Real-time scheduling of STM transactions on multi-core platforms

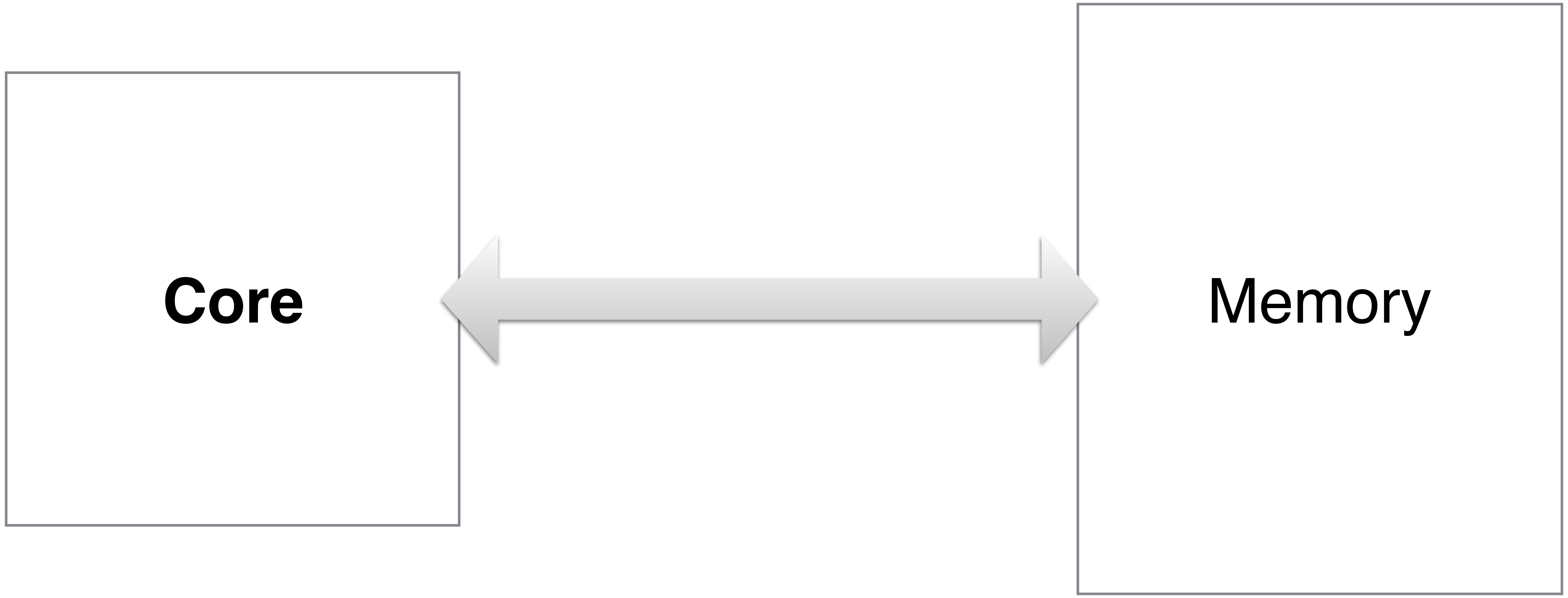
António Barros, Patrick Meumeu Yomsi, Luís Miguel Pinho

CISTER seminar series

xxth February 2015

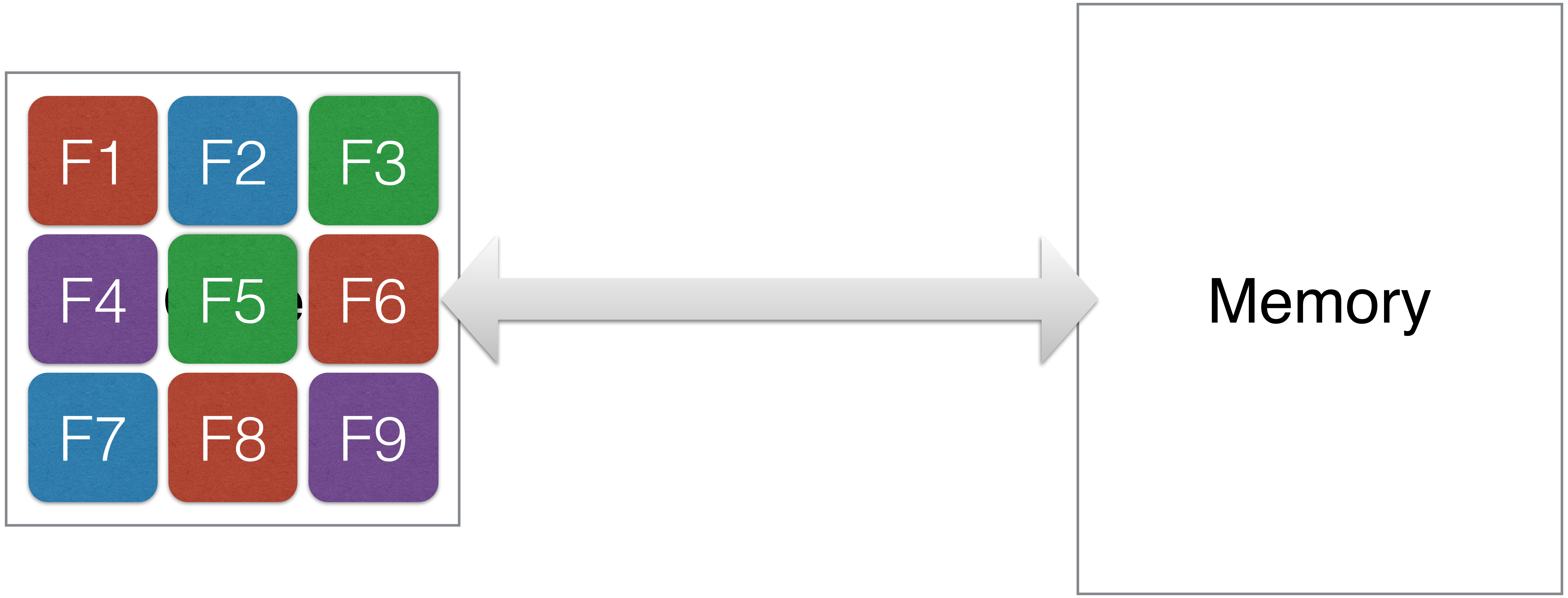
The problem

Single core



The problem

Single core

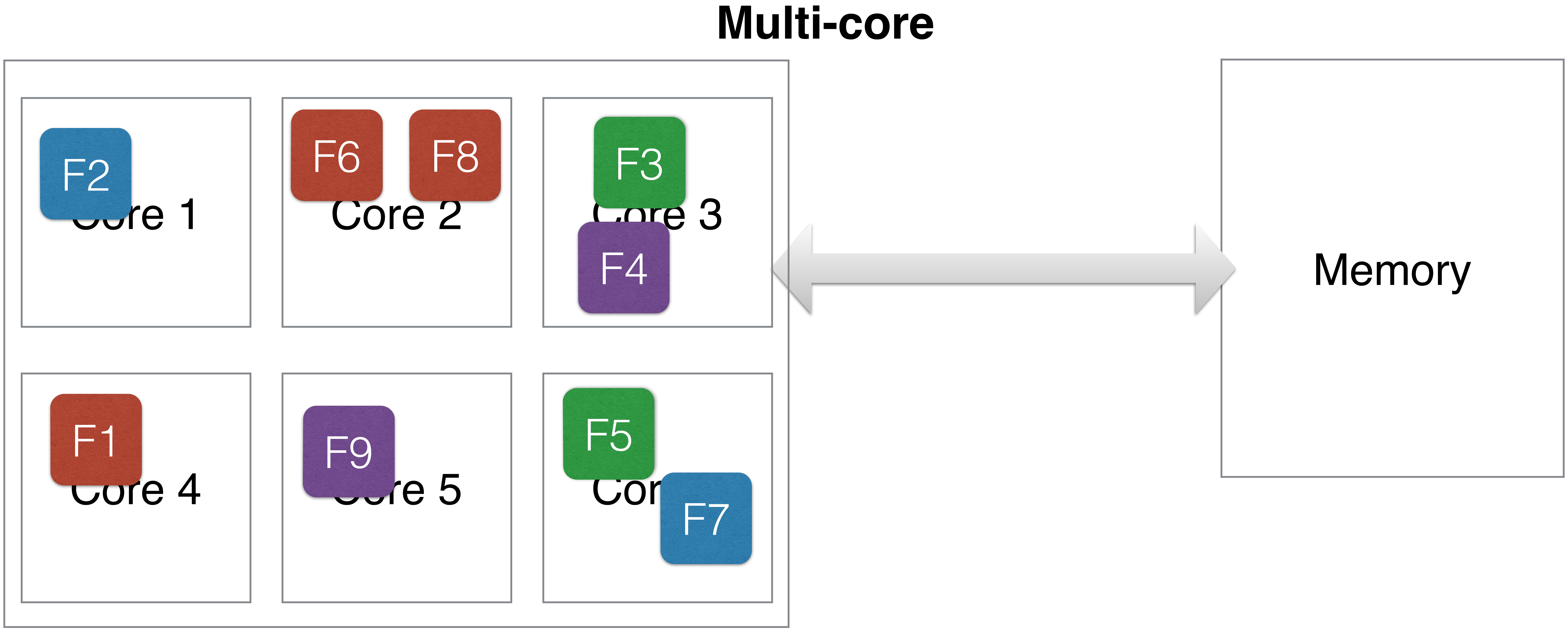


The problem

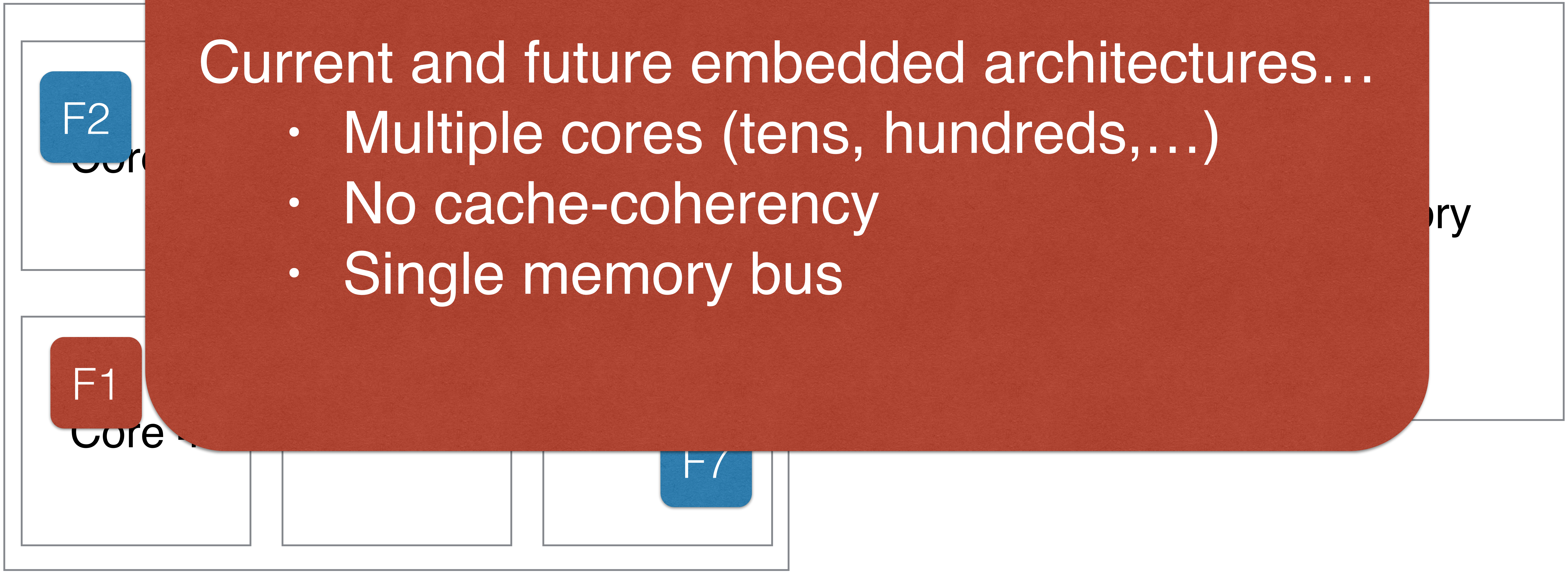
Well understood theory and practice on unicore platforms!

Memory

The problem


Multi-core
F2
Core 1
F6
F8
Core 2
F3
Core 3
F4
F1
Core 4
F9
Core 5
F5
Cor
F7
Memory

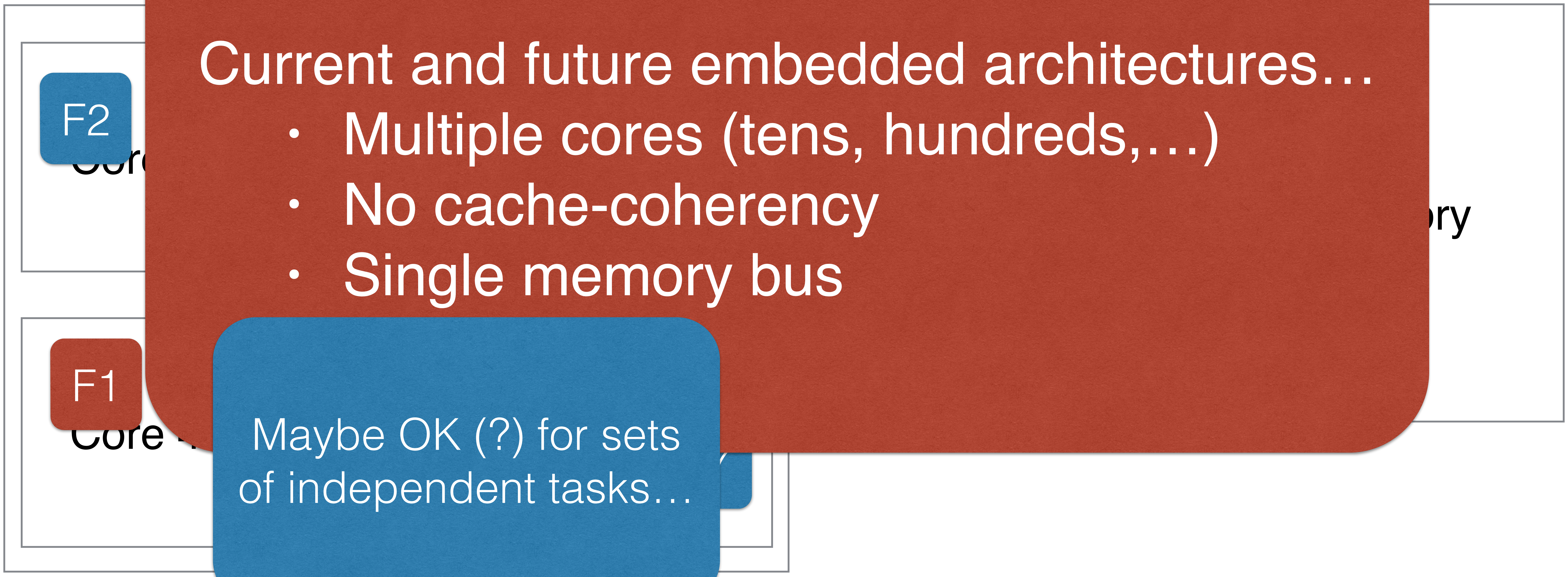
The problem



Current and future embedded architectures…

- Multiple cores (tens, hundreds,…)
- No cache-coherency
- Single memory bus

The problem

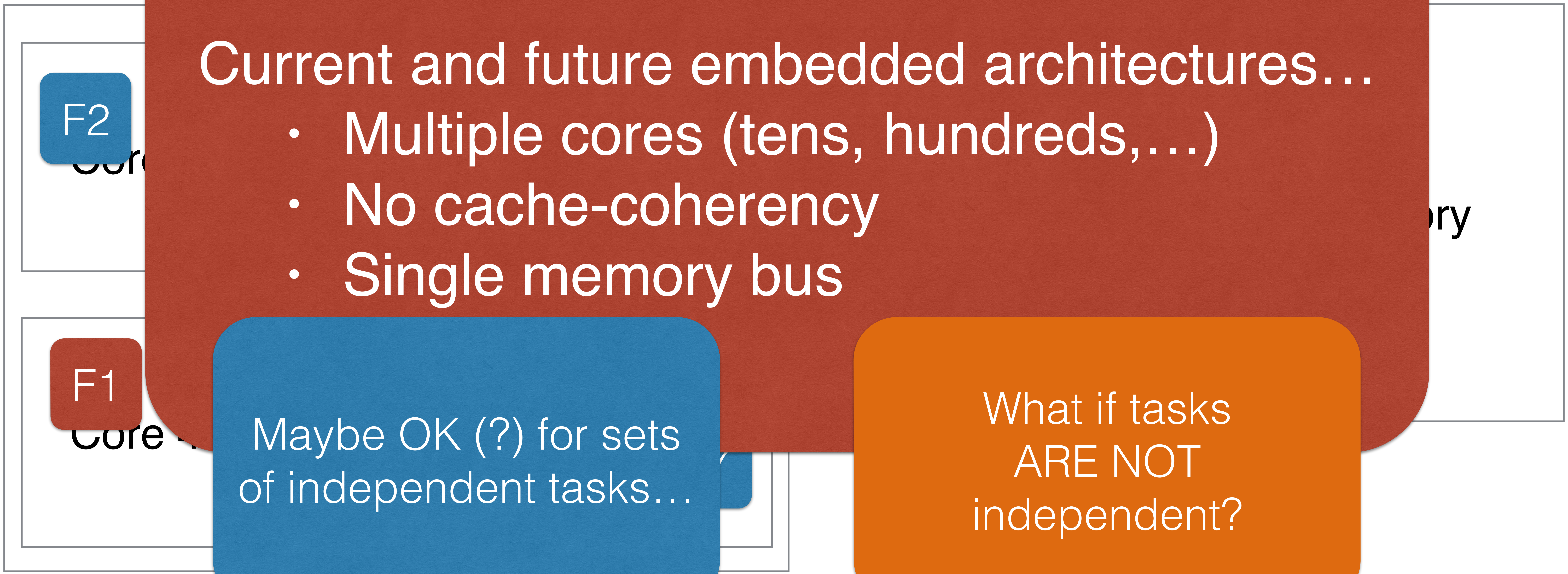


Current and future embedded architectures…

- Multiple cores (tens, hundreds,…)
- No cache-coherency
- Single memory bus

Maybe OK (?) for sets of independent tasks…

The problem



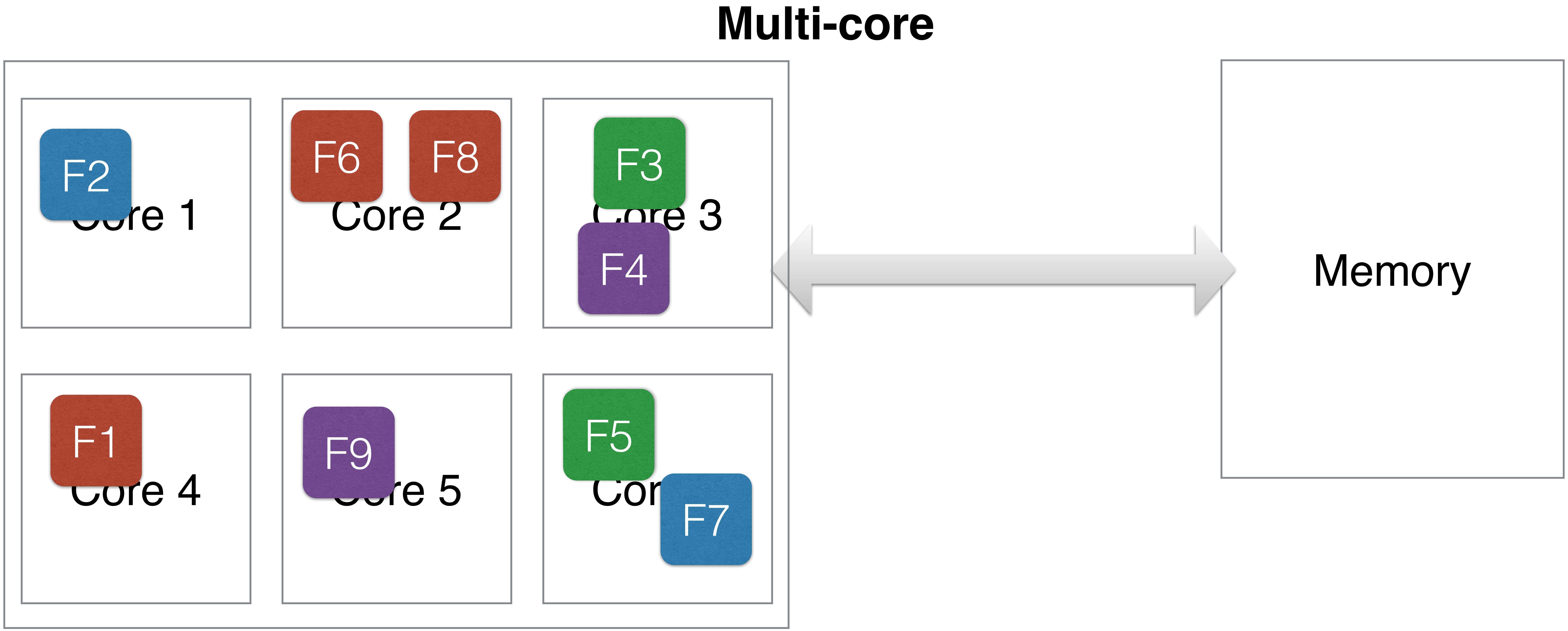
Current and future embedded architectures…

- Multiple cores (tens, hundreds,…)
- No cache-coherency
- Single memory bus

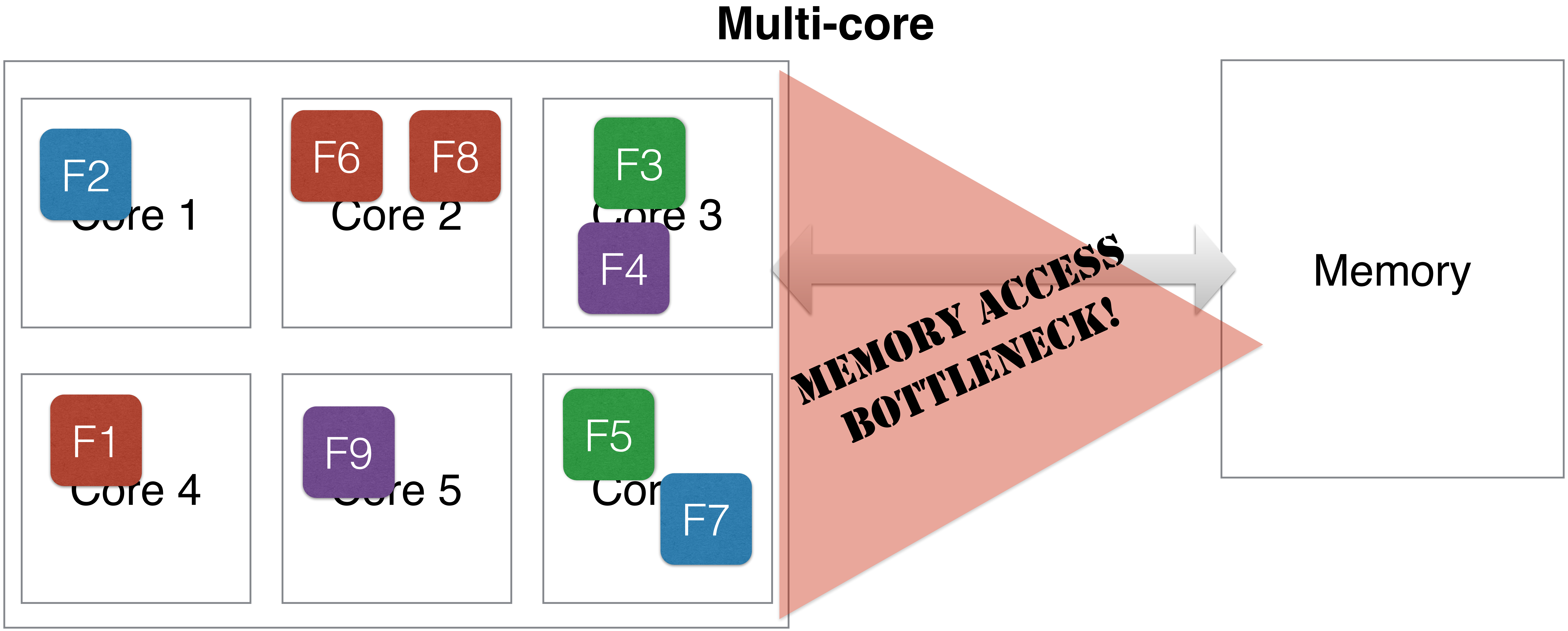
Maybe OK (?) for sets of independent tasks…

What if tasks ARE NOT independent?

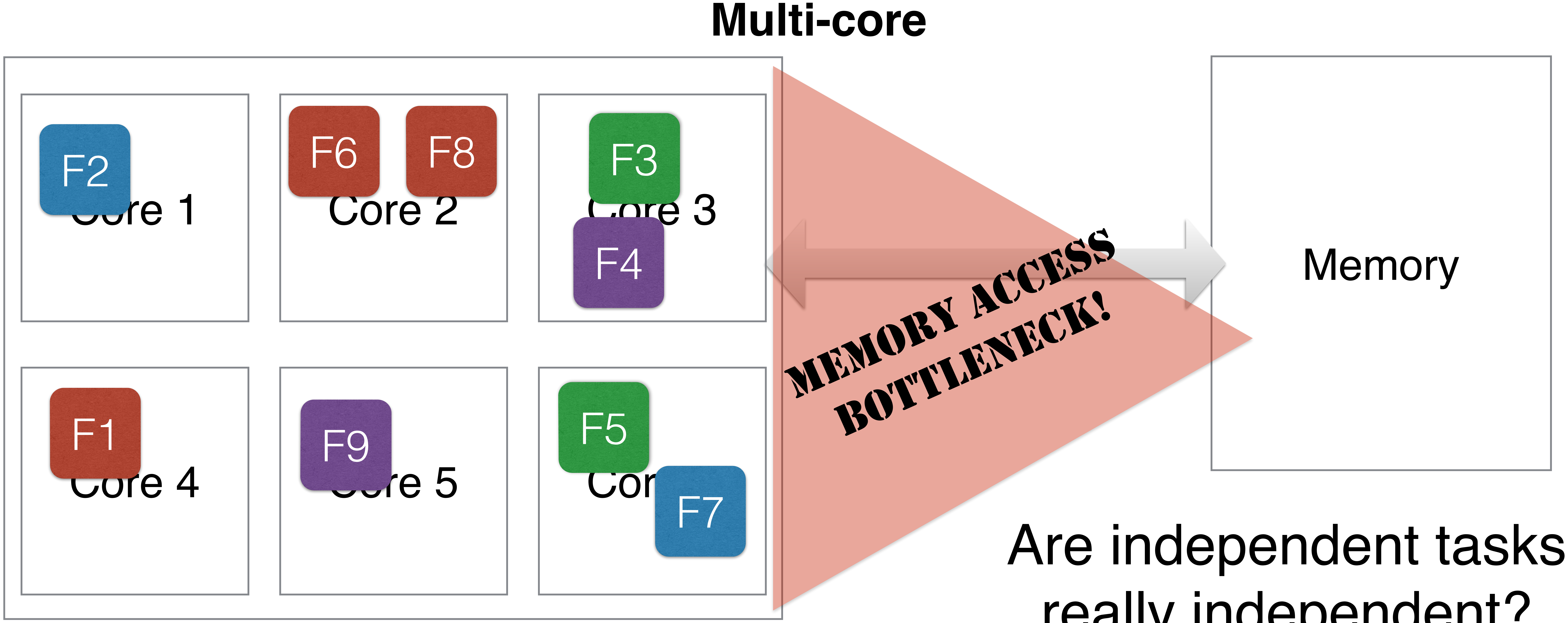
The problem


Multi-core
F2
Core 1
F6
F8
Core 2
F3
Core 3
F4
F1
Core 4
F9
Core 5
F5
Core 6
F7
Memory

The problem


Multi-core
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
F2
F6
F8
F3
F4
F1
F9
F5
F7
Memory
MEMORY ACCESS
BOTTLENECK!

The problem



Are independent tasks really independent?

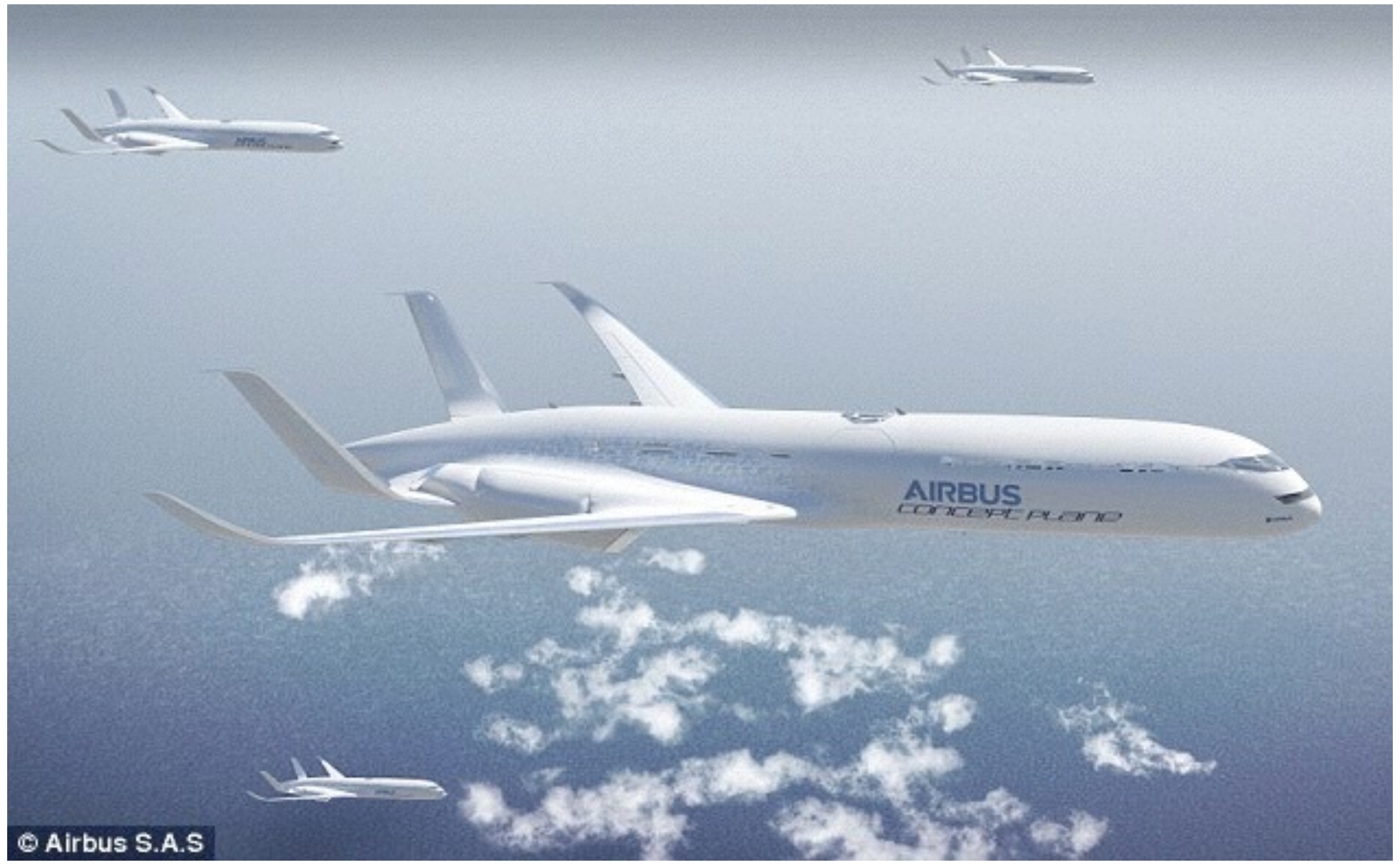
Practical case: DO-178C

- New programming paradigm (enhancing explicit dependencies between functionalities).

- Spatial and temporal isolation among functionalities, depending on their criticality.

- Functionalities must be statically assigned to cores.

- Data dependencies must be mapped.



Attempted solutions

Recent proposal: FMLP*

- Global resources can be short or long (designer’s choice), depending on length of critical sections
    - When blocked: busy waits on short, suspends on long
- Nested requests dictates joining resources into resource groups
    - one lock (queue lock or semaphore) per group
    - exclusively short- and long-groups
- Critical section code is executed non-preemptively

* A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In Proceedings of RTCSA 2007, pages 71–80, 2007.

Attempted solutions

Our idea: STM + SRP-TM

- No groups and locks (at least, seen by the programmer)
- Contention is checked at run-time: just-in-time parallelism
- Upper-bound atomic section response time
- Limit blocking times

Background

Locks

- Coarse-grained locking

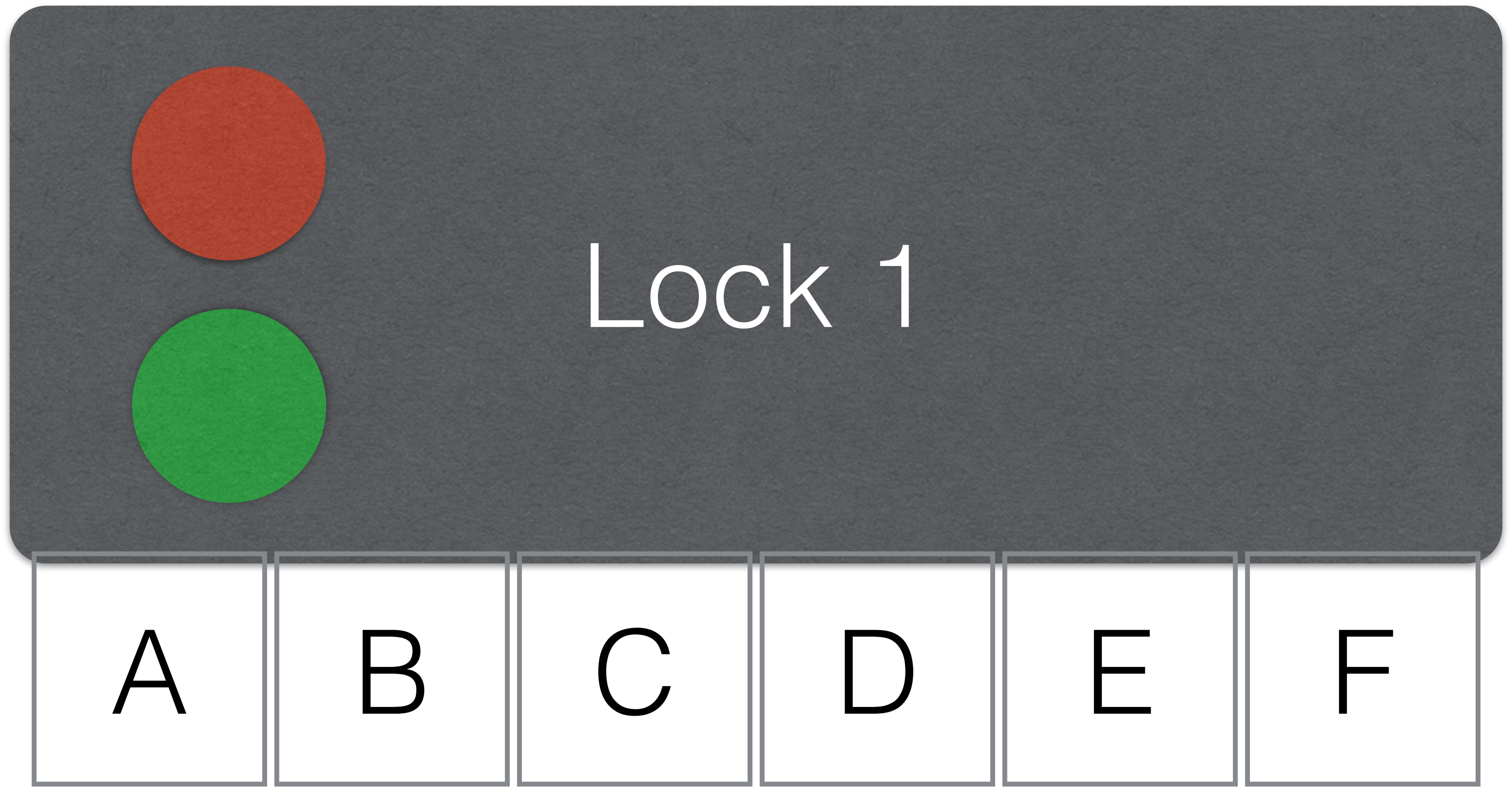
Locks

- Coarse-grained locking

A	B	C	D	E	F

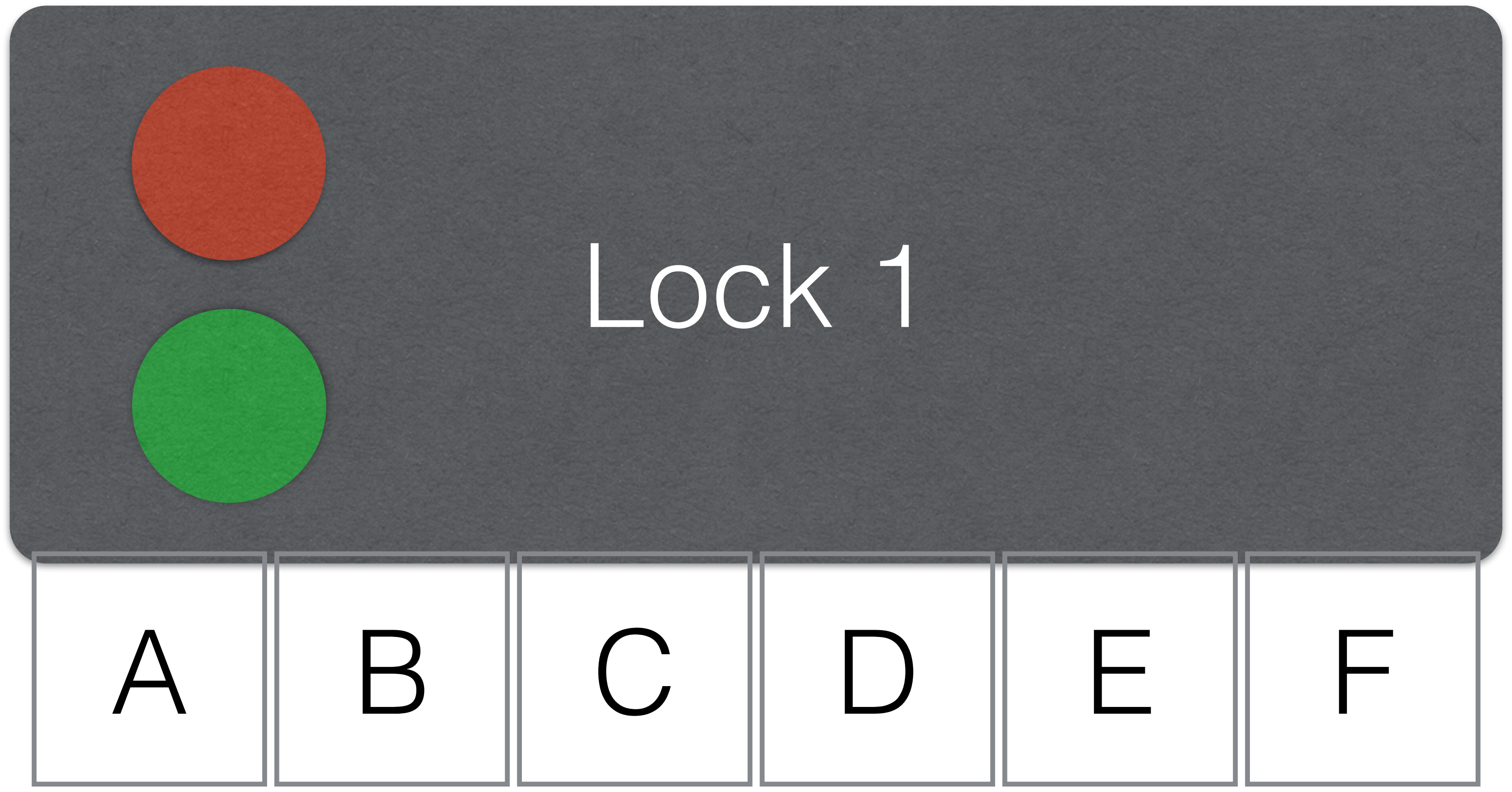
Locks

- Coarse-grained locking



Locks

- Coarse-grained locking

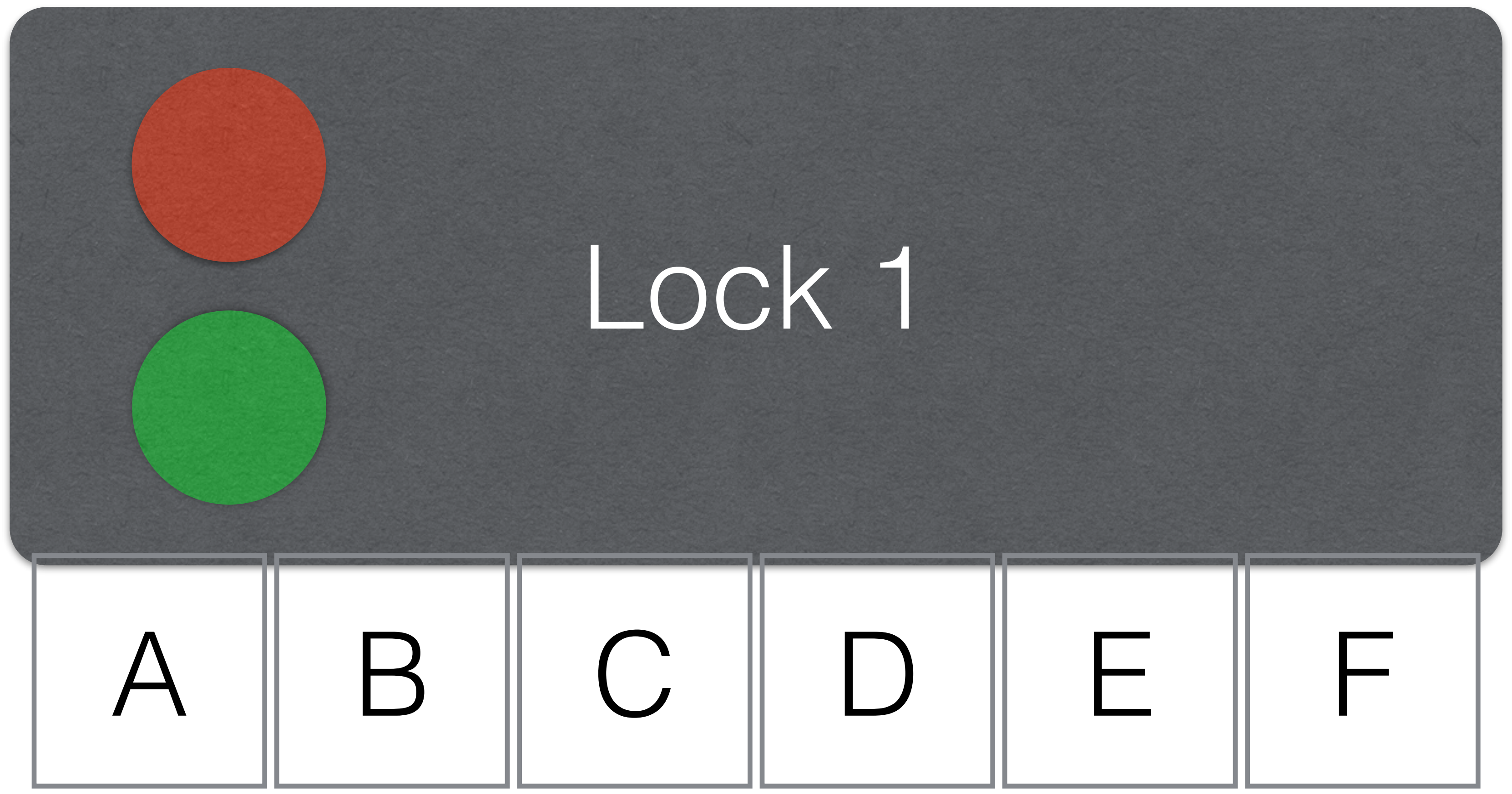


TASK 1

```
Get_Lock(1)
Write(A, x)
Release_Lock(1)
```

Locks

- Coarse-grained locking



TASK 1

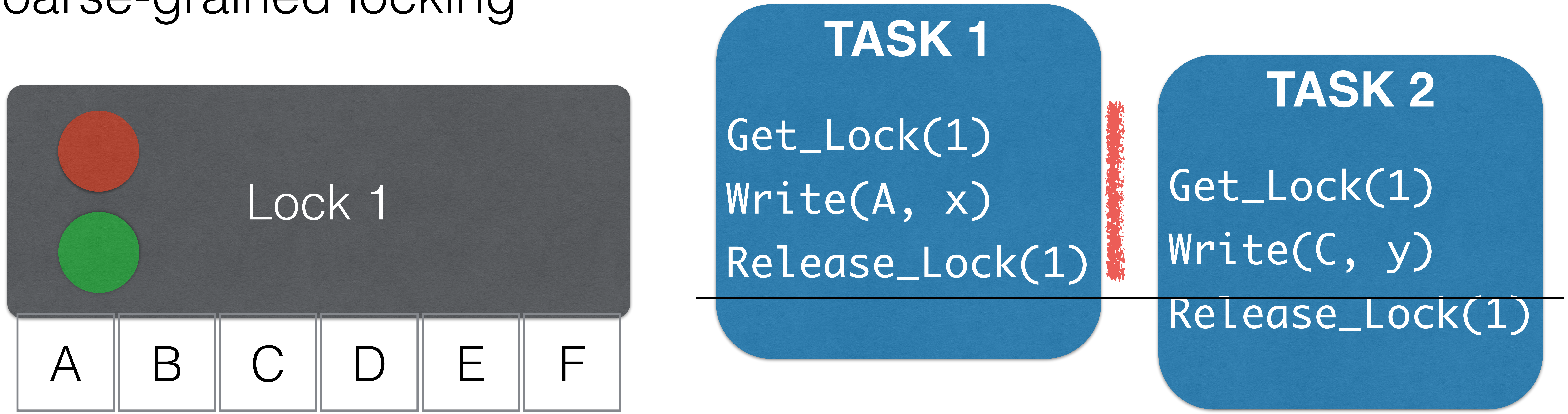
```
Get_Lock(1)
Write(A, x)
Release_Lock(1)
```

TASK 2

```
Get_Lock(1)
Write(C, y)
Release_Lock(1)
```

Locks

- Coarse-grained locking



TASK 1

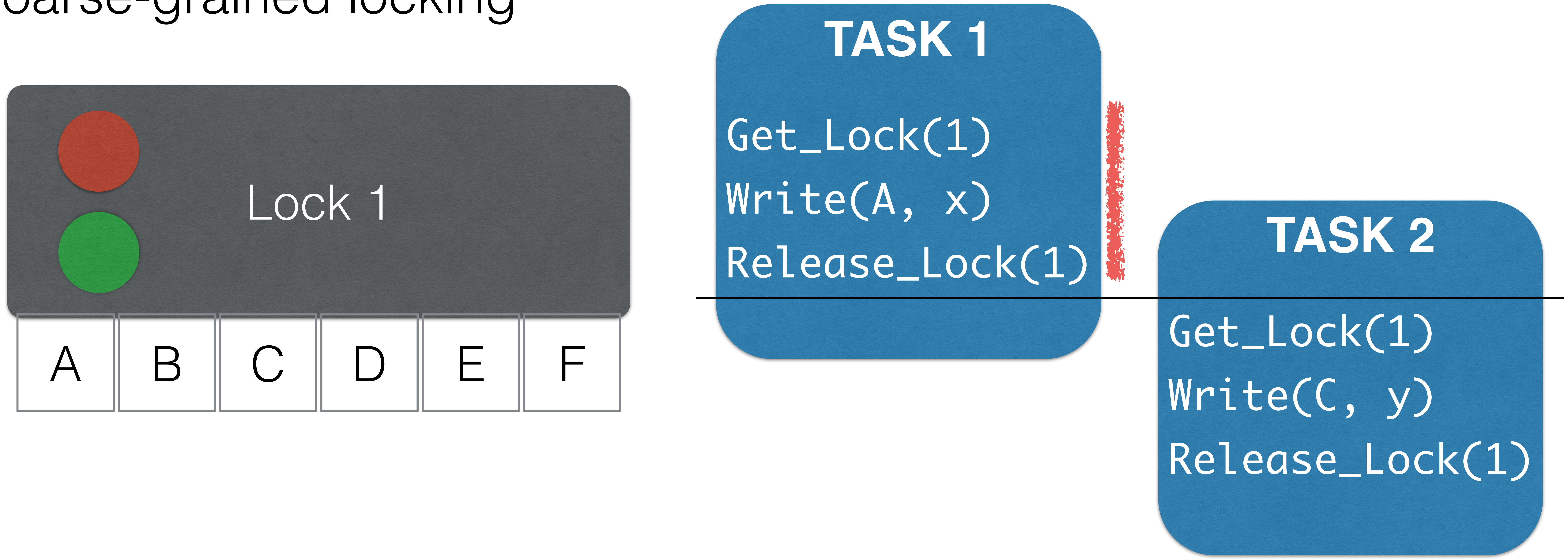
```
Get_Lock(1)
Write(A, x)
Release_Lock(1)
```

TASK 2

```
Get_Lock(1)
Write(C, y)
Release_Lock(1)
```

Locks

- Coarse-grained locking

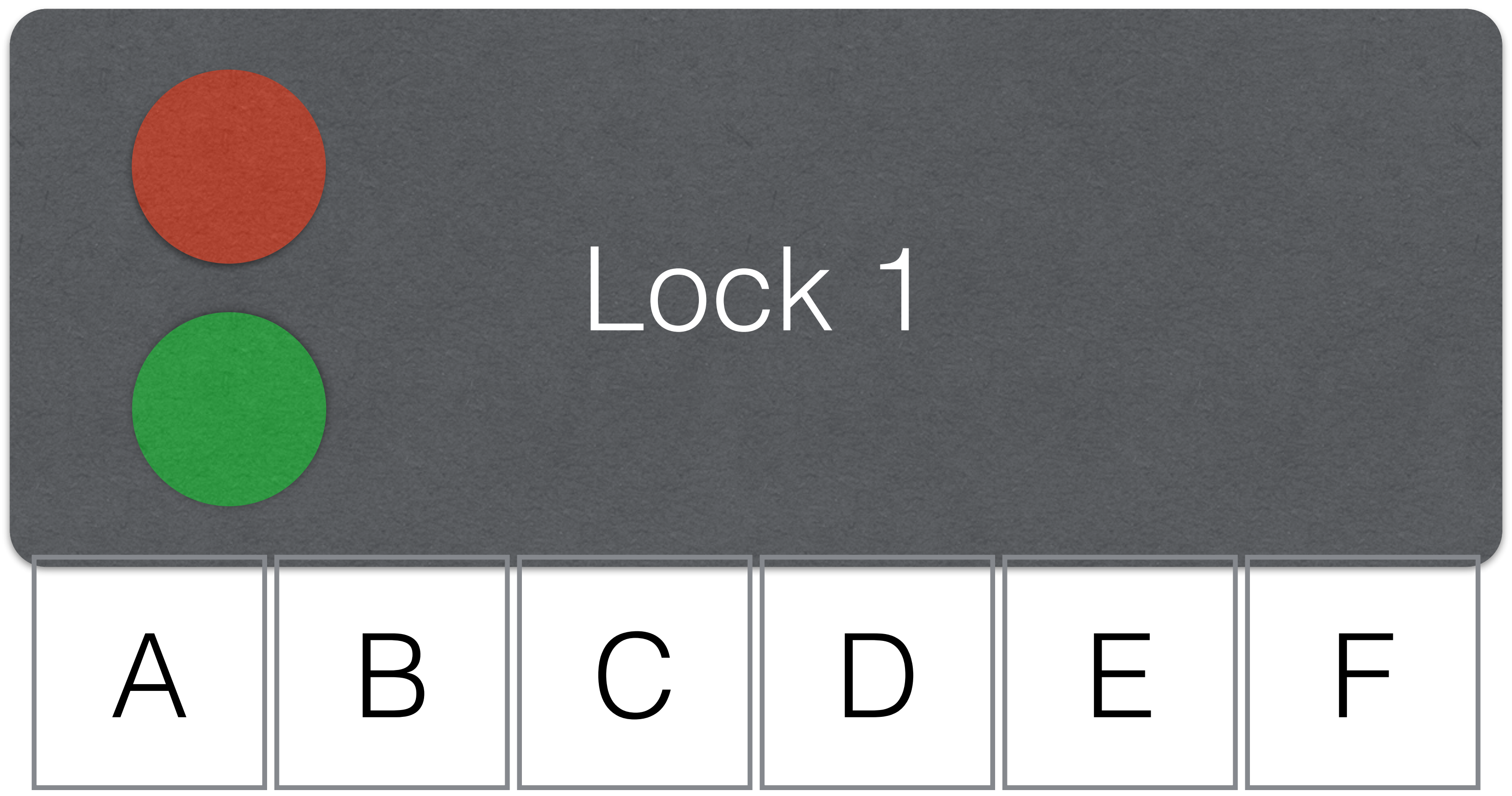


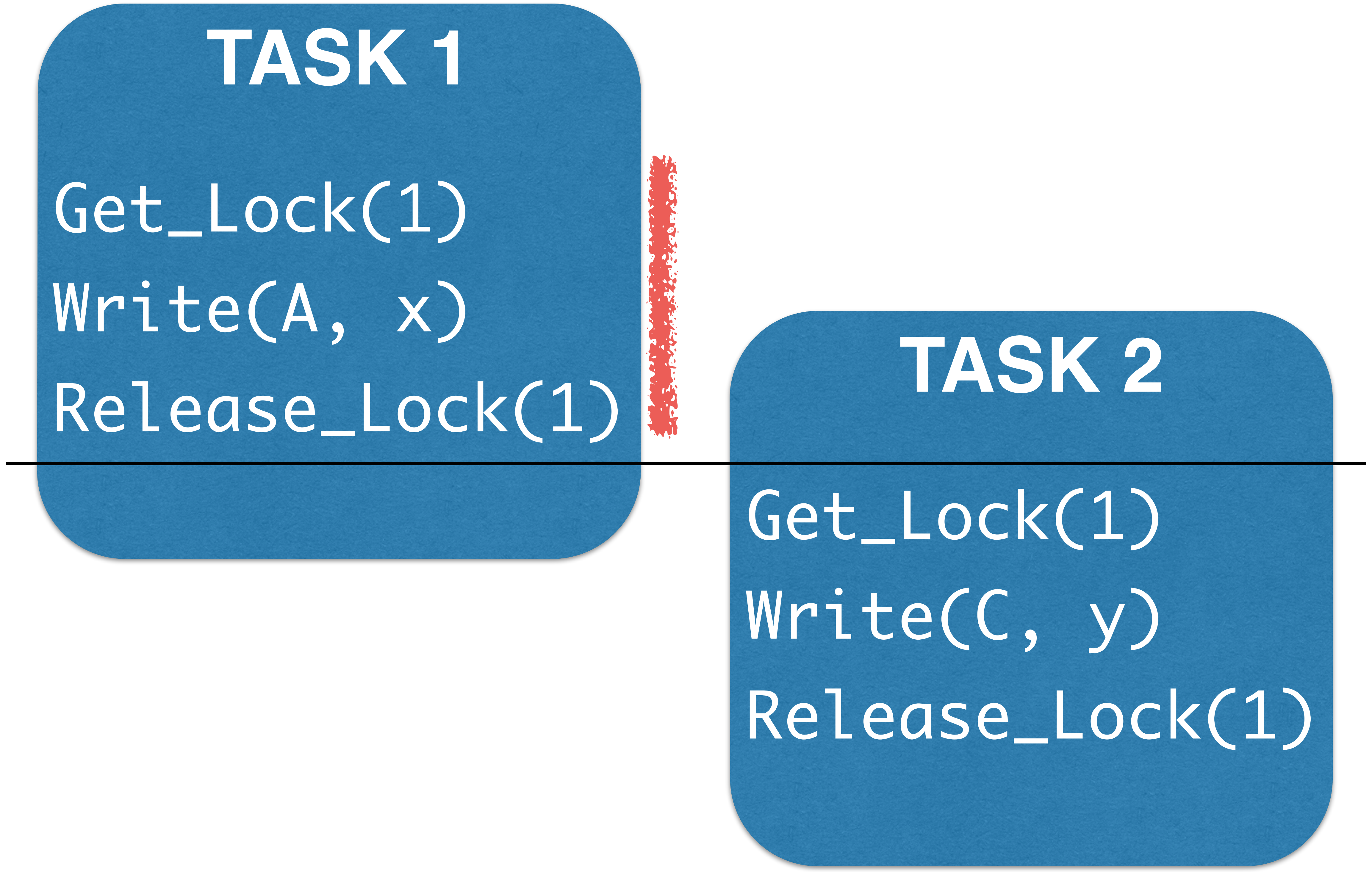
```
Get_Lock(1)
Write(A, x)
Release_Lock(1)
```

```
Get_Lock(1)
Write(C, y)
Release_Lock(1)
```

Locks

- Coarse-grained locking





TASK 1

```
Get_Lock(1)
Write(A, x)
Release_Lock(1)
```

TASK 2

```
Get_Lock(1)
Write(C, y)
Release_Lock(1)
```

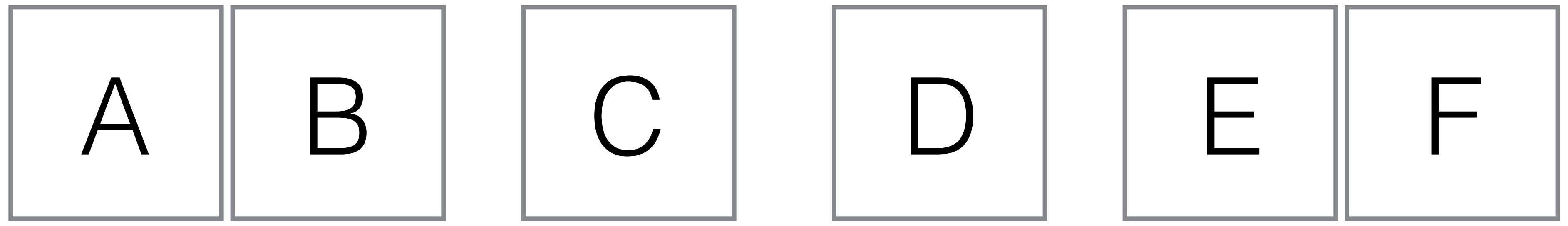
Critical sections can not progress in parallel!

Locks

- Fine-grained locking

Locks

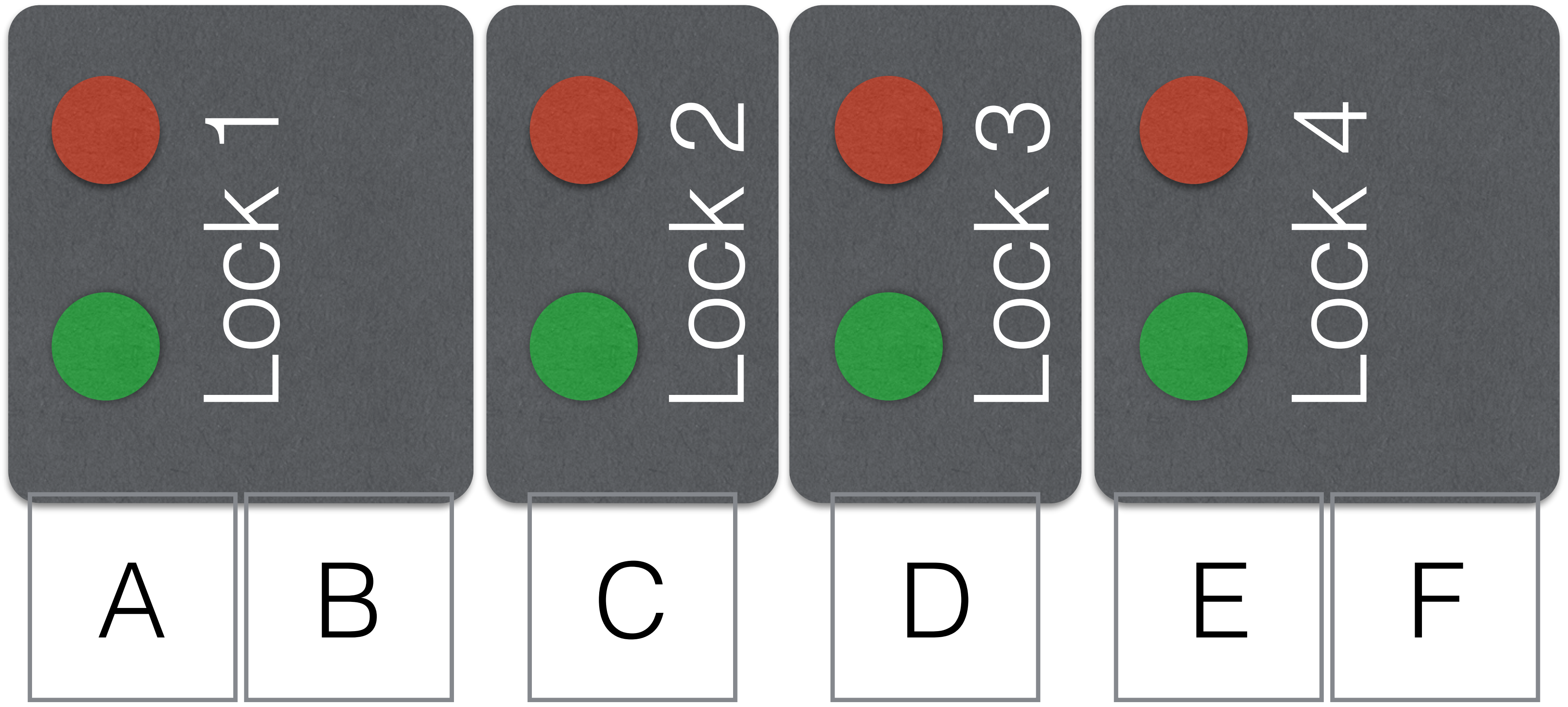
- Fine-grained locking



A B C D E F

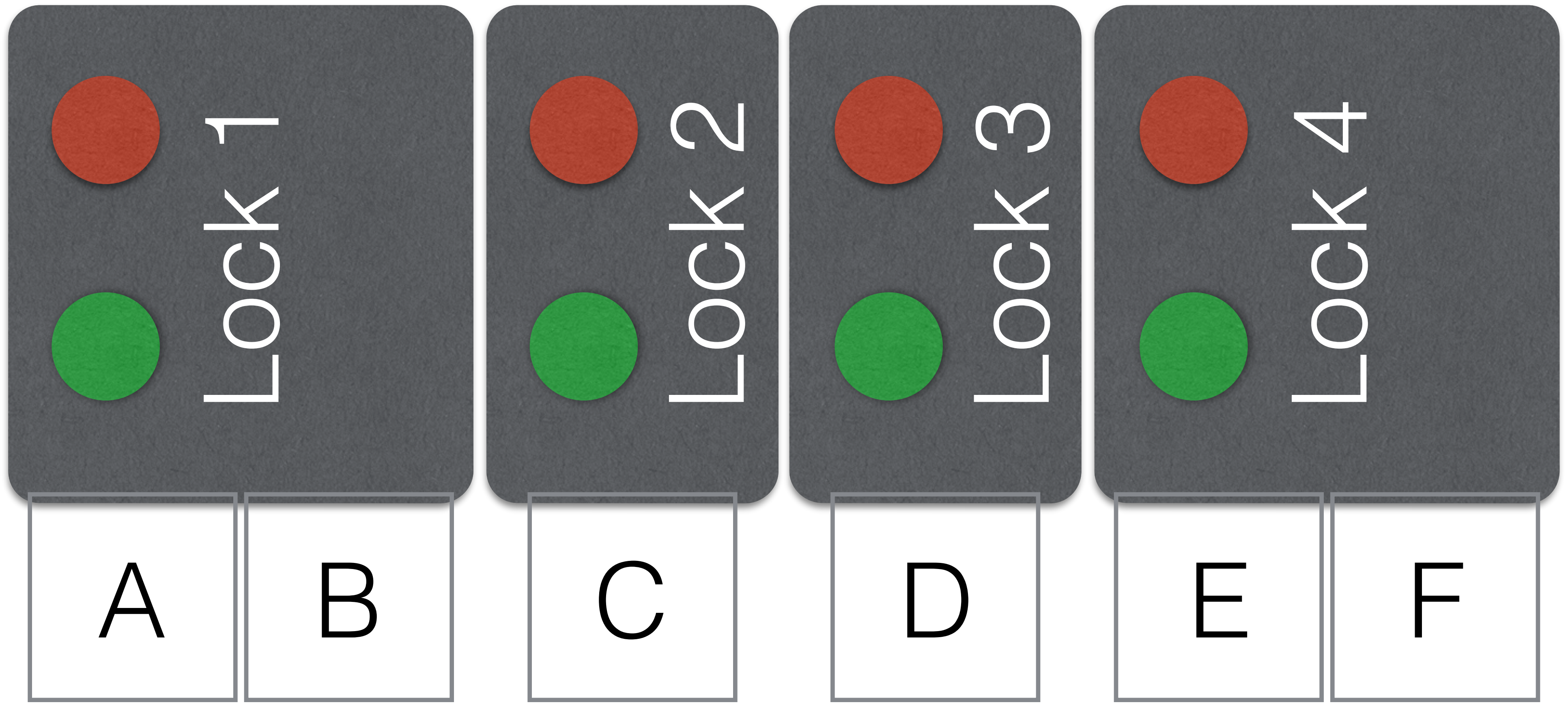
Locks

- Fine-grained locking



Locks

- Fine-grained locking

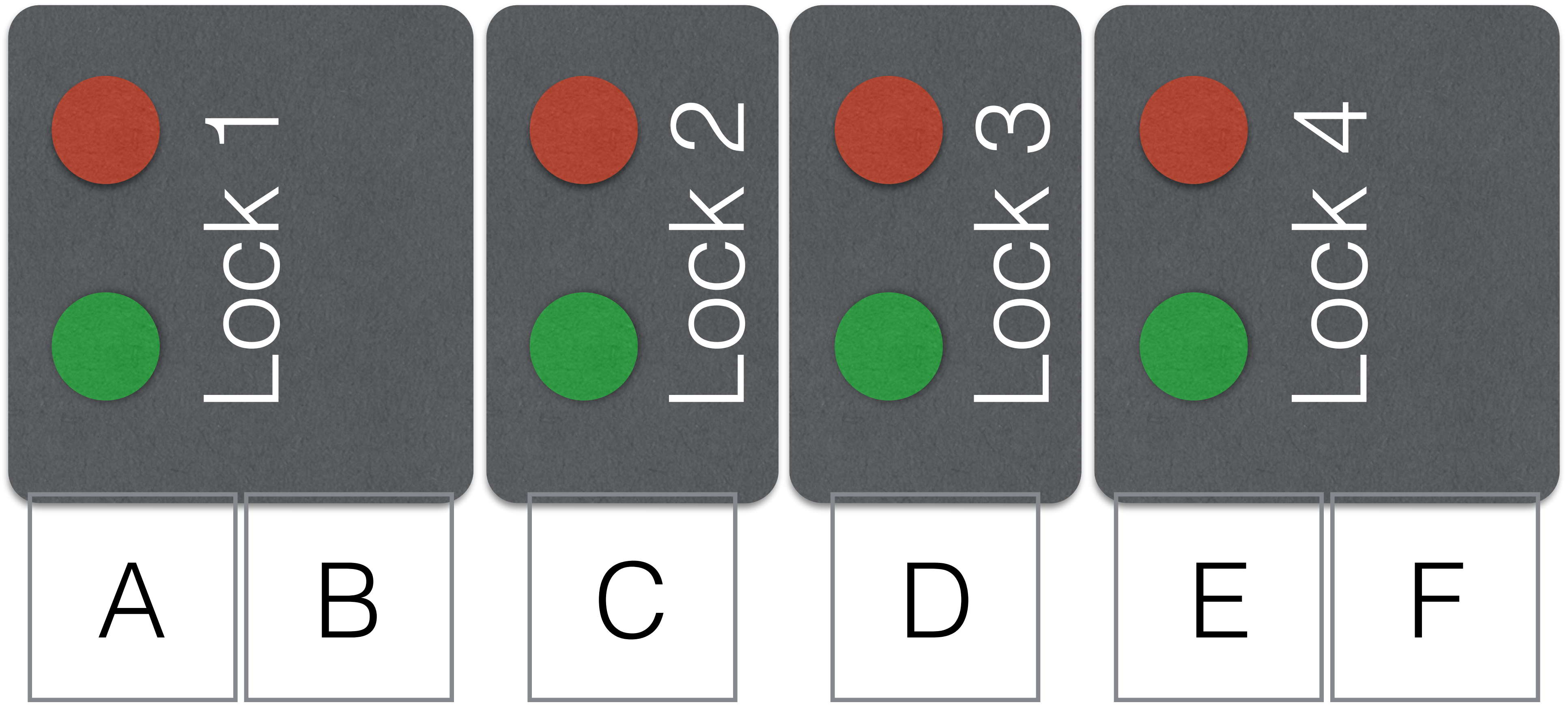


TASK 1

```
Get_Lock(1)
Get_Lock(3)
Read(C, x)
Write(A, x)
Release_Lock(3)
Release_Lock(1)
```

Locks

- Fine-grained locking



TASK 1

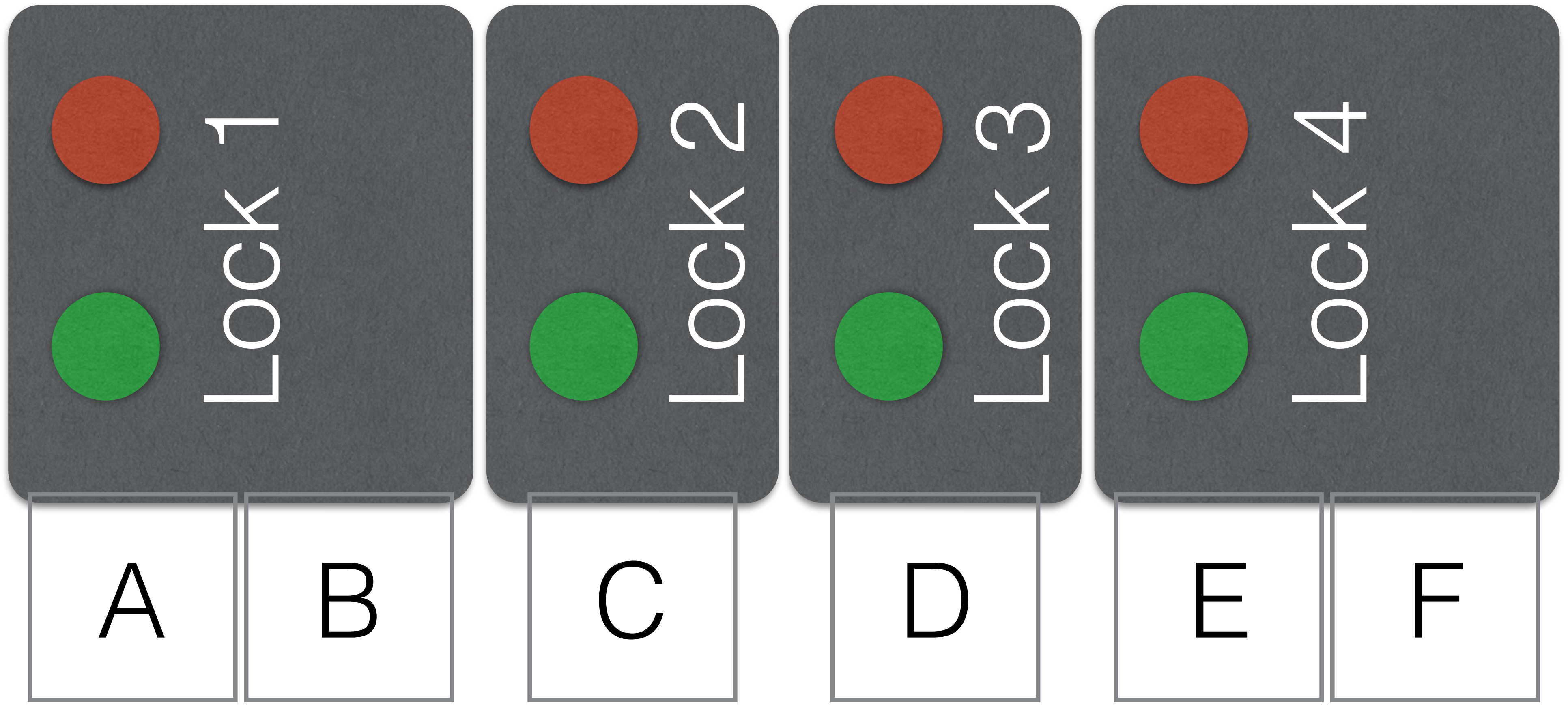
```
Get_Lock(1)
Get_Lock(3)
Read(C, x)
Write(A, x)
Release_Lock(3)
Release_Lock(1)
```

TASK 2

```
Get_Lock(3)
Get_Lock(1)
Read(B, x)
Write(C, y)
Release_Lock(1)
Release_Lock(3)
```

Locks

- Fine-grained locking



TASK 1

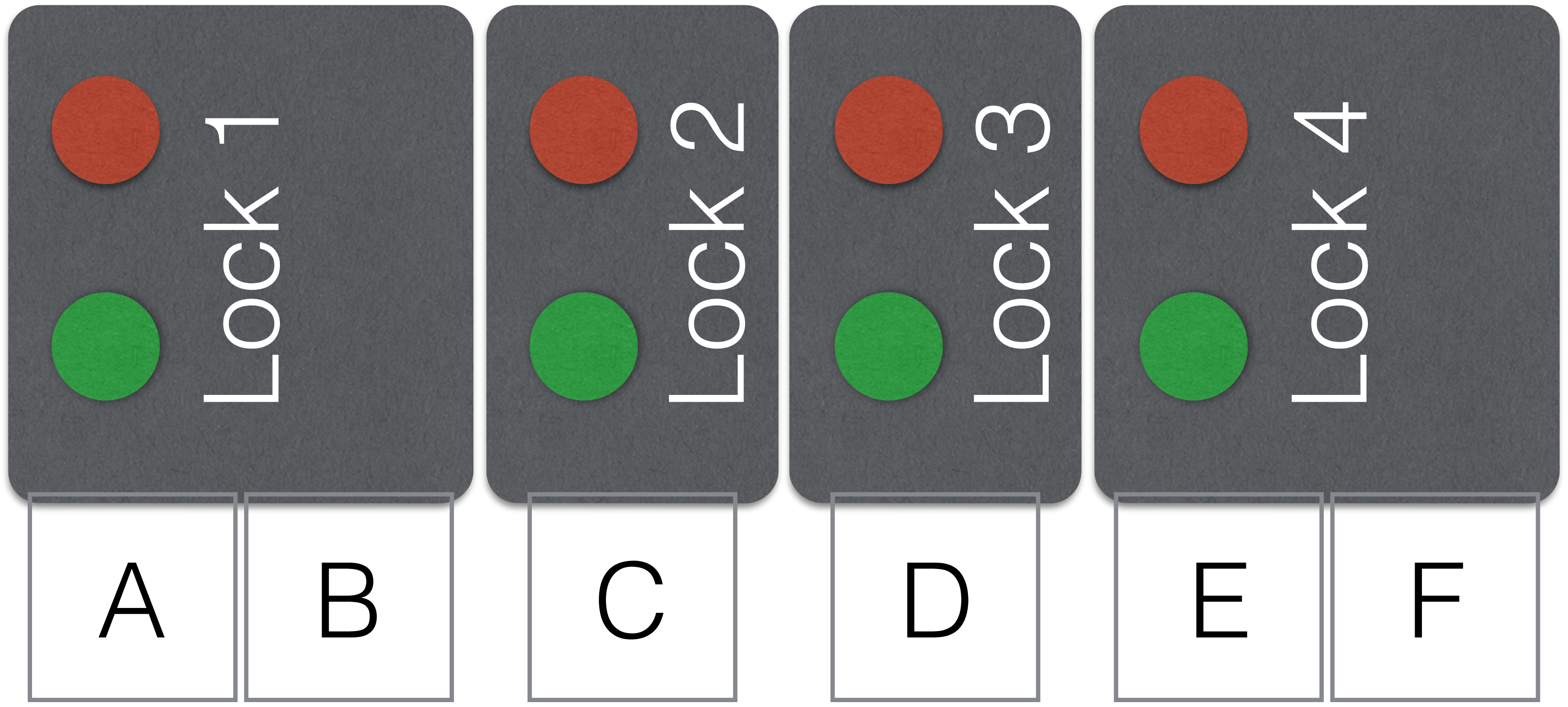
```
Get_Lock(1)
Get_Lock(3)
Read(C, x)
Write(A, x)
Release_Lock(3)
Release_Lock(1)
```

TASK 2

```
Get_Lock(3)
Get_Lock(1)
Read(B, x)
Write(C, y)
Release_Lock(1)
Release_Lock(3)
```

Locks

- Fine-grained locking



TASK 1

```
Get_Lock(1)
Get_Lock(3)
Read(C, x)
Write(A, x)
Release_Lock(3)
Release_Lock(1)
```

TASK 2

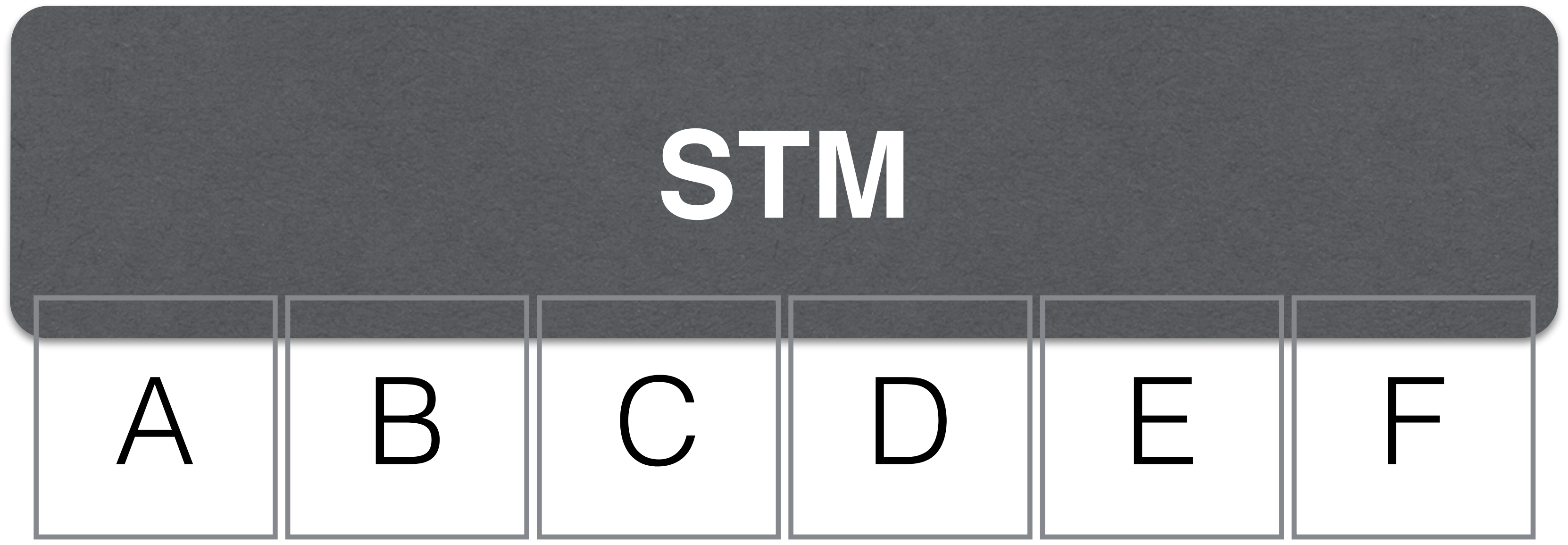
```
Get_Lock(3)
Get_Lock(1)
Read(B, x)
Write(C, y)
Release_Lock(1)
Release_Lock(3)
```

Increases system complexity with a negative impact on composability and maintainability!

Software Transactional Memory

A	B	C	D	E	F

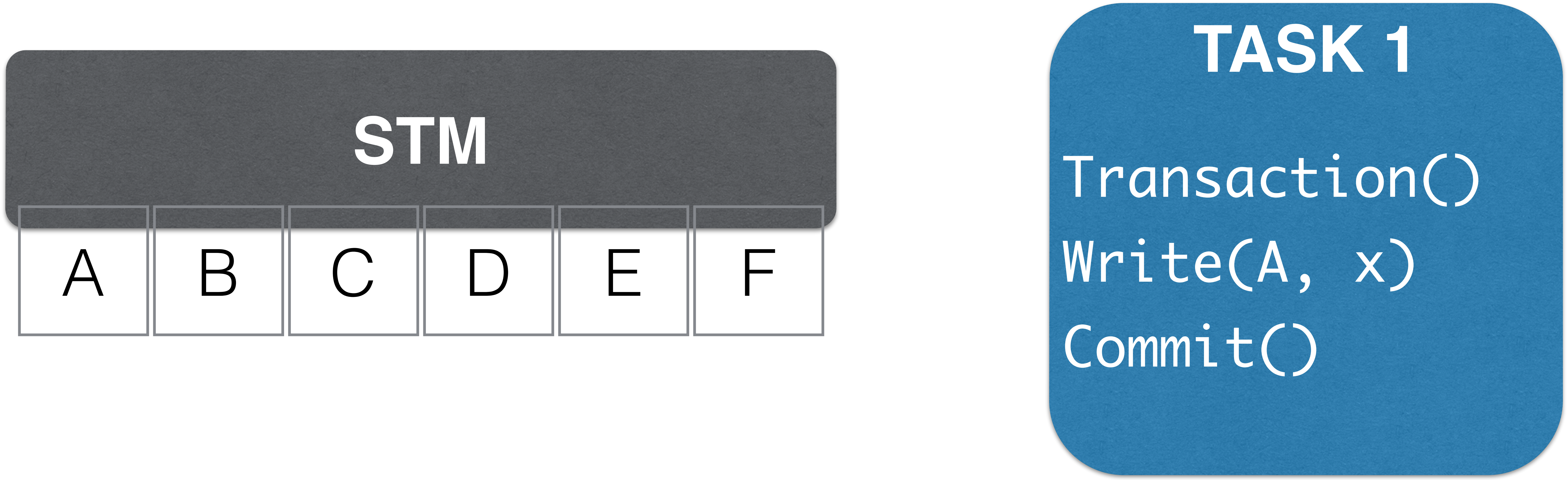
Software Transactional Memory



STM

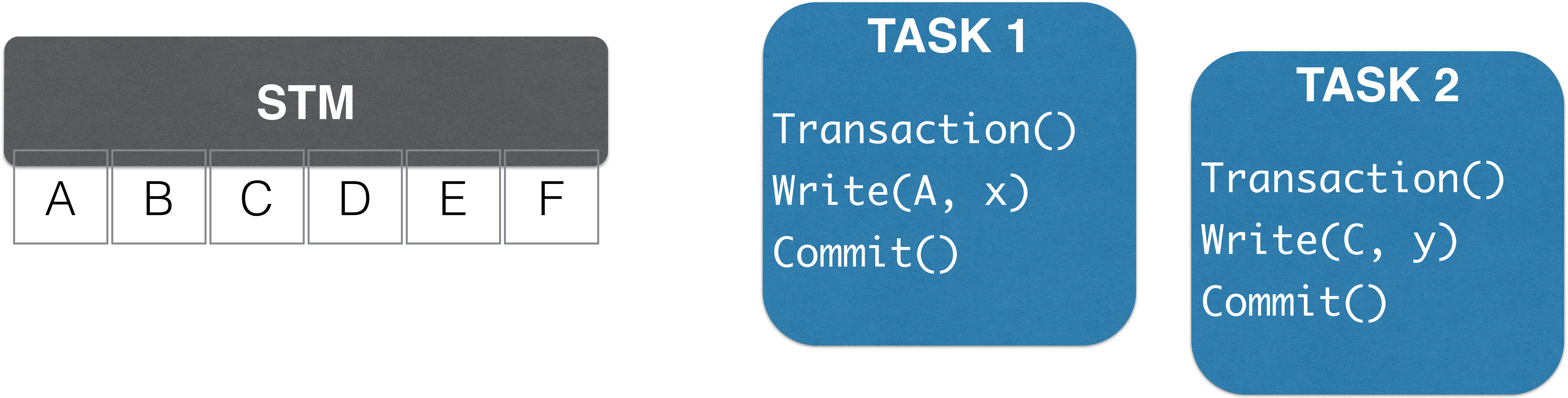
A B C D E F

Software Transactional Memory



```
Transaction()
Write(A, x)
Commit()
```

Software Transactional Memory



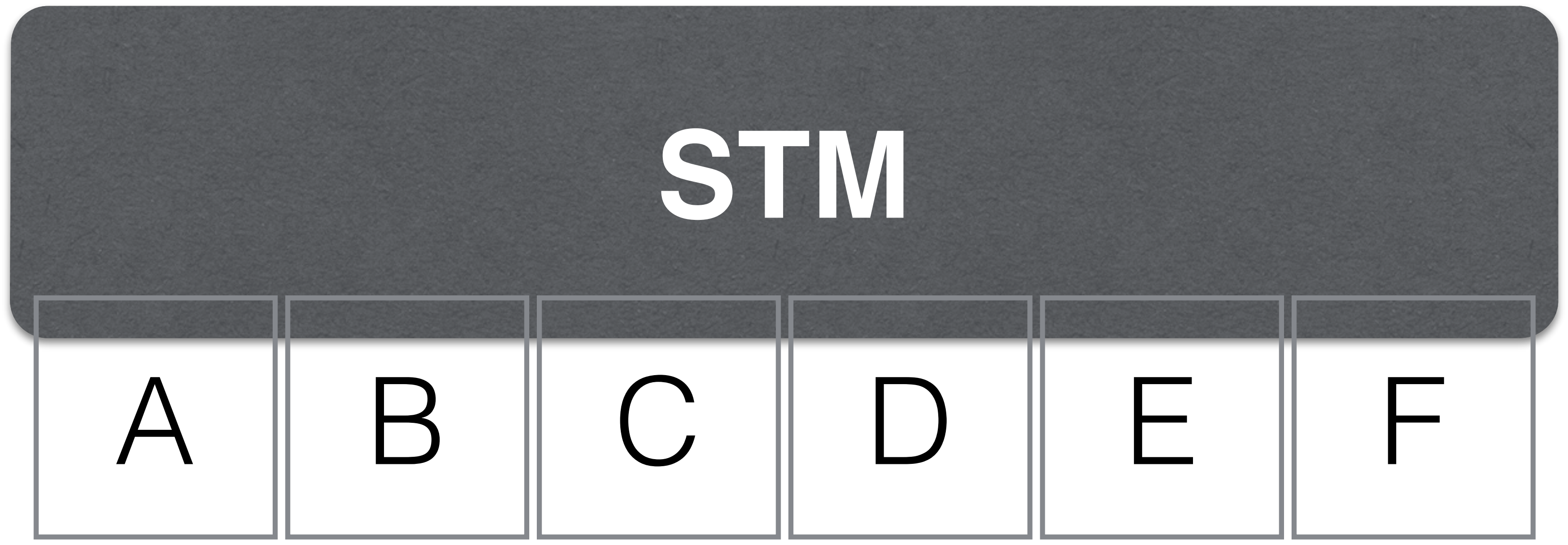
TASK 1

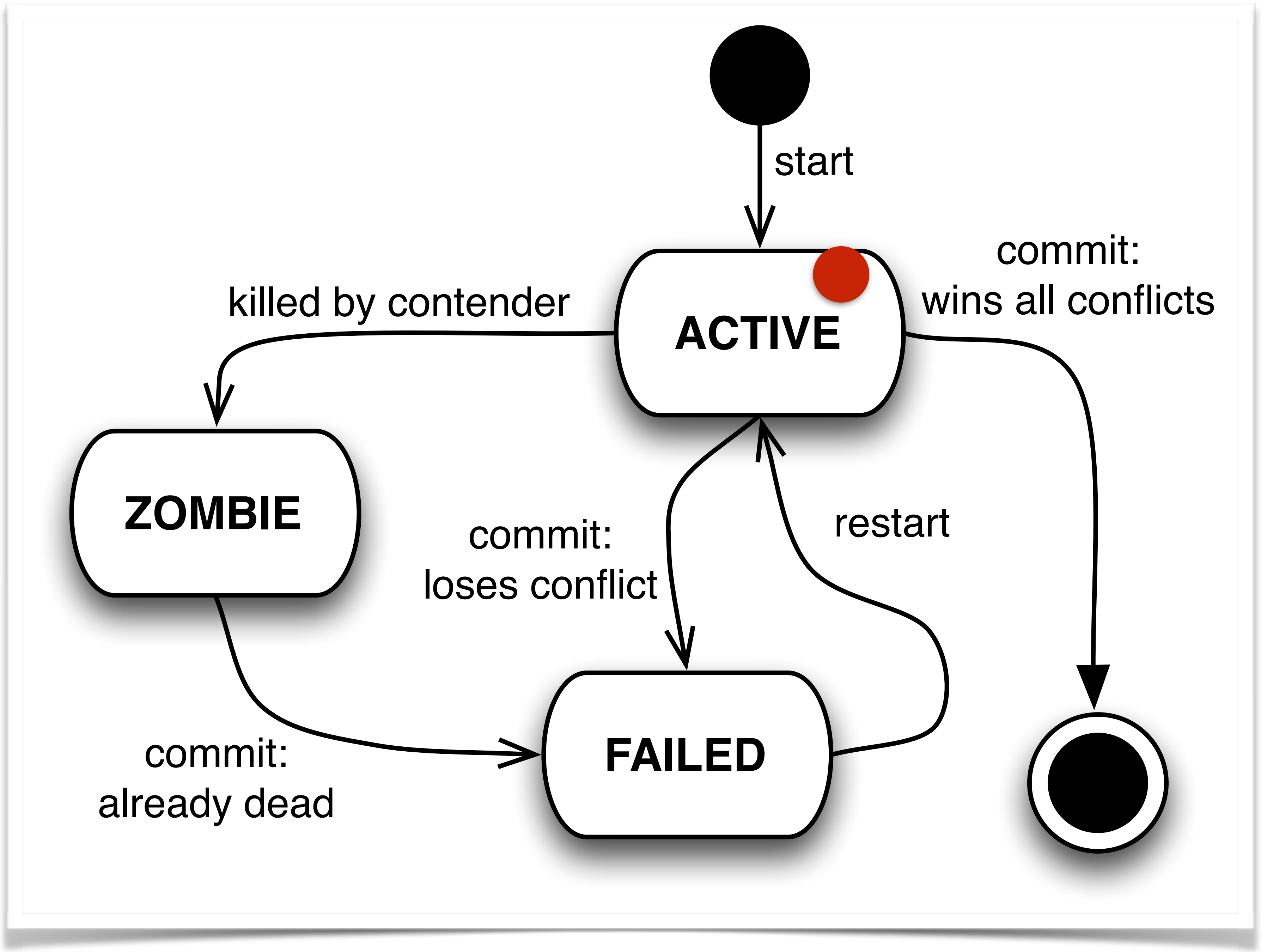
```
Transaction()
Write(A, x)
Commit()
```

TASK 2

```
Transaction()
Write(C, y)
Commit()
```

Software Transactional Memory





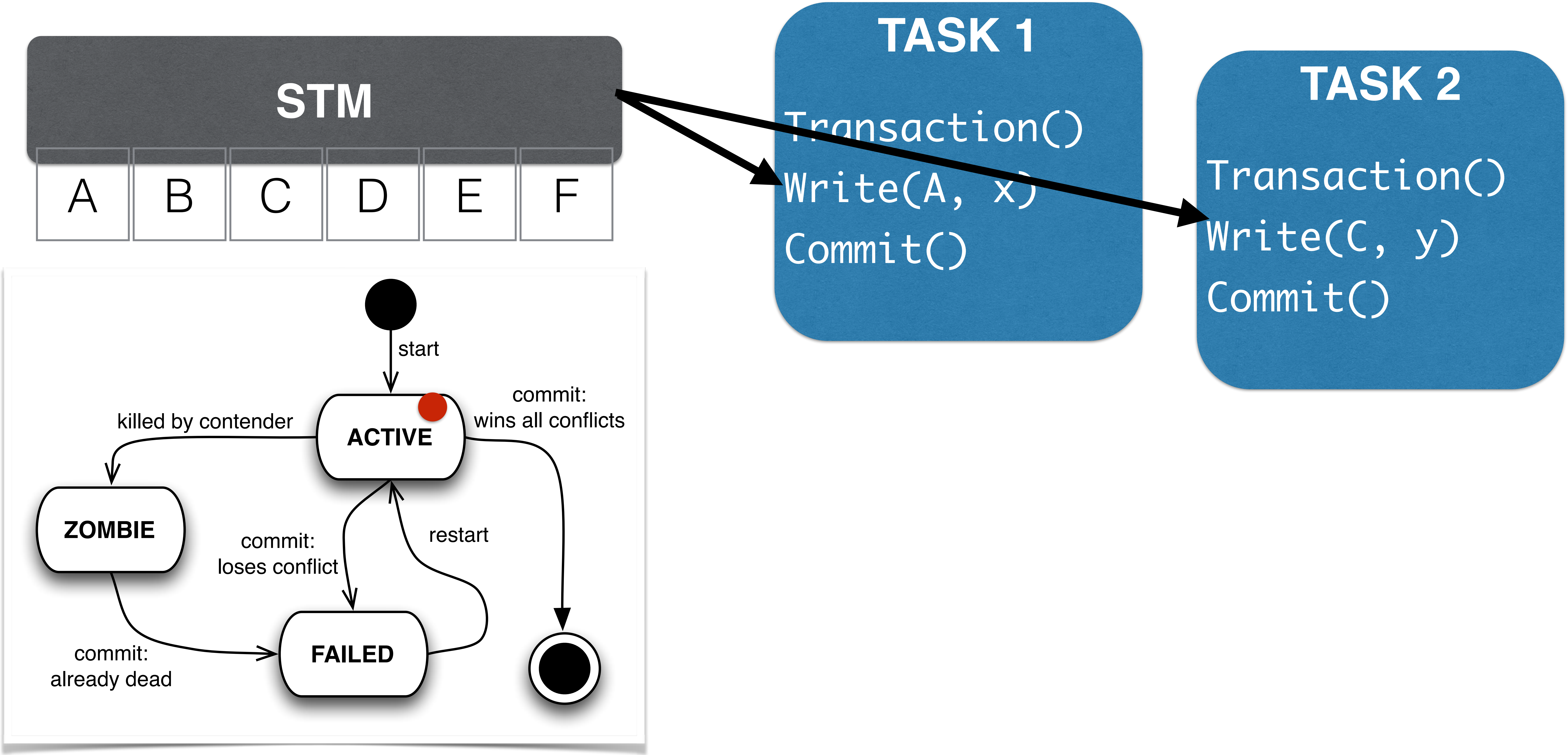
TASK 1

```
Transaction()
Write(A, x)
Commit()
```

TASK 2

```
Transaction()
Write(C, y)
Commit()
```

Software Transactional Memory



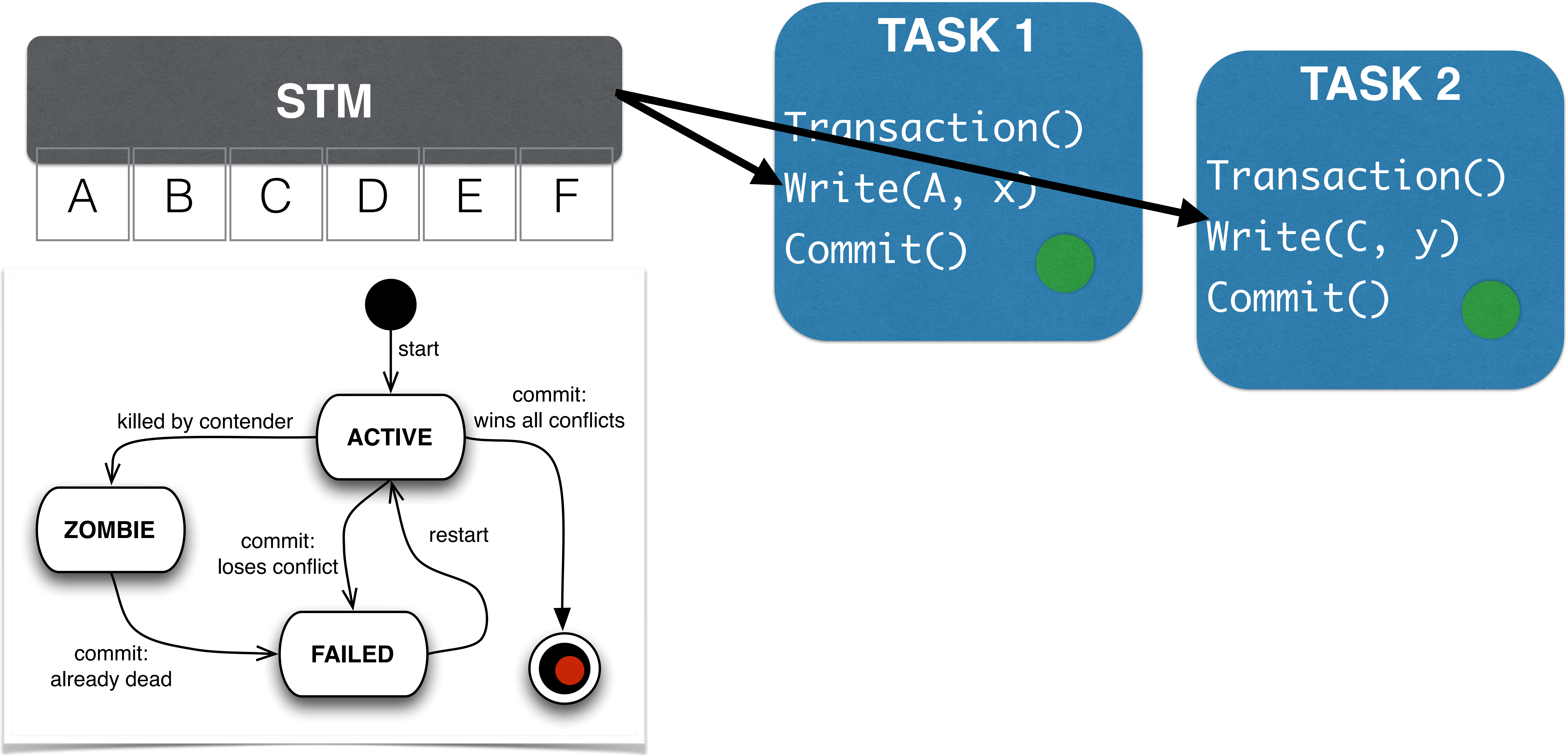
TASK 1

```
Transaction()
Write(A, x)
Commit()
```

TASK 2

```
Transaction()
Write(C, y)
Commit()
```

Software Transactional Memory



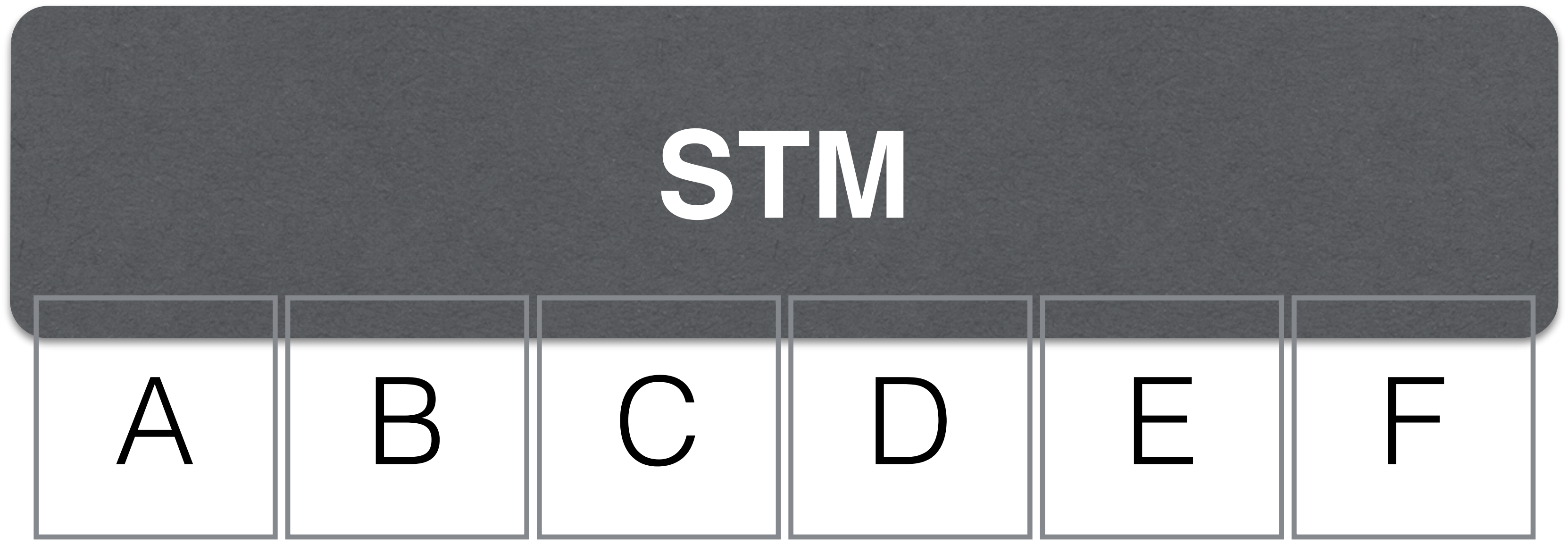
TASK 1

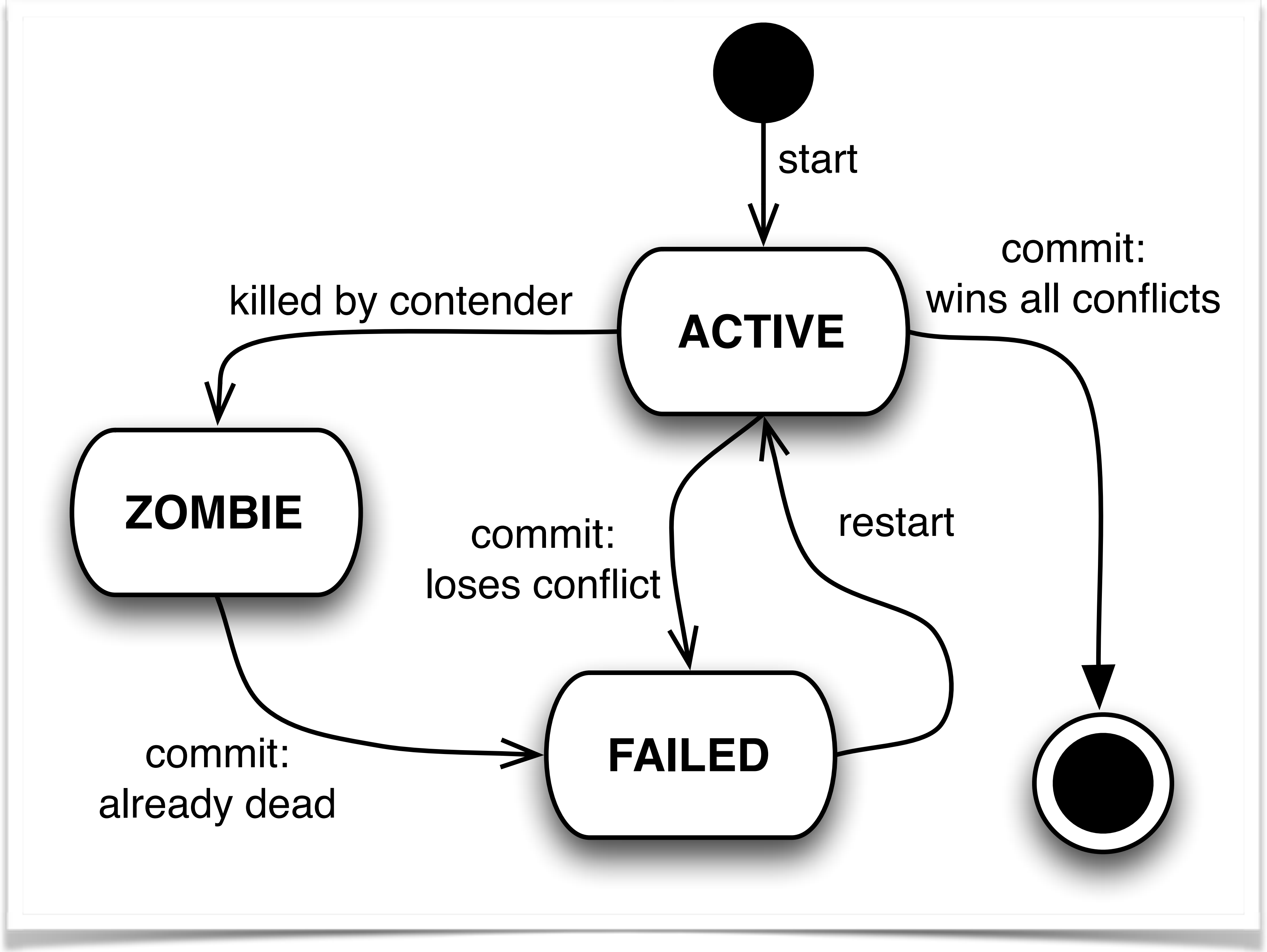
```
Transaction()
Write(A, x)
Commit()
```

TASK 2

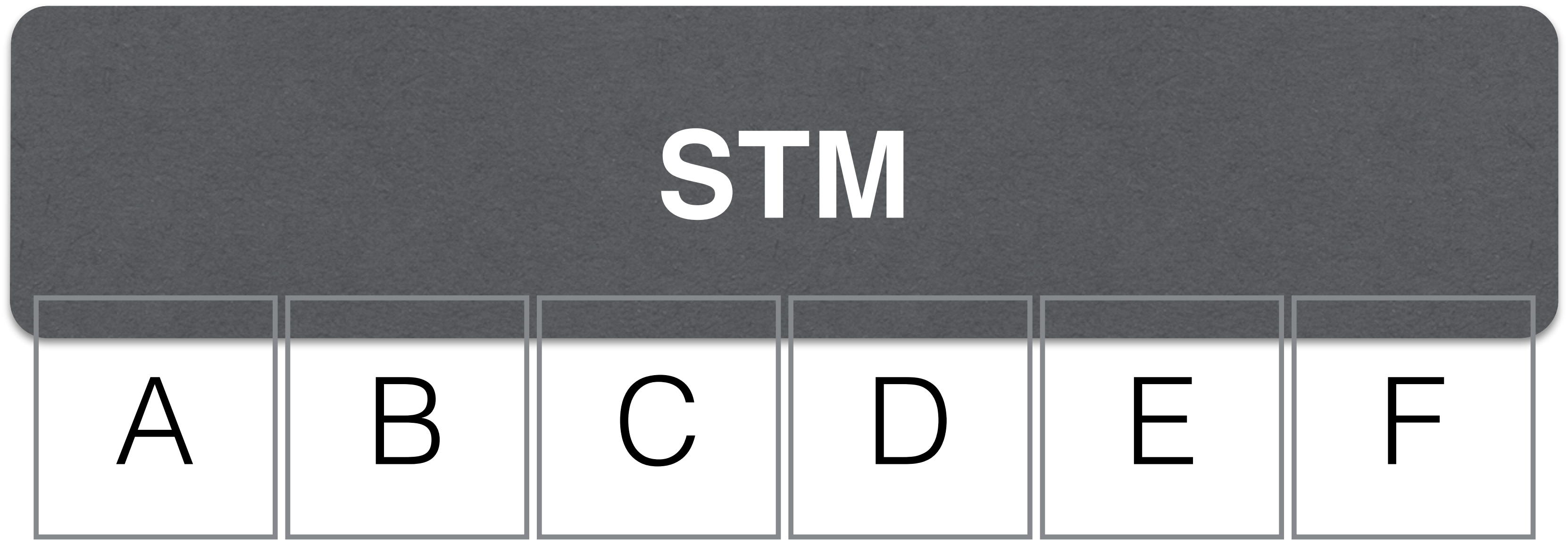
```
Transaction()
Write(C, y)
Commit()
```

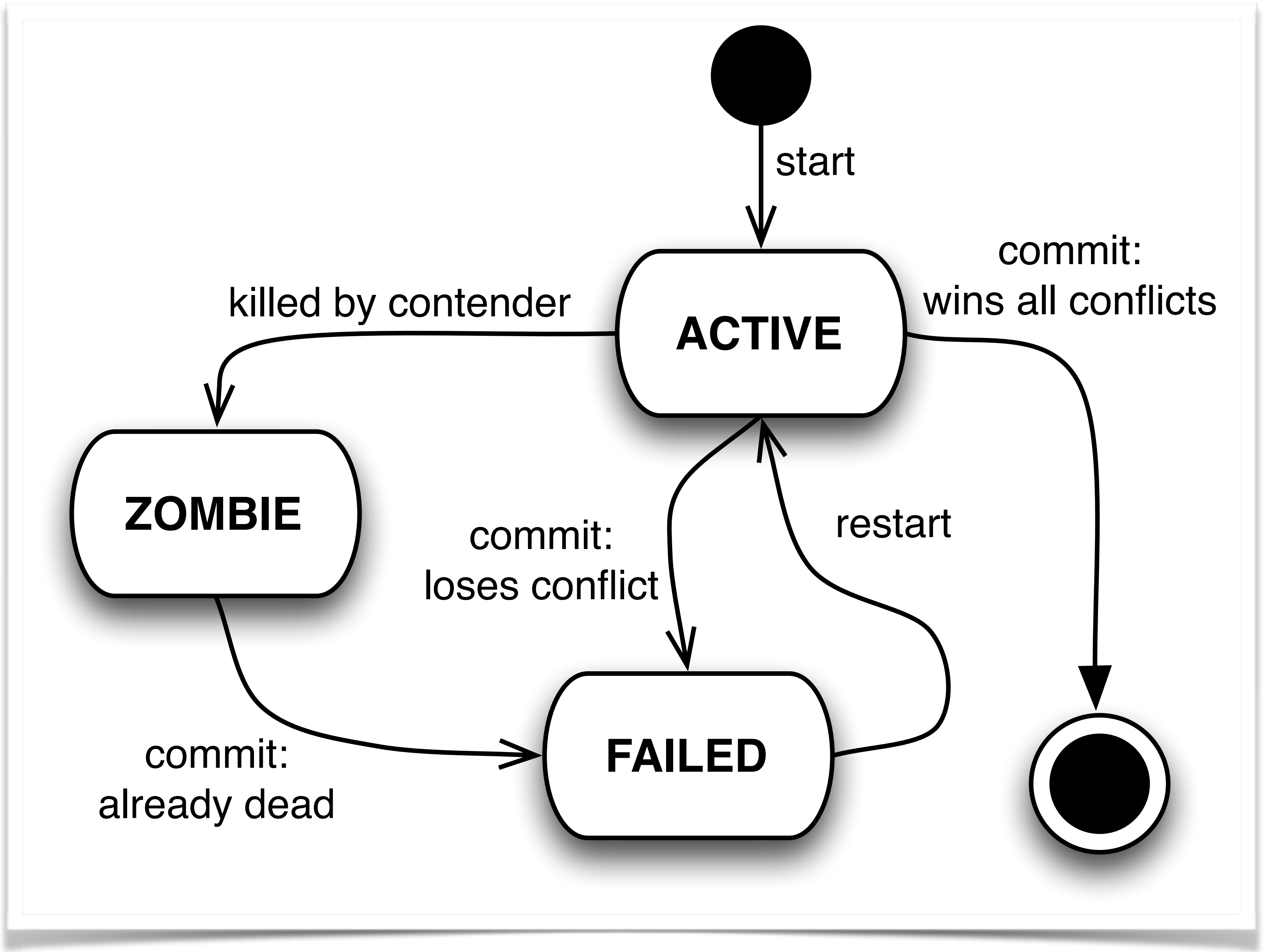
Software Transactional Memory





Software Transactional Memory





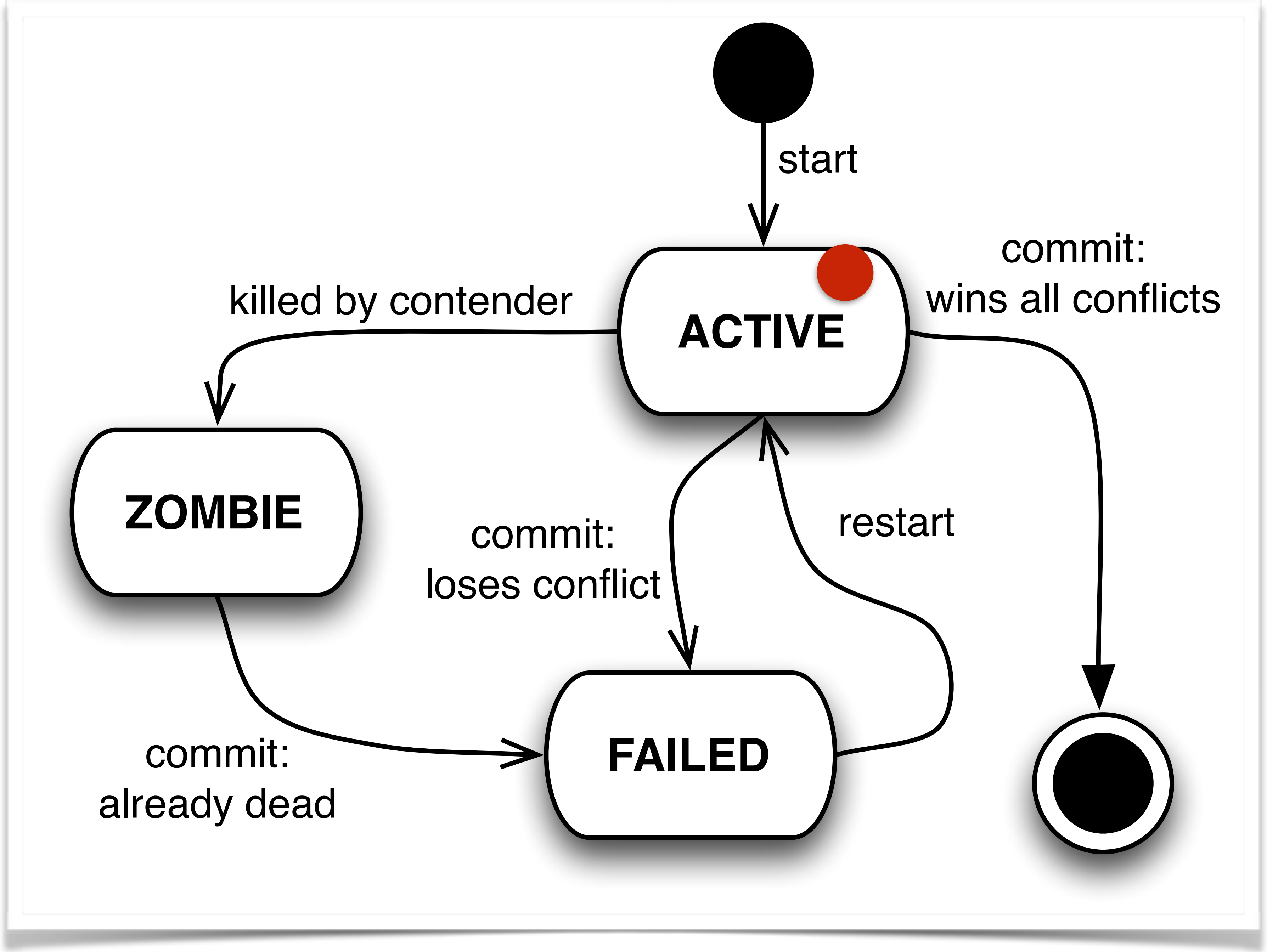
TASK 1

```
Transaction()
Write(E, x)
Commit()
```

Software Transactional Memory

STM

A	B	C	D	E	F



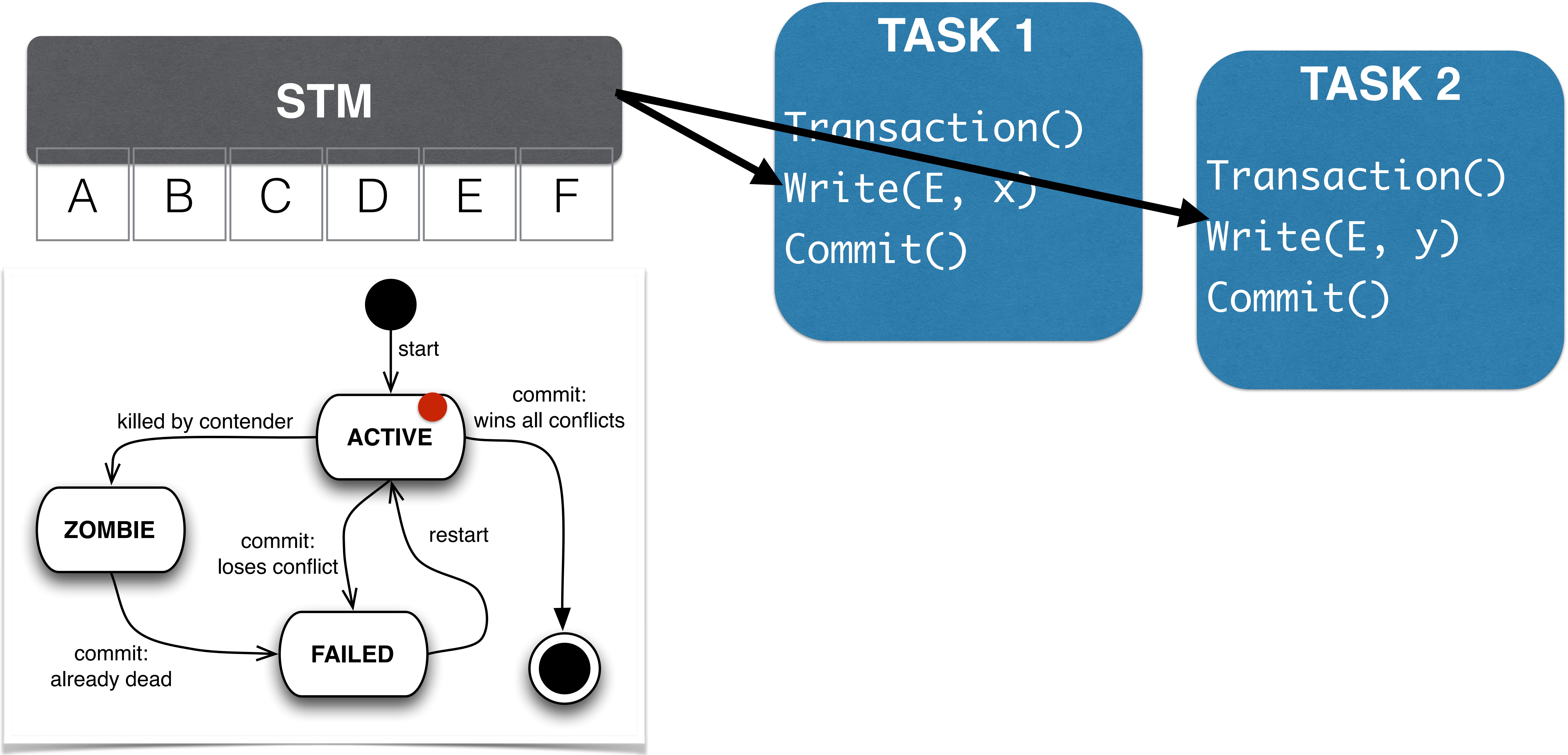
TASK 1

```
Transaction()
Write(E, x)
Commit()
```

TASK 2

```
Transaction()
Write(E, y)
Commit()
```

Software Transactional Memory



STM

A B C D E F

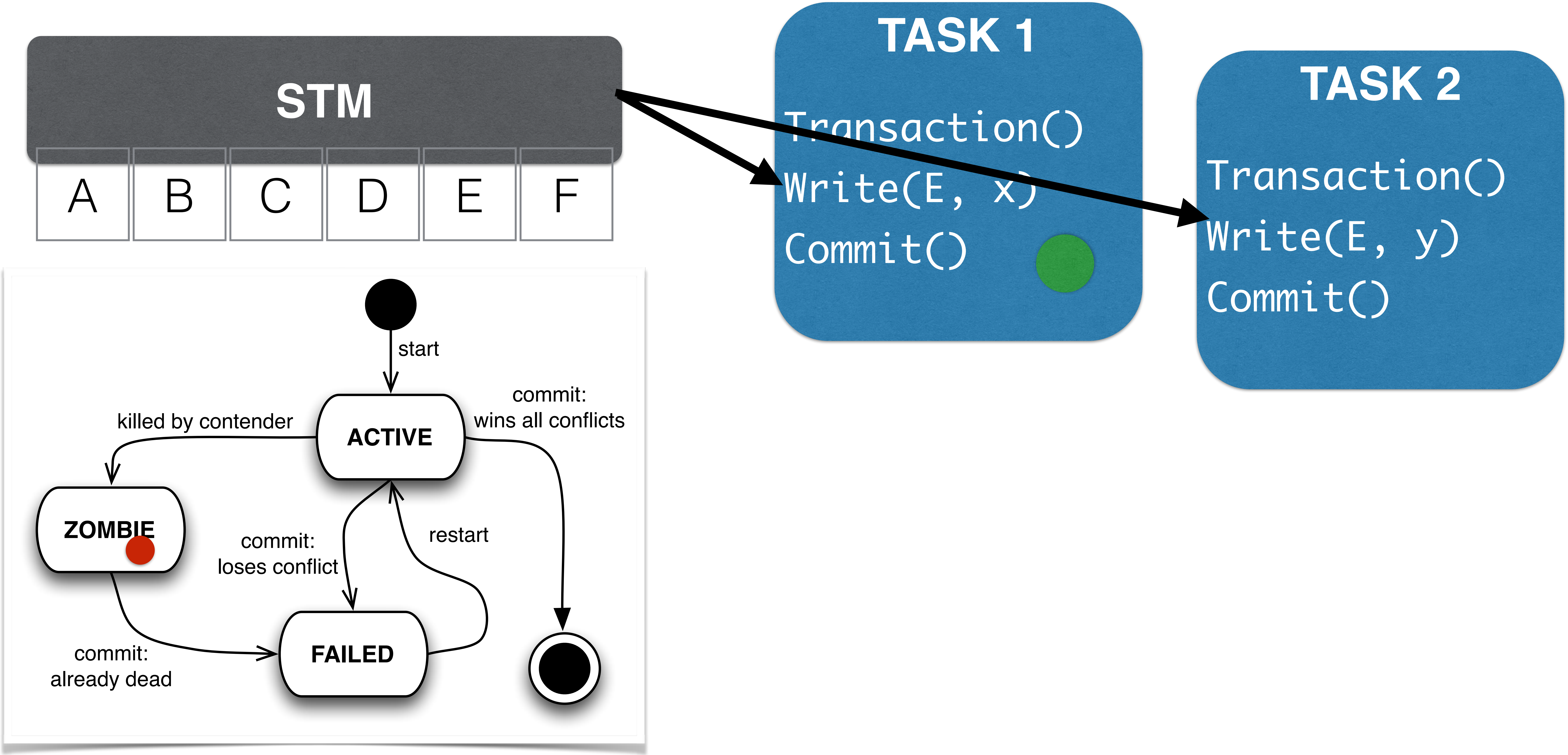
TASK 1

```
Transaction()
Write(E, x)
Commit()
```

TASK 2

```
Transaction()
Write(E, y)
Commit()
```

Software Transactional Memory



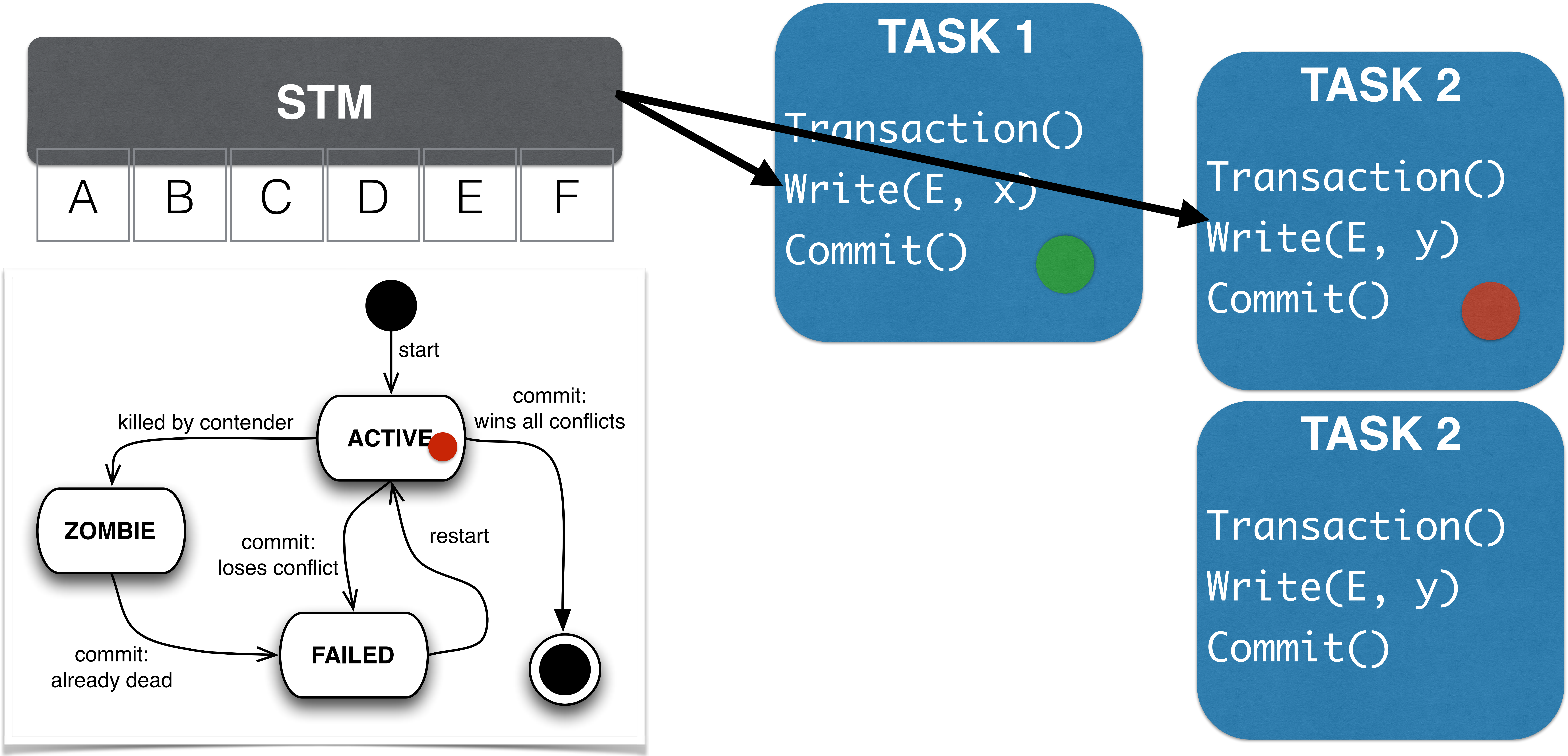
TASK 1

```
Transaction()
Write(E, x)
Commit()
```

TASK 2

```
Transaction()
Write(E, y)
Commit()
```

Software Transactional Memory



STM

A	B	C	D	E	F

TASK 1

```
Transaction()
Write(E, x)
Commit()
```

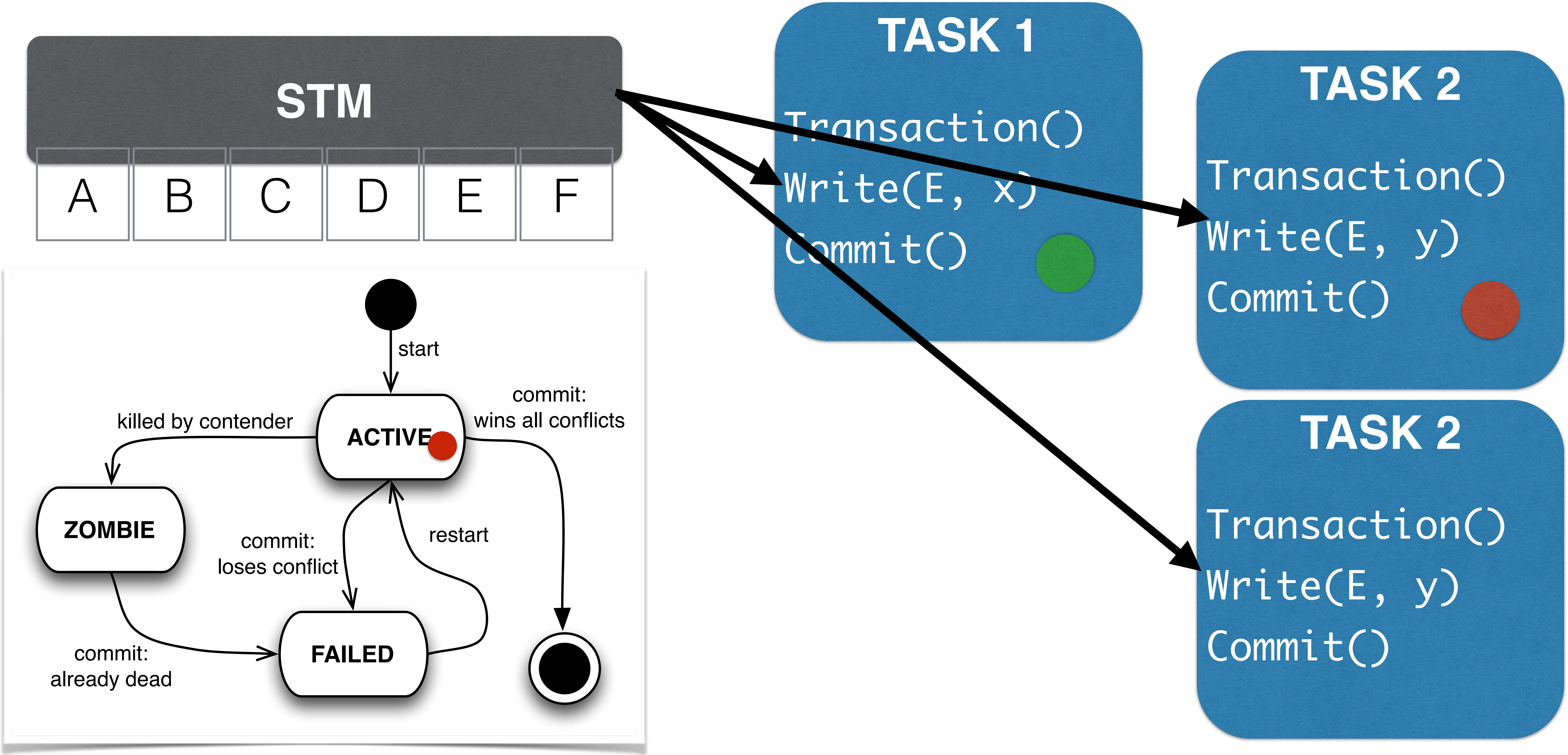
TASK 2

```
Transaction()
Write(E, y)
Commit()
```

TASK 2

```
Transaction()
Write(E, y)
Commit()
```

Software Transactional Memory



TASK 1

```
Transaction()
Write(E, x)
Commit()
```

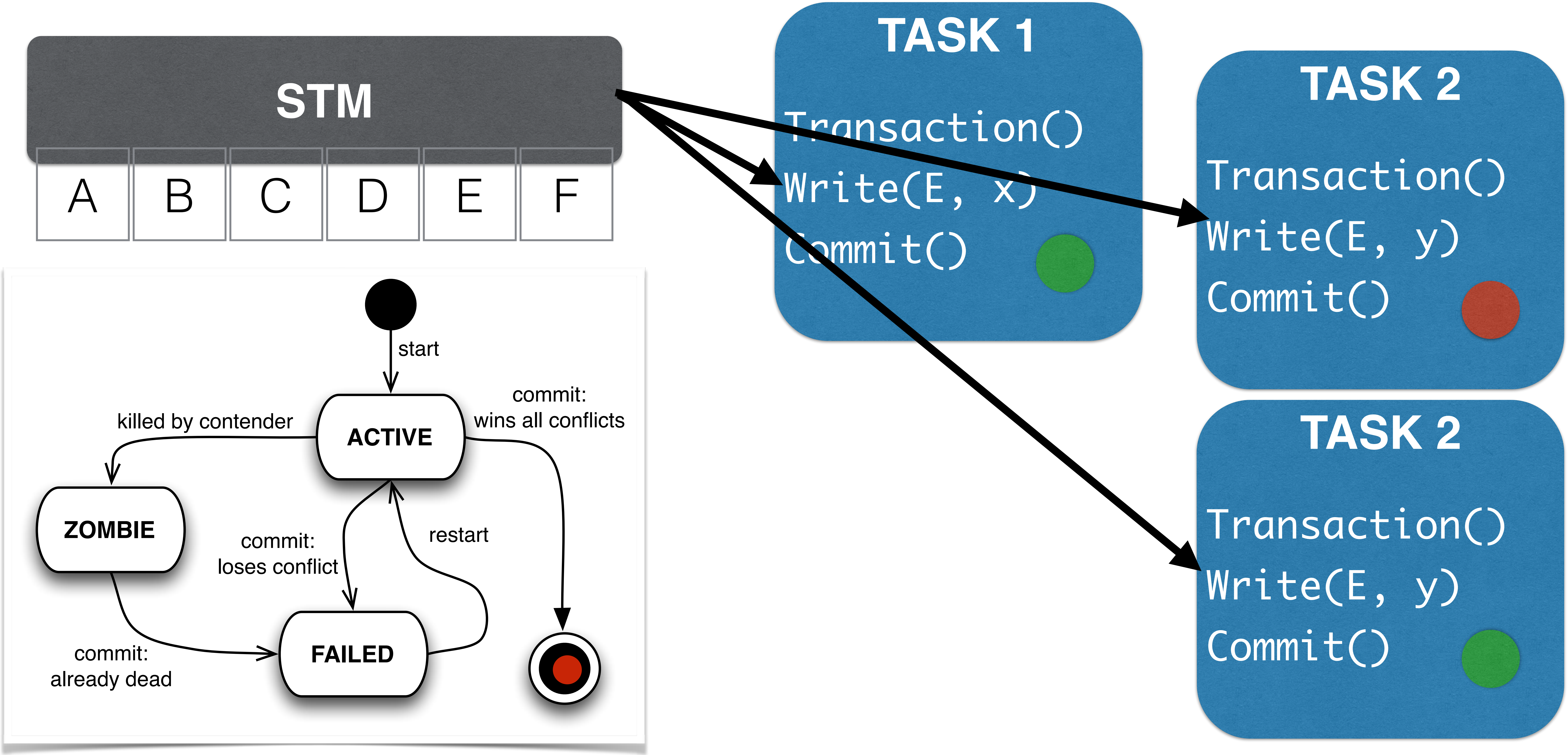
TASK 2

```
Transaction()
Write(E, y)
Commit()
```

TASK 2

```
Transaction()
Write(E, y)
Commit()
```

Software Transactional Memory



STM

A	B	C	D	E	F

TASK 1

```
Transaction()
Write(E, x)
Commit()
```

TASK 2

```
Transaction()
Write(E, y)
Commit()
```

TASK 2

```
Transaction()
Write(E, y)
Commit()
```

Managing contention

Managing contention

Polite
Exponential back-off,
eventually commit.

Managing contention

Aggressive
Kill the enemy!!!

Managing contention

Polite
Exponential back-off, eventually commit.

Aggressive
Kill the enemy!!!

Randomized
Abort with p or Wait with (1-p).

Managing contention

Timestamp
Older transaction survives.

gressive
e enemy!!!

Managing contention

Polite
Exponential back-off, eventually commit.

Aggressive
Kill the enemy!!!

Randomized
Abort with p or Wait with (1-p).

Karma
Accesses and aborts accounts for karma.

Managing contention

Aggressive
Kill the enemy!!!

omized
vith p or
th (1-p).

Eruption
Priority rises if others are waiting.

Managing contention

DETERMINISTIC

Timestamp
Older transaction survives.

NOT DETERMINISTIC

Polite
Exponential back-off, eventually commit.

Aggressive
Kill the enemy!!!

Randomized
Abort with p or Wait with (1-p).

Karma
Accesses and aborts accounts for karma.

Eruption
Priority rises if others are waiting.

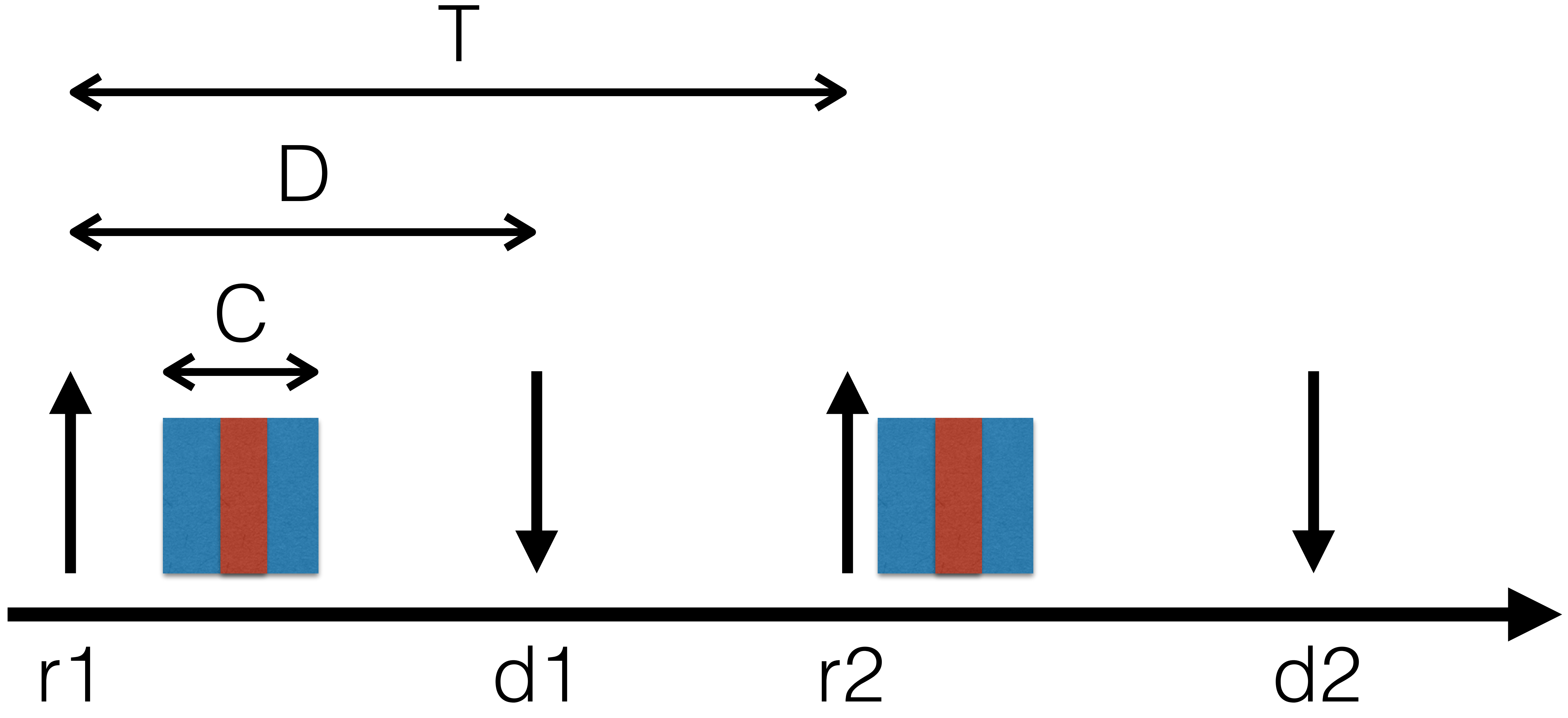
Model of computation and scheduling strategy

Computation platform

- Multi-core
- Single memory bus
- Data shared in globally accessed memory, controlled by a STM system

Application characteristics

- Application functionality divided into tasks.
- Each task is statically assigned to a core, before run-time.
- Each task releases a potentially infinite number of jobs.
  - Task: C (execution time), T (period), D (deadline)
  - Job: r (release time), d (absolute deadline)



Application characteristics

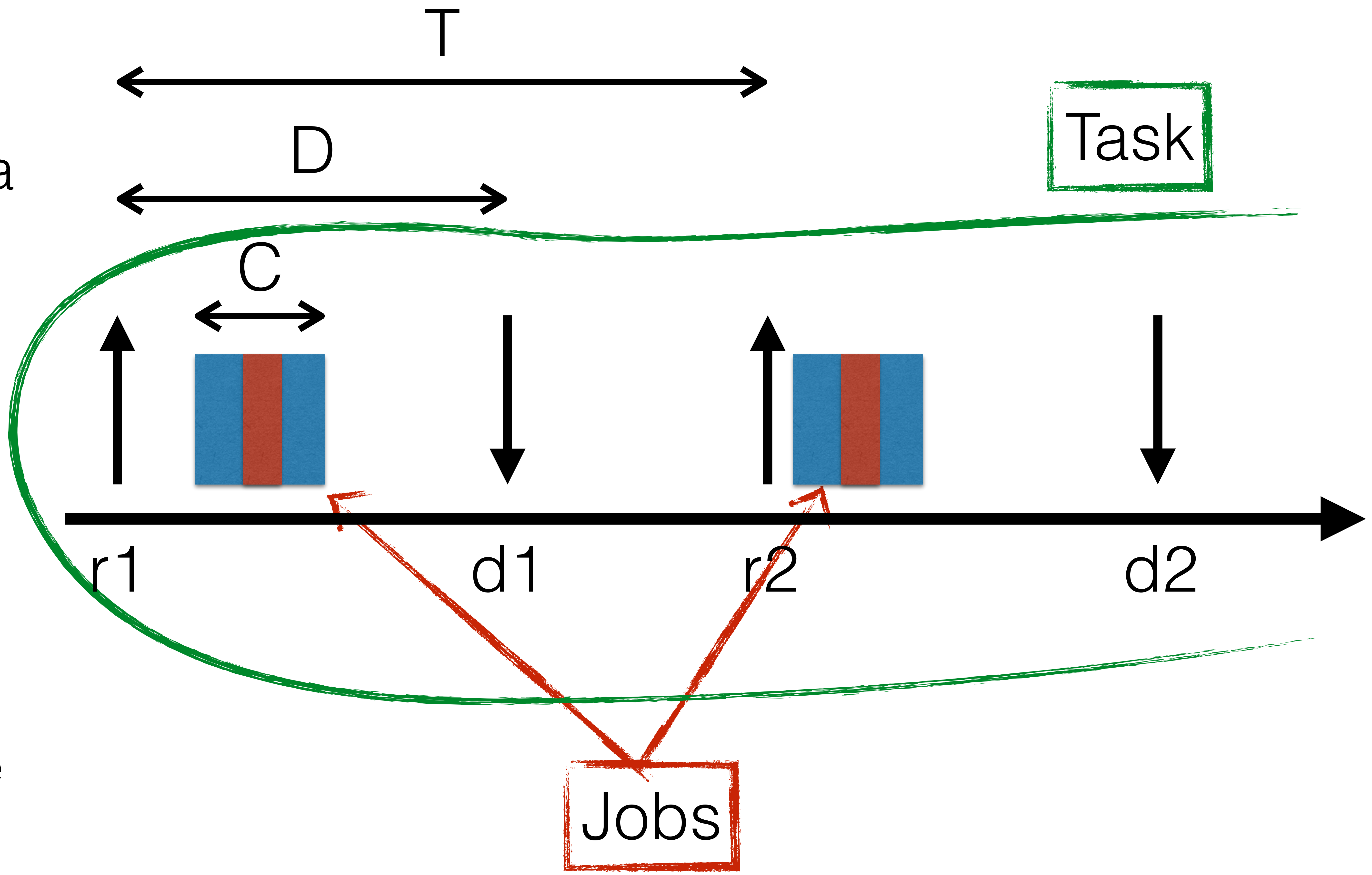
- Application functionality divided into tasks.

- Each task is statically assigned to a core, before run-time.

- Each task releases a potentially infinite number of jobs.

  - Task: C (execution time), T (period), D (deadline)

  - Job: r (release time), d (absolute deadline)



Serialisation of transactions in a RT environment

FIFO serialisation of transactions

FIFO serialisation of transactions

Problem solved!

- The order of serialisation of transactions in progress is determined once a transaction starts!

FIFO serialisation of transactions

Problem solved!

- The order of serialisation of transactions in progress is determined once a transaction starts!

… or maybe not!

FIFO serialisation of transactions

Problem solved!

- The order of serialisation of transactions in progress is determined once a transaction starts!

… or maybe not!

- What if jobs can be preempted while executing a transaction?

FIFO serialisation of transactions

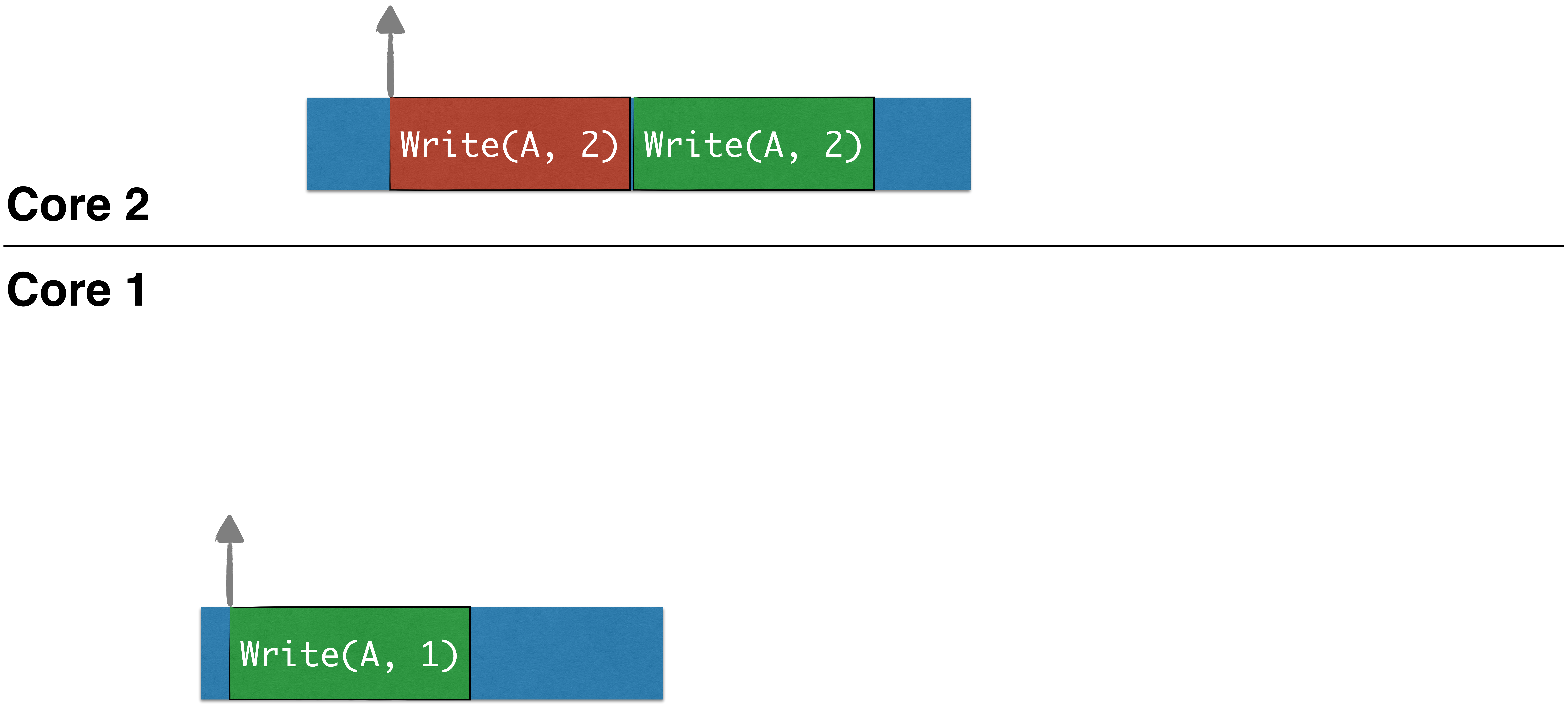
Problem solved!

- The order of serialisation of transactions in progress is determined once a transaction starts!

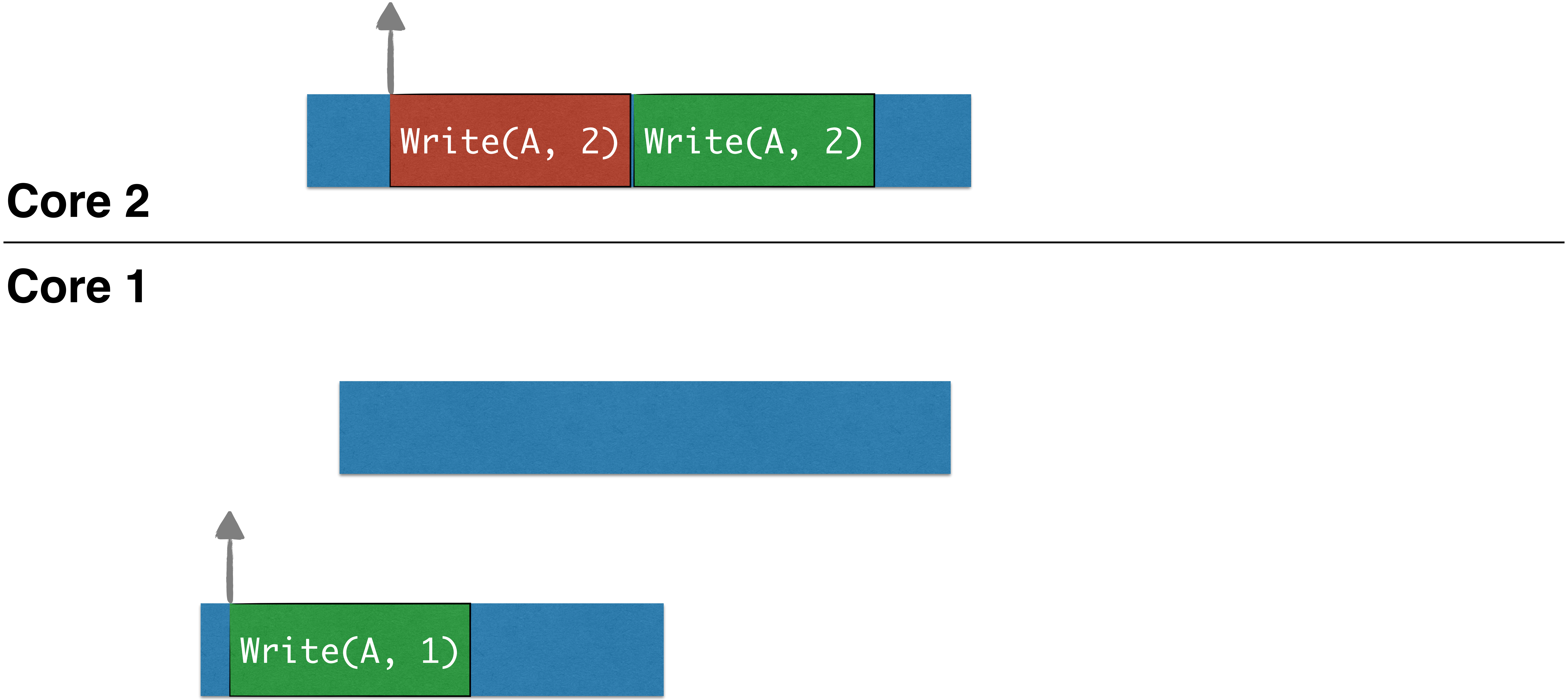
… or maybe not!

- What if jobs can be preempted while executing a transaction?
- What if multiple transactions can be simultaneously in progress on the same core?

Preemptions and serialisation



Preemptions and serialisation



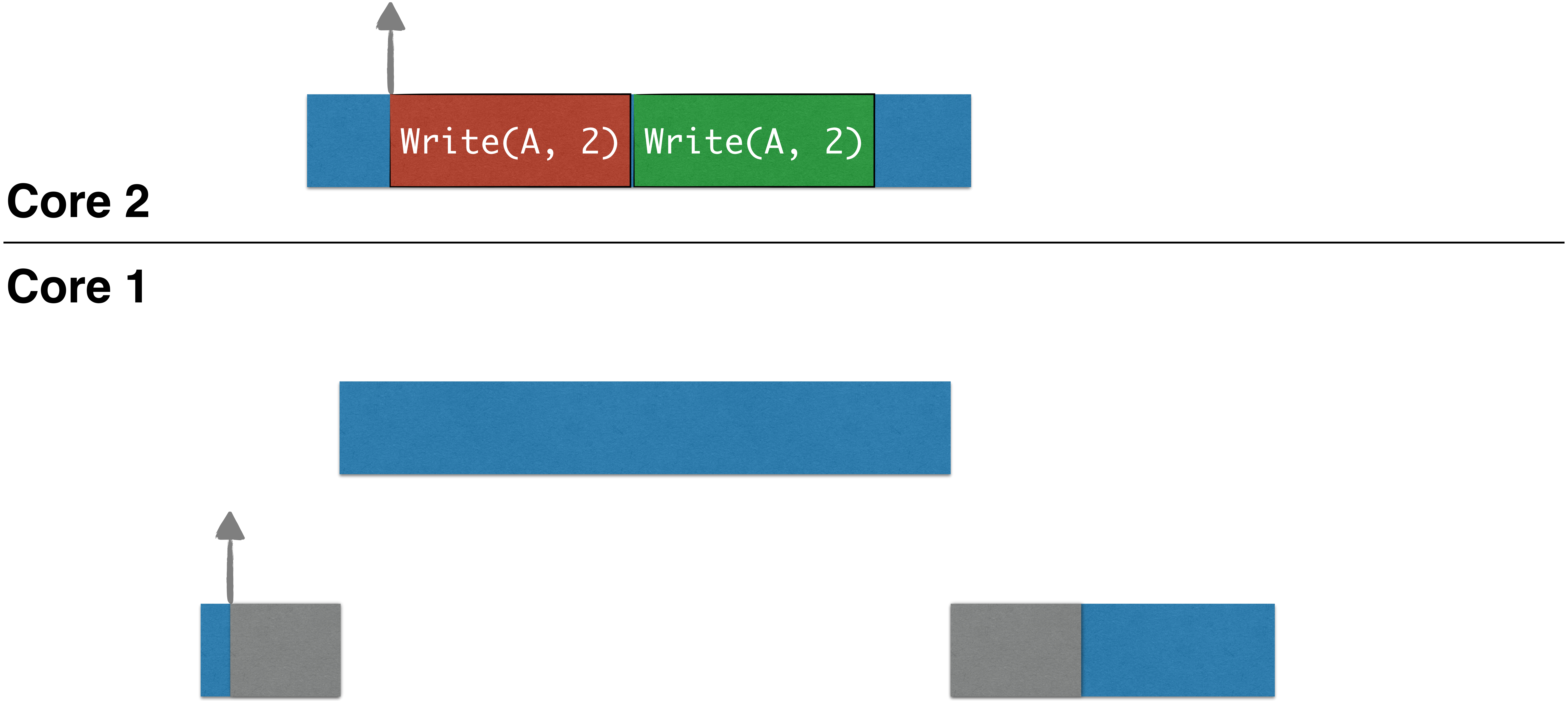
Write(A, 2) Write(A, 2)

Core 2

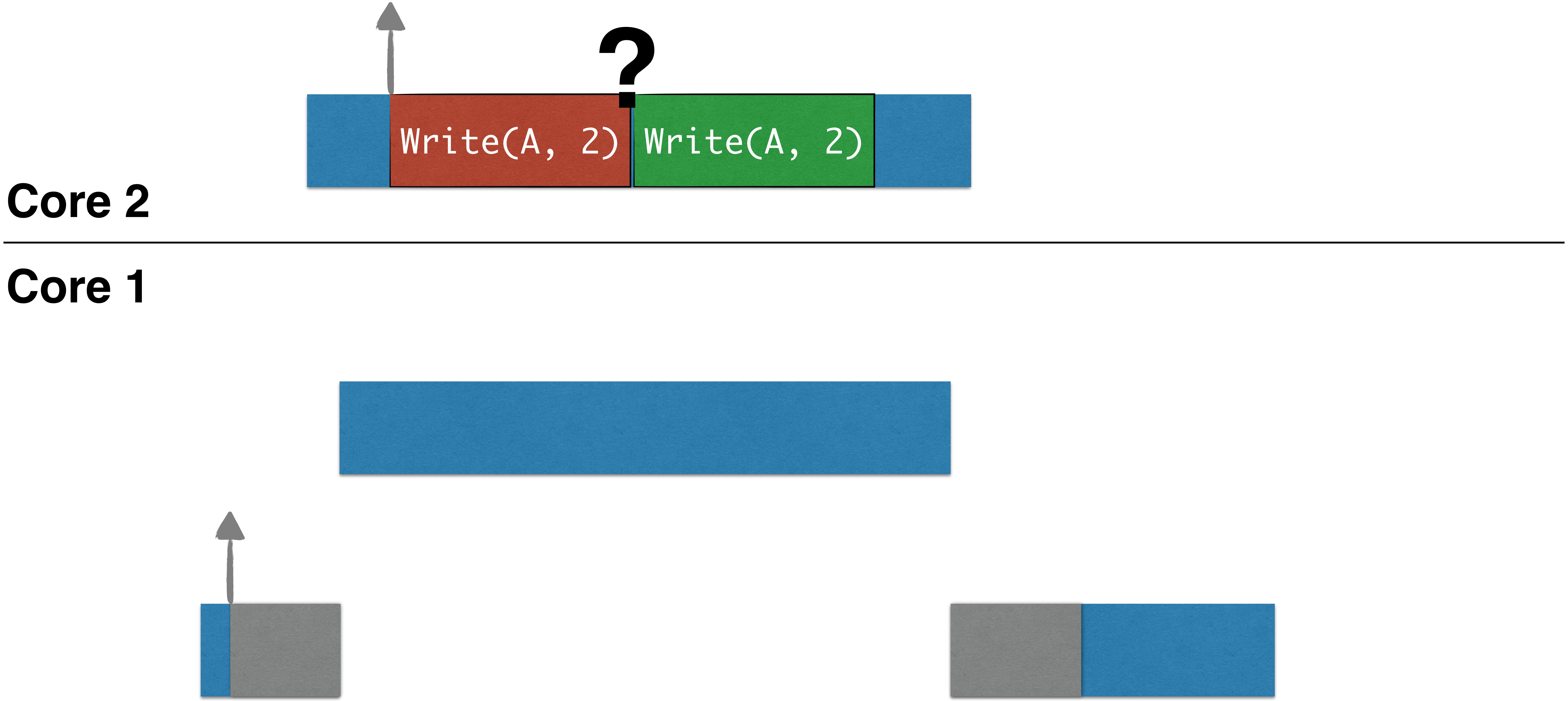
Core 1

Write(A, 1)

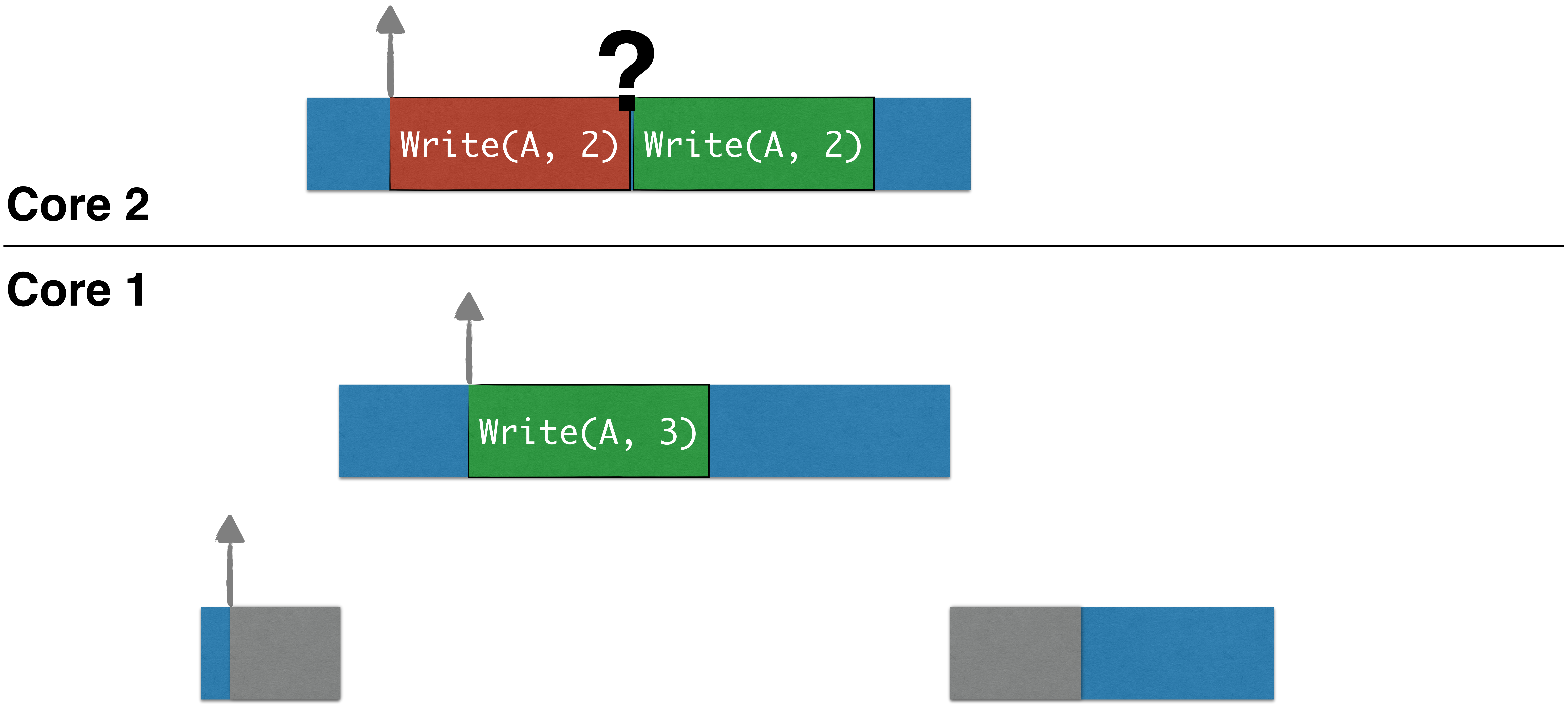
Preemptions and serialisation



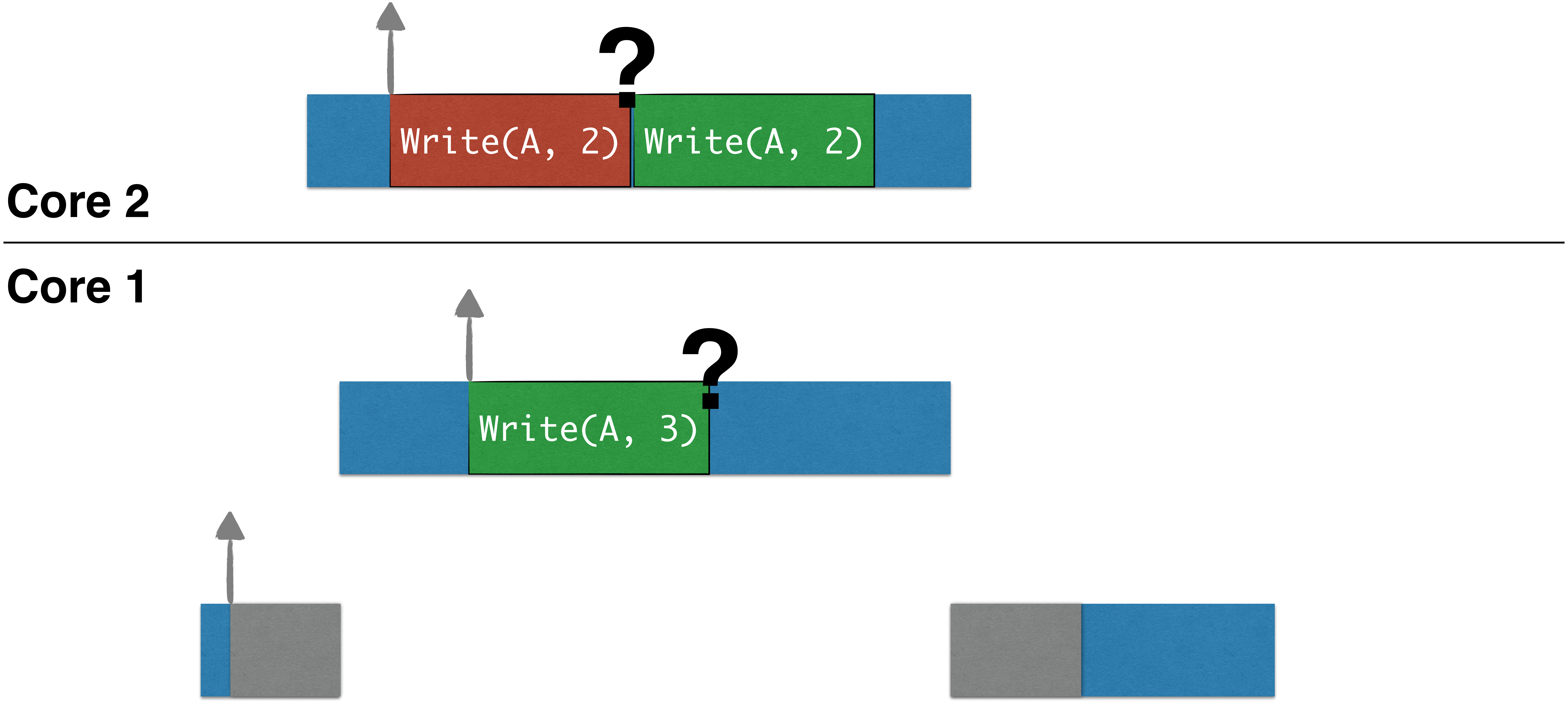
Preemptions and serialisation



Preemptions and serialisation



Preemptions and serialisation



?

Write(A, 2)

Write(A, 2)

Core 2

Core 1

?

Write(A, 3)

What to do?

Increase resistance to preemptions if a transaction can affect concurrent parallel transactions in jobs, while meeting all timing requirements.

Restrict to, at most, ONE transaction in progress, per core.

- No deadlocks.
- No transgression to FIFO serialisation.

What to do?

Increase resistance to preemptions if a transaction can affect concurrent parallel transactions in jobs, while meeting all timing requirements.

Restrict to, at most, ONE transaction in progress, per core.

- No deadlocks.
- No transgression to FIFO serialisation.

SRP-TM

Scheduling jobs with transactions: SRP-TM

Assumptions

General scheduling rule: P-EDF.

While a transaction is in progress on a core: SRP ⇒ SRP-TM.

- Adds static preemption levels to tasks.
- Adds static preemption level to transactions.
- Adds variable ceiling to cores.
    - Highest preemption level of a task that could be waiting for the current transaction in progress to commit.

Assigning preemption levels to tasks

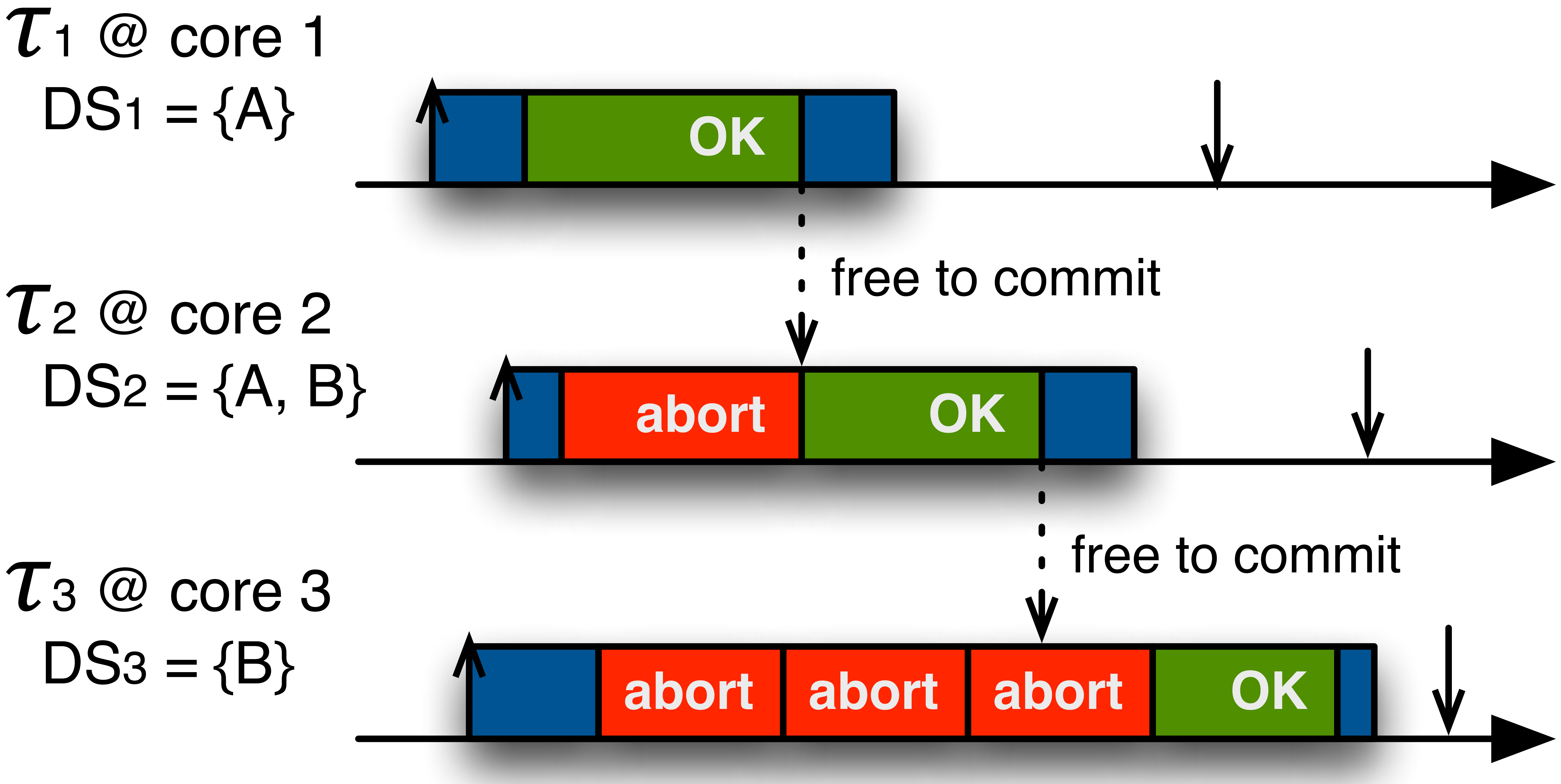
Just like SRP, assign preemption levels to all tasks in set by increasing order of relative deadline…

… independently of core affinities.

Task	**Relative deadline**	**Preemption level**
T5	120	1
T2	100	2
T3	80	3
T4	70	4
T6, T7	60	5
T1	50	6

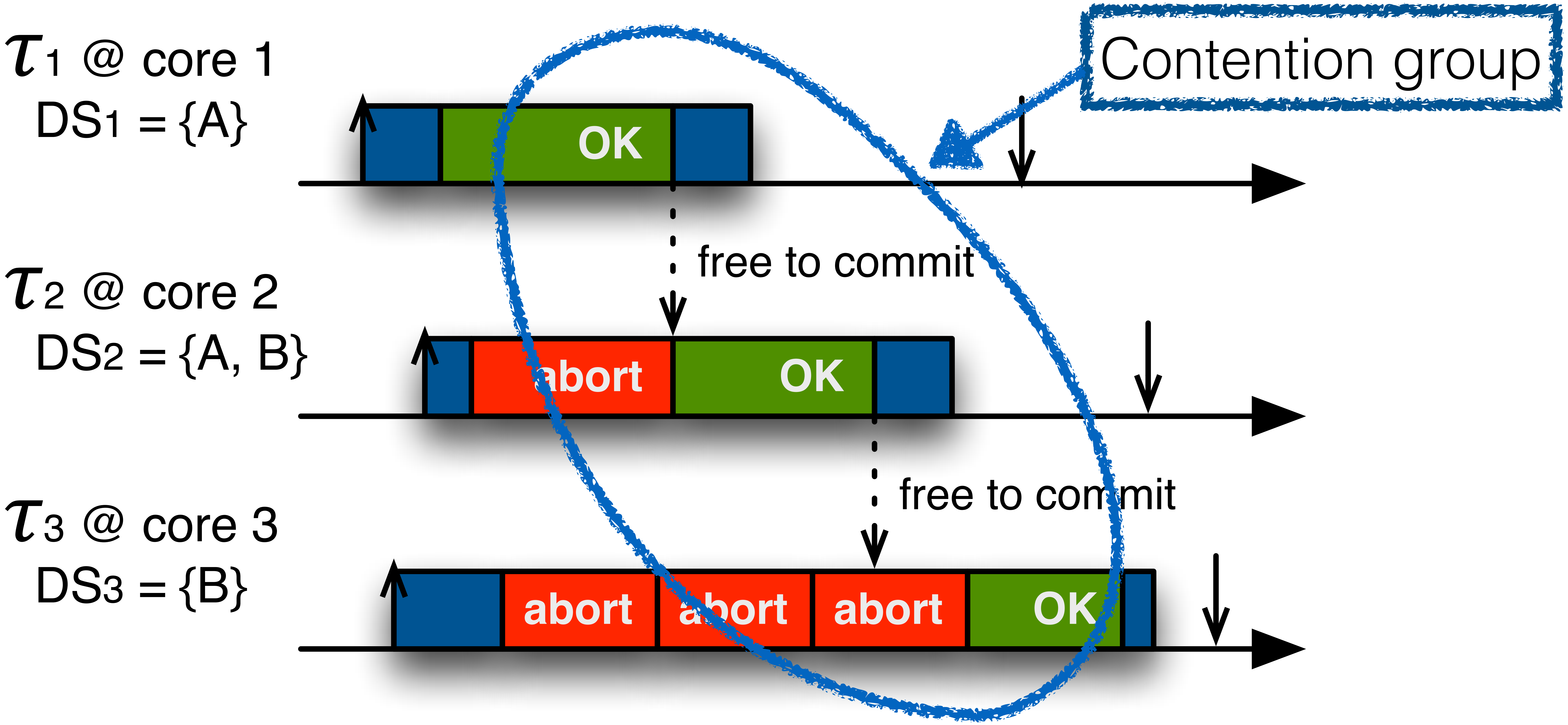
Assigning preemption levels to transaction

Assign to each transaction the highest preemption level from all tasks that have one transaction that may depend on it to progress.



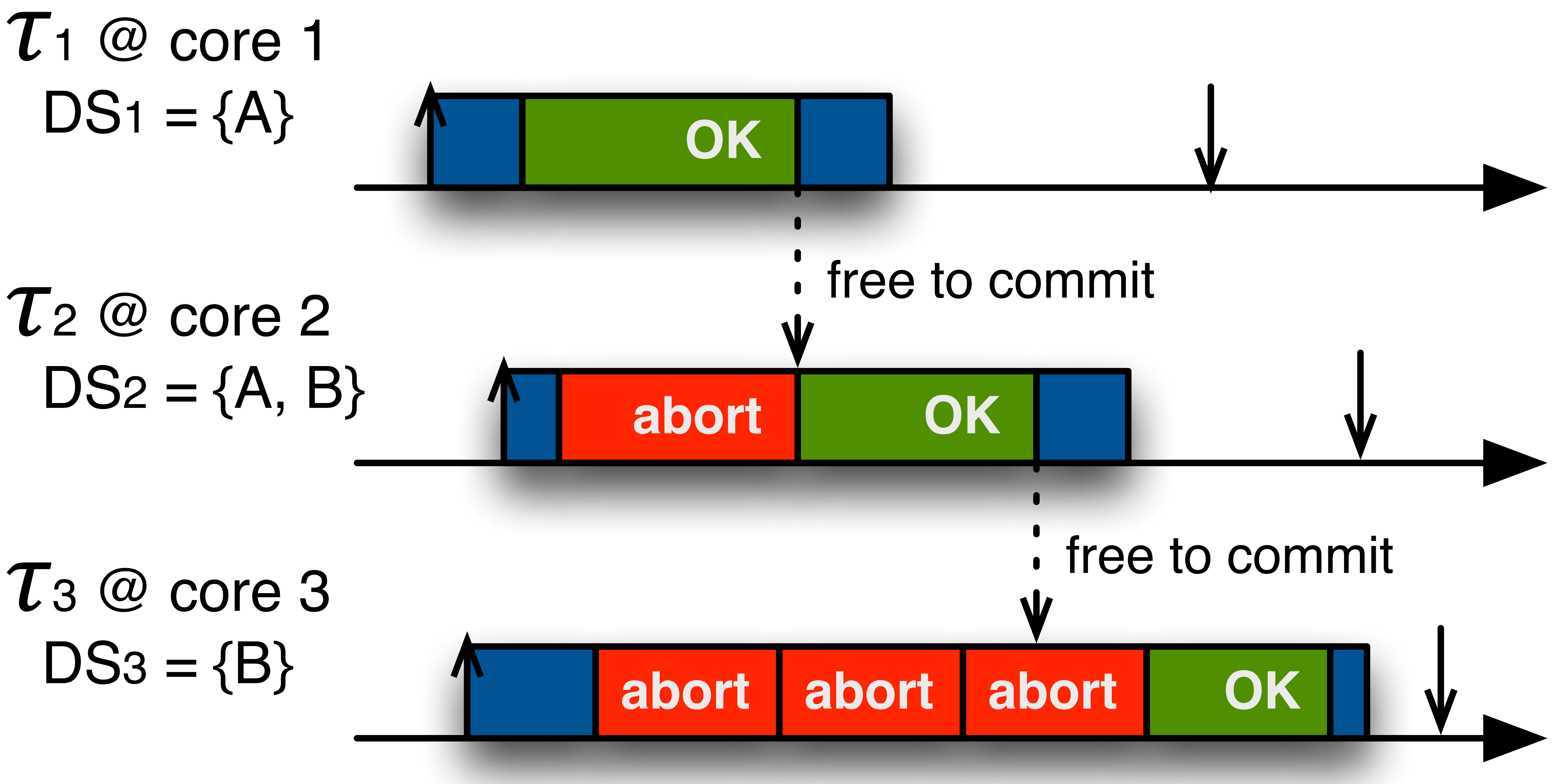
Assigning preemption levels to transaction

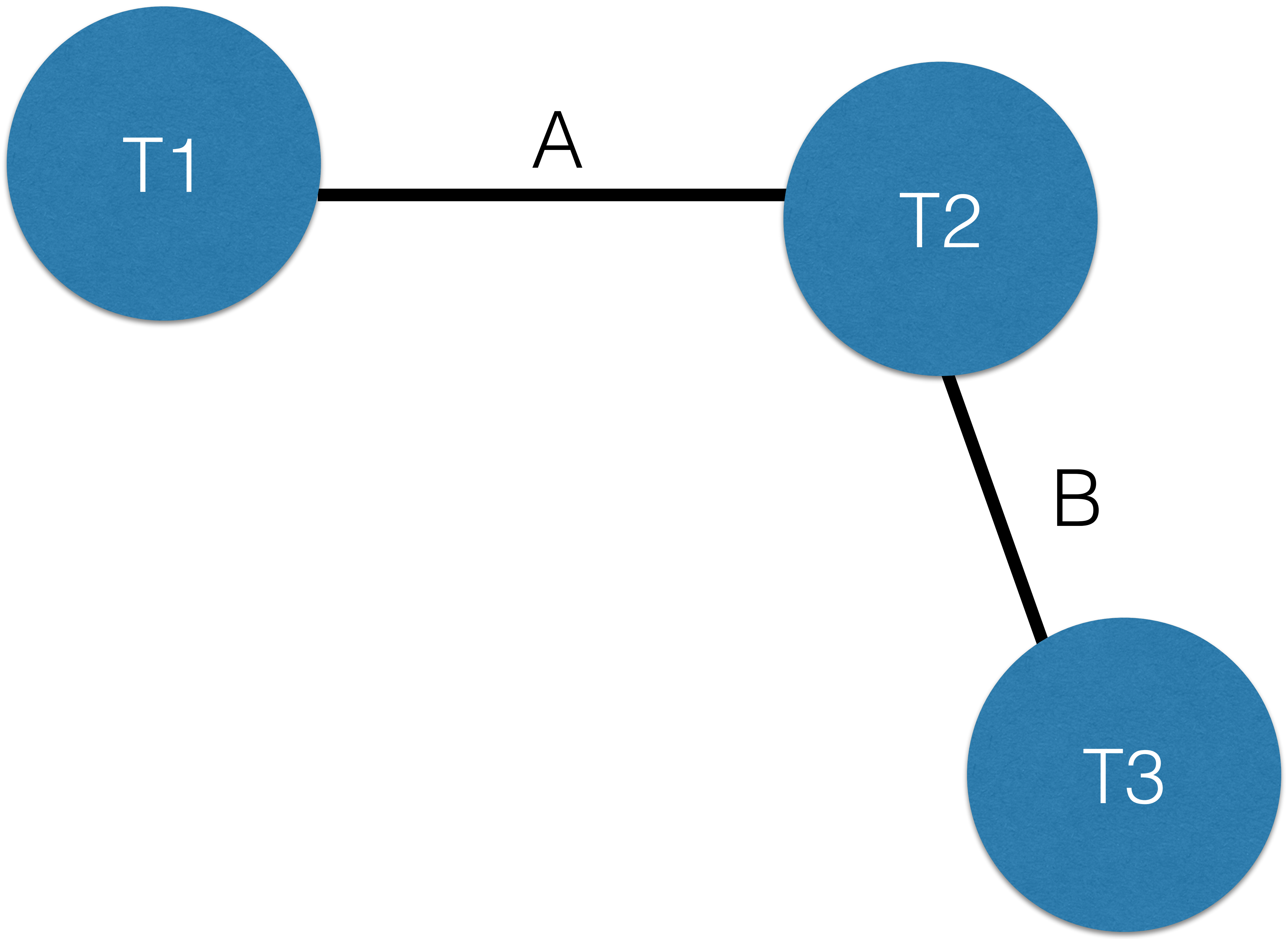
Assign to each transaction the highest preemption level from all tasks that have one transaction that may depend on it to progress.



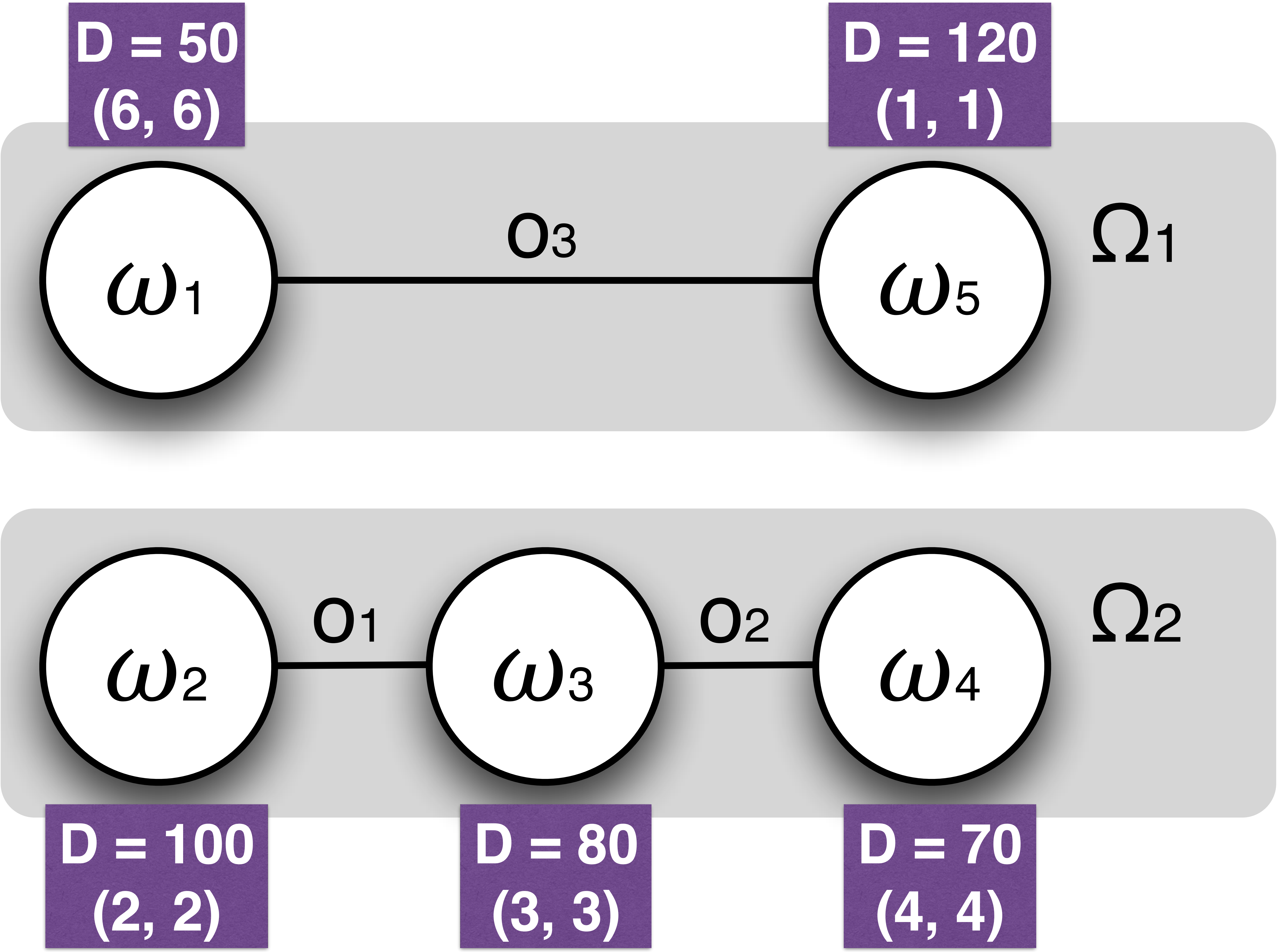
Assigning preemption levels to transaction

Assign to each transaction the highest preemption level from all tasks that have one transaction that may depend on it to progress.



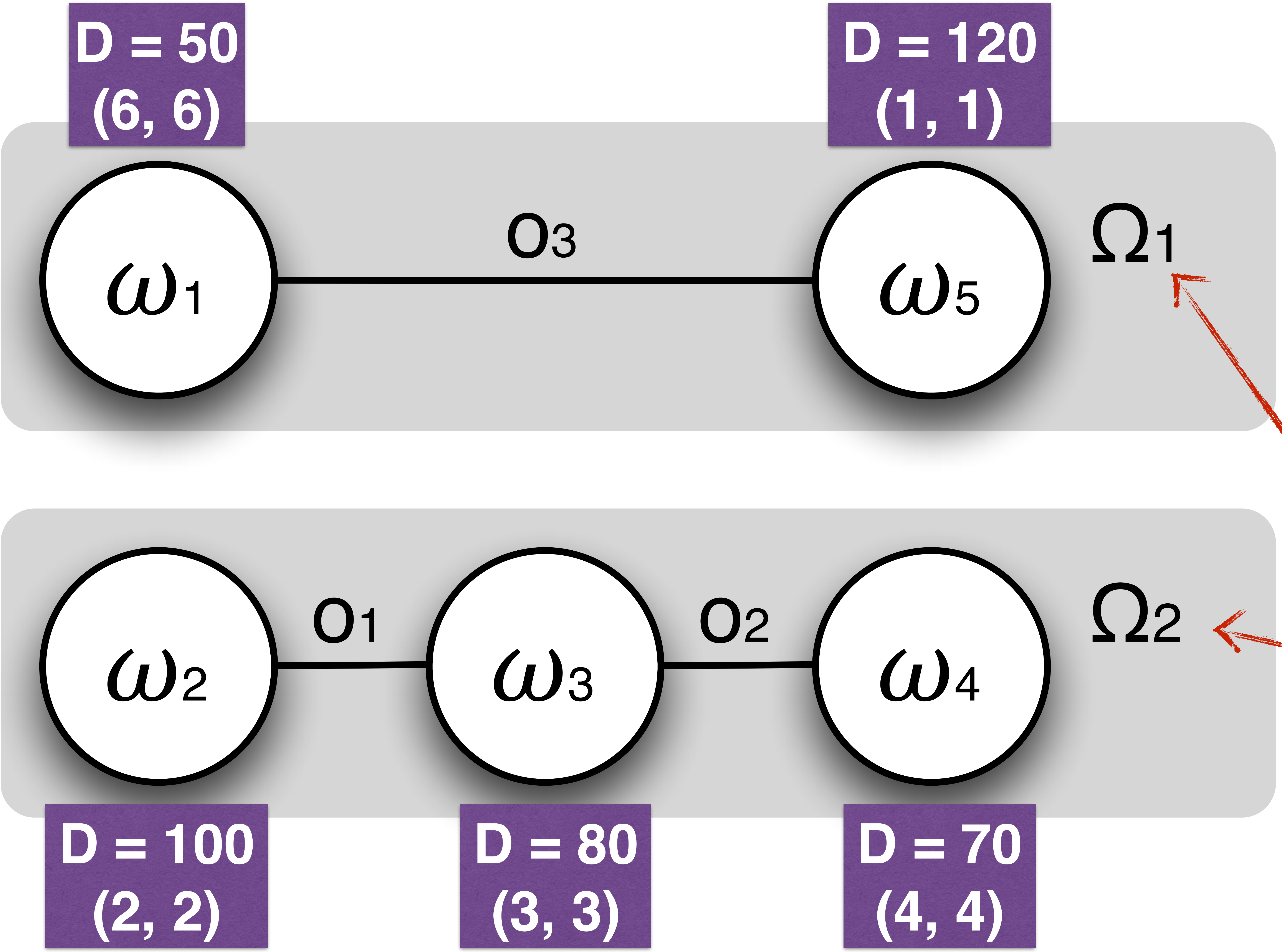


A practical example



$DS_1 = DS_5 = \{o_3\}$
$DS_2 = \{o_1\}$
$DS_3 = \{o_1, o_2\}$
$DS_4 = \{o_2\}$

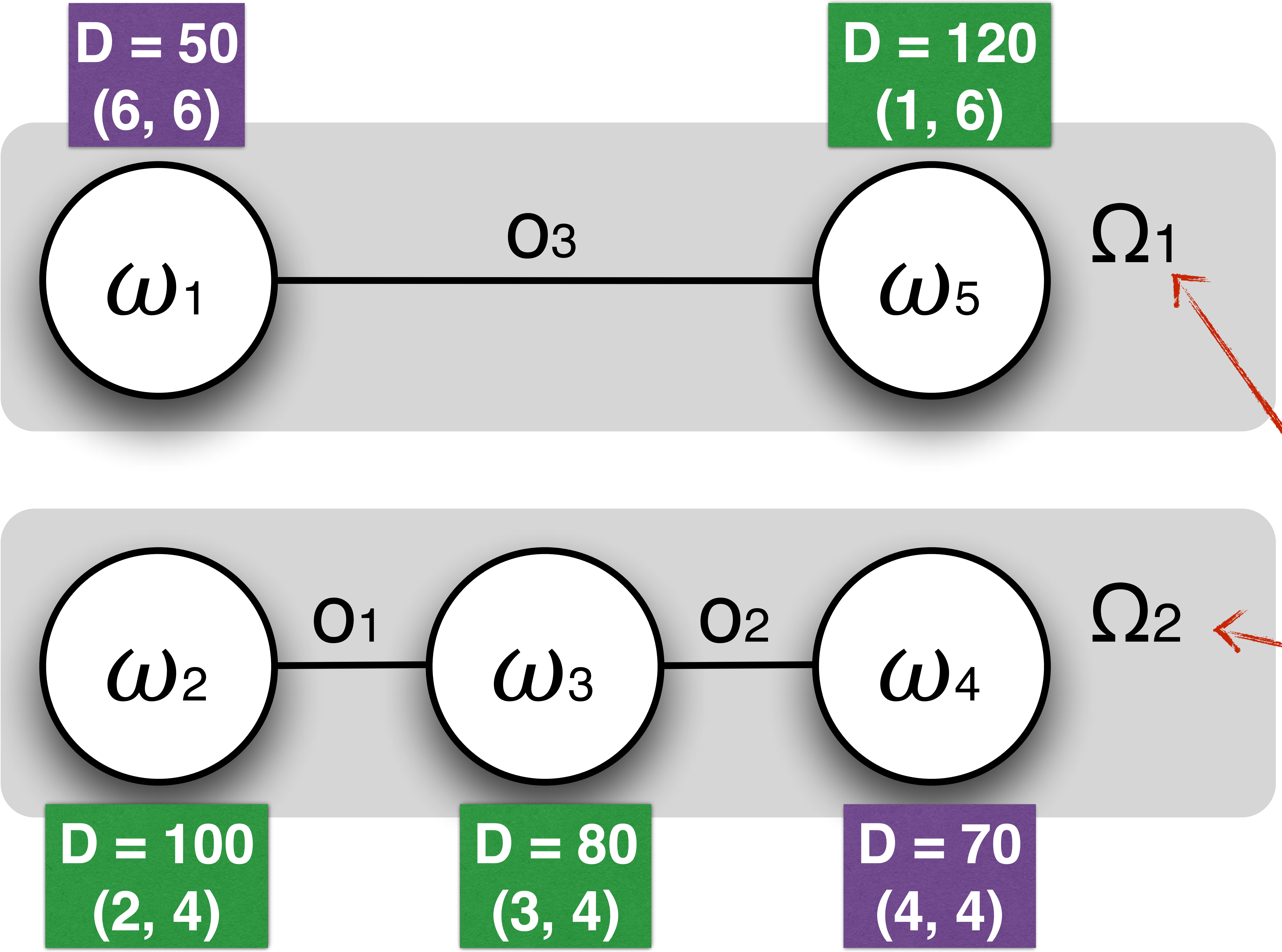
A practical example



$DS_1 = DS_5 = \{o_3\}$
$DS_2 = \{o_1\}$
$DS_3 = \{o_1, o_2\}$
$DS_4 = \{o_2\}$

Contention groups

A practical example

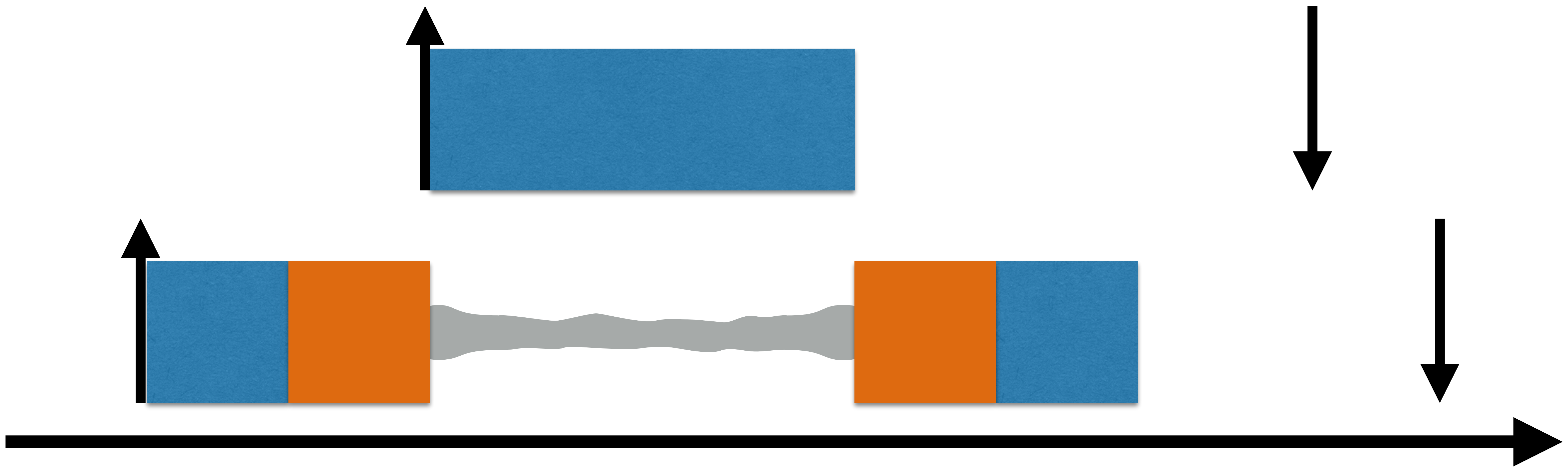


$DS_1 = DS_5 = \{o_3\}$
$DS_2 = \{o_1\}$
$DS_3 = \{o_1, o_2\}$
$DS_4 = \{o_2\}$

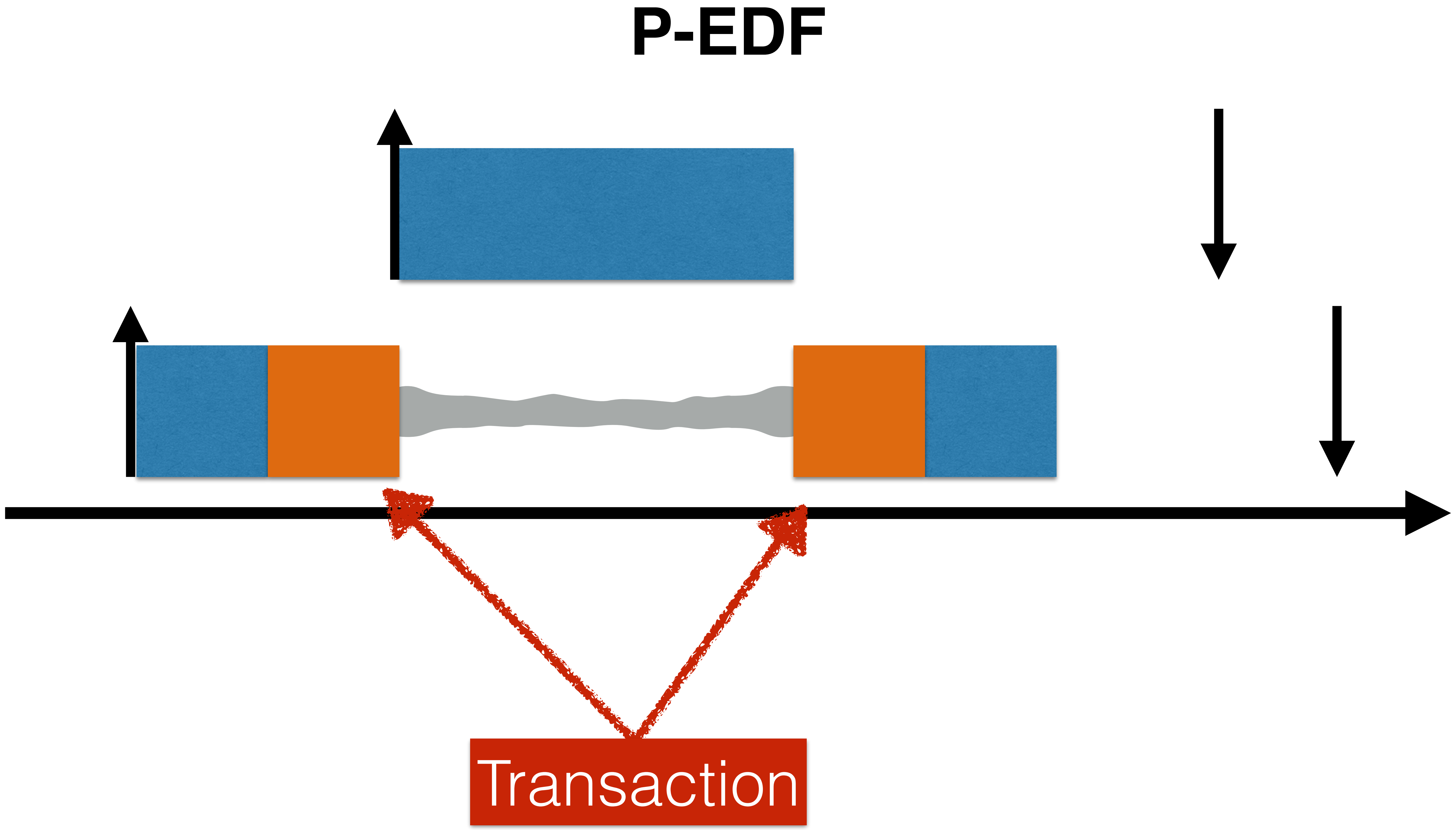
Contention groups

Transaction vs. Transactionless

P-EDF

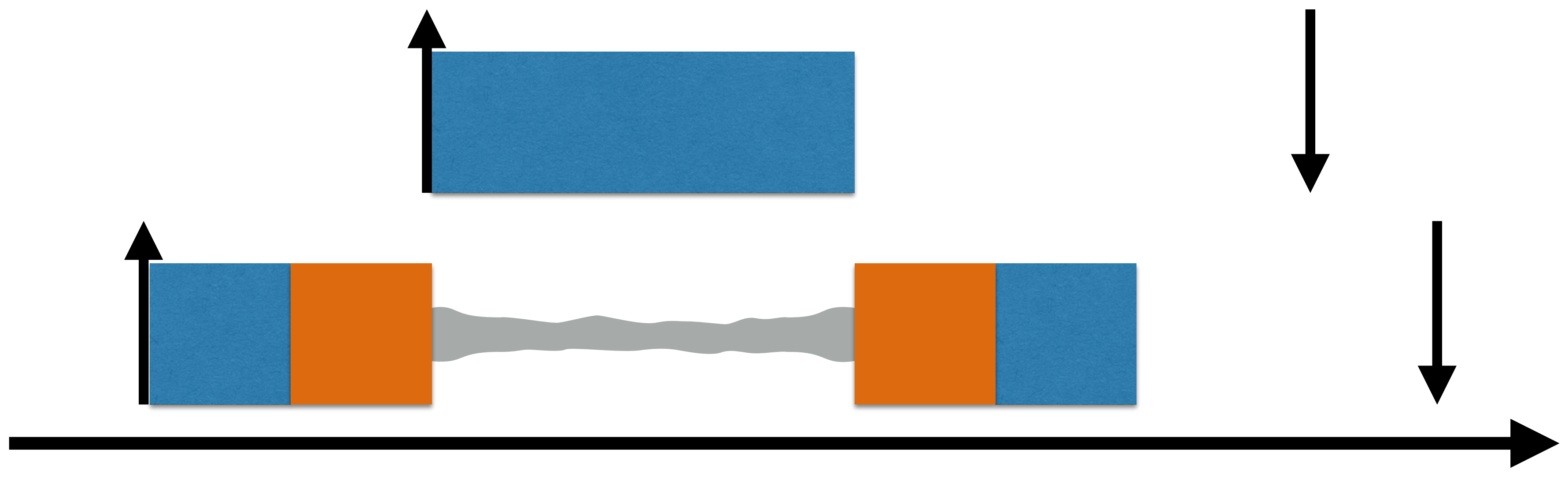


Transaction vs. Transactionless

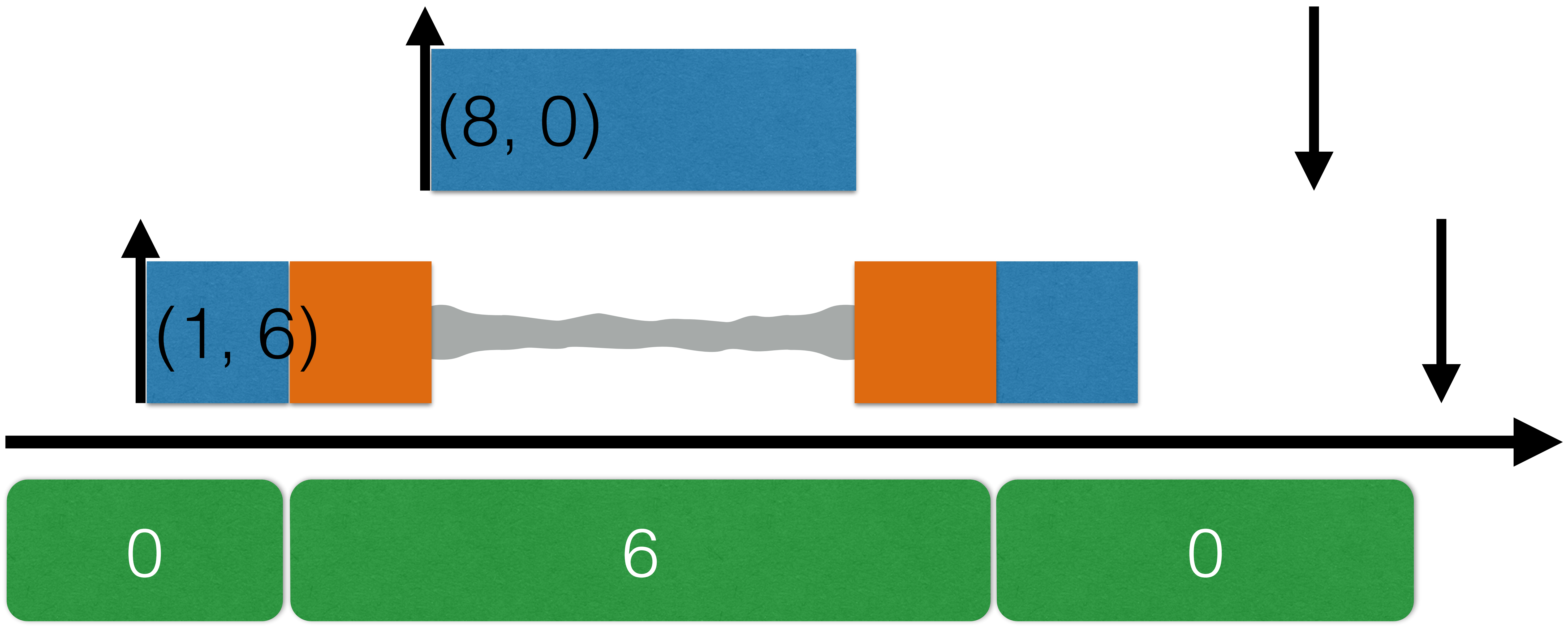


Transaction vs. Transactionless

P-EDF



SRP-TM

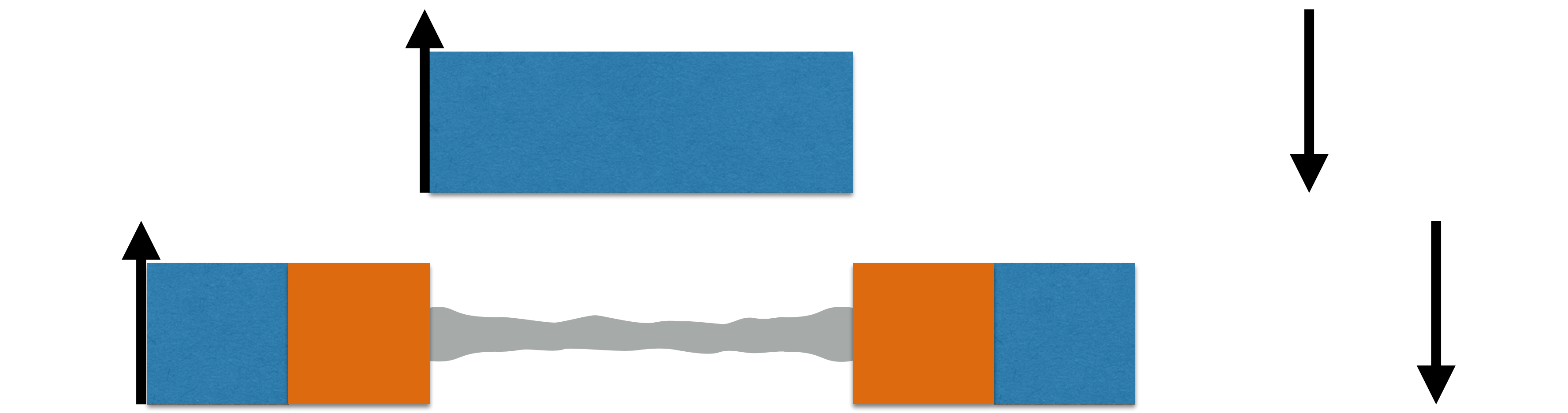


(Task PL, Transaction PL)

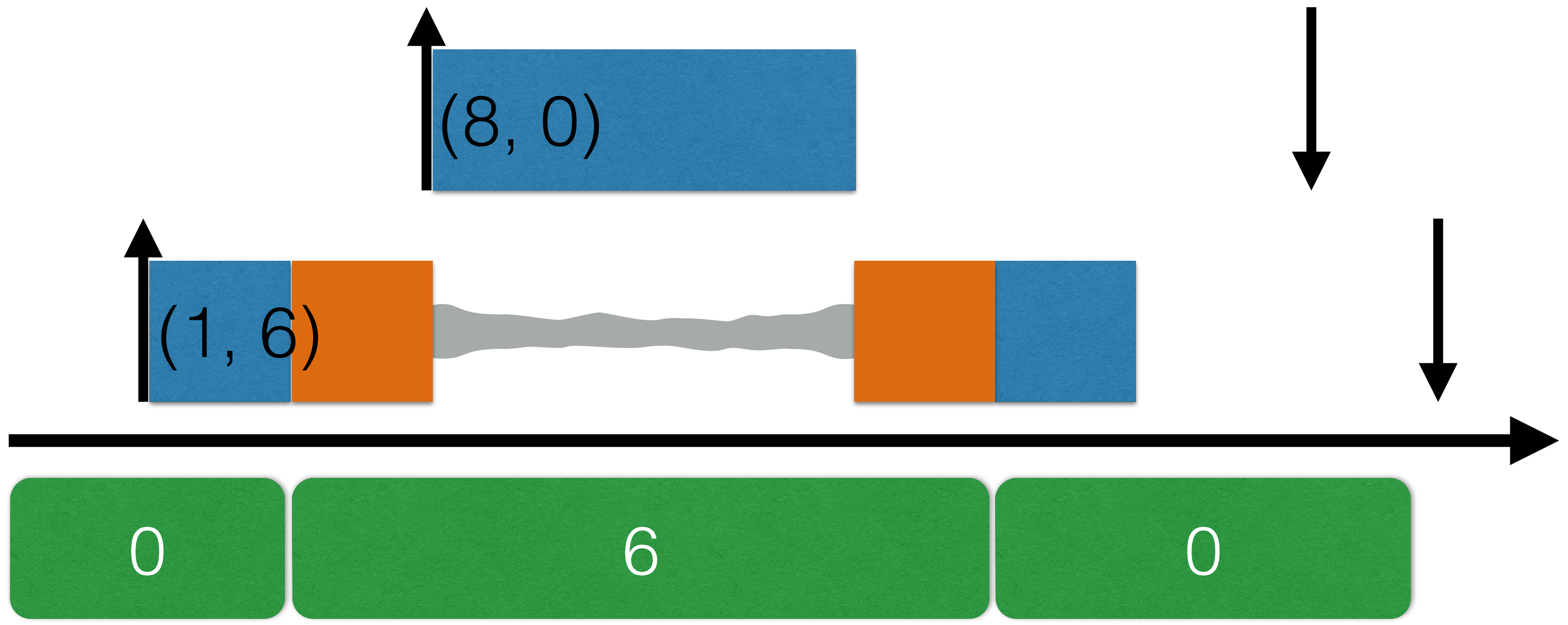
X Core ceiling

Transaction vs. Transactionless

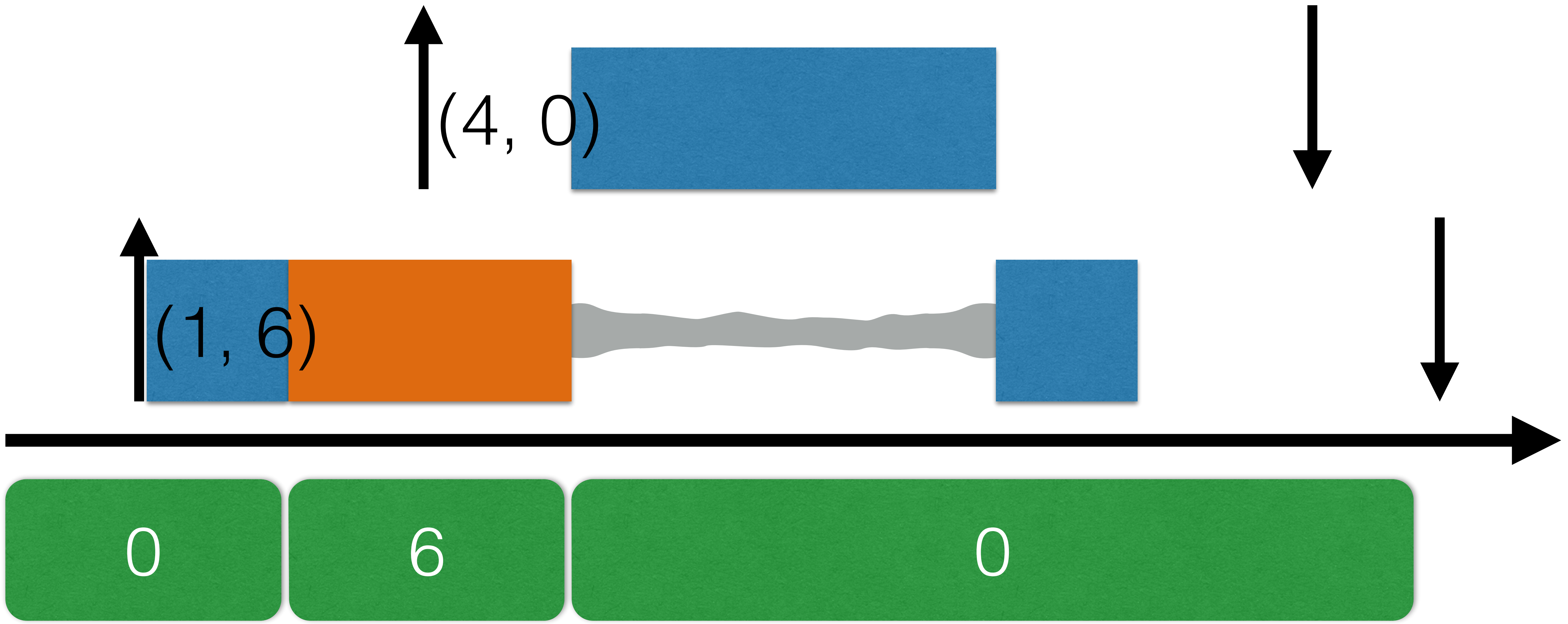
P-EDF



SRP-TM



SRP-TM

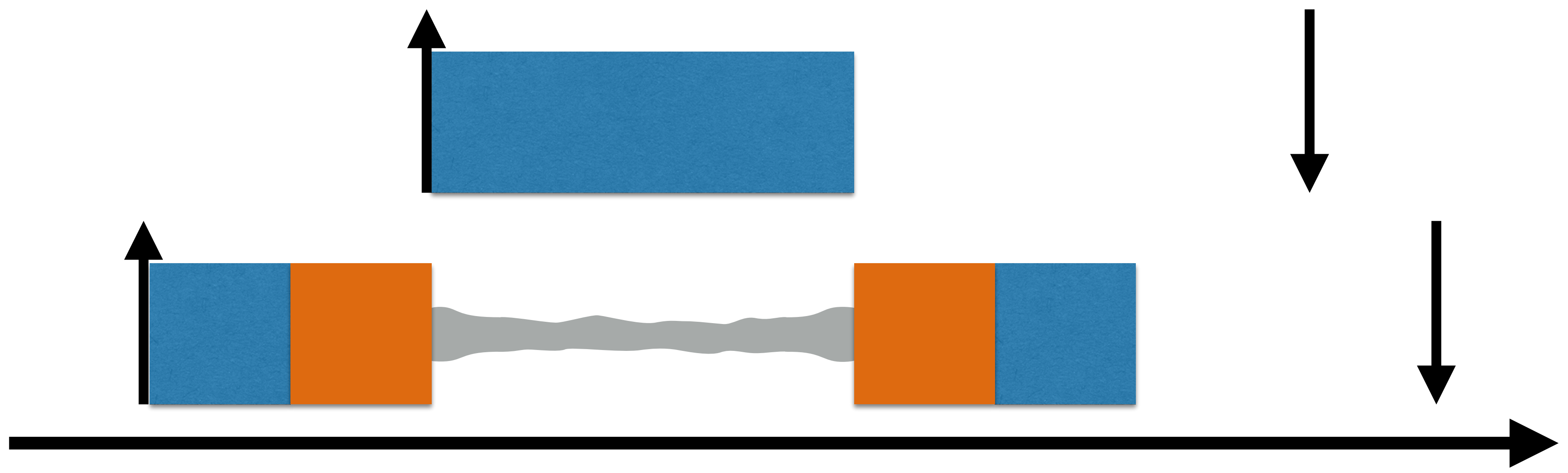


(Task PL, Transaction PL)

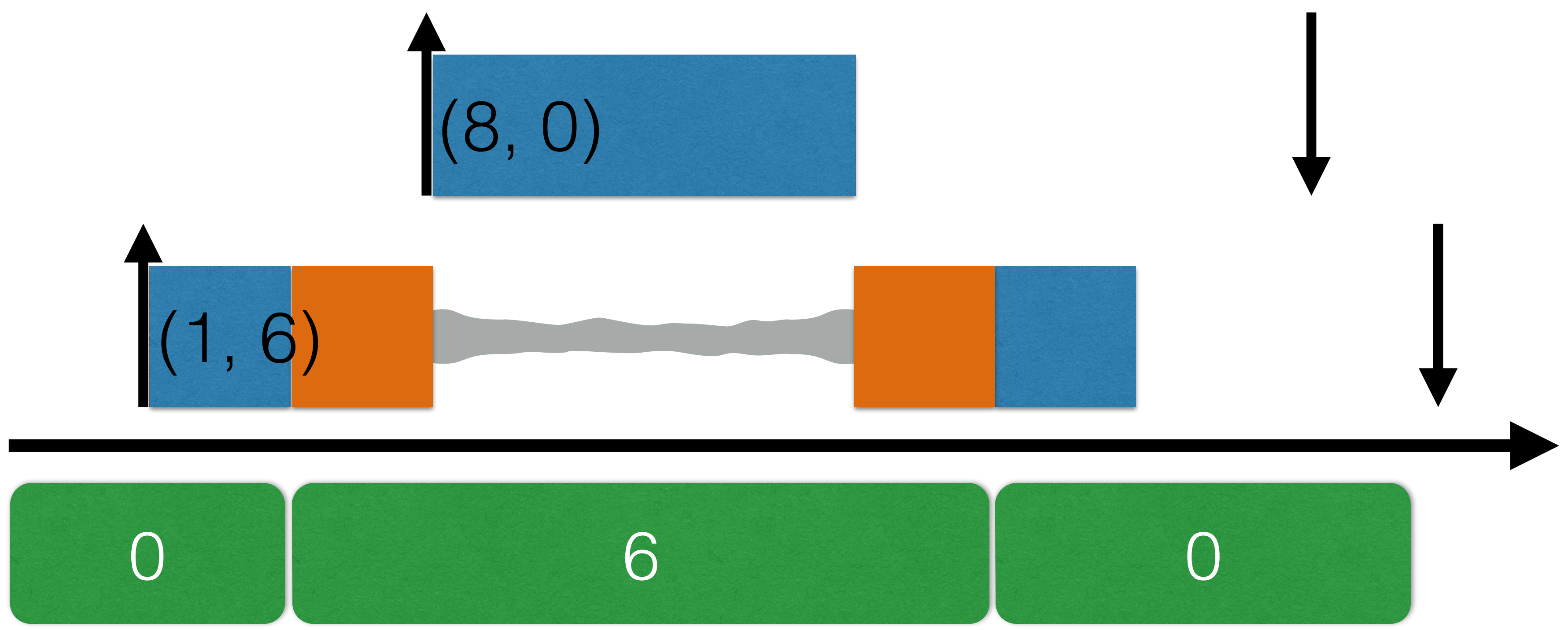
🟩 X Core ceiling

Transaction vs. Transactionless

P-EDF

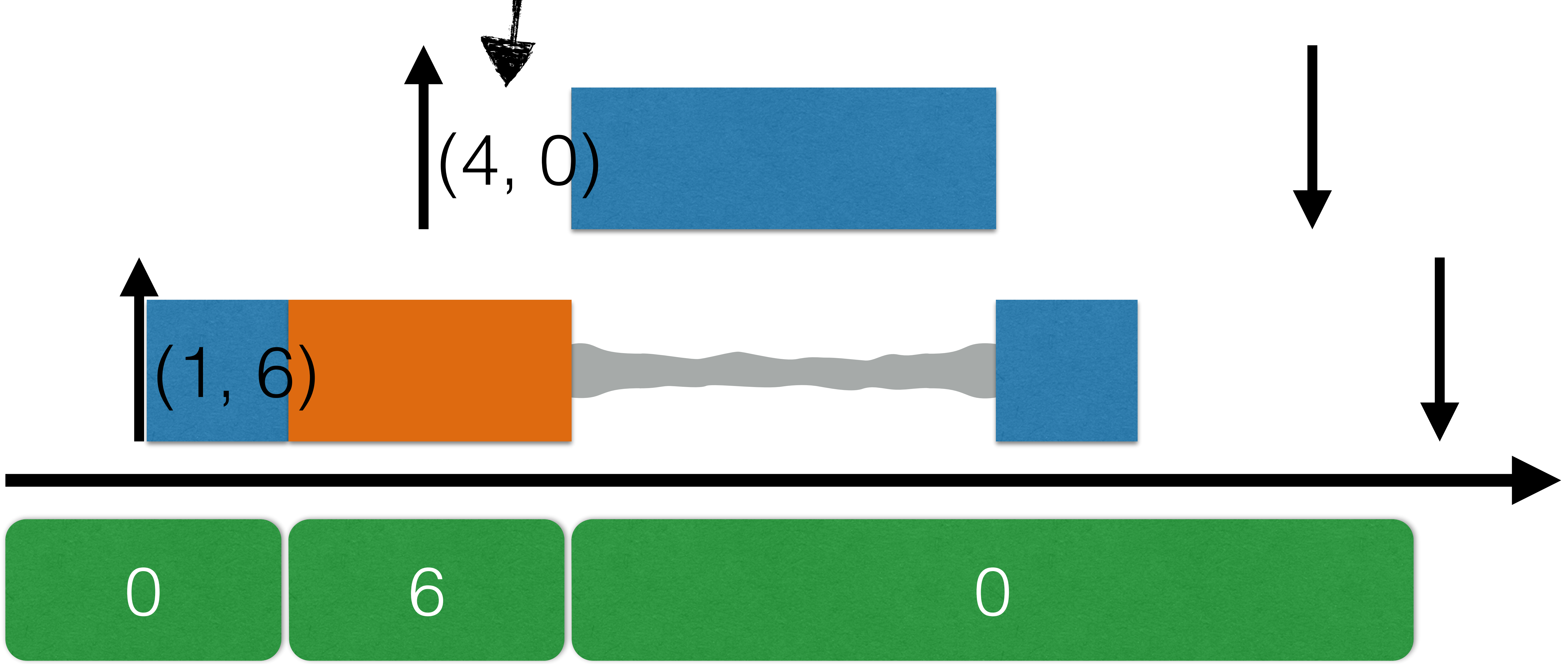


SRP-TM



Direct blocking

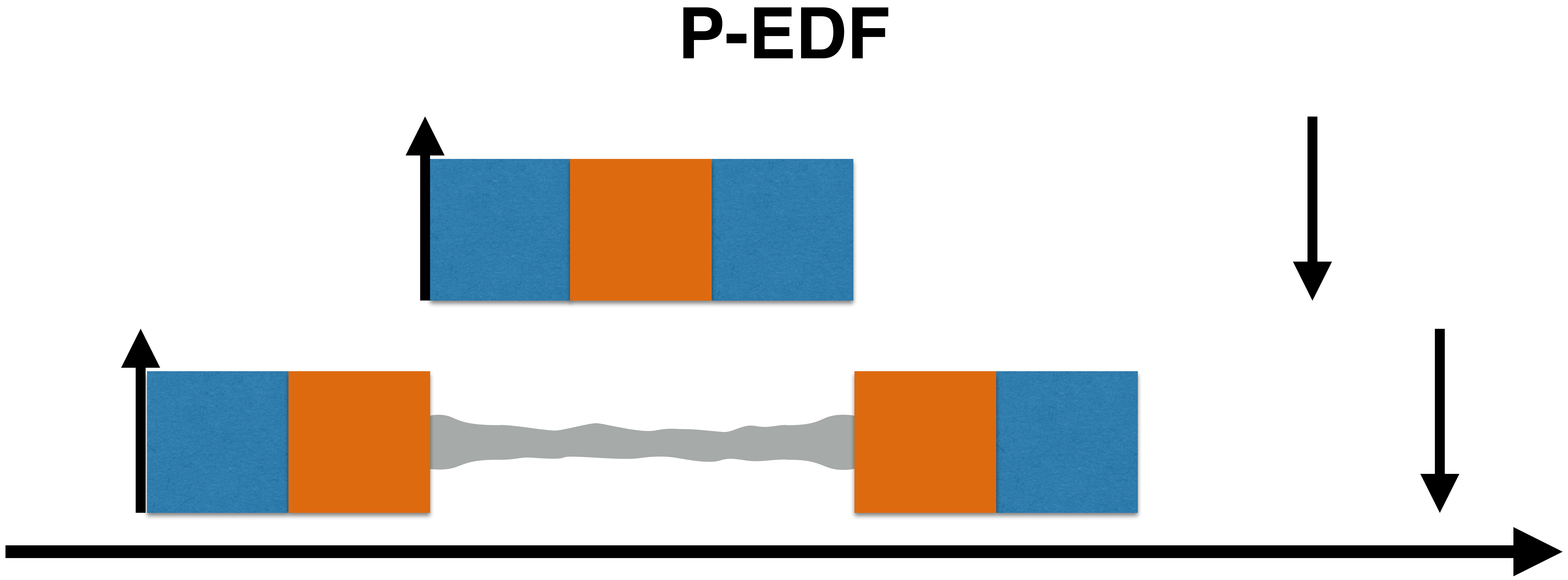
SRP-TM



(Task PL, Transaction PL)

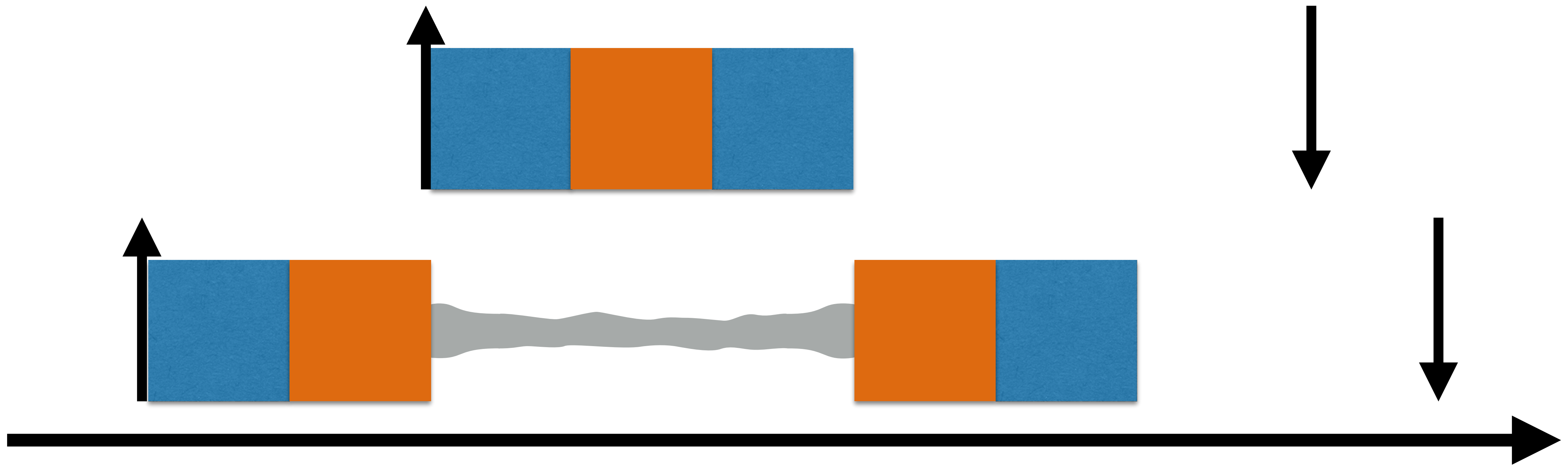
🟩 X Core ceiling

Transaction vs. Transaction

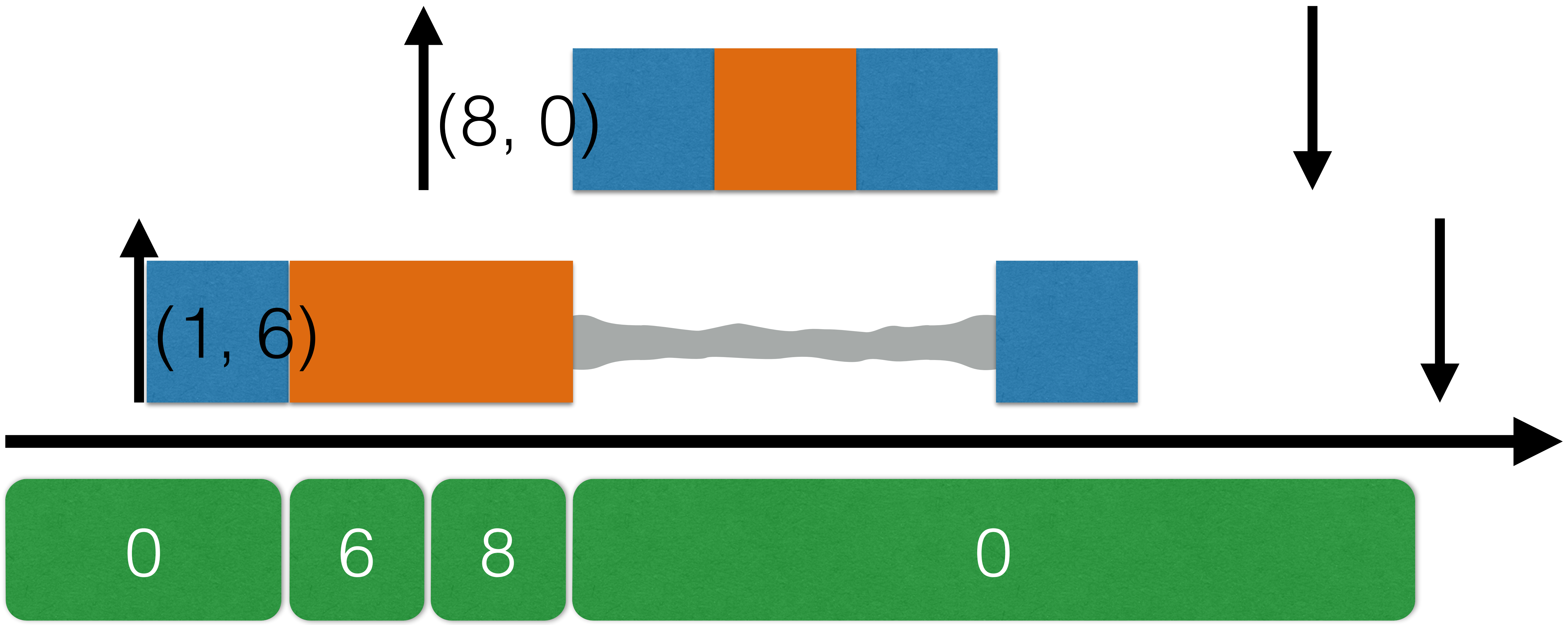


Transaction vs. Transaction

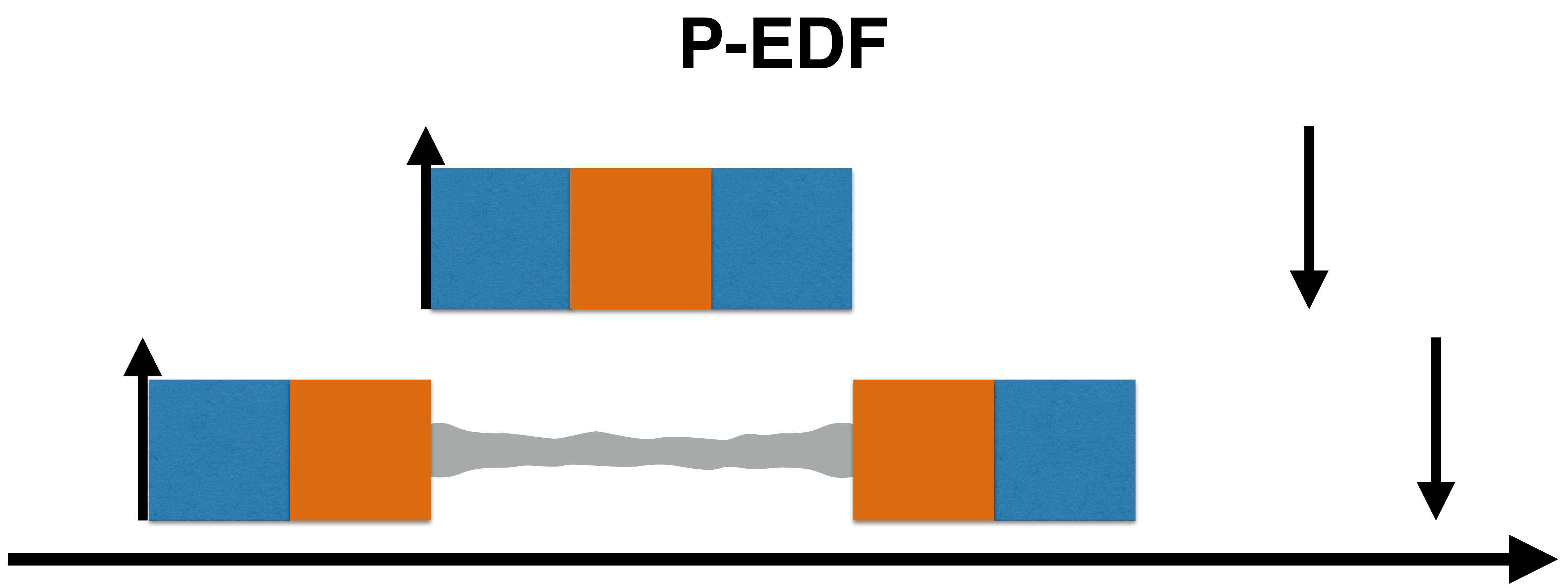
P-EDF

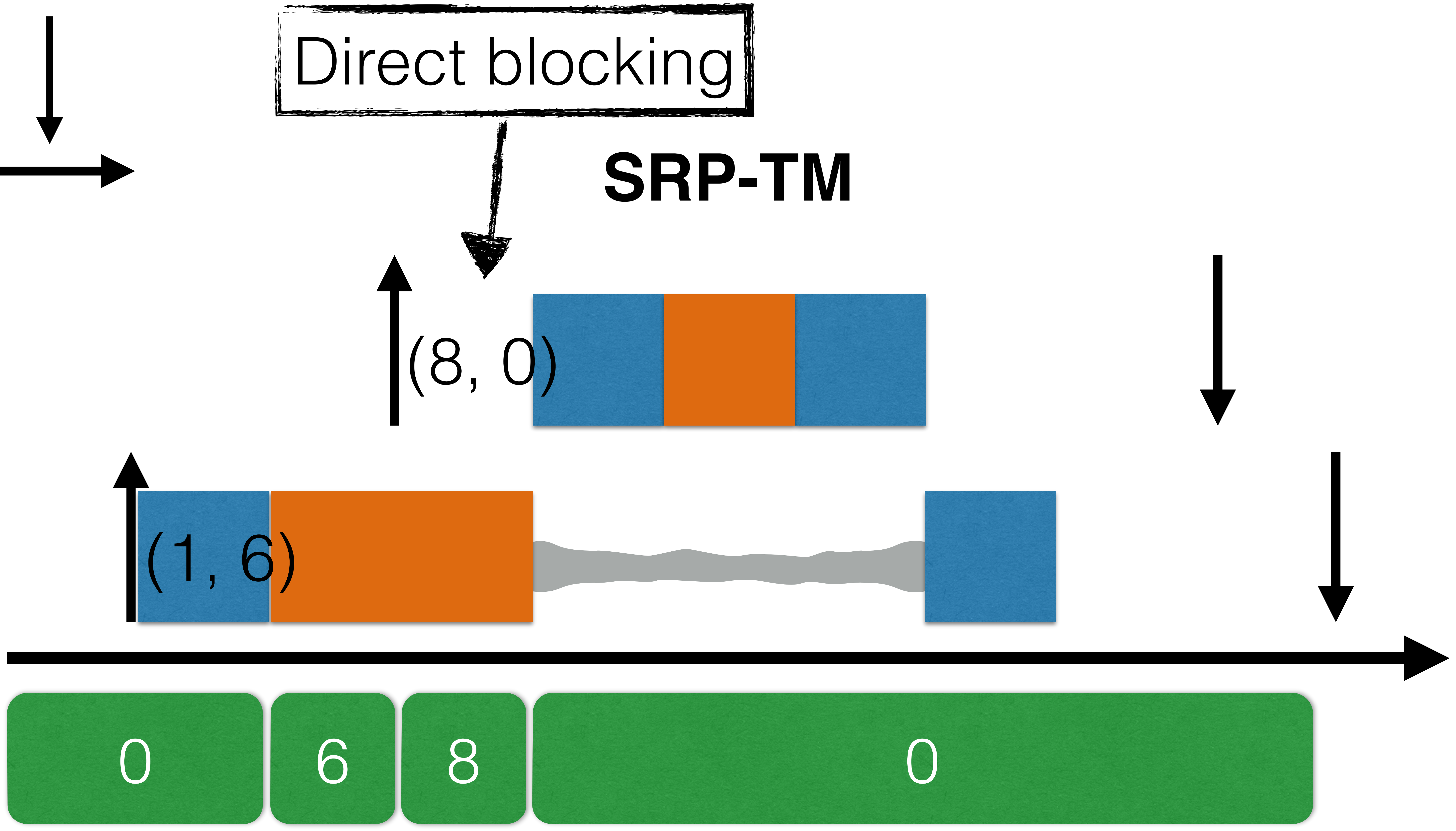


SRP-TM

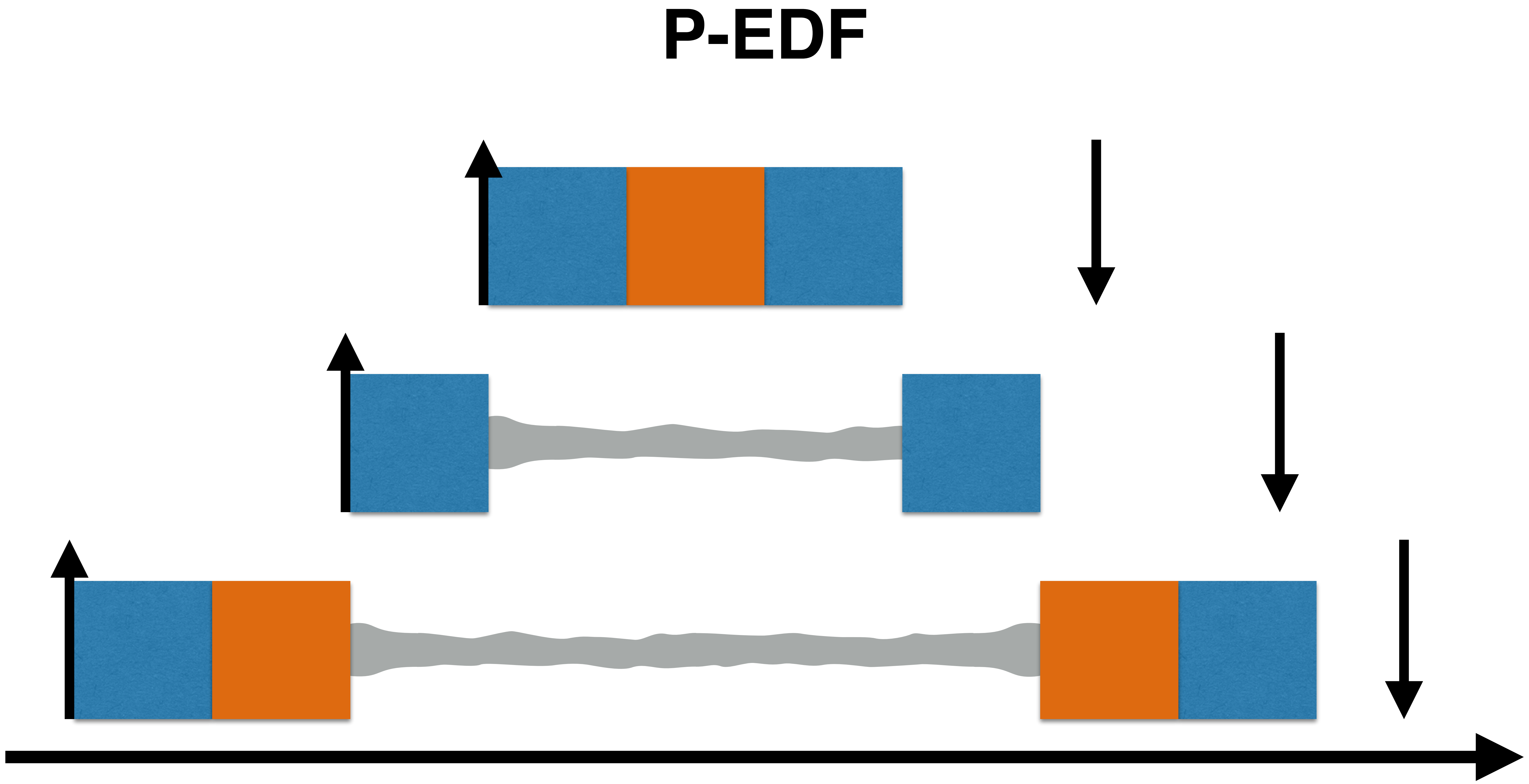


Transaction vs. Transaction

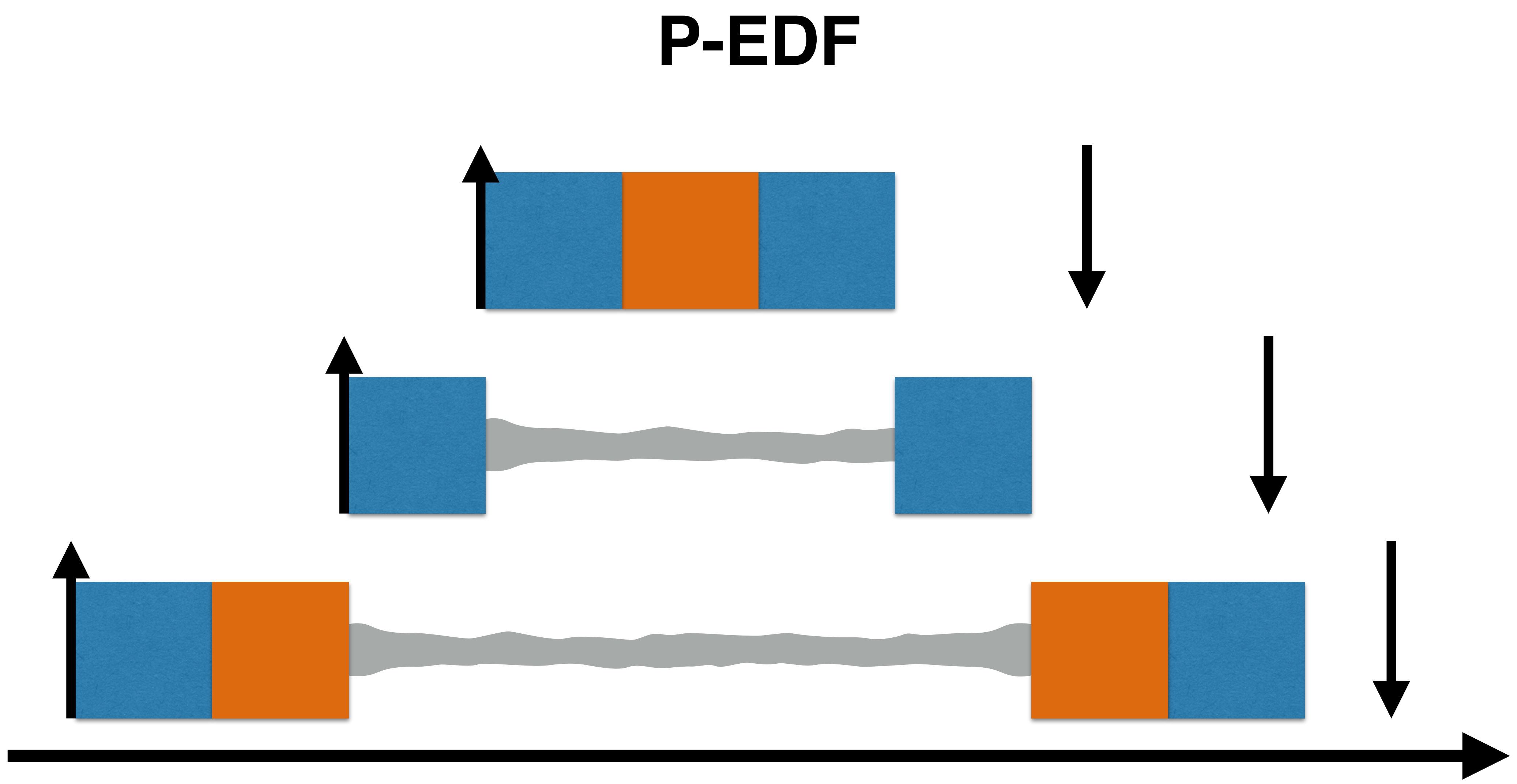


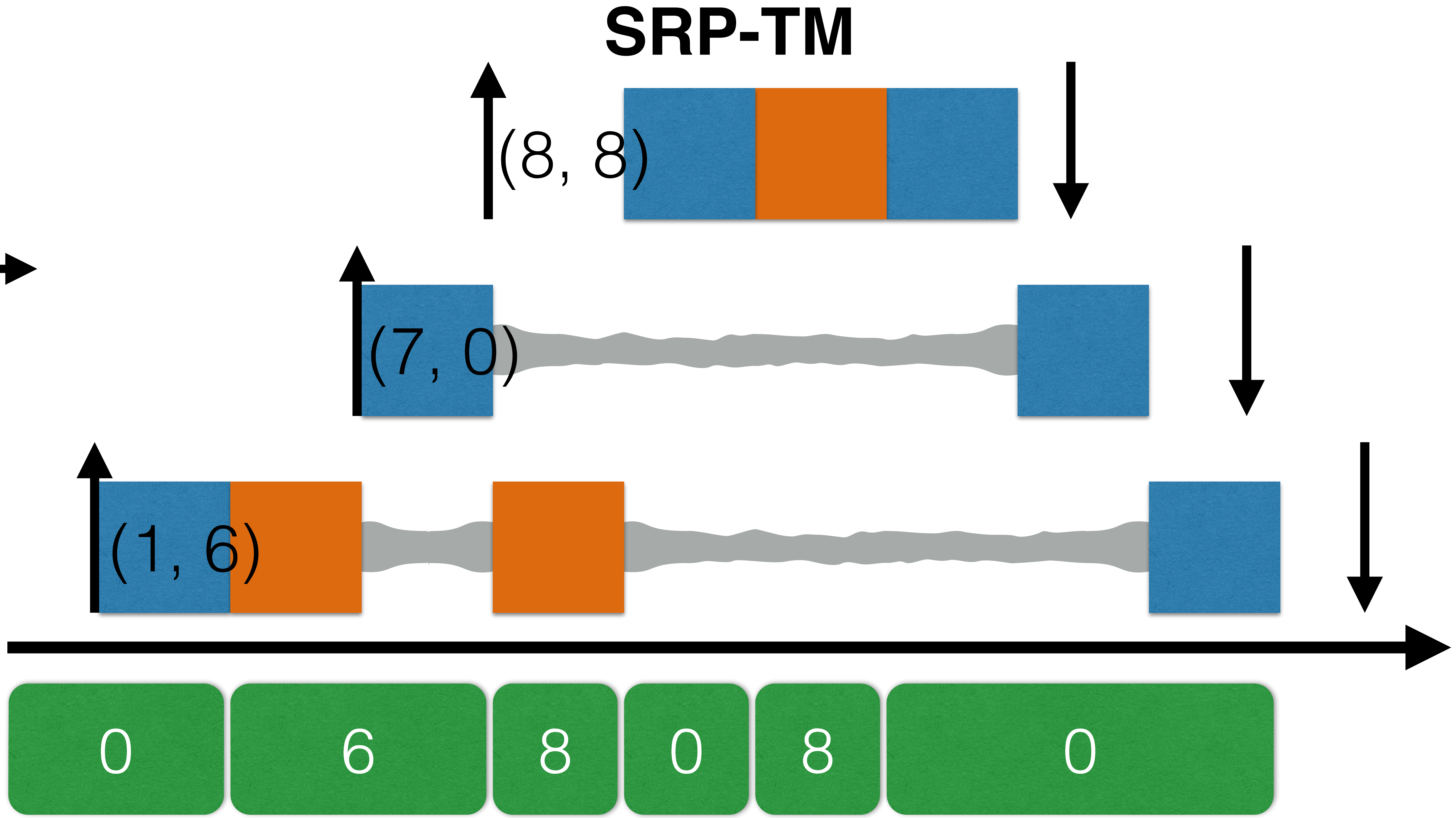


Mixing all together

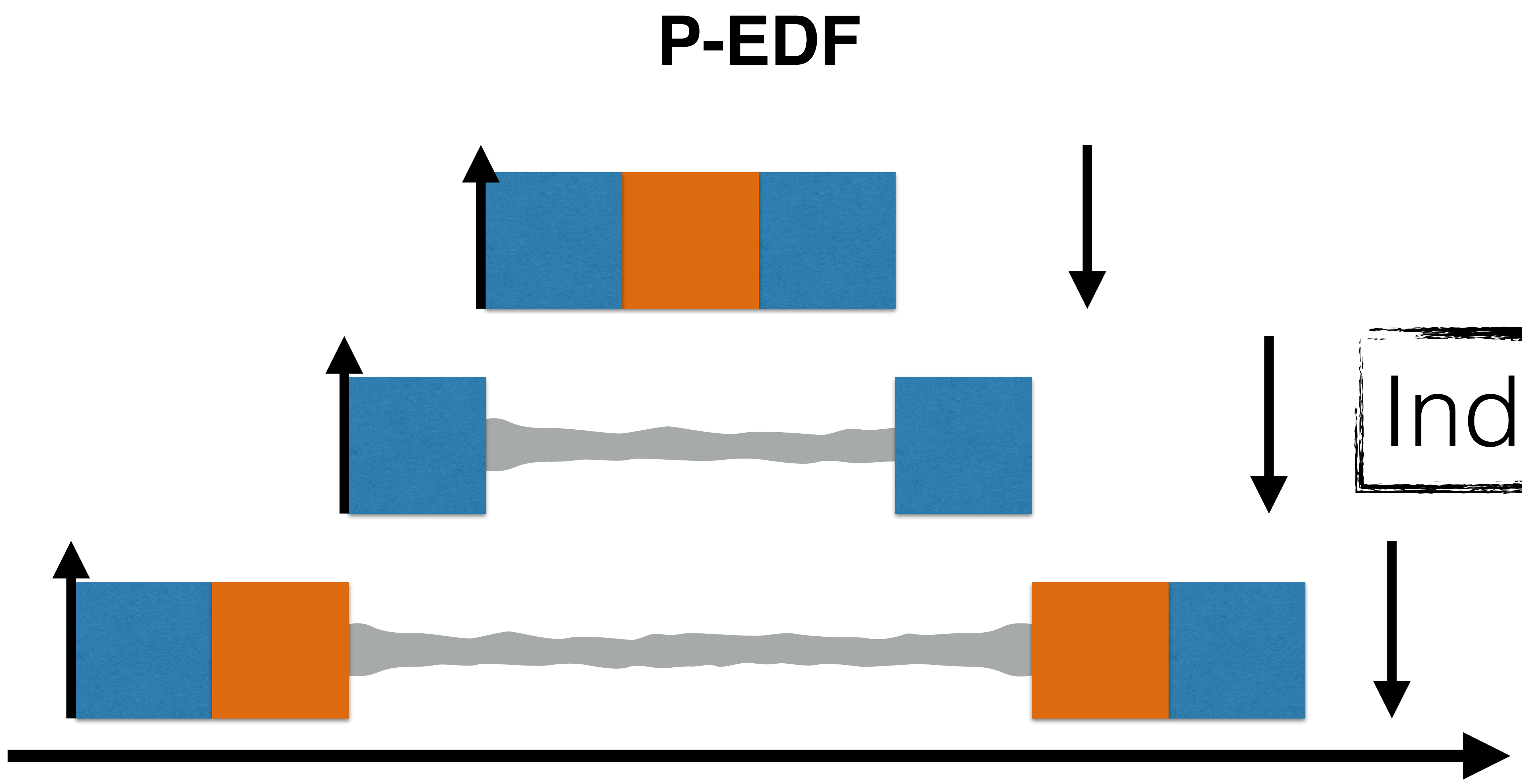


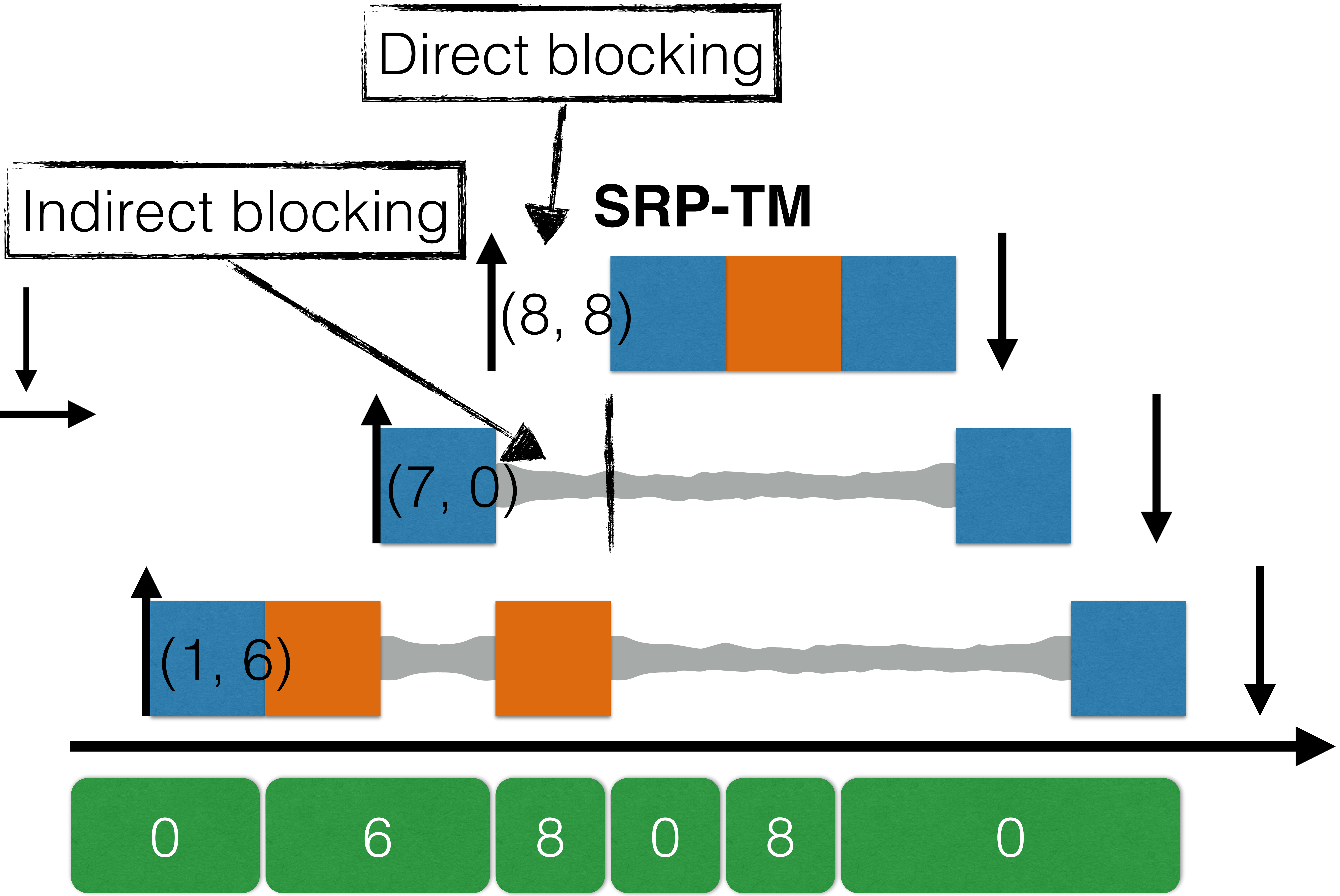
Mixing all together





Mixing all together




Direct blocking

Indirect blocking

SRP-TM operations in short

Transaction starts:

- Core ceiling is set to the preemption level of the transaction.

Transaction commits:

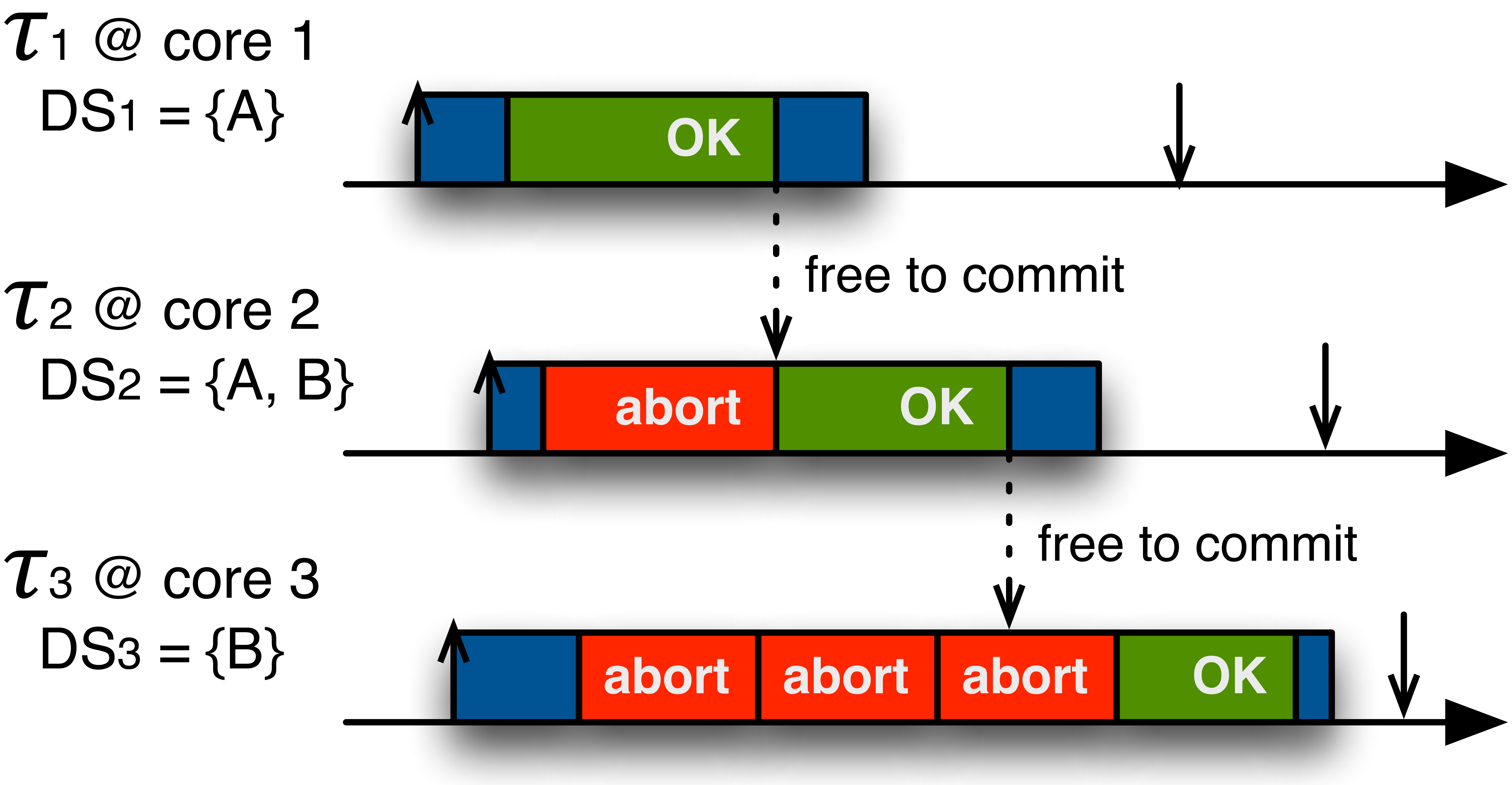
- Core ceiling is reset to zero.

SRP-TM scheduling decisions in short

- Job in front of ready queue has transaction:
  - Core ceiling is raised to the preemption level of this task.
  - Job with transaction in progress is executed on behalf of job in front of ready queue.
- Job in front of ready queue does not have transaction:
  - Preempt running job iff has **earlier absolute deadline** than running job, and **higher preemption level** than core ceiling.

Response time of a **transaction**

Response time of a transaction



τ_1 @ core 1

$DS_1 = \{A\}$

OK

free to commit

τ_2 @ core 2

$DS_2 = \{A, B\}$

abort

OK

free to commit

τ_3 @ core 3

$DS_3 = \{B\}$

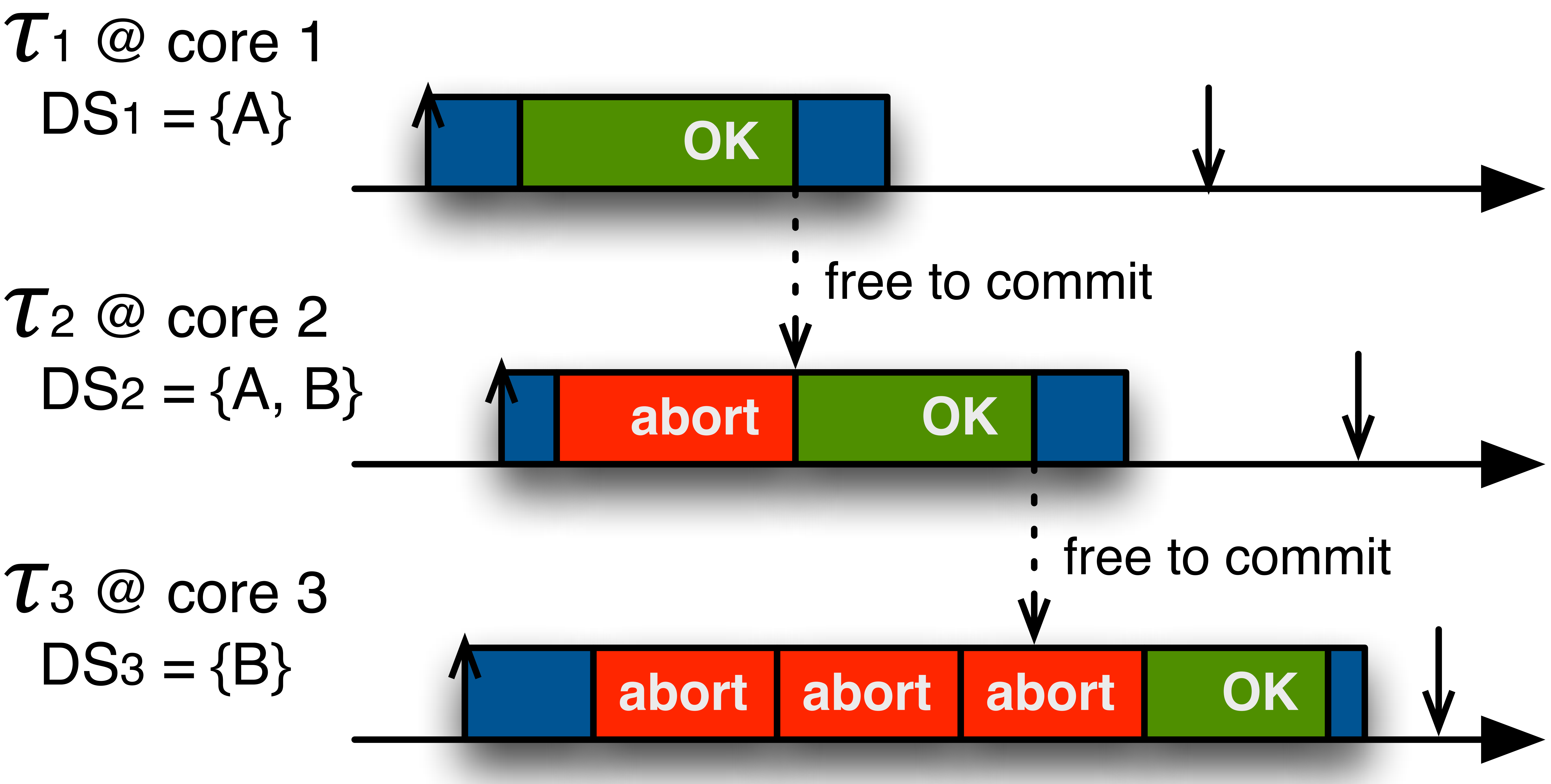
abort

abort

abort

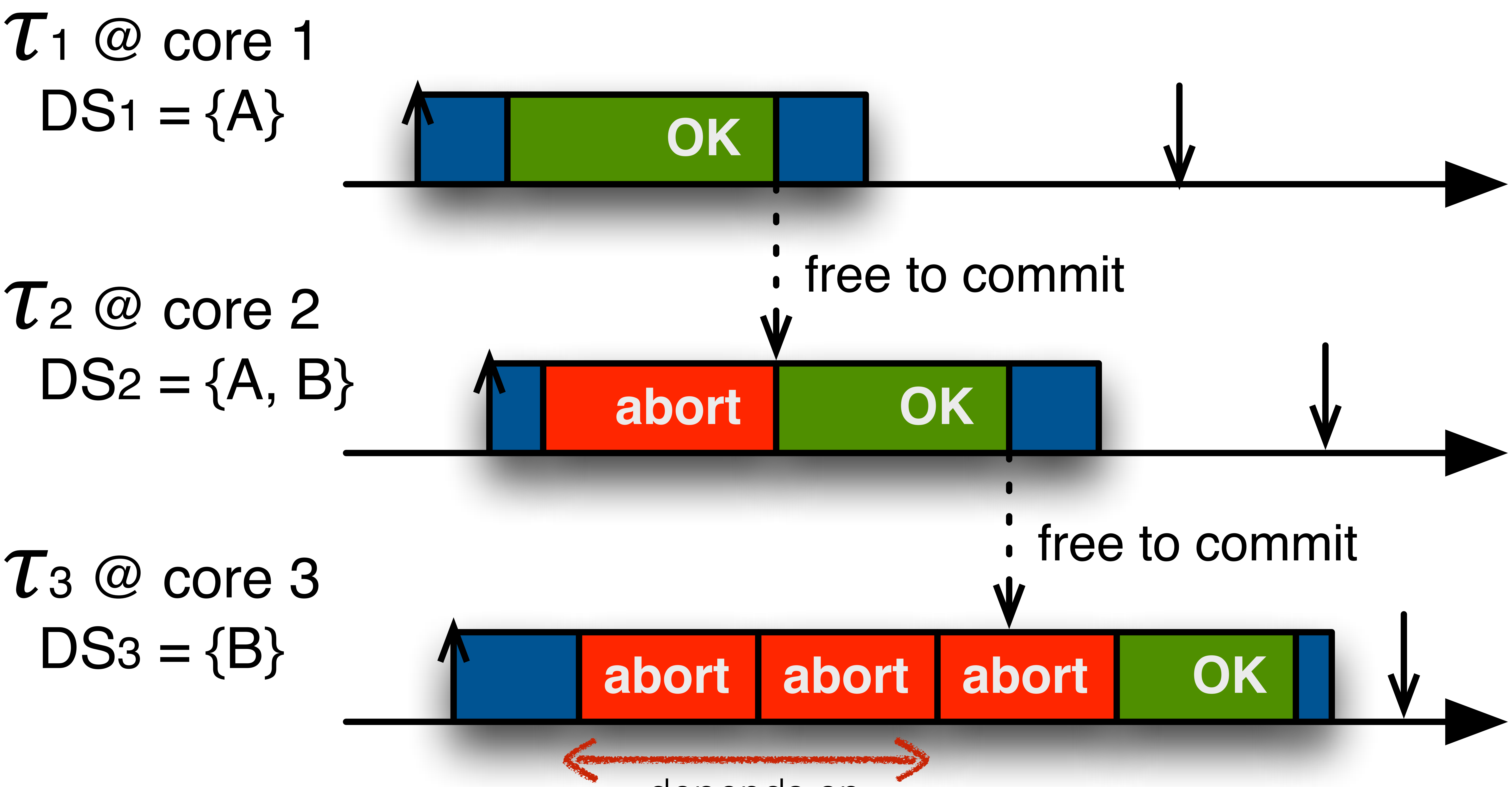
OK

Response time of a transaction



Transaction response time…

Response time of a transaction



τ_1 @ core 1

$DS_1 = \{A\}$

OK

free to commit

τ_2 @ core 2

$DS_2 = \{A, B\}$

abort

OK

free to commit

τ_3 @ core 3

$DS_3 = \{B\}$

abort

abort

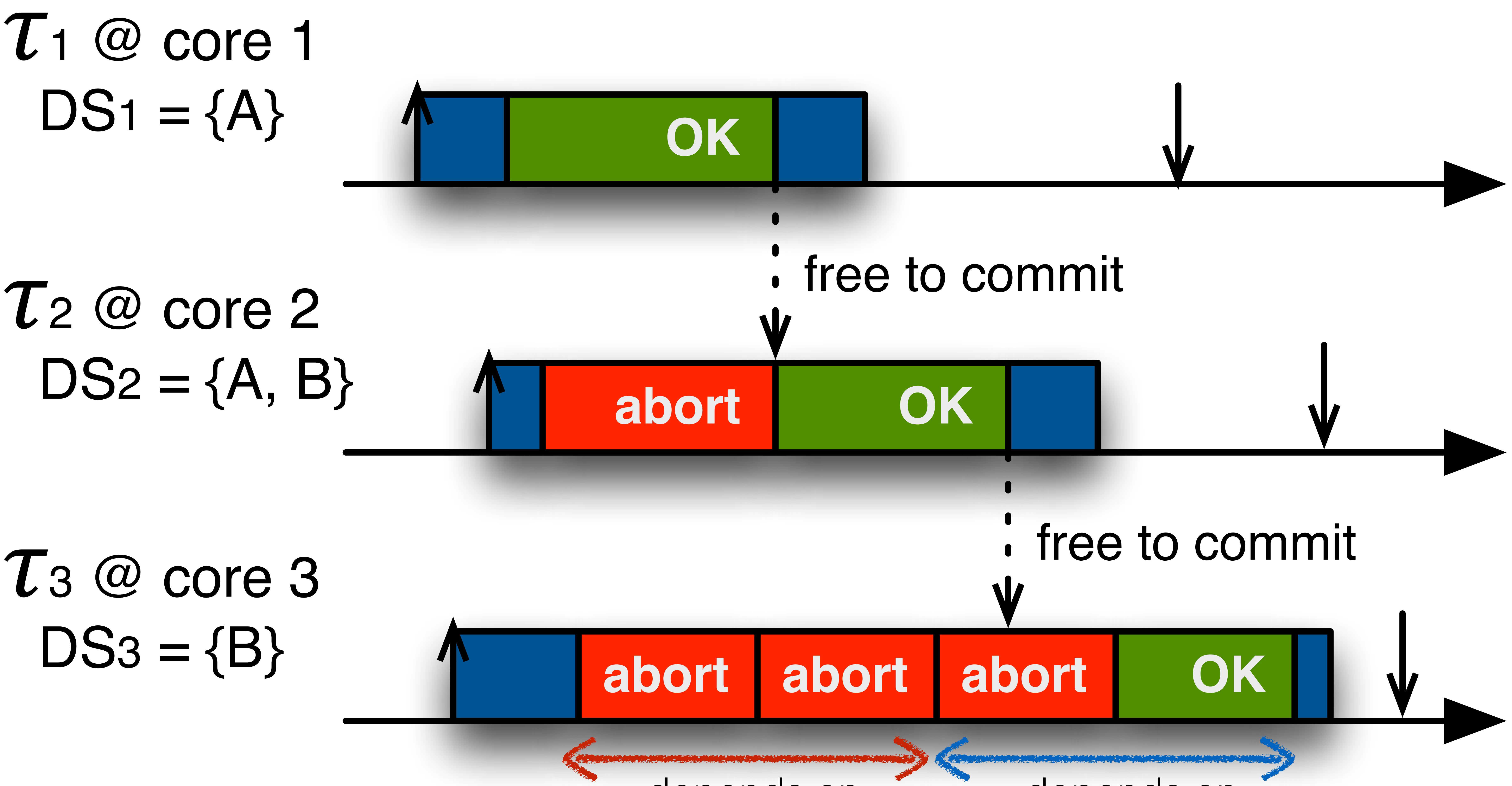
abort

OK

Transaction response time…

… depends on
parallel transactions

Response time of a transaction



τ_1 @ core 1

$DS_1 = \{A\}$

OK

free to commit

τ_2 @ core 2

$DS_2 = \{A, B\}$

abort

OK

free to commit

τ_3 @ core 3

$DS_3 = \{B\}$

abort

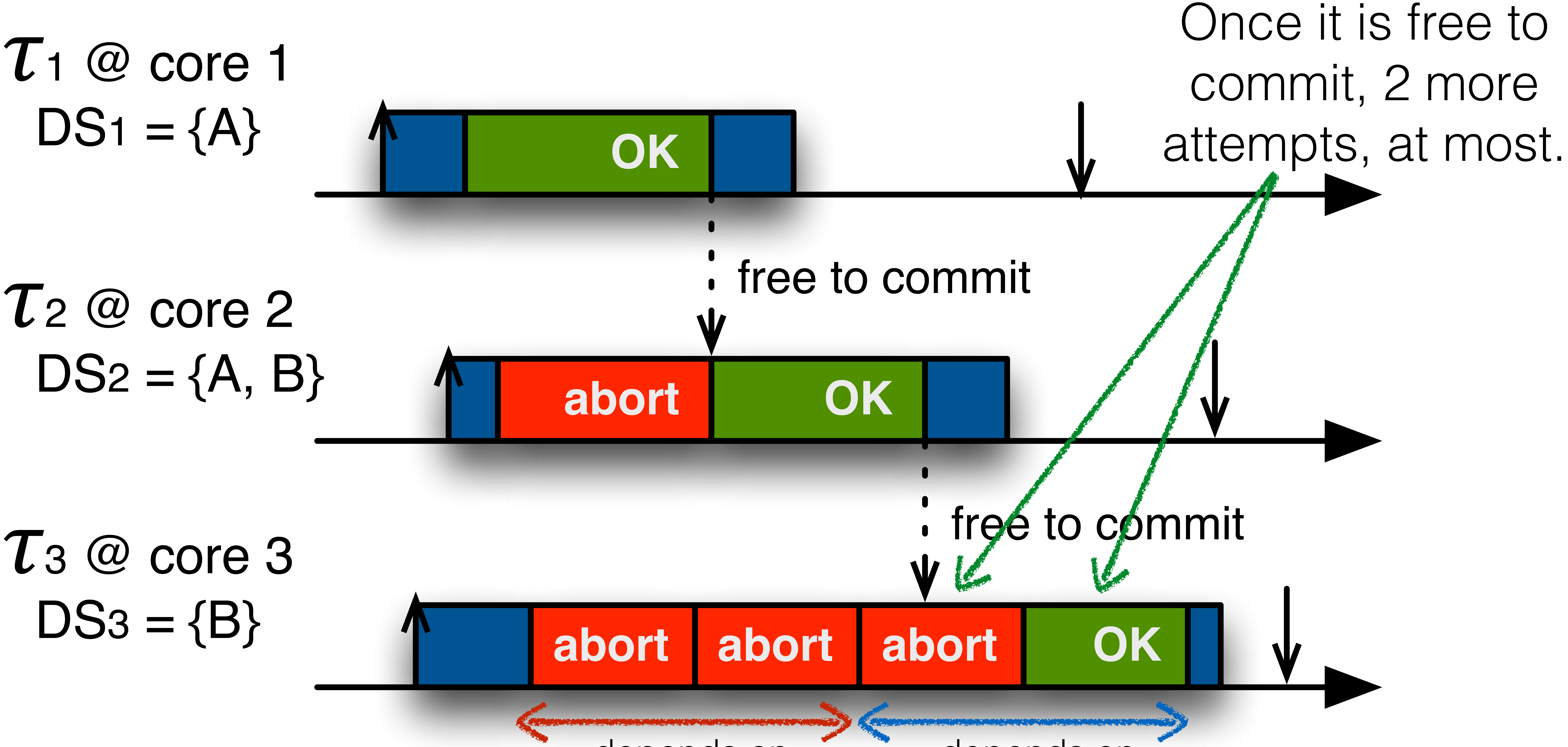
abort

abort

OK

Transaction response time… … depends on parallel transactions … depends on intra-core interference

Response time of a transaction



τ_1 @ core 1
$DS_1 = \{A\}$

τ_2 @ core 2
$DS_2 = \{A, B\}$

τ_3 @ core 3
$DS_3 = \{B\}$

Once it is free to commit, 2 more attempts, at most.

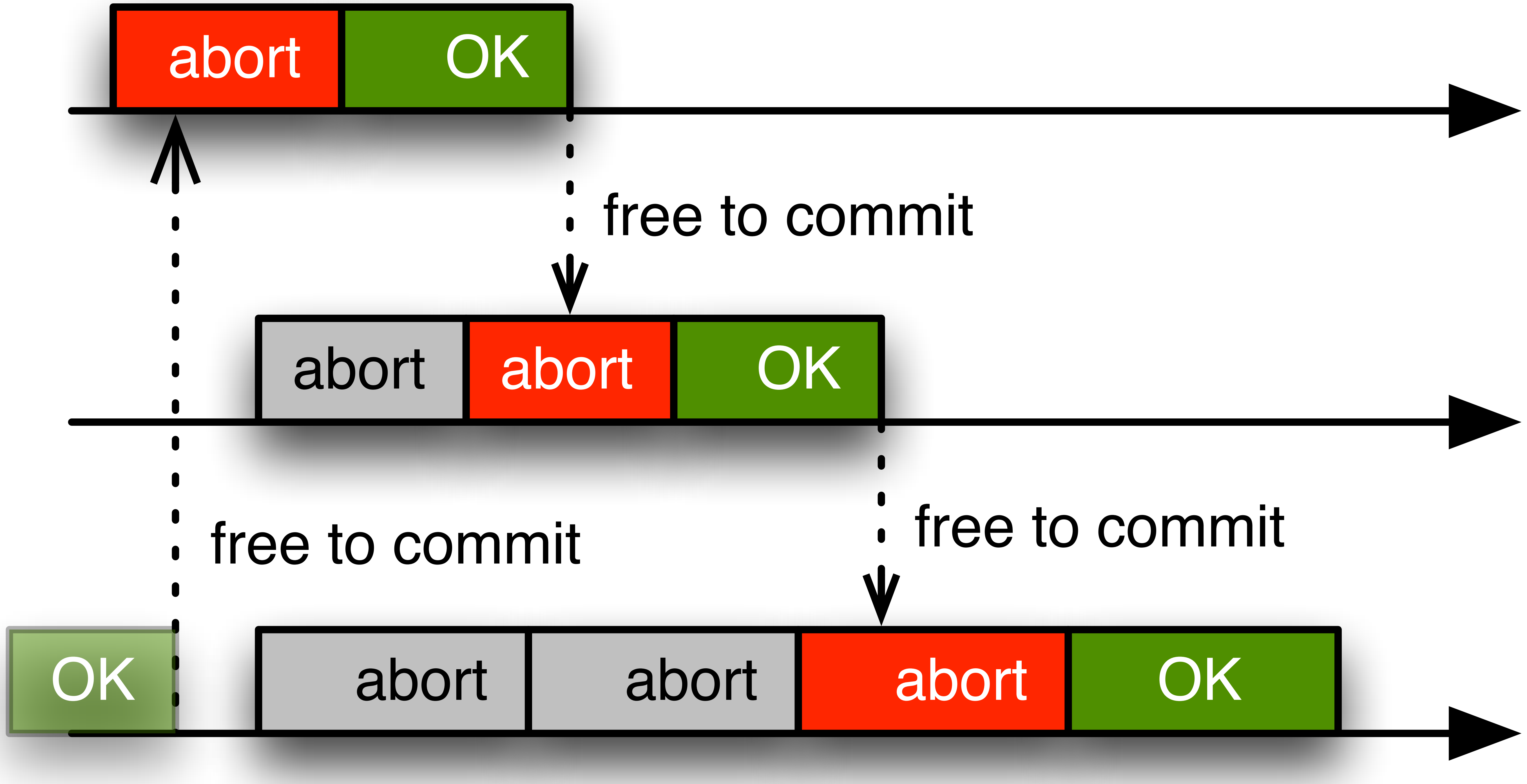
Transaction response time…

… depends on parallel transactions

… depends on intra-core interference

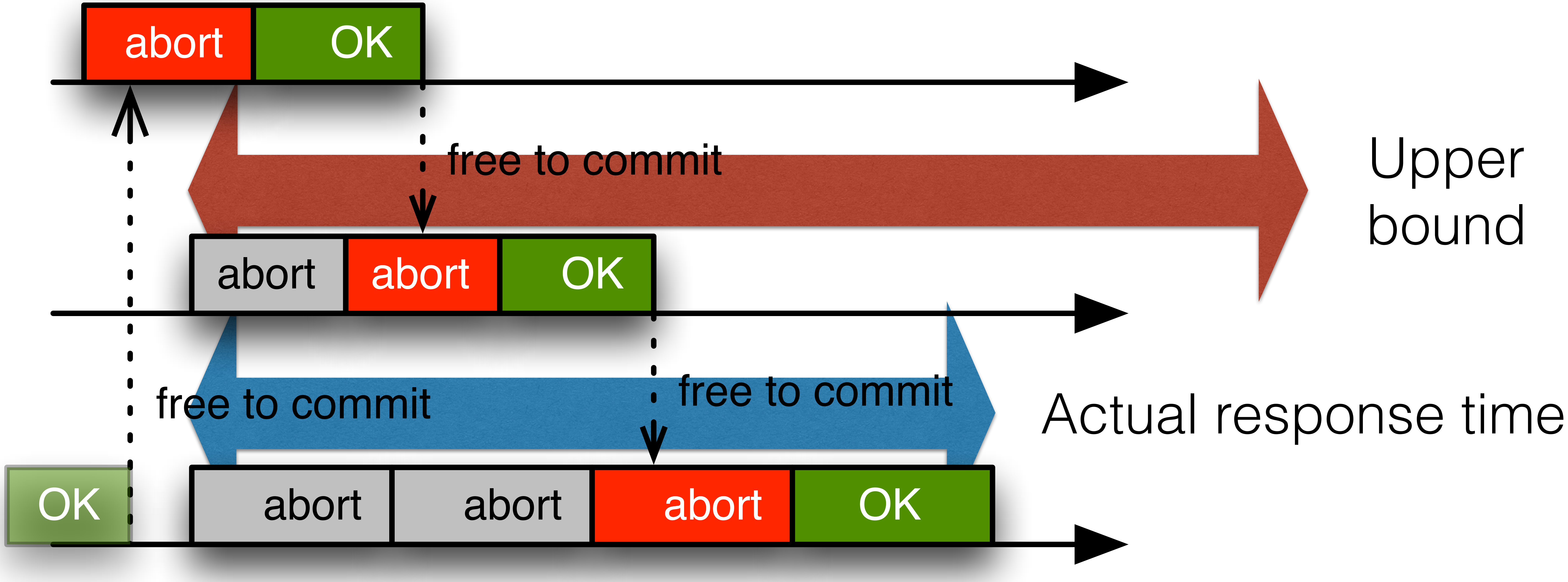
Response time of a transaction

- The response time of the last transaction in a sequence of transactions is upper bounded by the sum of the response time of the last two attempts, for each transaction in the sequence.



Response time of a transaction

- The response time of the last transaction in a sequence of transactions is upper bounded by the sum of the response time of the last two attempts, for each transaction in the sequence.

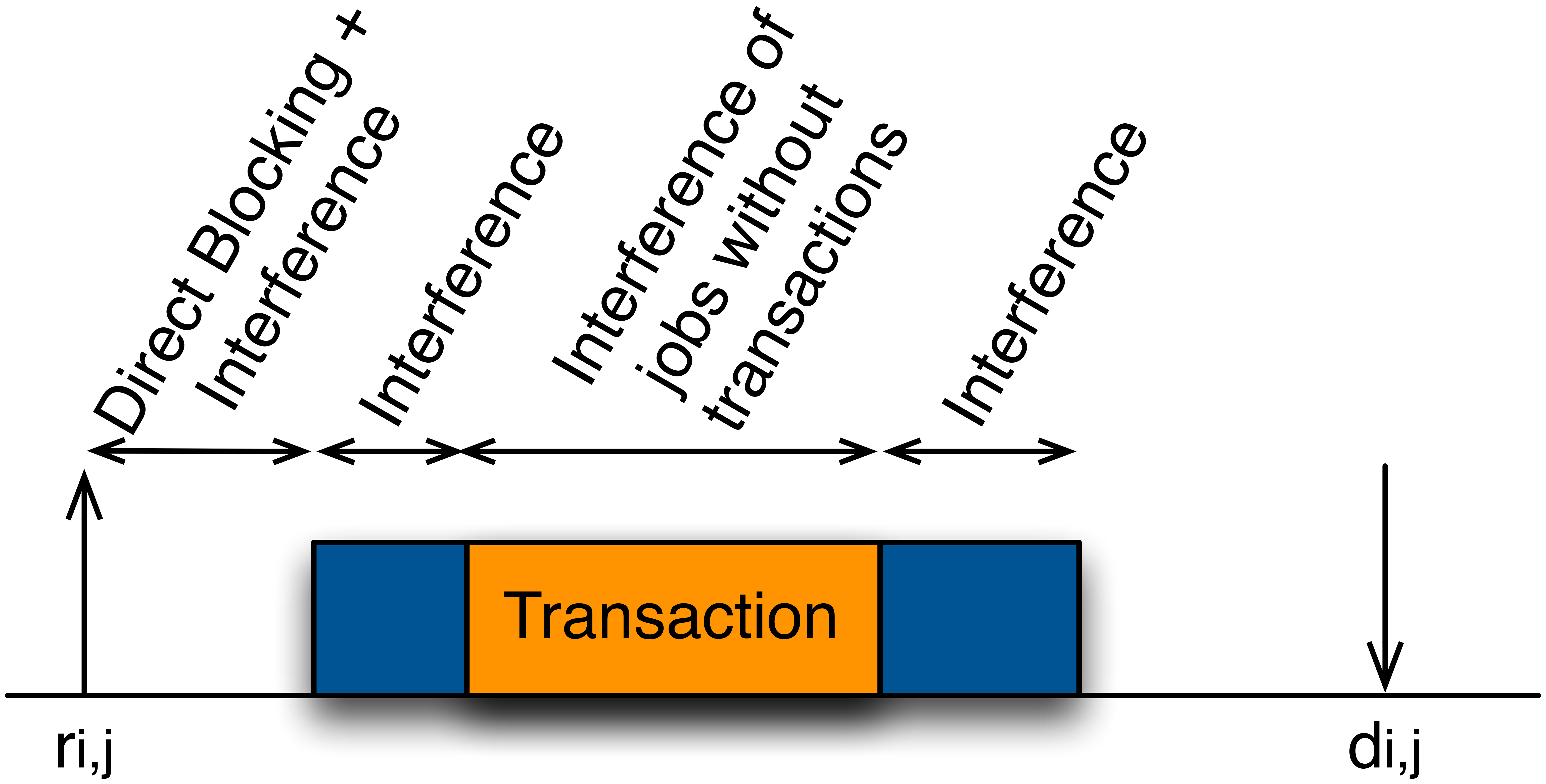


Response time of a transaction

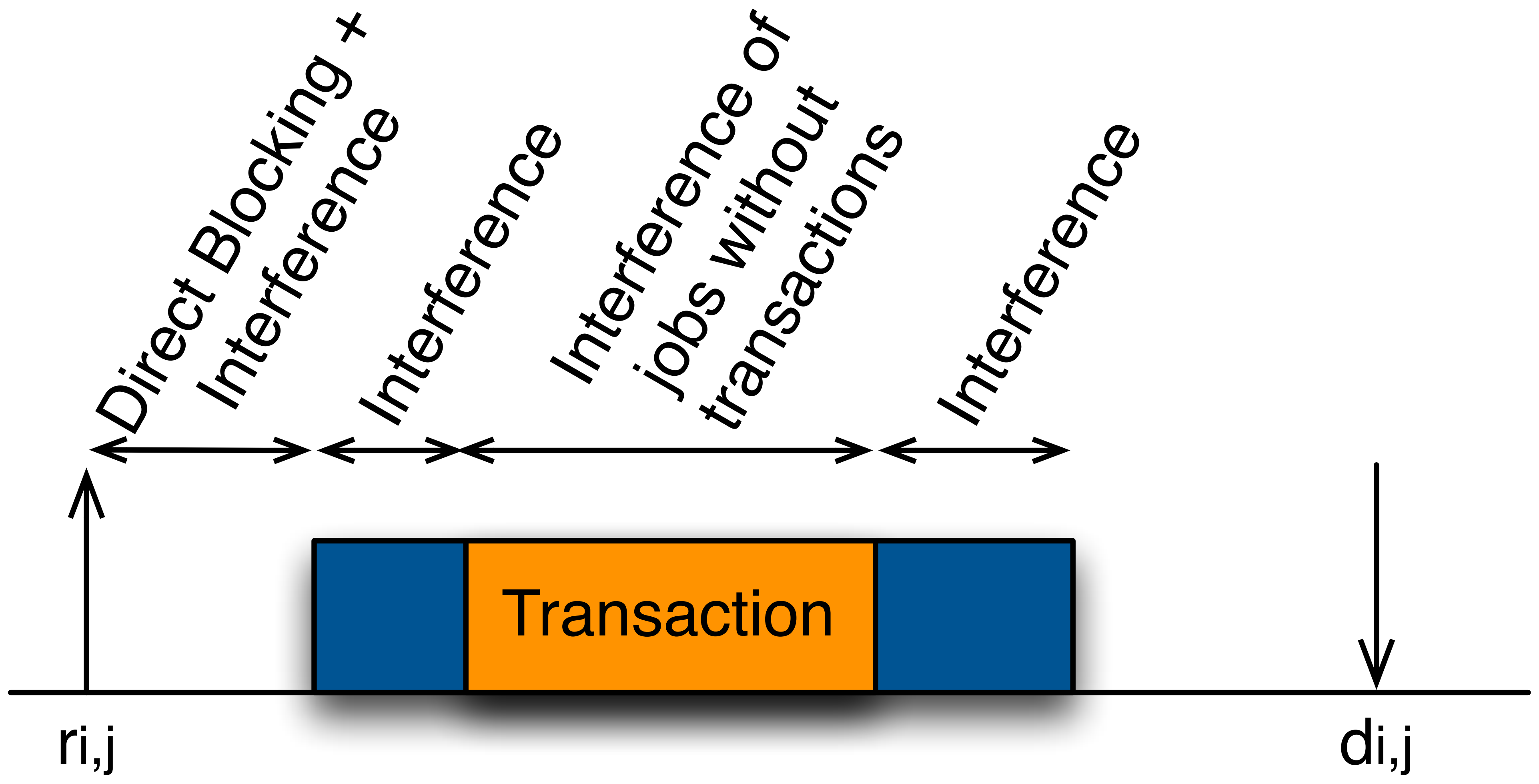
- The response time of the last two transactions depends exclusively on intra-core interference:
  - **IT CAN BE ANALYTICALLY UPPER BOUNDED!**
- Maximum response time of a transaction…
  - Determine every possible sequence, sum response times of last two attempts and choose the maximum value… **COMBINATIONAL ORDER!!!**
  - For every processor, choose the maximum response time of last two attempts of a transaction that belongs to the same contention group, and sum them all… **PESSIMISTIC, but LINEAR ORDER!**

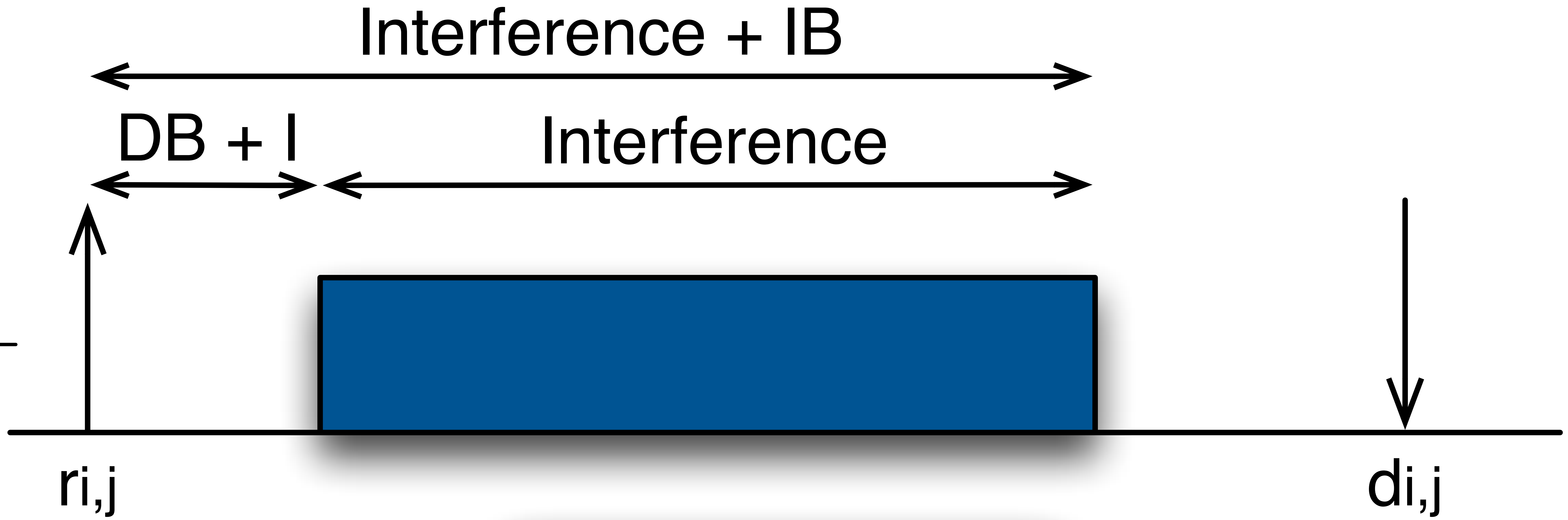
Response time of a **task**

Blocking and interference


Direct Blocking + Interference
Interference
Interference of jobs without transactions
Interference
Transaction
ri,j
di,j

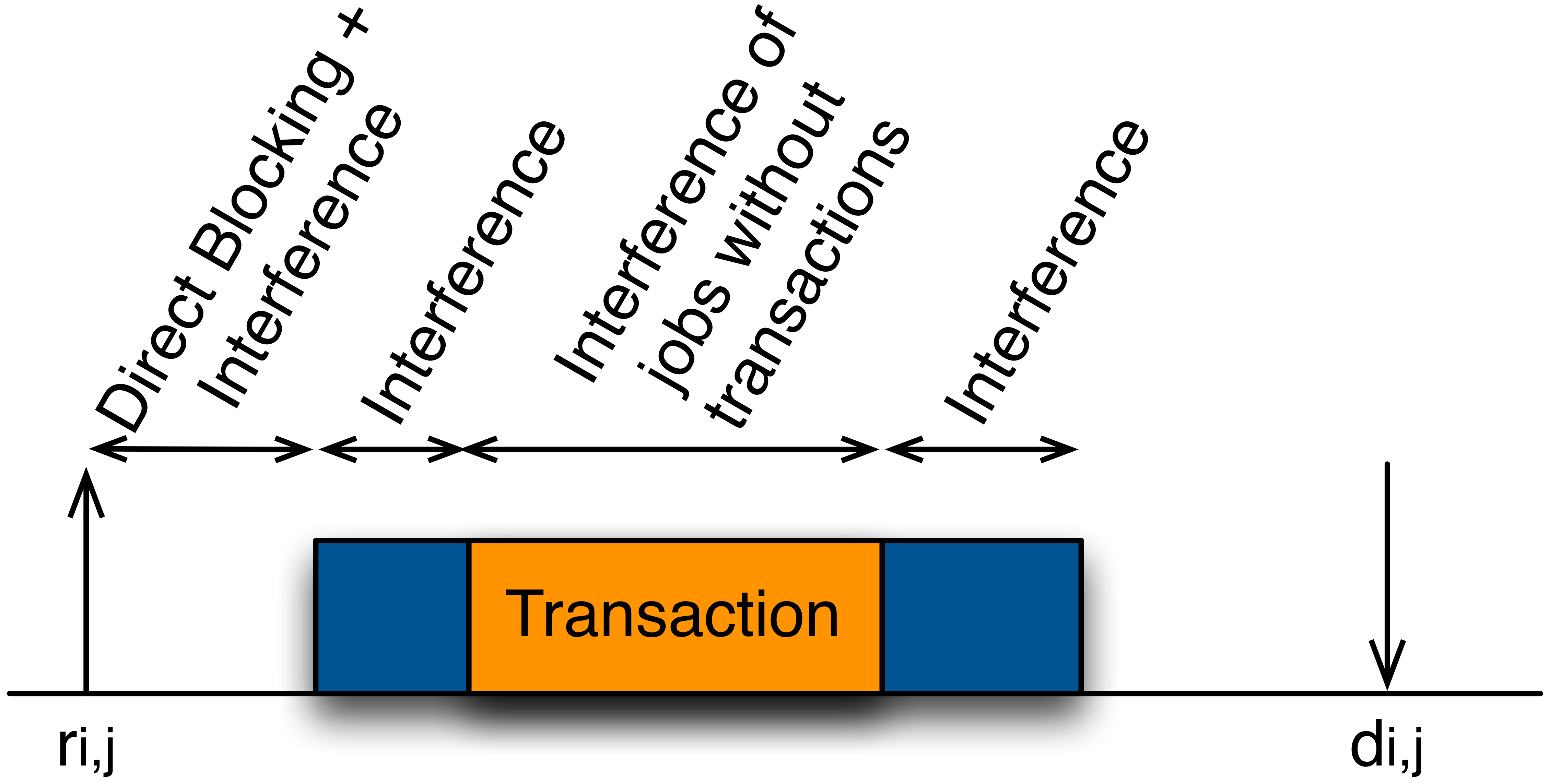
Blocking and interference

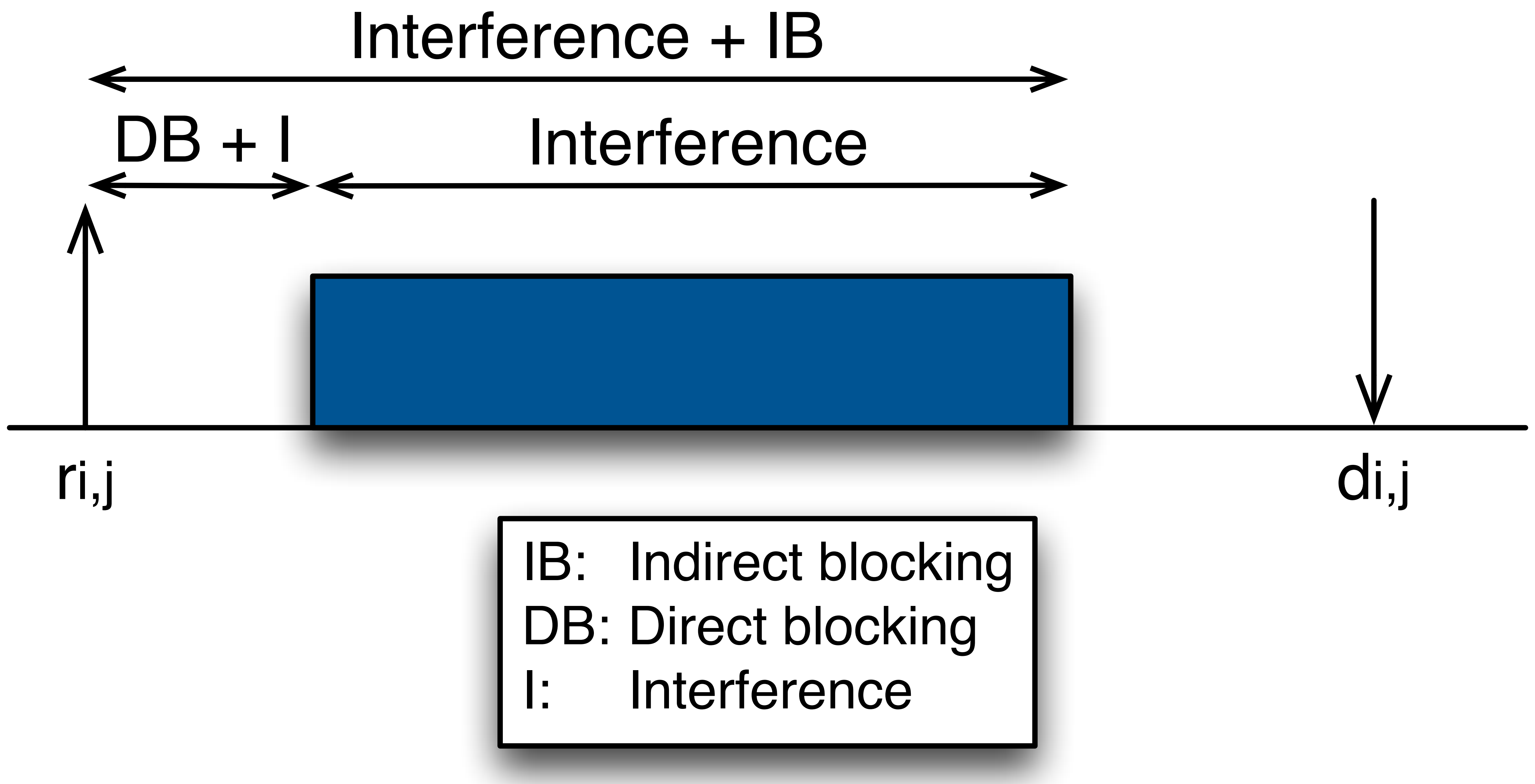

Direct Blocking + Interference
Interference
Interference of jobs without transactions
Interference
Transaction
ri,j
di,j



IB: Indirect blocking
DB: Direct blocking
I: Interference

Blocking and interference


Direct Blocking + Interference
Interference
Interference of jobs without transactions
Interference
Transaction
ri,j
di,j



IB: Indirect blocking
DB: Direct blocking
I: Interference

Simulation results

Simulation conditions

- Scheduling policies:
    - pure P-EDF
    - NPUC
    - NPDA
    - SRP-TM
    - FLMP

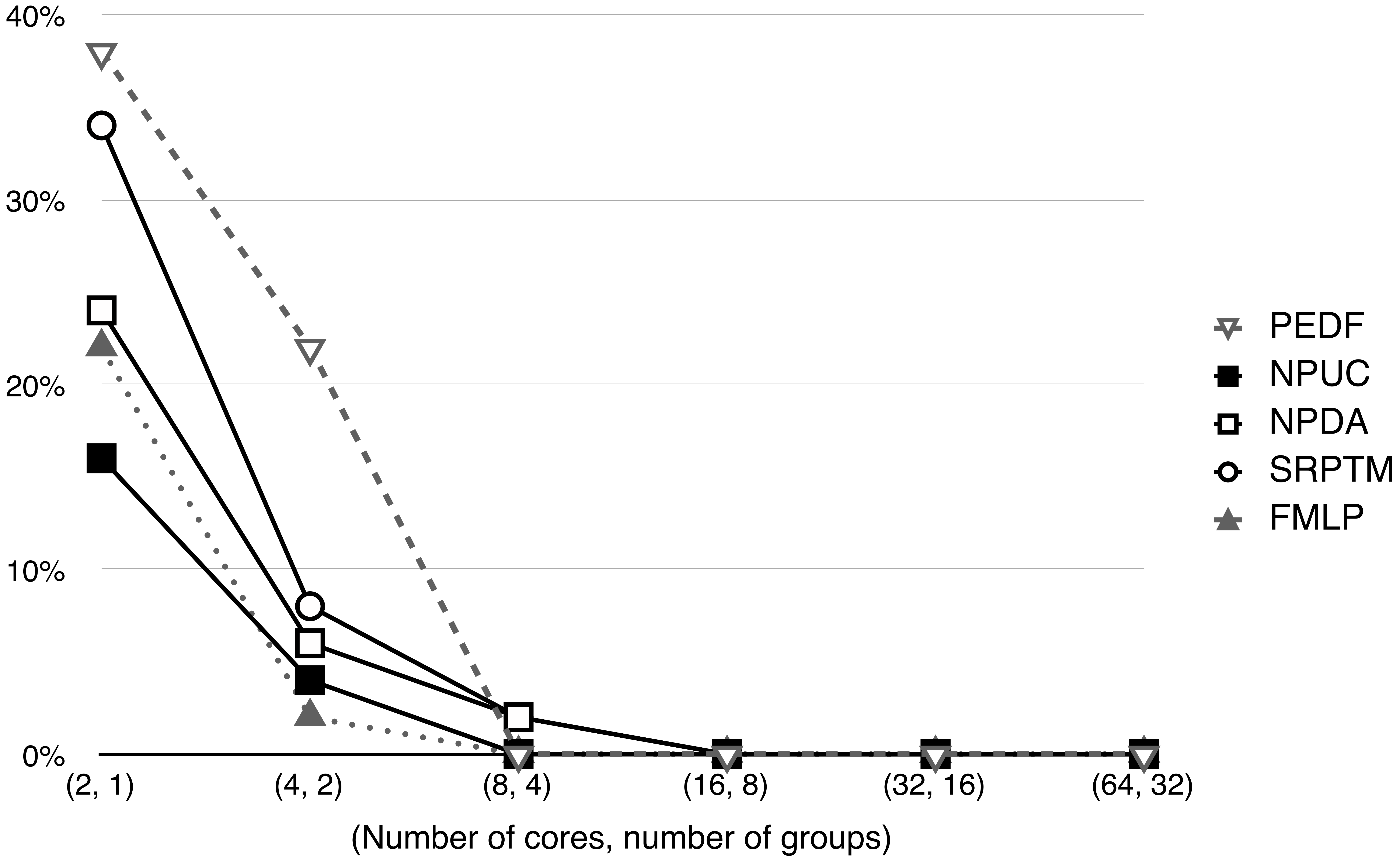
Simulation conditions

- Experiment 1: varying system size
  - Variable number of cores: $m \in \{2, 4, 8, 16, 32, 64\}$
  - Number of transactional objects linear with m: $p \in \{5, 10, 20, 40, 80, 160\}$, so each object is accessed by 3 task, on average.
  - Number of contention groups linear with m: $g \in \{1, 2, 4, 8, 16, 32\}$, so each group maintains the same size and the same expected number of tasks.

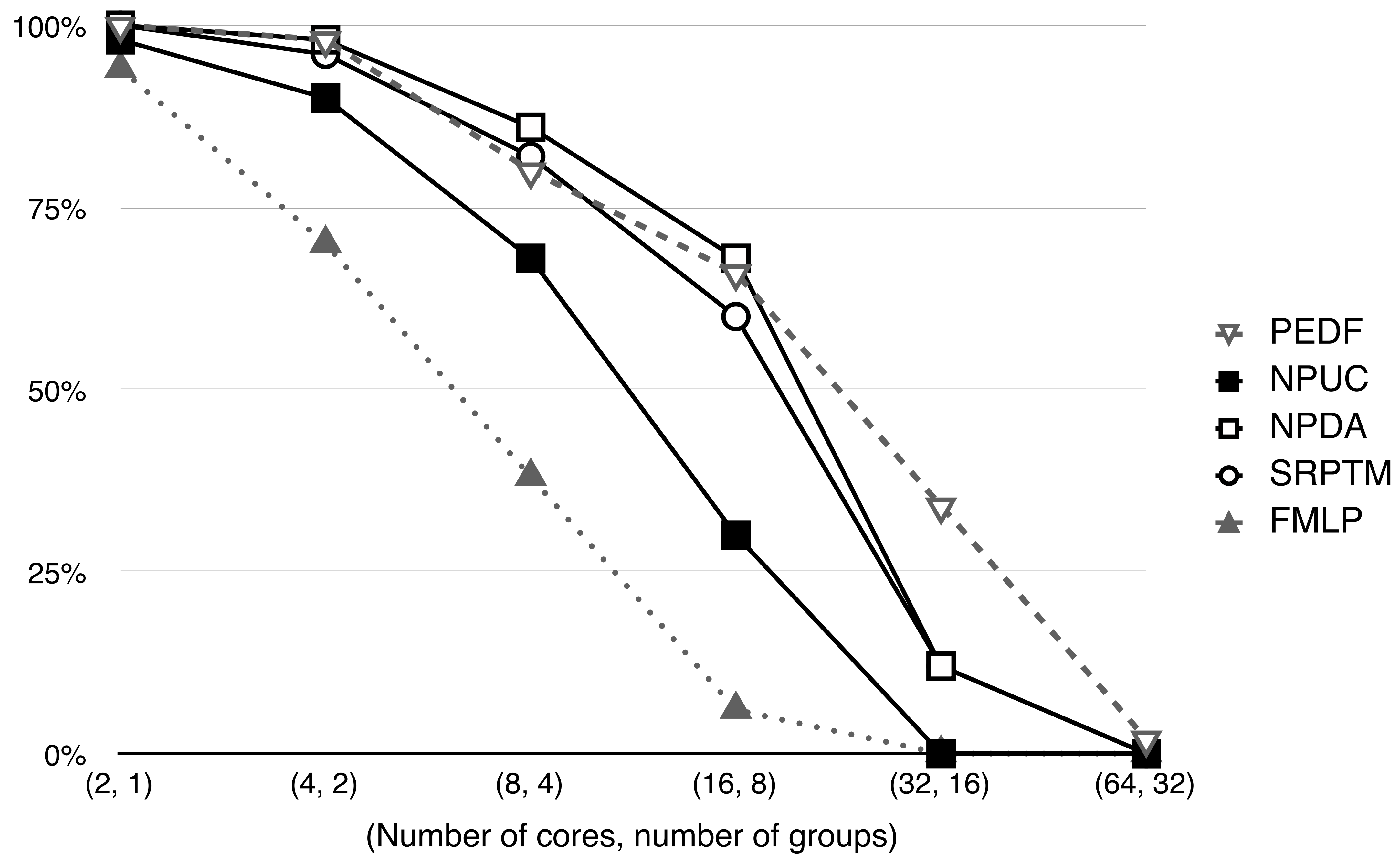
Simulation conditions

- Experiment 2: varying size of contention groups
  - Constant number of cores: m = 64.
  - Constant number of transactional objects linear: p = 160.
  - Variable number of contention groups: $g \in \{1, 2, 4, 8, 16, 32\}$, so to observe the effects of granularity of contention groups for systems with same size.

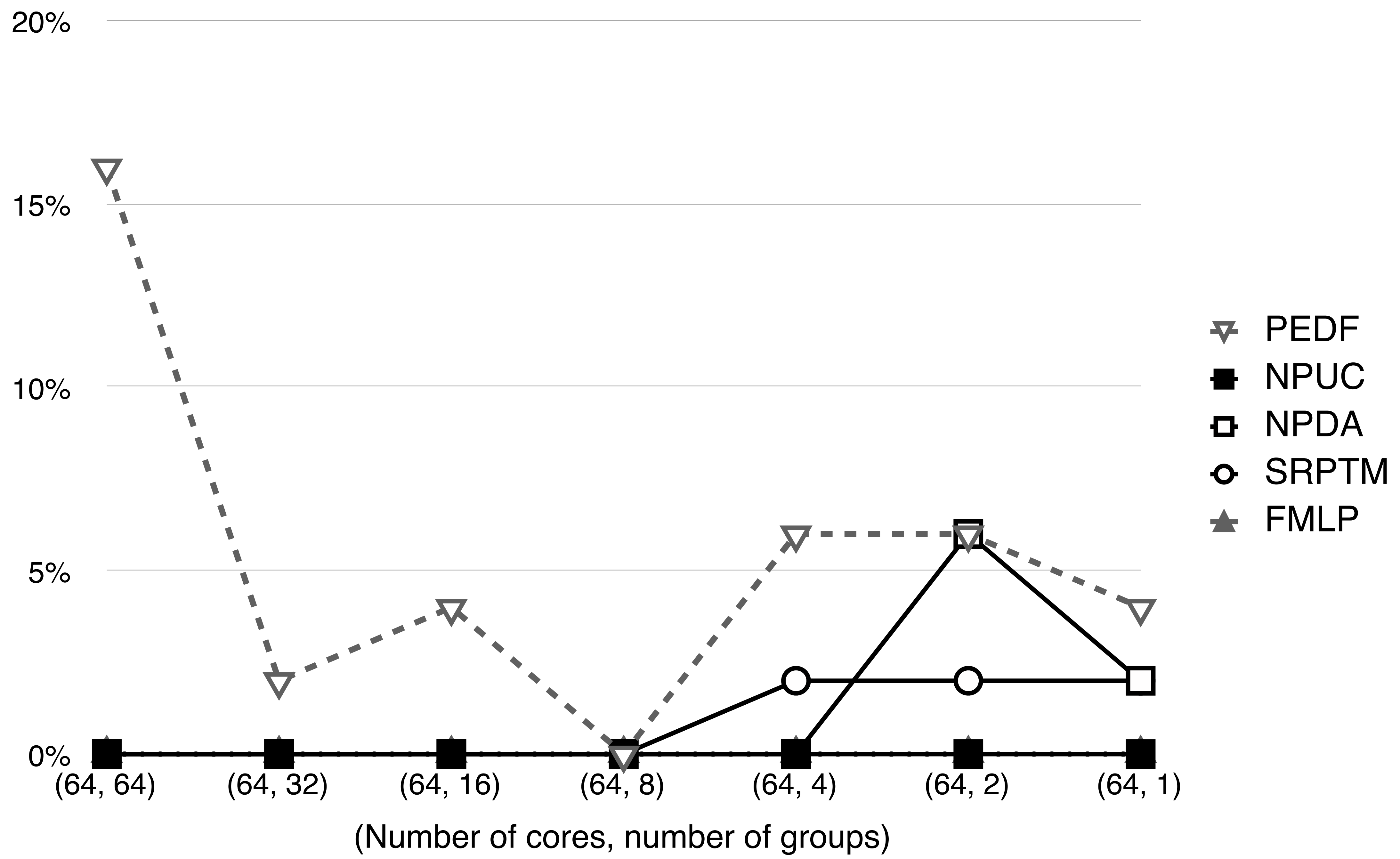
Feasibility (experiment 1)



Feasibility
(transaction execution time limited to 20 units)

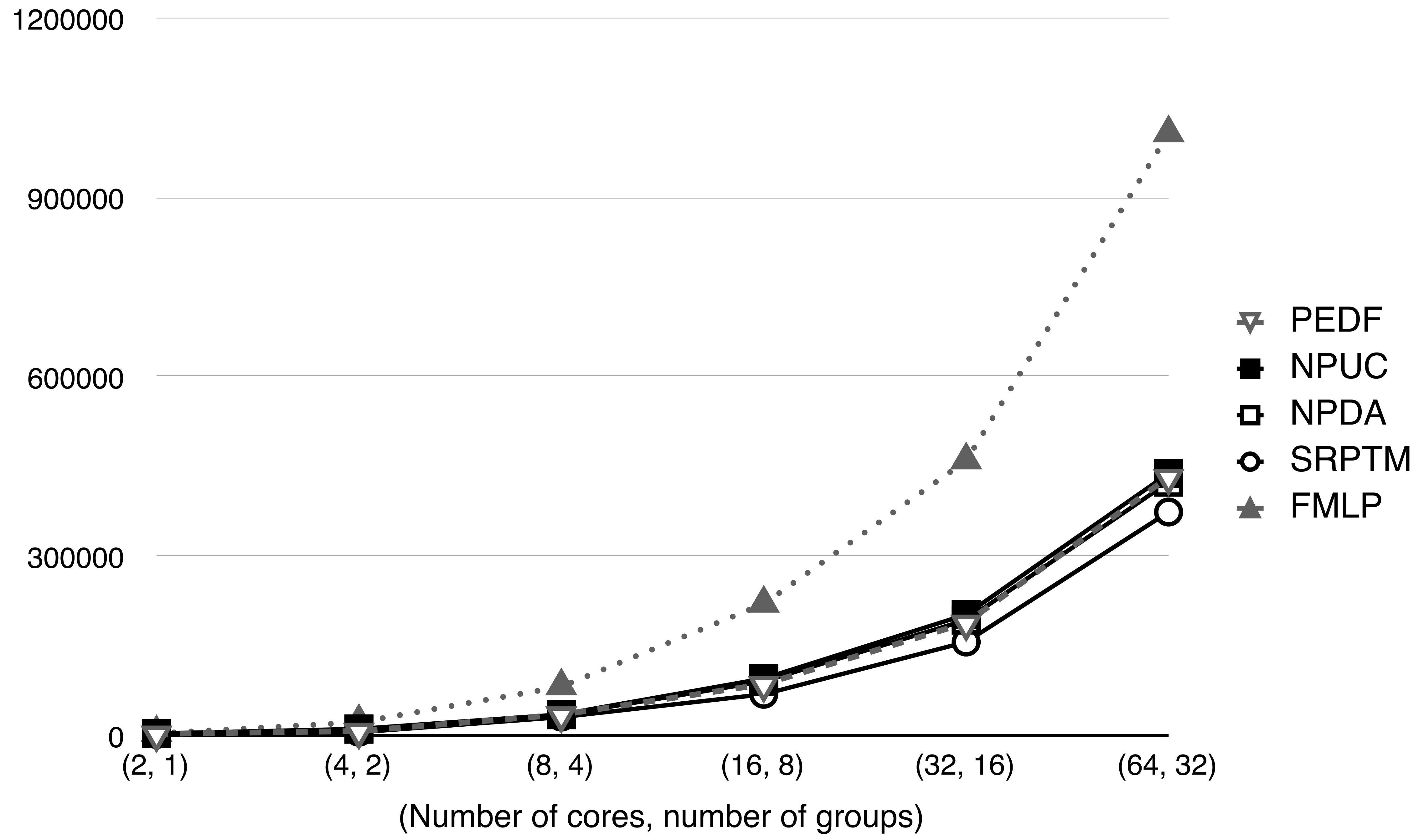


Experiment 1

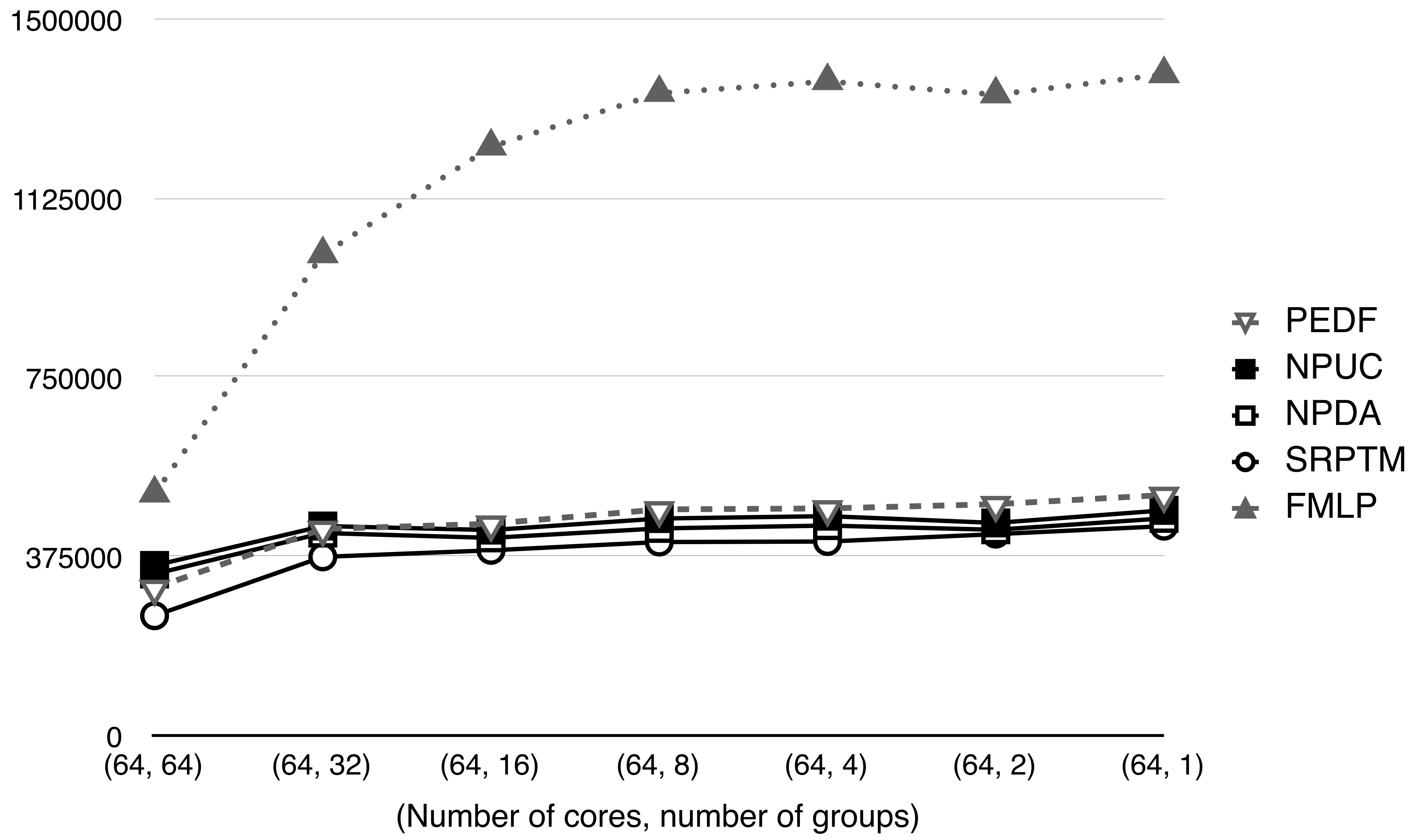


Experiment 2

Missed deadlines

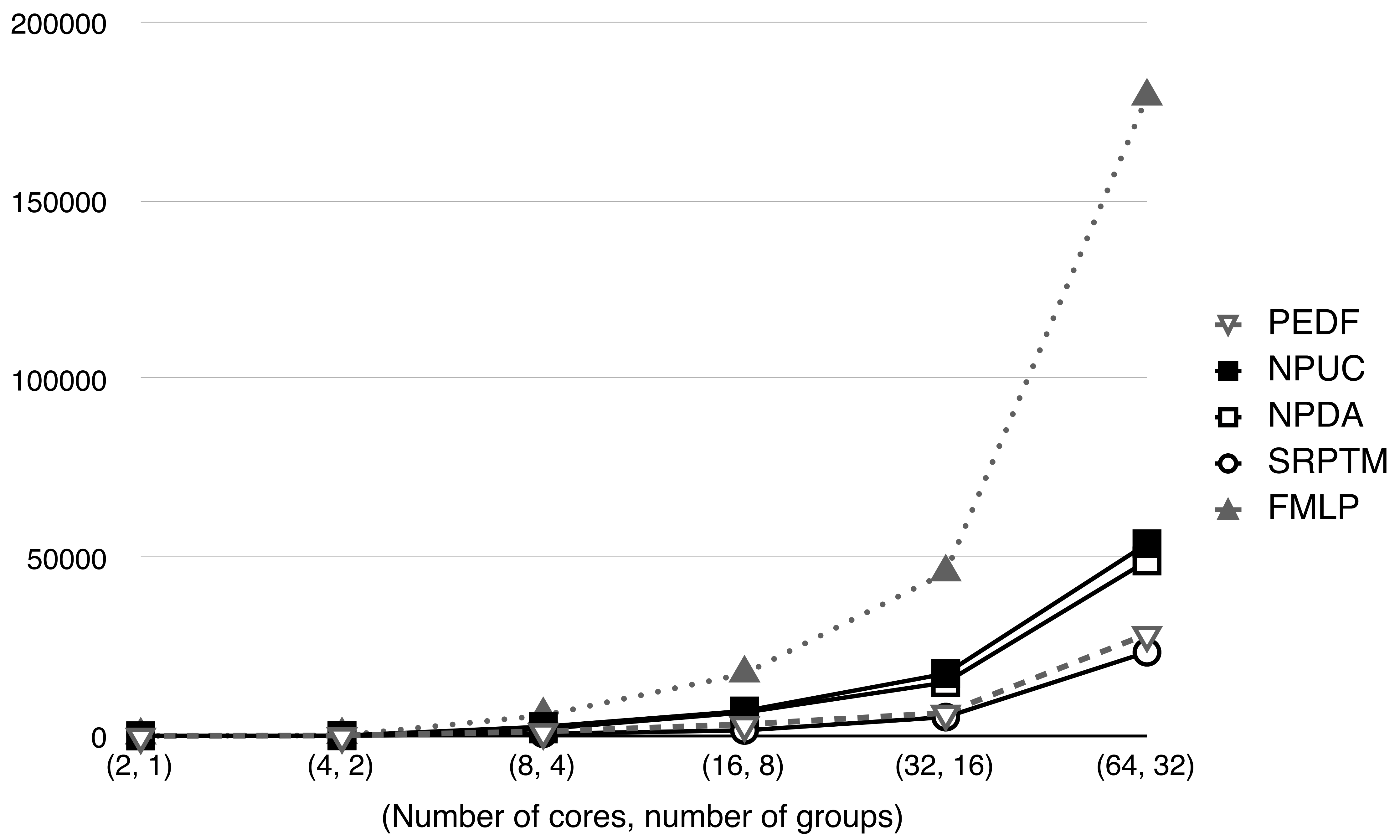


Experiment 1

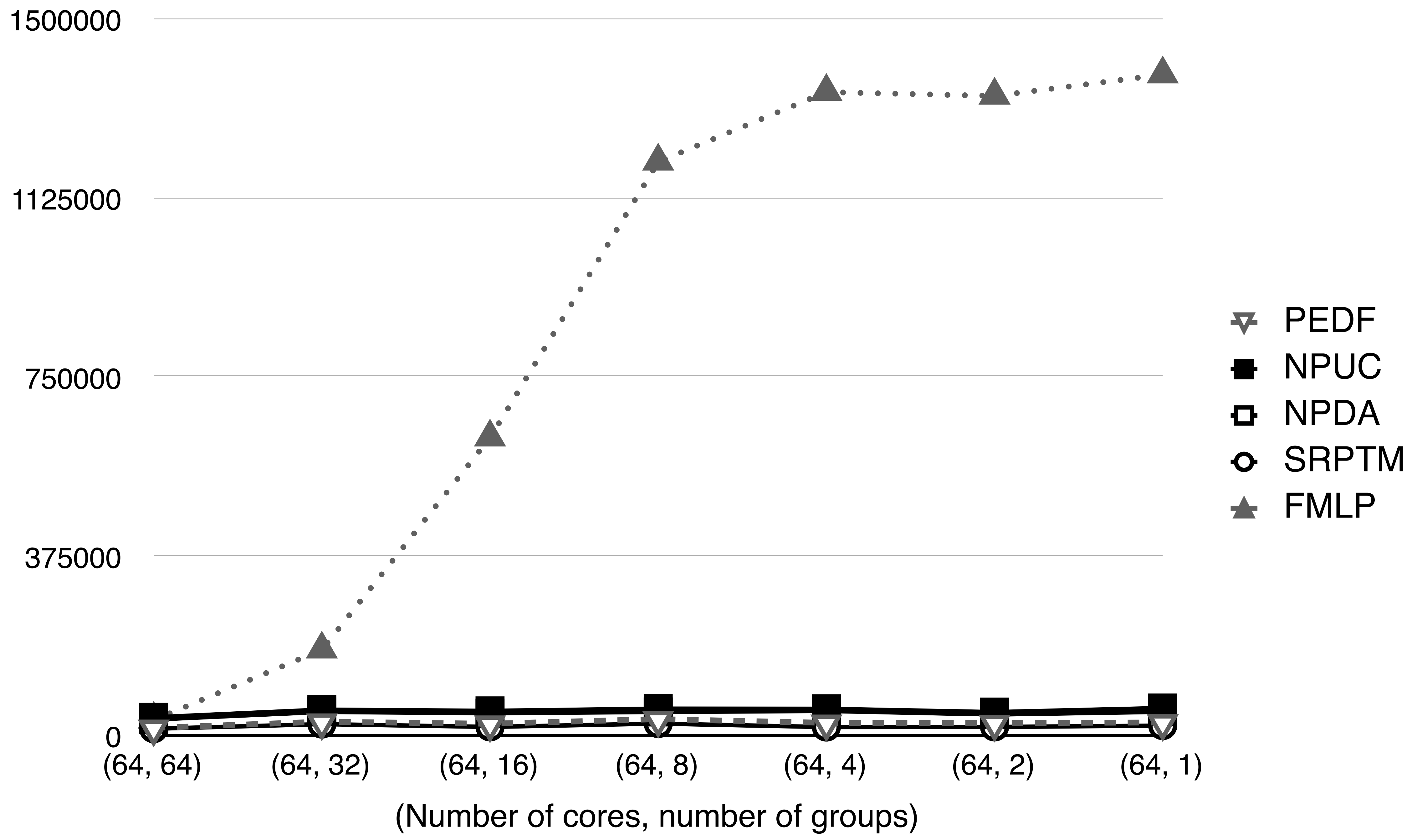


Experiment 2

Missed deadlines (transaction execution time limited to 20 units)

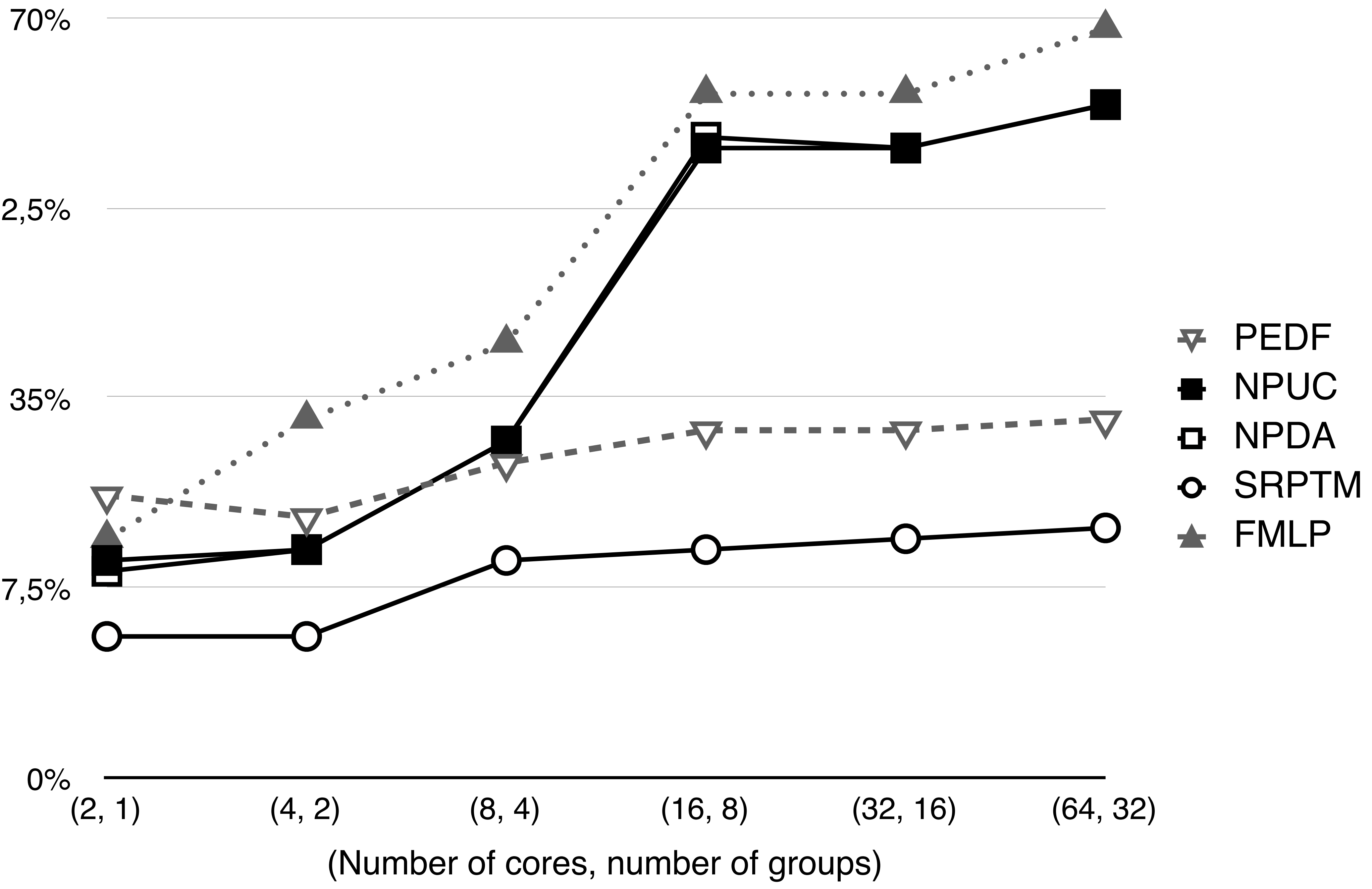


Experiment 1

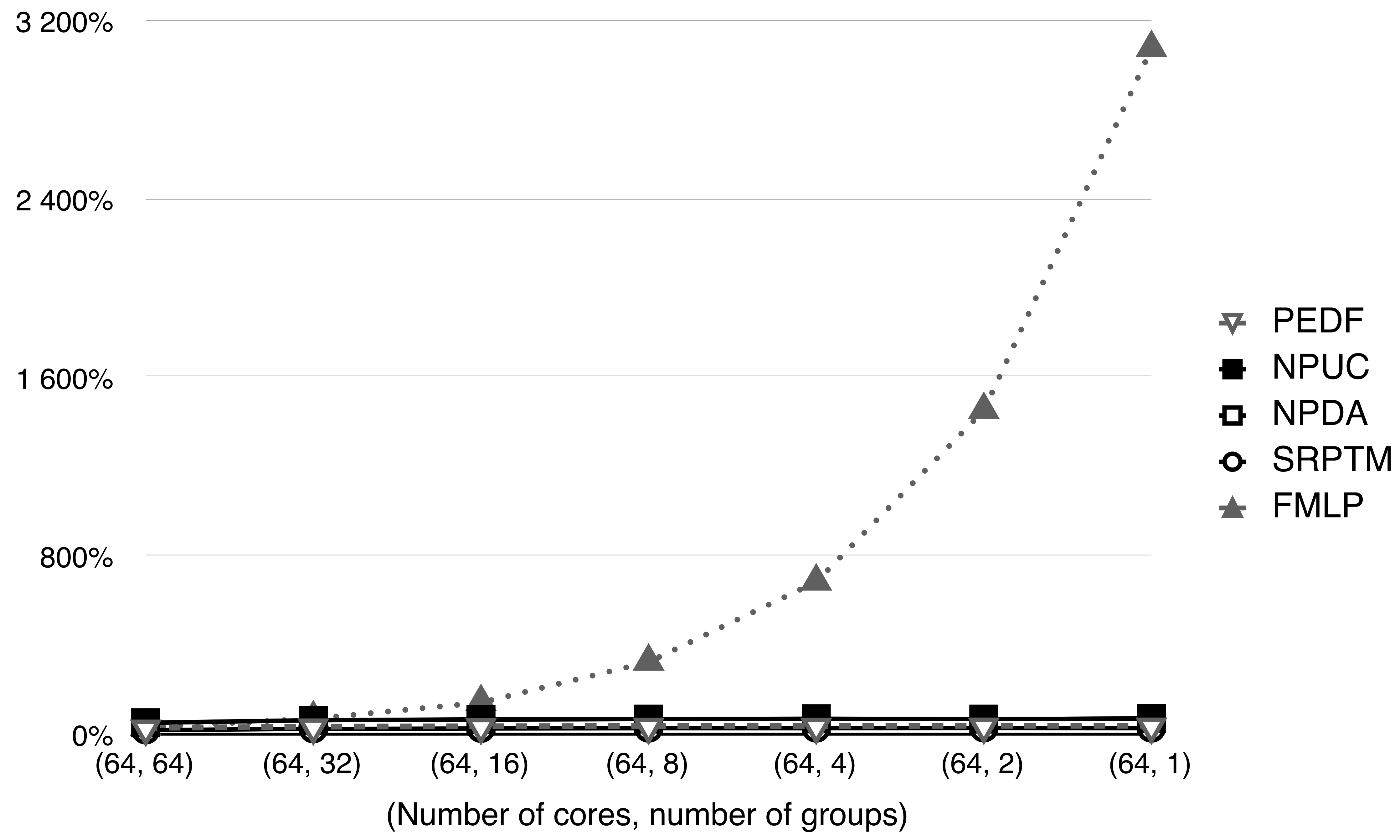


Experiment 2

Atomic section overheads

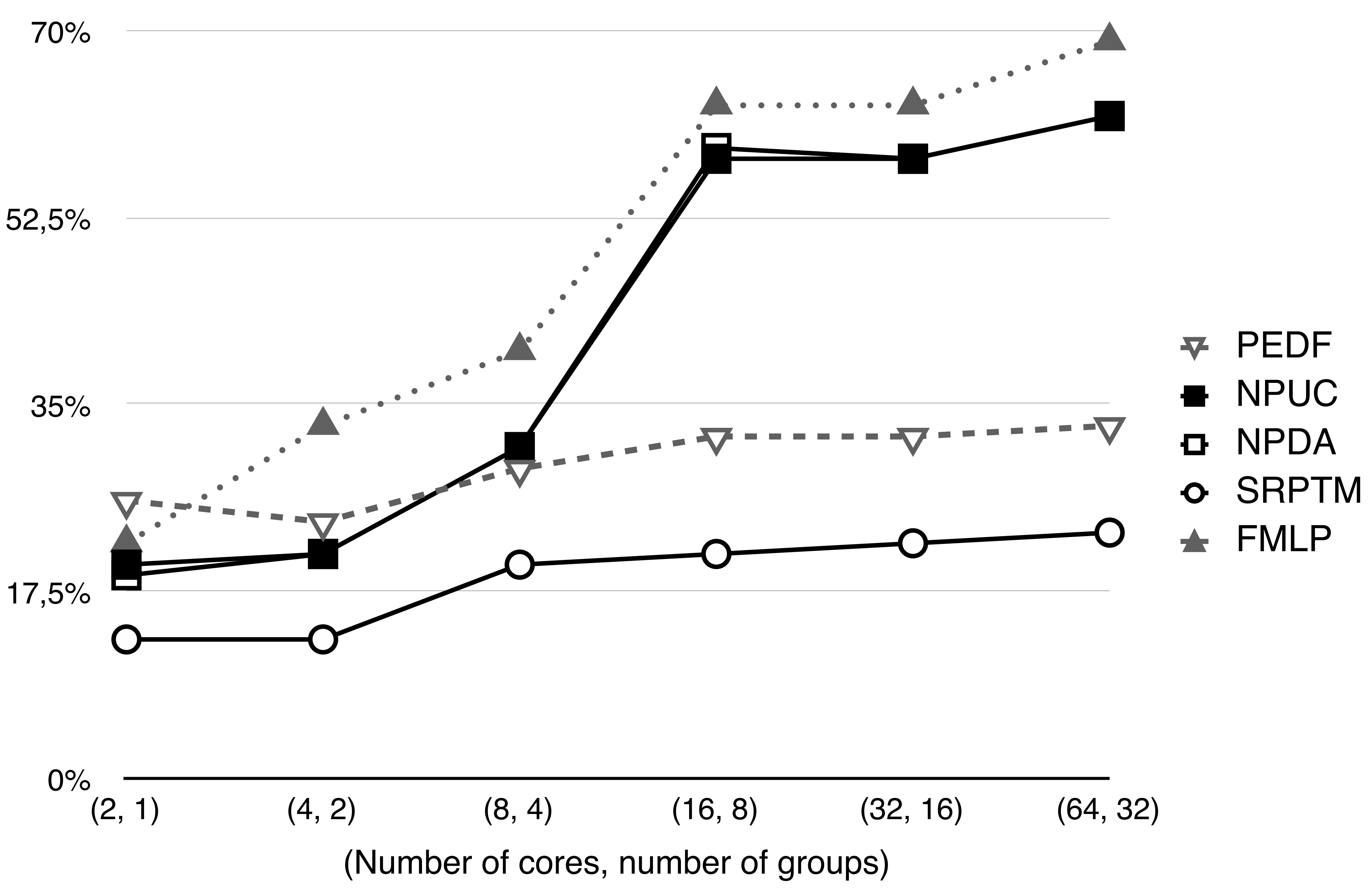


Experiment 1

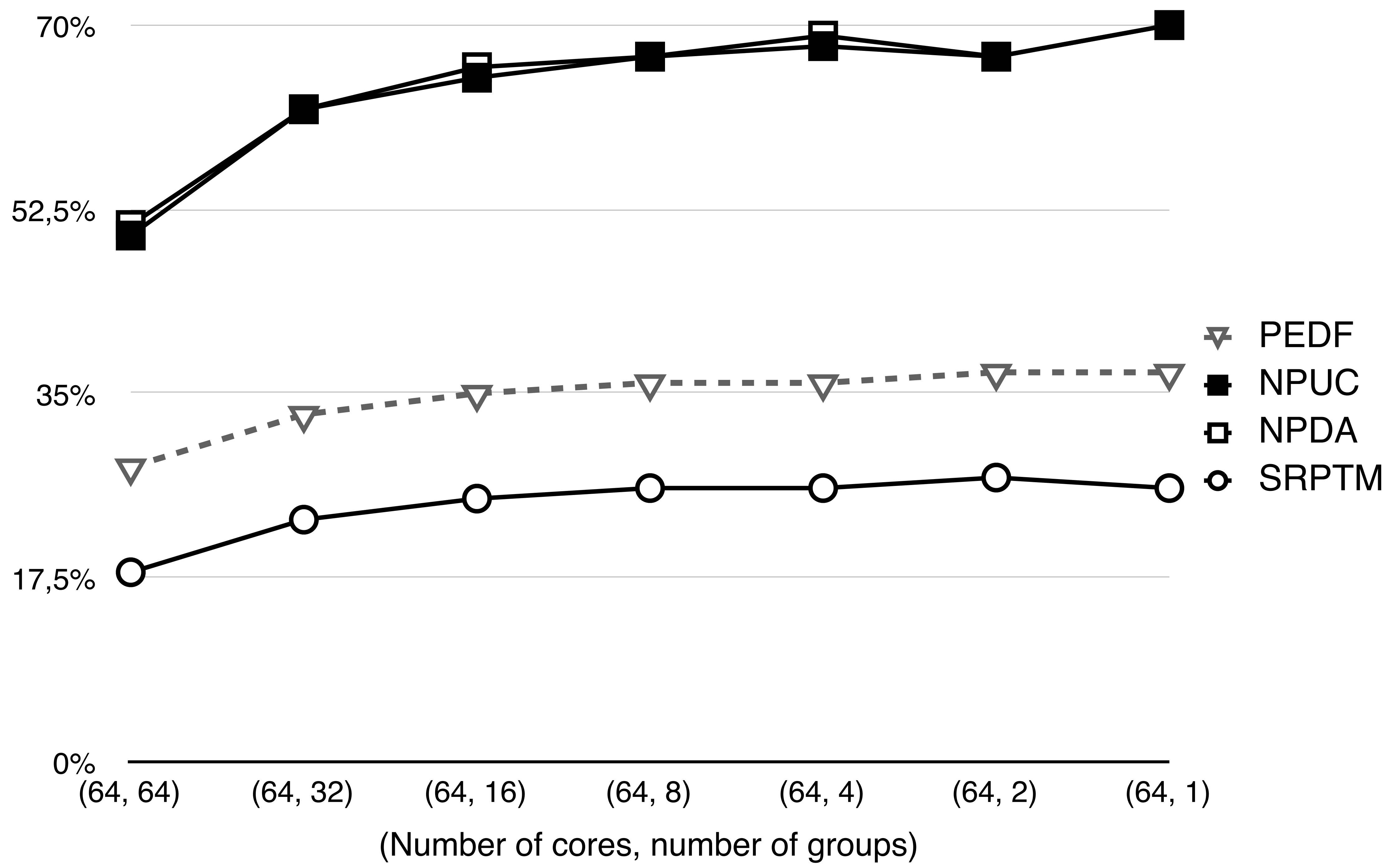


Experiment 2

Atomic section overheads



Experiment 1



Experiment 2

Wrapping up

Conclusion (1/2)

- FIFO serialisation is the predictable and fair.
- Scheduling has an effect on the performance of transactions.
  - SRP-TM extends P-EDF when a transaction is in progress.
    - Takes into account **possible** concurrent parallel transactions with earlier deadlines, **without sharing** scheduling data between cores.
    - Allows jobs with earlier deadlines to **preempt** or **speed up** a transaction in progress.

Conclusion (2/2)

- We provide an analytical method to upper bound the response time of transactions under SRP-TM.

- We provide an analytical method to upper bound the response time of tasks under SRP-TM.

That’s it! Thanks!
Questions?