Sharing Non-Processor Resources in Multiprocessor Real-Time Systems

Dissertation Defense

Bryan C. Ward
Under the Direction of Prof. Jim Anderson

Real-Time System

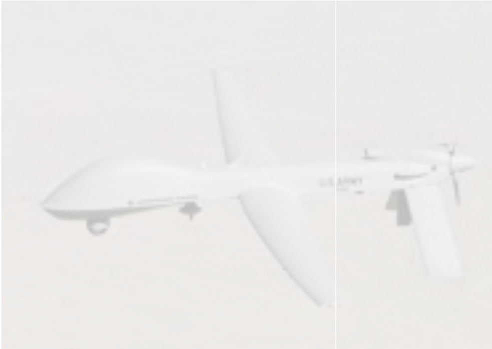
- System that requires both
  - Logical correctness, and
  - Temporal correctness.

- Common in safety-critical and cyber-physical systems.

Fundamentally affects how systems are designed and built.

Predictability is more important than performance.

- Systems that requires both
    - Logical correctness, and
    - Temporal correctness.
- Common in safety-critical and cyber-physical systems.



Temporal Correctness

- Temporal correctness requires:
    - **Models** of system components,
    - Sound mathematical **analysis**.

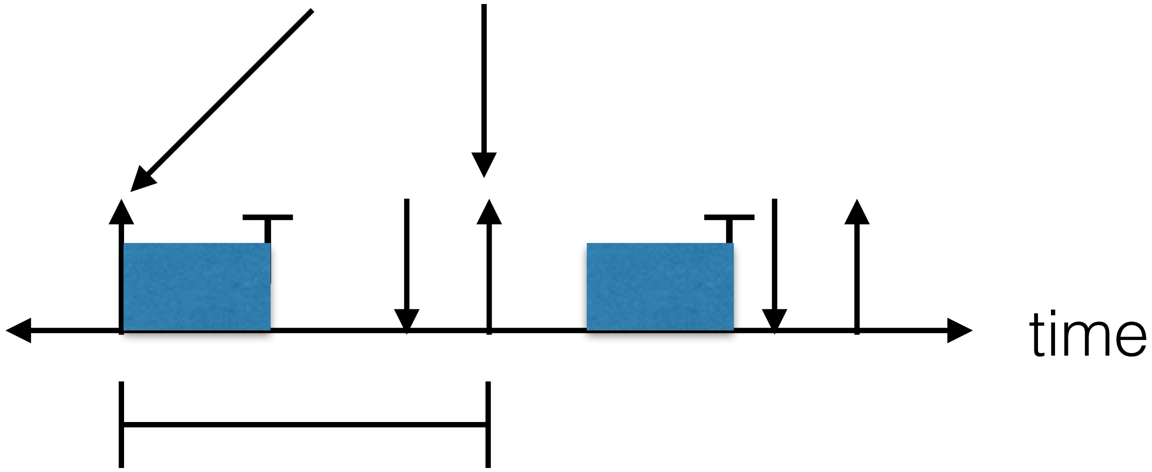
Real-Time Task Model

Pedestrian Detection

Real-Time Task Model

Pedestrian Detection

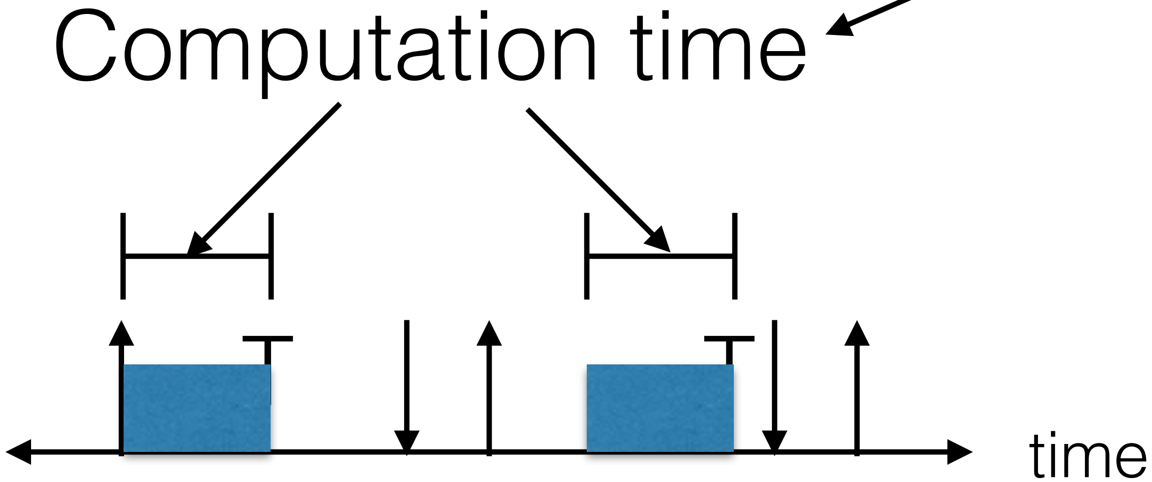
Job **releases**.
Next frame available from the camera.



Period: time between frames. *e.g.*, 33ms for 30FPS.

Real-Time Task Model

Pedestrian Detection



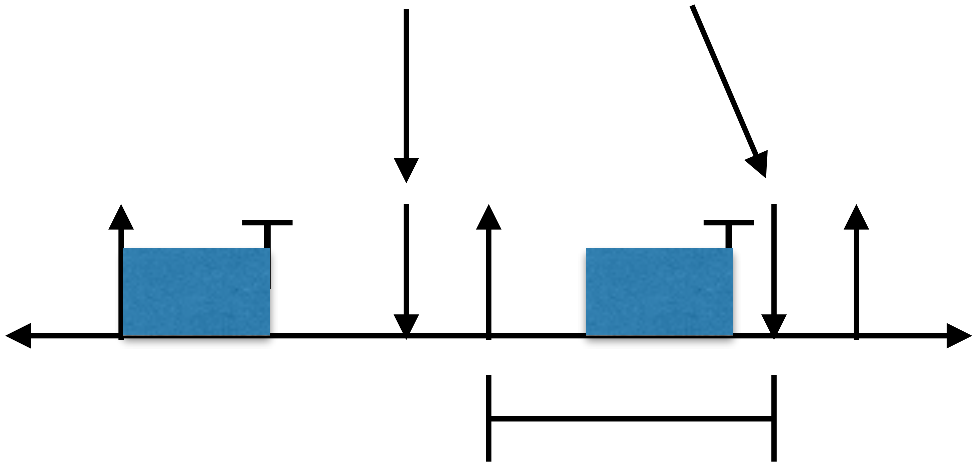
Computation time

Real-Time Task Model

Pedestrian Detection

Job **deadlines**.

Time by which the computation must complete.



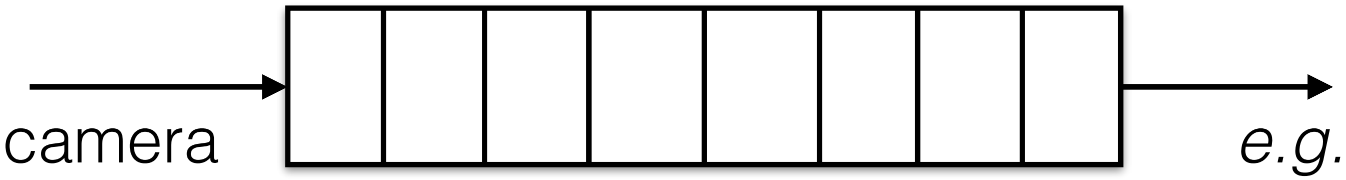
Relative Deadline: time between release and deadline.

Scheduling and Analysis

- **Scheduling algorithm** determines when jobs run.
  - Earliest-Deadline First (EDF).
  - Fixed Priority (FP).
- **Schedulability test** used to determine whether all jobs will provably finish before their deadlines.

Synchronization

- In practice, tasks share **resources** to which accesses must be synchronized.



Producer
e.g., read frame from camera

Consumer
e.g., pedestrian detection

Shared Buffer

- **Locking protocol** used to synchronize such accesses to ensure safety.

Synchronization Example

```
def transaction(from_acct, to_acct, amount)
 if(balances[from_acct] < amount)
  # Insufficient funds
  return False
 else
  balances[to_acct] = balances[to_acct] + amount
  balances[from_acct] = balances[from_acct] - amount
```

Sy e

Two threads **executing concurrently** can produce a **logically incorrect** or **unsafe** result.

```
def transaction(from_acct, to_acct, amount)
 if(balances[from_acct] < amount)
  # Insufficient funds
  return False
 else
  balances[to_acct] = balances[to_acct] + amount
  balances[from_acct] = balances[from_acct] - amount
```

Task 1: transaction(A, B, 10)

Task 2: transaction(A, C, 10)

	A	B	C
Balances	$12 $2	$8 $18	$4 $14

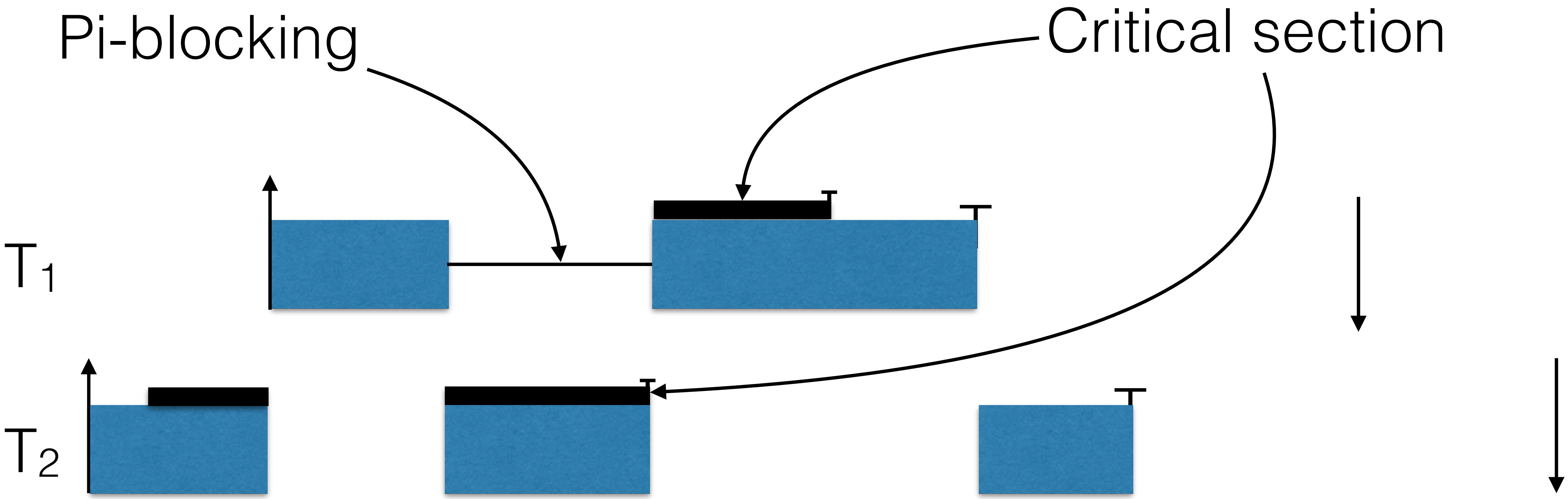
Synchronization Example

```
def transaction(to_acct, from_acct, amount)
 lock()
 if(balances[from_acct] < amount)
  # Insufficient funds
  return False
 else
  balances[to_acct] = balances[to_acct] + amount
  balances[from_acct] = balances[from_acct] - amount
 unlock()
```

Using a **locking protocol**, we can fix this issue by ensuring only one transaction executes at a time.

Priority-Inversion Blocking

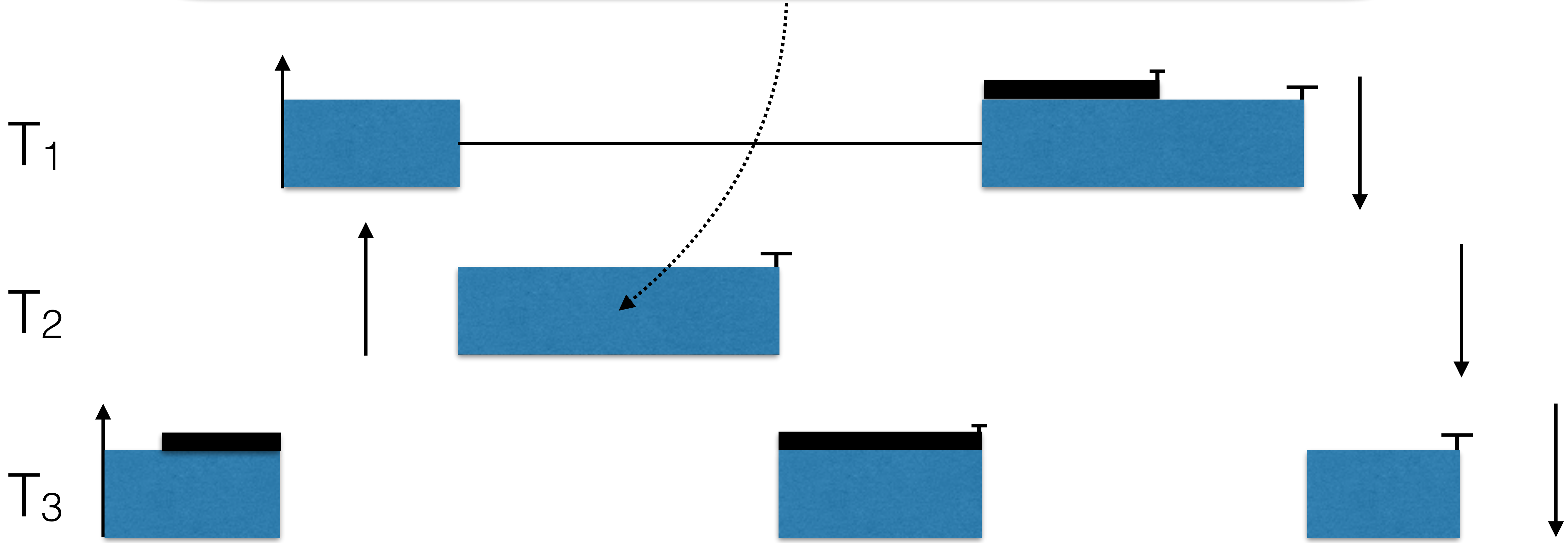
- Synchronization may cause **priority-inversion blocking (pi-blocking)**.
- Pi-blocking must be incorporated into schedulability analysis.
- Pi-blocking can cause significant **utilization loss**.



Progress Mechanisms

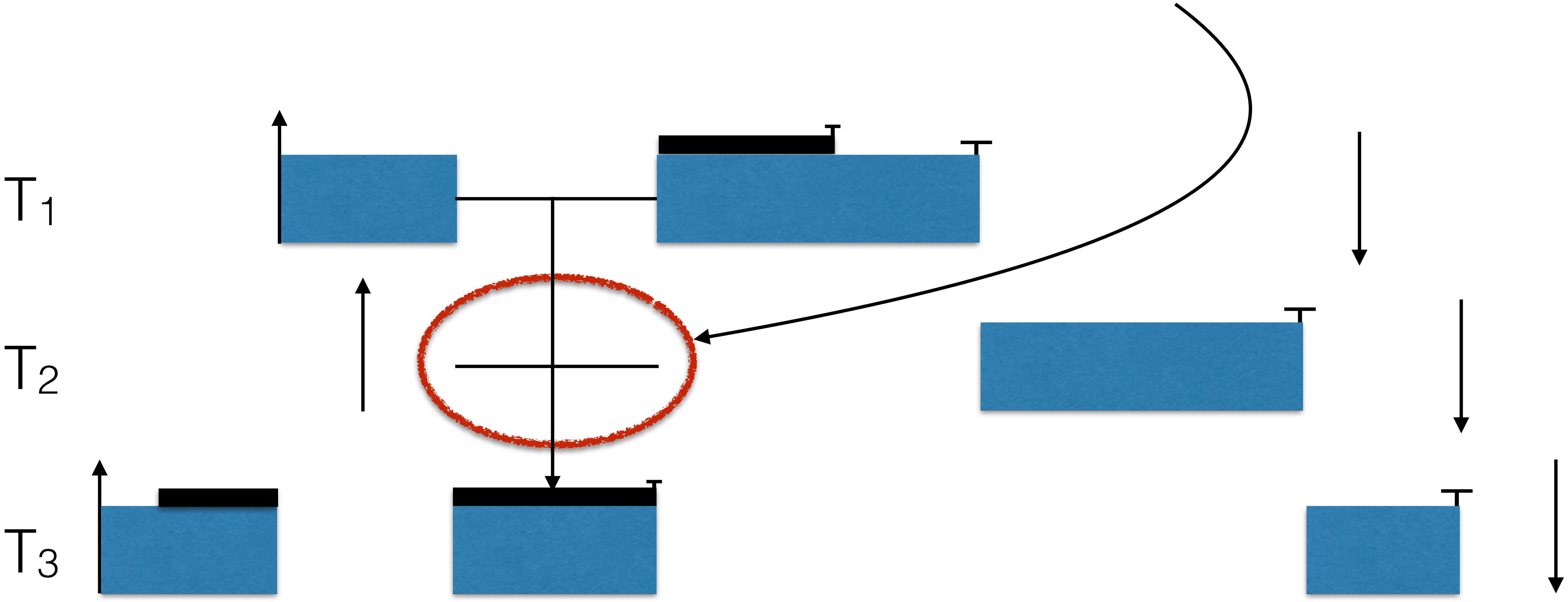
- Must consider scheduling and synchronization interactions in computing pi-blocking bounds.
-

Medium-priority non-resource using task can cause pi-blocking for blocked high-priority task.



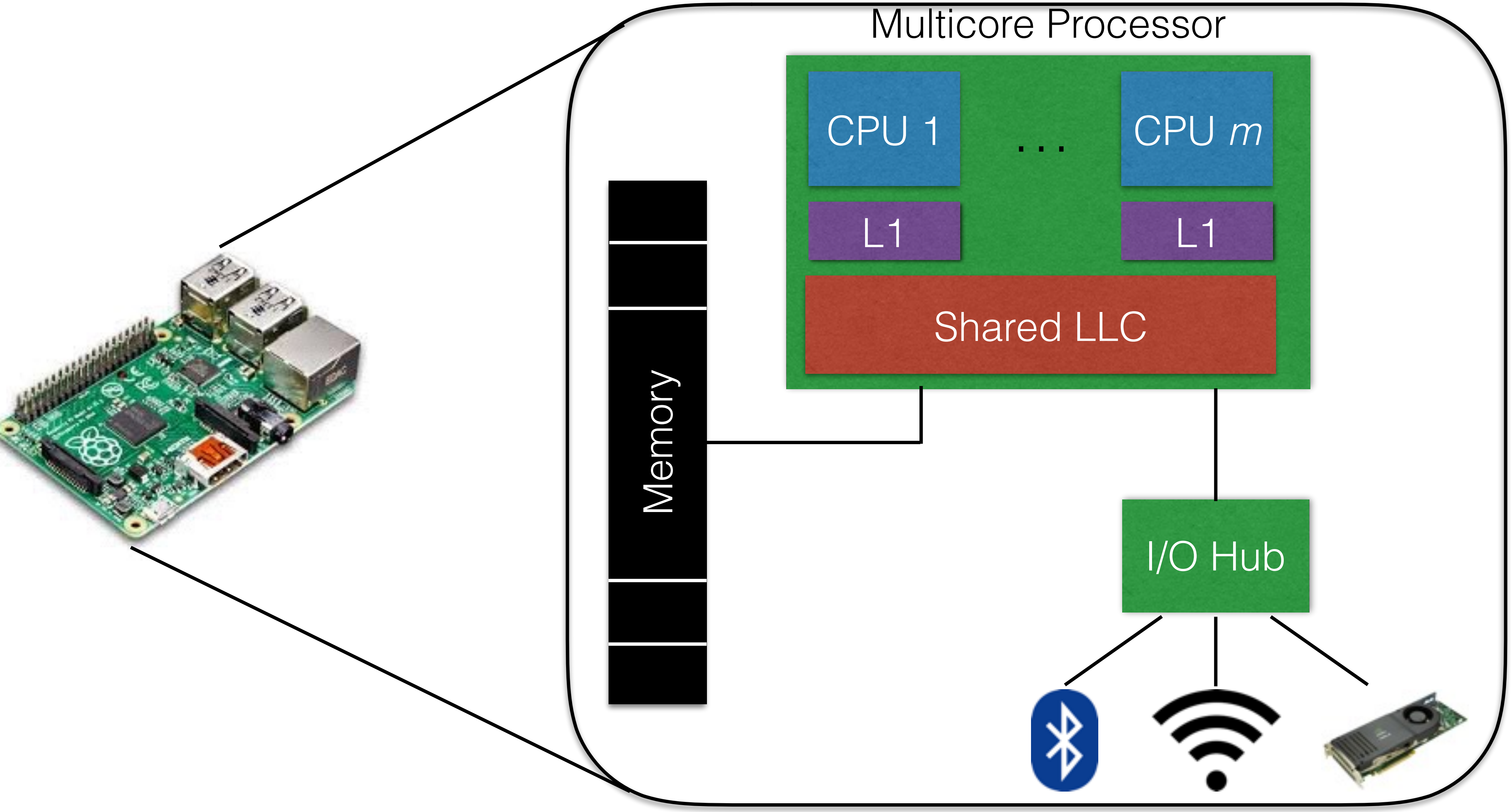
Priority Inheritance

- An example progress mechanism is priority inheritance.
- Resource-holding job inherits the priority of the highest-priority blocked job.
- Can cause pi-blocking for non-resource-using tasks.



Priority inheritance

Modern Multicore Architectures



Multicore Processor

CPU 1 … CPU *m*

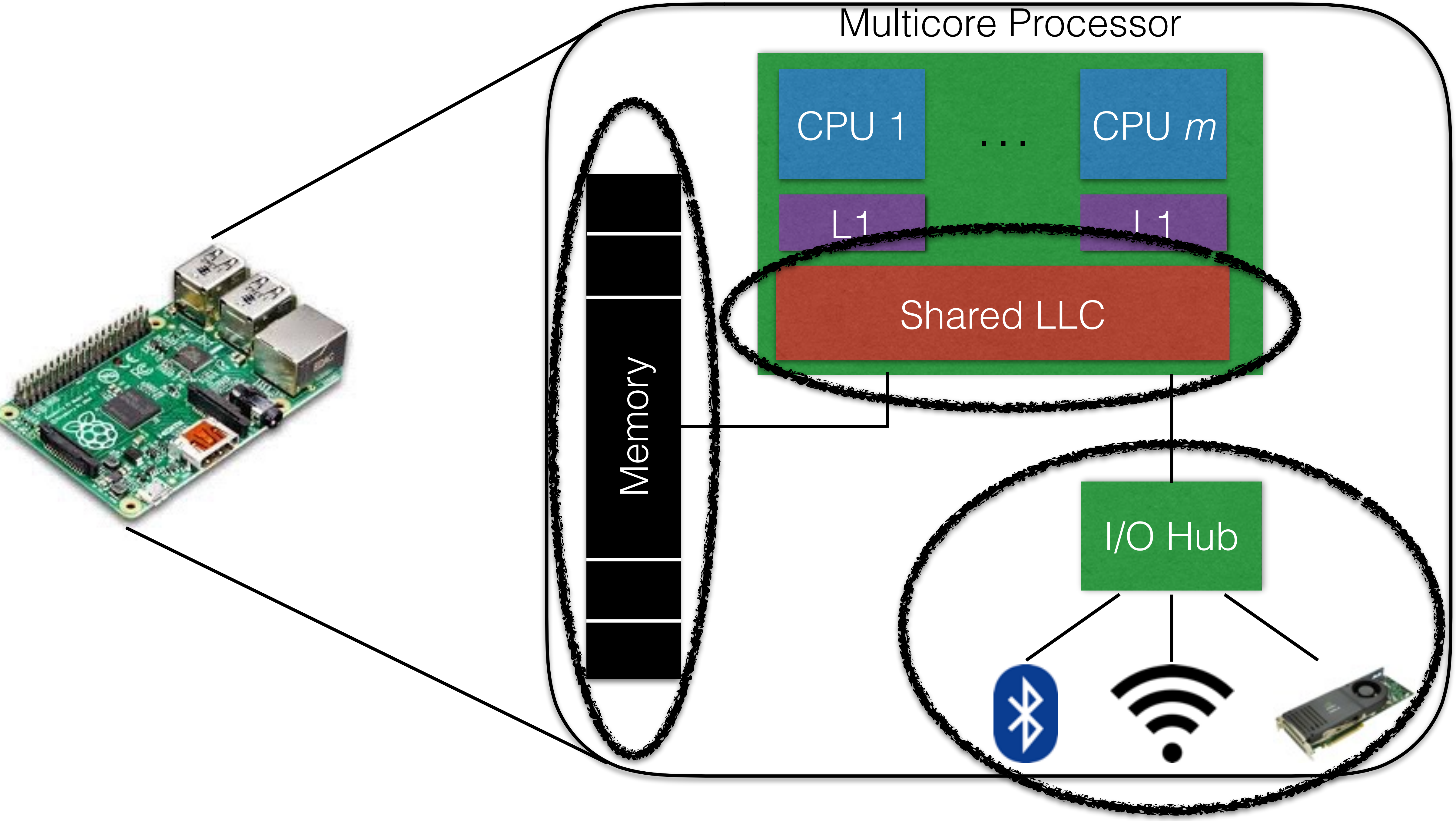
L1 L1

Shared LLC

Memory

I/O Hub

Multicore processors are designed with several shared hardware components. Explicitly synchronizing access to such resources can improve predictability.



“Dependencies among tasks in real-time systems through shared resources, both **memory objects**, as well as **shared hardware resources**, can be managed through synchronization protocols. Such protocols can be designed to exploit the inherent sharing constraints of the managed resources in order to achieve **improved resource utilization**.”

Thesis Statement

Outline

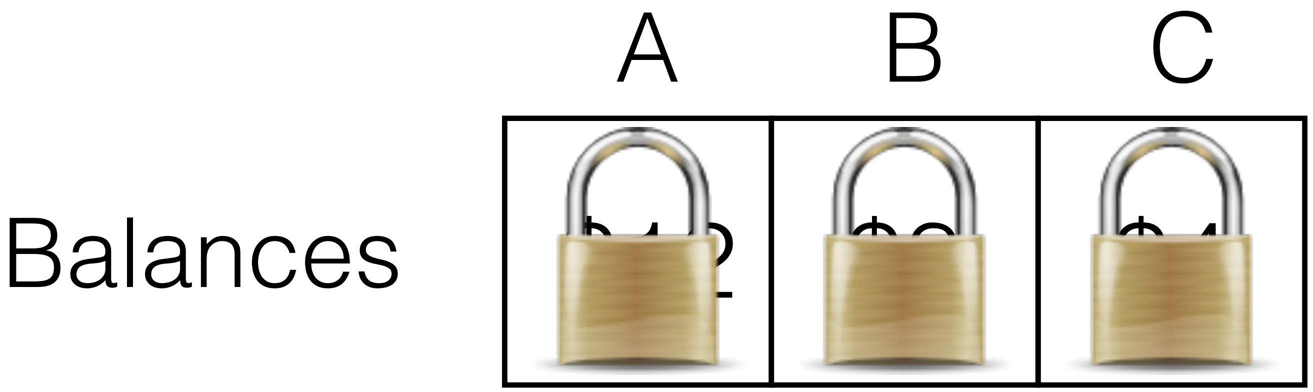
- Introduction & Background
- **Memory Objects**
  - **Fine-grained mutex locks (RNLP)**
  - Fine-grained reader/writer locks (R/W RNLP)
- Hardware Resources
  - Preemptive mutual exclusion
  - Half-protected exclusion
- Conclusions

Coarse-Grained Locking

```
def transaction(to_acct, from_acct, amount)
 lock()
 if(balances[from_acct] < amount)
  # Insufficient funds
  return False
 else
  balances[to_acct] = balances[to_acct] + amount
  balances[from_acct] = balances[from_acct] - amount
 unlock()
```

Fine-grained locking can **reduce blocking** by allowing non-conflicting transactions to be processed concurrently.

```
def transaction(to_acct, from_acct, amount)
 lock(from_acct)
 if(balances[from_acct] < amount)
  # Insufficient funds
  return False
 else
  lock(to_acct)
  balances[to_acct] = balances[to_acct] + amount
  unlock(to_acct)
  balances[from_acct] = balances[from_acct] - amount
 unlock(from_acct)
```

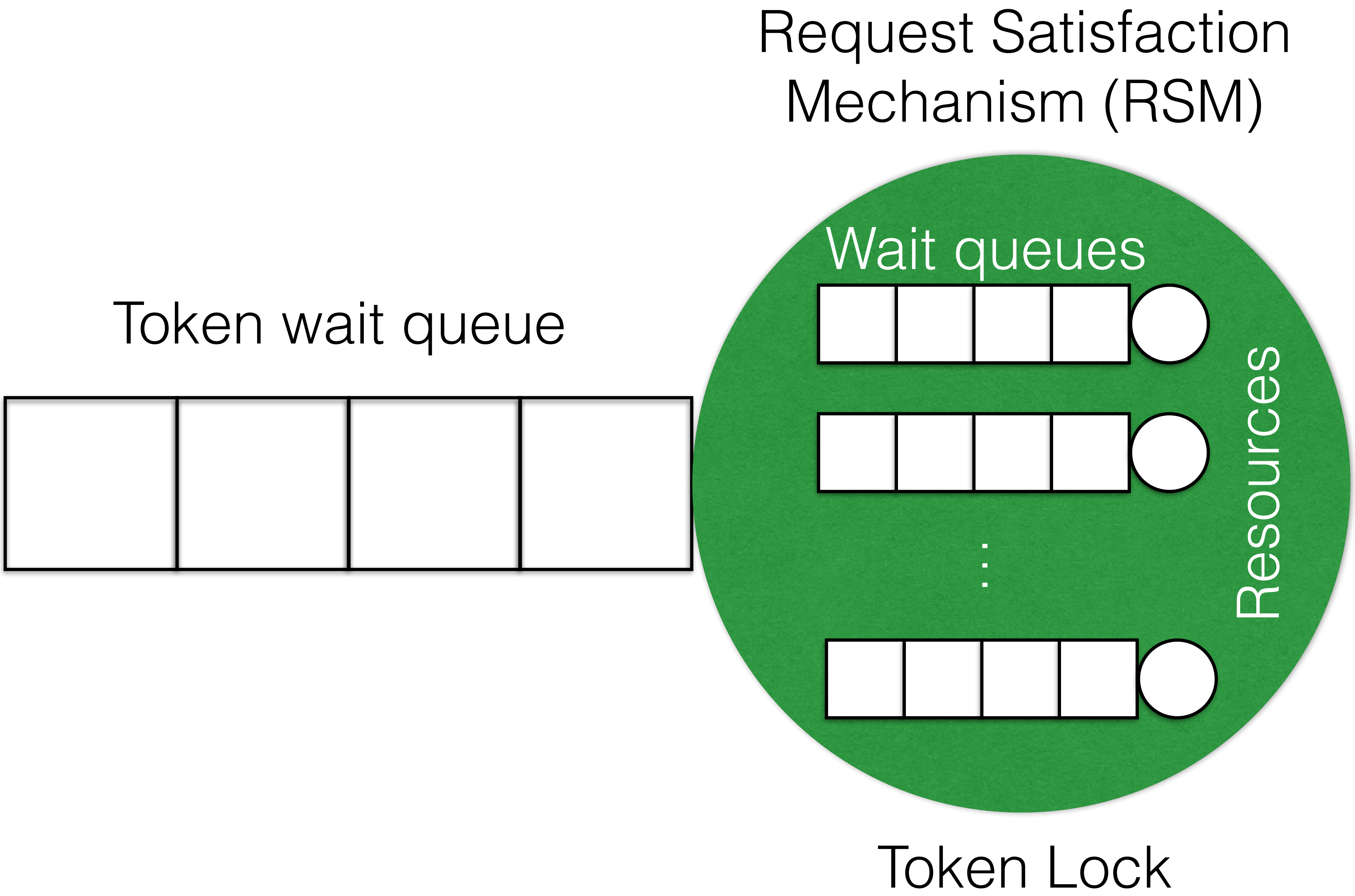


RNLP

- Real-Time Nested Locking Protocol (RNLP).
- First multiprocessor real-time locking protocol to support fine-grained locking.
- Modular, “plug-and-play” architecture can be configured optimally under different schedulers and analysis assumptions.

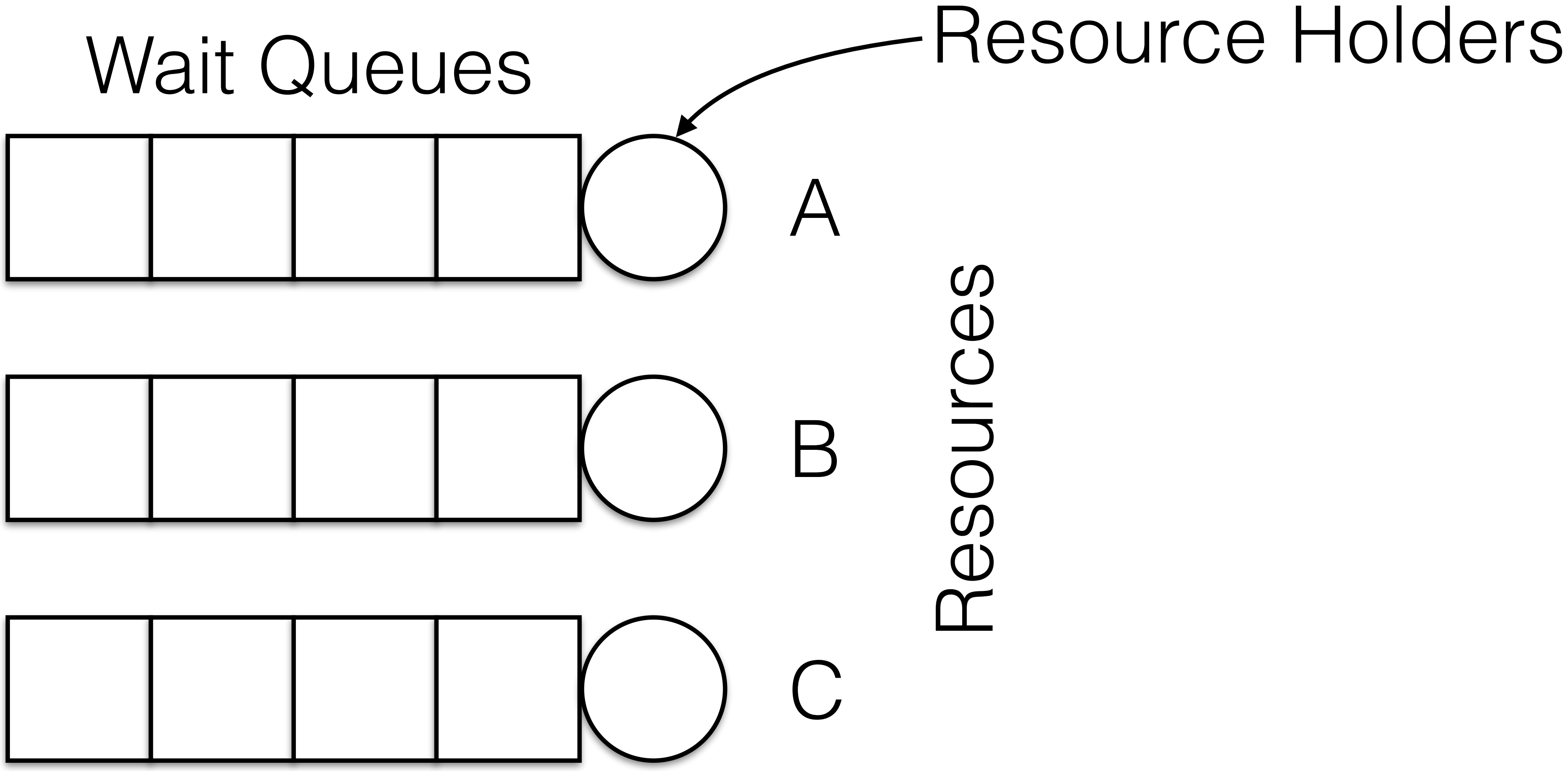
*B. Ward and J. Anderson. *“Supporting Nested Locking in Multiprocessor Real-Time Systems.”* ECRTS ‘12

RNLP


Request Satisfaction Mechanism (RSM)
Token wait queue
Wait queues
Resources
⋮
Token Lock

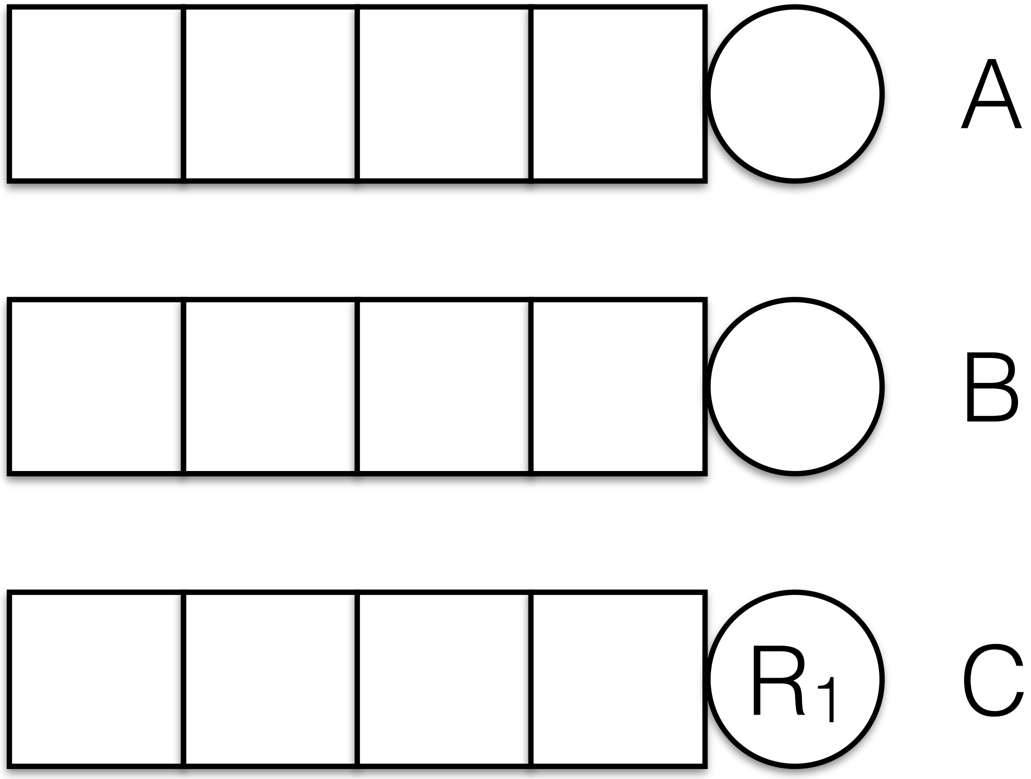
RSM

The key problem: later-issued requests hold resources that are requested in a nested fashion.



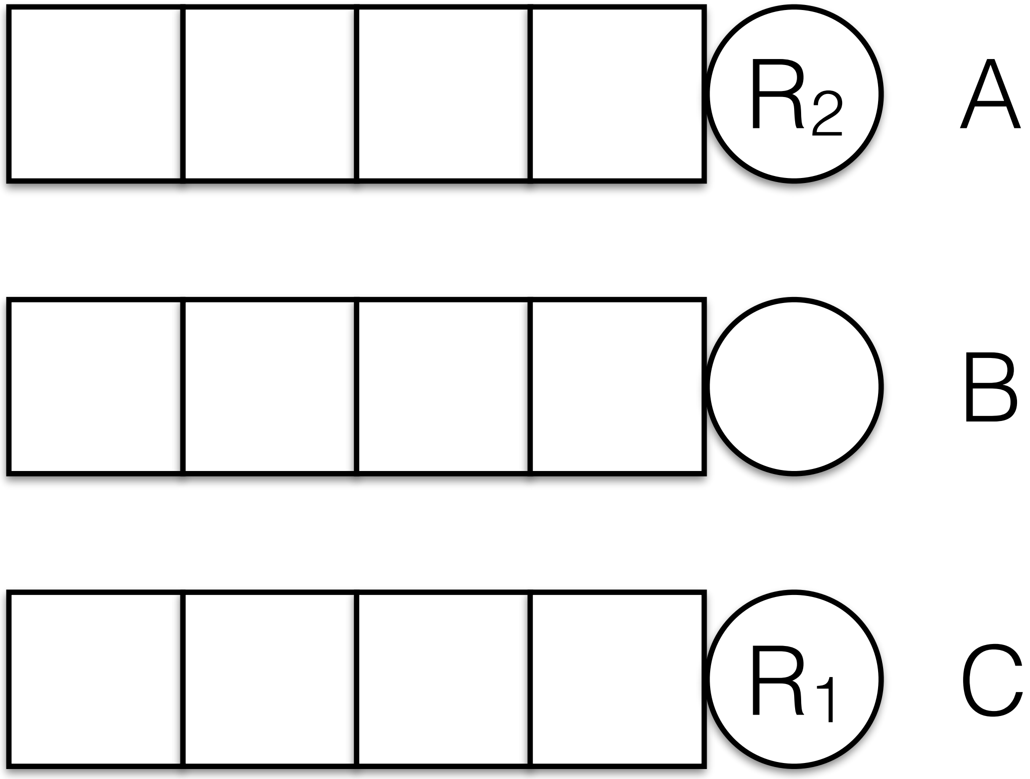
RSM

The key problem: later-issued requests hold resources that are requested in a nested fashion.



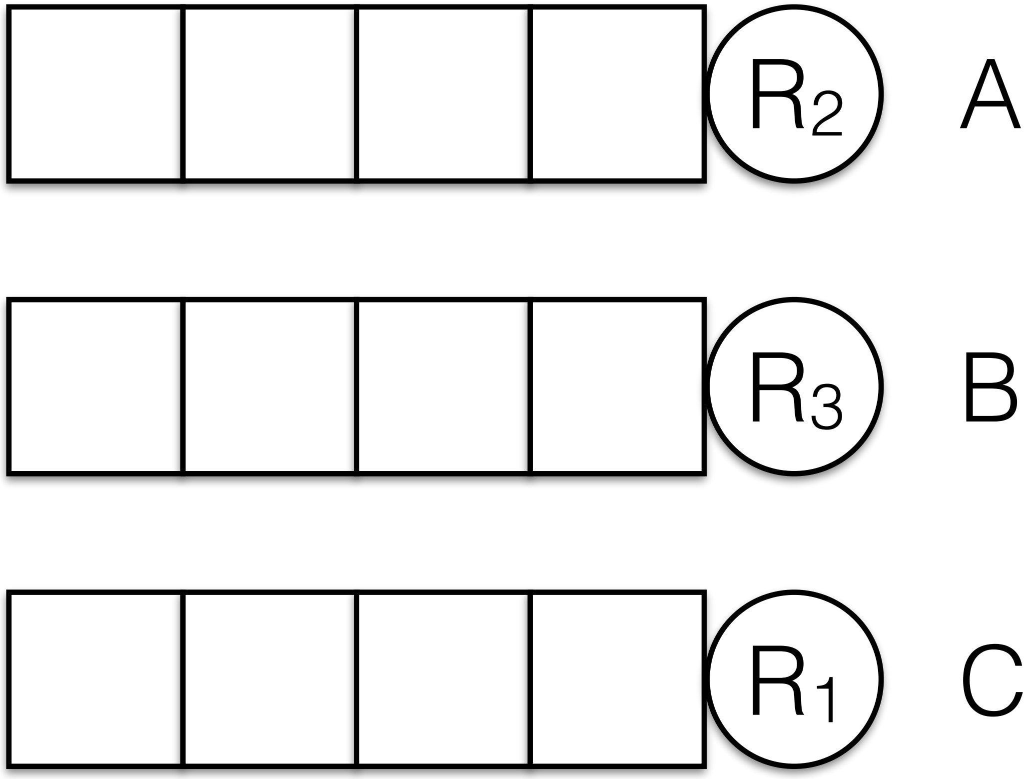
RSM

The key problem: later-issued requests hold resources that are requested in a nested fashion.



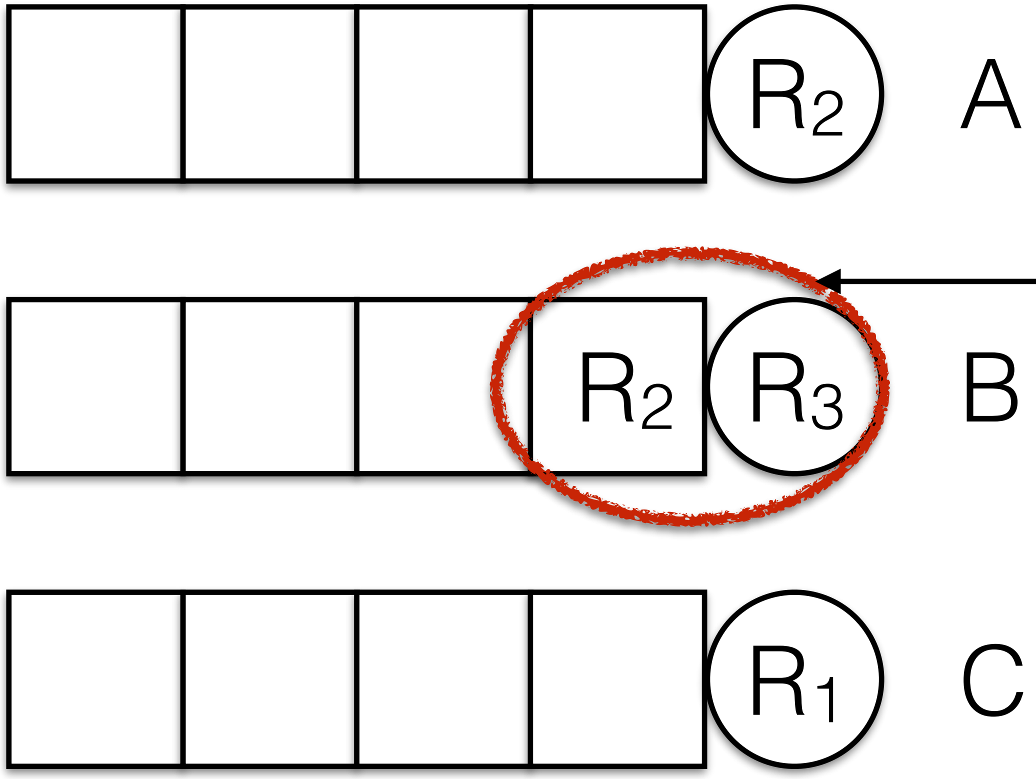
RSM

The key problem: later-issued requests hold resources that are requested in a nested fashion.



RSM

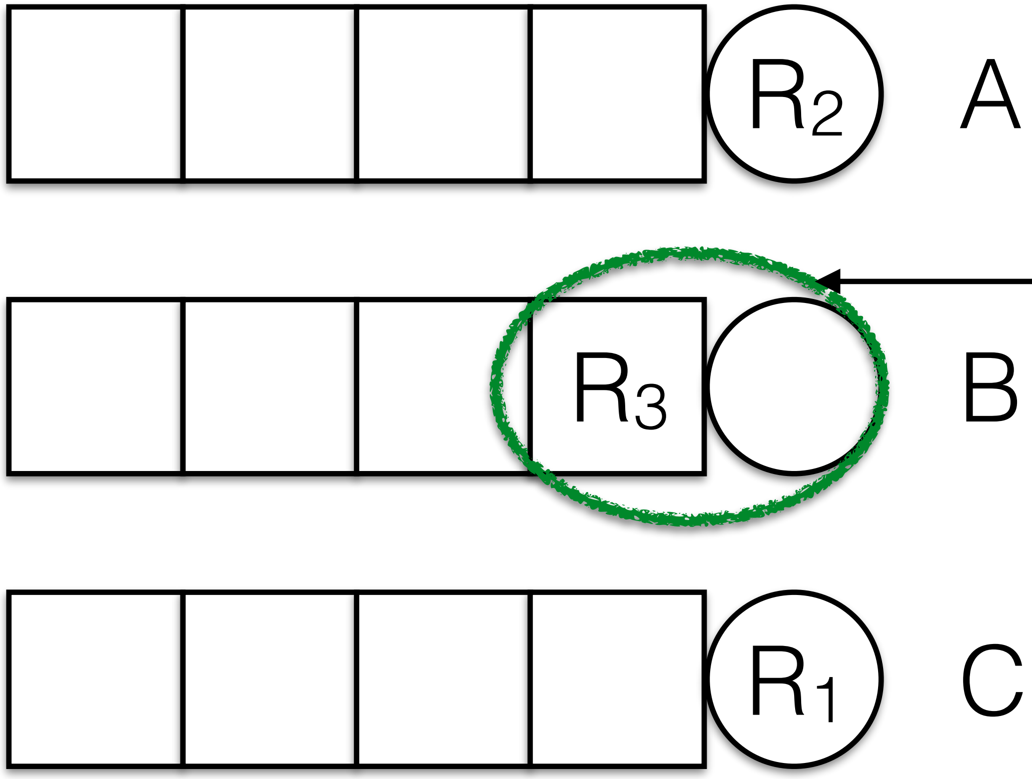
The key problem: later-issued requests hold resources that are requested in a nested fashion.



The earlier-issued R_2 is blocked by the later-issued R_3.

RSM

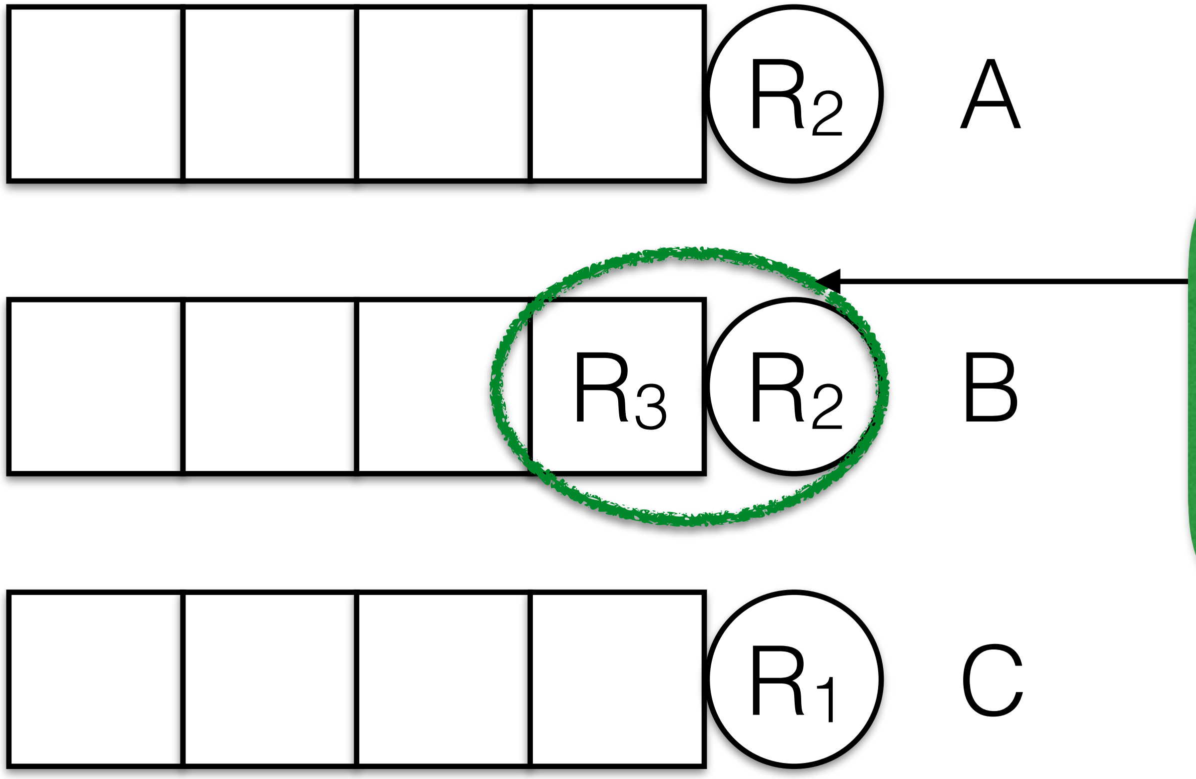
In the RNLP, this problem is avoided by preventing later-issued requests from acquiring resources that may be requested in a nested fashion.



Now R_2 can acquire B immediately.

RSM

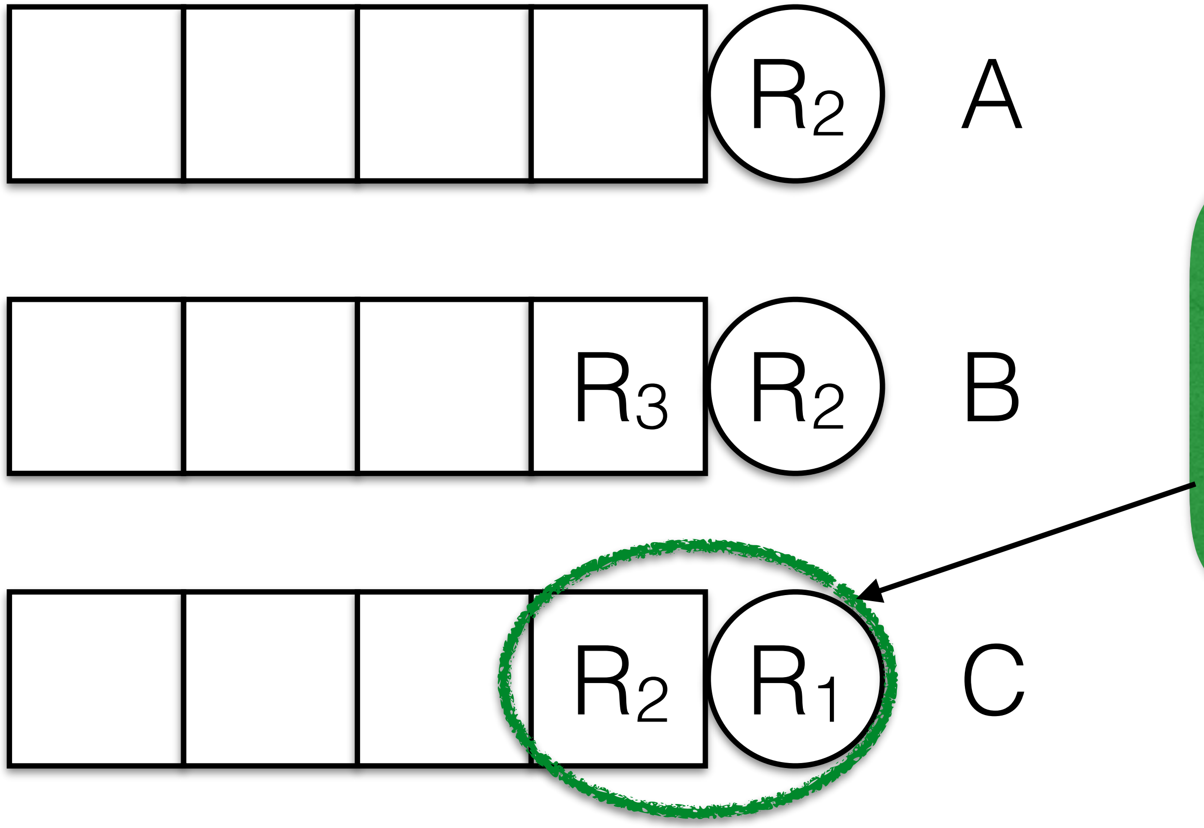
In the RNLP, this problem is avoided by preventing later-issued requests from acquiring resources that may be requested in a nested fashion.



Now R_2 can acquire B immediately.

RSM

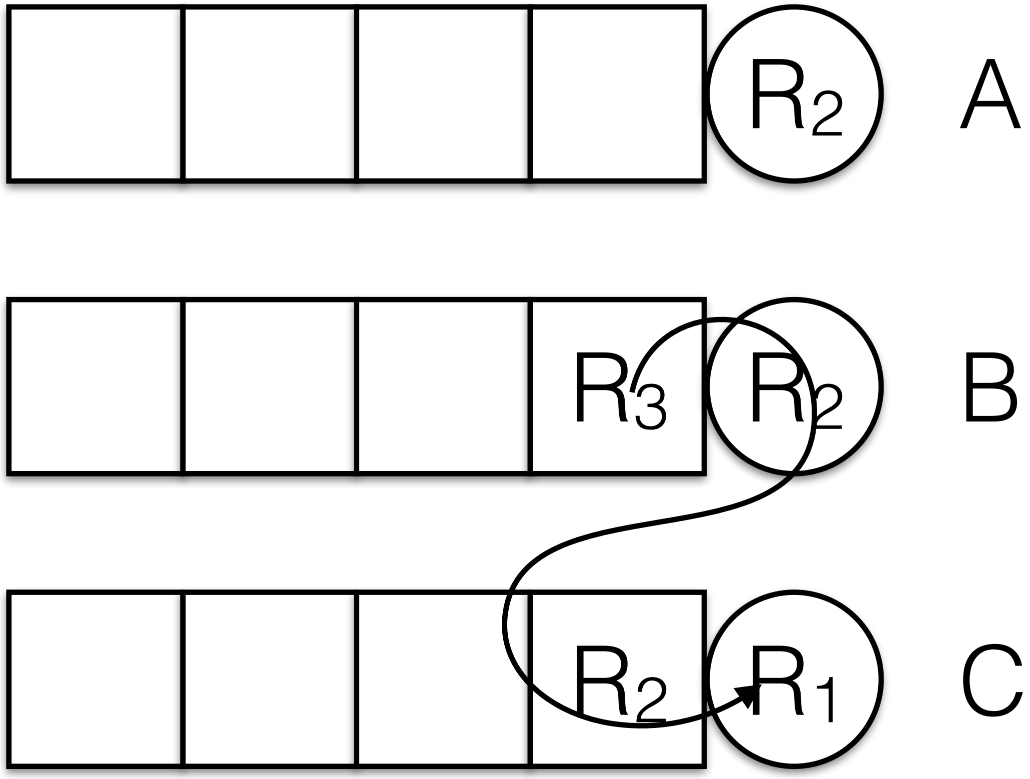
In the RNLP, this problem is avoided by preventing later-issued requests from acquiring resources that may be requested in a nested fashion.



Note that R_2 may still block on R_1. This is ok; R_1 was issued earlier than R_2.

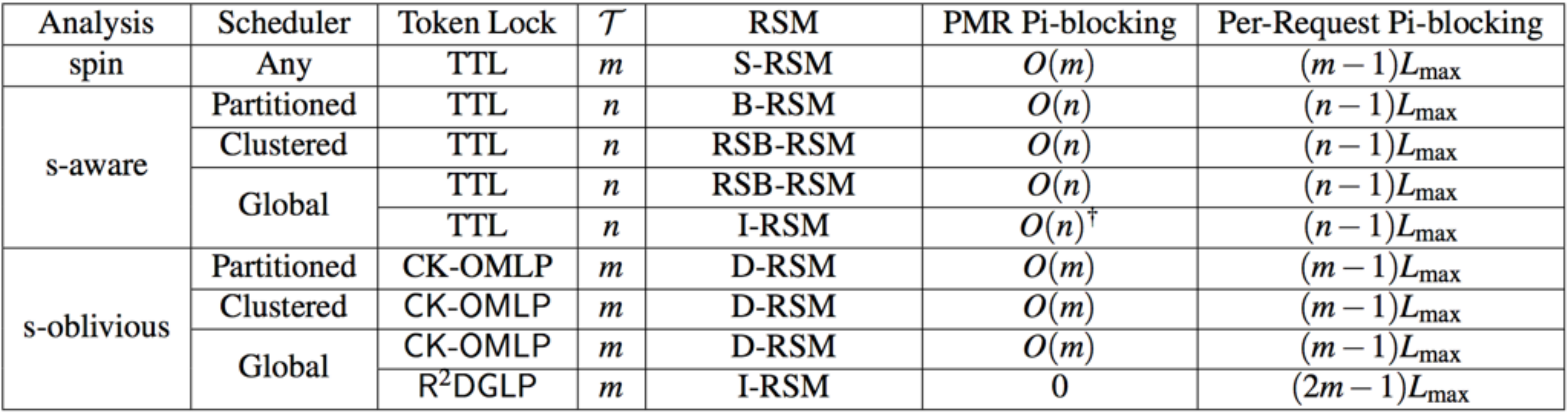
RSM

In the RNLP, this problem is avoided by preventing later-issued requests from acquiring resources that may be requested in a nested fashion.



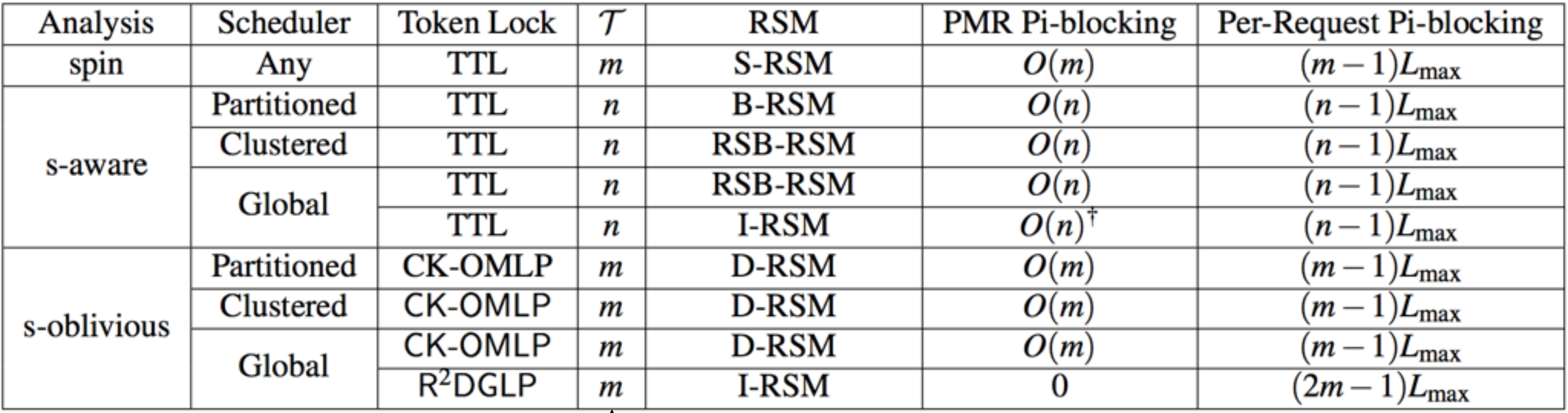
Transitive blocking is still possible, and must be considered.

Optimality Results



Analysis	Scheduler	Token Lock	$\mathcal{T}$	RSM	PMR Pi-blocking	Per-Request Pi-blocking
spin	Any	TTL	m	S-RSM	$O(m)$	$(m-1)L_{\max}$
s-aware	Partitioned	TTL	n	B-RSM	$O(n)$	$(n-1)L_{\max}$
	Clustered	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
	Global	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
		TTL	n	I-RSM	$O(n)^{\dagger}$	$(n-1)L_{\max}$
s-oblivious	Partitioned	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Clustered	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Global	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
		R^2DGLP	m	I-RSM	0	$(2m-1)L_{\max}$

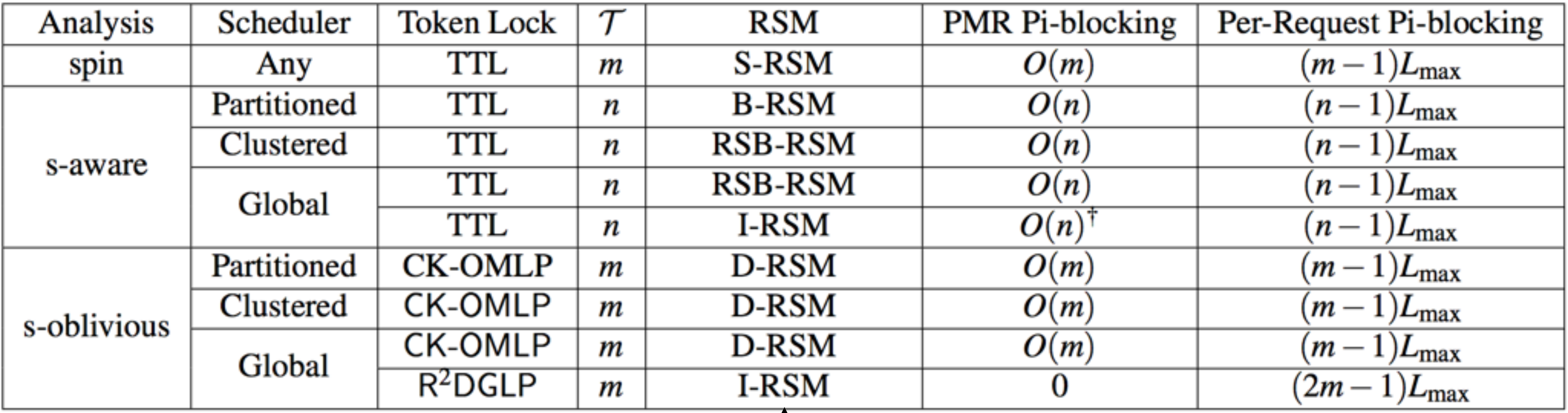
Optimality Results



Analysis	Scheduler	Token Lock	$\mathcal{T}$	RSM	PMR Pi-blocking	Per-Request Pi-blocking
spin	Any	TTL	m	S-RSM	$O(m)$	$(m-1)L_{\max}$
s-aware	Partitioned	TTL	n	B-RSM	$O(n)$	$(n-1)L_{\max}$
	Clustered	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
	Global	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
		TTL	n	I-RSM	$O(n)^{\dagger}$	$(n-1)L_{\max}$
s-oblivious	Partitioned	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Clustered	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Global	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
		R^{2}DGLP	m	I-RSM	0	$(2m-1)L_{\max}$

Number of tokens

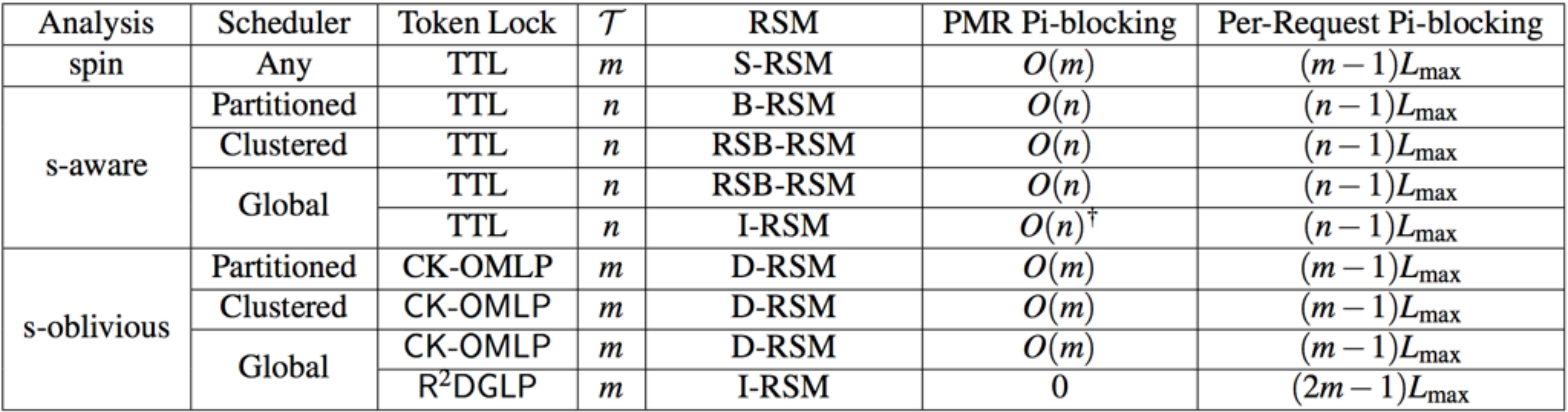
Optimality Results



Analysis	Scheduler	Token Lock	$\mathcal{T}$	RSM	PMR Pi-blocking	Per-Request Pi-blocking
spin	Any	TTL	m	S-RSM	$O(m)$	$(m-1)L_{\max}$
s-aware	Partitioned	TTL	n	B-RSM	$O(n)$	$(n-1)L_{\max}$
	Clustered	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
	Global	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
		TTL	n	I-RSM	$O(n)^{\dagger}$	$(n-1)L_{\max}$
s-oblivious	Partitioned	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Clustered	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Global	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
		R²DGLP	m	I-RSM	0	$(2m-1)L_{\max}$

Progress mechanism used

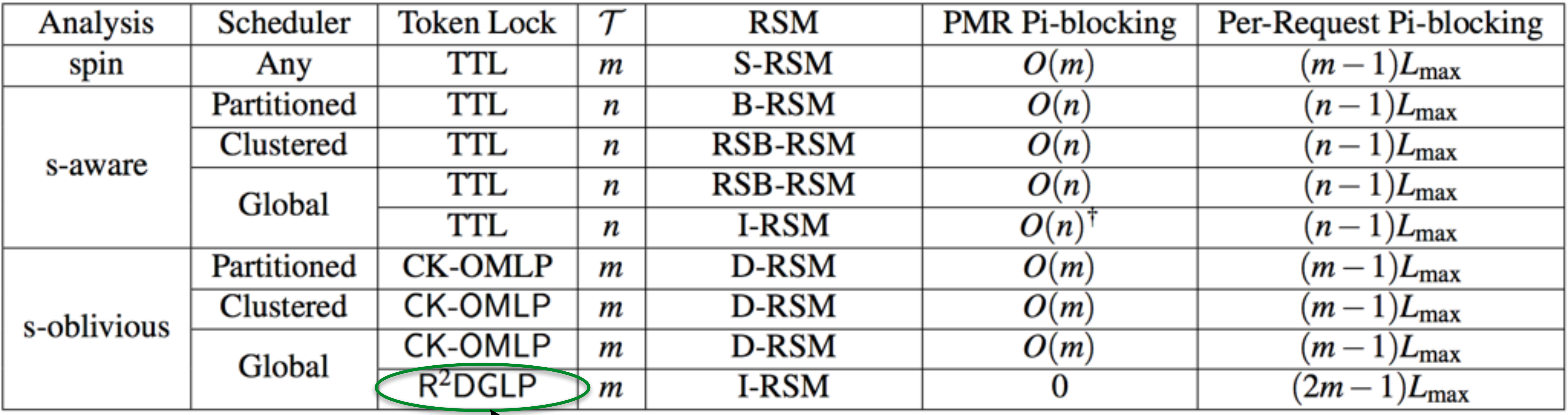
Optimality Results



Analysis	Scheduler	Token Lock	$\mathcal{T}$	RSM	PMR Pi-blocking	Per-Request Pi-blocking
spin	Any	TTL	m	S-RSM	$O(m)$	$(m-1)L_{\max}$
s-aware	Partitioned	TTL	n	B-RSM	$O(n)$	$(n-1)L_{\max}$
	Clustered	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
	Global	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
		TTL	n	I-RSM	$O(n)^\dagger$	$(n-1)L_{\max}$
s-oblivious	Partitioned	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Clustered	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Global	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
		R^{2}DGLP	m	I-RSM	0	$(2m-1)L_{\max}$

These blocking bounds match those of previous optimal coarse-grained protocols.

Optimality Results



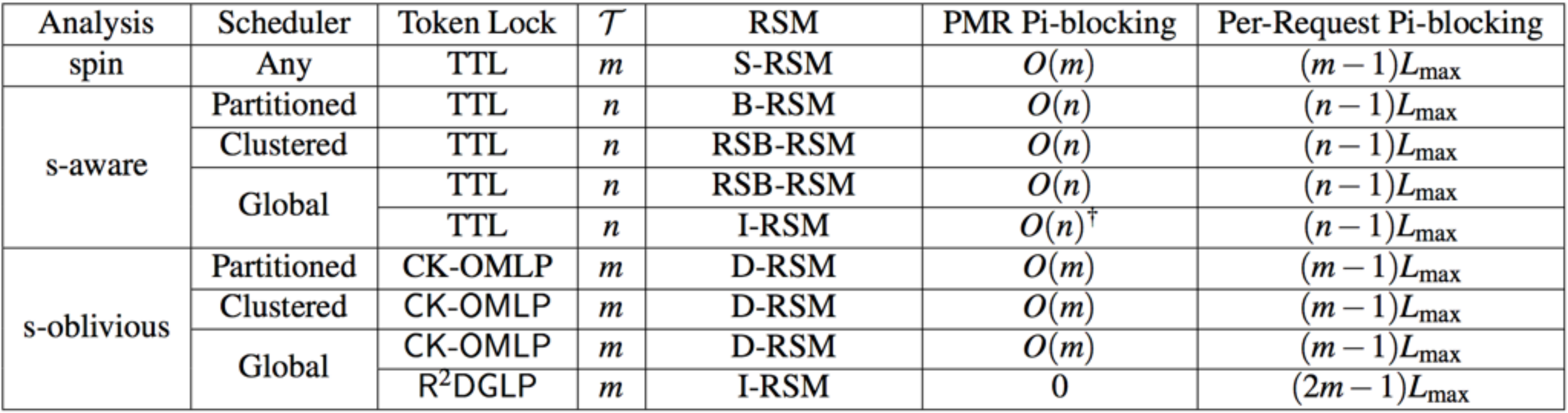
Analysis	Scheduler	Token Lock	$\mathcal{T}$	RSM	PMR Pi-blocking	Per-Request Pi-blocking
spin	Any	TTL	m	S-RSM	$O(m)$	$(m-1)L_{\max}$
s-aware	Partitioned	TTL	n	B-RSM	$O(n)$	$(n-1)L_{\max}$
	Clustered	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
	Global	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
		TTL	n	I-RSM	$O(n)^{\dagger}$	$(n-1)L_{\max}$
s-oblivious	Partitioned	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Clustered	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Global	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
		R^{2}DGLP	m	I-RSM	0	$(2m-1)L_{\max}$

A new *k*-exclusion locking protocol also proposed in this dissertation.

---

*B. Ward, G. Elliott and J. Anderson. *"Replica-Request Priority Donation: A Real-Time Progress Mechanism for Global Locking Protocols."* RTCSA '12

Optimality Results



Analysis	Scheduler	Token Lock	$\mathcal{T}$	RSM	PMR Pi-blocking	Per-Request Pi-blocking
spin	Any	TTL	m	S-RSM	$O(m)$	$(m-1)L_{\max}$
s-aware	Partitioned	TTL	n	B-RSM	$O(n)$	$(n-1)L_{\max}$
	Clustered	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
	Global	TTL	n	RSB-RSM	$O(n)$	$(n-1)L_{\max}$
		TTL	n	I-RSM	$O(n)^{\dagger}$	$(n-1)L_{\max}$
s-oblivious	Partitioned	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Clustered	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
	Global	CK-OMLP	m	D-RSM	$O(m)$	$(m-1)L_{\max}$
		R^2DGLP	m	I-RSM	0	$(2m-1)L_{\max}$

The problem of fine-grained locking in multiprocessor real-time systems stood open for over 20 years! The RNLP solves this problem optimally under all common analysis assumptions and platform configurations.

Outline

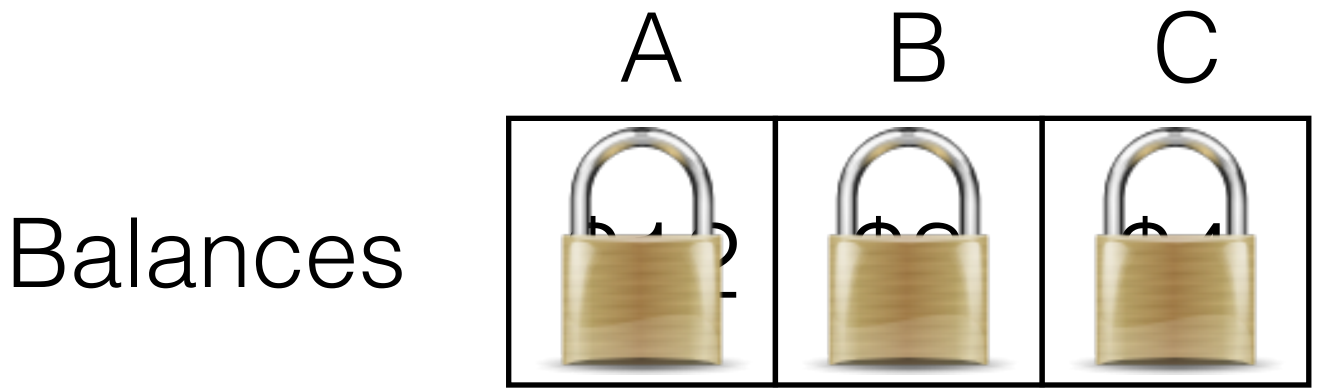
- Introduction
- **Memory Objects**
  - Fine-grained mutex locks (RNLP)
  - **Fine-grained reader/writer locks (R/W RNLP)**
- Hardware Resources
  - Preemptive mutual exclusion
  - Half-protected exclusion
- Conclusions

Motivation

```
def transaction(from_acct, to_acct, amount)
 lock(from_acct)
 if(balances[from_acct] < amount)
```

What if there are other routines that need only read the current balance of some accounts?

```
  unlock(to_acct)
  balances[from_acct] = balances[from_acct] - amount
 unlock(from_acct)
```



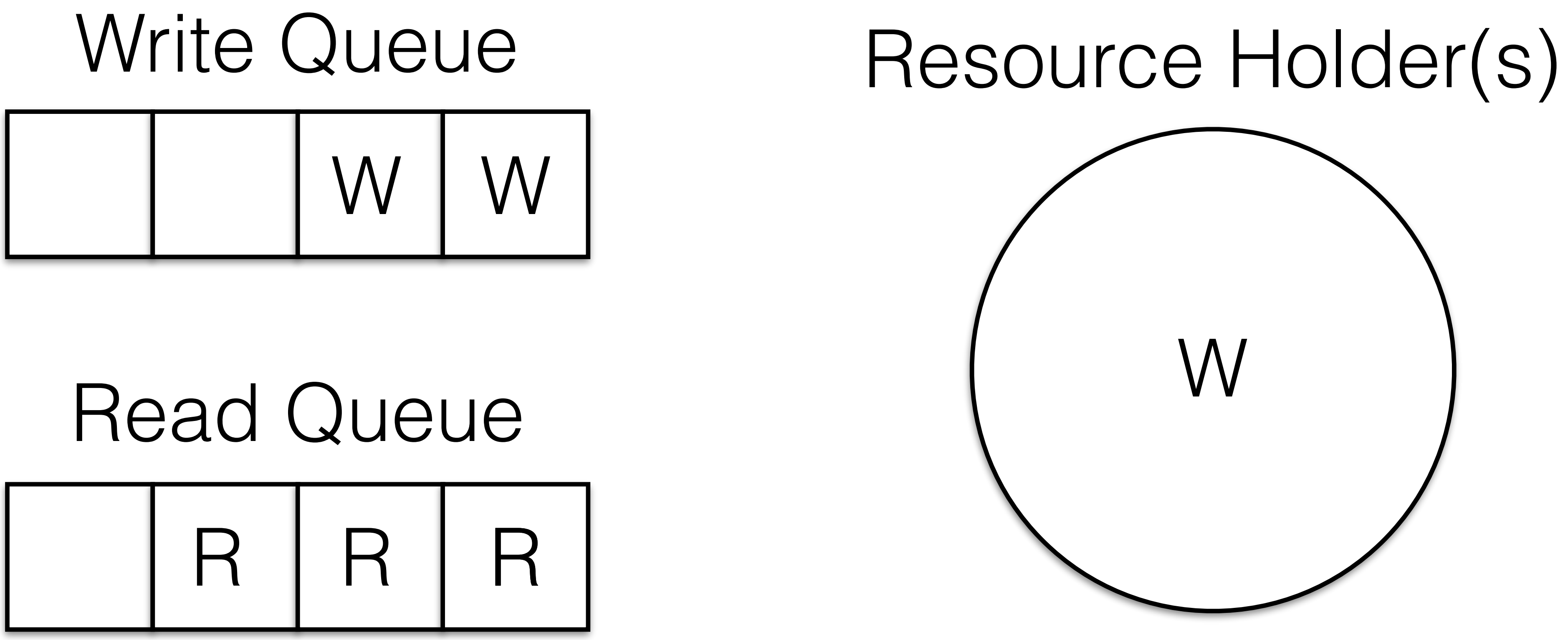
Reader/Writer Locking

- **Reader/writer locking:**
  - Reads can execute in parallel.
  - Writes require mutual exclusion.
- Reader/writer locking reduces blocking when reads are common.
- How do we extend the RNLP to support fine-grained reader/writer locking?

*B. Ward and J. Anderson. *"Multi-Resource Real-Time Reader/Writer Locks for Multiprocessors."* IPDPS '14

Phase-Fair Locking*

Key idea: read phases and write phases “take turns.”



Write Queue

W W

Read Queue

R R R

Resource Holder(s)

W

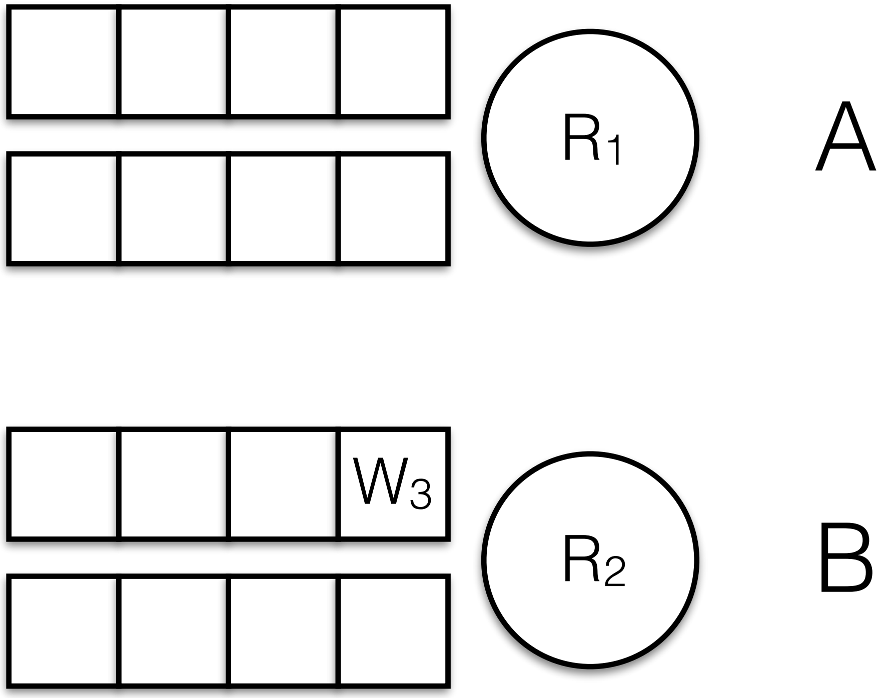
Result: O(1) read blocking, O(m) write blocking.

*B. Brandenburg and J. Anderson. *“Reader-Writer Synchronization for Shared-Memory Multiprocessor Real-Time Systems”* ECRTS ‘09

One Challenge

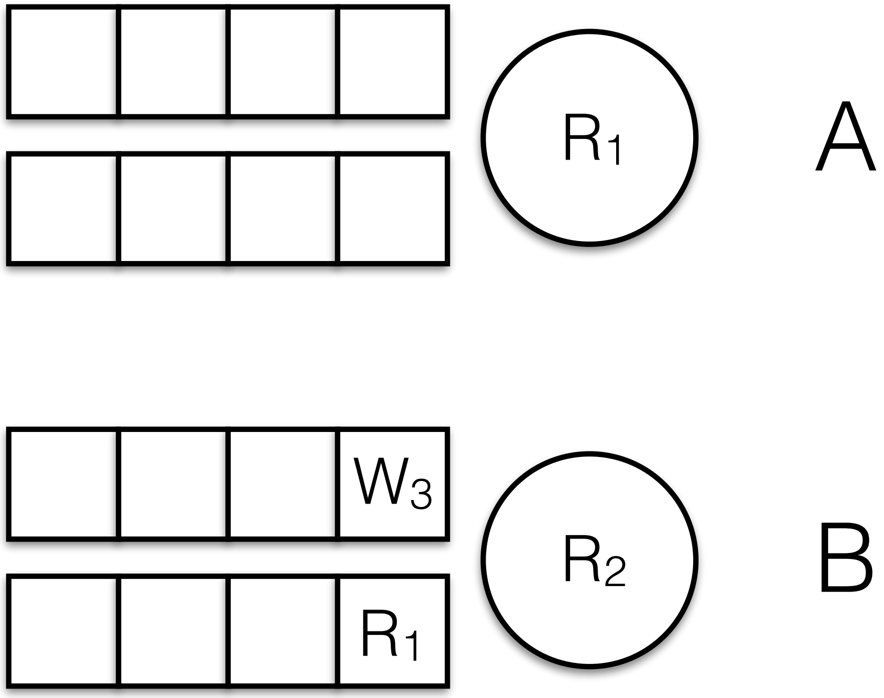
- In the RNLP, a request is never blocked by another later-issued request.
- To achieve O(1) reader blocking, later-issued read requests must “cut ahead” of earlier-issued write requests.

Multi-Resources Phases



What happens if R_1 issues a nested read request for B?

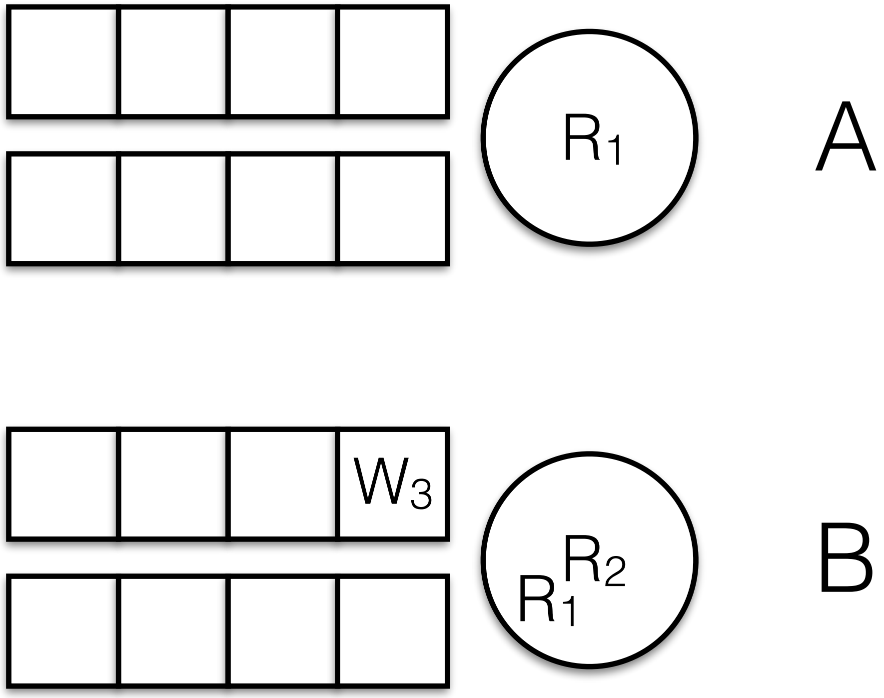
Multi-Resources Phases



Per-resource phase-fair logic dictates that it should wait.

But R_1 would be blocked by later-issued read and write requests!

Multi-Resources Phases



What if R_1 is allowed to cut ahead?

This may increase the read-phase duration.

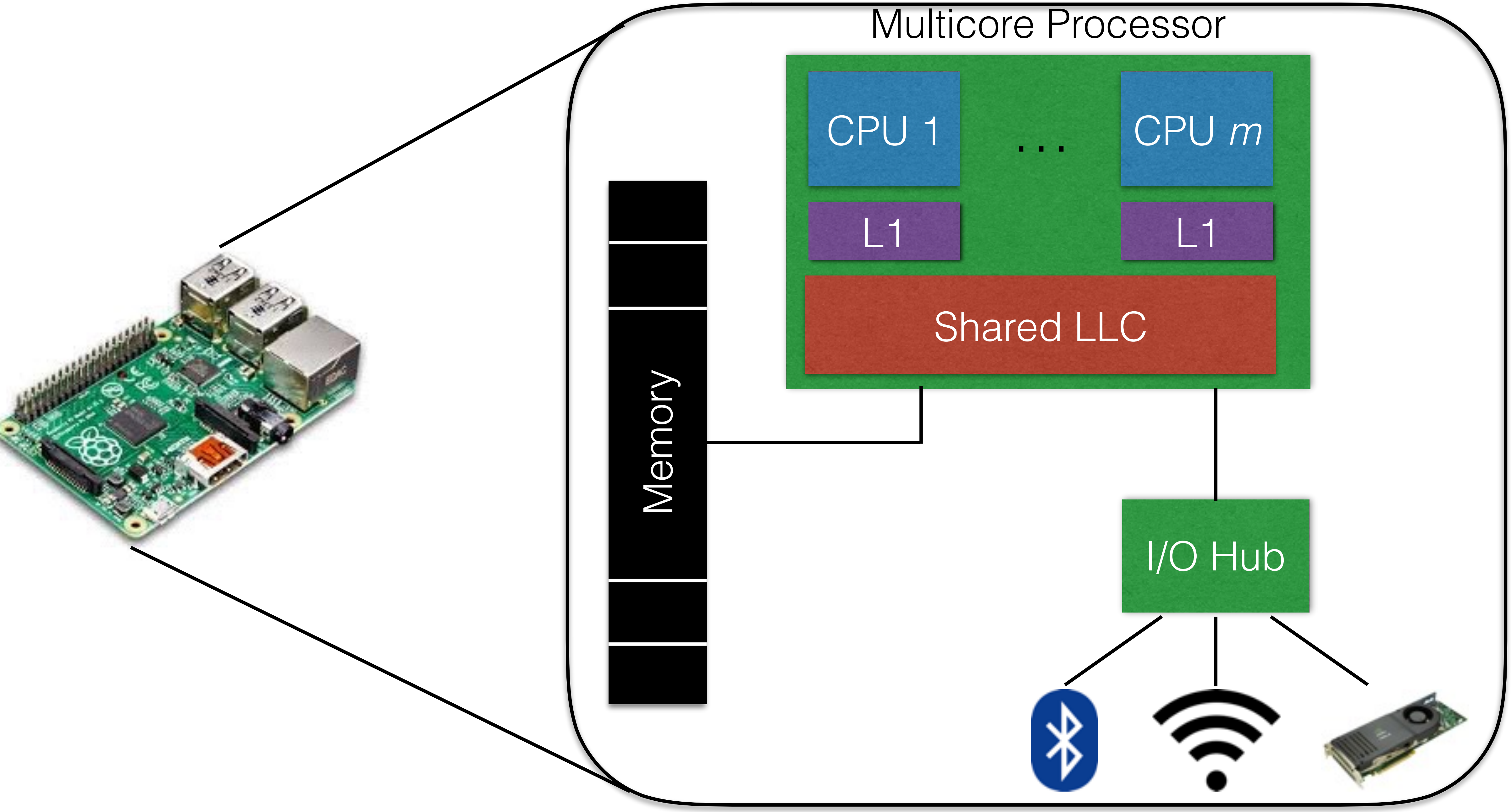
R/W RNLP

- These issues are resolved through a concept called **entitlement**.
  - Entitlement resolves the dilemma of which phase goes next.
- R/W RNLP Results:
  - First fine-grained multiprocessor real-time R/W lock.
  - O(1) reader pi-blocking.
  - O(m) writer pi-blocking.

Outline

- Introduction
- Memory Objects
    - Fine-grained mutex locks (RNLP)
    - Fine-grained reader/writer locks (R/W RNLP)
- **Hardware Resources**
    - **Preemptive mutual exclusion**
    - Half-protected exclusion
- Conclusions

Modern Multicore Architectures



Multicore Processor

CPU 1 … CPU *m*

L1

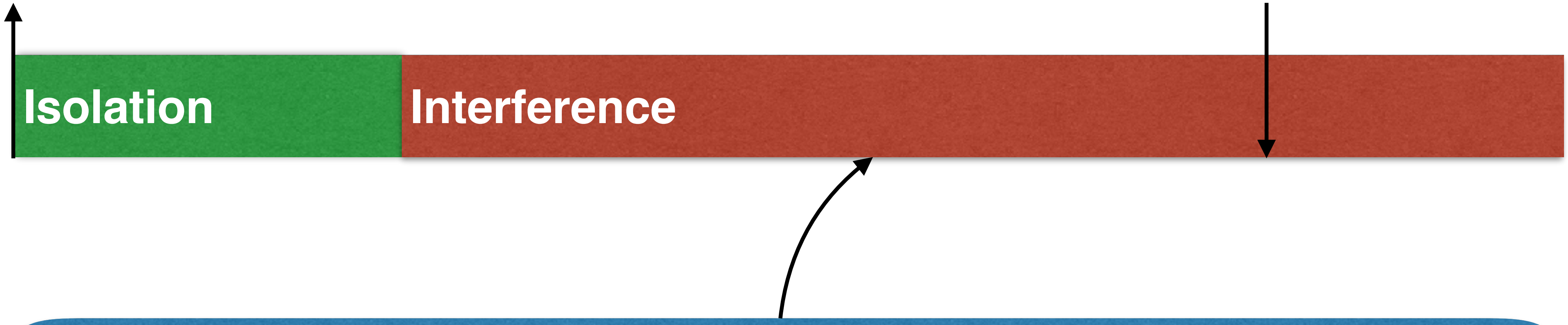
L1

Shared LLC

Memory

I/O Hub

Hardware-based Timing Interference



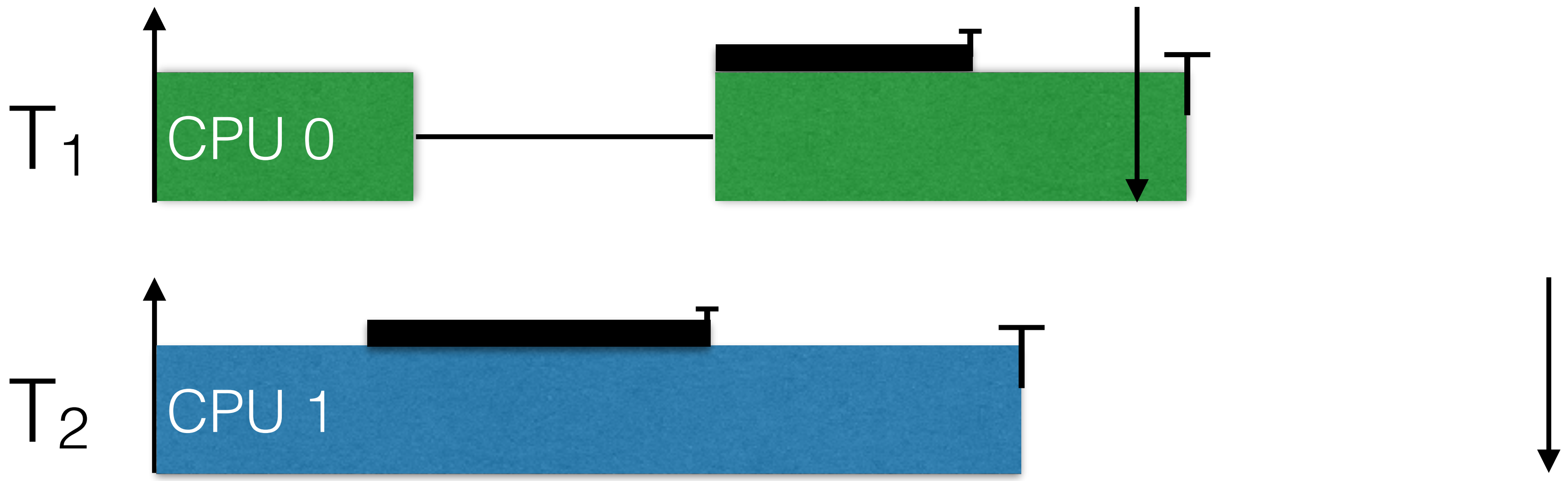
Interference caused by other tasks concurrently scheduled on other processors

Hardware-based interference can entirely negate the benefits of having additional cores. *De facto* industry standard is to disable all but one core!

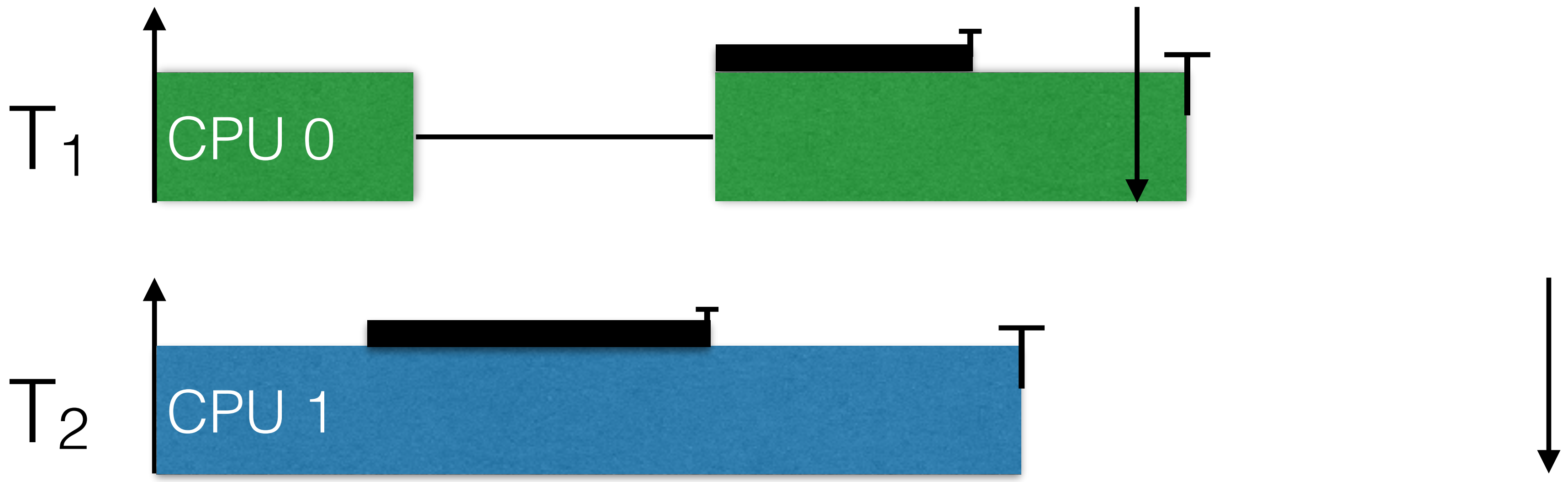
Locking Hardware Resources

- Locking hardware resources improves predictability, but is not (necessarily) required for logical correctness.
- Example: Shared cache.
    - **Problem**: a job may evict cached data of another concurrently executing job.
    - **Goal**: “lock” cache resources to prevent such evictions, thereby improving timing predictability.
    - **Observation**: cache critical sections can be safely preempted.

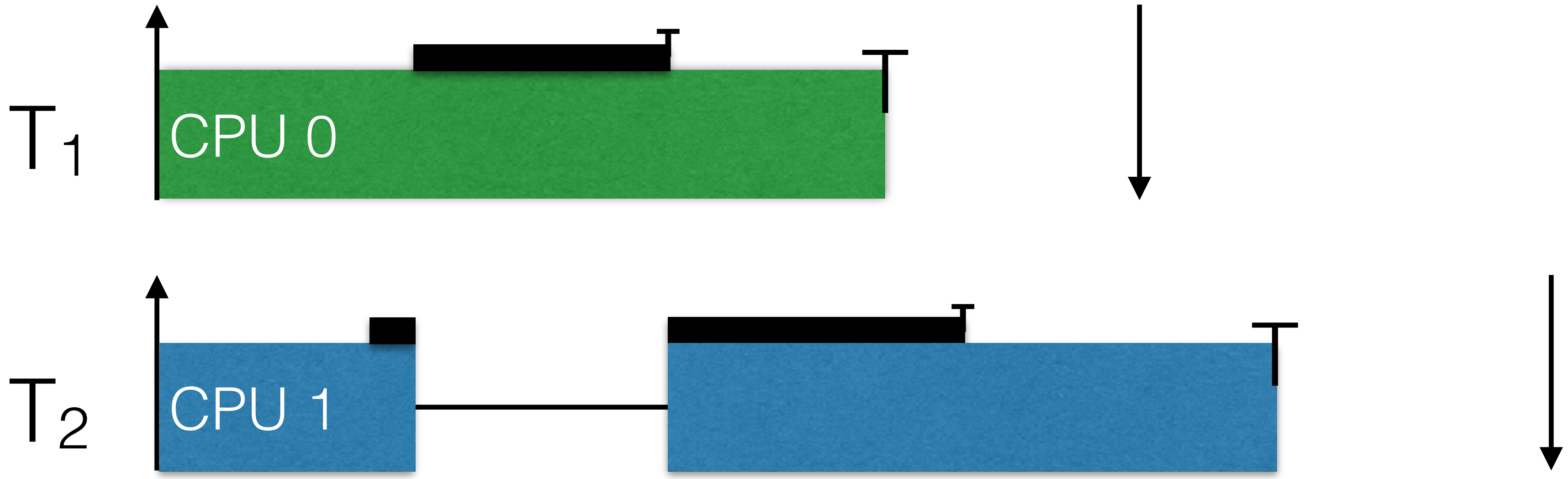
Non-Preemptive Mutual Exclusion



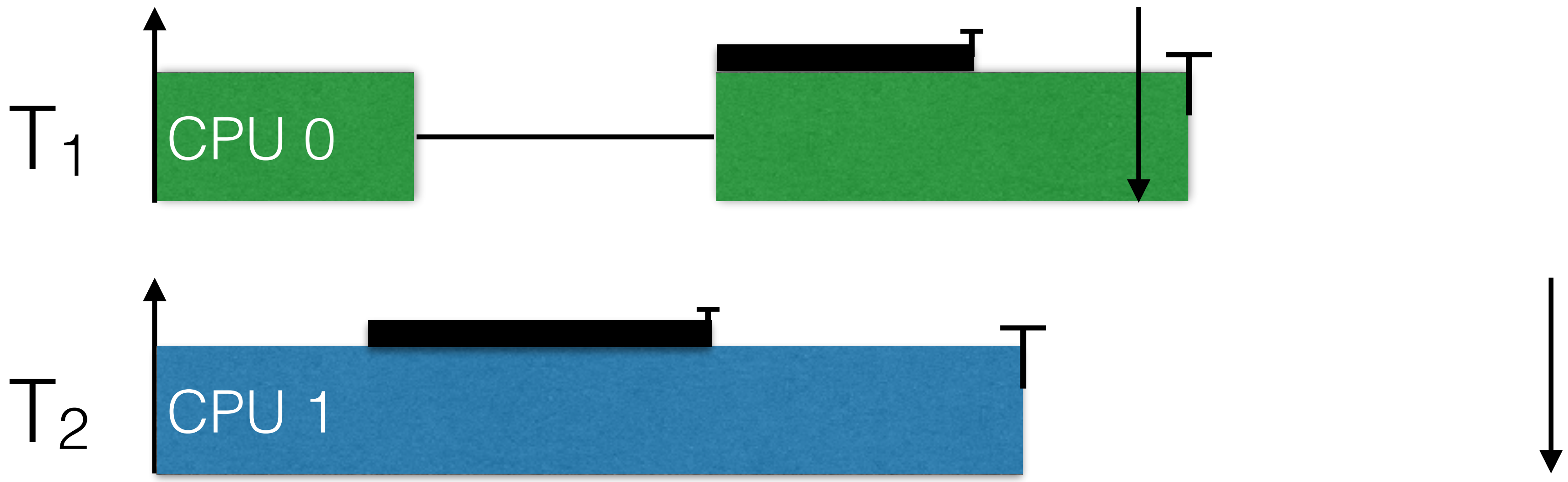
Non-Preemptive Mutual Exclusion



Preemptive Mutual Exclusion

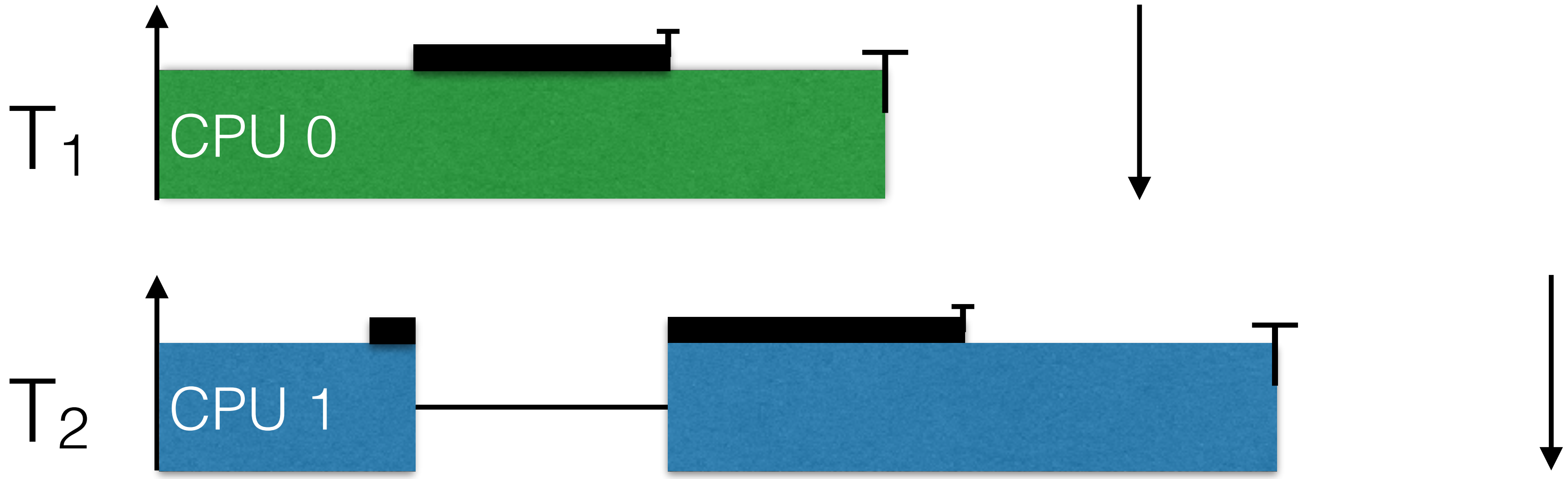


Non-Preemptive Mutual Exclusion



Preemptivity allows the higher-priority critical section to preempt the lower-priority one, which prevents the deadline miss.

Preemptive Mutual Exclusion

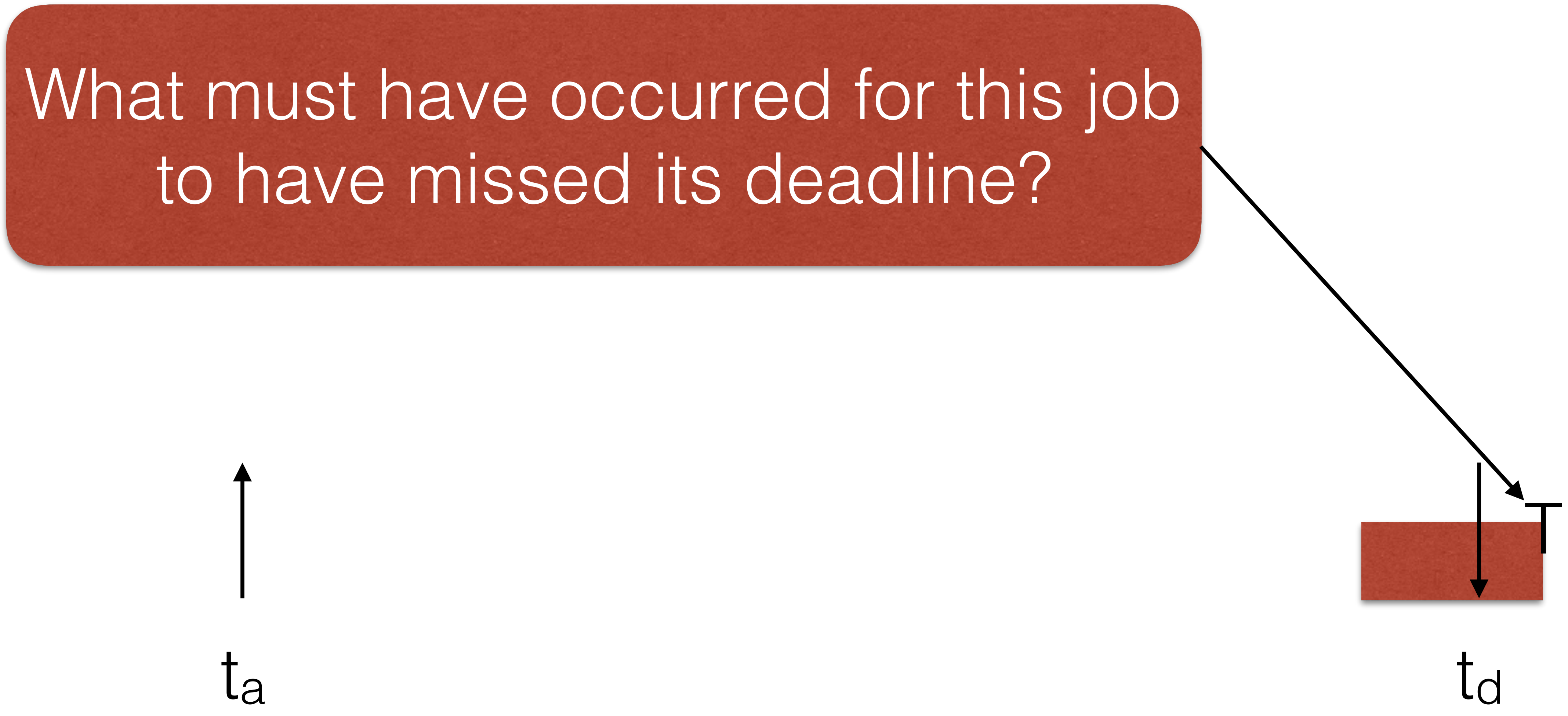


Preemptive Mutual Exclusion*

- At most one task may execute a critical section at any time, but resource preemptions are allowed.
- This problem is unique to multiprocessors.
- Potential applications:
  - Arbitrating bus accesses, *e.g.*, memory bus.

*B. Ward. *“Relaxing Resource-Sharing Constraints for Improved Hardware Management and Schedulability.”* RTSS ‘15

Schedulability Analysis

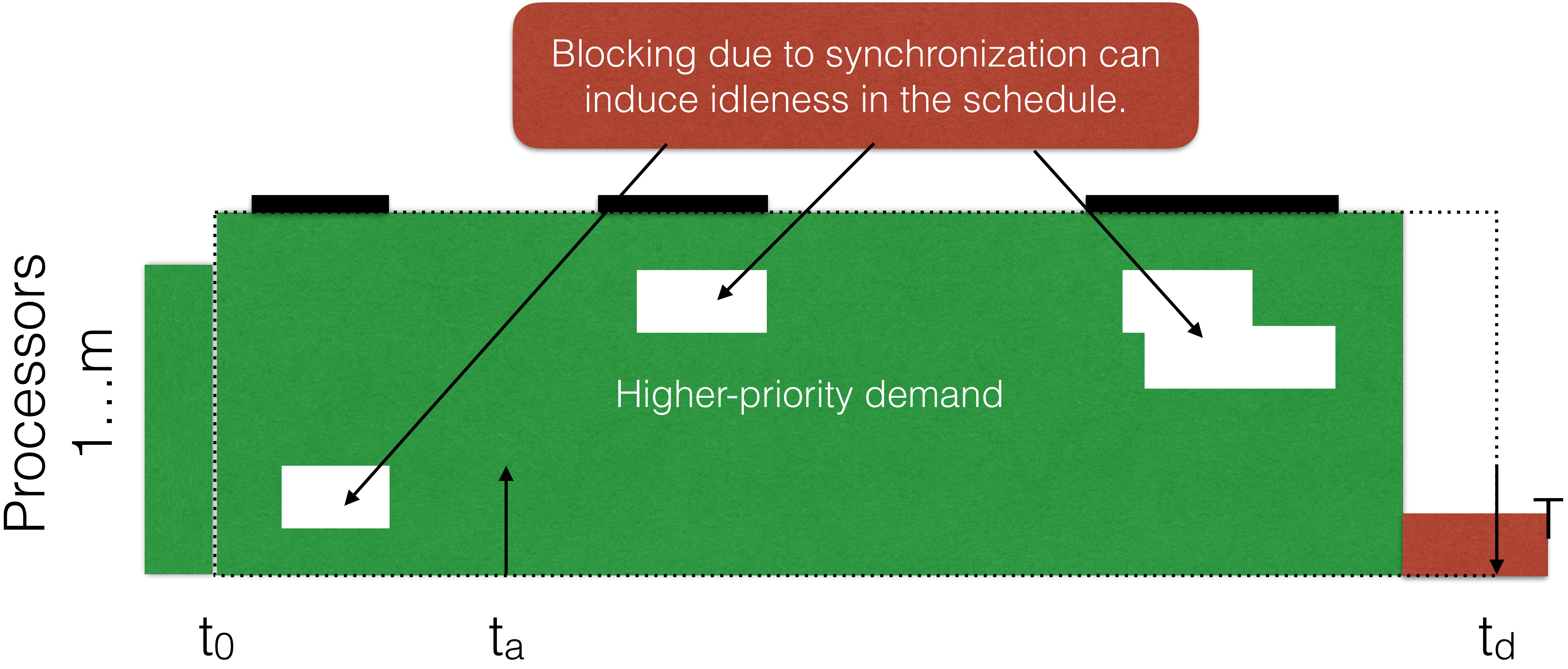


What must have occurred for this job to have missed its deadline?

Schedulability Analysis

S. Baruah, "*Techniques for Multiprocessor Global Schedulability Analysis*", RTSS '07.

Idleness Analysis



Blocking due to synchronization can induce idleness in the schedule.

Idleness

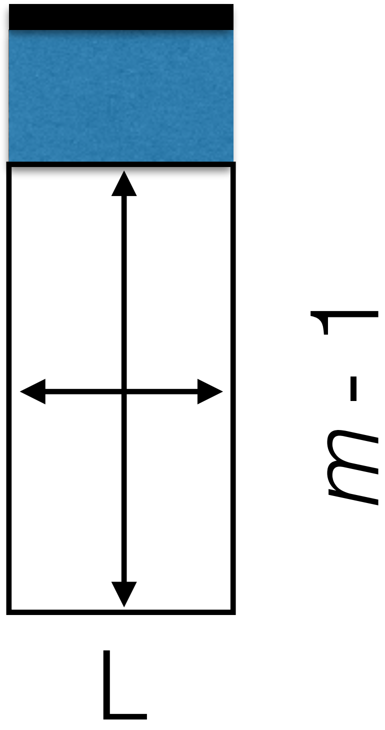
- Idleness in this context is caused by blocking.
- Traditionally, blocking modeled by suspensions.

Depending upon the analysis assumptions and locking protocol (2m-1)L or (n-1)L.

How long can this request be blocked?

Idleness Analysis

How much idleness can this request induce in the schedule?



Idleness Analysis

New analysis question:

Is there sufficient demand <u>plus induced idleness</u> to cause the deadline miss?

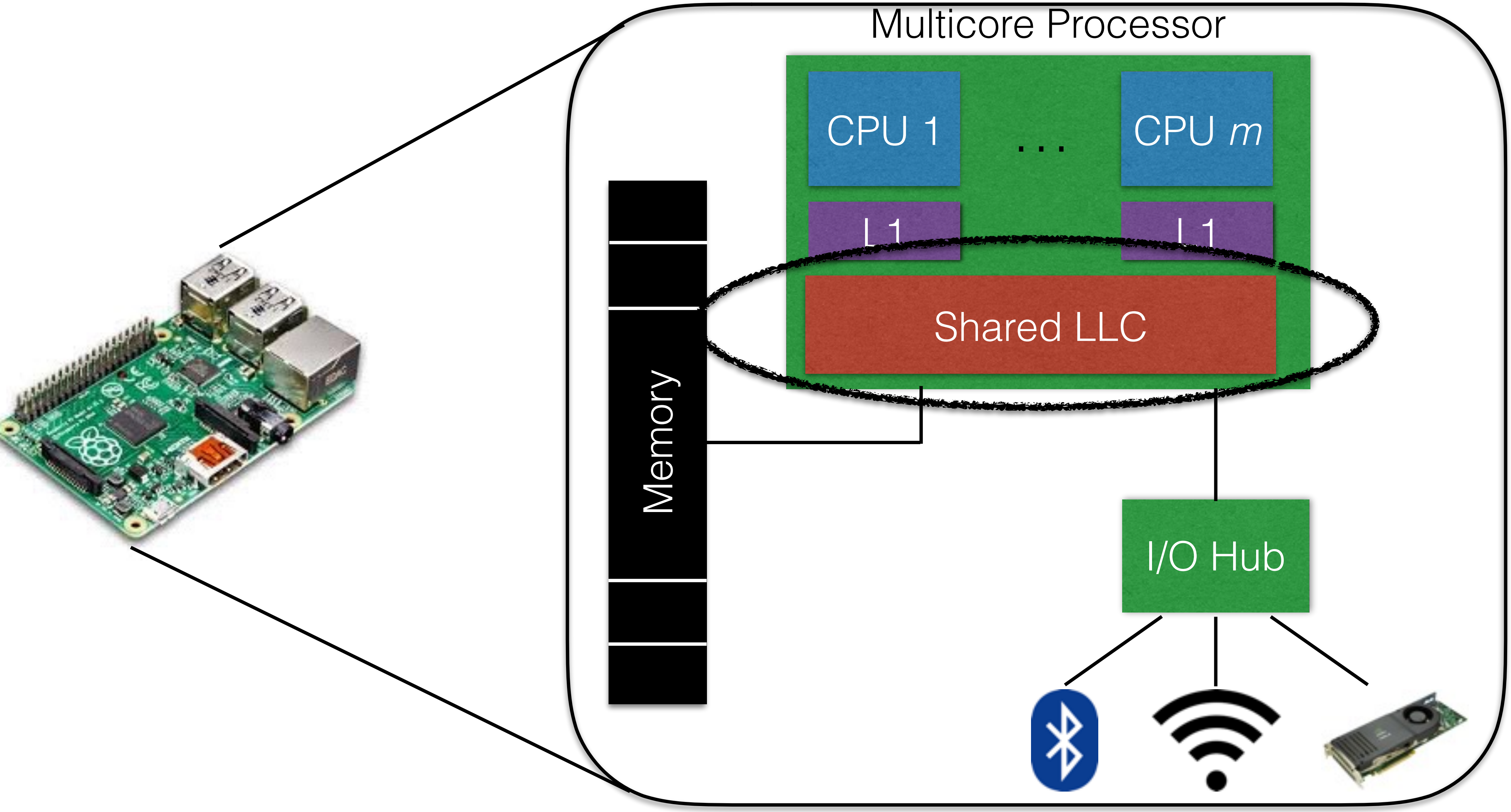
Idleness Analysis

- **Advantages:**
  - Simple.
  - Flexible.
  - Incomparable with previous blocking-analysis techniques.
- **Disadvantages:**
  - Can be pessimistic for high processor counts.
  - Does not incorporate protocol-specific information to reduce utilization loss.

Outline

- Introduction
- Memory Objects
  - Fine-grained mutex locks (RNLP)
  - Fine-grained reader/writer locks (R/W RNLP)
- **Hardware Resources**
  - Preemptive mutual exclusion
  - **Half-protected exclusion**
- Conclusions

Modern Multicore Architectures



Multicore Processor

CPU 1

. . .

CPU *m*

L1

L1

Shared LLC

Memory

I/O Hub

UCBs vs. ECBs

- In cache-related preemption-delay (CRPD) analysis, there are two classes of cache blocks*:
    - **Useful (UCB):** A cache block that is reused.
    - **Evicting (ECB):** A cache block that may be accessed, but may not be reused.
- Would like to protect UCBs from ECBs, but need not protect ECBs at all.

*Lee et al. *"Analysis of cache-related preemption delay in fixed-priority preemptive scheduling."* Transactions on Computing '98.
S. Altmeyer and C Maiza. *"Cache-related preemption delay via useful cache blocks: Survey and redefinition."* Journal of Systems Architecture '11.

Half-Protected Exclusion*

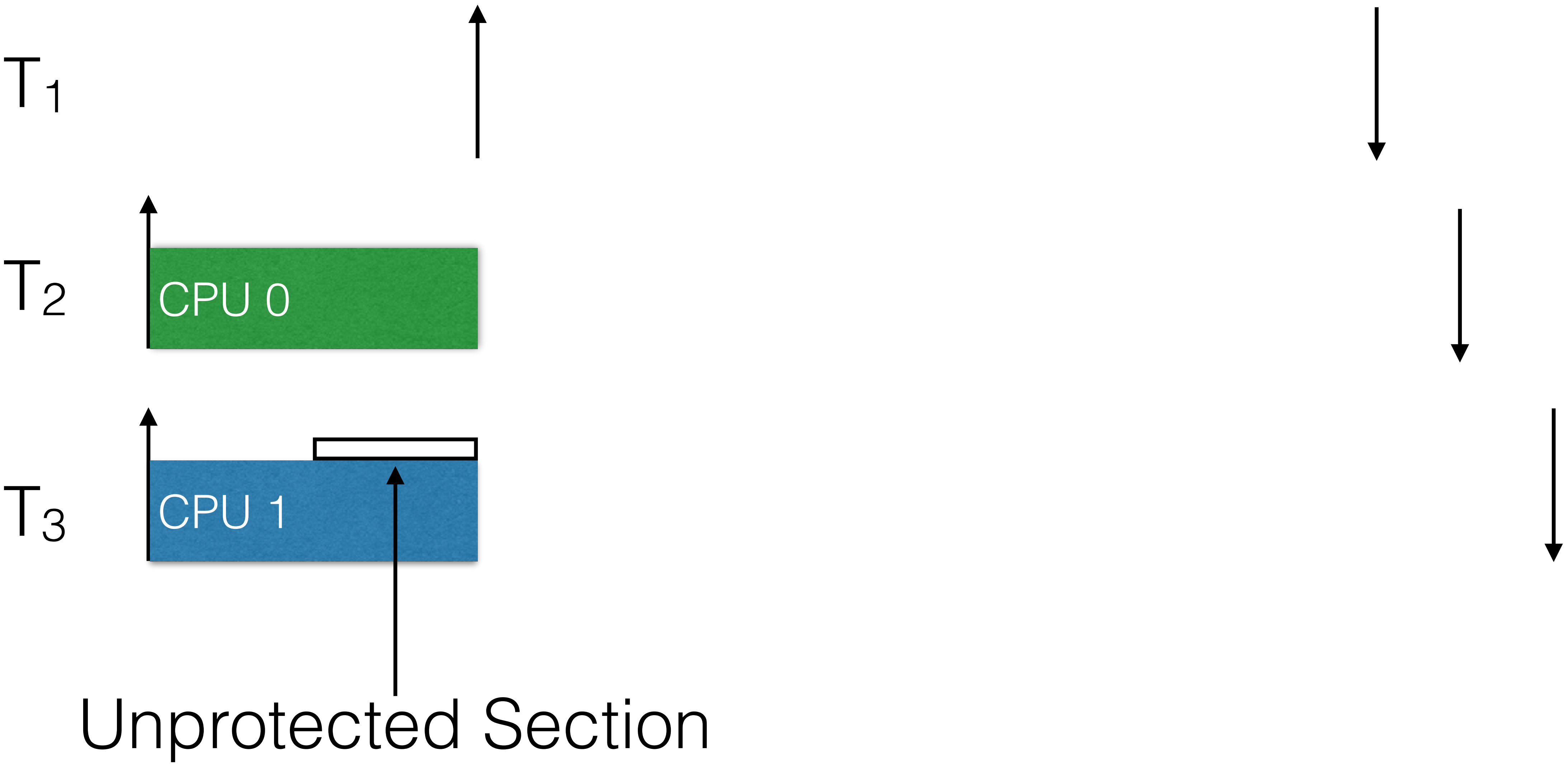
- **Protected sections:** require non-preemptive mutual exclusion w.r.t both protected and unprotected sections.
- **Unprotected sections:** may execute whenever a protected section does not.
- Can be seen as a weaker variant of reader/writer sharing: protected = write, unprotected ~ read.

*B. Ward. *“Relaxing Resource-Sharing Constraints for Improved Hardware Management and Schedulability.”* RTSS ‘15

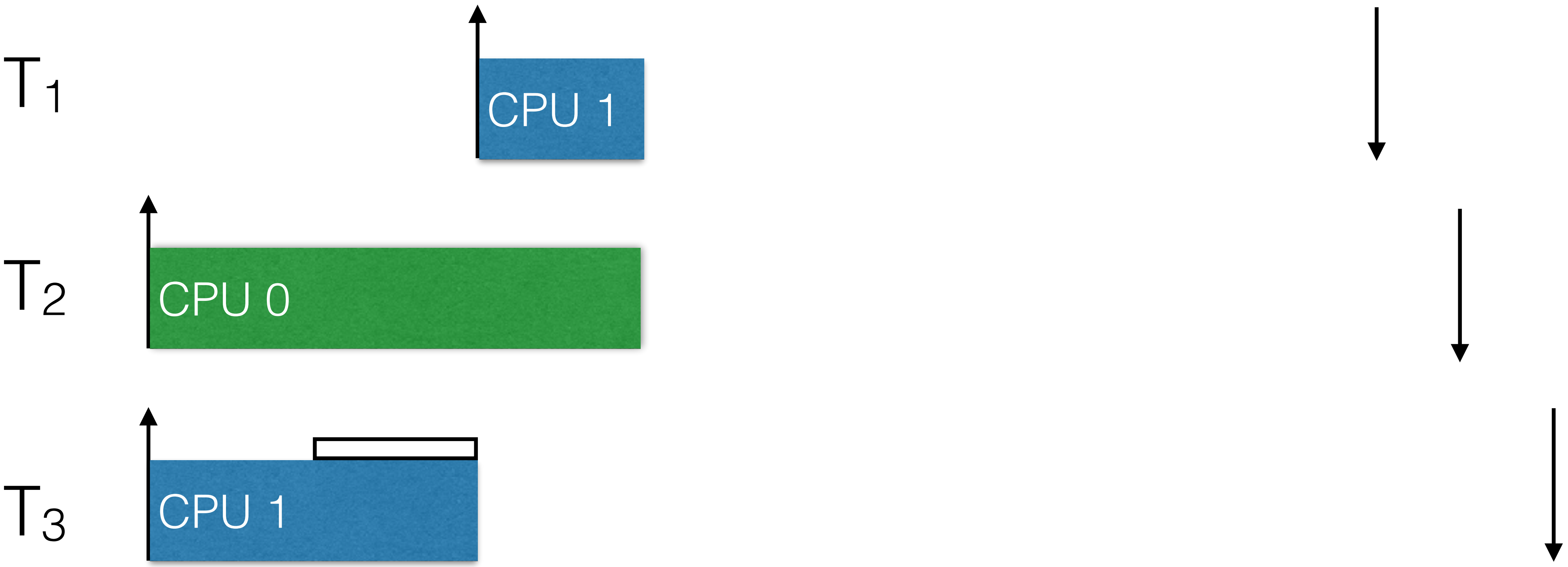
Half-Protected Example

Simple example:
Three tasks, T1, T2, and T3 scheduled on m = 2 processors.

Half-Protected Example

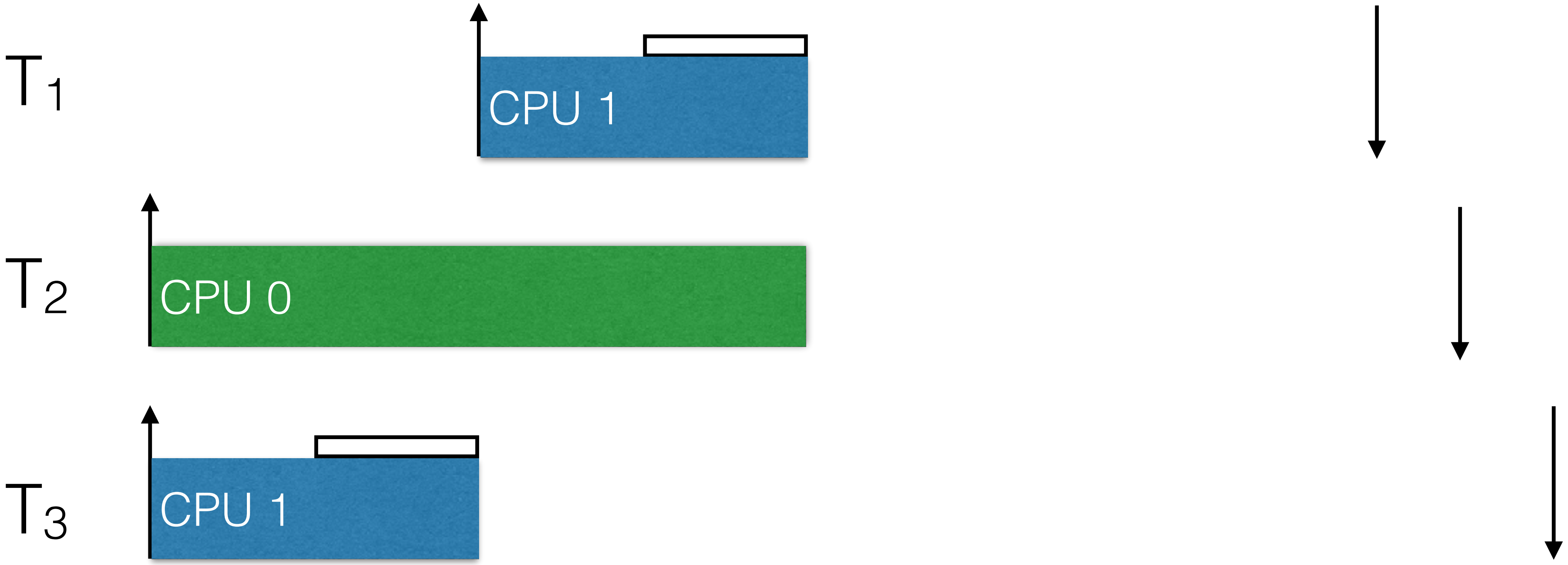


Half-Protected Example



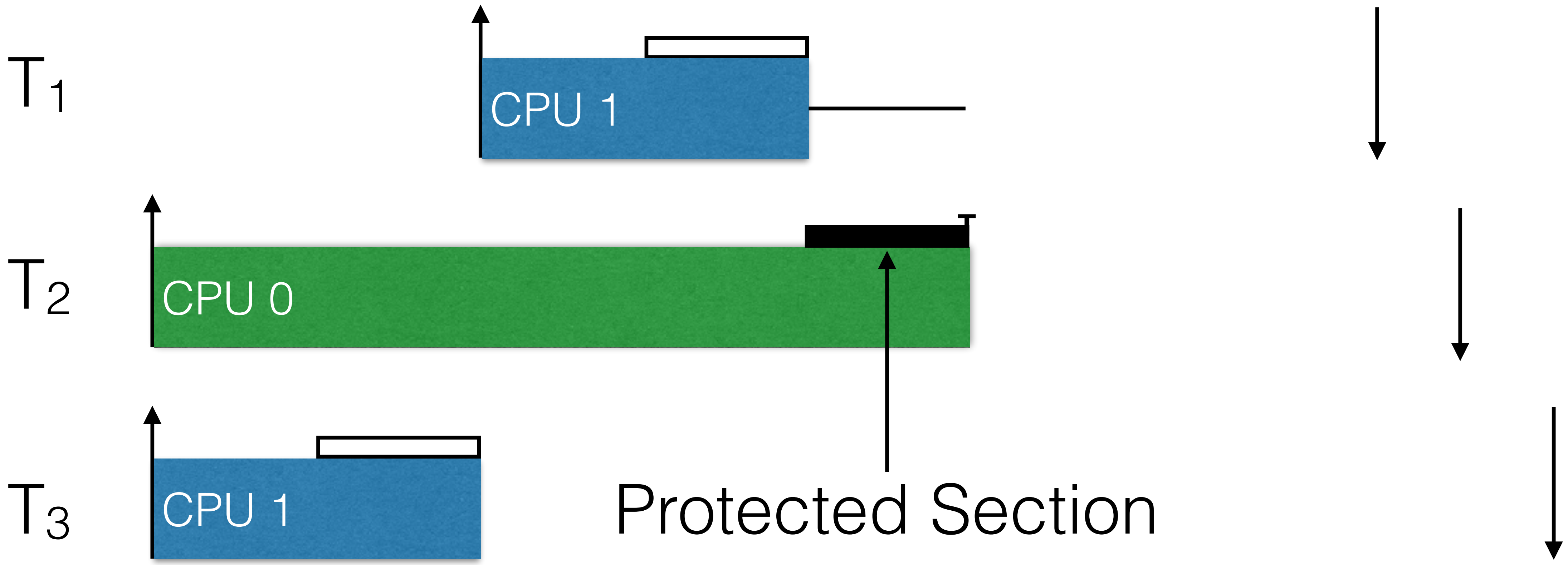
T1 preempts T3 while it is in an unprotected section

Half-Protected Example



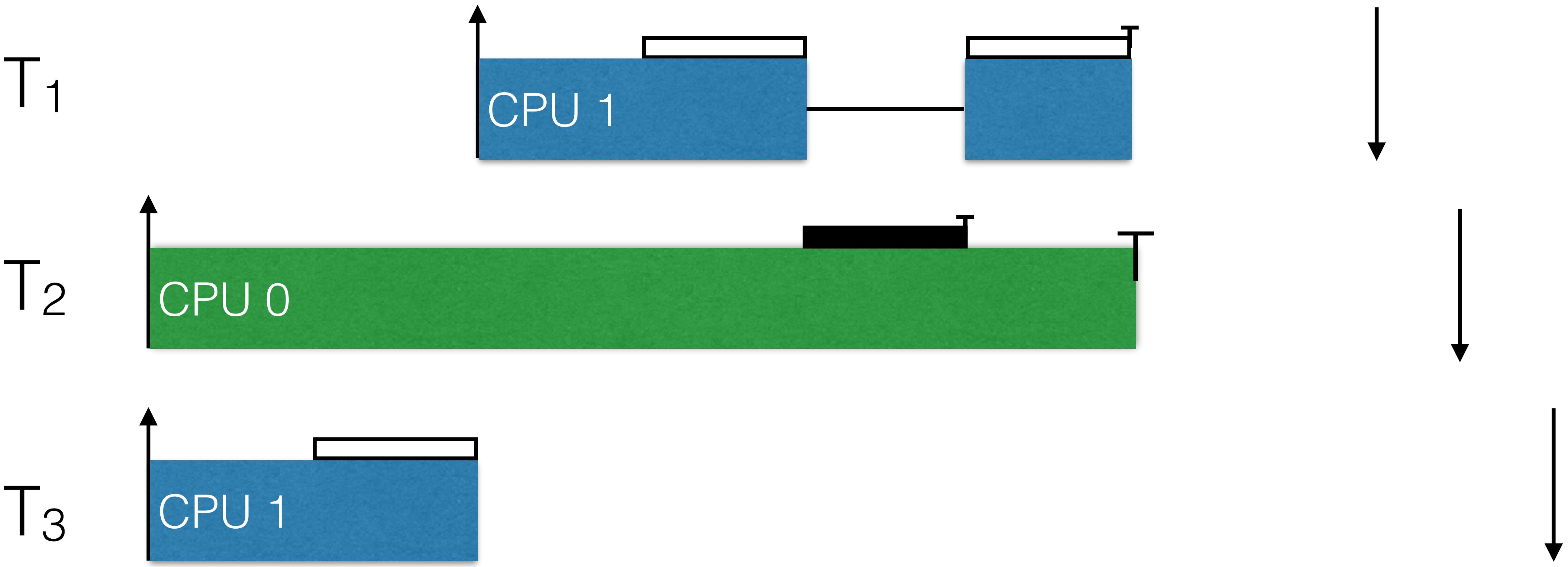
T1 begins unprotected section,
even though T3 has not completed
its unprotected section.

Half-Protected Example



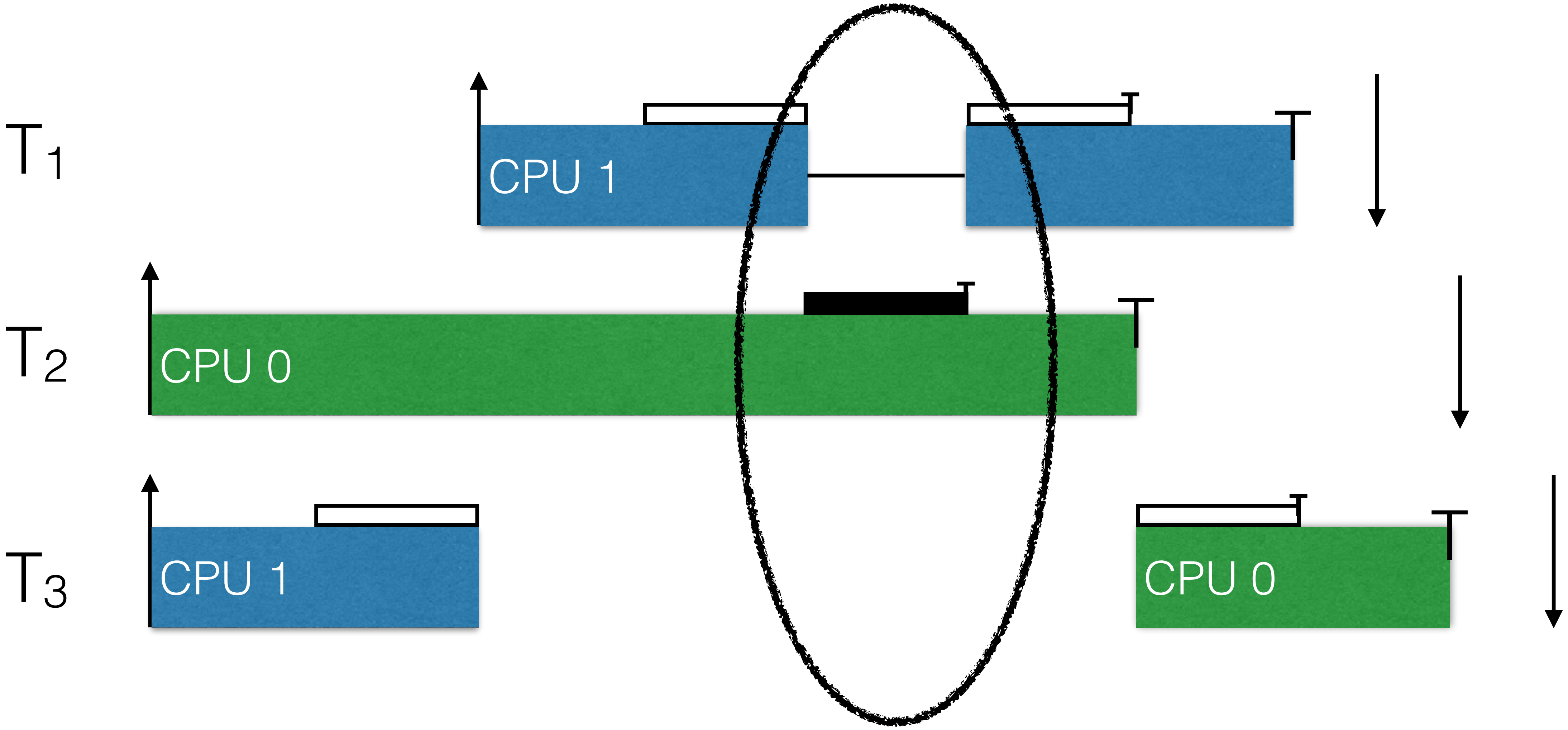
Protected section “preempts” the unprotected sections, inducing idleness on CPU 1.

Half-Protected Example



Unprotected section may resume after the protected section completes.

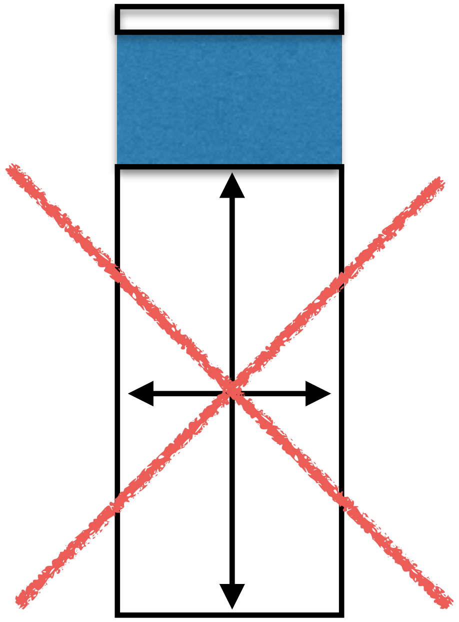
Half-Protected Example



With a reader/writer lock, such a "preemption" is not safe.

Unprotected Sections

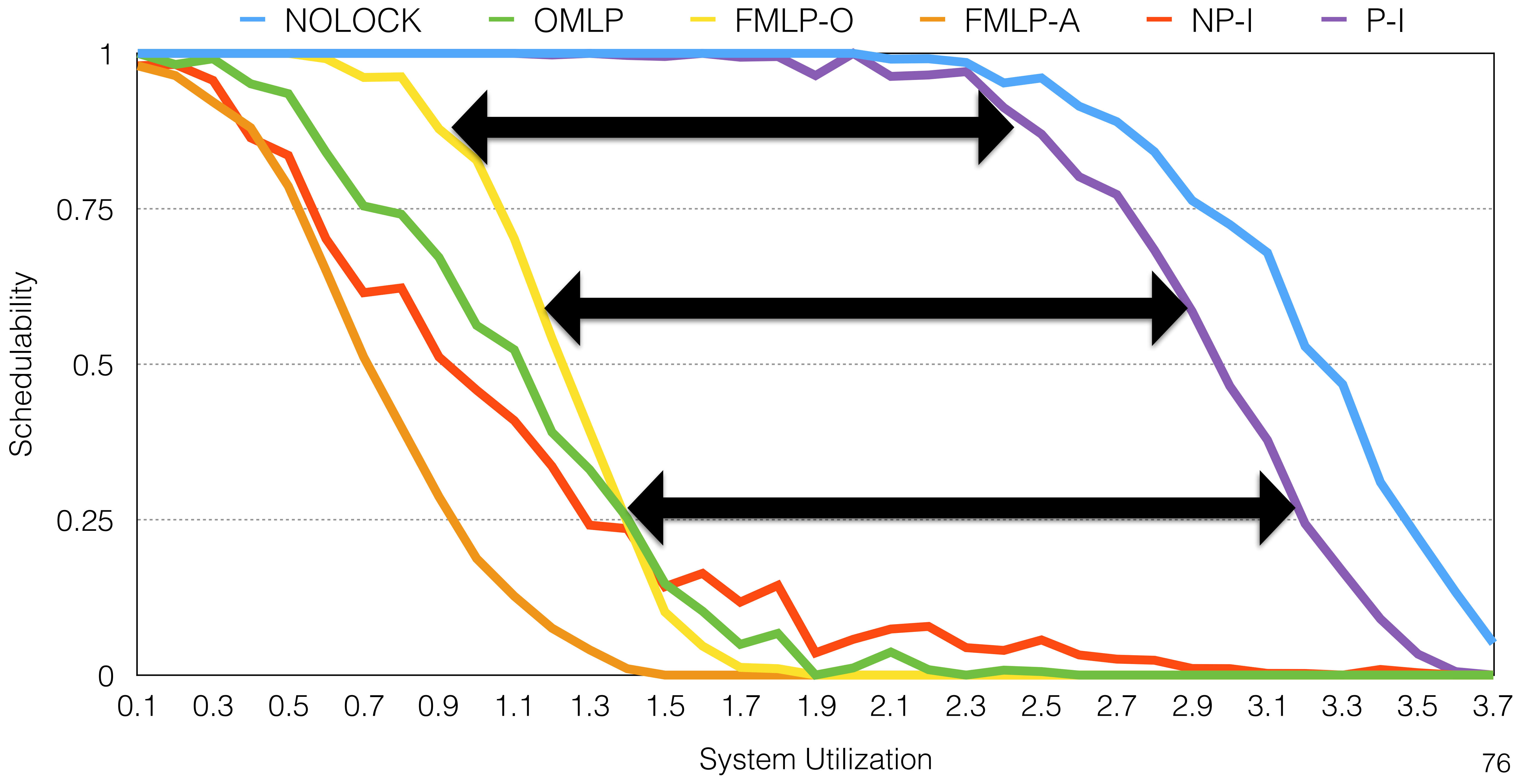
Observation: unprotected sections cannot cause blocking.



Result: Ignore them in idleness analysis.

Schedulability benefit made possible by preemptive mutual exclusion.

Per-task critical-section utilization = 0.1



Conclusions

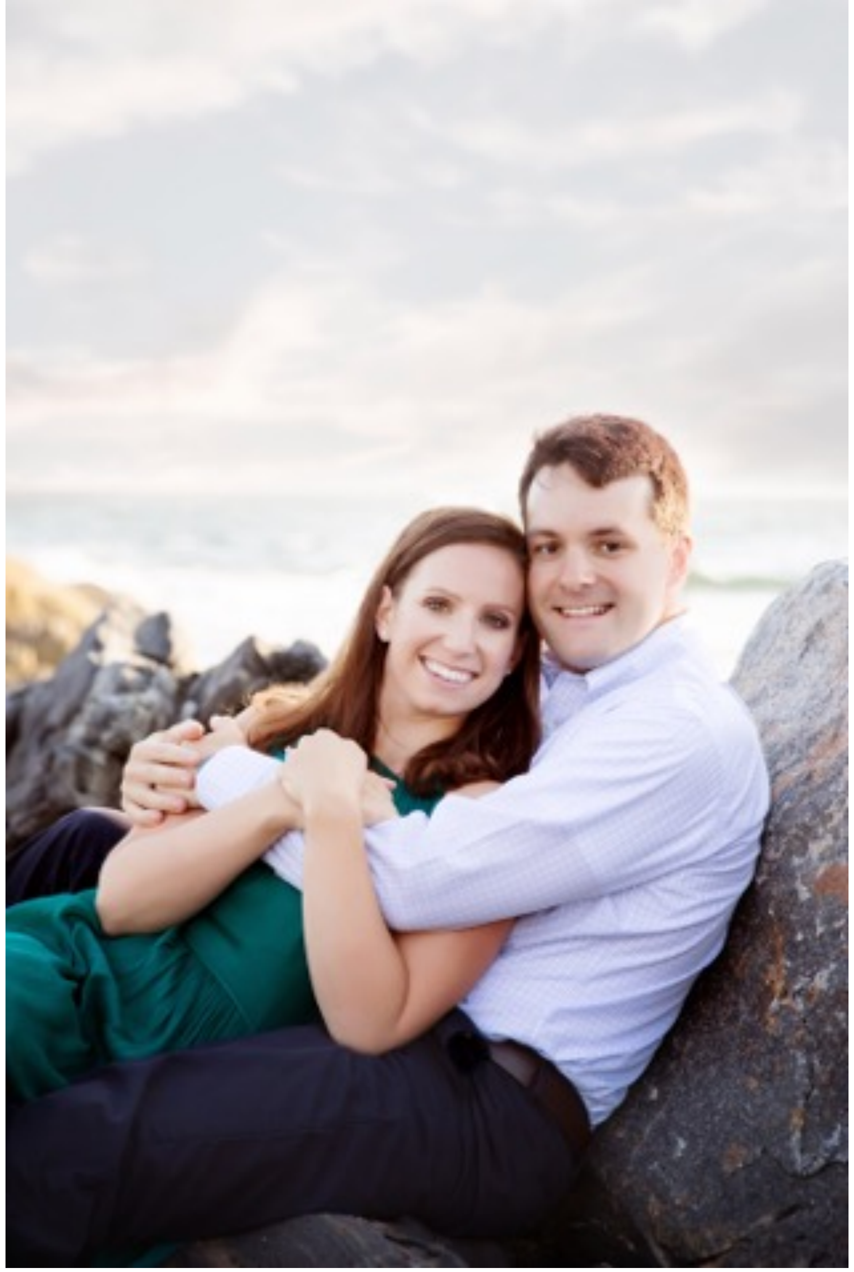
- **RNLP**: First fine-grained mutex, *k*-exclusion, and reader/writer multiprocessor real-time locking protocols.
- **Idleness analysis**: New analysis technique for accounting for blocking in schedulability analysis.
- **Preemptive and half-protected synchronization**: New models for synchronizing access to hardware resources that reduce utilization loss.

Acknowledgements

- My advisor and committee: Jim Anderson, Sanjoy Baruah, Björn Brandenburg, Alan Burns, and Don Smith.
- UNC Real-Time Systems Group.
- UNC CS department staff

Acknowledgements

Acknowledgements



- Namhoon Kim, **Bryan C. Ward**, Micaiah Chisholm, James H. Anderson, and F. Donelson Smith. "Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning", (in submission).
- Micaiah Chisholm, Namhoon Kim, **Bryan C. Ward**, Nathan Otterness, James H. Anderson, and F. Donelson Smith. "Reconciling the Tension Between Hardware Isolation and Data Sharing in Mixed-Criticality, Multicore Systems", RTSS ‘16 (to appear).
- Namhoon Kim, **Bryan C. Ward**, Micaiah Chisholm, Cheng-Yang Fu, James H. Anderson, and F. Donelson Smith. "Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning”, RTAS ‘16, April 2016. **Best Student Paper Award**.
- **Bryan C. Ward**. "Relaxing Resource-Sharing Constraints for Improved Hardware Management and Schedulability", RTSS ‘15, December 2015.
- Micaiah Chisholm, **Bryan C. Ward**, Namhoon Kim, and James H. Anderson. "Cache Sharing and Isolation Tradeoffs in Multicore Mixed-Criticality Systems", RTSS ‘15, December 2015.
- Catherine Jarrett, **Bryan C. Ward**, and James H. Anderson. "A Contention-Sensitive Multi-Resource Locking Protocol for Multiprocessor Real-Time Systems", RTNS ‘15, November 2015.
- **Bryan C. Ward** and James H. Anderson. "A Contention-Sensitive Multi-Resource Locking Protocol for Multiprocessor Real-Time Systems", RTSS ’14 WIP, December 2014.
- **Bryan C. Ward**, Abhilash Thekkilakattil, and James H. Anderson. "Optimizing Preemption-Overhead Accounting in Multiprocessor Real-Time Systems", RTNS ‘14, October 2014.
- **Bryan C. Ward** and James H. Anderson. "Multi-Resource Real-Time Reader/Writer Locks for Multiprocessors", IPDPS ‘14, May 2014.
- Jeremy P. Erickson, James H. Anderson, and **Bryan C. Ward**. "Fair Lateness Scheduling: Reducing Maximum Lateness in G-EDF-like Scheduling", Real-Time Systems, January 2014.
- Glenn A. Elliott, **Bryan C. Ward** and James H. Anderson. "GPUSync: A Framework for Real-Time GPU Management", **RTSS ‘13**, December 2013.
- **Bryan C. Ward** and James H. Anderson. "Fine-Grained Multiprocessor Real-Time Locking with Improved Blocking", RTNS ‘13, October 2013.
- **Bryan C. Ward**, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. "Making Shared Caches More Predictable on Multicore Platforms", ECRTS ‘13, July 2013. **Outstanding Paper Award**.
- **Bryan C. Ward**, Jeremy P. Erickson and James H. Anderson. "A Linear Model for Setting Priority Points in Soft Real-Time Systems", Proceedings of Real-Time Systems: The Past, the Present, and the Future -- A Conference Organized in Celebration of Alan Burns's Sixtieth Birthday, March 2013.
- **Bryan C. Ward**, Glenn A. Elliott and James H. Anderson. "Replica-Request Priority Donation: A Real-Time Progress Mechanism for Global Locking Protocols", RTCSA ‘12, August 2012.
- **Bryan C. Ward** and James H. Anderson. "Supporting Nested Locking in Multiprocessor Real-Time Systems", ECRTS ‘12, July 2012.