



departamento de informática  
FACULDADE DE CIÊNCIAS E TECNOLOGIA  
UNIVERSIDADE NOVA DE LISBOA

# Synchronization

Concurrency and Parallelism — 2017-18  
Master in Computer Science  
(Mestrado Integrado em Eng. Informática)

Joao Lourenço <[joao.lourenco@fct.unl.pt](mailto:joao.lourenco@fct.unl.pt)>

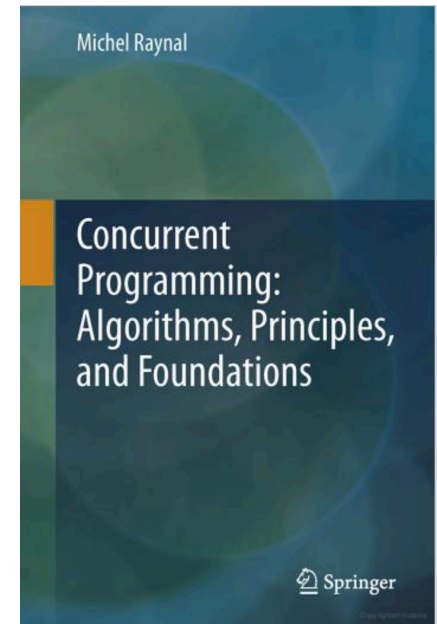
# Summary

- **Synchronization**

- Competition and Cooperation
- Properties
- Invariants

- **Reading list:**

- **Chapter 1** of the book  
Raynal M.;  
**Concurrent Programming: Algorithms,  
Principles, and Foundations;**  
Springer-Verlag Berlin Heidelberg (2013);  
ISBN: 978-3-642-32026-2



# Algorithms, Programs and Processes

---

- A **sequential algorithm** is a formal description of the behavior of a sequential state machine
  - The text of the algorithm states the transitions that have to be sequentially executed
- When written in a specific programming language, an algorithm is called a **program**
- A **process** is an instance of an algorithm

# Multiprocess Programs

---

- A **concurrent algorithm** (or **concurrent program**) is the description of a set of sequential state machines that cooperate through a communication medium, e.g., a shared memory
- Concurrent algorithm is a **multiprocess program**
  - Each process corresponding to the sequential execution of a given state machine

# Process Synchronization

---

- **Process synchronization** occurs when the progress of one or several processes depends on the behavior of other processes
- Two types of process interaction require synchronization: **competition** and **cooperation**

# Synchronization: Competition

---

- This kind of interaction occurs when processes compete to execute some statements and only one process at a time (or a bounded number of them) is allowed to execute them
  - Example: when processes compete for a shared resource

# Competition: Example

- Consider a disk with random (atomic) access I/O operations
  - seek(p) moves the disk head to location 'p'
  - read() reads the contents of location 'p'
  - write(v) writes 'v' in the location 'p' (overwriting previous contents)
- Process operations to be implemented/used:
  - disk\_read(x) returns the contents of disk location 'x'
  - disk\_write(x,v) writes 'v' into disk location 'x'

# Operations on disk 'D'

**operation** `disk_read( $x$ )` **is**

*%  $r$  is a local variable of the invoking process %*

*$D.\text{seek}(x); r \leftarrow D.\text{read}(); \text{return}(r)$*

**end operation.**

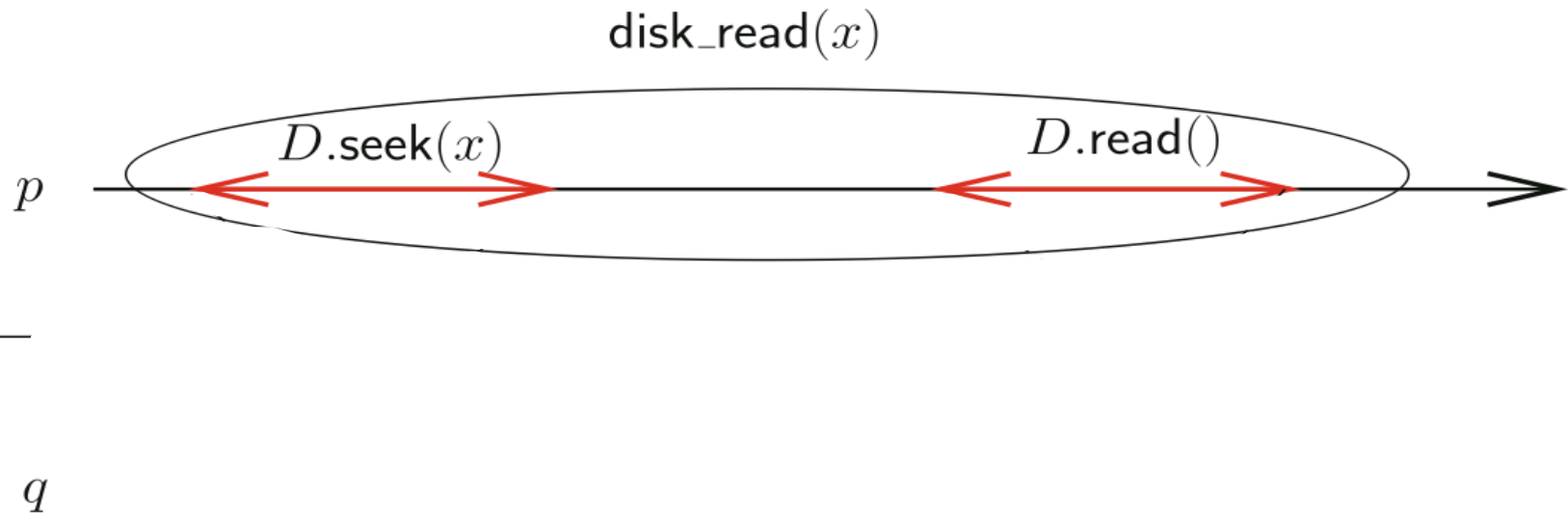
**operation** `disk_write( $x, v$ )` **is**

*$D.\text{seek}(x); D.\text{write}(v); \text{return}()$*

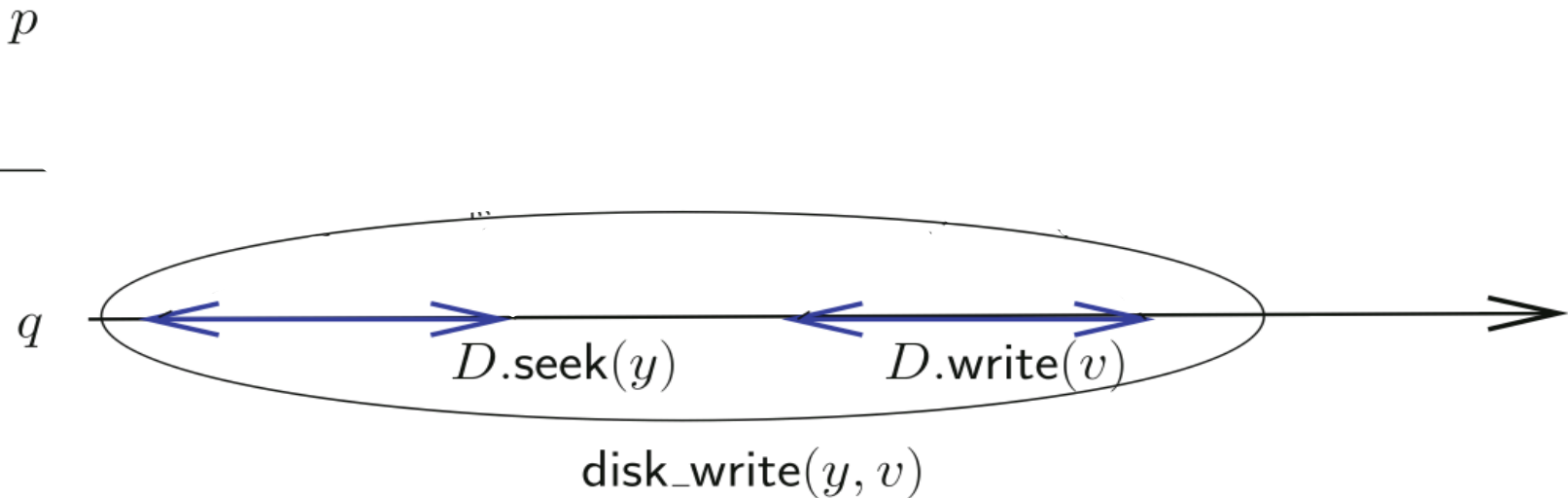
**end operation.**



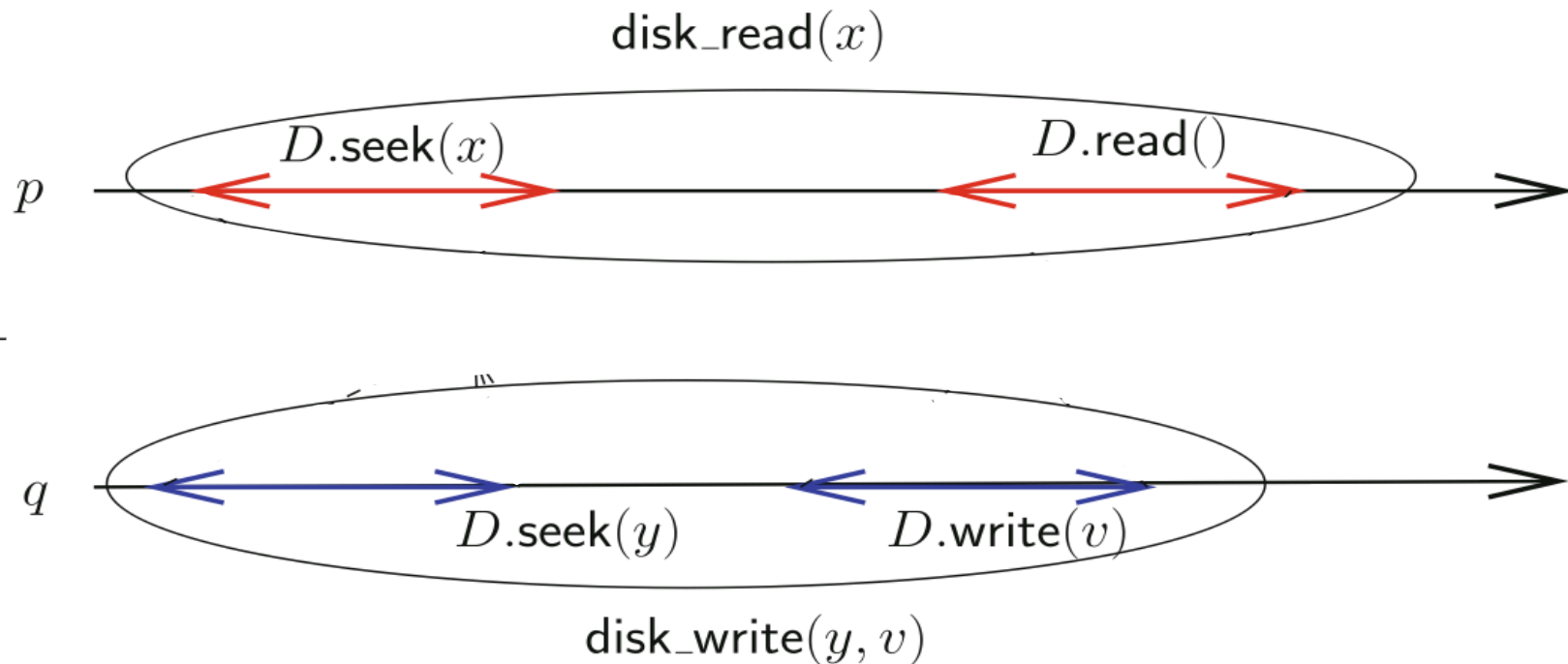
# An interleaving of disk operations



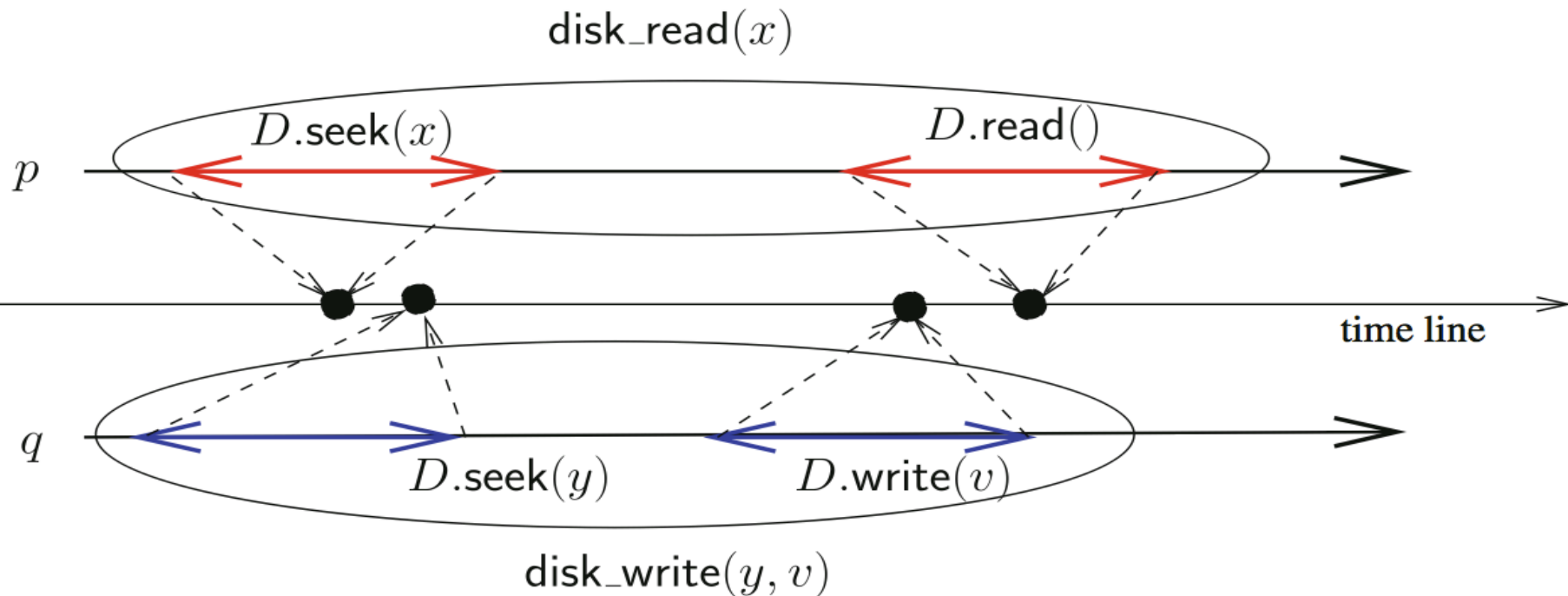
# An interleaving of disk operations



# An interleaving of disk operations



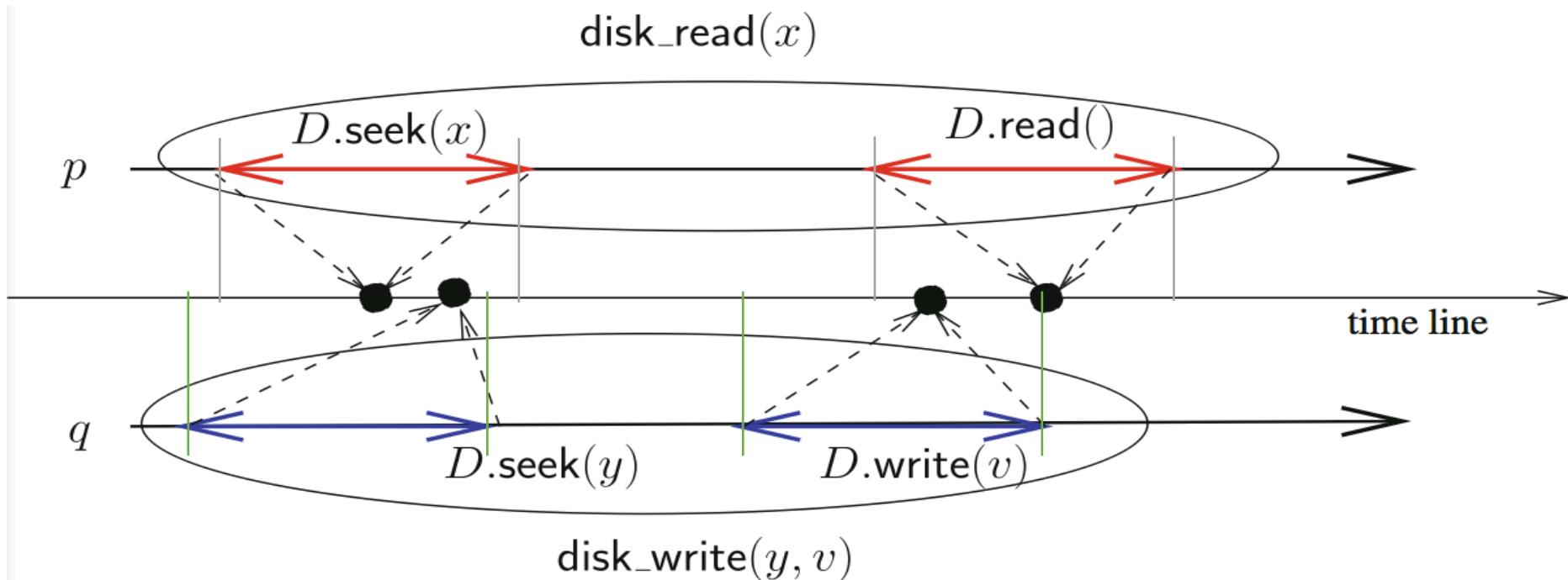
# An interleaving of disk operations



# An interleaving of disk operations

Linearization

Non-determinism



# Synchronization: Cooperation

---

- Barrier (or rendezvous)
  - A set of control points, one per process involved in the barrier, such that each process is allowed to pass its control point only when all other processes have attained their own control points
  - From an operational point of view, each process has to stop until all other processes have arrived at their control point

# Synchronization: Cooperation

---

- A producer–consumer problem
  - The producer loops forever on producing data items
  - The consumer loops forever on consuming data items
- The problem consists in ensuring that
  - a) only data items that were produced are consumed, and
  - b) each data item that was produced is consumed exactly once
- How to solve this problem?

# Solutions to the prod-cons problem

---

- Use a synchronization barrier
  - Both the producer (when it has produced a new data item) and the consumer (when it wants to consume a new data item) invoke the barrier operation
  - When, they have both attained their control point, the producer gives the data item it has just produced to the consumer
- This coordination pattern works but...
  - ...it is not very efficient (overly synchronized): for each data item, the first process that arrives at its control point has to wait for the other process



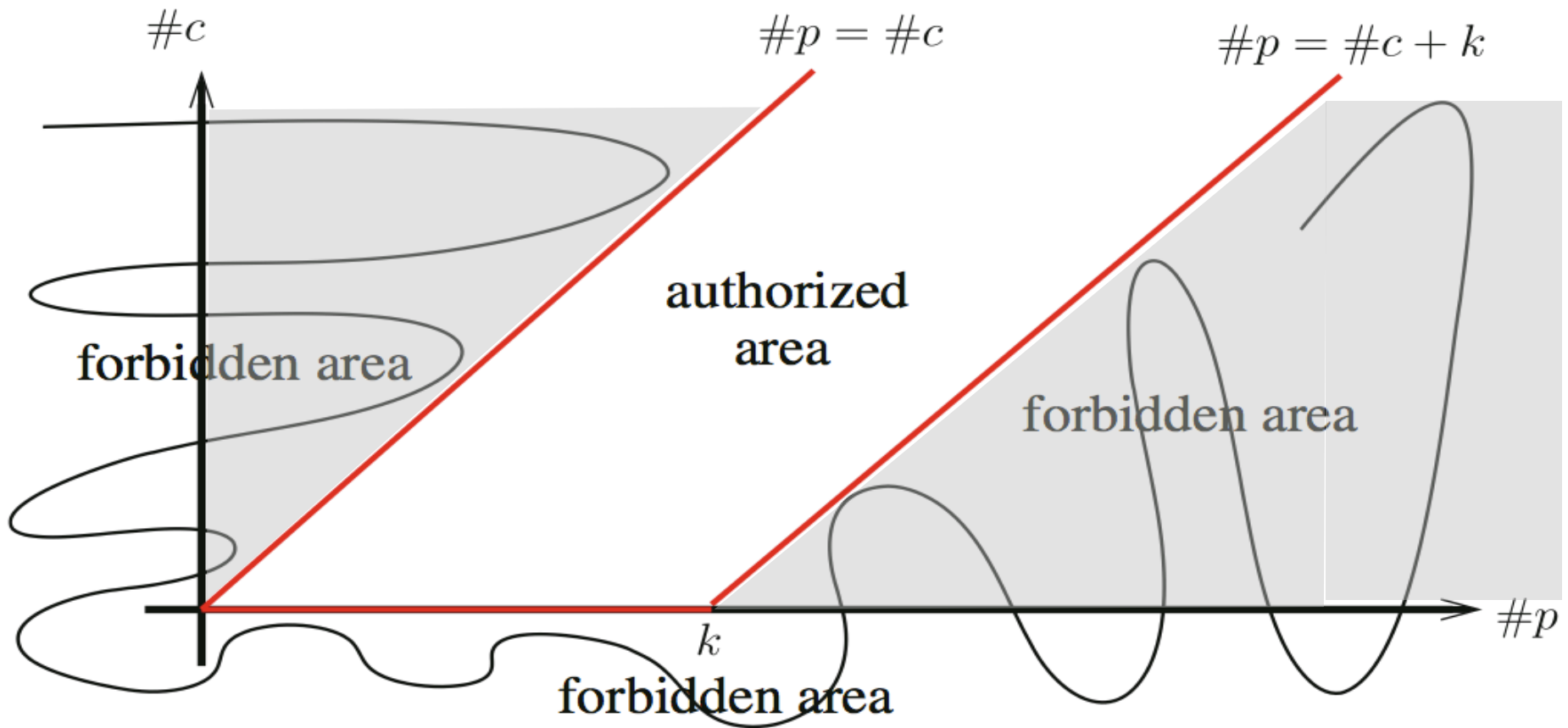
# Solutions to the prod-cons problem

- Use a shared buffer of size  $k \geq 1$ 
  - I.e., a queue or a circular array
  - The producer adds new data items to the end of the queue
  - The consumer withdraws the data item at the head of the queue
- Properties of a p-c with a buffer of size  $k$ :
  - A producer has to wait only when the buffer is full
    - It contains  $k$  data items produced and not yet consumed
  - A consumer has to wait only when the buffer is empty
    - Occurs each time all data items that have been produced have been consumed

# Synchronization: Invariants

- The **aim** of **synchronization** is to **preserve invariants**
- $\#p$  = number of data items produced so far
- $\#c$  = number of data items consumed so far
- Invariant for a buffer of size  $k$  is
$$(\#c \geq 0) \wedge (\#p \geq \#c) \wedge (\#p \leq \#c + k)$$

# Synchronization: Invariants



# The Mutual Exclusion Problem

- Critical section
  - Part of code *A* (i.e., an algorithm) or several parts of code *A*, *B*, *C*, ... (i.e., different algorithms) that, for some consistency reasons, must be executed by a single process at a time
    - E.g., if a process is executing code *B*, no other process can be simultaneously executing the codes *A* or *B* or *C* or etc.
- The operations `disk_read()` and `disk_write()` from before were critical sections

# Mutual exclusion

- How to provide the application processes with an appropriate abstraction level?
- Designing
  - an **entry algorithm** (also called **entry protocol**); and
  - an **exit algorithm** (also called **exit protocol**)
  - that, when used to **delimit a critical section** `cs_code(in)`, **ensure** that the **critical section code is executed** by at most **one process at a time**
- Operations:
  - **acquire\_mutex()** and **release\_mutex()**

# Mutual exclusion: acquire\_mutex

- One of the invocations terminates
  - The corresponding process  $p$  is called the winner
- The other invocations stay in hold
  - The competing processes  $q_i$  are the losers
    - Their invocations remain pending
- A well formed process executes the entry and exit protocols appropriately

**procedure** protected\_code( $in$ ) **is**  
    acquire\_mutex();  $r \leftarrow$  cs\_code( $in$ ); release\_mutex(); return( $r$ )  
**end procedure.**

# Mutual exclusion: definition

---

- The mutual exclusion problem consists in implementing the operations *acquire\_mutex()* and *release\_mutex()* in such a way that the following properties are always satisfied:
- **Mutual exclusion**, i.e., at most one process at a time executes the critical section code
- **Starvation-freedom**, i.e., for any process 'p', each invocation of *acquire\_mutex()* by 'p' eventually terminates

# Mutual exclusion: properties

- Safety
  - Safety properties state that nothing bad happens
  - They can usually be expressed as invariants
    - The invariant here is the mutual exclusion property, which states that **at most one process at a time** can execute the critical section code
  - *Note that a solution in which no process is ever allowed to execute the critical section code would trivially satisfy the safety property*



# Mutual exclusion: properties

- Liveness
  - Liveness properties state that something good eventually happens
- Examples of liveness properties:
  - Starvation freedom
    - Means that a process that wants to enter the critical section can be bypassed an arbitrary but finite number of times by each other process
  - Deadlock-freedom
    - Whatever the time  $\tau$ , if before  $\tau$  one or several processes have invoked the operation `acquire_mutex()` and none of them has terminated its invocation at time  $\tau$ , then there is a time  $\tau' > \tau$  at which a process that has invoked `acquire_mutex()` terminates its invocation

# Mutual exclusion: properties

---

- Starvation-freedom implies deadlock-freedom
  - If a process requests access to the critical section it will eventually get permission
  - To get permission the system cannot be deadlocked
- Deadlock-freedom does not imply starvation-freedom
  - The system is operating
  - There is a process willing to get access to the critical section that is always overcome by another later process

# The Lock Object

- A lock is a shared object with two methods:
  - **LOCK.acquire\_lock()** and **LOCK.release\_lock()**
- A lock can be in one of two states:
  - **free** or **locked**
- And is initialized
  - to the value **free**
- Its behavior is defined by a sequential specification
  - from an external observer point of view, all the **acquire\_lock()** and **release\_lock()** invocations appear as if they have been invoked one after the other
  - Sequence: ( **LOCK.acquire\_lock(); LOCK.release\_lock()** )\*

# The END

---