Cyber-Physical Systems (CSC.T431)

Asynchronous Model (1)

Instructor: Takuo Watanabe (Department of Computer Science)

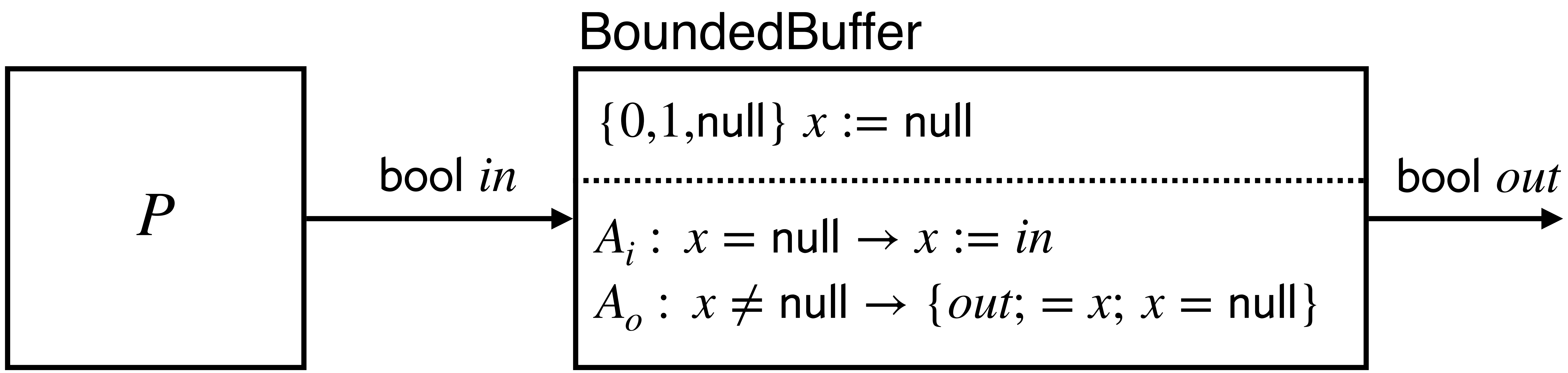
Agenda

- Asynchronous Model (2)

Course Support & Material

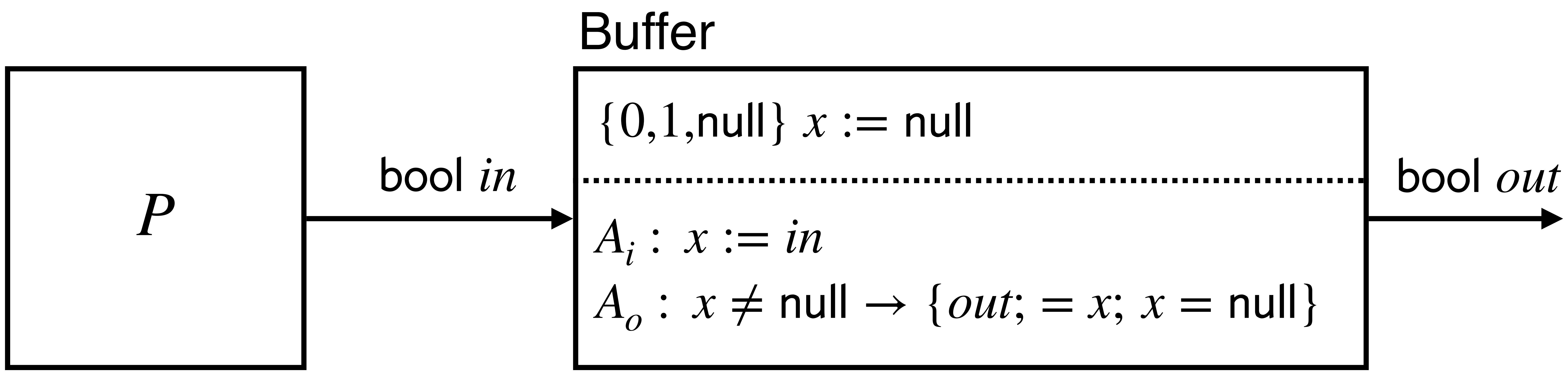
- Slides: OCW-i
- Course Web: https://titech-cps.github.io
- Course Slack: titech-cps.slack.com

Blocking Synchronization



- Let A be an output task of process P corresponding to the channel in.
- While the state variable x of BoundedBuffer is not null, A cannot be executed even if it is enabled.
- Such communication is called *blocking*.

Non-Blocking Synchronization



- P can communicate with Buffer at anytime it wants because the input task for in is always enabled.
- An asynchronous process is said do be *non-blocking* if for every input channel x and for every state s, some task in $\mathbf{A}_x$ is enabled in s.

Deadlocks

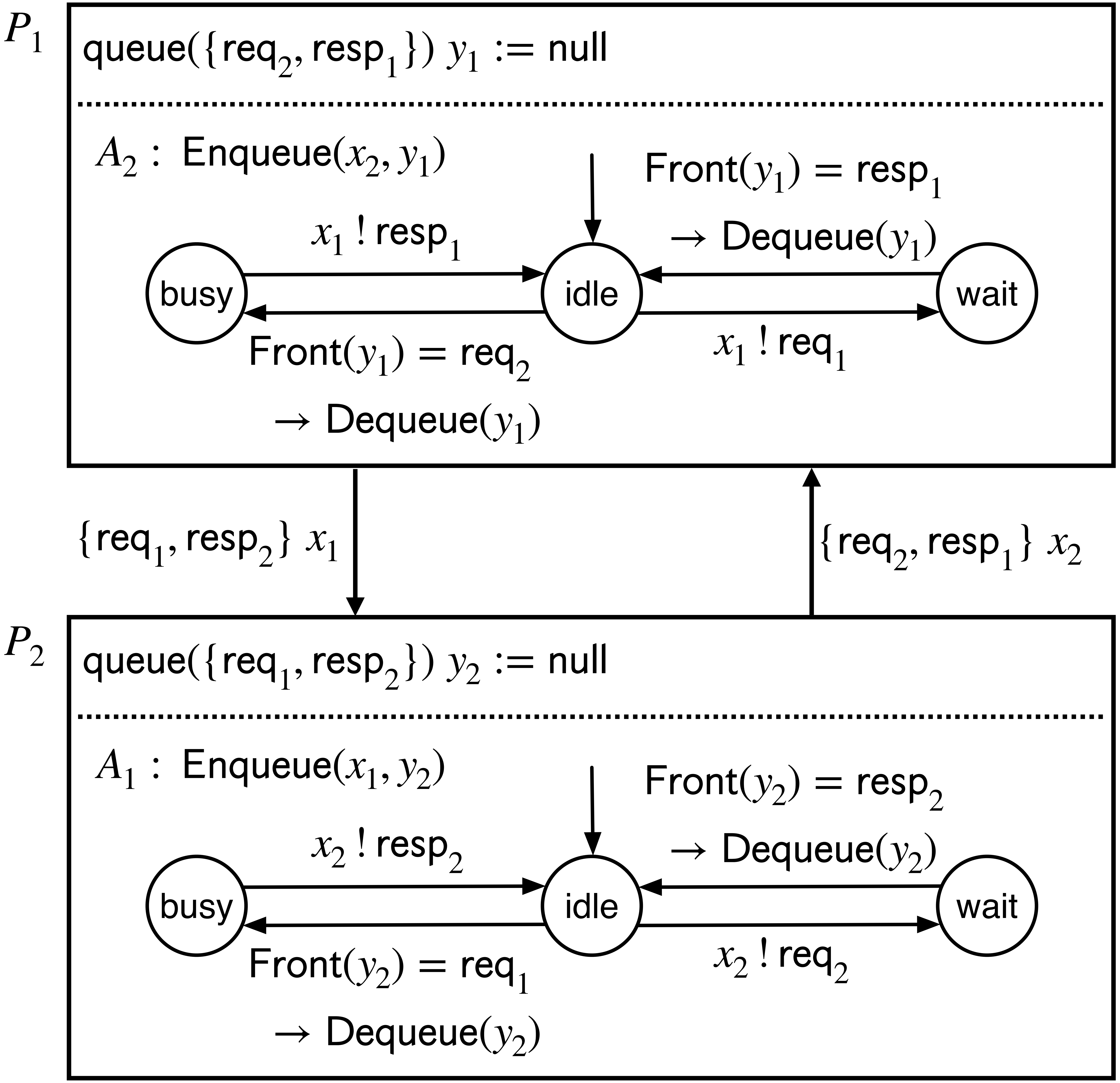
- A system consists of two or more processes is in a deadlock if each process is waiting for some other processes to execute, but no task is enabled.

- A state s of an asynchronous process P is a deadlock state if

  (1) no task is enabled in s, and

  (2) s does not correspond to a successful termination of the system.

Deadlock Example



state: $(P_1.mode, y_1, P_2.mode, y_2)$

(idle, null, idle, null)

$\downarrow x_1$!req$_1$

(wait, null, idle, [req$_1$])

$\downarrow \varepsilon$

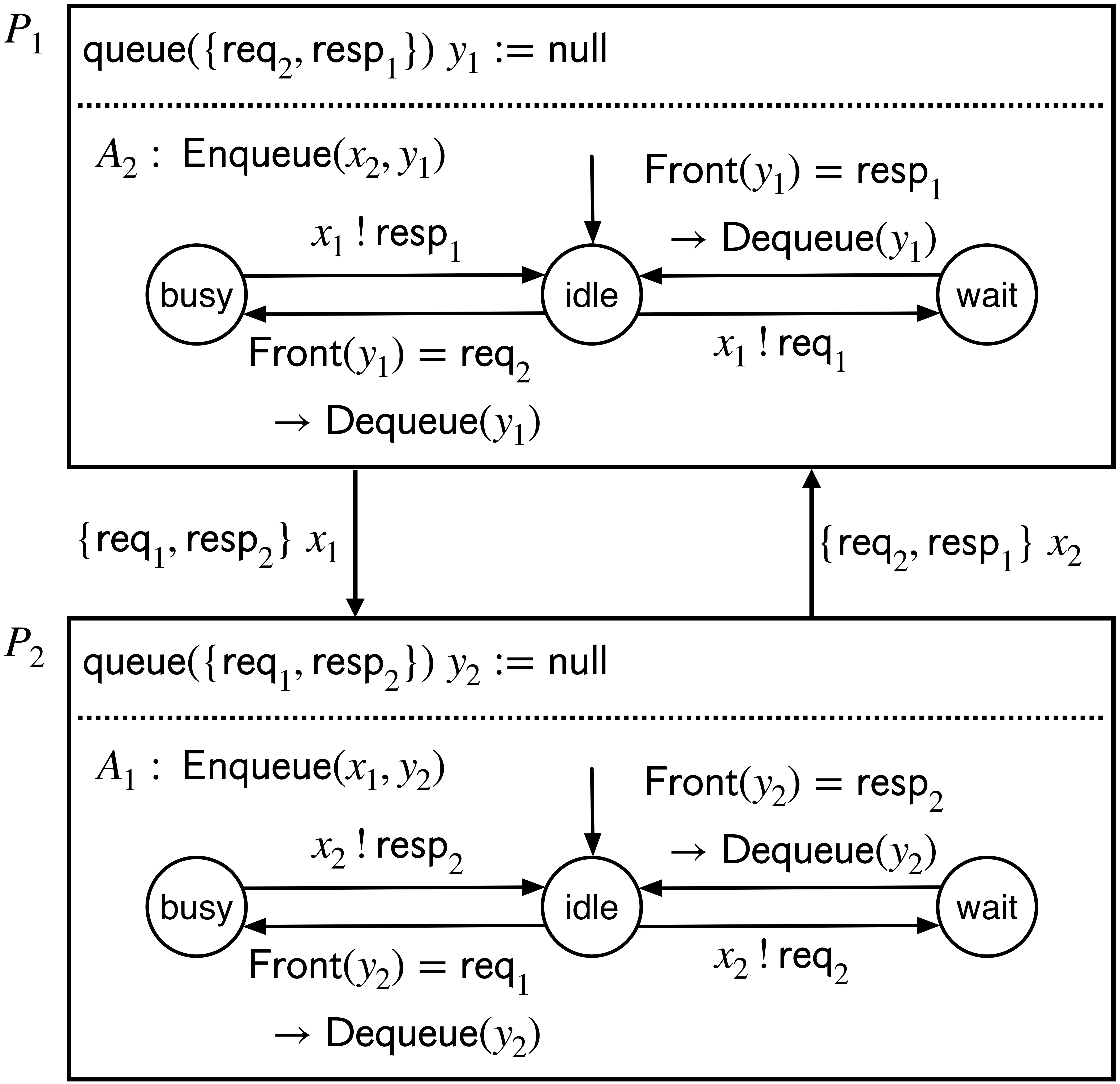
(wait, null, busy, null)

$\downarrow x_2$!resp$_2$

(wait, [resp$_2$], idle, null)

$\downarrow \varepsilon$

(idle, null, idle, null)



Deadlock Example

state: $(P_1.mode, y_1, P_2.mode, y_2)$

(idle, null, idle, null)

$\downarrow x_1$!req$_1$

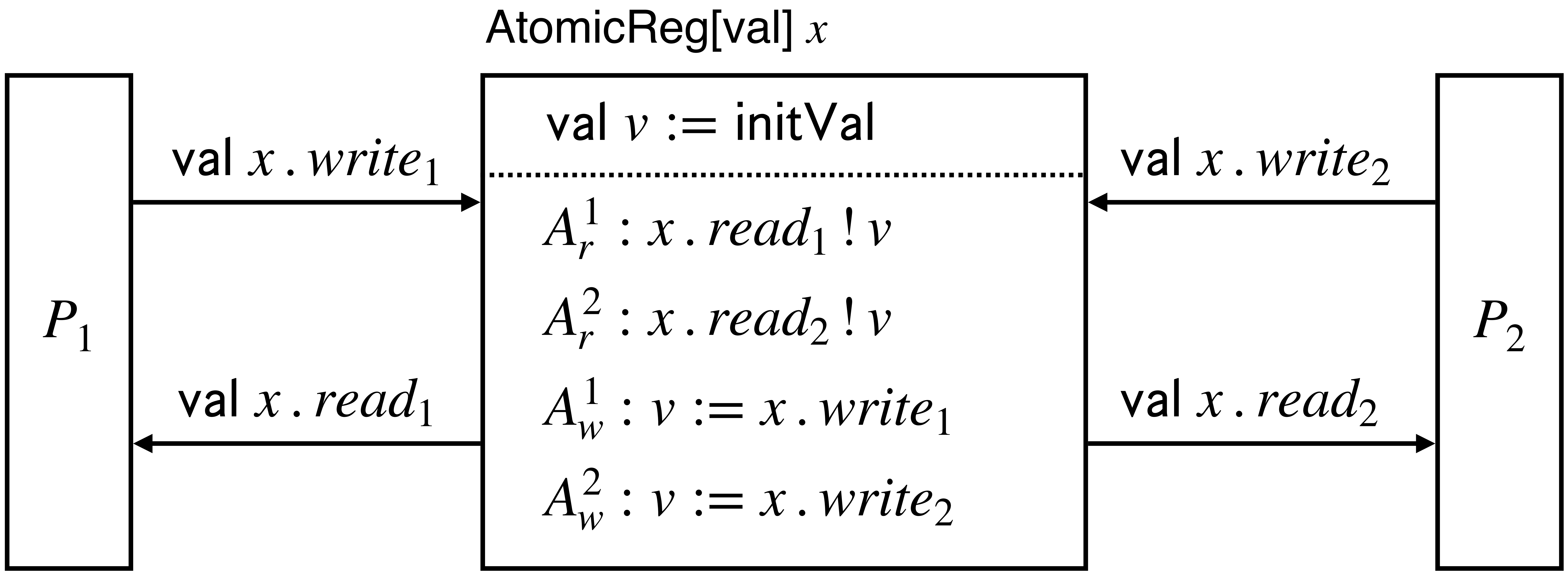
(wait, null, idle, [req$_1$])

$\downarrow x_2$!req$_2$

(wait, [req$_2$], wait, [req$_1$])

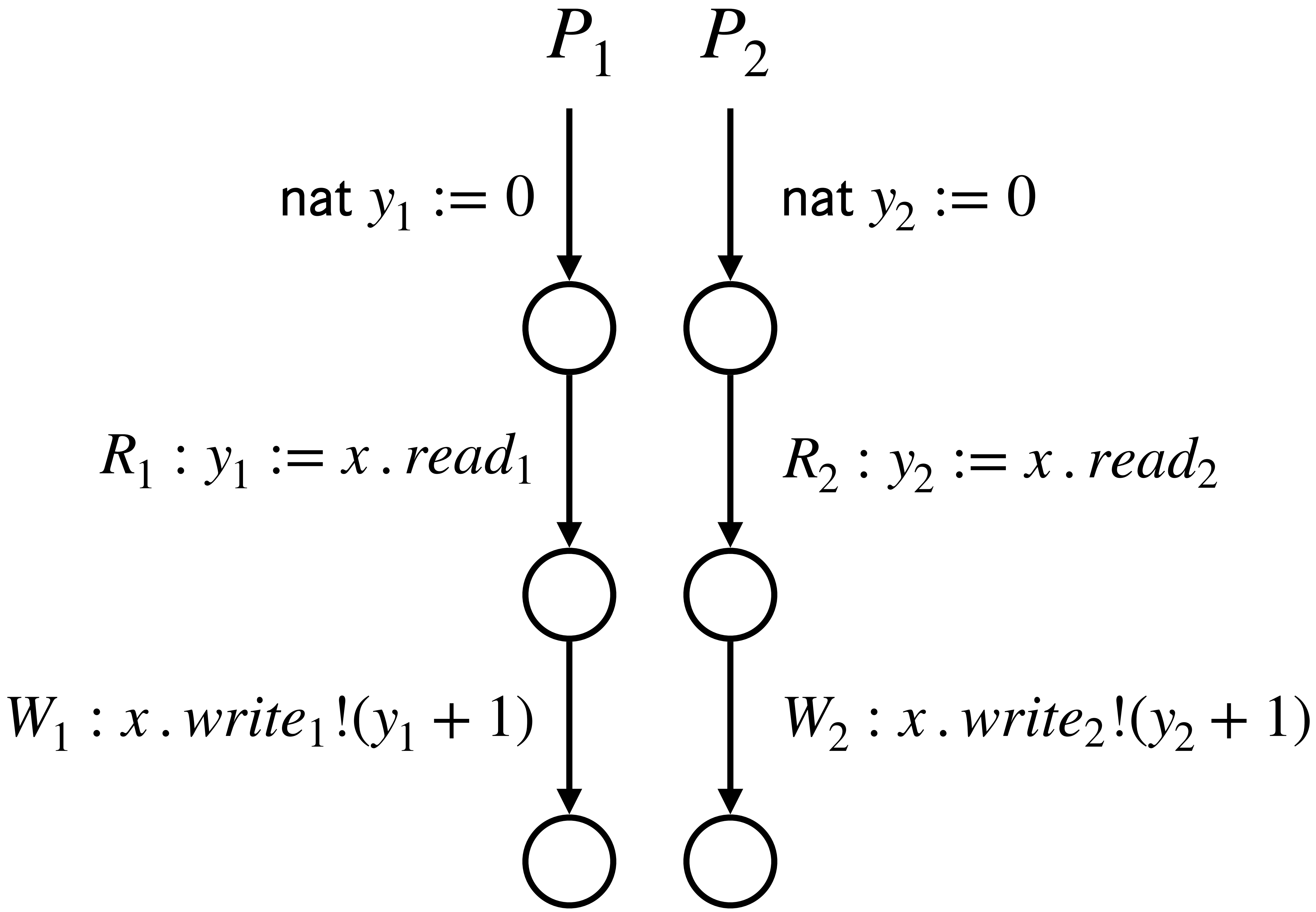
deadlock!

Atomic Register



Data Races

AtomicReg[nat] $x := 0$



Interleaving	x	y_1	y_2
$R_1; R_2; W_1; W_2$	1	0	0
$R_1; W_1; R_2; W_2$	2	0	1
$R_1; R_2; W_2; W_1$	1	0	0
$R_2; R_1; W_2; W_1$	1	0	0
$R_2; W_2; R_1; W_1$	2	1	0
$R_2; R_1; W_1; W_2$	1	0	0

Mutual Exclusion Problem

// shared variable
AtomicReg[int] $x := 0$

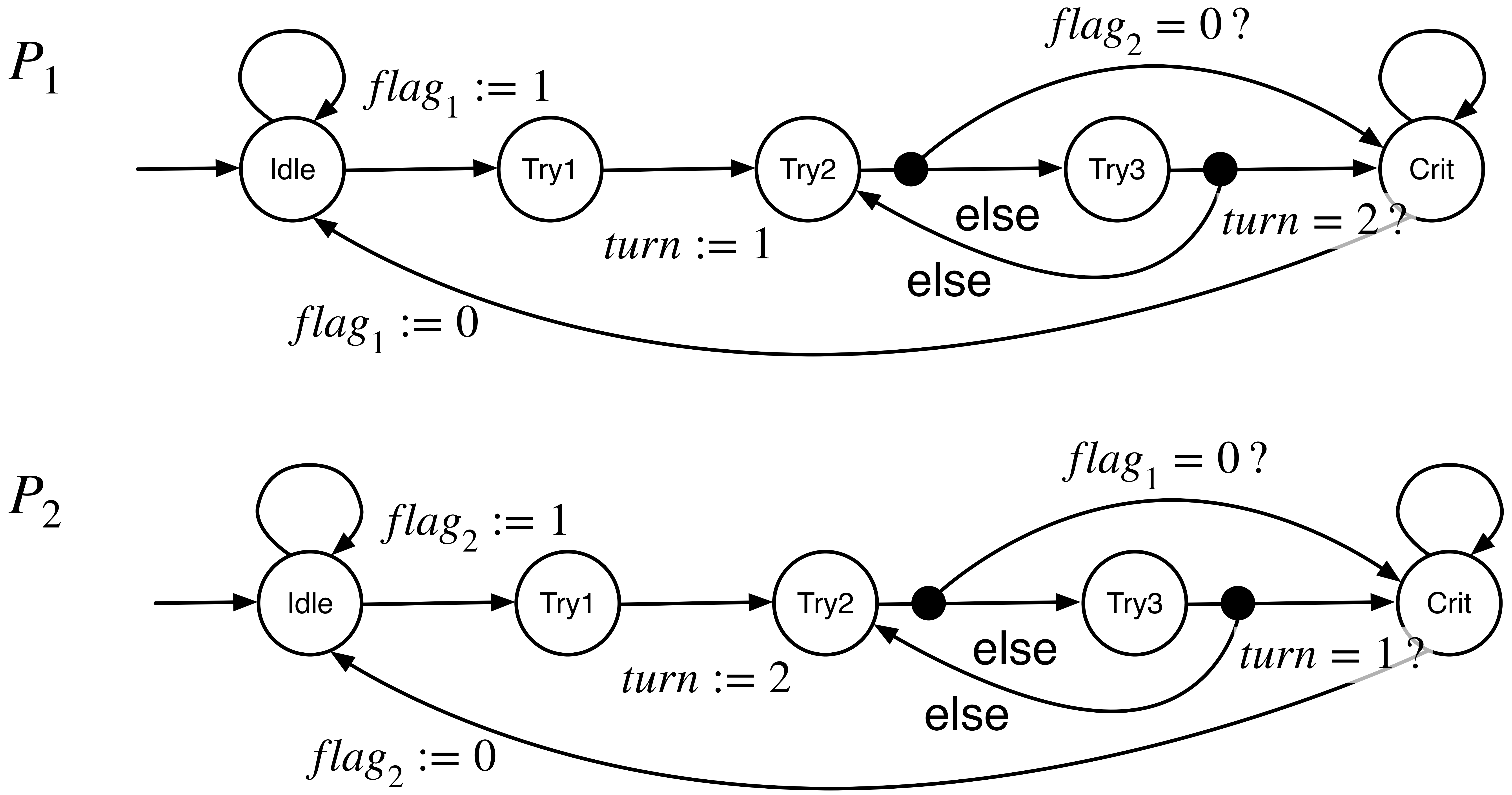
// P_1
EnterCS;
$y_1 := x.read_1;$
$x.write_1!(y_1 + 1);$
ExitCS

// P_2
EnterCS;
$y_2 := x.read_2;$
$x.write_2!(y_2 + 1);$
ExitCS

- Critical Section
  - A part of a program that have accesses to resources shared by two or more processes
- Mutual Exclusion
  - Safety: No two processes can enter the critical section at the same time.
  - Liveness: Once a process wants to enter the critical section, it should eventually be able to enter (aka freedom from deadlocks).

Peterson's Mutual Exclusion Protocol

AtomicReg[$bool$] $flag_1 := 0$; $flag_2 := 0$; AtomicReg[$\{1,2\}$] $turn$



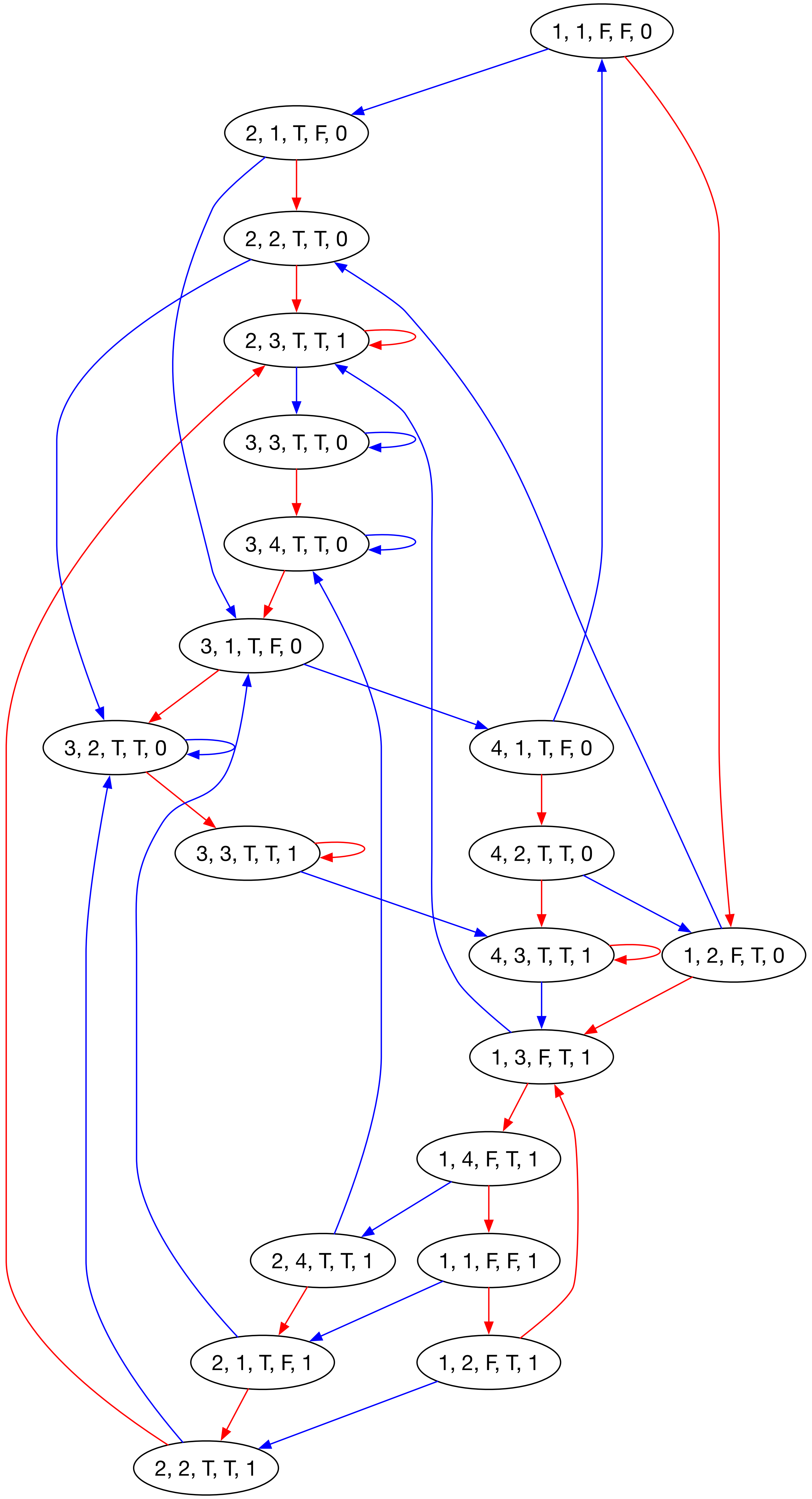
Safety Requirement

- Let $\rho = s_0, s_1, \ldots, s_k$ be a shortest execution such that the modes of both processes is Crit in s_k.
- In this case, the last step corresponds to a process (say P_1) switching to Crit. There may be two possible situations:
  - From Try2 to Crit (provided $flag_2 = 2$): In s_{k-1}, P_2 is already in Crit. Thus $flag_2 = 1.$ So this should not the case.
  - From Try3 to Crit (provided $turn = 2$): Suppose that $s_{j-1} \to s_j$ is the latest write to $turn$ by P_1 and $s_{l-1} \to s_l$ is the latest write to $turn$ by P_2. Since $turn = 2, j < l$. However, at s_l, $flag_1 = 1$. So P_2 should not have chances to enter Crit within s_l to s_{k-1}.

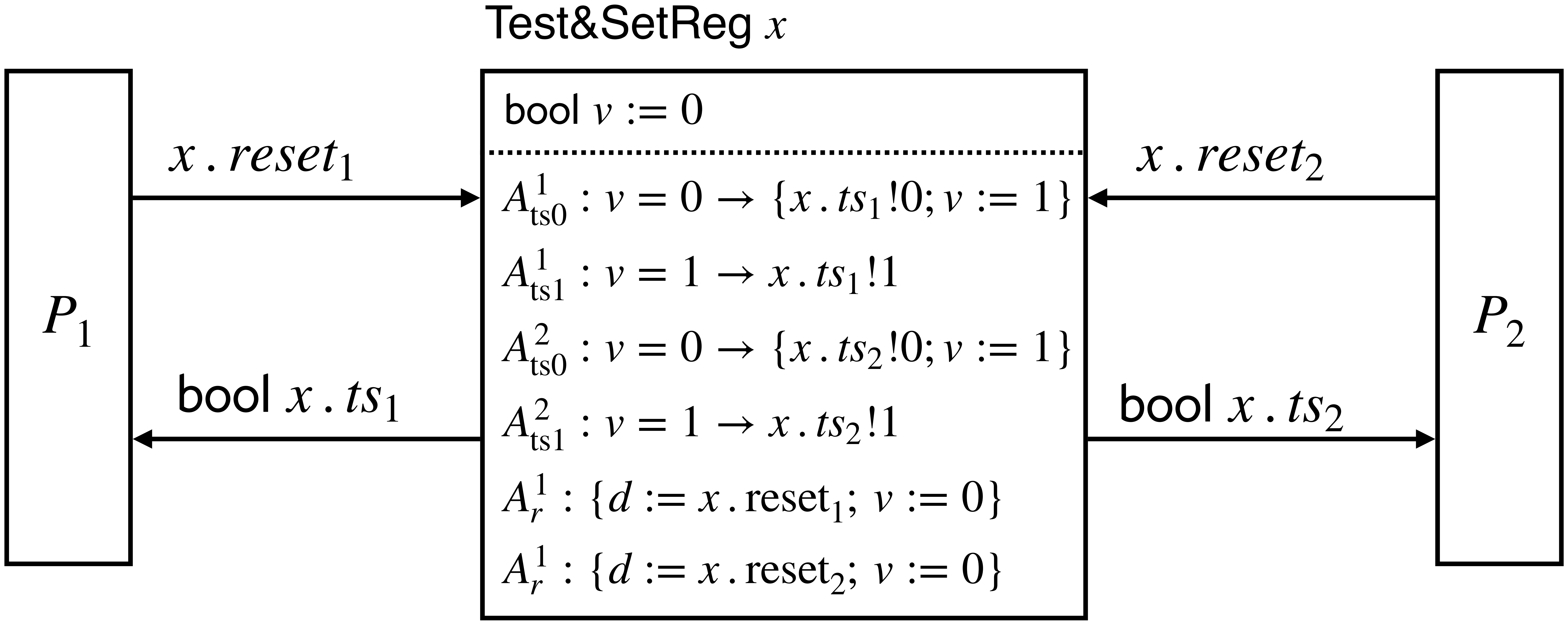
Peterson's Mutual Exclusion Protocol

```
// shared variables
bool flag1 = false, flag2 = false;
int turn;
```

```
// P1
while (true) {
  NC
  flag1 = true;
  turn = 0;
  while (flag2 && turn == 0);
  CS
  flag1 = false;
}
```

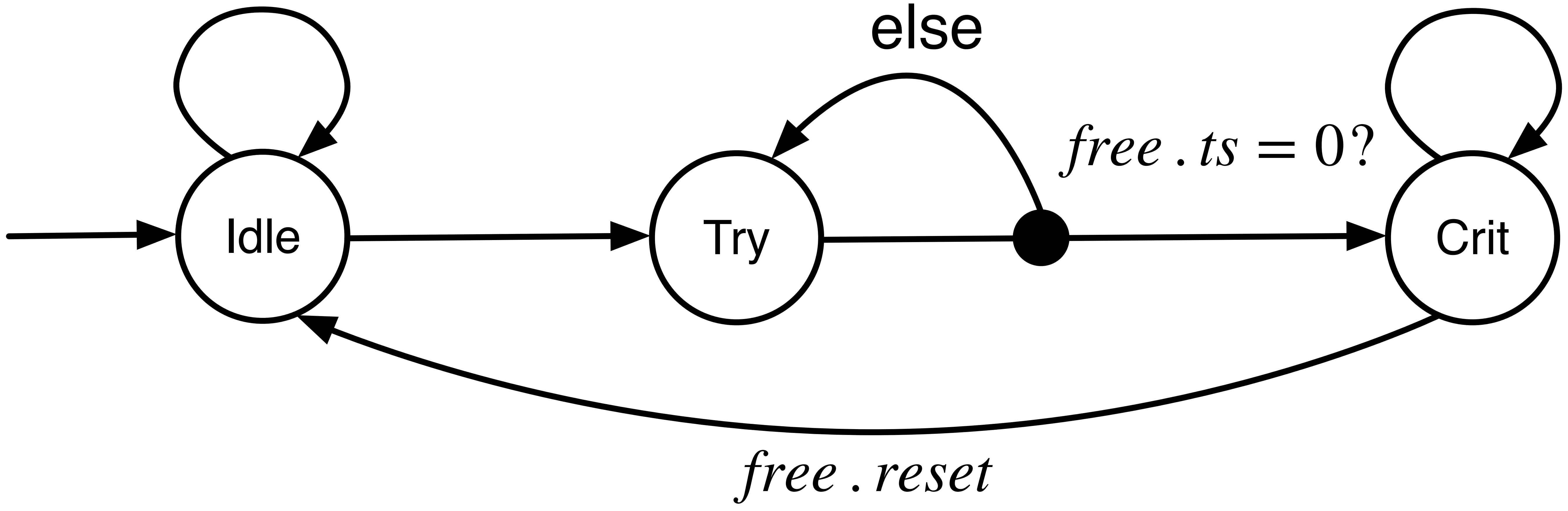


Test&Set Register



Mutual Exclusion using Test&SetRegister

Test&SetReg $free := 0$



Test-and-Set Instruction

```
// pseudo code for test_and_set
atomic bool test_and_set(bool *var) {
    bool tmp = *var;
    *var = true;
    return tmp;
}
```

```
// shared variable
bool in_use = false;
```

```
// P1 & P2
while (true) {
  NC
  while (test_and_set(&in_use));
  CS
  in_use = false;
}
```

Exchange Instruction

```
// pseudo code for test_and_set
atomic bool exchange(bool *var, bool new) {
    bool tmp = *var;
    *var = new;
    return tmp;
}
```

```
// shared variable
bool in_use = false;
```

```
// P1 & P2
while (true) {
  NC
  while (exchange(&in_use, true));
  CS
  in_use = false;
}
```

Cf. XCHG (x86)

Summary

- Asynchronous Model (2)