Algoritmos Avançados para implementação de seção crítica

- Bibliografia básica:
  - Ben-Ari, M. Principles of concurrent and distributed programming. 2nd Ed. Addison-Wesley, 2006.

Algoritmo de Peterson

- Baseado no algoritmo de Dekker
- Utiliza esquema ”Busy-wait”
  - Espera ocupada é geralmente ineficiente
- Gary L. Peterson: "Myths About the Mutual Exclusion Problem", Information Processing Letters 12(3) 1981, 115–116

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

Lamport's Bakery Algorithm

- Leslie Lamport: A New Solution of Dijkstra's Concurrent Programming Problem Communications of the ACM 17, 8 (August 1974), 453-455

- Utiliza esquema ”Busy-wait”

- Fácil generalização para um número N de Threads

Lamport's Bakery Algorithm (cont.)

- Basea-se na ídeia de que:
  - Um processo que quer entrar na SC deverá receber um ticket sequencialmente numerado.
    - O valor numérico do ticket recebido deverá ser maior que o valor de qualquer outro ticket existente
  - O mesmo processo espera que o número do seu ticket seja o menor existente para entrar na SC
    - Semelhante a uma fila (FIFO)
- Como ocorre quando se espera por pão, com uma senha, em uma fila na padaria (*bakery*)

Usando apenas 2 processos

Algorithm 5.1: Bakery algorithm (two processes)	
integer np ← 0, nq ← 0	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: np ← nq + 1	q2: nq ← np + 1
p3: await nq = 0 or np ≤ nq	q3: await np = 0 or nq < np
p4: critical section	q4: critical section
p5: np ← 0	q5: nq ← 0

Generalização para utilizar 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
```

Explicações adicionais

- Onde:

```
for all other process
```

- Significa:

```
for j from 1 to N
    if j != i
```

- Onde:

```
(number[i] << number[j])
```

- Significa:

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

Algoritmo de Bakery menos restritivo

- O algoritmo anterior é uma simplificação do modelo original de L. Lamport.
  - Considera cada acesso a uma variável na memória (*Load/Store*) como atômico
  - Pode não corresponder a realidade, pois não se pode ter controle total do momento em que o SO faz *interleaving*
  - No algoritmo a seguir um processo/*thread* deve esperar que todos os outros *threads* tenham definido seus *tickets* para verificar se está ou não na sua vez.

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
```

Algoritmos rápidos para SC

- Evitar o cálculo do máximo no algoritmo de Lamport
  - Ineficiente se houverem muitos processos concorrentes
- Usa um número fixo de passos para alcançar a SC
  - Dois níveis (”portões”) de controle e contenção
- Fácil generalização

Utilizando 2 processos

Algorithm 5.4: Fast algorithm for two processes (outline)

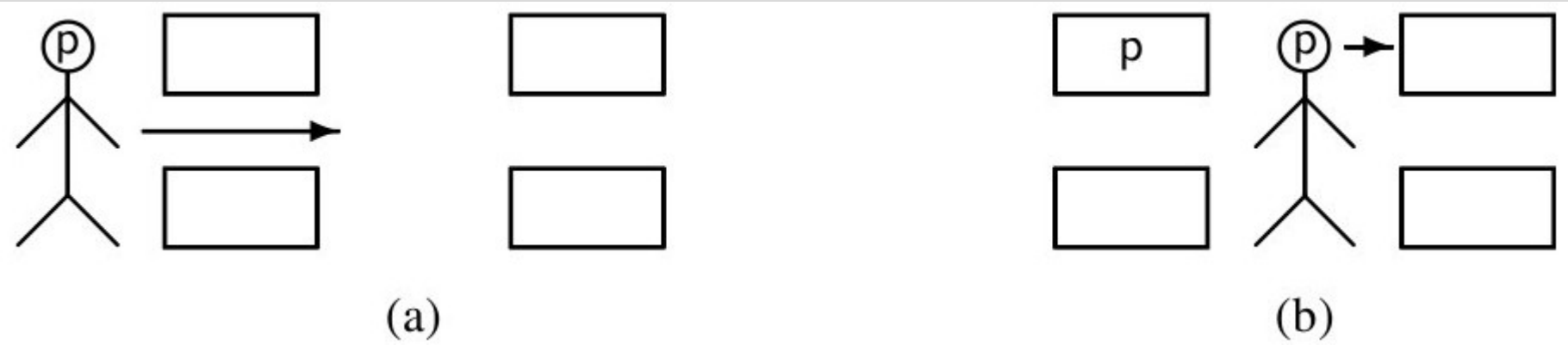
integer gate1 ← 0, gate2 ← 0

p		q	
	loop forever		loop forever
	non-critical section		non-critical section
p1:	gate1 ← p	q1:	gate1 ← q
p2:	if gate2 ≠ 0 goto p1	q2:	if gate2 ≠ 0 goto q1
p3:	gate2 ← p	q3:	gate2 ← q
p4:	if gate1 ≠ p	q4:	if gate1 ≠ q
p5:	if gate2 ≠ p goto p1	q5:	if gate2 ≠ q goto q1
	critical section		critical section
p6:	gate2 ← 0	q6:	gate2 ← 0

Representação gráfica

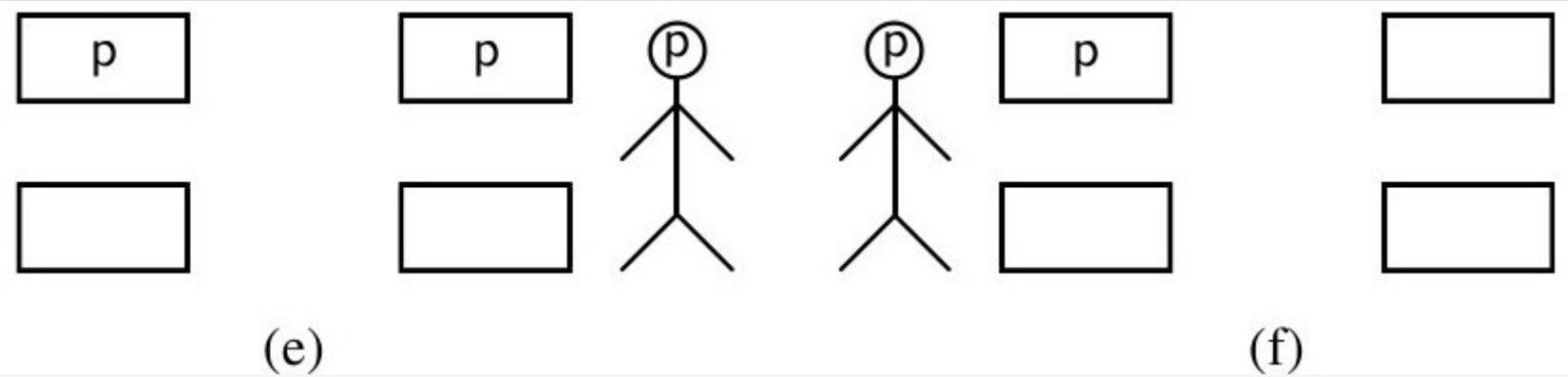
- Processo representado pela figura humana
- A SNC está a esquerda
- A SC está a direita
- Caixas representam portões ou níveis de acesso à SC

Entrada sem contenção



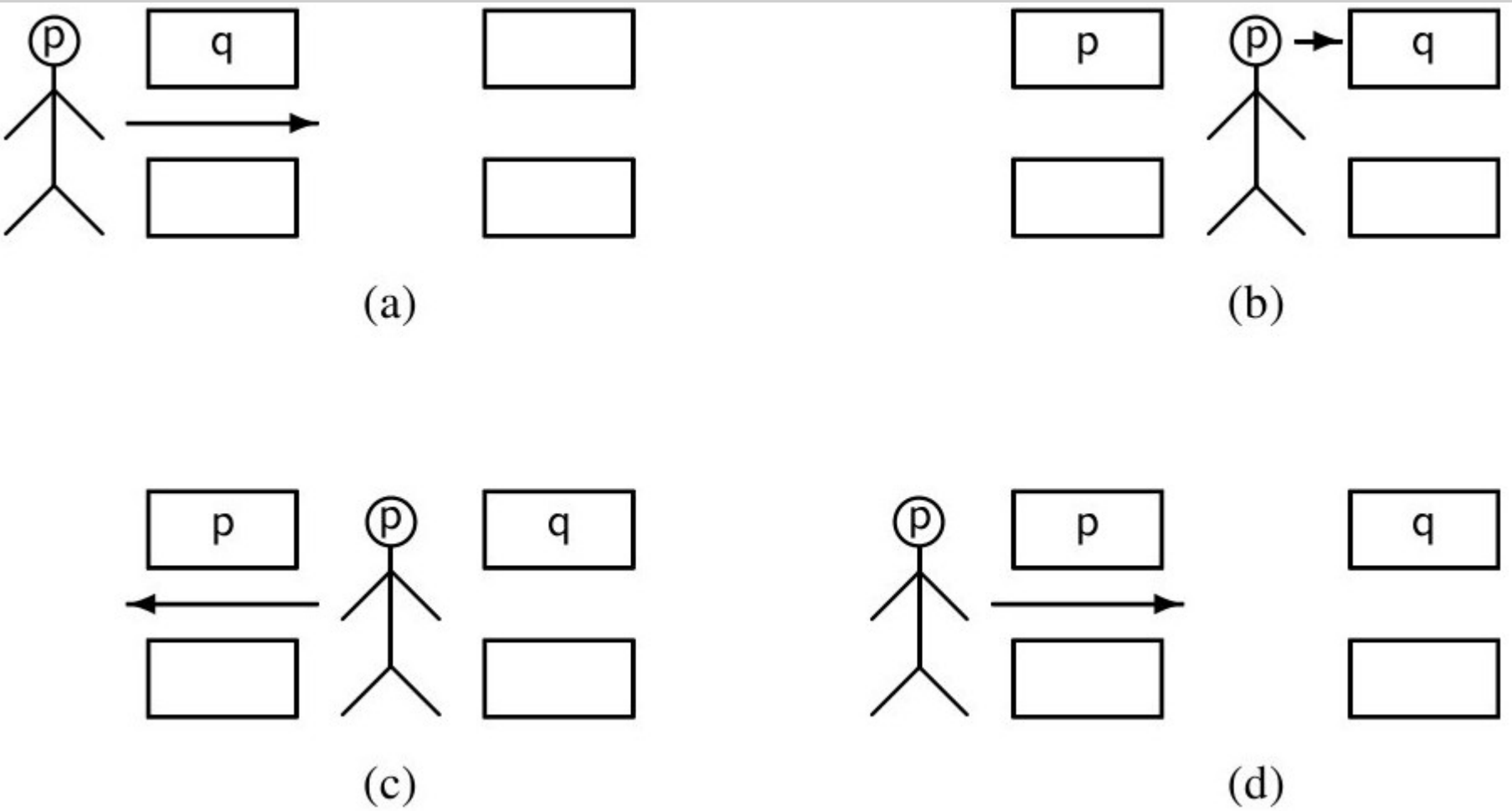
a) Processo pronto para entrar no primeiro portão e escrever seu ID
b) ID escrito no primeiro portão, examinando o segundo portão

c) Não tendo ninguem no segundo portão marca seu ID
b) Olha para o primeiro portão atraz e verifica se alguem tentou entrar



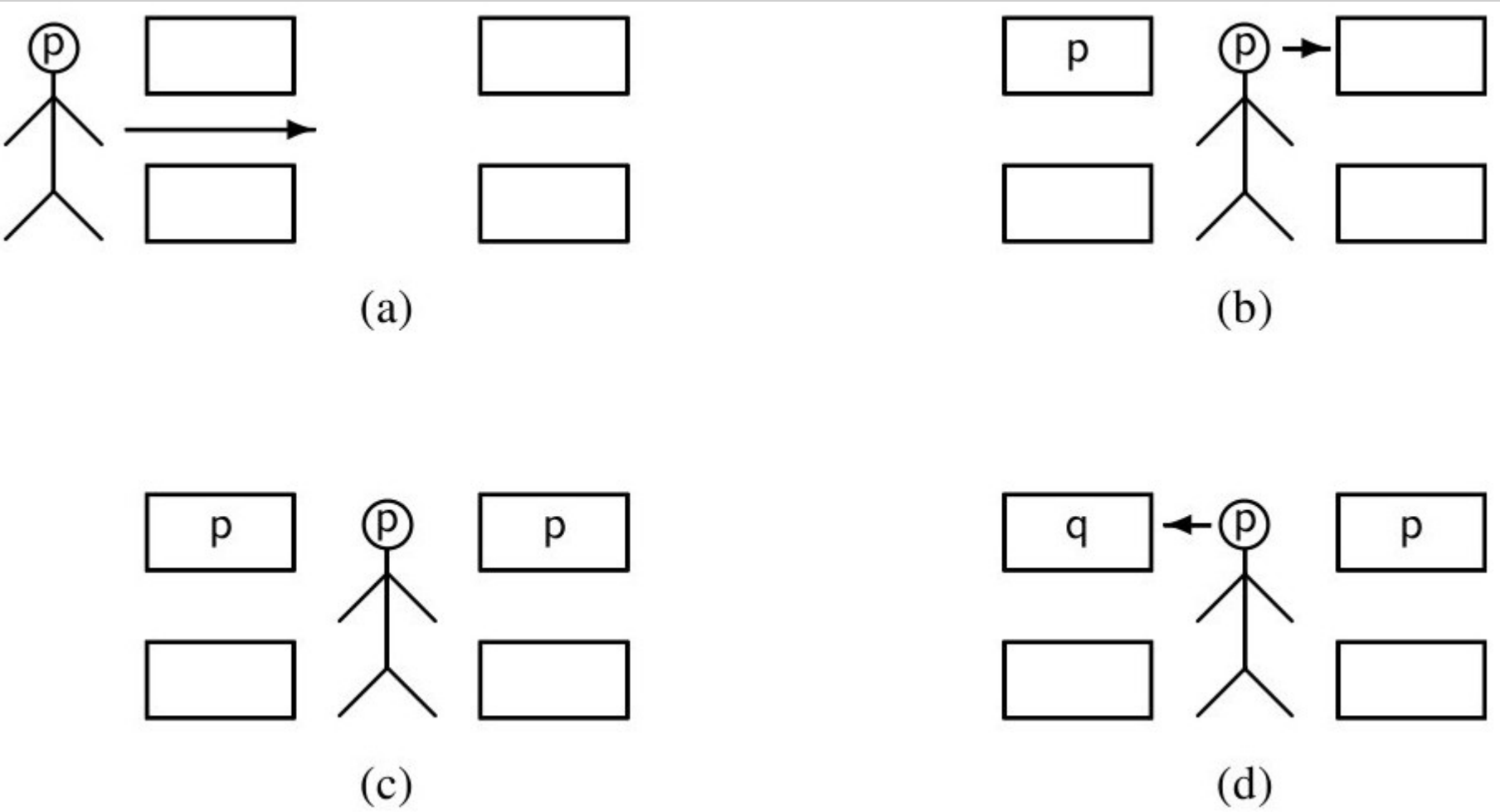
e) Processo na SC
f) Após sair da SC retira os IDs e volta a SNC

Sem contenção o acesso é muito rápido (custo de acessar a SC é de três instruções)



Ocorrência de contenção no Portão 2

a) Processo P tenta entrar na SC escrevendo ID no portão 1
b) Tenta acessar portão 2 e verifica que outro processo já marcou seu espaço
c) Processo P volta atraz para nova tentativa
d) similar ao passo (a)



Ocorrência de contenção no Portão 1

a) Processo P tenta entrar na SC
b) Escreve ID no portão 1
c) Verifica portão 2 e por estar vazio escreve ID
d) Verifica atraz o portão 1 e nota outro processo tentanto entrar

Ocorrência de contenção no Portão 1

Duas possibilidades:

a) ID do processo P ainda está no portão 2 e basta entrar na SC
b) ID de outro processo já está no portão 2 e deve-se voltar para nova tentativa

Algorithm 5.6: Fast algorithm for two processes

integer gate1 ← 0, gate2 ← 0
boolean wantp ← false, wantq ← false

p		q	
p1:	gate1 ← p	q1:	gate1 ← q
	wantp ← true		wantq ← true
p2:	if gate2 ≠ 0	q2:	if gate2 ≠ 0
	wantp ← false		wantq ← false
	goto p1		goto q1
p3:	gate2 ← p	q3:	gate2 ← q
p4:	if gate1 ≠ p	q4:	if gate1 ≠ q
	wantp ← false		wantq ← false
	await wantq = false		await wantp = false
p5:	if gate2 ≠ p goto p1	q5:	if gate2 ≠ q goto q1
	else wantp ← true		else wantq ← true
	critical section		critical section
p6:	gate2 ← 0	q6:	gate2 ← 0
	wantp ← false		wantq ← false

Generalizações

- Usar array lógico para ”wantx”

- Substituir a declaração:

```
await wanq=false
```

- Por:

```
for all other process j
    await wan[j]=false
```

Resumo

- Todos os algorimtos utilizam ”Busy-Wait” implementadas em linguagem de programação de alto nivel
- Vantagens:
  - Implementação algoritmica sem necessidade de interferencia a rotinas do SO
- Desvantagens:
  - O processo em espera poderia ser colocado em estado de ”espera” onde não gasta CPU
    - O gasto de CPU sem necessidade pode modificar as prioridades do SO na alocação de tarefas

Alternativas

- Usar recuros do SO para controlar o acesso a SC
  - Exemplo: semáforos, monitores, etc
  - Programação de mais baixo nível
  - Não causam ”busy-wait”