The SRP Resource Sharing Protocol for Self-Suspending Tasks

Geoffrey Nelissen and Alessandro Biondi

CISTER – Research Centre in Real-Time & Embedded Computing Systems

Scuola Superiore Sant'Anna di Studi Universitari e di Perfezionamento

The SRP Resource Sharing Protocol for Self-Suspending Tasks

Geoffrey Nelissen and Alessandro Biondi

CISTER – Research Centre in Real-Time & Embedded Computing Systems

Scuola Superiore Sant'Anna di Studi Universitari e di Perfezionamento

The SRP Resource Sharing Protocol for Self-Suspending Tasks (on single core platforms)

Geoffrey Nelissen and Alessandro Biondi

CISTER – Research Centre in
Real-Time & Embedded Computing Systems

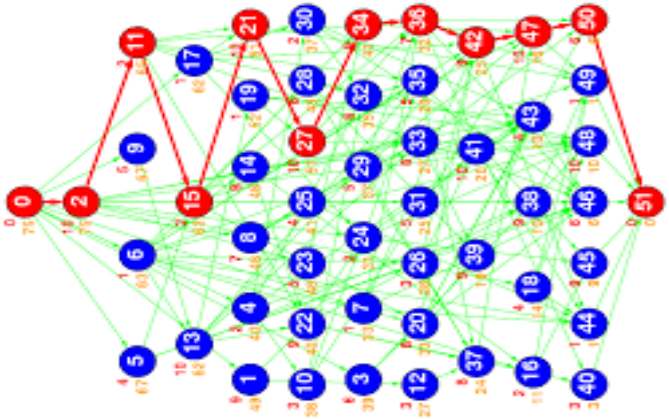
Scuola Superiore
Sant'Anna
di Studi Universitari e di Perfezionamento

Motivation

Evolution of the state of practice

Motivation

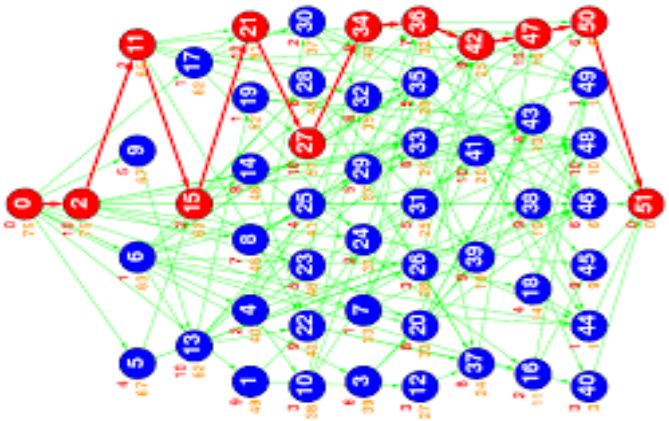
Evolution of the state of practice



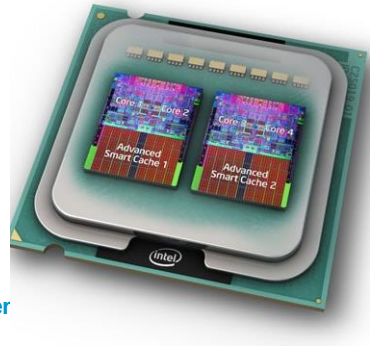
Parallel applications

Motivation

Evolution of the state of practice



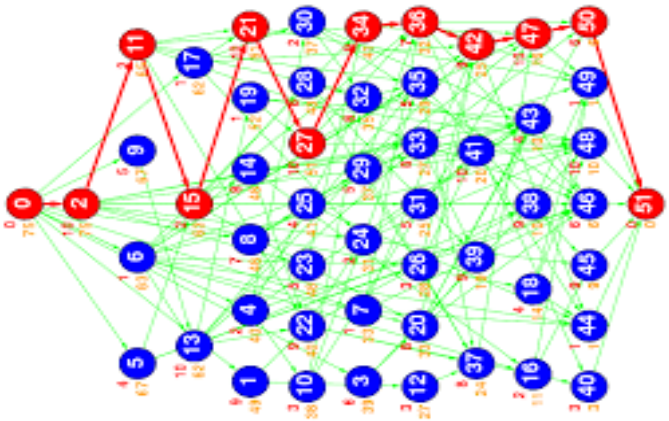
Parallel applications



Multicore processors

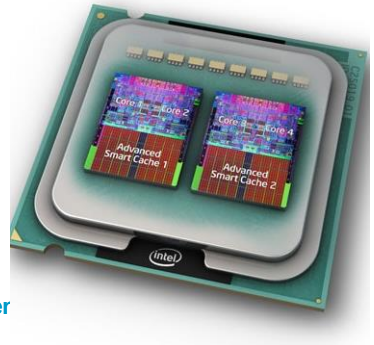
Motivation

Evolution of the state of practice



Parallel applications

Partitioned FP scheduling



Multicore processors

Motivation

Response Time Analysis of Sporadic DAG Tasks under Partitioned Scheduling

*José Fonseca, **Geoffrey Nelissen**, Vincent Nélis and Luis Miguel Pinho*

Parallel applications partitioned on multicore platforms can be modelled as sets of self-suspending tasks running on single core processors

Motivation

Response Time Analysis of Sporadic DAG Tasks under Partitioned Scheduling

*José Fonseca, **Geoffrey Nelissen**, Vincent Nélis and Luis Miguel Pinho*

Parallel applications partitioned on multicore platforms can be modelled as sets of self-suspending tasks running on single core processors

Partitioned Fixed-Priority Scheduling of Parallel Tasks Without Preemptions

***Daniel Casini**, Alessandro Biondi, Geoffrey Nelissen and Giorgio Buttazzo*

*This afternoon @ **RTSS***

Motivation

Parallel applications partitioned on multicore platforms can be modelled as sets of self-suspending tasks running on single core processors

Most applications will share resources through locks

Motivation

Parallel applications partitioned on multicore platforms can be modelled as sets of self-suspending tasks running on single core processors

Most applications will share resources through locks

Standards such as AUTOSAR mandates the use of the SRP for sharing resources

The Stack Resource Policy (SRP)

The Stack Resource Policy (SRP)

Task 1

Task 2

Shared
resource 1

The Stack Resource Policy (SRP)

Task 1 Prio=2

Task 2 Prio=1

Shared
resource 1

The Stack Resource Policy (SRP)

Task 1 Prio=2

Task 2 Prio=1

Shared resource 1 Prio=2

priority of **highest priority** task **accessing** that resource

The Stack Resource Policy (SRP)

Task 1 Prio=2

Task 2 Prio=1

Shared resource 1 Prio=2

System ceiling

0

priority of the last resource locked and not unlocked

The Stack Resource Policy (SRP)

Task 1 Prio=2

Task 2 Prio=1

Shared resource 1 Prio=2

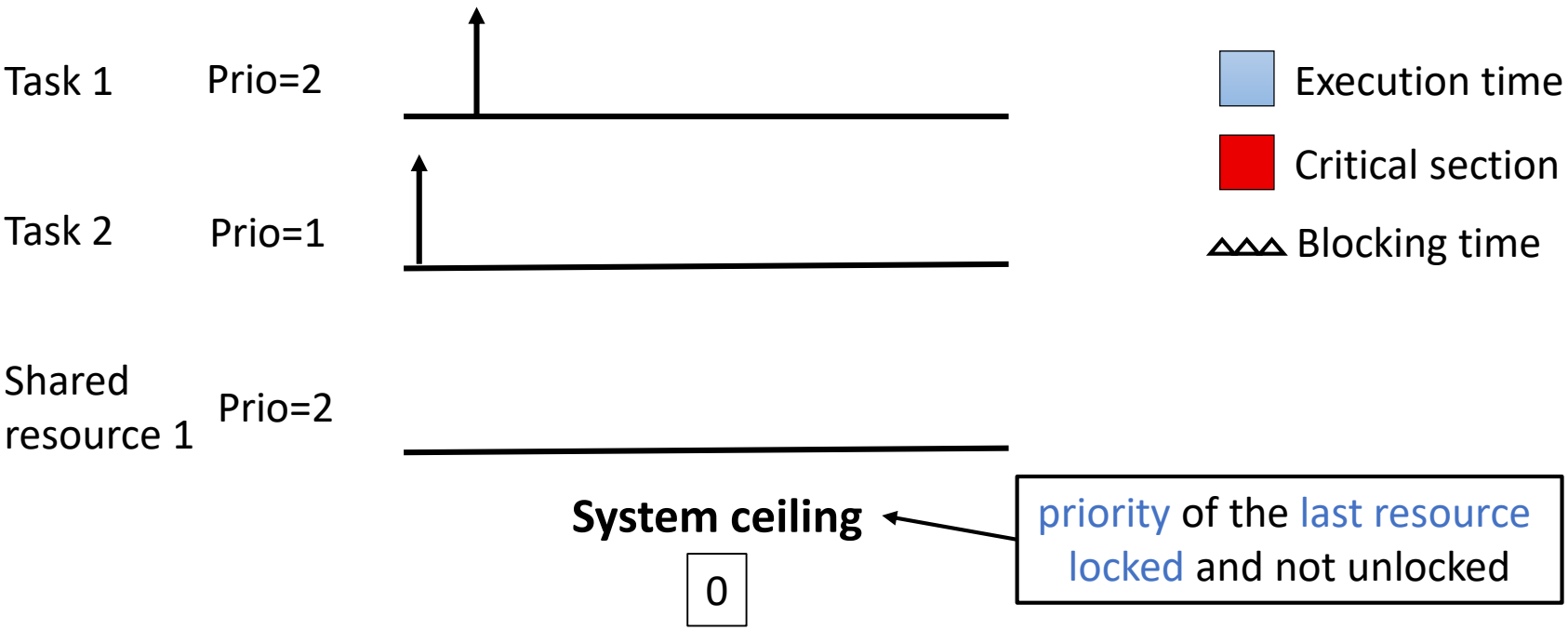
System ceiling

0

priority of the last resource locked and not unlocked

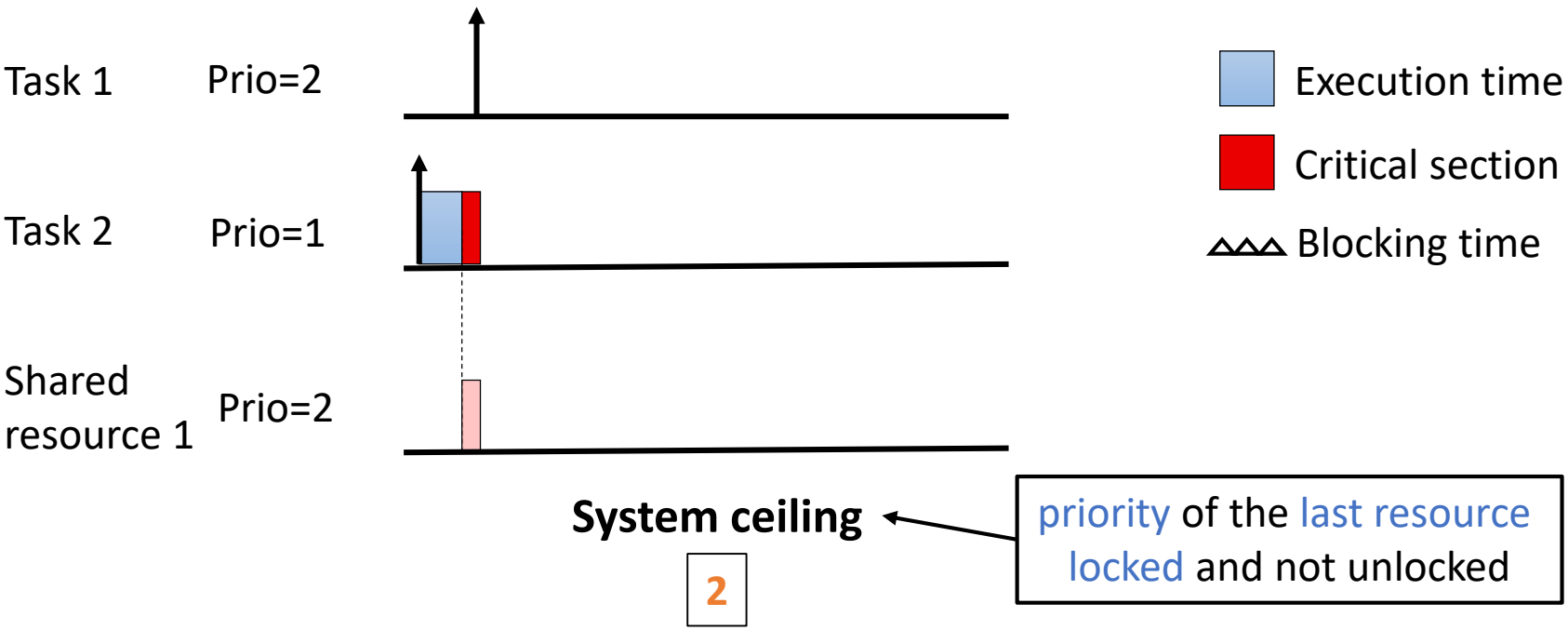
A task T1 can **preempt** another task T2 if
prio(T1) > prio(T2) **and** prio(T1) > system ceiling

The Stack Resource Policy (SRP)



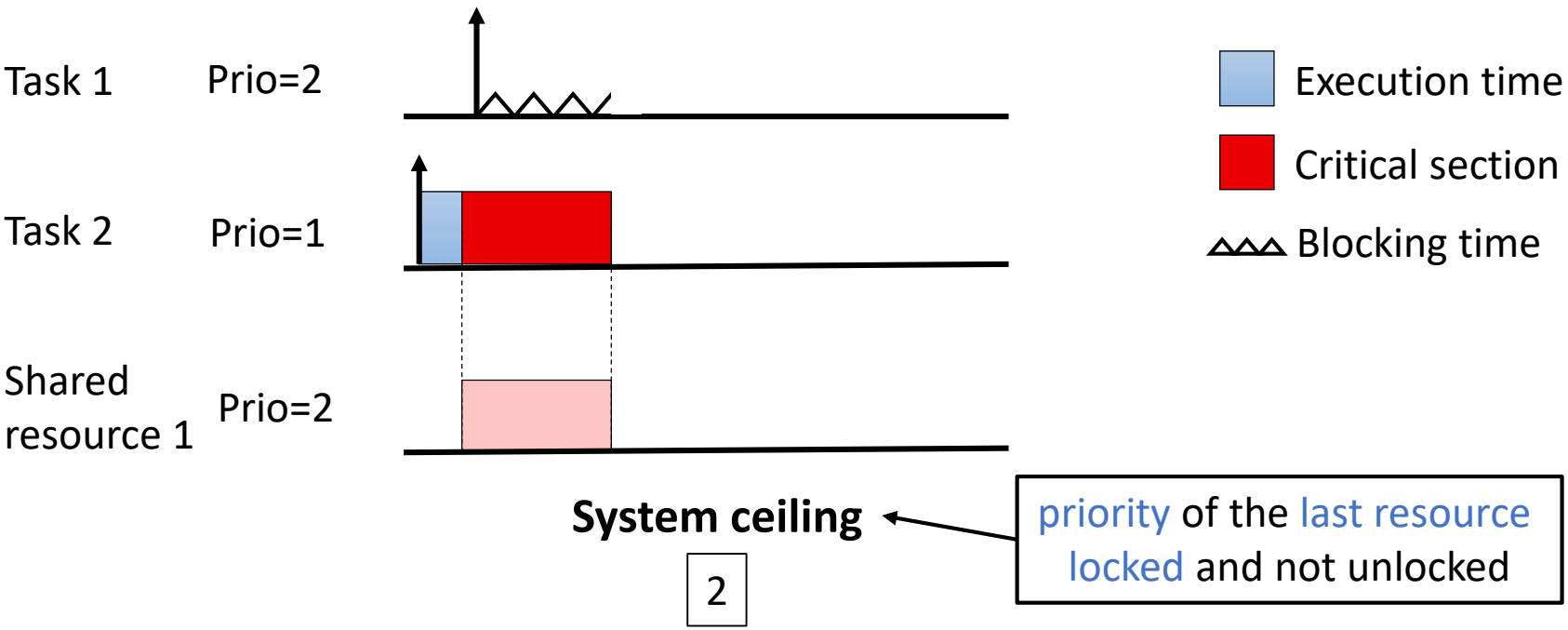
A task T1 can **preempt** another task T2 if
prio(T1) > prio(T2) **and** prio(T1) > system ceiling

The Stack Resource Policy (SRP)



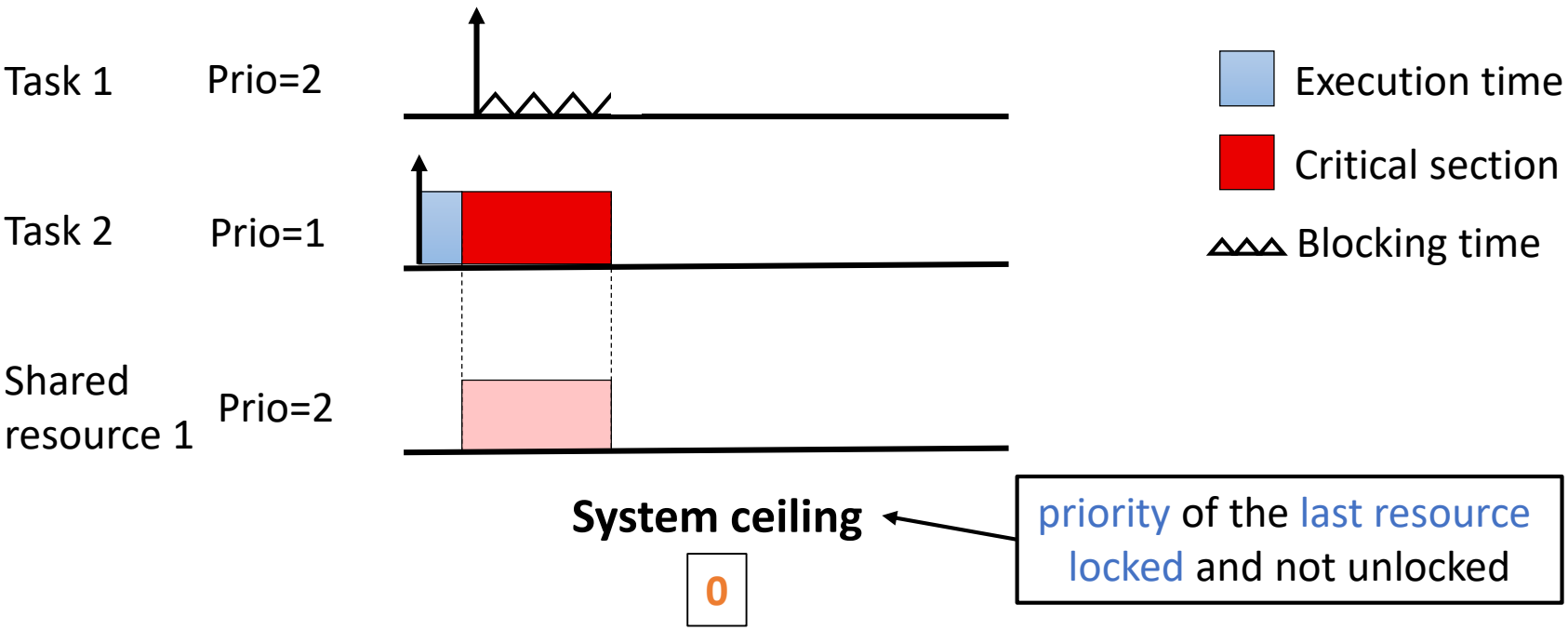
A task T1 can **preempt** another task T2 if
prio(T1) > prio(T2) **and** prio(T1) > system ceiling

The Stack Resource Policy (SRP)



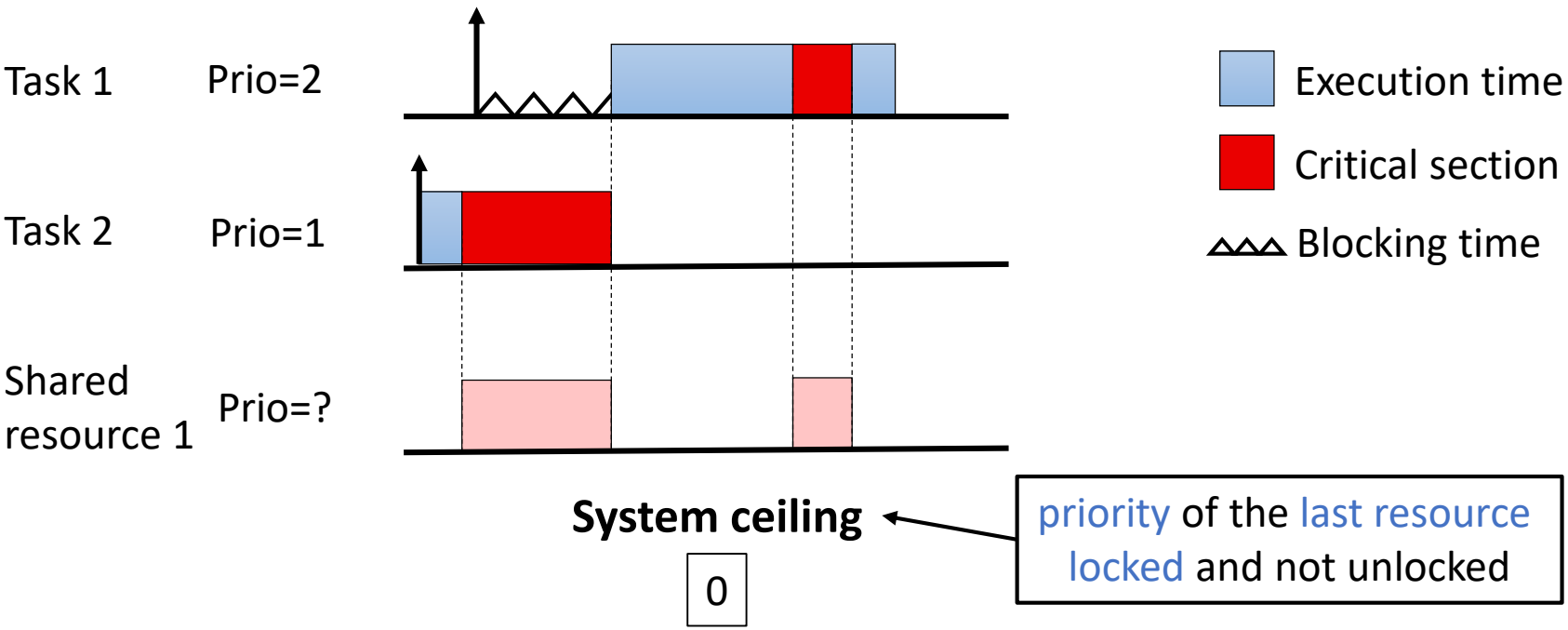
A task T1 can **preempt** another task T2 if
prio(T1) > prio(T2) **and** prio(T1) > system ceiling

The Stack Resource Policy (SRP)



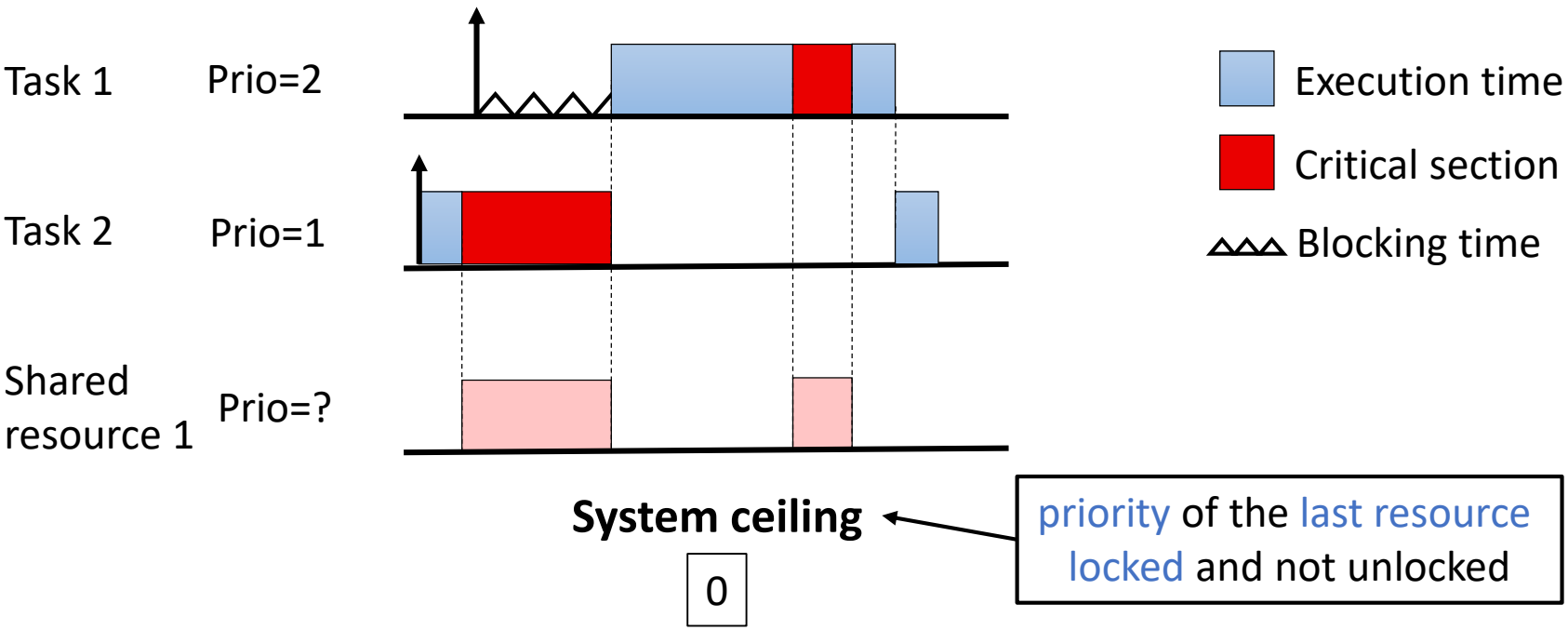
A task T1 can **preempt** another task T2 if
prio(T1) > prio(T2) **and** prio(T1) > system ceiling

The Stack Resource Policy (SRP)



A task T1 can **preempt** another task T2 if
prio(T1) > prio(T2) **and** prio(T1) > system ceiling

The Stack Resource Policy (SRP)



A task T1 can **preempt** another task T2 if
prio(T1) > prio(T2) **and** prio(T1) > system ceiling

WCRT analysis for the SRP

$$R_i = B_i + C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

$$B_i = \max_{\tau_j \in lp(i), \ell_k \in Q} \{L_{j,k} \mid \pi(\ell_k) \geq \pi_i\}_0$$

WCRT analysis for the SRP

Blocking time

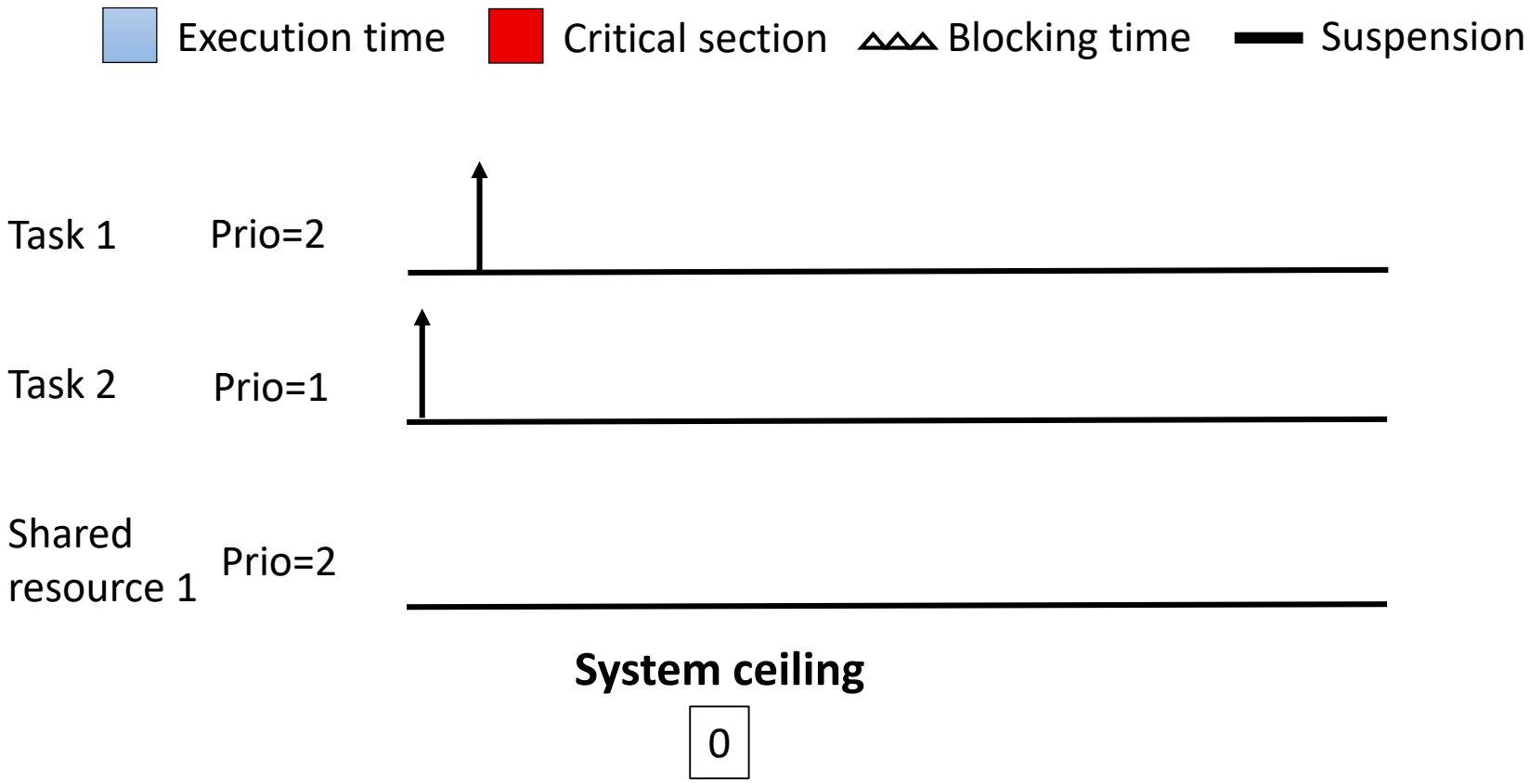
$$R_i = B_i + C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

$$B_i = \max_{\tau_j \in lp(i)} \max_{\ell_k \in Q} \{L_{j,k} \mid \pi(\ell_k) \geq \pi_i\}_0$$

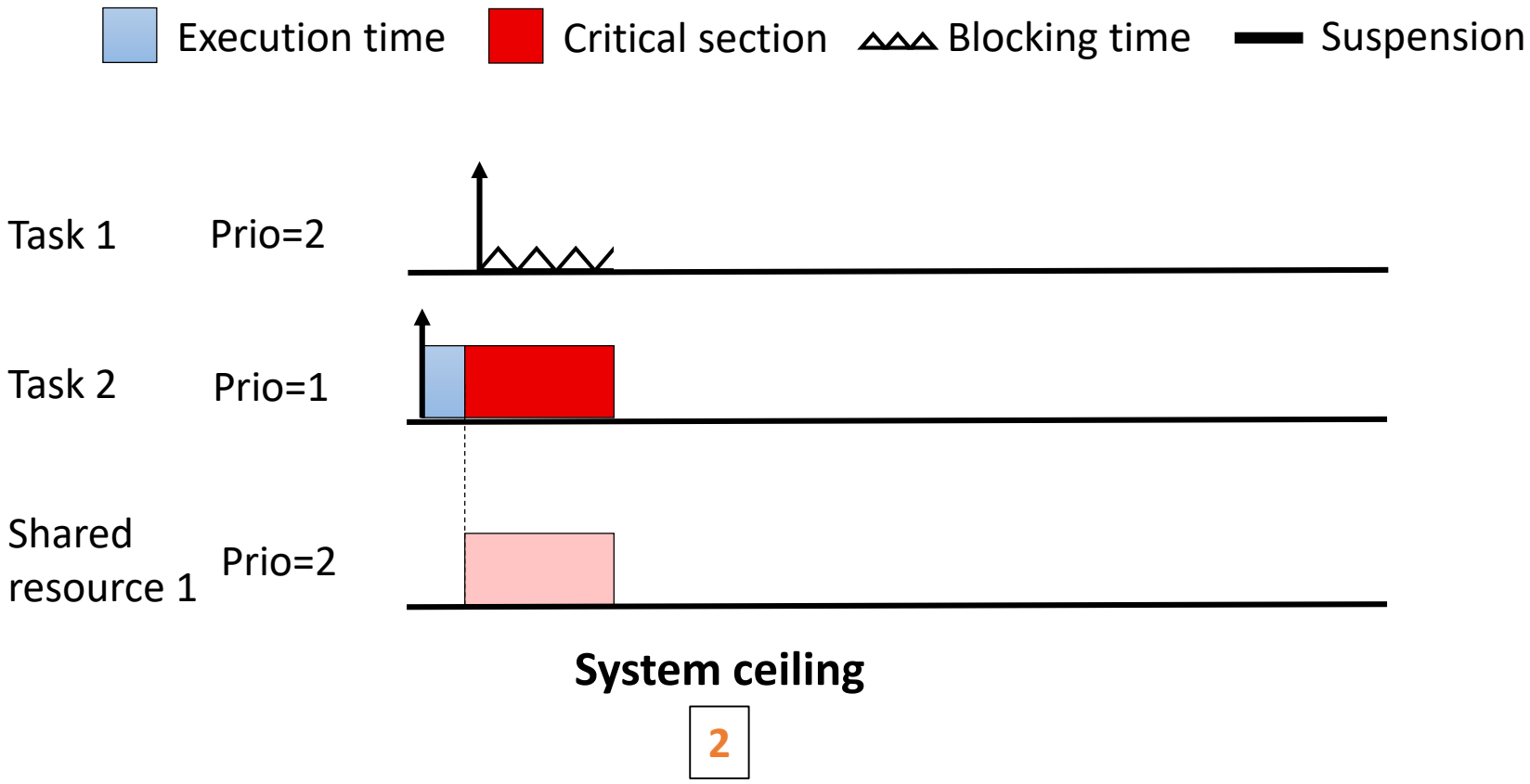
Critical section length	whose priority ≥ prio(task *i*)

SRP and self-suspending tasks

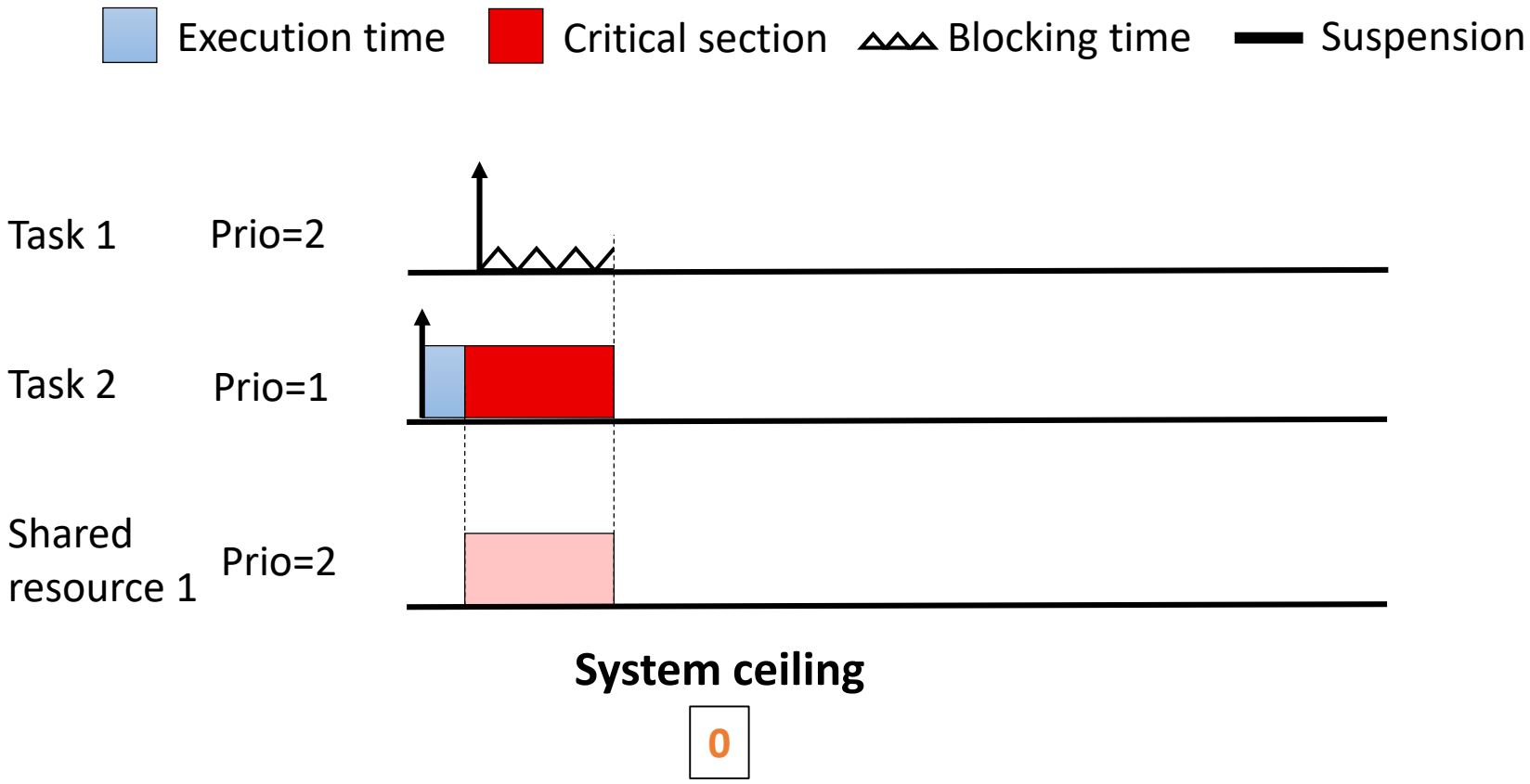
SRP and self-suspending tasks



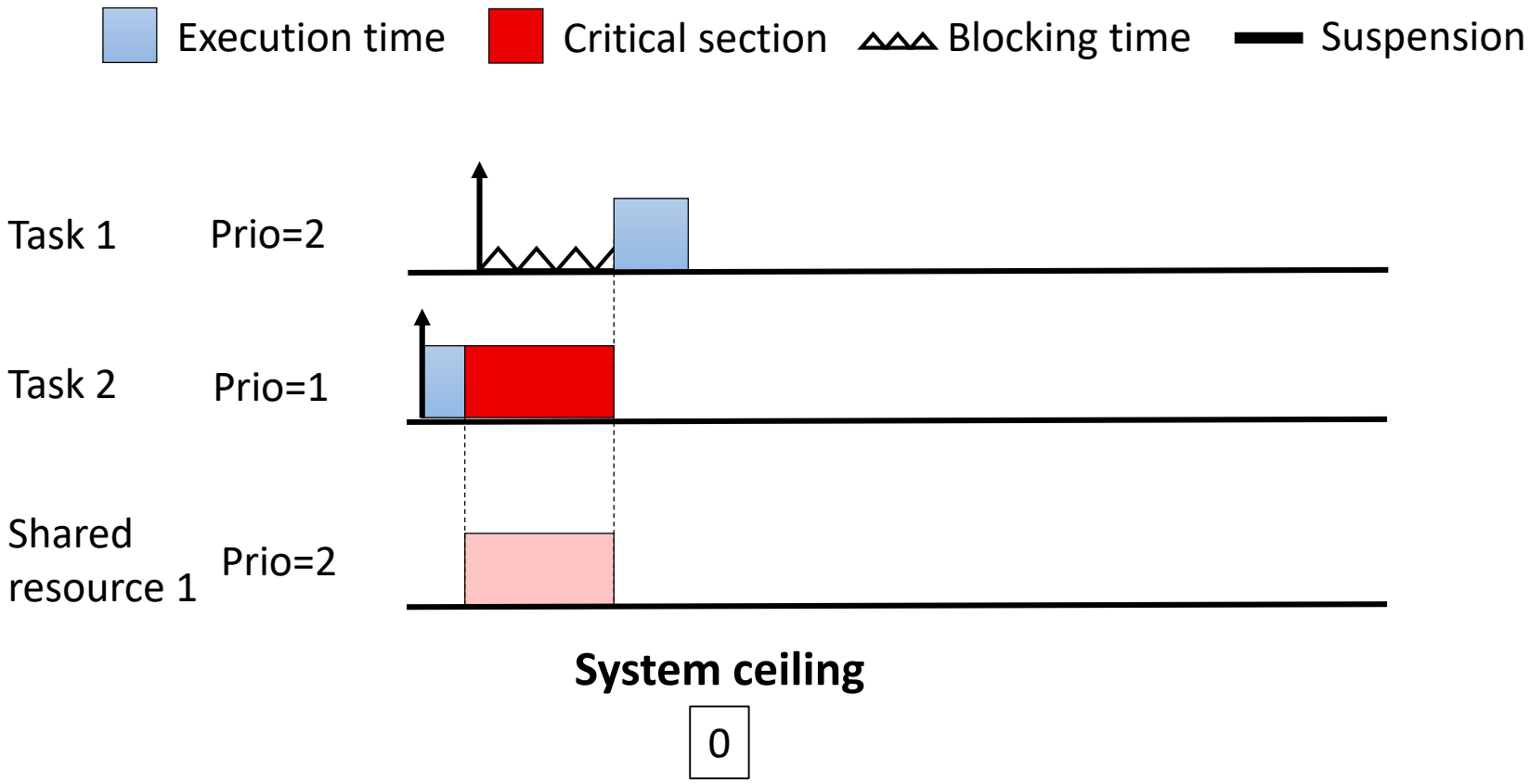
SRP and self-suspending tasks



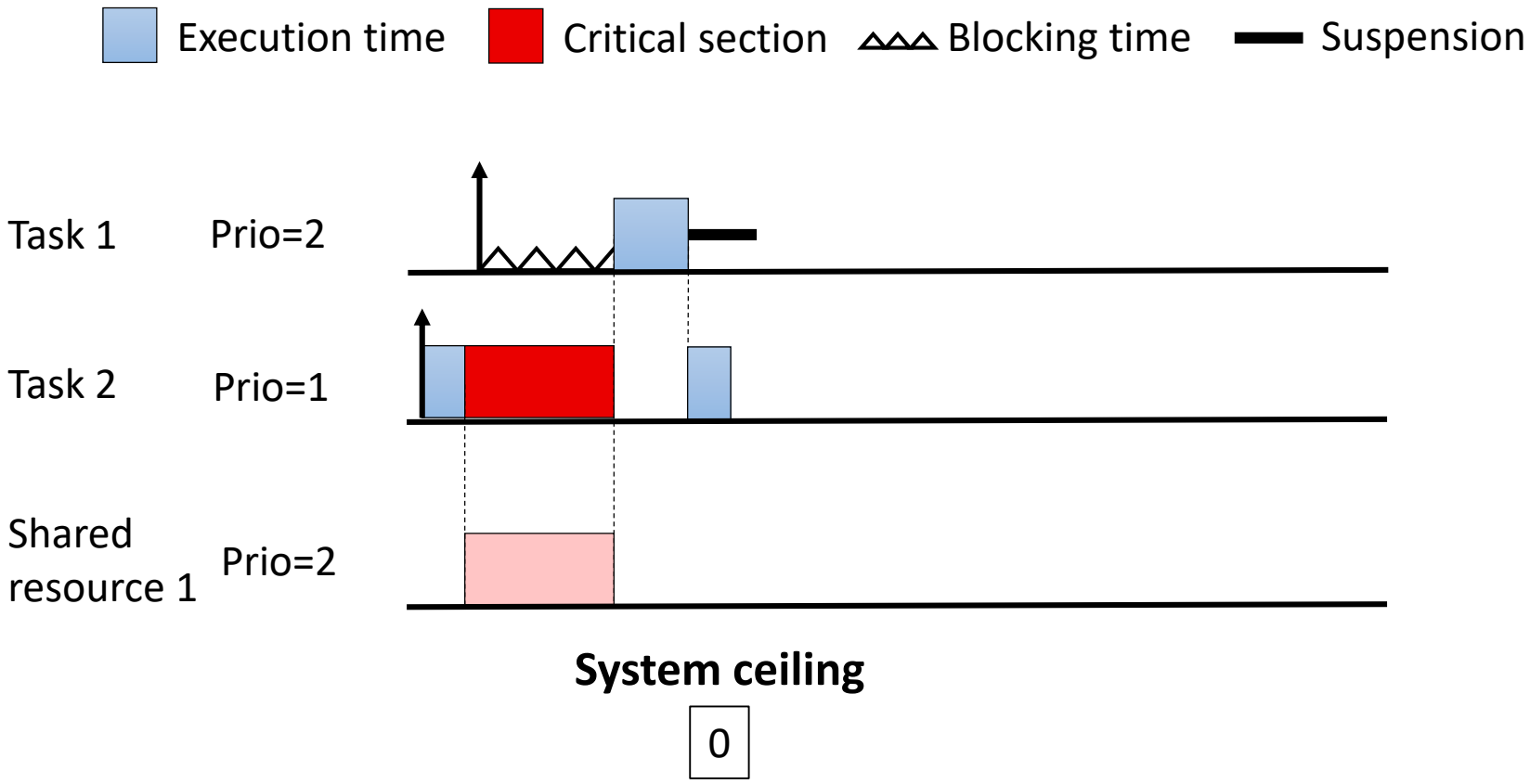
SRP and self-suspending tasks



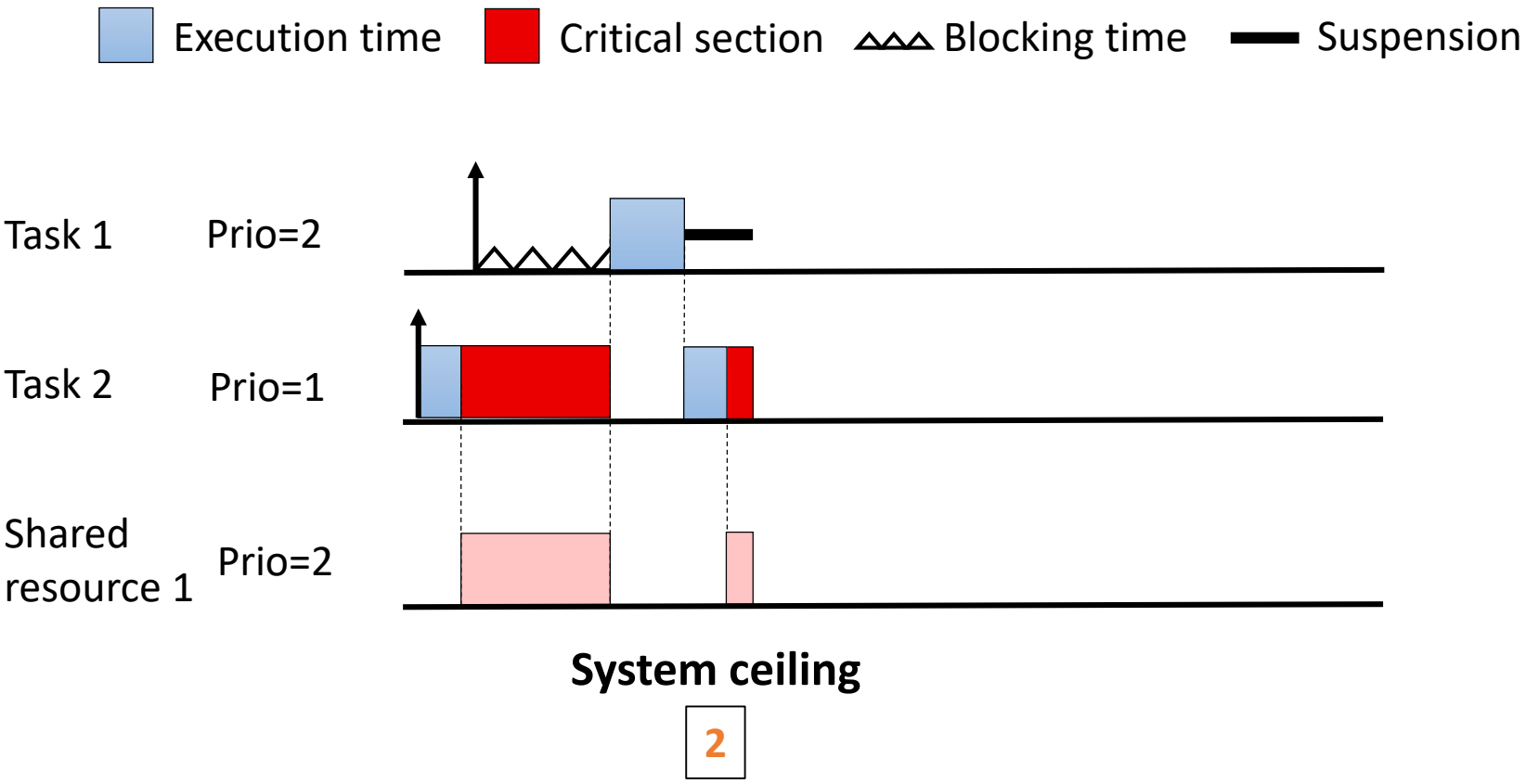
SRP and self-suspending tasks



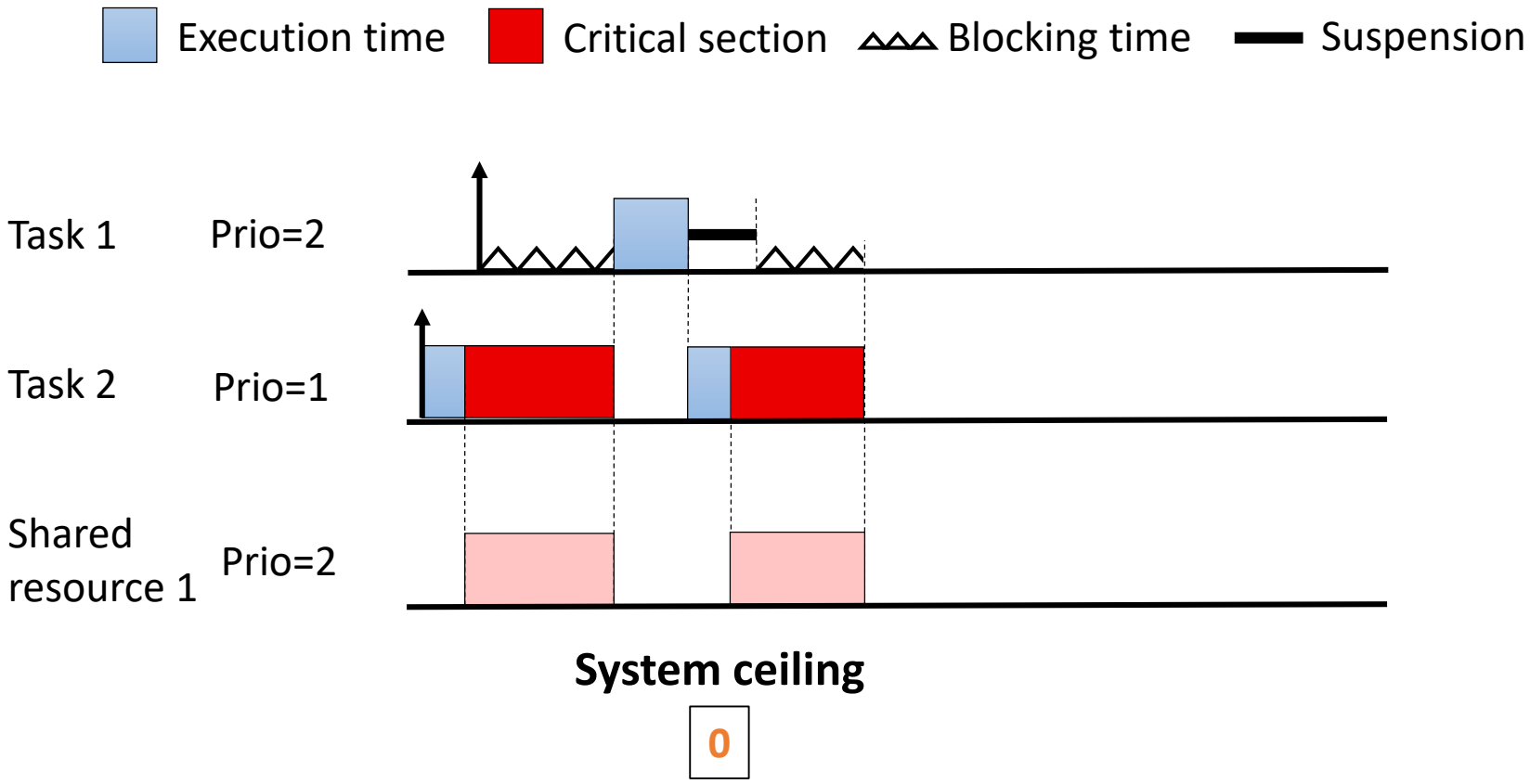
SRP and self-suspending tasks



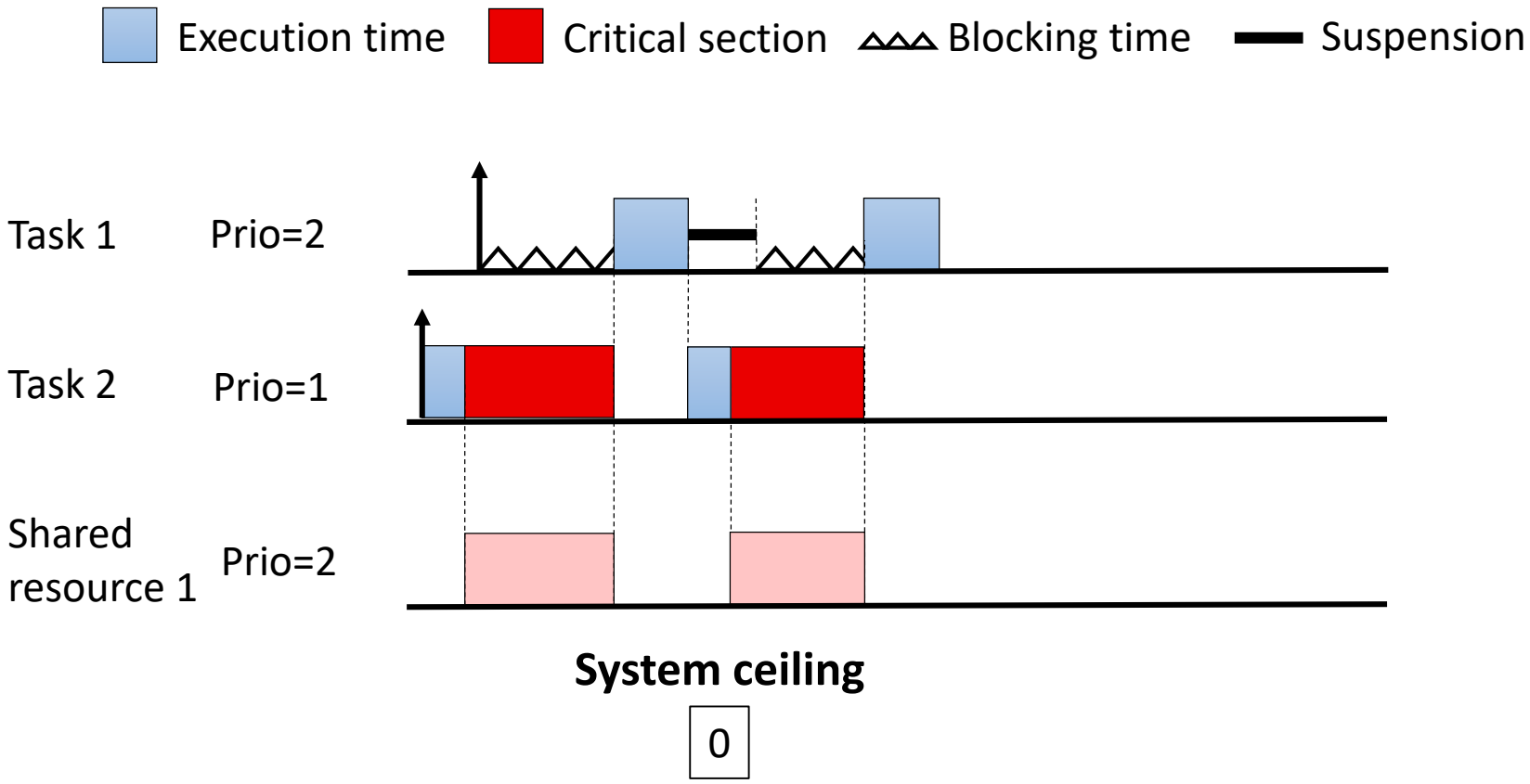
SRP and self-suspending tasks



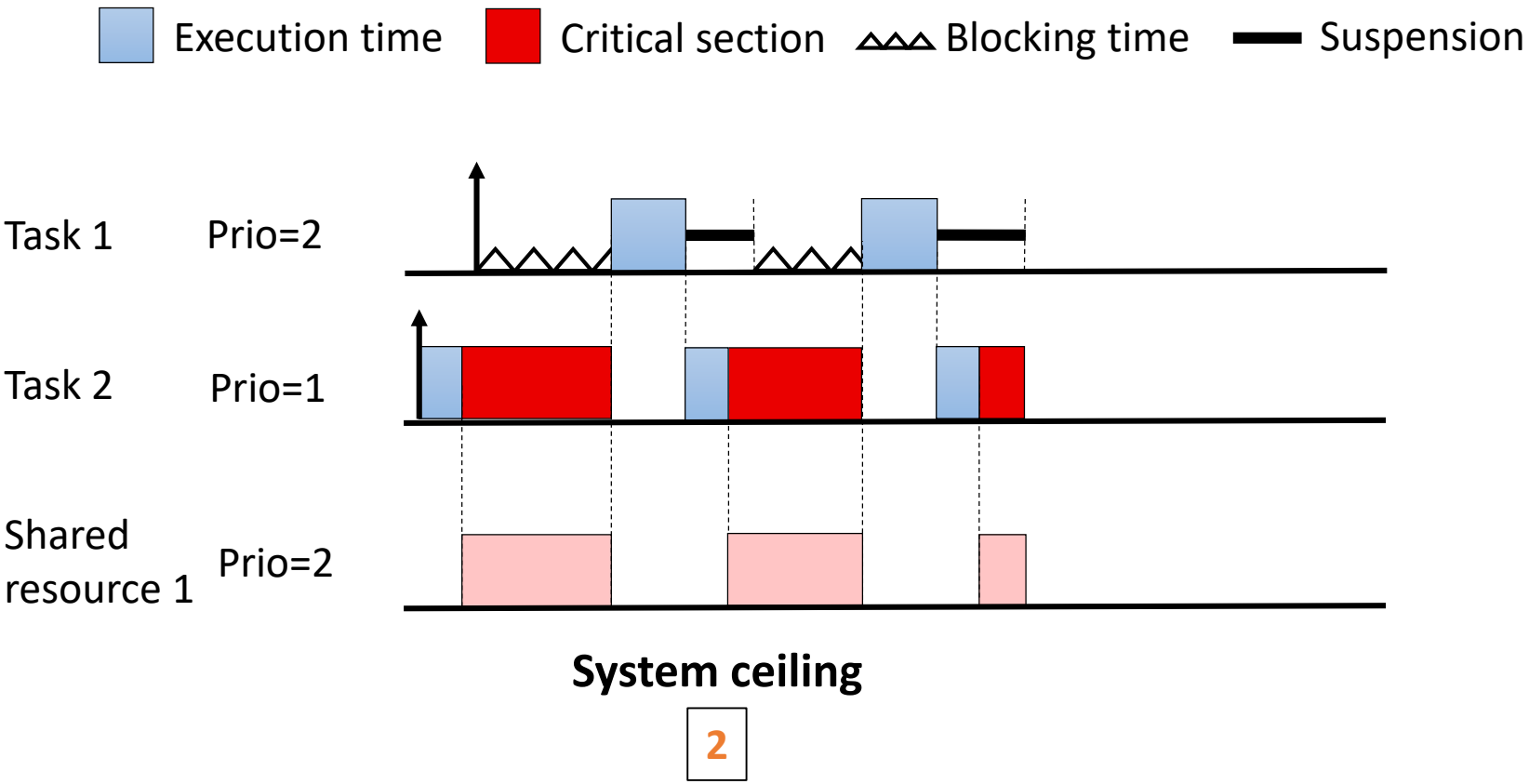
SRP and self-suspending tasks



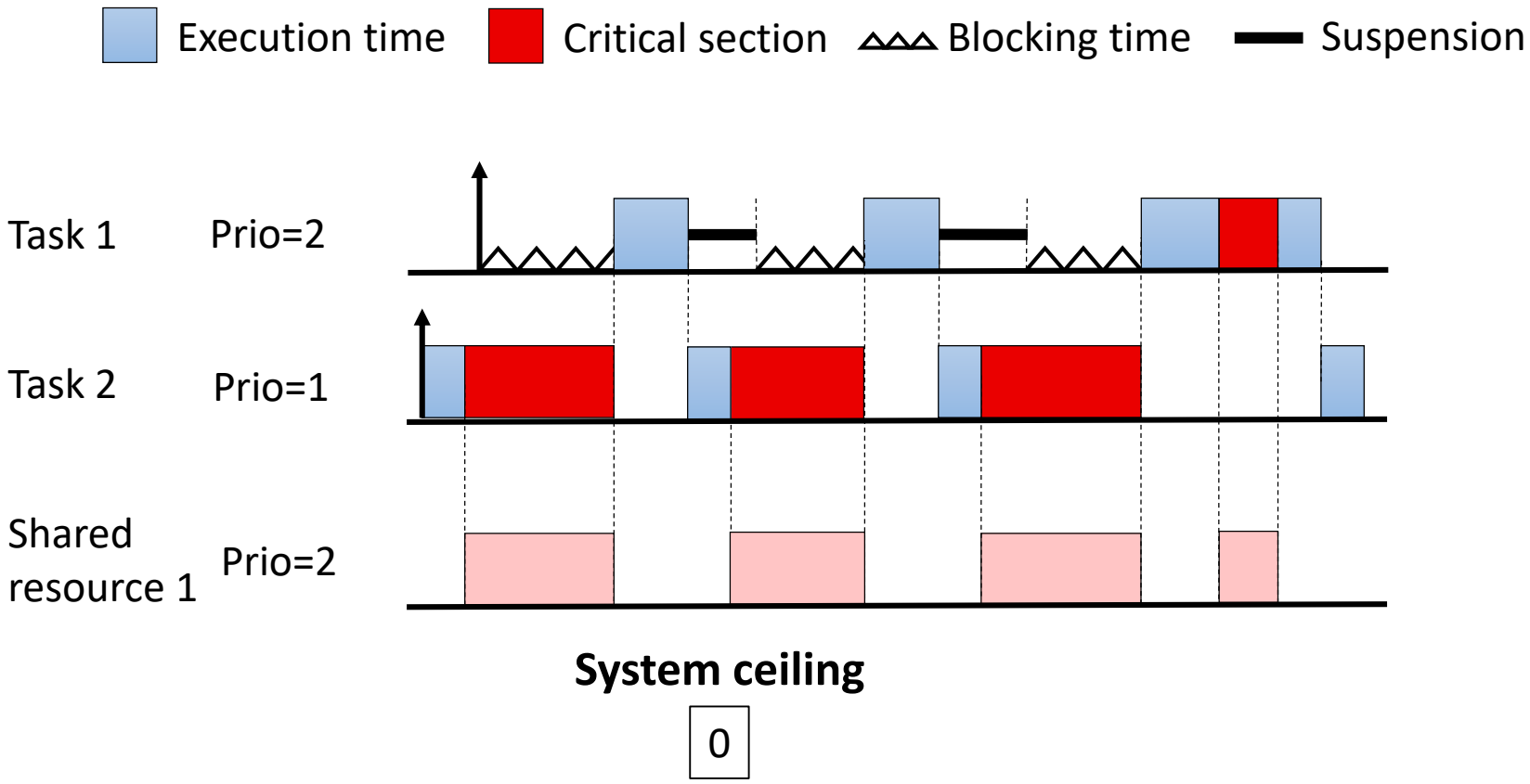
SRP and self-suspending tasks



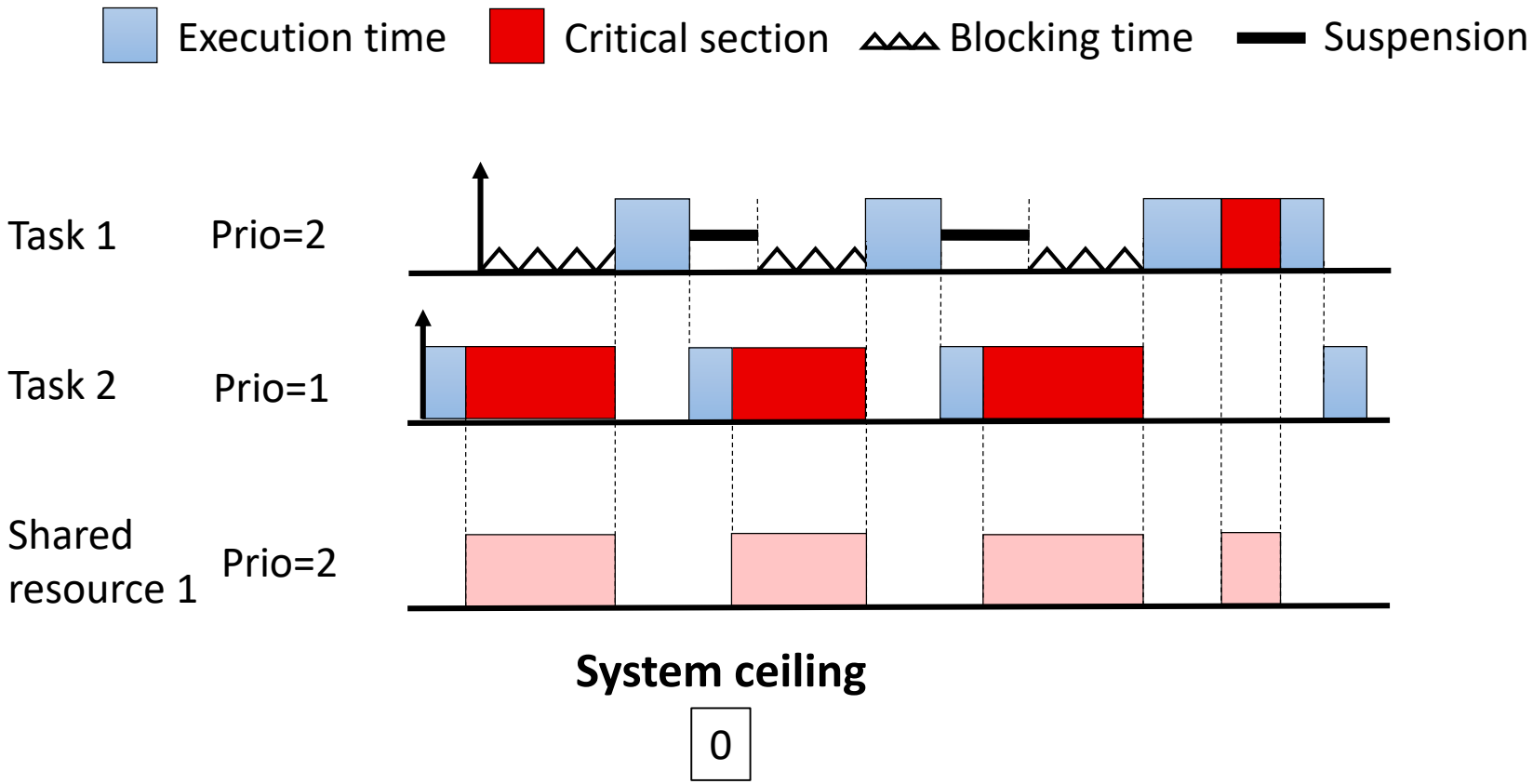
SRP and self-suspending tasks



SRP and self-suspending tasks



SRP and self-suspending tasks



A task may be **blocked** **as many times as it self-suspends**

SRP + self-suspending tasks Straightforward analysis

$$R_i = (C_i + S_i) + B_i(R_i, \overline{\mathbf{R}}) + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i + \overline{R}_j - C_j}{T_j} \right\rceil C_j$$

$$B_i(R_i, \overline{\mathbf{R}}) = (X_i + 1) \times \max_{\tau_j \in lp(i), \ell_k \in Q} \{L_{j,k} \mid \pi(\ell_k) \geq \pi_i\}$$

Maximum number of times task *i* self-suspends

SRP + self-suspending tasks Improved analysis

SRP + self-suspending tasks Improved analysis

- Identify **all** the critical sections that may block task *i* during its response time

SRP + self-suspending tasks Improved analysis

- Identify **all** the critical sections that may block task *i* during its response time
- Save their length in a multi-set $S_i(R_i)$

SRP + self-suspending tasks Improved analysis

- Identify **all** the critical sections that may block task *i* during its response time
- Save their length in a multi-set $S_i(R_i)$
- The blocking B_i is bounded by the sum of the $(X_i + 1)$ longest critical sections in $S_i(R_i)$

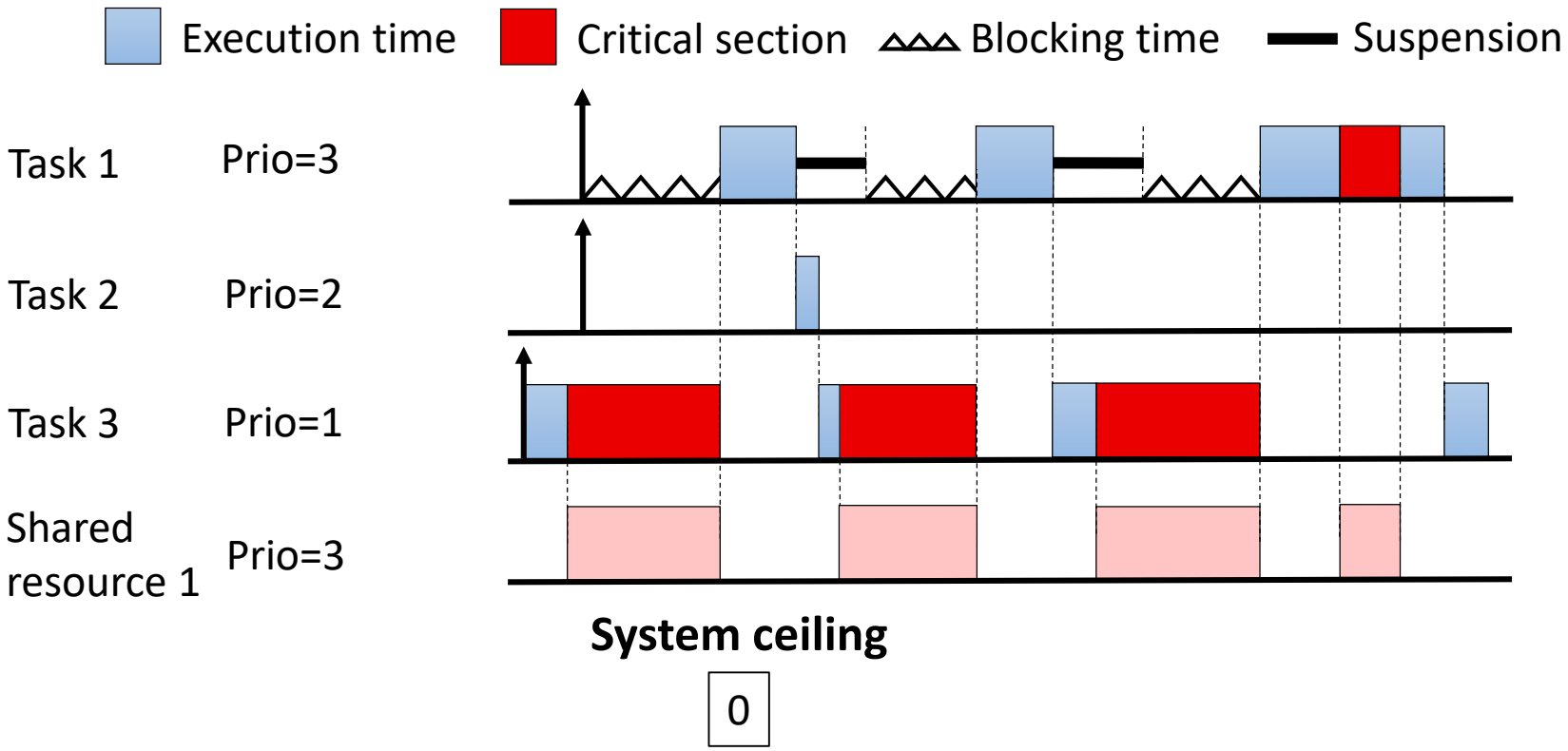
SRP + self-suspending tasks Improved analysis

- Identify **all** the critical sections that may block task *i* during its response time
- Save their length in a multi-set $S_i(R_i)$
- The blocking B_i is bounded by the sum of the $(X_i + 1)$ longest critical sections in $S_i(R_i)$

Note that the content of $S_i(R_i)$ depends on the worst-case response time of all tasks
➔ $S_i(R_i)$ must be iteratively computed

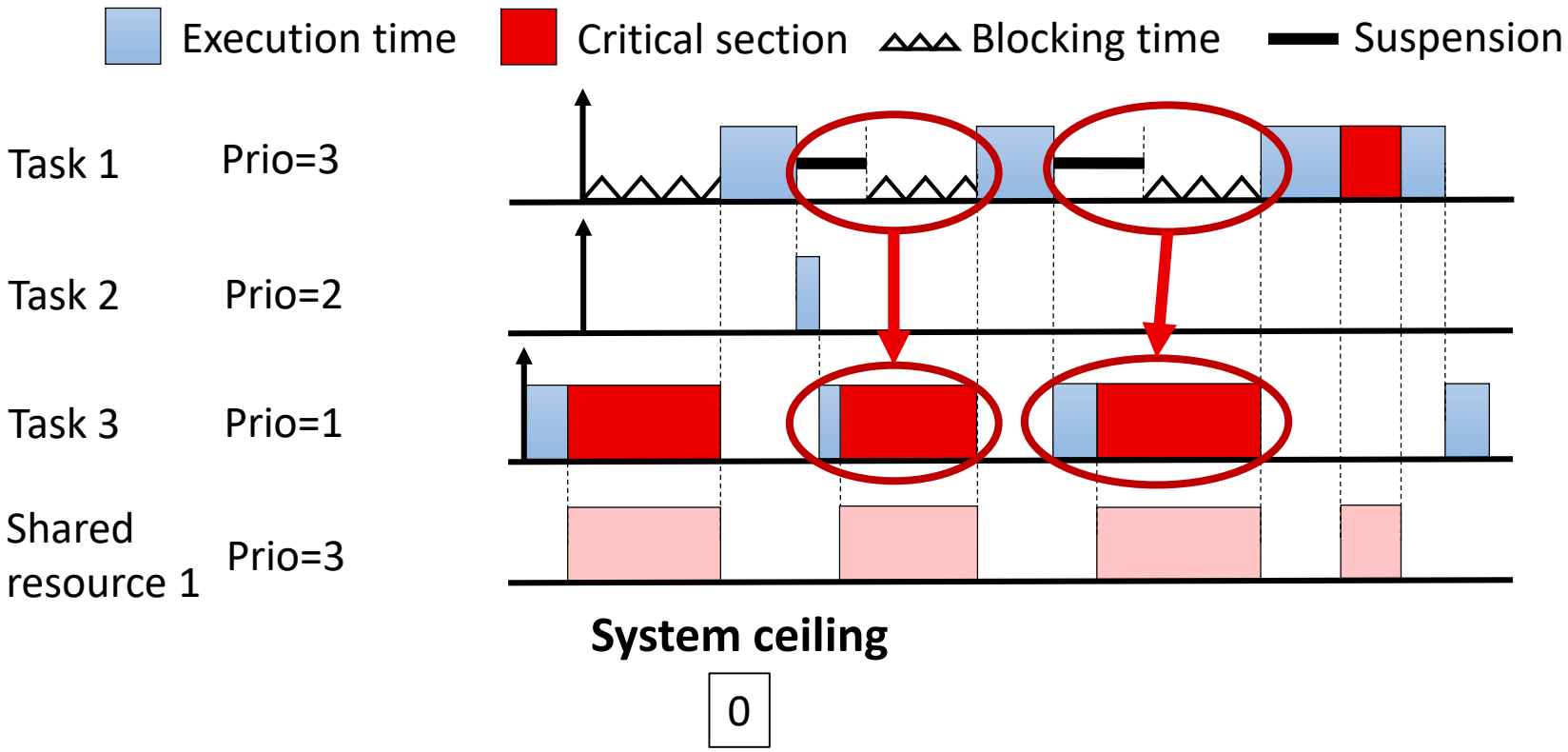
Improving the locking protocol

Improving the locking protocol



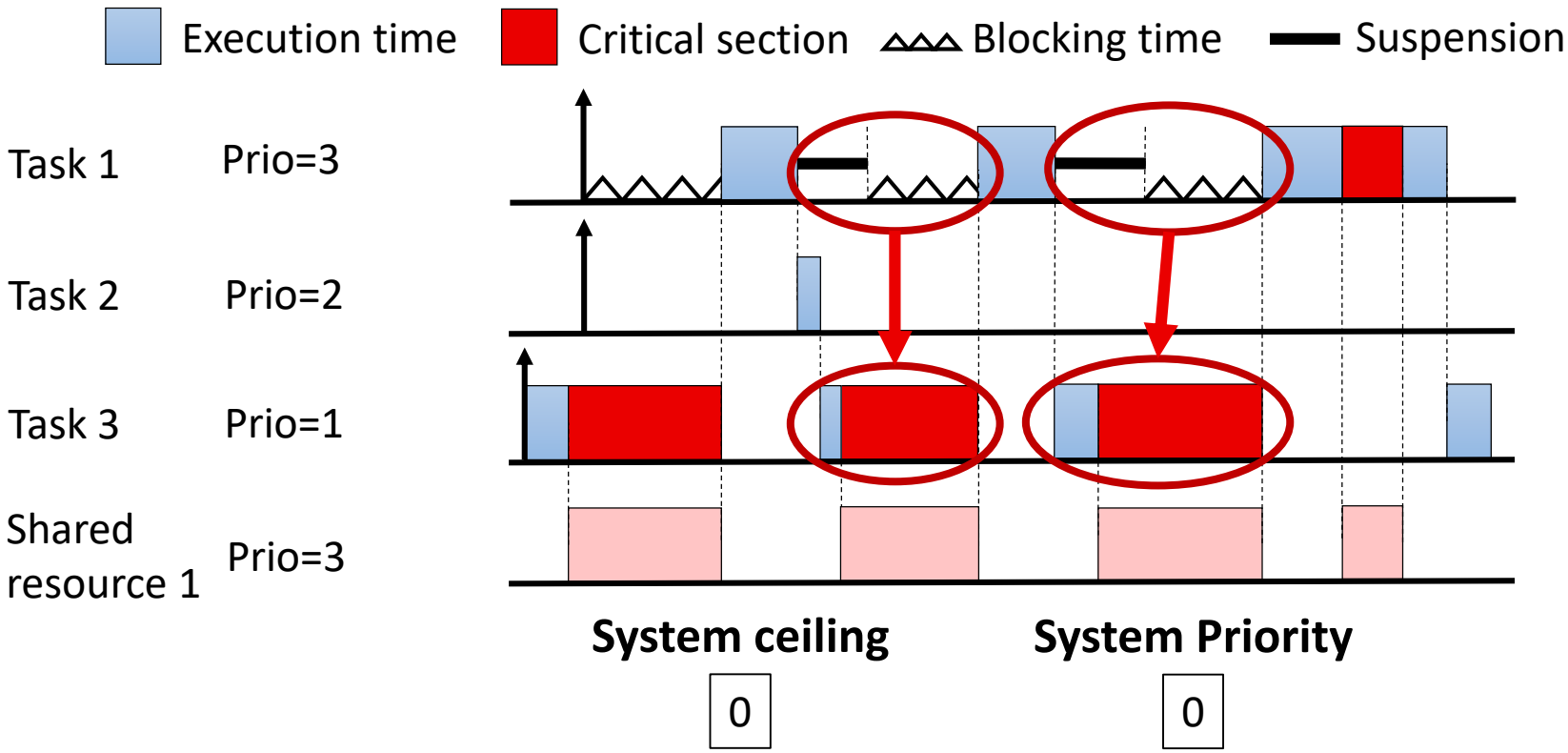
A task T1 can ***preempt*** another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

Improving the locking protocol



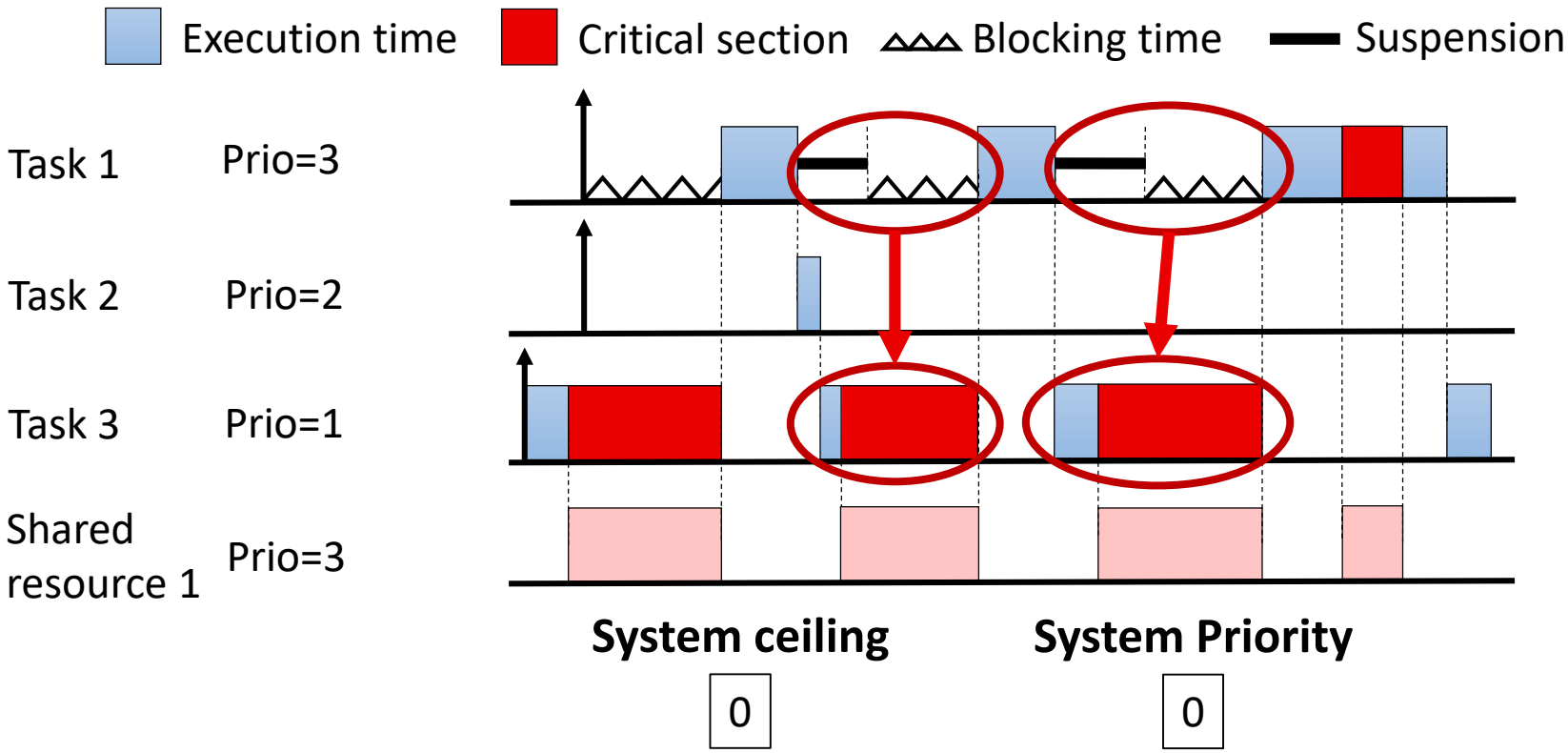
A task T1 can ***preempt*** another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

Improving the locking protocol



A task T1 can ***preempt*** another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

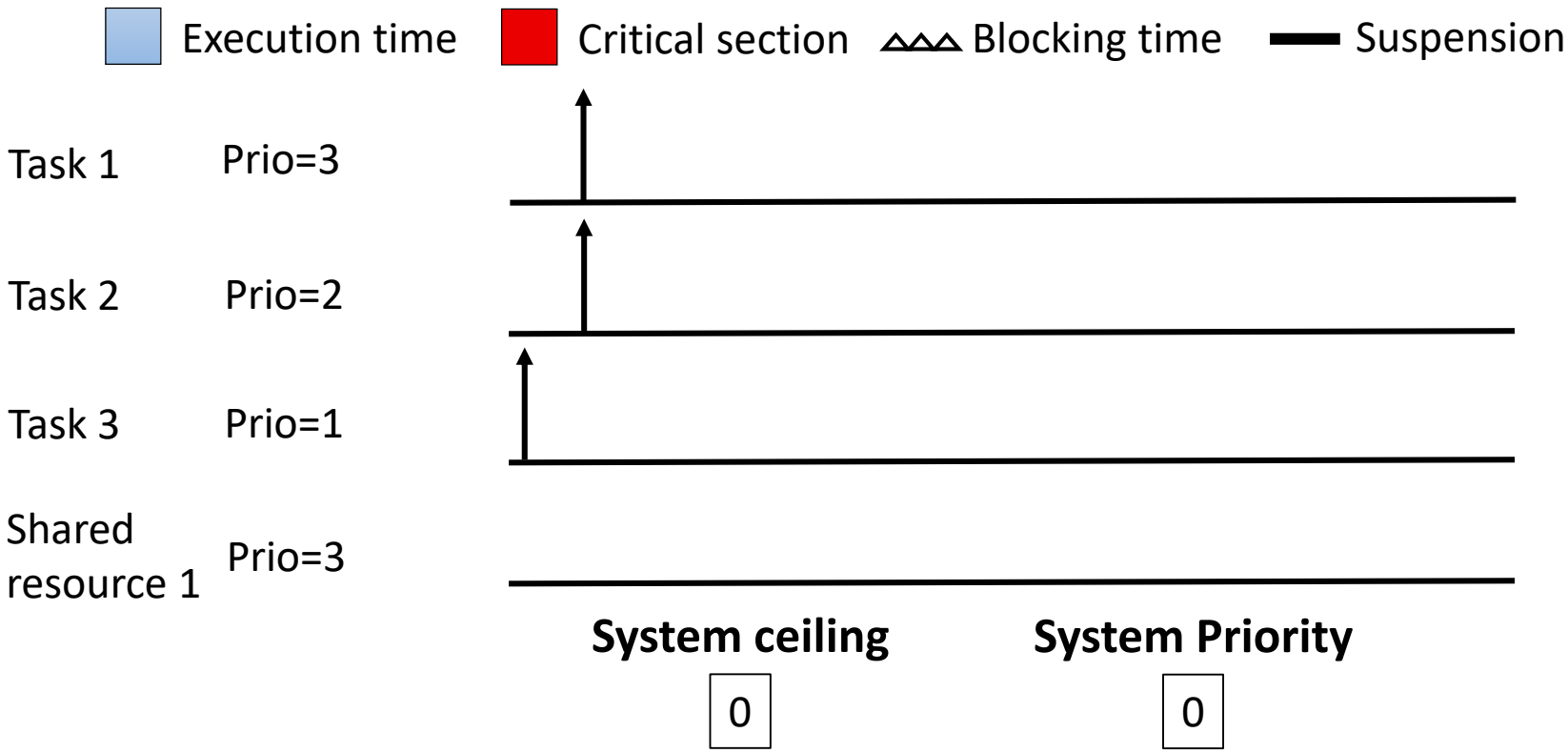
Improving the locking protocol



A task Ti is ***eligible*** to execute if
prio(Ti) > system priority

A task T1 can ***preempt*** another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

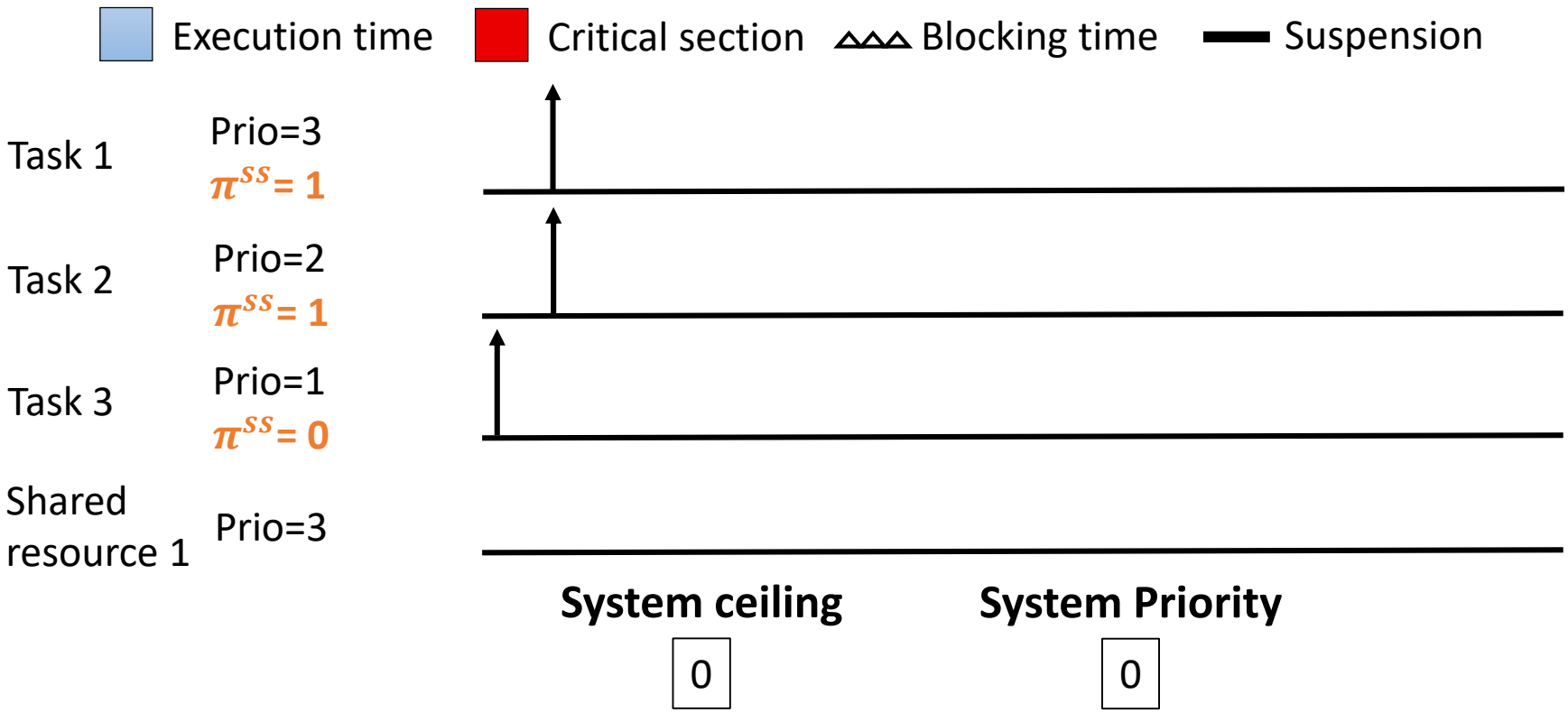
Improving the locking protocol



A task Ti is ***eligible*** to execute if
prio(Ti) > system priority

A task T1 can ***preempt*** another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

The Stack Resource Policy for Self-Suspending tasks (SRP-SS)



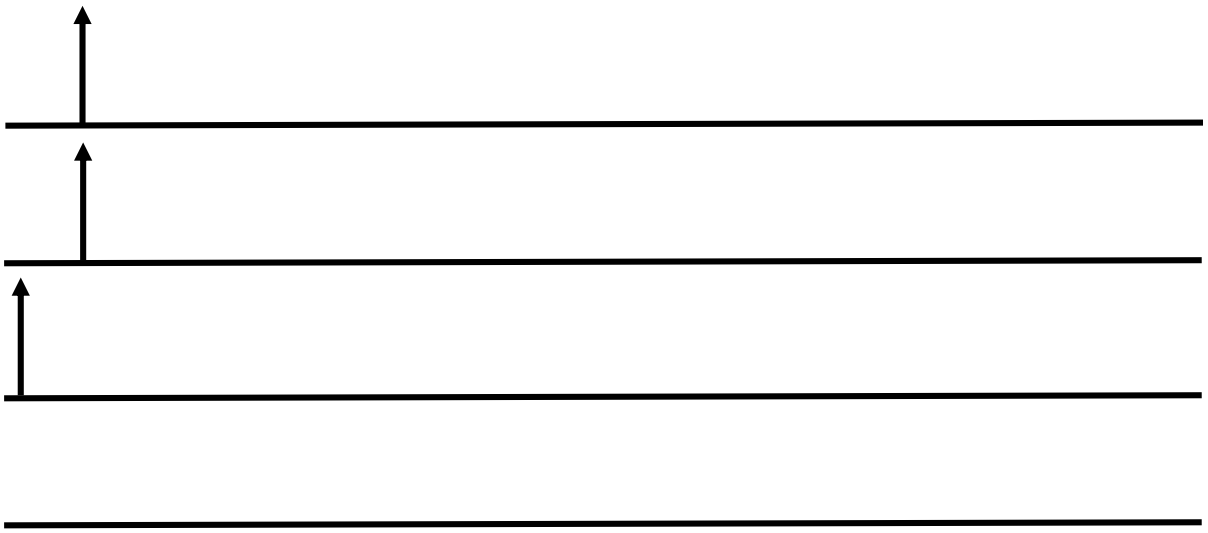
A task Ti is *eligible* to execute if
prio(Ti) > system priority

A task T1 can *preempt* another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

The Stack Resource Policy for Self-Suspending tasks (SRP-SS)

Execution time | Critical section | Blocking time | Suspension

Task 1	Prio=3	π^{SS}= 1
Task 2	Prio=2	π^{SS}= 1
Task 3	Prio=1	π^{SS}= 0
Shared resource 1	Prio=3	



System ceiling 0

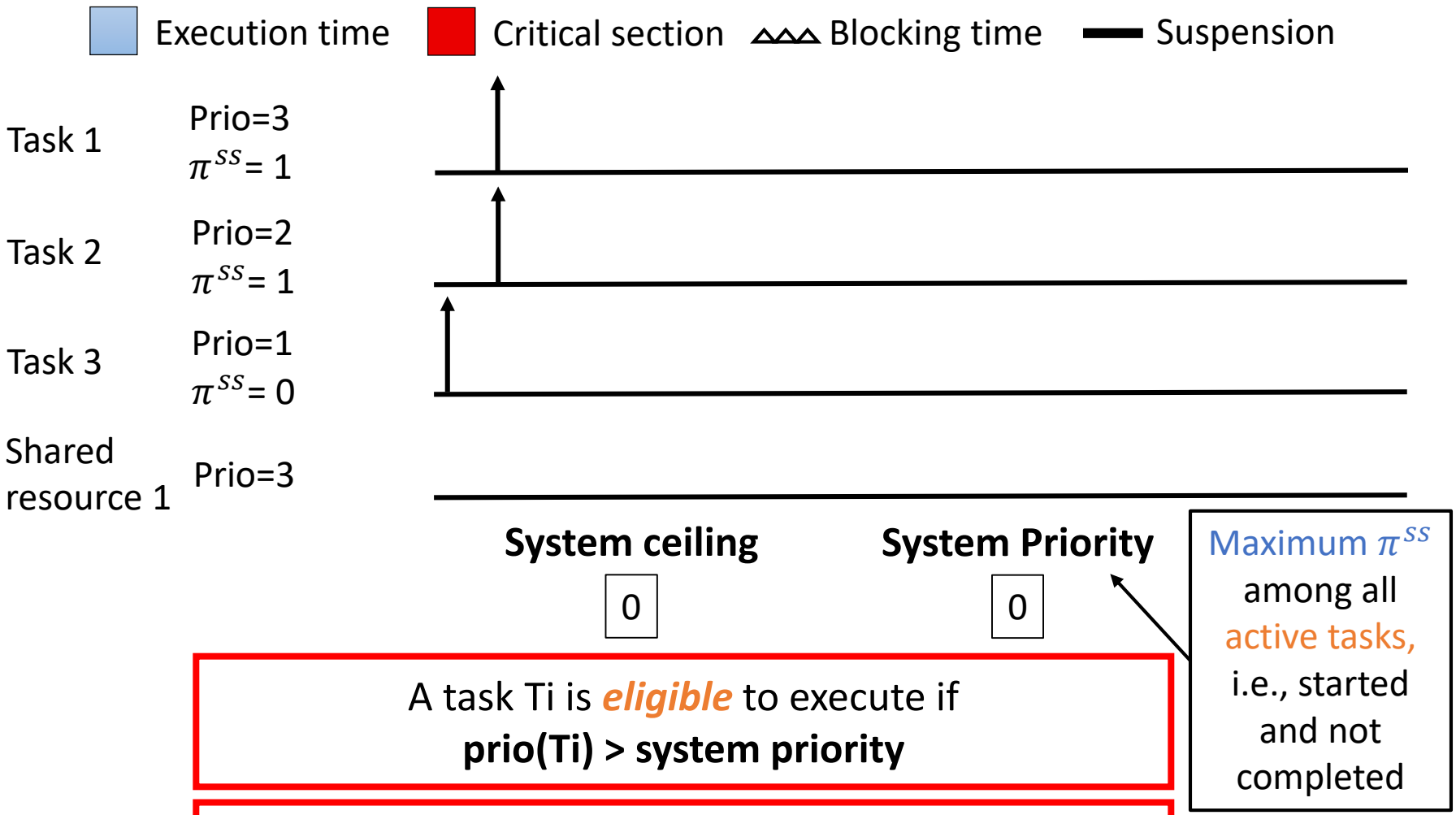
System Priority 0

Maximum π^{SS} among all active tasks, i.e., started and not completed

A task Ti is *eligible* to execute if
prio(Ti) > system priority

A task T1 can *preempt* another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

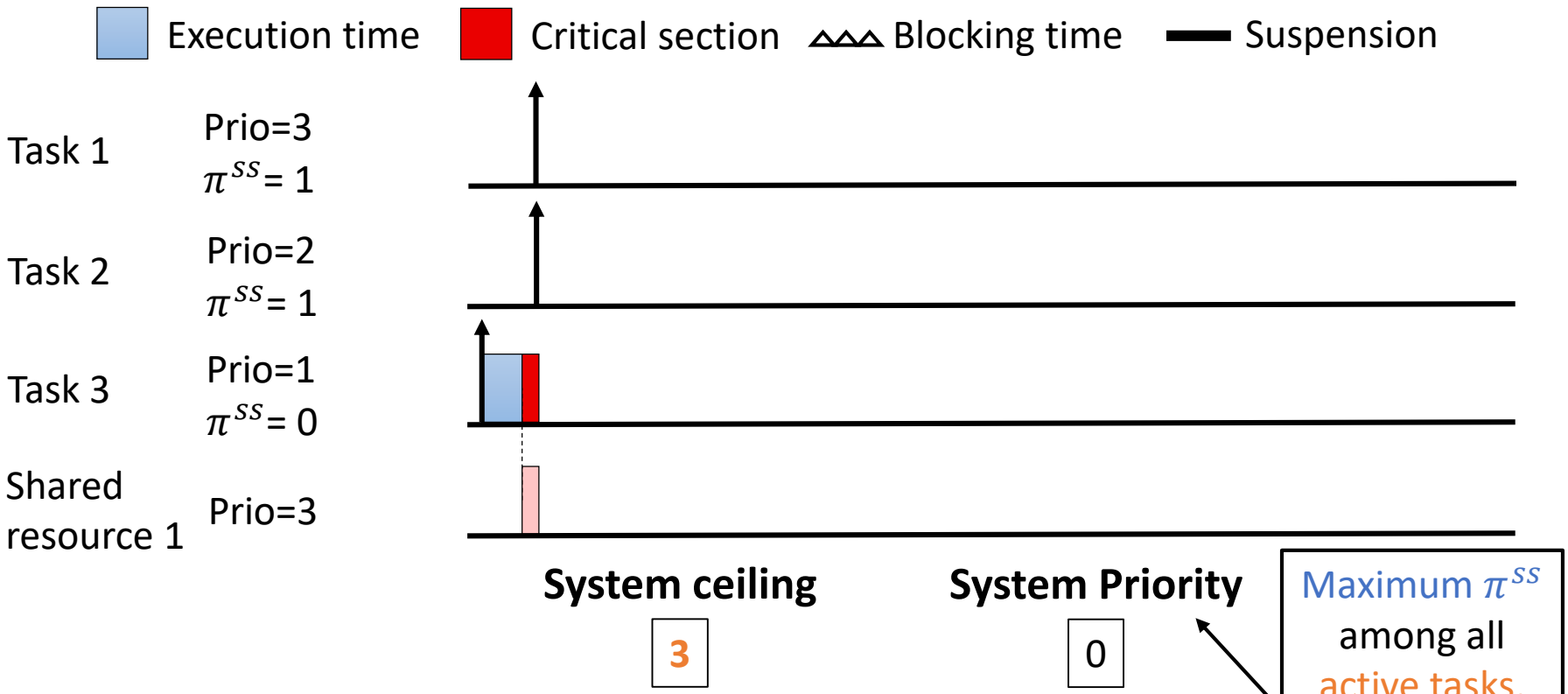
The Stack Resource Policy for Self-Suspending tasks (SRP-SS)



A task Ti is ***eligible*** to execute if
prio(Ti) > system priority

A task T1 can ***preempt*** another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

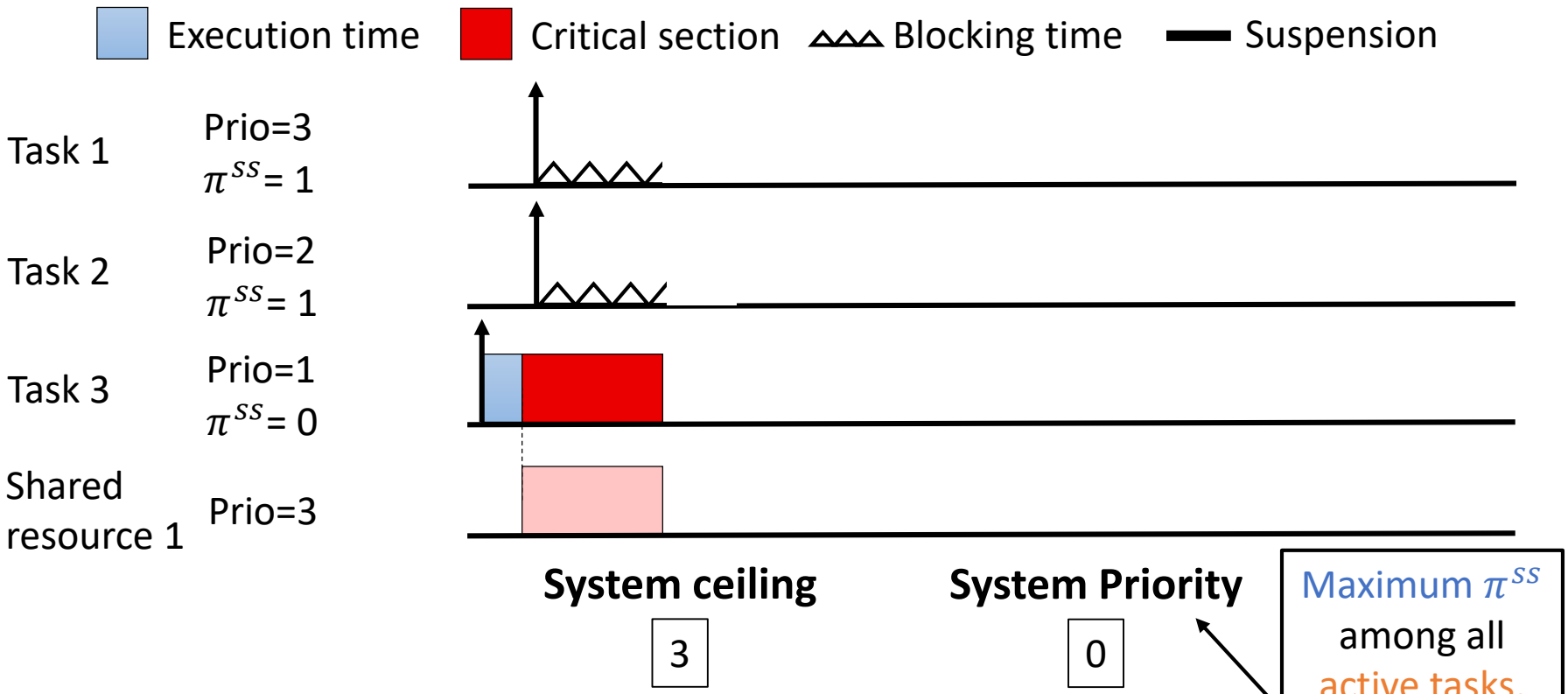
The Stack Resource Policy for Self-Suspending tasks (SRP-SS)



A task Ti is ***eligible*** to execute if
prio(Ti) > system priority

A task T1 can ***preempt*** another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

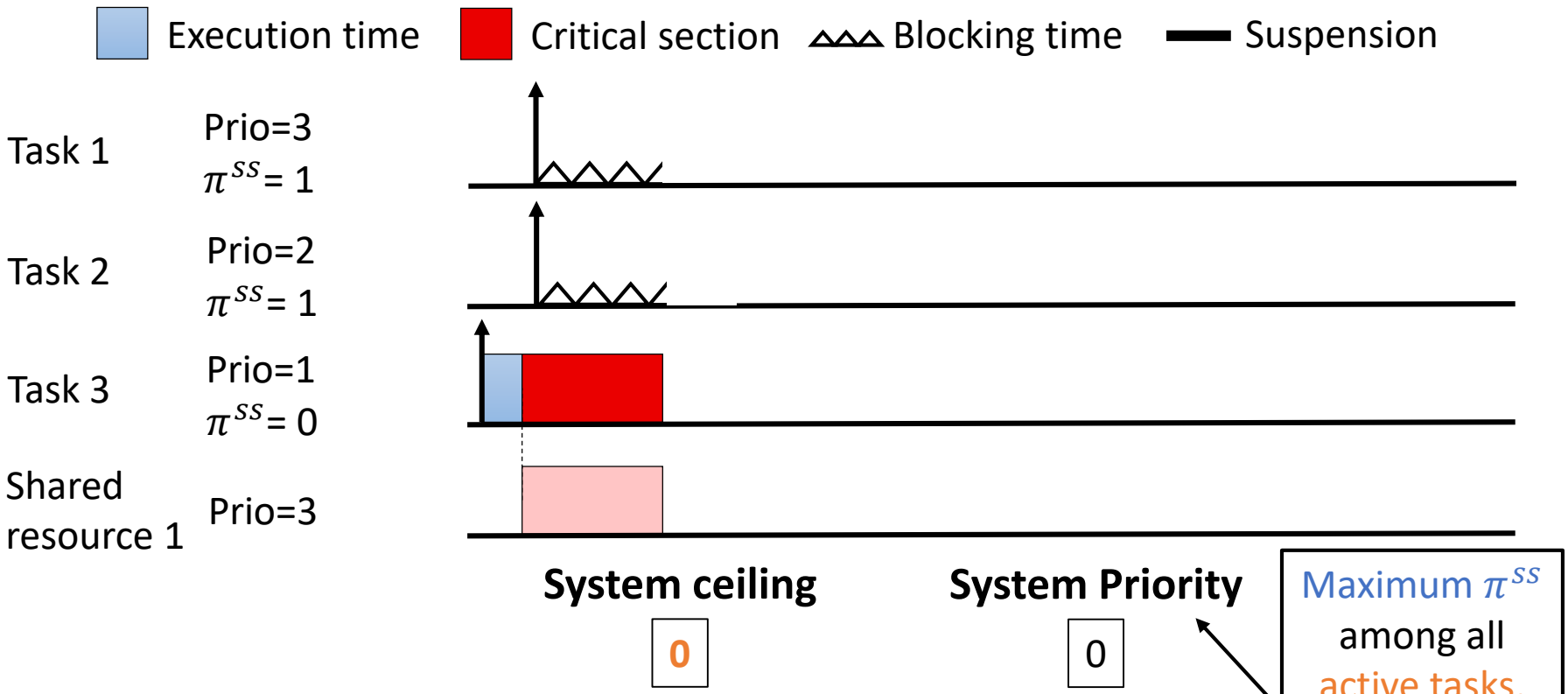
The Stack Resource Policy for Self-Suspending tasks (SRP-SS)



A task Ti is ***eligible*** to execute if
prio(Ti) > system priority

A task T1 can ***preempt*** another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

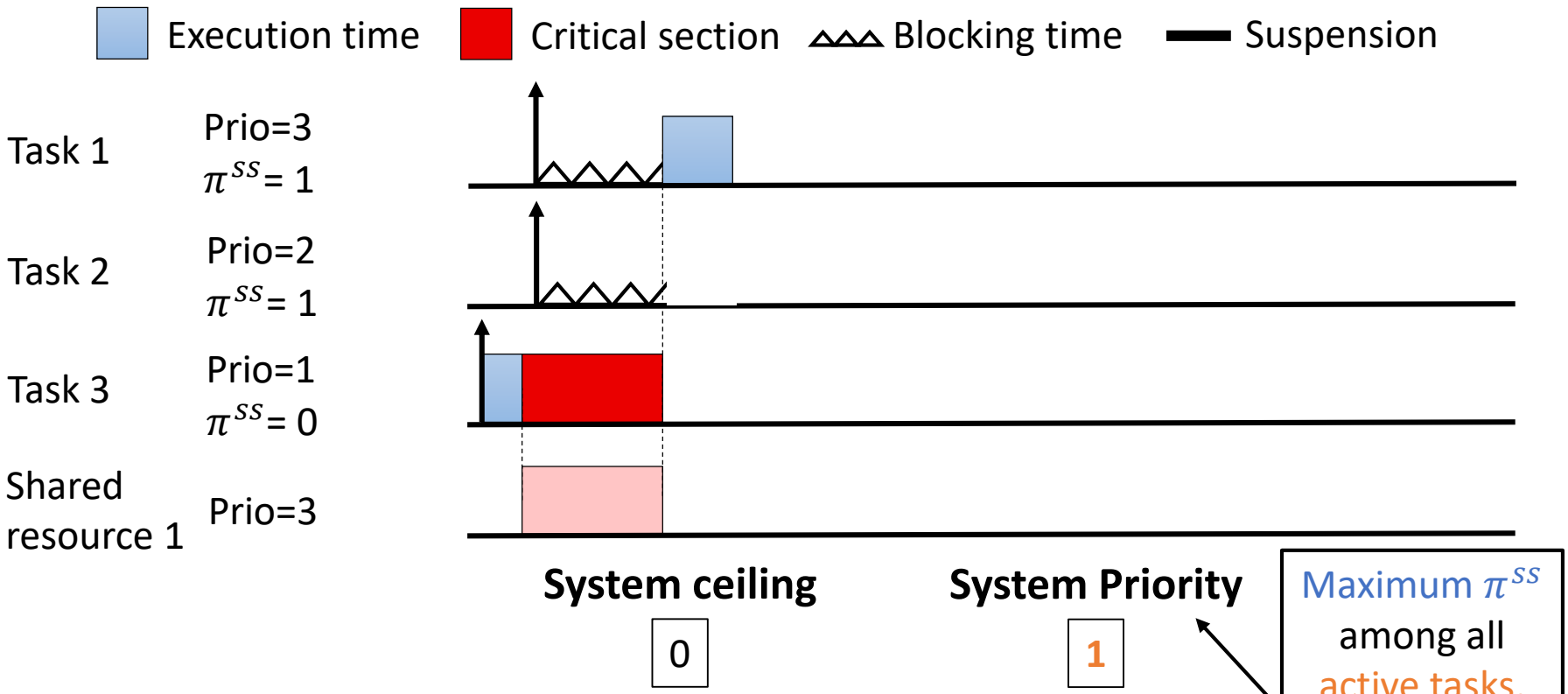
The Stack Resource Policy for Self-Suspending tasks (SRP-SS)



A task Ti is ***eligible*** to execute if
prio(Ti) > system priority

A task T1 can ***preempt*** another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

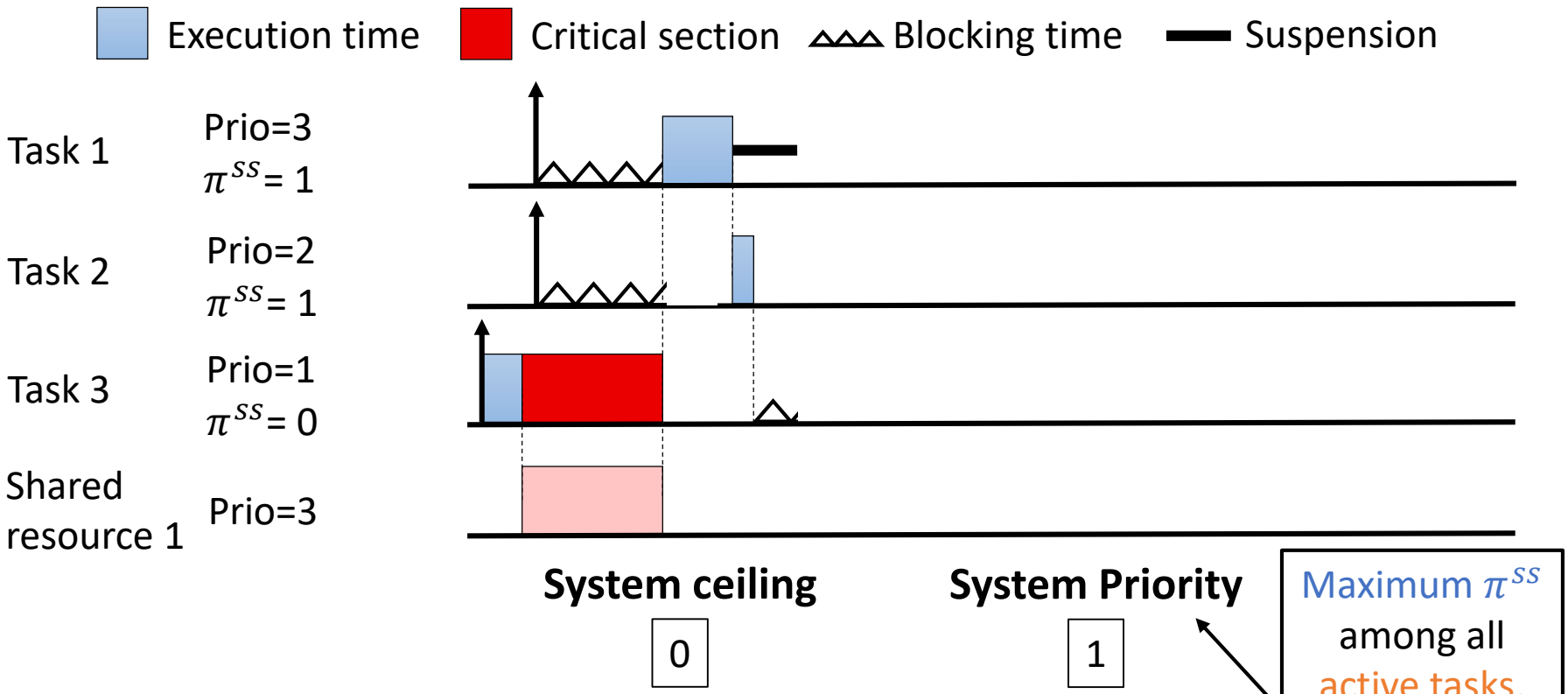
The Stack Resource Policy for Self-Suspending tasks (SRP-SS)



A task Ti is ***eligible*** to execute if
prio(Ti) > system priority

A task T1 can ***preempt*** another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

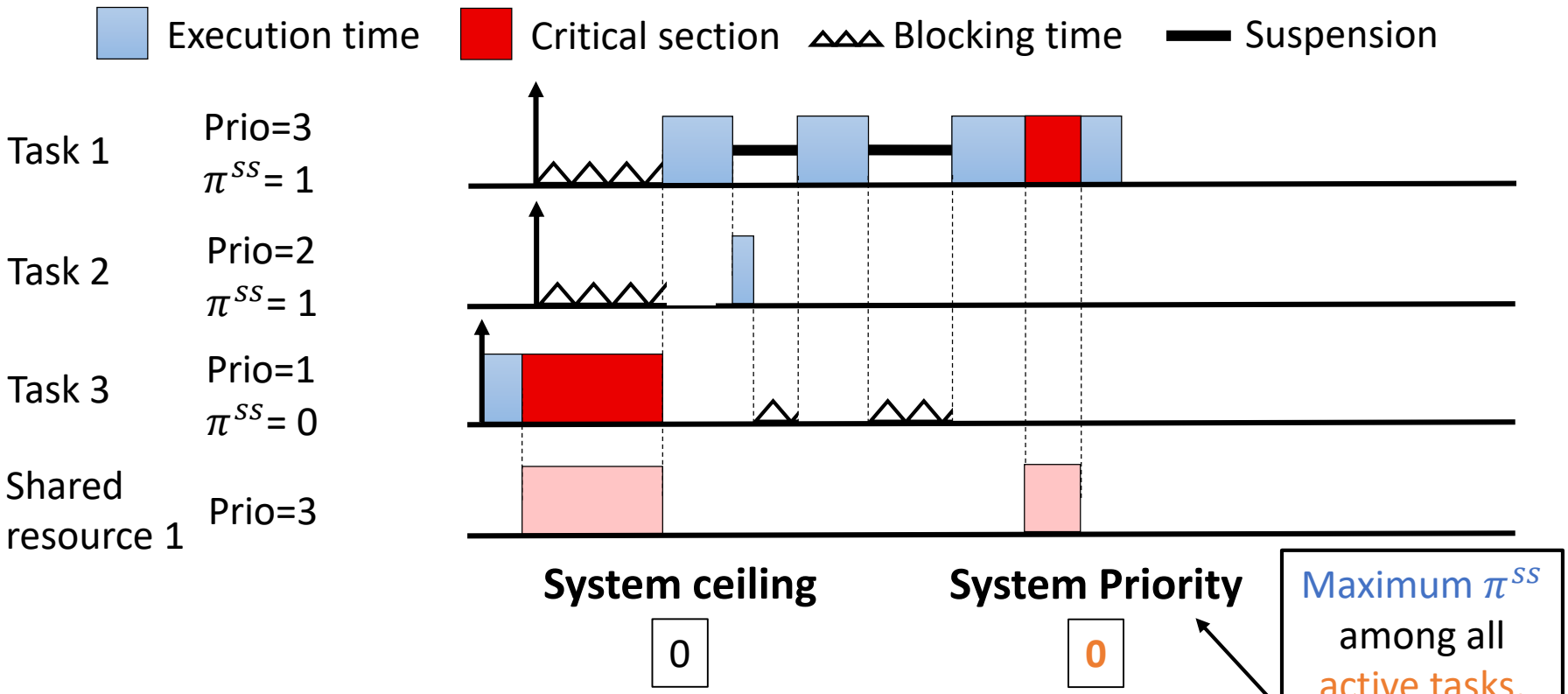
The Stack Resource Policy for Self-Suspending tasks (SRP-SS)



A task Ti is ***eligible*** to execute if
prio(Ti) > system priority

A task T1 can ***preempt*** another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

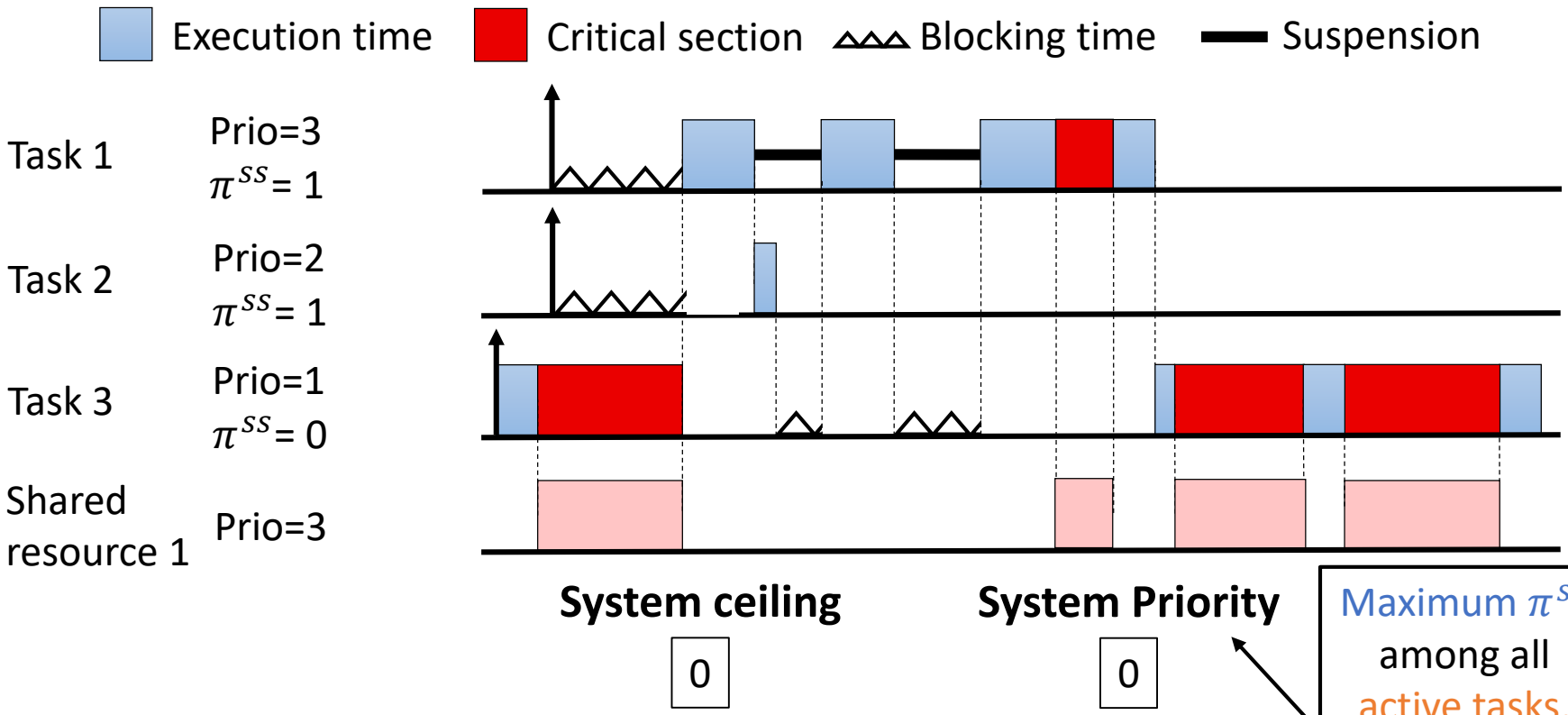
The Stack Resource Policy for Self-Suspending tasks (SRP-SS)



A task Ti is ***eligible*** to execute if
prio(Ti) > system priority

A task T1 can ***preempt*** another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

The Stack Resource Policy for Self-Suspending tasks (SRP-SS)

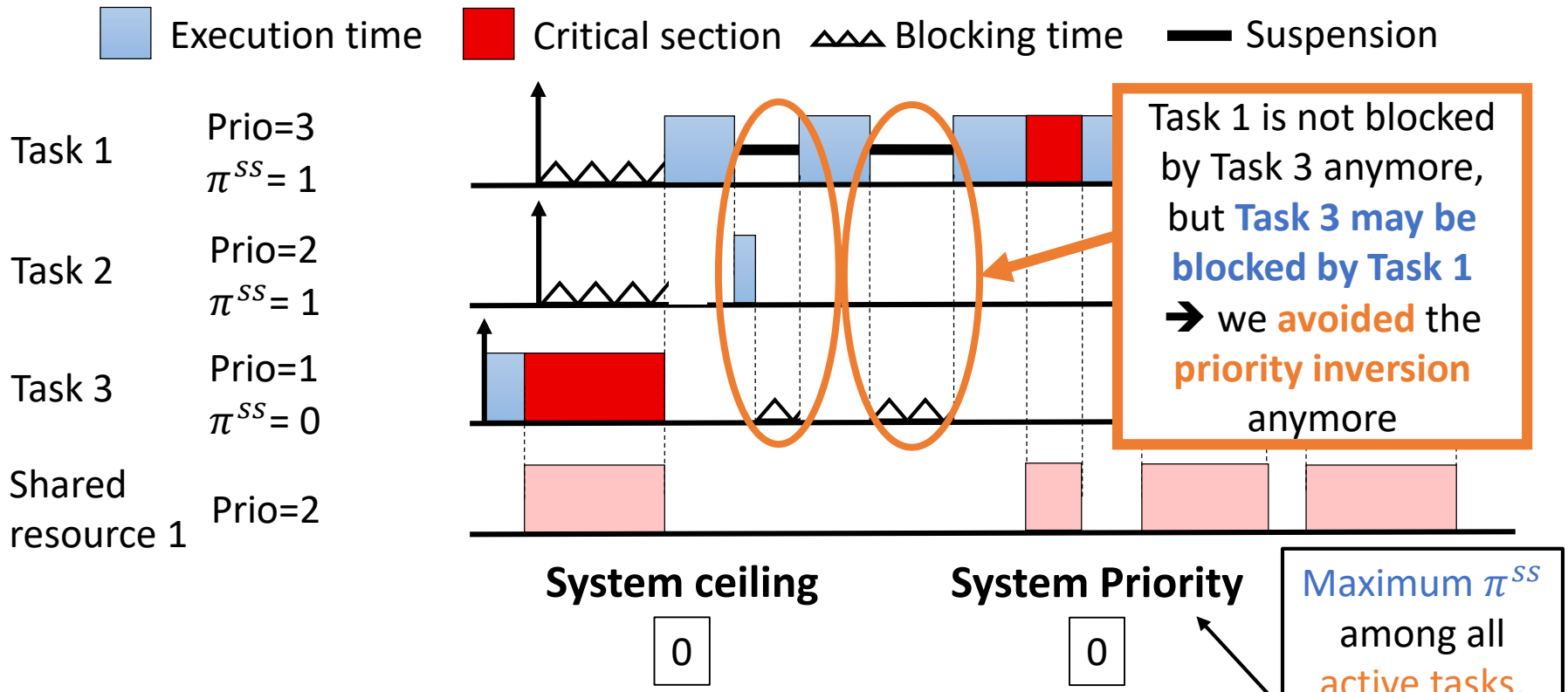


Maximum π^{SS} among all active tasks, i.e., started and not completed

A task Ti is ***eligible*** to execute if
prio(Ti) > system priority

A task T1 can ***preempt*** another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

The Stack Resource Policy for Self-Suspending tasks (SRP-SS)



A task Ti is *eligible* to execute if
prio(Ti) > system priority

A task T1 can *preempt* another task T2 if
prio(T1) > prio(T2) and prio(T1) > system ceiling

Configuring SRP-SS

Configuring SRP-SS

- **New challenge:**
  How do we assign values to the parameters π^{SS} in order to balance PI-blocking and blocking by self-suspensions?

Configuring SRP-SS

- **New challenge:**
  How do we assign values to the parameters π^{SS} in order to balance PI-blocking and blocking by self-suspensions?

- A schedulability test and an iterative configuration algorithm is presented in the paper
  - Effective but not optimal

Configuring SRP-SS

- **New challenge:**
  How do we assign values to the parameters π^{SS} in order to balance PI-blocking and blocking by self-suspensions?

- A schedulability test and an iterative configuration algorithm is presented in the paper
  - Effective but not optimal

- If all π^{SS}=0, then SRP-SS is identical to SRP

Configuring SRP-SS

- **New challenge:**
  How do we assign values to the parameters π^{SS} in order to balance PI-blocking and blocking by self-suspensions?

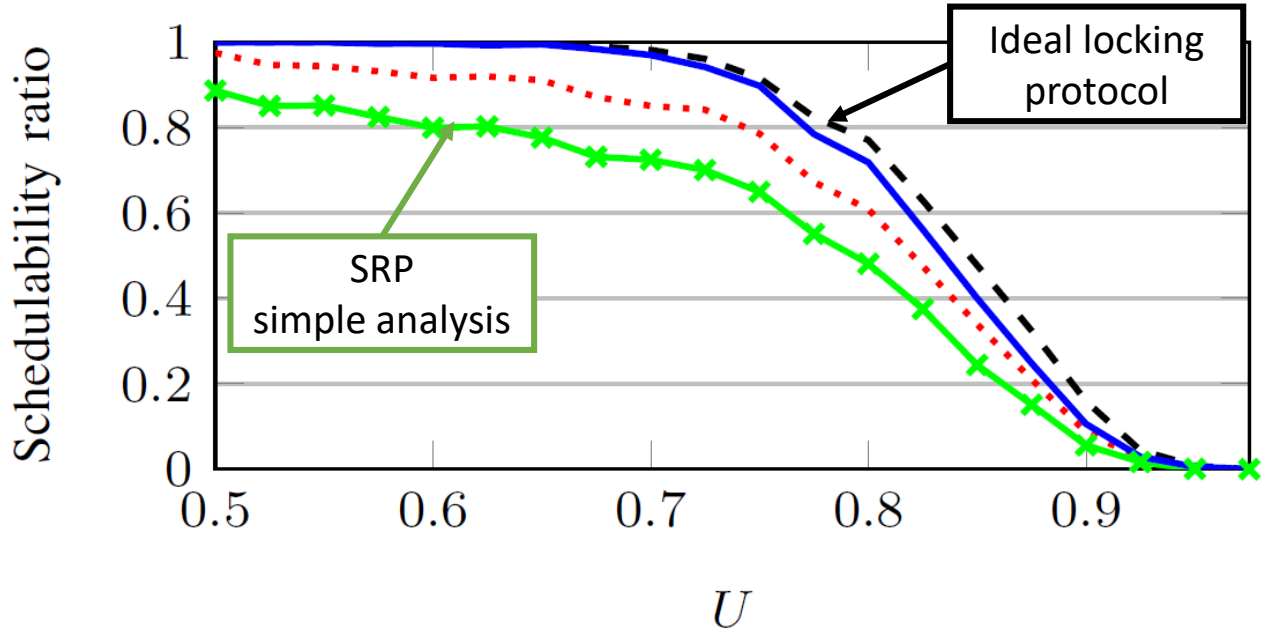
- A schedulability test and an iterative configuration algorithm is presented in the paper
  - Effective but not optimal

- If all π^{SS}=0, then SRP-SS is identical to SRP
  - ➔ SRP-SS dominates SRP

Results

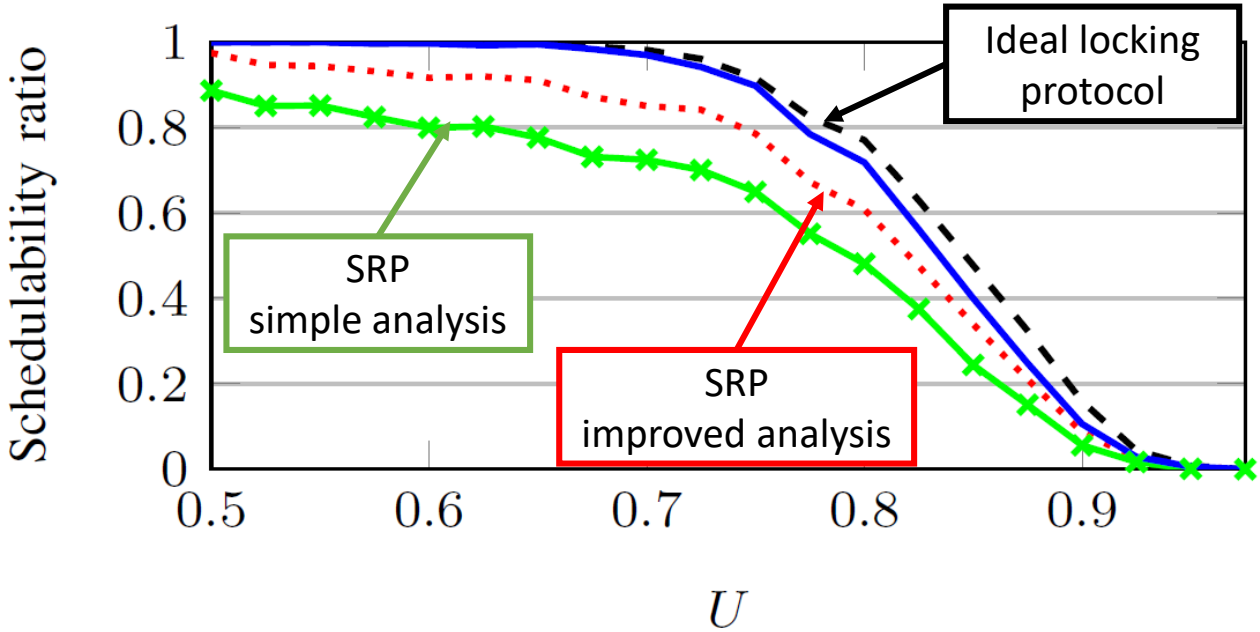
Results

$$n = 10,\ n_r = 4,\ N^{max} = 3, L^{min} = 10,\ L^{max} = 500,$$
$$\sigma^{min} = 0,\ \sigma^{max} = 0.05,\ X^{min} = 2,\ X^{max} = 5$$



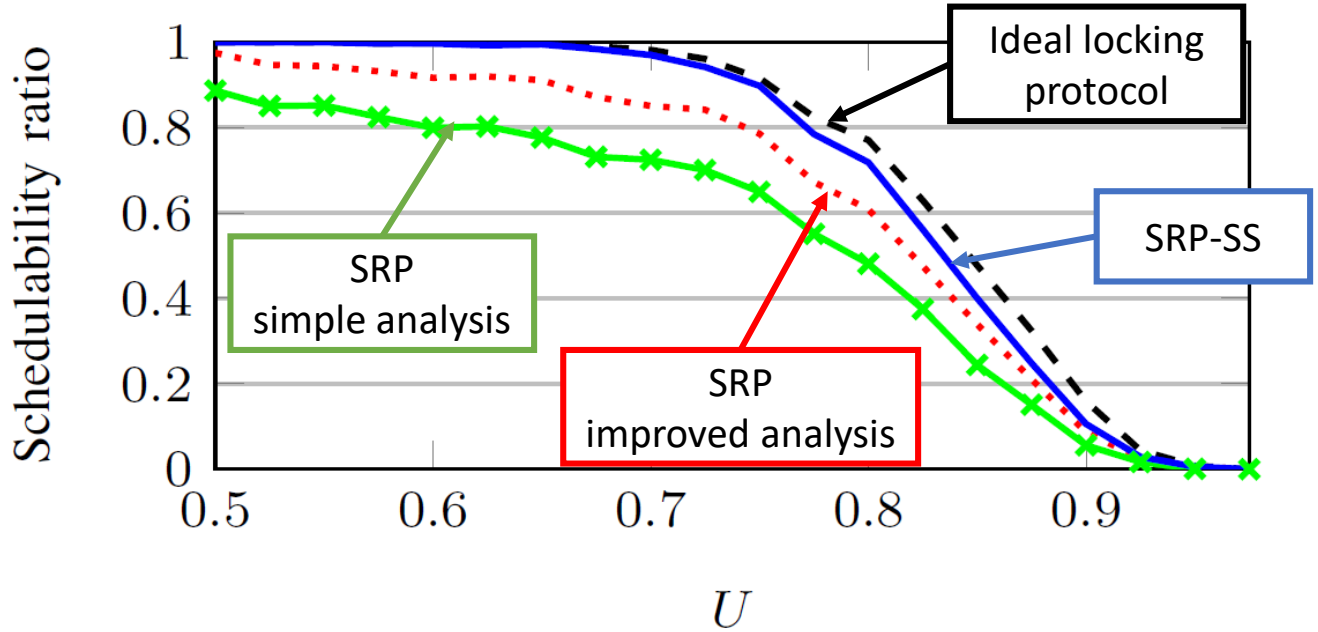
Results

$$n = 10,\ n_r = 4,\ N^{max} = 3, L^{min} = 10,\ L^{max} = 500,$$
$$\sigma^{min} = 0,\ \sigma^{max} = 0.05,\ X^{min} = 2,\ X^{max} = 5$$



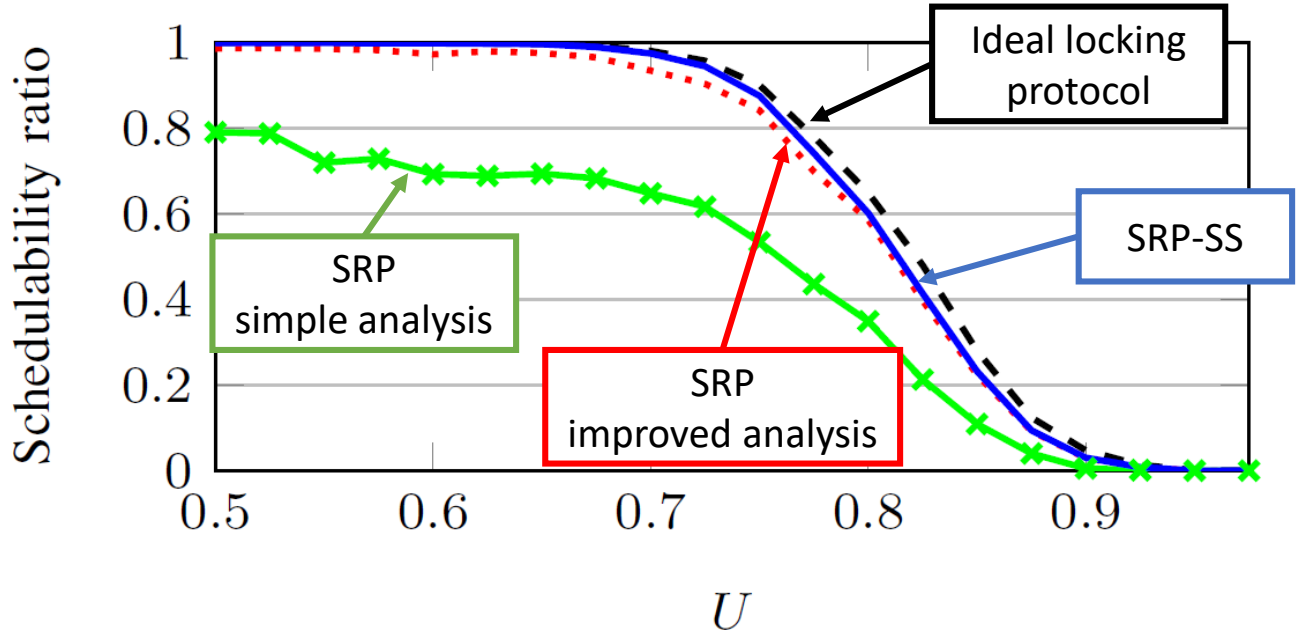
Results

$$n = 10,\ n_r = 4,\ N^{max} = 3, L^{min} = 10,\ L^{max} = 500,$$
$$\sigma^{min} = 0,\ \sigma^{max} = 0.05,\ X^{min} = 2,\ X^{max} = 5$$



Results

$$n = 15,\ n_r = 8,\ N^{max} = 1, L^{min} = 10,\ L^{max} = 200,$$
$$\sigma^{min} = 0,\ \sigma^{max} = 0.1,\ X^{min} = 4,\ X^{max} = 8$$



Conclusion

Conclusion

- We developed a new response time analysis for self-suspending tasks using SRP

Conclusion

- We developed a new response time analysis for self-suspending tasks using SRP
- We extended SRP to limit the number of priority inversions self-suspending tasks may suffer
  - Adding one task parameter and system priority

Conclusion

- We developed a new response time analysis for self-suspending tasks using SRP
- We extended SRP to limit the number of priority inversions self-suspending tasks may suffer
  - Adding one task parameter and system priority
- The SRP-SS dominates the SRP

Conclusion

- We developed a new response time analysis for self-suspending tasks using SRP
- We extended SRP to limit the number of priority inversions self-suspending tasks may suffer
  - Adding one task parameter and system priority
- The SRP-SS dominates the SRP
- We propose a configuration algorithm of the SRP-SS to balance PI-blocking and interference suffered by low priority tasks