LITMUS^{RT}: An Overview

(based on a talk given at the Real-Time Linux Workshop 2007)

Björn B. Brandenburg
bbb@cs.unc.edu

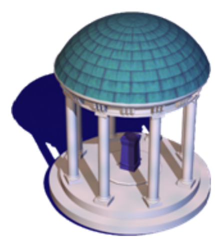
Aaron D. Block
block@cs.unc.edu

John M. Calandrino
jmc@cs.unc.edu

UmaMaheswari Devi
uma@cs.unc.edu

Hennadiy Leontyev
leontyev@cs.unc.edu

James H. Anderson
anderson@cs.unc.edu



The University *of* North Carolina *at* Chapel Hill

Work supported by a grant from Intel Corp., by NSF grants CCR 0408996, CCR 0541056, and CCR 0615197, and by ARO grant W911NF-06-1-0425.

LITMUS? What?

LITMUS^{RT}

=

LInux **T**estbed for **MU**ltiprocessor **S**cheduling

in **R**eal-**T**ime Systems

A new Linux real-time extension developed at UNC.

What Kind of Real-Time?

real-time:

What Kind of Real-Time?

real-time:

low latency

predictability

What Kind of Real-Time?

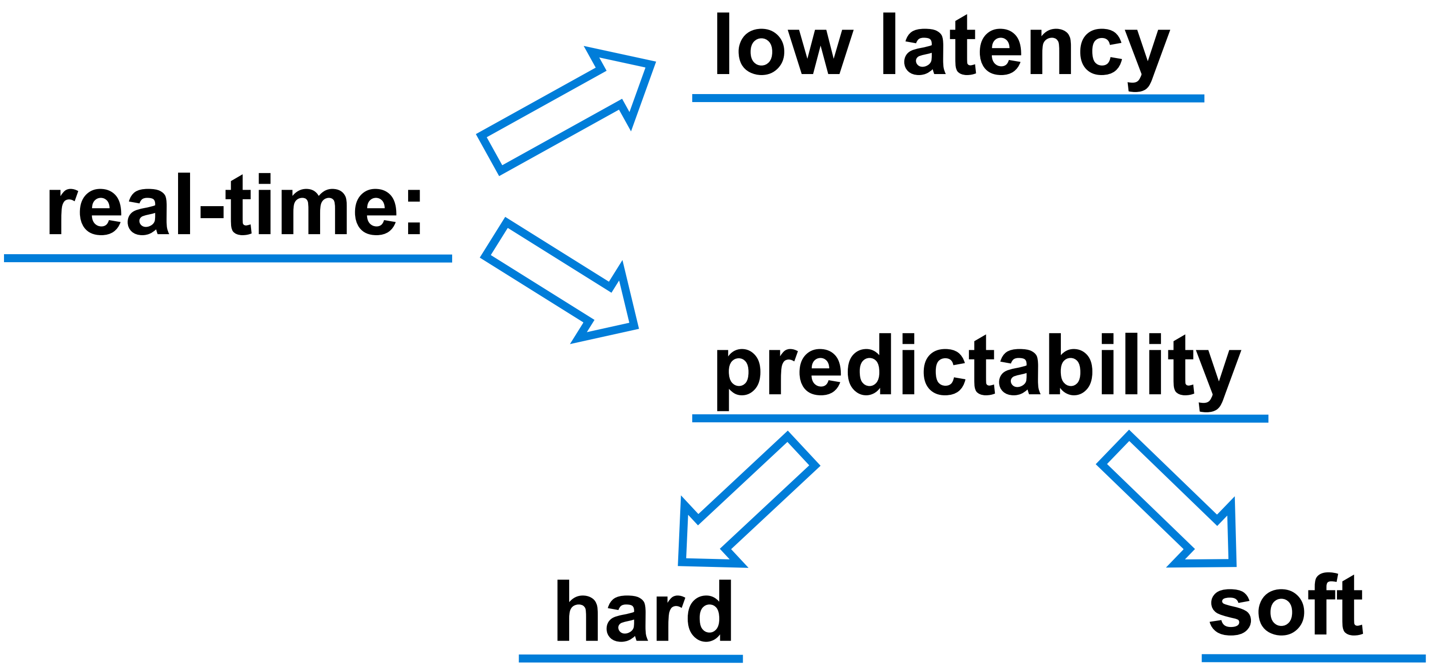
real-time:

low latency

predictability

PREEMPT_RT
RTLinux
RTAI
L4Linux/DROPS
…
Plenty of other RTOSs

What Kind of Real-Time?



What Kind of Real-Time?

real-time:

low latency

predictability

hard

soft

optimal
=
NO deadlines missed

(if system is at most fully utilized)

What Kind of Real-Time?

real-time: → **low latency**

real-time: → **predictability**

predictability → **hard**

predictability → **soft**

hard

optimal
=
NO deadlines missed

soft

optimal
=
deadlines missed
by at most
bounded amount

(if system is at most fully utilized)

The "Gap"

Optimality of real-time scheduling algorithms:

	uniproc.	partitioned	global
static priority			
by deadline			
PFAIR			

The "Gap"

Optimality of r **Algorithm Family** orithms:

	uniproc.	partitioned	global
static priority			
by deadline			
PFAIR			

The "Gap"

Optimality of real-time scheduling algorithms:

	uniproc.	partitioned	global
static priority			
by deadline			
PFAIR			

(Multi)processor Setting

The "Gap"

Optimality of real-time scheduling algorithms:

	uniproc.	partitioned	global
static priority	Hard: NO Soft: YES	Hard: NO Soft: NO	Hard: NO Soft: NO
by deadline	Hard: YES Soft: YES	Hard: NO Soft: NO	Hard: NO Soft: YES
PFAIR	Hard: (YES) Soft: (YES)	Hard: (NO) Soft: (NO)	Hard: YES Soft: YES

The "Gap"

Optimality of real-time scheduling algorithms:

	uniproc.	partitioned	global
static priority	Hard: NO Soft: YES	Hard: NO Soft: NO	Hard: NO Soft: NO
by deadline	Hard: YES Soft: YES	Hard: NO Soft: NO	Hard: NO Soft: YES
PFAIR	Hard: (YES) Soft: (YES)	Hard: (NO) Soft: (NO)	Hard: YES Soft: YES

Theory

The "Gap"

Optimality of real-time scheduling algorithms:

	Implemented Systems		
			global
static priority	Hard: NO Soft: YES	Hard: NO Soft: NO	Hard: NO Soft: NO
by deadline	Hard: YES Soft: YES	Hard: NO Soft: NO	Hard: NO Soft: YES
PFAIR	Hard: (YES) Soft: (YES)	Hard: (NO) Soft: (NO)	Hard: YES Soft: YES
			Theory

The Gap

Real-Time on Multiprocessors?

Real-Time on Multiprocessors?

Multicore is here to stay.

Real-Time on Multiprocessors?

Multicore is here to stay.

- COTS will be multiprocessors in many cases.
- Real-Time Linux will be used on multicore platforms.

Real–Time on Multiprocessors?

Multicore is here to stay.

- COTS will be multiprocessors in many cases.
- Real-Time Linux will be used on multicore platforms.

Some real-time applications require a lot of computational power.

Real–Time on Multiprocessors?

Multicore is here to stay.

- COTS will be multiprocessors in many cases.
- Real-Time Linux will be used on multicore platforms.

Some real-time applications require a lot of computational power.

- HDTV-quality multimedia.
- Real-time business transactions.
- More to come as our capabilities increase.

Business Real-Time Computing

One example: AZUL Systems, Inc.

Consistent, Fast Response Times

When critical business applications pause, companies lose money. When it comes to fulfilling on-line purchases, executing stock trades at the real time price, acting on price fluctuations or approving loan applications, **completing only 85 percent of the requests in time is a failure**.

[From: http://www.azulsystems.com/products/compute_appliance.htm?p=p]

AZUL's appliances consist of up to **768 cores**!

One example: AZUL Systems, Inc.

Consistent, Fast Response Times

Predictability

Whe… critical business applications …mpanies lose money. When … fulfilling on-line purchases, executing stock trades at the real time price, acting on price fluctuations or approving loan applications, **completing only 85 percent of the requests in time is a failure**.

[From: http://www.azulsystems.com/products/compute_appliance.htm?p=p]

AZUL's appliances consist of up to **768 cores**!

Business Real-Time Computing

One example: AZUL Systems, Inc.

Consistent, Fast Response Times

Predictability

Low Latency

Whe critical bus ess applications pa companies e money. When fulfilling o purchases, executing stock tra price, acting on price fluctuations or approving loan applications, **completing only 85 percent of the requests in time is a failure**.

[From: http://www.azulsystems.com/products/compute_appliance.htm?p=p]

AZUL's appliances consist of up to **768 cores**!

Why is it a Testbed?

Why is it a Testbed?

What is the “best” multiprocessor real-time scheduling algorithm?

Why is it a Testbed?

What is the “best” multiprocessor real-time scheduling algorithm?

- Most proposed algorithms have **never been implemented** in a real system.
- Real-world performance in face of **overheads** is unclear.
- First implementation = **no proven way**

Goals of the LITMUS^{RT} Project

Goals of the LITMUS[RT] Project

Help to **bridge the gap** between theory and practice.

Goals of the LITMUS^{RT} Project

Help to **bridge the gap** between theory and practice.

Evaluate algorithm choices under real-world conditions.

Goals of the LITMUS^{RT} Project

Help to **bridge the gap** between theory and practice.

Evaluate algorithm choices under real-world conditions.

Prove that it's **feasible to implement** advanced scheduling algorithms.

Goals of the LITMUS^{RT} Project

Help to **bridge the gap** between theory and practice.

Evaluate algorithm choices under real-world conditions.

Prove that it's **feasible to implement** advanced scheduling algorithms.

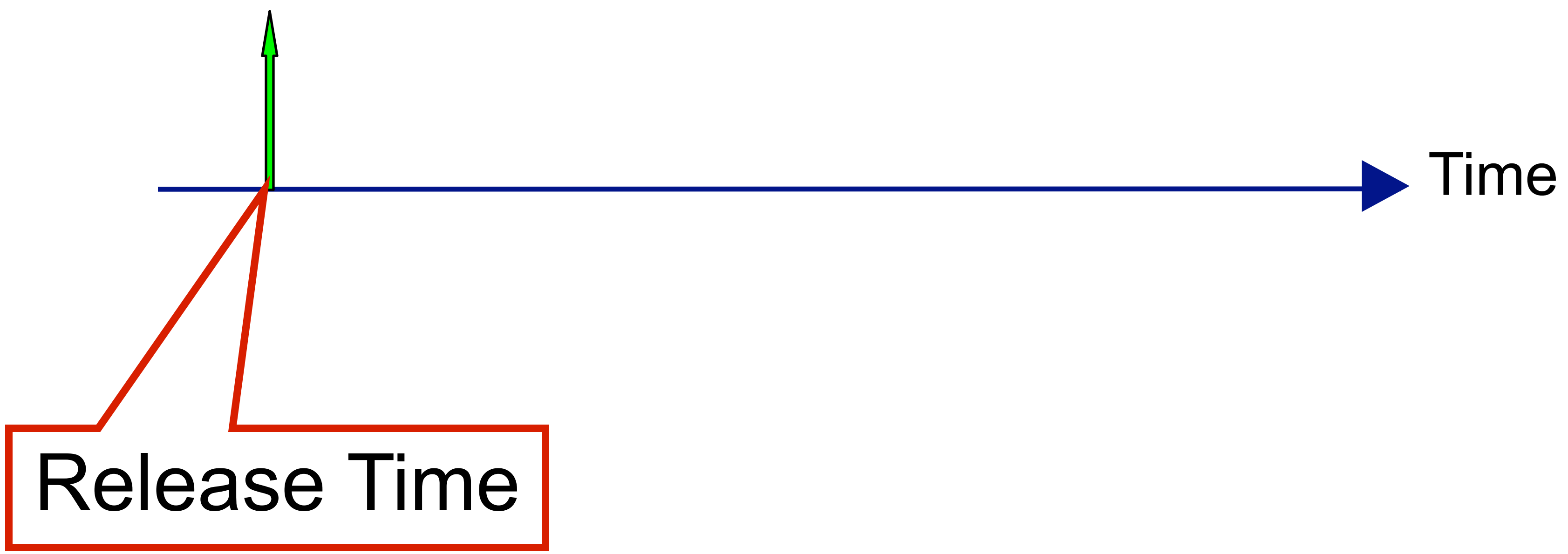
Provide inspiration to industry-grade real-time Linux variants.

Sporadic Jobs

A Job:

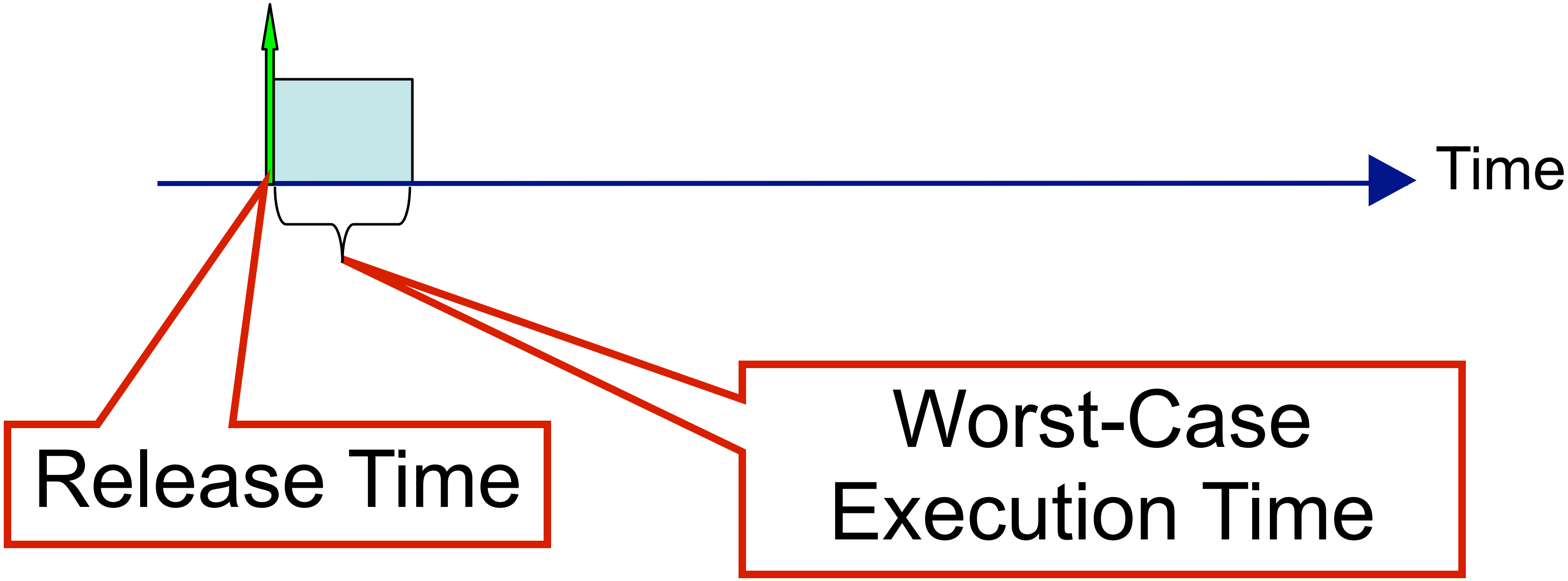
Sporadic Jobs

A Job:



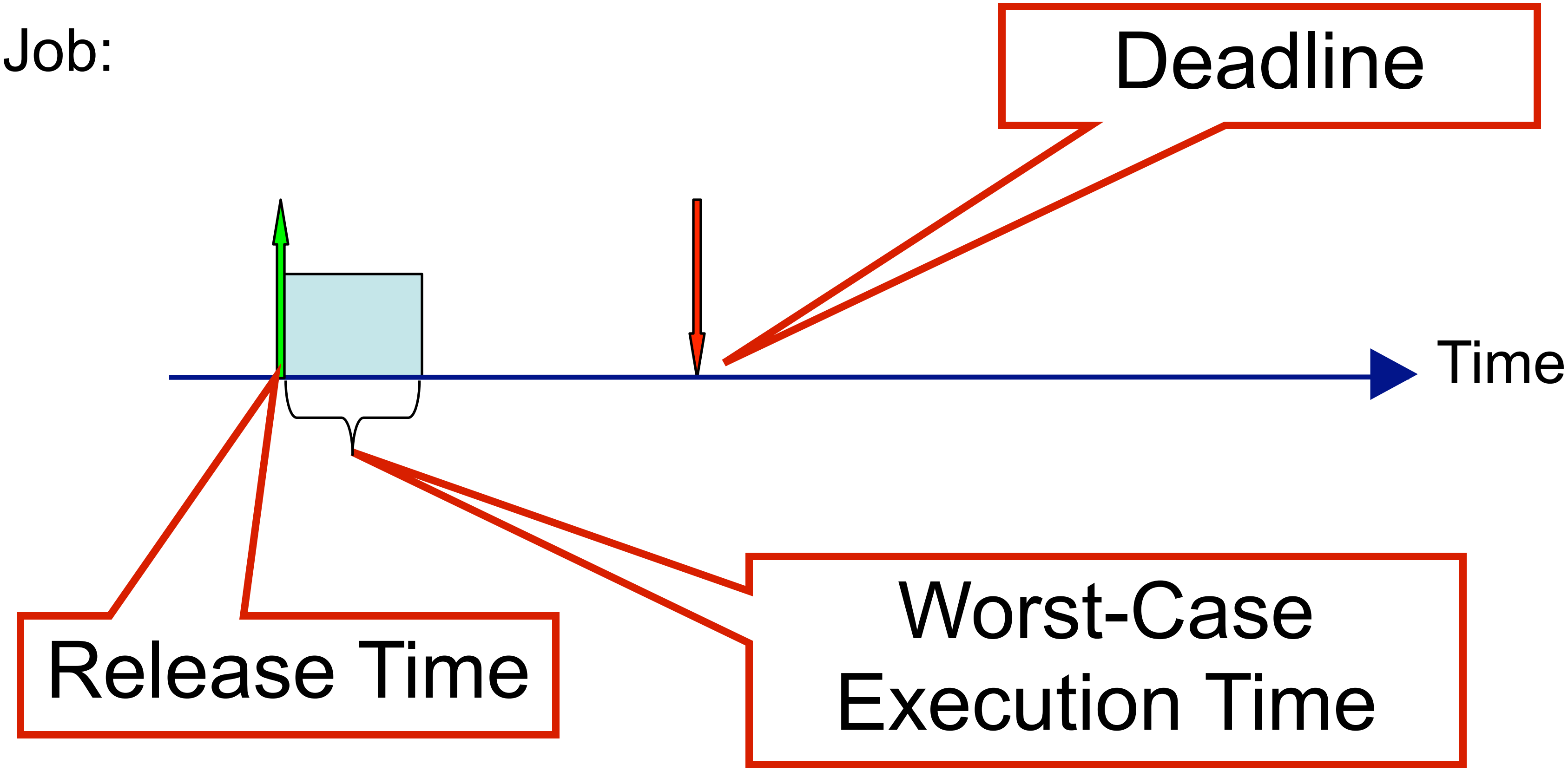
Sporadic Jobs

A Job:



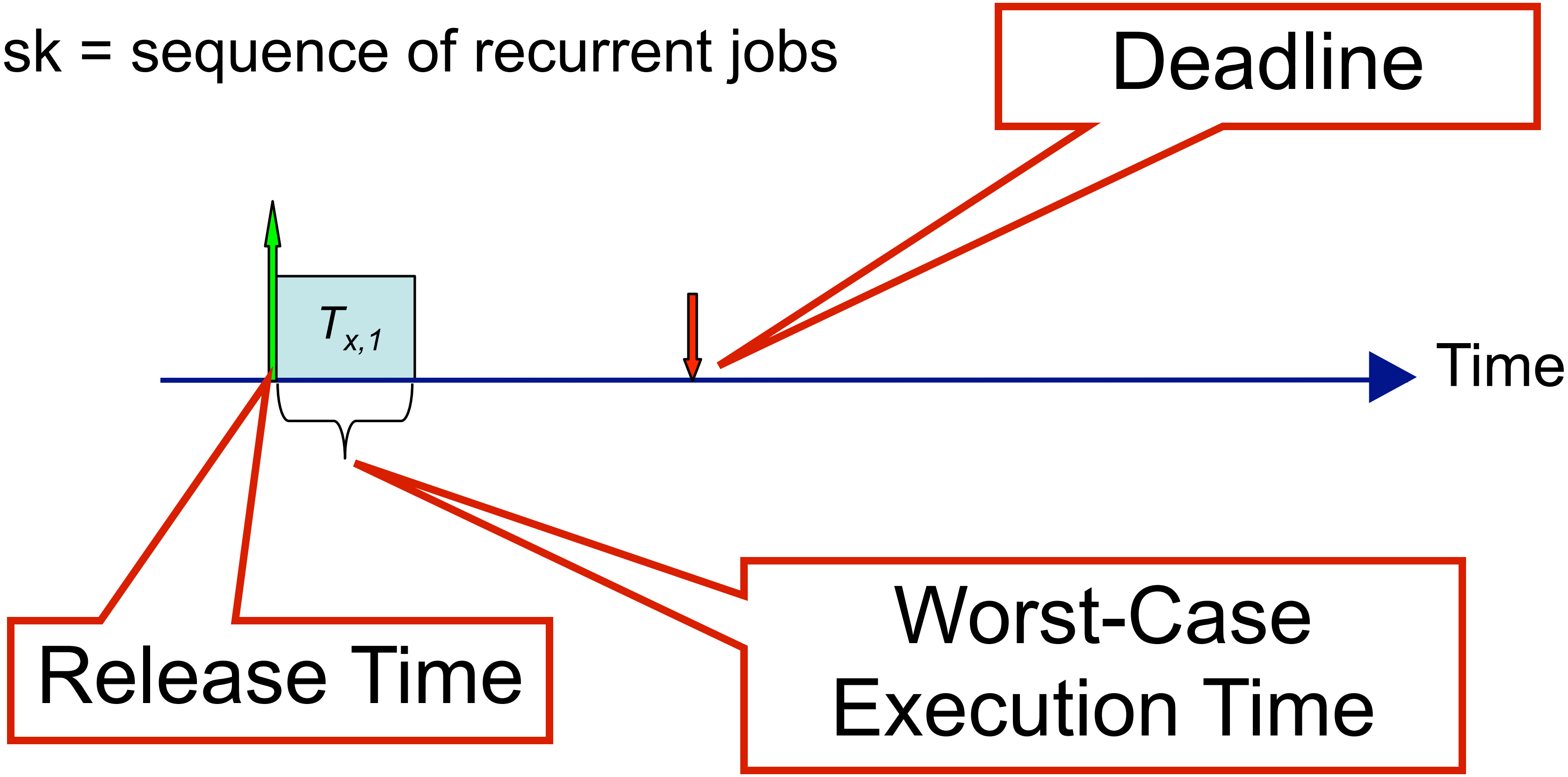
Sporadic Jobs

A Job:



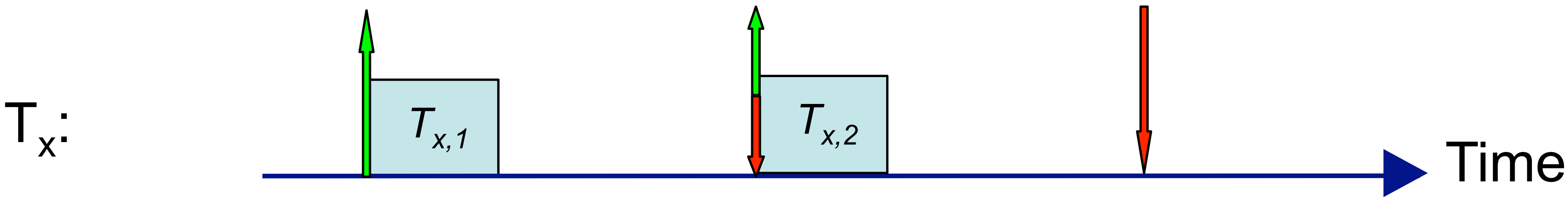
Sporadic Tasks

Task = sequence of recurrent jobs



Sporadic Tasks

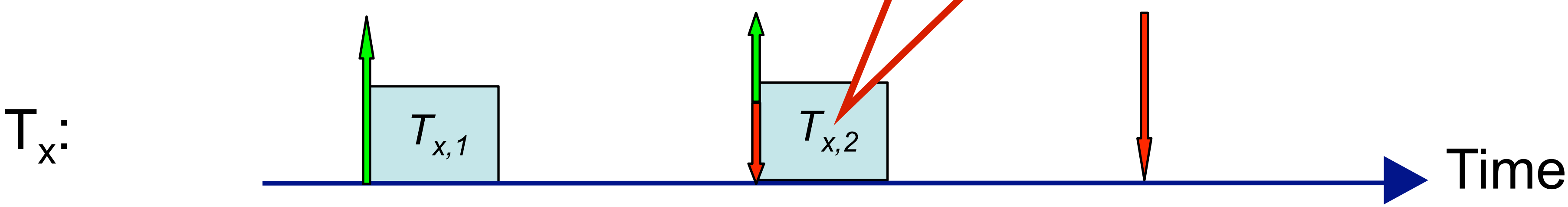
Task = sequence of recurrent jobs



Sporadic Tasks

Job: $T_{<\text{task no}>,<\text{job no}>}$

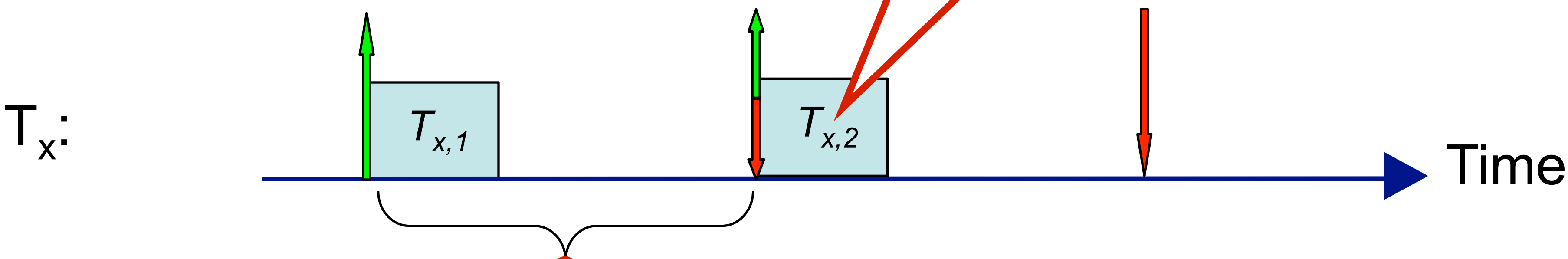
Task = sequence of recurrent jobs



Sporadic Tasks

Job: $T_{\text{<task no>,<job no>}}$

Task = sequence of recurrent jobs

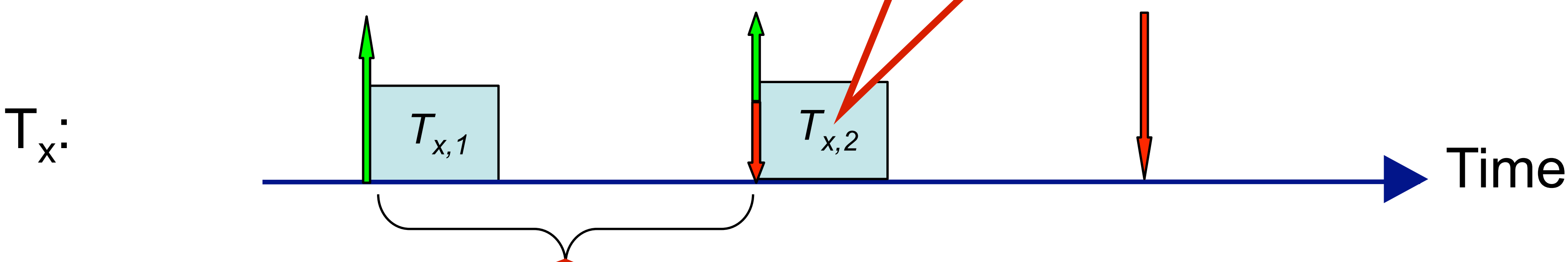


relative deadline =
min. inter-arrival separation / period

Sporadic Tasks

Job: $T_{<\text{task no}>,<\text{job no}>}$

Task = sequence of recurrent jobs

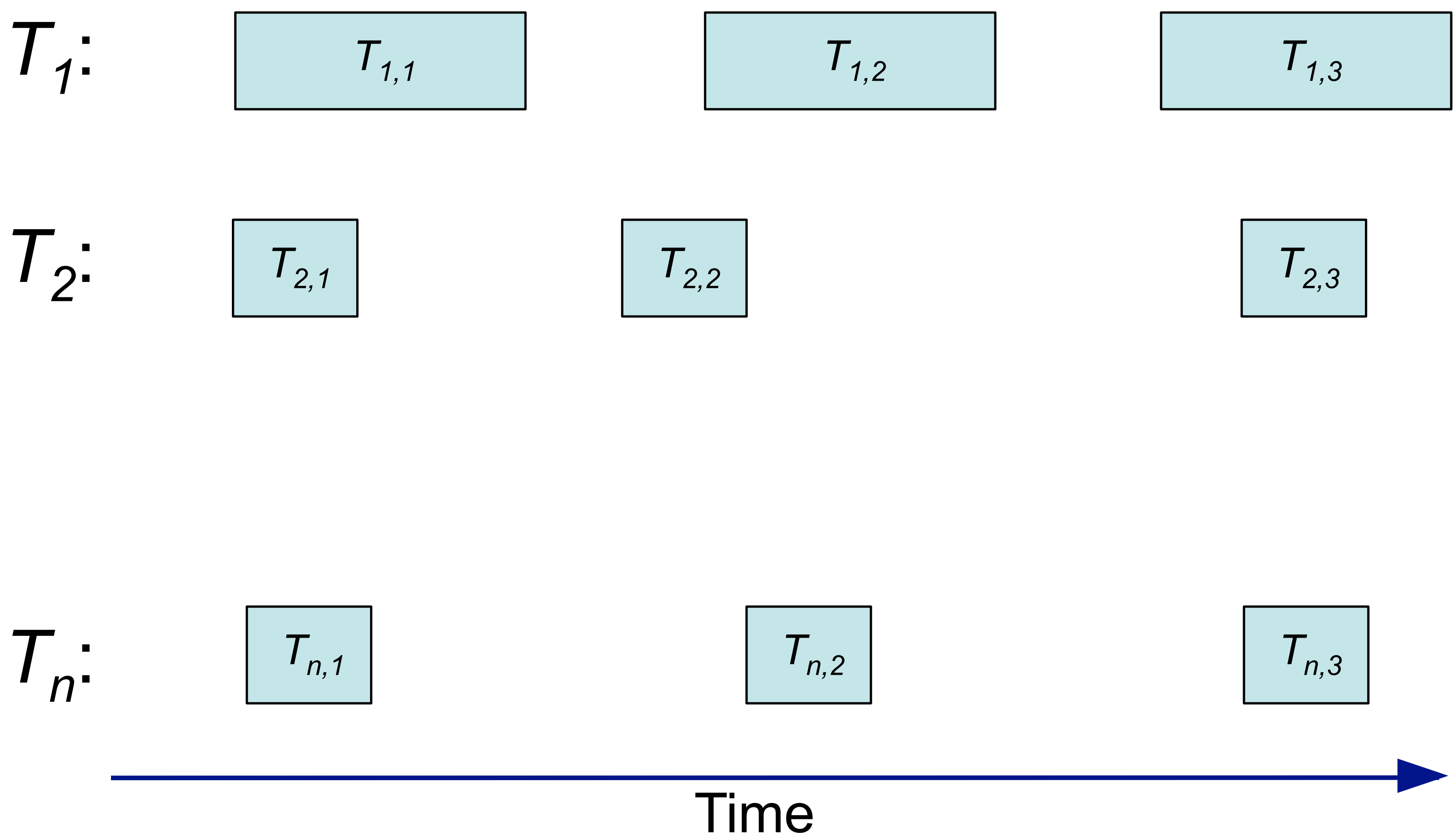


relative deadline =
min. inter-arrival separation / period

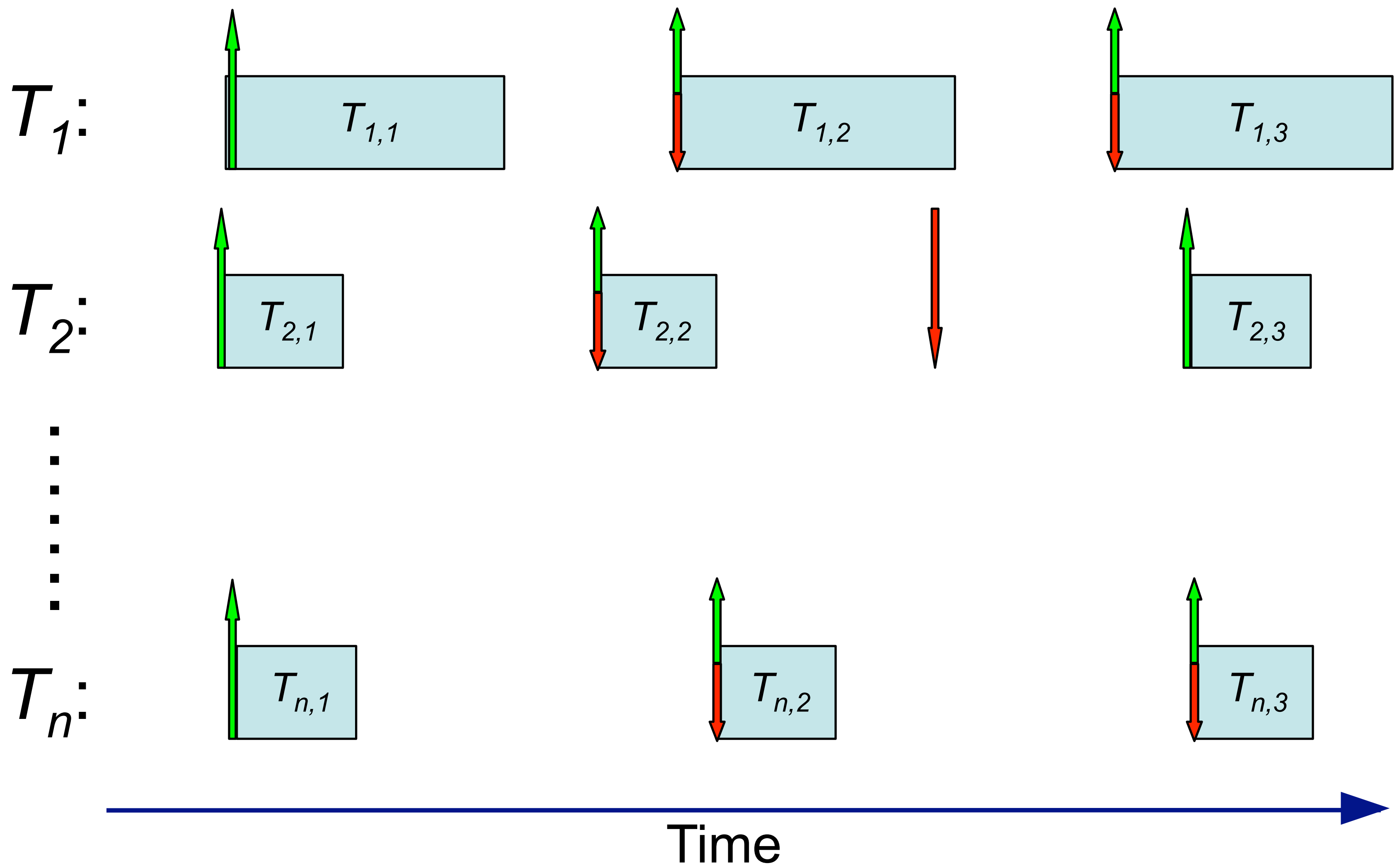
Task **T_x = (WCET, period)**

Utilization **u_x = WCET / period**

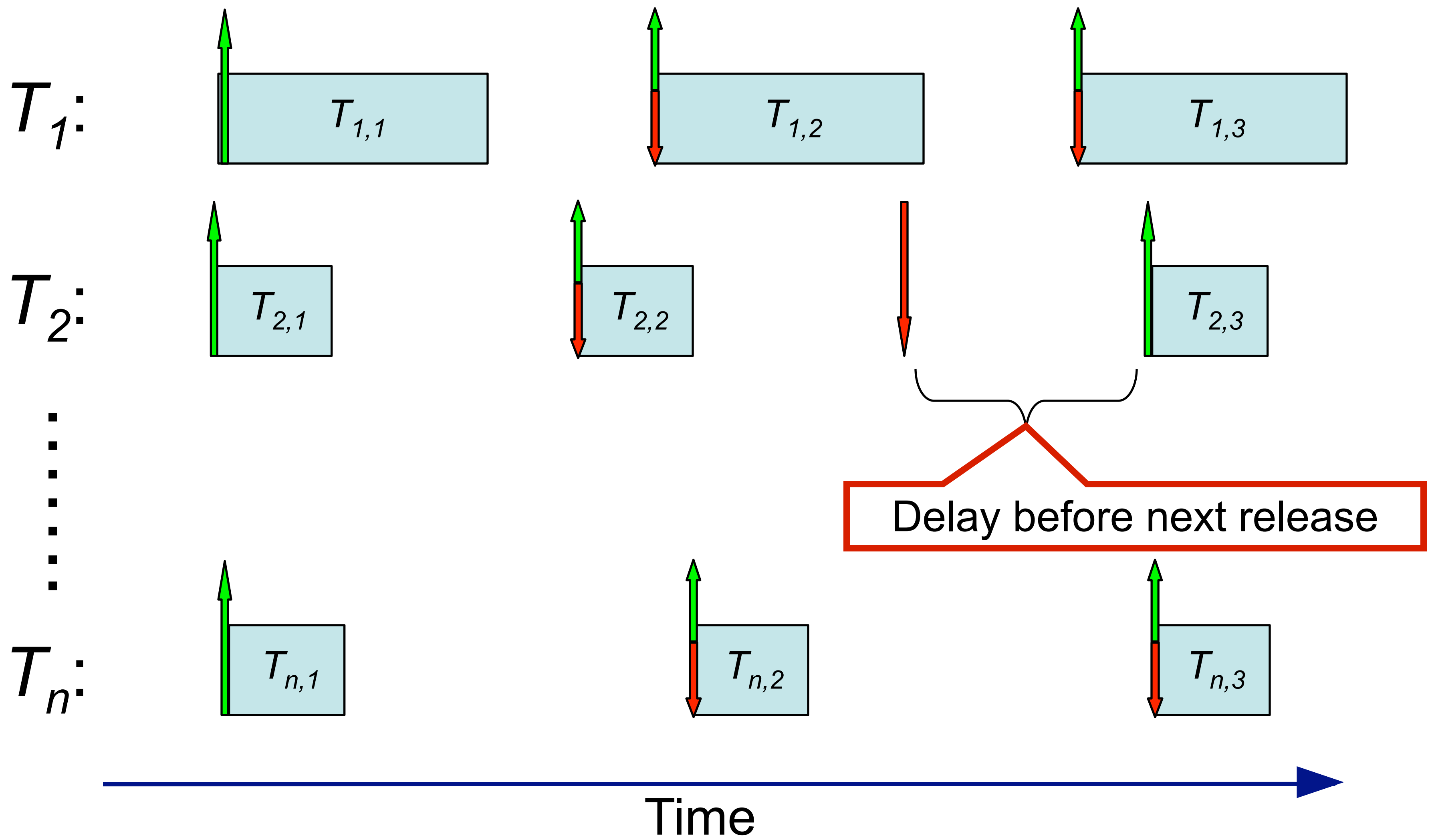
Sporadic Task System



Sporadic Task System



Sporadic Task System



Uniprocessor Real-Time Scheduling

Static (all jobs have same prio.)	**Dynamic** (jobs can differ in prio.)

Uniprocessor Real-Time Scheduling

Static (all jobs have same prio.)	**Dynamic** (jobs can differ in prio.)
Rate Monotonic (RM) **Prioritize by decreasing period**	

Uniprocessor Real-Time Scheduling

Static (all jobs have same prio.)	**Dynamic** (jobs can differ in prio.)
Rate Monotonic (RM) **Prioritize by decreasing period**	
Manual + Time Demand Analysis **Prioritize somehow and check**	

Uniprocessor Real-Time Scheduling

Static (all jobs have same prio.)	**Dynamic** (jobs can differ in prio.)
Rate Monotonic (RM) **Prioritize by decreasing period**	Earliest Deadline First (EDF) **Prioritize by decreasing deadlines**
Manual + Time Demand Analysis **Prioritize somehow and check**	

Uniprocessor Real-Time Scheduling

Static (all jobs have same prio.)	**Dynamic** (jobs can differ in prio.)
Rate Monotonic (RM) **Prioritize by decreasing period**	Earliest Deadline First (EDF) **Prioritize by decreasing deadlines**
Manual + Time Demand Analysis **Prioritize somehow and check**	Least Laxity First (LLF) **Prioritize by decreasing laxity**

Uniprocessor Real-Time Scheduling

Static (all jobs have same prio.)	**Dynamic** (jobs can differ in prio.)
Rate Monotonic (RM) **Prioritize by decreasing period**	Earliest Deadline First (EDF) **by decreasing deadlines**
Manual + Time Demand Analysis **Prioritize somehow and check**	Least Laxity First (LLF) **Prioritize by decreasing laxity**

Not further considered in this talk.

Uniprocessor Real-Time Scheduling

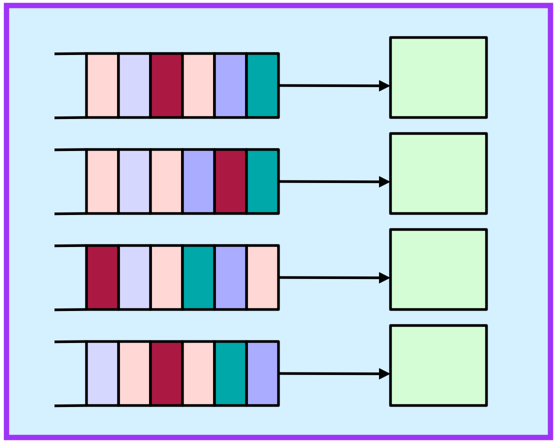
Only dynamic is (hard-)optimal!

(all jobs have same prio.)	(jobs can differ in prio.)
~~Rate Monotonic (RM)~~ **Prioritize by decreasing period**	Earliest Deadline First (EDF) **Prioritize by decreasing deadlines**
~~Manual + Time Demand Analysis~~ **Prioritize somehow and check**	Least Laxity First (LLF) **Prioritize by decreasing laxity**

Two Fundamental Approaches

Two Fundamental Approaches

Partitioning

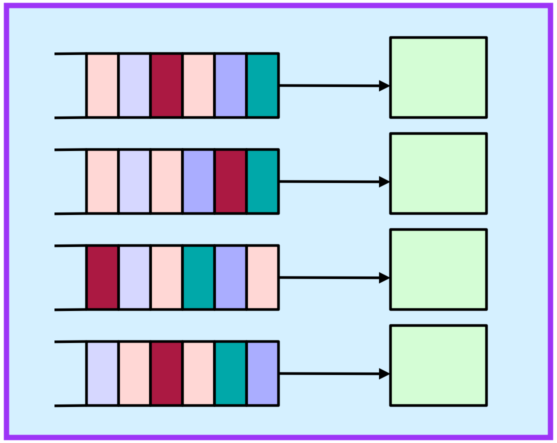


Steps:

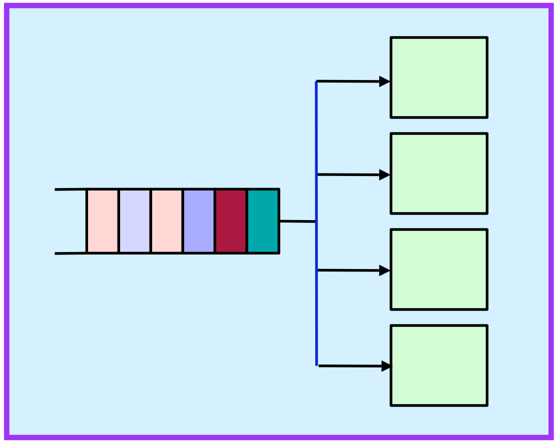
1. Assign tasks to processors (bin packing).
2. Schedule tasks on each processor using *uniprocessor* algorithms.

Two Fundamental Approaches

Partitioning



Global Scheduling



Steps:

1. Assign tasks to processors (bin packing).
2. Schedule tasks on each processor using *uniprocessor* algorithms.

Important Differences:

- One task queue.
- Tasks may *migrate* among the processors.

Algorithm Families

Partitioned	Global

Algorithm Families

Partitioned	Global
Static Priority (*e.g.* RMS, POSIX)	

Algorithm Families

Partitioned	Global
Static Priority (*e.g.* RMS, POSIX)	
Earliest Deadline First (EDF)	

Algorithm Families

Partitioned	Global
Static Priority (*e.g.* RMS, POSIX)	
Earliest Deadline First (EDF)	Earliest Deadline First (EDF)

Algorithm Families

Partitioned	Global
Static Priority (*e.g.* RMS, POSIX)	Fair (*e.g.* Prop. Share, PFAIR)
Earliest Deadline First (EDF)	Earliest Deadline First (EDF)

Algorithm Families

Only PFAIR is (hard-)optimal!

Partitioned	Global
Static Priority (*e.g.* RMS, POSIX)	Fair (*e.g.* Prop. Share, PFAIR)
Earliest Deadline First (EDF)	Earliest Deadline First (EDF)

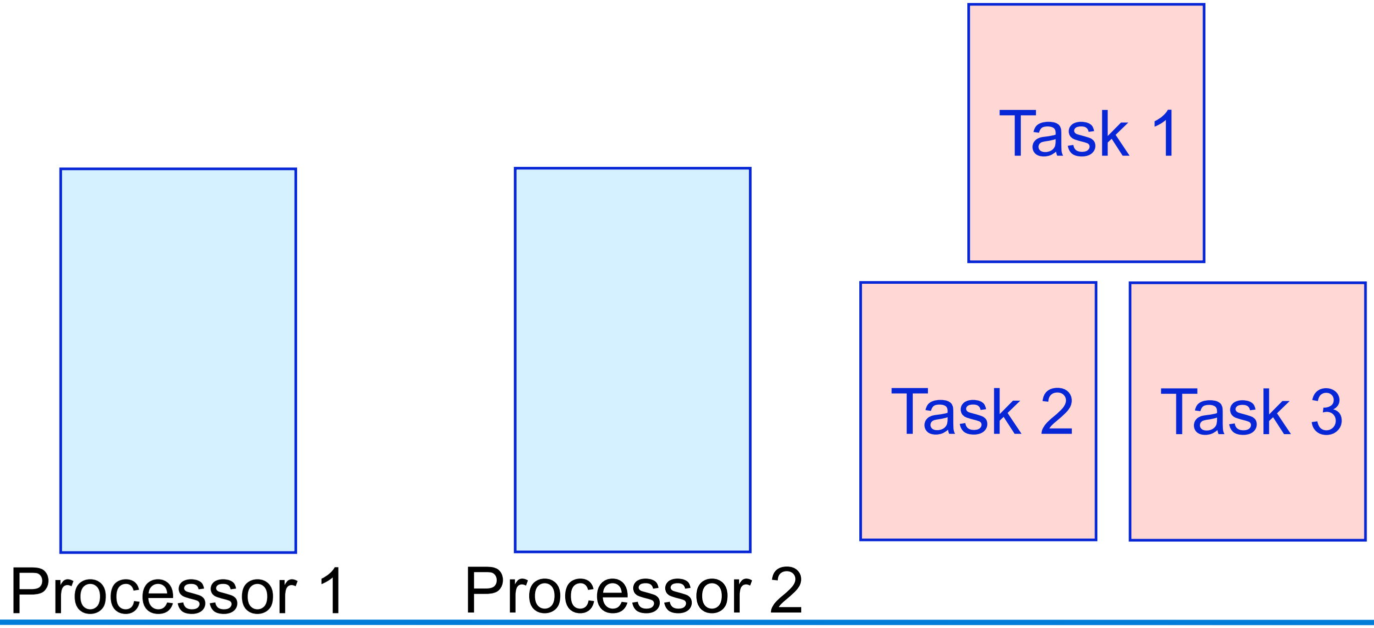
Partitioning is not Optimal

Partitioning suffers from bin-packing limitations.

Partitioning is not Optimal

Partitioning suffers from bin-packing limitations.

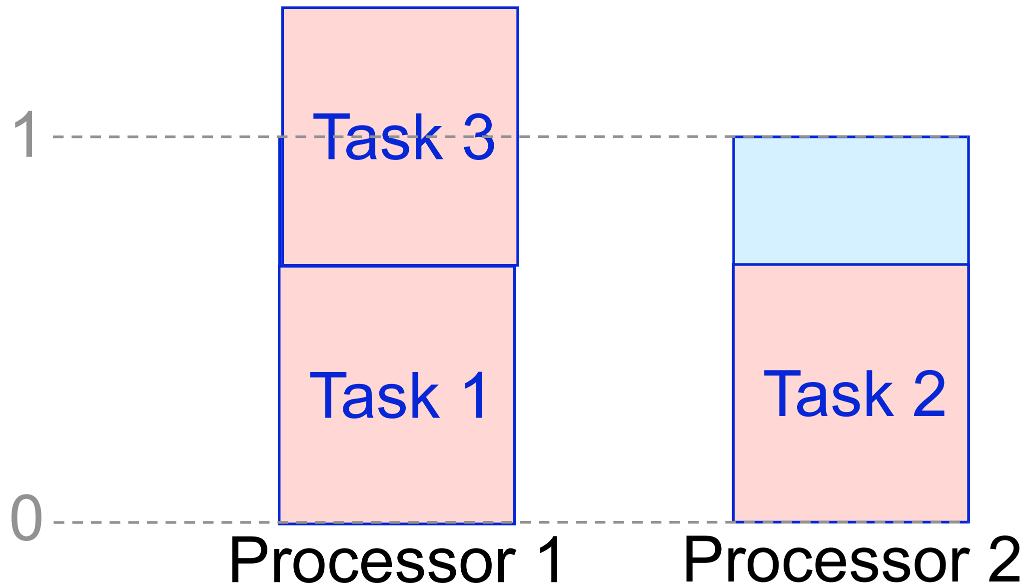
Example: Partitioning three tasks with parameters (2,3) on two processors will *overload* one processor.



Partitioning is not Optimal

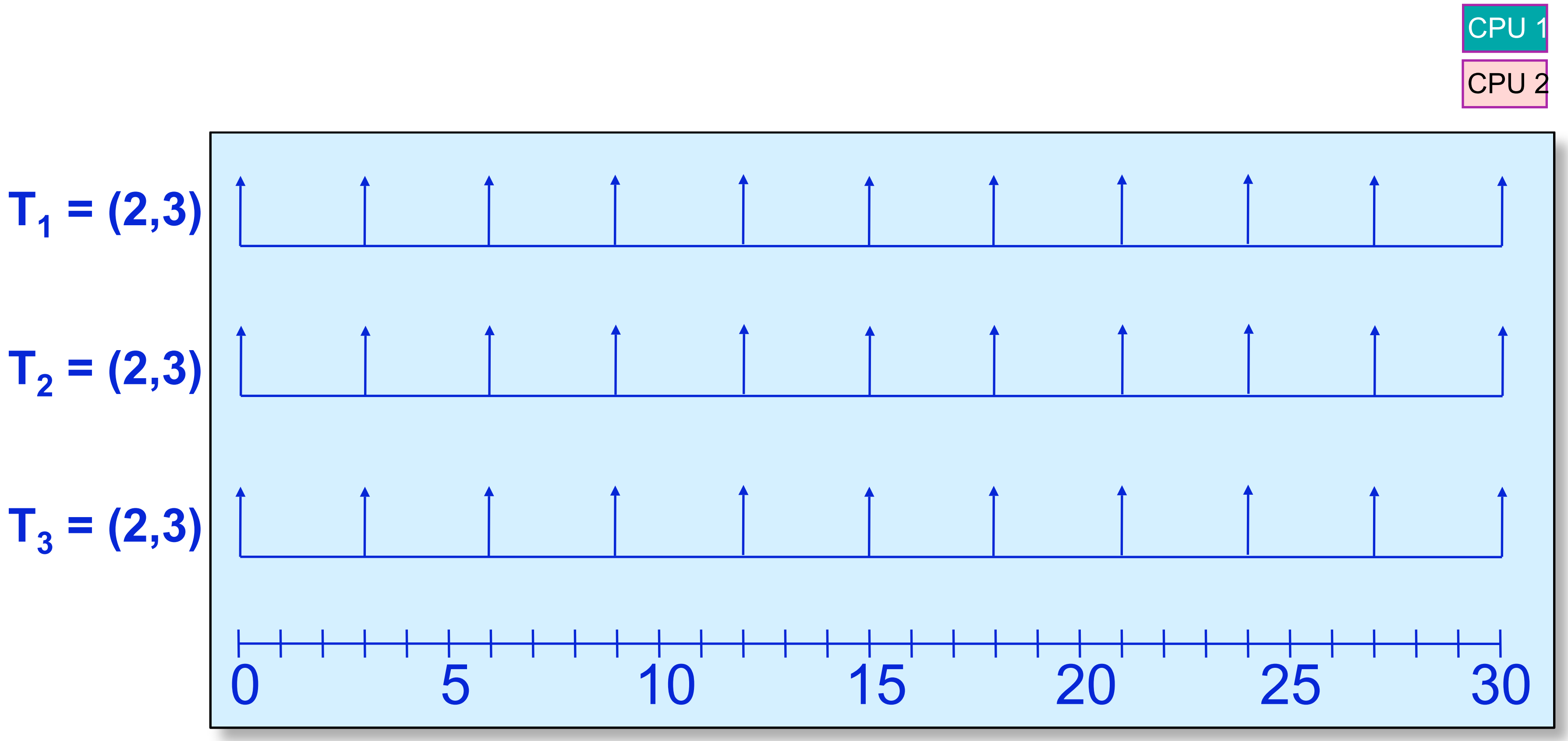
Partitioning suffers from bin-packing limitations.

Example: Partitioning three tasks with parameters (2,3) on two processors will *overload* one processor.



Global EDF

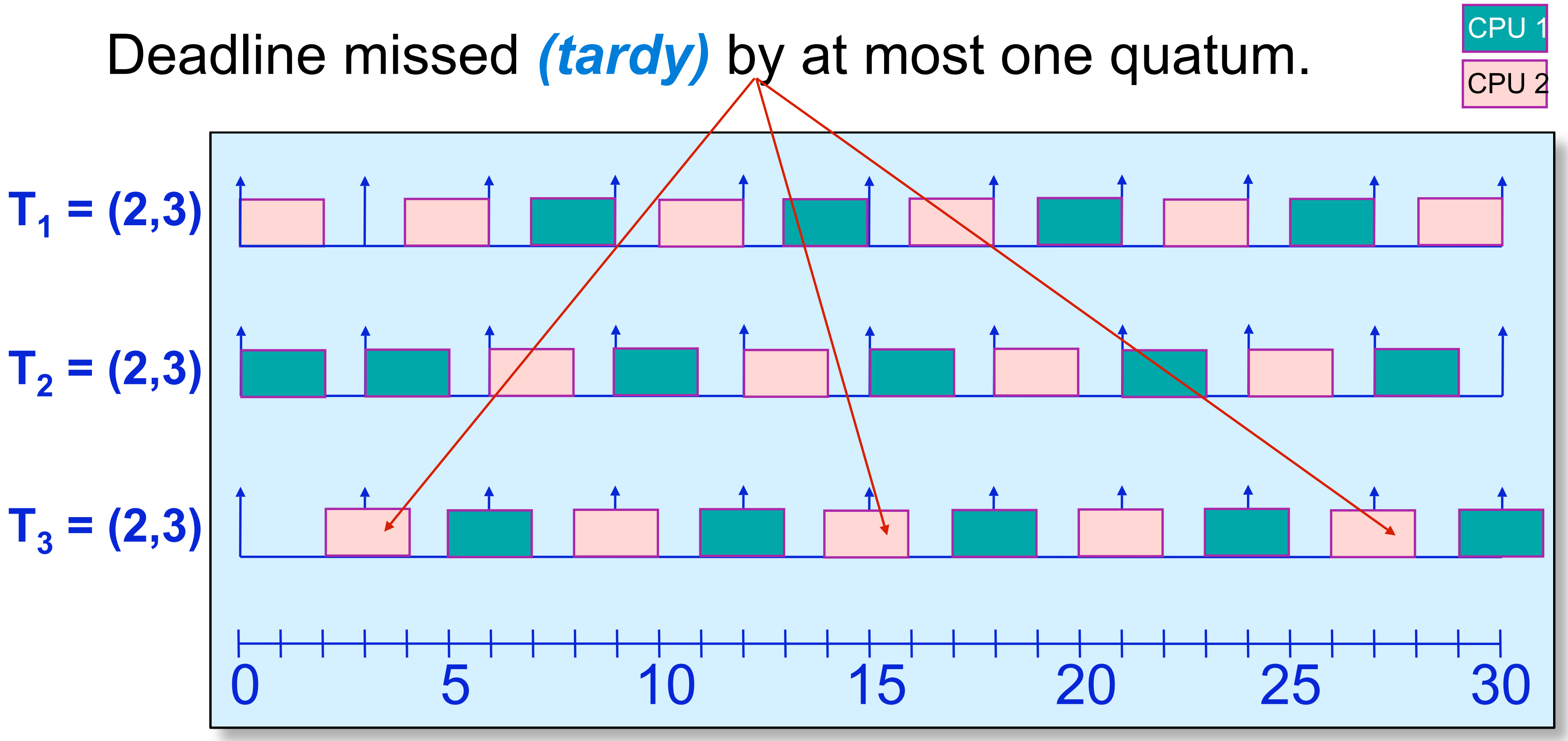
Previous example scheduled under **global EDF**…



Global EDF

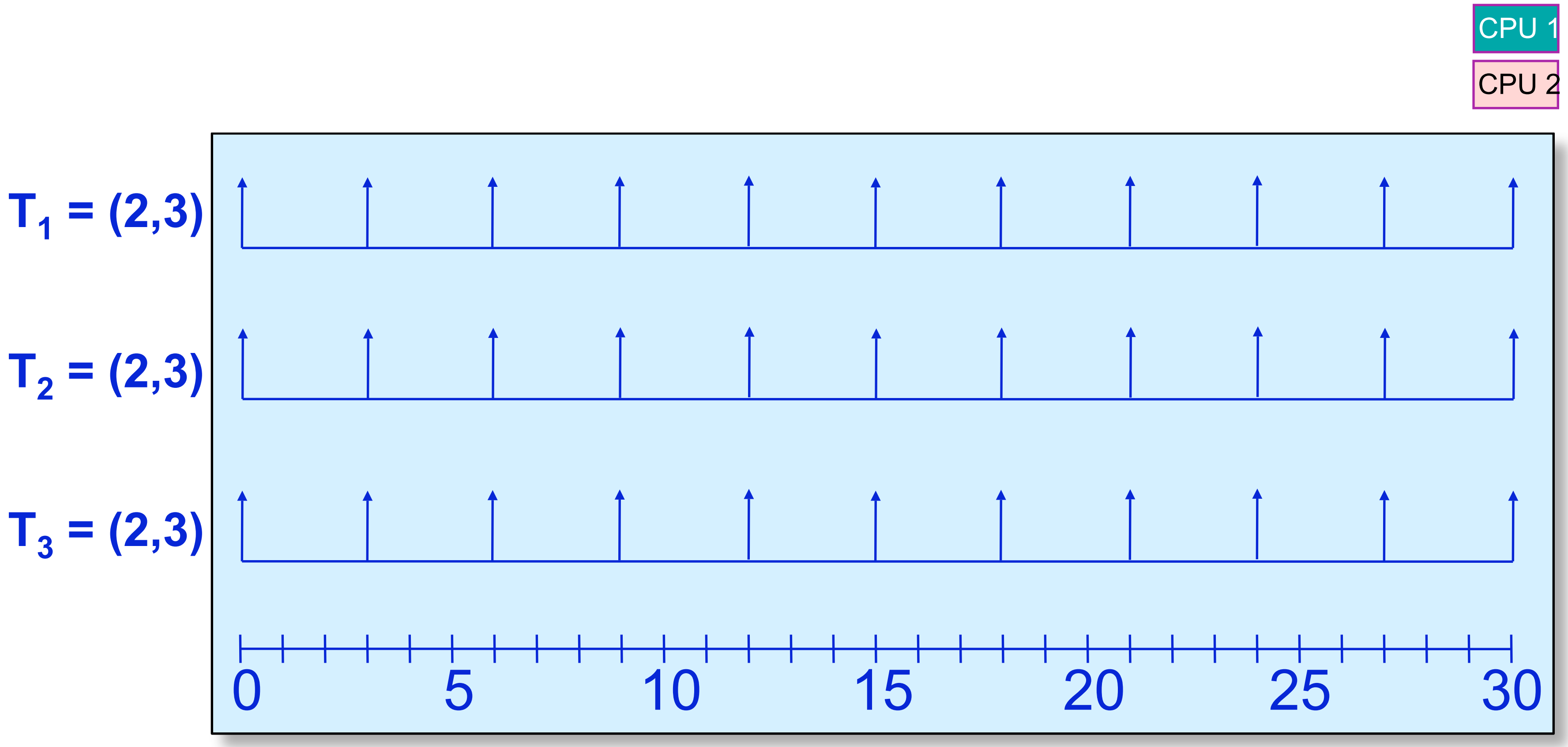
Previous example scheduled under **global EDF**...

Deadline missed ***(tardy)*** by at most one quatum.



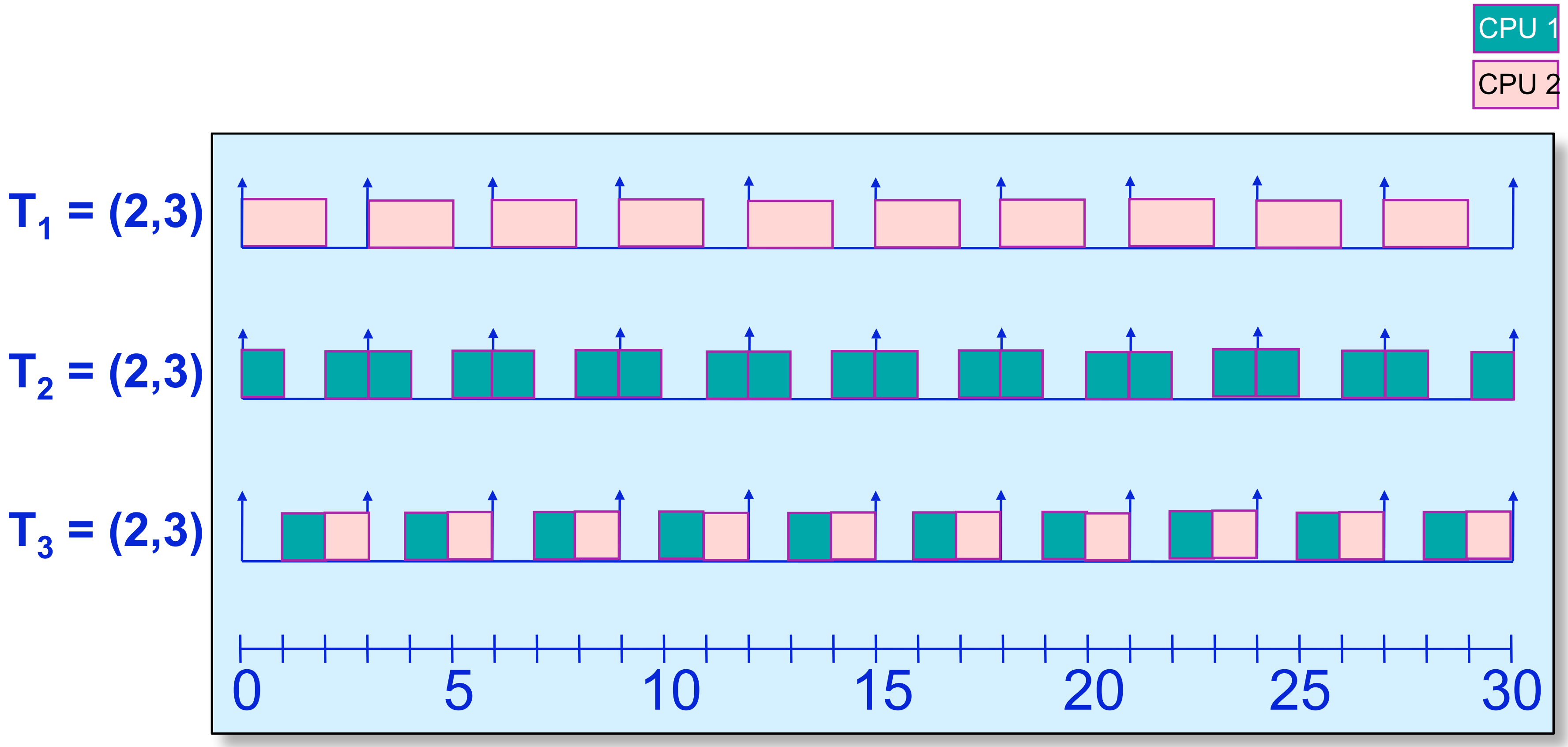
PFAIR (PD^2)

Previous example scheduled under PFAIR…



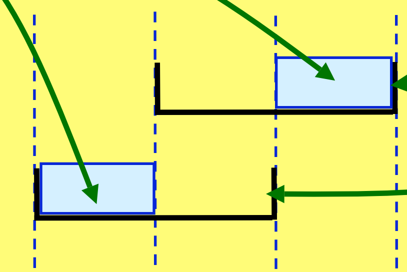
PFAIR (PD^2)

Previous example scheduled under PFAIR…



PFAIR (PD^2)

How does Pfair do it? T = (2,3) is scheduled by breaking each of its jobs into two quantum-length *subtasks* that must be scheduled within a *window* of length two:



Subtasks are prioritized on an EDF-basis and using two tie-breaking rules.

Optimality of real-time scheduling algorithms:

	uniproc.	partitioned	global
static priority			
by deadline			
PFAIR			

Real-Time Scheduling Algorithms

Optimality of real-time scheduling algorithms:

	uniproc.	partitioned	global
static priority	Hard: NO Soft: YES	Hard: NO Soft: NO	Hard: NO Soft: NO
by deadline	Hard: YES Soft: YES	Hard: NO Soft: NO	Hard: NO Soft: YES
PFAIR	Hard: (YES) Soft: (YES)	Hard: (NO) Soft: (NO)	Hard: YES Soft: YES

Real-Time Scheduling Algorithms

Optimality of real-time scheduling algorithms:

	uniproc.	partitioned	global
static priority	Hard: NO Soft: YES	Hard: NO Soft: NO	Hard: NO Soft: NO
by deadline	Hard: YES Soft: YES	Hard: NO Soft: NO	Hard: NO Soft: YES
PFAIR	Hard: (YES) Soft: (YES)	Hard: (NO) Soft: (NO)	Hard: YES Soft: YES

Optimal but high migration overheads.

Optimality of real-time scheduling algo

Less migrations but only soft-optimal!

	uniproc.	partitioned	al.
static priority	Hard: NO Soft: YES	Hard: NO Soft: NO	Hard: NO Soft: NO
by deadline	Hard: YES Soft: YES	Hard: NO Soft: NO	Hard: NO Soft: YES
PFAIR	Hard: (YES) Soft: (YES)	Hard: (NO) Soft: (NO)	Hard: YES Soft: YES

Optimality of real-time scheduling algorithms:

No migrations but not optimal at all!

	partitioned	global
static priority	Hard: **NO** Soft: **NO**	Hard: **NO** Soft: **NO**
by deadline	Hard: **NO** Soft: **NO**	Hard: **NO** Soft: **YES**
PFAIR	Hard: (**NO**) Soft: (**NO**)	Hard: **YES** Soft: **YES**

		partitioned	global
static priority	Hard: **NO** Soft: **YES**	Hard: **NO** Soft: **NO**	Hard: **NO** Soft: **NO**
by deadline	Hard: **YES** Soft: **YES**	Hard: **NO** Soft: **NO**	Hard: **NO** Soft: **YES**
PFAIR	Hard: (**YES**) Soft: (**YES**)	Hard: (**NO**) Soft: (**NO**)	Hard: **YES** Soft: **YES**

Optimality of real-time scheduling algorithms:

No migrations but

		partitioned	global
static			NO NO
by deadline	Soft: YES	Soft: NO	NO Soft: YES
PFAIR	Hard: (YES) Soft: (YES)	Hard: (NO) Soft: (NO)	Hard: YES Soft: YES

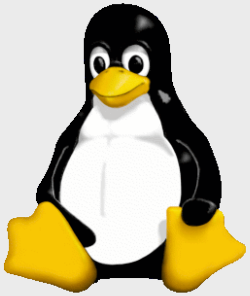
Question:

Given real overheads, what algorithm performs best in a given scenario?

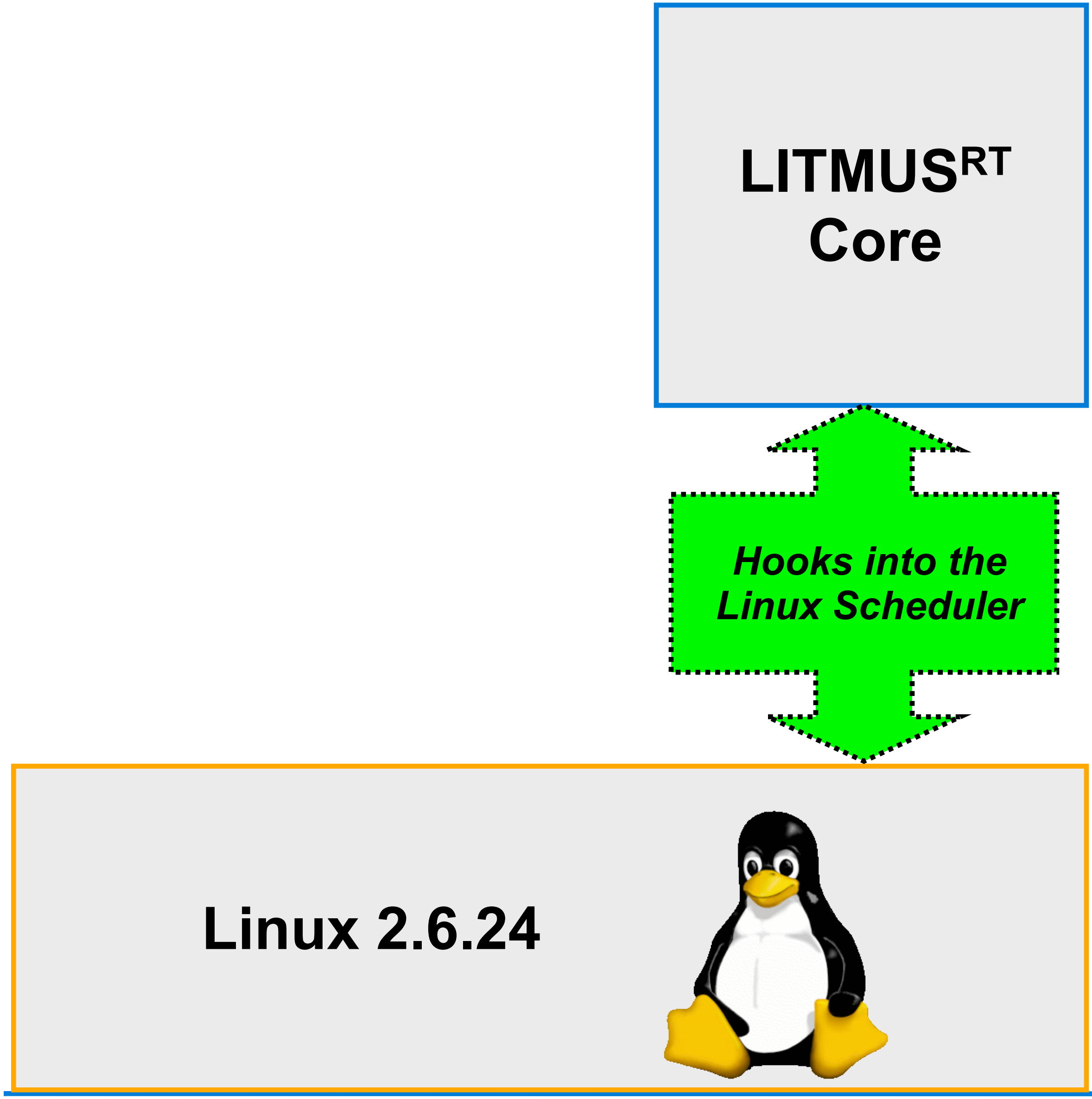
The Design of LITMUS^{RT}

The Design of LITMUS^RT

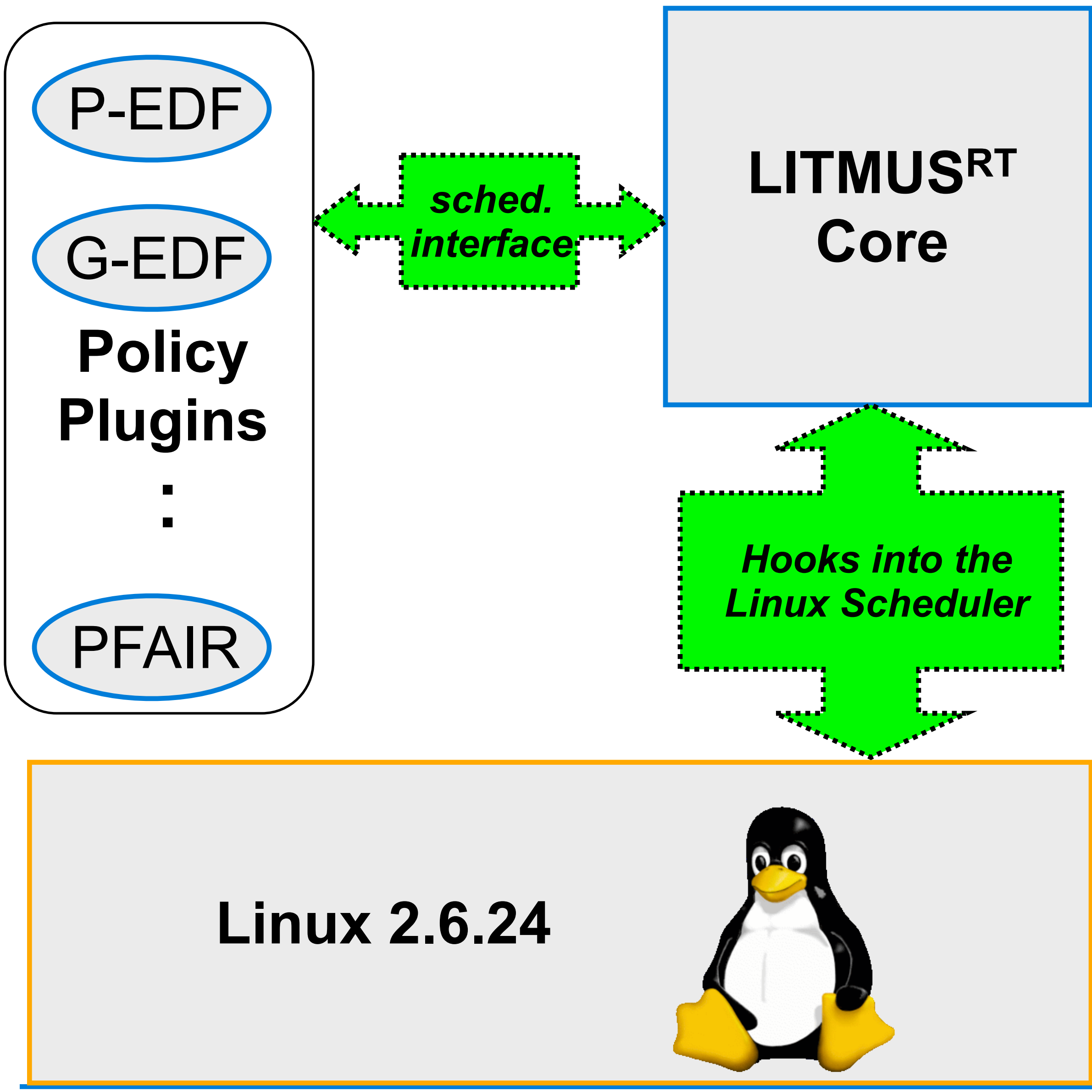
Linux 2.6.24



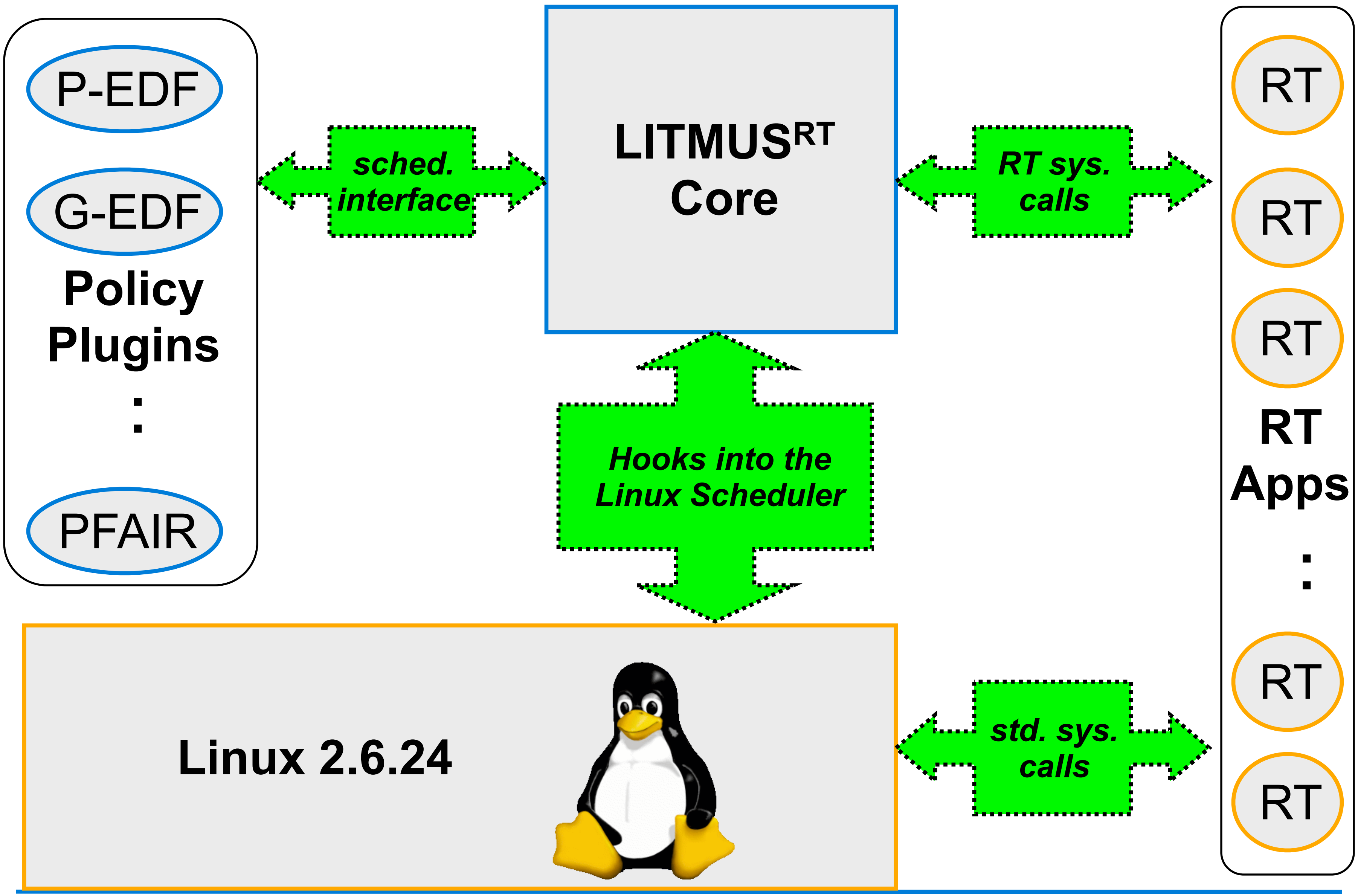
The Design of LITMUS^{RT}



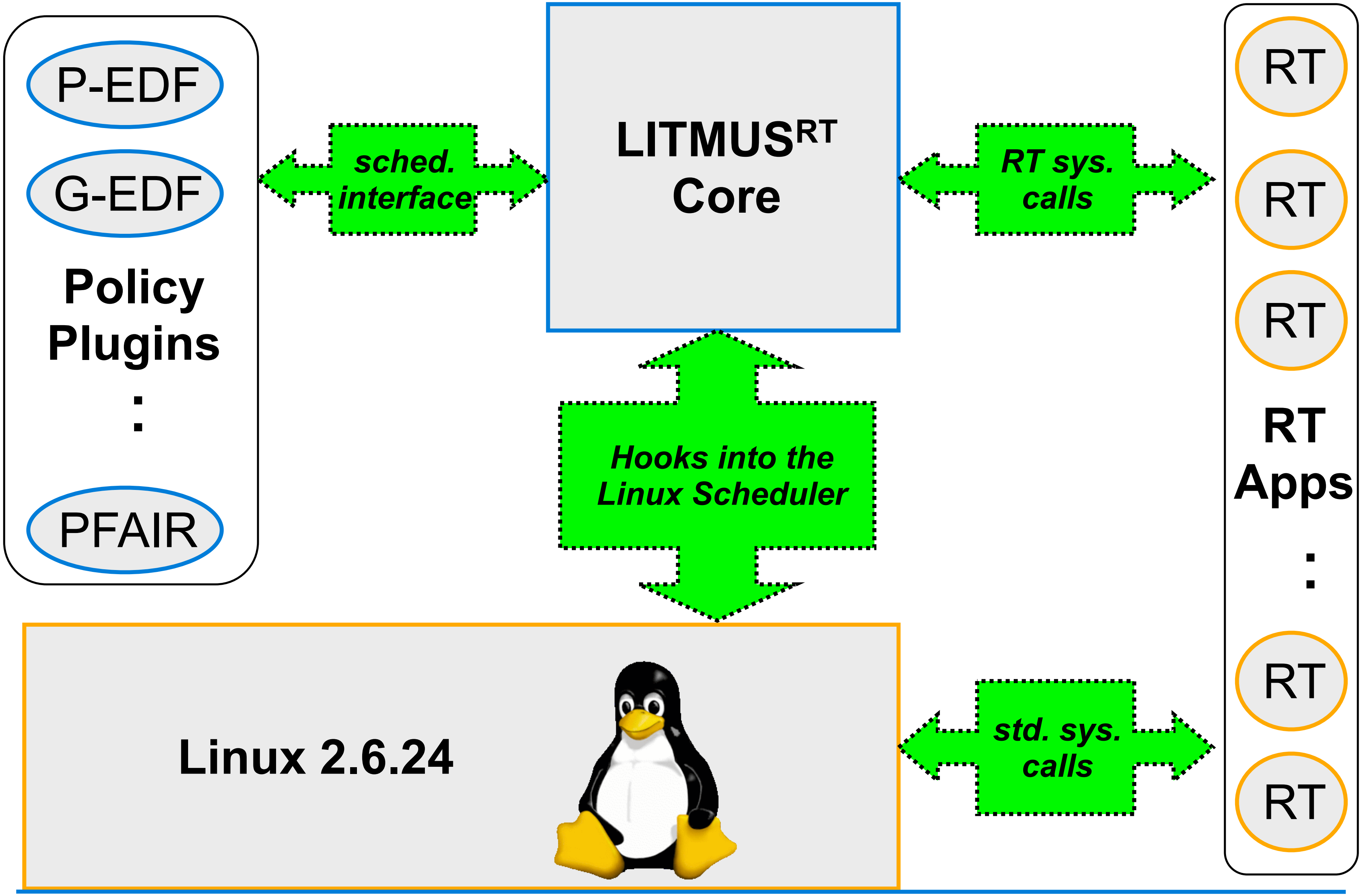
The Design of LITMUS^{RT}



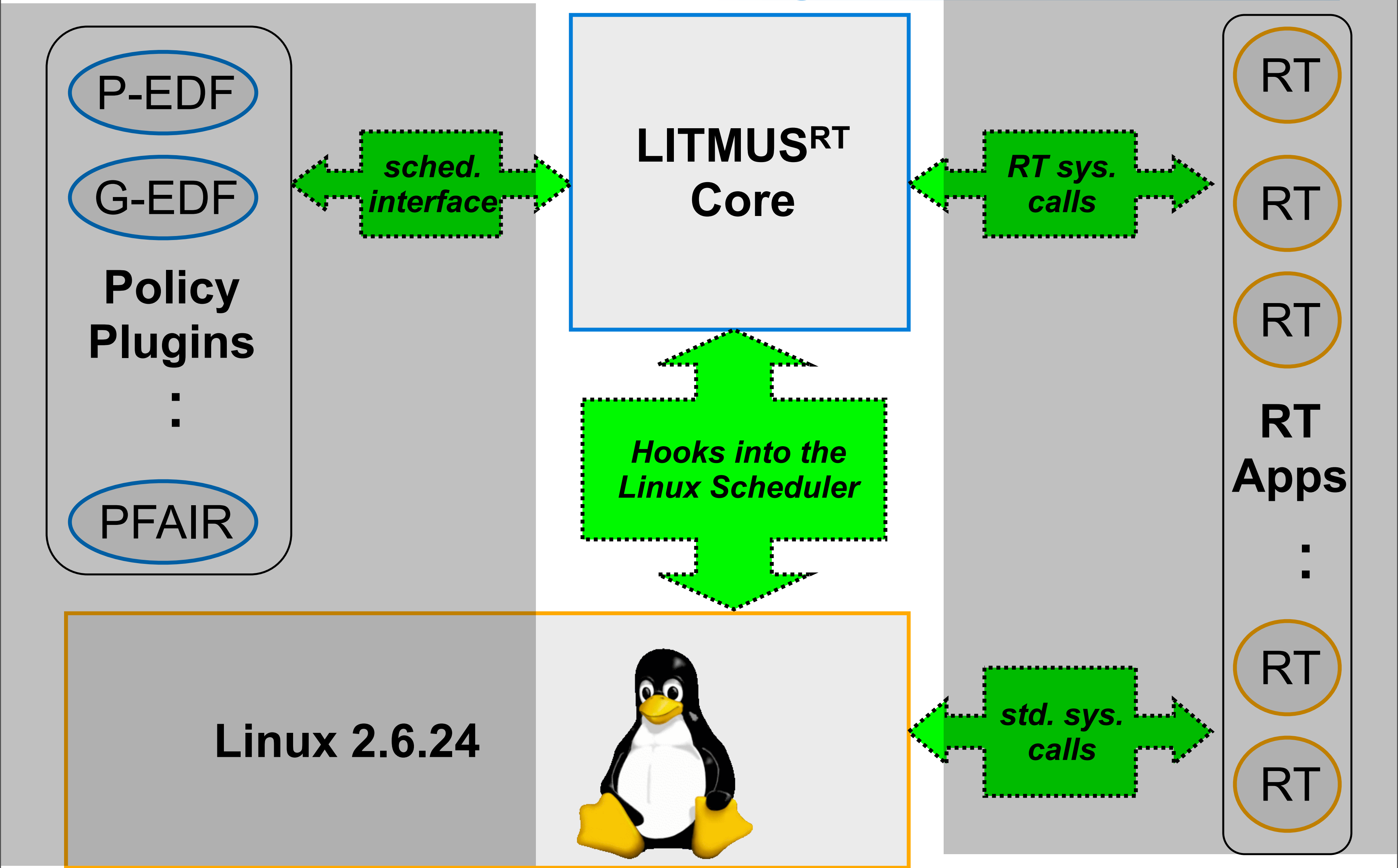
The Design of LITMUS^{RT}



The Design of LITMUS^{RT}



The Design of LITMUS^{RT}



The LITMUS^{RT} Core

LITMUS^{RT} Core = Infrastructure & Components

Linux 2.6.24

The LITMUS^{RT} Core

Binomial Heaps

- `heap_add()`
- `heap_union()`

…e = Infrastructure & Components

Linux 2.6.24

The LITMUS^{RT} Core

Binomial Heaps

- heap_add()
- heap_union()

Orders

- earlier_deadline()
- earlier_release()
- shorter_period()

...e ... Components

Linux 2.6.24

The LITMUS^{RT} Core

Binomial Heaps
- `heap_add()`
- `heap_union()`

Orders
- `earlier_deadline()`
- `earlier_release()`
- `shorter_period()`

`rt_domain_t`
- Ready queue
- Release queue
- `add()`, `take()`, etc.

Linux 2.6.24

The LITMUS^{RT} Core

Binomial Heaps
- `heap_add()`
- `heap_union()`

Orders
- `earlier_deadline()`
- `earlier_release()`
- `shorter_period()`

`rt_domain_t`
- Ready queue
- Release queue
- `add()`, `take()`, etc.

Tracing Facilities…

Synchronized Quanta…

Linux 2.6.24

The LITMUS^{RT} Core

Binomial Heaps
- `heap_add()`
- `heap_union()`

Orders
- `earlier_deadline()`
- `earlier_release()`
- `shorter_period()`

`rt_domain_t`
- Ready queue
- Release queue
- `add()`, `take()`, etc.

Tracing Facilities…

Synchronized Quanta…

scheduler_tick()

schedule()

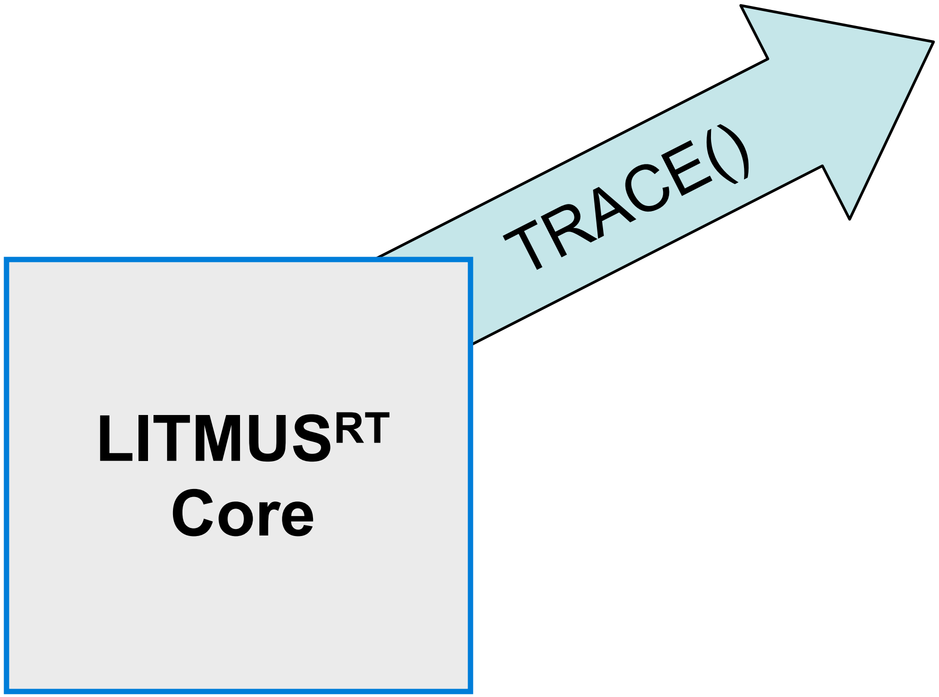
try_to_wake_up()

Linux 2.6.24

Three Tracing Facilities

LITMUS^{RT}
Core

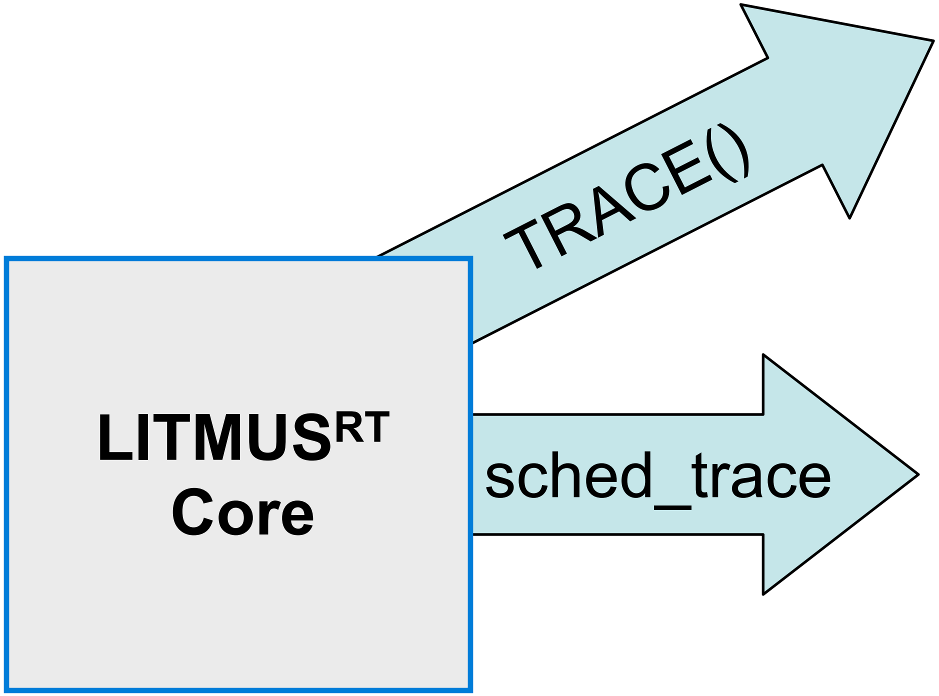
Three Tracing Facilities



Debug messages.

Plain text.

Three Tracing Facilities



Debug messages.

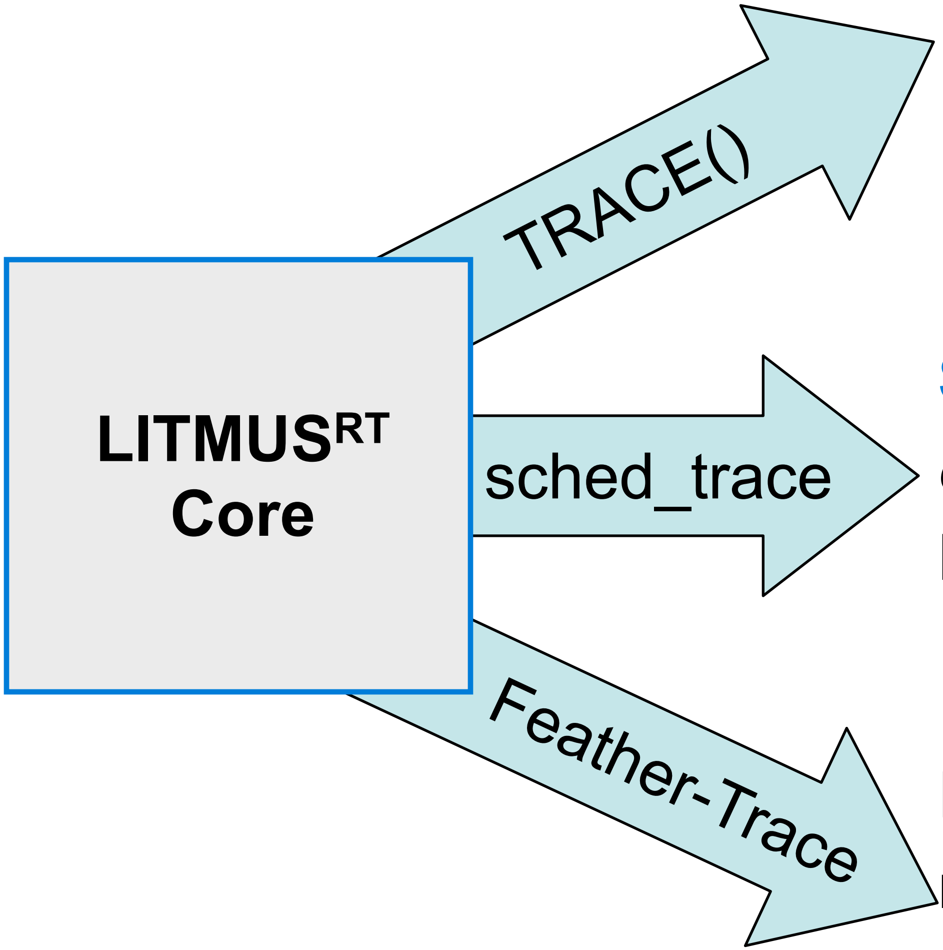
Plain text.

Scheduler events.

e.g. job completions

Binary stream.

Three Tracing Facilities



Debug messages.
Plain text.

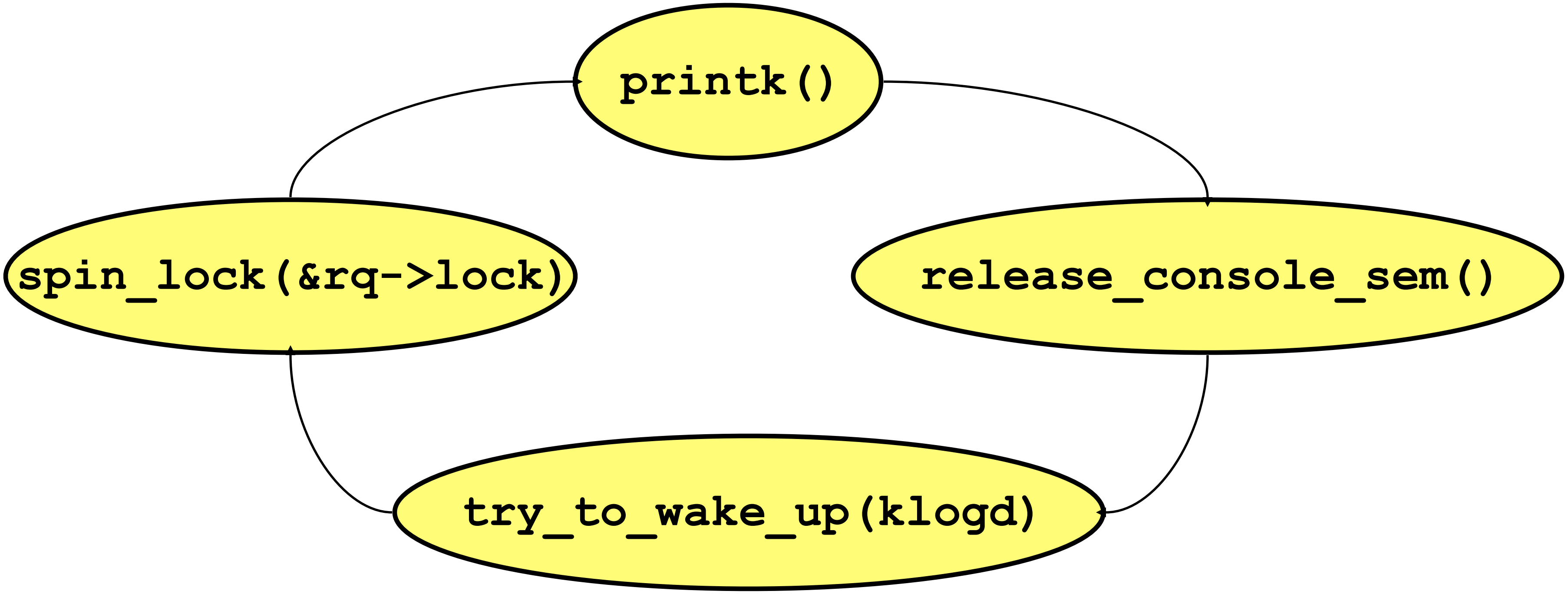
Scheduler events.
e.g. job completions
Binary stream.

Fine-grained **overhead** measurements.
Binary stream.

B. Brandenburg and J. Anderson, " Feather-Trace: A Light-Weight Event Tracing Toolkit ", *Proc. of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp. 20-27, July 2007.

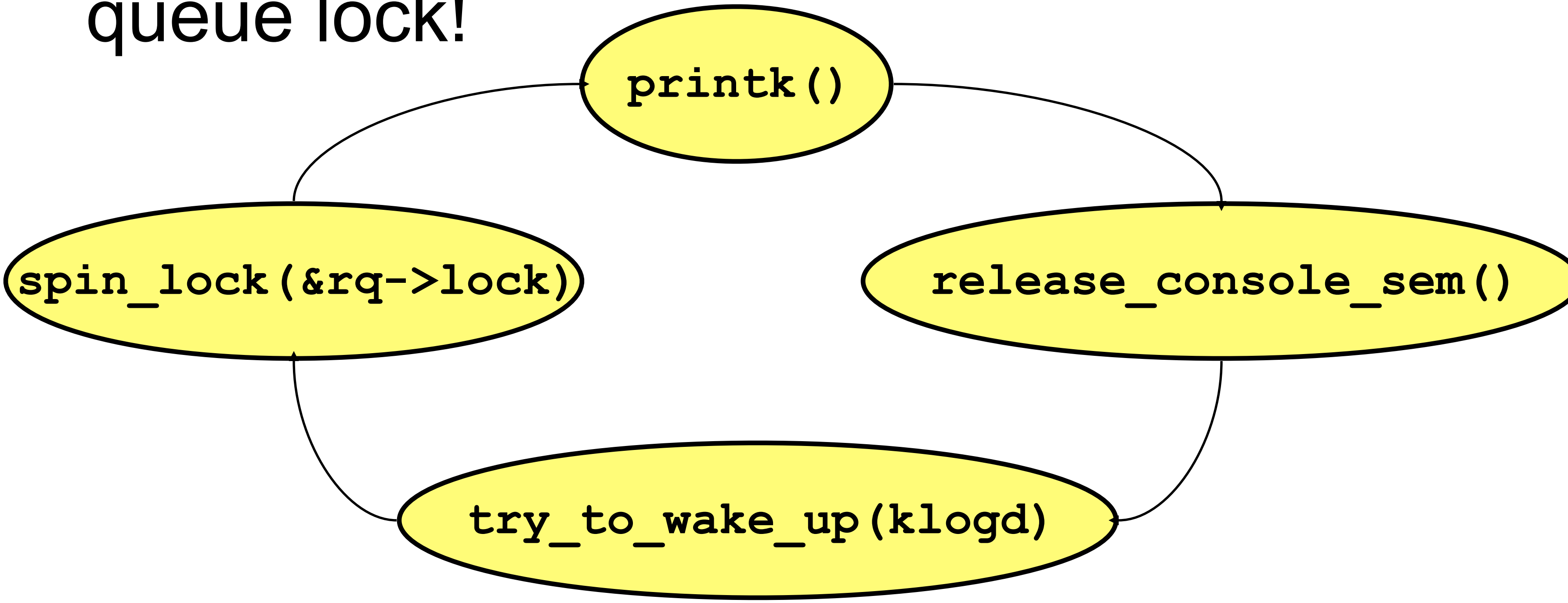
Why not use printk() for debugging?

Why not use printk() for debugging?



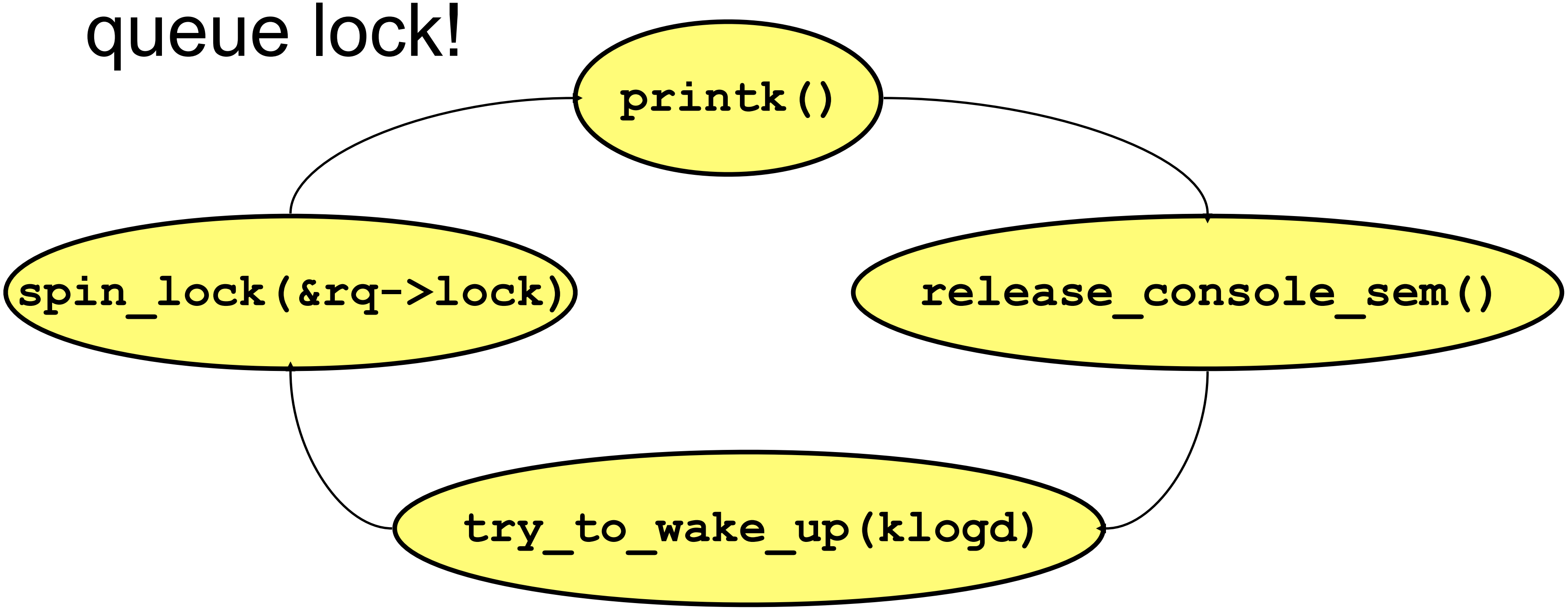
Why not use printk() for debugging?

- Can't use **`printk()`** while holding a run queue lock!



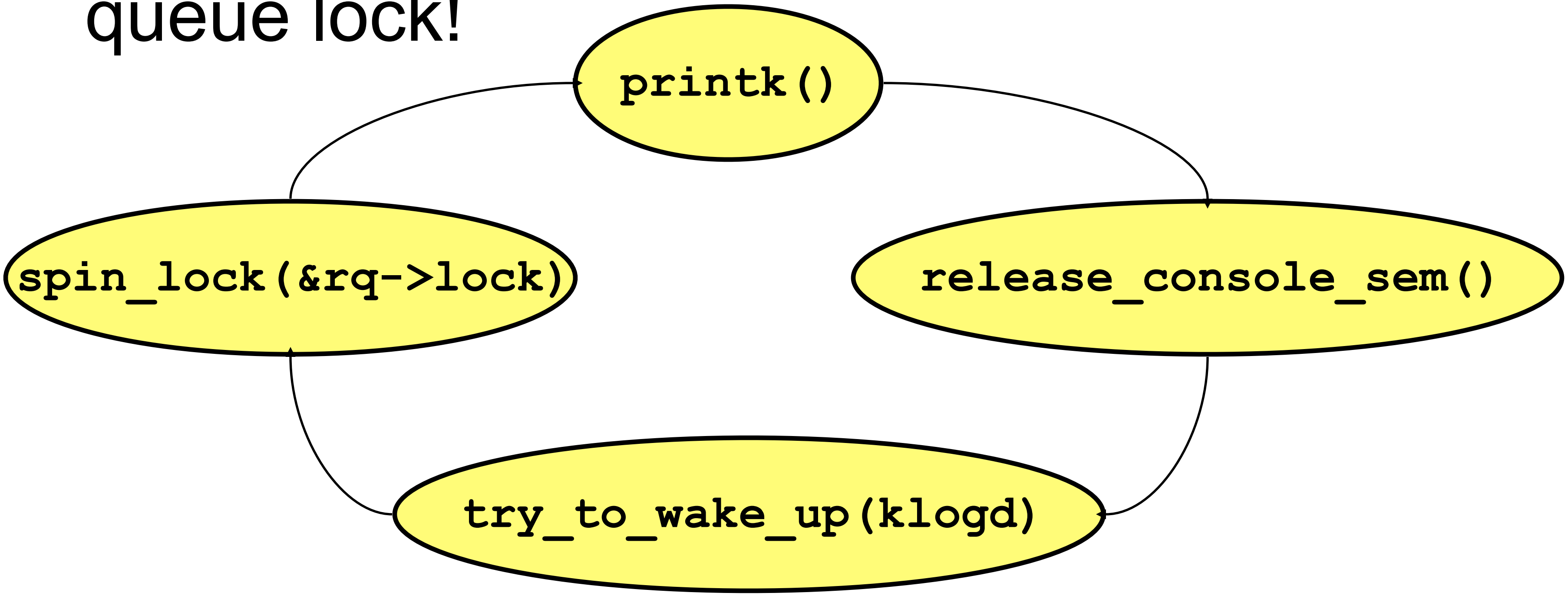
Why not use printk() for debugging?

- Can't use **`printk()`** while holding a run queue lock!



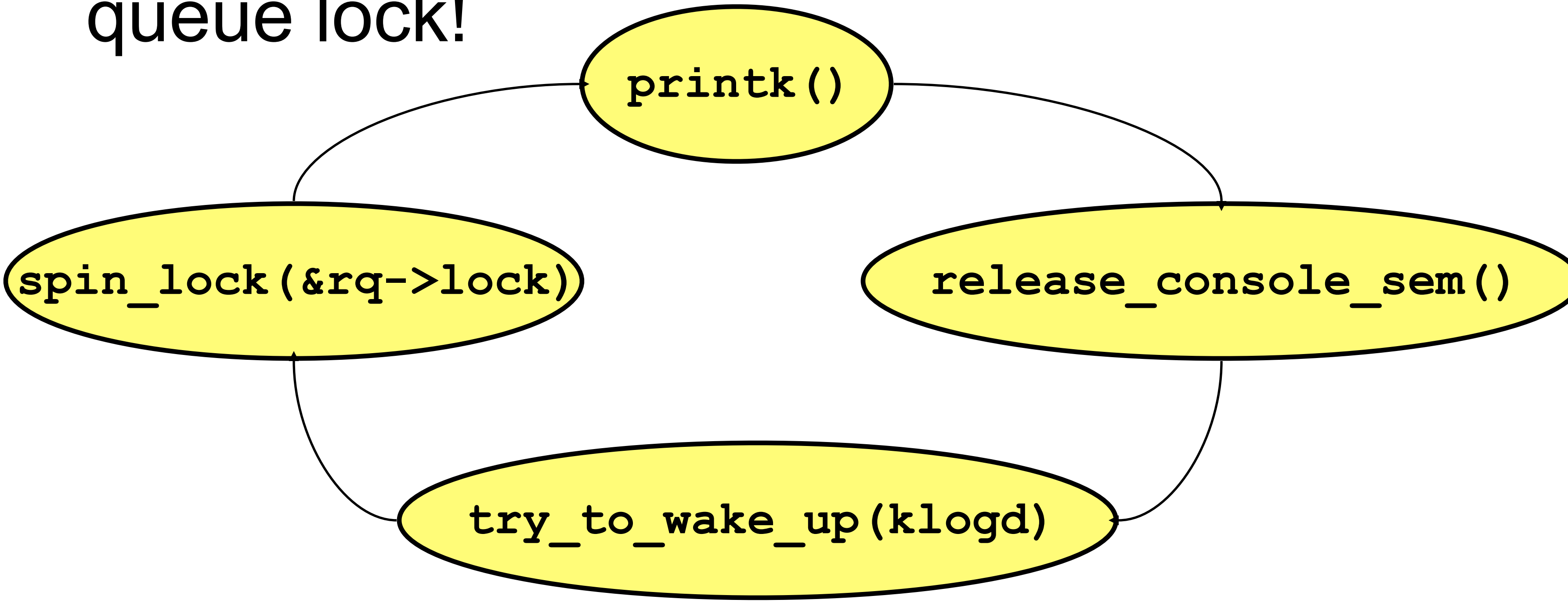
Why not use printk() for debugging?

- Can't use **`printk()`** while holding a run queue lock!



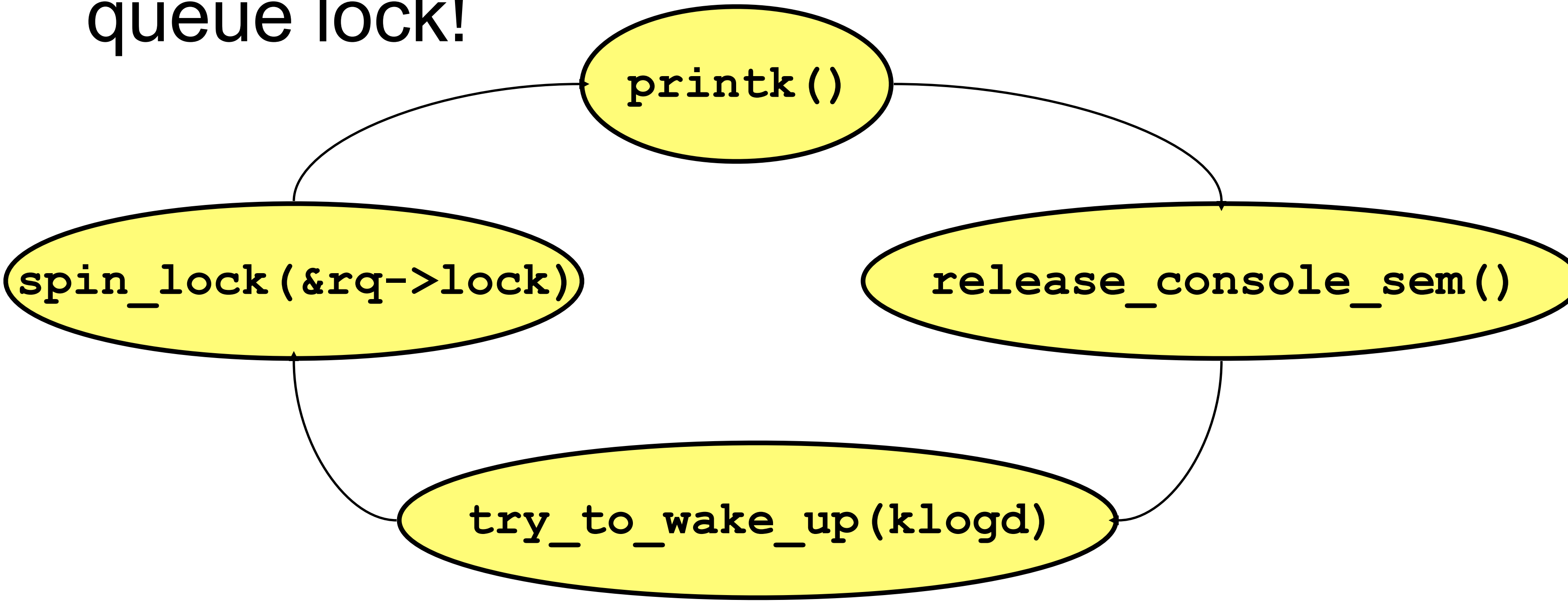
Why not use printk() for debugging?

- Can't use **`printk()`** while holding a run queue lock!



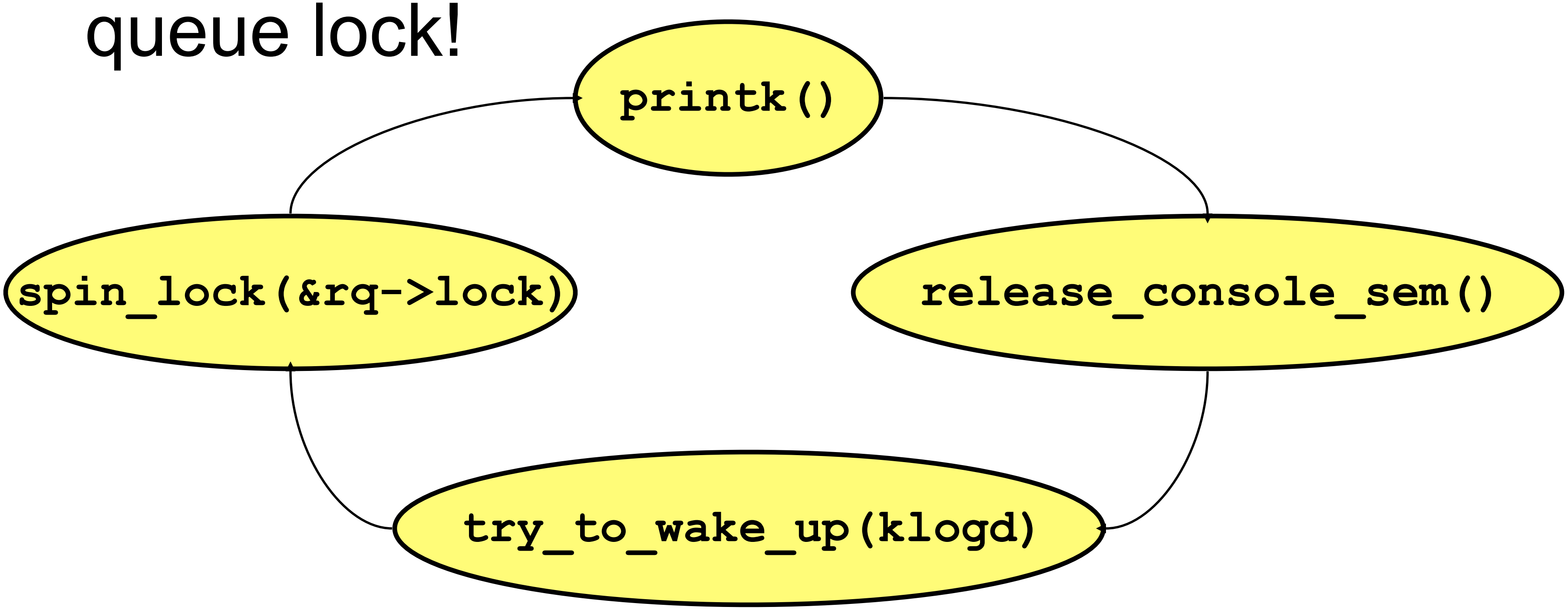
Why not use printk() for debugging?

- Can't use **`printk()`** while holding a run queue lock!



Why not use printk() for debugging?

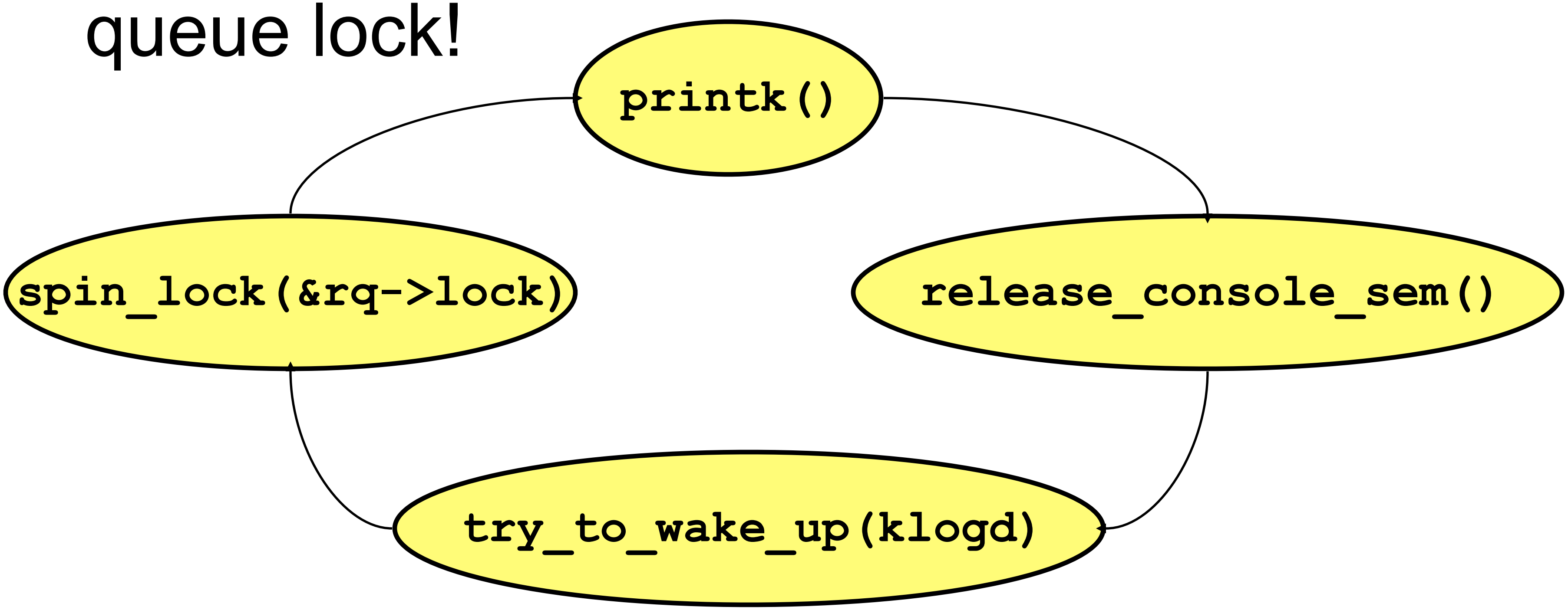
- Can't use **`printk()`** while holding a run queue lock!



- Our solution: **`TRACE()`** debugging macros.

Why not use printk() for debugging?

- Can't use `printk()` while holding a run queue lock!



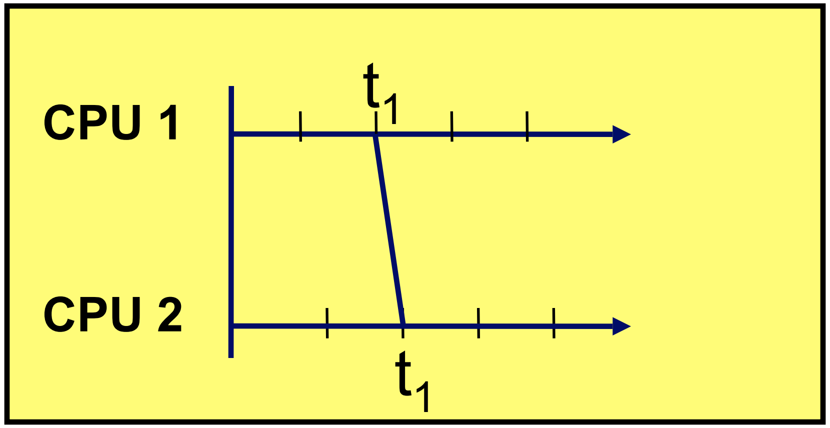
- Our solution: **`TRACE()`** debugging macros.
- Use custom **polling** char device driver.

Supporting Scheduling

- Some algorithms (esp. PFAIR) require synchronized quanta.

Supporting Scheduling

- Some algorithms (esp. PFAIR) require synchronized quanta.

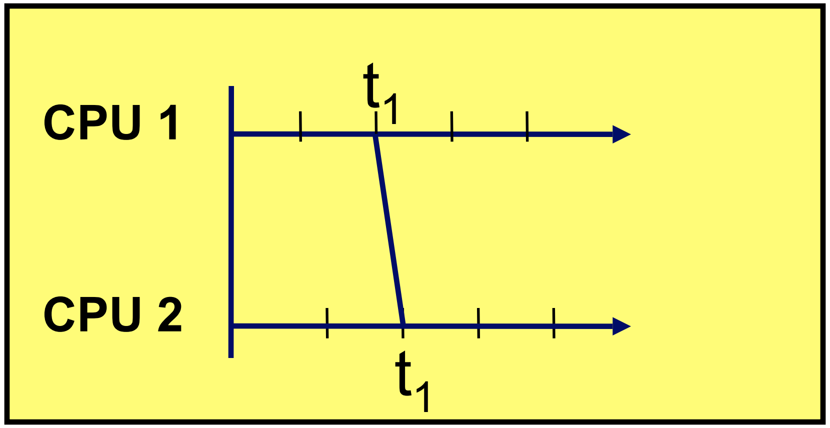


unsynchronized quanta

timer ticks are offset from each other across CPUs

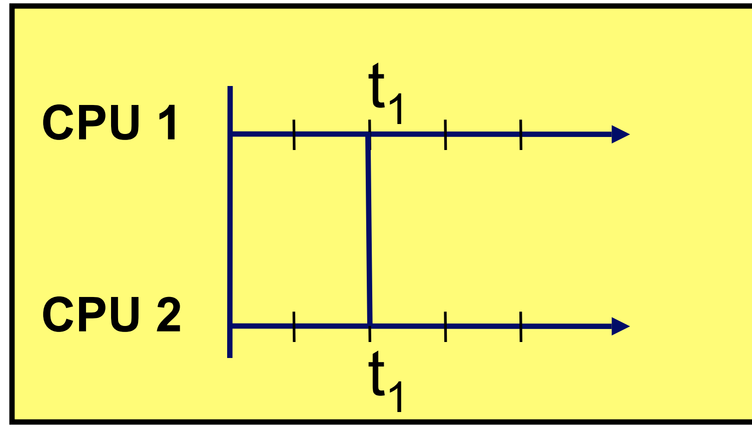
Supporting Scheduling

- Some algorithms (esp. PFAIR) require **synchronized** quanta.



unsynchronized quanta

timer ticks are offset from each other across CPUs



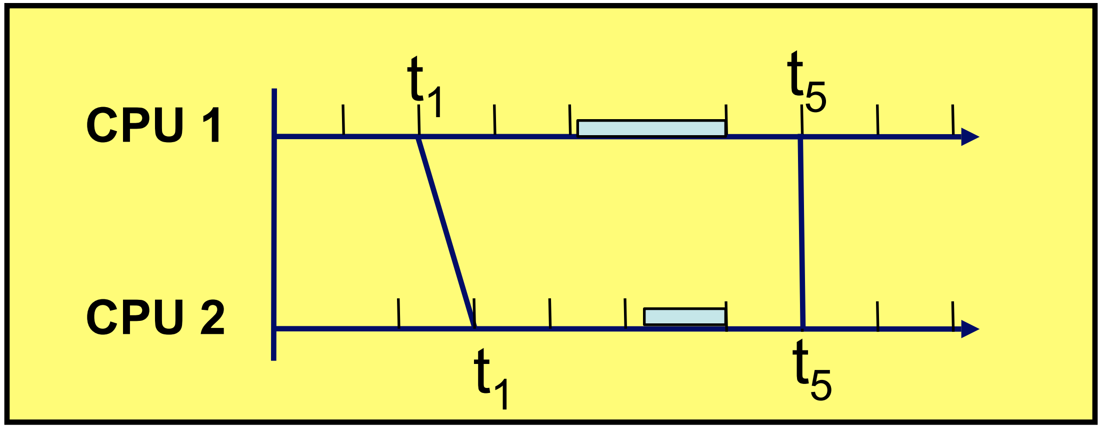
synchronized quanta

timer ticks occur at same time across CPUs

Vanilla Linux is not <u>**guaranteed**</u> to have synchronized quanta!

Synchronizing Quanta

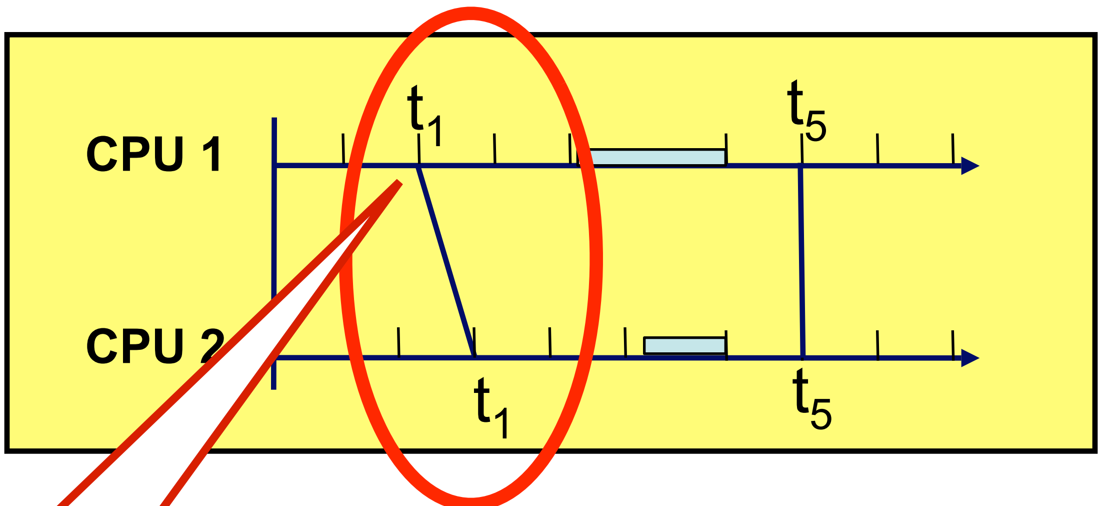
- We used to use a **barrier** to synchronize quanta at boot time (2007.x series).



Calandrino and Anderson, "Quantum Support for Multiprocessor Pfair Scheduling in Linux“, OSPERT’06

Synchronizing Quanta

- We used to use a **barrier** to synchronize quanta at boot time (2007.x series).

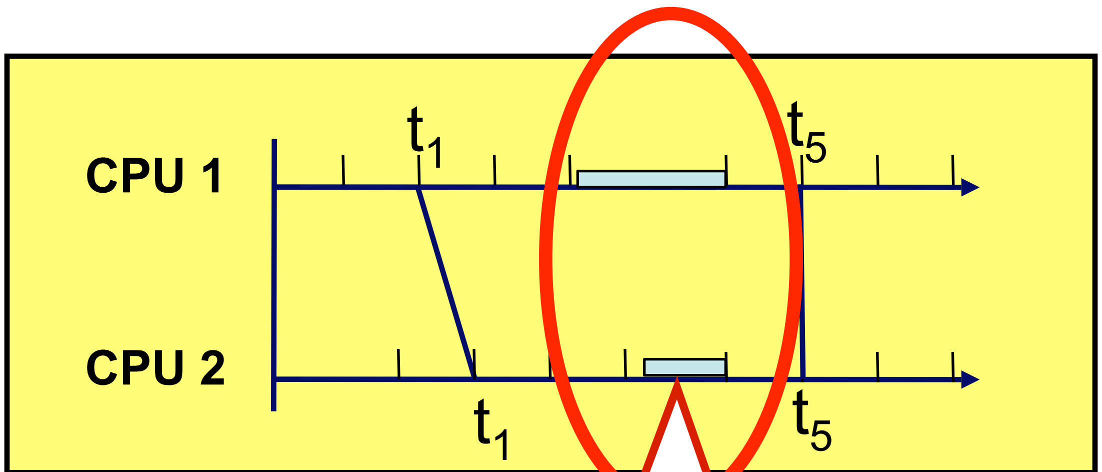


Initially quanta are unsynchronized.

ntum Support for Multiprocessor Pfair Scheduling in Linux", OSPERT'06

Synchronizing Quanta

- We used to use a **barrier** to synchronize quanta at boot time (2007.x series).

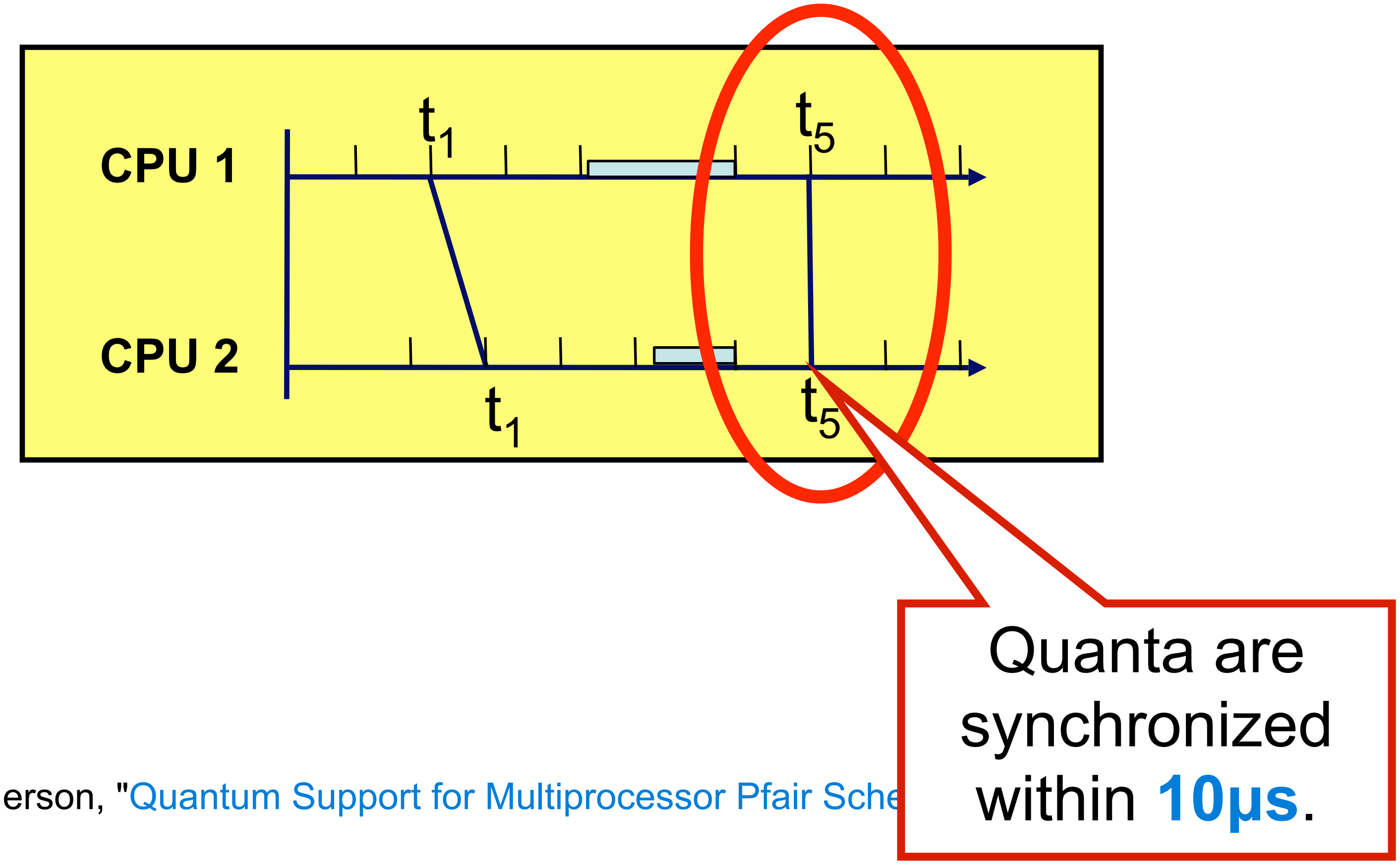


Disable APIC timer,
perform barrier,
re-enable timer.

Calandrino and Anderson, "Quan ... eduling in Linux", OSPERT'06

Synchronizing Quanta

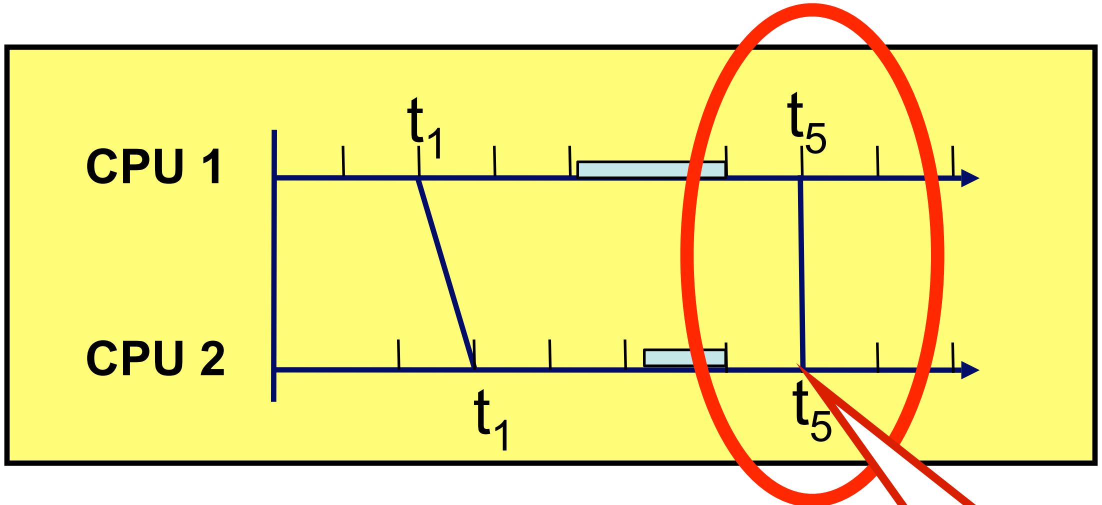
- We used to use a **barrier** to synchronize quanta at boot time (2007.x series).



Calandrino and Anderson, "Quantum Support for Multiprocessor Pfair Sche

Synchronizing Quanta

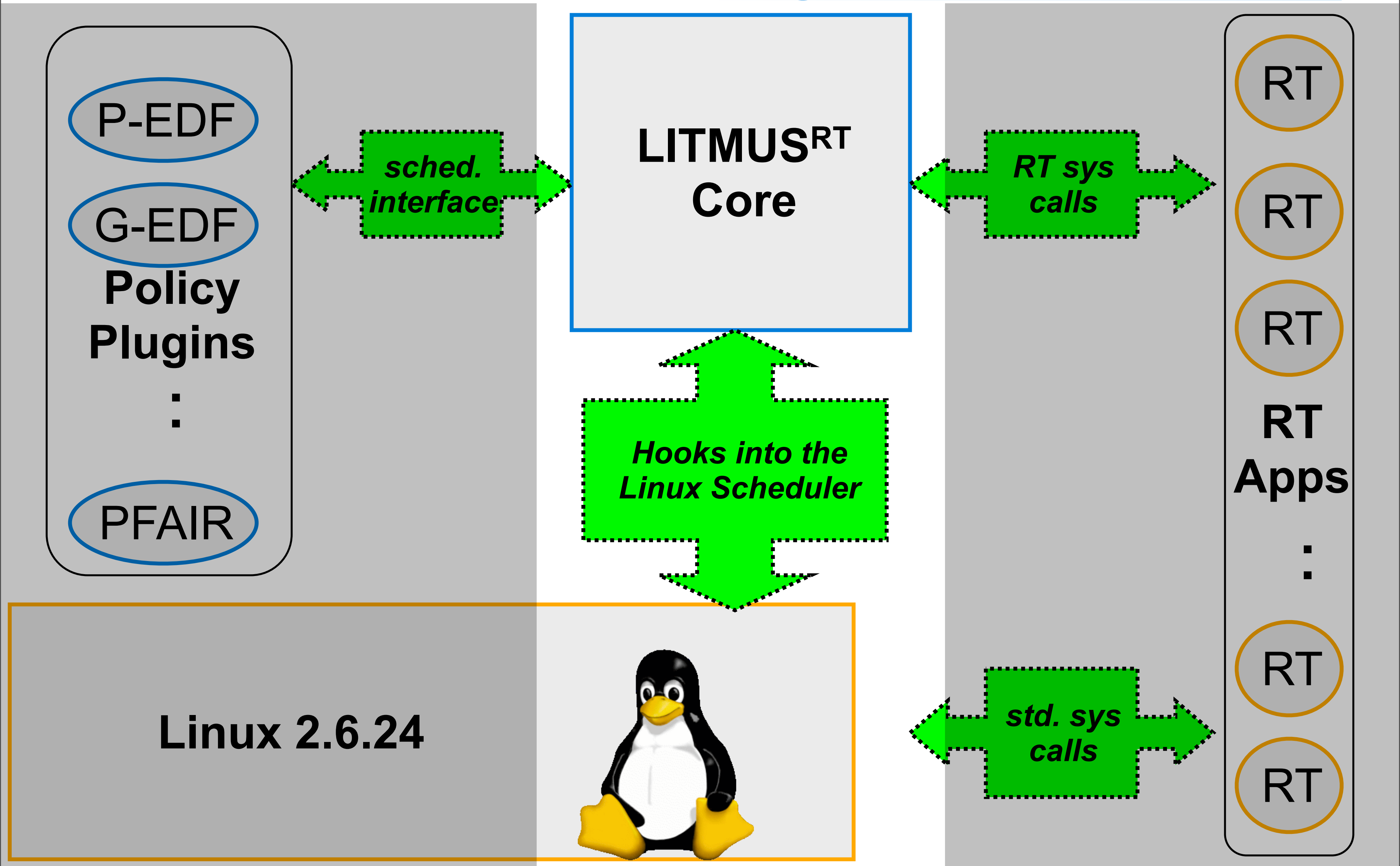
- We used to use a **barrier** to synchronize quanta at boot time (2007.x series).



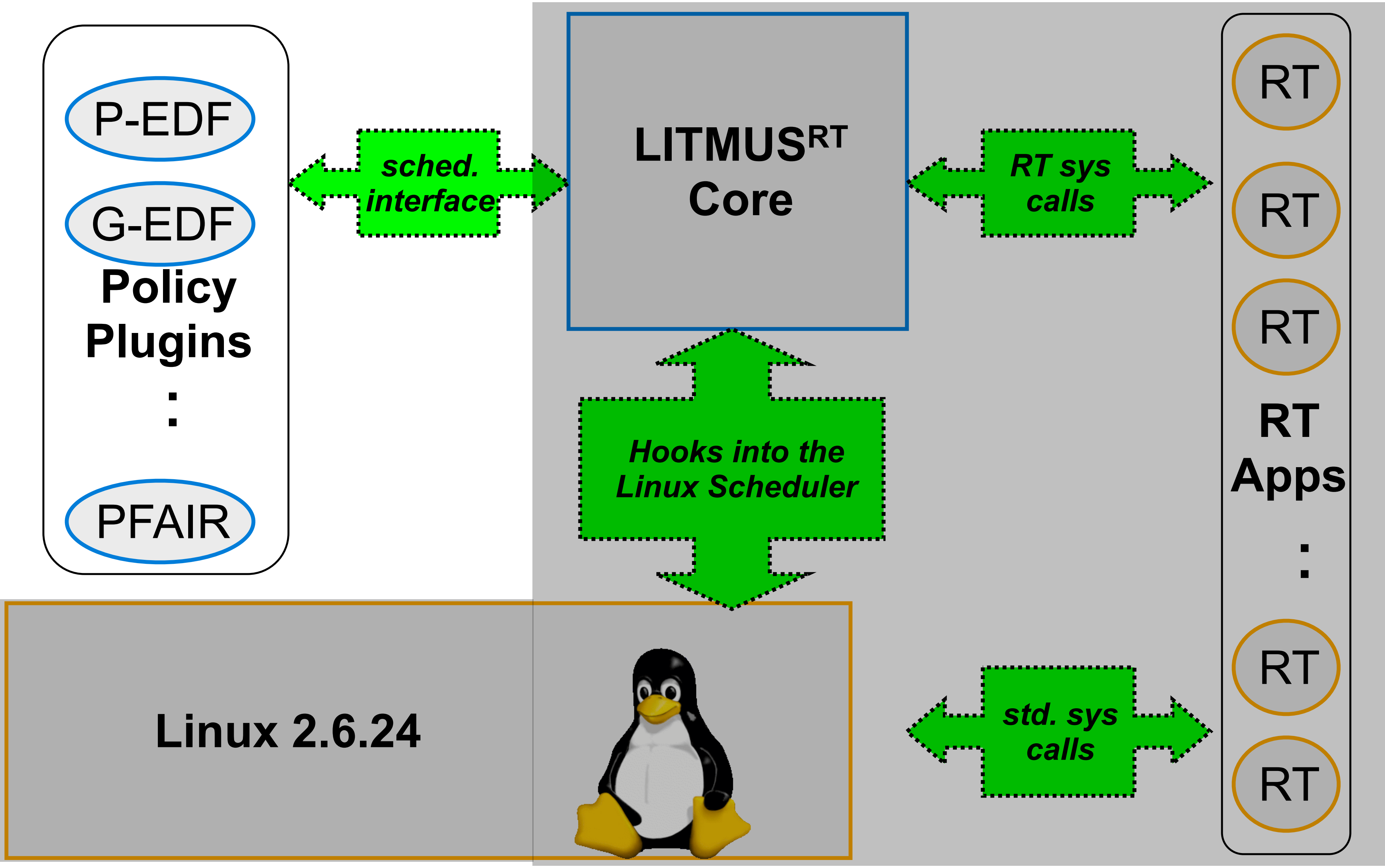
In the 2008.x series, we only need to recompute timer offsets.

Quanta are synchronized within **10μs**.

The Design of LITMUS^{RT}



The Design of LITMUS^{RT}



Scheduling Policy Plugins

- LITMUS^{RT} 2007.3 contains eight plugins

Partitioned	Global

Scheduling Policy Plugins

- LITMUS^RT 2007.3 contains eight plugins

Partitioned	Global
P-EDF	

Scheduling Policy Plugins

- LITMUS^{RT} 2007.3 contains eight plugins

Partitioned	Global
P-EDF	G-EDF

Scheduling Policy Plugins

- LITMUS^{RT} 2007.3 contains eight plugins

Partitioned	Global
P-EDF	G-EDF
	G-NP-EDF

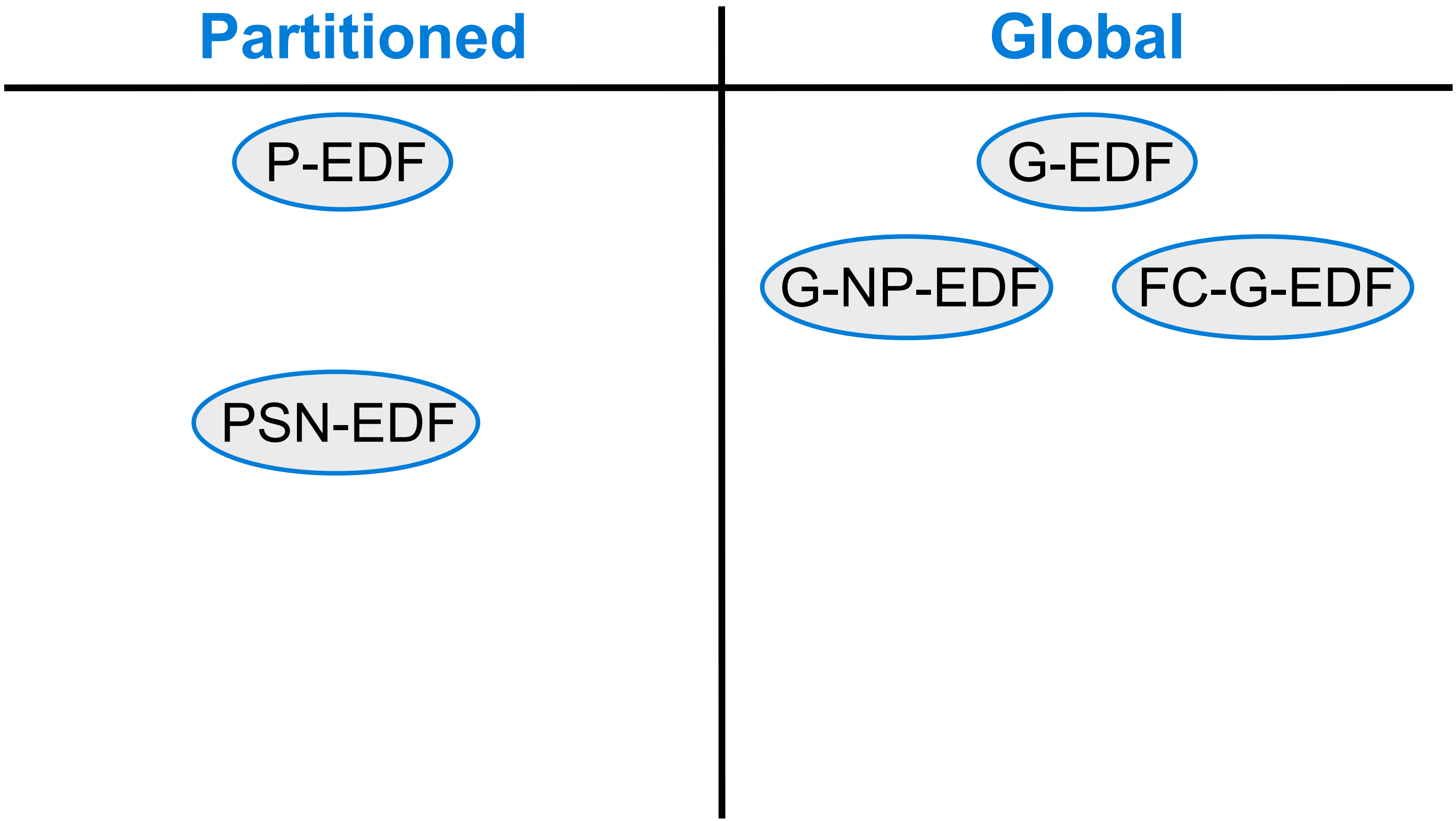
Scheduling Policy Plugins

- LITMUS^{RT} 2007.3 contains eight plugins

Partitioned	Global
P-EDF	G-EDF
	G-NP-EDF
	FC-G-EDF

Scheduling Policy Plugins

- LITMUS^{RT} 2007.3 contains eight plugins



Scheduling Policy Plugins

- LITMUS^{RT} 2007.3 contains eight plugins

Partitioned

- P-EDF
- PSN-EDF

Global

- G-EDF
- G-NP-EDF
- FC-G-EDF
- GSN-EDF

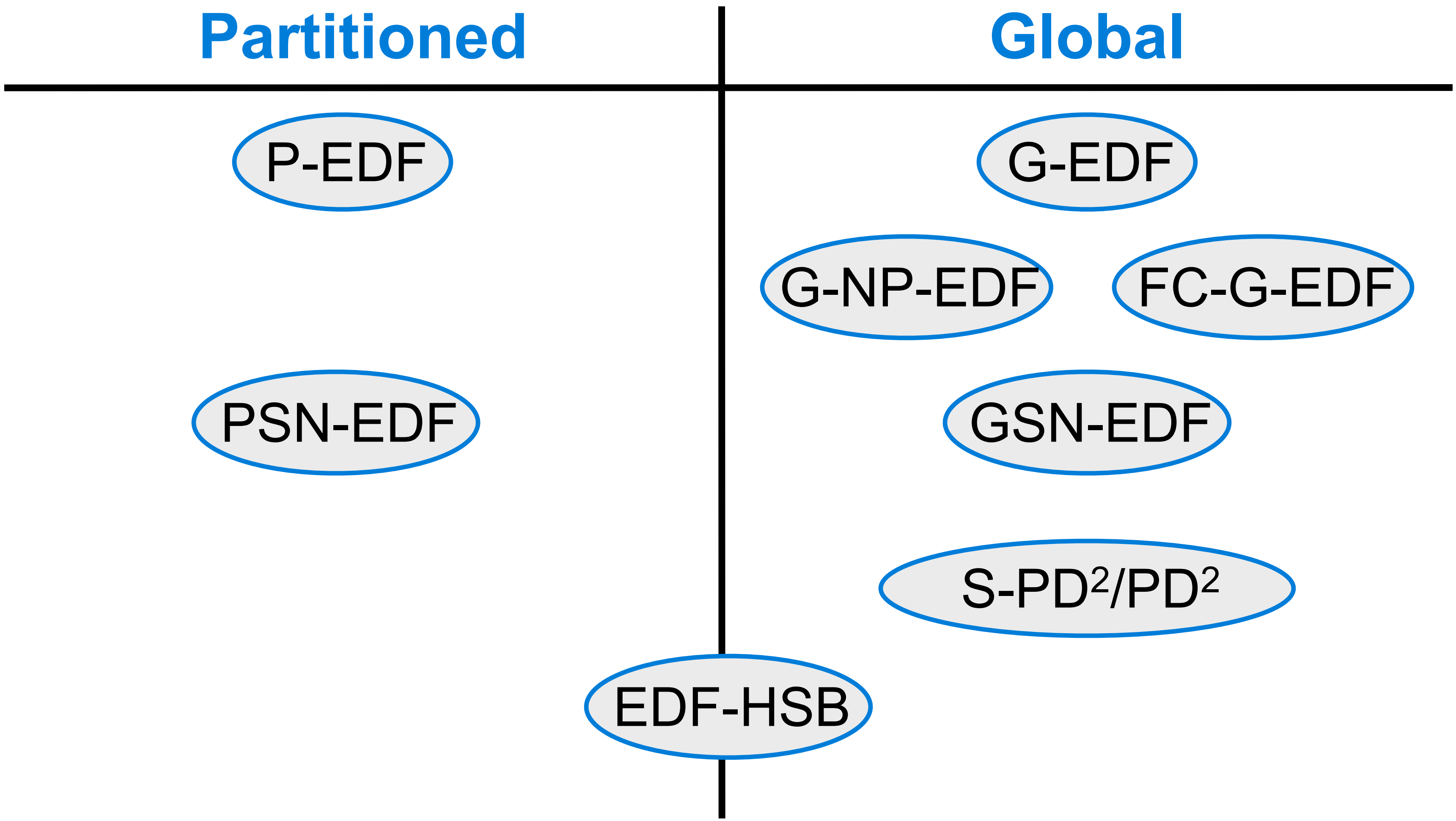
Scheduling Policy Plugins

- LITMUS^{RT} 2007.3 contains eight plugins

Partitioned	Global
P-EDF	G-EDF
	G-NP-EDF
	FC-G-EDF
PSN-EDF	GSN-EDF
	S-PD^2/PD^2

Scheduling Policy Plugins

- LITMUS^{RT} 2007.3 contains eight plugins



Scheduling Policy Plugins

- LITMUS^{RT} 2008.1 contains four plugins

Partitioned	Global

Scheduling Policy Plugins

- LITMUS^{RT} 2008.1 contains four plugins

Partitioned	Global
PSN-EDF	

Scheduling Policy Plugins

- LITMUS^{RT} 2008.1 contains four plugins

Partitioned	Global
PSN-EDF	GSN-EDF

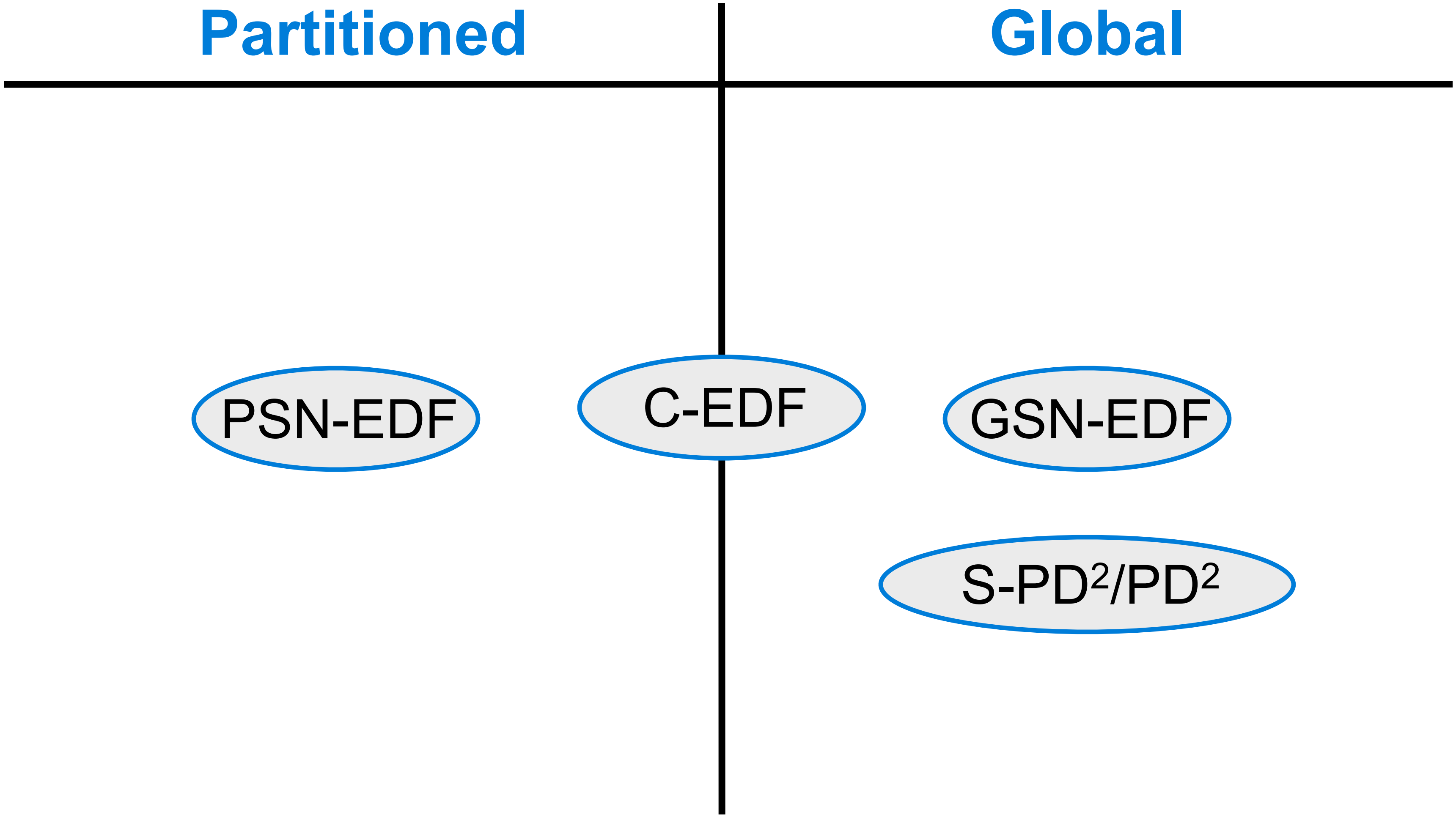
Scheduling Policy Plugins

- LITMUS^{RT} 2008.1 contains four plugins

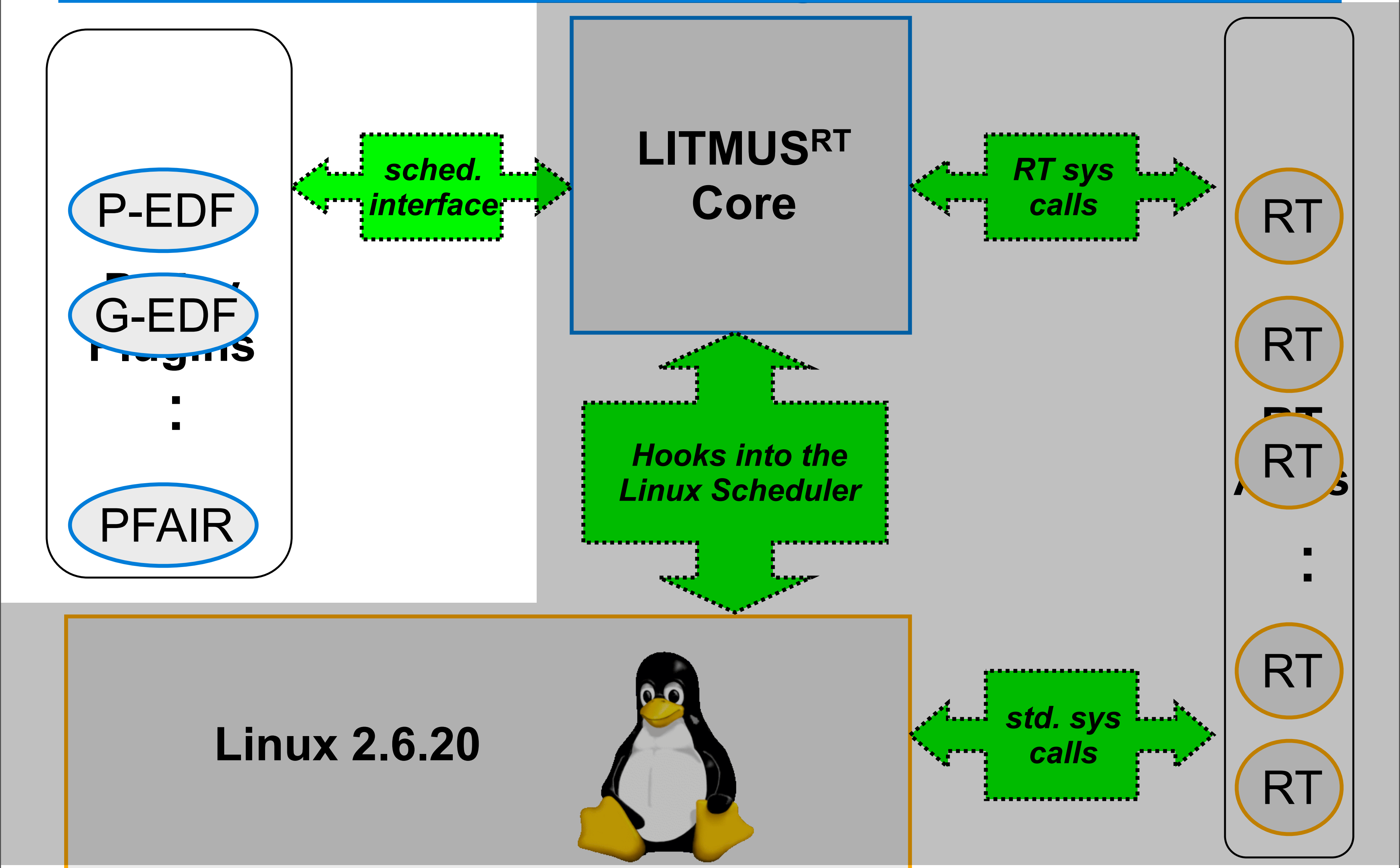
Partitioned	Global
PSN-EDF	GSN-EDF
	S-PD^2/PD^2

Scheduling Policy Plugins

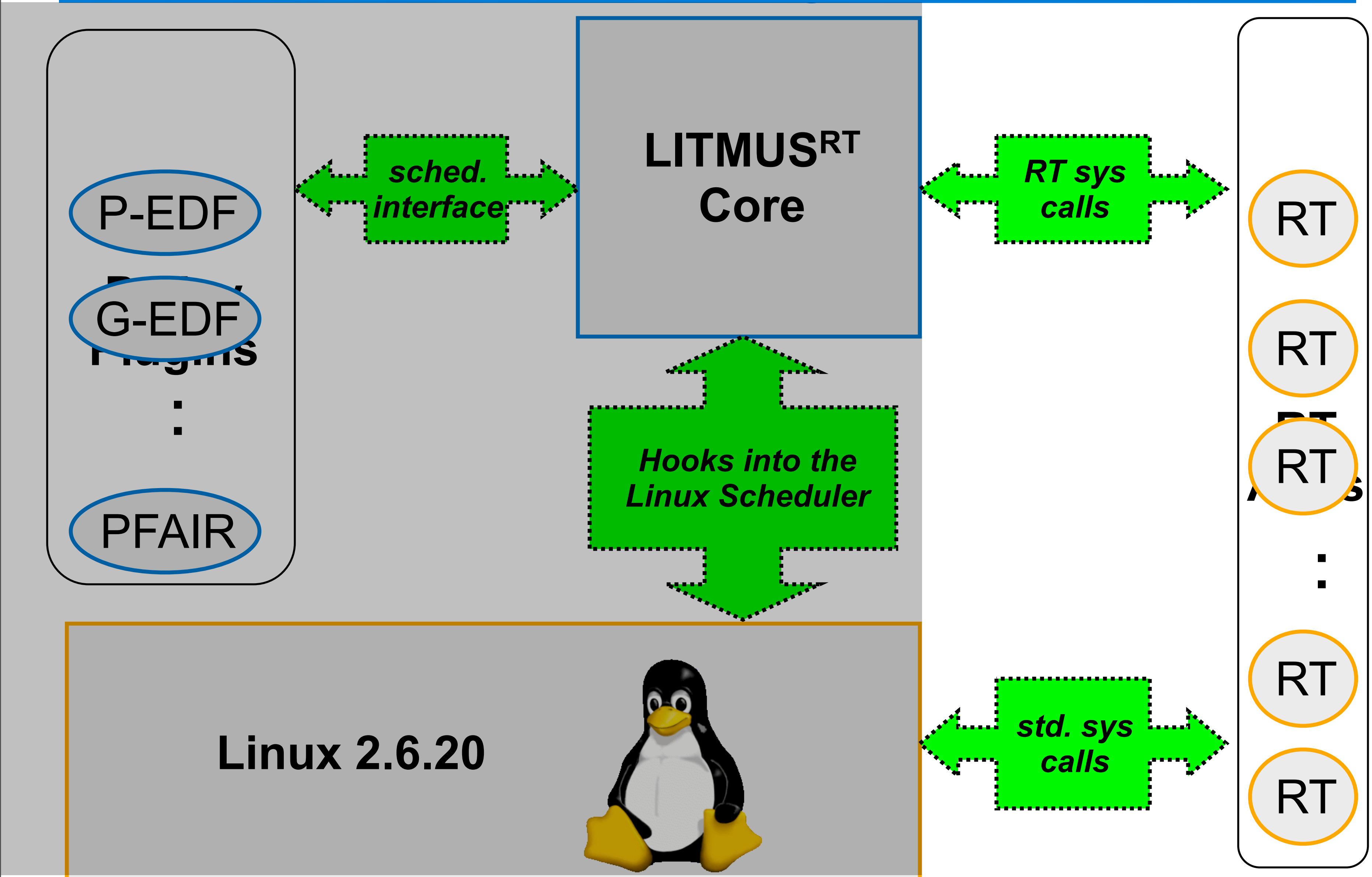
- LITMUS^{RT} 2008.1 contains four plugins



The Design of LITMUS^{RT}



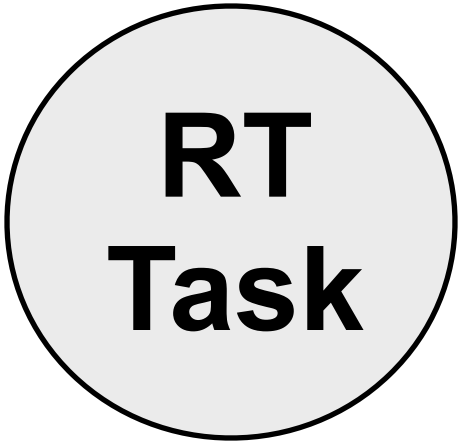
The Design of LITMUS^{RT}



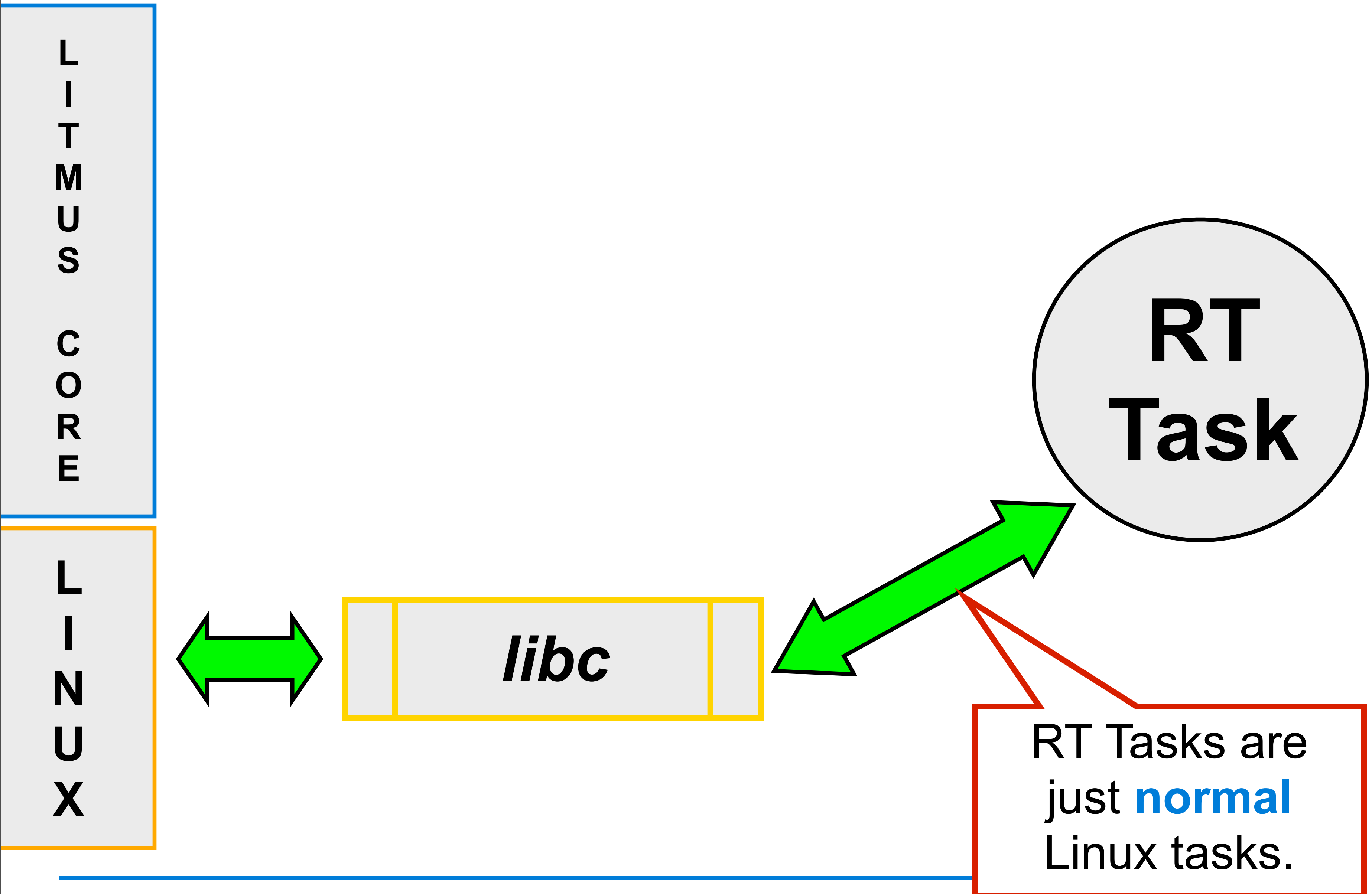
Userspace API

LITMUS CORE

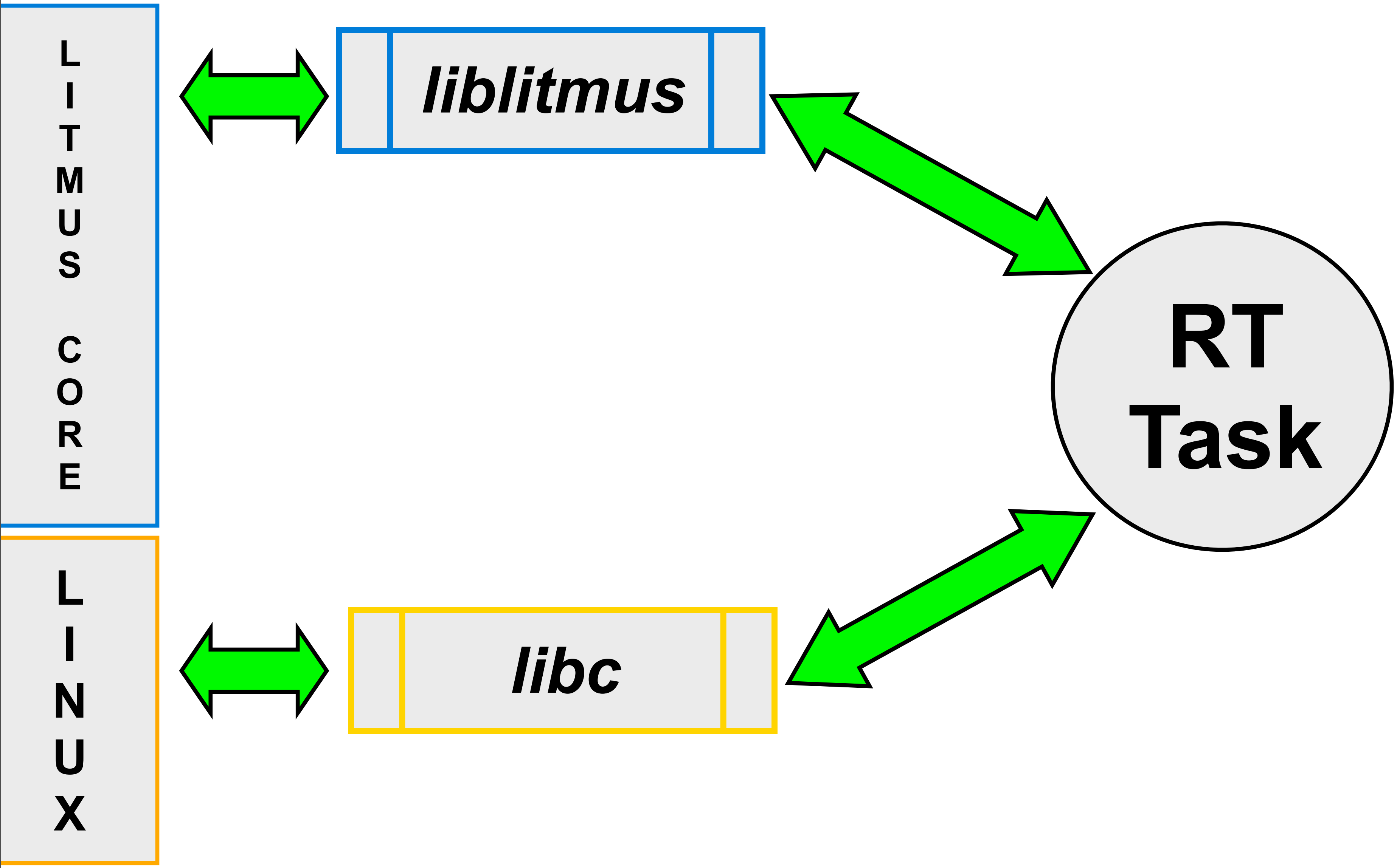
LINUX



Userspace API

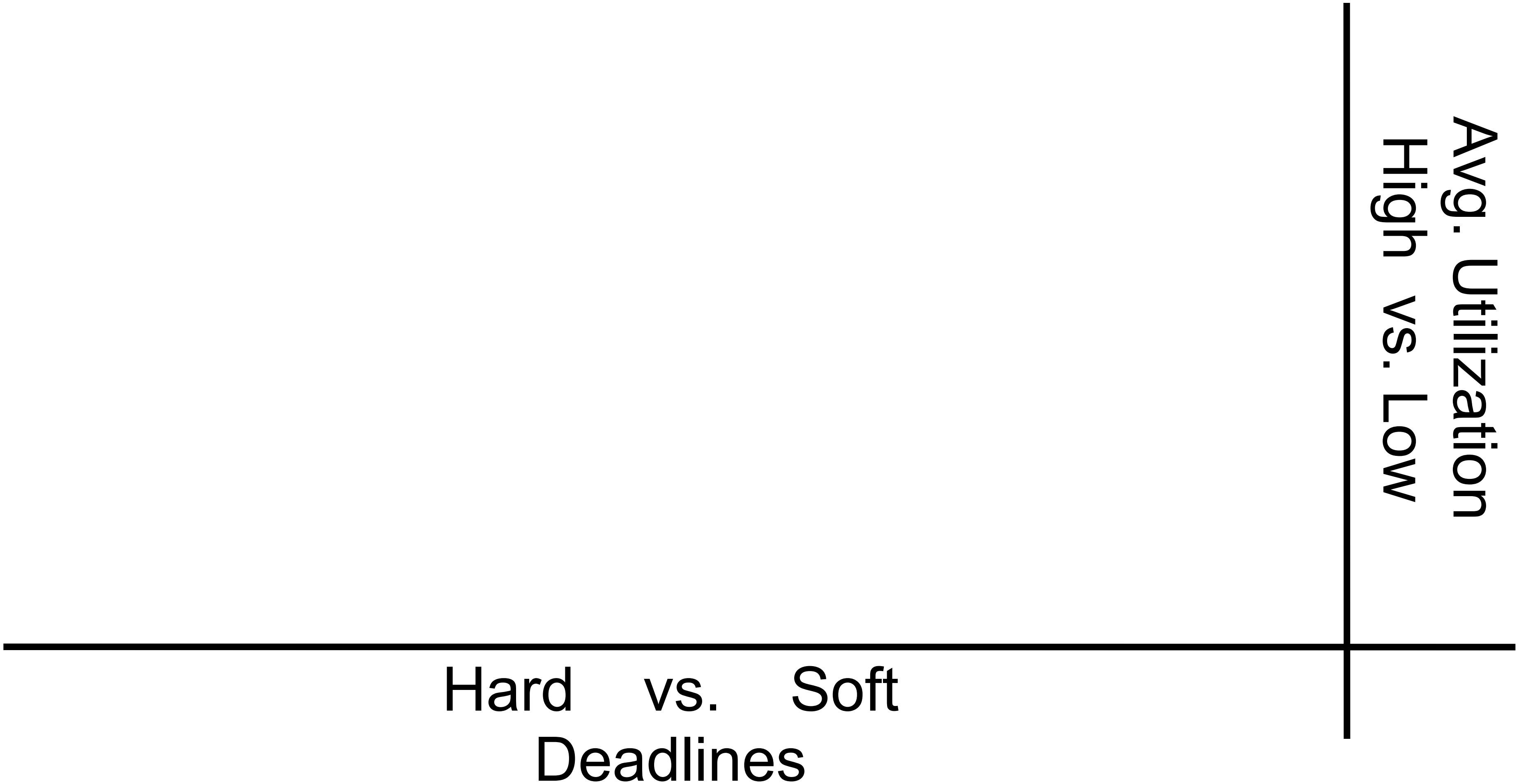


Userspace API



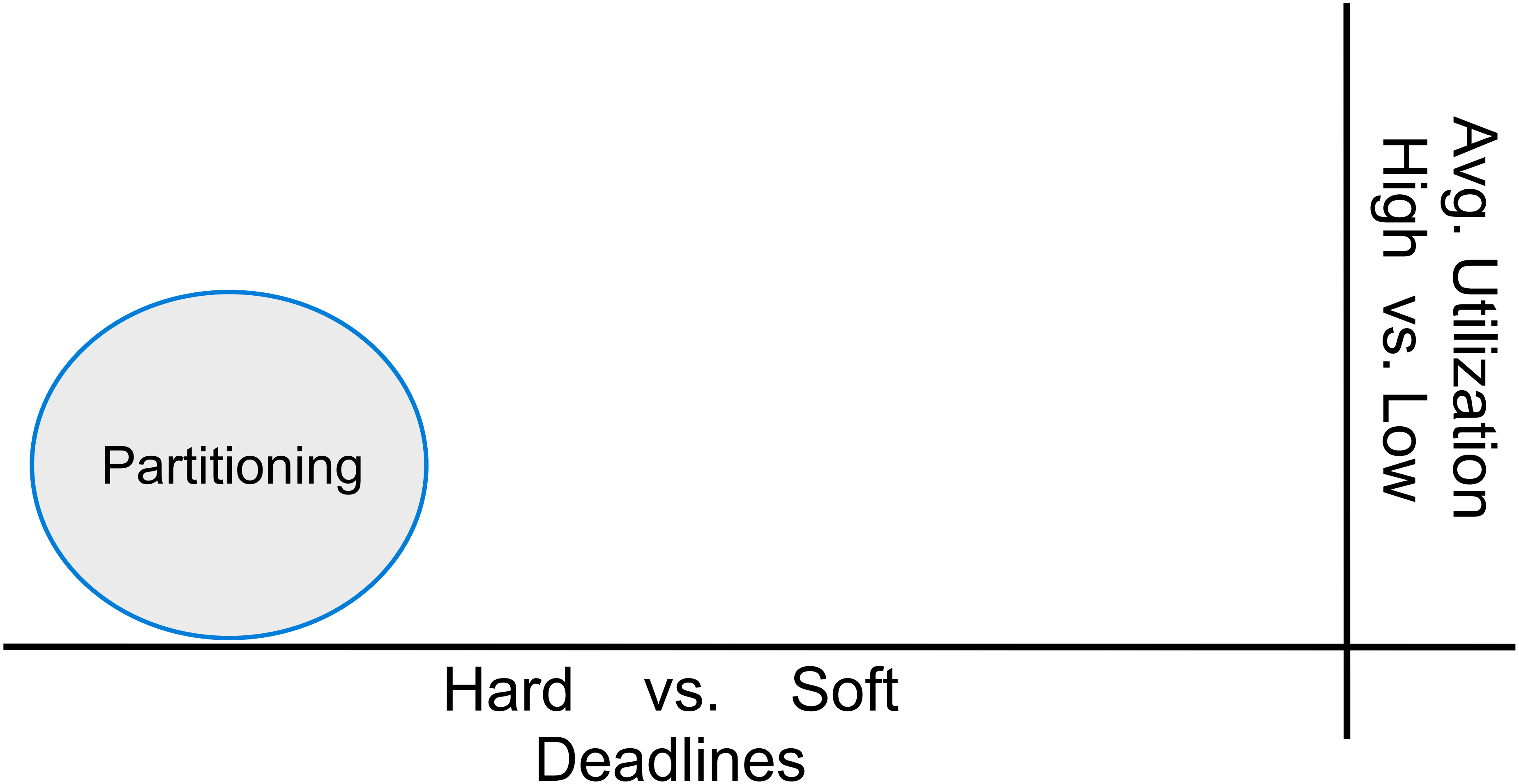
Evaluation of Scheduling Algorithms

When (if ever) should you use partitioning (global)?



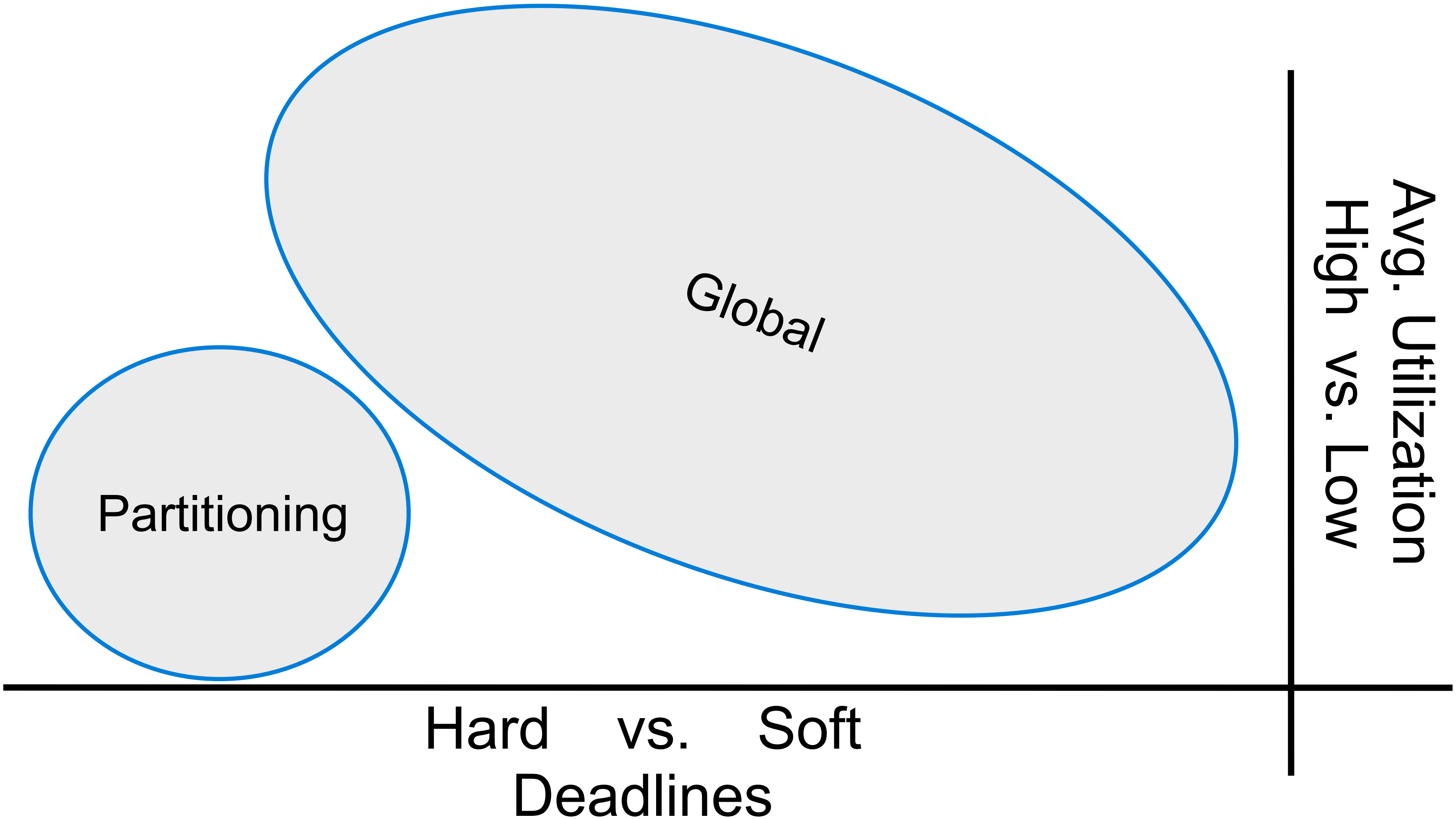
Evaluation of Scheduling Algorithms

When (if ever) should you use partitioning (global)?



Evaluation of Scheduling Algorithms

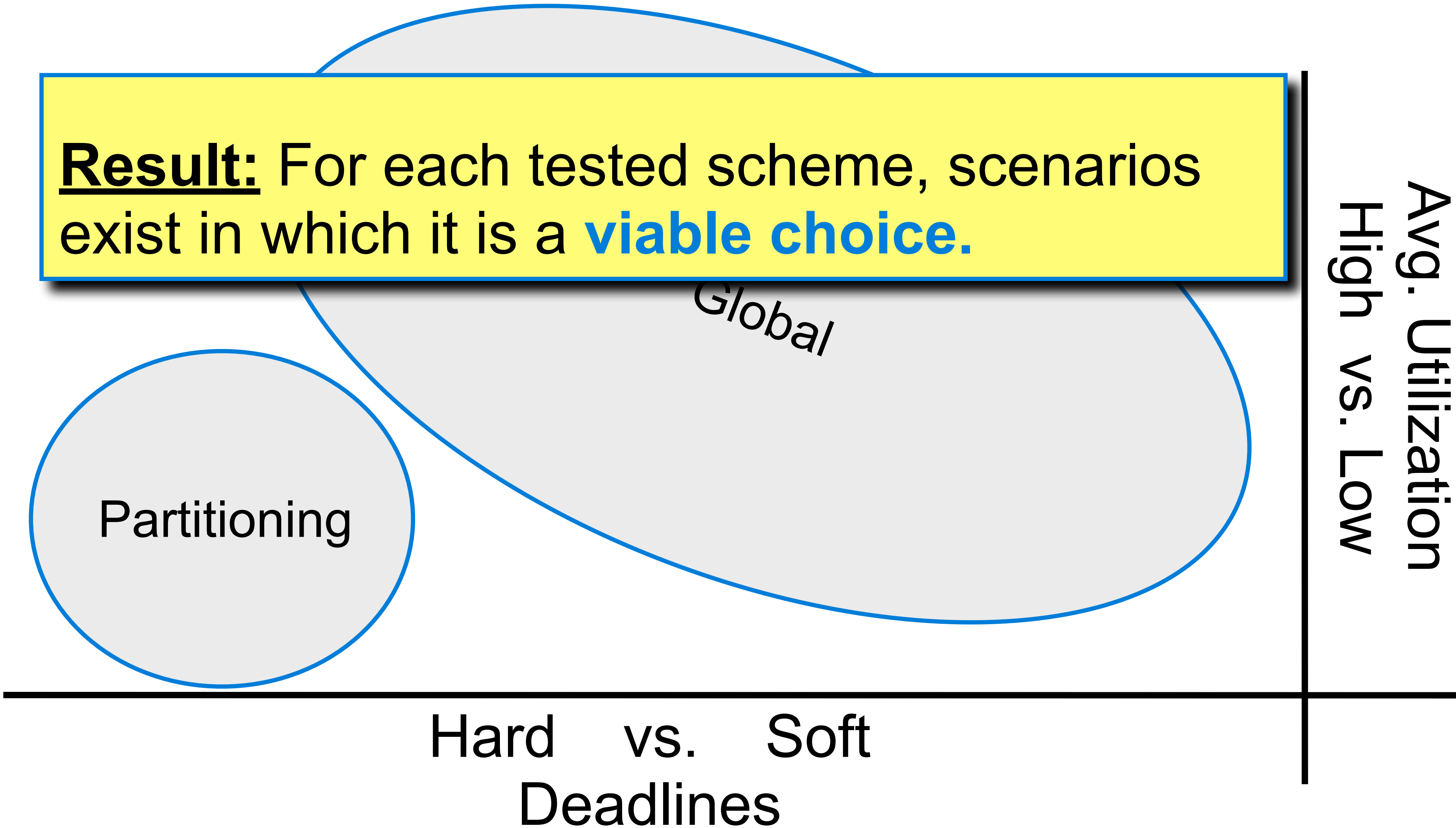
When (if ever) should you use partitioning (global)?



Evaluation of Scheduling Algorithms

When (if ever) should you use partitioning (global)?

Result: For each tested scheme, scenarios exist in which it is a **viable choice.**

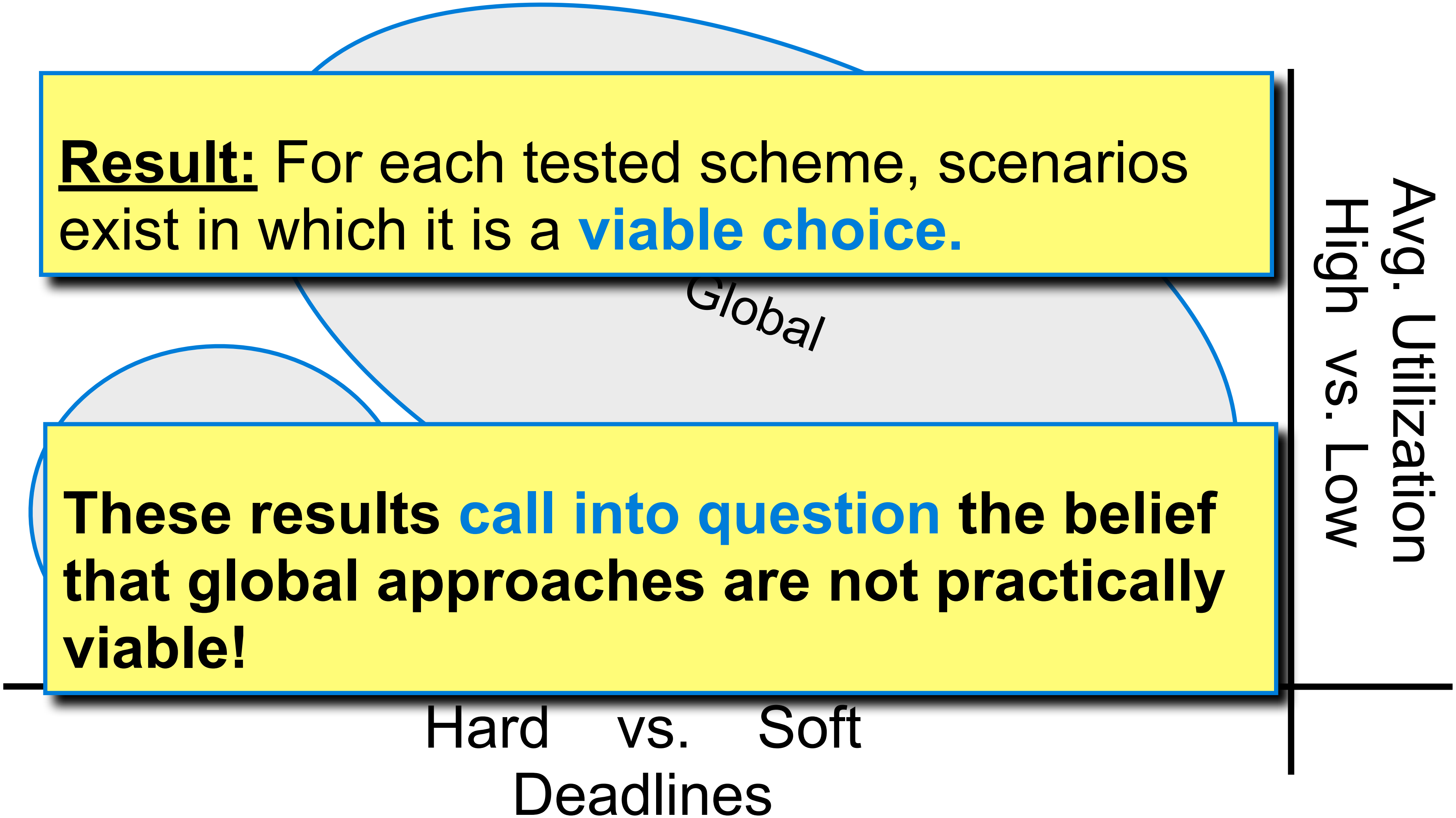


Evaluation of Scheduling Algorithms

When (if ever) should you use partitioning (global)?

Result: For each tested scheme, scenarios exist in which it is a **viable choice.**

These results call into question the belief that global approaches are not practically viable!



Some Results

Some Results

Slack scheduling can **improve the response time** of best-effort jobs significantly:

B. Brandenburg and J. Anderson, " **Integrating Hard/Soft Real-Time Tasks and Best-Effort Jobs on Multiprocessors** ", *Proc. of the 19th Euromicro Conference on Real-Time Systems*, pp. 61-70, July 2007.

Some Results

Slack scheduling can **improve the response time** of best-effort jobs significantly:

B. Brandenburg and J. Anderson, " **Integrating Hard/Soft Real-Time Tasks and Best-Effort Jobs on Multiprocessors** ", *Proc. of the 19th Euromicro Conference on Real-Time Systems*, pp. 61-70, July 2007.

A **flexible locking protocol** for EDF-scheduled multiprocessors:

A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, " A Flexible Real-Time Locking Protocol for Multiprocessors ", *Proc. of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 47-57, August 2007.

Some Results

Slack scheduling can **improve the response time** of best-effort jobs significantly:

B. Brandenburg and J. Anderson, " **Integrating Hard/Soft Real-Time Tasks and Best-Effort Jobs on Multiprocessors** ", *Proc. of the 19th Euromicro Conference on Real-Time Systems*, pp. 61-70, July 2007.

A **flexible locking protocol** for EDF-scheduled multiprocessors:

A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, " A Flexible Real-Time Locking Protocol for Multiprocessors ", *Proc. of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 47-57, August 2007.

Semaphores considered **harmful**:

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? ", *Proc. of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 342-353, April 2008.

The Next Steps

The Next Steps

Port to Linux 2.6.27.

The Next Steps

Port to Linux 2.6.27.

Port to ARM11 MPCore.

The Next Steps

Port to Linux 2.6.27.

Port to ARM11 MPCore.

Polish, fix bugs, **improve performance**…

LITMUS^{RT} – Features

Linux Testbed for Multiprocessor Scheduling in Real-Time systems

LITMUS^{RT} – Features

***Li**nux **T**estbed for **Mu**ltiprocessor **S**cheduling in **R**eal-**T**ime systems*

Many real-time plugins included.

(validate experiments, test userspace schemes, obtain overheads for your platform)

LITMUS^{RT} – Features

Linux Testbed for Multiprocessor Scheduling in Real-Time systems

Many real-time plugins included.

(validate experiments, test userspace schemes, obtain overheads for your platform)

Real-time policy can be switched at run-time.

(don't reboot during experiments)

LITMUS^{RT} – Features

Linux Testbed for Multiprocessor Scheduling in Real-Time systems

Many real-time plugins included.

(validate experiments, test userspace schemes, obtain overheads for your platform)

Real-time policy can be switched at run-time.

(don't reboot during experiments)

Writing plugins is easy.

(get your idea implemented quickly, you don't need to understand the whole kernel)

LITMUS^{RT} – Features

Linux Testbed for Multiprocessor Scheduling in Real-Time systems

Many real-time plugins included.

(validate experiments, test userspace schemes, obtain overheads for your platform)

Real-time policy can be switched at run-time.

(don't reboot during experiments)

Writing plugins is easy.

(get your idea implemented quickly, you don't need to understand the whole kernel)

Runs on x86-32, sparc64.

(x86-64 in the works, there is almost no platform dependent code, a research in Singapore ported it to ARM)

LITMUS^{RT} – Features

Linux Testbed for Multiprocessor Scheduling in Real-Time systems

Many real-time plugins included.

(validate experiments, test userspace schemes, obtain overheads for your platform)

Real-time policy can be switched at run-time.

(don't reboot during experiments)

Writing plugins is easy.

(get your idea implemented quickly, you don't need to understand the whole kernel)

Runs on x86-32, sparc64.

(x86-64 in the works, there is almost no platform dependent code, a research in Singapore ported it to ARM)

Real-time synchronization.

(working code for np-Q-locks, PCP, SRP, D-PCP, M-PCP, FMLP)

LITMUS^{RT} – Features

***Li**nux **T**estbed for **Mu**ltiprocessor **S**cheduling in **R**eal-**T**ime systems*

Many real-time plugins included.

(validate experiments, test userspace schemes, obtain overheads for your platform)

Real-time policy can be switched at run-time.

(don't reboot during experiments)

Writing plugins is easy.

(get your idea implemented quickly, you don't need to understand the whole kernel)

Runs on x86-32, sparc64.

(x86-64 in the works, there is almost no platform dependent code, a research in Singapore ported it to ARM)

Real-time synchronization.

(working code for np-Q-locks, PCP, SRP, D-PCP, M-PCP, FMLP)

It's just Linux.

(all your existing scripts still work, your real-time tasks can do everything a normal task can do)