CrossFS: A Cross-layered Direct-Access File System

Yujie Ren[1], Changwoo Min[2], and Sudarsun Kannan[1]

[1] Rutgers University; [2] Virginia Tech

Modern File System Limitations

- High software overheads
  - System call overheads compounded with layers of I/O stack
- Lack support for leveraging host and device-level compute
  - Host CPUs are fast, but low utilization for processing I/O requests
  - Device CPUs are under-utilized
- Coarse-grained locks leading to non-scalable concurrent access
  - Inode-level lock limits concurrent access for shared file
  - Even updates to non-overlapping range of blocks are serialized

Our Solution: CrossFS

- Disaggregates file system across user-level, OS, and firmware
  Divides work across host-level and device-level compute

- Applications directly access the firmware file system
  Avoids system call overheads for data and control plane

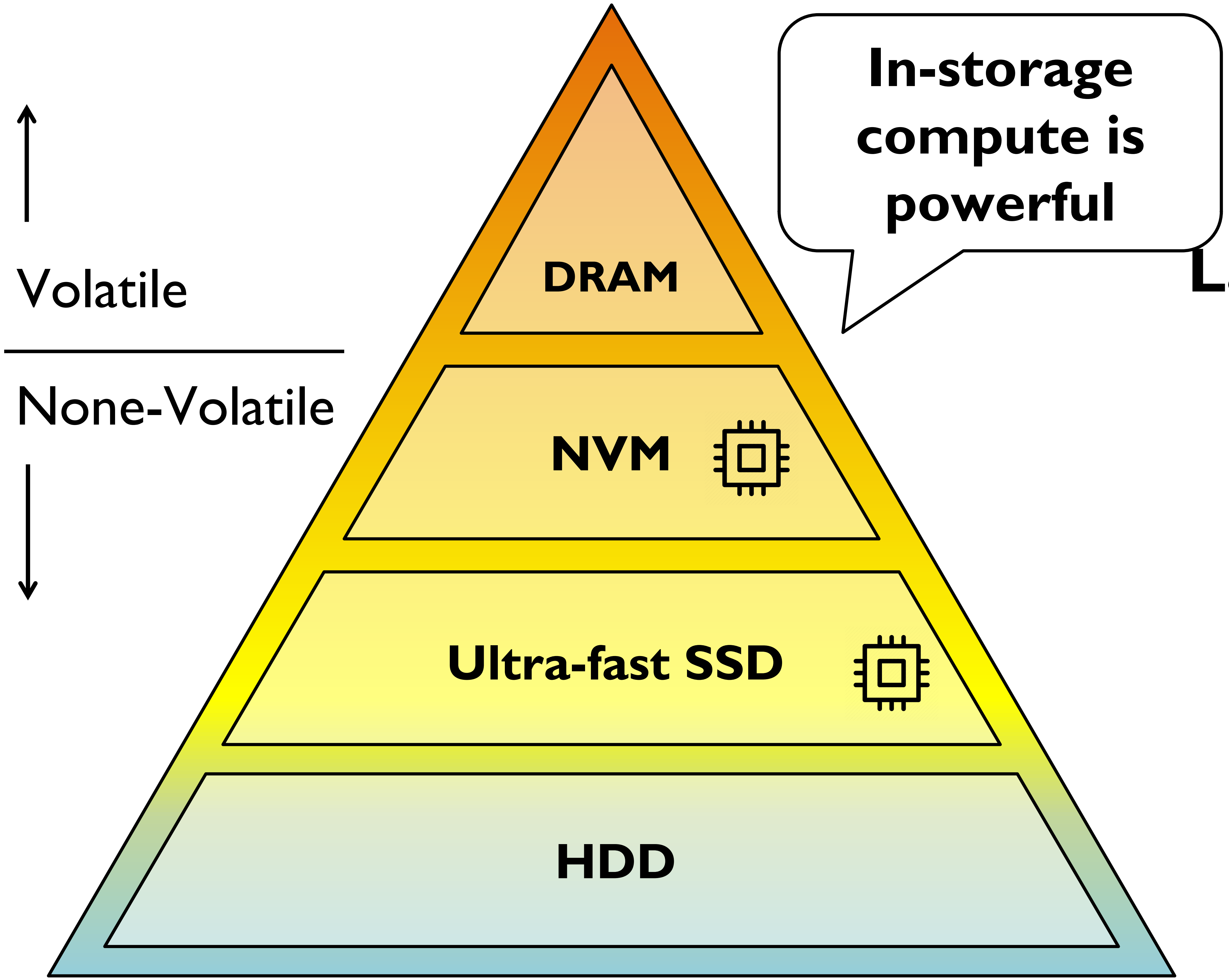
- Designs fine-grained file descriptor-level concurrency
  Replaces coarse-grained inode-level lock in current file systems

- Achieves Up to 4x higher throughput!

Outline

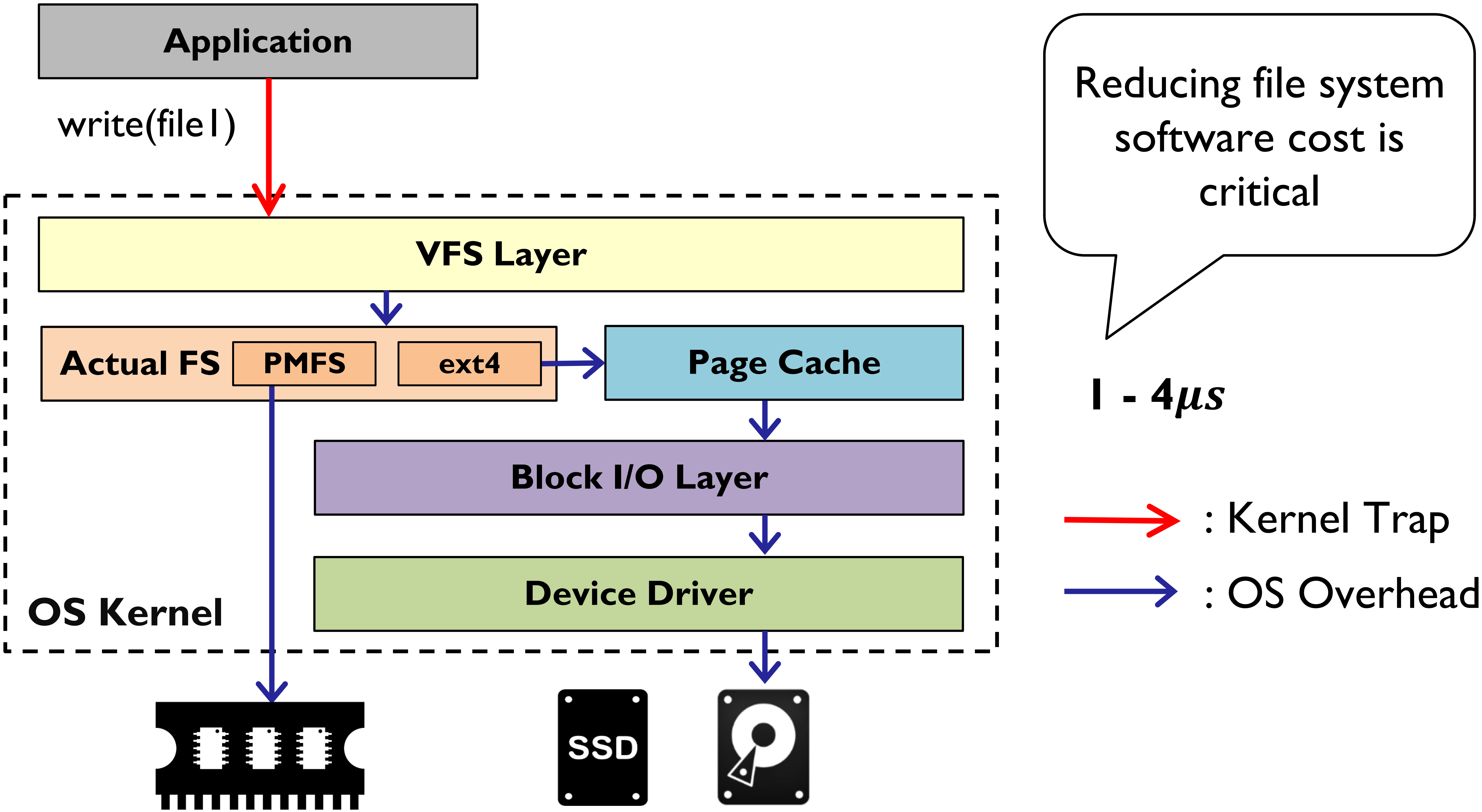
- Background
- Motivation
- Design
- Evaluation
- Conclusion

Modern Storage Devices

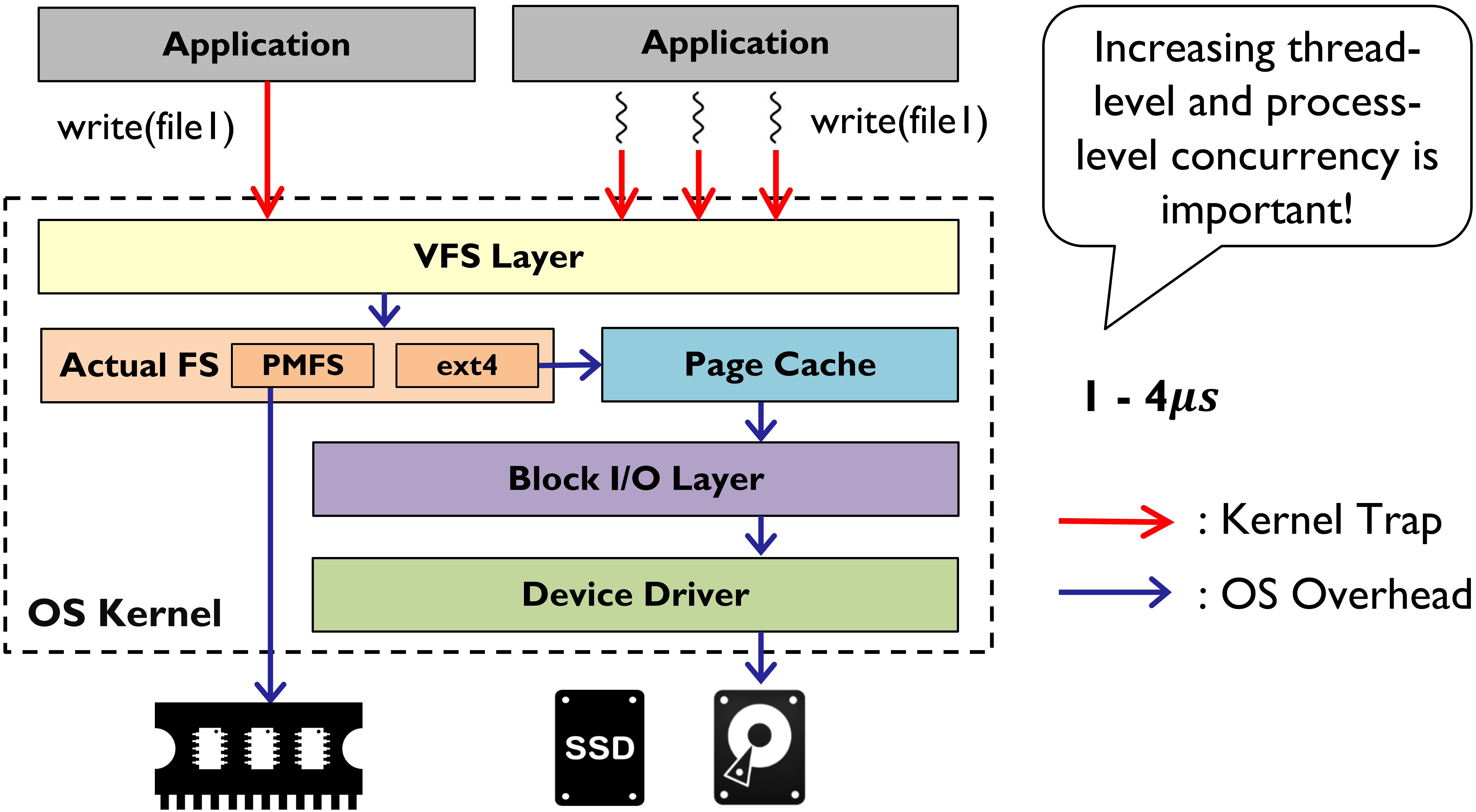


Latency	B/W	$/GB
300 ns	10 GB/s	4.0
40 μs	2 GB/s	0.25
5 ms	2.6 MB/s	0.02

I/O Software Overheads



I/O Software Overheads



Application I/O Behavior

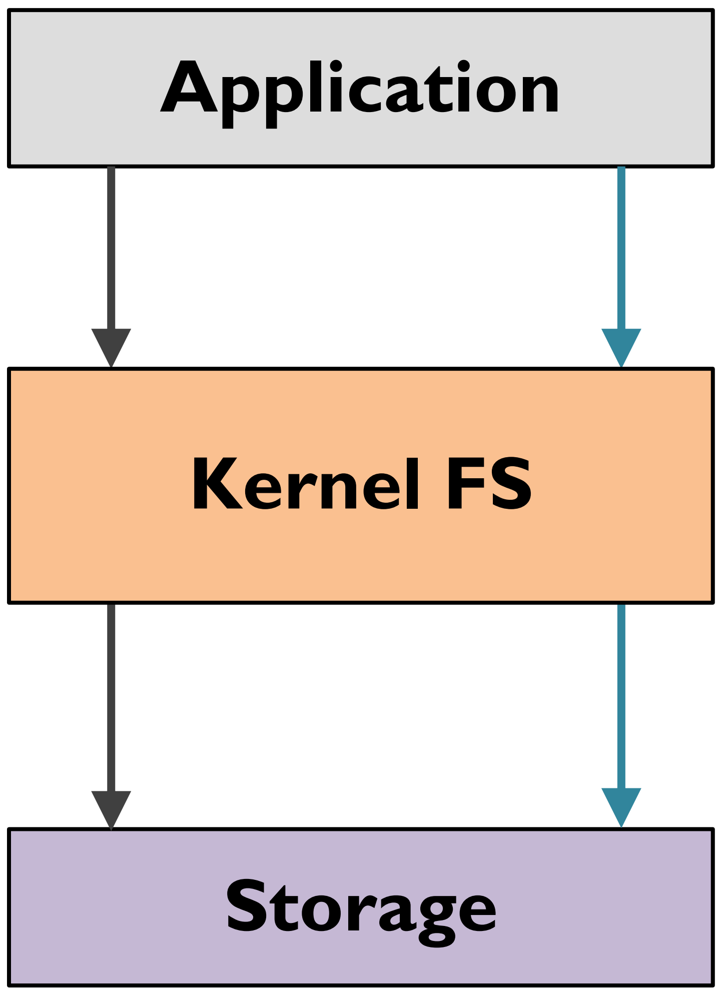
- Small random I/O dominates access patterns
  - Desktop applications (e.g., SQLite)
  - Server applications (e.g., RocksDB, SQL databases)

- Concurrent file access is critical for I/O scalability
  - Threads read/write to shared file concurrently (e.g., RocksDB, MySQL)
  - Processes share files (e.g., HPC applications)

- Crash consistency is important
  - Application-level crash consistency is difficult
  - Application relies on file system for crash consistency

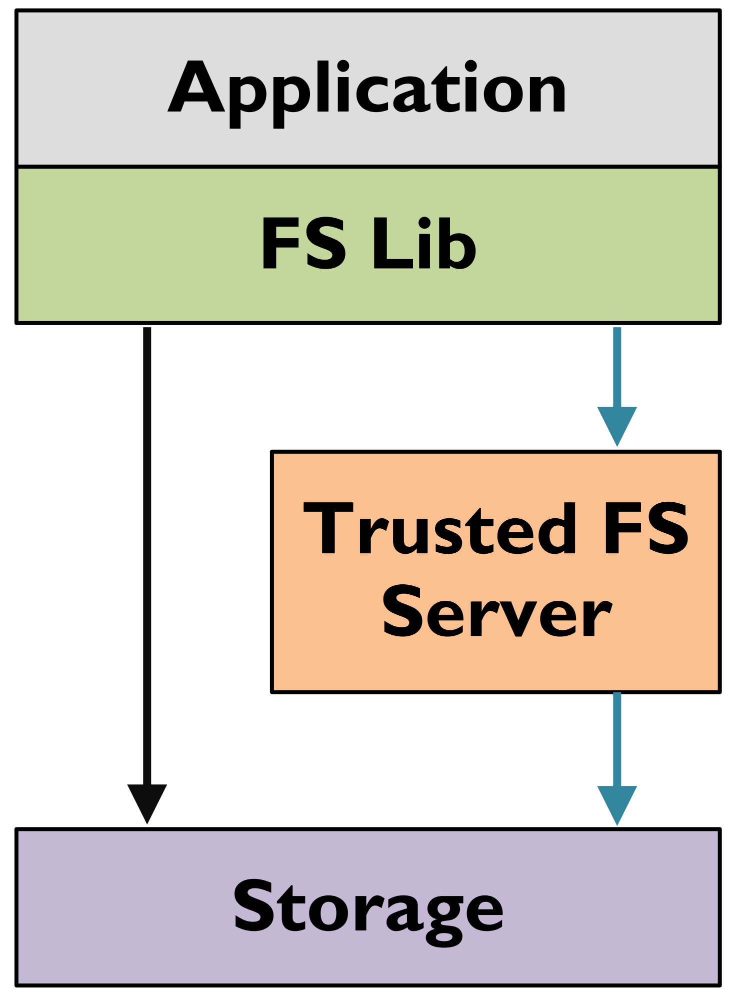
State-of-the-art Designs

Kernel-FS



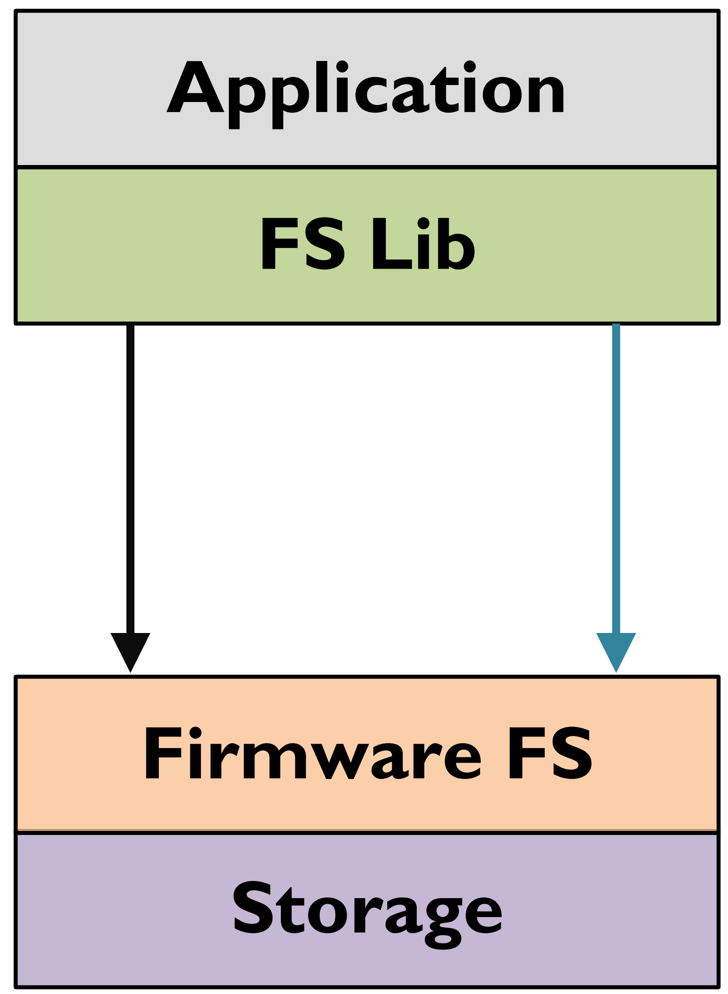
ext4-DAX
NOVA (FAST' 16)

User-FS



Strata (SOSP' 17)
SplitFS (SOSP' 19)

Firmware-FS



DevFS (FAST' 18)
Insider (ATC' 19)

→ : data-plane ops   → : control-plane ops

Outline

- Background
- Motivation
- Design
- Evaluation
- Conclusion

File System Approaches Summary

Classes	File System	Direct-Access	Utilize Host CPU	Utilize Storage CPU	Fine-grained Concurrency
Kernel-FS	ext4-DAX	✗	✓	✗	✗
User-FS	SplitFS	●	✓	✗	✗
Firmware-FS	DevFS	✓	✗	✓	✗
Cross-FS	**CrossFS**	✓	✓	✓	✓

✓ Satisfy

● Partially satisfy

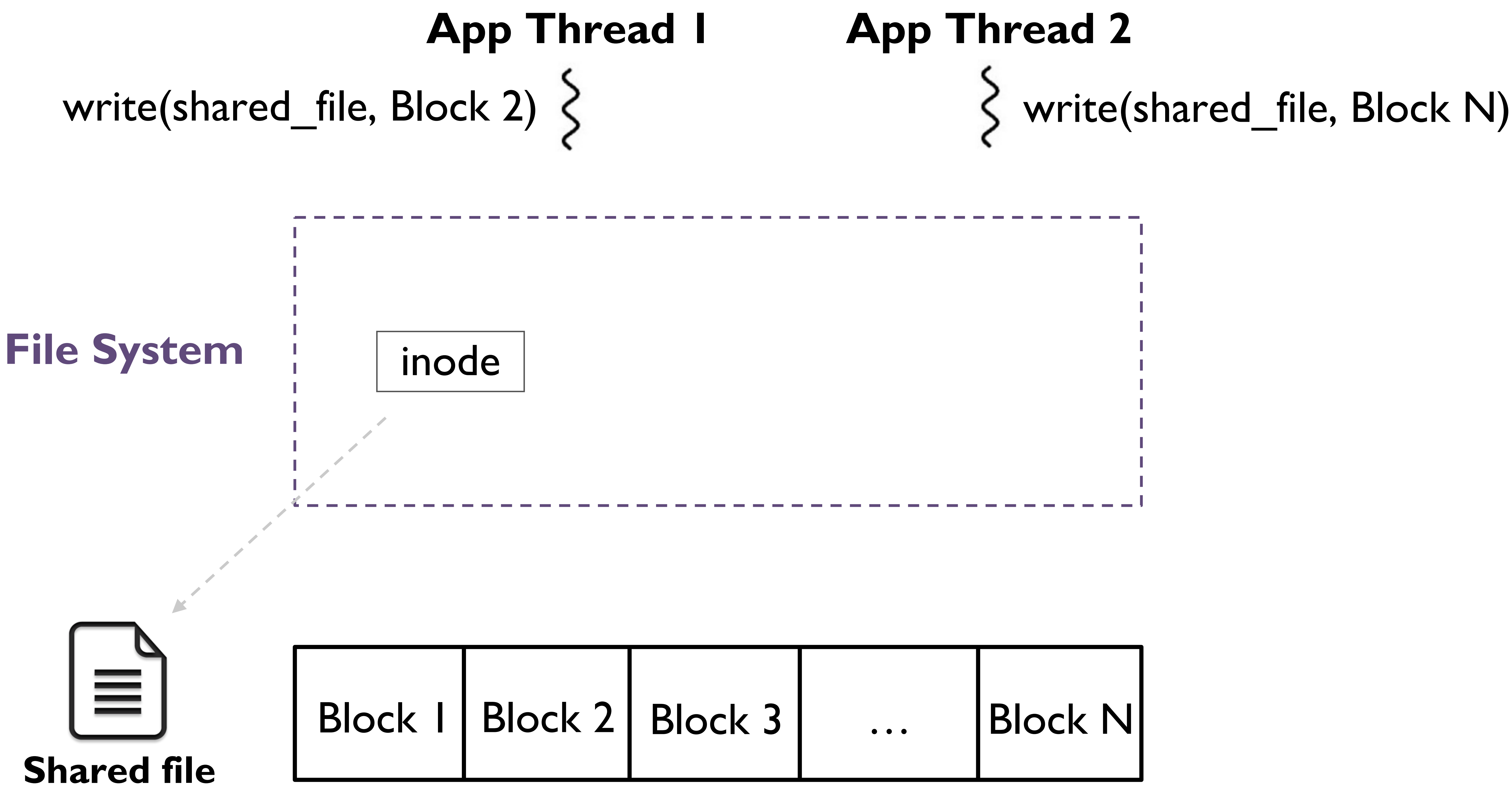
✗ Not satisfy

Ideal for achieving higher performance

More file systems discussed in the paper

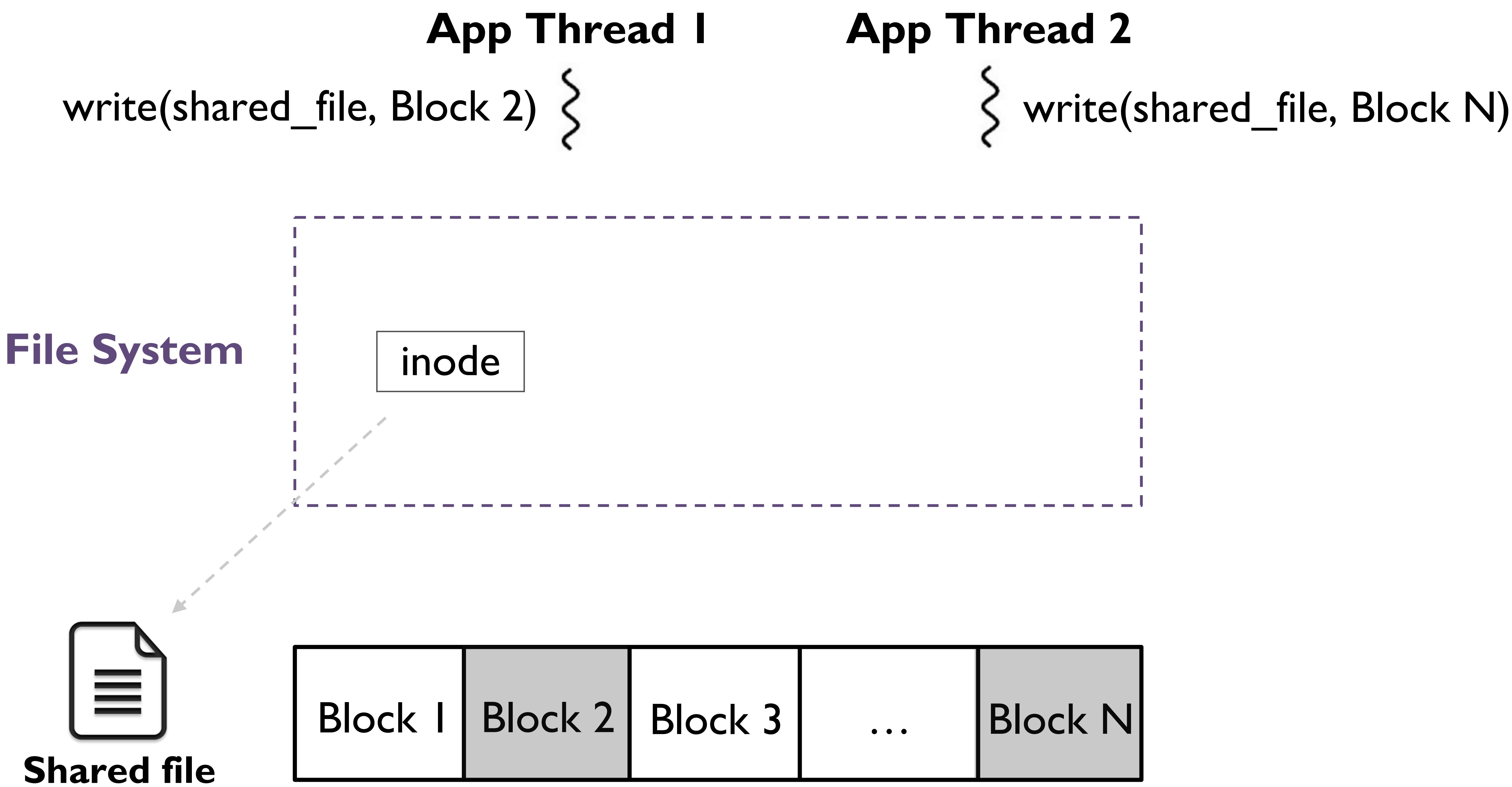
Concurrency Limitations - Analysis

Two threads write to disjoint blocks of a shared file



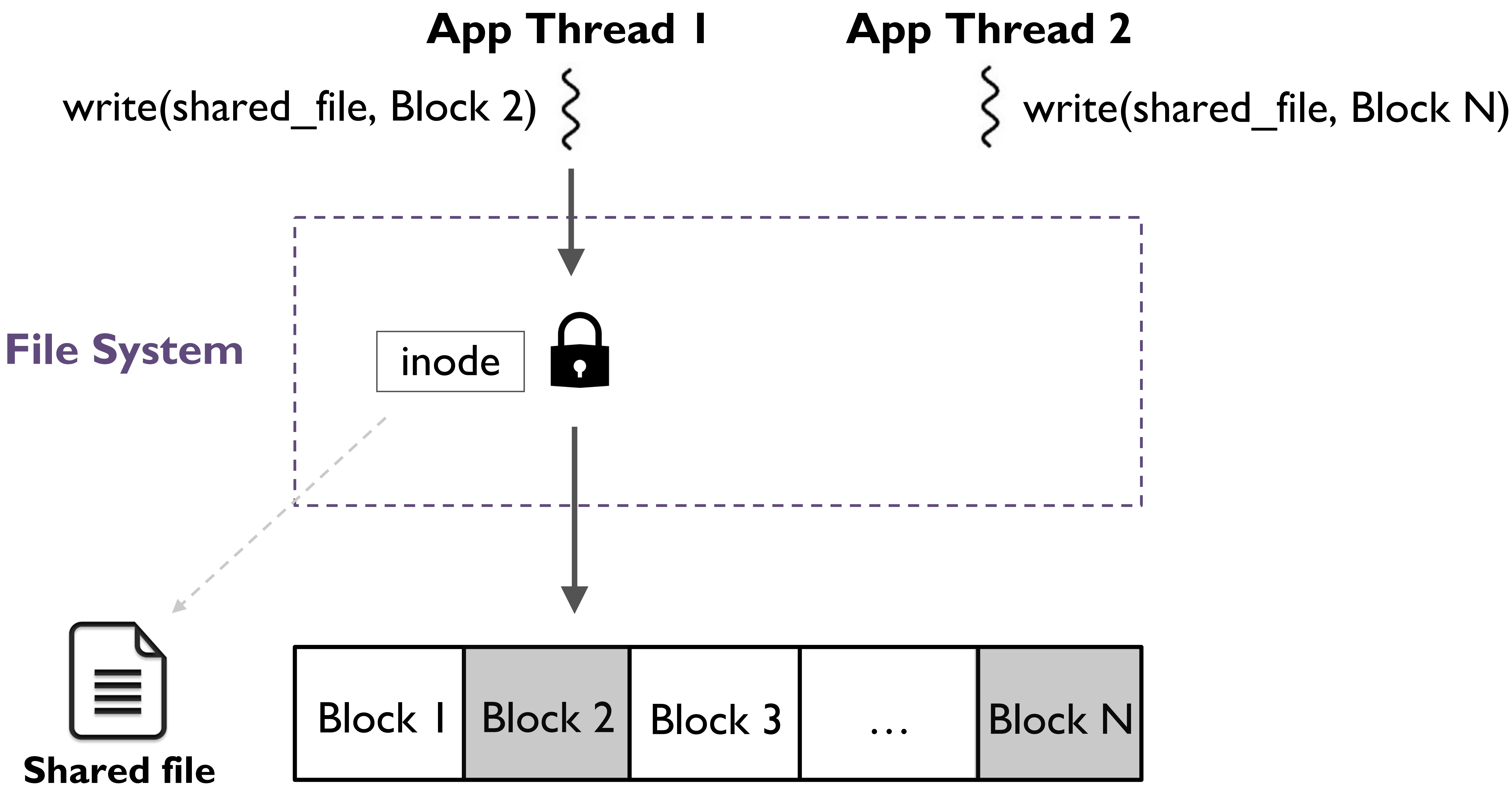
Concurrency Limitations - Analysis

Two threads write to disjoint blocks of a shared file



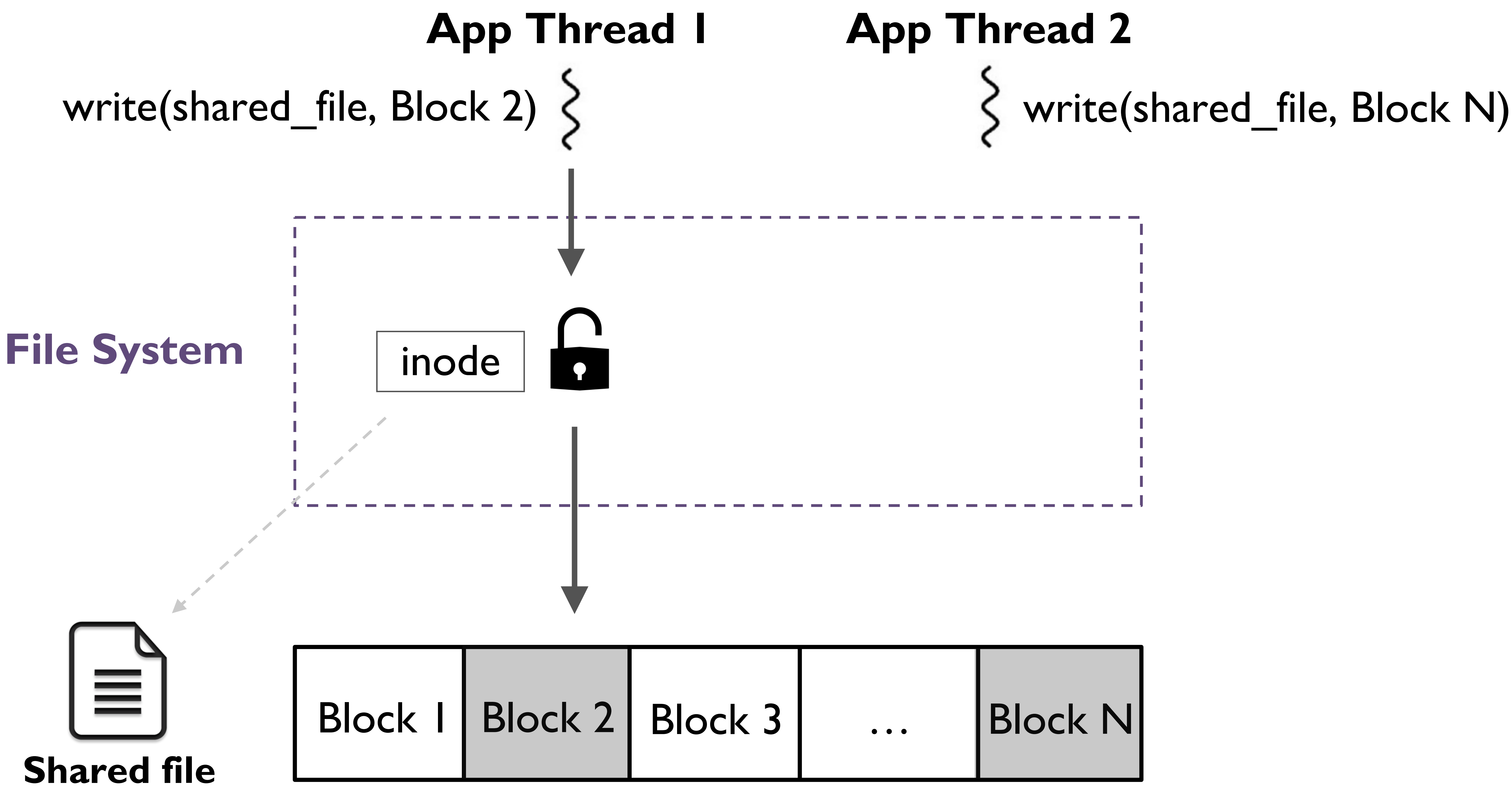
Concurrency Limitations - Analysis

Two threads write to disjoint blocks of a shared file



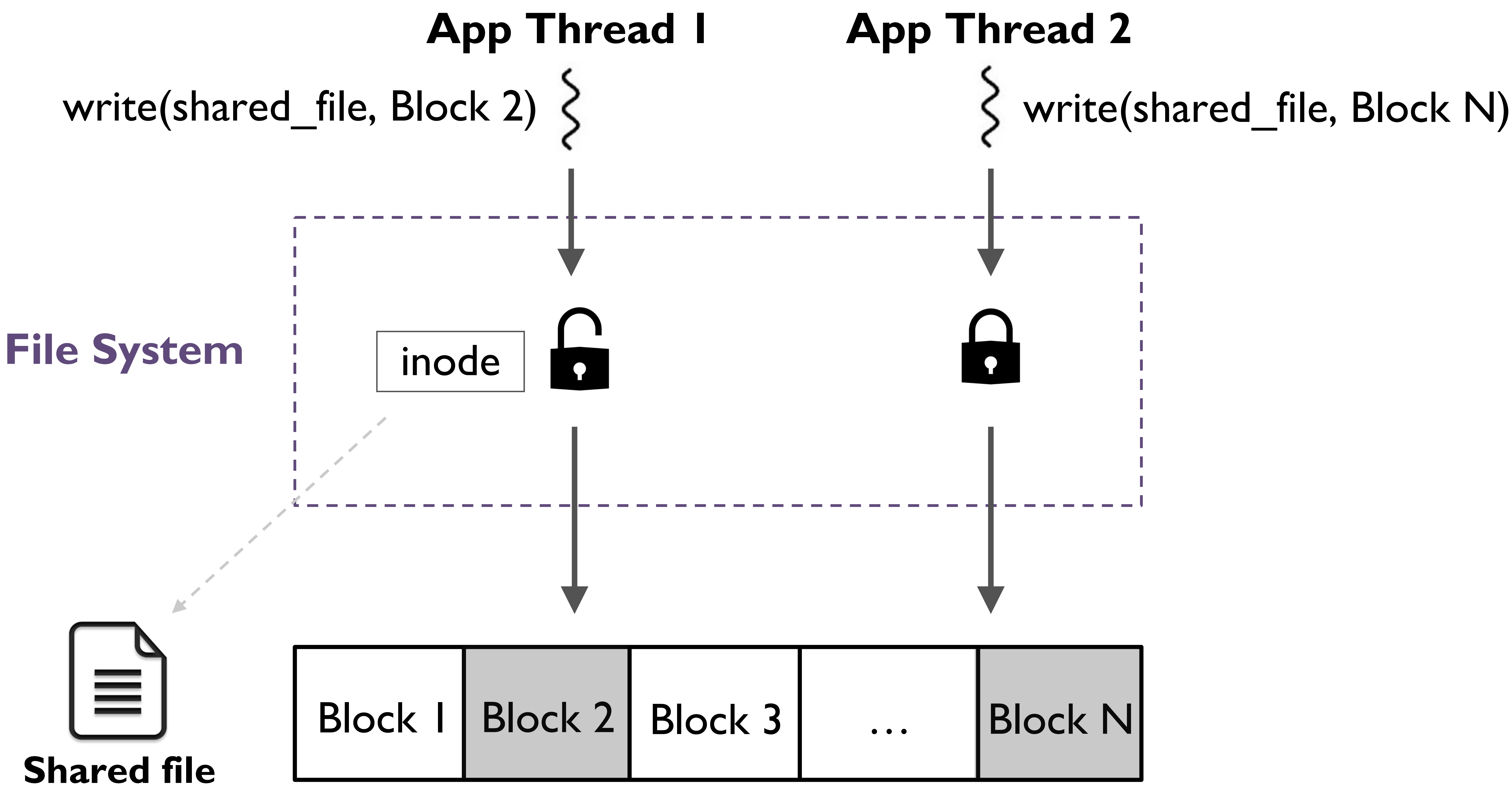
Concurrency Limitations - Analysis

Two threads write to disjoint blocks of a shared file



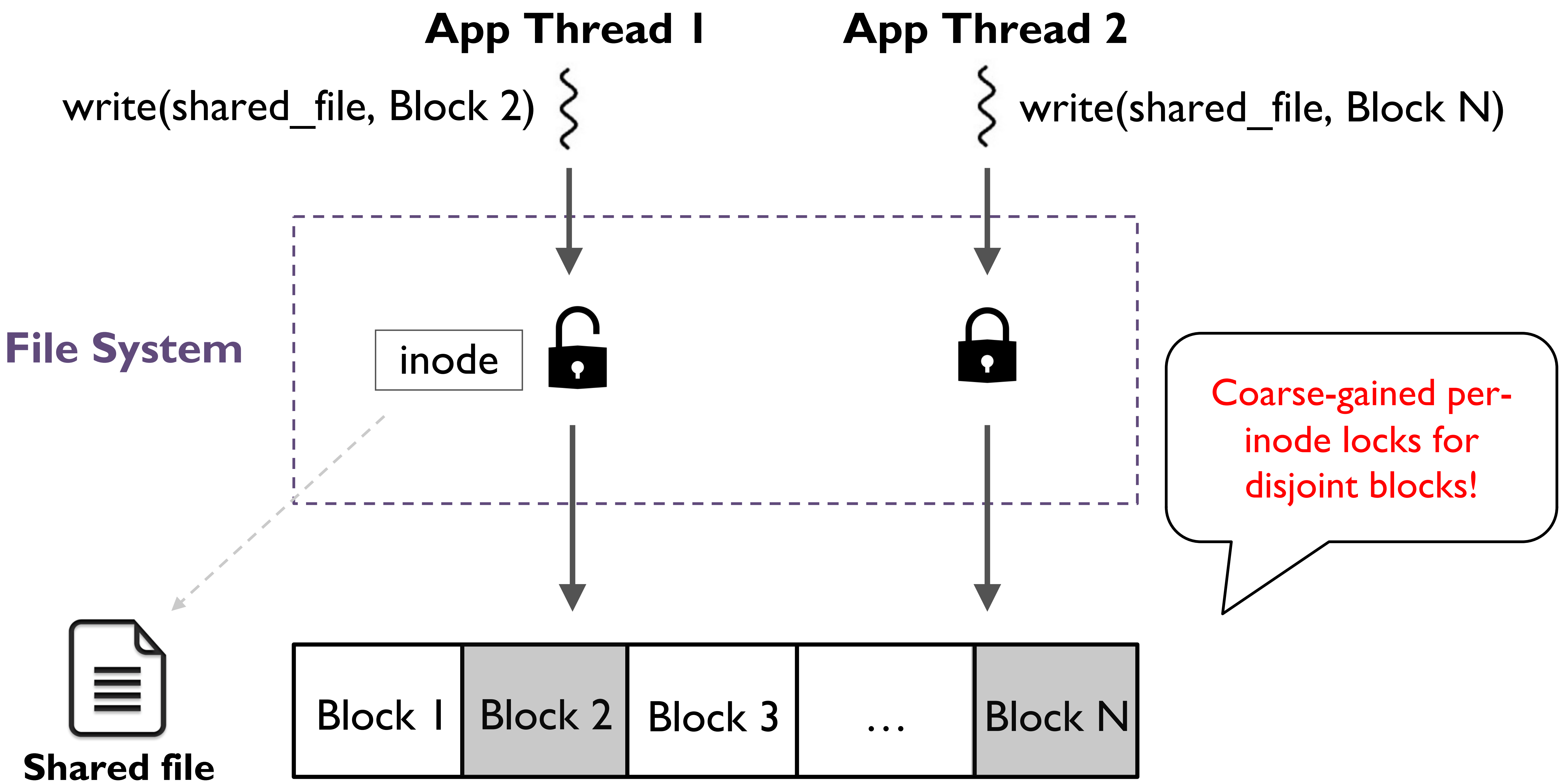
Concurrency Limitations - Analysis

Two threads write to disjoint blocks of a shared file



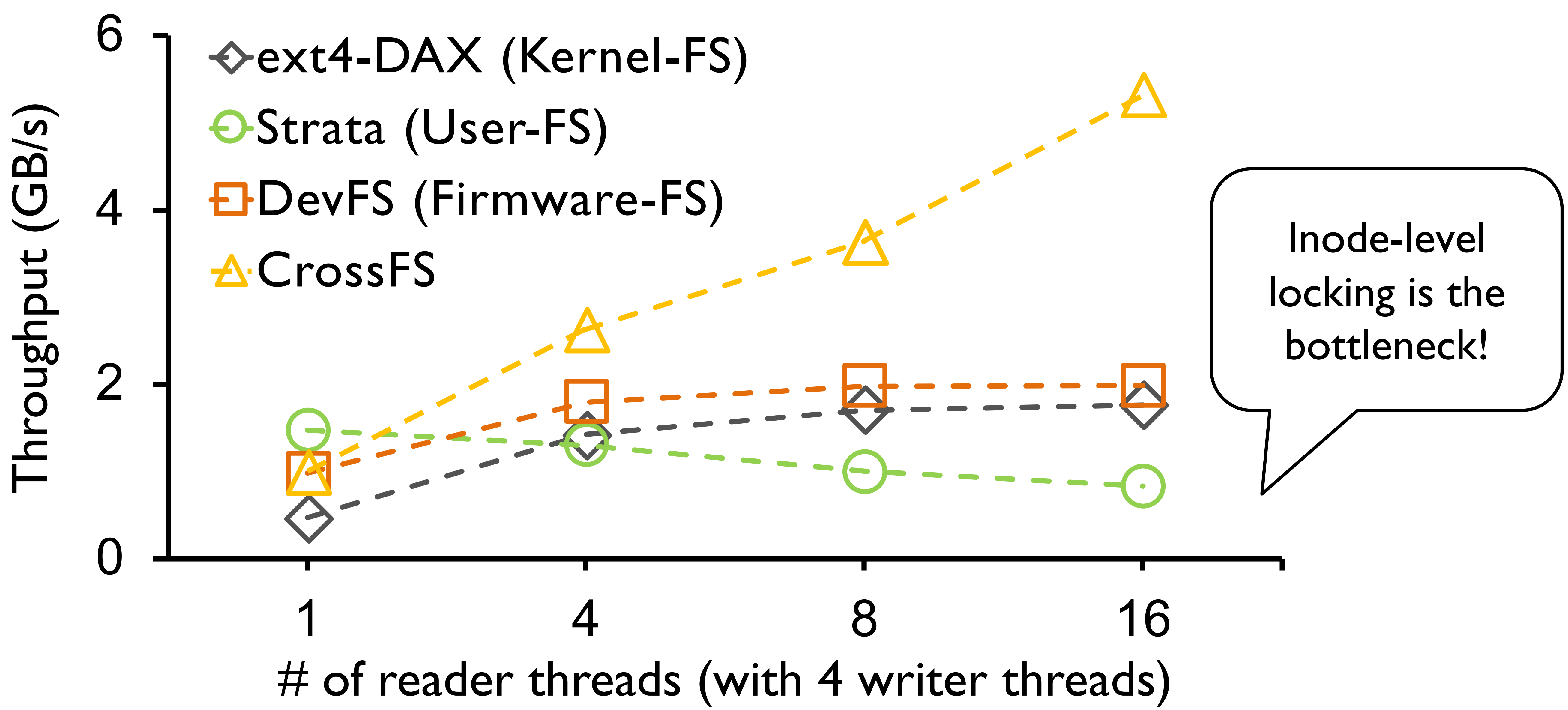
Concurrency Limitations - Analysis

Two threads write to disjoint blocks of a shared file



Concurrency Limitations - Analysis

Concurrent reader and writer threads randomly accessing a shared file



X-axis shows # of reader threads

Y-axis shows the aggregated throughput

Outline

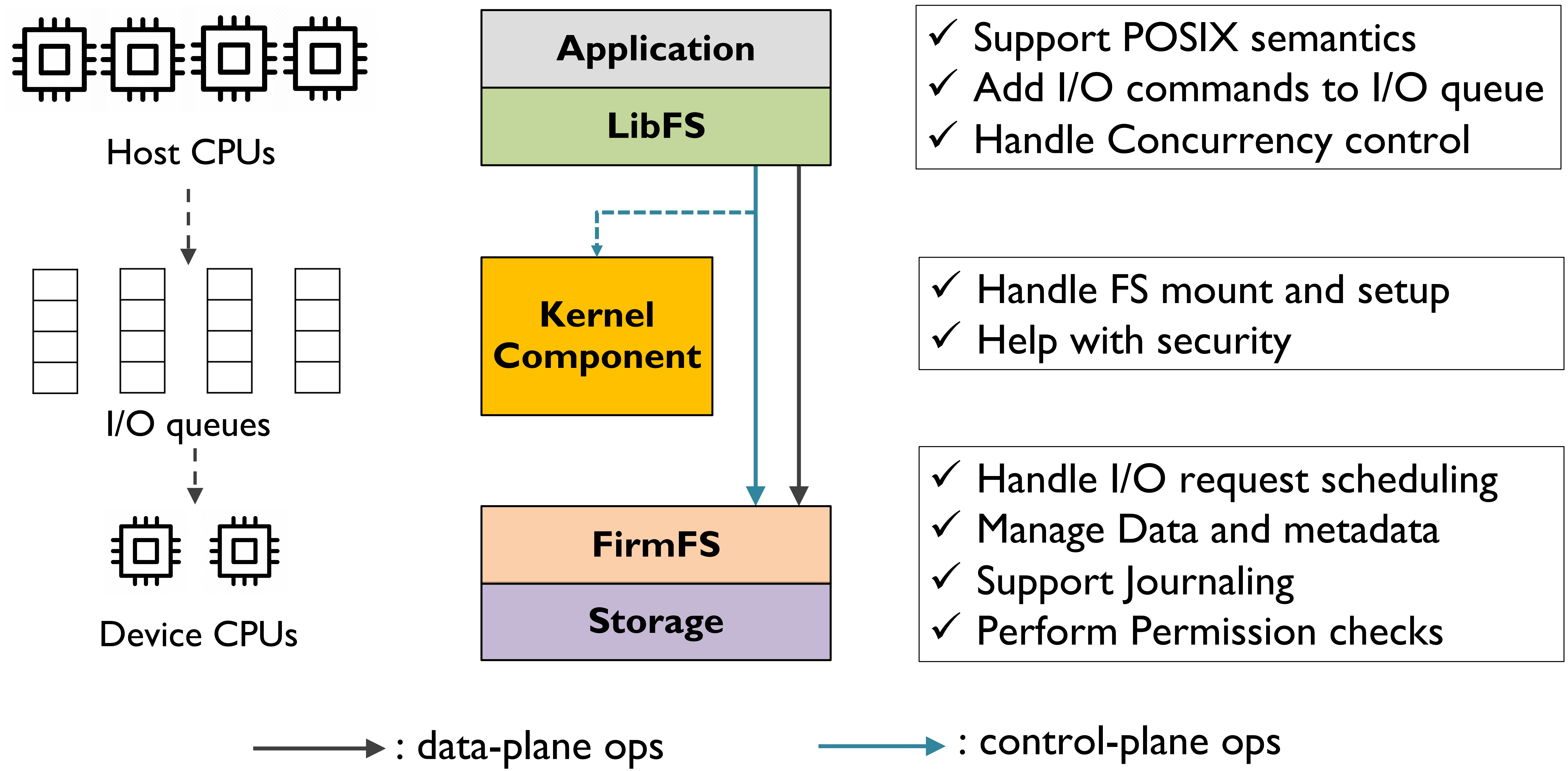
- Background
- Motivation
- Design
- Evaluation
- Conclusion

Our Solution: CrossFS

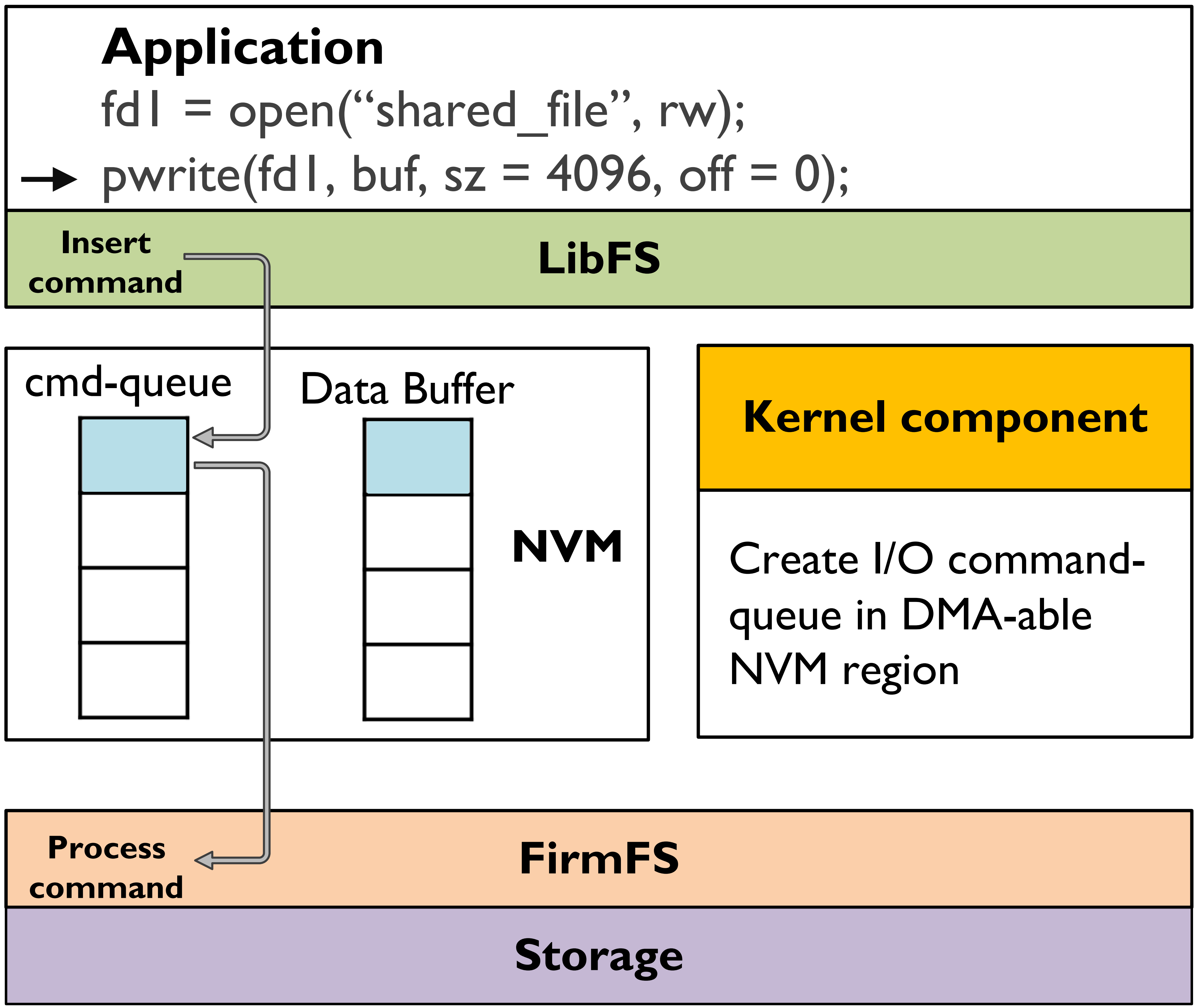
A cross-layered direct-access file system

- Disaggregated FS components to exploit host and device CPUs
- OS-bypass for data-plane and control-plane operations
- File descriptor-based fine-grained concurrency control
- Firmware-level file descriptor's I/O queue scheduling
- Cross-layered crash consistency

CrossFS Components



CrossFS I/O Processing Example



Convert POSIX system call to FirmFS IO commands

Insert I/O commands to I/O queue (cmd-queue + data buffer)

FirmFS fetches I/O commands from command queue

FirmFS performs permission check before processing

Fine-grained Concurrency Control

- Inode-level rw-lock is the bottleneck
  - Even non-overlapping reads and writes are serialized

- Non-overlapping writes could be parallelized
  - Different threads could open different file descriptors for a shared file

Thread 1	Thread 2
fd1 = open("shared_file", rw);	fd2 = open("shared_file", rw);
pwrite(fd1, buf, sz=4096, **off=0**);	pwrite(fd2, buf, sz=4096, **off=8192**);

- File descriptor is a natural concurrency abstraction
  - Independent file descriptors for a shared file
  - Map each file descriptor to an independent hardware I/O queue
  - 64K I/O queues in modern storage

Fine-grained Concurrency Control

Align each file descriptor to a dedicated I/O queue (**FD-queue**)

Thread 1
fd1 = open("shared_file", rw);
pwrite(fd1, buf, sz = 4096, **off = 0**);

Thread 2
fd2 = open("shared_file", rw);
pwrite(fd2, buf, sz = 4096, **off =8192**);

LibFS

NVM

Kernel component

Create FD-queue in DMA-able NVM region

FirmFS

Fine-grained Concurrency Control

Align each file descriptor to a dedicated I/O queue (**FD-queue**)

Thread 1

→ fd1 = open(“shared_file”, rw);

pwrite(fd1, buf, sz = 4096, **off = 0**);

Thread 2

fd2 = open(“shared_file”, rw);

pwrite(fd2, buf, sz = 4096, **off =8192**);

LibFS

NVM

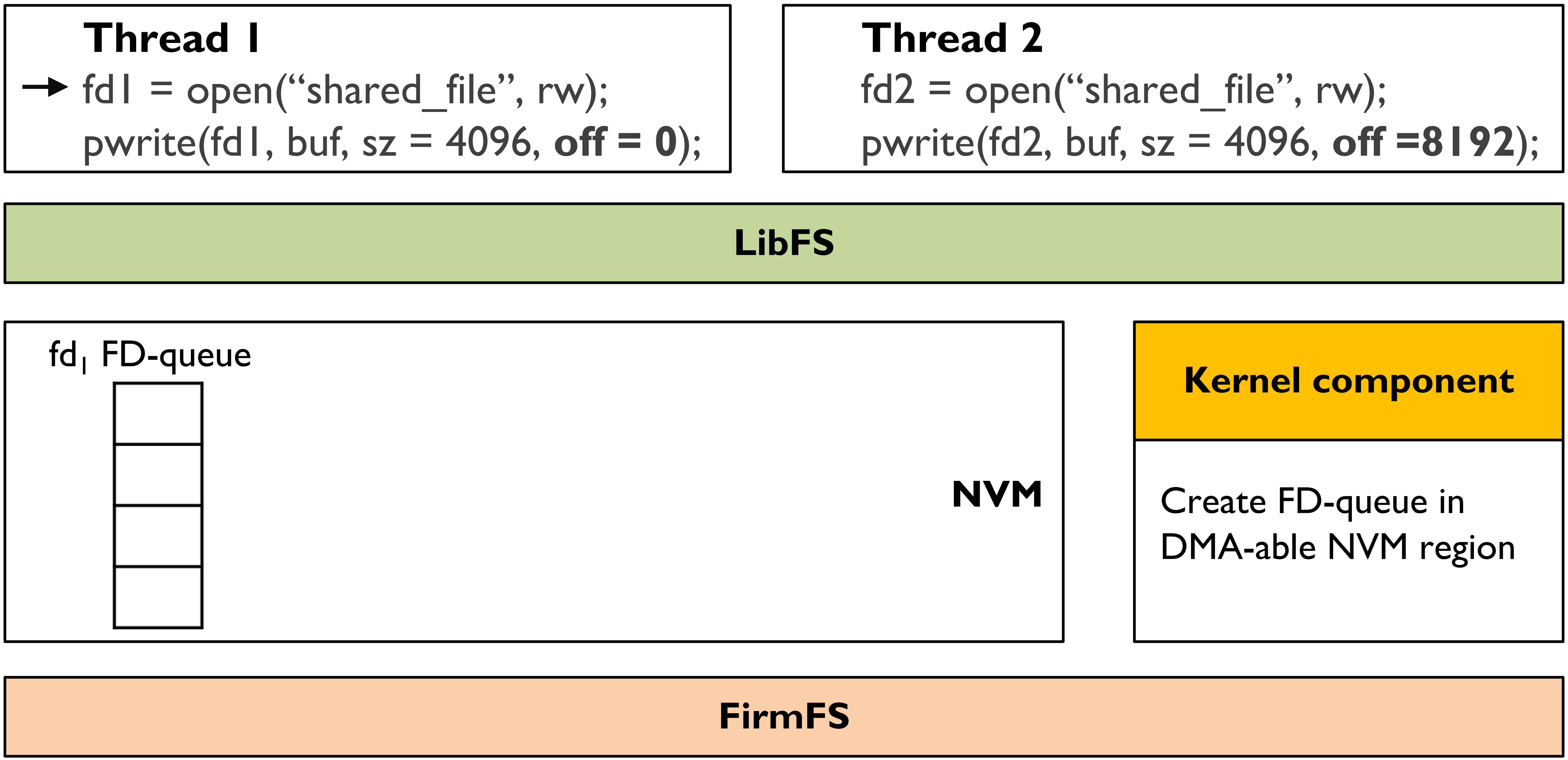
Kernel component

Create FD-queue in DMA-able NVM region

FirmFS

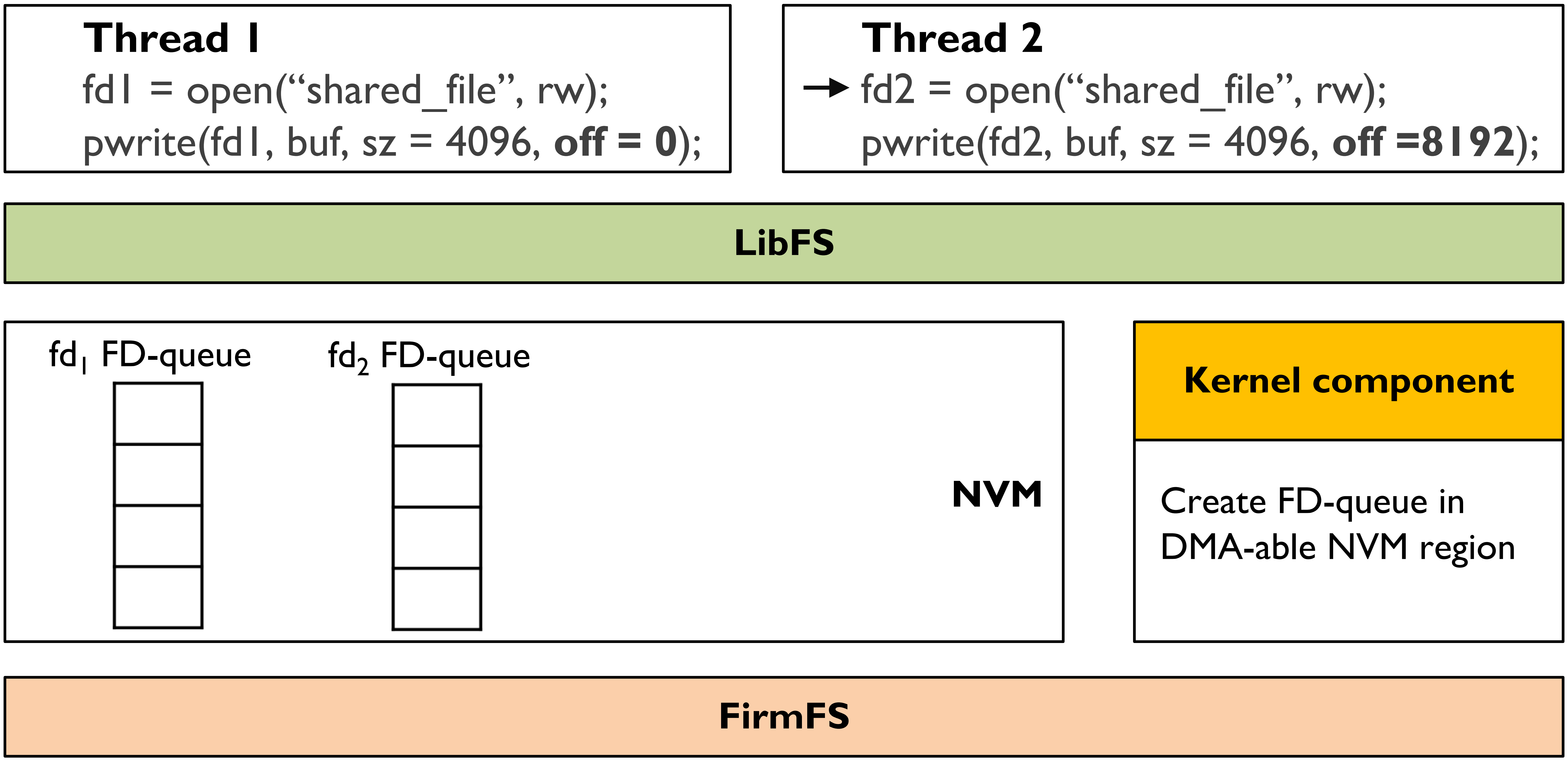
Fine-grained Concurrency Control

Align each file descriptor to a dedicated I/O queue (**FD-queue**)



Fine-grained Concurrency Control

Align each file descriptor to a dedicated I/O queue (**FD-queue**)



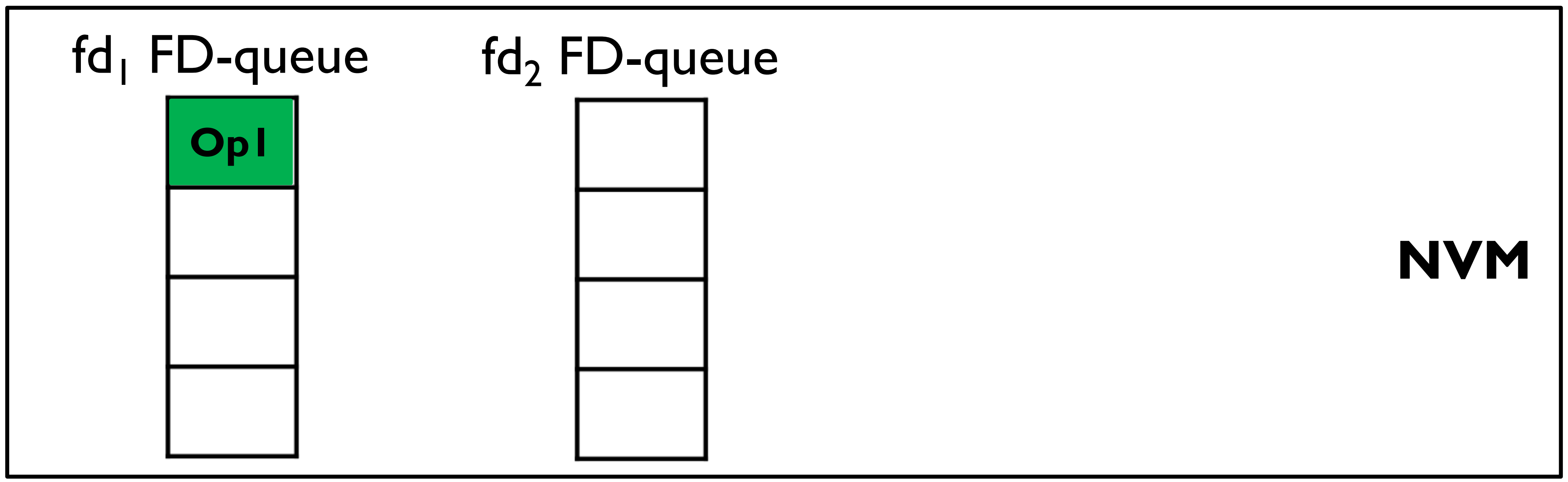
Fine-grained Concurrency Control

Align each file descriptor to a dedicated I/O queue (**FD-queue**)

Thread 1
fd1 = open("shared_file", rw);
→ pwrite(fd1, buf, sz = 4096, **off = 0**);

Thread 2
fd2 = open("shared_file", rw);
pwrite(fd2, buf, sz = 4096, **off =8192**);

LibFS



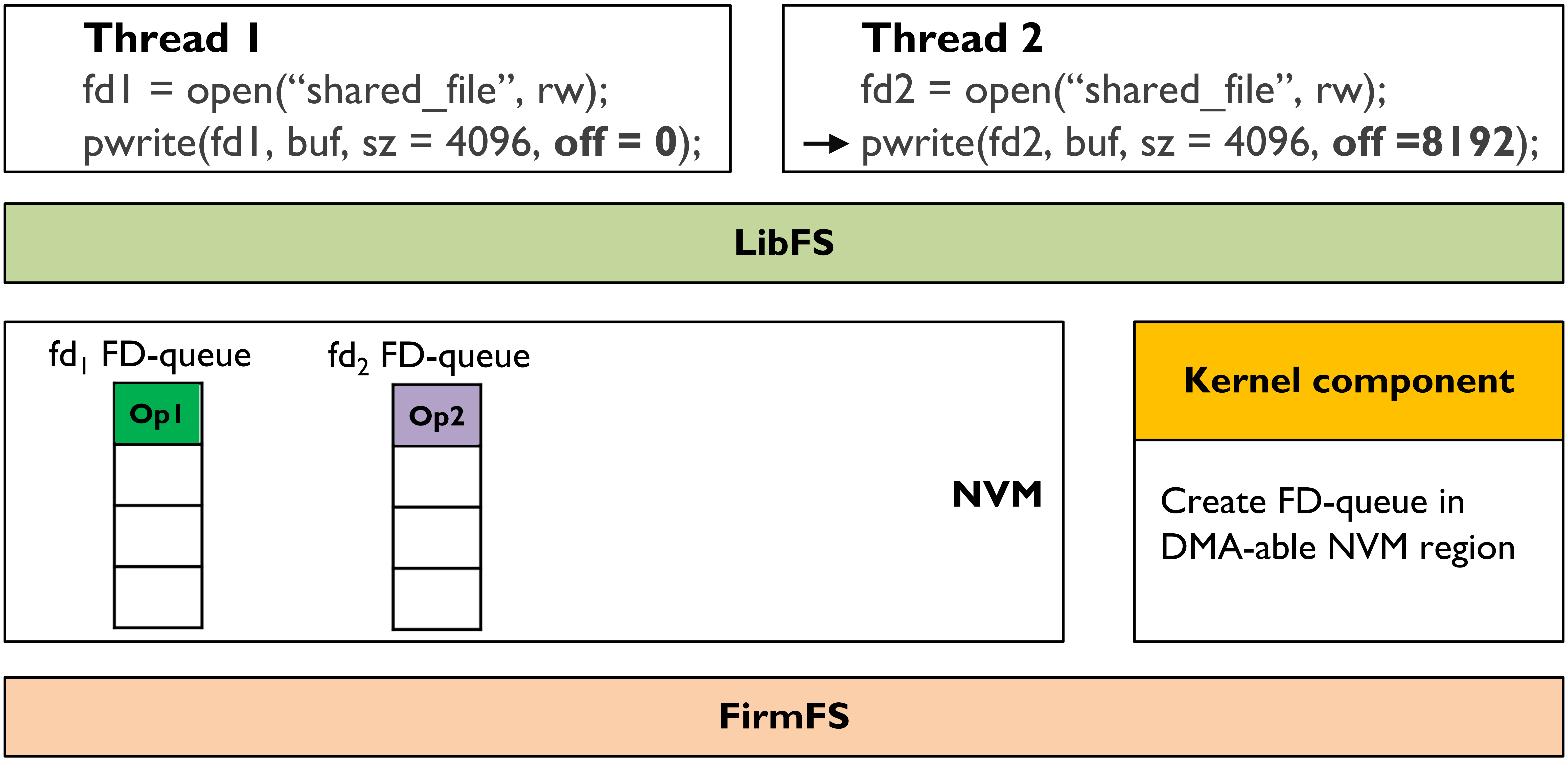
Kernel component

Create FD-queue in DMA-able NVM region

FirmFS

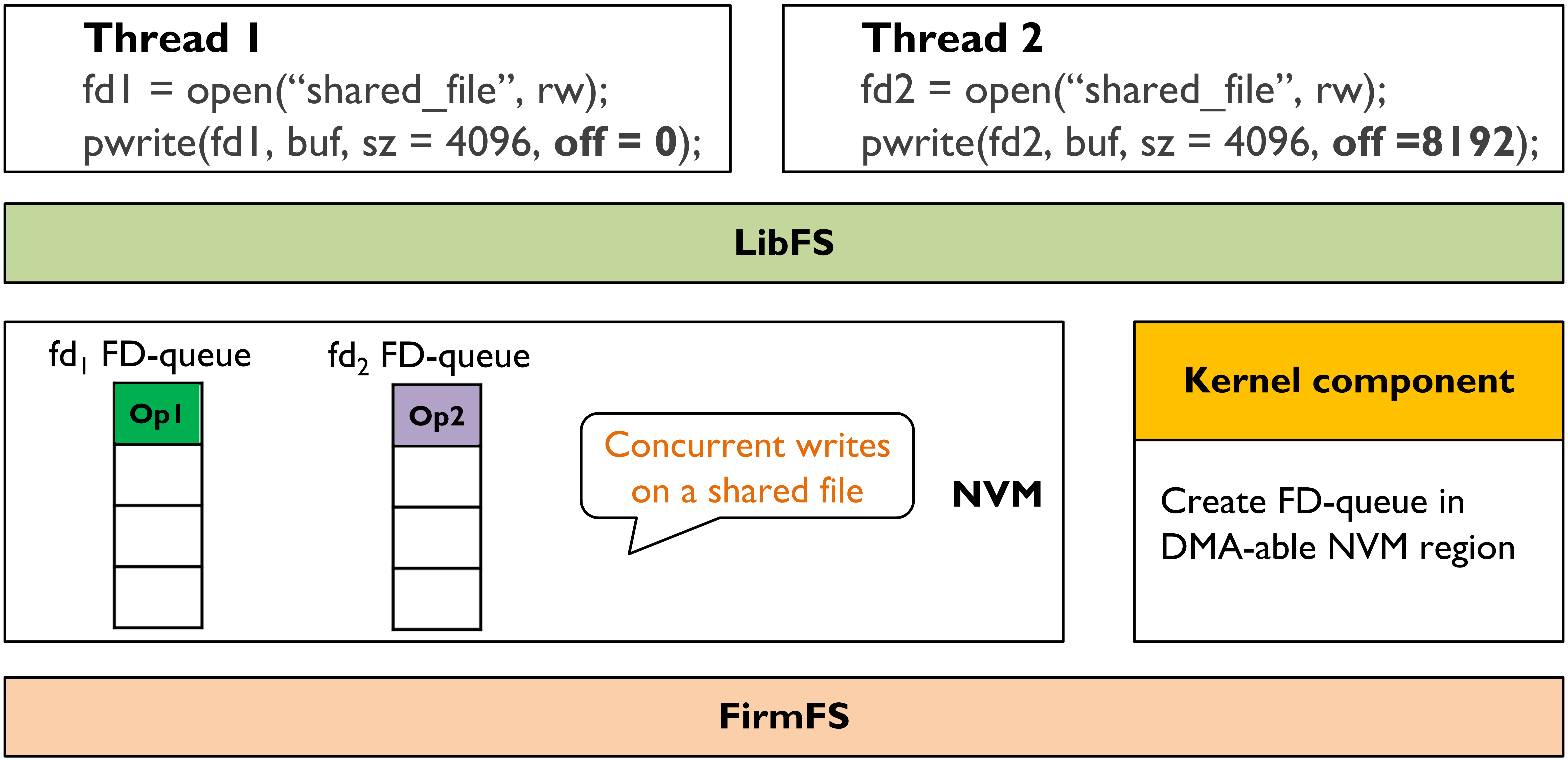
Fine-grained Concurrency Control

Align each file descriptor to a dedicated I/O queue (**FD-queue**)



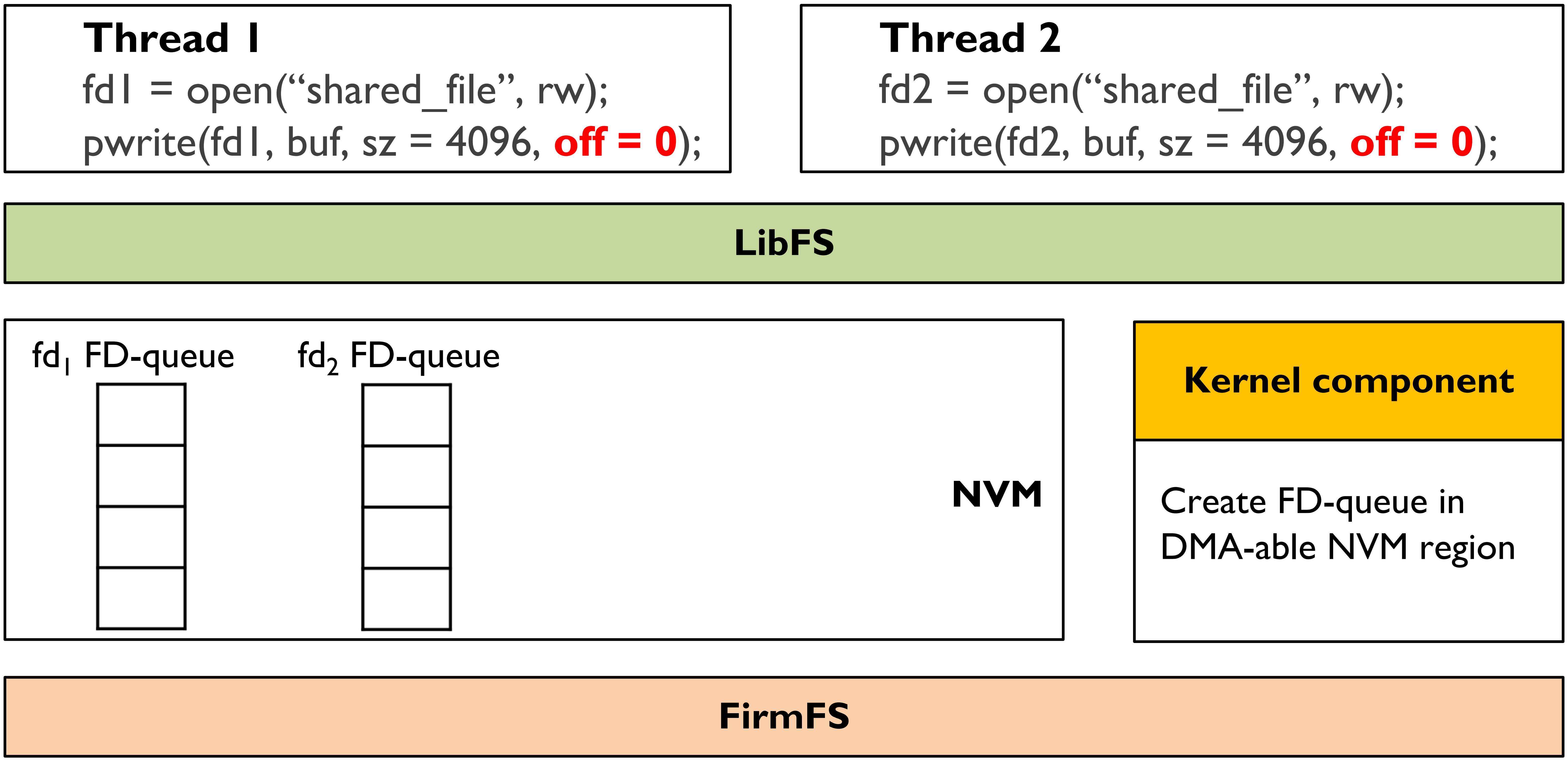
Fine-grained Concurrency Control

Align each file descriptor to a dedicated I/O queue (**FD-queue**)



Fine-grained Concurrency Control

What about overlapping concurrent writes?



Fine-grained Concurrency Control

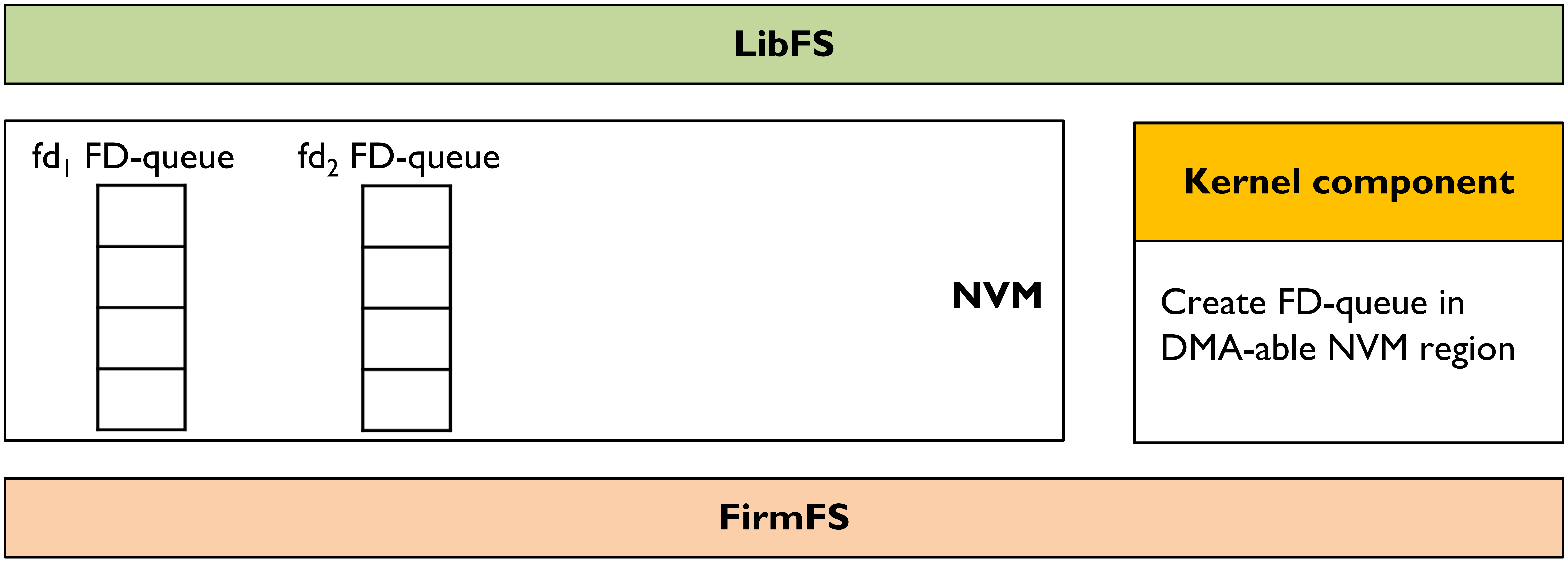
What about overlapping concurrent writes?

Thread 1

```
fd1 = open("shared_file", rw);
→ pwrite(fd1, buf, sz = 4096, off = 0);
```

Thread 2

```
fd2 = open("shared_file", rw);
→ pwrite(fd2, buf, sz = 4096, off = 0);
```



Fine-grained Concurrency Control

What about overlapping concurrent writes?

Thread 1

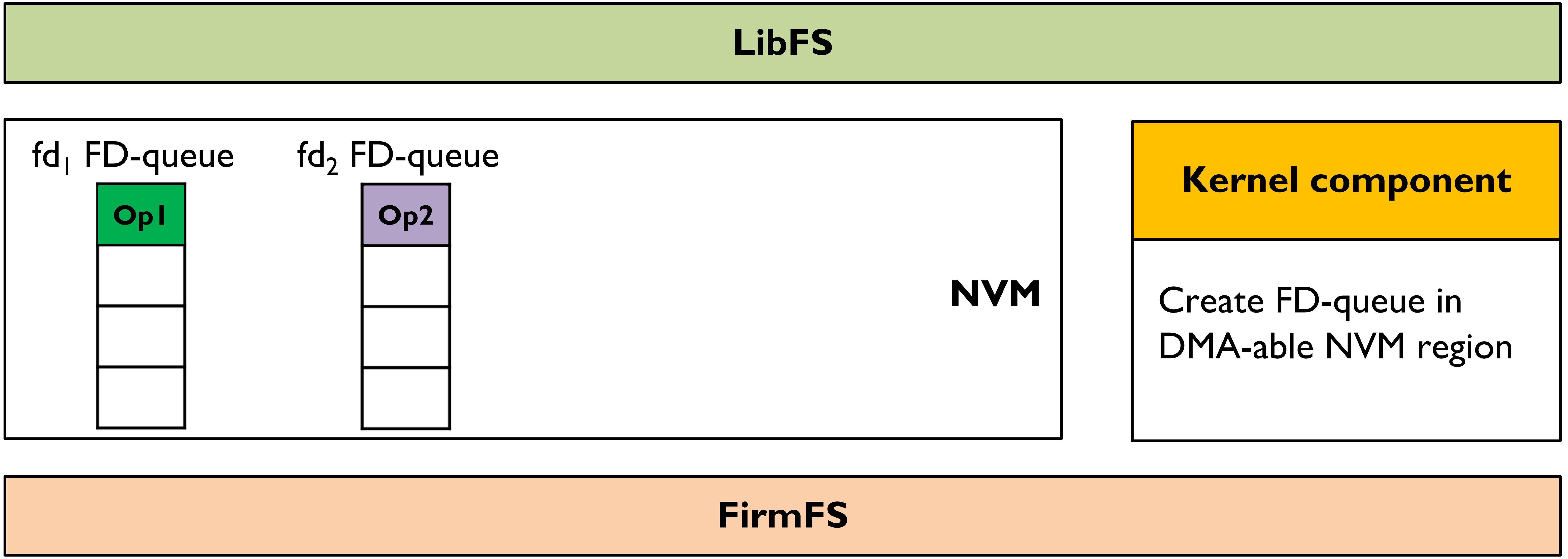
fd1 = open("shared_file", rw);

→ pwrite(fd1, buf, sz = 4096, **off = 0**);

Thread 2

fd2 = open("shared_file", rw);

→ pwrite(fd2, buf, sz = 4096, **off = 0**);

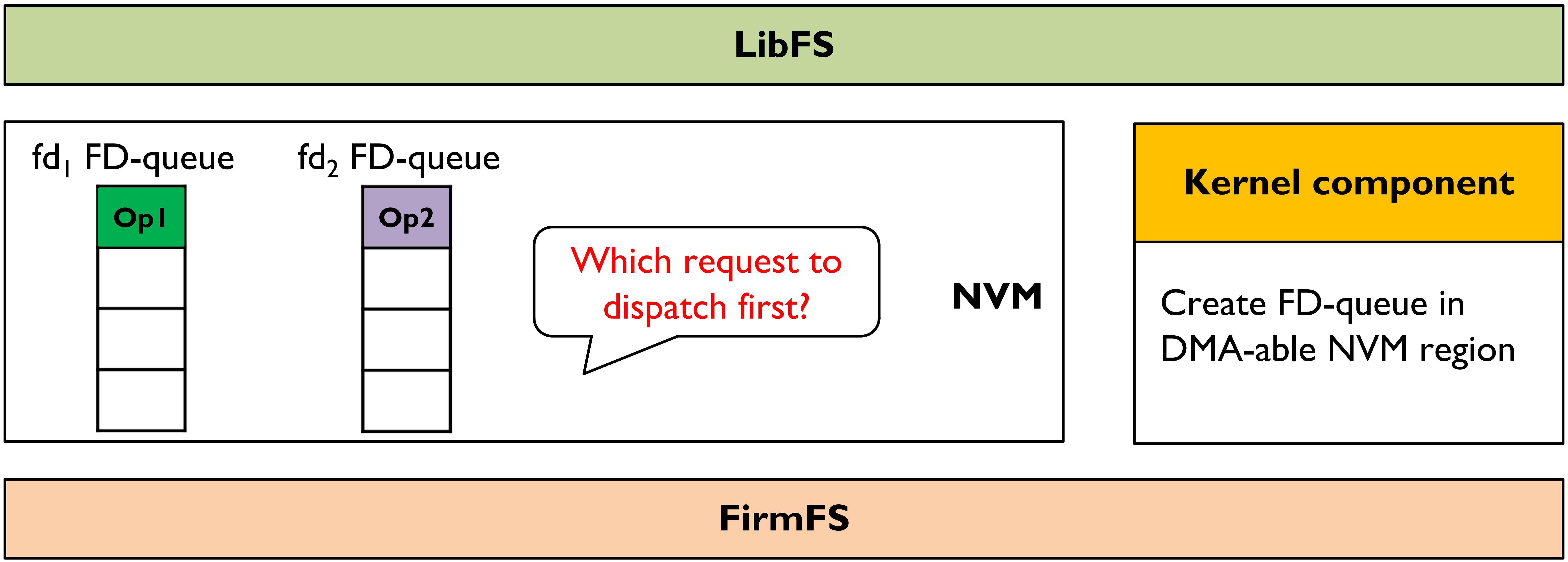


Fine-grained Concurrency Control

What about overlapping concurrent writes?

Thread 1
```
fd1 = open("shared_file", rw);
→ pwrite(fd1, buf, sz = 4096, off = 0);
```

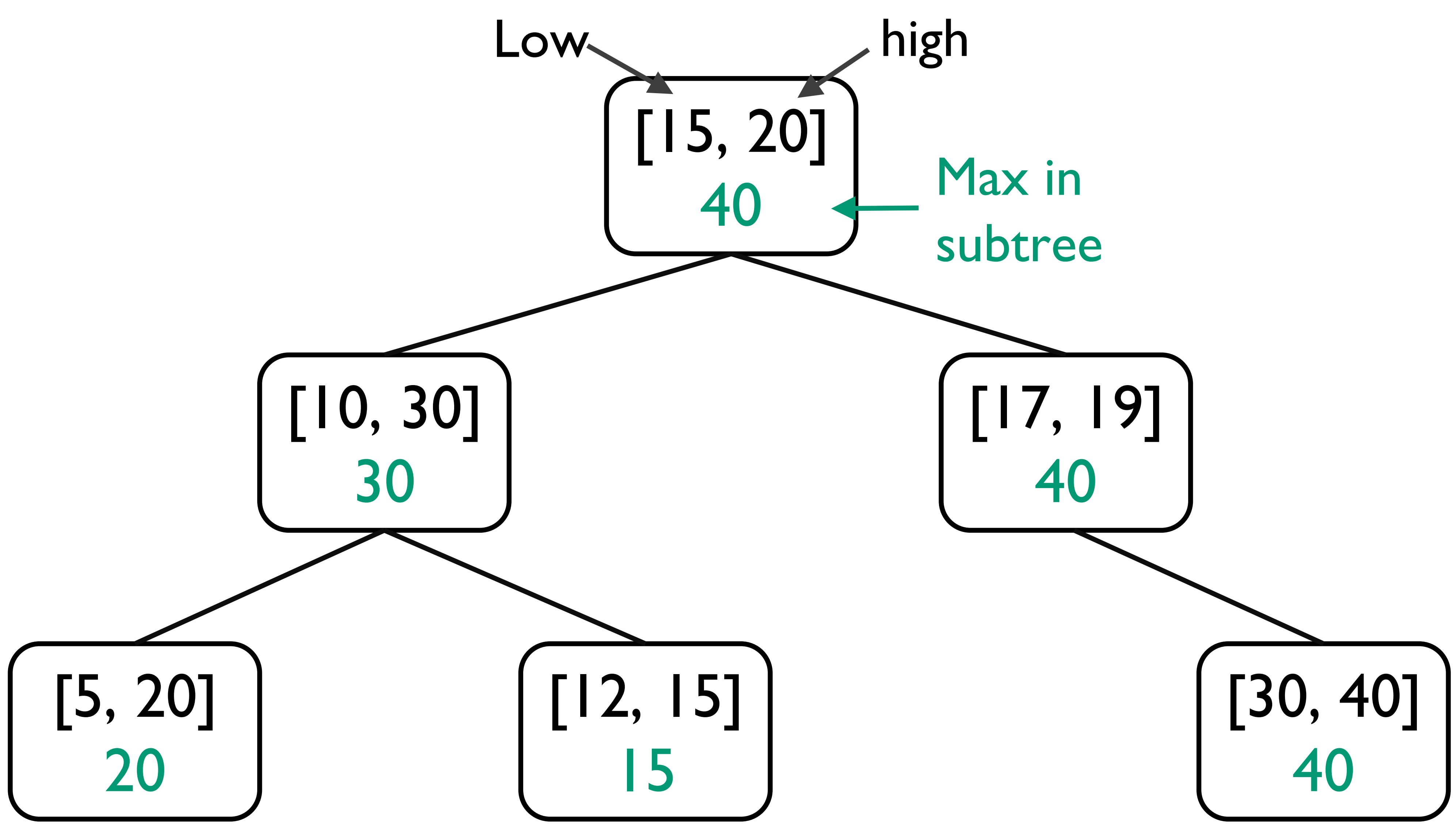
Thread 2
```
fd2 = open("shared_file", rw);
→ pwrite(fd2, buf, sz = 4096, off = 0);
```



Fine-grained Concurrency Control

Resolving overlapping writes – Interval tree data structure

Efficient lookup of overlapping blocks requests across FD-queues



CrossFS uses interval tree to store I/O block ranges for in-flight requests

Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

```
     fd1 = open("shared_file", rw);
Op1: pwrite(fd1, buf, sz=4096, off=0)
```

Thread 2

```
     fd2 = open("shared_file", rw);
Op2: pwrite(fd2, buf, sz=4096, off = 4096)
Op3: pwrite(fd2, buf, sz = 4096, off = 0);
```

LibFS

fd_1 FD-queue

fd_2 FD-queue

NVM

Kernel component

Create FD-queue in DMA-able NVM region

Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);

Op1: pwrite(fd1, buf, sz=4096, off=0)

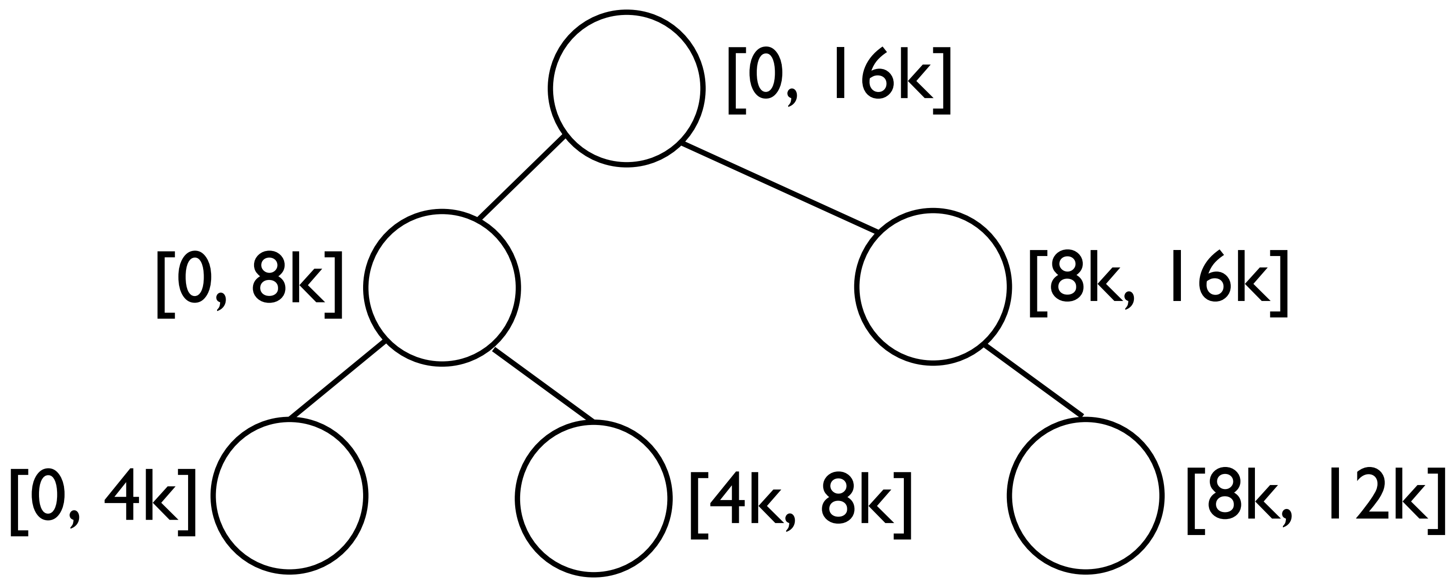
Thread 2

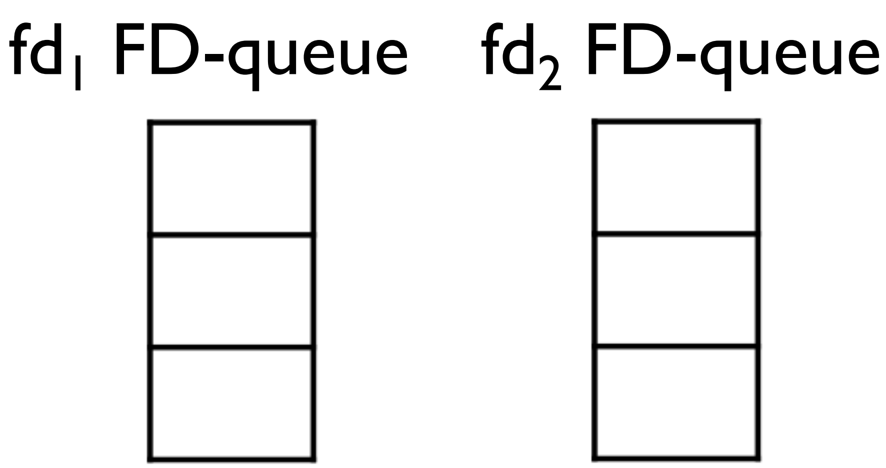
fd2 = open("shared_file", rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

Op3: pwrite(fd2, buf, sz = 4096, off = 0);

LibFS





NVM

Kernel component

Create FD-queue in DMA-able NVM region

Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open(“shared_file”, rw);

→ **Op1**: pwrite(fd1, buf, sz=4096, off=0)

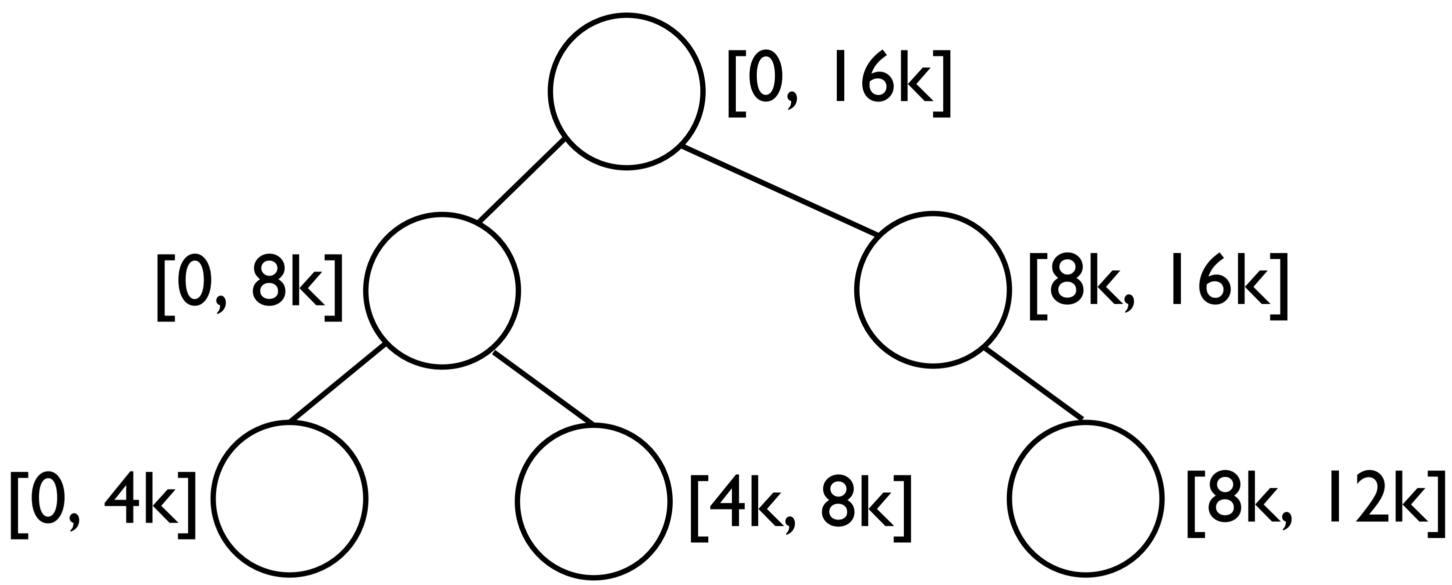
Thread 2

fd2 = open(“shared_file”, rw);

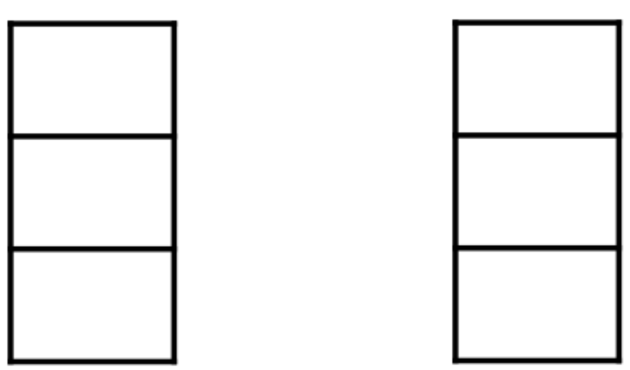
Op2: pwrite(fd2, buf, sz=4096, off = 4096)

Op3: pwrite(fd2, buf, sz = 4096, off = 0);

LibFS



fd_1 FD-queue fd_2 FD-queue



NVM

Kernel component

Create FD-queue in DMA-able NVM region

Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);

→ **Op1**: pwrite(fd1, buf, sz=4096, off=0)

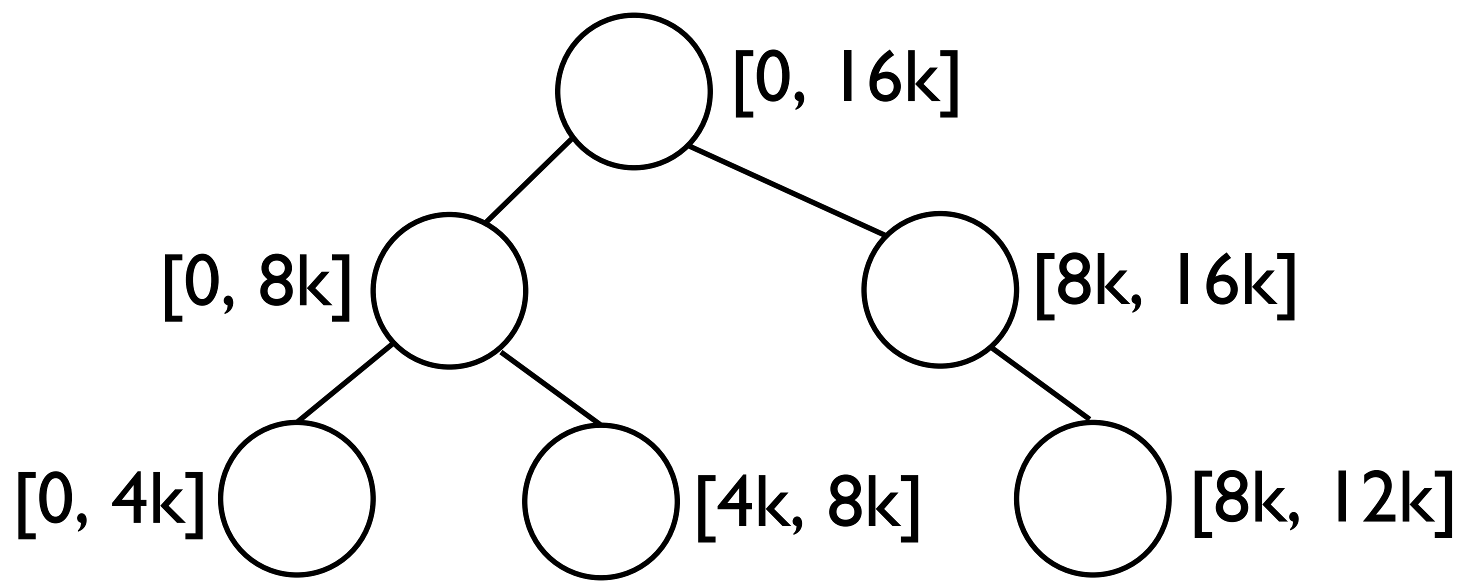
Thread 2

fd2 = open("shared_file", rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

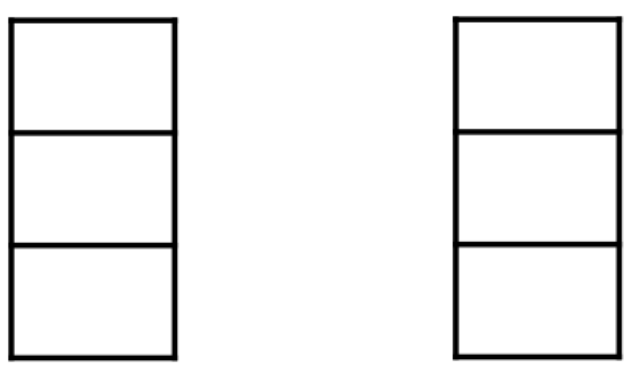
Op3: pwrite(fd2, buf, sz = 4096, off = 0);

LibFS



Interval tree lock

fd_1 FD-queue fd_2 FD-queue



NVM

Kernel component

Create FD-queue in DMA-able NVM region

Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);

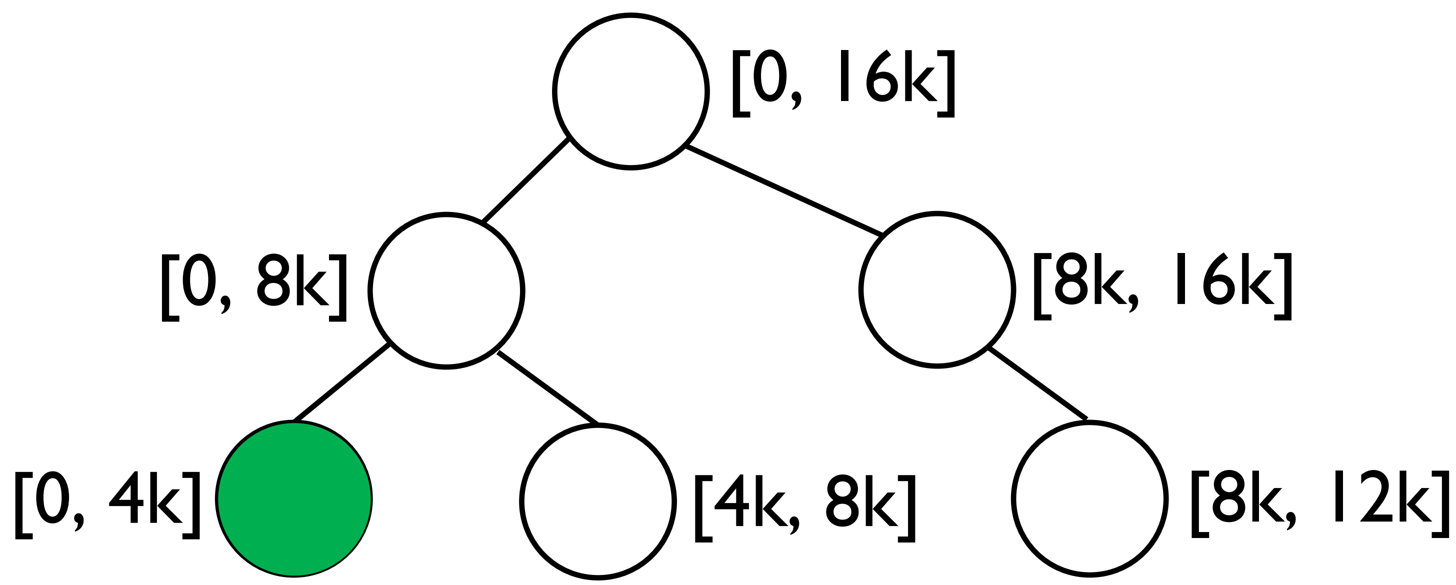
→ **Op1**: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open("shared_file", rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

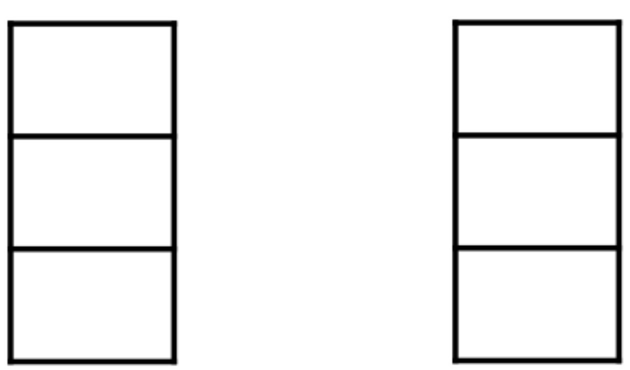
Op3: pwrite(fd2, buf, sz = 4096, off = 0);



Interval tree lock

LibFS

fd$_1$ FD-queue fd$_2$ FD-queue



NVM

Kernel component

Create FD-queue in DMA-able NVM region

Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);

→ **Op1**: pwrite(fd1, buf, sz=4096, off=0)

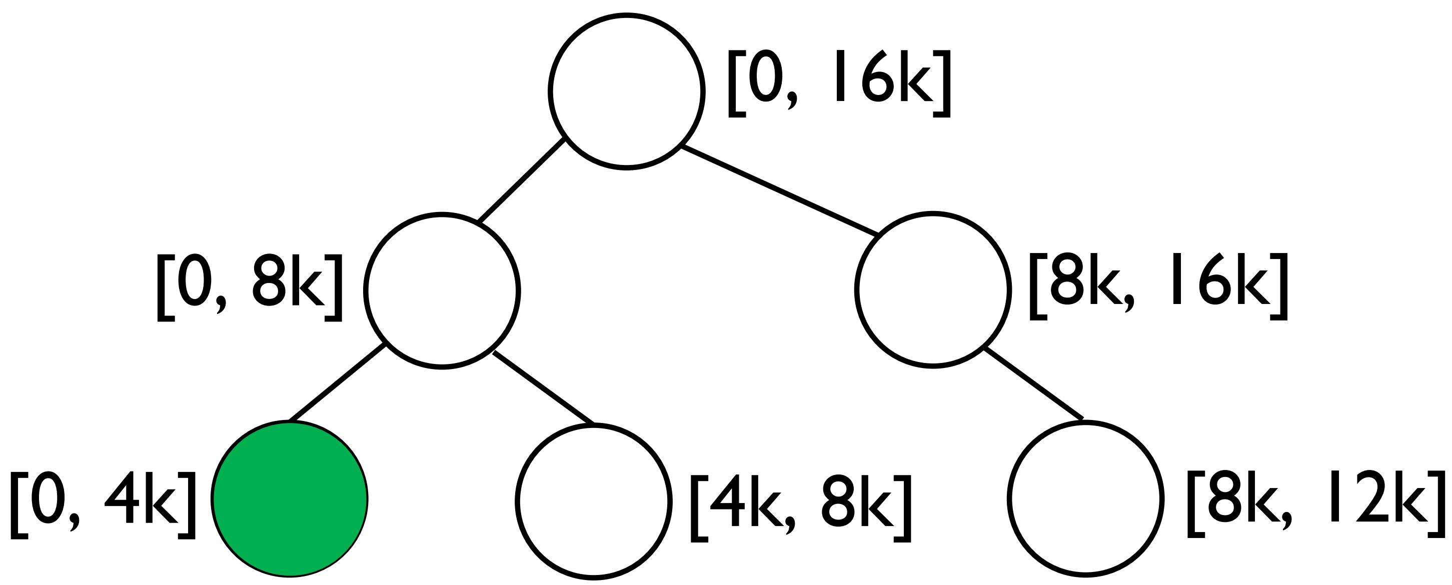
Thread 2

fd2 = open("shared_file", rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

Op3: pwrite(fd2, buf, sz = 4096, off = 0);

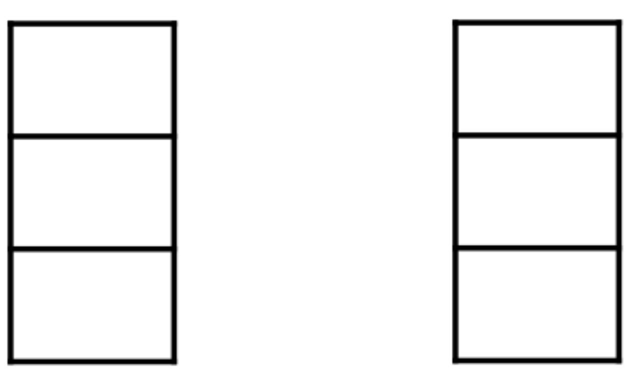
LibFS



Interval tree lock

Interval tree unlock

fd_1 FD-queue fd_2 FD-queue



NVM

Kernel component

Create FD-queue in DMA-able NVM region

Fine-grained Concurrency Control

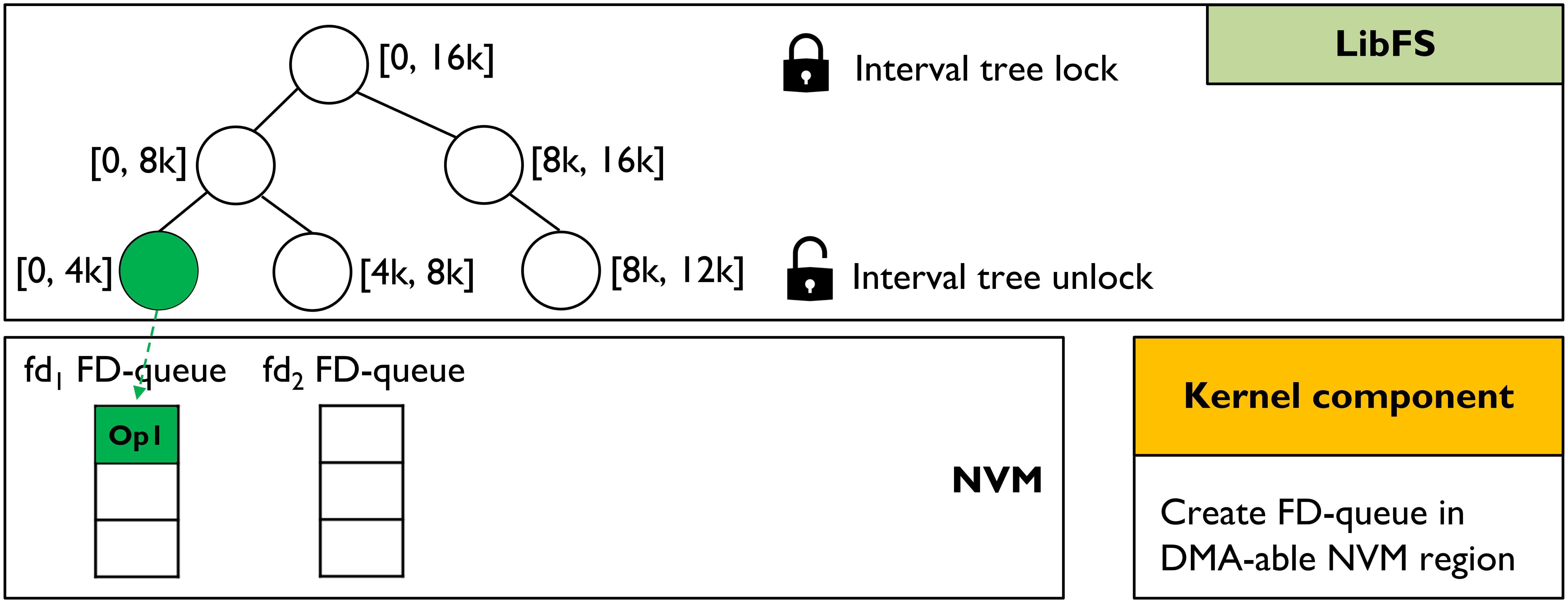
Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);
→ **Op1**: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open("shared_file", rw);
Op2: pwrite(fd2, buf, sz=4096, off = 4096)
Op3: pwrite(fd2, buf, sz = 4096, off = 0);



Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open(“shared_file”, rw);

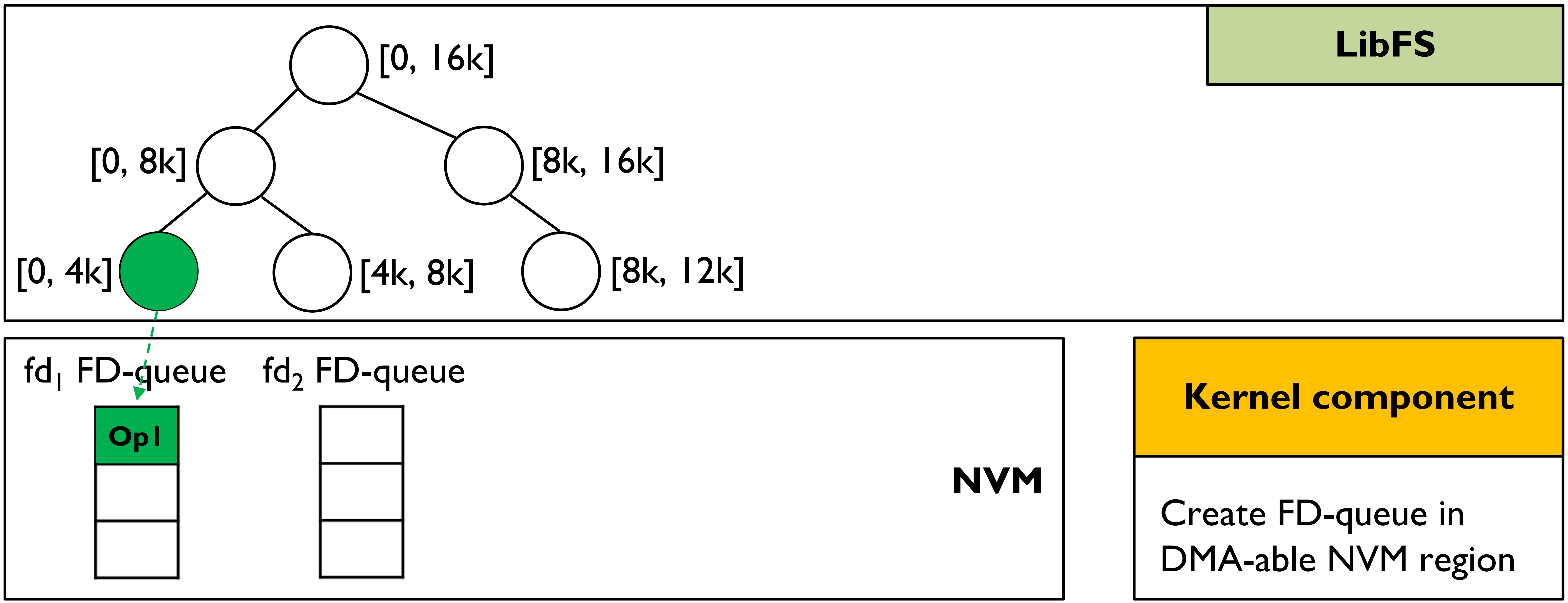
→ **Op1**: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open(“shared_file”, rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

Op3: pwrite(fd2, buf, sz = 4096, off = 0);



Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);

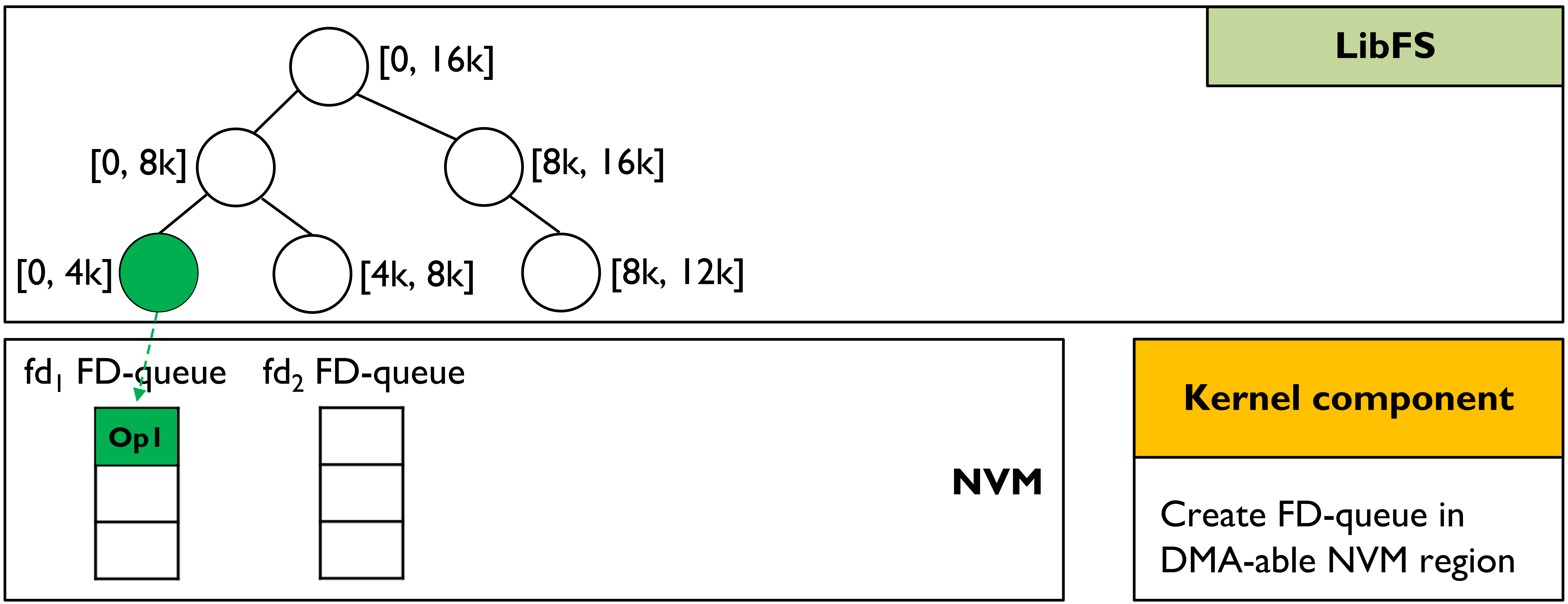
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open("shared_file", rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

Op3: pwrite(fd2, buf, sz = 4096, off = 0);



Fine-grained Concurrency Control

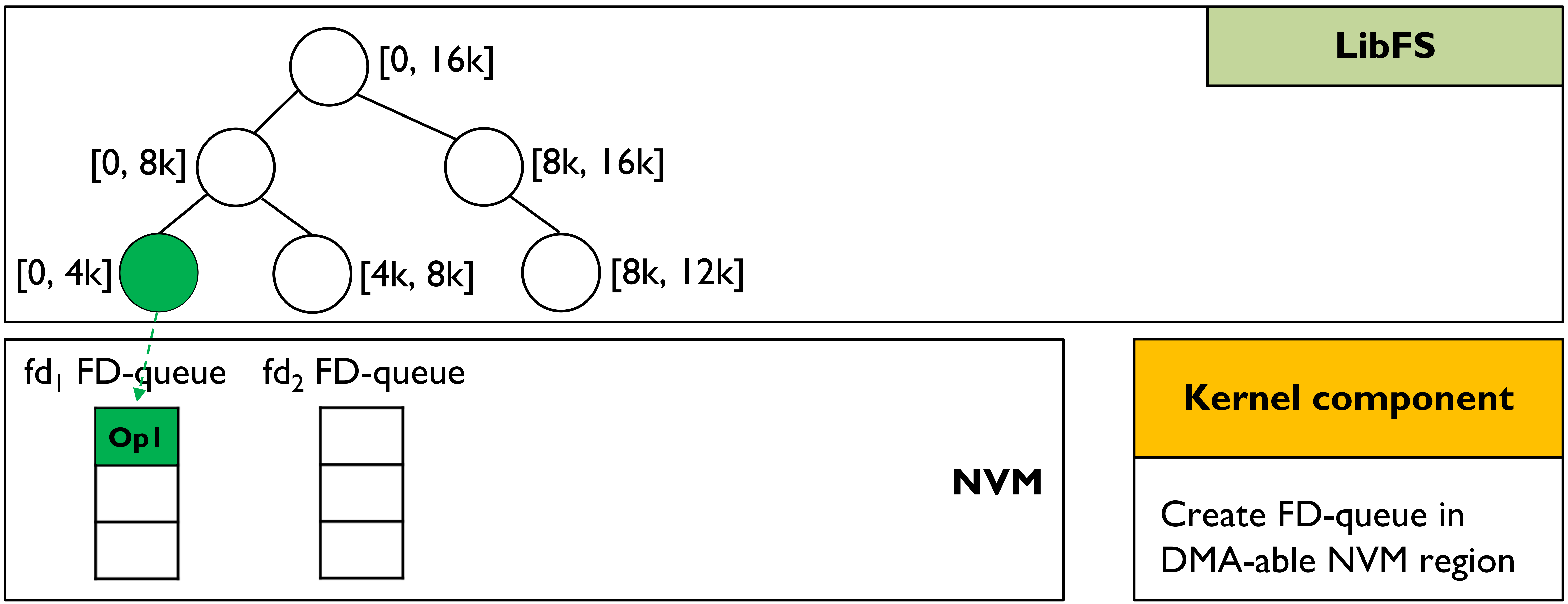
Resolving conflict writes – Interval tree structure

Thread 1

```
      fd1 = open("shared_file", rw);
Op1: pwrite(fd1, buf, sz=4096, off=0)
```

Thread 2

```
      fd2 = open("shared_file", rw);
→ Op2: pwrite(fd2, buf, sz=4096, off = 4096)
  Op3: pwrite(fd2, buf, sz = 4096, off = 0);
```



Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);

Op1: pwrite(fd1, buf, sz=4096, off=0)

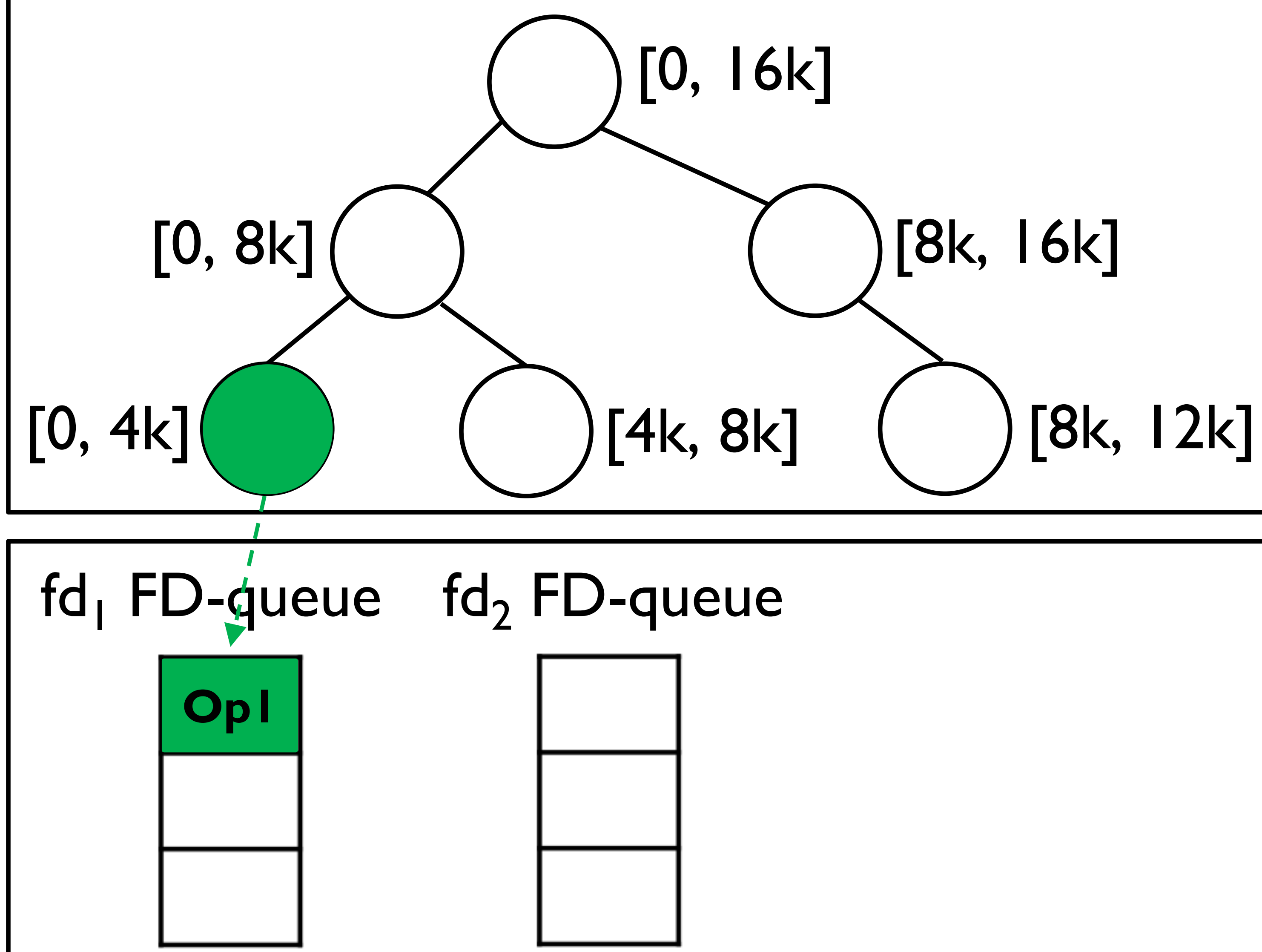
Thread 2

fd2 = open("shared_file", rw);

→ **Op2**: pwrite(fd2, buf, sz=4096, off = 4096)

Op3: pwrite(fd2, buf, sz = 4096, off = 0);

LibFS



Non-overlapping writes

NVM

Kernel component

Create FD-queue in DMA-able NVM region

Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open(“shared_file”, rw);

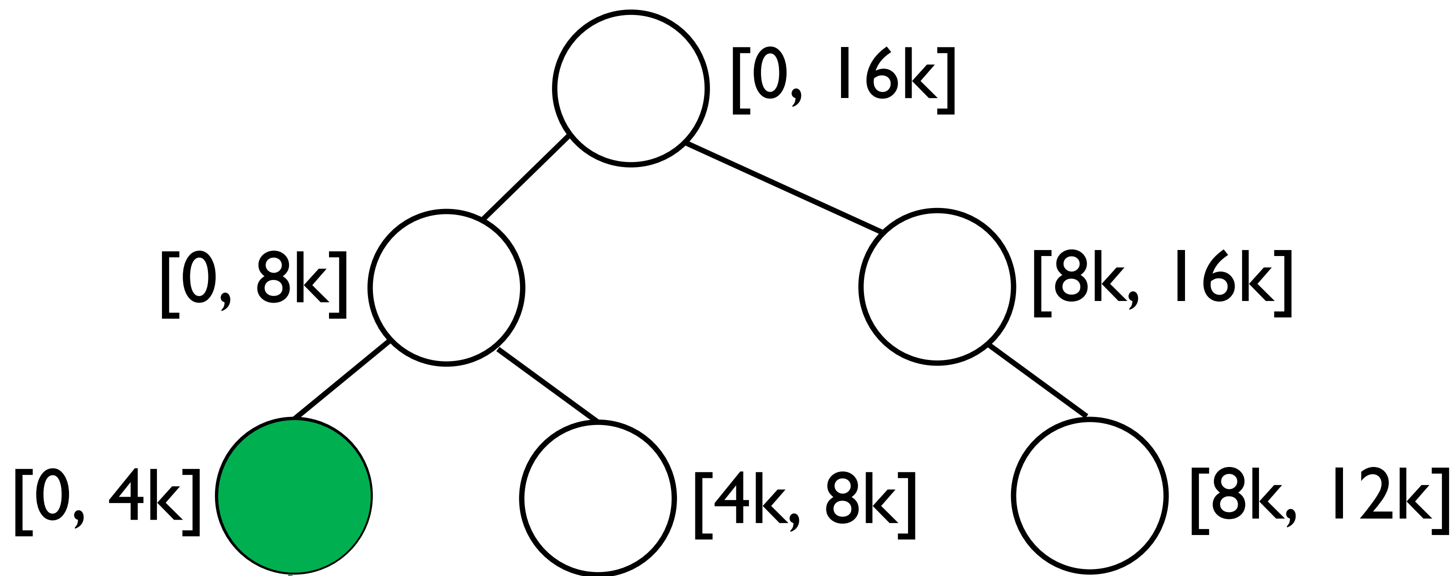
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open(“shared_file”, rw);

→ **Op2**: pwrite(fd2, buf, sz=4096, off = 4096)

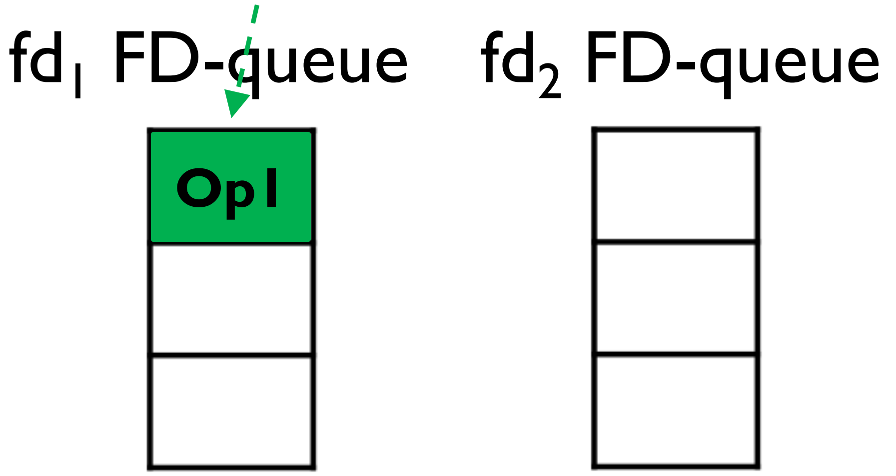
Op3: pwrite(fd2, buf, sz = 4096, off = 0);



Interval tree lock

LibFS

Non-overlapping writes



NVM

Kernel component

Create FD-queue in DMA-able NVM region

Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open(“shared_file”, rw);

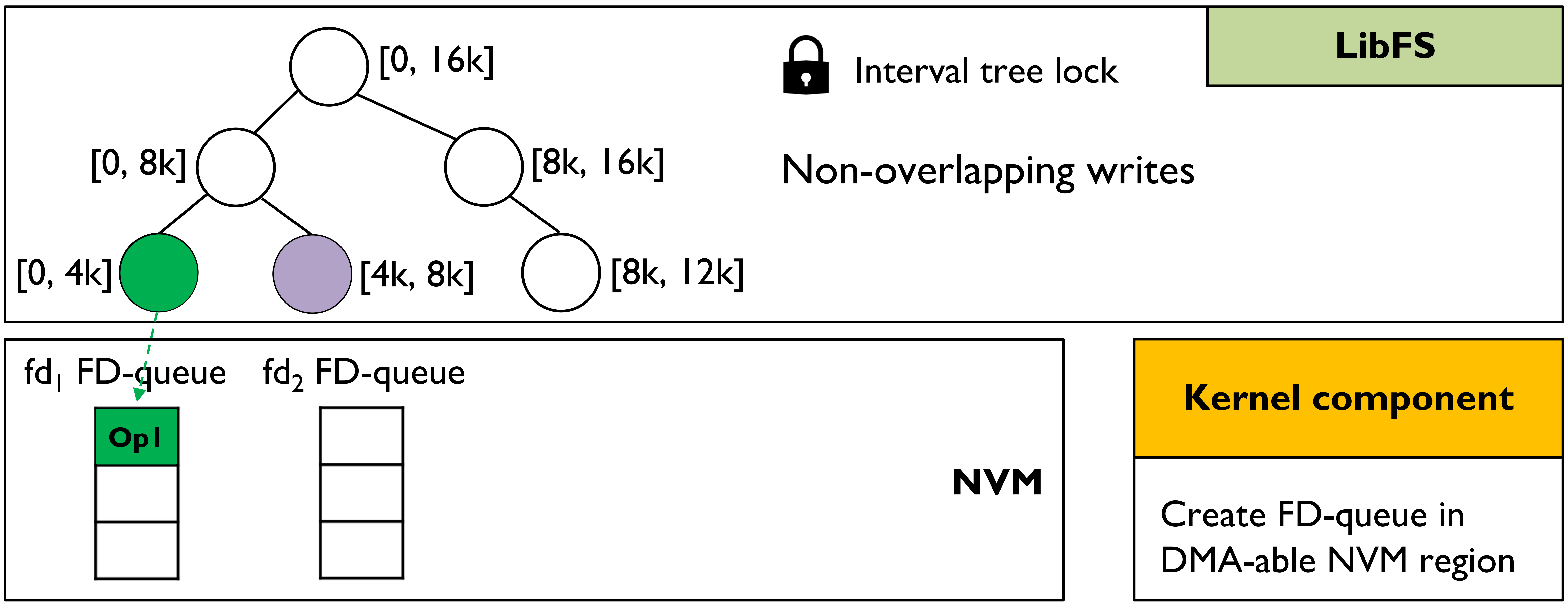
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open(“shared_file”, rw);

→ **Op2**: pwrite(fd2, buf, sz=4096, off = 4096)

Op3: pwrite(fd2, buf, sz = 4096, off = 0);



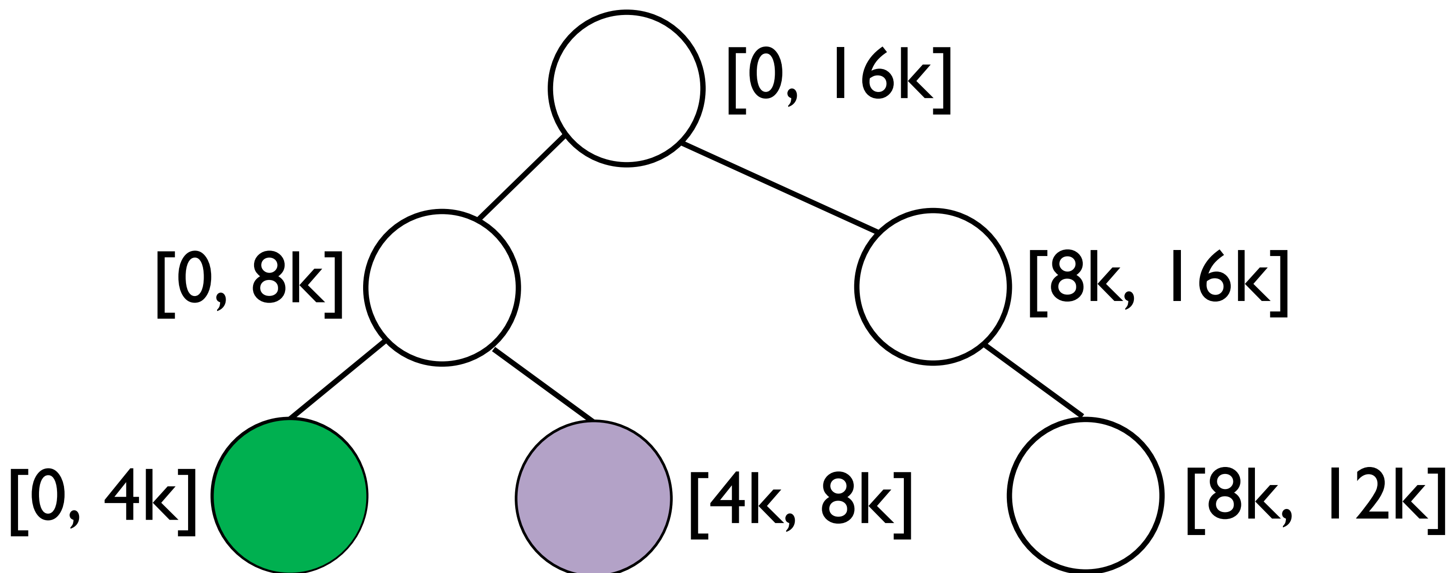
Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1
fd1 = open("shared_file", rw);
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2
fd2 = open("shared_file", rw);
→ **Op2**: pwrite(fd2, buf, sz=4096, off = 4096)
Op3: pwrite(fd2, buf, sz = 4096, off = 0);

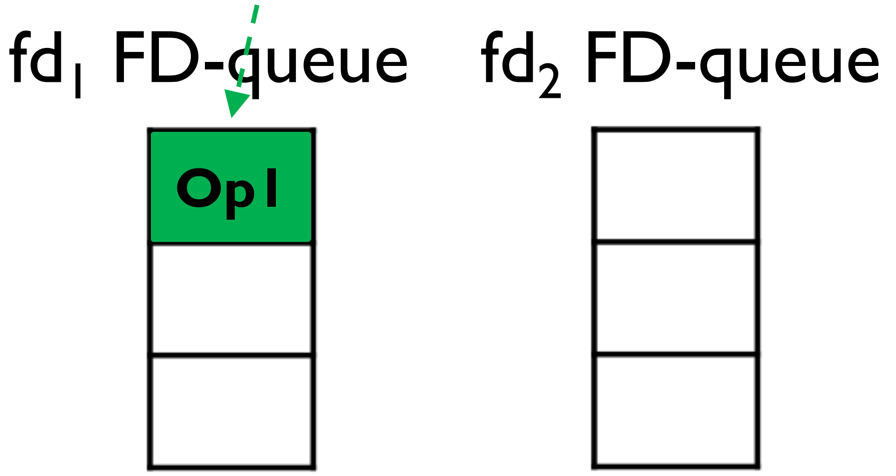
LibFS



Interval tree lock

Non-overlapping writes

Interval tree unlock



NVM

Kernel component

Create FD-queue in DMA-able NVM region

Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open(“shared_file”, rw);

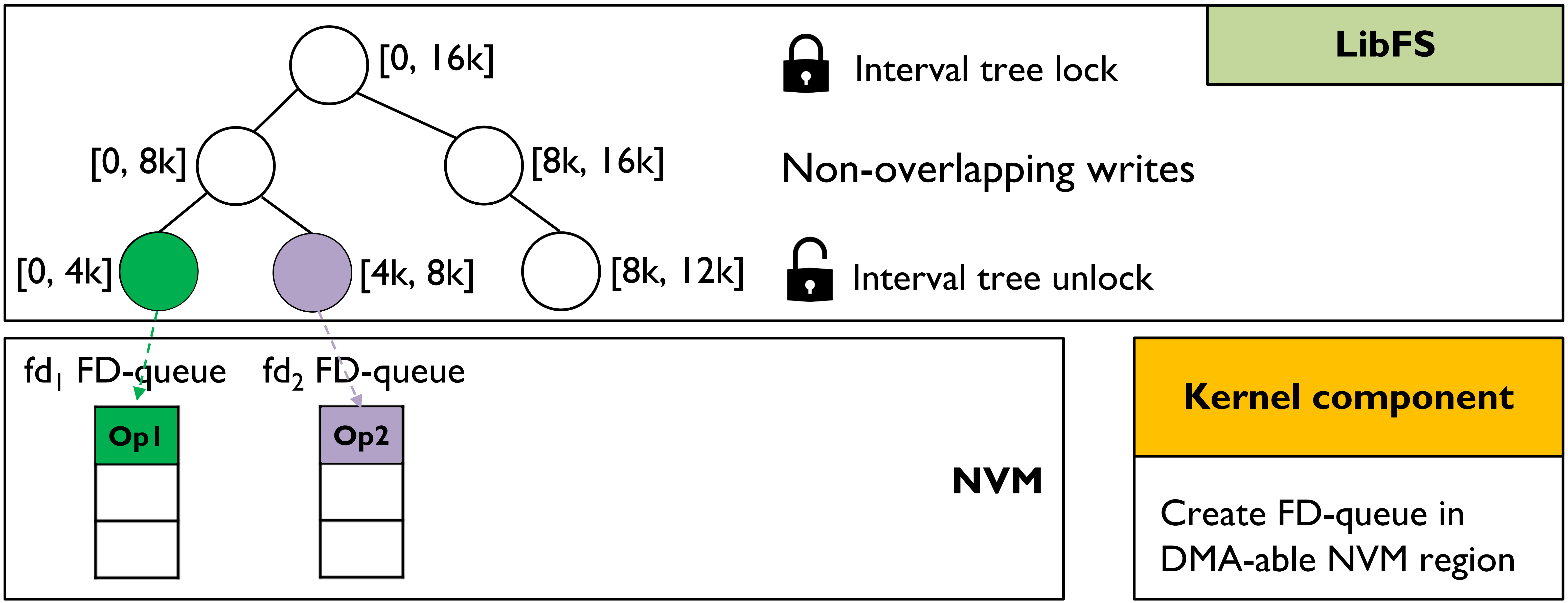
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open(“shared_file”, rw);

→ **Op2**: pwrite(fd2, buf, sz=4096, off = 4096)

Op3: pwrite(fd2, buf, sz = 4096, off = 0);



Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);

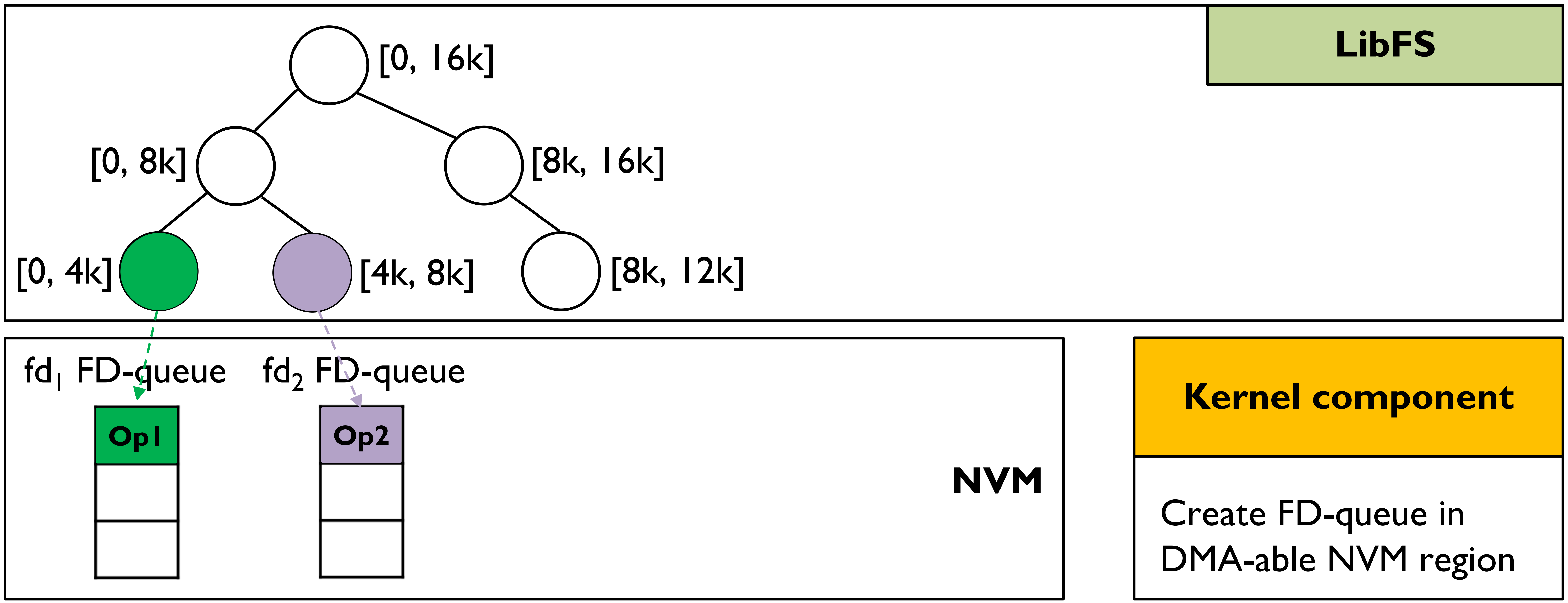
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open("shared_file", rw);

→ **Op2**: pwrite(fd2, buf, sz=4096, off = 4096)

Op3: pwrite(fd2, buf, sz = 4096, off = 0);

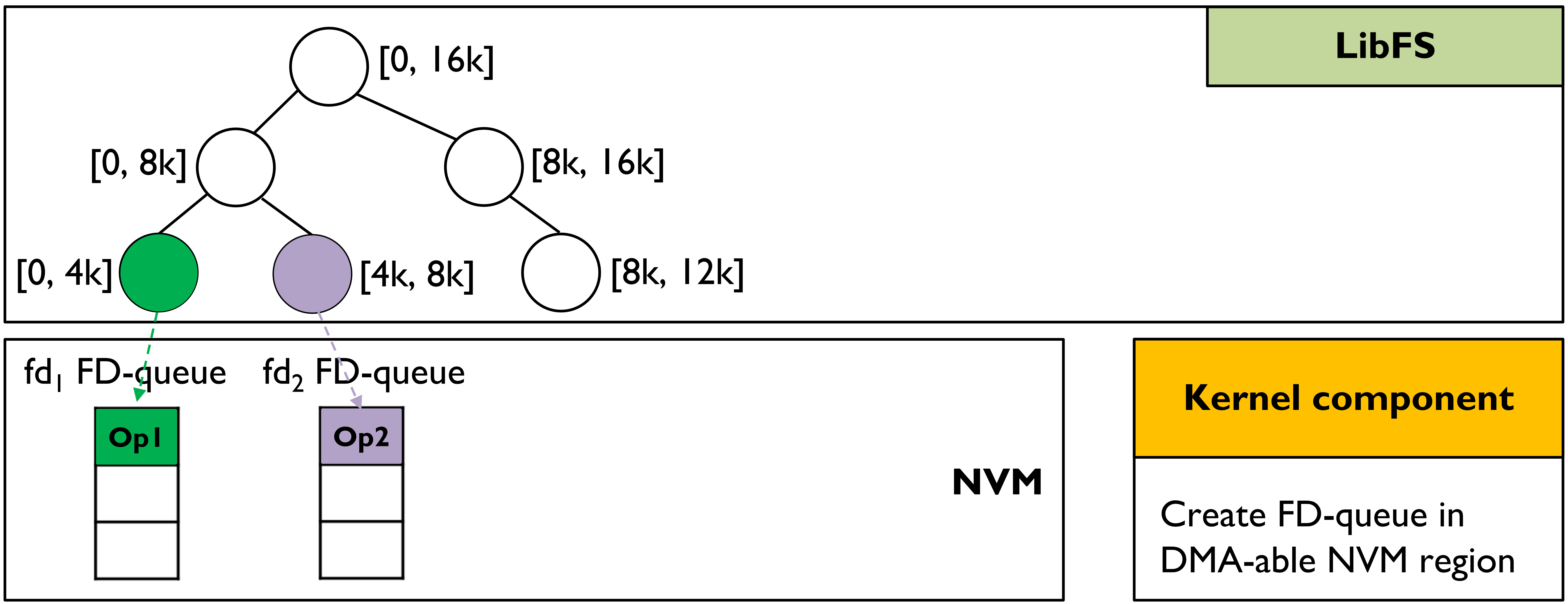


Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1
fd1 = open("shared_file", rw);
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2
fd2 = open("shared_file", rw);
Op2: pwrite(fd2, buf, sz=4096, off = 4096)
Op3: pwrite(fd2, buf, sz = 4096, off = 0);

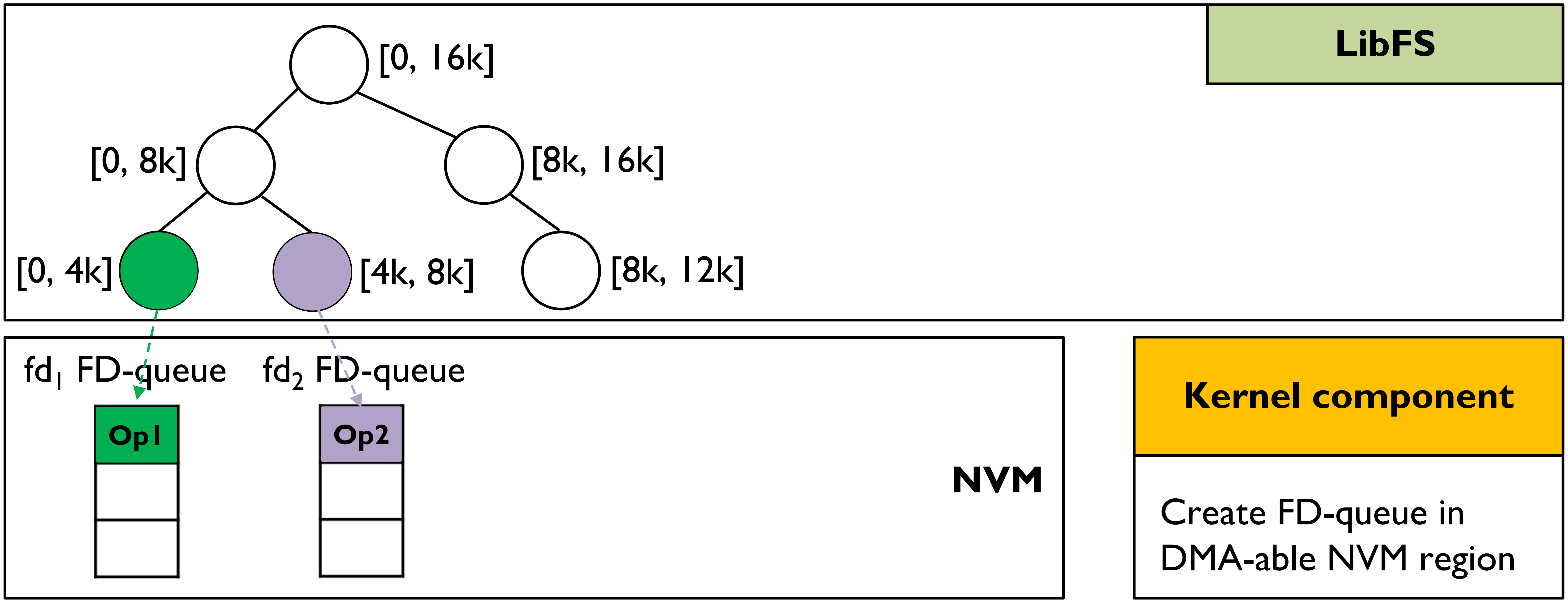


Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1
fd1 = open("shared_file", rw);
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2
fd2 = open("shared_file", rw);
Op2: pwrite(fd2, buf, sz=4096, off = 4096)
→ **Op3**: pwrite(fd2, buf, sz = 4096, off = 0);



Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open(“shared_file”, rw);

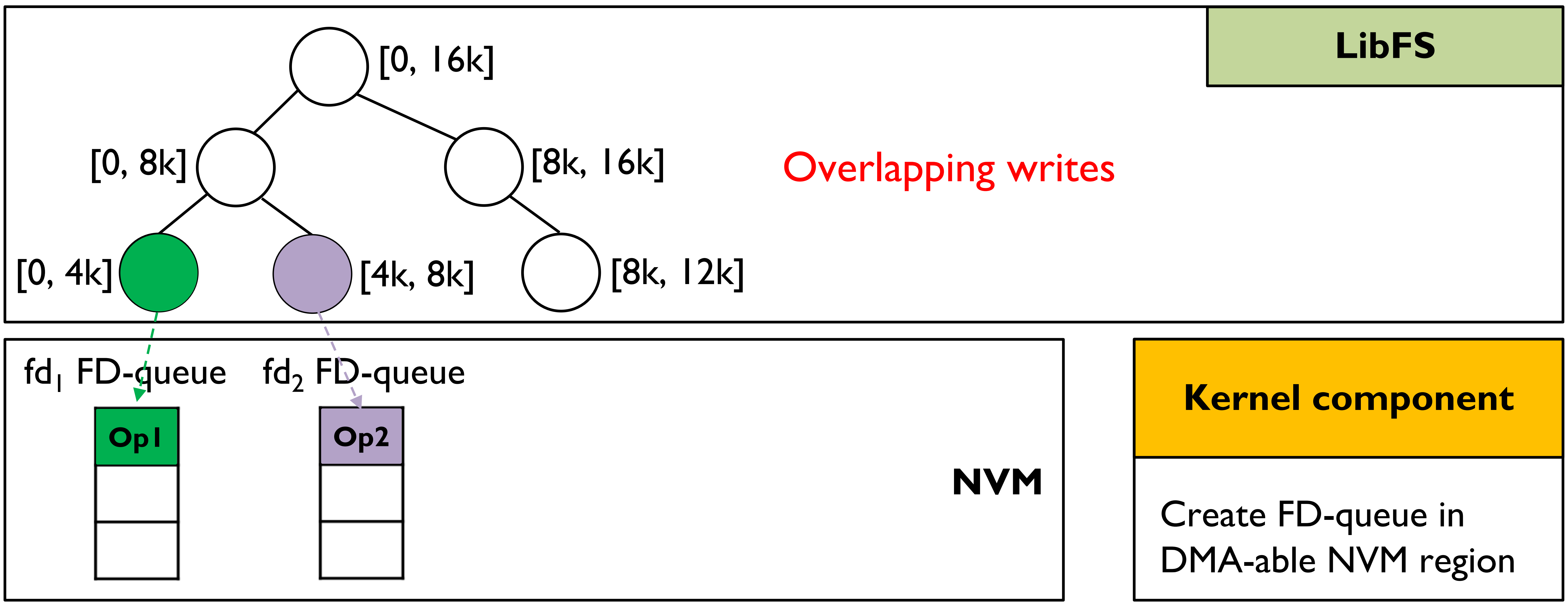
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open(“shared_file”, rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

→ **Op3**: pwrite(fd2, buf, sz = 4096, off = 0);



Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);

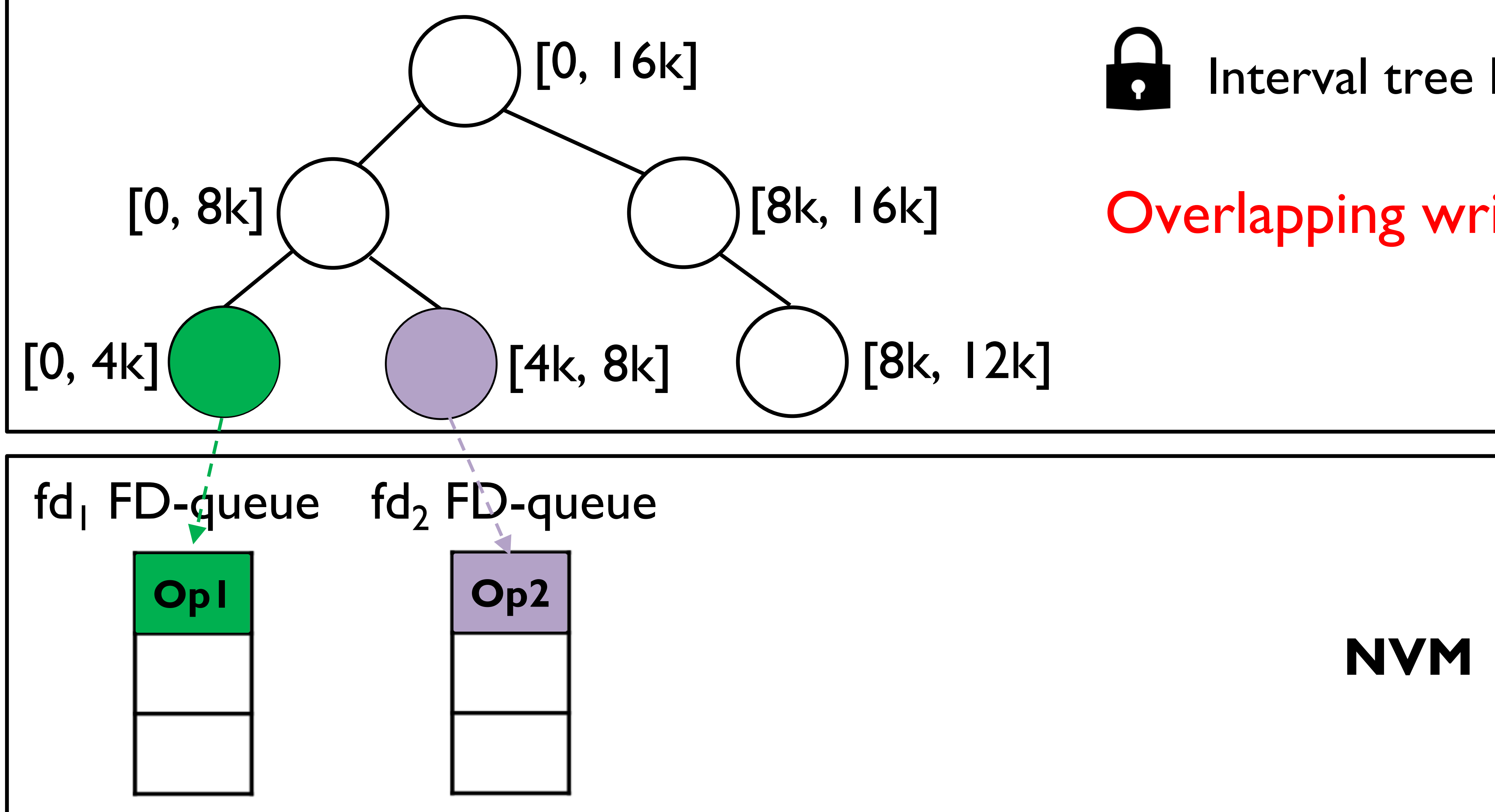
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open("shared_file", rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

→ **Op3**: pwrite(fd2, buf, sz = 4096, off = 0);



Interval tree lock

Overlapping writes

LibFS

Kernel component

Create FD-queue in DMA-able NVM region

Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);

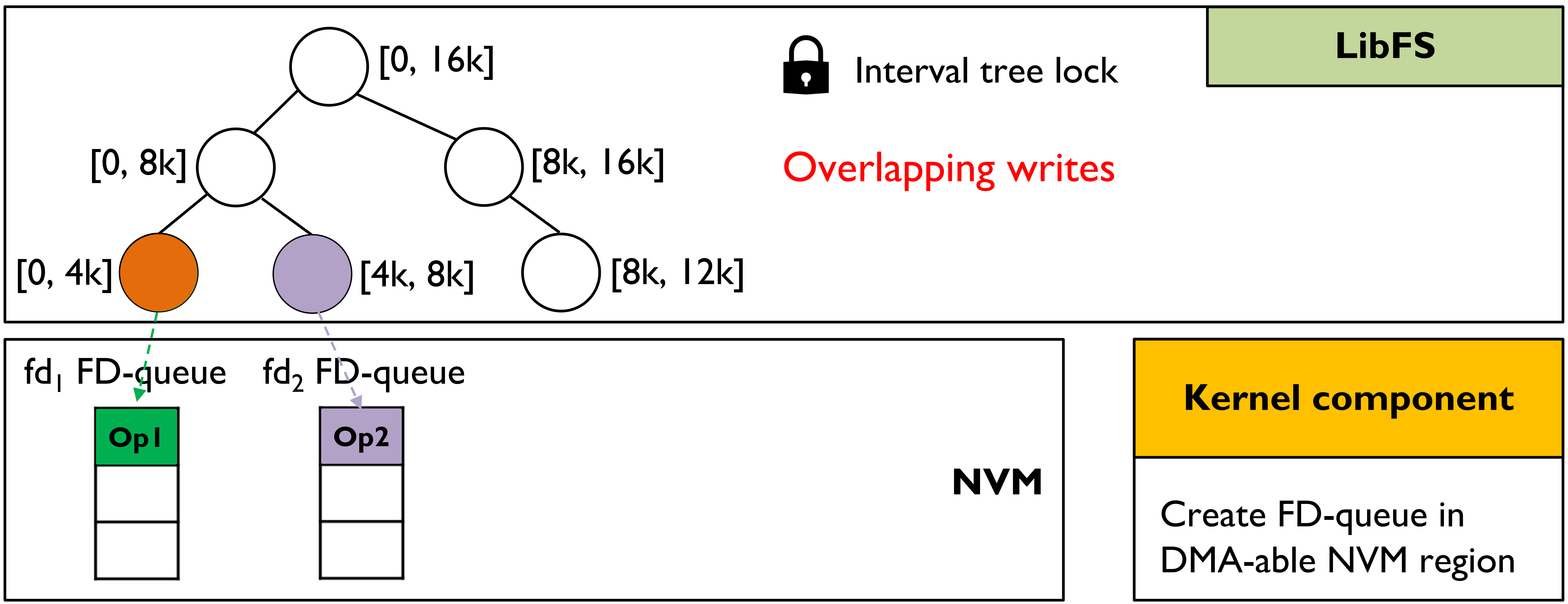
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open("shared_file", rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

→ **Op3**: pwrite(fd2, buf, sz = 4096, off = 0);



Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);

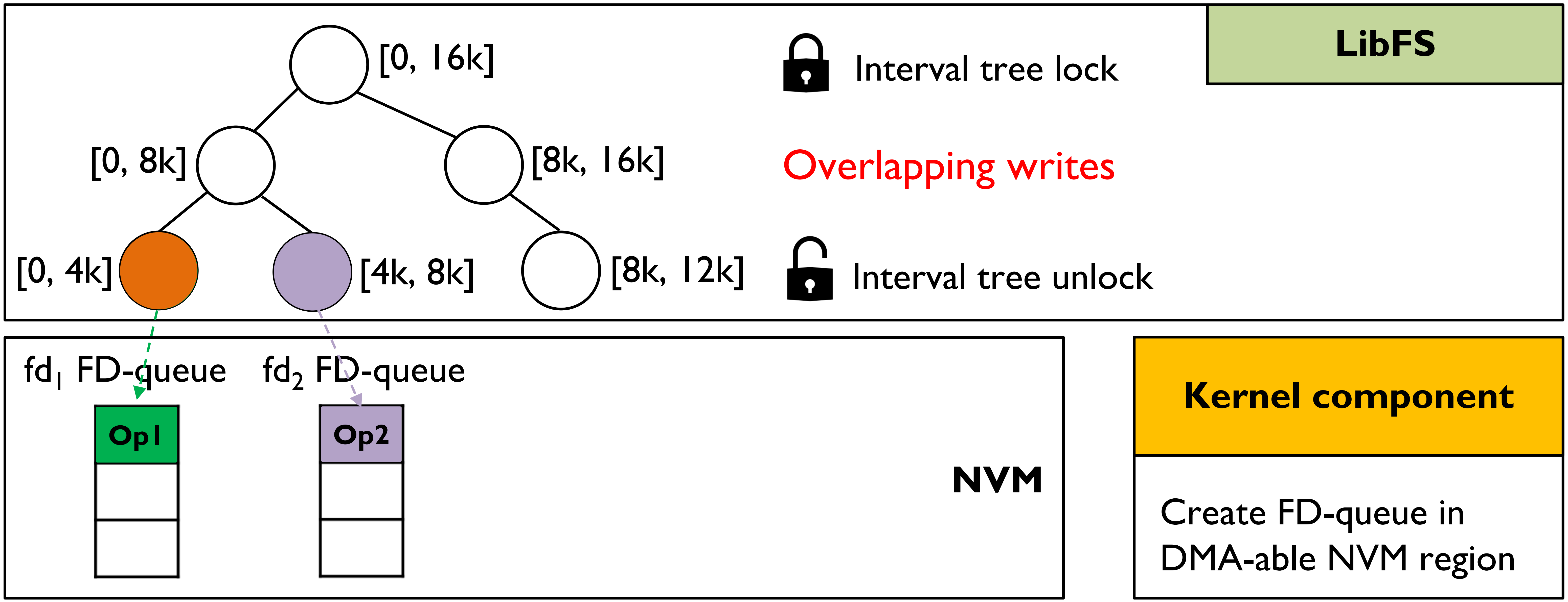
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open("shared_file", rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

→ **Op3**: pwrite(fd2, buf, sz = 4096, off = 0);



Kernel component

Create FD-queue in DMA-able NVM region

Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);

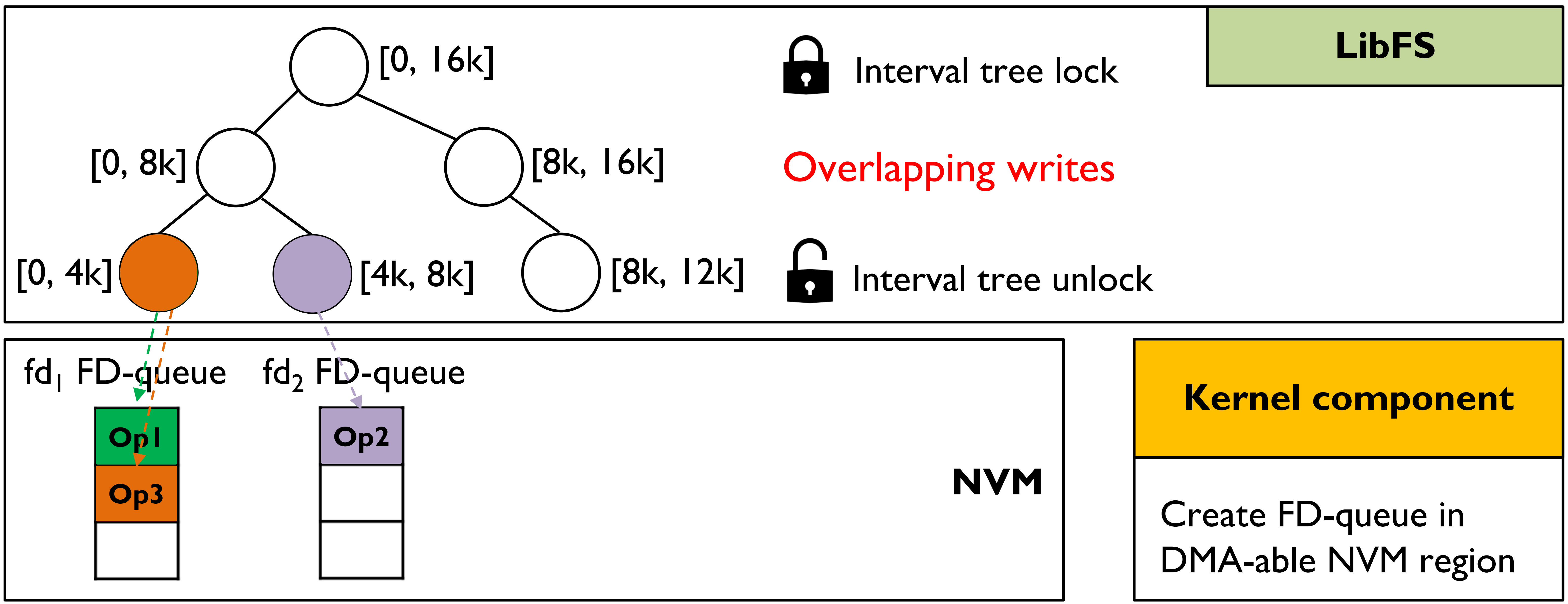
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open("shared_file", rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

→ **Op3**: pwrite(fd2, buf, sz = 4096, off = 0);



Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open(“shared_file”, rw);

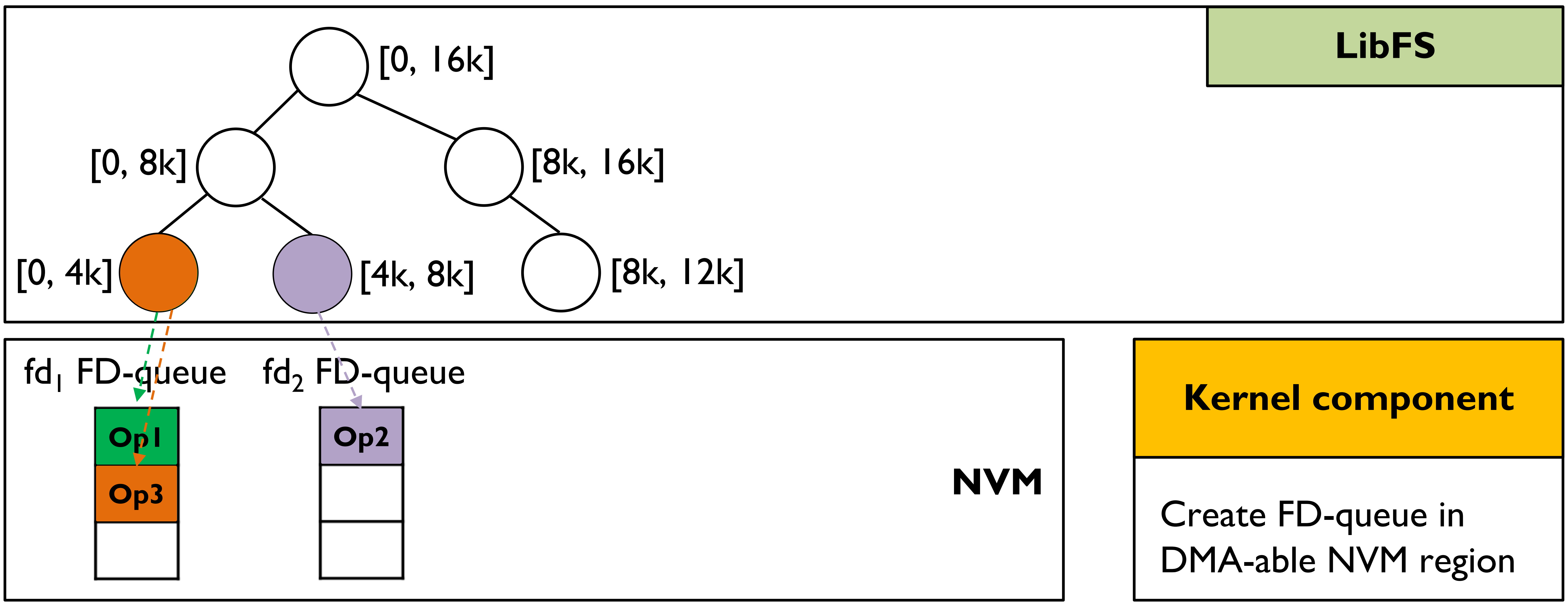
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open(“shared_file”, rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

→ **Op3**: pwrite(fd2, buf, sz = 4096, off = 0);



Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open(“shared_file”, rw);

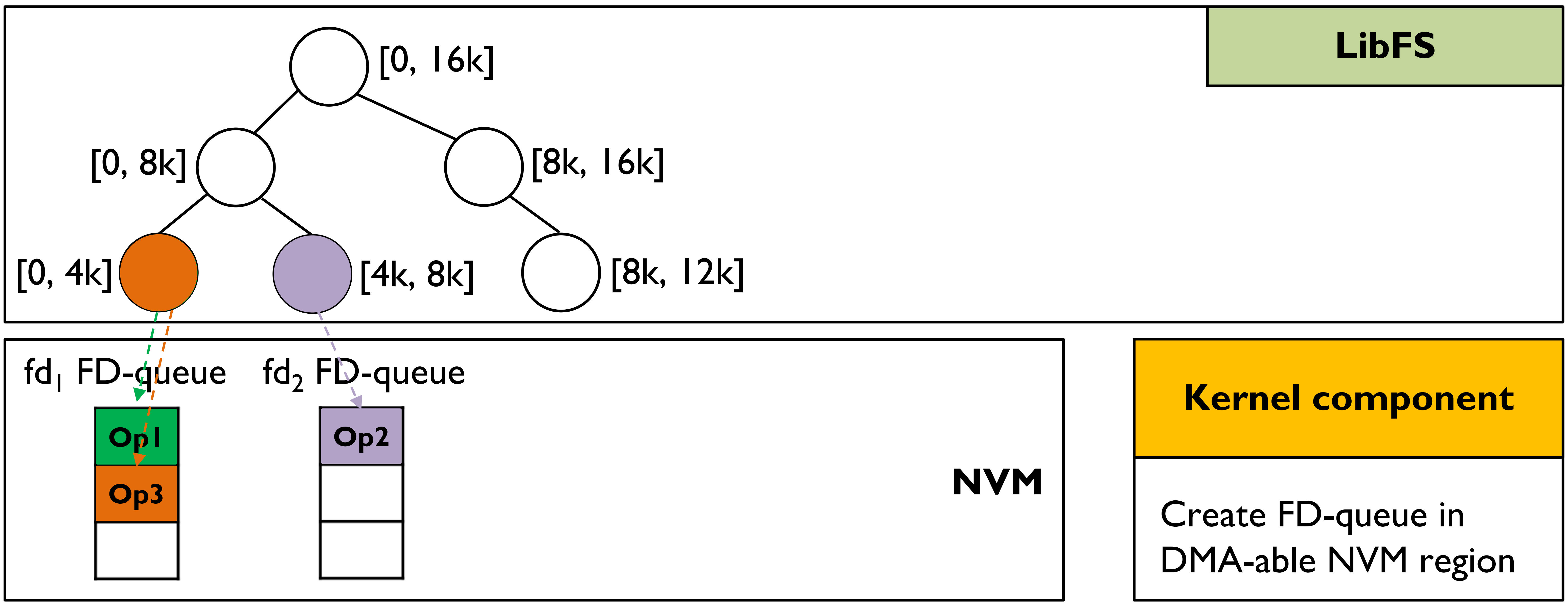
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open(“shared_file”, rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

Op3: pwrite(fd2, buf, sz = 4096, off = 0);



Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open(“shared_file”, rw);

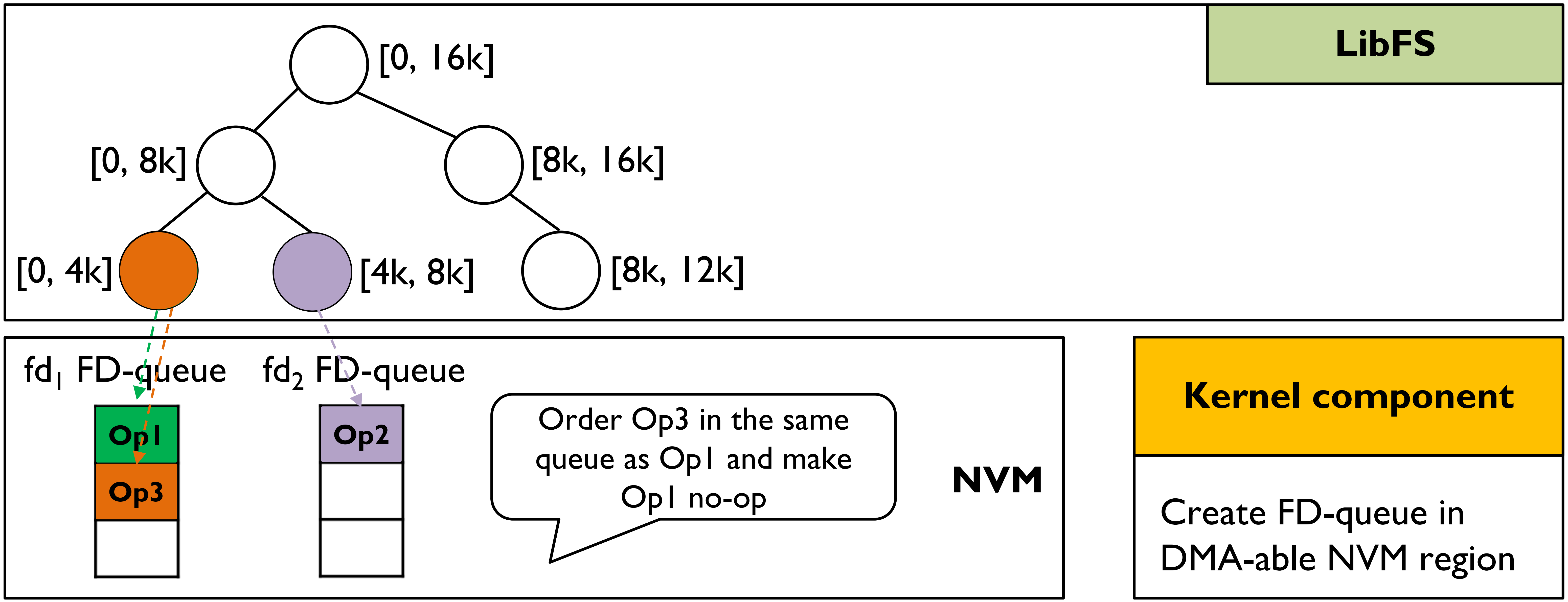
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open(“shared_file”, rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

Op3: pwrite(fd2, buf, sz = 4096, off = 0);



Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);

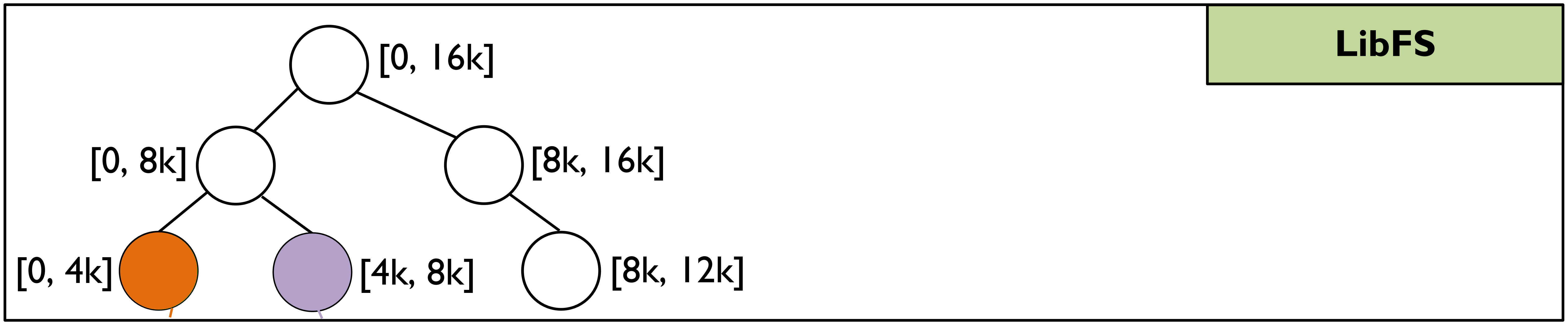
Op1: pwrite(fd1, buf, sz=4096, off=0)

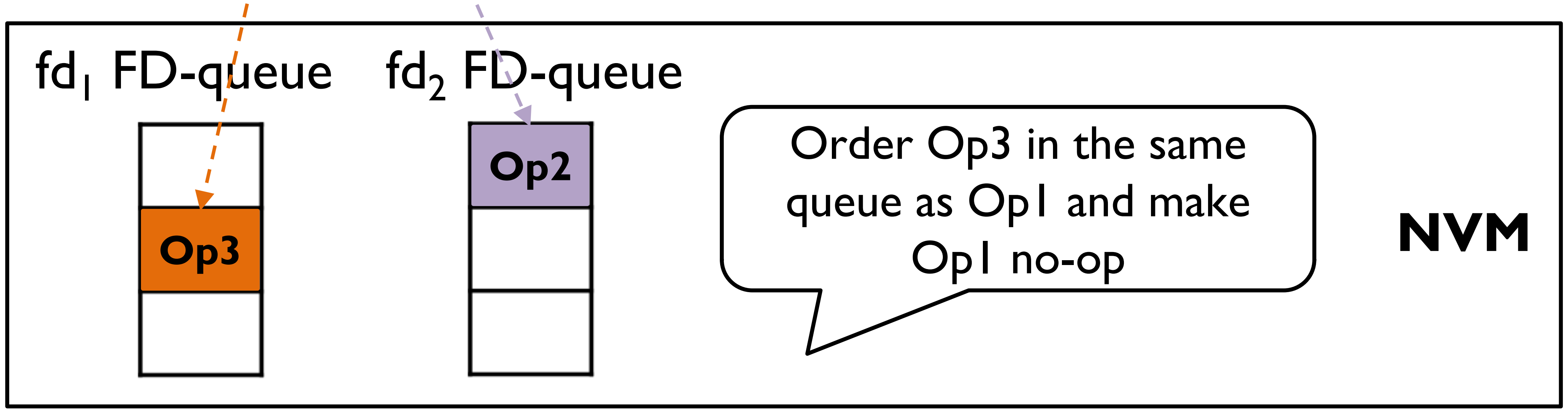
Thread 2

fd2 = open("shared_file", rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

Op3: pwrite(fd2, buf, sz = 4096, off = 0);





Kernel component

Create FD-queue in DMA-able NVM region

Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1

fd1 = open("shared_file", rw);

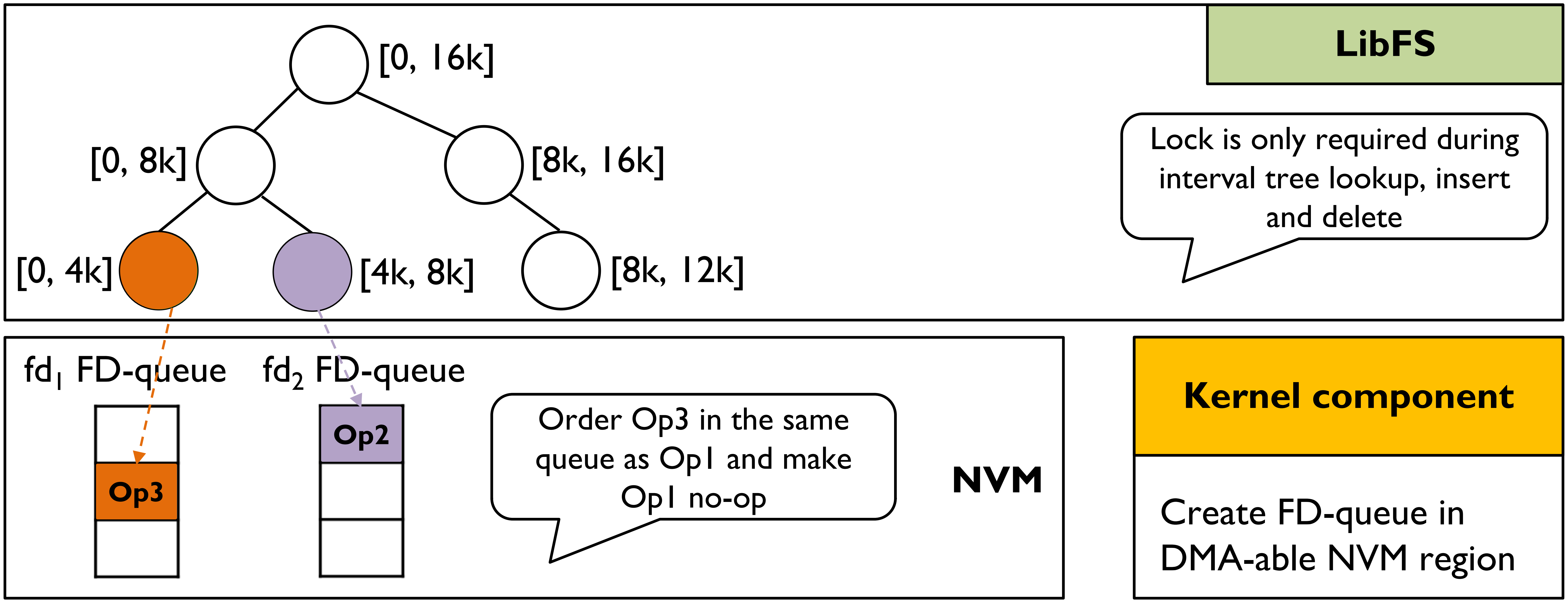
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2

fd2 = open("shared_file", rw);

Op2: pwrite(fd2, buf, sz=4096, off = 4096)

Op3: pwrite(fd2, buf, sz = 4096, off = 0);

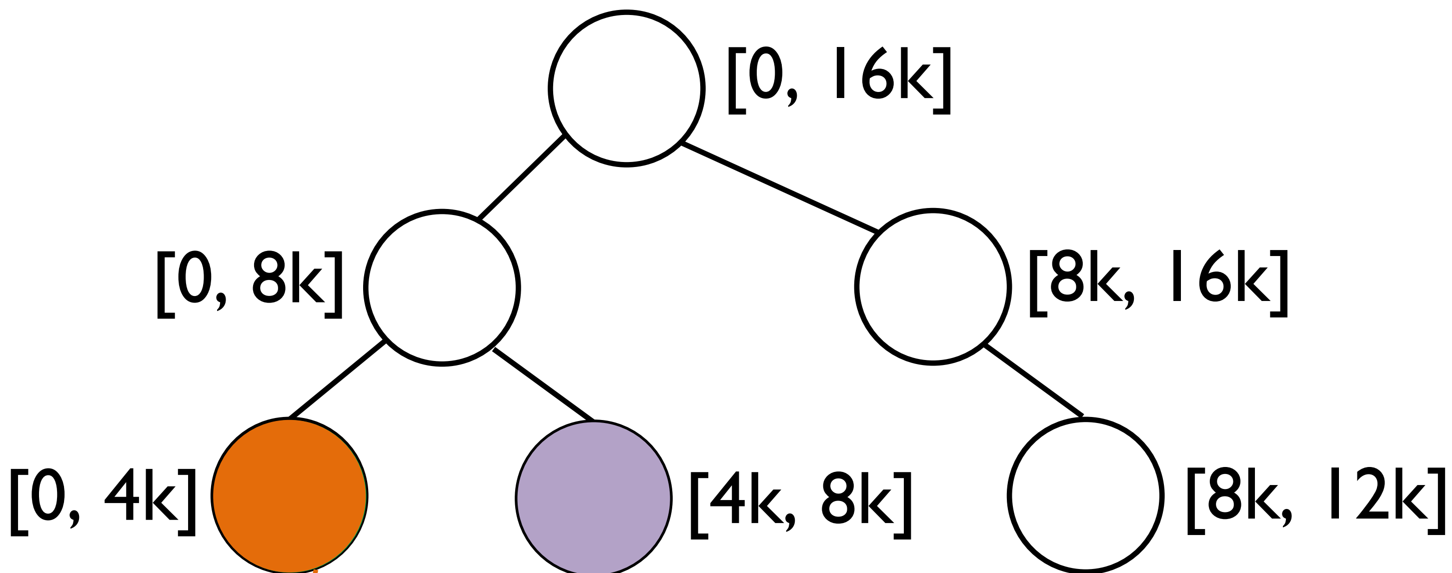


Fine-grained Concurrency Control

Resolving conflict writes – Interval tree structure

Thread 1
fd1 = open(“shared_file”, rw);
Op1: pwrite(fd1, buf, sz=4096, off=0)

Thread 2
fd2 = open(“shared_file”, rw);
Op2: pwrite(fd2, buf, sz=4096, off = 4096)
Op3: pwrite(fd2, buf, sz = 4096, off = 0);



LibFS

Lock is only required during interval tree lookup, insert and delete

CrossFS converts file concurrency control to queue ordering problem
Once requests are ordered, FirmFS dispatches requests from queues in parallel

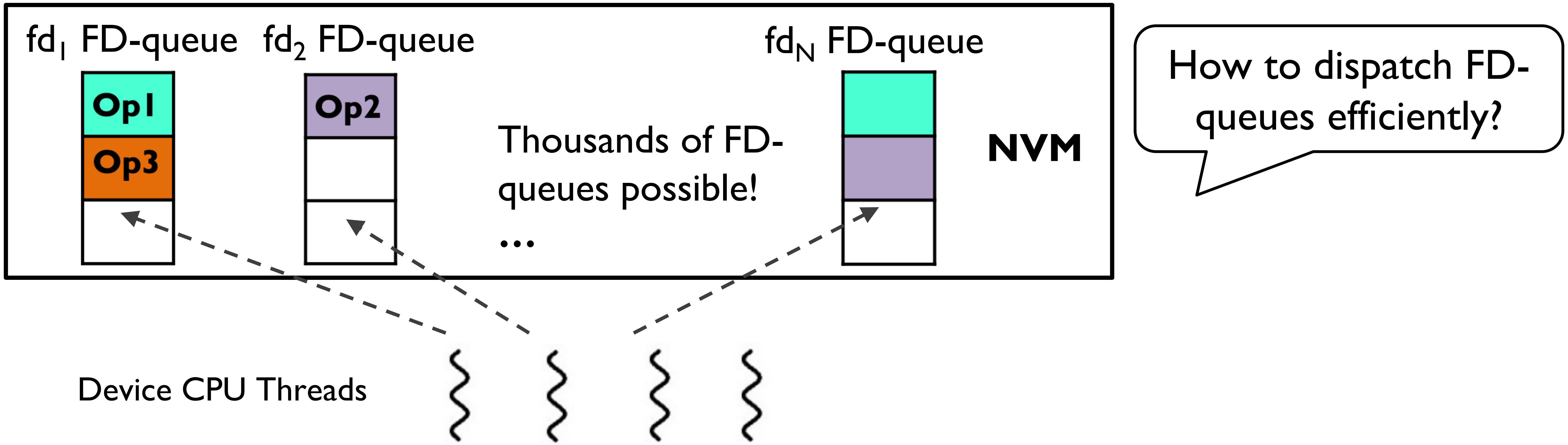
Op3

Op1 no-op

Create FD-queue in DMA-able NVM region

Unified I/O Scheduler Framework

- Need to dispatch and schedule FD-queues efficiently
  - Thousands of FD-queues for large scale applications
  - Few in-storage CPUs (four in our study)



- Insight: Unified file system + firmware I/O scheduler
  - Map FD-queues to FirmFS processing threads (i.e., device-level CPUs)
  - Separate scheduling mechanism from scheduling policy

I/O Scheduling Policies

- Round Robin
  - Each device CPU dispatches request from FD-queues
  - Provides fairness but delays blocking operations (e.g., read, fsync)
- Urgent Aware Scheduling
  - Prioritize blocking requests
  - Avoid write request starvation by limiting write delays
- More sophisticated policies – future work!

Cross-Layered Crash Consistency

- FD-queue and data buffer crash consistency
  - NVM provides persistence
  - CLWB and memory fence to provide crash consistency

- FirmFS crash consistency
  - Default meta-data journaling like current file systems
  - Add offset of data buffer in NVM to the journal entry
  - Get data journaling benefits at the cost of meta-data journaling

Please see our paper for more details!

Outline

- Background
- Motivation
- Design
- Evaluation
- Conclusion

Experimental Setup

- Hardware platform
  - Dual-socket 64-core Xeon Scalable CPU @ 2.6GHz
  - 512GB Intel Optane DC NVM

- Emulate firmware-level FS (no programmable storage H/W)
  - Reserve dedicated device threads for handling I/O requests
  - Add PCIe latency for all I/O operations
  - Reduce CPU frequency for device CPUs

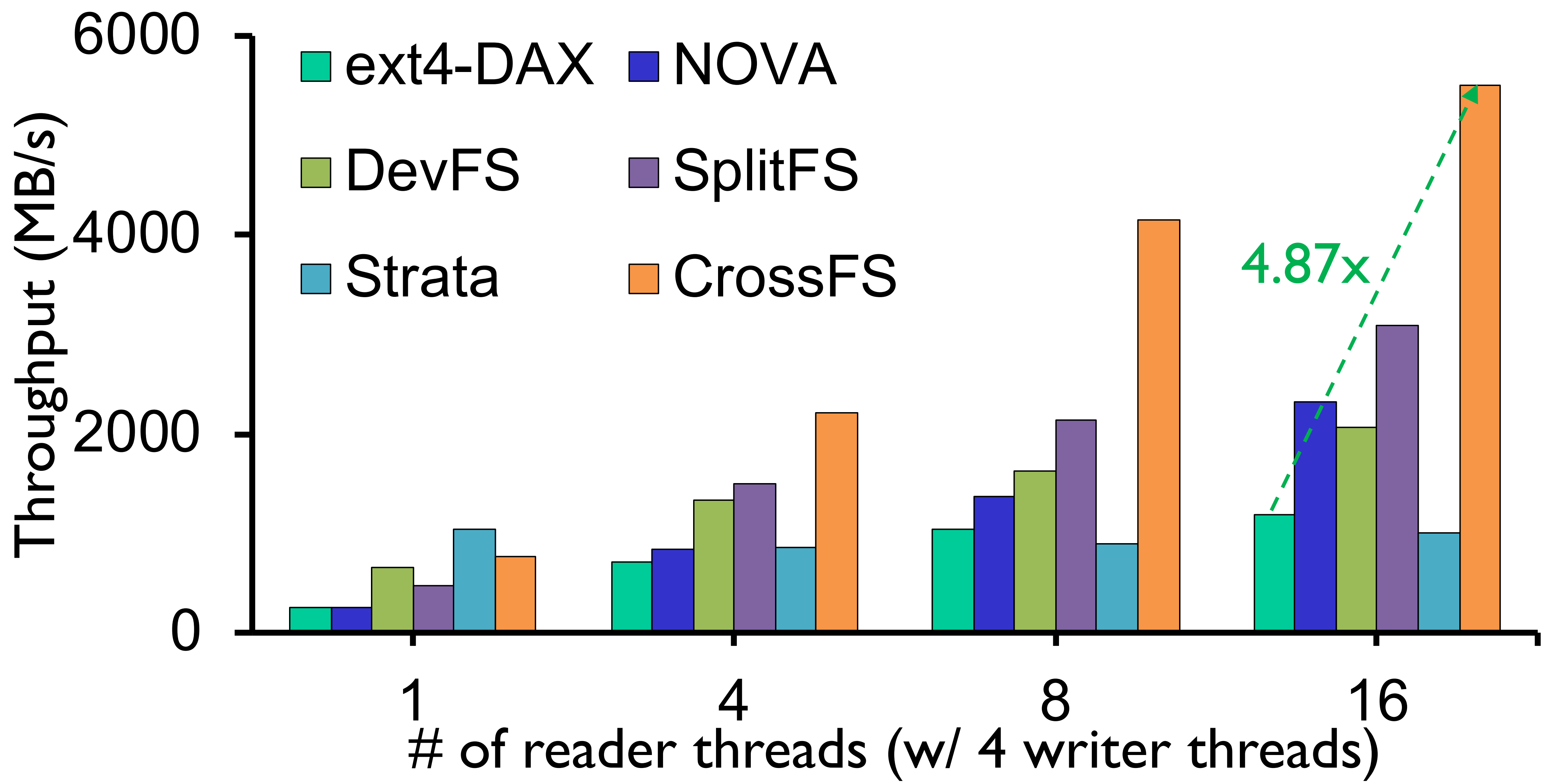
- State-of-the-art file systems
  - **ext4-DAX**, **NOVA** [FAST' 16] (Kernel-level file system)
  - **Strata** [SOSP '17], **SplitFS** [SOSP' 19] (User-level file system)
  - **DevFS** [FAST' 18] (Firmware-level file system)

Evaluation Goals

- **Concurrent accesses scaling when sharing files?**
- Reducing I/O software cost?
- I/O scaling without file sharing across threads?
- Real-world application goals?

Microbenchmark – Read Scalability

Multiple readers and 4 writer threads accessing a 12GB shared file

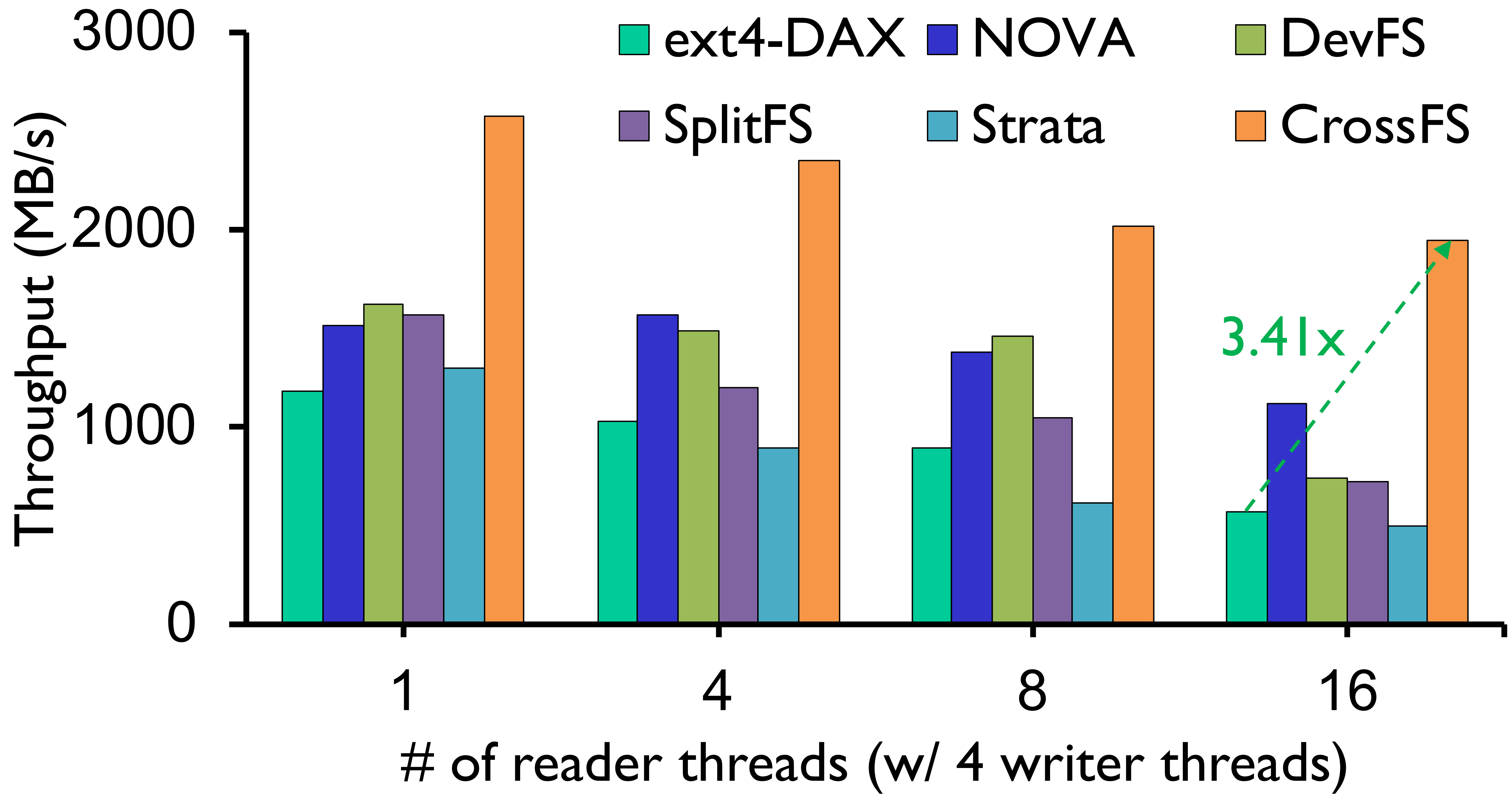


- X-axis: # of concurrent readers
- Y-axis: Aggregated readers' throughput

Readers do not have to wait for writers

Microbenchmark – Write Scalability

Multiple readers and 4 writer threads accessing a 12GB shared file



- X-axis: # of concurrent readers
- Y-axis: Aggregated writers' throughput

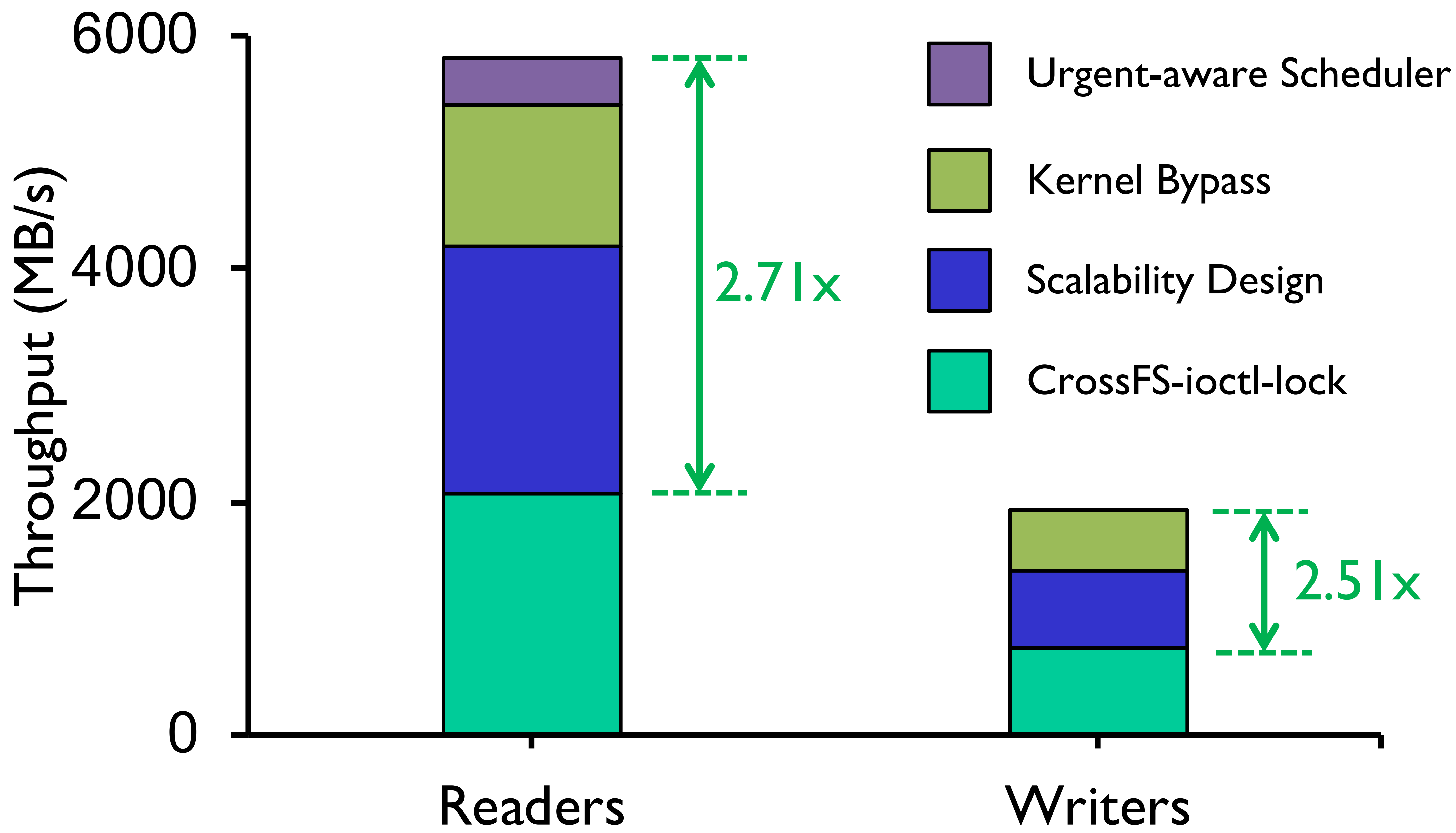
Non-overlapping writes dispatched in parallel

Evaluation Goals

- Concurrent accesses scaling when sharing files?
- **Reducing I/O software cost?**
- I/O scaling without file sharing across threads?
- Real-world application goals?

CrossFS Performance Breakdown

Multi-reader and multi-writer threads accessing a 12GB shared file



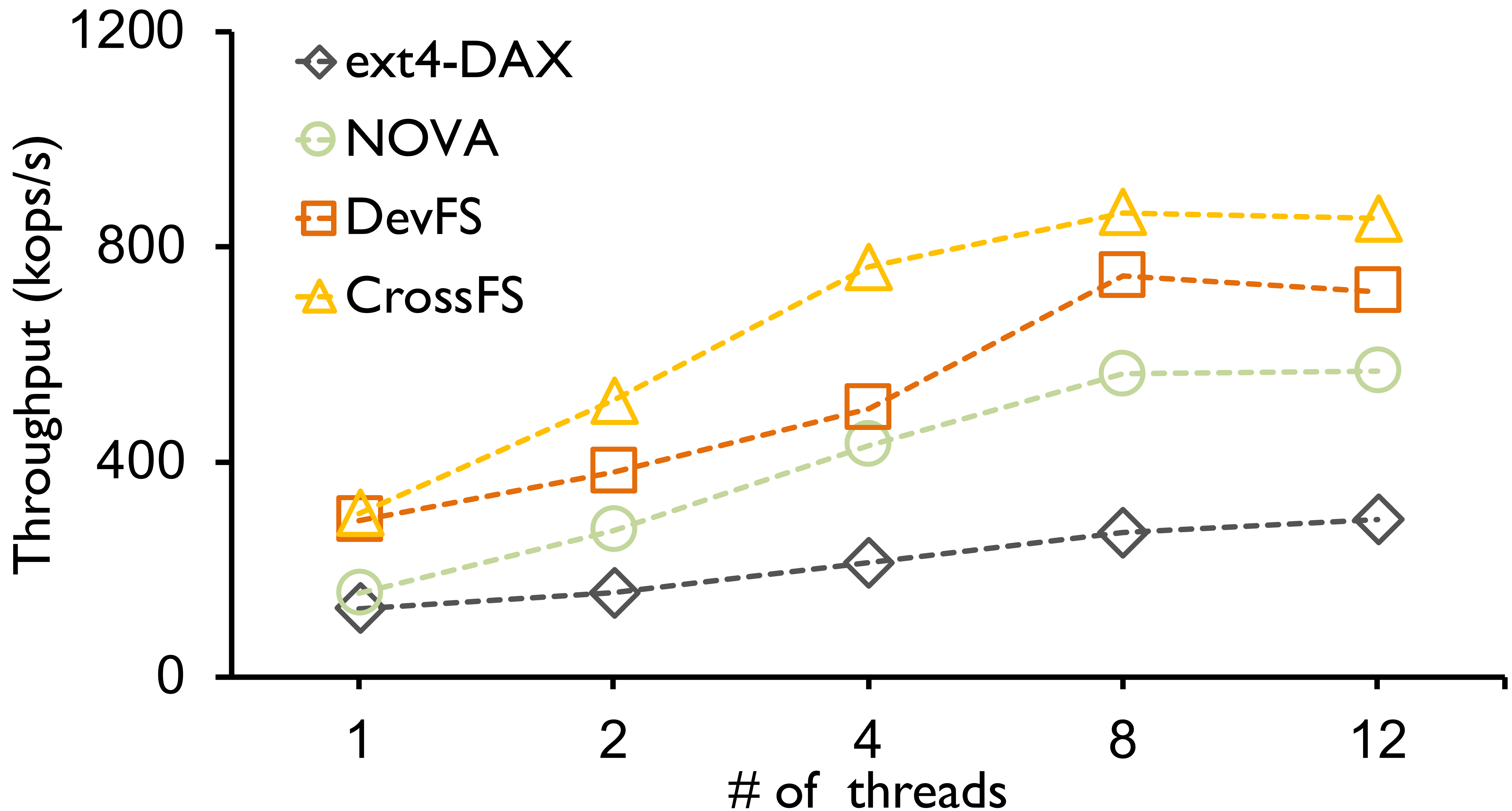
- X-axis: 16 concurrent reader threads, 4 concurrent writer threads
- Y-axis: Aggregated writers' throughput

Evaluation Goals

- Concurrent accesses scaling when sharing files?
- Reducing I/O software cost?
- **I/O scaling without file sharing across threads?**
- Real-world application goals?

Macro-benchmark: Filebench

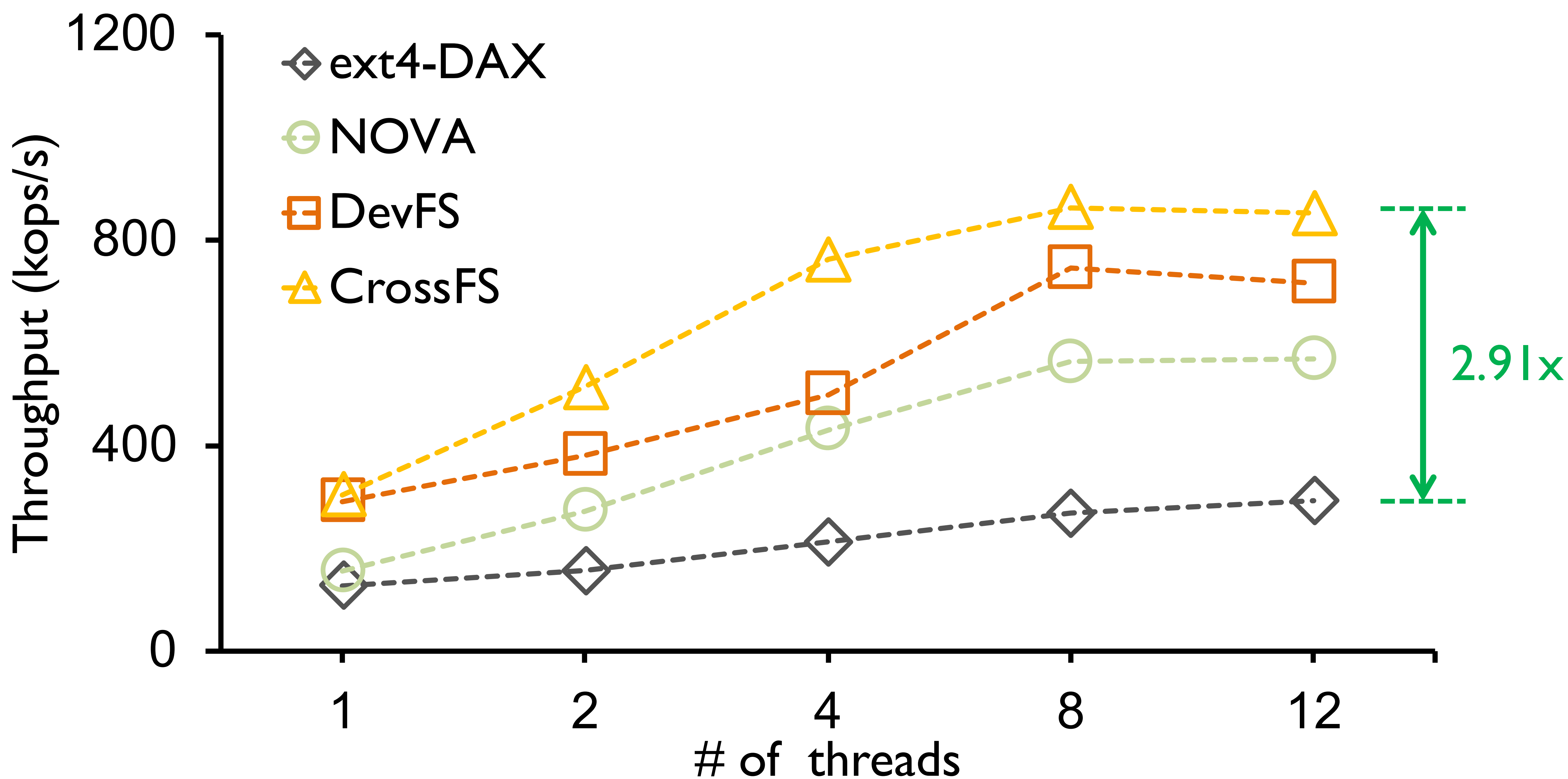
Fileserver (write-heavy workload)



- X-axis: # of filebench threads
- Y-axis: benchmark throughput

Macro-benchmark: Filebench

Fileserver (write-heavy workload)



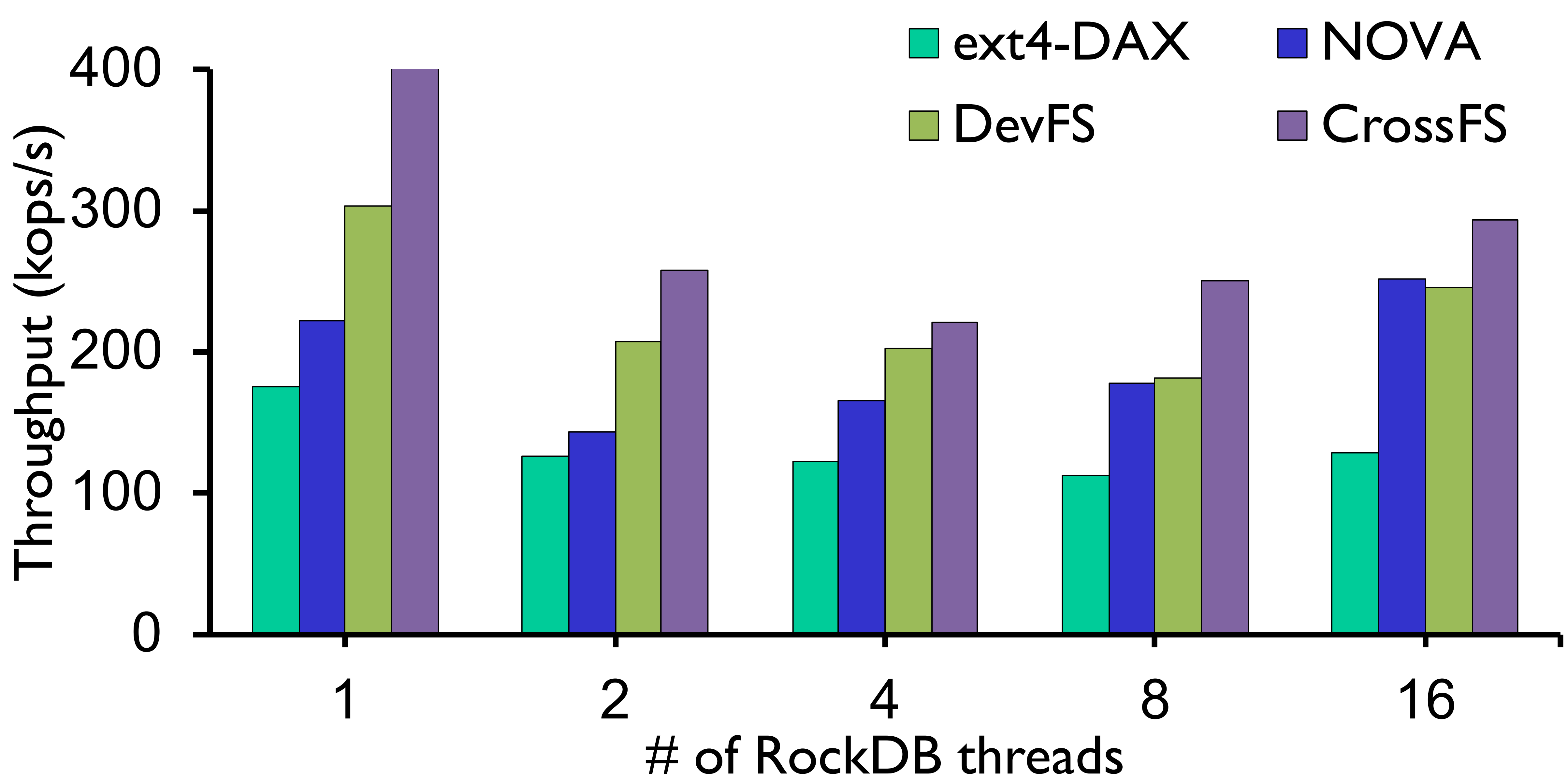
CrossFS writes to NVM buffers first and then asynchronously dispatches request, hence achieves high throughput

Evaluation Goals

- Concurrent accesses scaling when sharing files?
- Reducing I/O software cost?
- I/O scaling without file sharing across threads?
- **Real-world application goals?**

Application - RocksDB

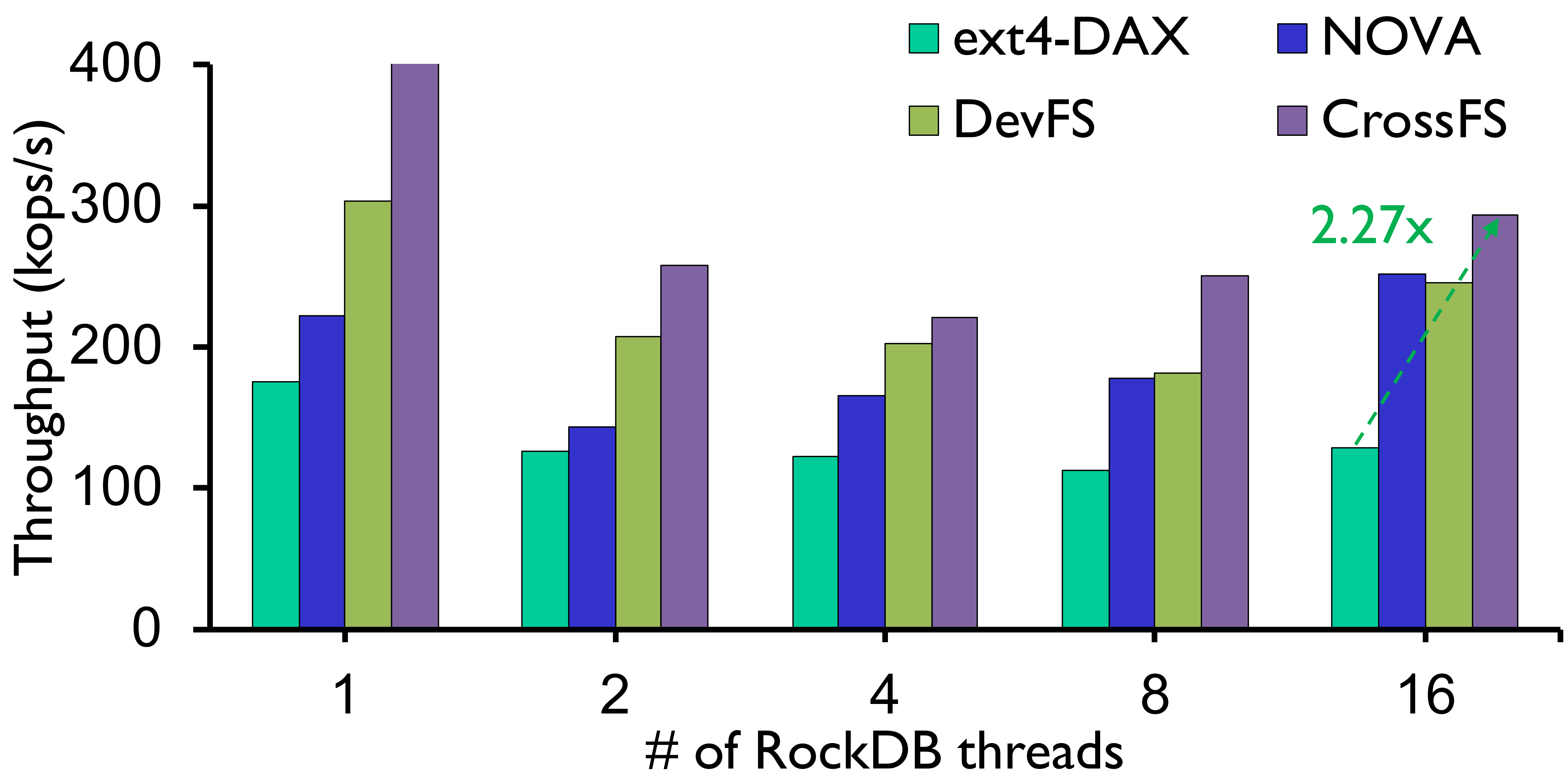
DBbench ***fillrandom*** (random write) benchmark



- X-axis: # of DBbench threads
- Y-axis: ***fillrandom*** benchmark throughput

Application - RocksDB

DBbench ***fillrandom*** (random write) benchmark



- RocksDB threads append kv-pairs to shared log files.
- CrossFS eliminates inode-level lock overheads

Summary

- Motivation
  - Software overhead matters and providing direct-access is critical
  - Poor coarse-grained concurrency in current file systems

- Solution – Cross-layered file system
  - Disaggregation of file system components across S/W and firmware
  - File descriptor-level parallelism replacing inode-level locking bottleneck
  - File descriptor scheduling and cross-layered crash consistency

- Evaluation
  - CrossFS shows up to 4x micro-benchmark performance gains
  - CrossFS shows up to 2x application performance gains

Conclusion

- Storage hardware (with compute capability) has reached the microsecond era
- Providing direct I/O and utilizing host and storage-level compute is critical
  - Our approach: Cross-layered storage file system design
- Fine grained concurrency control is important for I/O scalability
  - Our approach: File-descriptor concurrency control
- Future work:
  - H/W integration, support for sophisticated scheduling policies, other file system operations (e.g., mmap())

Thanks!

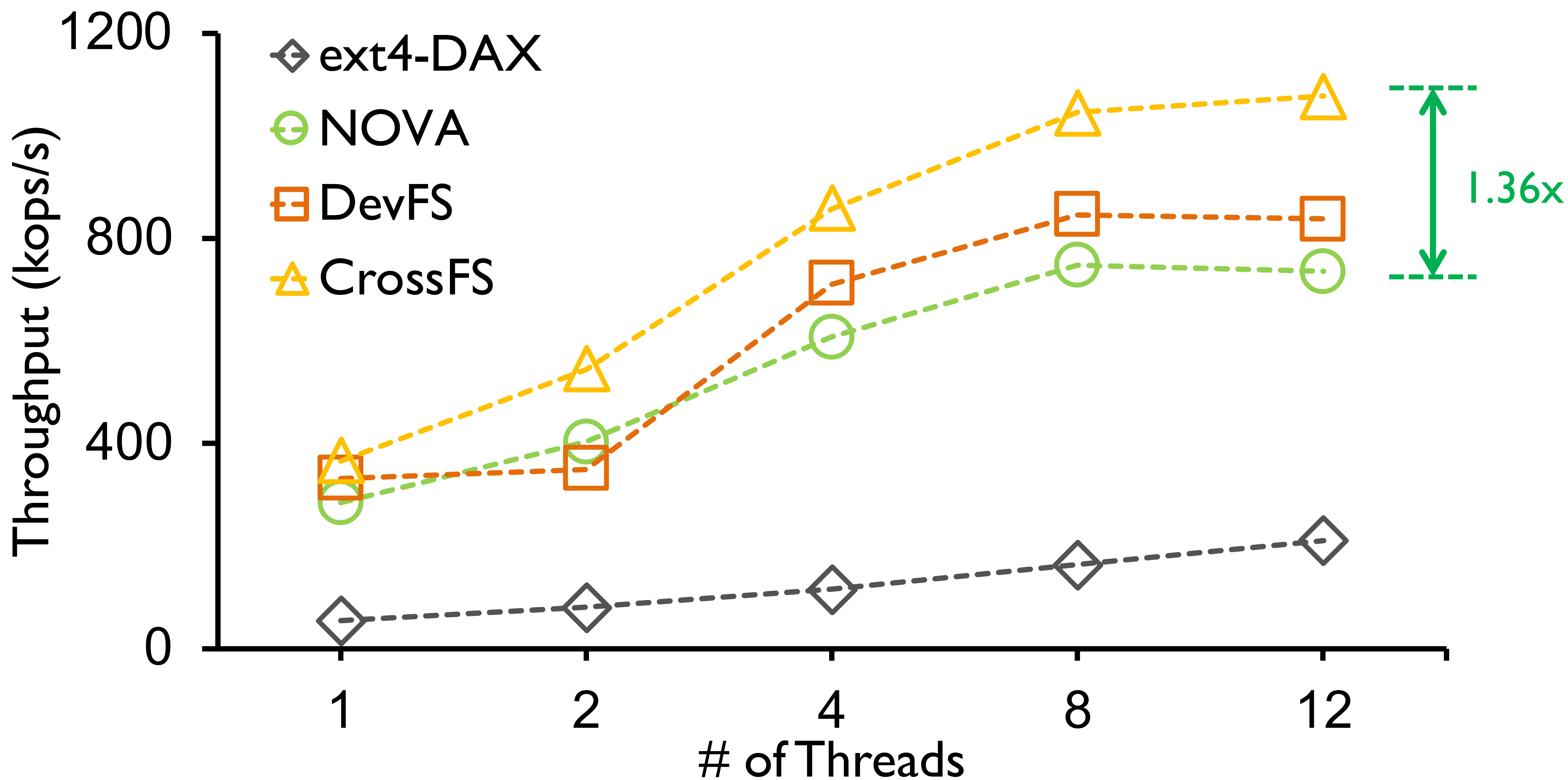
yujie.ren@rutgers.edu

Questions?

Backup Slides

Macro-benchmark: Filebench

Varmail (metadata-heavy workloads)

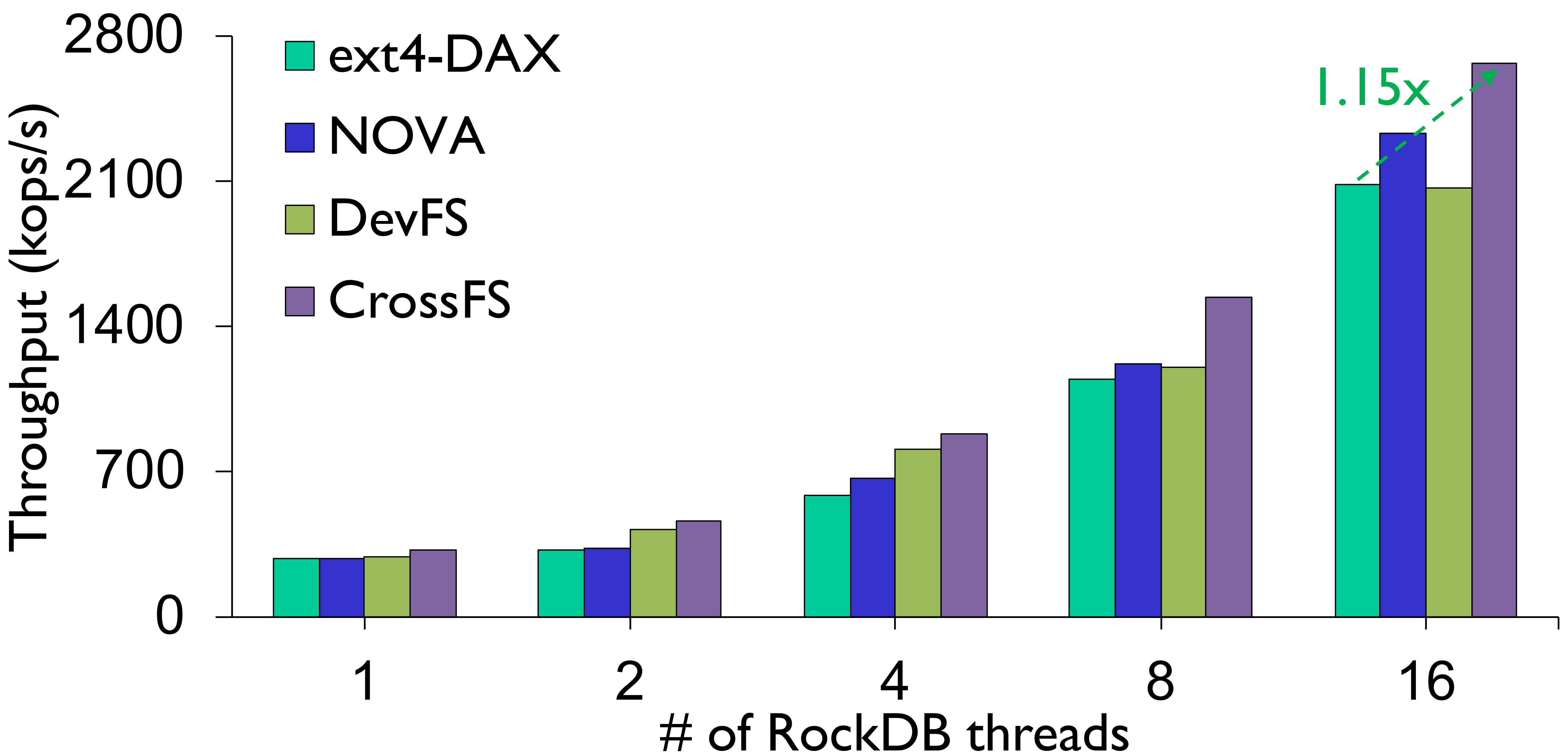


- X-axis: # of filebench threads
- Y-axis: benchmark throughput

CrossFS eliminate system call overheads

Application - RocksDB

DBbench ***readrandom*** benchmark



- X-axis: # of DBbench threads
- Y-axis: ***readrandom*** benchmark throughput

CrossFS eliminate system call overheads