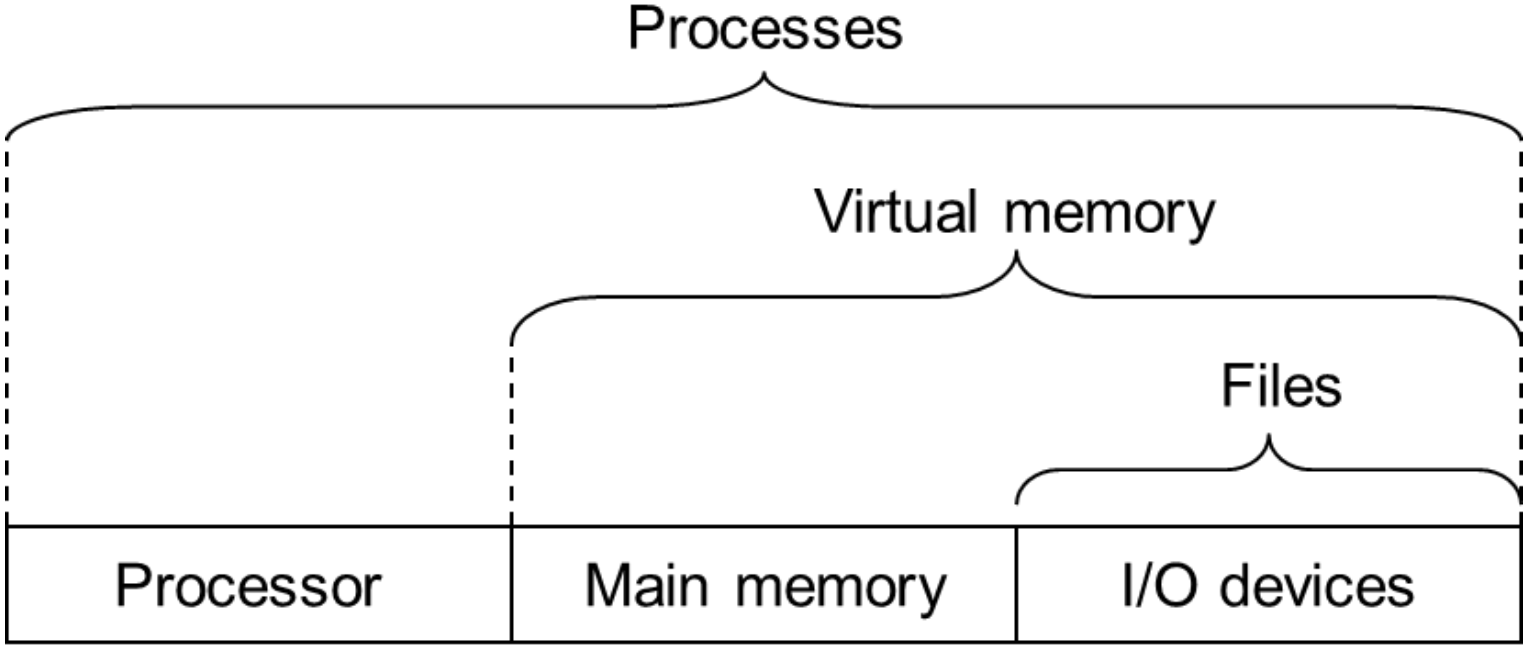
Persistence (I)
I/O and Files
(OSTEP Section 36,37,38,44)

OS: Three Easy Pieces

- 虚拟化
- 持久化
- 并发



Outline

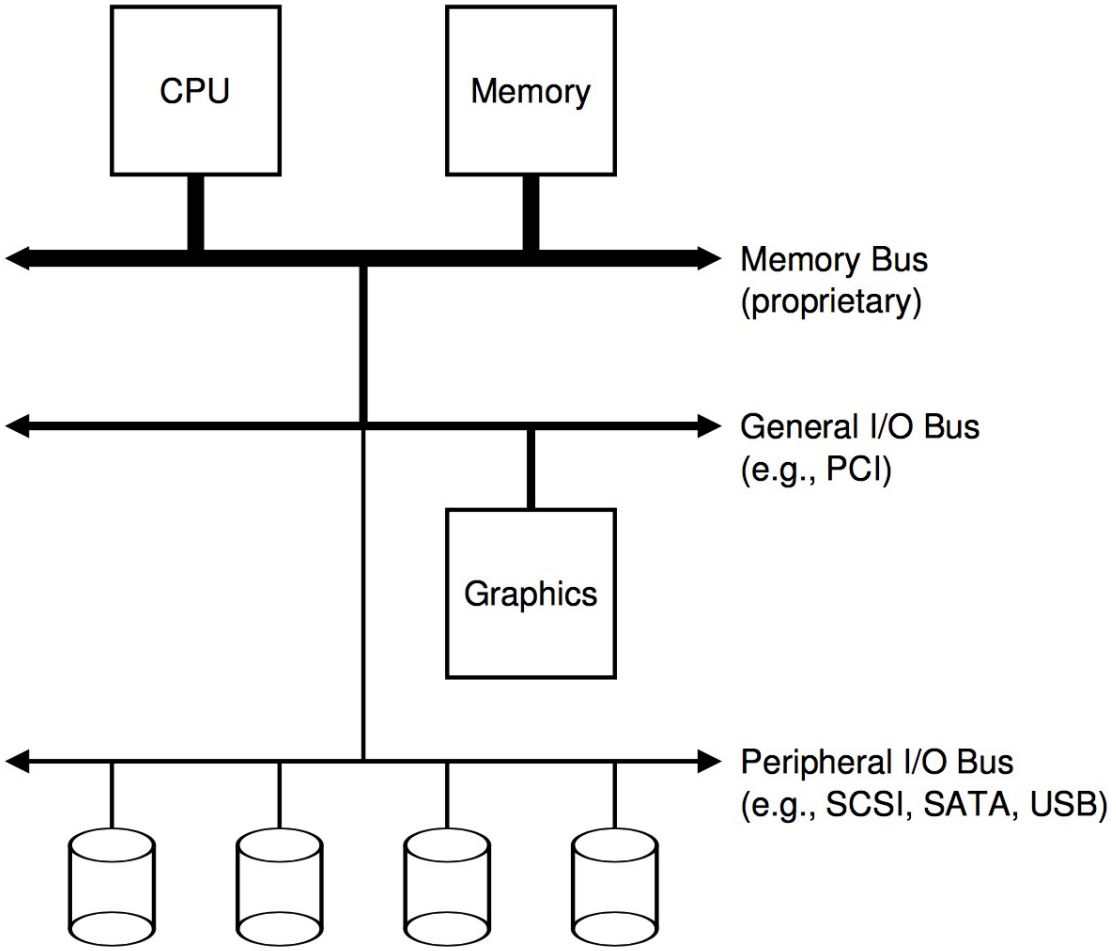
- I/O Devices
- Unix I/O
- Reading File Metadata
- Sharing Files & I/O redirection
- Robust I/O（了解）
- Standard I/O（了解）
- Fast I/O（了解）

I/O Devices

- **I/O: Input/Output**
  - 持久化设备
    - 加载程序、数据
    - 保存状态
  - **CPU, Cache, RAM** 都无法保存状态
  - 新的挑战者
    - 非易失内存
      - **Non-Volatile Memory**
      - **Storage Class Memory**
      - **Persistent Memory**

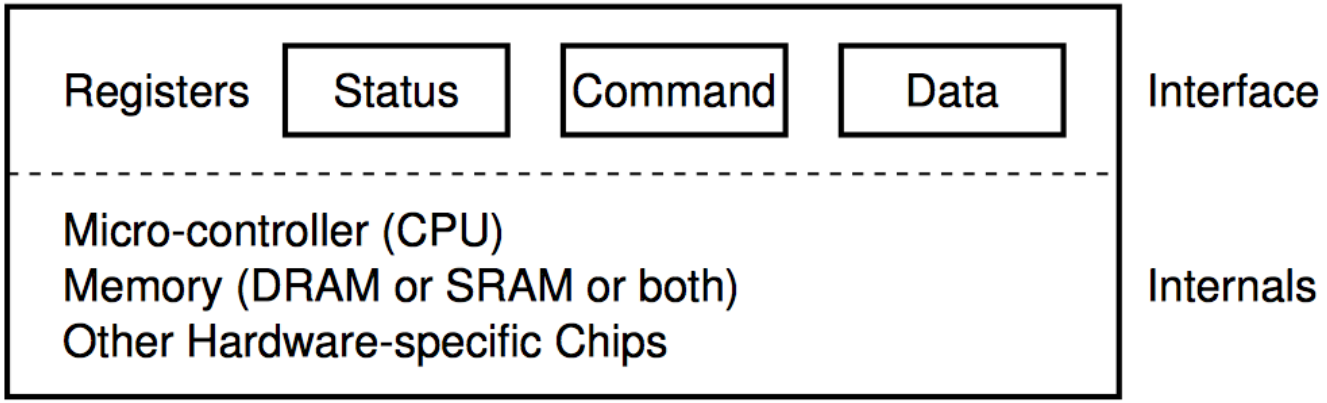
I/O Devices

- **System Architecture**
  - 层次结构原因
    - 快速设备近
    - 慢速设备远



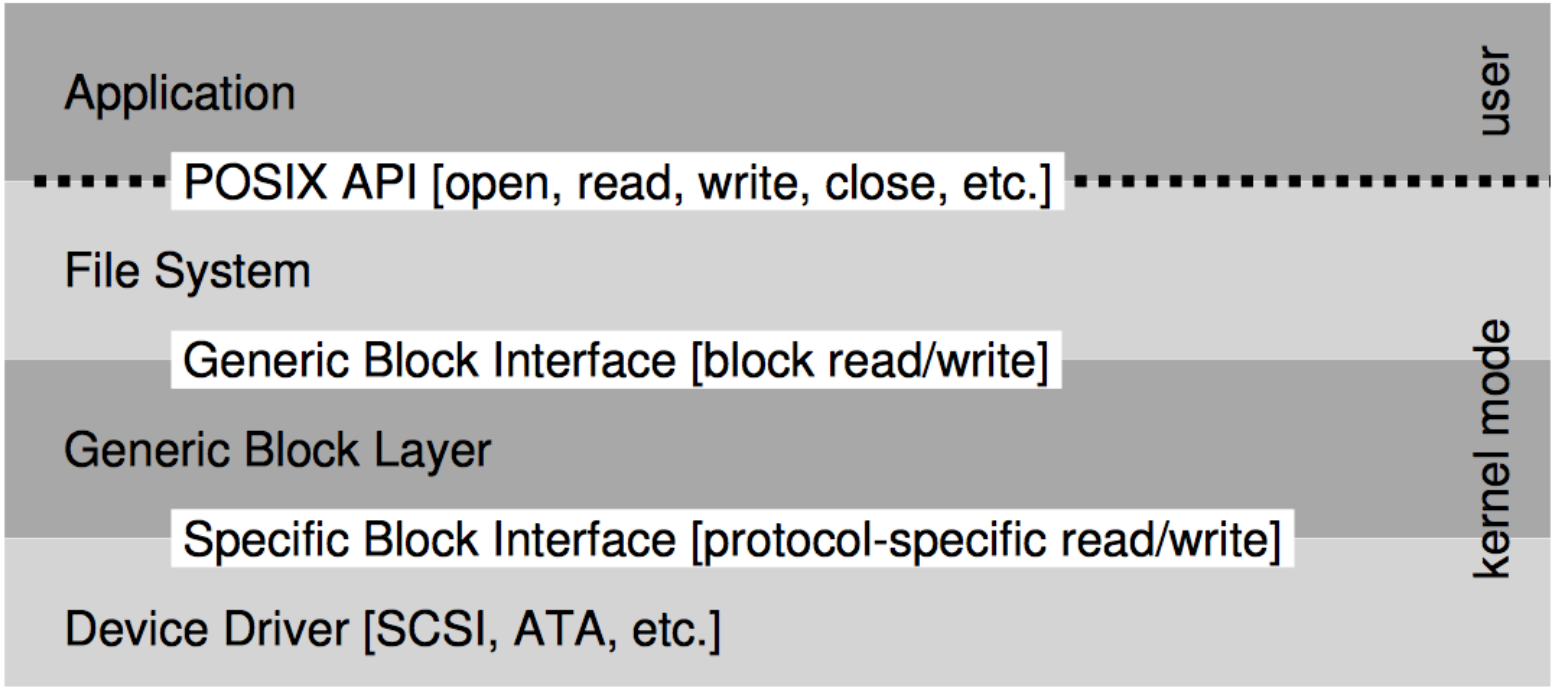
I/O Devices

- **I/O 设备的通用结构**
  - **Status**
  - **Command**
  - **Data**



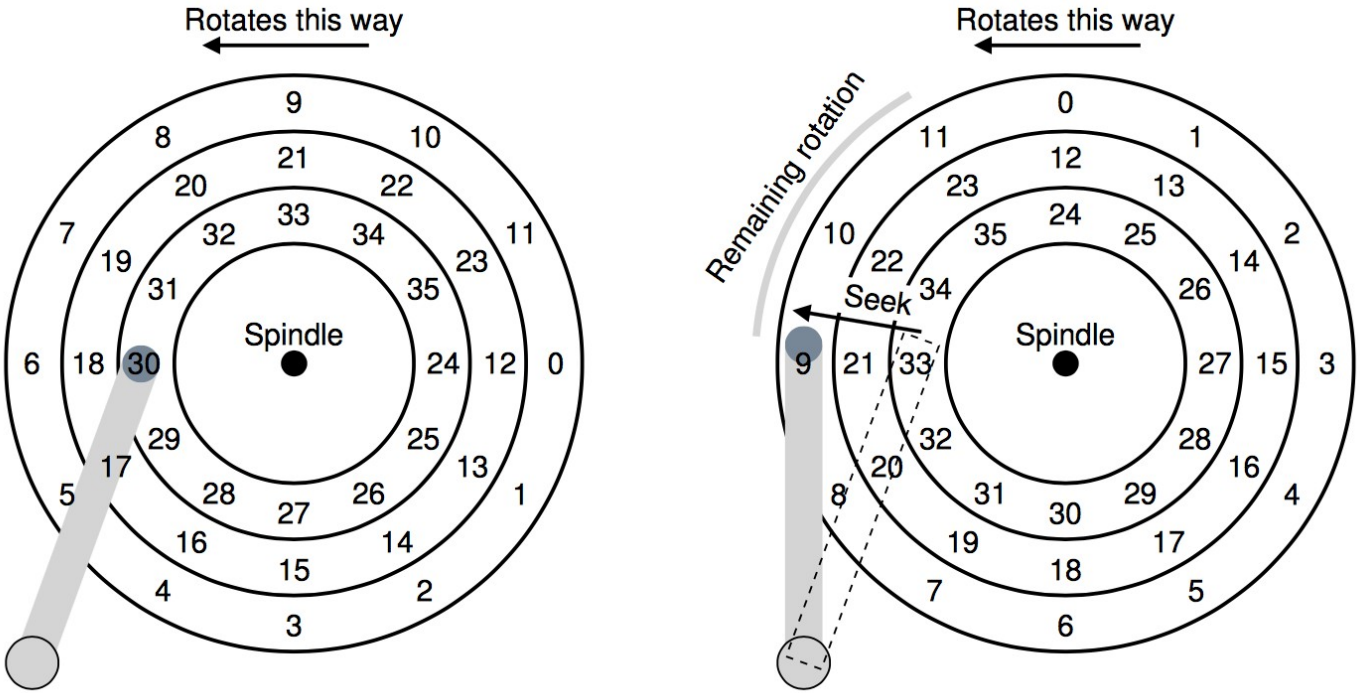
I/O Devices

- **File System Stack**



I/O Devices

- 磁盘



I/O Devices

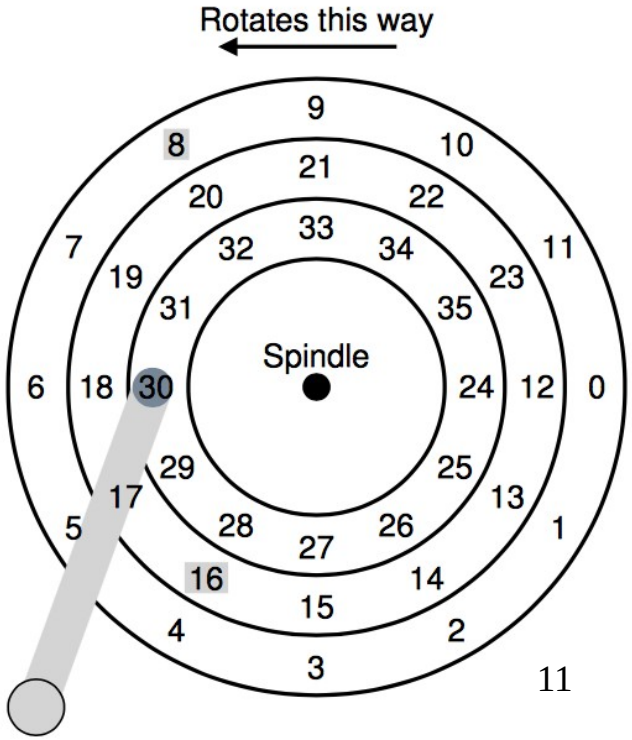
- 磁盘调度
  - : Shortest Seek Time First
    - picking requests on the nearest track first
  - 问题
    - 饥饿
    - 陷入局部最优，达不到全局最优

I/O Devices

- **Elevator**
  - 外 -> 内；内 -> 外；外 -> 内…
  - **F-SCAN**
    - **Freeze the queue，**一个方向扫完之后才处理新来的请求
    - 避免饥饿
    - 效率问题：新来的附近请求
  - **C-SCAN (Circular SCAN)**
    - 从外到内移动磁头，移动到目前已有请求中最内道请求的位置，再置位到已有请求中最外道的位置
    - 所有磁道更加公平，否则中间磁道访问的机会更多

I/O Devices

- **SSTF 和 Elevator 算法的通用问题**
  - 没有考虑到旋转的影响
- **SPTF: Shortest Positioning Time First**
  - 当前磁头在 30
  - 下一步到 16，还是 8？
  - It depends
  - 看 Seek time 和 rotation delay 哪个更大，最好在 Disk controller 里面实现



课堂练习

- **磁盘调度算法**

三、某移动臂磁盘的柱面由外向里顺序编号，假定当前磁头停在 100 号柱面且移动臂方向是向里的，现有如下表 1 所示的请求序列在等待访问磁盘：

表 1　　访问磁盘请求序列

请求次序	1	2	3	4	5	6	7	8	9	10
柱面号	190	10	160	80	90	125	30	20	140	25

回答下面的问题：

(F-SCAN)

① 写出分别采用“最短查找时间优先算法”和“电梯调度算法”时，实际处理上述请求的次序。

② 针对本题比较上述两种算法，就移动臂所花的时间（忽略移动臂改向时间）而言，哪种算法更合适？简要说明之。（考研）

练习答案

- 最短查找时间优先
  - 100-90-80-125-140-160-190-30-25-20-10
  - 10+10+45+15+20+30+160+5+5+10 = 310

- 电梯算法（F-SCAN)
  - 100-125-140-160-190-90-80-30-25-20-10
  - 25+15+20+30+100+10+50+5+5+10 = 270

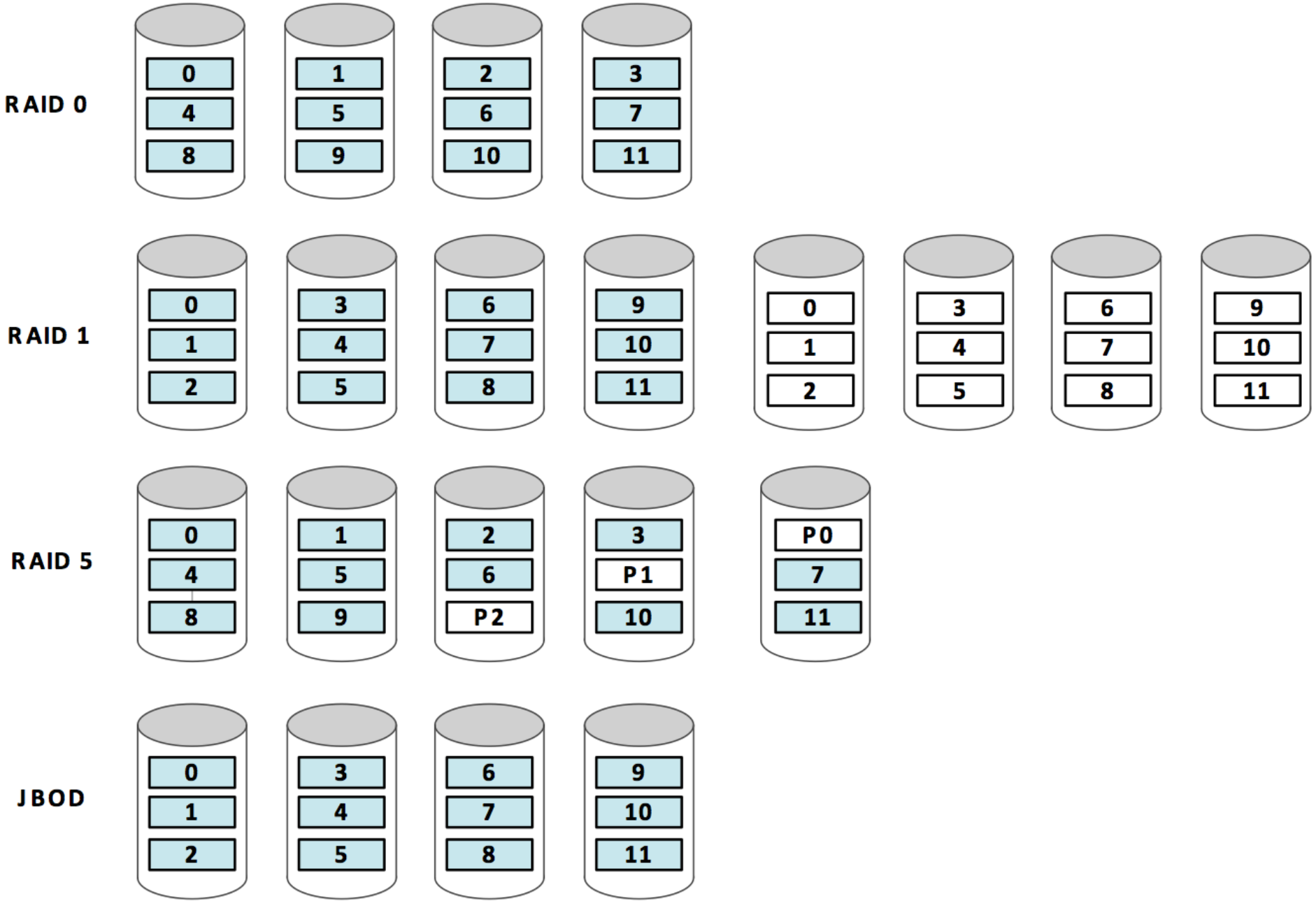
I/O Devices

- **RAID**
  - **1988 年美国加州大学伯克利分校的 D. A. Patterson 教授等首次在论文“A Case of Redundant Array of Inexpensive Disks”中提出了 RAID 概念，即廉价冗余磁盘阵列（Redundant Array of Inexpensive Disks）**
  - **后来 RAID 咨询委员会决定用“独立”替代“廉价”，于时 RAID 变成了独立磁盘冗余阵列（Redundant Array of Independent Disks）**

图灵奖获得者 David Patterson

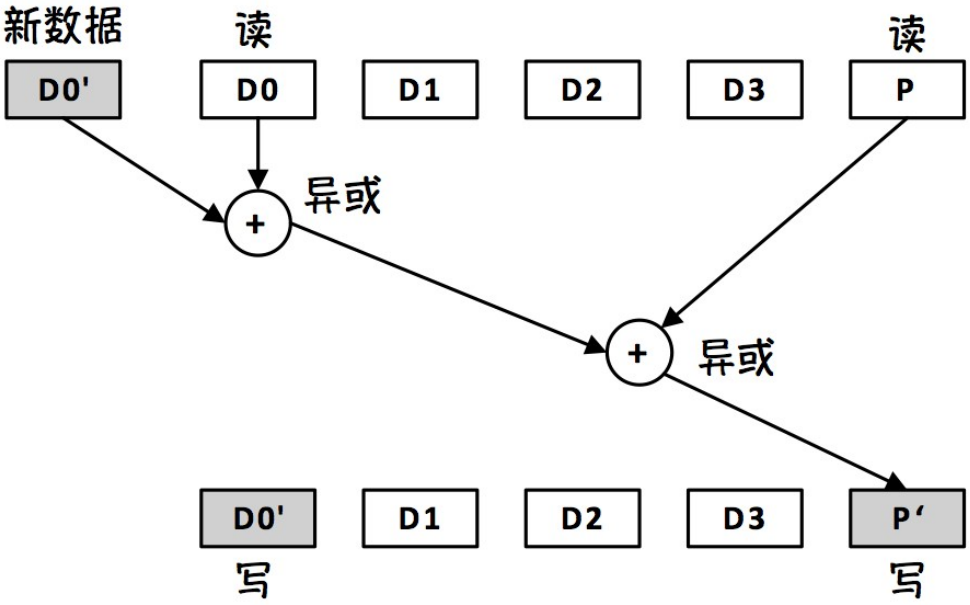
I/O Devices

- 无冗余（RAID 0）
  - 条带化（striping）
- 镜像（RAID 1）
- 块交叉奇偶校验（RAID 5）
- RAID 10/RAID 01
- JBOD


RAID 0
0
4
8
1
5
9
2
6
10
3
7
11
RAID 1
0
1
2
3
4
5
6
7
8
9
10
11
0
1
2
3
4
5
6
7
8
9
10
11
RAID 5
0
4
8
1
5
9
2
6
P2
3
P1
10
P0
7
11
JBOD
0
1
2
3
4
5
6
7
8
9
10
11

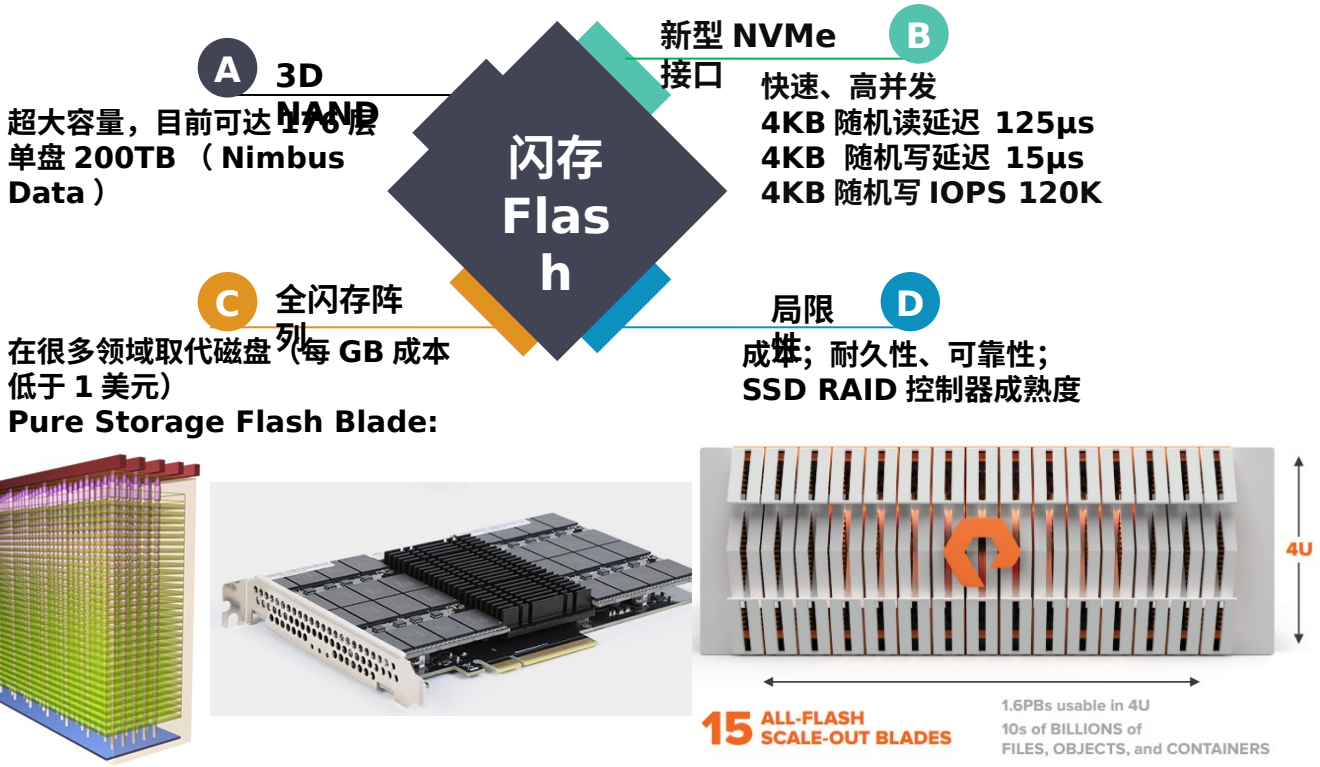
I/O Devices

- **RAID5**
  - 四份原始数据分别是0、1、1、0，采用偶校验，校验位P是0（=0⊕1⊕1⊕0）
  - 当Do从0修改为1后，新校验位P’=Do’⊕Do⊕P=1⊕0⊕0=1



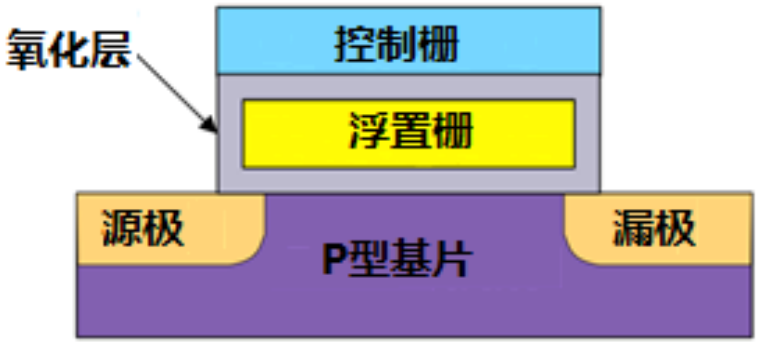
I/O Devices

- **Flash-based SSD**



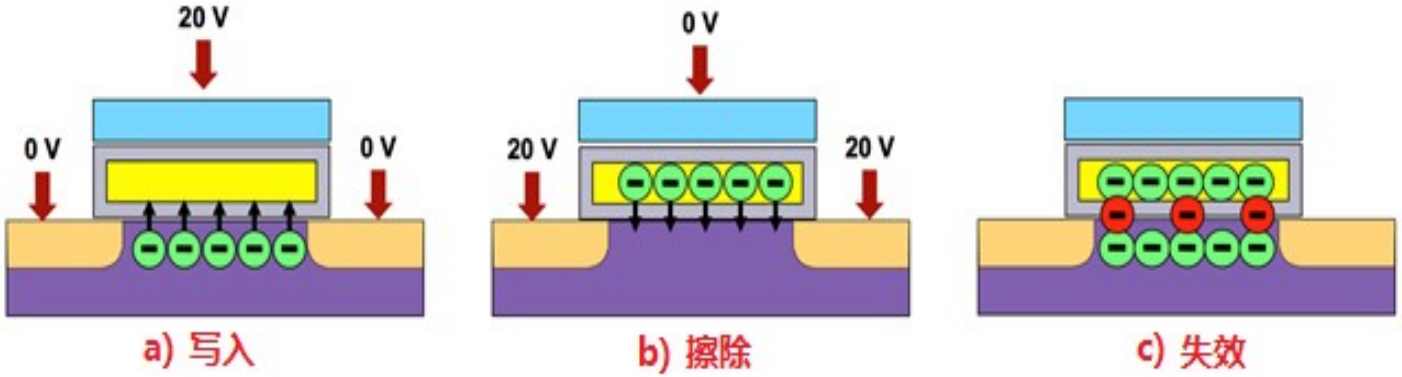
I/O Devices

- **Flash-based SSD**
  - **Flash：与 E2PROM 类似，利用捕获电子存储信息**
  - **闪存的基本存储单元由 P 型基片、源极、漏极、控制栅和浮置栅等构成**



I/O Devices

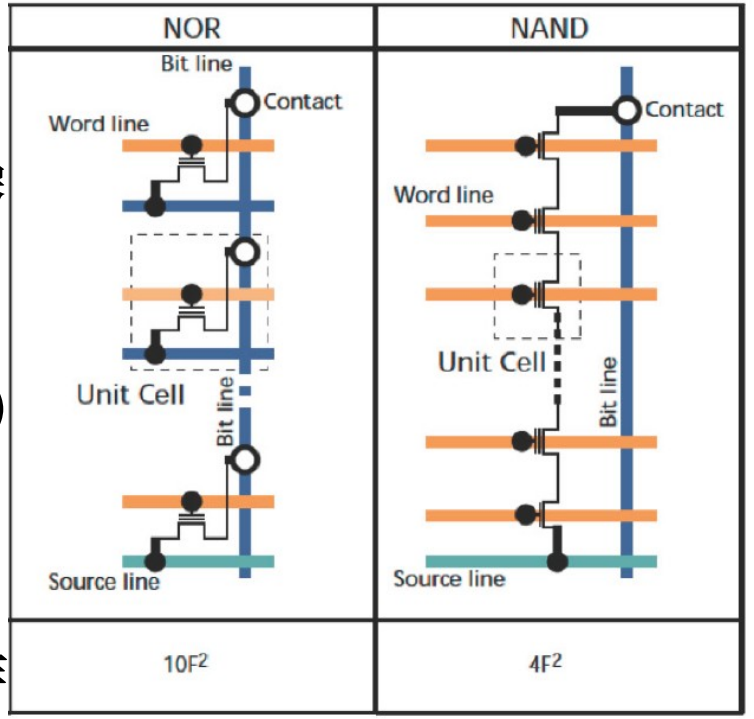
- 闪存操作接口
  - 读、写、擦除
  - 耐久性问题：由于氧化层很薄，允许擦除的次数存在上限。



		iiii	*Initial: pages in block are invalid* (i)
Erase()	→	EEEE	*State of pages in block set to erased* (E)
Program(0)	→	VEEE	*Program page 0; state set to valid* (V)
Program(0)	→	**error**	*Cannot re-program page after programming*
Program(1)	→	VVEE	*Program page 1*
Erase()	→	EEEE	*Contents erased; all pages programmable*

I/O Devices

- **Flash 的分类:**
  - **NOR Flash**
    - **以 bit 为基本访问单位;**
    - **读操作很快、写操作较慢、擦除操作很慢**
  - **NAND Flash**
    - **NAND Flash 是以页 (若干 KB) 为单位进行读写操作，以块 (几百 KB 或几 MB) 为单位进行擦除操;**
    - **NAND Flash 的读操作相对 NOR Flash 慢，但写和擦除操作比 NOR Flash 块**



I/O Devices

- **Flash 的分类:**
  - 一个存储单元上存储多个二进制位
    - **SLC （Single-Level Cell）**
      - **SLC 的最大可擦除次数一般在 10 万次**
    - **MLC （Multi-Level Cell）**
      - **MLC 在 5000~10000 次**
    - **TLC （Triple-Level Cell）**
      - **TLC 一般在 1000 次以内**

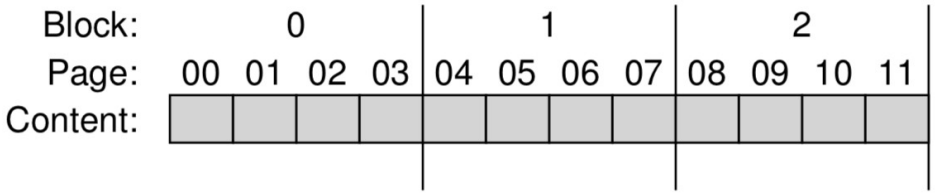
Device	Read (μs)	Program (μs)	Erase (μs)
SLC	25	200-300	1500-2000
MLC	50	600-900	~3000
TLC	~75	~900-1350	~4500

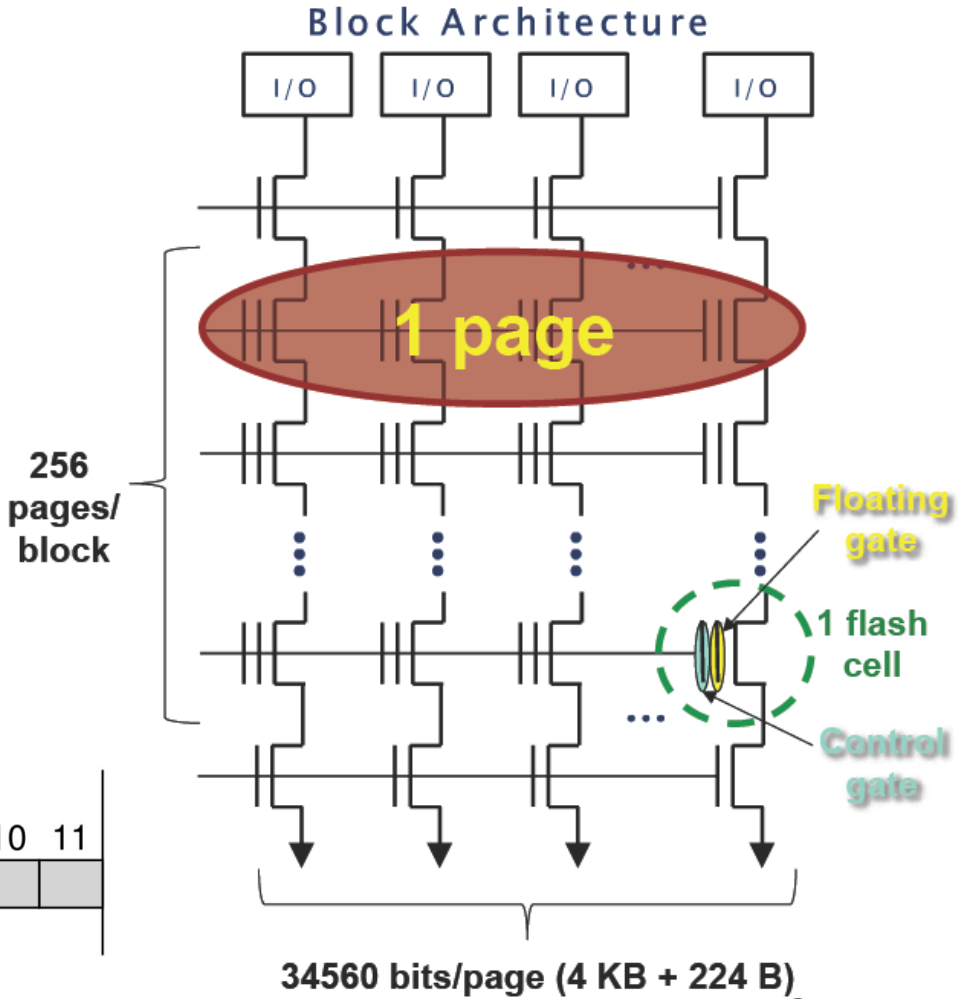
I/O Devices

- **Flash 的内部结构**

	MLC
Page Size	4 KB
Block Size	1 MB
Chip Size	16 GB
Read Page (µs)	150
Program Page (µs)	1000
Erase Block (µs)	3000

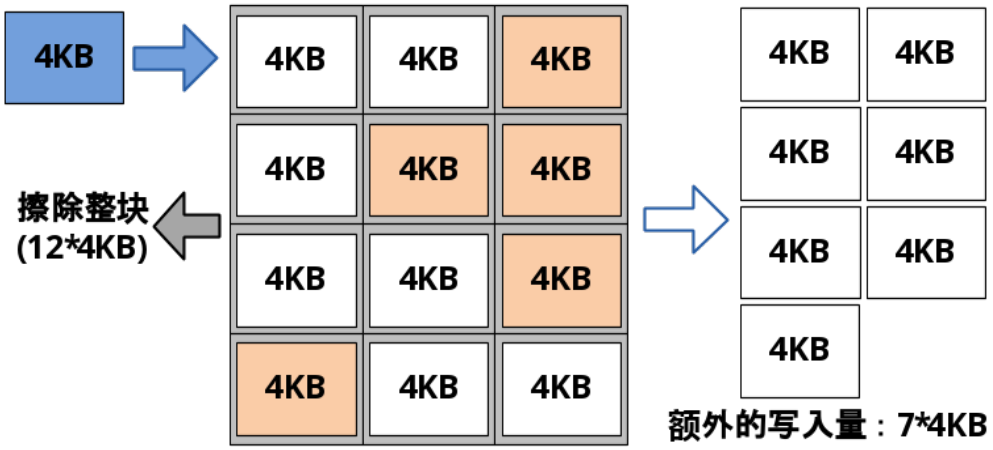
NAND flash MICRON MLC:
MT29F128G08CJABB





I/O Devices

- 闪存的写放大现象
  - 根本原因是擦除块 > 读写单元
  - 擦除块前，先要将其中有效数据搬移到其他未知，即产生了额外的写入量

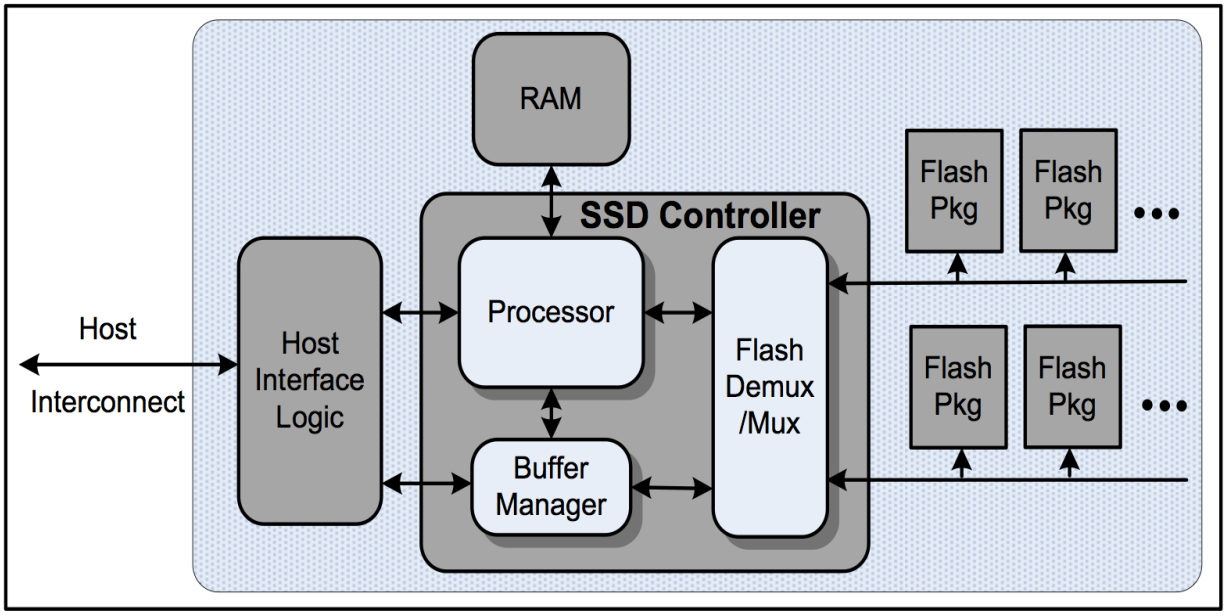


I/O Devices

- ***Flash* 的优点**
  - 高存储密 (3D NAND)
  - 非易失性 (Non-Volatile)
  - 节能
  - 高性能
  - 高可靠性 (闪存的 MTTF 更长；没有机械装置，不受震动等影响)
- ***Flash* 的缺点**
  - 价格较高
  - 读写不平衡
    - 写本身就比读慢，还有更慢的擦除
  - 写入耐久性有限

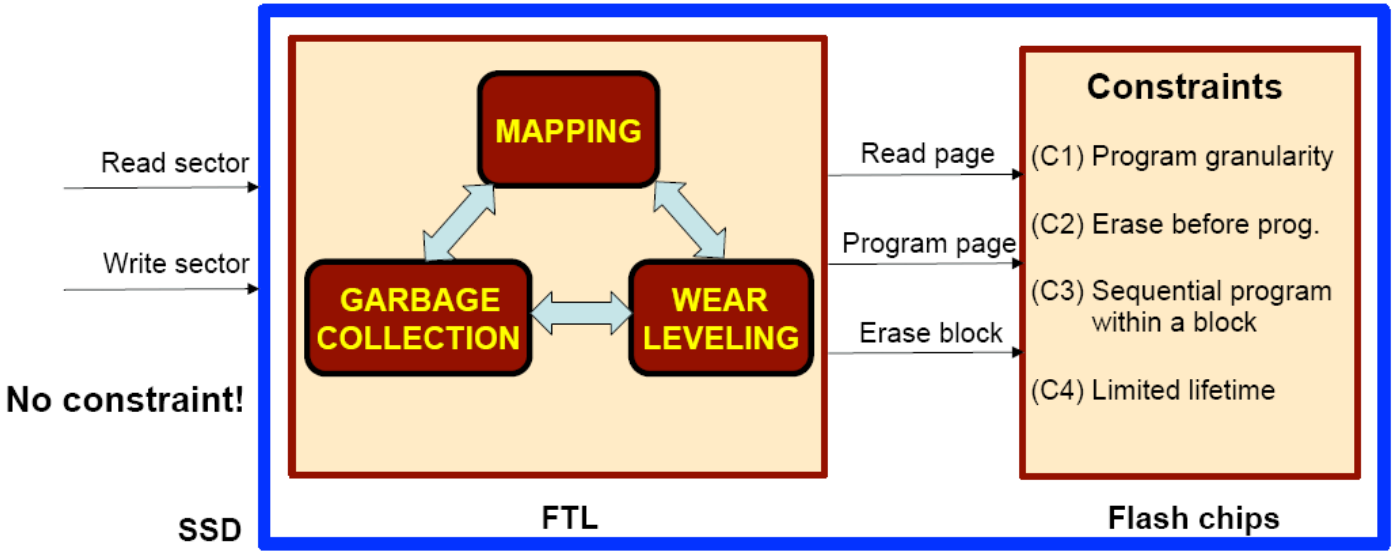
I/O Devices

- 固态硬盘的内部结构
  - 包括接口逻辑模块、处理器、内存、缓冲区管理、多路访问通道和很多闪存芯片
  - 直观的说，固态硬盘本身相当于一台微型的计算机



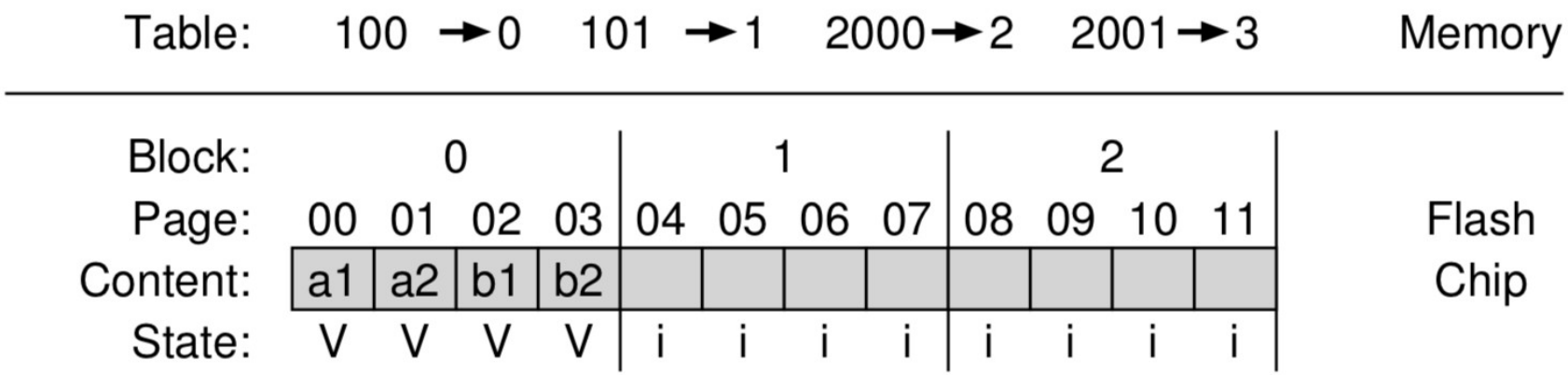
I/O Devices

- **FTL （Flash Translation Layer）**
  - **将 Flash Chip 封装成标准块设备（只有 Read, Write 两种操作）**
  - **地址映射（Mapping）、垃圾回收（Garbage Collection）、磨损平衡（Wear-Leveling）**



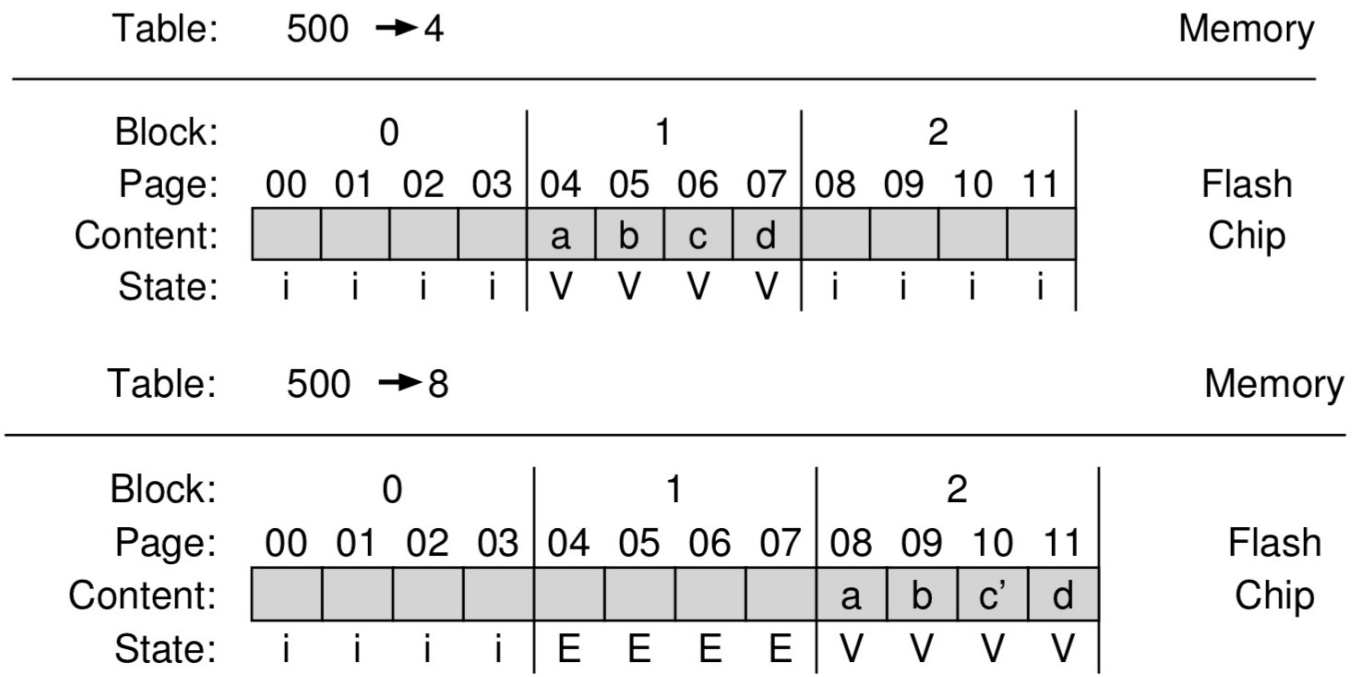
I/O Devices

- **Mapping**
  - 逻辑页 -> 物理页的映射关系
  - **In-memory**，可以通过 **Flash** 上内容 **recovery**
  - 最简单是 **page-level mapping**，但占用内存空间较大



I/O Devices

- **Block-level mapping**
  - 节省空间
  - 但是粒度粗，不太方便
    - 例如只修改 c’，也会导致整个 block 要搬移



I/O Devices

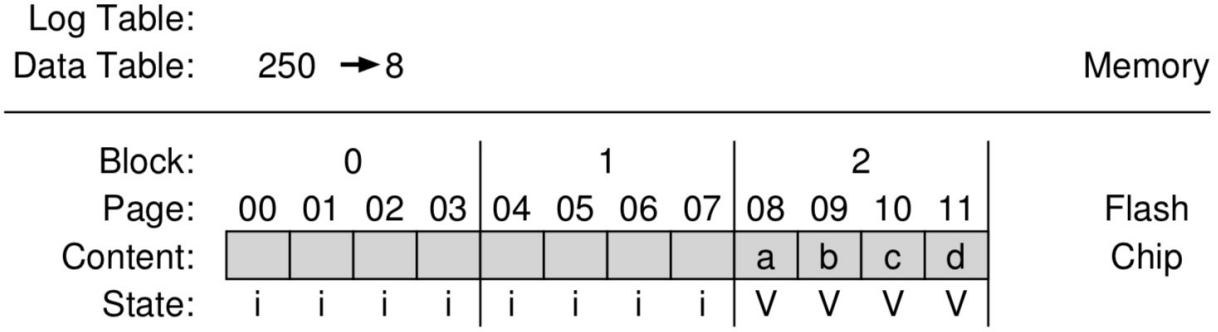
- **Hybrid Mapping**

  基于一个假设:
  当前时刻只有少量的 block 需要进行局部的更新

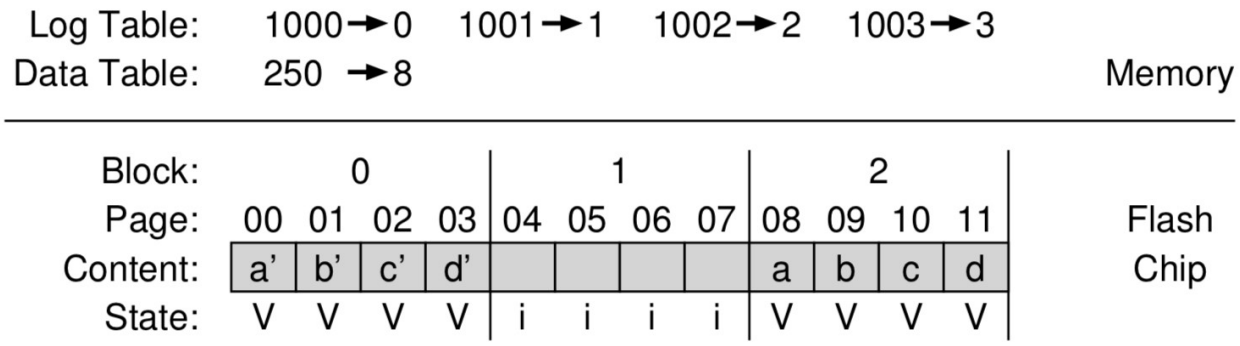
  - 借鉴以上两者的优点，对多数 block 采用 block-level mapping (data table)，对少数需要局部更新的 block，创建 log blocks 并采用 page-level mapping (log table)
  - 访问流程:
    - 先访问 log table；
    - 不在其中的话，再找 data table

I/O Devices

- **Hybrid Mapping**
  - 例如，初始状态 a, b, c, d 是一个 block 中的连续

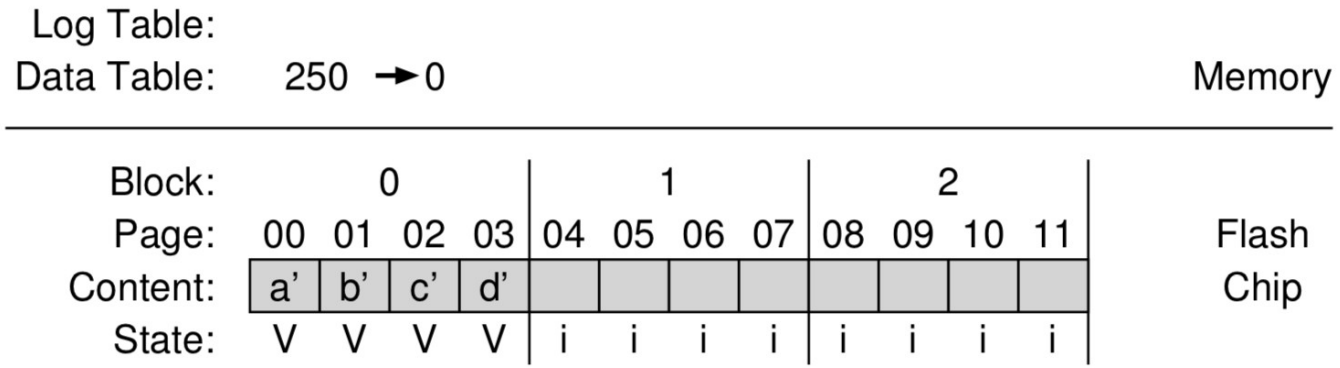


  - 对 block 2 的修改会写在一个新的 log block 中



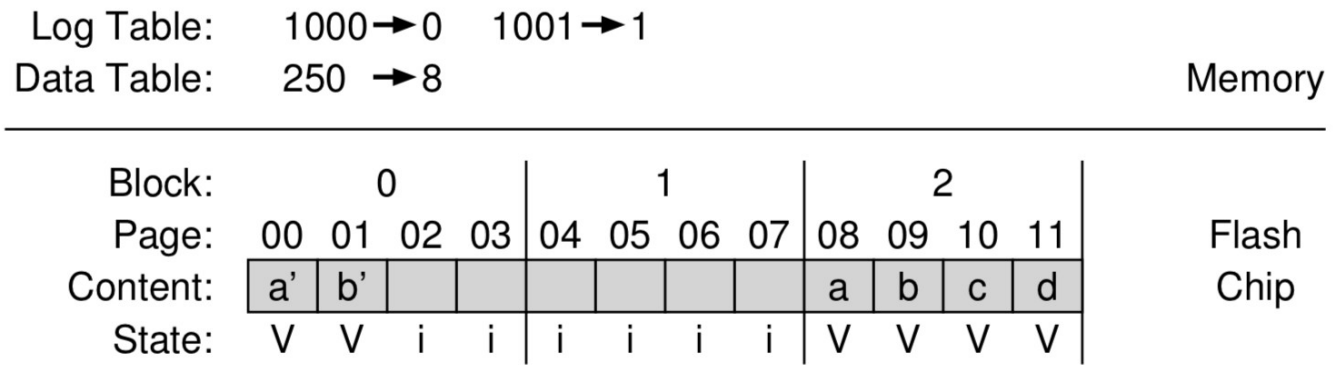
I/O Devices

- **Hybrid Mapping**
  - 为了减少内存开销， log table 要尽量通过合并操作，转移到 data table 中
  - 例如以上例子中， block 0 中的新数据和 block 2 中的原始数据，定期会合并为一个新的 block，然后只需要 block-level mapping
  - 这个例子很幸运，只需要切换 block，因此叫 switch merge；开销是擦除一个 block



I/O Devices

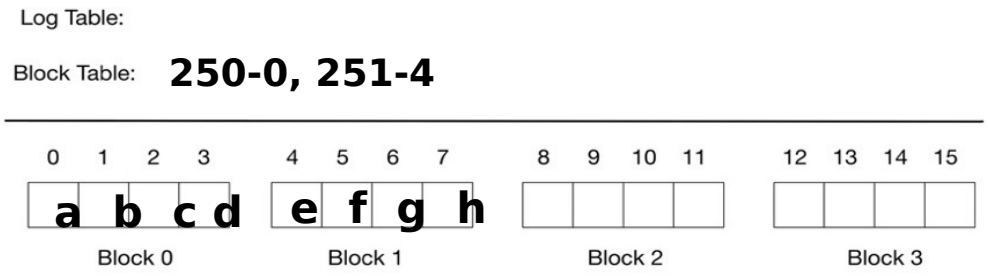
- **Hybrid Mapping**
  - **Partial merge**：把旧块中未修改的数据拷贝到新块中即可；开销是搬移 pages + 擦除一个 block



  - **Full merge**：其他情况；开销是搬移一个 block 的 pages，擦除两个 blocks

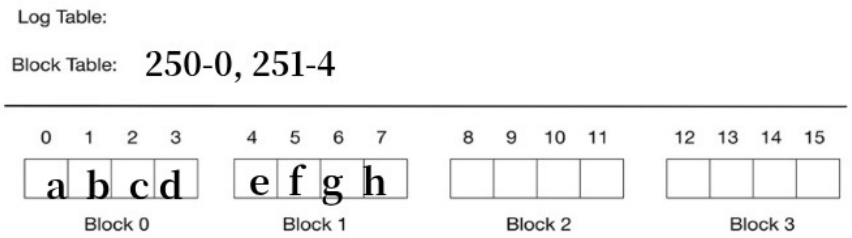
课堂练习

- SSD 内部 FTL 算法采用 hybrid mapping 的方式进行虚拟地址到物理地址的映射， block table 中记录逻辑 block id 与物理地址的映射关系, log table 记录逻辑 page id 与物理地址的映射关系（page id = 4*block id）；
- 同时最多只能有 1 个 log block 存在，每个原始 block 对应一个 log block （即一个 log block 中的数据不能来自于不同的原始 block），选择 log block 进行合并操作时优先选择代价小的进行，选择空白 block 写入时采用 first-fit 原则；
- 初始状态如下图所示，到达的请求序列为 a, b, c, d, e, e, f, g, e, c ， 请画出序列结束后的状态，计算发生了几次 merge ，分别是什么类型?

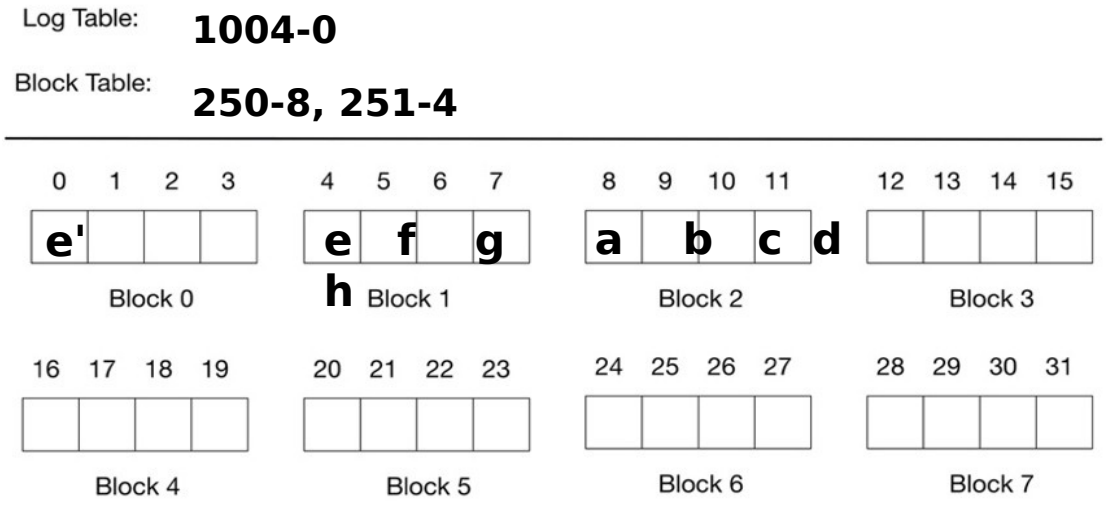


练习答案（1/3）

- 初始状态如下图所示，到达的请求序列为 a, b, c, d, e, e, f, g, e, c ， 请画出序列结束后的状态



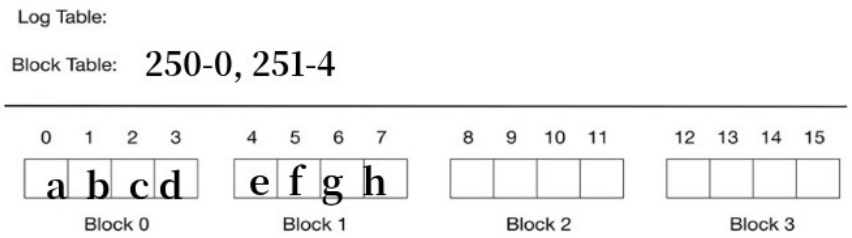
第一个 e 到达：



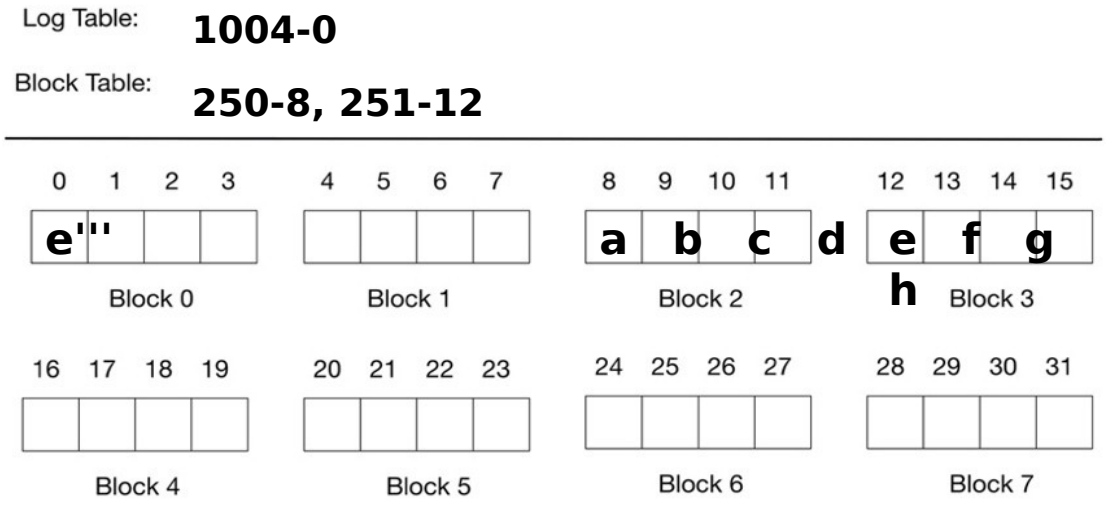
Switch merge

练习答案（2/3）

- 初始状态如下图所示，到达的请求序列为 a, b, c, d, e, e, f, g, e, c ， 请画出序列结束后的状态



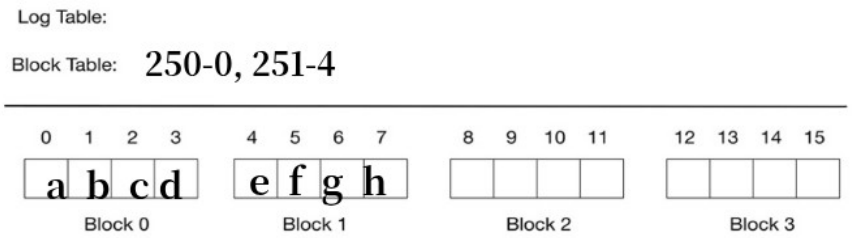
第 3 个 e 到达:



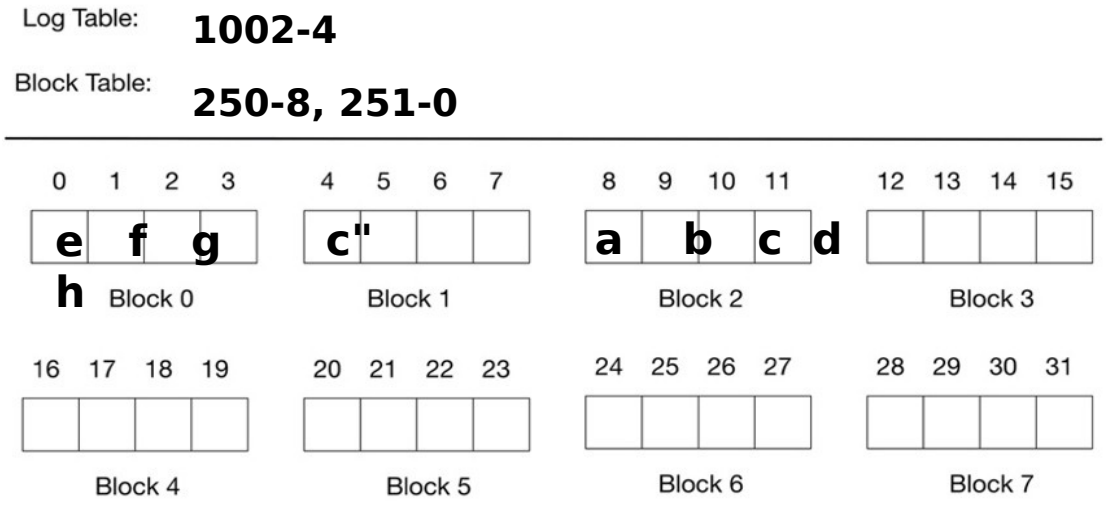
Full merge

练习答案（3/3）

- 初始状态如下图所示，到达的请求序列为 a, b, c, d, e, e, f, g, e, c ， 请画出序列结束后的状态



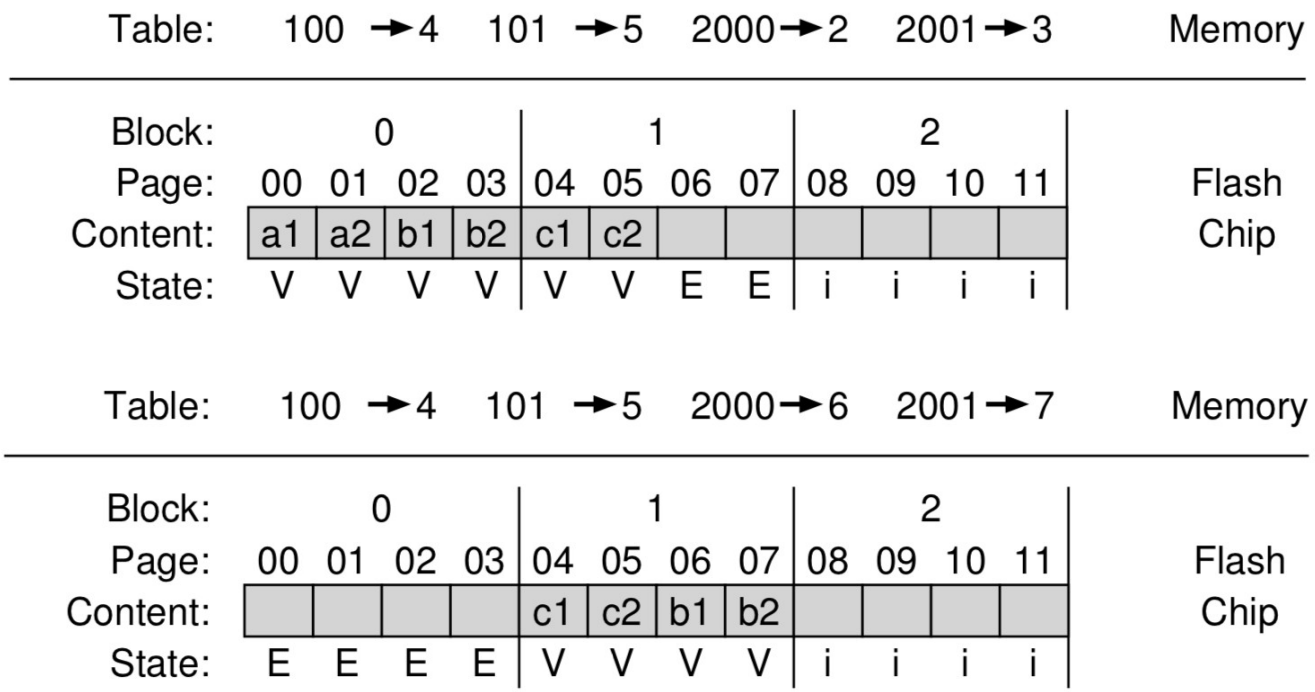
c 到达：



Partial merge

I/O Devices

- **Garbage Collection**
  - 回收 **invalid page** 占用空间
  - 例如 **a1** 和 **a2** 失效后，可以搬移 **b1/b2**，然后擦除



I/O Devices

- **Wear Leveling (WL)**
  - 磨损平衡
  - 静态 WL：定期交换磨损最严重和最不严重的两个 block 中的数据
  - 动态 WL：实时观测磨损情况，尽量使用磨损小的 block

- **WL 关系到 SSD 的写耐久性，也和 GC、Address Mapping 有着紧密的联系**
  - 反映 SSD 写耐久性的两个指标： Drive Write Per Day (DWPD) 和 Petabyte Write (PBW)

    PBW=DWPD*SSD 容量 * 质保天数

I/O Devices

- 更新一代的 SSD
  - Intel Optane
    - 内部采用 PCM 芯片 + 大容量缓存
    - 性能和耐久性比 Flash-based SSD 提升很多
    - 企业级： P4800X 、 P5800X
    - 消费级： 900P

DWPD: 100(P4800X)， 10（900P）

I/O Devices

- 更新一代的 SSD
  - 也有基于 SLC/MLC Flash 优化而来的存储介质：
    - Samsung ZNAND: SZ1735,
    - Kioxia XL-Flash: FL6 系列

DWPD: 30-60

I/O Devices

- 叠瓦式 (SMR) 硬盘
  - Shingled magnetic recording (SMR)
  - 后续技术： BPMR, HAMR, MAMR

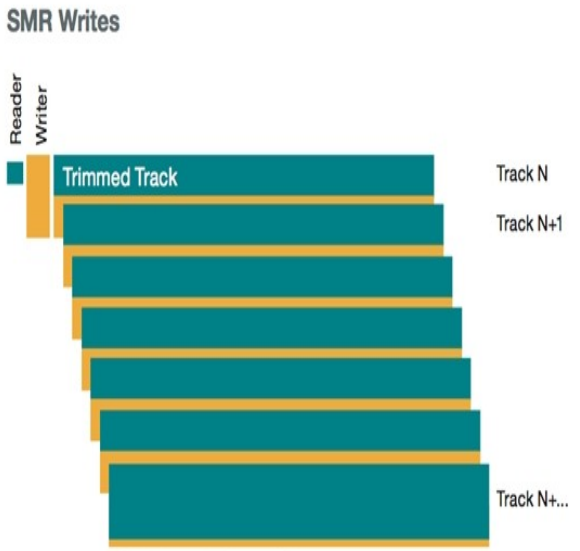
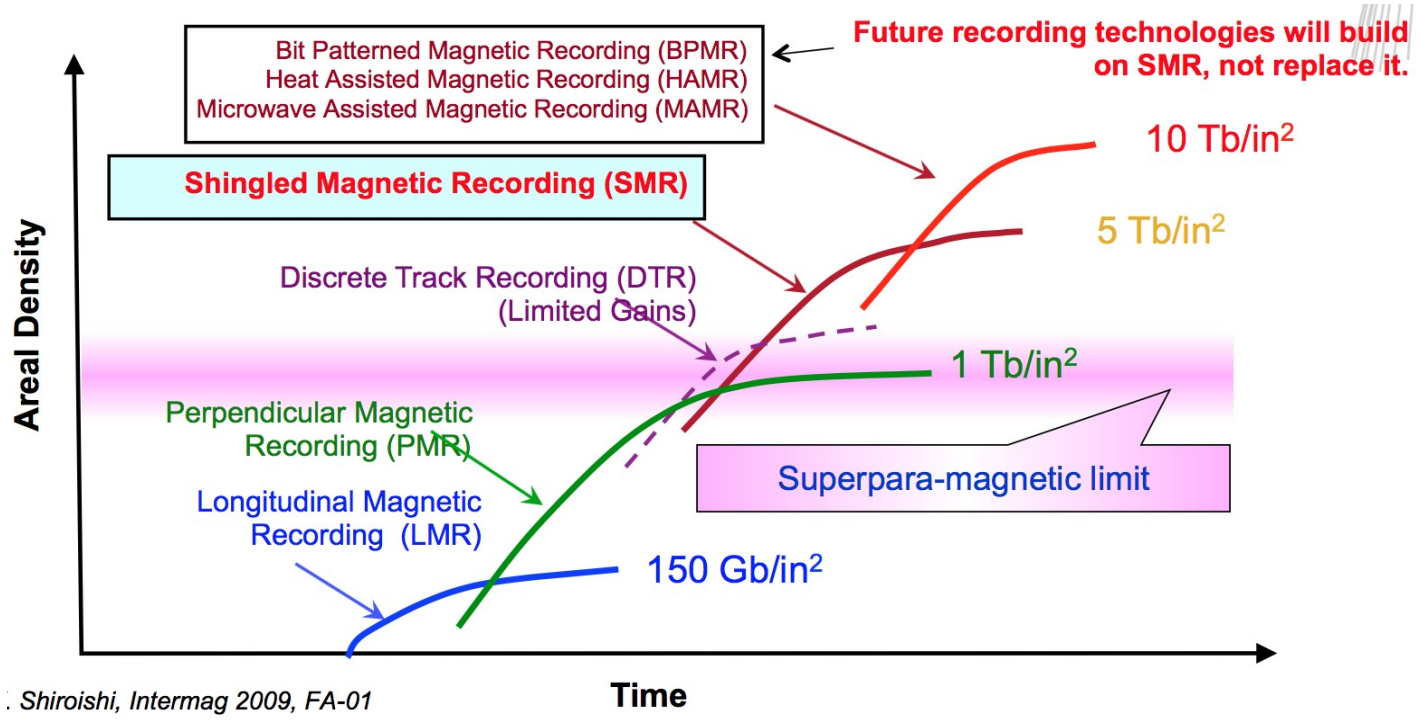


Figure 2. Track Spacing Enabled by SMR Technology

Figure 4. SMR Band Structure

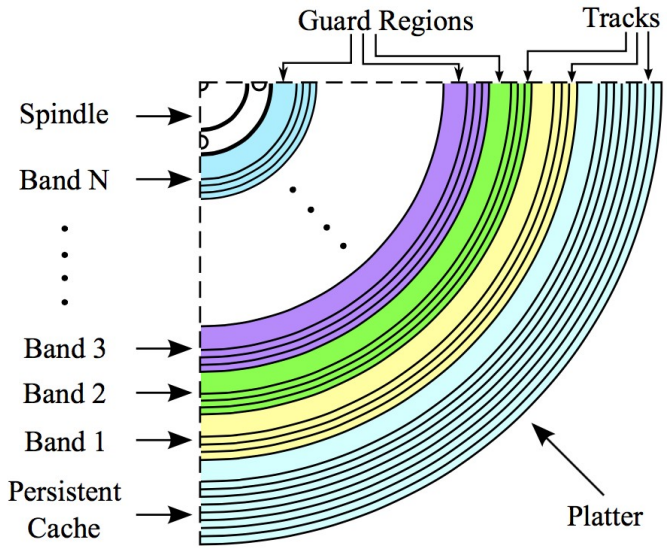
I/O Devices

- 叠瓦式 (SMR) 硬盘
  - 增加存储密度，降低成本，随机写性能较差
  - 适用于备份系统、冷存储、科学计算（天文、生物、医学）、日志等

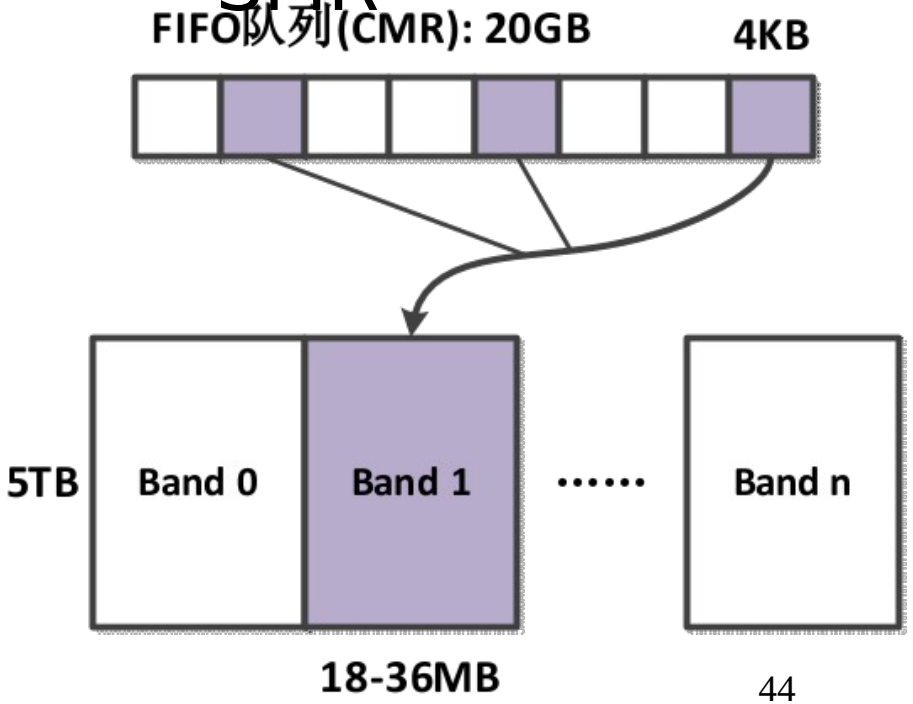


I/O Devices

- **SMR 分类**
  - **Driver Managed**
  - **Host Managed**

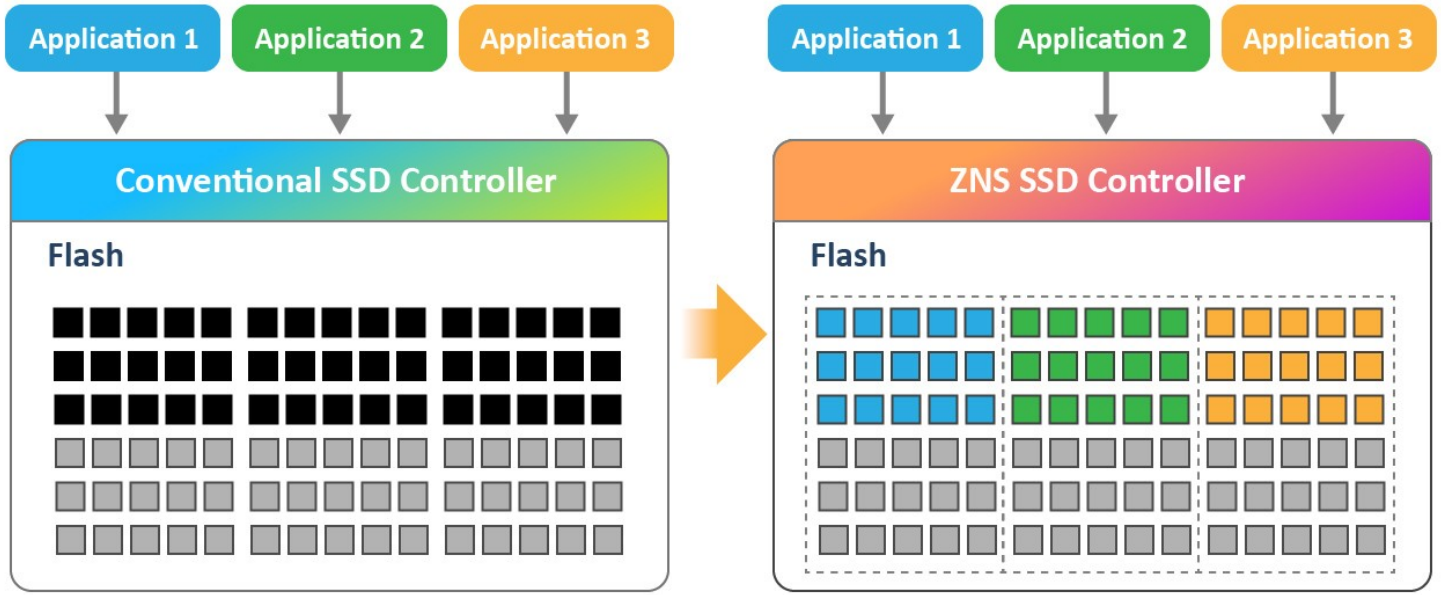


- Seagate 8TB SMR



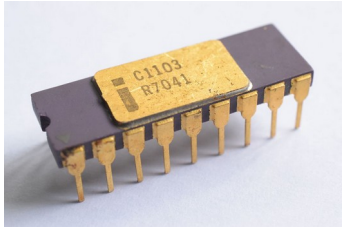
I/O Devices

- **ZNS SSD**
  - **Zoned Namespace SSD**
  - **将 FTL 的部分功能（主要是 GC）上移到应用中，可能减少开销**



扩展内容

国产半导体存储的发展 - DRAM



- 1966 年， IBM 工程师 Robert H. Dennard 发明 DRAM
- 1970 年， Intel 发布第一款商业化 DRAM C1103（1Kb）
- 1975 年，北大物理系与 109 厂 1Kb DRAM
- 1981 年，中科院半导体所 16Kb DRAM
- 1985 年，无锡江南无线电器材厂 64Kb DRAM
- 1990 年前后，日本 DRAM 产业被迫向韩国和美国转移，三星、美光等企业在半导体存储方面开始领先
- 1993 年，无锡华晶 256Kb DRAM，此前一年三星已经发布 16Mb DRAM
- 1999 年，上海华虹引进日本 NEC 技术生产 64Mb DRAM， 2004 年转型晶圆代工

扩展内容

国产半导体存储的发展 - DRAM

- 2004 年，中芯国际开始投产 DRAM，后为奇梦达 (Qimonda，原西门子 DRAM 部门) 等厂商代工 DRAM
- 2008 年，中芯国际收缩业务，放弃 DRAM 制造
- 国产 DRAM 开始长达 8 年的空白期。。。

- 2010s 中，智能手机和数据中心的发展使 DRAM 需求剧增
- 2016 年，福建晋华与台湾联电签订协议，由联电代为研发 DRAM 技术，联电从美光 (Micron) 挖了部分技术人员
  - 2017 年 12 月，美光起诉晋华和联电，晋华和联电反诉
  - 2018 年 10 月，晋华被美国商务部禁运

扩展内容

国产半导体存储的发展 - DRAM

- 奇梦达在 2008 金融危机前是世界第四大 DRAM 厂商，”欧洲半导体存储之光”
  - 2009 年 4 月破产，华芯、浪潮、紫光先后等接手了奇梦达在西安和欧洲的部分研发中心和产线（今天的紫光 DRAM 业务）
  - 2015 年 6 月，奇梦达核心专利被英飞凌打包出售
- 2016 年，合肥长鑫成立，获得了奇梦达 DRAM 相关的核心技术
- 2019 年，长鑫在奇梦达 46nm DRAM 基础上，实现了 10nm DRAM 技术，并继续收购了奇梦达其余专利
- 2023 年，长鑫发布 12Gb LPDDR5 DRAM 颗粒

扩展内容

国产半导体存储的发展 - NAND Flash

- 1986 年，东芝工程师舛冈富士雄博士发明 NAND Flash
- 1991 年，东芝发布 4MB NAND
- 1992 年， Intel 发布 12MB NAND
- 1999 年，三星发布 1GB NAND
  - 2002 年朗科发明 U 盘
- 2007 年，东芝发布 3D NAND 技术 BiCS
- 2013 年，三星发布 3D NAND 技术 V-NAND
- 2016 年， Intel 和美光发布 3D NAND 技术 CuA
- 2016 年，武汉长江存储成立，进入 NAND 行业
- 2018 年，长江存储发布 3D NAND 技术 XTacking，以及一款 32 层国产 NAND
  - 此时国际领先水平是 96 层

国产半导体存储的发展 - NAND Flash

- 2019 年 9 月，长江存储发布基于 XTacking 的 64 层 NAND
  - 此时国际领先水平是 128 层
- 2020 年 4 月，长江存储试产 128 层 NAND，并于 2021 年 9 月量产（此时国际领先水平是 192 层）
- 2022 年 8 月，长江存储试产 232 层 NAND，领先或追平三星、 Intel/ 美光、 kioxia （原东芝 NAND 部门）
- 2022 年 12 月， TechInsights 在一款海康威视的 SSD 中拆解确认了长江存储 232 层 XTacking NAND 已经商用
  - TechInsights 称这是其确认的世界上首款商用 200+ 层 3D NAND
  - 2022 年 12 月底，美国商务部将长江存储加入实体清单
  - 2023 年 3 月，国家半导体大基金宣布对长江存储增加 129 亿投

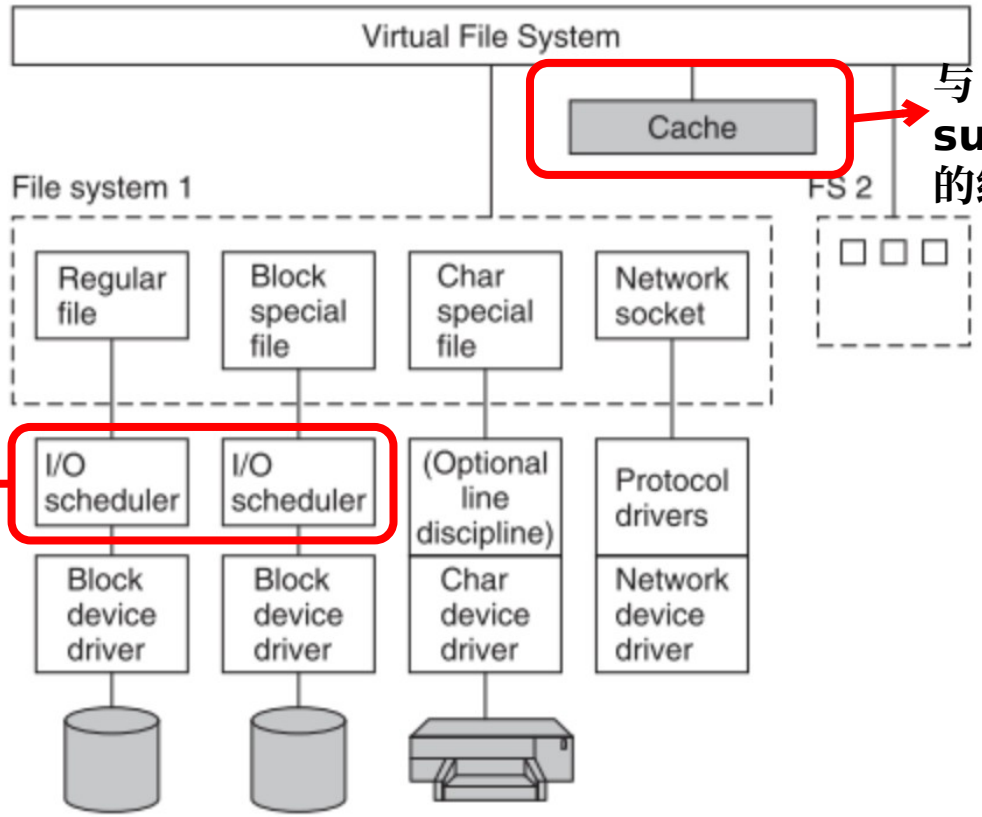
技术发展的根本动力是市场需求

Outline

- I/O Devices
- Unix I/O
- Reading File Metadata
- Sharing Files & I/O redirection
- Robust I/O
- Standard I/O
- Fast I/O

The I/O subsystem in Unix/Linux-like OS

Unix I/O



与 memory subsystem 的纽带

Linux 中的 I/O scheduler 负责对 FS 中产生的 read/write requests 进行 reordering 和 merging，以协助 device 提高 I/O 效率，常用的包括：

- noop
- cfq (completely fair queuing)
- deadline

Why Unix I/O

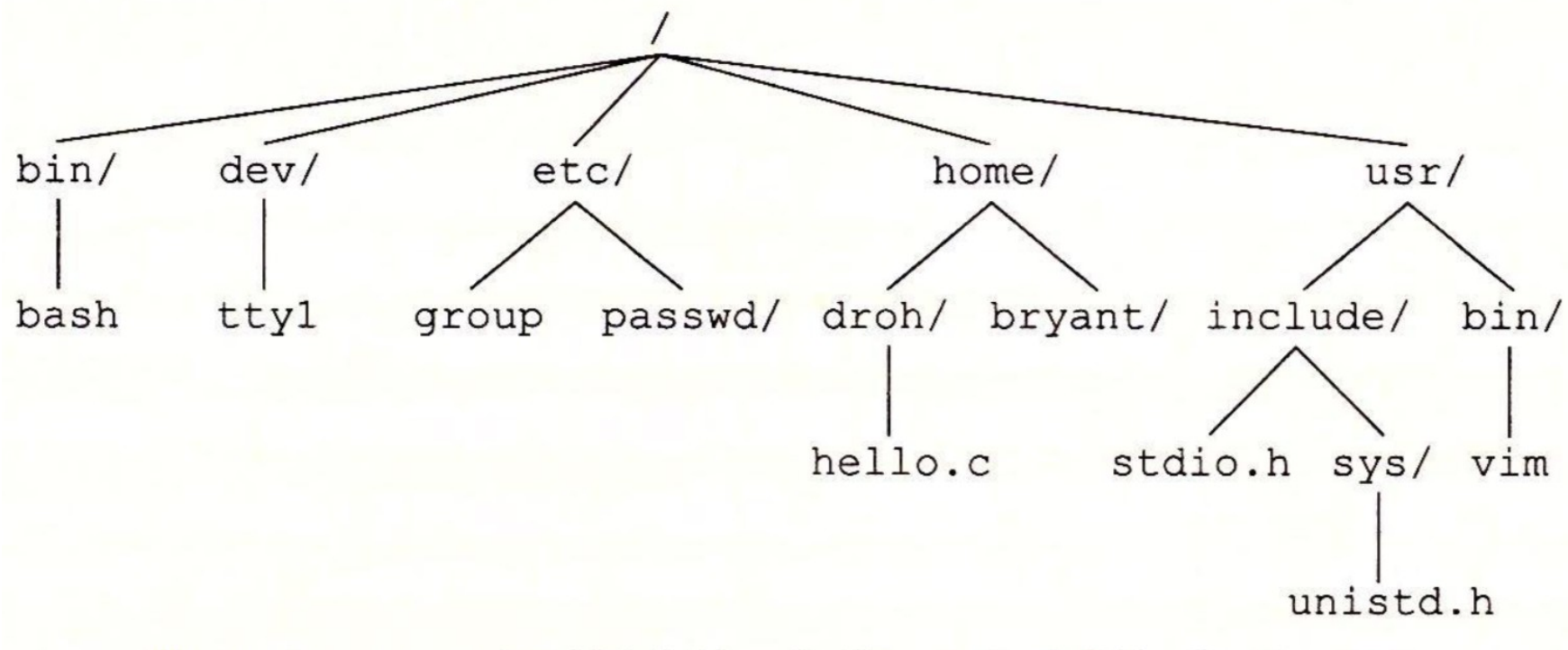
- **Input/Output**
  - 在内存和外部设备之间进行数据拷贝的过程
- **Standard I/O library**
  - 用户态的 I/O 函数集合
- **Unix I/O**
  - 由 kernel 提供的 I/O 函数集合，是 I/O 相关的 system call

Unix I/O

- Unix File 是一个字节序列 (byte sequence)
  - B_0, B_1, ……, B_k, ……, B_m
- 所有的 I/O devices 都被当作 **files**
  - such as networks, disks, and terminals
- 所有的 I/O 操作都由对文件的读写操作完成

Unix I/O

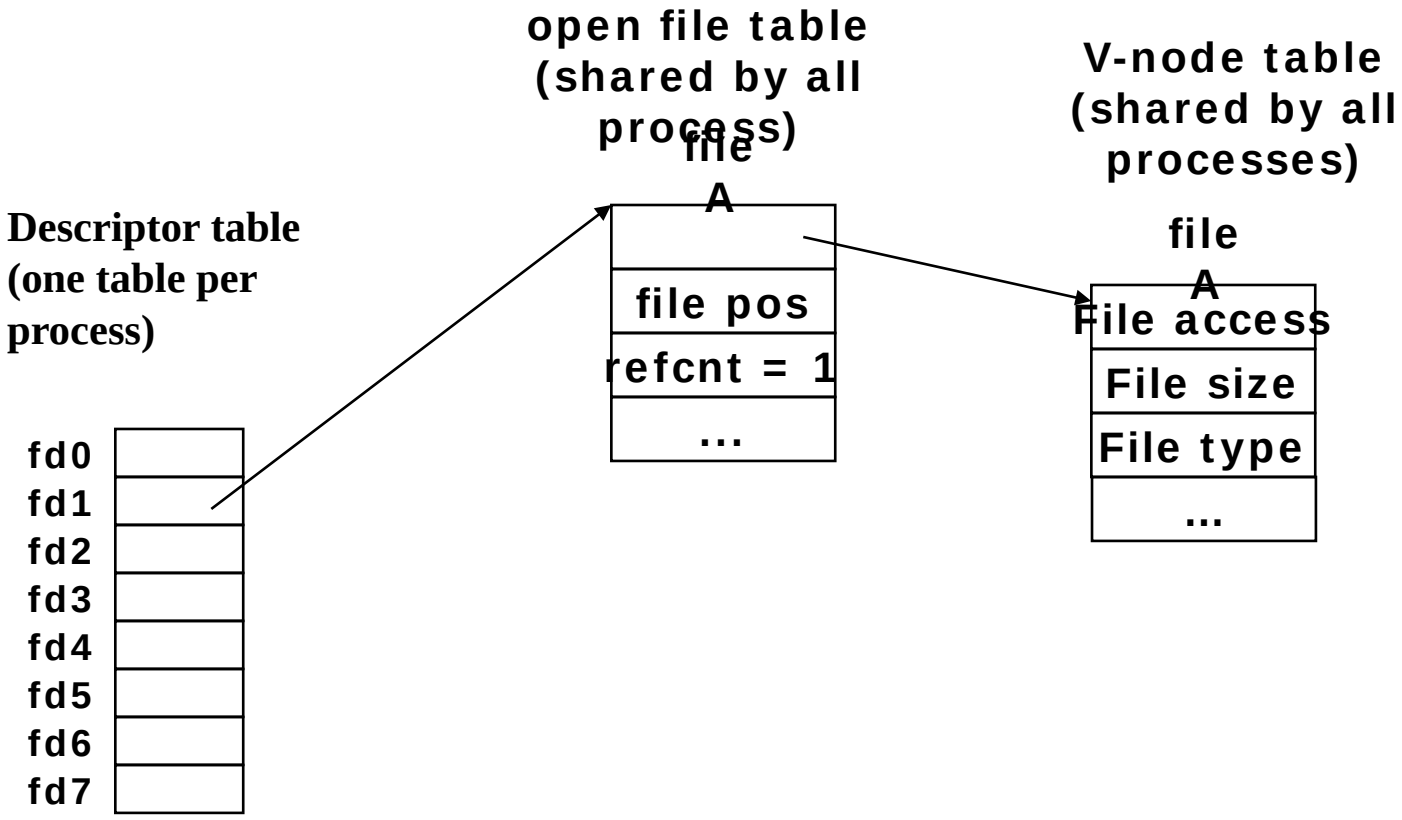
- **Linux 系统目录层次结构（部分）**



Open Files

- 应用访问 I/O 设备前需要先打开文件
  - Kernel 每个打开的文件都有一个非负整数的文件描述符 (file descriptor, fd)
    - 后续文件操作都以 fd 标识文件
  - Kernel 维护每个已打开文件的信息，包括：
    - file position *k*，初始为 0

Kernel Data Structures for Files



Descriptor table

- 每个进程有自己独立的 descriptor table
  - 每个文件描述符是一个非负数字，作为 descriptor table 的索引
    - 0：标准输入（默认已打开）
    - 1 ：标准输出（默认已打开）
    - 2 ：标准错误（默认已打开）
    - 其他打开的文件，从 3 开始编号
  - 每个打开的文件描述符指向一个 open file table 中的一个 entry

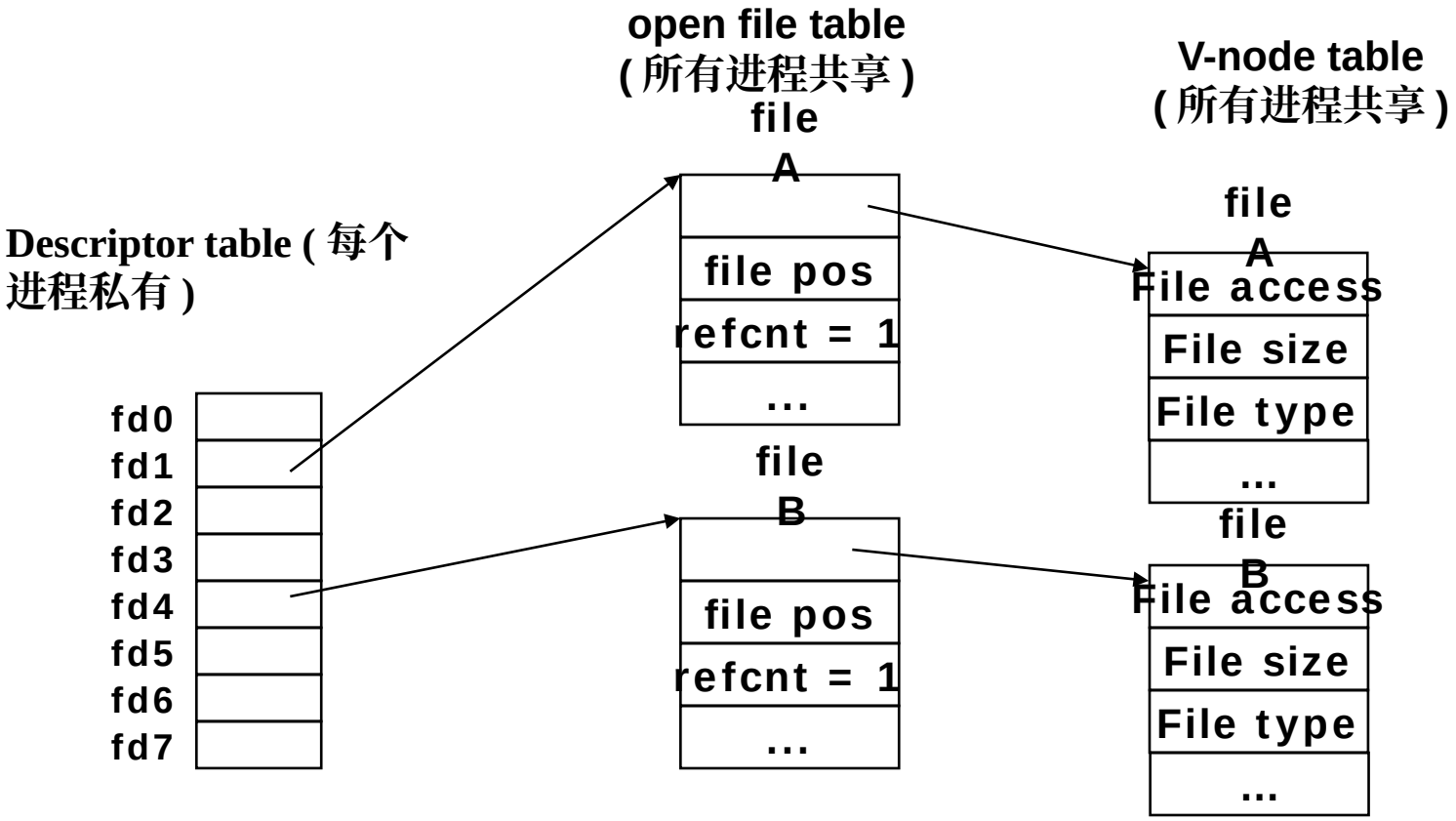
(Open) File table

- 进程中 打开的文件被记录在一个 **file table** 中
- **File table** 被所有进程共享
- 每个 **file table entry** 包含
  - 当前文件的读写偏移 **(position)**
  - 文件的引用计数 **(reference count)**
  - 指向 **v-node** 的指针
  - 当引用计数为 **0** 时， **kernel** 会删除 **file table entry**

V-node table

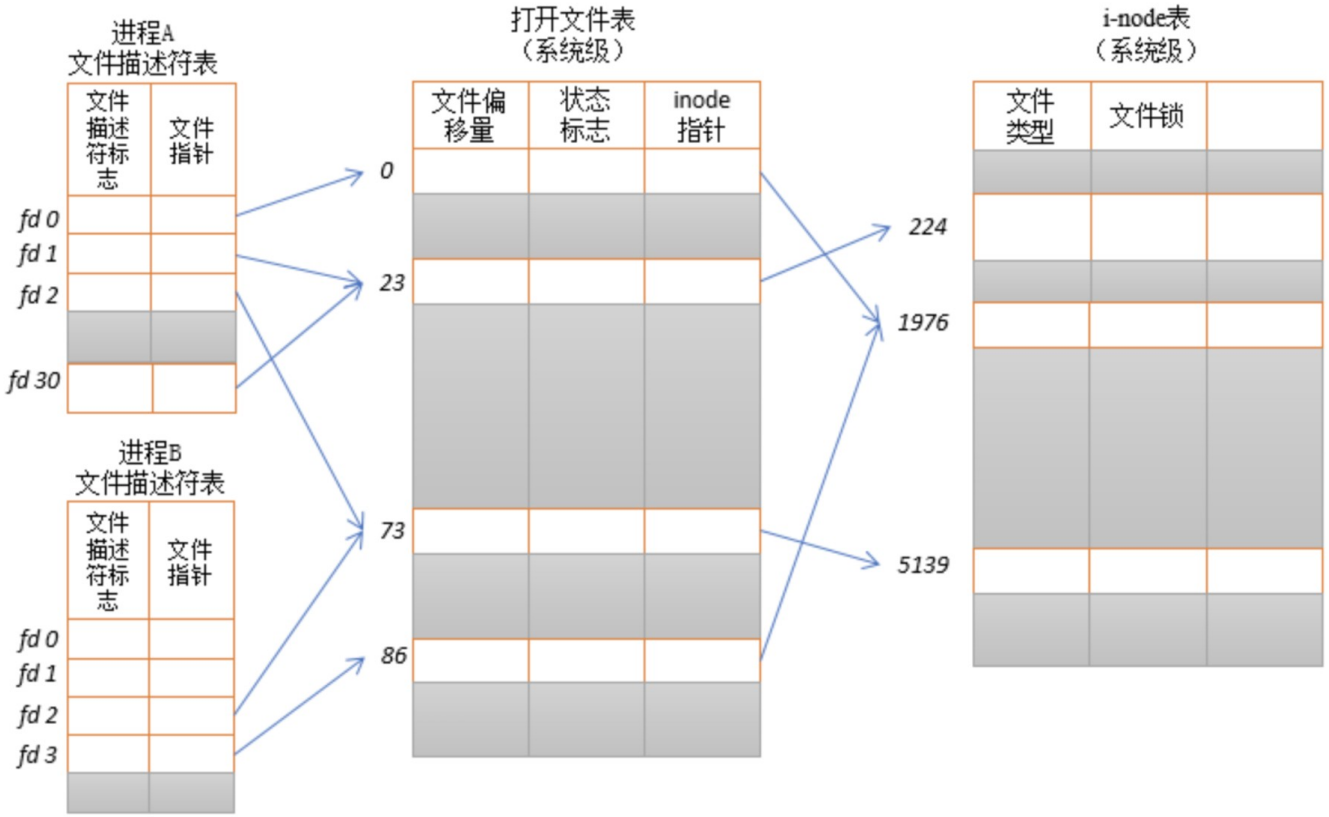
- V-node 也是所有进程共享的
- 每个 entry 包含文件的大部分 metadata

文件相关的内核数据结构



例子

- 可能一个进程多个 fd 对应同一个打开的文件 (如 dup2)
- 可能多个进程对应同一个打开文件 (如 fork)
- 可能多个进程对应不同打开文件，但是同一个文件



打开文件

- 表明一个应用要访问一个文件的意图
  - 应用程序只需要维护文件描述符，其他信息都在 kernel 数据结构中
  - 一个应用程序可以通过 seek 操作，显式的设置文件的当前访问位置
    - Read/write 操作都从当前位置开始向后进行

打开文件

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


int open(char *filename, int flags, mode_t mode);
                Returns: new file descriptor if OK, -1 on
                                                    error


flags
    – O_RDONLY, O_WRONLY, O_RDWR (must have one)
    – O_CREAT, O_TRUNC, O_APPEND, O_DIRECT (optional)
mode
    – S_IRUSR, S_IWUSR, S_IXUSR
    – S_IRGRP, S_IWGRP, S_IXGRP
    – S_IROTH, S_IWOTH, S_IXOTH
```

打开文件

- **Flags:**
  - **O_CREAT: 如果文件不存在，就创建一个空文件**
  - **O_TRUNC : 如果文件已经存在，就截断它**
    - **再写入时，从头写入，覆盖原来的内容**
  - **O_APPEND: 在每次写入操作钱，设置文件位置到文件的结尾处**
  - **O_DIRECT: 跳过 (by pass) cache 直接以读写文件**

- **Note: open 集成了 create file 功能**

打开文件

- **Mode:**

掩码	描述
S_IRUSR	使用者（拥有者）能够读这个文件
S_IWUSR	使用者（拥有者）能够写这个文件
S_IXUSR	使用者（拥有者）能够执行这个文件
S_IRGRP	拥有者所在组的成员能够读这个文件
S_IWGRP	拥有者所在组的成员能够写这个文件
S_IXGRP	拥有者所在组的成员能够执行这个文件
S_IROTH	其他人（任何人）能够读这个文件
S_IWOTH	其他人（任何人）能够写这个文件
S_IXOTH	其他人（任何人）能够执行这个文件

打开文件

umask(): set mask of process

利用 **umask** 命令可以指定哪些权限将在新文件的默认权限中被删除

```
#define DEF_MODE S_IRUSR | S_IWUSR |      \
                 S_IRGRP | S_IWGRP |      \
                 S_IROTH | S_IWOTH
#define DEF_UMASK S_IWGRP | S_IWOTH

umask(DEF_UMASK);
fd = open (“foot.txt”,
           O_CREAT|O_TRUNC|O_WRONLY, DEF_MODE);
/* permission: S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH*/
```

打开文件

- 一个进程的文件描述符有上限
  - 超过了会报错：“Too many open files”
  - 单个进程的上限一般是 1024
    - 可以通过 ulimit –n 查询
  - 系统的最大值： sysctl -a | grep fs.file-max
  - 及时关闭不再被使用的文件描述符

关闭文件

- **Kernel：**
  - 释放打开文件时创建的数据结构
  - 将 fd 放回可用 fd 的 pool 中
    - 下一个打开的文件将获得 pool 中最小的可用 fd

关闭文件

- 当进程终止时， kernel：
  - 默认关闭所有已打开的文件
  - 释放内存资源

```
#include <unistd.h>
int close(int fd) ;
            Returns: zero if OK, -1 on error
```

课堂练习

```
1   #include "csapp.h"
2
3   int main()
4   {
5       int fd1, fd2;
6
7       fd1 = Open("foo.txt", O_RDONLY, 0);
8       Close(fd1);
9       fd2 = Open("baz.txt", O_RDONLY, 0);
10      printf("fd2 = %d\n", fd2);
11      exit(0);
12  }
```

屏幕输出什么?

读写文件

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
                    returns: number of bytes read if OK,
                                        0 on EOF, -1 on error

ssize_t write(int fd, const void *buf, size_t count);
                  returns: number of bytes written if OK,
                                              -1 on error
```

读写文件

- 一次 read 操作:
  - **从文件拷贝 m > 0 字节到内存中**
    - **从当前的 file position k 开始**
    - **k+=m**
  - **如果文件中剩余的字节数少于 m，会触发** end-of-file **(EOF)**
    - **文件的末尾没有 "EOF character"**

读写文件

```
1 #include "csapp.h"
2
3 int main(void)
4 {
5       char c;
6
7       /* copy stdin to stdout, one byte at a time
  */
8       while(Read(STDIN_FILENO, &c, 1) != 0)
9             Write(STDOUT_FILENO, &c, 1);
10      exit(0);
11 }
```

STDIN_FILENO(0), STDOUT_FILENO(1), STDERR_FILENO(2)

读写文件

- ssize_t vs. size_t
  - X86-64:
    - ssize_t: signed long
    - size_t: unsigned long
  - Read 和 write 可能传输比应用请求更少的字节数
    - read 遇到了 EOF
    - 从 terminal 中读取字符串
    - 读写 network sockets

Outline

- I/O Devices
- Unix I/O
- Reading File Metadata
- Sharing Files & I/O redirection
- Robust I/O
- Standard I/O
- Fast I/O

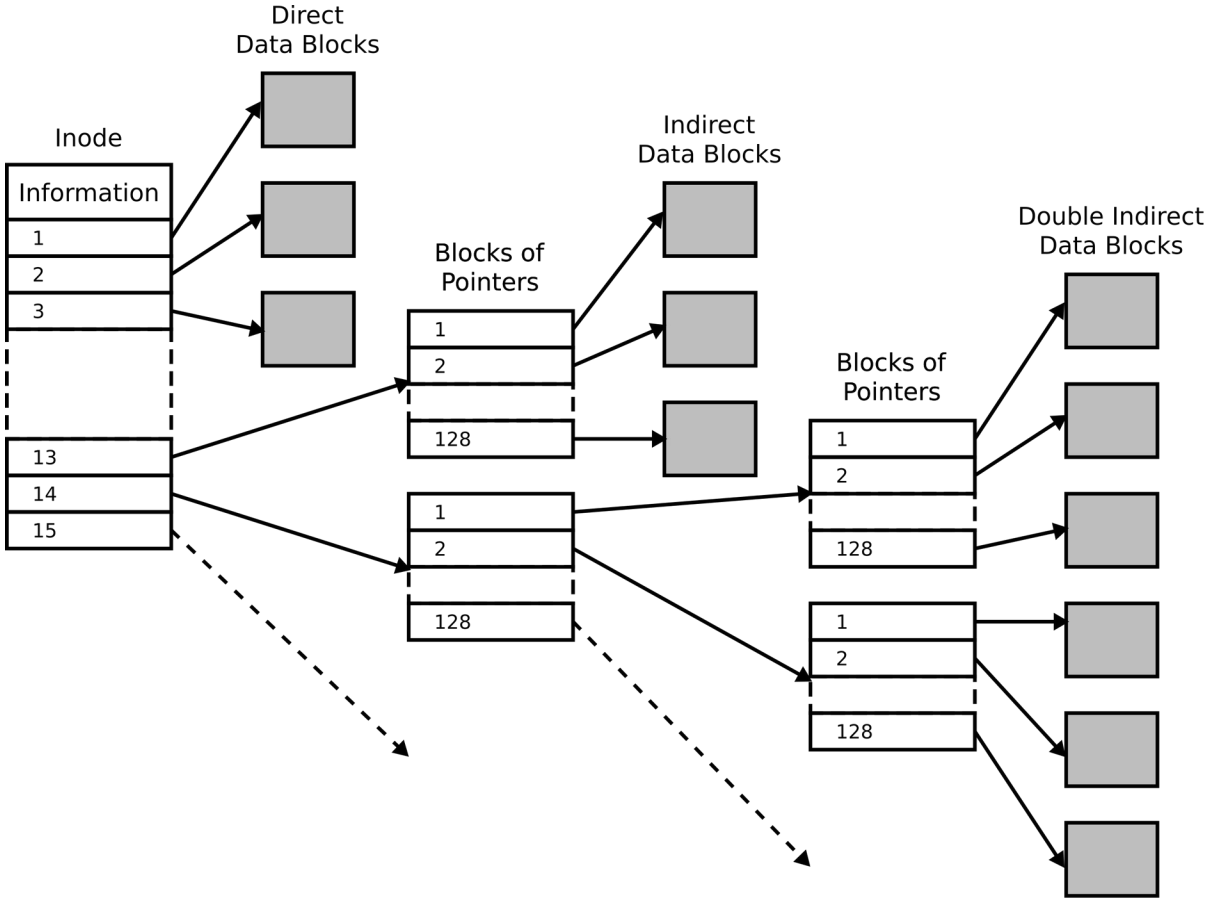
读取文件 Metadata

```
#include <unistd.h>

#include <sys/stat.h>

int stat(const char *filename, struct stat *buf);

int fstat(int fd, struct stat *buf) ;

                          returns: 0 if OK, -1 on error
```

读取文件 Metadata

```
/* file info returned by the stat function */
struct stat {
   dev_t st_dev;          /* device */
   ino_t st_ino;          /*  inode number */
   mode_t st_mode;              /* protection and file type */
   nlink_t st_nlink;            /* number of hard links */
   uid_t st_uid;          /* user ID of owner */
   gid_t st_gid;          /* group ID of owner */
   dev_t st_rdev;         /* device type (if inode device) */
   off_t st_size;         /* total size, in bytes */
   unsigned long st_blksize;     /* blocksize for filesystem I/O
   */
   unsigned long st_blocks;      /* number of blocks allocated */
   time_t st_atime;              /* time of last access */
   time_t st_mtime;              /* time of last modification */
   time_t st_ctime;              /* time of last change */
};
```

inode and vnode



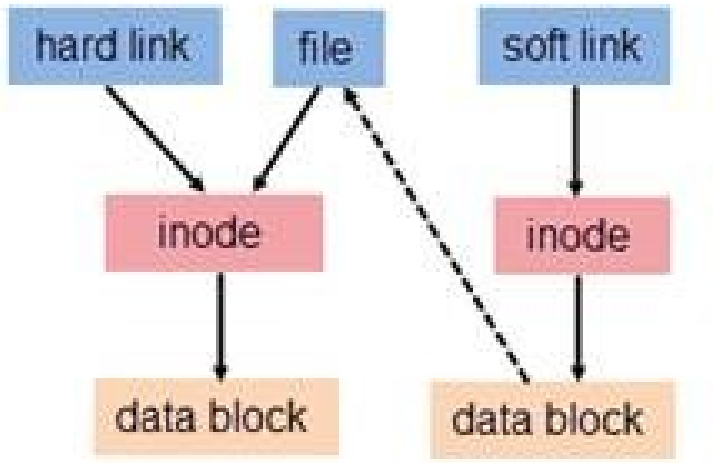
inode 由文件系统保存在外存设备上，打开文件时， inode 加载到内存中，构建对应的 vnode
在 Linux 系统中， vnode 通常称为 generic inode(定义为 struct inode)

硬链接和软链接

- 解决文件共享使用问题
- 硬链接 (hard link)
  - 一个 inode 号对应多个文件名
  - 同一个文件使用了多个别名
  - 硬链接可由命令 link 或 ln 创建
  - 文件有相同的 inode 及 data block；
  - 只能对已存在的文件进行创建；
  - 不能交叉文件系统进行硬链接的创建；
  - 不能对目录进行创建，只可对文件创建；
  - 删除一个硬链接文件并不影响其他有相同 inode 号的文件

硬链接和软链接

- 软链接
  - 又称符号链接，即 soft link 或 symbolic link
  - 文件用户数据块中存放的内容是另一文件的路径名的指向，则该文件就是软连接
  - 软链接就是一个普通文件，只是数据块内容有点特殊。软链接有着自己的 inode 号以及用户数据块



硬链接和软链接

```
$ ln myfile hard
$ ls -li

25869085 -rw-r--r--  2 unixzii  staff  27  7  8 17:39 hard
25869085 -rw-r--r--  2 unixzii  staff  27  7  8 17:39 myfile
```

```
$ ln -s myfile soft
$ ls -li

25869085 -rw-r--r--  2 unixzii  staff  36  7  8 17:45 hard
25869085 -rw-r--r--  2 unixzii  staff  36  7  8 17:45 myfile
25869216 lrwxr-xr-x  1 unixzii  staff   6  7  8 17:47 soft -> myfile
```

```
int main(int argc, char **argv)
{
    struct stat stat ;
    char *type, *readok ;
    Stat(argv[1], &stat) ;
    if (S_ISREG(stat.st_mode)) /* Determine file type */
        type = “regular” ;
    else if (S_ISDIR(stat.st_mode))
        type = “directory” ;
    else
        type = “other” ;


    if ((stat.mode & S_IRUSR)) /* check read access */
        readok = “yes”;
    else
        readok = “no”;

    printf(“type: %s, read: %s\n”, type, readok) ;
    exit(0) ;
 }
```

S_ISREG(): Is this a regular file?
S_ISDIR(): Is this a directory file?

读取目录

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir (const char *name);
```

returns: 成功则返回处理的指针；若出错，则返回 NULL

```
struct dirent *readdir (DIR *dirp);
```

returns: 成功则返回指向下一个目录项的指针；没有更多目录或出错则返回 NULL

读取目录

- 目录项

```
struct dirent {
        ino_t d_ino;                /* inode number
*/
        char d_name[256];    /* filename */
}
```

有些 linux 版本包括更多内容

读取目录

```
#include <dirent.h>


int closedir (DIR *dirp);
```

returns: 成功则返回 0；若出错，则返回 -1

读取目录

```
1   #include "csapp.h"
2
3   int main(int argc, char **argv)
4   {
5       DIR *streamp;
6       struct dirent *dep;
7
8       streamp = Opendir(argv[1]);
9
10      errno = 0;
11      while ((dep = readdir(streamp)) != NULL) {
12          printf("Found file: %s\n", dep->d_name);
13      }
14      if (errno != 0)
15          unix_error("readdir error");
16
17      Closedir(streamp);
18      exit(0);
19  }
```

Outline

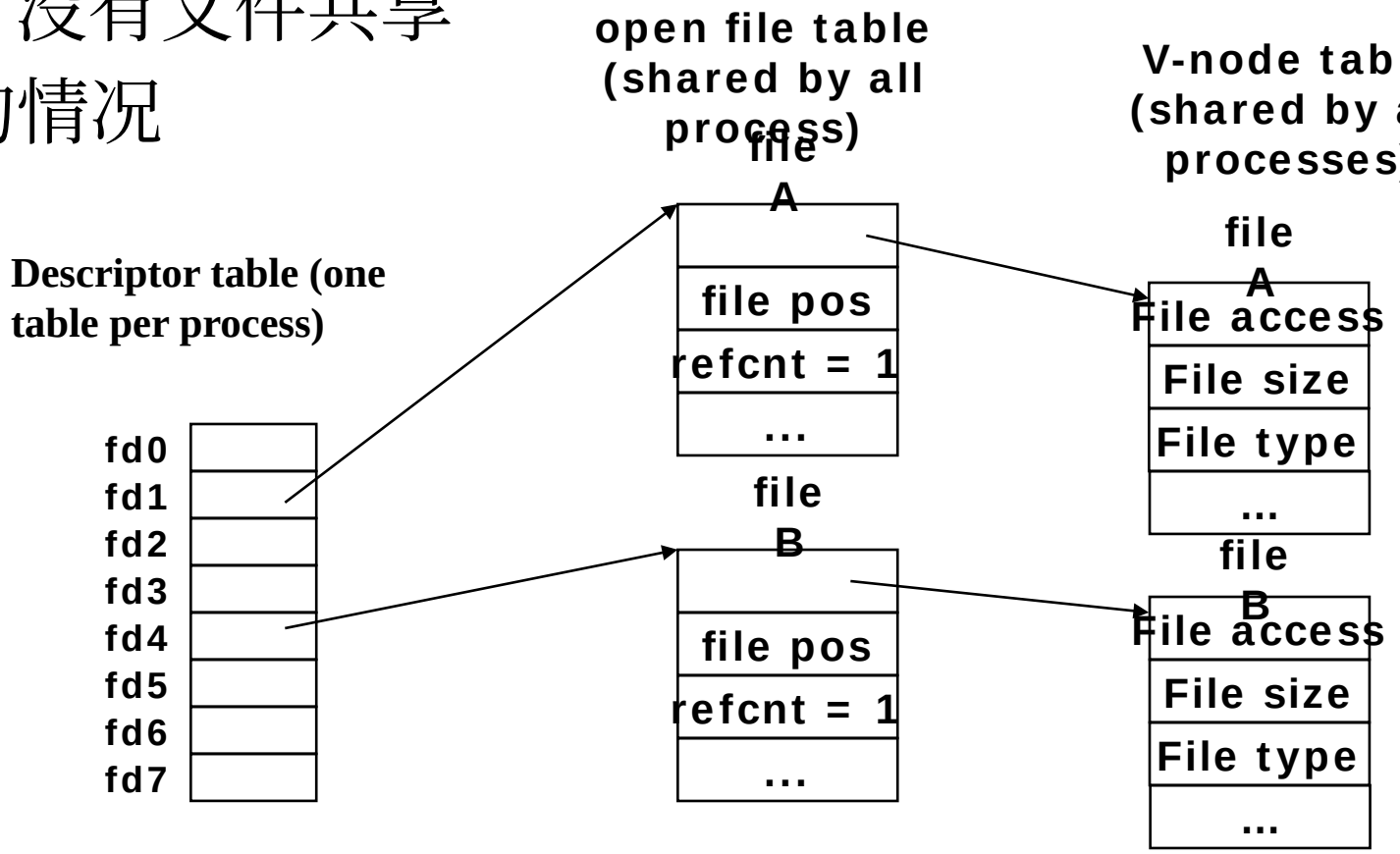
- I/O Devices
- Unix I/O
- Reading File Metadata
- Sharing Files & I/O redirection
- Robust I/O
- Standard I/O
- Fast I/O

V-Node table

- 系统中所有进程共享
- 每个表项包含 **stat** 数据结构的大部分信息，如：
  - **st_mode**
  - **st_size**

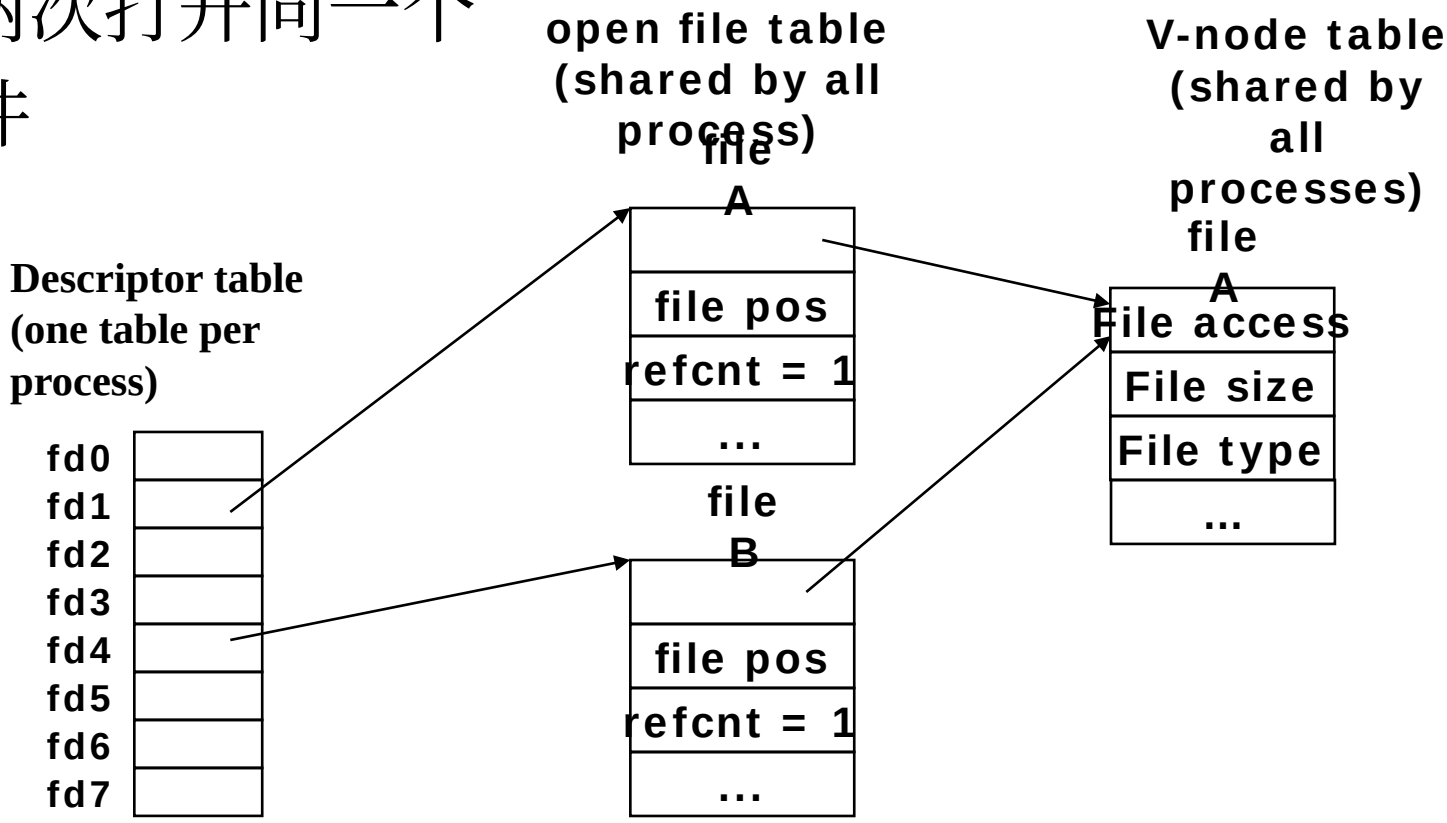
共享文件

- 没有文件共享的情况



共享文件

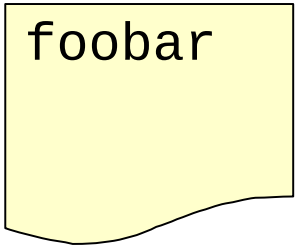
- 两次打开同一个文件



共享文件

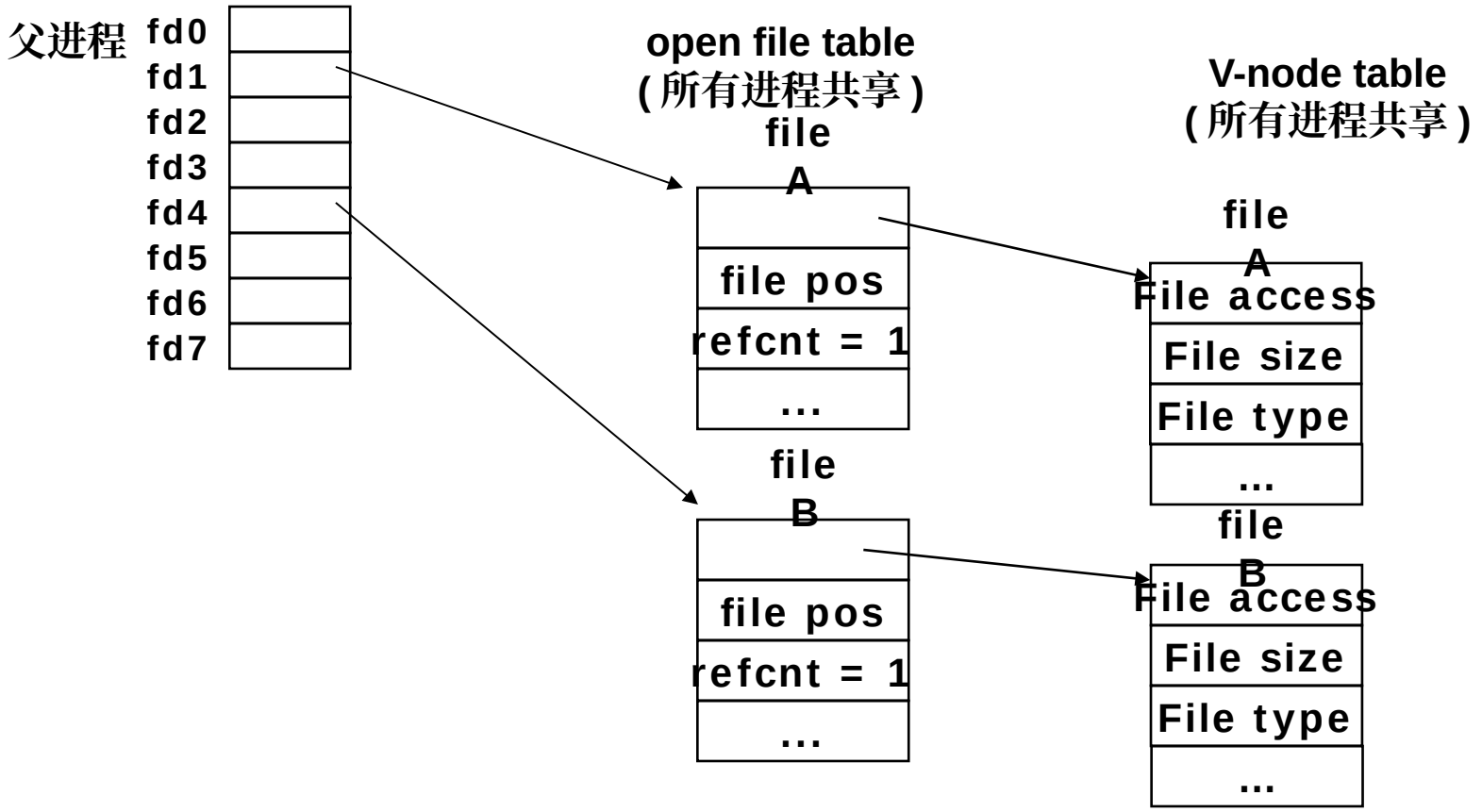
```
#include “csapp.h”

int main()
{
      int fd1, fd2;
      char c;

      fd1 = open(“foobar.txt”, O_RDONLY, 0);
      fd2 = open(“foobar.txt”, O_RDONLY, 0);
      read(fd1, &c, 1);
      read(fd2, &c, 1);
      printf(“c = %c\n”, c);
      exit(0)
}
```

foobar.txt

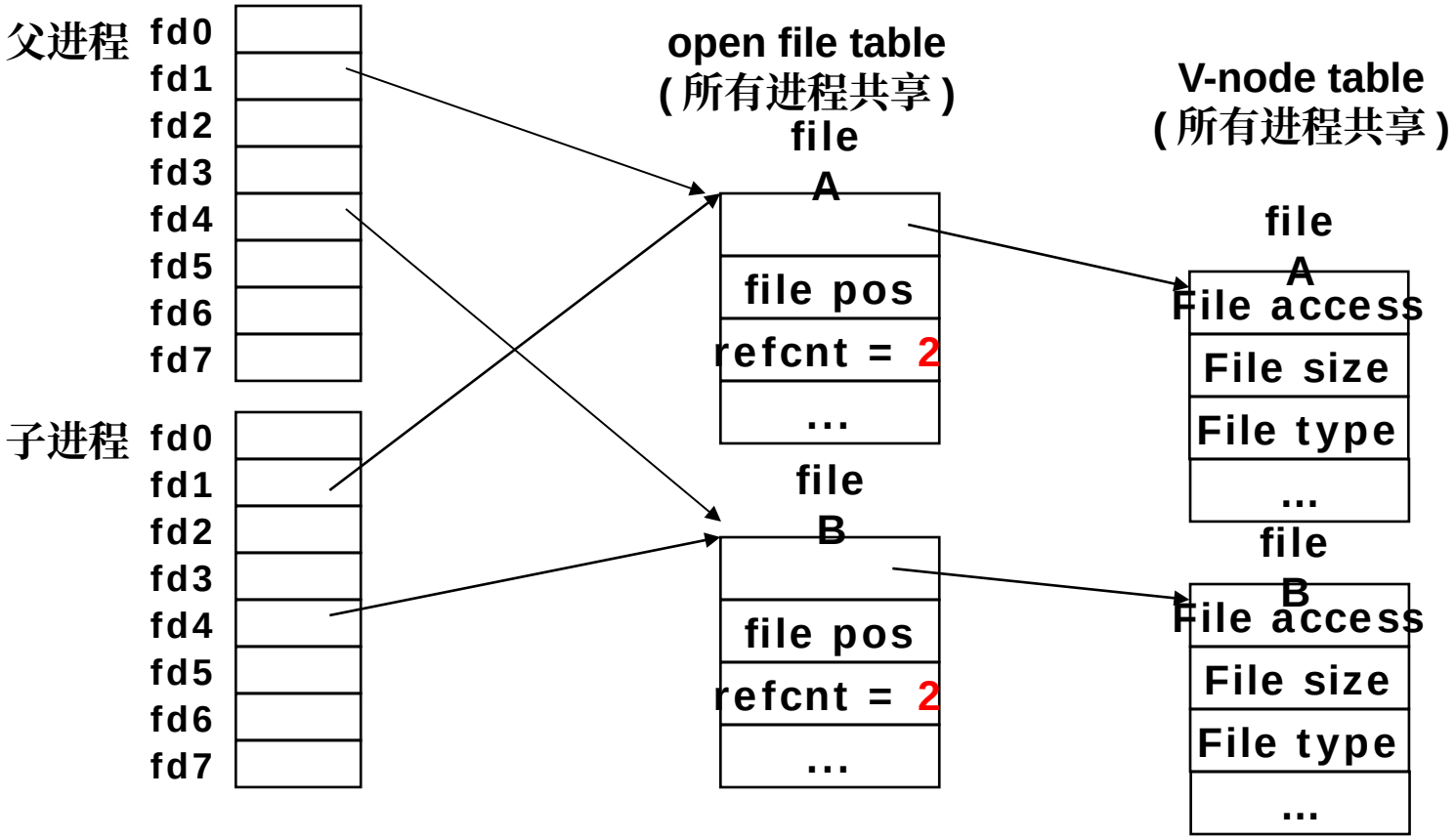


屏幕输出什么?

父子进程间共享文件


父进程
fd0
fd1
fd2
fd3
fd4
fd5
fd6
fd7
open file table
(所有进程共享)
file
A
file pos
refcnt = 1
...
file
B
file pos
refcnt = 1
...
V-node table
(所有进程共享)
file
A
File access
File size
File type
...
file
B
File access
File size
File type
...

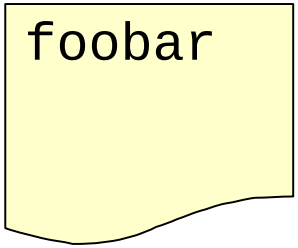
父子进程间共享文件


父进程
fd0
fd1
fd2
fd3
fd4
fd5
fd6
fd7
子进程
fd0
fd1
fd2
fd3
fd4
fd5
fd6
fd7
open file table
(所有进程共享)
file A
file pos
refcnt = 2
...
file B
file pos
refcnt = 2
...
V-node table
(所有进程共享)
file A
File access
File size
File type
...
file B
File access
File size
File type
...

父子进程间共享文件

```
#include “csapp.h”

int main()
{
        int fd;
        char c;

        fd = open(“foobar.txt”, O_RDONLY, 0);
        if (fork() == 0) {
                read(fd, &c, 1) ;
                exit(0) ;
        }
        wait(NULL);
        read(fd, &c, 1);
        printf(“c = %c\n”, c);
        exit(0);
}
```

foobar.txt



屏幕输出什么?
为什么?

I/O 重定向（Redirection）

- unix > ls > foo.txt
  - 重定向操作， ls 本来应该输出到屏幕（标准输出, 1）
  - 但 > 将其改为 foo.txt
  - 如何实现?
  - 将 fd=1 的指针修改，指向打开的 foo.txt 文件
    - 通过 Dup2 系统调用实现
  - Ps. 与 ls | grep xxx 的区别
    - pipe 是具有 2 个文件描述符的特殊 in-memory buffer

I/O Redirction

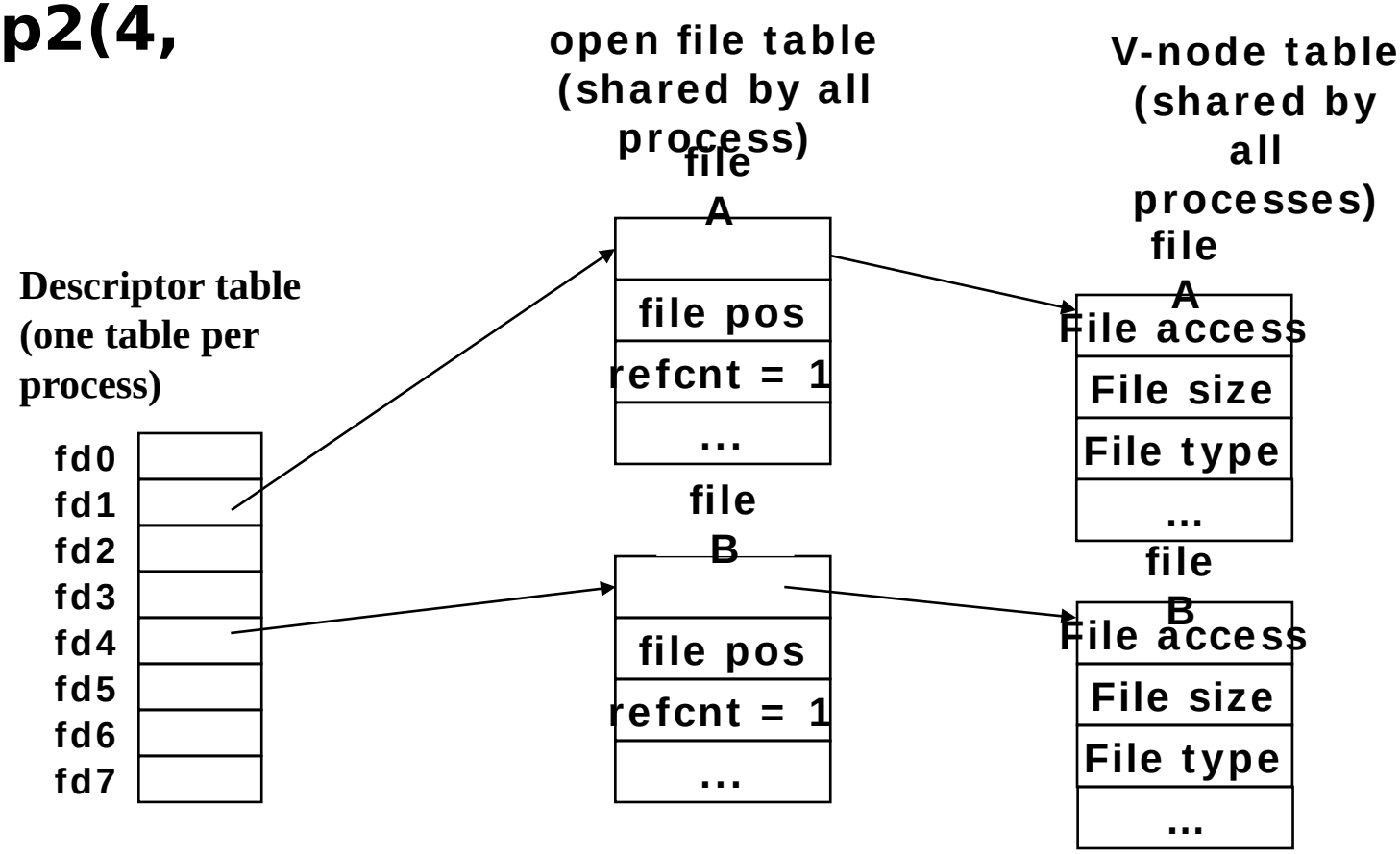
- dup2 可以复制 fd table 的表项

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
          returns: nonnegative descriptor if OK,
                                        -1 on error
```

- dup2(oldfd, newfd) 用 oldfd 在 fd table 中的 entry 覆盖 newfd 在 fd table 中的 entry.
- 相当于强行将 newfd 指向 oldfd 的 open file

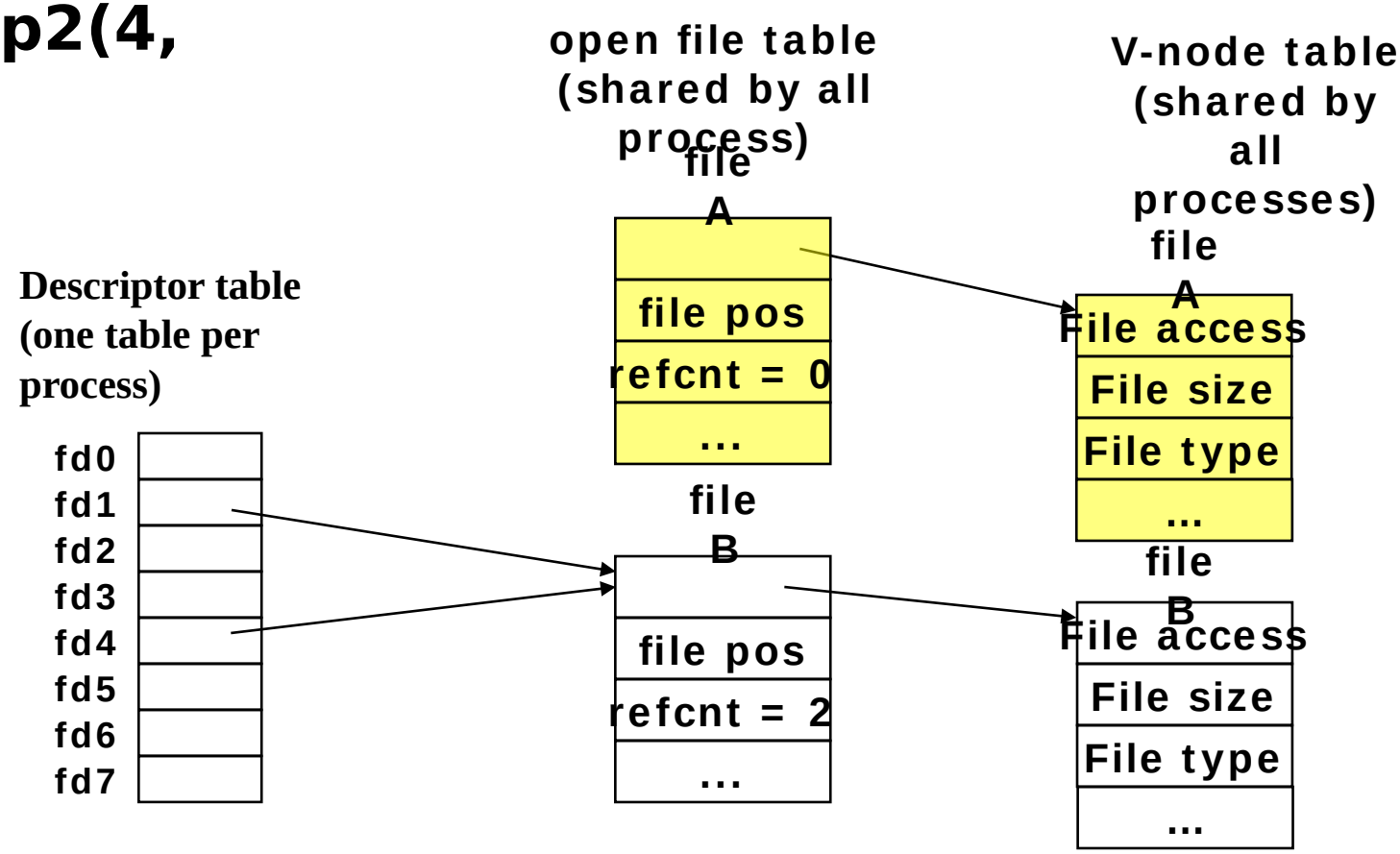
I/O Redirection

dup2(4, 1)



Redirection

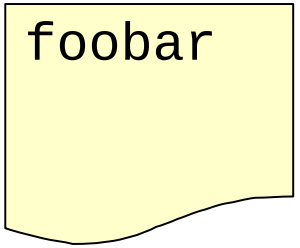
dup2(4, 1)



I/O Redirection

```
#include “csapp.h”

int main()
{
      int fd1, fd2;
      char c;

      fd1 = open(“foobar.txt”, O_RDONLY, 0);
      fd2 = open(“foobar.txt”, O_RDONLY, 0);
      read(fd2, &c, 1);
      dup2(fd2, fd1);
      read(fd1, &c, 1);
      printf(“c = %c\n”, c);
      exit(0);
}
```

foobar.txt



屏幕输出什么?

Outline

- I/O Devices
- Unix I/O
- Reading File Metadata
- Sharing Files & I/O redirection
- Robust I/O
- Standard I/O
- Fast I/O

The RIO Package

- **RIO is a set of wrappers that provide efficient and robust I/O in apps, such as network programs that are subject to short counts**

- **RIO provides two different kinds of functions**
  - **Unbuffered input and output of binary data**
    - **`rio_readn` and `rio_writen`**
  - **Buffered input of binary data and text lines**
    - **`rio_readlineb` and `rio_readnb`**

Robust I/O

- **Unbuffered Input and Output**
  - **Transfer data directly between memory and a file, with no application-level buffering**

```
#include "csapp.h"
ssize_t rio_readn(int fd, void *usrbuf, size_t count);
ssize_t rio_writen(int fd, void *usrbuf, size_t count);
      return: number of bytes read (0 if EOF) or written,
                                                -1 on error
```

```
1  ssize_t rio_readn(int fd, void *buf, size_t count)
2  {
3          size_t nleft = count;
4          ssize_t nread;
5          char *ptr = buf;
6
7          while (nleft > 0) {
8              if ((nread = read(fd, ptr, nleft)) < 0) {
9                  if (errno == EINTR)
10                     nread = 0; /* and call read() again */
11                 else
12                     return -1; /* errno set by read() */
13             }
14             else if (nread == 0)
15                 break; /* EOF */
16             nleft -= nread;
17             ptr += nread;
18         }
19         return (count - nleft); /* return >= 0 */
20 }
```

EINTR 错误的产生：当阻塞于某个慢系统调用的一个进程捕获某个信号且相应信号处理函数返回时，该系统调用可能返回一个 EINTR 错误

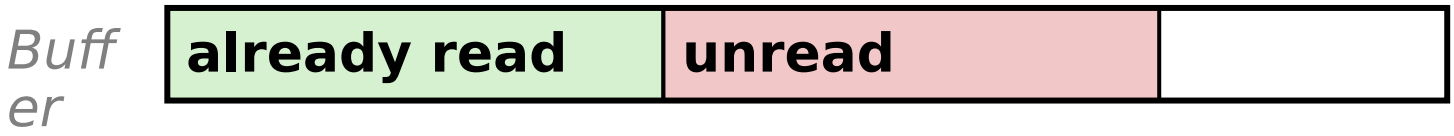
```
1  ssize_t rio_writen(int fd, const void *buf, size_t count)
2  {
3          size_t nleft = count;
4          ssize_t nwritten;
5          const char *ptr = buf;
6
7          while (nleft > 0) {
8              if ((nwritten = write(fd, ptr, nleft)) <= 0) {
9                  if (errno == EINTR)
10                     nwritten = 0; /* and call write() again
   */
11                 else
12                     return -1; /* errorno set by write() */
13             }
14             nleft -= nwritten;
15             ptr += nwritten;
16         }
17         return count - nleft;
18 }
```

Buffered I/O: Motivation

- **Applications often read/write one character at a time**
  - `getc, putc, ungetc`
  - `gets, fgets`
    - **Read line of text on character at a time, stopping at newline**
- **Implementing as Unix I/O calls expensive**
  - `read` **and** `write` **require Unix kernel calls**
    - **> 10,000 clock cycles**

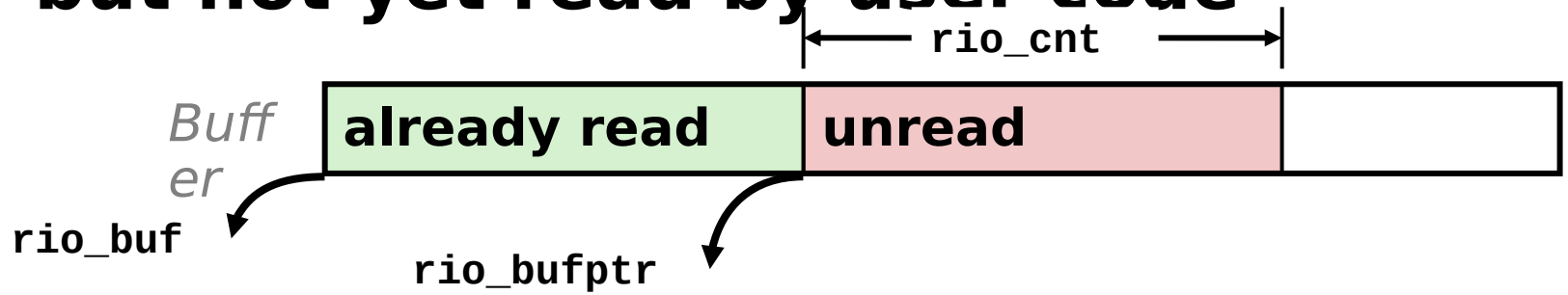
Buffered I/O: Motivation

- **Solution: Buffered read**
  - **Use Unix `read` to grab block of bytes**
  - **User input functions take one byte at a time from buffer**
    - **Refill buffer when empty**



Buffered I/O: Implementation

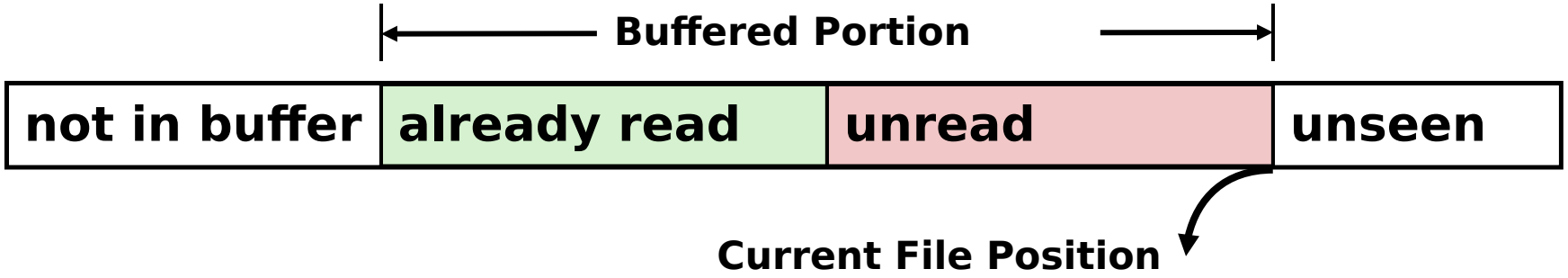
- **For reading from file**
- **File has associated buffer to hold bytes that have been read from file but not yet read by user code**




```
typedef struct {
    int rio_fd;                /* descriptor for this internal buf */
    int rio_cnt;               /* unread bytes in internal buf */
    char *rio_bufptr;          /* next unread byte in internal buf */
    char rio_buf[RIO_BUFSIZE]; /* internal buffer */
} rio_t;
```

Buffered I/O: Implementation

- **Layered on Unix file:**



Robust I/O

- **Buffered Input and Output**
  - **Efficiently read text lines and binary data from a file whose contents are cached in an application-level buffer**

```
#include "csapp.h"
void rio_readinitb(rio_t *rp, int fd) ;
ssize_t rio_readlineb(rio_t *rp,
                      void *usrbuf, size t maxlen);
ssize_t rio_readnb(rio_t *rp,
                      void *usrbuf, size t maxlen);
  returns: number of bytes read (0 if EOF), -1 on error
```

Robust I/O

```
#define RIO_BUFSIZE 8192
typedef struct {
   int rio_fd;
   int rio_cnt;
   char *rio_bufptr;
   char rio_buf[RIO_BUFSIZE];
} rio_t ;


void rio_readinitb(rio_t *rp, int fd)
{
   rp->rio_fd = fd ;
   rp->rio_cnt = 0 ;
   rp->rio_bufptr = rp->rio_buf ;
}
```

Robust I/O

```
#include “csapp.h”

int main(int argc, char **argv)
{
   int n;
   rio_t rio;
   char buf[MAXLINE];

   rio_readinitb(&rio, STDIN_FILENO);
   while ((n = rio_readlineb(&rio, buf, MAXLINE))!=0)
       rio_writen(STDOUT_FILENO, buf, n);
}
```

```
1  static ssize_t rio_read(rio_t *rp, char *usrbuf, size_t n)
2  {
3    int cnt = 0;
4
5    while (rp->rio_cnt <= 0) { /* refill if buf is empty */
•      rp->rio_cnt = read(rp->rio_fd, rp->rio_buf,
•                                 sizeof(rp->rio_buf));
8      if (rp->rio_cnt < 0) {
9          if (errno != EINTR)
10             return –1 ;
11     }
12     else if (rp->rio_cnt  ==  0) /* EOF */
13         return 0;
14     else
15         rp->rio_bufptr = rp->rio_buf;/* reset buffer ptr */
16   }
17
```

```
18    /* Copy min(n, rp->rio_cnt) bytes
          from internal buf to user buf */
19    cnt = n ;
20    if ( rp->rio_cnt < n)
21        cnt = rp->rio_cnt ;
22    memcpy(usrbuf, rp->rio_bufptr, cnt) ;
23    rp->rio_buffer += cnt ;
24    rp->rio_cnt -= cnt ;
25    return cnt ;
26 }
```

```
1  ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n)
2  {
3      size_t nleft = n;   ssize_t nread ;
4      char *bufp = usrbuf;
5      while (nleft > 0) {
6          if ((nread = rio_read(rp, bufp, nleft)) < 0) {
7              if ( errno = EINTR)
8                  /* interrupted by sig handler return */
9                  nread = 0;
10             else
11                 return –1;
12         }
13         else if (nread == 0)
14             break;
15         nleft -= nread;
16         bufp += nread;
17     }
18     return (n – nleft);
19 }
```

```
1  ssize_t rio_readlineb (rio_t *rp,
                 void *usrbuf, size_t maxlen)
2  {
3       int n, rc;
4       char c, *bufp = usrbuf;
5       for (n=1; n < maxlen; n++) {
6           if ((rc = rio_read(rp, &c, 1)) == 1) {
7               *bufp++ = c;
8               if (c == '\n')
9                   break;
10          } else if (rc == 0) {
11              if (n== 1)
12                  return 0; /* EOF, no data read */
13              else
14                  break;
15          } else
16              return –1; /* error */
17      }
18      *bufp = 0 ;
19      return n ;
20 }
```

Outline

- I/O Devices
- Unix I/O
- Reading File Metadata
- Sharing Files & I/O redirection
- Robust I/O
- Standard I/O
- Fast I/O

Standard I/O

- **The C standard library (`libc.so`) contains a collection of** higher-level standard I/O **functions**

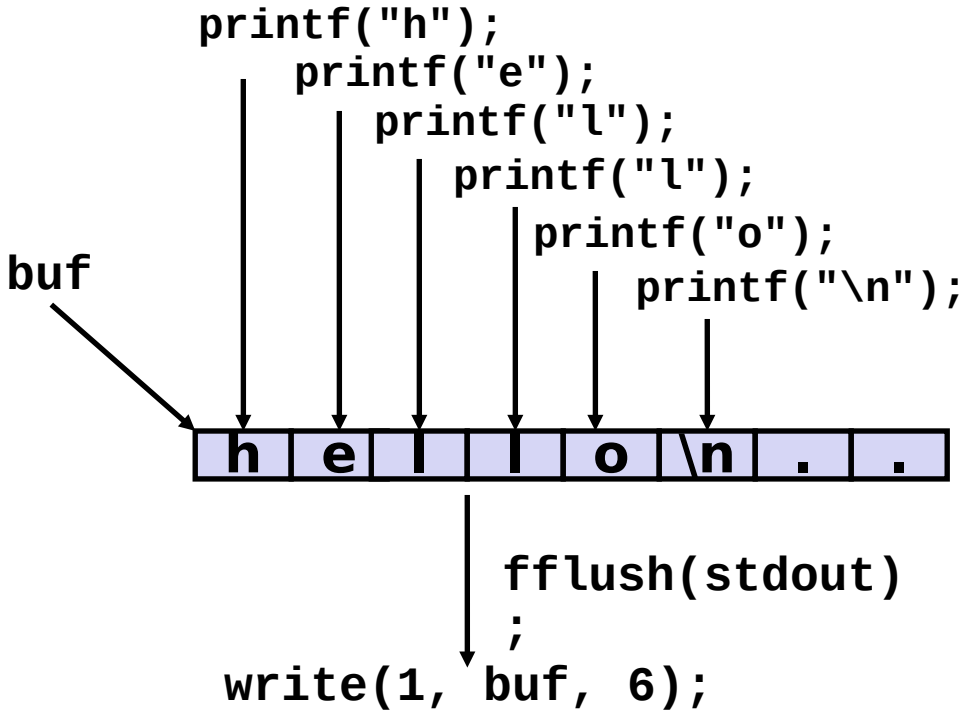
- **Examples of standard I/O functions:**
  - **Opening and closing files (`fopen` and `fclose`)**
  - **Reading and writing bytes (`fread` and `fwrite`)**
  - **Reading and writing text lines (`fgets` and `fputs`)**
  - **Formatted reading and writing (`fscanf` and**

Standard I/O

- **Standard I/O models open files as streams**
  - **Abstraction for a file descriptor and a buffer in memory.**
  - **Similar to buffered RIO**
- **C programs begin life with three open streams (defined in `stdio.h`)**
  - **`stdin` (standard input `fd=0`)**
  - **`stdout` (standard output `fd=1`)**
  - **`stderr` (standard error `fd=2`)**

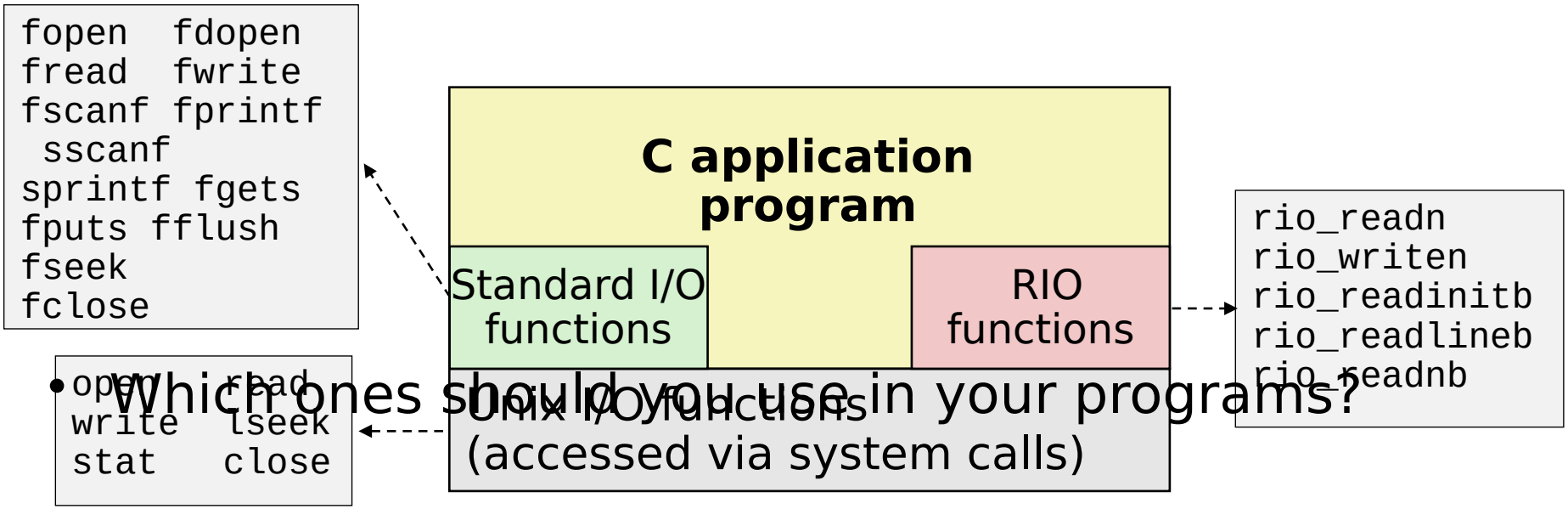
Buffering in Standard I/O

- **Standard I/O functions use buffered I/O**



- **Buffer flushed to output fd on “\n” or `fflush()` call**

- Standard I/O and RIO are implemented using low-level Unix I/O



- Which ones should you use in your programs?

Pros and Cons of Unix I/O

- **Pros**
  - **Unix I/O is the most general and lowest overhead form of I/O.**
    - **All other I/O packages are implemented using Unix I/O functions.**
  - **Unix I/O provides functions for accessing file metadata. (Standard I/O 没有 stat 对应接口)**
  - **Unix I/O functions are async-signal-safe and can be used safely in signal handlers.**

Pros and Cons of Unix I/O

- **Cons**
  - **Dealing with short counts is tricky and error prone.**
  - **Efficient reading of text lines requires some form of buffering, also tricky and error prone.**
  - **Both of these issues are addressed by the standard I/O and RIO packages.**

Pros and Cons of Standard I/O

- **Pros:**
  - **Buffering increases efficiency by decreasing the number of `read` and `write` system calls**
  - **Short counts are handled automatically**

Pros and Cons of Standard I/O

- **Cons:**
  - **Provides no function for accessing file metadata**
  - **Standard I/O functions are not async-signal-safe, and not appropriate for signal handlers.**
  - **Standard I/O is not appropriate for input and output on network sockets**
    - **There are poorly documented restrictions on streams that interact badly with restrictions on sockets**

选择使用哪种 I/O 函数?

- 用户尽可能使用 Standard I/O
  - 有 buffer，减少 syscall 数量
  - 专业软件可能还是用 Unix I/O，自己写 buffer
- 不要使用 scanf 或 rio_readlineb 来读取二进制文件
  - 它们是用处理文本文件的
  - 二进制中的 0xA 会被误解
- 对网络套接字的 I/O 使用 RIO 函数
  - 网络也可以看作文件
  - 网络传输不稳定性更强，不一定能够访问到预定的大小
  - 网络套接字只能连续访问，不能使用 lseek

Restrictions on Streams

- **Stream 是全双工的**
  - 同一个流，可以输入，可以输出
- **跟在输出函数之后的输入函数**
  - 如果中间没有插入 **fflush, fseek, fsetpos, rewind**（文件指针指向开始位置）函数，输入函数不能跟在输出函数之后
    - **The fflush function empties the buffer associated with a stream.**
    - **The latter three functions use the Unix I/O lseek function to reset the current file position.**

Restrictions on Streams

- 跟在输入函数之后的输出函数
  - 如果中间没有插入对 **fseek, fsetpos, rewind** 的调用，输出函数不能跟在输入函数后面
    - 写入操作可能覆盖已有内容
    - 除非输入函数已经到了 **end-of-file**

Restrictions on Streams

- **The only way to work around the second restriction is to**
  - **open two streams on the same open socket descriptor**

```
FILE *fpin, *fpout;
fpin = fdopen(sockfd, "r");
fpout = fdopen(sockfd, "w");

fclose(fpin);
fclose(fpout);
```

Choosing I/O Functions

- **General rule: use the highest-level I/O functions you can**
  - **Many C programmers are able to do all of their work using the standard I/O functions**

Choosing I/O Functions

- **When to use standard I/O**
  - **When working with disk or terminal files**
- **When to use raw Unix I/O**
  - **Inside signal handlers, because Unix I/O is async-signal-safe.**
  - **In rare cases when you need absolute highest performance.**
- **When to use RIO**
  - **When you are reading and writing network sockets.**
  - **Avoid using standard I/O on sockets.**

Outline

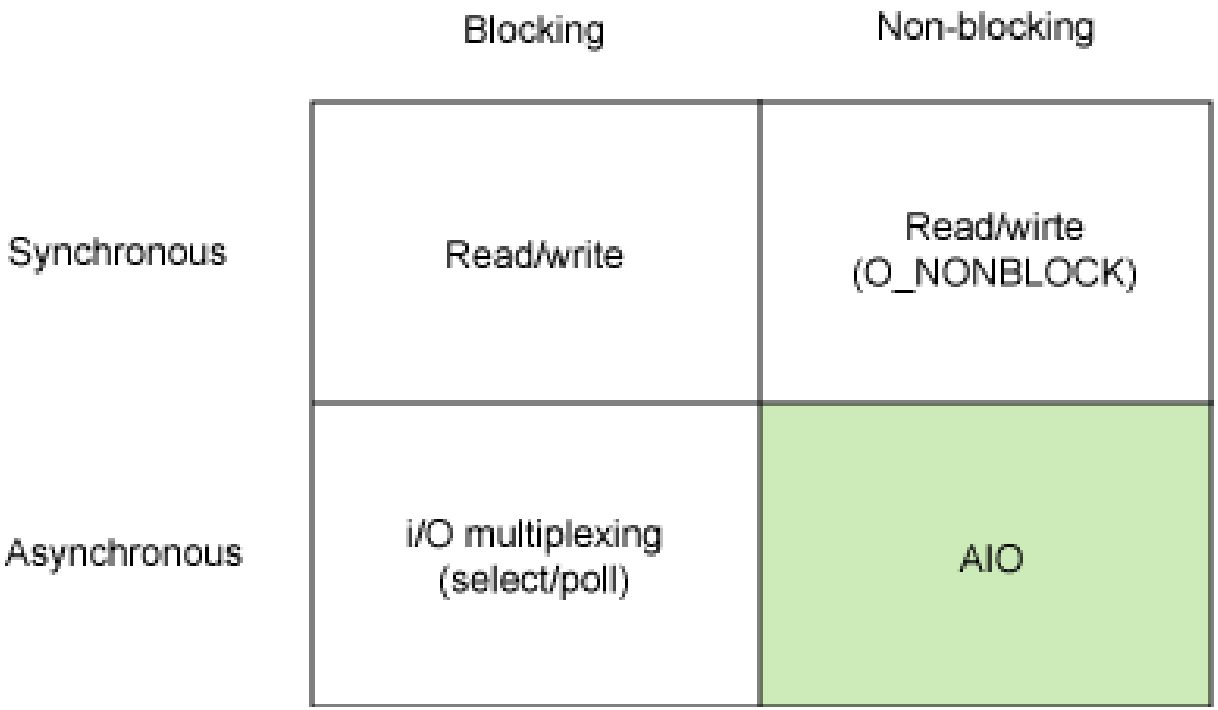
- I/O Devices
- Unix I/O
- Reading File Metadata
- Sharing Files & I/O redirection
- Robust I/O
- Standard I/O
- Fast I/O
  - Linux AIO
  - SPDK

Linux AIO

- Linux 2.6 内核开始
- 异步 IO （ AIO ）的基本思想
  - 允许进程发起很多 I/O 操作，而不用阻塞或等待任何操作完成
  - 稍后或在接收到 I/O 操作完成的通知时，进程就可以检索 I/O 操作的结果

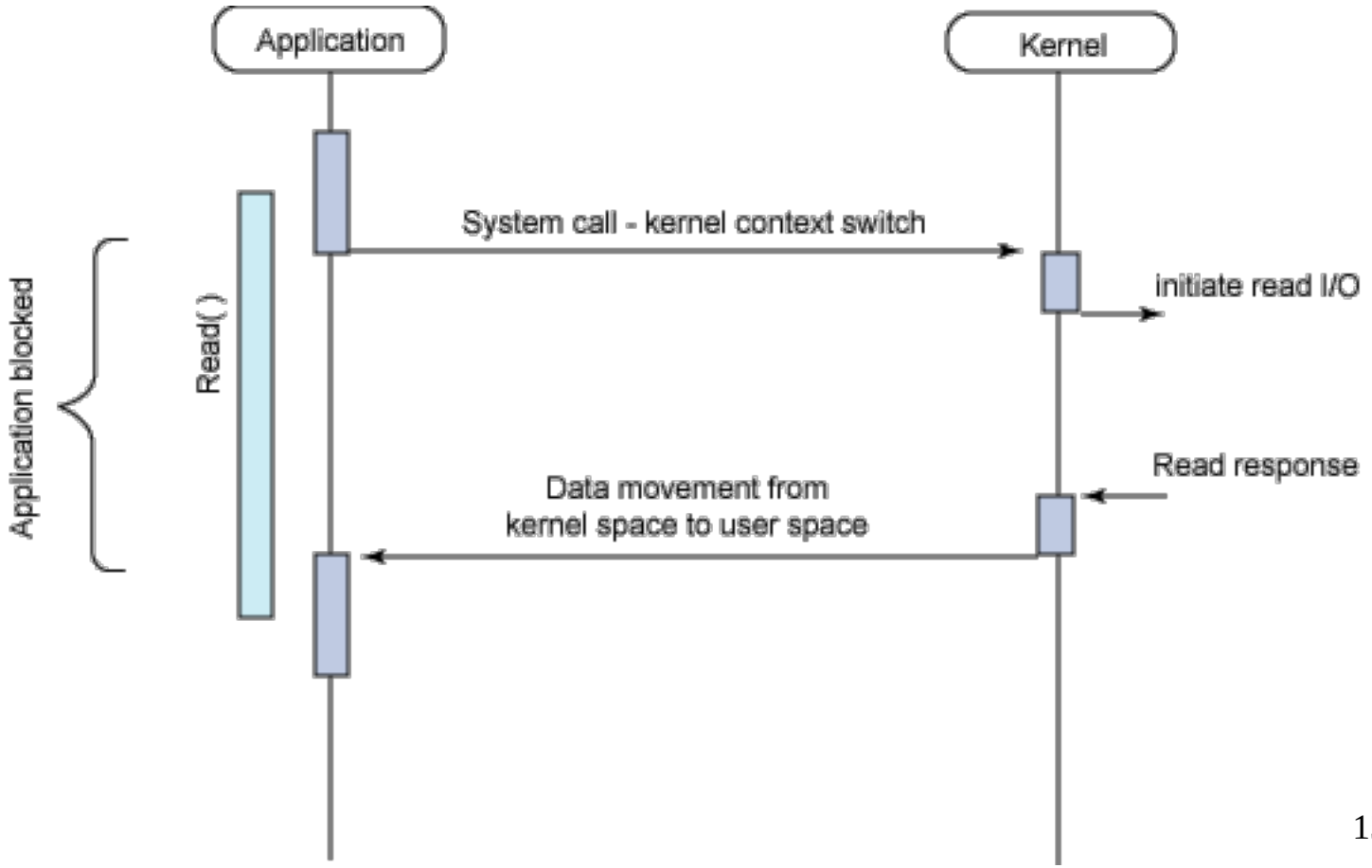
Linux AIO

- 基本 Linux I/O 模型的简单矩阵



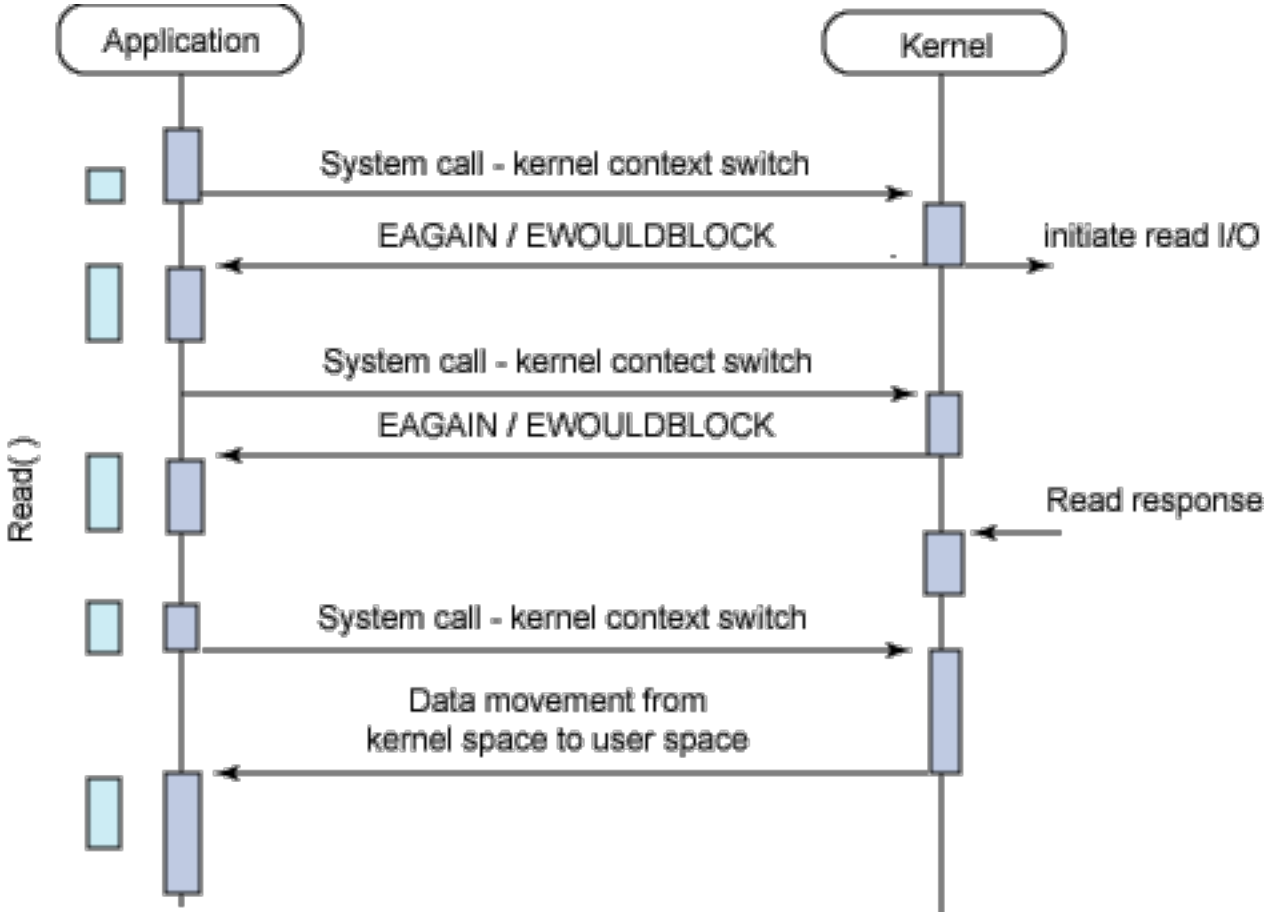
Linux AIO

- 同步阻塞模型



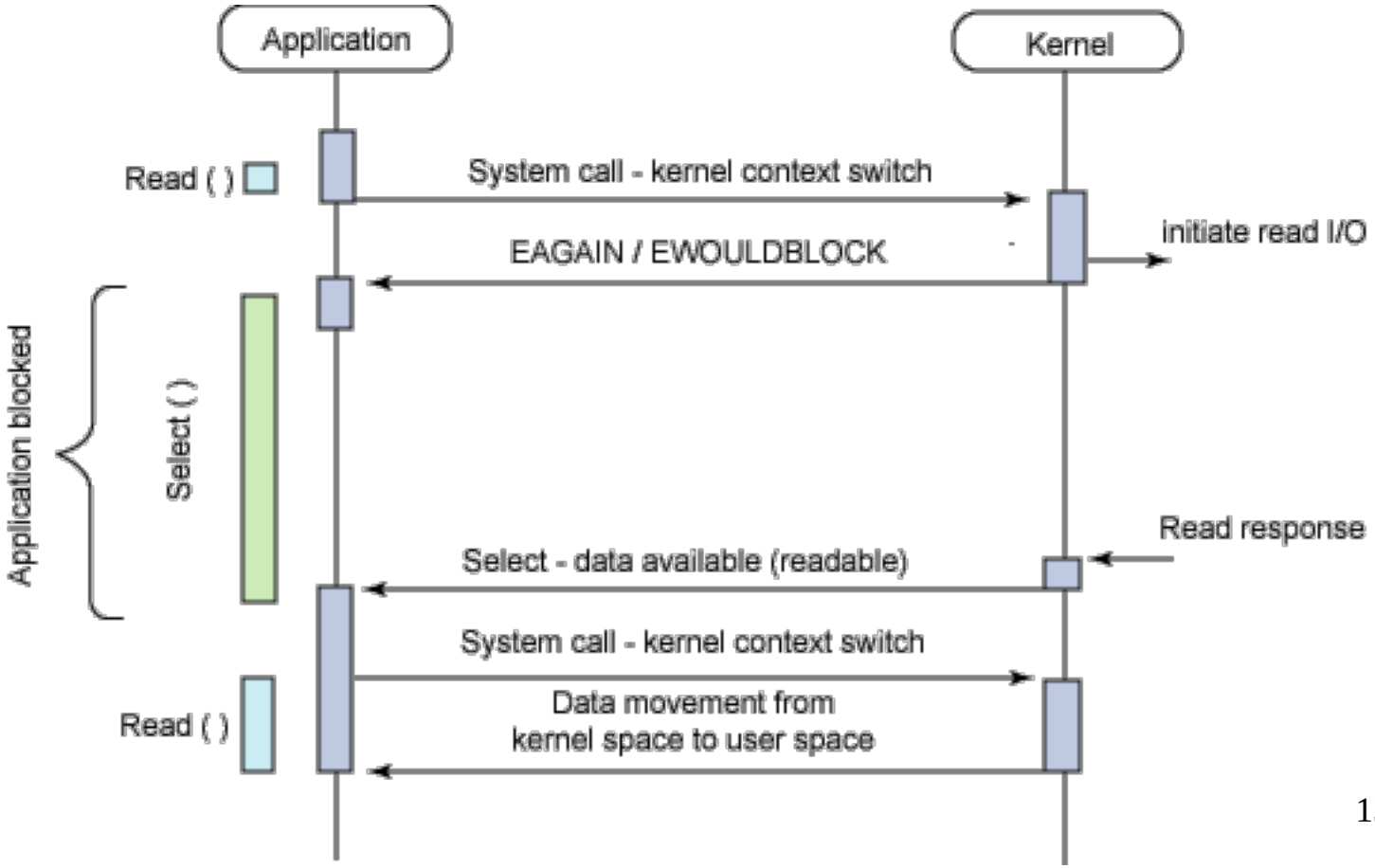
Linux AIO

- 同步非阻塞模型



Linux AIO

- 异步阻塞模型



Linux AIO

- **Select 简介**
  - **I/O 端口的复用**
  - **传递给 select 函数的参数会告诉内核:**
    - **我们所关心的文件描述符**
    - **对每个描述符，我们所关心的状态。（我们是要想从一个文件描述符中读或者写，还是关注一个描述符中是否出现异常）**
    - **我们要等待多长时间。（我们可以等待无限长的时间，等待固定的一段时间，或者根本就不等待）**
- **#include <sys/select.h>**
- **int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,struct timeval *timeout);**

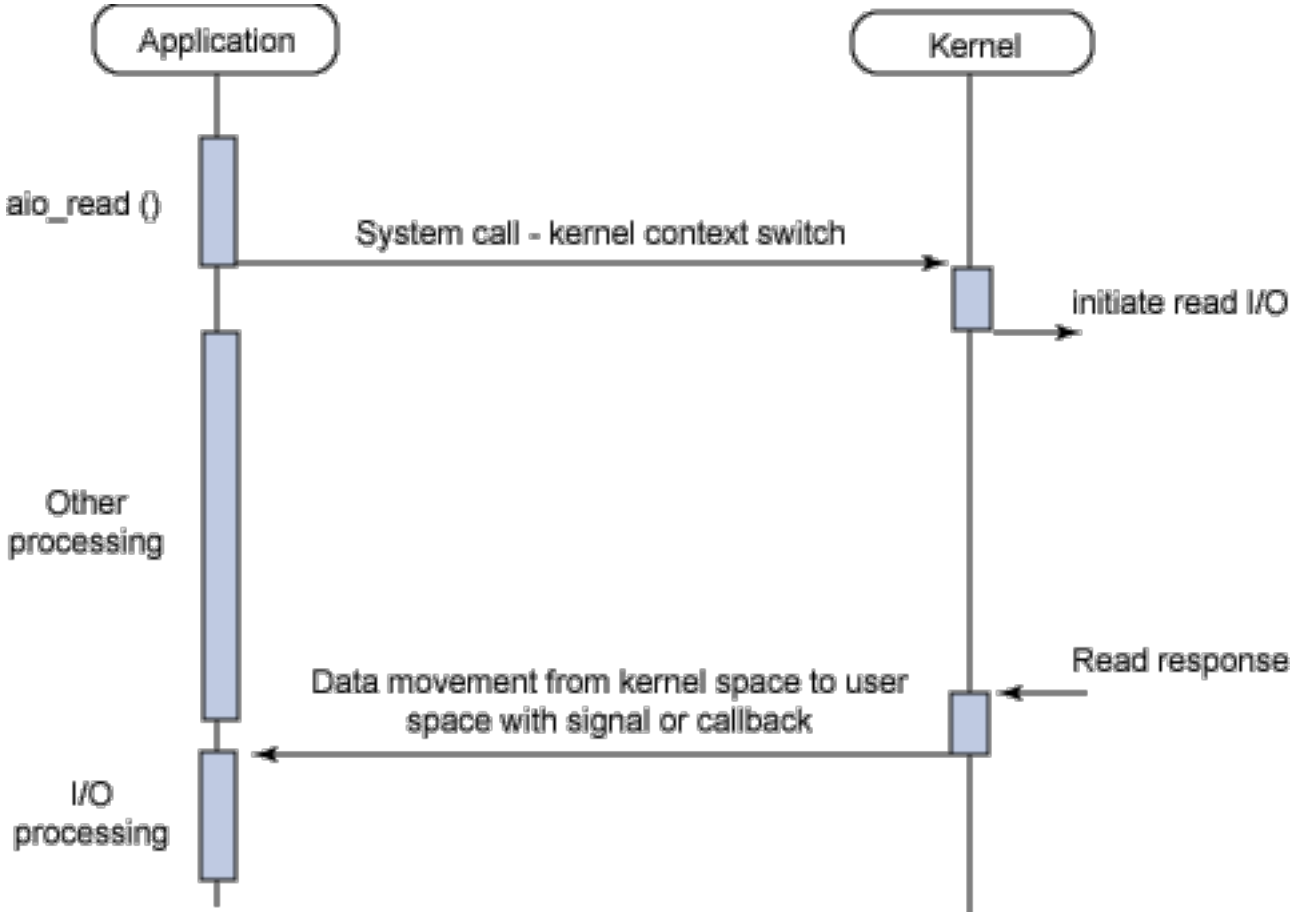
Linux AIO

- **Select 简介**
  - **I/O 端口的复用**
  - **从 select 函数返回后，内核告诉我们一下信息：**
    - **对我们的要求已经做好准备的描述符的个数**
    - **对于三种条件哪些描述符已经做好准备 .(读，写，异常)**
  - **有了这些返回信息，我们可以调用合适的 I/O 函数 (通常是 read 或 write)，并且这些函数不会再阻塞**

- **#include <sys/select.h>**
- **int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,struct timeval *timeout);**

Linux AIO

- 异步非阻塞模型



Linux AIO

API 函数	说明
aio_read	请求异步读操作
aio_error	检查异步请求的状态
aio_return	获得完成的异步请求的返回状态
aio_write	请求异步写操作
aio_suspend	挂起调用进程，直到一个或多个异步请求已经完成（或失败）
aio_cancel	取消异步 I/O 请求
lio_listio	发起一系列 I/O 操作

Linux AIO

- 核心结构体

```
1  struct aiocb {
2
3    int aio_fildes;               // File Descriptor
4    int aio_lio_opcode;           // Valid only for lio_listio (r/w/nop)
5    volatile void *aio_buf;       // Data Buffer
6    size_t aio_nbytes;            // Number of Bytes in Data Buffer
7    struct sigevent aio_sigevent; // Notification Structure
8
9    /* Internal fields */
10   ...
11
12 };
```

Linux AIO

- **aio_read**
  - **aio_read 函数在请求进行排队之后会立即返回。如果执行成功，返回值就为 0；如果出现错误，返回值就为 -1，并设置 errno 的值**

- **int aio_read(struct aiocb *aiocbp);**

Linux AIC

```c
#include <aio.h>

...

  int fd, ret;
  struct aiocb my_aiocb;

  fd = open( "file.txt", O_RDONLY );
  if (fd < 0) perror("open");

  /* Zero out the aiocb structure (recommended) */
  bzero( (char *)&my_aiocb, sizeof(struct aiocb) );

  /* Allocate a data buffer for the aiocb request */
  my_aiocb.aio_buf = malloc(BUFSIZE+1);
  if (!my_aiocb.aio_buf) perror("malloc");

  /* Initialize the necessary fields in the aiocb */
  my_aiocb.aio_fildes = fd;
  my_aiocb.aio_nbytes = BUFSIZE;
  my_aiocb.aio_offset = 0;

  ret = aio_read( &my_aiocb );
  if (ret < 0) perror("aio_read");

  while ( aio_error( &my_aiocb ) == EINPROGRESS ) ;

  if ((ret = aio_return( &my_iocb )) > 0) {
    /* got ret bytes on the read */
  } else {
    /* read failed, consult errno */
  }
```

也可以通过信号机制进行通知和后处理

SPDK

- **SPDK (Storage Performance Development Kit)**
- **主要特性:**
  - **将驱动器移动到用户态，尽可能避免系统调用以及实现零拷贝**
  - **使用轮询硬件的方式取代中断，降低了平均延迟以及尾延迟**
  - **依赖消息传递，避免 I/O 路径的锁定**

SPDK

- 用户态控制硬件的实现
  - 传统驱动程序工作在内核态
    - 操作系统根据权限分离虚拟内存
    - 由 CPU 硬件支持
  - SPDK 实现驱动器在用户态
    - 1） linux 系统下，通过修改 sysfs 文件指示操作系统放弃对设备的控制
    - 2） SPDK 将驱动程序绑定到 uio 或 vfio
    - 3） SPDK 通过 uio 和 vfio 将设备的 PCI BAR 映射到当下进程，允许程序直接执行 MMIO

SPDK

- 驱动程序用户态技术
  - UIO
    - UserSpace I/O
    - 允许驱动程序的用户态实现
    - 不支持 DMA 、中断
  - IOMMU
    - input/output memory management unit
    - 设备独立页表，不同直通设备互相隔离
  - VFIO
    - Virtual Function I / O
    - 场景： PCI 互联， IOMMU 不满足此场景
    - VFIO 兼顾 UIO 和 IOMMU

SPDK

- SPDK 采用轮询方式而非中断方式
  - 机械磁盘主要时耗
    - 磁盘物理 IO
  - NVMe & SSD
    - 低延迟设备
    - 依赖中断方式的传统 IO 栈成为瓶颈
  - 在用户空间将中断路由到处理程序
    - 对于大部分硬件不可行
  - 中断所导致强制的上下文切换
    - 造成尾延迟
  - SPDK 允许 IO 完成时提供回调， SPDK 采用轮询方式响应回调

SPDK

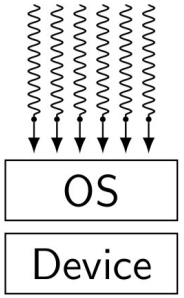
- 消息传递避免 I/O 路径的锁定
  - SPDK 主要目标
    - 通过增加硬件让性能线性拓展，例如
      - 从一个 SSD 到两个 SSD 应该是每秒 I/O 数量的两倍
      - CPU 核心数量加倍应该可以使计算量增加一倍
  - 传统上，线程通过锁并发访问共享数据
    - 单线程程序转换为多线程程序简单（资源加锁）
    - 高效利用 CPU 资源
    - 缺点：线程数增加造成资源争用
  - SPDK 做法
    - 数据分配给不同的线程，而非将共享数据放到全局位置由所有线程获得锁后访问
    - 单个 CPU core 始终访问相同的数据
      - CPU core cache 命中率更高
    - 保证了线程可以提交请求而不必与其他线程协的开销

SPDK

- SPDK 使用方式
  - Blobstore
    - 持久化、断电非易失的块分配器
    - 代替传统文件系统提供存储服务
    - 不支持 POSIX 标准
    - 允许对块设备上的一系列块进行异步、 uncached 、并行读写
  - Block Device Layer
    - 使用 bdev 访问块设备

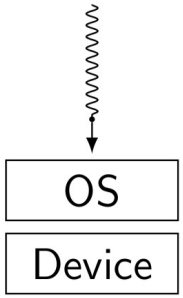
总结

Synchronous IO



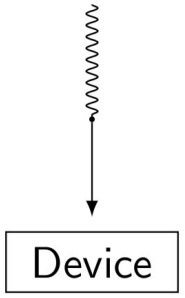
- one thread per request
- syncronous (blocking) syscalls (e.g., `pread`)

Asynchronous IO



- issue IO requests
- receive IO completions
- e.g., Linux AIO

User-space IO



- Directly access the device
- Polling instead of interrupts
- e.g., SPDK