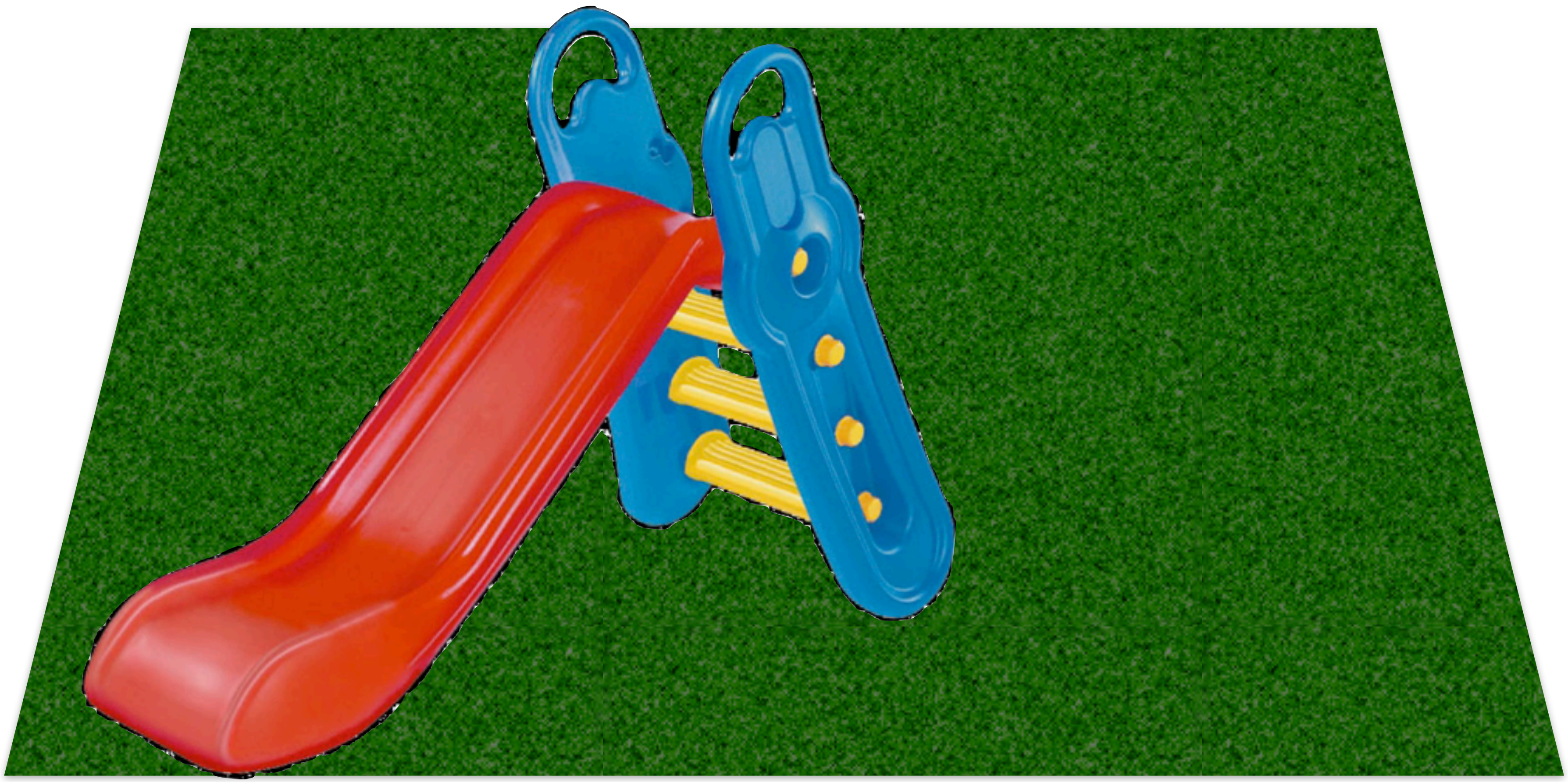
Asynchronous Shared-Memory Mutual Exclusion in $O(\log^2 \log n)$ RMRs

Michael A. Bender
Stony Brook & Tokutek, Inc

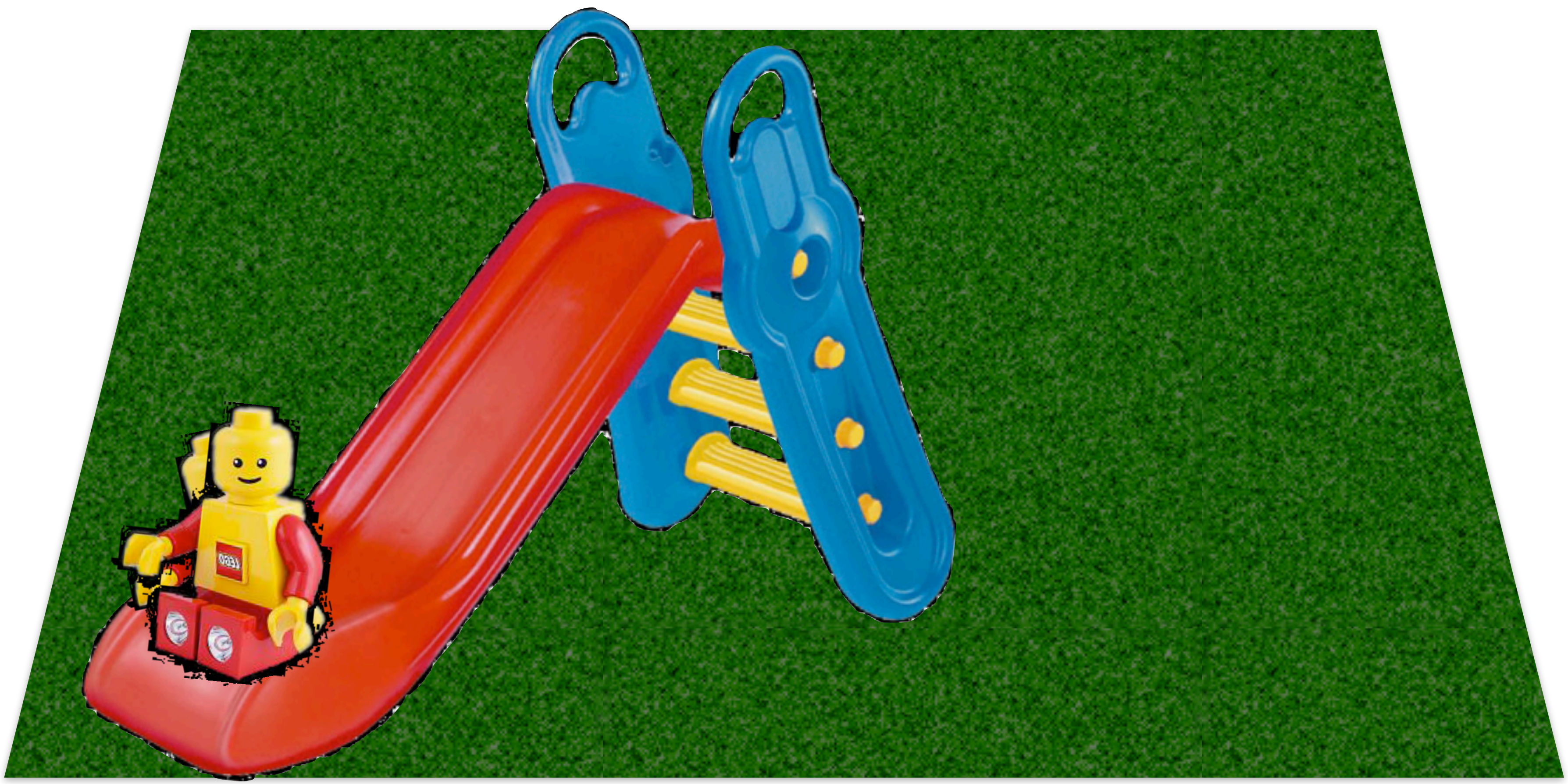
Seth Gilbert
National University of Singapore

How to share… efficiently

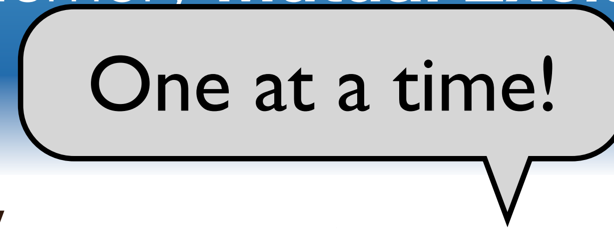


How to share... efficiently

How to share… efficiently



How to share… efficiently

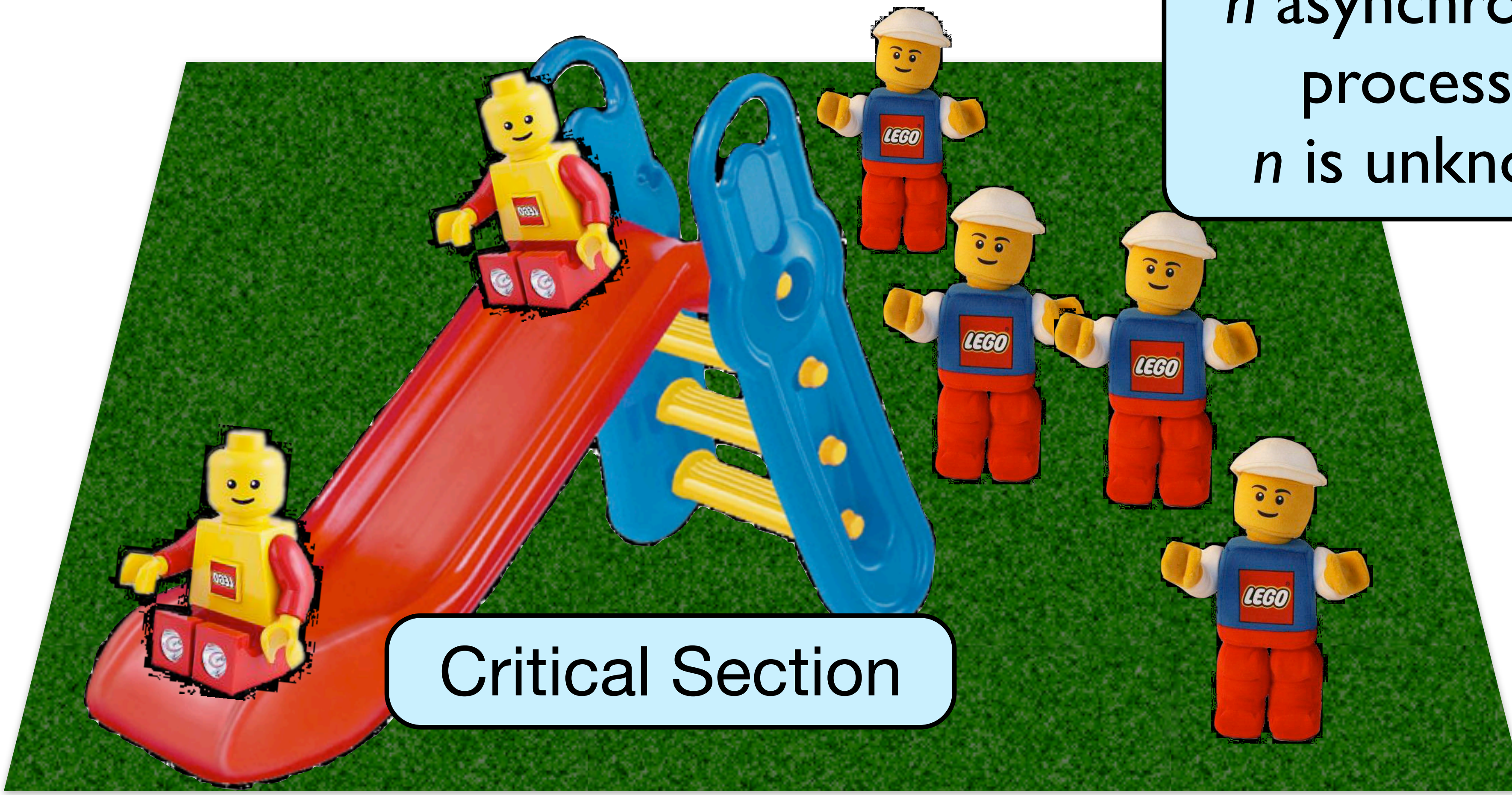


How to share… efficiently

How to share… efficiently

Asynchronous Shared-Memory **Mutual Exclusion** in $O(\log^2 \log n)$ RMRs

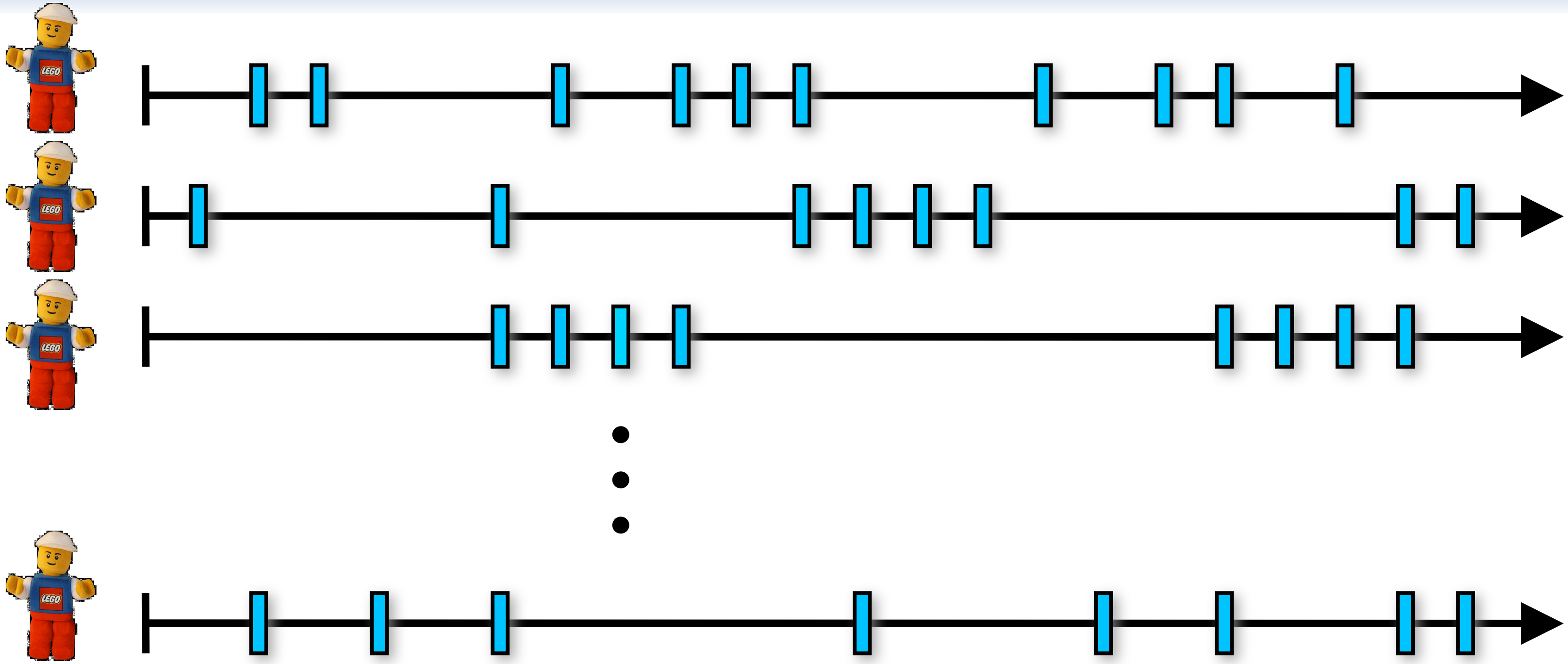
n asynchronous processes.
n is unknown.



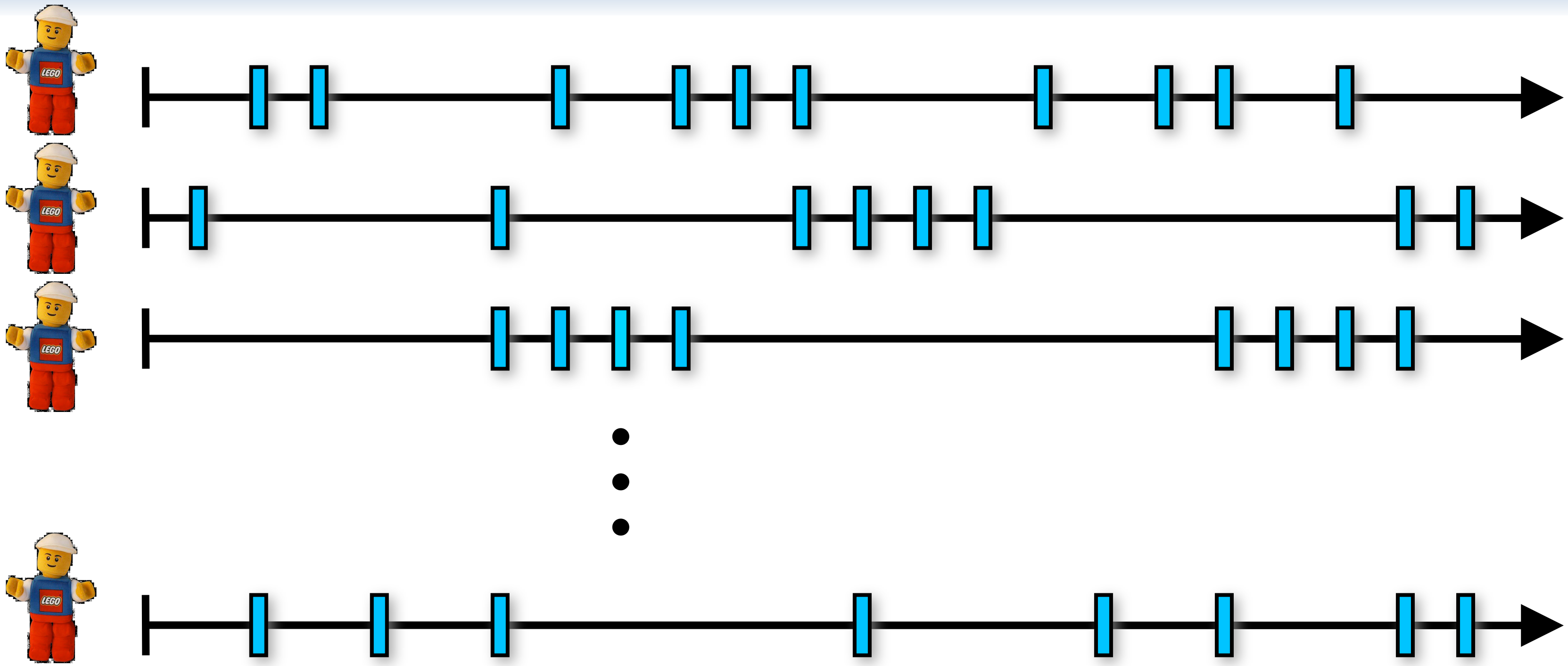
Objective: each process should pass through the critical section exactly once.

Asynchronous Shared-Memory **Mutual Exclusion** in $O(\log^2 \log n)$ RMRs

Asynchronous Shared-Memory Mutual Exclusion in $O(\log^2 \log n)$ RMRs



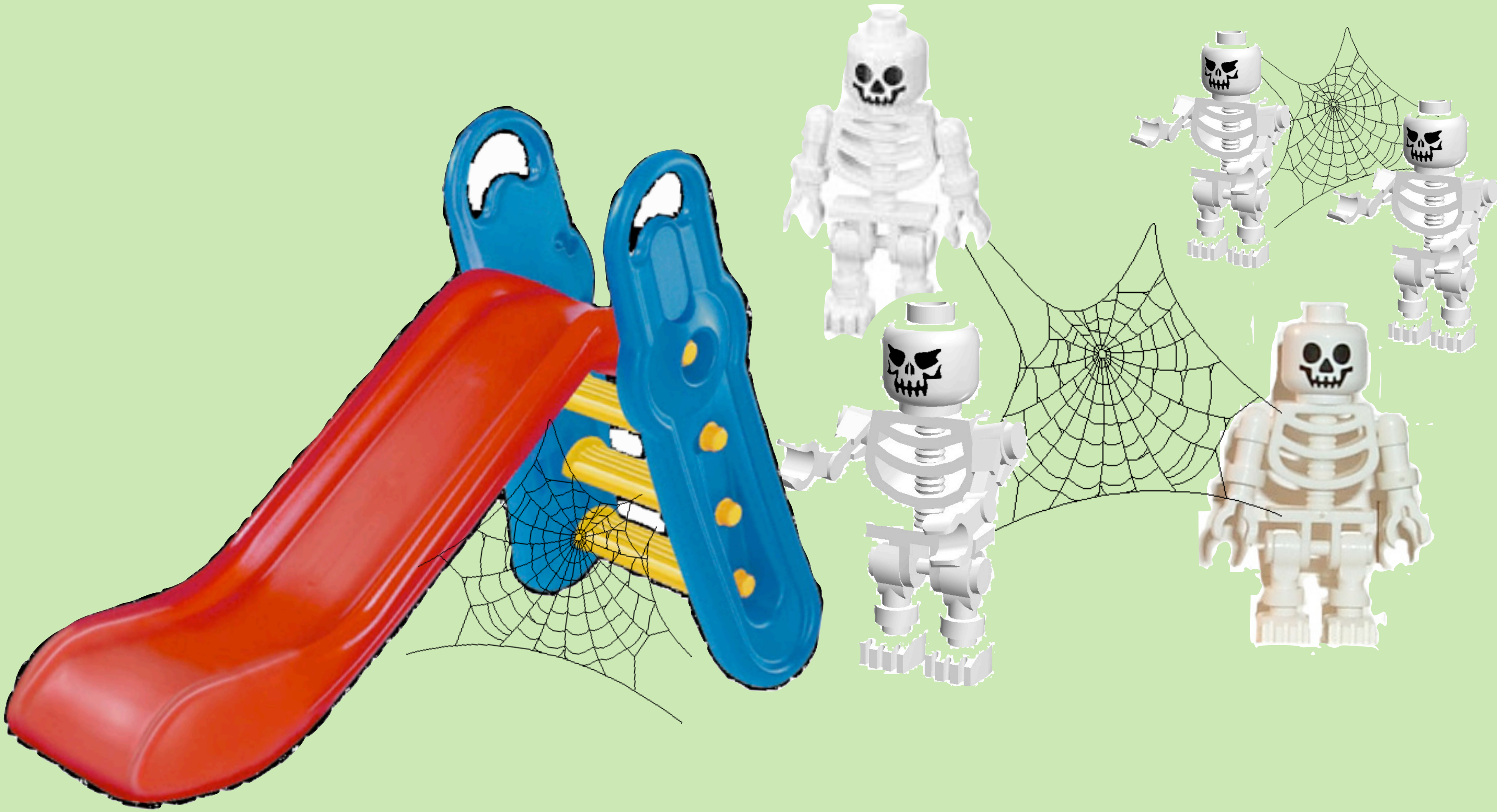
Asynchronous Shared-Memory Mutual Exclusion in $O(\log^2 \log n)$ RMRs



An “adversary” determines schedule.

- **Full-knowledge**: knows all but future coin flips.
- **Oblivious**: Determines schedule in advance.

We don't want deadlock.

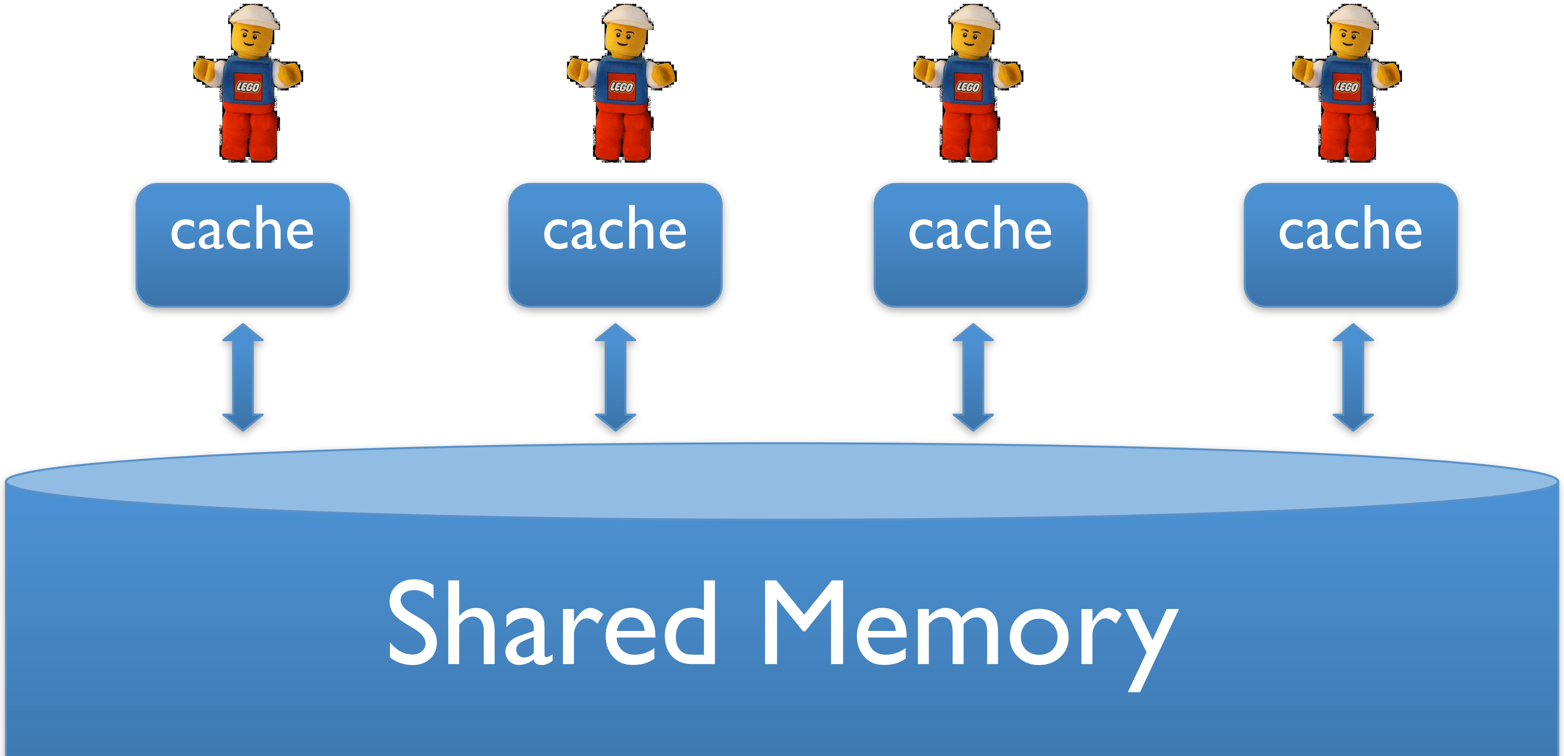


Cache-coherent shared memory:

- System keeps caches consistent.

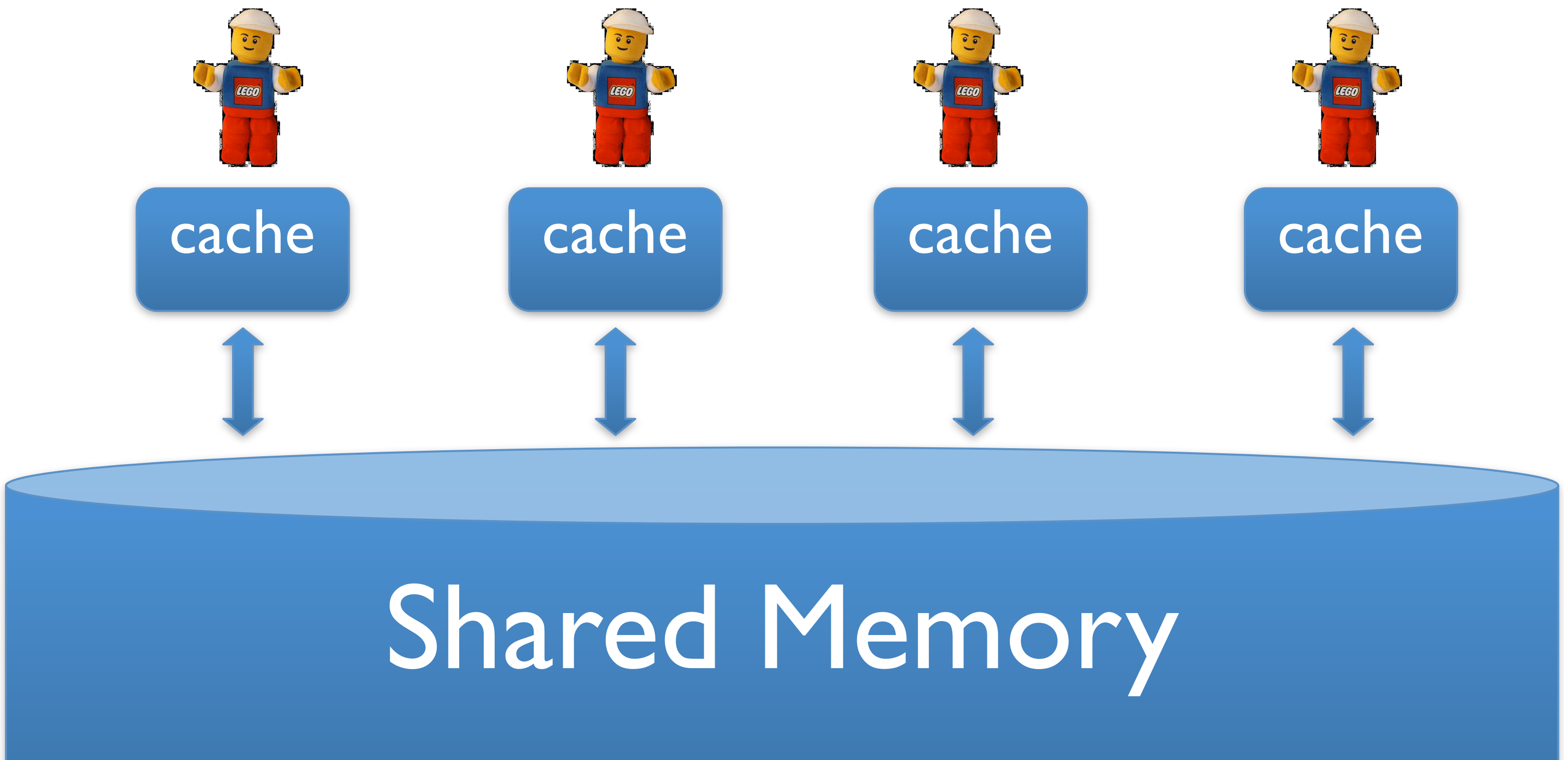
Cost model:

- Accessing local cache is effectively free.
- Accessing shared memory is expensive.



Asynchronous **Shared-Memory** Mutual Exclusion in $O(\log^2 \log n)$ **RMRs**

- Remote memory reference (atomic read/write): O(1).
- Spinning on a local variable is free,
  but costs 1 each time the variable changes.
- Compare and swap (CAS): O(1).
  [Golab et al.]. Uses atomic reads/writes + spinning.



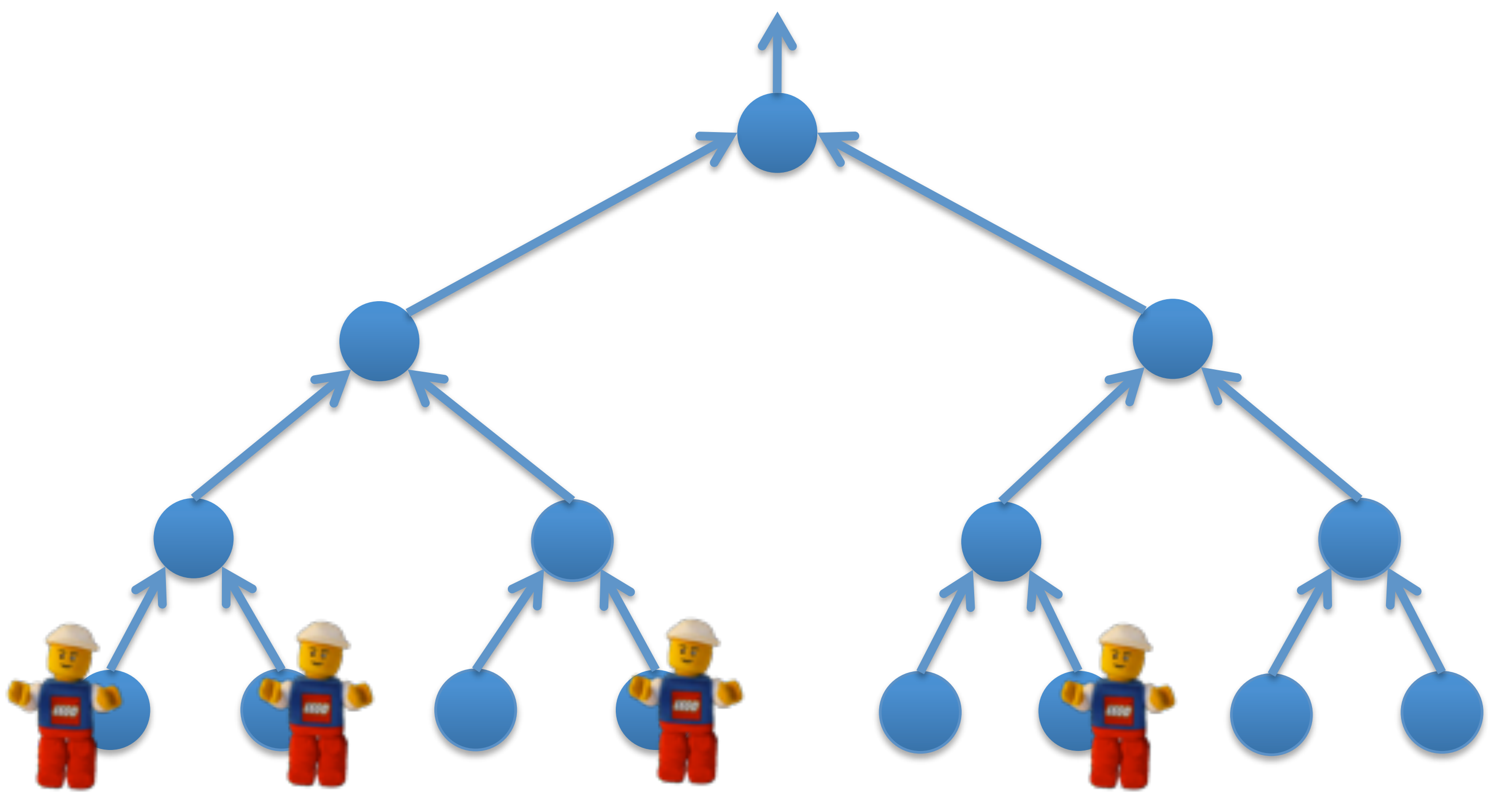
Naive Solution: O(*n*) RMRs per process.

- Protect the critical section with a lock

- Upon arrival: try to acquire (CAS) lock.
- Repeat:
  - Spin on lock until it becomes available.
  - Try to acquire (CAS) lock.

Better solution: O(log *n*) RMRs per process

- Maintain a tournament tree of locks.
- Processes compete to walk up tree.



Prior Results

→ $O(\log n)$ **(deterministic, tight)**

Yang and Anderson, "A fast, scalable mutual exclusion algorithm." 1995

→ $\Omega(\log n)$ **(deterministic, tight)**

Fan and Lynch, "An $\Omega(n \log n)$ lower bound on the cost of mutual exclusion." 2006

Attiya, Hendler, Woelfel, "Tight RMR lower bounds for mutual exclusion and other problems." 2008

→ $O(\log n / \log\log n)$ **(randomized, tight for adaptive?)**

Hendler and Woelfel, "Randomized mutual exclusion in $O(\log n / \log\log n)$ RMR." 2009

Our Results

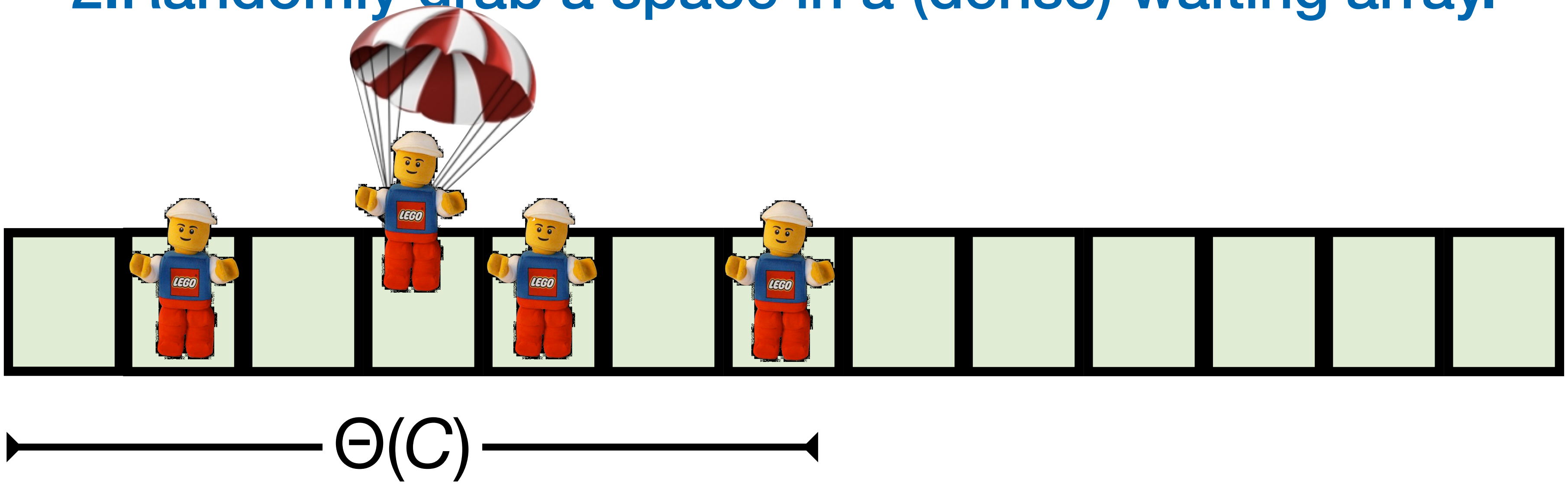
- New mutual exclusion algorithm with:
  - $O(\log^2 \log n)$ RMRs.
  - Randomized, subject to an oblivious adversary.
  - Each process enters the critical section whp.

- Incomparable with previous results because:
  - Weaker adversary: oblivious, not adaptive.
  - Liveness: guaranteed with high probability (instead of deterministic).

High-level Idea: Mutex ≈ Contention Resolution

Upon arriving:

1. Increment a process counter.
2. Randomly grab a space in a (dense) waiting array.

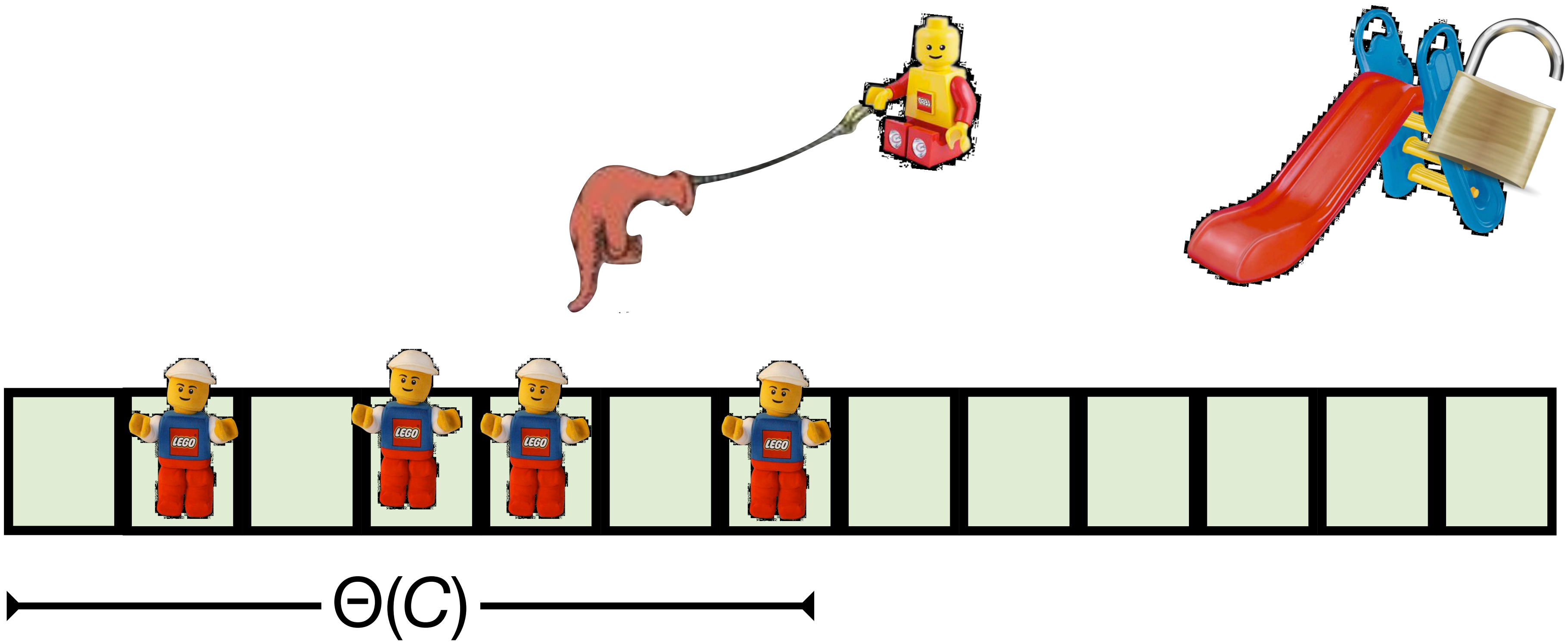


3. Try to grab mutex lock.
4. Sleep until awakened.

High-level Idea: Mutex ≈ Contention Resolution

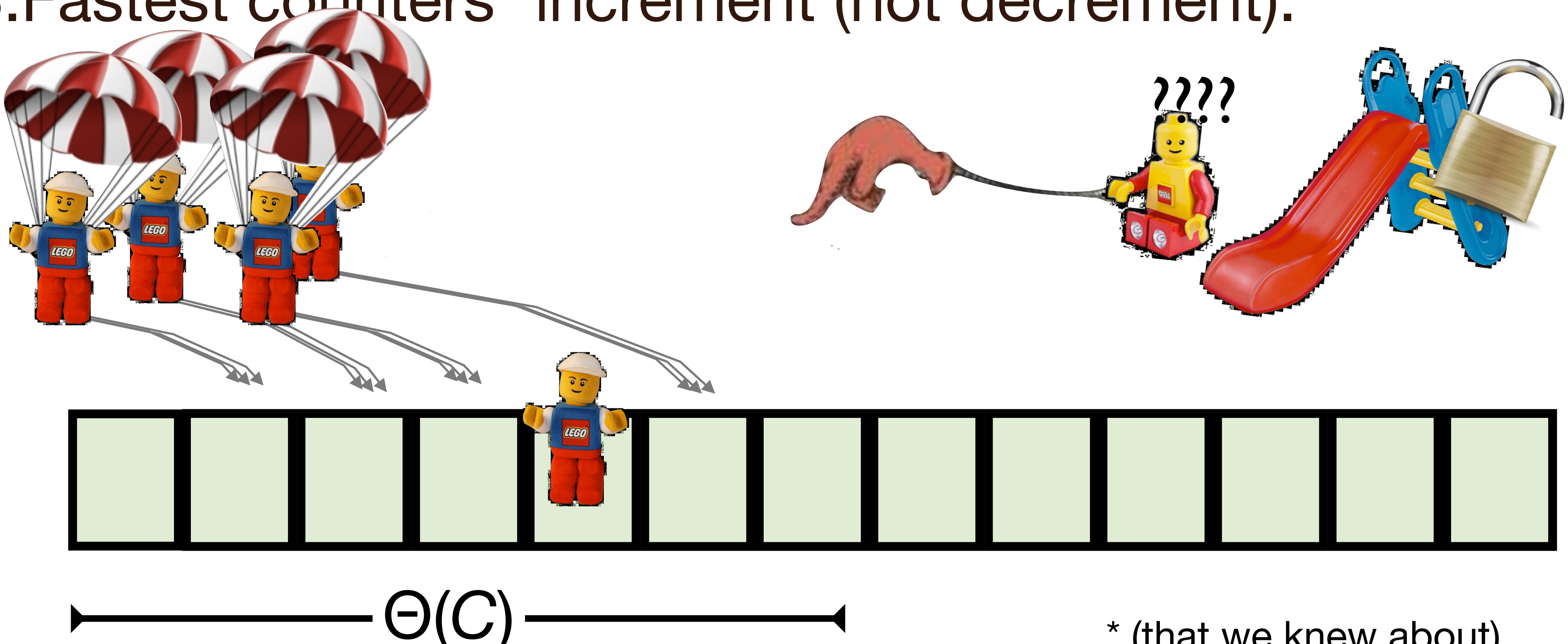
When departing:

1. Decrement the process counter.
2. Release the mutex lock.
3. Randomly pick a successor from the waiting array.



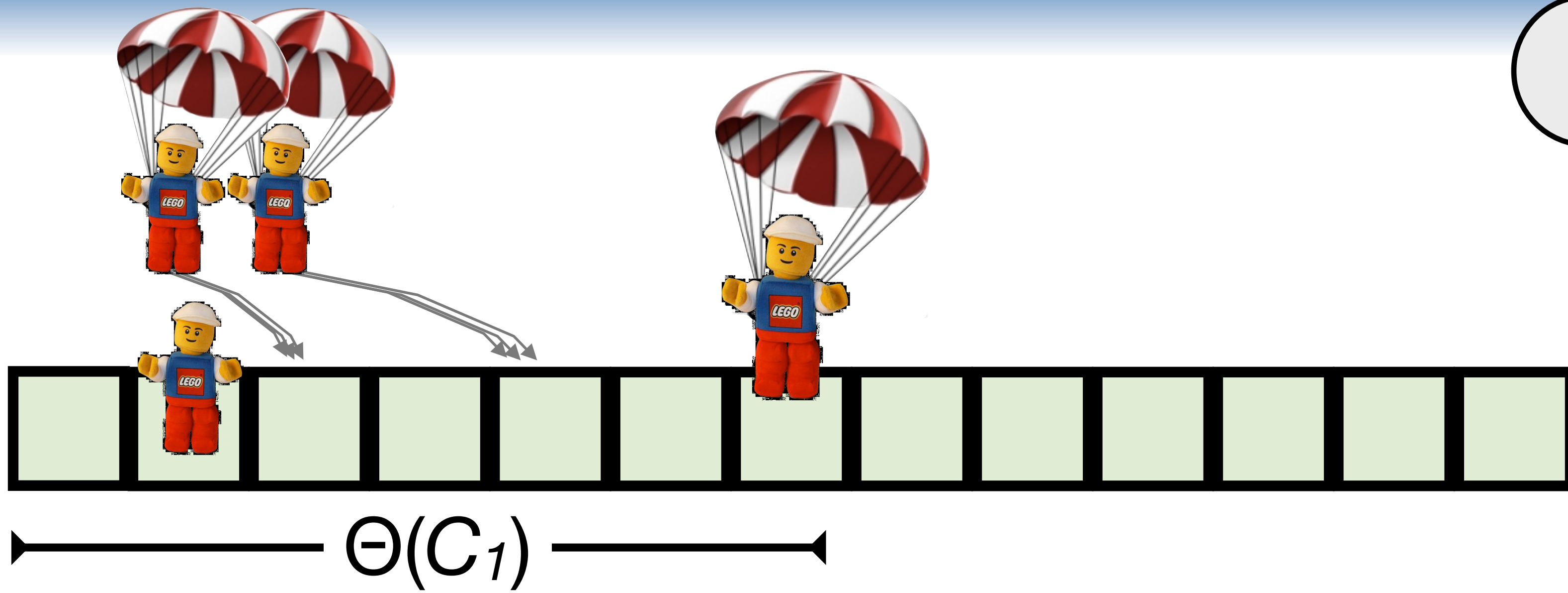
Problems with High-level Idea

1. The array could be sparse if many procs have “arrived” but not joined the array. Sparse array ⇒ slow to find successor.
2. Counting isn't cheap.
3. Fastest counters* increment (not decrement).

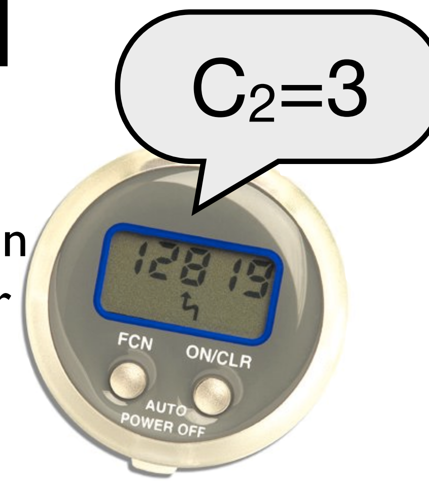


* (that we knew about)

(1) Dealing with Sparse Arrays



array-join
counter



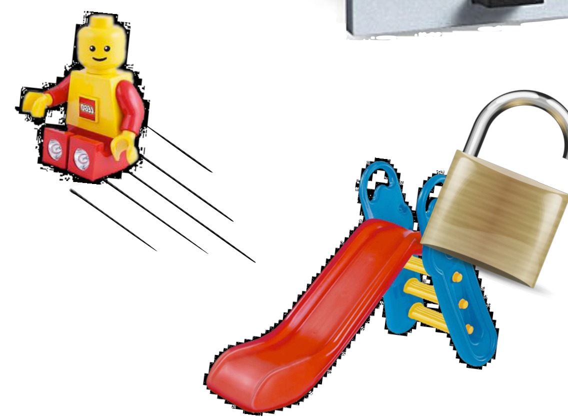
Upon arriving:

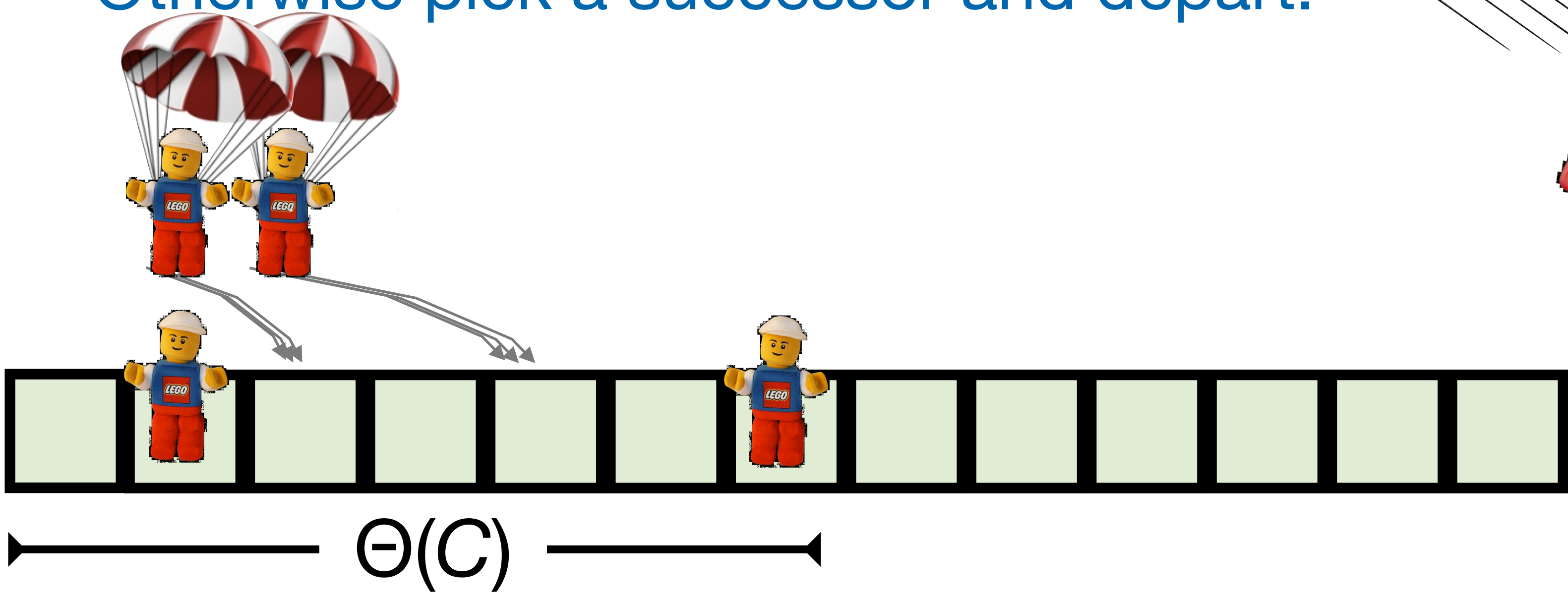
1. Increment process counter.
2. Randomly grab a space in a (dense) waiting array.
3. **Increment array-join counter.**
4. Try to grab mutex lock and then sleep.

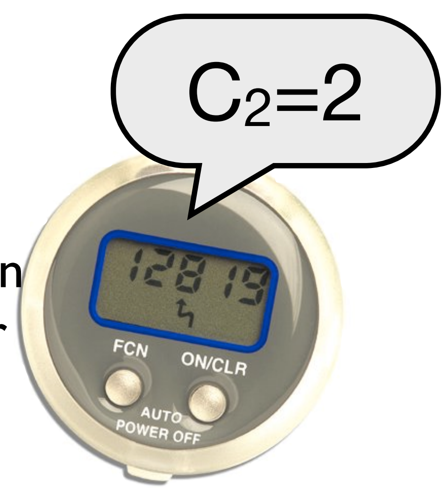
(1) Dealing with Sparse Arrays

When departing:

1. Decrement both counters.
2. Release the mutex lock.
3. **If $C_1>C_2/2$ then depart.**
   Otherwise pick a successor and depart.







array-join
counter

(2) Fast Approximate Counters

Approximate counting (standard trick):

- *increment*:
  - Choose **cnt** with probability $\mathbf{1/2^{cnt}}$
  - Write **cnt** to max-register
- read:
  - **cnt** = read-max-register
  - return $\mathbf{2^{cnt}}$.

max value = log(n)
Cost: O(loglog n) [AAC'09]

- Example: 64 processes
  - With constant probability, at least one process chooses 6
  - With constant probability, no process chooses a value larger than 6
  - => With constant probability, counter returns 64.
  - => Constant factor approximation

(2) Fast Approximate Counters

Approximate counting (standard trick):

- State:
  - Bounded counters C[1], C[2], C[3], ..., C[log n]
  - max-register: max-value log(n)

- *increment*:
  - Choose **cnt** with probability **$1/2^{cnt}$**
  - Increment counter **C[cnt]**.
  - If **C[cnt]** > log(n), then write **cnt** to max-register

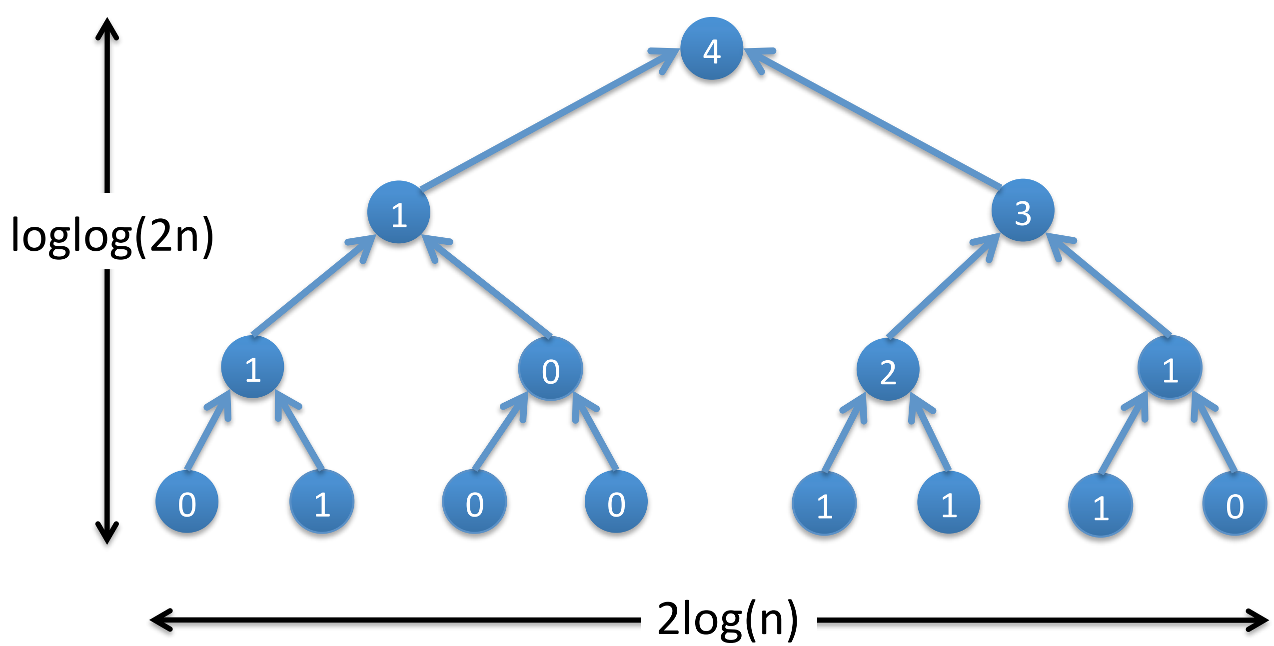
- read:
  - **cnt** = read-max-register
  - return **$\log(n) \cdot 2^{cnt}$**.

(2) Fast Approximate Counters

Approximate counting (standard trick):

- Cost of [AAC'09] counter: O(log n*loglog n)
  - One leaf per process

- Small tweak: 2log(n) leaves total
  - On increment, choose a random leaf



(2) Fast Approximate Counters

Approximate counting (standard trick):

- State:
  - Bounded counters C[1], C[2], C[3], ..., C[log n]
  - max-register: max-value log(n)
- *increment*:
  - Choose **cnt** with probability **$1/2^{cnt}$**
  - Increment counter **C[cnt]**.
  - If **C[cnt]** > log(n), then write **cnt** to max-register
- read:
  - **cnt** = read-max-register
  - return **$\log(n)*2^{cnt}$**.

Cost: O(loglog n)

(2) Fast Approximate Counters

Approximate counting (standard trick):

- If (> log n) increments) then return value is a constant-factor approximation with high probability.

- What about small values?
  - Use small-counter with max-value log(n).
  - Increment small-counter and ...
  - Read both small-counter and ...

(2) Fast Approximate Counters

Approximate counting (standard trick):

- State:
  - Bounded counters C[1], C[2], C[3], ..., C[log n]
  - max-register: max-value log(n)

- *increment*:
  - Choose **cnt** with probability **$1/2^{cnt}$**
  - Increment counter **C[cnt]**.
  - If **C[cnt]** > log(n), then write **cnt** to max-register

- read:
  - **cnt** = read-max-register
  - return **$\log(n)*2^{cnt}$**.

Constant-factor approximation, whp

(3) Counters that Increment/Decrement

Replace process counter with two counters.

process counter

≈

arrival counter
(increments)

-

departure counter
(decrements)

(3) Counters that Increment/Decrement

Replace process counter with two counters.

≈

-

process counter

arrival counter
(increments)

departure counter
(decrements)

Problem: Does this still work, since we use approximate counters?

?
≈

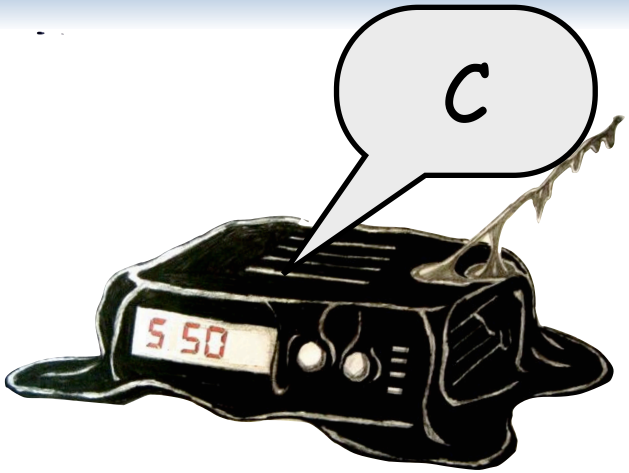
-

approx process counter

arrival counter
(approx after poylog n)

departure counter
(exact)

(3) Counters that Increment/Decrement



approx process counter

$\stackrel{?}{\approx}$

approx arrival counter (increments)

-

exact departure counter (decrements)

Good approximation when:

- $C_{arrive} > 2\ C_{leave}$.
- C_{arrive}, C_{leave} = polylog n.

(3) Counters that Increment/Decrement

approx process counter

approx arrival counter (increments)

exact departure counter (decrements)

Good approximation when:

- $C_{arrive} > 2\ C_{leave}$.
- C_{arrive}, C_{leave} = polylog n.

Poor approximation otherwise.

(3) Counters that Increment/Decrement

approx process counter

approx arrival counter (increments)

exact departure counter (decrements)

Good approximation when:

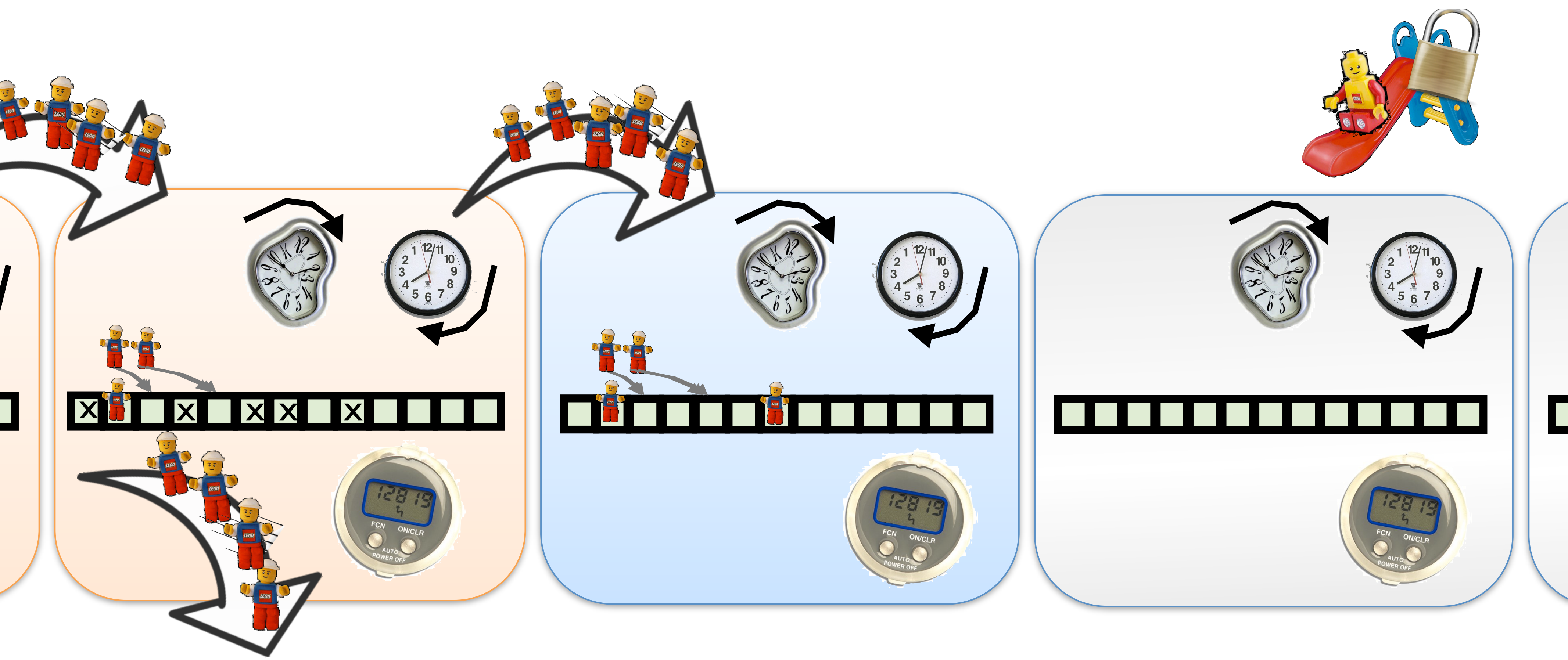
- $C_{arrive} > 2\ C_{leave}$.
- C_{arrive}, C_{leave} = polylog n.

Poor approximation otherwise.

When the approximation gets bad... reset counter.

Computation Proceeds in Epochs

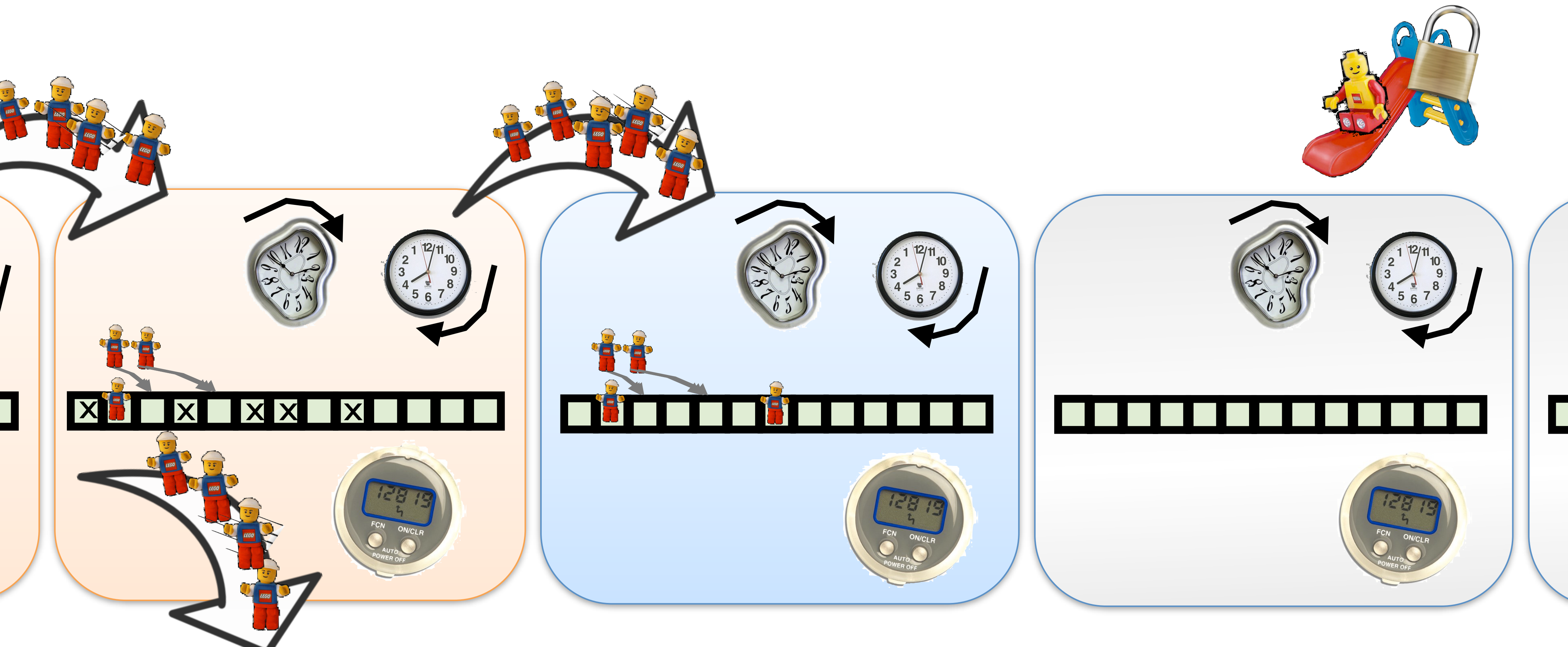
Each epoch uses a new copy of the data structure.



Computation Proceeds in Epochs

Each epoch uses a new copy of the data structure.

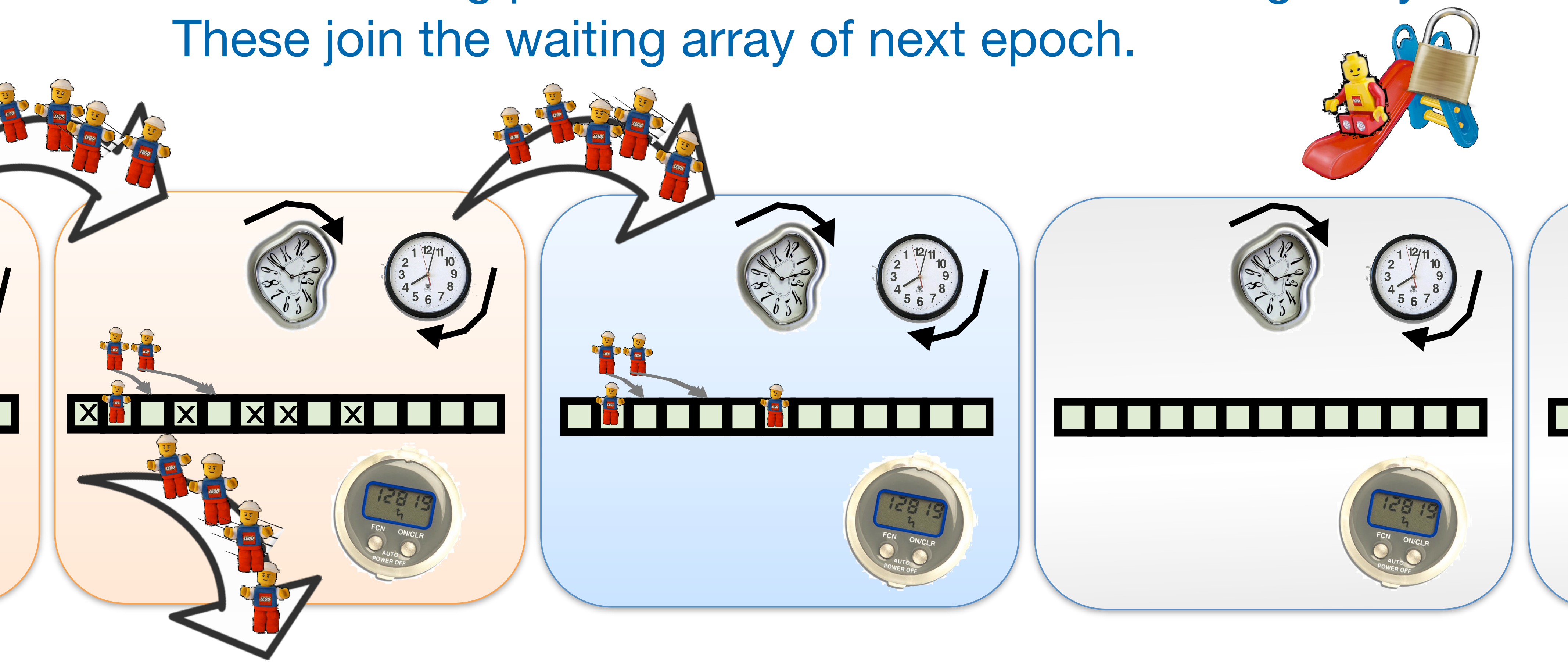
- An O(1)-fraction of procs finish in each epoch.
- The remaining procs are kicked out of the waiting array. These join the waiting array of next epoch.



Computation Proceeds in Epochs

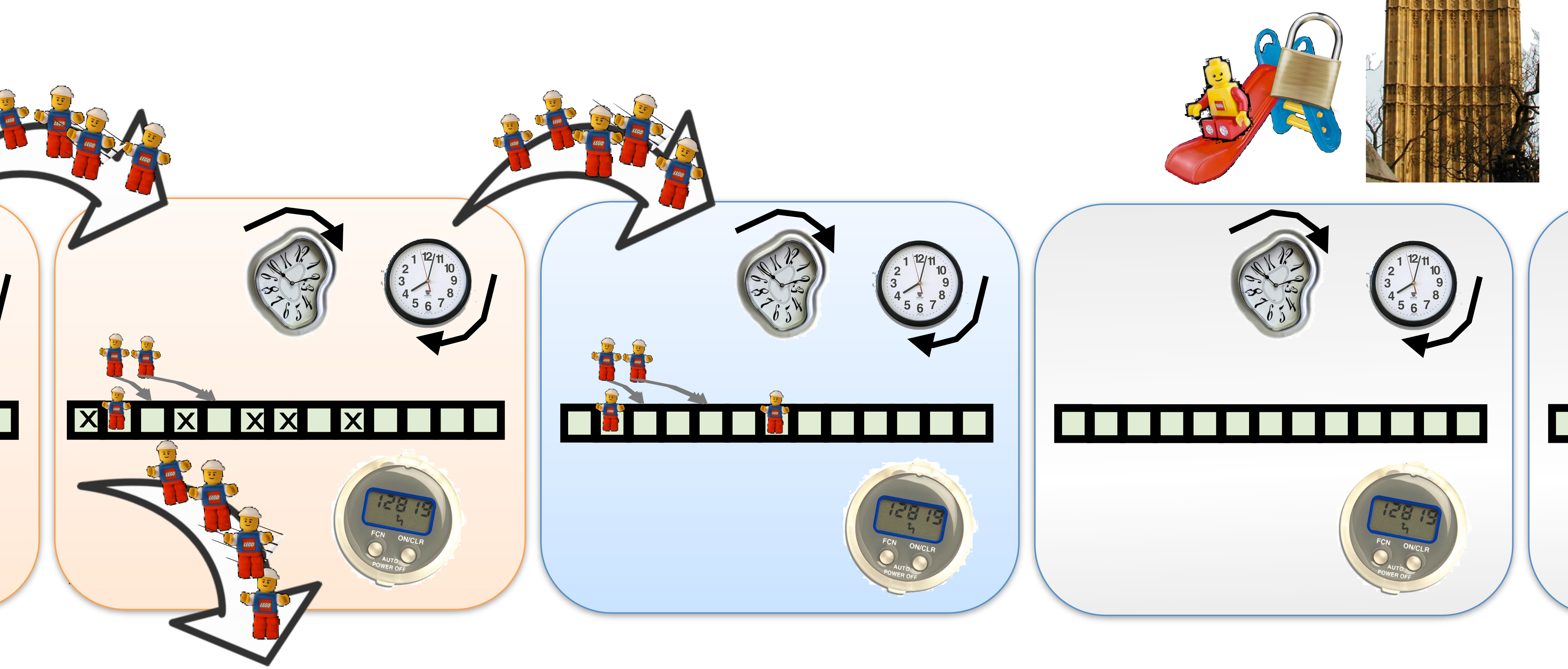
The cost to rejoin a waiting array is amortized against the procs that complete in the epoch.

- An O(1)-fraction of procs finish in each epoch.
- The remaining procs are kicked out of the waiting array. These join the waiting array of next epoch.



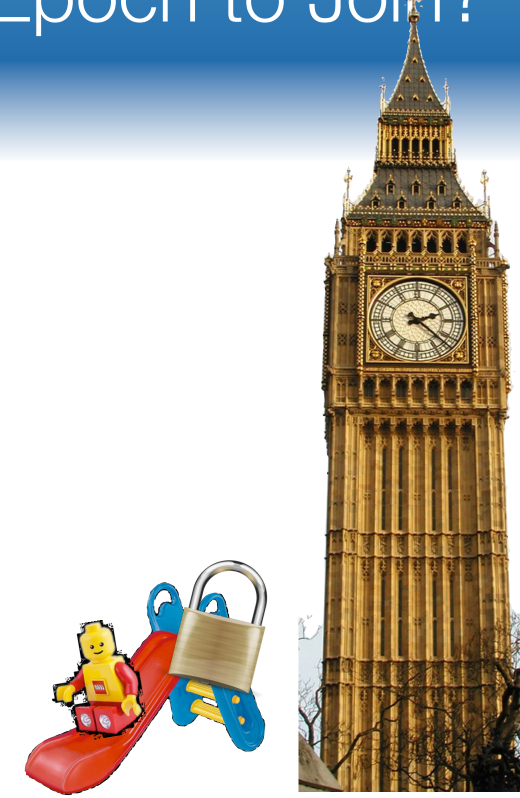
How Does a Process Determine Which Epoch to Join?

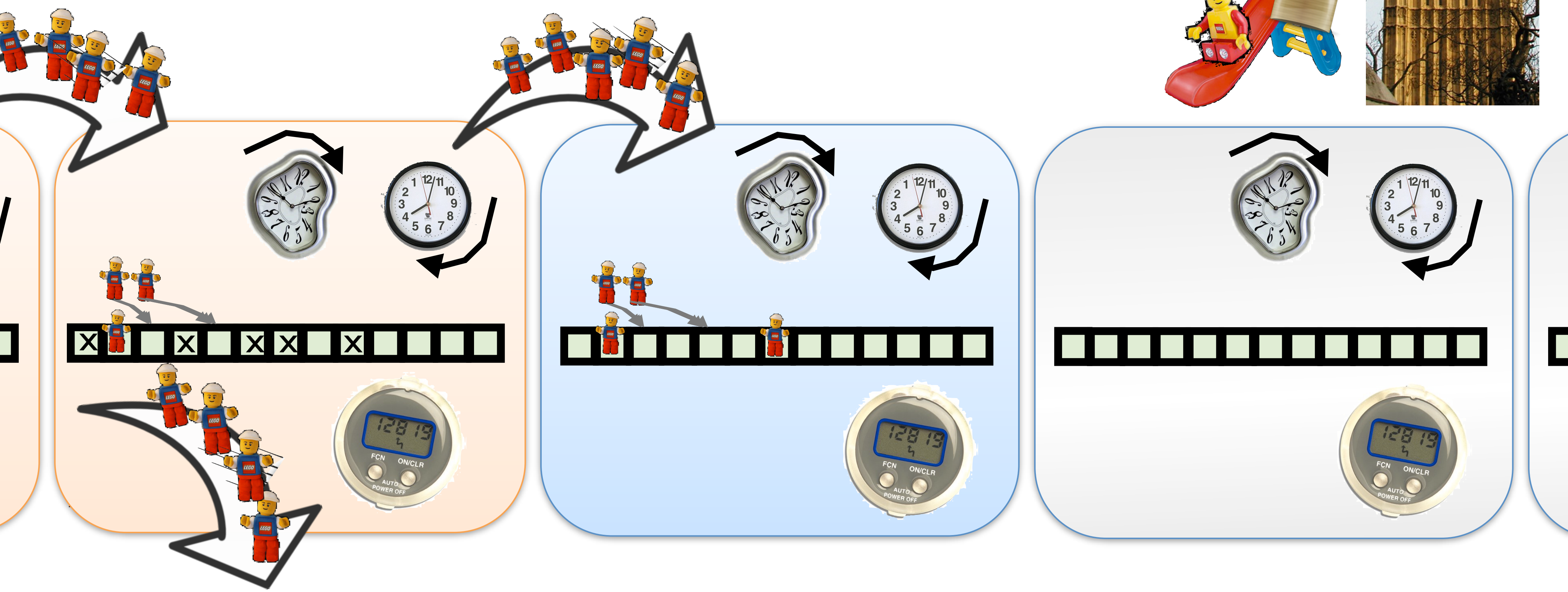
An epoch counter.



How Does a Process Determine Which Epoch to Join?

So a process arrives, reads, the counter, and joins current epoch.

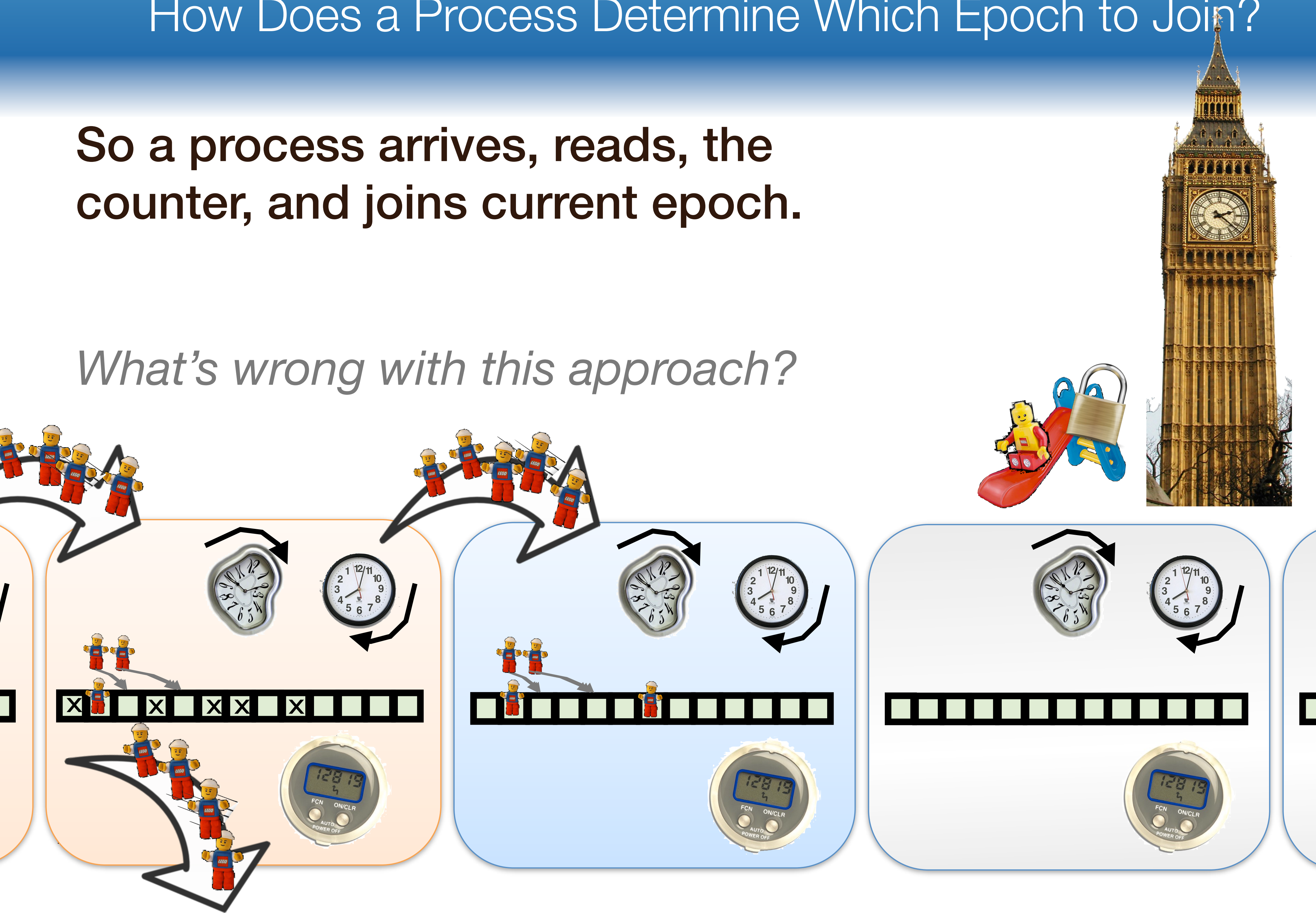




How Does a Process Determine Which Epoch to Join?

So a process arrives, reads, the counter, and joins current epoch.

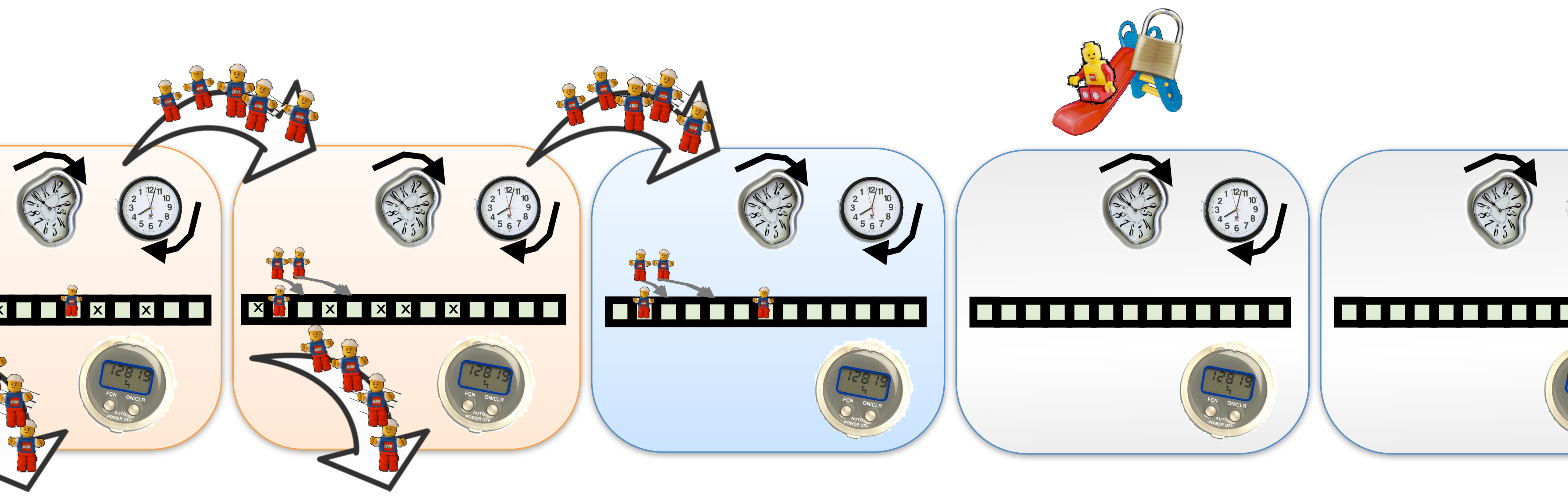
What's wrong with this approach?



How Does a Process Determine Which Epoch to Join?

Problem: what amortizes cost to read counter?

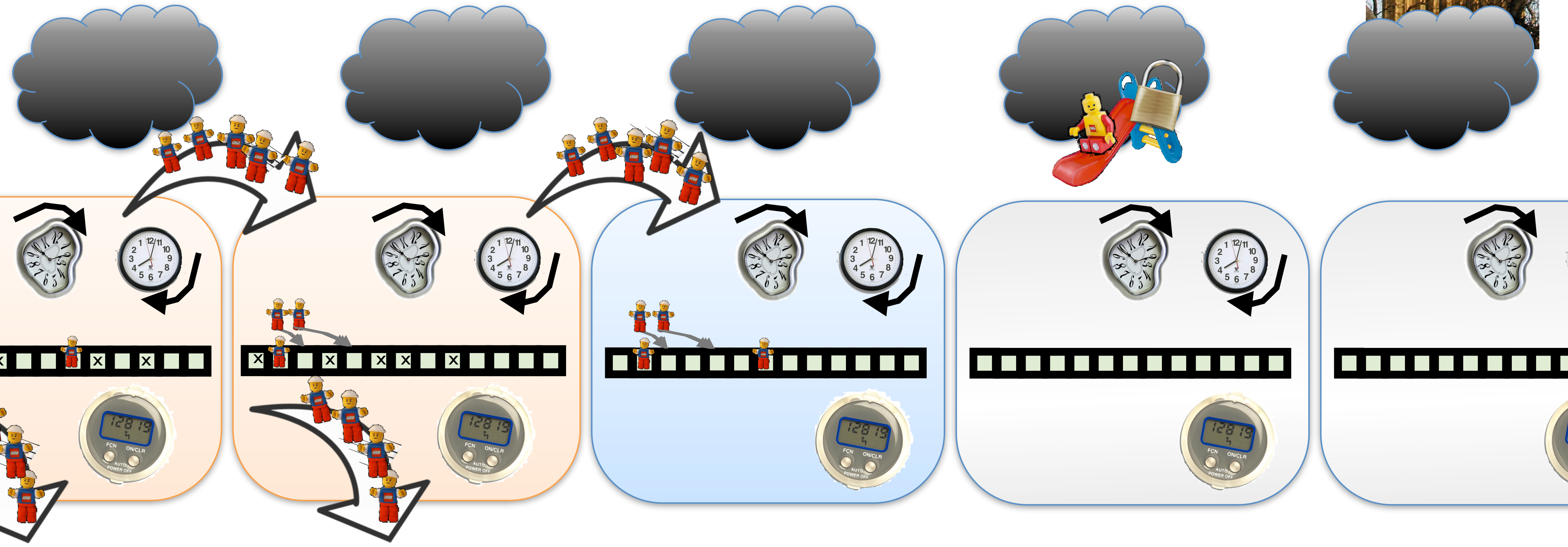
- Θ(n) procs may read the counter in each epoch.
  - Proc reads epoch counter. Falls asleep.
  - Discovers the epoch has changed.
  - Reads the counter again. Falls asleep again.
  - Etc



How Does a Process Determine Which Epoch to Join?

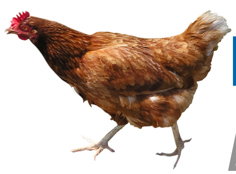
Problem: what amortizes cost to read counter?

- Θ(n) procs may read the counter in each epoch.
  - Proc reads epoch counter. Falls asleep.
  - Discovers the epoch has changed.
  - Reads the counter again. Falls asleep again.
  - Etc



Chicken-and-Egg Problem with Epoch Counter

Which comes first?

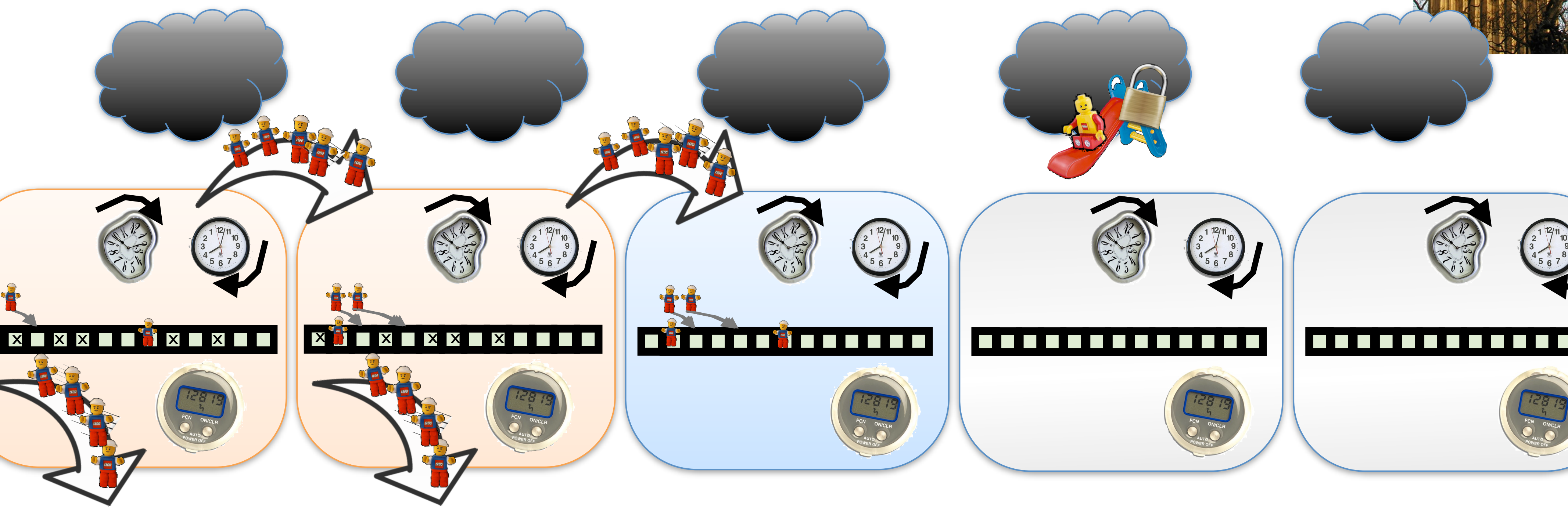


Read the epoch number.

But this read isn't amortized.

Increment some process counter.

But which one? We don't know the epoch.



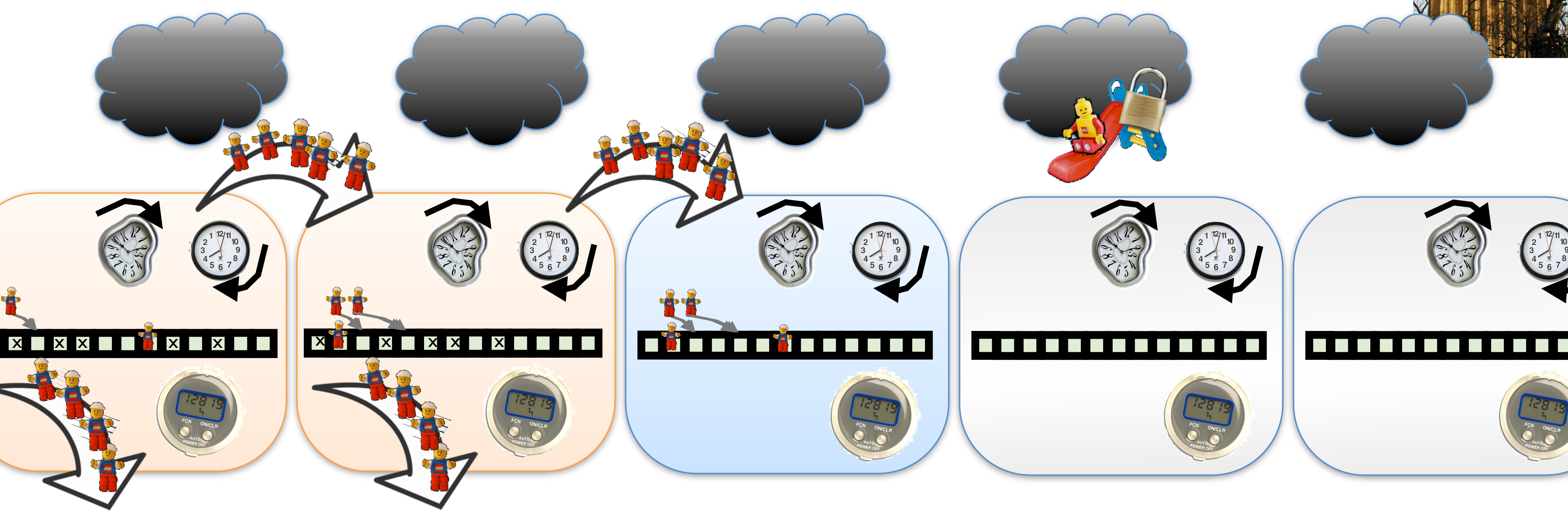
Chicken-and-Egg Problem with Epoch Counter

Chickens are fairly recent. E.g., 10s of millions of years. Eggs have been around for >400 Million years.

Chicken-and-Egg Problem with Epoch Counter

When a process arrives, its first operation must be a write.

- Proc must write even without knowing epoch #.
- The write must be visible to other procs in the epoch.



Chicken/Egg Problem Resolved

A process's first operation must be a write.

This write must increment the population count.

Chicken/Egg Problem Resolved

A process's first operation must be a write.
This write must increment the population count.

Idea: there isn't one arrival counter per phase.
There are only 3. These are reused.

For epochs
1,4,7,10,....

For epochs
2,5,8,11,....

For epochs
3,6,9,12,....

Chicken/Egg Problem Resolved

Which counter should the proc increment?

Answer: choose one randomly.

Recall:

- writing one bit in a random (not uniform) location is enough to record one's presence.
- Once this bit is written, the procs can read the epoch counter and proceed as before.....

For epochs
1,4,7,10,....

For epochs
2,5,8,11,....

For epochs
3,6,9,12,....

Chicken/Egg Problem Resolved

Resetting the counter.

- When the phase ends, reset the counter.
- This reset is not atomic (cannot use pointer swings).

For epochs
1,4,7,10,....

For epochs
2,5,8,11,....

For epochs
3,6,9,12,....

Conclusion

Trend: a distributed realization that many classic problems have sublogarithmic solutions.

- O(log n/loglog n) mutex [Hendler and Woelfel 09]
- O(log* n) test ‘n’ set (George’s talk)
- O(loglog n) consensus (Jim’s talk)
- O(polyloglog) mutual exclusion (this talk)

We are collectively discovering what can/can’t be done in sublogarithmic time.

- Most of these results have oblivious adversaries.
- Yes: approximate counting, leader election.
- No: exact counting, random sampling.

Conclusion

Our algorithm is composed of building blocks.

- Counters, approximate counters, max registers, arrays, CAS, etc.

Alas, most of these aren't strongly linearizable.

- Even when they are (e.g., CAS), it's not not relevant to an oblivious adversaries.

Result:

- Inelegant proofs.
- Lisa, Philipp, Wojciech, George, please hurry up :-)

Open Questions

Stronger adversary?

Adaptive to number of participants?

- This paper: running time depends on n.

Monte Carlo vs. Las Vegas?

- This paper: No deadlock with high probability

Lower bounds?

Alternative constructions that are simpler?