File Systems as Processes

Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin Madison

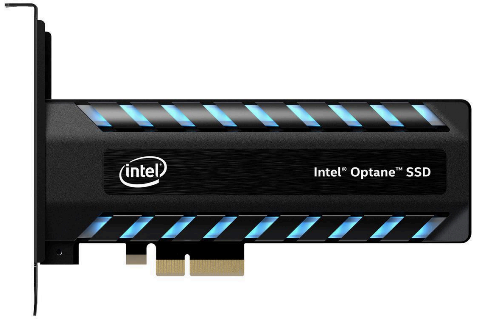
Sudarsun Kannan
Rutgers University

Motivation: #1 **Storage Devices Evolve Fast**

HDD

PCIe SSD

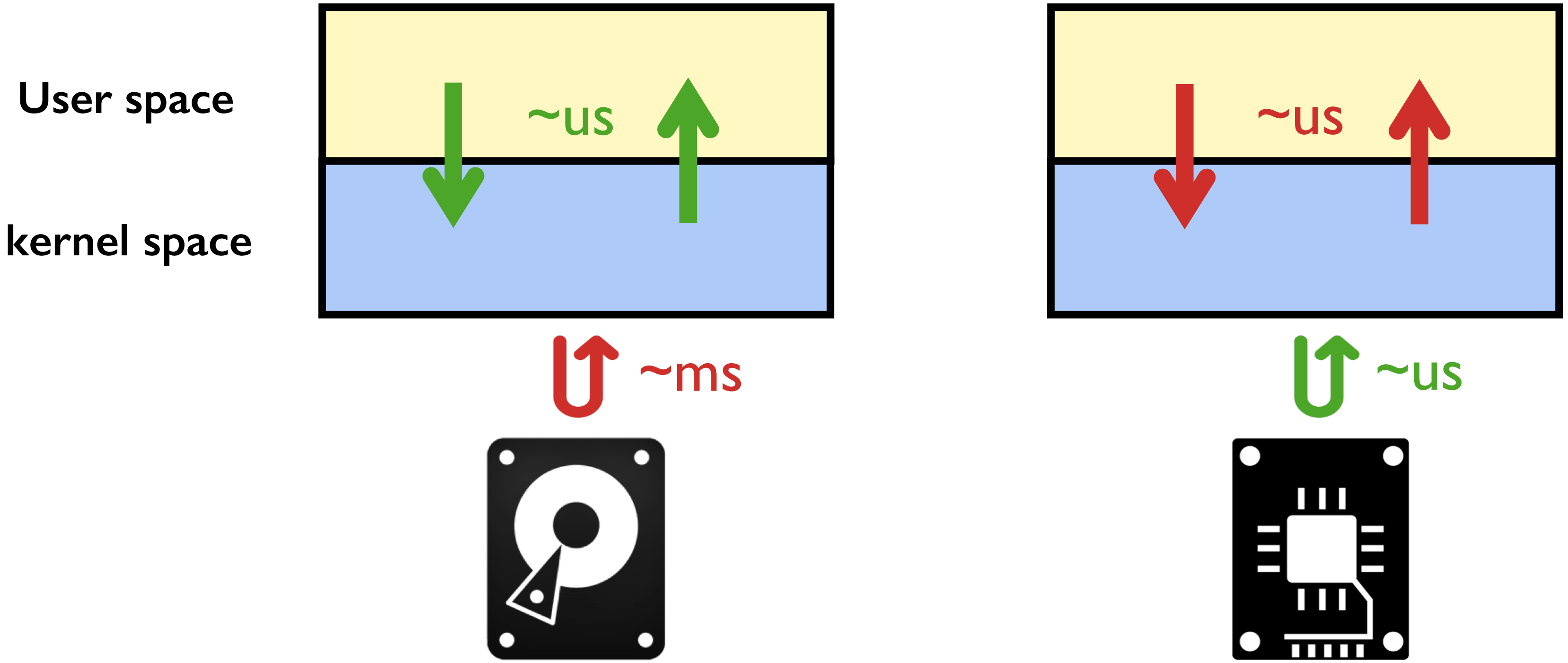
Ultra-fast Devices



	HDD	PCIe SSD	Ultra-fast Devices
IOPS:	1,000	47,000	550,000
BW:	55 MB/s	500 MB/s	2500 MB/s
Latency:	7.1 ms	160 us	10 us

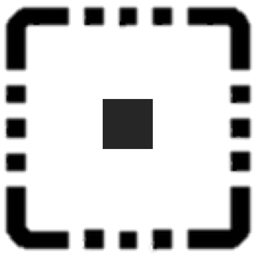
Motivation: #2 **OS Architectures fails behind**

- OS design decisions were made for millisecond-scale I/O devices
  - e.g., HDD access outweighs the cost of two context switches (microseconds)

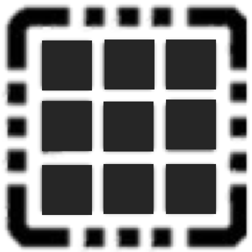


Motivation: #3 **File Systems born in single-core era**

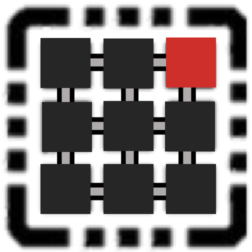
- Poor multi-core scalability
- Hard to leverage multi-core hardware features
  - e.g., fast inter-core communication, cache locality



Single-core CFS
& Kernel FS



Multi-core CFS
& Kernel FS



What if ?

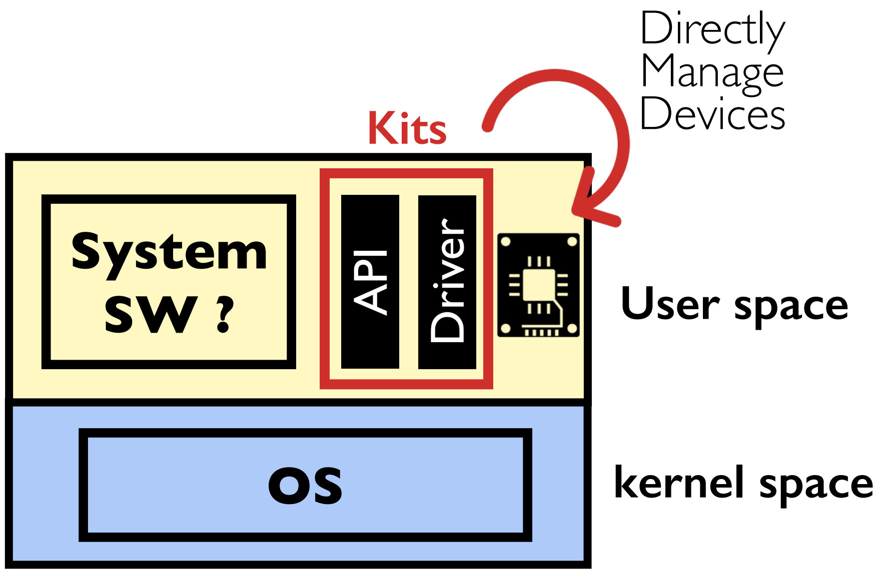
 core is running FS

CFS: Completely Fair Scheduler

Motivation: #4 **HW optimized toolkits are in the wild**

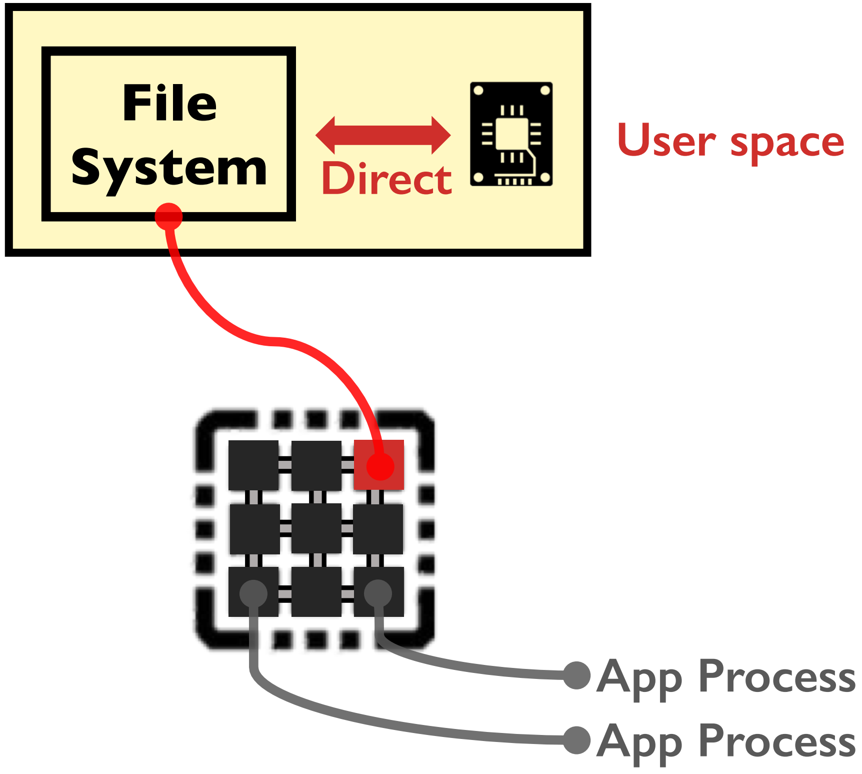
- Developing toolkits for high performance in userland:
  - Data Plane Development Kit (DPDK)
  - Storage Performance Development Kit (SPDK)
  - Threading Building Blocks (TBB)

- Valuable cornerstone for Storage Stack
  - Make FS development easier (than kernel)
  - Reconsider "legacy" OS design decisions:
    - Interrupt-based notification
    - Operating system managed threading



Our Idea: File Systems as Processes

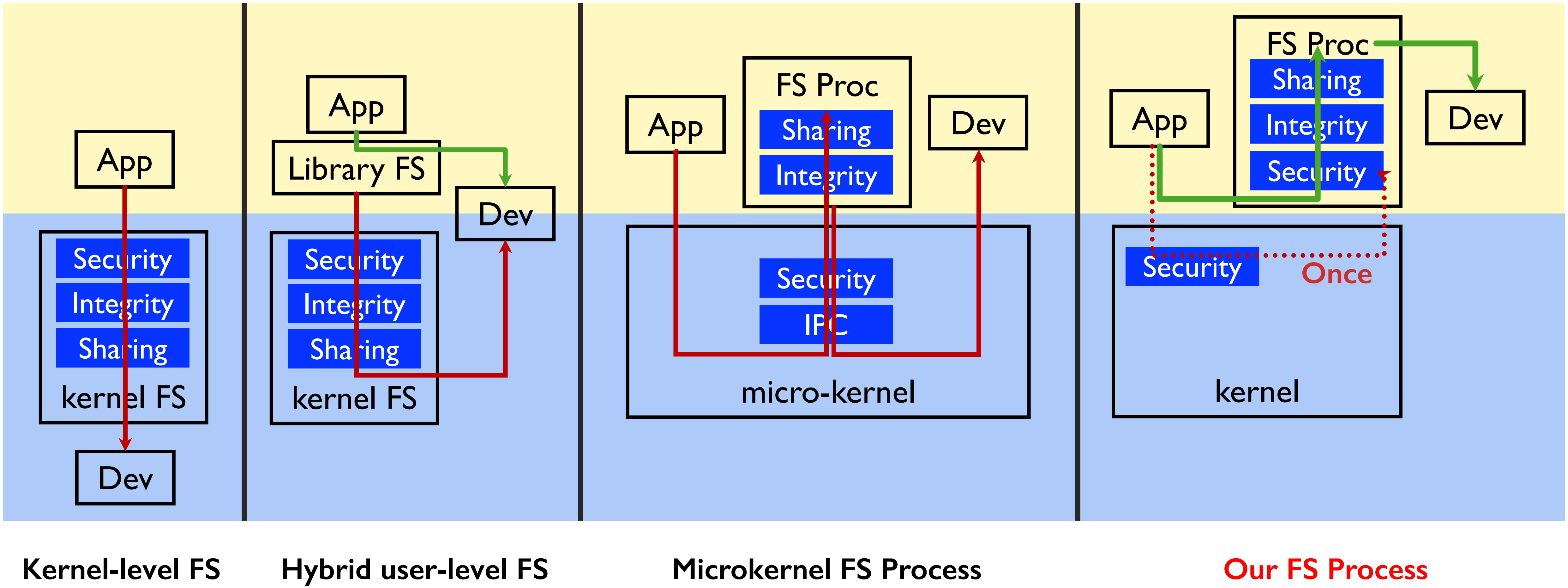
- A direct-access file system as a user-level process
- Advantages:
  - Developer velocity
  - Guarantee essential file system properties
    - integrity, concurrency, crash-consistency and security
  - High performance
- Prototype - DashFS



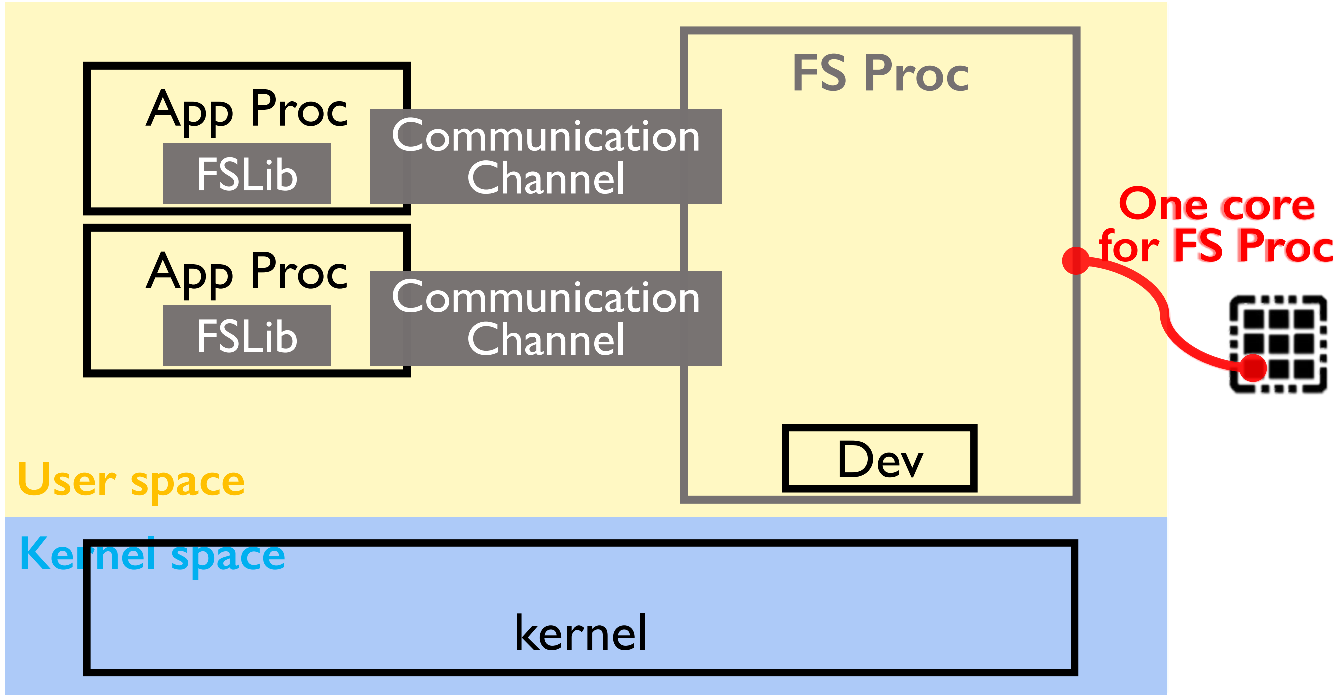
Outline

- Introduction
- FSP Architecture
- Challenges
- Prototype - DashFS
- Conclusion

Classes of File System Architectures



File Systems as Processes (FSP) Architecture

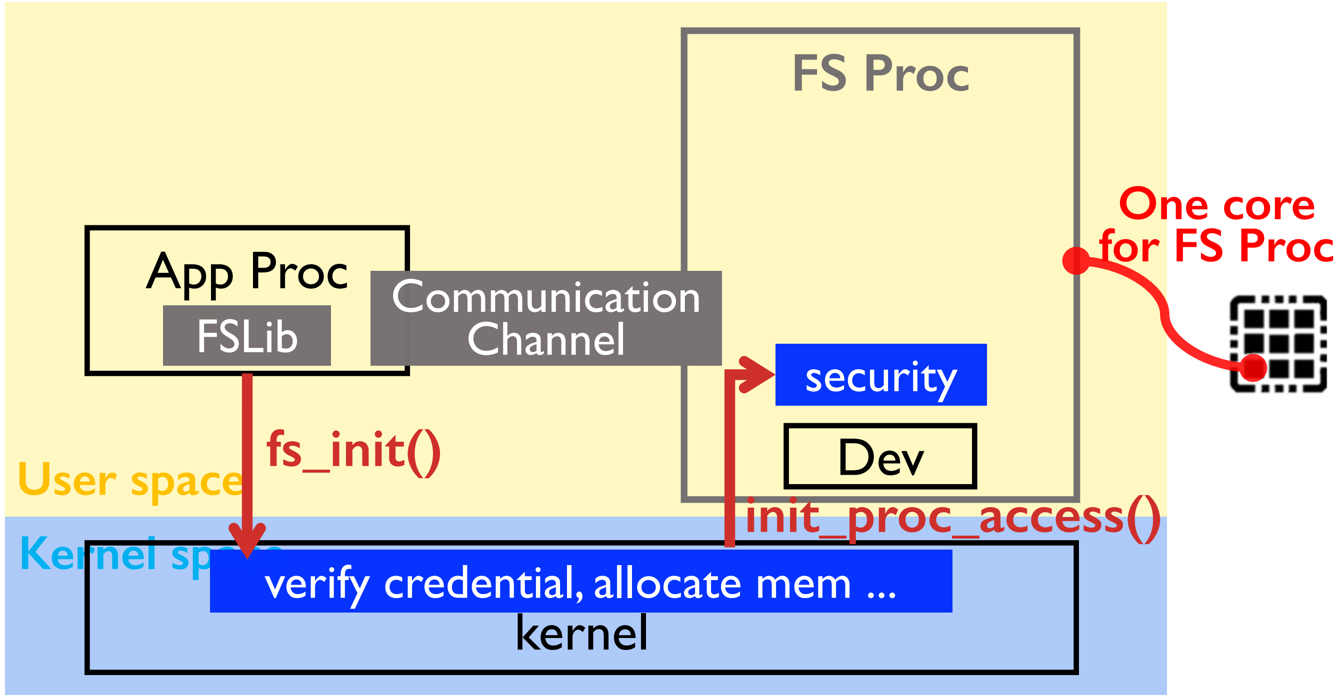


- FS Proc: a standalone user-level process
- FSLib: provides POSIX compatibility; send(recv) req(reply) to(from) Fs Proc
- Communication Channel: shared memory between App and FS Proc

Kernel is only involved **once** to securely set up Communication Channel

File Systems as Processes (FSP) Architecture

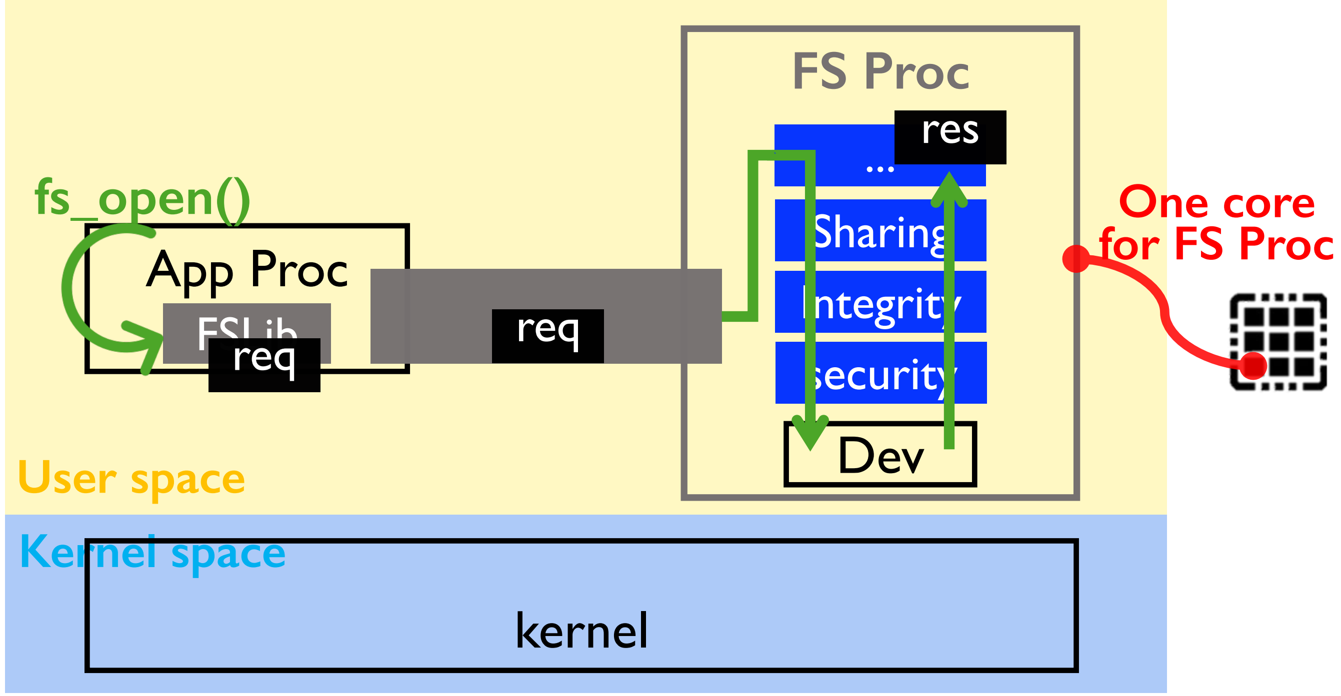
FS access initialization



- FS Proc: a standalone user-level process
- FSLib: provides POSIX compatibility; send(recv) req(reply) to(from) Fs Proc
- Communication Channel: shared memory between App and FS Proc

File Systems as Processes (FSP) Architecture

open a file



- FS Proc: a standalone user-level process
- FSLib: provides POSIX compatibility; send(recv) req(res) to(from) Fs Proc
- Communication Channel: shared memory between App and FS Proc

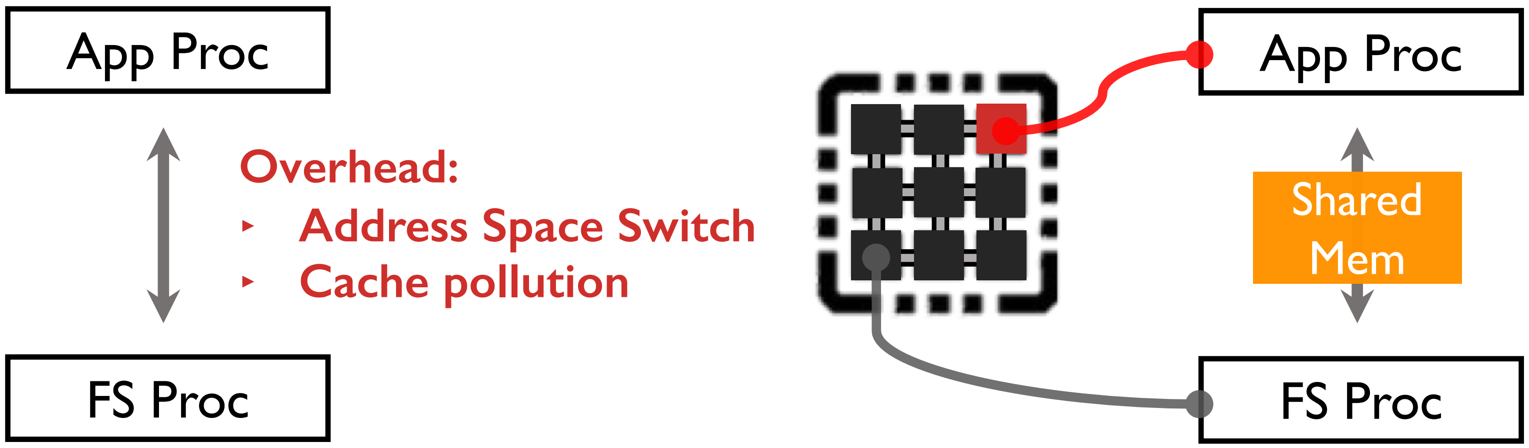
Challenges of FSP

- ➡ Efficient Communication
- ➡ Scheduling & Concurrency
- ➡ OS Coordination
- ➡ Reliability

Focus on challenges unique to FSP approach

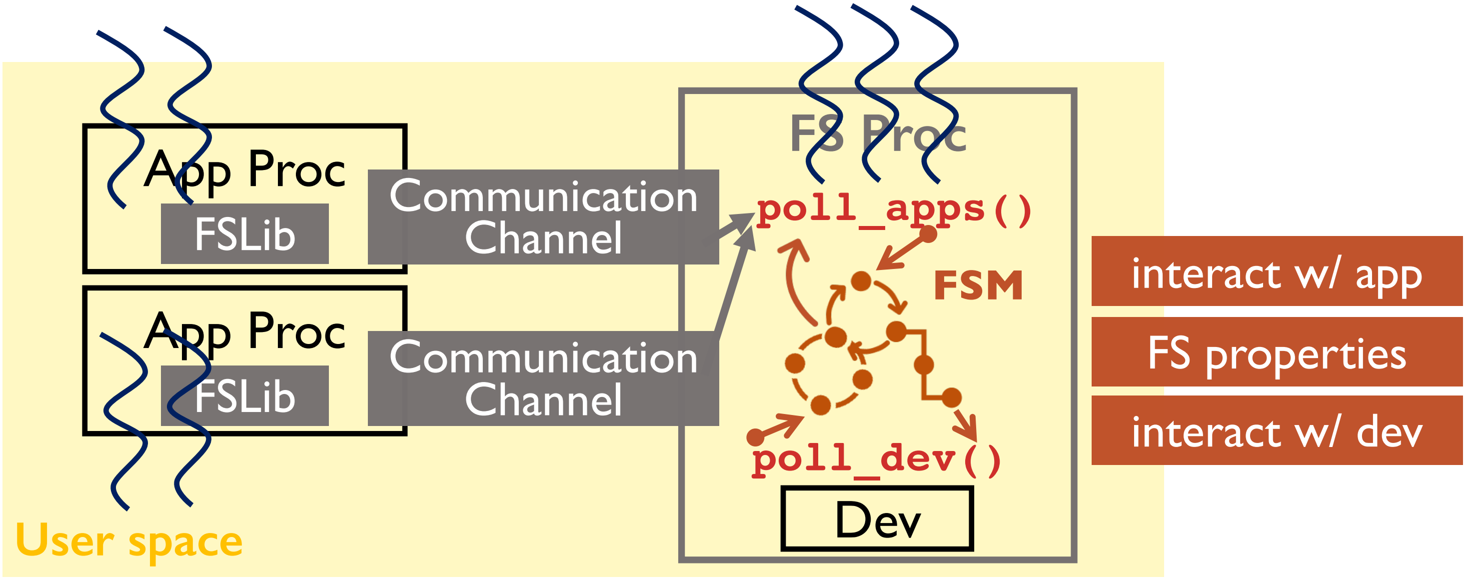
Efficient Communication

- The foundation of a high-performance file system process



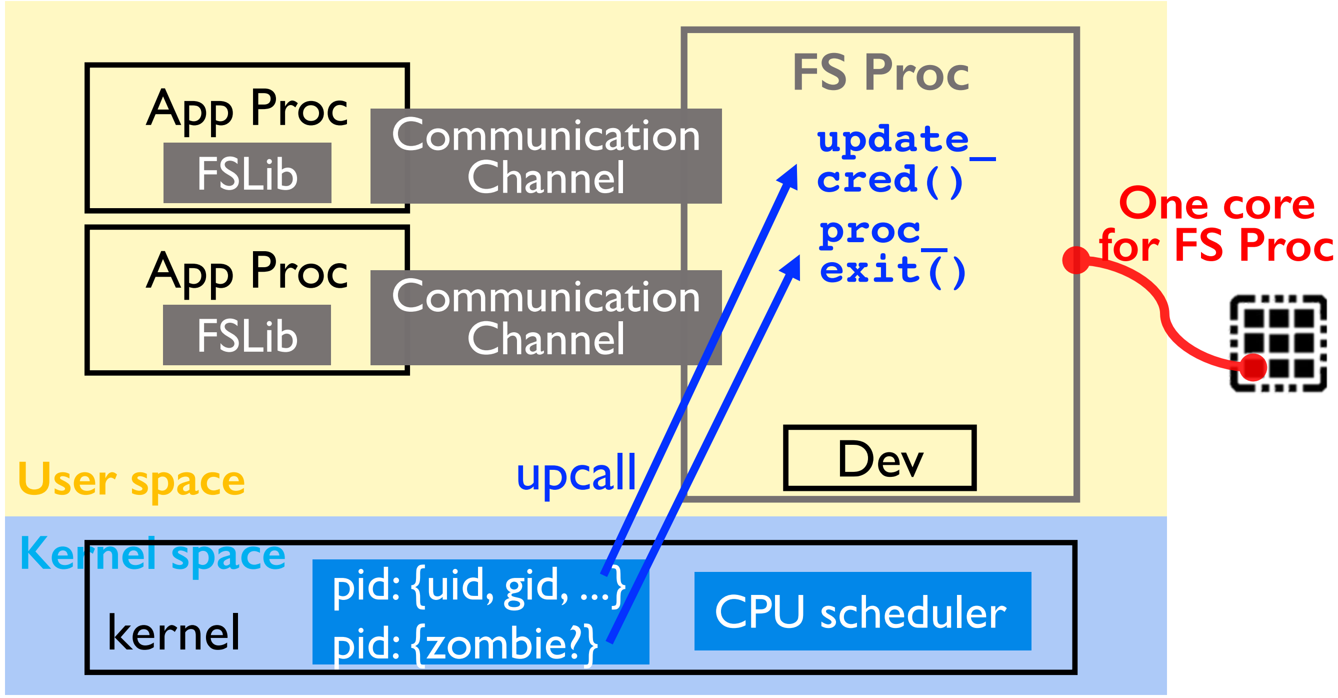
- Solution:
  - Leverage fast inter-core communication and cache-to-cache transfer
  - Specialized memory management

Scheduling & Concurrency



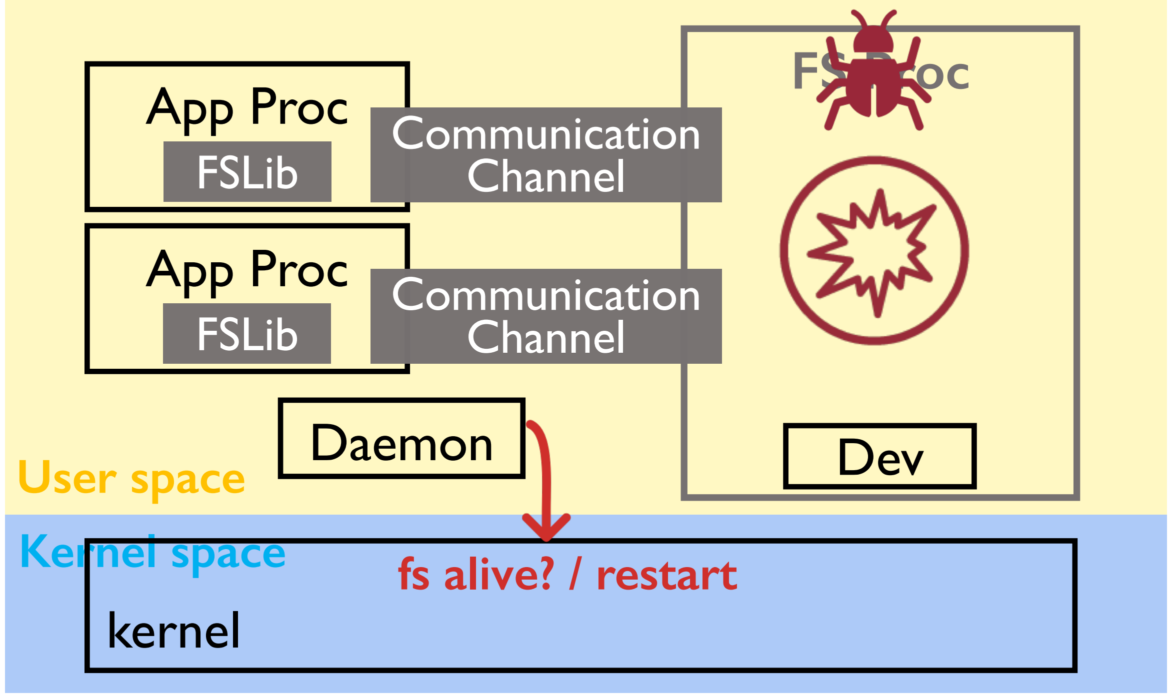
- More concurrency (threads) to be managed
- The complexity of threading (similar to building a web server)
- The complexity of asynchronous programming
  - Poll-mode driver (no interrupt) and complicated FSM cross several layers

OS Coordination



- I/O related information is maintained as part of the process's OS state
  - e.g., credential and process aliveness
- CPU scheduler should be aware of the core running FS

Reliability



- An new opportunity for applications to stay alive when FS crashes
  - Problems: crash detection and states reconstruction
- Backward mode which resembles kernel FS crash semantics

Outline

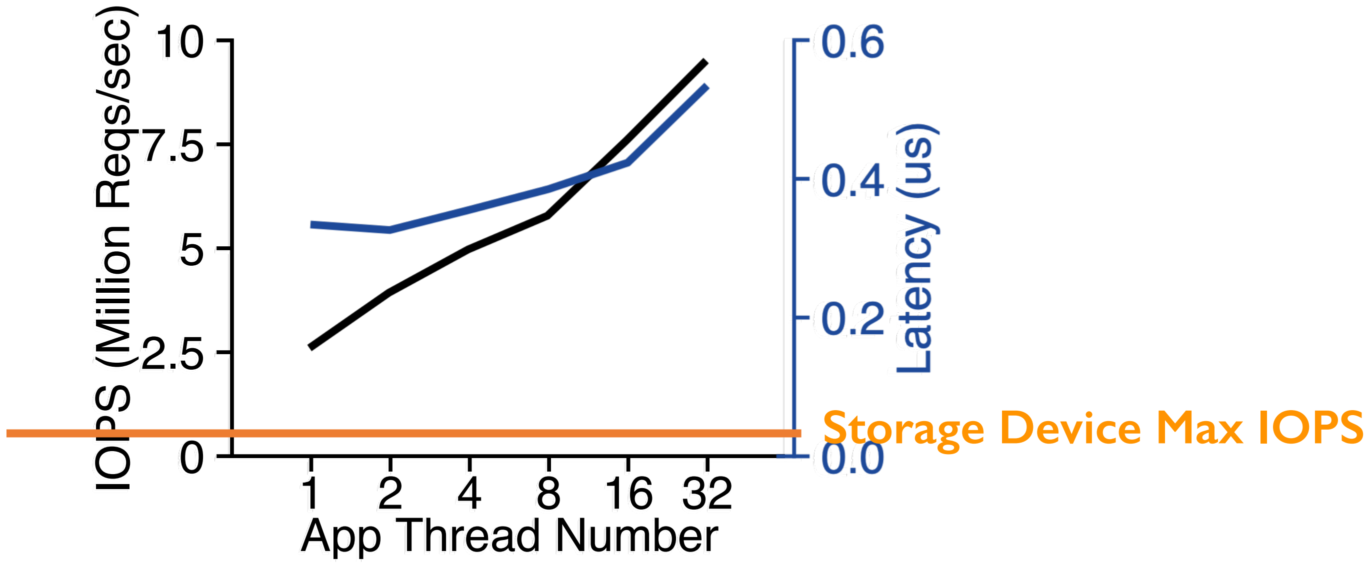
- Introduction
- FSP Architecture
- Challenges
- DashFS Prototype
- Conclusion

DashFS Prototype

- Current Status:
  - Support open(), read(), write(), close(), stat(), sync() and init()
  - **Efficient Communication** is in hand
  - Working on the rest three challenges
- Evaluation:
  - The communication channel is efficient
  - Micro-benchmark results are promising
- Experiment Platform:
  - Intel i7-8700K CPU, 32G RAM and an Intel Optane SSD 905P (960GB)

The communication channel is efficient

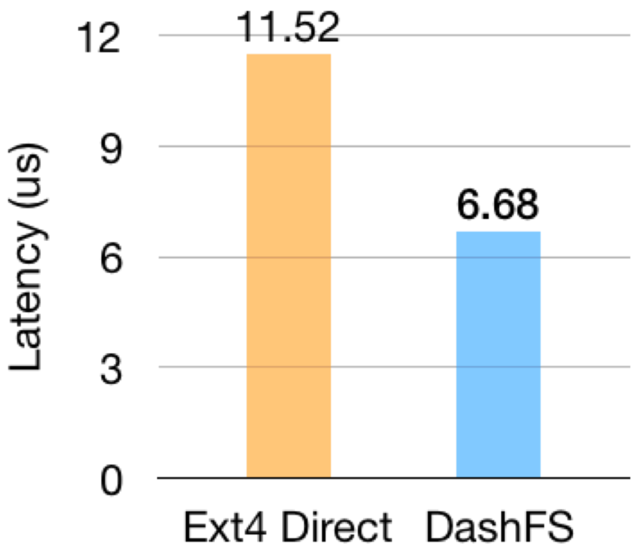
- An Application issues 4KB sequential write requests through various # of threads
  - Uses memory as backend



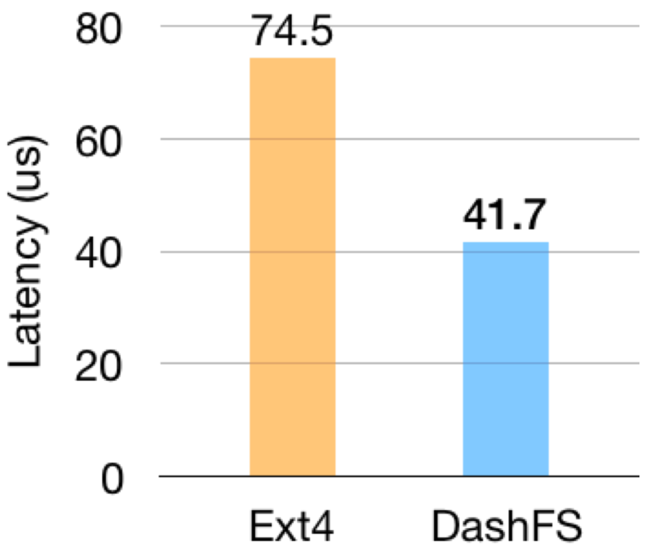
- Unlikely to be a throughput bottleneck
- Able to achieve sub-microsecond latency

Micro-benchmark Results

- Single Operation:
  - 4K Random Read to single file



- Multiple operations:
  - create() ➡ write() ➡ sync() ➡ close()
  - Several traps when using ext4



Conclusion

- Towards a storage era of microsecond latency
  - Eliminating software (OS) overhead is critical
  - Without compromising essential file system properties
- Building a file system as a user-level process can be a promising avenue
  - Great development velocity
  - Leverage inter-core communication
  - Initial results present significant performance gain
- We are working on tackling more challenges via DashFS