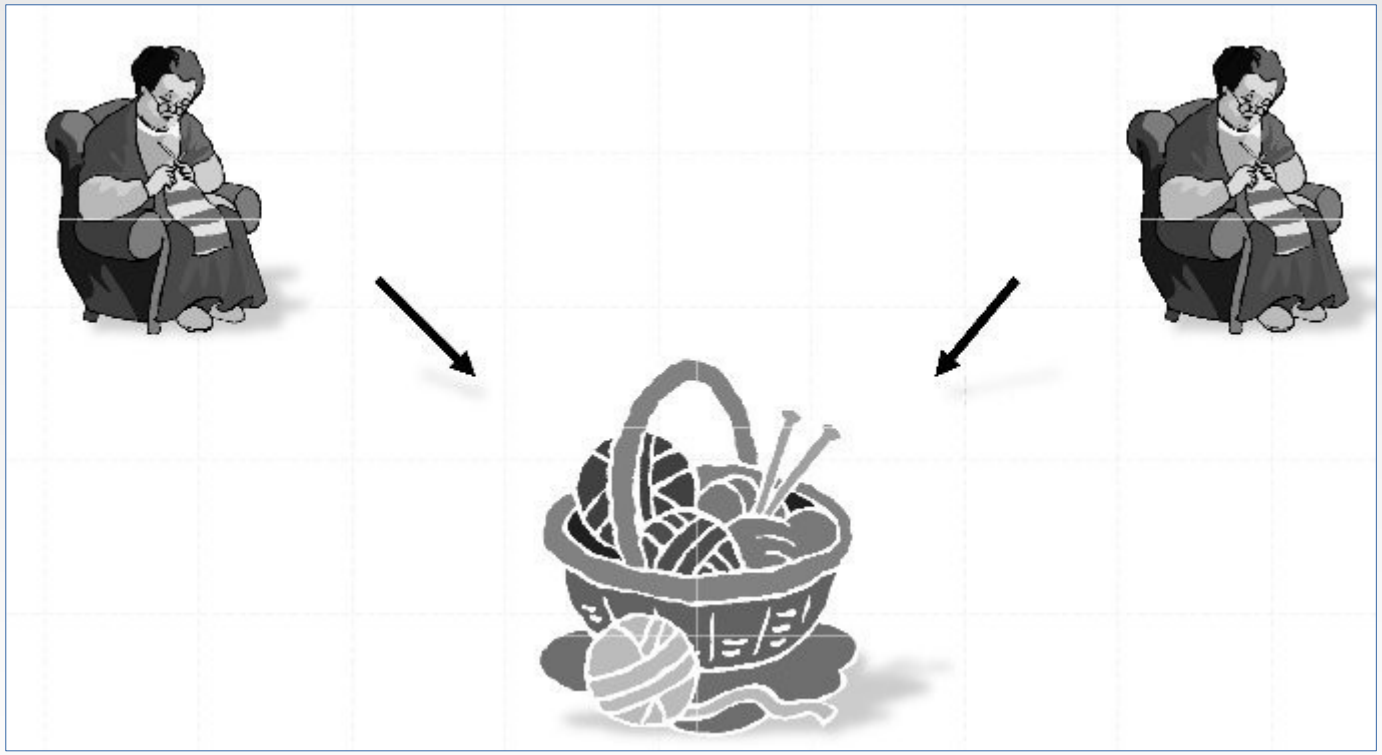
Comportamento de processos concorrentes

- Projetar processos concorrentes exige tratamento especial quanto à sincronização e comunicação entre os mesmos
- Comportamentos usuais:
  - Processos concorrentes que competem por recursos
    - Típicos em SO e redes devido à existência de recursos comuns compartilhados
  - Processos concorrentes que cooperam
    - Combinados resolvem um problema comum

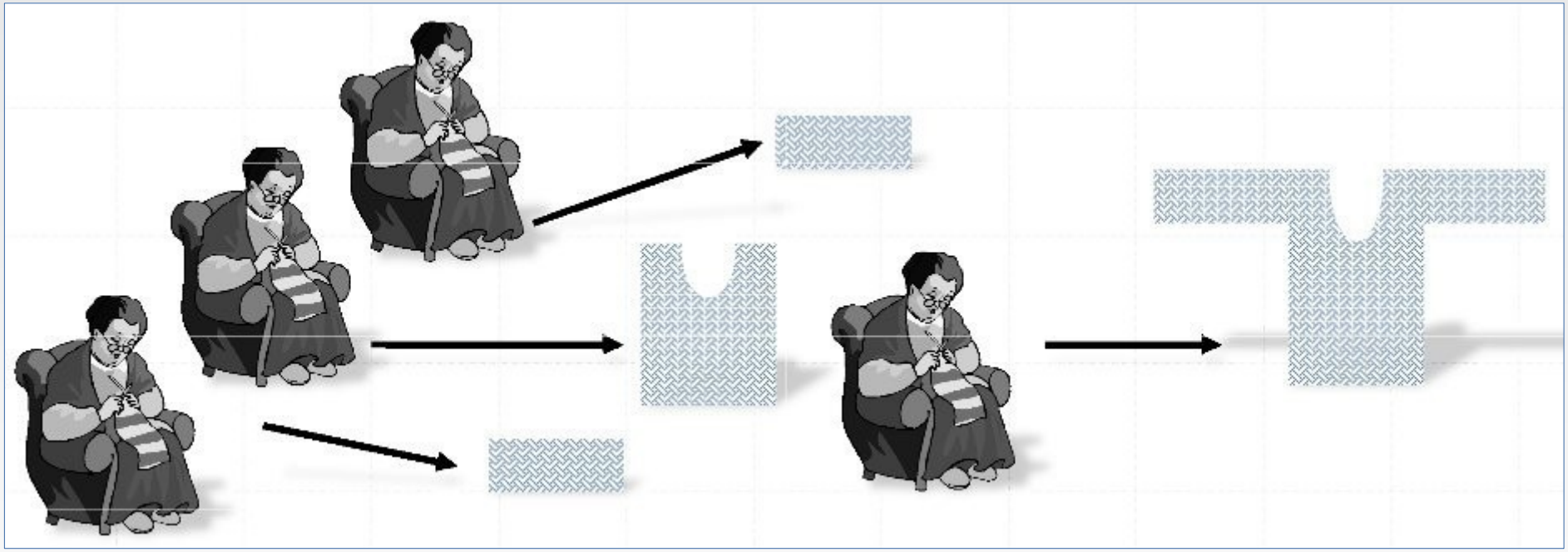
Competição por recursos
Estado inicial



Competição por recursos - *deadlocks* (travamento)

Competição por recursos - *Starvation* (falta de recurso)

Cooperação entre processos



Comunicação e Sincronização de proc. conc.

- Porque a comunicação e sincronização de processos concorrentes (necessária) é difícil
  - Fácil entre os humanos
  - Comunicação nos computadores é restrita à:
  - Ler/escrever áreas comuns (típica em sistemas de multiprocessadores com memória compartilhada)
  - enviar/receber mensagens (típica em sistemas de multicomputadores com memória distribuída)

Exemplo: "problema do excesso de leite"

Time	Alice	Bob
5:00	Chegou em casa	
5:05	Checou geladeira: sem leite	
5:10	Saiu pra comprar leite	
5:15		Chegou em casa
5:20	Chegou na padaria	Checou geladeira: sem leite
5:25	Comprou leite	Saiu pra comprar leite
5:30	Chegou em casa; guardou o leite na geladeira	
5:40		Chegou na padaria; Comprou leite
5:45		Chegou em casa; Excesso de leite!

Resolvendo o problema

Alternativas para resolver o problema

- **Exclusão mútua (*Mutual Exclusion*):** Somente uma pessoa compra o leite por vez
- ***Progress*:** alguém sempre está disponível para comprar o leite

Primitivas de comunicação

- Deixar um recado (*set a flag*)
- Remover o recado (*reset a flag*)
- Ler o recado (*test the flag*)

Solução 1

Alice	Bob

if (no note)
{ if (no
milk) {
leave Note
buy milk
remove note
}
}
	if (no note)
	{ if (no
	milk) {
	leave Note
	buy milk
	remove note
	}
	}

Funciona?

Solução 1

Alice	Bob

if (no note)
{ if (no
milk) {
leave Note
buy milk
remove note
}
}
	if (no note)
	{ if (no
	milk) {
	leave Note
	buy milk
	remove note
	}
	}

Funciona?

Não, pode-se ter muito leite, caso os dois processos (pessoas) funcionem ao mesmo tempo

Solução 2: recados personalizados

Alice	Bob
leave note A	leave note B
if (no note B) {	if (no note A) {
if (no milk) {buy milk}	if (no milk) {buy milk}
}	}
remove note A	remove note B

Funciona?

Solução 2: recados personalizados

Alice	Bob
leave note A	leave note B
if (no note B) {	if (no note A) {
if (no milk) {buy milk}	if (no milk) {buy milk}
}	}
remove note A	remove note B

Funciona?

Não, pode-se ficar sem leite

```
Alice
leave note A
while (note B) {skip}
if (no milk){
    buy milk
}

remove note A
```

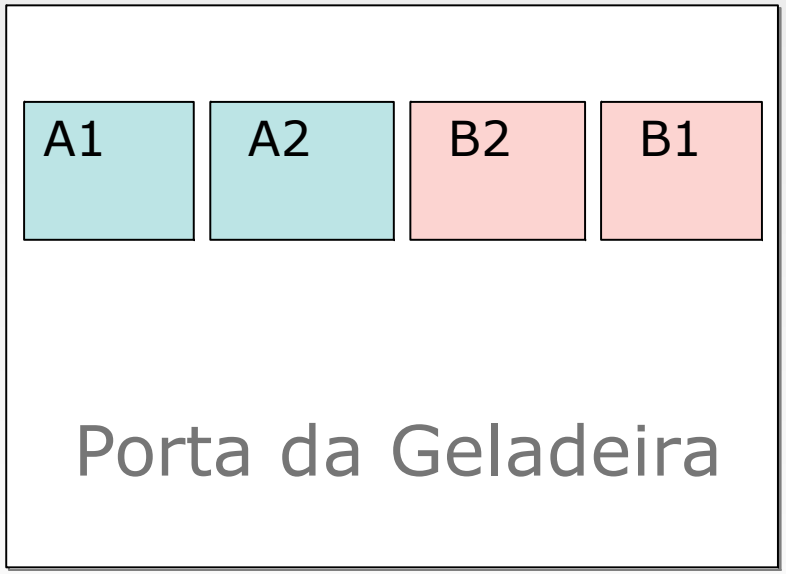
```
Bob
leave note B
if (no note A) {
    if (no milk) {
        buy  milk
    }
}

remove note B
```

Solução assimétrica!

Um dos processadores fica sempre processando (*busy waiting*)

Solução 4

Usar recados com rótulos identificadores (*tags*)



Solução 4 (cont.)

Alice

```
leave A1
if B2 {leave A2} eles {remove A2}
while B1 and ((A2 and B2) or
                (no A2 and no B2))
    {skip}
if(no milk) {buy milk}


remove A1
```

Bob

```
leave B1
if (no A2){leave B2}else{remove B2}
while A1 and ((A2 and no B2) or
                (no A2 and B2))
    {skip}
if (no milk) {buy milk}


remove B1
```

Algoritmo que resolve o problema

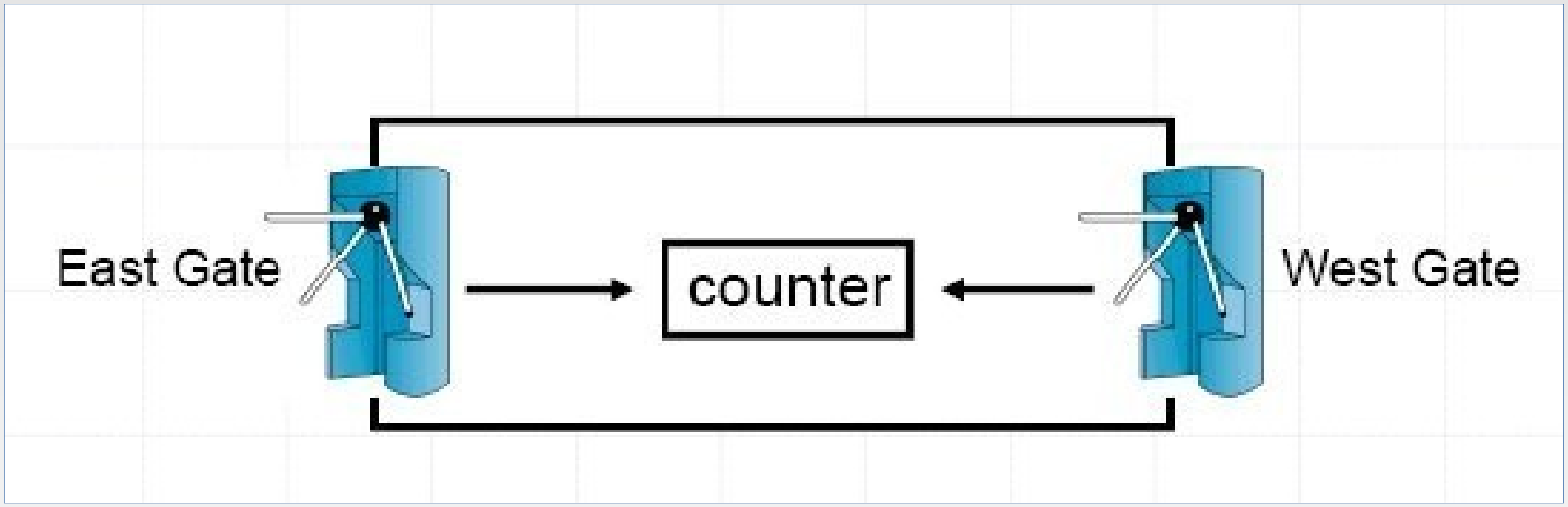
Questões a serem respondidas

- Cada operação (instrução) do algoritmo é executada sem interrupções?
    - Operações atômicas?
- Se Alice e Bob tiverem uma filha, o algoritmo irá funcionar? Quais as modificações necessárias?
    - Generalizações para várias *threads*
- Existe uma maneira clássica de se construir a solução.

Atomicidade

- **Em uma operação atômica sua ação ocorre sem interrupções do início ao fim**
- Uma vez que as operações de diferentes *threads* é intercalada, quais operações são atômicas? Isto é, não tem seu processamento interrompido para execução de outra *thread*?
  - Instruções?
  - Blocos de código?
- Resposta: troca de contexto pode ocorrer em qualquer instante como, por exemplo, no meio de instrução simples

Exemplo: Catraca

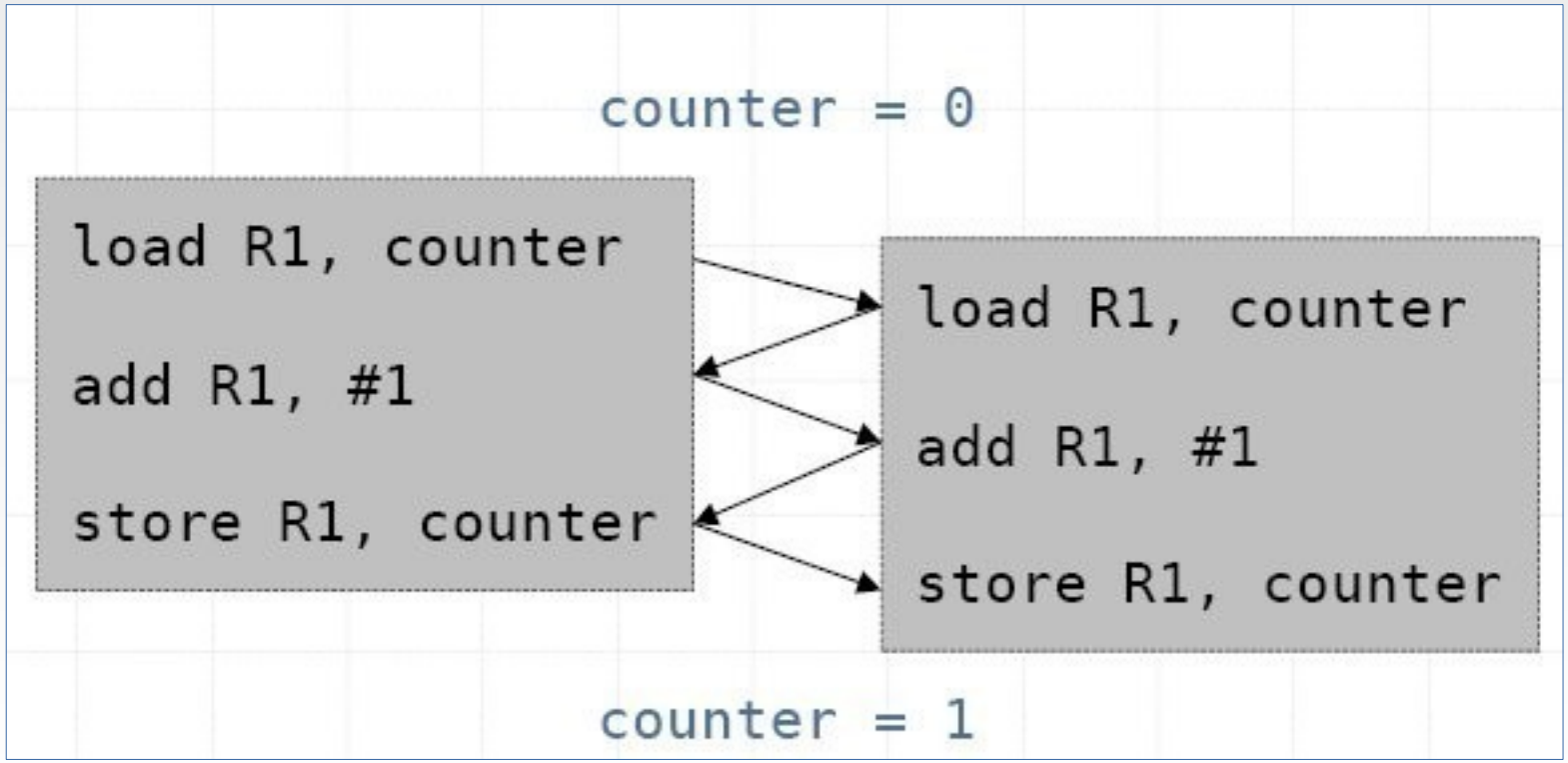


- Como obter instantaneamente o número total de pessoas em um dado ambiente controlado por duas catracas distintas?
  - Considerando que pessoas podem entrar e sair

Formas de intercalação de operações

- Para cada entrada no ambiente através de uma catraca pode-se considerar uma operação do tipo:

```
counter++
```



Solução do problema da catraca

- Resolver cada ação de incremento (`counter++`) ou decremento (`counter--`) **atomicamente**
  - Isto é, sem ser interrompida para intercalação e/ou sem sofrer ação de outro processo em paralelo
- Criação de uma **Seção Crítica**
  - Parte do programa que deve ser executada atomicamente
  - Normalmente implementada como um procedimento conhecido por **Exclusão Mútua (*Mutual Exclusion*)**: garante que apenas uma *thread* faz uma dada operação por vez.
    - Recursos da linguagem de prog.: Semáforos, monitores

Propriedades de programas concorrentes

- Dois tipos de propriedades garantem que a execução de um programa concorrente irá ocorrer corretamente como esperado:
  - Propriedade de segurança (*Safety property*)
    - Garantia de que o fluxo de execução nunca se atingirá um estado indesejável. Exemplo:
      - O programa nunca gera uma resposta errada
  - Propriedade de vivacidade (*Liveness property*)
    - Garantia que o fluxo de execução tem potencialidade de atingir, eventualmente, um estado almejado. Exemplo:
      - O processamento termina e eventualmente chama uma determinada rotina

Diagramas de Estado

- Diagrama de Estados:
  - Descrição de todos os estados possíveis
    - A raiz contém o estado inicial
    - Obtém-se os demais estados, combinando as execuções de instruções em cada *thread*
    - Estados idênticos devem ser agrupados em um único
- Estado:
  - Descrição de todos os elementos do programa
    - Apontadores de instrução das *threads*
    - Variáveis
    - Tudo o que for necessário para caracterizar a situação atual de execução das *threads*.

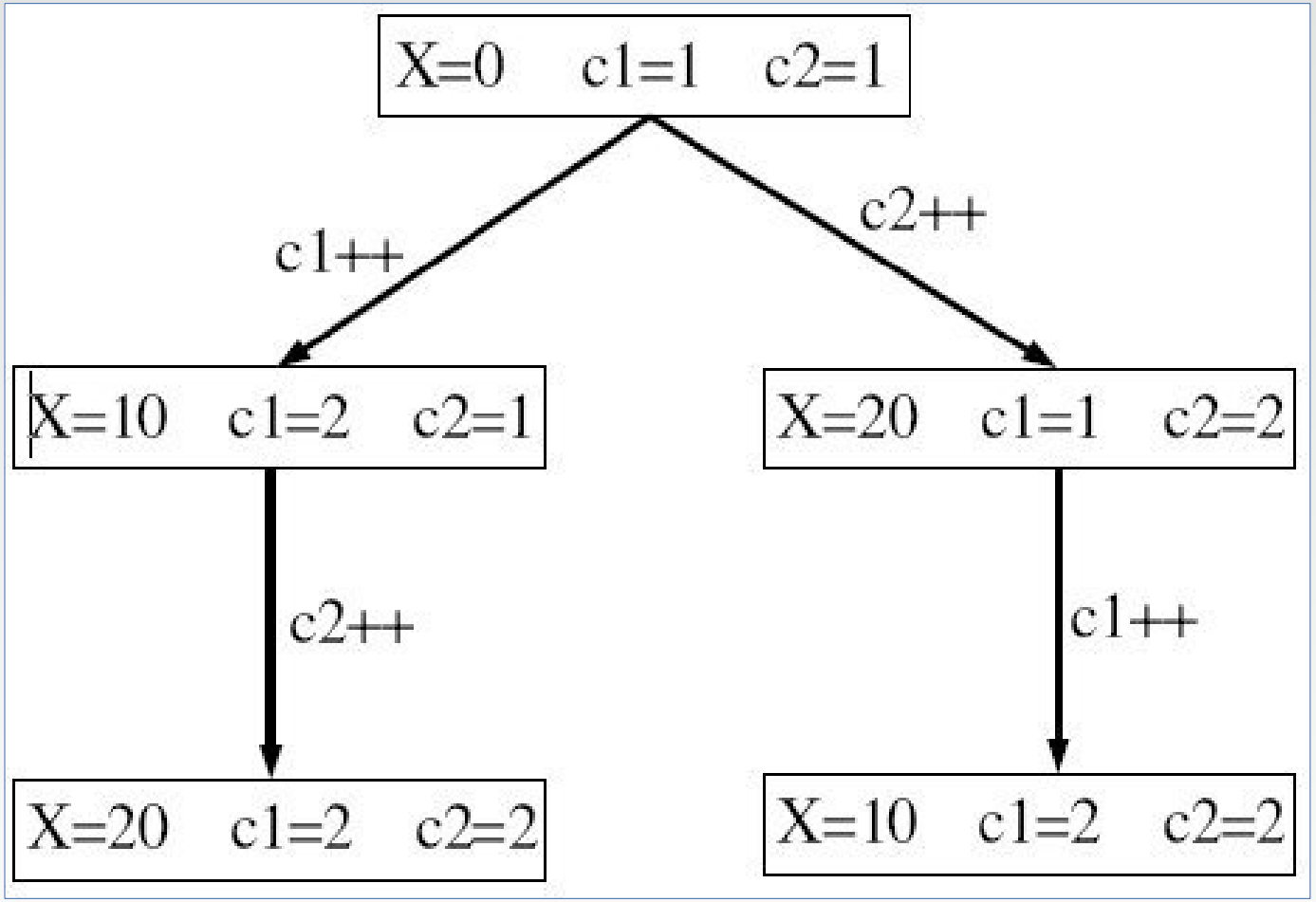
Exemplo 1

- Considere o seguinte programa em pseudo-código:

```
int x = 0;
 thread 1:
  1: x=10;
 Thread 2:
  1: x=20;
```

- Ao final da execução, qual o resultado final de x?

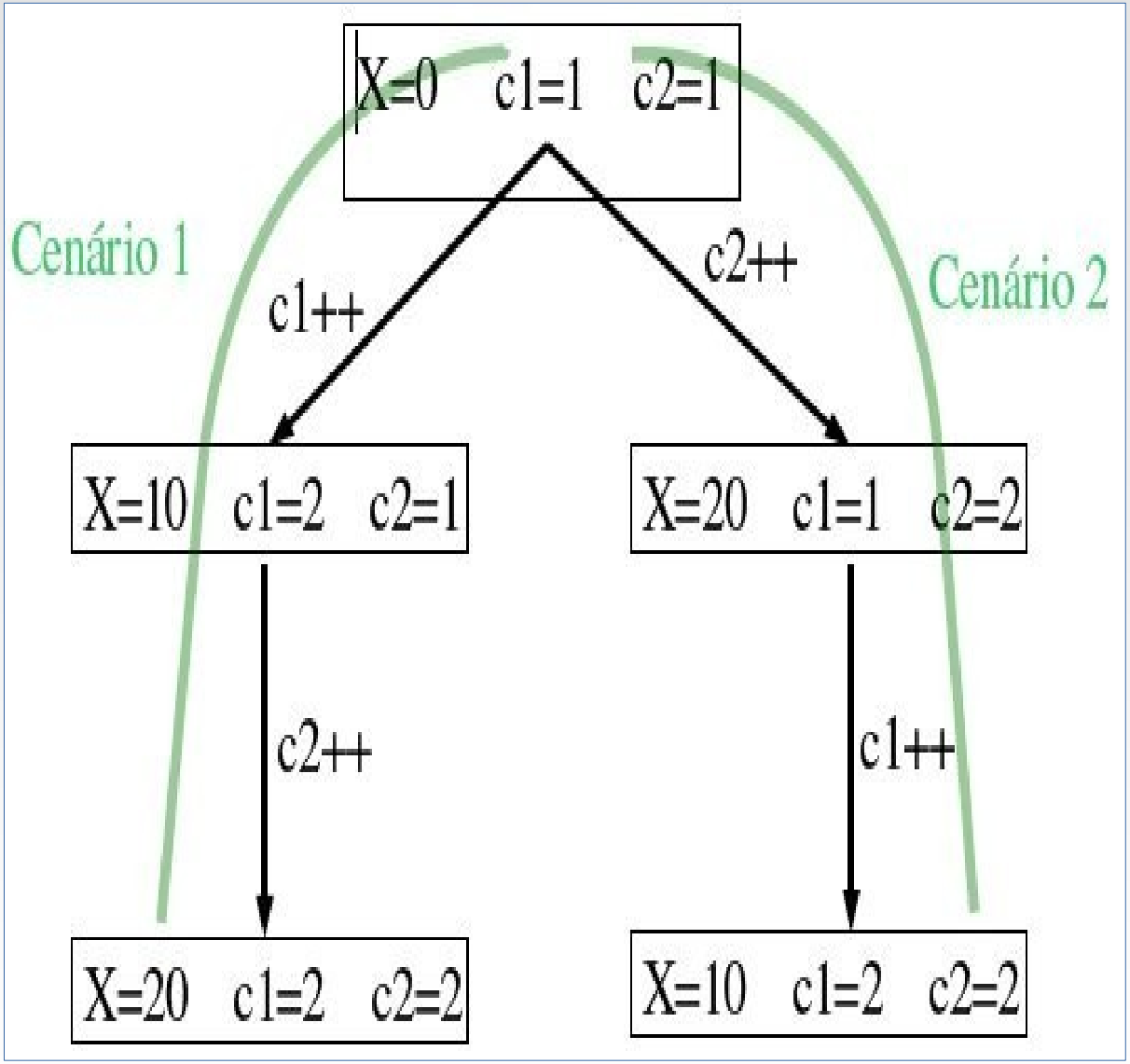
Exemplo 1 (cont.)



Onde *c1* e *c2* representam os contadores de programa para as *threads* 1 e 2, respectivamente

Cenários

- Cenários descrevem uma sequencia de estados
  - Contém apenas uma ramificação do diagrama de estados
  - Utiliza uma descrição tabular (forma de tabela)



Cenários: Descrição tabular

- Colunas:
  - Instruções executadas (ou apontadores de instrução)
  - Variáveis
- Linhas:
  - Cada linha representa um estado
  - A primeira instrução executada está na primeira linha da tabela
  - A linha seguinte contém o próximo estado
  - Se a instrução é uma atribuição, o valor da variável é alterado no próximo estado

Tabelas

Cenário 1:	C1	C2	x	Thread 1	Thread 2
	2 (fim)	1	0	x=10	
	2 (fim)	2 (fim)	10		x=20
	2 (fim)	2(fim)	20		

Cenário 2:	C1	C2	x	Thread 1	Thread 2
	1	2 (fim)	0		x=20
	2 (fim)	2 (fim)	20	x=10	
	2 (fim)	2(fim)	10		

Exemplo 2

int n = 0;	
Thread 1:	**Thread 2:**
1: n = n + 1	1: n = n + 1

Cenário 1

C1	C2	n	Thread1	Thread2
2	1	0	n=n+1	
2	2	1		n=n+1
2	2	2		

Cenário 2

C1	C2	n	Thread1	Thread2
1	2	0		n=n+1
2	2	1	n=n+1	
2	2	2		

Exemplo 3

- Considerando que cada operação atômica pode referenciar uma variável global apenas uma única vez.
    - Alteração do código que considera variáveis locais auxiliares: ***x*** e ***y*** para as *threads* 1 e 2 respectivamente

int n = 0;	
Thread 1:	**Thread 2:**
int x; 1: x = n 2: n = x + 1	int y; 1: y = n 2: n = y + 1

Cenários para exemplo 3

1

C1	C2	n	x	y	Thread 1	Thread 2
2	1	0			x = n	
3	1	0	0		n = x+1	
3	2	1	0			y = n
3	3	1	0	1		n = y+1
3	3	**2**	0	1		

2

C1	C2	n	x	y	Thread 1	Thread 2
2	1	0			x = n	
2	2	0	0			y = n
3	2	0	0	0	n = x+1	
3	3	1	0	0		n = y+1
3	3	**1**	0	0		

Razoabilidade em cenários

- Um cenário é razoável se, em qualquer estado no cenário, a próxima instrução de um processo/*thread* aparece em um estado posterior
- Exemplo:

int n = 0 boolean s = false	
Thread 1:	**Thread 2:**
1: while s == false 2: n = 1 - n	1: s = true

Razoabilidade (cont.)

- Pergunta: Este código termina?
- Resposta: Pode não terminar.
    - Considere o cenário formado apenas pela execução das instruções 1 e 2 da *thread* 1
    - A instrução da *thread* 2 nunca será executada
    - Mas se considerarmos apenas cenários razoáveis, a instrução 1 da *thread* 2 será executada em algum momento, e o programa termina.
- A razoabilidade diz que se há uma instrução em condições de ser executada (a próxima de um processo/*thread*) então ela será executada em algum estado seguinte