



Threads

Prof. Edwar Saliba Júnior
Janeiro de 2013



Introdução

- *Multithreading*:
 - fornece múltiplas *threads* de execução para a aplicação;
 - permite que programas realizem tarefas concorrentemente;
 - às vezes faz-se necessário a sincronização das *threads* para que estas funcionem de acordo com o que se propõe no programa.



Dica de desempenho

- Um problema com aplicativos de uma única *thread* é que atividades longas devem ser concluídas antes que outras atividades possam ser iniciadas.
- Em um aplicativo com múltiplas *threads*, as *threads* podem ser distribuídas por múltiplos processadores (se estiverem disponíveis), de modo que múltiplas tarefas são realizadas concorrentemente e o aplicativo pode operar de modo mais eficiente.
- *Multithreading* também pode aumentar o desempenho em sistemas de um único processador que simula a concorrência; quando uma *thread* não puder prosseguir, outra pode utilizar o processador.



Dica de Portabilidade

- Ao contrário das linguagens que não têm capacidades de *multithreading* integradas (como C e C++) e, portanto, devem fazer chamadas não-portáveis para primitivos de *multithreading* do sistema operacional, o Java inclui primitivos de *multithreading* como parte da própria linguagem e de suas bibliotecas. Isso facilita a manipulação de *threads* de maneira portátil entre plataformas.



Estados de *thread*: Classe *Thread*

- Estados da *thread*:
 - **novo**:
 - uma nova *thread* inicia seu ciclo de vida no estado **novo**;
 - permanece nesse estado até o programa iniciar a *thread*, colocando-a no estado **executável**;
 - **executável**:
 - uma *thread* que entra nesse estado está executando sua tarefa;
 - **em espera**:
 - uma *thread* entra nesse estado a fim de esperar que uma outra *thread* realize uma tarefa.



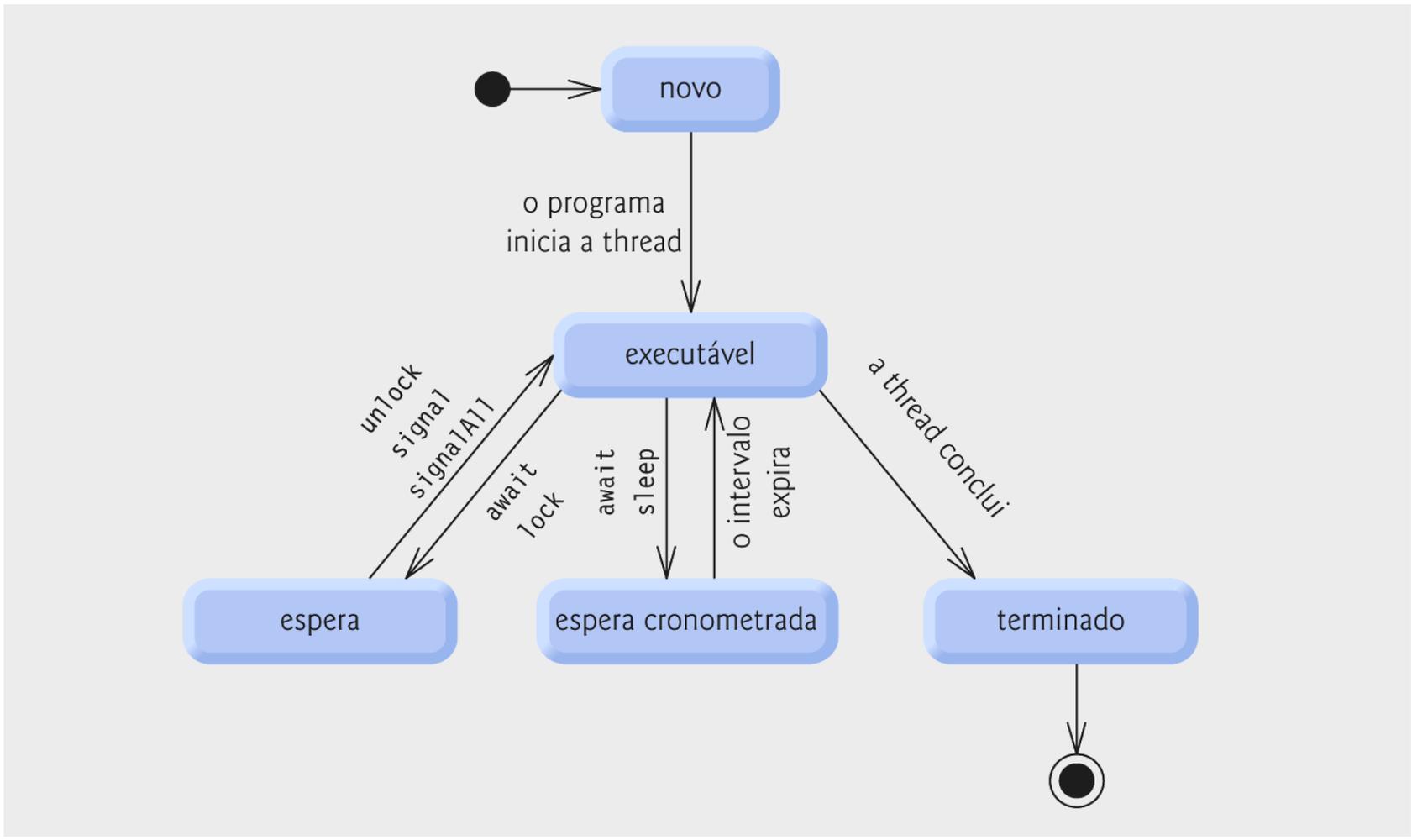
Estados de *thread*: Classe *Thread*

- Estados de *thread*: (continuação)
 - **espera cronometrada:**
 - uma *thread* entra nesse estado para esperar uma outra *thread* ou para transcorrer um determinado período de tempo;
 - uma *thread* nesse estado retorna ao estado **executável** quando ela é sinalizada por uma outra *thread* ou quando o intervalo de tempo especificado expirar;
 - **terminado:**
 - uma *thread* no estado **executável** entra nesse estado quando completa sua tarefa.



Programação Orientada a Objetos

Diagrama de Estado do Ciclo de Vida da *Thread*



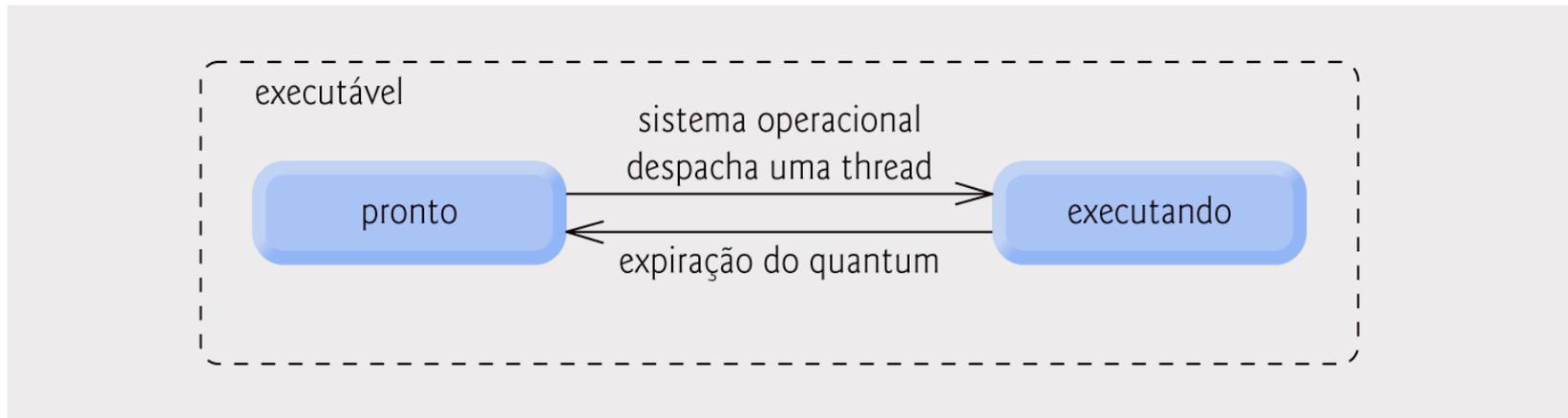


Estados da *thread*: Classe *Thread*

- Visão do sistema operacional do estado **executável**:
 - **pronto**:
 - uma *thread* nesse estado não está esperando uma outra *thread*, mas está esperando que o sistema operacional atribua a *thread* a um processador;
 - **em execução**:
 - uma *thread* nesse estado tem atualmente um processador e está executando;
 - uma *thread* no estado **em execução** frequentemente utiliza uma pequena quantidade de tempo de processador chamada fração de tempo, ou *quantum*, antes de migrar de volta para o estado **pronto**.



Visualização Interna do Sistema Operacional do Estado Executável da Thread em Java





Prioridades e Agendamento de *Threads*

- Prioridades:
 - cada *thread* Java tem uma prioridade;
 - as prioridades do Java estão no intervalo entre **MIN_PRIORITY** (uma constante de 1) e **MAX_PRIORITY** (uma constante de 10);
 - as *threads* com prioridade mais alta são mais importantes e terão um processador alocado antes das *threads* com prioridades mais baixas;
 - a prioridade-padrão é **NORM_PRIORITY** (uma constante de 5).



Prioridades e agendamento de *Threads*

- Agendador de *thread*:
 - determina qual *thread* é executada em seguida;
 - uma implementação simples executa *threads* com a mesma prioridade no estilo rodízio;
 - *threads* de prioridade mais alta podem fazer preempção da *thread* atualmente **em execução**.
 - em alguns casos, as *threads* de prioridade alta podem adiar indefinidamente *threads* de prioridade mais baixa; o que também é conhecido como inanição.

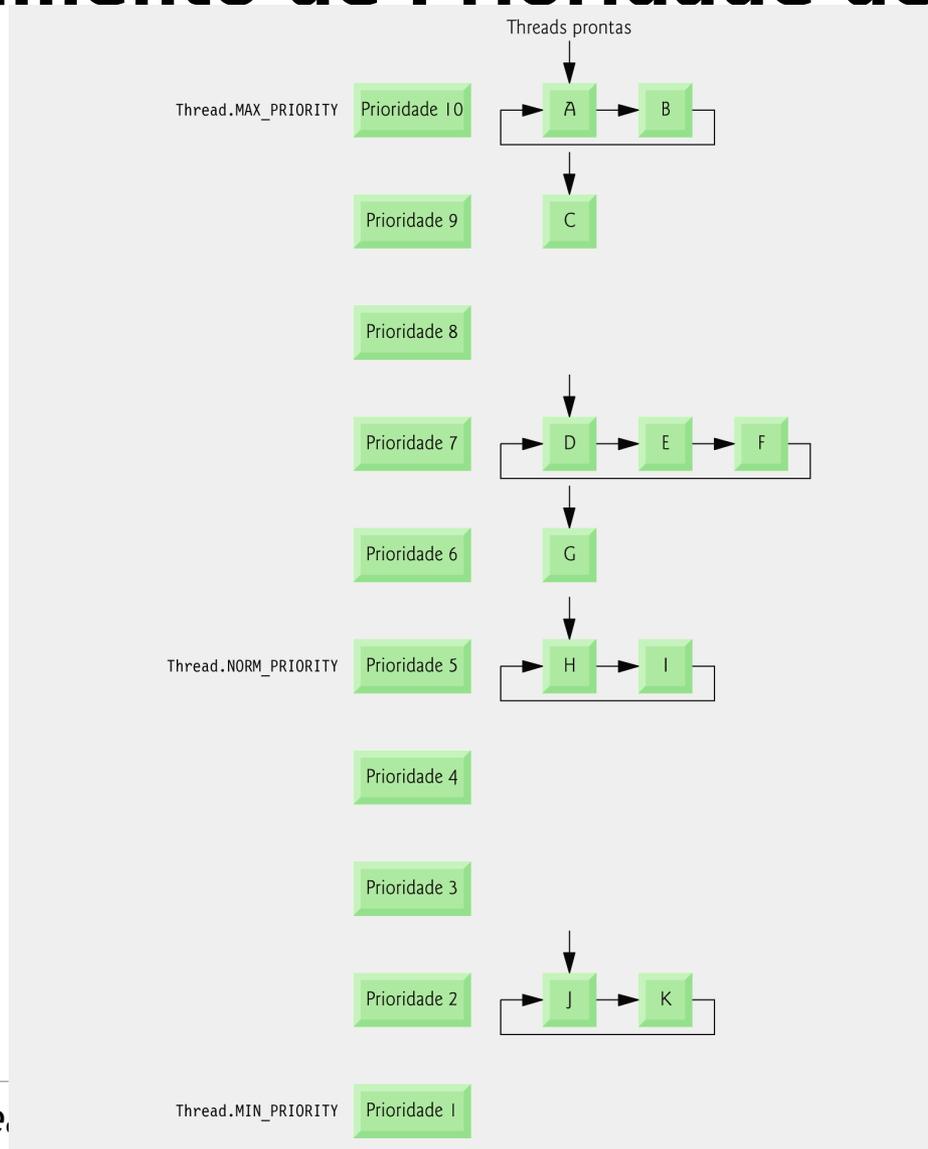


Dica de portabilidade

- O agendamento de *thread* é dependente de plataforma;
- um aplicativo que utiliza *multithreading* pode comportar-se diferentemente em plataformas distintas (e. g. Windows e GNU/Linux).



Agendamento de Prioridade de *Threads*





Criando e Executando *Threads*

- A interface **Runnable**:
 - meio preferido de criar um aplicativo com *multithreads*;
 - declara o método **run**;
 - executado por um objeto que implementa a interface **Executor**.
- Interface **Executor**:
 - eeclara o método **execute**;
 - cria e gerencia um grupo de *threads* chamado *pool de threads*.



Criando e Executando *Threads*

- Interface **ExecutorService**:
 - é uma subinterface de **Executor** que declara outros métodos para gerenciar o ciclo de vida de um **Executor**;
 - pode ser criada utilizando os métodos `static` da classe **Executors**;
 - o método **shutdown** finaliza as *threads* quando as tarefas são concluídas.
- Classe **Executors**:
 - o método **newFixedThreadPool** cria um *pool* que consiste em um número fixo de *threads*;
 - o método **newCachedThreadPool** cria um *pool* que cria novas *threads* conforme necessário.



Exemplo 01

- Interface Runnable
 - Parte 01 - Classe Main.
 - Parte 02 - Classe ImprimeTarefa



Sincronização de *Thread*

- Sincronização de *threads*:
 - fornecido ao programador com exclusão mútua,
 - acesso exclusivo a um objeto compartilhado;
 - implementado no Java utilizando bloqueios.
- **Interface Lock**:
 - o método **lock** obtém o bloqueio, impondo a exclusão mútua;
 - o método **unlock** libera o bloqueio;
 - a classe **ReentrantLock** implementa a interface **Lock**.



Sincronização de *Thread*

- Variáveis de condição:
 - se uma *thread* que mantém o bloqueio não puder continuar a sua tarefa até uma condição ser satisfeita, a *thread* pode esperar uma variável de condição;
 - criadas chamando **newCondition** do método **Lock**;
 - representadas por um objeto que implementa a **interface Condition**.
- **Interface Condition**:
 - declara os métodos: **await** - para fazer uma *thread* esperar; **signal** - para acordar uma *thread* em espera; e **signalAll** - para acordar todas as *threads* em espera.



Erro comum de Programação

- O impasse (*deadlock*) ocorre quando uma *thread* em espera (vamos chamá-la de *thread1*) não pode prosseguir porque está esperando (direta ou indiretamente) outra *thread* (vamos chamá-la de *thread2*) prosseguir;
- Simultaneamente, a *thread2* não pode prosseguir porque está esperando (direta ou indiretamente) a *thread1* prosseguir.
- Como duas *threads* estão esperando uma à outra, as ações que permitiriam a cada *thread* continuar a execução nunca ocorrem.



Observação de Engenharia de *Software*

- O bloqueio que ocorre com a execução dos métodos **lock** e **unlock** pode levar a um impasse se os bloqueios nunca forem liberados;
- As chamadas para método **unlock** devem ser colocadas em blocos **finally** para assegurar que os bloqueios sejam liberados e evitar esses tipos de impasses.



Erro Comum de Programação

- É um erro se uma *thread* emitir um `await`, um `signal` ou um `signalAll` em uma variável de condição sem adquirir o bloqueio dessa variável de condição.
- Isso causa uma `IllegalMonitorStateException`.



Relacionamento entre Produtor e Consumidor Sem Sincronização

- Relacionamento produtor/consumidor:
 - O produtor gera dados e os armazena na memória compartilhada;
 - O consumidor lê os dados da memória compartilhada;
 - A memória compartilhada é chamada *buffer*.



Exemplo 02

Buffer Não Sincronizado

- Parte 01 - Classe Main.
- Parte 02 - Interface Buffer
- Parte 03 - Classe BufferNaoSincronizado
- Parte 04 - Classe Produtor
- Parte 05 - Classe Consumidor



Relacionamento entre Produtor e Consumidor com Sincronização

- Relacionamento produtor/consumidor:
 - Este exemplo utiliza **Locks** e **Conditions** para implementar a sincronização.



Erro Comum de Programação

- Esquecer de sinalizar (`signal`) uma *thread* que está esperando por uma condição, é um erro de lógica;
- A *thread* permanecerá no estado de *espera*, o que a impedirá de continuar trabalhando;
- Essa espera pode levar a um adiamento indefinido ou a um impasse.



Exemplo 03

Buffer Sincronizado

- Parte 01 - Classe Main.
- Parte 02 - Interface Buffer.
- Parte 03 - Classe BufferSincronizado.
- Parte 04 - Classe Produtor.
- Parte 05 - Classe Consumidor.



Relacionamento de Produtor / Consumidor: *Buffer Circular*

- *Buffer* circular:
 - Fornece espaço extra no *buffer*, onde o produtor pode colocar valores e o consumidor pode lê-los.



Dica de Desempenho

- Mesmo ao utilizar um *buffer* circular, é possível que uma *thread* produtora possa preencher o *buffer*, o que a forçaria a esperar até que uma consumidora consumisse um valor para liberar um elemento do *buffer*;
- De maneira semelhante, se o *buffer* estiver vazio em qualquer dado momento, a *thread* consumidora deve esperar até que a produtora produza outro valor;
- A chave para utilizar um *buffer* circular é otimizar o tamanho do *buffer* para minimizar a quantidade de tempo de espera da *thread*.



Exemplo 04 ***Buffer Circular***

- Parte 01 - Classe Main.
- Parte 02 - Interface Buffer.
- Parte 03 - Classe BufferCircular.
- Parte 04 - Classe Produtor.
- Parte 05 - Classe Consumidor.



Relacionamento Produtor / Consumidor: ArrayBlockingQueue

- ArrayBlockingQueue
 - Versão completamente implementada do *buffer* circular;
 - Implementa a interface **BlockingQueue**
 - Declara os métodos **put** e **take** para gravar e ler dados do *buffer*, respectivamente.



Exemplo 05 ArrayBlockingQueue

- Parte 01 - Classe Main.
- Parte 02 - Interface Buffer.
- Parte 03 - Classe ArrayBlockingQueue.
- Parte 04 - Classe Produtor.
- Parte 05 - Classe Consumidor.



***Multithreading* com GUI**

- Componentes GUI Swing:
 - Não são seguros para *threads*;
 - As atualizações devem ser realizadas no caso de uma *thread* de despacho de evento:
 - Utiliza o método `static invokeLater` da classe `SwingUtilities` e passa para ele um objeto `Runnable`.



Exemplo 06 ***Threads / GUI***

- Parte 01 - Classe RandomCharacters / Main.
- Parte 02 - Classe RunnableObject.



Outras Classes e Interfaces em `java.util.concurrent`

- Interface **Callable**:
 - declara o método **call**;
 - o método **call** permite que uma tarefa concorrente retorne um valor ou lance uma exceção;
 - o método **ExecutorService** `submit` recebe uma **Callable** e retorna uma **Future** que representa o resultado da tarefa.
- Interface **Future**:
 - declara o método **get**.
 - o método **get** retorna o resultado da tarefa representada pela **Future**.



Monitores e Bloqueios de Monitor

- Monitores:
 - cada objeto Java tem um monitor;
 - permite que uma *thread* por vez execute dentro de uma instrução **synchronized**;
 - *threads* que esperam para adquirir o bloqueio de monitor são colocadas no estado *bloqueado*;
 - o método **Object wait** coloca uma *thread* no estado de espera;
 - o método **Object notify** acorda uma *thread* em espera e
 - o método **Object notifyAll** acorda todas as *threads* em espera.



Observação de Engenharia de *Software*

- O bloqueio que ocorre com a execução dos métodos **synchronized** podem levar a um impasse se os bloqueios nunca forem liberados;
- Quando ocorrem exceções, o mecanismo de exceção do Java se coordena com o mecanismo de sincronização para liberar bloqueios e evitar esses tipos de impasses.



Erro Comum de Programação

- Ocorre um erro se uma *thread* emite um **wait**, um **notify** ou um **notifyAll** sobre um objeto sem adquirir um bloqueio para ela;
- Isso causa uma **IllegalMonitorStateException**.



Exemplo 07

Método *Synchronized*

- Parte 01 - Classe Main.
- Parte 02 - Interface Buffer.
- Parte 03 - Classe SynchronizedBuffer.
- Parte 04 - Classe Produtor.
- Parte 05 - Classe Consumidor.

Bibliografia

- DEITEL, H. M.; DEITEL, P. J. **Java Como Programar**; tradução Edson Furmankiewicz; revisão técnica Fábio Lucchini. 6a. ed., São Paulo: Pearson, 2005.