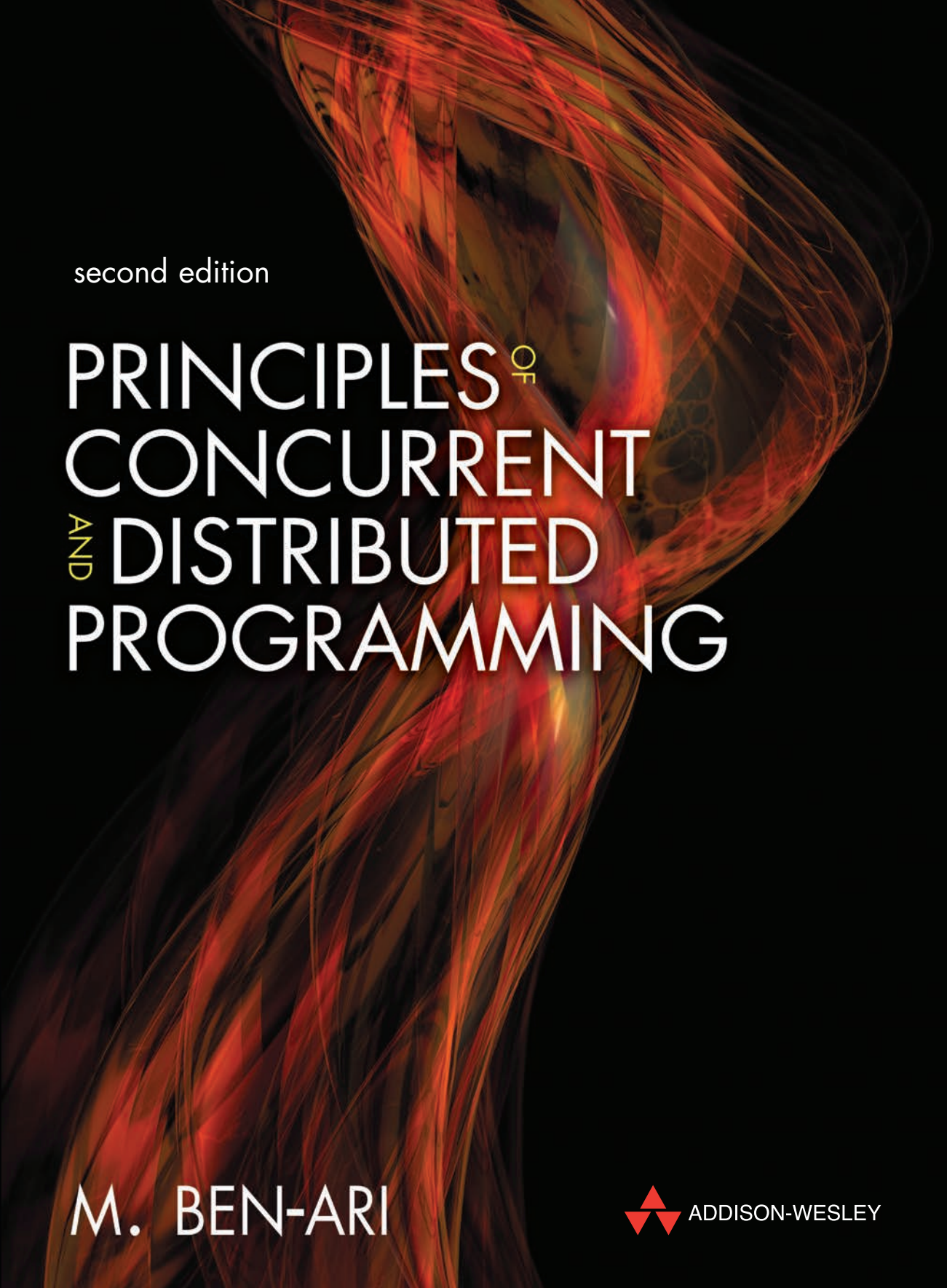
second edition
PRINCIPLES OF CONCURRENT AND DISTRIBUTED PROGRAMMING
M. BEN-ARI
ADDISON-WESLEY

Principles of Concurrent and Distributed Programming

Visit the *Principles of Concurrent and Distributed Programming, Second Edition* Companion Website at **www.pearsoned.co.uk/ben-ari** to find valuable **student** learning material including:

- Source code for all the algorithms in the book
- Links to sites where software for studying concurrency may be downloaded.

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
```

Algorithm 5.3 differs from Algorithm 5.2 by the addition of a boolean array choosing. If choosing[i] is true, a process is in the act of choosing a ticket number, and other processes must wait for the choice to be made before comparing numbers.

Each global variable is written to by exactly one process, so there are no cases in which multiple write operations to the same variable can overlap. It is also reasonable to assume that if a set of read operations to a variable does not overlap a write operation to that variable, then they all return the correct value. However, if read operations overlap a write operation, it is possible that inconsistent values may be obtained. Lamport has shown that the bakery algorithm is correct even if such read operations return arbitrary values [36].

5.4 Fast algorithms

In the bakery algorithm, a process wishing to enter its critical section must read the values of all the elements of the array number to compute the maximum ticket number, and it must execute a loop that contains await statements that read the values of all the elements of the two arrays. If there are dozens of processes in the system, this is going to be extremely inefficient. The overhead may be unavoidable: if, in general, processes attempt to enter their critical sections at short intervals, there will be a lot of contention, and a process really will have to query the state of all the other processes. If, however, contention is low, it makes sense to search for an algorithm for the critical section problem that is very efficient, in the sense that only if there is contention does the process incur the significant overhead of querying the other processes.

An algorithm for the critical section problem is *fast*, if in the absence of contention a process can access its critical section by executing pre- and postprotocols consisting of a *fixed* (and small) number of statements, none of which are await statements. The first fast algorithm for mutual exclusion was given by Lamport in [40], and this paper initiated extensive research into algorithms that are efficient under various assumptions about the characteristics of a system, such as the amount of contention and the behavior of the system under errors. Here we will describe and prove Lamport's algorithm restricted to two processes and then note the changes needed for an arbitrary number of processes.

Outline of the fast algorithm

Here is the outline of the algorithm, where we assume that the process identifiers p and q are encoded as nonzero integer constants so that their values can be assigned to variables and compared with the values of these variables:

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

(To simplify the proof of correctness, we have given an abbreviated algorithm as was done in the previous chapters.)

The concept of the algorithm is displayed graphically in Figures 5.1–5.3. A process is represented by a stick figure. The non-critical section is at the left of each diagram; a process must pass through two gates to reach the critical section on the right side of the diagram. Figure 5.1 shows what happens in the absence of contention. The process enters the first gate, writing its ID number on the gate [p1, (a)]. (The notation correlates the line number p1 in the algorithm with diagram (a) in the figure.) It then looks at the second gate [p2, (b)]; in the absence of contention, this gate will not have an ID written upon it, so the process writes its ID on that gate also [p3, (c)]. It then looks back over its shoulder at the first gate to check

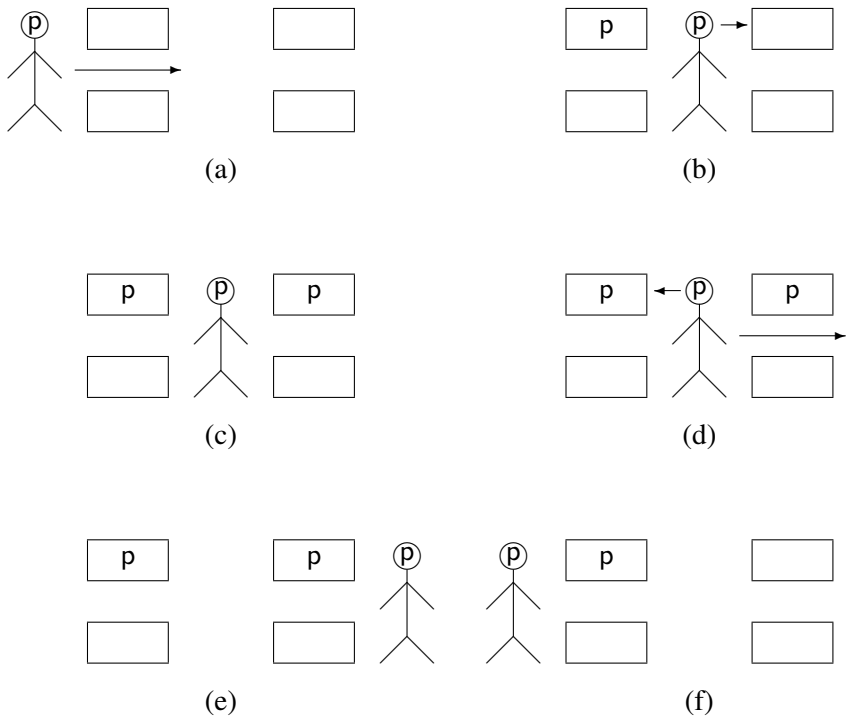

Figure 5.1: Fast algorithm—no contention

if its ID is still written there [p4, (d)]. If so, it enters the critical section [p6, (e)]. Upon leaving the critical section and returning to the non-critical section, it erases its ID from the second gate [p6, (f)].

In the absence of contention, the algorithm is very efficient, because a process can access its critical section at the cost of three statements that assign a constant to a global variable (p1, p3, p6), and two if statements that check that the value of a global variable is not equal to a constant (p2, p4).

The case of contention at the second gate is shown in Figure 5.2. The process enters the first gate [p1, (a)], but when it checks the second gate [p2, (b)], it sees that the other process q has already written its ID, preparing to enter the critical section. Since q has gotten through both gates sooner, process p returns to before the first gate [p2, (c)] to try again [p1, (d)].

In the case of contention at the first gate, the algorithm is a bit more complex (Figure 5.3). Initially, the algorithm proceeds as before [p1, (a)], [p2, (b)], [p3, (c)], until looking back over its shoulder, process p perceives that process q has entered the first gate and written its ID [p4, (d)]. There are now two possibilities:

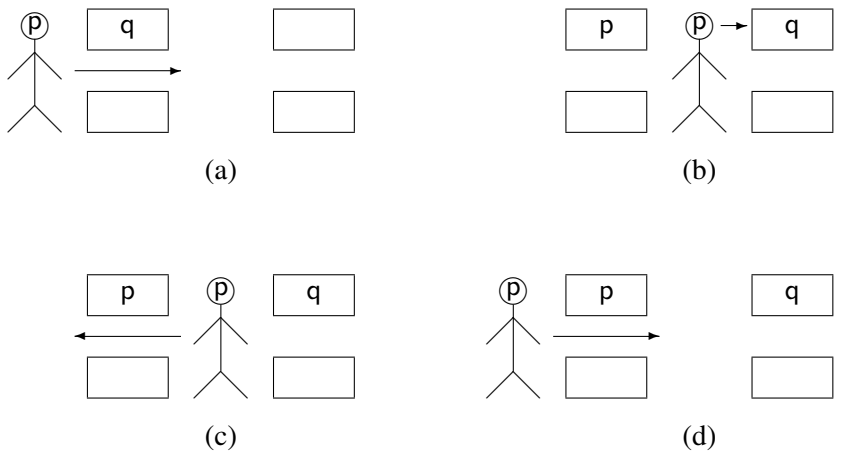

Figure 5.2: Fast algorithm—contention at gate 2

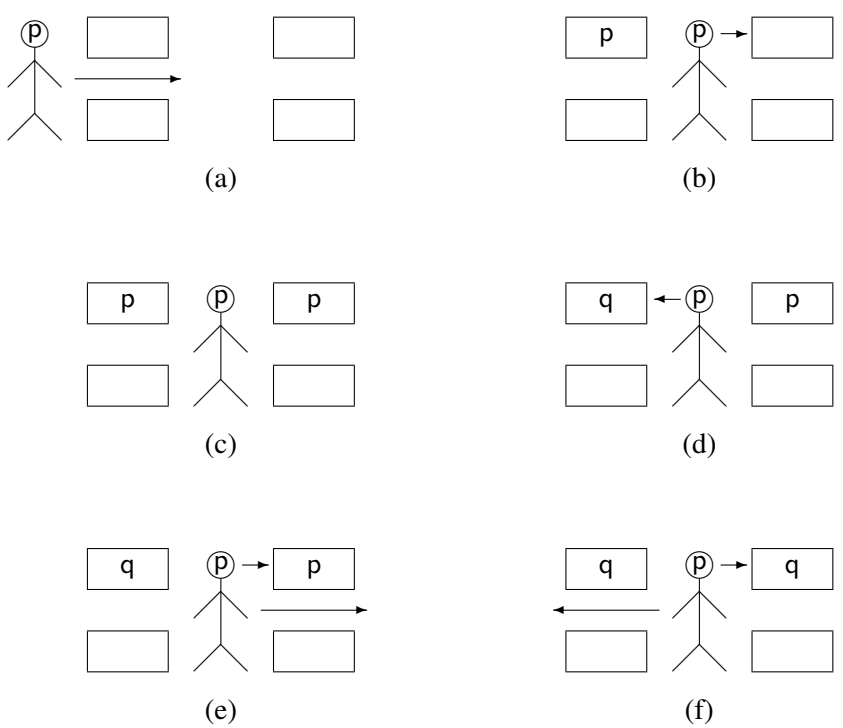

Figure 5.3: Fast algorithm—contention at gate 1

the ID of process p is still written on the second gate; if so, it continues into the critical section [p5, (e)]. Or, process q has passed process p and written its ID on the second gate; if so, the new process defers and returns to the first gate [p5, (f)].

The outline of the algorithm is not correct; in the exercises you are asked to find a scenario in which mutual exclusion is not satisfied. We will now partially prove the outline of the algorithm and then modify the algorithm to ensure correctness.

Partial proof of the algorithm

To save turning pages, here is the algorithm again:

Algorithm 5.5: Fast algorithm for two processes (outline)

integer gate1 $\leftarrow$ 0, gate2 $\leftarrow$ 0

	p		**q**
	loop forever		loop forever
	non-critical section		non-critical section
p1:	gate1 $\leftarrow$ p	q1:	gate1 $\leftarrow$ q
p2:	if gate2 $\neq$ 0 goto p1	q2:	if gate2 $\neq$ 0 goto q1
p3:	gate2 $\leftarrow$ p	q3:	gate2 $\leftarrow$ q
p4:	if gate1 $\neq$ p	q4:	if gate1 $\neq$ q
p5:	if gate2 $\neq$ p goto p1	q5:	if gate2 $\neq$ q goto q1
	critical section		critical section
p6:	gate2 $\leftarrow$ 0	q6:	gate2 $\leftarrow$ 0

Lemma 5.4 The following formulas are invariant:

$$p5 \wedge gate2 = p \quad \rightarrow \quad \neg(q3 \vee q4 \vee q6) \tag{5.1}$$
$$q5 \wedge gate2 = q \quad \rightarrow \quad \neg(p3 \vee p4 \vee p6) \tag{5.2}$$

We *assume* the truth of this lemma and use it to prove the following lemma:

Lemma 5.5 The following formulas are invariant:

$$p4 \wedge gate1 = p \quad \rightarrow \quad gate2 \neq 0 \tag{5.3}$$
$$p6 \quad \rightarrow \quad gate2 \neq 0 \wedge \neg q6 \wedge (q3 \vee q4 \rightarrow gate1 \neq q) \tag{5.4}$$
$$q4 \wedge gate1 = q \quad \rightarrow \quad gate2 \neq 0 \tag{5.5}$$
$$q6 \quad \rightarrow \quad gate2 \neq 0 \wedge \neg p6 \wedge (p3 \vee p4 \rightarrow gate1 \neq p) \tag{5.6}$$

Mutual exclusion follows immediately from invariants (5.4) and (5.6).

Proof: By symmetry of the algorithm, it is sufficient to prove that (5.3) and (5.4) are invariant. Trivially, both formulas are true initially.

Proof of 5.3: Executing p3 makes $p4$, the first conjunct of the antecedent, true, but it also makes $gate2 = p$, so the consequent is true. Executing statement p1 makes the second conjunct of the antecedent $gate1 = p$ true, but $p4$, the first conjunct, remains false. Executing q6 while process p is at p4 can make the consequent false, but by the inductive hypothesis, (5.6) is true, so $p3 \vee p4 \rightarrow gate1 \neq p$, and therefore the antecedent remains false.

Proof of 5.4: There are two transitions that can make the antecedent $p6$ true—executing p4 or p5 when the conditions are false:

p4 to p6: By the if statement, $gate1 \neq p$ is false, so $gate1 = p$ is true; therefore, by the inductive hypothesis for (5.3), $gate2 \neq 0$, proving the truth of the first conjunct of the consequent. Since $gate1 = p$ is true, $gate1 \neq q$, so the third conjunct is true. It remains to show $\neg q6$. Suppose to the contrary that $q6$ is true. By the inductive hypothesis for (5.6), if both $q6$ and $p4$ are true, then so is $gate1 \neq p$, contradicting $gate1 = p$.

p5 to p6: By the if statement, $gate2 = p$, so $gate2 \neq 0$, proving the truth of the first conjunct. By the assumed invariance of (5.1), $\neg(q3 \vee q4 \vee q6)$, so $\neg q6$, which is the second conjunct, is true, as is the third conjunct since its antecedent is false.

Assume now that the antecedent is true; there are five transitions of process q that can possibly make the consequent false:

q6: This statement cannot be executed since $\neg q6$ by the inductive hypothesis.

q4 to q6: This statement cannot be executed because $q3 \vee q4 \rightarrow gate1 \neq q$ by the inductive hypothesis.

q5 to q6: This statement cannot be executed. It can only be executed if $gate2 \neq q$ is false, that is, if $gate2 = q$. By the assumed invariance of (5.2) this can be true only if $\neg(p3 \vee p4 \vee p6)$, but $\neg p6$ contradicts the truth of the antecedent $p6$.

q1: This can falsify $gate1 \neq q$, but trivially, $q3 \vee q4$ will also be false.

q2 to q3: This can falsify the third conjunct if $gate1 \neq q$ is false. But the statement will be executed only if $gate2 = 0$, contradicting the inductive hypothesis of the first conjunct. ∎

Lemma 5.4 is not true for Algorithm 5.5. The algorithm must be modified so that (5.1) and (5.2) are invariant without invalidating the invariants we have already proved. This is done by adding local variables wantp and wantq with the usual meaning of wanting to enter the critical section (Algorithm 5.6). Clearly, these additions do not invalidate the proof of Lemma 5.5 because the additional variables