CSP

1 Intro

1: Way to use hardware

–Knowledge of OS, hardware, compiler, runtime, etc.

–E.g., page table, TLB, file system

2: Way to think systematically

–A way of analyzing and solving problems

–E.g., system design principles

OS Architecture

Simplicity：在发展演变的过程中，内核态数据、逻辑呈减少趋势，降低系统复杂度，便于理解。

简化甚至大于正确性

Why system

1. for fun
2. for profit

General design principles

Adopt sweeping simplifications
Avoid excessive generality
Avoid rarely used components
Be explicit
Decouple modules with indirection
Design for iteration

End-to-end argument
Escalating complexity principle
Incommensurate scaling rule
Keep digging principle
Law of diminishing returns
Open design principle

Principle of least astonishment
Robustness principle
Safety margin principle
Unyielding foundations rule

Specific Area Principles

Atomicity: Golden rule of atomicity
Coordination: One-writer principle
Durability: The durability mantra
Security: Minimize secrets
Security: Complete mediation
Security: Fail-safe defaults

Security: Least privilege principle
Security: Economy of mechanism
Security: Minimize common mechanism

Design Hints

Exploit brute force
Instead of reducing latency, hide it
Optimize for the common case
Separate mechanism from policy

Properties of Computer Systems

Correctness
Latency
Throughput
Scalability
Utilization
Performance Isolation
Energy Efficiency

Consistency
Fault Tolerance
Security
Privacy
Trust
Compatibility
Usability

Case Study: Map-Reduce

So, how to be more “KISS”?

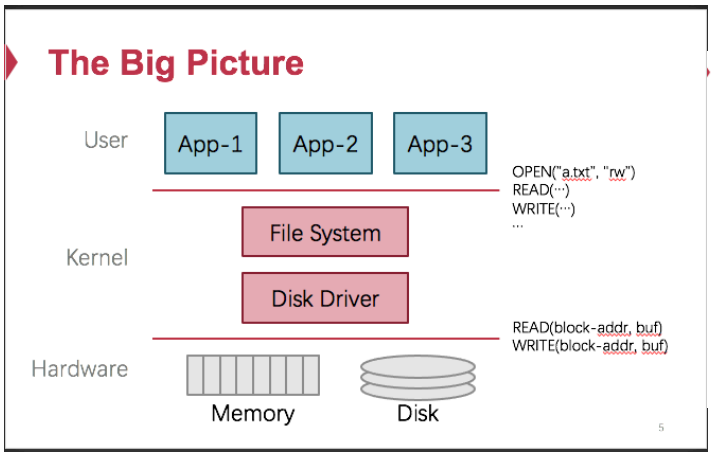
CASE Study: Bash Attack

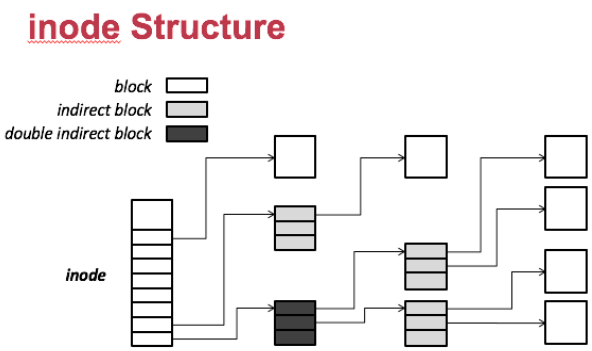
Adopt KISS to Kernel

Monolithic Kernel, Micro-Kernel,Hybrid-Kernel, Kernels, Exokernel

Library Operating System

2 系统分层设计 file system





底层基础设施

内核里包含了文件系统和disk driver

对上 提供文件系统的API

对下 memory和disk组成的hardware

承上启下

inode Structure

组织磁盘块的方式：inode（index node） 设计成了多层次的结构

inode记录了文件的元信息。磁盘会被分成两部分，一部分元数据（重要），一部分数据（量大）。元数据头上是inode，inode的大小是小于一个block（4k->几个bite）

inode连起来组成一个table，使用bitmap记录哪些inode是空闲的，哪些是使用的。

inode每一个项都指向disk的block id，知道一个文件的数据保存在哪些磁盘块上面，如果不够的话会有二级、三级间接指针指到磁盘块，再由磁盘块指到真正的数据项。

inode把文件的offset（文件数据在哪个偏移量）转换成block id（偏移量所记载的实际数据）

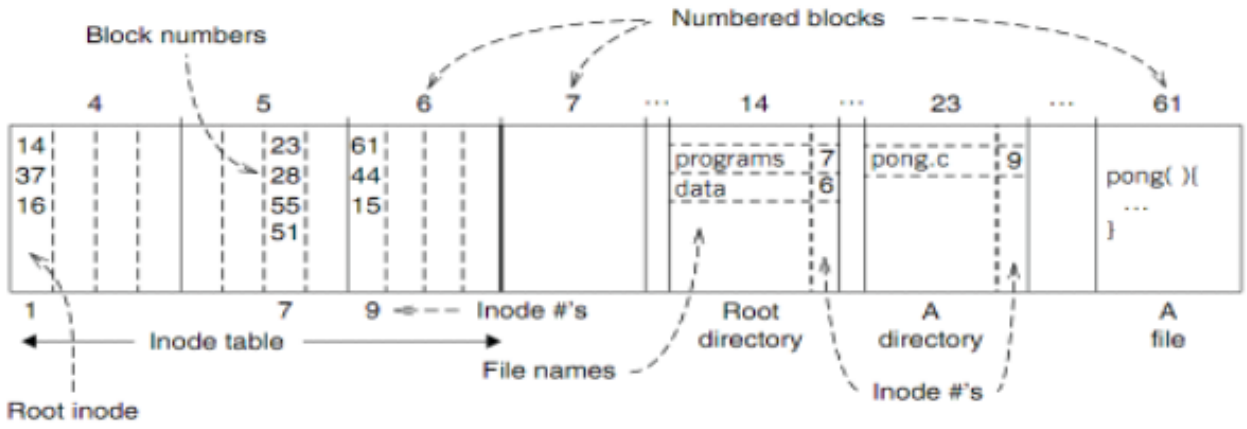
ext4时，可以指向一块block（1M 4M）

NVRAM，会写SSD，但SSD会磨损。

文件系统中头部是元数据（如bitmap），总是写元数据，SSD头部非常容易坏掉。在SSD里加了一层转换层，OS总写的id转换到了物理id，让每次写看起来是写SSD头部，内部转换后写别的地址，从而达到写SSD的磨损负载均衡。

“在任何系统中的问题，都可以通过增加一层抽象来解决”

An example: find blocks of “/programs/pong.c”



1. 根目录（default：第一个inode），记录了block num
2. 找到14，第14个block记录了根目录数据：文件名到inode num的映射关系
3. 字符串匹配，program --> 7(inode num)
4. 去table里找到7号inode，其中记录了block num，7号inode本身就是文件，也是目录
5. 找到23，字符串匹配pong.c-->9
6. 找到9号inode table，找到61
7. 于是发现数据在61

Two Types of Links (Synonyms)

hard link

目录里加一句 datalink=6

soft link

文件名本身作为数据保存在磁盘上

My-soft-link真的是一个文件，文件就是my file.txt

FAT

不用inode的索引方法

从0到n-1都是记录了磁盘的cluster（单位），每一个cluster都会记录我属于或者不属于一个文件（是free的）。如果属于一个文件，e.g.：31——保存在第31个cluster的文件是什么，如果放不下，还需要一个cluster，31就会指向下一个cluster，直到end of file

创建文件和文件布局方面都不一样

增加数据块到文件时，从free里找一块给文件；删除文件时，加到free list里去。因此FAT表格非常重要，一旦表格坏了，就完全找不到文件了。

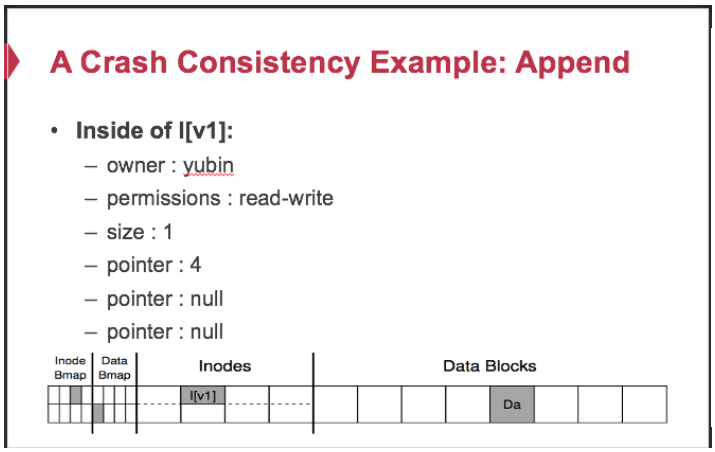
Crash Consistency Example:Append

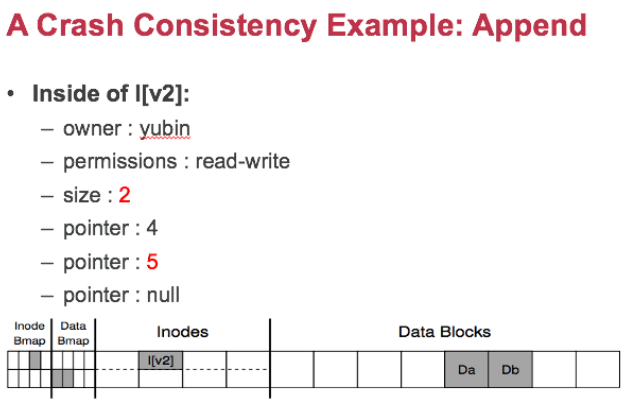
build reliable system on unreliable component

最重要的错误并不来自于数据丢失，而是来自于元数据的不一致。

需要保证元数据在磁盘中的不一致：

example中4次写操作，需要去考虑一次成功，两次成功的情况等等。





1 succeed

Imagine only a single write succeeds; there are thus three possible outcomes:

1. Just the data block (Db) is written to disk

   nothing 等价于没写

2. Just the updated inode (I[v2]) is written to disk
   有可能读到别人的数据，从而导致数据泄露

   有可能既被这个文件指向，又被另一个文件指向，会出现文件莫名其妙被修改的情况

   可以通过遍历，发现inode指向了处于free状态的块。

3. Just the updated bitmap (B[v2]) is written to disk

   浪费一块<--本来没用，标记成了已使用

   找到这块浪费的地址的方法<--遍历inode，发现这里没人指过去，就可以发现它是被浪费的

2 succeed

Two writes succeed and the last one fails:

1. The inode (I[v2]) and bitmap (B[v2]) are written to disk, but not data (Db)

   会导致数据泄露。而且扫描的时候检查不出来。（最危险的情况）

2. The inode (I[v2]) and the data block (Db) are written, but not the bitmap (B[v2])

   可能被另外一个文件用

3. The bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2])

   会导致浪费，但不会出现安全问题，也可以扫描出来

如果把三次磁盘操作排个序，会把inode放到最后，就算它没写，也顶多是浪费，但不会导致数据泄露。因此，在写的时候应该有顺序。

如何保证crash consistency

do nothing; journal; copy-on-write; soft update

journaling

加日志

write-ahead logs　Record before update

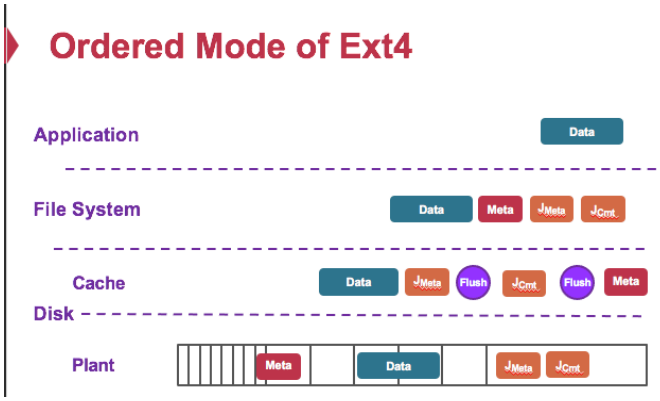
把append操作记录下来写在journal，问题化归到怎么让日志有原子性。

只用journal保护metadata，如果数据很重要，提供多个选项：Ext4 option，可以用来保护数据。

journal：所有东西都用journal保护。

ordered：元数据保护，数据不保护。（default）

writeback：都不保护。



1. write data into disk
2. journal metadata
3. （flush）
4. journal commit
5. （flush）
6. write metadata into disk

要让磁盘保证按照这个顺序执行<--flush操作

写了A，flush之后再写B，就可以保证顺序了

但使用flush，只是为了保证顺序，并不是真的非要flush。（flush很慢 想办法只排序，不持久化）

journal without ordering

会有probabilistic crash consistency,可能会导致crash

两种技术来减少crash inconsistency：

1. checksums

   检查D和journal metadata的checksum，和Jc中存储的是否一致，如果不一致，则是nothing，一致则是everything。但仍然需要一个flush

2. re-order：remove flush by delay writes

   把metadata挂在中断函数里，前面的写完后发个中断，收到中断后再写metadata。

以上就是**optFS**，结果非常好。

soft update

想update之间的dependency是什么，谁在谁之前之后，需要对文件系统的语义了解的非常清楚。

File systems for NVM

发展历程：

1. 最初，为了继续利用传统的DiskFS,将NVM当成一个很快的disk使用->暴殄天物
2. 应该把NVM当成内存来用，内核态提供文件系统，上层应用通过mmap的方式直接映射到NVM，这样可以让上层应用直接看到的是内存。依然对上提供文件系统的接口（read/write），但自己用的是load/store。
3. 在用户态放一个library，如果要文件系统的接口，就用library来实现（用户态实现）。用户态想当成文件系统或者内存都可以。
4. 两者结合。内核维护一个module，内核尽可能暴露多的接口给用户态，用户态维护一个libFS，接口可以提供文件系统的接口，也可以提供object的存储。

Why

Why FS

FS的兼容性好，大量的使用都是基于文件系统的接口

why NVMFS

DiskFS中很多层是不需要的。可以删掉。

但仍然会有crash consistency的问题 <- cache会丢

write-back cache怎么保证order：

CLFLUSH 指令

写到memory

但很慢，会停住CPU，flush完后还会把cache清掉。（没办法cache hit）

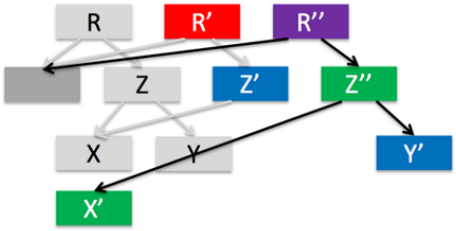
CLFLUSHOPT：可以并行了

CLWB：不会invalidate cache。

copy-on-write（CLWB的使用）

把最后一个操作变成原子的

Example: Copy-on-Write




```
STORE Y'
STORE Z'
CLWB Y'
CLWB Z'
SFENCE
STORE R'
CLWB R'
SFENCE
STORE X'
STORE Z''
CLWB X'
CLWB Z''
SFENCE
STORE R'
CLWB R'
```

commit point在R->R',因此在STORE R'之前加了SFENCE

NVM for storage

设计独立的软件栈来实现

NVMFS用load/store instead of Block I/O

crash consistency

从CPU到内存可能不一致

如何保证断电后正确恢复

CPU写到内存时，有指令可以把内容flush到disk上

Copy-on-write:通过新写内容来使修改内容生效->会导致大量garbage

需要GC