GIAN Course on Distributed Network Algorithms

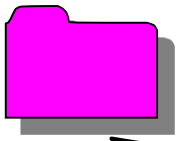
Distributed Objects

Distributed Objects

How to access and manipulate data in an efficient and consistent manner? Yet another application of spanning trees!

A fundamental challenge in distributed systems: coordinated access to a shared resource / object.

read-write



read-write

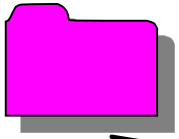
Challenge: conditional modifications require mutual exclusion: avoid race conditions.

A fundamental challenge in distributed systems: coordinated access to a shared resource / object.

Simple solution:

get and lock file

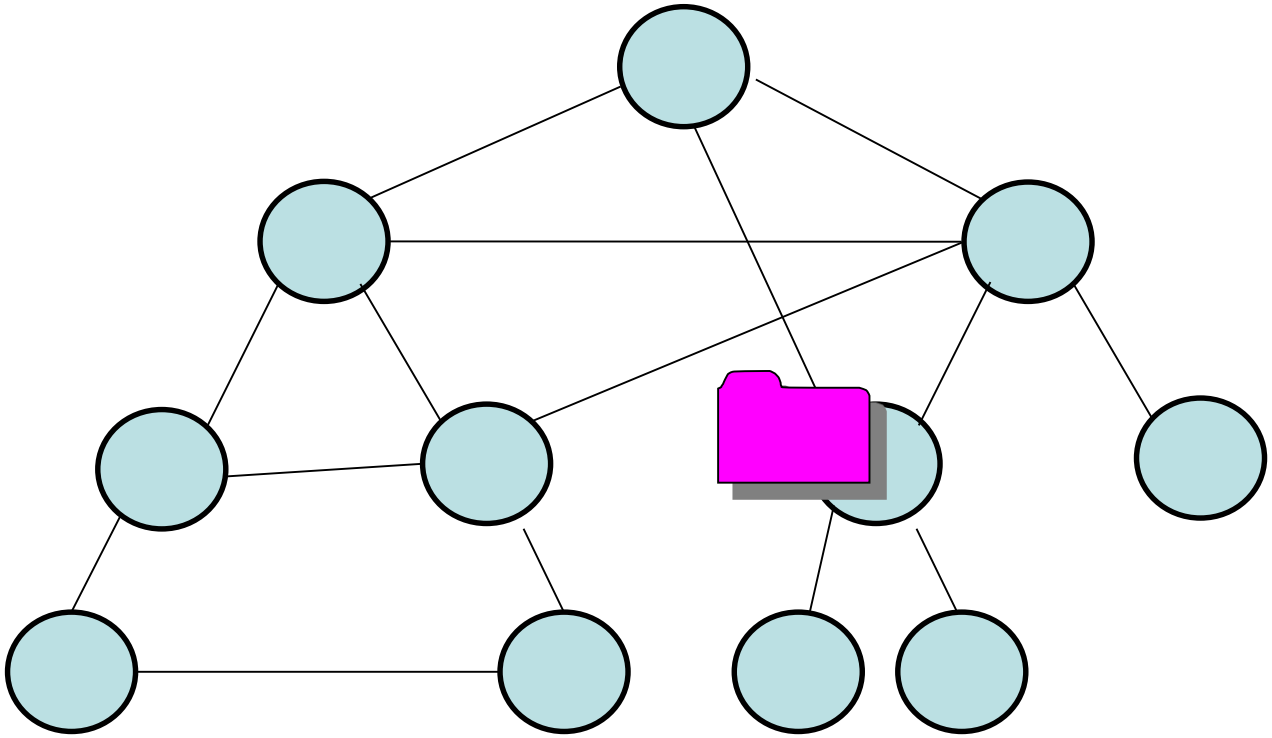
read-write



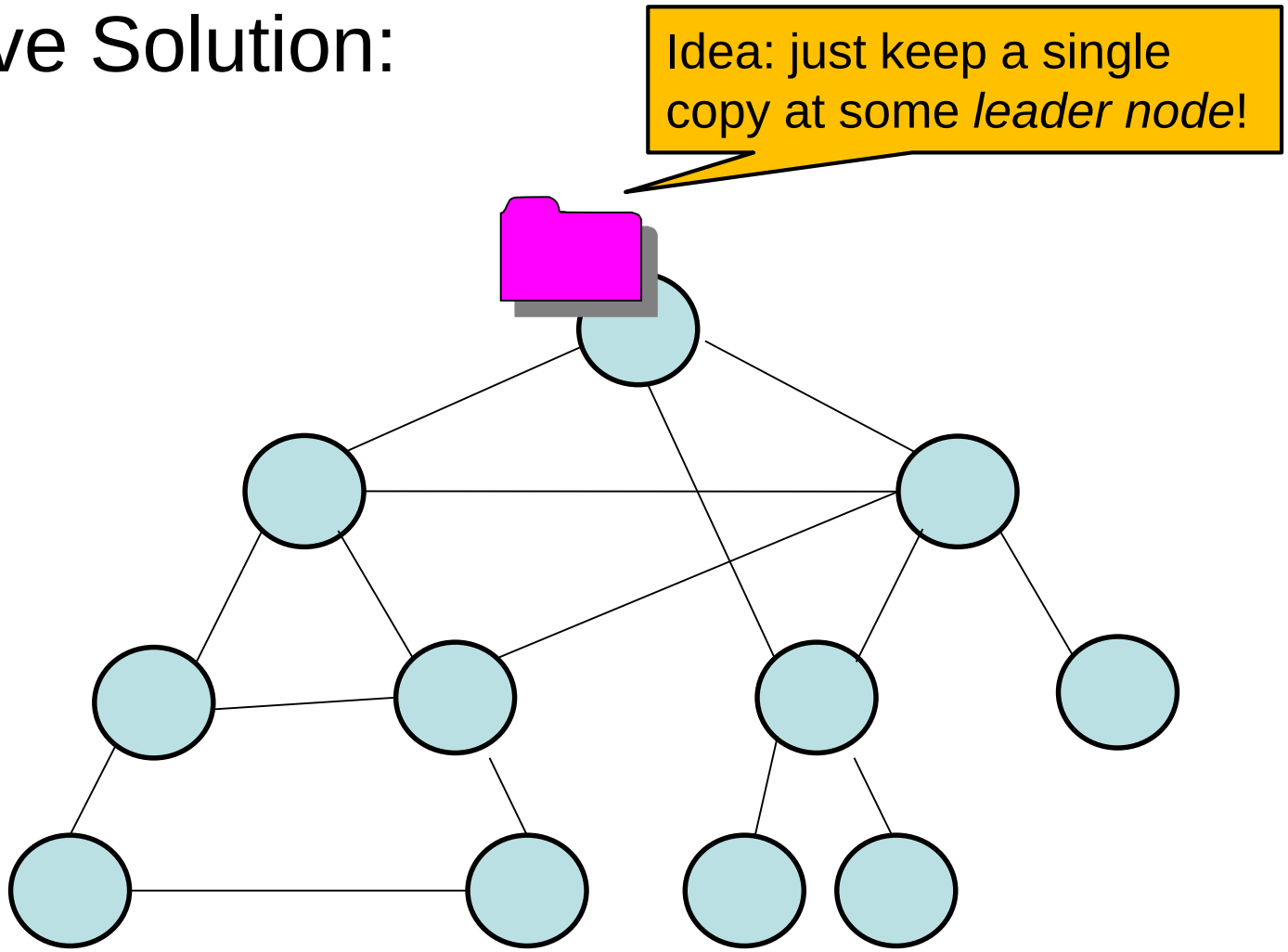
read-write

Challenge: conditional modifications require mutual exclusion: avoid race conditions.

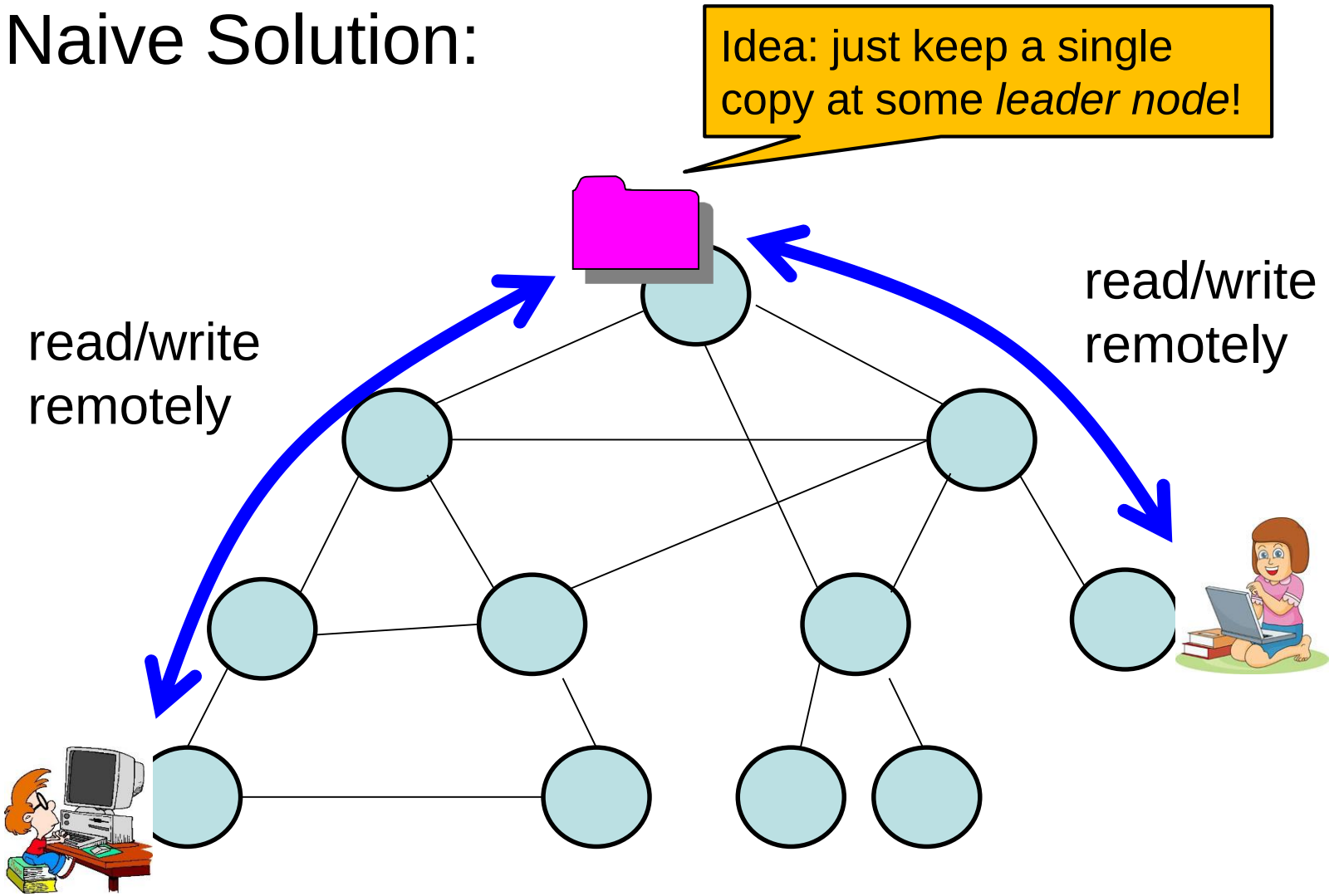
How to implement consistent read/write access *in a network*?



Naive Solution:



Naive Solution:

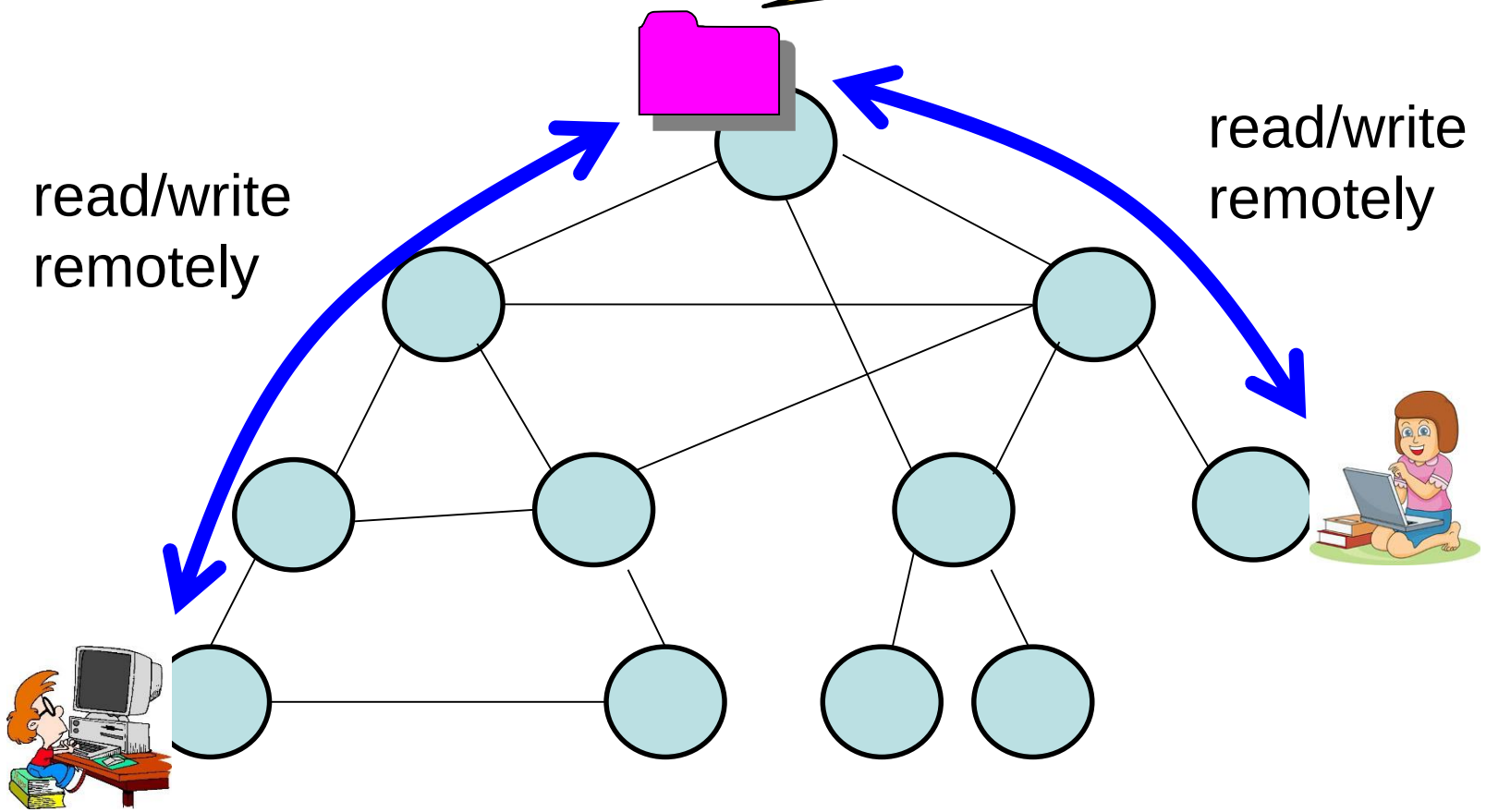


Access: 1. send message to leader,

2. leader processes request, 3. result sent back down the tree.

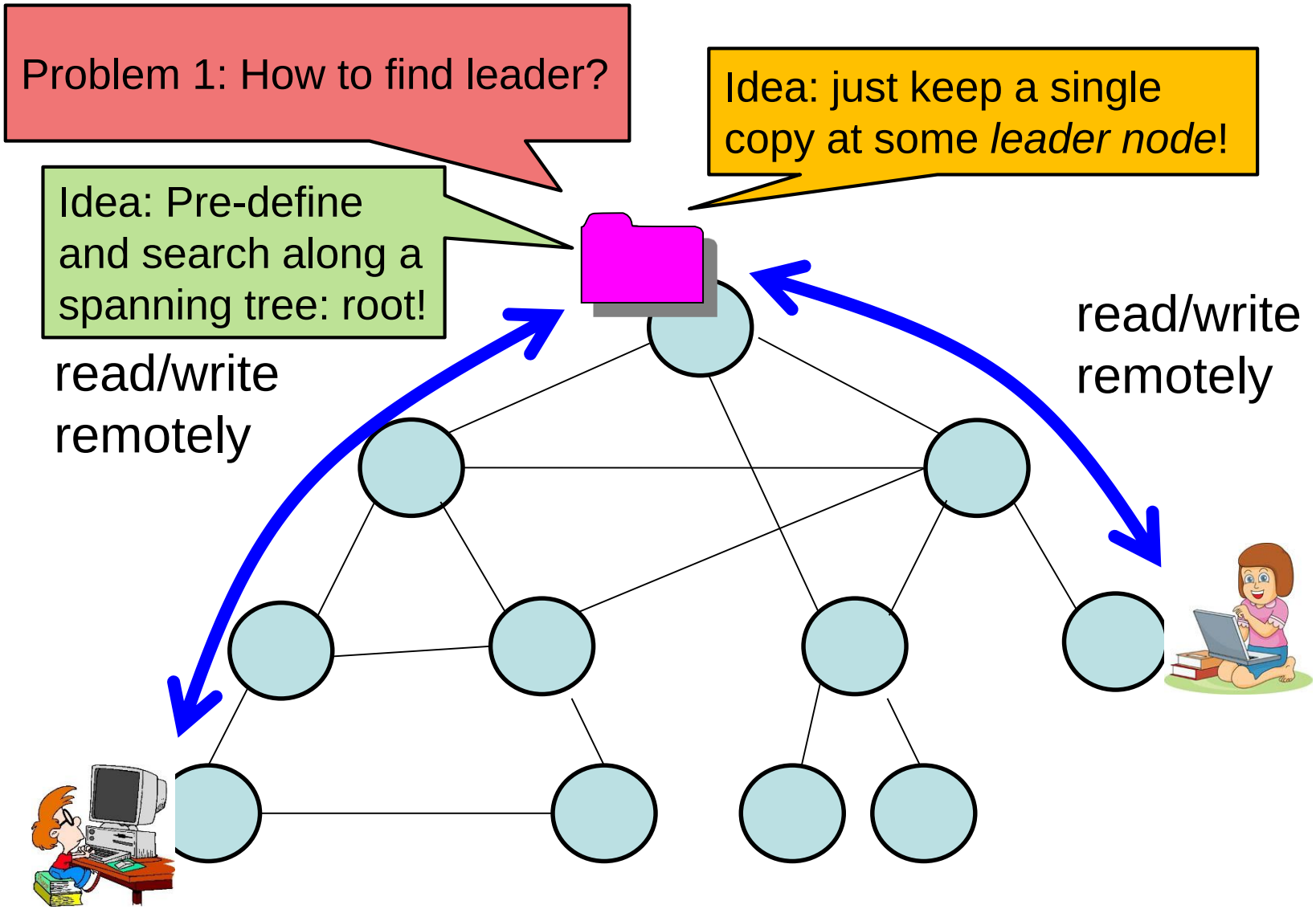
Problem 1: How to find leader?

Idea: just keep a single copy at some *leader node*!



Access: 1. send message to leader,

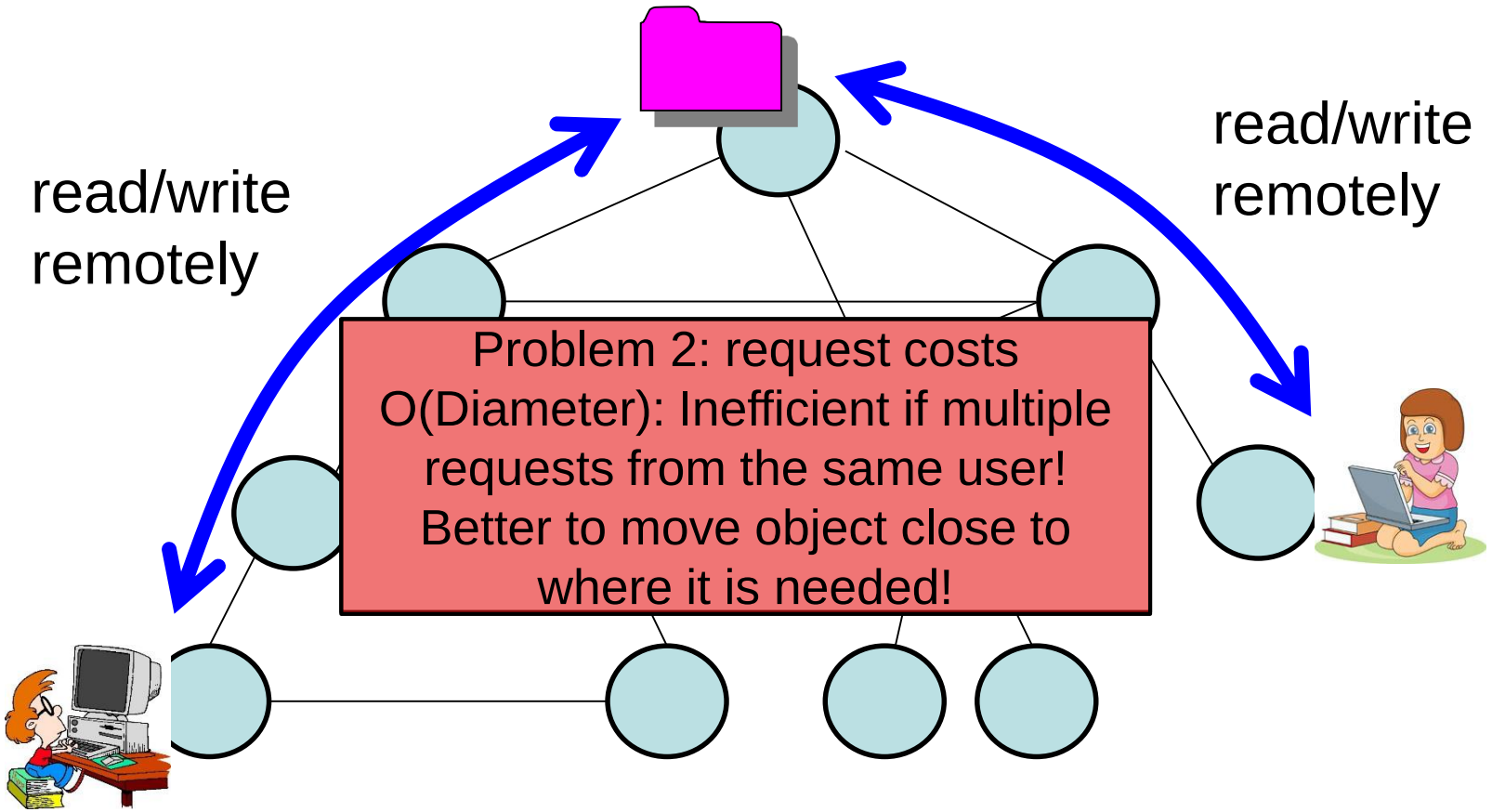
2. leader processes request, 3. result sent back down the tree.



Access: 1. send message to leader,

2. leader processes request, 3. result sent back down the tree.

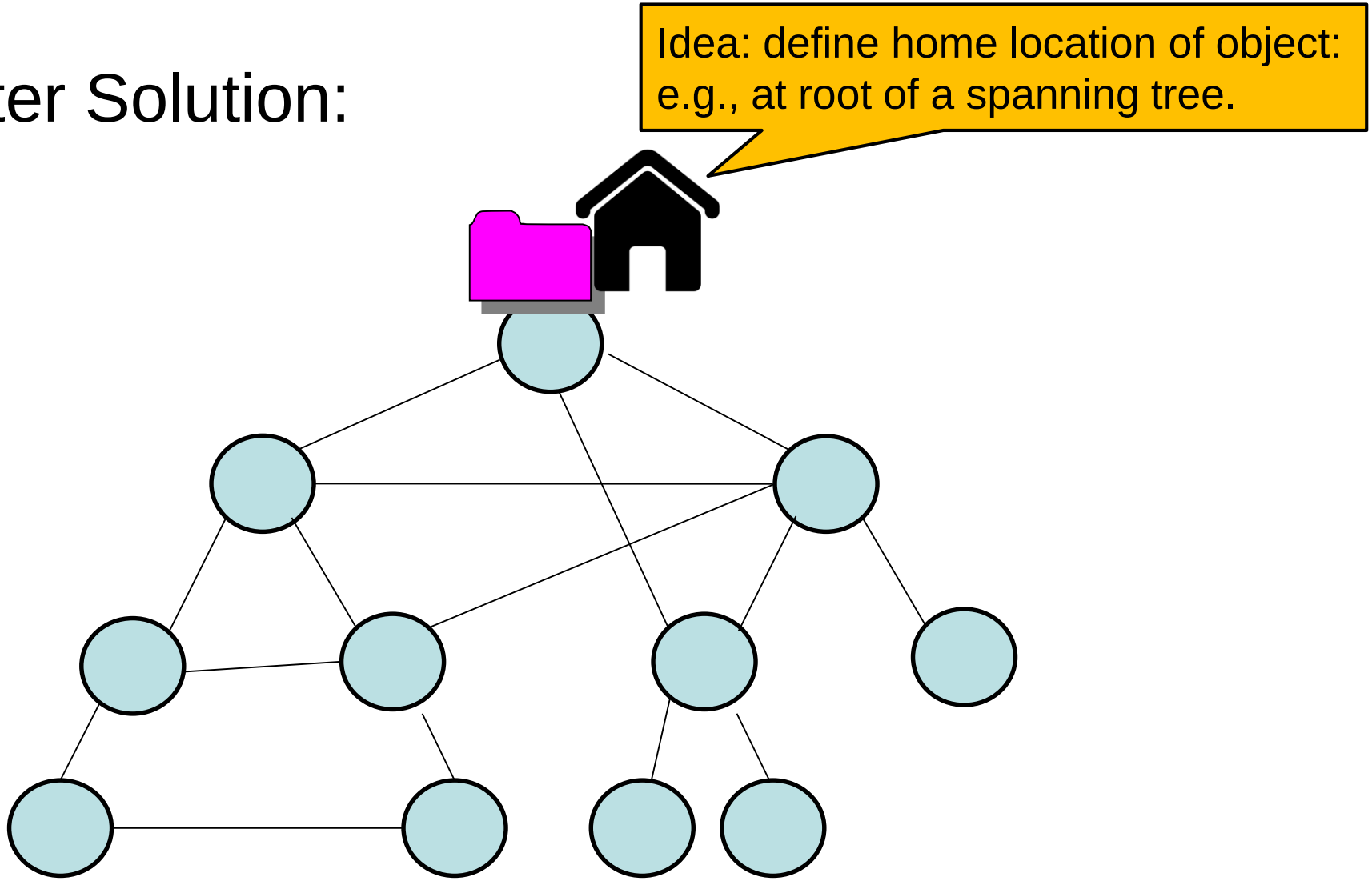
Naive Solution:



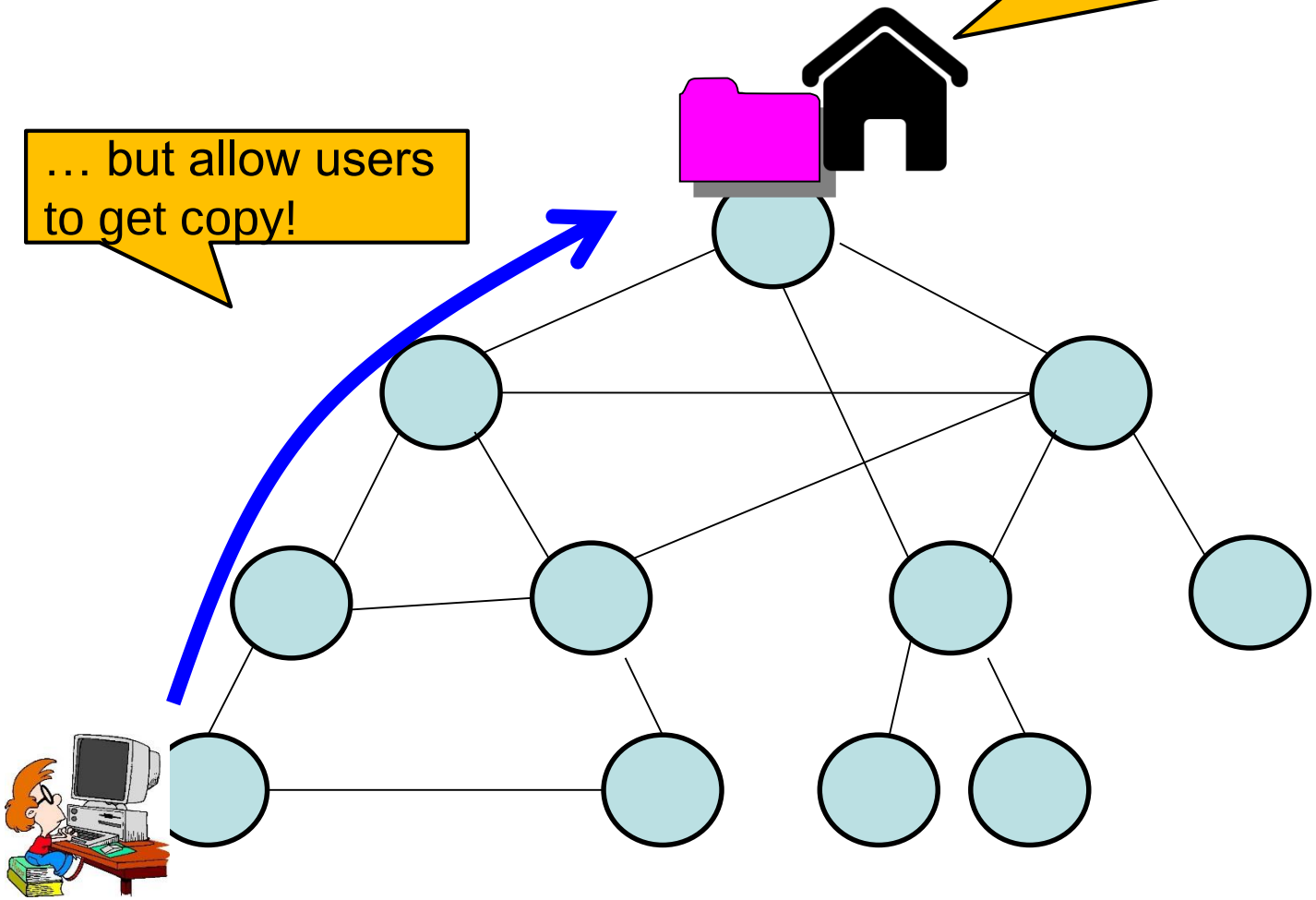
Access: 1. send message to leader,

2. leader processes request, 3. result sent back down the tree.

Better Solution:

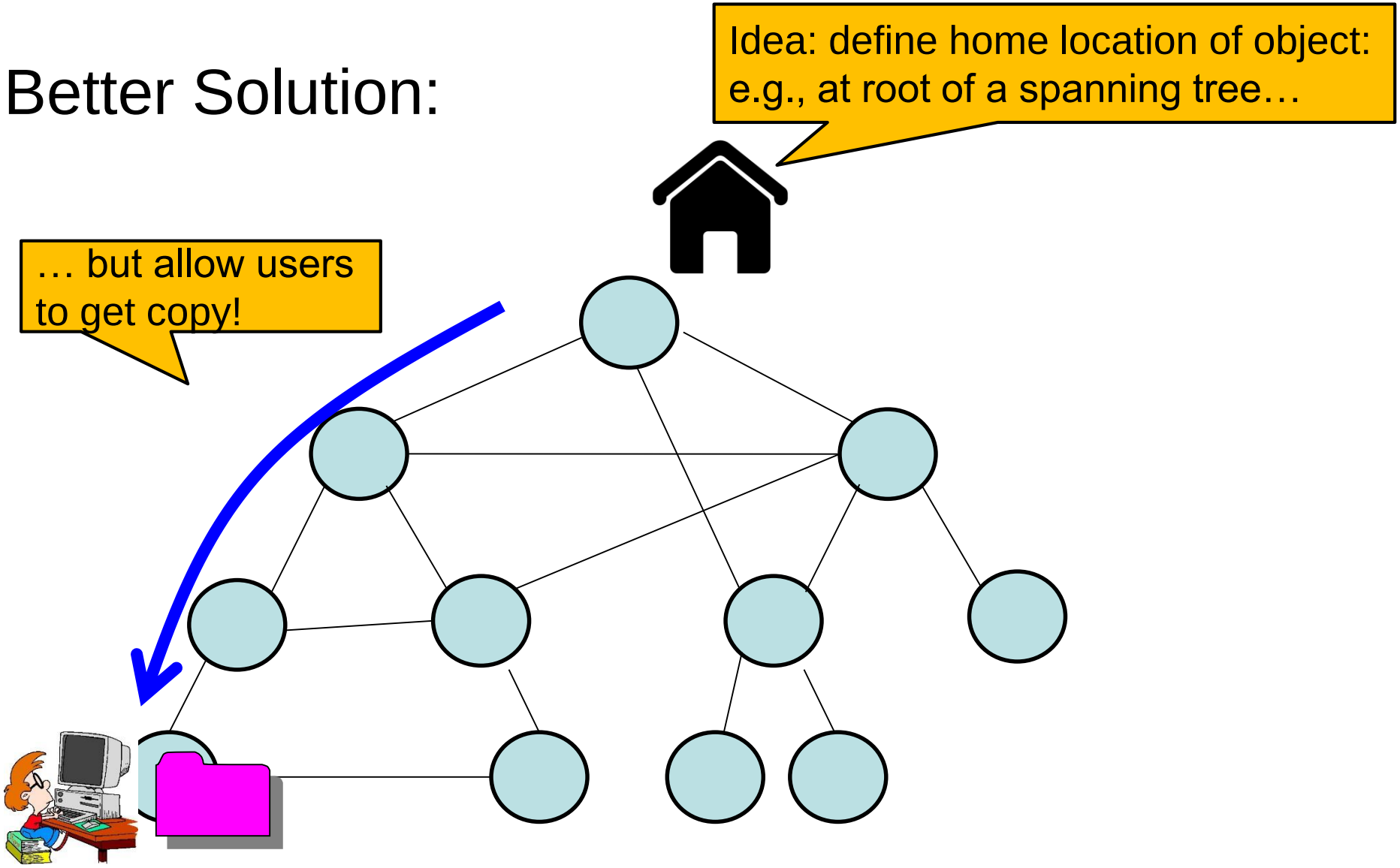


Better Solution:



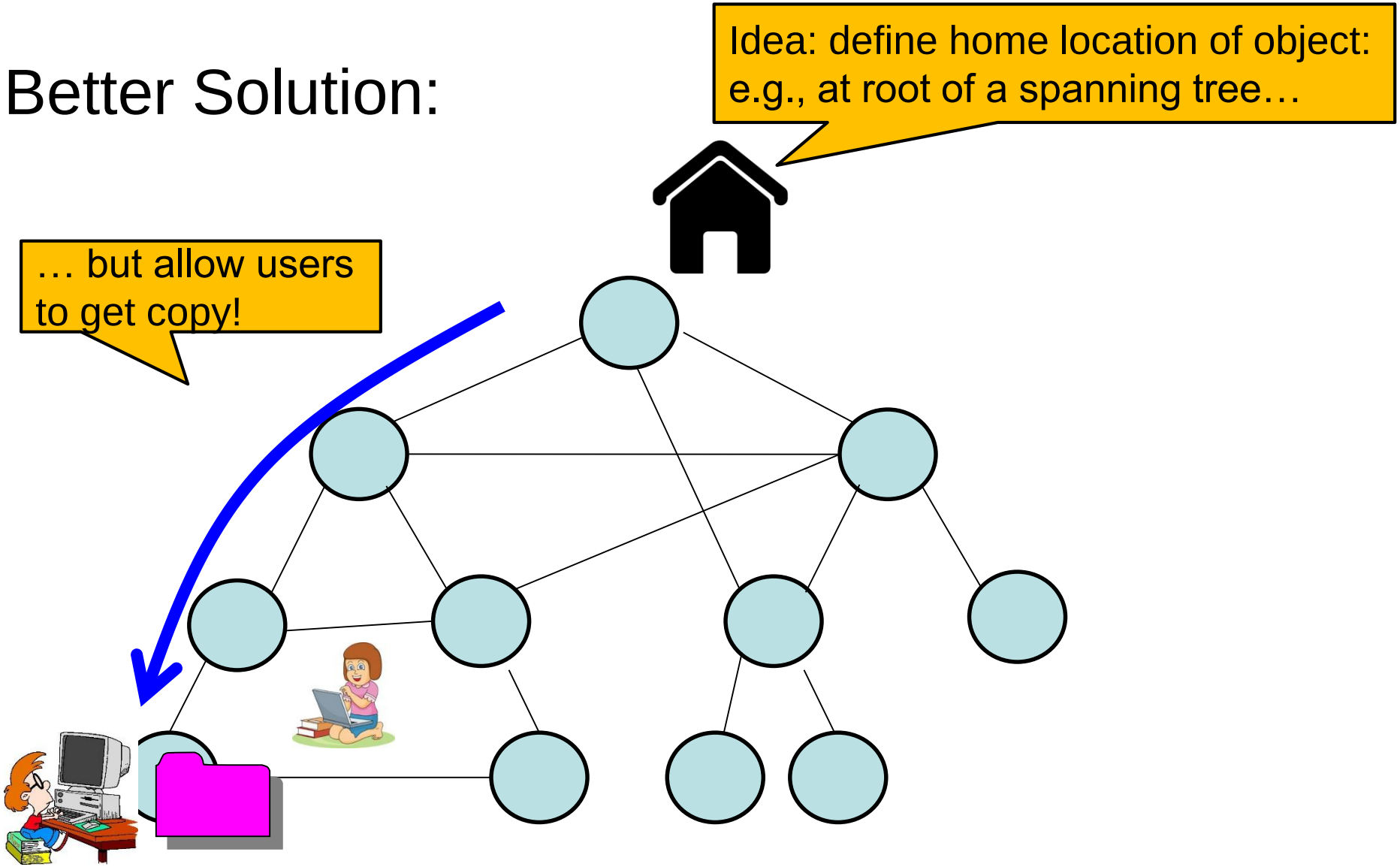
Assume: Access by Bob

Better Solution:



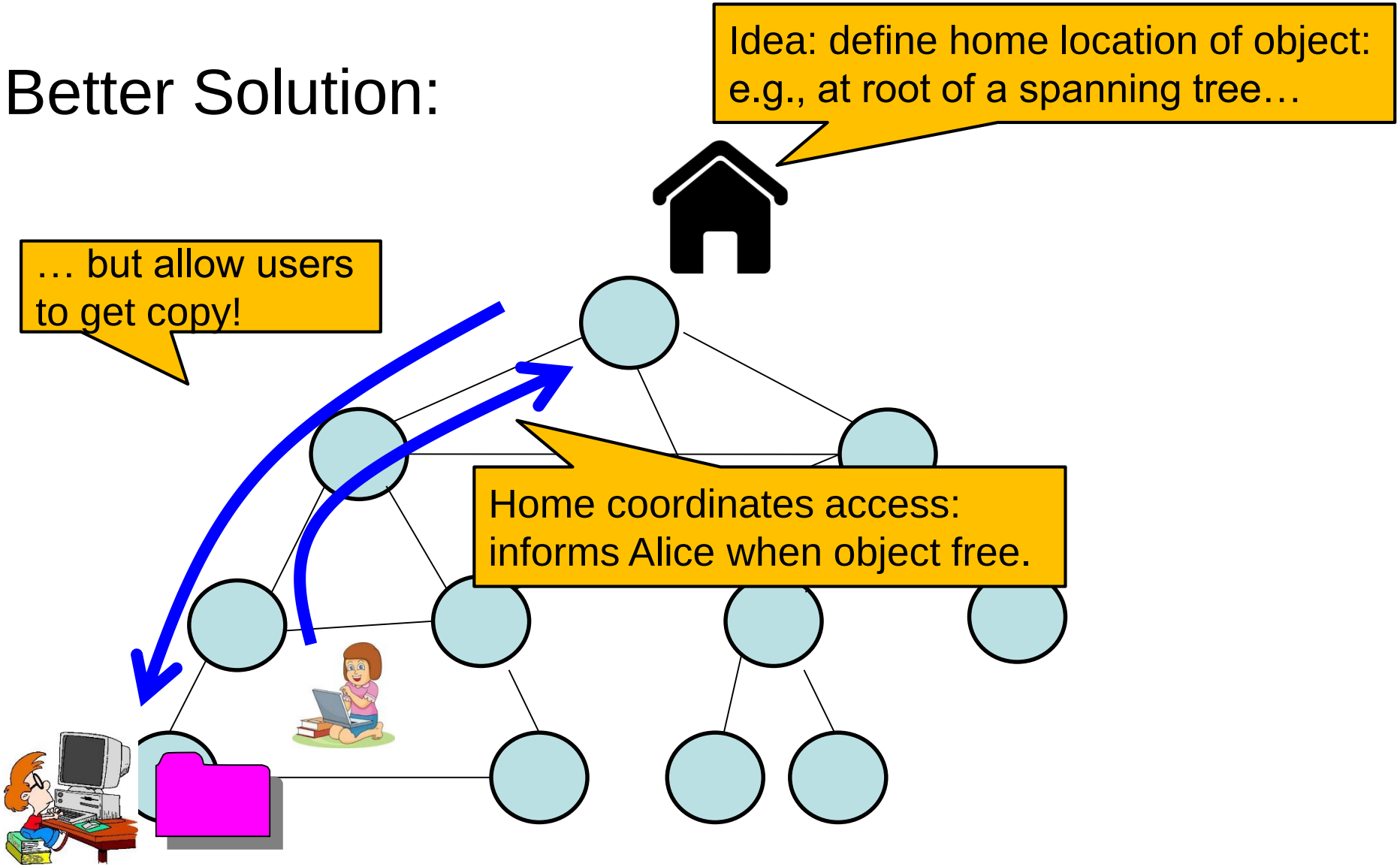
Assume: Access by Bob

Better Solution:



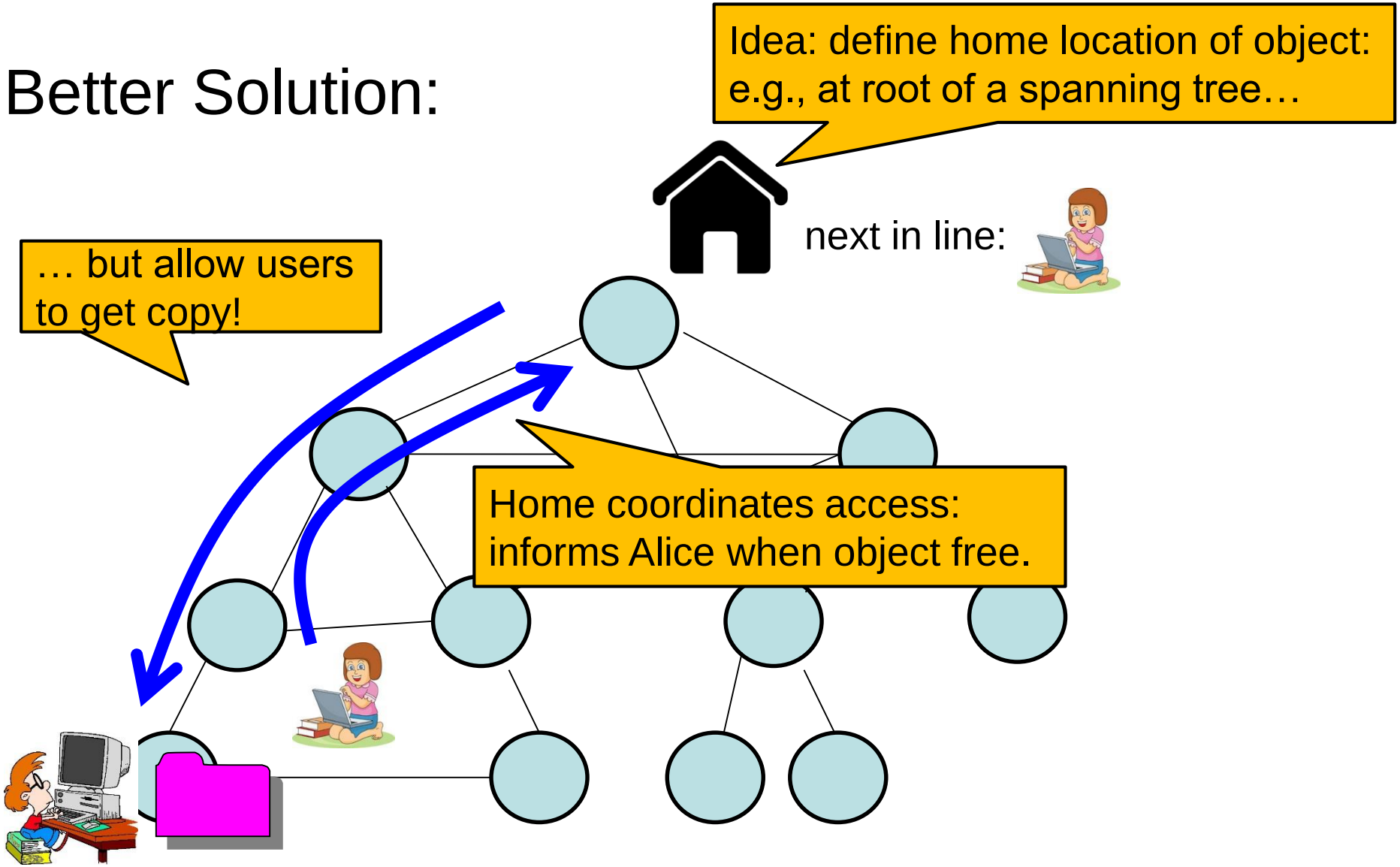
What if Alice wants access?

Better Solution:



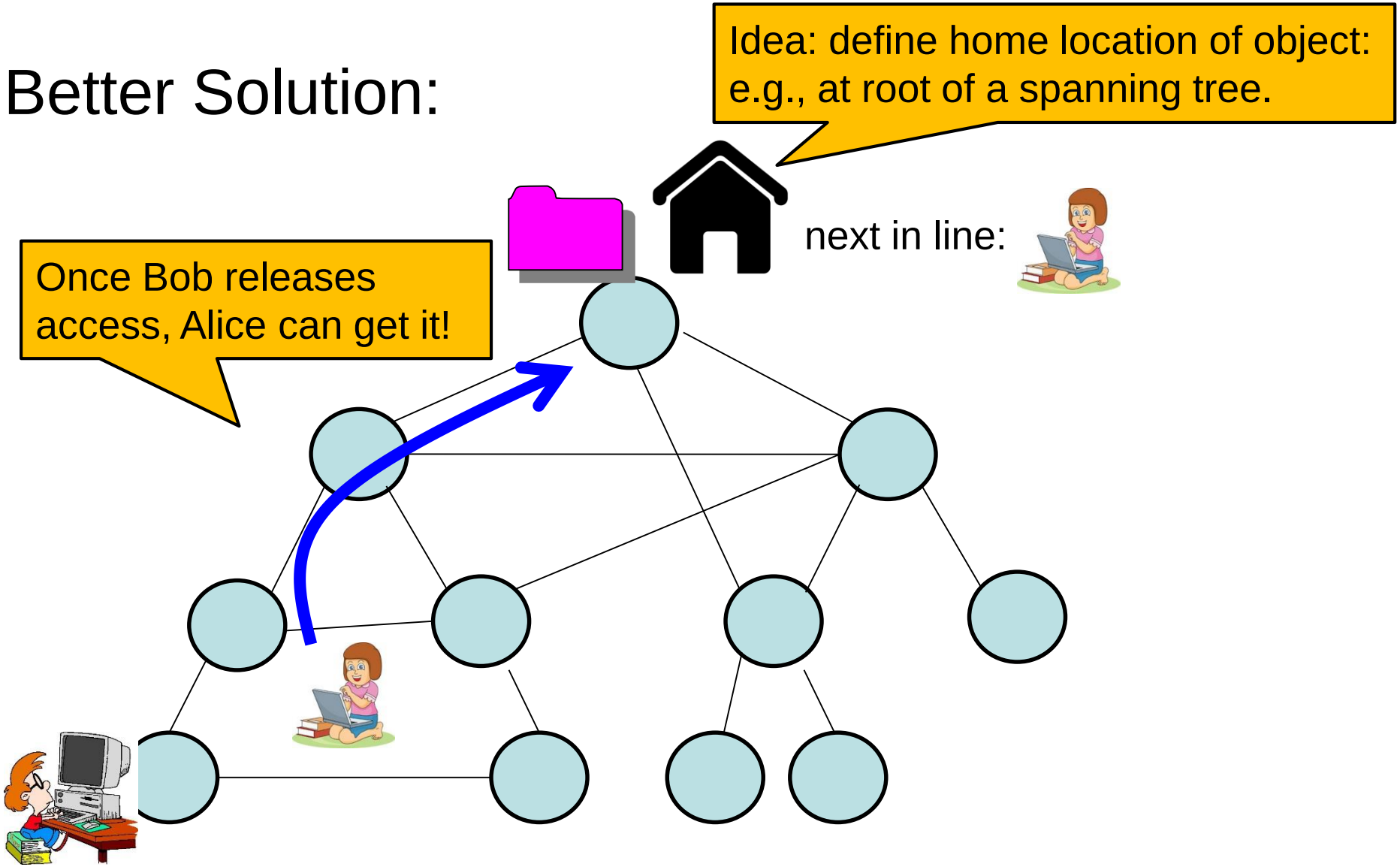
What if Alice wants access?

Better Solution:



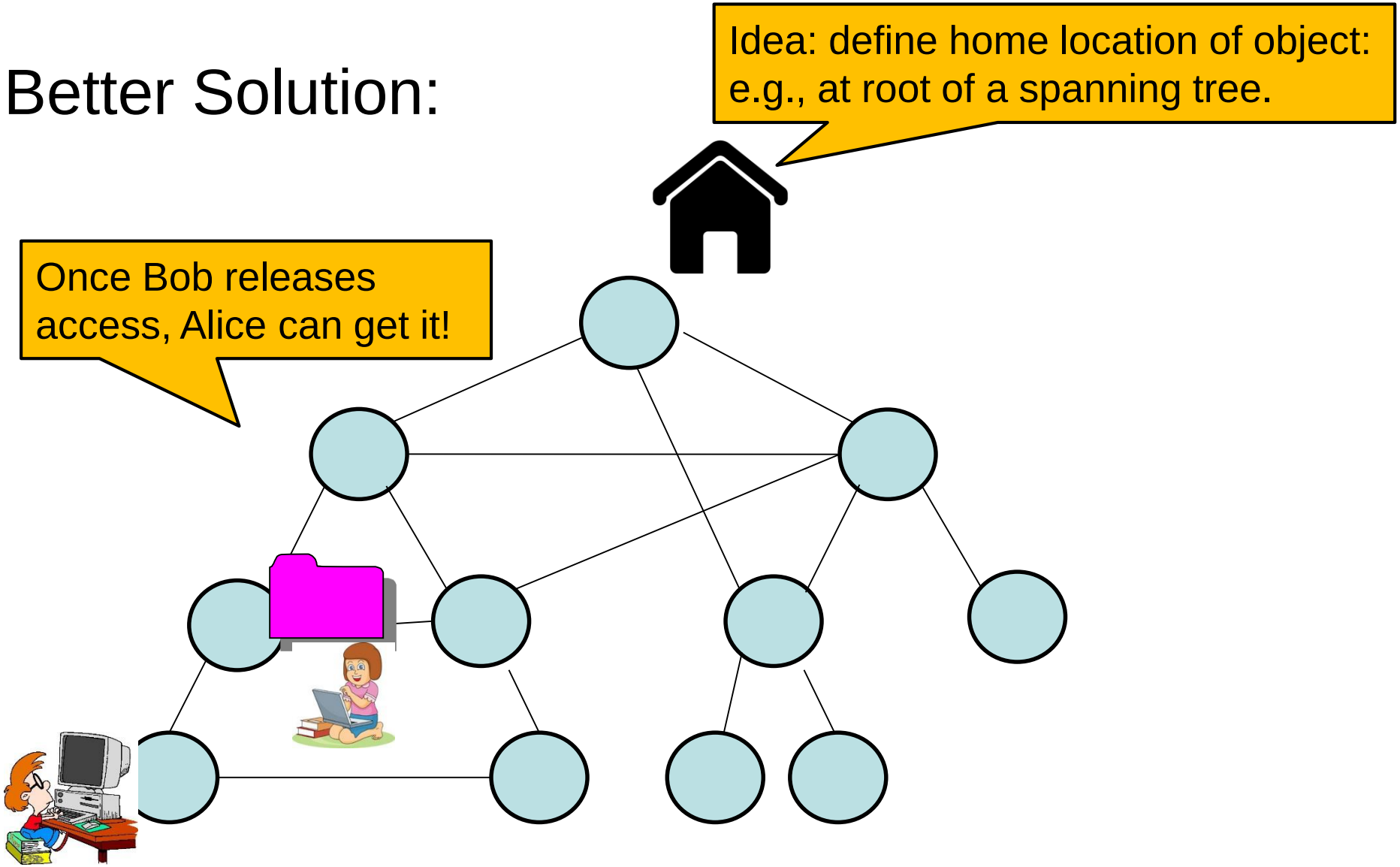
What if Alice wants access?

Better Solution:



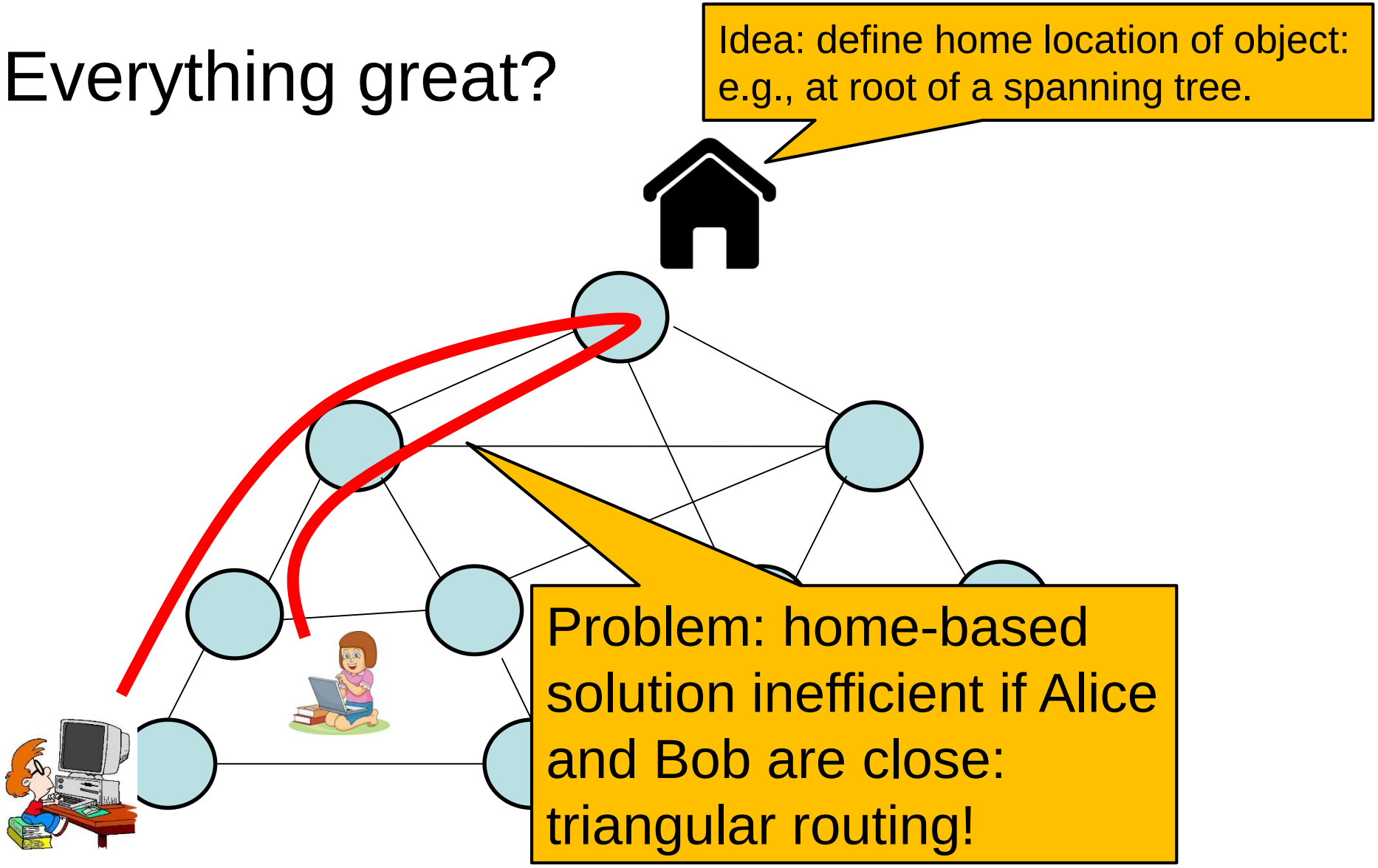
What if Alice wants access?

Better Solution:



What if Alice wants access?

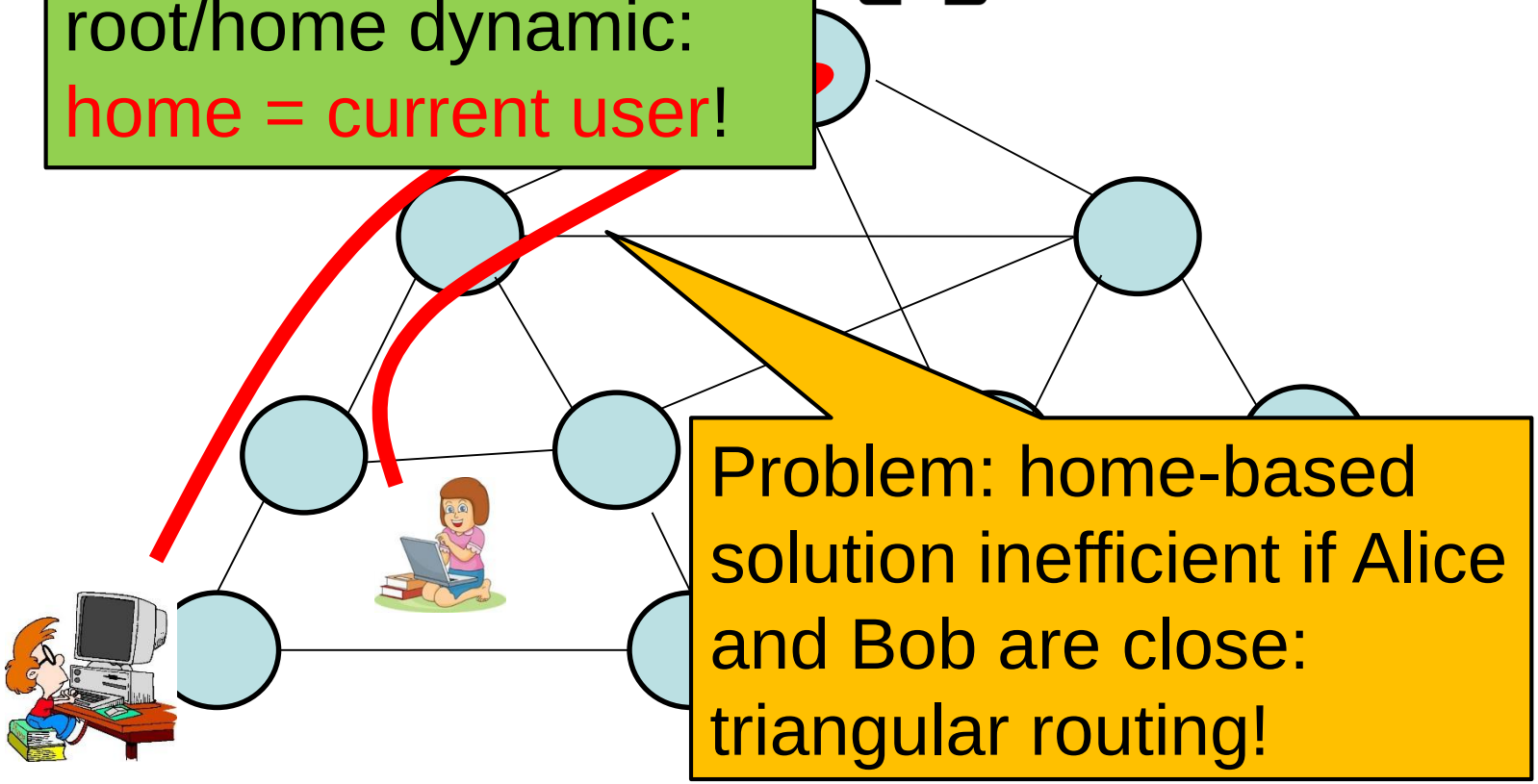
Everything great?



Everything great?

Idea: define home location of object: e.g., at root of a spanning tree.

Idea: make the root/home dynamic: home = current user!

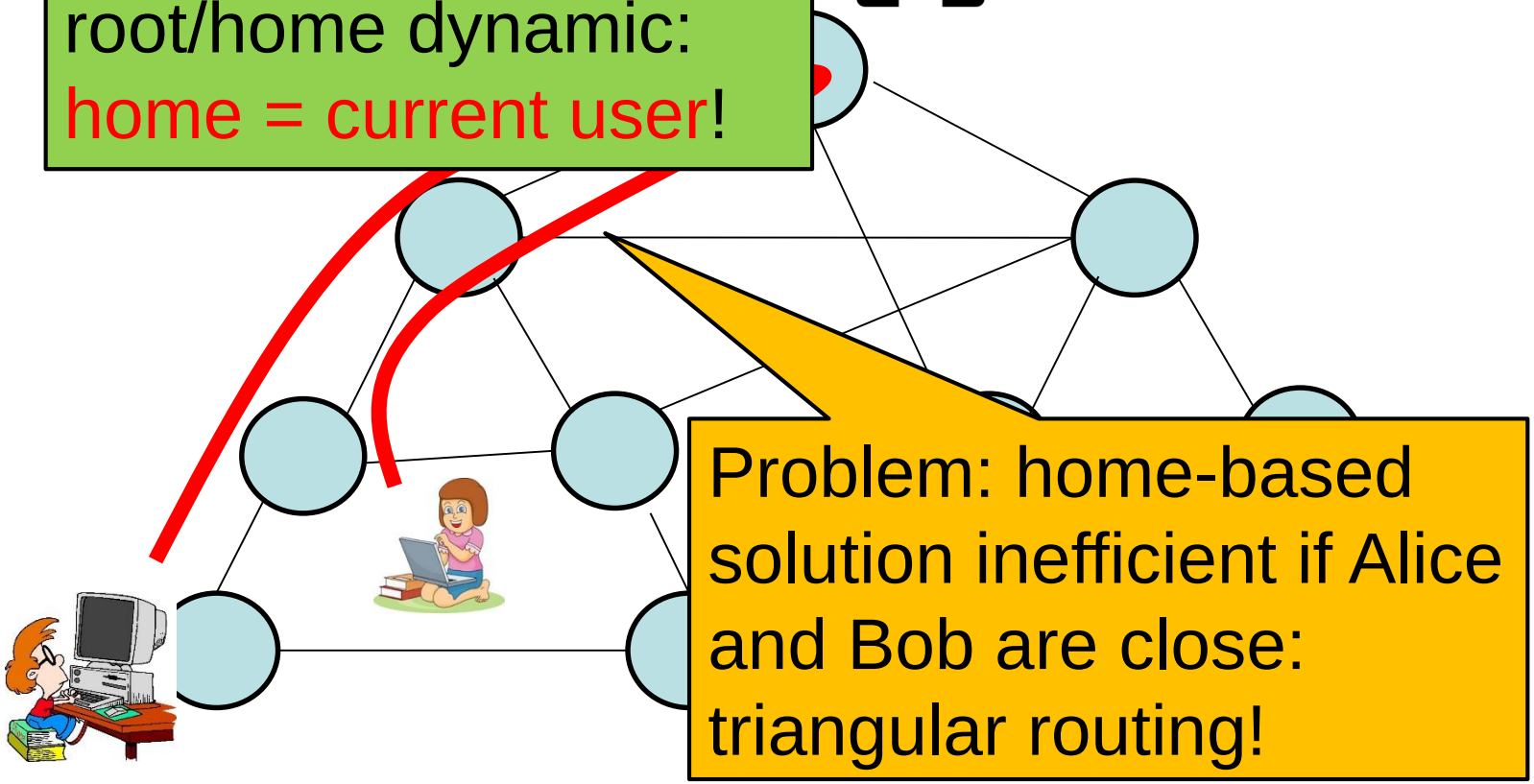


Problem: home-based solution inefficient if Alice and Bob are close: triangular routing!

Everything great?

Idea: define home location of object: e.g., at root of a spanning tree.

Idea: make the root/home dynamic: home = current user!



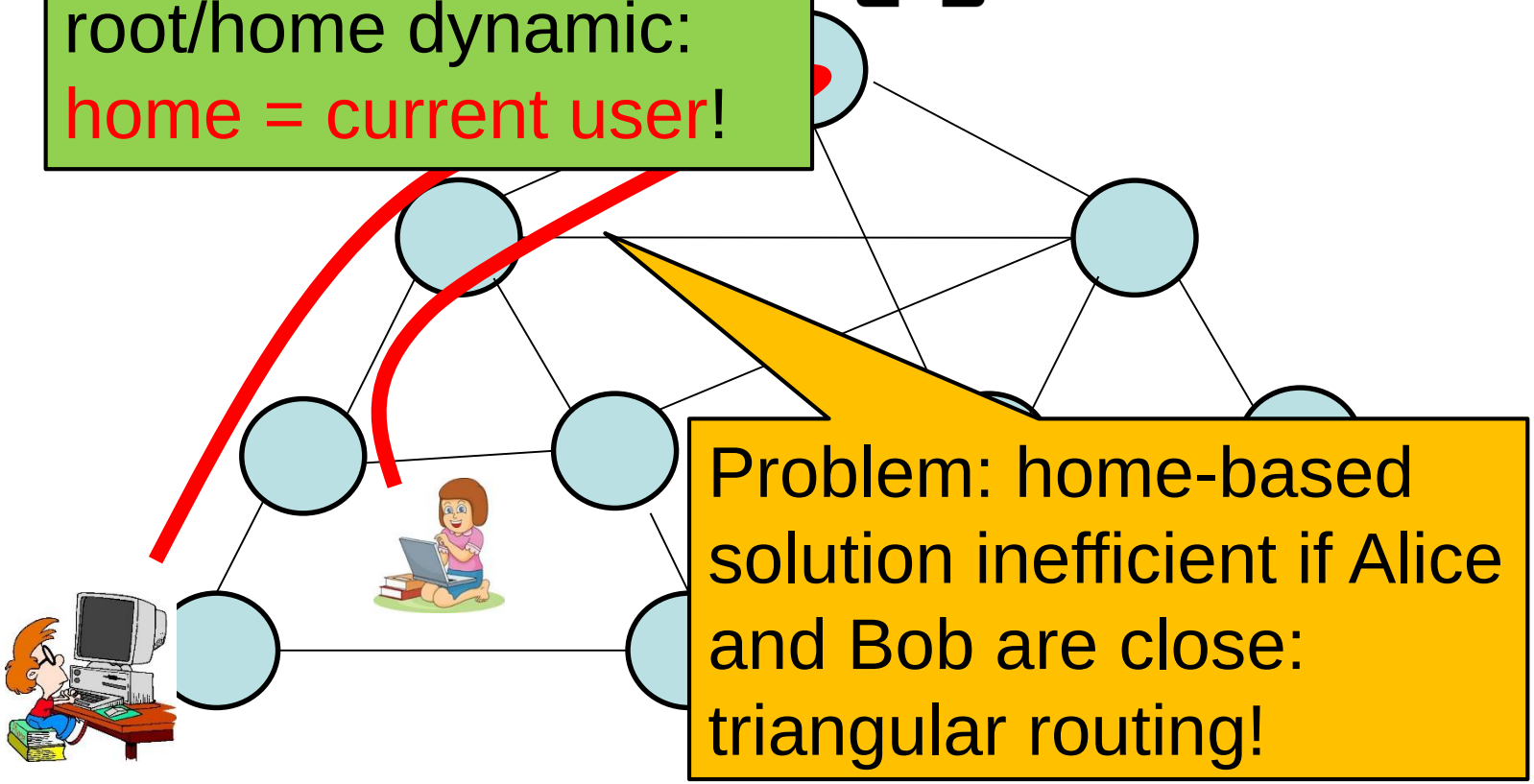
Problem: home-based solution inefficient if Alice and Bob are close: triangular routing!

The Arrow Protocol!

Everything great?

Idea: define home location of object: e.g., at root of a spanning tree.

Idea: make the root/home dynamic: home = current user!

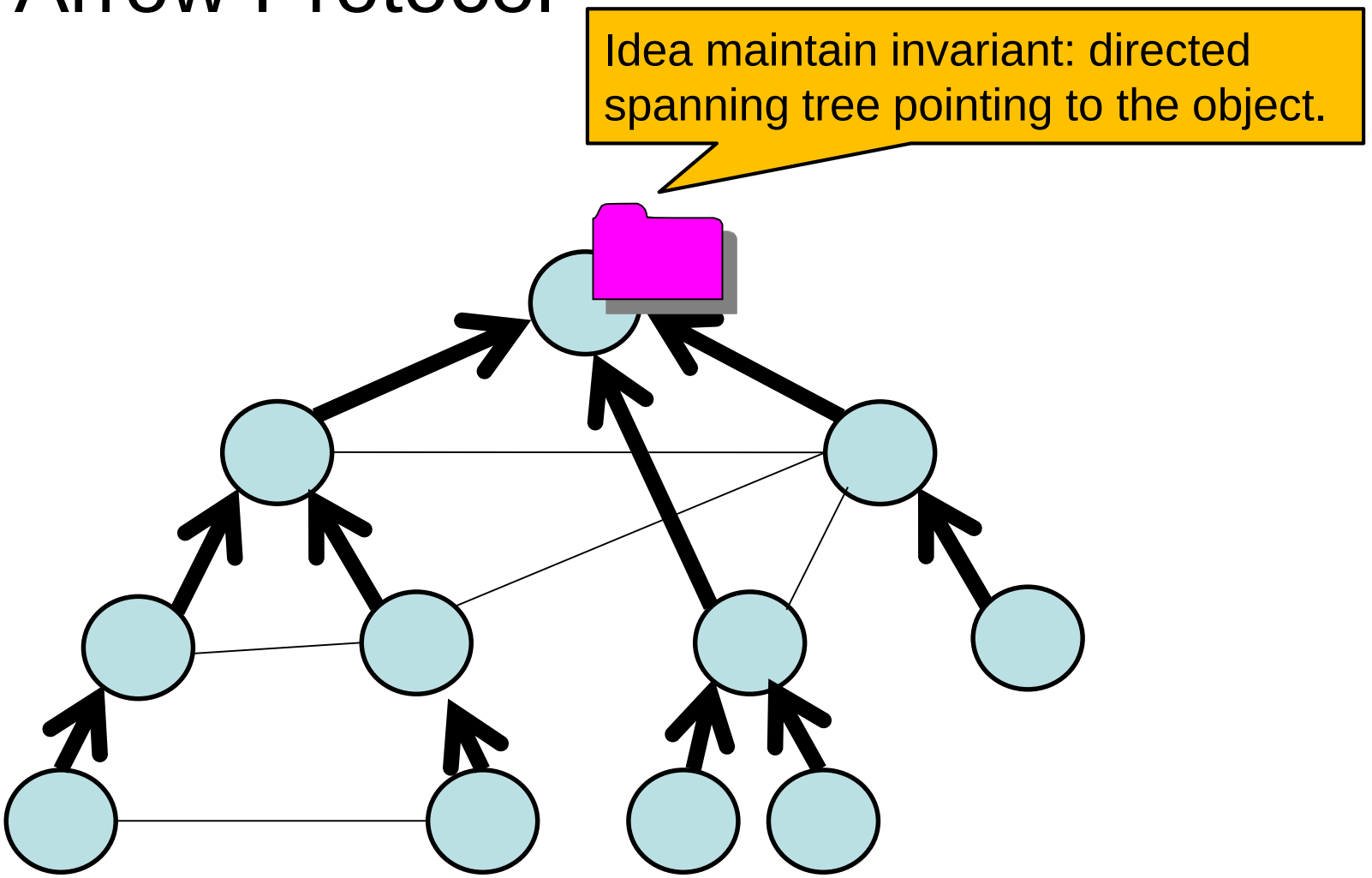


Problem: home-based solution inefficient if Alice and Bob are close: triangular routing!

The Arrow Protocol!

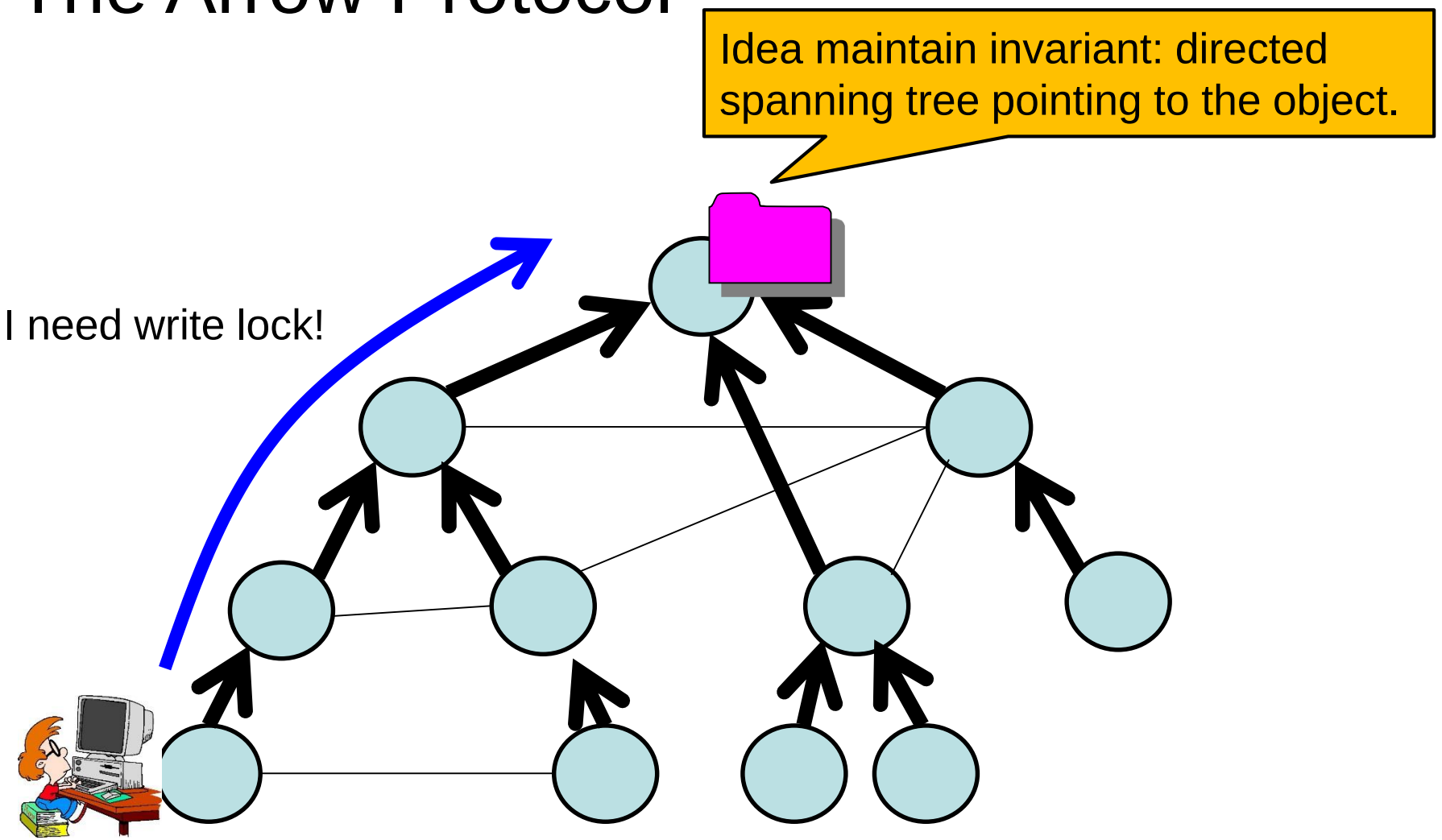
Based on link reversals: remember MST!

The Arrow Protocol



In the Beginning

The Arrow Protocol

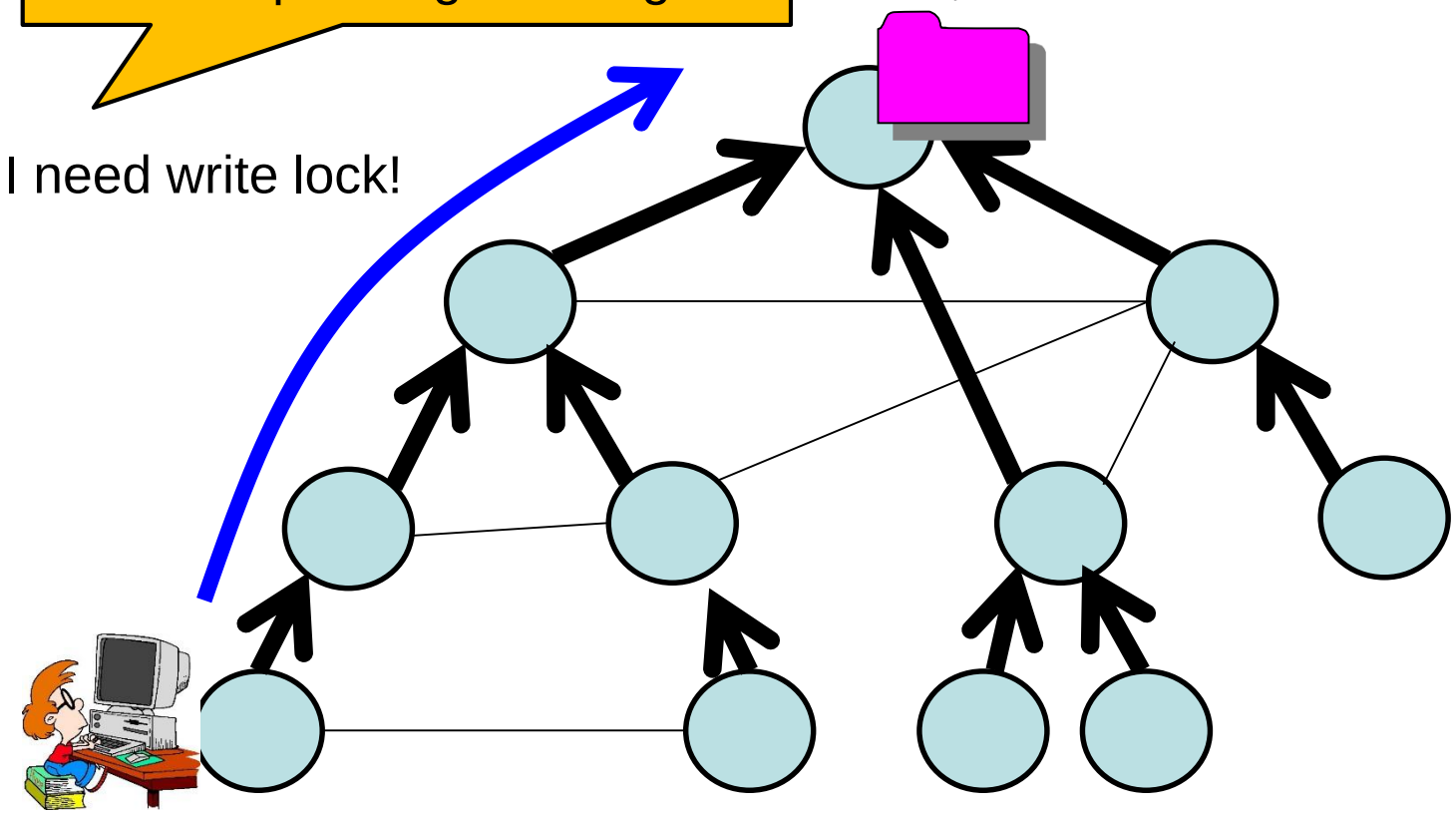


At Runtime

The Arrow Protocol

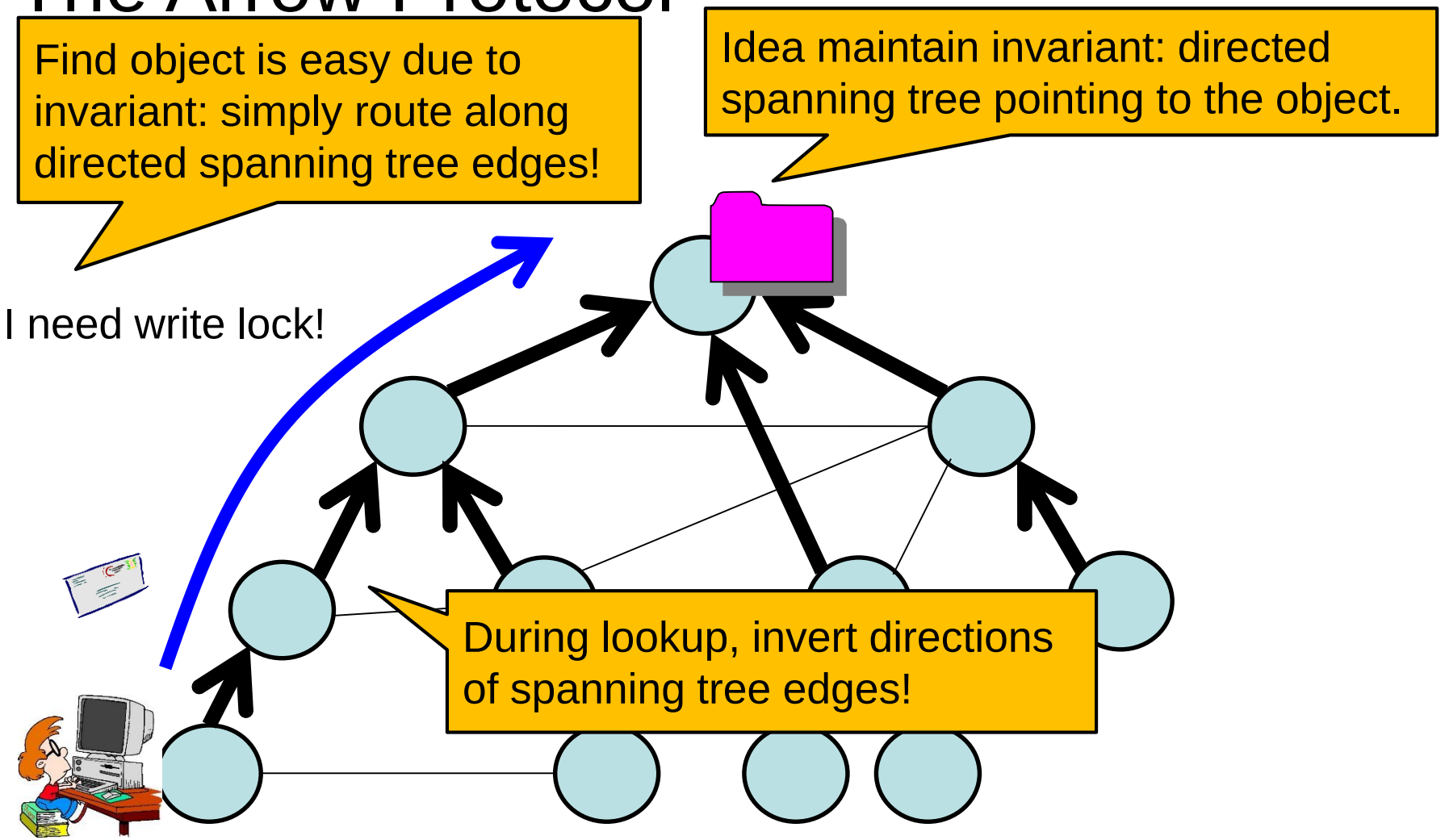
Find object is easy due to invariant: simply route along directed spanning tree edges!

Idea maintain invariant: directed spanning tree pointing to the object.



At Runtime

The Arrow Protocol



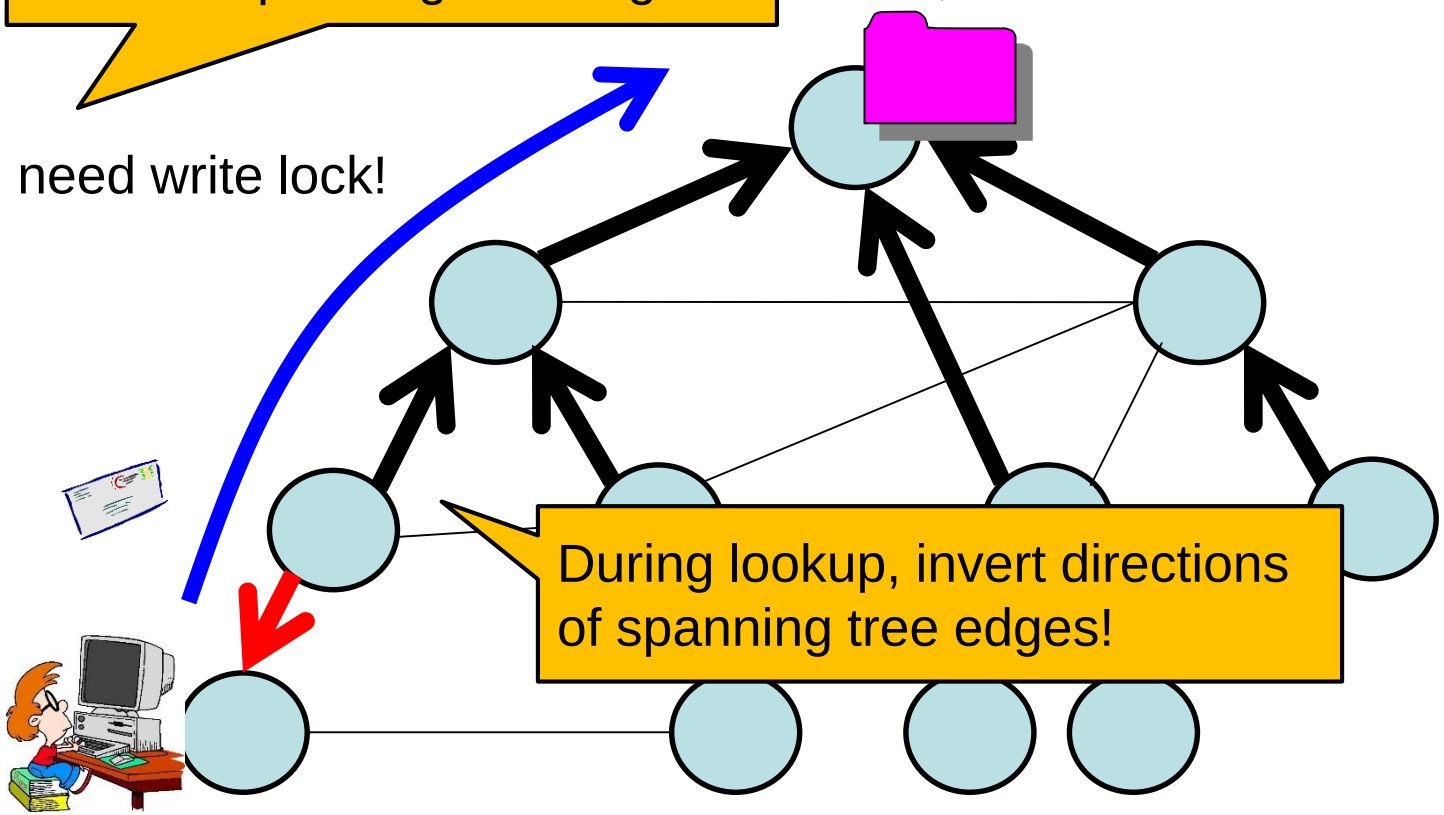
At Runtime

The Arrow Protocol

Find object is easy due to invariant: simply route along directed spanning tree edges!

Idea maintain invariant: directed spanning tree pointing to the object.

I need write lock!



During lookup, invert directions of spanning tree edges!

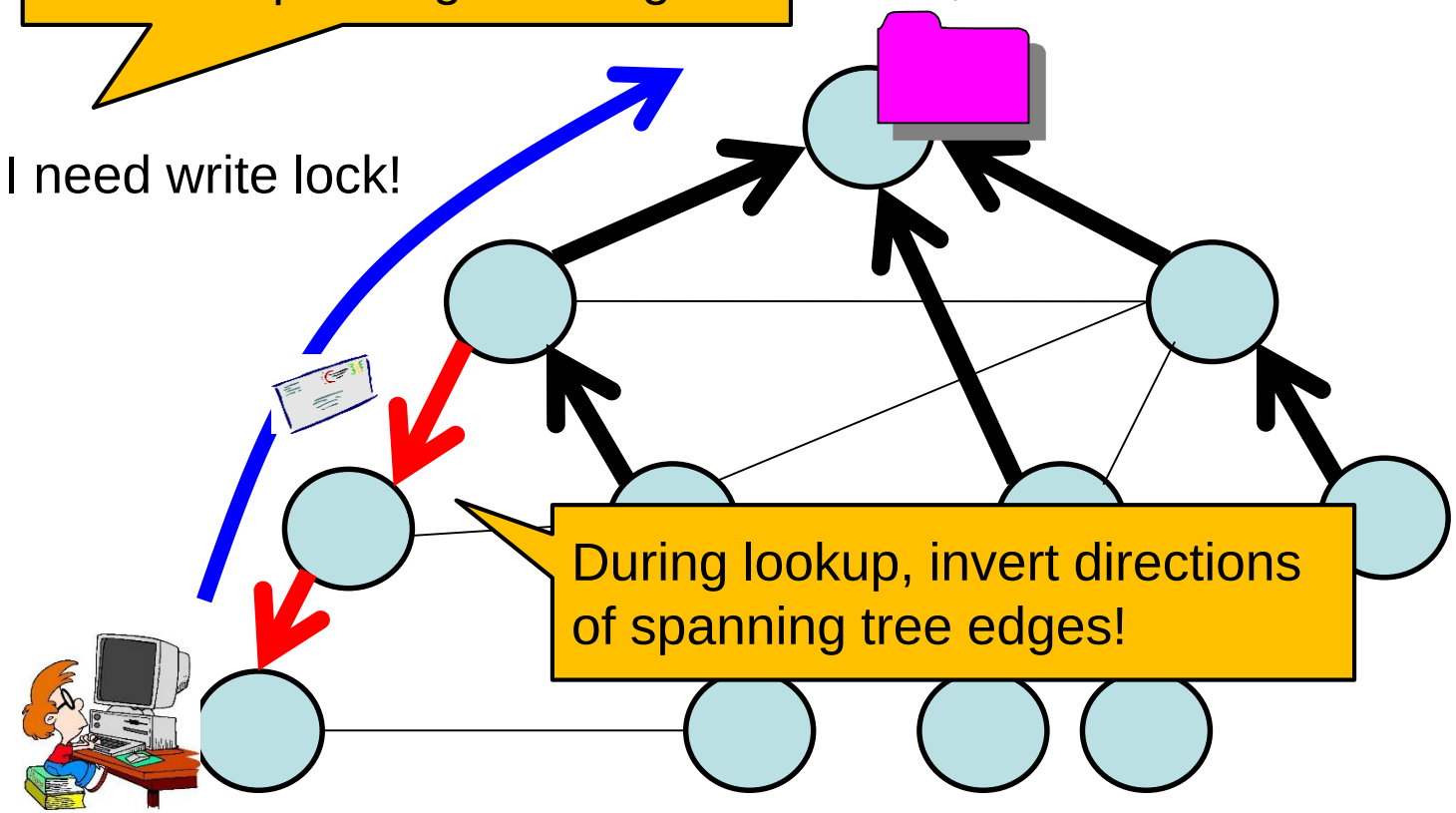
At Runtime

The Arrow Protocol

Find object is easy due to invariant: simply route along directed spanning tree edges!

Idea maintain invariant: directed spanning tree pointing to the object.

I need write lock!



During lookup, invert directions of spanning tree edges!

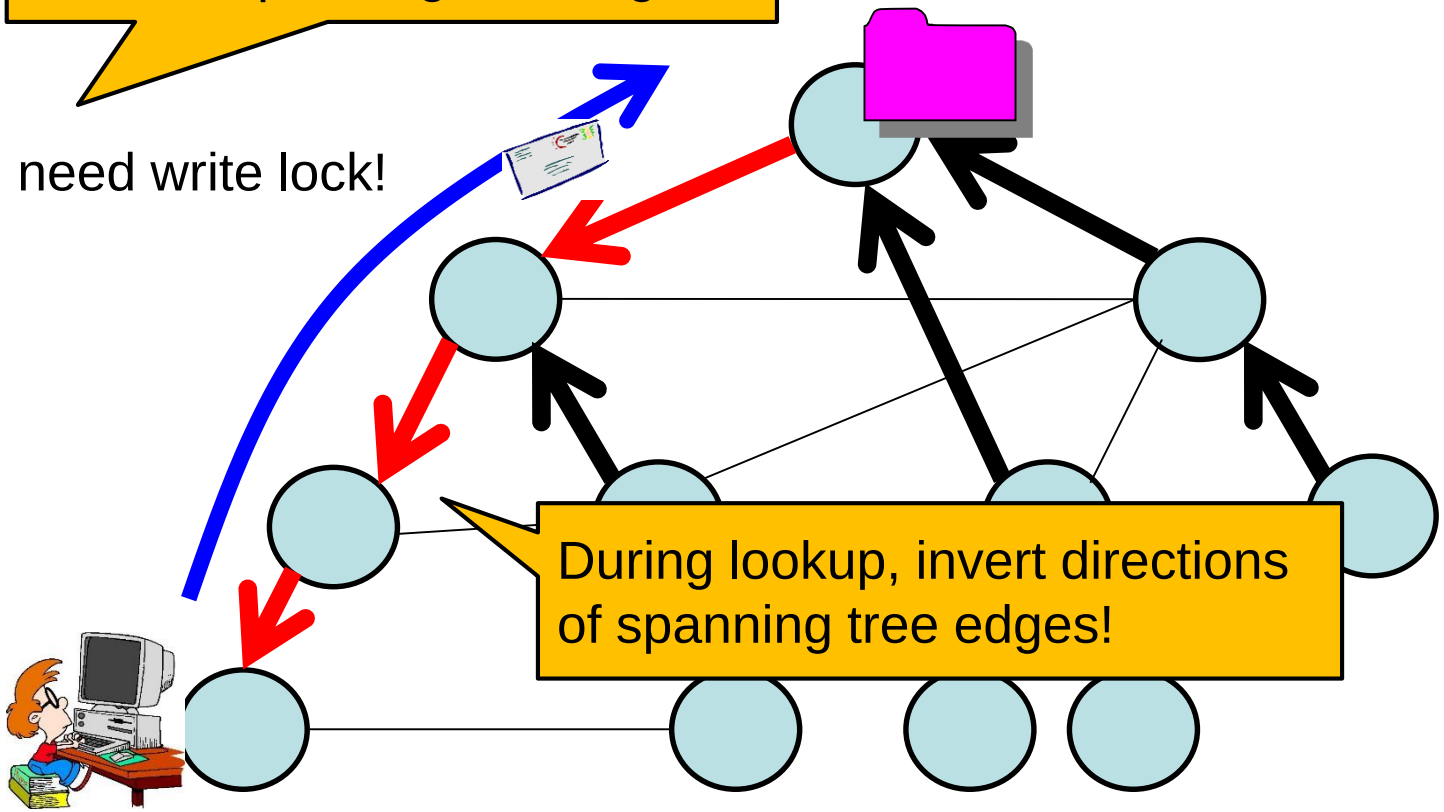
At Runtime

The Arrow Protocol

Find object is easy due to invariant: simply route along directed spanning tree edges!

Idea maintain invariant: directed spanning tree pointing to the object.

I need write lock!



During lookup, invert directions of spanning tree edges!

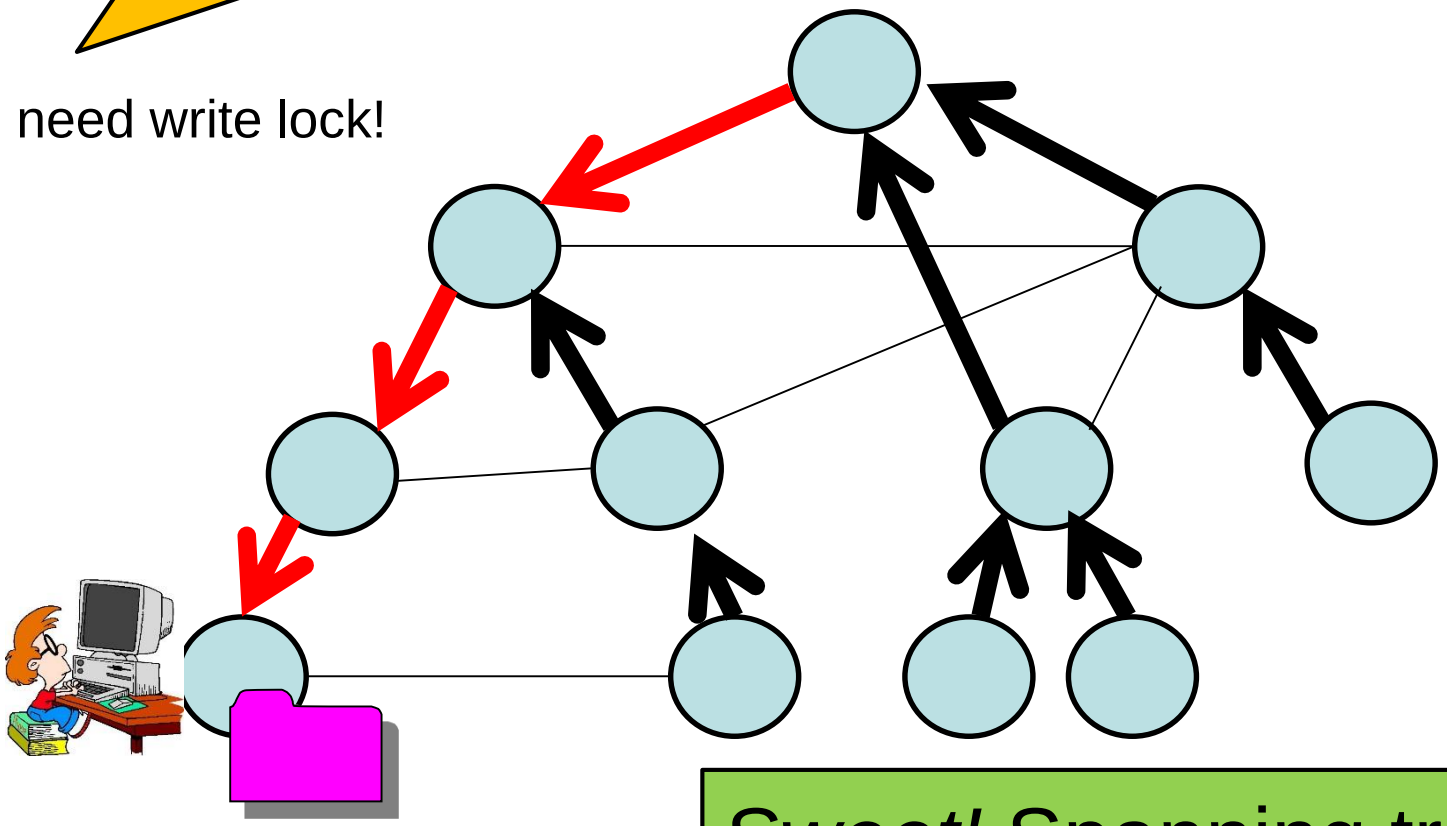
At Runtime

The Arrow Protocol

Find object is easy due to invariant: simply route along directed spanning tree edges!

Idea maintain invariant: directed spanning tree pointing to the object.

I need write lock!



At Runtime

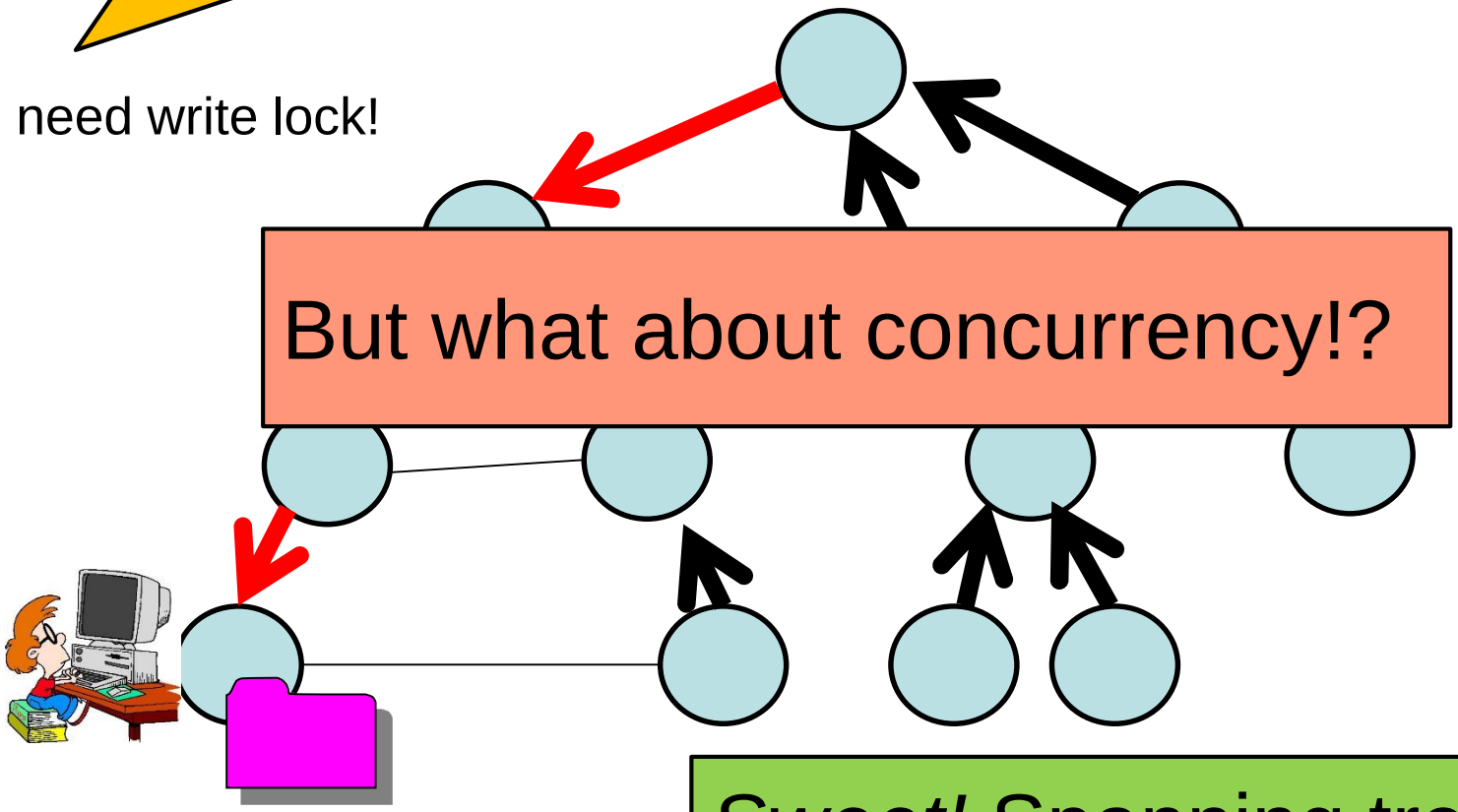
Sweet! Spanning tree rooted at object location again. ☺

The Arrow Protocol

Find object is easy due to invariant: simply route along directed spanning tree edges!

Idea maintain invariant: directed spanning tree pointing to the object.

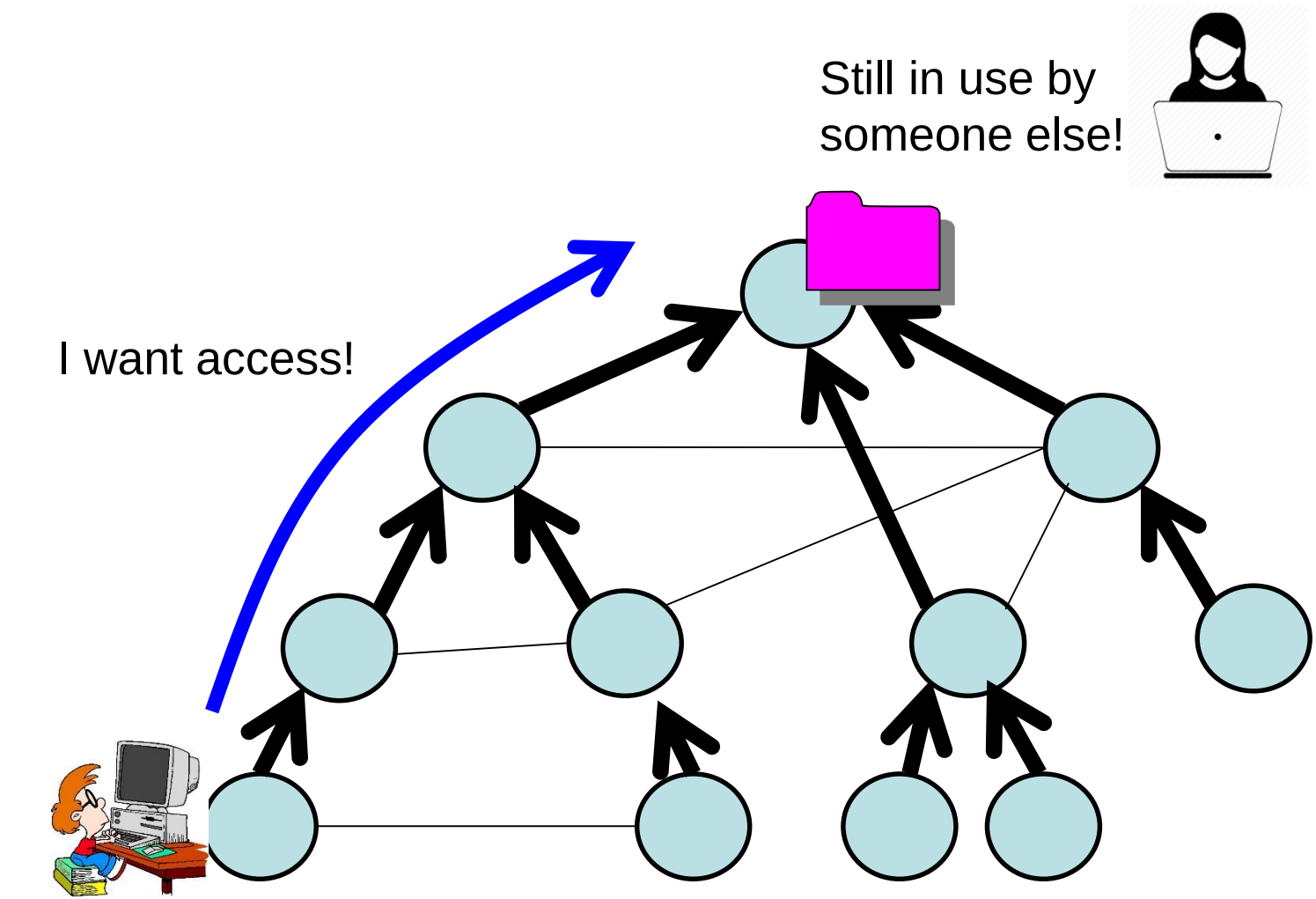
I need write lock!



Sweet! Spanning tree rooted at object location again. ☺

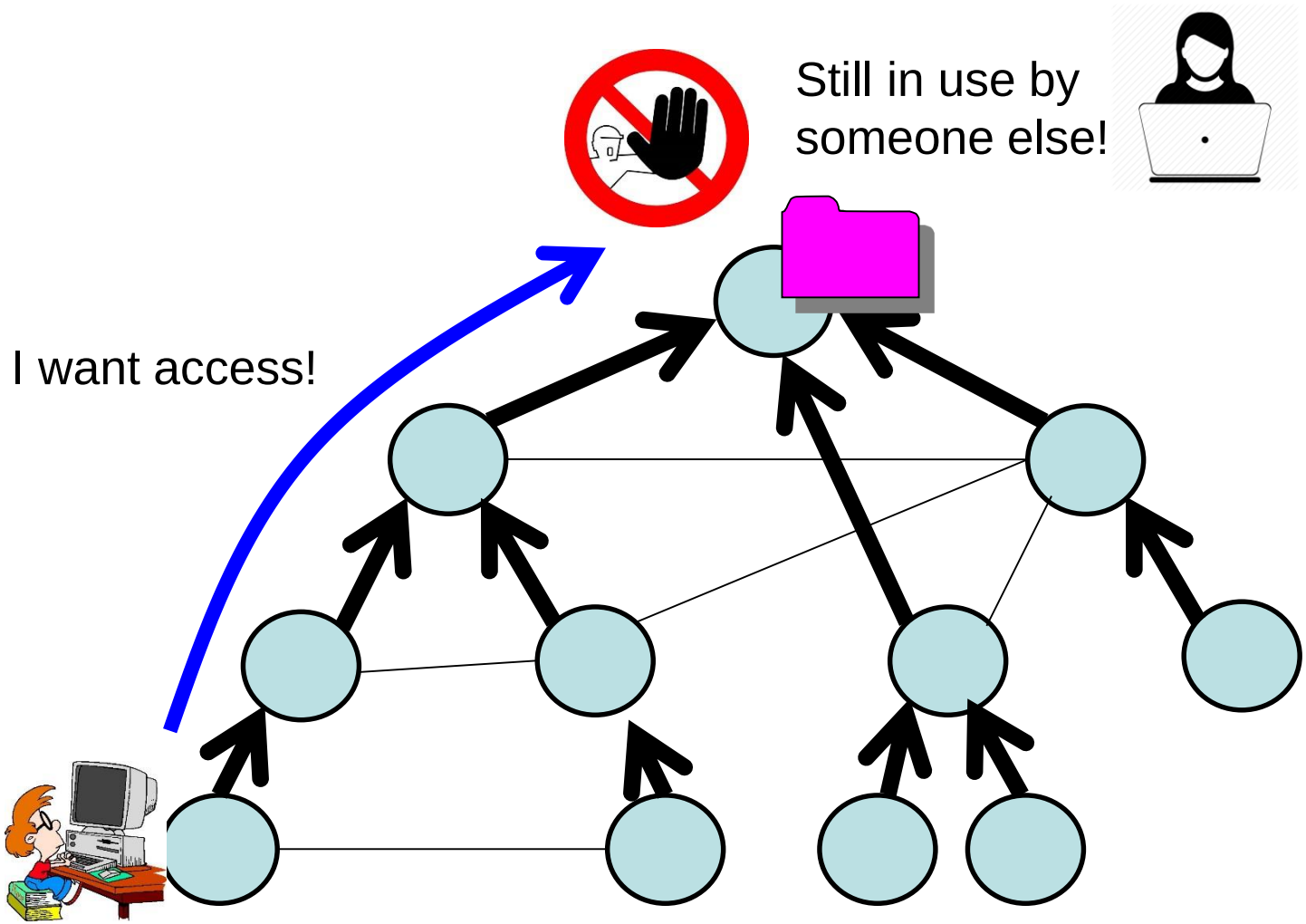
At Runtime

The Arrow Protocol: Concurrency



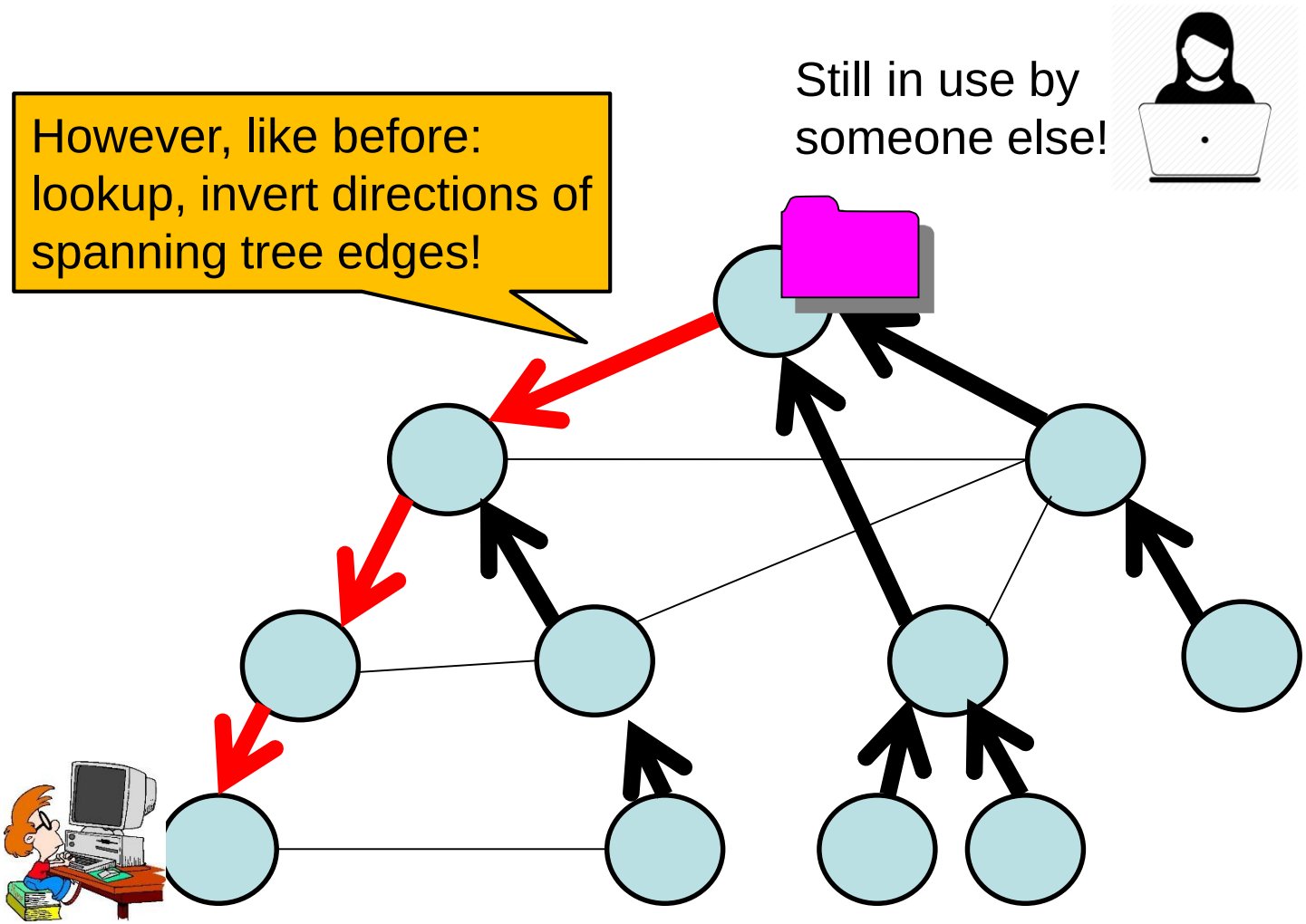
In the Beginning

The Arrow Protocol: Concurrency



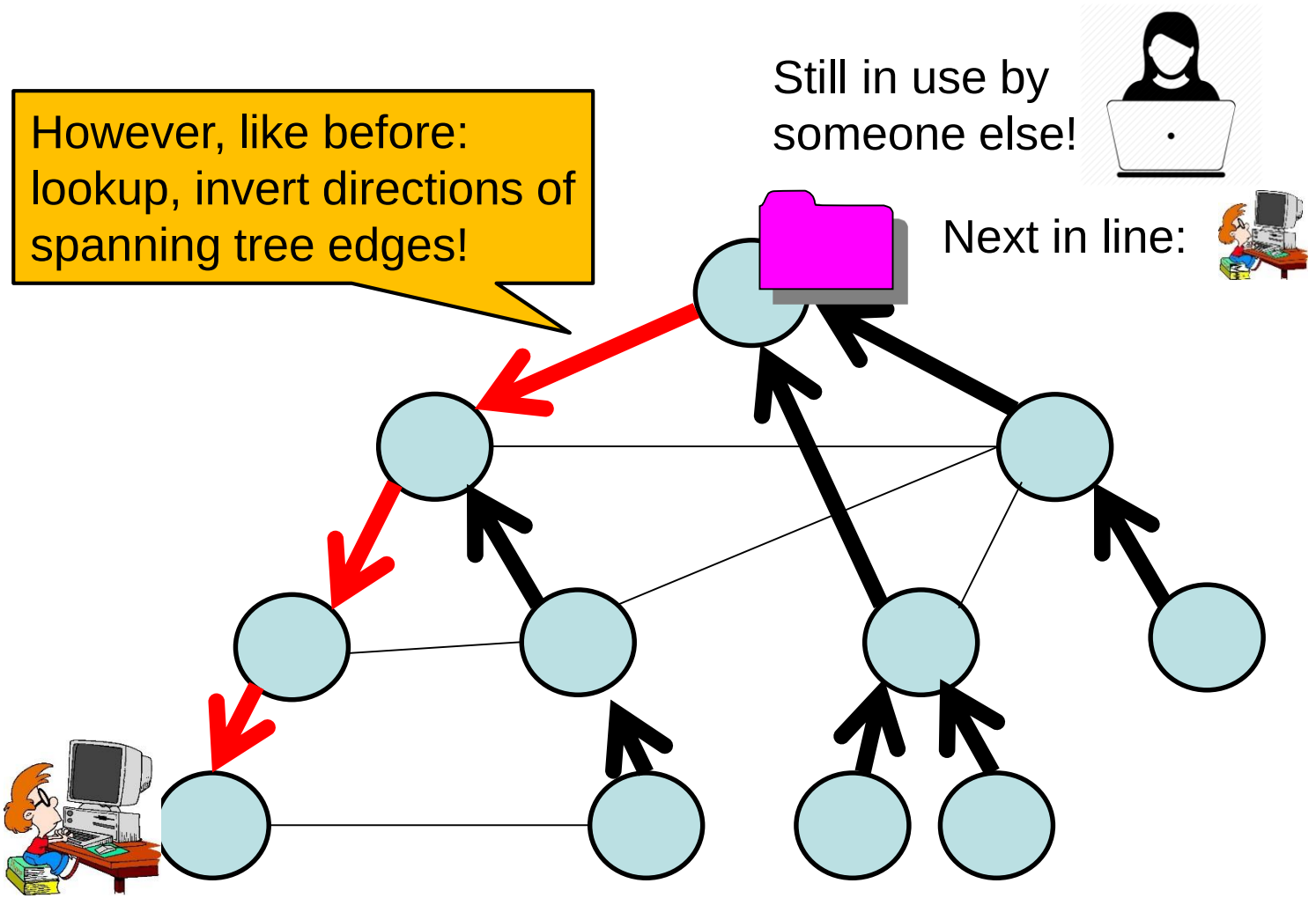
In the Beginning

The Arrow Protocol: Concurrency



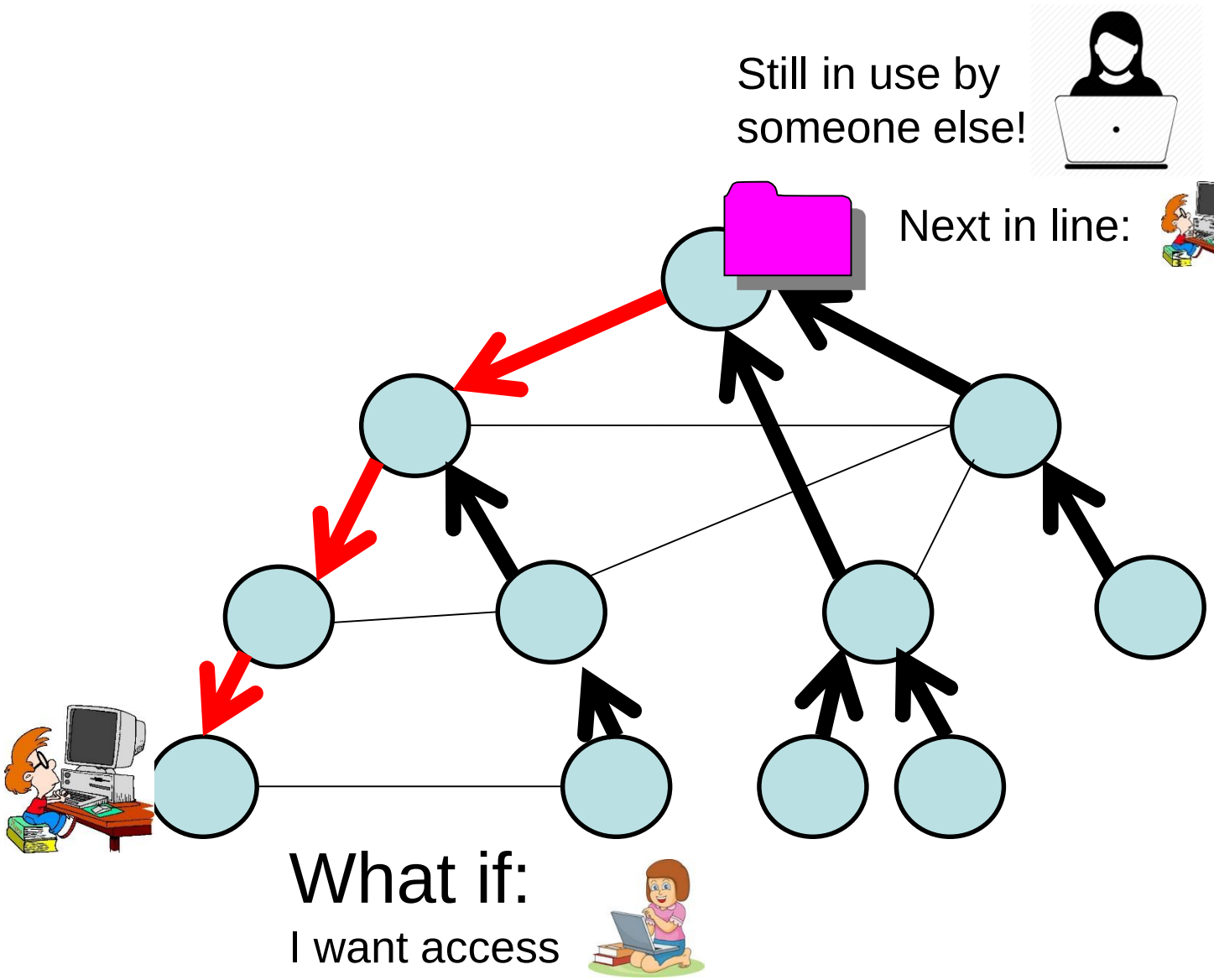
Phase 1

The Arrow Protocol: Concurrency



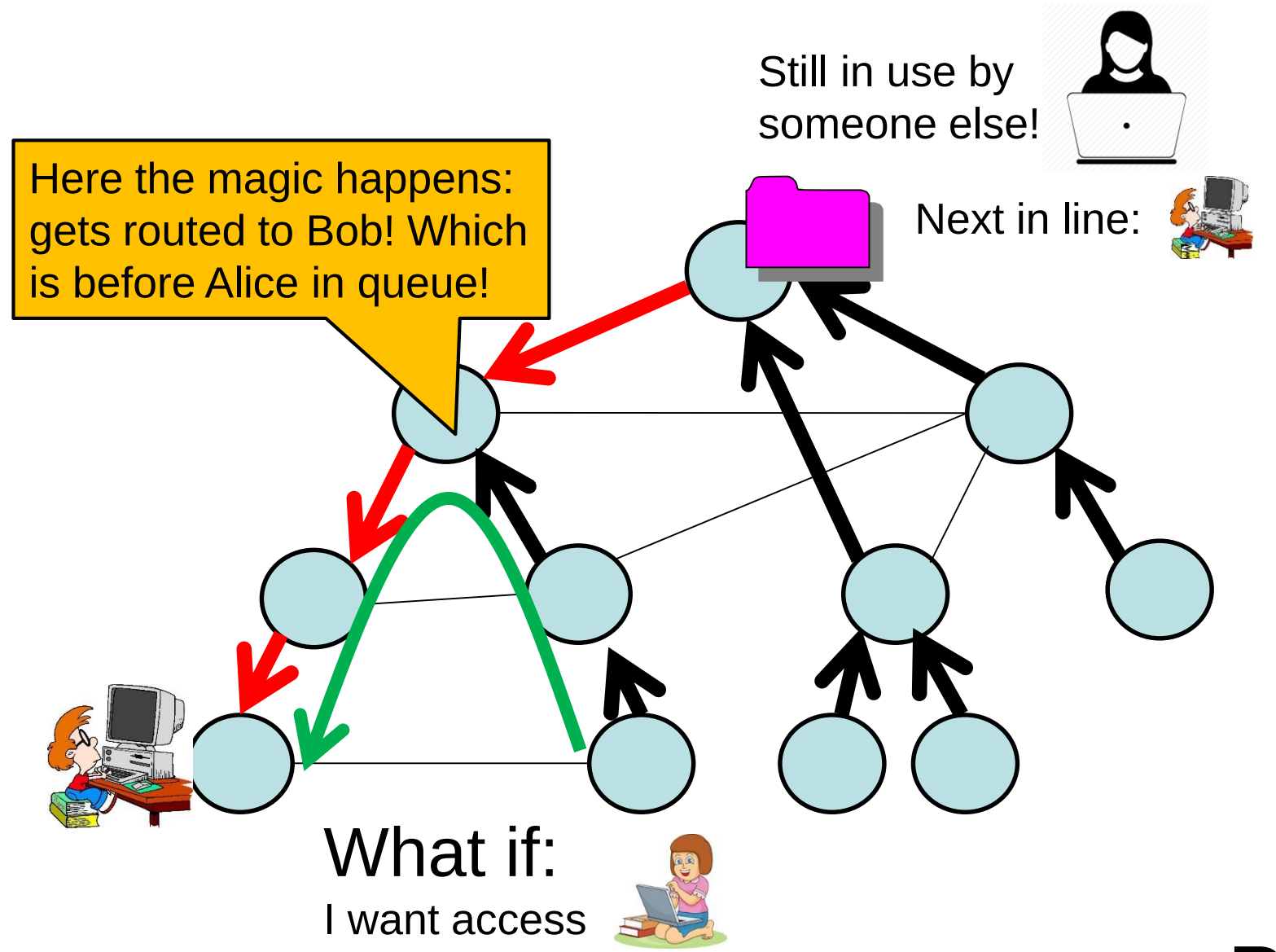
Phase 1

The Arrow Protocol: Concurrency



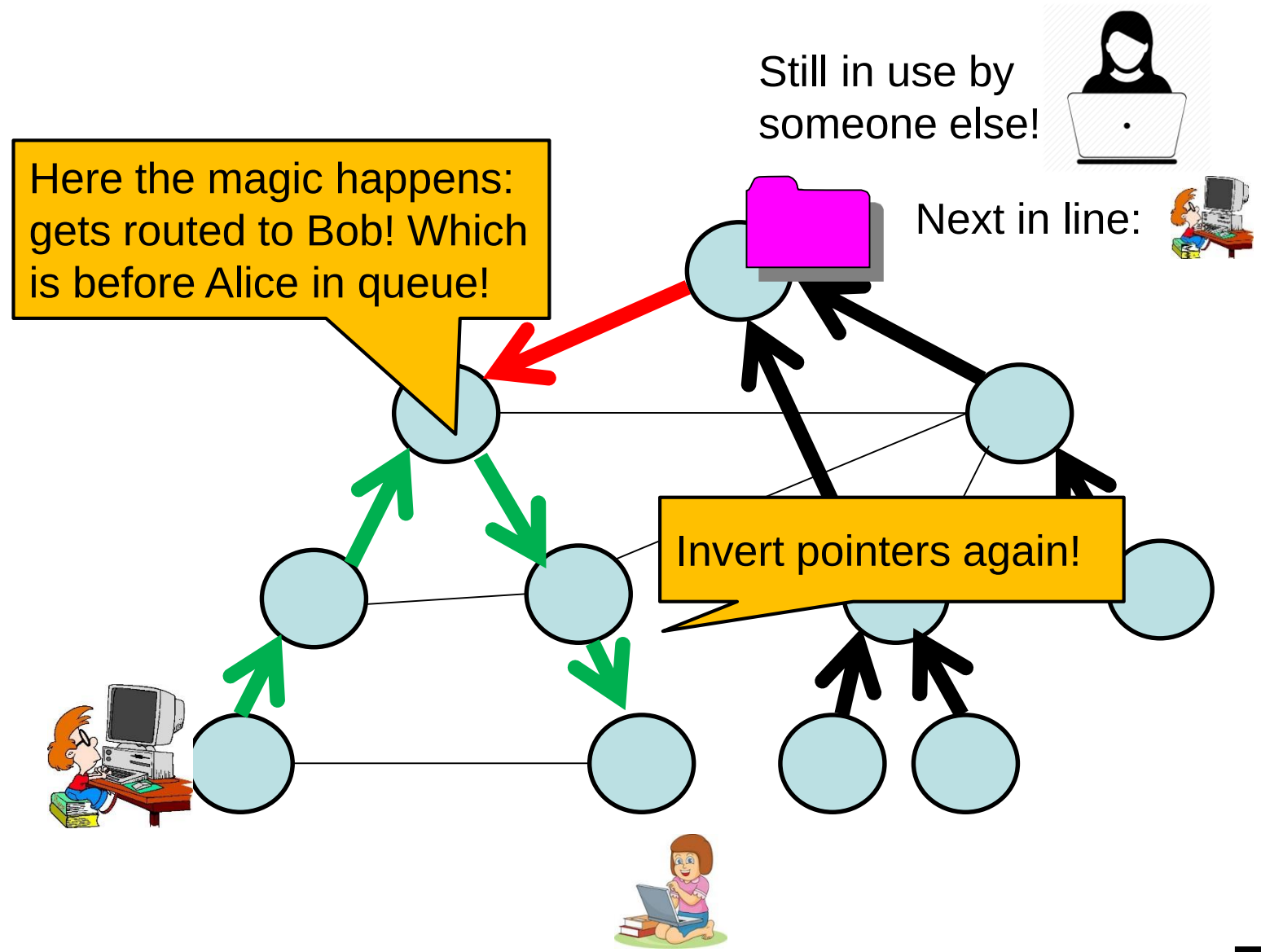
Phase 2

The Arrow Protocol: Concurrency



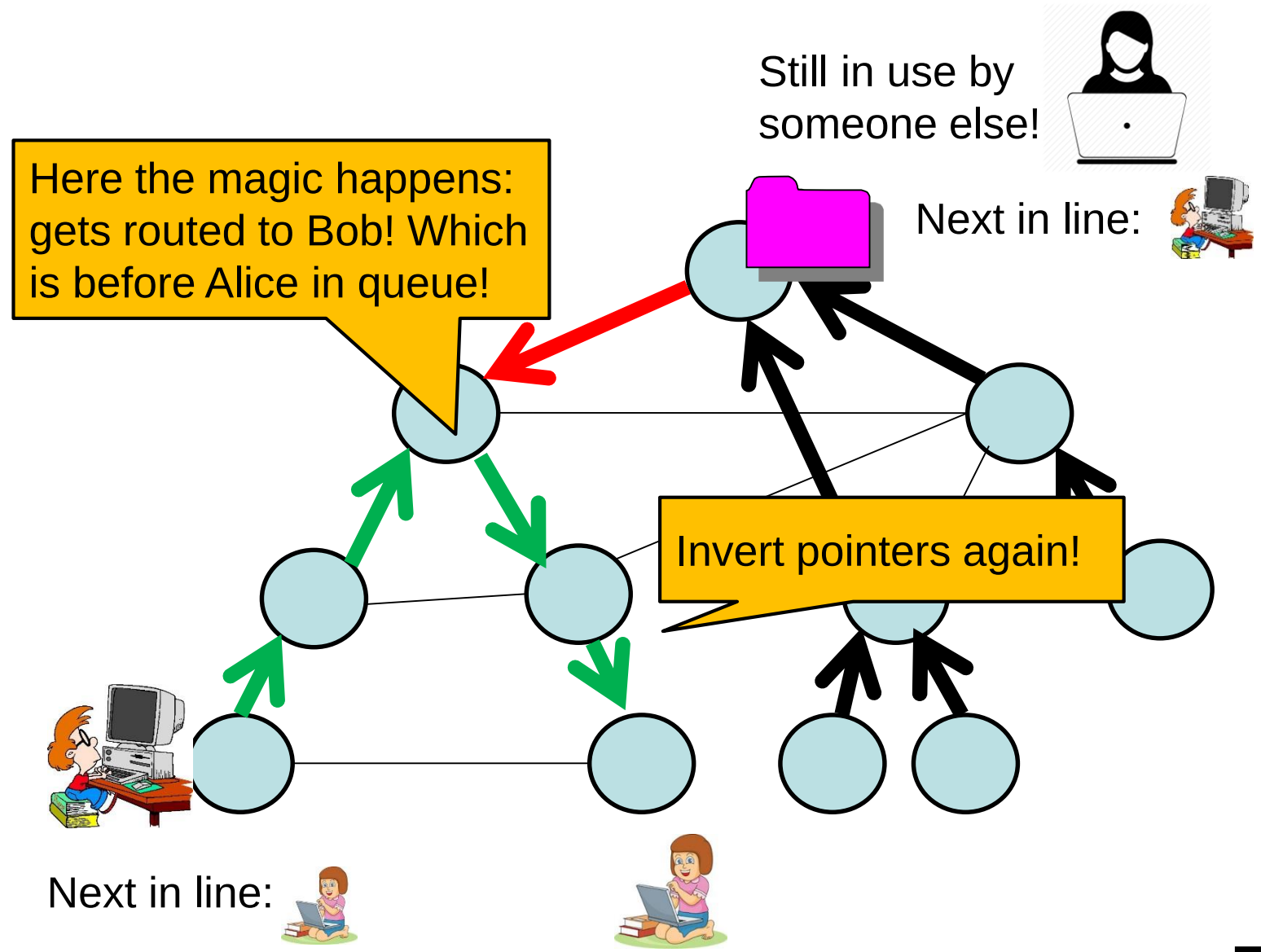
Phase 2

The Arrow Protocol: Concurrency



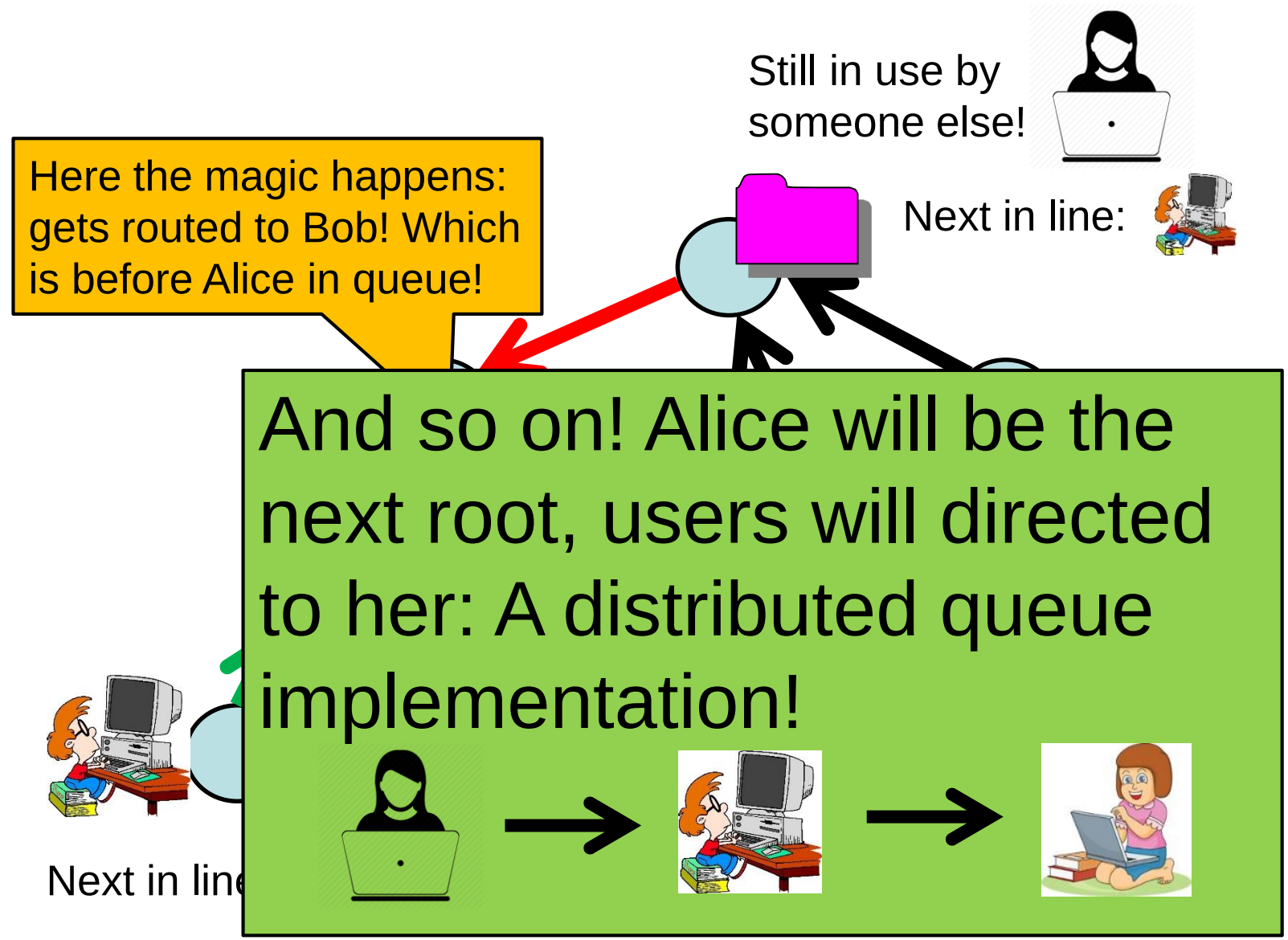
Phase 2

The Arrow Protocol: Concurrency



Phase 2

The Arrow Protocol: Concurrency



Remark: Of course, once object becomes free, it can be sent to Bob directly. (No link reversals, no spanning tree.)

ency

Still in use by someone else!

Next in line:

Next in line:

Arrow

Start Find Request at Node u:

```
1: do atomically
2:    u sends "find by u" message to parent node
3:    u.parent := u
4:    u.wait := true
5: end do
```

Upon w Receiving "Find by u" Message from Node v:

```
6: do atomically
7:    if w.parent ≠ w then
8:       w sends "find by u" message to parent
9:       w.parent := v
10:   else
11:      w.parent := v
12:      if not w.wait then
13:         send variable to u      // w holds var. but does not need it any more
14:      else
15:         w.successor := u        // w will send variable to u a.s.a.p.
16:      end if
17:   end if
18: end do
```

invert edge!

wait myself?

Upon w Receiving Shared Object:

```
19: perform operation on shared object
20: do atomically
21:    w.wait := false
22:    if w.successor ≠ null then
23:       send variable to w.successor
24:       w.successor := null
25:    end if
26: end do
```

- Message will only travel on static tree!
- And can never traverse an edge twice (in opposite direction).

Arrow

Arrow is correct: find() terminates with message and time complexity D, where D is the diameter of the spanning tree. Completely asynchronous and concurrent environments!

Proof.

- Each edge {u,v} in the spanning tree is in one of four states:
  - (A) u points to v, no message on the edge, v does not point to u
  - (B) Message on the move from u to v (no pointer along edge)
  - (C) v points to u, no message on edge, u does not point to v
  - (D) Message on the move from v to u (no pointer along edge)

QED

End of Lecture