EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors

Junfeng Yang, Can Sar, Dawson Engler
Stanford University

Why check storage systems?

- Storage system errors are among the worst
  - kernel panic, data loss and corruption
- Complicated code, hard to get right
  - Simultaneously worry about speed, failures and crashes
- Hard to comprehensively test for failures, crashes

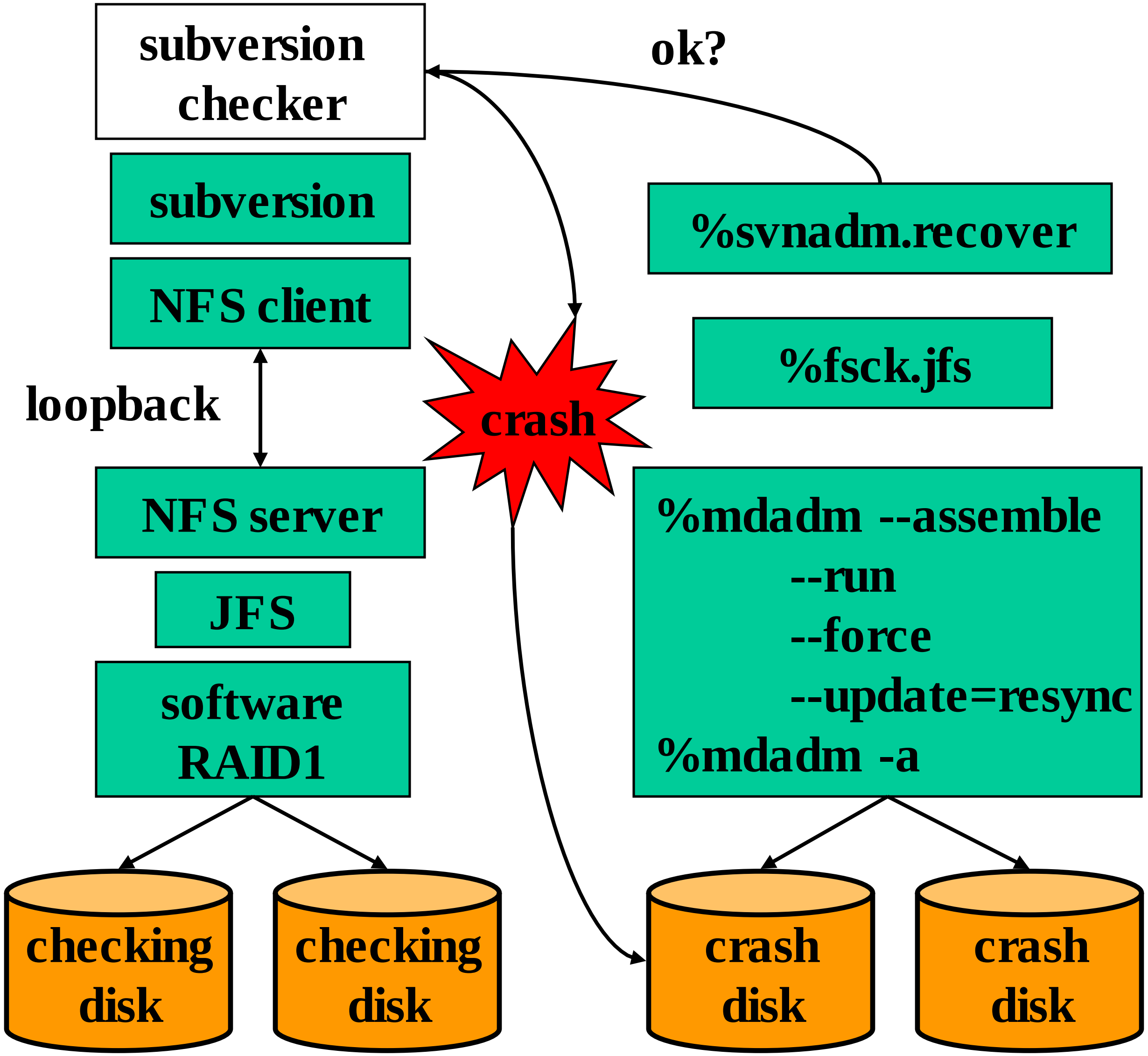
Goal: ***comprehensively*** check ***many*** storage systems with ***little*** work

EXPLODE summary

- Comprehensive: uses ideas from model checking
- Fast, easy
  - Check new storage system: 200 lines of C++ code
  - Port to new OS: 1 device driver + optional instrumentation
- General, real: check live systems.
  - Can run (on Linux, BSD), can check, even w/o source code
- Effective
  - checked 10 Linux FS, 3 version control software, Berkeley DB, Linux RAID, NFS, VMware GSX 3.2/Linux
  - Bugs in all, 36 in total, mostly data loss
- Subsumes our old work FiSC [OSDI 2004]

Checking complicated stacks

- All real
- Stack of storage systems
  - subversion: an open-source version control software
- User-written checker on top
- Recovery tools run after EXPLODE-simulated crashes



Outline

➡ Core idea

- Checking interface
- Implementation
- Results
- Related work, conclusion and future work

Core idea: explore all choices

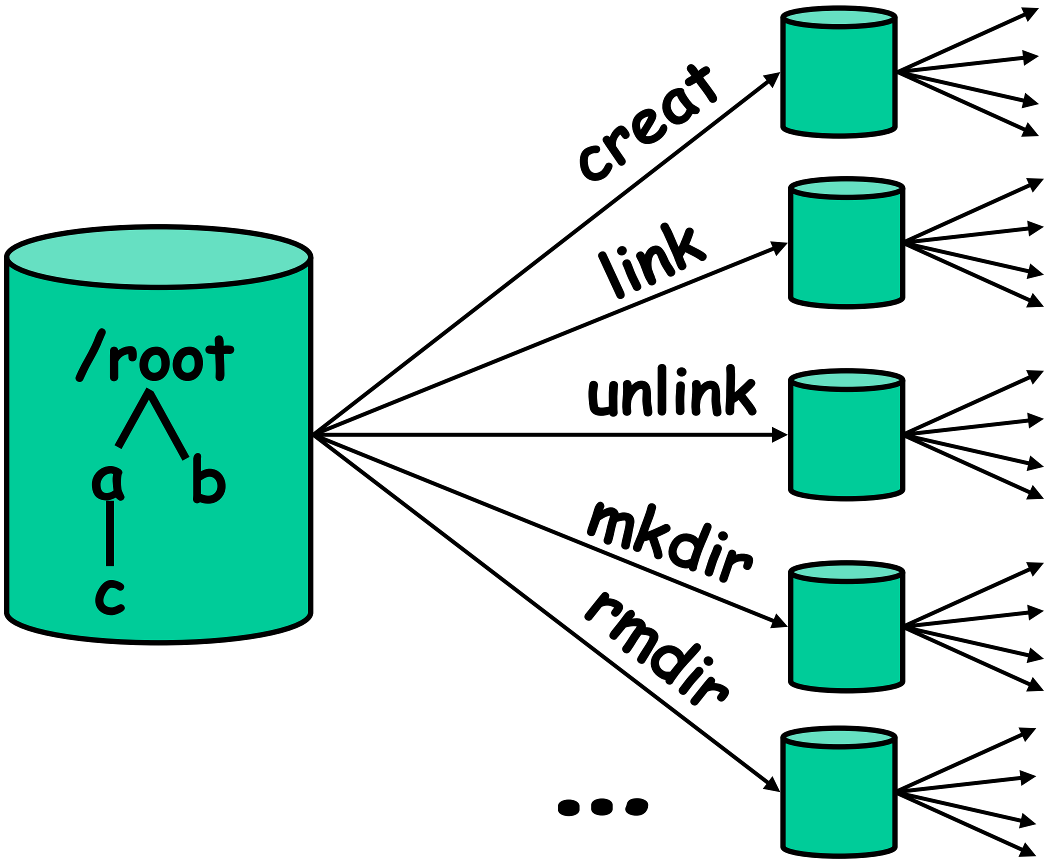
- Bugs are often triggered by corner cases

- How to find: drive execution down to these tricky corner cases

When execution reaches a point in program that can do one of N different actions, fork execution and in first child do first action, in second do second, etc.

External choices

- Fork and do every possible operation



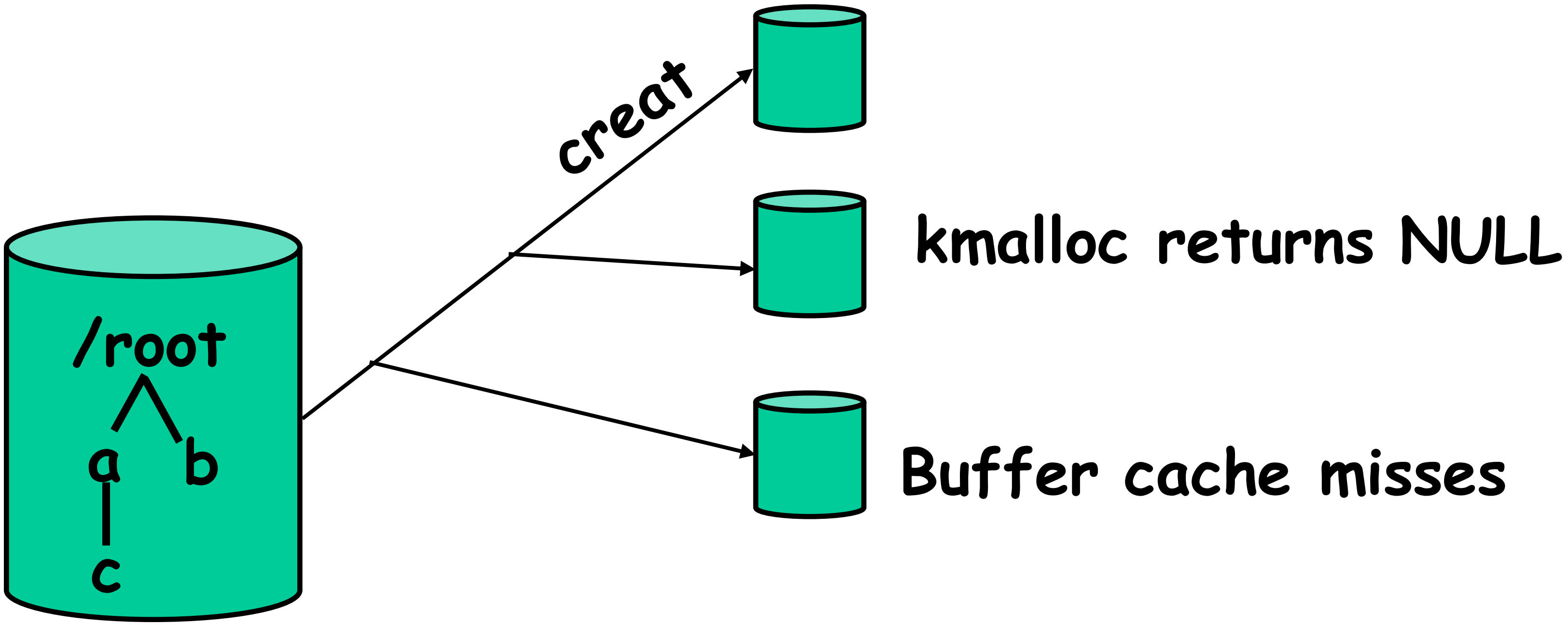
Users write code to check FS valid. EXPLODE "amplifies" the check

Explore generated states as well

Speed hack: hash states, discard if seen

Internal choices

- Fork and explore all internal choices



How to expose choices

- To explore N-choice point, users instrument code using choose(N)

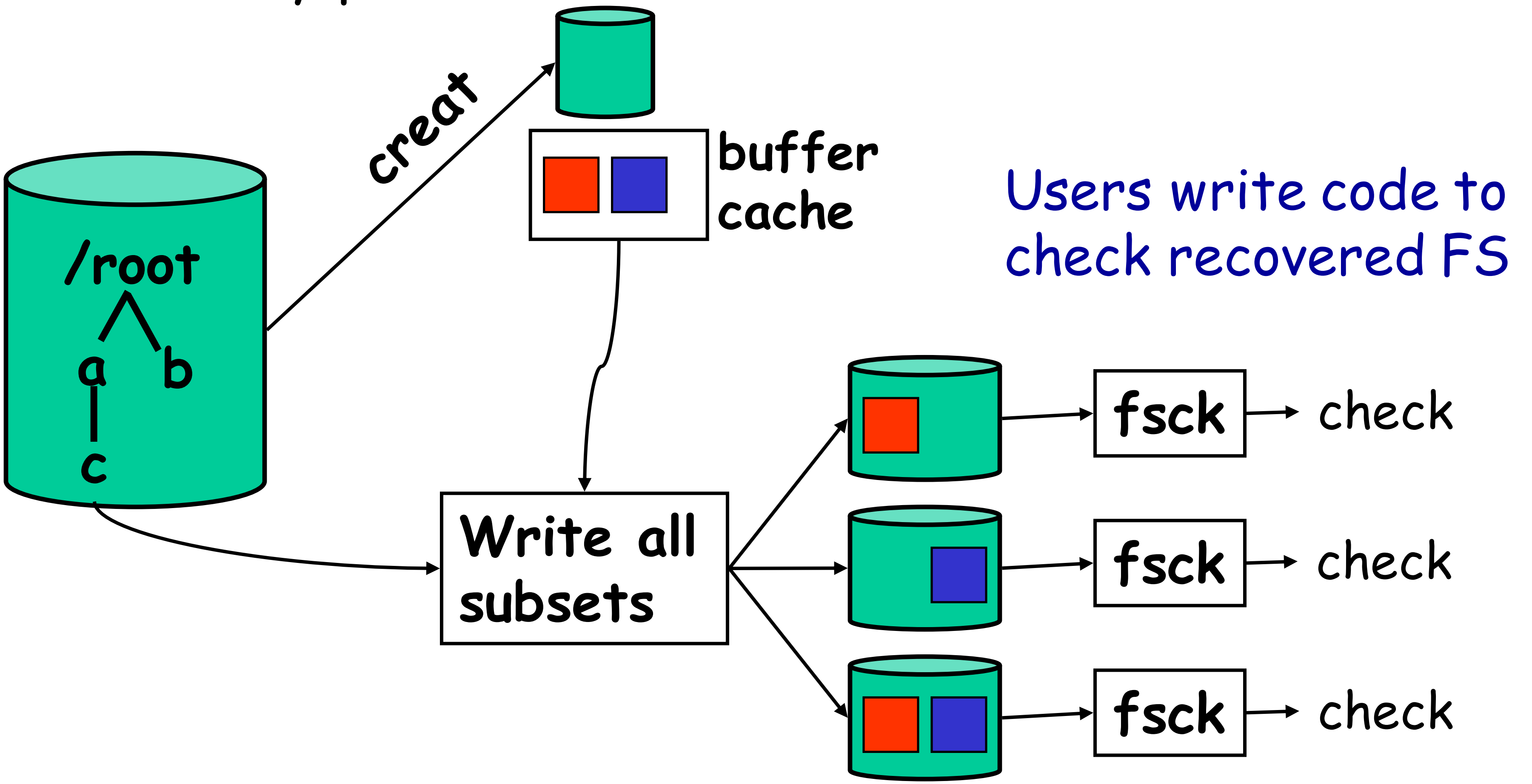
- choose(N): N-way fork, return K in K'th kid

```
void* kmalloc(size s) {
  if(choose(2) == 0)
     return NULL;
  …  // normal memory allocation
}
```

- We instrumented 7 kernel functions in Linux

Crashes

- Dirty blocks can be written in any order, crash at any point



Users write code to check recovered FS

Outline

- Core idea: explore all choices
- Checking interface
  - What EXPLODE provides
  - What users do to check their storage system
- Implementation
- Results
- Related work, conclusion and future work

What EXPLODE provides

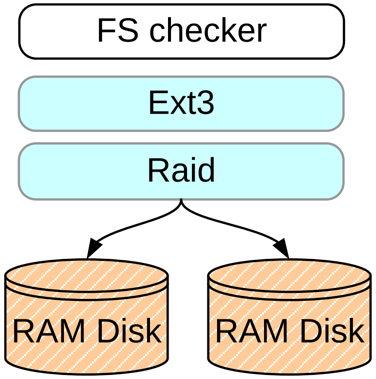
- choose(N): conceptual N-way fork, return K in K'th child execution

- check_crash_now(): check all crashes that can happen at the current moment
  - Paper talks about more ways for checking crashes
  - Users embed non-crash checks in their code. EXPLODE amplifies them

- error(): record trace for deterministic replay

What users do

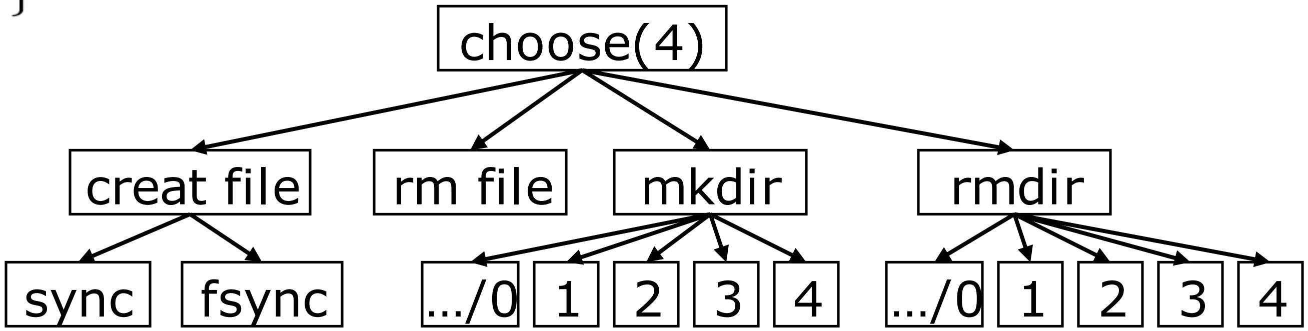
- Example: ext3 on RAID



- checker: drive ext3 to do something: mutate(), then verify what ext3 did was correct: check()
- storage component: set up, repair and tear down ext3, RAID. Write once per system
- assemble a checking stack

- FS Checker
  - mutate
- ext3 Component
- Stack

```
const char *dir = "/mnt/sbd0/";
const char *file = "/mnt/sbd0/test-file";
void FsChecker::mutate(void) {
  switch(choose(4)) {
  case 0: systemf("echo \"test\" > %s", file);
    if (choose(2) == 0)
      sync();
    else {
      do_fsync(file);
      // fsync parent to commit the new directory entry
      do_fsync("/mnt/sbd0");
    }
    check_crash_now(); // invokes check() for each crash
    break;
  case 1: systemf("rm %s", file); break;
  case 2: systemf("mkdir %s%d", dir, choose(5)); break;
  case 3: systemf("rmdir %s%d", dir, choose(5)); break;
  }
}
```



- FS Checker
  - check
- ext3 Component
- Stack

```
void FsChecker::check(void) {
  ifstream in(file);
  if(!in)
    error("fs", "file gone!");
  char buf[1024];
  in.read(buf, sizeof buf);
  in.close();
  if(strncmp(buf, "test", 4) != 0)
    error("fs", "wrong file contents!");
}
```

Check file exists

Check file contents match

Found JFS fsync bug, caused by re-using directory inode as file inode

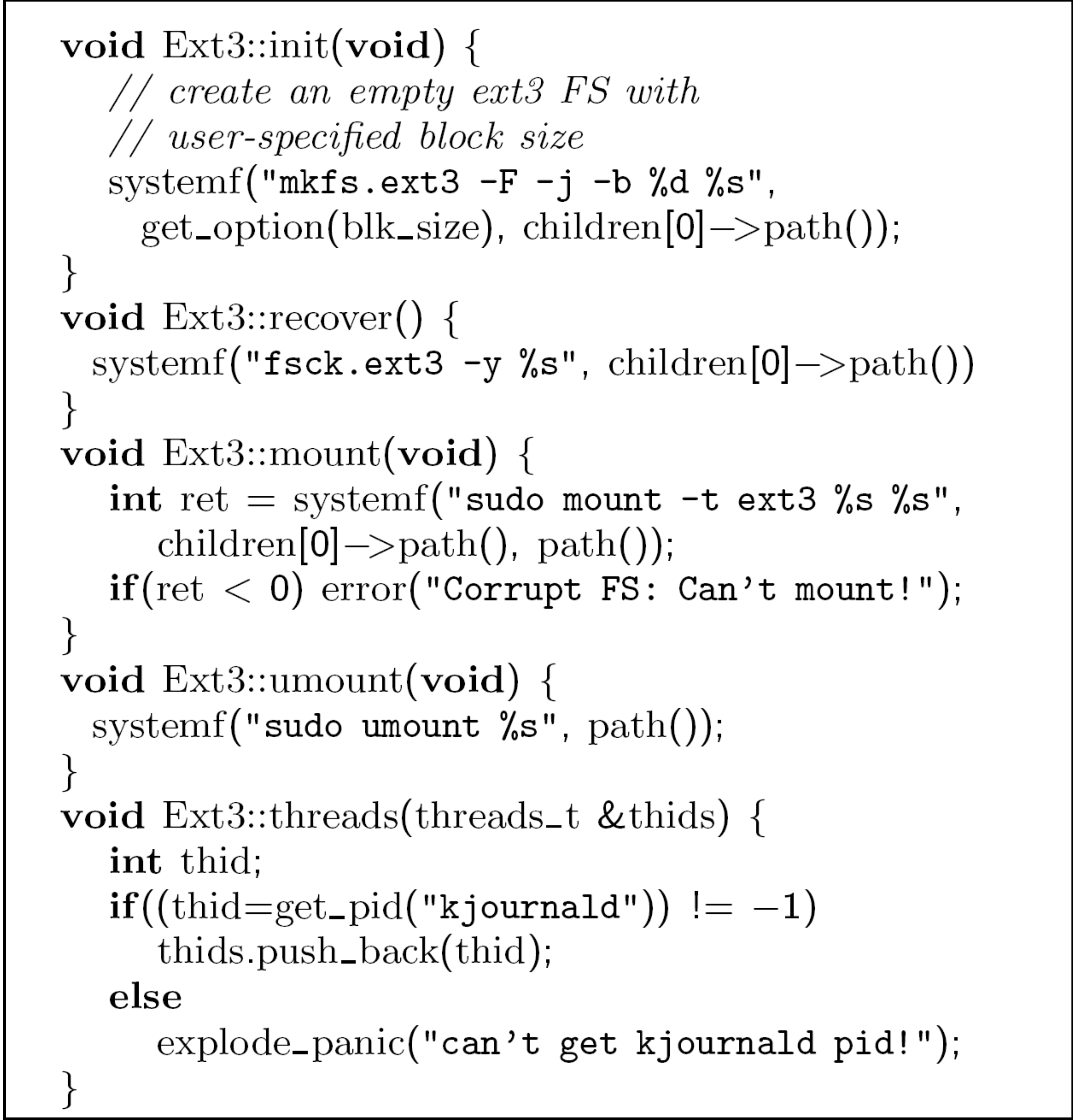
Checkers can be simple (50 lines) or very complex(5,000 lines)

Whatever you can express in C++, you can check

- FS Checker
- **ext3 Component**
- Stack

- storage component: initialize, repair, set up, and tear down your system
  - Mostly wrappers to existing utilities. “mkfs”, “fsck”, “mount”, “umount”
  - threads(): returns list of kernel thread IDs for deterministic error replay
- Write once per system, reuse to form stacks
- Real code on next slide

- ❑ FS Checker
- ❑ **ext3 Component**
- ❑ Stack

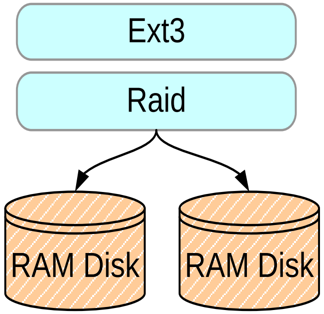


```
void Ext3::init(void) {
   // create an empty ext3 FS with
   // user-specified block size
   systemf("mkfs.ext3 -F -j -b %d %s",
     get_option(blk_size), children[0]−>path());
}
void Ext3::recover() {
  systemf("fsck.ext3 -y %s", children[0]−>path())
}
void Ext3::mount(void) {
   int ret = systemf("sudo mount -t ext3 %s %s",
      children[0]−>path(), path());
   if(ret < 0) error("Corrupt FS: Can't mount!");
}
void Ext3::umount(void) {
  systemf("sudo umount %s", path());
}
void Ext3::threads(threads_t &thids) {
   int thid;
   if((thid=get_pid("kjournald")) != −1)
      thids.push_back(thid);
   else
      explode_panic("can't get kjournald pid!");
}
```

- FS Checker

- ext3 Component

- Stack

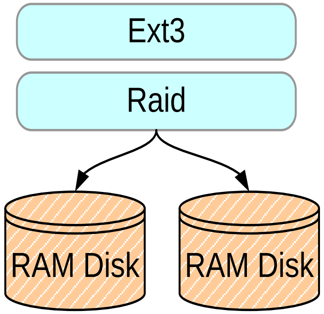


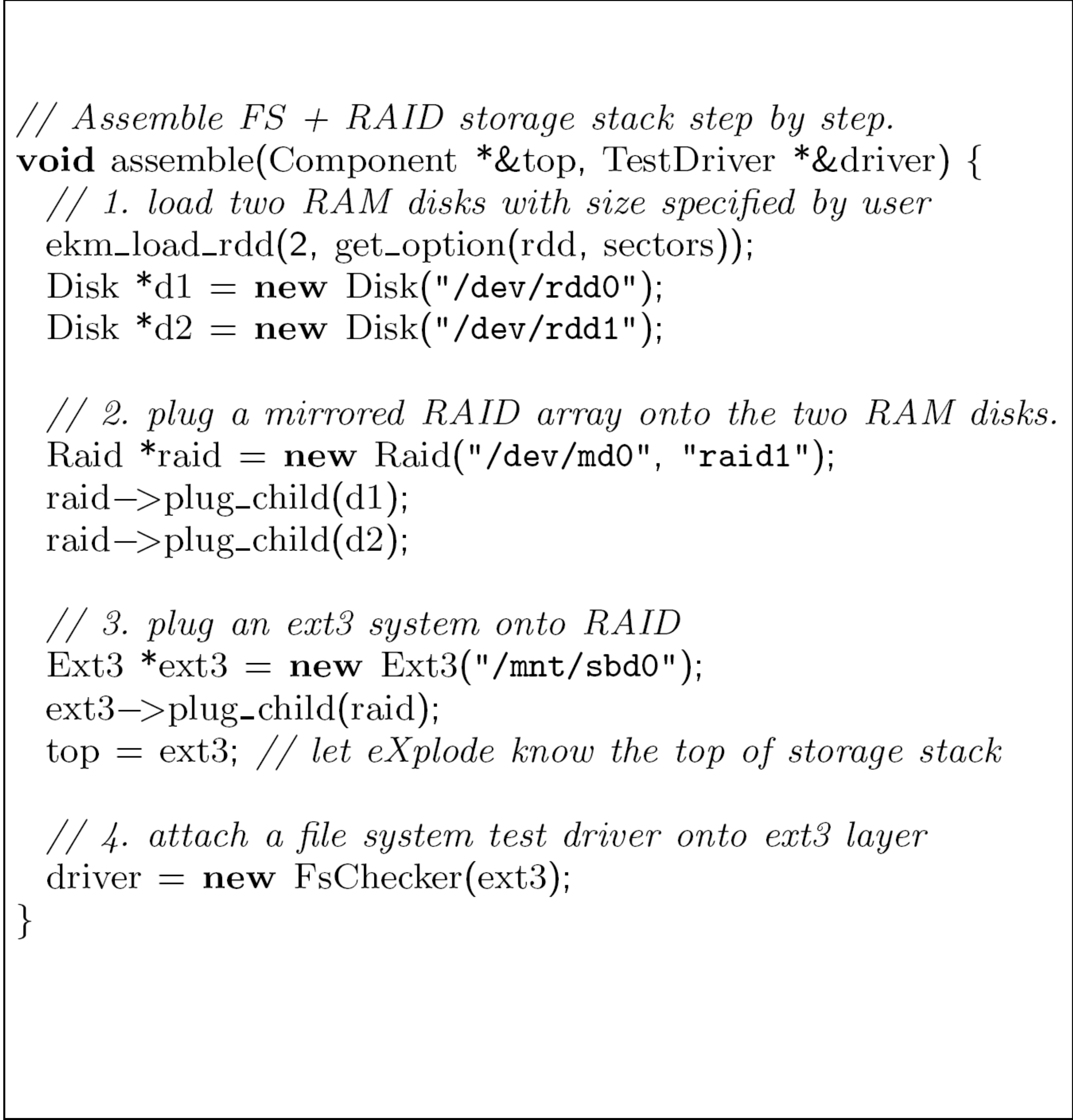
- assemble a checking stack

- Let EXPLODE know how subsystems are connected together, so it can initialize, set up, tear down, and repair the entire stack

- Real code on next slide

- FS Checker
- ext3 Component
- Stack





```
// Assemble FS + RAID storage stack step by step.
void assemble(Component *&top, TestDriver *&driver) {
  // 1. load two RAM disks with size specified by user
  ekm_load_rdd(2, get_option(rdd, sectors));
  Disk *d1 = new Disk("/dev/rdd0");
  Disk *d2 = new Disk("/dev/rdd1");

  // 2. plug a mirrored RAID array onto the two RAM disks.
  Raid *raid = new Raid("/dev/md0", "raid1");
  raid->plug_child(d1);
  raid->plug_child(d2);

  // 3. plug an ext3 system onto RAID
  Ext3 *ext3 = new Ext3("/mnt/sbd0");
  ext3->plug_child(raid);
  top = ext3; // let eXplode know the top of storage stack

  // 4. attach a file system test driver onto ext3 layer
  driver = new FsChecker(ext3);
}
```

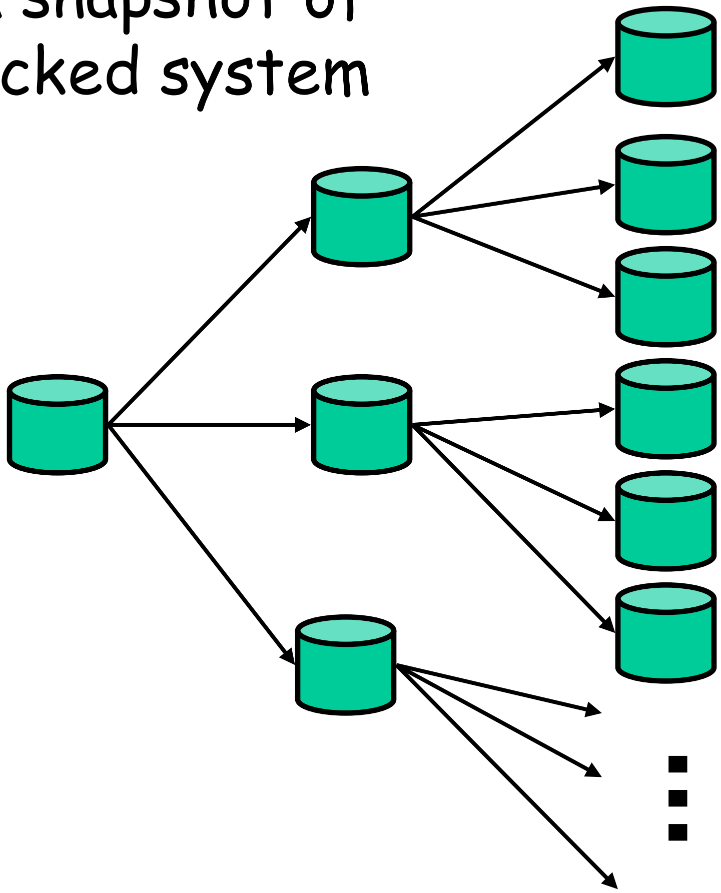
Outline

- Core idea: explore all choices
- Checking interface: 200 lines of C++ to check a system
- Implementation
  - Checkpoint and restore states
  - Deterministic replay
  - Checking process
  - Checking crashes
  - Checking "soft" application crashes
- Results
- Related work, conclusion and future work

Recall: core idea

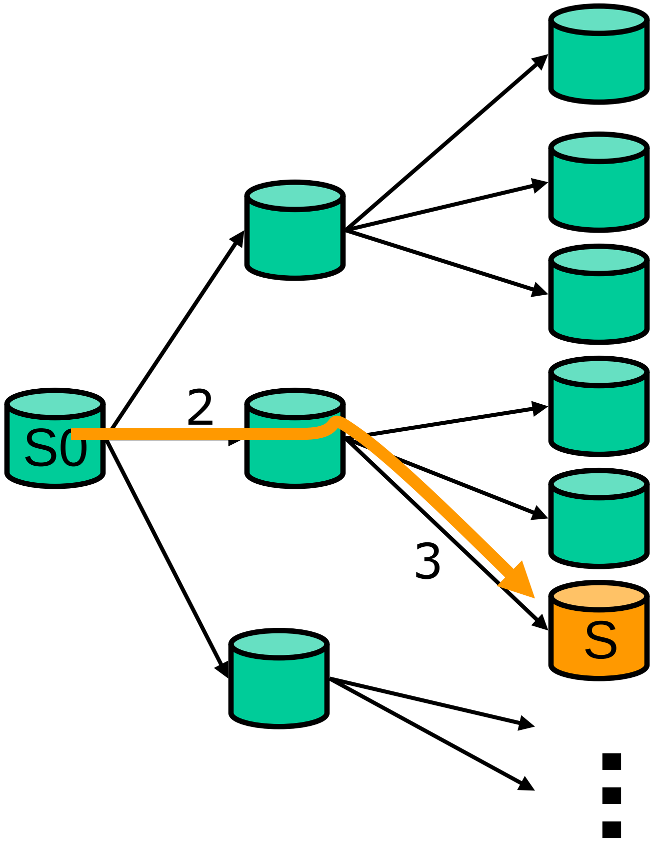
- "Fork" at decision point to explore all choices

state: a snapshot of the checked system



How to checkpoint live system?

- Hard to checkpoint live kernel memory
  - VM checkpoint heavy-weight
- checkpoint: record all choose() returns from S0
- restore: umount, restore S0, re-run code, make K'th choose() return K'th recorded values
- Key to EXPLODE approach



S = S0 + redo choices (2, 3)

Deterministic replay

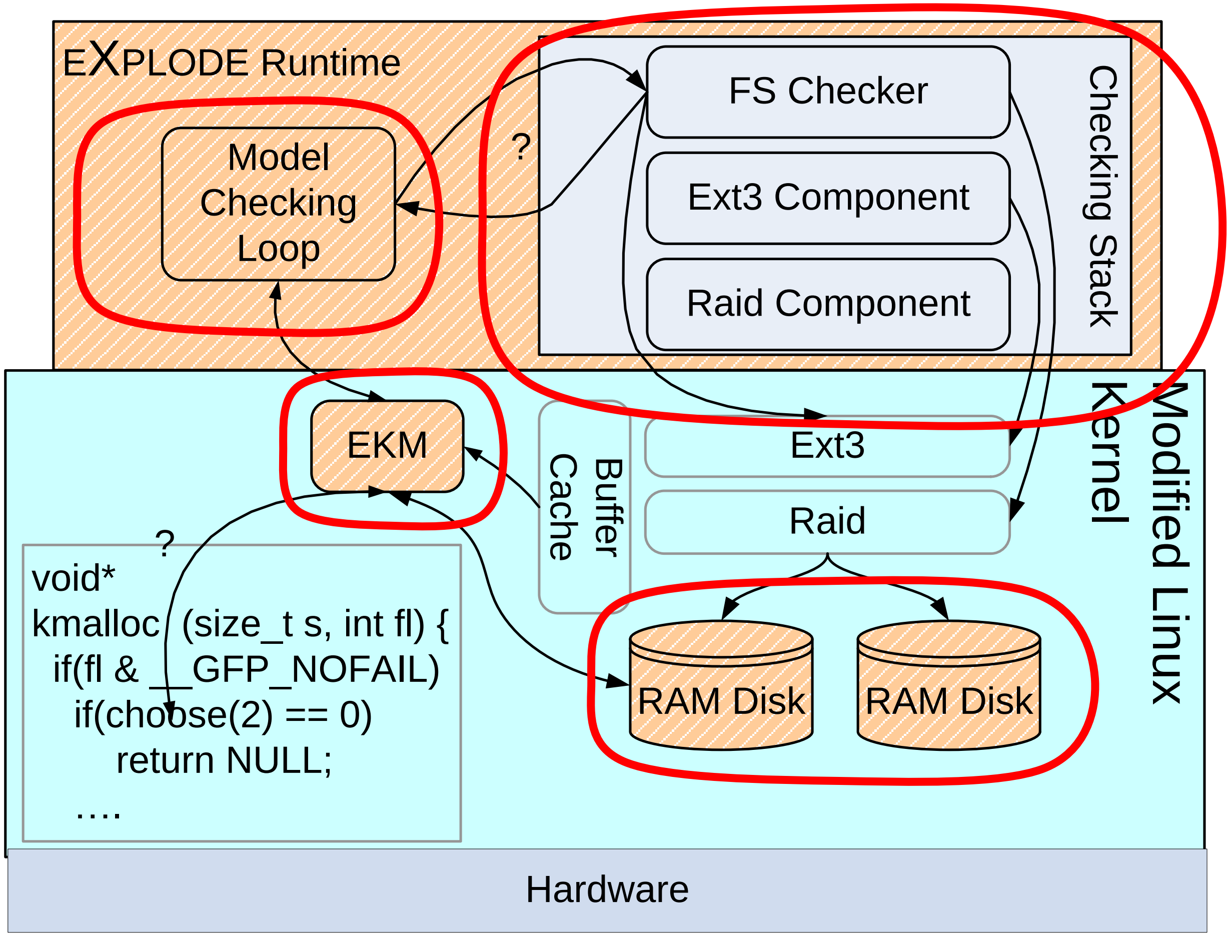
- Need it to recreate states, diagnose bugs

Sources of non-determinism

- Kernel choose() can be called by other code
  - Fix: filter by thread IDs. No choose() in interrupt
- Kernel scheduler can schedule any thread
  - Opportunistic hack: setting priorities. Worked well
  - Can't use lock: deadlock. A holds lock, then yield to B
- Other requirements in paper

- Worst case: non-repeatable error. Automatic detect and ignore

EXPLODE: put it all together



```
void*
kmalloc (size_t s, int fl) {
  if(fl & __GFP_NOFAIL)
    if(choose(2) == 0)
      return NULL;
  ….
```

▨ EXPLODE □ User code

EKM = EXPLODE device driver

Outline

- Core idea: explore all choices

- Checking interface: 200 lines of C++ to check a system

- Implementation

➡ Results
  - Lines of code
  - Errors found

- Related work, conclusion and future work

EXPLODE core lines of code

		Lines of code
Kernel patch	Linux	1,915 (+ 2,194 generated)
	FreeBSD	1,210
User-level code		6,323

3 kernels: Linux 2.6.11, 2.6.15, FreeBSD 6.0.
FreeBSD patch doesn't have all functionality yet

Checkers lines of code, errors found

Storage System Checked		Component	Checker	Bugs
10 file systems		744/10	5,477	18
Storage applications	CVS	27	68	1
	Subversion	31	69	1
	“EXPENSIVE”	30	124	3
	Berkeley DB	82	202	6
Transparent subsystems	RAID	144	FS + 137	2
	NFS	34	FS	4
	VMware GSX/Linux	54	FS	1
Total		1,115	6,008	36

Outline

- ❑ Core idea: explore all choices

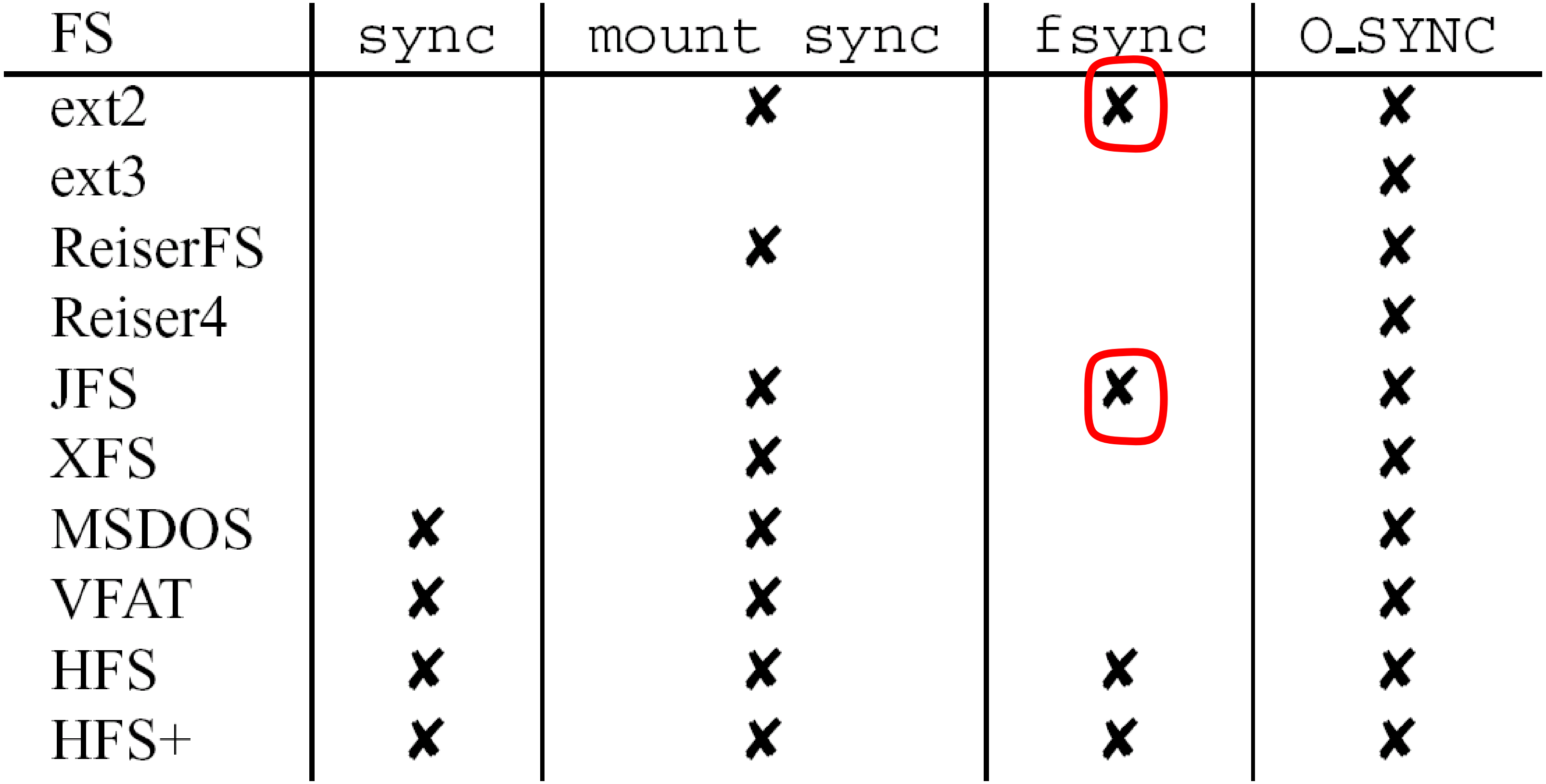
- ❑ Checking interface: 200 lines of C++ to check new storage system

- ❑ Implementation

- ❑ Results
  - ▪ Lines of code
  - ➡ Errors found

- ❑ Related work, conclusion and future work

FS Sync checking results



FS	sync	mount sync	fsync	O_SYNC
ext2		✗	✗	✗
ext3				✗
ReiserFS		✗		✗
Reiser4				✗
JFS		✗	✗	✗
XFS		✗		✗
MSDOS	✗	✗		✗
VFAT	✗	✗		✗
HFS	✗	✗	✗	✗
HFS+	✗	✗	✗	✗

✗ indicates a failed check

App rely on sync operations, yet they are broken

ext2 fsync bug

Events to trigger bug

truncate A

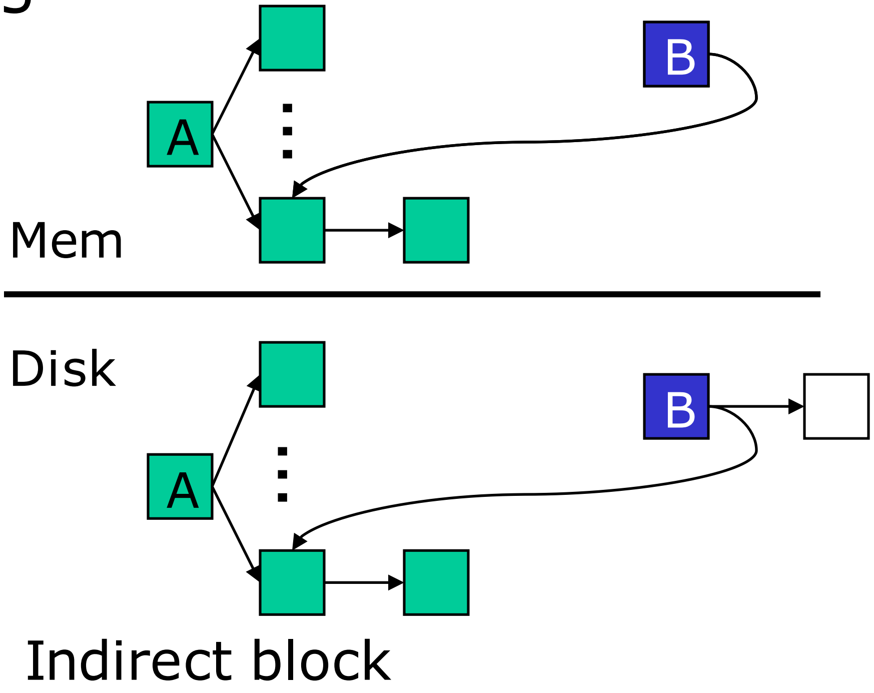
creat B

write B

fsync B

crash!

fsck.ext2



Bug is fundamental due to ext2 asynchrony

Classic app mistake: “atomic” rename

- All three version control app. made this mistake

- Atomically update file A to avoid corruption

```
fd = creat(A_tmp, …);
write(fd, …);
fsync(fd); // missing!
close(fd);
rename(A_tmp, A);
```

- Problem: rename guarantees nothing abt. Data

Outline

- Core idea: explore all choices
- Checking interface: 200 lines of C++ to check a system
- Implementation
- Results: checked many systems, found many bugs

➡ Related work, conclusion and future work

Related work

- FS testing
  - IRON

- Static analysis
  - Traditional software model checking
  - Theorem proving
  - Other techniques

Conclusion and future work

- EXPLODE
  - Easy: need 1 device driver. simple user interface
  - General: can run, can check, without source
  - Effective: checked many systems, 36 bugs

- Future work:
  - Work closely with storage system implementers to check more systems and more properties
  - Smart search
  - Automatic diagnosis
  - Automatically inferring "choice points"
  - Approach is general, applicable to distributed systems, secure systems, …