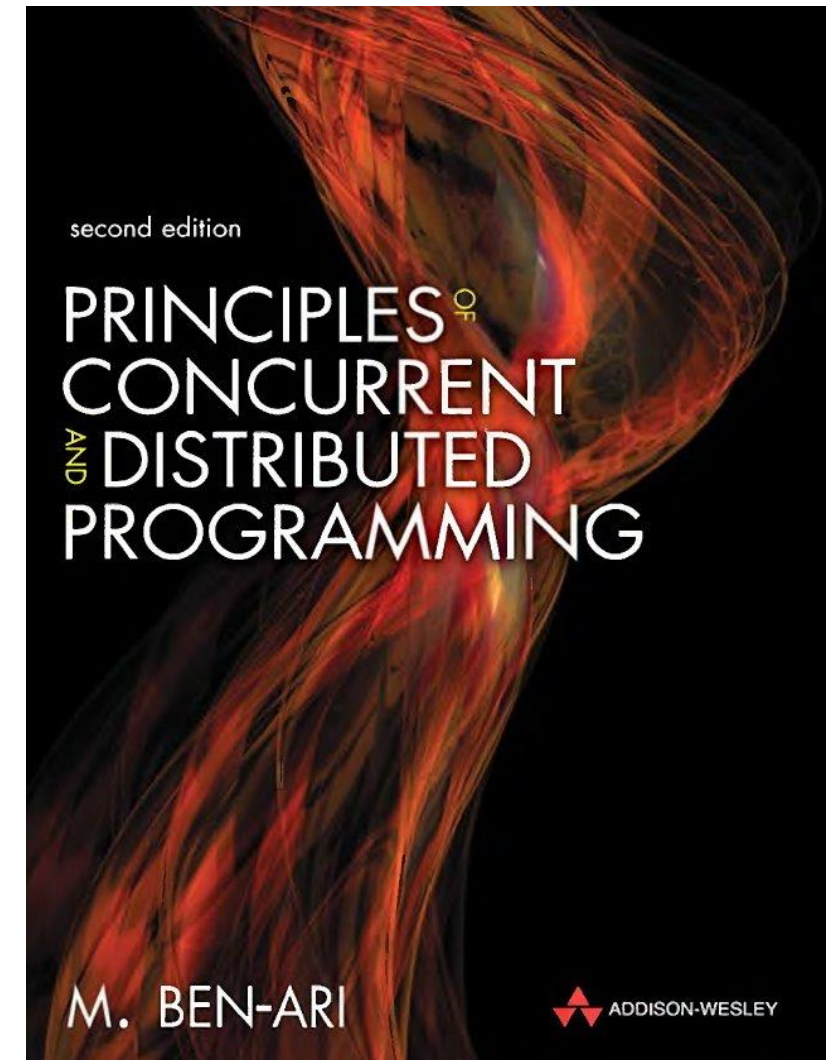


Introdução a Sistemas Concorrentes

Marcelo Veiga Neves
marcelo.neves@pucrs.br

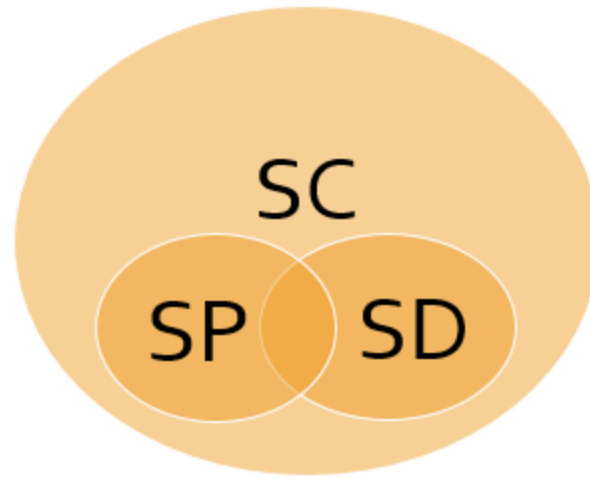
Roteiro

- Definições
- Terminologia
- Motivação
- Desafios
- Abstrações de concorrência
 - Cenários, Estados e Transições
 - Representação tabular
 - Entrelaçamento arbitrário
 - Instruções atômicas
 - Condição de corrida
 - Seção crítica



Por que concorrência?

- Por que estudar concorrência?
- Sistemas Paralelos e Distribuídos são sistemas concorrentes
 - Múltiplos processos executam de "forma simultânea"





Concorrência

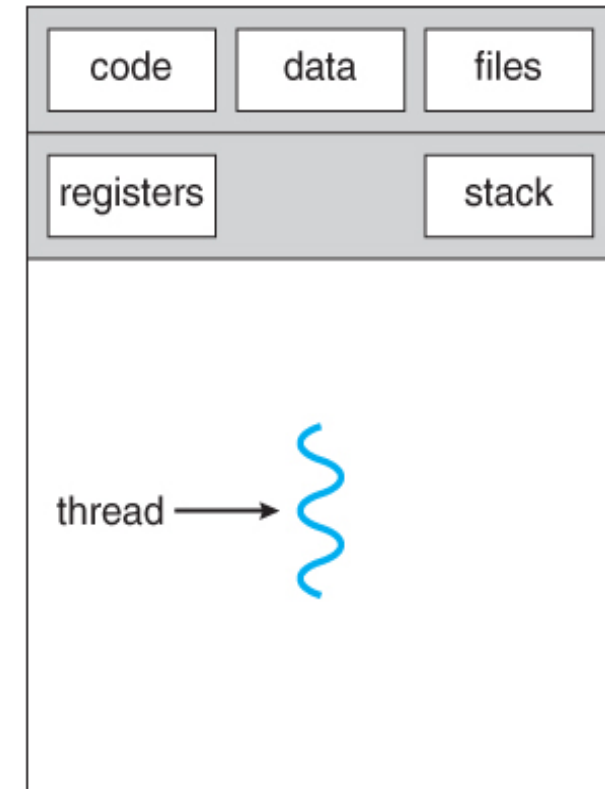
- Um programa regular tem declarações, atribuições e controle de fluxo
- Linguagens modernas se utilizam de funções e módulos para organizar códigos grandes, mas normalmente as instruções são executadas de forma **sequencial**
- Em um programa **concorrente** um conjunto de processos sequenciais é executado “ao mesmo tempo” / simultaneamente
- Exemplo: rotinas em Go

Definições de concorrência

- “ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or at the same time simultaneously, without affecting the final outcome” (Leslie Lamport)
- Concurrent computing is a form of computing in which several computations are executed concurrently — during overlapping time periods — instead of sequentially—with one completing before the next starts. (Wikipedia)

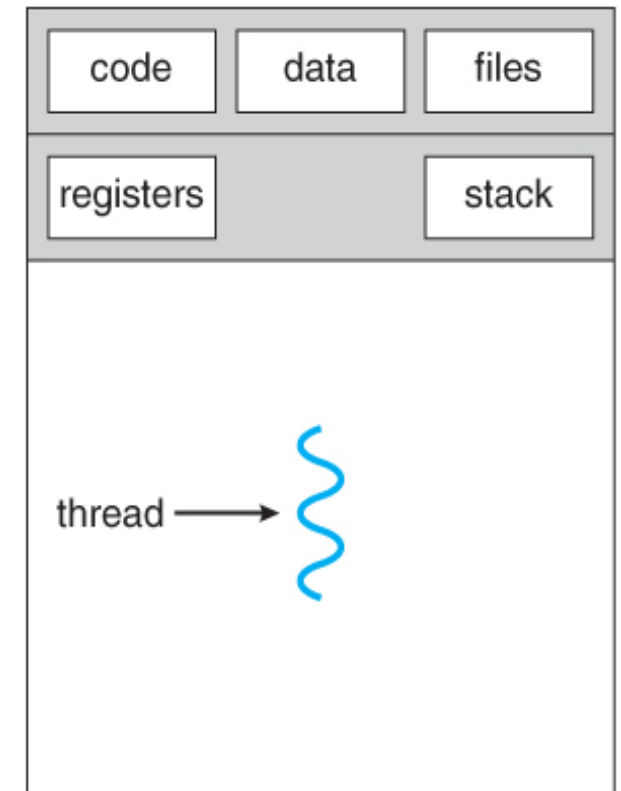
Processos

- Em Sistemas Operacionais, um processo é uma instância de programa em execução
- Um programa pode "gerar" vários processos
- Cada processo contém diferentes seguimentos:
 - Instruções que serão executadas (code/text)
 - Variáveis globais e estruturas de dados dinâmicas (data/heap)
 - Memória usada para variáveis locais (stack)
 - Registradores usados para execução das instruções (registers)
 - E outros recursos como arquivos, sockets, etc.
- Todo processo tem pelo menos um fluxo de execução



Processo e Thread

- Uma **thread** representa um fluxo de execução e contém um contador de programa (*program counter*), uma *stack* e um conjunto de registradores)
- Um **processo pesado** é um processo com uma única thread
- Podemos criar programas concorrentes formados por múltiplos processos pesados
 - Processos pesados não compartilham a memória diretamente
 - Necessidade de comunicação entre processos (IPC – Inter-processing communication)



Terminologia

- Para facilitar, daqui pra frente usaremos:
- **Processo** para os códigos sequenciais que rodam de forma concorrente
 - No contexto de SO e linguagens de programação: processos (pesados) e threads (leves, associadas a um processo)
- **Programa** para o conjunto destes processos que colaboram na resolução de um problema (comunicam, compartilham dados, etc.)
 - Posso ter concorrência sem esta colaboração, ou seja entre diferentes programas, mas aí não preciso me preocupar com sincronização e comunicação entre eles.
 - Exemplo: diferentes aplicações executando em uma mesma máquina

Tipos de Concorrência

- Concorrência física (*physical concurrency*)
 - cada processo executa em um processador dedicado
 - processamento efetivamente paralelo
- Concorrência lógica (*logical concurrency*)
 - concorrência física é obtida pela multiplexação de um número menor de recursos do que de processos
 - no caso extremo time-sharing de apenas um processador
 - em um caso menos extremo 6 processos executando em uma máquina com 4 *cores*

Motivação

- Forma diferente de pensar e modelar sistemas que pode ser muito útil em função de várias situações da vida real envolverem concorrência
 - linguagens e sistemas modernos já oferecem ferramentas padrão para a construção de aplicações concorrentes
- Atualmente vários sistemas são inerentemente concorrentes
 - interfaces gráficas orientados à eventos, sistemas operacionais, sistemas tempo real, aplicações de internet como jogos *multiplayer*, *chats* e *e-commerce*
- Isolamento de falhas
 - se um processo falhar outros podem continuar.
- Responsividade
- Arquiteturas atuais são concorrentes
 - processadores *multicore* são amplamente utilizados, o que resulta na prática em concorrência física.
 - ganho de desempenho com paralelismo.

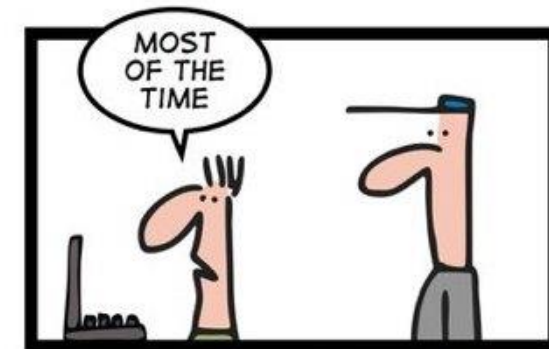
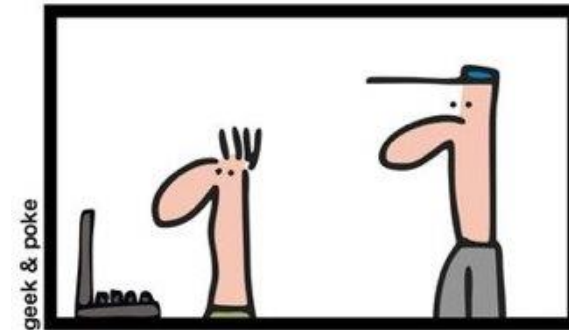
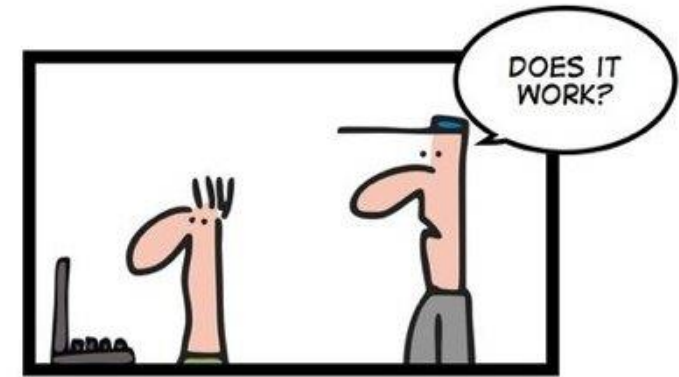
Desafios

- Temos dois principais desafios na programação concorrente:
 - **Comunicação** entre os processos
 - **Sincronização** entre os processos
- Se pertencem ao mesmo **programa**:
 - estes processo estão **colaborando** na resolução de um mesmo problema, de forma que precisam se comunicar
 - E pelo mesmo motivo, vai ser necessário um trabalho articulado entre eles, ou seja sua sincronização

Desafios

- É muito difícil implementar comunicação e sincronização de forma eficiente e segura!
 - Quando algo “congela” se trata normalmente de um erro de comunicação e/ou sincronização
- Também é muito difícil depurar problemas desse tipo

SIMPLY EXPLAINED



CONCURRENCY



Exemplo de desafio

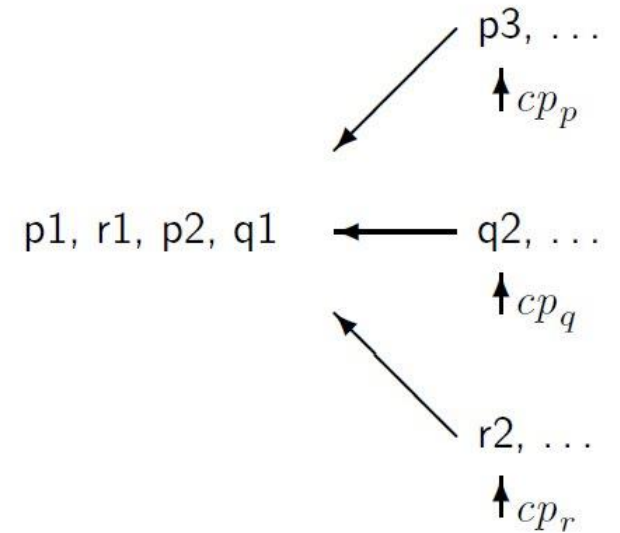
- Perda do controle de fluxo dos processos: **condição de parada**
- Como esperar por todos?
 - Em alguns ambientes, quando o processo que disparou os outros processos termina, ele mata os restantes
 - Ex: go
- Como garantir que todos terminaram?
 - Em alguns ambientes enquanto todos os processos não terminam, o programa não termina graciosamente
 - Ex: MPI

Programa Concorrente

- Um **programa concorrente** consiste de um conjunto finito de processos sequenciais
- Os processos são escritos usando um conjunto finito de **instruções atômicas**
- A execução de um programa concorrente se dá pela execução destas instruções atômicas dos processos seguindo um **entrelaçamento arbitrário** das mesmas
- Uma **computação** (*computation*) é uma sequência de execução que pode ser dar como resultado deste entrelaçamento
- Os diferentes resultados possíveis de uma computação são chamados de **cenários**

Cenários

- A próxima instrução a ser executada em um processo é definida pelo cp (*control pointer*) do processo
 - cada processo tem o seu cp
- Em cada passo da computação, a próxima instrução a ser executada será “escolhida” entre as instruções apontadas pelo cp dos processos
 - Exemplo de um cenário possível resultante da computação de 3 processos (p, q e r)



Cenários

- No caso de um exemplo com 2 processos (p e q), cada um contendo 2 instruções atômicas, teríamos os seguintes cenários possíveis

$p1 \rightarrow q1 \rightarrow p2 \rightarrow q2,$
 $p1 \rightarrow q1 \rightarrow q2 \rightarrow p2,$
 $p1 \rightarrow p2 \rightarrow q1 \rightarrow q2,$
 $q1 \rightarrow p1 \rightarrow q2 \rightarrow p2,$
 $q1 \rightarrow p1 \rightarrow p2 \rightarrow q2,$
 $q1 \rightarrow q2 \rightarrow p1 \rightarrow p2.$

- Note que $p2 \rightarrow p1 \rightarrow q1 \rightarrow q2$ não é um cenário válido, pois temos que respeitar o ordenamento da execução sequencial de cada processo
 - Assim como todos os outros cenários que contenham esta quebra no sequenciamento local

Notação independente de linguagem

- Podemos representar programas concorrentes usando uma notação independente de linguagem, pois os conceitos são universais:
 - Notação contém título do programa, declaração de variáveis globais e depois de duas colunas uma para cada um dos dois processos (chamados de p e q)
 - Cada processo pode ter declarações de variáveis locais seguidas de instruções atômicas deste processo (numeradas p1, p2, p3 ...)

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

Estados

- A execução de um programa concorrente é definida por **estados** e transições entre estados
- Vamos exemplificar estes conceitos com uma versão sequencial do programa anterior

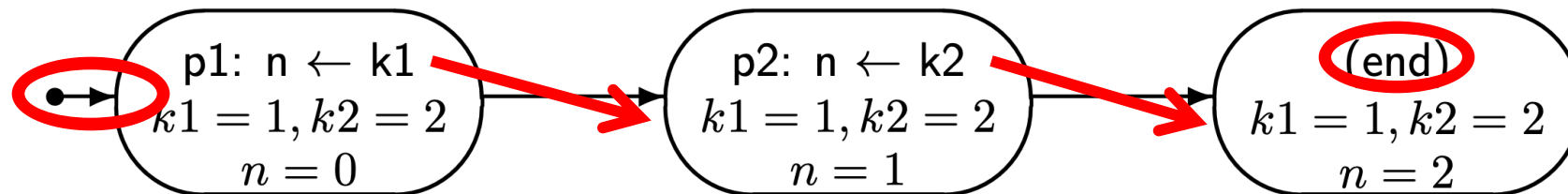
Algorithm 2.2: Trivial sequential program	
integer n \leftarrow 0	
integer k1 \leftarrow 1	
integer k2 \leftarrow 2	
p1: n \leftarrow k1	
p2: n \leftarrow k2	

Estados

- Em qualquer momento da execução, o programa tem que estar em um estado definido pelo valor de *cp* (**próxima** instrução) e das 3 variáveis
- Executar uma instrução corresponde a fazer uma transição de um estado para o próximo

Algorithm 2.2: Trivial sequential program

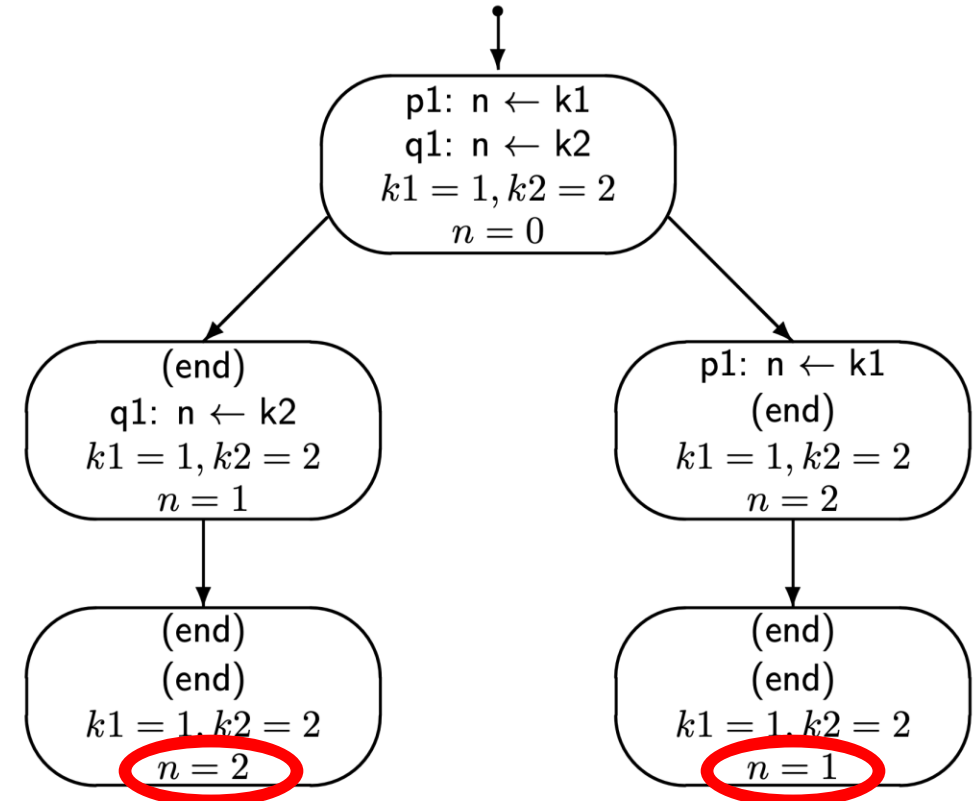
integer $n \leftarrow 0$
integer $k1 \leftarrow 1$
integer $k2 \leftarrow 2$
p1: $n \leftarrow k1$
p2: $n \leftarrow k2$



Estados e Transições

- Considere agora a execução do programa concorrente com 2 processos
 - Estado agora inclui o cp de dois processos
 - Temos duas transições possíveis do estado inicial, gerando dois cenários diferentes)

Algorithm 2.1: Trivial concurrent program	
integer $n \leftarrow 0$	
p	q
integer $k1 \leftarrow 1$ $p1: n \leftarrow k1$	integer $k2 \leftarrow 2$ $q1: n \leftarrow k2$



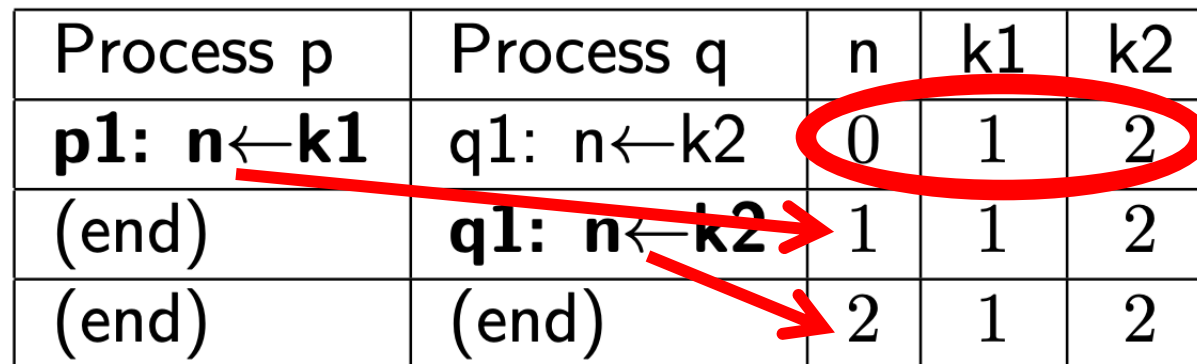
Estados

- Em resumo: o **estado** de um algoritmo concorrente é uma tupla contendo:
 - os valores para o cp de cada processo
 - os valores de cada variável global e local de todos os processos
- Para cada estado de um programa só existe um nodo no diagrama de estados
- Todos os estados em um diagrama de estados são estados alcançáveis
- Não devemos ter ciclos em um diagrama de estados
 - Significaria a possibilidade de computação infinita

Representação Tabular

- Diagramas podem complicados de desenhar para programas grandes
- Uma **representação tabular** pode ser usada para diferentes cenários de computação de um programa
 - Para cada estado do cenário temos uma linha na tabela
 - Negrito é usado para representar a instrução executada dentre as possíveis
 - Se instrução executada for uma atribuição, a atualização só é mostrada na próxima linha
 - Exemplo abaixo para o lado esquerdo do diagrama anterior

Process p	Process q	n	k1	k2
p1: $n \leftarrow k1$	q1: $n \leftarrow k2$	0	1	2
(end)	q1: $n \leftarrow k2$	1	1	2
(end)	(end)	2	1	2



Entrelaçamento Arbitrário (Arbitrary Interleaving)

- Na abstração que vamos usar, vamos nos preocupar “apenas” com o **entrelaçamento arbitrário** das instruções dos processos concorrentes
 - Não vamos nos preocupar com o tempo de execução destas instruções
 - Isto permite que possamos analisar os programas concorrentes independente do hardware que foi usado na sua execução
 - Pois só precisamos analisar as sequencias possíveis de instruções, que são contáveis e finitas
 - Não precisamos nos preocupar com o intervalo de tempo entre elas
- Desta forma é possível **analisar formalmente** estes programas
 - Provar que independente do entrelaçamento arbitrário que for ocorrer na hora da execução em uma determinada plataforma o resultado estará correto
 - Sendo assim não preciso refazer a verificação formal se mudar de máquina
- Existem ferramentas específicas para esta verificação formal caso ela seja necessária
 - Spin (<http://spinroot.com>)

Repetibilidade e Depuração

- Por causa do entrelaçamento arbitrário, fica difícil, senão impossível, repetir precisamente uma execução
- Sendo assim algumas técnicas de teste e de depuração tradicionais não fazem sentido, por exemplo:
 - 1) Executar o programa
 - 2) Diagnosticar o problema
 - 3) Corrigir o código
 - 4) Executar novamente pra ver se o problema persiste
- Cada nova execução pode seguir um entrelaçamento arbitrário diferente, ou seja, gerado um cenário diferente
 - Que não apresenta o problema que foi diagnosticado
 - Que apresenta um problema diferente do que foi diagnosticado
- O próprio diagnóstico pode mudar o entrelaçamento
 - Imprimir mensagens ao longo da execução nem sempre ajuda (stdout)

Instruções Atômicas

- A abstração que estamos usando foi definida em termos de entrelaçamento de **instruções atômicas**
- Isso significa que instruções são executadas até o final, sem a possibilidade de entrelaçamento de outras instruções durante esta execução
- A especificação do nível de atomicidade das instruções é importante pois pode afetar a corretude de um programa
 - Chamadas de bibliotecas
 - Linguagens de alto nível
 - Linguagem de máquina

Instruções Atômicas

- Exemplo 1

Algorithm 2.3: Atomic assignment statements	
integer $n \leftarrow 0$	
p	q
p1: $n \leftarrow n + 1$	q1: $n \leftarrow n + 1$

- A computação do mesmo resulta em apenas dois cenários possíveis:

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

- Nos dois, o valor final da variável global n é 2
- O algoritmo é considerado **correto** em relação ao valor final de $n = 2$

Instruções Atômicas

- Exemplo 2

Algorithm 2.4: Assignment statements with one global reference	
integer $n \leftarrow 0$	
p	q
integer temp p1: $\text{temp} \leftarrow n$ p2: $n \leftarrow \text{temp} + 1$	integer temp q1: $\text{temp} \leftarrow n$ q2: $n \leftarrow \text{temp} + 1$

- Vários cenários possíveis, entre eles:

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

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

- O algoritmo é considerado **incorreto** em relação ao valor final de $n = 1$

Instruções Atômicas

- O que mudou para o segundo exemplo?
 - Especificação do nível de atomicidade
 - Poderia ter acontecido em nível de máquina com o exemplo anterior também
- Por que o algoritmo agora não está mais correto em relação a condição final?
 - Precisa que atomicidade seja mantida
 - Que p2 execute logo depois de p1 e que o mesmo ocorra para q2 em relação a q1
- A **corretude** de um programa concorrente vai depender da especificação do **nível de atomicidade**
 - Na nossa abstração instruções serão sempre atômicas
 - Pode parecer artificial mas será usado apenas para entendermos o problema

Condição de Corrida

- Resultados inesperados causados por problemas de entrelaçamento são muitas vezes chamados de condição de corrida (race condition ou race hazard)
 - Comportamento do sistema depende da sequência ou tempo de execução de eventos que não temos controle
 - Pode se tornar um erro se um ou mais dos entrelaçamentos resultantes for indesejável
 - Pode acontecer em diferentes níveis: circuitos lógicos, programas com várias linhas de execução ou em sistemas distribuídos
 - Termo introduzido em uma artigo de 1954 sobre circuitos digitais (David A. Huffman)
- Problema pode ser difícil de reproduzir por sua natureza indeterminista

"Heisenbug"

- Pode desaparecer quando executamos em modo de depuração, incluímos funcionalidade de log, ou mensagens de acompanhamento
- Um erro que desaparece desta forma é muitas vezes denominado de “Heisenbug” em referência ao físico Werner Heisenberg – observar muda o estado



“the act of observing a system inevitably alters its state”

Werner Karl Heisenberg - 1925

físico teórico alemão e um dos pioneiros da mecânica quântica

Condição de corrida data race

- Um tipo particular de condição de corrida é uma ***data race***
- Ocorre quando dois processos concorrentes:
 - Acessam a mesma memória compartilhada
 - Querem fazer uma atualização (escrita)
 - Ler + alterar + atualizar
- Nem toda condição de corrida é uma *data race*
 - Posso ter uma condição de corrida no acesso a um recurso
 - Posso ter condição de corrida no envio de mensagens

Seção Crítica

- Programas que geram entrelaçamentos que interrompem a execução de instruções de seus processos em regiões críticas e resultam em erro sofrem do problema da **seção crítica**
- Exemplos:
 - atualização do saldo de uma conta bancária (data race)
 - impressão formatada na tela (ex: programa GO visto em aula)
 - contador global (data race)
- Para evitar que isto aconteça vamos entender e prevenir o problema
- Isto será feito com uma modelagem que se utilize de **mecanismos de sincronização**

Prevenção do problema de seção crítica

- Solução é prevenir: evitar que o problema aconteça
- Tratar a parte do programa que acessa dados e/ou recursos críticos como uma região crítica e intercalar o acesso
 - Exclusão mútua: só um processo pode entrar em sua seção crítica de cada vez
 - As instruções na região crítica precisa ser executada de forma atômica

Algorithm 3.1: Critical section problem	
global variables	
p	q
local variables loop forever non-critical section preprotocol critical section postprotocol	local variables loop forever non-critical section preprotocol critical section postprotocol

Prevenção do problema de seção crítica

- Uma solução satisfatória para o problema da seção crítica precisa garantir:
 - **Exclusão mútua**: apenas um processo de um programa concorrente estará em sua seção crítica de cada vez
 - Não ocorrerá **starvation**: nenhum processo ficará indefinidamente esperando para entrar em sua seção crítica
 - Não correrá **deadlock**: se um ou mais processos querem entrar em sua seção crítica uns não podem bloquear os outros indefinidamente
- A solução também deve funcionar para qualquer número de processos compartilhando o mesmo dado ou recurso

Exemplo de Deadlock

- **Deadlock** é uma situação onde dois ou mais processos esperam para sempre em função de cada um deles estar esperando por um recurso que está alocado pro um processo do próprio grupo (espera circula)
- Exemplos:
 - alocação de recursos em sequência (AB e BA)
 - Problemas de protocolo: só entro depois que todos entrarem



Via, resti servita Madama brillante
– E. Tommasi Ferroni, 2012

Condições e Prevenção de Deadlocks

- Deadlock só ocorre se tivermos as seguintes condições em um programa (todas):
 - Exclusão mútua: processos têm acesso exclusivo a recursos compartilhados
 - Alocar e segurar: um processo pode pedir um recurso enquanto tiver outro alocado
 - Não existe preempção: recursos não podem ser retirados a força de processos que os alocaram
 - Espera circular: dois ou mais processos formam uma cadeia circular onde cada processo espera por um recurso que o próximo processo está alocado
- Podemos evitar deadlock eliminando qualquer uma destas condições

Exercício

Considere o programa a seguir usando a notação de concorrência utilizada na disciplina:

boolean flag := false, turn := false	
p	q
p1: while not flag p2: turn := not turn	q1: while not flag q2: if not turn q3: flag := true

- a) Construa a representação tabular de dois cenários de execução do programa, um que o valor de turn termina em true e outro que termina em false.
- b) Existe algum cenário em que programa nunca termine? Justifique sua resposta.

Referências

- Ben-Ari, M. Principles of Concurrent and Distributed Programming, 2nd Edition,

