Concurrent Algorithms 2018

Midterm Exam Solutions

Problem 1

- **Task:** Write an algorithm that implements a MRSW atomic M-valued register using (any number of) SRSW regular M-valued registers.

- **Solution:**

  SRSW regular M-valued → SRSW atomic M-valued → MRSW atomic M-valued

  (see lecture slides)

Problem 2 – `register-swap`

- **Task:** Write an algorithm that implements wait-free consensus for n processes in this setting.

Variables:
Shared MWMR atomic registers A and B.

```
procedure register-swap(A, B)
    tempA = A
    tempB = B
    A = tempB
    B = tempA
```

Problem 2 – register-swap

- R[1, …, N] = {⊥, …, ⊥}
- Winner[1, …. N] = {⊥, …, ⊥}
- Decided = won

```
procedure propose(v)
   R[i] = v
   register-swap(Winner[i], Decider)
   j = unique index in Winner with Winner[j] = won
   return R[j]
```

Problem 2 – `register-swap`

- `R[1, …, N] = {⊥, …, ⊥}`
- `Winner[1, …. N] = {⊥, …, ⊥}`
- `Decided = won`

First processes that does the swap “wins” the consensus

```
procedure propose(v)
   R[i] = v
   register-swap(Winner[i], Decider)
   j = unique index in Winner with Winner[j] = won
   return R[j]
```

Problem 3 – `test-and-set`

Variables:

`V=0` (binary register)

```
procedure test-and-set()
    temp = V
    if temp = 0 then
        V = 1
     return temp
```

Problem 3 – `test-and-set`

```
R[2] = {⊥, ⊥}
X // test-and-set object

procedure propose_i(v) //  i in {0, 1}
    R[i] = v
    result = x.test-and-set()
    if (result == 0)
        return R[i]
    else
        return R[1 – i]
```

`test-and-set` solves consensus for 2 processes.

Problem 3 – `test-and-set`

```
R[2] = {⊥, ⊥}
X // test-and-set object

procedure propose_i(v) //  i in {0, 1}
    R[i] = v
    result = x.test-and-set()
    if (result == 0)
        return R[i]
    else
        return R[1 – i]
```

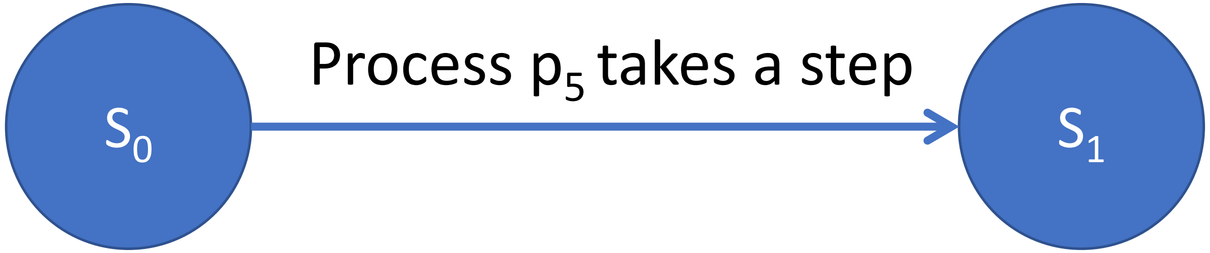
`test-and-set` solves consensus for 2 processes.

But **not** for **3 processes**.

Problem 3 – `test-and-set`

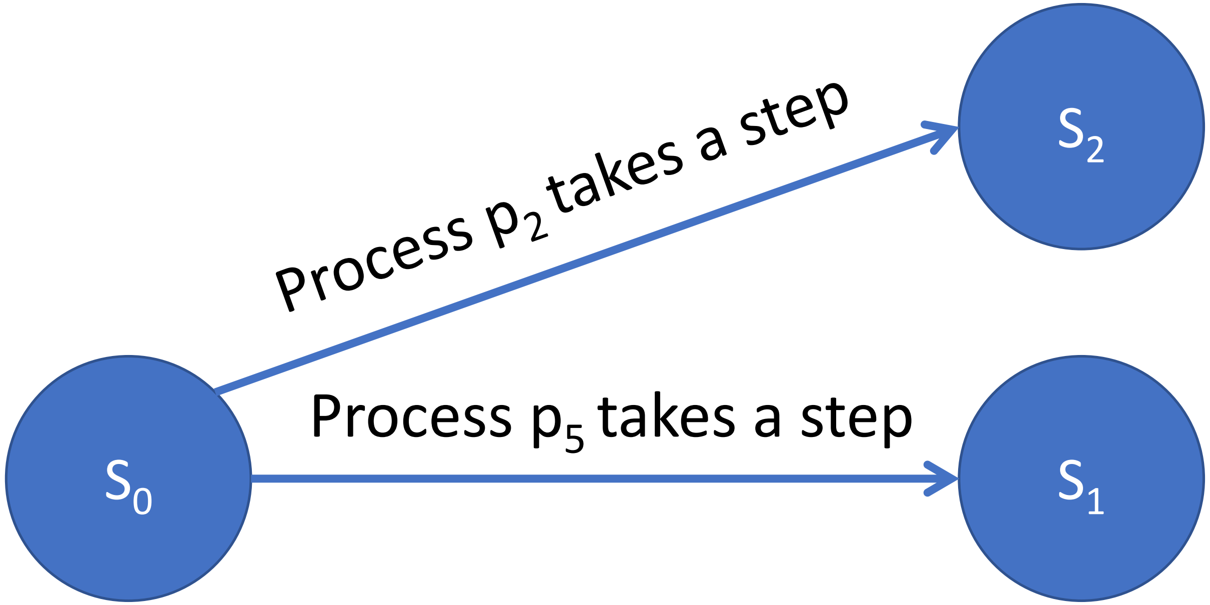
State: state of all the processes and of the shared objects

Problem 3 – `test-and-set`

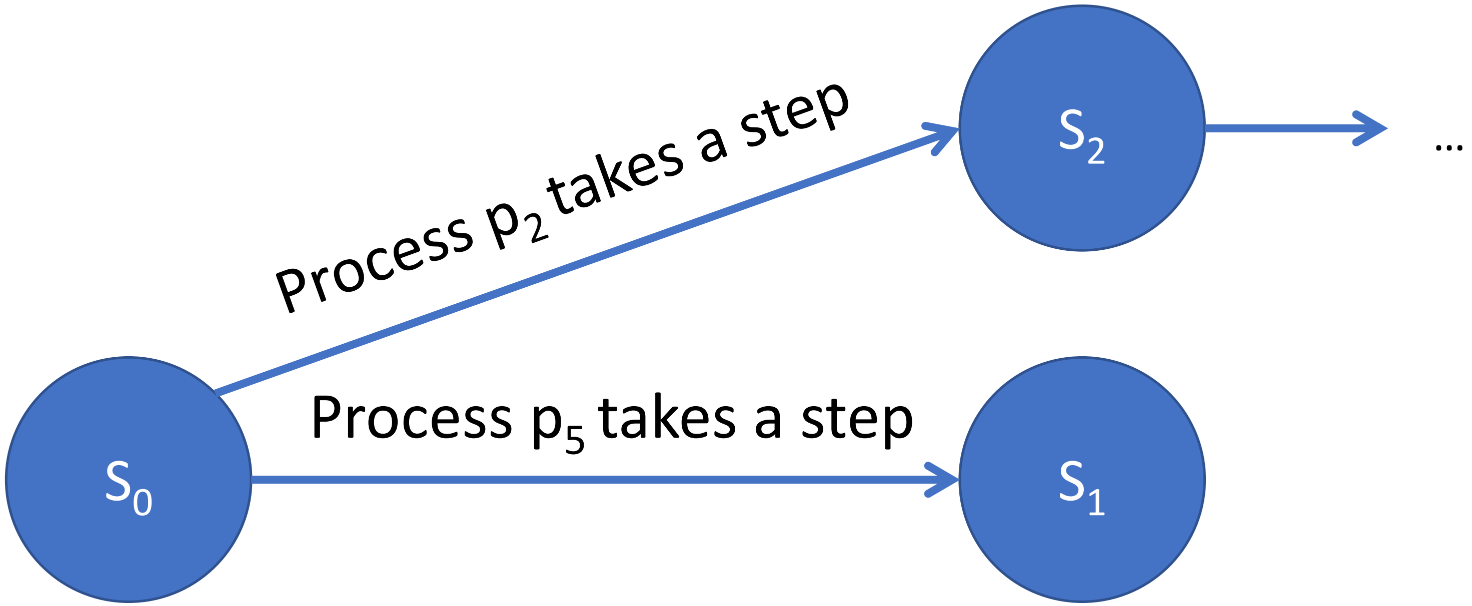


A **step** corresponds to the access (**read** or modify) of some shared object.

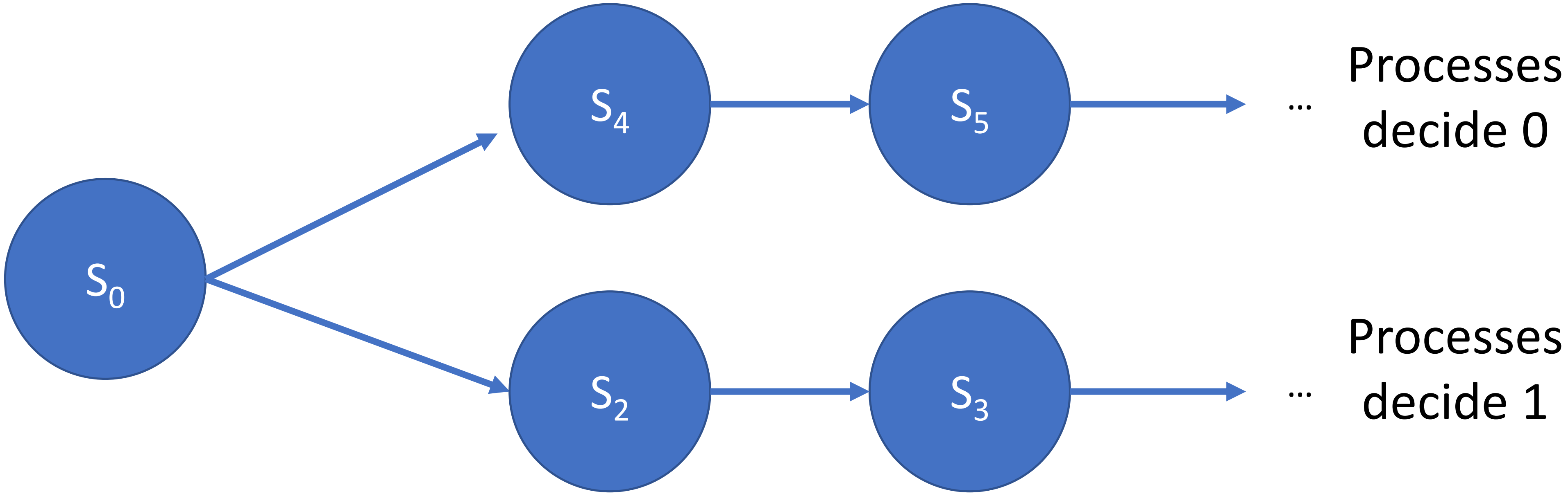
Problem 3 – `test-and-set`



Problem 3 – `test-and-set`

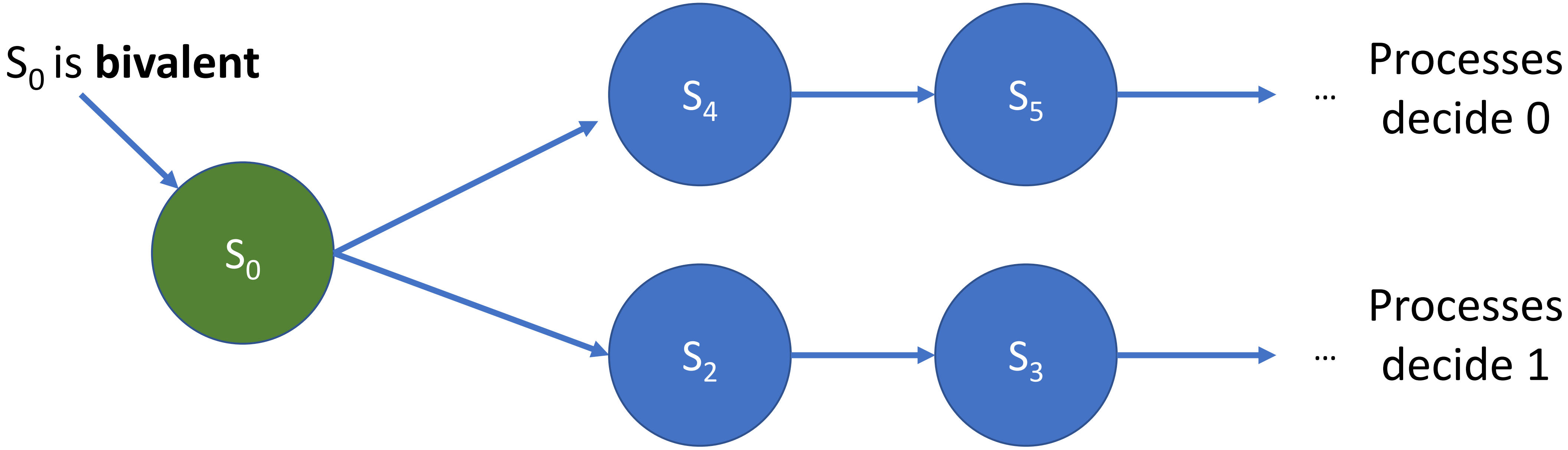


Problem 3 – `test-and-set`



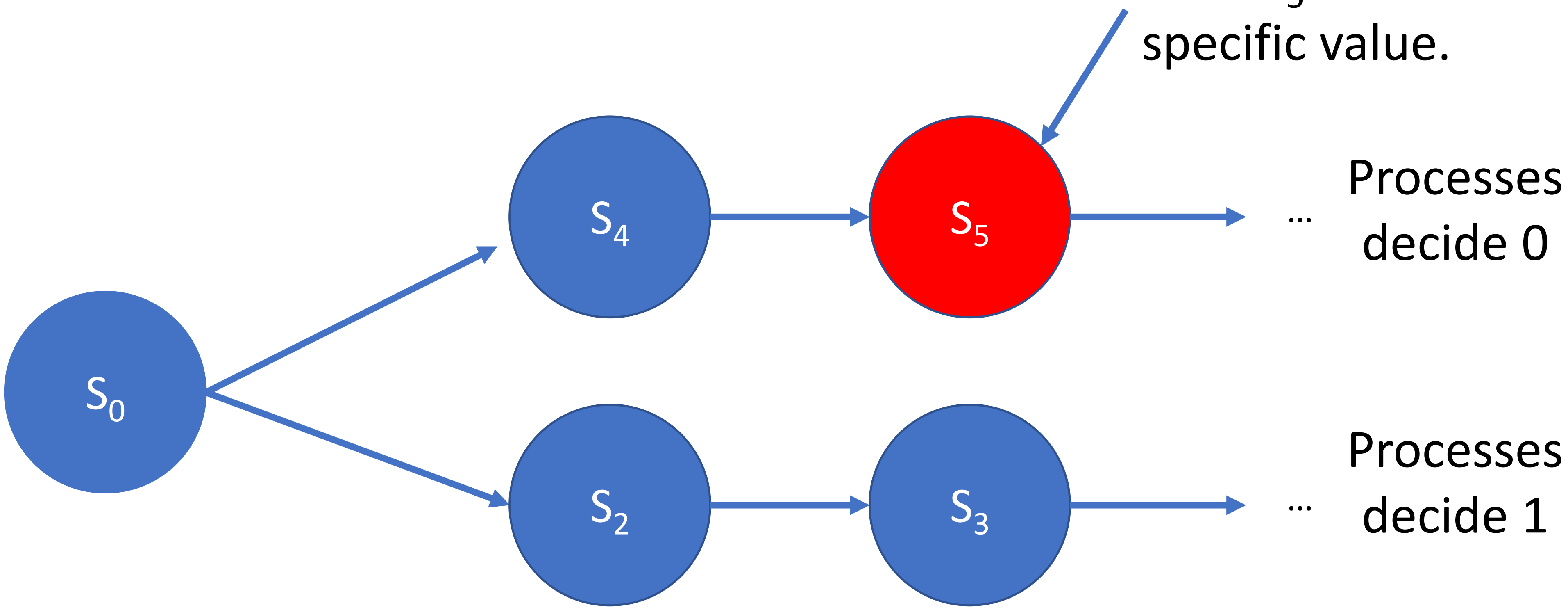
A state is **bivalent** if the decision is **not** yet fixed.
Processes could decide 0 or 1.

Problem 3 – `test-and-set`



A state is **bivalent** if the decision is **not** yet fixed.
Processes could decide 0 or 1.

Problem 3 – `test-and-set`



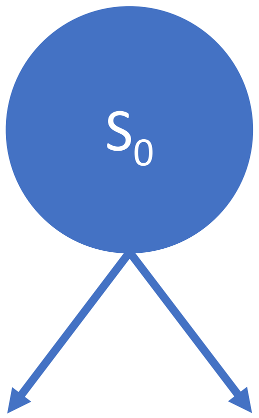
A state is **univalent** if the decision is fixed.

Problem 3 – `test-and-set`

p_0 `propose(0)` | p_1 `propose(1)`

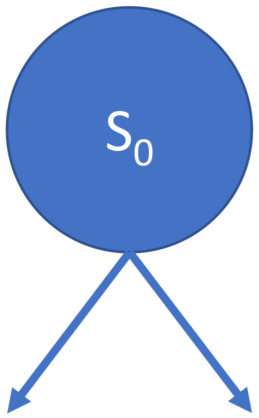
solo run

will decide 0.



Problem 3 – `test-and-set`

p_0 `propose(0)`	p_1 `propose(1)`
	solo run
	will decide 1.

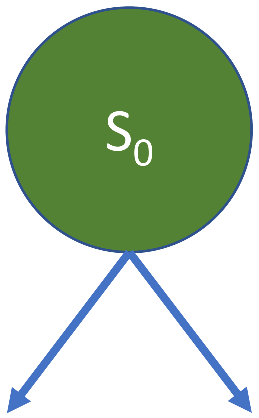


Problem 3 – `test-and-set`

p_0 `propose(0)` | p_1 `propose(1)`

solo run

will decide 1.



Every consensus algorithm has an **initial bivalent state.**

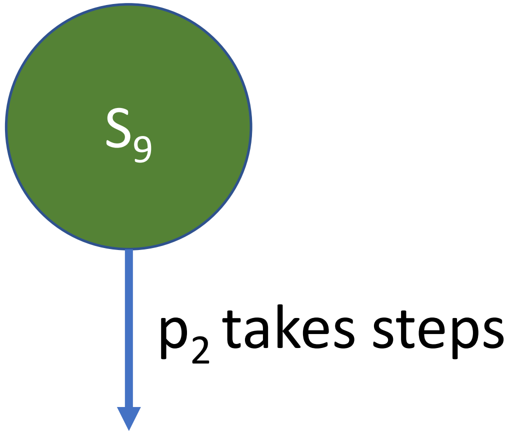
Problem 3 – `test-and-set`

Every consensus algorithm has a state that:

- **is bivalent;**
- **if any process takes a step, the new state is univalent.**

Also known as a **critical state**.

Problem 3 – `test-and-set`



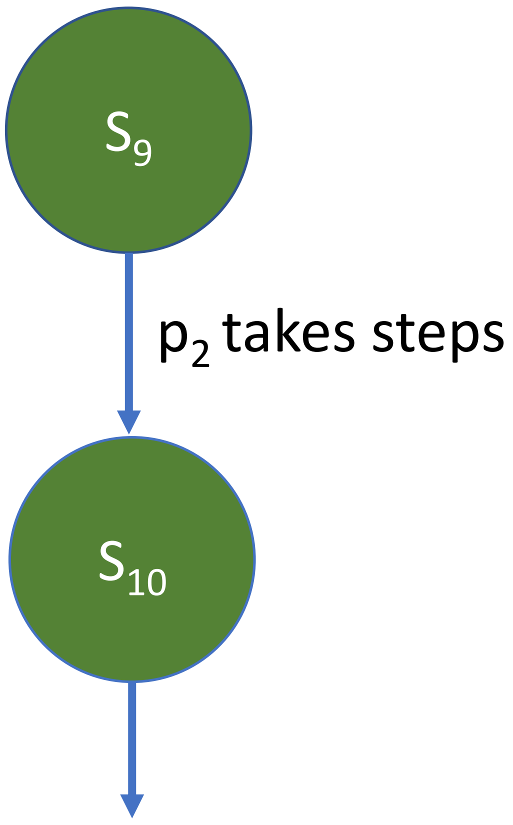
Every consensus algorithm has a state that:

- **is bivalent;**
- **if any process takes a step, the new state is univalent.**

Also known as a **critical state**.

Suppose **not**. As long as a process can take steps without reaching a univalent state, let that process take steps.

Problem 3 – `test-and-set`



Every consensus algorithm has a state that:

- **is bivalent;**
- **if any process takes a step, the new state is univalent.**

Also known as a **critical state**.

Suppose **not**. As long as a process can take steps without reaching a univalent state, let that process take steps.

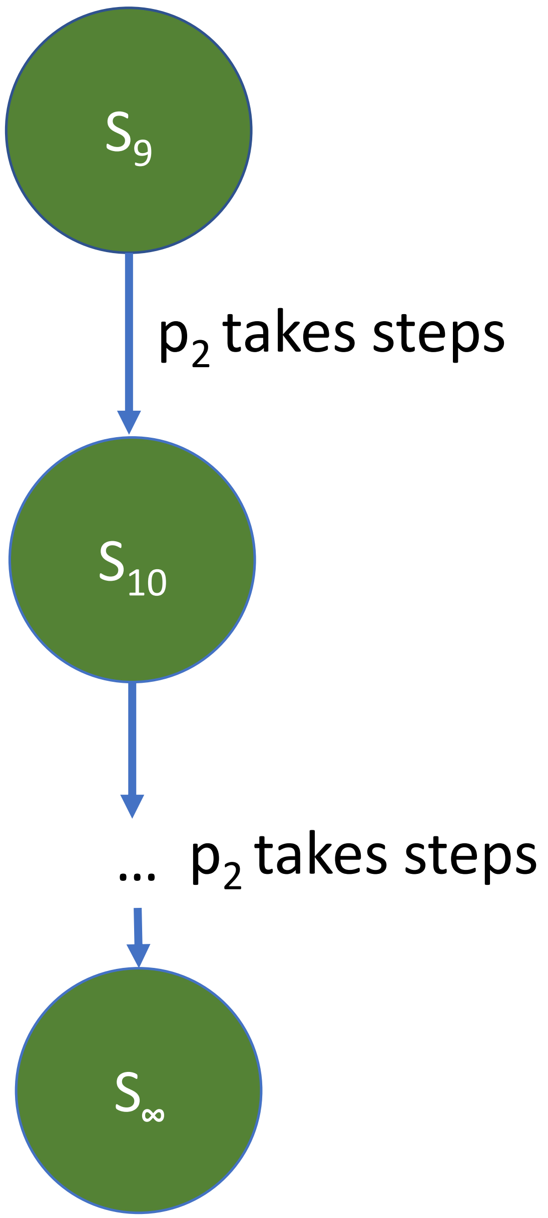
Problem 3 – `test-and-set`

Every consensus algorithm has a state that:

- **is bivalent;**
- **if any process takes a step, the new state is univalent.**

Also known as a **critical state**.

Suppose **not**. As long as a process can take steps without reaching a univalent state, let that process take steps.



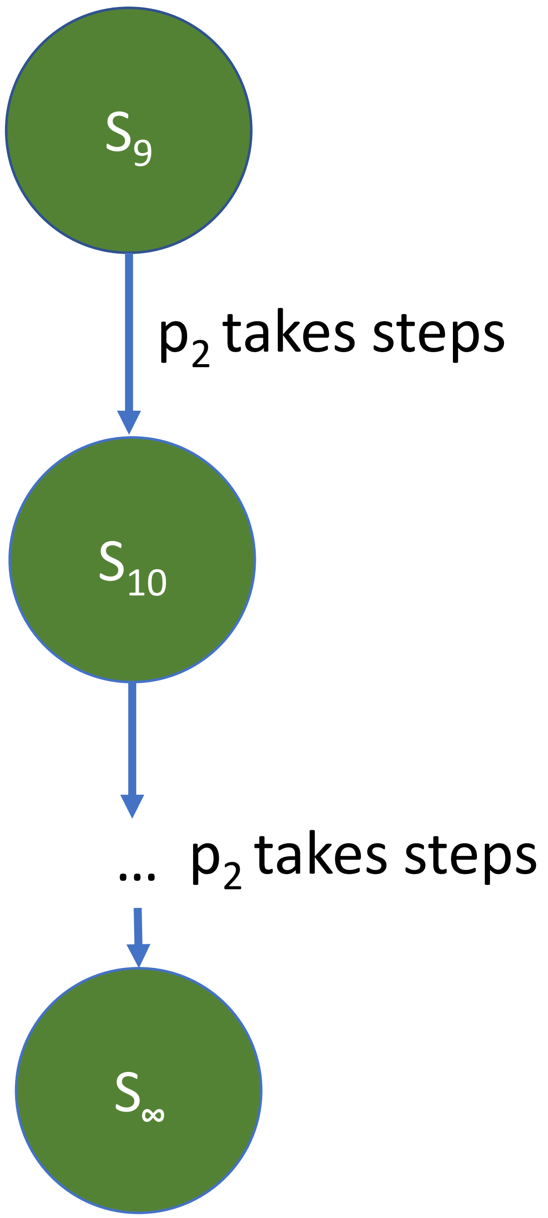
Problem 3 – `test-and-set`

Every consensus algorithm has a state that:

- **is bivalent;**
- **if any process takes a step, the new state is univalent.**

Also known as a **critical state**.

Suppose **not**. As long as a process can take steps without reaching a univalent state, let that process take steps.

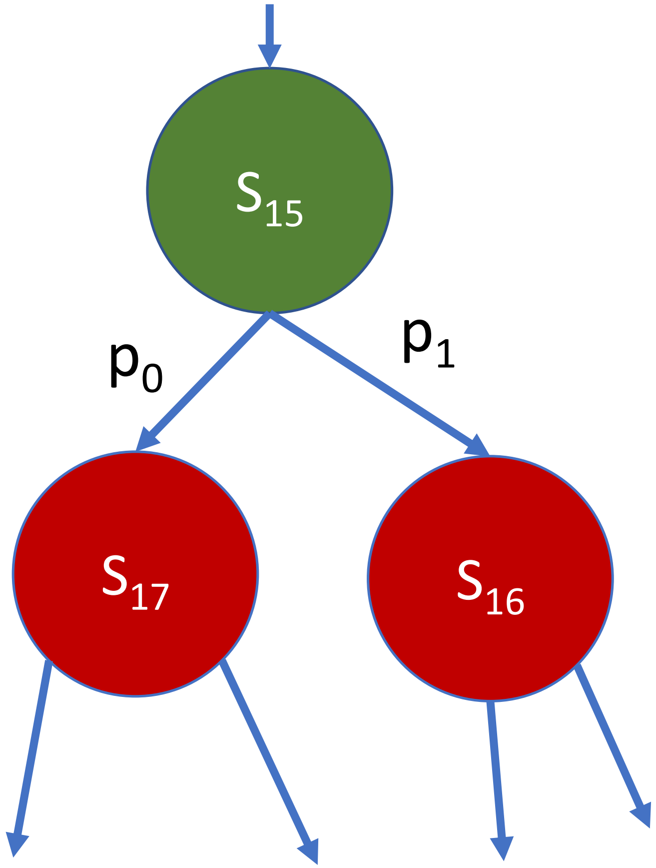


not wait-free

Problem 3 – `test-and-set`

S_{15} is a **critical state.**
In other words:

- S_{15} is bivalent
- Any process that takes a step reaches a univalent state

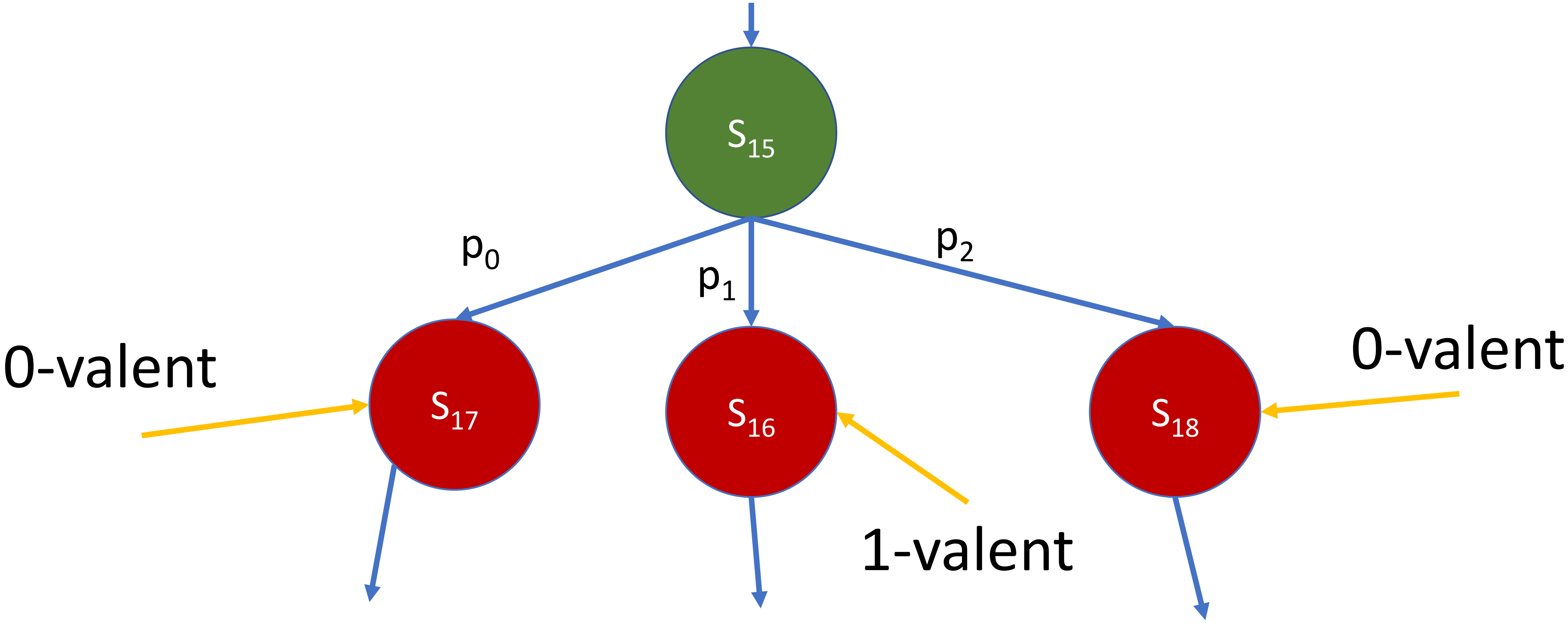


Problem 3 – `test-and-set`

Assume there is a consensus algorithm for 3 processes p_0, p_1, and p_2 that only uses read/write and test-and-set objects.

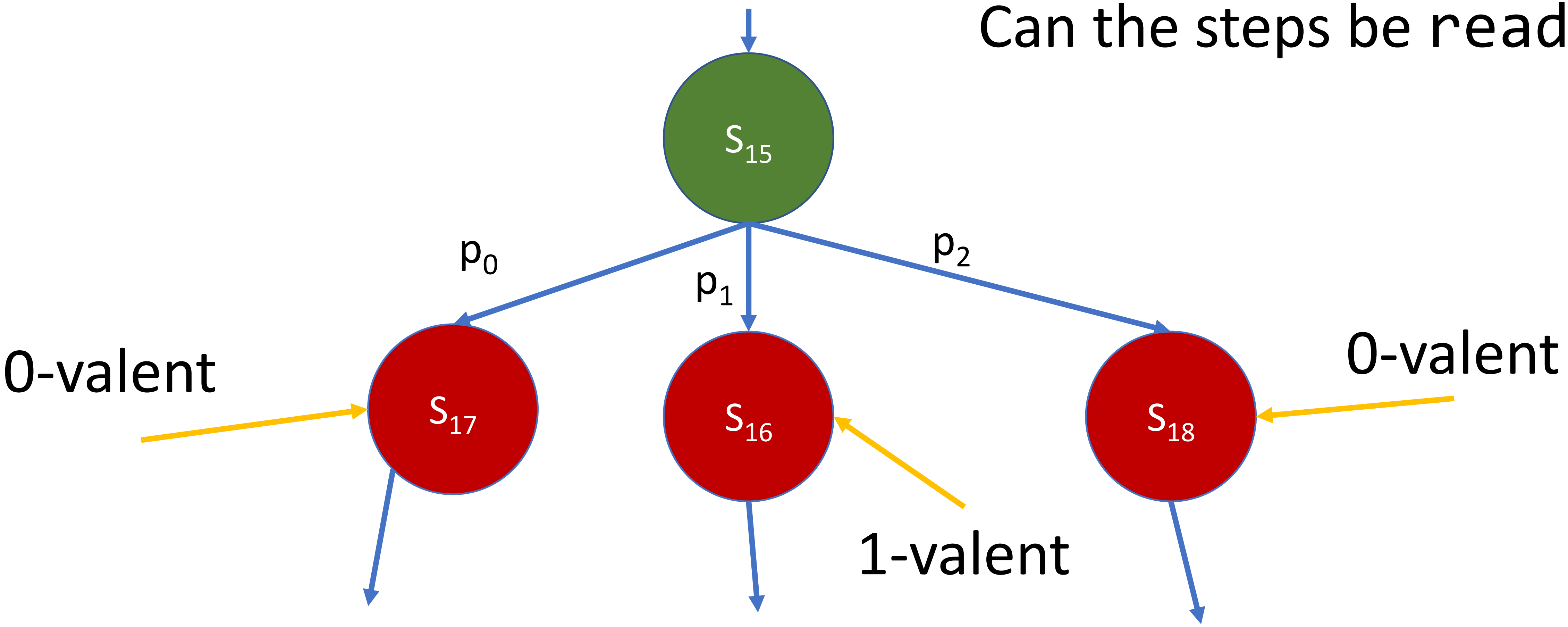
There should be a **critical state**.

Problem 3 – `test-and-set`



Problem 3 – `test-and-set`

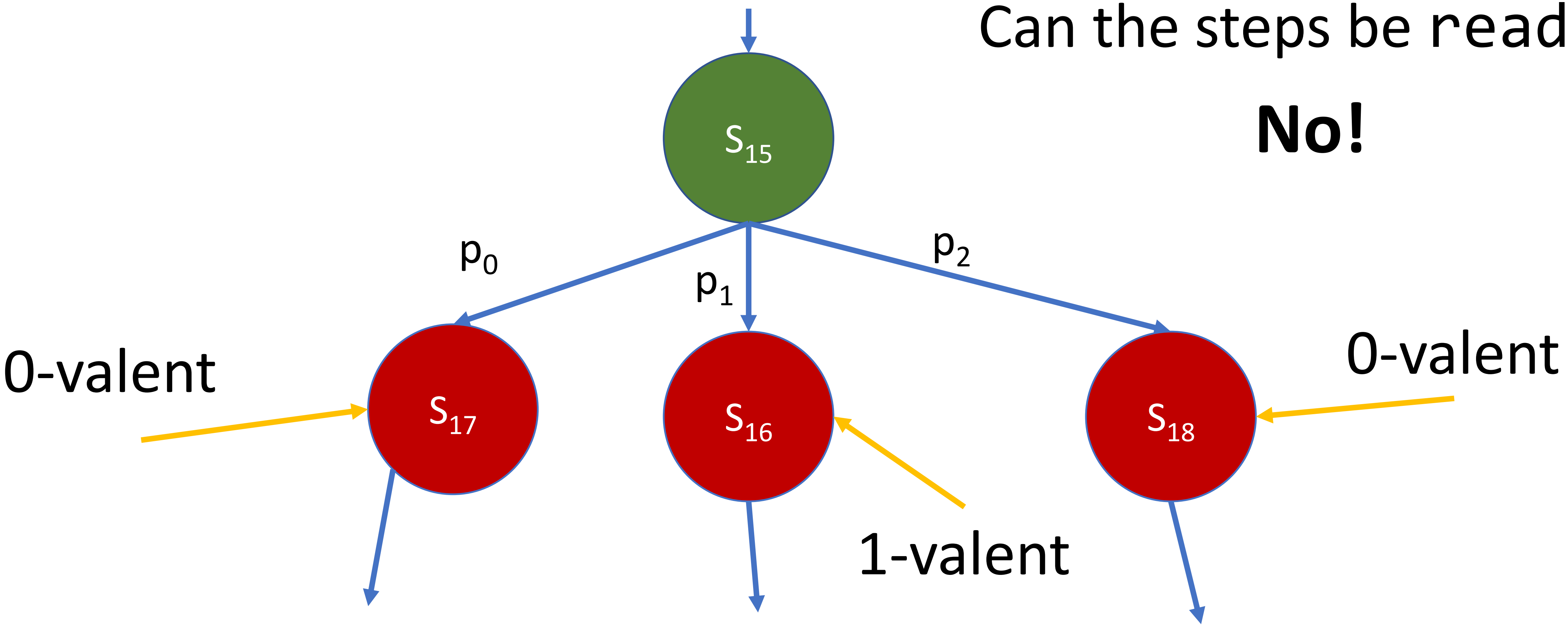
Can the steps be `reads`?



Problem 3 – `test-and-set`

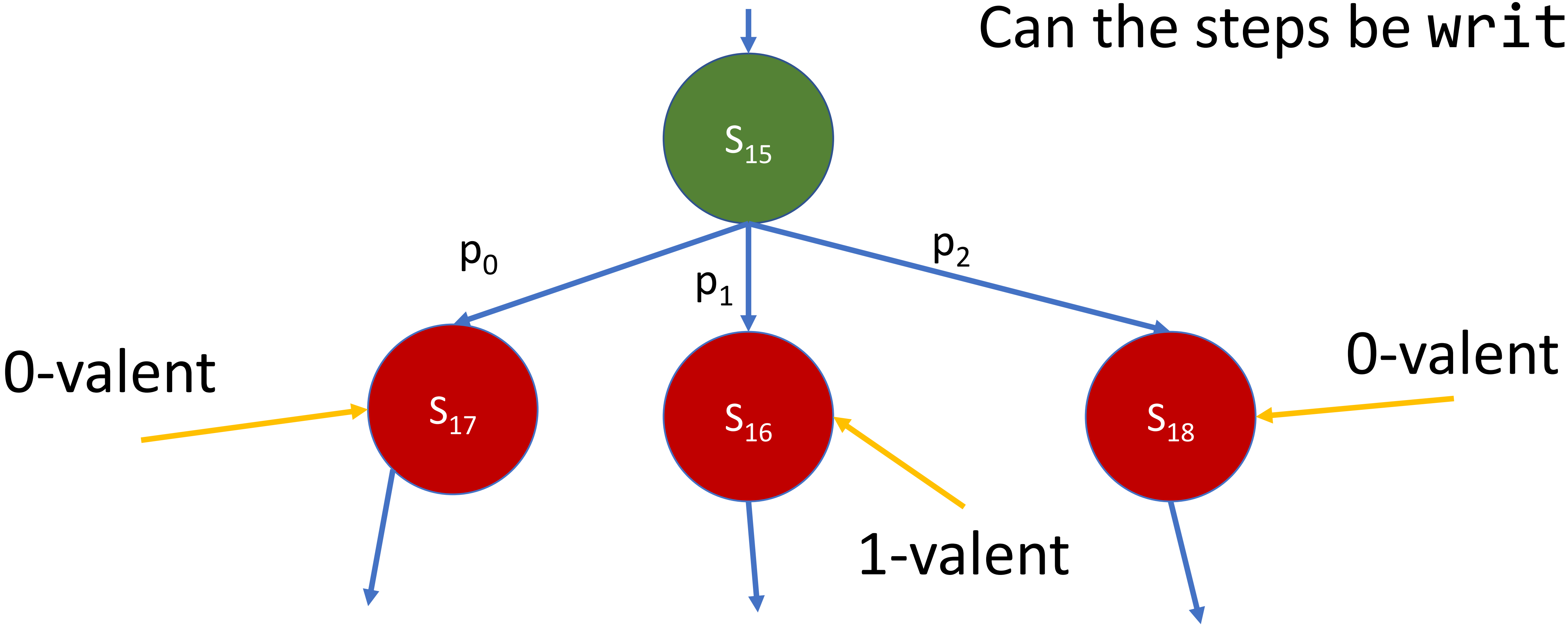
Can the steps be `reads`?

No!



Problem 3 – `test-and-set`

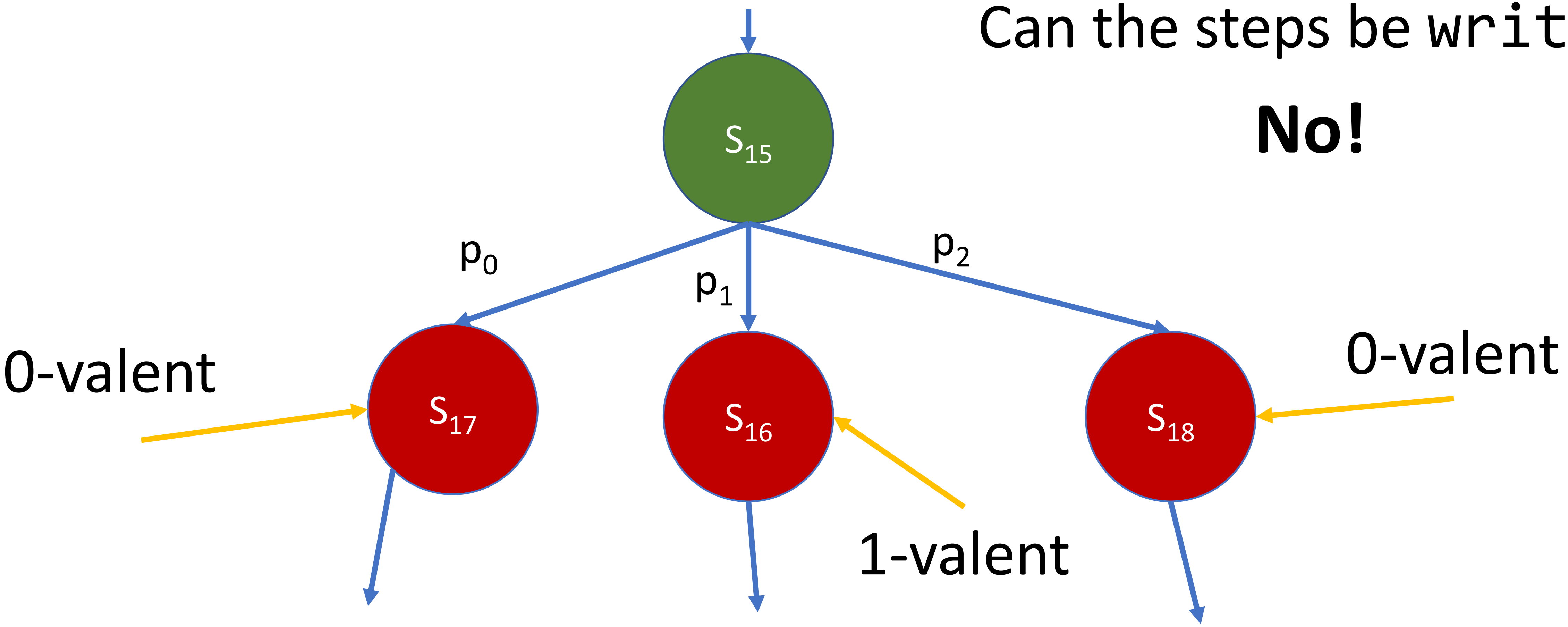
Can the steps be `writes`?



Problem 3 – `test-and-set`

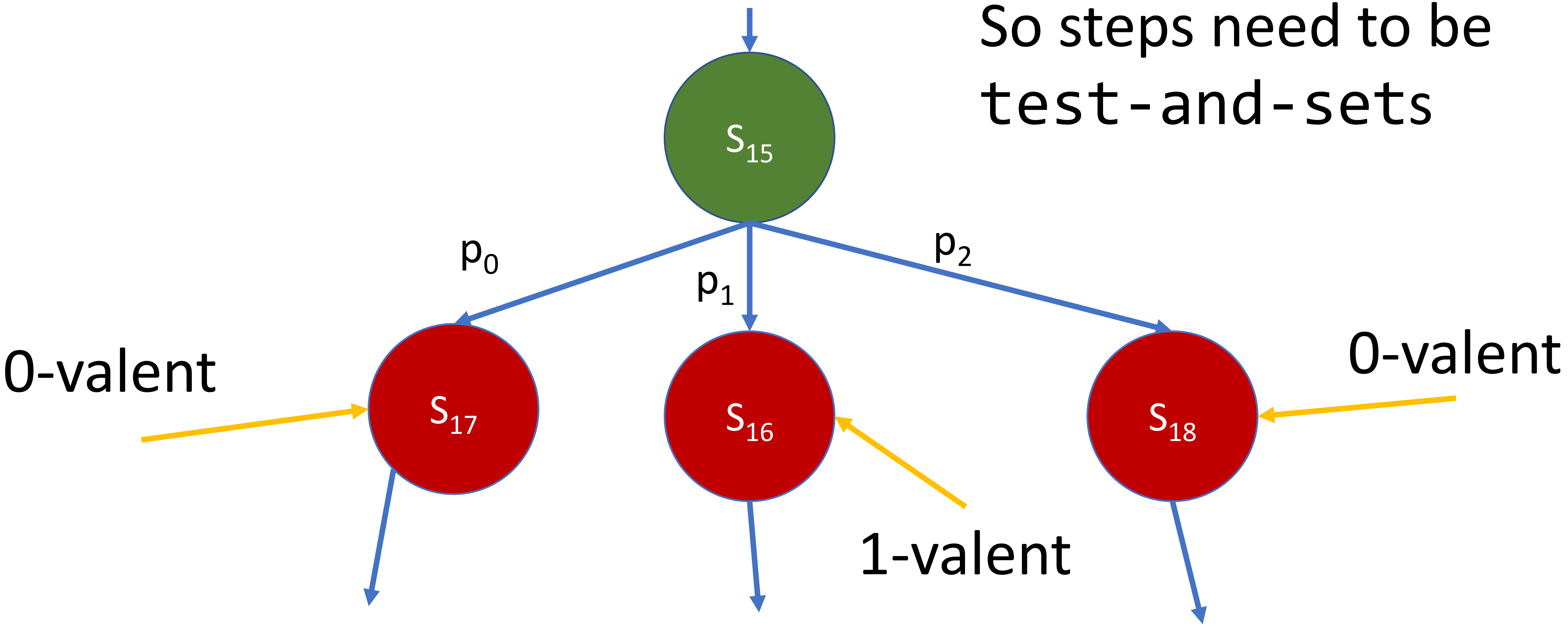
Can the steps be `writes`?

No!



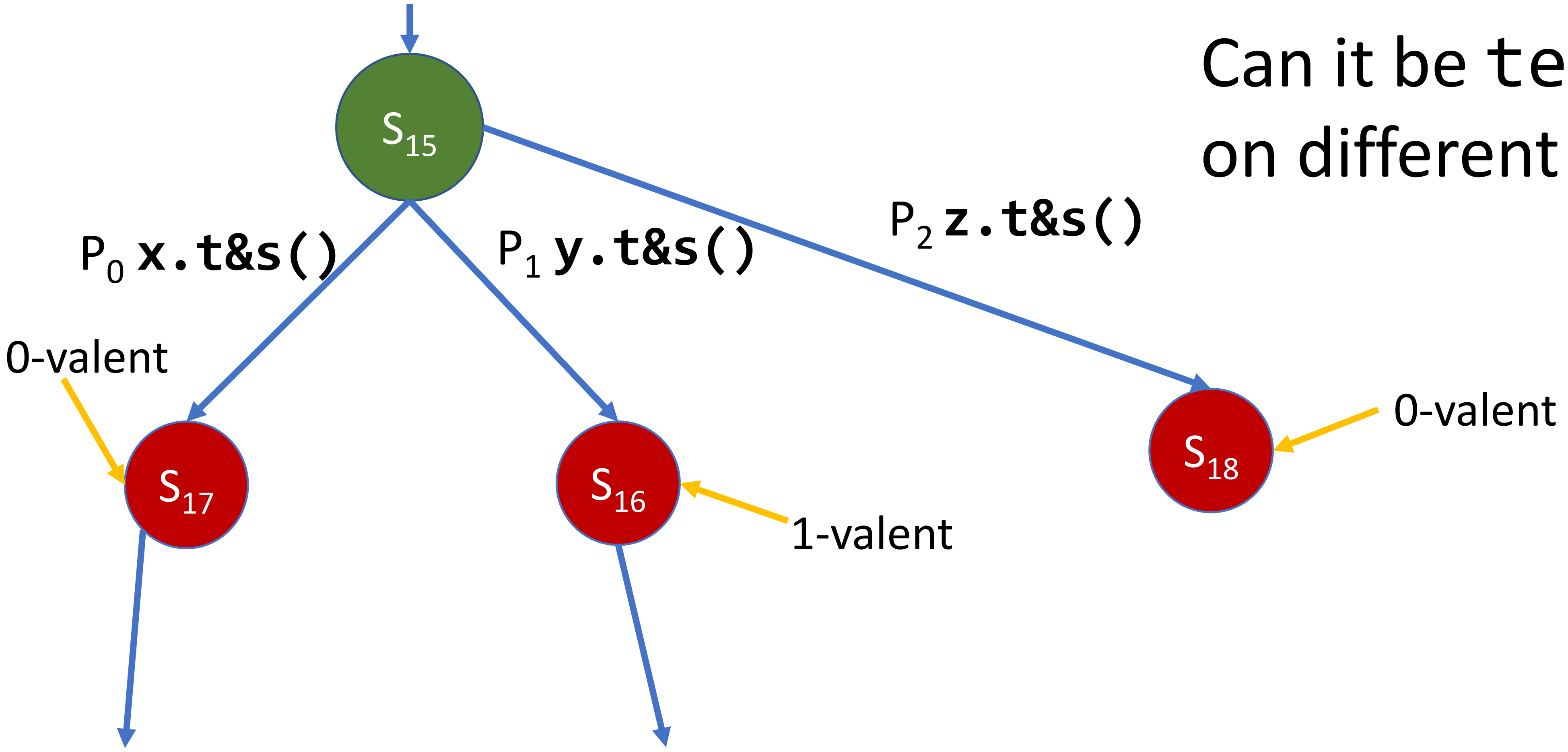
Problem 3 – `test-and-set`

So steps need to be `test-and-sets`

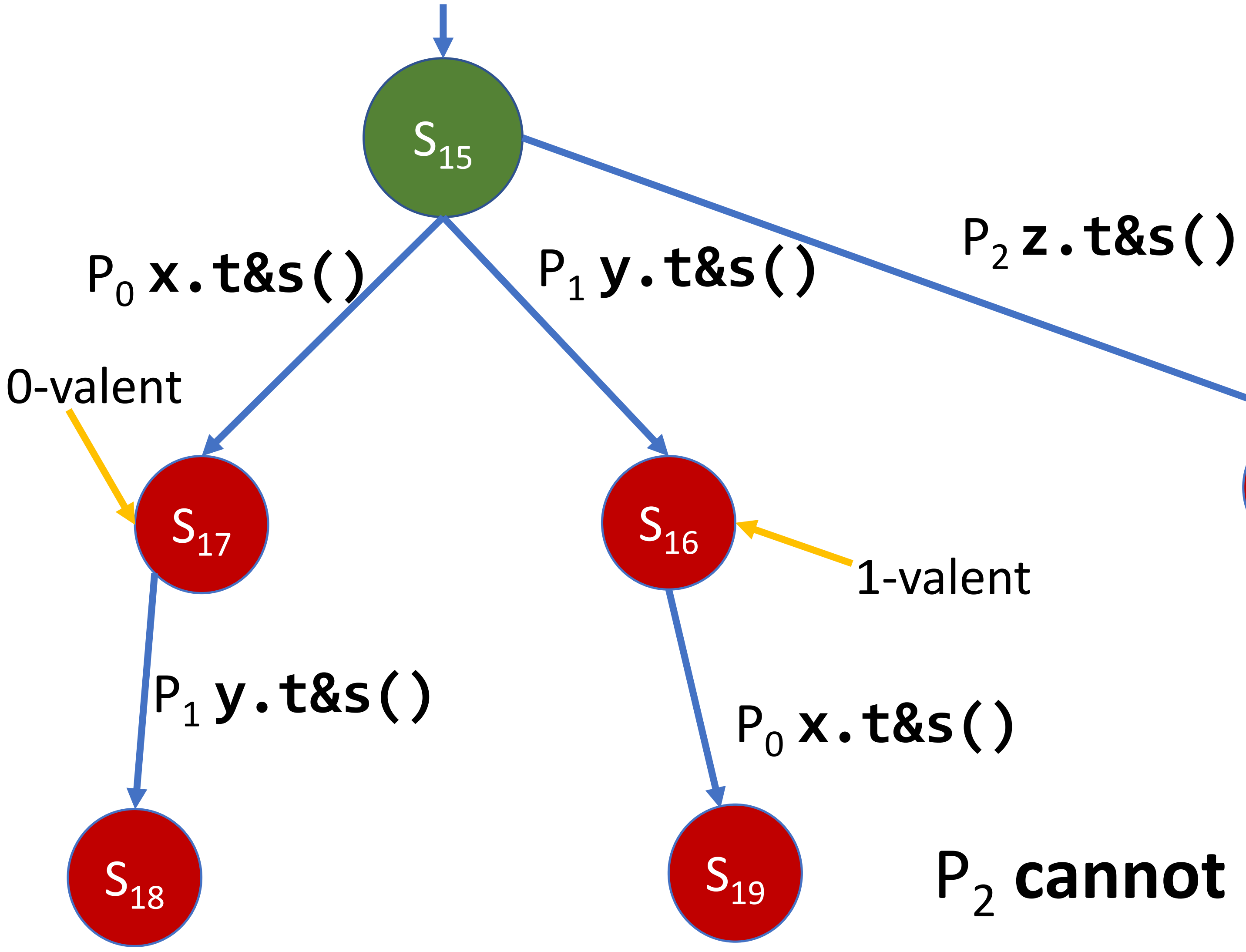


Problem 3 – `test-and-set`

Can it be `test-and-sets` on different objects?



Problem 3 – `test-and-set`



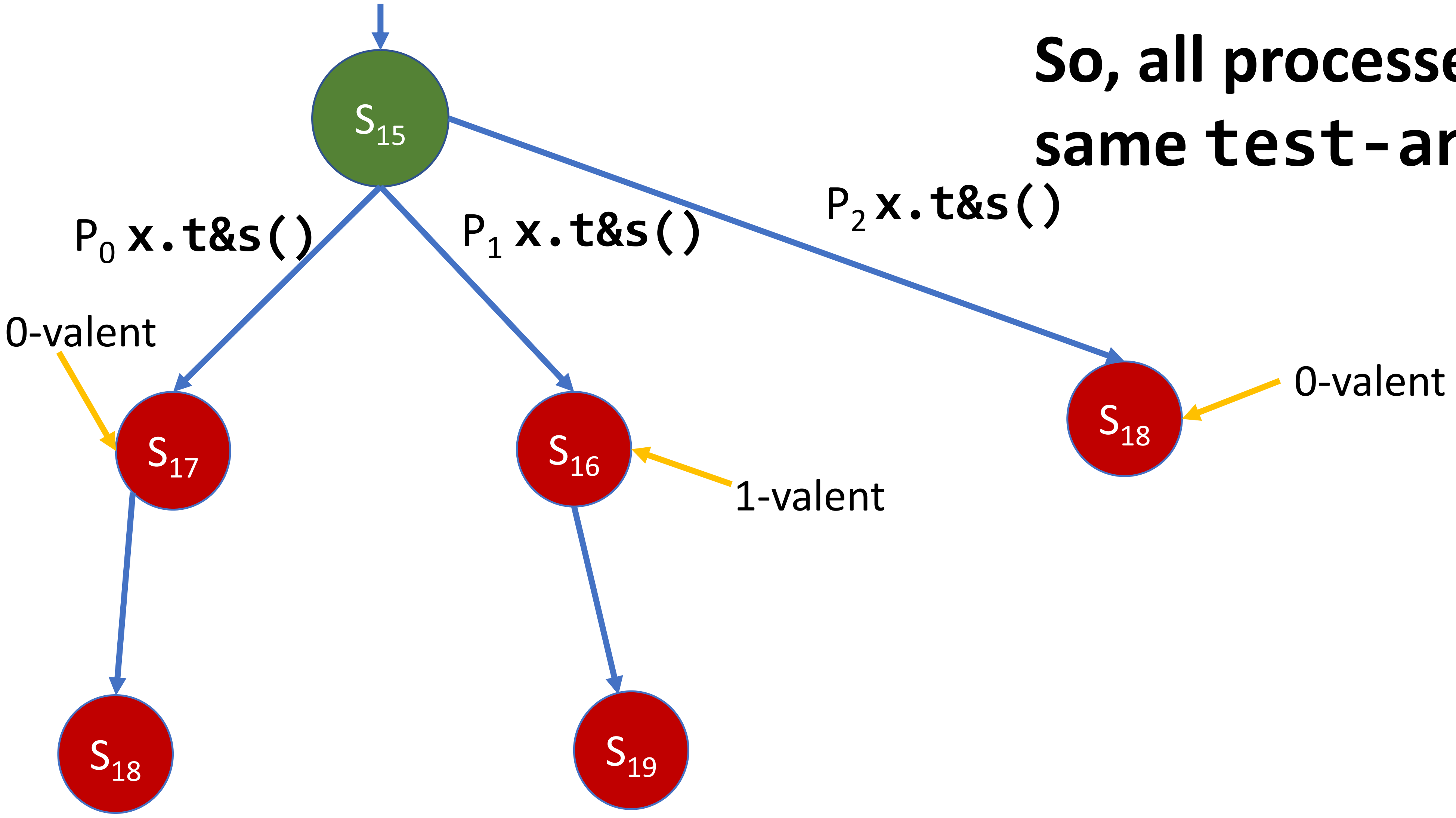
Can it be `test-and-sets` on different objects?

A **contradiction**.

P_2 **cannot** distinguish between S_{18} and S_{19}!

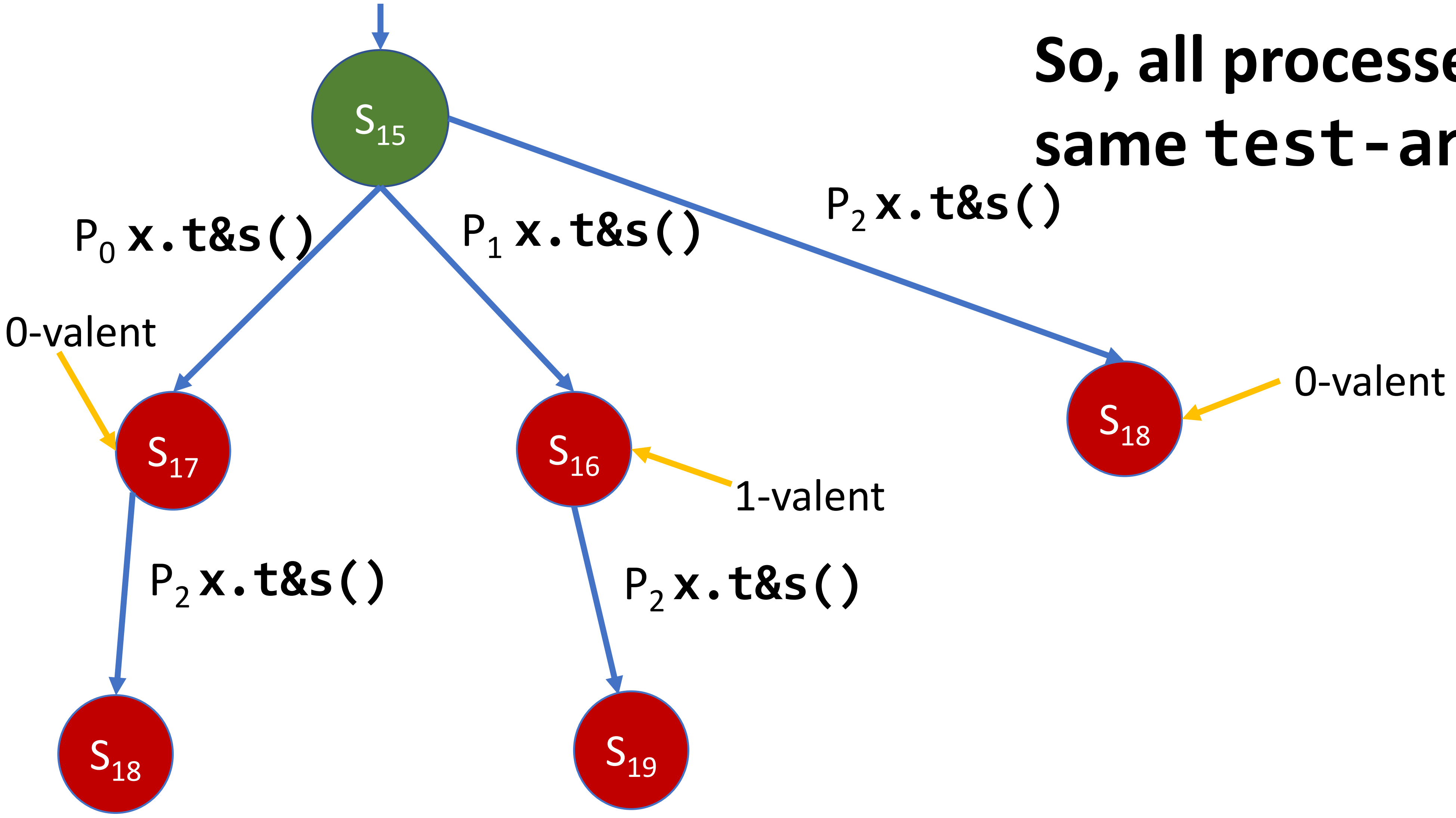
Problem 3 – `test-and-set`

So, all processes use the same `test-and-set` object.



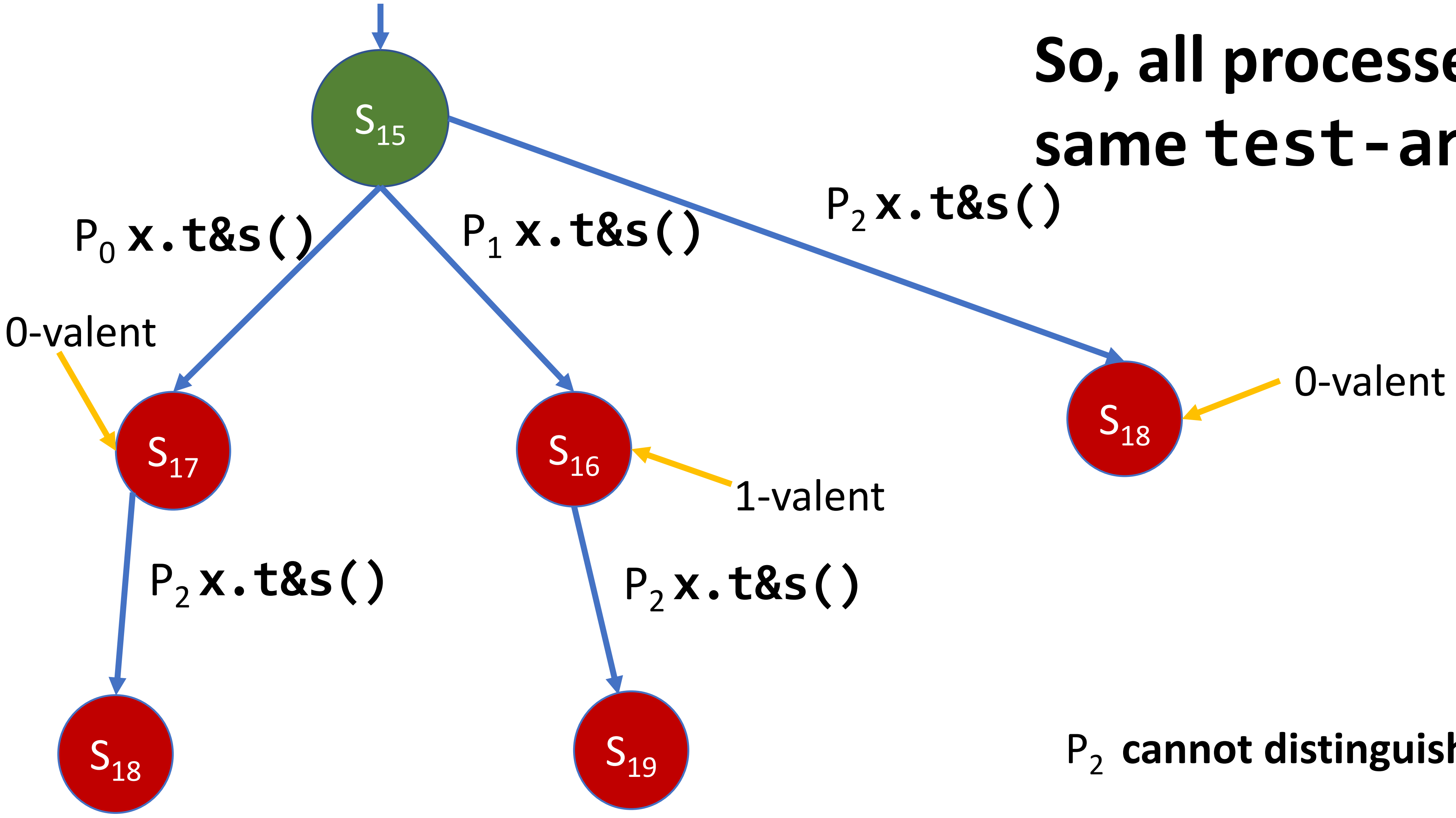
Problem 3 – `test-and-set`

So, all processes use the same `test-and-set` object.



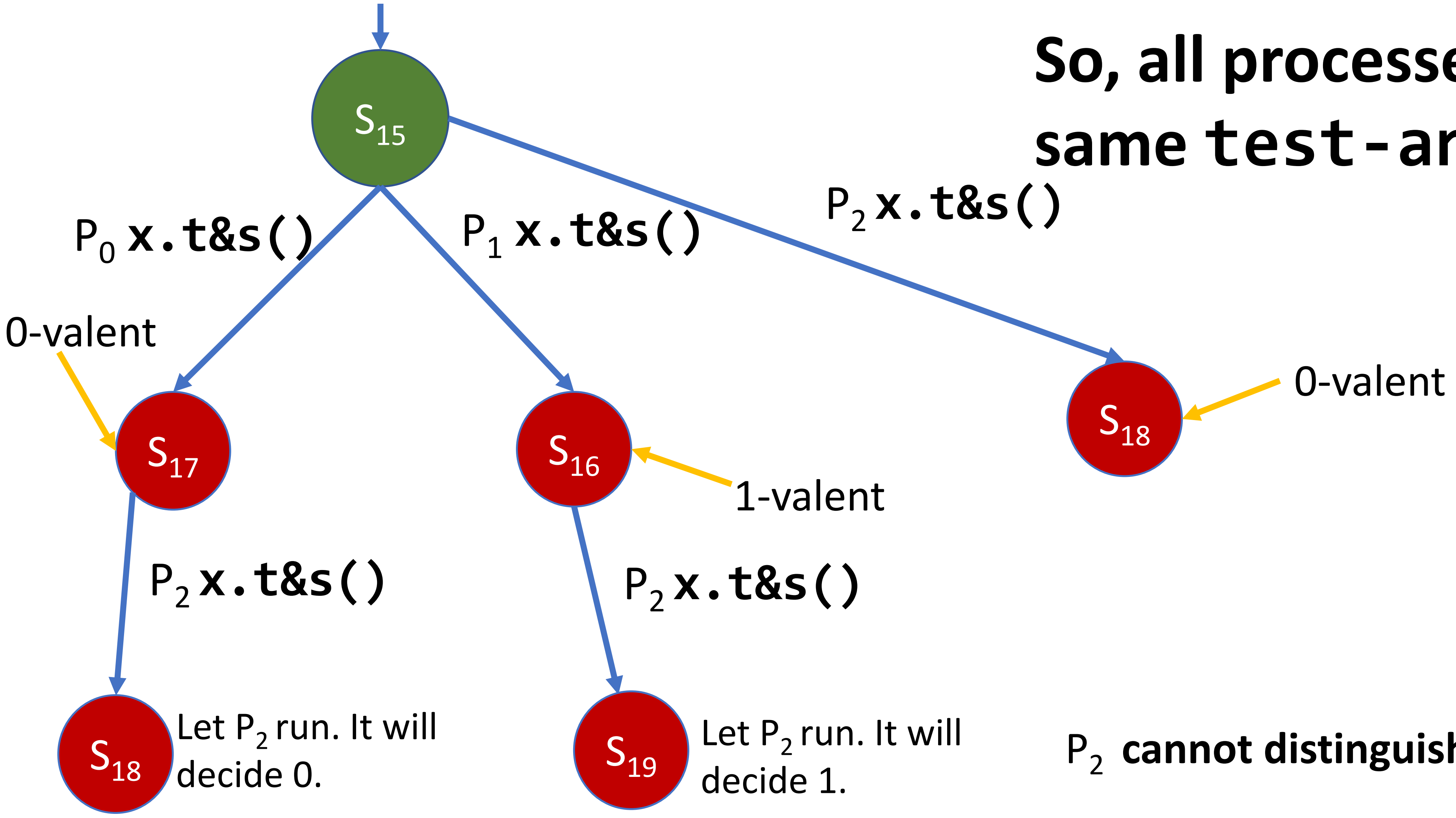
Problem 3 – `test-and-set`

So, all processes use the same `test-and-set` object.



P_2 **cannot distinguish** between S_{18} and S_{19}!

Problem 3 – `test-and-set`

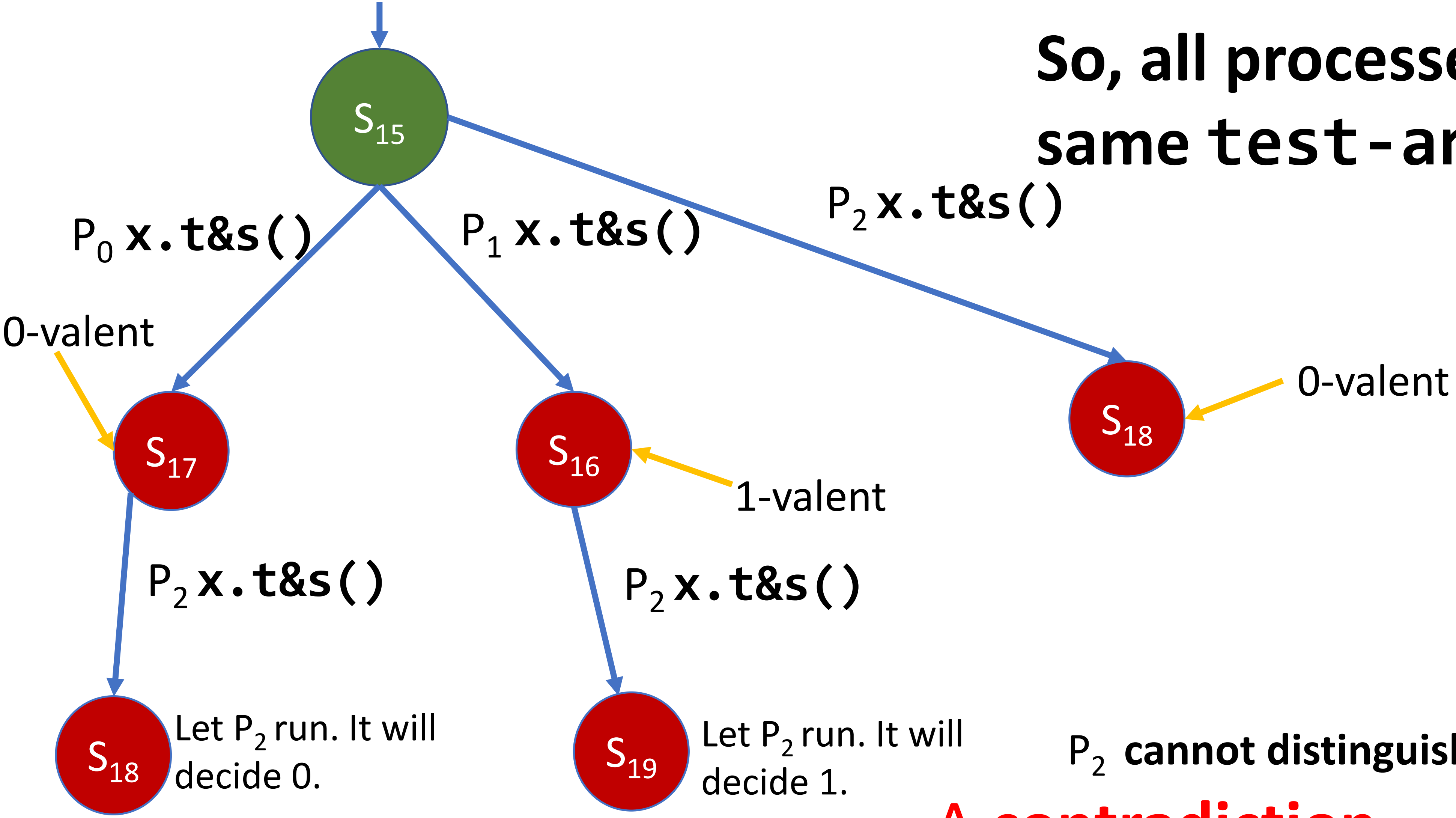


So, all processes use the same `test-and-set` object.

P_2 **cannot distinguish** between S_{18} and S_{19}!

Problem 3 – `test-and-set`

So, all processes use the same `test-and-set` object.



P_2 **cannot distinguish** between S_{18} and S_{19}!

A contradiction.

Problem 3 – `test-and-set`

In other words, the consensus number of `test-and-set` is 2.

Problem 4 – queue

Double-ended queue with a total of 3 peek operations.

```
procedure peek(end)
    if peeks_invoked == 3
        return ⊥
    peeks_invoked=peeks_invoked+1
    if end = HEAD
        return list.first()
    else
        return list.last()
```

Problem 4 – `queue`

Double-ended queue with a total of 3 peek operations.

```
procedure peek(end)
    if peeks_invoked == 3
        return ⊥
    peeks_invoked=peeks_invoked+1
    if end = HEAD
        return list.first()
    else
        return list.last()
```

Task: Solve consensus for 4 processes.

Problem 4 – queue

Double-ended queue with a total of 3 peek operations.

```
procedure propose(v)
    deque.enqueue(HEAD, v)
    winner = deque.peek(TAIL)
    if winner != ⊥
        return winner
    else
        return deque.dequeue(TAIL)
```

Problem 4 – queue

Double-ended queue with a total of 3 peek operations.

```
procedure propose(v)
    deque.enqueue(HEAD, v)
    winner = deque.peek(TAIL)
    if winner != ⊥
        return winner
    else
        return deque.dequeue(TAIL)
```

At most 1 process would dequeue.