Modelos para Computação Concorrente ou Sistemas Operacionais

Memória Compartilhada - O Problema da Seção Crítica – Soluções de SW

(com slides de Ben-Ari)

Fernando Luís Dotti

PUCRS | ESCOLA POLITÉCNICA

Síntese

- Soluções de SW para seção crítica
  - 2 processos
  - N processos
  - Raciocínio sobre corretude
    - Modelo e verificação em CSP (algoritmo de Peterson)
    - Remete a bibliografia (capítulo 4 de Ben-Ari) para prova dedutiva de algoritmos concorrentes

Bibliografia Base

[disponível na biblioteca]

M. Ben-Ari

Principles of Concurrent and Distributed Programming

Second Edition

Addison-Wesley, 2006

- Seção Crítica (SC)
  - sistema com N processos, N > 1
  - cada processo pode ter um código próprio
  - os processos compartilham dados variáveis, de qualquer tipo
  - cada processo possui SC's de código, onde atualizam os dados compartilhados
  - a execução de 1 SC deve ser de forma mutuamente exclusiva no tempo

- Seção Crítica
  - prover exclusão mútua
  - Progresso
    - não bloqueio
    - processos fora da SC não devem bloquear outros processos
    - somente os processos querendo entrar na SC devem participar da seleção do próximo a entrar
  - Espera limitada (não postergação)
    - um processo espera um tempo limitado na *entry-section*
  - velocidades indeterminadas
    - não se faz suposições sobre a velocidade relativa dos processos

Forma geral para o estudo de soluções de SC para dois processos:

- protocolos de entrada e saída
- seções críticas e não críticas
- variáveis locais e globais
- cada linha destes pseudo-códigos é considerada atômica

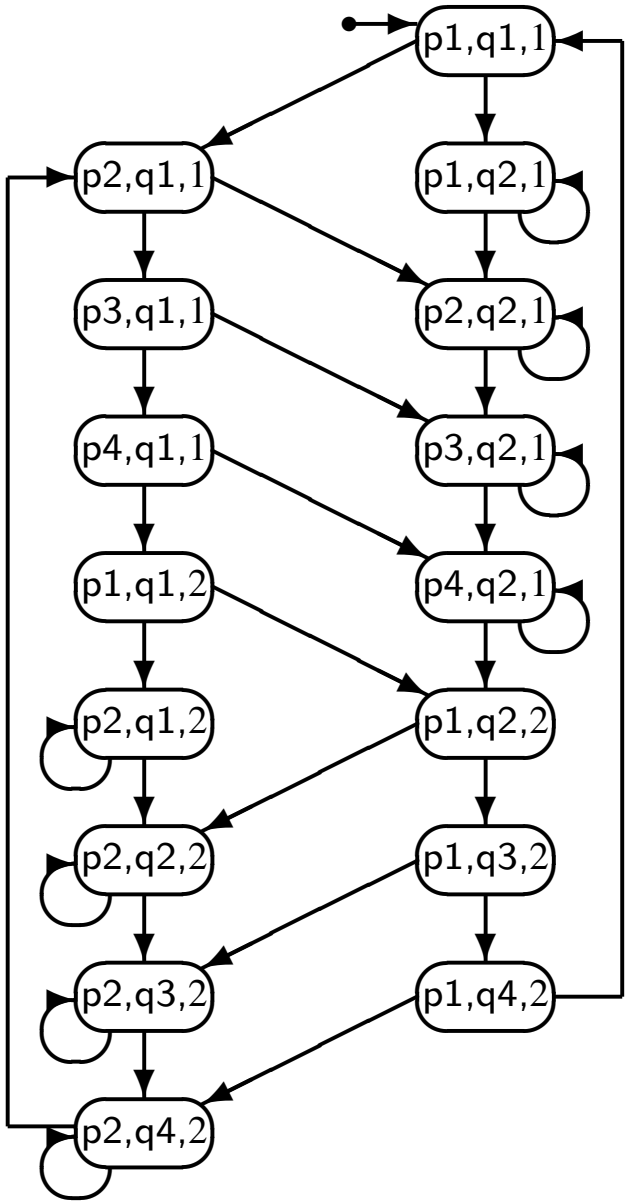
Algorithm 3.1: Critical section problem	
global variables	
p	**q**
local variables loop forever non-critical section preprotocol critical section postprotocol	local variables loop forever non-critical section preprotocol critical section postprotocol

Primeira Tentativa - Algoritmo 3.2

Algorithm 3.2: First attempt	
integer turn ← 1	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: await turn = 1	q2: await turn = 2
p3: critical section	q3: critical section
p4: turn ← 2	q4: turn ← 1

Algoritmo 3.2

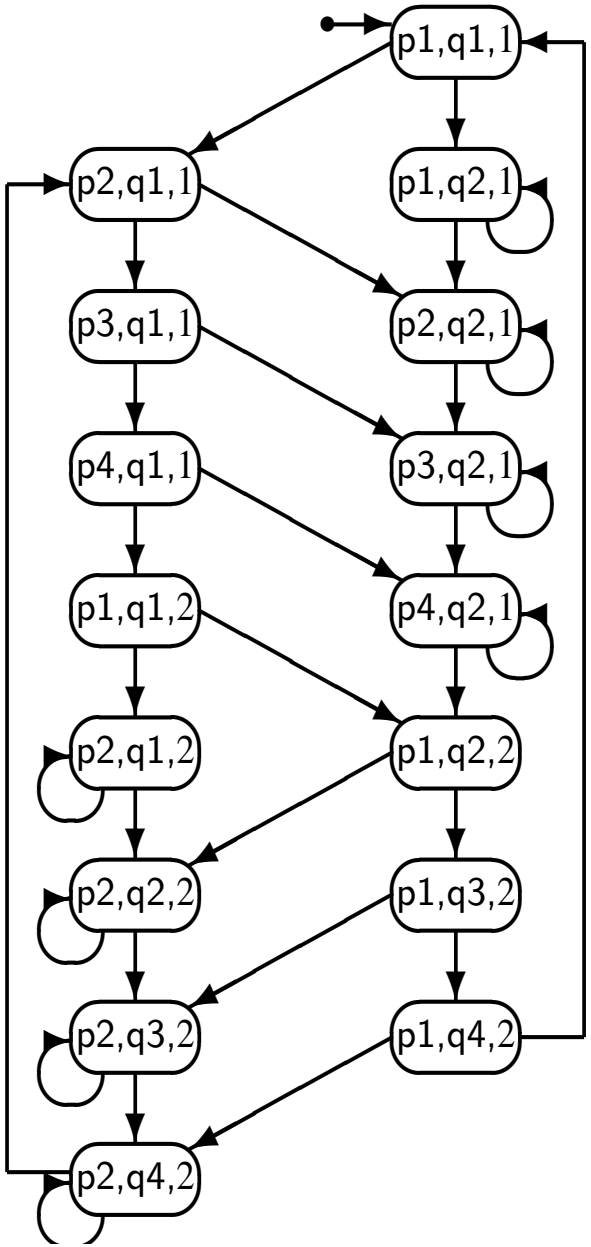
State Diagram for the First Attempt



Algoritmo 3.2

Algorithm 3.2: First attempt	
integer turn ← 1	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: await turn = 1	q2: await turn = 2
p3: critical section	q3: critical section
p4: turn ← 2	q4: turn ← 1

State Diagram for the First Attempt



Algoritmo 3.2

- Algo 3.2:
  - Exclusão mútua
    - ok
  - Progresso ?
    - processos fora da SC não devem bloquear outros processos
    - Um processo tem que usar a seção crítica para passar a vez para outro
  - Espera limitada
    - ok (se ambos processos permanecem disputando a SC)

Segunda Tentativa - Algoritmo 3.6

Algorithm 3.6: Second attempt	
boolean wantp ← false, wantq ← false	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: await wantq = false	q2: await wantp = false
p3: wantp ← true	q3: wantq ← true
p4: critical section	q4: critical section
p5: wantp ← false	q5: wantq ← false

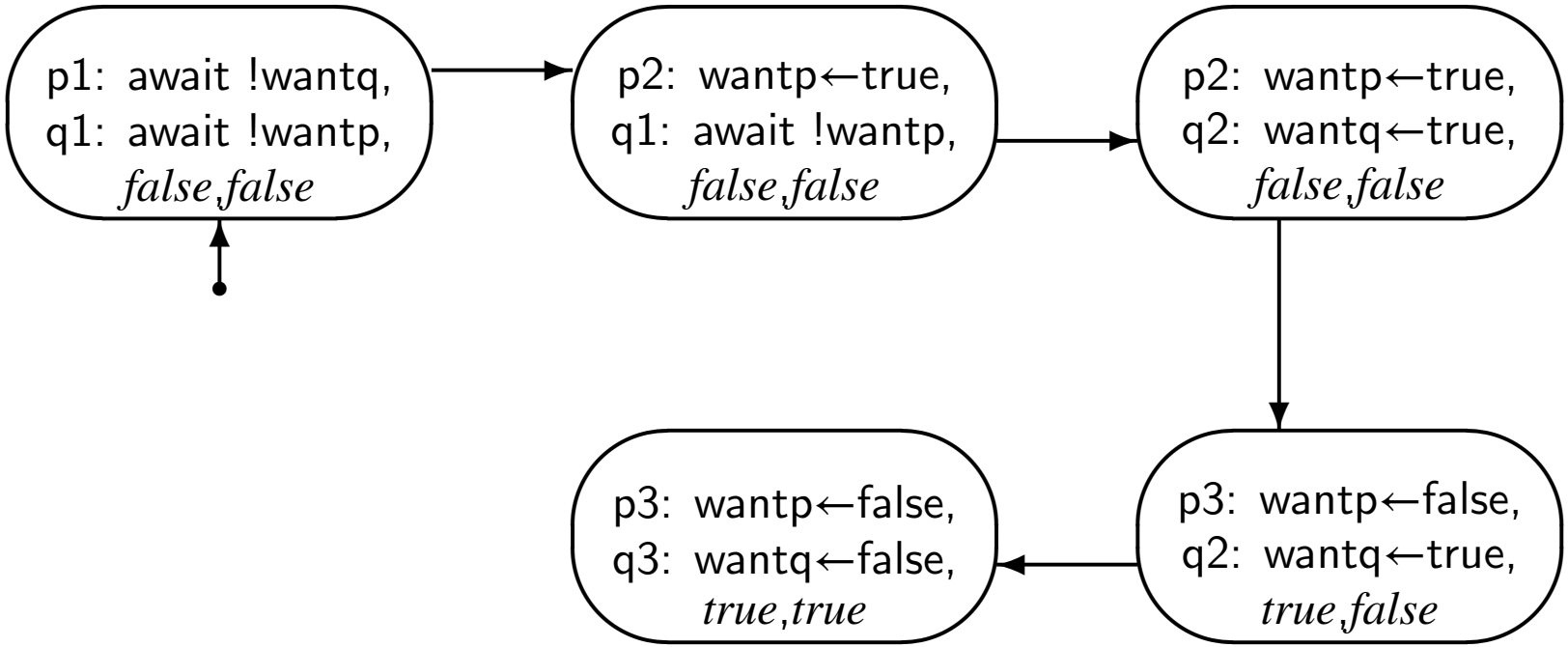
Algoritmo 3.6

Scenario Showing that Mutual Exclusion Does Not Hold

Process p	Process q	wantp	wantq
p1: await wantq=false	q1: await wantp=false	*false*	*false*
p2: wantp←true	**q1: await wantp=false**	*false*	*false*
p2: wantp←true	q2: wantq←true	*false*	*false*
p3: wantp←false	**q3: wantq←true**	*true*	*false*
p3: wantp←false	q3: wantq←false	*true*	*true*

Algoritmo 3.6

Fragment of the State Diagram for the Second Attempt



Algoritmo 3.6

- Algo 3.6:
  - Exclusão mútua
    - X
  - Progresso (?)
    - ok
  - Espera limitada (?)
    - ok

Terceira Tentativa Algoritmo 3.8

Algorithm 3.8: Third attempt	
boolean wantp ← false, wantq ← false	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: await wantq = false	q3: await wantp = false
p4: critical section	q4: critical section
p5: wantp ← false	q5: wantq ← false

Algoritmo 3.8

Scenario Showing Deadlock in the Third Attempt

Process p	Process q	wantp	wantq
p1: non-critical section	q1: non-critical section	*false*	*false*
p2: wantp←true	**q1: non-critical section**	*false*	*false*
p2: wantp←true	q2: wantq←true	*false*	*false*
p3: await wantq=false	**q2: wantq←true**	*true*	*false*
p3: await wantq=false	q3: await wantp=false	*true*	*true*

Algoritmo 3.8

- Algo 3.8:
  - Exclusão mútua
    - ?
  - Progresso
    - deadlock
  - Espera limitada

Algoritmo 3.13

Algorithm 3.13: Peterson's algorithm

```
boolean wantp ← false, wantq ← false
integer last ← 1
```

p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: last ← 1	q3: last ← 2
p4: await wantq = false or last = 2	q4: await wantp = false or last = 1
p5: critical section	q5: critical section
p6: wantp ← false	q6: wantq ← false

Algoritmo 3.13

- Exclusão mútua
  - ?
- Progresso
  - ?
- Espera limitada
  - ?

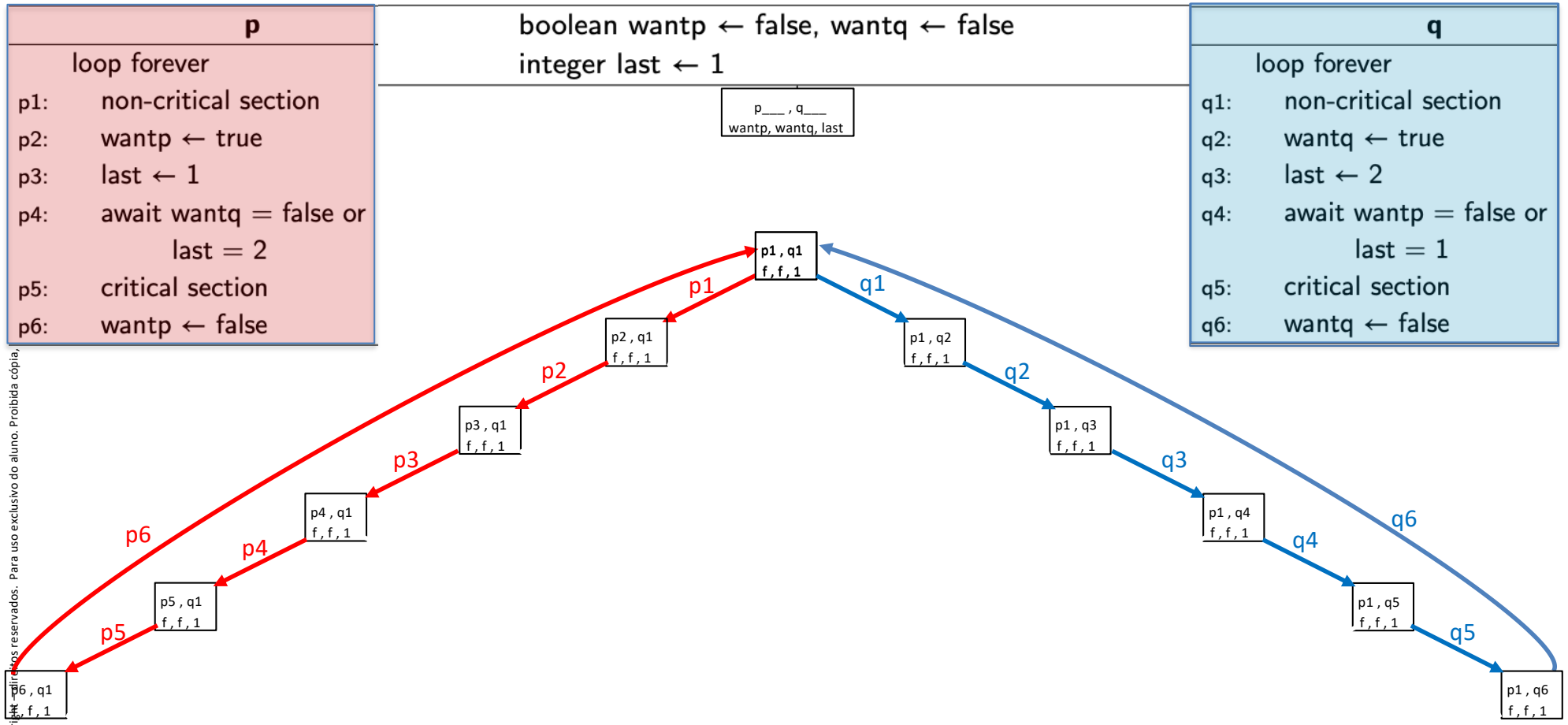
Algoritmo 3.13

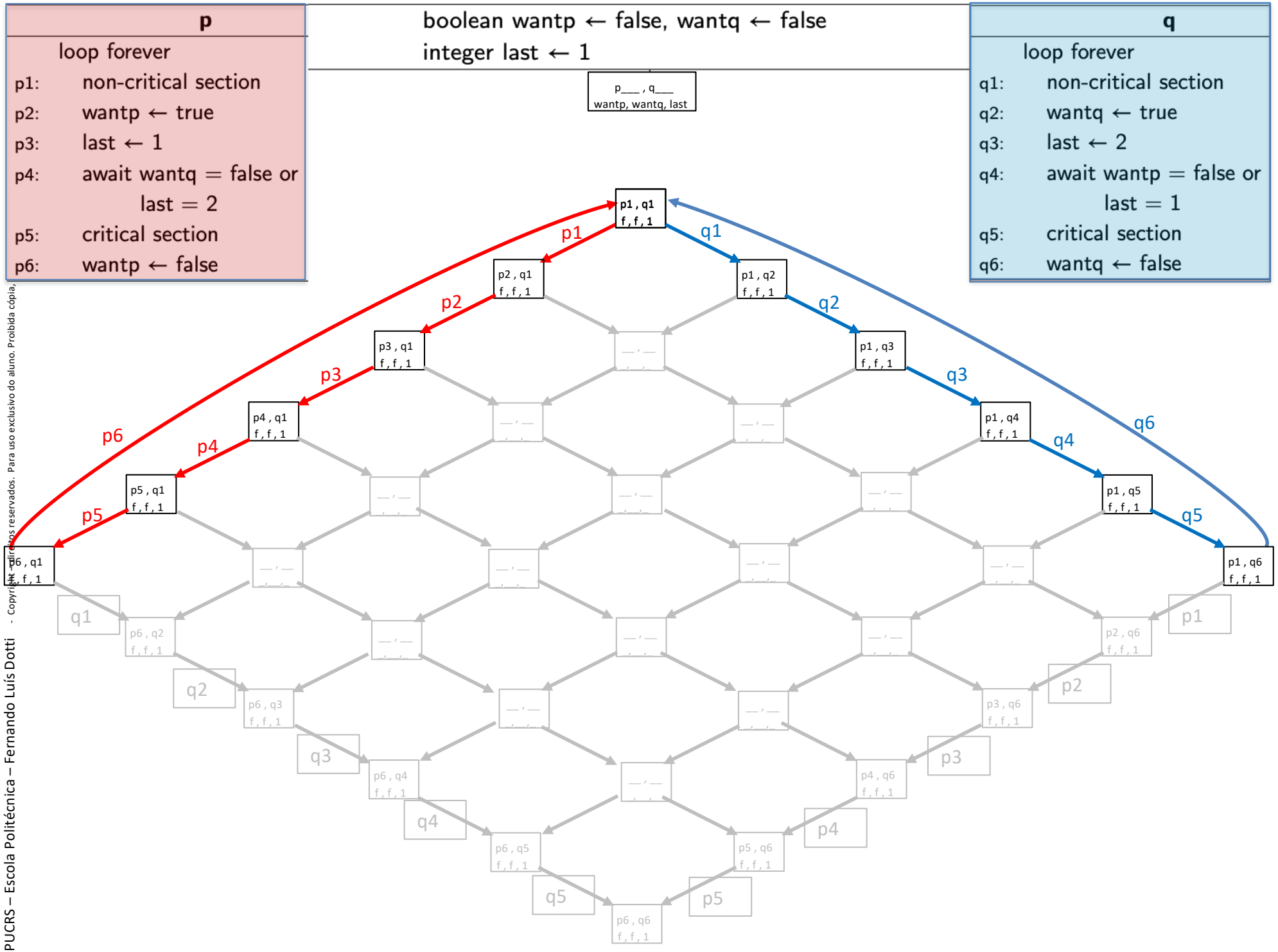
- Exclusão mútua
  - ok
- Progresso
  - ok
- Espera limitada
  - ok

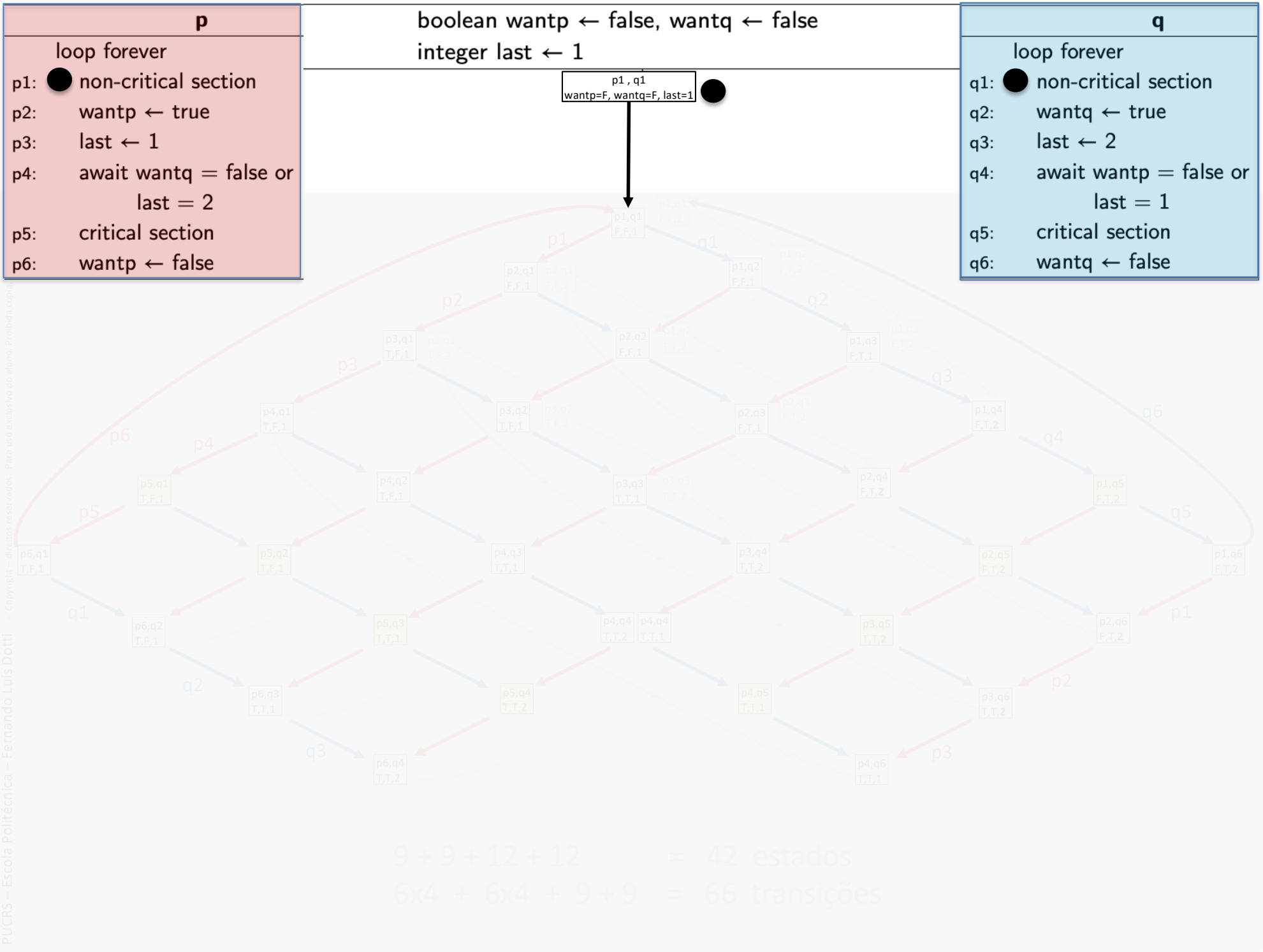
- Mais fácil achar problema do que mostrar que não há problema ...
- Como demonstrar

- Estratégia 1:
  - Modelo para as computações do sistema
    - a partir do estado inicial,
      enumerar estados e transições do algoritmo
      - estado inclui *program counter* de cada processo e variáveis
      - transições a partir de um estado:
        todas as ações possíveis a partir daquele estado
    - gera um diagrama de estados
  - Propriedades:
    - Exclusão Mútua:
      - avaliar se em algum estado possível
      - (p5 and q5) é verdade - então viola a exclusão mútua
    - Postergação:
      - avaliar se podemos ter ciclos em que um processo disputa a SC mas nunca acessa
    - Progresso:
      - avaliar se podemos ter:
        - » bloqueio (estado sem aresta de saída) ou
        - » situação em que um processo só entra na SC
          se outro processo *na seção não-crítica* faz algum passo.

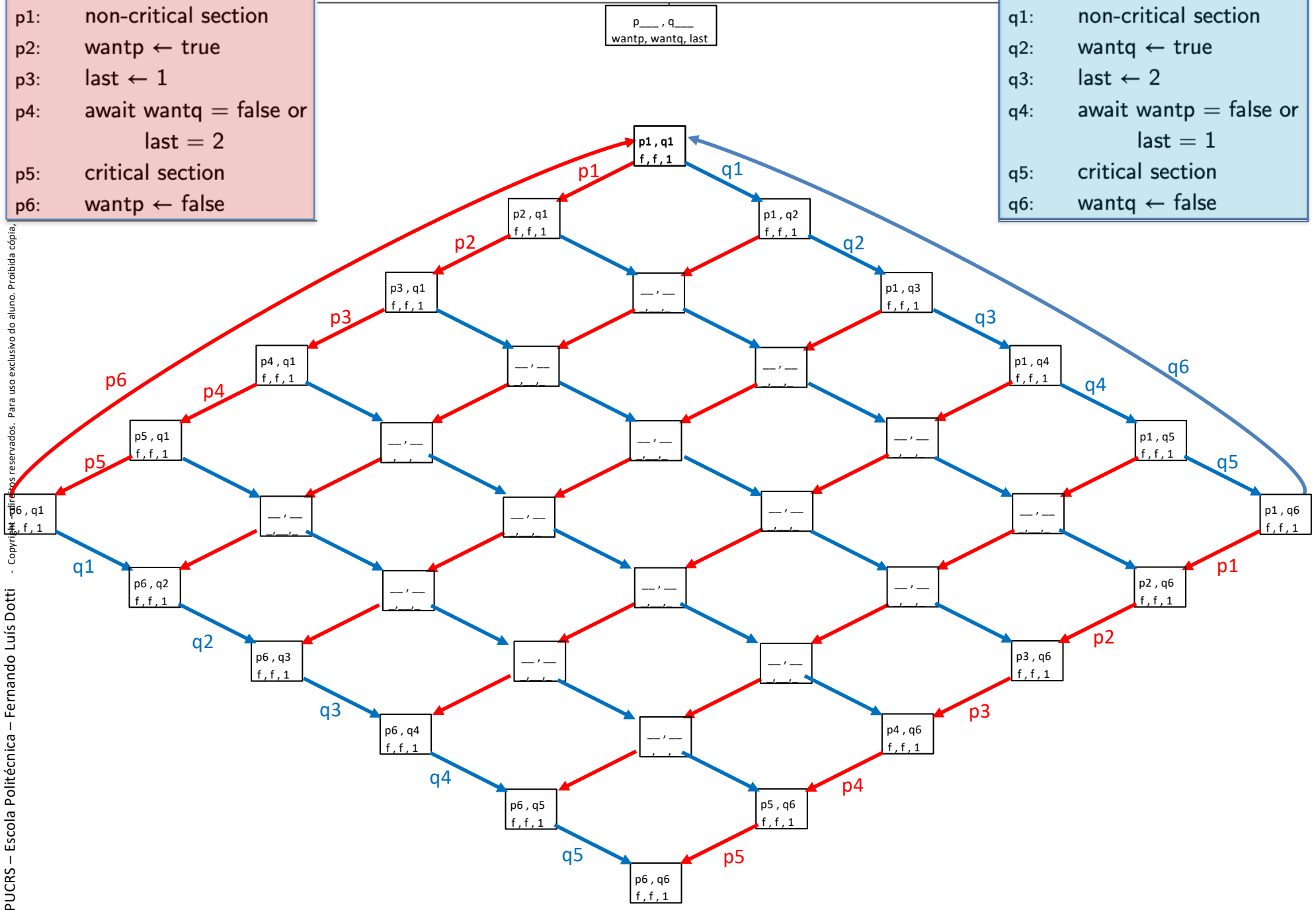
- Construção do Diagrama de Estados para Algoritmo de Peterson
  - exercício de interleaving







p	boolean wantp ← false, wantq ← false integer last ← 1	q
loop forever		loop forever
p1: non-critical section		q1: non-critical section
p2: wantp ← true		q2: wantq ← true
p3: last ← 1		q3: last ← 2
p4: await wantq = false or last = 2		q4: await wantp = false or last = 1
p5: critical section		q5: critical section
p6: wantp ← false		q6: wantq ← false



Exercício

- Para o algoritmo de Peterson para seção crítica para dois processos
  1. monte o diagrama de estados e transições
  2. com o diagrama, argumente sobre:
     1. exclusão mútua
     2. progresso
     3. espera limitada

Exercício

- exclusão mútua
  - não existe estado é (p5, q5)
  - eles não são alcançáveis

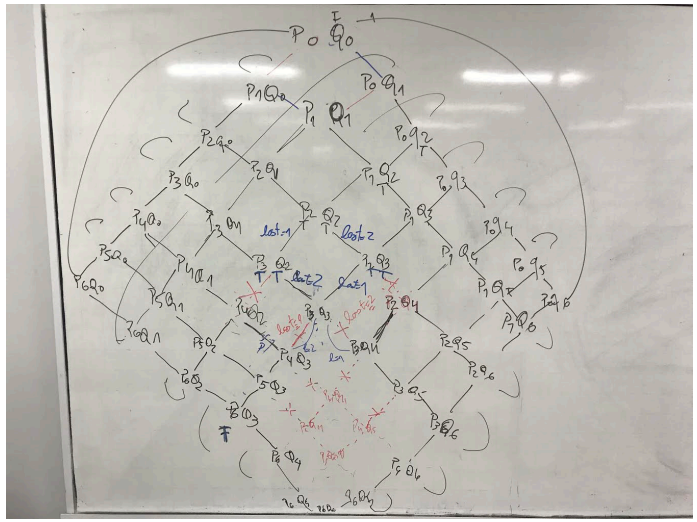
Exercício

- progresso: um processo não é impedido de entrar na SC crítica se outro processo não a disputa
  - para tal você deve avaliar as possíveis transições a partir do estado de interesse (um processo quer entrar e outro não)

Exercício

- espera limitada: se um processo disputa a SC então ele recebe o direito de acessar em um tempo limitado (não é indefinidamente postergado)
  - para tal você deve avaliar as possíveis transições a partir do estado de interesse
    (dois disputam SC, um recebe o direito, outro espera – a partir disso, o que esperou deve acessar a SC em um número finito de passos)

- A Estratégia 1, vista acima equivale à técnica de **“Model Checking”**
  - descrevemos algoritmo em uma linguagem de modelagem apropriada
  - um programa constrói o espaço de estados
  - especificamos as propriedades em uma linguagem (como lógica temporal, ou a mesma do modelo)
    e
    um programa verifica a propriedade sobre o modelo



Exemplos de uso de “**Model Checking**”

- Desenvolver modelo CSP do algoritmo de Peterson, enunciar propriedades em CSP, usar ferramenta FDR, e avaliar se o modelo respeita a propriedade (o modelo refina a propriedade)
  - CSP: Communicating Sequential Processes
  - FDR: Failures and Divergences Refinement
- Desenvolver modelo ProMELA, enunciar propriedades em lógica temporal (LTL), usar SPIN para verificar
  - ProMELA: Process/Protocol Meta-Language
  - LTL: Linear Temporal Logic
  - SPIN: Simple ProMELA Interpreter
- Desenvolver modelo na linguagem de SMV, enunciar propriedades em lógica temporal (CTL), usar nuSMV
  - Symbolic Model Verifier
  - Computation Tree Logic
  - nuSMV: ambiente de verificação SMV

- Estratégia 2:
  - Descreve-se as invariantes do sistema. Invariante é uma propriedade (que deve ser) válida para todos estados.
  - Por exemplo: not(p5 and q5)
    - Raciocínio por indução:
      - prova propriedade para estado inicial,
      - para toda transição possível, prova que, partindo-se de um estado que respeita a invariante, aquela transição gera um estado que respeita a invariante

- Estratégia 2 equivale à técnica de "prova dedutiva"
  - uso de *provadores de teoremas*
  - ferramentas de sw que *ajudam* na aplicação dos passos da dedução

- Exemplo de prova dedutiva:
  - a partir de 4.1, livro de Ben-Ari,
  - seção 4.5 do livro – algoritmo de Dekker

- Model Checking
  - totalmente automatizado
  - explosão do espaço de estados (crescimento exponencial)
  - trata modelos limitados – estado finito

- Prova Dedutiva
  - parcialmente automatizado
  - provas crescem linearmente com o número de ações do sistema
  - maior complexidade para usuário

Algoritmo de Peterson em Java

```java
/*
   Algoritmo de exclusao mutua por SW para dois processos,
   de Peterson, em Java, exemplificado com contador compartilhado.
   Disciplina:  Sistemas Operacionais
   PUCRS - Escola Politecnica Prof: Fernando Dotti

   Note o uso do modificador volatile.
   Retire as protecoes da sc e veja o resultado.
   Retire volatile das definicoes de Peterson e veja o resultado.
*/
// ============= Peterson ==============
// algoritmo de exclusao mutua para dois processos, de Peterson.
// metodos implementam protocolos de entrada e saida da SC
class Peterson {
    private volatile boolean[] flag = new boolean[2];
    private volatile int last; //

    public Peterson(){
        flag[0]=false;
        flag[1]=false;
        last=1;
    }

    public void lock(int id) {
     int j = 1 - id;
     flag[id] = true;
     last = id;
     while (!(!flag[j] || last == j)) {};
    }

    public void unlock(int id) {
     flag[id] = false;
    }
}

// ================================================
// aqui temos uma aplicacao de Peterson.  Um contador usado por
// multiplas threads. Cada thread invoca incr.   incr usa os protocolos
// do algoritmo de Peterson para proteger a variavel compartilhada
class CounterSC {
    private  int n;
    private Peterson sc;

    public CounterSC(){
        n = 0;
        sc = new Peterson();
    }

    public void incr(int id){
        sc.lock(id);
        n++;
        sc.unlock(id);
    }
    public int value() { return n; }
}
```

```java
// thread que usa o contador compartilhado.
// faz um nro de incrementos cfe sua instanciacao
class CounterThread extends Thread {

    private int id;
    private CounterSC c_sc;
    private int limit;

    public CounterThread(int _id, CounterSC _c_sc, int _limit){
        id = _id;
        c_sc = _c_sc;
        limit = _limit;
    }

    public void run() {
       for (int i = 0; i < limit; i++) {
             c_sc.incr(id);
      }
    }
}

// teste de peterson cria varias threads que vao
// acessar o mesmo contador,que usa o alg de peterson
class TestePeterson {
    public static void main(String[] args) {

      int nrIncr = 100000;

      CounterSC c = new CounterSC();
      CounterThread p = new CounterThread(0,c,nrIncr);
      CounterThread q = new CounterThread(1,c,nrIncr);

      p.start();
      q.start();
      try { p.join(); q.join(); }

      catch (InterruptedException e) { }
      System.out.println("The value of n is " + c.value());
    }

}
```

Algoritmo de Peterson em Go

```go
// Utilizado por Fernando Dotti
// Filename: peterson_spinlock-Vatom.go
// Use below command to run:
// go run peterson_spinlock-Vatom.go

package main

import (
    "fmt"
    "sync/atomic"
)

// ------------------------------------------------------------------------------------------------
// ------- variaveis e procedimentos do algoritmo de peterson para exclusao mutua por duas threads
// ------------------------------------------------------------------------------------------------

var flag [2]int // variaveis do algoritmo de peterson
var turn uint32

func lock_init() {
    flag[0] = 0
    flag[1] = 0 // inicia lock resetando o desejo de ambas threads de adquirirem o lock.
    turn = 0    // daa a vez a uma delas
}

func lock(self uint32) { // executado antes da secao critica
    flag[self] = 1                    // flag[self] = 1 diz que quer o lock
    atomic.StoreUint32(&turn, 1-self) // mas antes daa aa outra trhead a chance de adquirir o lock
    for flag[1-self] == 1 && turn == 1-self {
    }
}

func unlock(self uint32) { // executado depois da secao critica
    flag[self] = 0 // voce nao quer obter o lock, isso permite aa outra thread obter
}

// ------------------------------------------------------------------------------------------------
// ------- exemplo de uso do algoritmo de peterson em dois processos
// ------------------------------------------------------------------------------------------------

const MAX int = 2000000

var ans int = 0 // a variavel compartilhada!!   a ser protegida

func processo(self uint32, fin chan int) {
    fmt.Println("Processo entrou: ", self) // diz qual o identificador deste processo:   0 ou 1

    for i := 0; i < MAX; i++ { // entra e sai MAX vezes na SC
        lock(self)   // CODIGO DE ENTRADA NA SC
        ans++        // SECAO CRITICA
        unlock(self) // CODIGO DE SAIDA DA SECAO CRITICA
    }
    fin <- 1
}
```

```go
func main() {
    lock_init() // inicia as variaves de peterson
    fin := make(chan int)
    go processo(0, fin) // cria os processos
    go processo(1, fin)
    <-fin // espera fim de ambos
    <-fin
    fmt.Println("Valor do contador: ", ans, " | Valor esperado: ", MAX*2)
}
```

- Soluções de software para N processos

Algorithm 5.2: Bakery algorithm (N processes)

integer array[1..n] number ← [0,. . . ,0]

```
      loop forever
p1:       non-critical section
p2:       number[i] ← 1 + max(number)
p3:       for all other processes j
p4:           await (number[j] = 0) or (number[i] ≪ number[j])
p5:       critical section
p6:       number[i] ← 0
```

Algorithm 5.3: Bakery algorithm without atomic assignment

```
                boolean array[1..n] choosing ← [false,...,false]
                integer array[1..n] number ← [0,...,0]
    loop forever
p1:     non-critical section
p2:     choosing[i] ← true
p3:     number[i] ← 1 + max(number)
p4:     choosing[i] ← false
p5:     for all other processes j
p6:         await choosing[j] = false
p7:         await (number[j] = 0) or (number[i] ≪ number[j])
p8:     critical section
p9:     number[i] ← 0
```

- number[i] << number[j] =
  (number[i] << number[j]) or
  ((number[i] == number[j]) and i<j)

Algorithm 5.8: Lamport's one-bit algorithm

```
boolean array[1..n] want ← [false,...,false]

     loop forever
         non-critical section
p1:      want[i] ← true
p2:      for all processes j < i
p3:          if want[j]
p4:              want[i] ← false
p5:              await not want[j]
                 goto p1
p6:      for all processes j > i
p7:          await not want[j]
         critical section
p8:      want[i] ← false
```

- Provê exclusão mútua, livre de deadlock
- Mas starvation é possível !

- Qual a dificuldade das seções de entrada na seção crítica ?

- Observação:
  - Necessária operação que lê e escreve de forma atômica

- Suporte de operações atômicas em HW
  - Test and set (compartilhada, local)

```
atomic {
    local <- compartilhada
    compartilhada <- 1
}
```

  - Exchange ou swap(a,b)

Algorithm 3.11: Critical section problem with test-and-set

integer common ← 0

p	**q**
integer local1	integer local2
loop forever	loop forever
p1: non-critical section	q1: non-critical section
repeat	repeat
p2: test-and-set(common, local1)	q2: test-and-set(common, local2)
p3: until local1 = 0	q3: until local2 = 0
p4: critical section	q4: critical section
p5: common ← 0	q5: common ← 0

Algorithm 3.12: Critical section problem with exchange

integer common ← 1

	p		**q**
	integer local1 ← 0		integer local2 ← 0
	loop forever		loop forever
p1:	non-critical section	q1:	non-critical section
	repeat		repeat
p2:	exchange(common, local1)	q2:	exchange(common, local2)
p3:	until local1 = 1	q3:	until local2 = 1
p4:	critical section	q4:	critical section
p5:	exchange(common, local1)	q5:	exchange(common, local2)

Instruções de HW: vantagens

- Aplicável a qualquer número de processos em um único processador ou múltiplos processadores compartilhando memória principal
- Simples e fácil de verificar
- Pode ser usado para suportar várias seções críticas; cada seção crítica pode ser definida por sua própria variável

Instruções de HW: desvantagens

- Espera ocupada é empregada, portanto, enquanto um processo está aguardando acesso a uma seção crítica, ele continua a consumir tempo do processador
- Postergação é possível quando um processo deixa uma seção crítica e mais de um processo está esperando