Operating Systems: Three Easy Pieces

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组

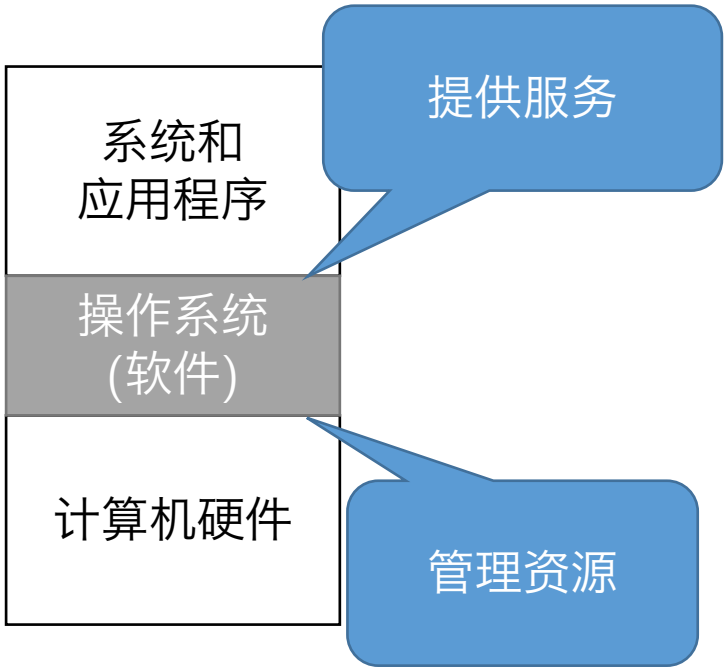
操作系统：必学部分复习

Operating Systems: Three Easy Pieces

- 虚拟化
  - 进程抽象(fork, execve, exit)
  - 虚存抽象(mmap)
- 并发
  - 进程与线程(pthread)
  - 并发控制：同步与互斥(mutex, semaphore)
- 持久化
  - 设备管理、文件抽象(open, close, readdir, ...)
  - 持久数据可靠性：崩溃一致性、RAID

Operating Systems: Three Easy Pieces

- 看起来是没什么关系的三部分
  - 是应用程序的各种需求把它们聚合到一起



操作系统的使命：为应用服务

- 提供全面、易用、高效的服务
  - OS APIs (Windows API, Linux系统调用, Android SDK, ...)

- 从应用的需求考虑
  - 需要干净(独立不被打扰)的运行环境：虚拟化
  - 需要有与其他程序交互的手段：并发 & 进程/线程通信
  - 需要有保存数据的手段：持久化

- 但应用还有很多其他的需求
  - 网络访问、设备(例如图形)、安全.....根本无法在一学期讲完

我们学到了什么?

- 一组OS API的设计(UNIX系统调用)
  - fork, execve, exit, mmap, read, link, ...
- OS API在硬件上的实现
  - 理论讲解：各种机制和策略
  - 动手实践(不做不知道那么多坑)
- 大概上来说，你知道
  - 怎么用操作系统API编程
  - 怎么构造一个操作系统，包括任何你想加入的部分
    - 网络、安全、图形、.....

操作系统：其他趣事 (MOS Cover)

什么？我们就学完了？

最后做个小总结：execve的实现

```
int execve(const char *path,
  char *const argv[],
  char *const envp[]);
```

- (看起来的)流程
  - 打开并解析path所对应的文件
  - 创建新地址空间、把argv, envp保存到堆栈
  - 把可执行文件的sections (.text, .data, ...) mmap到新地址空间
  - 替换当前进程的地址空间

- 在这里会有什么坑?

execve的行为你知道吗?

- 进程/线程还有很多资源
  - 属于同一进程的线程本身呢?
  - 信号(signals)?
  - 各种文件相关的资源(mmap, fd, ...)
  - 还没执行完的异步I/O (aio)?

- 明确这些行为本身就很难
- 怎么把这件事做对就更难了

以及还有……

- execve()可能并发
  - 不同的进程调用、甚至同一个进程的两个线程同时execve
  - 大量并发代码，例如kalloc()
- execve()可能和其他系统调用并发
  - 系统中的其他进程正在删除被execve的文件
  - ……
- execve()中随时可能出现错误
  - 文件
  - 内存分配失败
  - ……

怎么办?

- 好办(做作业时候的办法)
  - 在关中断的情况下执行系统调用 ← 来自你们的学长
  - sem_wait(&big_lock); ← 不用关中断了(Big Kernel Lock)

- 实际
  - Linux Kernel里execve()的代码直接开始运行
  - 在多处理器情况下，绝大部分代码都是最大程度并行执行的

- 必须经过必要的训练，才能驾驭实际的系统
  - 虽然按没有编程课是非常糟糕的
  - 照我那时候的教法，有了也没啥用

计算机系统学习之路

入门：从玩具系统开始

- 教科书
  - “Modern Operating Systems”, “Operating Systems: Three Easy Pieces”, ...
- 参考代码
  - busybox (应用程序集合)
  - newlib (库函数)
  - xv6 (操作系统内核)
- 动手实践：熟悉操作系统里的概念、机制.....
  - (top-down, Mini Labs) 用操作系统API编各种好玩的东西
  - (bottom-up, OS Labs) 操作系统是利用了硬件提供机制的C程序

进阶：Hack一个真实的系统

- 走出第一步
  - 哪怕是在Linux Kernel里加一个调试信息、写一个驱动.....
  - 但会遇到很多阻碍
    - 环境配置一大堆文档
    - make干了一坨事情，根本不知道发生了什么
    - 很多不明所以的概念(比如经常看到的initrd/initramfs)

- Don't Panic
  - 试着把系统里未知的东西映射到你已知的概念(玩玩具的重要性)
  - 理解系统中的抽象
    - 有些black boxes你“大概知道是什么”，但细节不明
    - 逐渐找你希望理解的细节

进阶：Hack一个真实的系统 (cont'd)

- Linux Kernel小技巧
  - ARCH=x86 make cscope (或者用你喜欢的IDE)
  - 做一个只包含busybox的最小镜像
  - 实现一键qemu + gdb调试
    - 这个技巧在大家做OSLab的时候也很有用！
- 用对的工具做对的事非常重要
  - 所谓的“系统能力”
  - 给你一个合理的需求，你总能在基本可控的时间里实现出来

进阶：计算机系统研究

- 可以做一些更有趣的事情了！
  - 现在还有什么事做不到？做不好？
  - 最近世界发生了什么变化？它们会给计算机系统带来什么？
    - GPU, FPGA
    - non-volatile memory
    - 虚拟化指令集
    - ...
- 计算机系统研究
  - 更可用、更可靠、更高效
  - 大家可以看一下最近SOSP/OSDI的接收论文

RAID @ FAST18

Operating System Research: 例子

- 复习：RAID-5, RAID-6 (性能、容错)
  - 系统研究者仍然沿用Redundant Array of Inexpensive Disks的说法
    - 所以看Remzi的书就对了

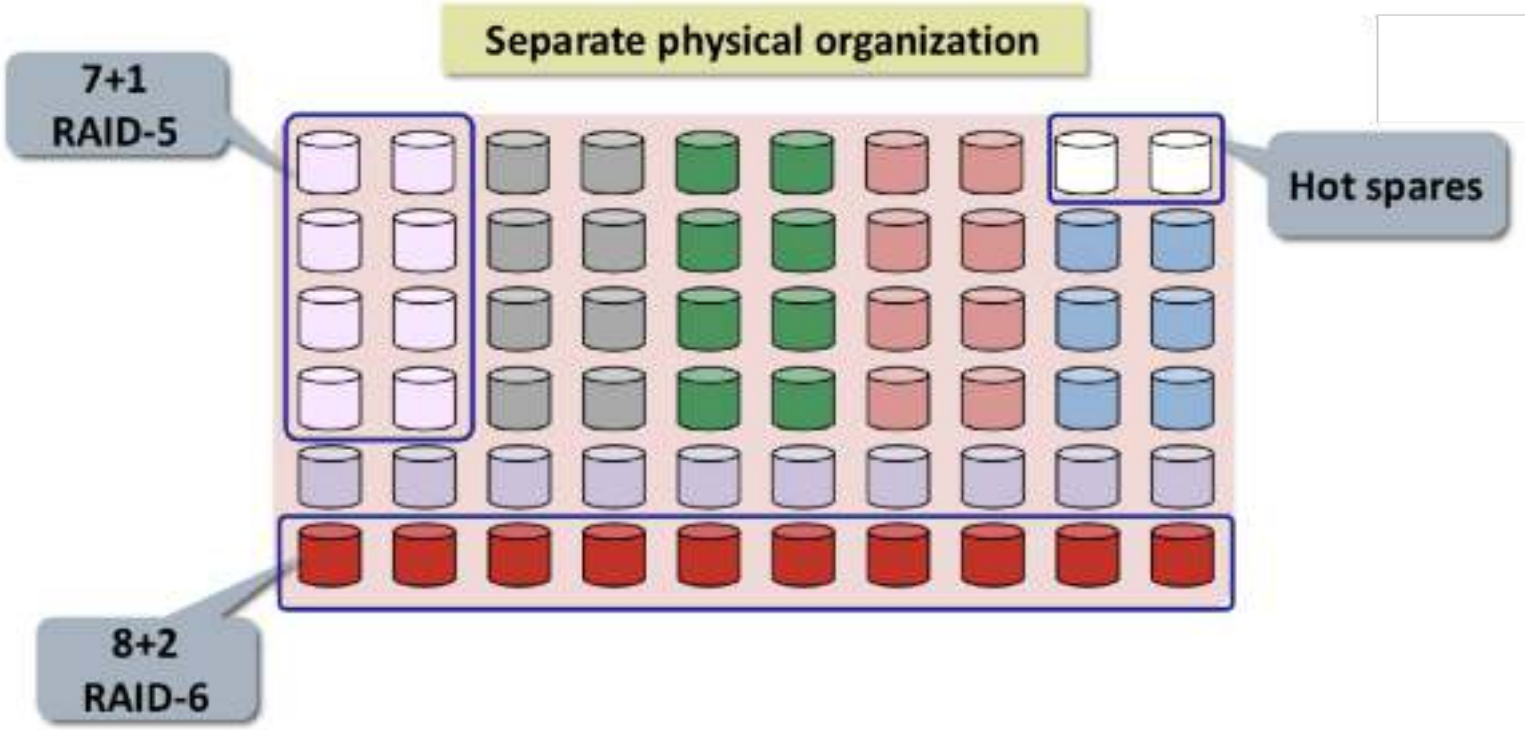
- 但如果是大磁盘阵列呢?
  - 60块盘, 120块盘, ... 每块2TB, ... 是否有新的挑战?



[1] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, Weimin Zheng. RAID+: Deterministic and balanced data distribution for large disk enclosures. In *Proc. of FAST*, 2018. 图片来自他们的slides.

怎么在60 x 2TB磁盘阵列上组RAID?

- RAID-50 (5+1 RAID-5) x 10
  - 一块100TB的……盘？这fsck得一万年吧?
- 更科学的方法：分区(虚拟磁盘)

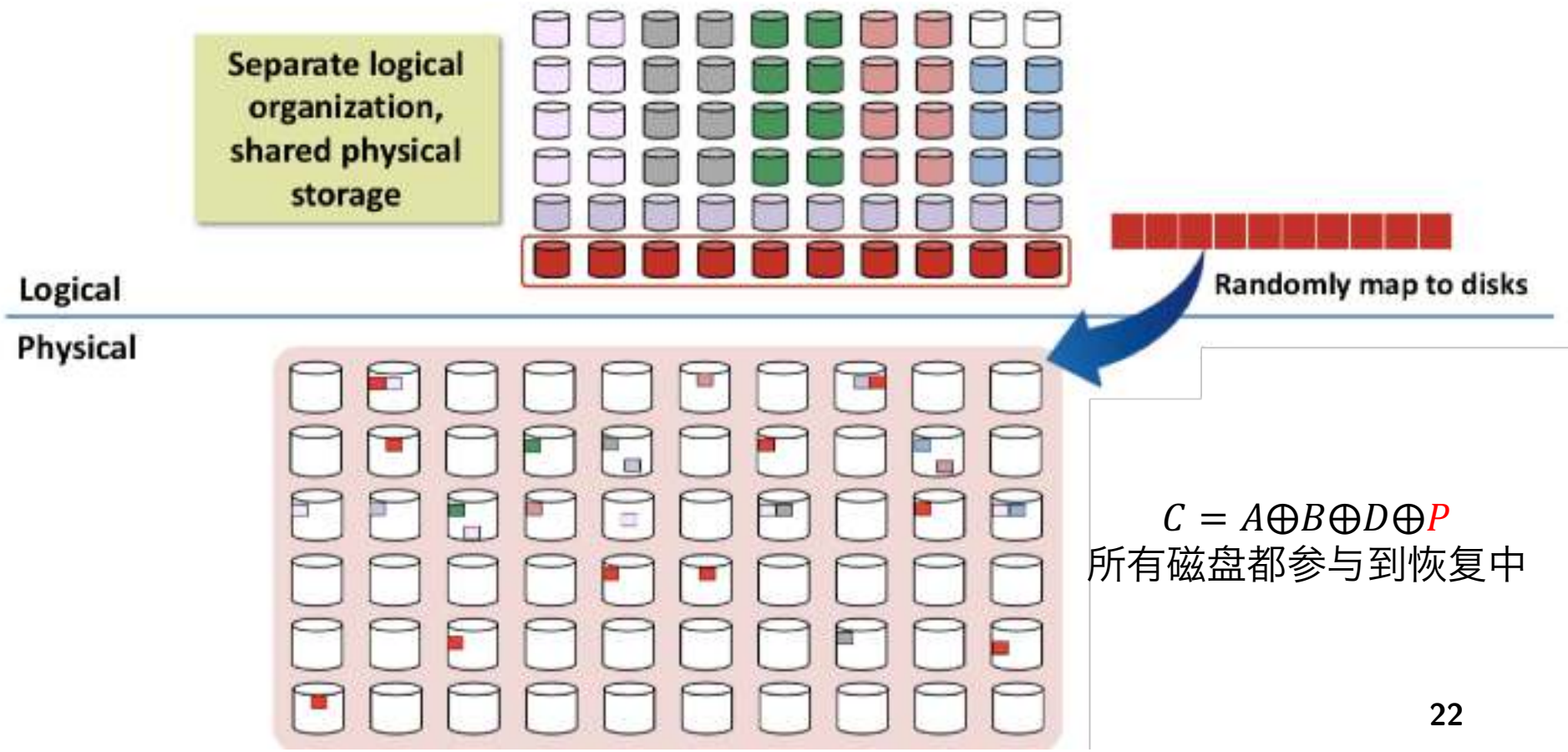


虚拟磁盘

- RAID-5的rebuild速度依然很慢
  - NetApp: 2TB 7200rpm 全盘扫描 12.8h
  - 如果这时候又有一块盘坏了...RAID-5就没得帮你了
  - 而且坏掉的那个阵列瞬间成为瓶颈
    - 数据是通过奇偶校验算出来的
- 我们有那么多磁盘，能不能把数据“均匀分布”到各盘？
  - 坏一块盘，损坏的数据是“各个盘”的
  - 不会成为单个RAID的瓶颈

虚拟化(随机映射，已有技术)

- 需要索引(额外开销)，但坏盘rebuild都只影响任何一个虚拟盘中的部分数据了！减少rebuild时间→增加可靠性

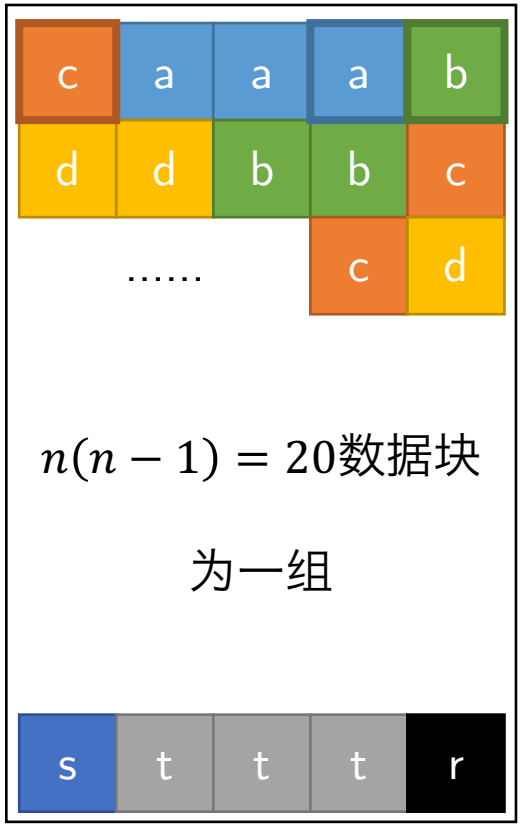


$C = A \oplus B \oplus D \oplus P$

所有磁盘都参与到恢复中

RAID+：确定(Deterministic)的映射方法

- 例子：$n = 5$块盘、组成$k = 2 + 1$条带的RAID-5
  - 容错：k条带组成RAID
  - 负载均衡：数据块均匀分布到所有磁盘
  - 容易维护：确定性的分配方法
    - 类似RAID-5：0,1,P;2,P,3;...
- 分配方法：Orthogonal Latin Squares
  - a (1, 2, 3), b (2, 3, 4), c (3, 4, 0), ...
  - 把磁盘分为很多组



神奇的好处：出错恢复

- 如果0号盘坏了
  - 可以立即得到一个“备盘号”
  - 数据并行向备盘写入
    - 挂了一块盘，不再是把这块盘重写一遍，而是把workload均匀分布到每块盘上
- 通过在每块盘预留一些空间
  - 不需要备盘就能实现热备份

a	1	2	3	4
b	2	3	4	0
c	3	4	0	1
d	4	0	1	2
e	0	1	2	3
f	2	4	1	3
g	3	0	2	4
h	4	1	3	0
i	0	2	4	1
		…		

小结

- 更多细节可以阅读FAST'18的原文
  - RAID+: Deterministic and balanced data distribution for large disk enclosures
  - 有更多细节：读/写优化；出错恢复.....
- 计算机系统：更可用、更可靠、更高效
  - 让大型磁盘阵列更快、更可靠、更容易地使用
  - 是一份非常典型的计算机系统(storage system)研究

Crash Consistency @ FAST18

当我们说崩溃一致性的时候，FSCK呢?

- 当系统断电崩溃以后，FSCK会扫描文件系统/重做日志
  - 在这时候会修改文件系统的状态
  - 但如果修改到一半的时候崩溃了……呢……

- FSCK也是程序，谁来保证FSCK的崩溃一致性?

[1] Om Rameshwar Gatla, et al. Towards robust file system checkers. In *Proc. of FAST*, 2018.

? ? ?

Subject: Update: HPCC Power Outage
Date: Monday, January 11, 2016 at 8:50:17 AM Central Standard Time
From: HPCC - Support
Attachments: image001.png, image003.png

TEXAS TECH UNIVERSITY
Information Technology Division

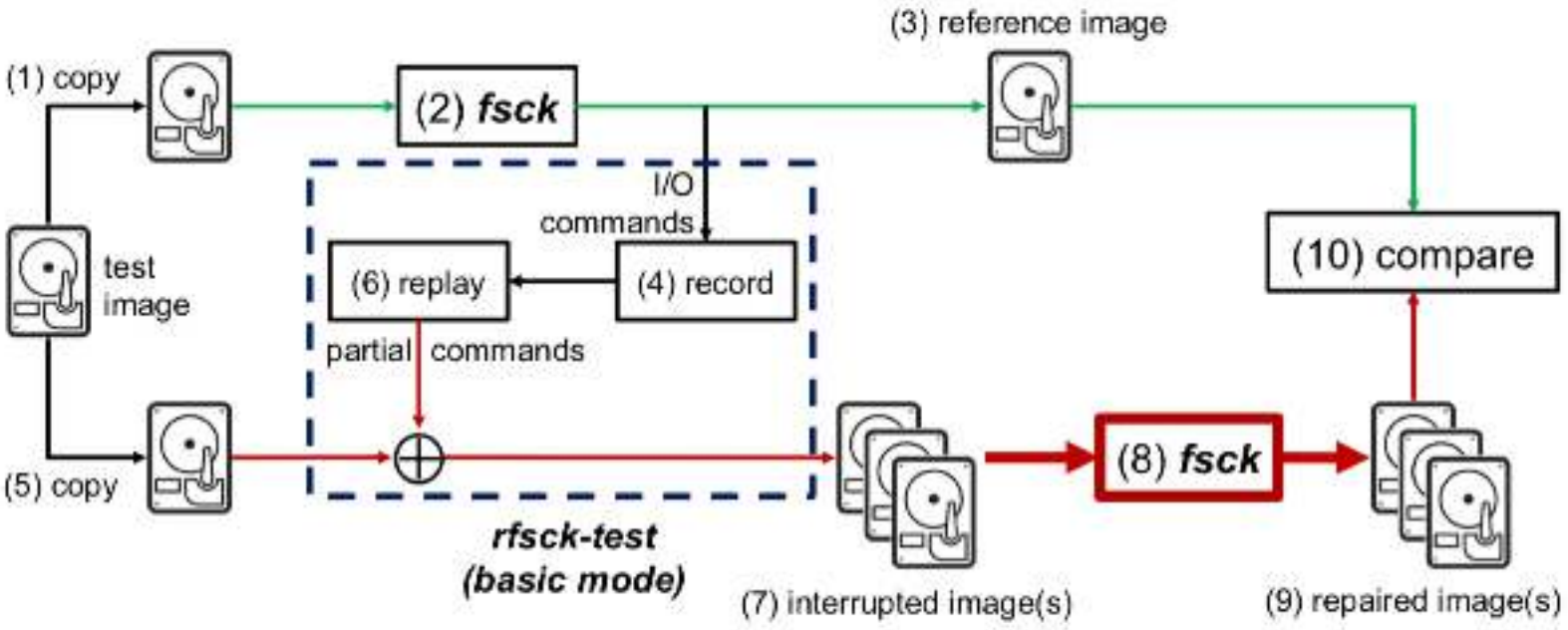
High Performance Computing Center

To All HPCC Customers and Partners,

As we have informed you earlier, the Experimental Sciences Building experienced a major power outage Sunday, Jan. 3 and another set of outages Tuesday, Jan. 5 that occurred while file systems were being recovered from the first outage. As a result, there were major losses of important parts of the file systems for the work, scratch and certain experimental group special Lustre areas.

The HPCC staff have been working continuously since these events on recovery procedures to try to restore as much as possible of the affected file systems. These procedures are extremely time-consuming, taking days to complete in some cases. Although about a third of the affected file systems have been recovered, work continues on this effort and no time estimate is possible at present.

模拟崩溃，收集现场


(1) copy
(2) fsck
(3) reference image
I/O
commands
test
image
(6) replay
(4) record
(10) compare
partial commands
(5) copy
(8) fsck
rfsck-test
(basic mode)
(7) interrupted image(s)
(9) repaired image(s)

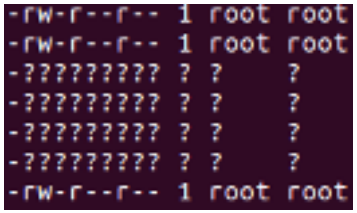
检测结果

- 四种类型的问题(再次运行fsck无法解决)

无法挂载

数据损坏

文件错位



？？？

- e2fsck (Linux ext2/3/4 fsck)

	磁盘数据有损坏		损坏 + 崩溃	
	512B	4KB	512B	4KB
无法挂载	20	1	41	3
数据损坏	9	5	107	10
文件错位	9	11	82	23
？？？	1	1	10	1

如何解决FSCK的问题?

- 如何设计和实现一个“Robust File System Checker”？

- 核心问题
  - fsck中write操作的原子性
  - 用之前讲过的TX就解决问题啦(rfsck-lib)!

小结

- 构造可靠的系统绝非易事
- 计算机系统太复杂了
  - 编译器、Linux Kernel动辄百万行、千万行代码、无数的依赖
  - 不仅要考虑功能，还有崩溃一致性、安全问题、……
- 目前我们生活在一个并不绝对可靠的世界里
  - 等待大家去改变