CONCORRÊNCIA EM JAVA

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

Escola de Artes, Ciências e Humanidades | EACH | USP

POR QUE ESCREVER PROGRAMAS CONCORRENTES?

Desempenho para explorar arquiteturas paralelas com memória compartilhada (SMP, hyperthreaded, multi-core, NUMA, etc.)

Modelagem para descrever o paralelismo natural de algumas aplicações (tarefas independentes, sobreposição de operações de E/S, etc.).

PROCESSOS

- Cada processo tem seu próprio espaço de endereçamento:
  - *segmento de texto*: contém o código executável
  - *segmento de dados*: contém as variáveis globais
  - *pilha*: contém dados temporários (variáveis locais, endereços de retorno, etc.)
- O contador de programa e os registradores da CPU fazem parte do contexto do programa
- O contexto do programa é o conjunto mínimo de dados usados por um processo e deve ser gravado quando um processo é interrompido e relido quando um processo é retomado

THREADS

- Permitem múltiplas atividades independentes dentro de um único processo
- *Threads* de um mesmo processo compartilham:
  - Todo o espaço de endereçamento, exceto a pilha, os registradores e o contador de programa
  - Arquivos abertos
  - Outros recursos
- Também são chamados de *processos leves*, pois a criação das *threads* e a troca de contexto são mais rápidas
- Permitem um alto grau de cooperação entre as atividades!

TODO PROGRAMA EM JAVA EXECUTA VÁRIAS THREADS

- Java foi a primeira linguagem de programação a incorporar *threads* desde a concepção da linguagem
- Se você já programou em Java, já escreveu um programa multithreaded:
  - A máquina virtual mantém várias *threads*: *thread* Main, *threads* de coletor de lixo, *threads* de finalização de objetos, etc.

OBJETOS DO TIPO THREAD

Cada *thread* é associada a uma instância da classe `java.lang.Thread`. Há duas estratégias básicas para usar esses objetos para criar aplicações concorrentes:

- controlar e gerenciar manualmente instâncias de `Thread`
- abstrair o gerenciamento de *threads* do resto da aplicação utilizando um *executor*

Uma aplicação que cria uma instância de `Thread` pode definir o código a ser executado concorrentemente de duas formas:

- usando herança, criamos uma nova classe que estende `Thread`
- escrevendo uma classe que implementa a interface `Runnable`

Uma classe que estende `Thread` deve sobrescrever o método `public void run()`:

```
class Tarefa1 extends Thread {
    public void run() {
        for(int i=0; i<1000; i++) {
            System.out.println("Usando herança");
        }
    }

    public static void main(String args[]) {
        (new Tarefa1()).start();
    }
}
```

USANDO A INTERFACE `runnable`

A interface `Runnable` nos obriga a implementar o método `public void run()`:

```
class Tarefa2 implements Runnable {
    public void run() {
        for(int i=0; i<1000; i++) {
            System.out.println("Usando Runnable");
        }
    }

    public static void main(String args[]) {
        (new Thread(new Tarefa2())).start();
    }
}
```

THREAD.SLEEP()

`Thread.sleep()` suspende uma thread por um período de tempo determinado. Útil para permitir que outras *threads* possam utilizar a CPU.

```
public class SleepMessages {
    public static void main(String args[])
        throws InterruptedException {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };

        for (int i = 0; i < importantInfo.length; i++) {
            //Pause for 4 seconds
            Thread.sleep(4000);
            //Print a message
            System.out.println(importantInfo[i]);
        }
    }
}
```

THREAD.INTERRUPT()

`Thread.interrupt()` avisa uma *thread* que ela deve interromper o que está fazendo (por exemplo, terminar a execução). Uma *thread* que permite ser interrompida deve verificar periodicamente a flag de interrupção usando o método `Thread.interrupted()`. O modo mais comum de responder a uma interrupção é lançando uma `InterruptedException`.

```
for (int i = 0; i < importantInfo.length; i++) {
    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message
    System.out.println(importantInfo[i]);
}
```

THREAD.INTERRUPT()

`Thread.interrupt()` avisa uma *thread* que ela deve interromper o que está fazendo (por exemplo, terminar a execução). Uma *thread* que permite ser interrompida deve verificar periodicamente a flag de interrupção usando o método `Thread.interrupted()`. O modo mais comum de responder a uma interrupção é lançando uma `InterruptedException`.

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // We've been interrupted: no more crunching.
        return;
    }
}
```

THREAD.INTERRUPT()

`Thread.interrupt()` avisa uma *thread* que ela deve interromper o que está fazendo (por exemplo, terminar a execução). Uma *thread* que permite ser interrompida deve verificar periodicamente a flag de interrupção usando o método `Thread.interrupted()`. O modo mais comum de responder a uma interrupção é lançando uma `InterruptedException`.

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        throw new InterruptedException();
    }
}
```

THREAD.JOIN()

`Thread.join()` faz a *thread* atual esperar que uma outra *thread* termine sua execução

```
public class SimpleThreads {
    // Display a message, preceded by the name of the current thread
    static void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s%n", threadName, message);
    }

    private static class MessageLoop
        implements Runnable {
        public void run() {
            String importantInfo[] = {"Mares eat oats", "Does eat oats",
                                      "Little lambs eat ivy", "A kid will eat ivy too"};
            try {
                for (int i = 0; i < importantInfo.length; i++) {
                    Thread.sleep(4000); // Pause for 4 seconds
                    threadMessage(importantInfo[i]); // Print a message
                }
            } catch (InterruptedException e) {
                threadMessage("I wasn't done!");
            }
        }
    }
```

THREAD.JOIN() (II)

```
    public static void main(String args[]) throws InterruptedException {
        // Delay, in milliseconds before we interrupt MessageLoop
        long patience = 1000 * 60 * 60; // 1 per hour

        threadMessage("Starting MessageLoop thread");
        long startTime = System.currentTimeMillis();
        Thread t = new Thread(new MessageLoop());
        t.start();

        threadMessage("Waiting for MessageLoop thread to finish");
        // loop until MessageLoop thread exits
        while (t.isAlive()) {
            threadMessage("Still waiting...");
            // Wait maximum of 1 second for MessageLoop thread to finish.
            t.join(1000);
            if (((System.currentTimeMillis() - startTime) > patience)
                  && t.isAlive()) {
                threadMessage("Tired of waiting!");
                t.interrupt();
                // Shouldn't be long now
                // -- wait indefinitely
                t.join();
            }
        }
        threadMessage("Finally!");
    }
}
```

Threads compartilham o mesmo espaço de endereçamento de memória!

Suponha que duas *threads* A e B compartilhem um contador.

```
int contador = 0;
```

A incrementa o contador:

```
contador++;
```

Logo em seguida, B imprime o contador:

```
System.out.println(contador);
```

Qual a saída na tela?

- Se A e B fossem a mesma *thread*, certamente 1
- Mas a saída pode ser 0 ou 1; as mudanças em uma *thread* podem não ser visíveis para a outra. Sincronização estabelece uma relação de ordem entre as ações! (*happens-before relationship*)

SINCRONIZAÇÃO

A linguagem Java provê duas formas de sincronização básicas:

- métodos sincronizados
- expressões sincronizadas

A linguagem Java provê duas formas de sincronização básicas:

- métodos sincronizados
- expressões sincronizadas

```
public class ContadorSincronizado {
    private int contador = 0;

    public synchronized void incrementa() {
        contador++;
    }

    public synchronized int valor() {
        return contador;
    }
}
```

A linguagem Java provê duas formas de sincronização básicas:

- métodos sincronizados
- expressões sincronizadas

```
public void incrementa() {
    synchronized(this) {
        contador++;
    }
}
```

As sincronizações são reentrantes
Uma *thread* não pode pegar o *lock* de uma outra *thread*, mas pode usar um mesmo *lock* já adquirido mais de uma vez.

- Todo objeto tem um monitor (também chamado de *lock* intrínseco) associado a si
- Quando uma *thread* invoca um método sincronizado, ele adquire (e depois libera) automaticamente o monitor para aquele objeto
  - se o método for `static`, então é usado o monitor do objeto `Class` associado à classe

```
public class MsLunch {
    private long c1 = 0; private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
```

ACESSO ATÔMICO

- Uma ação atômica é aquela que ocorre toda de uma única vez, não pode parar no meio
- Operações que parecem simples podem não ser atômicas, como pode ser o caso com a expressão `i++`
- Mas você pode especificar ações atômicas:
  - leituras e escritas são atômicas para variáveis de referência e pra maior parte das variáveis primitivas (todas menos `long` e `double`)
  - leituras e escritas em todas as variáveis declaradas como `volatile` (incluindo `long` e `double`)

Situação onde duas ou mais *threads* ficam bloqueadas para sempre porque uma fica esperando a outra:

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                + "  has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s"
                + " has bowed back to me!%n",
                this.name, bower.getName());
        }
    }
    public static void main(String[] args) {
        final Friend alphonse =
            new Friend("Alphonse");
        final Friend gaston =
            new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

É extremamente provável que ambas as *threads* fiquem bloqueadas. Ambas ficarão esperando (para sempre) até que uma das *threads* saia de **bow**

STARVATION E LIVELOCK

Outros dois problemas comuns são:

Starvation (ou inanição) é a situação onde uma *thread* nunca consegue acesso ao recurso compartilhado de que precisa

Livelock caso particular de *starvation* onde *threads* reagem às mudanças uma das outras, sem que nenhuma consiga avançar (ex: imagine duas pessoas num corredor, uma de frente pra outra, que sempre se movem pro mesmo lado na esperança que o outro possa passar)

WAIT(), NOTIFY() E NOTIFYALL()

Estes métodos da classe `Object` implementam o conceito de monitores sem utilizar espera ativa. Ao invés disso, notificam as *threads* indicando se estas devem ser suspensas ou se devem voltar a ficar em execução.

O *lock* do objeto chamado pela *thread* para realizar as notificações será utilizado. Por isso, antes de chamar um dos três métodos, o *lock* deve ser obtido utilizando-se o comando `synchronized`.

wait(), notify() e notifyAll()

`Object.wait()` suspende a thread que chamou o método até que outra thread a acorde ou até que o tempo especificado como argumento tenha passado

`Object.notify()` acorda, se existir, alguma thread que esteja esperando um evento neste objeto

`Object.notifyAll()` acorda todas as threads que estejam esperando neste objeto

wait()

```
synchronized (obj) {
  while (<condição não for satisfeita>)
    obj.wait(timeout);
    ... // Realiza ações que
        // assumem a condição satisfeita
}
```

notify()

```
synchronized (obj) {
  condição = true
  obj.notify()
}
```

Uma expressão idiomática comum é usar a construção de *guarded block* (bloco protegido) para coordenar ações entre *threads*

```
// Thread A
public synchronized void guardedJoy() {
    // This guard only loops once for each special event, which may not
    // be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}

// Thread B
public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}
```

Note que:
Antes de antes de invocar o método `wait()`, a *thread* precisa obter o monitor do objeto (nesse exemplo, obtido automaticamente pelo método `synchronized`)

REFERÊNCIAS

- The Java Tutorials: Concurrency: `http://docs.oracle.com/javase/tutorial/essential/concurrency/`