Operation System

本文档主要是操作系统相关的一些八股文。

死锁的发生条件，如何解决

死锁是指一组进程/线程互相等待对方持有的资源,导致它们都无法继续执行的情况。死锁的产生有四个条件，互斥，不可剥夺，请求和保持以及循环等待。要解决死锁问题只要破解其中一个条件即可。
进程调度算法可以通过以下方式来预防和解决死锁：

- 银行家算法: 预先分配资源,确保不会进入死锁状态。
- 资源分配图算法: 检测资源分配图中是否存在环路,从而判断是否会发生死锁。
- 死锁检测与解除: 定期检测系统中是否存在死锁,如果存在则通过抢占资源或者终止进程来解除死锁。

进程和线程的区别?

进程是资源分配的基本单位，线程是CPU调度的基本单位。

从资源角度看：

- 进程拥有独立的虚拟地址空间和页表，切换时需要切换页表，导致TLB快表失效
- 线程共享进程的虚拟内存空间，切换时不会使TLB失效，开销更小

从切换成本看：

- 进程切换涉及虚拟内存空间切换和完整的上下文切换（寄存器组、页表等）
- 线程切换只需切换少量上下文（栈指针、程序计数器等），速度更快

从共享资源看：

- 进程间资源隔离，通信需要IPC机制
- 线程间共享全局变量、静态变量、堆、共享库和动态链接库，通信更便捷

进程的调度策略

- 先到先服务：非抢占式调度，实现简单但可能导致"护航效应"
- 最短作业优先：最小化平均等待时间，长进程可能饥饿
- 轮转调度：性能开销大或退化为先来先服务
- 多级反馈队列：I/O密集型进程：升至高优先级队列（快速响应）。CPU密集型进程：降至低优先级队列（避免抢占开销）。进程老化：低优先级进程随等待时间自动升级

虚拟内存是如何工作的?

虚拟内存是现代操作系统的核心内存管理技术，通过硬件（如MMU）与软件（操作系统）的协同，为进程提供连续的虚拟地址空间，并动态映射到物理内存或磁盘空间。

- 虚拟地址空间机制：每个进程拥有独立的虚拟地址空间（如32位系统为4GB），进程仅使用虚拟地址（VA）访问内存。虚拟内存包含RAM和磁盘交换空间，拓展可用内存，如手机使用虚拟内存机制提高可用内存。
- 分页机制：虚拟地址空间被划分为固定大小的页，操作系统为每个进程维护的映射表，存储虚拟页号（VPN）到物理页框号（PFN）的映射关系。

虚拟内存的优点：防止进程越界访问，提升安全性。允许程序使用超过物理内存的空间。

常用的虚拟内存性能优化技术:

- 加速地址转换：TLB（快表）是MMU内部的硬件缓存，存储近期使用的PTE，避免频繁访问主存页表。
- 多级页表：仅加载活跃部分的子页表，减少内存占用。

介绍一下用户态和内核态?

用户态（User Mode）和内核态（Kernel Mode）是操作系统中的两种核心运行模式，用于隔离应用程序与系统核心功能，保障系统安全性和稳定性。通过系统调用、中断和异常实现动态切换，平衡效率与安全。

- 用户态：应用程序独占，通过用户级页表映射物理内存，禁止执行特权指令，尝试执行会触发异常并切换到内核态
- 核心态：所有进程共享，通过内核级页表映射操作系统代码和数据，用户程序无法修改，可执行所有指令，可以访问内核资源。

上下文切换时需要保存用户态的哪些内容?

操作系统保存的内容如下（所有用户态上下文保存在进程控制块中）。上下文切换的本质是保存与恢复进程的执行现场。

- 寄存器状态：如程序计数器，堆栈指针，通用寄存器和状态寄存器的状态，需要记录
- 栈信息：保存了函数调用帧，切换时需要完整保存。
- 内存管理信息：上下文切换时需要记录内存管理信息，如页表基址和段寄存器。
- 其他用户态状态：如浮点寄存器或者线程的局部存储指针。

触发场景：主动切换如系统调用，被动切换如时间片耗尽。如果是内核态切换还会记录中断状态、内核栈等系统级信息。

操作系统如何管理堆上的碎片?

内存碎片分为内部和外部碎片。管理方法：

- 分页机制：将物理内存划分为固定大小的页框（如4KB），虚拟地址空间划分为相同大小的页。通过页表映射虚拟页到物理页框，解除物理地址连续性要求，大幅减少外部碎片
- 内存压缩：移动进程内存位置，合并空闲区域形成连续大块。
- 伙伴系统：内存按2^n大小分级。分配时递归二分大块，释放时检查相邻块（"伙伴"）是否空闲，若空闲则合并为更大块。
- Slab分配器：预分配固定大小内存池（如Linux kmalloc的Slab缓存），直接从池中分配/释放对象。
- 用户级分配器优化：合并空闲块，分离空闲列表，延迟释放内存。

CPU的多级缓存是什么样的?

CPU的多级缓存是一种分层存储结构，旨在解决CPU与主存（DRAM）之间的速度鸿沟。其核心设计基于局部性原理（时间局部性与空间局部性），通过多级高速缓存（SRAM）减少访问延迟，提升系统性能。通过空间换时间，将平均内存访问延迟降低10-100倍

CPU访问数据的顺序为：
寄存器 → L1 → L2 → L3 → 主存

缓存命中：数据在当前缓存层找到，直接返回（如L1命中仅需1-3周期）
缓存未命中：逐级向下查找，若L3未命中则访问主存，延迟骤增（主存延迟达100-300周期）
数据加载：未命中时，缓存以缓存行（通常64字节）为单位从下级存储加载数据，利用空间局部性。

缓存一致性?

缓存一致性（Cache Coherence）是计算机系统中确保多个缓存副本数据一致的关键机制，尤其在多核处理器、分布式系统或多级缓存架构中至关重要。其核心目标是解决因缓存复制导致的数据不一致问题，避免程序逻辑错误。

多核CPU的缓存一致性协议：MESI协议

每个缓存行（Cache Line）通过4种状态管理数据一致性：

M（Modified）：数据被修改，仅当前缓存有效，主存数据已过期。
E（Exclusive）：数据未被修改，仅当前缓存持有，主存数据最新。
S（Shared）：数据未被修改，多个缓存共享，主存数据最新。
I（Invalid）：缓存行数据无效，需重新加载。

运行机制：
当核心A修改数据时，通过总线广播消息，使其他核心中该数据的缓存行状态变为Invalid（写无效策略）。其他核心再次访问时需重新从主存或核心A的缓存加载最新数据

操作系统是如何检测死锁的?

操作系统检测死锁的核心机制是通过实时监控进程的资源请求与分配关系，判断是否满足死锁的四个必要条件（互斥、持有等待、不可剥夺、循环等待）。
主流有2种检测方法：

- 资源分配图法：将进程和资源建模为有向图，通过检测环路判断死锁。构建当前系统快照，简化图结构，判断死锁。
- 银行家算法：通过模拟资源分配，检查系统是否处于安全状态。

什么时候使用进程？什么时候使用线程?

进程的适用场景：

- 需要强隔离性与稳定性：如核心服务（数据库、OS模块）、安全敏感模块（支付系统）。
- 跨多核CPU的并行计算：科学计算、视频渲染、机器学习训练
- 资源独占需求：独立内存管理（如JVM进程）、专用硬件访问（GPU）

线程的适用场景：

- 高并发IO密集型任务：Web服务器（Nginx/Tomcat）、爬虫、实时聊天系统。
- 共享数据频繁的任务：GUI应用（后台线程更新UI）、多线程下载（分块共享进度）
- 轻量级任务调度：定时任务、后台日志写入、低延迟实时控制。

进程和线程有没有独立空间？具体的说一下？

进程是操作系统资源分配的基本单位，拥有完全独立的虚拟地址空间，其内存结构包含以下区域:代码段、数据段、堆、栈、内存映射区

线程是CPU调度的基本单位，隶属于进程，其内存空间分为两部分：

- 共享区域：代码段、数据段、堆、内存映射区
- 私有区域：栈(存储局部变量和函数调用链)、线程局部存储、寄存器上下文

CPU调度是以进程还是线程的形式实现的？

CPU调度的基本单位是线程，优点是：

- 轻量级上下文切换
- 细粒度并发控制
- 资源分配与调度的分离

Linux中使用轻量级进程LWP，直接调度ready状态的线程分配时间片。
线程调度策略主要有：

- 时间片轮转
- 优先级调度
- 负载均衡

操作系统如何管理内存？

操作系统管理内存的方法主要是物理内存管理和虚拟内存管理两种。虚拟内存上面已经讲过了。
物理内存管理策略：

- 连续分配策略：连续分配策略分为固定分区、动态分区和伙伴系统
- 非连续分配策略：分页管理、分段管理和段页式结合。

虚拟内存有哪些页面调度算法？

虚拟内存中的页面调度算法是操作系统在物理内存不足时，决定哪些页面应被换出到磁盘的核心策略（基于程序局部性原理）

- 先进先出（FIFO）：维护页面进入时间的队列，置换队首页面。适用于简单嵌入式系统或对性能要求不高的场景。
- 最近最少使用（LRU）：记录每个页面的最后访问时间，淘汰最久未被访问的页面。
- 最近最不常用（LFU）：为每个页面设置计数器，访问时计数器递增；定期（如每秒）清零计数器避免历史权重过大。适用于访问模式稳定的应用（如数据库缓存）
- 最佳置换（OPT）：需预知未来页面访问序列（实际不可实现），仅用于理论对比。
- 随机置换：依赖硬件随机数生成器

优化版本：

- Clock算法：LRU的近似实现，通过访问位（Reference Bit）标记页面是否被访问
- 工作集模型：基于时间窗口（如最近τ次访问），统计活跃页面
- 页面缓冲算法：将被淘汰页面暂存缓冲池，若短期内再次访问可直接恢复，减少磁盘I/O

什么是僵尸进程？什么是孤儿进程？这两者会带来什么风险？

僵尸进程和孤儿进程是操作系统中两种特殊的进程状态，尤其在类UNIX系统（如Linux）中常见。

- 僵尸进程：子进程已执行结束，但是未被父进程回收，进程表中仍保留其记录。不占用CPU或内存，但占用进程ID，无法被 kill -9 终止，只能由父进程回收。会导致进程表耗尽、资源泄漏隐患和安全风险。
- 孤儿进程：父进程先于子进程终止，子进程失去父进程管理，由初始进程负责其资源回收。仍在运行，但父进程ID（PPID）变为1。会导致资源占用、管理复杂性和安全风险。

页表是进程还是线程级别？

页表是进程管理的。
进程拥有独立的页表：每个进程创建时，操作系统会为其分配独立的页表结构，用于管理虚拟地址到物理地址的映射关系。这种独立性确保了进程间内存空间的隔离性。同一进程内的所有线程共享该进程的页表。

特例：
内核线程（如kthreadd）不关联用户空间，因此无需用户态页表，仅使用内核页表（所有进程共享）。但此类线程属于内核调度实体，与用户级线程/进程的设计目标不同。

OS检测core dump发生的机制是什么？

操作系统（OS）检测并生成 Core Dump 的机制是一个多步骤的协作过程，涉及信号触发、内核处理、资源检查和内存捕获等关键环节。
信号触发捕获异常事件，操作系统内核接收到信号后，递送信号，检查资源，调用 do_coredump()，然后内存捕获，生成core dump，最后终止进程。

缺页中断什么时候触发？

- 当程序试图访问的页面（Page）不在物理内存（RAM）中时，操作系统无法立即满足这个请求，这将触发一个缺页中断。这种情况通常发生在虚拟内存系统中，其中只有部分页面被加载到物理内存中。
- 当程序试图执行的操作违反了内存保护规则时，也会触发缺页中断。例如，程序试图写入一个只读页面，或者试图访问一个没有权限访问的页面。

有名管道和匿名管道的区别？

- 匿名管道：只能用于具有父子关系的进程之间通讯。在创建管道时，调用pipe() 函数即可，管道不存在于文件系统中，只存在于内存中，因此无法被其他进程访问。
- 有名管道：可以用于任意两个进程之间通讯，不管它们是否具有父子关系。在创建管道时，需要调用 mkfifo() 函数，管道会创建一个文件系统中的特殊文件，可以像普通文件一样被多个进程访问。

共享内存在Linux系统中是如何实现的？

共享内存在Linux系统中是一种高效的进程间通信（IPC）机制，通过让多个进程直接访问同一块物理内存区域实现数据共享，避免了数据复制的开销。其实现主要依赖两种机制：System V共享内存和POSIX共享内存。共享内存的核心原理就是在物理内存中开辟一块区域，多个进程通过页表映射将其映射到各自的虚拟地址空间

- System V共享内存：先创建共享内存，然后映射到进程地址空间，通过指针进行数据读写，如果不用了可以解除映射并删除共享内存。
- POSIX共享内存：POSIX共享内存基于内存映射文件，更符合文件操作习惯。先创建/打开共享内存，调整大小之后映射到进程空间，不使用该文件时解除映射和删除。

共享内存和管道的区别?

- 共享内存是一种高效的进程通讯方式，允许多个进程直接读写共享的内存区域，因此可以实现数据的快速交换（共享内存通常是最快的进程间通信方式）。但是需要特别注意同步和互斥的问题，以防止数据访问冲突。
- 管道是一种单向的通讯方式，通常用于父子进程之间的通讯，只支持单向的数据流动。管道数据的传输是通过内核进行缓冲的，因此可能会存在一定的性能开销。

共享内存的优点?

- 零拷贝：共享内存避免了数据的复制过程，数据直接存放在内存中，进程可以直接读写内存中的数据，因此具有零拷贝的特性。
- 低开销：由于共享内存是直接存放在内存中的，因此在数据传输过程中几乎没有额外的开销，避免了缓冲操作和系统调用的开销。
- 高效性：共享内存的读写操作速度非常快，适合于高频率和大数据量的通信需求。

并发和并行的区别?

操作系统通过引入进程和线程，使得程序能够并发运行。

- 并发是指宏观上在一段时间内能同时运行多个程序
- 并行则指同一时刻能运行多个指令

并行需要硬件支持，如多流水线、多核处理器或者分布式计算系统。

了解的I/O模型有哪些?

I/O模型有5种：

- 阻塞式I/O:程序发起I/O系统调用后线程挂起，内核准备好数据并将数据复制到用户空间后才继续执行线程。适用于低并发的场景和一些单线程服务场景。
- 非阻塞式I/O:发起I/O调用后立刻返回状态，轮询检查内核数据是否就绪，就绪后发起数据复制。高频轮询会导致CPU飙高，实时性较差。适用于传感器数据采集等轻任务场景。
- I/O多路复用：通过select/poll/epoll等系统调用单线程监控多个I/O描述符。如果某一个描述符就绪就通知程序进行处理。I/O多路复用支持了高并发，但是需要处理多路事件状态。I/O多路复用适用于Nginx等高并发服务。
- 信号驱动I/O:应用程序注册回调函数，发起I/O时立刻返回。内核就绪时发送信号，进程调用recvfrom复制数据。避免了轮询等待，但是处理信号复杂，适用于嵌入式设备的低并发实时系统。
- 异步I/O：调用aio_read后立刻返回，内核负责数据准备和复制，完成后通过回调、事件通知程序。适用于大规模分布式系统等场景。

读饥饿是什么?

读饥饿指在并发访问共享资源时，读者进程因资源分配策略不公平而长期无法获取资源的现象。
在读写问题中，当系统采用写者优先策略时，读者的读事务被无限延迟。

读饥饿的解决方法：

- 公平锁
- 限制读者数量
- 优先级动态调整