

什么是操作系统：

- 1、计算机核心程序 (kernel program)
- 2、用户接口程序 (interface program)
- 3、资源分配与回收的控制程序 (control program)

操作系统是位于 Hardware 与 User 之间的**程序集合**

基本目标：方便性与高效性

工作原理：中断驱动

存储结构：内存，二级存储，分成结构

硬件保护：双模式（系统保护），特权指令，内存，IO，CPU（硬件保护）

服务：程序，IO，文件，进程，通信

硬件 (Hardware)：

启动：执行引导程序，载入 operation program kernel (ROM/EPROM 中)

中断：告知操作系统执行中断处理程序/策略 (interrupt/trap)

Interrupt driven：中断-挂起-中断处理-中断返回

通用中断功能：Interrupt service routine (由**中断向量**保存)、save address (程序恢复)、disable interrupt (进程同步)

Memory：寄存器-main memory (内存)-electronic disk (U 盘)-magnetic disk (磁盘)-optical disk

内存：CPU 可以**直接访问**的唯一的**大型存储介质**

IO 设备：控制器 **local buffer** 和**外设交互**的过程 (I/O 过程无需 CPU 干预，中断响应方式)

系统保护：Dual-Mode (用户，内核态)、CPU (进程时间控制)、内存 (程序地址控制，有效-无效位，页表长度)、IO 保护 (全 privilege)

Privileged Instruction 特权指令：可能对硬件造成伤害的指令，在内核态执行

发展 (Develop)：

单任务系统、批处理系统 (批处理文件)、多道程序系统 (提高 CPU 利用率，发挥程序**并发性**，内存，CPU，任务，IO 资源控制决策)、分时系统 (多用户共享，CPU 调度，virtual memory)

并行：多 CPU 同时进行 并发：单 CPU 资源调配 (时间段)

操作系统特征：**并发** (Concurrence)，**共享** (Sharing)，**虚拟** (Virtual)，**异步** (Asynchronous)

CPU 利用率：CPU **非空闲时间**占比

Desktop System：单处理器 (Single-Processor)，多处理器 (Multiprocessor)，紧耦合 (Tightly coupled)，分布式 (network 云端，shared storage 集群系统 (hot-standby 非对称，对称))，实时 (时间约束)，手势终端 (Pocket-PC，Cellular telephone，资源少) 系统

优点：高吞吐量 (Throughput)，资源共享，可靠性高

功能 (Functions)：

系统内部具有的能力

进程 (运行中的程序)，内存，磁盘 (Secondary-Storage)，文件 (文件系统)，IO 管理，用户接口

服务 (Services):

对外可以提供的服务

程序执行, I/O 操作, 文件系统操作 (File-system manipulation), 进程通信, 错误监控, 系统执行 (资源分配, 统计, 保护)

接口 (interface):

命令行 (Command-Line), 批处理 (Batch 配置命令文件 .bat), 图形用户界面 (Graphical User)

Command-line Shell 命令行解释器: 解读命令

System Calls Interface **系统调用** (基本功能实现): 系统调用就是一种特殊的接口, 用户可以访问内核的具体位置。参数放置 (寄存器, 内存, 栈),

Application Programming Interface (API): 应用程序接口, 是一些预定义的函数。跟内核没有必然的联系, 调用 OS 层功能, 提供应用程序与开发人员基于某软件或硬件的以访问一组例程的能力, 而又无需访问源码, 或理解内部工作机制的细节。(run-time support library 接口使用说明)

区别: API 是函数的定义, 规定了这个函数的功能, 跟内核无直接关系。而系统调用是通过中断向内核发请求, 实现内核提供的某些服务。

联系: 一个 API 可能会需要一个或多个系统调用来完成特定功能。通俗点说就是如果这个 API 需要跟内核打交道就需要系统调用, 否则不需要。

结构 (Structure):

简单 (MS-DOS, UNIX 结构区分不明显), **分层** (分为多层 hardware...功能模块化...user layer), **微内核** (Microkernels 仅保留进程, 内存管理和通信功能 易于扩展和移植, user 和 kernel 转换产生额外开销), **模块化** (面向对象的方法, 灵活性高)

Virtual machine: 模拟硬件, 为操作系统提供 illusion hardware interface, 虚拟机间资源不共享 (JVM: 解释 Java, 可移植性很高)

设计 (Design):

Policy (策略 做什么), Mechanism (机制 怎么做)

C, C++, 汇编

进程 Process:

概念: 按照顺序结构执行中的程序, 程序决定进程执行顺序 (非一一对应)

组成: 程序代码 (Text section), 程序计数器 (Program Counter), 指令寄存器 (执行内容 Processor Register), 栈 (堆: 程序申请的空间 栈: 操作系统分配 Heap-stack), 数据段 (Data section)

动态性 (Dynamic), **独立性** (Independency), **并发性** (Concurrency), **结构化** (Structure)

进程状态: New (初始) - Ready (准备, 获得内存资源) - Running (获得 CPU 使用权) - Waiting (等待 I/O 响应) - Terminated (exit 终止)

CPU 核心/用户态: 执行 kernel/user process

Process Control Block (PCB 进程控制块): 链表方式连接进程, 控制进程状态转换 (program counter, pointer) (管理进程)

Job queues 所有进程队列: Ready queue **就绪** (等待执行), Device queue **设备** (等待

IO 操作), 其他队列 (均在 Main memory 作业队列中)

Scheduler 调度器: Long-term (作业队列 程序内存加载选择, 控制多道程序度 (内存中存放进程个数)), Short-term (就绪队列 CPU 执行进程选择), Medium-term (进程镜像保存, 交换 swapping 腾出内存空间)

IO 进程 (以 IO 操作为主), CPU 进程 (以 CPU 计算为主) 通过调度器控制

Context switch 上下文切换: 保存 PCB, CPU, 内存等信息值, 用于区分进程状态 (在进程切换时保存 context 状态)

进程创建: 父进程创建子进程形成进程树, 并由 Process ID 标识每个进程

子进程 (PID 标识子进程 (PID<0 创建失败), 子进程执行 PID=0 的分支):

fork(): 子进程复制父进程代码与状态, 包含所有子线程 (不保留数据段, 在父进程位置继续执行)

exec(): 父进程执行 exec(), 覆盖子进程和其子线程代码段执行新功能

子进程通过 getpid() 获得自己的 ID 号

进程终止: 结束, 挂起, 随父进程终止 (级联终止)

Inter-Process Communicate 进程通信: 共享内存 (临界区问题, 需互斥访问), 消息传递 (直接/间接 producer consumer)

Bounded-Buffer: 通过循环队列控制进程的内存数据读写 (可用空间 BUFFER_SIZE-1)

Message-passing: 通信连接, 可供不同主机间通信 (物理介质+逻辑链接)

Indirect communication 间接通信: 进程通过 mailbox 间接通信, 仅当进程共享 mailbox 时才能建立链接

进程通信: Block 同步传输 (Synchronization), Non-block 异步传输 (asynchronous)

Buffering: zero (零容量, 等待应答), bound (有限容量, 滑动窗口), unbound (无限容量, Post)

CS 通信方式: Socket (套接字, 应用层访问网络层接口, IP& port, 无结构字节流), Remote Procedure Calls (通过 Stubs 调用远程进程和函数, 函数 ID 与参数, 高度结构化), Remote Method Invocation (远程方法调用, 调用远程对象和方法, stub(C)-skeleton(S))

线程 Thread:

进程中的**控制流**, CPU 执行的最小单位, 多个线程**共享进程地址空间**

线程: 线程 ID, 程序计数器, 寄存器组, 堆栈 (不共享), 属于同一进程的线程共享 code, data (全局变量), OS recourse (响应性, 资源共享, 创建和删除开销低, 多处理器架构利用率提升)

User thread 用户线程: 由**用户线程库**管理, 内核不知道线程存在 (无需状态切换, 可能出现线程阻塞导致进程阻塞), 竞争进程资源

Kernel thread 内核线程: 由内核管理, 不会导致线程阻塞, 竞争系统资源

Multithreading models 多线程模型: Many-One (同时只能有一个线程代表进程访问同一个内核), One-One (线程同时访问内核 并发性, 费资源), Many-Many (多个用户线程对应多个内核线程), Two level 模型 (**多对多, 一对一处理紧急线程**)

Thread states 线程状态: join (等待子线程先执行), cancel (取消线程 **EN/DIS 立即/延迟** 设置取消点), kill (杀死)

API 线程创建: 属性, 大小, 线程函数, 是否立即执行 (程序中最后的线程退出, 程序退出)

Suspend and Resuming: 通过栈挂起/恢复线程

并发访问: 读写冲突 (本地存储, 临界区 (对共享变量操作的代码段) 控制)

Java 线程: extend 线程类或采用 runnable 接口 (JVM 控制)

Signal: 由特定事件产生, 通知进程进行挂起 (**无优先级** 进程可选择忽略)

Thread Pools 线程池: 事先创建**内核线程**并放在线程池中, 进程需要使用时分配 (多对多 分配并新建线程效率高 对**线程总数可控**)

Upcalls: 内核对线程表的提醒

信号与中断的相似点:

- (1) 采用了相同的异步通信方式;
- (2) 当检测出有信号或中断请求时, 都暂停正在执行的程序而转去执行相应的处理程序;
- (3) 都在处理完毕后返回到原来的断点;
- (4) 对信号或中断都可进行屏蔽。

信号与中断的区别:

- (1) 中断有优先级, 而信号没有优先级, 所有的信号都是平等的;
- (2) 信号处理程序是在用户态下运行的, 而中断处理程序是在核心态下运行;
- (3) 中断响应是及时的, 而信号响应通常都有较大的时间延迟。

CPU 调度 Scheduling

CPU 调度: **CPU 空闲时**从就绪队列中选择进程, 取决于 CPU, IO Burst 代码 (进行 CPU, IO Burst cycle 替换)

CPU/IO 操作交替执行: CPU (进程少, 周期长) / IO (进程多, 周期短) bound 进程

调度: Preemptive 抢占 (控制策略) / Non-Preemptive 非抢占调度 (进程可执行完)

Scheduler 调度器: 进程选择, Dispatcher 分配器: CPU 控制权调度

策略评价: CPU Utilization (CPU 非空时间占比), Throughput (吞吐量 单位时间执行的进程数) (系统), Turnaround (周转时间 提交-任务完成的时间 (包含执行时间)), Waiting time (就绪队列中等待时间), Response time (用户请求-第一个响应) (用户)

调度策略 Scheduling:

First Come First Served **先到先服务**: 按照就绪队列顺序依次执行 (**取队头** 非抢占式 convey effect 护航效应 CPU 占用 短进程与分时系统不适用)

Gantt Chart: 用队列表示调度过程 (进程等待时间等于其前队列中进程的所有执行时间之和)

Shortest Job First **最短最优先**: 按照剩余 CPU 数 (burst length) 进行排序等待时间: 终止-burst-arrival 周转时间: 终止-arrival (理想参考模型)

Priority **优先级**: 选择最高优先级执行, 可按等待时间长短更改进程的优先级 (抢占 进程饥饿 (采用 Aging 老化方法解决) / 非抢占)

Round Robin **轮询**: **响应时间快, 适用于分时系统**, 设置进程最长时间片, 进程执行完自动释放 CPU (**时间片未必均分**, 应使进程在时间片整数倍里完成 **大多进程能在一个时间片中完成**)

Multilevel Queue 多级队列: 根据进程需求而分别采用不同调度策略, 队列间按照优先级 (feedback **多级队列反馈**调度 (老化)) / 时间片 (轮询, 不同队列设置不同时间片) 调度 (队列调度策略, 进程队列间移动 提高吞吐量, 防止 CPU 占用)

Multiple Processor 多处理器调度: 负载均衡 (push/pull),

非平衡 (系统数据仅运行一个处理器访问, 降低数据共享需求), SMP (避免处理器选择同一进程)

软亲和 (防止进程迁移策略), **硬亲和** (不允许执行中进程迁移)

Real Time 实时调度: **资源预留** (resource reservations) 确保进程按时完成 (磁盘与虚拟内存无效), 硬 (确保执行完成) / 软 (优先级高) 实时 (**无法使用老化**来防止饥饿)

Priority Inversion 优先级退让: 改变执行中进程优先级防止抢占, 执行完后恢复

线程调度: 进程级/系统级

策略评价方式: 决策 (人为设定), 队列 (人为模拟), 模拟 (模拟真实系统), 实现

进程同步 Synchronization:

规定进程执行顺序, 确保共享数据一致性

Race Condition 竞争条件: 多个进程同时访问同一共享数据时产生

Critical Section 临界区问题: 更改共享数据的进程代码段, 设置互斥访问 (同步的子问题) (临界资源: 共享数据)

临界区协议: entry (进入 互斥访问), critical (临界区), exit (退出 进展性) 部分

解决方法: **互斥, 进展性, 有限等待** (独立三个条件需同时满足, 证明满足 (反证))

让权等待: 进程在等待进入临界区且区中进程执行 IO 时, 不能占用 CPU 资源 (多 CPU 时可以使用自旋锁)

临界区中进程不一定运行 (可能为 IO), 等待进入临界区进程不一定处于等待状态

Perterson's Solution: turn (权限控制), flag (进入预约) (**双进程** 申请+让步)

Bakery Algorithm: 想进入线程取号, (取号, 线程号) 先比较取得号大小再比较线程号

循环直到无人取号且进程序号最小, 执行 CS(), 更改 number[i]=0;

硬件支持同步: 禁止中断 (单 CPU 适用, 影响系统功能), 特殊指令

软件支持同步:

Test And Set: 检测并更正 lock 为 true, 占用临界区 (TAS 需一次性执行, 易产生饥饿)

Swap: 本地变量 key, 全局锁 lock, 交换 key 与 lock 满足互斥访问 (易产生饥饿)

满足临界区问题解决 **TAS**: waiting[i], lock, key

进入: waiting || key != true (waiting[i]=false)

退出: 查找并使下一个进程跳出临界区 (waiting[j]=false)

Semaphore 信号量: 软件方式, 分布式同步机制, 仅由 wait/single 操作 (类型声明)

Counting: 奇数信号量, 可用资源个数 **Binary**: 0-1 互斥信号量

信号量改进: 获取资源失败, 阻塞在信号量队列, 资源空闲时取出队头执行

双 Semaphore: P: wait(Q);wait(S) Q: wait(S);wait(Q) (**相互等待, 死锁**)

Starvation 饥饿: 一个进程一直得不到使用权 **Deadlock 死锁**: 多个进程相互等待

经典同步问题: **临界资源数** (互斥信号量 mutex), **同步关系** (同步信号量 counter)

Bound-Buffer: 仅通过 wait 与 single 操作

Producer 生产者: wait(empty);wait(mutex);
single(mutex);single(full);

Consumer 消费者: wait(full);wait(mutex);
single(mutex);single(empty);

Reader-Writer: 读者 (仅读取), 写者 (可读写)

Reader first 读者优先: counter 记录读者数, 临界区有读者时新读者直接进入, 写者等待

无饥饿读写者：增加互斥信号量，读写者进入均争夺（进入后立即释放）
Dining-Philosophers 哲学家就餐问题：5 根筷子（5 互斥信号量）

dp.pickup();EAT;dp.putdown()
pickup() {设置 hungry, 调用 test 判断是否可以进食},
putdown() {设置 thinking, 调用 test 观察左右是否需要进食}
test() {测试左右是否在进食, 若不均开始进食, 否则阻塞}

独木桥问题: mutex = 1(独木桥信号量), MA = 1, MB = 1; counterA = 0, counterB = 0;

Entry: 获取更改counterA信号量(MA), 若counterA == 1获取mutex, 释放MA

Exit: 获取更改counterA信号量(MA), 若counterA == 0释放mutex, 释放MA

Monitor 管程: 高级的**同步数据结构**(construct), 将分布的临界区集中管理, 进程通过管程访问共享数据, 确保访问互斥性

Condition Variable 条件变量: 进程**进入管程**等待, 仅允许 x. wait() (**挂起 x**) 与 x. single() (**唤醒** 唤醒后等待/继续) 操作 (**不进行加减操作**)

管程 (集中管理, 解决共享资源的公用数据结构), 进程 (占有数据资源实现并发性**的私有数据结构**) (操作系统需确保进程按照管程规定调用管程操作)

死锁 Deadlocks:

模型: 资源, 进程 (请求, 使用, 释放资源)

死锁: 进程集合内进程互相等待使集合处于死锁 (两个以上)

资源分配图: V (进程, 资源点), E (请求, 分配边)

单实例资源分配图出现环路则死锁, 无环则不会死锁

死锁条件: **互斥** (mutual exclusion), **占有等待**, **非抢占**, **循环等待** (不独立)

处理死锁: 拒绝 (预防 (破坏死锁条件之一), 避免 (满足/延迟资源分配)), 发现恢复, 忽略死锁

Prevention 预防: 增加可用资源实例数 (互斥), 不允许占有 (占有等待), 抢占 (抢别人/被抢), 仅申请更高级资源 (循环等待) (**破坏死锁条件之一**)

Avoidance 避免:

资源分配图算法 (单实例): 单实例 (声明, 请求, 占有边 判断是否成环)

银行家算法 (多实例 **计算**): 假装更新 (更改Available, Allocated, Need), 寻找新的安全序列, 满足则分配

安全状态 Safe State: 系统中包含所有进程的安全序列 ($Need \leq Available$, 累加 Available)

安全一定不死锁, 非安全可能死锁

Detection 发现恢复:

等待图: 将资源节点移除, 边表示等待关系 (单实例)

死锁发现: Available, Allocation, **Request** (Work: 可用资源, Finish: true 占有为 0 (不参与判断) 得到 finish 为 false **死锁队列**) (**多实例** 进程请求资源, CPU 负载轻运行) (将未占有资源的进程拿掉, 寻找死锁队列)

死锁恢复: 按一定策略**终止/抢占**相应进程, 终止进程后回退到初始安全状态, 考虑回退次数避免饥饿 (优先级, 剩余时间片, 进程占有, 需要资源, 交互/批处理)

内存管理 Memory Management:

内存管理: Main Memory, Register, Cache

地址分类: Symbolic 符号, Relocatable 重定位 (相对), Absolute 物理地址 (绝对)

地址捆绑: 编译 Compile, 装载 Load, 执行 Execution

逻辑地址: 程序所在的虚拟地址 物理地址: 内存单元地址 (编译装载同 执行不同)

重定位寄存器 Relocation: 逻辑地址+重定位=物理地址 (便于编程与移植性)

执行时: 动态加载 (使用时加载程序), 动态链接 (程序中函数标识, 执行时导入 (API)), 交换 Swapping (虚拟内存 Baking Store, CPU 直接访问磁盘)

连续分配:

固定分区 (内存固定分为大小不同块, 放入单个进程, 独立/共用等待队列 空间浪费)

可变分区 (Hole 可用内存块, First-fit (首次适配, 首个合适), Best-fit (最佳适配, 最小可行 空间排序->首次), Worst-fit (最差适配, 最大剩余))

空间利用率: 首次=最佳>最差 时间效率: 首次>最佳=最差

Fragmentation 碎片: 未被使用且不能分配的空间 (External 外部碎片 可消除 (分配片段间过小的空闲内存), Internal 内部碎片 (已分配但未使用 不可消除))

解决外部碎片问题: Compaction 紧缩, 不连续分配

紧缩: 将已分配内存上移, 腾出可用块

不连续分配 (分页):

Paging 分页: 将物理内存分为帧 Frames, 逻辑地址分为大小相应的页 Pages (页表记录映射关系 未解决内部碎片)

页号 (p): 页的基址 (帧起始地址), 页偏移量 offset (d): 页中位置

逻辑地址: 索引 页表内容: 实际地址

页表: 索引-页表项, 内容-帧号 (起始地址)

页表项: 帧与页的映射关系 (用于表示所有页)

页表存放: 直接由寄存器存放, TLB 快表寄存器 (Associate memory 存放经常使用的页 Page-Frame, ASIDs 记录页所属的进程 (上下文切换时不用更改 TLB)), PTBR 页表地址寄存器 (存页表起始地址, 内存两次访问) (先查 TLB 快表, 未找到则通过 PTBR 找内存)

Hit ratio 命中率: $a = \frac{\text{页在TLB中}}{\text{TLB查找}}$ (在 TLB 中找到页的频率)

Effective Access Time = $\epsilon * a + 2\epsilon * (1 - a)$ (ϵ 为访存时间)

内存保护: 基址-界限寄存器, 页表中增加有效-无效位, 页表长度寄存器 (控制进程访问空间)

Shared Pages 共享页: 共享只读代码, 维护独立代码与数据

Hierarchical 分层页表: Two-level (拆分页条目 p1=子页表物理地址, p2=页表项位置, offset: 帧偏移量, 采用外部页表做为页表索引)

帧大小=页大小, 页表项储存于页表中, 程序需要帧则增加页表项

物理地址: 帧地址+页内偏移量 (用帧地址取代页号)

逻辑地址空间: 进程需要空间大小

哈希页表: 对页号哈希定位到页表项, 页表项内轮询帧号

Invert 反向页表: 一张页表覆盖所有帧, 页表项存放页与页的所属进程 (无法共享)

表项中存物理地址, 查找时层层替换直到全为物理地址

Segment 分段: 程序中的逻辑单元, 段表表示段在内存中存放位置 (动态段分配)

段内分页, 解决外部碎片

虚拟内存 Virtual Memory:

在磁盘上开辟内存缓存区，对物理内存扩充，按需调页

Copy-on-write 写时复制：在子进程修改页内数据时才进行复制

Demand paging: 页需要被引用时调入内存（按需调页，判断引用是否有效）

缺页调入：页表项中**有效-无效位**记录缺页，合法时调入（缺页中断，页表载入（有空闲帧），更新页表）

缺页率 Page Fault Rate: $\frac{\text{缺页数}}{\text{请求次数}}$

有效访问时间: $EAT = (1-p) * \text{memory access time} + p * (\text{页置换时间})$

页置换: 寻找磁盘中页位置，寻找空闲帧（空闲帧/牺牲帧置换），页表导入空闲帧同时更改帧/页表，重启进程（设置 **modify bit 脏位** 记录页是否更改，写时置换）

Reference string 页请求队列: 用于评估页置换算法（实例法）

页置换算法（计算）:

First-In-First-Out 先进先出: 先进入内存的页先被置换（

Belady Anomaly 异常: 进程帧数增加缺页次数不降低

Optimal 最优算法: 替换将来最长时间不会被使用的页（未来页请求未知，作为标准评价其它算法）

Least Recently Used 最近最久未使用: **计数器/栈** 记录上次使用时间，替换最小时间/底部页（将表项放在顶部，替换底部表项）

引用位 **Reference bit**: 记录内存未被使用的页（仅表示是否曾被使用）

附加引用位: 8 比特字节，每周期右移 可区分页使用频率

二次机会: 轮询置引用位 0，一轮之后置换引用位仍为 0 的页

Counting 计数: 记录页**访问次数**（**LFU**: 置换次数小者，**MFU** 最频繁使用: 置换次数多者）

固定分配: 相同分配，按比例分配（进程所占比例）

优先级分配: 全局置换/本地置换

Thrashing 颠簸: 进程 **swap** 时间大于运行时间（**Locality** 进程局部性空间大于拥有空间）

多道程序度提升 CPU 利用率不一定提升，**swap** 增加

解决方法:

工作集模型 Working-Set: 进程分配的物理页面（块）的集合（进程工作集 < 分配空间则出现颠簸）

缺页频率阈值 Page-Fault Frequency: 设置缺页频率最高/最低**阈值**（频率高: 新增进程帧，频率低: 增加多道程序度）

内存映射文件: 建立磁盘块索引表，访问时无需通过 I/O

Buddy: 分配空间不足翻倍，空间减半（内部碎片多）

Slab: 多个连续物理空间

预调页: 页面被引用前调入，调入准确率 α ，衡量节省开销与浪费开销

页大小因素: 内部碎片（小页），页表大小（大页，页表项少），I/O（大页，命中率大），局部性（小页，**加载未使用少**）

TLB Reach 快表命中: $TLB * \text{页大小}$

程序数据结构: 数据存储与访问顺序（例如: 矩阵赋值）

I/O **Interlock** 互锁: 文件拷贝的进程页需加锁，不能被置换

文件系统接口 *File System Interface:*

文件: 连续的逻辑地址空间, 储存于磁盘的已命名的相关联信息集合

属性: 名字, ID, 类型, 位置, 大小, 权限

文件属性存储于文件目录 Directory Structure

打开表: 记录已经打开的文件 (关闭时修改)

访问方式: 顺序访问 (Sequential 顺序读写/倒带 rewind), 直接访问 Direct

Director 目录结构: 分隔磁盘分区并做索引 (单目录, 两级目录 (以用户定位))

树形 (可分组与定位), 非循环目录结构

文件系统实现:

分层存储: 用户程序 - 逻辑文件系统-文件组织模块-基本文件系统-I/O-物理设备

文件控制块 FCB: 文件属性+磁盘 block 指针

虚拟文件系统: 屏蔽不同文件系统, 提供统一接口 (云平台)

目录实现: 线性表, 哈希页表

分配方法: 连续分配 (起始位置+长度 出现碎片, 增长困难), 基于增长 (Extend-base 设置增长区间), 链接分配 (通过指针连接文件磁盘块, 避免外部碎片 不支持随机访问, 寻址时间长), 索引分配 (index 将文件指针索引表存放于索引块, 实现随机访问 局部损坏不影响全局, 单点故障 存在内部碎片)

空闲空间管理: 建立空闲空间链表 (位向量 (1,0 表示空闲/非空闲), 链接 (指针链接空闲块 无浪费, 难找连续), 分组 (首块记录其它块), 计数 (记录连续空间首地址+计数))

系统恢复: 检测修复目录与磁盘相关性

日志系统: 审计 (控制记录访问), 统计 (记录优化服务)

磁盘管理 *Mass Storage Structure:*

磁盘结构: platter 磁片, track 磁道, sector 扇区, spindle 柱面

传输速度, 定位时间 (寻道时间, 旋转时间), 磁头损坏

组织: 扇区从第一柱面最外圈磁道开始分配

SAN 存储网: 通过网络连接磁盘

指标: 访问时间 (寻道, 旋转), 带宽 (单位时间传输资源数量)

磁盘调度 (计算):

FCFS: 磁头按请求顺序访问柱面 (Seek distance 寻道距离: 移动柱面数和)

SSTF 最短 seek 优先: 尽快解决附近柱面 (存在饥饿)

SCAN 电梯算法: 告知移动方向, 单向扫描到头折返 (双向处理)

C-SCAN: 单向移动, 折返时不做处理 (请求公平)

LOOK: 扫描到最小请求停止 (双向)

C-LOOK: 单向扫描不到头, 折返不做处理

SSTF 性能较好, SCAN 与 C-SCAN 适合磁盘任务重 (LOOK 同理)

磁盘管理: 物理格式化 (将磁盘按大小分区), 高级格式化 (生成文件系统)

ROM: 储存引导程序, 用于加载操作系统

坏块 Bad sector: 使用好块逻辑替换, 重新编址跳过坏块

Swap Space 交换分区: 不通过文件系统访问的磁盘区域

RAID 磁盘冗余阵列: 通过冗余提高可靠性, 并行性