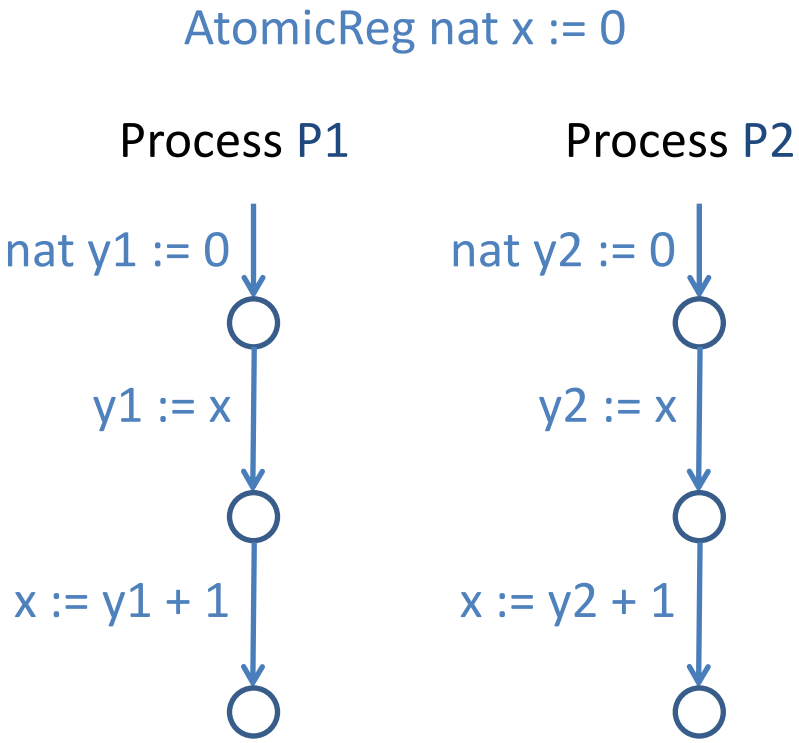
CS:4980
Foundations of Embedded Systems

Asynchronous Model
Part II

Shared Memory Programs



Declaration of shared variables

+ code for each process

Key restriction:

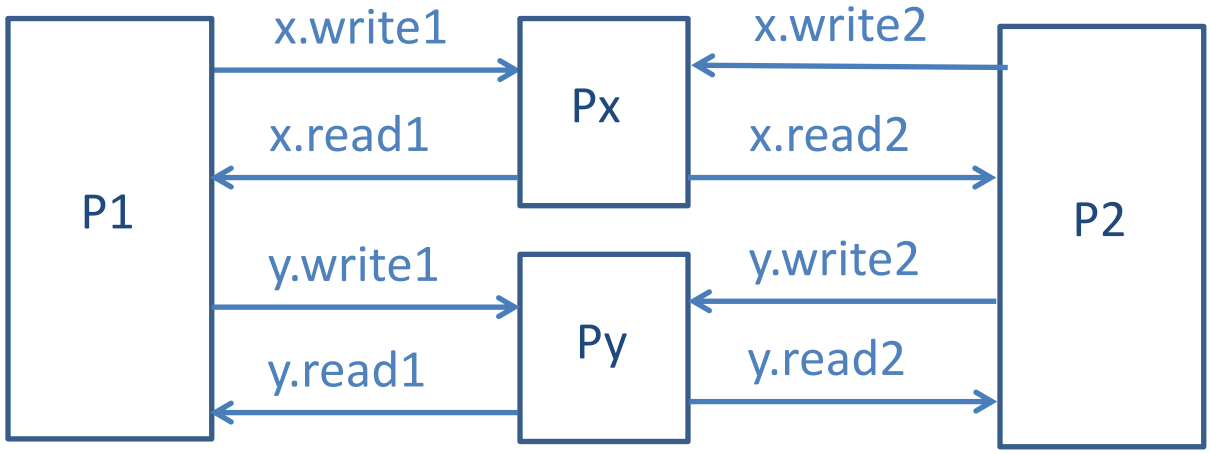
each process statement either

- changes local variables,
- reads (single) shared var x, or
- writes shared var x

Execution model: execute one step of one of the processes

Can be formalized as asynchronous processes

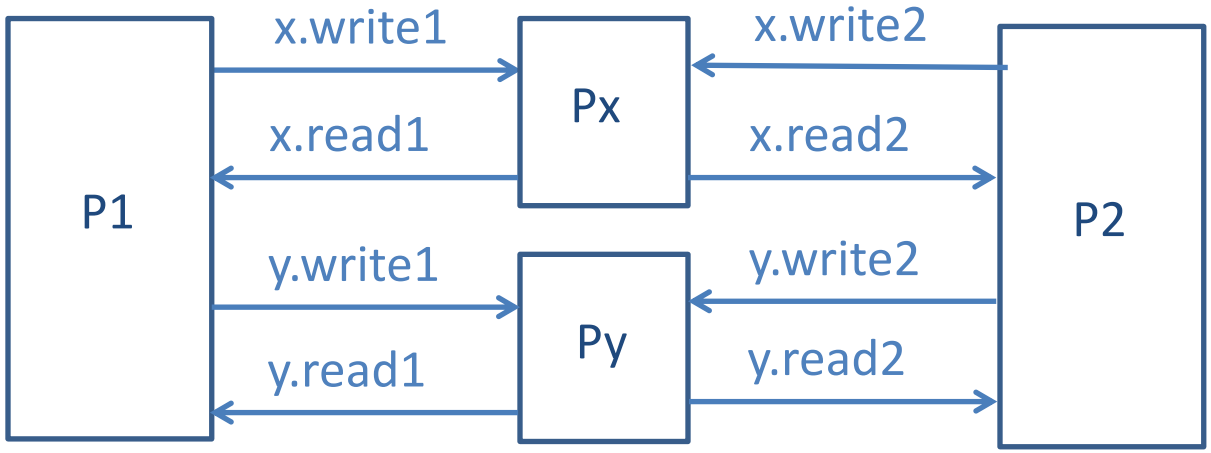
Shared Memory Processes



- Processes P_1 and P_2 communicate by reading/writing shared variables
- Each shared variable s can be modeled as an asynchronous process P_s

  For each such s, P_s has output channels s.read1, s.read2, and input channels s.write1, s.write2; its state stores the value of s
- In example above:
  - To write x, P1 synchronizes with Px on x.write1 channel
  - To read x, P2 synchronizes with Px on x.read2 channel

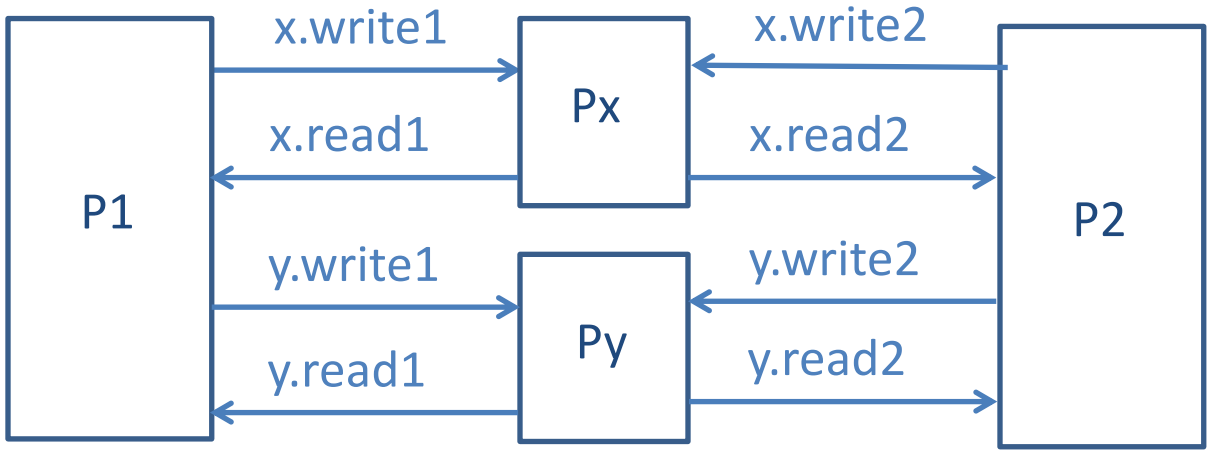
Atomic Registers



Note: By def. of asynchronous model, each step of above model is either

1. internal to P1 or P2, or
2. involves exactly one synchronization (read or write of one shared variable by one of the processes)

Atomic Registers

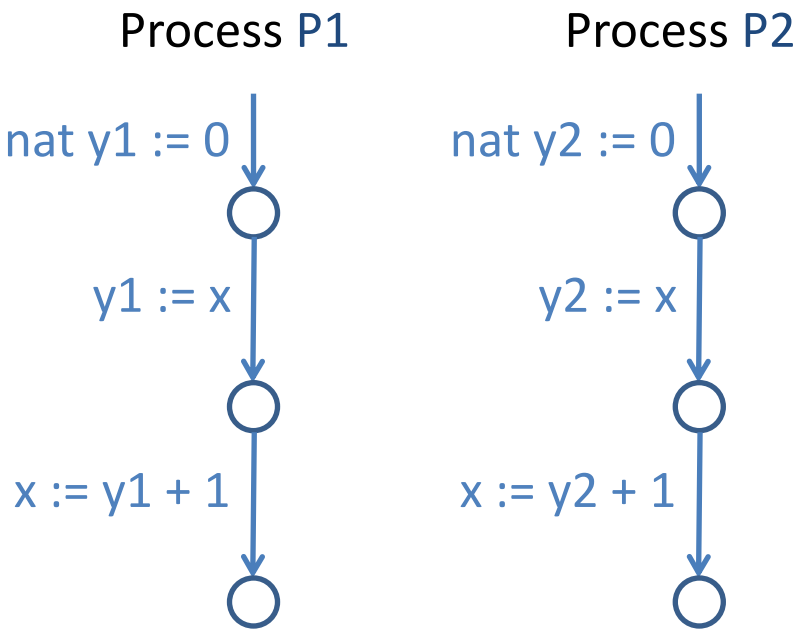


Atomic register: Basic primitives are read and write

- 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

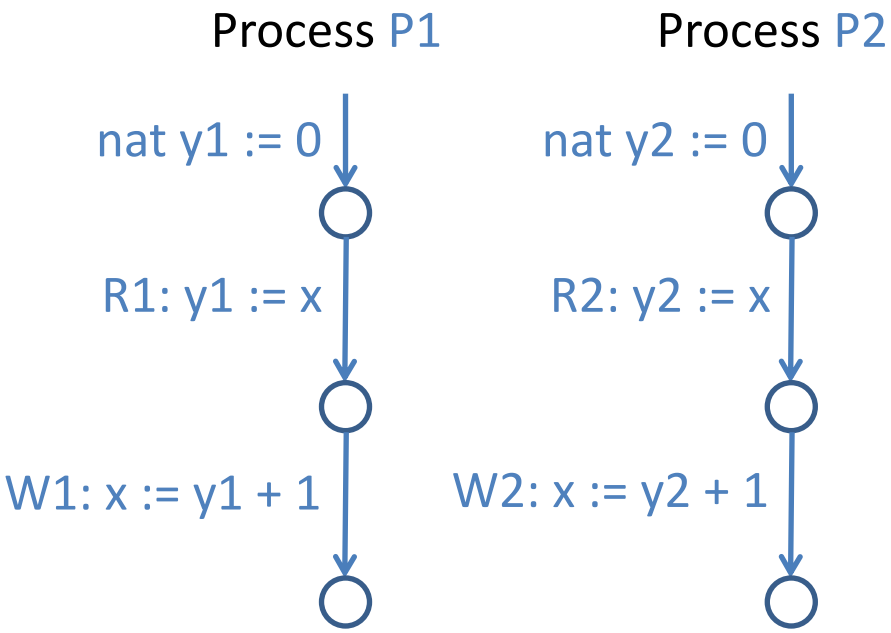
Shared Memory Programs

AtomicReg nat x := 0



Data Races

AtomicReg nat x := 0



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

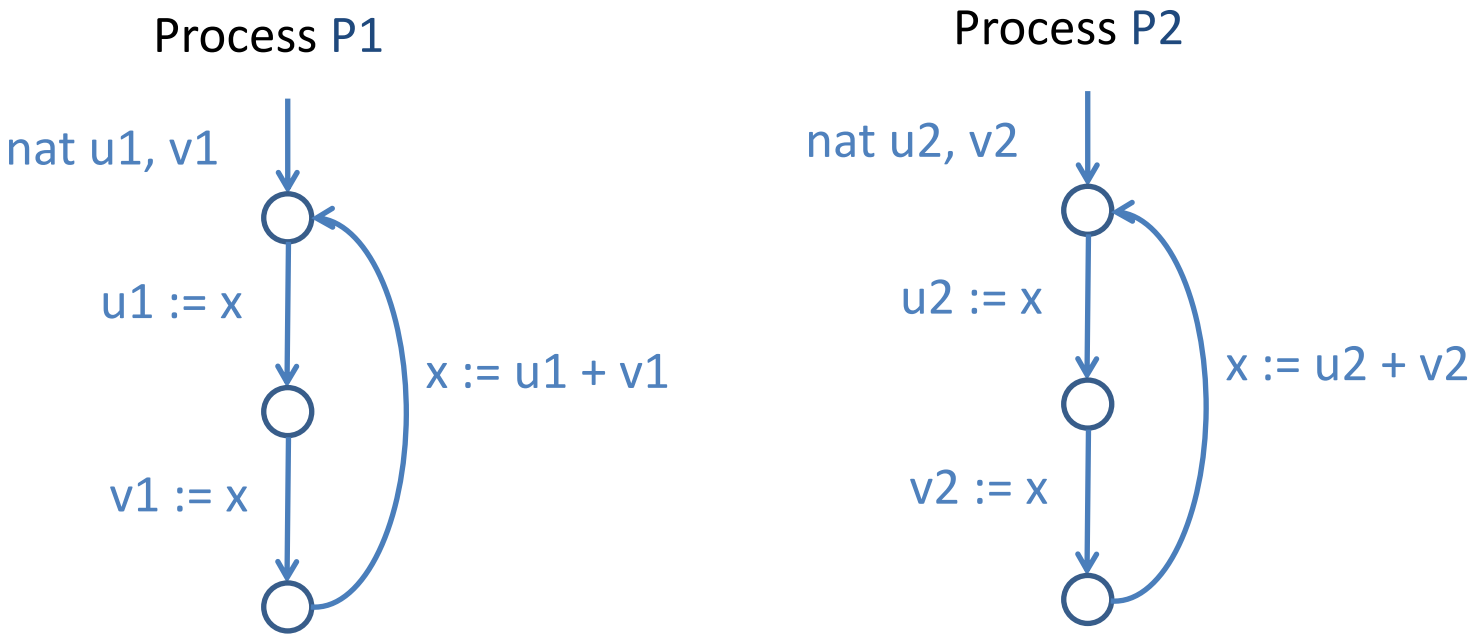
…

Data race: Concurrent accesses to shared object where the result depends on order of execution.

It should be avoided!

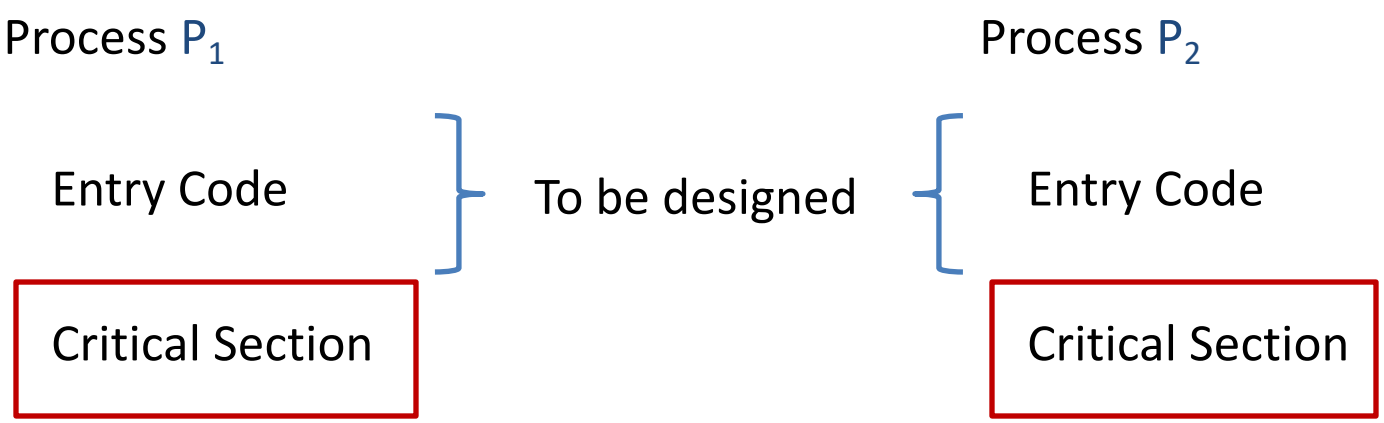
Exercise

AtomicReg nat x := 1



What are the possible values for the shared register x?

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

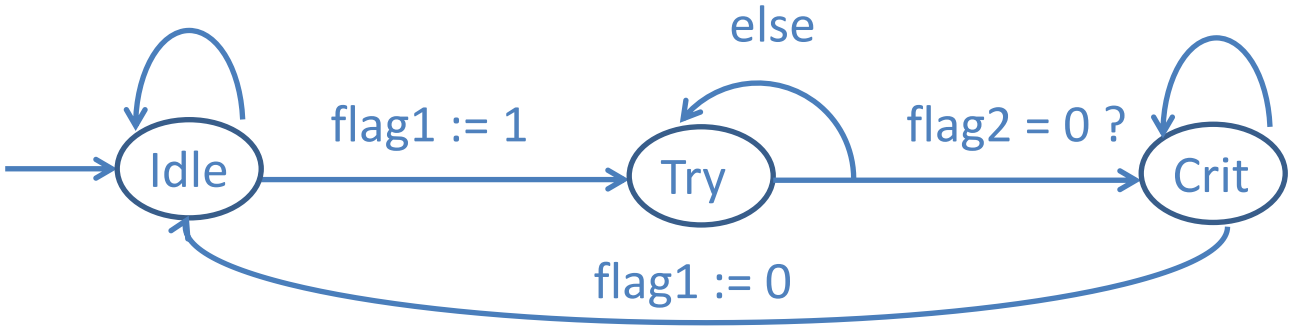
Safety requirement: both processes should not be in critical section simultaneously (can be formalized using invariants)

Progress requirement: if any process is trying to enter, then some process should be able to enter (no deadlocks)

Mutual Exclusion: First Attempt

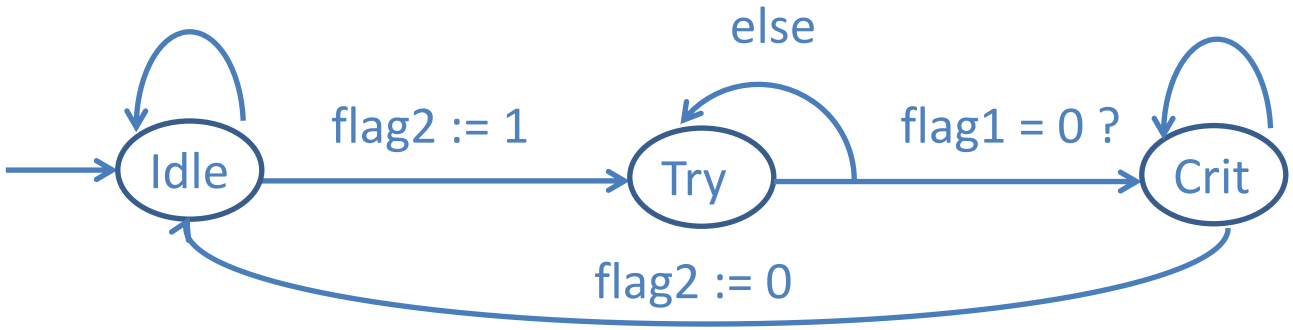
AtomicReg bool flag1 := 0 ; flag2 := 0

Process P1



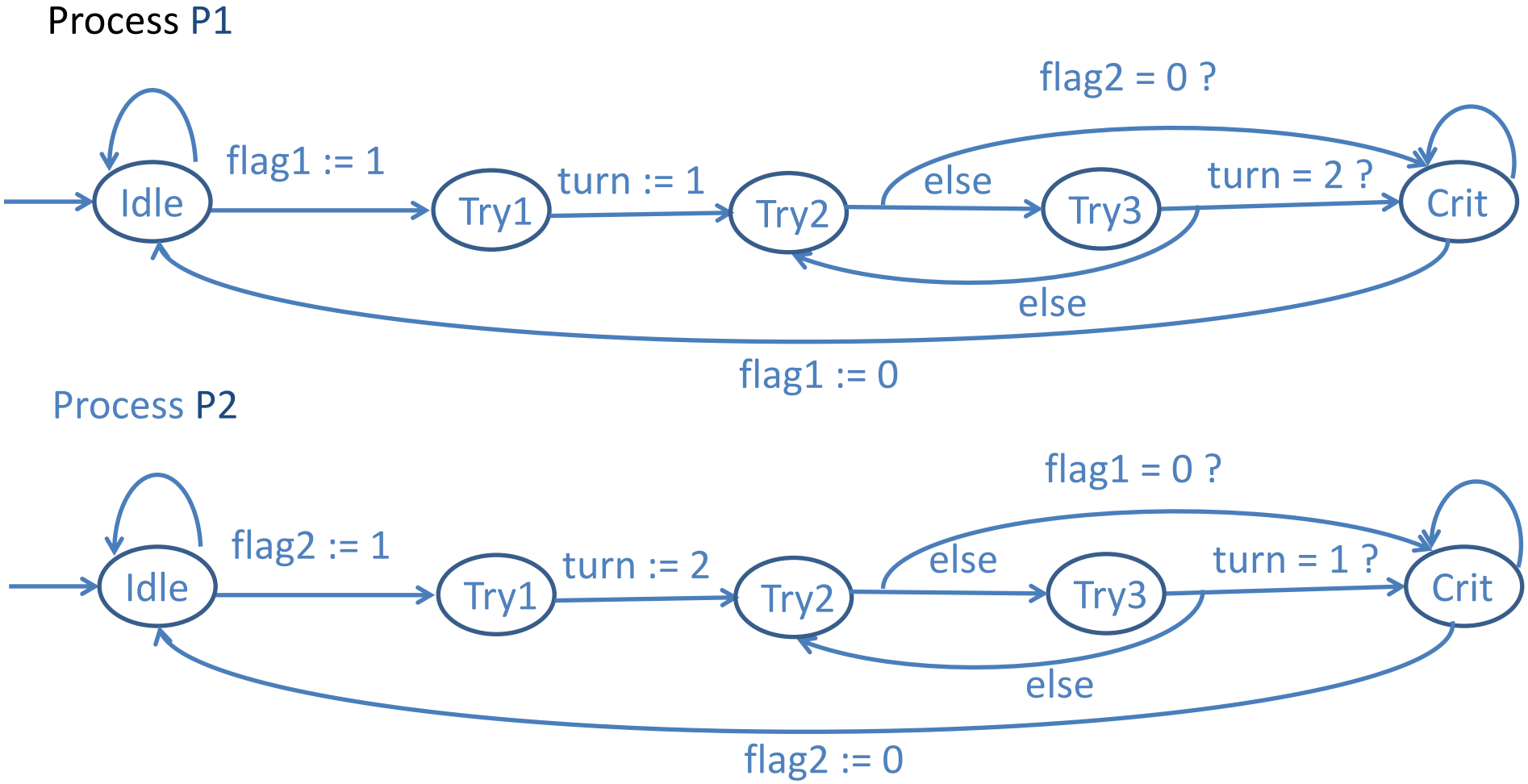
Is this correct?

Process P2



Peterson's Mutual Exclusion Protocol

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



Test&Set Register

- Beyond atomic registers:

  If in one atomic step, can do more than just read or write

  then we have stronger synchronization primitives

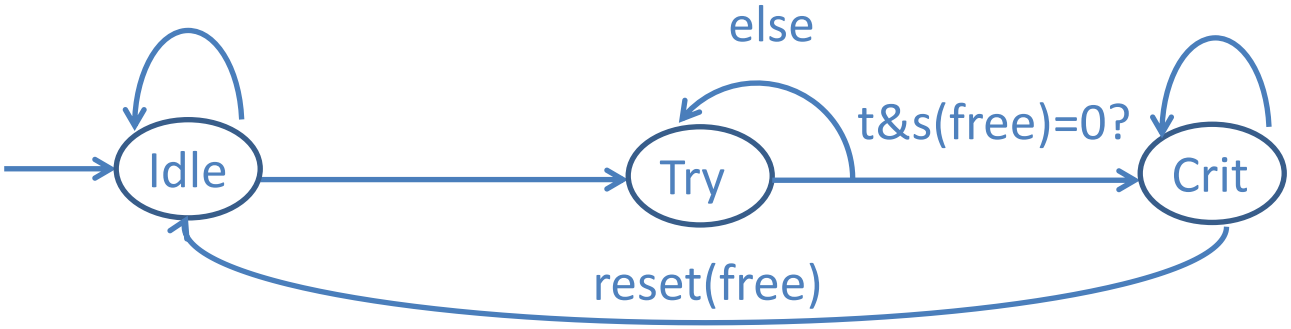
- *Test&Set Register*: holds a Booleans value
  - *Reset* operation: changes the value to 0
  - *Test&Set* operation: returns the old value and changes value to 1
  - If two processes are competing to execute Test&Set on a register with value 0, one will get back 0 and other will get back 1

- Modern processors support strong atomic operations
  - Ex: compare-and-swap; load-linked-store-conditional
  - Implementation is expensive (compared to read/write operations)

Mutual Exclusion using Test&Set Register

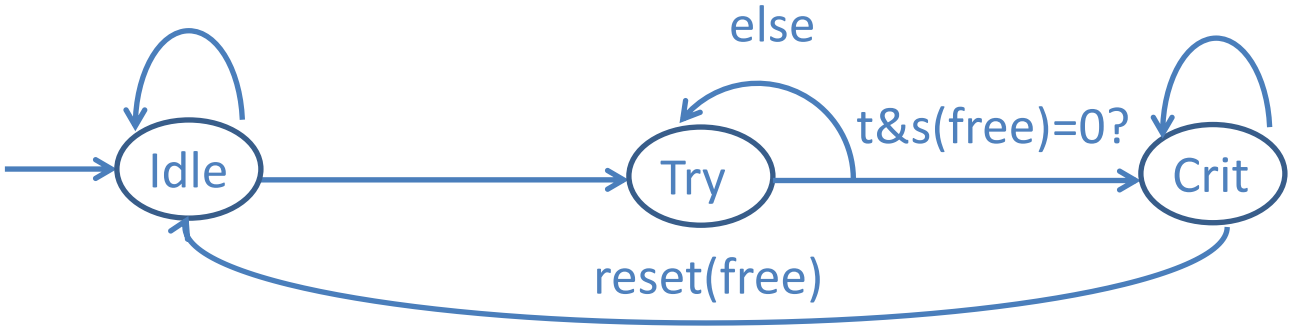
Test&SetReg free := 0

Process P1



Is this correct?

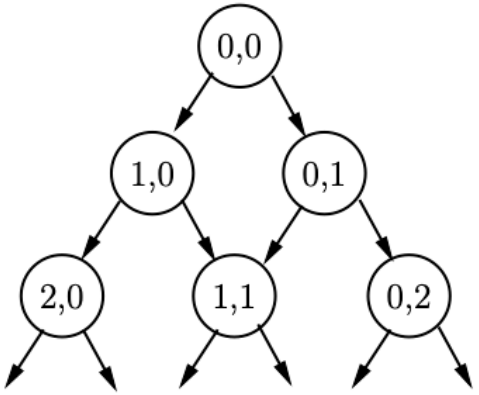
Process P2



Another Look at Asynchronous Execution Model

Process P

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



- Tasks A_x and A_y execute in an arbitrary order

  Motivation: If we establish that all possible executions of this design satisfy some requirement R, then every implementation of P will satisfy R

- Are the following realistic executions?
  - (0,0) $-A_x->$ (1,0) $-A_x->$ (2,0) $-A_x->$ (3,0) … $-A_x->$ (95,0) $-A_x->$ …
  - (0,0) $-A_x->$ (1,0) $-A_x->$ (2,0) $-A_y->$ (2,1) $-A_y->$ (2,2) … $-A_y->$ (2,95) …

- Does the system satisfy the following requirement:

  In every execution, values of both x and y eventually exceed 10

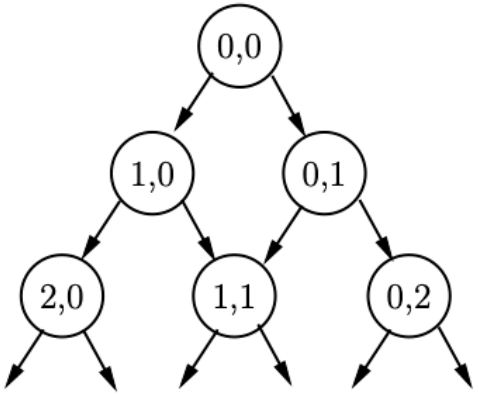
Fairness Assumption

P

nat x := 0 ; y := 0

A_x: x := x + 1

A_y: y := y + 1



Fairness assumption for a task

- Assumption about the underlying platform/scheduler
- Informally, an infinite execution is *unfair* to a task if the task does not get a chance to execute

- Unfair to A_y: (0,0) -A_x-> (1,0) -A_x-> (2,0) -A_x-> (3,0) … -A_x-> (95,0) …
- Unfair to A_x: (0,0) -A_x-> (1,0) -A_x-> (2,0) -A_y-> (2,1) -A_y-> (2,2) … (2,95) …
- Fairness assumptions restrict the set of possible executions to realistic ones without putting concrete bounds on relative speeds

Formalizing Fairness

Process P1

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

Process P2

nat x := 0 ; y := 0
B_x : x := x + 1
B_y : x = 0 -> y := y + 1

Definition 0. An infinite execution is *fair* to a task A, if the task A is executed repeatedly (i.e., infinitely often) during that execution

- Is this execution fair to task A_y? How about B_y?

  (0,0) -B_x-> (1,0) -B_x-> (2,0) -B_x-> (3,0) -B_x-> … (105,0) -B_x-> …

- After first step, the task B_y is not enabled, and so cannot be executed. This execution should not be considered unfair

Definition 1 (Weak fairness). An infinite execution is *fair* to a task A if, repeatedly, either A is executed or is disabled
(If enabled then eventually executed or disabled)

Weak vs Strong Fairness

Process P3

nat x := 0 ; y := 0
A_x : x := x + 1
A_y : even(x) -> y := y + 1

- Is this execution fair to task A_y?

  (0,0) $-A_x->$ (1,0) $-A_x->$ (2,0) $-A_x->$ (3,0) … $-A_x->$ (105,0) $-A_x->$ …

- According to weak fairness, yes, because A_y is disabled infinitely often

Definition 2 (Strong fairness). An infinite execution is *fair* to a task A, if task A is either executed repeatedly or disabled continuously from a certain step onwards
(If repeatedly enabled then repeatedly executed)

- Above execution is weakly fair to task A_y, but not strongly fair

Fairness Assumption

- Fairness assumptions for an asynchronous process P:

  For each output and internal task, either

  - no assumption,
  - weak fairness assumption, or
  - strong fairness assumption

- Restricts the set of possible infinite executions
  - If weak/strong fairness is assumed for a task A, then task scheduling should be such that executions are weakly/strong fair to A

- Affects whether process P meets a requirement R or not:
  - Maybe not all executions satisfy R, but all fair executions satisfy it

Requirements under Fairness Assumptions

Process P1

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

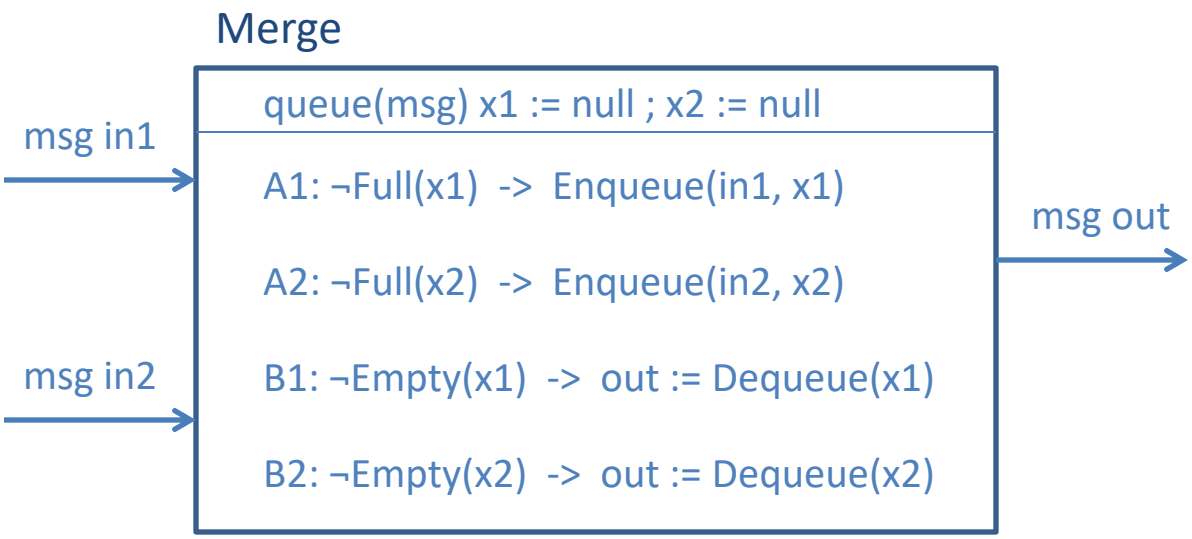
Process P3

nat x := 0 ; y := 0
B_x : x := x + 1
B_y : even(x) -> y := y + 1

Under what fairness assumptions do P1 and P3 satisfy these requirements?

- **R1:** eventually, x + y > 10
  - P1 and P3 both satisfy this, without any fairness assumption
- **R2:** eventually, x > 10
  - P1 with weak fairness for A_x, P3 with weak fairness for B_x
- **R3:** eventually, y > 10
  - P1 with weak fairness for A_y, P3 with strong fairness for B_y
- **R4:** eventually, x > y
  - neither, no matter what fairness assumption we make!

Asynchronous Merge

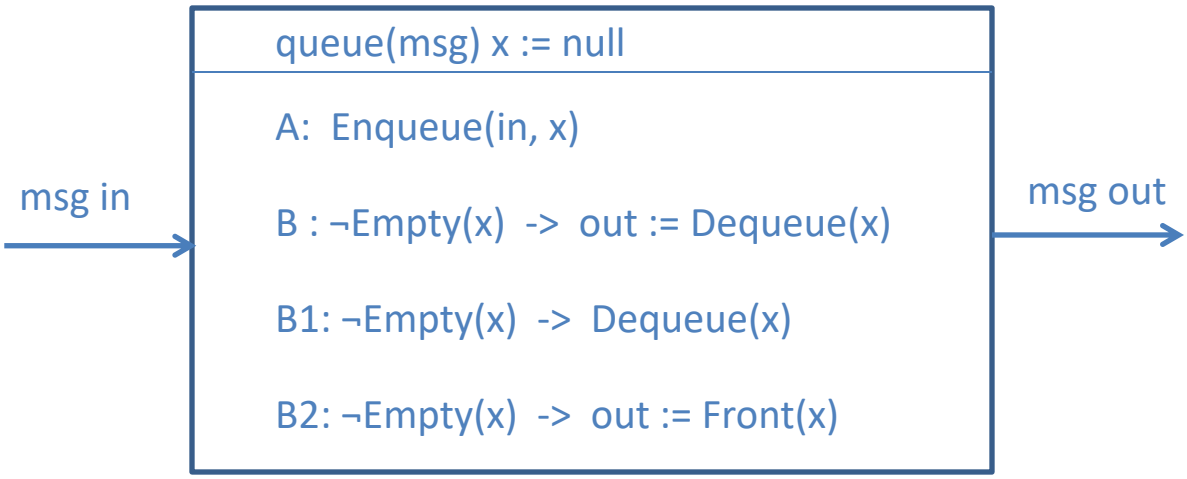


Requirement: whenever an input message is received, it is eventually output

Under which fairness assumptions does the requirement hold?

Weak fairness for tasks B1 and B2 suffices

Unreliable (Unbounded) FIFO



Tasks A and B model normal input/output behavior

Task B1 models message loss

Task B2 models message duplication

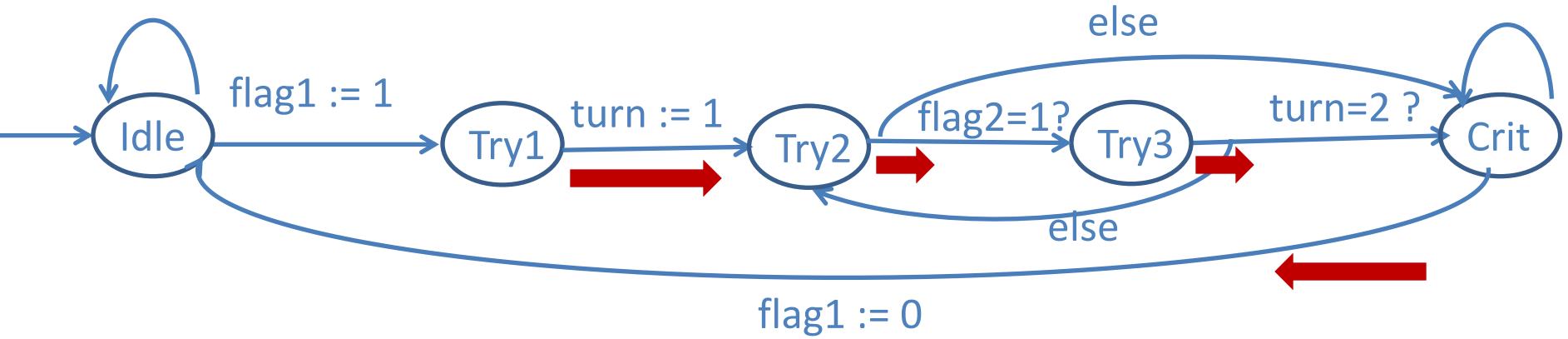
What are natural fairness assumptions for these tasks?

Strong fairness for B, no assumptions for B1 and B2

Fairness Assumptions for Mutual Exclusion Protocol

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

Process P1



Requirement: if a process ever wants to enter critical section,
it eventually will

What fairness assumptions should we make?

Weak fairness for highlighted steps/tasks

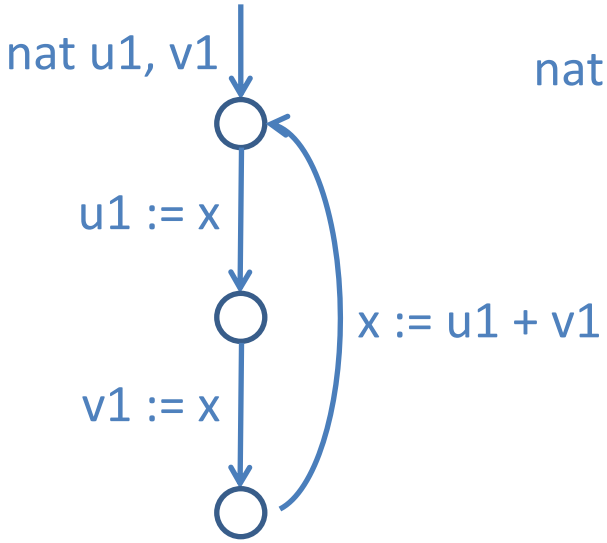
Fairness Summary

- A fairness assumption is an assumption made about the underlying platform or scheduler
  - The weaker the assumption, the better
- It restricts the set of possible infinite executions, allowing the satisfaction of more requirements
  - Does not affect the set of reachable states and safety properties
  - Does not change underlying coordination
- For each output and internal task, we can assume weak or strong fairness, as needed
  - Strong fairness is needed if the task can switch between enabled and disabled due to the execution of other tasks
- **Key distinction:** Fairness assumption for tasks (which ensures tasks get executed as expected) vs “fairness” requirements for protocols (which are about high-level goals of the problem being solved)

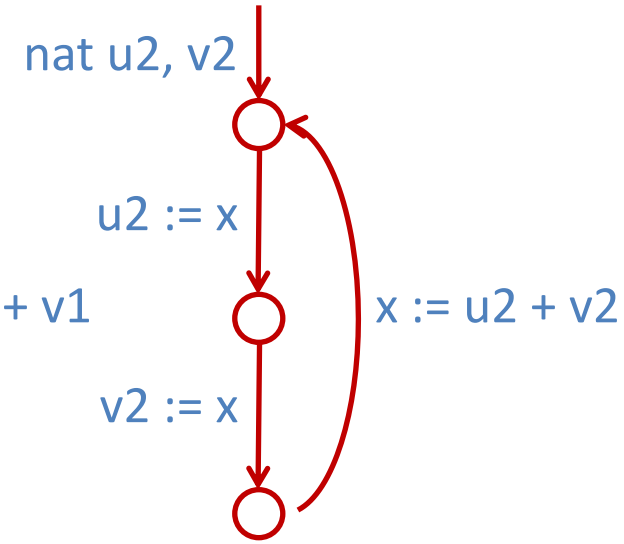
Exercise: Solution

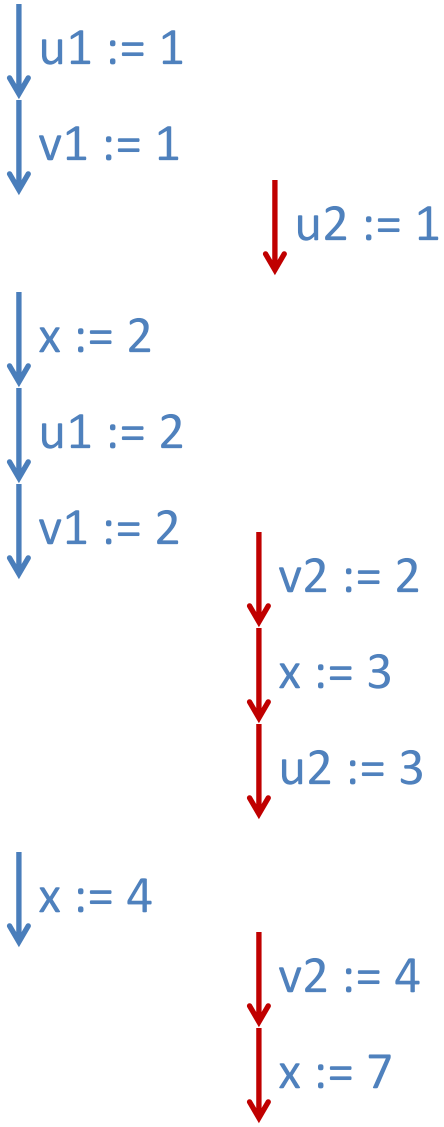
AtomicReg nat x := 1

Process P1



Process P2





What possible values can x take?

Every possible natural number!

Credits

Notes based on Chapter 4 of

Principles of Cyber-Physical Systems
by Rajeev Alur
MIT Press, 2015