

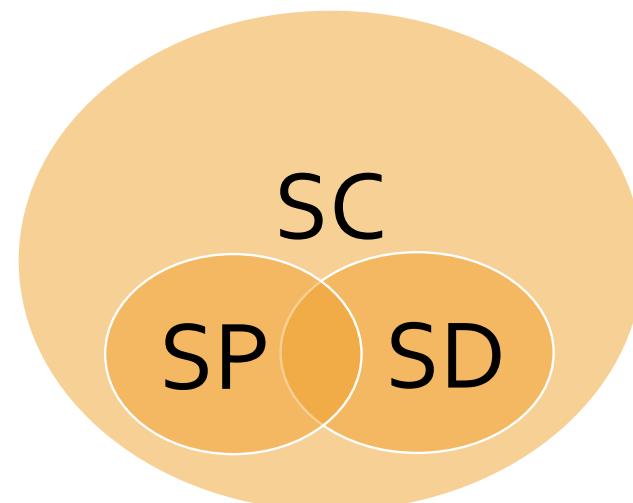
Fundamentos de Processamento
Paralelo e Distribuído

Conceitos
Fundamentais de
Sistemas Concorrentes

Prof. César A. F. De Rose

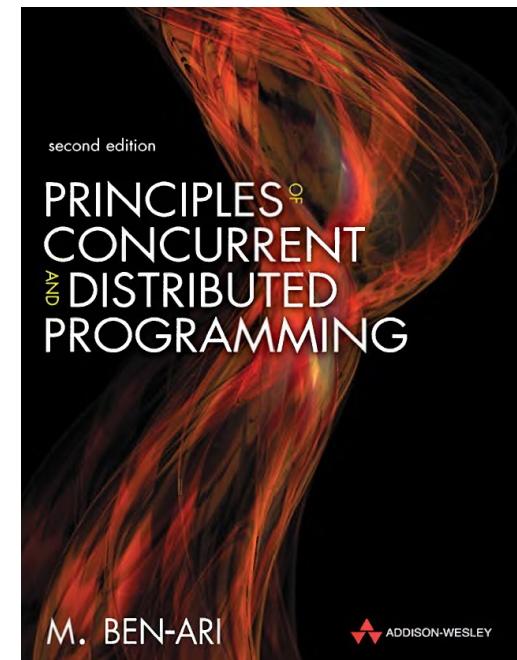
Por que Concorrência

- Por que estudar concorrência?
- Sistemas Paralelos e Distribuídos são sistemas concorrentes
 - Processos que disputam recursos
 - Modelagem: responsividade, desempenho, corretude
- Sendo assim propriedades de concorrência serão herdadas por estas sistemas



Roteiro

- Terminologia
- Motivação
- Desafios
- Abstrações usadas
 - *Estados/Transições/Cenários*
 - *Representações*
 - *Arbitrary Interleaving*
 - Estados atômicos
 - Condição de Corrida
 - Seção crítica
 - *Correctness*
 - *Safety*
 - *Liveness*
 - *Fairness*
- Concorrência em Linguagens de alto nível





Terminologia, Motivação e Desafios

Terminologia

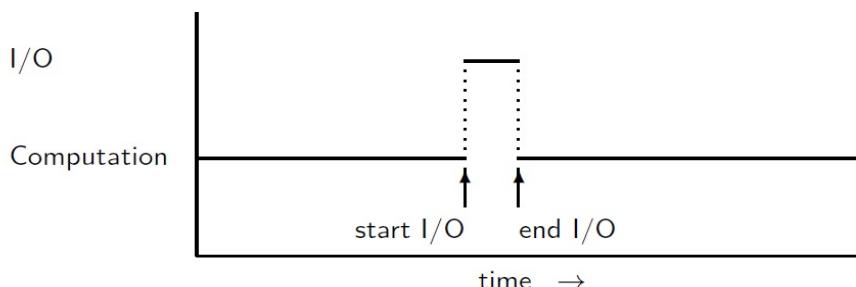


- Definição de Concorrência
 - Um **programa “normal”** tem declarações, atribuições e controle de fluxo
 - Linguagens modernas se utilizam de funções e módulos para organizar códigos grandes, mas normalmente as instruções são executadas de forma **sequencial**
[executar exemplo v0.go]
 - Em um **programa concorrente** um conjunto de programas sequenciais é executado **“ao mesmo tempo”**
[executar exemplo v1.go]

*Ben-Ari

Terminologia

- Como computadores são muito mais rápidos que humanos “ao mesmo tempo” ...
 - Pode ser apenas uma ilusão (*multitasking* de apenas um recurso)
 - Para o observador parece que todos avançam
 - Ou verdadeiramente “ao mesmo tempo”, ou seja em **paralelo**, se temos um recurso dedicado a cada programa sequencial
- O termo concorrente é uma abstração para analisarmos o comportamento destes programas sem a necessidade de sabermos exatamente como são executados e quanto tempo demoram
- Exemplo: tratamento de entrada e saída no SO



Terminologia

- Outras Definições
 - “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**”
**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*

Terminologia

- Para facilitar, daqui pra frente usaremos:
- **Processo** para os programas 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 ai não preciso me preocupar com sincronização e comunicação entre eles!
 - Exemplo: diferentes aplicações executando em uma mesma máquina

Terminologia

- Tipos de Concorrência
 - Concorrência Física (*Physical concurrency*) - cada processo executa em um processador dedicado
 - Processamento efetivamente paralelo
 - Concorrência Logica (*Logical concurrency*) – impressão de 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

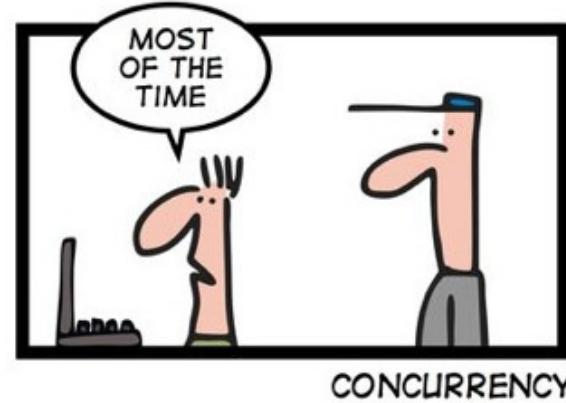
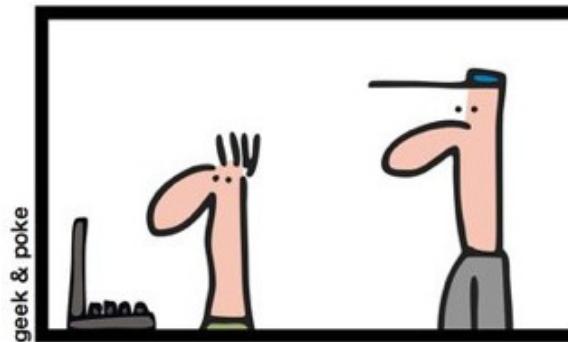
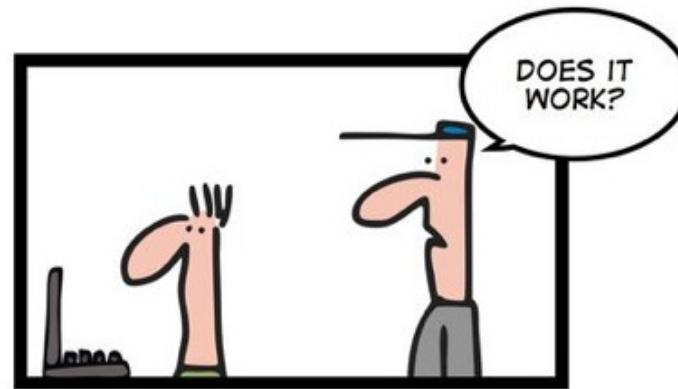
- Forma diferente de pensar e modelar sistemas que pode ser muito útil em função de várias situações da vida real envolverem concorrência
 - linguagens e sistemas modernos já oferecem ferramentas padrão para a construção de aplicações concorrentes
- Atualmente vários sistemas são inherentemente concorrentes
 - 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
- Como vimos, processadores multicore são amplamente utilizados, o que resulta na prática que em muitas casos vou ter concorrência física
 - Ganho de desempenho com paralelismo!

Desafios

- Temos dois principais desafios na programação concorrente
 - Permitir a **comunicação** entre os processos
 - E a necessidade de **sincronizar** a sua execução
- Se pertencem ao mesmo programa, estes processo estão colaborando na resolução de um mesmo problema, de forma que precisam se comunicar
 - Como no exemplo da entrada e saída no SO
- E pelo mesmo motivo acima, vai ser necessário um trabalho articulado entre eles, ou seja sua sincronização
 - Processo que precisa uma tecla, só pode continuar depois que ela for lida
- **Spoiler: é muito difícil implementar comunicação e sincronização nestes sistemas de forma eficiente e segura!**
 - Quando algo “congela” se trata normalmente de um erro de comunicação e/ou sincronização

Desafios

SIMPLY EXPLAINED



Desafios

- Objetivo desta parte do conteúdo é:
- Apresentar mecanismos, padrões, algoritmos e sistemas que são usados para se obter o funcionamento correto de programas concorrentes
 - Evitar problemas de comunicação e sincronização vistos no slide anterior
 - Ou seja, que execute **sempre** corretamente!
- **A escolha de qual mecanismo usar vai ficar a critério do desenvolvedor, dependendo dos requisitos do sistema e da arquitetura em que executa**

Desafios Exemplos

- Perda de controle que pode comprometer o resultado
 - Perda em relação ao acesso aos recursos
 - Perda em relação ao controle de fluxo



- Processo sequencial: alimentar galinha (acesso ao recurso milho)
 - composto de apenas um processo (sequencial): galinha
- Enquanto tem milho no chão
 1. procuro milho mais próximo
 2. movimento bico até milho
 3. como milho
- Controle total em relação a quanto a galinha come
 - Controlando a frequência de liberação do recurso milho

- Programa concorrente:
alimentar galinhaS
 - composto de múltiplos processos (sequenciais): galinha
- Consigo garantir que todas as galinhas comam a mesma quantidade de milho?
- Consigo garantir que uma galinha não morra de fome?
- Como retomar o controle?
 - com sincronização entre os processos galinha





Desafios Exemplos

- Perda do controle de fluxo dos processos
 - **Condição de parada:** como sei quando posso parar
 - Como esperar por todos?
 - Em alguns ambientes quando o processo que disparou os outros processos termina ele mata os restantes
 - Ex: go
 - [executar exemplo v1.go sem a linha 43]
- Como garantir que todos terminaram
 - Em alguns ambientes enquanto todos os processos não terminam, o programa não termina **graciosamente**
 - Ex: MPI



Abstrações que
serão usadas

Por usar abstrações

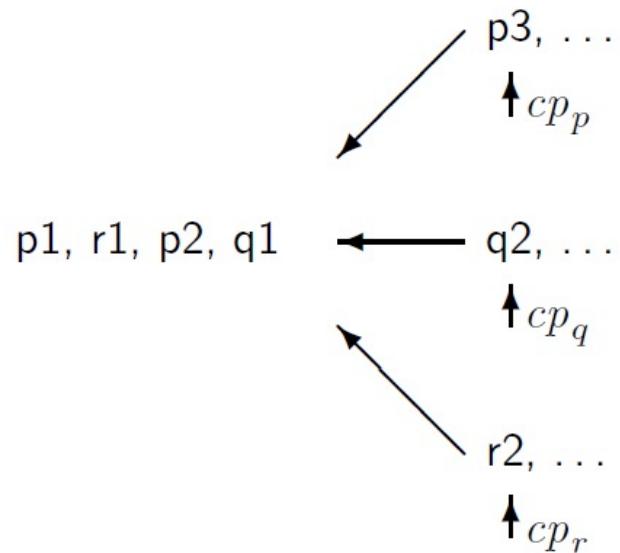
- Concorrência é um abstração para permitir que analisemos o comportamento dinâmico de alguns programas
- Agora vamos detalhar como esta abstração será usada daqui pra frente
 - Baseada no conceito de processos (sequenciais)
 - Um fragmento de um programa “normal” escrito em uma linguagem de programação

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 sequencia de execução que pode ser dar como resultado deste entrelaçamento
 - Os diferentes resultados possíveis de uma computação são chamados de **cenários**

Cenários

- Durante a execução de uma **computação** o cp (*control pointer*) de um processo indica próxima instrução a ser executada daquele processo
 - cada processo tem o seu cp
- Em cada passo da **computação** de um programa concorrente 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
 - $p_1 \rightarrow q_1 \rightarrow p_2 \rightarrow q_2,$
 - $p_1 \rightarrow q_1 \rightarrow q_2 \rightarrow p_2,$
 - $p_1 \rightarrow p_2 \rightarrow q_1 \rightarrow q_2,$
 - $q_1 \rightarrow p_1 \rightarrow q_2 \rightarrow p_2,$
 - $q_1 \rightarrow p_1 \rightarrow p_2 \rightarrow q_2,$
 - $q_1 \rightarrow q_2 \rightarrow p_1 \rightarrow p_2.$
- Note que $p_2 \rightarrow p_1 \rightarrow q_1 \rightarrow q_2$ 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 usada em programas concorrentes

- Vamos apresentar programas concorrentes usando uma notação independente de linguagem, pois os conceitos são universais

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

- O programa recebe um título, seguido da 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 ...)

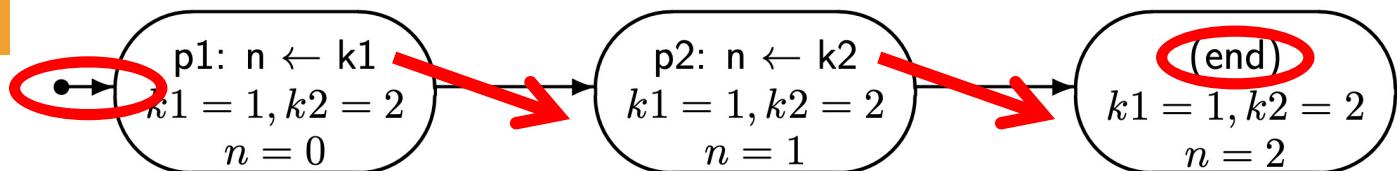
Estados

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

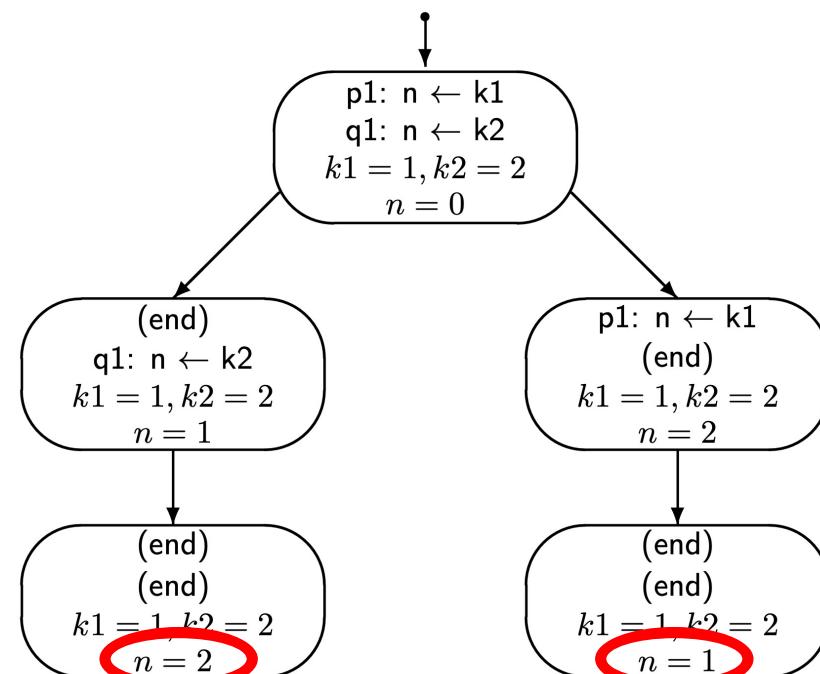
- Em qualquer momento da execução deste programa ele tem que estar em um estado definido pelo valor de cp (no caso incrementar p) e das 3 variáveis
- Executar uma instrução corresponde a fazer uma transição de um estado para o próximo



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)
 - Iniciando por p1 ou iniciando por q1

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 - Definições

- O **estado** de um algoritmo concorrente é uma tupla contendo um valor para o pc de cada processo e um valor para cada variável global e local dos processos
- Sejam S_1 e S_2 estados em um programa concorrente, existe uma transição que leva de S_1 para S_2 se executado um instrução em S_1 se chega a S_2
 - A instrução tem que estar apontada por um pc em S_1
- 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 deveríamos ter ciclos em um diagrama de estados
 - Significaria a possibilidade de computação infinita

Representação Tabular

- Como vimos um **cenário** é definido por um sequencia de estados de uma computação possível de um programa
- Como diagramas podem complicados de desenhar para programas grandes, usaremos daqui pra frente uma representação tabular 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←k1	q1: n←k2	0	1	2
(end)	q1: n←k2	1	1	2
(end)	(end)	2	1	2

Entrelaçamento Arbitrário

- 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 sequências 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

- Como em programas concorrentes estamos lidando com execuções baseadas em entrelaçamento arbitrário, fica difícil, senão impossível, repetir precisamente a sua 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
- A 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 e a consequente correção são difíceis de identificar em função de não sabermos exatamente como a execução se deu
 - 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	Process p	Process q	n
p1: n \leftarrow n + 1	q1: n \leftarrow n + 1	0	p1: n \leftarrow n + 1	q1: n \leftarrow n + 1	0
(end)	q1: n \leftarrow n + 1	1	p1: n \leftarrow n + 1	(end)	1
(end)	(end)	2	(end)	(end)	2

- Nos dois, o valor final da variável global n é 2
- O algoritmo é considerado correto em relação a condição posterior n = 2

Instruções Atômicas Exemplo 2

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	q
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

- Vários cenários possíveis

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

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 sequencia 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 especialmente em 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 indeterminística
- Pode desaparecer quando executamos em modo de depuração, incluimos 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
- Solução é evitar condições de corrida através de uma boa modelagem de software!



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

Werner Karl Heisenberg - 1925

físico teórico alemão e um dos pioneiros da mecânica quantica

Condição de corrida

Data race

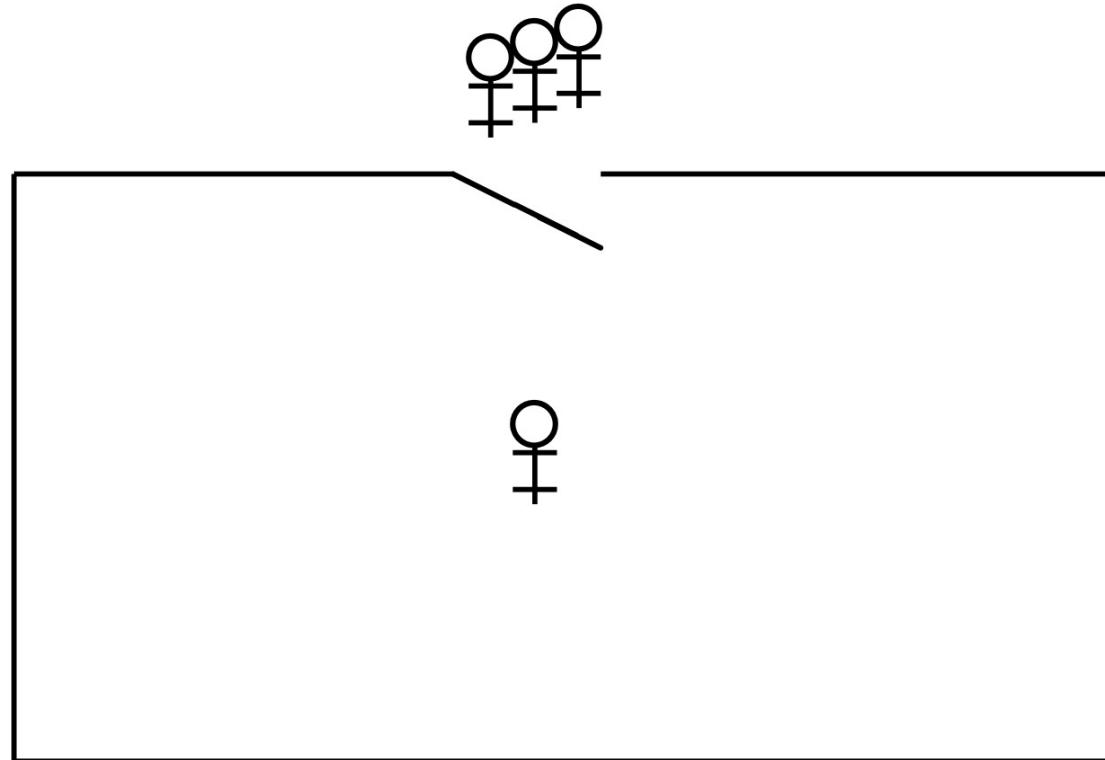
- Um tipo particular de condição de corrida é uma **data race**
- Ela ocorre quando dois processos concorrentes:
 - Acessam a mesma memória compartilhada
 - Querem fazer uma atualização (escrita)
 - Ler + alterar + atualizar
- Nem toda condição de corrida é uma *data race*
 - *Posso ter uma condição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

Seção Crítica Definição

- 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
 - 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**

Seção Crítica Prevenção

- Exemplos:
 - Fila para um banheiro
 - Acesso a uma máquina de chocolates



Seção Crítica Prevenção

- 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 critica de cada vez
 - Assim a seção crítica será executada garantidamente 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

Seção Crítica Prevenção

- 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 um grupo de processos espera para sempre em função de cada um deles estar esperando por um recurso que está alocado pro um processo do próprio grupo (*circular waiting*)
- Exemplo:
 - alocação de recursos em sequencia (AB e BA)
 - Problemas de protocolo: só entro depois que todos entrarem



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

Deadlock

Condições e Prevenção

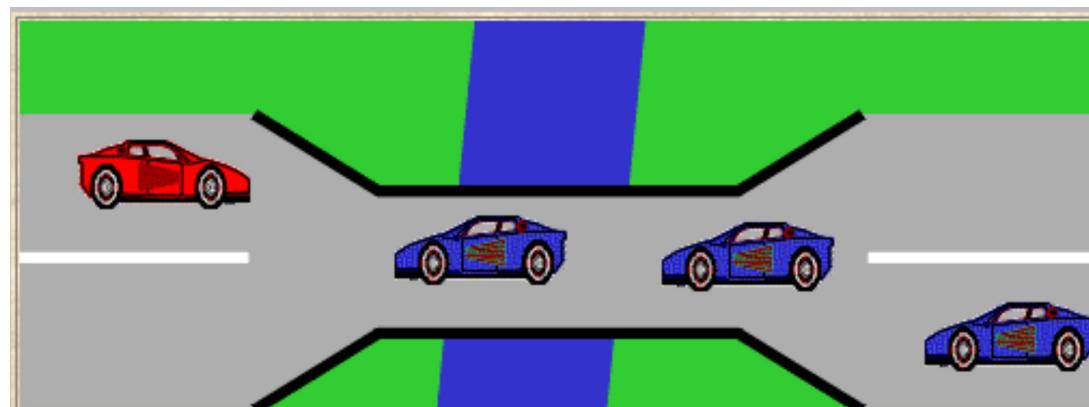
- Deadlock só ocorre se tivermos as seguintes condições em um programa (todas)
 1. Exclusão mútua: processos têm acesso exclusivo a recursos compartilhados
 2. Alocar e segurar: um processo pode pedir um recurso enquanto tiver outro alocado
 3. Não existe preempção: recursos não podem ser retirados a força de processos que os alocaram
 4. 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

Corretude

- Como vimos, não é tão fácil depurar um programa concorrente porque não sabemos com certeza qual cenário rodou em cada execução
 - Alguns cenários podem dar o resultado correto e outros não
 - Posso estar vendo só os resultados dos cenários corretos e achar que o programa está funcionando
- Nossos exemplos até agora eram simples para facilitar o entendimento, e ficou fácil ver se estavam certos ou errados
- Mas as aplicações concorrentes reais serão bem mais complexas
 - Como fazemos para saber se estão corretos?
- **A definição de corretude em programas concorrentes é baseada em propriedades de computação:**
 - Safety
 - Liveness

Safety e Liveness

- Definições
 - Safety: não vai acontecer nada de ruim
 - Liveness: eventualmente vai acontecer algo de bom
- Exemplo da ponte de mão única
 - Safety: não vai dar batida frontal na ponte
 - Liveness: eventualmente um carro que está esperando vai conseguir passar
 - Não vai ter **postergação indefinida** de um carro



Safety e Liveness

- Definições
 - Safety: não vai acontecer nada de ruim
 - Liveness: eventualmente vai acontecer algo de bom
- Exemplo do galinheiro
 - Safety: não vai ter uma galinha passando mal por que comeu demais
 - Liveness: eventualmente uma galinha vai conseguir comer também
 - Não vai ter **postergação indefinida** de uma galinha



Garantindo Safety e Liveness

- Não temos como garantir Safety e Liveness (corretude) de um programa concorrente testando com várias execuções
- Caso seja necessário ter esta garantia em sistemas críticos podemos usar provas formais
 - Como por exemplo LTL (*linear temporal logic*)
 - Ex: programa de controle de uma usina Nuclear



Fairness Justiça

- Existe uma exceção em relação a nossa definição que qualquer entrelaçamento arbitrário é uma execução válida de um programa concorrente
- Não faz sentido aceitar que por causa deste entrelaçamento um processo **nunca** consiga rodar
 - Diferente de Liveness, onde executou e não conseguiu o recurso desejado (ponte ou milho)
 - Neste caso nunca foi escolhido para rodar (não está em nenhum cenário)
- **Definição: um cenário é justo (fair) se uma instrução que pode rodar acaba eventualmente aparecendo neste cenário**
 - Se só aceitarmos cenários justos, teríamos consequentemente um programa justo

Algorithm 2.5: Stop the loop A	
integer n \leftarrow 0 boolean flag \leftarrow false	
p	q
p1: while flag = false p2: n \leftarrow 1 - n	q1: flag \leftarrow true q2:

Concorrência em Linguagens de alto nível

Expressando concorrência em linguagens GO

- Várias linguagens modernas já permitem que se expresse concorrência de uma forma muito simples
- O nosso código exemplo v1.go é uma prova disso para a linguagem Go
 - Só precisamos incluir “go” na frente de uma chamada de função

```
func main() {  
    fmt.Println("\n Beginning main goroutine")  
    fmt.Println("\n [T1] [T2] [T3] [T4]\n")  
  
    go DoWork(1, 4)  
    go DoWork(2, 5)  
    go DoWork(3, 4)  
    go DoWork(4, 5)  
  
    fmt.Println("\n Hello from main goroutine")  
  
    time.Sleep(16 * time.Second) // give the other goroutine time to finish  
  
    fmt.Println("\n Ending main goroutine \n")  
  
    // At this point the program execution stops and all  
    // active goroutines are killed.  
}
```

Expressando concorrência em linguagens GO

- No caso de exclusão mútua (*mutex*) podemos usar diretivas alto nível de sincronização para proteger uma seção crítica

```
12 var (
13     mutex      sync.Mutex // mutex is used to define a critical section of code
14     simulation_time int    = 0
15 )
16
17 func DoWork(TaskId, iterations int) {
18     for i := 1; i <= iterations; i++ {
19         mutex.Lock()
20         fmt.Printf("%2d:", simulation_time)
21         simulation_time++
22         for s := 1; s <= TaskId*6; s++ {
23             fmt.Printf(" ")
24         }
25         fmt.Printf("f%dn", i) // working
26         time.Sleep(1 * time.Second)
27         mutex.Unlock()
28     }
29 }
```

Expressando concorrência em linguagens Java

• Objeto do tipo Thread, método Run

```
1  class Count extends Thread {  
2      static volatile int n = 0;  
3  
4      public void run() {  
5          int temp;  
6          for (int i = 0; i < 10; i++) {  
7              temp = n;  
8              n = temp + 1;  
9          }  
10     }  
11  
12    public static void main(String[] args) {  
13        Count p = new Count();  
14        Count q = new Count();  
15        p.start();  
16        q.start();  
17        try {  
18            p.join();  
19            q.join();  
20        }  
21        catch (InterruptedException e) { }  
22        System.out.println ("The value of n is " + n);  
23    }  
24 }
```

Expressando concorrência em linguagens Java

- No caso de exclusão mútua podemos usar a diretiva de sincronização *synchronized* em um método para proteger uma seção crítica

```
public void motoComeIn() {  
    synchronized (controlMotorcycles) {  
        numberMotorcycles++;  
    }  
}  
  
public void motoGoOut() {  
    synchronized (controlMotorcycles) {  
        numberMotorcycles--;  
    }  
}
```

Expressando concorrência em linguagens C/C++

- Com o auxilio de bibliotecas (pthreads) ou OpenMP (pré-compilação)

```
4  #include <omp.h>
5
6  int main (int argc, const char * argv[])
7  {
8      const int intervalo = 5000;
9      double starttime, stoptime;
10
11     int i,j,k;
12     int prime;
13     int total;
14
15     starttime = omp_get_wtime();
16
17     omp_set_num_threads(8);
18
19     #pragma omp parallel private ( i, j, k, prime, total )
20     #pragma omp for schedule (dynamic)
21
22     for (k = 1 ; k <= intervalo ; k++)
23     {
24         total = 0;
25
26         for ( i = 2; i <= k ; i++ )
27         {
28             prime = 1;
29             for ( j = 2; j < i; j++ )
30             {
```

Expressando concorrência em linguagens C/C++

- No caso de exclusão mútua podemos usar diretivas alto nível de sincronização para proteger uma seção crítica
 - Definindo um conjunto de instruções como atômicas

```
sum = 0.0;

#pragma omp parallel for private (i) shared (sum)
#pragma omp parallel for reduction(+: sum) schedule (dynamic)

for (i=0; i < n; i++)
    #pragma omp atomic
    sum = sum + (a[i] * b[i]);

printf("\nSum = %f\n",sum);
```



Dúvidas?