

Programação Concorrente

Abstração da Programação Concorrente

aula-02

Agenda

- O papel da abstração.
- Execução concorrente como entrelaçamento de estados atômicos.
- Justificativa da abstração.
- Entrelaçamento arbitrário (interleaving).
- Corretude.
- Fairness (Justiça).

Abstrações

- Conceito
 - Abstrair é entender como funciona, em alto nível, sem se importar com os detalhes...
- Níveis de abstração
 - Sistemas e bibliotecas (API)
 - Linguagens de programação
 - Conjunto de instruções

Execução concorrente (interleaving)

- “Um programa concorrente consiste em um conjunto **finito** de processos, os quais são escritos usando um conjunto **finito** de declarações **atômicas**. A execução de um programa concorrente consiste na execução das declarações atômicas de forma **entrelaçada**.” -Ben Ari
- Um computação é então o resultado gerado pelo entrelaçamento dessas declarações.

Execução Concorrente (motivações)

- Programação Moderna
 - Web sites devem ser capazes de gerenciar múltiplos usuários
 - Aplicações mobile devem fazer parte do seu processamento em um servidor (“na cloud”)
 - GUI efetuam trabalhos em background que não devem perturbar o usuário. Por exemplo, o Eclipse compilar o seu código enquanto você edita.
- *“Being able to program with concurrency will still be important in the future. Processor clock speeds are no longer increasing. Instead, we’re getting more cores with each new generation of chips. So in the future, in order to get a computation to run faster, we’ll have to split up a computation into concurrent pieces.”*

Execução concorrente

- Durante a computação, o ponteiro de controle indica qual a instrução será executada por um determinado processo. Cada **processo (p)** tem seu **ponteiro de controle (cp)**.

p
p1
p2

q
q1
q2

Suponha dois processos p e q:

- **Instruções de p: p1 e p2**
- **Instruções de q: q1 e q2**

Quais são as possíveis computações (também chamadas de cenários)?

Execução concorrente

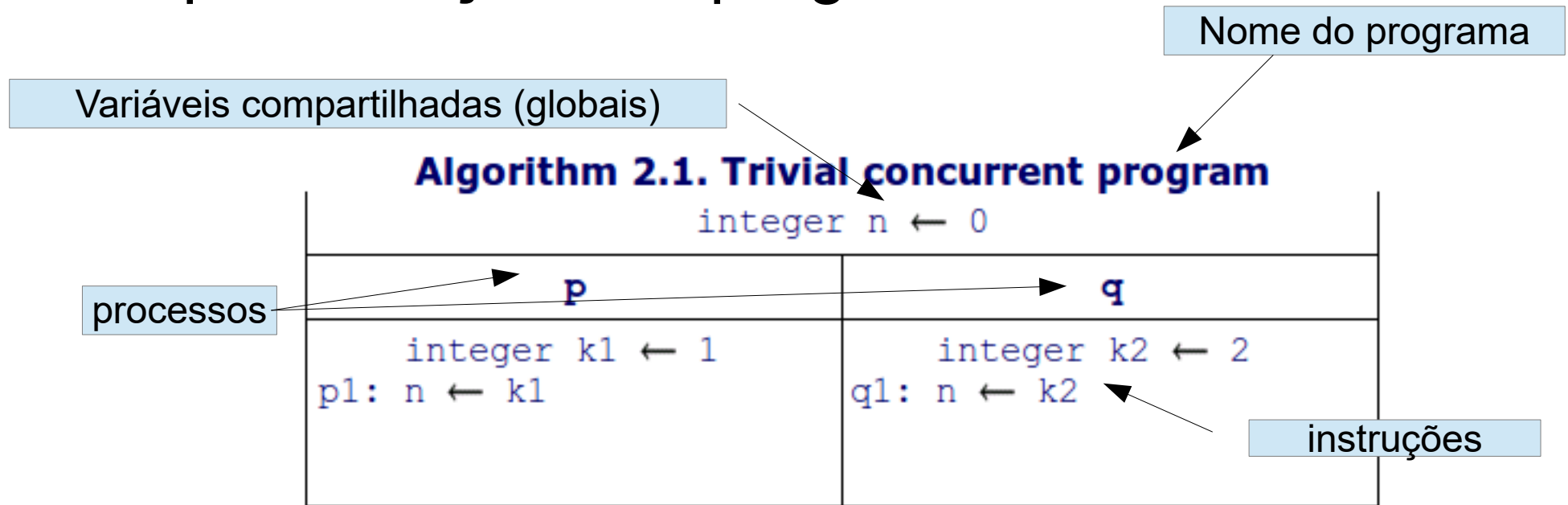
- Possíveis computações entrelaçadas.

- $p1 \rightarrow q1 \rightarrow p2 \rightarrow q2$
- $p1 \rightarrow p2 \rightarrow q1 \rightarrow q2$
- $q1 \rightarrow p1 \rightarrow q2 \rightarrow p2$
- $q1 \rightarrow q2 \rightarrow p1 \rightarrow p2$
- $p1 \rightarrow q1 \rightarrow q2 \rightarrow p2$
- $q1 \rightarrow p1 \rightarrow p2 \rightarrow q2$

Quais cenários não são possíveis?

Execução concorrente

- Representação dos programas



Execução concorrente


- Representação dos programas
 - O que é atômico?
 - O que pode ser divisível (não atômico)?

Execução concorrente

- Estados e transições
 - A execução de um programa é definida por seus **estados e transições** entre eles.
 - Seja o seguinte programa sequencial:

Algorithm 2.2. Trivial sequential program

```
integer n ← 0  
  
integer k1 ← 1  
integer k2 ← 2  
p1: n ← k1  
p2: n ← k2
```

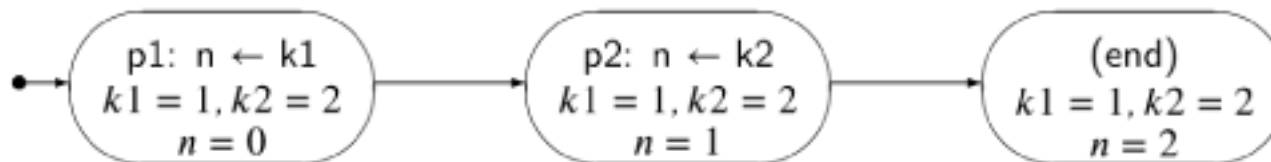


Execução concorrente

- Estados e transições

Algorithm 2.2. Trivial sequential program

```
integer n ← 0  
integer k1 ← 1  
integer k2 ← 2  
p1: n ← k1  
p2: n ← k2
```



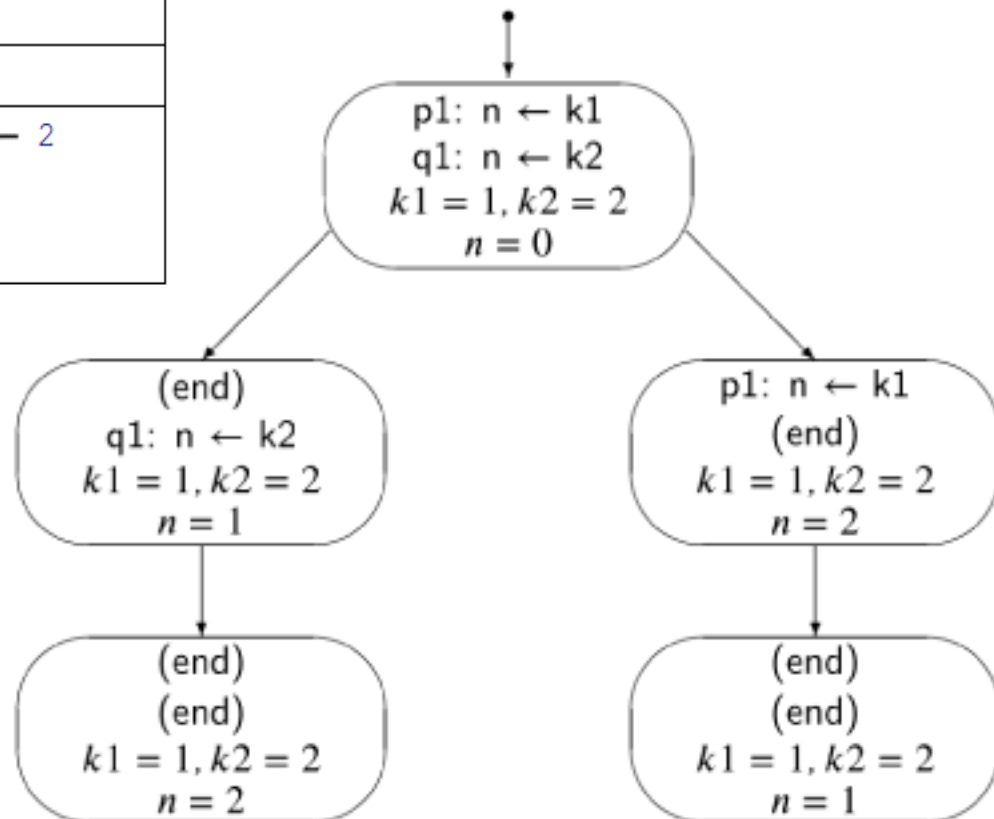
Execução concorrente

- Estados e transições

Algorithm 2.1. Trivial concurrent program

integer $n \leftarrow 0$	
p	q
integer $k1 \leftarrow 1$ $p1: n \leftarrow k1$	integer $k2 \leftarrow 2$ $q1: n \leftarrow k2$

p1->q1
q1->p1



Execução concorrente

- Estados e transições
 - Estados: é definido por uma *tupla* que consiste de um única instrução de um determinado processo e a definição das variáveis locais e globais aos processos.
 - Transição: sejam os estados s_1 e s_2 . Um transição é uma execução de um instrução que muda o estado do programa de s_1 para s_2 .
 - Diagrama de estado: pode ser visto como um *grafo* de estados em um programa concorrente.
 - Cenário: um cenário é um caminho nesse grafo até chegar em um estado final.

Execução concorrente

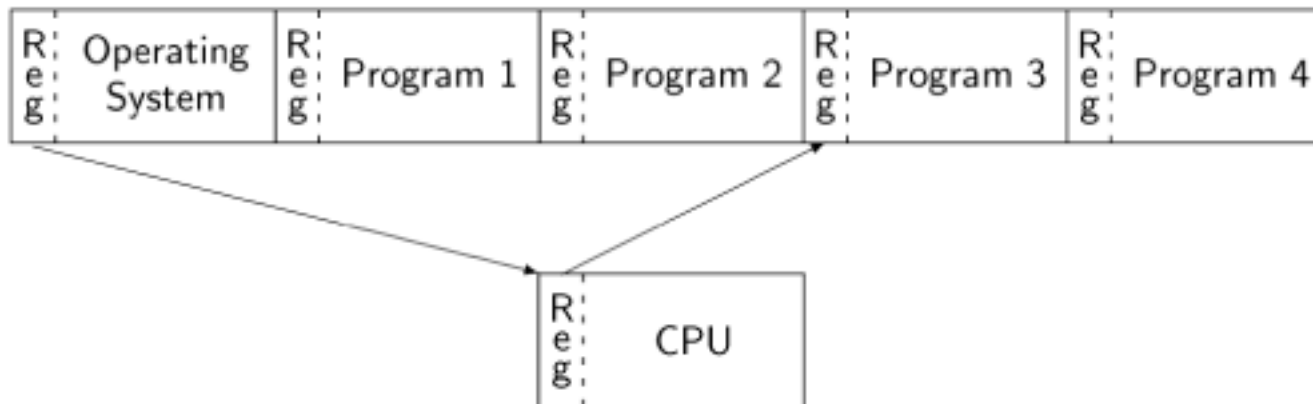
- Cenários

Process p	Process q	n	k1	k2
p1: n ← k1	q1: n ← k2	0	1	2
(end)	q1: n ← k2	1	1	2
(end)	(end)	2	1	2

Justificativa

- Sistemas multitarefas

The diagram below shows the memory divided into five segments, one for the operating system code and data, and four for the code and data of the programs that are running concurrently.

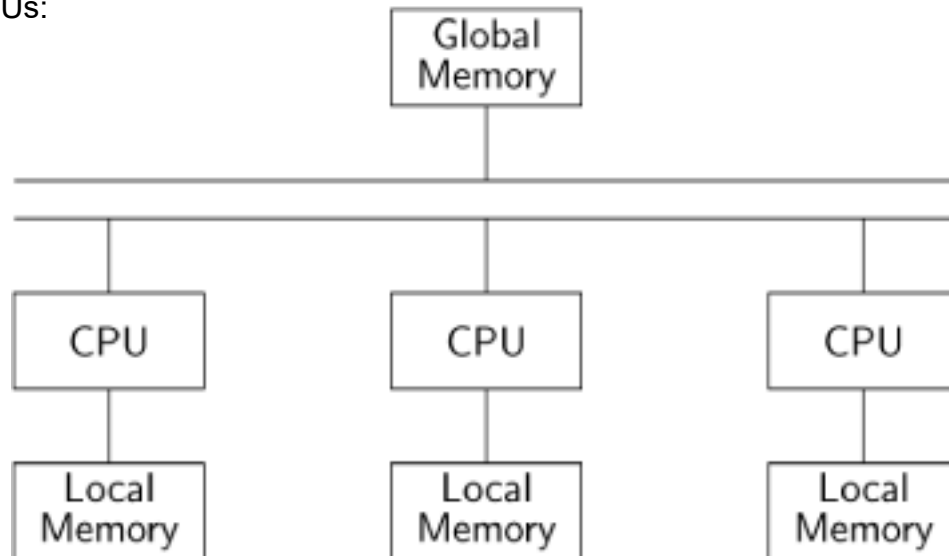


- Quando a execução de um programa é interrompida, os registradores da CPU são salvos em uma área específica do programa.
- Depois, o conteúdo do registro responsável pela interrupção é carregado na memória.
- Terminada a interrupção, o processo inverso acontece, gerenciado pelo scheduler que pode decidir carregar o estado de outro programa e não o que foi interrompido previamente.

Justificativa

- Computadores com multiprocessadores

A multiprocessor computer is a computer with more than one CPU. The memory is physically divided into banks of local memory, each of which can be accessed only by one CPU, and global memory, which can be accessed by all CPUs:

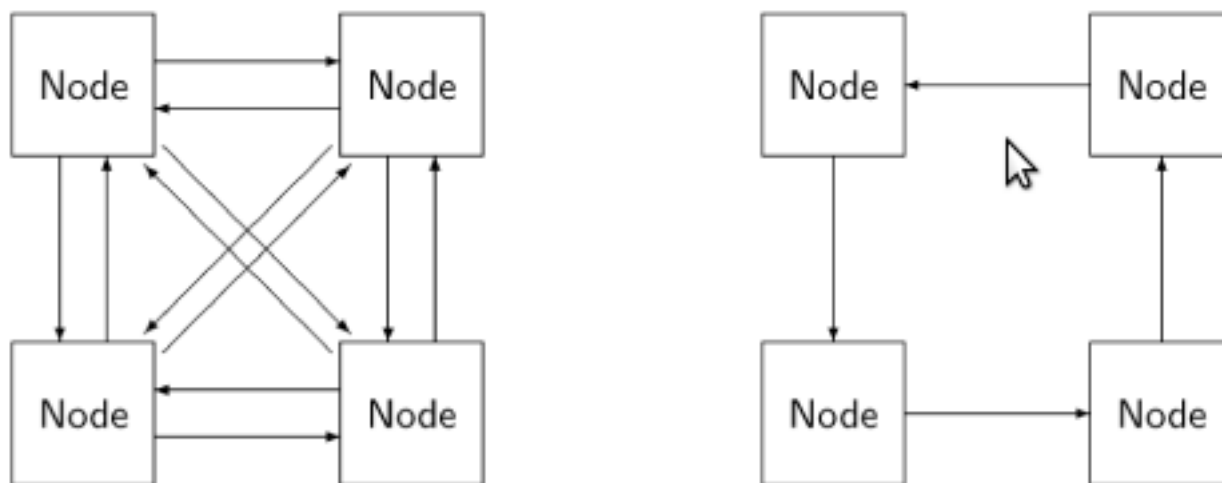


- Caso tenhamos um número suficiente de CPU, o entrelaçamento não existe mais de fato.
- O problema acontece quando duas ou mais CPUs querem ler/escrever na memória global (situação de contenção).

Justificativa

- **Sistemas distribuídos**

A distributed system is composed of several computers that have no global resources; instead, they are connected by communications channels enabling them to send messages to each other. The language of graph theory is used in discussing distributed systems; each computer is a node and the nodes are connected by (directed) edges.



- A abstração do entrelaçamento é falsa.
- Vê apenas eventos discretos: ou enviar ou recebe uma mensagem ou faz uma computação local.
- Sistema mais robusto.

Estados atômicos

Algorithm 2.3. Atomic assignment statements

integer $n \leftarrow 0$	
p	q
$p1: n \leftarrow n + 1$	$q1: n \leftarrow n + 1$

Process p	Process q	n
p1: $n \leftarrow n+1$	$q1: n \leftarrow n+1$	0
(end)	q1: $n \leftarrow n+1$	1
(end)	(end)	2

Process p	Process q	n
$p1: n \leftarrow n+1$	q1: $n \leftarrow n+1$	0
p1: $n \leftarrow n+1$	(end)	1
(end)	(end)	2

O estado final será sempre com $n=2$

Estados atômicos

Algorithm 2.4. Assignment statements with one global reference

integer $n \leftarrow 0$	
p	q
integer temp p1: temp $\leftarrow n$ p2: $n \leftarrow temp + 1$	integer temp q1: temp $\leftarrow n$ q2: $n \leftarrow temp + 1$

Process p	Process q	n	p.temp	q.temp
p1: temp $\leftarrow n$	q1: temp $\leftarrow n$	0	?	?
p2: $n \leftarrow temp + 1$	q1: temp $\leftarrow n$	0	0	?
(end)	q1: temp $\leftarrow n$	1		?
(end)	q2: $n \leftarrow temp + 1$	1		1
(end)	(end)	2		

Pós condição de $n=2$ aceita!
 Mas será que podemos garantir
 que sempre será assim?

Estados atômicos

Algorithm 2.4. Assignment statements with one global reference

integer n \leftarrow 0	
p	q
integer temp p1: temp \leftarrow n p2: n \leftarrow temp + 1	integer temp q1: temp \leftarrow n q2: n \leftarrow temp + 1

Process p	Process q	n	p.temp	q.temp
p1: temp \leftarrow n	q1: temp \leftarrow n	0	?	?
p2: n \leftarrow temp+1	q1: temp \leftarrow n	0	0	?
p2: n \leftarrow temp+1	q2: n \leftarrow temp+1	0	0	0
(end)	q2: n \leftarrow temp+1	1		0
(end)	(end)	1		

Pós condição de n=2 não aceita!

A corretude de um programa concorrente depende da especificação de seus estados Atômicos.

Será que esses exemplos são realistas, Levando em consideração o compilador?

Corretude

- Em programas sequenciais, a execução de seu código n vezes não irá interferir no resultado. O cenário é sempre o **mesmo**!
- Em programas **concorrentes**, alguns cenários podem dar uma saída correta e outros não (pós condição esperada). A saída depende do **entrelaçamento** dos processos envolvidos.
- Em programação concorrentes, estamos interessados em programas onde acontece o entrelaçamento.

Corretude

- Propriedades
 - **Safety:** uma propriedade deve ser sempre verdadeira
 - **Liveness:** uma propriedade deve eventualmente se tornar verdadeira
- É impossível demonstrar a corretude de um programa apenas por testes. Provas formais são necessárias.

Corretude

- More precisely, for a **safety** property P to hold, it must be true that in every state of every computation, **P is true**. For example, we might require as a safety property of the user interface of an operating system: **Always, a mouse cursor is displayed**. If we can prove this property, we can be assured that no customer will ever complain that the mouse cursor disappears, no matter what programs are running on the system.

Corretude

- For a **liveness** property P to hold, it must be true that in every computation there is some state in which P is **true**. For example, a liveness property of an operating system might be: **If you click on a mouse button, eventually the mouse cursor will change shape.** This specification allows the system not to respond immediately to the click, but it does ensure that the click will not be ignored indefinitely.

Fairness

- Um cenário é considerado justo se, em qualquer estado, uma determinada instrução é sempre alcançada.

Algorithm 2.5. Stop the loop A

<pre>integer n ← 0 boolean flag ← false</pre>	
p	q
<pre>p1: while flag = false p2: n ← 1 - n</pre>	<pre>q1: flag ← true q2:</pre>

Esse programa necessariamente para (halt)?

Exercícios

- Livro
 - 2, 3, 4, 6, 7, 8, 10.

Referências

- Referências das Imagens:

Ben-Ari, M. (2006). Principles of Concurrent and Distributed Programming. Boston: Addison-Wesley (Second Edition). ISBN 978-0-321-31283-9.