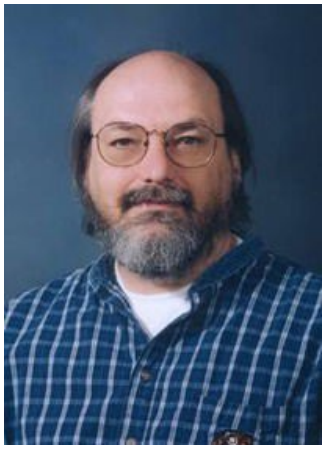
Persistence (II)
File System - 2
(OSTEP Section 41,42)

Outline

- Linux I/O Stack
- File System
- File System Implementation
- Fast File System
- FSCK and Journaling
- Log-structured File Systems
- Data Integrity and Protection

Fast File System

- **Old Unix File System**
  - **Ken Thompson (UNIX wizard)**
  - **Super block – inode – data**



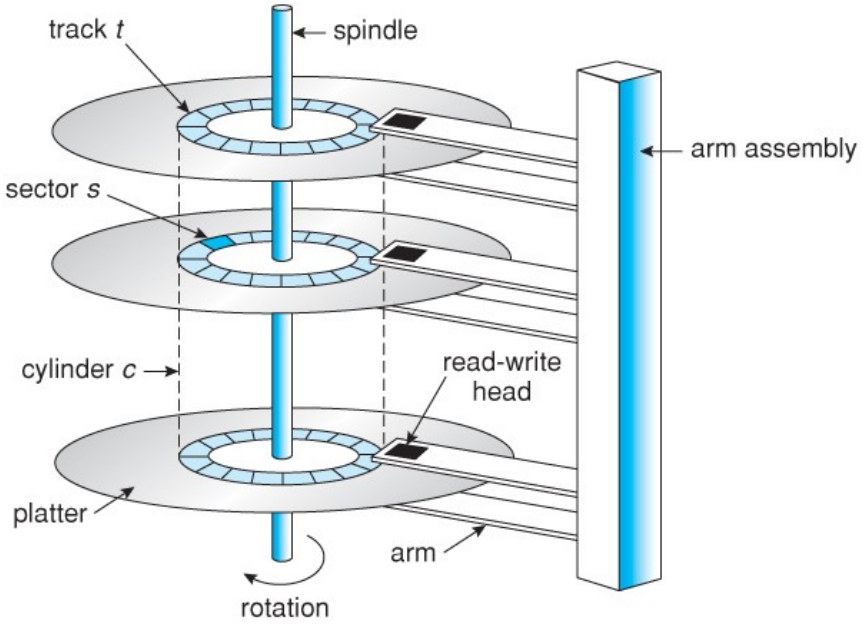
  - **Poor performance**
    - **吞吐量相当于磁盘 raw i/o 的 2%！！！**
    - **因为把磁盘当做了随机访问的内存，没有针对性的优化**

Fast File System

- **Old Unix File System**
  - 性能不好的另一个原因： **quite fragmented**
    - **Free space** 管理过于简单
    - 同一个文件的数据分散，磁头来回移动
    - 例如， **ABCD** 四个文件， **BD** 删除后，写入新文件 **E**

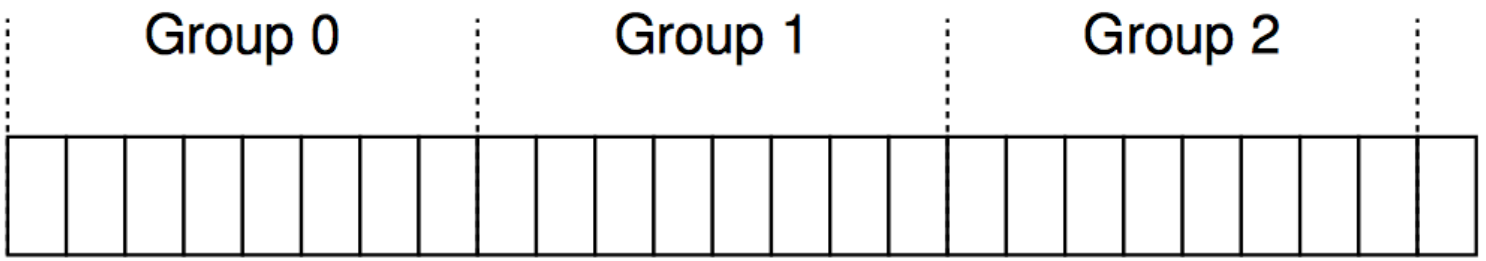
Fast File System (FFS)

- **FFS: Disk-aware**
  - **Cylinder Group： FFS 基于 CHS addressing，将连续（相邻）的 N 个 cylinder 看做一个 cylinder group**



Fast File System

- **FFS: Disk-aware**
  - **Cylinder**
    group 也可以看做是一组连续的 Block/sector
    - 例如， 8 个 block 组成一个 group （下图）
    - 访问同一个 group 中的 block 比较快，因为磁头移动较少



Data Locality of Reference

- 数据访问的局部性 (locality) 假设：
  - 数据中的某些部分之间存在某种联系，这些相关联的部分会经常在一段时间内被一起访问。例如：
    - FS 中经常按照 super block->bitmaps->inode->data blocks 的顺序访问块设备
    - 同一个目录下的多个文件经常在较短的时间内被访问
    - 同一个文件中“距离”较近的 data blocks 经常被连续访问

怎么利用 Cylinder Group 来优化性能?

Fast File System

- **FFS: Disk-aware**
  - **cylinder group:**
    - **每个 group 都包括 super block，可以并发访问**
    - **也包括组内的 inode bitmap, data bitmap，和 inode table**
    - **文件的元数据和数据放在临近的位置，访问会比较快**

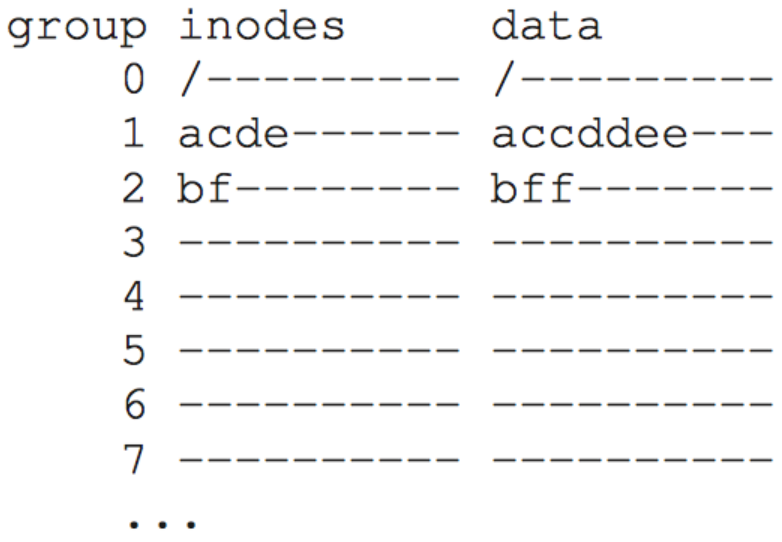
Fast File System

- 核心问题：如何放置文件到 cylinder group？
  - *“keep related stuff together, keep unrelated stuff far apart.”*
    - 但 FS 中数据是动态变化的
  - 对于目录
    - 将新目录放在一个目录比较少、 free inodes 比较多的 cg 中
  - 对于文件
    - 同一个文件的 inode 和 data blocks 放在同一个 group
    - 同目录下的文件与目录放在同一个 group

System 优化的两个要点：
1）适应硬件特点（尤其是重点考虑缺点）；
2）适应上层应用的特征（workload）

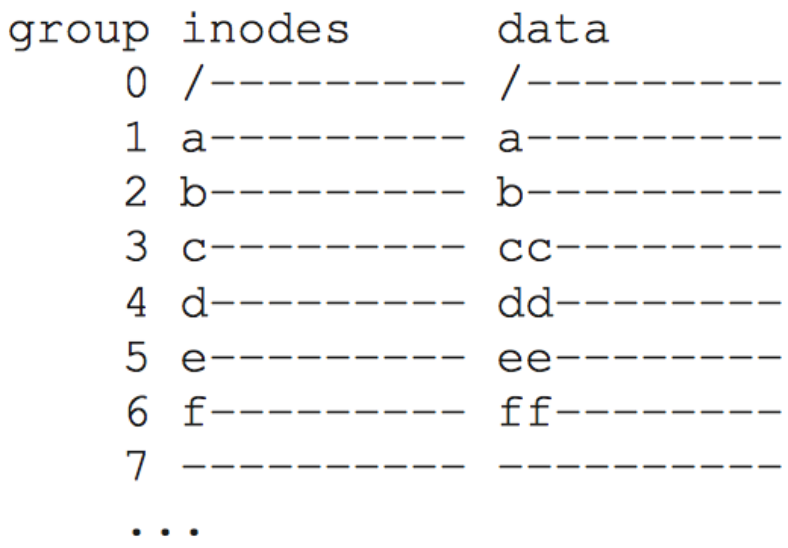
Fast File System

- 同目录下的文件与目录放在同一个 group
  - 例如：假设每个 group 包括 10 个 inode， 10 个 data block
  - 目录有 3 个： /, /a, /b
  - 文件有 4 个： /a/c, /a/d, /a/e, /b/f
  - FFS 放置方案



Fast File System

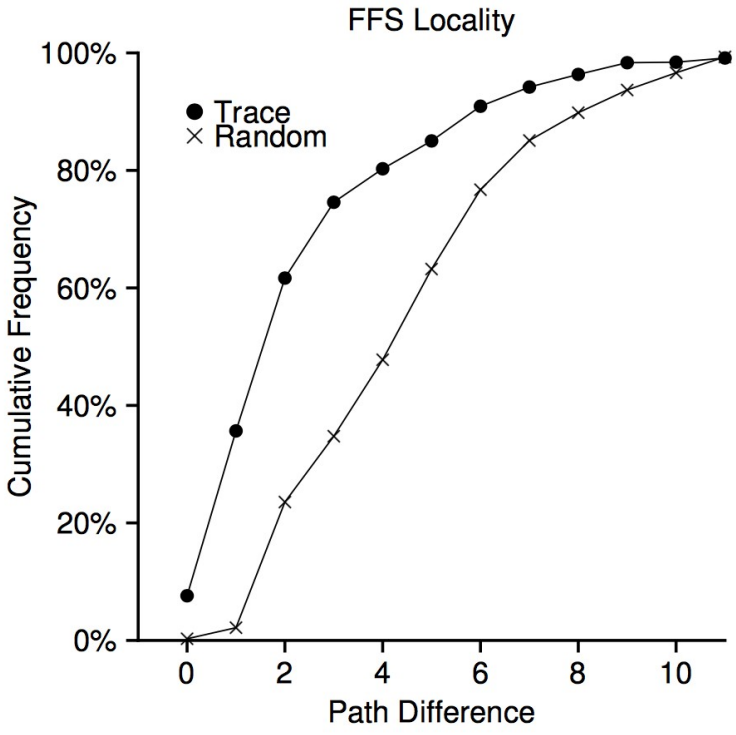
- 对比：简单的放置策略



两种策略下，访问同一个目录中的文件，分别需要访问多个group？

Fast File System

- **Measuring File Locality**
  - 连续两次打开文件的目录深度统计
    - 例如先后打开 /dir/f, /dir/g，距离（path difference）为 1
  - 结论：多数连续文件访问的目录深度差距不大，局部性好
    - 40% 是同一个文件或同一个目录的文件
    - 25% 是两级目录以内
    - 所以， FFS 的局部性假设成立

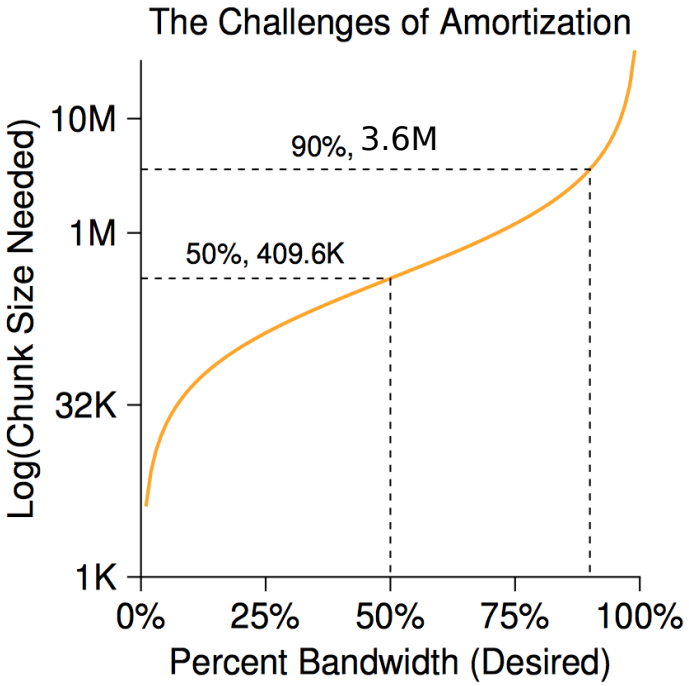


Fast File System

- **The Large File Exception**
  - 不使用 large file exception
    - /a 占用了大部分 data blocks， / 下没有空间用来创建新文件
```
group inodes     data
    0 /a-------- /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a---------
    1 ---------- ---------- ---------- ---------- ----------
    2 ---------- ---------- ---------- ---------- ----------
    ...
```
  - 解决办法：把大文件的 data blocks 分散到多个不同的 ...
```
group inodes     data
    0 /a-------- /aaaaa---- ---------- ---------- ----------
    1 ---------- aaaaa----- ---------- ---------- ----------
    2 ---------- aaaaa----- ---------- ---------- ----------
    3 ---------- aaaaa----- ---------- ---------- ----------
    4 ---------- aaaaa----- ---------- ---------- ----------
    5 ---------- aaaaa----- ---------- ---------- ----------
    6 ---------- ---------- ---------- ---------- ----------
    ...
```

Fast File System

- **The Large File Exception**
  - 问题：数据分散，性能下降
  - 解决：合理设置 chunk 大小，获得“足够好“的性能
    - 50% 带宽： 409.6KB
    - 90% 带宽： 3.6MB
    - 99% 带宽： 39.6MB



计算过程：
设 disk sequencial read bandwidth=40MB/s , 即 40.96KB/ms, disk positioning time=10ms, 则：
target_ratio=(10ms * 40.96KB/ms) / chunk_size

target_ratio=1/9 时，chunk_size=3.6MB

代价分摊（Cost Amortization）！

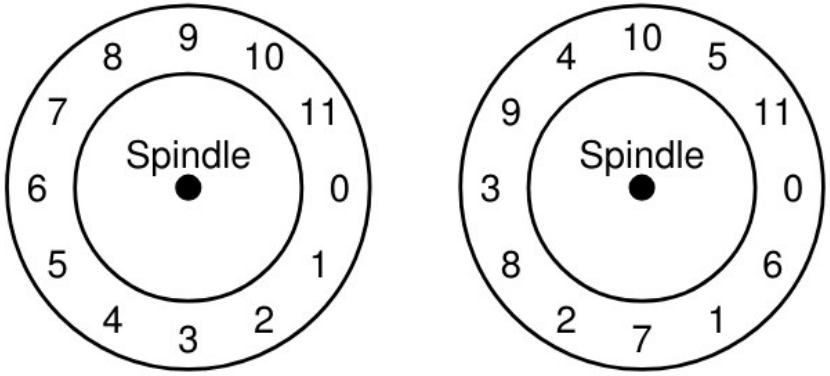
Another Example of Cost Amortization

- **HDFS (Hadoop Distributed File System):**
  - 每个大文件（如 **100GB**）拆分成多个数据块（**block**）
  - **blocks** 分散存储在不同的机器上
  - **client** 找到并打开一个 **block** 需要一定的时间
  - 如何选择 **block** 大小，保证顺序读取一个大文件的效率?

Fast File System

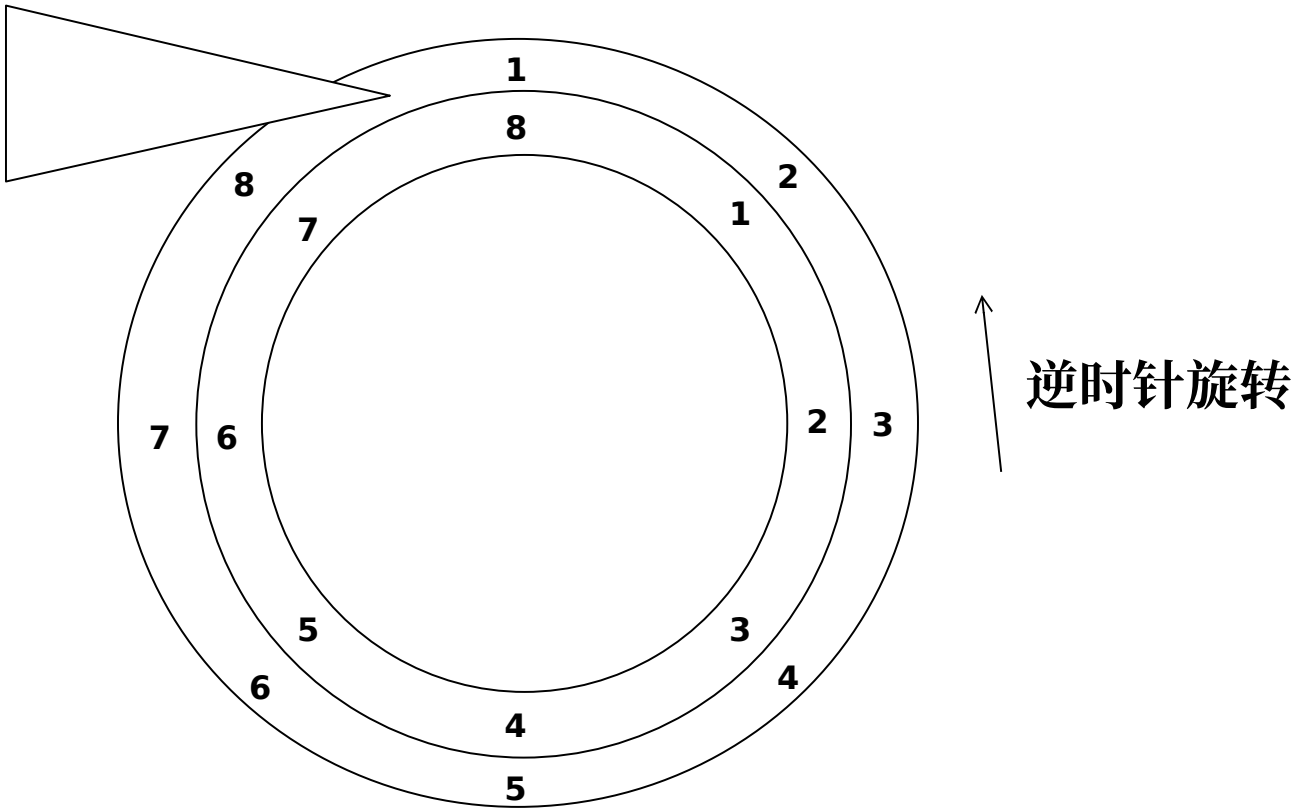
- 小文件的问题（<2KB）
  - internal fragmentation
  - 无法充分利用 4KB 的 data block，空间浪费
  - 解决： sub-block (512B)
    - 先以 sub-block 为单位分配空间
    - 文件空间如果增长超过 2KB，则迁移到一个完整的 block 上
    - 新的问题：额外的数据拷贝
      - 解决：修改 libc，缓冲写数据，再分配到 4KB 的块上

Reading the next block



- **FFS: 间隔放置 data blocks，这会浪费磁盘带宽**
- **更好的解决方案： read-ahead**
  - **可以由 disk， block I/O layer，或者 FS 完成**

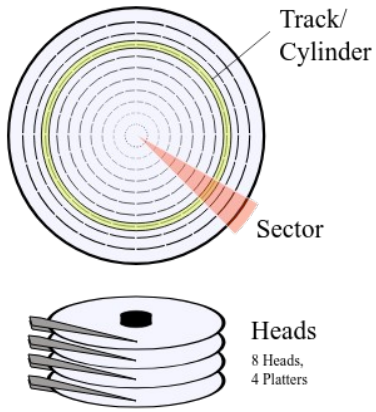
Reading the next cylinder



磁臂 (arm) 从外圈 cylinder 向内圈 cylinder 移动，至少需要个 settle time，如 2ms，为了使 arm 移动到内圈 cylinder 时磁头 (head) 刚好读到第一个 sector，内圈第一个 sector 需要错后一些

Seek time in a FS backed by HDD

- 磁盘（HDD）在 FS 看来是一个一维 Block 向量
- FS 中的 seek 实际是从 block 向量中的一个 position 调到另一个 position
- FS 中的 seek time 如何计算?



如果两个 data block 在同一个，只有 spin time（对于 7200rpm 磁盘， 0-8.33ms）
如果两个 data block 相距 2-4 个 cylinder， seek time=spin time+settle time
如果两个 data block 相距稍远一点？
如果两个 data block 相距很远?

总体上， seek time 与 seek distance 正相关

How to place data in a file?

- Use a file system that supports extent (e.g., Ext4, XFS)
  - Extent ensures a large byte range in the file is store sequencially in the disk
- Put frequently co-accessed data chunks closer in a file

Outline

- Linux I/O Stack
- File System
- File System Implementation
- Fast File System
- FSCK and Journaling
- Log-structured File Systems
- Data Integrity and Protection

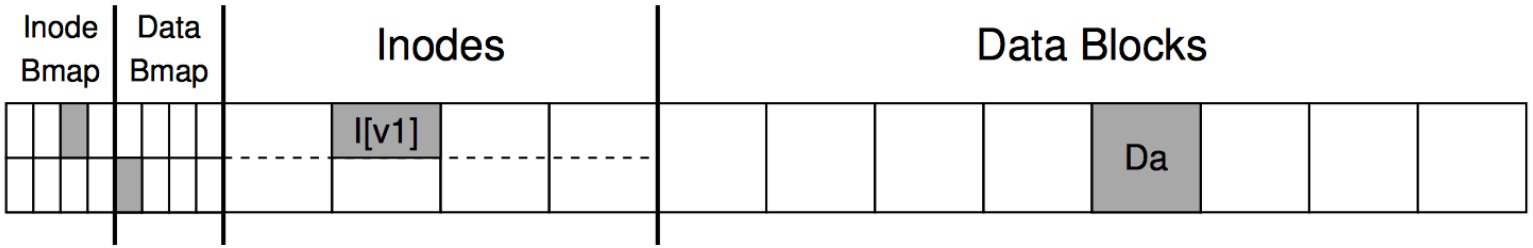
FSCK and Journaling

- 可容忍的系统状态：
  - 完全正常、完全失效（e.g., RAM）
  - 部分失效【存储 / 数据库、分布式系统】
- **Crash-consistency problem**
  - 例如连续写入高度关联的 A 和 B，在中间发生 crash
    - 原子性问题
  - 系统进入一个 inconsistent 状态
  - 解决方法
    - **FSCK: file system checker**
    - **Journaling: write-ahead log**

FSCK and Journaling

- 一个例子
  - 一个文件大小为 1，只包括一个数据块
    - inode 部分域:

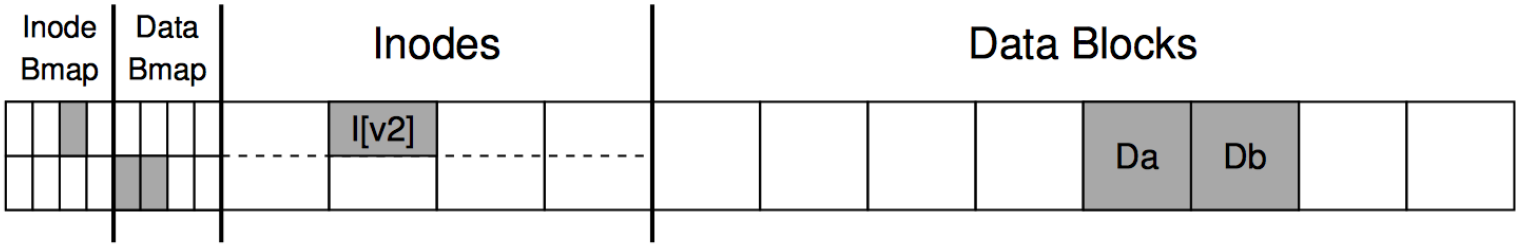
```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```



FSCK and Journaling

- 一个例子
  - 后面文件又追加写入一个 **block Db**
  - **inode** 部分域:

```
owner       : remzi
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```



FSCK and Journaling

- 三个具体的写操作
  - **Bitmap**
  - **Inode**
  - **Data block (Db)**
- 先写入 **page cache**，延迟写入磁盘
  - 在写盘之前计算机 **crash**，有多种可能:

有几种 inconsistent 的可能?

FSCK and Journaling

- **#1. Just the data block (Db) is written to disk**
  - 文件的数据在磁盘上
  - 但是 inode 没有记录， bitmap 也没有
  - 不是问题:
    - 相当于 Db 没有写入，不会引入不一致的问题

FSCK and Journaling

- **#2. Just the updated inode (I[v2]) is written to disk**
  - **问题 1： 按照 inode 信息，会从 data block 5 位置读出垃圾数据**
  - **问题 2： file-system inconsistency**
    - **Inode 告诉我们 data block 5 有数据**
    - **Bitmap 告诉我们 data block 5 没有数据**

FSCK and Journaling

- **#3. Just the updated bitmap (B[v2]) is written to disk**
  - **还是 bitmap 和 inode 不一致**
  - **Space leak**
    - **Data block 被占用，但是不会被利用**

FSCK and Journaling

- **#4. inode (I[v2]) and bitmap (B[v2]) written, but not data (Db)**
  - 元数据一致，但是数据是垃圾数据

FSCK and Journaling

- **#5. inode (I[v2]) and data block (Db) written, but not the bitmap (B[v2])**
  - 读取正确，但数据可能被覆盖

FSCK and Journaling

- **#6. bitmap (B[v2]) and data block (Db) written, but not the inode (I[v2])**
  - 元数据不一致
  - 无法读取 db

FSCK and Journaling

Crash 不可避免

- **一致性问题本质**
  - **完成一个文件写操作，需要多次 device 写操作**
  - **这些 device 写操作应该具有原子性**
    - **要么都成功，要么都不成功**

FSCK and Journaling

- **Solution #1: The File System Checker**
  - **Fsck – UNIX tool**
  - **扫描所有的 block，尽最大努力修复文件系统**
  - **目标：让元数据及数据恢复一致**
  - **在文件系统被 mount 之前检查**
    - **避免 fsck 的同时， fs 修改数据和元数据**

FSCK and Journaling

- **Solution #1: The File System Checker**
  - **Fsck 检查什么?**
    - **Superblock:**
      - **检查 superblock 看起来是不是正常**
        - » 如果不正常，则通过备份恢复
        - » 恢复不了怎么办?

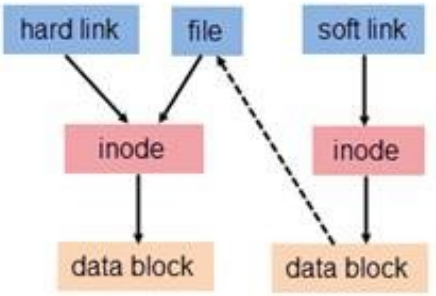
FSCK and Journaling

- **Solution #1: The File System Checker**
  - **Fsck 检查什么?**
    - **Free blocks:**
      - **检查 inode, indirect blocks, double indirect blocks, 看数据在磁盘上如何分布;**
      - **然后基于这些信息去纠正 bitmaps 和 inodes 之间不一致的问题**
        - **» 如果不一致, 则以 inodes 为准（凭啥?）**

FSCK and Journaling

- **Solution #1: The File System Checker**
  - **Fsck 检查什么?**
    - **Inode state:**
      - **检查每个 inode 看起来是否正常;**
      - **如果错误纠正不了，就会删除该 inode，同时更新 inode bitmap**

FSCK and Journaling



- **Solution #1: The File System Checker**
  - **Fsck 检查什么?**
    - **Inode links:**
      - **inode 的 link count 属性记录了有多少个目录包含指向该文件的 name-inode mapping**
      - **扫描整个目录树，重新计算 link count，纠正 inode 记录的 link count；**
      - **如果一个 inode 的 link count 是 0，则将其放到 lost+found 目录下**

FSCK and Journaling

- **Solution #1: The File System Checker**
  - **Fsck 检查什么?**
    - **Deduplicated pointers:**
      - **检查是否存在多个不同的 Inode 指向同一个 data/pointer block ;**
        - **» 如果其中某个 Inode 本身不正确，则将其清除**
        - **» 对于剩余的 inode ， 复制 block**

FSCK and Journaling

- **Solution #1: The File System Checker**
  - **Fsck 检查什么?**
    - **Bad block pointers:**
      - **检查 inode 和 pointer block 中的 block pointers 是否有问题，例如指向一个越界的 block**
        - » **从 inode 或 pointer block 中清除这些 bad pointers**

FSCK and Journaling

- **Solution #1: The File System Checker**
  - **Fsck 检查什么?**
    - **Directory Check:**
      - **directory 有自己的 data block，其中的数据结构是由 FS 自己定义的，因此可以被 fsck 理解**

FSCK and Journaling

- **Solution #1: The File System Checker**
  - **FSCK 的问题:**
    - **扫描文件系统中的全部 blocks，太慢了!**
    - **有一些不一致问题无法修复**

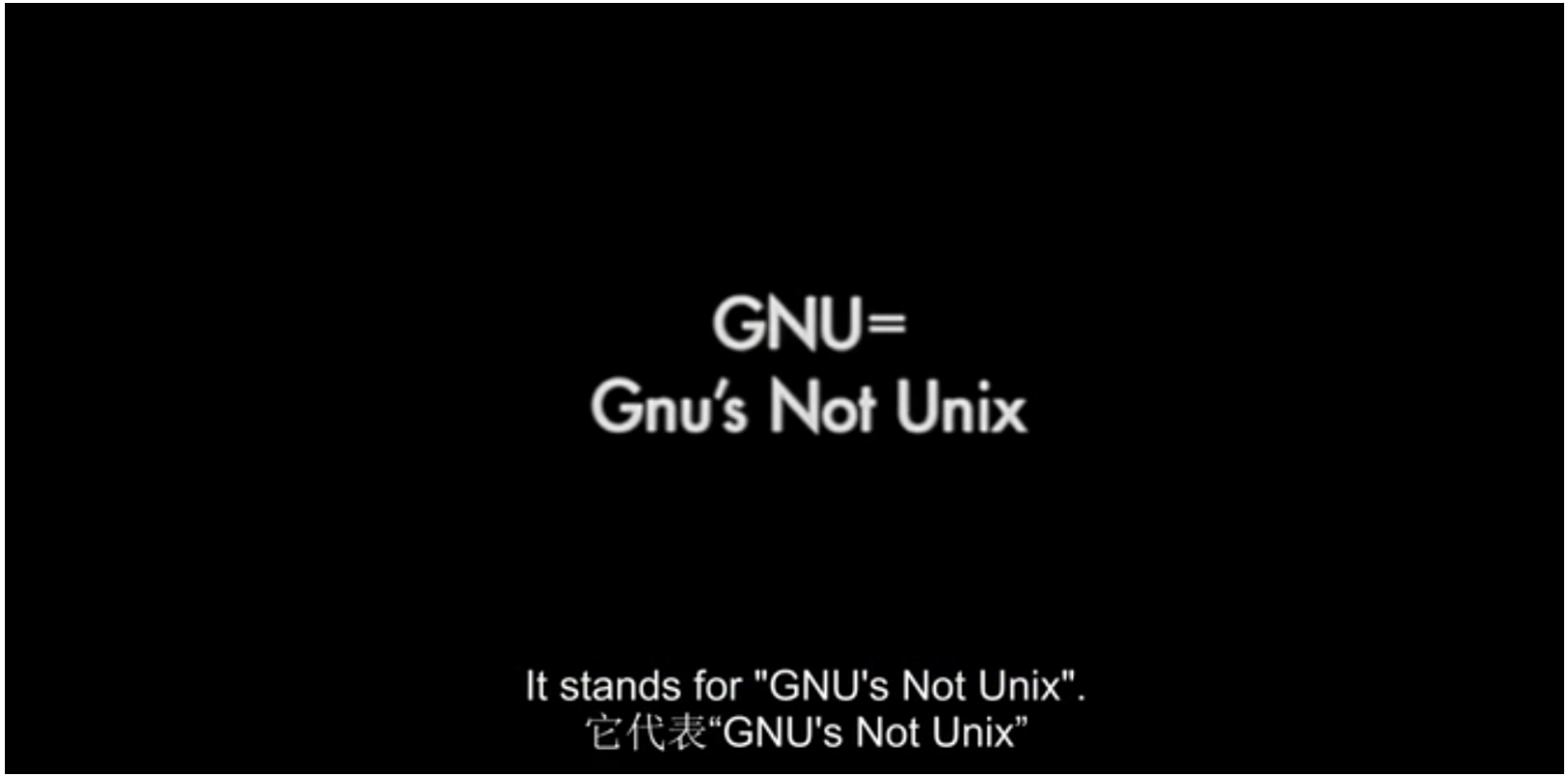
FSCK and Journaling

- **Solution #2: Journaling (Write-Ahead Logging)**
  - **WAL 来自于 database 领域**
  - **最早使用 journaling 的文件系统是 Cedar(1987)**
  - **现在广范使用： Linux ext3, ext4, reiserfs, IBM’s JFS, SGI’s XFS, Windows NTFS**

电影推荐

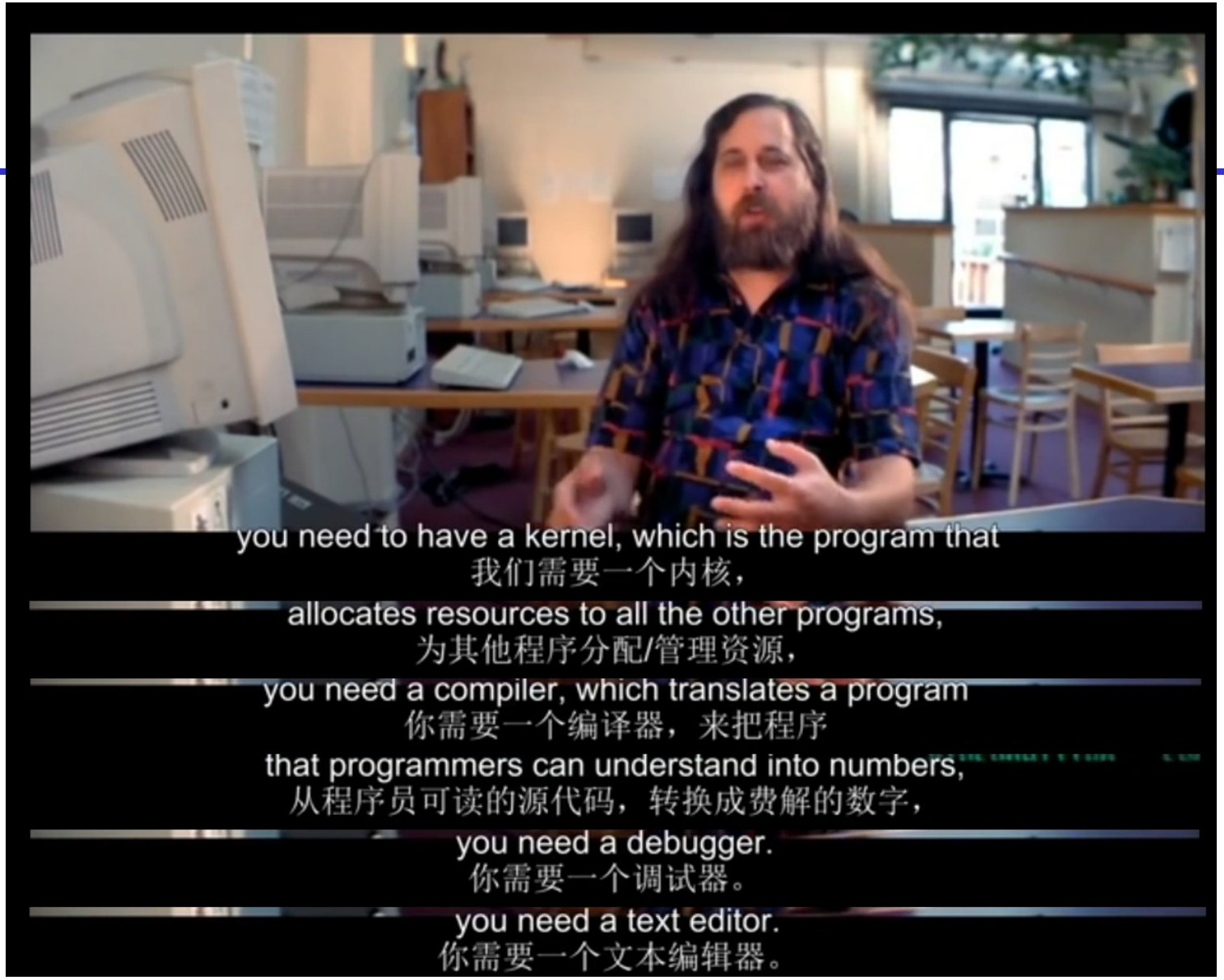
- **Revolution OS（操作系统革命）**
  - **2001 年上映的纪录片电影**
  - **记录了 GNU 、 Linux 、自由软件运动、开源软件运动的兴起**
  - **Richard Stallman 、 Linus Torvalds 等真人主演**

https://www.bilibili.com/video/BV1gx411Q79H/


GNU=
Gnu's Not Unix
It stands for "GNU's Not Unix".
它代表“GNU's Not Unix”

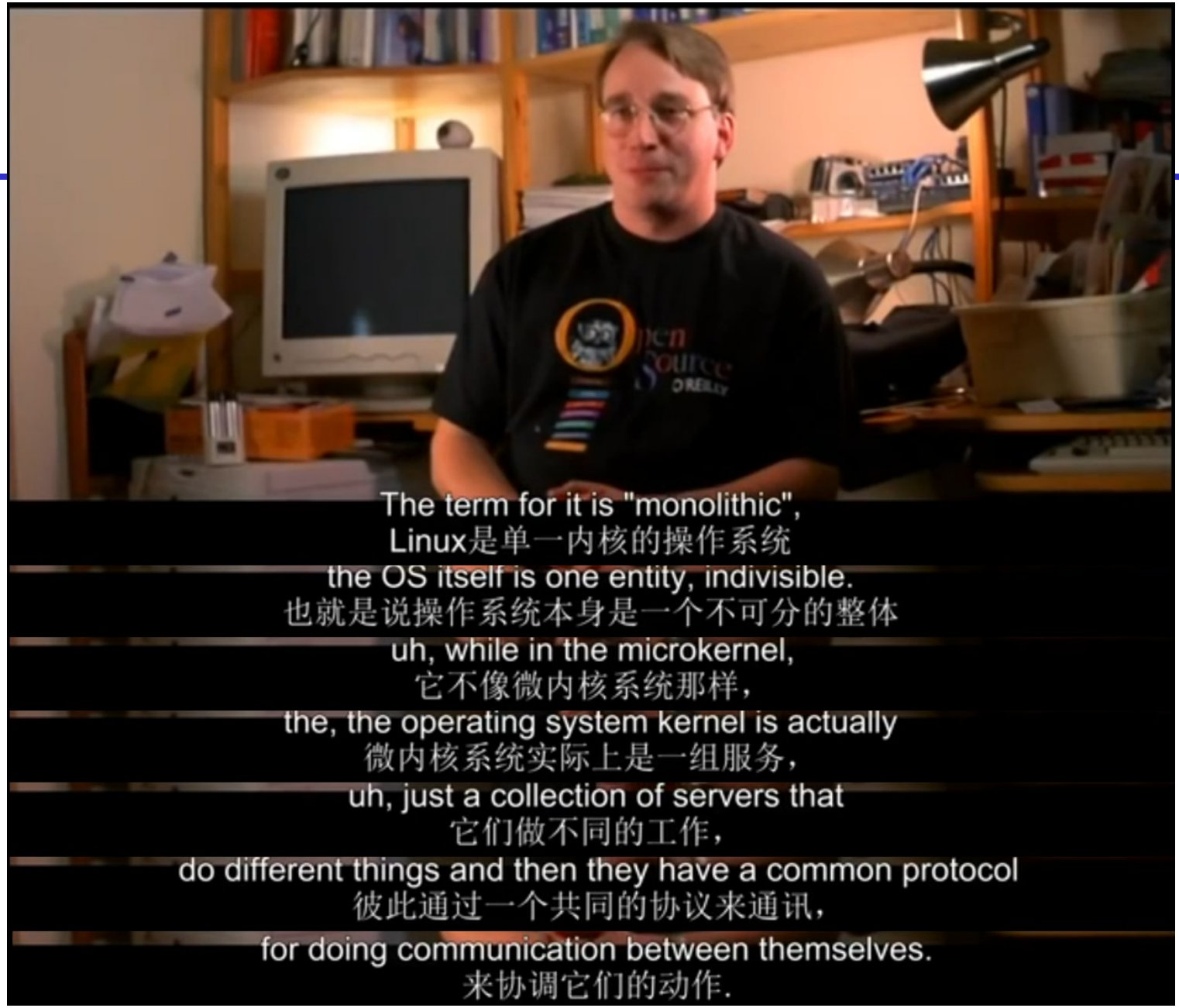

Known as Berkeley Unix, or BSD,
被称为Berkeley Unix,或者叫BSD.

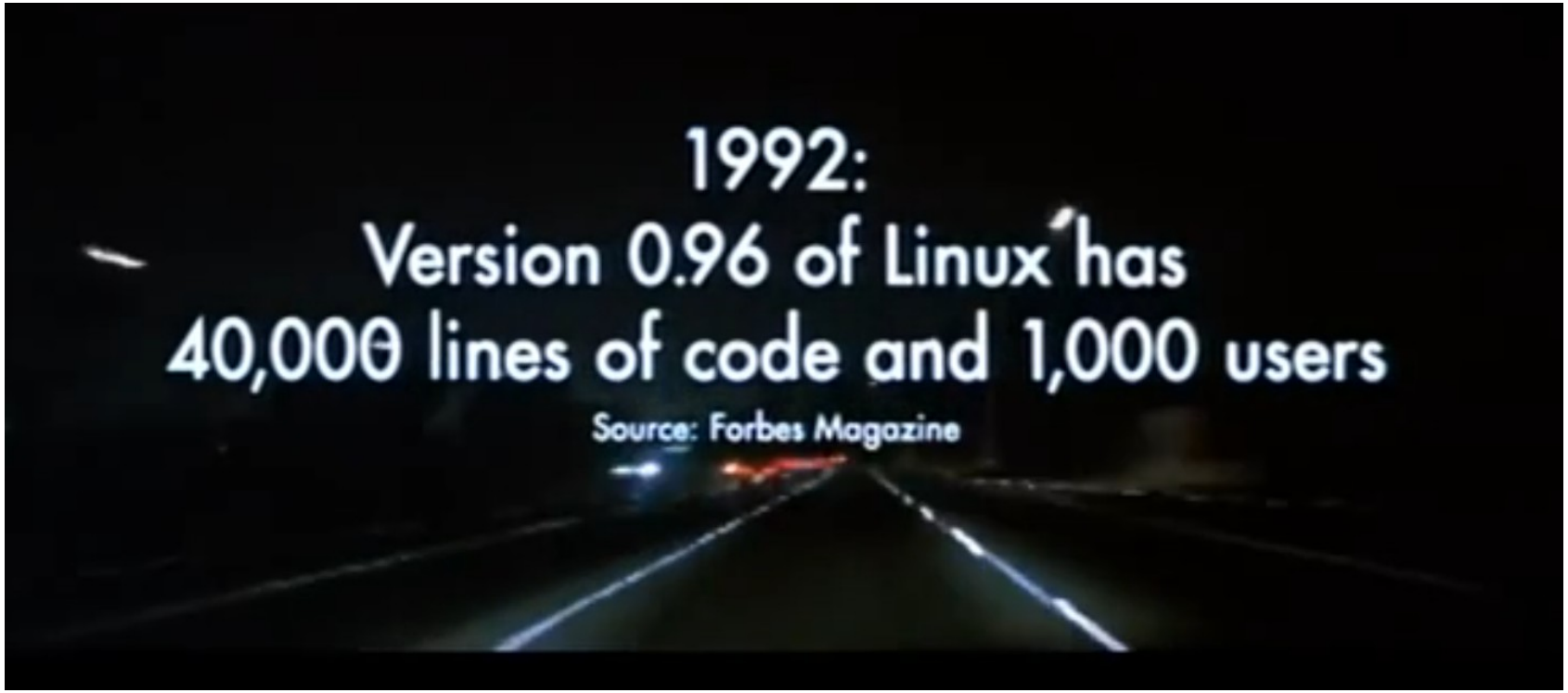

Well, Unix consisted of a large number of
separate programs
嗯，Unix由大量的分离程序组成。


you need to have a kernel, which is the program that
我们需要一个内核，
allocates resources to all the other programs,
为其他程序分配/管理资源，
you need a compiler, which translates a program
你需要一个编译器，来把程序
that programmers can understand into numbers,
从程序员可读的源代码，转换成费解的数字，
you need a debugger.
你需要一个调试器。
you need a text editor.
你需要一个文本编辑器。

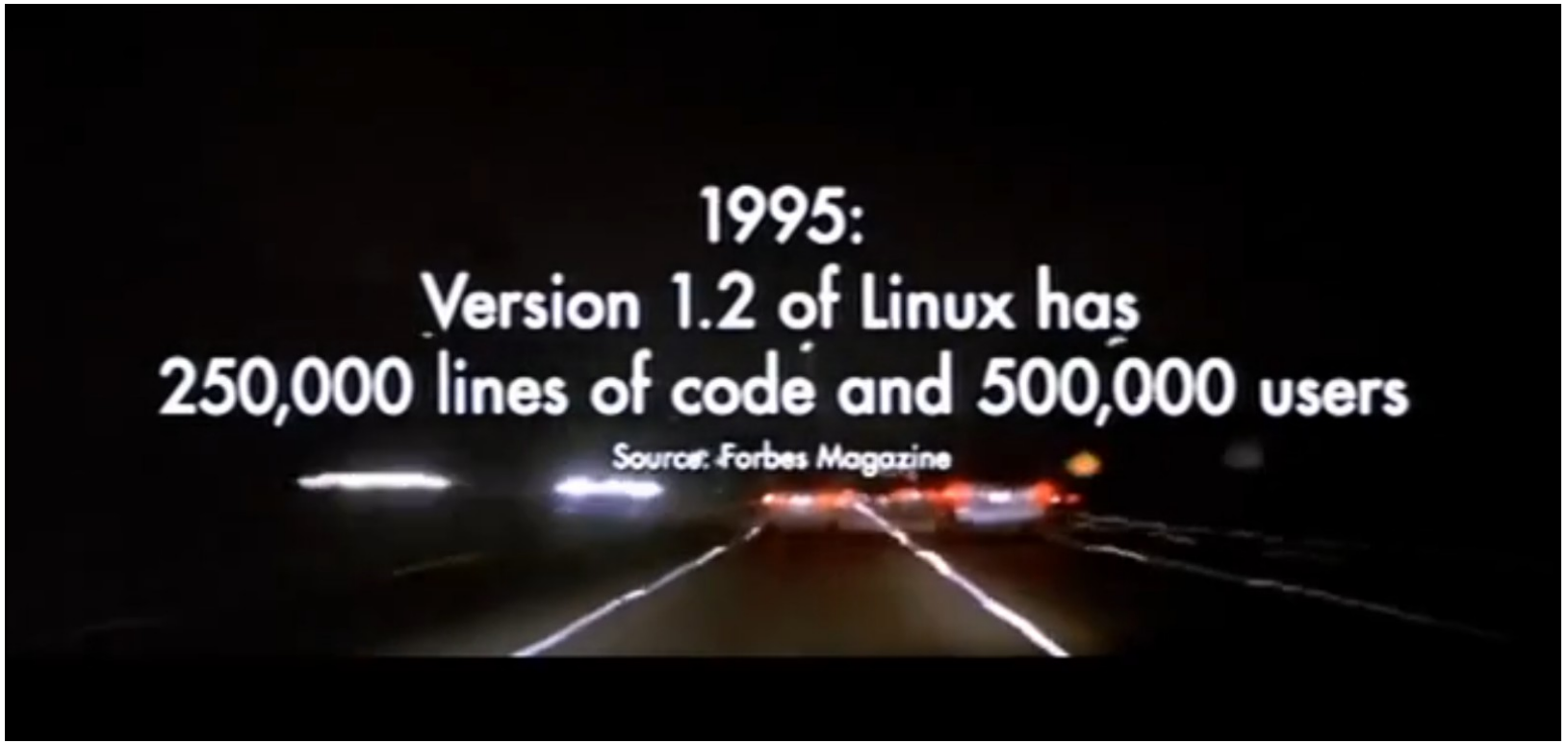

under Richard Stallman
created a great tool called "bison".
开发了一个很棒的工具，叫做"bison"。


To prevent that, we use a technique called "Copyleft".
为了防止这事情，我们用了一种技术，叫"copyleft"。


The term for it is "monolithic",
Linux是单一内核的操作系统
the OS itself is one entity, indivisible.
也就是说操作系统本身是一个不可分的整体
uh, while in the microkernel,
它不像微内核系统那样，
the, the operating system kernel is actually
微内核系统实际上是一组服务，
uh, just a collection of servers that
它们做不同的工作，
do different things and then they have a common protocol
彼此通过一个共同的协议来通讯，
for doing communication between themselves.
来协调它们的动作.


1992:
Version 0.96 of Linux has
40,000 lines of code and 1,000 users
Source: Forbes Magazine


The killer app of Linux was undoubtedly
the Apache web server.
Linux里最招人喜欢的软件无疑就是Apache



中国 1994 年 5 月接入互联网， 1995 年 9 月第一个 ISP

1995 年 MySQL 发布， 1996 年 PostgreSQL 发布

1999 年前后，中国最早一批国产数据库公司相继成立，包括人大金仓、武汉达梦等