INFO-F404, Operating Systems II

Joël GOOSSENS

Université libre de Bruxelles

A) Concurrency

Main reference

- M. BEN-ARI, "Principles of Concurrent and Distributed Programming", Addison-Wesley, 2006.
- The algorithms and diagrams shown in this chapter are mainly from that book.

Introduction

- An “ordinary” program is said to be **sequential**. After compilation, it is made up of a **sequence** of instructions that are executed in order. They may access main memory (RAM) or secondary memory (disks).
- A “concurrent” program is a set of sequential programs that may be executed in **parallel**.
- The word **parallel** is used to denote hardware architectures made up of several processing units.
- The word **concurrent** is reserved for designating cases where there can **potentially** be parallelism, i.e. cases where execution of different programs may be interlaced (through time sharing).
- Concurrency is thus an **abstraction**.

Concurrent execution I

- A concurrent program is a set of processes.
- A process is a sequence of **atomic** instructions (will be defined later)
- The execution of a concurrent program produces a sequence of atomic instructions among all possible interleavings.
- Example: if a process p is made of instructions p_1 and p_2, and a process q is made of instructions q_1 and q_2, the possible concurrent execution scenarios of p and q are the following:

$$
\begin{aligned}
&p1 \rightarrow q1 \rightarrow p2 \rightarrow q2,\\
&p1 \rightarrow q1 \rightarrow q2 \rightarrow p2,\\
&p1 \rightarrow p2 \rightarrow q1 \rightarrow q2,\\
&q1 \rightarrow p1 \rightarrow q2 \rightarrow p2,\\
&q1 \rightarrow p1 \rightarrow p2 \rightarrow q2,\\
&q1 \rightarrow q2 \rightarrow p1 \rightarrow p2.
\end{aligned}
$$

Concurrent execution II

Let's take a look at a first concurrent program:

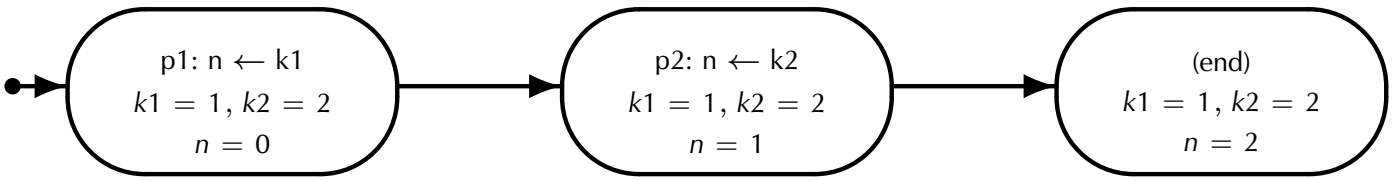
Trivial concurrent program	
integer n $\leftarrow$ 0	
p	q
integer k1 $\leftarrow$ 1	integer k2 $\leftarrow$ 2
p1: n $\leftarrow$ k1	q1: n $\leftarrow$ k2

State diagrams I

- We shall describe concurrent program execution through the use of state diagrams.
- Let's first have a look at the particular case of a sequential program:

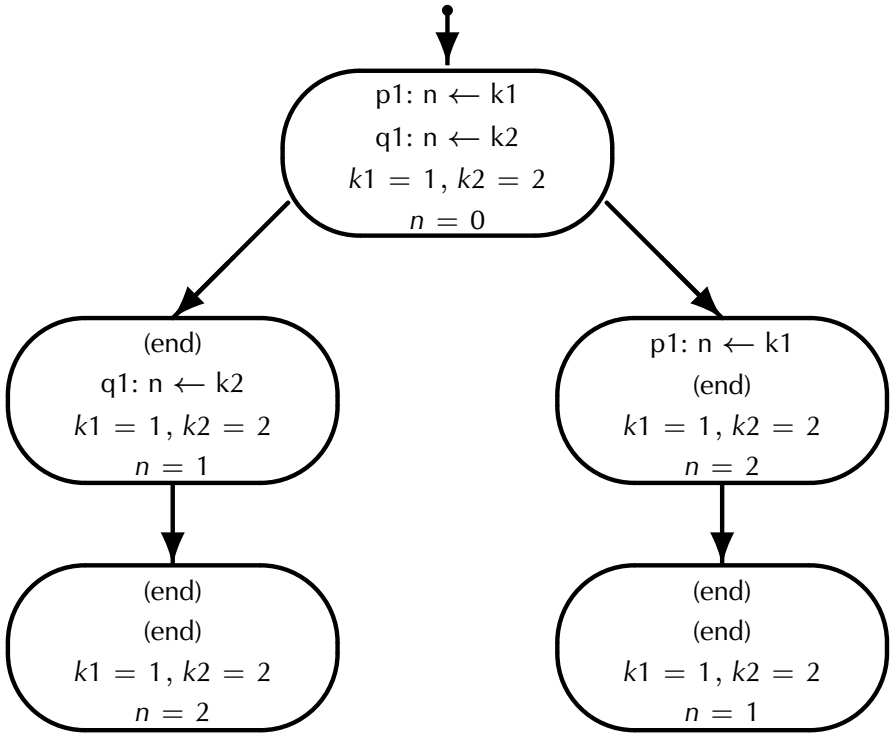
Trivial sequential program	
	integer n ← 0
	integer k1 ← 1
	integer k2 ← 2
p1:	n ← k1
p2:	n ← k2

The following state diagram shows the possible states and state transitions of the program:



- For the previously shown concurrent algorithm, there were two processes. The diagram shows all possible interleavings:

State diagrams II



Scenario

- A **scenario** is a sequence of states. We shall model them using tables.
- In a given state, we may have to choose the instruction to be next executed. We shall denote the choice by using **bold text**.

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

Atomic instructions I

- An instruction is atomic in the sense where it is always executed completely without being interleaved with other processes.
- Consequently, if two atomic instructions are executed simultaneously, the result will be the same as if they would be executed sequentially, in any order.
- In the following algorithms, labeled instructions are assumed to be atomic:

Atomic assignment statements	
integer n $\leftarrow$ 0	
p	q
p1: $n \leftarrow n + 1$	q1: $n \leftarrow n + 1$

Atomic instructions II

- There are two possible scenarios:

Process p	Process q	n
p1: n←n+1	q1: n←n+1	0
(end)	**q1: n←n+1**	1
(end)	(end)	2

Process p	Process q	n
p1: n←n+1	**q1: n←n+1**	0
p1: n←n+1	(end)	1
(end)	(end)	2

- In both cases, the final value of n is 2. The algorithm is thus correct under postcondition $n = 2$.
- Let's change the algorithm slightly:

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

There are correct scenarios under postcondition $n = 2\ldots$

Atomic instructions III

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

However, some scenarios violate that postcondition:

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

Atomic instructions IV

- In our study, assignment statements and evaluation of boolean conditions will be considered atomic.
- This is unrealistic in practice, but gives us a simple model to examine more complex cases where instruction interleaving occurs at machine code (i.e. assembly language) level.

B) Critical sections

- The objective of this section is to solve the critical section problem with two processes. We'll study DEKKER's algorithm in particular.
- DEKKER's algorithm is seldom used in practice, but studying it will allow us to showcase frequent design errors in concurrent algorithms.

Problem definition I

- Each of N processes executes an infinite loop split into two parts:
  - critical section
  - non-critical section
- A solution to the problem will be correct if the following properties hold:
  - **Mutual exclusion**: execution of critical sections must never be interleaved;
  - **No deadlocks**: we must avoid situations where processes wait for each other.
  - **No starvation**: each process wishing to enter its critical section must be allowed to do so without having to wait indefinitely.
- To solve the problem, we will define synchronization mechanisms made of a protocol that precedes the critical section (preprotocol) and a second protocol that executes after the critical section (postprotocol). The general structure is the following:

Problem definition II

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

First protocol attempt I

First attempt	
integer turn $\leftarrow$ 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 $\leftarrow$ 2	q4: turn $\leftarrow$ 1

- We can easily see that mutual exclusion is ensured.

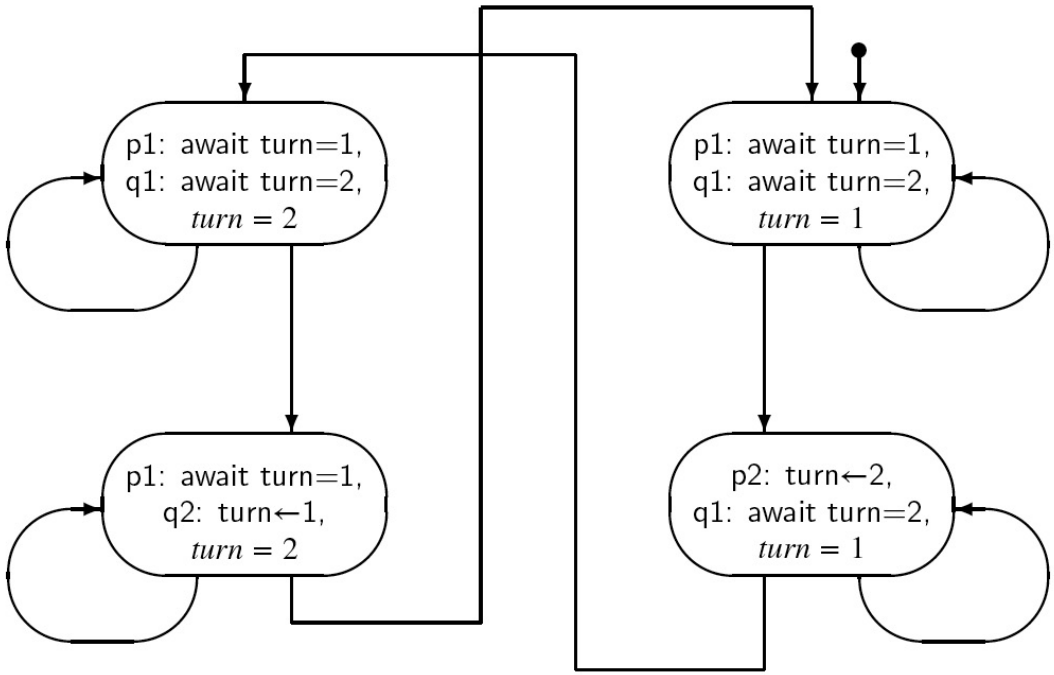
First protocol attempt II

- We can also check that deadlocks are avoided by reasoning over an abbreviated version of the algorithm:

First attempt (abbreviated)	
integer turn $\leftarrow$ 1	
p	q
loop forever	loop forever
p1: await turn $=$ 1	q1: await turn $=$ 2
p2: turn $\leftarrow$ 2	q2: turn $\leftarrow$ 1

- We can also check the following state diagram:

First protocol attempt III



- However, there are situations where p wishes to enter its critical section but is forced to wait indefinitely (i.e. starves) because q stays in its non-critical section, never setting the turn variable to 1.
- The issue arises because we've made no hypotheses on the progression of non-critical sections.

Second protocol attempt I

- Our first attempt was flawed due to the fact that both processes assign and test the same global variable. If a process dies, the other inevitably starves.
- To fix this problem, we'll try introducing variables that are local to each process (one per process).
- The boolean variable "wanti" indicates whether process *i* wishes to enter its critical section.
- This variable remains true until the process exits its critical section.

Second protocol attempt II

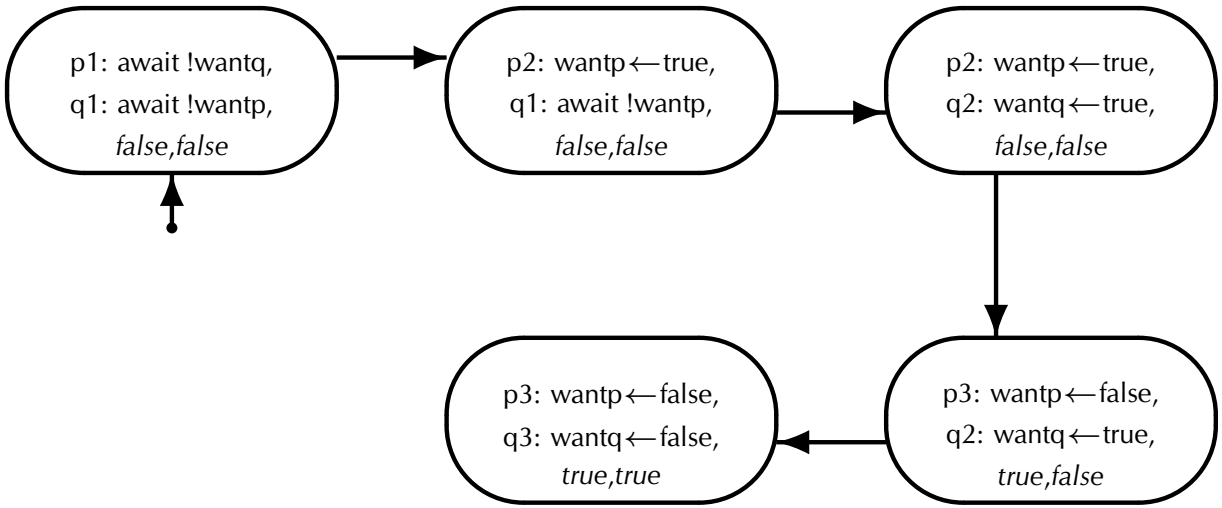
- This solution does not suffer from the starvation problem caused by our first attempt:

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

Second protocol attempt III

- Let's have a look at the state diagram for the following abbreviated version:

Second attempt (abbreviated)	
boolean wantp ← false, wantq ← false	
p	q
loop forever	loop forever
p1: await wantq = false	q1: await wantp = false
p2: wantp ← true	q2: wantq ← true
p3: wantp ← false	q3: wantq ← false



Second protocol attempt IV

- The state diagram shows that our second attempt is flawed as well. Indeed, mutual exclusion is violated, as can be seen in state (p3,q3,true,true).

Third protocol attempt I

- Our third attempt recognizes that the "await" instruction should be part of the critical section, which amounts to moving the variable assignment before the "await" instruction:

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

Third protocol attempt II

- Unfortunately, this protocol causes deadlocks to arise:

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*

- Deadlocks are situations where the system is frozen. We can distinguish them from **livelocks** where processes execute instructions but are otherwise stuck (i.e. nothing useful gets done).

Fourth protocol attempt I

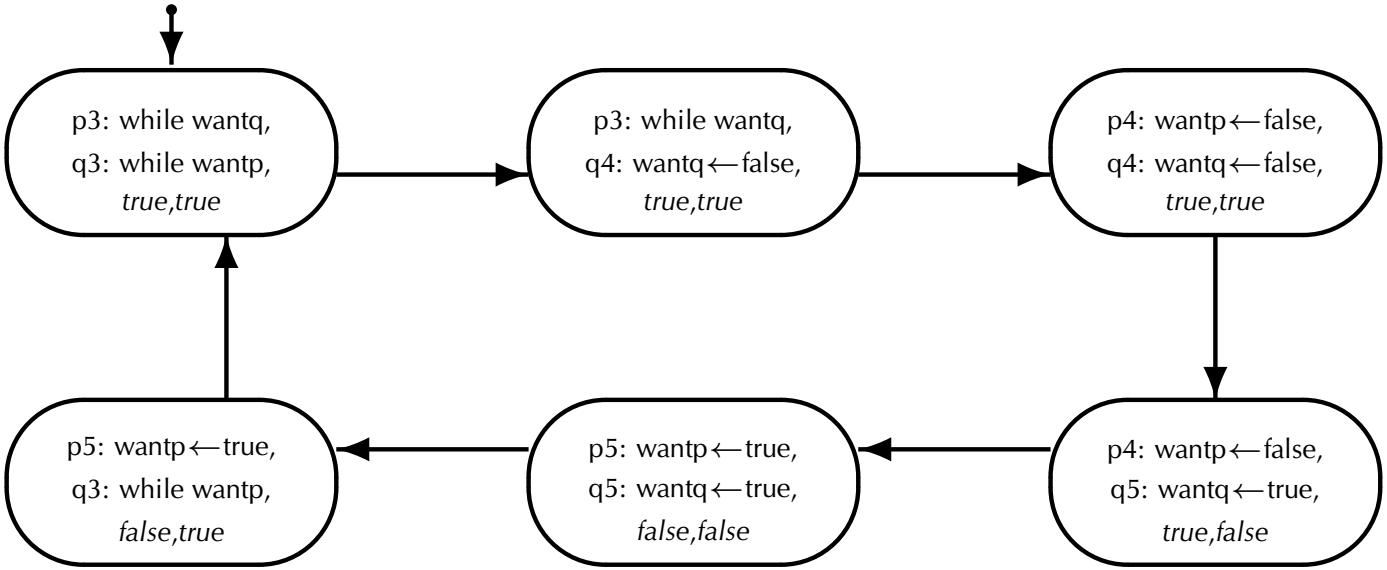
- To fix the problem of our third attempt, we'll try adding two assignments in each process that make no sense in a sequential program but that will allow to tell the other process the wish to enter the critical section:

Fourth 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: while wantq	q3: while wantp
p4: wantp ← false	q4: wantq ← false
p5: wantp ← true	q5: wantq ← true
p6: critical section	q6: critical section
p7: wantp ← false	q7: wantq ← false

- We could show, through a state diagram, that mutual exclusion is guaranteed and that no deadlocks can arise.

Fourth protocol attempt II

- Unfortunately, this attempt also suffers from starvation:



- If we have "perfect" interleaving, i.e. each instruction of p is followed by an instruction of q and vice versa, then both processes will starve indefinitely!

Dekker's algorithm I

- Dekker's algorithm to solve the critical section problem is, in essence, a combination of our first and fourth attempts:

Dekker's algorithm	
boolean wantp ← false, wantq ← false ; integer turn ← 1	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: while wantq	q3: while wantp
p4: if turn = 2	q4: if turn = 1
p5: wantp ← false	q5: wantq ← false
p6: await turn = 1	q6: await turn = 2
p7: wantp ← true	q7: wantq ← true
p8: critical section	q8: critical section
p9: turn ← 2	q9: turn ← 1
p10: wantp ← false	q10: wantq ← false

Dekker's algorithm II

- Dekker's solution is similar to our fourth attempt. The algorithms differ in the fact that the right to enter the critical section is explicitly passed between processes.
- Dekker's algorithm is valid: it guarantees mutual exclusion and the absence of both deadlocks and starvation.

C) Critical sections with N processes

- We will study two algorithms that solve the critical section problem.
- These algorithms are important, as they give a solution for the general case of an arbitrary number N of processes.
- These algorithms are the results of research by Leslie LAMPORT.

Bakery algorithm I

- The idea behind this algorithm is to give **tickets** to each process, like a bakery would give numbered tickets to its clients to serve them in order. The ticket numbers increase, and we serve the waiting client whose ticket number is the smallest.
- We'll first examine a simplified version for two processes:

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

Bakery algorithm II

- np and nq represent the ticket number of p and q respectively. A nil value indicates that the process does not currently wish to enter its critical section. If ticket numbers are identical (and not nil), we arbitrarily choose to give p higher priority.
- The bakery algorithm is valid for two processes: it guarantees mutual exclusion and the absence of both deadlocks and starvation.
- We will now look at the algorithm for N processes:

Bakery algorithm (N processes)
integer array[1..n] number $\leftarrow$ [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
```

Bakery algorithm III

- Each process executes the same algorithm, with only the value of i being different. i is a constant which acts as a process identifier.
- (number[i] $\ll$ number[j]) is an abbreviation for (number[i] $<$ number[j]) or ((number[i] $=$ number[j]) and (i$<$j)).
- Each process picks a number greater than the current maximum ticket number.
- A process may enter its critical section if it has the smallest number. In case of a tie, the process with the lowest identifier gets higher priority.
- This version of the algorithm makes an unrealistic hypothesis by assuming that computing the maximum value of an array is an atomic instruction. We will lift this assumption at the cost of simplicity:

Bakery algorithm IV

Bakery algorithm without atomic assignment	
boolean array[1..n] choosing $\leftarrow$ [false,. . . ,false]	
integer array[1..n] number $\leftarrow$ [0,. . . ,0]	
	loop forever
p1:	non-critical section
p2:	choosing[i] $\leftarrow$ true
p3:	number[i] $\leftarrow$ 1 + max(number)
p4:	choosing[i] $\leftarrow$ false
p5:	for all *other* processes j
p6:	await choosing[j] = false
p7:	await (number[j] = 0) or (number[i] $\ll$ number[j])
p8:	critical section
p9:	number[i] $\leftarrow$ 0

- This version uses a boolean array. If choosing[i] is true, it implies that process *i* is currently computing a ticket number. The other processes have to wait until the number has been computed.

Bakery algorithm V

- Each global variable is modified by one and one only process. Thus, there cannot be any overlapping writes.
- However, reads may overlap. LAMPORT has proved that the algorithm remains valid nonetheless (see [1] for details).

Related course

In the “Software and Critical Systems Design” module, the following course will expand on some aspects of this chapter:

- MA 1, INFO-F-412, Formal verification of computer systems, MASSART, RASKIN.

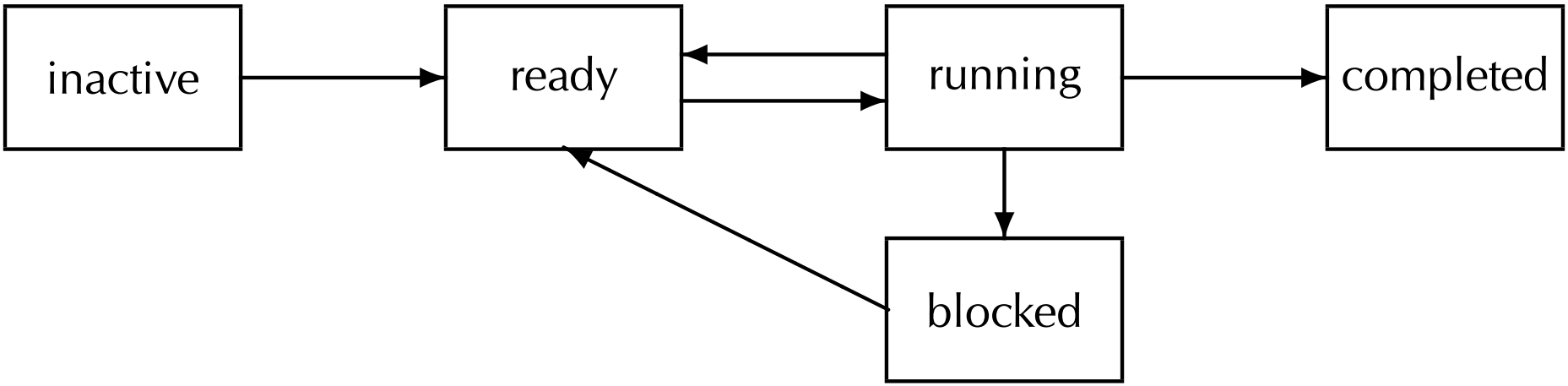
D) Semaphores

Introduction

- The solutions presented in the previous chapter for the critical section problem are dedicated to programs written in **machine language** and used low-level constructs.
- In this chapter, we will study **semaphores**, which are higher-level constructs used for concurrent program design.
- We will start by introducing the concept of process state.

Process state

- We distinguish 5 possible states for a given process:
  - Inactive: initial state;
  - Ready: when the process gets activated;
  - Running: when its computations are progressing;
  - Blocked: a blocked process is not eligible to run;
  - Completed: when the last instruction has been executed.



The semaphore data type I

- A semaphore S is made of two components:
  - an unsigned integer S.V;
  - a set of processes S.L.
- A semaphore must be initialized with a value $k \geqslant 0$ for S.V and with the empty set for S.L:
  semaphore S $\leftarrow (k, \emptyset)$
- We have two **atomic** operations on a semaphore S: wait(S) and signal(S). In the following, p denotes the process which executes the operation.
- The original names for the operations, as given by Dijkstra, were P(S) and V(S) respectively, from the Dutch *proberen* (to try) and *vrijgeven* (to free).

The semaphore data type II

- **wait(S)**
  if S.V $>$ 0
    S.V $\leftarrow$ S.V - 1
  else
    S.L $\leftarrow$ S.L $\bigcup$ p
    p.state $\leftarrow$ blocked
- **signal(S)**
  if S.L $= \emptyset$
    S.V $\leftarrow$ S.V $+$ 1
  else
    let q be an arbitrary element of S.L
    S.L $\leftarrow$ S.L $\setminus \{q\}$
    q.state $\leftarrow$ ready

- A semaphore whose integer component can take all possible unsigned values is called a **general semaphore**.

The semaphore data type III

- A semaphore whose integer component can only take values 0 and 1 is called a **binary semaphore**. In this particular case, the signal(S) instruction can be adapted as such:

```
if S.V = 1
    // undefined
else if S.L = ∅
    S.V ← 1
else // (as above)
    let q be an arbitrary element of S.L
    S.L ← S.L \ {q}
    q.state ← ready
```

Critical section with two processes I

- We shall use a binary semaphore to solve the critical section problem;
- The preprotocol then maps to wait(S);
- The postprotocol maps to signal(S):

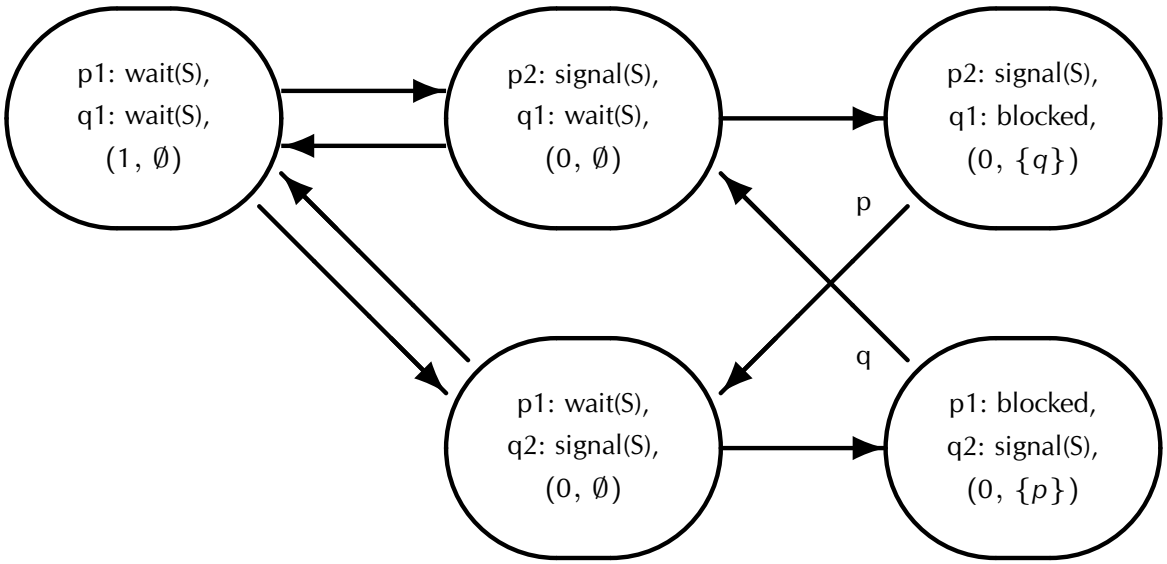
Critical section with semaphores (two processes)	
binary semaphore S $\leftarrow (1, \emptyset)$	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wait(S)	q2: wait(S)
p3: critical section	q3: critical section
p4: signal(S)	q4: signal(S)

- To convince ourselves of the validity of the protocol, we can reason upon the following abridged version:

Critical section with two processes II

Critical section with semaphores (two proc., abbrev.)	
binary semaphore S $\leftarrow (1, \emptyset)$	
p	q
loop forever	loop forever
p1: wait(S)	q1: wait(S)
p2: signal(S)	q2: signal(S)

- The corresponding state diagram is the following:



Critical section with N processes I

- Using a binary semaphore, the same protocol can be extended to be used with an arbitrary number of processes:

Critical section with semaphores (N proc.)	
binary semaphore S $\leftarrow (1, \emptyset)$	
	loop forever
p1:	non-critical section
p2:	wait(S)
p3:	critical section
p4:	signal(S)

- A state diagram can be used to show that mutual exclusion is guaranteed and deadlocks are avoided.
- However, due to our definition of signal(S), processes may starve because we **arbitrarily** pick which process to wake up.
- Starvation can be avoided by changing the definition of signal(S) so that it uses a queue (FIFO order) instead of an unordered set.

Synchronization I

- The critical section problem is an abstraction of the **synchronization problem** that arises when several processes compete for a resource.
- Synchronization allows coordination of the execution order of the processes' diverse operations.
- For example, a merge sort can be broken up into 3 concurrent processes:
  - sort the first half of the array;
  - sort the other half of the array;
  - merge both halves of the array.
- the two sorting procedures are independent and do not require synchronization. However, the merging procedure must wait for the completion of the sorting procedures.
- The following is a solution using two binary semaphores:

Synchronization II

Algorithm Mergesort					
integer array A binary semaphore S1 $\leftarrow (0, \emptyset)$ binary semaphore S2 $\leftarrow (0, \emptyset)$					
sort1		**sort2**		**merge**	
p1:	sort 1st half of A	q1:	sort 2nd half of A	r1:	wait(S1)
p2:	signal(S1)	q2:	signal(S2)	r2:	wait(S2)
p3:		q3:		r3:	merge halves of A

Producer-consumer problem I

- We now consider two kinds of processes:
  - **Producers** that generate data and transmit them to consumers;
  - **Consumers** that make use of the data transmitted by producers.
- The producer-consumer problem arises frequently is computer systems:

Producer	Consumer
Communications line	Web browser
Web browser	Communications line
Keyboard	Operating system
Word processor	Printer

- In general, communications between producers and consumers are **asynchronous** and the communication channel has limited capacity. Usually, buffers are used in this context.
- Let's first consider the case of an infinite buffer (i.e. with unlimited memory). In this case, we must only ensure that a consumer never manages to read an empty buffer.

Producer-consumer problem II

Producer-consumer (infinite buffer)	
infinite queue of dataType buffer $\leftarrow$ empty queue	
semaphore notEmpty $\leftarrow (0, \emptyset)$	
producer	consumer
dataType d	dataType d
loop forever	loop forever
p1: d $\leftarrow$ produce	q1: wait(notEmpty)
p2: append(d, buffer)	q2: d $\leftarrow$ take(buffer)
p3: signal(notEmpty)	q3: consume(d)

Producer-consumer problem with finite buffer I

- In the case where buffers are finite, we must also ensure that a producer never writes into a full buffer.
- We will use another semaphore notFull whose integer component is initialized with buffer size N:

Producer-consumer (finite buffer, semaphores)	
finite queue of dataType buffer $\leftarrow$ empty queue	
semaphore notEmpty $\leftarrow (0, \emptyset)$ semaphore notFull $\leftarrow (N, \emptyset)$	
producer	consumer
dataType d	dataType d
loop forever	loop forever
p1: d $\leftarrow$ produce	q1: wait(notEmpty)
p2: wait(notFull)	q2: d $\leftarrow$ take(buffer)
p3: append(d, buffer)	q3: signal(notFull)
p4: signal(notEmpty)	q4: consume(d)

E) Distributed algorithms

Introduction I

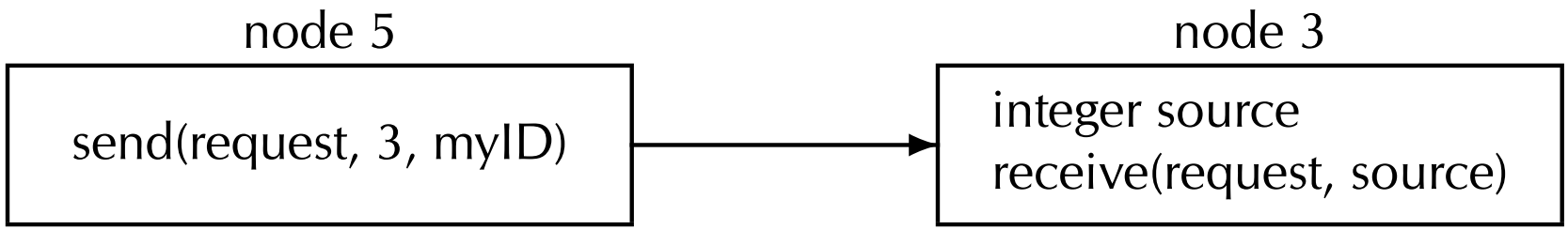
- In this chapter, we consider loosely coupled and distributed systems. Processes communicate by sending and receiving messages through a communications network (see MPI paradigm).
- We may consider various distributed architectures such as clusters, i.e. computers with "average" power connected through a high-speed network (LAN).
- Another example of architecture is a computing grid, made by connecting several clusters through a wide-area network (WAN). This allows sharing of computational resources between laboratories, universities and institutions throughout the world.
- We will use the notions of **nodes** and **processes** as abstractions in this chapter.
  - A **node** models a physical object such as a computer, whether it has one or several processors. We assume synchronization between processes within a node is ensured by way of shared memory. Synchronization between processes belonging to different nodes is done by sending and receiving messages.

Introduction II

- Every node is connected via a bidirectional channel to all other nodes. Channels are assumed to be perfect (no data loss) and transmission delays are assumed to be bounded.
- The **send(MessageType, Destination[, Parameters])** primitive models a message send from one node to another with possible optional parameters.
- The **receive(MessageType, [, Parameters])** primitive models a message receive with possible optional parameters.

Introduction III

- Example:



Distributed critical section problem I

- In the context of distributed systems, the critical section problem relates to a set of nodes. Updating a database could is an example of an instance of the problem.
- Solutions to this problem are inspired by the bakery algorithm we studied previously (see slide 31 and further). Nodes must pick ticket numbers and the smallest number gains entrance to its critical section.
- In a distributed system, ticket numbers **cannot** be compared directly due to the absence of a shared memory. Messages must be exchanged.
- We will present the solution in several steps. We'll start with an outline of the algorithm made of two parts:
  - the process body
  - the routine that handles the reception of a message

Distributed critical section problem II

Ricart-Agrawala algorithm (outline)	
integer myNum ← 0	
set of node IDs deferred ← empty set	
	main
p1:	non-critical section
p2:	myNum ← chooseNumber
p3:	for all *other* nodes N
p4:	send(request, N, myID, myNum)
p5:	await reply's from all *other* nodes
p6:	critical section
p7:	for all nodes N in deferred
p8:	remove N from deferred
p9:	send(reply, N, myID)
	receive
	integer source, reqNum
p10:	receive(request, source, reqNum)
p11:	if reqNum < myNum
p12:	send(reply,source,myID)
p13:	else add source to deferred

Distributed critical section problem III

- We will now look at how a process should pick its ticket number. Making an arbitrary choice can easily break the mutual exclusion property (see exercise sessions).
- Indeed, ticket numbers must be **monotonic**. Each node must keep track of the greatest number used so far in the system (through a variable called **highestNum**).
- The algorithm assumes that nodes always reply to an incoming message by way of an outgoing message. The "Main" process may die at some point, but only outside of its critical section.
- We can now present the complete algorithm:

Distributed critical section problem IV

Ricart-Agrawala algorithm	
integer myNum ← 0	
set of node IDs deferred ← empty set	
integer highestNum ← 0	
boolean requestCS ← false	
Main	
	loop forever
p1:	non-critical section
p2:	requestCS ← true
p3:	myNum ← highestNum + 1
p4:	for all *other* nodes N
p5:	send(request, N, myID, myNum)
p6:	await reply's from all *other* nodes
p7:	critical section
p8:	requestCS ← false
p9:	for all nodes N in deferred
p10:	remove N from deferred
p11:	send(reply, N, myID)

Distributed critical section problem V

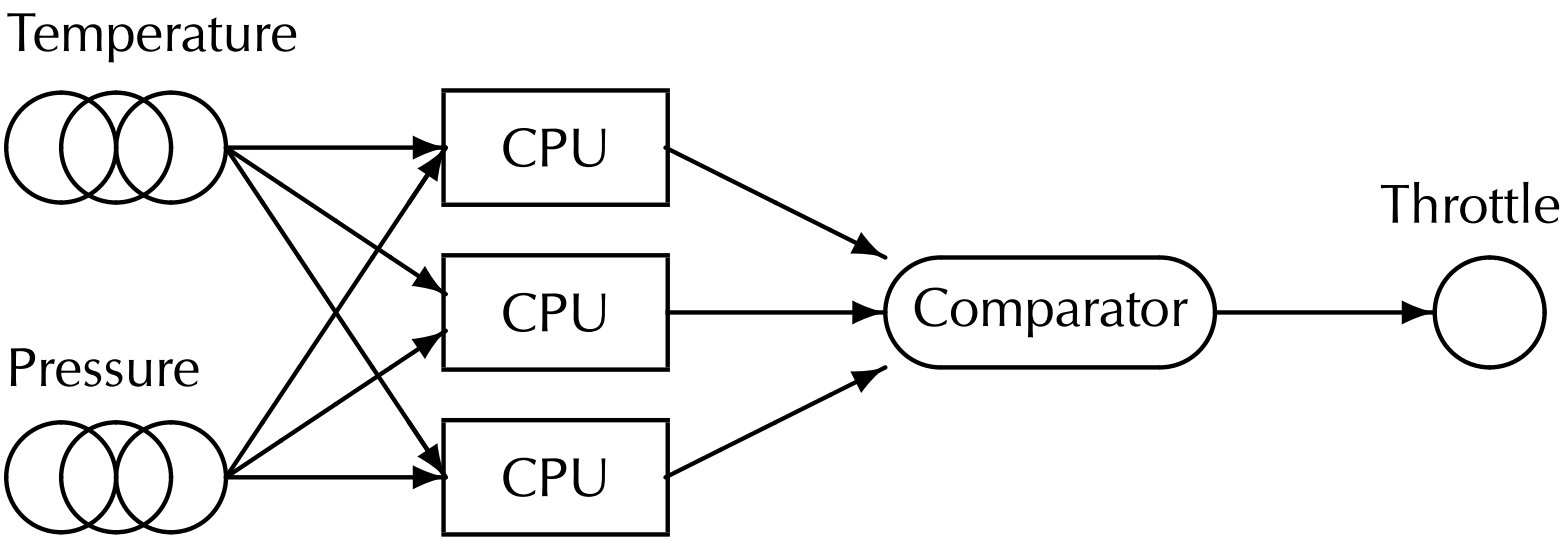
Ricart-Agrawala algorithm (continued)	
	Receive
	integer source, requestedNum
	loop forever
p1:	receive(request, source, requestedNum)
p2:	highestNum $\leftarrow$ max(highestNum, requestedNum)
p3:	if not requestCS or requestedNum $\ll$ myNum
p4:	send(reply, source, myID)
p5:	else add source to deferred

Consensus I

- A recurring motivation behind the use of distributed systems is to improve reliability by duplicating computations on several independent processors.
- In this context, there are two desirable properties:
  - fail-safety: one or more failures do not cause damage to the system or its users.
  - fault-tolerance: the system continues to fulfill its requirements even if there are one or more failures.

Consensus II

- This is a characteristic diagram of a reliable system:



- Sensors and CPUs are duplicated. Results are filtered through a comparator (which could be, for example, a majority vote algorithm) and the final result is returned.
- We will study a specific problem called the consensus problem in distributed systems.

Consensus III

- Each node picks an initial value. It is then expected of all nodes to agree on (find a consensus on) one of those values.
- If no failures arise, we can devise a simple solution to the problem. Each node sends its choice to all other nodes. Then, a deterministic vote algorithm makes a choice based on all votes (e.g. through a majority vote). All nodes use the same algorithm and the same source data, thus making the same final choice.
- We will consider two types of failures that could arise in practice:
  - **crash failure**: a node ceases to send messages.
  - **byzantine failure**: a node sends erroneous or arbitrary messages.

Byzantine Generals problem I

- Let's imagine several divisions of the Byzantine army besieging an enemy city. Each division is led by its own general. We assume generals can communicate through the use of messengers. After observing enemy activity, all generals must agree on a common plan of action: attack ("A") or retreat ("R"). However, some generals may be traitors and will attempt to cause loyal generals not to be able to find a consensus.
- Thus, the generals must have a decision algorithm that guarantees that:
  1. **All loyal generals agree on a plan of action.**

     Loyal generals will all do what the algorithm tells them to, but traitors will do whatever they want. The algorithm must thus guarantee condition 1, regardless of the course of action chosen by the traitors.
  2. **A "small" number of traitors cannot cause the loyal generals to pick the wrong course of action.**

Byzantine Generals problem II

- In terms of distributed systems, generals model nodes and messengers model communication channels.
- We shall consider two kinds of failures:
  - **crash failure**: a traitor ceases to send messages.
  - **byzantine failure**: a traitor sends arbitrary messages, not only those required by the algorithm.
- A probable crash can be detected through the use of timeouts.
- Byzantine failures force us to consider malicious messages.

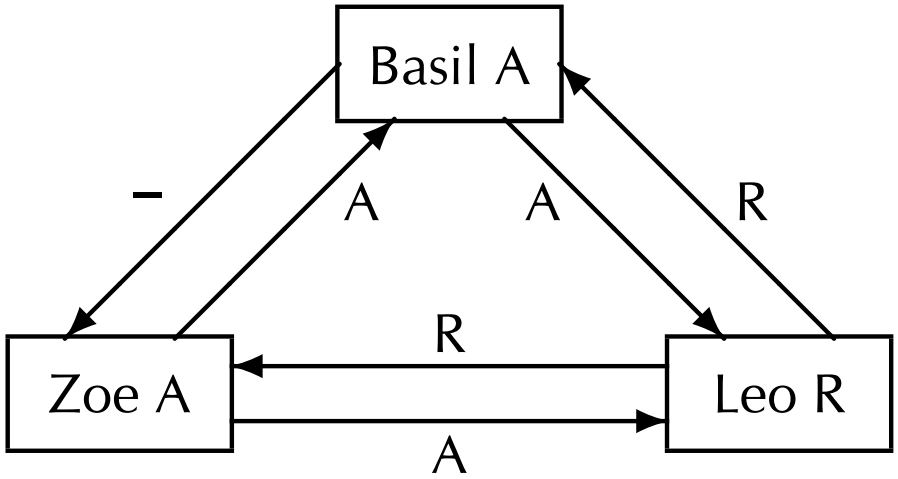
Consensus — one-round algorithm I

- Let's begin by looking at the aforementioned simple solution:

Consensus - one-round algorithm
planType finalPlan
planType array[generals] plan
p1: plan[myID] ← chooseAttackOrRetreat p2: for all *other* generals G p3: send(G, myID, plan[myID]) p4: for all *other* generals G p5: receive(G, plan[G]) p6: finalPlan ← majority(plan)

- In case of a tie, the *majority* function returns "R" (retreat).
- Assume we have 3 generals. Zoe and Leo are loyal, whereas Basil is a traitor. The following diagram illustrates what would happen if Basil crashes after sending a message to Leo:

Consensus — one-round algorithm II



- The following tables show the data collected by both loyal generals. They end up taking a different course of action!

Leo	
general	plan
Basil	A
Leo	R
Zoe	A
majority	A

Zoe	
general	plan
Basil	–
Leo	R
Zoe	A
majority	R

Consensus — Byzantine Generals algorithm I

- The general solution to the Byzantine Generals problem is to use a two-round algorithm. During the first round, every general broadcasts his own plan. During the second round, every general broadcasts the data received from all other generals. By definition, loyal generals will keep all information intact. Thus, if they are numerous enough, they will be able to make a common decision.
- The two-round solution follows:

Consensus — Byzantine Generals algorithm II

Consensus - Byzantine Generals algorithm		
planType finalPlan		
planType array[generals] plan, majorityPlan planType array[generals, generals] reportedPlan		
p1:	plan[myID] ← chooseAttackOrRetreat	
p2:	for all *other* generals G	// First round
p3:	send(G, myID, plan[myID])	
p4:	for all *other* generals G	
p5:	receive(G, plan[G])	
p6:	for all *other* generals G	// Second round
p7:	for all *other* generals G′ except G	
p8:	send(G′, myID, G, plan[G])	
p9:	for all *other* generals G	
p10:	for all *other* generals G′ except G	
p11:	receive(G, G′, reportedPlan[G, G′])	
p12:	for all *other* generals G	// First vote
p13:	majorityPlan[G] ← majority(plan[G] ∪ reportedPlan[*, G])	
p14:	majorityPlan[myID] ← plan[myID]	// Second vote
p15:	finalPlan ← majority(majorityPlan)	

Consensus — Byzantine Generals algorithm III

Leo				
general	plans	reported by		majority
		Basil	Zoe	
Basil	A		A	A
Leo	R			R
Zoe	A	**R**		R
majority				R

[1] Lamport, L.
A new solution of Dijkstra's concurrent programming problem.
Communications of the ACM 17, 8 (1974), 453–455.