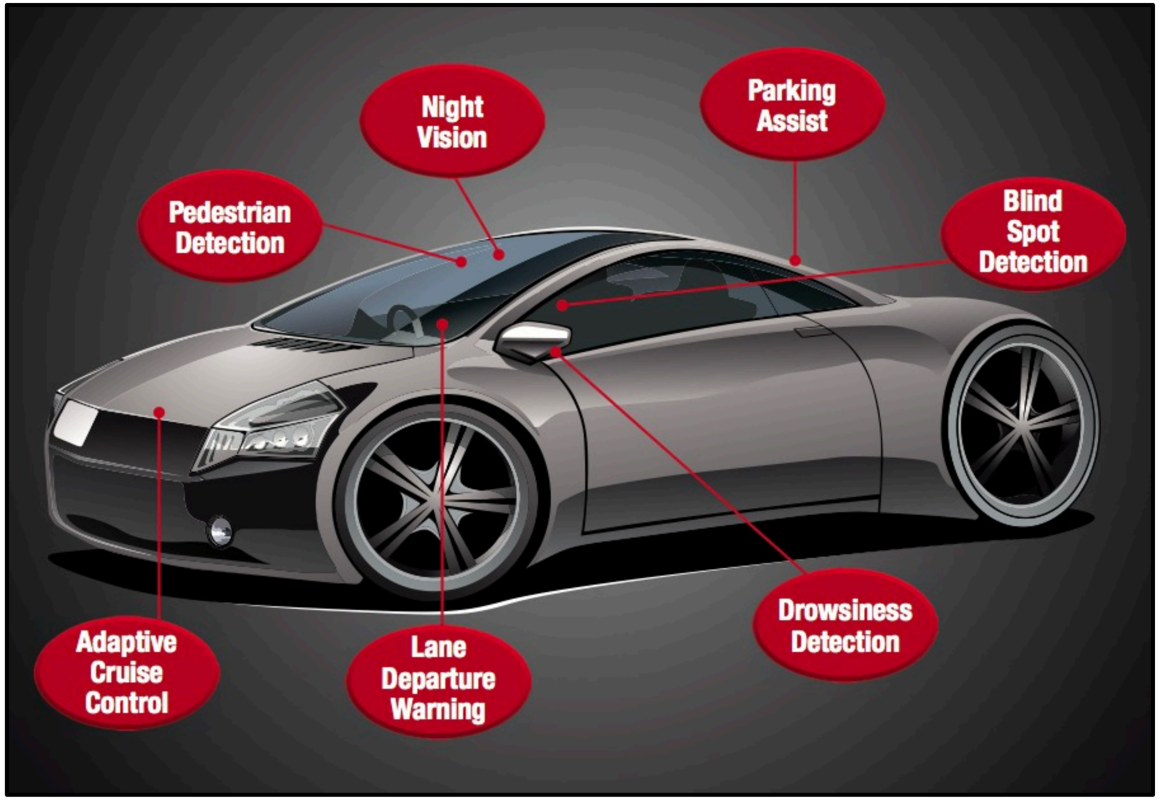
Scalable Memory Reclamation for Multi-Core, Real-Time Systems

Yuxin Ren, Guyue Liu, Gabriel Parmer
The George Washington University

BjÖrn Brandenburg
Max Planck Institute for Software Systems

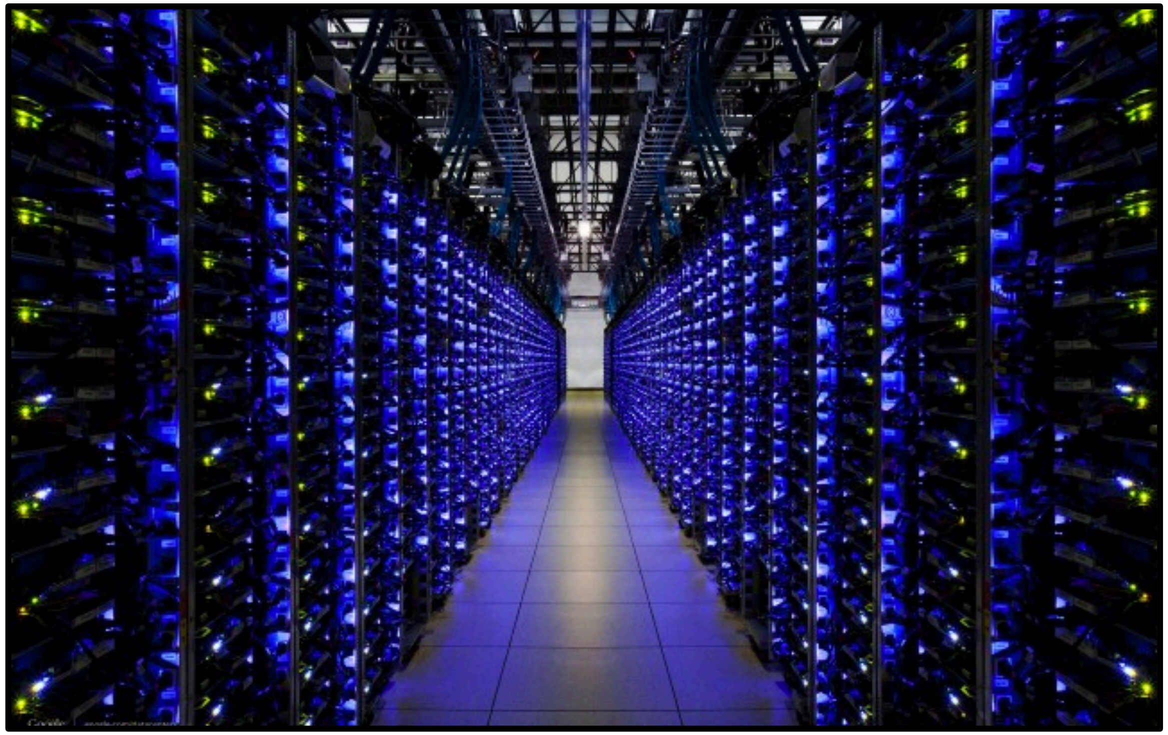
Background: Multicore, Real Time System

How to efficiently share resources?

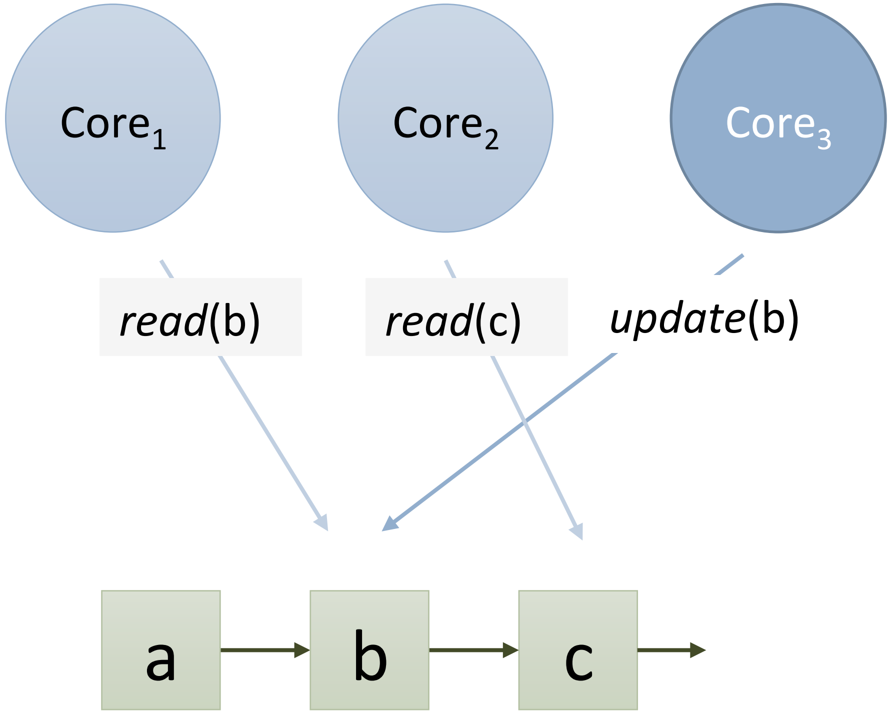


Background: Multicore, Real Time System

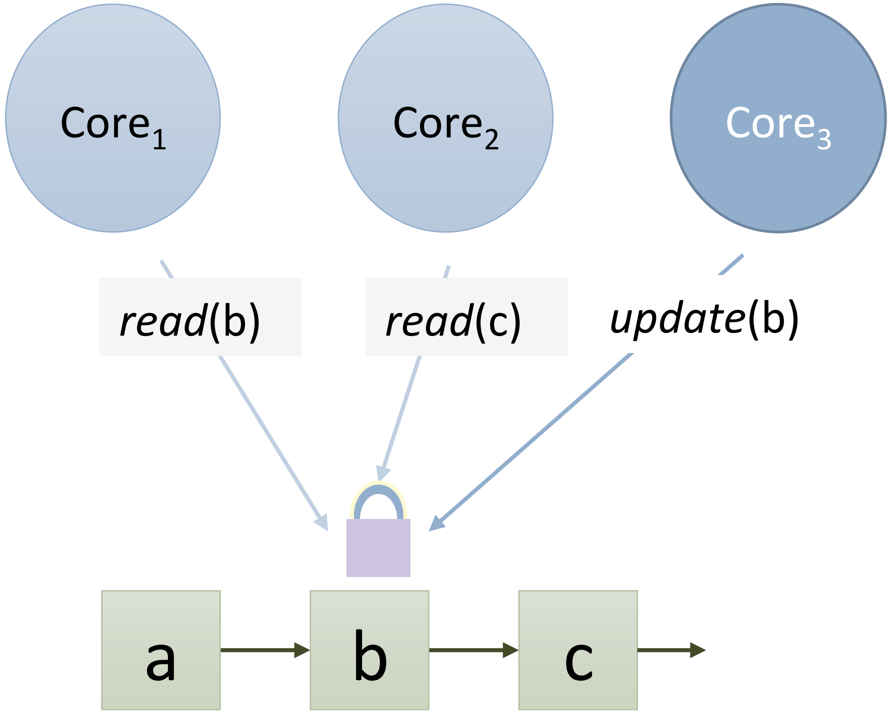
How to provide real-time guarantee?



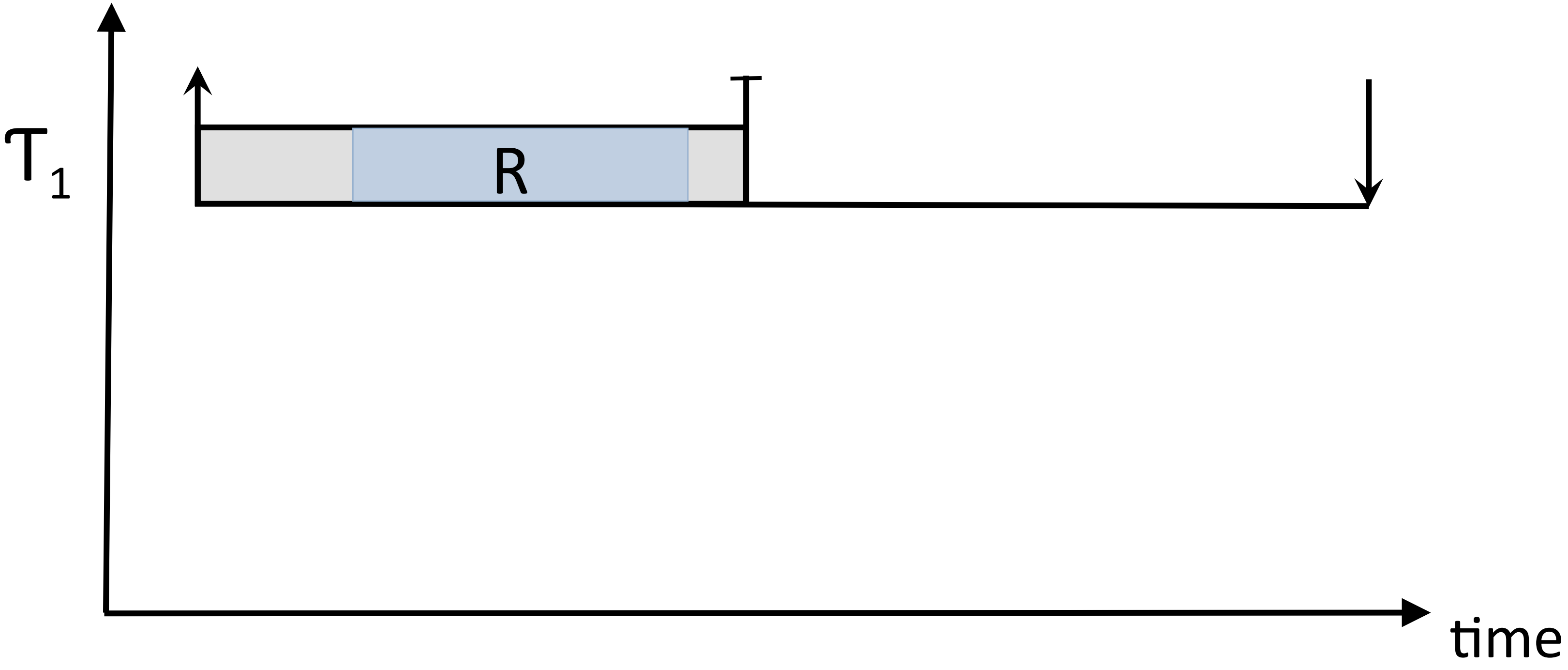
Shared Linked List Example



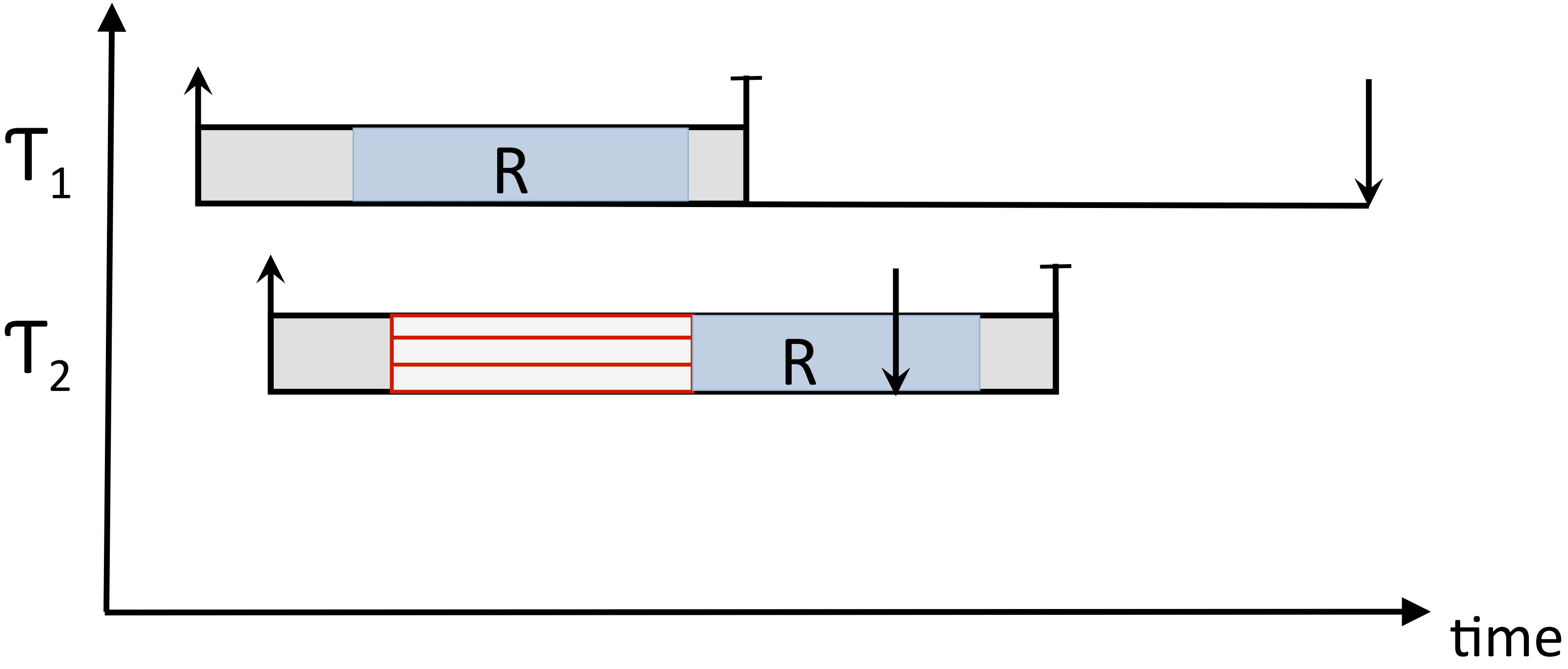
Linked List Example: Global Lock



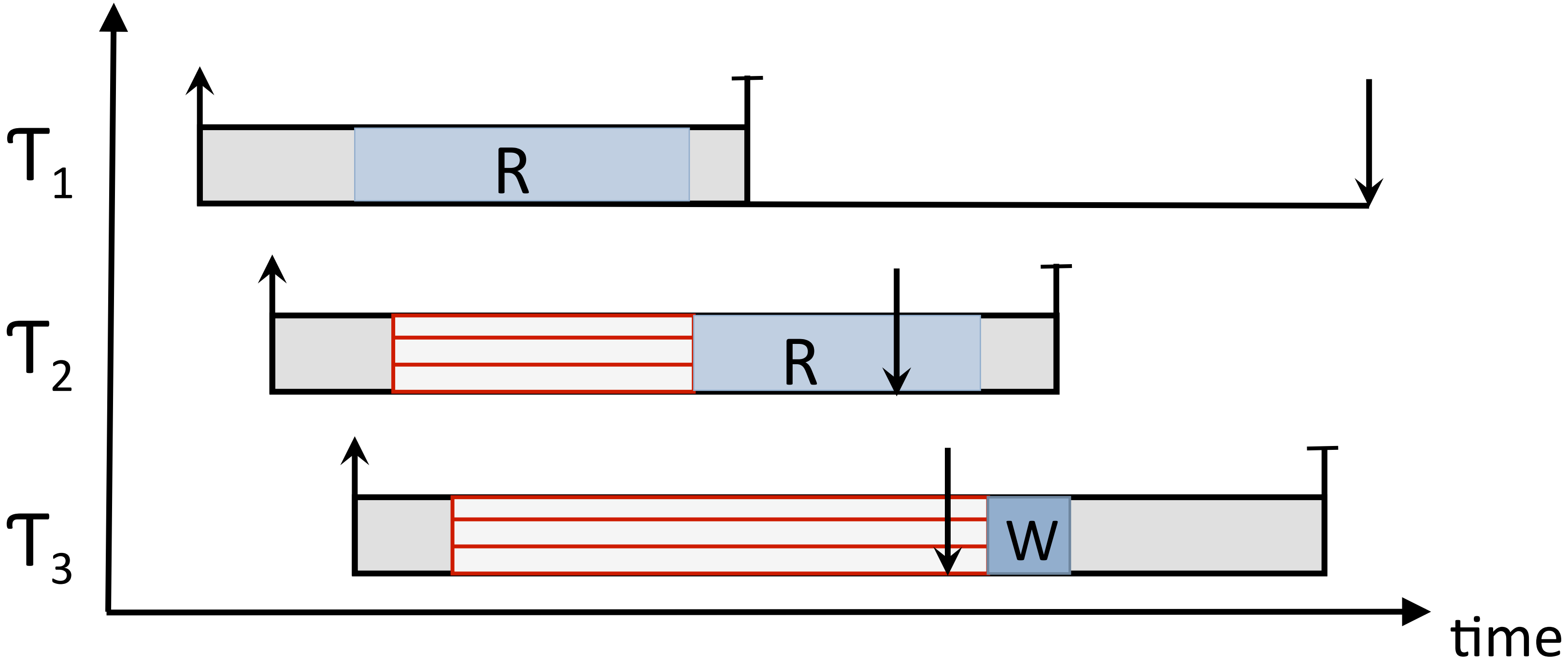
Global Lock: blocking



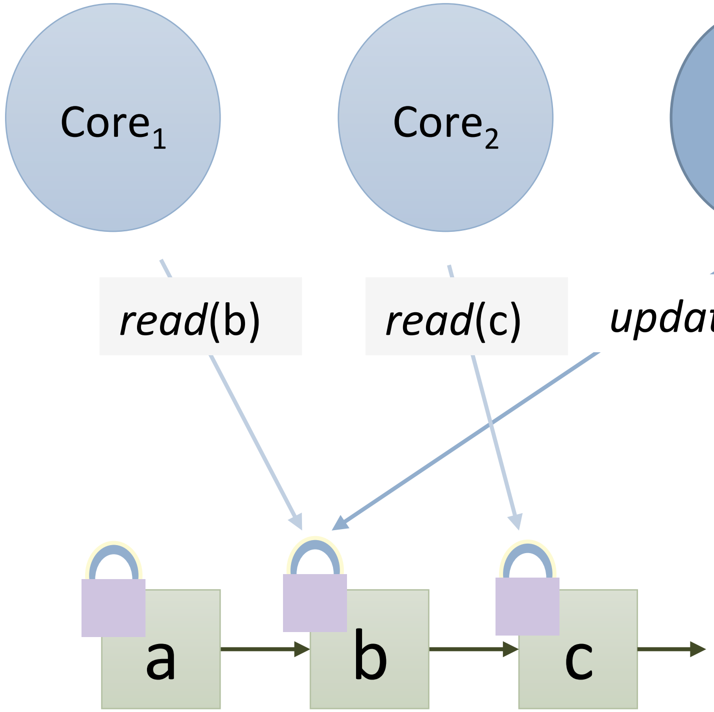
Global Lock: blocking



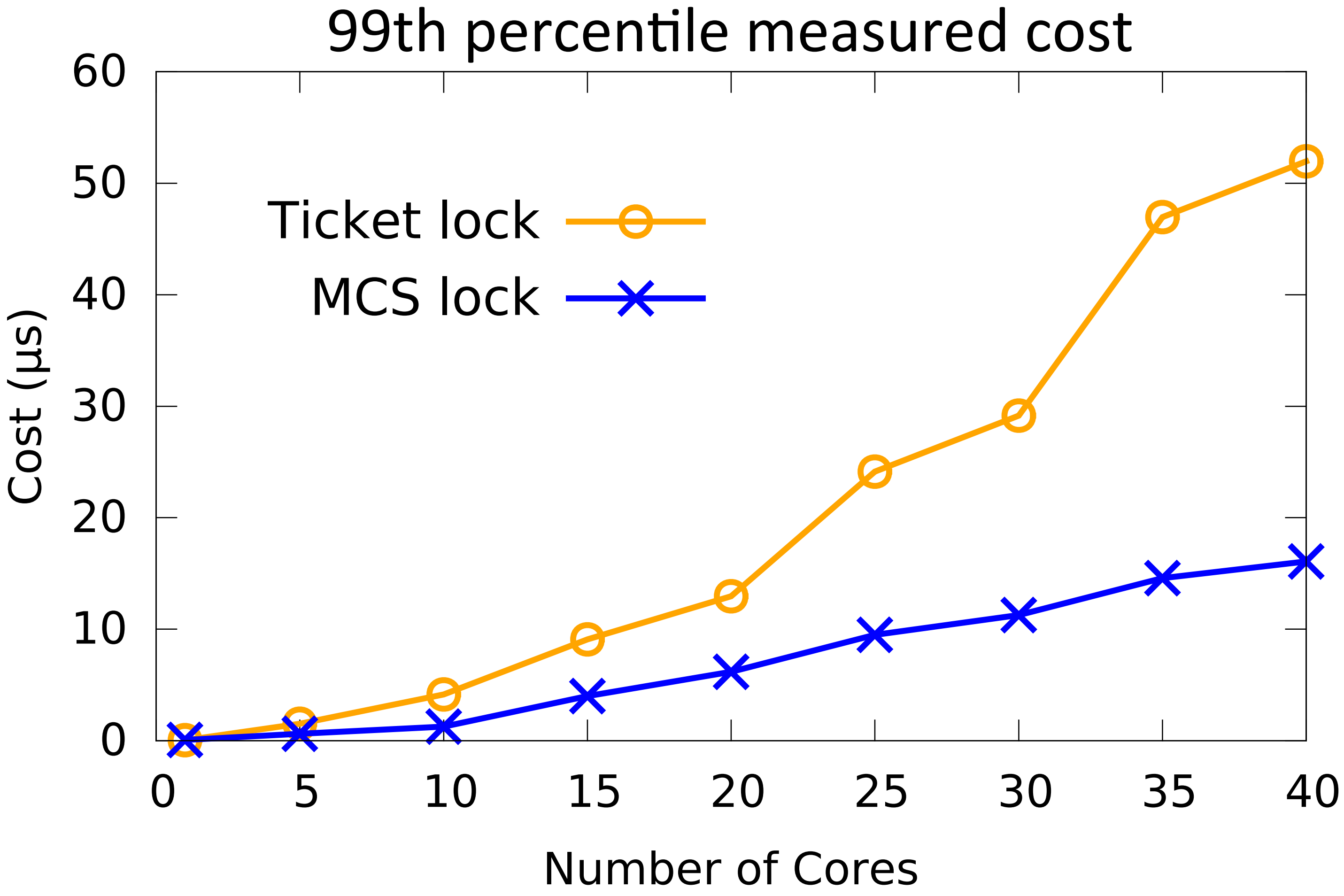
Global Lock: blocking



Linked List Example: Fine-grained Lock



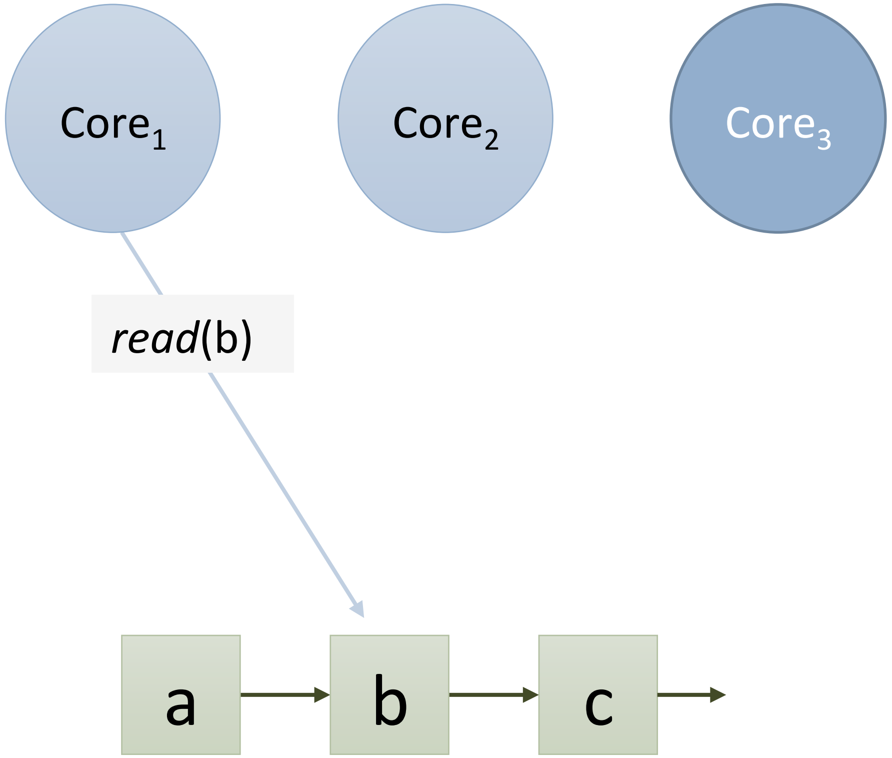
Locking: scalability



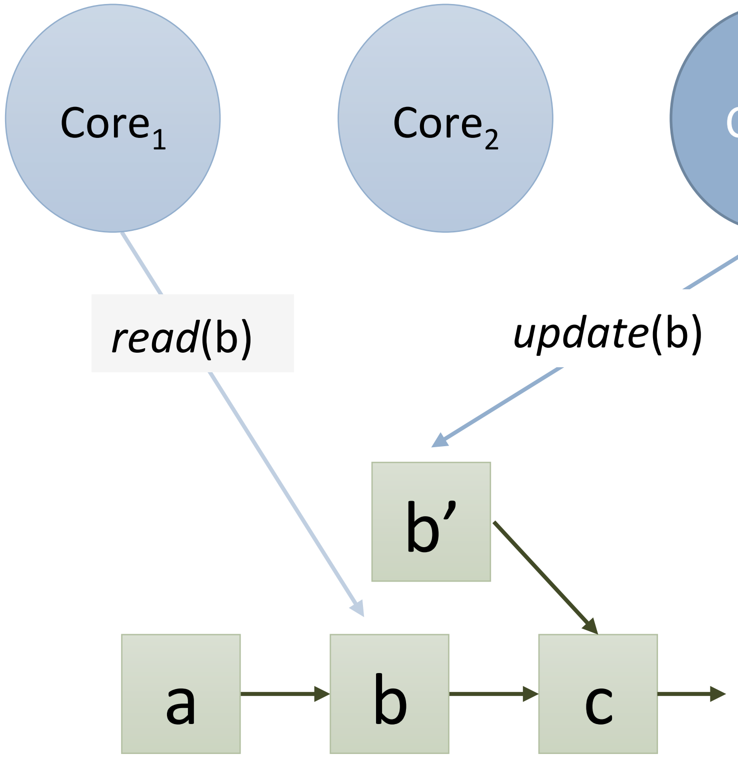
Intel Xeon E7-4850

4 sockets, 40 cores

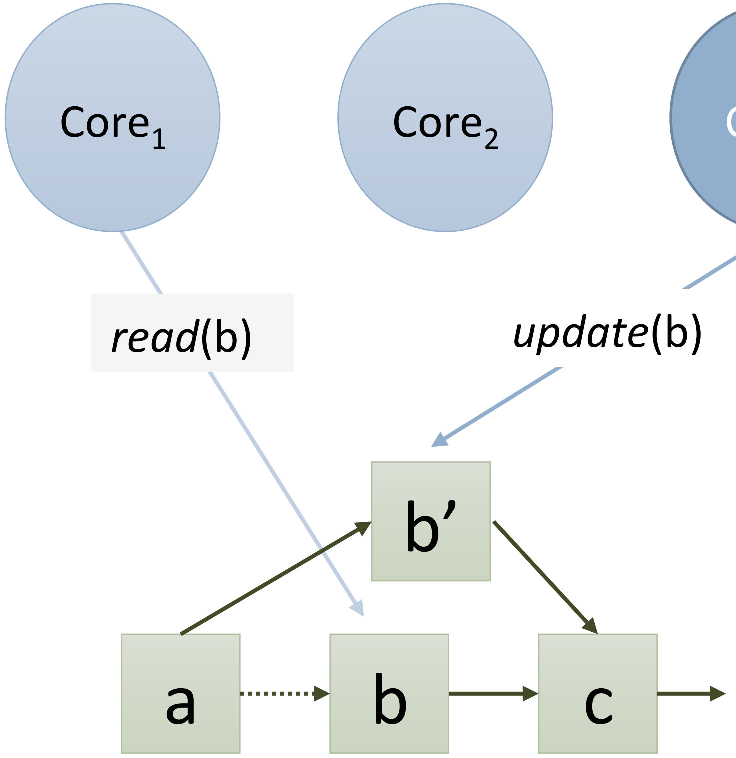
Linked List Example:
Read Copy Update (RCU)
Scalable Memory Reclamation (SMR)



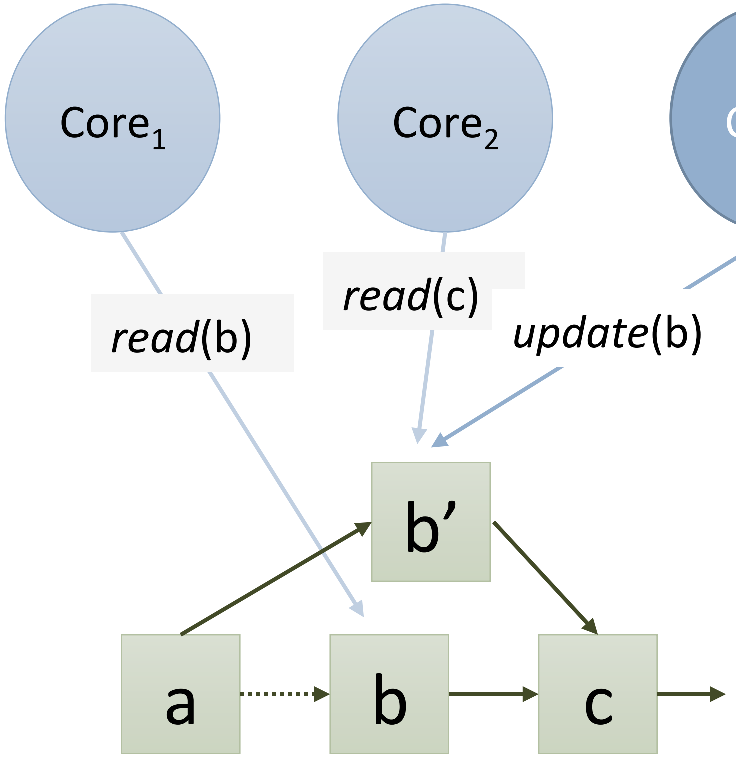
Linked List Example: RCU/SMR



Linked List Example: RCU/SMR



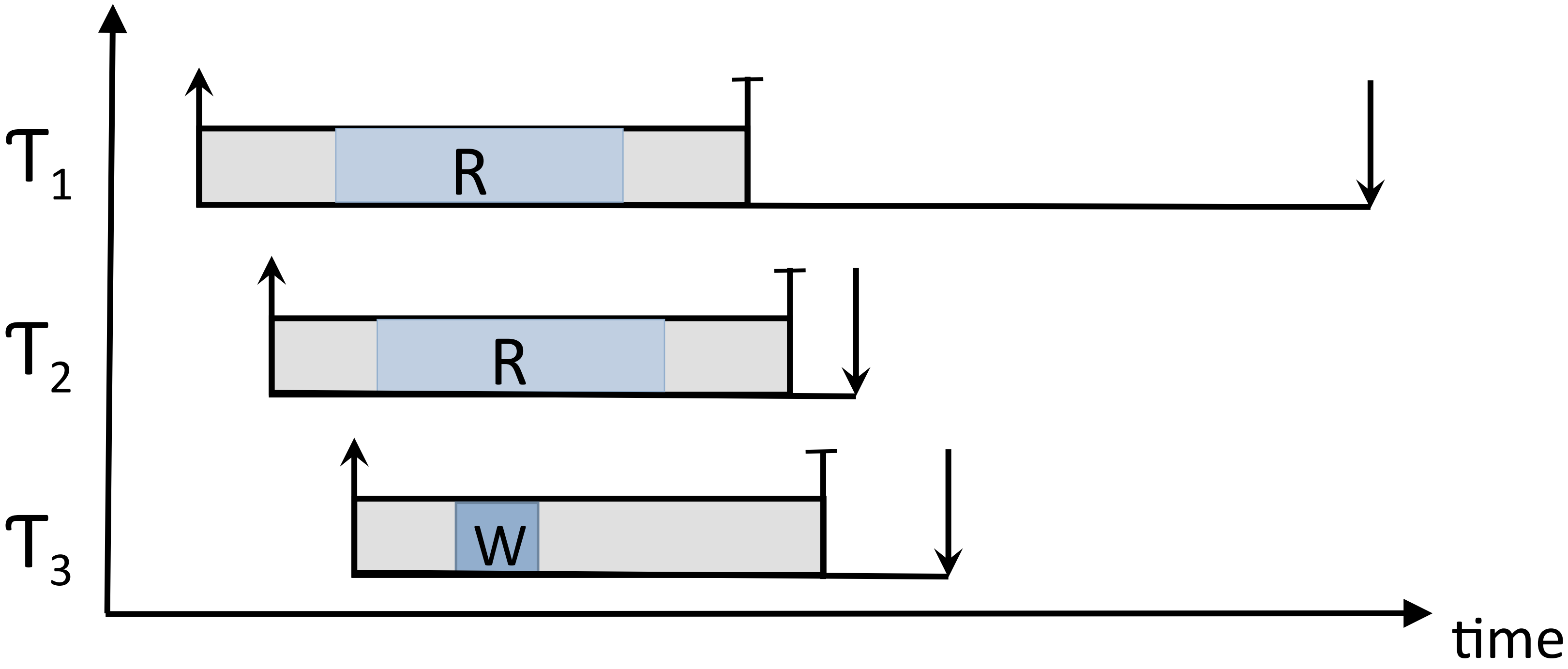
Linked List Example: RCU/SMR



Linked List Example: RCU/SMR

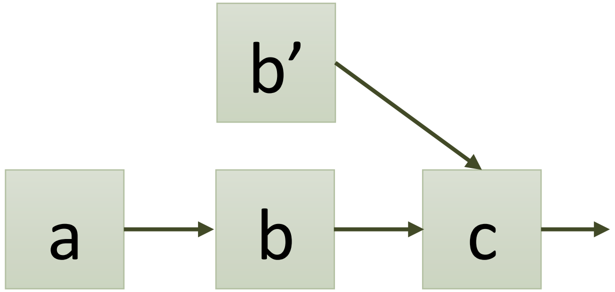
Parallel Readers

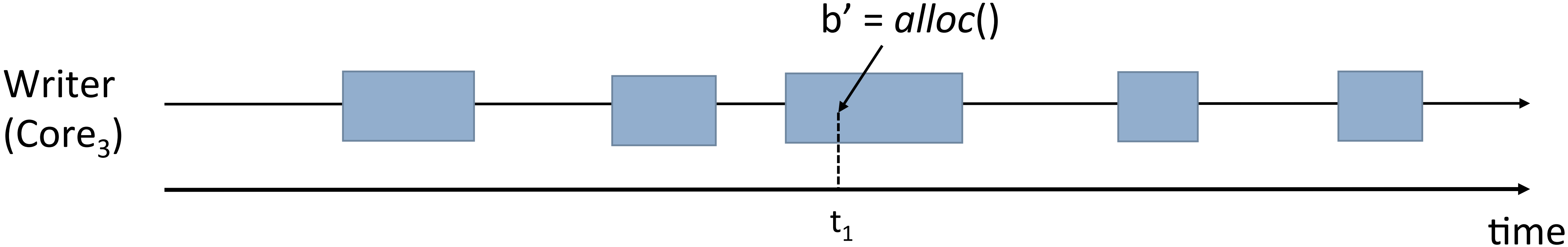
Reduced response time



Scalable Memory Reclamation (SMR)

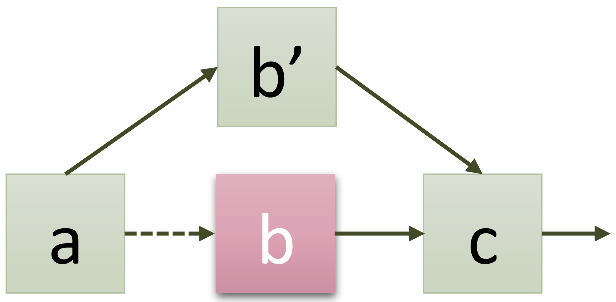
update(b)

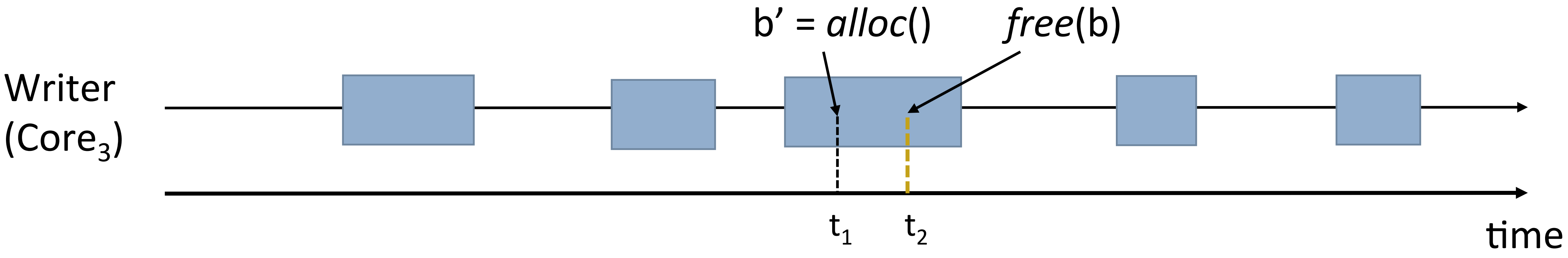




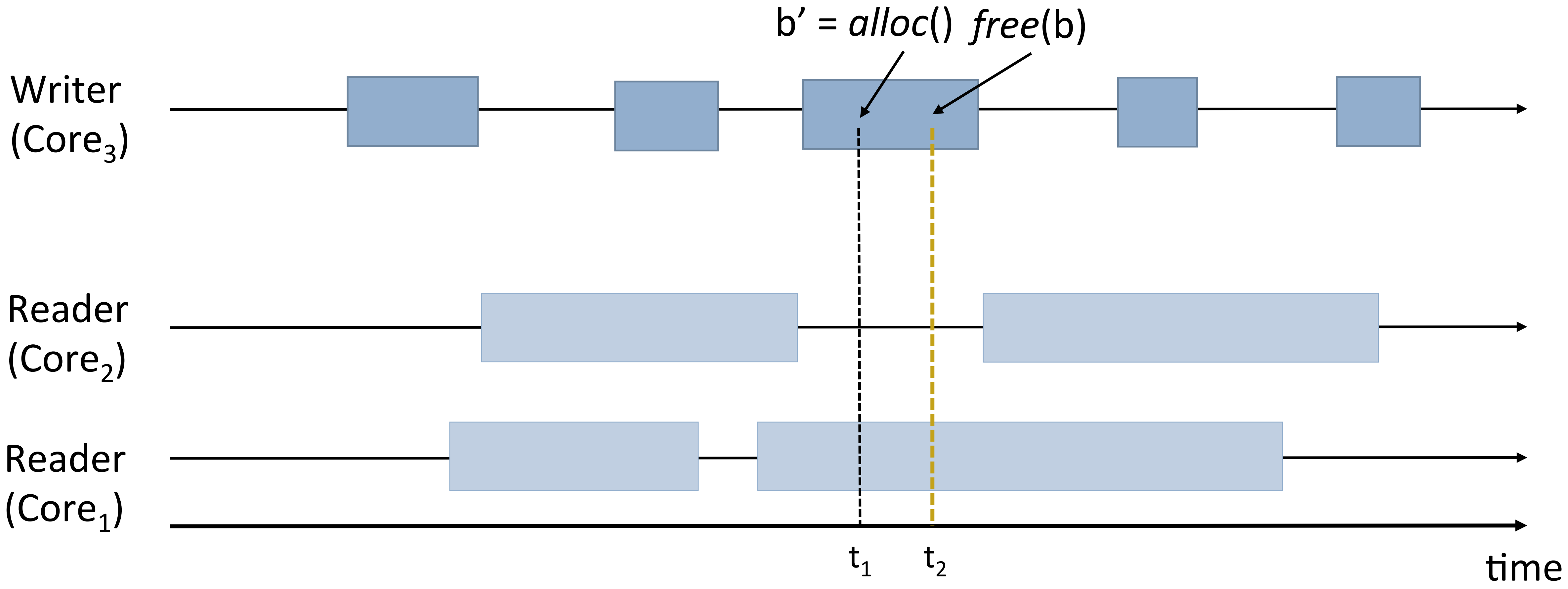
Scalable Memory Reclamation (SMR)

update(b)

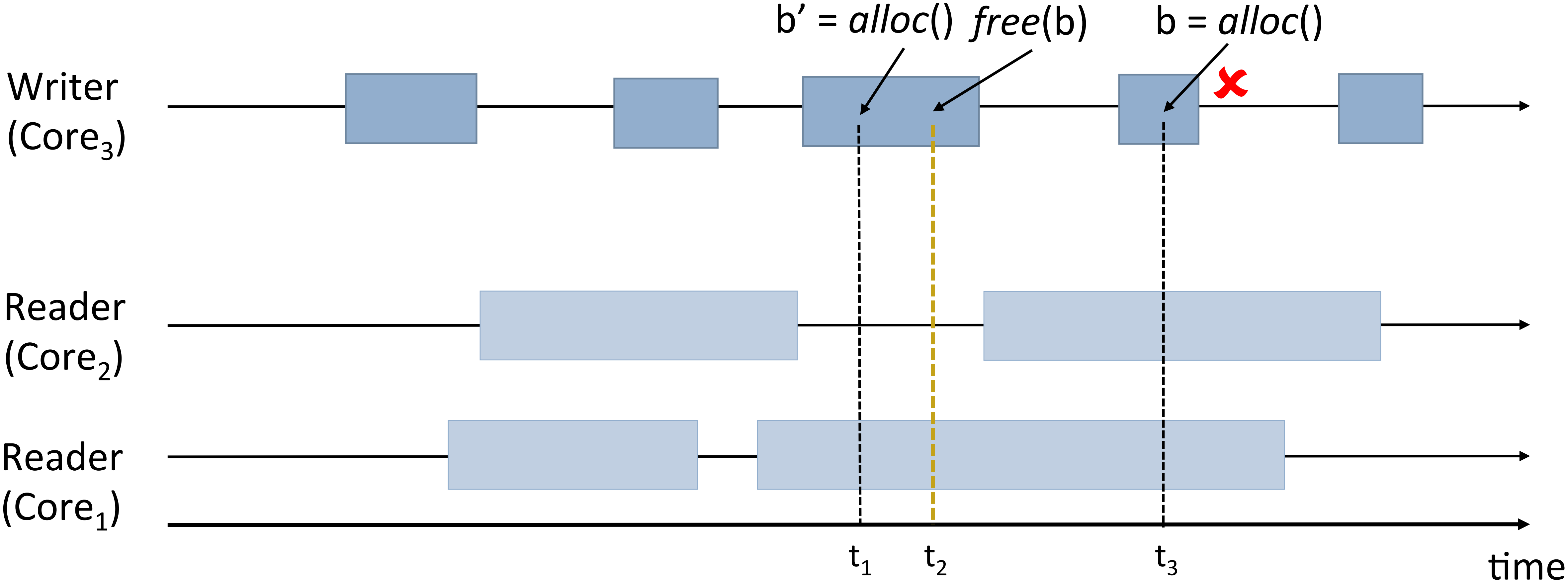




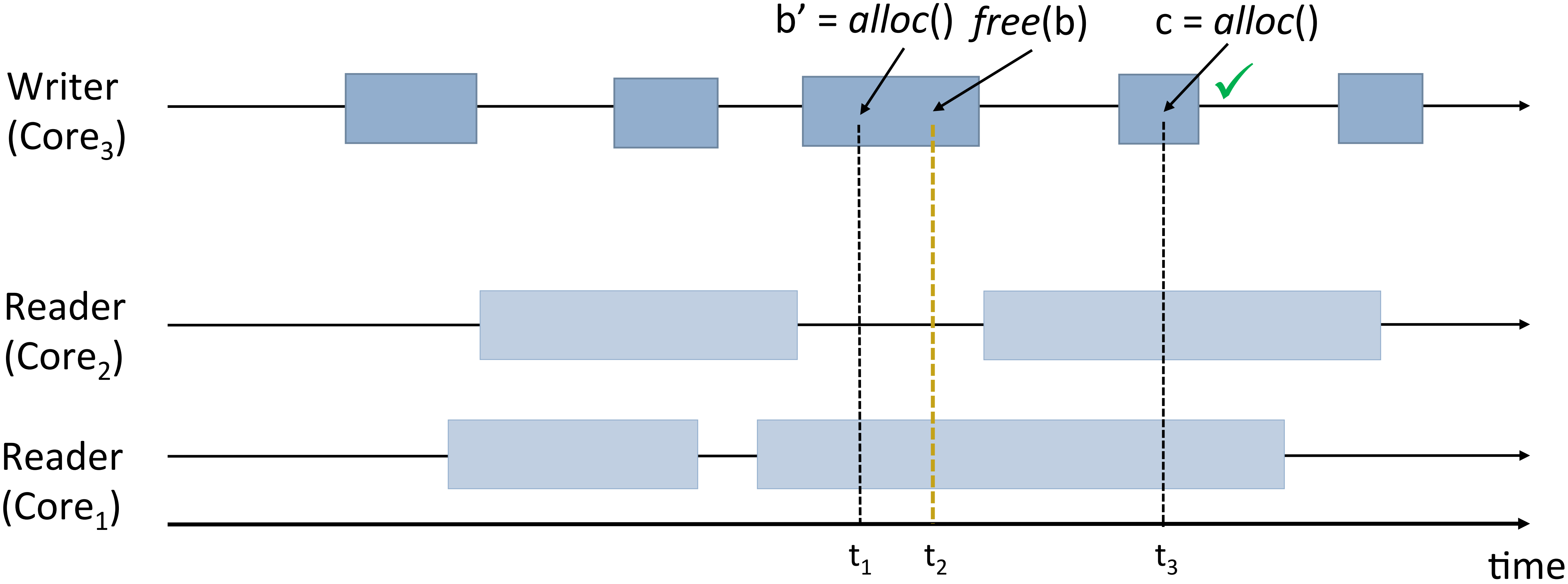
Scalable Memory Reclamation (SMR)



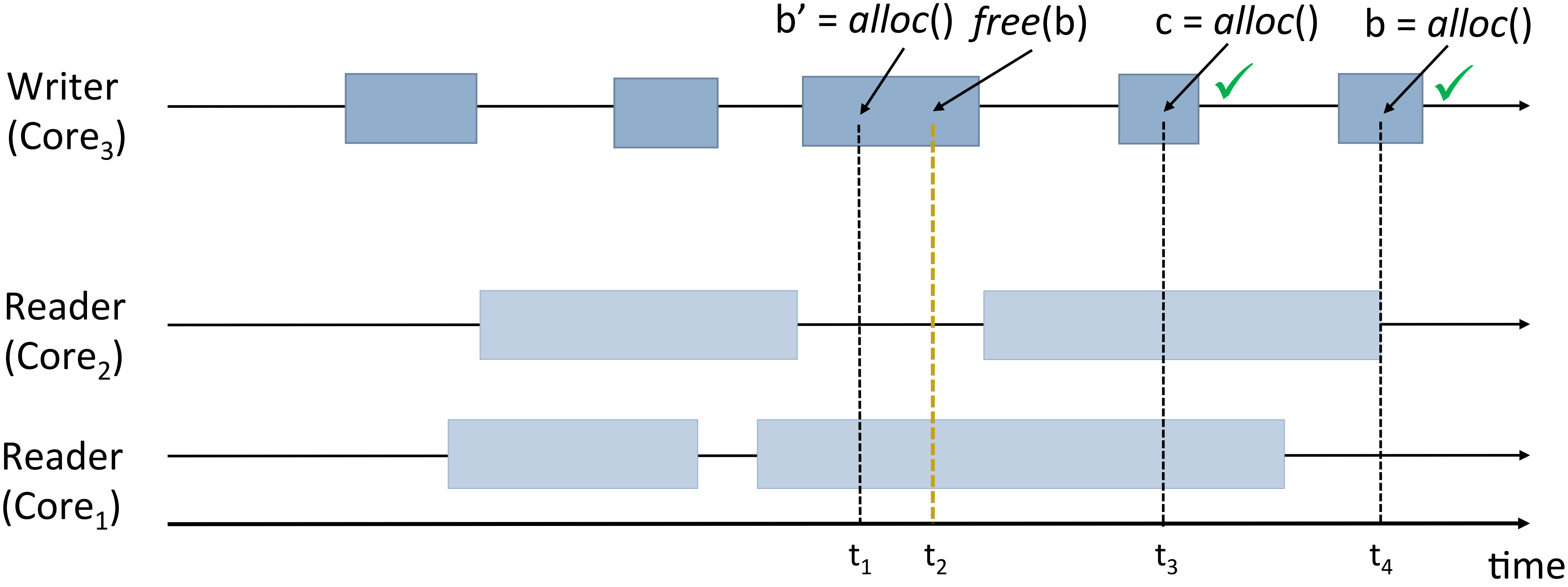
Scalable Memory Reclamation (SMR)



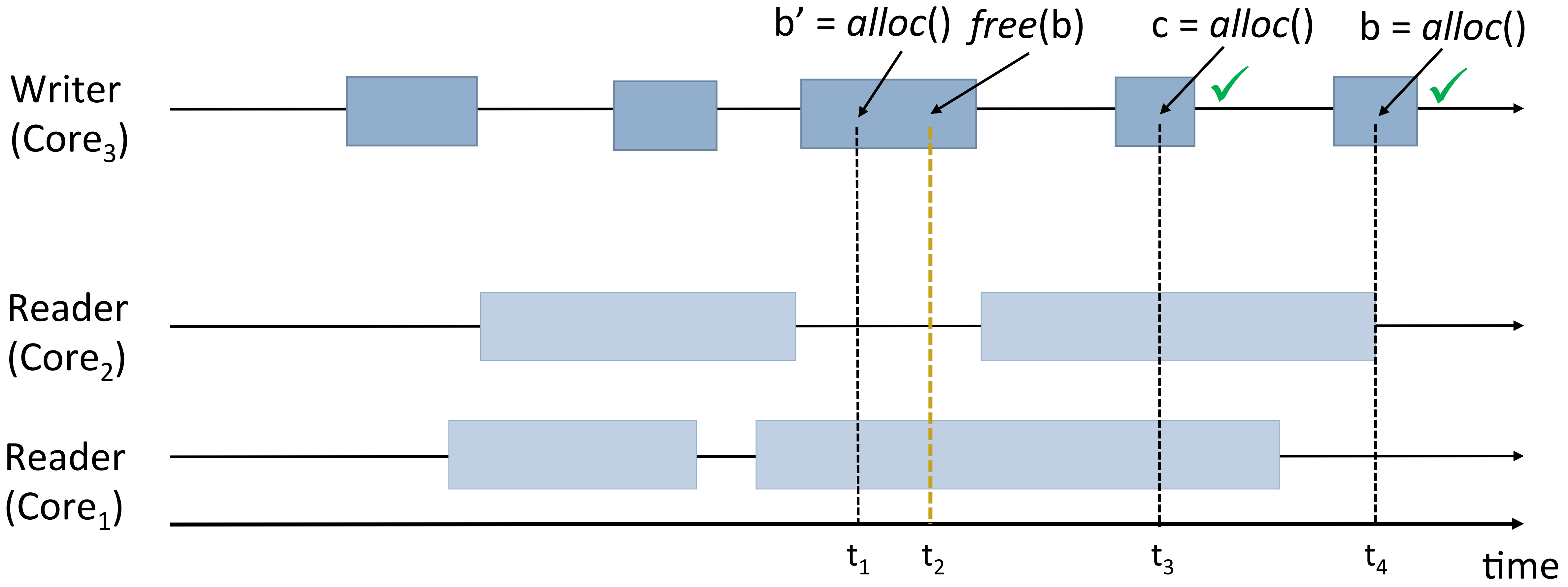
Scalable Memory Reclamation (SMR)



Scalable Memory Reclamation (SMR)

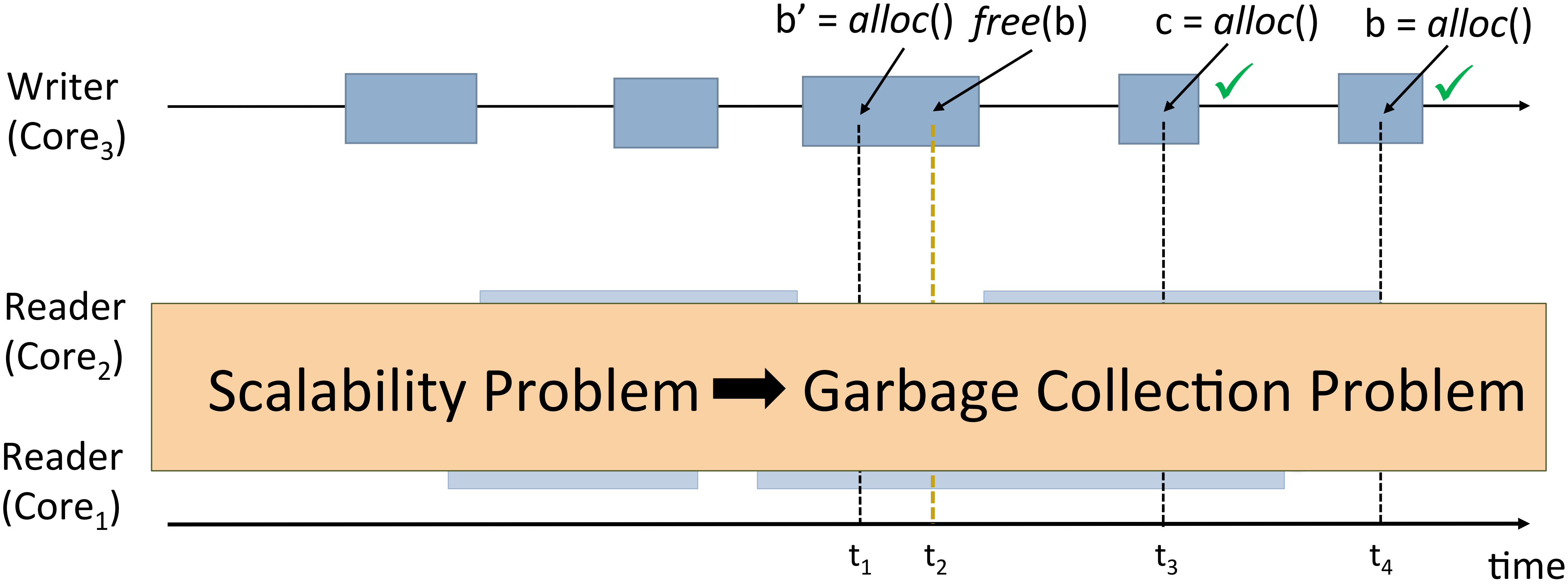


Scalable Memory Reclamation (SMR)



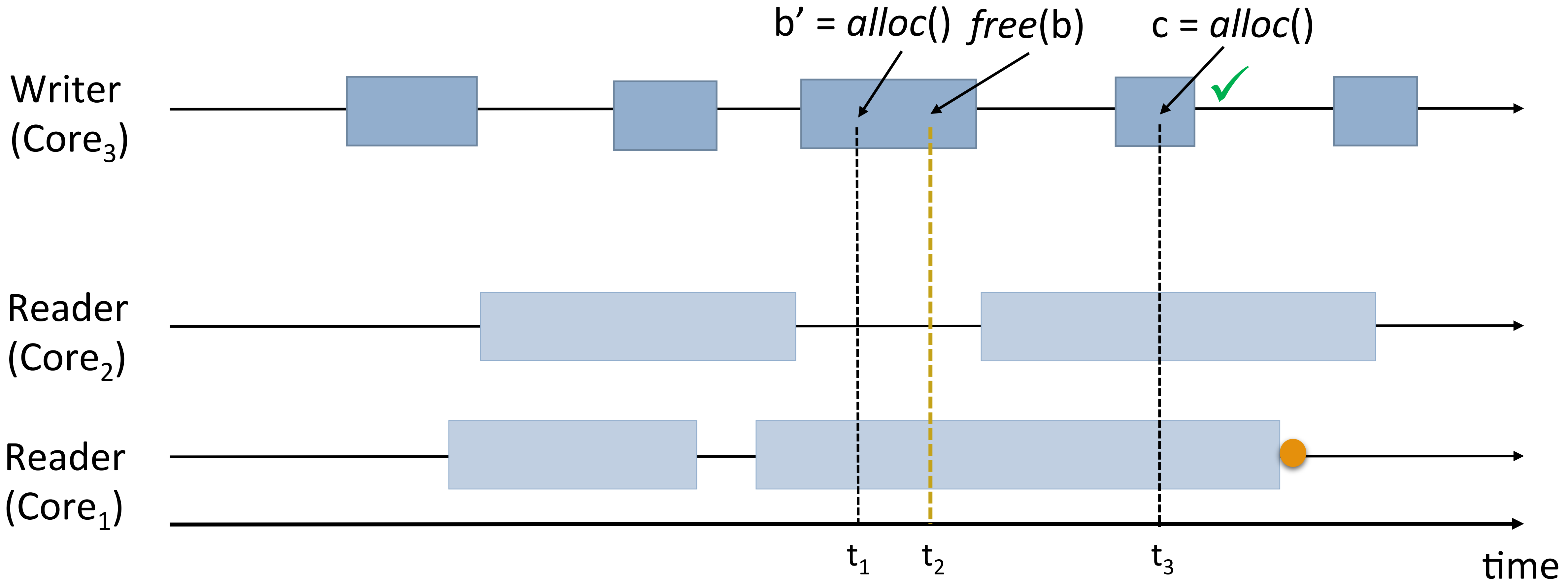
When to reuse freed memory?

Scalable Memory Reclamation (SMR)



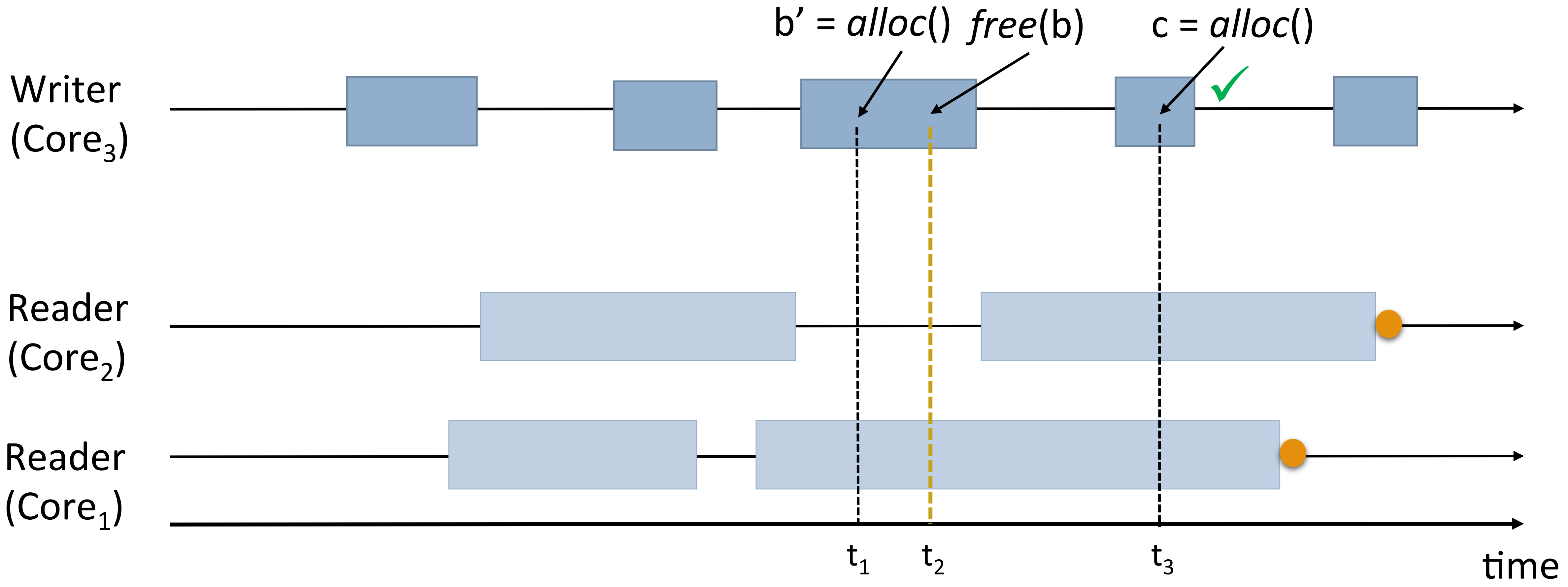
Scalability Problem ➡ Garbage Collection Problem

Scalable Memory Reclamation (SMR)



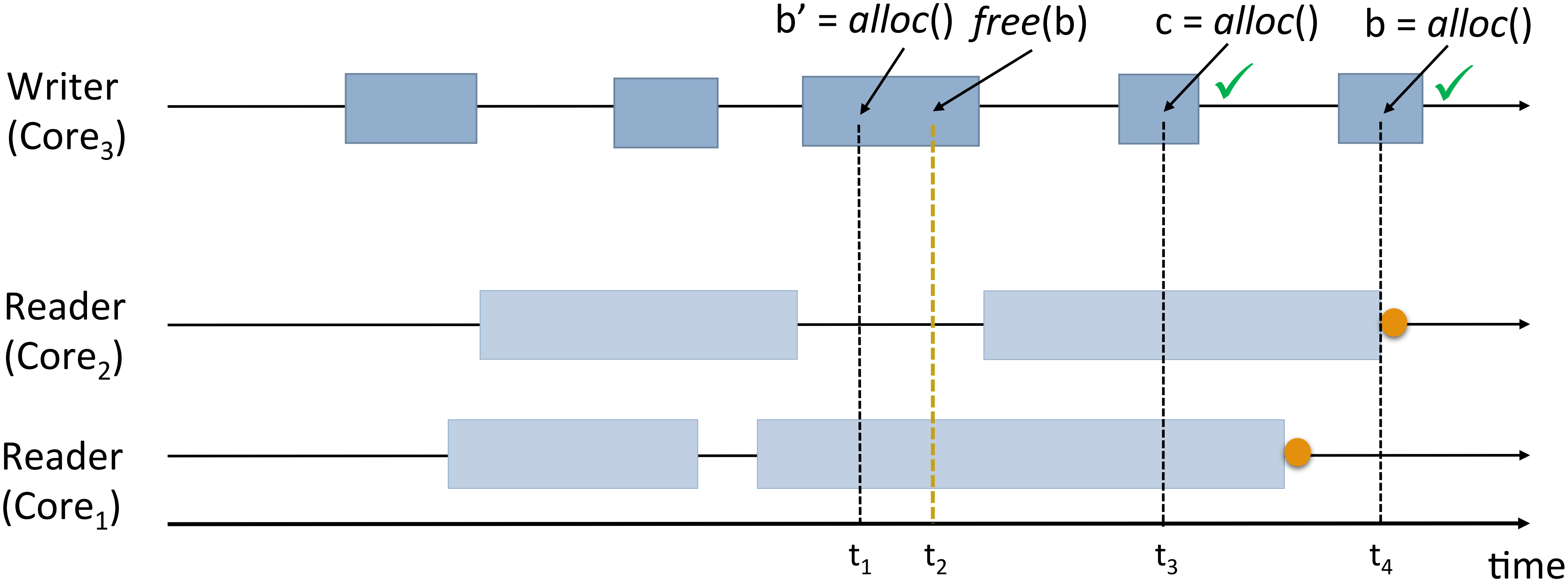
When to reuse freed memory? Quiescence state

Scalable Memory Reclamation (SMR)



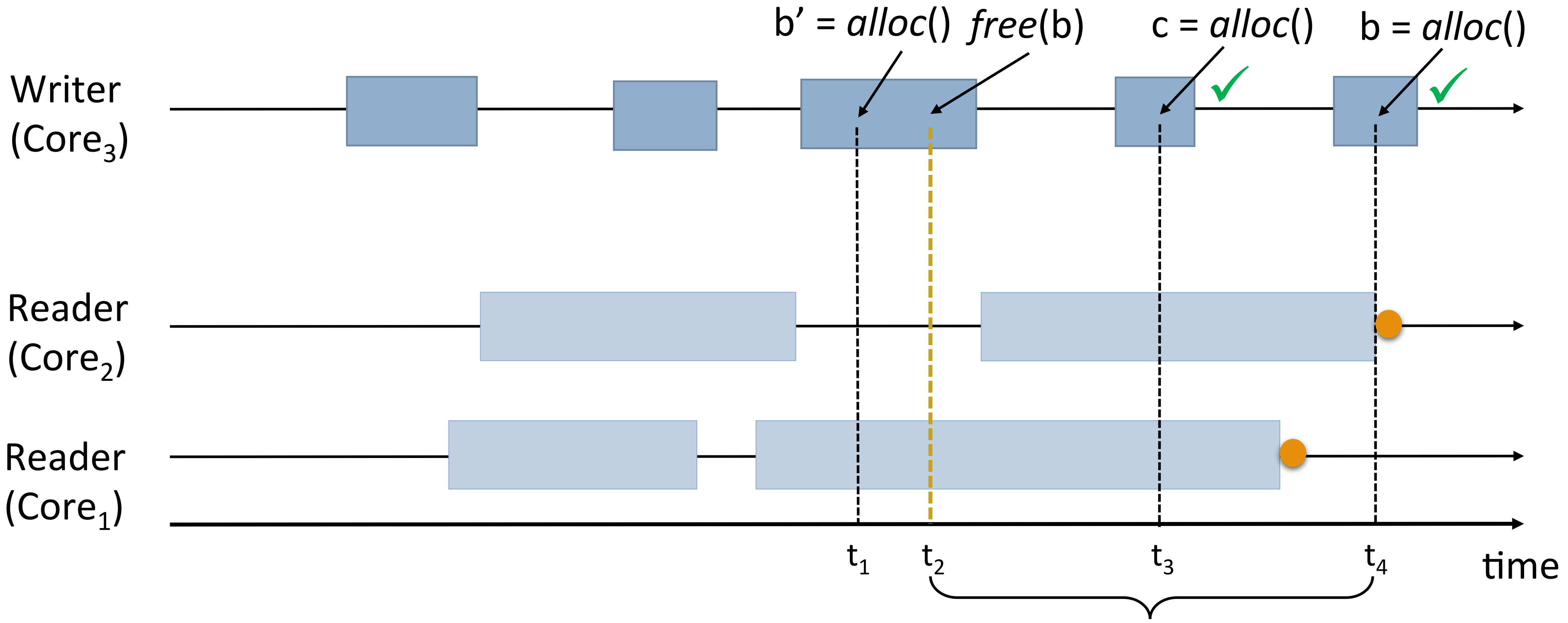
When to reuse freed memory? Quiescence state

Scalable Memory Reclamation (SMR)



When to reuse freed memory? Quiescence state

Scalable Memory Reclamation (SMR)



When to reuse freed memory? Quiescence state → Grace period

SMR: API

- *enter*: declare the start of code referencing a shared data structure
- *exit*: declare the end of code referencing that data structure

- *sync*: calculate grace period
  - wait till memory is reusable
  - delay the reuse of memory

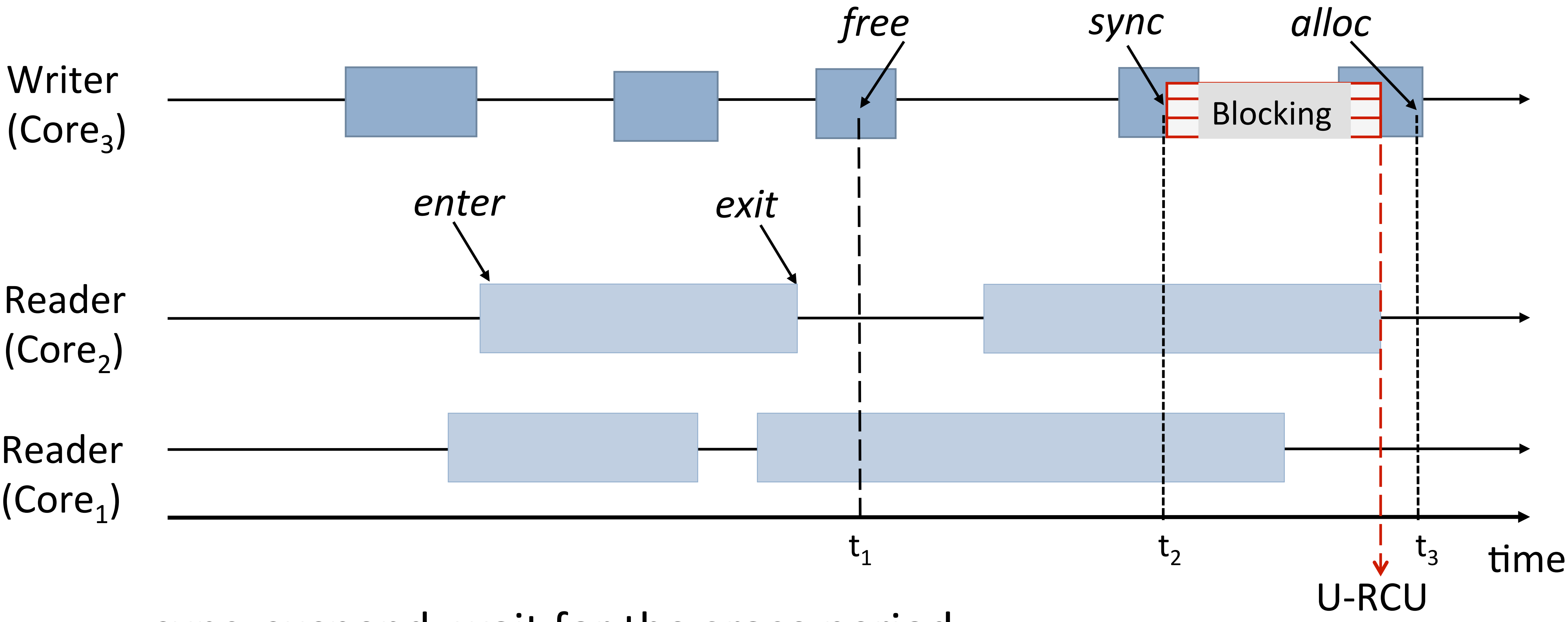
```
enter();
n = walk(path);
process(n);
exit();
// no references into DS remain
```

Read-path

```
enter();
n = unlink(path);
exit();
sync();
// reuse prior memory
```

Update-path

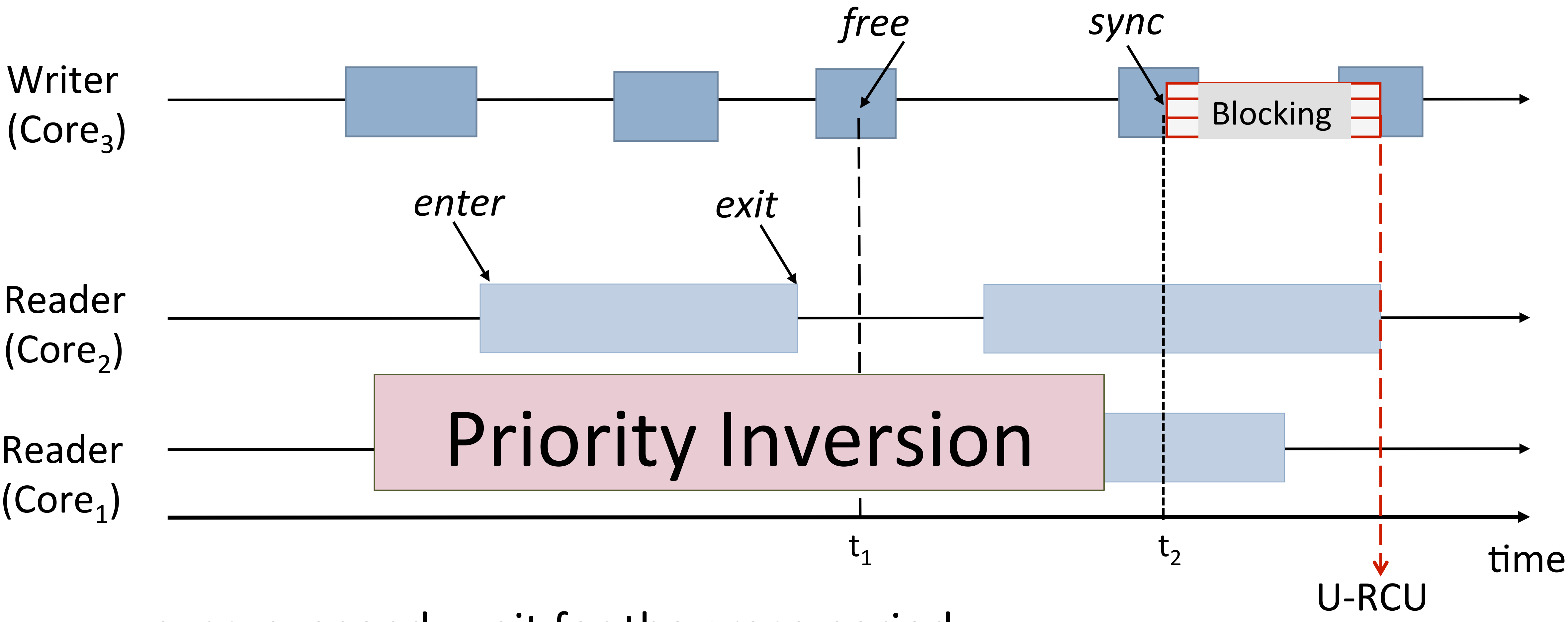
SMR Implementation: U-RCU example



sync: suspend, wait for the grace period

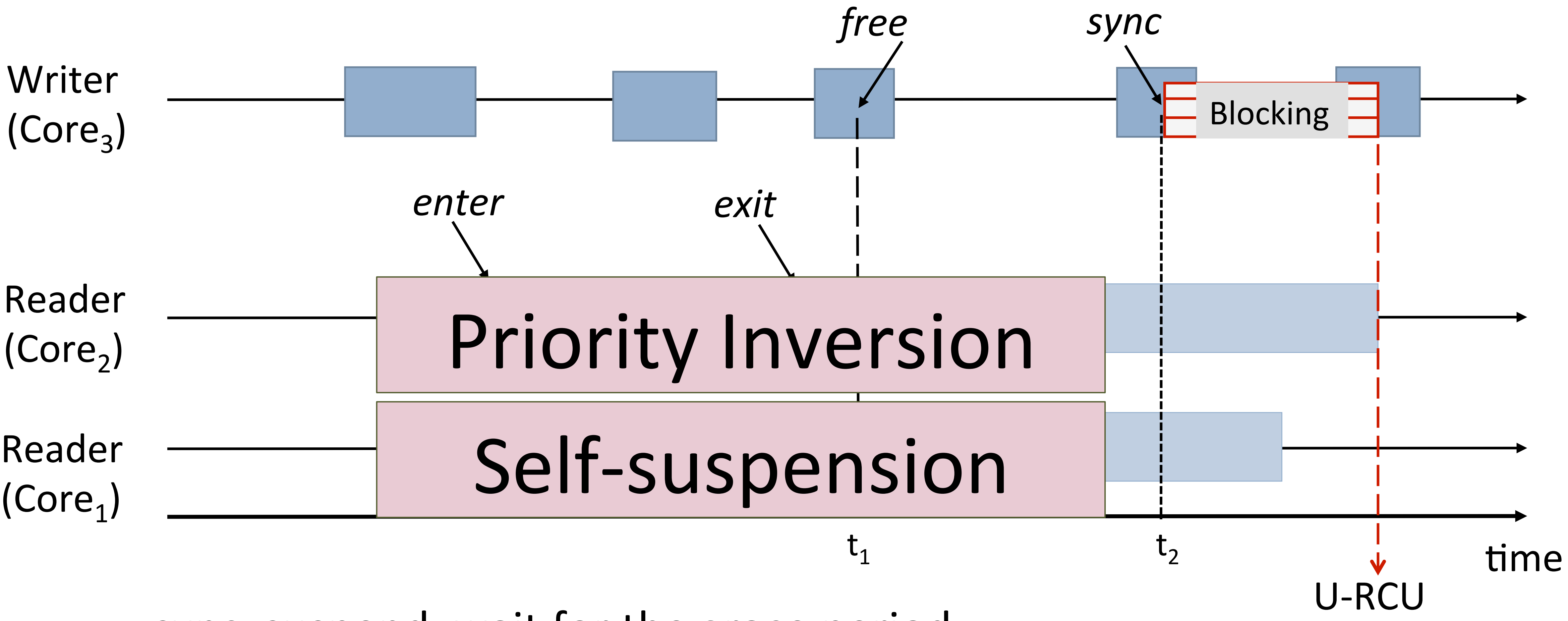
Blocking increases the worst case response time.

SMR Implementation: U-RCU example



sync: suspend, wait for the grace period

SMR Implementation: U-RCU example



sync: suspend, wait for the grace period

Summary of Existing Solutions

	Predictability	Scalability	Efficiency
Lock	✓	✗	✗
U-RCU			
Real-time SMR			

Summary of Existing Solutions

	Predictability	Scalability	Efficiency
Lock	✓	✗	✗
U-RCU	✗	✓	○
Real-time SMR			

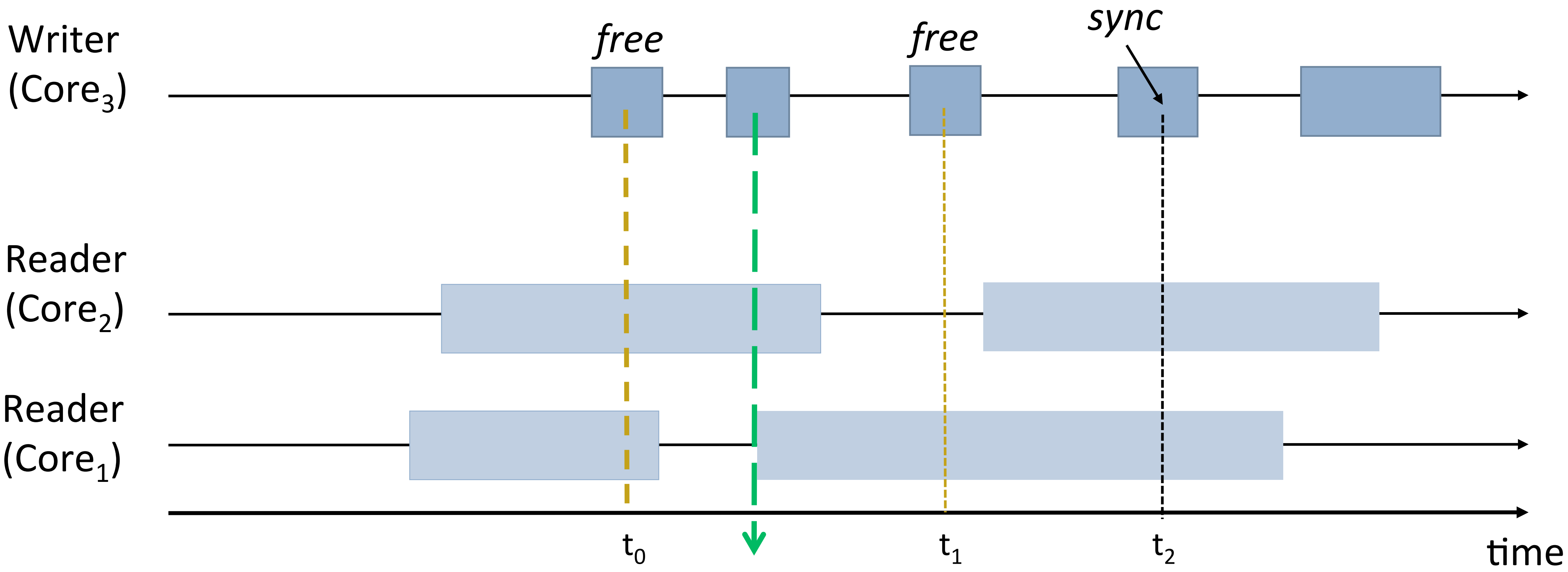
Summary of Existing Solutions

	Predictability	Scalability	Efficiency
Lock	✓	✗	✗
U-RCU	✗	✓	○
Real-time SMR	✓	✓	✓

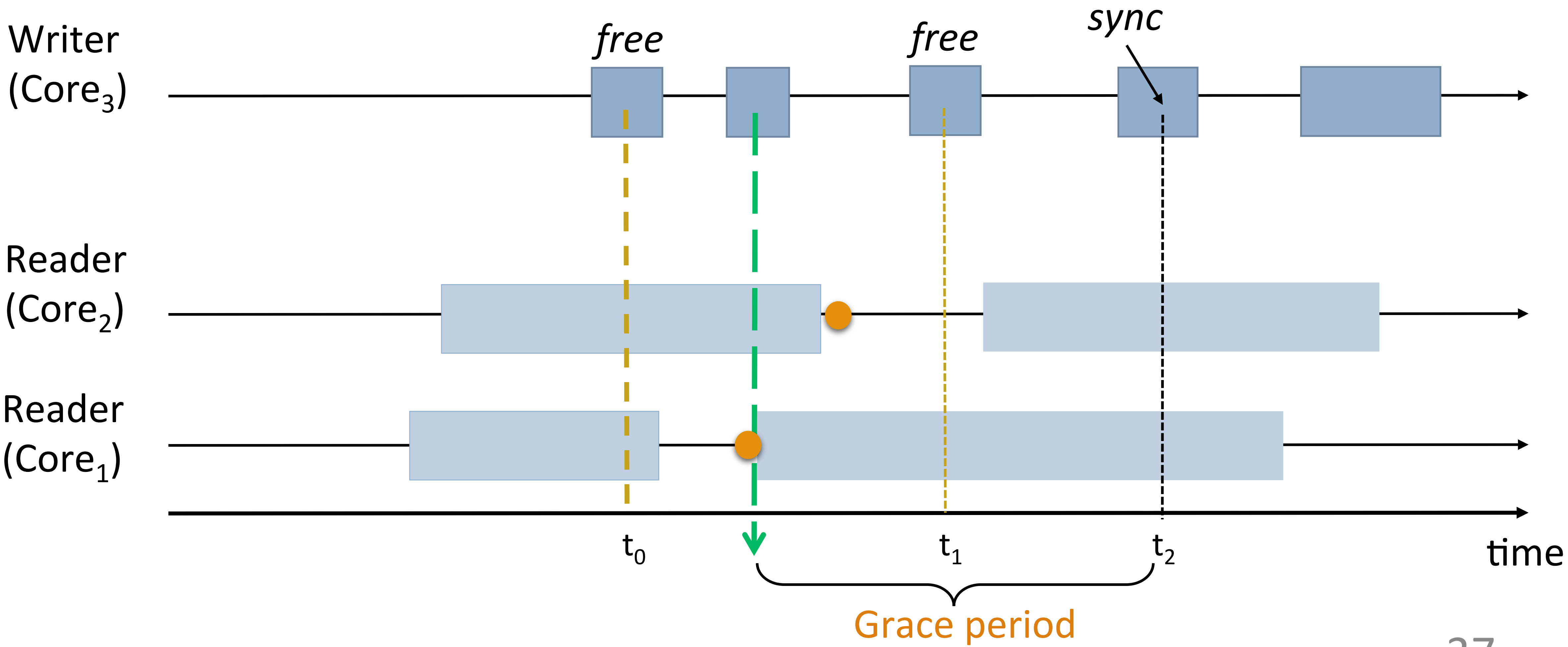
Real-Time SMR

- Two Implementations:
  - RT-ParSec: scalable bounded memory consumption
  - Temporal Quiescence: optimized for hard real time system
- Analytical Model:
  - Worst case memory consumption.
  - Response Time Analysis of quiescence calculations.

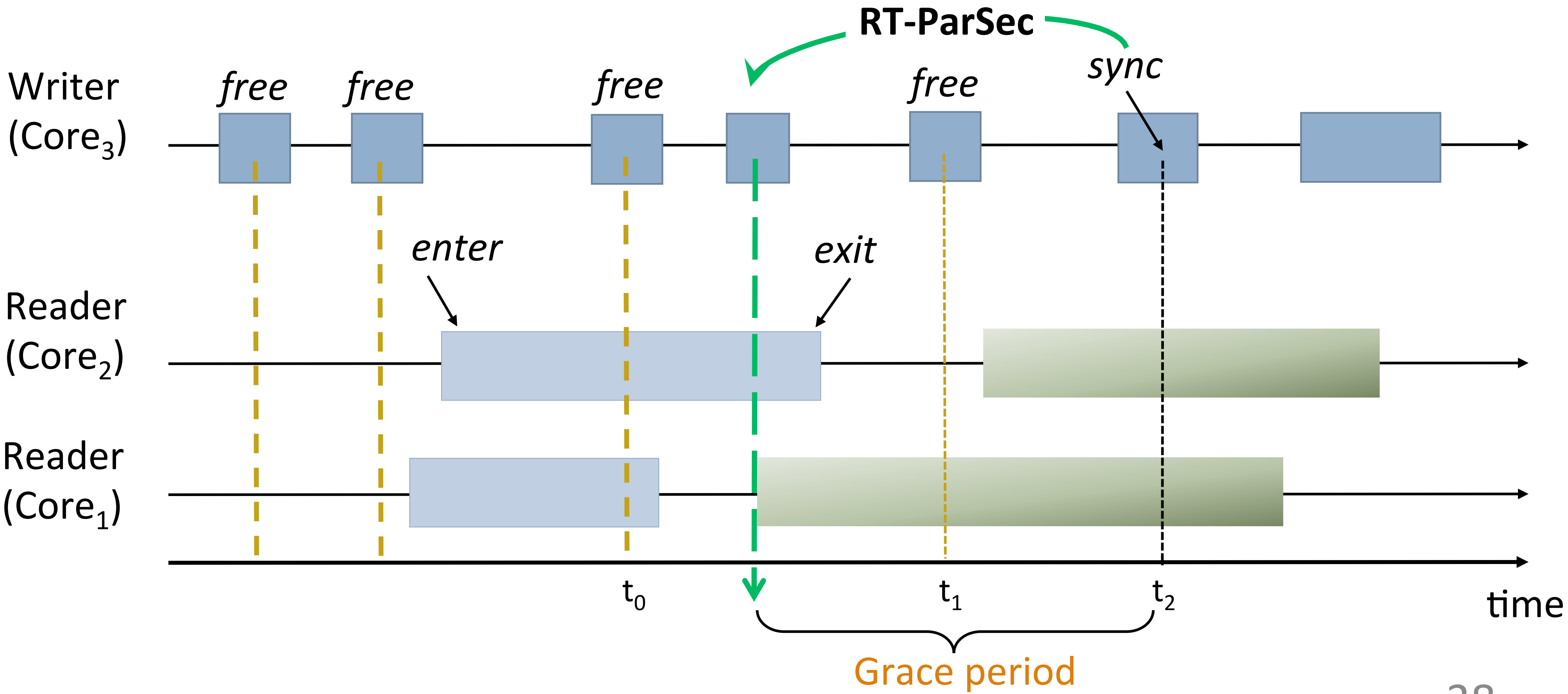
Real-Time SMR: RT-ParSec



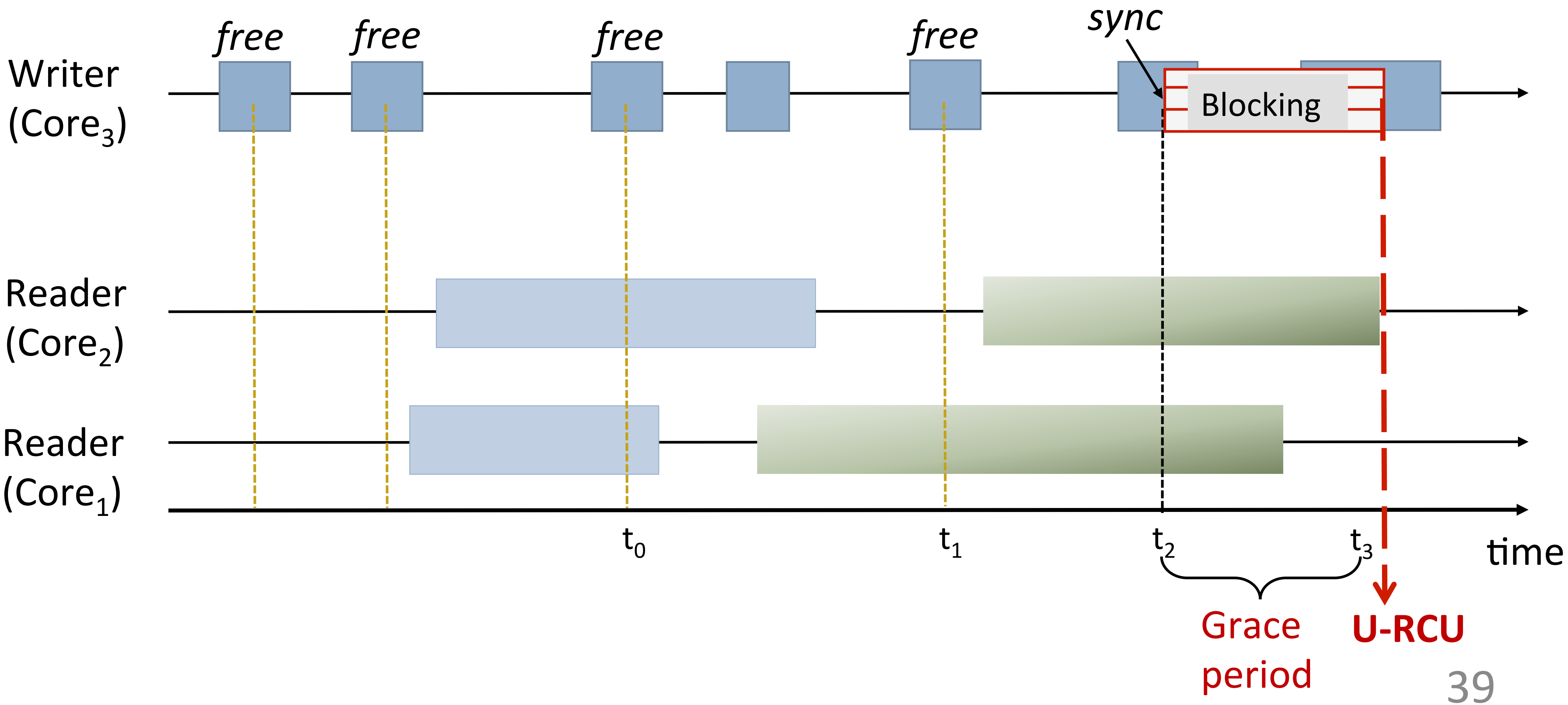
Real-Time SMR: RT-ParSec



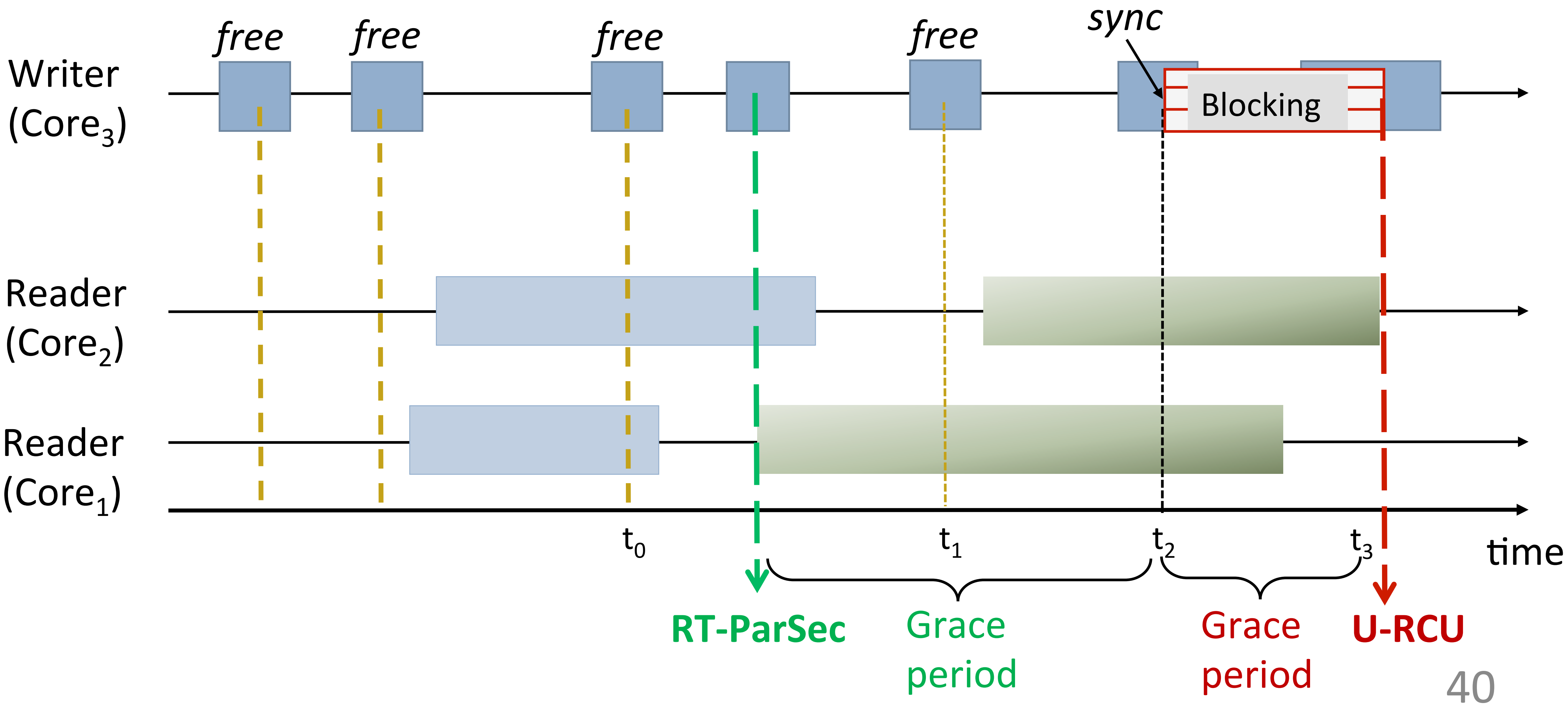
Real-Time SMR: RT-ParSec



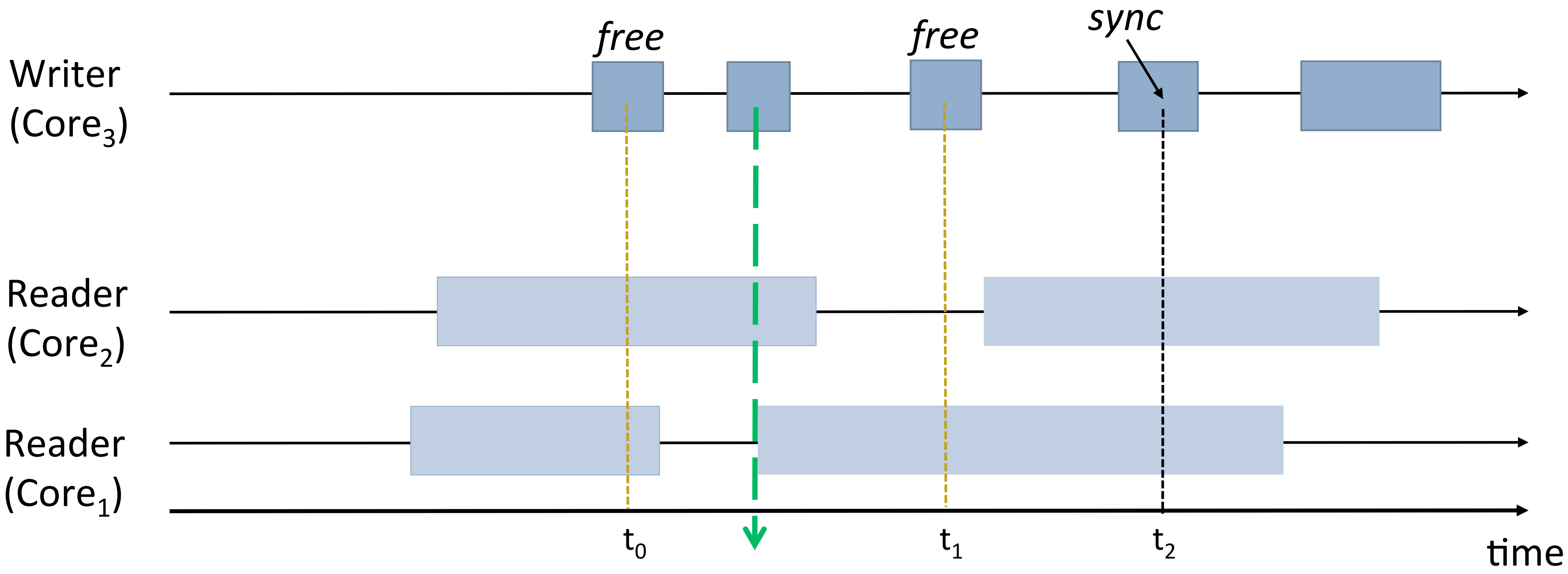
Real-Time SMR: RT-ParSec



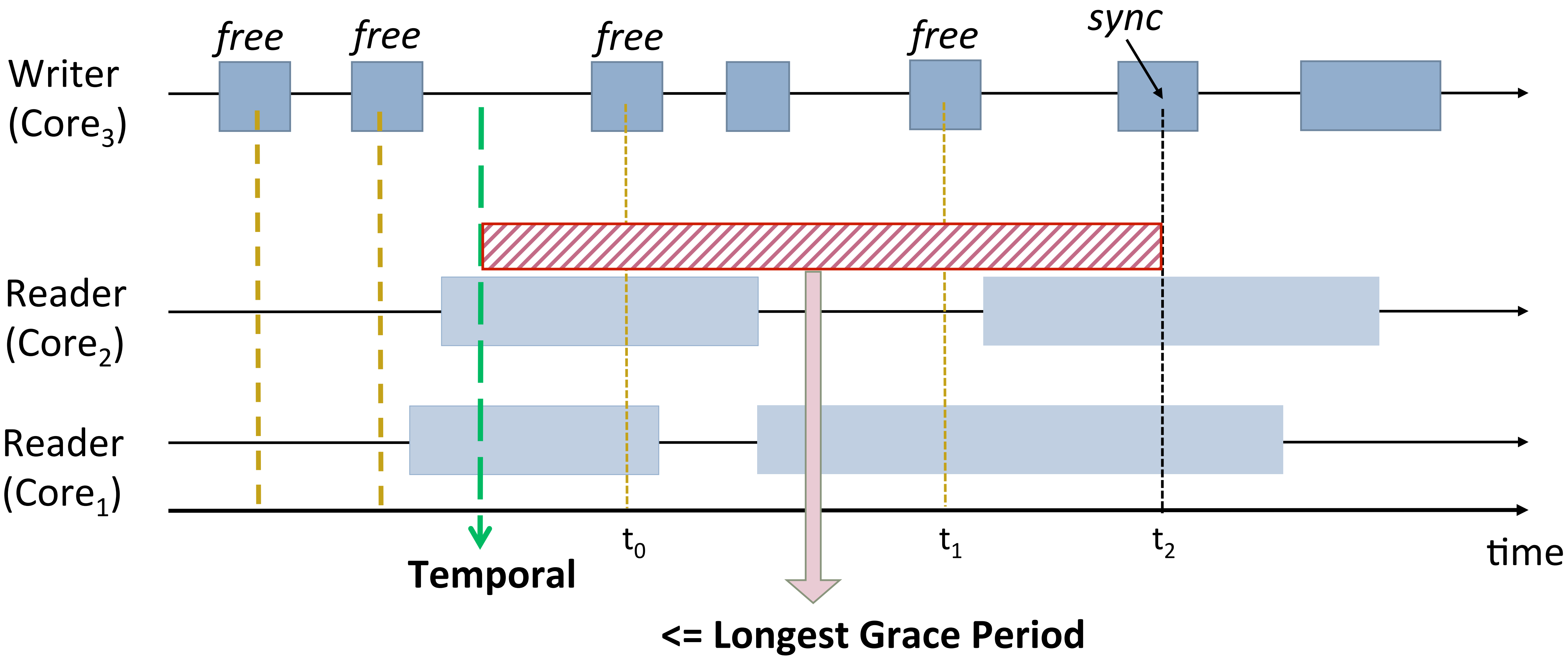
Real-Time SMR: RT-ParSec



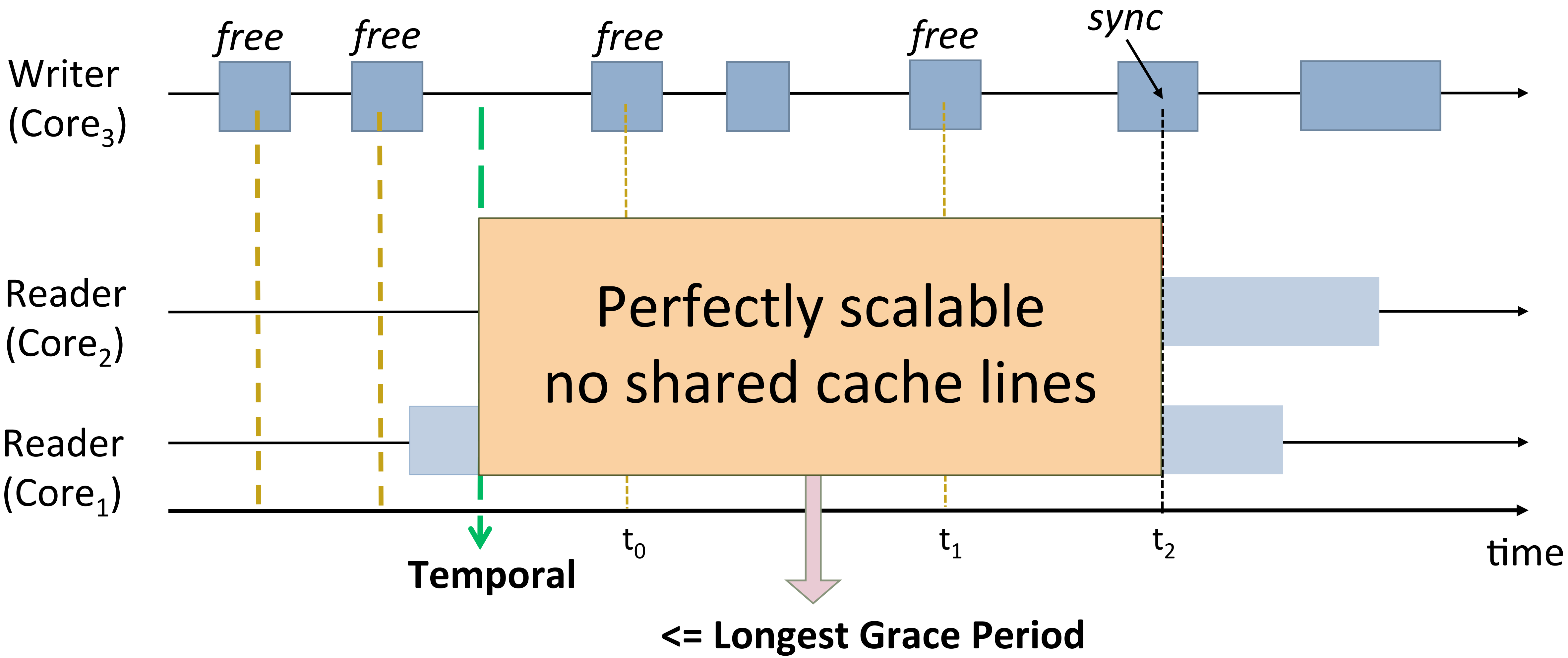
Real-Time SMR: Temporal Quiescence



Real-Time SMR: Temporal Quiescence



Real-Time SMR: Temporal Quiescence



Perfectly scalable
no shared cache lines

<= Longest Grace Period

System Model and Assumptions

- Sporadic task model
- Partitioned fixed priority scheduling
- Non nested requests
- Memory reclamation as quiescence detection task

Worst Case Memory Consumption

Number of writers

$$A(W) = \sum_{i=1}^{N} \Delta(W) * a_i$$

Worst Case Memory Consumption

Number of writers

$$A(W) = \sum_{i=1}^{N} \Delta(W) * a_i$$

Max number of memory allocations

Worst Case Memory Consumption

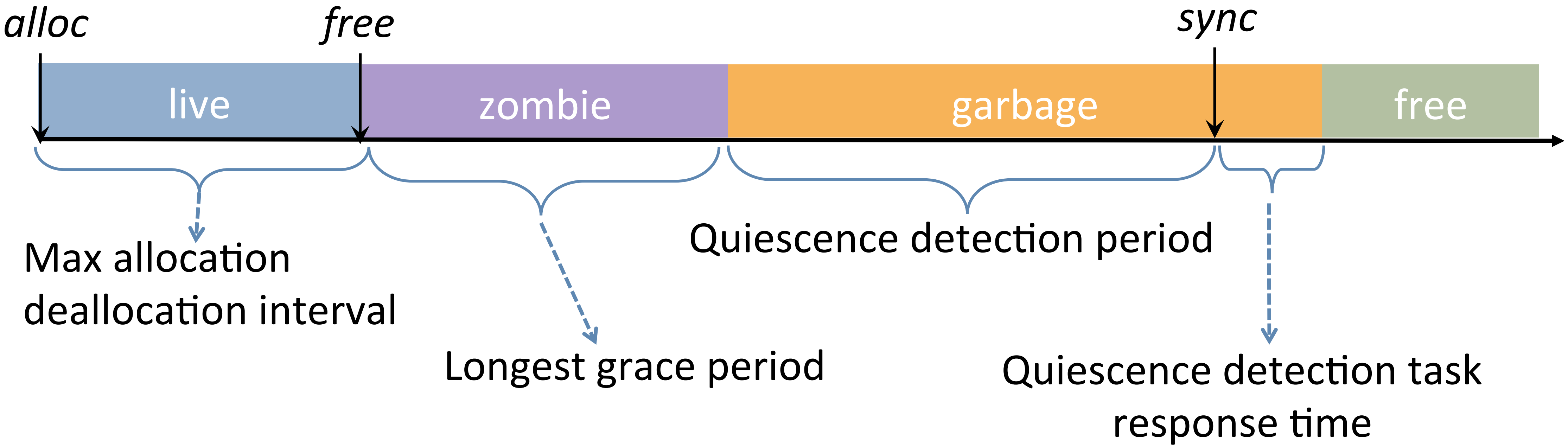
Number of writers

$$A(W) = \sum_{i=1}^{N} \Delta(W) * a_i$$

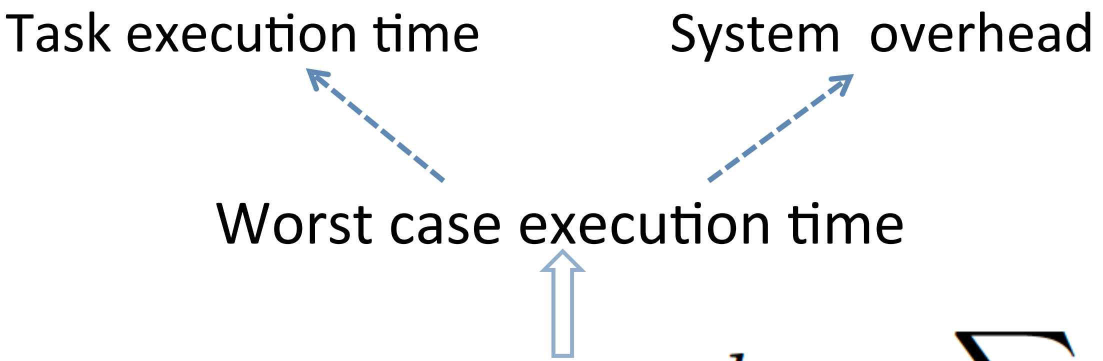
Max number of memory allocations

Time interval where memory accumulates

Worst Case Memory Consumption



Response Time Analysis



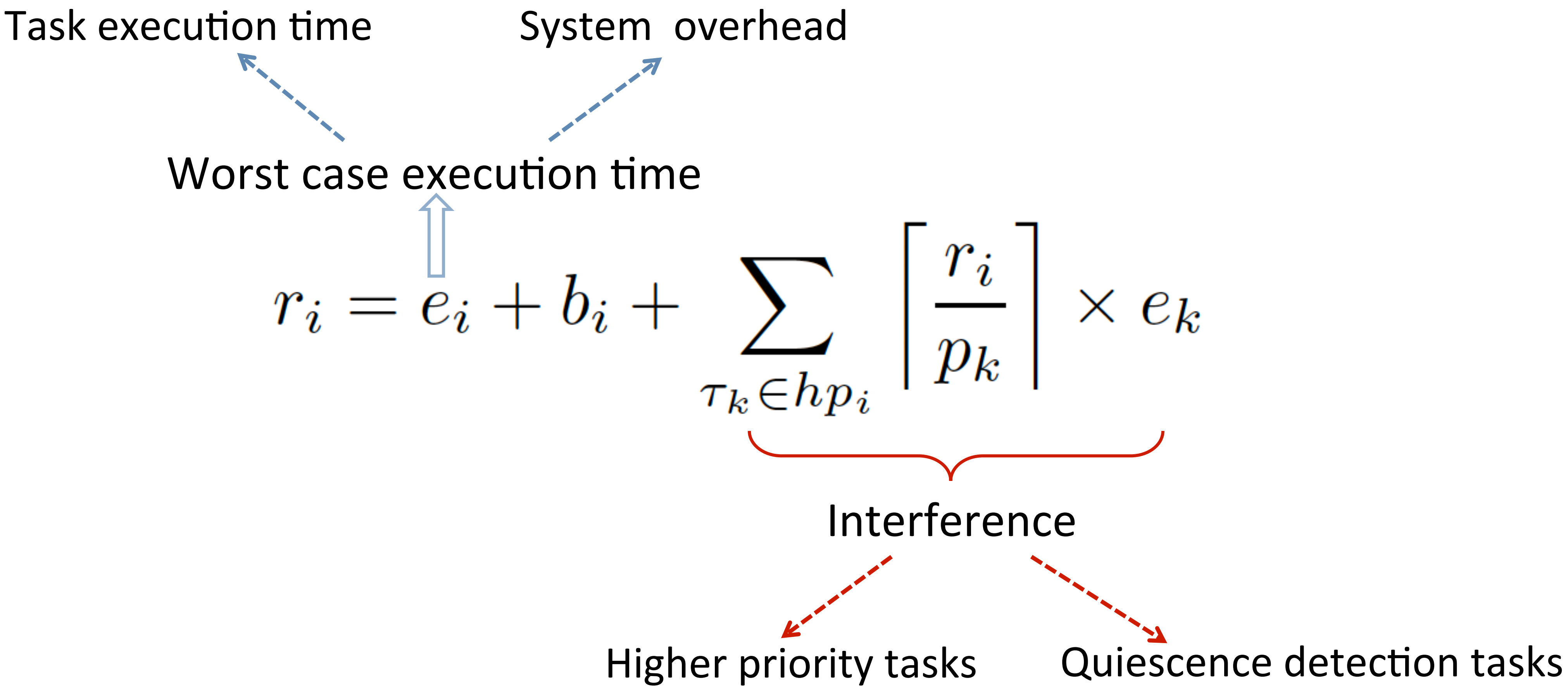
Task execution time

System overhead

Worst case execution time

$$r_i = e_i + b_i + \sum_{\tau_k \in hp_i} \left\lceil \frac{r_i}{p_k} \right\rceil \times e_k$$

Response Time Analysis



Task execution time

System overhead

Worst case execution time

$$r_i = \overset{\Uparrow}{e_i} + b_i + \underbrace{\sum_{\tau_k \in hp_i} \left\lceil \frac{r_i}{p_k} \right\rceil \times e_k}_{\text{Interference}}$$

Higher priority tasks

Quiescence detection tasks

Response Time Analysis

Task execution time

System overhead

Worst case execution time

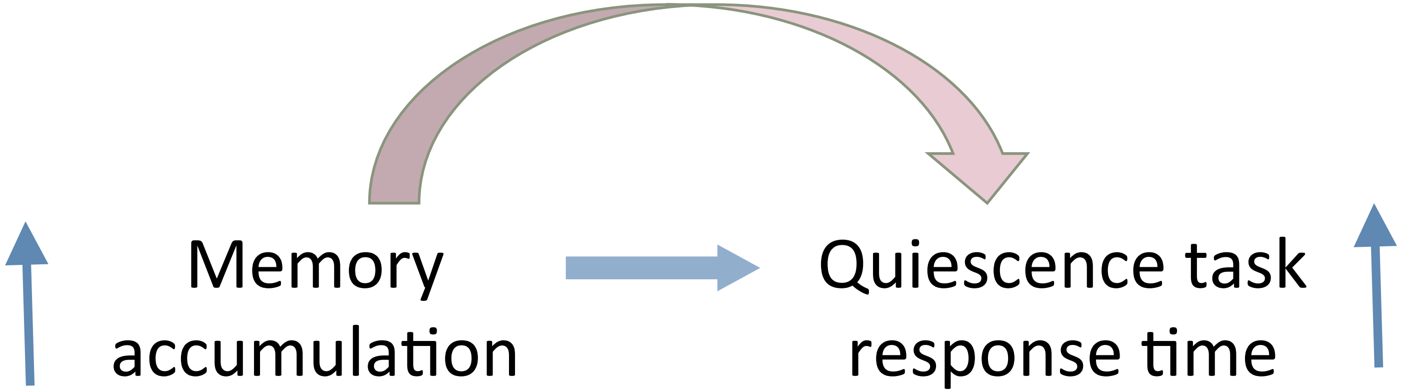
$$r_i = e_i + b_i + \underbrace{\sum_{\tau_k \in hp_i} \left\lceil \frac{r_i}{p_k} \right\rceil \times e_k}_{\text{Interference}}$$

Total blocking time (U-RCU)

Higher priority tasks

Quiescence detection tasks

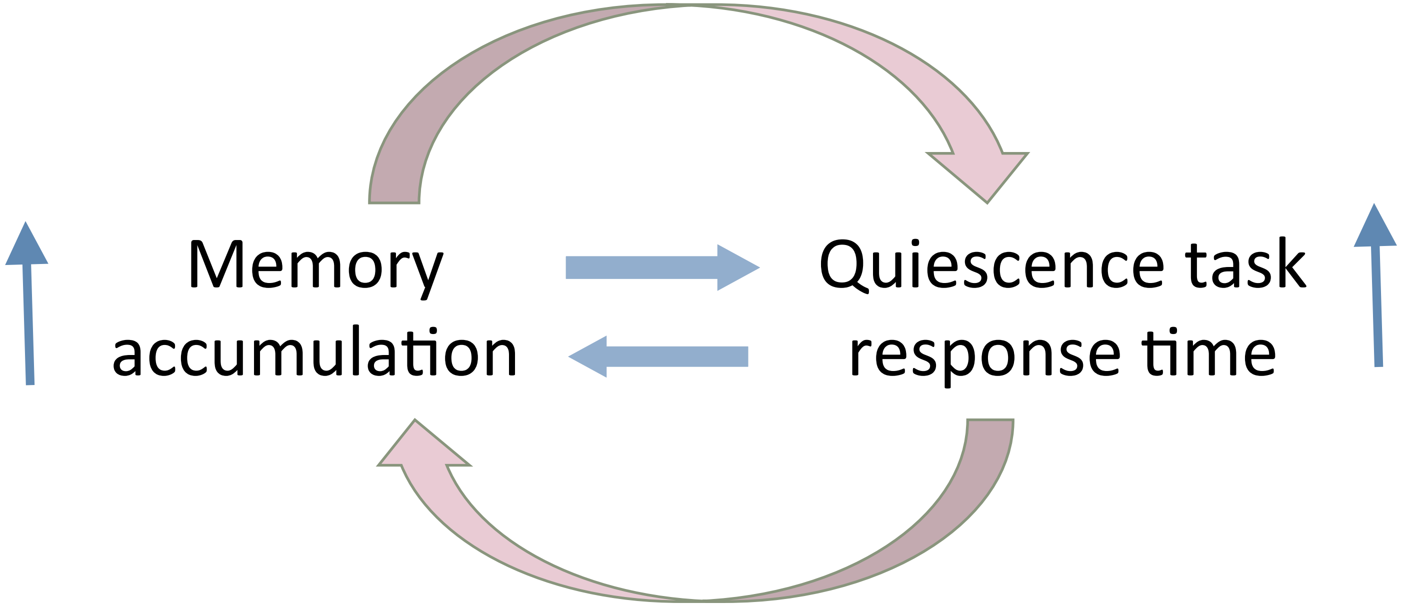
Interdependency



Memory accumulation

Quiescence task response time

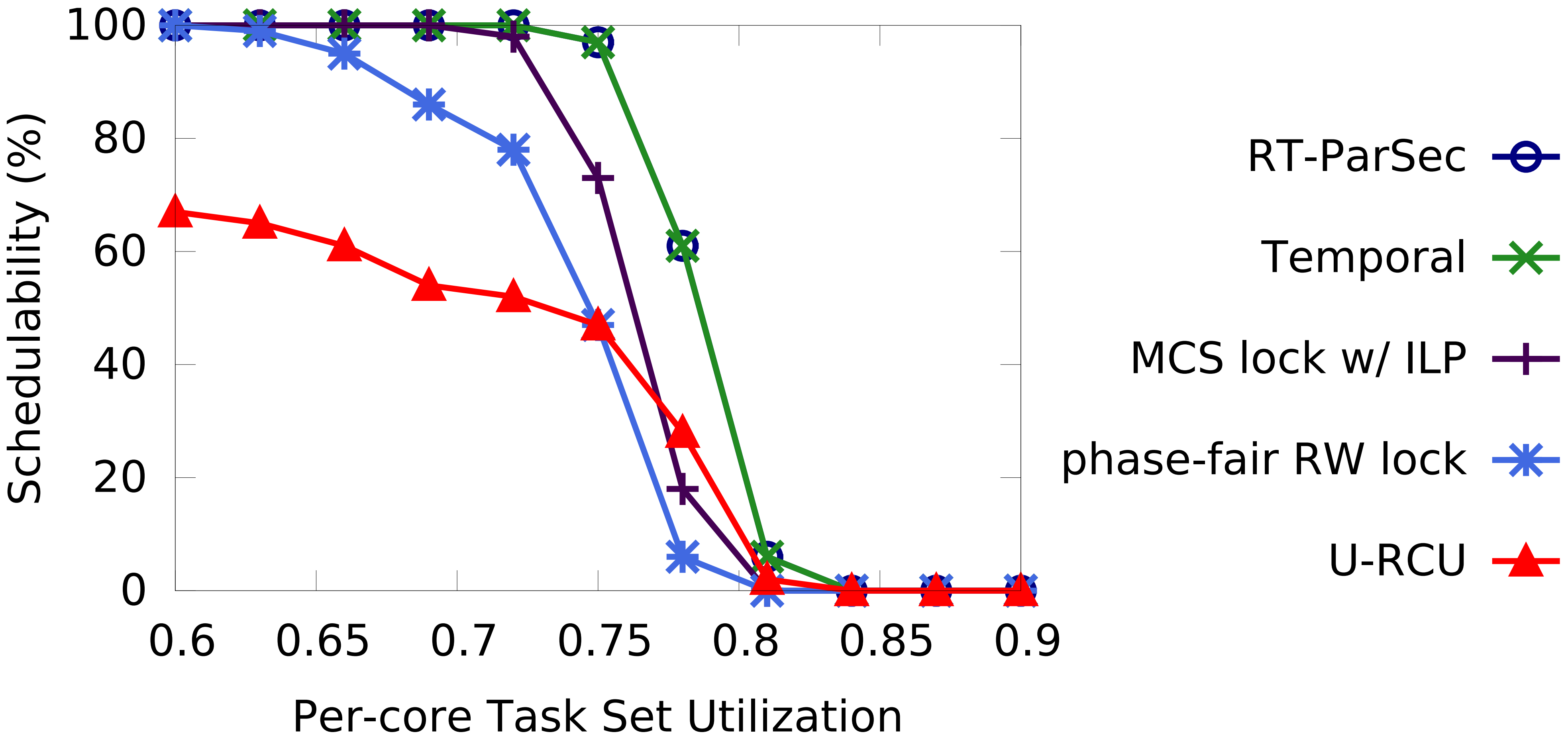
Interdependency



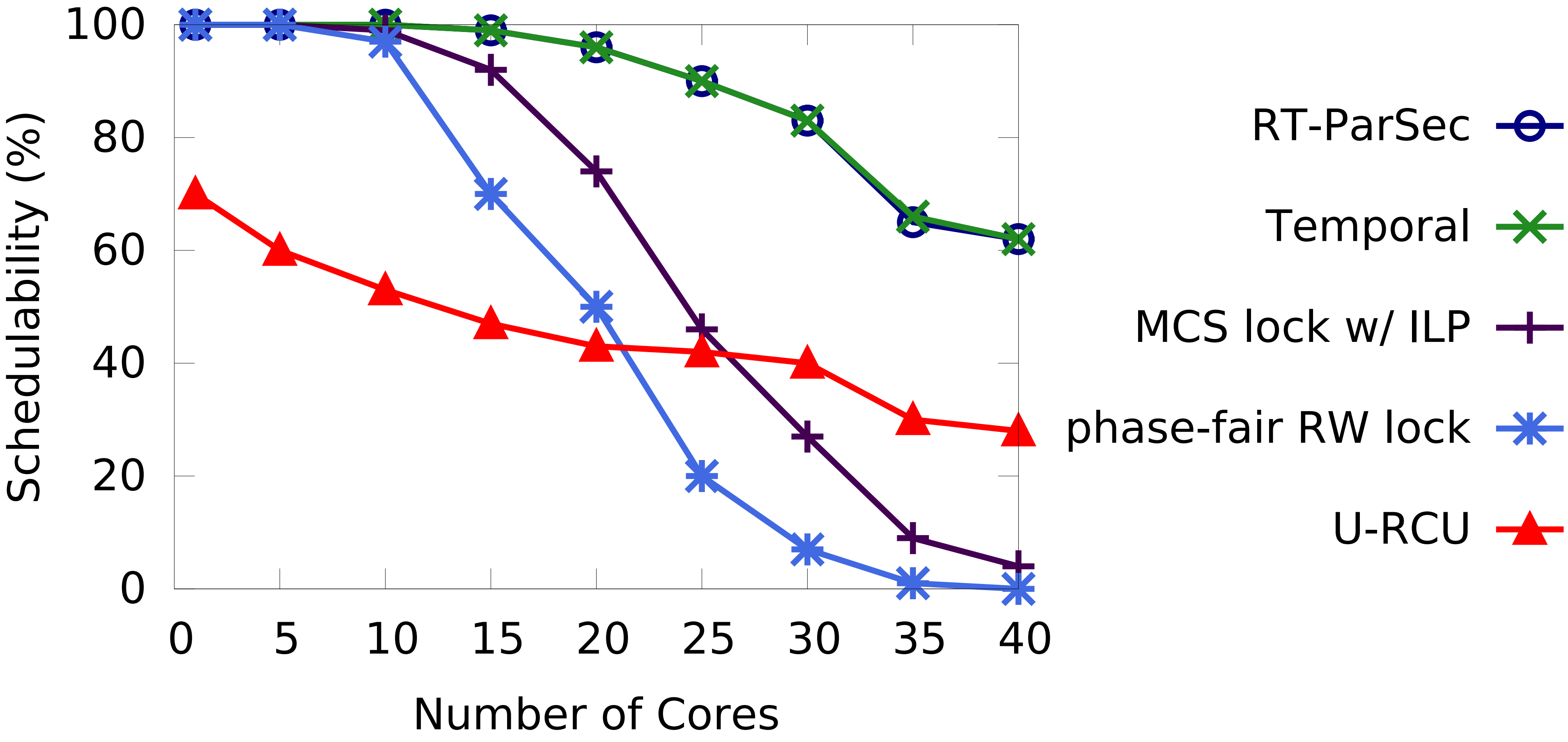
Memory accumulation

Quiescence task response time

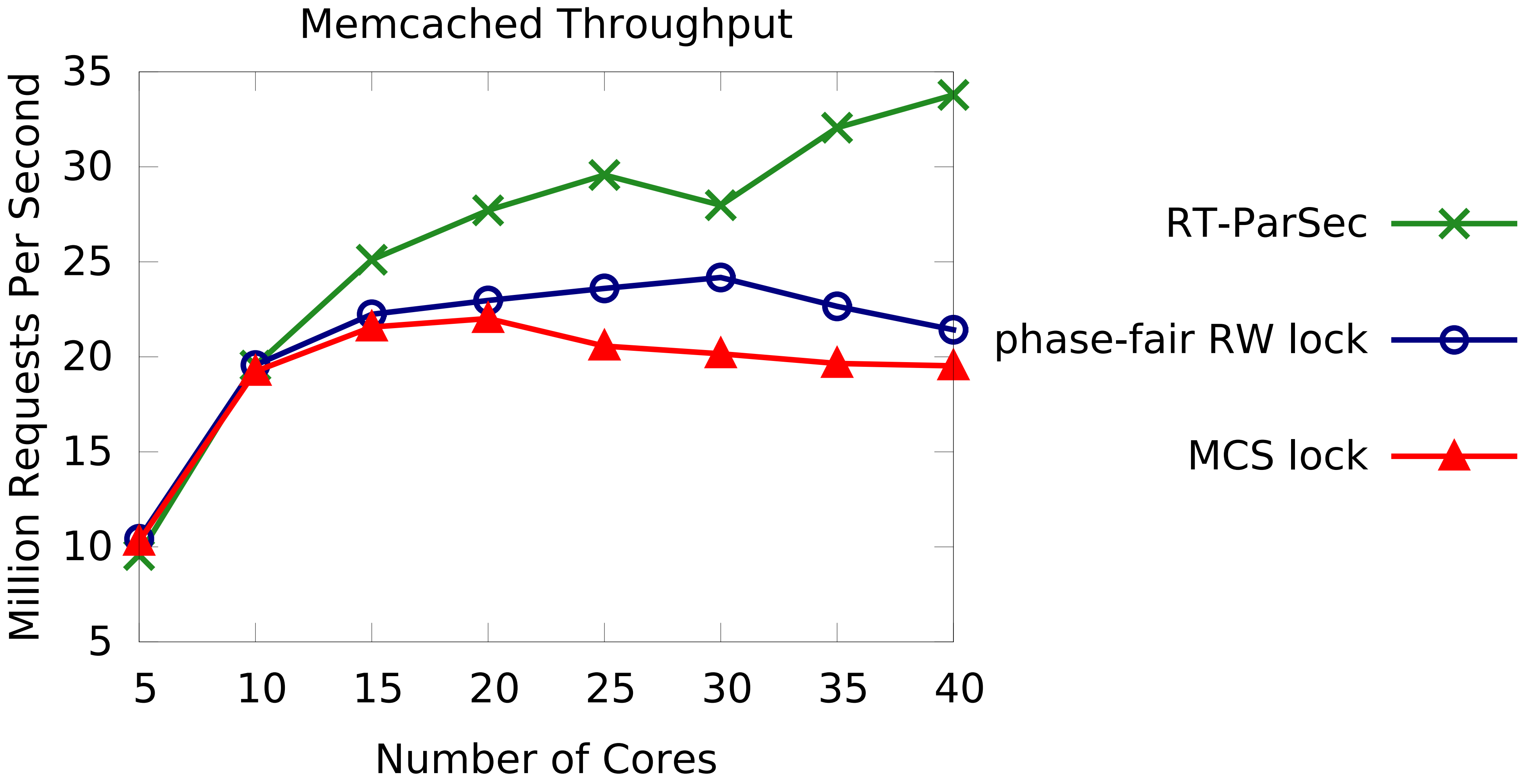
Evaluation: Schedulability Tests



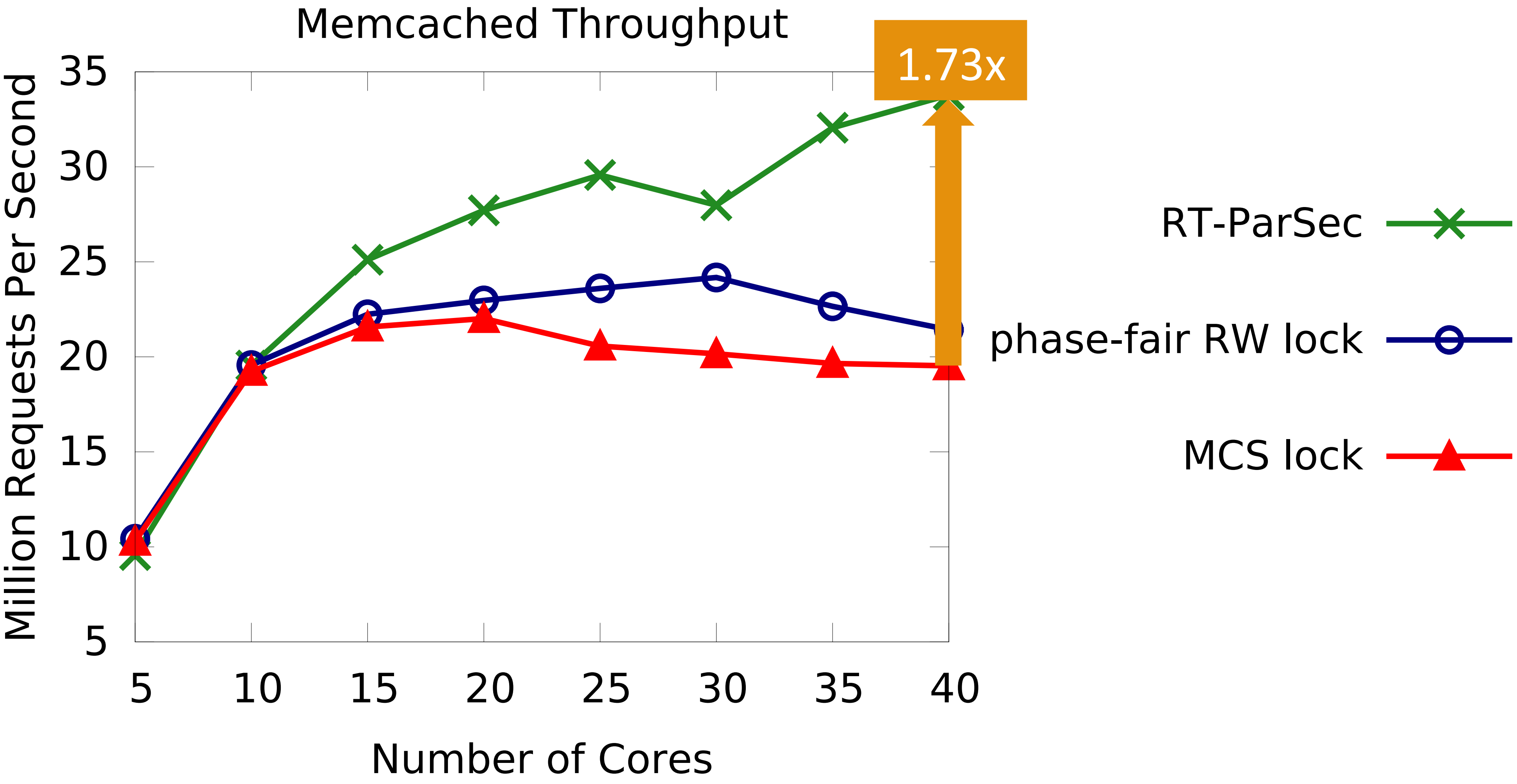
Evaluation: Schedulability Tests



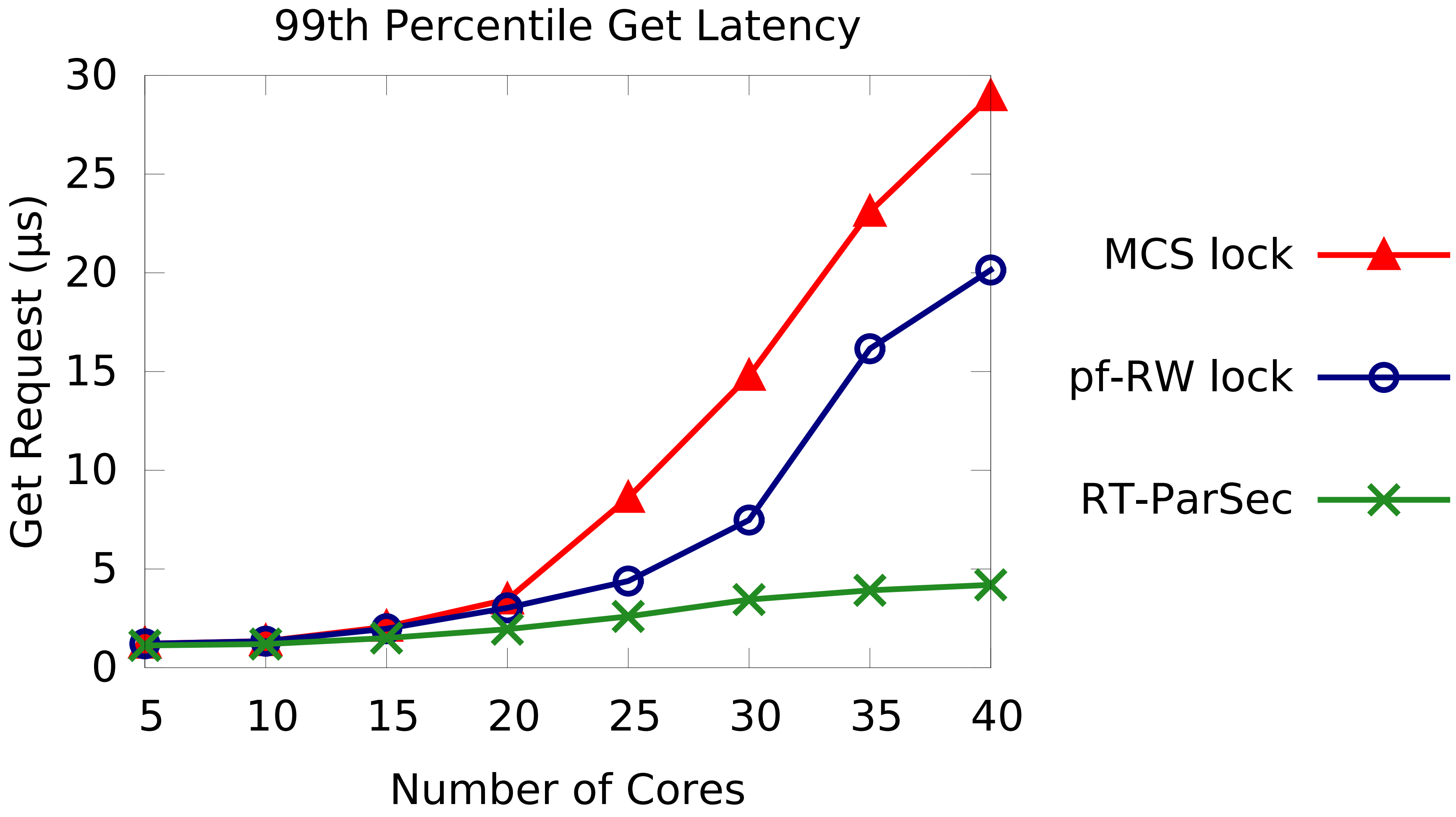
Evaluation: Memcached



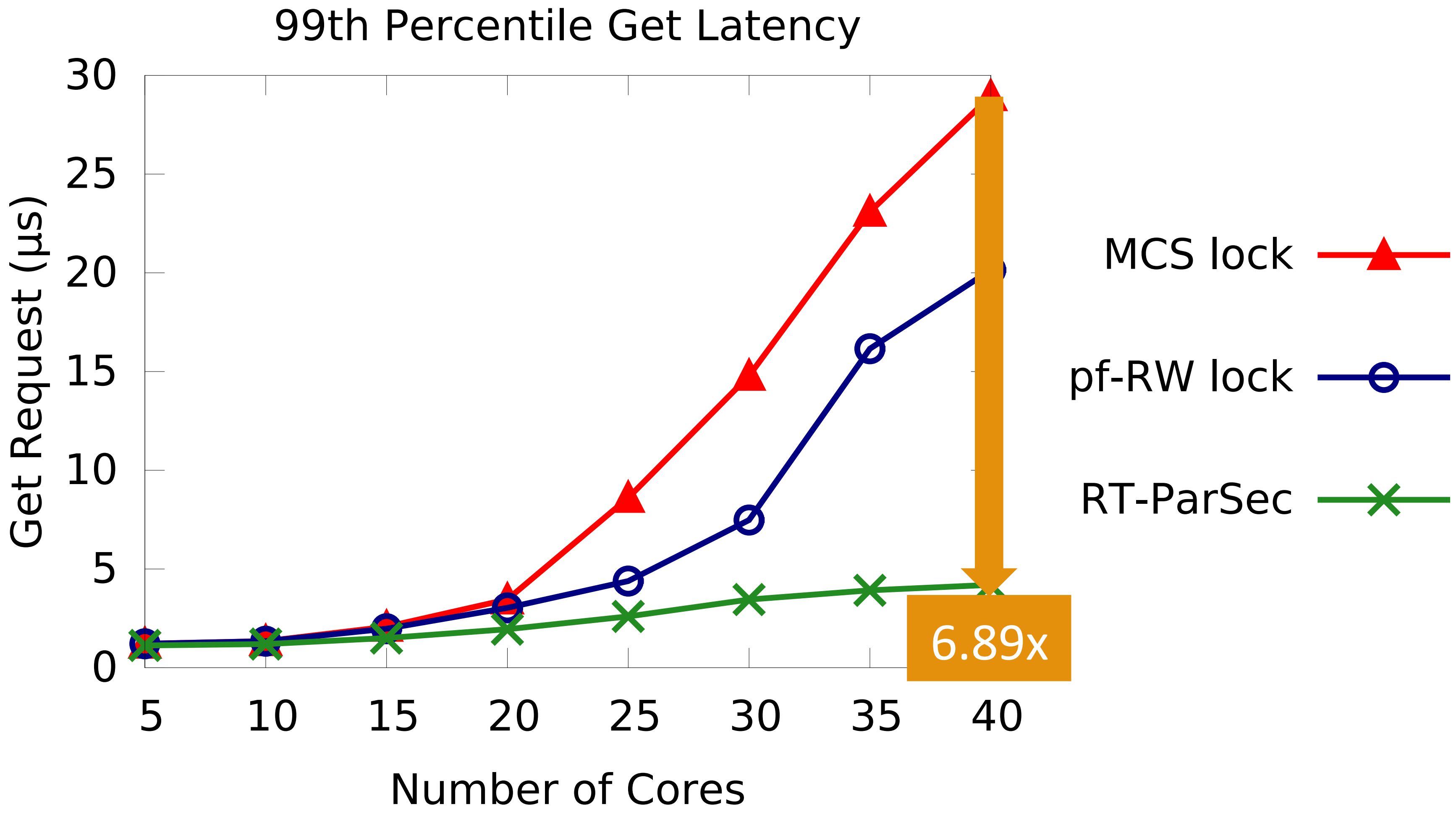
Evaluation: Memcached



Evaluation: Memcached



Evaluation: Memcached



Conclusion

Real-time SMR: Scalable predictable resource sharing

- Scalable and predictable quiescence detection
- Bounds on memory utilization
- Bounds on response time

? || /* */

https://github.com/gwsystems/ps