Distributed algorithms in the shared memory model

.

Emmanuelle Anceaume

emmanuelle.anceaume@irisa.fr

Shared memory

- You have already seen different distributed algorithms in the message passing paradigm
- We now turn our attention to the other major communication paradigm for distributed systems: the shared memory paradigm
- In a shared memory system, processors communicate asynchronously via a common memory area
- This memory area contains a set of shared variables
- Several types of variables can be employed
  - Type specifies the operations that can be performed on it and the values that can be returned by the operations

Shared memory (cont’d)

- The most common type is a read/write variable
- 2 Operations: read() and write()
  - write(X, v): variable x is assigned value v
  - read(X) returns the value of X (i.e. the value written by the last preceding write)

Shared memory (cont'd)

- Other types of shared variables exist, which are more powerful in the operations they perform

  - Test&Set(): allows to atomically read and update the shared variable

  - Read-Modify-Write(): allows to atomically read the current value of the shared variable, computes a new value as a function of the current value, write the new value to the variable, and returns the previous value of the variable.

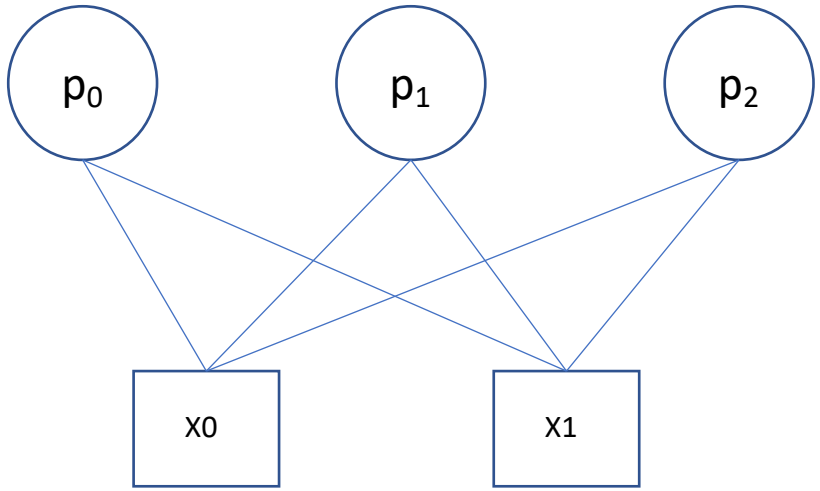
  - Compare&Swap(): allows to atomically performs a conditional write. Takes two arguments: an expected value and an update value. If the current value of the variable is equal to the expected one, then it is replaced by the update value. Returns true if a change occurred.

Shared memory(cont'd)

- Shared variables (also called registers) can be further characterized according to their access patterns, i.e., how many processors can access the variable
- The type of shared variables determines the possibility of solving a given problem
- We now formalize the notion processes, shared objects, history (execution)

Shared Memory System

- There are n processors/cores/processes p_1,..,p_n running concurrently in the system
- Besides accessing local variables, processes may execute operations on shared objects
- These objects X1, X2, … allow processes to synchronize their computations
- Each process is modelled as a state machine



Shared memory system: configuration

- As in the case of message-passing systems,
  - we model processors as state machines and
  - We model executions as alternating sequences of configurations and events

- The difference is the nature of configurations and events

Shared memory system: configuration

- **A configuration describes the distributed system at some point in time**

- A configuration is a vector

$$C = (q_1, q_2, \dots, qn, r_1, r_2, \dots rm),$$

  where
  - q_i is the state of processor p_i , $1 \leq i \leq n$, and
  - r_j is the value of register R_j, $1 \leq j \leq m$

- In the initial configuration C_0, all the processors are in their initial state and the registers are initialised with their initial values

Shared memory model: event

- **An event is a computational step by any of the processors**

- At each computational step by some processor p_i the following happens atomically:

    1. p_i chooses the shared variable (register) to access with a specific operation based on p_i's current state

    2. The specified operation is performed on the shared variable

    3. p_i's state changes according to p_i's current state and the value returned by the shared variable operation

Shared memory model: Execution

- **An execution** of an algorithm is a finite or infinite alternating sequence of the form

$$C_0 \dots, C_{k-1}, \Phi_k, C_k, \Phi_{k+1}, \dots$$

where each C_k is a configuration and each Φ_k is an event

- The application of Φ_k to C_{k-1} results in C_k as:
  - Suppose that $\Phi_k = i$ (computational step made by p_i) and p_i's state in C_{k-1} indicates that R_j is the shared register to be accessed by p_i
  - Then C_k is the result of changing C_{k-1} in accordance with p_i's state in C_{k-1} and the value R_j in C_{k-1}
  - ➢ Thus the only change are to p_i's state and the value of register R_j

$$C_{k\_1} = (q_1, \dots, q_i, \dots, qn, R_1, \dots, R_j, \dots R_m) \quad \xrightarrow{\Phi_k = i} \quad C_k = (q_1, \dots, q_i, \dots, qn, R_1, \dots, R_j, \dots R_m)$$

Shared object

- Each object has a type, which specifies
  1. The values that can be taken on by the object
  2. The operations that can be performed on the object
  3. The values that can be returned by each operation (if any)
  4. The new value of the variable resulting from each operation (if any)

- An initial value can be specified for each object

- e.g. an integer-valued read/write object X can be accessed by two operations
  1. **X.read()** which returns the value in X, without modifying X's content
  2. **X.write(v)** changes X's value to v, does not return any value

Shared object

- An object is defined by a sequential specification describing for each operation its effect when executed (alone) on the object
  - By effect we mean the response that the object returns and the new state of the object after the operation executed

- Note that we assume that each operation can be applied on each state of the object (we say that the object is complete). This may require some care: for instance if a dequeue() operation is invoked on an empty queue then a specific response *nil* must be returned

Shared object

- The read/write object models a shared memory word, a shared file, a shared disk…
  - read(): no input parameter. Returns the value stored in the object
  - write(*v*): has an input parameter *v*, representing the new value of the object. The operation returns ok indicating to the invoking process that the operation has terminated

Shared memory model: Histories

- The interaction between processes and objects is modeled as a sequence of events H
- H is called a history or a trace of the system execution
- A history models executions of processes on shared objects
  - say differently, a history represents the sequence of events of the execution of a concurrent system

Shared memory model: Histories

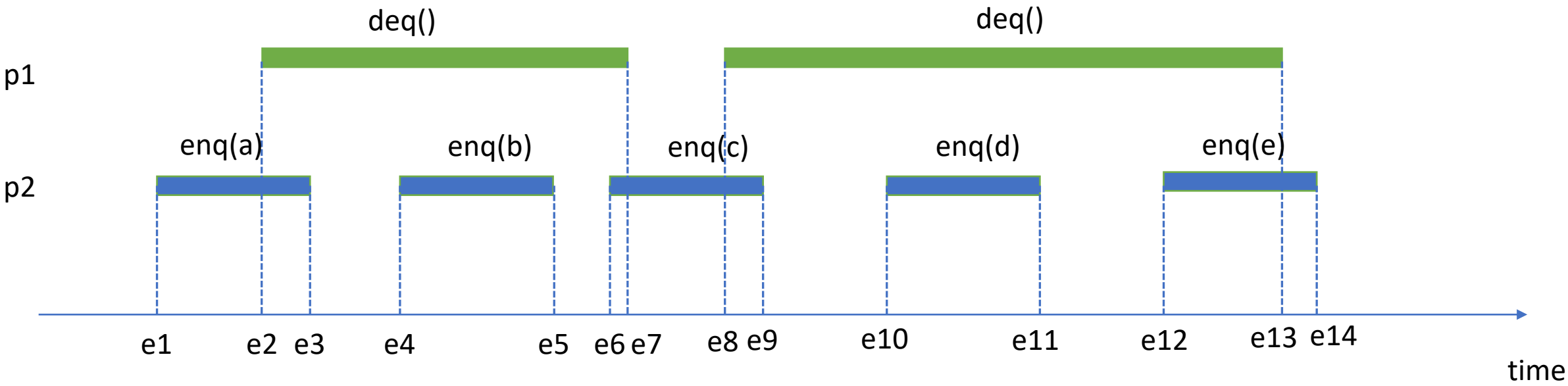
- While the system of processes is concurrent, each process is individually sequential
  - A process executes at most one operation on an object at a time
  - inv(X.write(4)) res(X.write(4)) inv(Y.read()) res(Y.read()->4)

- A local history of p, denoted by H|p is a projection of H on p, i.e., the subsequence H consisting of the events generated by p
- Two histories H and H' are equivalent if they have the same local histories, i.e., for each p, H|p = H'|p

Shared memory model: Histories

- As we are considering only sequential processes, we focus on histories *H* s.t. for each process *p*, $H|p$ is sequential
  - The history starts with an invocation followed by the matching response, followed by an invocation ,….
  - The history is call well-formed
- An history is called complete if both the invocation and the response belong to the history. Uncomplete histories model executions in which some processes may crash

H: history made of the operations executed (events) on a FIFO queue



$H|p_1 = \{e2,e7,e8,e13\}$
$H|p_2 = \{e1,e3,e4,e5,e6,e9,e10,e11,e12,e14\}$

H is a complete history

$H'|p_1 = \{e2,e7,e8\}$
$H'|p_2 = \{e1,e3,e4,e5,e6,e9,e10,e11,e12\}$

H' is an uncomplete history

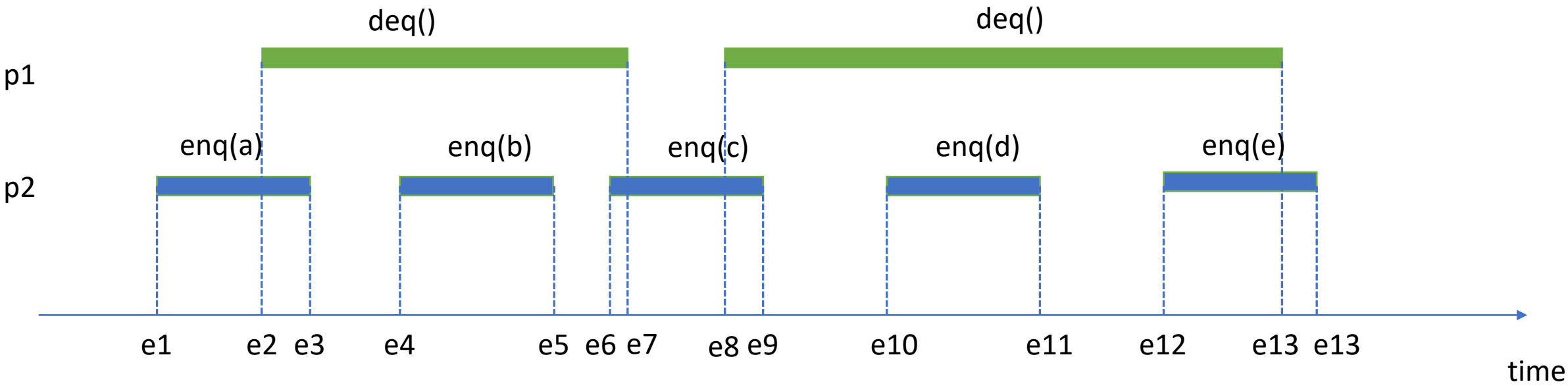
Shared memory model: Histories

- A history induces a partial order on its operations.
- Operation op precedes operation op' (denoted by op $->_H$ op') if op terminates before op' starts, where terminate and start refer to real-time order
- The real-time order in which events actually occur in H is denoted by $<_H$

$$(op ->_H op') = (resp(op) <_H inv(op'))$$

- Two operations are concurrent if neither precedes the other one

H: history made of the operations executed (events) on a FIFO queue



Operations enq(a), enq(b), enq(c), enq(d) and enq(e) are sequential

Operations deq(), enq(a), enq(b) and enq(c) are concurrent

Operations deq(), enq(c), enq(d) and enq(e) are concurrent

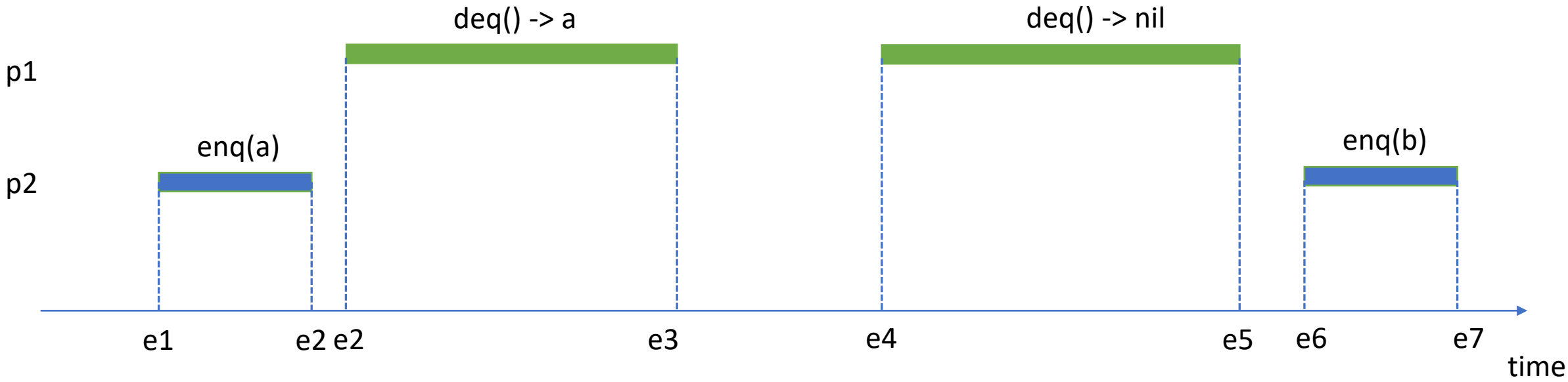
=> H is a concurrent history

Shared memory model: Histories

- A history S is sequential is the first event is an invocation followed by the matching response, followed by an invocation followed by the matching response and so on
- A history S that is not sequential is concurrent
- The associated partial order $->_S$ defined on its operations is thus a total order

- Hence, the sequential specification of an object is the set of all possible sequential histories involving only this object

S: history made of the operations executed (events) on a FIFO queue

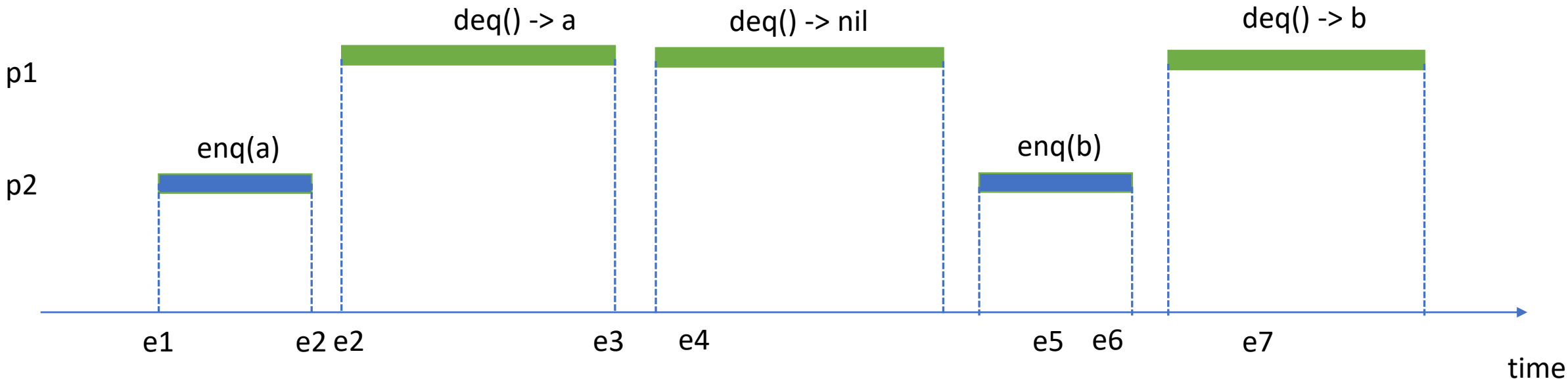


History S is a sequence of events in which e1 is an invocation, e2 is the matching response, …
…. and e7 is the matching event of e6. Thus S is a sequential history, and is complete

Shared memory model: Histories

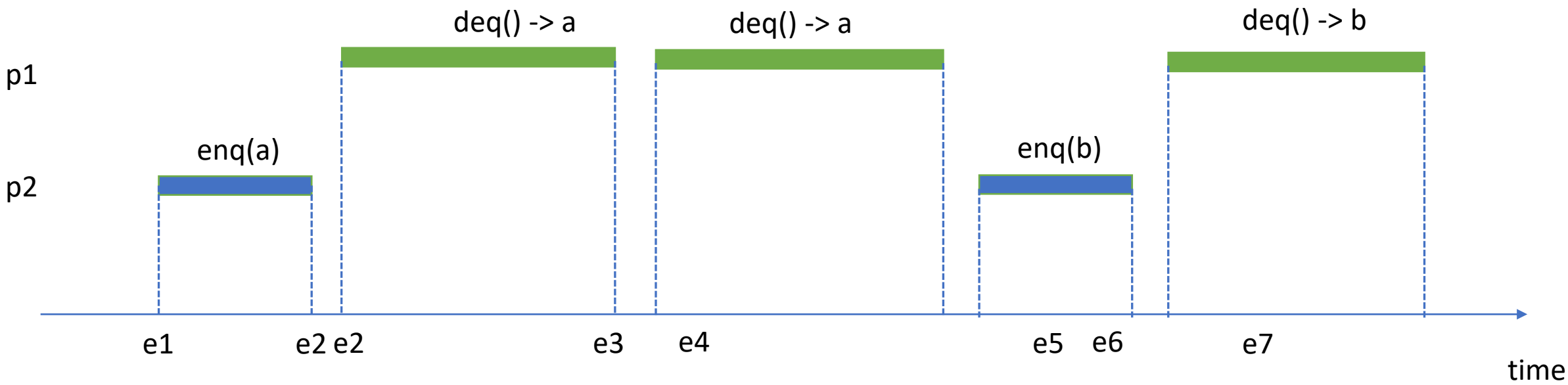
- Given a history S and an object X, H|X represents the subsequence of S composed of all the events involving only the object X
- H is legal if, for each object X in H, H|X belongs to the sequential specification of X

S: history made of the operations executed (events) on a FIFO queue



S is legal : the restriction of S to the FIFO queue belongs to the sequential specification of a FIFO queue

S: history made of the operations executed (events) on a FIFO queue



S is not legal : S | FIFO queue does not belong to the sequential specification of a FIFO queue since the second dequeue operation cannot return a

Shared memory model: Histories

Safety properties

- A condition that must hold in every finite prefix of the sequence
- It states that nothing bad has happened yet!
- Ex: traffic light: all green

Liveness properties

- A condition that must hold a certain number of times (probably an infinite number of times)
- It states that eventually something good must happen!
- Ex: traffic light: eventually one green

Shared Memory System

Complexities measures

In shared memory systems, there are no messages to count

The focus is on the **space complexity**, i.e., the amount of shared memory needed to solve given problem.

The amount shared memory is measured in two ways:

- the number of distinct shared registers/variables, and
- the amount of shared space (i.e. number of bits, equivalently, how many distinct values)

required by the algorithm.

The mutual exclusion problem

- We now see a classic problem that classically arrives in distributed computing
- The mutual exclusion problem that allows to access a given resource in a mutually exclusive way

Mutual exclusion problem

The problem concerns a group of processors which occasionally need access to some resource that cannot be used simultaneously by more than a single processor.

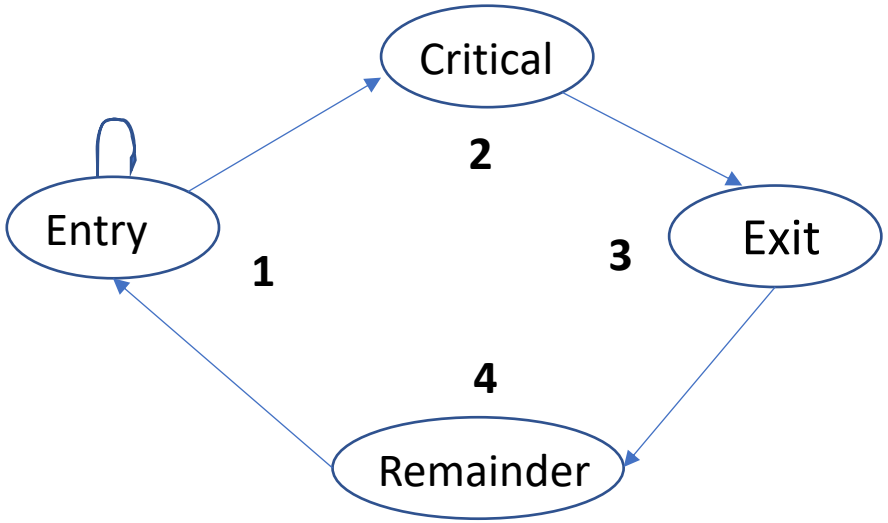
- E.g a printer
- A record of a shared database or a shared data structure, etc.

Each processor may need to execute a code segment called **critical section**, such that at any time:

- at most one processor is in the critical section (mutual exclusion)
- If one or more processors try to enter the critical section, then one of them eventually succeeds as long as no processor stays in the critical section forever (deadlock prevention)

Mutual exclusion problem

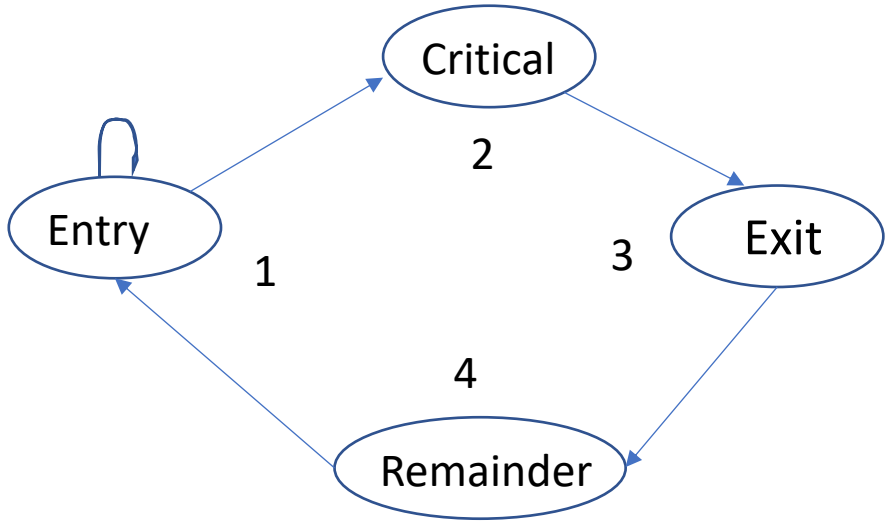
- One can partition the code of a processor as follows
  1. **Entry**: the code executed in preparation for entering the critical section
  2. **Critical**: the code to be protected from concurrent execution
  3. **Exit**: the code executed on leaving the critical section
  4. Remainder: the rest of the code



The problem is to design the **entry** and **exit** code in a way that guarantee that the mutual exclusion and deadlock-freedom properties are satisfied.

Mutual exclusion problem

- We assume that the variables (both local and shared) accessed in the Entry and Exit sections are not accessed in the Critical and Remainder sections
- We also assume that processors do not stay forever in the critical section (no crash)



Mutual exclusion problem

- An algorithm for a shared memory system solves the mutual exclusion problem with no deadlock (no lockout) if the following holds:

1. Mutual exclusion: In every configuration of every execution, at most one processor is in the critical section
2. Deadlock-freedom: In every execution, if some processor is in the entry section in a configuration, then there is a later configuration in which some processor is in the critical section
3. Starvation-freedom: In every execution, if some processor is in the entry section in a configuration, then there is a later configuration in which that same processor is in the critical section

Note that mutual exclusion is a safety property (invariants), and deadlock-freedom and starvation-freedom are liveness properties.

Mutual exclusion problem

- There are several ways to design mutual exclusion algorithms.
- They depend on the operations and properties provided to the processors by the underlying shared memory communication system
- We distinguish 3 different families of mutual exclusion algorithms

Mutual exclusion problem

1. **Specialized hardware primitives**
   - hardware primitives are offered by the multiprocessors architectures (e.g. binary test&set registers).
   - Those primitives are more sophisticated (i.e. stronger) than simple atomic r/w registers
2. **Atomic read/write registers**
   - the only way processors communicate is through those objects
3. **Mutex without atomicity**
   - Is atomicity at a lower level required to solve atomicity at a higher level ?
   - The response is no!
   - Mutual exclusion algorithms can be design with weaker registers than atomic ones

Mutual exclusion based on specialized hardware primitives

- A binary Test&Set shared variable V is a binary variable that sets V to 1 and returns its previous value
- A Compare&Swap(old,new) shared variable V is a variable that compares whether V's current value is equal to old, and if so sets it to new and return true. Otherwise it returns false

Binary Test & Set variable (we also say Test & Set register)

- We will see that with Test&Set variables, one bit is sufficient to guarantee mutual exclusion to a CS with n competing processes with no deadlock

Binary Test & Set register

- A Binary Test&Set variable V is a binary variable that supports 2 operations
  1. **test&set**
  2. **reset**

```
test&set(V: memory address ):{
        temp:=V
        V:=1
        return (temp)
}
```

```
reset(V : memory address):{
        V:=0
        return
}
```

- The **test&set**(*V*) operation atomically reads and updates the variable:
  The variable is "tested" to see if it is equal to 0, and if so it is set to 1
  The previous value is returned
- The **reset**(*V*) operation is a write (0)

Binary Test & Set register

- Mutual Exclusion algorithm that uses a test&set register.
  n processors p1 … pn want to access to the critical section

```
Algorithm: {    // Mutual_Exclusion using  T&S – executed by pi, i=1 …n
Initially V = 0     // Shared variable
while true do
        <Entry>:
        wait until (test&set(V) = 0)
        <Critical Section>: /* execute cs */
        <Exit>:
        Reset(V)
        <Remainder>: /* execute the rest of the code */
}
```

Binary Test & Set register

- Mutual Exclusion algorithm that uses a test&set register

Algorithm: *{ // Mutual_Exclusion using T&S*

Initially V = 0

while true do

<Entry>:

*wait until (**test&set(V)** = 0)*

<Critical Section>:

<Exit>:

Reset(V)

<Remainder>:

}

Assume that initially V=0

- In the entry section, p_i repeatedly test V until it returns 0
- The last test by p_i assigns 1 to V, which causes the next test to return 1, prohibiting any p_j, $i \neq j$, from entering the CS

Binary Test & Set register

Algorithm: *{*

Initially V = 0

while true do

<Entry>:

*wait until (**test&set(V)** = 0)*

<Critical Section>:

<Exit>:

Reset(V)

<Remainder>:

}

Th: Mutual exclusion is guaranteed

Proof: by contradiction

- Suppose both p_i and p_j are in the CS together
- Let t be the earliest time at which both p_i and p_j are in the CS
- Assume that p_i is already in the CS when p_j enters it
- When p_i enters the CS, it tests V, sees that V=0, and sets V to 1
- V remains equal to 1 until some processor leaves the CS
- By assumption of t, no processor other than p_i is in the CS when p_j enters the CS
- Thus V is always equal to 1 up to the time p_j enters the CS
- Thus when p_j tests V it should return 1 and not 0
- Thus p_j cannot enter the CS
- A contradiction

Binary Test & Set register

Algorithm: *{*

Initially V = 0

while true do

<Entry>:

*wait until (**test&set(V)** = 0)*

<Critical Section>:

<Exit>:

Reset(V)

<Remainder>:

}

Th: No Deadlock is guaranteed

Proof: by contradiction

- Suppose that there is an execution in which, after some point at least p_i is in the Entry section but no processor ever enters the CS
- Since no processor remains forever in the CS then there is some point from which at least p_i is in the Entry section but no processor is in the CS
- The key is to note that V=0 iff no processor is in the CS
- Thus any processor that executes the first line of the algorithm discovers that V=0 and enters the CS
- A contradiction

Binary Test & Set register

Algorithm: *{*

Initially V = 0

while true do

<Entry>:

*wait until (**test&set(V)** = 0)*

<Critical Section>:

<Exit>:

Reset(V)

<Remainder>:

}

Th: This algorithm guarantees mutual exclusion without deadlock with one binary Test&Set register

The algorithm does not provide starvation-freedom: nothing prevents a single fast process from grabbing at V every time it goes through the outer loop, while the other processes grab at it just after it is taken away. Lockout-freedom requires a more sophisticated turn-taking strategy.

A starvation-free algorithm using an atomic queue (FIFO)

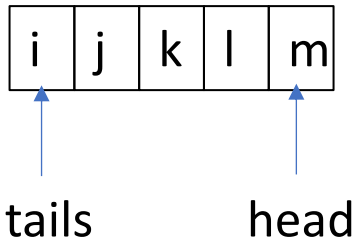
Basic idea:

- In the trying phase, each process enqueues itself at the end of a shared queue (assumed to be an atomic operation).
- When a process comes to the head of the queue, it enters the critical section, and when exiting it dequeues itself.

A starvation-free algorithm using an atomic queue (FIFO)

Shared (FIFO) queue

- enq(Q, id) : inserts id at the tail of the queue
- deq(Q): removes the first element of the head of the queue
- head(Q): returns the first element of the queue



A starvation-free algorithm using an atomic queue (FIFO)

```
Algorithm: {    // Mutual_Exclusion

while true  do
        <Entry>:
        enq(Q,myId)
        while (head(Q) not equal to myId) do nothing
        <Critical Section>:
        ….
        <Exit>:
        deq(Q)
        <Remainder>:
        ….
}
```

Mutual exclusion:

Idea: only the process whose id is at the head of the queue can enter its critical section.

The following invariant holds:

« any process that is between the inner while loop and the call to deq(Q) must be at the head of the queue »

Proof:

- a process can't leave the while loop unless the test fails (i.e., it is already at the head of the queue),
- no enq operation changes the head value (if the queue is nonempty),
- and the deq operation (which does change the head value) can only be executed by a process already at the head.

A starvation-free algorithm using an atomic queue (FIFO)

```
Algorithm: {    // Mutual_Exclusion

while true  do
      <Entry>:
      enq(Q,myId)
      while (head(Q) not equal to myId) do nothing
      <Critical Section>:
      ….
      <Exit>:
      deq(Q)
      <Remainder>:
      ….
}
```

Starvation freedom

Proof:

- any element of the queue is the id of some process in the trying, critical, or exiting sections
- so eventually the process at the head of the queue passes the inner loop, executes its critical section, and dequeues its id

Reducing the complexity

- We can give an implementation of this algorithm using a single read-modify-write (RMW) register instead of a queue;
- This drastically reduces the (shared) space needed by the algorithm
- log(n) bits are sufficient to guarantee mutual exclusion with no starvation

A starvation-free algorithm using RMW shared object

- A read-modify-write (RMW) object V is a variable that allows a process to read the current value of the variable, compute a new value as a function of the current value of the variable, and write the new value to the variable
- This is achieved in one atomic operation: rmw(V,f)
- The rmw(V,f) returns the previous value of the variable

- that supports 1 operation : rmw

```
rmw(V: memory address, f: function ):{
        temp:=V
        V:=f(V)
        return (temp)
}
```

A starvation-free algorithm using RMW shared object

- Formally, a rmw operation is defined as follows:

```
rmw(V: memory address, f: function ):{
        temp:=V
        V:=f(V)
        return (temp)
}
```

- The rmw operation takes a function f that specifies how the new value is related to the old one
- The Test&Set object is a special case of a RMW objet, where f(V)=1

A starvation-free algorithm using RMW shared object

- We now present a mutual exclusion problem that uses only one sharded RMW object
- The algorithm organizes all the processes that want to access the critical section as if they were in a FIFO queue
- Only the process that is at the head of the queue is allowed to enter the critical section
- Each process locally maintains two variables (local): position and queue
- The algorithm uses a RMW register V that contains two fields: first and last
- V.first contains the ticket of the process at the head of the queue
- V.last contains the ticket of the process at the end of the queue

A starvation-free algorithm using RMW shared object

- When a new process pi arrives in the entry section, it “enqueues” in the queue by setting V to its local variable position and by incrementing V.last (achieved atomically)
    - position = rmv(V, <V.first,V.last+1>)
- The current value of V.last serves as pi’s ticket
- Process pi waits until it is at the head of the queue, i.e., until V.first is equal to its ticket position.last
- At this point pi enters the critical section

A lockout-free algorithm using RMW shared object

```
Algorithm: {   // Mutual_Exclusion with V

Initially V =<0,0>

while true  do
    <Entry>:
    position :=  RMW(V, <V.first, V.last+ 1>)
    repeat
        queue:= RMW(V, <V.first, V.last>)
    until (queue.first =position.last)
    <Critical Section>:
    ....
    <Exit>:
    RMW(V,<V.first+1,V>)
    <Remainder>:
    ....
}
```

The RMW has two fields : *first* and *last*
(initially both fields are equal to 0).

Incrementing *last* simulates an enqueue
Incrementing *first* simulates a dequeue

Trick: instead of testing if I am at the head of the queue
I simply remember the value of the *last* field when I « enqueued » myself, and waits for the *first* field to be equal to my id

A lockout-free algorithm using RMW shared object

```
Algorithm: {   // Mutual_Exclusion with V
Initially V =<0,0>
while true  do
    <Entry>:
    position :=  RMW(V, <V.first, V.last+ 1>)
    repeat
        queue:= RMW(V, <V.first, V.last>)
    until (queue.first =position.last)
    <Critical Section>:
    ….
    <Exit>:
    RMW(V,<V.first+1,V>)
    <Remainder>:
    ….
}
```

Mutual exclusion is satisfied

Proof:

- Only the process at the head of the queue can enter the CS
- Thus all the other processes cannot enter the CS

Lockout-freedom is satisfied

Proof:

- The FIFO construction together with the fact that processes do not stay forever in the CS provide the lock-out freedom property

A lockout-free algorithm using RMW shared object

Algorithm: { *// Mutual_Exclusion with V*

Initially V =<0,0>

while true do

<Entry>:

position := RMW(V, <V.first, V.last+ 1>)

repeat

queue:= RMW(V, <V.first, V.last>*)*

until (queue.first =position.last)

<Critical Section>:

….

<Exit>:

RMW(V,<V.first+1,V>)

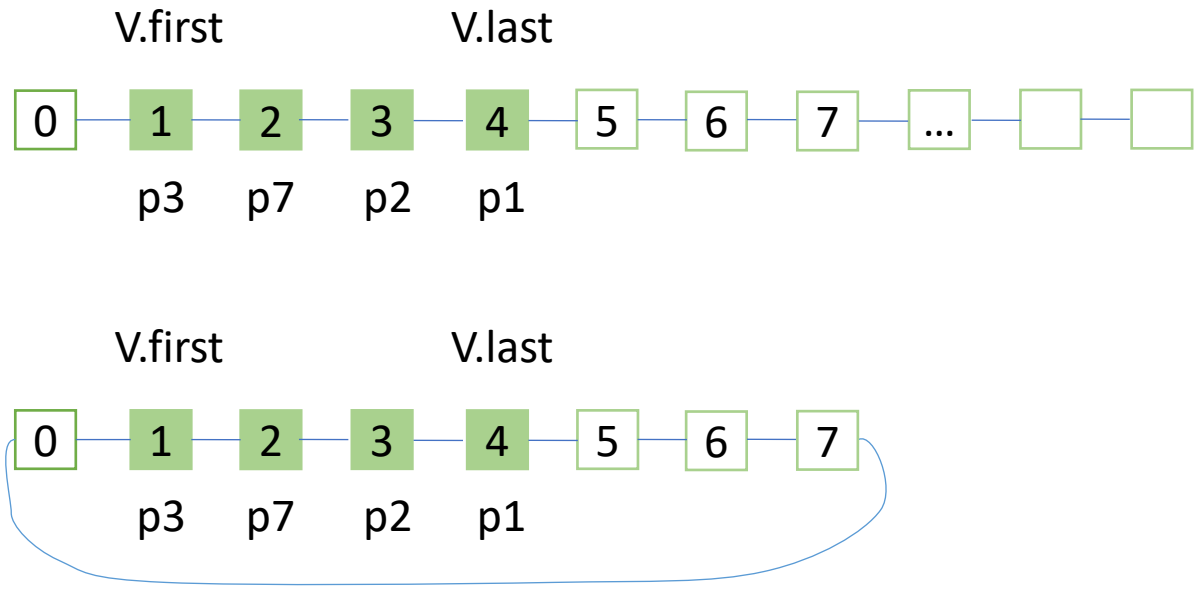
<Remainder>:

….

}

Note that no more than *n* processes can be in the queue at the same time.

Thus all the additions can be done modulo n
Thus the RMW object has 2 ($\log_2$ n)



Mutual exclusion based on atomic Read/Write objects

- We now concentrate on systems in which processors access the shared variables only by read and write operations
- We will present two algorithms that provide mutual exclusion and no starvation for n processors
- Both algorithms use O(n) distinct shared registers/variables

Read/Write Register

1 – Which kind of information is contained in a register?

binary (boolean) / multivalued

2 – Who is allowed to access a register ?

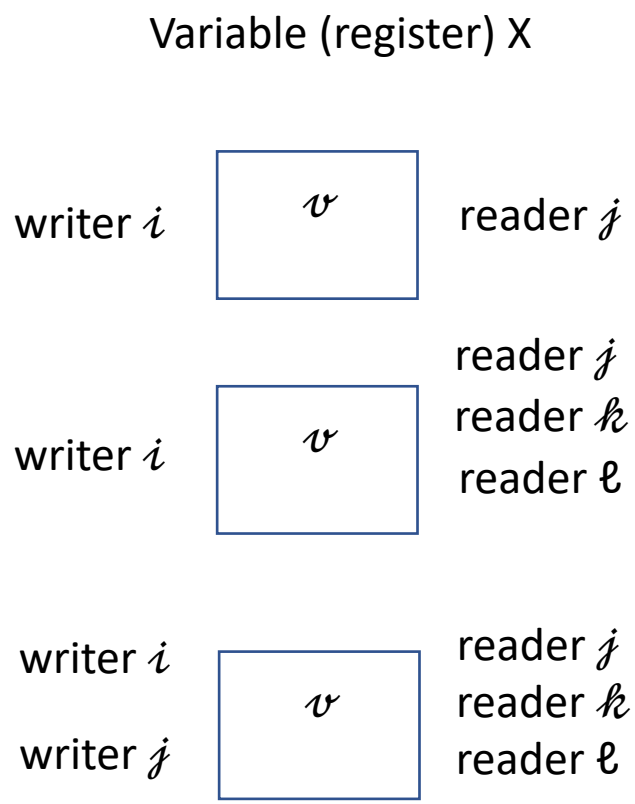
SWSR (Single Writer / Single Reader)

SWMR (Single Writer / Multiple Readers)

MWMR (Multiple Writer / Multiple Readers)

3 – How behave the register when concurrently accessed ?

safe – regular - atomic

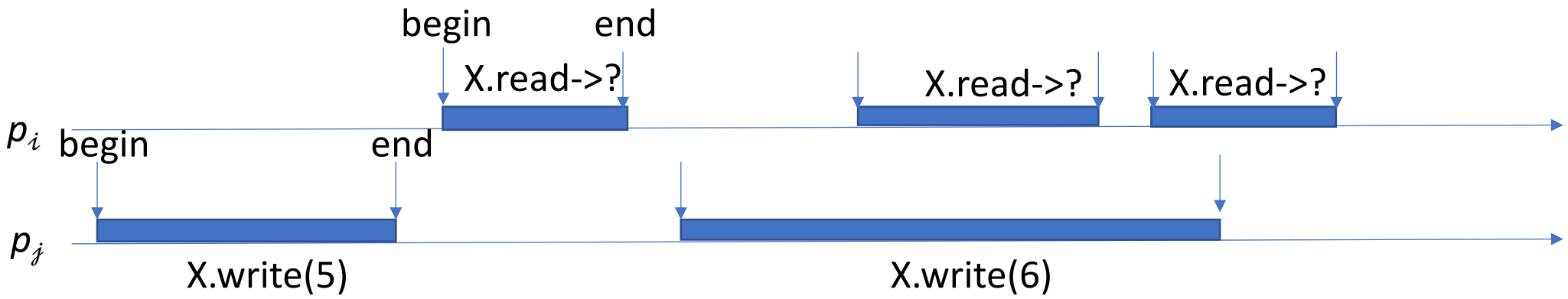


Safe read/write register

Properties:

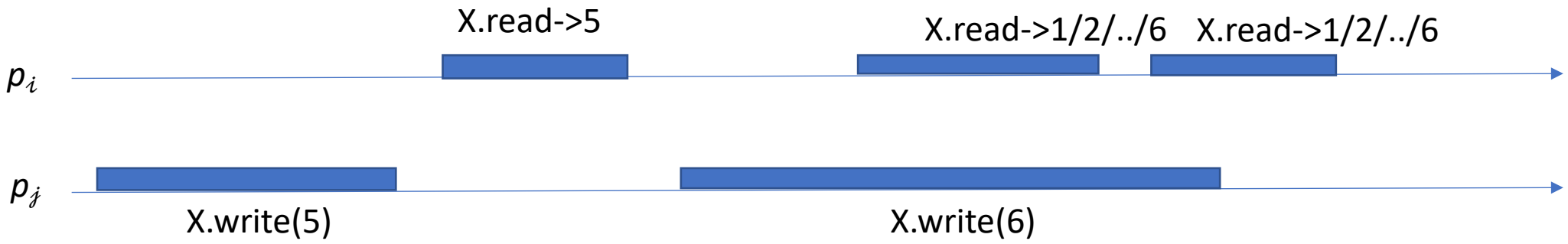
- a read() not concurrent with any write() obtains the correct value, i.e., the most recently written one
- a read() that overlaps a write() returns any possible values of the register
- This is the simplest kind of register we can think of to enable any 2 processes to communicate

Safe read/write register



[1,..6]-valued safe register X

Safe read/write register



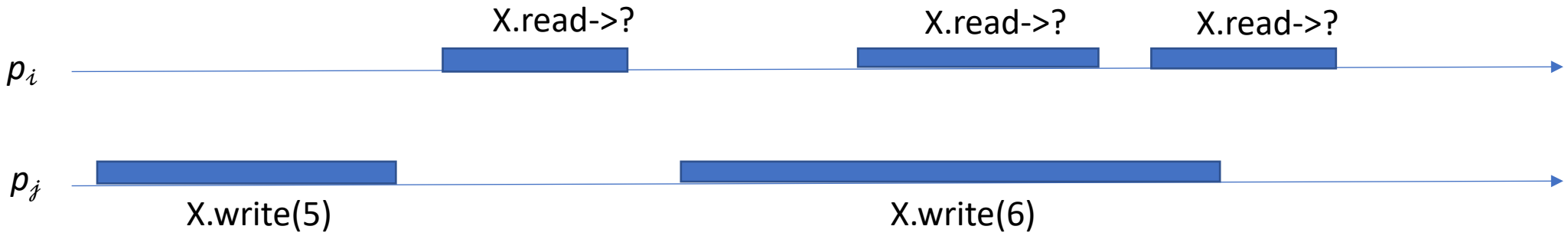
[1,..6]-valued safe register X

Regular read/write register

Properties:

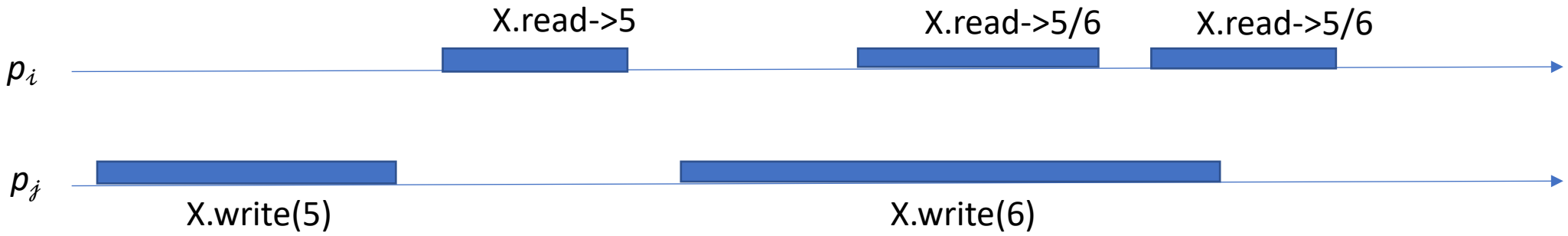
- a read() not concurrent with any write() obtains the correct value, i.e., the most recently written one
- a read() that overlaps a write() returns either the old or the new written value
- Provides stronger properties than the safe one

Regular read/write register



[1,..6]-valued regular register X

Regular read/write register

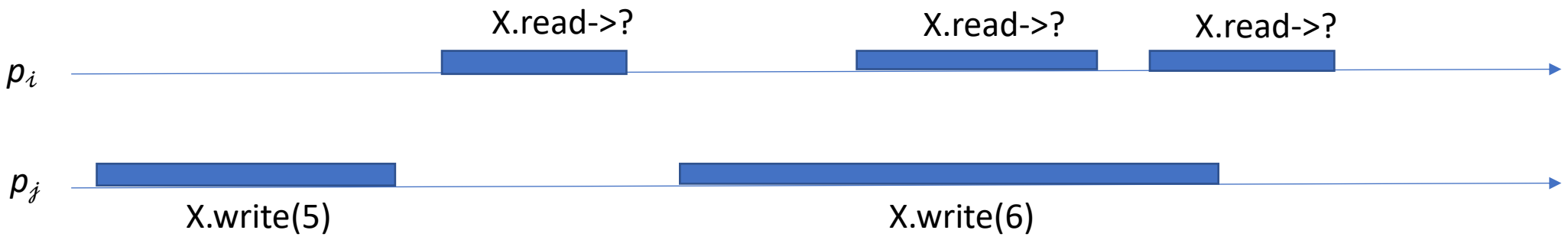


[1,..6]-valued regular register X

Atomic read/write register

Properties:

- a read() not concurrent with any write() obtains the correct value, i.e., the most recently written one

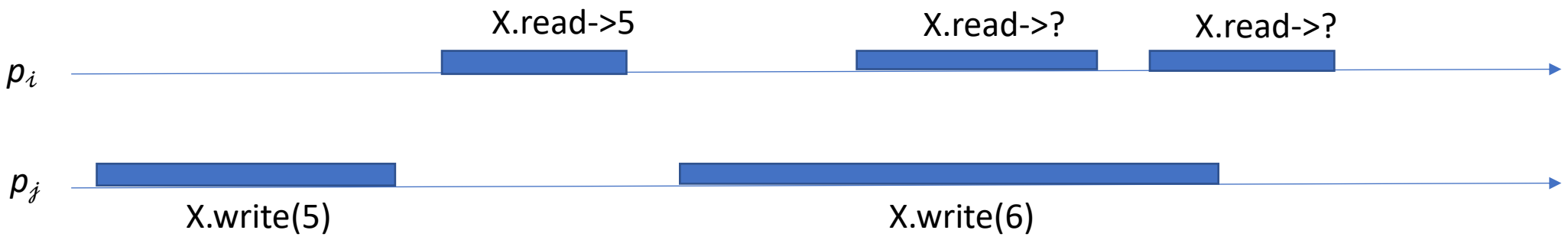


[1,..6]-valued regular register X

Atomic read/write register

Properties:

- A read() not concurrent with any write() obtains the correct value, i.e., the most recently written one
- Concurrent reads() and writes() behave as if they occur in some definite order, i.e., as if reads() and writes() occur sequentially

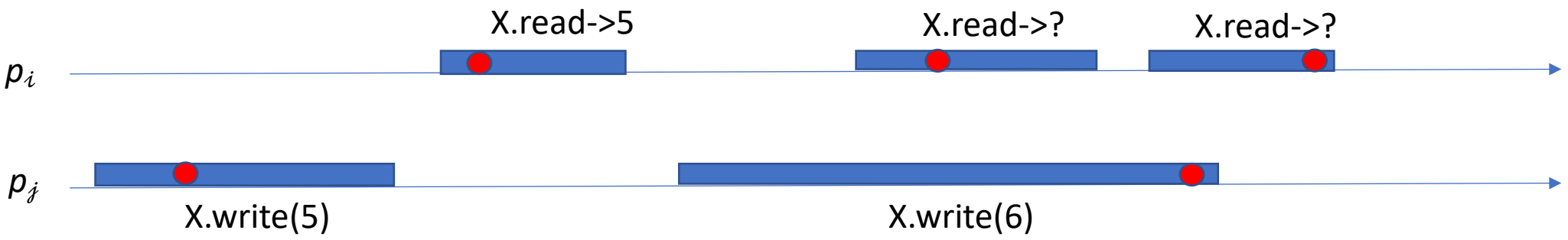


[1,..6]-valued atomic register X

Atomic read/write register

More precisely, in presence of concurrency

- (a) Each operation invocation appears as if it had been executed instantaneously at a point of the time line between its start event and its end event, and
- (b) the resulting sequence of invocations is such that any read invocation returns the value written by the closest preceding write invocation.

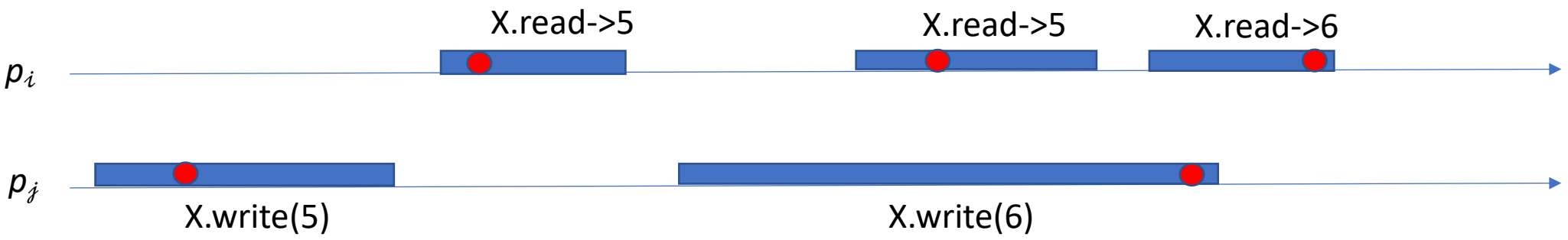


[1,..6]-valued atomic register X

Atomic read/write register

More precisely, in presence of concurrency

- (a) Each operation invocation appears as if it had been executed instantaneously at a point of the time line between its start event and its end event, and
- (b) the resulting sequence of invocations is such that any read invocation returns the value written by the closest preceding write invocation.

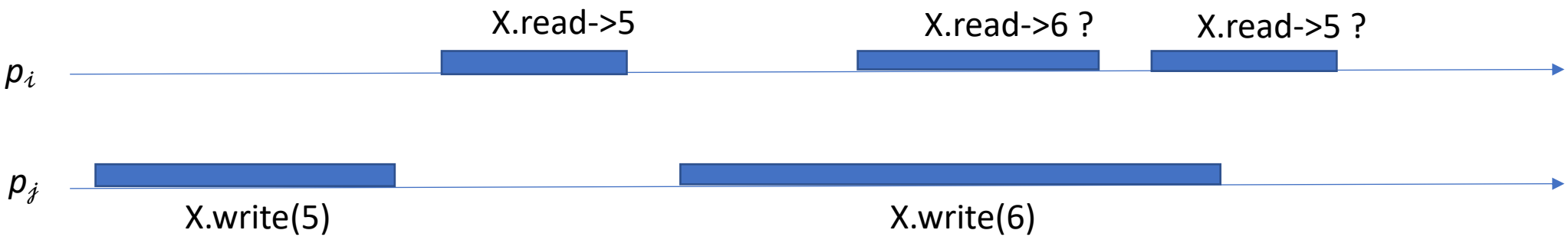


[1,..6]-valued atomic register X

Atomic read/write register

More precisely, in presence of concurrency

- (a) Each operation invocation appears as if it had been executed instantaneously at a point of the time line between its start event and its end event, and
- (b) the resulting sequence of invocations is such that any read invocation returns the value written by the closest preceding write invocation.



[1,..6]-valued atomic register X

Atomic Read/Write Register

Properties of an atomic shared register

1. Each invocation of a read or write operation
   1. Appears as if it was executed at a single point ℓ(op) of the time line
   2. ℓ(op) is such that $s(\text{op}) \leq \ell(\text{op}) \leq e(\text{op})$
   3. For any two operations op1 and op2: $(\text{op1} \neq \text{op2}) \implies \ell(\text{op1}) \neq \ell(\text{op2})$.

2. Each read invocation returns the value written by the closest preceding write operation in the sequence defined by the $\ell()$ instants associated with the invocations.

➢ This means that an atomic register is such that all its operations invocations *appear* as if they have been executed sequentially

Atomic Read/Write Register

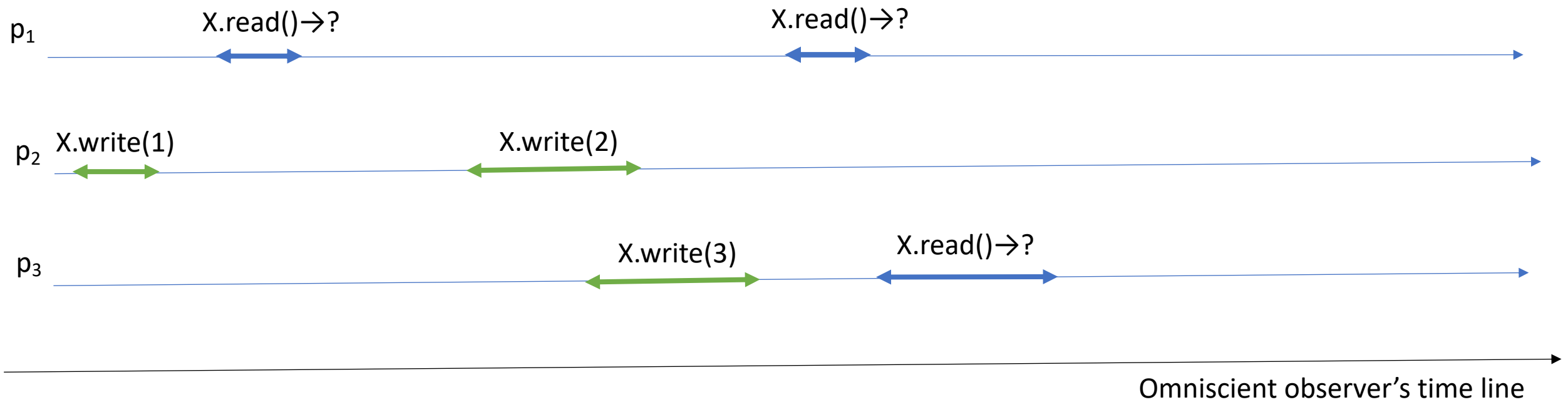
Properties of an atomic shared register

An atomic read-write register can be

- single-writer/single-reader (SWSR) : a single process can write the register and a single one can read it, both processes being different
- single-writer/multiple-reader (SWMR) : a single process can write the register but it can be read by multiple processes
- multiple-writer/multiple-reader (MWMR) : the register can be read and written by multiple processes

Atomic Read/Write Register

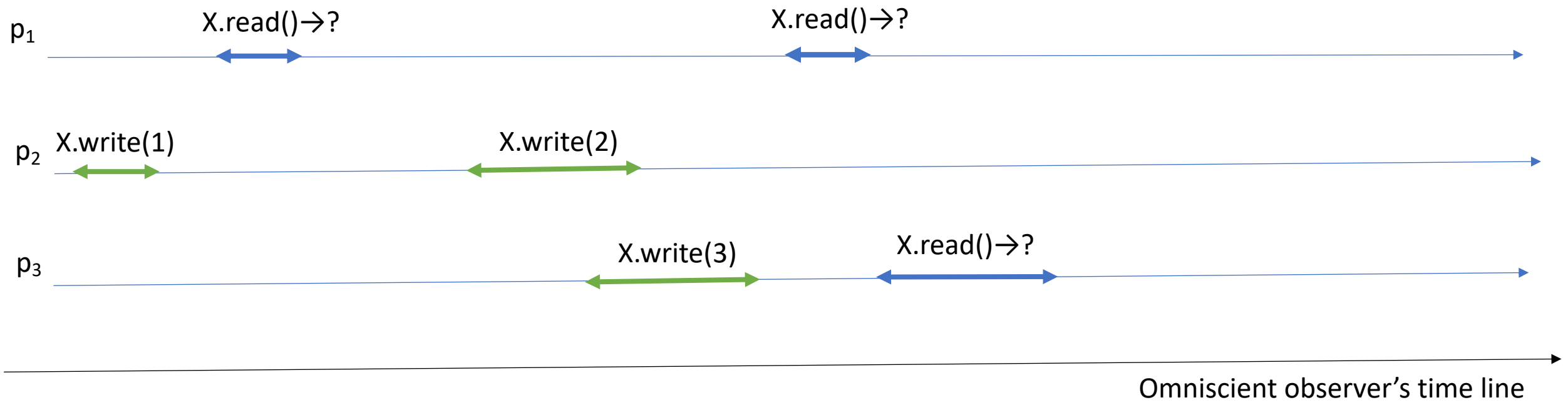
Example of a multi-writer- multi-reader register X



Space-time diagram depicting an execution of a MWMR register

Atomic Read/Write Register

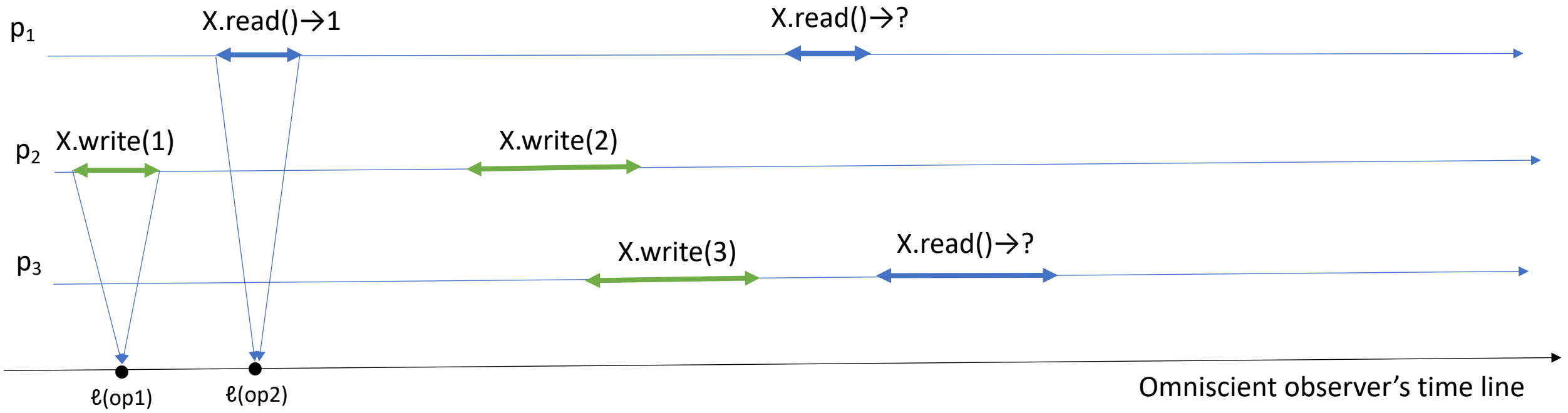
Question: What are the correct execution(s) ?



Space-time diagram depicting an execution of a MWMR register

Atomic Read/Write Register

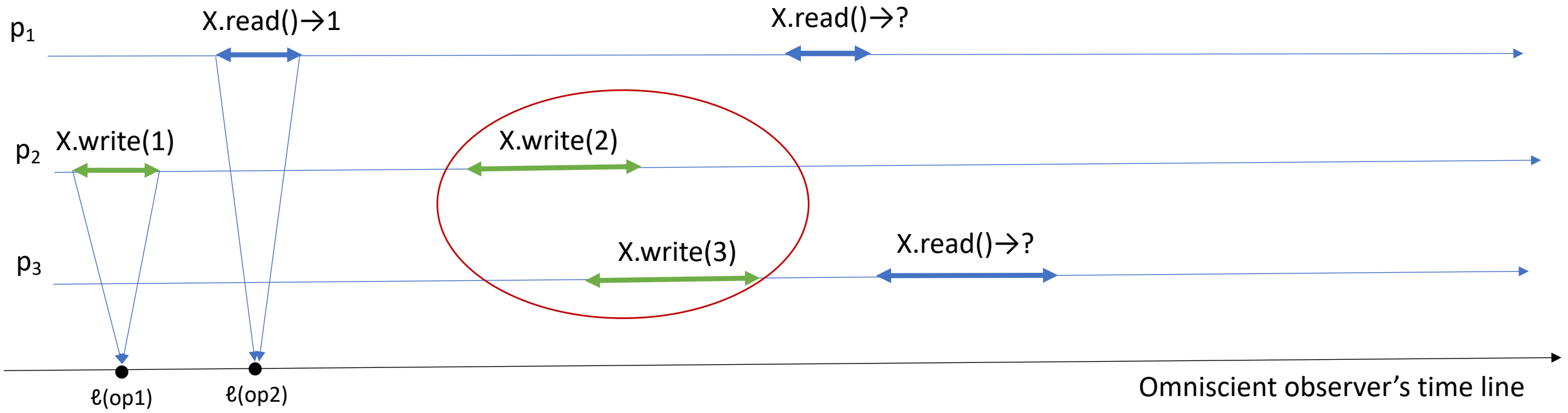
Question: What are the correct execution(s) ?



X.write(1), X.read() →1,

Atomic Read/Write Register

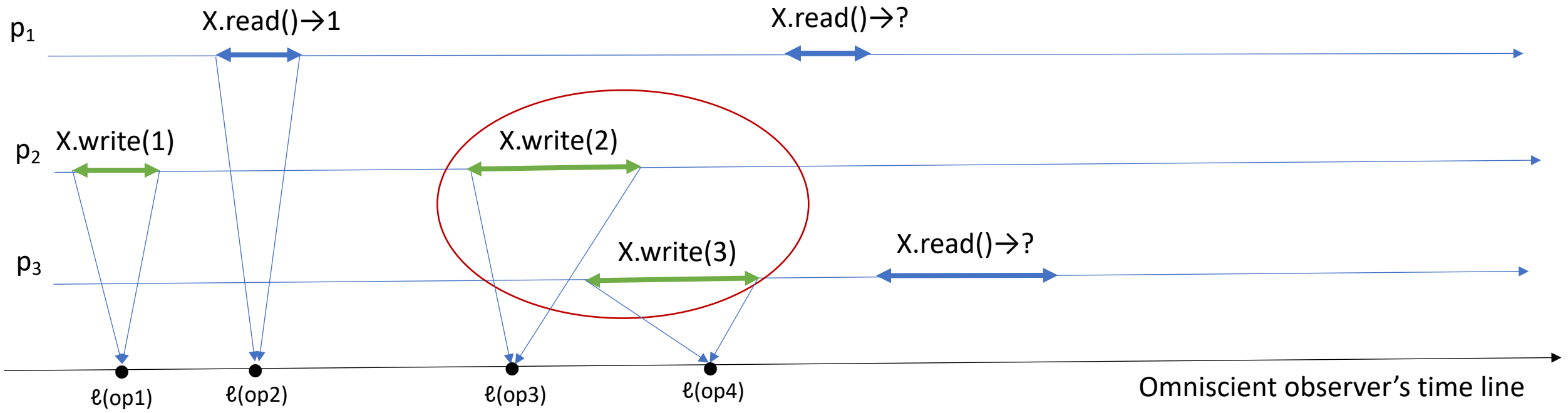
Questions What are the correct execution(s) ?



X.write(1), X.read() →1,

Atomic Read/Write Register

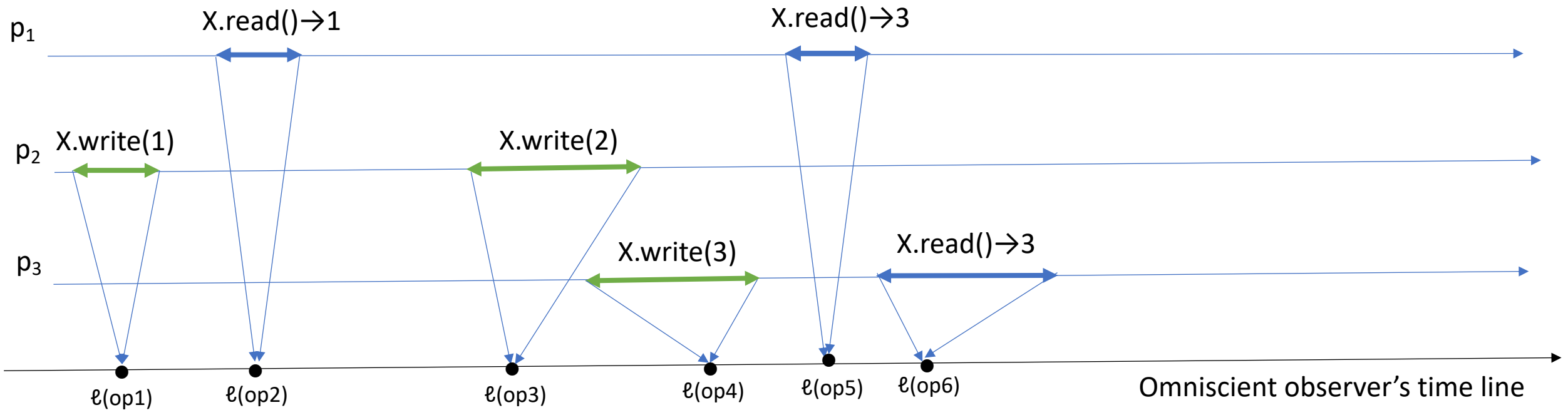
Questions What are the correct execution(s) ?



X.write(1), X.read() →1, X.write(2), X.write(3),

Atomic Read/Write Register

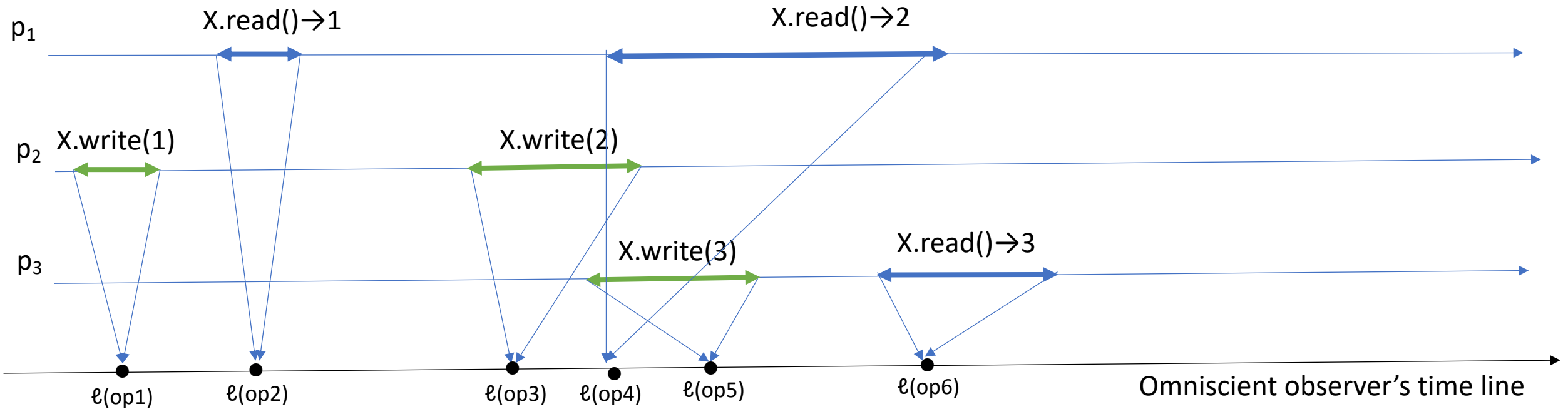
Questions What are the correct execution(s) ?



X.write(1), X.read() →1, X.write(2), X.write(3), X.read()→3, X.read()→3

Atomic Read/Write Register

Question: What are the correct execution(s) ?



X.write(1), X.read() →1, X.write(2), X.read()→2, X.write(3), X.read()→3

Atomic Read/Write Register

Concurrency is intimately related to non-determinism

- It is not possible to predict which execution will be produced
- It is only possible to enumerate the set of possible correct executions that could be produced

• Why Atomicity is important ?

- Atomicity allows the composition of shared objects for free
- i.e. if you consider two atomic registers X1 and X2, the composite object [X1,X2] made of X1 and X2, and which offers to processes 4 operations, is also atomic
- Everything appears as if at most one operation at a time was executed
- So we can reason on sequences not only for each register taken separately but also on the whole set of registers as if they were a single atomic register

“Mutual exclusion” using SWMR atomic registers

Mutual exclusion

- The algorithm ensures mutual exclusion without deadlock for two processors p_0 and p_1
- It is due to Peterson (1981)
- This algorithm gives priority to one processor p_0
- We will convert this algorithm to one that guarantees no starvation

- Each processor p_i uses 1 binary SWMR atomic register **Want[i]**
  - ***Want*[*i*]**=1 if processor p_i is interested in entering the critical section and
  - **Want[i]**=0 otherwise

"2-Mutual exclusion" using SWMR atomic registers

Algorithm: Mutual exclusion for two processes p_0 and p_1

Initially ***Want[0]***=0; ***Want[1]***=0;

Code executed by p_0	Code executed by p_1
<Entry>:	<Entry>:
1:	1: ***Want[1]***:=0
2:	2: wait until (***Want[0]***=0)
3: ***Want[0]***:=1	3: ***Want[1]***:=1
4:	4:
5:	5: if (***Want[0]*** = 1) then goto Line1
6:wait until ***Want[1]***=0	6:
<Critical Section>	<Critical Section>
<Exit>:	<Exit>:
7:	7:
8: ***Want[0]***=0	8: ***Want[1]***=0
<Remainder>	<Remainder>

The algorithm is asymmetric:

- p_0 enters the CS whenever CS is empty, without considering that p_1's wants to do so
- p_1 enters the CS only if p_0 is not interested in it at all

p_i raises its flag to notify that it wishes to enter the CS

p_i observes p_{1-i}'s flag, and if up waits

If p_i executes this part of the code then Want[i]=1

"2-Mutual exclusion" using SWMR atomic registers

Algorithm: Mutual exclusion for two processes p_0 and p_1

Initially ***Want[0]***=0; ***Want[1]***=0;

Code executed by p_0	Code executed by p_1
<Entry>:	<Entry>:
1:	1: ***Want[1]***:=0
2:	2: wait until (***Want[0]***=0)
3: ***Want[0]***:=1	3: ***Want[1]***:=1
4:	4:
5:	5: if (***Want[0]*** = 1) then goto Line1
6:wait until ***Want[1]***=0	6:
<Critical Section>	<Critical Section>
<Exit>:	<Exit>:
7:	7:
8: ***Want[0]***=0	8: ***Want[1]***=0
<Remainder>	<Remainder>

Th: The algorithm provides Mutual Exclusion

Proof: by contradiction

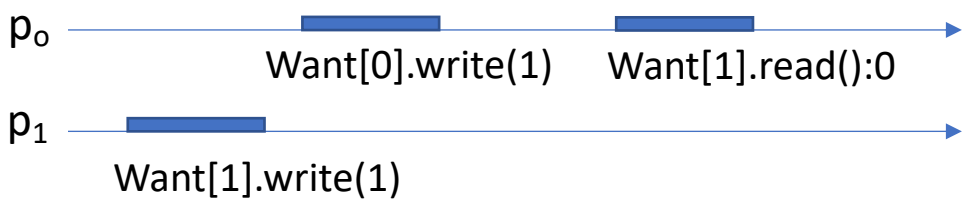
- Assume both p_0 and p_1 are in the CS
- By construction both ***Want[i]***=1
- Assume that wlog that p_0 's last write to ***Want[0]*** *before entering* the CS follows p_1 's last write to ***Want[1]*** *before entering* the CS
- We have ***Want[0]***.write(1) < ***Want[1]***.read()

and ***Want[1]***.write(1) < ***Want[0]***.write(1),

- thus p_0's read of ***Want[1]*** should return 1

(***Want*** registers are atomic, thus return the last written value).

A contradiction



“2-Mutual exclusion” using 2 SWMR and 1 MWMR atomic registers

- The algorithm is deadlock-free (left as exercise)
- However it is not starvation-free: if p_0 is continuously interested in entering the CS then p_1 will never enter it because it gives up when p_0 is interested.
- To achieve no lockout we modify the algorithm so that instead of always giving priority to p_0, each process gives priority to the other process on leaving the CS

- Each processor p_i uses 1 binary SWMR register **Want[i]**
  - ***Want[i]***=1 if processor p_i is interested in entering the critical section and
  - **Want[i]**=0 otherwise
- and 1 MWMR register **Priority**
  - ***Priority*** = i if processor p_i has priority at the moment. It is initialized to *0.*

The process with the priority will execute the code of p_0

“2-Mutual exclusion” using 2 SWMR and 1 MWMR atomic registers

Algorithm: 2-Mutual exclusion
Code executed by p_i

Initially **Want[i]**=0; ***Priority***:=0

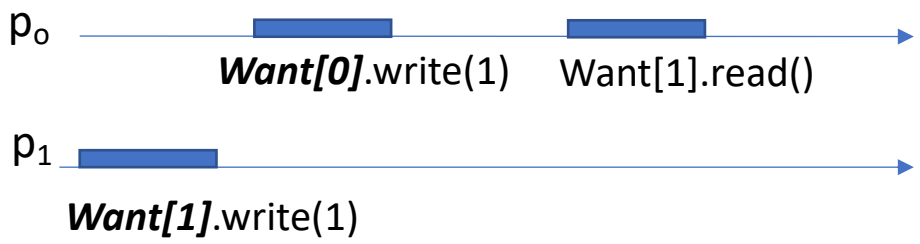
<Entry>:
1: *Want[i]*:=0
2: wait until (*Want[1-i]*=0 or *Priority* = i)
3: *Want[i]*:=1
4: if (*Priority* =1-i) then
5: if (*Want[1-i]* = 1) then goto Line 1
6: wait until *Want[1-i]*=0
<Critical Section>
<Exit>:
7: *Priority* =1-i
8:*Want[i]*=0
<Remainder>

Th: The algorithm provides Mutual Exclusion

Proof: by contradiction
Assume both p_0 and p_1 are in the CS

- By construction both ***Want[i]***=1
- Assume that wlog that p_0 ‘s last write to ***Want[0]*** *before entering* the CS follows p_1 ‘s last write to ***Want[1]*** *before entering* the CS
- Note that p_0 can enter the CS either through Line 5 or Line 6.
- In both cases p_0 reads ***Want[1]***=0.
- However by assumption we have “**Want[0]**.write(1)” < “**Want[1]**.read()” and “***Want[1]***.write(1)” < “**Want[0]**.write(1)”, and thus

p_0 reads **Want[1]** should return 1
A contradiction



“2-Mutual exclusion” using 2 SWMR and 1 MWMR atomic registers

Algorithm: 2-Mutual exclusion for two processes
Code executed by p_i

Initially **Want[i]**=0; ***Priority***:=0

<Entry>:
1: ***Want[i]***:=0
2: wait until (***Want[1-i]***=0 or ***Priority*** = i)
3: ***Want[i]***:=1
4: if (***Priority*** =1-i) then
5: if (***Want[1-i]*** = 1) then goto Line 1
6: wait until ***Want[1-i]***=0
<Critical Section>
<Exit>:
7: ***Priority*** =1-i
8: ***Want[i]***=0
<Remainder>

Th: The algorithm is without deadlock

Proof: by contradiction
Suppose that there is a point after which some process is forever in the <Entry> section, and no processor enters the CS

Case 1: both p_0 and p_1 are in the <Entry> section.

- Thus ***Priority*** never changes. Wlog ***Priority=0***.
- Thus p_0 passes the test in Line 2 and loops forever in Line 6 with ***Want[0]***:=1
- Since ***Priority=0***, p_1 does not reach Line 6 and wait in Line 2, with ***Want[1]***:=0.
- Thus p_0 will pass the test in Line 6 and enters the CS.

A contradiction.

Case 2: A single process is forever in the <Entry> section. Say p_0.

- Since p_1 does not stay forever in the CS or in the <Exit> section, it will set ***Priority*** =0 and ***Want[1]***=0 forever.
- Thus p_0 does not loop forever in the <Entry> section (lines 2,5,6)
- and enters the CS.

A contradiction.

“2-Mutual exclusion” using 2 SWMR and 1 MWMR atomic registers

Algorithm: 2-Mutual exclusion for two processes
Code executed by p_i

Initially **Want[i]**=0; ***Priority***:=0

<Entry>:
1: ***Want[i]***:=0
2: wait until (***Want[1-i]***=0 or ***Priority*** = i)
3: ***Want[i]***:=1
4: if (***Priority*** =1-i) then
5: if (***Want[1-i]*** = 1) then goto Line 1
6: wait until ***Want[1-i]***=0
<Critical Section>
<Exit>:
7: ***Priority*** =1-i
8: ***Want[i]***=0
<Remainder>

Th: The algorithm is starvation-free

Proof: by contradiction
Suppose that there is a configuration after which some process is starved and thus is forever in the <Entry> section. Wlog this is p_0.

Case 1: Suppose p_1 executes line 7 at some later point.

- ***Priority*** = 0 and Want[1]= 0 forever after.
- Thus p_0 passes the test Line 2 and skips Line 4
- Thus p_0 executing line 6
- Then p_0 cannot be stuck in the entry section!

Case 2: p_1 never executes line 7 at any later point.

- Since no-deadlock holds, p_1 is forever in the remainder section.
- Thus, want[1] = 0 henceforth.
- Then p_0 cannot be stuck in the entry section!

A contradiction.

Mutual exclusion without relying on atomic primitives

- So far, we have seen that one can solve mutual exclusion either by using high level hardware primitives or by using atomic objects, that is objects which, when concurrently accessed by different processes, behave as if they were accessed sequentially

- We will now see that we can solve mutual exclusion with weaker types of objects

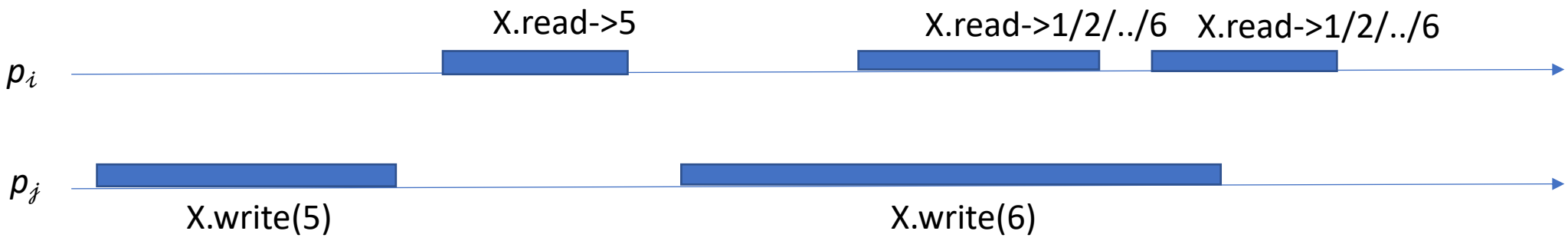
Safe read/write registers

- By doing so, we will show that we can implement atomic operations without relying on underlying atomic objects

Read/Write safe register

Properties:

- a read() not concurrent with any write() obtains the correct value, i.e., the most recently written one
- a read() that overlaps a write() returns any possible values of the register



[1,..6]-valued safe register X

Bakery Algorithm
Mutual exclusion problem using safe read/write registers

Bakery algorithm (Lamport)

- Algorithm that guarantees mutual exclusion among n processors
- Very simple algorithm that guarantees first-in-first-served property
- safe SWMR registers

Main idea

- Considering processors that wish to enter the critical section as customers in a bakery
- Each customer arriving at the bakery gets a ticket and the next with the smallest ticket is the next to be served
- A customer which is not waiting in the line is ticket “0”

Bakery Algorithm
Mutual exclusion problem using safe read/write registers

In practice

- Shared safe registers (SWMR):
  - **Number[n]** = array of n non negative integers
    - Number[i] = ticket number of processor p_i - writable solely by p_i
  - **Flag[n]** = array of n Boolean values
    - Flag[i] = true while p_i is in the process of obtaining its ticket - writable solely by p_i

- Each processor p_i wishing to enter the CS
  - chooses a number which is greater than the numbers of all the other processors and writes it to its ticket
  - after getting its ticket p_i waits until its ticket is the smallest one, and then enters the CS

Bakery Algorithm
Mutual exclusion problem using safe read/write registers

Algorithm: Bakery (code executed by processor pi, $1 \leq i \leq n$)
Initially **Flag[i]**=false and **Number[i]**=0 for $1 \leq i \leq n$

<Entry>:
1: **Flag[i]** = true
2: **Number[i]** = max(**Number[1]** ,.., **Number[n]**)+1
3: **Flag[i]** = false
4: for j=1 to n ($j \neq i$) do
5: wait until $\mathbf{Flag}[j]$=false
6: wait until **Number[j]=0 or** (**Number[j]**,j) > (**Number[i]**,i))
<Critical Section>
<Exit>:
7: **Number[i]** = 0
<Remainder>:

1. Operations on registers are not atomic, i.e., they cannot be abstracted as having been executed "instantaneously"
   - ➢ Thus we need to consider their beginning and ending times
2. Terminology:
   p_i is "in the doorway" when it executes L2
   p_i is "in the bakery" when it executes lines L3-7
3. Note that several processes can be in the doorway at the same time. So to break ties, their numbers are set to (Number,index)

Bakery Algorithm

Algorithm: Bakery (code executed by processor pi, $1 \leq i \leq n$)
Initially **Flag[i]**=false and **Number[i]**=0 for $1 \leq i \leq n$

<Entry>:
1: **Flag[i]** = true
2: **Number[i]** = max(**Number[1]** ,.., **Number[n]**)+1
3: **Flag[i]** = false
4: for j=1 to n ($j \neq i$) do
5: wait until $\mathbf{Flag}[j]$=false
6: wait until **Number[j]=0 or** (**Number[j]**,j) > (**Number[i]**,i))
<Critical Section>
<Exit>:
7: **Number[i]** = 0
<Remainder>:

Lemma 1: Let p_i and p_j be both in the bakery, and such that p_i entered the bakery (L3-7) before p_j entered the doorway (L2). Then

Number[i] < **Number[j]**

Proof

- Since p_i enters the bakery before p_j entered the doorway, **Number[i]** was written before p_j reads it.
- Thus there is no concurrent access to **Number[i]**
- And thus p_j reads the correct value of **Number[i]**
- Thus **Number[j]** $\geq$ **Number[i]+1**

Bakery Algorithm

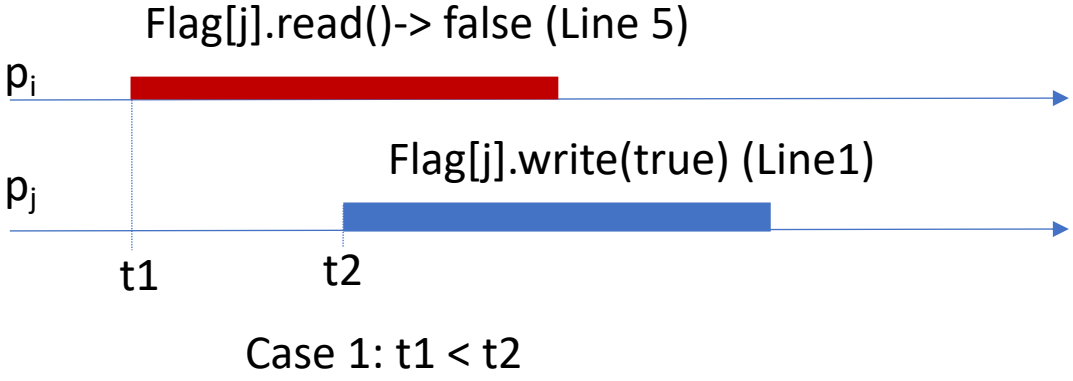
Algorithm: Bakery (code executed by processor pi, $1 \leq i \leq n$)
Initially **Flag[i]**=false and **Number[i]**=0 for $1 \leq i \leq n$

<Entry>:
1: **Flag[i]** = true
2: **Number[i]** = max(**Number[1]** ,.., **Number[n]**)+1
3: **Flag[i]** = false
4: for j=1 to n ($j \neq i$) do
5: wait until $\mathbf{Flag}[j]$=false
6: wait until **Number[j]=0 or** (**Number[j]**,j) > (**Number[i]**,i))
<Critical Section>
<Exit>:
7: **Number[i]** = 0

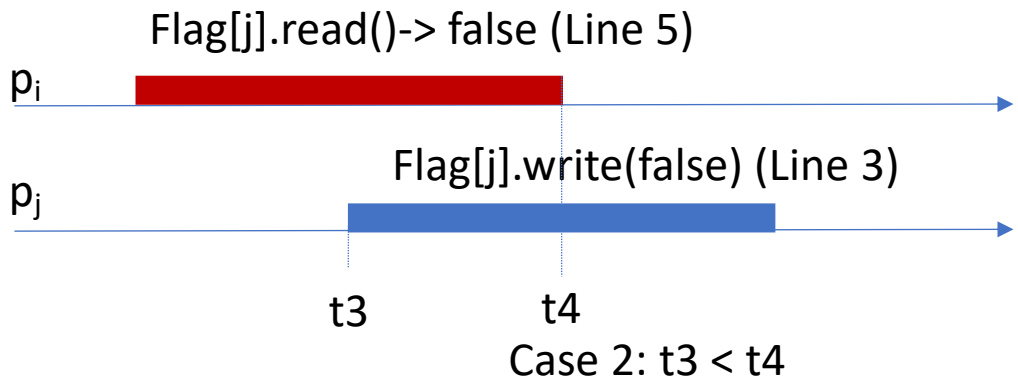
Lemma 2: Let p_i and p_j be such that p_i is inside the CS while p_j is in the bakery (L3-7). Then
(**Number[i],i)** < (**Number[j],j)**

Proof

- Notice that since p_j is in the bakery it can be in the CS
- As p_i is in the CS it read **Flag[j]** = false (L5)
- According to the time of that read and the time at which p_j wrote **Flag[j]** (L1 or L3) we have
  - Either t1 < t2
  - Or t3 < t4



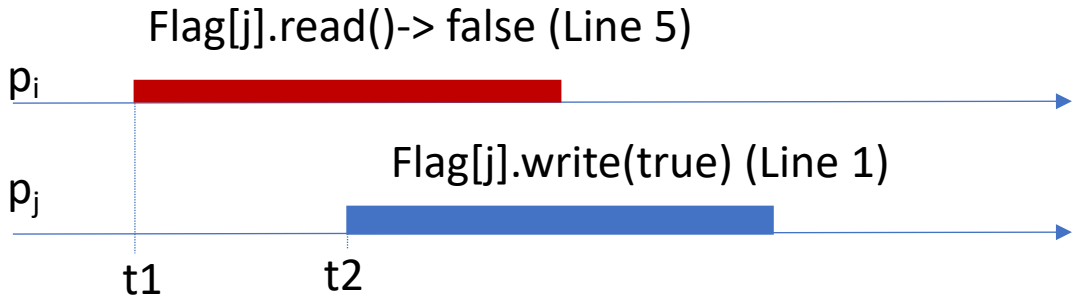
Case 1: t1 < t2

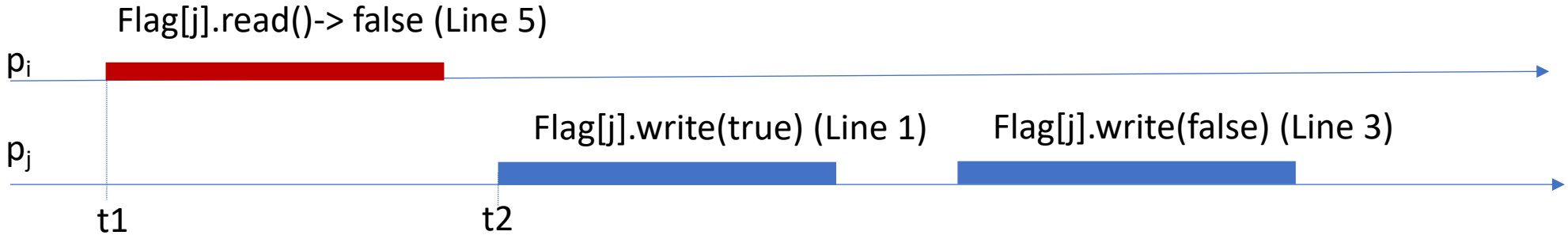


Case 2: t3 < t4

Bakery Algorithm

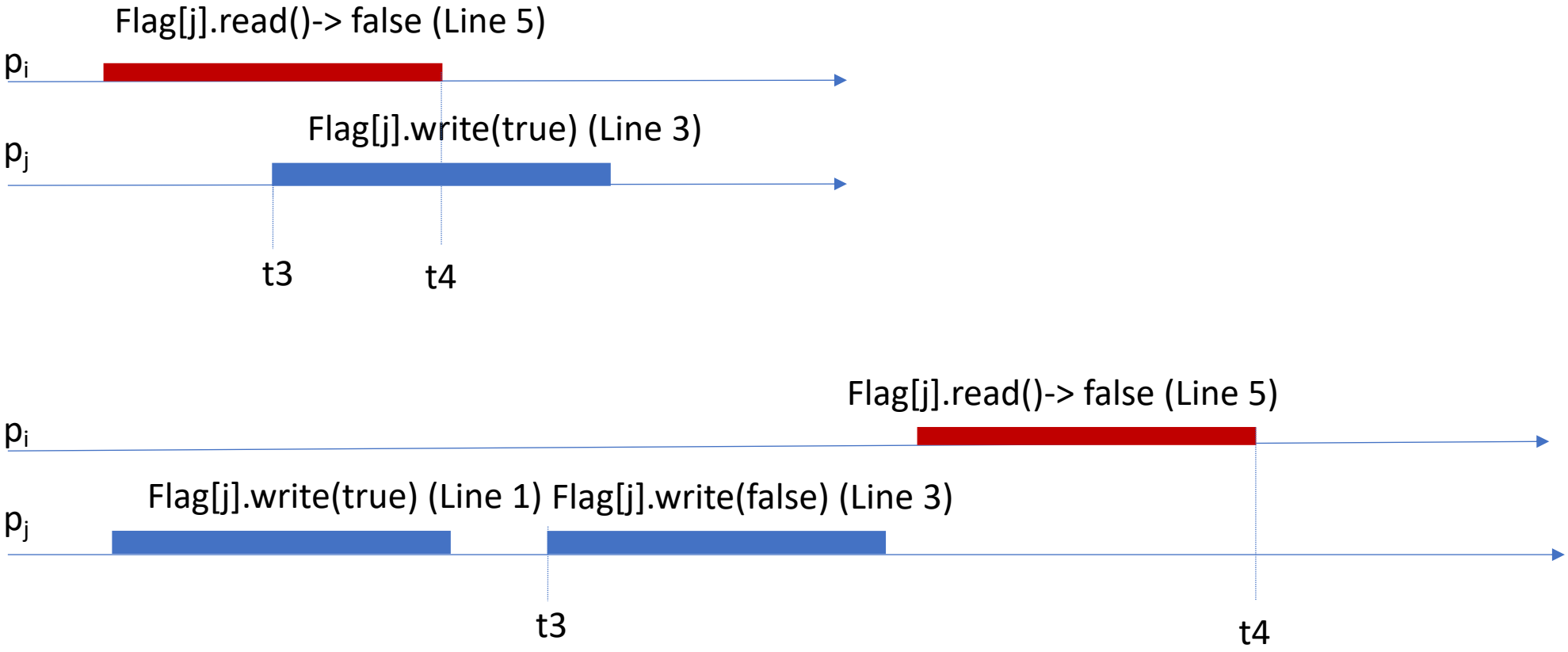
Case 1: t1 < t2





Bakery Algorithm

Case 1: t3 < t4



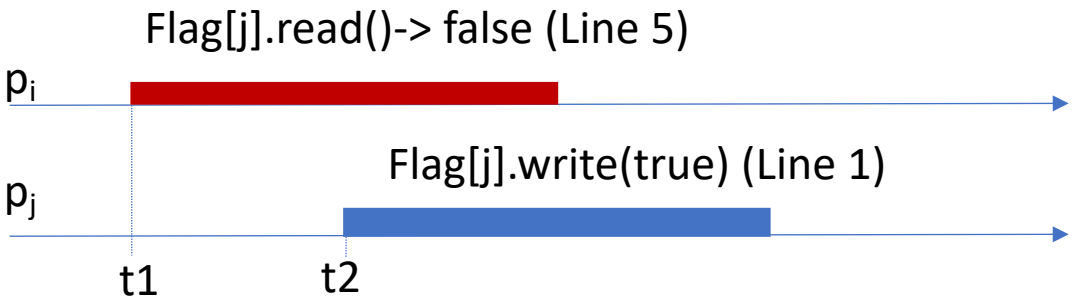
Bakery Algorithm

Algorithm: Bakery (code executed by processor pi, $1 \leq i \leq n$)
Initially **Flag[i]**=false and **Number[i]**=0 for $1 \leq i \leq n$

<Entry>:
1: **Flag[i]** = true
2: **Number[i]** = max(**Number[1]** ,.., **Number[n]**)+1
3: **Flag[i]** = false
4: for j=1 to n ($j \neq i$) do
5: wait until $\mathbf{Flag}[j]$=false
6: wait until **Number[j]=0 or** (**Number[j]**,j) > (**Number[i]**,i))
<Critical Section>
<Exit>:
7: **Number[i]** = 0

Lemma 2: Let p_i and p_j be such that p_i is inside the CS while p_j is in the bakery. Then
(Number[i],i) < (Number[j],j)
Proof (continue)

- Case t1 < t2 :
- p_i entered the bakery before p_j enters the doorway
- Thus from Lemma 1, **Number[i]<Number[j]**
- Which ends this case

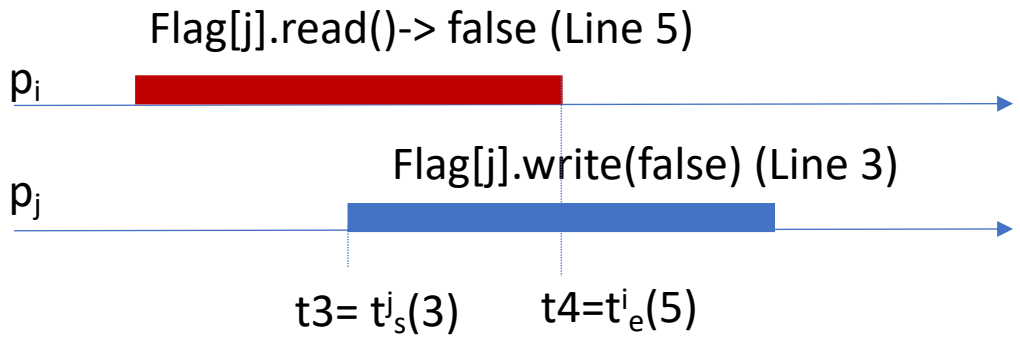


Case 1: t1 < t2

Bakery Algorithm

Algorithm: Bakery (code executed by processor pi, $1 \leq i \leq n$
Initially **Flag[i]**=false and **Number[i]**=0 for $1 \leq i \leq n$

<Entry>:
1: **Flag[i]** = true
2: **Number[i]** = max(**Number[1]** ,.., **Number[n]**)+1
3: **Flag[i]** = false
4: for j=1 to n ($j \neq i$) do
5: wait until $\mathbf{Flag}[j]$=false
6: wait until **Number[j]=0 or** (**Number[j]**,j) > (**Number[i]**,i))
<Critical Section>
<Exit>:
7: **Number[i]** = 0



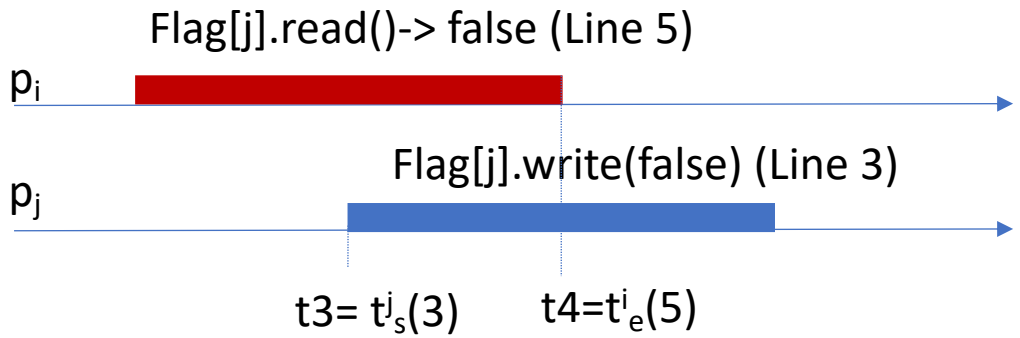
Lemma 2: Let p_i and p_j be such that p_i is inside the CS while p_j is in the bakery. Then
(**Number[i]**,i) < (**Number[j]**,j)
Proof (continue)

- Case t3 < t4 (i.e., $t^j_s(3) < t^i_e(5)$) (H1)
- p_j is sequential thus p_j ended L2 < t3 (i.e $t^j_e(2) < t^j_s(3)$) (P1)
- p_i is sequential thus $t^i_s(5) < t^i_s(6)$ (P2)
- (P1)+(P2) and (H1) :
  $t^j_e(2) < t^j_s(3)$ and $t^j_s(3) < t^i_e(5)$ and $t^i_s(5) < t^i_s(6)$
  $t^j_e(2) < t^j_s(3) < t^i_e(5) < t^i_s(6)$
  $t^j_e(2) < t^i_s(6)$ (P3)

Bakery Algorithm

Algorithm: Bakery (code executed by processor pi, $1 \leq i \leq n$
Initially **Flag[i]**=false and **Number[i]**=0 for $1 \leq i \leq n$

<Entry>:
1: **Flag[i]** = true
2: **Number[i]** = max(**Number[1]** ,.., **Number[n]**)+1
3: **Flag[i]** = false
4: for j=1 to n ($j \neq i$) do
5: wait until $\mathbf{Flag}[j]$=false
6: wait until **Number[j]=0 or** (**Number[j]**,j) > (**Number[i]**,i))
<Critical Section>
<Exit>:
7: **Number[i]** = 0



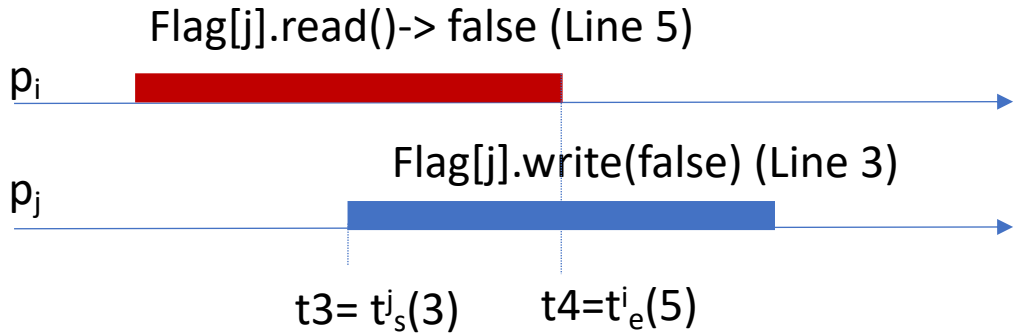
Lemma 2: Let p_i and p_j be such that p_i is inside the CS while p_j is in the bakery. Then
(**Number[i]**,i) < (**Number[j]**,j)
Proof (continue)

- Case t3 < t4 (i.e., $t^j_s(3) < t^i_e(5)$) (H1)
- p_j is sequential thus p_j ended L2 < t3 (i.e $t^j_e(2) < t^j_s(3)$) (P1)
- p_i is sequential thus $t^i_s(5) < t^i_s(6)$ (P2)
- (P1)+(P2) and (H1) :
  $t^j_e(2) < t^j_s(3)$ and $t^j_s(3) < t^i_e(5)$ and $t^i_s(5) < t^i_s(6)$
  $t^j_e(2) < t^j_s(3) < t^i_e(5) < t^i_s(6)$
  $t^j_e(2) < t^i_s(6)$ (P3)
- Thus the read by p_i of Number[j] in Line 6 occurred after Number[j] was written by p_j on Line2

Bakery Algorithm

Algorithm: Bakery (code executed by processor pi, $1 \leq i \leq n$
Initially **Flag[i]**=false and **Number[i]**=0 for $1 \leq i \leq n$

<Entry>:
1: **Flag[i]** = true
2: **Number[i]** = max(**Number[1]** ,.., **Number[n]**)+1
3: **Flag[i]** = false
4: for j=1 to n ($j \neq i$) *do*
5: wait until $\mathbf{Flag}[j]$=false
6: wait until **Number[j]=0 or** (**Number[j]**,j) > (**Number[i]**,i))
<Critical Section>
<Exit>:
7: **Number[i]** = 0



Lemma 2: Let p_i and p_j be such that p_i is inside the CS while p_j is in the bakery. Then
(**Number[i]**,i) < (**Number[j]**,j)
Proof (continue)

- Case t3 < t4 (i.e., $t^j_s(3) < t^i_e(5)$) (H1)
- p_j is sequential thus p_j ended L2 < t3 (i.e $t^j_e(2) < t^j_s(3)$) (P1)
- p_i is sequential thus $t^i_s(5) < t^i_s(6)$ (P2)
- (P1)+(P2) and (H1) :
  $t^j_e(2) < t^j_s(3)$ and $t^j_s(3) < t^i_e(5)$ and $t^i_s(5) < t^i_s(6)$
  $t^j_e(2) < t^j_s(3) < t^i_e(5) < t^i_s(6)$
  $t^j_e(2) < t^i_s(6)$ (P3)
- Thus the read by p_i of Number[j] in Line 6 occurred after Number[j] was written by p_j on Line2
- As p_i is in the CS it exited the 2nd wait statement, i.e., **Number[j]=0 or** (**Number[j]**,j) > (**Number[i]**,i))
- As $t^j_e(2) < t^i_s(6)$ (P3) we have **Number[j]≠0**
- Thus **Number[j]**,j) > (**Number[i]**,i))
- Which ends this case

Bakery Algorithm

Algorithm: Bakery (code executed by processor pi, $1 \leq i \leq n$
Initially **Flag[i]**=false and **Number[i]**=0 for $1 \leq i \leq n$

<Entry>:
1: **Flag[i]** = true
2: **Number[i]** = max(**Number[1]** ,.., **Number[n]**)+1
3: **Flag[i]** = false
4: for j=1 to n (j $\neq i$) do
5: wait until $\mathbf{Flag}[j]$=false
6: wait until **Number[j]=0 or** (**Number[j]**,j) > (**Number[i]**,i))
<Critical Section>
<Exit>:
7: **Number[i]** = 0
<Remainder>:

Th: The Bakery algorithm satisfies mutual exclusion

Proof
By contradiction

- Suppose that both p_i and p_j are both in the CS
- As p_i is in the CS and p_j is in the bakery, by Lemma 2, we have (**Number[i]**,i) < (**Number[j]**,j)
- Similarly, as p_j is in the CS and p_i is in the bakery, by Lemma 2, we have (**Number[j]**,j) < (**Number[i]**,i)
- As (j $\neq i$) both pairs are totally ordered. It follows that each pair contradicts the other, from which the mutual exclusion property holds

Bakery Algorithm

Algorithm: Bakery (code executed by processor pi, $1 \leq i \leq n$
Initially **Flag[i]**=false and **Number[i]**=0 for $1 \leq i \leq n$

<Entry>:
1: **Flag[i]** = true
2: **Number[i]** = max(**Number[1]** ,.., **Number[n]**)+1
3: **Flag[i]** = false
4: for j=1 to n (j $\neq i$) do
5: wait until $\mathbf{Flag}[j]$=false
6: wait until **Number[j]=0 or** (**Number[j]**,j) > (**Number[i]**,i))
<Critical Section>
<Exit>:
7: **Number[i]** = 0
<Remainder>:

Th: The Bakery algorithm is lock-free

Proof

By contradiction

- Let (**Number[i]**,i) be the smallest pair of a proc that is starved before the CS
- All the processes that will enter the entry section after p_i will get a ticket greater than p_i's one
- Thus they will not enter the CS before p_i
- All the processes that have a smaller ticket than p_i will enter the CS (by assumption they are not starved)
- They will exit it (since no process stays in the CS forever)
- So p_i will pass the for loop and enter the CS, a contradiction

Bibliography

- Specification part comes from
  - Rachid Guerraoui and Petr Kuztnetsov's book, "Algorithms for Concurrent Systems", Eyrolles
- Mutual exclusion part comes from
  - Hagit Attiya and Jennifer Welch's book, "Distributed computing, Fundamentals, simulations, and advanced topics", Wiley series.
  - Michel Raynal's book, Concurrent programming: Algorithms, Principles, and Foundations, Springer
- Constructions part comes from
  - On interprocess communication, Part I and II. Distributed computing. Vol 1, Number 2, pages 77 – 101.
  - Michel Raynal's book, Concurrent programming: Algorithms, Principles, and Foundations, Springer