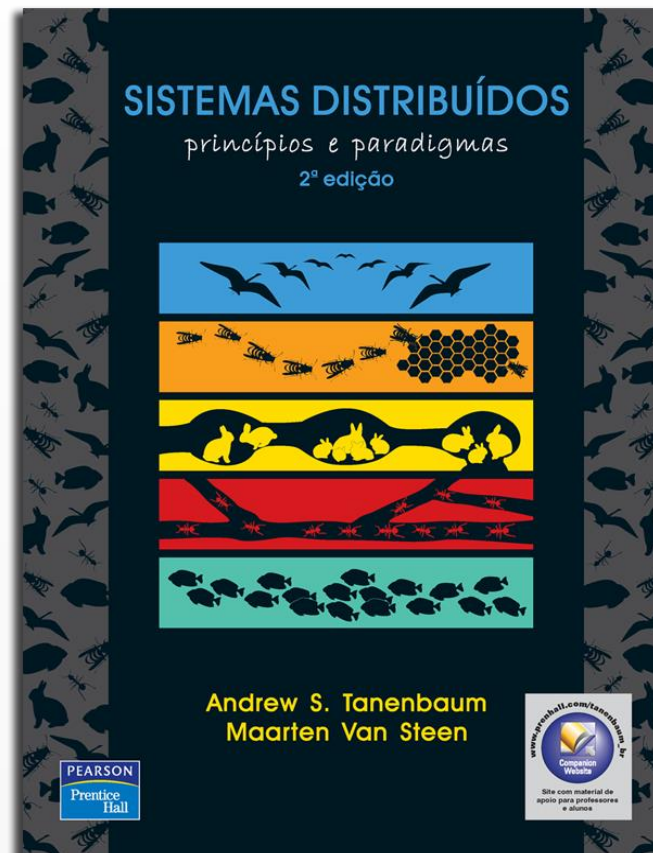
# SMD0050 - SISTEMAS DISTRIBUÍDOS

1



Slides são baseados nos slides do Coulouris e Tanenbaum

# Processos



capítulo

3



# Sistema Operacional



- Um dos principais aspectos de sistemas distribuídos é o compartilhamento de recursos
- Os aplicativos clientes invocam operações em recursos que frequentemente estão em outro nó, ou pelo menos em outro processo
- Aplicativos e serviços usam a camada de middleware para suas interações



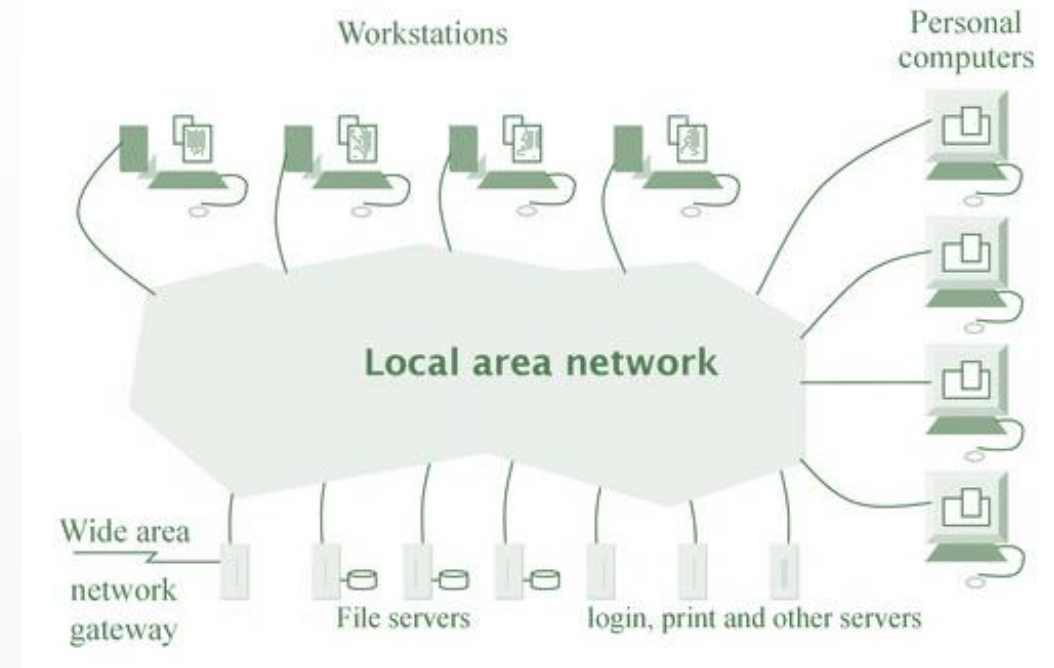
# Sistema Operacional



- Abaixo da camada de middleware está a camada do sistema operacional (SO)
- A tarefa de qualquer sistema operacional é fornecer abstrações dos recursos físicos subjacentes
- Ex.: processadores, memória, comunicação e mídias de armazenamento, rede

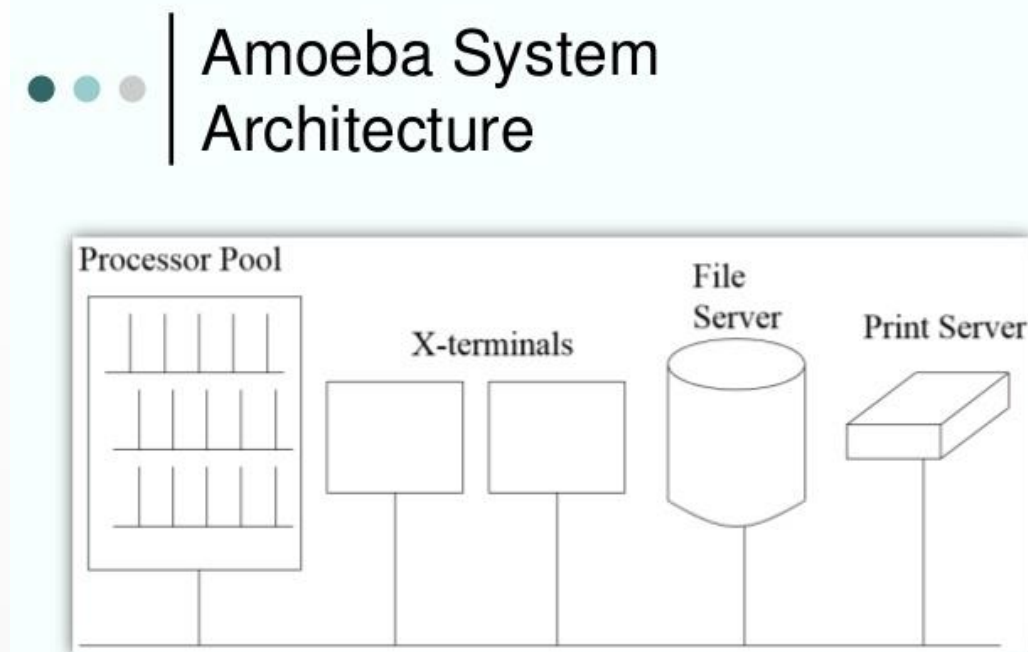
# Sistema Operacional Distribuído?

- Um sistema operacional de rede não escala os processos nos vários nós
- Um sistema operacional com uma única imagem do sistema com controle sobre todos os nós seria um sistema operacional distribuído



# Sistema Operacional Distribuído?

- Funcionar como uma máquina única
  - MicroKernel e MultiThread
  - Comunicação via RPC
- Os nós funcionam como terminais e um deles executa um pool de processos para decidir onde ocorrerá a execução



Amoeba foi desenvolvido pelo Andrew S. Tanenbaum  
Python foi desenvolvido inicialmente para esse S O

# Razões para não termos SO distribuídos

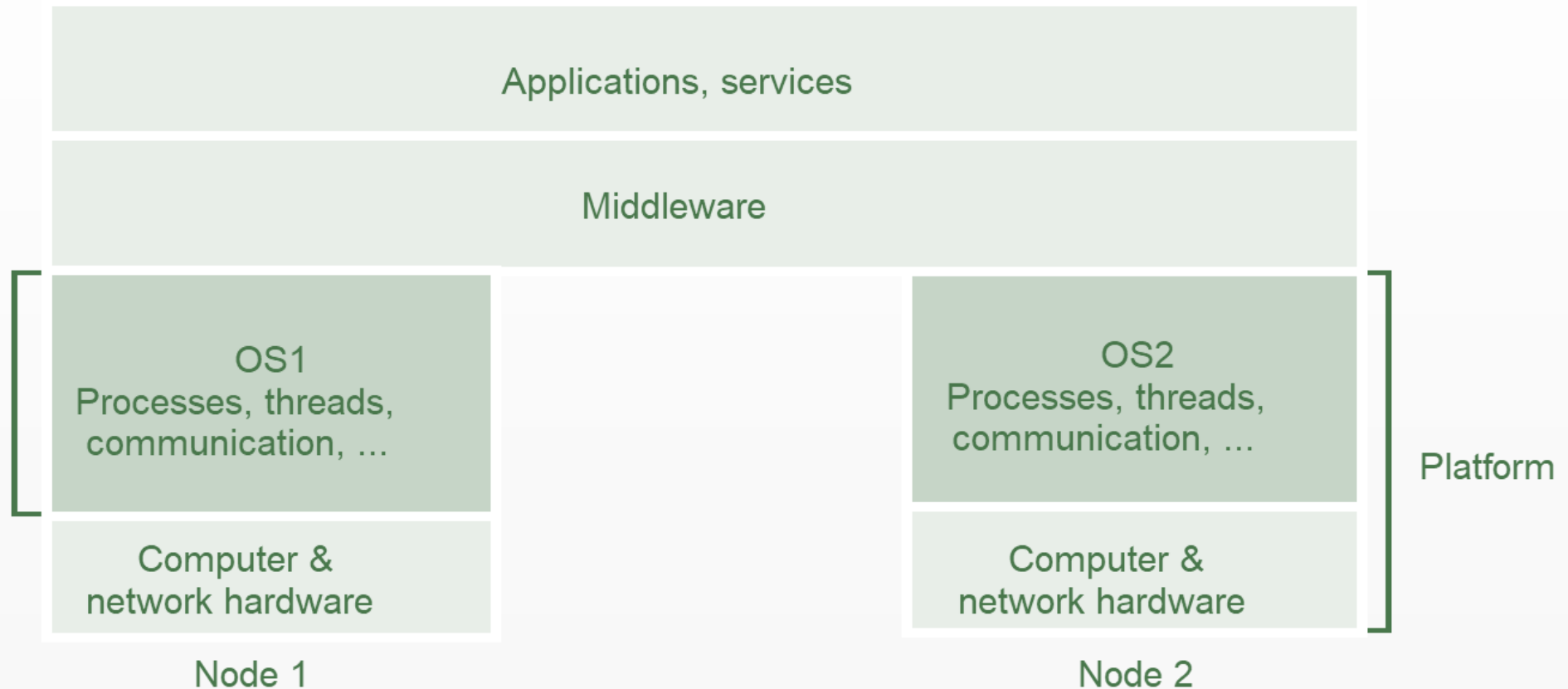
- Exigência de homogeneidade
  - Investimentos feitos em softwares aplicativos específicos
- Desempenho ruim
  - Afetado pela rede



A combinação de middleware e SO de rede proporciona um equilíbrio aceitável entre os requisitos de autonomia e o acesso aos recursos

# Sistema Operacional e Middleware

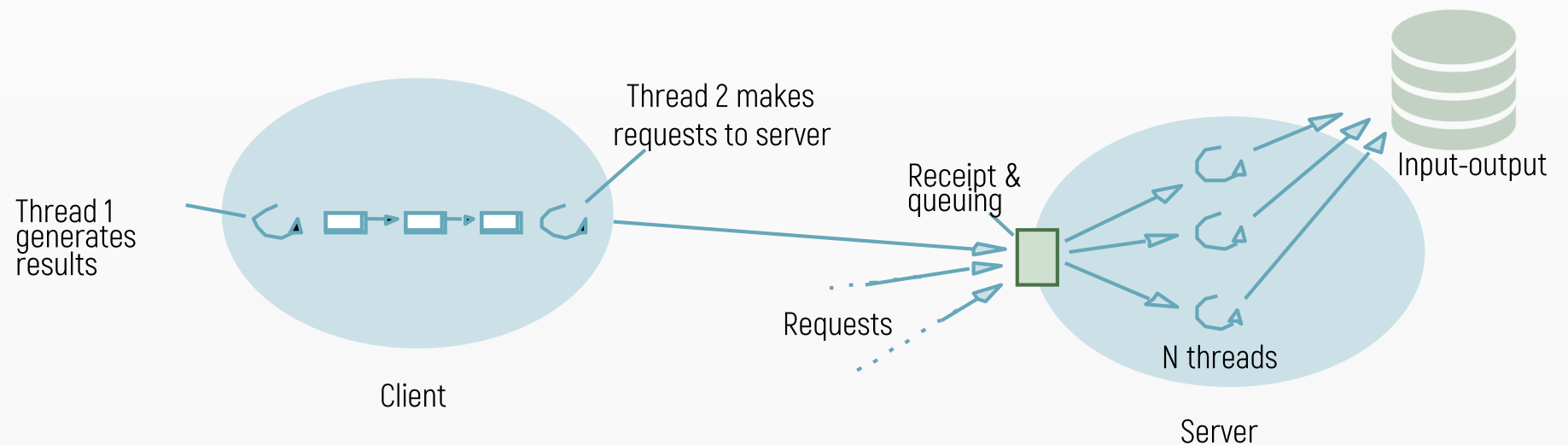
OS: kernel,  
libraries &  
servers





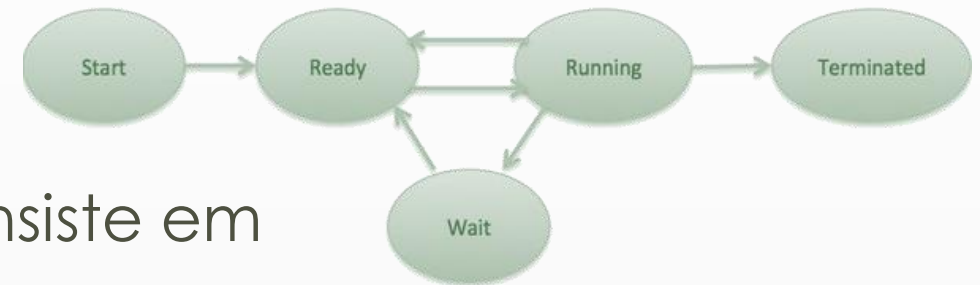
# Processos

- Diferentes tipos de processos desempenham papel crucial em sistemas distribuídos
  - Programas em execução em cada SO
  - ID, Ciclo de vida e espaço de memória
- Usar processos multithreading auxiliam em melhoria de desempenho em sistemas cliente-servidor



# Sistema operacional

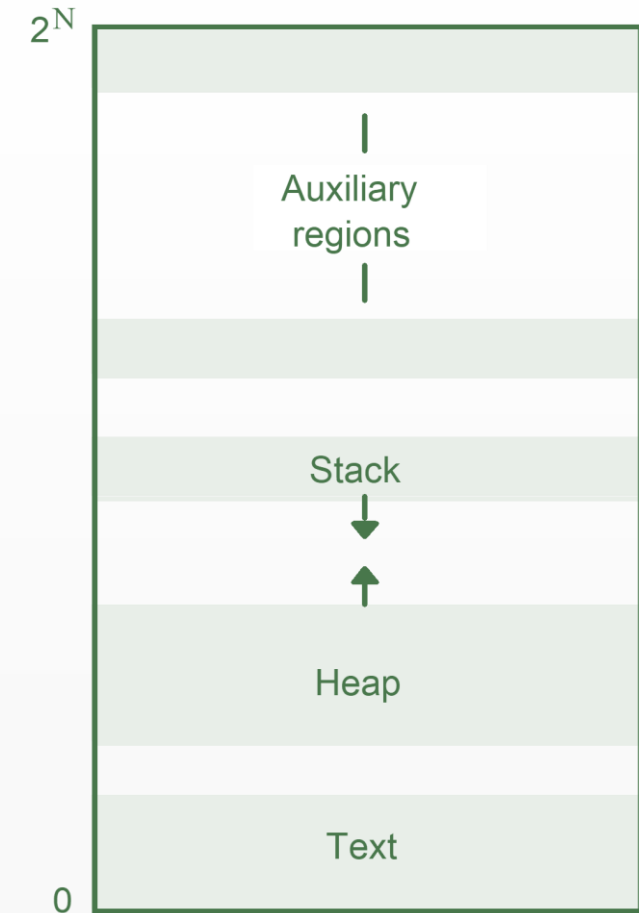
- O ambiente de execução é a unidade de gerenciamento de recursos
  - Um conjunto de recursos locais gerenciados pelo núcleo
  - Ciclo de vida



- Um ambiente de execução consiste em
  - um espaço de endereçamento
  - recursos de sincronização e comunicação entre threads, como semáforos e interfaces de comunicação
  - recursos de nível mais alto, como arquivos

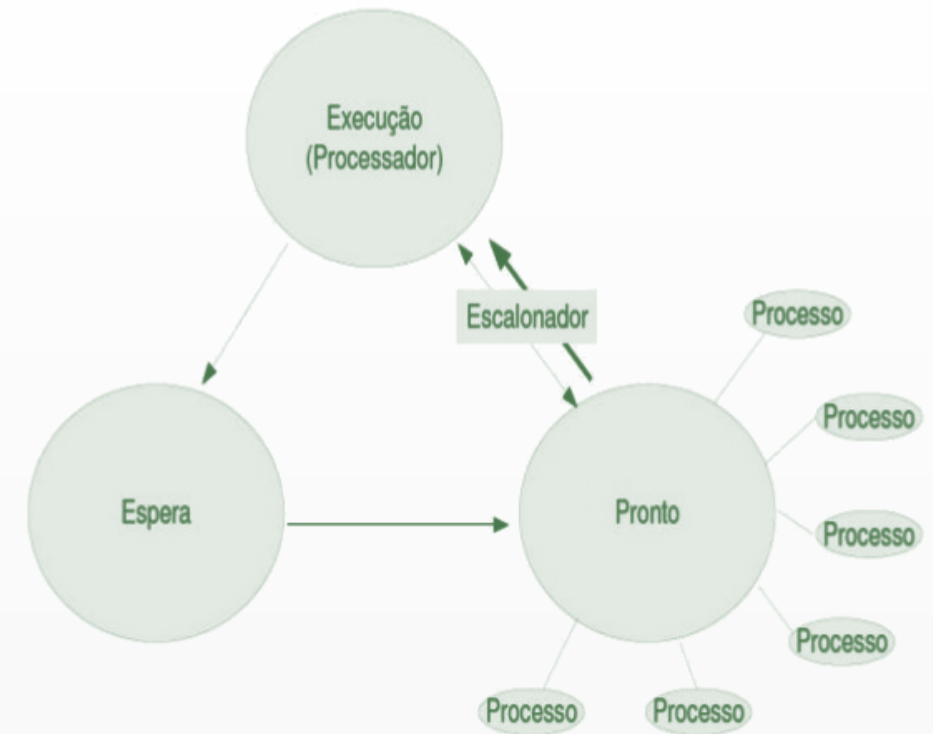
# Espaço de endereçamento

- Um espaço de endereçamento é a unidade de gerenciamento da memória virtual de um processo
  - Extensão da memória
  - Permissões de leitura/gravação e execução
  - Pilha de execução
  - Valores dos registradores



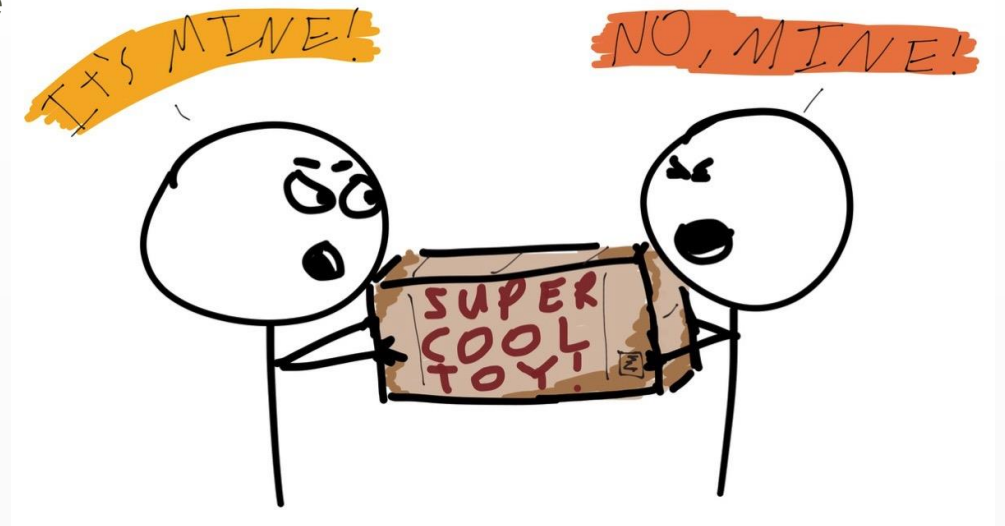
# Processos

- O S.O. garante transparência sobre uso de CPU e dispositivos de I/O concorrentemente. Para isso é usado o escalonamento de processos
  - Uso demasiado de chaveamento no modo dual (usuário/monitor);
  - Uso demasiado de chaveamento entre processos;
  - Modificações constantes no MMU (Memory Management Unit)



# Regiões Compartilhadas

- A necessidade de compartilhar memória entre processos, ou entre processos e o núcleo, é um dos fatores que leva ao uso de regiões compartilhadas
  - Bibliotecas
  - Núcleo
  - Compartilhamento de dados e comunicação
  - Hardware



# Tarefa

- Quais os problemas de compartilhar recursos entre processos?
- Como a estratégia de escalonamento pode causar os seguintes problemas?
  - Deadlock
  - Inanição (Starvation)

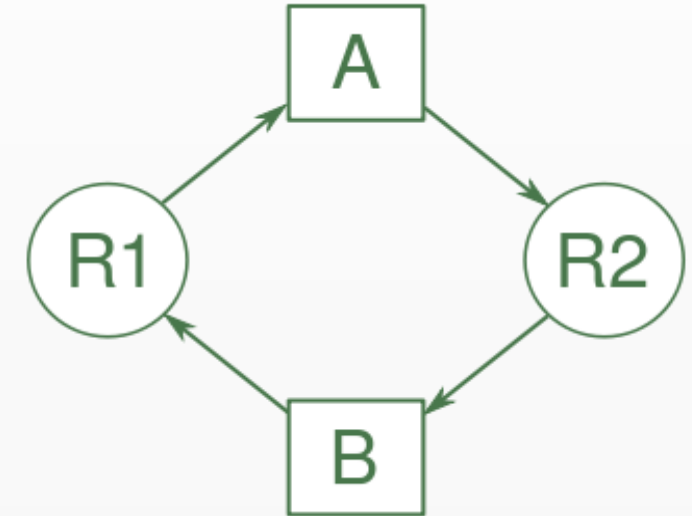


# Problemas da concorrência de recursos

- Processos podem compartilhar variáveis e recursos cujo uso concorrente pode levar a um estado de inconsistência
  - Dois processos enviados bytes via serial
  - Dois processos alterando um mesmo arquivo
  - Dois processos acessando a câmera

# Deadlock

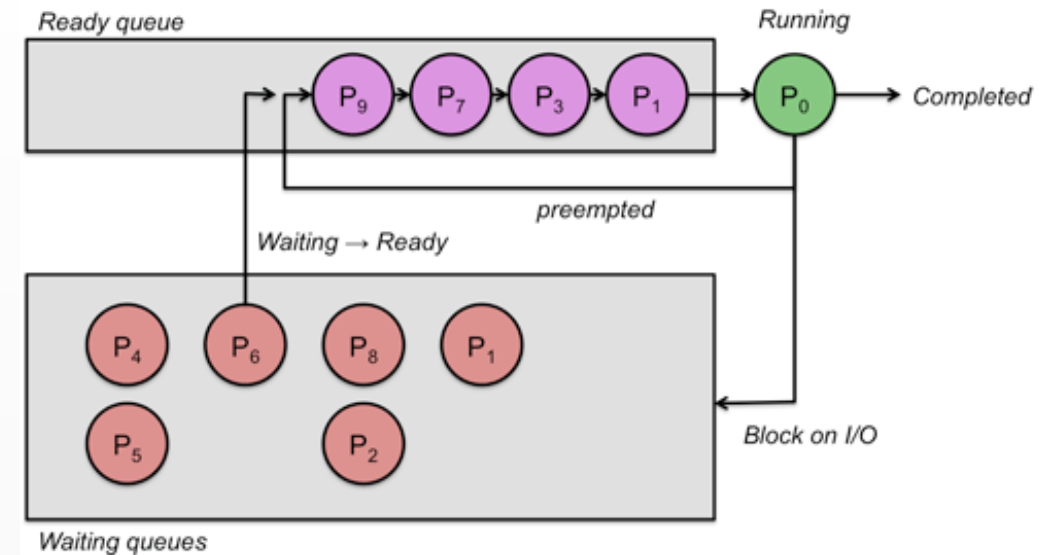
- Efeito colateral do uso de reserva de recursos que pode afetar a execução de processos
  - A pede recurso r1 => bloqueia recurso
  - B pede recurso r2 => bloqueia recurso
  - A pede recurso r2 => espera
  - B pede recurso r1 => espera





# Starvation-Inanição

- Escalonamento
  - Não-preemptivo, FIFO
  - Uso de fila de prioridades
- Efeito colateral de um algoritmo de escalonamento que pode causar a não execução de um processo ou thread
- Soluções
  - Round-robin
    - Aumento da prioridade com tempo
    - Process-Aging
  - Completely Fair Scheduler



# Região Crítica

- A região crítica é uma área de código de um algoritmo que acessa um recurso compartilhado
  - Não pode ser acessado concorrentemente por mais de uma linha de execução
- Uso de técnicas para controlar a concorrência
  - Semáforos, interrupções, bloqueio/reserva de recursos





# Threads: Introdução a Concorrência em Java



# Definindo um Programa Concorrente

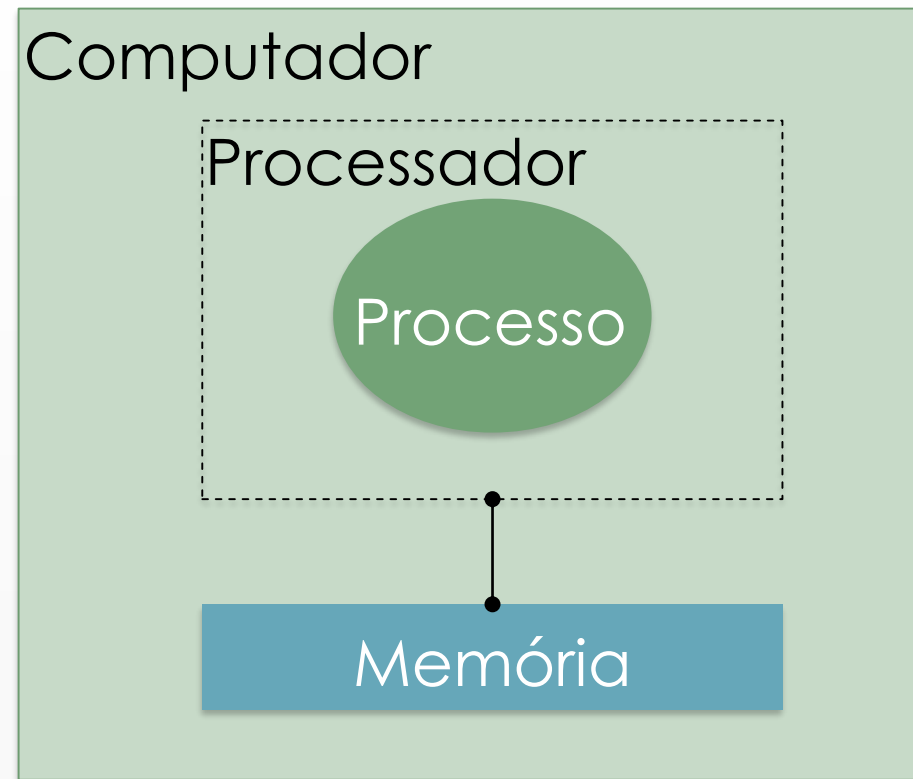
"A concurrent program has multiple threads of control allowing it perform multiple computations in parallel and to control multiple external activities which occur at the same time." (Magee and Kramer, 2000)

Referência: Jeff Magee and Jeff Kramer. 2000. Concurrency: State Models & Java Programs. John Wiley & Sons, Inc., New York, NY, USA.

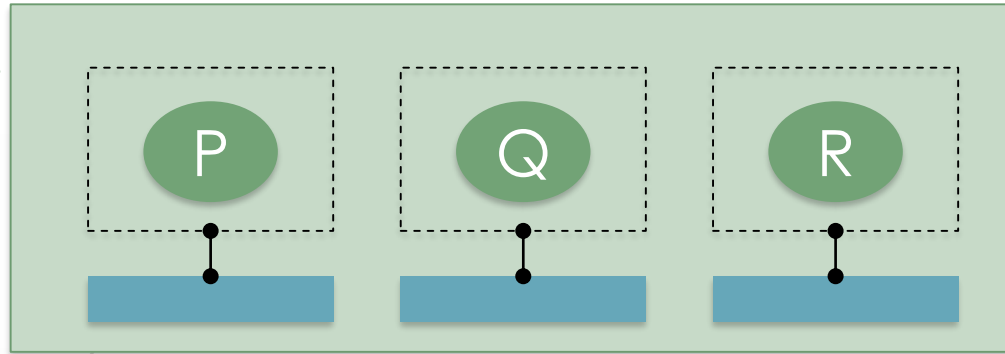


## Paralelismo, Concorrência e Distribuição

# Notação

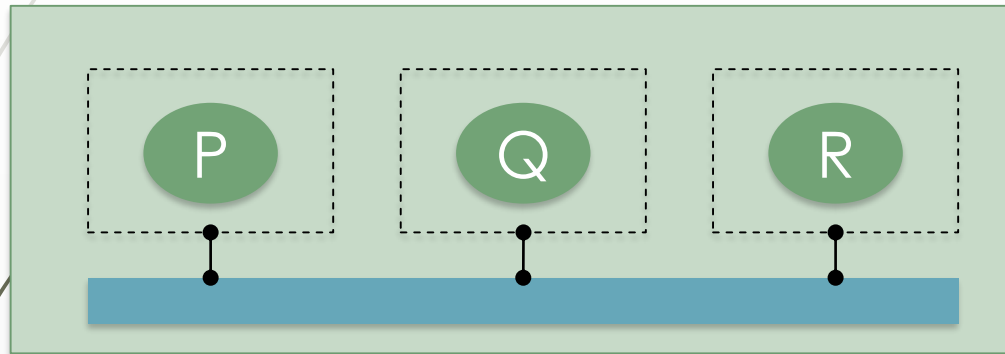


# Processos Paralelos



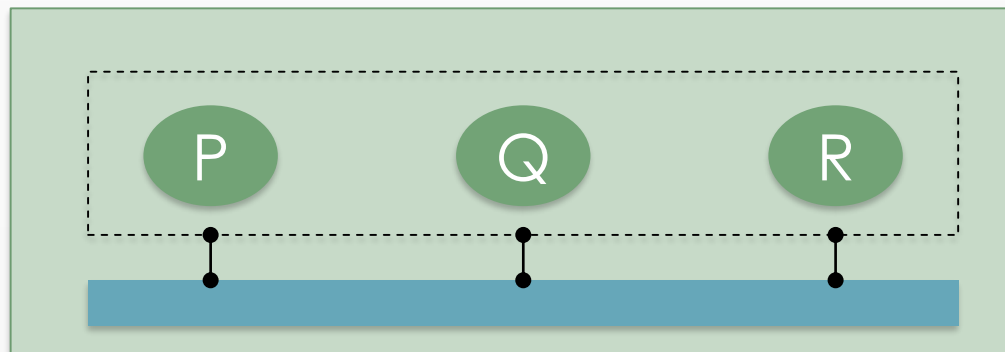
(1)

Processos executando em processadores distintos e acessando espaço de memória distintos



(2)

Processos executando em processadores distintos e acessando espaço de memória compartilhado

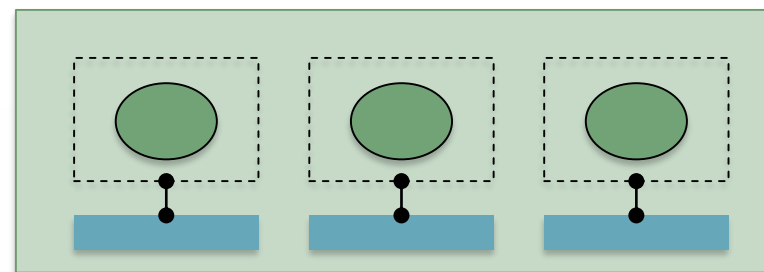
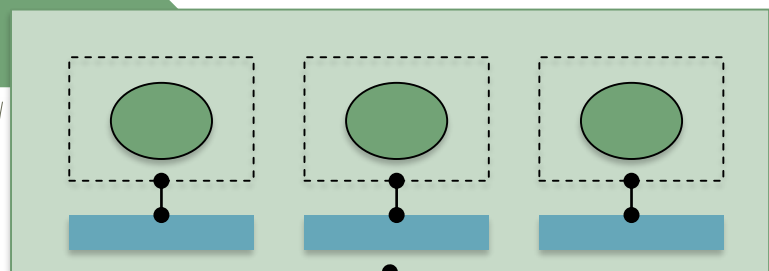


(3)

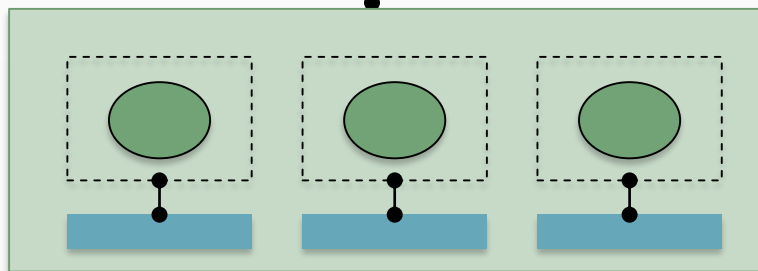
Processos executando no mesmo processador e acessando espaço de memória compartilhado

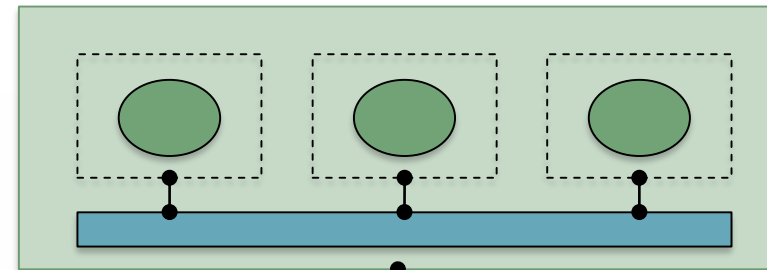
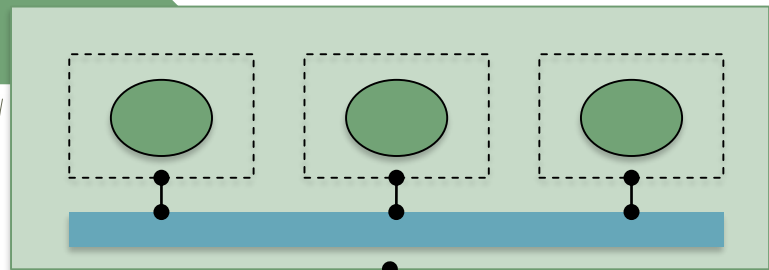


# Processos Distribuídos

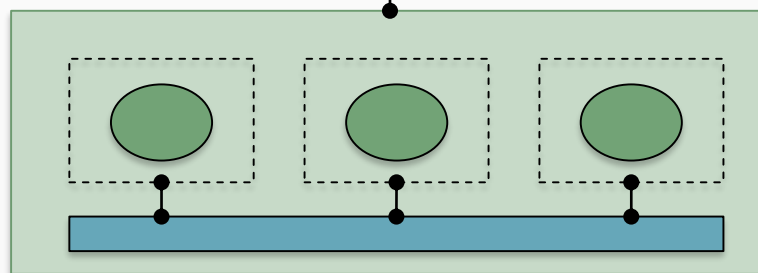


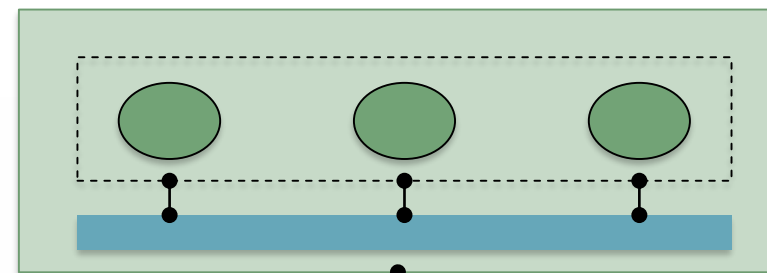
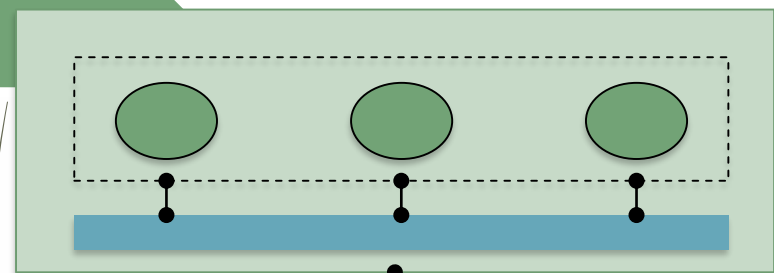
Elemento de Conexão



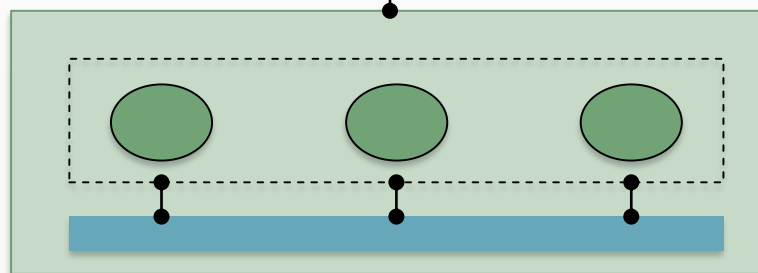


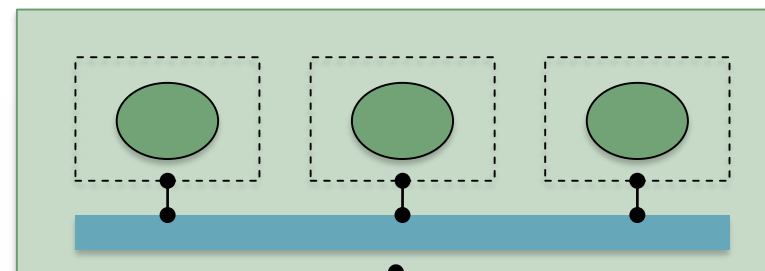
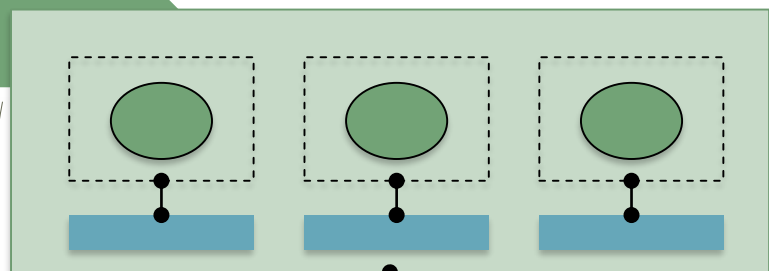
Elemento de Conexão



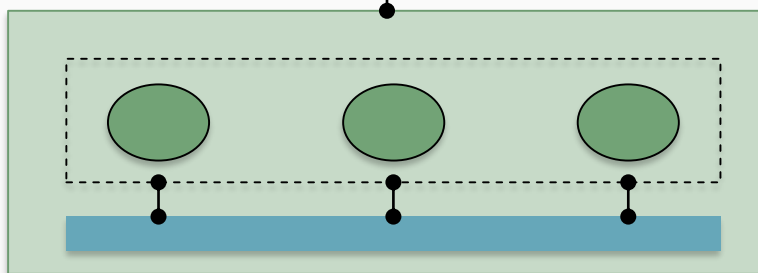


Elemento de Conexão





Elemento de Conexão



# Tipos de Concorrência em Programas

**“Competitive concurrency** exists when two or more active components are designed separately, are not aware of each other, but use the same passive components. The former have to compete for the latter and keep them at their disposal until there is no more need in them. Normally, components compete for a resource which knows nothing about the components that can use it.

Fonte: Alexander Romanovsky. On Structuring Cooperative and Competitive Concurrent Systems. The Computer Journal (1999) 42 (8): 627-637. Link: <http://comjnl.oxfordjournals.org/content/42/8/627.short>


# Tipos de Concorrência em Programas

"Cooperative concurrency exists when several components cooperate, i.e. do some job together and are aware of this. They can communicate by resource sharing or explicitly, but the important thing is that they have been designed together so that they would cooperate to achieve their joint goal and use each other's help and results. They synchronise their execution and can wait for information computed by another cooperating component. This is a joint activity of several concurrent components with equal rights which are aware of each other." (Romanovsky, 1999)

Fonte: Alexander Romanovsky. On Structuring Cooperative and Competitive Concurrent Systems. The Computer Journal (1999) 42 (8): 627-637. Link: <http://comjnl.oxfordjournals.org/content/42/8/627.short>



# Threads - Uso em sistemas não distribuídos

- Distribuição de várias tarefas concorrentes sem que o processo inteiro seja bloqueado em espera a determinada resposta
    - Em sistemas multi-core, cada thread pode ser executada ao mesmo tempo em processadores distintos
  - Cooperação entre programas através do uso de IPC (InterProcess Communication)
    - Comunicação requer chaveamento de contexto em 3 pontos diferentes
    - Threads não
- 





# Implementação de Thread - SO

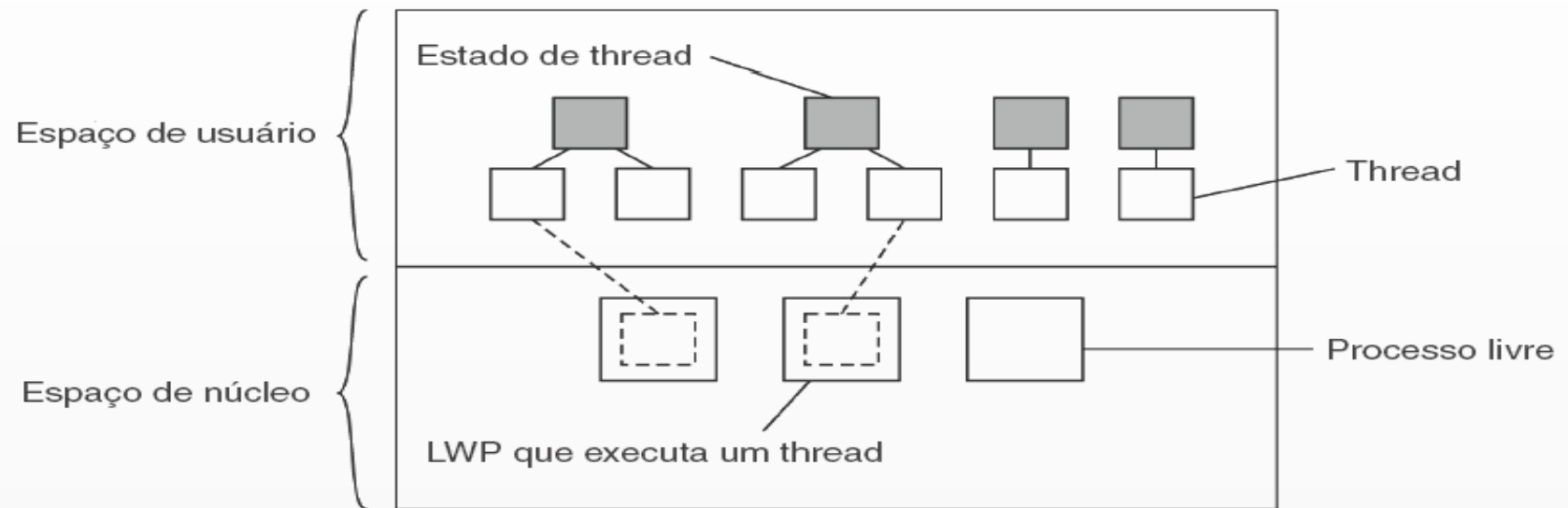
- Construir uma biblioteca de threads que é executada inteiramente em modo usuário
  - Criar e terminar threads é barato
  - Escalonamento é feito internamente
  - Uma chamada bloqueadora bloqueia todo o processo.
- Fazer com que o núcleo fique ciente dos threads e os escalone
  - Criar e terminar threads tem alto custo
  - Escalonamento feito pelo S.O.



# Implementação de Thread LWPs

- Abordagem híbrida: LWP (Lightweight Process)
  - Executa em um único contexto (pesado)
  - Vários LWPs por processo
- Todas operações em threads são realizadas sem intervenção do núcleo
  - Sincronização entre LWPs não requer intervenção do núcleo
  - Uma chamada bloqueante bloqueia um LWP, mas não os outros LWPs, que compartilham a tabela de threads entre si

# Uso de LWPs para implementação de threads



**Figura 3.2** Combinação de processos leves de nível de núcleo e threads de nível de usuário.



# Threads em Java

- Podem ser entendidas como *lightweight processes*
- Todo processo Java possui, pelo menos, uma *thread* em execução
- As *threads* compartilham os recursos (memória e arquivos abertos) do processo
- Aumenta a eficiência, mas impõem cuidados durante o desenvolvimento



# Definindo e Iniciando uma Thread

- Em Java existem duas formas de implementar uma thread
  - Estender da classe Thread
  - Implementar a interface Runnable
- Se o primeiro caso é utilizado, basta chamar o método start da thread para iniciar a execução
- No segundo caso, é preciso instanciar a classe que implementa Runnable, passa-la como argumento do construtor de uma instância da classe Thread e, em seguida, chamar o método start desta última

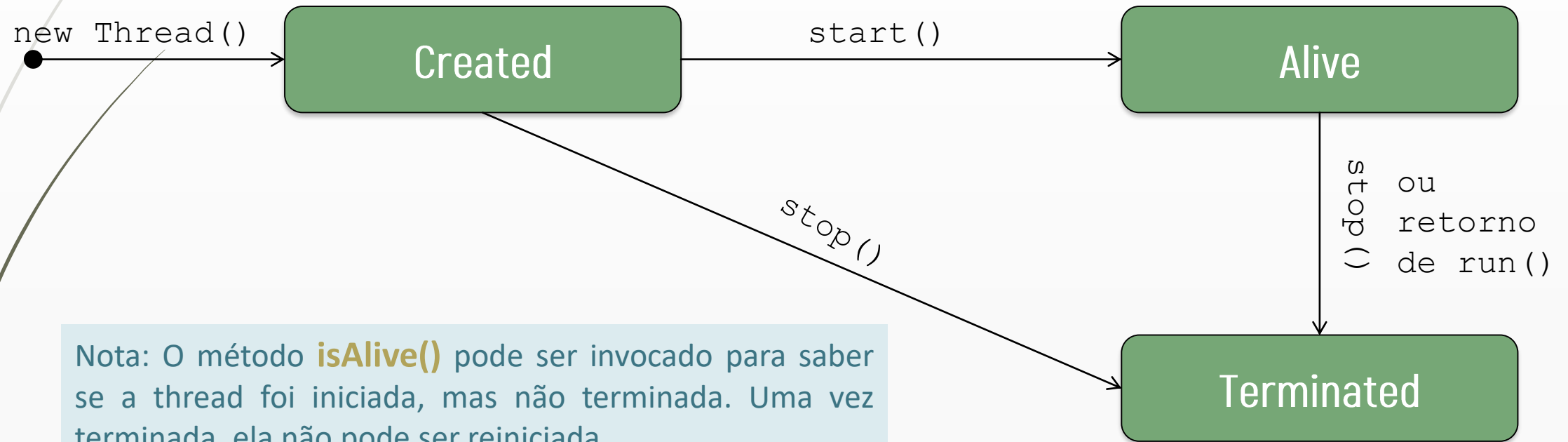
# Definindo e Iniciando uma Thread

```
public class HelloRunnable implements Runnable {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

```
public class HelloThread extends Thread {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
}
```

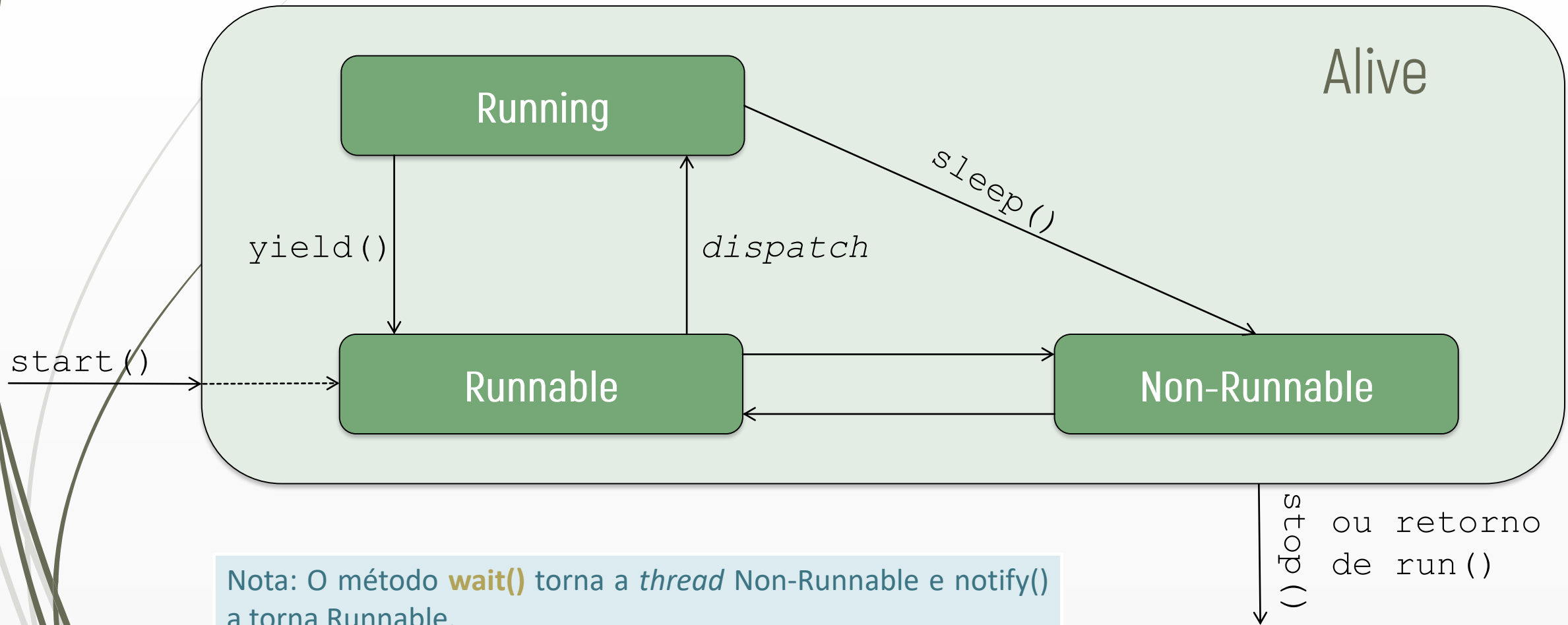
# Ciclo de Vida de uma Thread

`start()` faz com que a *thread* invoque seu próprio método `run()`



Nota: O método **`isAlive()`** pode ser invocado para saber se a thread foi iniciada, mas não terminada. Uma vez terminada, ela não pode ser reiniciada.

# Ciclo de Vida de uma Thread



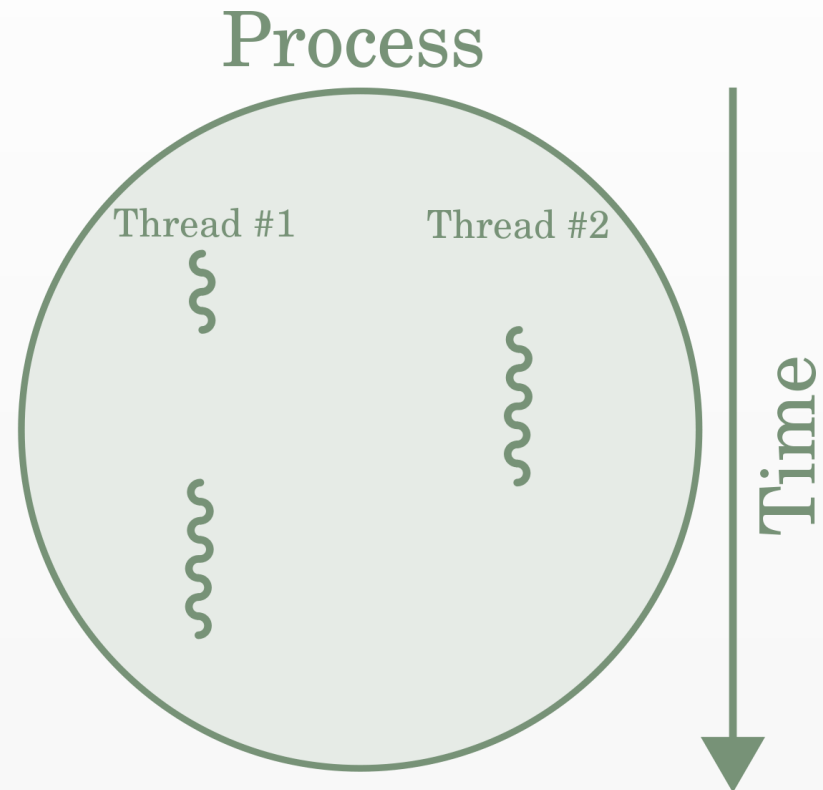


# Principais Métodos

Método	Descrição
run()	Encapsula o código que irá executar de fato e precisa estar presente em toda thread
start()	Registra a thread no thread scheduler
yield()	Faz com que a thread corrente pause, possibilitando que outra thread seja despachada
sleep()	Faz com que a thread fique em estado de espera uma quantidade mínima de tempo, em milissegundos
stop()	Finaliza a thread (depreciado no Java)
interrupt()	Atribui à thread o estado de interrompível

# Interleaving de Threads

- A ordem do processamento das instruções de duas ou mais threads que executam em paralelo não é determinístico!



# Tarefa para entregar 1

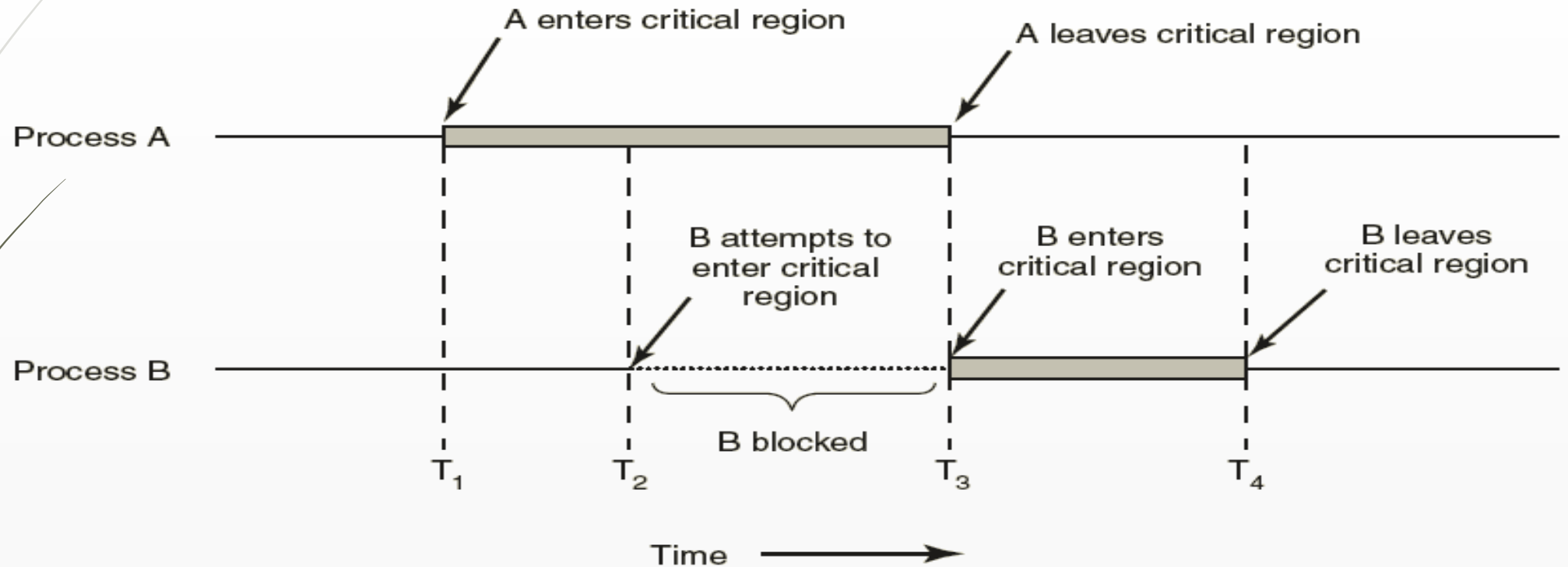
- 1) Crie uma classe `Racer` que possui um `"while (true)"` e imprime a frase `"Racer i – imprimindo"` onde `i` deve ser um parâmetro do seu construtor. Transforme esta classe em uma `Thread` usando uma das formas de criação e instaciação .
- 2) Crie uma classe `Race` que cria e executa 10 racers (identificadores de 1 a 10).
  - A) Como se deu o comportamento dos prints?
  - B) Adiciona um tempo de espera (usando o método `sleep`) nos `Racers`, o que houve com comportamento do sistema?
  - C) Utilize o método `setPriority` para definir as condições de corrida. Houve mudanças na execução? Se sim, descreva-as.
- 3) Modifique a classe `Racer` para que ela imprima apenas 1000 vezes. Em seguida, modifique a classe `Race` para que os carros pares só iniciem suas corridas quando os ímpares terminarem. Use o método `join` para tal tarefa



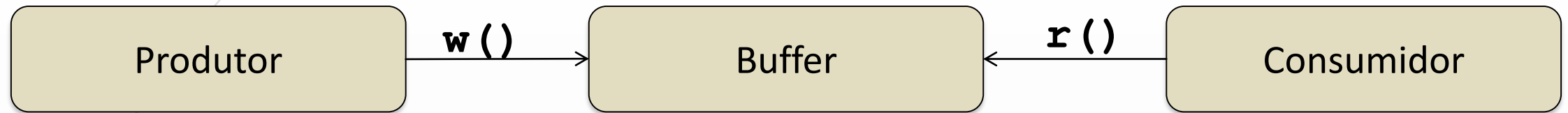
# Condições de Corrida

- São situações onde duas ou mais threads estão acessando recursos compartilhados, e o resultado final do processamento depende de quem executa e quando executa
- A região do código fonte do programa onde o acesso a recursos compartilhados é feito é denominada de Região Crítica
- Como evitar?
  - Exclusão Mútua
  - Em Java isso é feito usando sincronização de processos (synchronized)

# Exclusão Mútua



# O Problema do Produtor/Consumidor



O problema consiste em garantir que o consumidor só leia quando houver dados para ler e que o produtor só escreva quando o buffer tiver espaço livre, i.e., depois que o consumidor tenha lido.

## Tarefa II - Produtor/Consumidor em Java

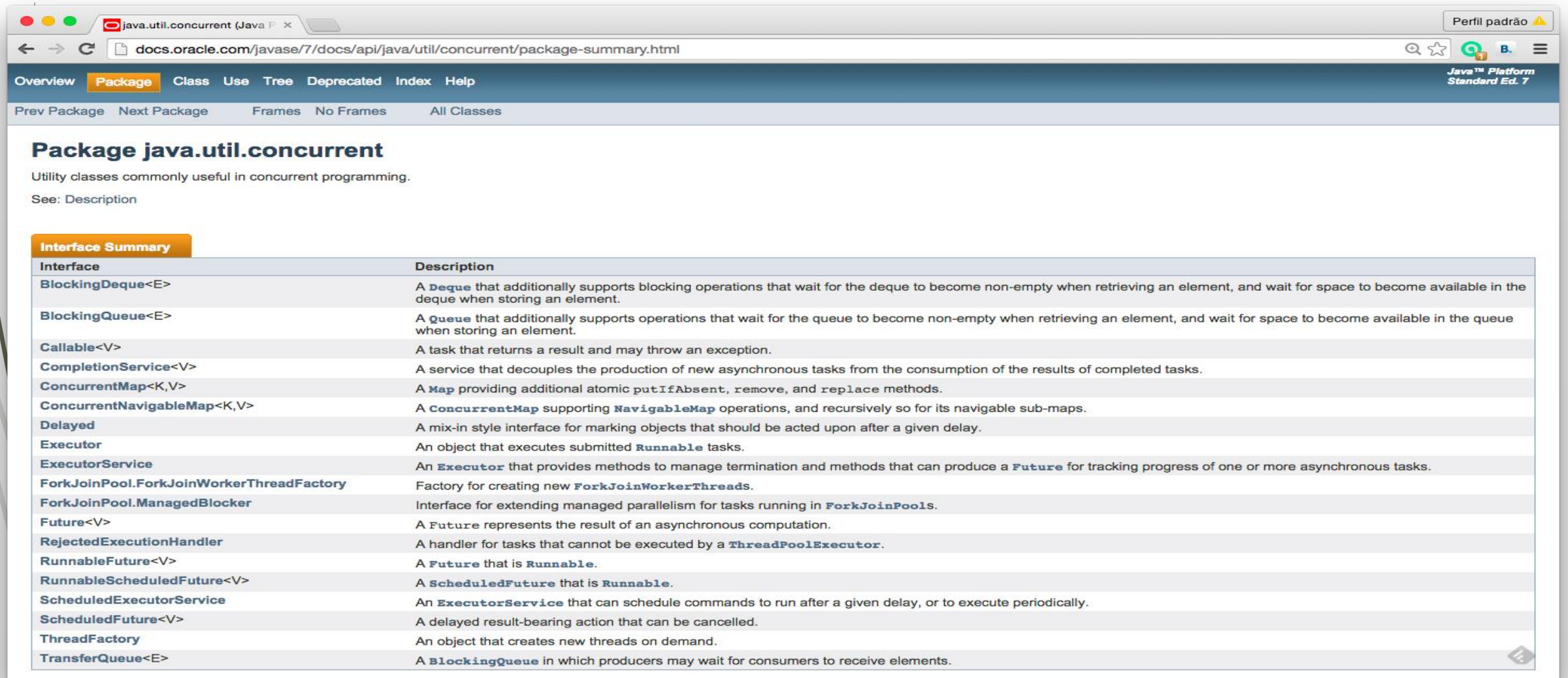




Indo além....



# Pacote de Concorrência do Java



The screenshot shows the Oracle Java API documentation for the `java.util.concurrent` package. The browser address bar shows the URL `docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`. The page has a navigation bar with tabs for Overview, Package (selected), Class, Use, Tree, Deprecated, Index, and Help. Below the navigation bar, there are links for Prev Package, Next Package, Frames, No Frames, and All Classes. The main heading is "Package java.util.concurrent" with a subtitle "Utility classes commonly useful in concurrent programming." and a link to "See: Description". A section titled "Interface Summary" contains a table with two columns: "Interface" and "Description".

Interface	Description
<code>BlockingDeque&lt;E&gt;</code>	A <code>Deque</code> that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element.
<code>BlockingQueue&lt;E&gt;</code>	A <code>Queue</code> that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
<code>Callable&lt;V&gt;</code>	A task that returns a result and may throw an exception.
<code>CompletionService&lt;V&gt;</code>	A service that decouples the production of new asynchronous tasks from the consumption of the results of completed tasks.
<code>ConcurrentMap&lt;K,V&gt;</code>	A <code>Map</code> providing additional atomic <code>putIfAbsent</code> , <code>remove</code> , and <code>replace</code> methods.
<code>ConcurrentNavigableMap&lt;K,V&gt;</code>	A <code>ConcurrentMap</code> supporting <code>NavigableMap</code> operations, and recursively so for its navigable sub-maps.
<code>Delayed</code>	A mix-in style interface for marking objects that should be acted upon after a given delay.
<code>Executor</code>	An object that executes submitted <code>Runnable</code> tasks.
<code>ExecutorService</code>	An <code>Executor</code> that provides methods to manage termination and methods that can produce a <code>Future</code> for tracking progress of one or more asynchronous tasks.
<code>ForkJoinPool.ForkJoinWorkerThreadFactory</code>	Factory for creating new <code>ForkJoinWorkerThreads</code> .
<code>ForkJoinPool.ManagedBlocker</code>	Interface for extending managed parallelism for tasks running in <code>ForkJoinPools</code> .
<code>Future&lt;V&gt;</code>	A <code>Future</code> represents the result of an asynchronous computation.
<code>RejectedExecutionHandler</code>	A handler for tasks that cannot be executed by a <code>ThreadPoolExecutor</code> .
<code>RunnableFuture&lt;V&gt;</code>	A <code>Future</code> that is <code>Runnable</code> .
<code>RunnableScheduledFuture&lt;V&gt;</code>	A <code>ScheduledFuture</code> that is <code>Runnable</code> .
<code>ScheduledExecutorService</code>	An <code>ExecutorService</code> that can schedule commands to run after a given delay, or to execute periodically.
<code>ScheduledFuture&lt;V&gt;</code>	A delayed result-bearing action that can be cancelled.
<code>ThreadFactory</code>	An object that creates new threads on demand.
<code>TransferQueue&lt;E&gt;</code>	A <code>BlockingQueue</code> in which producers may wait for consumers to receive elements.

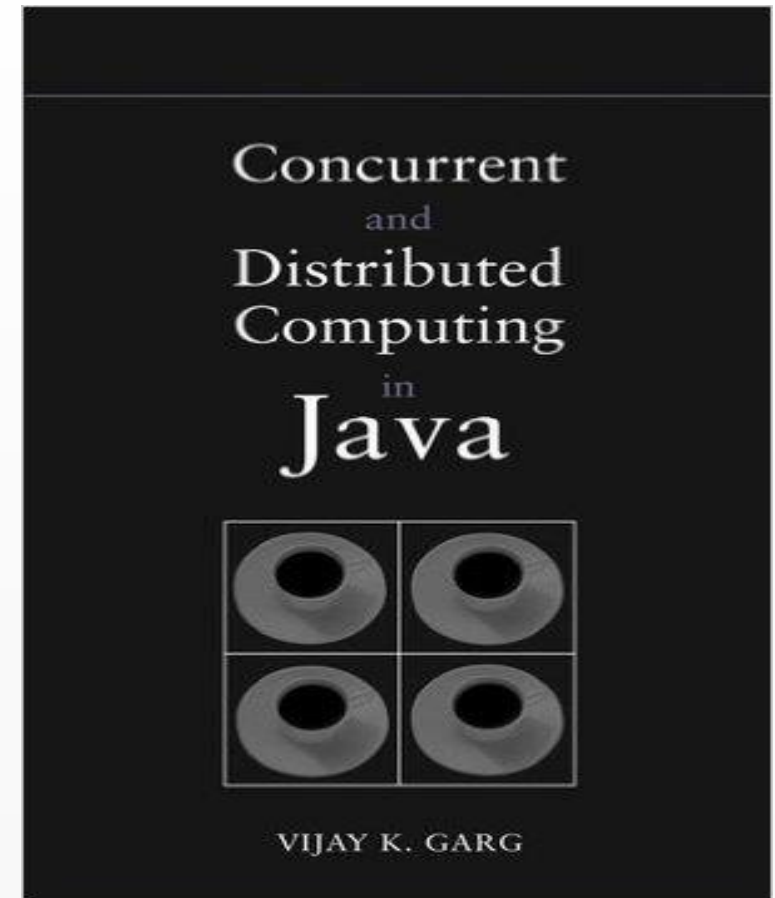
# Classe Java Futures

- Inspirado no Coroutines do Kotlin
- Future: Classe que encapsula uma chamada feita em paralelo
  - Cancelar a execução de uma tarefa,
  - Descobrir se a execução já terminou com sucesso ou erro, entre outras operações;
- FutureTask: É uma implementação da interface Future a ser executada numa chamada em paralelo.
- Callable: Interface para a implementação de uma execução em paralelo.
  - Similar a interface Runnable
  - A Callable deve retornar um valor ao final da execução;
- ExecutorService: Classe para o gerenciamento de execuções em paralelo
  - Cria um pool de threads, iniciando e cancelando as execuções.
  - Também é possível cancelar este, evitando assim a criação de novas tarefas.

# Recomendação de Livro

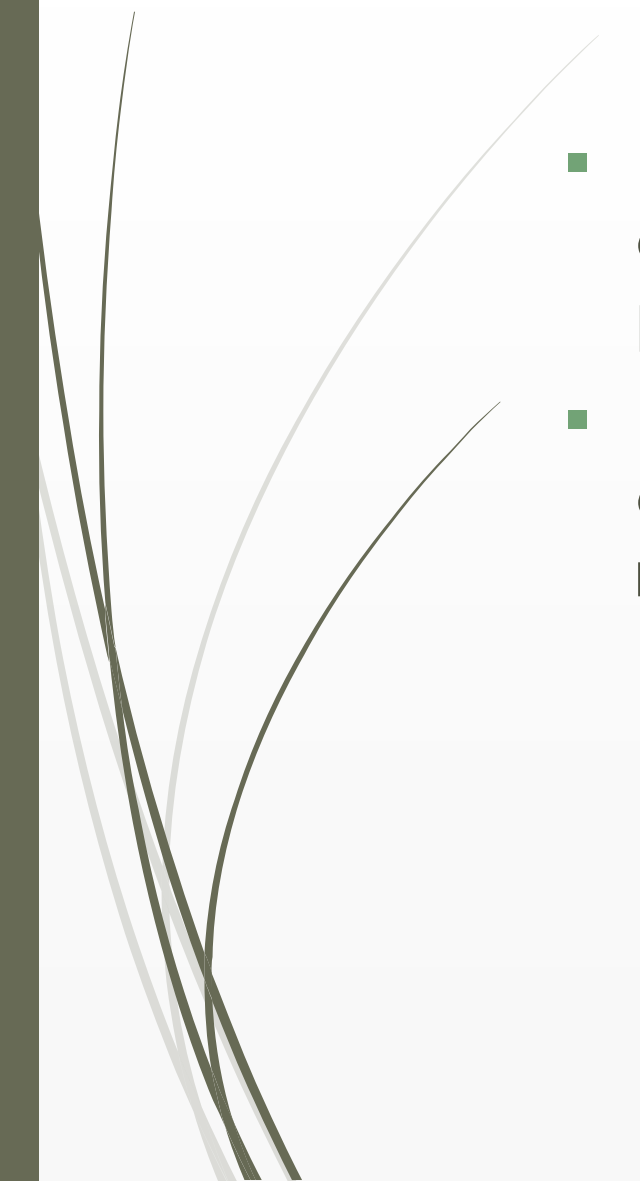
- O livro aborda questões práticas de concorrência e distribuição usando Java.
- Muita Thread e Socket!

Fonte: <http://www.wiley.com/WileyCDA/WileyTitle/productCd-047143230X,miniSiteCd-IEEE2.html>





# Threads em Sistemas Distribuídos

- Proporcionam um meio conveniente para permitir chamadas bloqueadoras de sistema sem bloquear o processo inteiro no qual o thread está executando;
  - Imagine um processo monothread: O que aconteceria com o processo quando a interrupção da placa de rede é feita para envio/recebimento de dados?
- 



# Clientes Multithread

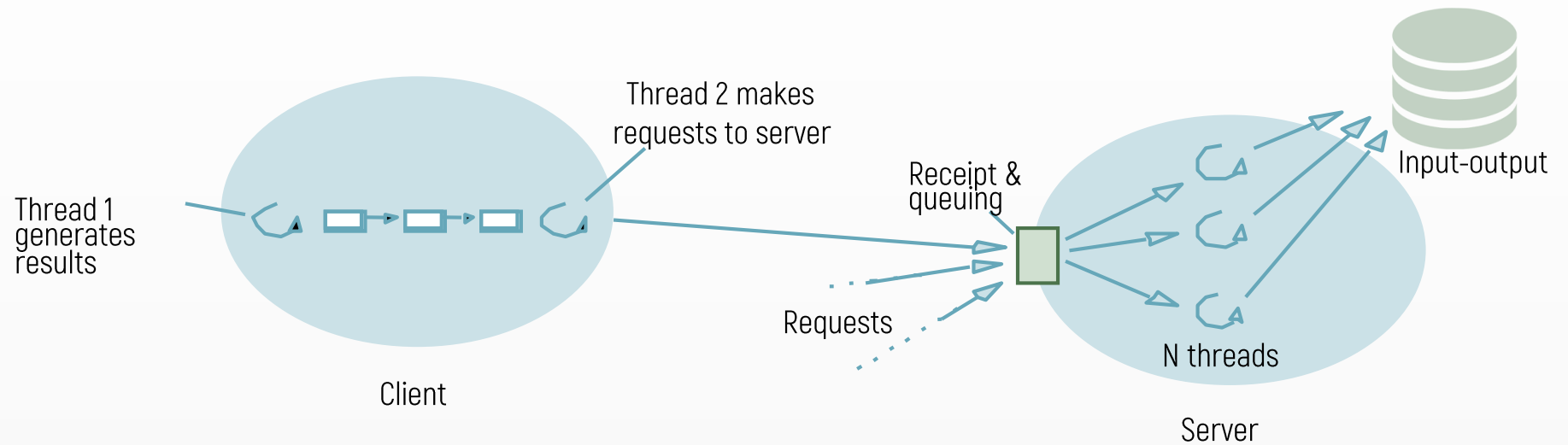
- Usados para ocultar latências de comunicação, separando threads de envio/recebimento de dados com threads de processamento da interface.
  - Torna possível recebimento de vários arquivos de uma página WEB ao mesmo tempo;
  - Torna possível acesso a vários servidores (redundantes), que servirão os dados independentemente, gerando maior velocidade.



# Servidores Multithread

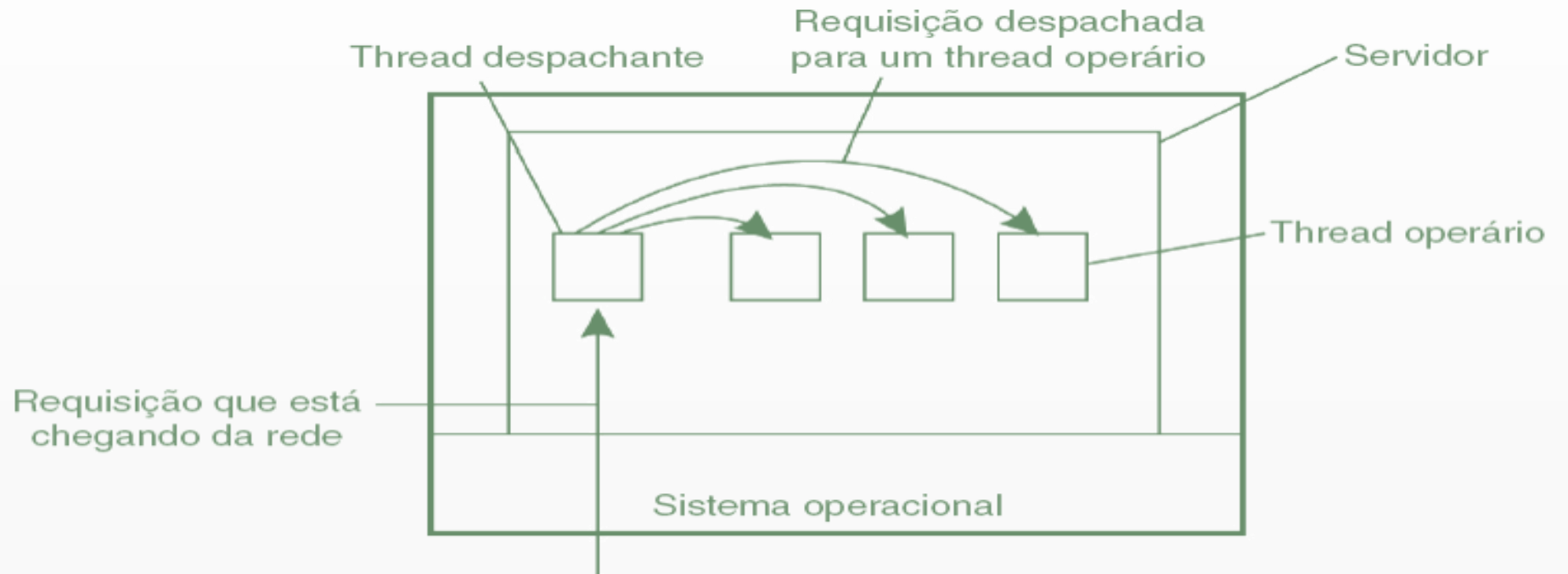
- Além de simplificar o código do servidor, explora paralelismo para obter alto desempenho, mesmo em sistemas monoprocessadores;
  - Um thread despachante cria a divisão de vários threads com tarefas distintas, como ler disco, receber dados de socket, enviar dados para socket, atender N usuários simultaneamente;
  - O thread despachante atribui a requisição a um thread operário ocioso (bloqueado).
- Servidores Monothread não poderiam atender a um segundo usuário enquanto lê disco!

# Abordagem Cliente-Servidor com Thread



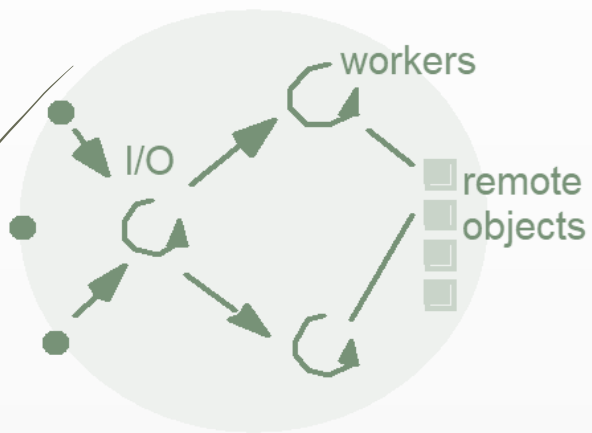
# Servidores Multithread

## Modelo despachante/operário

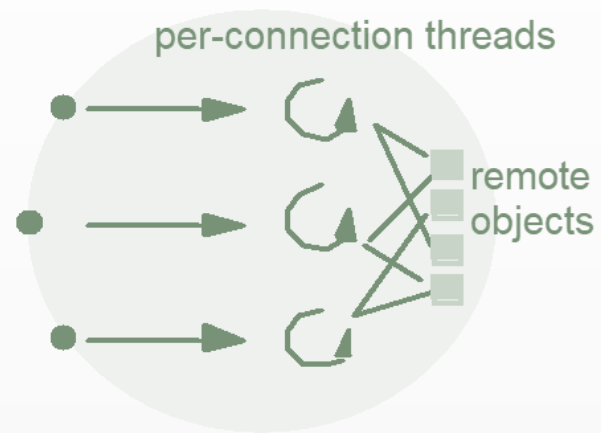




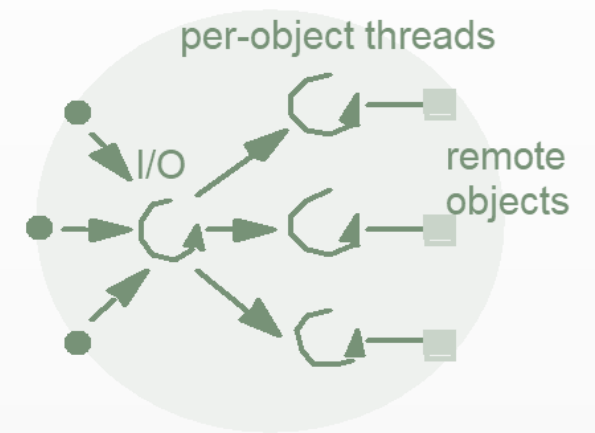
# Abordagem Cliente-Servidor com Thread



a. Thread-per-request



b. Thread-per-connection



c. Thread-per-object

# Servidores Multithread

- Uma terceira alternativa seria usar uma máquina de estados finitos, que consiste em apenas um thread que usa chamadas não bloqueadoras como *send* e *receive*.

Modelo	Características
Threads	Paralelismo, chamadas bloqueadoras de sistema
Processo monothread	Sem paralelismo, chamadas bloqueadoras de sistema
Máquina de estado finito	Paralelismo, chamadas de sistema não bloqueadoras

**Tabela 3.1** Três modos de construir um servidor.