UNIDADE 2 – CONCEITOS BÁSICOS DE UM APLICATIVO MULTITHREADING

MÓDULO 1 – CONCEITOS BÁSICOS - PROCESSOS E THREADS

01

1 - PROCESSO

Partindo dos fundamentos apresentados na unidade anterior para criação de um código Java robusto, apresentaremos, a partir de agora, os principais conceitos de alto nível envolvidos na programação paralela ou concorrente para qualquer sistema operacional multitarefa, inclusive o Android, que é um sistema operacional baseado no kernel Linux.

Todos os computadores, *smartphones*, tablets, dentre outros aparelhos modernos, são capazes de fazer várias coisas ao mesmo tempo. As pessoas acostumadas a trabalhar com esses dispositivos podem não estar completamente cientes deste fato, contudo é importante compreendê-lo.

Em qualquer sistema multiprogamado, a CPU cria chaves de programa para programa, executando cada um deles por dezenas ou centenas de milissegundos. Estritamente falando, enquanto a cada instante do tempo a CPU executa somente um programa, no decorrer de um único segundo do tempo ela pode trabalhar por vários programas, dando aos usuários a ilusão de um paralelismo. Um dos principais objetivos da multiprogramação é melhorar o turnaround do software, ou seja, o tempo transcorrido desde o momento em que o software entra em execução e o instante em que termina sua execução.

Algumas vezes, nesse contexto, fala-se de pseudoparalelismo, para diferenciar do verdadeiro paralelismo, por exemplo, dos sistemas multiprocessados simétricos (SMP) que são aqueles constituídos de duas ou mais CPUs/cores que compartilham simultaneamente a mesma memória principal física.

Neste modelo, todos os programas/*softwares* que podem ser executados no computador são organizados em vários processos. Um conceito fundamental para todos os sistemas operacionais, inclusive o Android, é o de **processo**.

Um **processo** é basicamente um programa (conjunto de instruções e dados) em execução, ou seja, a cada processo está associado um espaço de endereçamento.

02

Esse **espaço de endereçamento** nada mais é que uma lista de posições de memória principal/trabalho/RAM que vai de 0 até um valor máximo “n” cujas posições podem ser lidas ou escritas pelo respectivo processo.

O espaço de endereçamento contém o programa executável, os dados do programa e sua pilha. Os processos geralmente são divididos em três segmentos:

- **text** (código do programa),
- **data** (variáveis estáticas do programa) e
- **heap** (parte dinâmica do programa).

Também associado a cada processo está um conjunto de recursos, normalmente incluindo registradores (que inclui o contador e um ponteiro para a pilha), uma lista de arquivos abertos, alarmes pendentes, lista de processos relacionados e todas as demais informações necessárias para executar um programa.

Um processo é fundamentalmente um contêiner que armazena todas as informações necessárias para executar um programa. Conceitualmente, cada processo tem sua própria CPU virtual. É óbvio que na realidade isso não existe, pois o que acontece é que a CPU troca, a todo momento, de um processo para outro conforme explicado anteriormente. Essa troca de CPU entre processos é denominada de **escalonamento de processos**.

03

Atualmente, os escalonadores da maioria dos sistemas operacionais iterativos são classificados como preemptivos.

Preemptivos são escalonadores soberanos que têm a capacidade de interromper qualquer processo em execução para que outro processo de prioridade mais alta possa executar.

Um outro modelo de escalonamento é chamado de colaborativo

Escalonamento **colaborativo** é aquele no qual o escalonador é autônomo e necessita que os processos presentes no sistema cooperem com o suposto escalonador para que a política de troca de processos possa funcionar, ou seja, nestes casos o escalonador somente consegue retirar o processo de execução se o processo, explicitamente, solicitar isso, independente do motivo.

Convém notar que, se um mesmo programa está sendo executado duas vezes, isso conta como dois processos na memória. Por exemplo, se o editor de textos (word) está aberto em um documento onde você está editando seu currículo e o mesmo editor está aberto novamente em um outro documento onde você está editando uma carta para seus avós, apesar de ser o mesmo programa, tem-se dois processos distintos (um para cada documento aberto).

04

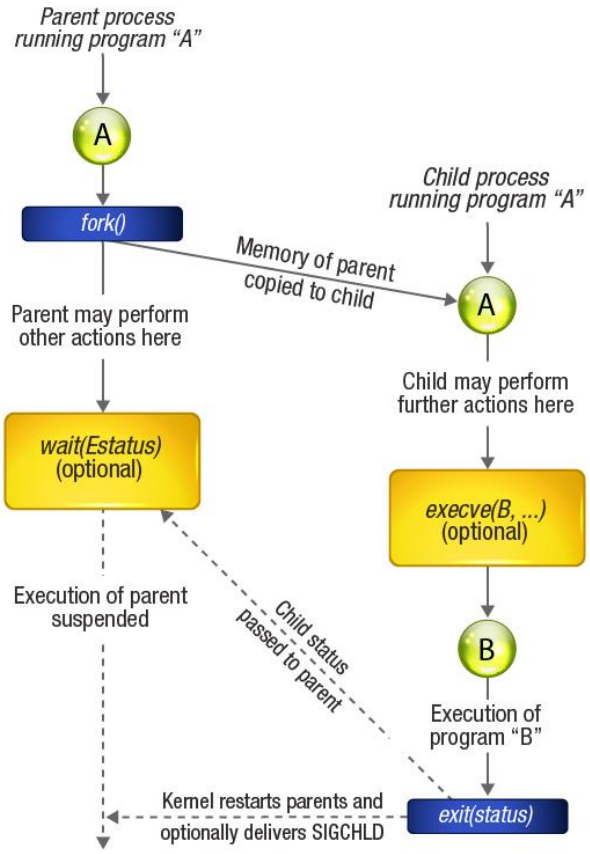
E por fim, a criação de processos é feita sempre com base em um processo matriz. Isso significa que a **criação de um novo processo envolve dois passos** essenciais:

Dependendo do sistema operacional, isso é feito por duas *syscalls* ou apenas uma. No caso dos sistemas padrão POSIX, existem duas syscalls:

- uma para criar a cópia o processo, denominada de **fork** e
- outra para substituir o arquivo binário, chamada de **exec**.

No caso dos sistemas padrão win32 (microsoft), existe apenas uma syscall denominada de **CreateProcess** a qual executa a criação da cópia e também a substituição do binário executável. No caso do Linux, que é aderente ao padrão POSIX, o processo matriz do sistema operacional é chamado de **init** (processo Pai de todos os demais presentes no sistema operacional). Consequentemente todos os demais processos do sistema operacional Linux são descendentes, diretos ou indiretos, de init.

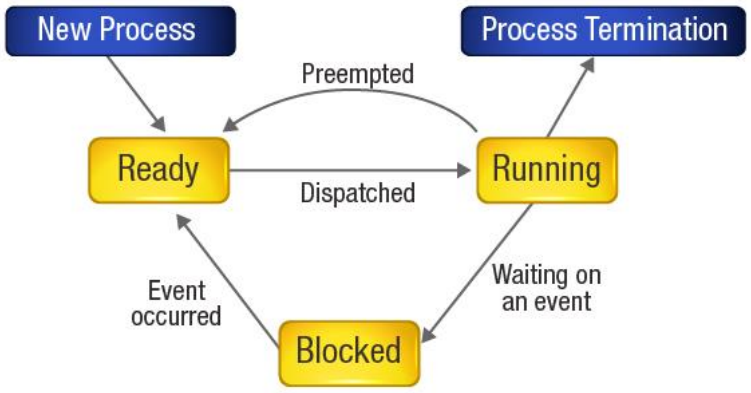
A figura a seguir exibe esse procedimento.



05

1.1- Estados de um processo

Embora cada processo seja uma entidade independente, com seu próprio contador de programa e estado interno, muitas vezes os processos precisam interagir uns com os outros, como por exemplo, um processo pode gerar uma saída que outro processo utiliza como entrada. Deste modo, classicamente, o sistema operacional classifica o processo em três estados distintos, conforme a figura:



Os três estados são representados no diagrama pelas elipses azuis e significam:

- **Running**

De fato usando a CPU naquele instante, ou seja, conjunto de instruções e dados alocados e em execução dentro dos registradores da CPU.

- **Ready**

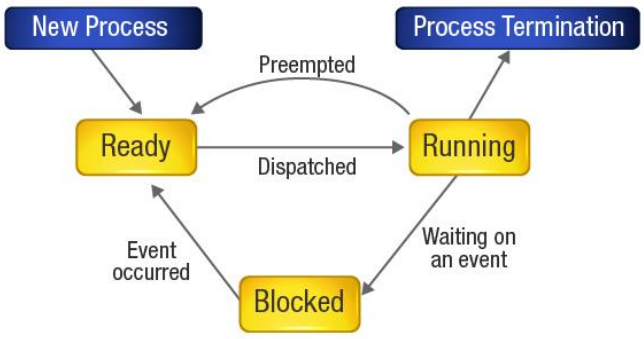
Um processo pronto para executar, contudo está parado em função da CPU está sendo ocupada por outro processo.

- **Blocked**

Incapaz de executar enquanto não ocorrer um evento externo ao processo em questão.

06

Ainda analisando a imagem, logicamente, os dois primeiros estados são similares, pois em ambos o processo vai executar, contudo, no segundo estado (Ready) não há, temporariamente, CPU disponível para ele. Já o terceiro estado (Blocked) é diferente dos dois primeiros, pois o processo não pode executar mesmo que a CPU não tenha nada para fazer.



Observando as setas que unem as elipses, podem-se observar quatro possíveis transições de estado. A transição “Waiting on an event” ocorre quando o sistema operacional descobre que um processo não pode prosseguir.

Essa descoberta pode ser feita explicitamente pelo processo com uma chamada de sistema (syscall) específica que avisa ao SO sobre a sua condição ou de forma implícita através do acesso a um recurso que não esteja disponível naquele instante.

As transições “preempted” e “dispatched” são causadas pelo escalonador de processos – uma parte/módulo do SO - sem que o processo saiba disso. A transição “preempted” ocorre quando o escalonador decide que o processo em execução já teve tempo suficiente de CPU e é o momento de deixar outro processo ocupar o tempo da CPU. A transição “dispatched” ocorre quando todos os outros

processos já compartilharam a CPU, de uma maneira justa, e é hora de o primeiro processo obter novamente a CPU.

07

O escalonamento, isto é, a decisão sobre quando e por quanto tempo cada processo deve executar é um tópico que extrapola o conteúdo desta disciplina.

No caso específico do Android, cujo kernel é Linux, para saber mais, procure sobre Completely Fair Schedule - CFS. A transição “Event Ocurred” ocorre quando acontece um evento externo pelo qual um processo estava aguardando.

Com o modelo de processo, torna-se muito mais fácil saber o que está ocorrendo dentro do sistema. Alguns dos processos chamam programas que executam comando digitados pelo usuário. Outros processos são parte do sistema operacional e manejam tarefas como fazer requisições por serviço de arquivos ou gerenciar detalhes de uma conexão de rede.

Completely Fair Schedule

https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt

08

2 - THREAD

Em sistemas operacionais tradicionais, cada processo tem um espaço de endereçamento e uma única thread de controle. Na verdade, isso é quase uma definição de processo.

Uma **thread** é o fluxo de controle de um processo.

Frequentemente há situações em que é desejável ter múltiplas threads de controle no mesmo espaço de endereçamento executando em paralelo ou pseudoparalelo, como se eles fossem processos separados (exceto pelo espaço de endereçamento compartilhado). As threads são classificadas como sendo “processos leves” ou miniprocessos.

A principal razão para existirem threads é que em muitas aplicações ocorrem múltiplas atividades ao mesmo tempo. A programação multithread é praticamente idêntica a de multiprocesso.

Um detalhe importante e fundamental no multithread é que o espaço de endereçamento e todos os seus dados são compartilhados por todas as threads pertencentes ao mesmo processo.

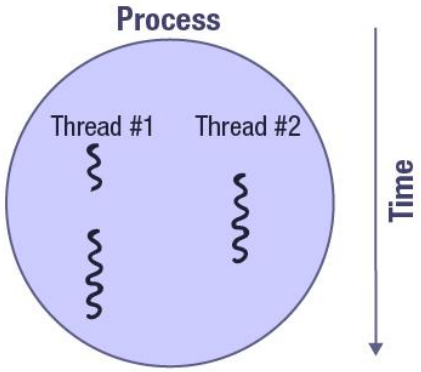
Uma segunda razão para existência de threads é que elas são fáceis de criar e destruir, ou seja, mais rápidas para começar e finalizar algo, uma vez que a quantidade de recursos associada a elas é bem menor que a quantidade de recursos associados a um processo inteiro/completo.

09

Uma terceira razão para o uso de thread é quando existe grande quantidade de atividades CPU bound e I/O bound no mesmo processo.

Nesse caso, é quase que uma certeza que as threads irão se sobrepor em função da diferença de velocidade (uns rápidos e outros lentos) entre os vários dispositivos envolvidos.

A figura abaixo mostra um desenho clássico do referido conceito:



É importante perceber que threads distintas em um processo não são tão independentes quanto processos distintos. Todas as threads de um mesmo processo têm exatamente o mesmo espaço de endereçamento, o que significa que elas compartilham as mesmas variáveis globais, por exemplo.

10

Como cada thread pode ter acesso a qualquer endereço de memória dentro do espaço de endereçamento do processo, uma thread pode ler, escrever ou até mesmo apagar completamente a pilha de outra thread. Portanto, não existe qualquer tipo de proteção entre as threads de um mesmo processo. Já no caso de processos diversos, essa proteção é uma consequência natural de seu espaço de endereçamento separado.

Além disso, as threads também possuem estados de execução, assim como os processos, o que significa que uma thread pode estar nos estados de *running*, *ready* ou *blocked*.

É importante salientar, que, no caso da máquina virtual Java, a implementação de threads da referida tecnologia possui mais dois estados que são:

- iniciado e
- finalizado.

Na próxima etapa do nosso estudo, será mostrado esse diagrama com os cinco estados de uma thread para a JVM.

A tabela abaixo lista, na primeira coluna, alguns itens compartilhados por todas as threads em um mesmo processo, enquanto que a segunda coluna lista alguns itens específicos a cada thread.

Itens por Processo	Itens por Thread
Espaço de endereçamento	Contador de programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos Filhos	Estado
Alarmes pendentes	
Sinais e seus manipuladores	

11

3 - Zygote

No caso específico do sistema operacional Android, Zygote é considerado um processo Pai de todos os demais processos do sistema operacional.

Trata-se de uma instância da VM (Dalvik ou Art) que se encontra em estado ideal para dar origem a novos processos do sistema operacional Android. Ou seja, o Zygote nada mais é que o processo **init** do Linux acrescido de todas as classes-base da API Android necessárias para se executar os aplicativos Android.

Essa alteração se deve ao simples fato de tentar tornar a inicialização de aplicativos Android mais eficiente, evitando que todo aplicativo posto em execução (novo processo) tenha de carregar, sempre, o mesmo conjunto de classes-base essenciais. Assim como no Linux, no Android cada aplicativo executa em um processo separado e utiliza-se das syscalls **fork** e **exec** para criar as cópias de processos e substituir os binários executáveis, conforme explicado anteriormente.

Podemos dizer, portanto, que o init está para o sistema operacional Linux assim como o Zygote está para o sistema operacional Android.

12

4 - SANDBOXES: PROCESSOS E USUÁRIOS

A segurança do Android depende muito das restrições de segurança no nível do sistema operacional Linux, especificamente no que diz respeito aos processos e limites de nível de usuário. Como o Android é projetado para dispositivos pessoais, utilizados por uma única pessoa, ele utiliza, de forma muito interessante, a habilidade dos sistemas Linux de oferecer suporte a múltiplos usuários: o Android cria um novo usuário para cada fornecedor de aplicativo, o que significa que cada aplicativo será executado com privilégios de usuários diferentes (exceto aqueles assinados pelo mesmo fornecedor).

Deste modo, todos os arquivos de propriedade de um mesmo fornecedor/aplicativo são, por padrão, inacessíveis a outros fornecedores/aplicativos.

Por outro lado, o Android permite que o mesmo usuário logado no dispositivo utilize múltiplos aplicativos, executados em níveis de usuários Linux diferentes. O efeito prático disso é uma maior segurança, decorrente de cada aplicativo ser executado em seu próprio “espaço”.

O sistema foi construído desta forma, pois os projetistas imaginaram um ambiente de inúmeros pequenos aplicativos, criados por muitos fornecedores e que não podem ser sondados em termos de confiabilidade. Assim, os aplicativos não têm acesso direto aos dados uns dos outros.

Sabia+ sobre sandboxes.

Sandbox

O modo sandbox permite que programas potencialmente exploráveis (como navegadores) ou qualquer outro aplicativo rodem em um ambiente virtual seguro, isolado do restante do seu sistema.

13

RESUMO

O objetivo deste módulo foi apresentar os fundamentos relativos a programação paralela/concorrente em sistemas operacionais de tempo compartilhado. Além disso, apresentou também os conceitos fundamentais relativos a qualquer sistema operacional moderno como:

- programa: é um conjunto de instruções e dados;

- processo: é um programa em execução;

- thread: é um fluxo de controle do processo;

- multithreading: capacidade de um sistema em executar várias threads simultaneamente;

- zygote: é um processo Pai de todos os processos criados em um sistema operacional Android;

- sandbox: no caso específico do Android, pode-se dizer que é um ambiente isolado de execução para cada fornecedor de aplicativos visando garantir o isolamento, segurança e a autonomia de cada ambiente.

UNIDADE 2 – CONCEITOS BÁSICOS DE UM APLICATIVO MULTITHREADING

MÓDULO 2 – CONCEITOS BÁSICOS – THREADS EM JAVA (PARTE 1)

01

1 - ESTADOS DE THREAD: CICLO DE VIDA DE UMA THREAD

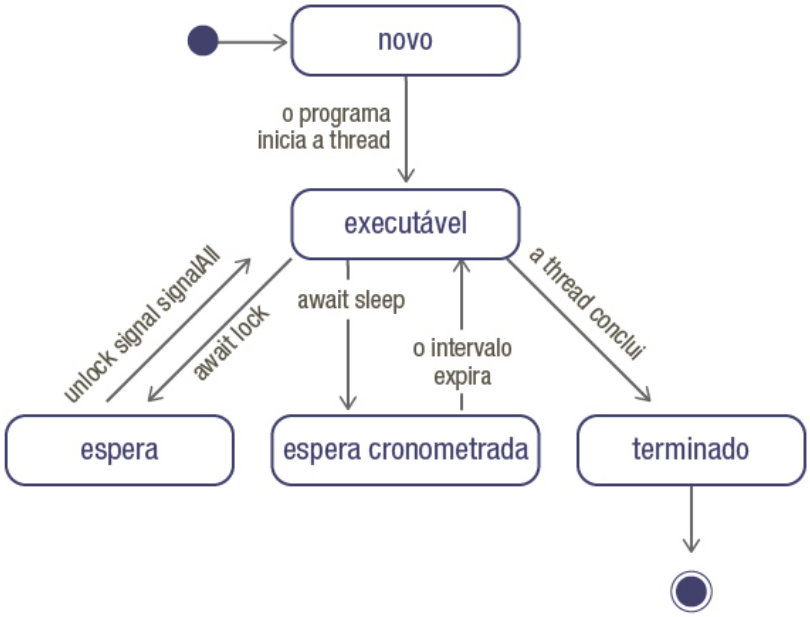
O Java disponibiliza a concorrência por meio da linguagem e das APIs.

É possível criar aplicativos que contenham threads, ou linhas de execução separadas, nas quais cada uma tenha sua própria pilha de chamadas de métodos e seu próprio contador de programa, permitindo a **execução simultânea com outras threads** ao compartilhar recursos no nível do aplicativo como a memória. Essa capacidade é chamada de **multithreading** e, o caso da linguagem Java, esse recurso independe do sistema operacional no qual o aplicativo está sendo executado.

Ao contrário de linguagens que não possuem capacidade de multithreading integradas, e, portanto, devem fazer chamadas não portáveis para primitivas de multithreading do sistema operacional, o Java inclui primitivas de multithreading como parte da própria linguagem e de suas bibliotecas permitindo a manipulação portável de aplicativos multithreading entre diversas plataformas (CPU + Sistema Operacional) diferentes. Portanto, os aplicativos Android escritos em linguagem Java podem fazer uso desse poderoso recurso de programação de computadores.

A qualquer instante, uma thread pode estar em qualquer um dos seus vários estados disponíveis. Esse modelo de estados é similar àquele modelo de estado de processos visto no modulo anterior.

A figura a seguir exibe esses possíveis estados:



multithreading

Um programa single - thread (thread única) inicia na etapa 1 e continua sequencialmente (etapa 2, etapa 3, etapa 4) até atingir a etapa final. Aplicações multithread permitem que você execute várias threads ao mesmo tempo, cada uma executando um passo por exemplo.

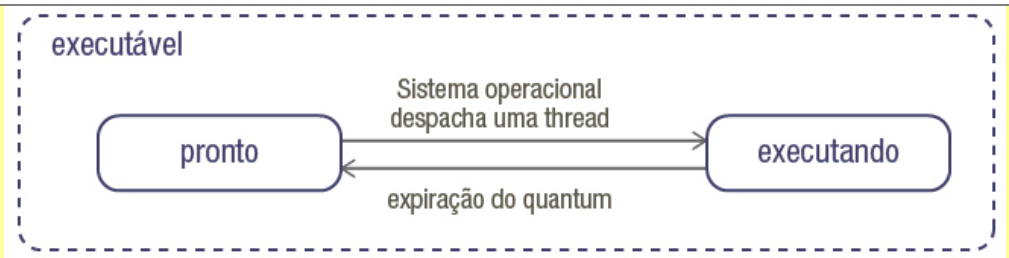
Cada thread é executada em seu próprio processo, então, teoricamente, você pode executar o passo 1 em uma thread e, ao mesmo tempo executar o passo 2 em outra thread e assim por diante. Isso significa que a etapa 1, etapa 2, etapa 3 e etapa 4 podem ser executadas simultaneamente. Utiliza-se multithread quando precisamos de desempenho e eficiência. Por exemplo, em um jogo de videogame, o ideal seria criar threads para o processamento das rotinas de imagens e outras para rotinas de áudio.

Novo (New)

Uma nova thread sempre inicia seu ciclo de vida no estado de novo permanecendo nele até que o processo inicie a thread.

Executável (Runnable)

O estado de executável é aquele em que a thread está de fato executando sua tarefa. Contudo, é importante observar que esse estado pertence a uma máquina virtual que é uma camada de abstração sobre o sistema operacional. Isso significa que do ponto de vista do sistema operacional a situação é demonstrada pela figura abaixo:



Ou seja, para o sistema operacional, o único estado de executável das threads que executam sobre máquina virtual, é representado como sendo dois possíveis estados ao nível do SO: **pronto** (ready) ou **executando** (running). O sistema operacional oculta esses dois estados da máquina virtual que visualiza apenas o estado executável (runnable).

Espera (Wait)

Às vezes uma thread executável transita para o estado de espera enquanto aguarda uma outra thread realizar uma tarefa. Uma thread neste estado de espera transita de volta para o estado de executável apenas quando outra thread a notifica para continuar executando.

Espera Cronometrada (Timed Wait)

Uma thread pode entrar no estado de espera cronometrada por um intervalo especificado de tempo. Ela transita para o estado de executável quando o intervalo de tempo expira ou quando o evento pelo qual ela está esperando ocorre.

Terminado (Finished)

Uma thread executável entra no estado de terminado quando completa sua tarefa com sucesso ou, de outra forma, a termina, talvez por causa de um erro durante o seu processamento.

02

2- Prioridades de Threads

Toda thread do java tem uma prioridade que ajuda a determinar a ordem em que as mesmas serão agendadas. As prioridades do Java variam entre MIN_PRIORITY (uma constante de valor 1) até MAX_PRIORITY (uma constante de valor 10). Por padrão, toda thread criada recebe a prioridade NORM_PRIORITY (uma constante de valor 5).

Cada nova thread herda a prioridade da thread que a cria. Informalmente, as threads de prioridade mais alta são mais importantes para um programa e devem ser alocadas em tempo de CPU antes das

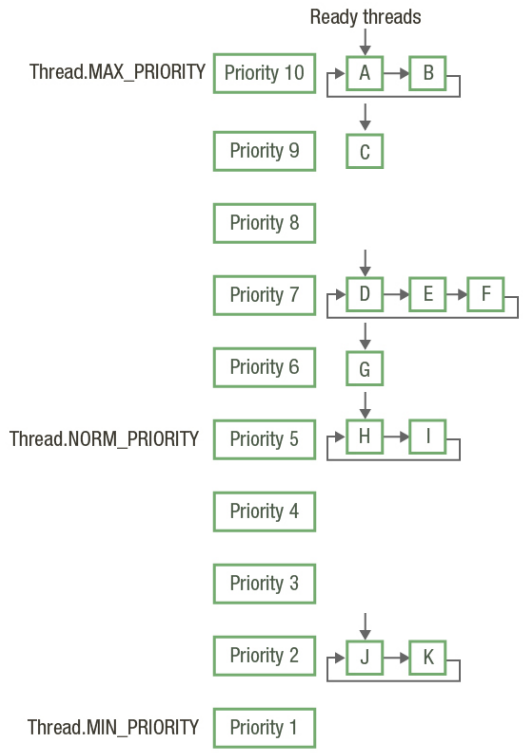
threads de prioridade mais baixa. Entretanto, as prioridades de threads não podem garantir a ordem em que elas serão executadas.

A maioria dos sistemas operacionais suporta o fracionamento de tempo (*timesharing*), o que permite que as threads de igual prioridade compartilhem um processador. Sem o fracionamento de tempo, cada thread em um conjunto de threads de igual prioridade executa até a sua conclusão (a menos que ela deixe o estado executável e transite para o estado de espera ou espera cronometrada ou, ainda, seja interrompida por uma thread de prioridade mais alta) antes de outras threads de igual prioridade terem uma chance de executar.

Com o *timesharing*, mesmo se a thread não tiver concluído a execução quando seu quantum expirar, o processador é tirado da thread e recebe outra thread de igual prioridade, se houver alguma disponível. O escalonador de um sistema operacional determina que thread execute em seguida.

03

Por **exemplo**, uma simples implementação do scheduler de thread mantém a thread de prioridade mais alta executando o tempo todo e, se houver mais de uma thread de prioridade mais alta, isso assegura que cada uma dessas threads execute por um quantum no estilo rodízio. A figura a seguir ilustra esse cenário.



A figura apresenta uma fila de prioridade de múltiplos níveis para as threads.

Na figura, supondo um computador de um único processador, as threads A e B executam por um quantum no esquema de rodízio até que ambas as threads completem sua execução. Isso significa que A obtém um quantum de tempo para executar. Então B obtém um quantum. Então A obtém outro quantum. Então B obtém outro quantum. E isso continua até que uma thread complete a sua execução.

O processador dedica então toda a sua energia à thread que resta (a menos que outra thread de prioridade 10 se torne pronta). Em seguida, a thread C executa até sua conclusão (assumindo que nenhuma thread de prioridade igual ou mais alta chegará). E esse procedimento continua por todas as threads presentes até as suas respectivas conclusões.

04

Quando uma thread de prioridade mais alta entra no estado de pronto, o sistema operacional geralmente faz preempção da thread atualmente em execução.

Dependendo do SO, as threads de prioridade mais alta poderiam adiar, possivelmente por um tempo indefinido, a execução de thread de prioridade mais baixa. Esse adiamento indefinido é conhecido pelo nome de **inanição** ou ***starvation***.

É importante não confundir essa definição com outro problema muito comum em programação concorrente/paralela que é o **deadlock**. Deadlock ocorre quando os processos/threads permanecem bloqueados indefinidamente uns pelos outros aguardando a liberação (que nunca acontece) de recursos por eles alocados.

É obvio que para ambos os problemas, a consequência imediata é a não execução do programa. Contudo, o programa que se encontra em *starvation* pode se autorrecuperar, desde que o estado do ambiente mude para uma condição que lhe seja favorável (lembre que isso pode levar pouco tempo ou muito tempo, ou seja, o tempo de espera é indefinido), enquanto que no deadlock essa autorrecuperação do programa não é possível. A recuperação de um programa em deadlock deve ser feito por uma intervenção externa ao processo para que o mesmo volte a funcionar.

Ambos os problemas são complexos e difíceis de identificar, uma vez que suas características não se apresentam de modo regular e constante. Ou seja, um programa que possua qualquer um dos problemas citados pode eventualmente, sob determinadas circunstâncias, apresentar o problema ou não. Isso significa que o programa pode funcionar em determinados momentos e em outros não.

05

3 - Portabilidade

O scheduler de threads é dependente da plataforma, o que significa que o comportamento de um programa de múltiplas threds escritos em Java pode variar nas diferentes implementações existentes.

Esteja atento que, ao projetar programas de múltiplas threads, a **condição do sistema operacional** onde elas executarão é um ponto importante a ser considerado na sua implementação. Utilizar prioridades diferentes da prioridade padrão tornará seu programa dependente da plataforma. Portanto, se o objetivo for a portabilidade, não ajuste as prioridades da thread.

06

4 - Exemplos

A forma preferida para se criar aplicativos Java multithread é utilizando a interface *Runnable* do pacote *java.lang* e a classe *Executors* do pacote *java.util.concurrent*.

Um objeto runnable representa uma "tarefa" que pode ser executada concorrentemente/paralelamente com outras tarefas. A interface *Runnable* especifica uma única operação denominada de **run,** que contém o código que define a tarefa que um objeto *Runnable* deve realizar.

Quando uma thread executando um *Runnable* é criada e iniciada, ela chama o método **run** do objeto *Runnable*, que executa na nova thread. O código a seguir é um exemplo de como criar e executar threads bem como as tarefas a serem executadas por elas.

07

a) Classe ImprimeTarefa.java

```
package br.aiec.multithread;

/**
 * Cada objeto desta classe representa uma thread que é inicializada, entra em
 * execução, espera por um tempo
 * aleatório (yield), retorna a execução após o tempo aleatório de espera, e então
 * finaliza sua execução.
 *
 * @author Guilherme Veloso
 *
 */
public class ImprimeTarefa implements Runnable {
	private String nomeThread;

	public ImprimeTarefa(String nome) {
		nomeThread = nome;
	}

	public void run() {
		System.out.printf("%s pausando.\n",     nomeThread);

		Thread.yield();

		System.out.printf("%s voltou a funcionar.\n", nomeThread);
	}
}
```

08

b) Classe Principal

```
package br.aiec.multithread;

import java.util.concurrent.Executors;
```

```
import java.util.concurrent.ExecutorService;

/**
 * Essa classe apenas cria um pool (conjunto) de threads por
 * meio da classe "Executors" e então cria três objetos do tipo
 * "ImprimeTarefa" que serão executados
 * concorrentemente/paralelamente pelas threads (uma para cada tarefa)
 * presentes no pool.
 *
 * @author Guilherme Veloso
 *
 */

public class Principal {
	public static void main(String[] args) {

		Runnable tarefa1 = new ImprimeTarefa("thread1");
		Runnable tarefa2 = new ImprimeTarefa("thread2");
		Runnable tarefa3 = new ImprimeTarefa("thread3");

		ExecutorService pool = Executors.newCachedThreadPool();

		System.out.println("Iniciando as threads secundárias");
		pool.execute(tarefa1);
		pool.execute(tarefa2);
		pool.execute(tarefa3);

		pool.shutdown();

		System.out.println("Threads secundárias iniciadas.");
		System.out.println("Thread principal terminando!");
	}
}
```

09

Um ponto importante a ser observado após a execução deste exemplo, é que a sequência de execução das threads, muito possivelmente, será sempre diferente. Apesar de o exemplo utilizar-se de uma pausa aleatória (método *yield*) para tornar mais perceptível a aleatoriedade da execução das threads, considere que tal situação (execução aleatória) acontece mesmo sem a presença deste tipo de instrução no código.

O método *yield* na verdade avisa ao scheduler que está liberando a CPU e, portanto, o scheduler coloca outra thread para executar enquanto retorna a thread que disparou a mensagem *yield* para o final da fila de threads prontas (ready).

O uso do método *yield* é totalmente desaconselhado para aplicações reais, uma vez que causa atrasos na execução do código. Seu uso deve ser feito apenas para fins de teste ou debug do *software*.

Portanto, a finalidade de utilizar o método *yield* nesse exemplo é para apenas tornar perceptível ao ser humano a questão relativa à execução aleatória. Lembre-se de que o escalonador do sistema operacional e da máquina virtual definem, ambos, cada qual a sua política de escalonamento. Desta forma, é importante perceber que na programação concorrente/paralela não é possível fazer suposições de qual thread e nem por quanto tempo uma thread executará.

10

4.1- Exemplo "Antigo"

Muitos códigos Java escritos antes da versão 1.5 da linguagem utilizavam a forma abaixo, que criavam threads usando o recurso de herança (*extends*). Contudo, a referida forma de implementação é, atualmente, desaconselhada, pois não faz um uso ótimo do processador uma vez que a quantidade de threads criadas (é fixa no código, observe a classe PrincipalOld.java) pode ser muito além ou muito aquém do ótimo necessário para determinada arquitetura computacional.

a) ImprimeTarefaOld.java

```
package br.aiec.multithread;

public class ImprimeTarefaOld extends Thread{

	private String nomeThread;

	public ImprimeTarefaOld(String nome) {
		this.nomeThread = nome;
	}


	public void run(){
		System.out.printf("%s pausando.\n",     nomeThread);

		Thread.yield();

		System.out.printf("%s voltou a funcionar.\n", nomeThread);
	}
}
```

11

b) PrincipalOld.java

```
package br.aiec.multithread;
```

```
/**
 * Essa classe apenas cria três threads (de forma fixa no código) por
 * meio do construtor da referida classe passando como parâmetro
 * os objetos da classe ImprimeTarefaOld para que possam ser executados
 * concorrentemente/paralelamente pelas threads criadas.
 *
 * ESSA FORMA DE PROGRAMAÇÃO É DESACONSELHADA!
 *
 * @author Guilherme Veloso
 *
 */

public class PrincipalOld {

	public static void main (String[] args){
		ImprimeTarefaOld tarefa1 = new ImprimeTarefaOld("thread1");
		ImprimeTarefaOld tarefa2 = new ImprimeTarefaOld("thread2");
		ImprimeTarefaOld tarefa3 = new ImprimeTarefaOld("thread3");

		Thread x = new Thread(tarefa1);
		Thread y = new Thread(tarefa2);
		Thread z = new Thread(tarefa3);

		System.out.println("Iniciando as threads secundárias");
		x.start();
		y.start();
		z.start();

		System.out.println("Threads secundárias iniciadas.");
		System.out.println("Thread principal terminando!");

	}
}
```

12

RESUMO

O objetivo deste módulo foi apresentar como a linguagem Java oferece suporte nativo a implementação de threads (independente de plataforma), além de seus possíveis estados:

- novo: o estado inicial assumido por todas as threads novas;

- executável: o estado em que a thread está executando sua tarefa;

- espera: o estado em que a thread aguarda por um evento externo;

- espera cronometrada: o estado em que a thread aguarda por um temporizador;

- terminado: o estado que indica que a thread finalizou sua tarefa.

Além disso, os níveis de prioridade que variam de 1 (menor prioridade) até 10 (maior prioridade), bem como a consequência de seu uso, também foram apresentados neste módulo.

UNIDADE 2 – CONCEITOS BÁSICOS DE UM APLICATIVO MULTITHREADING

MÓDULO 3 – CONCEITOS BÁSICOS – THREADS EM JAVA (PARTE 2)

01

1 - SINCRONIZAÇÃO DE THREAD

Quando múltiplas threads compartilham um objeto e ele é modificado por uma ou várias delas, podem ocorrer resultados indeterminados (como veremos nos exemplos mais adiante), a menos que o acesso ao objeto compartilhado seja gerenciado adequadamente.

Se uma thread estiver atualizando um objeto compartilhado e outra thread também tentar atualizá-lo no mesmo instante, não se pode garantir a integridade do objeto em questão, ou seja, não é clara a atualização de qual thread entrará em vigor. Isso é conhecido como **condição de corrida**.

Quando isso acontecer, não se pode confiar no comportamento do programa, uma vez que ele poderá produzir tantos resultados corretos quanto resultados errados, e da pior maneira possível, esporadicamente.

Isso significa que o programa às vezes funciona e às vezes não. Além disso, essa oscilação entre funcionar e não funcionar não possui um padrão de comportamento, o que torna a descoberta do problema algo ainda mais complexo.

O referido problema pode ser resolvido fornecendo a somente uma thread, por vez, o acesso ao conjunto de instruções compartilhadas (**região/seção crítica**), ou seja, acesso exclusivo à parte do código que manipula o objeto compartilhado.

02

Durante esse tempo, outras threads que desejarem manipular o objeto são mantidas em espera. Quando a thread com acesso exclusivo ao objeto termina de manipulá-lo, uma das threads que foi mantida na espera tem a permissão de prosseguir.

Esse processo é chamado de **sincronização de threads** e tem por objetivo coordenar o acesso a dados compartilhados por múltiplas threads/processos concorrentes ou paralelos.

Sincronizando threads/processos dessa maneira, você pode assegurar que cada thread/processo que acessa um objeto compartilhado exclui todas as outras threads/processos de fazer isso simultaneamente ficando, tal conceito, batizado com o nome de **exclusão mútua**.

Existem algumas técnicas diferentes para sincronizar threads/processos concorrentes/paralelos. A seguir serão descritas apenas três técnicas (**semáforo**, **mutex** e **monitor**) que irão auxiliar o desenvolvimento dos exemplos que estudaremos a partir de agora. Contudo, existem outras técnicas que extrapolam o conteúdo desta disciplina e não serão tratadas aqui.

03

1.1- Semáforo

Um **semáforo** é uma variável inteira que, dentre outras coisas, é utilizada para garantir a exclusão mútua.

Um semáforo tem a propriedade de contar qualquer coisa que se deseje. Geralmente, um semáforo é usado para contar o número de sinais de acordar salvos para uso futuro, a quantidade de itens presentes ou ausentes em um buffer, a posição onde se deve inserir ou remover itens de um buffer, dentre outros. Geralmente, o tipo de dado “int” é usado para representar um semáforo em diversas linguagens de programação.

1.2- Mutex

Um **mutex** é uma versão simplificada de um semáforo e por vezes, é também chamado de semáforo binário. Um mutex é uma variável que pode estar em um de dois estados:

- impedido ou
- desimpedido.

Consequentemente, somente 1 bit é necessário para representá-lo. Nas linguagens onde existe o tipo de dado “boolean”, o mesmo é uma opção suficiente e adequada para tal representação. Contudo, em linguagens onde o tipo de dado “boolean” não existe, geralmente utiliza-se o tipo “int” para um domínio de apenas dois valores, geralmente, zero para desimpedido e qualquer outro valor para impedido.

Frequentemente, zero (0) e um (1) são os valores escolhidos para representar o estado de desimpedido e de impedido, respectivamente.

04

1.3- Monitor

O **monitor** foi uma solução proposta para uso de semáforos ou mutex ou alguma outra técnica de exclusão mútua de forma indireta pelas threads/processos que necessitam acessar um recurso compartilhado. Ou seja, um monitor pode fazer uso, por exemplo, de semáforo ou mutex para implementar as suas regras.

Isso foi proposto em função dos inúmeros detalhes necessários para se implementar (principalmente aqueles ligados à ordem de escrita das instruções) o semáforo ou o mutex de forma direta nas rotinas que acessam o recurso compartilhado. Essa implementação não é trivial, como será percebido nos códigos de exemplo.

Com o uso de monitores, as rotinas não precisam mais conhecer os detalhes envolvidos no uso de semáforo ou mutex. Contudo, na construção do monitor em si, esses detalhes devem ser considerados, sempre!

No caso dessa disciplina, o monitor será construído usando-se um semáforo para um exemplo específico (buffer com cinco posições compartilhadas da memória) e um mutex para um outro exemplo (buffer com uma única posição compartilhada de memória). Os códigos serão apresentados na próxima etapa do nosso estudo.

Um **monitor** é uma coleção de rotinas, variáveis e estruturas de dados, tudo isso agrupado em um tipo especial de módulo ou pacote. Os processos/threads podem chamar as rotinas em um monitor quando quiserem, mas não podem ter acesso direto às estruturas internas do monitor. O acesso é feito por meio de sua interface pública.

05

Os monitores apresentam uma propriedade importante que os tornam úteis para realizar a exclusão mútua, assim como os semáforos e mutexes**: somente um processo pode estar ativo** em um monitor em um dado momento.

No caso do Java, todo objeto criado pela linguagem possui um monitor intrínseco que pode ser acessado/utilizado pelo uso da palavra reservada **synchronized**. Além disso, a classe Object (a classe cósmica do Java, aquela que é o Pai de todas!) possui três métodos que são usados com esses monitores intrínsecos:

- **wait,**

- **notify** e
- **notifyAll**.

O uso desta palavra reservada e desses métodos é muito comum em códigos Java que fazem uso de multithreading.

Observe, contudo, que os exemplos apresentados a seguir irão um pouco além do uso destes recursos próprios da linguagem Java, demonstrando em um nível maior de detalhes, como funcionam um monitor e como você poderá construir um, caso julgue necessário.

Não se esqueça de que tal programação é extremamente peculiar e deve ser feita com muito zelo, pois erros na ordem de escrita das instruções de um monitor podem por tudo a perder provocando erros graves como *deadlock*, *starvation* ou mesmo comportamentos imprevisíveis e irreprodutíveis.

Deste modo, um monitor pode ser representado como um tipo de dado abstrato com suas operações e dados, ou seja, no caso da linguagem Java será representado por uma classe com seus métodos e atributos.

06

2 - Exemplo: o problema do Produtor-Consumidor

As primitivas de sincronização serão demonstradas por meio de um exemplo clássico chamado de **produtor-consumidor**.

O problema consiste, por exemplo, em duas threads, uma **produtora** e uma **consumidora**, que acessam um buffer comum (variável de memória compartilhada) de tamanho fixo. O produtor grava um número inteiro no buffer enquanto que o consumidor lê o referido número. Deste modo, todo número gravado no buffer deve ser lido apenas uma única vez.

O código abaixo é de uso comum tanto para este módulo como para a próxima etapa do nosso estudo. Aqui será demonstrado o problema, enquanto que no módulo seguinte, serão apresentadas duas possíveis soluções.

2.1- Interface IBuffer.java

```
package br.aiec.multithread.comum;

/**
 * Essa interface define as duas operações utilizadas para acesso ao monitor:
 *
```

```
 *  gravar que é utilizada pelo produtor;
 *
 *  ler que é utilizada pelo consumidor.
 *
 *  @author Guilherme Veloso
 *
 */
public interface IBuffer {

	public void gravar(int value);

	public int ler();
}
```

07

2.2- Classe Produtor.java

```
package br.aiec.multithread.comum;

import java.util.Random;

/**
 * Essa classe produz números de 1 a 10 que serão gravados no buffer
 * que representa a variável de memória compartilhada.
 *
 * Observe que a variável local somatório existe apenas para fins didáticos
 * e como uma tentativa, amadora e inicial, de medir a integridade no acesso ao
buffer.
 * Devido a sua forma simples, e suficiente para nossos propósitos, a mesma será
utilizada.
 *
 * Uma medição adequada extrapola o conteúdo desta disciplina e, por ora, da forma
 * em que se apresenta, a variável local somatório será considerada suficiente
para
 * os propósitos de demonstração.
 *
 *
 * @author Guilherme Veloso
 *
 */

public class Produtor implements Runnable {

	private IBuffer buffer;

	public Produtor(IBuffer buffer) {
		this.buffer = buffer;
	}

	@Override
	public void run() {
		int somatorio = 0;
```

```
			for(int i = 1; i < 11; i++){
				somatorio += i;
				buffer.gravar(i);
				esperaCronometrada();
			}

			System.out.println("Somatorio de valores GRAVADOS: " + somatorio);
		}



		/**
		* Esse método tem por finalidade potencializar a aleatoriedade do
		* escalonamento. Ou seja,
		* apenas tornar visível à percepção humana o procedimento de alternância de
		* threads/processos
		* (multiprogramação) em um computador.
		*/
		private void esperaCronometrada() {
			long aleatorio = (long)Math.random() * 5;
			try {
				Thread.sleep(aleatorio);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
}
```

08

2.3- Classe Consumidor.java

```
package br.aiec.multithread.comum;

/**
 * Essa classe lê os números de 1 a 10 (produzidos pelo produtor) que serão
 * obtidos
 * no acesso ao buffer que representa a variável de memória compartilhada.
 *
 * Observe que a variável local somatorio existe apenas para fins didáticos
 * e como uma tentativa, amadora e inicial, de medir a integridade no acesso ao
 * buffer.
 * Devido a sua forma simples, e suficiente para nossos própositos, a mesma será
 * utilizada.
 *
 * Uma medição adequada extrapola o conteúdo desta disciplina e, por ora, da forma
 * em que se apresenta, a variável local somatorio será considerada suficiente
para
 * os propósitos de demonstração.
 *
 *
 * @author Guilherme Veloso
```

```
 *
 */

public class Consumidor implements Runnable {

	private IBuffer buffer;

	public Consumidor(IBuffer buffer) {
		this.buffer = buffer;
	}

	@Override
	public void run() {
		int somatorio = 0;

		for(int i = 1; i < 11; i++){
			somatorio += buffer.ler();
			esperaCronometrada();
		}

		System.out.println("Somatorio de valores LIDOS: " + somatorio);
	}

	/**
	* Esse método tem por finalidade potencializar a aleatoriedade do
	* escalonamento. Ou seja,
	* apenas tornar visivel à percepção humana o procedimento de alternancia de
	* threads/processos
	* (multiprogramação) em um computador.
	*/
	private void esperaCronometrada() {
		long aleatorio = (long)Math.random() * 5;
		try {
			Thread.sleep(aleatorio);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}
```

09

3- Acesso não Sincronizado ao Buffer

Os referidos códigos apresentados neste item, utilizados em conjunto com os do item anterior, demonstram o acesso de duas threads (produtor e consumidor) ao buffer de memória que, neste exemplo, não oferece qualquer controle de sincronização no seu acesso. Deste modo, o objetivo é demonstrar o problema da **falta de integridade** no acesso ao buffer.

Oportunamente serão demonstradas duas possíveis soluções para o referido problema.

3.1- Classe BufferNaoSincronizado.java

```
package br.aiec.multithread.monitor;

import br.aiec.multithread.comum.IBuffer;


/**
 * Essa classe representa um esboço inicial de um monitor, e, portanto,
 *
 * ELA AINDA NÃO É SEGURA PARA MULTITHREAD!
 *
 * Essa classe não oferece qualquer mecanismo de sincronização para acesso
concorrente/paralelo
 * ao buffer, o que consequentemente, irá gerar problemas de falta de integridade
no buffer.
 *
 * As próximas versões já serão seguras e serão baseadas nessa estrutura inicial.
 *
 * @author Guilherme Veloso
 *
 */

public class BufferNaoSincronizado implements IBuffer {

	private int buffer;

	@Override
	public void gravar(int value) {
		System.out.println("GRAVANDO: " + value);
		this.buffer = value;

	}

	@Override
	public int ler() {
		int value = this.buffer;
		System.out.println("LENDO: " + value);
		return value;
	}
}
```

10

3.2- Classe PrincipalBufferNaoSincronizado.java

```
package br.aiec.multithread.teste;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import br.aiec.multithread.comum.Consumidor;
import br.aiec.multithread.comum.IBuffer;
```

```
import br.aiec.multithread.comum.Produtor;
import br.aiec.multithread.monitor.BufferNaoSincronizado;

/**
 *
 * Essa classe serve para testar o BufferNaoSincronizado demonstrando o problema
 * da falta de integridade dos dados, gravados e lidos, do buffer.
 *
 * @author Guilherme Veloso
 *
 */


public class PrincipalBufferNaoSincronizado {
	public static void main(String[] args) {

		IBuffer buffer = new BufferNaoSincronizado();

		Runnable produtor = new Produtor(buffer);
		Runnable consumidor = new Consumidor(buffer);

		ExecutorService pool = Executors.newCachedThreadPool();

		pool.execute(produtor);
		pool.execute(consumidor);

		pool.shutdown();
	}
}
```

11

Observe que após a execução do exemplo, o somatório de valores gravados e valores lidos, quase sempre, serão diferentes.

Somente não podemos afirmar isso com 100% de certeza porque existe uma possibilidade remotíssima de acontecer de o somatório dos valores gravados e lidos sejam iguais, tanto por conta da metodologia utilizada para verificação como também por uma questão de sorte. Ou seja, esses valores podem, mais em função da sorte que da metodologia propriamente dita, até apresentarem o mesmo resultado durante uma execução esporádica qualquer do programa.

Portanto se isso acontecer, valores gravados e lidos iguais, considere-se uma pessoa extremamente sortuda! A regra geral em 99,99% das vezes é de que os valores gravados e lidos serão diferentes.

Além disso, observe também que a ordem não sincronizada de gravação e leitura geram problemas de falta de integridade do buffer. Dentre esses **problemas** podemos citar:

- **a sobrescrita de valores no caso do produtor,** ou seja, o produtor sobrescreve o valor anteriormente gravado sem ao menos saber se o referido valor já foi lido;
- **a leitura suja de valores no caso do consumidor,** ou seja, o consumidor lê valores que não foram gravados (no caso desse exemplo, existe apenas um único possível valor, que é o zero) no buffer ou lê os mesmos valores mais de uma vez.

A solução para esse problema será demonstrada no próximo módulo.

12

RESUMO

O objetivo deste módulo foi apresentar alguns conceitos relativos a programação concorrente/paralela como:

- condição de corrida: quando a integridade dos dados não pode ser garantida devido ao uso compartilhado dos recursos;

-região crítica: conjunto de instruções compartilhadas;

- exclusão mútua: acesso exclusivo a uma thread por vez a região crítica.

Também foram apresentados três mecanismos possíveis de sincronização de processos/threads como:

- semáforo: uma variável inteira com a capacidade de contar que, dentre outras coisas, é utilizada para garantir a exclusão mútua;

- mutex: um semáforo simplificado que assume apenas dois estados: ligado e desligado;

- monitor: um tipo de dado abstrato com suas operações e dados que visa também garantir a exclusão mútua.

Todos esses conceitos foram apresentados por meio do problema de integridade relativo a falta de sincronização por meio de um exemplo clássico denominado de produtor/consumidor que acessam um buffer compartilhado de memória.

UNIDADE 2 – CONCEITOS BÁSICOS DE UM APLICATIVO MULTITHREADING

MÓDULO 4 – CONCEITOS BÁSICOS - THREADS EM JAVA (PARTE 3)

01

1 - PROBLEMA DO PRODUTOR-CONSUMIDOR (CONTINUAÇÃO)

Nesta etapa do nosso estudo serão demonstradas duas soluções para o problema do produtor-consumidor visto no módulo anterior. Neste referido problema é possível generalizá-lo para ***m* produtores**, ***n* consumidores** e **k posições de buffer**. Porém, somente consideraremos duas situações distintas com apenas um produtor e um consumidor:

- a primeira, o buffer terá **uma única posição de memória**, como no exemplo do módulo anterior, que demonstra o problema em si. Só que agora, o problema da falta de integridade não mais ocorrerá, pois as duas threads que acessam o buffer serão sincronizadas;
- a segunda, o **buffer terá um número inteiro fixo de posições de memória**. No nosso exemplo esse número será de cinco posições. A finalidade aqui, além da integridade, é conseguir melhorar a performance do programa, como será explicado logo a seguir.

Observe que o buffer sempre é de tamanho fixo, seja de uma única posição ou de cinco posições. Apesar de possuírem mais semelhanças que diferenças, podemos citar duas principais **diferenças** entre esses dois exemplos:

- a primeira diz respeito ao tipo de controle necessário: para o buffer de uma única posição será utilizado o mutex e para o buffer de cinco posições será utilizado o semáforo. Observe que o uso de mutex ou semáforo sempre será por meio de um monitor. **Não se esqueça de observar a ordem (bloqueio e sinais) das instruções nos métodos do monitor. Essa ordem é ESSENCIAL**;
- a segunda, e não menos importante, resume-se na quantidade de vezes que uma thread deve esperar pela outra (performance) no acesso ao buffer. É óbvio que inúmeras outras variáveis, de dimensões diversas, podem interferir nessa performance. Contudo, a análise aqui proposta, diz respeito apenas a quantidade de vezes que uma thread terá que interromper seu processamento em função do buffer estar cheio ou vazio. O objetivo é demonstrar, de modo mais simples possível, a solução para ambos os problemas, tanto do ponto de vista da integridade do buffer como também da performance de acesso.

02

2- BUFFER DE UMA ÚNICA POSIÇÃO

O problema se origina quando o produtor quer colocar um novo item no buffer, mas ele já está cheio (significa que o dado ainda não foi lido). Ou seja, o produtor não poderá gravar o dado no buffer, pois caso seja permitida a gravação do novo dado, aquele anteriormente gravado será sobrescrito e nunca mais poderá ser lido pelo consumidor.

A solução é pôr o produtor para dormir e somente despertá-lo quando o consumidor realizar a leitura do item presente no buffer.

De modo análogo, se o consumidor quiser ler um item do buffer e o mesmo estiver vazio, ele não poderá prosseguir, pois corre o risco de ler um "lixo" de memória (valor inválido ou já lido anteriormente).

Deste modo, o consumidor deve ser posto para dormir até que o produtor escreva novamente no buffer e possa despertá-lo.

Veja a seguir o código de exemplo.

Vazio

Isso significa que ou o dado não existe ainda ou o dado já foi lido.

03

2.1- Classe BufferSincronizado.java

```
package br.aiec.multithread.monitor;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import br.aiec.multithread.comum.IBuffer;

/**
 * Essa classe representa um monitor de fato, e, portanto,
 *
 * ELA É SEGURA PARA MULTITHREAD!
 *
 * Essa classe oferece um mecanismo de sincronização para acesso
 * concorrente/paralelo ao buffer, o que consequentemente, irá garantir a
 * integridade no buffer.
 *
 *
 * @author Guilherme Veloso
 *
 */

public class BufferSincronizado implements IBuffer {

      // Variáveis utilizadas garantir o acesso exclusivo
```

```
	private Lock bloqueio;
	private boolean cheio; // essa variável do tipo boolean representa o mutex

	// Variáveis utilizadas para guardar os sinais e
	// permitir transitar as threads entre os estados de executável
	// e espera e vice-versa
	private Condition podeGravar;
	private Condition podeLer;

	// Variável compartilhada
	private int buffer;

	public BufferSincronizado() {
		bloqueio = new ReentrantLock();
		podeGravar = bloqueio.newCondition();
		podeLer = bloqueio.newCondition();

		// Essas duas inicializações abaixo são desnecessárias, pois o Java
já
		// faria isso!
		// Foram escritas apenas para tornar o código mais claro!
		cheio = false;
		buffer = 0;
	}

	@Override
	public void gravar(int value) {

		try {
			bloqueio.lock();

			while (cheio) {
				System.out.println("Produtor esperando... Buffer
cheio!");
				podeGravar.await();
			}

			System.out.println("GRAVANDO: " + value);

			this.buffer = value;

			this.cheio = true;

			podeLer.signal();

		} catch (InterruptedException e) {
			e.printStackTrace();
		} finally {
			bloqueio.unlock();
		}
	}

	@Override
	public int ler() {
```

```
		int value = 0;

		try {
			bloqueio.lock();

			while (!cheio) {
				System.out.println("Consumidor esperando... Buffer
vazio!");
				podeLer.await();
			}

			System.out.println("LENDO: " + this.buffer);

			value = this.buffer;

			this.cheio = false;

			podeGravar.signal();

		} catch (InterruptedException e) {
			e.printStackTrace();
		} finally {
			bloqueio.unlock();
		}

		return value;
	}
}
```

04

2.2- Classe PrincipalBufferSincronizado.java

```
package br.aiec.multithread.teste;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import br.aiec.multithread.comum.Consumidor;
import br.aiec.multithread.comum.IBuffer;
import br.aiec.multithread.comum.Produtor;
import br.aiec.multithread.monitor.BufferSincronizado;

/**
 *
 * Essa classe serve para testar o BufferSincronizado demonstrando que o buffer
sempre
 * será mantido integro não importando a forma como as threads o acessam.
 *
 * @author Guilherme Veloso
 *
 */
```

```
public class PrincipalBufferSincronizado {
	public static void main(String[] args) {

		IBuffer buffer = new BufferSincronizado();

		Runnable produtor = new Produtor(buffer);
		Runnable consumidor = new Consumidor(buffer);

		ExecutorService pool = Executors.newCachedThreadPool();

		pool.execute(produtor);
		pool.execute(consumidor);

		pool.shutdown();
	}
}
```

Observe que, em relação ao exemplo do módulo anterior, agora os valores gravados e lidos **sempre terão o mesmo valor**, ou seja, para o exemplo presente no código, o resultado da somatória dos valores gravados e lidos sempre será 55, em quaisquer circunstâncias que se apresentem. Portanto, o exemplo apresentado resolve o problema da falta de integridade do buffer.

05

Caso ainda tenha dúvidas sobre a eficiência e eficácia da solução, altere os códigos do produtor e consumidor de modo que os laços de repetição girem um número bem maior de vezes, por exemplo, 9000 vezes.

Lembre-se de que tal alteração deve ser feita simetricamente no produtor e no consumidor ao mesmo tempo. Alterar apenas um deles irá descaracterizar o exemplo e consequentemente seus resultados! Execute novamente o programa e observe os resultados. Certamente continuarão íntegros!

Contudo, observe também que a solução ora apresentada possui um **problema de performance**. Como dito anteriormente, a performance será avaliada apenas sob o aspecto da quantidade de vezes que uma thread produtora tem que esperar pela consumidora e vice-versa. Quaisquer outras variáveis ou dimensões estão sendo desconsideradas, por ora.

No caso de uma única posição no buffer, ambos, produtor e consumidor, não conseguem realizar acessos consecutivos ao buffer. Isso significa que o produtor não pode gravar uma vez e, na sequência (consecutivamente), gravar de novo no mesmo buffer, pois o mesmo sempre estará cheio. Isso é uma certeza!

Para o consumidor também não é possível ler uma vez, e na sequência, ler novamente, pois o buffer estará sempre vazio. Deste modo, ambas as threads não conseguem realizar mais de um único acesso consecutivo ao buffer. Isso se deve ao fato de o buffer ter apenas uma única posição disponível e a

solução óbvia é aumentar o número de posições para que mais acessos consecutivos possam ser realizados por ambas as threads, como será demonstrado no próximo item.

06

3- BUFFER DE CINCO POSIÇÕES (BUFFER CIRCULAR)

A solução para o problema de um único acesso por vez pode ser minimizado por meio do aumento da quantidade de posições do buffer. Essa técnica é muito utilizada por threads que compartilham recursos e operam nas mesmas velocidades médias.

E importante lembrar que a integridade do buffer deve e continuará a ser mantida. Não se pode admitir perdê-la, nunca! Resolver esse problema relacionado à performance sob o aspecto específico do qual estamos tratando, envolve aumentar a quantidade de posições do buffer para, por exemplo, cinco posições.

Observe que o buffer continuará de tamanho fixo, contudo agora ele terá cinco posições disponíveis. Esse buffer não será dinâmico (tamanho variável), apesar de também ser uma solução possível. Portanto, essa solução que iremos demonstrar aqui, de um buffer de cinco posições fixas, é denominada de Buffer Circular.

Obviamente, cada uma das threads (produtor ou consumidor) poderá realizar, no máximo, cinco acessos consecutivos antes de ter que parar/esperar/interromper o seu processamento e aguardar pela outra thread. Deste modo iremos aumentar efetivamente e significativamente o tempo em que cada thread permanece no estado de executável melhorando o *turnaround* do programa como um todo. A seguir, veja o código e exemplo.

07

3.1- Classe BufferCircular.java

```
package br.aiec.multithread.monitor;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import br.aiec.multithread.comum.IBuffer;

/**
 * Essa classe representa um monitor de fato, e, portanto,
 *
 * ELA É SEGURA PARA MULTITHREAD!
 *
```

```
 * Essa classe oferece um mecanismo de sincronização para acesso
 * concorrente/paralelo ao buffer circular, o que consequentemente, irá garantir a
 * integridade no buffer.
 *
 *
 * @author Guilherme Veloso
 *
 */

public class BufferCircular implements IBuffer {

	/**
	 *  Variáveis utilizadas garantir o acesso exclusivo
	 */
	private Lock bloqueio;
	private int w; // semáforo exclusivo do produtor (indice de gravação)
	private int r; // semáforo exclusivo do consumidor (indice de leitura)
	private int quantidade; // semáforo de uso compartilhado pelo produtor e
consumidor

	/**
	 *  Variáveis utilizadas para guardar os sinais e
	 *  permitir transitar as threads entre os estados de executável -> espera e
vice-versa
	 */
	private Condition podeGravar;
	private Condition podeLer;

	// Variável compartilhada
	private int buffer[];

	public BufferCircular() {
		bloqueio = new ReentrantLock();
		podeGravar = bloqueio.newCondition();
		podeLer = bloqueio.newCondition();
		buffer = new int[5];

		// Essas três inicializações abaixo são desnecessárias, pois o Java
já
		// faria isso! Foram escritas apenas para tornar o código mais claro!
		w = 0;
		r = 0;
		quantidade = 0;
	}

	@Override
	public void gravar(int value) {

		try {
			bloqueio.lock();

			while (quantidade == buffer.length) {
				System.out.println("Produtor esperando... Buffer
cheio!");
```

```
                        podeGravar.await();
                    }

                    System.out.println("GRAVANDO - BUFFER["+ w + "]: " + value);

                    this.buffer[w] = value;

                    this.quantidade++;

                    w = (w + 1) % buffer.length;

                    podeLer.signal();

            } catch (InterruptedException e) {
                    e.printStackTrace();
            } finally {
                    bloqueio.unlock();
            }
      }
      @Override
      public int ler() {
            int value = 0;
            try {
                    bloqueio.lock();

                    while (quantidade == 0) {
                          System.out.println("Consumidor esperando... Buffer
vazio!");
                          podeLer.await();
                    }

                    System.out.println("LENDO - BUFFER["+ r + "]: " +
this.buffer[r]);

                    value = this.buffer[r];
                    this.quantidade--;
                    r = (r + 1) % buffer.length;
                    podeGravar.signal();

            } catch (InterruptedException e) {
                    e.printStackTrace();
            } finally {
                    bloqueio.unlock();
            }

            return value;
      }
}
```

08

3.2- Classe PrincipalBufferCircular.java

```
package br.aiec.multithread.teste;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import br.aiec.multithread.comum.Consumidor;
import br.aiec.multithread.comum.IBuffer;
import br.aiec.multithread.comum.Produtor;
import br.aiec.multithread.monitor.BufferCircular;

/**
 *
 * Essa classe serve para testar o BufferCircular demonstrando que o buffer sempre
 * será mantido integro não importando a forma como as threads o acessam.
 *
 * @author Guilherme Veloso
 *
 */

public class PrincipalBufferCircular {
	public static void main(String[] args) {

		IBuffer buffer = new BufferCircular();

		Runnable produtor = new Produtor(buffer);
		Runnable consumidor = new Consumidor(buffer);

		ExecutorService pool = Executors.newCachedThreadPool();

		pool.execute(produtor);
		pool.execute(consumidor);

		pool.shutdown();
	}
}
```

09

Observe que agora cada thread, produtora ou consumidora, poderá realizar, no máximo, cinco acessos consecutivos antes de ter que esperar pela outra. Além da integridade do buffer que continua sendo preservada como no exemplo do item anterior, agora o acesso ao buffer foi otimizado, melhorando o *turnaround* da execução do programa.

Um ponto importante a ser observado é que a quantidade de posições que o buffer deve ter é algo de resposta complexa. Para fins didáticos, foi escolhido o número 5 sem qualquer tipo de critério, uma vez que o objetivo foi de apenas demonstrar como funciona um buffer circular. Contudo, em aplicações comerciais e de produção esse valor deve ser configurado adequadamente para cada ambiente diferente onde o *software* será executado.

Deste modo, para determinar um número ideal de posições do buffer, ou seja, o número necessário, sem faltar e nem sobrar posições, é necessário um estudo à parte das conveniências, ameaças, necessidades e oportunidades de toda a estrutura computacional e organizacional onde o *software* será executado. Obviamente, isso está além do escopo desta disciplina e por este motivo não será tratado aqui.

Um outro ponto importante a ser lembrado é que a linguagem de programação Java oferece outras formas de se implementar tais exemplos. Além disso, a API da linguagem oferece também um conjunto de classes já prontas com suporte à programação concorrente/paralela. Para os exemplos deste módulo, o uso da classe ArrayBlockingQueue também simplificaria os exemplos demonstrados.

Portanto, caso não se sinta confortável em utilizar os recursos da forma como foram apresentados, opte pelo uso das estruturas já presentes na linguagem e tenha em mente que, se está simples de usar, é porque alguém tornou isso simples e esse alguém está fazendo o trabalho por você.

A partir dos módulos seguintes, o framework Android será apresentado. Um detalhe importante que deve ser considerado é que apesar da linguagem Java poder ser utilizada para a programação de aplicativos Android, a API de desenvolvimento do Android é uma API própria, que possui muitas semelhanças com a API do Java. Deste modo, o Java utilizado pelo Android não necessita da Java Standard Edition (JSE) muito utilizado por diversos aplicativos “comuns” escritos em Java.

Outras formas

Por exemplo, o uso dos métodos **wait, notity e notityAll** da classe **Object** conjuntamente com a palavra reservada **synchronized** podem simplificar a implementação mostrada neste módulo.

10

RESUMO

O objetivo deste módulo foi apresentar duas soluções possíveis para o problema de integridade do produtor-consumidor que acessam um buffer de memória compartilhada. A primeira solução utiliza-se de um monitor que encapsula um mutex para o buffer compartilhado enquanto que a segunda solução utiliza-se de um monitor que encapsula três semáforos para o buffer compartilhado.