MAX : A Multicore-Accelerated File System for Flash Storage

Xiaojian Liao, Youyou Lu, Erci Xu, Jiwu Shu,

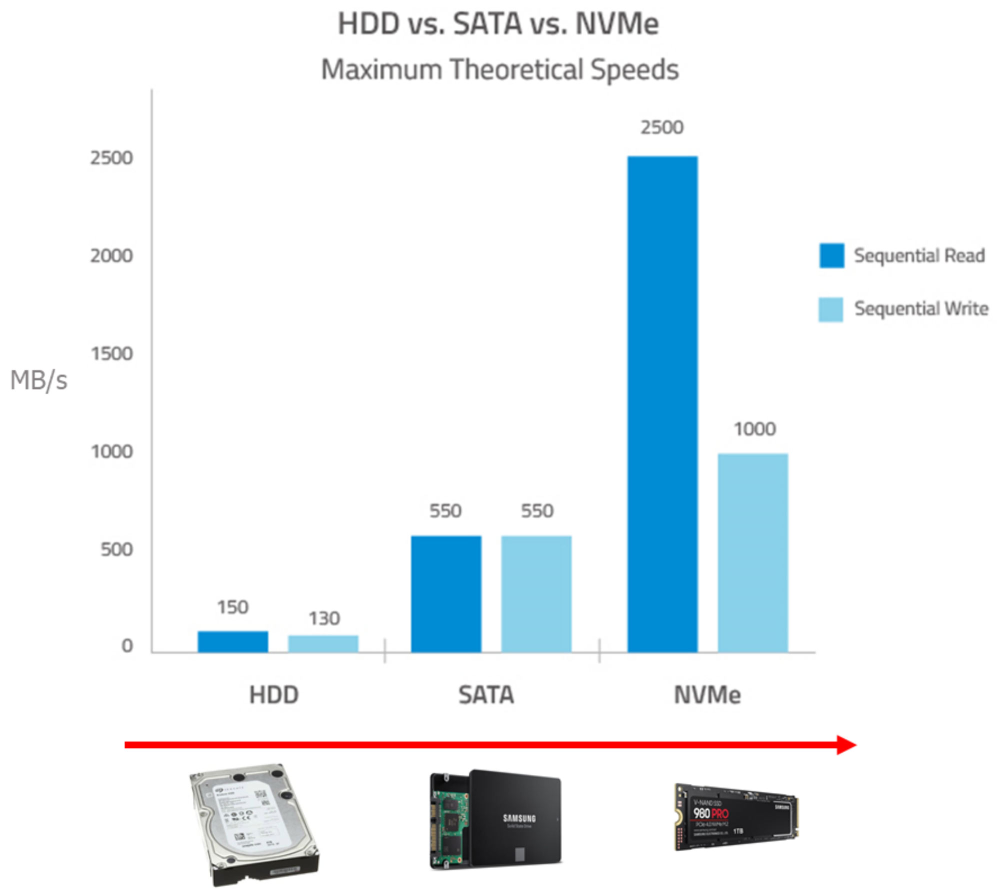
In 2021 USENIX Annual Technical Conference

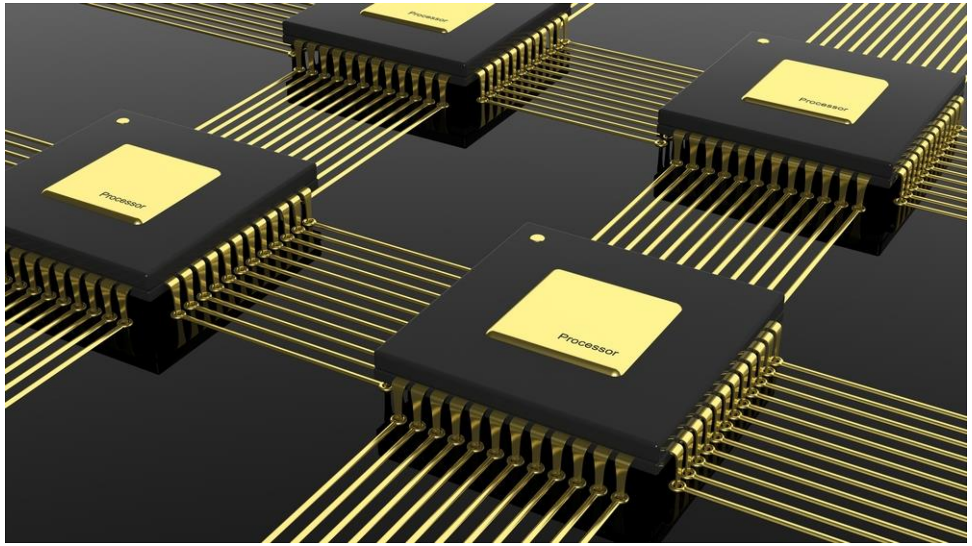
2021. 08. 30

Presentation by Han, Yejin
hyj0225@dankook.ac.kr

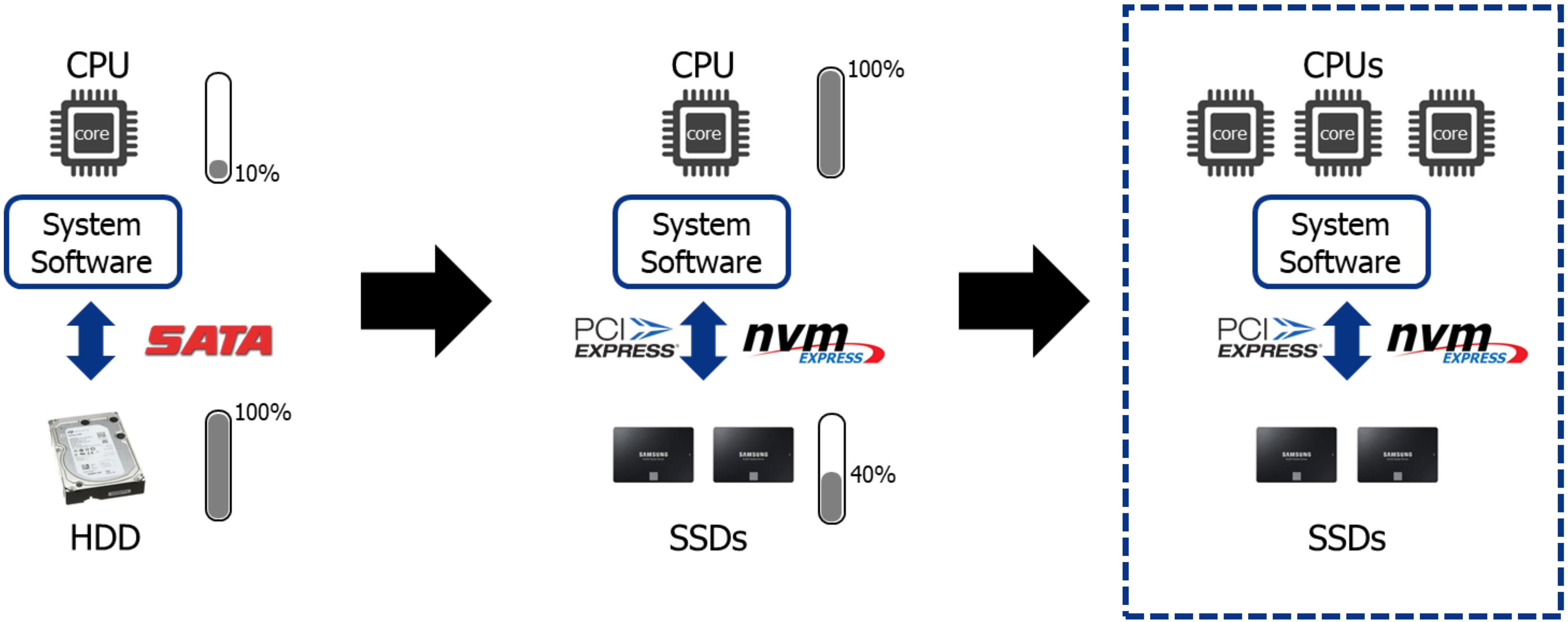
Contents

- The bandwidth of modern storage hardware has been surging in recent years
- Employing multicores for high bandwidth becomes a must



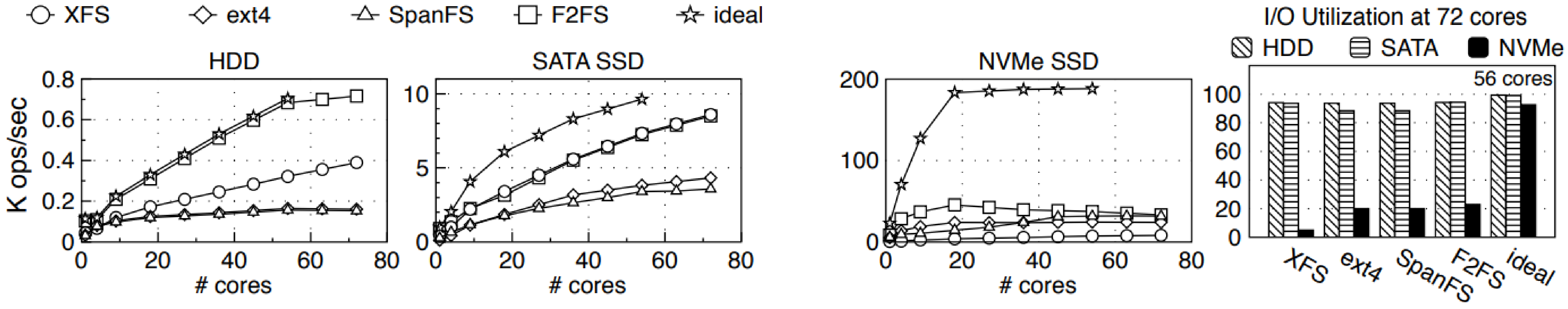


Employing multicore CPUs for high bandwidth



Multicore scalability problem

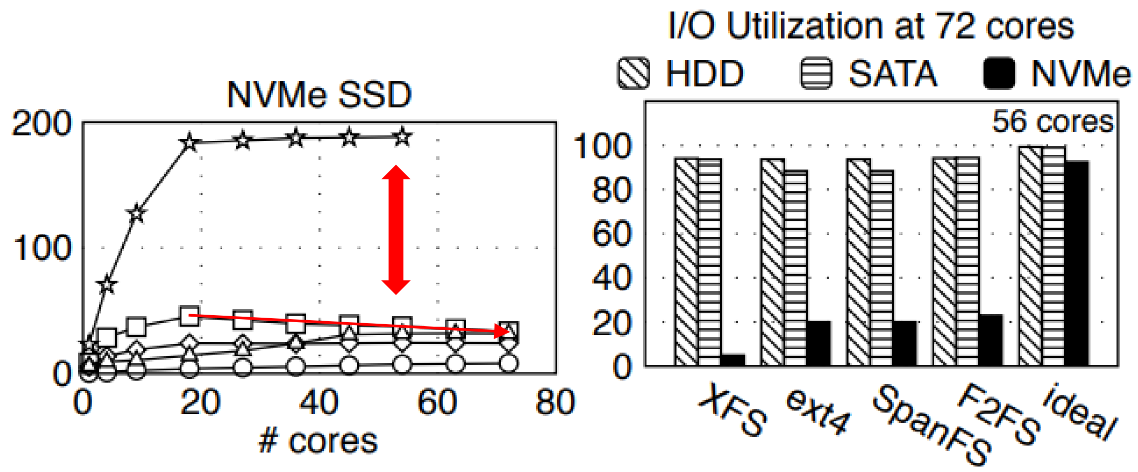
- Workload: create(), write(), fsync(), unlink()
- Ideal: partition the drive and run an independent F2FS on each partition
- Others: multiple CPU cores share a single drive partition



Most file systems scale well
on traditional storage devices

Multicore scalability problem

- Workload: create(), write(), fsync(), unlink()
- Ideal: partition the drive and run an independent F2FS on each partition
- Others: multiple CPU cores share a single drive partition



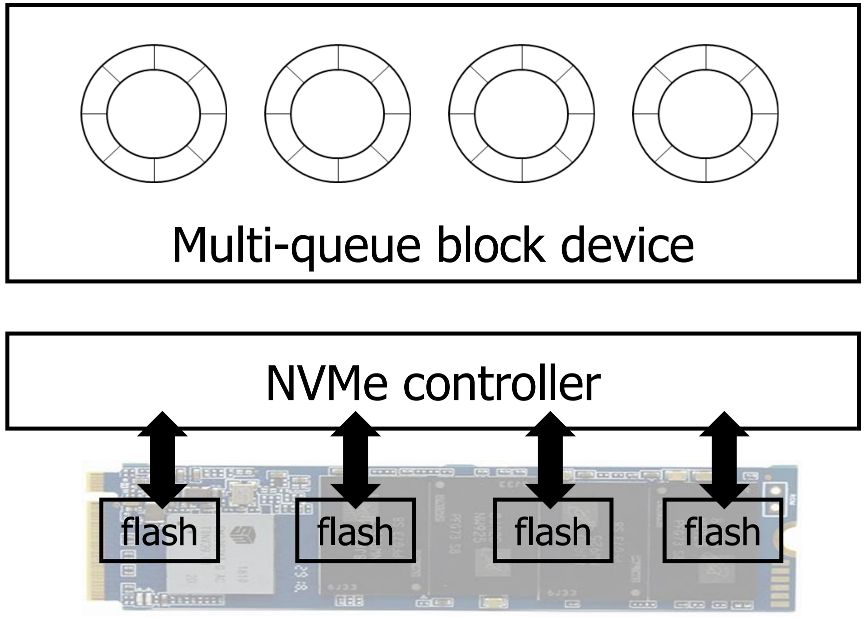
Most file systems scale well
on traditional storage devices

Cannot efficiently utilize the bandwidth
of high performance drives!!!

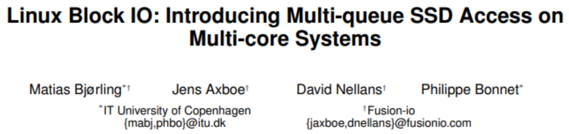
Embedded System Lab.

Multicore scalability problem: File system

Concurrent I/Os and
Multicore CPUs



Multicore-friendly
Design of block layer
And device driver


Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems

Matias Bjørling*† Jens Axboe† David Nellans† Philippe Bonnet*

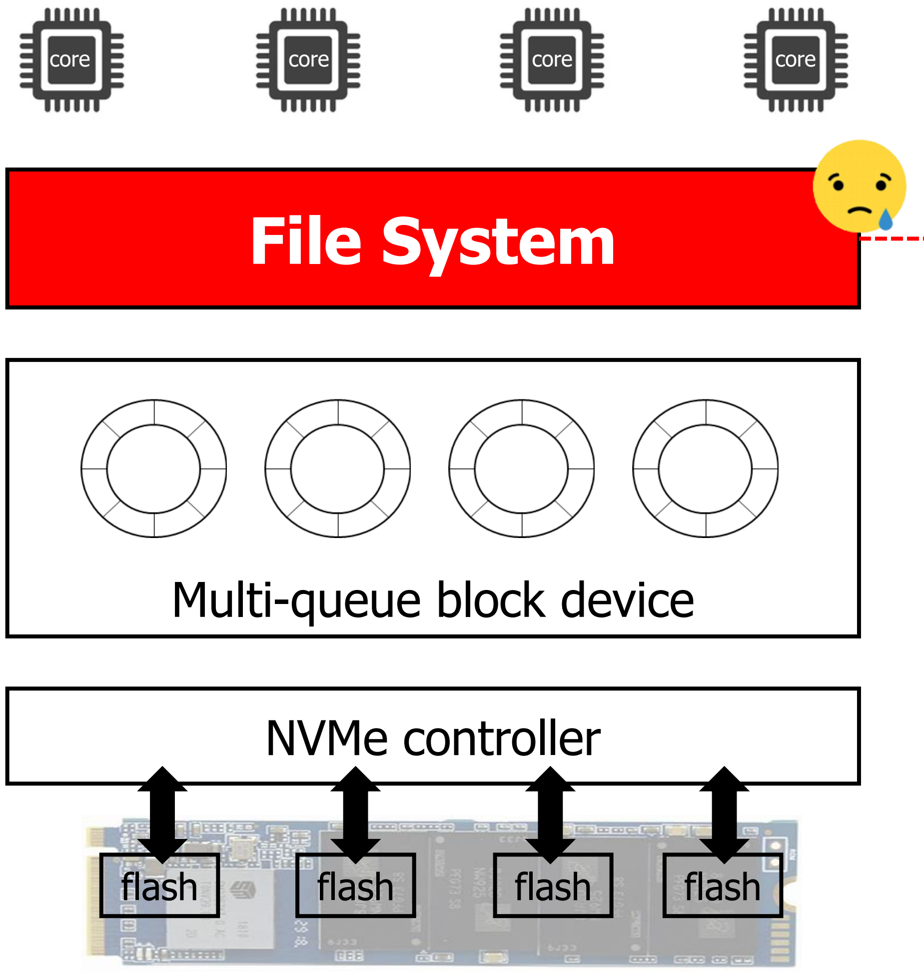
*IT University of Copenhagen {mabj,phbo}@itu.dk

†Fusion-io {jaxboe,dnellans}@fusionio.com

High internal data
Parallelism of the SSD

DANKOOK UNIVERSITY

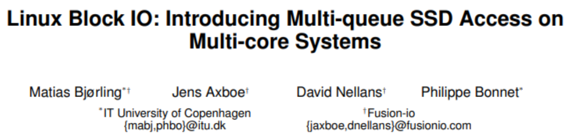
Multicore scalability problem: File system



Concurrent I/Os and Multicore CPUs

File system becomes the scalability bottleneck

Multicore-friendly Design of block layer And device driver


Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems

Matias Bjørling[*†] Jens Axboe[†] David Nellans[†] Philippe Bonnet[*]

[*]IT University of Copenhagen {mabj,phbo}@itu.dk

[†]Fusion-io {jaxboe,dnellans}@fusionio.com

High internal data Parallelism of the SSD

What are the root causes of poor scalability in F2FS?

- Lock contention from unscalable data structure cause CPU overhead
- Lock level: Concurrency Control, In-Memory Data Structure, Space Allocation

Operations	Lock/Sharing	Overhead	Lock level
write()	sbi->cp_rwsem	50.98%	CC
create()	nm_i->nat_tree_lock	3.74%	IMDS
	sbi->inode_lock	3.22%	IMDS
	curseg->curseg_mutex	1.84%	SA
	nm_i->nid_list_lock	1.09%	IMDS
unlink()	nm_i->nat_tree_lock	22.68%	IMDS
	im->ino_lock	6.54%	IMDS
	sbi->inode_lock	3.05%	IMDS
	node_inode	2.66%	IMDS
	sit_i->sentry_lock	2.41%	SA
fsync()	sbi->writepages	45.76%	SA
	sit_i->sentry_lock	1.07%	SA

Table 2: **The scalability bottlenecks of F2FS.**

51% CPU cycles for locking on write()

10% CPU cycles for locking on create()

37% CPU cycles for locking on unlink()

45% CPU cycles for locking on fsync()

What are the reasons for causing CPU overhead? 🤔

Concurrency Control → **Cache coherence on the *shared lock counter value***

In-Memory Data Structure → ***Serialization* in accessing the three radix trees**
(FS metadata, File metadata, File data)

Space Allocation → ***One logical space allocator***

Overall Architecture

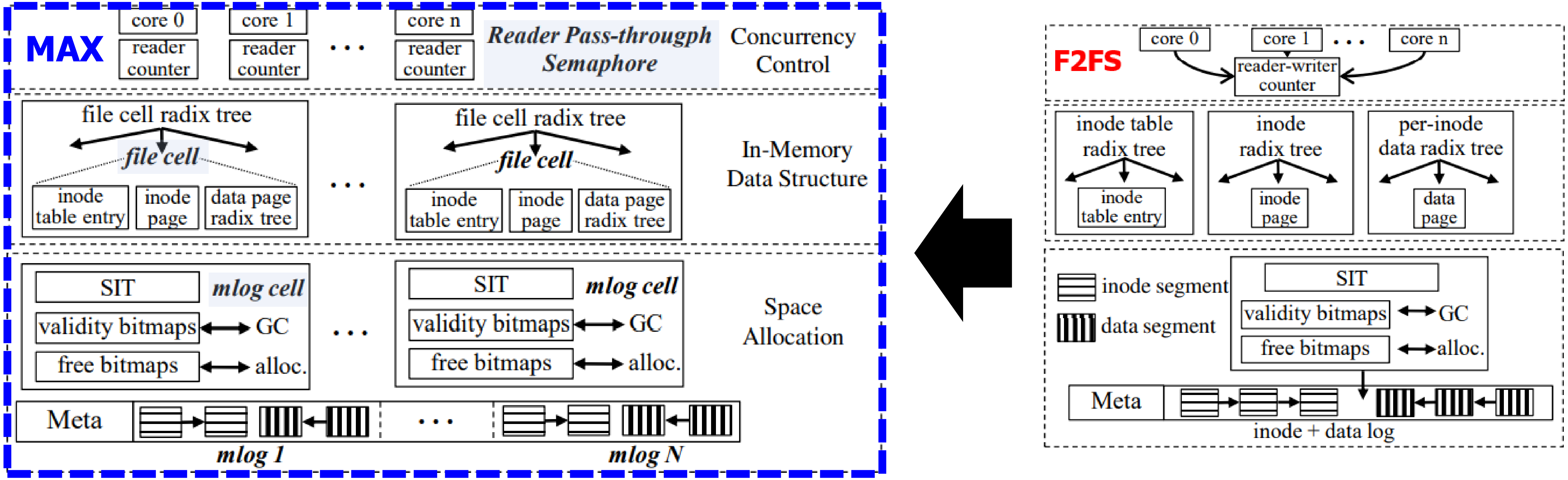


Figure 2: **A comparison of F2FS and MAX.** MAX *introduces Reader Pass-through Semaphore (RPS) (§4.1) at CC level, file cell (§4.2) at IMDS level and mlog cell (§4.3) at SA level for higher concurrency.*

Multicore-Accelerated File System

1) CC: Reader Pass-through Semaphore

2) IMDS: file cell

3) SA: mlog cell

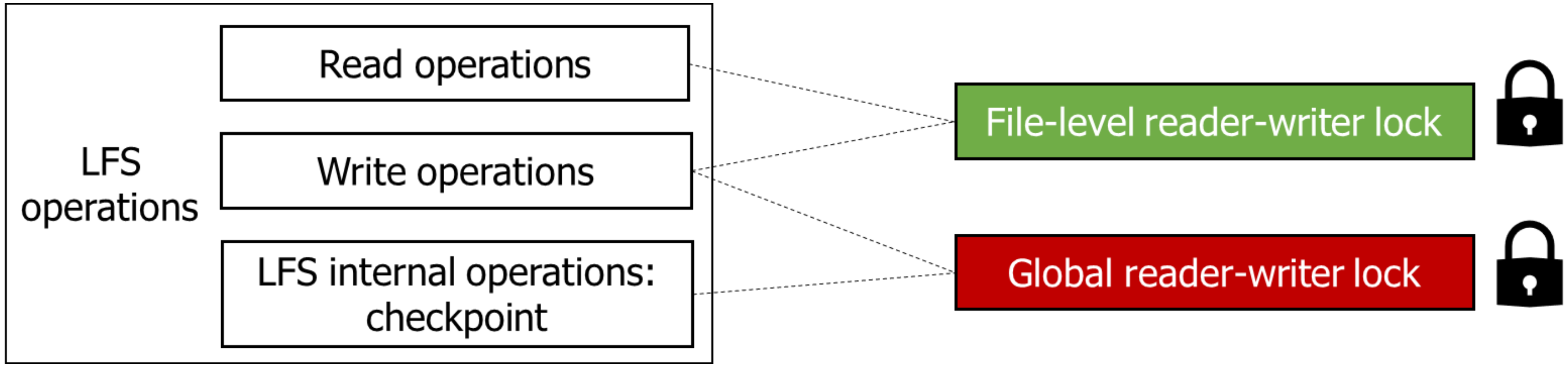
CC: Reader Pass-through Semaphore

Classic LFS's concurrency control

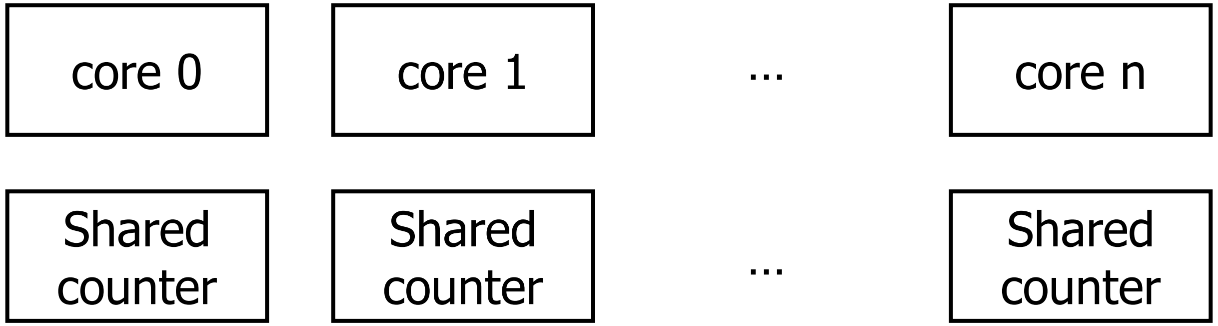
- Operations and concurrenу control

Lock level \ Lock type	**Shared mode**	**Exclusive mode**
Concurrency Control (CC)	write operations (e.g., `write`)	global operations (e.g.,sync)

Table 1: **Current practices of the file system sharing.**

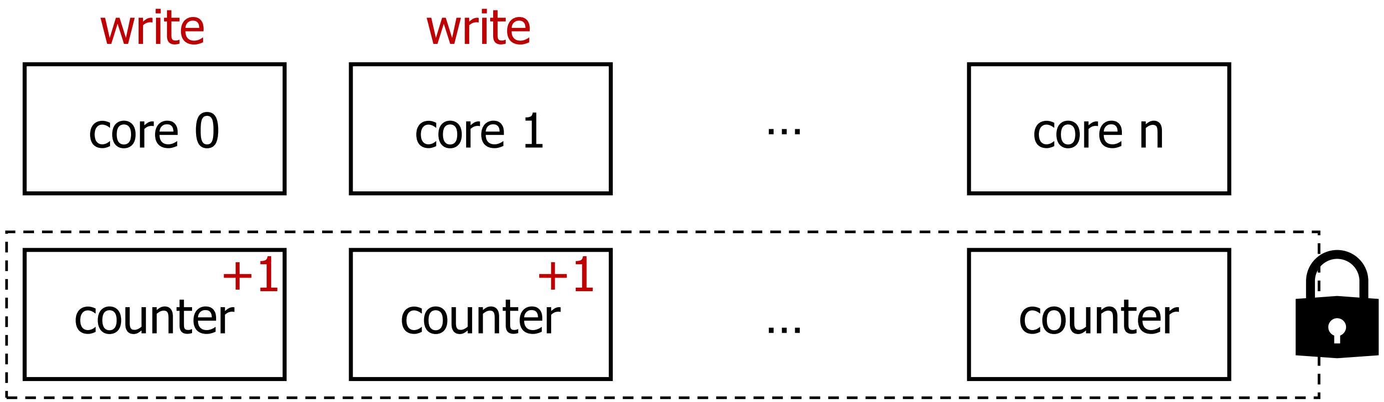


- Lock counter is shared among cores, so cache coherence can be severe with the scaling core
  - ✓ Write requires a shared-mode lock
  - ✓ Checkpoint requires an exclusive-mode lock



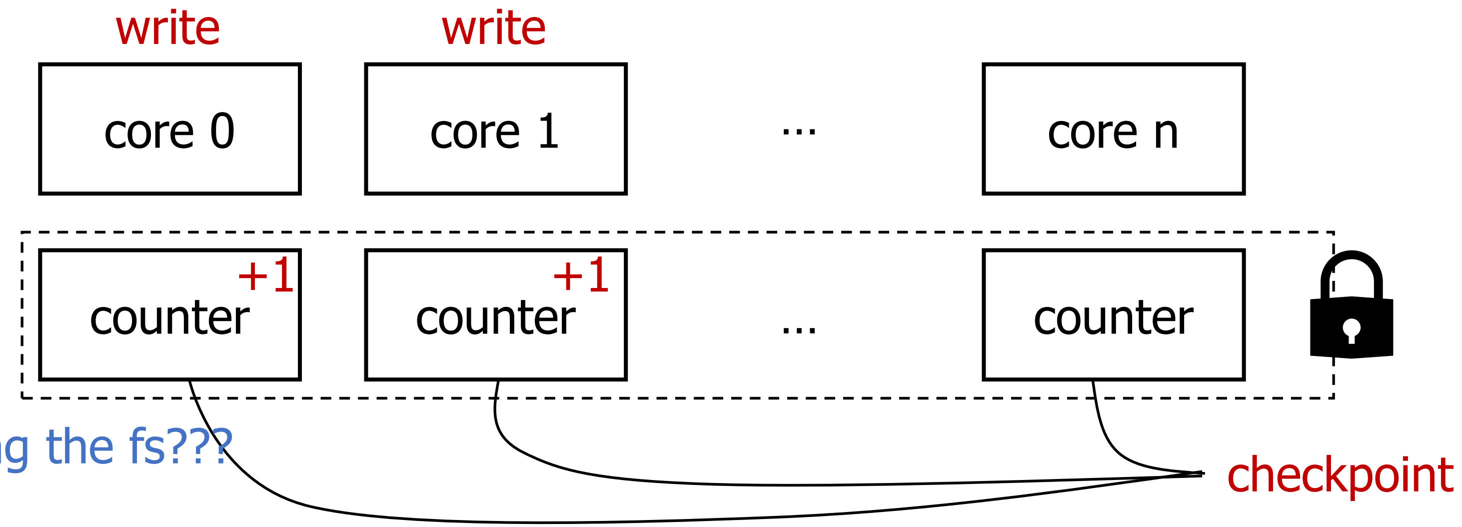
CC: Reader Pass-through Semaphore

- Concurrent readers: Per-core lock counters
- Scheduler Free Rides: CPU scheduler checks per core counter value



CC: Reader Pass-through Semaphore

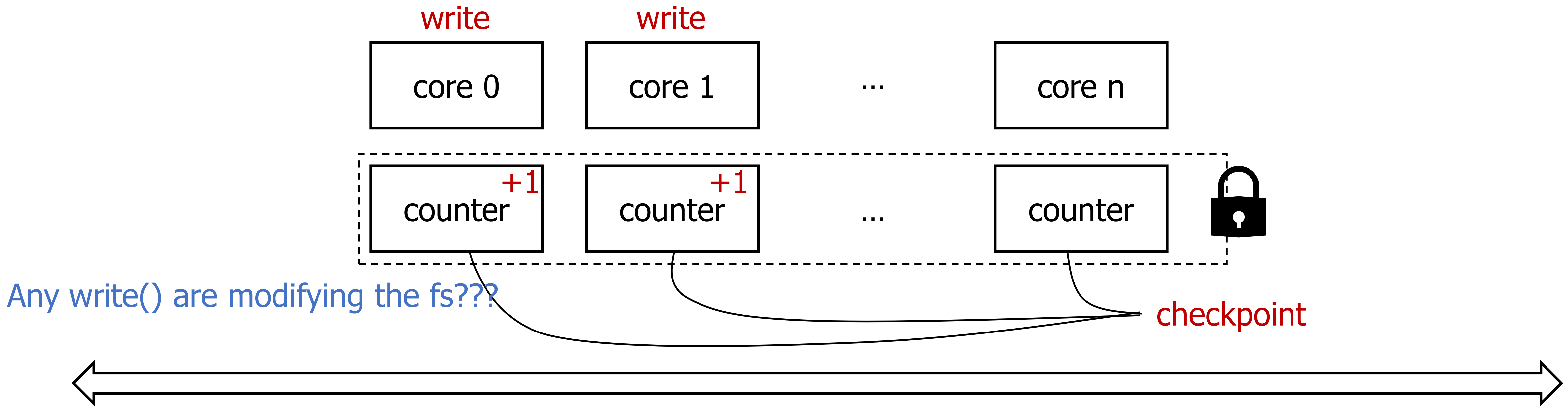
- Concurrent readers: Per-core lock counters
- Scheduler Free Rides: CPU scheduler checks per core counter value



Any write() are modifying the fs???

CC: Reader Pass-through Semaphore

- Concurrent readers: Per-core lock counters
- Scheduler Free Rides: CPU scheduler checks per core counter value



aggressive polling

: High overhead

Scheduler-assisted check

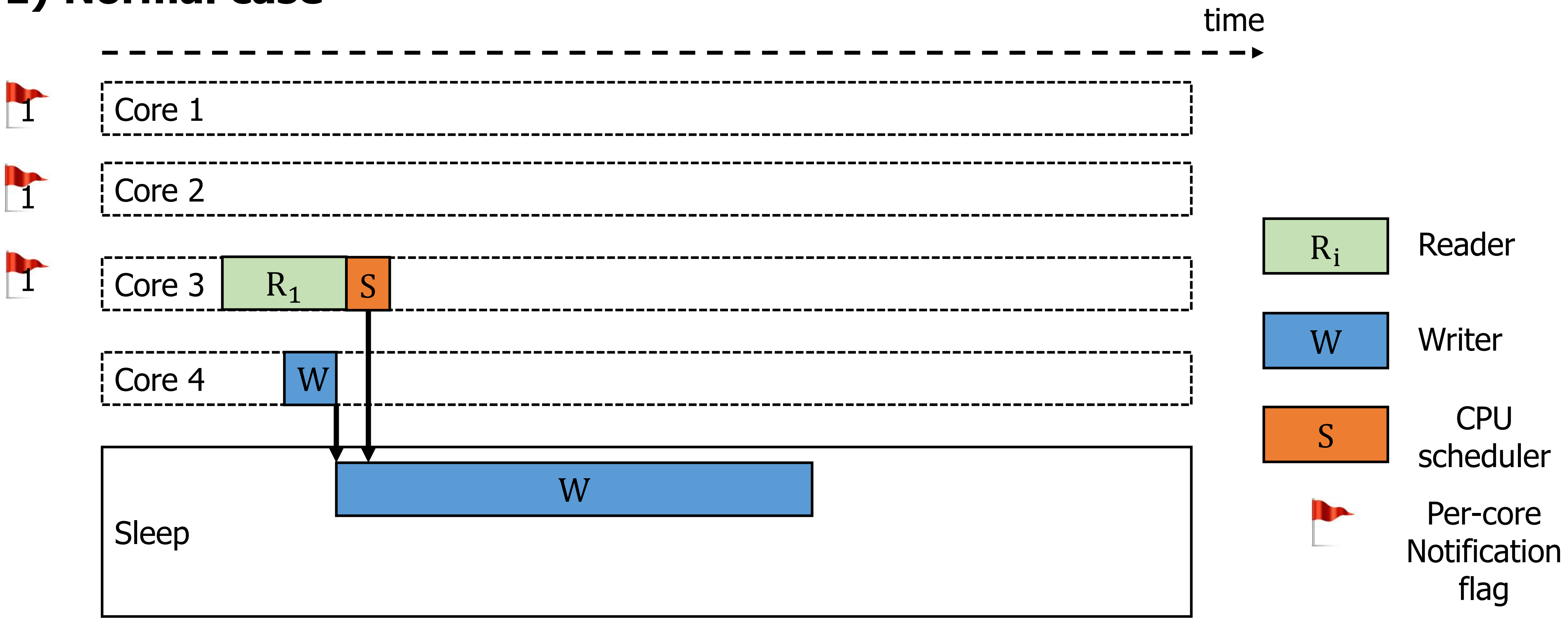
lazy wait

: High latency and slows the checkpoint

CC: Reader Pass-through Semaphore

- The FS in kernel space frequently triggers CPU scheduler

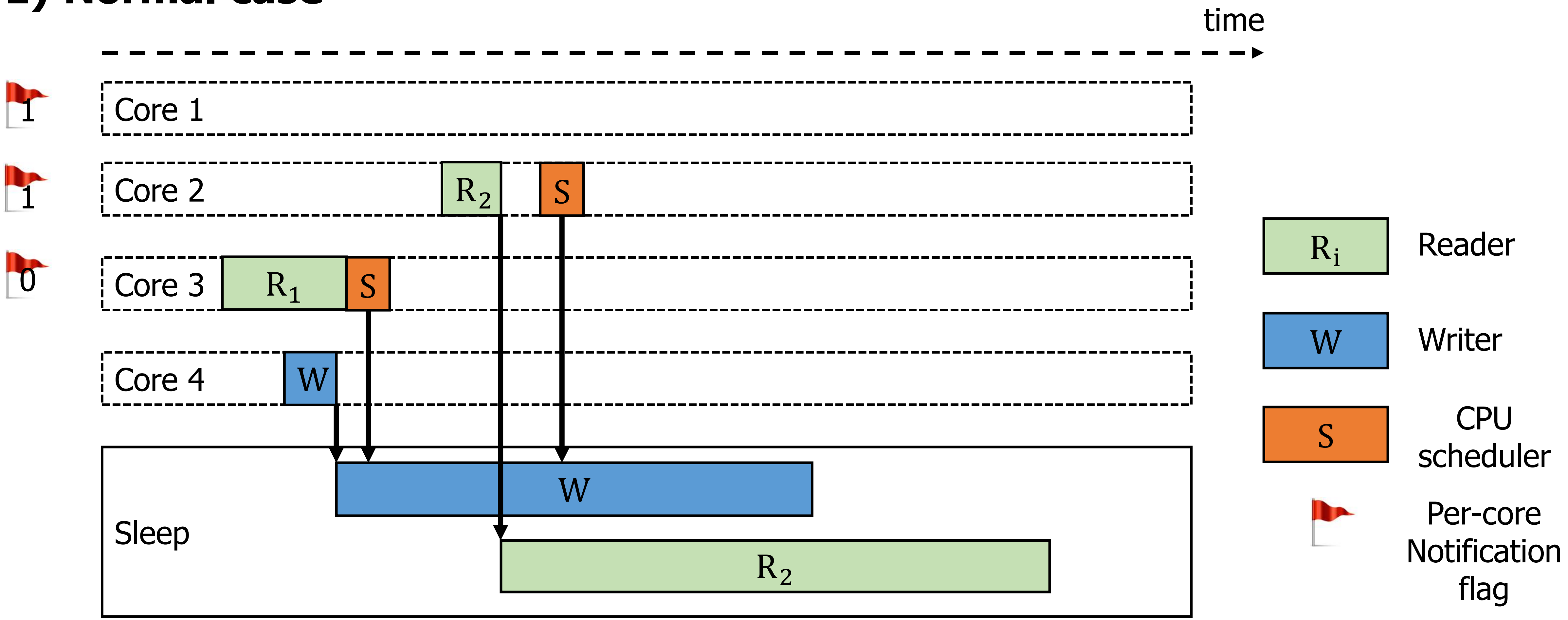
1) Normal case



CC: Reader Pass-through Semaphore

- The FS in kernel space frequently triggers CPU scheduler

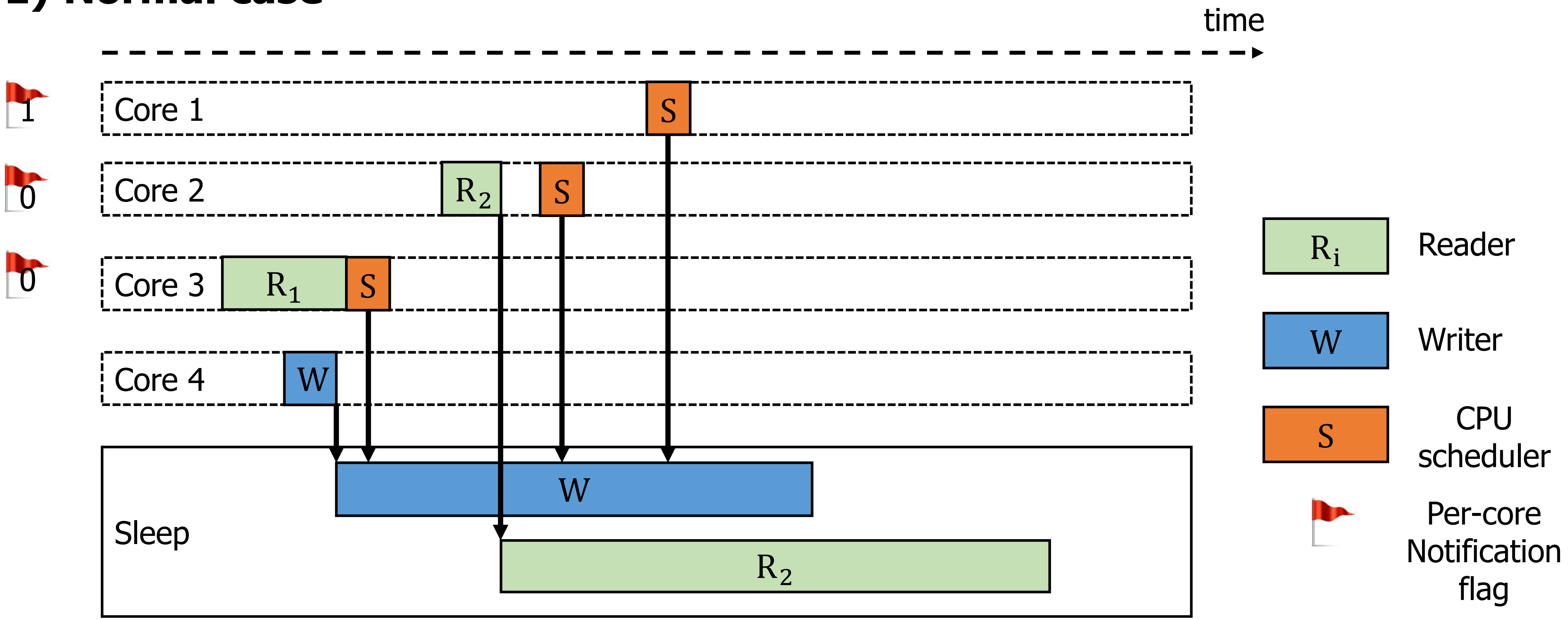
1) Normal case



CC: Reader Pass-through Semaphore

- The FS in kernel space frequently triggers CPU scheduler

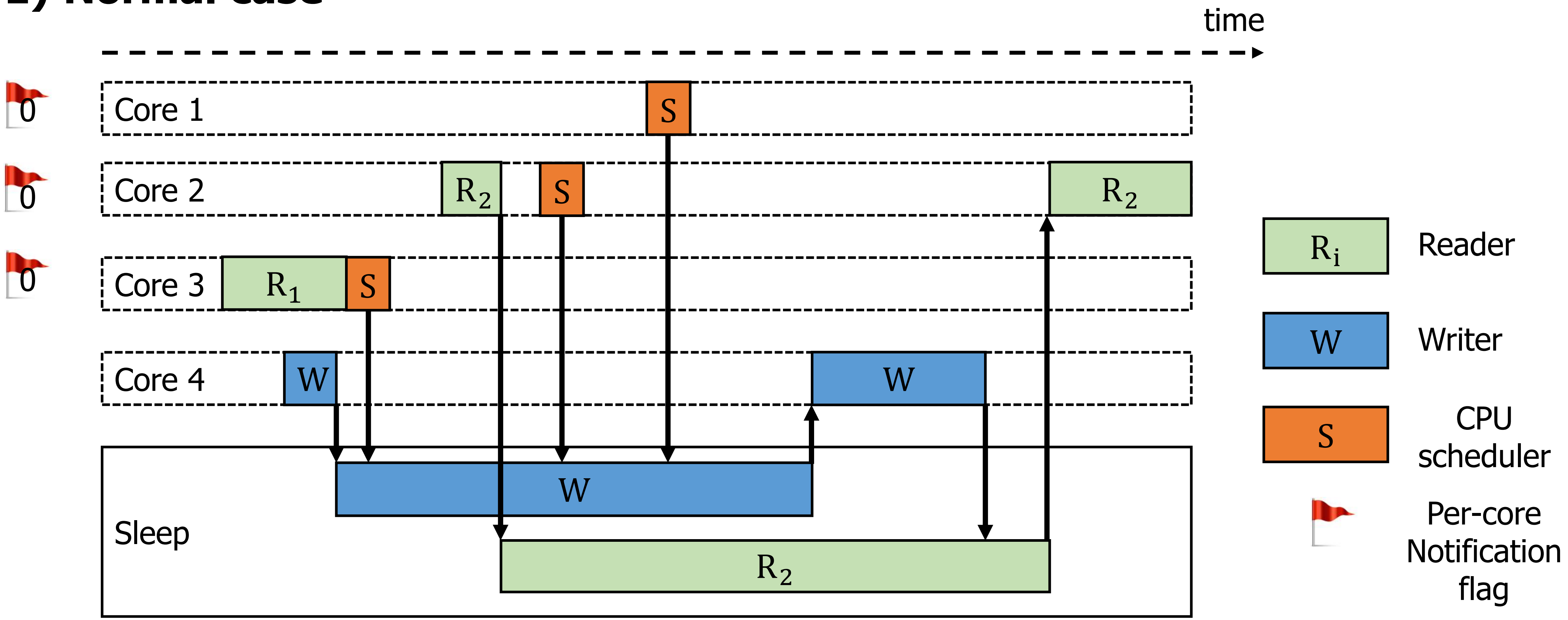
1) Normal case



CC: Reader Pass-through Semaphore

- The FS in kernel space frequently triggers CPU scheduler

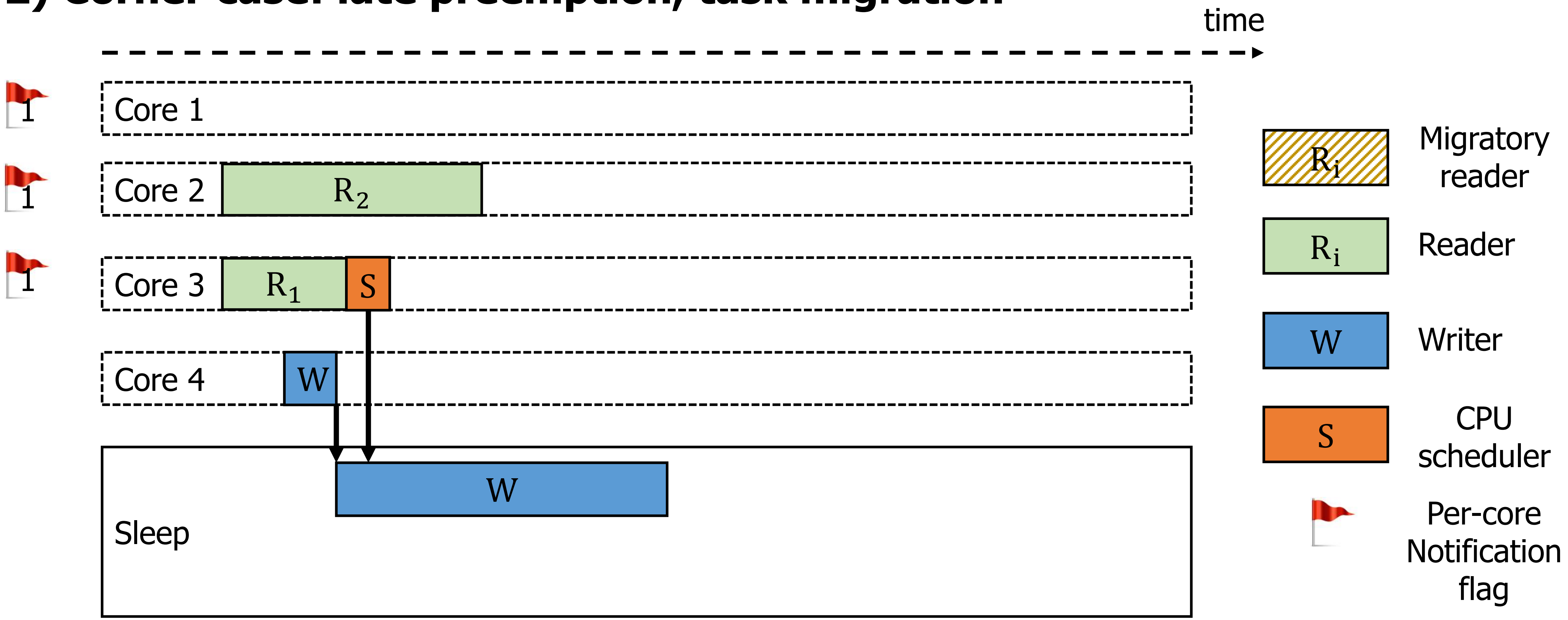
1) Normal case



CC: Reader Pass-through Semaphore

- The FS in kernel space frequently triggers CPU scheduler

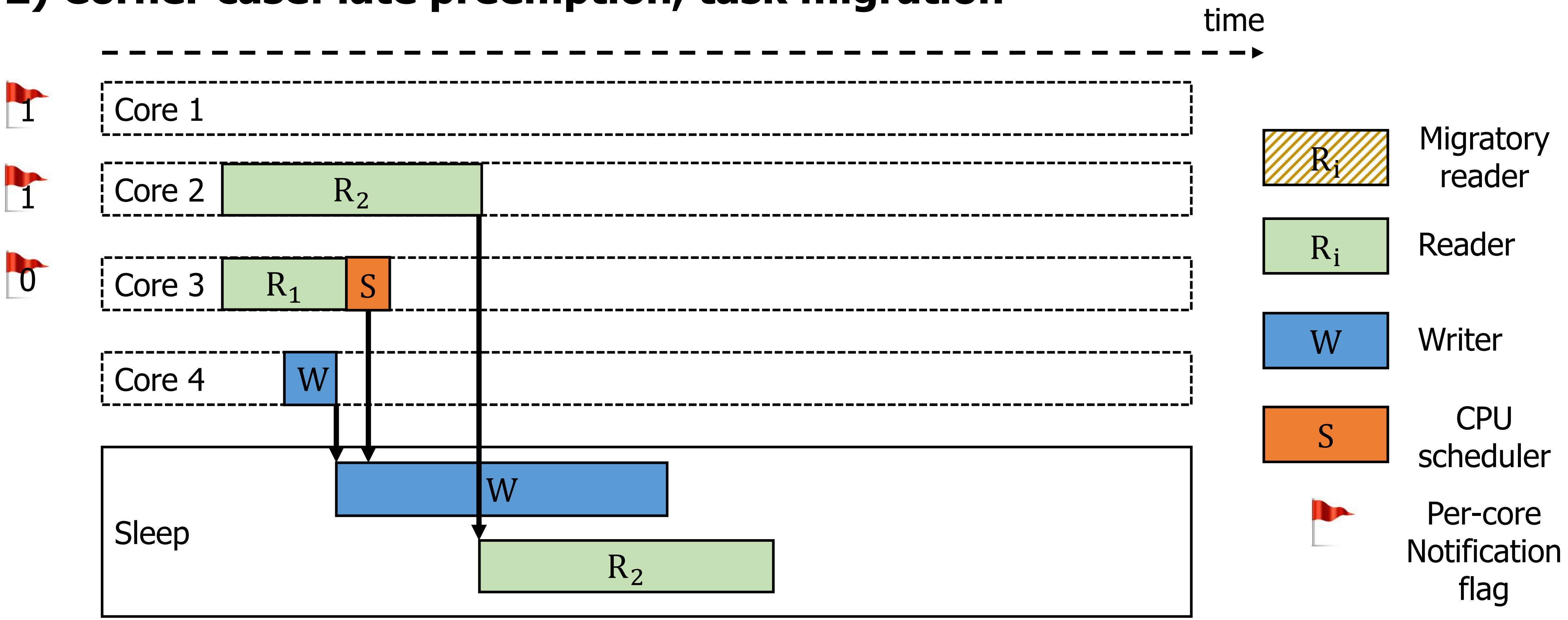
2) Corner case: late preemption, task migration



CC: Reader Pass-through Semaphore

- The FS in kernel space frequently triggers CPU scheduler

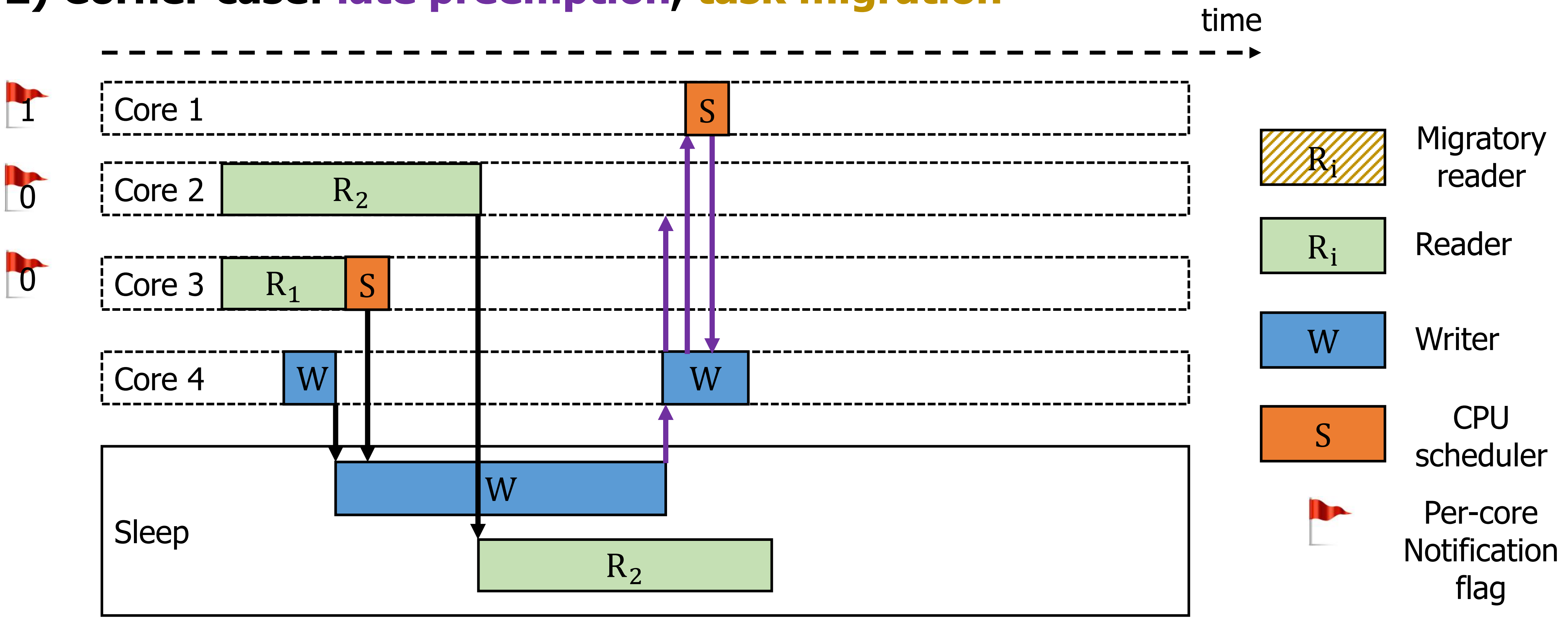
2) Corner case: late preemption, task migration



CC: Reader Pass-through Semaphore

- The FS in kernel space frequently triggers CPU scheduler

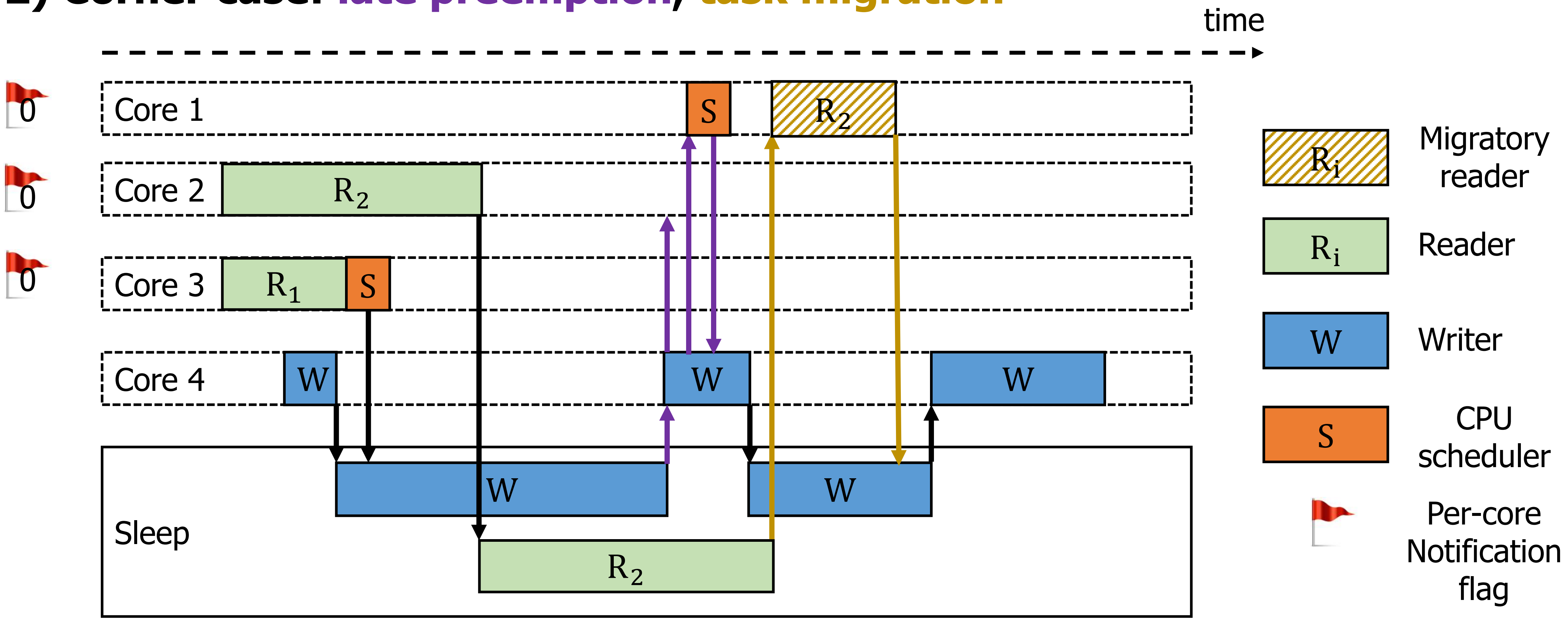
2) Corner case: late preemption, task migration



CC: Reader Pass-through Semaphore

- The FS in kernel space frequently triggers CPU scheduler

2) Corner case: late preemption, task migration



Multicore-Accelerated File System

1) CC: Reader Pass-through Semaphore

2) IMDS: File Cell

3) SA: mlog cell

IMDS: File Cell

- Partition and reorganize in-memory data structure
- Encompasses inode table entry, inode page, data page
- Places each file cell to a tree by hashing the inode ID

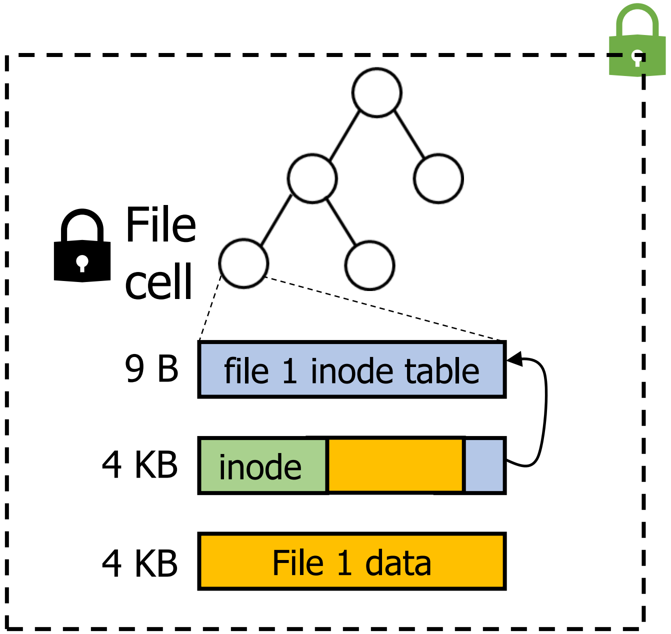
Lock level \ Lock type	Shared mode	Exclusive mode
In-Memory Data Structure (IMDS)	index read operations (e.g., `write`)	index write operations (e.g., `create`)

Table 1: **Current practices of the file system sharing.**

File cell group 0
Inode number % N =0

…

File cell group N
Inode number % N =N-1



write(), read(), stat(), fsync():

1. Lock group/indexing in shared-mode
2. Lock file cell of target file
3. File operations

IMDS: File Cell

- Partition and reorganize in-memory data structure
- Encompasses inode table entry, inode page, data page
- Places each file cell to a tree by hashing the inode ID

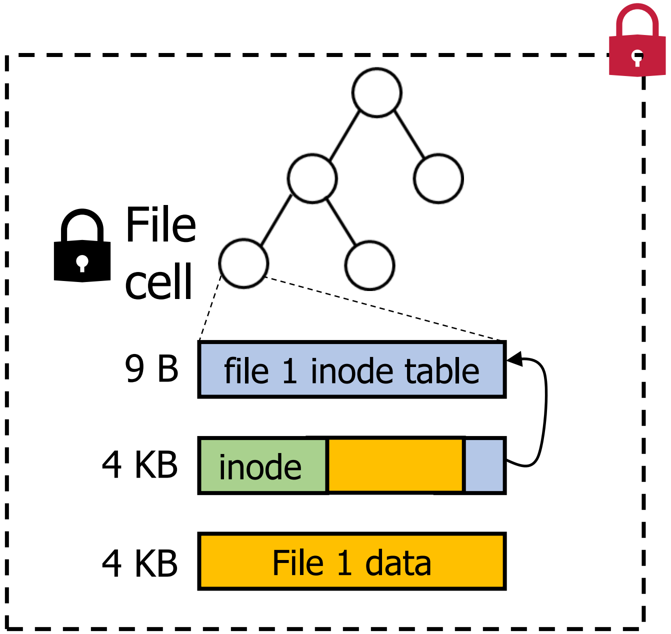
Lock level \ Lock type	Shared mode	Exclusive mode
In-Memory Data Structure (IMDS)	index read operations (e.g., `write`)	index write operations (e.g., `create`)

Table 1: **Current practices of the file system sharing.**

File cell group 0
Inode number % N =0

…

File cell group N
Inode number % N =N-1



write(), read(), stat(), fsync():

1. Lock group/indexing in shared-mode
2. Lock file cell of target file
3. File operations

create(), unlink(), mkdir():

1. Lock group/indexing in exclusive-mode
2. Lock file cell of target files and directory
3. File operations

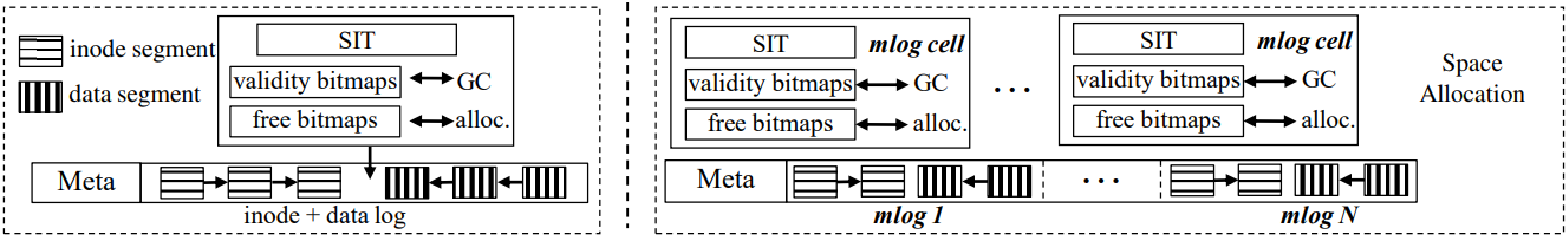
Multicore-Accelerated File System

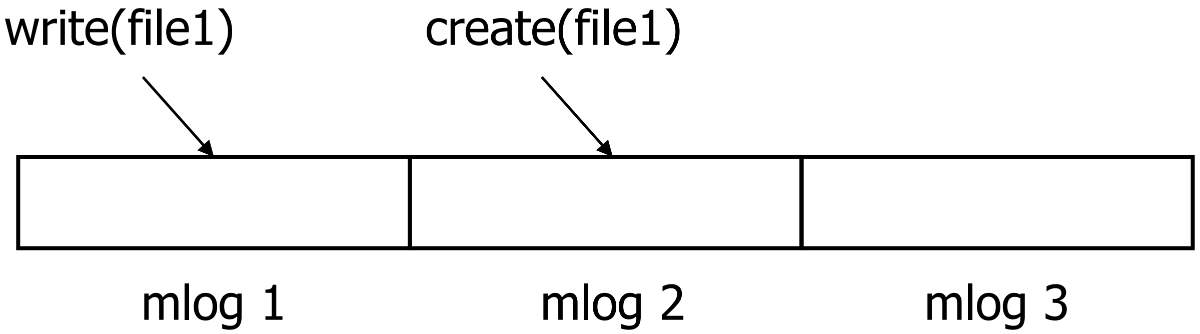
1) CC: Reader Pass-through Semaphore

2) IMDS: File Cell

3) SA: mlog cell

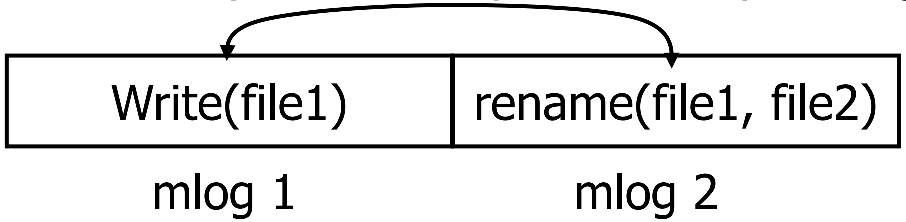
SA: Mlog Cell

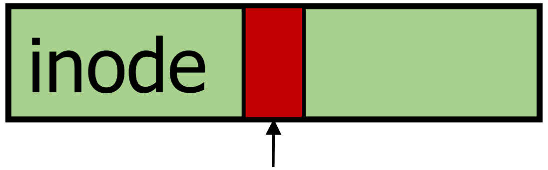




- dispatch atomic file operations in a round-robin fashion
- each mlog maintains its internal consistency

How to keep consistency over multiple mlog?





Use a global version number to decide the persistence ordering

Summary

Concurrency Control → **Reader Pass-through Semaphore**

In-Memory Data Structure → **File cell to scale the access to IMDS**
(FS metadata, File metadata, File data)

Space Allocation → **Mlog for concurrent SA and persistence**

Evaluation Environment

CPU	4 Intel Xeon Gold 6140 CPU, each with 18 physical cores (total: 72 CPU cores)
Memory	250 GB DRAM (only 10% DRAM used for page cache)
OS	Linux vanilla kernel 4.19.11
Filesystem	Ext4 / XFS / F2FS / SpanFS / MAX
Device	Intel DC P3700 SSD
Workloads	1) Data and metadata scalability 2) Varmail and RocksDB 3) Upper bound evaluation against tmpfs

- **File Operation**

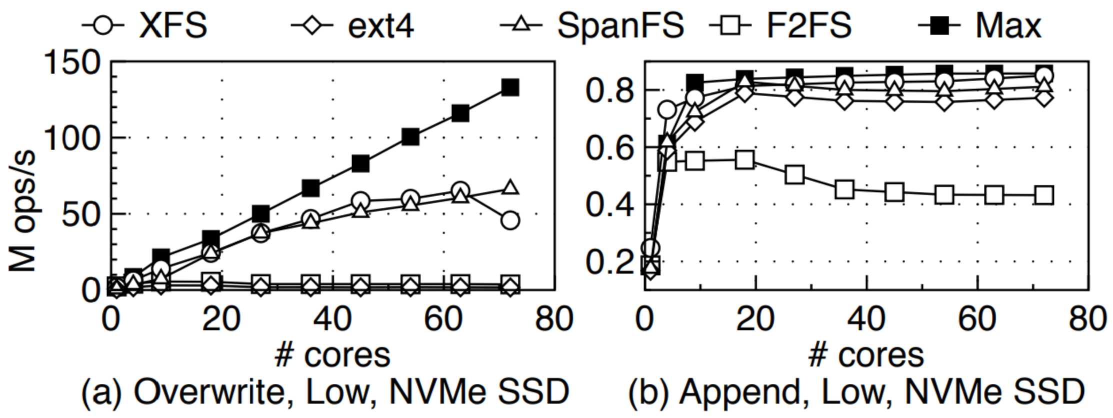


Figure 6: **Data scalability with FxMark.** *((a): overwriting blocks of private files, (b): appending blocks to private files.)*

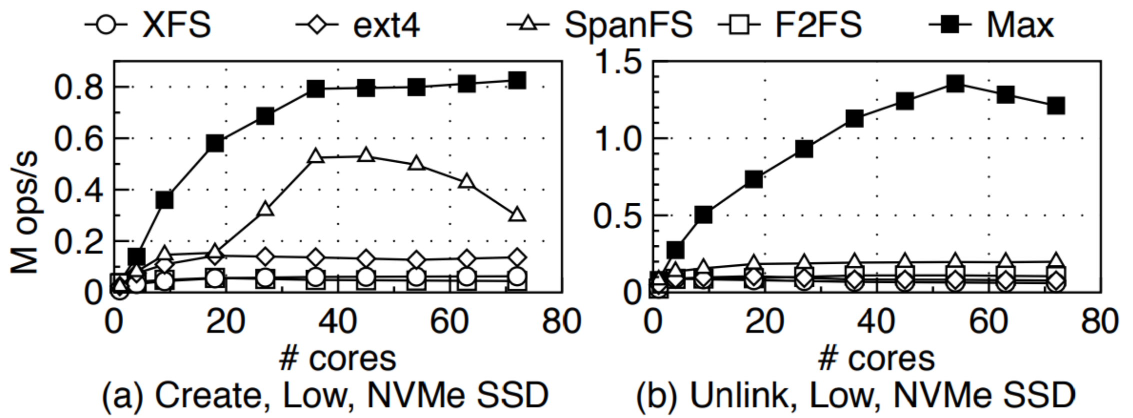


Figure 7: **Metadata scalability with FxMark.** *((a)-(b): creating or deleting empty files in private directories.)*

MAX = 35 x F2FS in overwrite,
= 2 x F2FS in append

MAX = 18.6 x F2FS in create,
= 11.5 x F2FS in unlink

- **Max-mem Performance**
  - Max-mem: disable fsync and page cache flushes to avoid duplicate on-disk copy
  - tmpfs: a simple wrapper of Linux VFS, a memory-based file system
  - Tested device: 20GB memory-backed RAMdisk

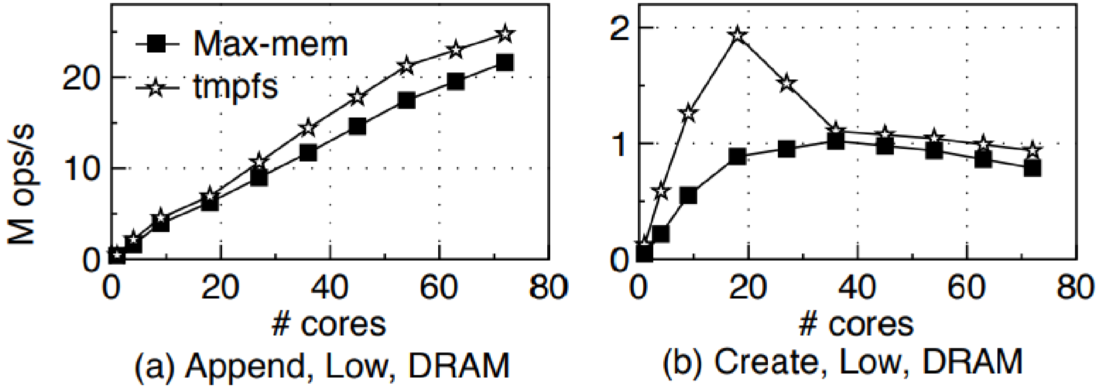


Figure 9: **Upper bound evaluation with FxMark.** *((a): appending blocks to private files, (b) creating empty files in private directories.)*

The throughput of Max-mem comes close to tmpfs

- **Macrobenchmark**

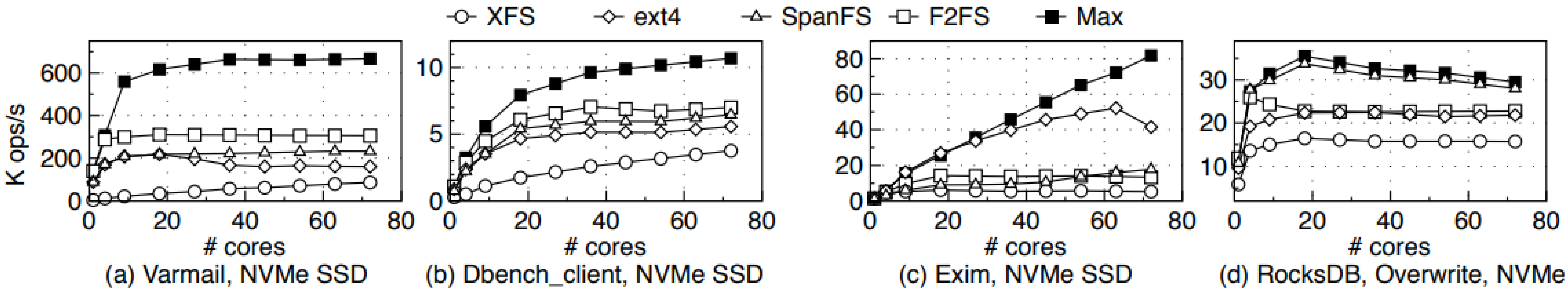
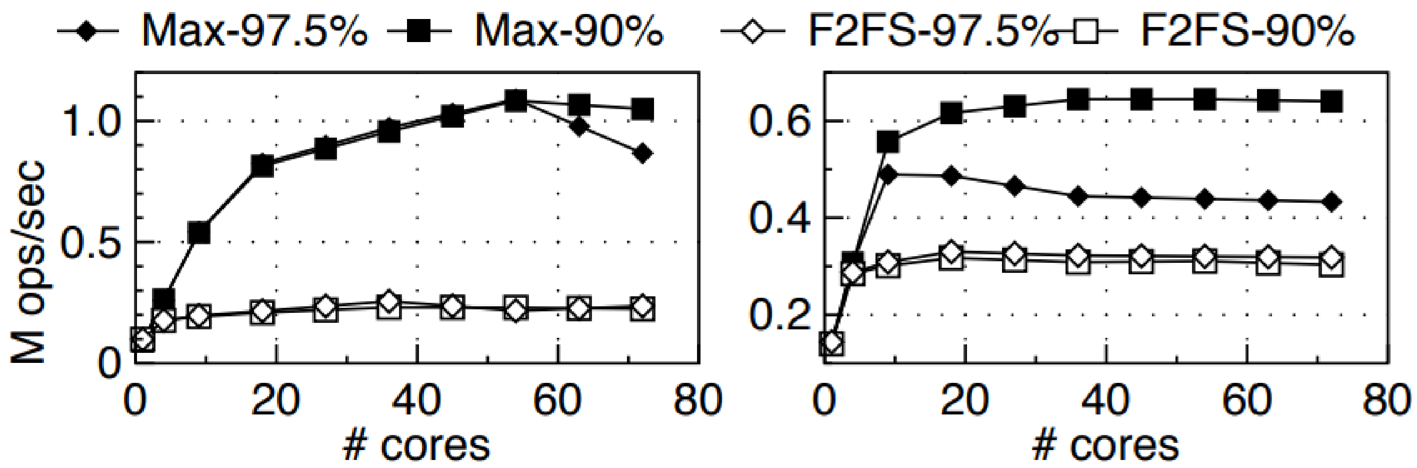


Figure 10: **Macrobenchmark.** *The workloads are write-intensive and stress underlying device. Described in §5.2.*

MAX performs best among all tested file system

- **High volume utilization**



(a) Fileserver, NVMe SSD (b) Varmail, NVMe SSD

Figure 11: **Performance under high disk volume utilization.** *The number next to the file system is volume utilization.*

MAX outperforms F2FS in Fileserver and varmail

MAX

- Reader Pass-through semaphore for CC, file cell for IMDS and mlog for SA
- Outperforms modern Linux file systems
- Offers multicore scalability and fully exploits the bandwidth of NVMe SSD

https://github.com/thustorage/max

MAX : A Multicore-Accelerated File System for Flash Storage

Xiaojian Liao, Youyou Lu, Erci Xu, Jiwu Shu,

In 2021 USENIX Annual Technical Conference

Thank You!

2021. 08. 30

Presentation by Han, Yejin

hyj0225@dankook.ac.kr