Using Model Checking to Find Serious File System Errors

Junfeng Yang, Paul Twohey, Dawson Engler
Stanford University
Madanlal Musuvathi
Microsoft Research

Authors

Junfeng

Paul

Dawson

Madan

FS Errors are Destructive

- Kernel crash, FS corruption
- Recovery code is error-prone
  - Crash at any point, must recover
- Hard to test
  - Slow reboot, reconstruction
  - many crash possibilities, hard to cover all

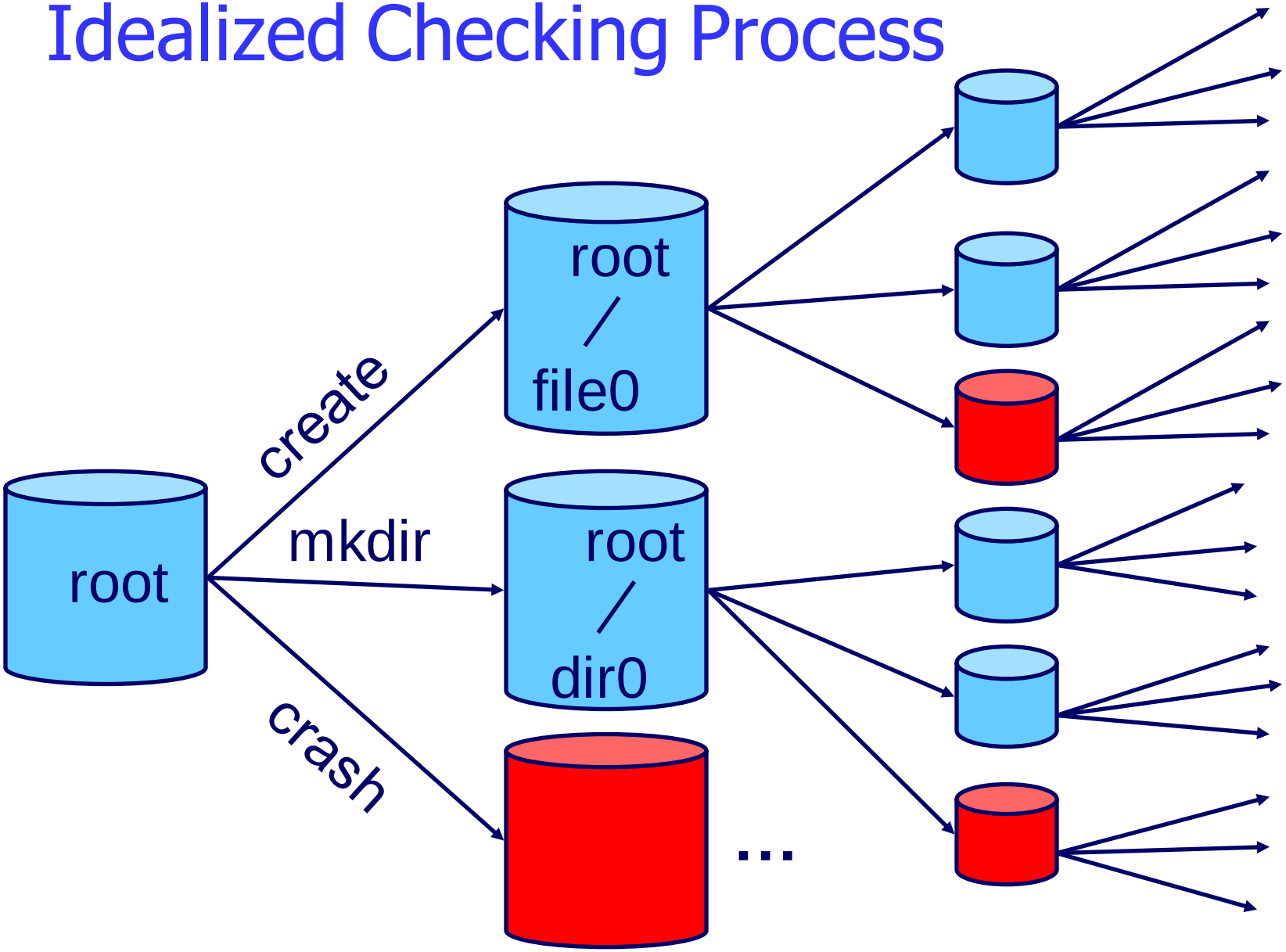
FiSC = File System Model Checker

- Leverages CMC [OSDI 02, NSDI 04]
  - Implementation-level Model Checker
- Generic and FS-specific checks
- Good at enumerating failures/crashes
- 32 Bugs on JFS, ReiserFS and ext3
  - 10 unrecoverable losses of '/', hard to get with static analysis
  - 3 security holes
  - 30 confirmed and 21 fixed quickly

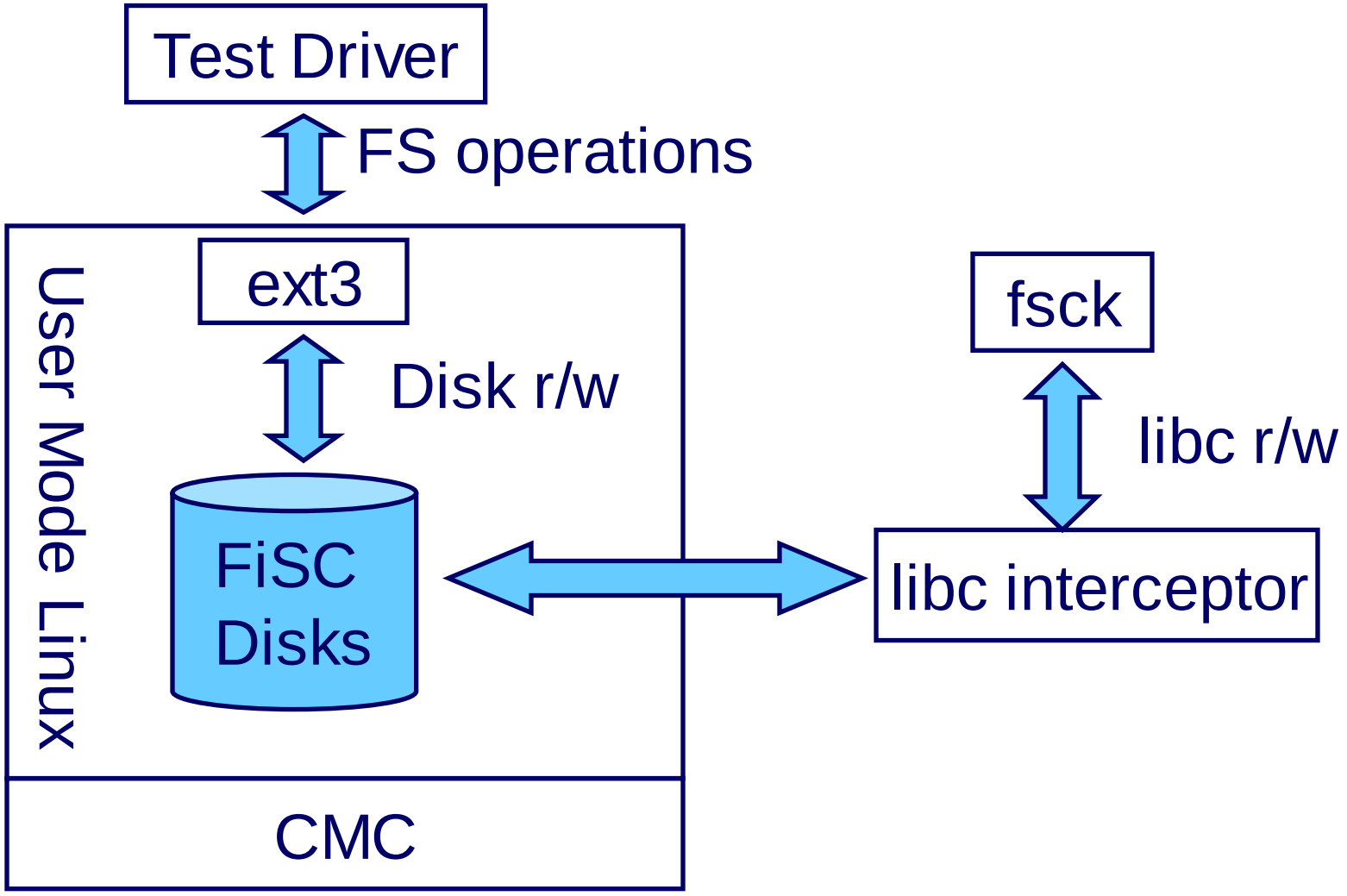
Outline

- How FiSC works
- Two consistency checks
- How to plug a file system into FiSC
- Checking crashes during recovery
- Results

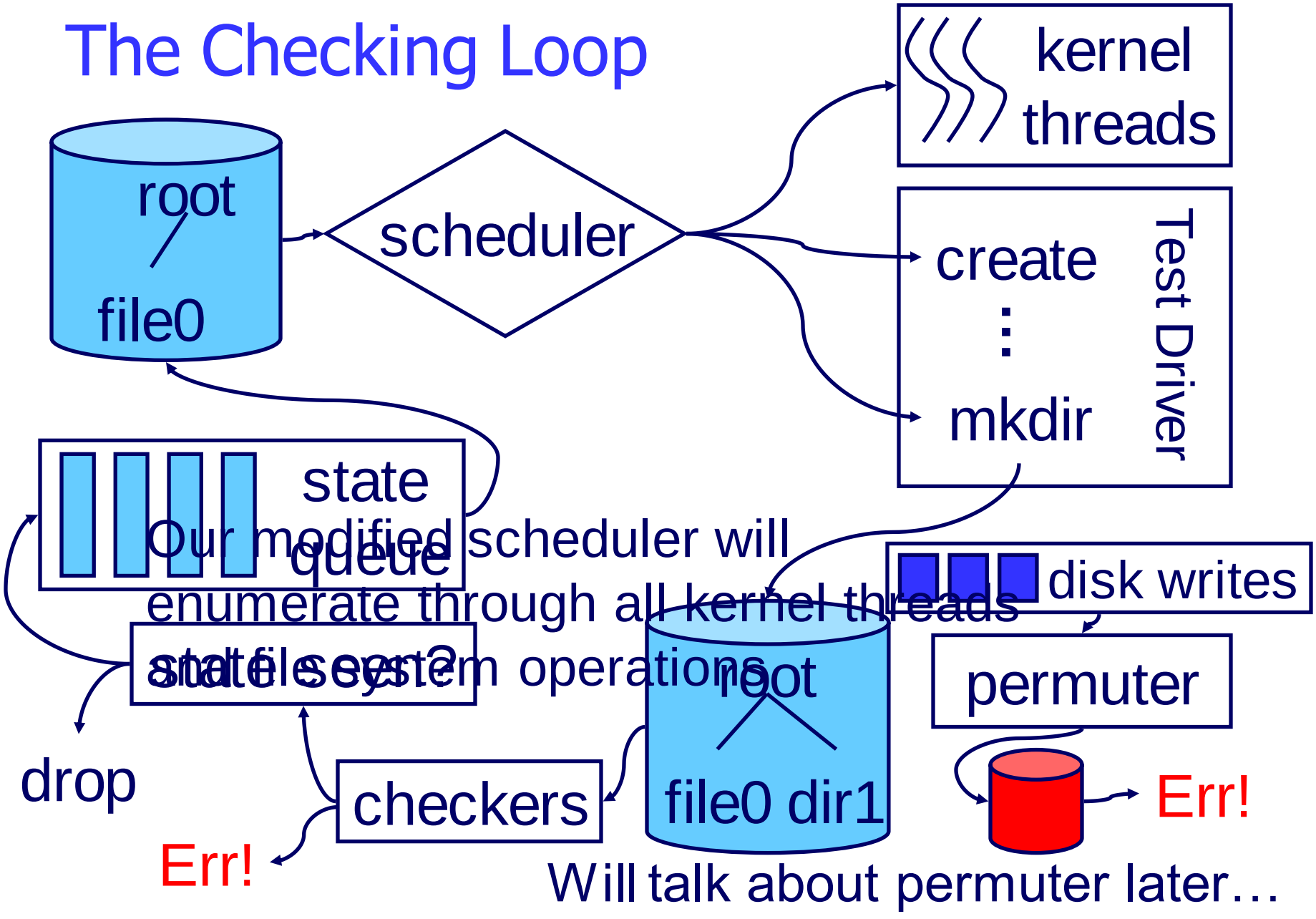
Idealized Checking Process



Galactic View of FiSC



The Checking Loop



Difference With Randomized Testing

- Randomized testing = only one possible execution
- Our approach = guided search
  - Systematic: enumerate through all actions
  - Better controlled: choose what to explore
  - Visibility: see all events
  - Repeatable: bugs are replicable

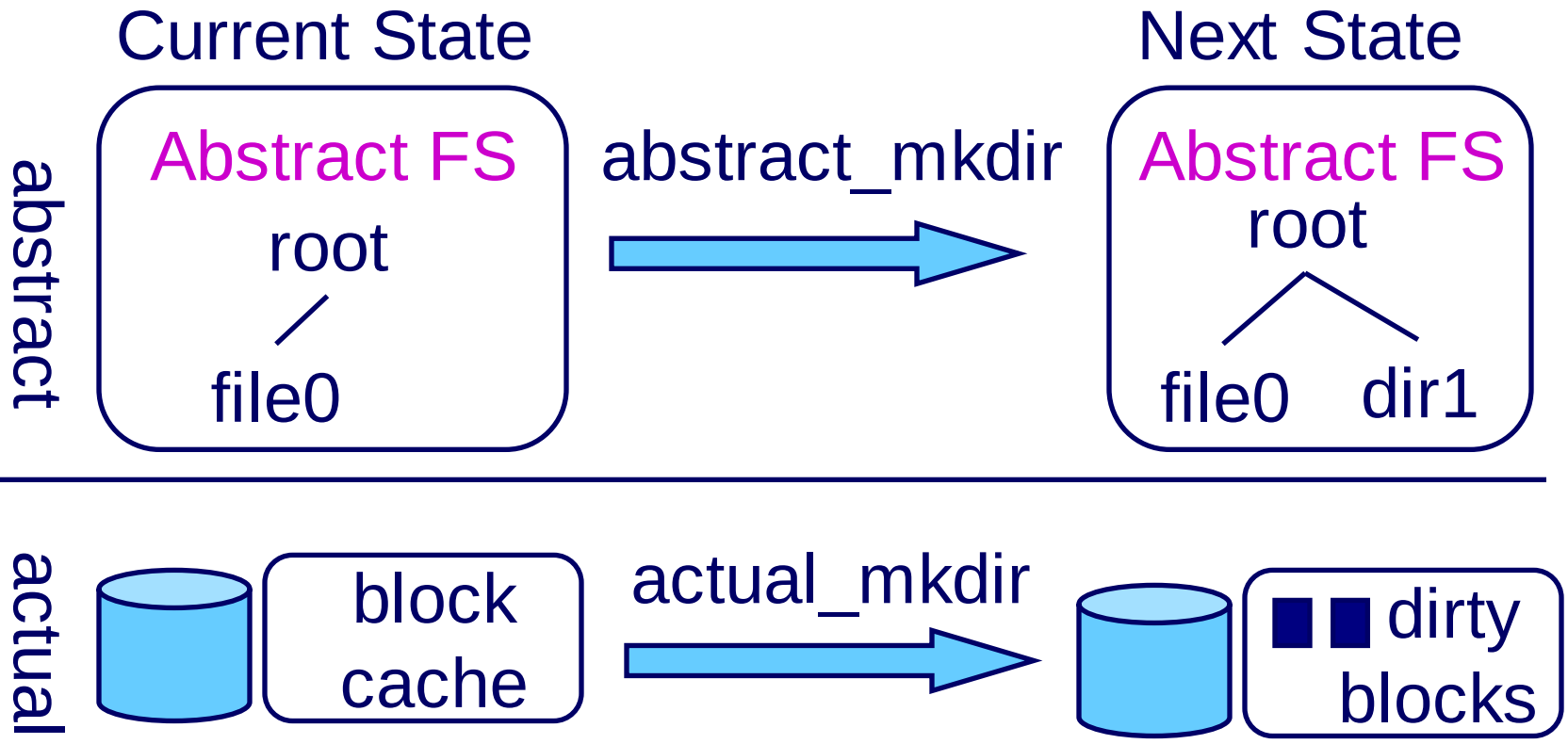
Long-lived JFS fsck Bug Fixed in 2 Days

- loss of an extent of inodes!
- 3 years old, ever since the first version!
- Caused serious data-loss
  - Dave Kleikamp (IBM JFS): "I'm sure this has bitten us before, but it's usually hard to go back and find out what causes the file system to get messed up so bad"
- Fixed in 2 days with our complete trace

Outline

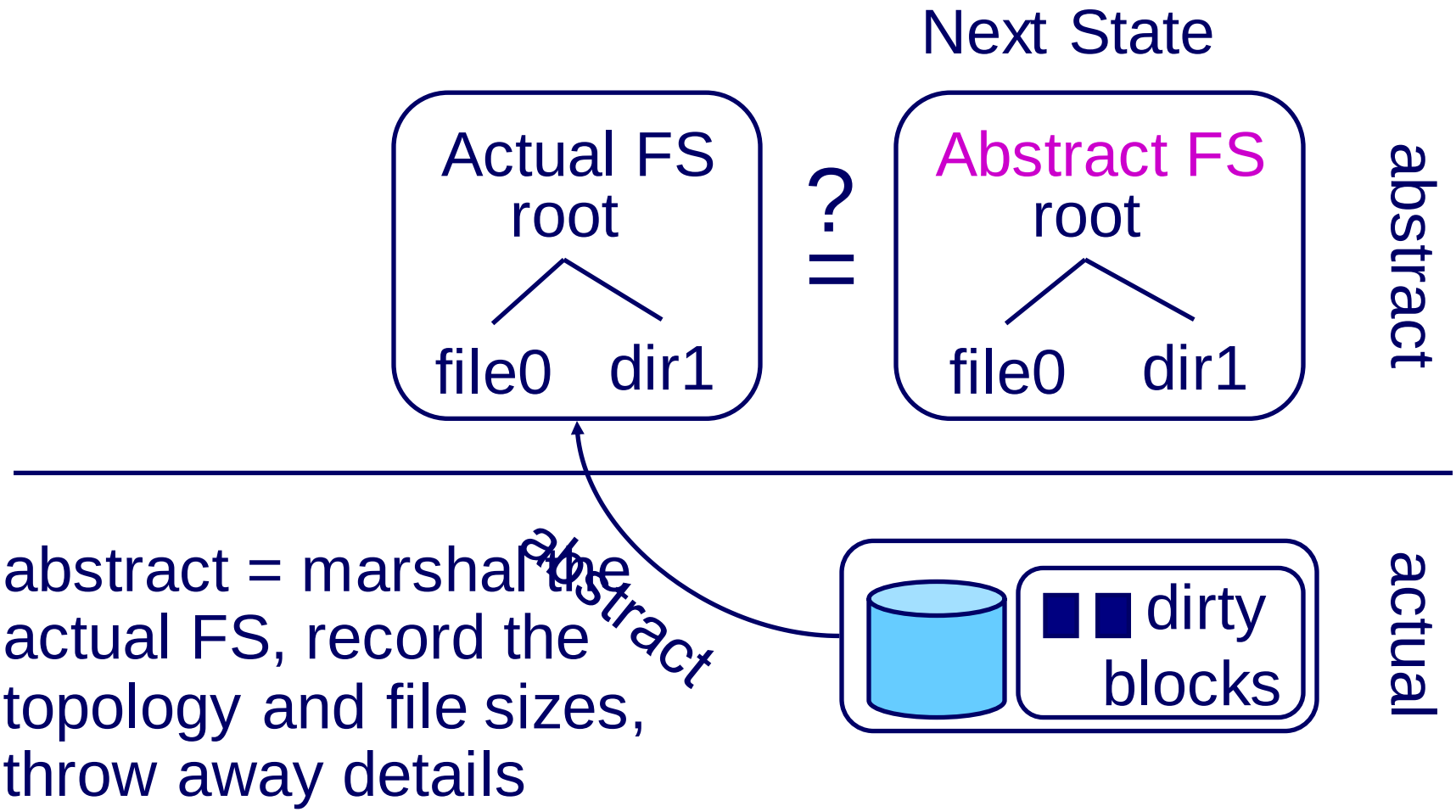
- How FiSC works
- Two consistency checks
- How to plug a file system into FiSC
- Checking crashes during recovery
- Results

Checking FS Operations are Correct



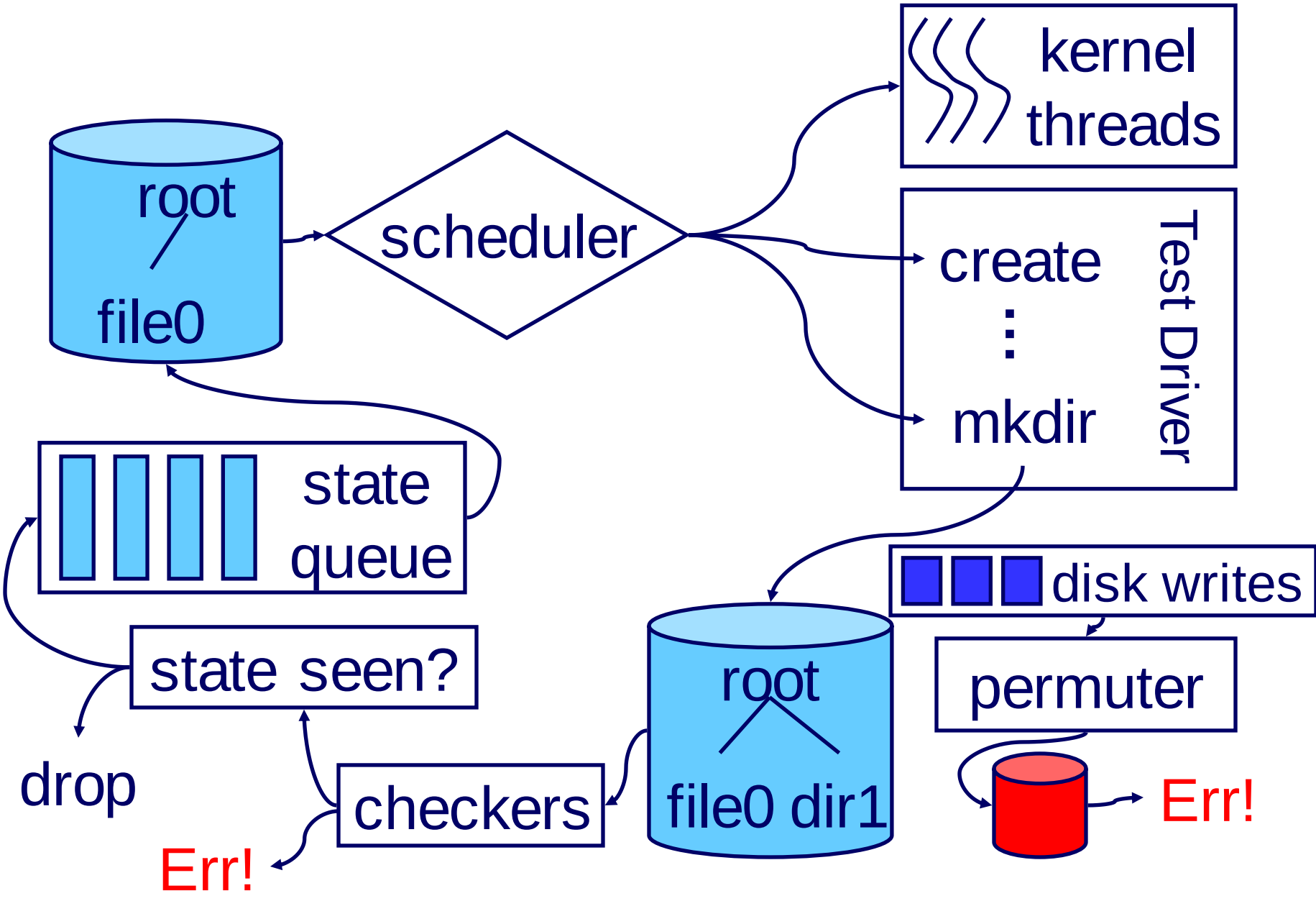
- Abstract FS: model of a file system. Currently tracks topology and file sizes. Can be extended
- Reference model, run in parallel with the actual FS

Checking FS Operations are Correct

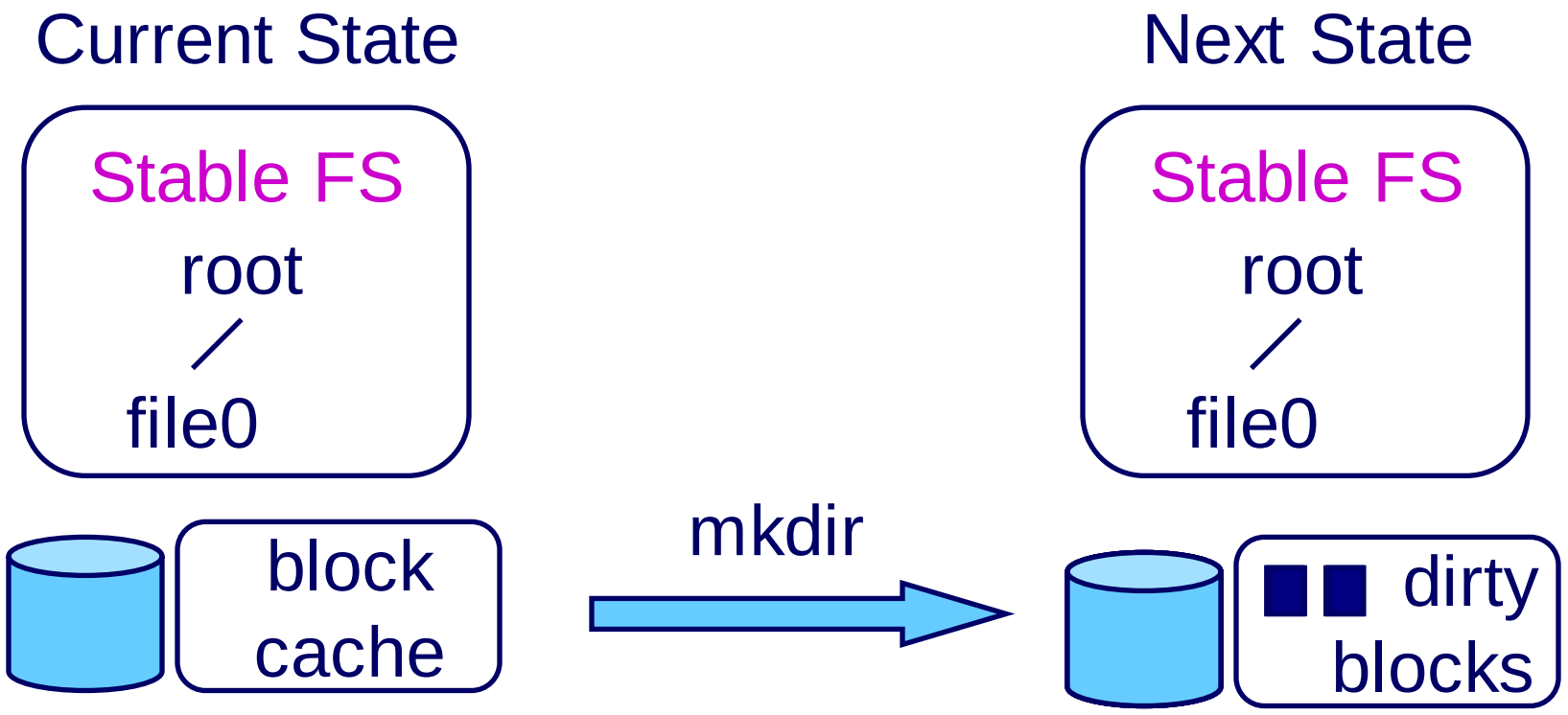


- Generic, implemented by FiSC

Permuter: Write Schedules are Recoverable

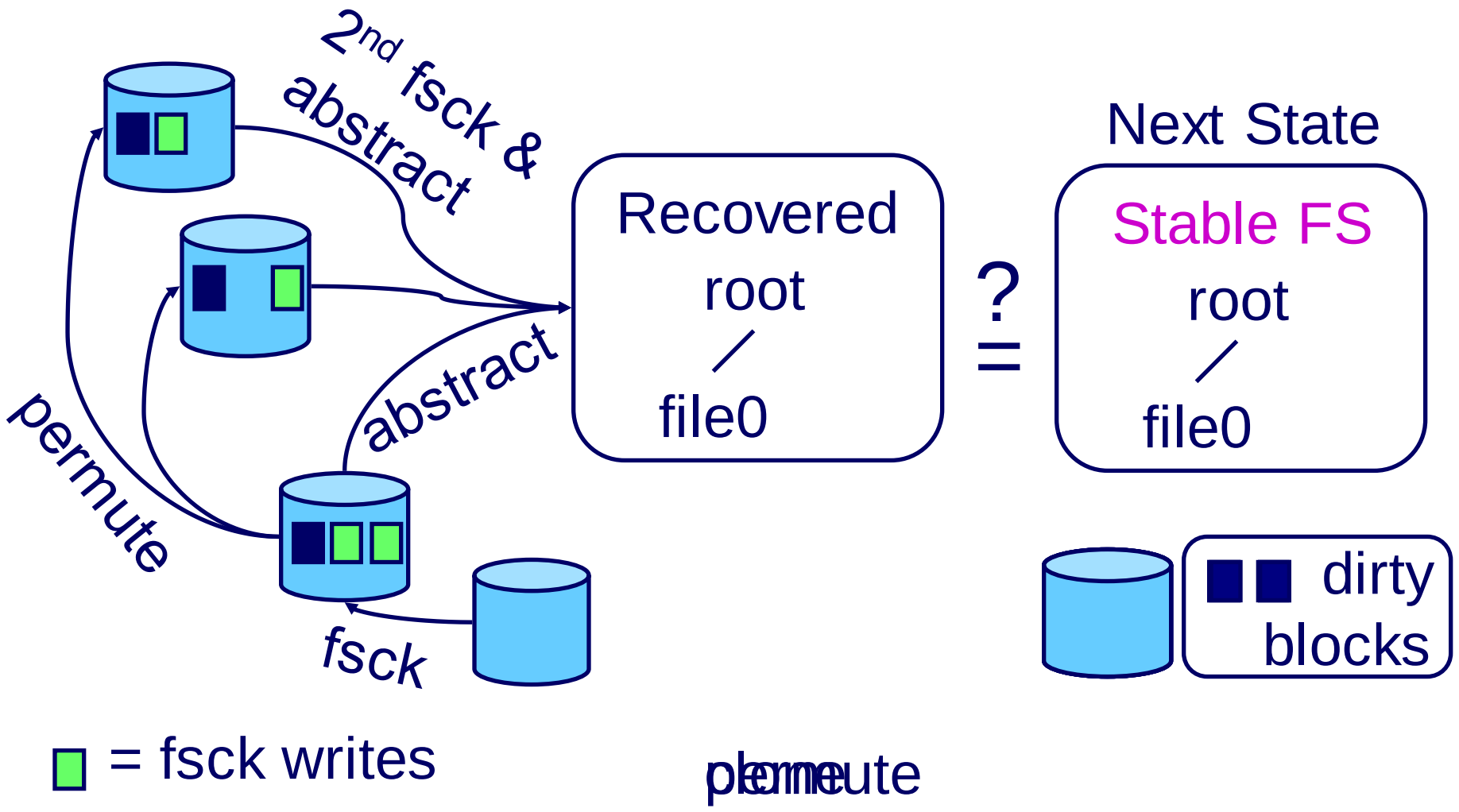


Permuter: Write Schedules are Recoverable



- Stable FS: what FS should recover to after crash
- FS-Specific, provided by FS developers

Permuter: Write Schedules are Recoverable



Outline

- How FiSC works
- Two consistency checks
- How to plug a file system into FiSC
- Checking crashes during recovery
- Results

Plugging an FS into FiSC

1. FS utilities: mkfs, fsck
2. Dirty buffers
   - Not needed if using standard system mark_dirty
3. Minimum disk and memory sizes
   - 2MB, 16 pages for ext3
4. Function to compute the Stable FS
   - Stable FS: What FS should recover to, FS-specific

.....

- Roughly 1-2 weeks for us

Stable FS Trick for Journaling FS

- Only log write can update the Stable FS
  - Log write ➔ use fsck to compute Stable FS
  - FS write ➔ fsck and abstract, compare result to Stable FS
  - FS writes cannot change Stable FS
- Log write = commit + normal log write
  - Only commit can update the Stable FS
  - If easy to recognize commit, update Stable FS on commit

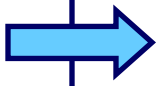
Checking More Thoroughly

- Downscale
  - Small disks. 2MB for ext3
  - Small memory. 16 pages for ext3
  - Tiny FS topology. 2-4 nodes
- Canonicalization
  - General rule: setting things to constants: e.g. inode generation #, mount count
  - Filenames. “x”, “y”, “z” == “1”, “2”, “3”

Exposing choice points

- Choice point = can abstractly do multiple actions, practically does one
- Want to explore all actions

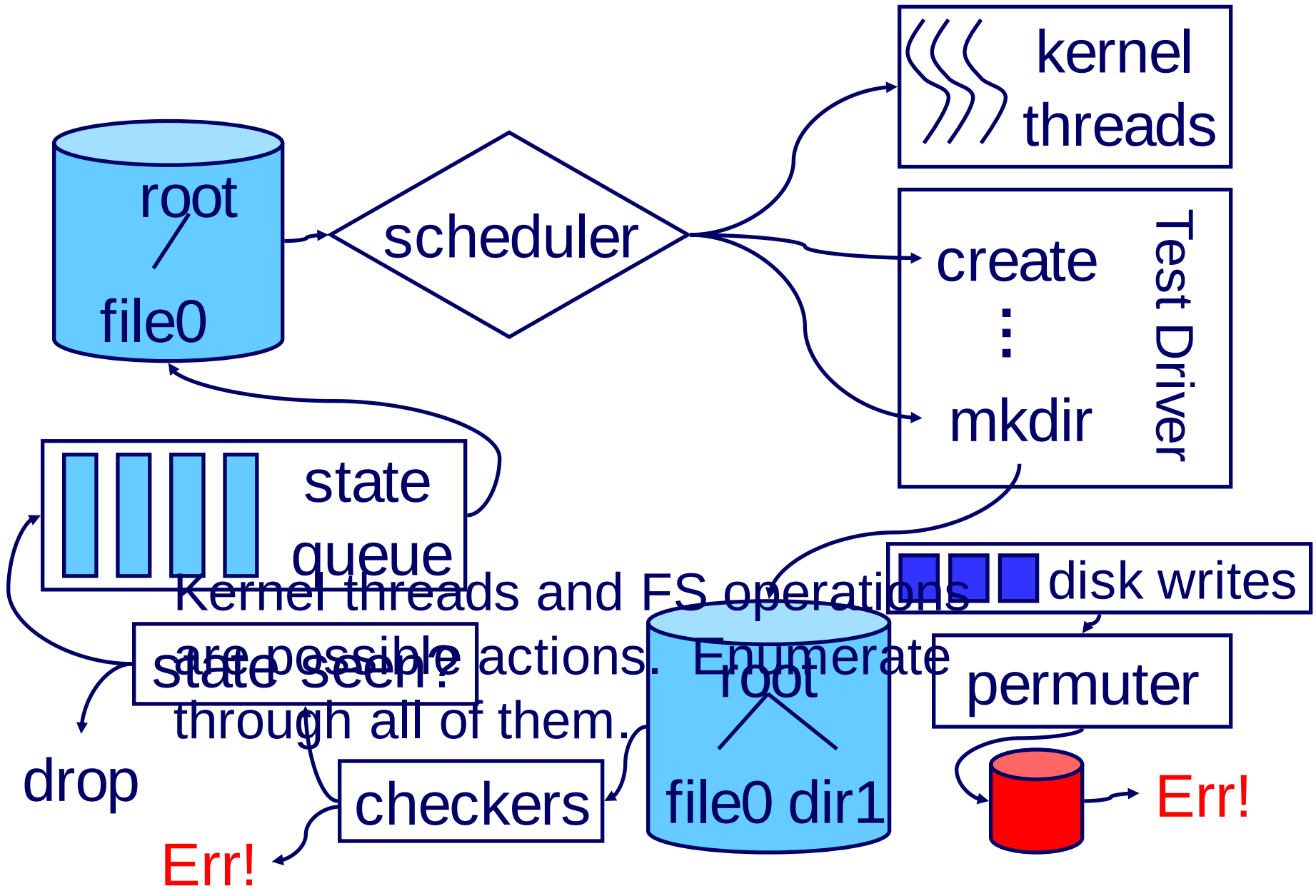
```
struct block* read_block (int i) {
    struct block *b;
    if ((b = cache_lookup(i)))
        if (fisc_choose(2) == 0)
            return b;
    return disk_read (i);
}
```



return twice,
1st time return 0,
2nd time return 1

if there are N possible actions, call fisc_choose(N) return 0, 1, …, N-1

Scheduler is a Built-in Choice Point



Outline

- How FiSC works
- Two consistency checks
- How to plug a file system into FiSC
- Checking crashes during recovery
- Results

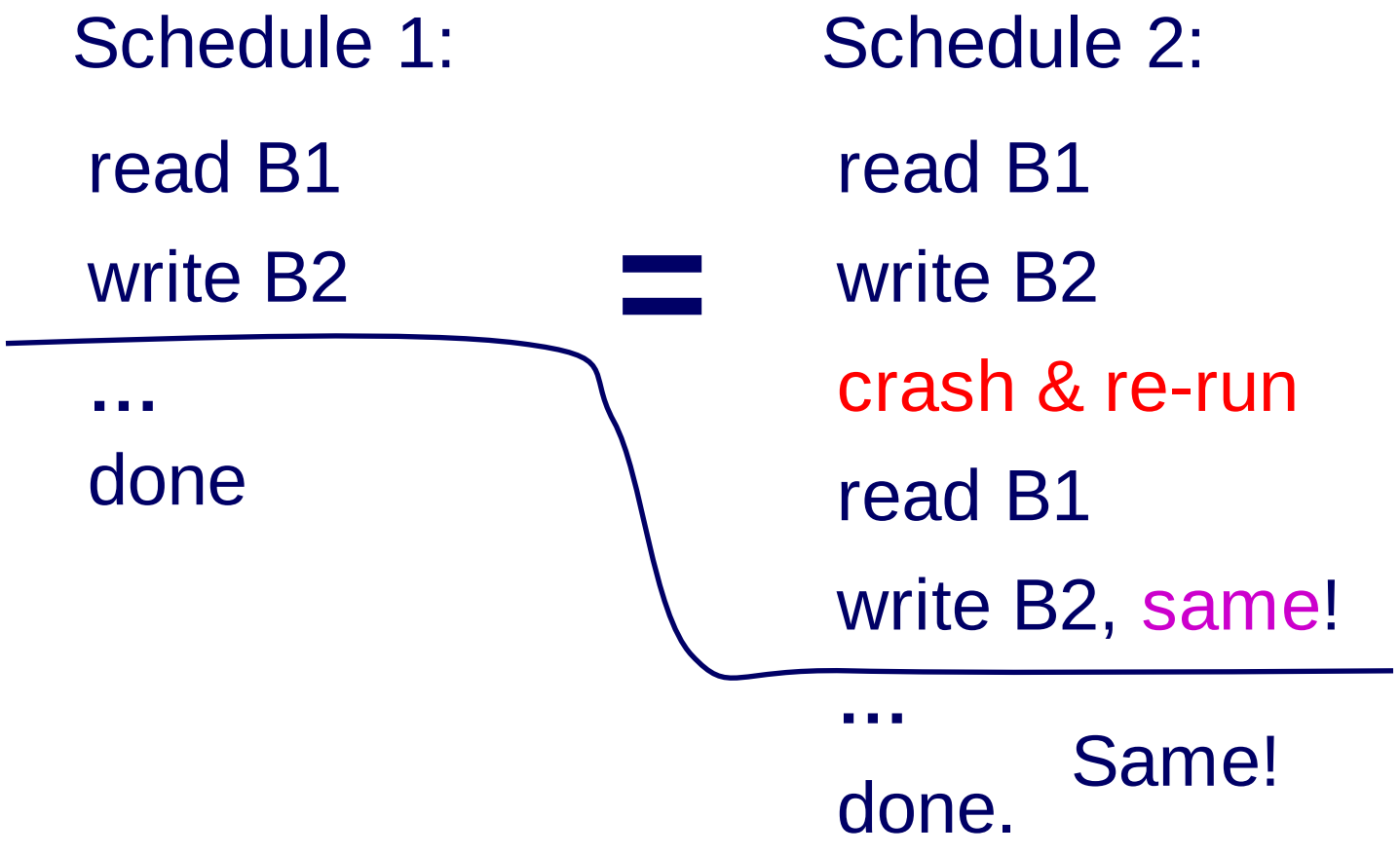
The Basic Check

- Obtain a crashed disk image D
- Run fsck, recording all writes
- Simulate a crash during recovery
  - Apply prefix to D
  - Re-run fsck
  - Compare to Stable FS
- Repeat until all the prefixes are tried
- Effective☺, Speed☹ (redundant crashes)

Assume: fsck is Deterministic

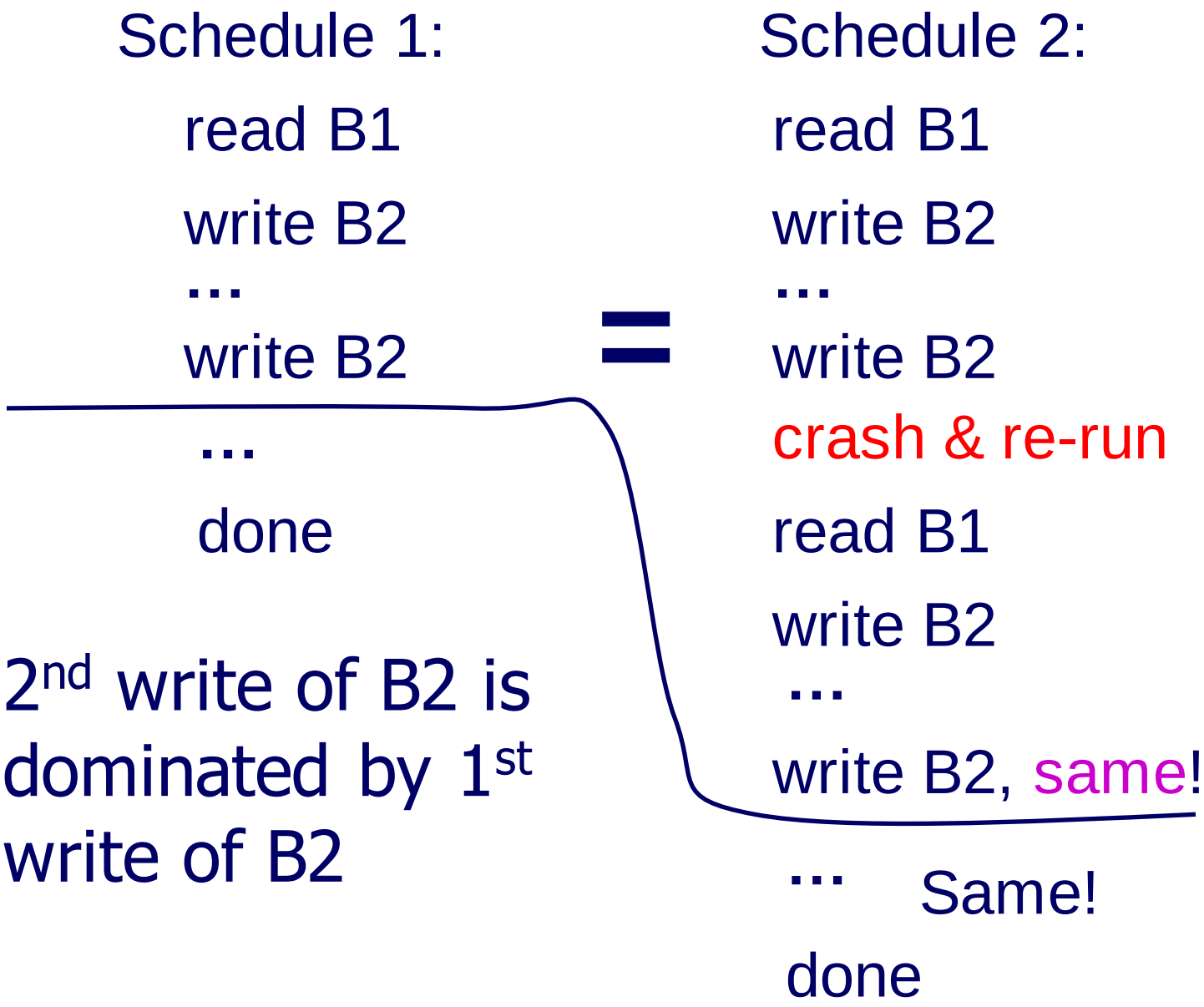
- Same inputs ➔ same outputs
  - Inputs = disk reads , outputs = writes
- Is crash after a write redundant?
  - A write doesn't change prior reads ➔ 2nd fsck computes the same write ➔ redundant crash, can be optimized away
- More optimizations in paper
  - Obvious: cache fsck results

Equivalent: Write But No Read



- No read of B2 prior to write of B2

Equivalent: Dominated Write



- 2nd write of B2 is dominated by 1st write of B2

Results

Error Type	VFS	ext2	ext3	JFS	Reiser	total
Data loss	N/A	N/A	1	8	1	10
False clean	N/A	N/A	1	1		2
Security		2	2	1		3 + 2
Crashes	1			10	1	12
Other	1		1	1		3
Total	2	2	5	21	2	32

32 in total, 21 fixed, 9 of the remaining 11 confirmed

Recovery Write Ordering Bugs

- Under Normal operation:
  - Changes must first be flushed to log before they can reach the actual FS
- All FS seem to get this right
- During Recovery:
  - Changes must first be flushed to the actual FS before the log can be cleared
- Found this type of bug in all FS, total 5

ext3 Recovery Bug

```
recover_ext3_journal(…) {
    // …
    retval = -journal_recover(journal)
    // …
    // clear the journal
    e2fsck_journal_release(…)
    // …
}
```

```
journal_recover(…) {
    // replay the journal
    //…
    // sync modifications to disk
    fsync_no_super(…)
}
```

```
// Error! Empty macro, doesn't sync data!
#define fsync_no_super(dev) do {} while (0)
```

- Code was directly adapted from the kernel
- But, fsync_no_super was defined as NOP !

Conclusion

- FiSC, a FS model checker
  - On average 1-2 weeks to plug in an FS
  - Checked JFS, ReiserFS and ext3
  - Serious data-loss bugs in all, 10 in total
- Model Checking worked well
  - Can crash everywhere. Must always be recoverable.
  - Systematic
- Future work: anything that must handle failure correctly, always
  - Raid, databases, consensus algorithms…