Scheduling and Locking in Multiprocessor Real-Time Operating Systems

Björn B. Brandenburg
James H. Anderson (Advisor)

Department of Computer Science
The University of North Carolina at Chapel Hill

Supported in part by a Fulbright Fellowship and a UNC Dissertation Completion Fellowship

What is a Real-Time System?

“right answer at the right time”

predictability = ***a priori*** **validation of temporal correctness**

Image credit: DaimlerChrysler AG via Wikimedia Commons

Scheduling in a Real-Time OS (RTOS)

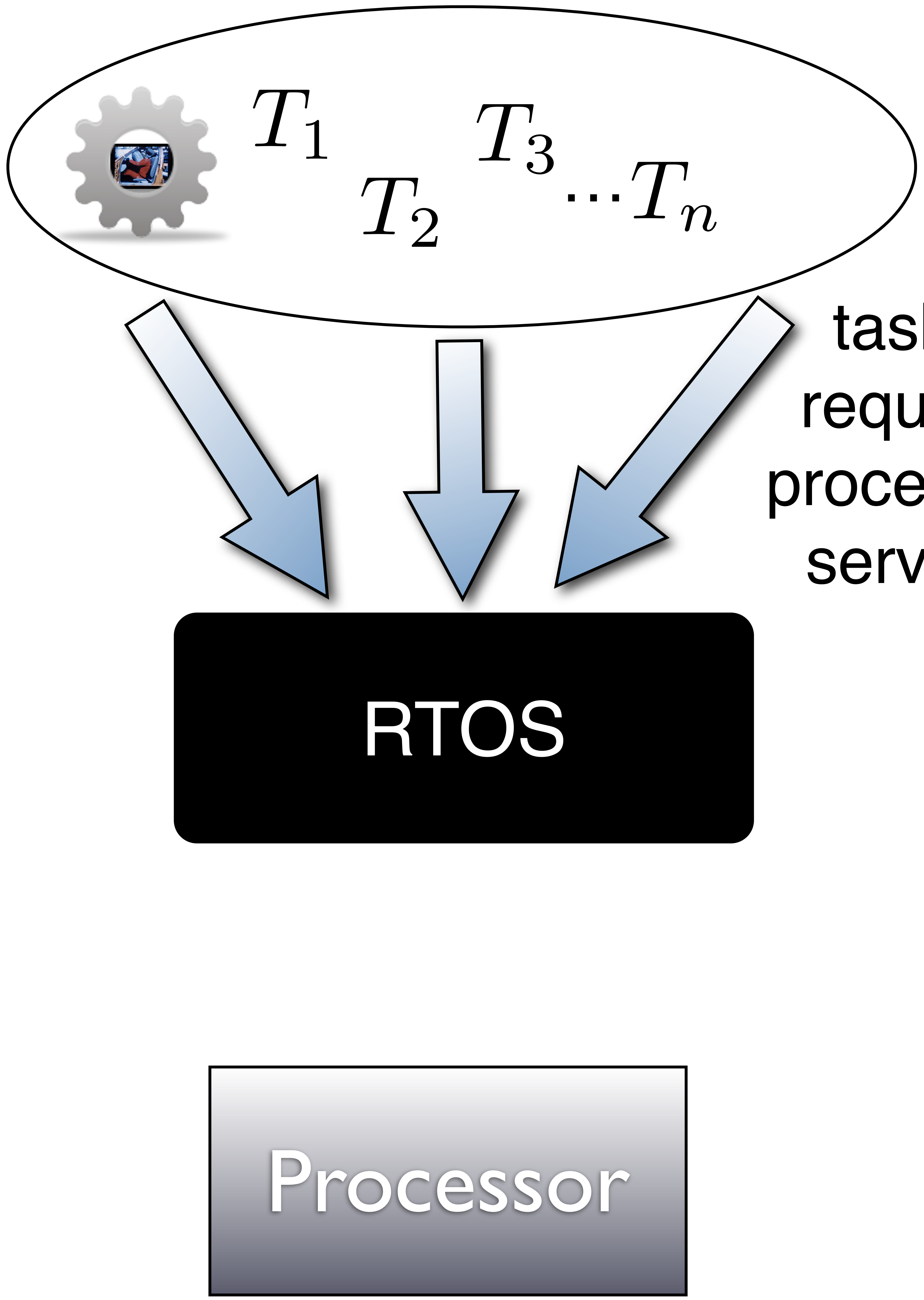
T_1 T_2 T_3 $\ldots T_n$

A set of recurrent tasks with temporal constraints (= deadlines).
Example: poll acceleration sensor every 10ms

RTOS

Processor

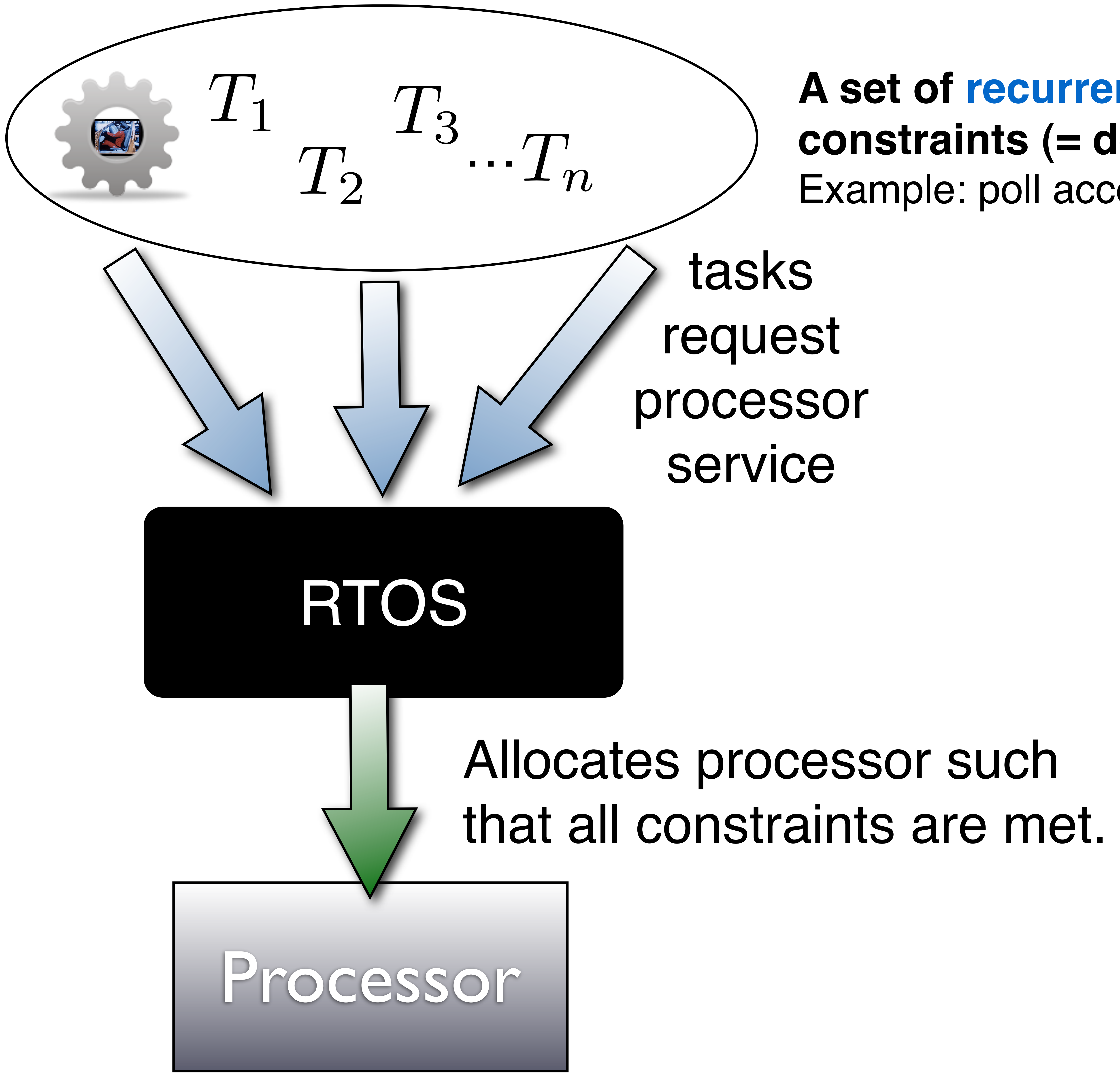
Scheduling in a Real-Time OS (RTOS)



A set of recurrent tasks with temporal constraints (= deadlines).
Example: poll acceleration sensor every 10ms

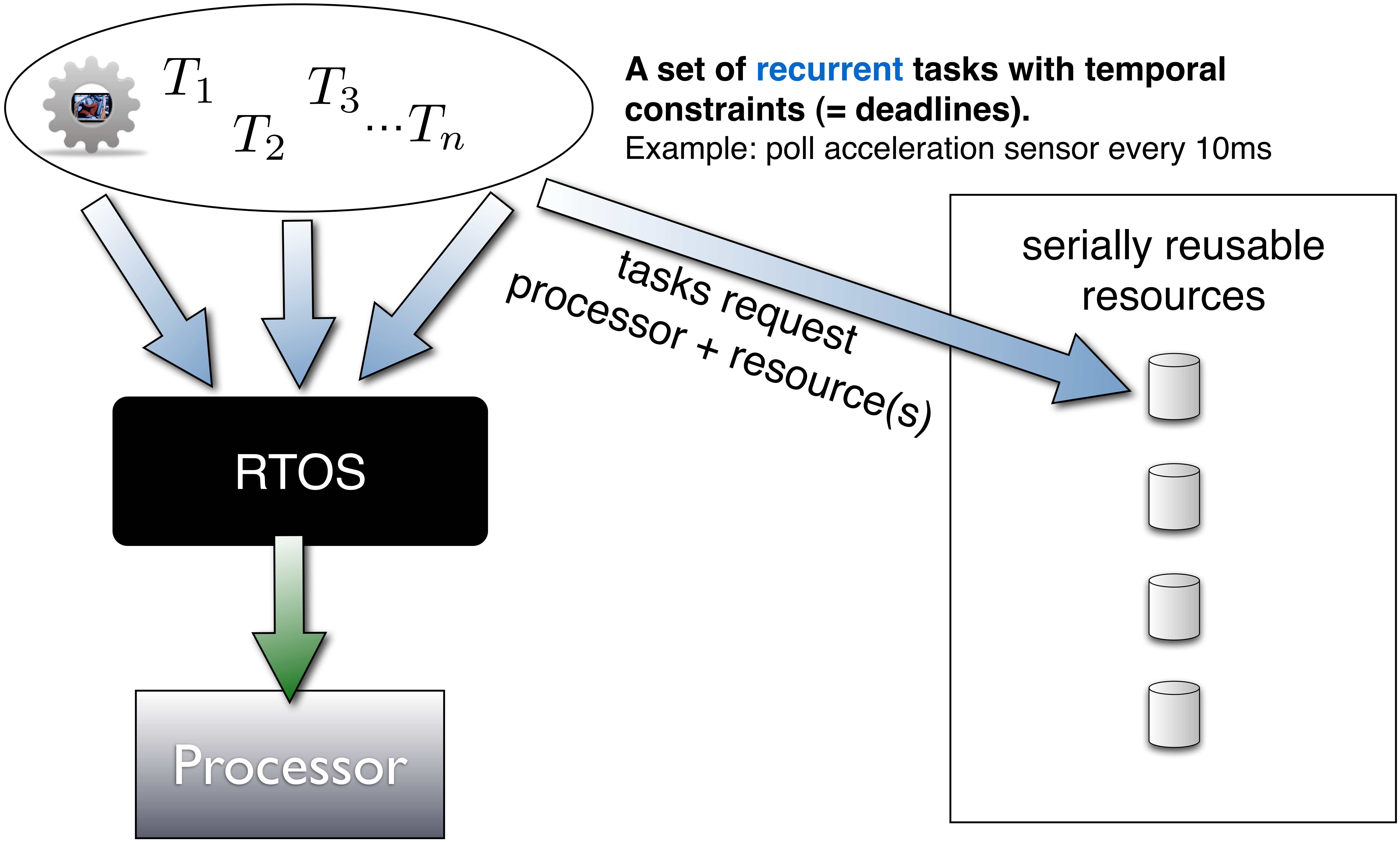
Scheduling in a Real-Time OS (RTOS)

A set of recurrent tasks with temporal constraints (= deadlines).
Example: poll acceleration sensor every 10ms



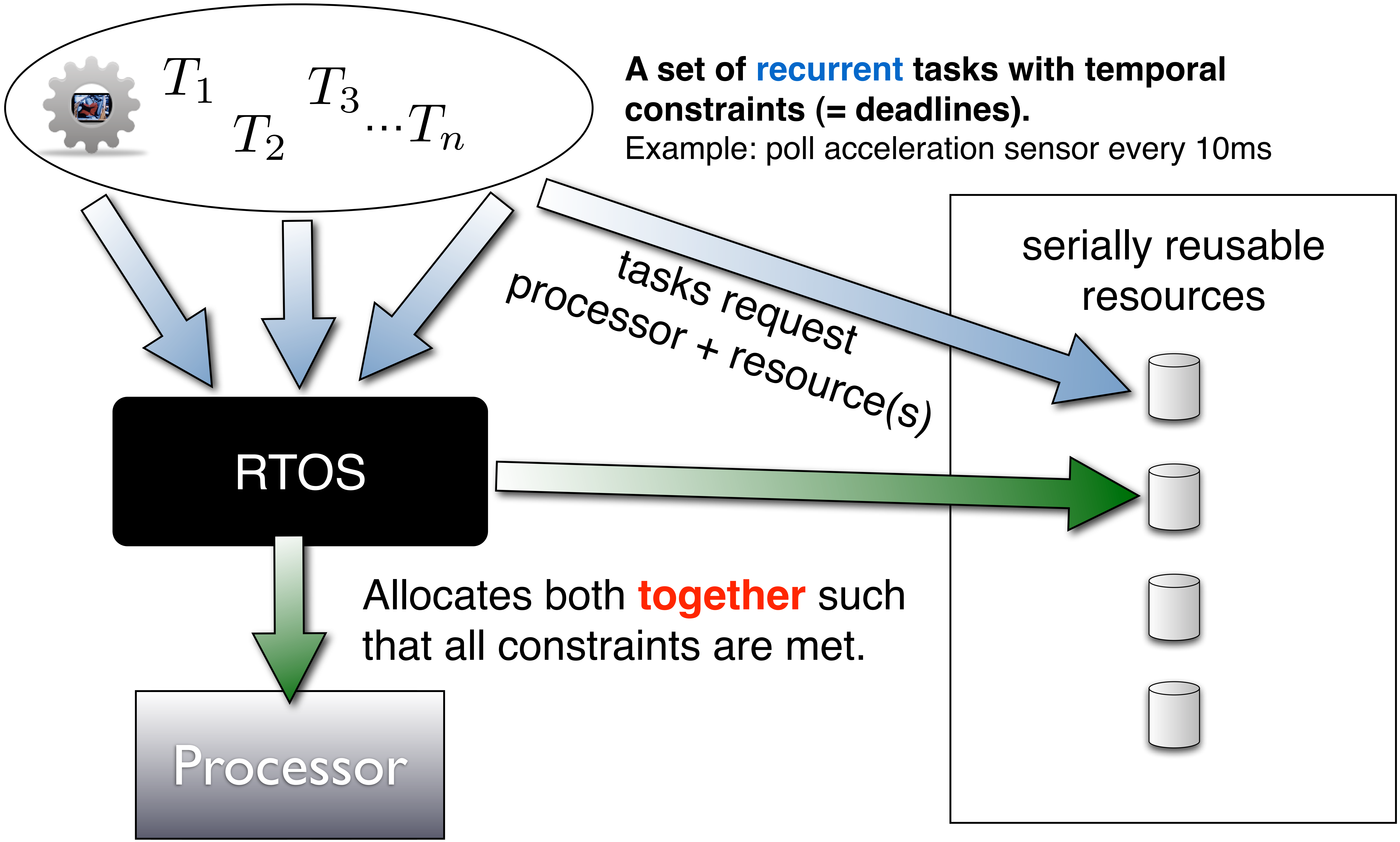
Locking in an RTOS

A set of recurrent tasks with temporal constraints (= deadlines).
Example: poll acceleration sensor every 10ms



Locking in an RTOS

A set of recurrent tasks with temporal constraints (= deadlines).
Example: poll acceleration sensor every 10ms



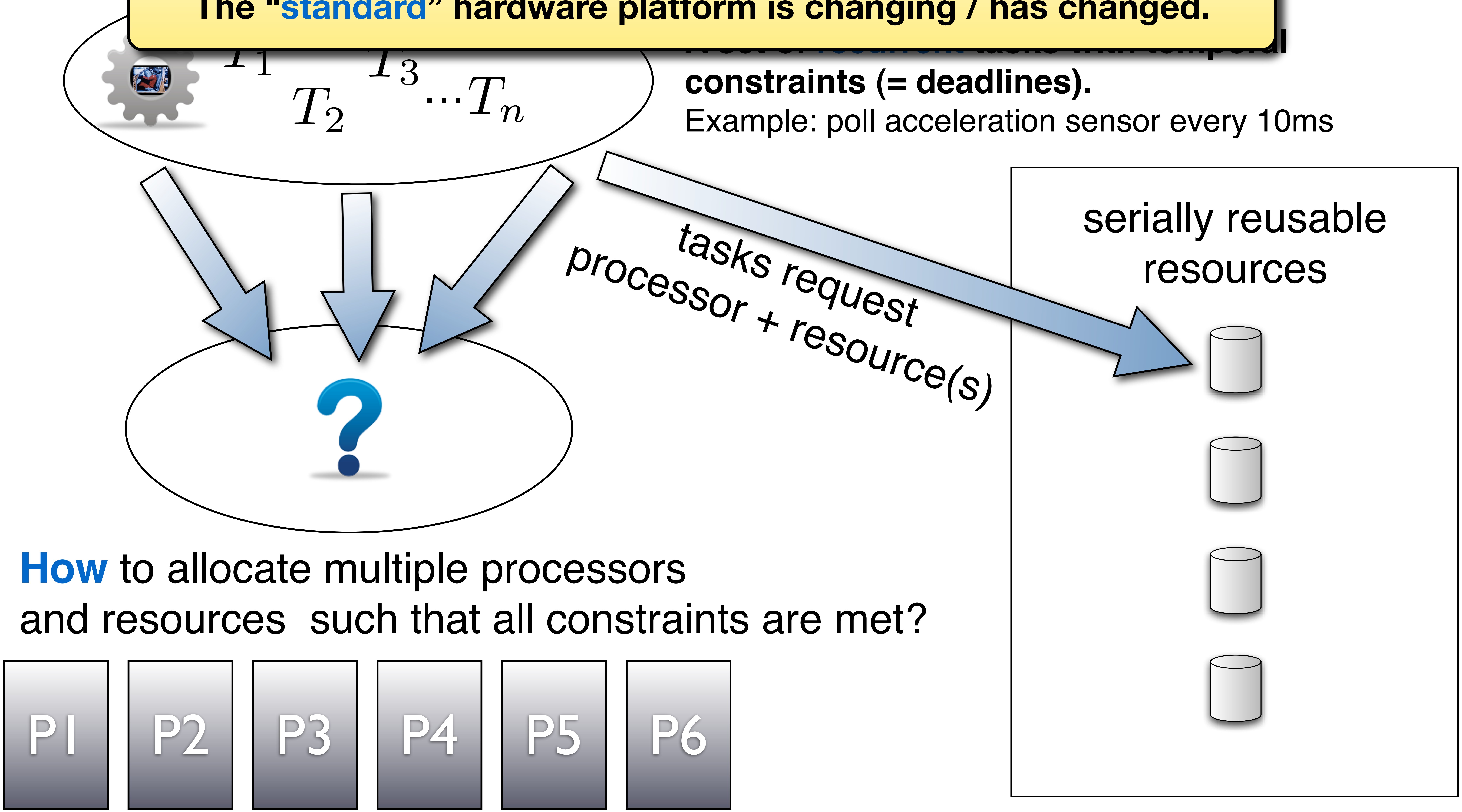
Allocates both **together** such that all constraints are met.

The Emergence of Multicore Processors

The "standard" hardware platform is changing / has changed.

constraints (= deadlines).

Example: poll acceleration sensor every 10ms



How to allocate multiple processors
and resources such that all constraints are met?

Why Real-Time on Multicore?

*To reduce **size**, **weight**, and **power** (**SWaP**) requirements.*

*Cost, **availability**: commercial-off-the-shelf (COTS) processors likely to be multicore chips.*

*High **computational demands**: HD media, computer vision, motion planning…*

Why Real-Time on Multicore?

*To reduce **size**, **weight**, and **power** (**SWaP**) requirements.*

Motivating example:

MARVIN Mk II*: unmanned autonomous vehicle (UAV)*

Technische Universität Berlin

Musial et al., 2006

Mission
Detect forest fires
during dry summer months.

Why Real-Time on Multicore?

*To reduce **size**, **weight**, and **power** (**SWaP**) requirements.*

Motivating example:

***MARVIN Mk II**: unmanned autonomous vehicle (UAV)*

Payload: pan & tilt camera and infrared sensor.

UAV tethered to ground-based mission planning.

Technische Universität Berlin

Musial et al., 2006

Why Real-Time on Mul

Two computers for flight controller + payload

2x CPUs, 2x power supply (batteries), 2x cabling, 2x cooling…

Mission planning

Not enough on-board computational resources!

Would need more space, weight, power, cooling, maintenance…

***MARVIN Mk II**: unmanned autonomous vehicle (UAV)*

Payload: pan & tilt camera and infrared sensor.

UAV tethered to ground-based mission planning.

Why not use just one, more powerful multicore chip…?

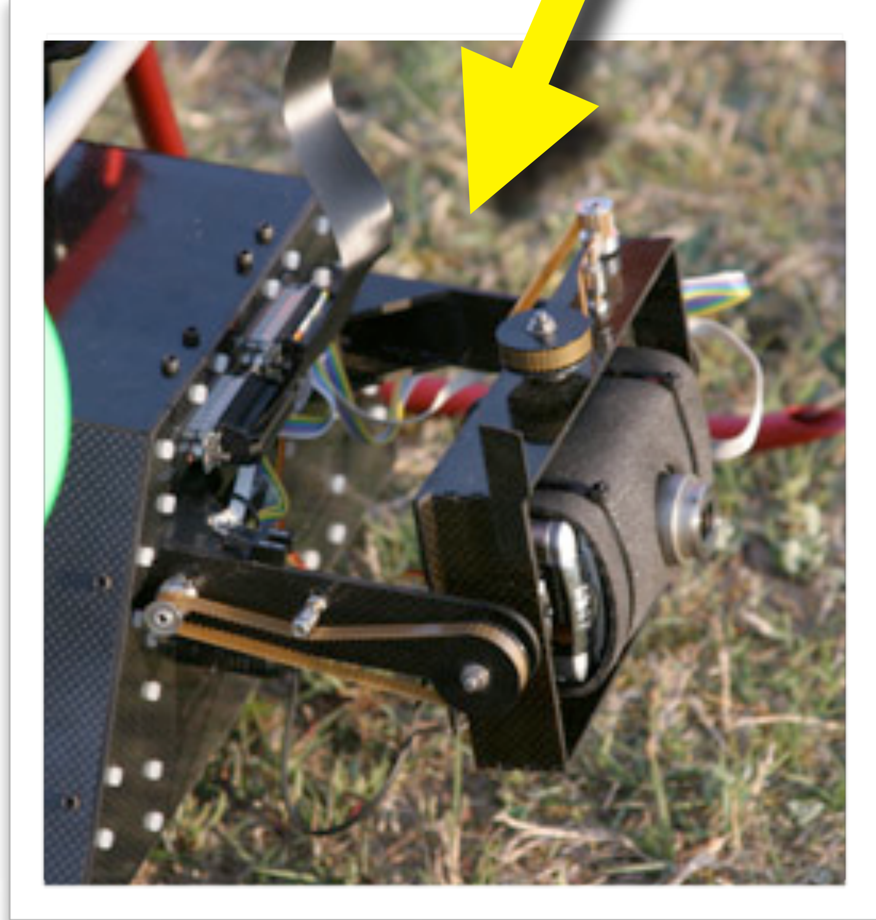
Predictable temporal isolation required.

Temporal failure =
wobbly flight or crash.

Temporal failure =
briefly "looks in wrong direction."

M

***MARVIN Mk II**: unmanned autonomous vehicle (UAV)*



Payload:
pan & tilt camera
and infrared sensor.

UAV tethered to ground-based mission planning.

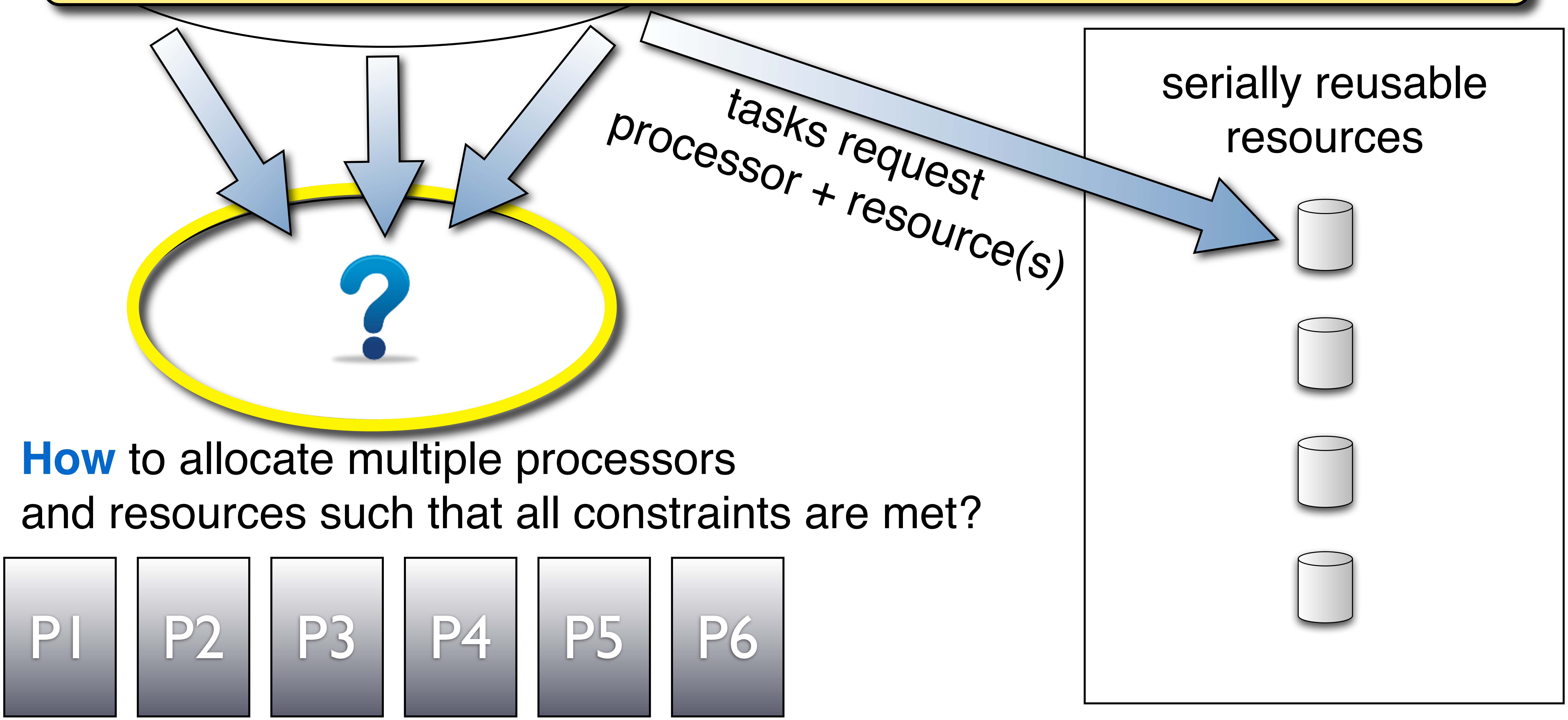
Technische Universität Berlin

Musial et al., 2006

Temporal failure =
UAV "hesitates" a little longer.

Predictable Real-Time Kernel

Algorithms must be both
analytically sound and efficiently implementable.



How to allocate multiple processors
and resources such that all constraints are met?

Capacity Loss

Processor utilization that **cannot be allocated** to real-time tasks without risking **temporal failure**.

(*i.e.*, idle time required to meet all timing constraints)

Two main causes:

1. **Algorithmic limitations** (non-optimal scheduling decisions).
2. **Runtime overheads** (RTOS inefficient).

Thesis Statement

When both overhead-related and algorithmic capacity loss are considered on a current multicore platform,

Part 1:

Which scheduler to use.

Parts 2 & 3: How to implement locking.

(underlined terms will be defined shortly)

Thesis Statement

When both overhead-related and algorithmic capacity loss are considered on a current multicore platform,

(i) partitioned scheduling is preferable to global and clustered approaches in the hard real-time case,

(ii) partitioned earliest-deadline first (P-EDF) scheduling is superior to partitioned fixed-priority (P-FP) scheduling and

(iii) clustered scheduling can be effective in reducing the impact of bin-packing limitations in the soft real-time case. Further,

Parts 2 & 3: How to implement locking.

(underlined terms will be defined shortly)

Thesis Statement

When both overhead-related and algorithmic capacity loss are considered on a current multicore platform,

(i) partitioned scheduling is preferable to global and clustered approaches in the hard real-time case,

(ii) partitioned earliest-deadline first (P-EDF) scheduling is superior to partitioned fixed-priority (P-FP) scheduling and

(iii) clustered scheduling can be effective in reducing the impact of bin-packing limitations in the soft real-time case. Further,

(iv) multiprocessor locking protocols exist that are both efficiently implementable and asymptotically optimal with regard to the maximum duration of blocking.

(underlined terms will be defined shortly)

Part 1

Scheduling

Scheduling in Theory and Practice

Scheduling Theory:

"we consider overheads to be **negligible**"

RTOS Developers:

overheads, overheads, overheads…

Scheduling in Theory and Practice

Scheduling Theory:

"we consider overheads to be **negligible**"

My contribution: an evaluation that reflects both overhead-related and algorithmic capacity loss.

RTOS Developers:

overheads, overheads, overheads…

Methodology & Case Study

Choosing a Scheduler for a 24-Core Intel System

Linux Testbed for Multiprocessor Scheduling in Real-Time systems

RTOS Platform:

- ➡ Real-time Linux extension (v2.6.36).
- ➡ Supports **scheduler plugins**.
- ➡ Principle developer, project lead.
- ➡ Since 2006: 9 releases, spanning 17 kernel versions.

Methodology & Case Study

Choosing a Scheduler for a 24-Core Intel System

Linux Testbed for Multiprocessor Scheduling in Real-Time systems

RTOS Platform:

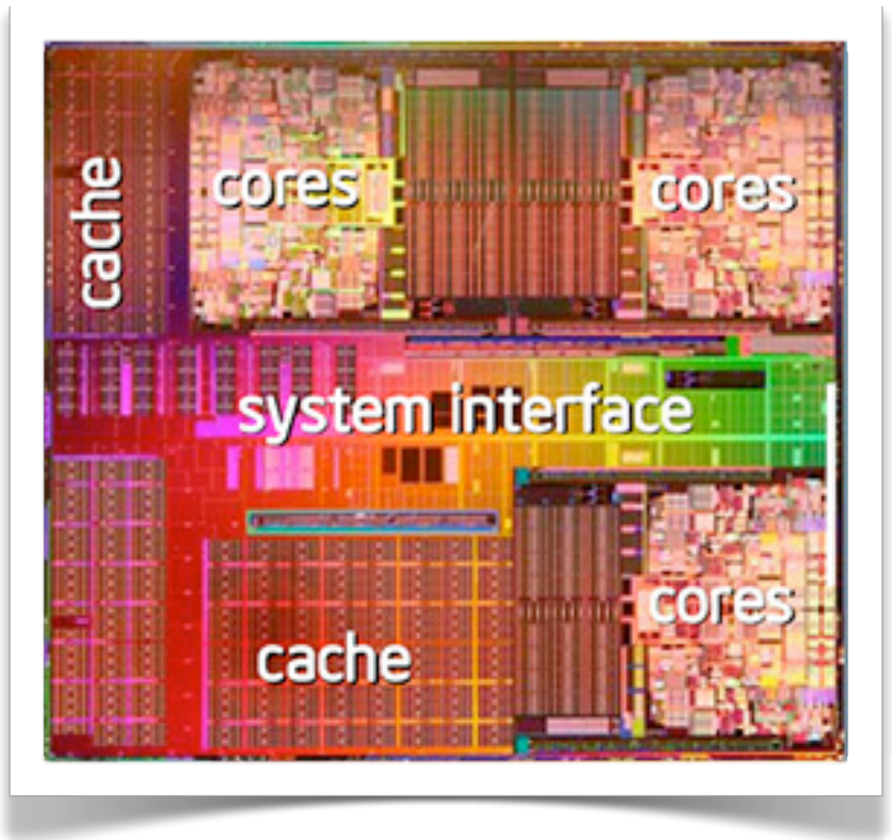
- Real-time Linux extension (v2.6.36).
- Supports **scheduler plugins**.
- Principle developer, project lead.
- Since 2006: 9 releases, spanning 17 kernel versions.

LITMUS^RT

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

HW Platform:

- 4 sockets
- 6 cores per socket (Intel 64bit Xeon L7455)
- 3 levels of cache (2 shared + 1 private)
- Details later…



Methodology & Case Study

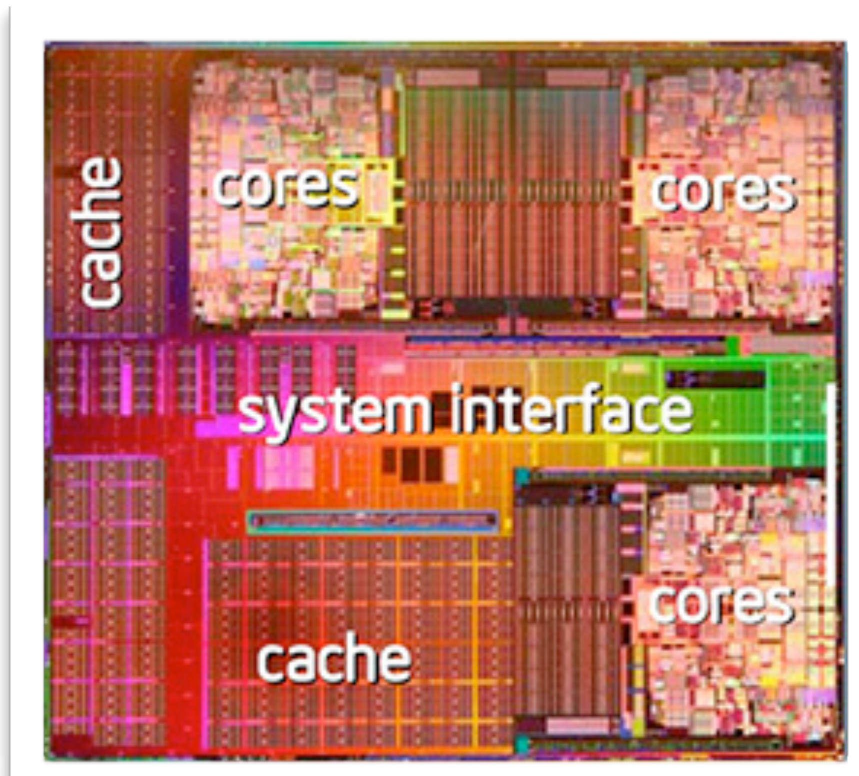
Choosing a Scheduler for a 24-Core Intel System

Linux Testbed for Multiprocessor Scheduling in Real-Time systems

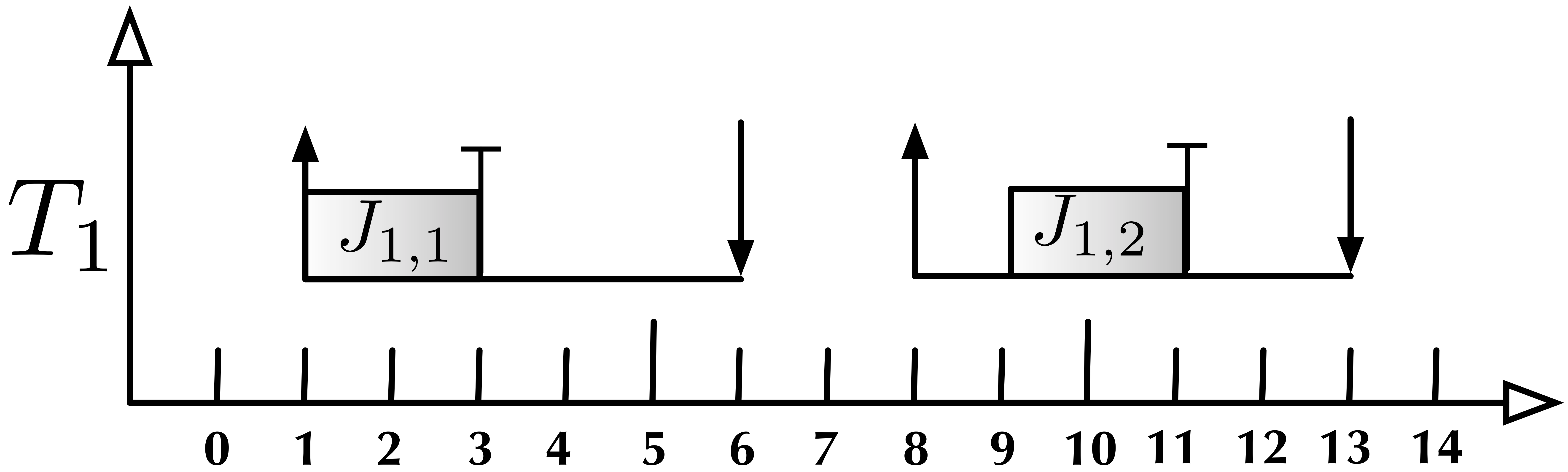
Next:
background review.

Then:
case study details and results.

HW Platform:
- 4 sockets
- 6 cores per socket (Intel 64bit Xeon L7455)
- 3 levels of cache (2 shared + 1 private)
- Details later…



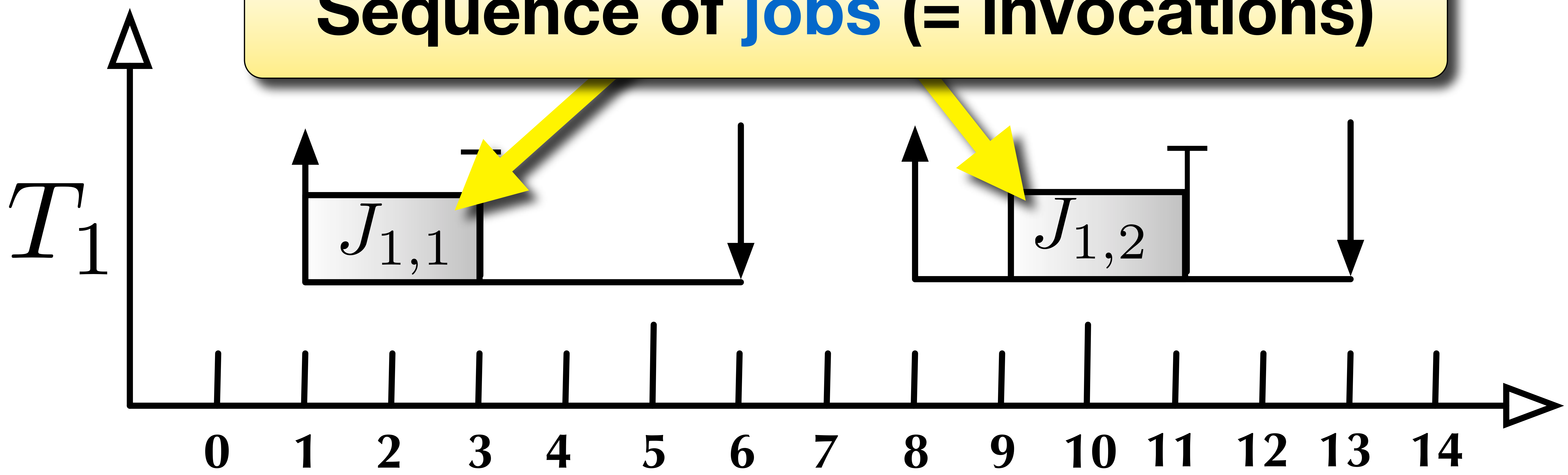
Sporadic Task Model




```
void recurrent_task() {
  while (true) {
    wait_for_event();
    process_event();
    signal_event_processed();
  }
}
```

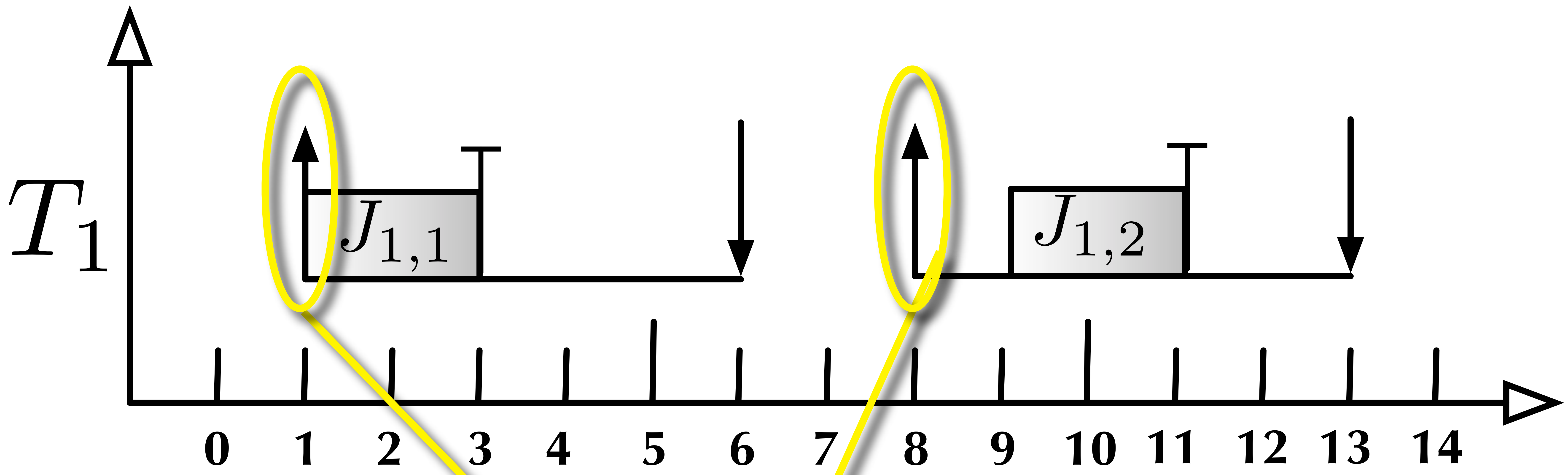
Sporadic Task Model

Sequence of jobs (= invocations)




```
void recurrent_task() {
  while (true) {
    wait_for_event();
    process_event();
    signal_event_processed();
  }
}
```

Sporadic Task Model

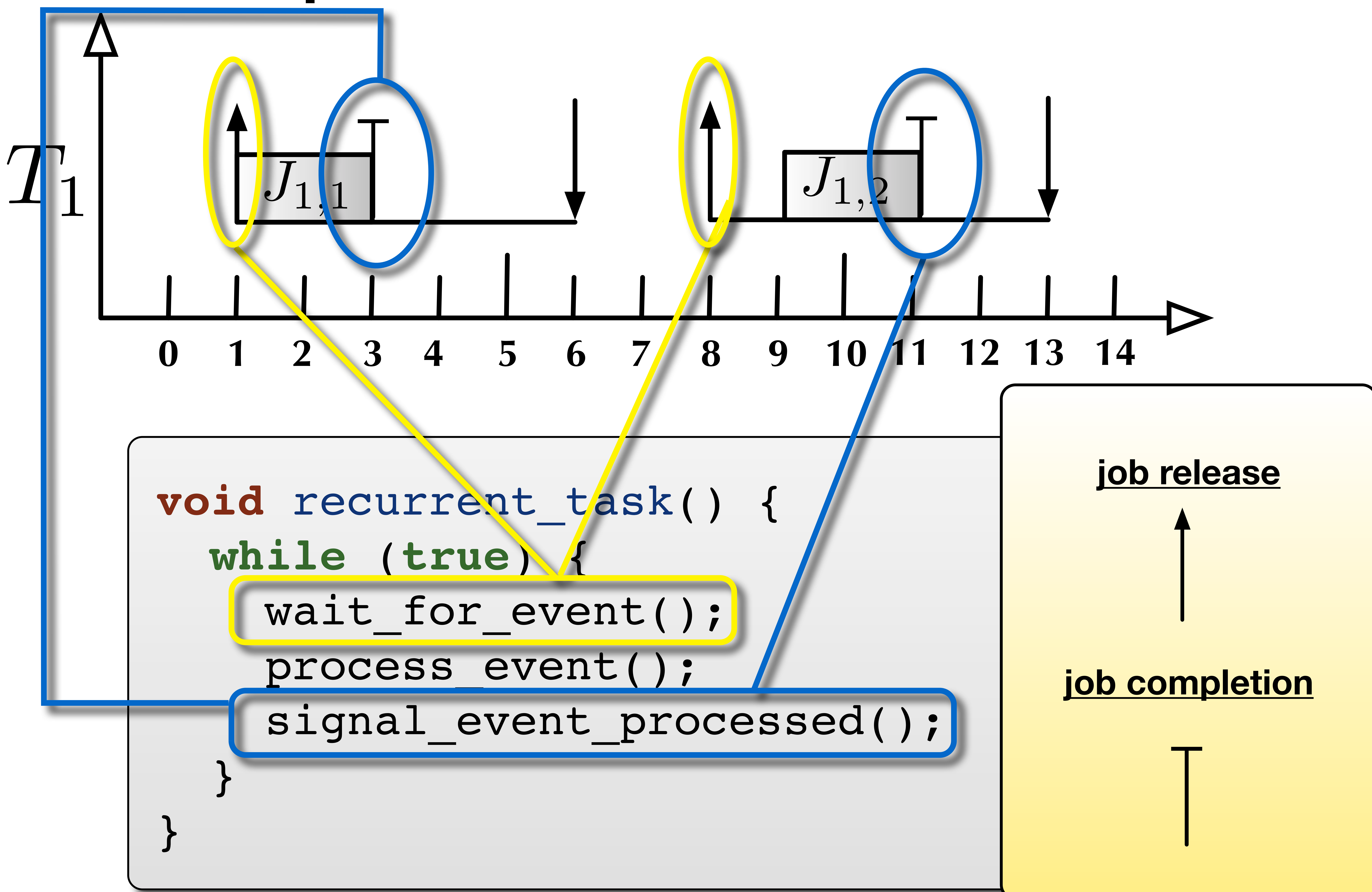



```
void recurrent_task() {
  while (true) {
    wait_for_event();
    process_event();
    signal_event_processed();
  }
}
```

job release

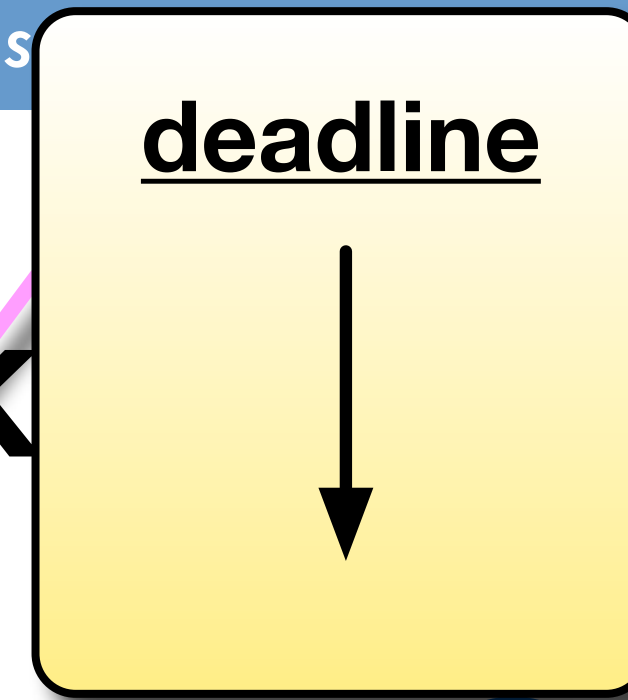
job completion

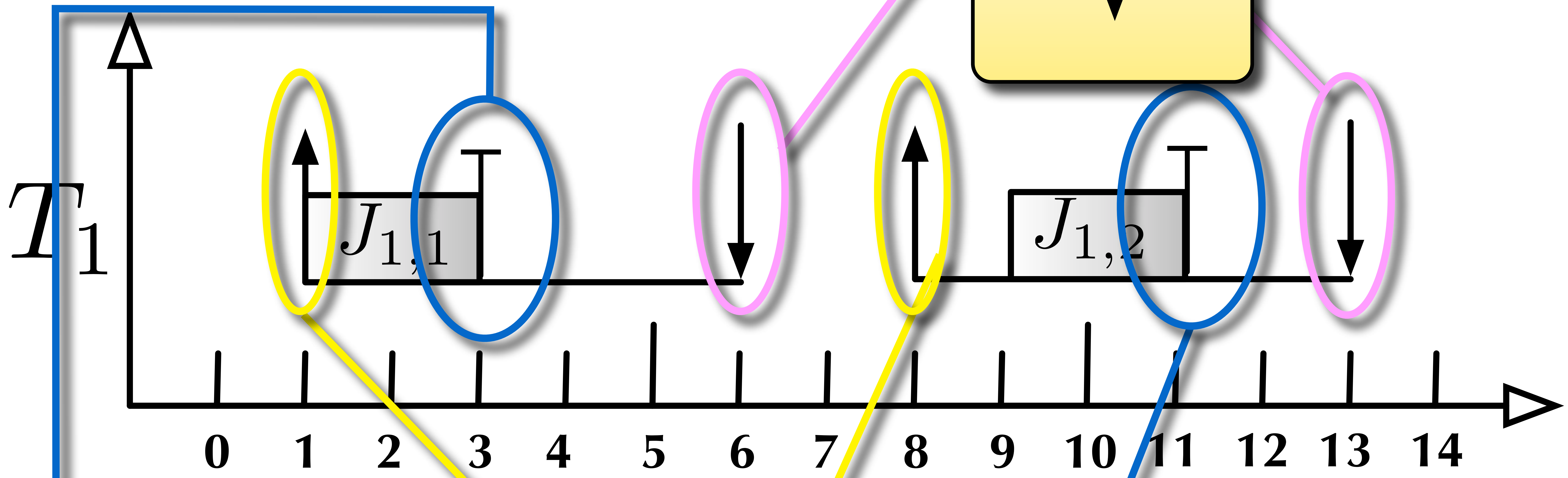
Sporadic Task Model



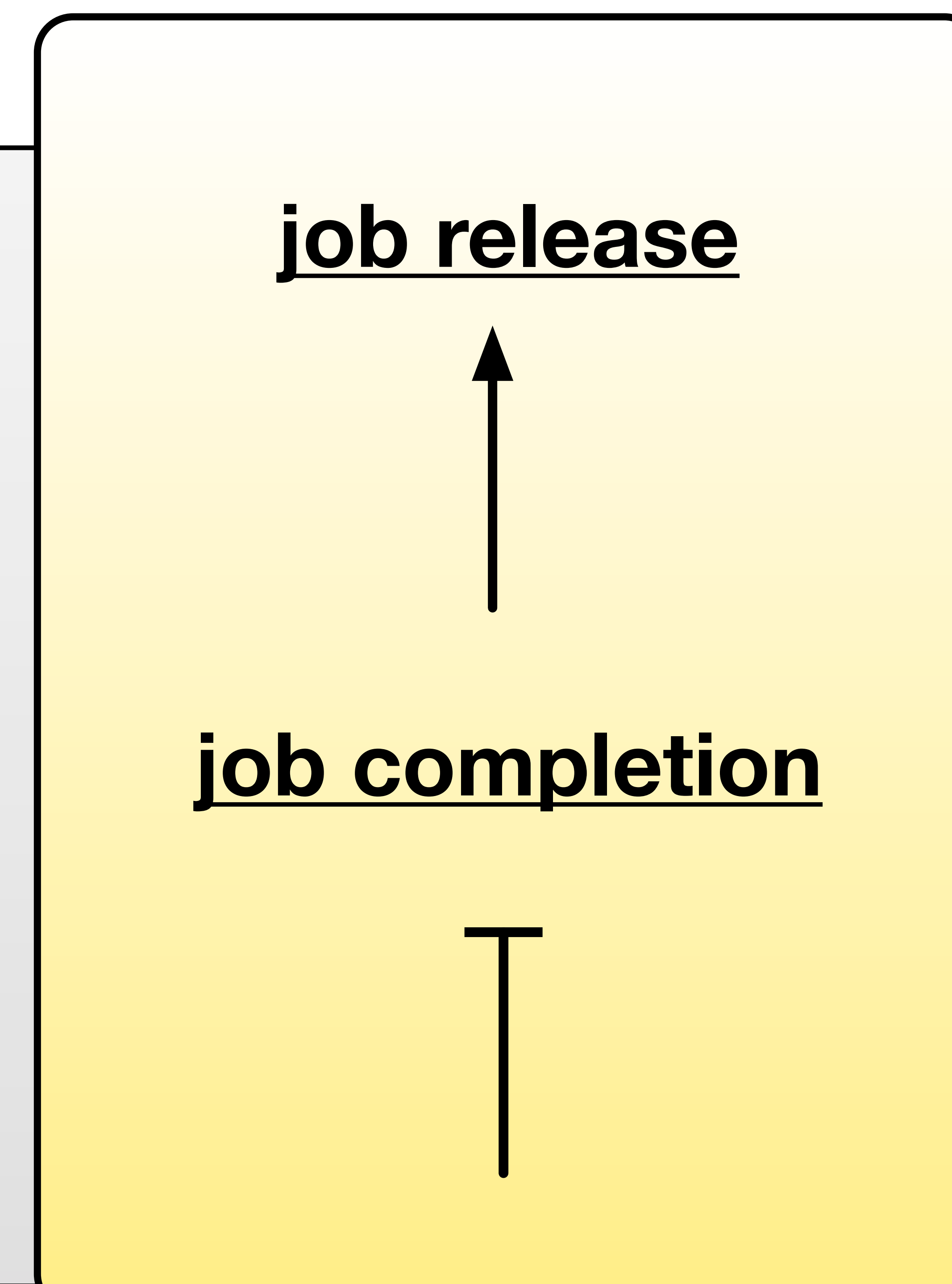

```
void recurrent_task() {
  while (true) {
    wait_for_event();
    process_event();
    signal_event_processed();
  }
}
```

Sporadic Task el





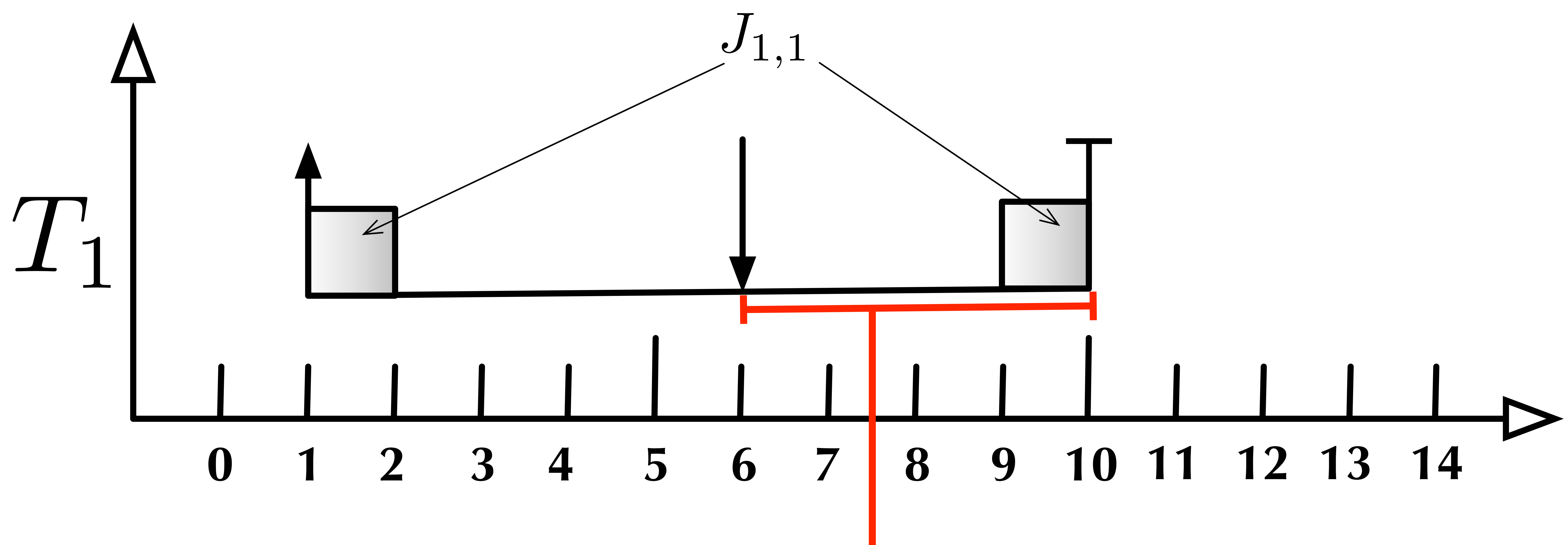

```
void recurrent_task() {
  while (true) {
    wait_for_event();
    process_event();
    signal_event_processed();
  }
}
```



Deadline Constraint

A job **should** complete by its deadline.
If it does not, it is **tardy**.

Implicit: next job does not arrive before deadline.



Tardiness: extent of deadline miss

Hard vs. Soft Real-Time

Hard Real-Time (HRT)

Each job **meets its deadline** (= **zero** tardiness).

Soft Real-Time (SRT)

Maximum deadline tardiness is **bounded** by a (reasonably small) **constant**.

Hard vs. Soft Real-Time

Hard Real-Time (HRT)

Each job **meets its deadline** (= **zero** tardiness).

If computation is "**bufferable,**" deadline miss may be masked with **finite buffer** (e.g., video decoding).

Soft Real-Time (SRT)

Maximum deadline tardiness is **bounded** by a (reasonably small) **constant**.

Processor Requirement

Task Utilization

fraction of **processor capacity** required by task

Total Utilization

Sum of all task utilizations:
min. processor capacity required by **task set**.

Schedulers for Sporadic Tasks

Task schedulable:

Task can be shown *a priori* to
always satisfy its temporal constraint
under a given scheduler
(w.r.t. HRT or SRT interpretation).

In this talk:
5 selected schedulers.

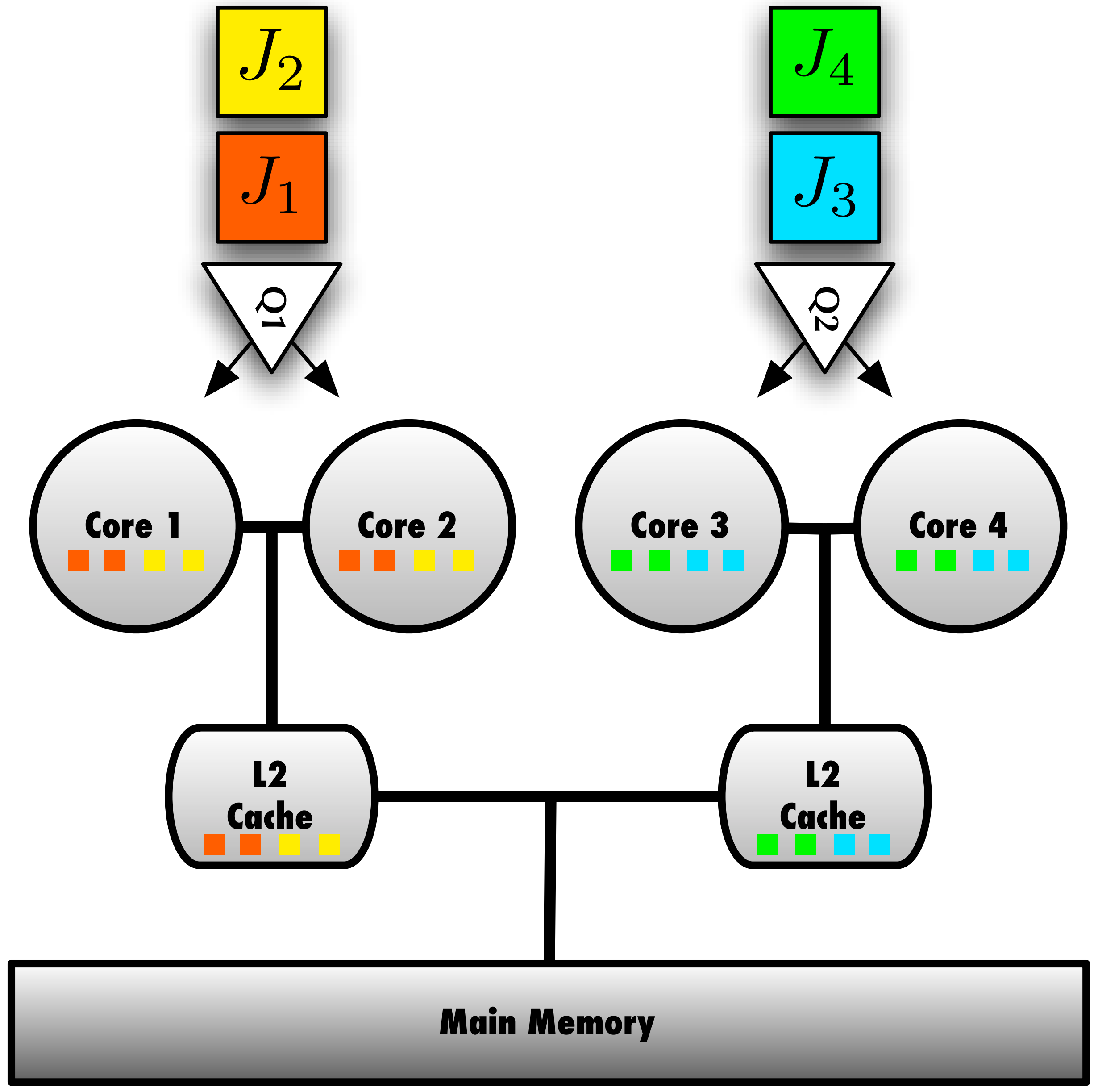
In my dissertation:
22 schedulers.

Clustered Multiprocessor Scheduling

(1) Group cores into **clusters**.

(2) **Statically** assign tasks to clusters before runtime.

(3) Schedule each cluster **individually** from a **per-cluster job queue**.



clustered scheduling

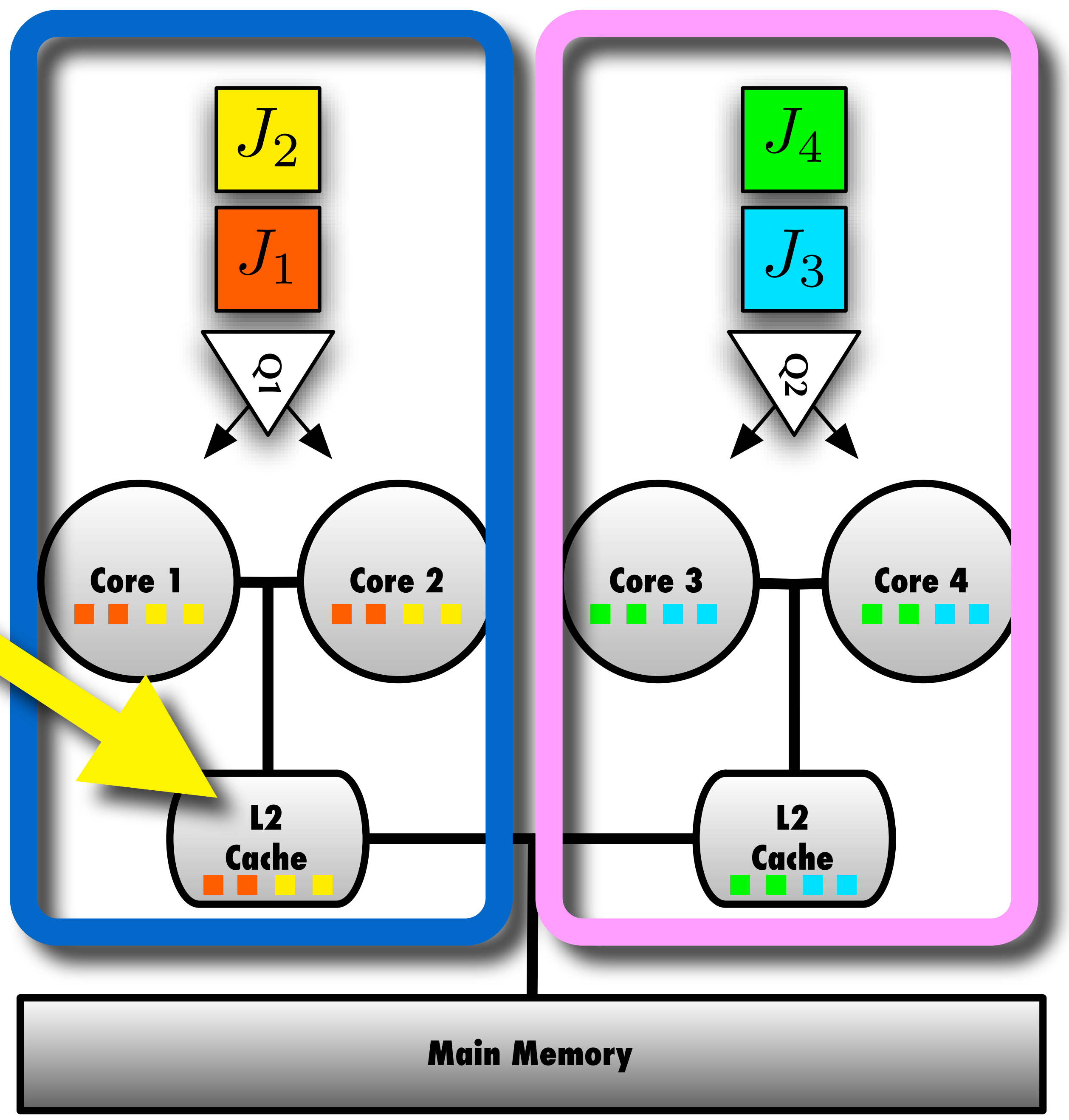
Clustered Multiprocessor Scheduling

(1) Group cores into **clusters**.

(2) **Statically** assign tasks to clusters before runtime.

(3) Schedule each cluster **individually** from a **per-cluster job queue**.

Example: cores that share a cache form a cluster.



clustered scheduling

Clustered Multiprocessor Scheduling

(1) Group cores into **clusters**.

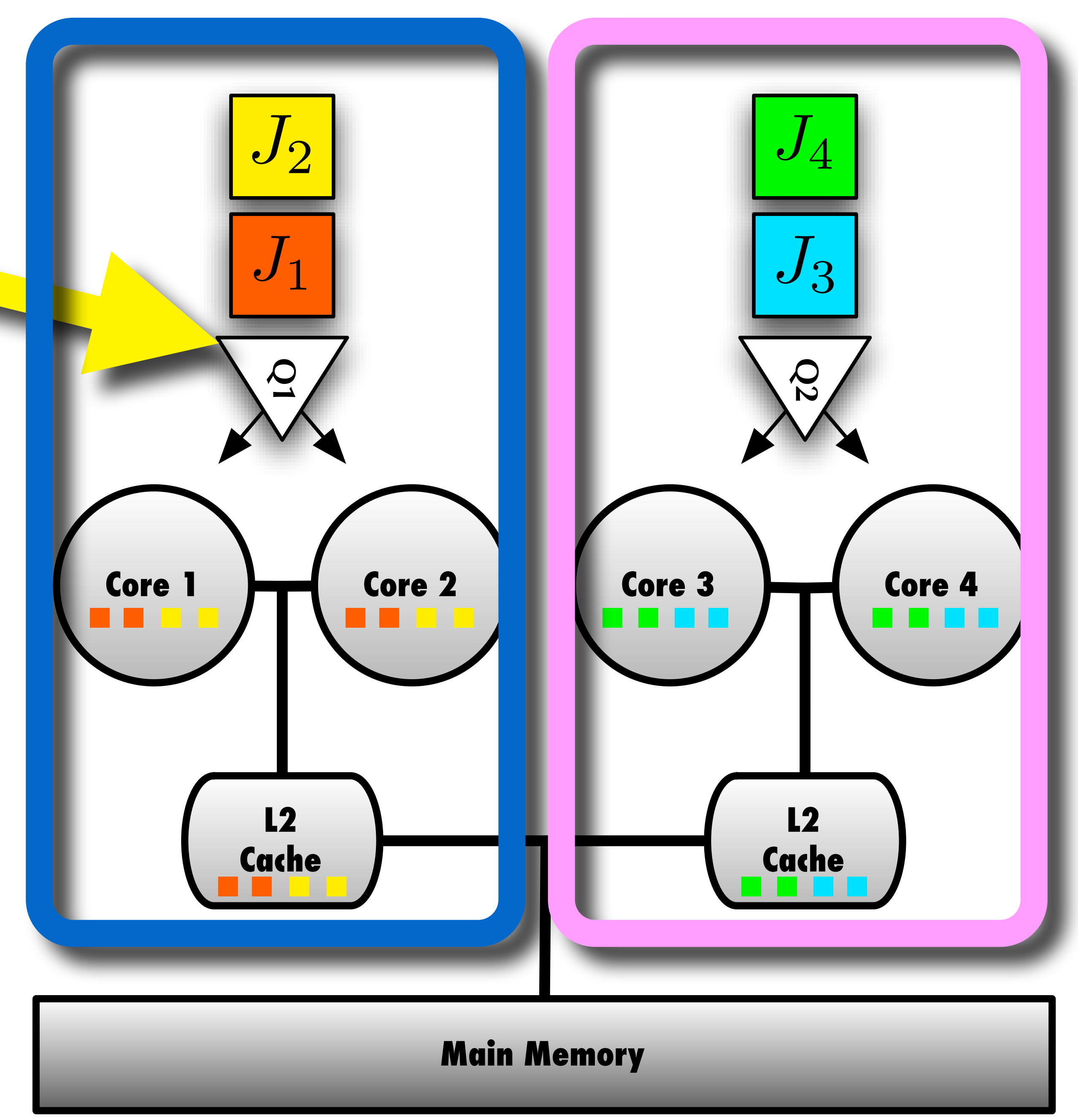
(2) **Statically** assign tasks to clusters before runtime.

(3) Schedule each cluster **individually** from a **per-cluster job queue**.

Offline: assign tasks to clusters.

Online: schedule jobs preemptively from a priority queue.

Jobs may migrate, but only within cluster.



clustered scheduling

Clustered Multipr

Job Priority Order

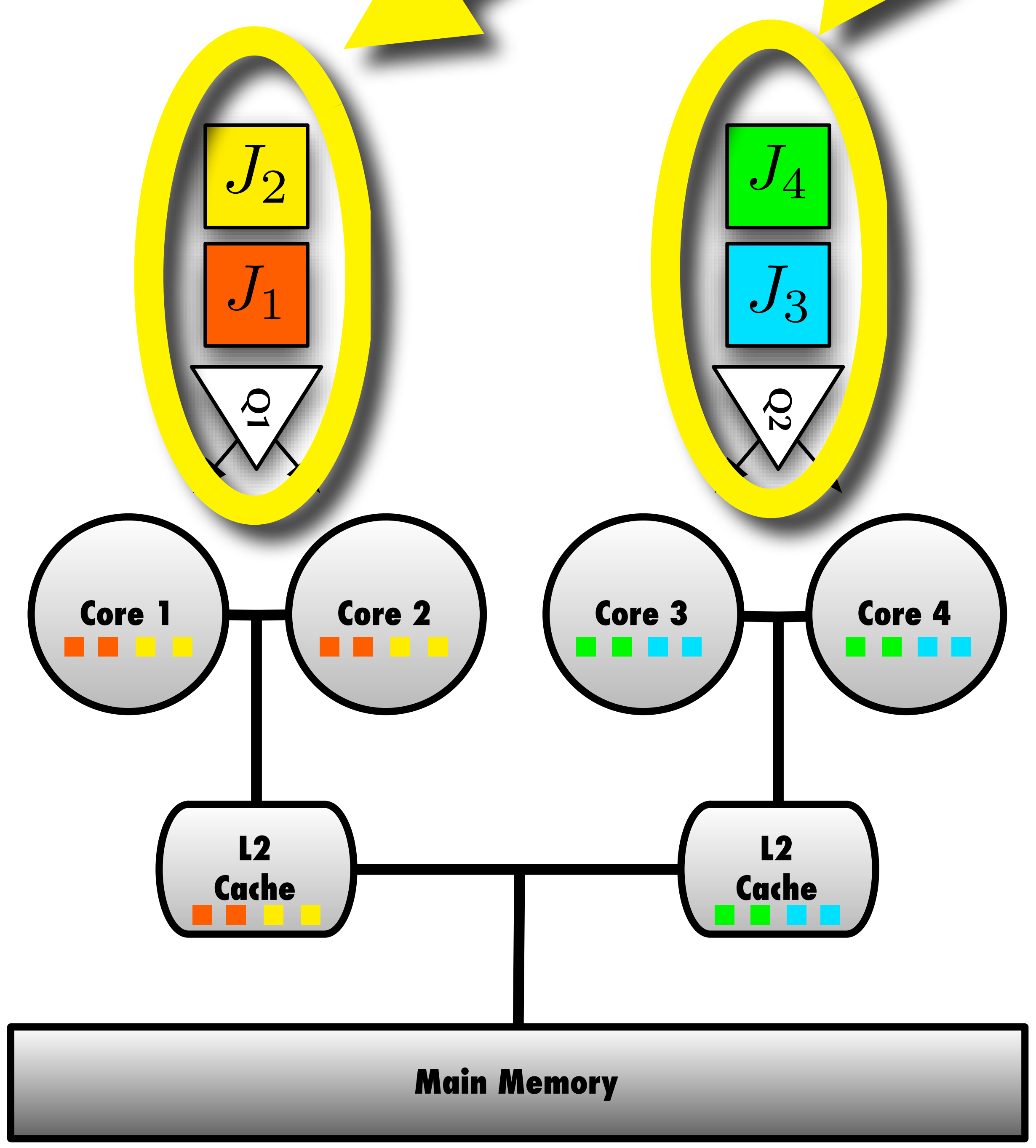
Earliest-Deadline First (EDF)
(order by increasing deadline)

Fixed-Priority (FP)
(manually assign priorities to tasks)

(1) Group cores into **clusters**.

(2) **Statically** assign tasks to clusters before runtime.

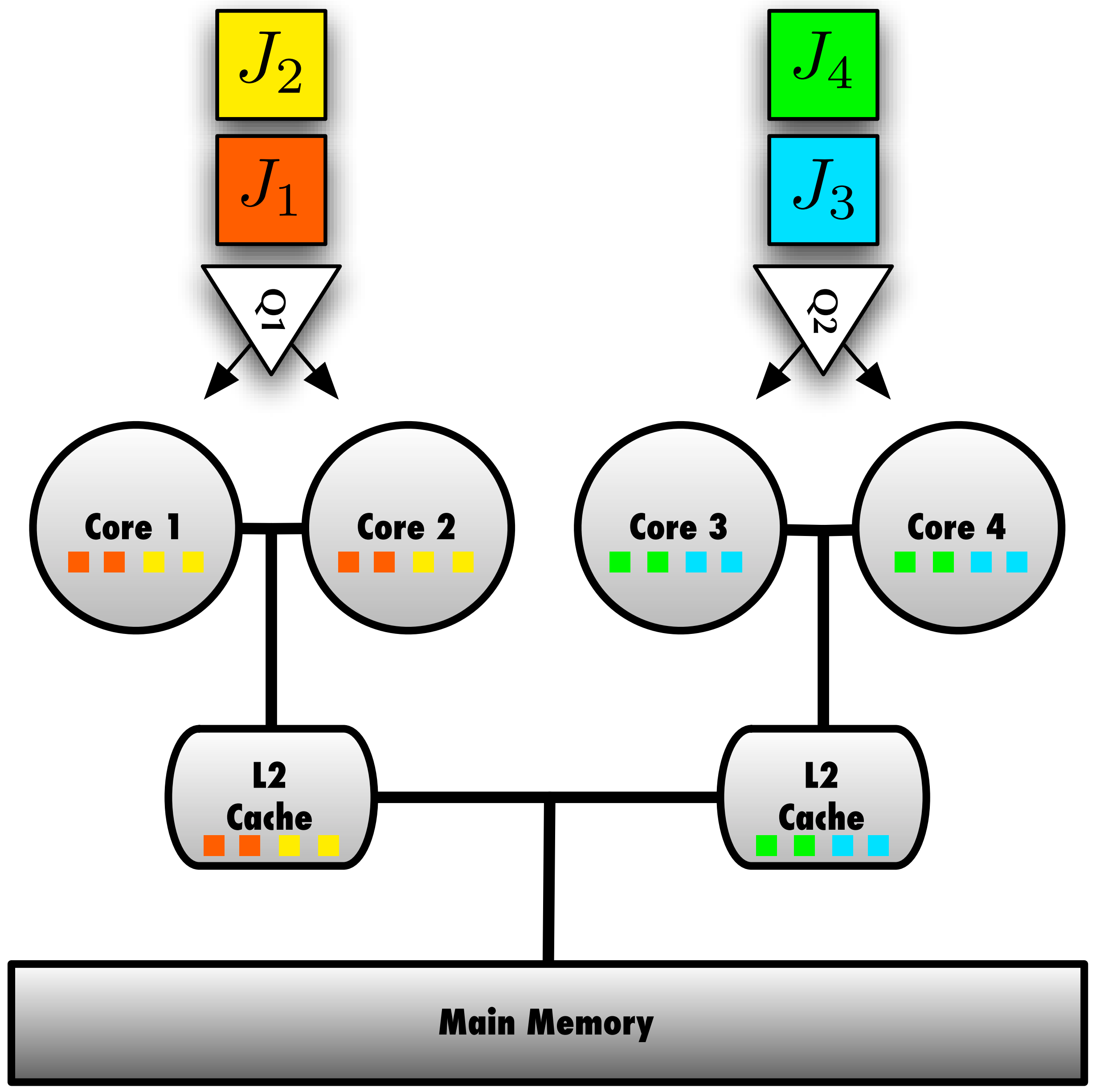
(3) Schedule each cluster **individually** from a **per-cluster job queue**.



clustered scheduling

Clustered Multiprocessor Scheduling

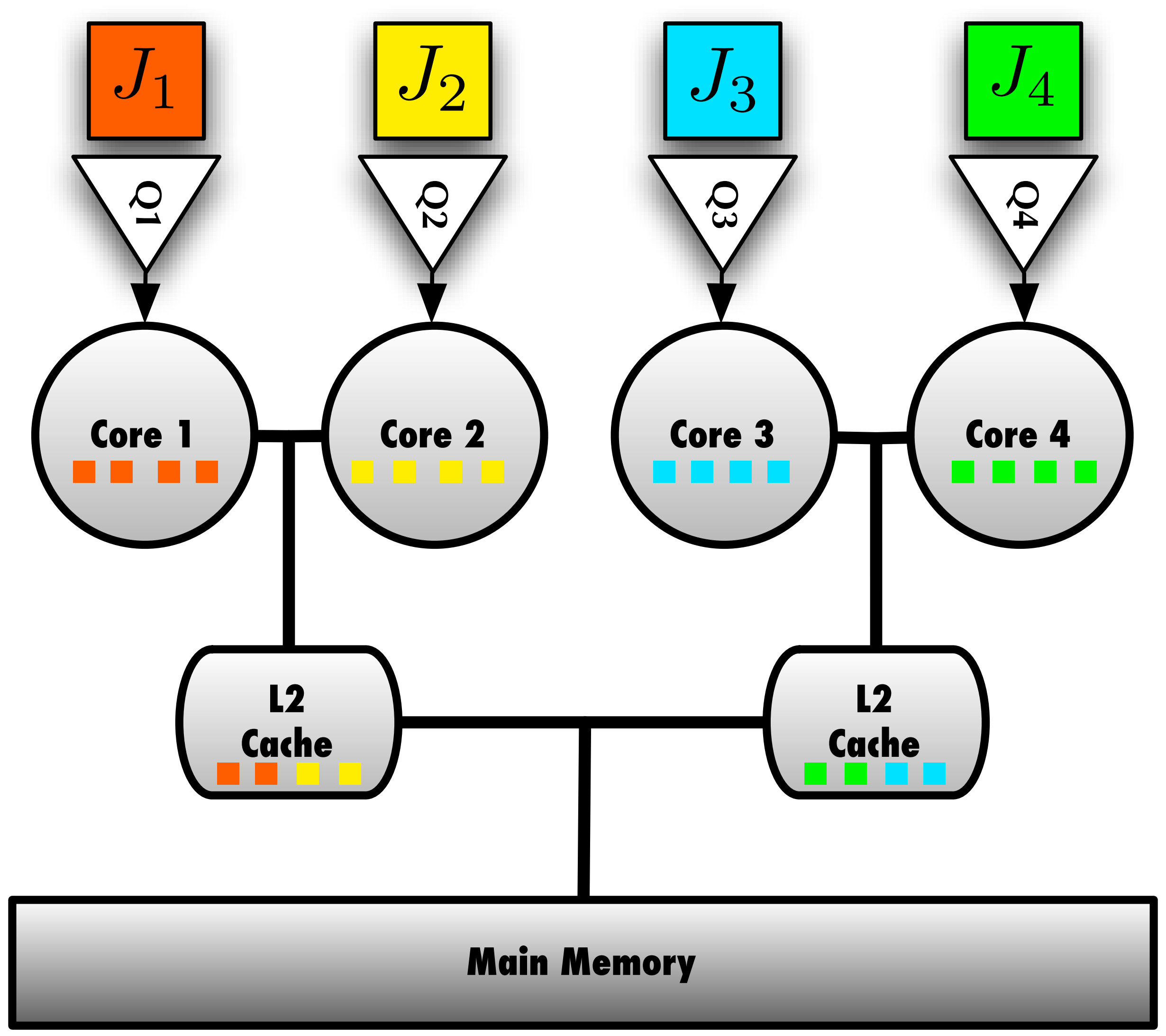
Two common special cases:
one-core clusters and a **single cluster**



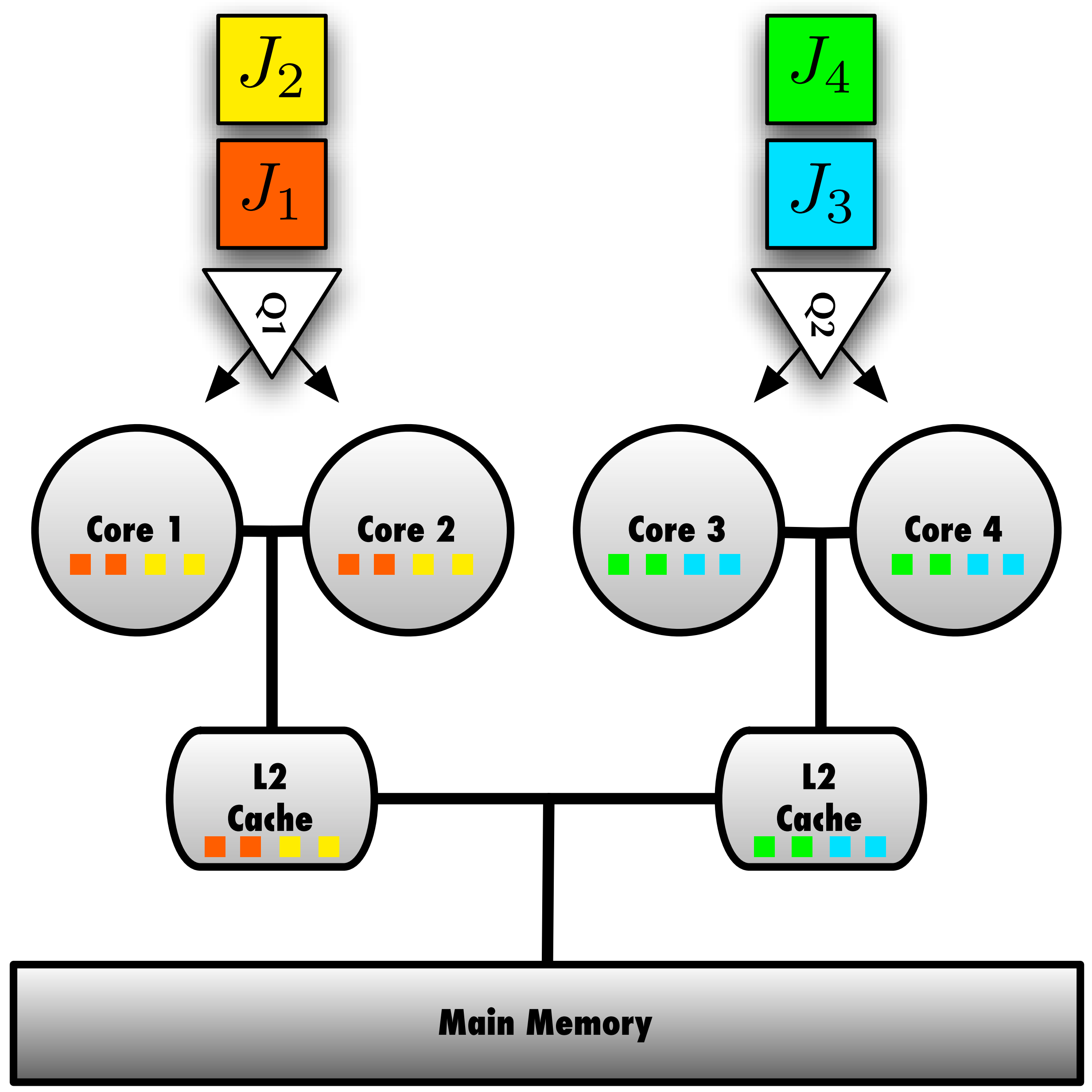
clustered scheduling

Clustered Multiprocessor Scheduling

Two common special cases:
one-core clusters and a **single cluster**



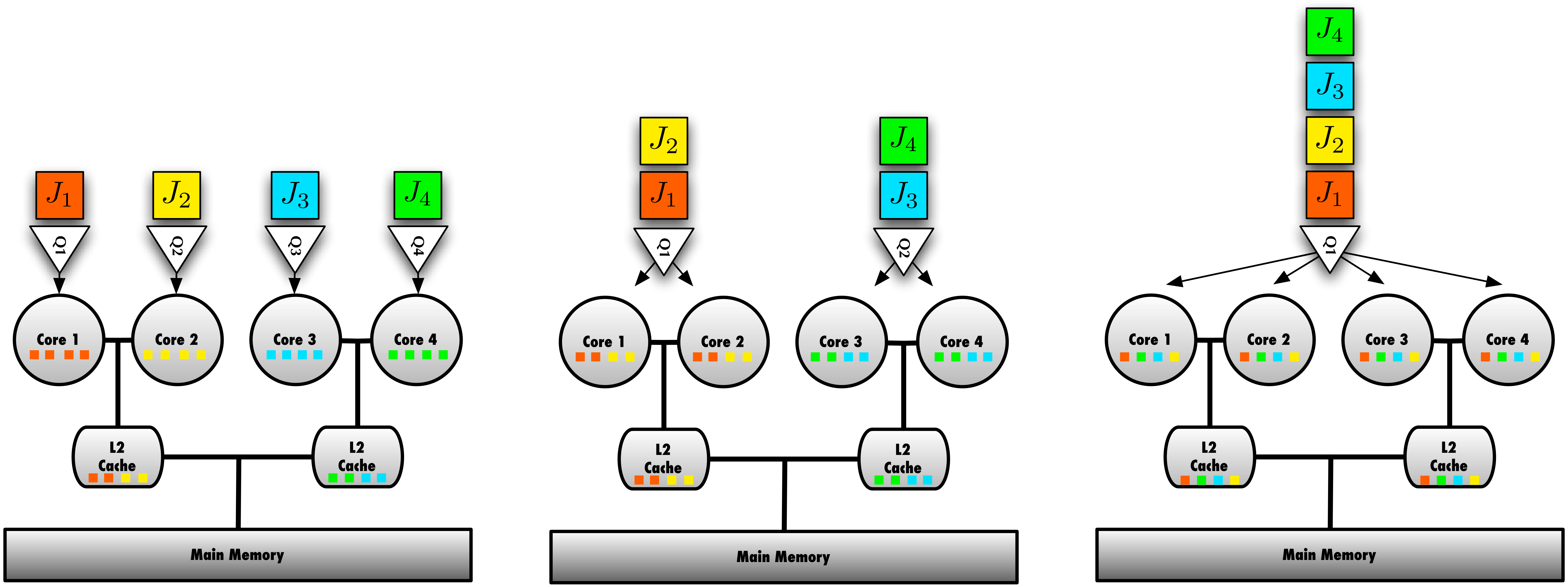
partitioned scheduling



clustered scheduling

Clustered Multiprocessor Scheduling

Two common special cases:
one-core clusters and a **single cluster**



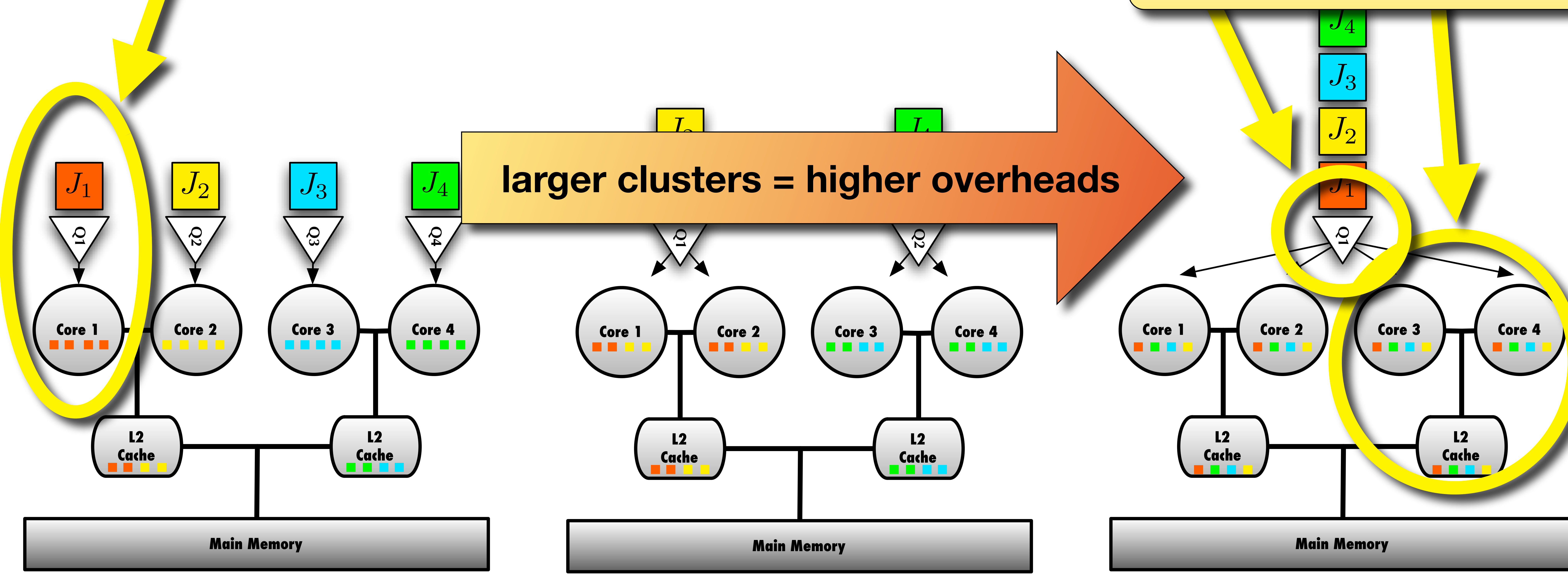
partitioned scheduling — clustered scheduling — global scheduling

Clustered Multiprocessor Scheduling

Good cache affinity, low contention.

Weak cache affinity, high contention.

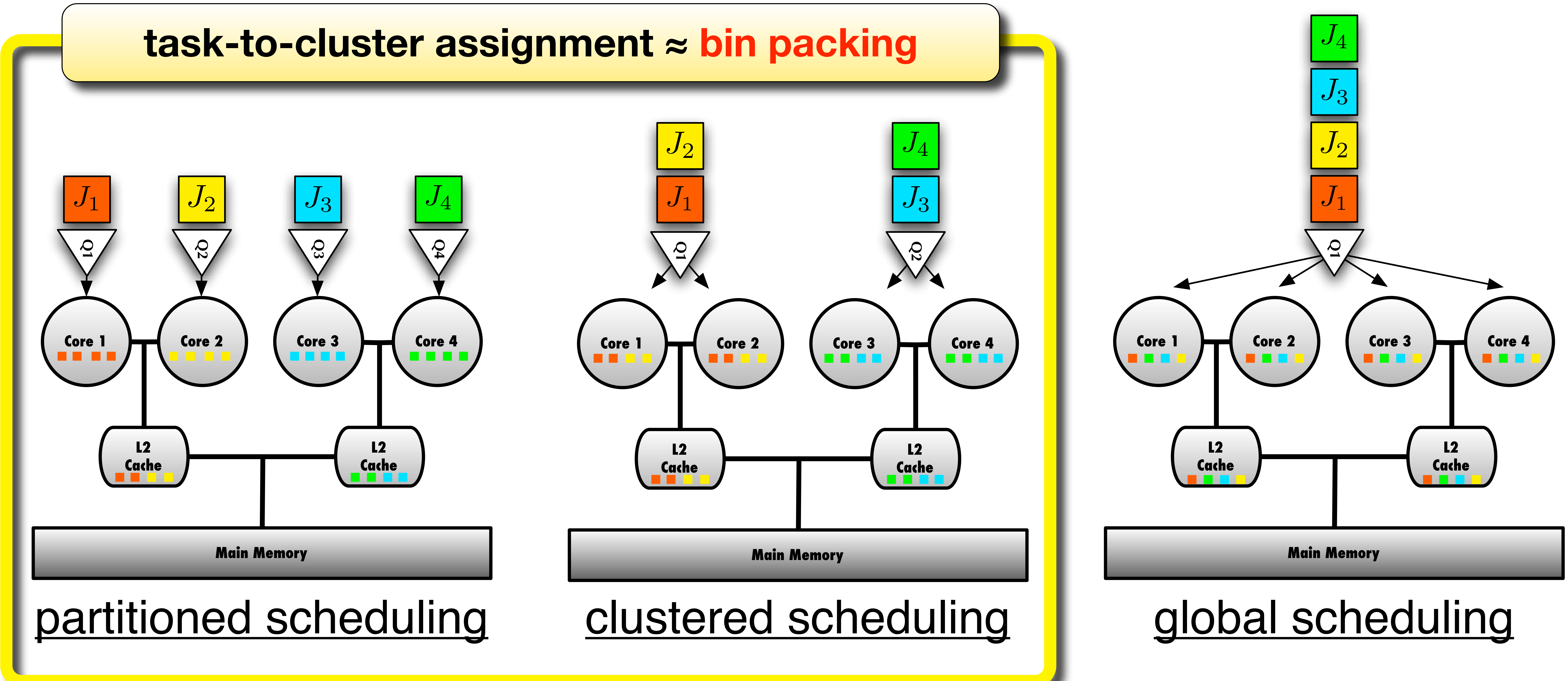
o common special cases
e **clusters** and a **single**



partitioned scheduling clustered scheduling global scheduling

Clustered Multiprocessor Scheduling

Two common special cases:
one-core clusters and a **single cluster**

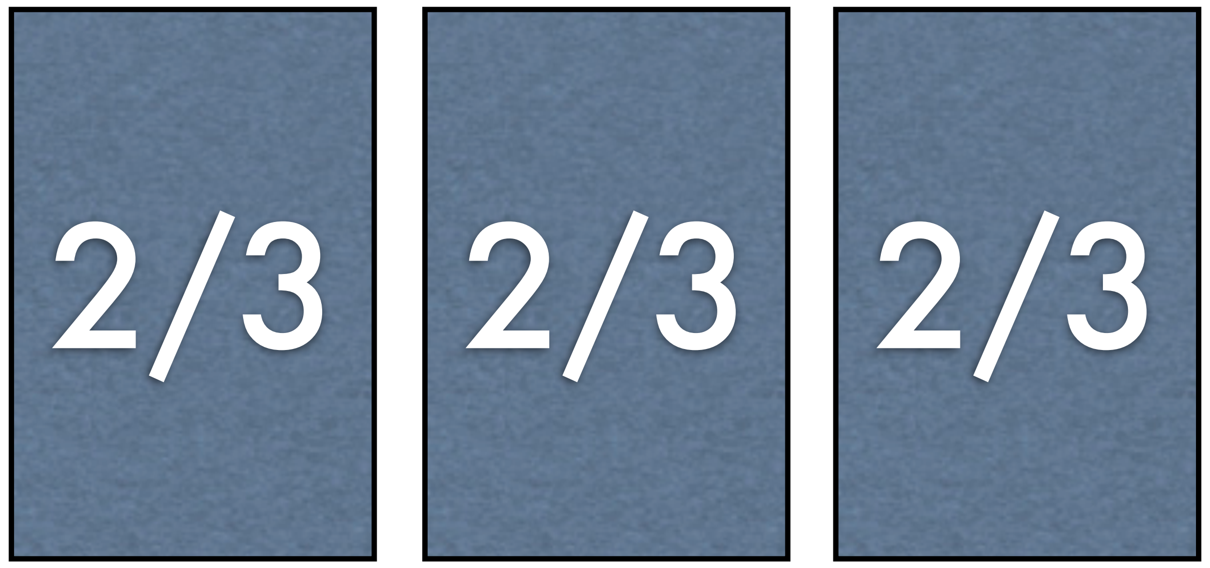


Bin Packing

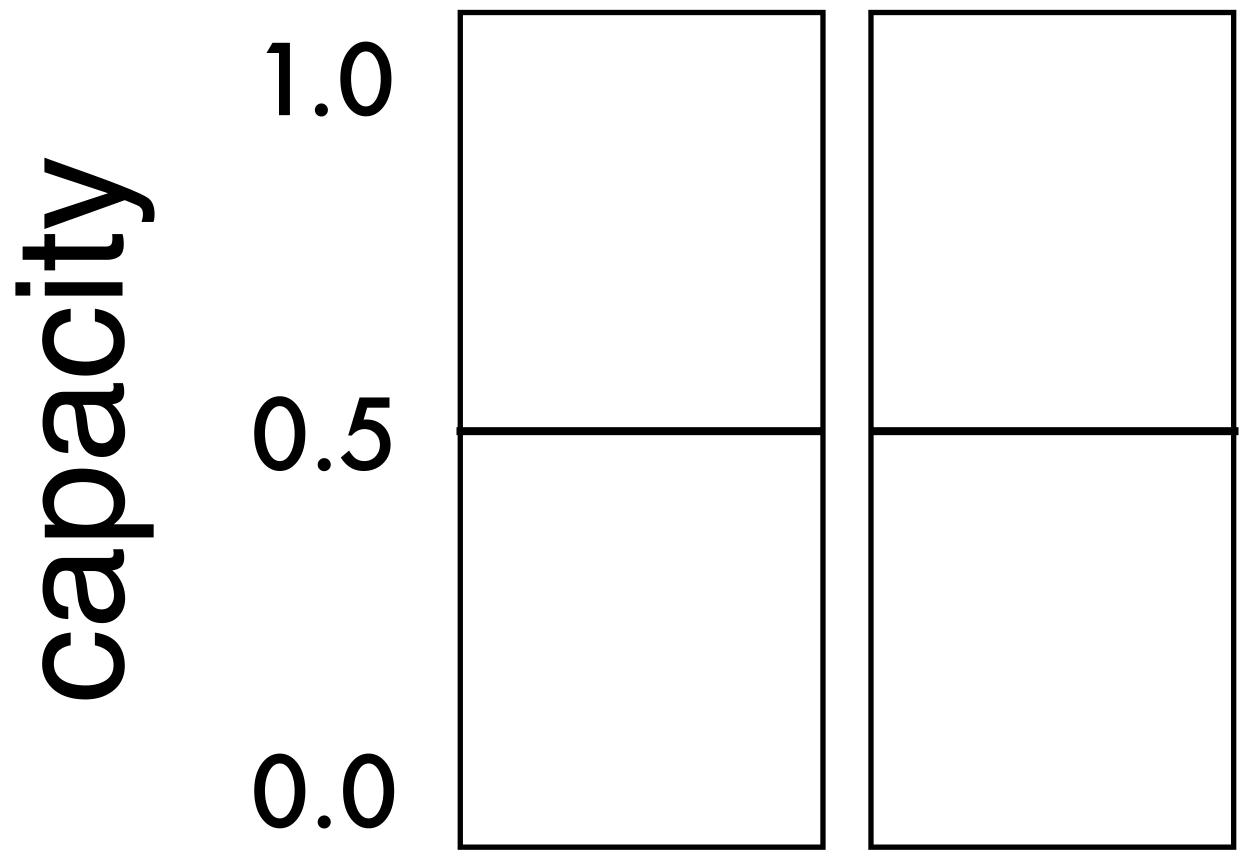
three identical tasks

task utilization = 2/3

total utilization = 2



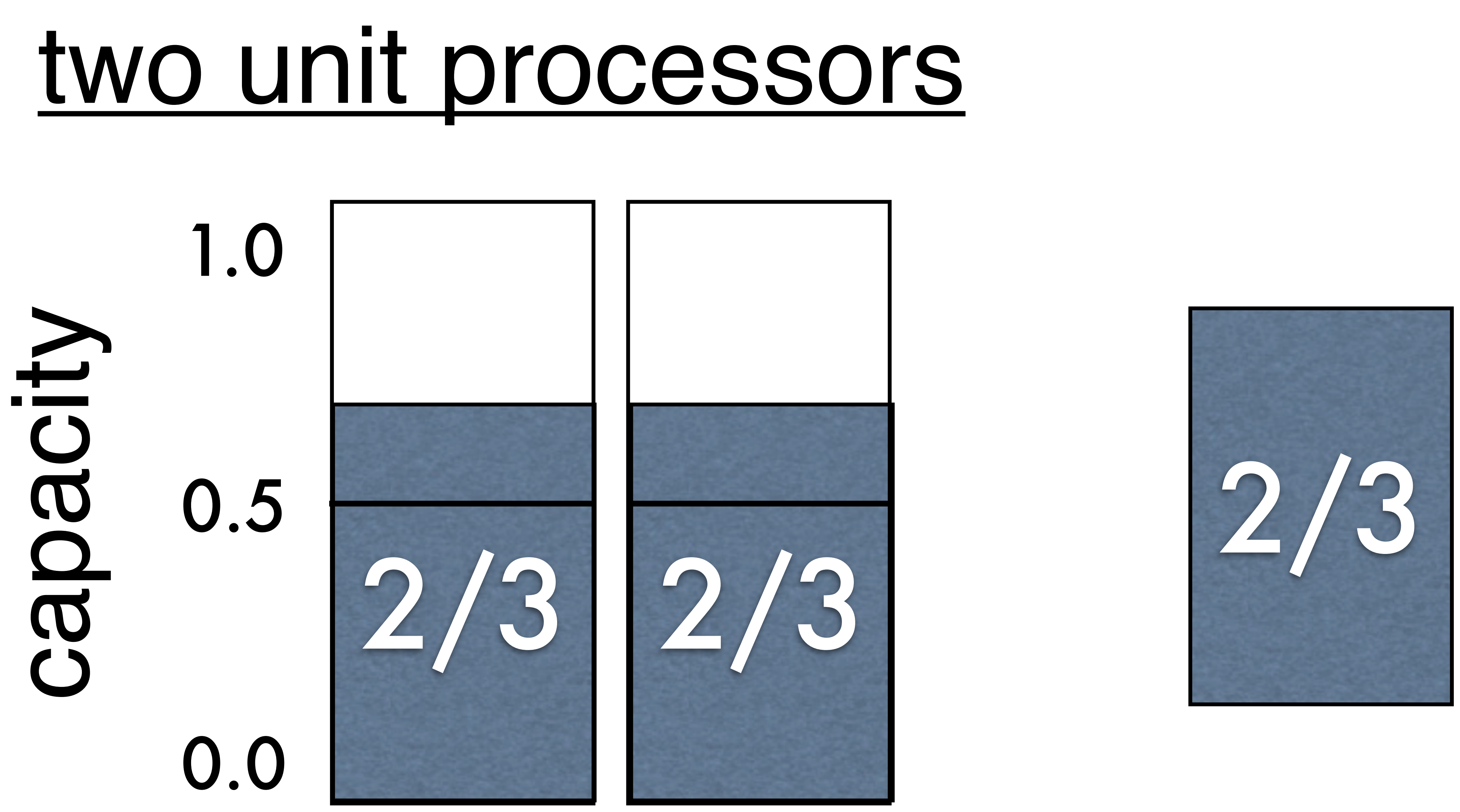
two unit processors



Bin Packing

Processor Overloading

Even though there is sufficient total capacity, the last task cannot be placed.

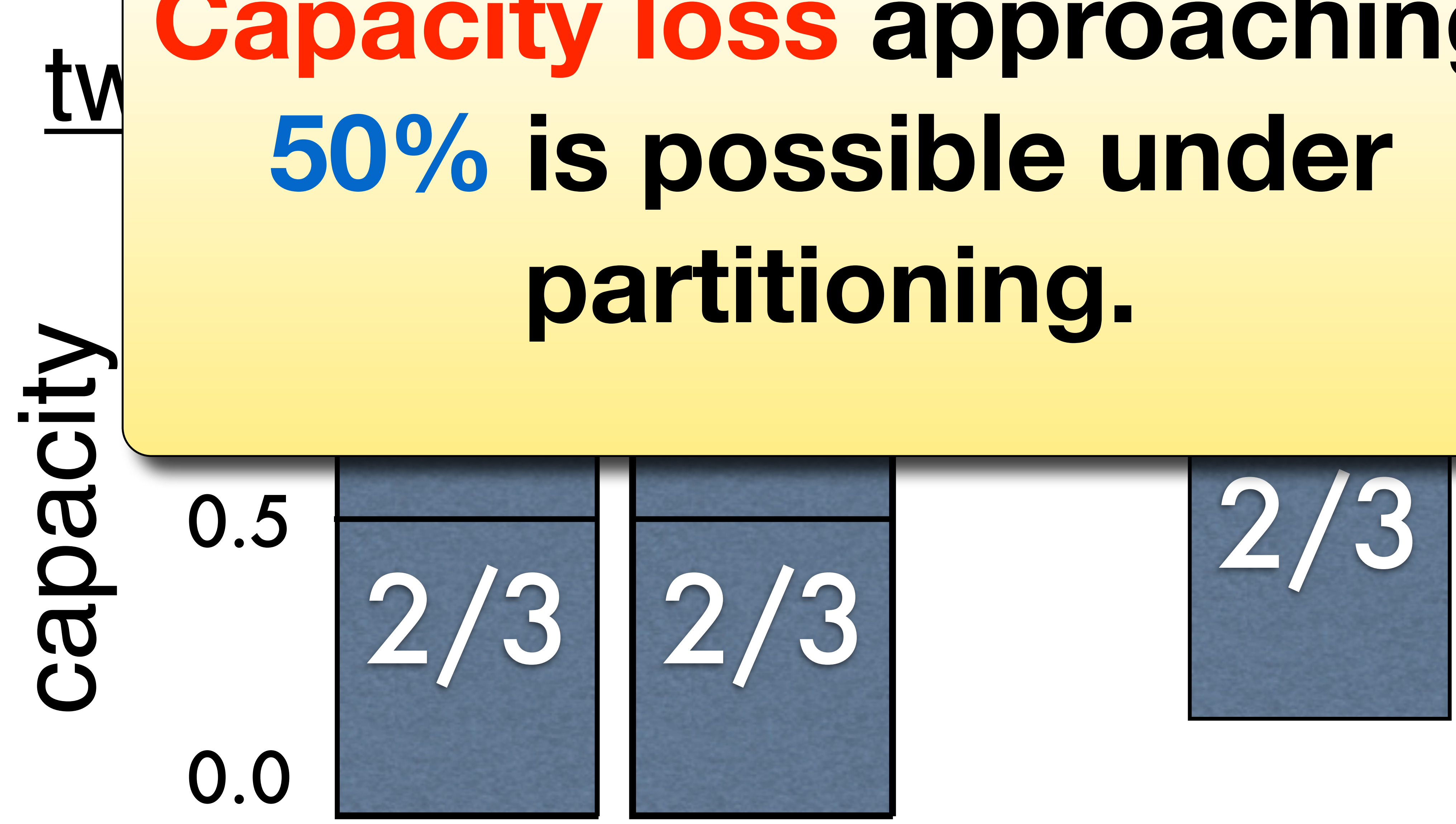


Bin Packing

<u>Processor Overloading</u>

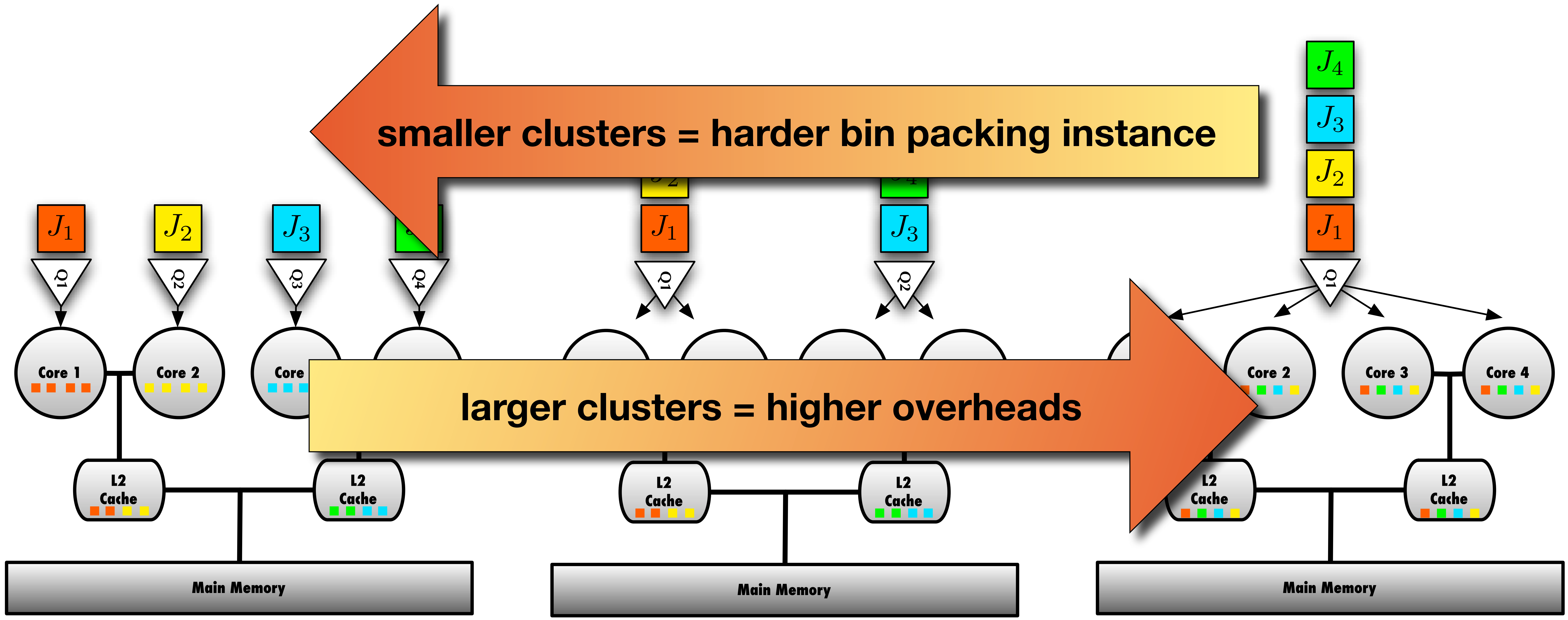
Even though there is sufficient total capacity, the last task cannot be placed.

Capacity loss approaching 50% is possible under partitioning.



Clustered Multiprocessor Scheduling

Two common special cases:
one-core clusters and a **single cluster**

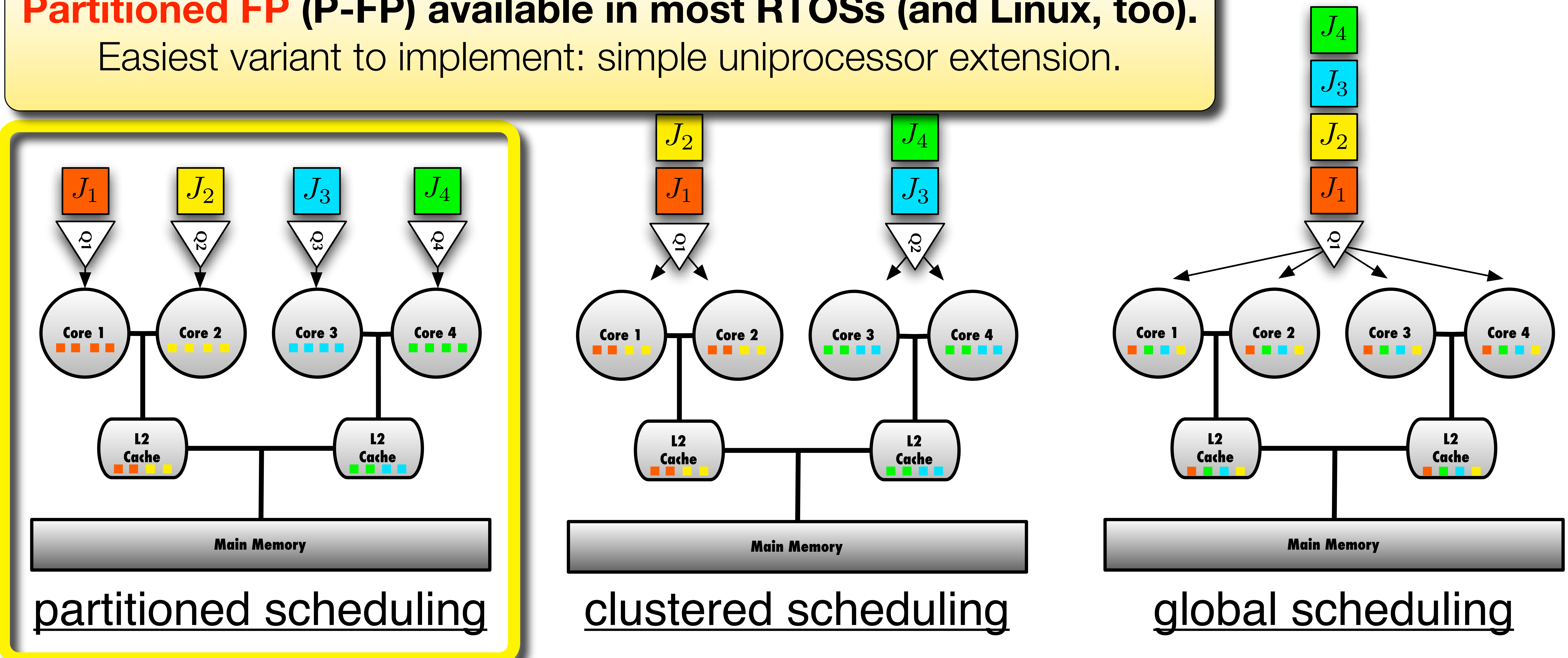


partitioned scheduling | clustered scheduling | global scheduling

Clustered Multiprocessor Scheduling

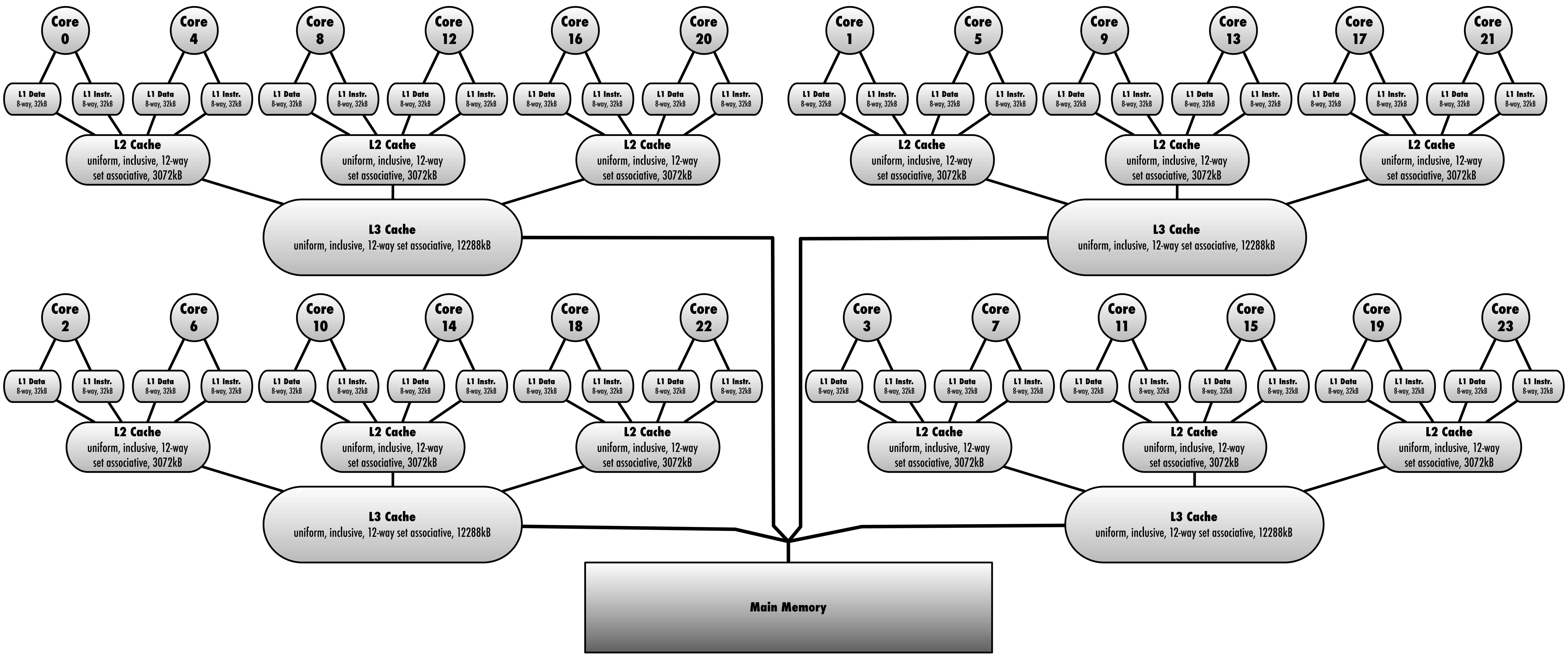
Two common special cases:
one-core clusters and a **single cluster**

Partitioned FP (P-FP) available in most RTOSs (and Linux, too).
Easiest variant to implement: simple uniprocessor extension.



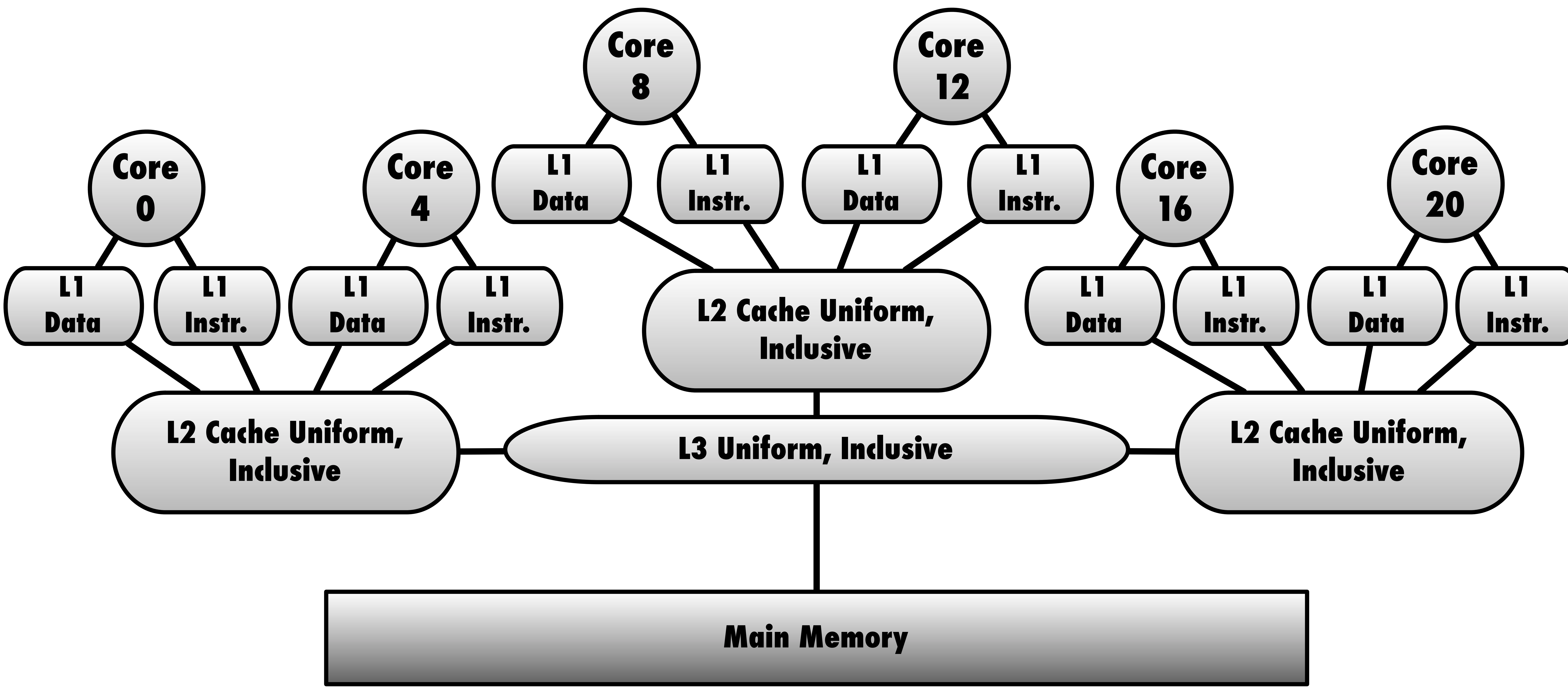
Xeon L7455 Hardware Topology

Six cores per socket; each clocked at 2.16 GHz.
Four sockets for a total of 24 cores.



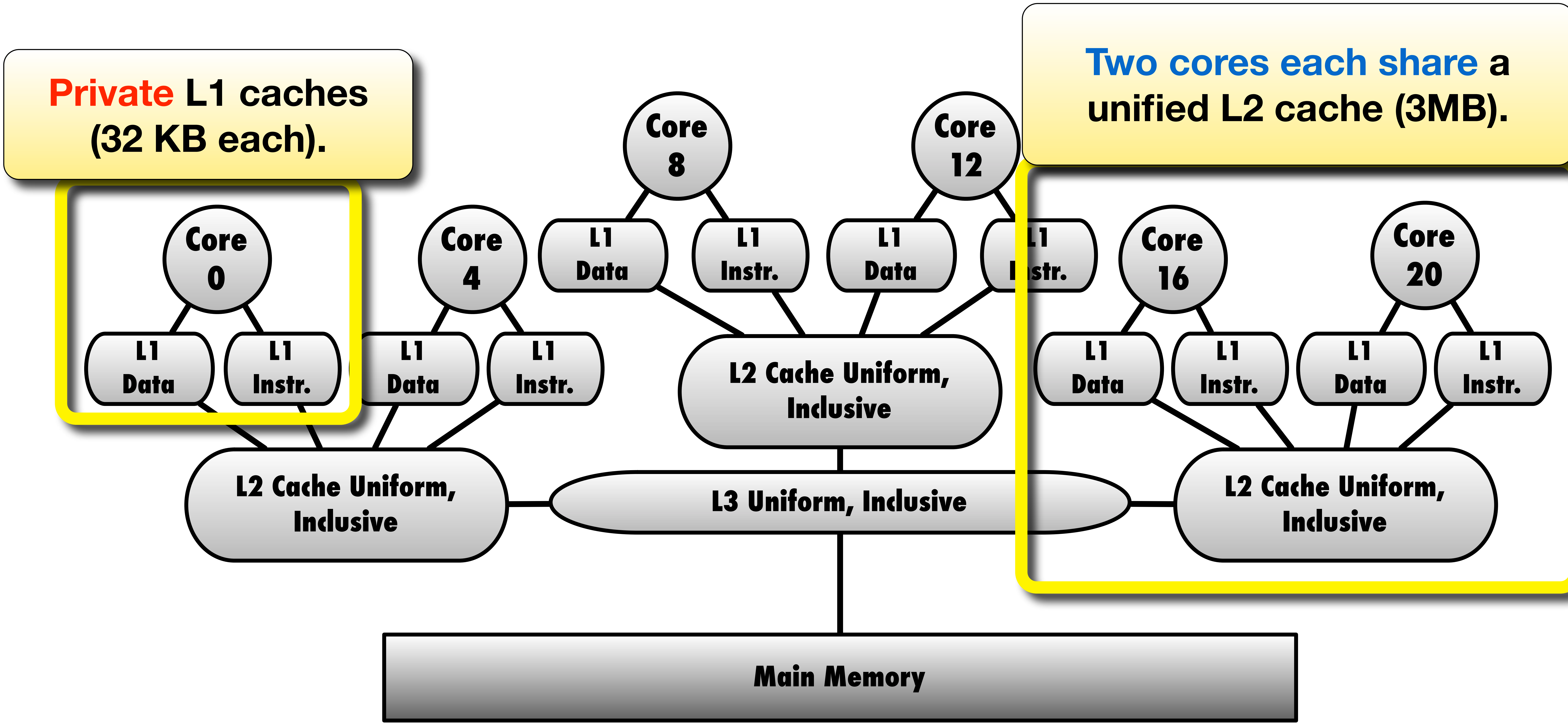
Hardware Topology — Single Socket

Six cores per socket; each clocked at 2.16 GHz.
Four sockets for a total of 24 cores.



Hardware Topology — Single Socket

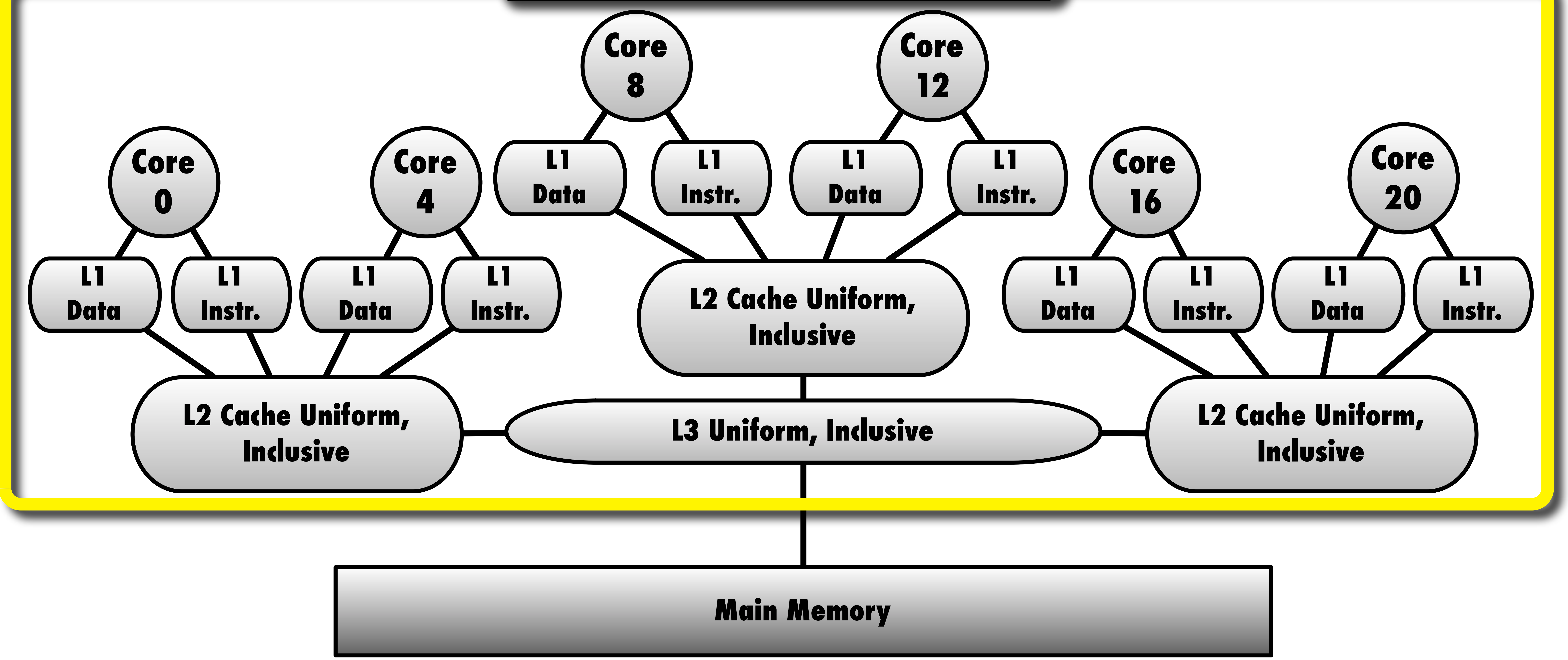
Six cores per socket; each clocked at 2.16 GHz.
Four sockets for a total of 24 cores.



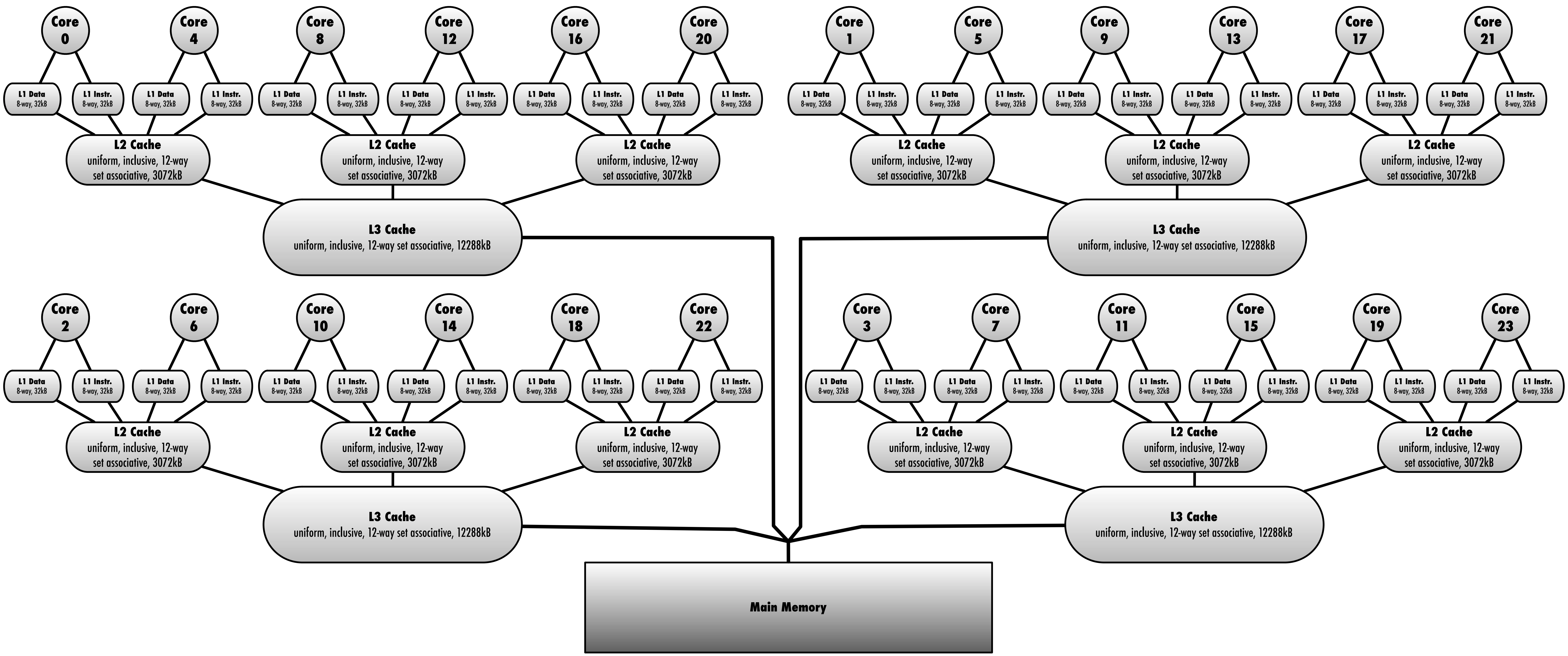
Hardware Topology — Single Socket

Six cores per socket; each clocked at 2.16 GHz.
Four sc[...] cores.

All six cores share a unified L3 cache (12 MB).

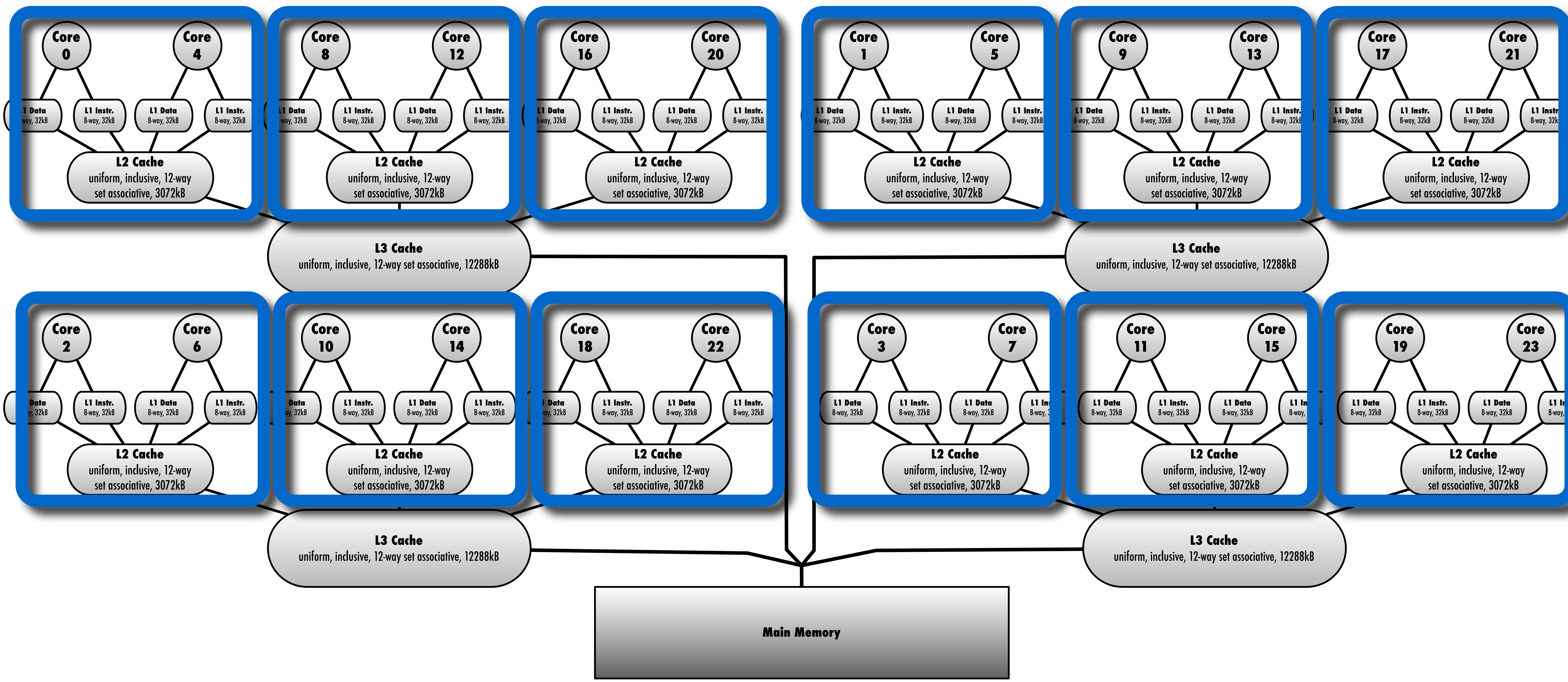


Clustered Scheduling Options



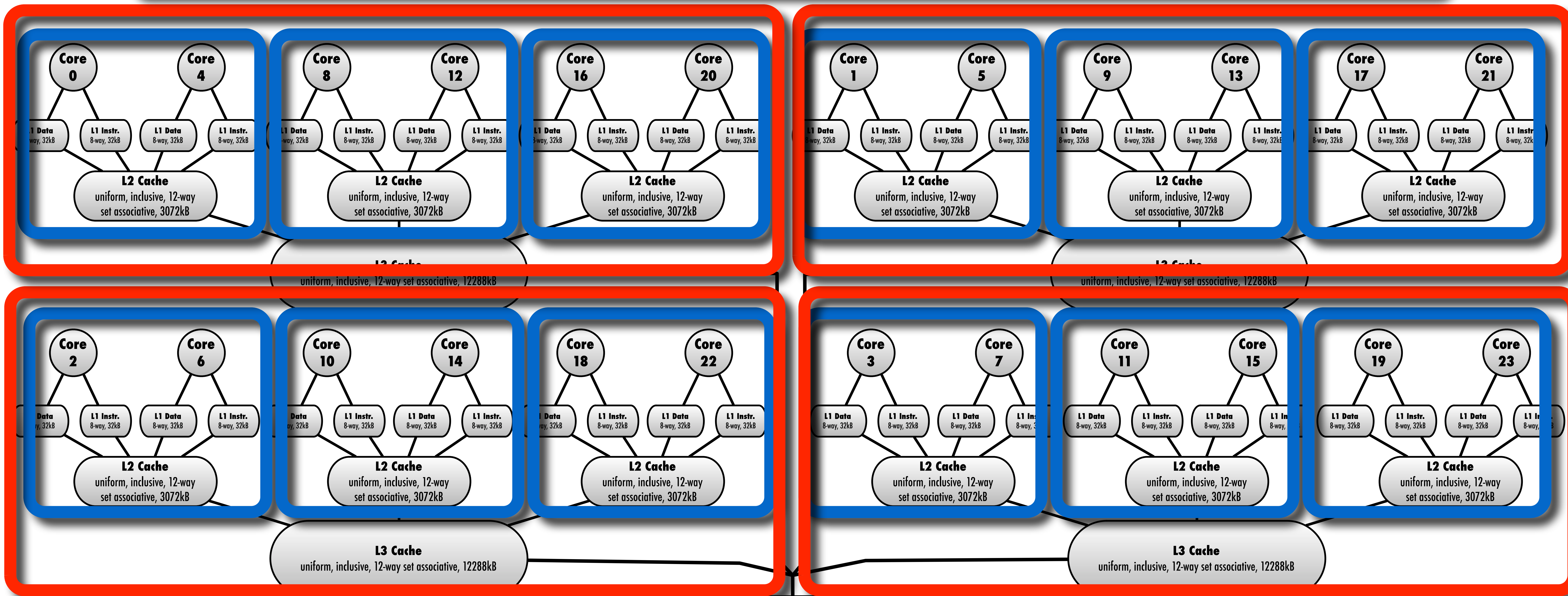
Clustered Scheduling Options

Either 12 L2-based clusters of two cores each…



Clustered Scheduling Options

Either 12 L2-based clusters of two cores each...



...or four L3-based clusters of six cores each.

Five Evaluated Schedulers

(*dissertation: study with 22 scheduler configurations*)

	Global	Clustered	Partitioned
FP (baseline)			P-FP
EDF	G-EDF	C-EDF-L2 C-EDF-L3	P-EDF

Five Evaluated Schedulers

(*dissertation: study with 22 scheduler configurations*)

	Global	Clustered	Partitioned
FP (baseline)			P-FP
EDF	G-EDF	C-EDF-L2 C-EDF-L3	P-EDF

smaller clusters = harder bin packing instance

larger clusters = higher overheads

Five Evaluated Schedulers

(*dissertation: study with 22 scheduler configurations*)

What dominates capacity loss:

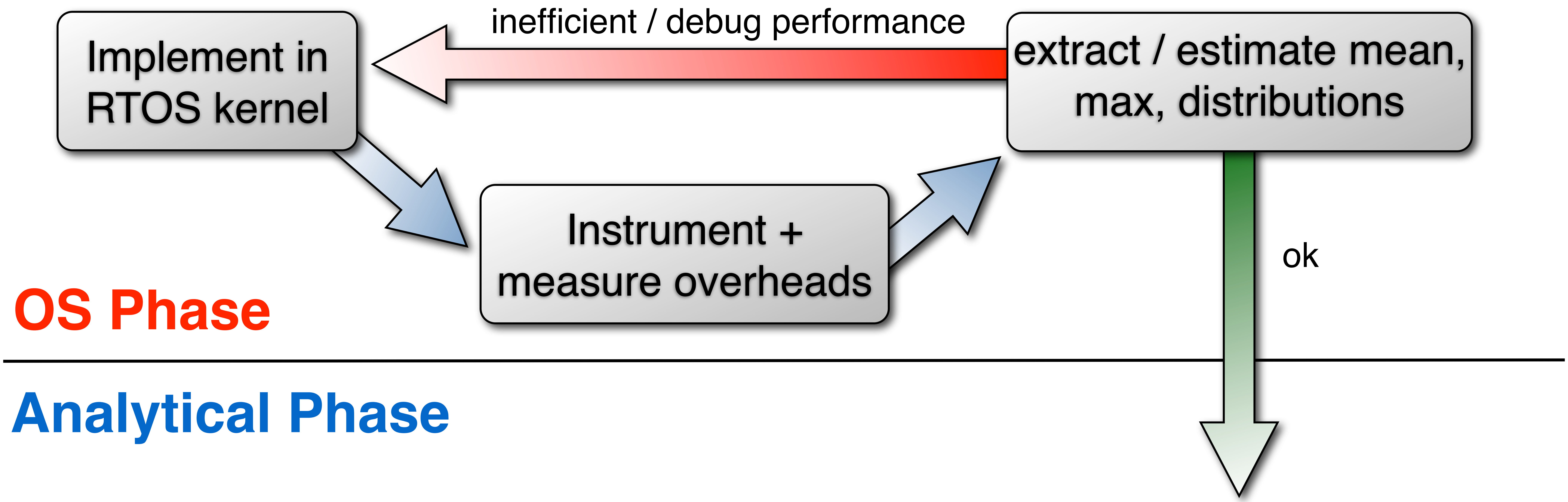
Algorithmic or overhead issues?

Scheduler Evaluation Methodology

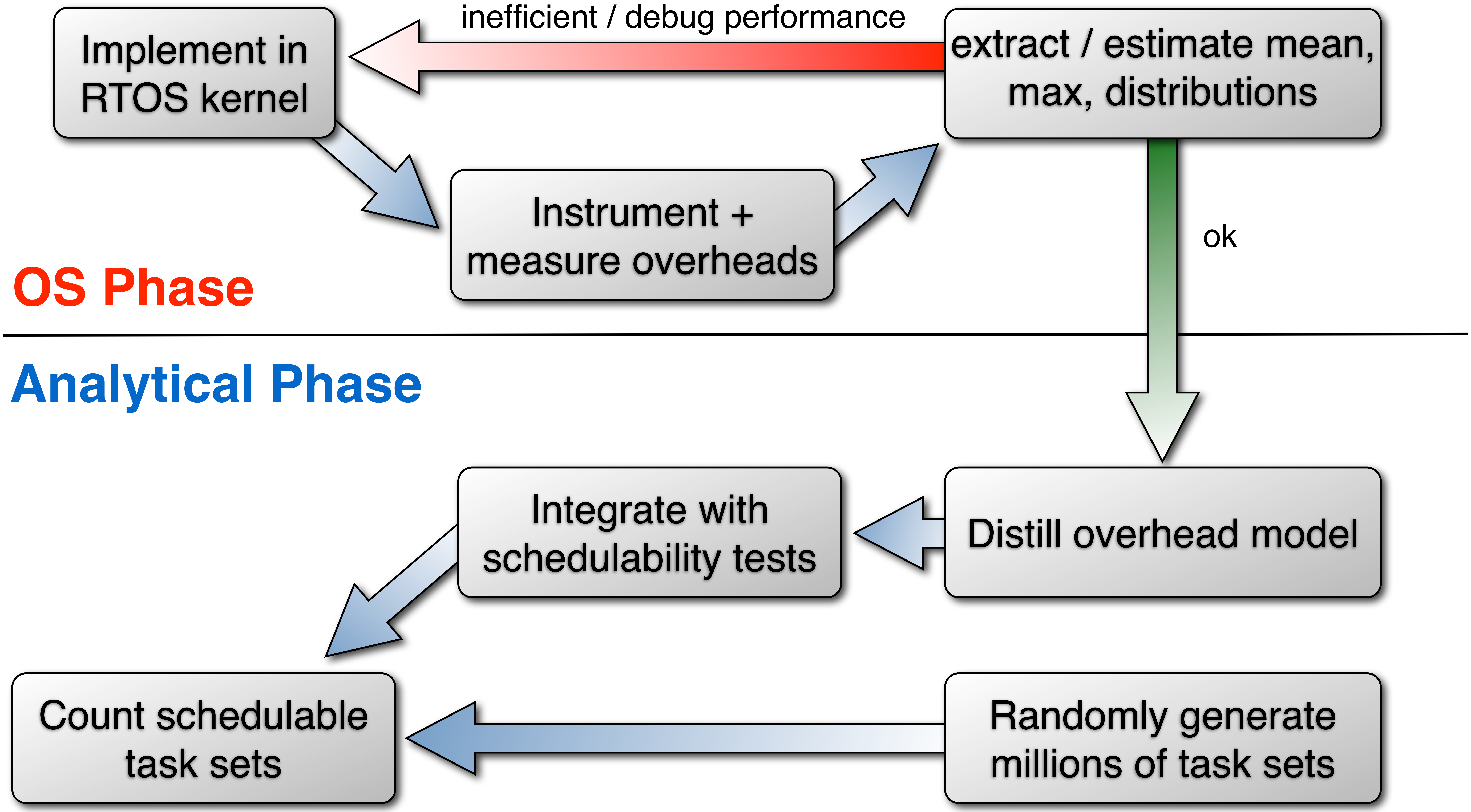
OS Phase

Analytical Phase

Scheduler Evaluation Methodology

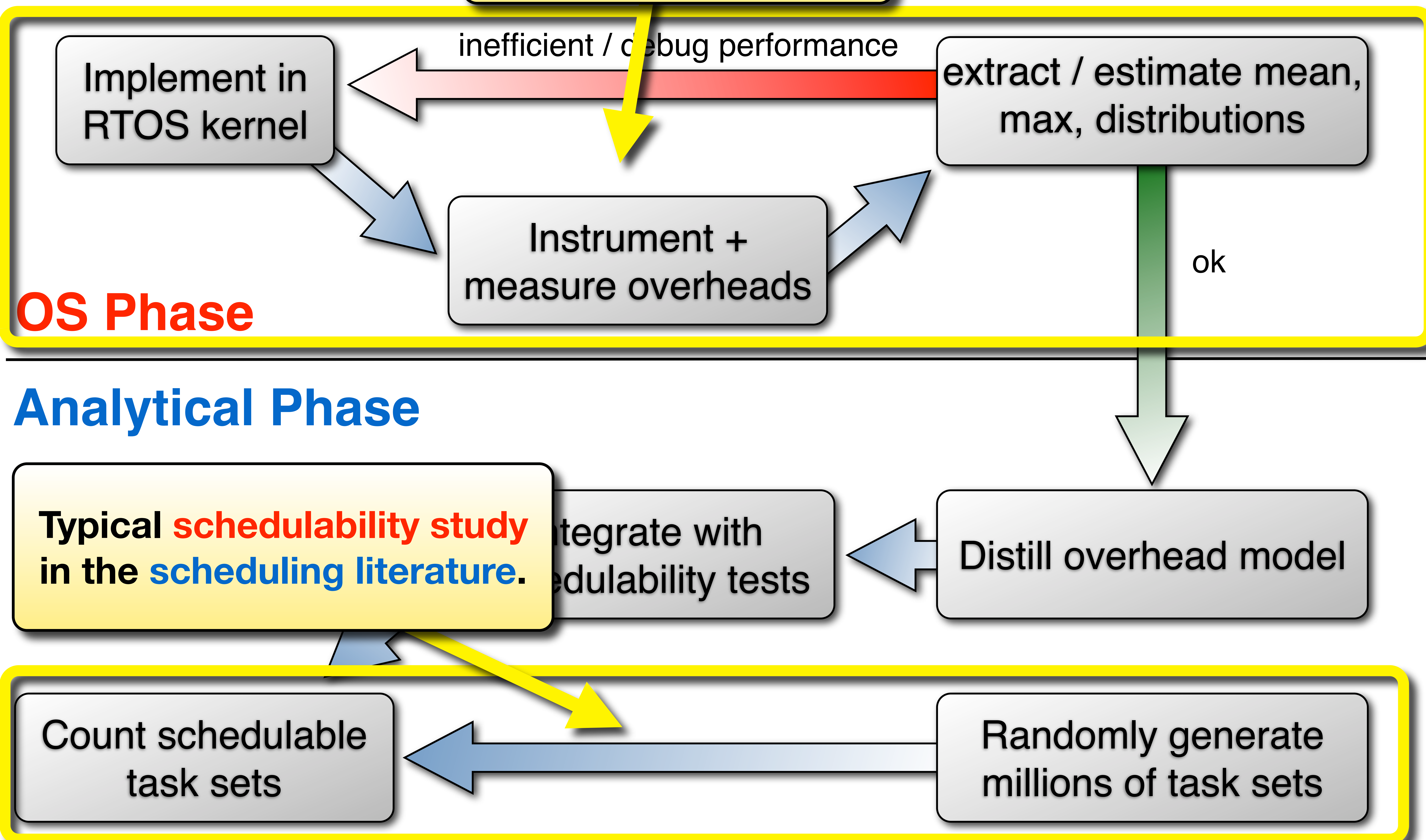


Scheduler Evaluation Methodology



Scheduler … Methodology

Typical **RTOS** study.



Schedu… y

Implement in RTOS kernel

LITMUS RT

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

Developed over span of 5 years.

Current diff to Linux 2.6.36:

93 files changed, **14,465** insertions, **36** deletions

…ean,

measure overheads

OS Phase

Analytical Phase

Integrate with schedulability tests

Distill overhead model

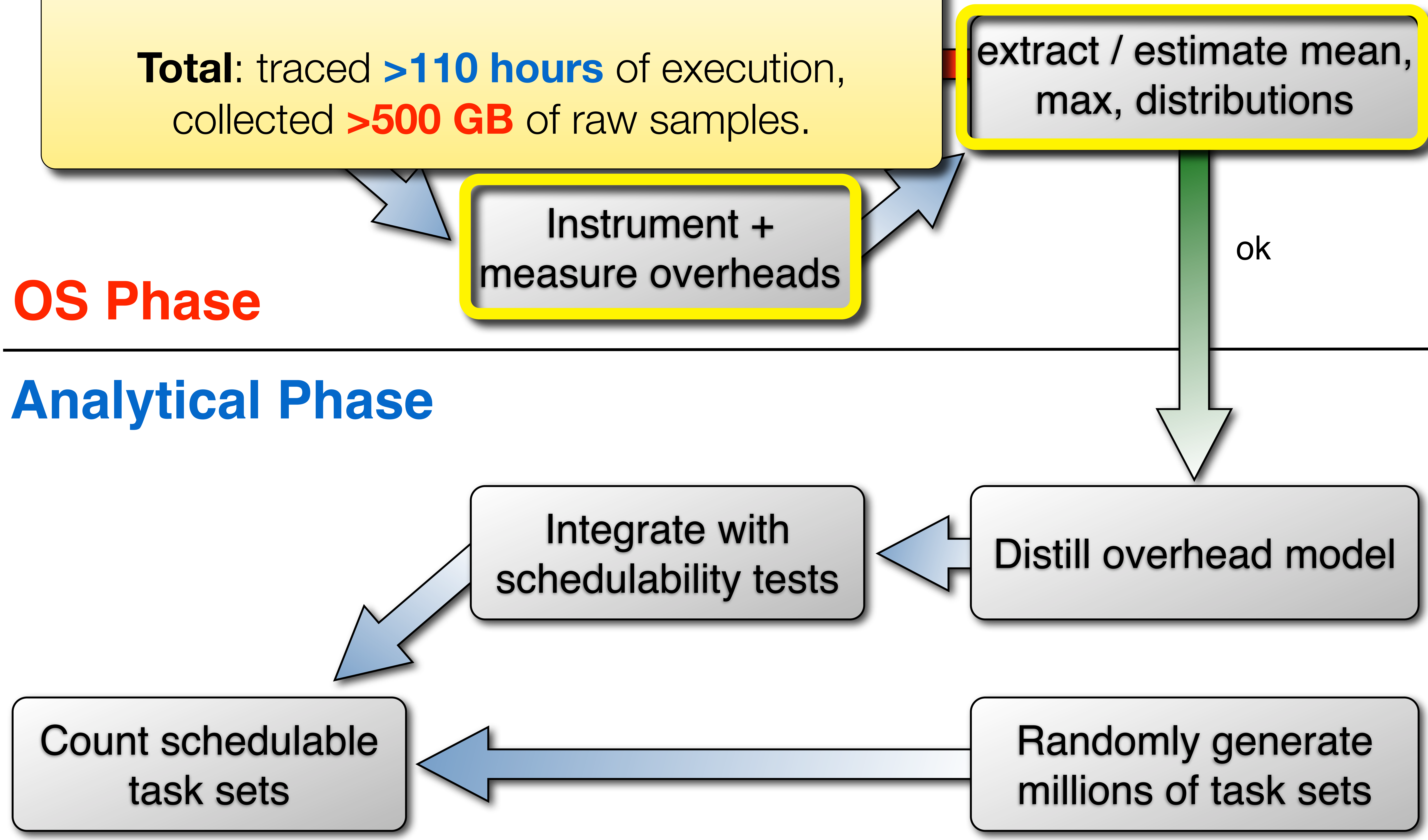
Count schedulable task sets

Randomly generate millions of task sets

Methodology

For **each** scheduler,
ran **200 task sets** with 1-20 tasks per core.

Total: traced **>110 hours** of execution,
collected **>500 GB** of raw samples.



Scheduler Evaluation Methodology

Implement in RTOS kernel

inefficient / debug performance

extract / estimate mean

Model: monotonic piece-wise linear interpolation

Kernel overheads: function of **task count**

Cache affinity loss:
function of **working set size (WSS).**

OS Phase

Analytical Phas

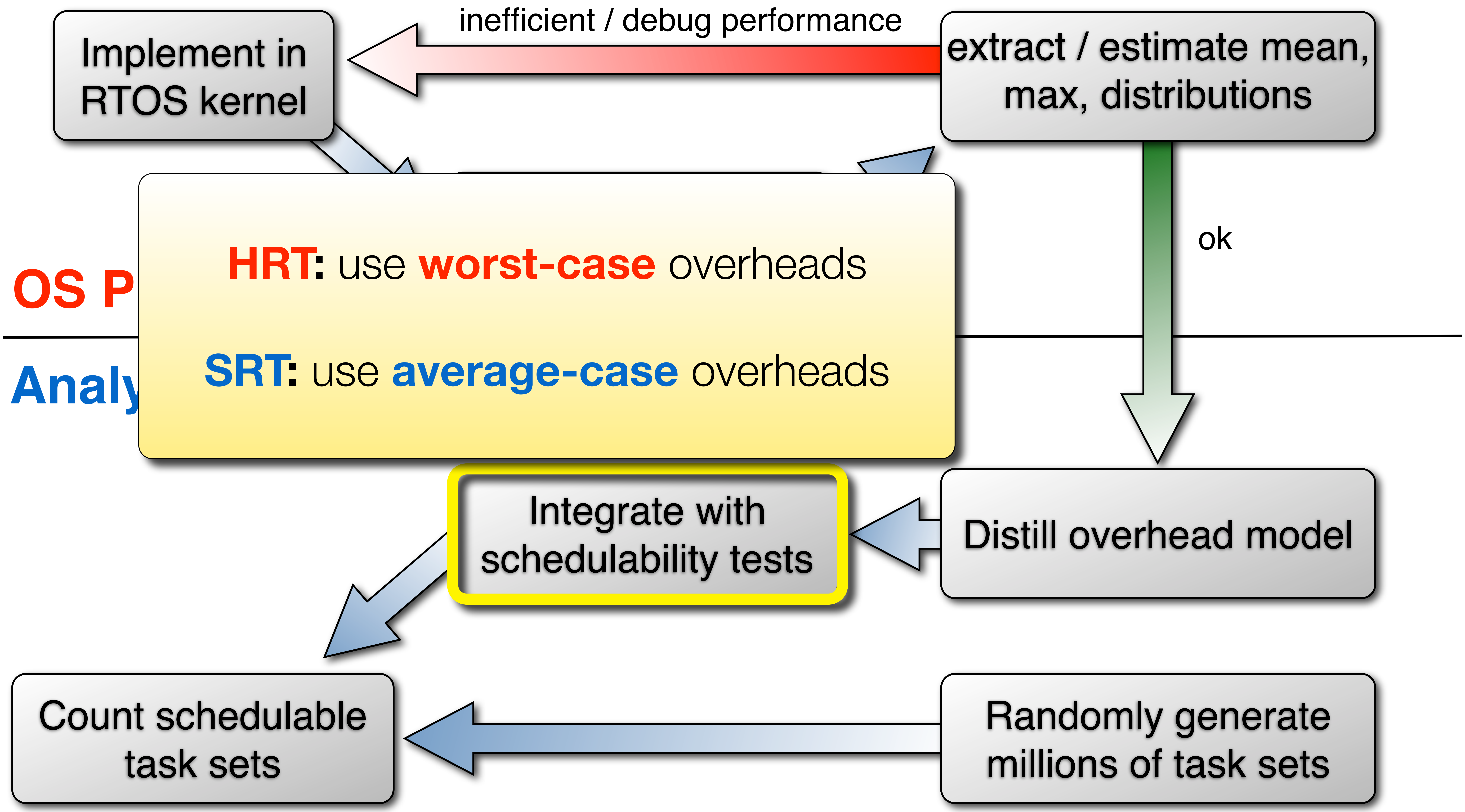
Integrate with schedulability tests

Distill overhead model

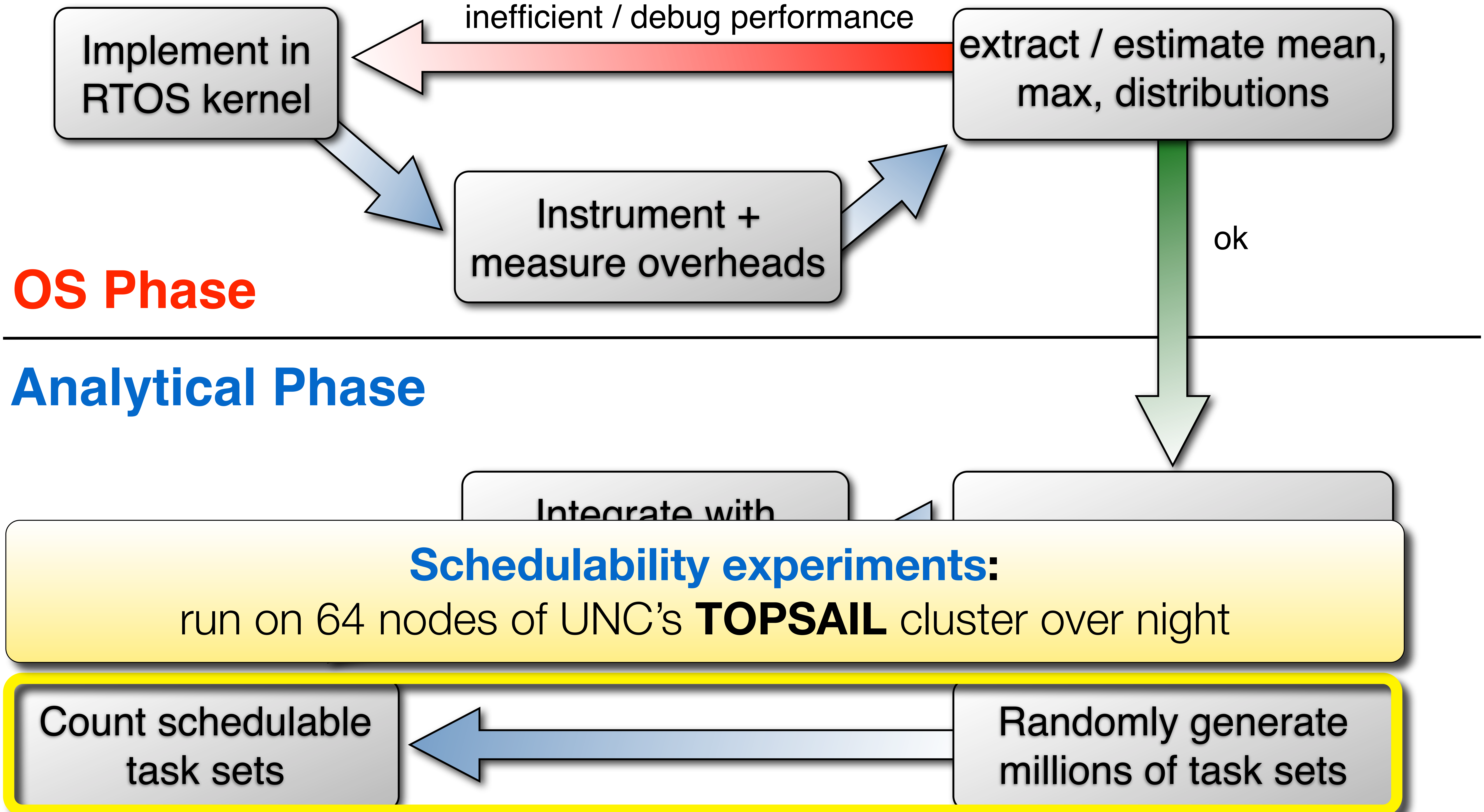
Count schedulable task sets

Randomly generate millions of task sets

Scheduler Evaluation Methodology

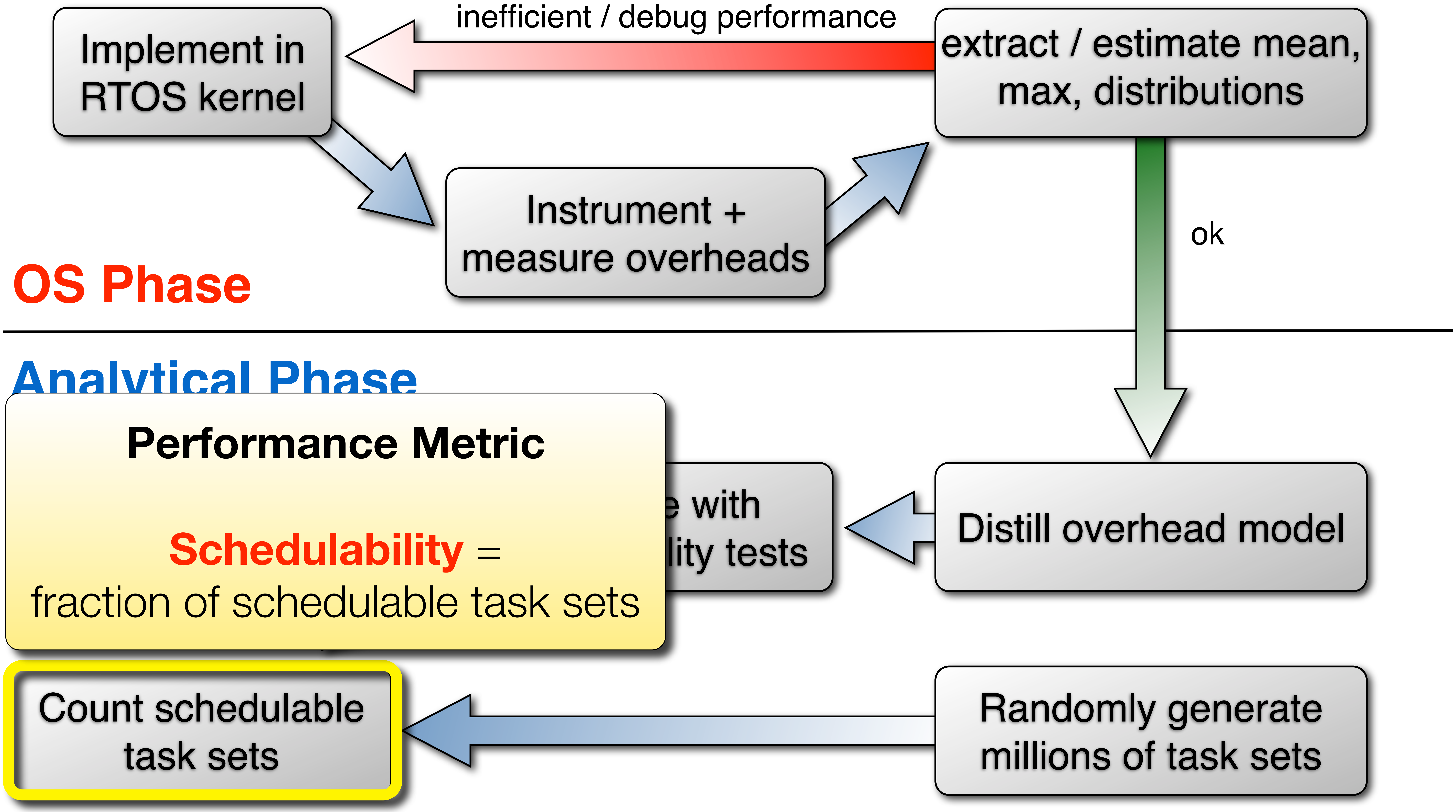


Scheduler Evaluation Methodology



Schedulability experiments:
run on 64 nodes of UNC's **TOPSAIL** cluster over night

Scheduler Evaluation Methodology



Thesis Statement

When both overhead-related and algorithmic capacity loss are considered on a current multicore platform,

(i) partitioned scheduling is preferable to global and clustered approaches in the hard real-time case,

(ii) partitioned earliest-deadline first (P-EDF) scheduling is superior to partitioned fixed-priority (P-FP) scheduling and

(iii) clustered scheduling can be effective in reducing the impact of bin-packing limitations in the soft real-time case. Further,

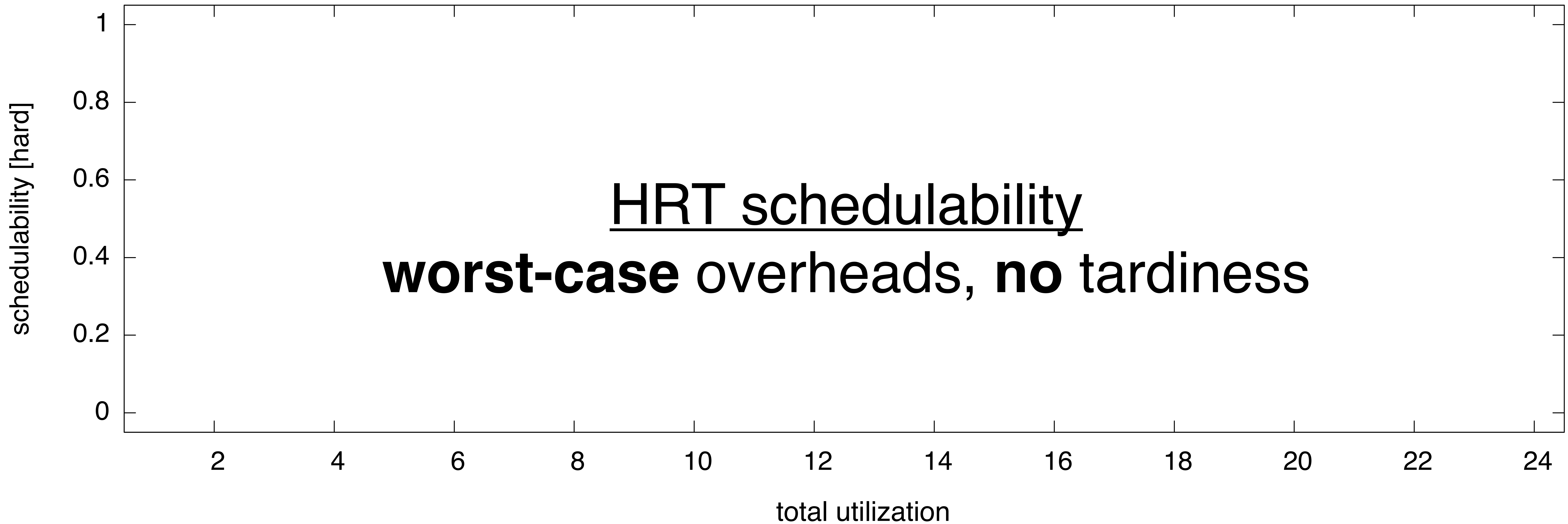
(iv) <u>multiprocessor locking protocols</u> exist that are both efficiently implementable and asymptotically optimal with regard to the maximum duration of <u>blocking</u>.

(underlined terms will be defined shortly)

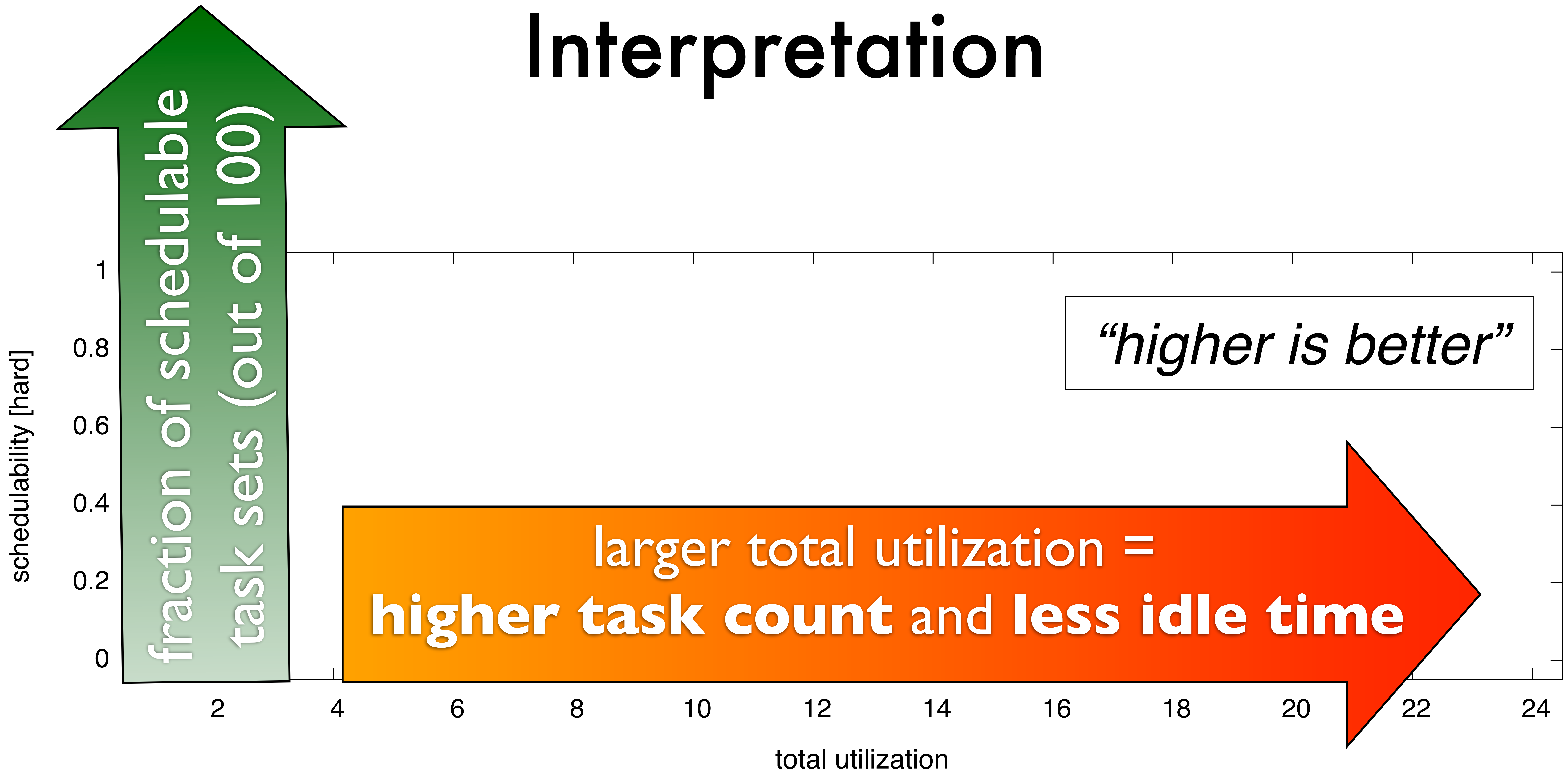
Task Parameters

	In this talk	In my dissertation
Utilizations	uniformly in HRT: 10% – 40% SRT: 50% – 90%	27 utilization & period distributions
Task Periods / Implicit Deadlines	uniformly in [10, 100] ms	
Working Set Size (WSS)	64 KB	0 KB – 3072 KB

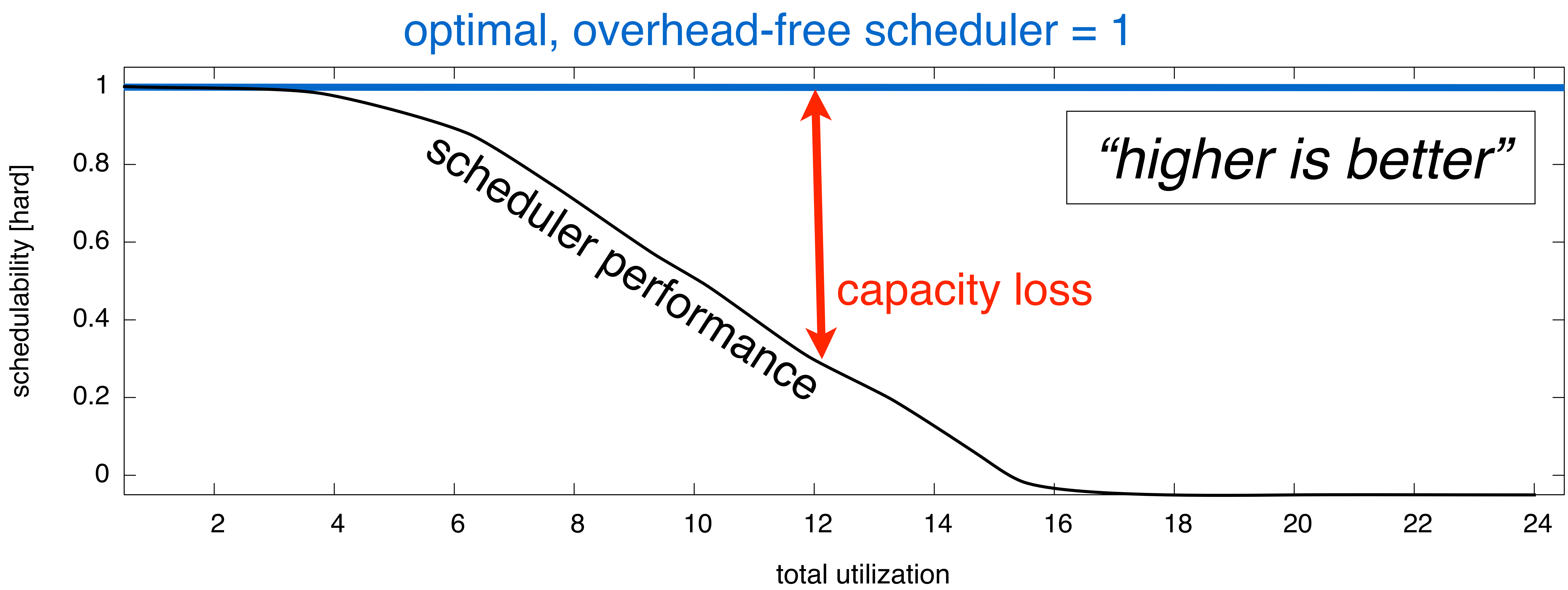
First Result: HRT Schedulability



Interpretation

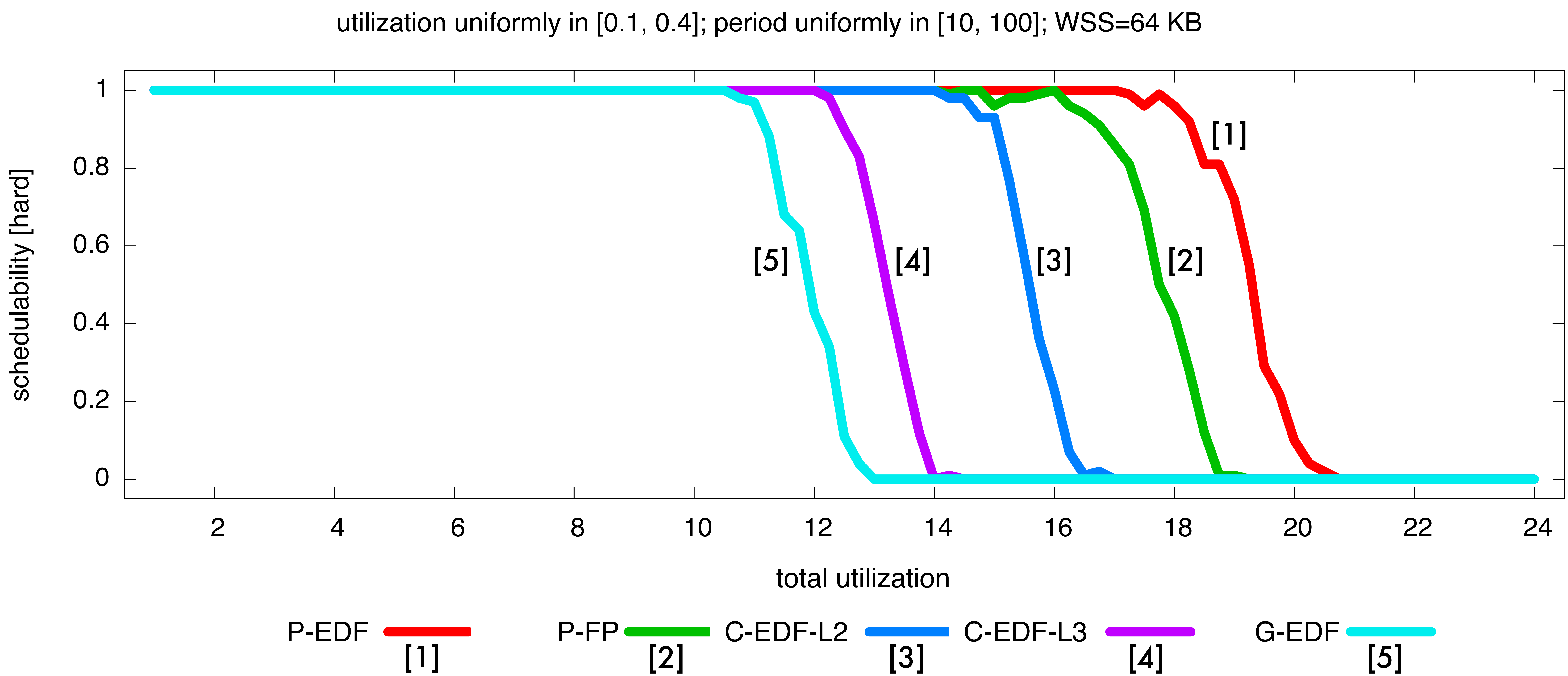


Interpretation



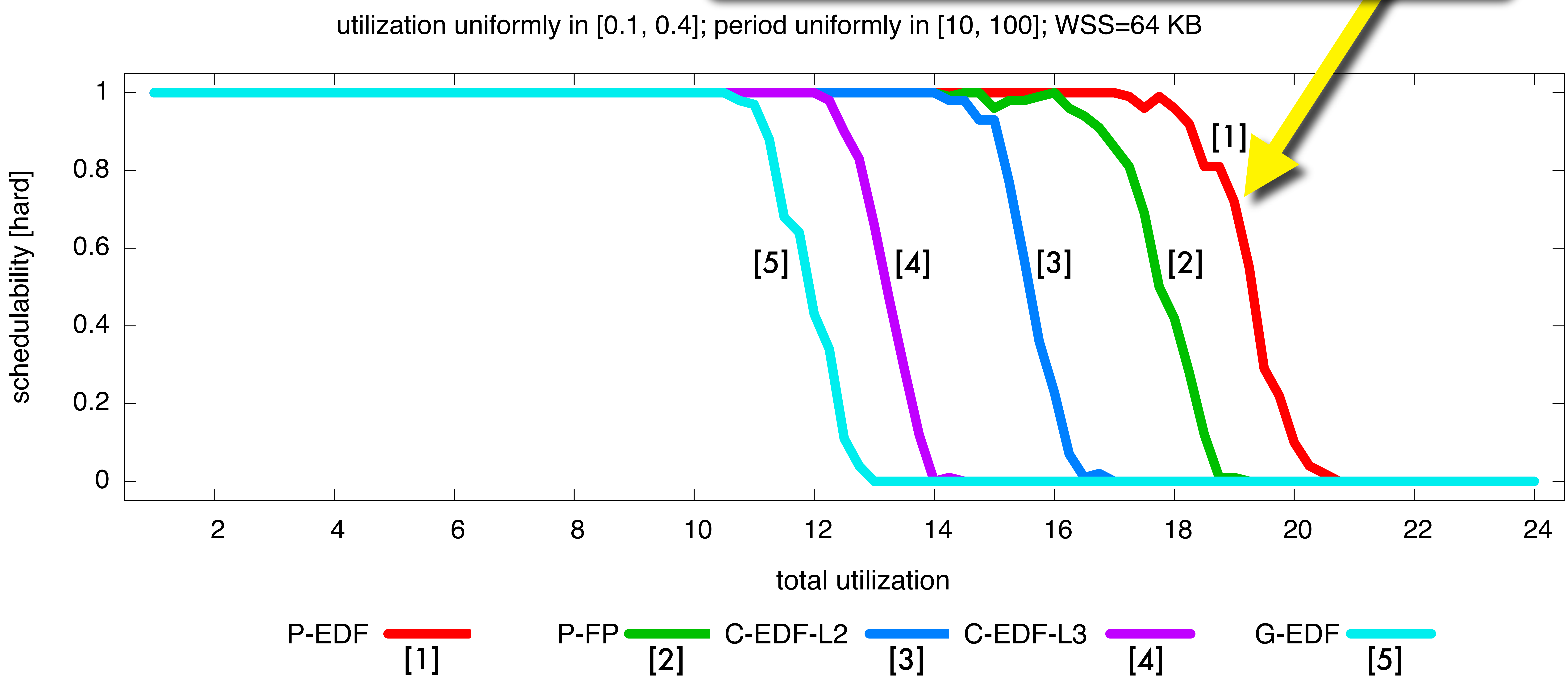
Gap to **y=1** (all task sets schedulable) reflects **capacity loss**.

First Result: HRT Schedulability



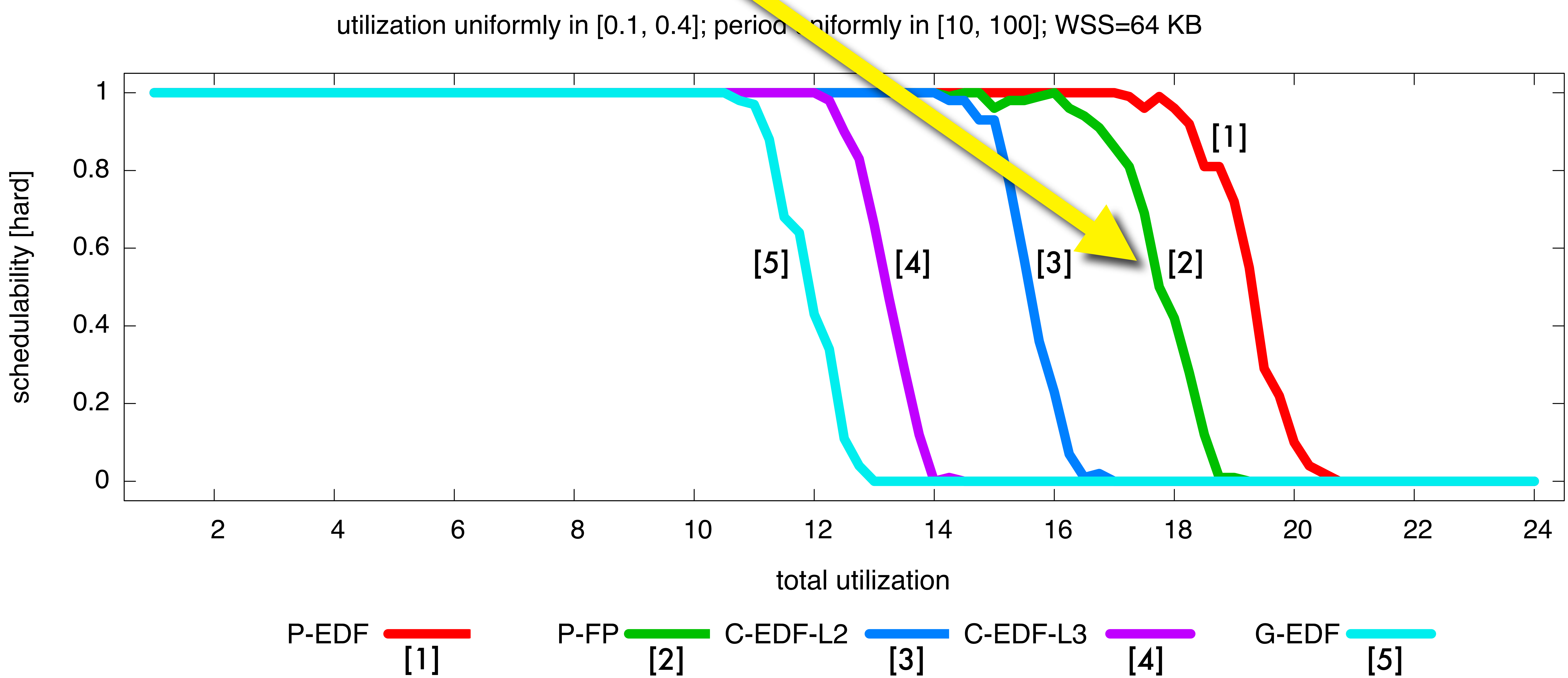
First Result: H

Partitioned EDF suffers least capacity loss.
Low overheads & little algorithmic loss.



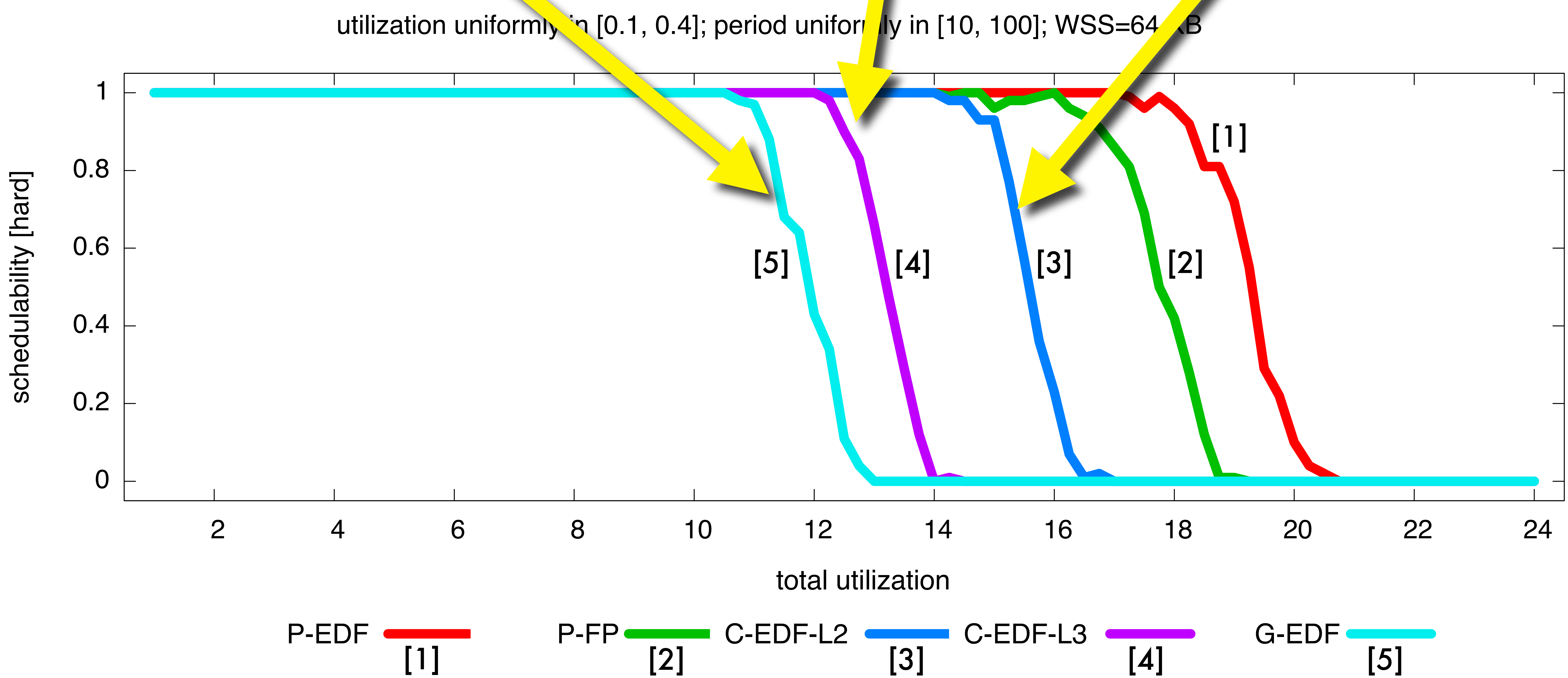
F ity

Partitioned FP performs worse than Partitioned EDF.
Low overheads & more algorithmic loss.



First F

Larger cluster sizes less competitive.
Larger cluster size = higher overheads.



Thesis Statement

When both overhead-related and algorithmic capacity loss are considered on a current multicore platform,

(i) partitioned scheduling is preferable to global and clustered approaches in the hard real-time case,

(ii) partitioned earliest-deadline first (P-EDF) scheduling is superior to partitioned fixed-priority (P-FP) scheduling and

(iii) clustered scheduling can be effective in reducing the impact of bin-packing limitations in the soft real-time case. Further,

(iv) multiprocessor locking protocols exist that are both efficiently implementable and asymptotically optimal with regard to the maximum duration of blocking.

(underlined terms will be defined shortly)

Thesis Statement

When both overhead-related and algorithmic capacity loss are considered on a current multicore platform,

(i) partitioned scheduling is preferable to global and clustered approaches in the hard real-time case,

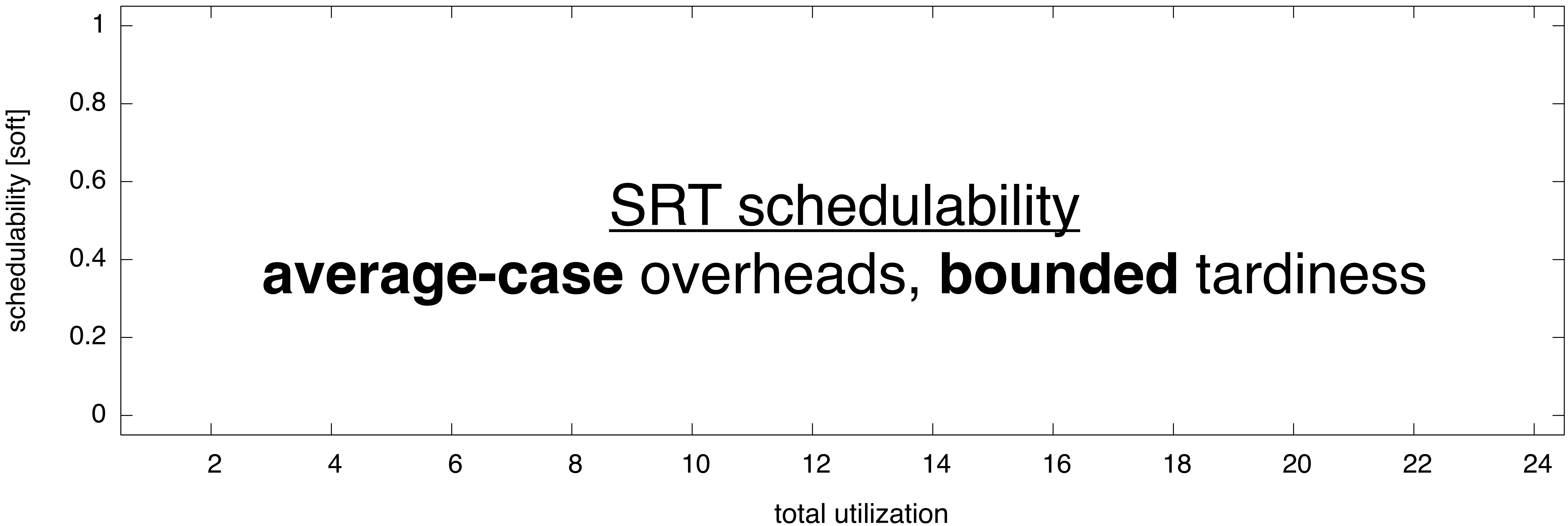
(ii) partitioned earliest-deadline first (P-EDF) scheduling is superior to partitioned fixed-priority (P-FP) scheduling and

(iii) clustered scheduling can be effective in reducing the impact of bin-packing limitations in the soft real-time case. Further,

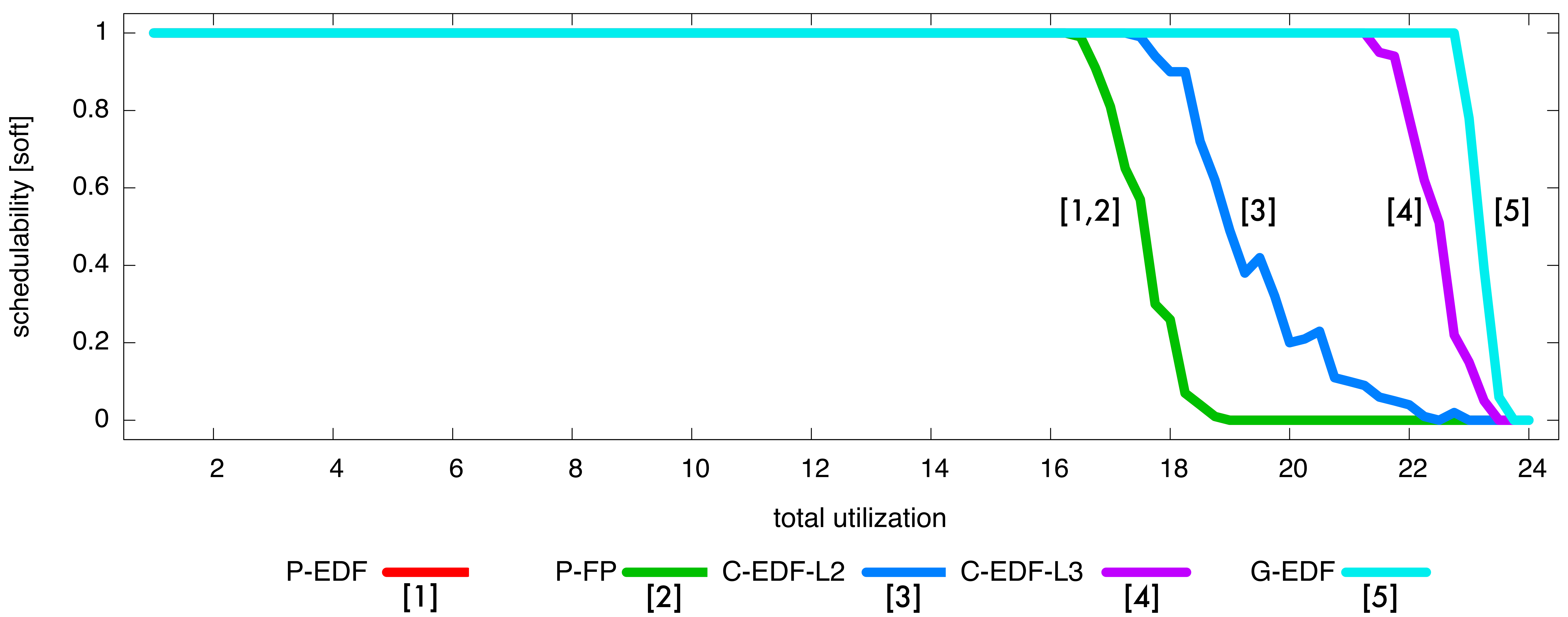
(iv) multiprocessor locking protocols exist that are both efficiently implementable and asymptotically optimal with regard to the maximum duration of blocking.

(underlined terms will be defined shortly)

Second Result: SRT Schedulability

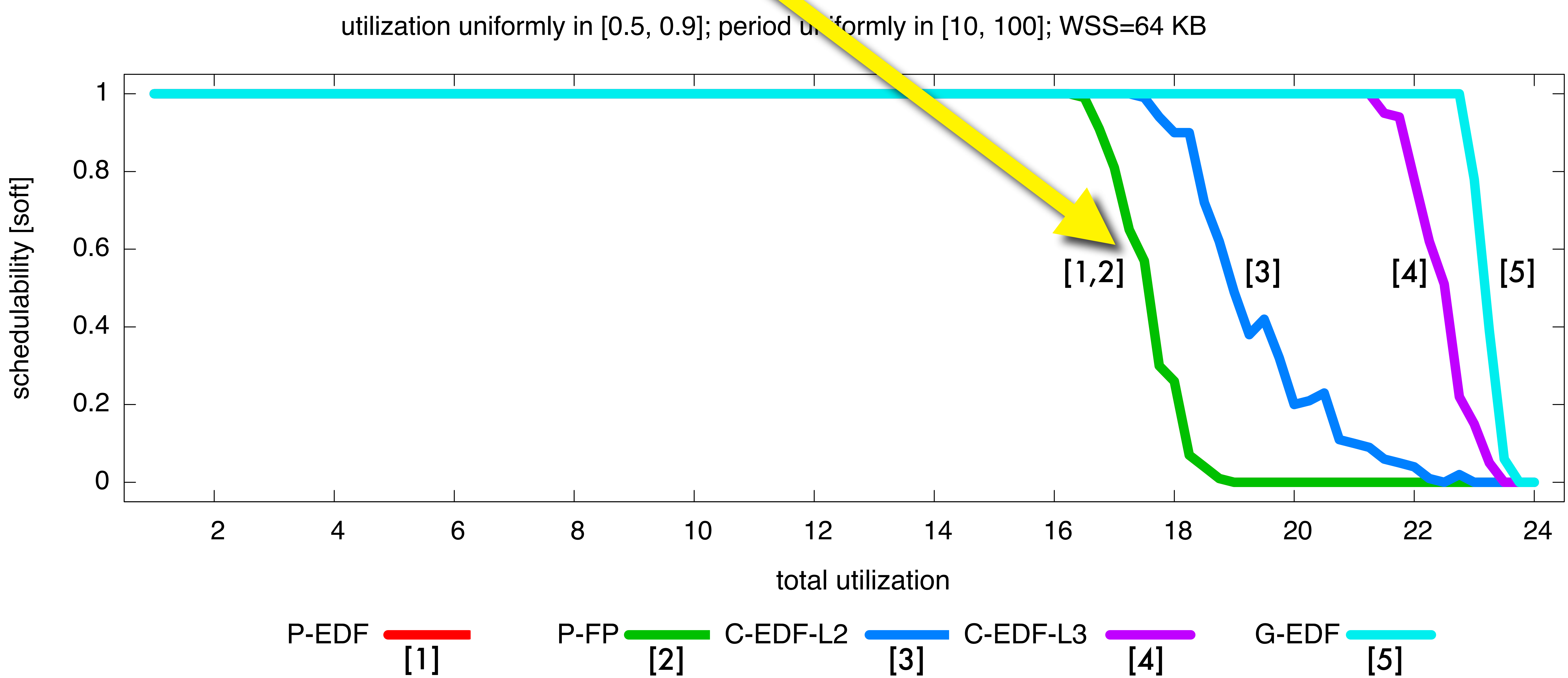


Second Result: SRT Schedulability

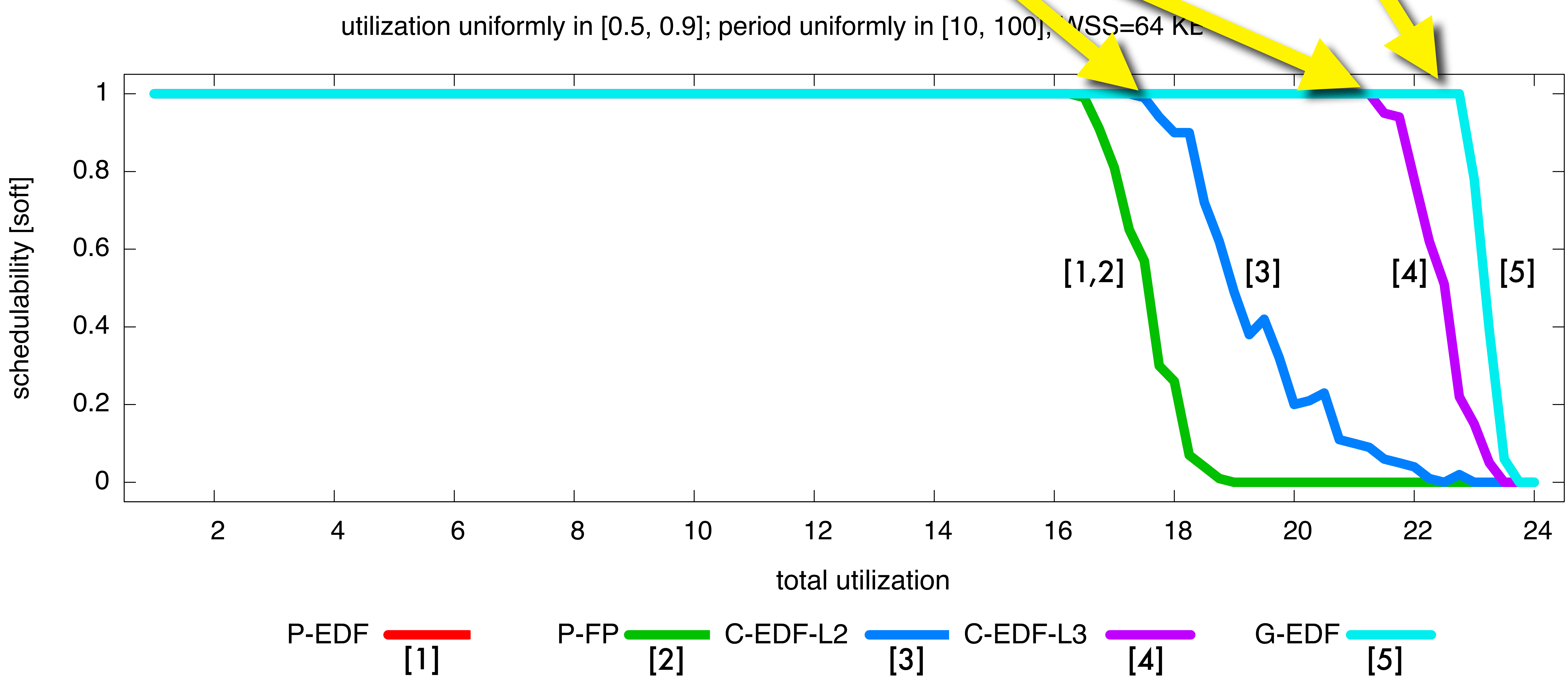


ulability

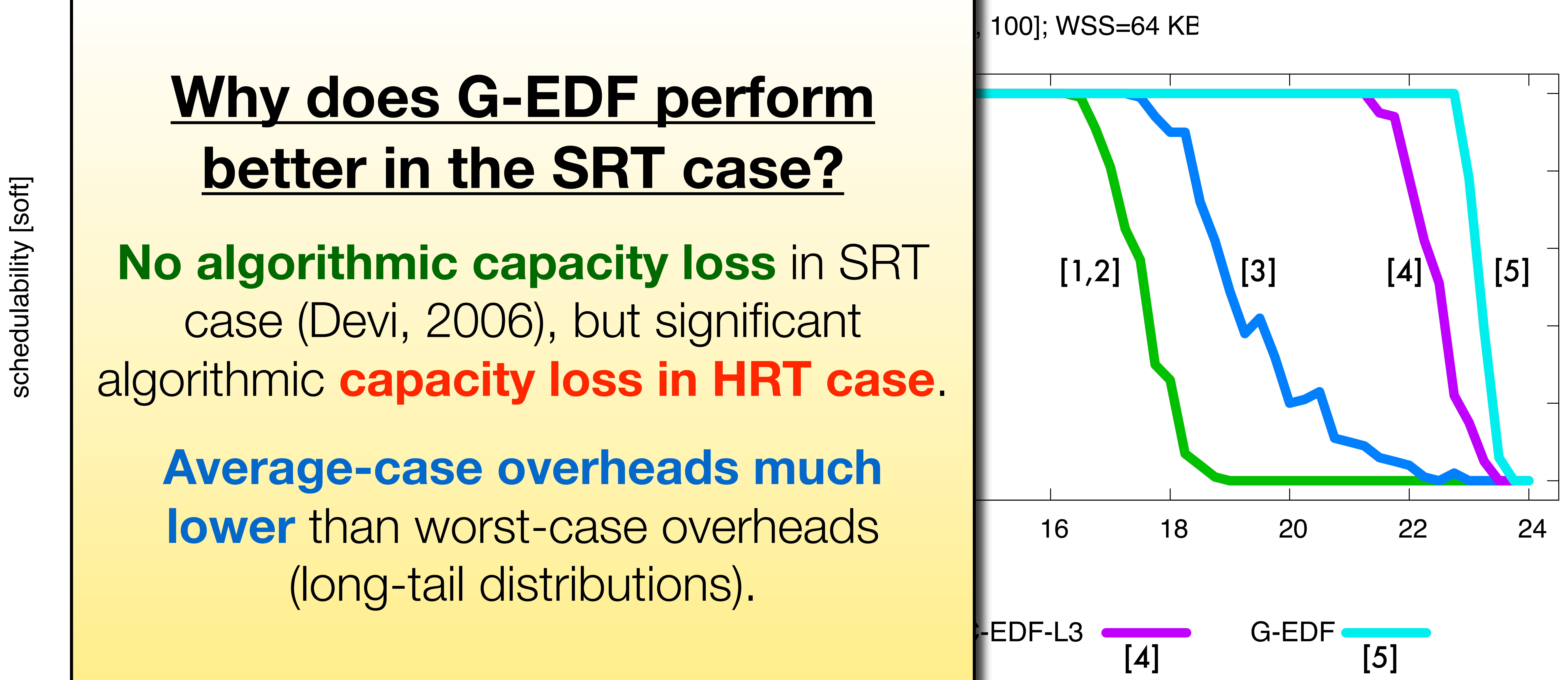
Partitioned FP and Partitioned EDF curves overlap.
Equally affected by bin-packing limitations.



Increasingly competitive with larger cluster sizes.
Effective at overcoming bin-packing issues.



SRT Schedulability



Why does G-EDF perform better in the SRT case?

No algorithmic capacity loss in SRT case (Devi, 2006), but significant algorithmic **capacity loss in HRT case**.

Average-case overheads much lower than worst-case overheads (long-tail distributions).

Thesis Statement

When both overhead-related and algorithmic capacity loss are considered on a current multicore platform,

(i) partitioned scheduling is preferable to global and clustered approaches in the hard real-time case,

(ii) partitioned earliest-deadline first (P-EDF) scheduling is superior to partitioned fixed-priority (P-FP) scheduling and

(iii) clustered scheduling can be effective in reducing the impact of bin-packing limitations in the soft real-time case. Further,

(iv) multiprocessor locking protocols exist that are both efficiently implementable and asymptotically optimal with regard to the maximum duration of blocking.

(underlined terms will be defined shortly)

Full study:

— evaluated more than **92,000,000** task sets.

— results in more than **60,000** schedulability plots.

(i) partitioned scheduling is preferable to global and clustered approaches in the hard real-time case,

(ii) partitioned earliest-deadline first (P-EDF) scheduling is superior to partitioned fixed-priority (P-FP) scheduling and

(iii) clustered scheduling can be effective in reducing the impact of bin-packing limitations in the soft real-time case. Further,

(iv) multiprocessor locking protocols exist that are both efficiently implementable and asymptotically optimal with regard to the maximum duration of blocking.

(underlined terms will be defined shortly)

Part 2

Mutual Exclusion

Serially-Reusable Shared Resources

message buffers, I/O devices, device state,…

Mutual Exclusion

Resources protected by **locks**.

Real-Time Locking Protocol

Avoid **unpredictable** / **unbounded** **blocking** due to unavailable resources.

Spinlocks vs. Semaphores

Jobs must **wait** for resources to become available.

Suspend

=

taken off the ready queue
by the RTOS

=

semaphore

Busy-Wait / Spin

=

non-preemptively
execute delay loop

=

spinlock

Part 2: Contributions

Concerning semaphore protocols.

- ➡ Notion of **blocking optimality**.
- ➡ Several **asymptotically optimal semaphore protocols**.
- ➡ These protocols **perform well in practice**.

Part 2: Contributions

Concerning semaphore protocols.

- ➡ Notion of **blocking optimality**.
- ➡ Several **asymptotically optimal semaphore protocols**.
- ➡ These protocols **perform well in practice**.

Concerning spinlock protocols.

- ➡ Improved blocking analysis (very technical; not discussed).
- ➡ **Overhead-aware comparison** of semaphores and spinlocks in terms of schedulability.

Part 2: Co

High-level view of semaphore protocols first.

Concerning semaphore protocols.

- Notion of **blocking optimality**.
- Several **asymptotically optimal semaphore protocols**.
- These protocols **perform well in practice**.

Concerning spinlock protocols.

- Improved blocking analysis (very technical; not discussed)
- **Overhead-aware comparison** of semaphores and spinlocks in terms of schedulability.

What is "Blocking"?

Not every delay is "blocking" in a real-time system.

<u>Uniprocessor</u>:

Higher-priority jobs *should not have to wait for* ***lower-priority jobs****.*

Lower-priority jobs *should always wait for* ***higher-priority jobs****.*

What is "Blocking"?

Not every delay is "blocking" in a real-time system.

Priority Inversion

A **higher-priority** job is **delayed**
because it **waits** for a **lower-priority** job.

(job **should** be scheduled, but **is** not)

What is "Blocking"?

"blocking in a real-time system"

=

times of priority inversion

=

pi-blocking

The Generalization Question

Uniprocessor PI-Blocking Optimality

On a uniprocessor, the real-time mutual exclusion problem can be solved with ***O*(1) maximum pi-blocking**.

[Sha, Rajkumar, and Lehozcky, 1990; Baker, 1991]

The Generalization Question

Uniprocessor PI-Blocking Optimality

On a uniprocessor, the real-time mutual exclusion problem can be solved with ***O*(1) maximum pi-blocking**.

[Sha, Rajkumar, and Lehoczky, 1990; Baker, 1991]

Any task in any task set: pi-blocked by at most one critical section.

The Generalization Question

Uniprocessor PI-Blocking Optimality

On a uniprocessor, the real-time mutual exclusion problem can be solved with ***O*(1) maximum pi-blocking**.

[Sha, Rajkumar, and Lehozcky, 1990; Baker, 1991]

How does the bound generalize to **multiprocessor**?

***O*(1)?** ***O*(*m*)?** **O(*n*)?** Worse?

m identical processors

n sporadic tasks

The Generalization Question

My Result: it depends.

— there are two kinds of schedulability analysis —

How does the bound generalize to **multiprocessor**?

$O(1)$? **$O(m)$**? **$O(n)$**? Worse?

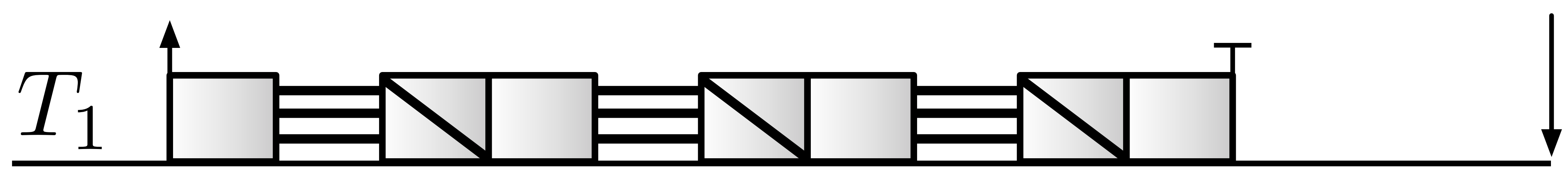
m identical processors

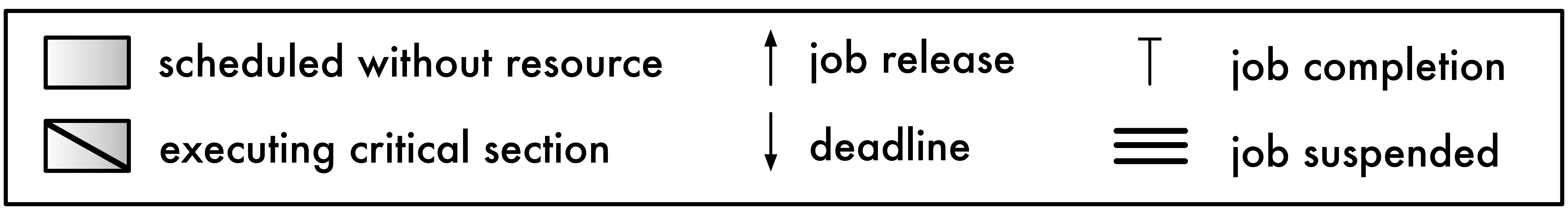
n sporadic tasks

Two Kinds of Schedulability Analysis

analyzing suspensions is notoriously difficult

actual execution:



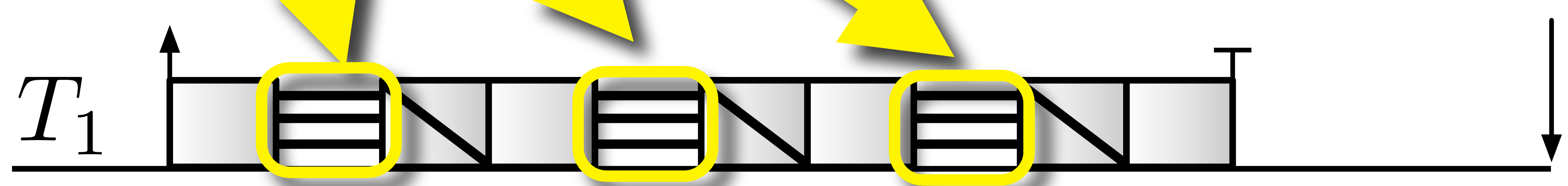


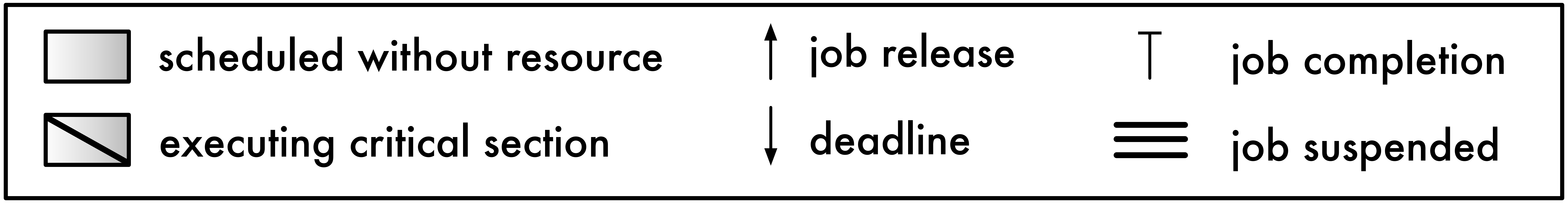
Two … sis

Processor not used = other jobs can execute.

analyzing suspensions is notoriously difficult

actual execution:



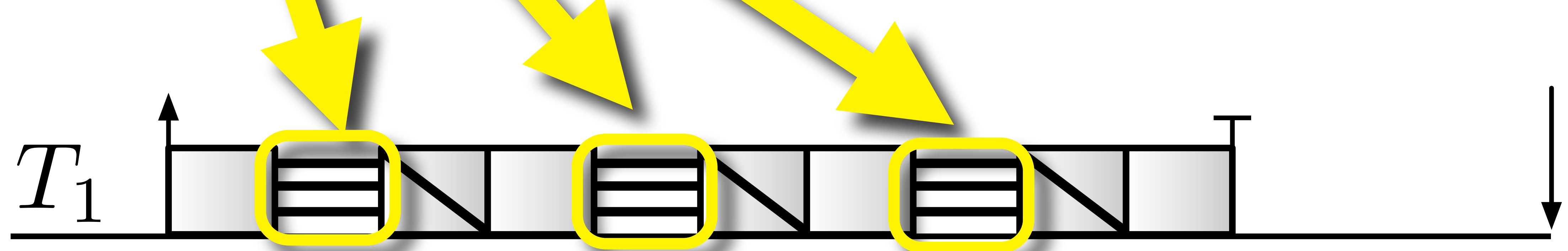


Two … sis

Processor not used = other jobs can execute.

analyzing suspensions is notoriously difficult

actual execution:



predictability requires *a priori* analysis

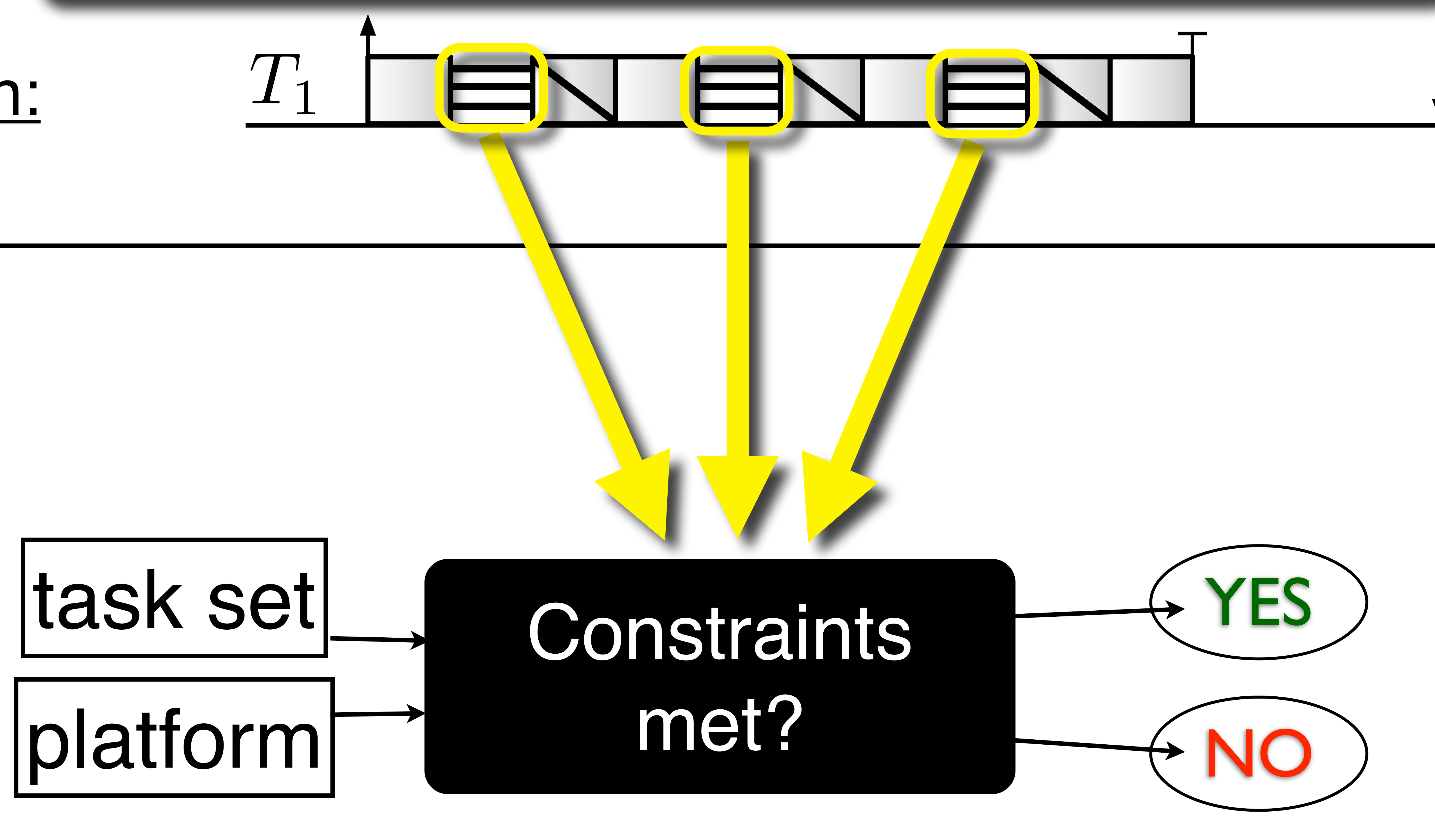
schedulability test

Two Kinds of Schedulability Analysis

ana … *ult*

Exploiting knowledge of suspensions in schedulability tests is very difficult.

actual execution:



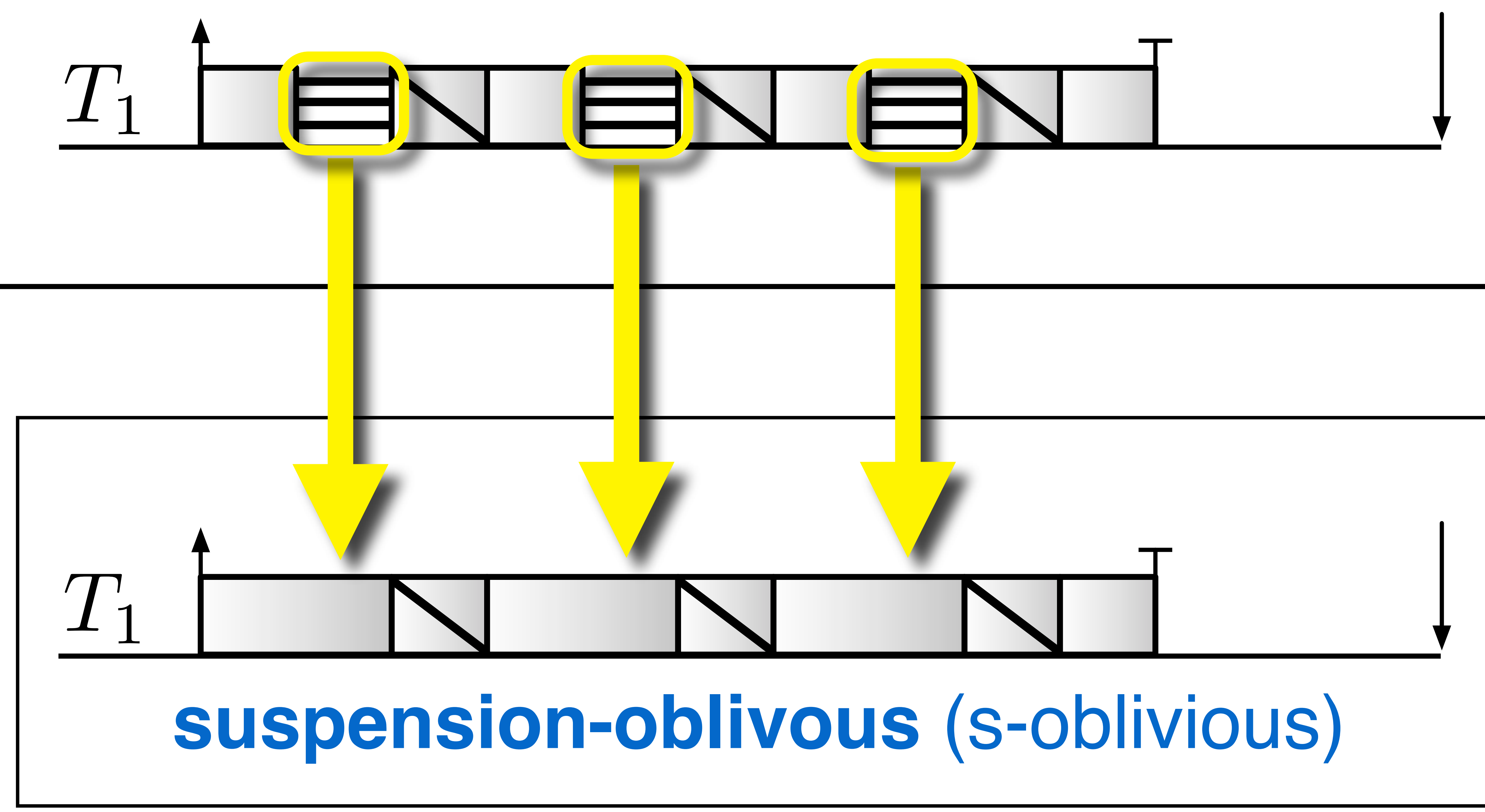
schedulability test

Two Kinds of Schedulability Analysis

analyzing suspensions is notoriously difficult

actual execution:

analyzed as:



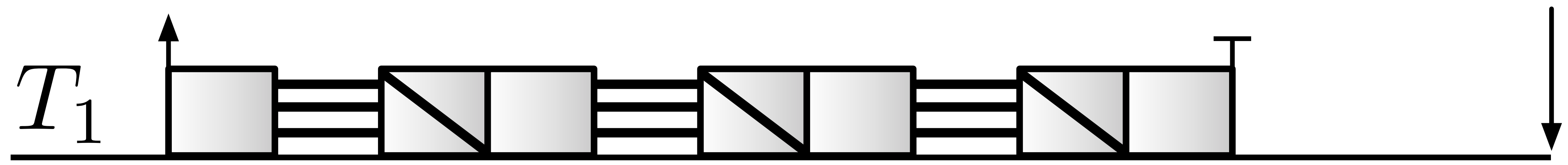
simplifying, safe assumption:

treat **suspension time** as **execution time**

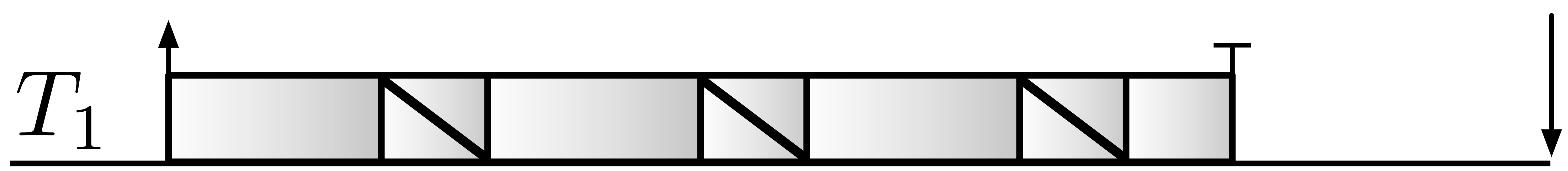
Two Kinds of Schedulability Analysis

analyzing suspensions is notoriously difficult

actual execution:



analyzed as:



suspension-oblivous (s-oblivious)

Ideal:
accurate analysis.

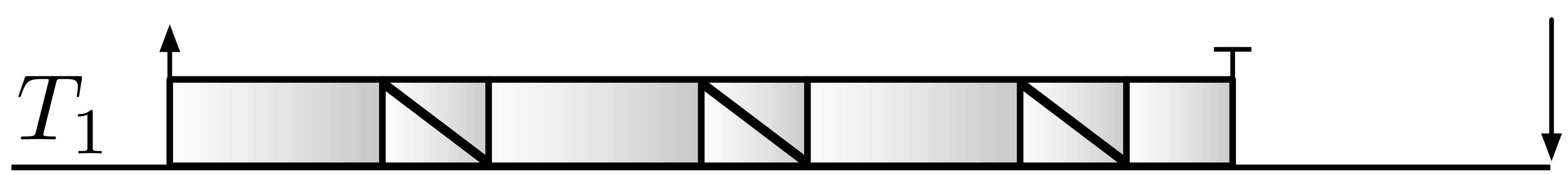
suspension-aware (s-aware)

Two Kinds of Schedulability Analysis

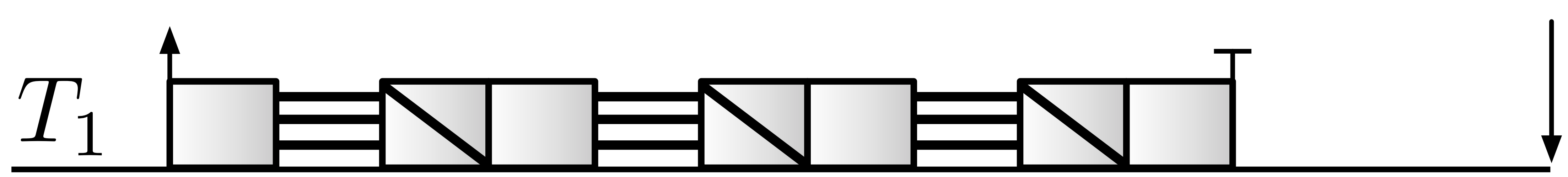
analyzing suspensions is notoriously difficult

The type of schedulability analysis in use subtly affects the definition of pi-blocking.

analyzed as:

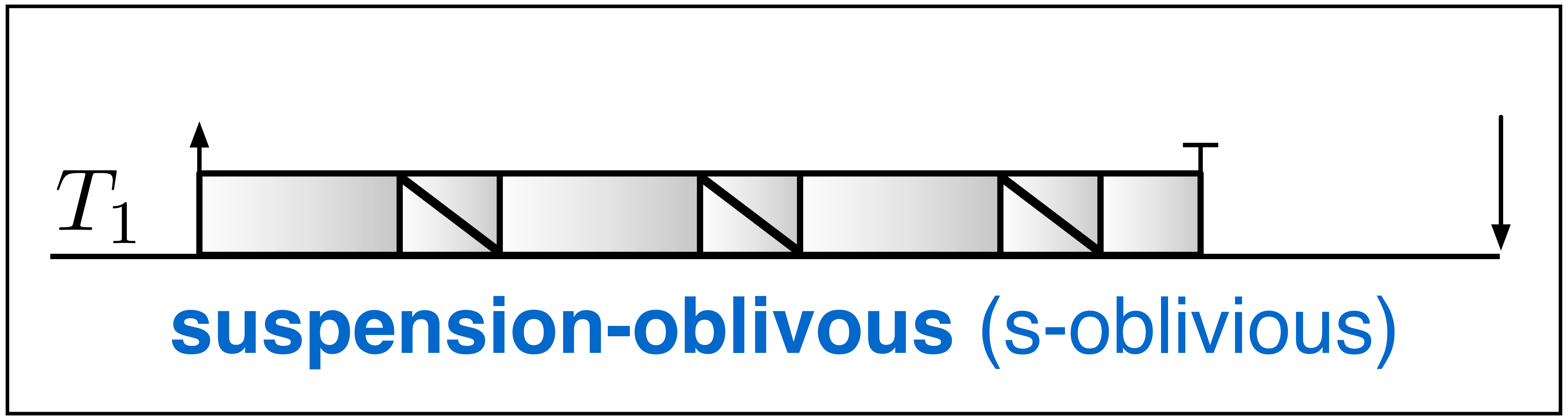


suspension-oblivous (s-oblivious)



suspension-aware (s-aware)

Suspension-Oblivious Results



Suspensions modeled as execution.

Suspension-Oblivious Results

m identical processors ***n*** sporadic tasks

	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global			
Partitioned			
Clustered			

Suspension-Oblivious Results

m identical processors

n sporadic tasks

	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global	$\Omega(m)$		
Partitioned	$\Omega(m)$		
Clustered	$\Omega(m)$		

Suspension-Oblivious Results

m identical processors n sporadic tasks

	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global	$\Omega(m)$		—
Partitioned	$\Omega(m)$		$\Omega(m \cdot n)$ / MPCP-VS (Lakshmanan et al., 2009)
Clustered	$\Omega(m)$		—

MPCP-VS = *Multiprocessor Priority Ceiling Protocol with Virtual Spinning*

Suspension-Oblivious Results

m identical processors ***n*** sporadic tasks

	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global	$\Omega(m)$	$O(m)$ / OMLP	—
Partitioned	$\Omega(m)$	$O(m)$ / OMLP	$\Omega(m \cdot n)$ / MPCP-VS (Lakshmanan et al., 2009)
Clustered	$\Omega(m)$	$O(m)$ / OMLP	—

OMLP = *O(m) Locking Protocol*
MPCP-VS = *Multiprocessor Priority Ceiling Protocol with Virtual Spinning*

Suspension-Oblivious Results

Asymptotically optimal
(approximately within **factor of two**)

n sporadic tasks

	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global	$\Omega(m)$	$O(m)$ / OMLP	—
Partitioned	$\Omega(m)$	$O(m)$ / OMLP	$\Omega(m \cdot n)$ / MPCP-VS (Lakshmanan et al., 2009)
Clustered	$\Omega(m)$	$O(m)$ / OMLP	—

OMLP = *O(m) Locking Protocol*
MPCP-VS = *Multiprocessor Priority Ceiling Protocol with Virtual Spinning*

Susper … us

m identical p… n

Uses FIFO queues.

Uses priority queues.

	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global	$\Omega(m)$	$O(m)$ / OMLP	—
Partitioned	$\Omega(m)$	$O(m)$ / OMLP	$\Omega(m \cdot n)$ / MPCP-VS (Lakshmanan et al., 2009)
Clustered	$\Omega(m)$	$O(m)$ / OMLP	—

OMLP = *$O(m)$ Locking Protocol*
MPCP-VS = *Multiprocessor Priority Ceiling Protocol with Virtual Spinning*

Suspension-Oblivious Results

m identical processors n sporadic tasks

	My Work Lower Bound	My Work	Prior Work
Global	$\Omega(m)$		
Partitioned	$\Omega(m)$	$O(m)$ / OMLP	$\Omega(m \cdot n)$ / MPCP-VS (Lakshmanan et al., 2009)
Clustered	$\Omega(m)$	$O(m)$ / OMLP	—

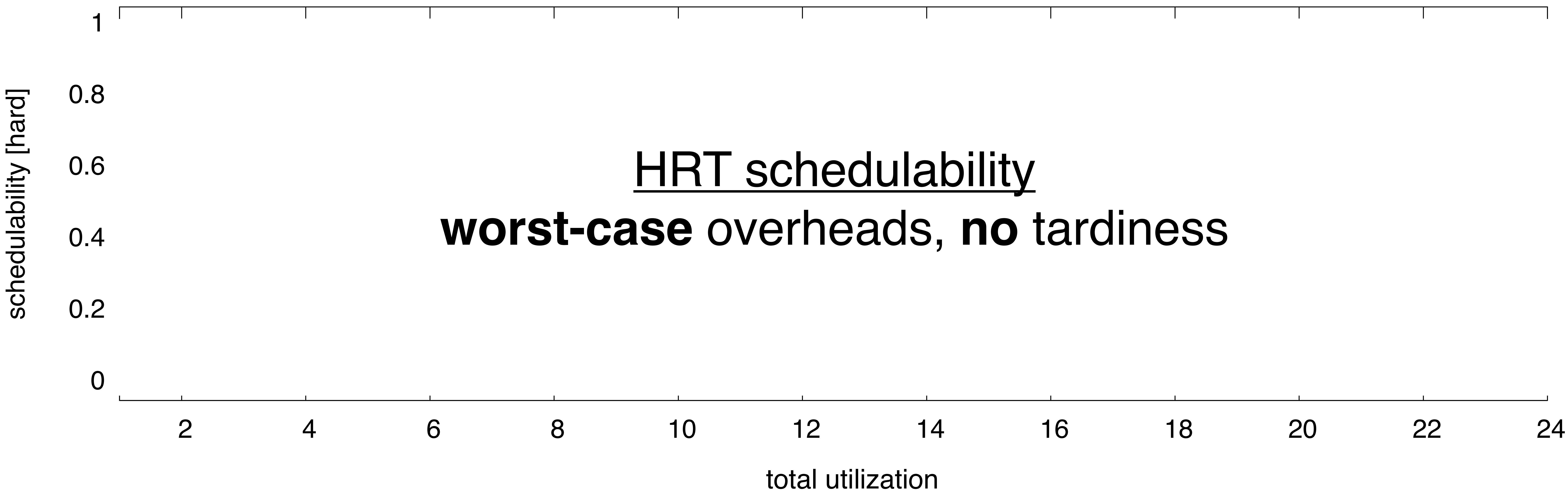
Next: overhead-aware schedulability study for non-asymptotic comparison.

OMLP = *$O(m)$ Locking Protocol*
MPCP-VS = *Multiprocessor Priority Ceiling Protocol with Virtual Spinning*

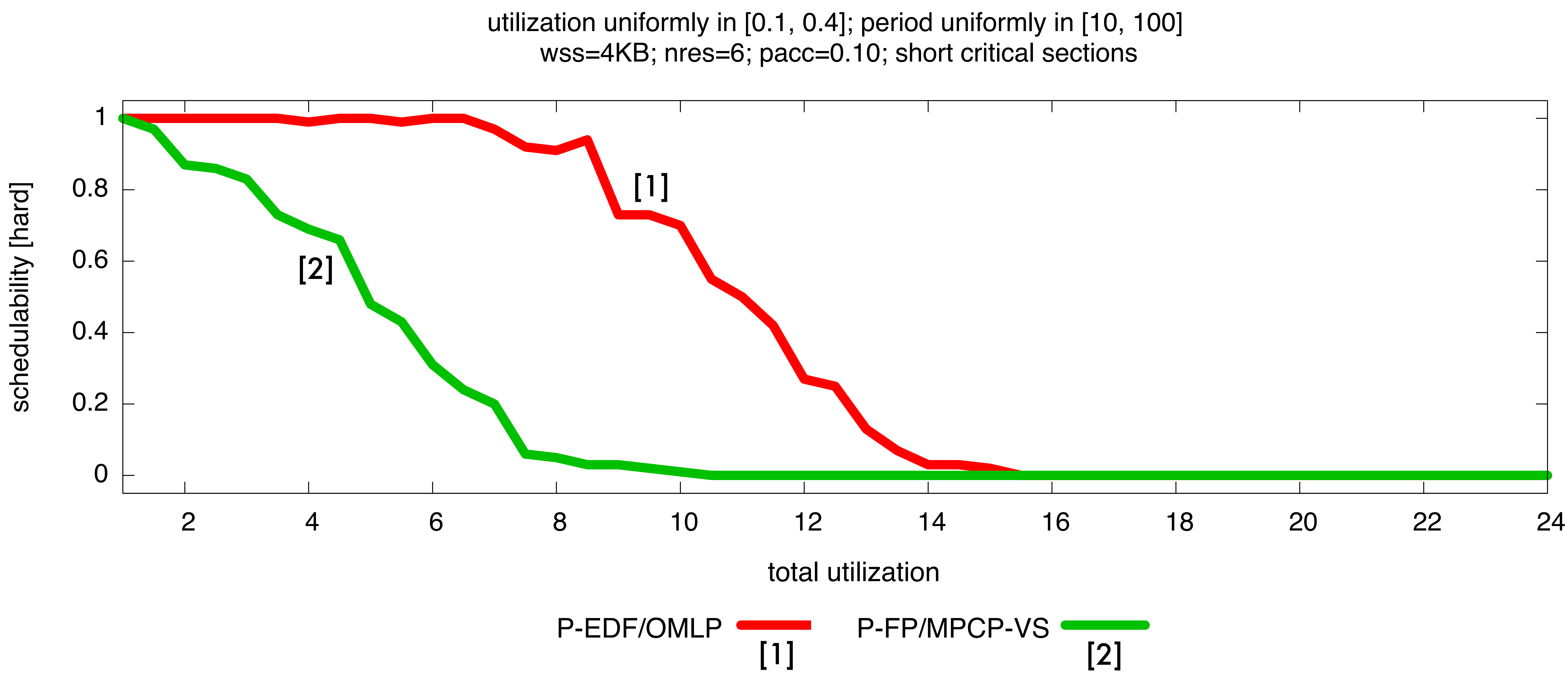
Resource-Sharing Parameters

	In this talk	In my dissertation
Number of resources	6	1, 3, 6, 12, 24
Access probability	25%	10%, 25%, 40%, 55%, 70%, 85%
Critical Section Lengths	uniformly in [1, 15] μs	short: [1, 15] μs medium: [1, 100] μs long: [5, 1280] μs

S-Oblivious Schedulability Comparison



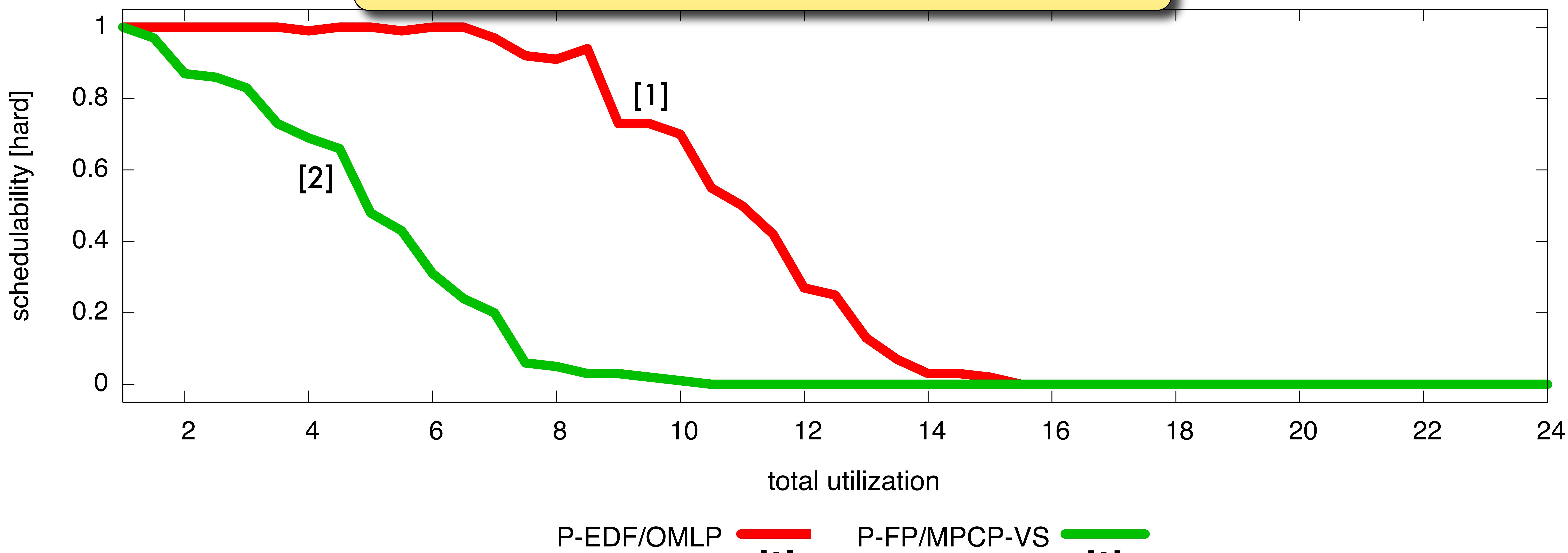
S-Oblivious Schedulability Comparison



S-Obliv... Comparison

OMLP yields better schedulability than the MPCP-VS in in virtually* all tested scenarios.

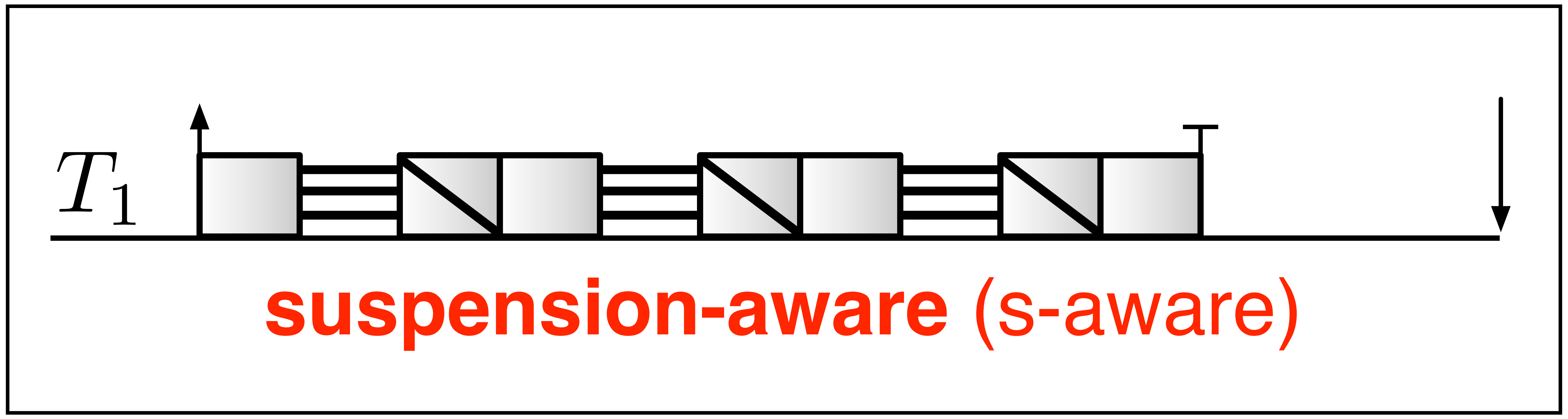
***Long critical sections are equally troublesome under each of the protocols.**



[1]

[2]

Suspension-Aware Results



Suspensions analyzed in detail.

Suspension-Aware Results

m identical processors   *n* sporadic tasks

	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global			
Clustered			
Partitioned			

Suspension-Aware Results

m identical processors n sporadic tasks

	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global	$\Omega(n)$		
Clustered	$\Omega(n)$		
Partitioned	$\Omega(n)$		

Suspension-Aware Results

m identical processors n sporadic tasks

	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global	$\Omega(n)$		$\Omega(m \cdot n)$ / other PCP variant
Clustered	$\Omega(n)$		—
Partitioned	$\Omega(n)$		$\Omega(m \cdot n)$ / MPCP $\Omega(m \cdot n)$ / DPCP

MPCP = *Multiprocessor Priority Ceiling Protocol* (Rajkumar, 1990)
DPCP = *Distributed Priority Ceiling Protocol* (Rajkumar et al., 1988)

Suspension-Aware Results

m identical processors n sporadic tasks

	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global	$\Omega(n)$	(special cases)	$\Omega(m \cdot n)$ / other PCP variant
Clustered	$\Omega(n)$	—	—
Partitioned	$\Omega(n)$	$O(n)$ / FMLP+	$\Omega(m \cdot n)$ / MPCP $\Omega(m \cdot n)$ / DPCP

FMLP+ = *FIFO Mutex Locking Protocol*
MPCP = *Multiprocessor Priority Ceiling Protocol* (Rajkumar, 1990)
DPCP = *Distributed Priority Ceiling Protocol* (Rajkumar et al., 1988)

Suspension-Aware Results

Asymptotically optimal

n sporadic tasks

	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global	$\Omega(n)$	(special cases)	$\Omega(m \cdot n)$ / other PCP variant
Clustered	$\Omega(n)$	—	—
Partitioned	$\Omega(n)$	$O(n)$ / FMLP+	$\Omega(m \cdot n)$ / MPCP $\Omega(m \cdot n)$ / DPCP

FMLP+ = *FIFO Mutex Locking Protocol*
MPCP = *Multiprocessor Priority Ceiling Protocol* (Rajkumar, 1990)
DPCP = *Distributed Priority Ceiling Protocol* (Rajkumar et al., 1988)

Suspension-Aware Results

m identical processor… …ks

Tightness is still an open problem in the general case.

	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global	$\Omega(n)$	— [$O(n)$ in special cases]	$\Omega(m \cdot n)$ / other PCP variant
Clustered	$\Omega(n)$	—	—
Partitioned	$\Omega(n)$	$O(n)$ / FMLP+	$\Omega(m \cdot n)$ / MPCP $\Omega(m \cdot n)$ / DPCP

FMLP+ = *FIFO Mutex Locking Protocol*
MPCP = *Multiprocessor Priority Ceiling Protocol* (Rajkumar, 1990)
DPCP = *Distributed Priority Ceiling Protocol* (Rajkumar et al., 1988)

Suspension-Aware

cessors n

Uses FIFO queues.

Uses priority queues.

	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global	$\Omega(n)$	— [$O(n)$ in special cases]	$\Omega(m \cdot n)$ / other PCP variant
Clustered	$\Omega(n)$	—	—
Partitioned	$\Omega(n)$	$O(n)$ / FMLP+	$\Omega(m \cdot n)$ / MPCP $\Omega(m \cdot n)$ / DPCP

FMLP+ = *FIFO Mutex Locking Protocol*
MPCP = *Multiprocessor Priority Ceiling Protocol* (Rajkumar, 1990)
DPCP = *Distributed Priority Ceiling Protocol* (Rajkumar et al., 1988)

Suspension-Aware Results

m identical processors n sporadic tasks

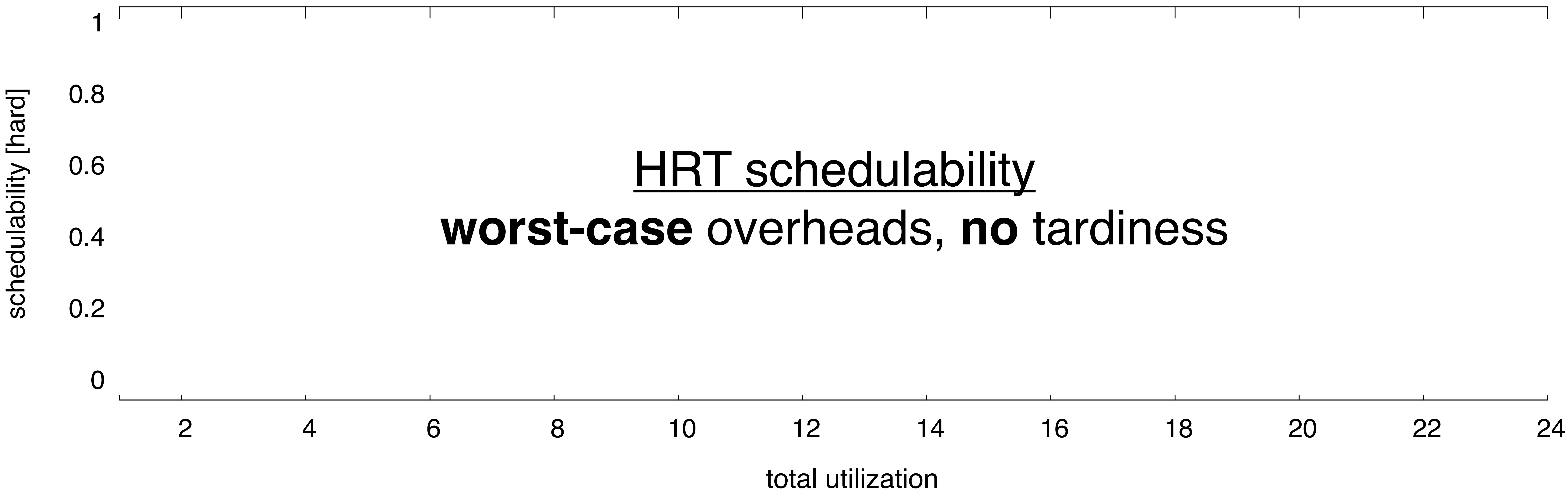
	My Work Lower Bound	My Work Bound / Protocol	Prior Work Bound / Protocol
Global	$\Omega(n)$	—	$\Omega(m \cdot n)$ /
Clustered	$\Omega(n)$		
Partitioned	$\Omega(n)$	$O(n)$ / FMLP+	$\Omega(m \cdot n)$ / MPCP $\Omega(m \cdot n)$ / DPCP

Next: overhead-aware schedulability study for non-asymptotic comparison.

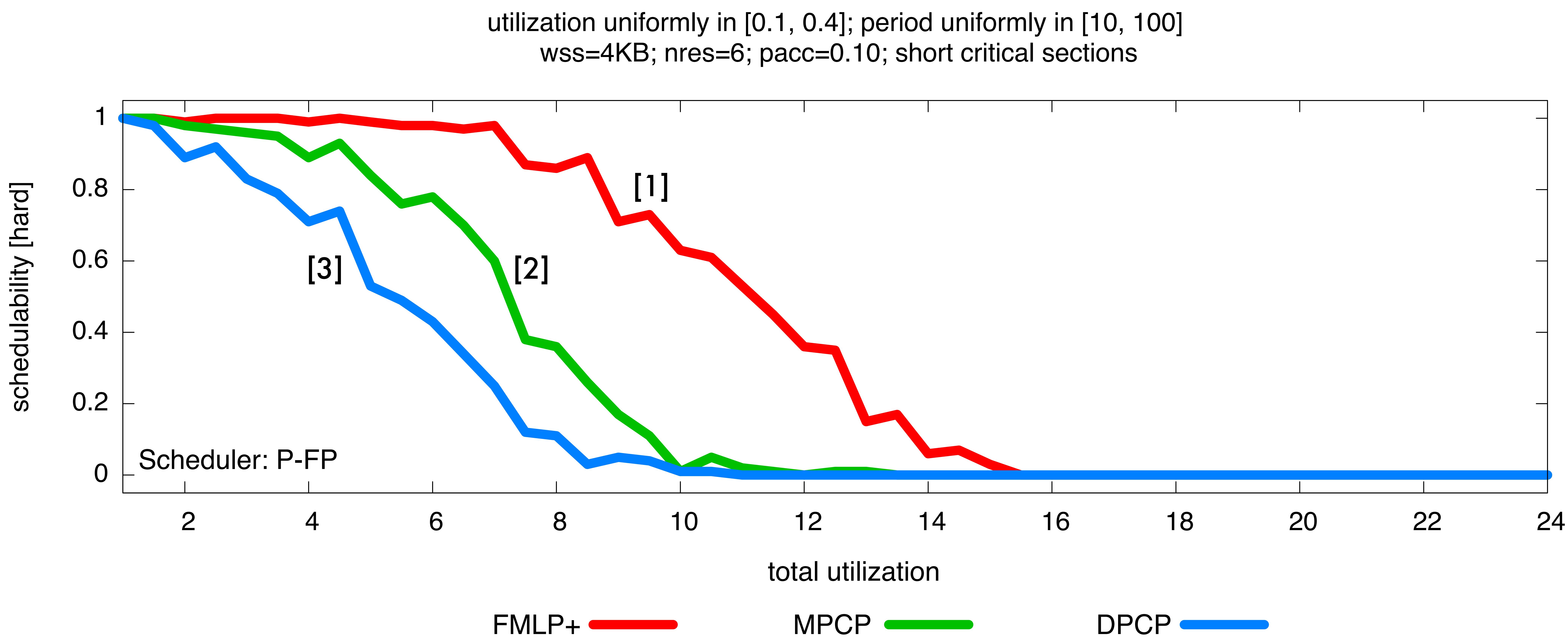
FMLP+ = *FIFO Mutex Locking Protocol*
MPCP = *Multiprocessor Priority Ceiling Protocol* (Rajkumar, 1990)
DPCP = *Distributed Priority Ceiling Protocol* (Rajkumar et al., 1988)

S-Aware Schedulability Comparison

same parameters as before

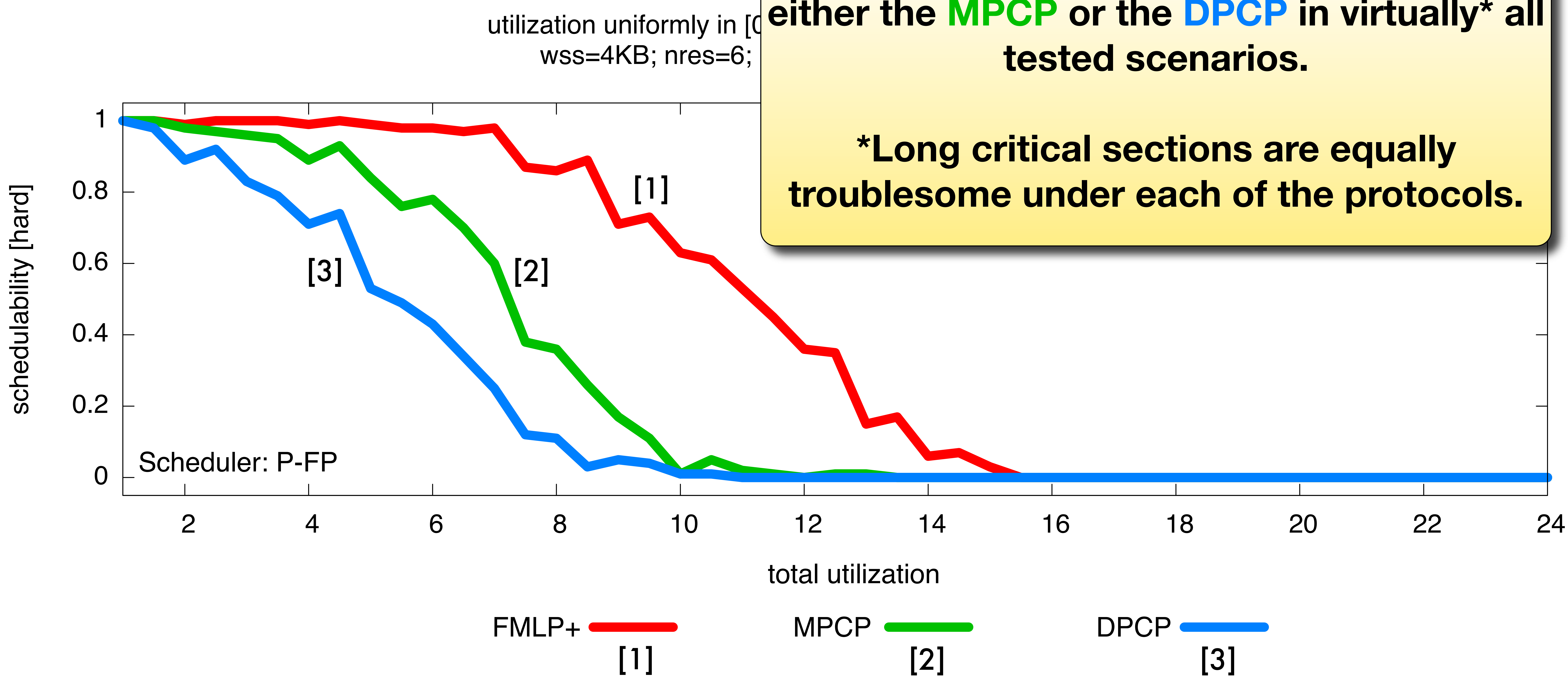


S-Aware Schedulability Comparison



[1]
[2]
[3]

S-Aware Schedulability Comparison



FMLP+ yields better schedulability than either the MPCP or the DPCP in virtually* all tested scenarios.

***Long critical sections are equally troublesome under each of the protocols.**

Part 2: Contributions

Concerning semaphore protocols.

➡ Notions of **blocking optimality**.

➡ Several **asymptotically optimal semaphore protocols**.

➡ These protocols **perform well in practice**.

Concerning spinlock protocols.

➡ Improved blocking analysis (very technical; not discussed).

➡ **Overhead-aware comparison** of semaphores and spinlocks in terms of schedulability.

Part 2: Contributions

Concerning semaphore protocols.

- ➡ Notions of **blocking optimality**.
- ➡ Several **asymptotically optimal semaphore protocols**.
- ➡ These protocols **perform well in practice**.

s-aware and **s-oblivious**

Concerning spinlock protocols.

- ➡ Improved blocking analysis (very technical; not discussed).
- ➡ **Overhead-aware comparison** of semaphores and spinlocks in terms of schedulability.

Part 2: Contributions

Concerning semaphore protocols.

- Notions o
- Several **asymptotically optimal semaphore protocols**.
- These protocols **perform well in practice**.

Three **OMLP** variants and the **FMLP+**.

Concerning spinlock protocols.

- Improved blocking analysis (very technical; not discussed).
- **Overhead-aware comparison** of semaphores and spinlocks in terms of schedulability.

Part 2: Contributions

Concerning semaphore protocols.

- ➡ Notions of **blocking optimality**.
- ➡ Achieve **higher schedulability** than "classic" protocols.
- ➡ These protocols **perform well in practice**.

Concerning spinlock protocols.

- ➡ Improved blocking analysis (very technical; not discussed).
- ➡ **Overhead-aware comparison** of semaphores and spinlocks in terms of schedulability.

Part 2: Contributions

Next: brief look at spinlocks.

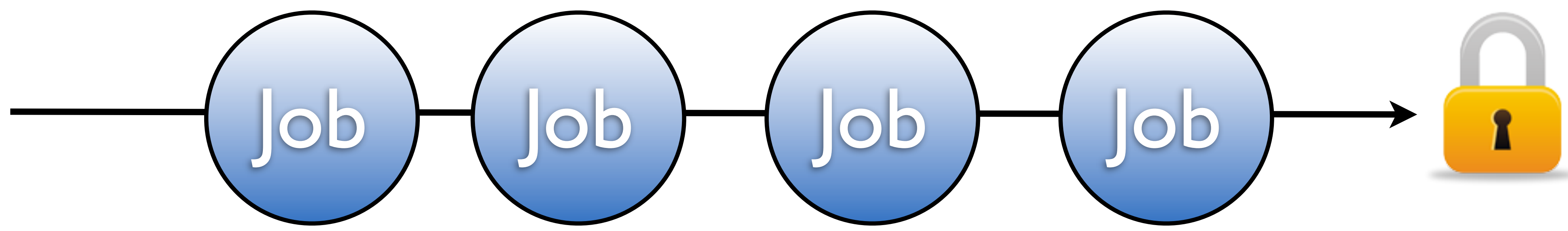
Concerning semaphore pr

- ➡ Notions of **blocking optim**
- ➡ Several **asymptotically optimal semaphore protocols**.
- ➡ These protocols **perform well in practice**.

Concerning spinlock protocols.

- ➡ Improved blocking analysis (very technical; not discussed).
- ➡ **Overhead-aware comparison** of semaphores and spinlocks in terms of schedulability.

Non-Preemptive Task-Fair Queue Lock



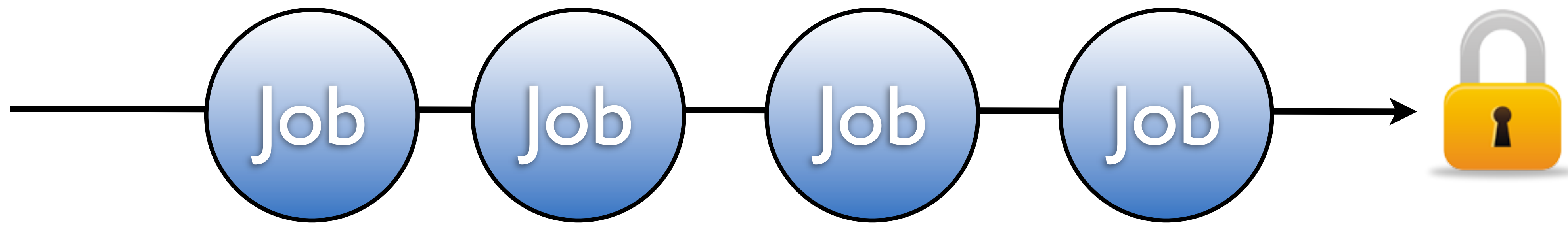
Non-Preemptive

Jobs cannot be preempted while **spinning** or executing their **critical section**.

Task-Fair Queue Lock

Waiting jobs form a **FIFO spin queue**.

Non-Preemptive Task-Fair Queue Lock



Advantages:

low overheads, no analysis of suspensions required.

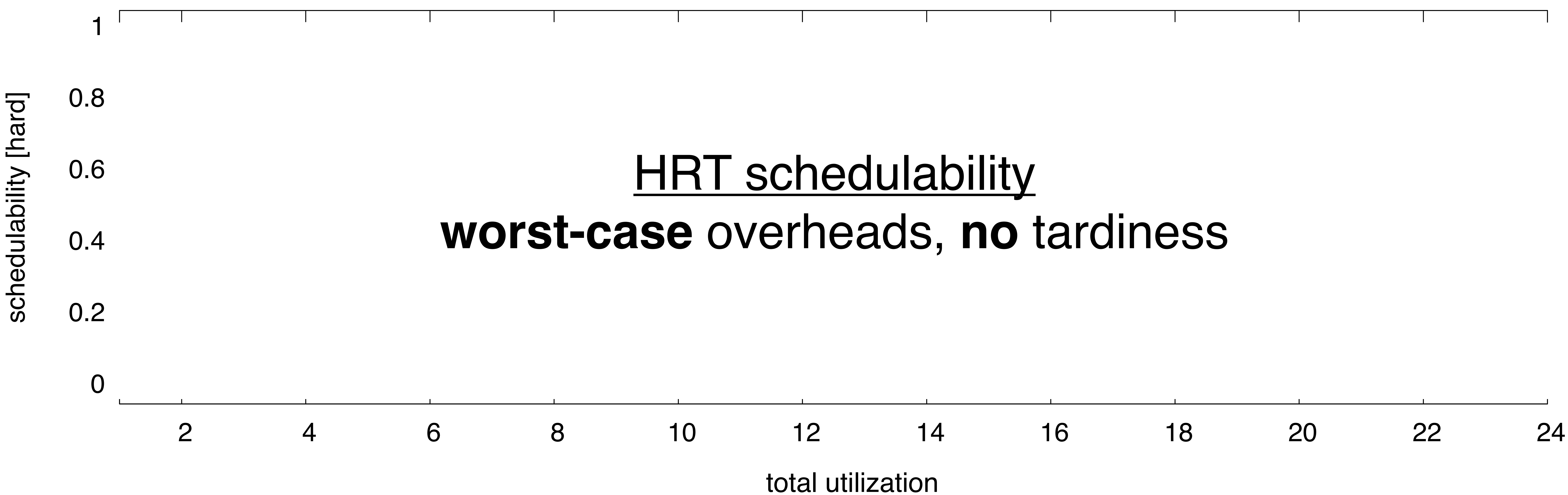
Disadvantages:

**waste processor cycles,
non-preemptivity can be problematic.**

Resource-Sharing Parameters

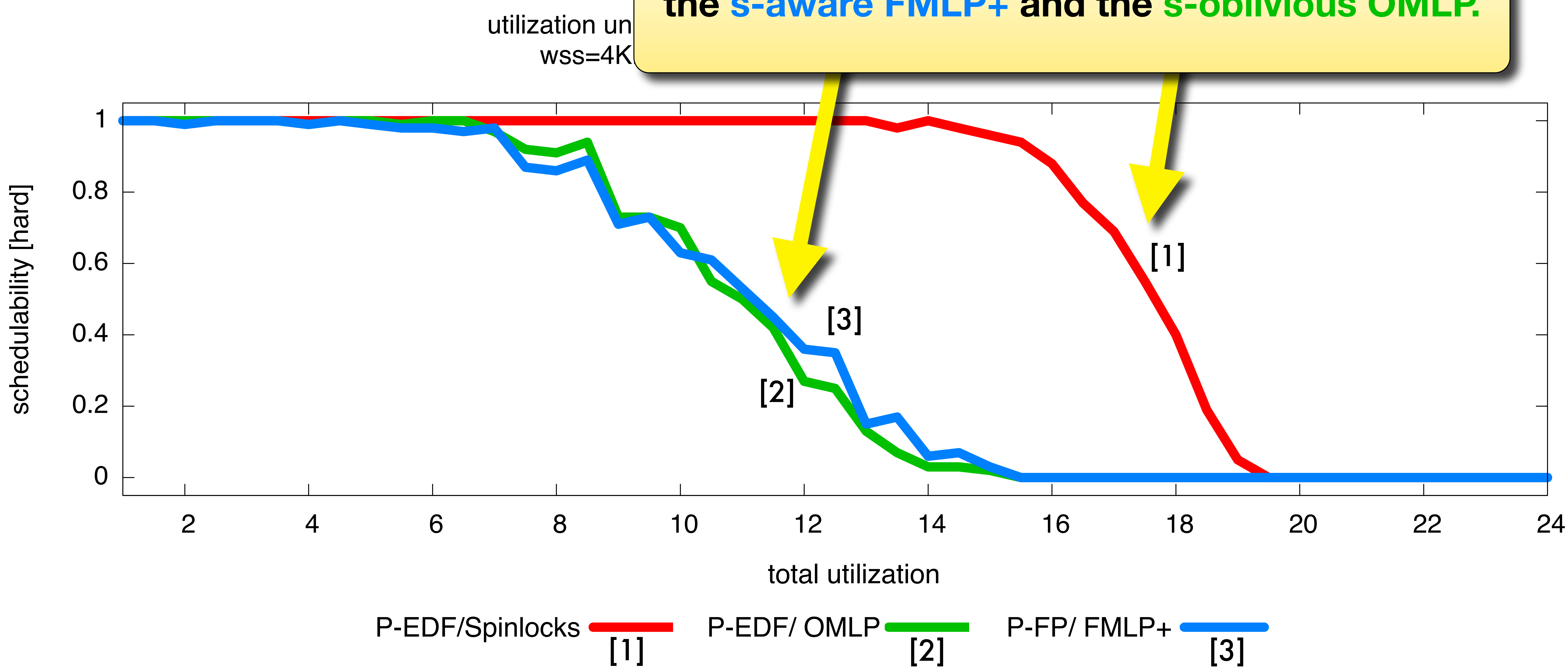
	In this talk	In my dissertation
Number of resources	6	1, 3, 6, 12, 24
Access probability	25%	10%, 25%, 40%, 55%, 70%, 85%
Critical Section Lengths	uniformly in [1, 15] μs	short: [1, 15] μs medium: [1, 100] μs long: [5, 1280] μs

S-Oblivious vs. S-Aware vs. Spinlocks



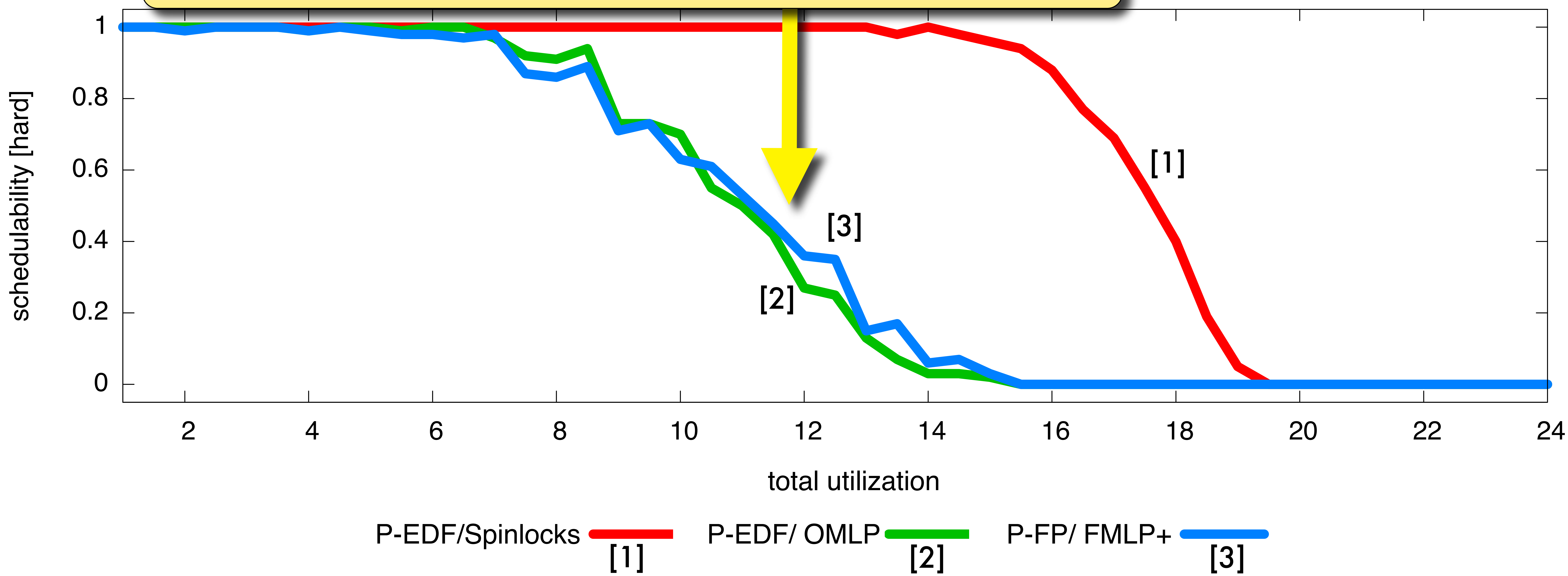
S-Oblivious vs. S-Aware vs. Spinlocks

Spinlocks improve schedulability compared to the s-aware FMLP+ and the s-oblivious OMLP.

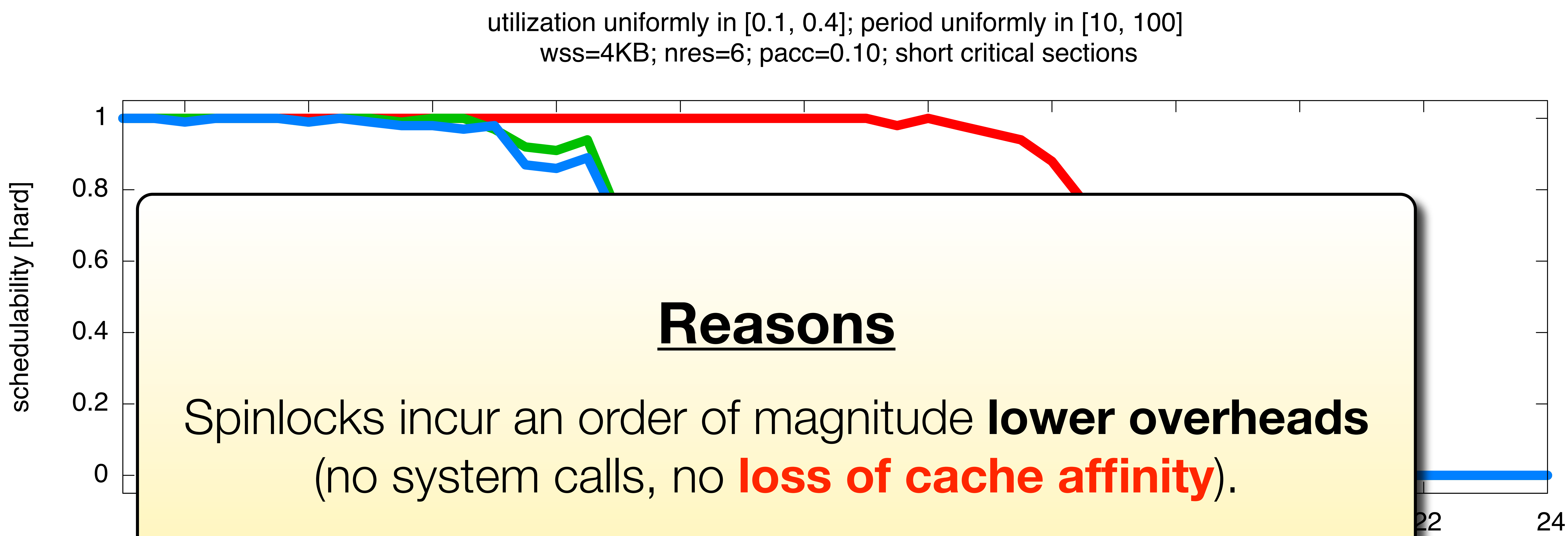


S-Oblivious vs. S-Aware vs. Spinlocks

Surprise: the s-aware protocol (FMLP+) is not much better then the best best s-oblivious protocol (OMLP).



S-Oblivious vs. S-Aware vs. Spinlocks



Reasons

Spinlocks incur an order of magnitude **lower overheads** (no system calls, no **loss of cache affinity**).

Analysis of suspensions is **very pessimistic**.

Existing s-aware analysis is **not much more precise** than the **much simpler s-oblivious approach**.

Part 2: Contributions

Concerning semaphore protocols.

- Notion of **blocking optimality**.
- Several **asymptotically optimal semaphore protocols**.
- These protocols **perform well in practice**.

Concerning spinlock protocols.

- Improved blocking analysis (very technical; not discussed).
- **Overhead-aware comparison** of semaphores and spinlocks in terms of schedulability.

Part 2: Contributions

Concerning semaphore protocols.

- ➡ Notion of **blocking optimality**.
- ➡ Several **asymptotically optimal semaphore protocols**.
- ➡ These protocols **perform well in practice**.

Concerning spinlock protocols.

- ➡ Improved blocking analysis (very technical; not discussed).
- ➡ **Overhead-aware comparison** of semaphores and

Use non-preemptive task-fair spinlocks in practice!

Part 3

Reader-Writer Exclusion

Reader-Writer (RW) Exclusion

(Courtois et al., 1971)

Readers

➡Only observe state of shared resource.

➡May **access** resource **concurrently** with other readers.

Writers

➡May modify state of shared resource.

➡Require **exclusive** access.

Courtois, P., Heymans, F., and Parnas, D. (1971). Concurrent control with “readers” and “writers”. Communications of the ACM, 14(10):667–668.

Reader-Writer (RW) Exclusion

(Courtois et al., 1971)

Readers

➡Only observe state of shared resource.

➡May **access** resource **concurrently** with other readers.

Writers

➡May modify state of shared resource.

➡Require **exclusive** access.

My contributions:

First analysis of RW locks in the context of multiprocessor real-time systems.

A new type of RW lock: phase-fair RW locks.

Courtois, P., Heymans, F., and Parnas, D. (1971). Concurrent control with “readers” and “writers”. Communications of the ACM, 14(10):667–668.

Prior Work: RW Lock Choices

How to order conflicting reads and writes?

RW Lock Type	Wort-case improvement?	Blocking analysis available?
Writer-Preference	?	?
Reader-Preference	?	?
Task-Fair	?	?
Other	?	?

Prior Work: RW Lock Choices

How to order conflicting reads and writes?

RW Lock Type	Wort-case improvement?	Blocking analysis available?
Writer-Preference	?	✗
Reader-Preference	?	✗
Task-Fair	?	✗
Other	?	✗

Prior Work: RW Lock Choices

How to order conflicting reads and writes?

RW Lock Type	Wort-case improvement?	Blocking analysis available?
Writer-Preference	?	✗
Reader-Preference	?	✗
Task-Fair	?	✗
Other	?	✗

No strong progress guarantees—ordering is HW dependent.

Prior Work: RW Lock Choices

How to order conflicting reads and writes?

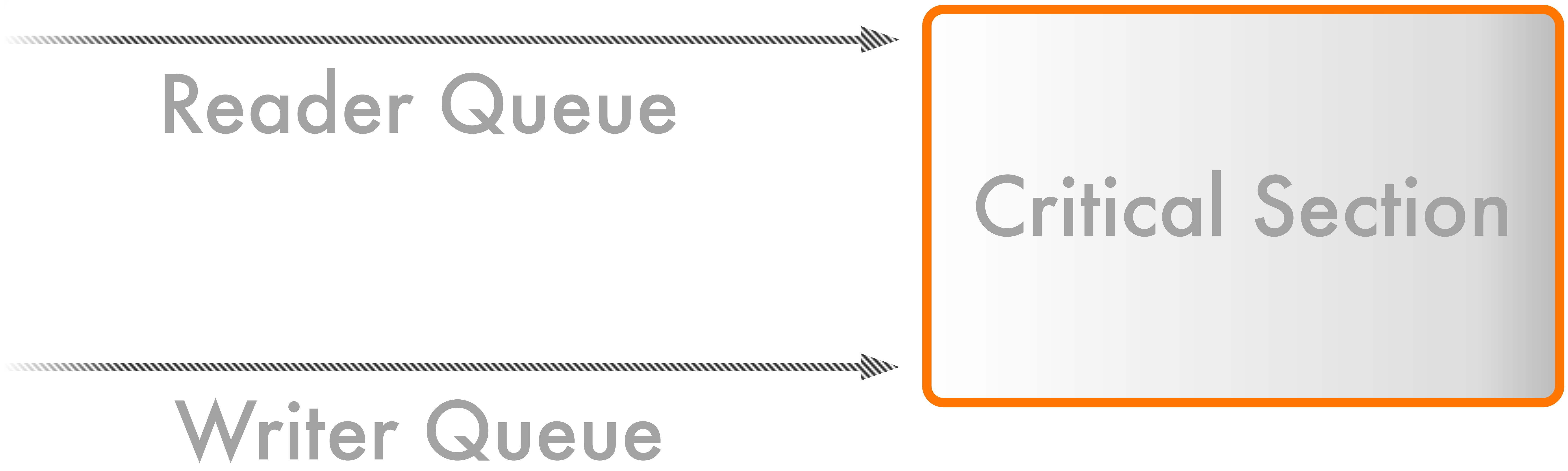
RW Lock Type	Wort-case improvement?	Blocking analysis available?
Writer-Preference	?	✗
Reader-Preference	?	✗
Task-Fair	?	✗
Other	✗	✗

Let's look at Writer-Preference RW Locks...

Writer-Preference RW Lock

i. **Readers** wait if **writers** are present.

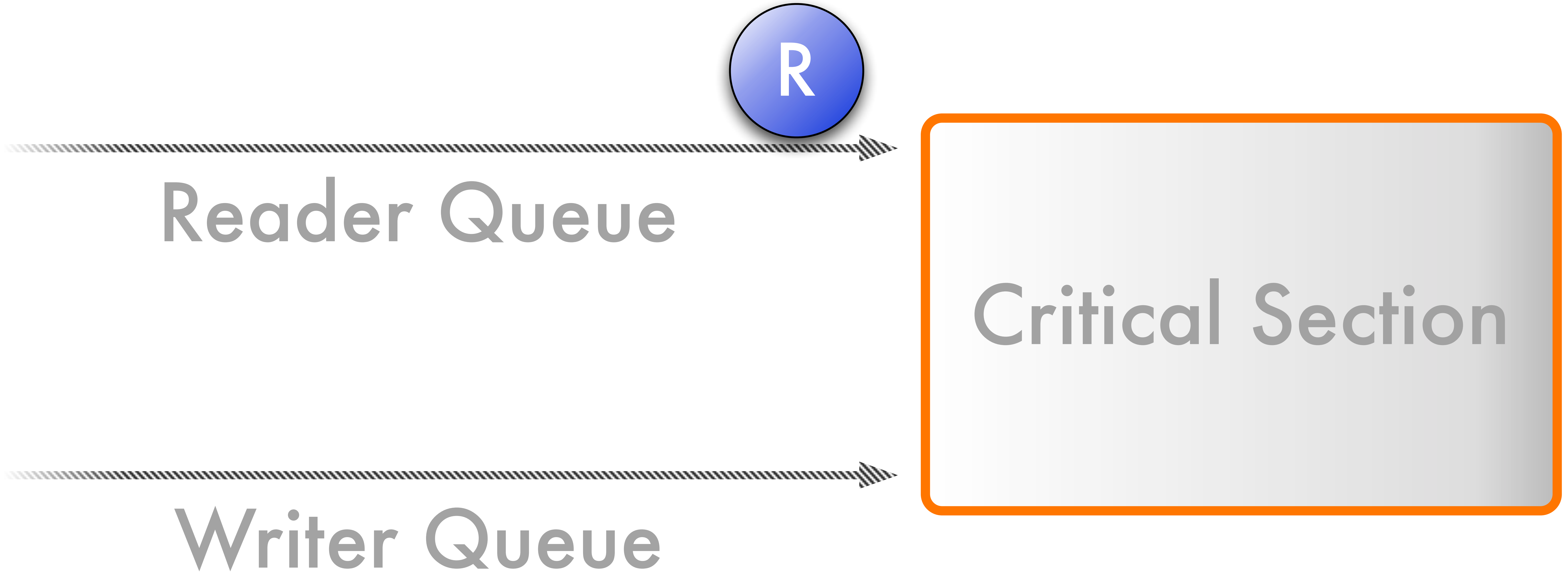
ii. **Writers** enter in **FIFO** order.



Writer-Preference RW Lock

i. **Readers** wait if **writers** are present.

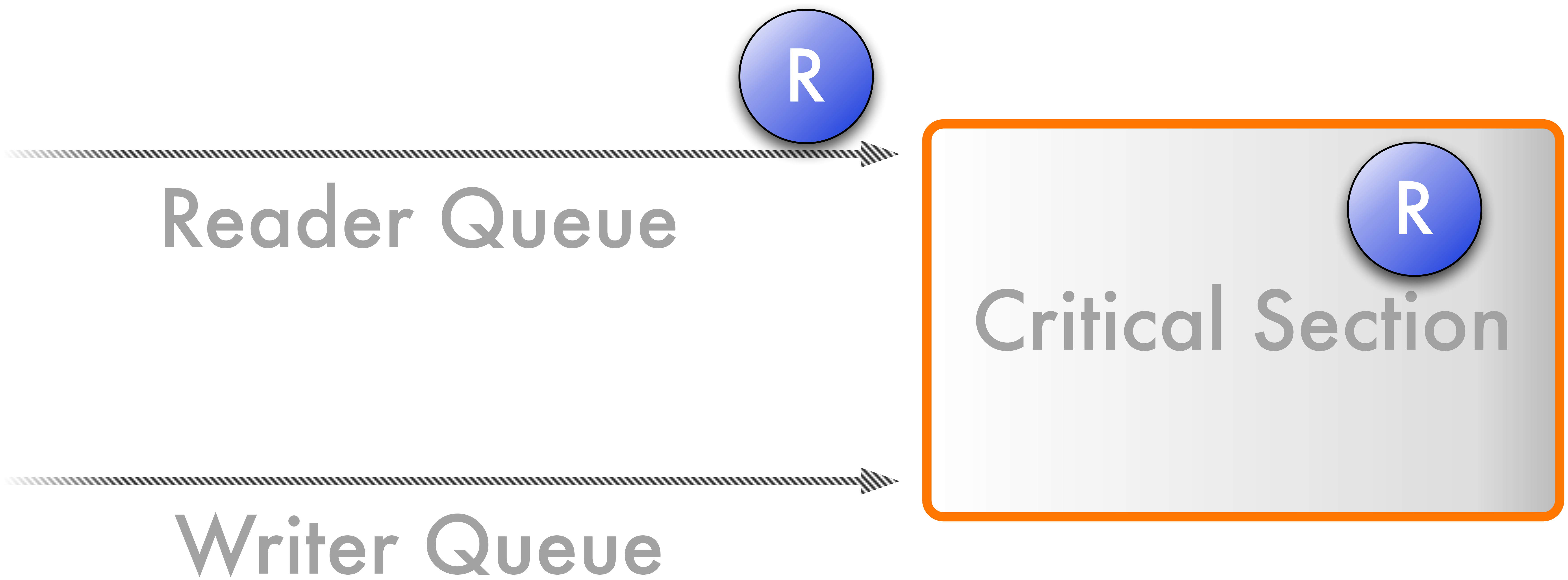
ii. **Writers** enter in **FIFO** order.



Writer-Preference RW Lock

i. **Readers** wait if **writers** are present.

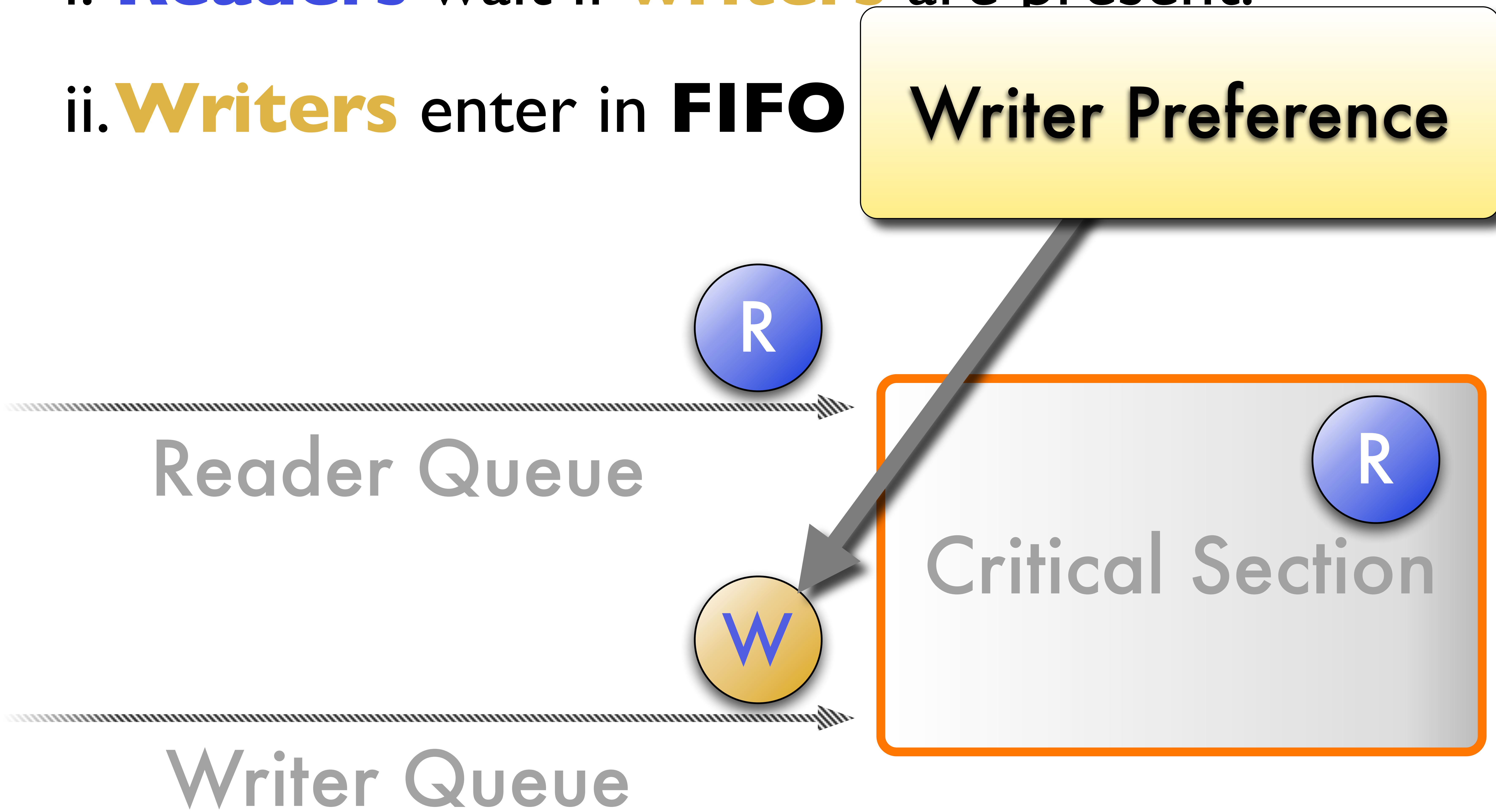
ii. **Writers** enter in **FIFO** order.



Writer-Preference RW Lock

i. **Readers** wait if **writers** are present.

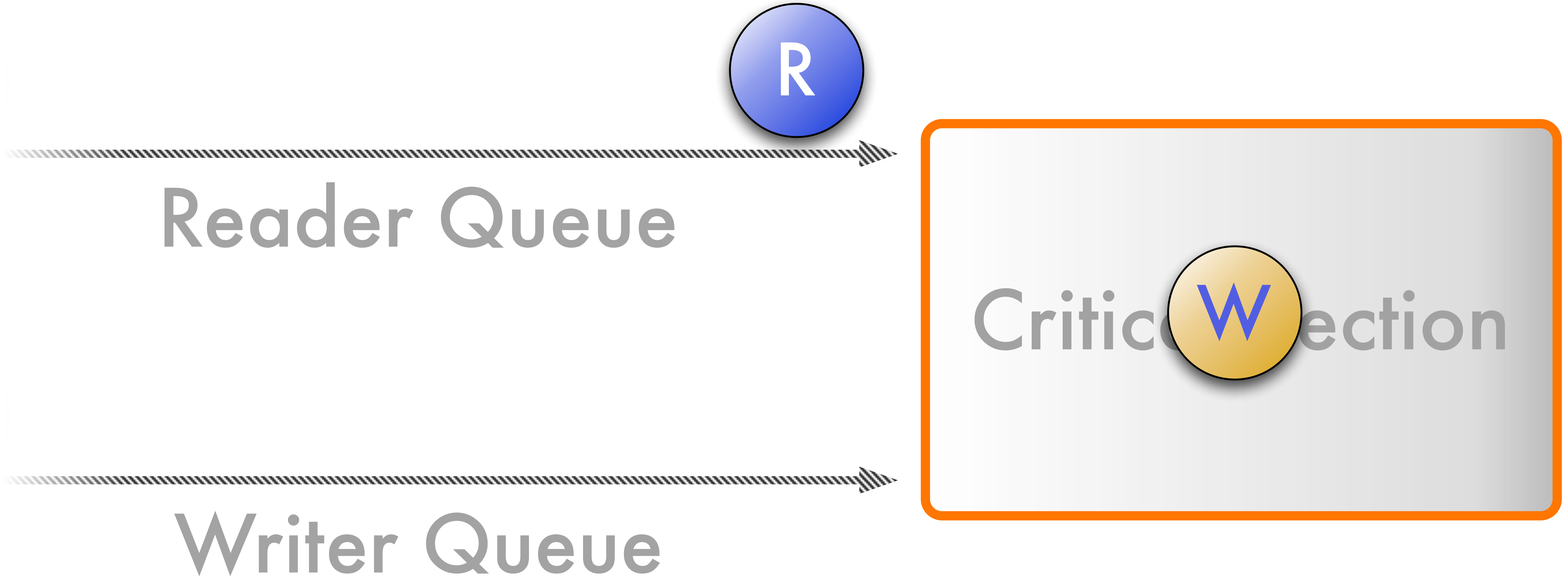
ii. **Writers** enter in **FIFO**



Writer-Preference RW Lock

i. **Readers** wait if **writers** are present.

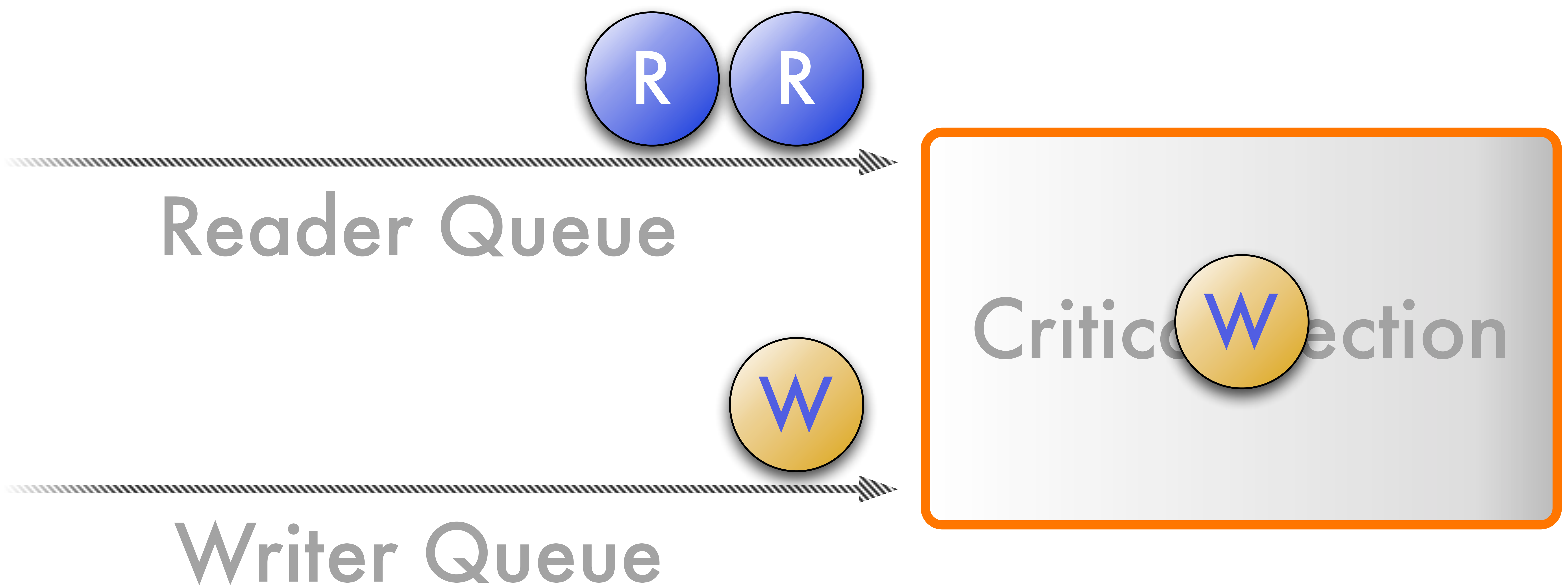
ii. **Writers** enter in **FIFO** order.



Writer-Preference RW Lock

i. **Readers** wait if **writers** are present.

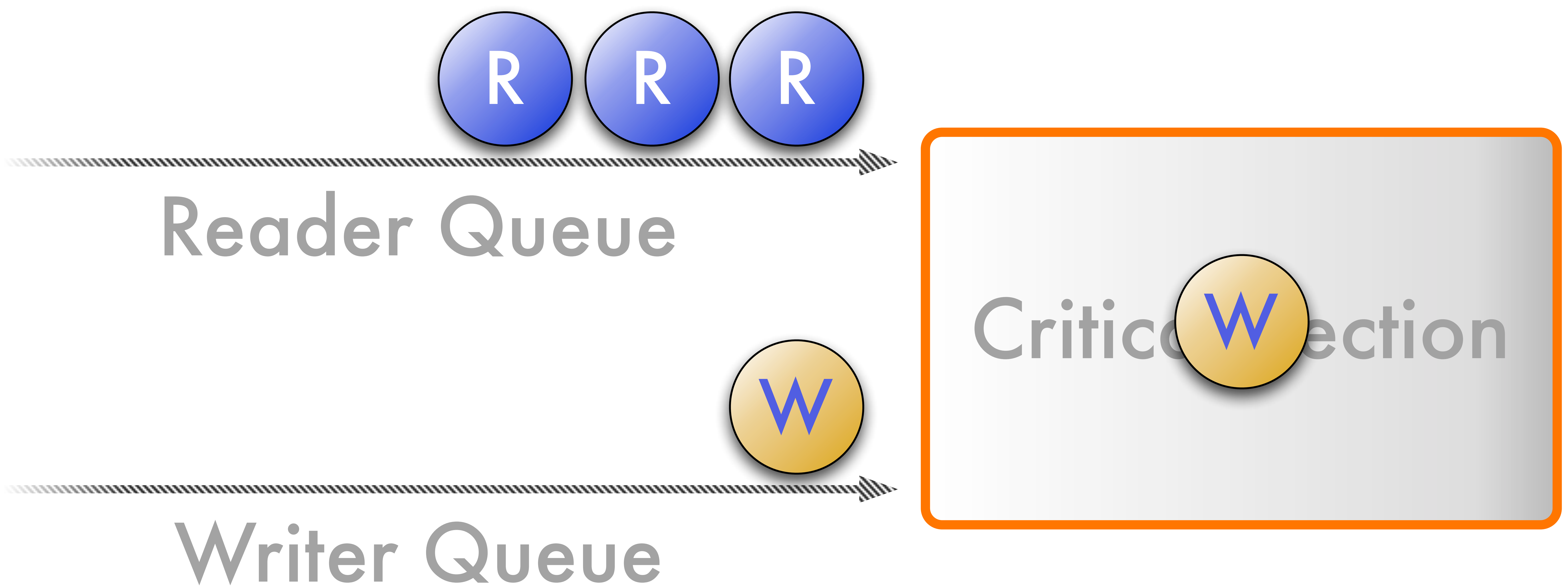
ii. **Writers** enter in **FIFO** order.



Writer-Preference RW Lock

i. **Readers** wait if **writers** are present.

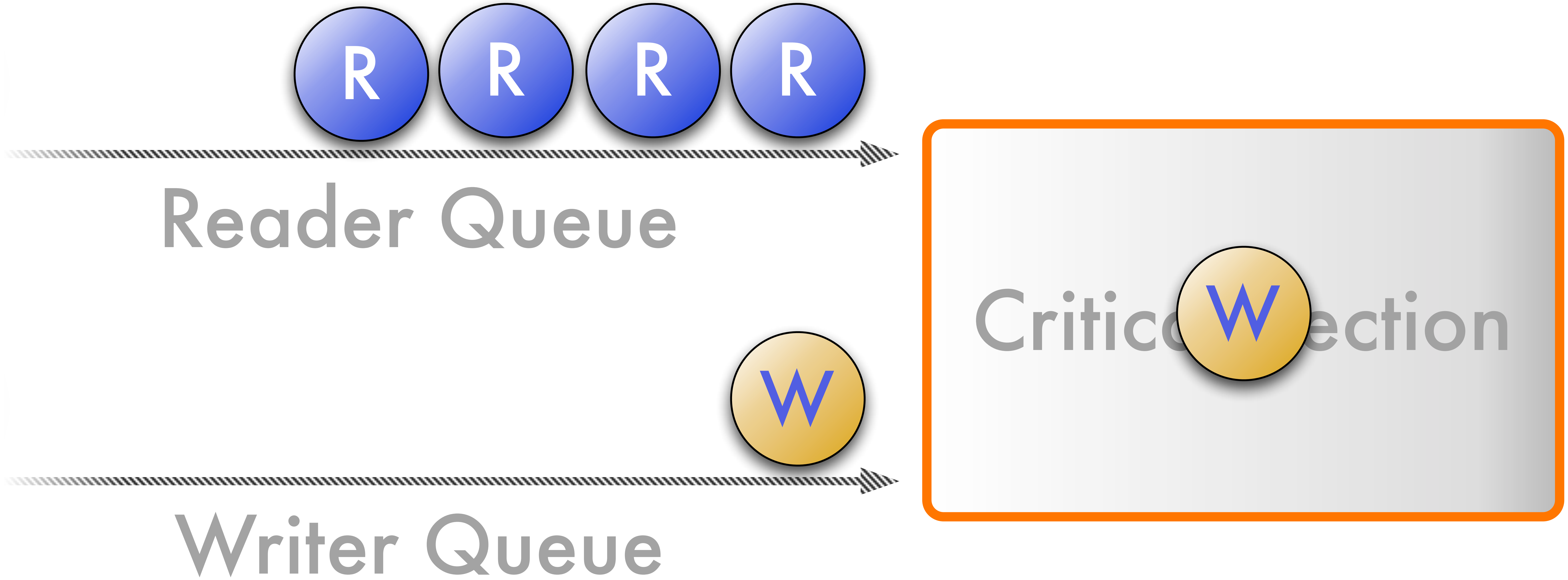
ii. **Writers** enter in **FIFO** order.



Writer-Preference RW Lock

i. **Readers** wait if **writers** are present.

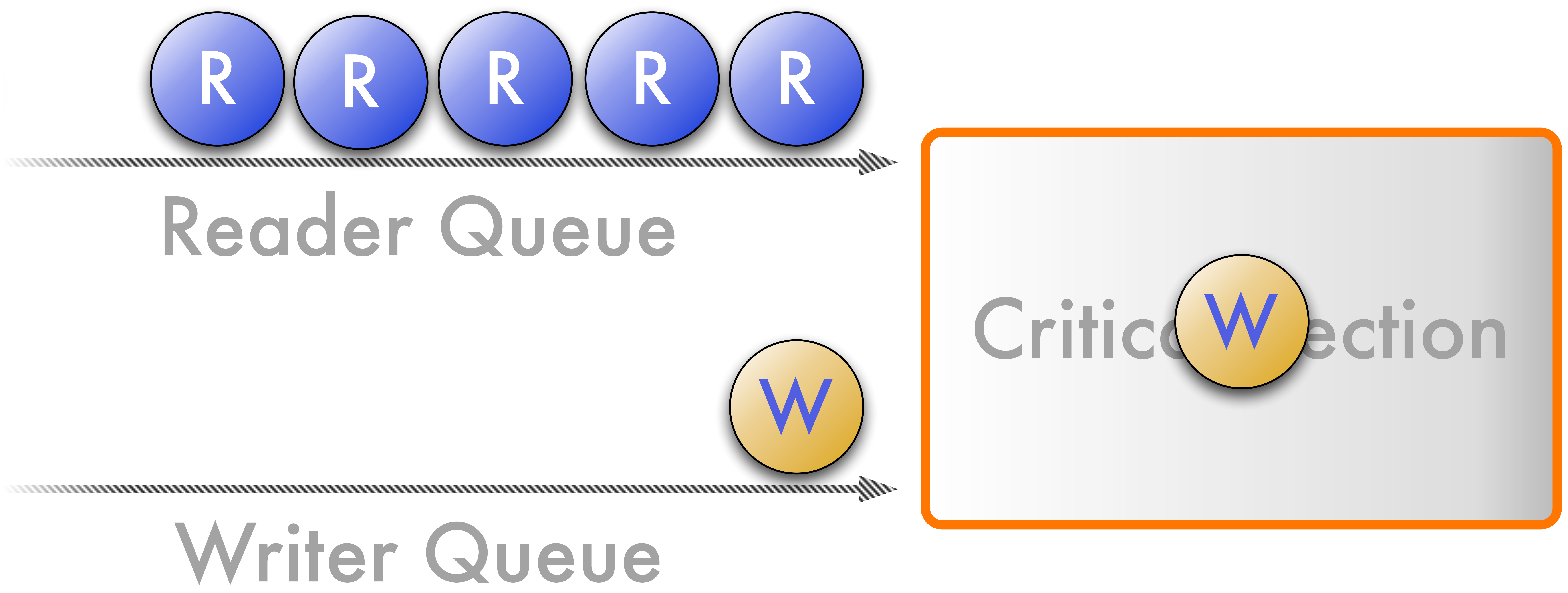
ii. **Writers** enter in **FIFO** order.



Writer-Preference RW Lock

i. **Readers** wait if **writers** are present.

ii. **Writers** enter in **FIFO** order.

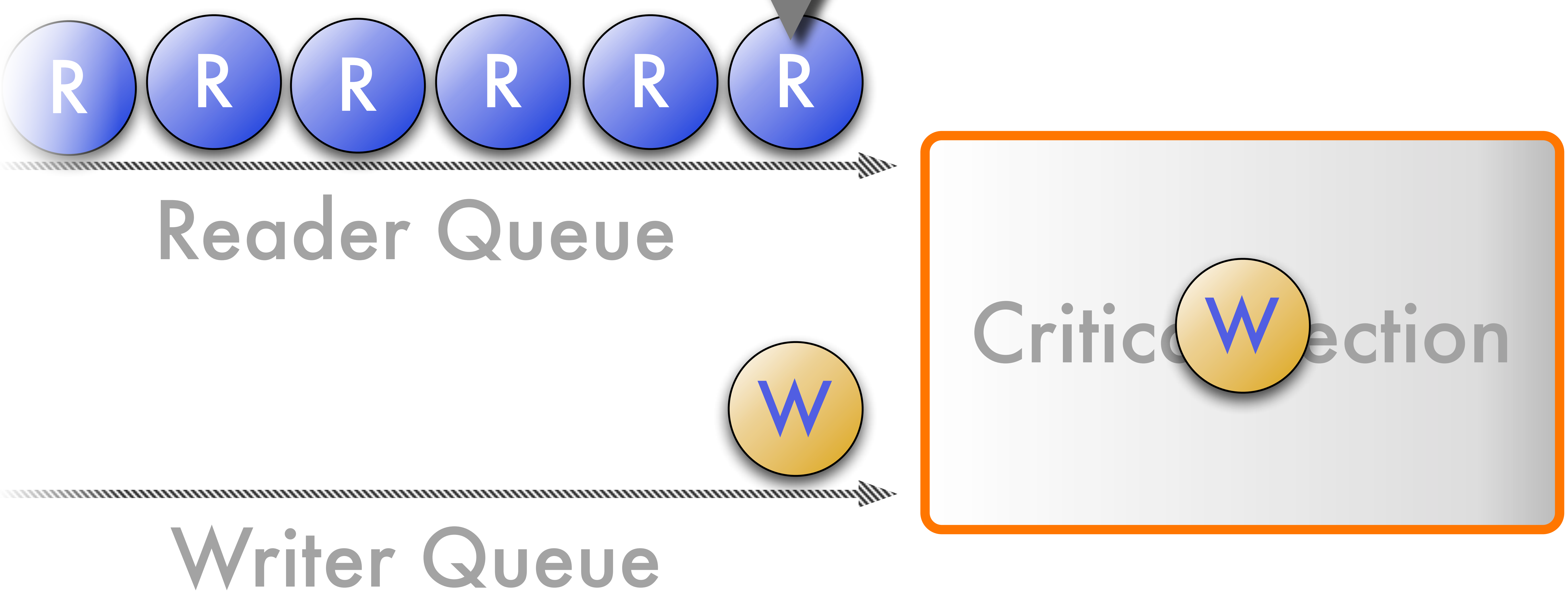


Writer-Preference RW Lock

i. **Readers** sent.

ii. **Writers** enter in FI FO order.

Starvation!



Prior Work: RW Lock Choices

How to order conflicting reads and writes?

RW Lock Type	Wort-case improvement?	Blocking analysis available?
Writer-Preference	✗	✗
Reader-Preference	?	✗
Task-Fair	?	✗
Other	✗	✗

Also allows starvation!

Prior Work: RW Lock Choices

How to order conflicting reads and writes?

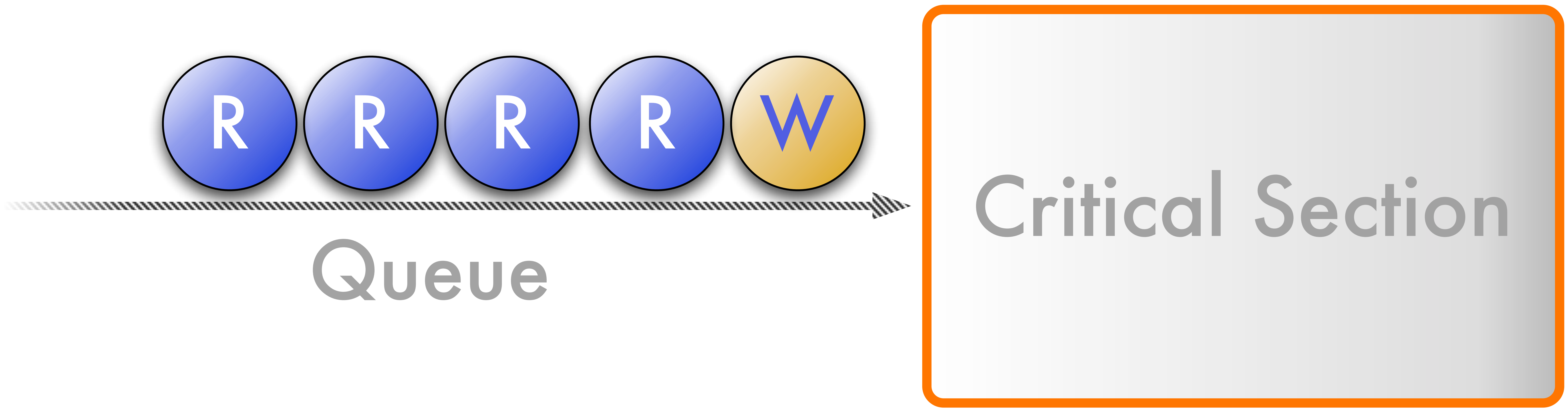
RW Lock Type	Wort-case improvement?	Blocking analysis available?
Writer-Preference	✗	✗
Reader-Preference	✗	✗
Task-Fair	?	✗
Other	✗	✗

Let's look at Task-Fair RW Locks...

Task-Fair RW Lock

i. **Readers** and **writers** both enter in **FIFO** order.

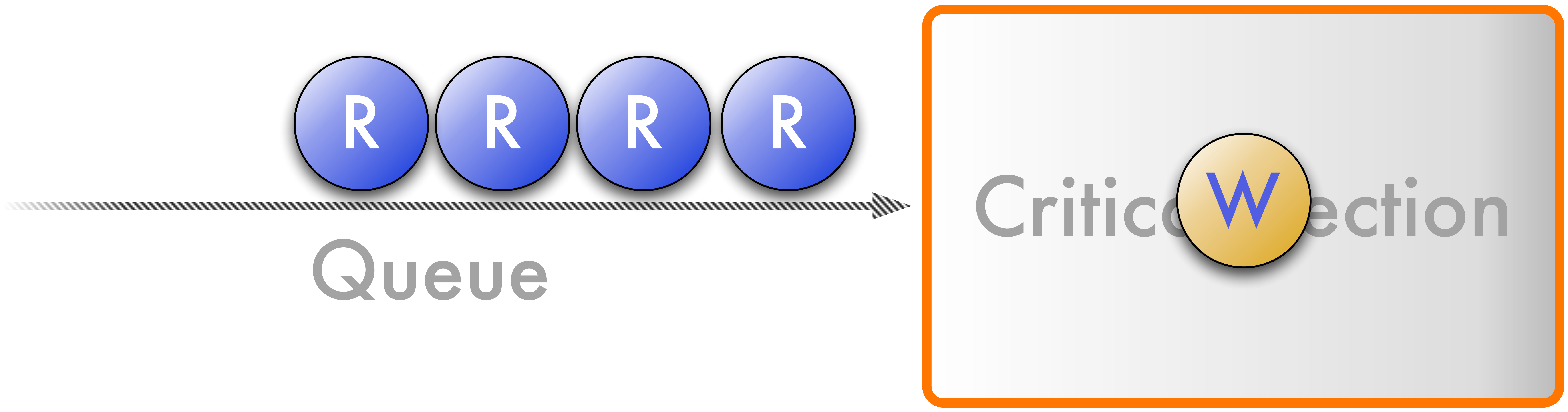
ii. Consecutive **readers** enter together.



Task-Fair RW Lock

i. **Readers** and **writers** both enter in **FIFO** order.

ii. Consecutive **readers** enter together.

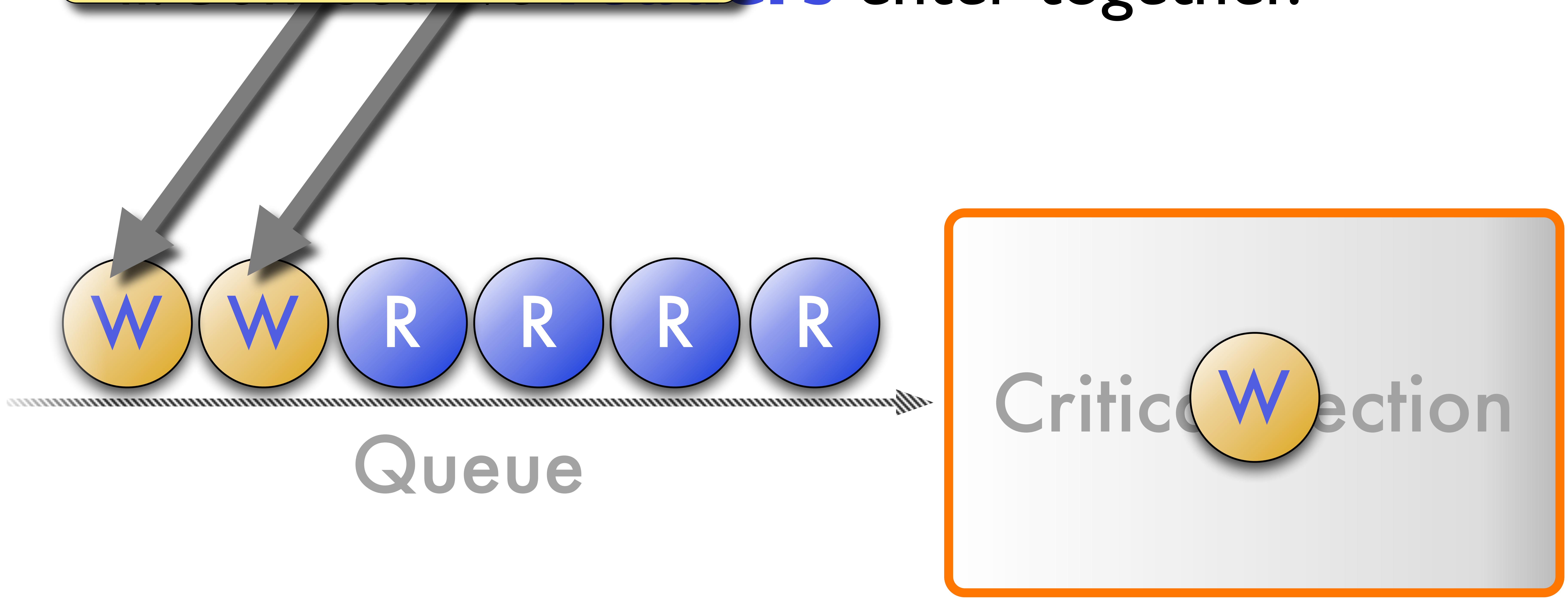


Task-Fair RW Lock

i. **Readers** and **writers** both enter in **FIFO**

Later Writer Arrival

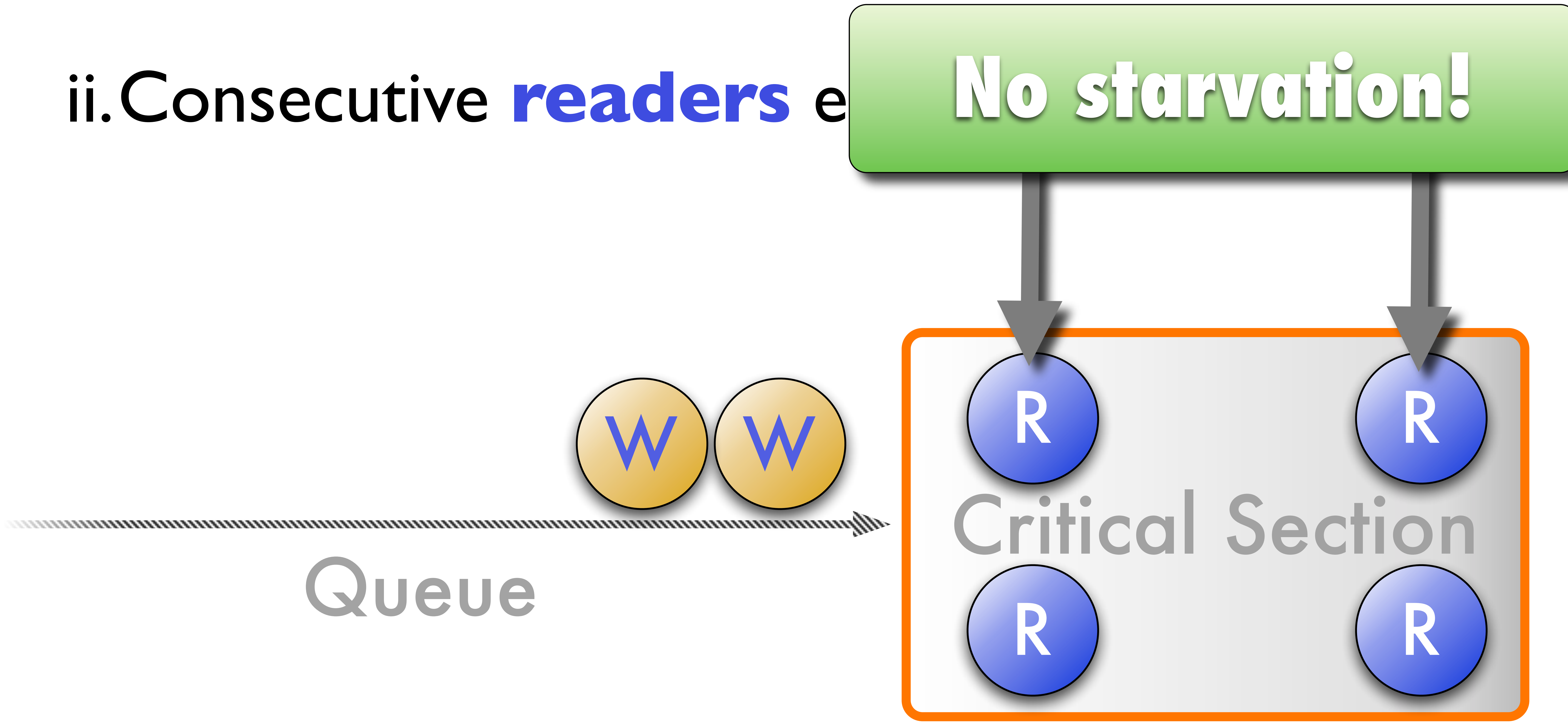
...ers enter together.



Task-Fair RW Lock

i. **Readers** and **writers** both enter in **FIFO** order.

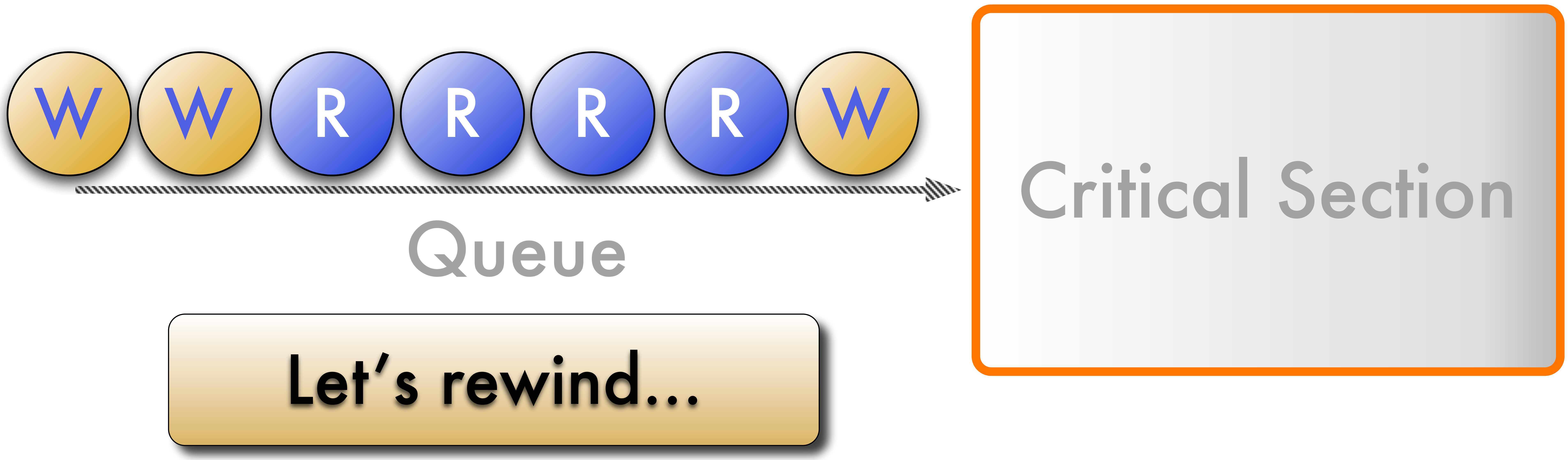
ii. Consecutive **readers** e



Task-Fair RW Lock

i. **Readers** and **writers** both enter in **FIFO** order.

ii. Consecutive **readers** enter together.

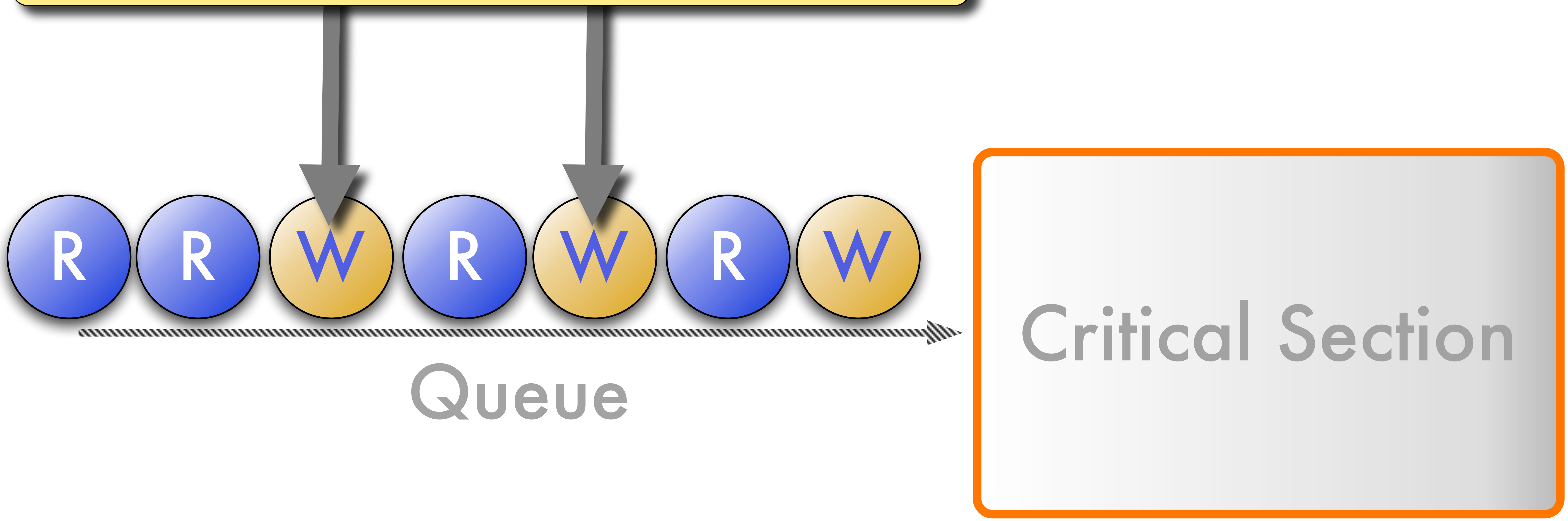


Task-Fair RW Lock

i. **Readers** and **writers** both enter in **FIFO** order.

ter together.

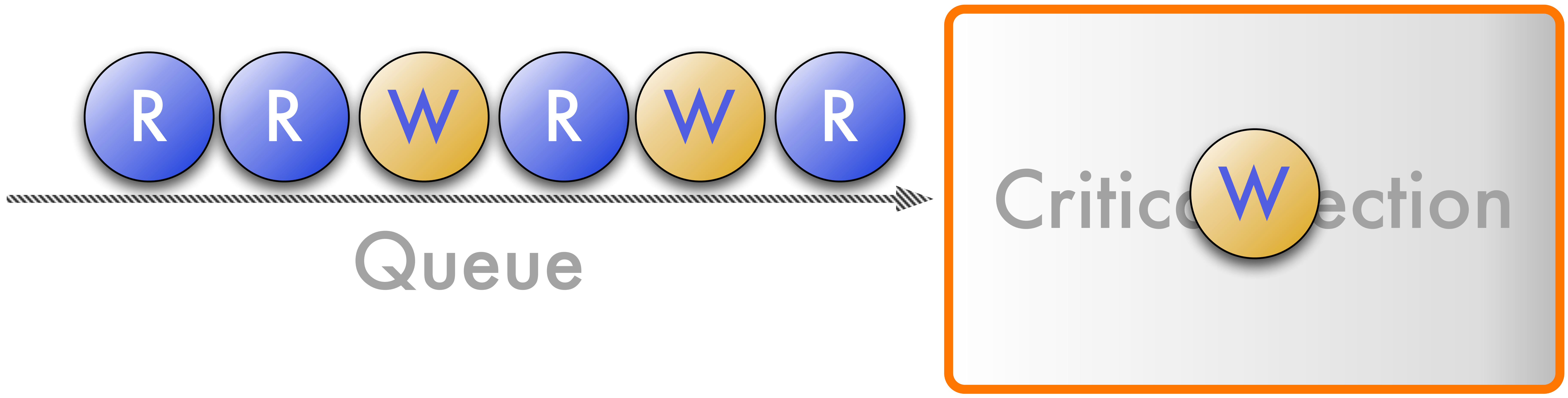
Change in arrival sequence.



Task-Fair RW Lock

i. **Readers** and **writers** both enter in **FIFO** order.

ii. Consecutive **readers** enter together.

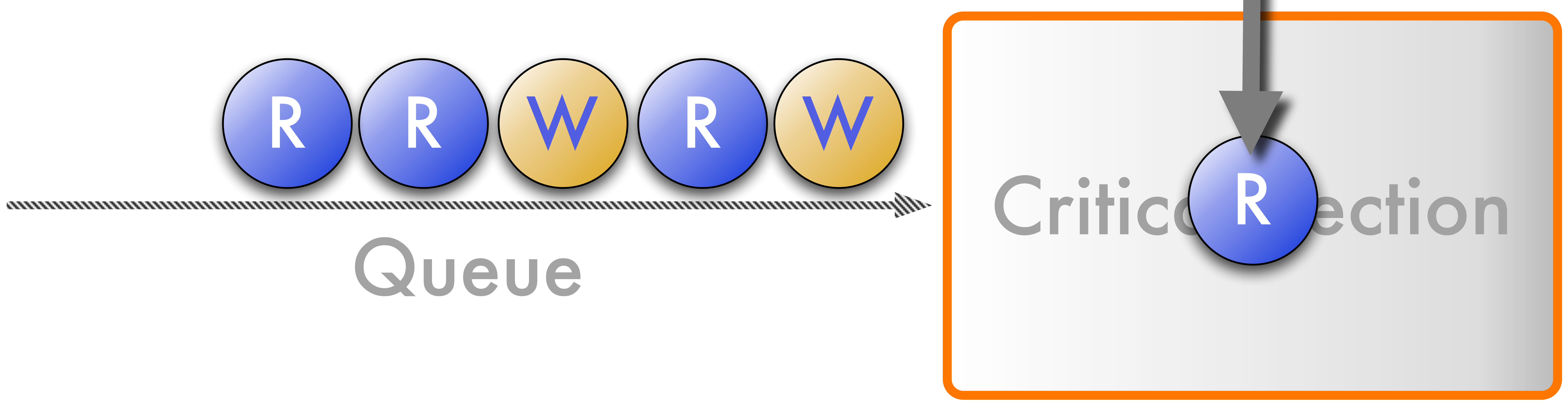


Task-Fair RW Lock

i. **Readers** and **writers** both enter in **FIFO** order.

ii. Consecutive **reader**

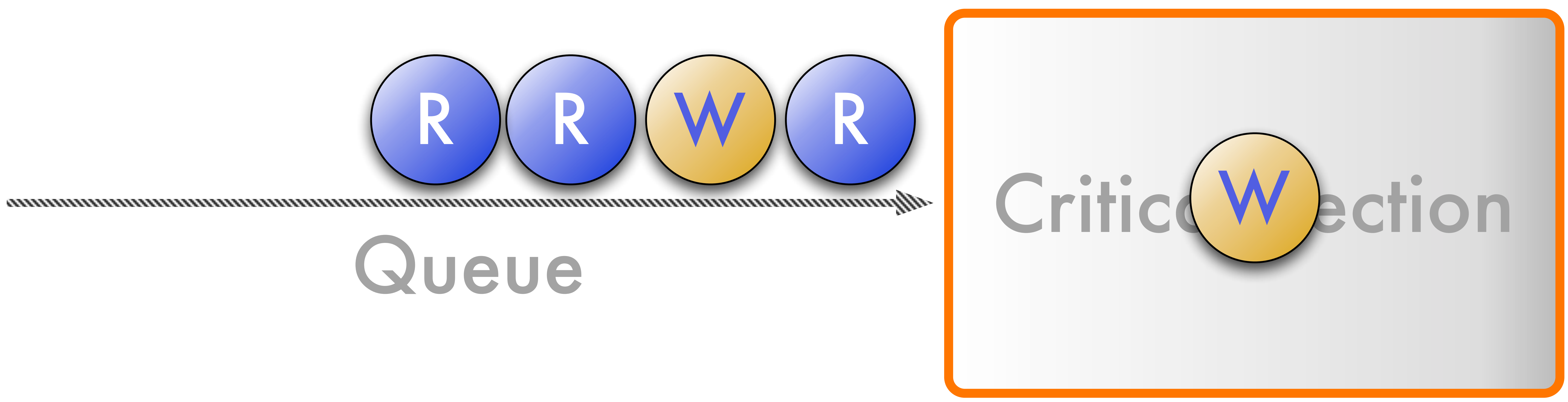
Only single reader enters!



Task-Fair RW Lock

i. **Readers** and **writers** both enter in **FIFO** order.

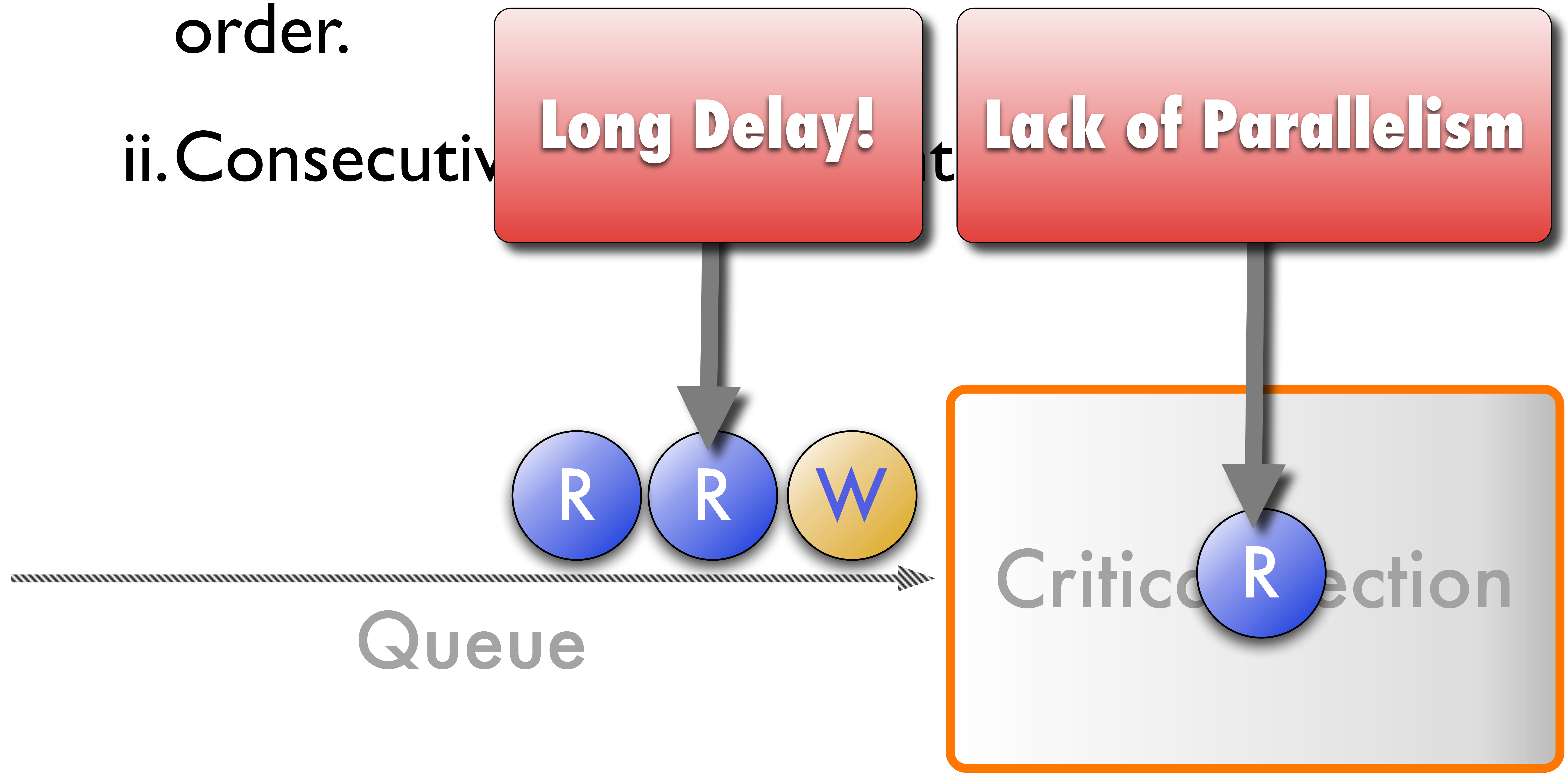
ii. Consecutive **readers** enter together.



Task-Fair RW Lock

i. **Readers** and **writers** both enter in **FIFO** order.

ii. Consecutiv



Prior Work: RW Lock Choices

How to order conflicting reads and writes?

RW Lock Type	Wort-case improvement?	Blocking analysis available?
Writer-Preference	✗	✗
Reader-Preference	✗	✗
Task-Fair	!	✗
Other	✗	✗

Can be analyzed, but **worst case** similar to mutex.

My contribution:

A new type of RW lock with analytical worst-case improvement.

RW Lock Type	Wort-case improvement?	Blocking analysis available?
Writer-Preference	✗	✗
Reader-Preference	✗	✗
Task-Fair	!	✗
Other	✗	✗

Design Space

esign Spac

Allows starvation!
=
Not "fair" enough!

Lack of parallelism!
=
Too "fair"!

Preference Locks

Task-Fair Locks

Increasing "fairness"

Design Space

Increasing "fairness"

A New Type of RW Lock

Phase-Fair
Reader-Writer Locks

"Polite" Readers and Writers

Phase-Fair Reader-Writer Locks

Readers give preference to **writers**.
Writers give preference to **readers**.

"Please, after you…"

Phase-Fairness

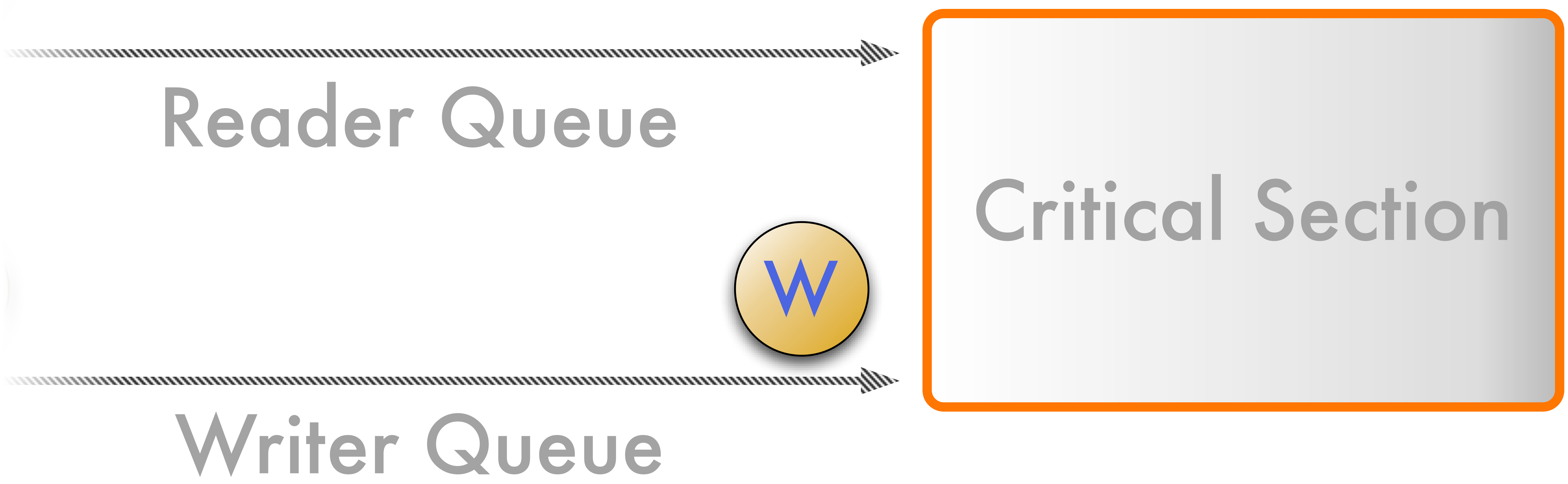
(paraphrased)

All **readers** enter when unblocked by an **exiting writer** (unless there are no writers).

A **writer** enters when unblocked by the **last exiting reader** (unless there are no writers).

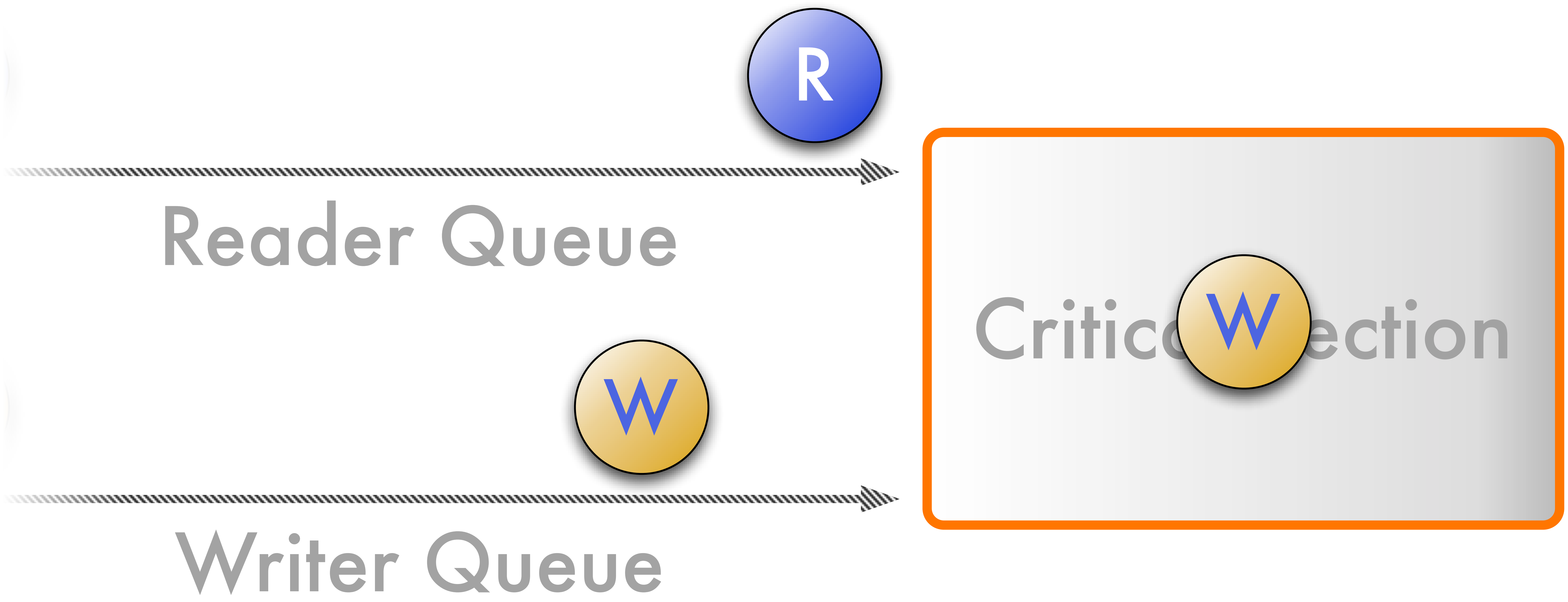
Effect: **reader phases** and **writer phases** alternate.

Phase-Fair RW Lock



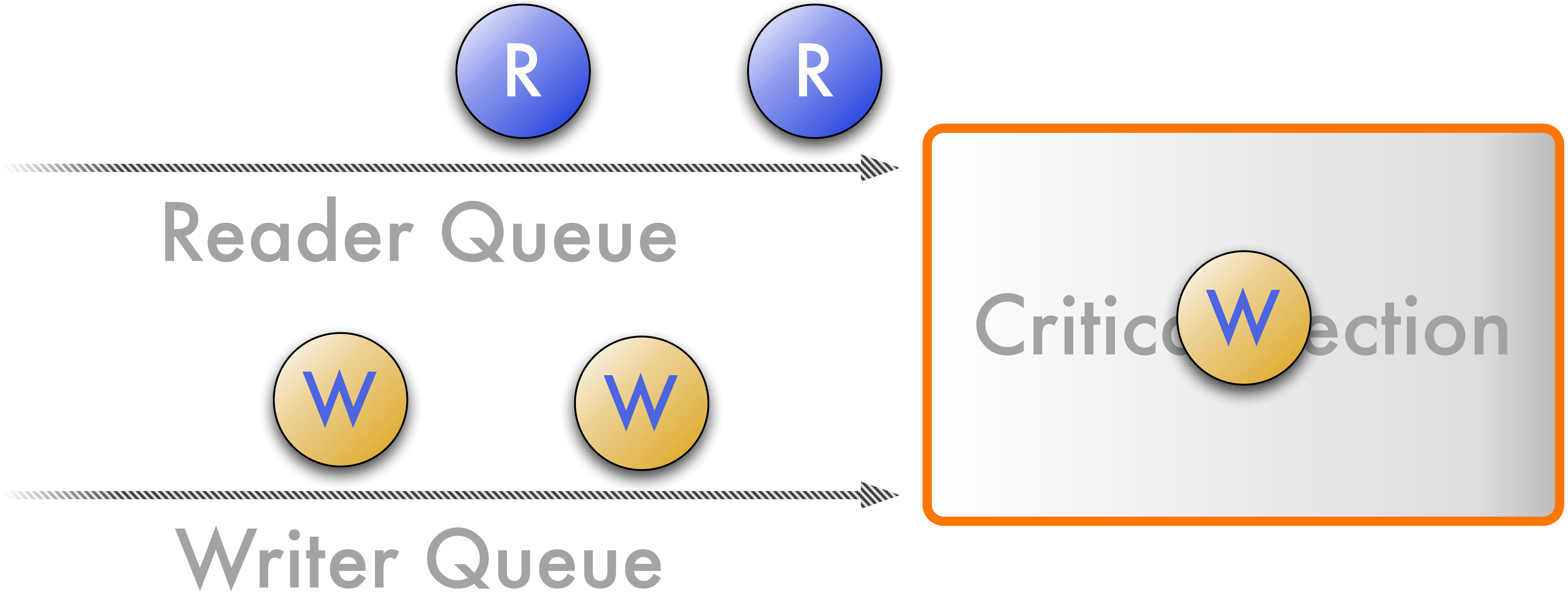
Phase-Fair RW Lock

staggering indicates arrival order



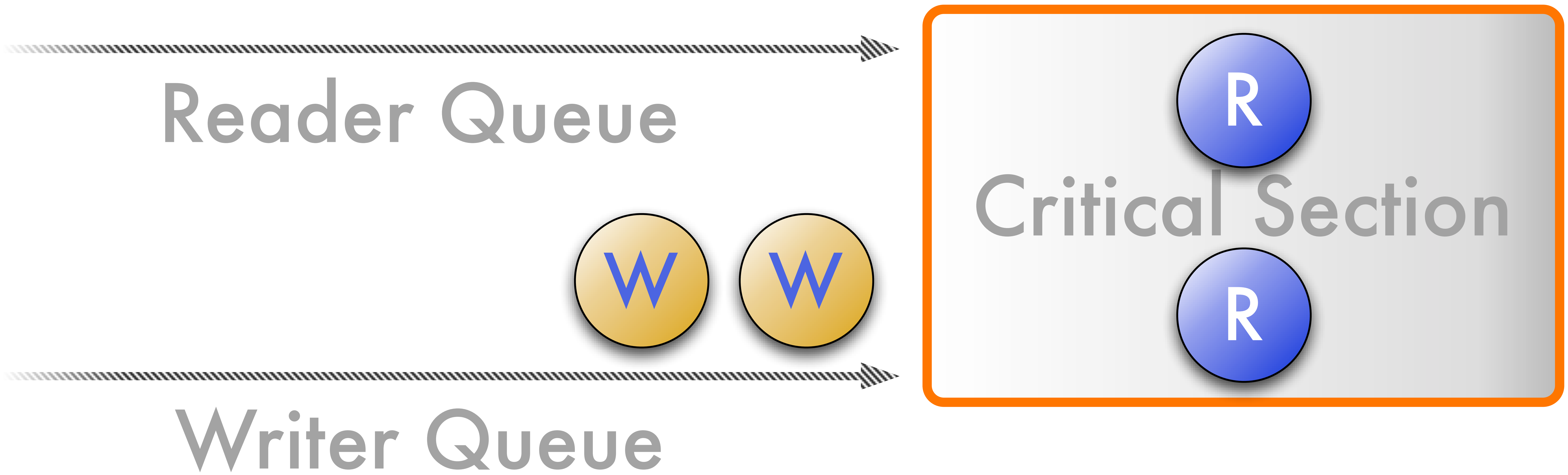
Phase-Fair RW Lock

staggering indicates arrival order



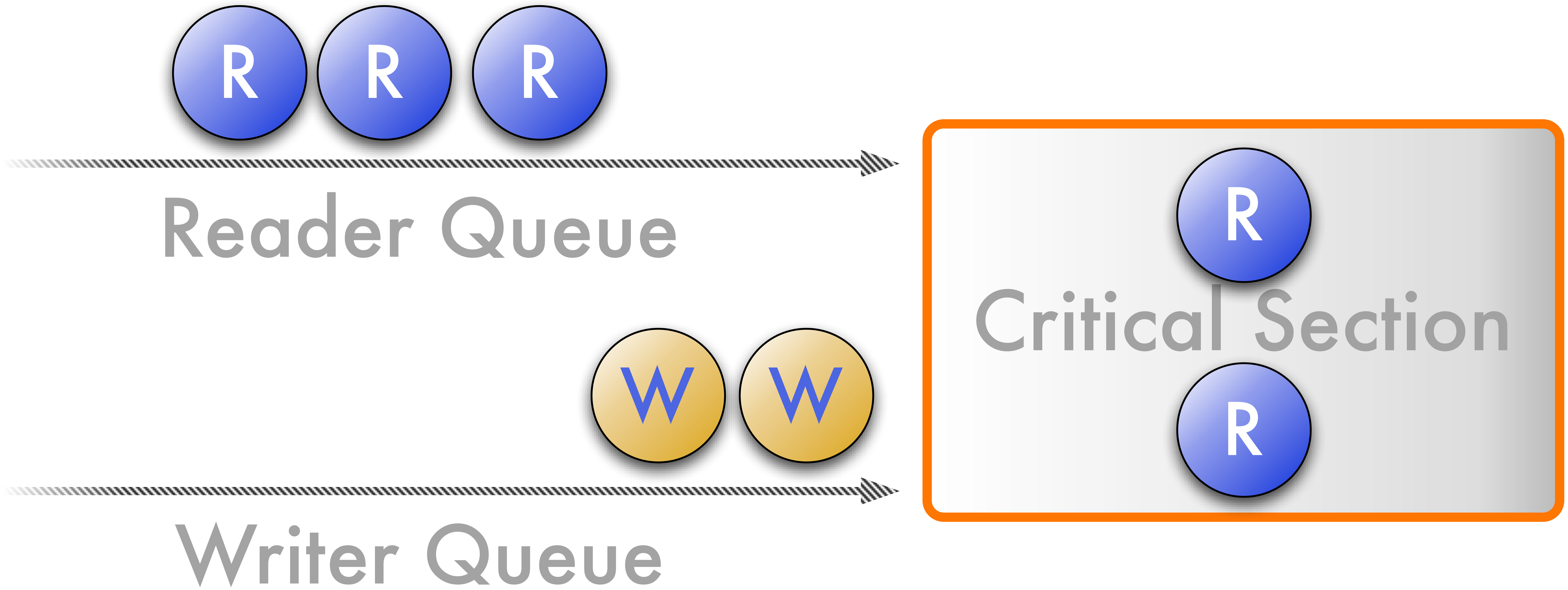
All **readers** enter when unblocked by an **exiting** **writer** (unless there are no writers).

Phase-Fair RW Lock



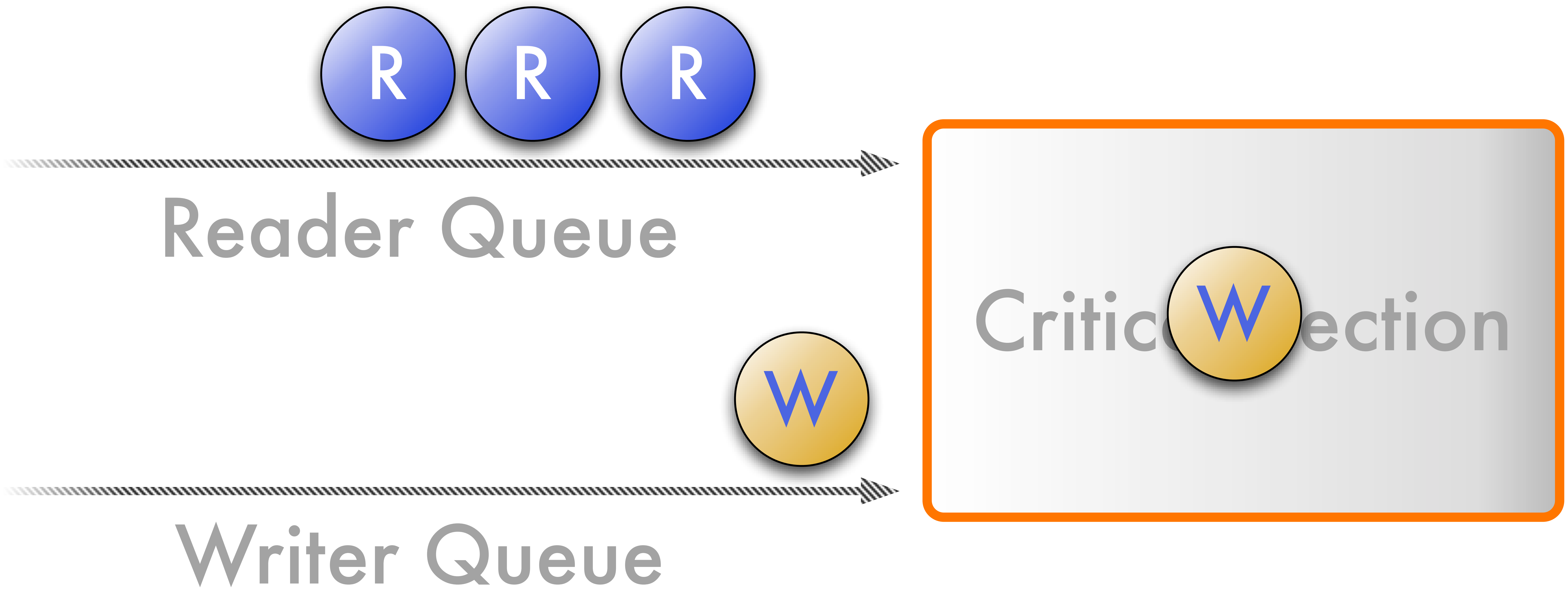
All **readers** enter when unblocked by an **exiting writer** (unless there are no writers).

Phase-Fair RW Lock



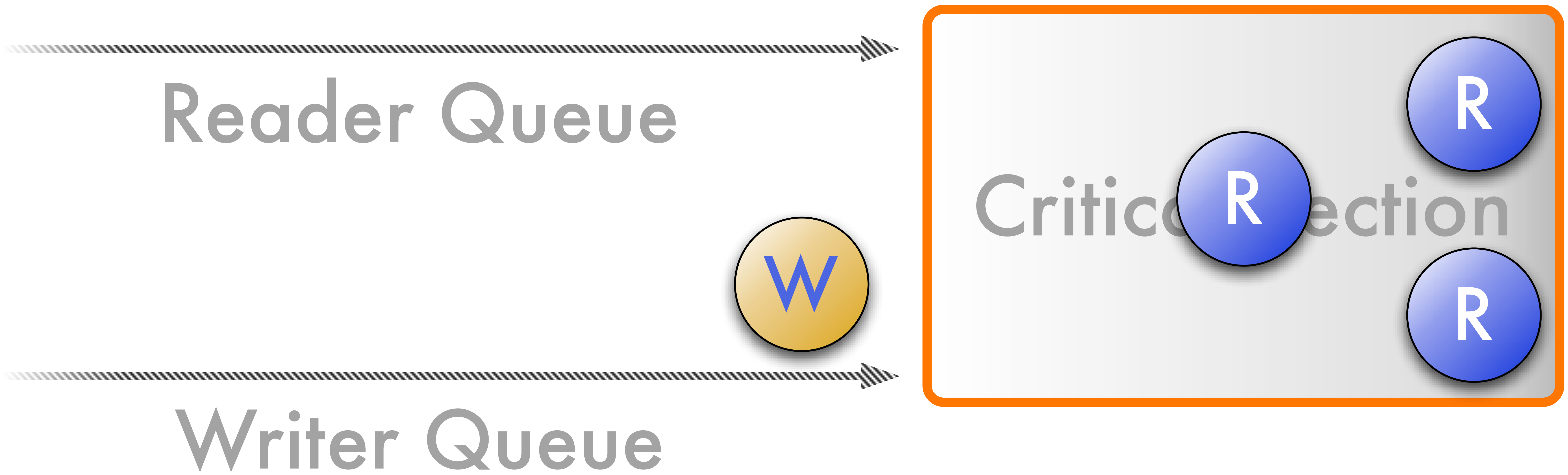
A **writer** enters when unblocked by the **last exiting reader** (unless there are no writers).

Phase-Fair RW Lock



All **readers** enter when unblocked by an **exiting writer** (unless there are no writers).

Phase-Fair RW Lock



Effect: **reader phases** and **writer phases** alternate.

Blocking Analysis

Assumptions

➡Resource request (protocol, spin loop, critical section) executed **non-preemptively**.

➡***m*** processors

Blocking Analysis

Assumptions

- ➡Resource request (protocol, spin loop, critical section) executed **non-preemptively**.
- ➡***m*** processors

Lock Type	Reader Blocking	Writer Blocking (# of phases)
Task-Fair Mutex	$O(m)$	$O(m)$
Task-Fair RW	$O(m)$	$O(m)$
Phase-Fair RW	**O(1)**	$O(m)$

Reader must wait for at most one reader and one writer phase.

Assumptions

- Resource request (protocol, spin loop, critical section) executed **non-preemptively**.
- ***m*** processors

Lock Type	Reader Blocking	Writer Blocking (# of phases)
Task-Fair Mutex	$O(m)$	$O(m)$
Task-Fair RW	$O(m)$	$O(m)$
Phase-Fair RW	**O(1)**	$O(m)$

Reader must wait for at most one reader and one writer phase.

As
➡R
ex
➡*n*

Blocking under Phase-Fair RW Locks is asymptotically optimal.

Lock Type	Reader Blocking	Writer Blocking (# of phases)
Task-Fair Mutex	O(*m*)	O(*m*)
Task-Fair RW	O(*m*)	O(*m*)
Phase-Fair RW	**O(1)**	O(*m*)

Reader must wait for at most one reader and one writer phase.

As

➡R

ex

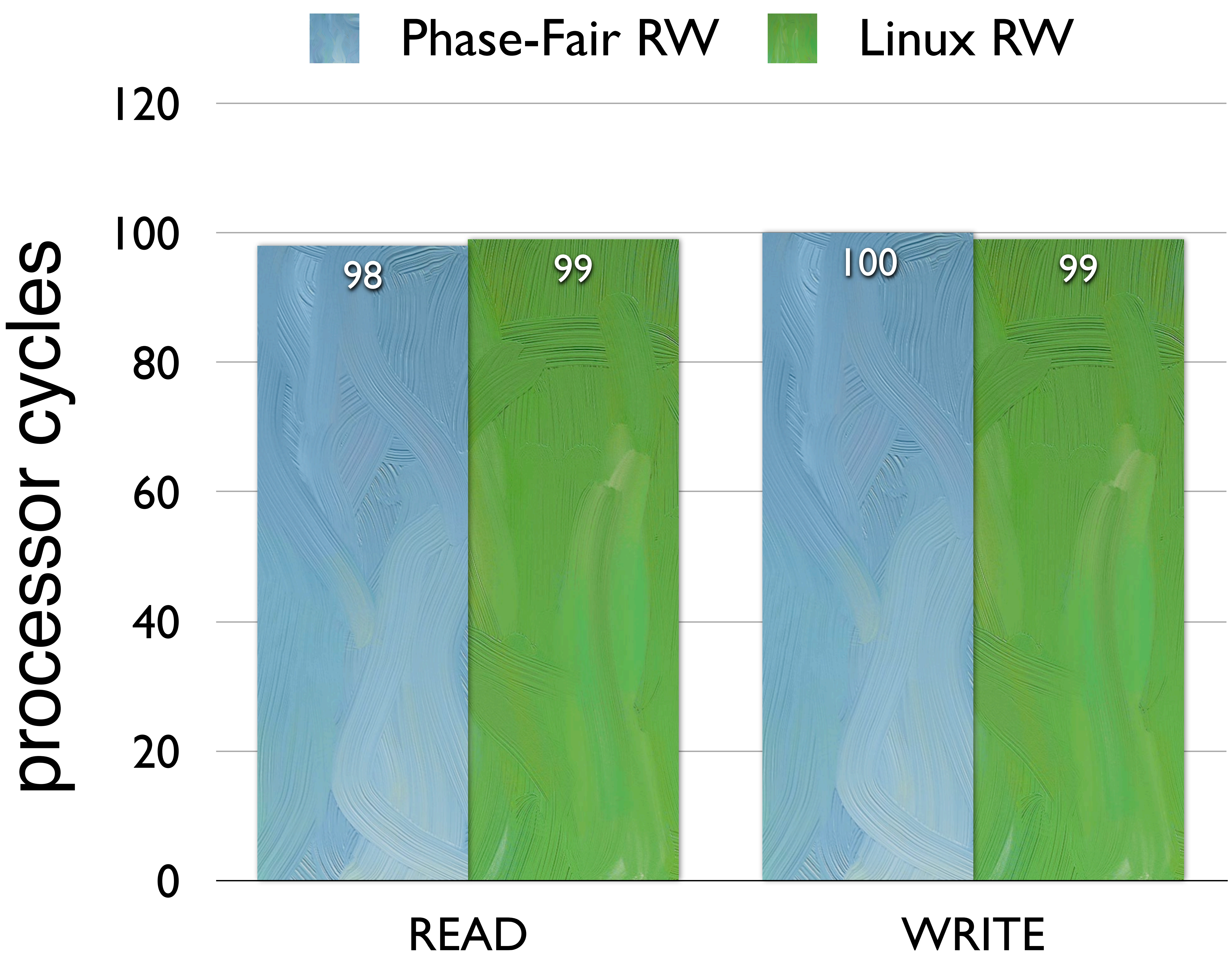
➡*m*

Blocking under Phase-Fair RW Locks is asymptotically optimal.

But can phase-fair locks be implemented **efficiently** on real hardware?

Phase-Fair RW	**O(1)**	O(m)

Lock/Unlock Overhead



Cache-hot micro-benchmark on an Intel Xeon X5650 (“Westmere”, Core i7).

Lock/Unlock Overhead

Do task-fair RW and phase-fair RW locks yield schedulability improvements?

Cache-hot micro-benchmark on an Intel Xeon X5650 ("Westmere", Core i7).

RW Resource Sharing Parameters

	In this talk	In my dissertation
Number of resources	6	6, 12, 24
Access probability	25%	10%, 25%, 40%
Write ratio	20%	10%, 20%, 30%, 50%, 75%
Critical Section Lengths	uniformly in [1, 15] μs	short: [1, 15] μs medium: [1, 100] μs long: [5, 1280] μs

R… …rs

Full study:

— In total, **7,290** parameter combinations.

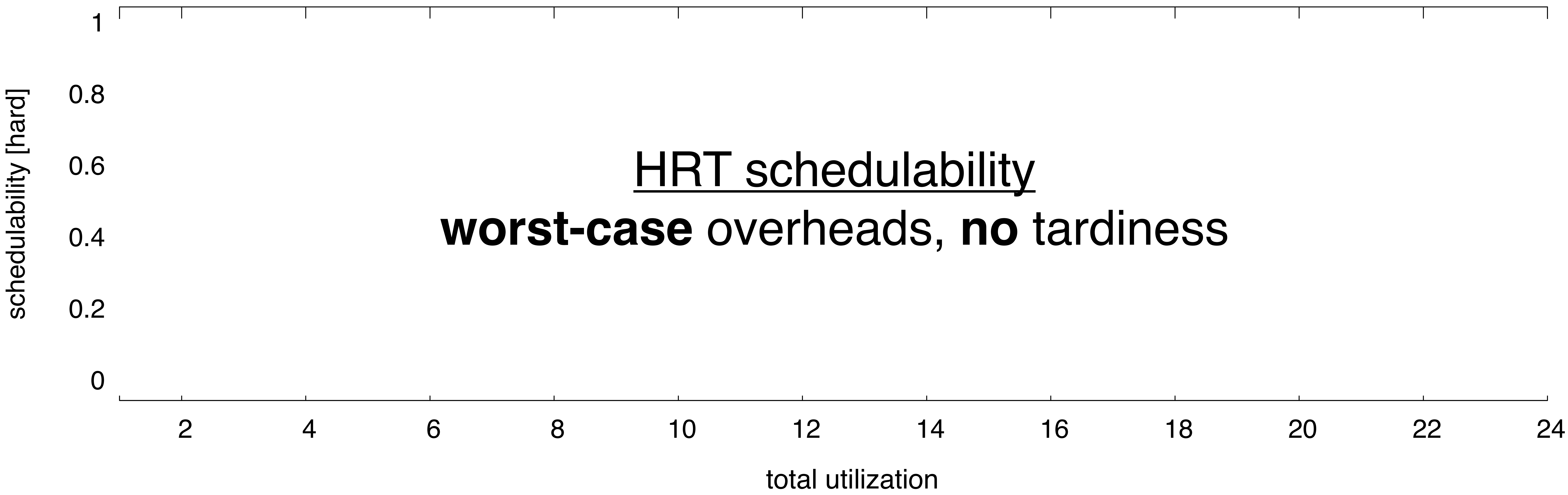
— Evaluated more than **34,000,000** task sets.

— Results in more than **100,000** schedulability plots.

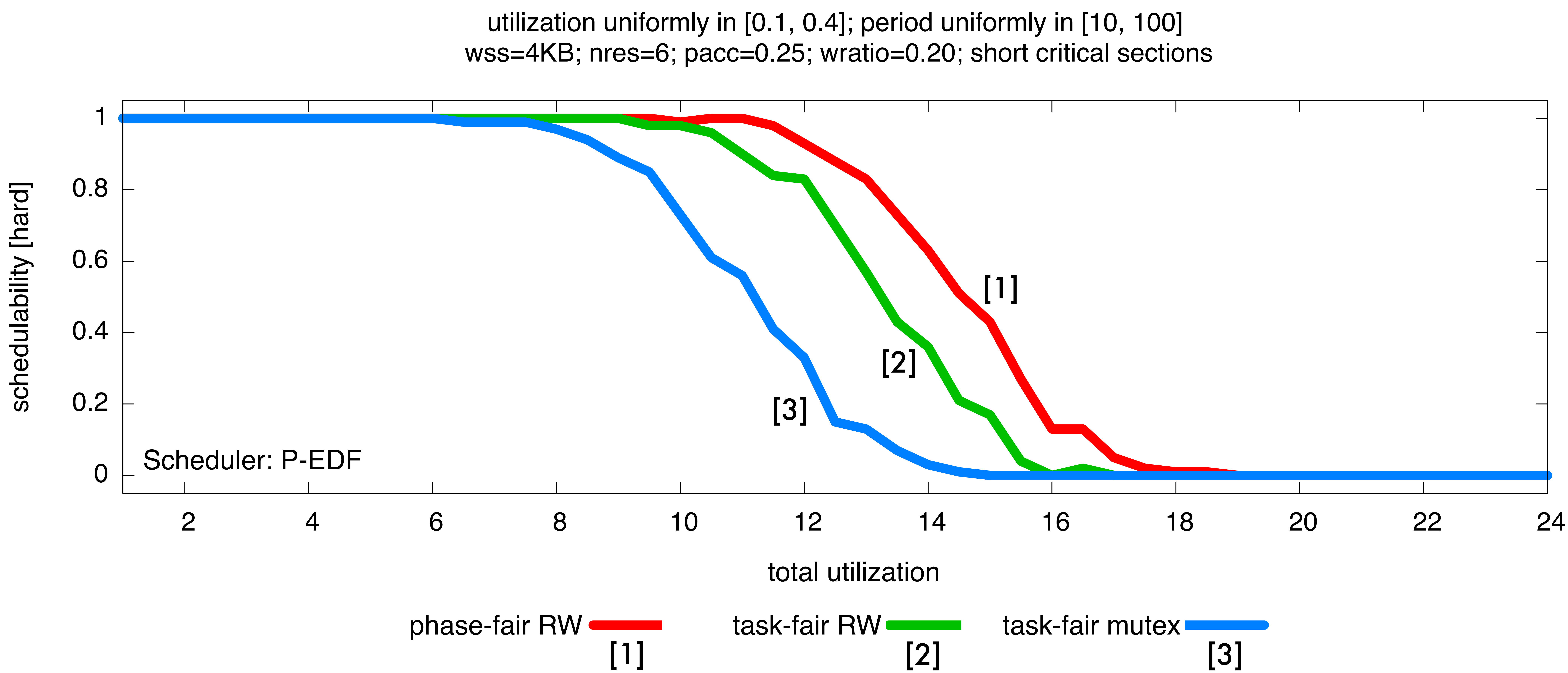
I'll show you one typical example…

		…ation
N… re…		…
Access probability	25%	10%, 25%, 40%
Write ratio	20%	10%, 20%, 30%, 50%, 75%
Critical Section Lengths	uniformly in [1, 15] µs	short: [1, 15] µs medium: [1, 100] µs long: [5, 1280] µs

HRT Schedulability Improvements

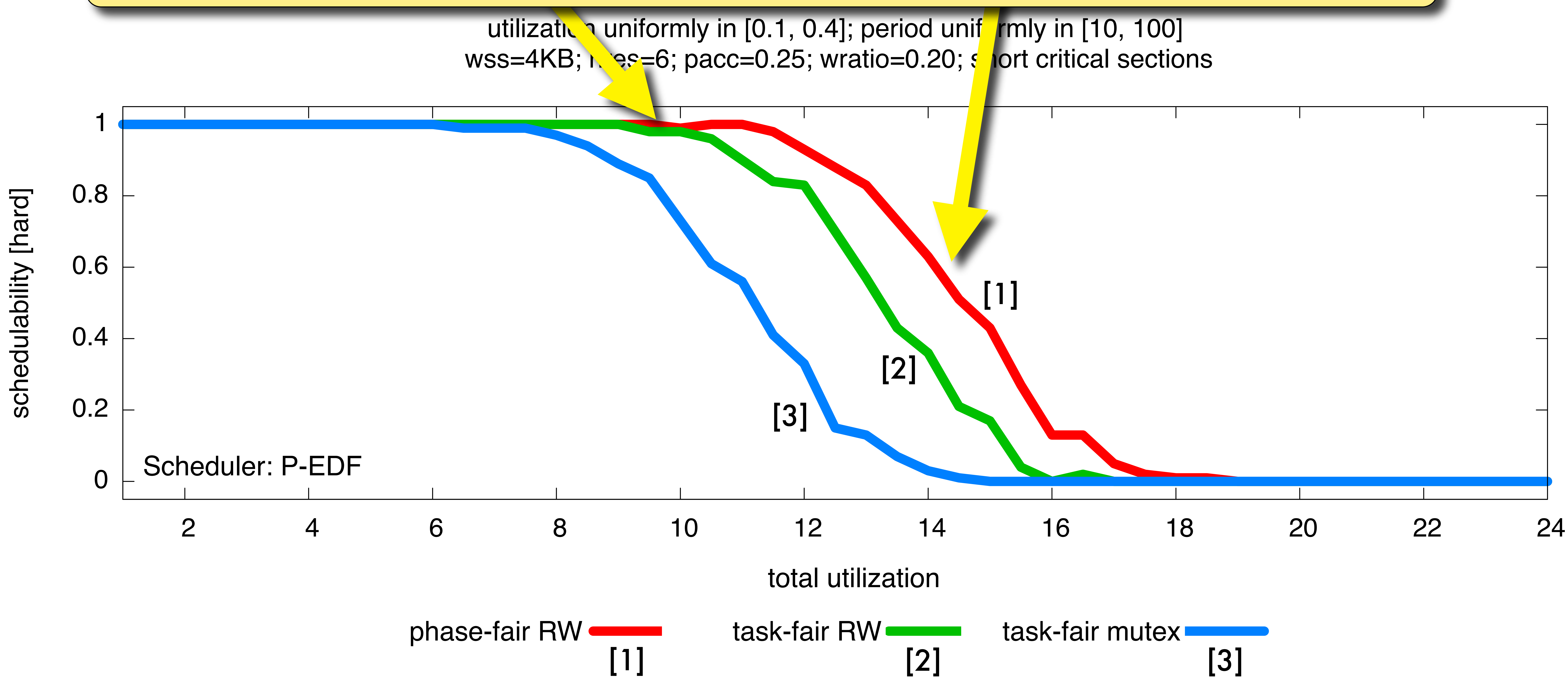


HRT Schedulability Improvements



RW spinlocks improve schedulability compared to mutex spinlocks.

Phase-fair RW locks yield greater improvement than task-fair RW locks.



Thesis Statement

When both overhead-related and algorithmic capacity loss are considered on a current multicore platform,

(i) partitioned scheduling is preferable to global and clustered approaches in the hard real-time case,

(ii) partitioned earliest-deadline first (P-EDF) scheduling is superior to partitioned fixed-priority (P-FP) scheduling and

(iii) clustered scheduling can be effective in reducing the impact of bin-packing limitations in the soft real-time case. Further,

(iv) multiprocessor locking protocols exist that are both **efficiently implementable** and **asymptotically optimal** with regard to the maximum duration of blocking.

Thesis Statement

When both overhead-related and algorithmic capacity loss are considered on a current multicore platform,

(i) partitioned scheduling is preferable to global and clustered approaches in the hard real-time case,

(ii) partitioned earliest-deadline first (P-EDF) scheduling is superior to partitioned fixed-priority (P-FP) scheduling and

(iii) clustered scheduling can be effective in reducing the impact of bin-packing limitations in the soft real-time case. Further,

(iv) multiprocessor locking protocols exist that are both **efficiently implementable** and **asymptotically optimal** with regard to the maximum duration of blocking.

Scheduling and Locking in Multiprocessor Real-Time Operating Systems

Scheduling and Locking in Multiprocessor Real-Time Operating Systems

Hard Real-Time

Use **partitioned EDF**.

Soft Real-Time

Support **clustered** scheduling.

Scheduling and Locking in Multiprocessor Real-Time Operating Systems

Keep it simple

Use non-preemptive **spinlocks**.
Use **FIFO** queues: **optimal** and **practical**.

Be polite

Phase-fair RW locks can be implemented **efficiently** and improve worst-case analysis.

Future Work

RTOS Implementation.

➡ Hierarchical scheduling / container framework.

➡ Reduce lock contention in global and clustered scheduling.

Locking Optimality.

➡ Improved bounds under s-aware analysis.

➡ Nested requests.

Non-blocking synchronization.

➡ Wait-free, lock-free.

➡ Read-copy update (RCU).

Experiments

➡ Use worst-case execution time analysis.

➡ Use more real applications.

Max Planck Institute for Software Systems

Kaiserslautern · Saarbrücken // Germany

http://www.mpi-sws.org

Acknowledgements (I/II)

My advisor and committee for their guidance and support.
Jim Anderson, Sanjoy Baruah, Hermann Härtig, Jan Prins, Don Smith, Paul McKenney

My co-authors and the real-time group.
Aaron Block, Andrea Bastoni, John Calandrino, Uma Devi, Glenn Elliott, Jon Herman, Hennadiy Leontyev, Chris Kenna, Alex Mills, Mac Mollison

FULBRIGHT

The Fulbright Program and the UNC Graduate School for funding my first year and my last year, respectively.

UNC THE GRADUATE SCHOOL

The CS Department's **amazing** staff!
Special thanks to *Mike Stone* for un-breaking everything the real-time group touches; *Murray Anderegg* and *John Sopko* for putting up with my Linux special requests; *Bil Hays* for keeping the real-time lab cool; and *Sandra Neely*, *Janet Jones*, *Jodie Turnbull*, and *Dawn Andres* for keeping me out of paperwork trouble.

Acknowledgements (II/II)

My Sitterson Hall friends.
Sasa Junuzovic, Jay Aikat, Sean Curtis, Stephen Olivier, Keith Lee, Jamie Snape, Srinivas Krishnan, Anish Chandak, Stephen Guy

Special thanks to my friends *Aaron* & *Nicki*, *Jasper*, *Dot*, and *Andrea* for keeping me sane.

To my parents Harald & Petra, and my girlfriend Nora, for their unwavering support, understanding, and encouragement.

Thank you!

RTOS & Scheduling

1. C. Kenna, J. Herman, **B. Brandenburg**, A. Mills, and J. Anderson, “Soft Real-Time on Multiprocessors: Are Analysis-Based Schedulers Really Worth It?”, Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS 2011), December 2011, to appear.
2. A. Bastoni, **B. Brandenburg**, and J. Anderson, “Is Semi-Partitioned Scheduling Practical?”, Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS 2011), pp. 125-135. IEEE, July 2011.
3. A. Bastoni, **B. Brandenburg**, and J. Anderson, “An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor EDF Schedulers”, Proceedings of the 31th IEEE Real-Time Systems Symposium (RTSS 2010), pp. 14-24. IEEE, December 2010.
4. **B. Brandenburg**, H. Leontyev, and J. Anderson, “An Overview of Interrupt Accounting Techniques for Multiprocessor Real-Time Systems”, Journal of Systems Architecture, special issue on selected papers from the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, in press, 2010.
5. A. Bastoni, **B. Brandenburg**, and J. Anderson, “Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability”, Proceedings of the Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010), pp. 33-44, July 2010.
6. **B. Brandenburg** and J. Anderson, “On the Implementation of Global Real-Time Schedulers”, Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009), pp. 214-224. IEEE, December 2009.
7. **B. Brandenburg** and J. Anderson, “Joint Opportunities for Real-Time Linux and Real-Time Systems Research”, Proceedings of the 11th Real-Time Linux Workshop (RTLWS 2009), pp. 19-30. Real-Time Linux Foundation, September 2009.
8. **B. Brandenburg**, H. Leontyev, and J. Anderson, “Accounting for Interrupts in Multiprocessor Real-Time Systems”, Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2009), pp. 273-283. IEEE, August 2009.
9. M. Mollison, **B. Brandenburg**, and J. Anderson, “Towards Unit Testing Real-Time Schedulers in LITMUS^{RT}”, Proceedings of the Fifth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009), pp. 33-39. Politécnico do Porto, July 2009.
10. J. Anderson, S. Baruah, and **B. Brandenburg**, “Multicore Operating-System Support for Mixed Criticality”, Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification (part of CPS Week 2009). April 2009.
11. **B. Brandenburg**, J. Calandrino, and J. Anderson, “On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study”, Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS 2008), pp. 157-169. IEEE, December 2008.
12. A. Block, **B. Brandenburg**, J. Anderson, and S. Quint, “An Adaptive Framework for Multiprocessor Real-Time Systems”, Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS 2008), pp. 23-33. IEEE, July 2008.
13. **B. Brandenburg**, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson, “LITMUS^{RT}: A Status Report”, Proceedings of the Ninth Real-Time Linux Workshop (RTLWS 2007), pp. 107-123. Real-Time Linux Foundation, November 2007.
14. **B. Brandenburg** and J. Anderson, “Feather-Trace: A Light-Weight Event Tracing Toolkit”, Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2007), pp. 19-28. National ICT Australia, July 2007.
15. **B. Brandenburg** and J. Anderson, “Integrating Hard/Soft Real-Time Tasks and Best-Effort Jobs on Multiprocessors”, Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS 2007), pp. 61-70. IEEE, July 2007

Locking Protocol Design & Analysis

16. **B. Brandenburg** and J. Anderson, “Real-Time Resource-Sharing under Clustered Scheduling: Mutex, Reader-Writer, and k-Exclusion Locks”, Proceedings of the International Conference on Embedded Software (EMSOFT 2011), to appear, October 2011.
17. **B. Brandenburg** and J. Anderson, “Optimality Results for Multiprocessor Real-Time Locking”, Proceedings of the 31th IEEE Real-Time Systems Symposium (RTSS 2010), pp. 49-60. IEEE, December 2010.
18. **B. Brandenburg** and J. Anderson, “Spin-Based Reader-Writer Synchronization for Multiprocessor Real-Time Systems”, Real-Time Systems, special issue on selected papers from the 21st Euromicro Conference on Real-Time Systems, Volume 46, Number 1, pp. 25-87, 2010.
19. **B. Brandenburg** and J. Anderson, “Reader-Writer Synchronization for Shared-Memory Multiprocessor Real-Time Systems”, Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS 2009), pp. 184-193. IEEE, July 2009.
20. **B. Brandenburg** and J. Anderson, “A Comparison of the M-PCP, D-PCP, and FMLP on LITMUS^{RT}”, Proceedings of the 12th International Conference On Principles Of Distributed Systems (OPODIS 2008), Lecture Notes in Computer Science 5401, pp. 105-124. Springer-Verlag, December 2008.
21. **B. Brandenburg** and J. Anderson, “An Implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP Real-Time Synchronization Protocols in LITMUS^{RT}”, Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2008), pp. 185-194. IEEE, August 2008.
22. **B. Brandenburg**, J. Calandrino, A. Block, H. Leontyev, and J. Anderson, “Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin?”, Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008), pp. 342-353. IEEE, April 2008.
23. A. Block, H. Leontyev, **B. Brandenburg**, and J. Anderson, “A Flexible Real-Time Locking Protocol for Multiprocessors”, Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007), pp. 47-57. IEEE, August 2007.