CrashMonkey: A Framework to Systematically Test File-System Crash Consistency

Ashlie Martinez

Vijay Chidambaram

University of Texas at Austin

The University of Texas at Austin

Crash Consistency

- File-system updates change multiple blocks on storage
  - Data blocks, inodes, and superblock may all need updating
  - Changes need to happen atomically
  - Need to ensure file system consistent if system crashes
- Ensures that data is not lost or corrupted
  - File data is correct
  - Links to directories and files unaffected
  - All free data blocks are accounted for
- Techniques: journaling, copy-on-write
- Crash consistency is complex and hard to implement

Testing Crash Consistency

- Randomly power cycling a VM or machine
  - Random crashes unlikely to reveal bugs
  - Restarting machine or VM after crash is slow
- Killing user space file-system process
  - Requires special file-system design
- Ad-hoc
  - Despite its importance, no standardized or systematic tests

What Really Needs Tested?

- Current tests write data to disk each time
- Crashing while writing data is not the goal
- True goal is to generate disk states that crash could cause

CrashMonkey

Framework to test crash consistency

Works by constructing crash states for given workload

Does not require reboot of OS/VM

File-system agnostic

Modular, extensible

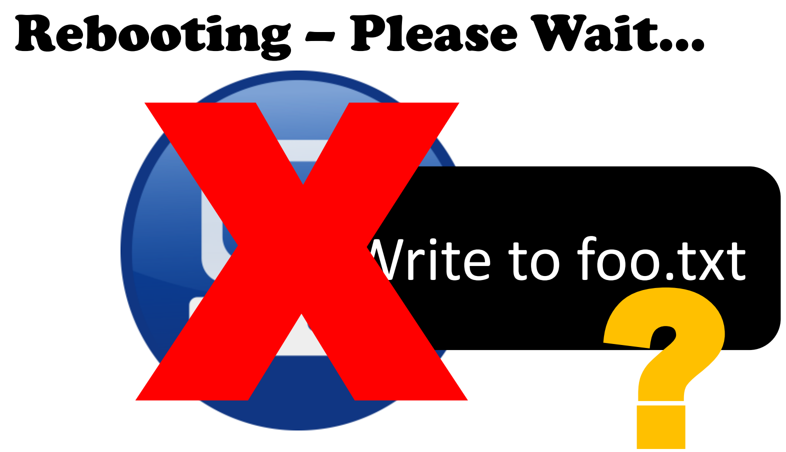
Currently tests 100,000 crash states in ~10min

Outline

- Overview
- How Consistency is Tested Today
- Linux Writes
- CrashMonkey
- Preliminary Results
- Future Plans
- Conclusion

How Consistency Is Tested Today

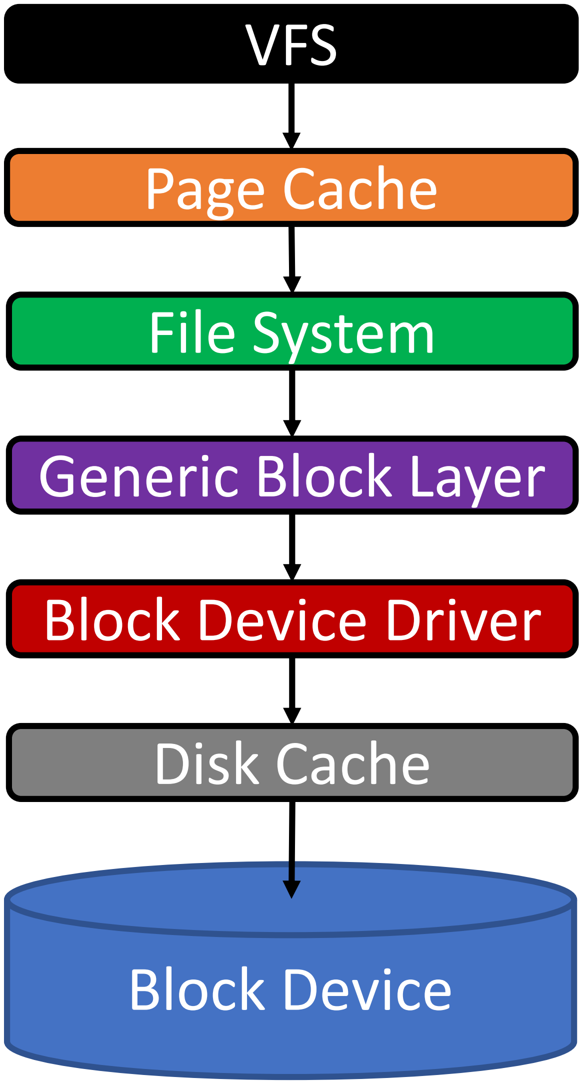
- Power cycle a machine or VM
  - Crash machine/VM while data is being written to disk
  - Reboot machine and check file system
  - **Random and slow**
- Run file system in user space
  - ZFS test strategy
  - Kill file system user process during write operations
  - **Requires file system have the ability to run in user space**



Outline

- Overview
- How Consistency is Tested Today
- Linux Writes
- CrashMonkey
- Preliminary Results
- Future Plans
- Conclusion

Linux Storage Stack



Provides consistent interface across file systems

Holds recently used files and data

Ext, NTFS, *etc.*

Interface between file systems and device drivers

Device specific driver

Caches data on block device

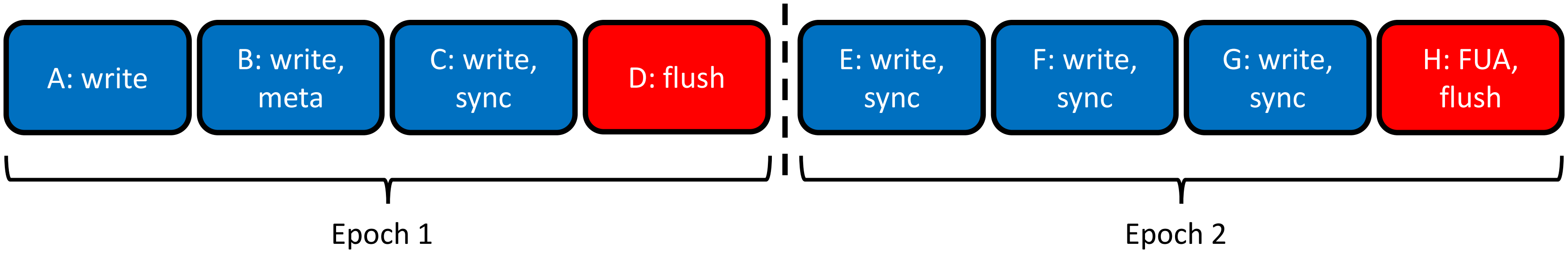
Persistent storage device

Linux Writes – Write Flags

- Metadata attached to operations sent to device driver
- Change how the OS and device driver order operations
  - Both IO scheduler and disk cache reorder requests
- sync – denotes process waiting for this write
  - Orders writes issued with sync in that process
- flush – all data in the device cache should be persisted
  - If request has data, data may not be persisted at return
- Forced Unit Access (FUA) – return when data is persisted
  - Often paired with flush so all data including request is durable

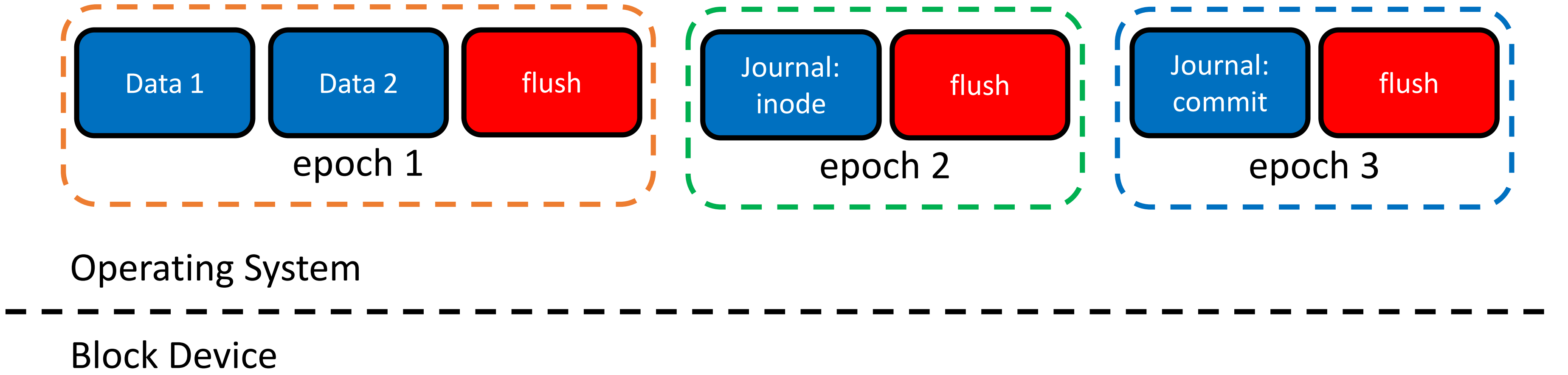
Linux Writes

- Data written to disk in epochs
  - each terminated by flush and/or FUA operations
- Reordering within epochs
  - Operating system adheres to FUA, flush, and sync flags
  - Block device adheres to FUA and flush flags



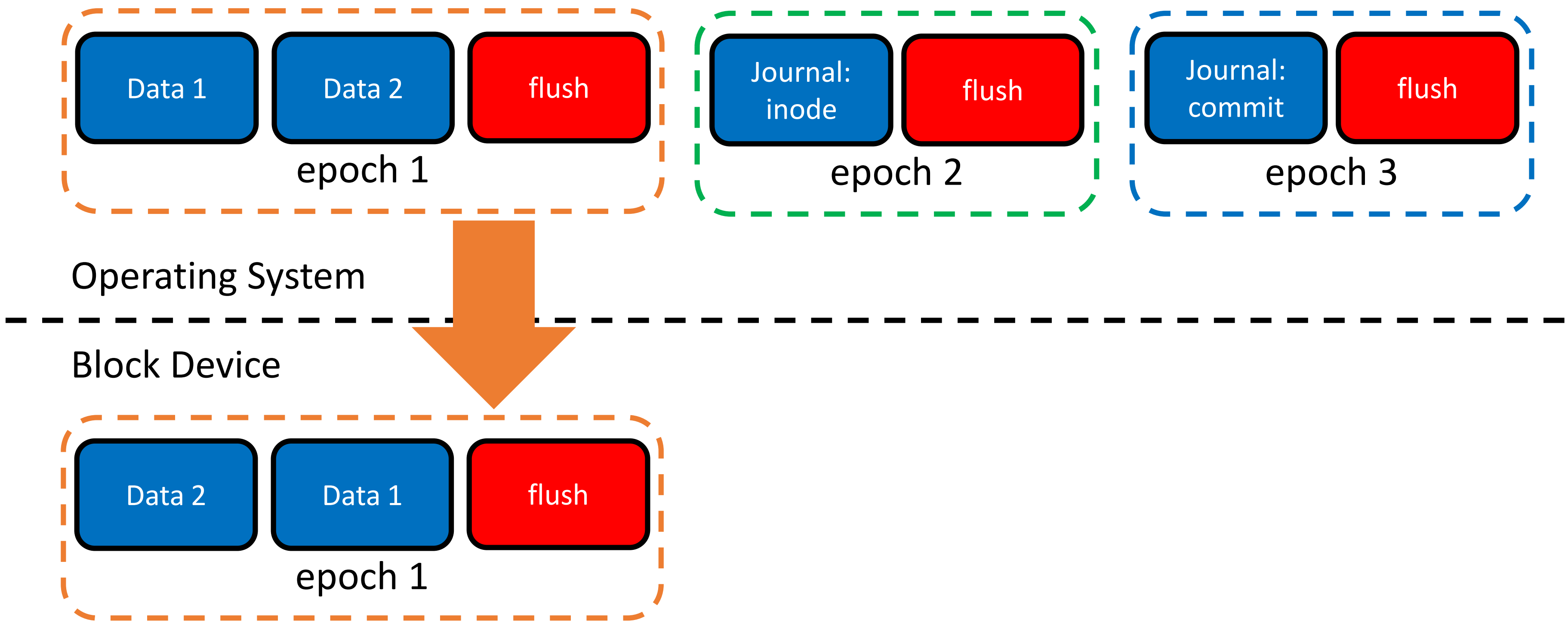
Linux Writes – Example

`echo "Hello World!" > foo.txt`



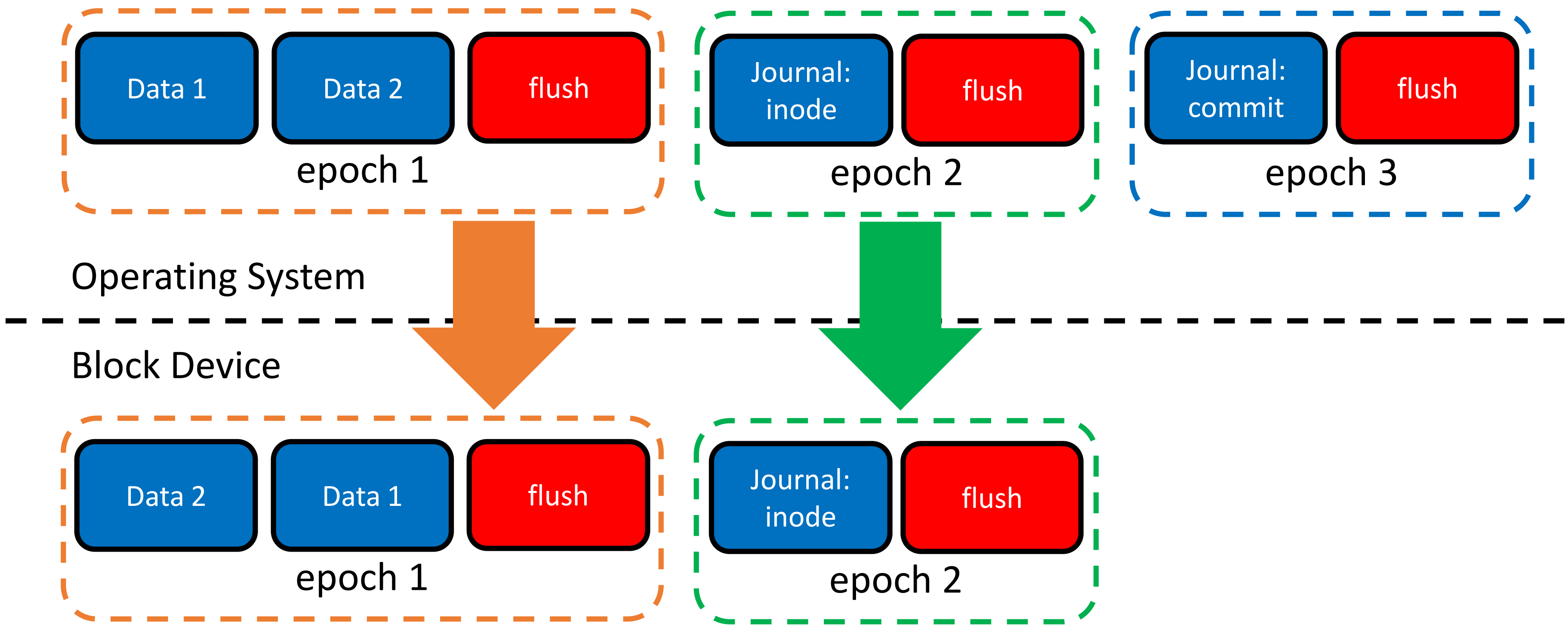
Linux Writes – Example

```
echo "Hello World!" > foo.txt
```



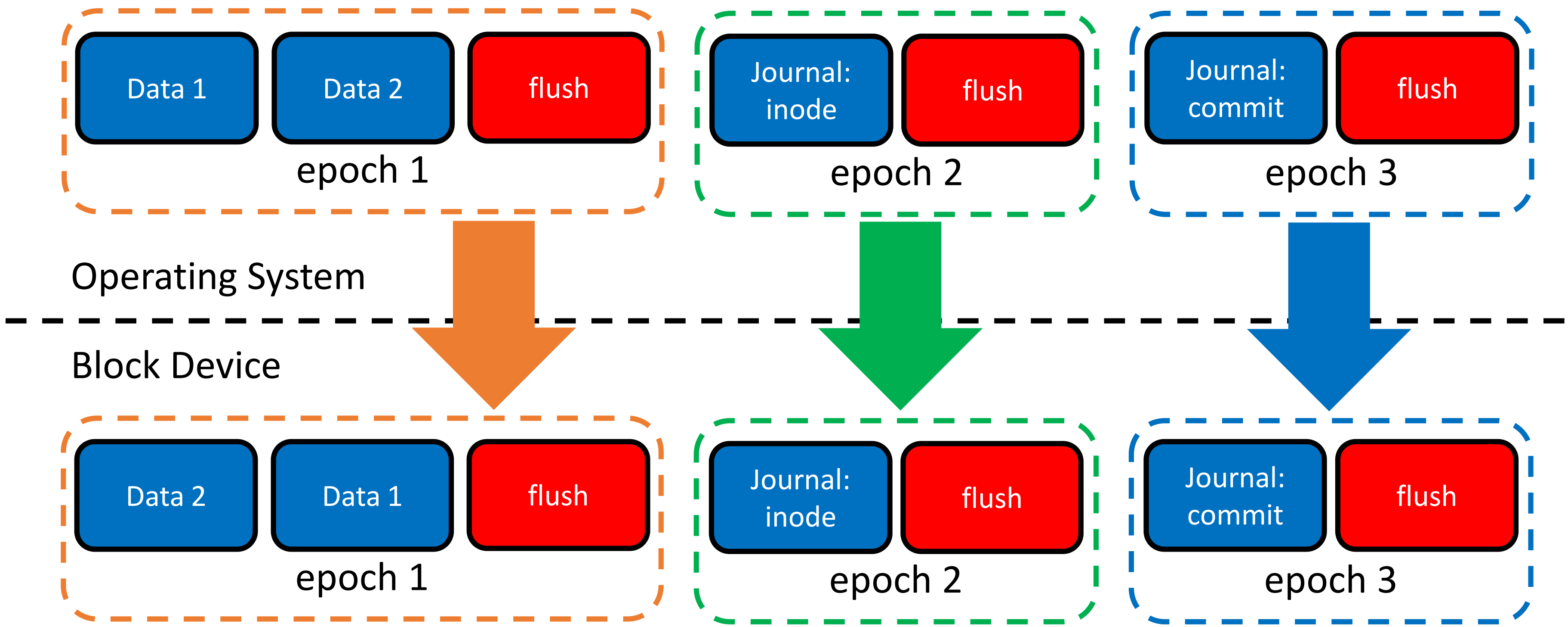
Linux Writes – Example

```
echo "Hello World!" > foo.txt
```



Linux Writes – Example

```
echo "Hello World!" > foo.txt
```



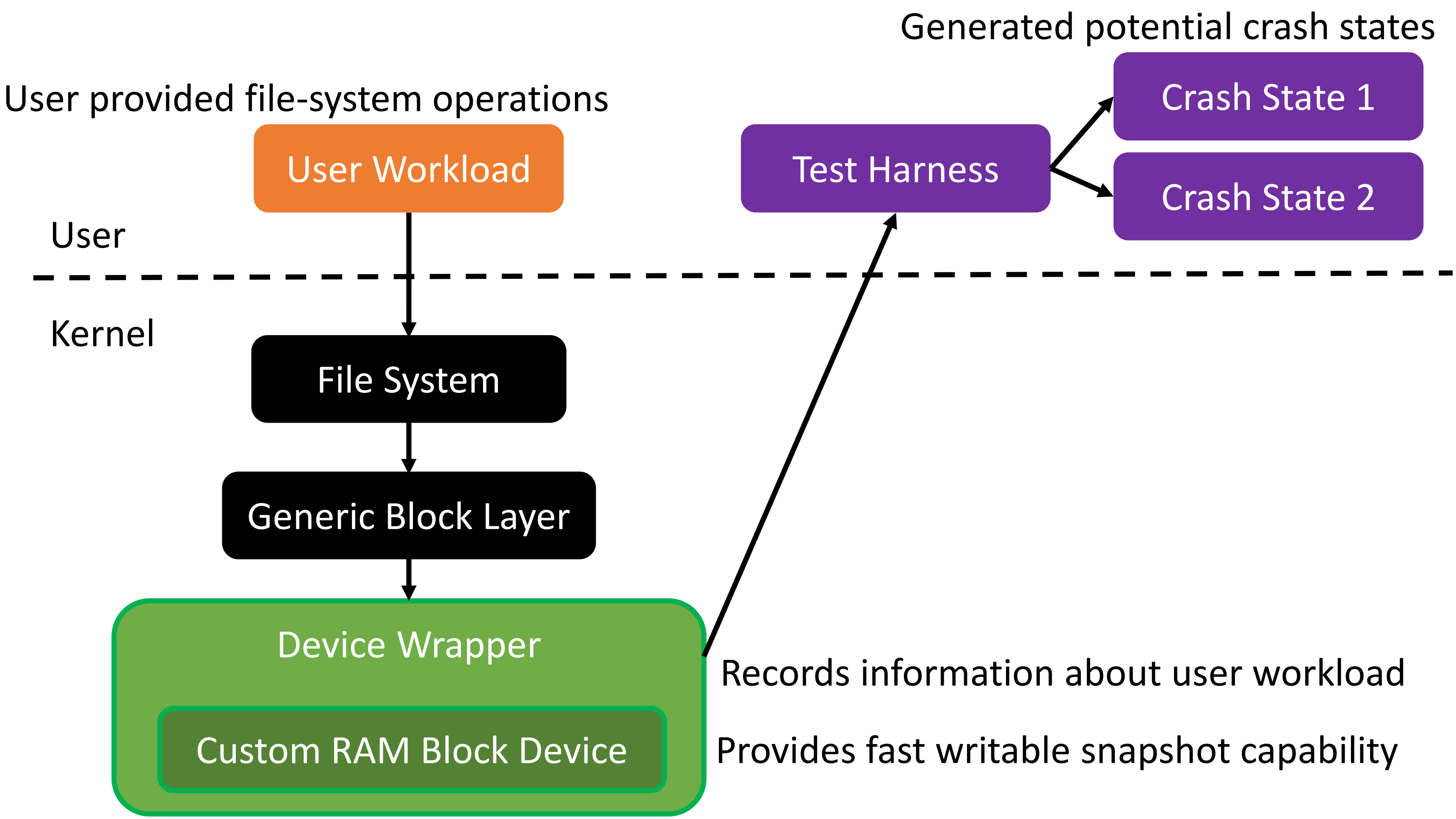
Outline

- Overview
- How Consistency is Tested Today
- Linux Writes
- CrashMonkey
- Preliminary Results
- Future Plans
- Conclusion

Goals for CrashMonkey

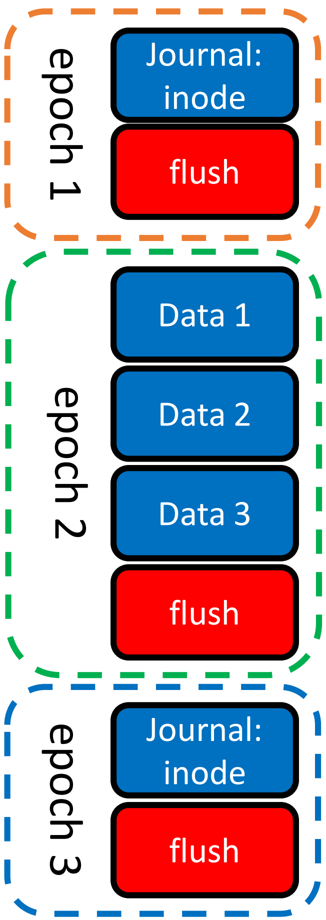
- Fast
- Ability to intelligently and systematically direct tests toward interesting crash states
- File-system agnostic
- Works out of the box without the need for recompiling the kernel
- Easily extendable and customizable

CrashMonkey: Architecture



Constructing Crash States

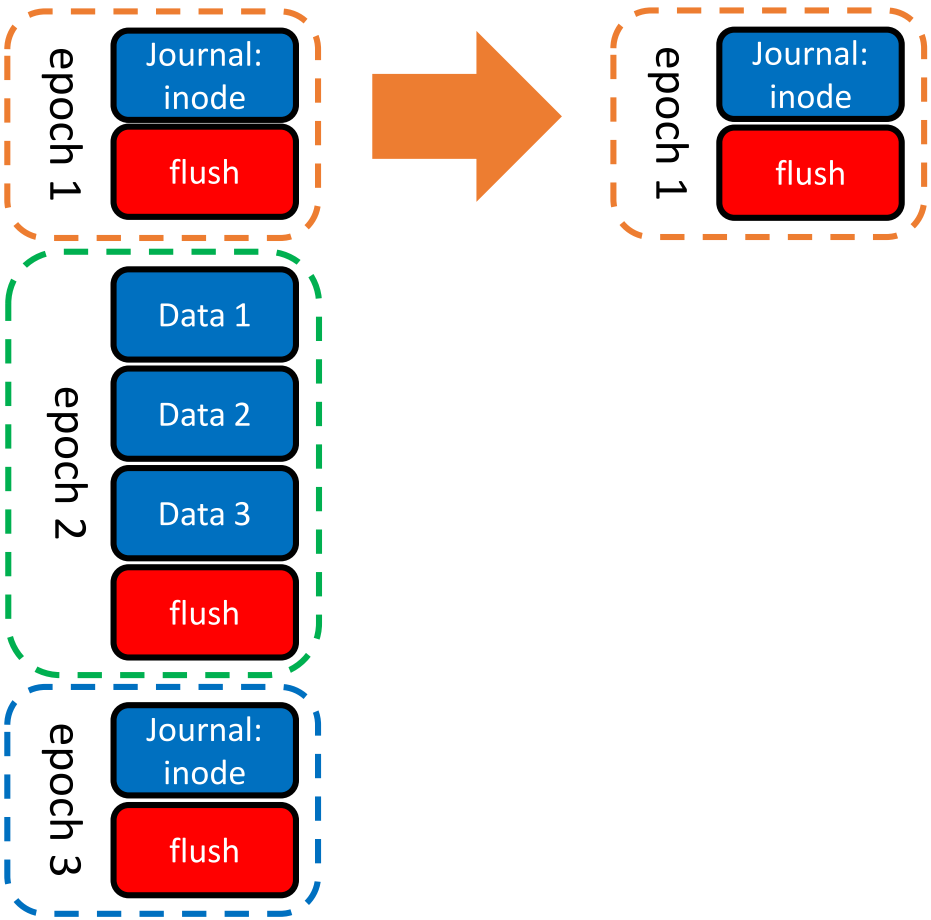
```
touch foo.txt
echo "foo bar baz" > foo.txt
```



Randomly choose n epochs to permute (n = 2 here)

Constructing Crash States

```
touch foo.txt
echo "foo bar baz" > foo.txt
```

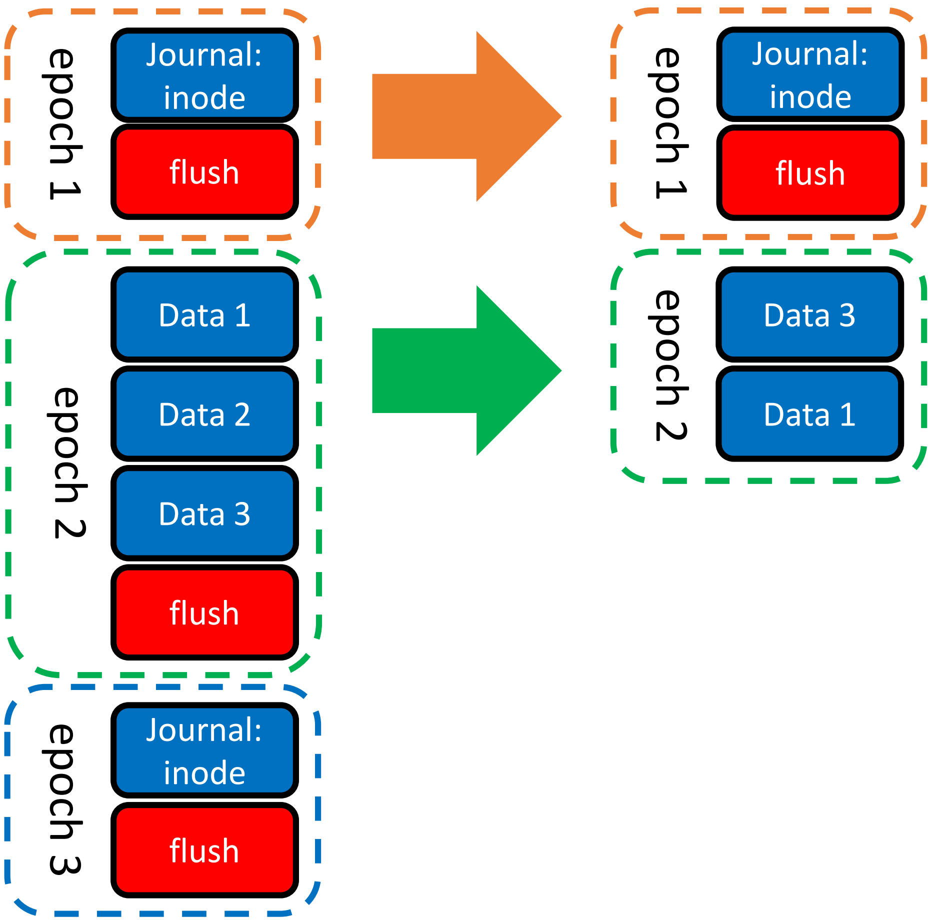


Randomly choose n epochs to permute (n = 2 here)

Copy epochs [1, n – 1]

Constructing Crash States

```
touch foo.txt
echo "foo bar baz" > foo.txt
```

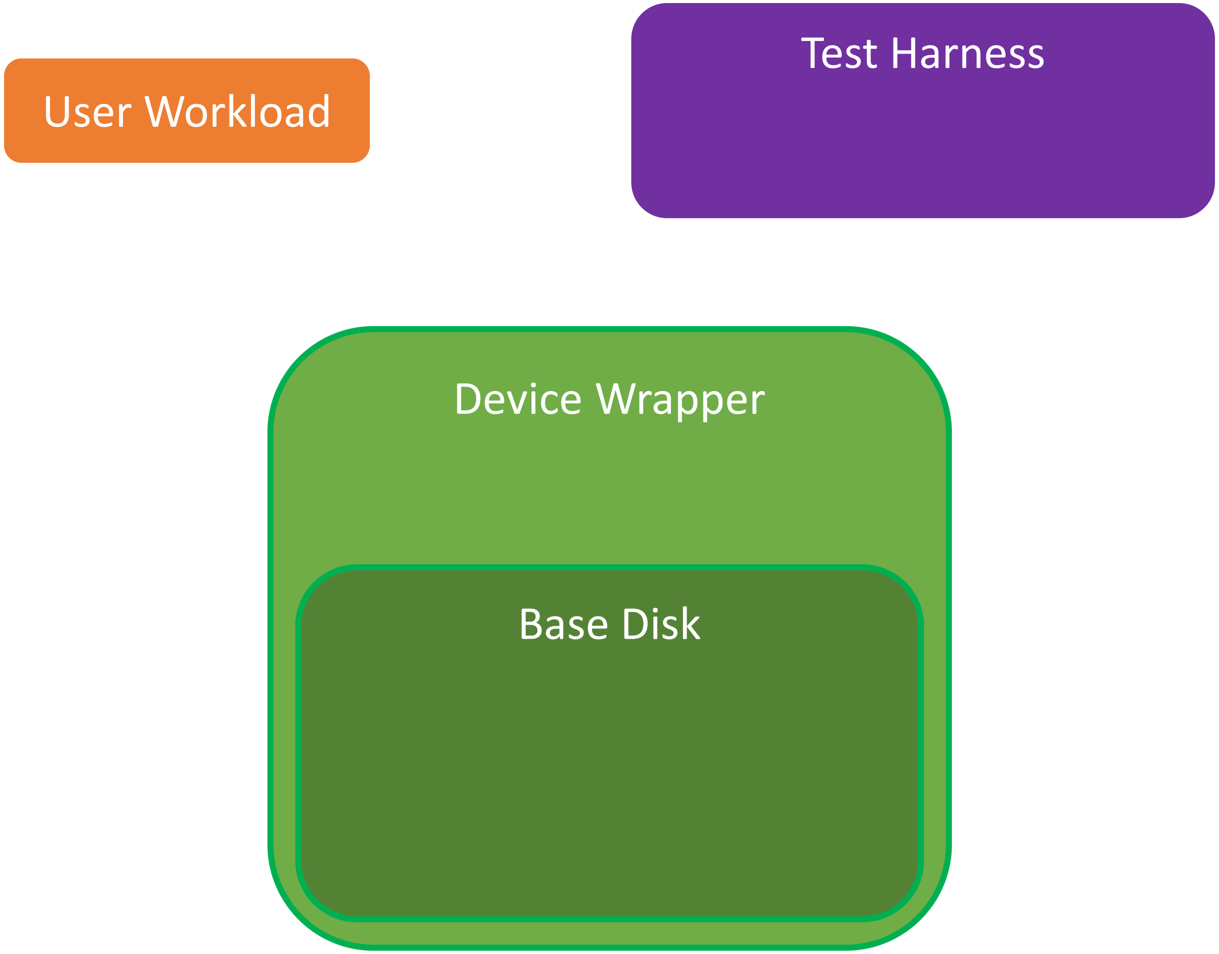


Randomly choose n epochs to permute (n = 2 here)

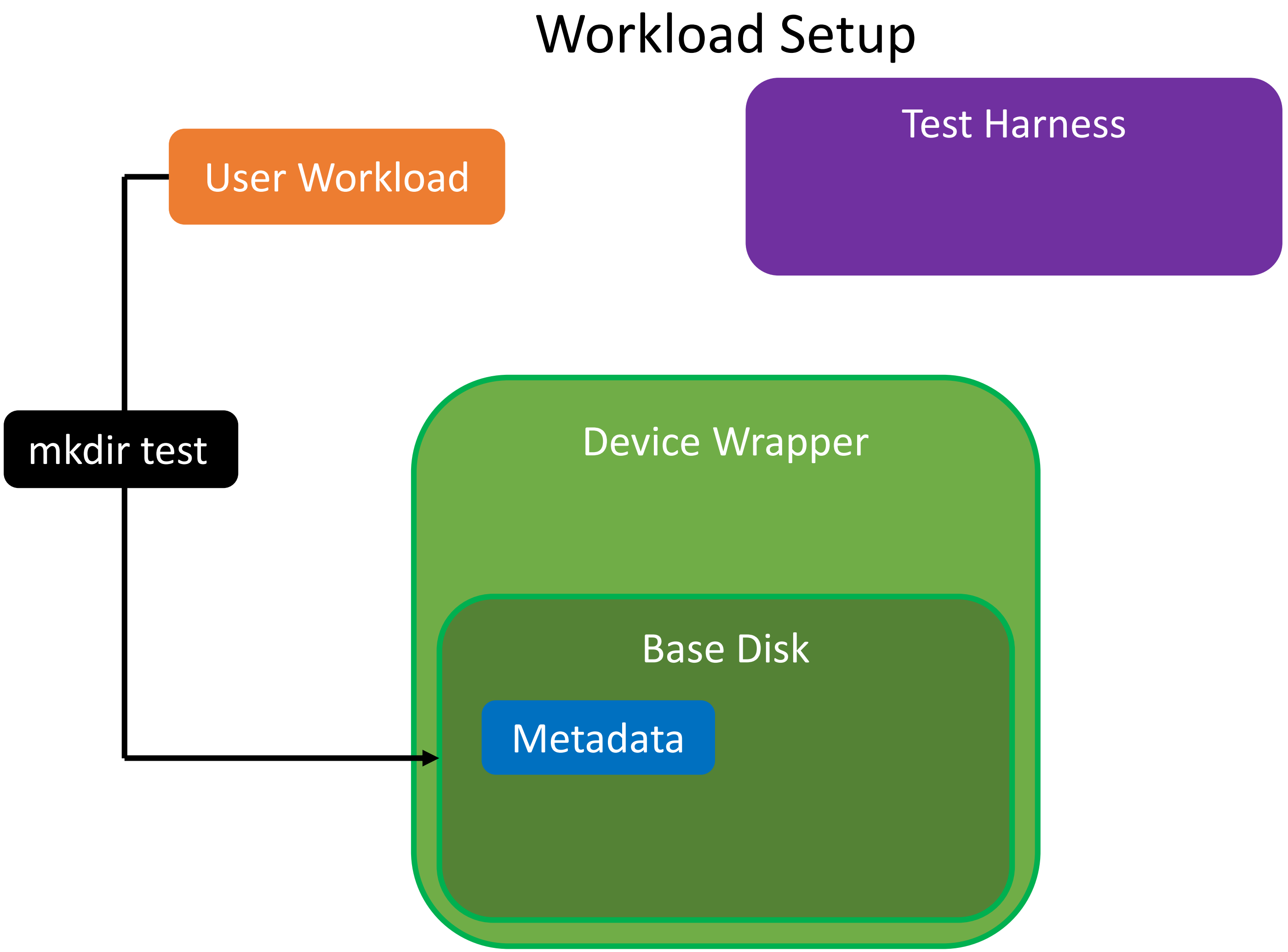
Copy epochs [1, n – 1]

Permute and possibly drop operations from epoch n

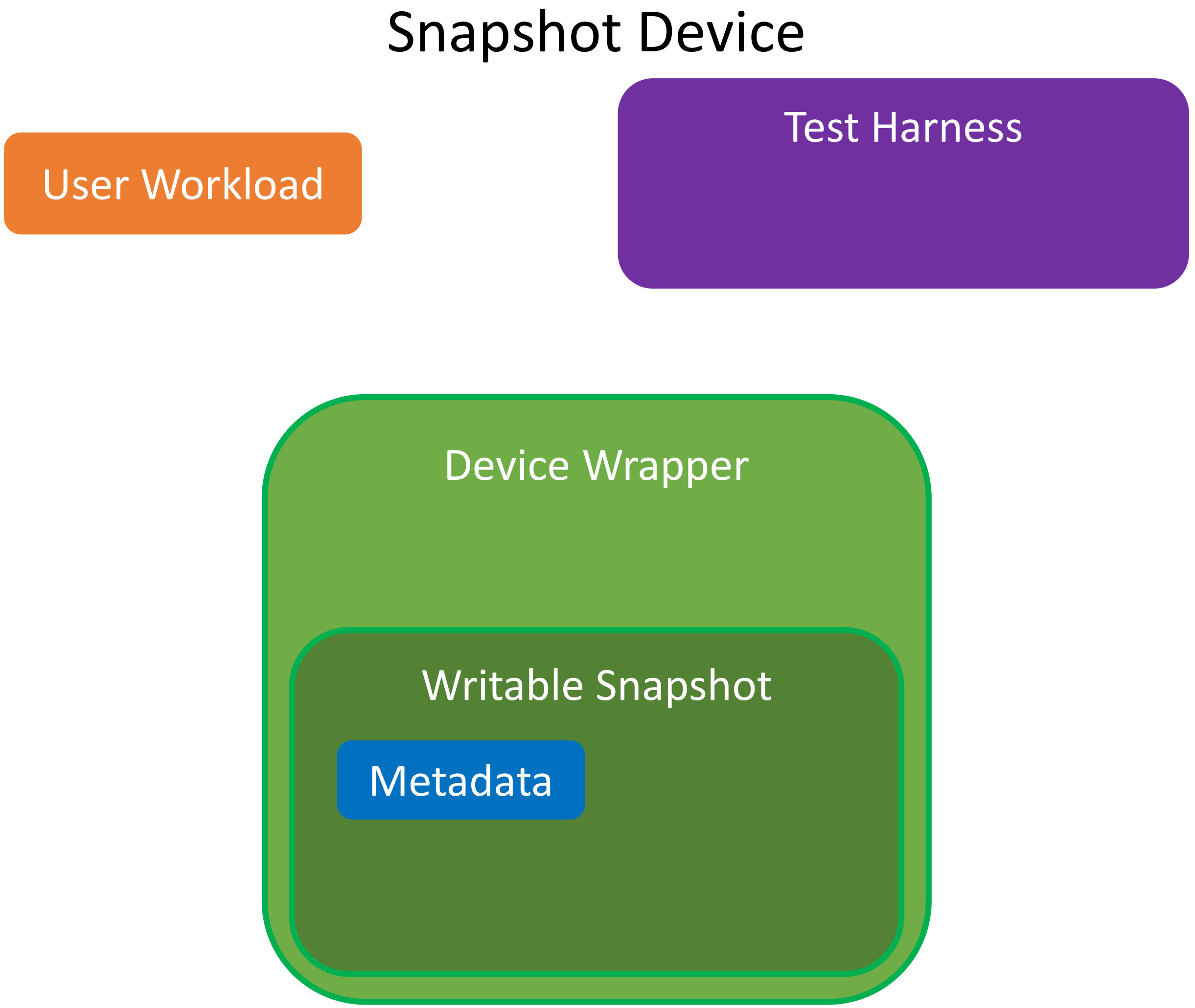
CrashMonkey In Action



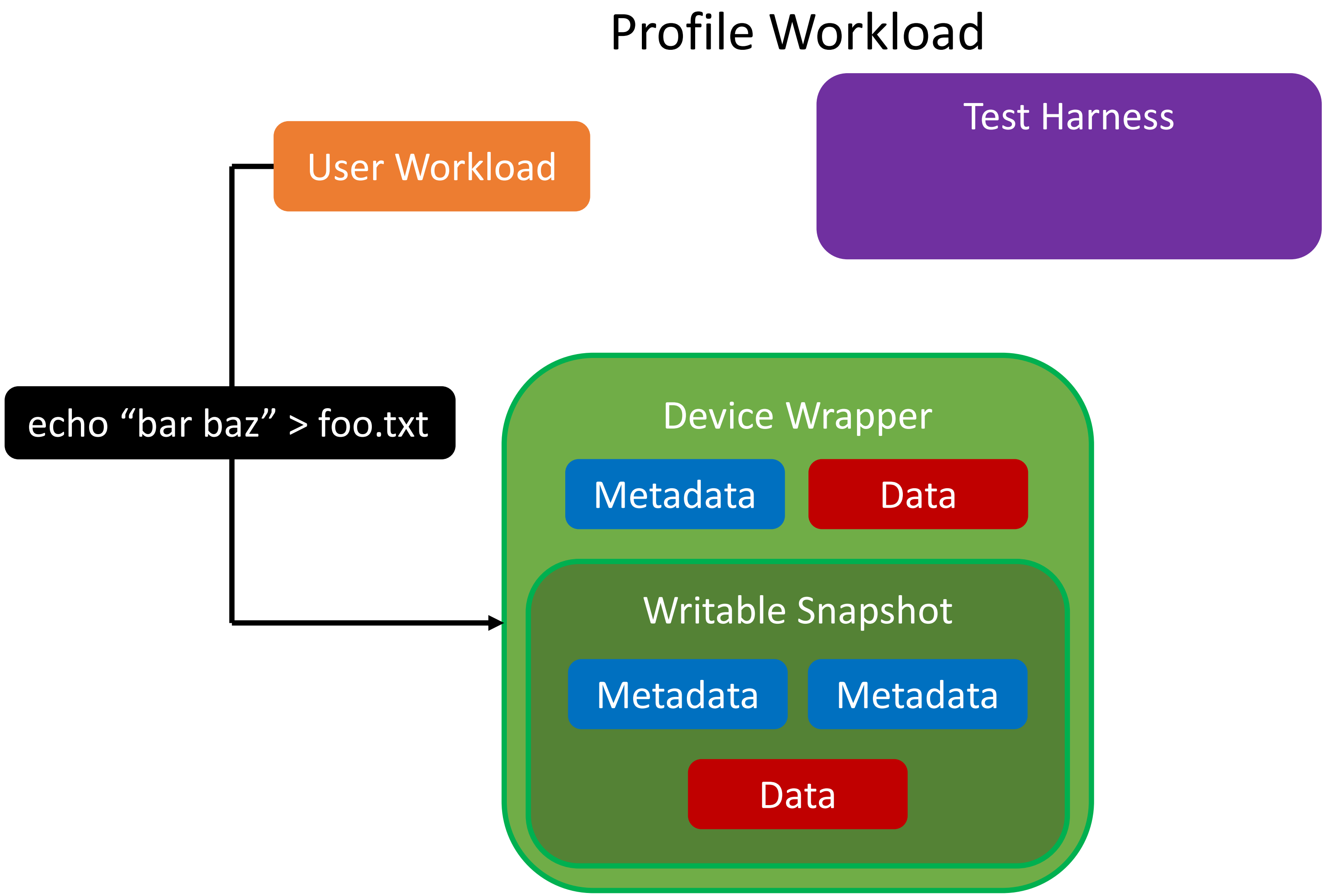
CrashMonkey In Action



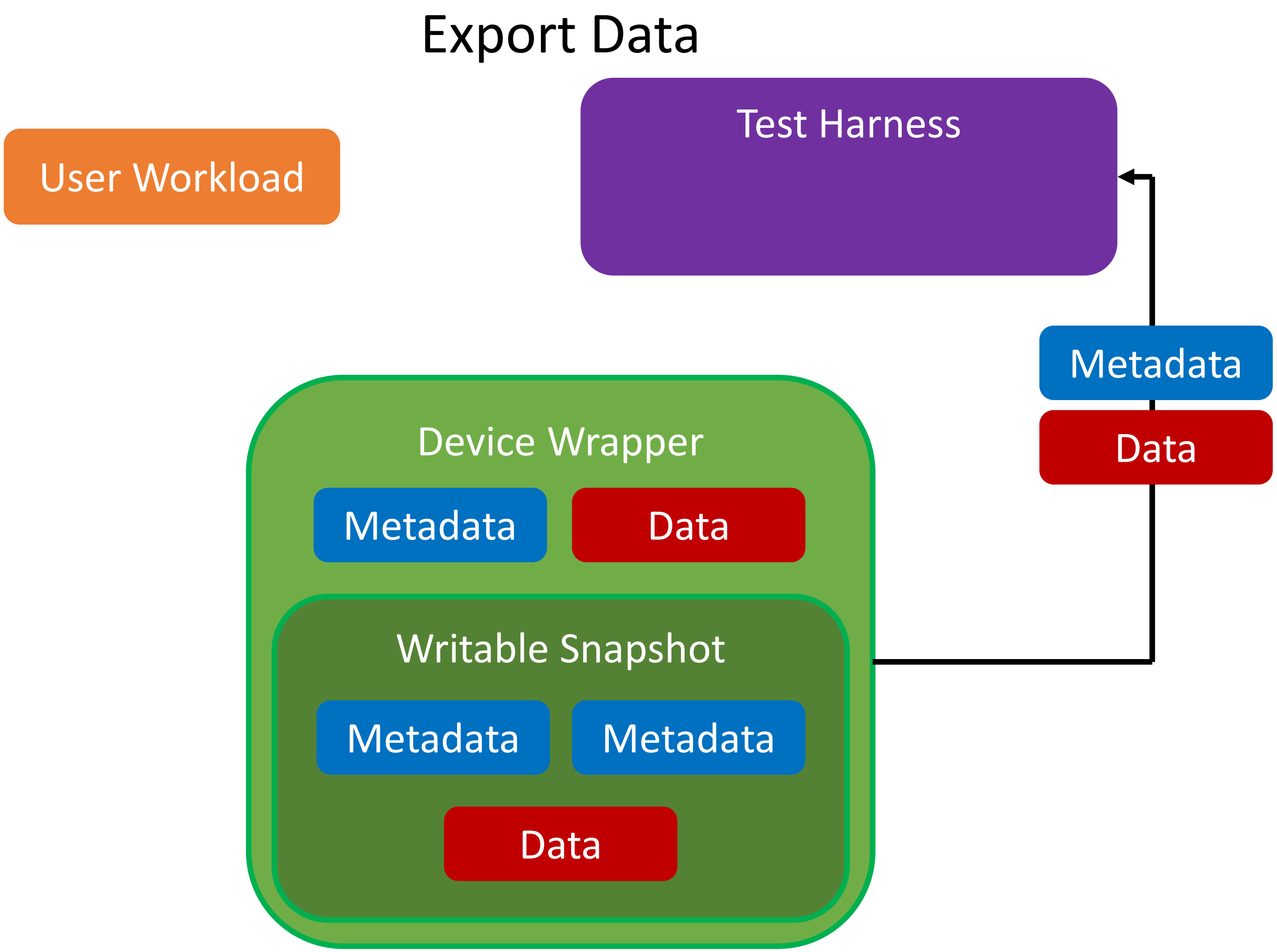
CrashMonkey In Action



CrashMonkey In Action

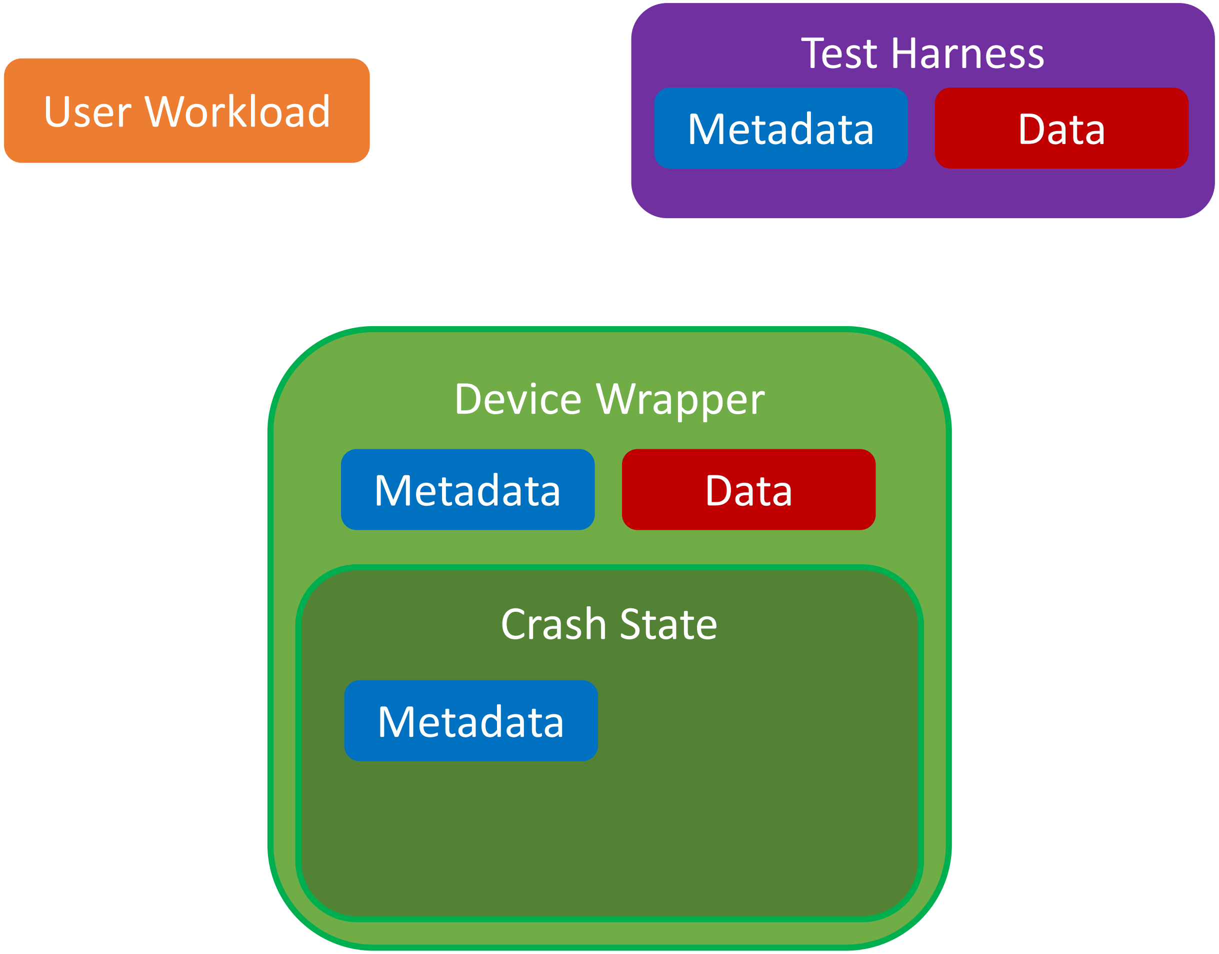


CrashMonkey In Action

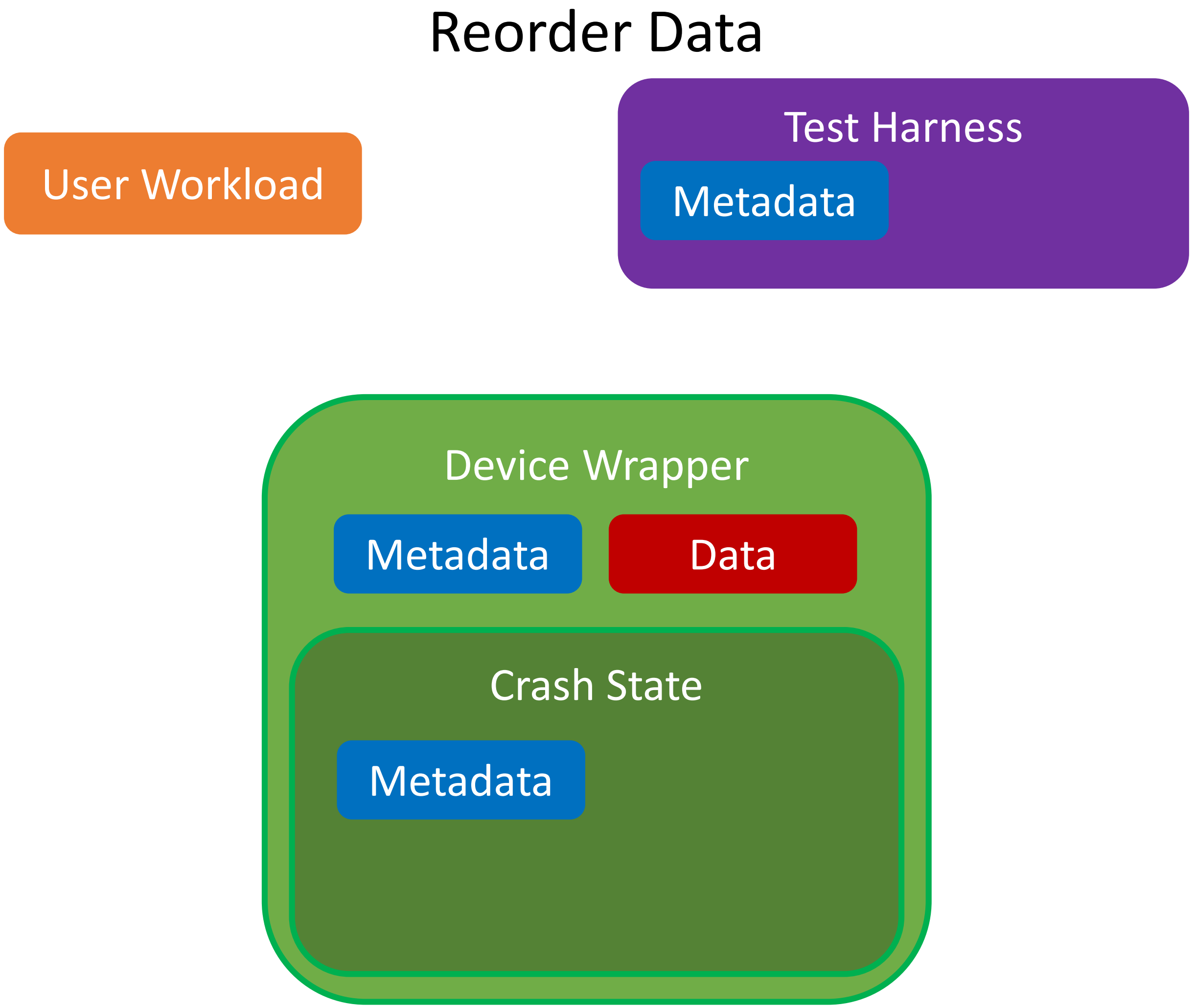


CrashMonkey In Action

Restore Snapshot

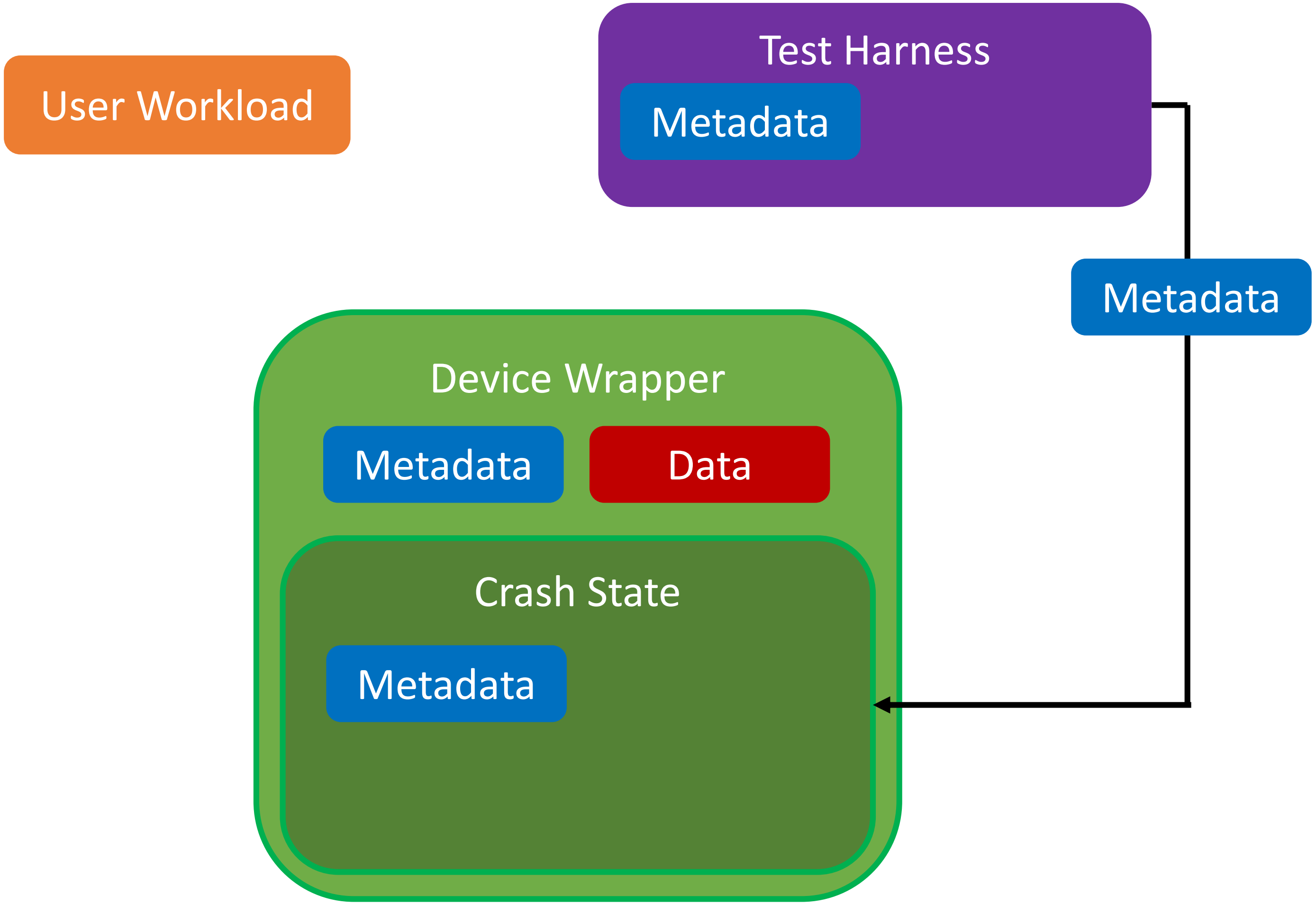


CrashMonkey In Action

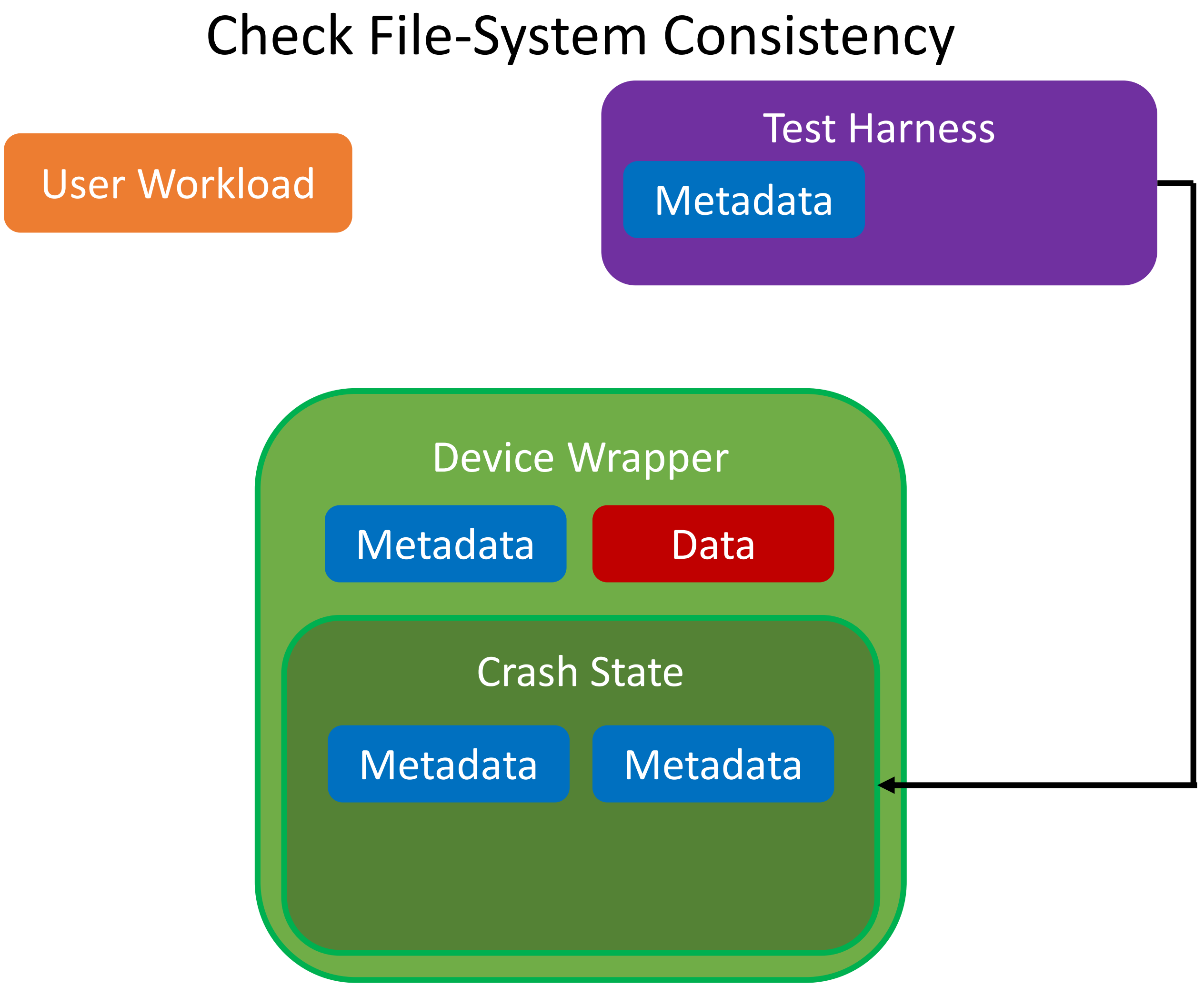


CrashMonkey In Action

Write Reordered Data to Snapshot



CrashMonkey In Action



Testing Consistency

- Different types of consistency
  - File system is inconsistent and unfixable
  - File system is consistent but garbage data
  - File system has leaked inodes but is recoverable
  - File system is consistent and data is good
- Currently run fsck on all disk states
  - Check only certain parts of file system for consistency
- Users can define checks for data consistency

Customizing CrashMonkey

- Customize algorithm to construct crash states

```
class Permuter {
 public:
  virtual void init_data(vector);
  virtual bool gen_one_state(vector);
};
```

- Customize workload:
  - Setup
  - Data writes
  - Data consistency tests

```
class BaseTestCase {
 public:
  virtual int setup();
  virtual int run();
  virtual int check_test();
};
```

Outline

- Overview
- How Consistency is Tested Today
- Linux Writes
- CrashMonkey
- Preliminary Results
- Future Plans
- Conclusion

Results So Far

- Testing 100,000 unique disk states takes ~10 minutes
  - Test creates 10 1KB files in a 10MB ext4 file system
  - Majority of time spent running fsck
- Profiling the workload takes ~1 minute
  - Happens only once per user-defined test
  - Want operations to write to disk naturally
    - `sync()` adds extra operations to those recorded
    - Must wait for writeback delay
    - Decrease delay through `/proc` file

Outline

- Overview
- How Consistency is Tested Today
- Linux Writes
- CrashMonkey
- Preliminary Results
- Future Plans
- Conclusion

The Path Ahead

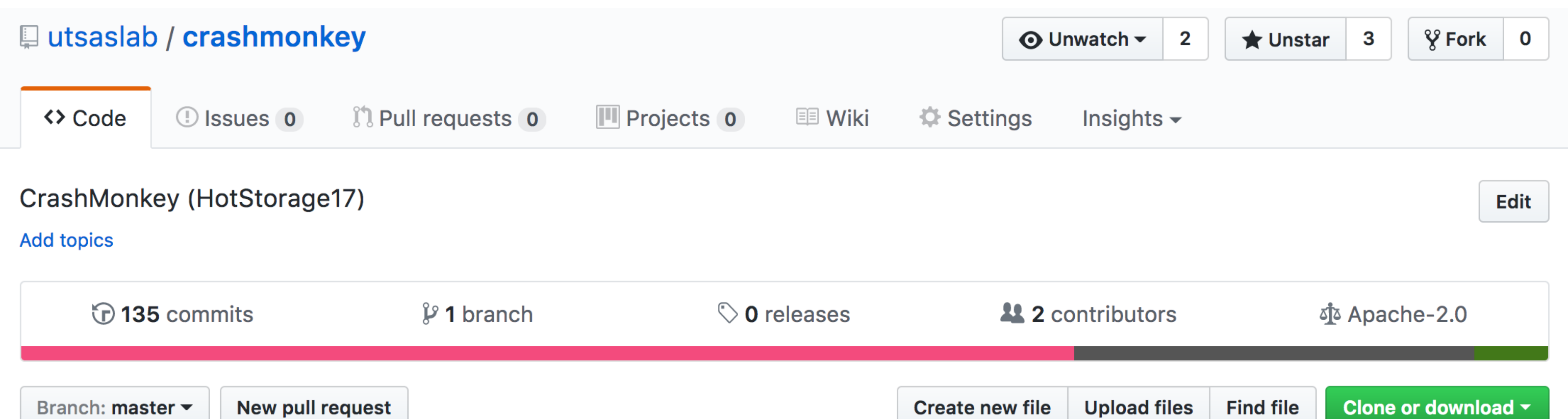
- Identify interesting crash states
  - Focus on states which have reordered metadata
  - Huge search space from which to select crash states
- Avoid testing equivalent crash states
  - Avoid generating write sequences that are equivalent
  - Generate write sequences then check for equivalence
- Parallelize tests
  - Each crash state is independent of the others
- Optimize test harness to run faster
  - Check only parts of file system for consistency

Outline

- Overview
- How Consistency is Tested Today
- Linux Writes
- CrashMonkey
- Preliminary Results
- Future Plans
- Conclusion

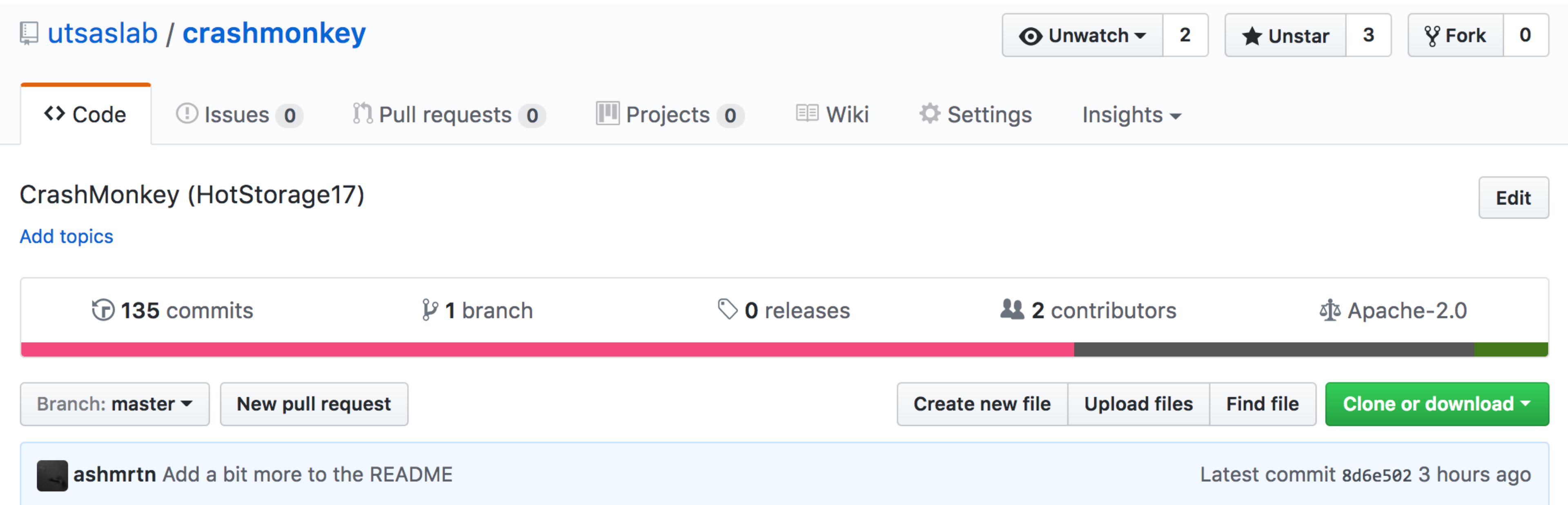
Conclusion

- Crash consistency is very important
  - Crash consistency is hard and complex to implement
  - Current crash consistency not well tested despite importance
- CrashMonkey seeks to alleviate these problems
  - Efficient, systematic,file-system agnostic
  - Work in progress
  - Code available at https://github.com/utsaslab/crashmonkey



Thank You!

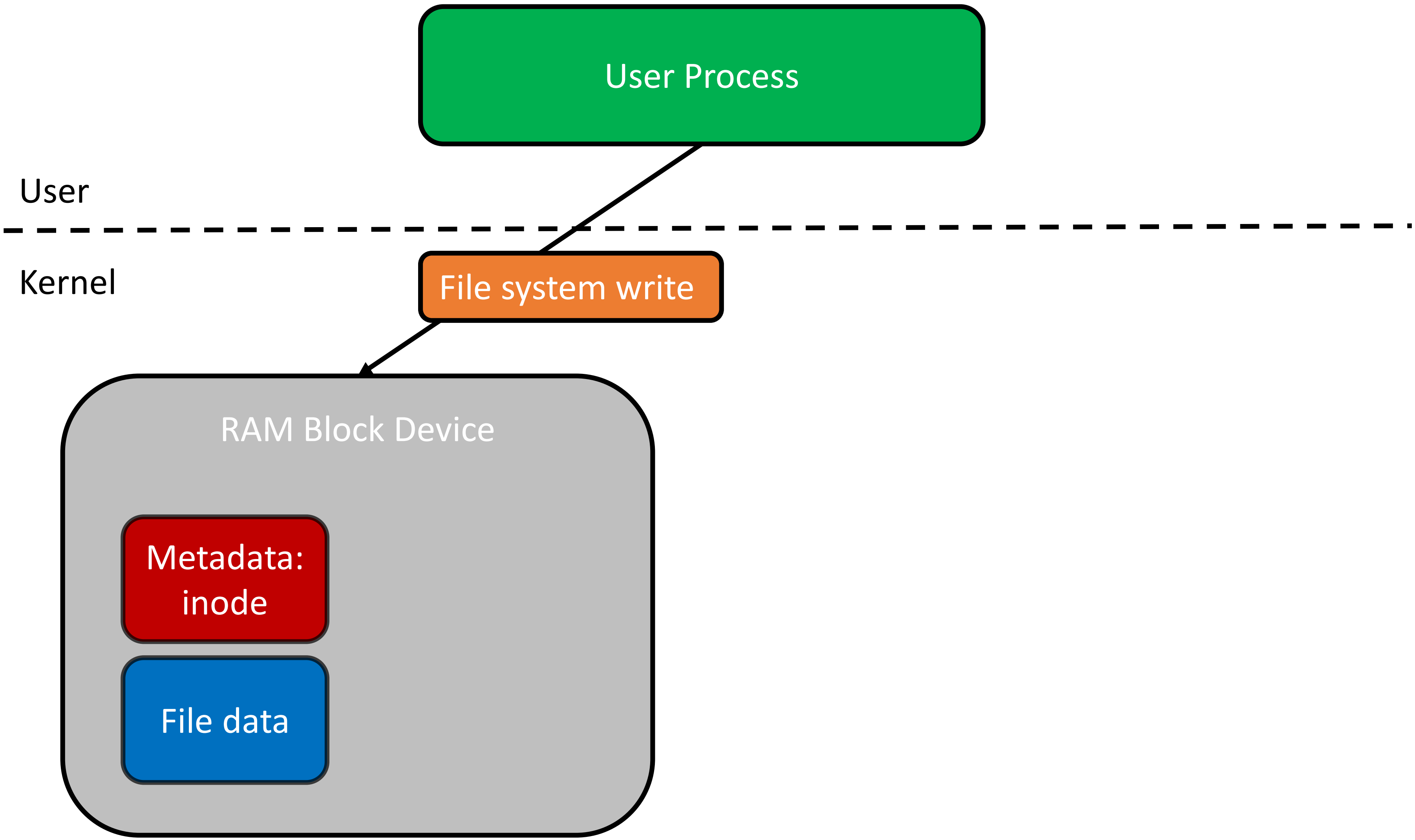
Questions?



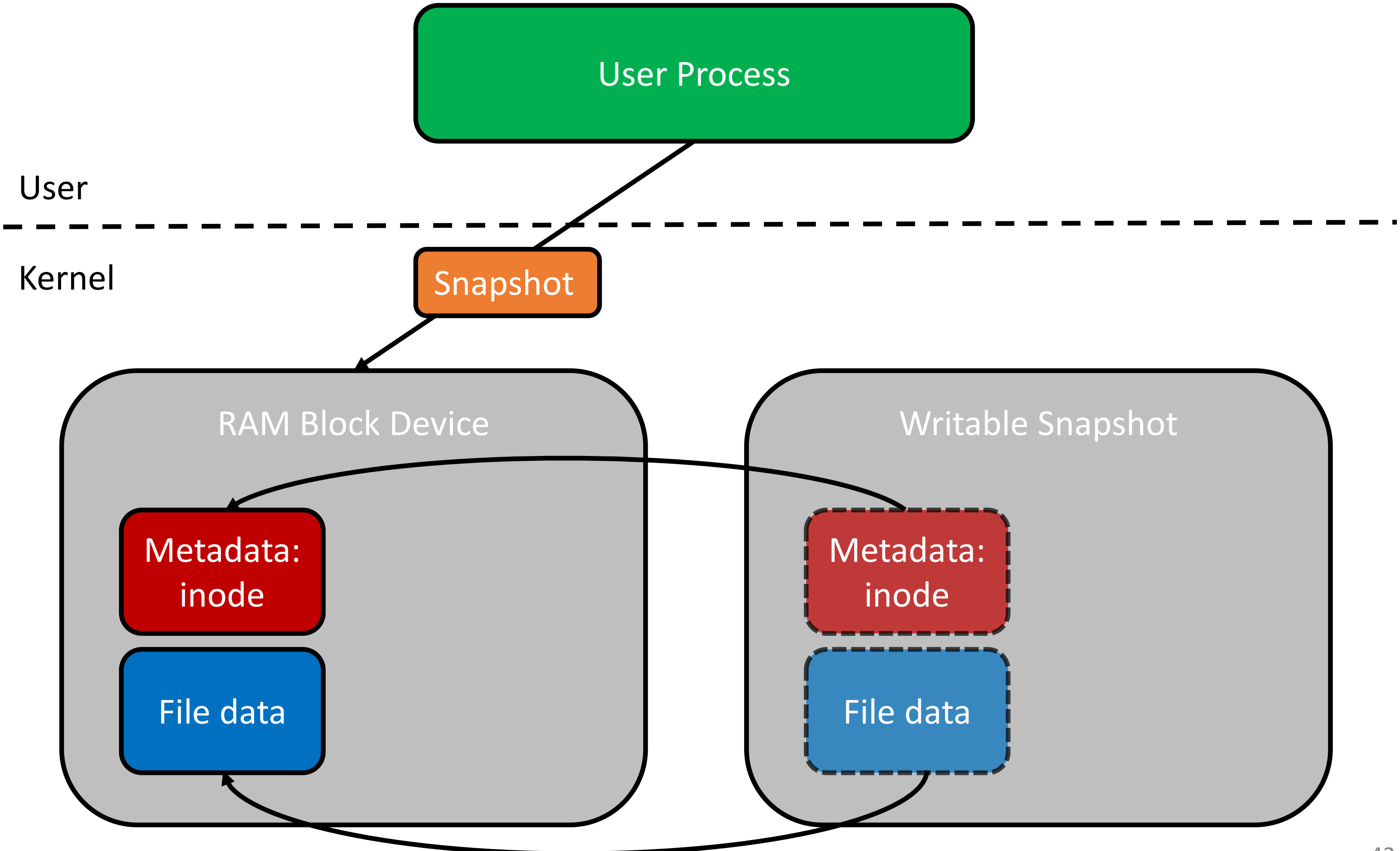
Related Work

- ALICE and BOB [Pillai *et al. OSDI’14*]
  - Very narrow scope – explore how file systems crash
    - No attempt to explore or test crash consistency
- Database Replay Framework [Zheng *et al. OSDI’14*]
  - Specifically targets databases
  - Works only on SCSI drives
  - Not open source
  - Does not allow user defined tests

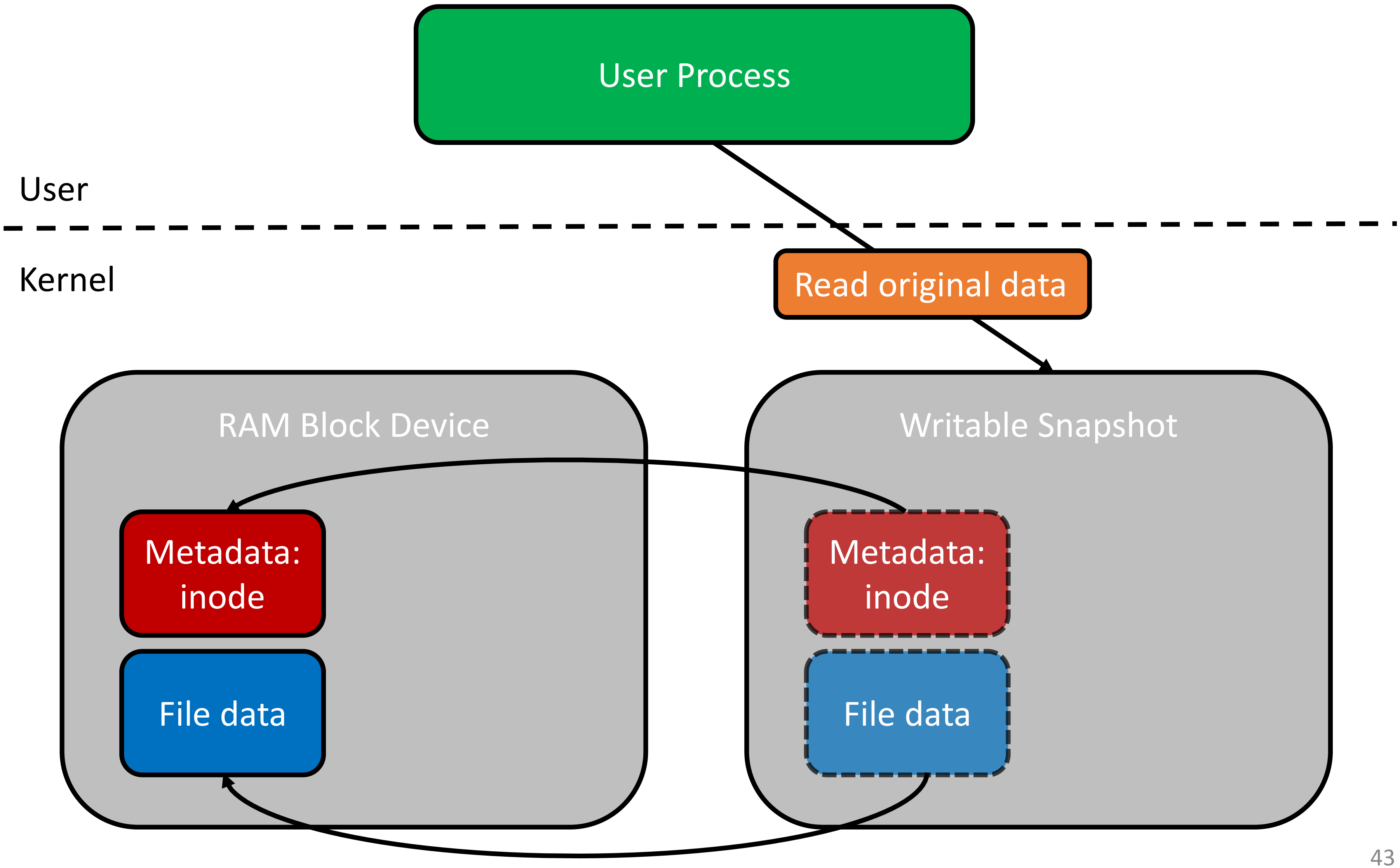
Custom RAM Block Device



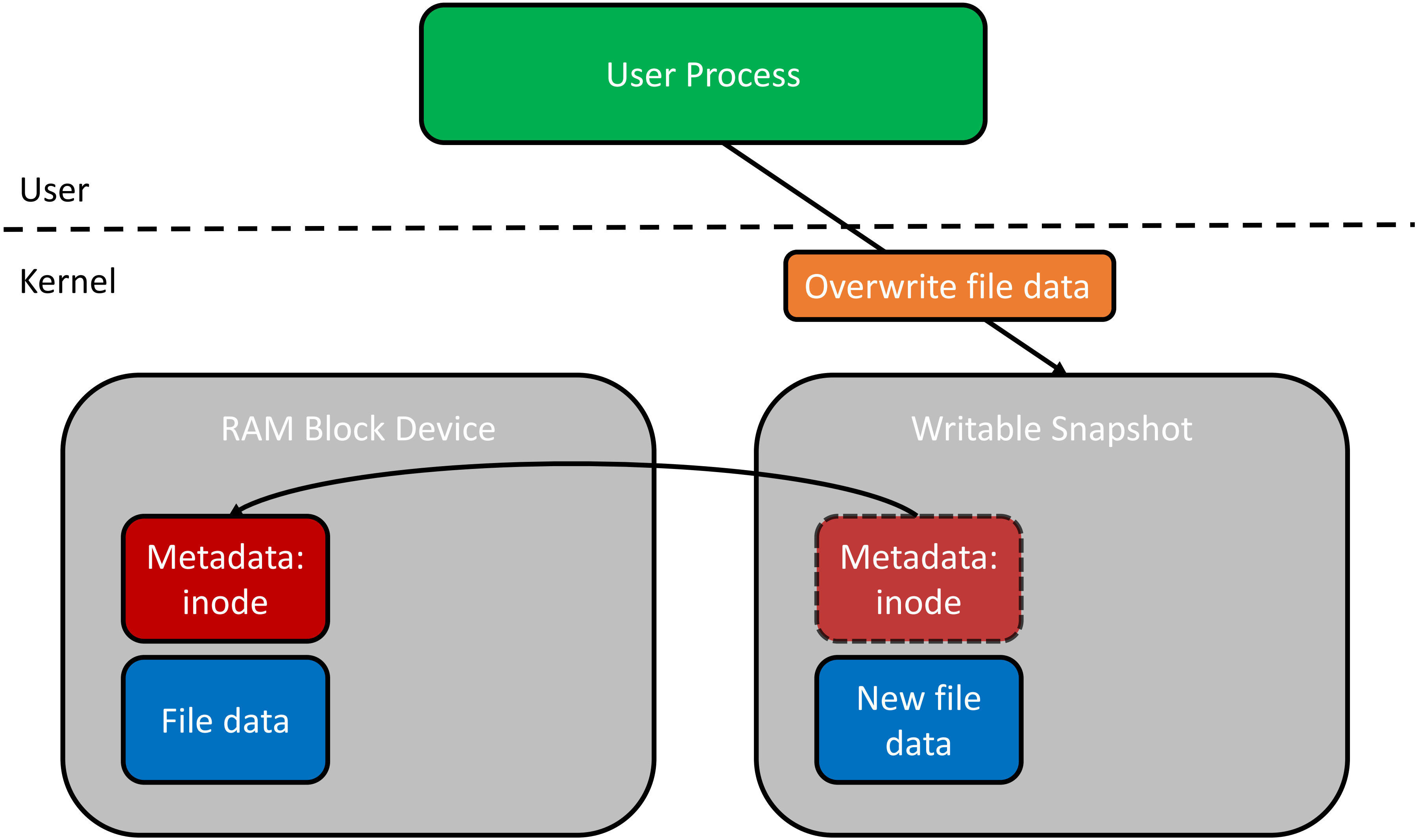
Custom RAM Block Device



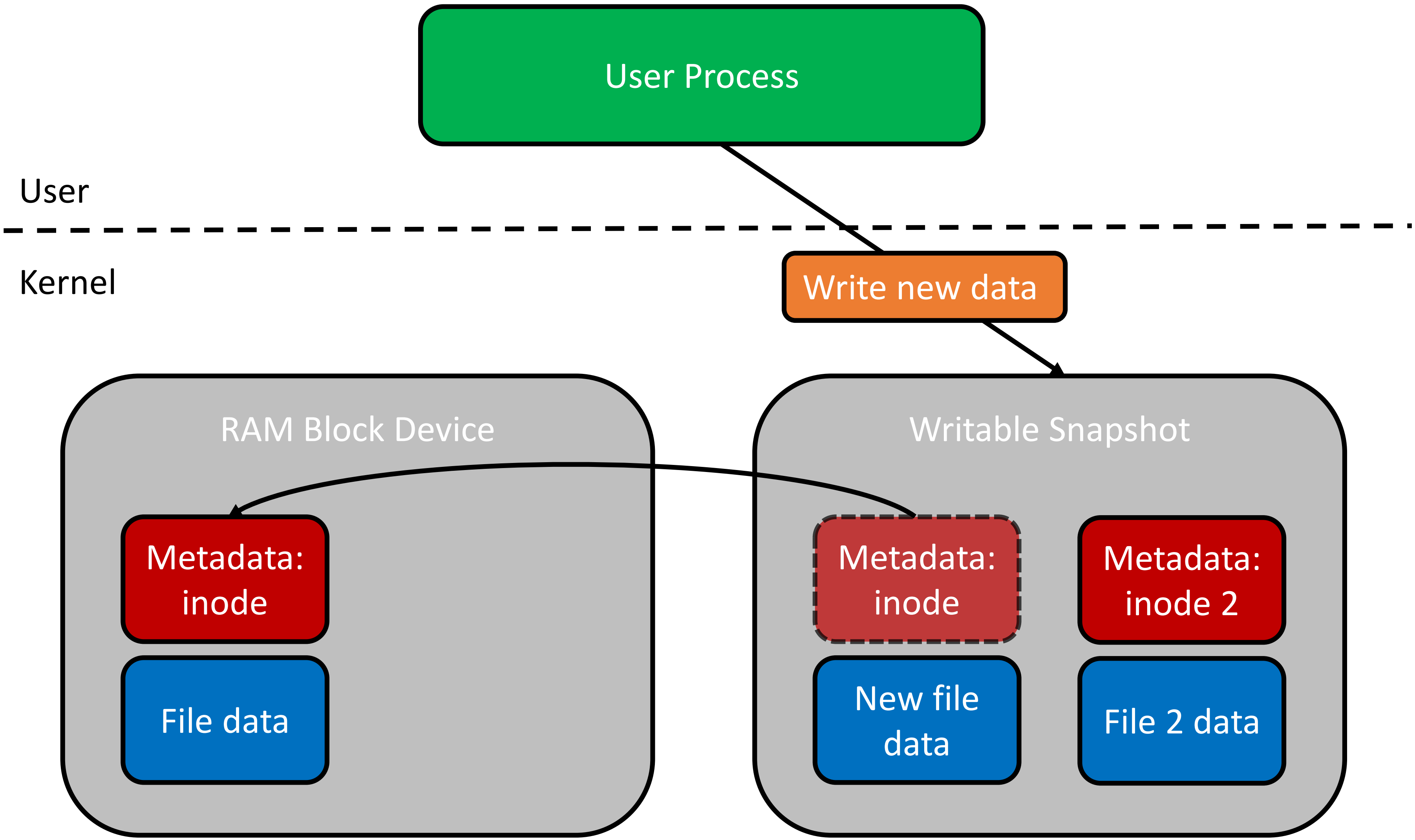
Custom RAM Block Device



Custom RAM Block Device



Custom RAM Block Device



Custom RAM Block Device