中国科学院大学
University of Chinese Academy of Sciences

WAFL、NFS与系统安全

中国科学院大学计算机学院

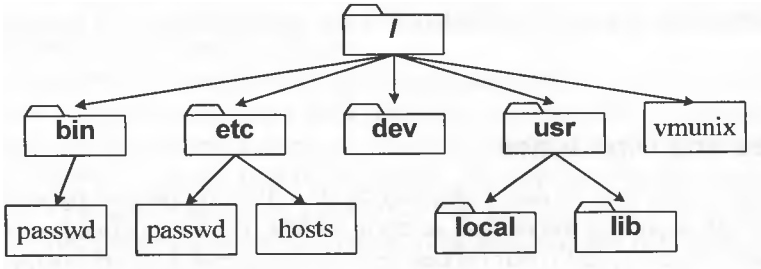
2026-01-14

内容提要

- 文件系统可靠性
- WAFL
- NFS
- 安全保护

威胁FS的因素

- FS要求
  - FS给用户提供持久化的数据存储
  - 文件一直要保存完好，除非用户显式删除它们

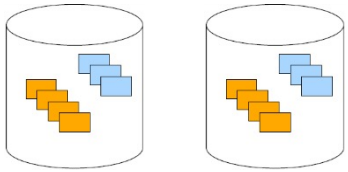


- 威胁一：宕机或掉电
- 威胁二：设备损坏
  - 磁盘块损坏
  - 超级块损坏：整个FS丢失
  - 位图块、 i-node table损坏：无法管理空间和inode
  - 数据块损坏：目录、文件、间址块无法读写

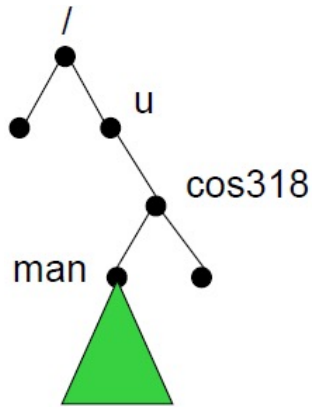
Boot block	Super block	Free space mgmt	I-nodes	Root dir	Files and directories

备份与恢复

- 物理备份与恢复：设备级
  - 将磁盘块逐一拷贝到另一个磁盘上（备份盘）
    - 全复制：原始盘与备份盘在物理上一模一样
    - 增量备份：只拷贝发生变化的块，与上次备份相比
    - 例如：使用dd进行磁盘备份



- 逻辑备份与恢复：文件系统级
  - 遍历文件系统目录树，从根目录开始
  - 把指定的目录和文件拷贝到备份磁盘
  - 在备份过程中验证文件系统结构
  - 恢复时可以将指定的文件或目录树恢复出来
  - 也有两种方式
    - 全备份：备份整个文件系统
    - 增量备份：只备份发生变化的文件/目录



备份与恢复

- 物理备份
  - 忽略文件和文件系统结构，处理过程简洁，备份性能高
  - 文件修改时只用备份修改的数据块，不用备份整个文件
  - 不受文件系统限制
- 逻辑备份
  - 备份文件时，文件块可能分散在磁盘上，备份性能受影响
  - 文件修改时需要备份整个文件
  - 受文件系统限制，按文件粒度备份，保证完整性

内容提要

- 文件系统可靠性
- WAFL
- NFS
- 安全保护

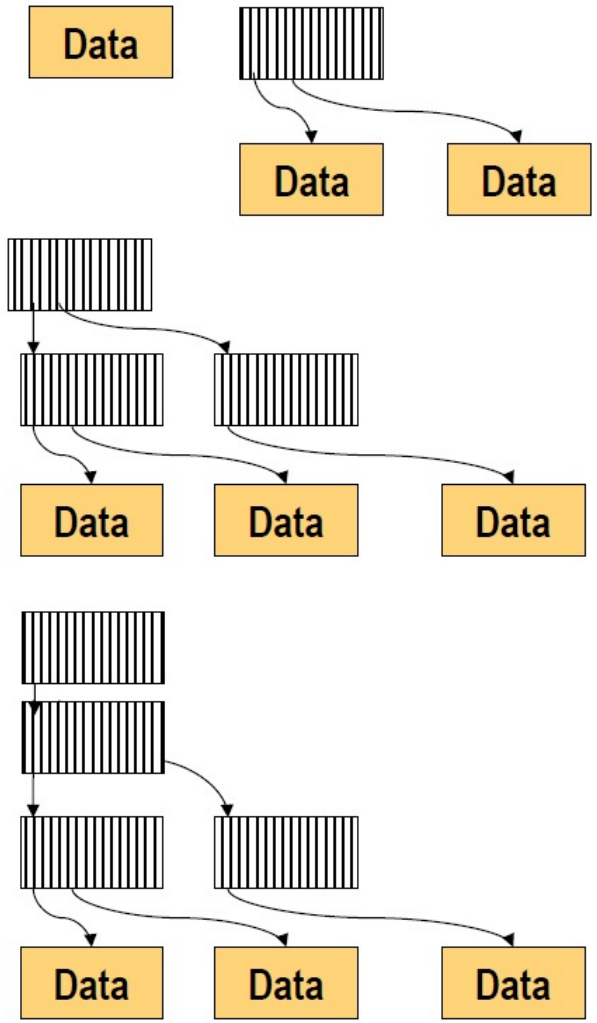
NetApp的企业级文件系统

- WAFL：Write Anywhere File Layout [1]
  - NetApp设计的企业级文件系统
- 设计目标
  - 请求服务速度快：吞吐率(op/s)高，I/O带宽高
  - 支持大文件，且文件系统能不断增长
  - 使用RAID
  - 宕机后快速恢复
- 独特之处
  - 磁盘布局受LFS启发
  - 引入快照
  - 使用NVRAM记录写前日志

[1] D. Hitz, J. Lau, M. Malcolm, **File System Design for an NFS File Server Appliance**, USENIX Winter Conference, 1994

i-node、间址块和数据块

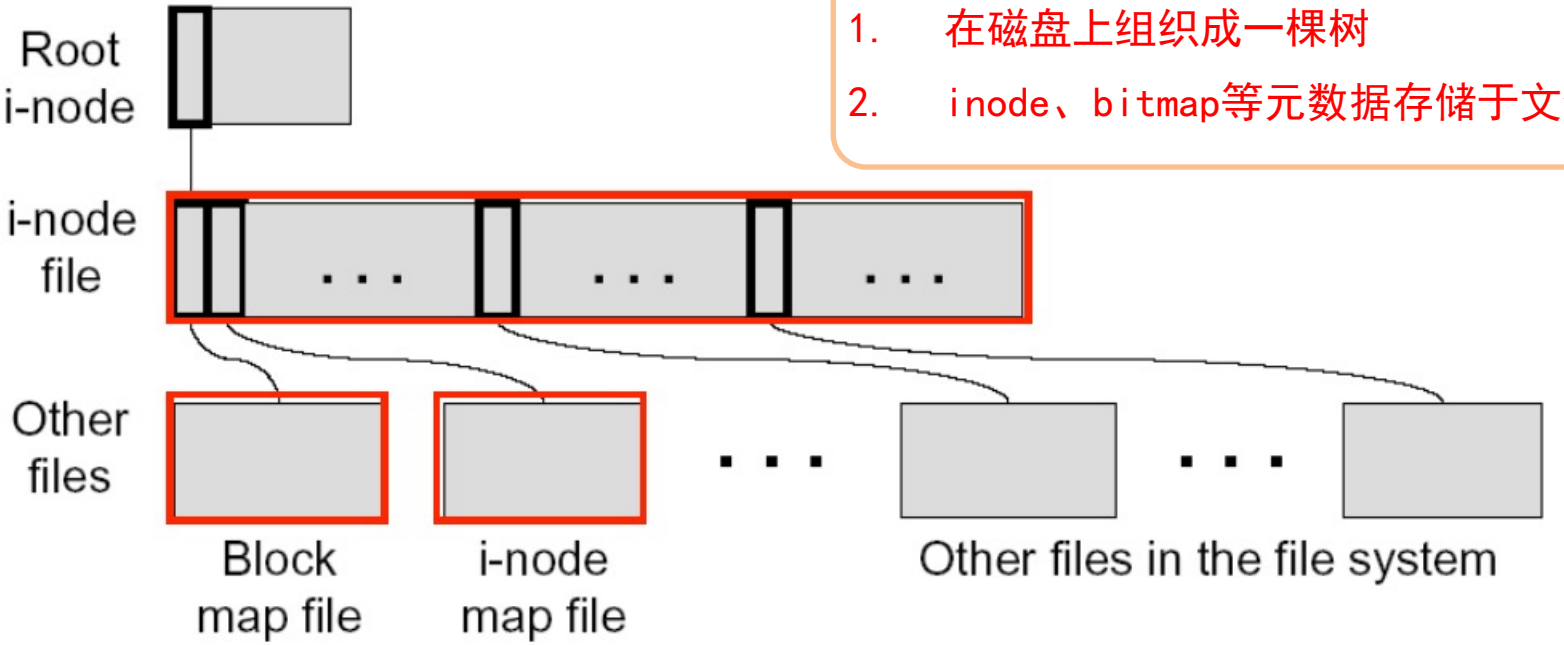
- WAFL使用4KB块
  - i-node: 借鉴UNIX FS
  - 16个指针（64B）用于文件块索引
- 文件大小 ≤ 64B
  - 文件数据直接存储在 i-node中
- 文件大小 ≤ 64KB
  - i-node存储16个指向数据块的指针
- 文件大小 ≤ 64MB
  - i-node存储16个指向间址块的指针
  - 每个间址块存储1024个指向数据块的指针
- 文件大小 > 64MB
  - i-node存储16个指向二级间址块的指针



二级间址能索引的最大文件有多大?

WAFL的磁盘布局

- 主要数据结构
  - 一个根i-node：整个FS的根，位于磁盘上固定位置
  - 一个i-node file：包含所有i-node
  - 一个block map file：指示所有空闲块
  - 一个i-node map file：指示所有空闲i-node



1. 在磁盘上组织成一棵树
2. inode、bitmap等元数据存储于文件中

为什么将元数据存储于文件中

- 元数据块可以写在磁盘上任何位置
  - 这是“WAFL”名字的由来，Write Anywhere File Layout
- 使得动态增加文件系统的大小变得容易
  - 增加一个磁盘会引发i-node个数的增加
  - inode保存在文件，扩展inode文件大小即可
- 能够通过Copy-on-Write(CoW)来创建快照
  - CoW：未写前共享数据，写时拷贝
  - 新的数据和元数据都可以CoW写到磁盘上的新位置
  - 元数据位置固定时无法使用CoW，否则无法定位元数据

对比**FFS**

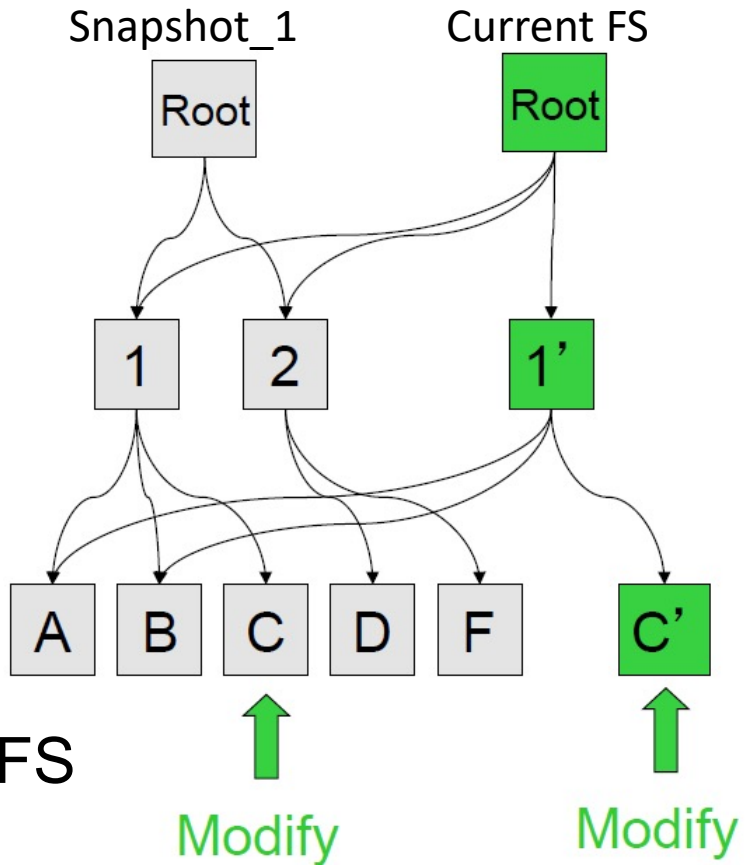
快照 (snapshot)

- 快照是文件系统的一个只读版本
  - 成为文件服务器必备特性
- 快照用法
  - 系统管理员配置快照的个数和频率
  - 用快照可以恢复其中任何一个文件
- 示例

```
% cd .snapshot
% ls
hourly.0  hourly.2  hourly.4  nightly.0  nightly.2  weekly.1
hourly.1  hourly.3  hourly.5  nightly.1  weekly.0
%
```

快照的实现

- WAFL：所有的块构成一棵树
- 创建快照
  - 复制根i-node
  - 新的根i-node用于当前的active FS
  - 旧的根i-node指向快照
- 创建快照之后
  - 第一次写一个块: 把从它到根的数据块都复制(CoW)
  - Active FS的根i-node指向新数据块
  - 后续对这些数据块的写不再触发CoW
- 每个快照都是一个一致状态的只读FS

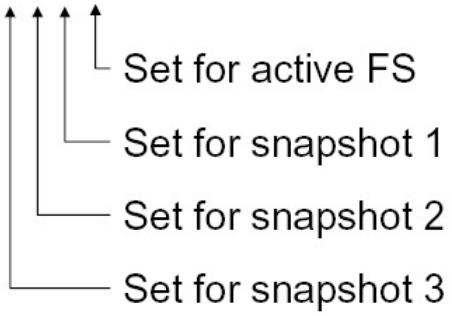


请思考：WAFL快照占用多少额外空间?

快照数据结构

- Block map file
  - 每个4KB磁盘块
    对应一个32-位的表项
  - 表项值为0：
    该块为空闲块
  - 第0位=1：
    该块属于活动文件系统
  - 第1位=1：
    该块属于第一个快照
  - 第2位=1：
    该块属于第二个快照
  - ……

Time	Block map entry	Description
T1	0 0 0 0 0 0 0 0	Block is free
T2	0 0 0 0 0 0 0 1	Active FS uses it
T3	0 0 0 0 0 0 1 1	Create snapshot 1
T4	0 0 0 0 0 1 1 1	Create snapshot 2
T5	0 0 0 0 0 1 1 0	Active FS deletes it
T6	0 0 0 0 0 1 0 0	Delete snapshot 1
T7	0 0 0 0 0 0 0 0	Delete snapshot 2



快照创建

- 问题
  - 创建快照时，除了拷贝根inode，需要把缓存的文件块写回磁盘
  - 此时，可能仍然有很多文件写请求到来
  - 若这些写请求都不处理，会导致文件系统长时间挂起
- WAFL的解决方案
  - 在创建快照前，将缓存中的脏块标记为“in-snapshot”，表示要写回磁盘
  - 所有对“in-snapshot”缓存块的修改请求被挂起
  - 没有标记为“in-snapshot”的缓存数据可以修改（即处理写请求），但不能写回磁盘
  - 本质：区分需要被写回的脏块和其他块，减少挂起的写请求数量

创建快照

- 步骤
  - 为所有“in-snapshot”的缓存块分配磁盘空间
    - 包括数据、inode
  - 更新 block map file
    - 对每个表项，将Active FS位的值（即1） 拷贝到 新快照位
  - 刷回
    - 把所有“in-snapshot” 缓存块写到它们新的磁盘位置
    - 每写回一个块，重启它上面被挂起文件请求
  - 复制根i-node
- 性能较快

快照删除

- 删除快照的根i-node
- 清除block map file中的位
  - 对于block map file的每一个表项，清除与该快照对应的位

文件系统宕机一致性

- 定期创建一致点
  - 一致点：存储控制器中使用NVRAM缓存的数据被刷回磁盘，并更新了文件系统中相应的指针
  - 每10秒创建一个一致点
  - 特殊的内部快照，用户不可见
- 在一致点之间的多个请求
  - 第i个一致点
  - 若干写操作
  - 第i+1个一致点（自动增长）
  - 若干写操作
  - ……
- 宕机恢复
  - 将文件系统恢复到最后一个一致点
  - 最后一个一致点之后到宕机前的写操作：靠日志进行恢复

非易失RAM（Non-Volatile RAM）

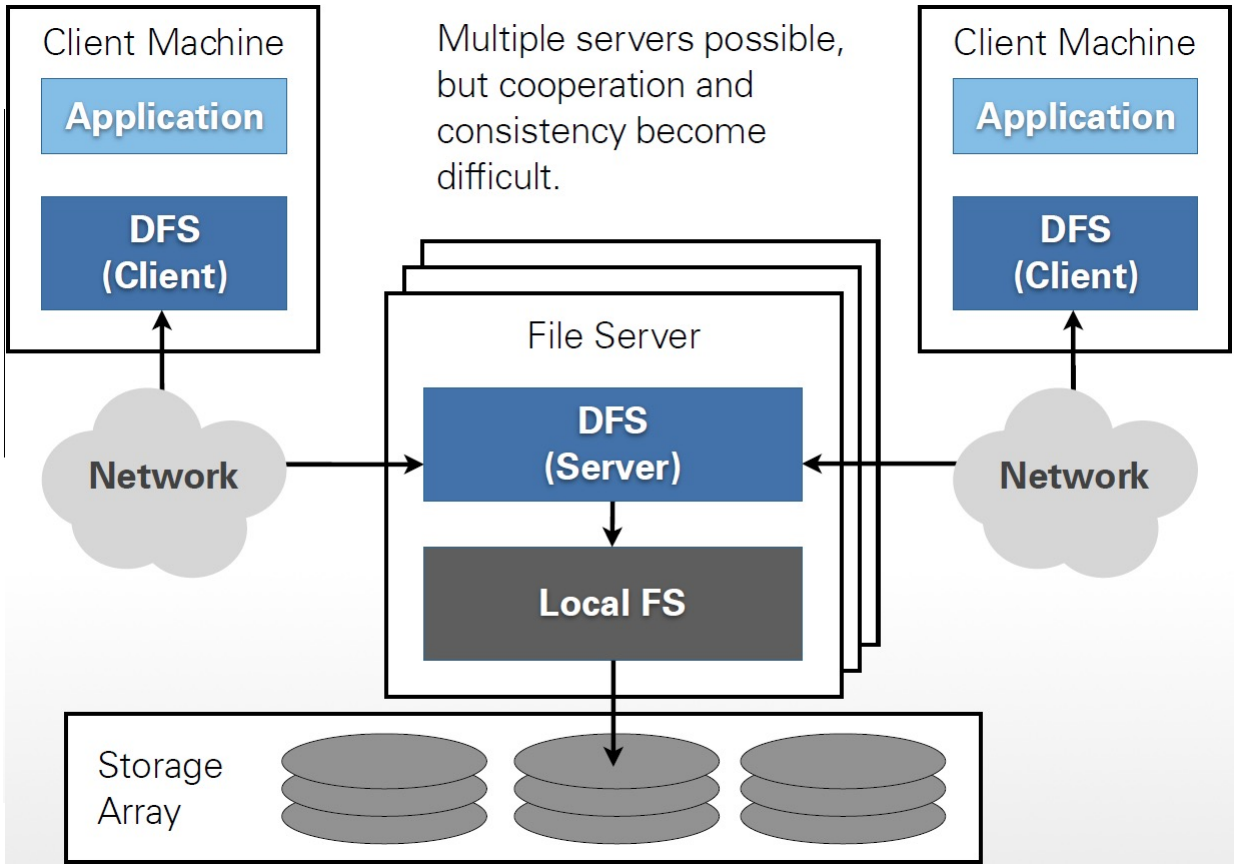
- NVRAM
  - 闪存：写比较慢 vs. NVRAM
  - 带电池的DRAM：快
    - 电池容量有限，持续时间不长
    - DRAM容量有限
- 日志写入NVRAM
  - 记录自上一个一致点以来的所有写请求
  - 宕机恢复：用NVRAM中的日志来恢复从最后一个一致点以后的修改
- 使用两个日志
  - 一个日志写回磁盘时，另一个日志写入NVRAM中缓冲
  - 可以避免写日志时，无法处理新的写请求

内容提要

- 文件系统可靠性
- WAFL
- NFS
- 安全保护

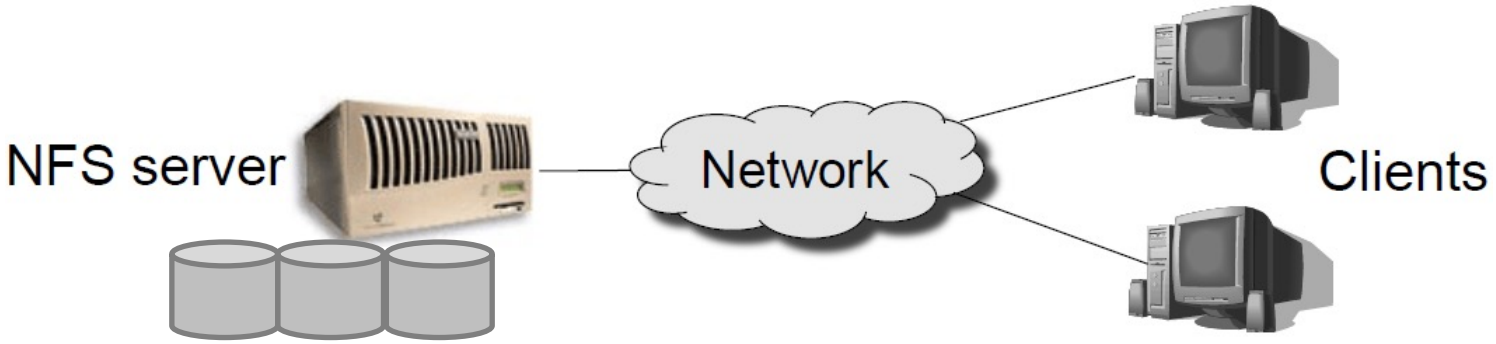
远程文件访问

- 如何访问通过网络连接的另一台计算机上的文件？



NFS：Network File System

- 多个客户端（计算机）共享一台文件服务器



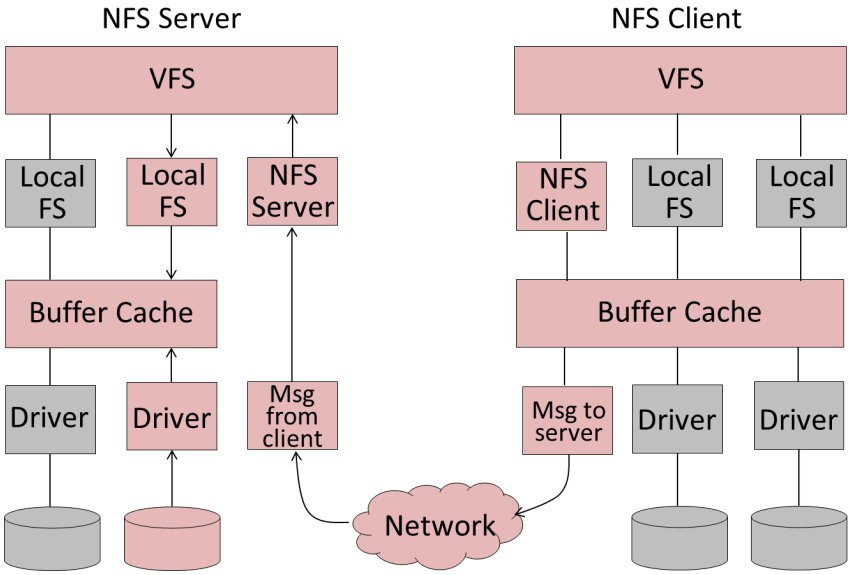
- 发展历程
  - NFS：SUN， 1985年
  - 开放协议：IETF标准
  - NFS v2：1989年，IETF RFC 1094
  - NFS v3[1]：1995年，RFC 1813[2]
  - NFS v4：2000年RFC 3010，03年RFC 3530，15年RFC 7530

[1] B. Pawlowski, C. Juszczak, P. Stauback, et al. NFS Version 3: Design and Implementation. USENIX Summer Conference, 1994

[2] B. Callaghan, B. Pawlowski, P. Staubach. **NFS Vesion 3 Protocol Specification**. RFC 1813, 1995 https://ietf.org/rfc/rfc1813.txt

NFS架构

- 多Client，单Server
  - NFS客户端（内核）：实现FS功能接口
    - 接口: syscall, 与本地FS相同接口 (透明性)
    - 把文件访问syscall转换成请求
    - 把请求发给服务器
    - 接收服务器发回的结果,
      并返回给调用者
  - NFS服务器（内核）：提供文件服务
    - 接收客户请求
    - 读写本地FS
    - 把结果发回给客户端
  - 缓存
    - 客户端缓存
    - 服务器端缓存



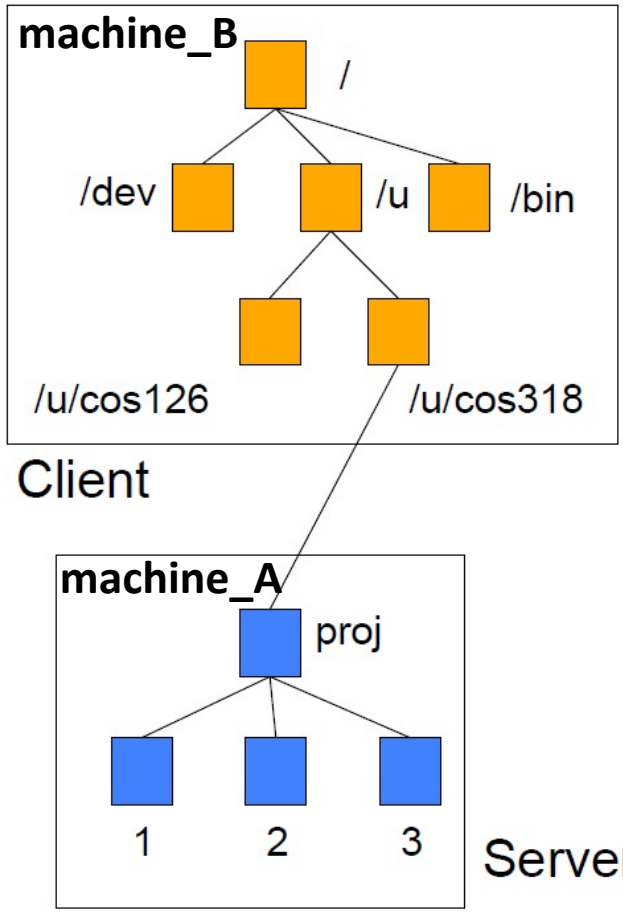
NFS设计

- 设计目标
  - 简单
  - 快速回复
- 核心思想（NFS v3）：**无状态**服务器 (stateless)
  - 服务器不记录客户端打开的文件
  - 服务器不记录每个打开文件的当前偏移
  - 服务器不记录被客户端缓存的数据块
- 核心数据结构：File Handle (FH)
  - 唯一标识客户端要访问的文件或目录
  - Volume ID
  - ino
  - Generation number

NFS挂载（mount）

- NFS服务器“export”一个目录给客户端
  - 输出目录表：/etc/exports
  - 输出目录命令：exportfs

```
# cat /etc/exports
/proj    machine_A(rw)
```



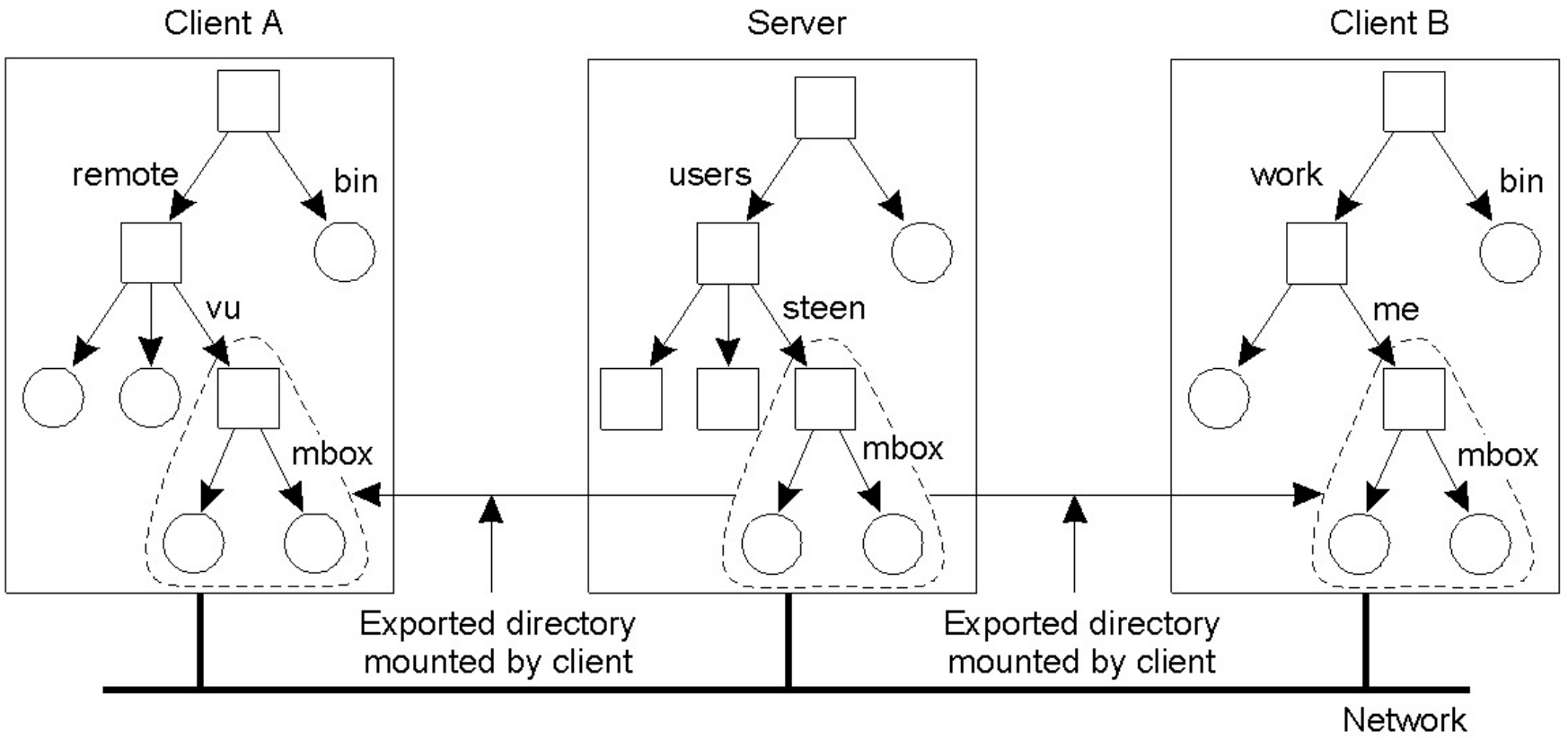
- NFS客户端挂载 (mount)
  - NFS服务器 (机器名或网络地址)
  - NFS服务器输出目录的路径名
  - 挂载点：本地目录的路径名

```
$ ls /usr/cos318
$ mount -t nfs  machine_A:/proj  /u/cos318/proj
$ ls /usr/cos318/proj
```

  - 服务器返回输出目录的File Handle

例子

- 两个客户端挂载同一个服务器输出的目录



- 三台机器共享一棵子树

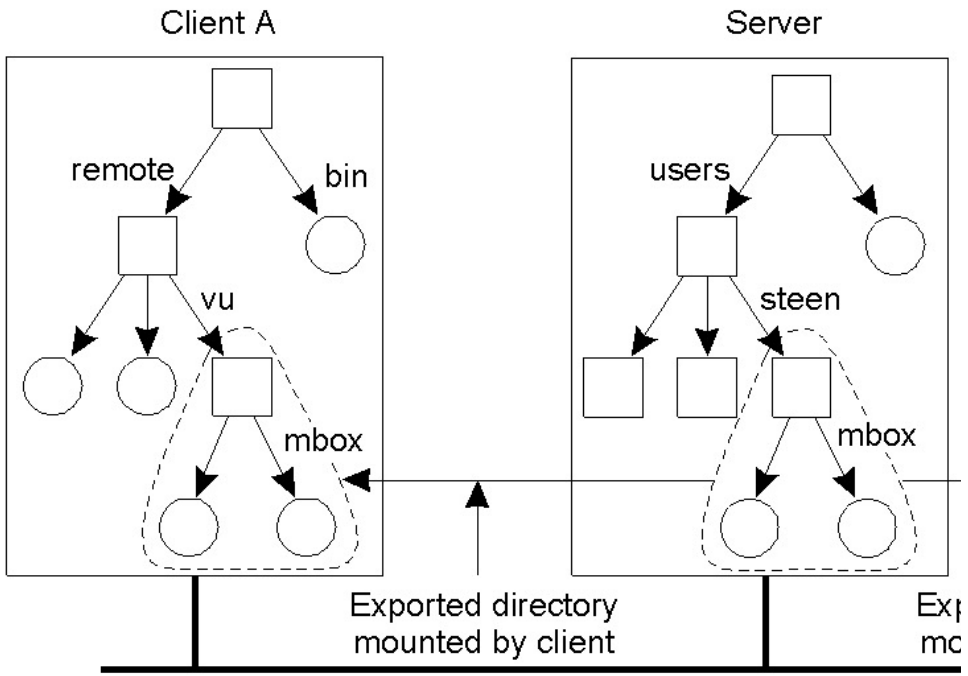
文件访问的实现原理

挂载记录为NFS，所以VFS会通过NFS发送请求

- 客户端open一个文件
  - open() syscall: 路径解析
  - 向服务器发LOOKUP请求
  - 接收服务器应答的FH
  - 将本地fd与FH关联
- 客户端read文件
  - read() syscall: fd, buf, count
  - 根据fd得到FH和偏移
  - 向服务器发READ请求
  - 参数为FH, 偏移, count
  - 接收服务器的应答数据
  - 把应答数据拷贝到buf
  - fd的偏移 += count
- 客户端close文件
  - 释放fd与打开文件结构
  - 无需与服务器交互

```
# mount –t nfs Server:/users/steen /remote/vu

fd = open ("/remote/vu/mbox", O_RDONLY);
n = read (fd, buf, 16384);
```

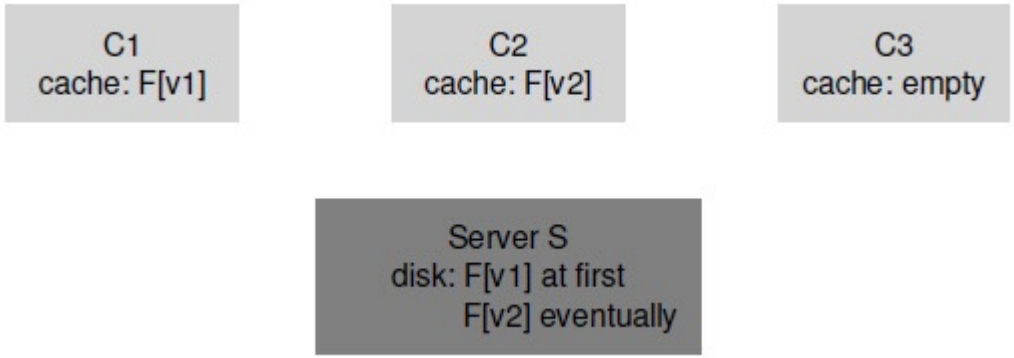


文件访问的实现原理

- 客户端open一个文件/目录
  - open() syscall: 路径解析
  - 向服务器发LOOKUP请求
  - 接收服务器应答的FH
  - 将本地fd与FH关联
- 客户端read文件
  - read() syscall: fd, buf, count
  - 根据fd得到FH和偏移
  - 向服务器发READ请求
  - 参数为FH, 偏移, count
  - 接收服务器的应答数据
  - 把应答数据拷贝到buf
  - fd的偏移 += count
- 客户端close文件
  - 释放fd与打开文件结构
  - 无需与服务器交互
- 服务器接收LOOKUP请求
  - 从目录FH中得到VID和目录ino
  - 读目录 i-node
  - 读目录块, 查找与name匹配的目录项 <name, ino>
  - 构造FH: VID, name的ino, gno
  - 发应答: name的FH
- 服务器接收READ请求
  - 从FH中得到VID和文件ino
  - 打开本地文件ino得到sfd
  - 设置本地文件的偏移 (lseek)
  - 读本地文件数据到sbuf中 read(sfd, sbuf, count)
  - 关闭本地文件close(sfd)
  - 发应答: sbuf的数据

客户端缓存

- 客户端用一部分内核内存来缓存数据和元数据
- 好处
  - 提高文件读写性能：减少与服务器的交互（网络 & 磁盘I/O）
- 缓存一致性问题
  - 当多个客户端同时读写同一个文件



  - 当某个客户端写，导致

1. 修改不可见：客户端C3打开文件时读到旧版本 (服务器不是最新版本)
2. 陈旧数据：客户端C1缓存中是旧数据

客户端缓存一致性问题

- NFS v3的解决办法
  - Close-to-open consistency：应对修改不可见
    - Flush-on-close：关闭文件时，客户端将所有更新刷回服务器端
  - 应对缓存陈旧数据
    - open时用GETATTR获取文件的mtime，对比检查缓存数据的有效性
    - 数据块 (文件/目录) 60s过期，属性缓存3s过期
  - 并不能解决并发多写导致的数据破坏问题
- NFS v4演进
  - 基于网络的文件锁管理 （ NLM v4)：应对多写
    - 只允许有一个写者：one writer or N readers

服务器端缓存

- 服务器用一部分内核内存来缓存数据和元数据
- 好处
  - 提高文件读写性能：服务器端减少磁盘I/O
- 问题
  - 服务器宕机可能丢数据
- 解决办法 (NFS v3)
  - COMMIT
    - 服务器收到COMMIT后，把之前WRITE写在缓存中的数据写到持久化存储
    - 参数：FH, 偏移, count
    - 如果COMMIT超时未收到应答
      - 之前的WRITE和COMMIT本身都要重发

内容提要

- 文件系统可靠性
- WAFL
- NFS
- 安全保护

一些简单的攻击

- 提权攻击
  - UNIX/Linux：root能做任何事情
    - 例如：读写任意文件
  - 普通进程提权后，就能攻击系统
- 拒绝服务（DoS）
  - 耗尽系统所有资源
  - 例如
    - 运行一个shell脚本： “while (1) {mkdir foo; cd foo; }
    - 运行一个C程序： “while (1) { fork(); malloc (1000)[40]=1;}
- 偷听
  - 侦听网络上传输的包

安全与保护

- 数据机密性：未经许可，不能看到数据
  - 任何用户不能读写其他用户的文件，可以采用加密
- 数据完整性：未经许可，不能修改或删除数据
  - 数据在网络传输过程中被拦截和修改
- 系统可用性：干扰系统使得它不可用
  - 给一个服务器发送大量的请求

Goal	Threat
Data confidentiality	Exposure of data
Data integrity	Tampering with data
System availability	Denial of service

保护：策略与机制

- 安全策略：定义目标，即要达到的效果
  - 通常是一组规则，定义可接受的行为和不可接受的行为
  - 例子
    - /etc/password文件只有root能写
    - 每个用户最多只能用50GB的磁盘空间
    - 任何用户都不允许读其他用户的mail文件
- 机制：用什么样的方法来达到目标

保护机制

- Authentication（身份认证）
  - 验明身份：密码
- Authorization（授权）
  - 决定“A是不是准许做某件事”
  - 通常使用角色（role）定义授予的操作权限
- Admission control（访问控制）
  - 做出“访问是否准许”的决定

身份认证

- 通常是用密码来验证
  - 一串字符（字母+数字）
  - 用户必须记住密码
- 密码是以加密形式存储
  - 使用一种单向的“安全hash”算法
- 缺点
  - 每个用户都要记很多密码
  - 弱密码风险，“dictionary attack”

LOGIN: ken
PASSWORD: FooBar
SUCCESSFUL LOGIN

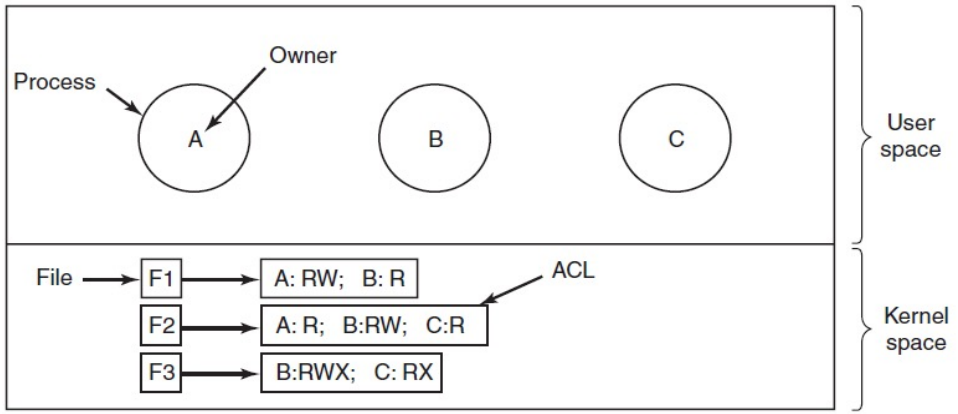
LOGIN: carol
PASSWORD: Idunno
INVALID LOGIN
LOGIN:

授权：Capabilities（权能）

- 超级用户具有特权，可以执行高权限操作
  - 例如passwd、chown、chmod等
- 权能：将超级用户特权细分，分成不同的、细粒度权限
  - 例如
  - CAP_CHOWN：对文件UIDs 和 GIDs 做修改
  - CAP_KILL：绕过发送信号时的权限检查
  - CAP_NET_ADMIN：执行多种网络有关的操作
- 可以为每个线程独立设置权能
- 实现
  - 权能表保存在内核

访问控制表（ACL）

- 每个文件对象有一个ACL表
  - 定义每个用户的权限
  - 每个表项为 <user, privilege>
- 大多数系统都采用
  - UNIX的owner, group, other
- 实现
  - ACL实现在内核中
  - 在登录系统时进行身份验证
  - ACL存储在每个文件元数据中
  - 打开文件时检查ACL



总结

- WAFL
  - 支持在任意位置上写的磁盘布局（受LFS的影响）
  - 基于CoW实现快照
- NFS
  - 网络文件系统
  - 客户端和服务器端通过File Handle交互
  - 客户端和服务器都有缓存，客户端缓存一致性问题
- 操作系统安全
  - 数据机密性、数据完整性、系统可用性
  - 基于密码的身份认证
  - ACL和Capability
  - 安全是一个系统性工程