操作系统

用户态和内核态

> 区别？

内核态 CPU可以执行所有的指令和访问所有的硬件资源 权限更高 用于操作系统内核的运行

用户态 CPU权限比较低 主要用于运行用户程序

避免恶意程序直接调用内核函数 有利于系统的维护模块化 故障分离

进程

底层

> 进程和线程的区别？

线程共享进程的资源 不会为线程分配内存 但是线程自己资源不共享 （ThreadLocal联想）

进程是资源分配的基本单位 线程是任务调度和执行的基本单位

故障隔离 进程的程序和上下文切换开销大，但是线程切换开销小（会有竞争）

进程也有隔离性和独立性 对其他进程不会有影响

> 进程分配的资源是？

虚拟内存 文件句柄、信号量

文件句柄指文件的标识 每个文件都有唯一标识

> 为什么要设计线程

串行化导致速度很慢

如果不是前后依赖的数据的话 可以并发执行　最后再组装数据

但是多进程会有通信问题，维护进程创建和销毁开销很大

需要一种可以共享资源和并发运行的实体

> 多线程的劣势和优势

劣势的话 有线程安全问题 需要锁机制

优势是 执行速度更快 利用多核处理器的优势

> 多线程的数量？

销毁始终要开销，而且锁要更复杂了 可能会死锁

> 进程切换和线程切换区别？

进程切换开销更大 进程涉及到地址空间 全局变量 文件描述符

线程只涉及到线程堆栈 寄存器 程序计数器等

线程堆栈： 递归调用 过程中，每次递归都会创建新的栈帧

> 线程切换的过程 *(实现并发执行)*

操作系统保存上下文信息---->执行权转移到调度器，选择下一个线程----->从保存的上下文信息中恢复执行---->执行权切换到新线程

上下文信息保存在TCB（每个线程一个）

> 进程状态

创建

就绪和阻塞 运行 三态的变化

结束

阻塞只能由运行态转变来 然后阻塞等到事件完成后去到就绪态

> 进程的上下文

进程的切换---》上下文切换 就是时间片分配（并发）的问题嘛

上下文切换问题主要的key：保存运行状态 就是说运行的位置 （指令位置--->PC计数器和寄存器）

进程上下文切换：虚拟内存 栈 全局变量 内核堆栈 寄存器   保存后 然后调度下一个进程 让这个进程读取自己的上下文信息

进程、线程通讯

> 管道 都是单向的

匿名管道：无格式的流，大小受限 单向通信，只能用于父子关系的进程之间通信

命名管道：关系无限制， 严格 先进先出

> 消息队列

消息的链表，有特定格式，FIFO 也可以随机查询 ，与管道不同的是消息队列放在内核中，内核重启（或显式删除）才会删除

> 信号量

计数器 锁机制

> 信号

和信号量不同，用于通知接收进程某个事件已经发生

进程有三种方式响应信号 **1.** 执行默认操作、**2.** 捕捉信号、**3.** 忽略信号

> 共享内存

分配共享空间 让大家访问 最快

> *Socket*

TCP UDP 本地进程间通信

> 共享内存怎么实现的？

拿出一块虚拟地址空间来，映射到相同的物理内存中

> 线程间的通讯？

互斥锁 条件变量 Condition 自旋锁 CAS 信号量 读写锁

这几个太常见了

自旋锁 不停地查锁的状态

线程调度

> 先来先服务

先来先服务 （FIFO）

先来后到

最先的运行完才能后面

> 最短作业优先调度算法

优先运行时间短的

对长作业不利 非抢占式

> 最短剩余时间优先

抢占式

运行时间比现在运行的进程剩余时间短 就运行这个

> 高响应比优先调度算法

权衡了短作业和长作业

$$优先权 = \frac{等待时间 + 要求服务时间}{要求服务时间}$$

等待时间相同，要求服务时间越长 则优先权越高 利于短作业

要求服务时间一样，等待时间越长，兼顾长作业

（但总体还是利于短作业）

> 时间片轮转调度算法

该进程在时间片结束前阻塞或结束，则 CPU 立即进行切换；

到时就切

通常时间片设为 `20ms~50ms` 通常是一个比较合理的折中值。

轮着来 没啥

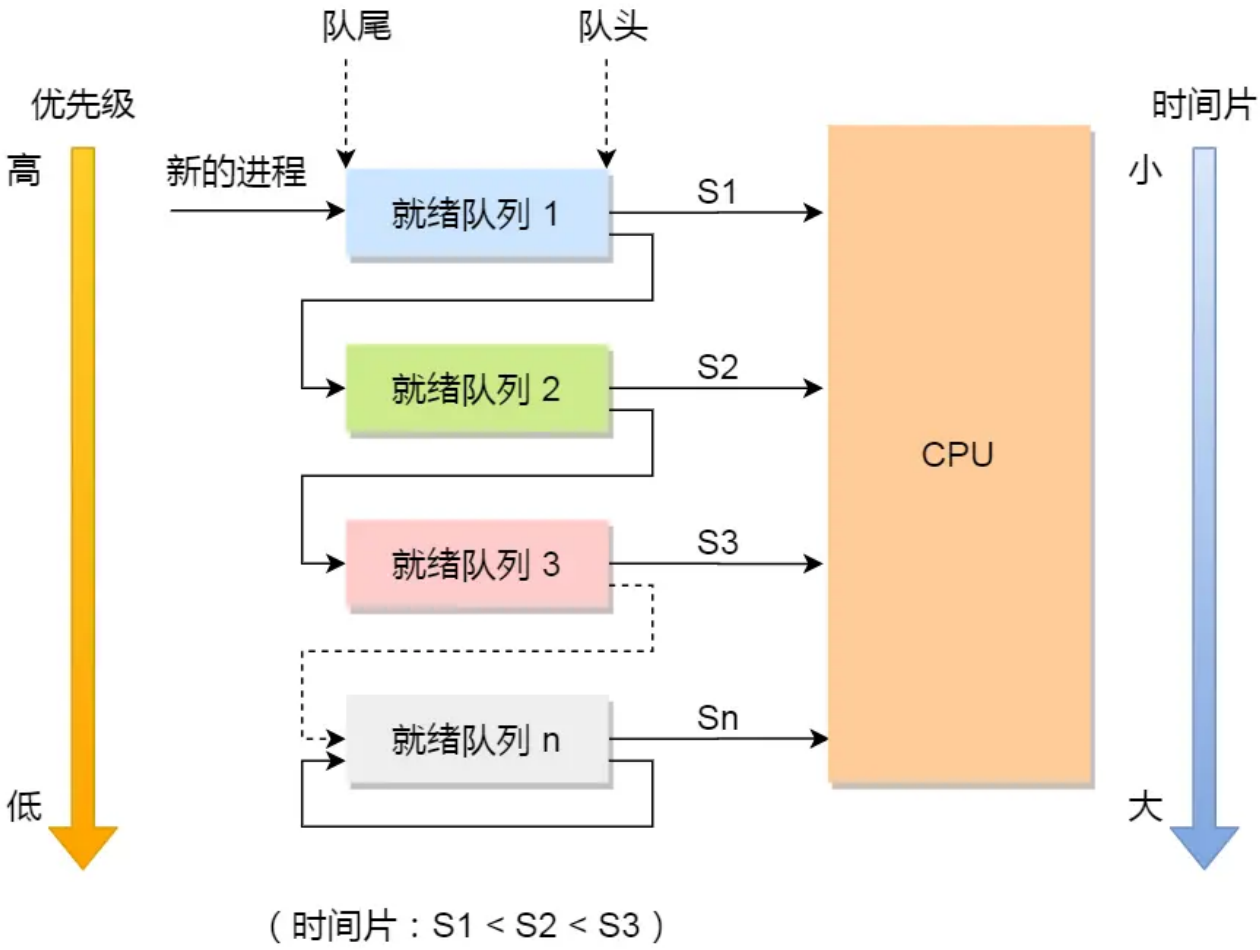
> 最高优先级调度算法

从就绪队列中选择最高优先级的进程进行运行

静态和动态优先级 (动态的话可以自己设置)

非抢占式和抢占式的(当就绪队列中出现优先级高的进程，当前进程挂起，调度优先级高的进程运行。)

> 多级反馈队列调度算法



每个队列优先级从高到低，同时优先级越高时间片越短；

新进程先进第一队列 然后没执行完就往下走

高的队列空了 才会去执行低的队列的进程

如果有新的加入（高的队列）则停下移入到原队列末尾 让优先级高的先运行

如果没新的加入 在这个队列还是没执行完 则继续往下走

锁 （结合**java**锁来看吧）

> 自旋锁

CAS函数把两道步骤合成一道原子命令

看锁的状态，加锁

用while来自旋

需要注意,在单核 CPU 上,需要抢占式的调度器(即不断通过时钟中断一个线程,运行其他程序），否则while循环会卡死

> 死锁条件

互斥条件

持有并等待条件

不可剥夺条件

环路等待条件

> 如何避免死锁 与银行家算法

使用资源有序分配法，来破环环路等待条件。

银行家算法：本质上是一种DFS 做算术罢了没啥

> 乐观锁和悲观锁

悲观锁 ：抢占式 先来者抢占锁 占有资源 互斥

乐观锁：适用于读多写少的场景 读数据时不会加锁 更新时检查版本 匹配则更新 否则认为发生冲突。

冲突的话看处理策略咯 可以抛异常 也可以读最新的来取

内存管理(可以结合**JVM)**

虚拟内存和页表

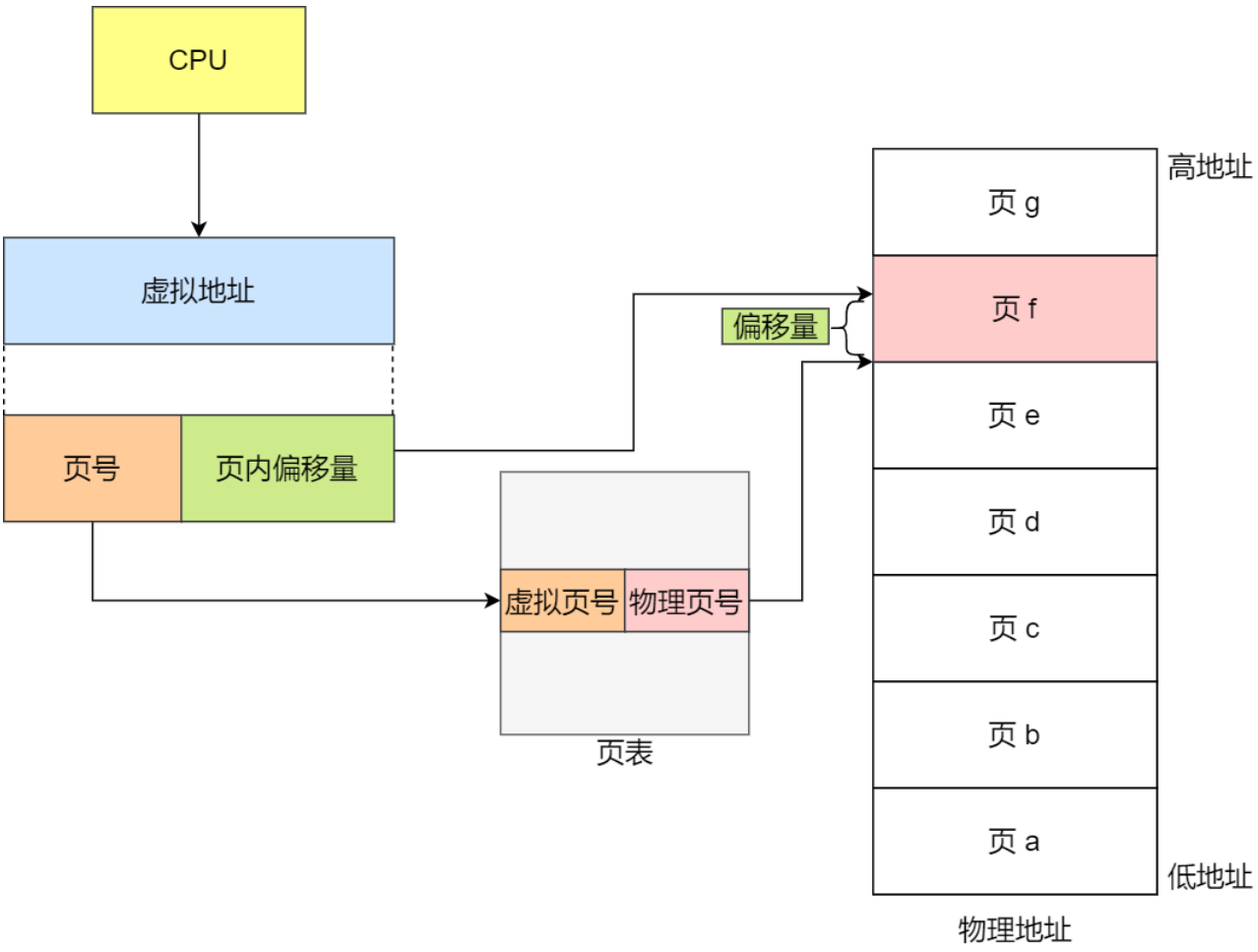
每个进程有自己的虚拟内存，使进程的运行内存超过物理内存大小（放磁盘上了）

每个进程的页表私有 互不干涉 每个进程页表只有一个？

标记是否存在和权限 比直接访问物理内存更安全

内存管理单元 （*MMU*）就做将虚拟内存地址转换成物理地址的工作。

缺页就重新找 更新页表



> 段表

应用程序的虚拟地址空间被分为大小不等的段，段是有实际意义的，每个段定义了一组逻辑信息，例如有主程序段 MAIN、子程序段 X、数据段 D 及栈段 S 等

> 区别?

页大小固定　而段不固定

分段会有外部内存碎片　而这些段可能在物理内存中不连续地分布。随着段的分配和释放，内存中可能会有一些空闲区域

分页有内部内存碎片　每一页的大小是固定的，当一个段的数据小于一页时，剩余的内存就会浪费掉

段表记录每个段的基地址和段的大小。 页表记录虚拟页号与物理页号的映射关系。

> 段页机制

段页机制将程序的内存划分为多个段，每个段内部再被划分成若干个页。这一机制将分段和分页的优点结合在一起

段号（**Segment Number**）：用来标识程序中的哪个段（如代码段、数据段等）。

页号（**Page Number**）：用来标识段内的哪一页。 （相对于基地址的）

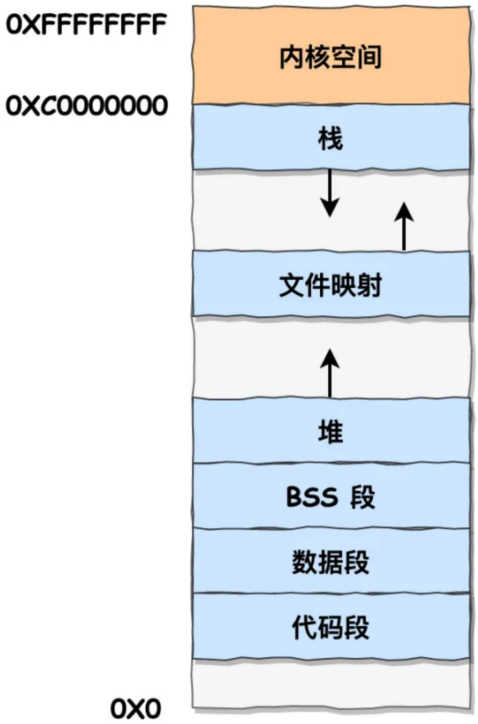
页内偏移（**Offset**）：标识页内的具体位置，表示数据在页内的位置。

> *TLB*和多级页表

TLB 存储了最近访问过的虚拟地址到物理地址的映射（即页表项）。当程序访问内存时，TLB 会首先检查该地址的映射是否存在，如果存在（称为TLB命中），则直接获取物理地址。如果映射不在TLB中（称为TLB未命中），系统会去查找页表，并将新的映射加载到TLB中。

内存其他

> 内存布局



代码段：可执行代码

数据段：包括已初始化的静态常量和全局变量

BSS段：包括未初始化的静态变量和全局变量

堆段：动态分配的内存

文件映射段 包括动态库 共享内存

栈段 包括局部变量和函数调用的上下文

> 堆和栈的区别

分配方式：堆是动态分配内存　程序员手动申请和释放的

栈是静态分配的，编译器自动分配和释放 存储函数的局部变量和调用信息

内存管理：　堆要手动　栈自动 （函数递归栈）

大小和速度：堆比栈大　 栈更快

> *fork*函数

用于创建一个新的进程。新创建的进程被称为子进程，它是从调用 `fork()` 的父进程复制过来的

父进程页表（Page Table）会被复制，但指向相同的物理内存。

所有共享的物理内存页都会被标记为 只读（Read-Only）。

CPU 进行写操作时，发现该页是只读的，就会触发 缺页异常（**Page Fault**）。

操作系统检测到该页是 COW 共享页，会：

- 为该进程分配新的物理内存页。
- 复制原数据到新的物理页中。

- 更新该进程的页表，使其指向新分配的内存。
- 解除新页的只读限制，让该进程可以正常写入。

只有真正需要修改的数据才会被复制，而不是 `fork()` 时就复制所有内容 节省大量时间

页面置换算法

当出现缺页异常，需调入新页面而内存已满时，选择被置换的物理页面

> 最佳页面置换算法

置换未来 下一次最长时间才访问到的页面

但是不可能预知未来 所以实现不了

> 先进先出置换算法

队列嘛 也没啥 但是性能较差 因为跟后续访问无关 导致缺页多

> *LRU*算法

最长时间没被访问的页面进行置换

问题是开销大

> 时钟页面置换方法

检查 访问位 然后轮询

访问过则1 没访问过位0

发生缺页则轮询 找到0则清楚该页面 为1则设为0 转到下一个

洗刷刷。。。。

现代系统基于CLOCK比较多

> *LFU算法*

当发生缺页中断时，选择「访问次数」最少的那个页面

只考虑了访问次数 没有区分冷热问题　上一个时段访问多 不代表这个时段访问多 误伤现在的进程

淘汰访问次数最少的页面

要存和查访问次数 成本高 开销大

LFU只考虑了访问次数 没有区分冷热问题　上一个时段访问多 不代表这个时段访问多 误伤现在的进程

解决的话可以定期减少访问次数

中断

> 流程

中断发生：收到中断信号 停止执行当前命令 保存当前执行现场 跳到中断处理程序执行

中断响应：根据中断向量表 找到处理程序的入口地址　处理器保存当前执行现场

中断处理：根据入口地址跳转 执行中断处理程序。

> 中断类型

外部中断：可屏蔽和不可屏蔽

又称硬件中断　显卡网卡。。。　INTR和NMI中断信号线　前者可屏蔽 后者不可屏蔽

不可屏蔽一般会导致系统无法正常运行

内部中断：软中断和异常

软中断一般是系统调用

异常是CPU内部产生的错误　不可屏蔽但是不会导致系统崩溃

> 中断作用

使得CPU可以处理突发事件 包括IO这种的

像IO这种的可以支持并发处理

但是频繁IO会导致开销大 所以就有了DMA

DMA（直接内存访问）** 只需要在 数据传输完成后 触发一次 中断通知 **CPU**，让 CPU 知道数据已经准备好，可以进行后续处理。

IO模型

BIO NIO AIO

这个看网络相关的md

磁盘

磁盘调度算法

> 先来先服务算法

寻道时间长，哪个先来就去那里 容易出现饥饿 （长作业先来）

> 最短寻道时间优先

哪个近去哪里

也容易饥饿 困在近的地方出不来

> 扫描算法 *scan*

也叫电梯，来回完整遍历

到了边界再换方向 错过了就要等换方向

> *C-SCAN*

只按一个方向扫描 到边界就马上回起点再来 问题是相比SCAN算法 离边界（终点）近的会被放在后面扫

> *LOOK* 边扫描边观察

如果在此方向上 后面的路都没请求 则马上调头扫

> *C-LOOK* 均衡循环扫描

该方向上剩余的路都没请求 就马上回到起点

C和非C区别在于 C是回起点 非C是仅调换方向

SCAN和LOOK区别是 前者到边界再调换方向 后者没请求了就换方向

文件系统 **(linux)**

软连接类似于快捷方式

硬链接：源文件和硬链接 删除一个对另外一个没影响 但是删除索引也相当于访问不到了（删了）

删除所有硬链接：

- inode 的 引用计数降为 **0**，文件数据才会被操作系统回收（释放磁盘空间）。

删除源文件和硬链接文件才算真正删除

不能直接删除 **inode**，但可以通过删除所有硬链接，使操作系统自动回收 inode。