Principles of Cyber-Physical Systems

Chapter 8-1: Asynchronous Model

Instructor: Lanshun Nie

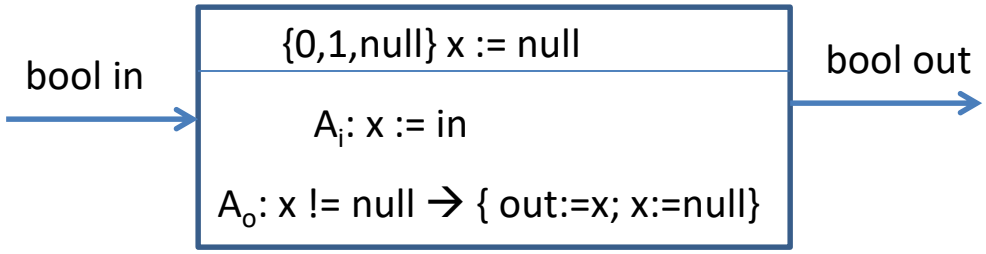
Asynchronous Models

- Recap: In a synchronous model, all components execute in a sequence of (logical) rounds in lock-step

- Asynchronous: Speeds at which different components execute are independent (or unknown)
  - Processes in a distributed system
  - Threads in a typical operating system such as Linux/Windows

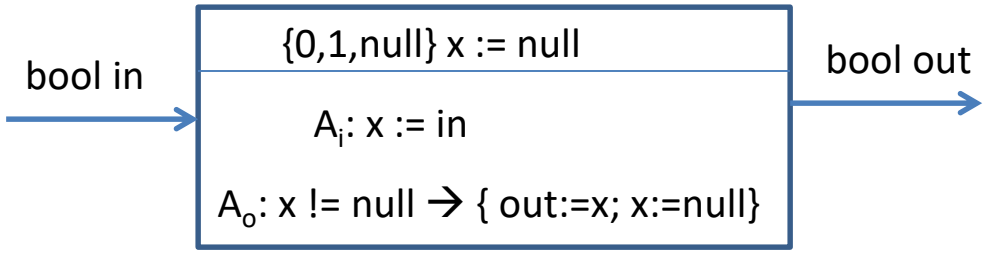
- Key design challenge: how to achieve coordination?

Example: Asynchronous Buffer

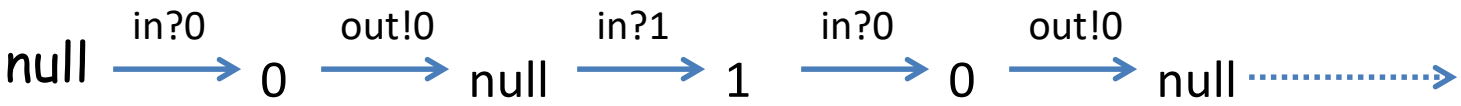


- Input channel: in of type Boolean
- Output channel: out of type Boolean
- State variable: x; can be empty/null, or hold 0/1 value
- Initialization of state variables: assignment x:=null
- Input task A_i for processing of inputs: code: x:= in
- Output task A_o for producing outputs:
  Guard: x != null; code: out:=x; x:=null

Example: Asynchronous Buffer

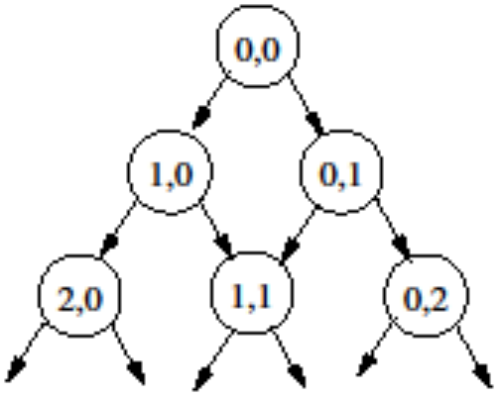


- Execution Model: In one step, only a single task is executed
  - Processing of inputs (by input tasks) is decoupled from production of outputs (by output tasks)
- A task can be executed if it is enabled, i.e., its guard condition holds
  - If multiple tasks enabled, one of them is executed
- Sample Execution:



Example: Asynchronous Increments

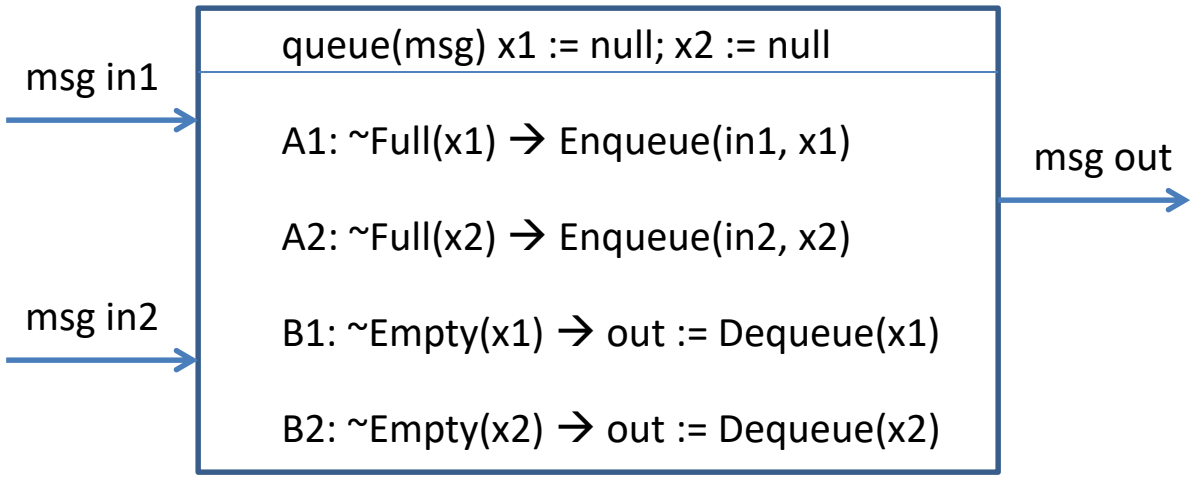
nat x:=0; y:=0
A_x: x := x+1
A_y: y :=y+1



- Internal task: Does not involve input or output channels
  - Can have guard condition and update code
  - Execution of internal task: Internal action
- In each step, execute, either task A_x or task A_y
- Sample Execution:

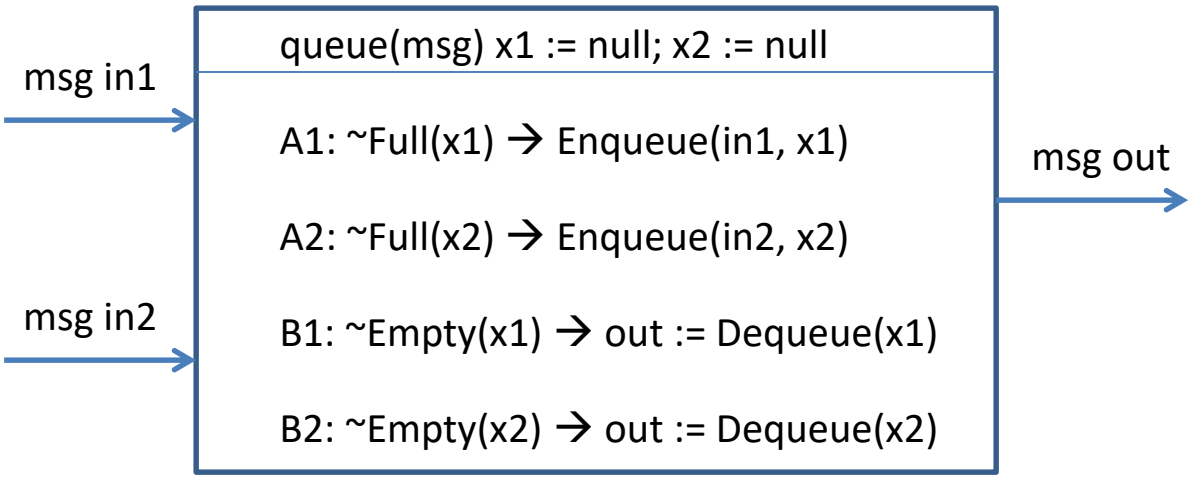
  (0,0) → (1,0) → (1,1) → (1,2) → (1,3) → … → (1,105) → (2, 105) …
- For every m, n, state (x=m, y=n) is reachable
  - Interleaving model of concurrency

Asynchronous Merge



Sequence of messages on output channel is an arbitrary merge of sequences of values on the two input channels

Asynchronous Merge

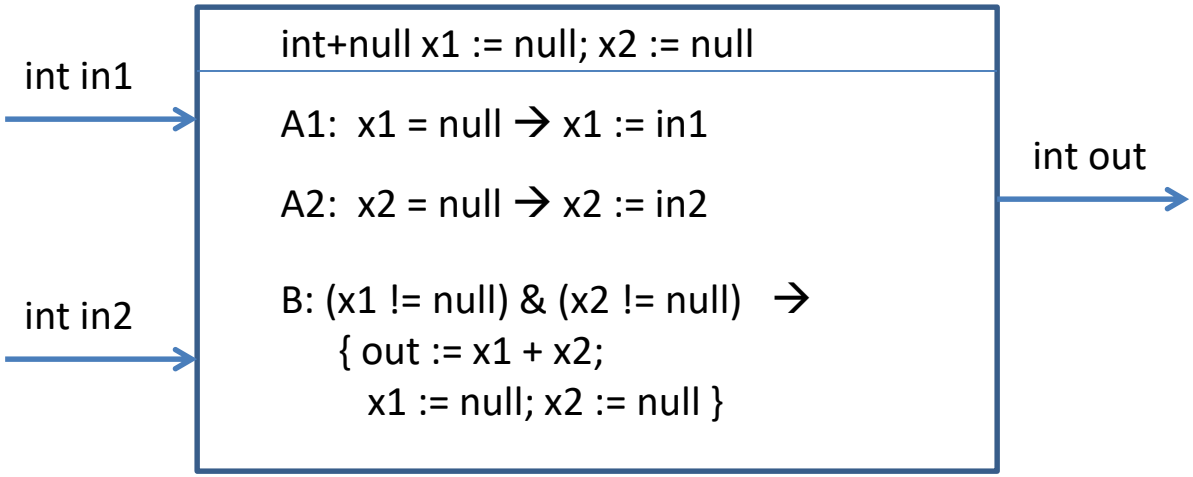


At every step exactly one of the four tasks executes, provided its guard condition holds

Sample Execution:

([],[]) – in1?5 -> ([5], []) – in2?0 -> ([5],[0]) – out!0 -> ([5],[])
– in1?6 -> ([5,6],[]) – in2?3 -> ([5,6],[3]) – out!5 -> ([6],[3]) …

What does this process do?



Definition: Asynchronous Process P

- Set I of (typed) input channels
  - Defines the set of inputs of the form x?v, where x is an input channel and v is a value
- Set O of (typed) output channels
  - Defines the set of outputs of the form y!v, where y is an output channel and v is a value
- Set S of (typed) state variables
  - Defines the set of states Q_S
- Initialization Init
  - Defines the set [Init] of initial states

Definition (contd): Asynchronous Process P

- Set of input tasks; each such task is associated with an input channel x
  - Guard condition over state variables S
  - Update code from read-set S U {x} to write-set S
  - Defines a set of input actions of the form s – x?v -> t

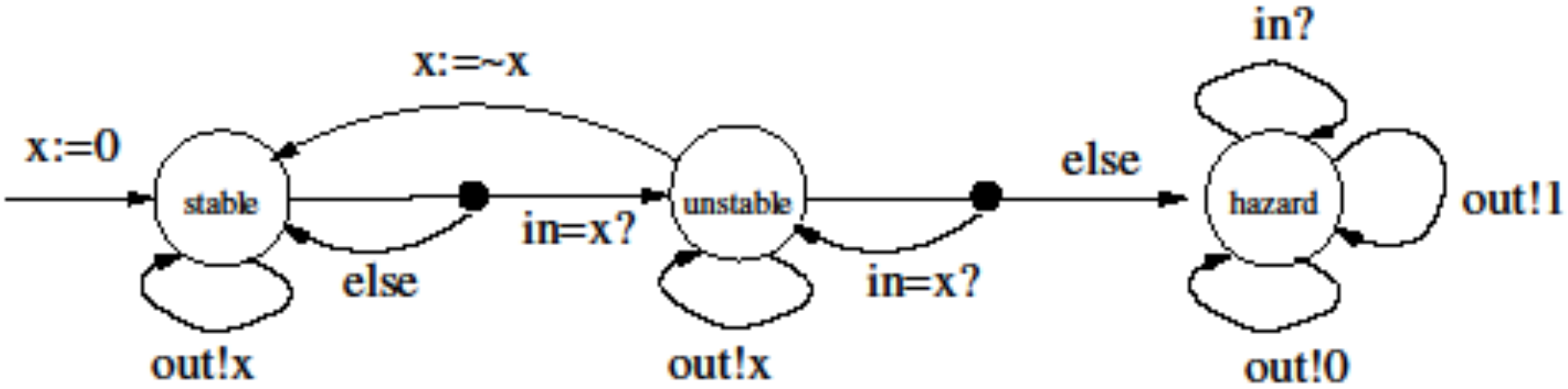
- Set of output tasks; each task is associated with an output channel y
  - Guard condition over state variables S
  - Update code from read-set S to write-set S U {y}
  - Defines a set of output actions of the form s – y!v -> t

- Set of internal tasks
  - Guard condition over state variables S
  - Update code from read-set S to write-set S
  - Defines a set of internal actions of the form s – ε -> t

Asynchronous Gates

- Why design asynchronous circuits?
  - Input can be changed even before the effect propagates through the entire circuit
  - Can be faster than synchronous circuits, but design more complex
- Modeling a NOT gate
  - When input changes, gate enters *unstable* state till it gets a chance to update output value
  - If input changes again in unstable state, then this causes a *hazard* where behavior is unpredictable

Asynchronous NOT Gate as an ESM



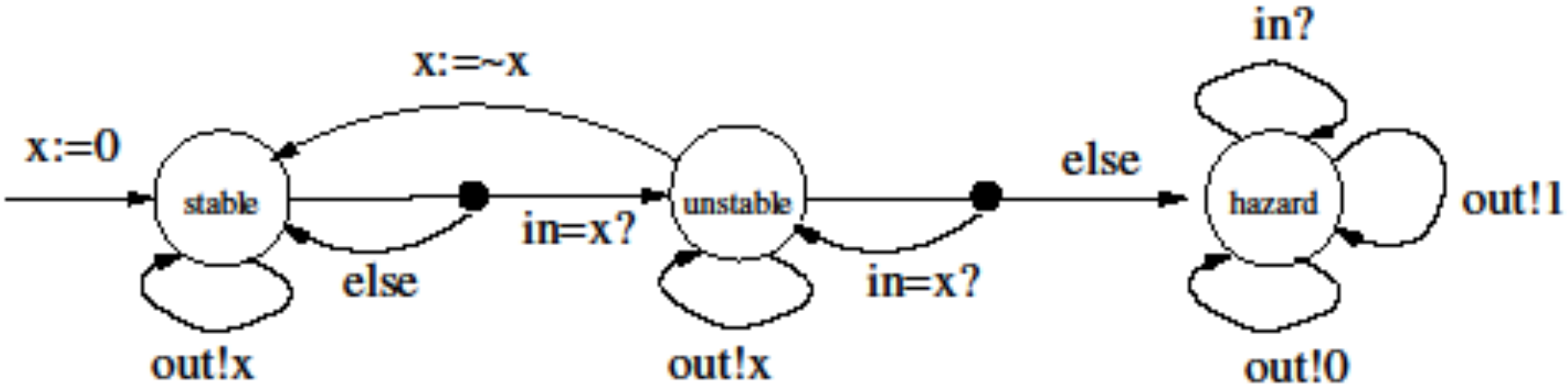
Sample Execution:

(stable,0) – out!0 -> (stable,0) – in?0 -> (unstable,0) –ε-> (stable, 1) – out!1 -> (stable,1) – in?1 -> (unstable,1) –out!1 -> (unstable,1) – in?0 -> (hazard,1) – out!0 -> (hazard,1) – out!1 -> (hazard,1) …

How to ensure that the gate does not enter hazard state?

When the input toggles, wait to observe a change in value of output!

Executing an ESM



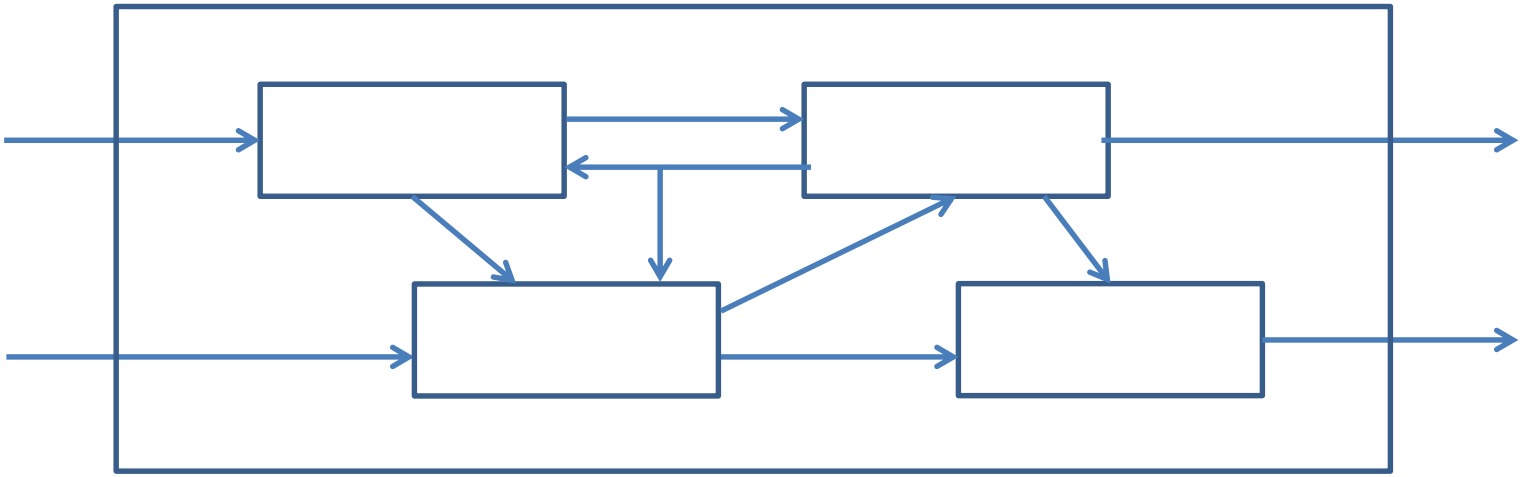
Each mode-switch corresponds to a task

Example input task: (mode = stable) → if (in=x) then mode := unstable

Example output task: (mode = stable) → out := x

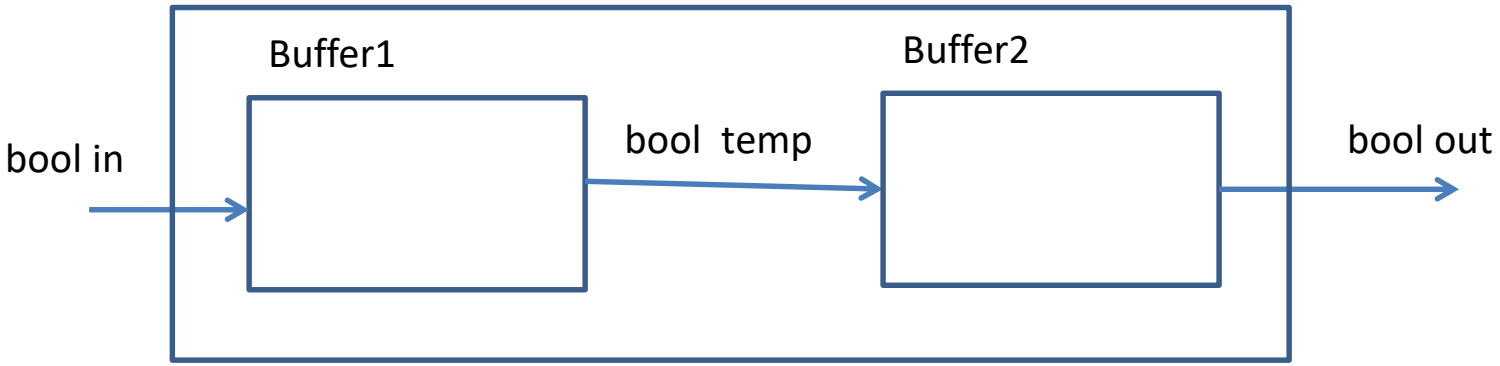
Example internal task: (mode = unstable) → { x:=~x; mode := stable }

Block Diagrams



- Visually the same as the synchronous case
  - Execution semantics different !

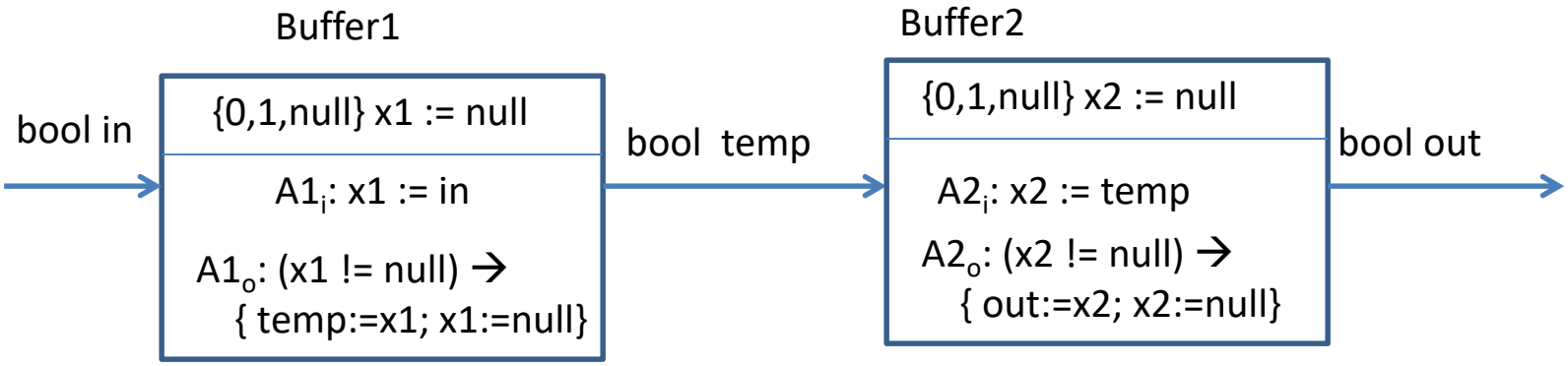
DoubleBuffer



(Buffer[out -> temp] | Buffer[in -> temp]) \ temp

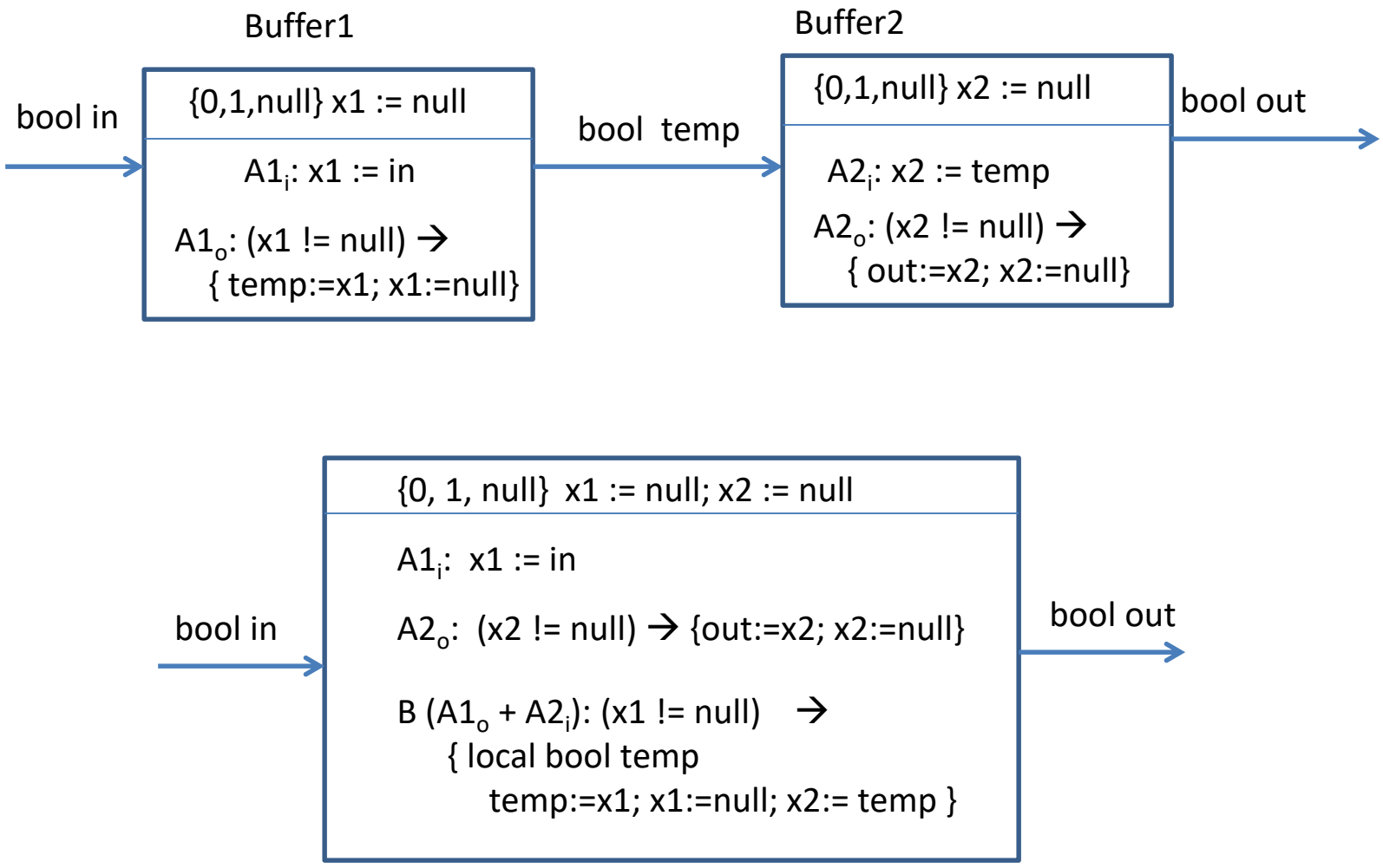
- Instantiation: Create two instances of Buffer
  - Output of Buffer1 = Input of Buffer2 = Variable temp
- Parallel composition: Asynchronous concurrent execution of Buffer1 and Buffer2
- Hide variable temp: Encapsulation (temp becomes local)

Composing Buffer1 and Buffer2



- Inputs, outputs, states, and initialization for composition obtained in the same manner as in synchronous case
- What are the tasks of the composition?
  - Production of output on temp by Buffer1 synchronized with consumption of input on temp by Buffer2

Compiled DoubleBuffer



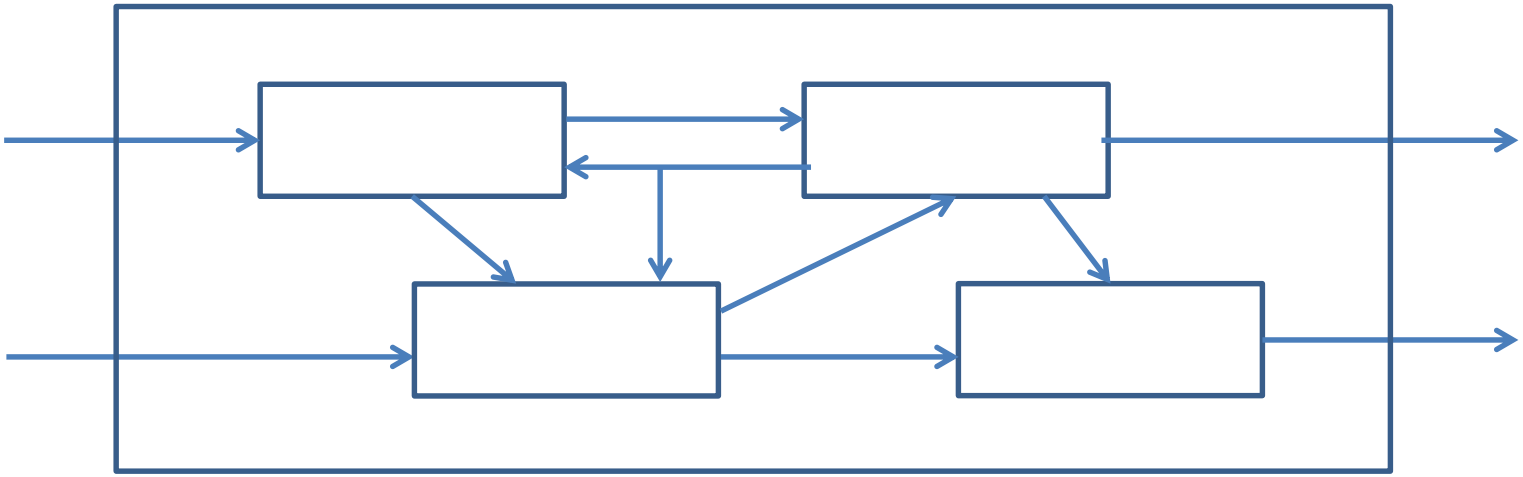
Definition of Asynchronous Composition

- Given asynchronous processes P1 and P2, how to define P1 | P2 ?

- Note: In each step of execution, only one task is executed
  - Concepts such as await-dependencies, compatibility of interfaces, are not relevant

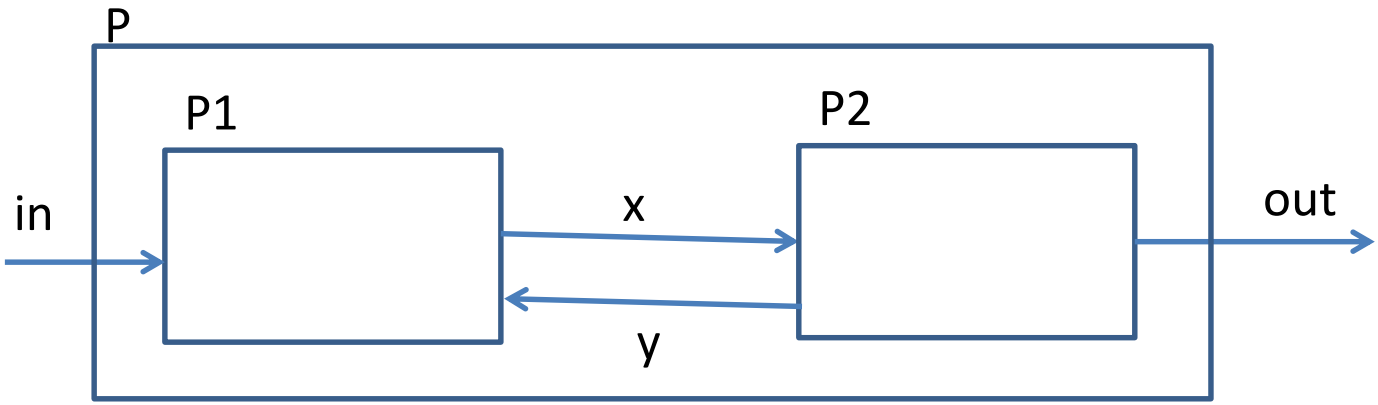
- Sample case (see textbook for complete definition):
  - if y is an output channel of P1 and input channel of P2, and
  - A1 is an output task of P1 for y with code: Guard1 → Update1
  - A2 is an input task of P2 for y with code: Guard2 → Update2, then
  - Composition has an output task for y with code:
    (Guard1 & Guard2) → Update1 ; Update2

Execution Model: Another View



- A single step of execution
  - Execute an internal task of one of the processes
  - Process input on an external channel x: Execute an input task for x of every process to which x is an input
  - Execute an output task for an output y of some process, followed by an input task for y for every process to which y is an input
- If multiple enabled choices, choose one non-deterministically
  - No constraint on relative execution speeds

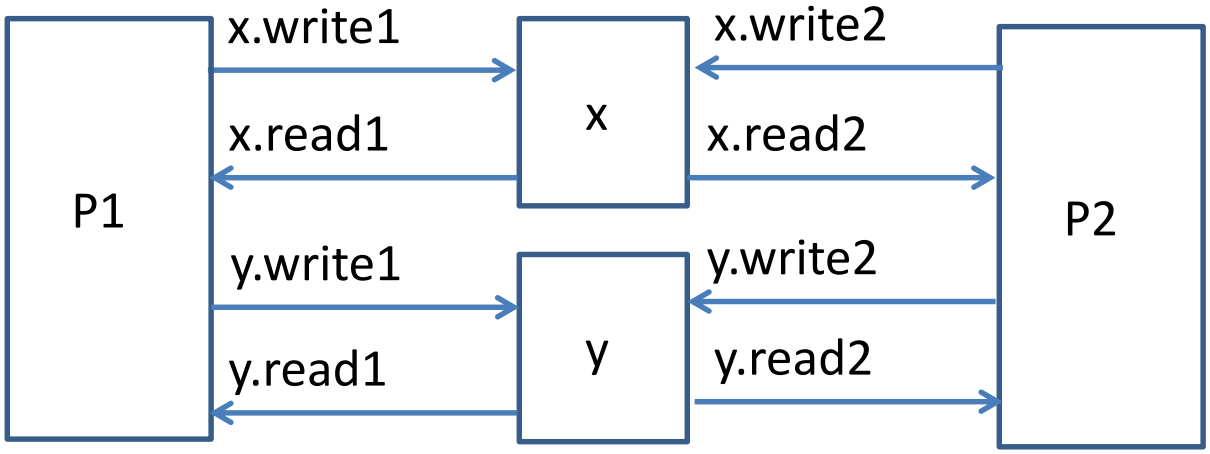
Asynchronous Execution



What can happen in a single step of this asynchronous model P?

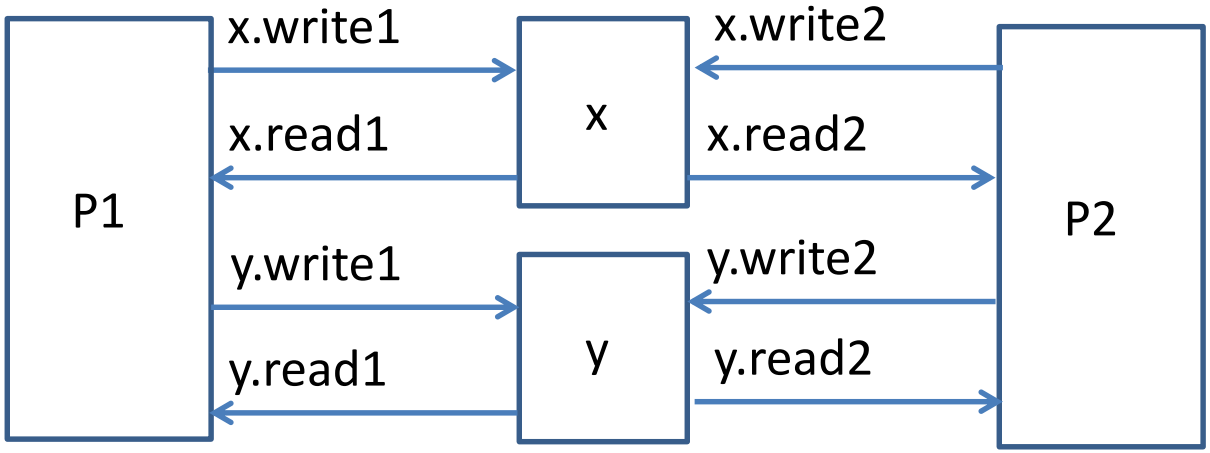
- P1 synchronizes with the environment to accept input on in
- P2 synchronizes with the environment to send output on out
- P1 performs some internal computation (one of its internal tasks)
- P2 performs some internal computation (one of its internal tasks)
- P1 produces output on channel x, followed by its consumption by P2
- P2 produces output on channel y, followed by its consumption by P1

Shared Memory Processes



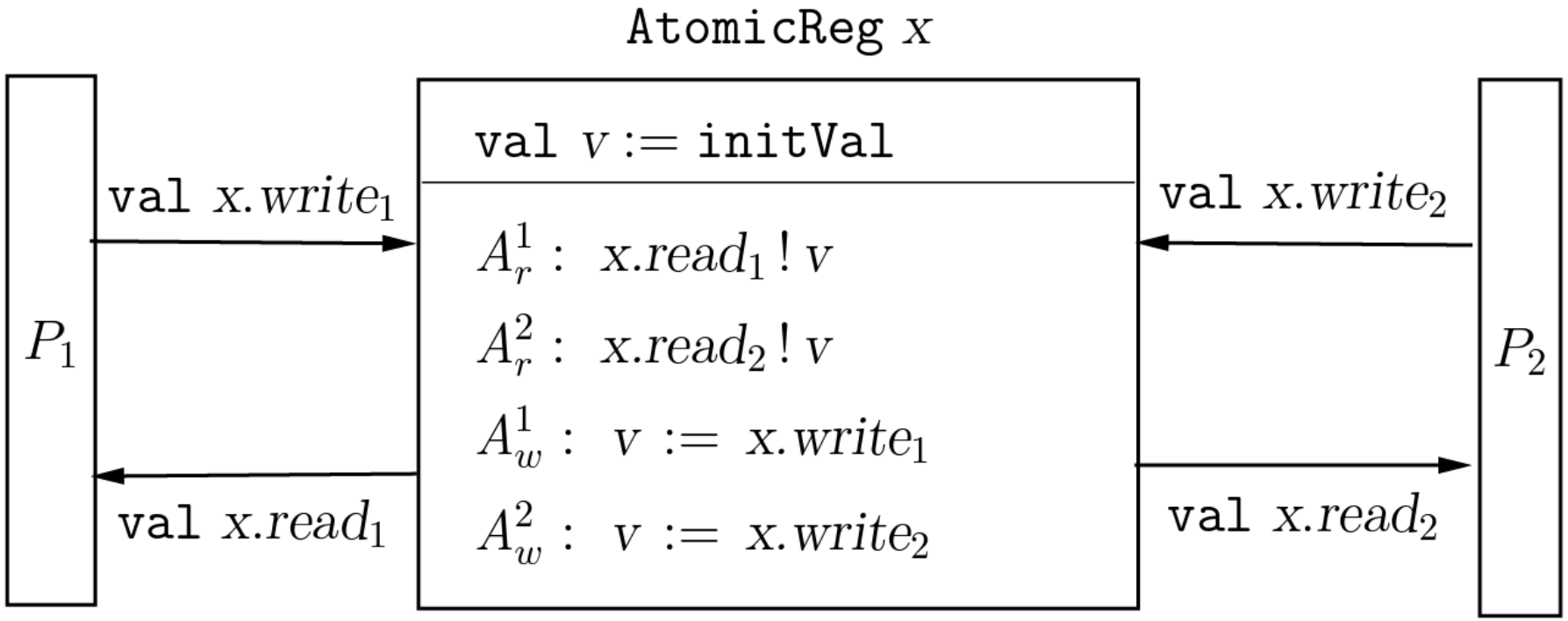
- Processes P1 and P2 communicate by reading/writing shared variables
- Each shared variable can be modeled as an asynchronous process
  - State of each such process is the value of corresponding variable
  - In implementation, shared memory can be a separate subsystem
- Read and write channel between each process and each shared variable
  - To write x, P1 synchronizes with x on "x.write1" channel
  - To read y, P2 synchronizes with y on "y.read2" channel

Atomic Registers

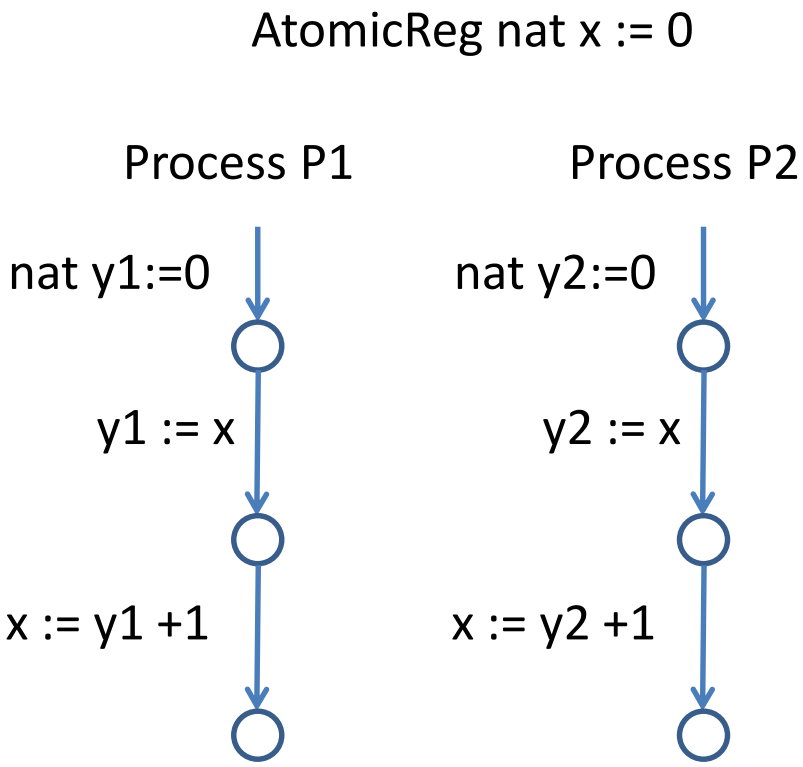


- By definition of our asynchronous model, each step of above is either internal to P1 or P2, or involves exactly one synchronization: either read or write of one shared variable by one of the processes
- Atomic register: Basic primitives are read and write
  - To "increment" such a register, a process first needs to read and then write back incremented value
  - But these two are separate steps, and register value can be changed in between by another process

Atomic Registers



Shared Memory Programs



Declaration of shared variables
+ Code for each process

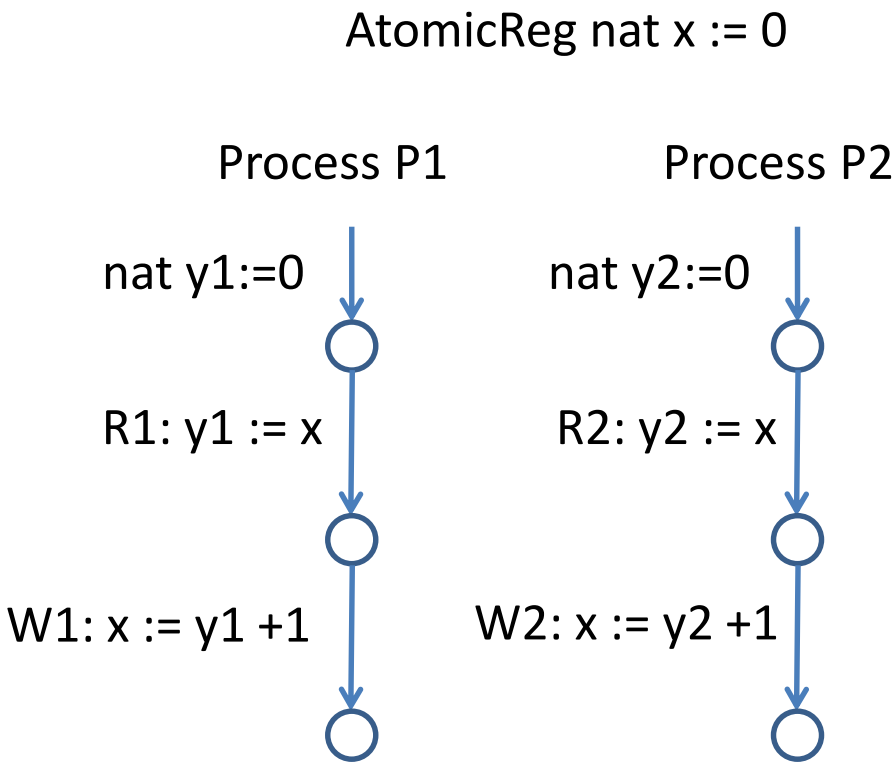
Key restriction: Each statement of a process either

- changes local variables,
- reads a single shared var, or
- writes a single shared var

Execution model: execute one step of one of the processes

Can be formalized as asynchronous processes

Data Races



What are the possible values of x after all steps are executed?

x can be 1 or 2

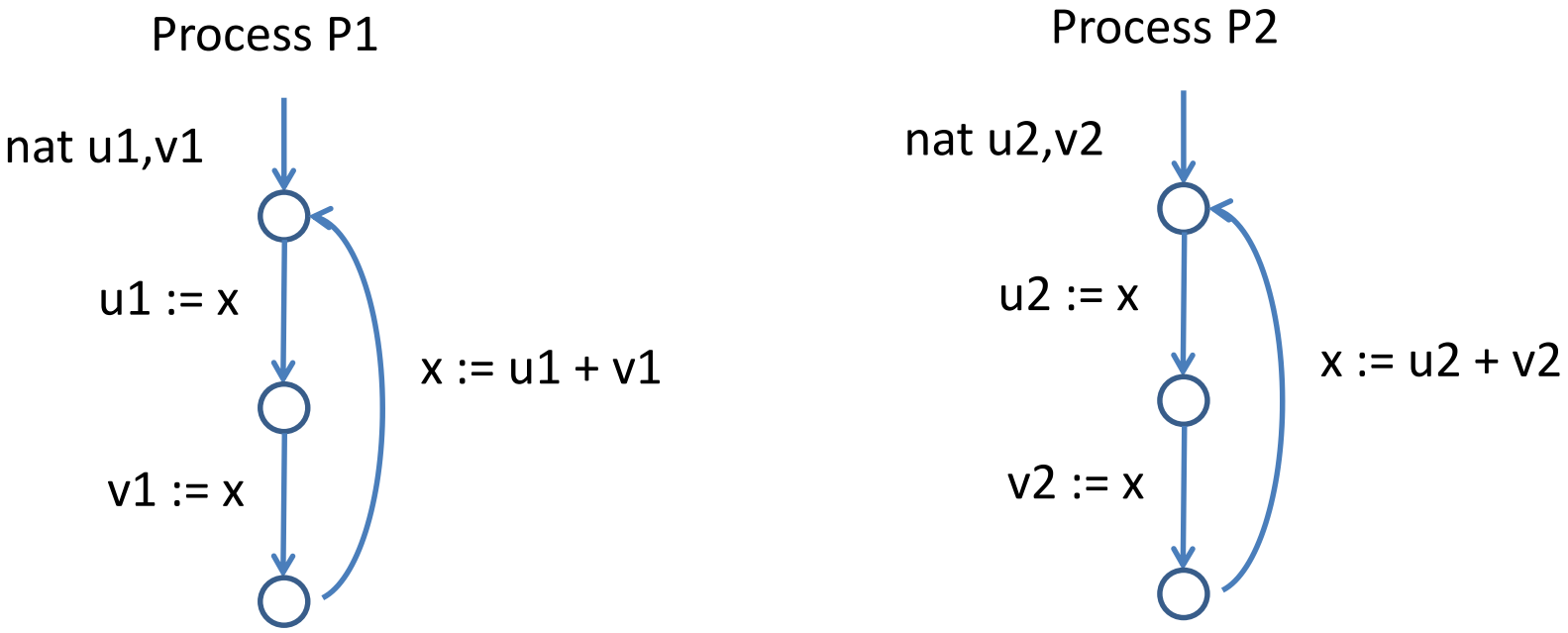
Possible executions:

R1, R2, W1, W2
R1, W1, R2, W2
R1, R2, W2, W1
R2, R1, W1, W2,
R2, W2, R1, W1
R2, R1, W2, W1

Data race: Concurrent accesses to shared object where the result depends on order of execution, Should be avoided!!

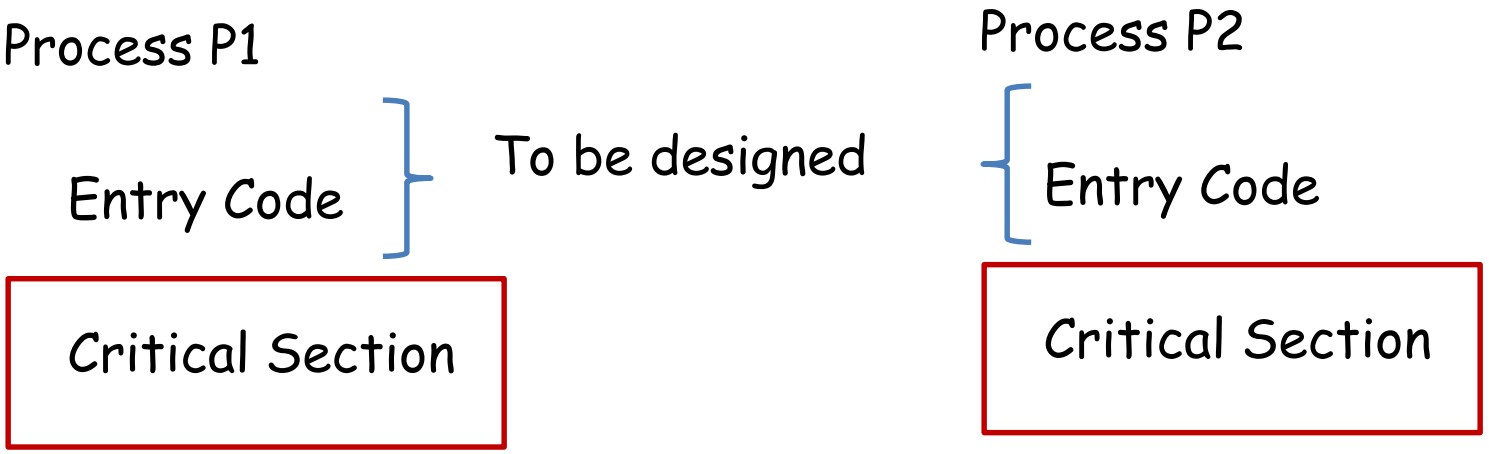
Puzzle

AtomicReg nat x := 1



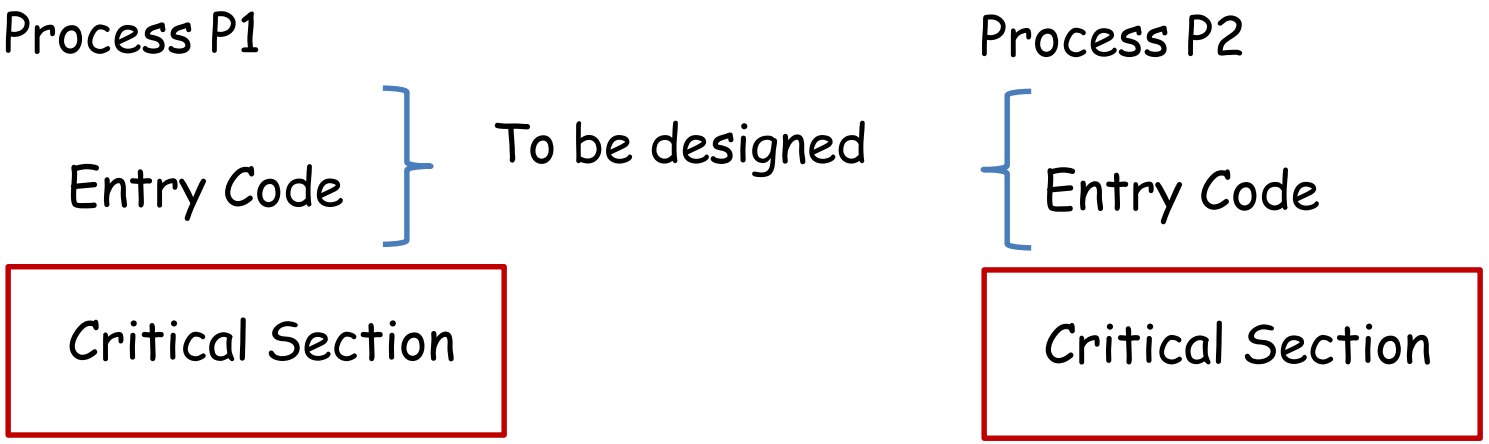
What possible values can the shared register x take?

Mutual Exclusion Problem



- Critical Section: Part of code that an asynchronous process should execute without interference from others
  - Critical section can include code to update shared objects/database
- Mutual Exclusion Problem: Design code to be executed before entering critical section by each process
  - Coordination using shared atomic registers
  - No assumption about how long a process stays in critical section
  - A process may want to enter critical section repeatedly

Mutual Exclusion Problem



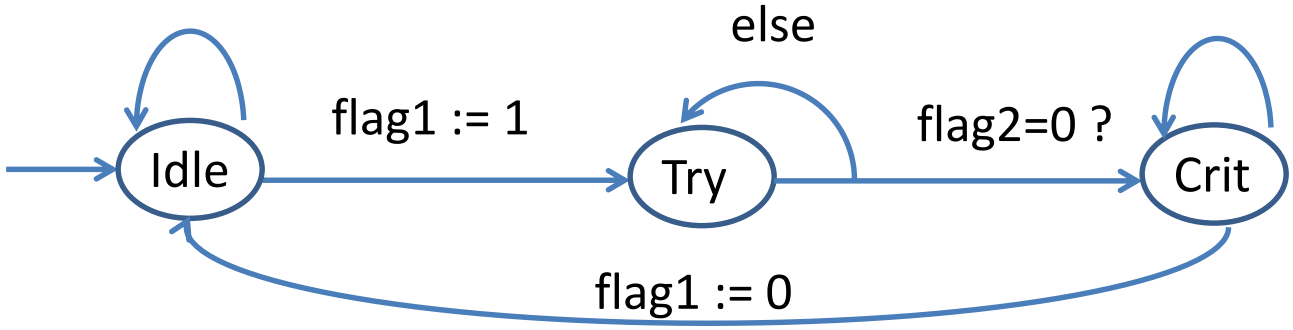
- Safety Requirement: Both processes should not be in critical section simultaneously (can be formalized using invariants)

- Absence of deadlocks: If a process is trying to enter, then some process should be able to enter

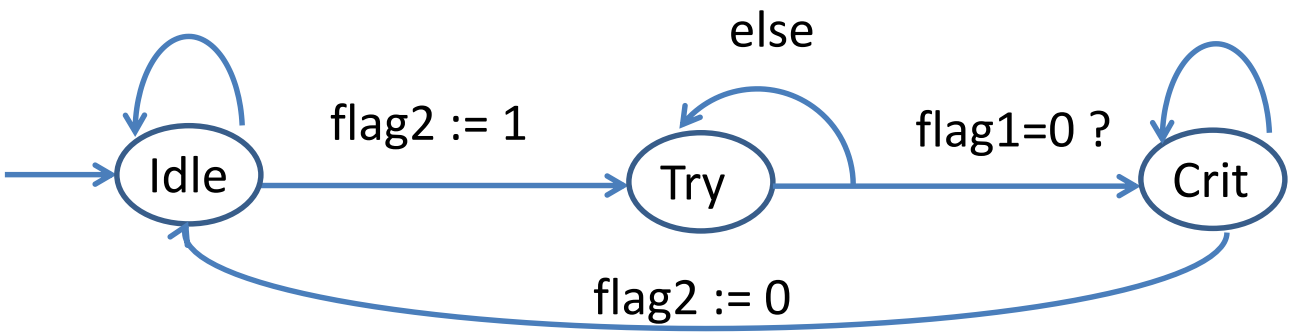
Mutual Exclusion: First Attempt

AtomicReg bool flag1 := 0; flag2 := 0

Process P1



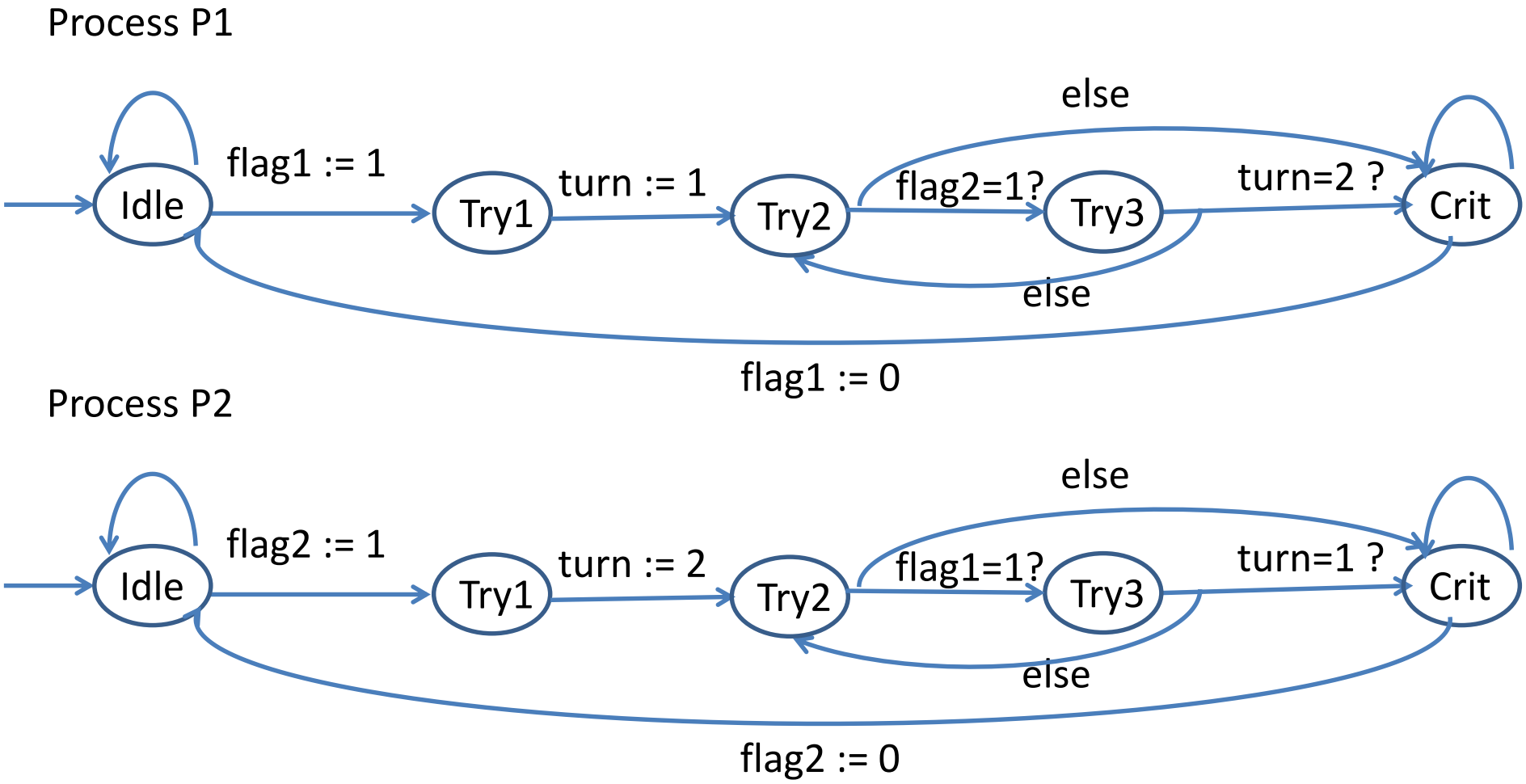
Process P2



Is this correct?

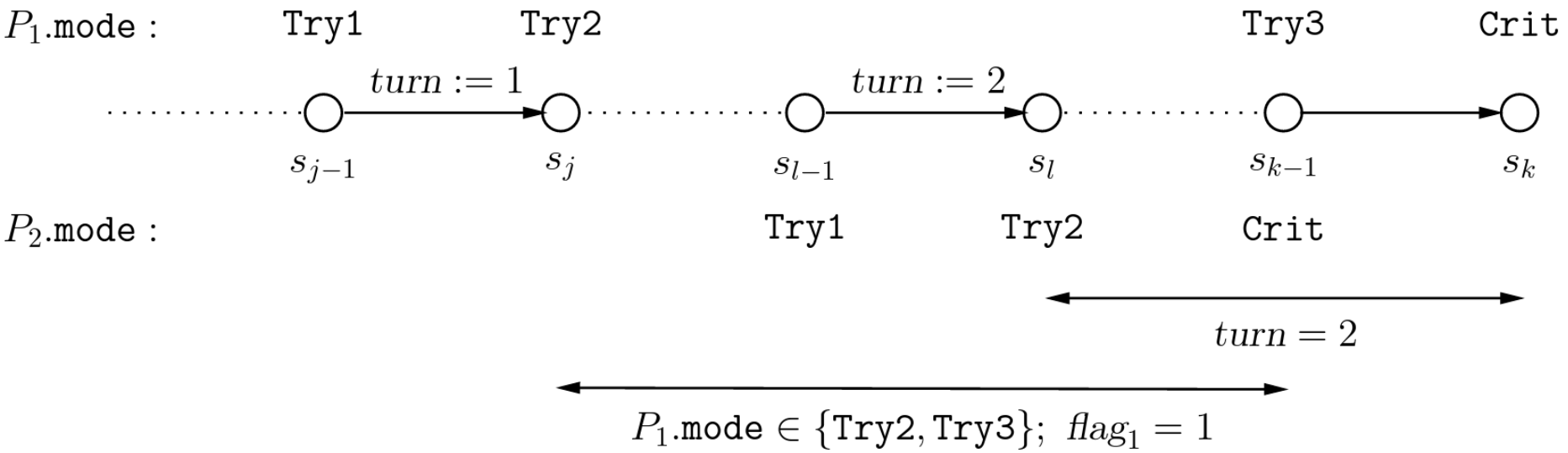
Peterson's Mutual Exclusion Protocol

AtomicReg bool flag1 := 0; flag2 := 0; {1,2} turn



Peterson's Mutual Exclusion Protocol

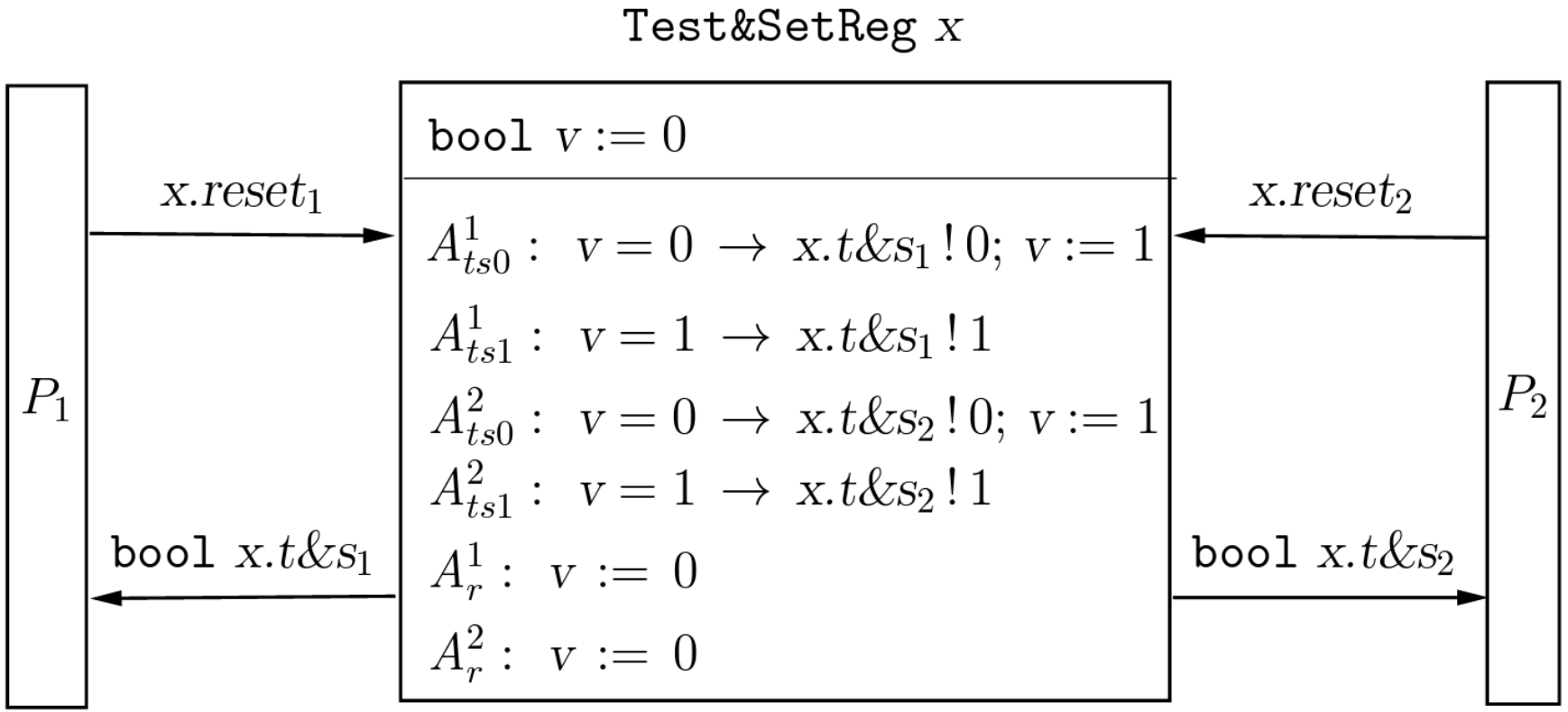
PROOF



Test&Set Register

- Beyond atomic registers:
  - In one (atomic) step, can do more than just read or write
  - Stronger synchronization primitives
- Test&Set Register: Holds a Booleans value
  - Reset operation: Changes the value to 0
  - Test&Set operation: Returns the old value and changes value to 1
  - If two processes are competing to execute Test&Set on a register with value 0, one will get back 0 and other will get back 1
- Modern processors support strong "atomic" operations
  - Compare-and-swap
  - Load-linked-store-conditional
  - Implementation is expensive (compared to read/write operations)!

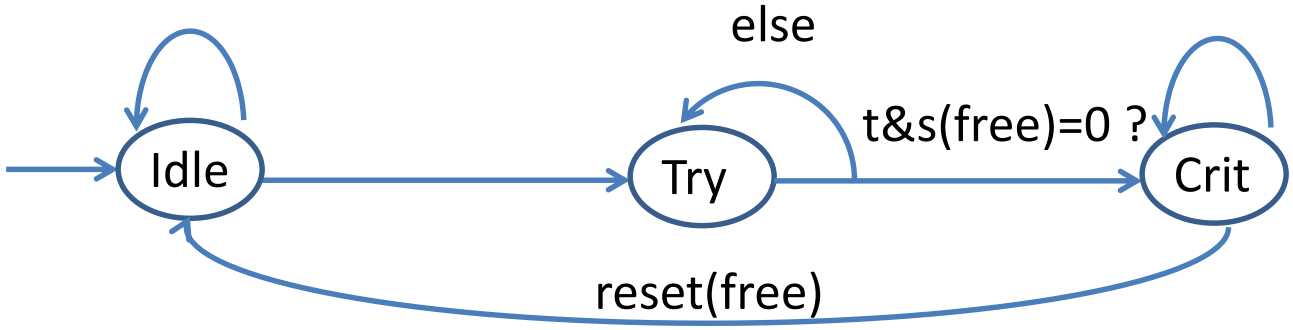
Test&Set Register



Mutual Exclusion using Test&Set Register

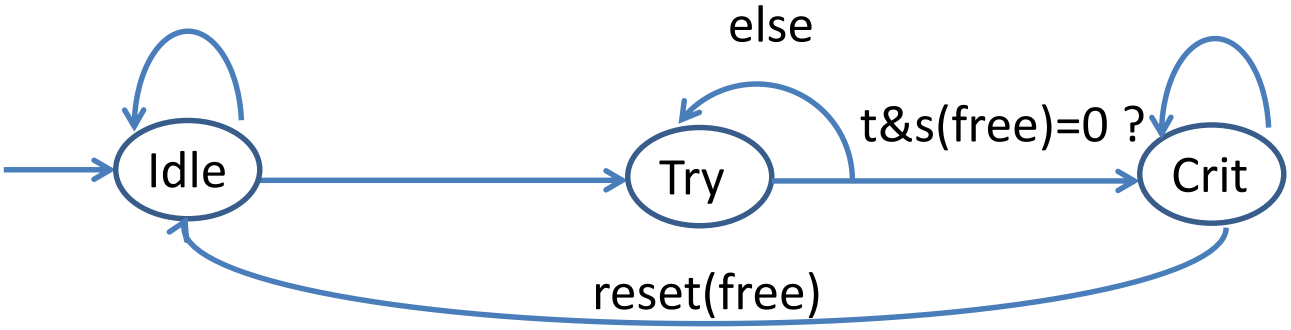
Test&SetReg free:= 0

Process P1

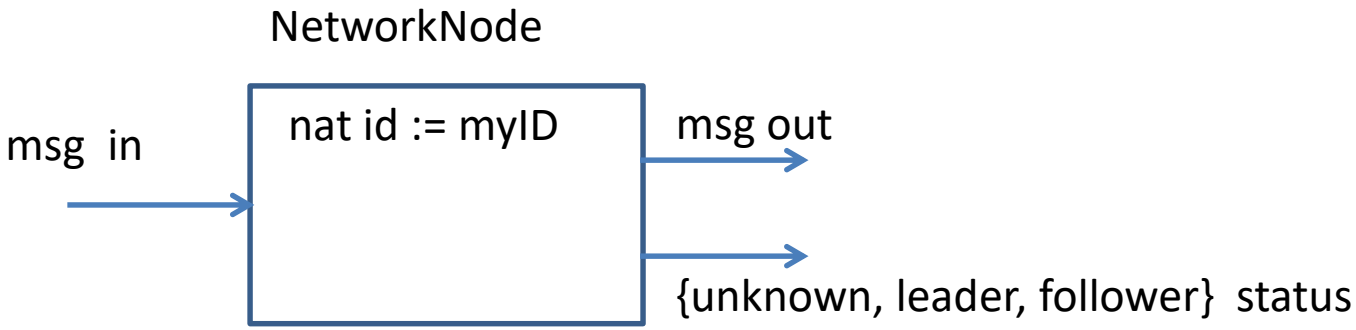


Is this correct?

Process P2



Leader Election



- Classical coordination problem: Elect a unique node as a leader
  - Exchange messages to find out which nodes are in network
  - Output the decision using the variable status
- Requirements
  - Eventually every node sets status to either leader or follower
  - Only one node sets status to leader

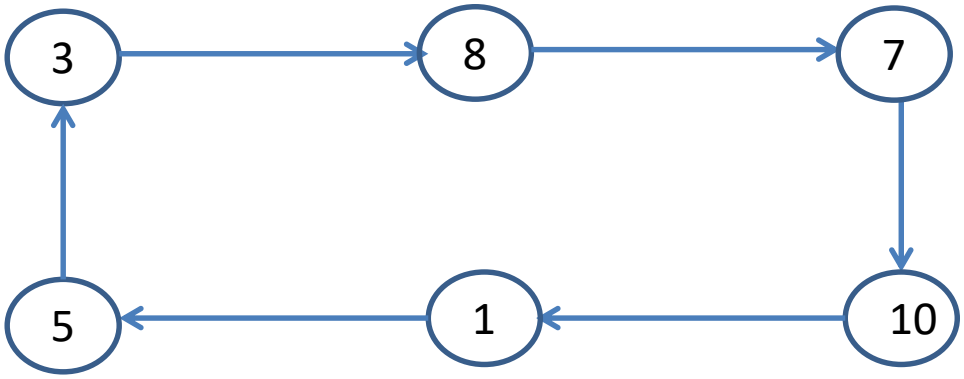
Asynchronous Leader Election

- Asynchronous network
  - Channel models directed network link
  - If there is a channel/link between nodes m and n, then synchronization on this channel allows m to send a message to n

- Key challenge compared to the synchronous case
  - There is no notion of a global round
  - Synchronous solution strategy relies on "executing protocol for k rounds implies that message has traveled k hops", does not work!

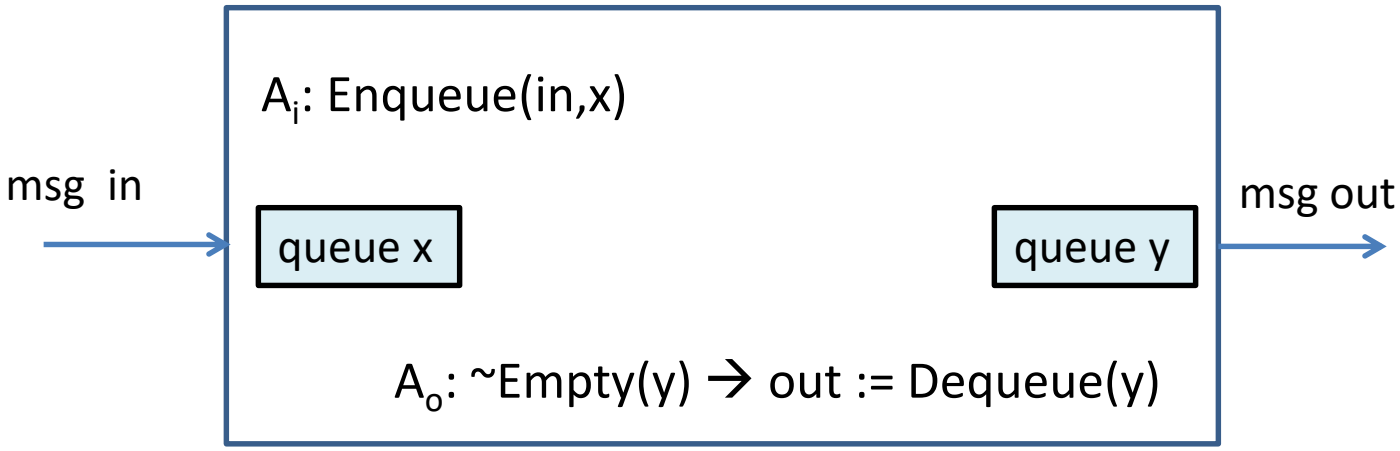
- Assume: Processes are connected in a unidirectional ring
  - Protocols for general topologies exist, but are more complex

Sample Asynchronous Ring Network



- Each process has a unique identifier
- A process does not know the size of the ring (number of processes)
- Execution model is asynchronous
- No failures: each process executes its protocol faithfully

Asynchronous Execution in a Ring



- One step in the execution of the system is either
  - A step local to one process, or
  - A communication step that transfers the message at front of the output queue y of a process to back of the input queue x of its right neighbor

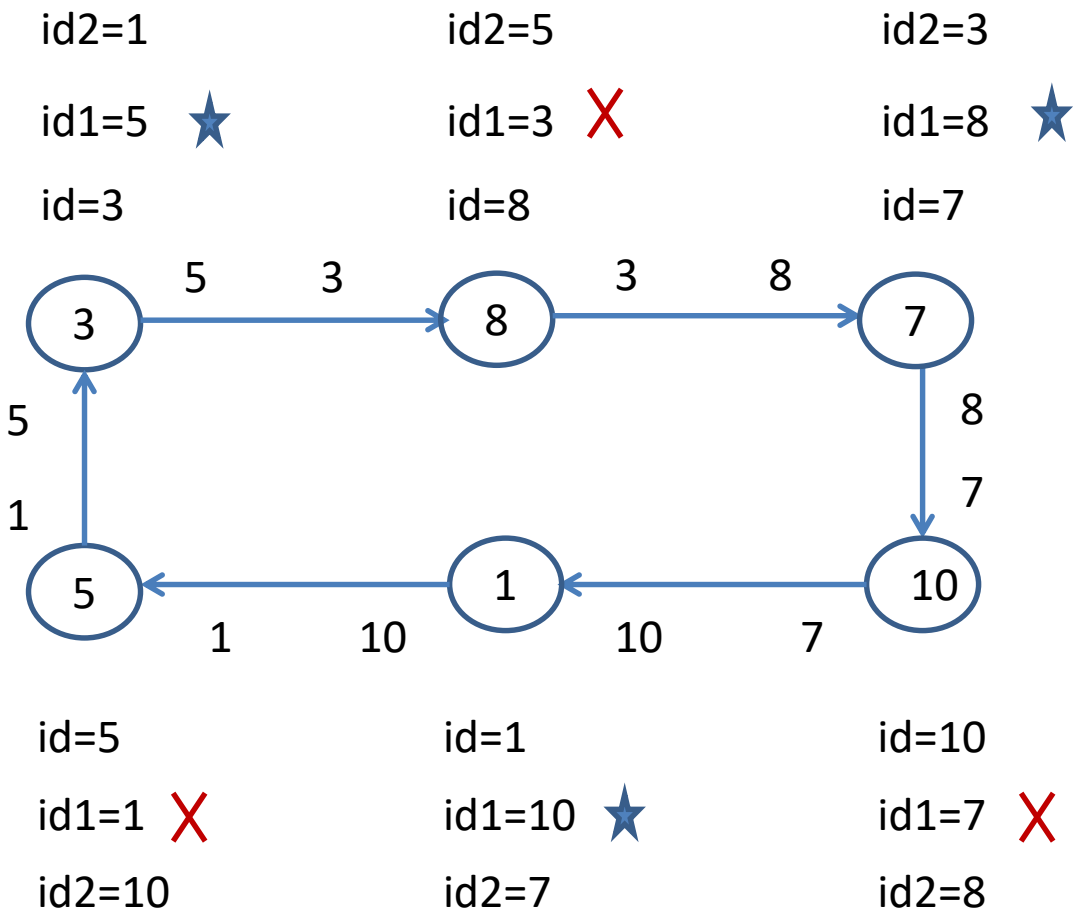
Adopting Synchronous Algorithm

- Set variable id to MyID, and initialize output queue y to contain id
- Local step/task
  - Remove a value v from queue x
  - If v > id, then change id to v, and enqueue this value in queue y
- When should a process stop and decide?
  - If v equals id !
  - This would imply that the value has traversed the entire ring
- What is an upper bound on the number of messages exchanged?
  - Quadratic, $O(N^2)$, where N is number of processes

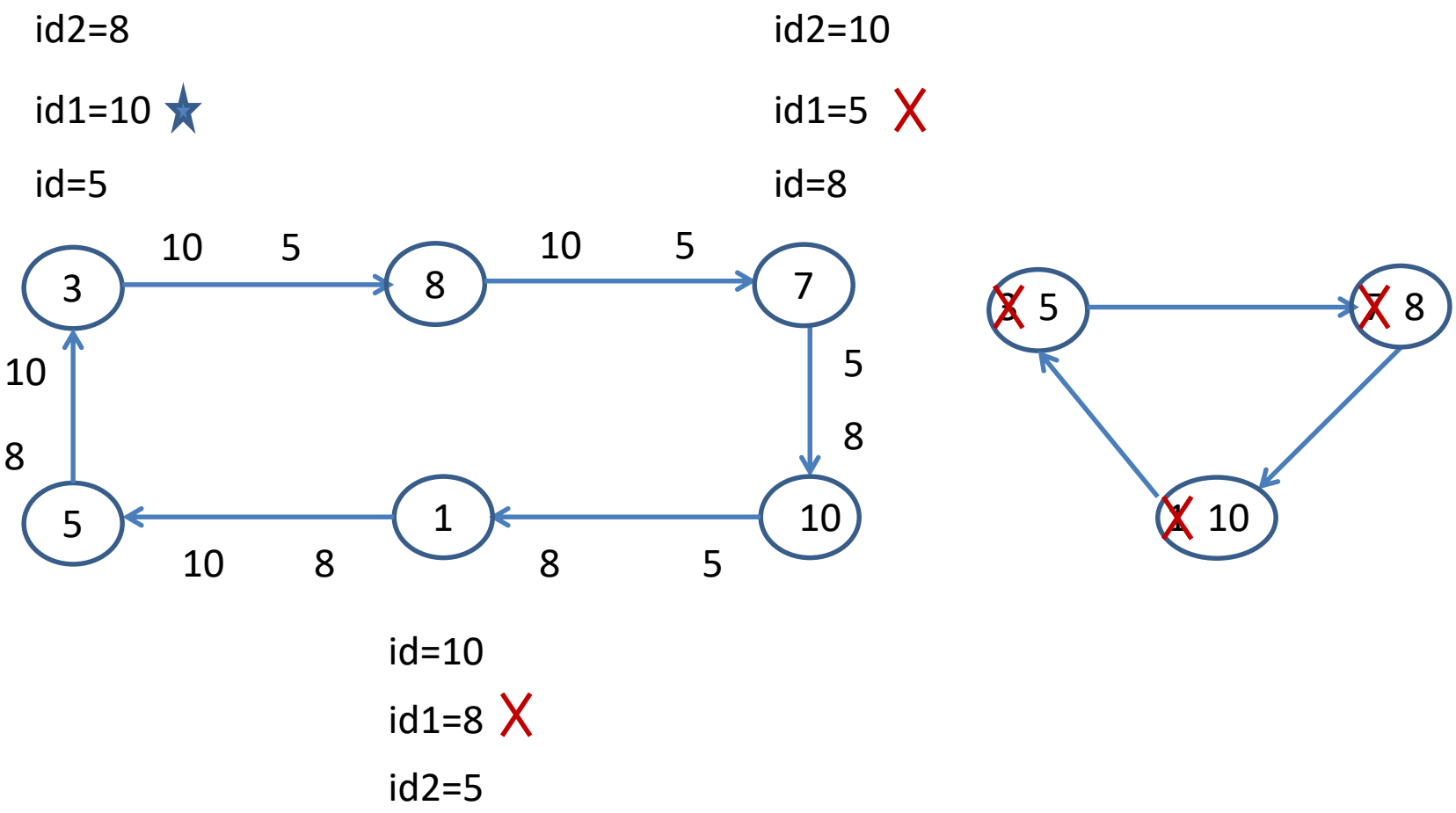
Improved Algorithm

- Set variable id to MyID, and initialize output queue y to contain id, which will be communicated to right neighbor
- When you receive a value from left neighbor, store it in state variable id1, and also relay it right neighbor (add it to output queue)
- Receive another value from left neighbor, call it id2
  - id = your value, id1 = left neighbor, id2 = left-left neighbor
- If id1 is the max of these three values, set id to id1, and repeat the above steps
  - Continue to next phase as active, but with different identifier
- If not, then decide to be a follower: continue as a passive participant
  - Does not generate any new messages, just transmits messages in input queue to output queue

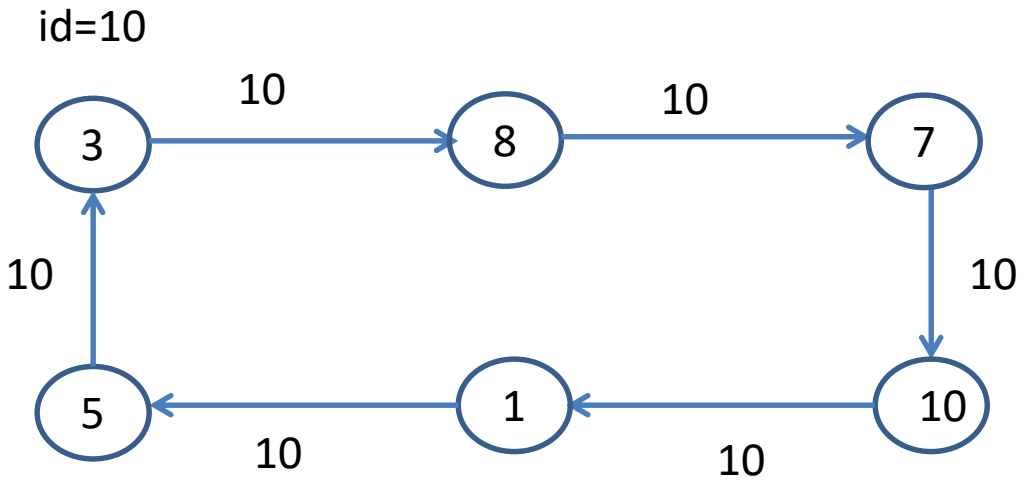
Example Execution

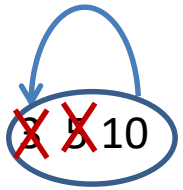


Example Execution



Example Execution



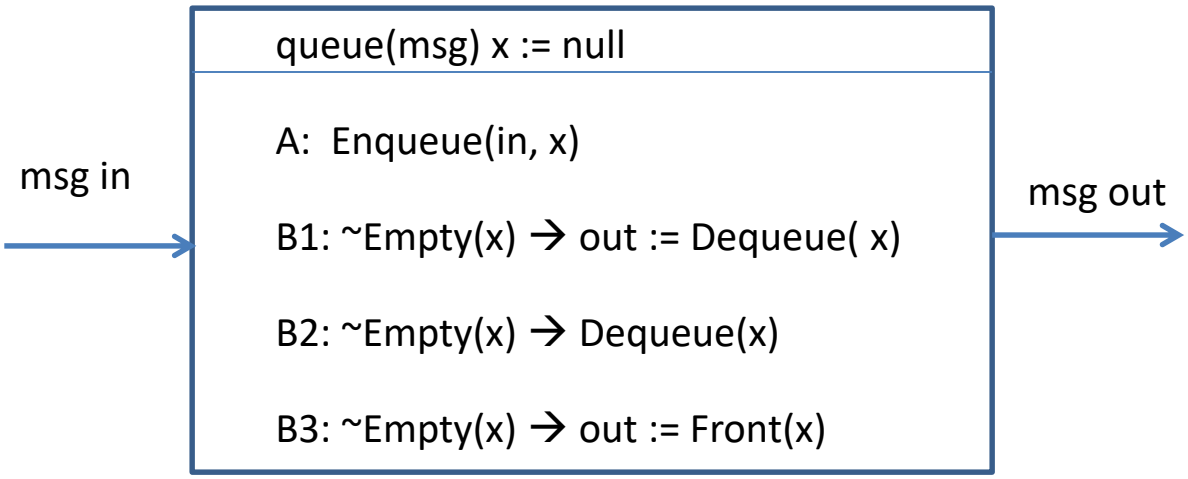


If first message from left neighbor equals id, stop and become the leader!

Algorithm Properties

- Actual execution proceeds asynchronously
  - Messages are processed at arbitrary times
  - Different processes may be executing different "phases"
- The process that becomes leader does not have highest (original) identifier
- In each phase, each process sends only 2 messages
- Among processes active during a phase, if a process continues to next phase as active, then its left neighbor cannot stay active (why?)
- At least one and at most half processes continue to next phase
  - Construct scenarios for these two extremes
  - For a ring of N processes, at most (log N) phases, so a total of O(N log N) messages
  - Matching lower bound: Cannot solve leader election in a ring while exchanging fewer messages

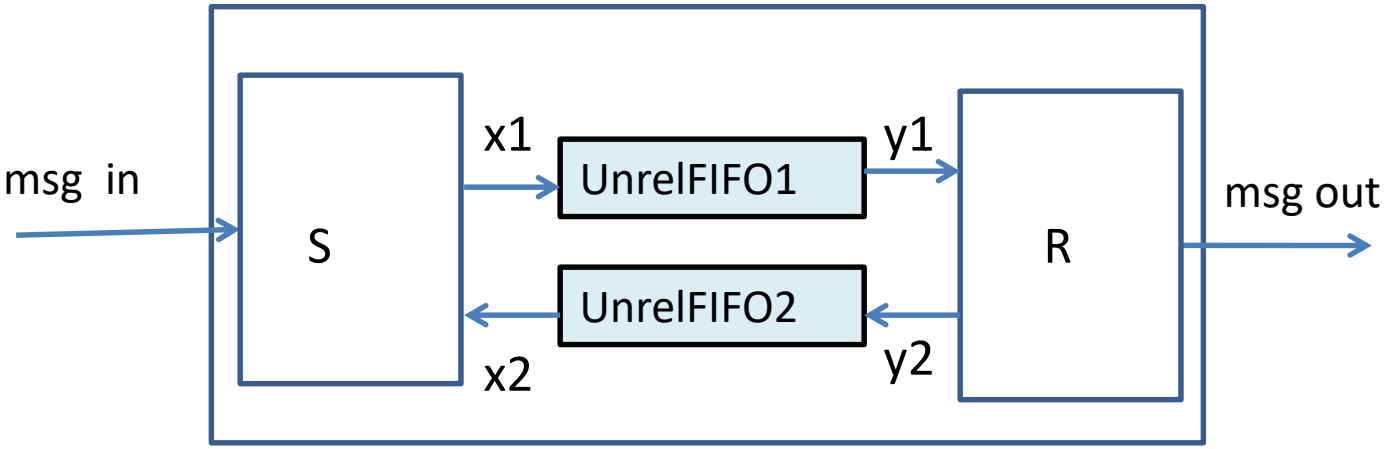
Unreliable FIFO



Models a link that may lose messages and/or duplicate messages

How to implement a reliable FIFO link using unreliable ones?

Reliable Transmission Problem

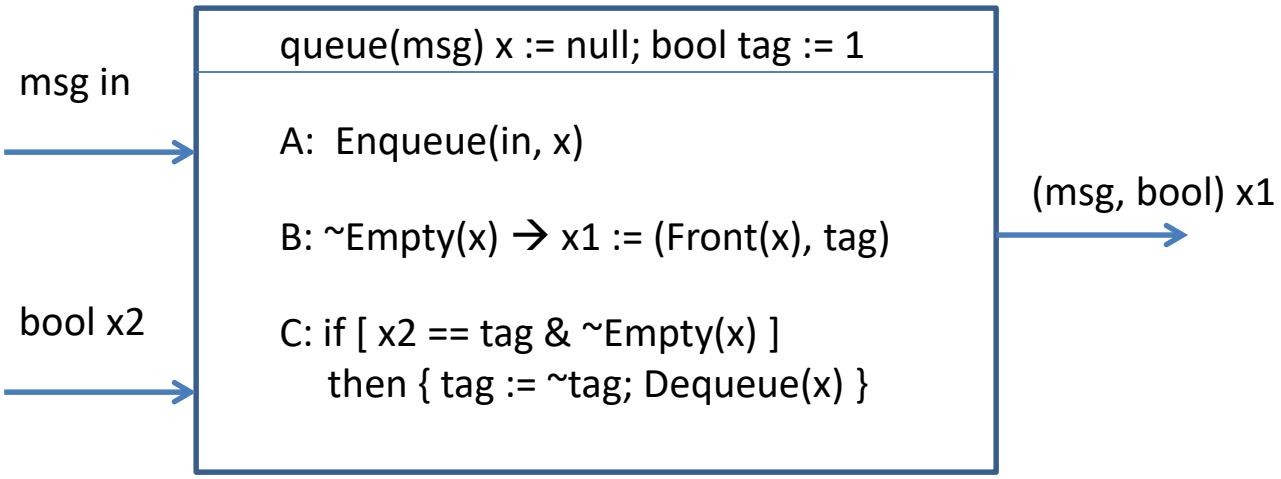


Design Asynchronous processes S and R so that the sequence of messages received on the channel in coincides with the sequence of messages delivered on the channel out

Alternating Bit Protocol

- How can the sender S be sure that receiver R got a copy of the message in presence of message losses?
  - S must repeatedly send a message
  - R must send back an acknowledgement, and do so repeatedly
- How can the receiver R distinguish between a duplicated/repeated copy and a fresh message?
  - Each message must be tagged with "extra" bits
- Alternating bit protocol:
  - Key insight: Tagging each message as well as acknowledgement with a single bit suffices
  - Both S and R keep a local tag bit
  - if the tag of incoming message matches with the local tag, message is considered "fresh", and local tag is toggled

ABP Sender



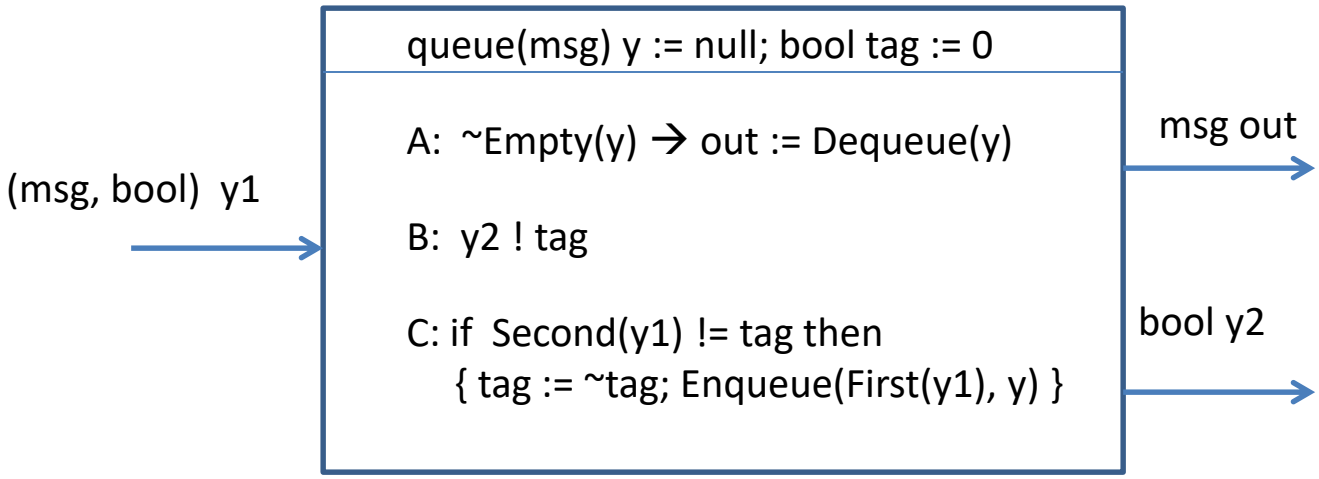
Task A: Store incoming messages in queue x

Task B: Transmit message at front of queue x tagged with local tag

Do not remove the message: this ensures it is transmitted repeatedly

Task C: If ack matches tag, then message successfully delivered; so remove it from x, and flip tag

ABP Receiver



Task A: Transmit outgoing messages from queue y to output channel

Task B: Transmit local tag as acknowledgement on channel y2

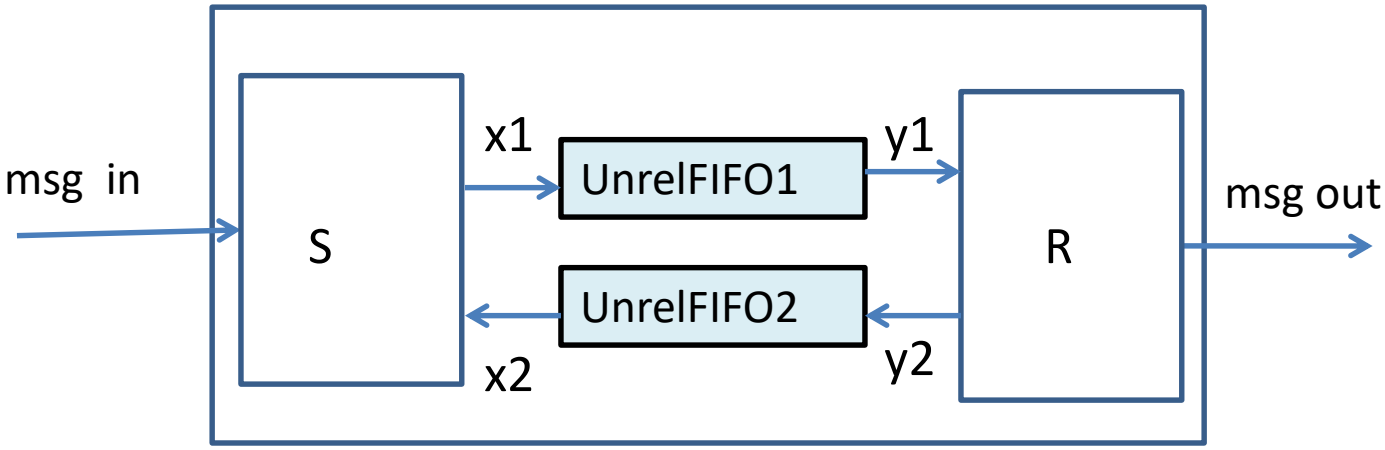
Note: Same ack is potentially transmitted repeatedly

Task C: If tag of incoming message matches local tag, then message is new; so add it y and flip tag

ABP Sample Execution

- Initially S.tag =1 and R.tag=0
- Suppose S receives a message m to be delivered
- S repeatedly sends (m,1) over unreliable link
- Eventually, R gets at least one, maybe multiple, copies of (m,1)
- Meanwhile, R is sending 0, possibly multiple times, as acknowledgement, but all these acks are simply ignored by S
- When R gets (m,1) the first time, it stores m in queue y (and this message will then eventually be transmitted on out), and sets tag to 1
- Duplicate versions of (m,1) are ignored by R
- R repeatedly send the acknowledgment 1 over unreliable link
- Eventually, S gets at least one ack = 1, and then, it removes m from input queue, and sets its tag to 0
- Duplicate versions of ack=1 are ignored by S
- Messages received as input are queued up in x, and S will now repeat the whole cycle by sending next message m' along with tag 0

ABP Variations



- Suppose unreliable link can lose messages, but is guaranteed not to duplicate a message, can we simplify the protocol?

- Suppose unreliable link can also reorder messages (in addition to losing and duplicating messages), how should we modify the protocol to ensure reliable transmission?