

To begin with...

操作系统

多道编程

并发 vs. 并行

硬件

处理器

- CPU寄存器

- Rank: 权限分配是一切架设的物理基础

内存

- 局部最优

- 管理

IO

- 中断

总线

计算机启动和引导

系统上电时: CPU跳转到启动地址 (0xFFFF0或全0) 取指【5】

OS加载过程: 硬盘数据载入内存【3】

进程环境建立: OS接管运行【6】

第一个进程建立时

进程的创建和管理

fork&exec——两个系统调用

进程与OS交互获得资源 (运行时)

资源回收

线程创建时

调度的代价

文件

- 修改时

- 写回机制! !

- 删除时

系统关机时

设备管理

设备接入方式

三种设备的通信模式 (打印机)

- 忙等待模式

- 中断模式

- DMA (Direct Memory Access) 模式

IO通道

IO软件

中断形成后, 软件做什么:

驱动程序

磁盘管理

死锁

对比图!! 【策略——模式——优点——缺点】

死锁预防: 由OS为资源请求&分配 定义约束规则 (规则可以破坏死锁条件)

死锁避免: 由OS分析当前资源分配状态, 试图安全分配

银行家算法:

死锁检测&恢复: 一个特殊进程负责检测, 杀死死锁进程

- 检测

- 恢复策略

非资源死锁问题【4】

文件系统

文件管理: OS里负责数据存储、索引、保护、共享的模块

文件结构

文件结构对比!

分层文件系统: 文件以目录方式组织

存储结构

文件别名：多个文件名关联同一个文件

一致性检查

磁盘访问时间！

访问权限

VFS——虚拟文件系统：解决FS的识别问题，对不同FS抽象

磁盘调度算法：优化磁盘访问请求顺序【7】

RAID——Redundant Array Of Inexpensive Disks

磁盘缓存

内存管理

虚拟内存：进程的内存布局

compile>>assemble>>LA>>link(.lib)>>relocate(+BA)>>PA

位图&链表——管理内存空洞

分区分配算法

碎片整理

覆盖技术

虚拟存储

内存管理单元——MMU (Memory Management Unit)

分页

 页表：以页号+PTE为索引&Frame (的物理) 地址为内容

多级页表——解决页表太大导致内存利用率低

TLB (Translation Look-aside Buffer)——减少二次访存

反置页表：页表项以Frame号为索引，Page号为内容

分页系统工作流程

 缺页中断

 页式存储管理性能度量——EAT (Effective Memory Access Time)：有效存储访问时间

页置换算法

 抖动——CPU利用率&进程并发数的关系

 负载控制：调节进程并发数 (MPL)

 工作集算法

 缺页率置换算法 (PFF: Page Fault Frequency)

 页置换算法比较！

段式

 段页式：用段技术组织程序内容，页技术组织物理内存 e.g. 给每个段加一级页表

 页&段式对比！

page_free_list：找页分配

进程管理

 - 创建

 - 调度

进程

进程：程序运行的过程，包括输入输出、程序和状态

PCB/进程描述符/属性

进程内存布局

进程切换——上下文切换

switch_to

进程镜像：对进程整个生命周期的描述

进程状态

 - 三状态模型

 进程挂起：对进程做分级处理，引入优先级会使某进程等待时间过长而被换至外存（运行镜像被换出到硬盘）

进程调度：CPU资源的时分复用

调度算法：改变响应时间、等待时间

 - 非抢占式

 - 抢占式

 基于优先级

进程通信方式

 几种常用方式

 Meltdown

Hammer Attack——KSM: Kernel Samepage Merging (默认用后出现的页面替代新出现的) : 申请三个连续页面, 中间的复制密码, 等待合并
线程: 进程内一个相对独立的、具有可调度特性的指令执行流的最小单元, 是CPU调度的基本单位
同步和互斥
IPC解决方法
用户与内核交互

To begin with...

操作系统

“操作”层面: 用高效、合理的方法管理计算机, 驱动硬件、管理软件, 控制程序运行和提供服务。

“系统”层面: 是一个特殊的系统软件, 一个虚拟机器和资源管理者。运用一系列理论、机制、算法及技术。给使用计算机的一个便利平台。

功能:

- **硬件抽象:** 统一接口、隐藏细节
- **应用程序集成:** 提供简单的编程方法、调度&控制程序运行
- **环境管理:** 监视并提供安全稳定的空间

特性:

- **并发:** 伪并行
- **共享和互斥:** 如何提高性能
- **虚拟化和抽象:** OS是一个统一的shell, 使用户便于控制计算机; 提供硬件的不同抽象接口。
- **不确定性:** 上下文不能在分时系统中复现 (?) 程序运行过程中不能建立计算机环境; 用户进程对其他进程和调度机制不可见。

多道编程

在内存同时放若干道程序, 使它们在系统中并发执行, 共享系统中的各种资源。当一道程序暂停执行时, CPU立即转去执行另一道程序。

静态

- 内存中有多于一个进程
- 每个进程独占CPU
- 等待IO时CPU空闲

并发

- 进程在需要时占领CPU
- 空闲进程主动放弃给其他进程

并发 vs. 并行

并行：多个任务同时执行 e.g. superscalar & super-pipeline

并发：任何时间仅有一个任务执行。一个time zone内完成多个任务。

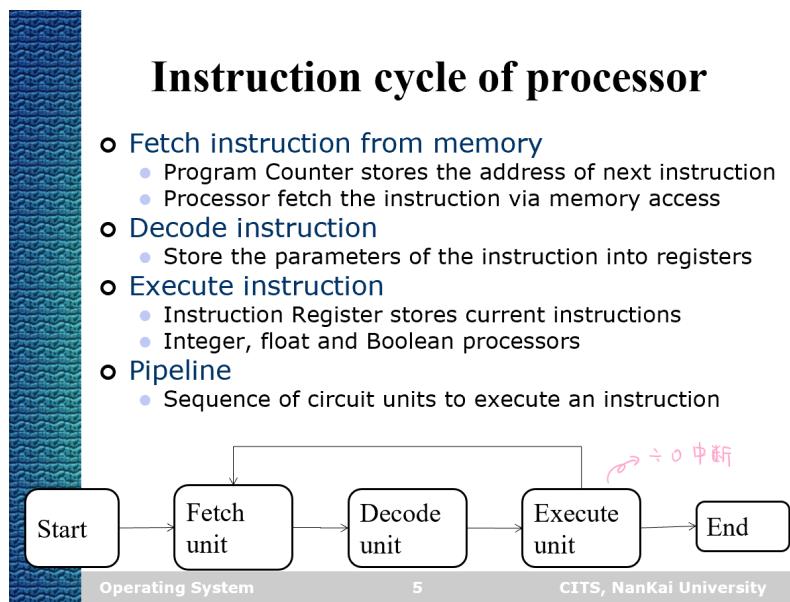
- 带来的挑战
 - 共享：资源分配、冲突解决；避免死锁
 - 不确定性：互斥和同步；不连续运行造成共享数据混乱--实现进程合作
 - 独立性：调度和保护
 - 实用性：进程交互；进程/线程等级

why进程？更好地使多道程序并发执行，提高资源利用率和系统吞吐量，增加并发程度。

why线程？减少程序在并发执行所付出的时空开销，提高OS的并发能力.

硬件

处理器

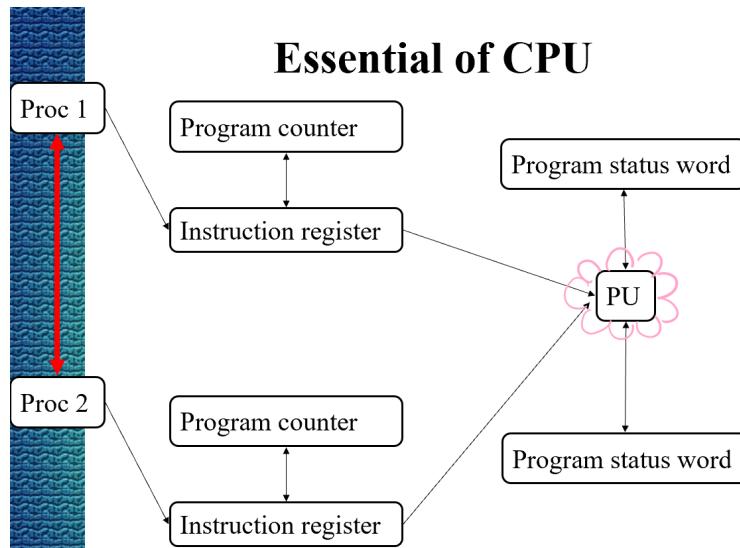


- CPU寄存器

- 地址、数据、标志、控制&状态寄存器
- PC：每个进程都有私有PC值
- IR：指令寄存器。执行阶段存要被执行的。同上。
- PSW (Program Status Word) : 存标志和CPU工作状态。是最重要的上下文！



PSW of Intel CPU



- Rank: 权限分配是一切架设的物理基础

- R0: 内核模式
- R1: 重要的设备驱动器和IO线路
- R2: 受保护的程序 e.g. IDE
- R3: 用户模式 普通程序

“trap” 用来从用户模式切换到内核模式

内存

- 局部最优

- 空间上: 下条指令的地址与当前指令相邻
- 时间上: 最近被使用的指令在近期会被再次使用

cache只存both空间&时间局部

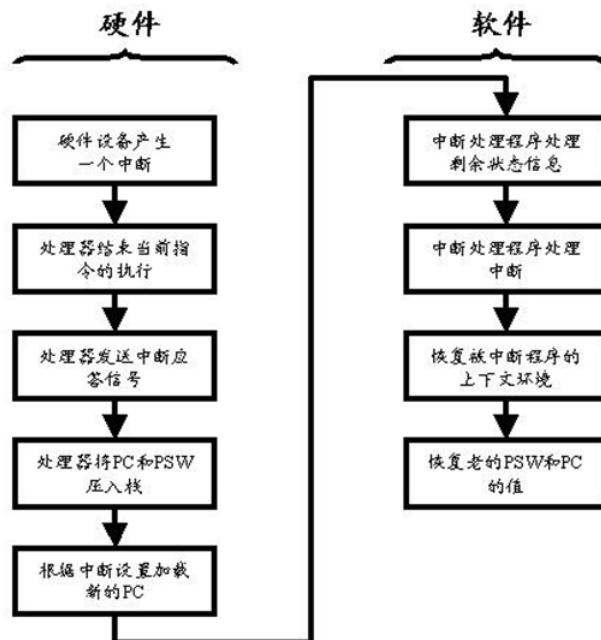
- 管理

硬件提供基址和界限寄存器为内存管理提供基础

IO

时钟——时间的意义: 执行的速度、资源管理的标准、数据传输的带宽、现实时间

- 中断



硬件：硬件产生 >> 处理器结束当前指令并信号响应 >> 处理器将PC&PSW压栈 >> 根据中断设置新PC（中断向量表找中断处理程序地址） >>

软件：处理剩余状态信息 >> 中断处理程序 >> 恢复上下文 >> 恢复PSW&PC

再来一次碎碎念：

- **异常**: 由程序产生、硬件发现的错误。
 - e.g. 除0、浮点下溢、拒绝访问
 - OS在内核模式下处理，或者通知程序由用户程序处理
- **中断**: 硬件发给CPU的信号
 - CPU发现并在内核模式下引起中断处理
 - 在分时共享的OST下，中断在时钟中断的方式下处理
- **系统调用**: 由OS提供的接口，使程序获得系统服务
 - trap指令会导致进入内核模式，引起OS执行服务

处理器与IO交互：

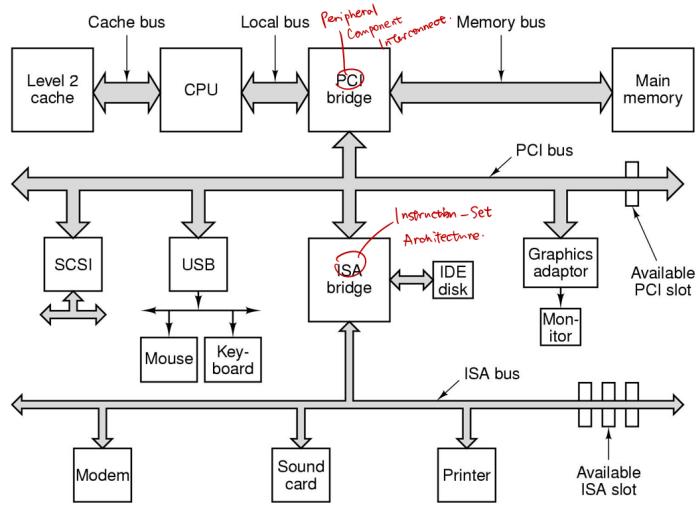
中断方式——发送完命令转去做其他工作

专用指令完成IO访问/（内存）指令格式不变

IO与存储器交互：DMA

程序控制IO: 由IO软件控制IO操作，进行设备状态检测、发送读写命令、传送数据等，**无中断**。

总线



- 北桥：在CPU和内存/AGP之间传数据
- 南桥：在IO总线和设备间传数据
- IO和CPU通信：
 - CPU不直接访问IO设备
 - 采用诸如DMA、IO通道的方式连接南北桥

计算机启动和引导

系统上电时：CPU跳转到启动地址（0xFFFF0或全0）取指【5】

- BIOS 存储基本指令
- BIOS 可适配大量硬件——兼容机的标准化
- 硬件自检：显示、内存、存储、输入
- 设备简易初始化
- 找到引导设备，读取引导信息 MBR (Master Boot Record)

OS加载过程：硬盘数据载入内存【3】

- MBR 中包含引导程序、分区表、校验和
- 读取分区表定位到分区，读取引导信息 VBR
- VBR 包含文件系统的读取方式和OS

进程环境建立：OS接管运行【6】

- 获得物理内存分布，建立 (1) 地址映射
- 管理 (2) 中断，设置响应函数
- 获得并初始化连接的 (3) 设备列表
- (4) 加载设备管理、文件系统、网络.....

- 创建 (5) 第一个进程 (“来自硬盘的application”)
- init进程 (6) 创建用户交互进程
 - 接受命令
 - 图形化交互
 - 回收资源

第一个进程建立时

- 内核空间
 - 有各式中断响应、设备驱动、文件系统 (optional)
 - PCB
 - 其他资源
- 用户空间：代码段、数据段在对应位置上
- 页表把连续的虚拟地址放到合适的物理地址
- 时钟及其他中断驱动系统运转

进程的创建和管理

fork&exec——两个系统调用

- **fork**: 创建新进程，为就绪态，资源共享，COW (WWWC)
- **exec**: 子进程调用，改变执行内容。依据**二进制文件格式**重建**页表映射**
 - 根据文件路径从FS中找到指定文件
 - 文件目录树
 - 磁盘驱动管理磁臂&扇区
 - 将文件映射入内存
 - 按编译器设定的地址建立映射
 - 加载外部依赖 (.dll)
 - 程序执行
 - 缺页
 - 分配物理页 (修改页表)
 - 执行指令、打开设备、访问驱动
 - 调度导致中断&恢复
 - 被换出的内存页表处理

进程与OS交互获得资源 (运行时)

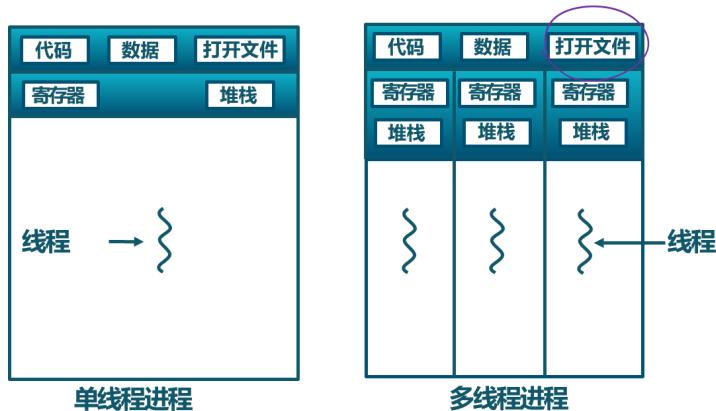
- 虚拟内存不是资源，因为可以共享
- 资源可能按块分配，有空闲
- 内存使用
- 文件使用：FS打开文件，文件映射进内存
- IO使用
- 当前目录、父子进程、线程

资源回收

- 主动: free、fclose
- 进程退出时:
 - 父进程接收子进程退出信号回收资源 (e.g. 父进程在 fork() 后 wait(& rtn)——等待子进程结束)
 - 根据**PCB**关闭文件回收物理页
 - init接管孤儿进程

线程创建时

- `pthread_create(`
`&ntid, //线程标识符指针`
`NULL, //线程属性`
`thr_fn, //执行函数的起始地址`
`NULL //函数参数`
`);`
- `pthread_join(//以阻塞的方式等待线程结束`
`ntid,`
`NULL //存储线程返回值`
`);`
- 创建新的“执行体”，用于执行某个函数
- 浅拷贝**PCB**
- 深拷贝执行相关的：**寄存器、状态字、栈**



调度的代价

对一个任务，多线程由于多核的调度 (e.g. 换页) 变慢，导致四核上的四线程和单线程速度相同。

文件

- 修改时

- 在内存中找到或创建一个空 inode 或索引节点
- 修改**文件夹**对应的数据节点
- 在**内存**中创建对应的数据节点
- 建立数据节点和索引节点的**对应关系**
- 写调用返回
- 系统负责在合适时机写回

- 写回机制！！

- 保证写回顺序用flush (从缓冲区到磁盘)
- 磁盘调度管缓冲区写回磁盘
- 置换算法管内存到缓冲区

BFS: Barrier-enabled File System——保证写操作顺序 (?)

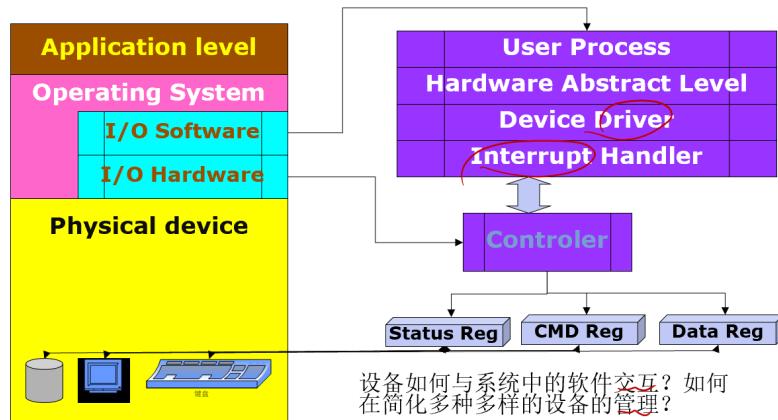
- 删除时

- 找对应inode
- 内存中删除inode
- 内存中释放页面
- 硬盘中删除inode
- 硬盘中释放数据块
- 硬盘中标记块未使用
- ==> 标记文件的块 (**硬盘上**) 不被使用 取代三个硬盘操作 提速
- 文件夹项清除

系统关机时

- **init** 收到特定信号时发起
- 用**停止信号**通知所有进程退出——SIGSTOP
- 用**杀死信号**保证所有进程退出——SIGKILL
- 各层各式内存**数据回写**
- 关闭电源

设备管理

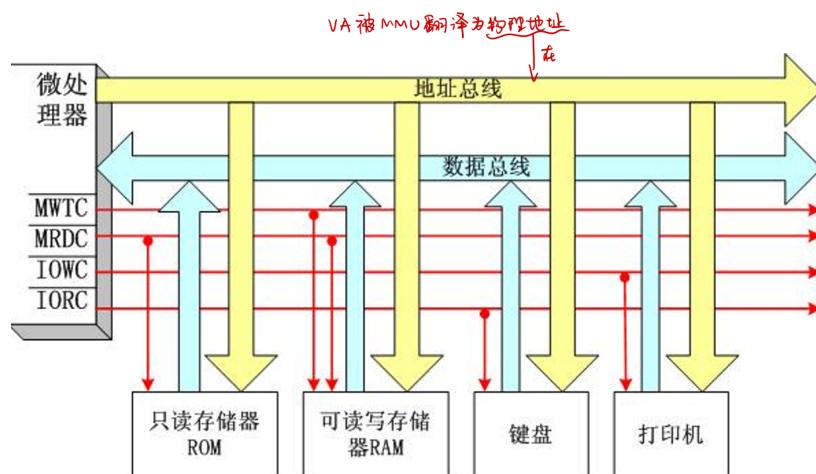


设备控制器/中间件（有芯片&寄存器）：地址转换、传输数据、执行命令、提升性能

- 中断向量表中一个控制器对应一个中断向量

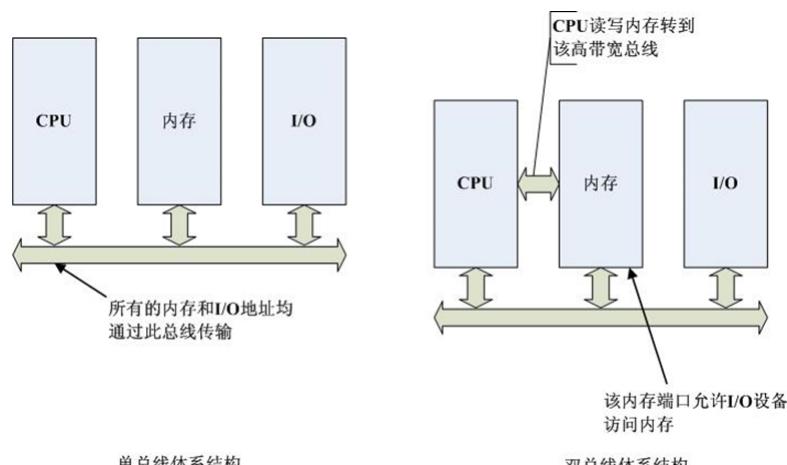
CPU和设备的通信机制：

- CPU设置状态和指令寄存器
- 设备执行指令和完成数据通信，完成任务后通过中断通知CPU。

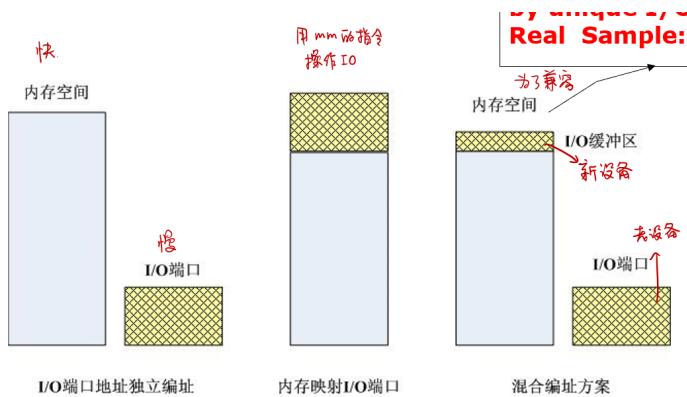


设备接入方式

- IO端口号：计算机维护一个IO端口号表。:(内存空间和设备寄存器分离。e.g. `inb 0x20, outw ...`
- **内存映射IO：所有设备寄存器被映射进内存空间。**
 - 地址格式统一、安全 e.g. `MOV [8000H]`
 - 管理代价高 e.g. 分线：CPU和内存通信要快，需要高带宽总线



- 独立编址、共享编址、混合式：

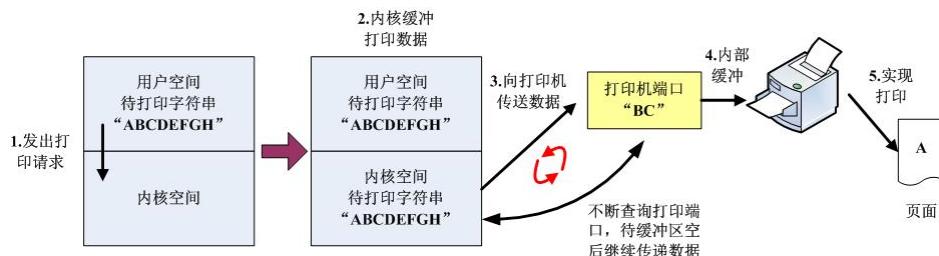


三种设备的通信模式（打印机）

下述用户进程均提交任务后阻塞

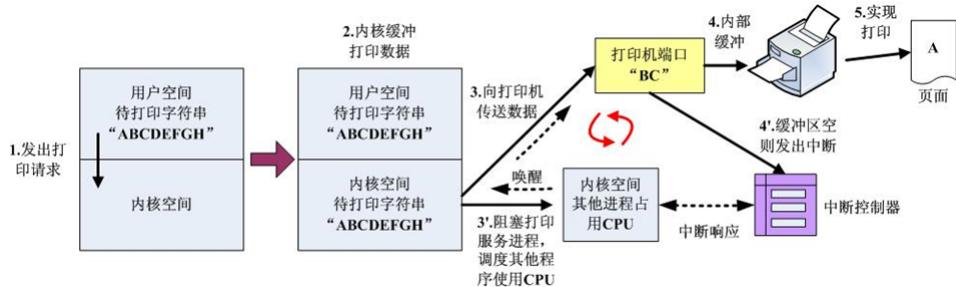
- 忙等待模式

特殊的内核进程给设备端口发送数据并且不断检查端口号是否可用并继续发送。用户进程在该内核进程执行完毕后继续执行。—— CPU被浪费 :(



- 中断模式

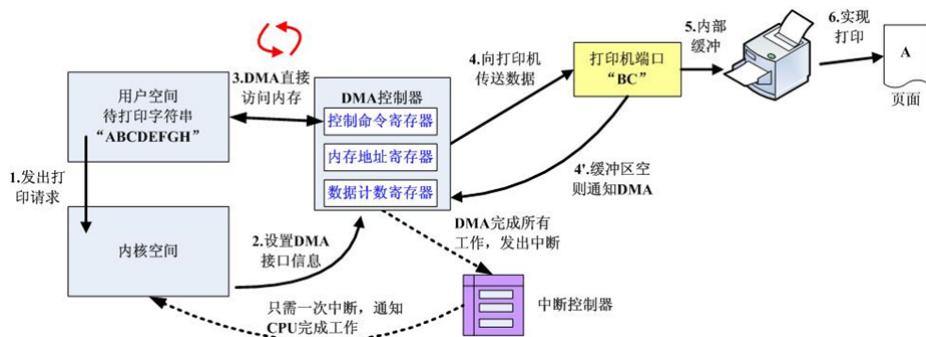
特殊的内核进程给设备端口发送数据，随后睡眠让出CPU等待设备数据缓冲区为空时被唤醒，继续发送。—— 过多中断耗时 :(



- DMA (Direct Memory Access) 模式

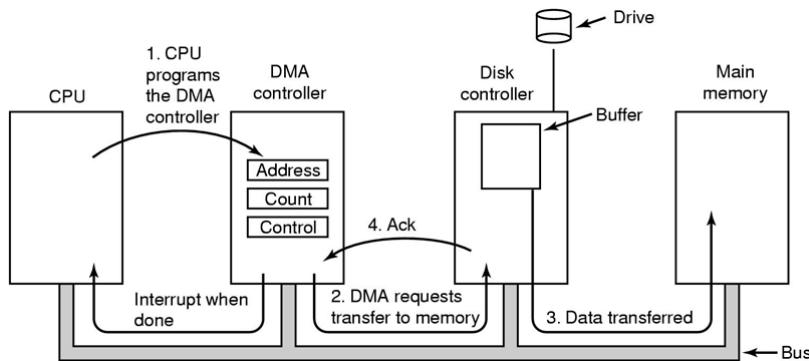
DMA: 外部设备不通过CPU而直接与内存交换数据的接口技术。这样数据的传送速度就取决于存储器和外设的工作速度。

用户进程产生trap，内核进程设置设备中的寄存器并退出。设备直接从内存取数，任务完成后对CPU产生中断唤醒用户进程。



三种模式

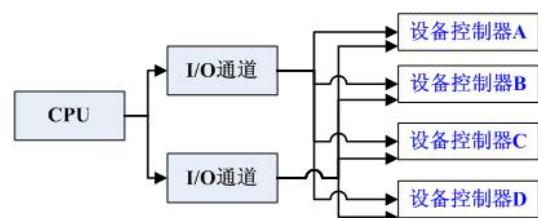
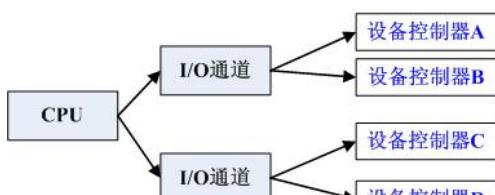
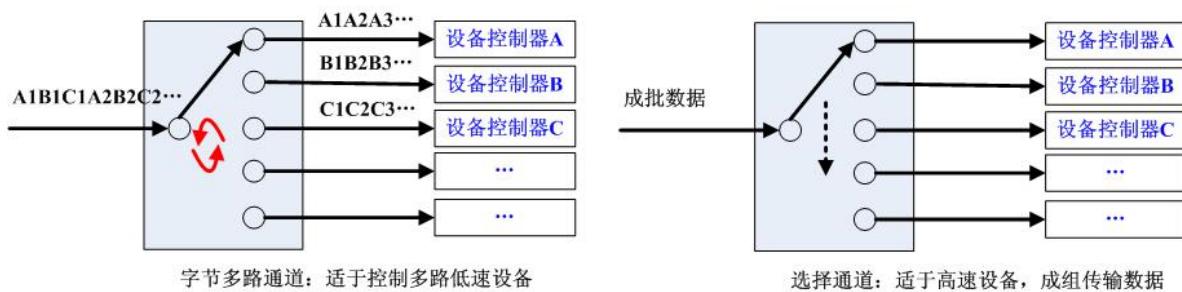
- 周期窃取: 向总线请求并获得一个字
- 突发模式: 获得总线, 开始传输一系列数据
- 飞跃模式: 告诉设备控制器直接给内存传数据
- 设备到设备&内存到内存: 从设备/内存获得一个字, 直接传送到目的地址



p.s. DMA 用什么地址由 MMU 的位置决定 (MMU在内存里就用VA)

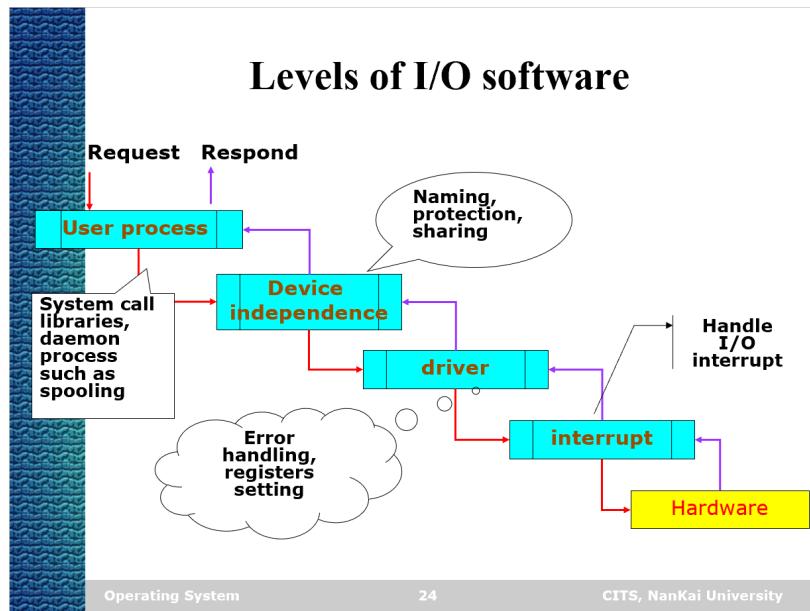
I/O通道

- 字节多路通道: 循环给每个通道发送一个字节, 每个通道传送给不同的设备控制器。适用于控制多路、低速设备。
- 选择通道: 选择一个通道传送一批数据。适用于高速设备, 成组传输。
- 单通道: 每个设备由一个通道负责。
- 多通道: 每个设备可由多个通道控制。



IO软件

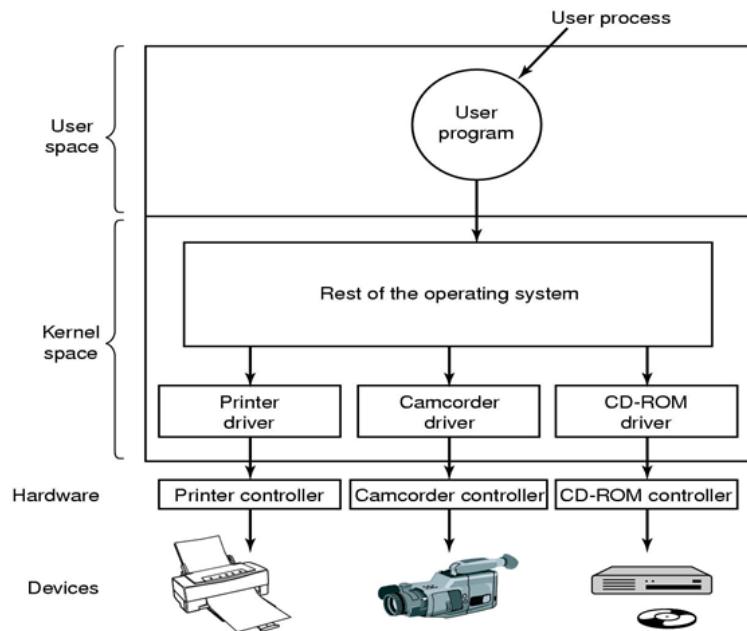
负责：同步阻塞/异步传输；错误处理（尽量底层）；统一命名（实现设备独立，做名字&地址映射）；缓存；分配/释放设备（共享&死锁问题）



中断形成后，软件做什么：

- 保存（没被硬件保存的所有）寄存器
- 给中断服务例程设置context（运行时环境）&栈
- 响应中断控制器，开放中断
- 寄存器复制到进程表
- 运行中断服务例程
- 为下个进程设置mmu
- 设置新的寄存器
- 开始新进程

驱动程序



- 在内核空间，被指明**逻辑位置**
- 和controller之间通过**总线通信**

是什么？

- 控制controller的**指令**
- 被当作“**内核进程**”
- OS给driver提供统一接口
- 静态/动态library

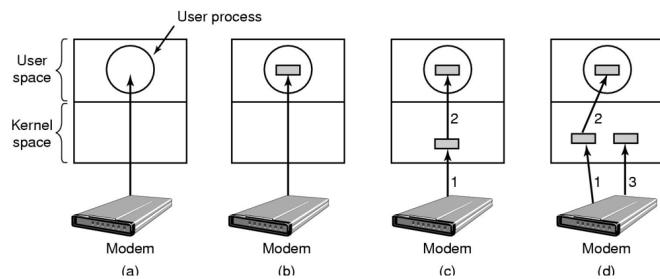
工作流？

- 接收&检查下来的参数
- 检查设备并启动
- 设置设备寄存器
- 接收IO操作

典型问题：

- 异步：中断
- 缓冲：

2) 引入缓冲的方式



- (a) 无缓冲输入
- (b) 缓冲区放在用户空间
- (c) 缓冲区在核心空间，然后被复制到用户空间
- (d) 在内核空间有双缓冲区

- Spooling: (见死锁)

磁盘管理

- RAID——CtrlF自取
- 磁臂管理——同上

死锁

- 定义：一个进程集合，其中，每个进程都在等待另一个进程产生某个事件发生。
- 条件【4】： (1) 资源互斥 (多进程不可共享 i.e. 不可抢占)； (2) 阻塞 (请求并保持)； (3) 资源不能被OS剥夺，只能被释放； (4) 环路等待序列
- 解决方法：预防、避免、检测和系统恢复、Ostrich (鸵鸟法：低概率—>ignore)

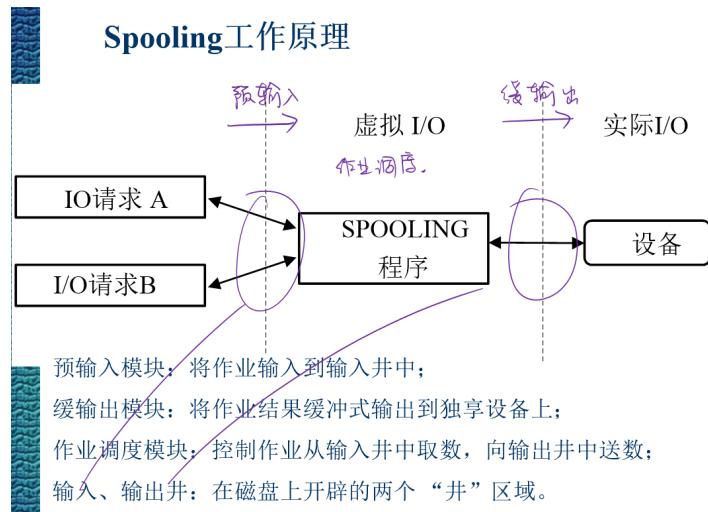
对比图！！【策略——模式——优点——缺点】

方法	资源分配策略	各种可能模式	主要优点	主要缺点
预防 Prevention	保守的；宁可资源闲置(从机制上使死锁条件不成立)	一次请求所有资源<条件 1>	适用于作突发式处理的进程；不必剥夺	效率低；进程初始化时间延长 剥夺次数过多；多次对资源重新启动 不便灵活申请新资源
		资源剥夺<条件 3>	适用于状态可以保存和恢复的资源	
		资源按序申请<条件 4>	可以在编译时(而不必在运行时)就进行检查	
避免 Avoidance	是“预防”和“检测”的折衷(在运行时判断是否可能死锁)	寻找可能的安全的运行顺序	不必进行剥夺	使用条件：必须知道将来的资源需求；进程可能会长时间阻塞
检测 Detection	宽松的；只要允许，就分配资源	定期检查死锁是否已经发生	不延长进程初始化时间；允许对死锁进行现场处理	通过剥夺解除死锁，造成损失
忽略 ignore	不理睬死锁问题，认为它不是主要问题(鸵鸟法)			

死锁预防：由OS为资源请求&分配 定义约束规则（规则可以破坏死锁条件）

- **SPOOLing (Simultaneous Peripheral Operations On-Line) 机制**

关于慢速字符设备如何与计算机主机交换信息的一种技术，通常称为“假脱机技术”，又称为“排队转储技术”。实现资源的虚拟共享。——破坏资源互斥条件。



- 分配预测：一次性分配，否则阻塞。——破坏阻塞条件。
- 资源剥夺：对死锁进程/状态可以保存和恢复的资源。——破坏资源不可剥夺条件。
- 资源排序：所有进程对资源统一按升/降序请求。——破坏环路等待条件。

死锁避免：由OS分析当前资源分配状态，试图安全分配

- “安全状态”：存在一个安全的分配序列
- 进程请求时，OS **检查分配方案是否安全**，仅在安全时分配。不安全时进程等待。

银行家算法：

- 记录：最大需求&当前分配状态
- 仿真：把资源分配给某个进程，检查所有进程可否运行到底
- **Attention:** 归还的资源是已分配矩阵中的，不是需求矩阵的！

死锁检测&恢复：一个特殊进程负责检测，杀死死锁进程

- 检测

- 方法：图/集合/向量/矩阵记录资源分配状态，试图找一个循环等待的状态
- 初始：unmark all，通过算法时mark
- 结束：存在unmarked ==> 有死锁
- **算法：**用**请求矩阵**和**剩余向量**找下一个可标记的进程，找到后从**分配矩阵**归还资源给**剩余向量**，mark进程。（全程不用动**请求矩阵**）
 - 初始：在**分配矩阵**中选一行，标记全0 (*i.e.* 不给它分配资源)
 - 初始化可用资源向量 W
 - 按index找，找到 $<= W$ ：标记行， $W +=$ 分配矩阵的index行。重复直到结束。

	R1	R2	R3	R4	R5	
P1	0	1	0	0	1	P1
P2	0	0	1	0	1	P2
P3	0	0	0	0	1	P3
P4	1	0	1	0	1	P4

Request Matrix Q

	R1	R2	R3	R4	R5	
P1	1	0	1	1	0	P1
P2	1	1	0	0	0	P2
P3	0	0	0	1	0	P3
P4	0	0	0	0	0	P4

Allocation Matrix A

Resource Vector

R1	R2	R3	R4	R5
2	1	1	2	1

$W = \text{Available Vector}$

- 标记P4
- 标记P3
- 更改 $W = 0\ 0\ 0\ 1\ 1$
- 无可标记进程-----说明有死锁

31

- 恢复策略

- 资源褫夺
- 后退：检查点
- 杀死进程

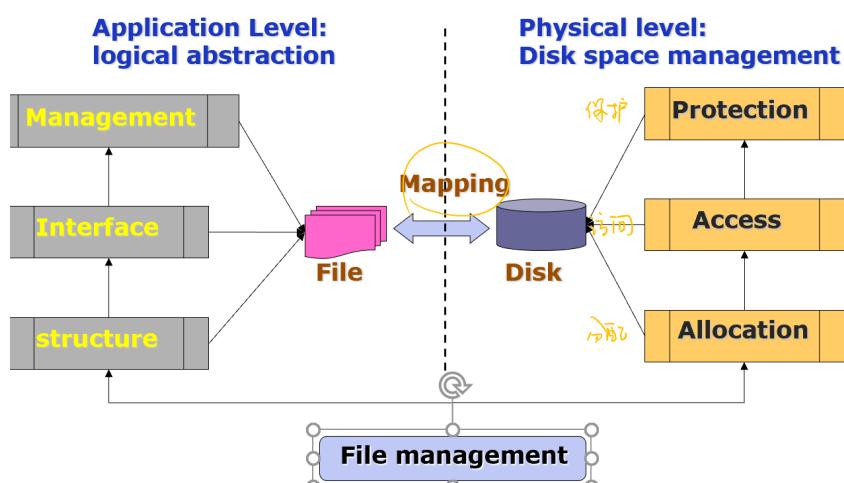
非资源死锁问题【4】

- **两阶段加锁：**进程对需要修改的数据先请求加锁再修改数据。若已被加锁，则释放此进程的所有加锁记录（“归还资源”）
- **通信死锁：**阻塞等待回复的情况下，若发送的信息丢失则死锁。解决：超时中断。
- **活锁：**没有死锁，但无法运行下去，只是不断尝试。e.g. 冲突检测
- **进程饿死：**条件限制使有些进程永远无法得到服务。

文件系统

文件管理：OS里负责数据存储、索引、保护、共享的模块

- “文件”(File)：是存储和IO设备的抽象，是存储信息和数据的物理结构。



- 面向用户：文件访问、目录管理、构成文件结构、访问控制、限额设定、审计

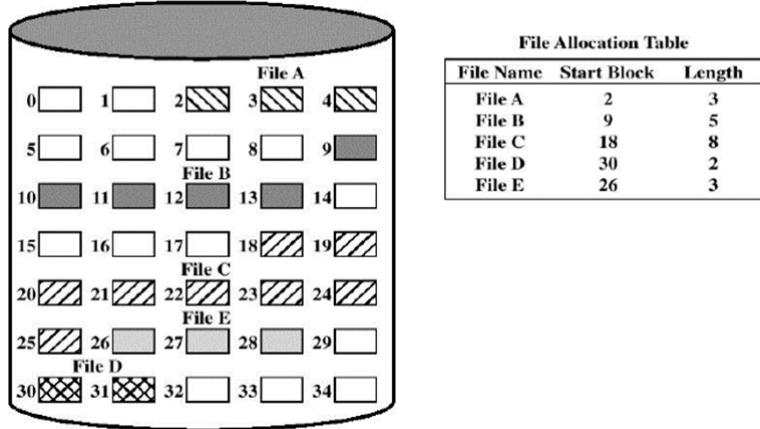
- 内部：分块存储、IO缓冲调度、文件定位、外存管理、外存访问控制

逻辑结构

- 字节流：难以进行快速的随机访问，每一次修改时都涉及到整个字节流的插入删除操作
- 块：能够进行快速的随机访问，但对记录进行增删改时，需要对整个记录序列进行处理。
- 树（数据库文件）：随机访问性能高于字节流式结构，低于记录式结构；但文件内容的修改效率极高。

文件结构

- 顺序分配：**简单；可扩展性差、外碎片化



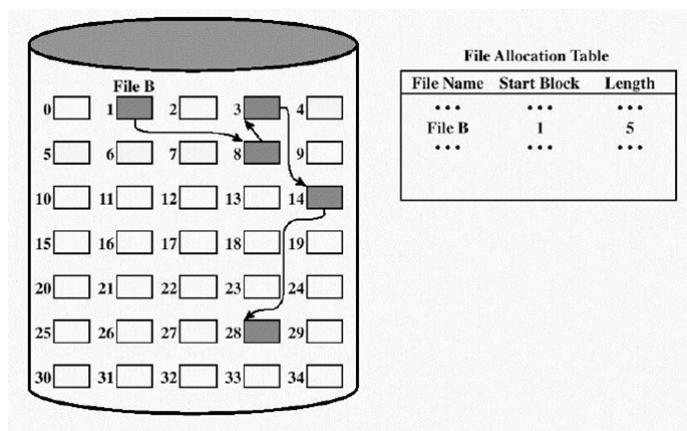
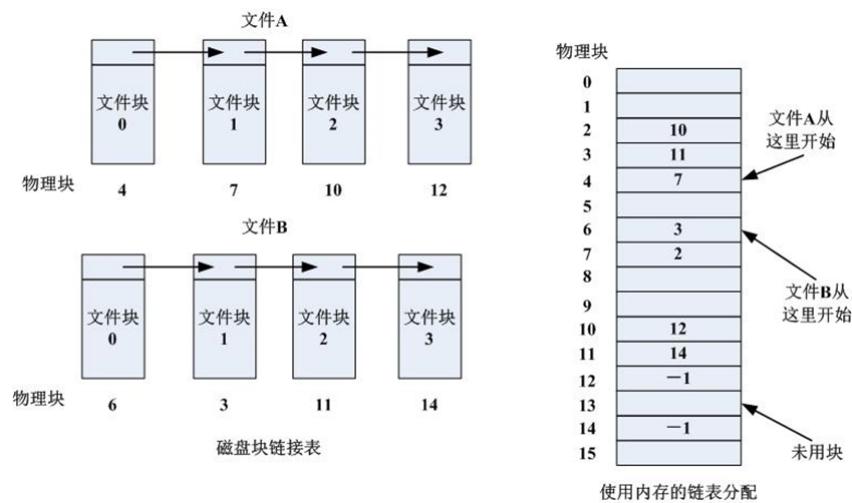
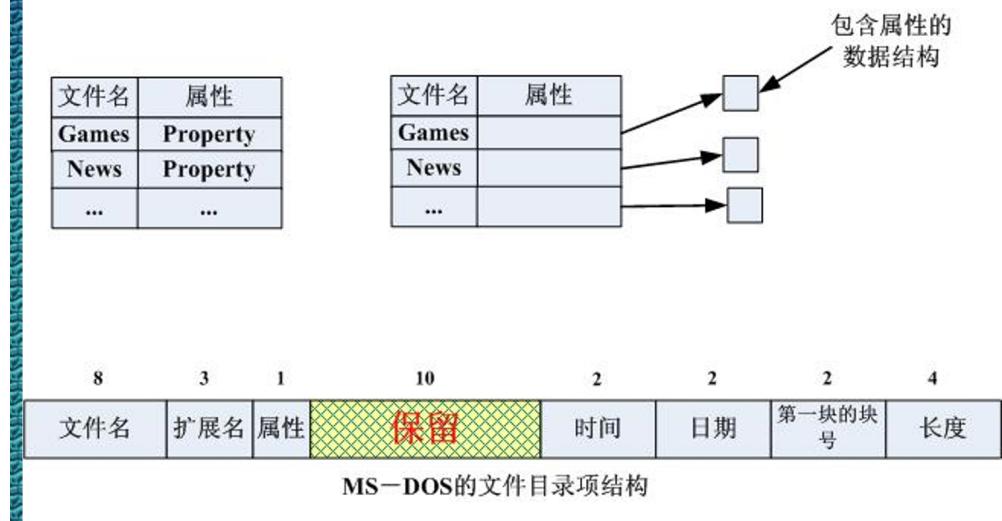
- 链式：**高效、好改；随机访问差、指针不安全

- 文件头包含第一块&最后一块指针



- : 创建、修改
- : 随机访问、指针安全（断一个全断）
- e.g. FAT (File Allocation Table)** in DOS: 一个文件的文件块串成链，在FAT里记录文件名对应的起始块号和链长

Case: FAT in DOS



■ :) 没有外碎片、修改、不需预先分配

- 索引分配 (e.g. inode) : 为每个文件创建一个索引数据块(I), 指向文件数据块的指针列表(1B), 文件头包含其指针(&I)



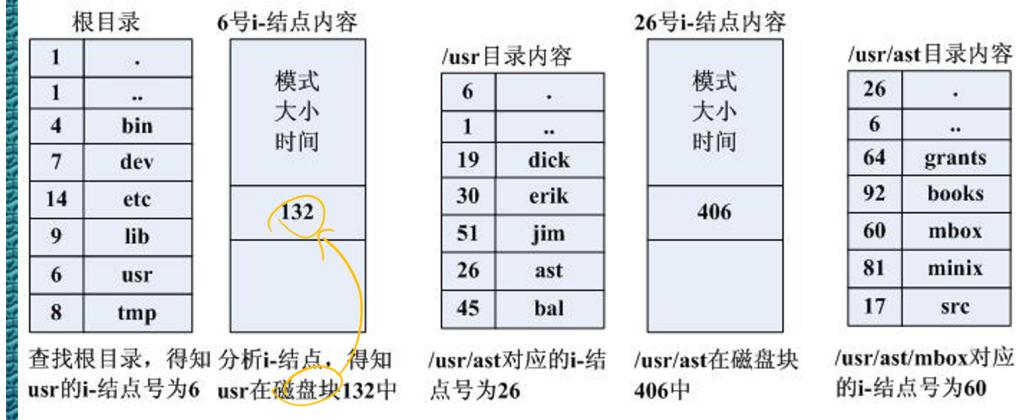
Case: I-Node in Unix

2 14

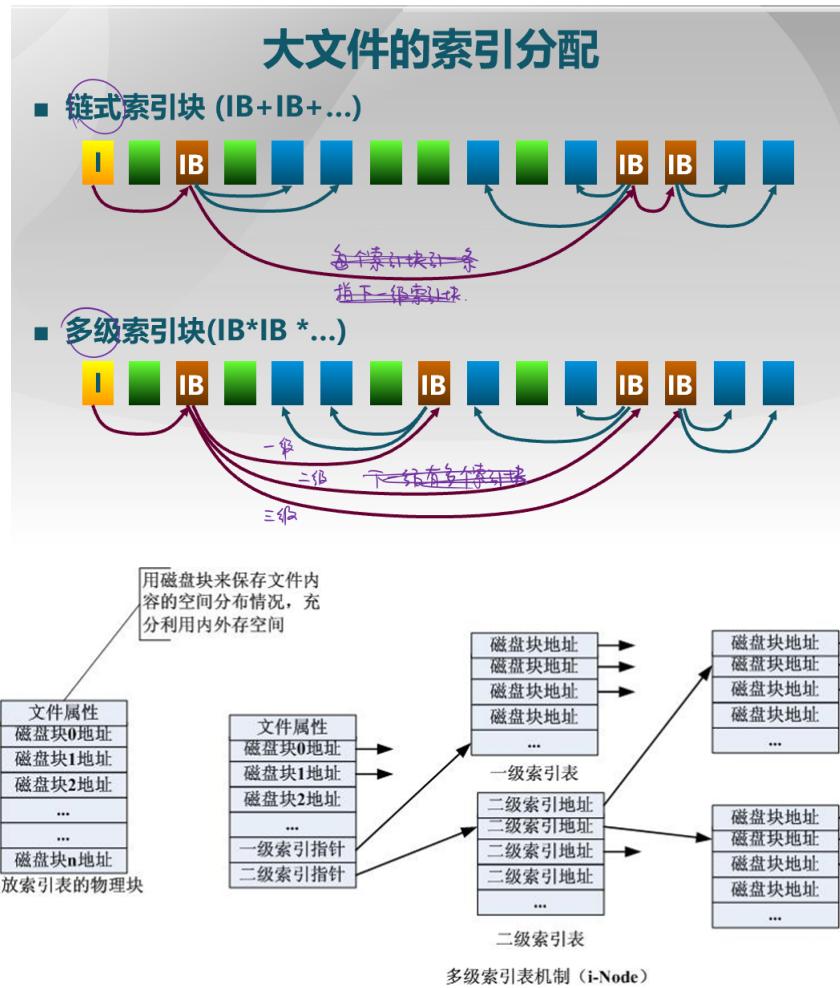
Unix中的目录项结构 [i-结点号 | 文件名]

查找/usr/ast/mbox

的过程说明



- (): 创建、修改、没有碎片、直接访问
- (文件很小时存储索引的开销大)
- 大文件——多级索引



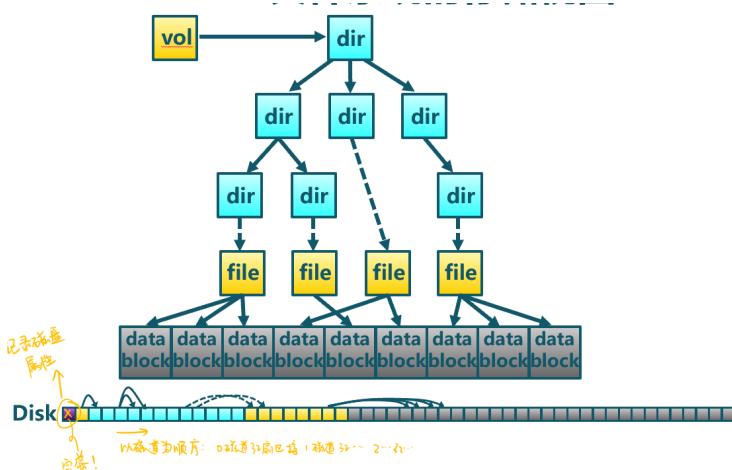
文件结构对比!

		Continuous	Link table	Index table
media	tape	Supported	Unsupported	unsupported
	disk	Supported	supported	supported
Access mode		Sequential & random	sequential	Sequential & random
Efficiency		Low	Middle	High
Application		To simple to be used	Not popular	Widely used

分层文件系统：文件以目录方式组织

“目录”：一类特殊的文件（文件索引表），表项为<文件名，指针>

- 性能：路径、文件名长度&格式、DS优化、快速索引（哈希缓存）
- 实现：线性列表+指针（简单；耗时）；哈希表（省时；冲突、大小固定）



存储结构

- 卷控制块 (vol/superblock, 每个FS一个)：FS挂载时load；记录块大小、空闲块、计数/指针
- 文件控制块 (file) e.g. inode：记录权限、拥有者、大小、位置
- 目录节点 (dir)：指向文件控制块、父&子目录

- 内核跟踪进程所有的打开文件 —— OS为每个进程维护一个**打开文件表**，文件描述符是打开文件的标识。
- 进程可以共享系统级打开文件表（唯一），从而共享缓存

文件描述符

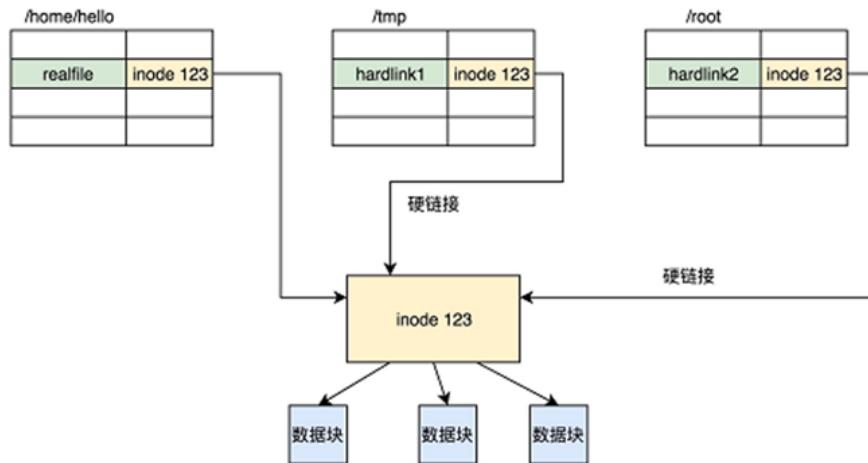
- 是OS在**打开文件表**中维护的**打开文件状态和信息**。
- 包括：文件指针、打开计数、磁盘位置、访问权限

打开文件锁：协调多进程文件访问

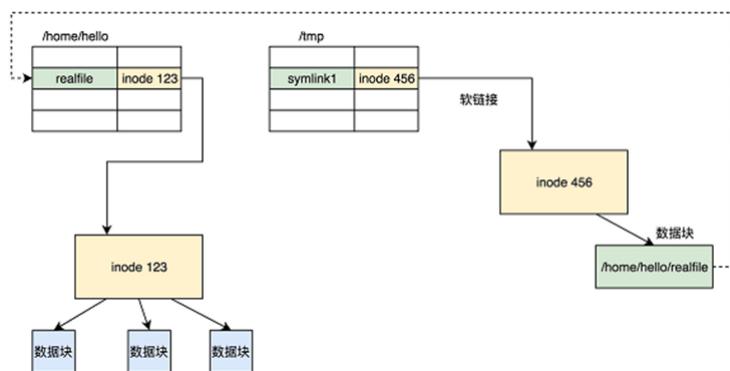
- “强制”：FS根据锁情况和访问需求确定是否**拒绝访问**
- “劝告”：**进程**可以查找锁的状态**决定**如何做

文件别名：多个文件名关联同一个文件

- 硬链接：**多个文件项**指向一个文件**

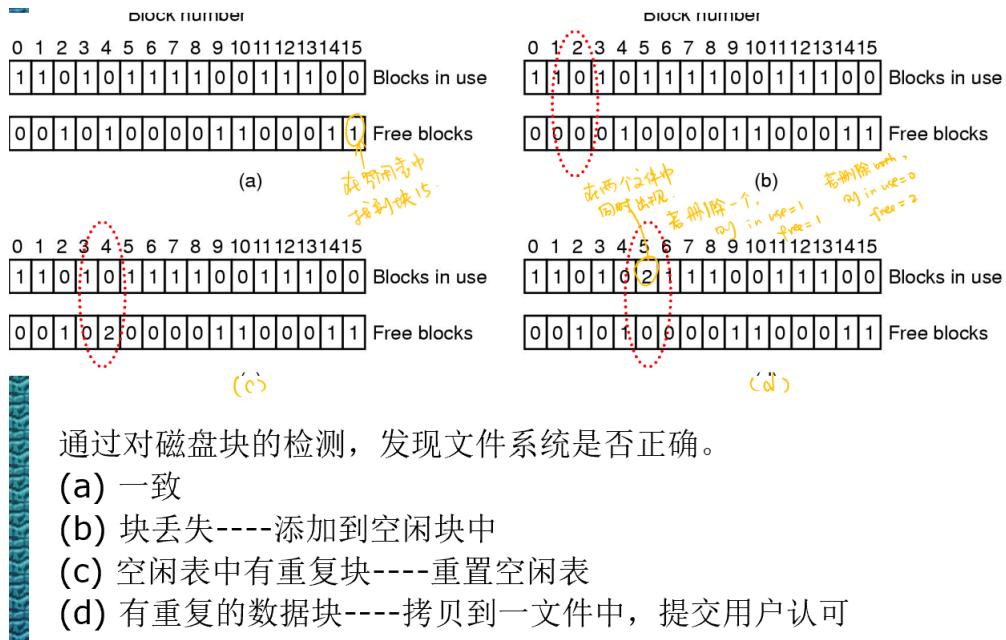


- 软链接：**以“快捷方式”指向文件，通过**存储真实文件的逻辑名称**实现
- 符号链接：**本质是**访问另一个文件**
 - 只有文件属主有指向i节点的指针
 - 建立**新文件类型**link用作**目录管理**
 - 有共享需求的用户只了解到共享文件的目录，而无指向i的指针
 - 删除符号链接不影响原文件
 - 属主可以删除被链接文件



PWD：当前工作目录。每个进程都会指向一个文件目录用于**解析文件名**。

一致性检查

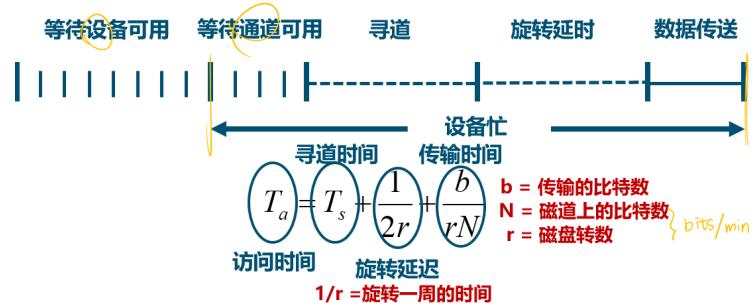


通过对磁盘块的检测，发现文件系统是否正确。

- (a) 一致
- (b) 块丢失----添加到空闲块中
- (c) 空闲表中有重复块----重置空闲表
- (d) 有重复的数据块----拷贝到一文件中，提交用户认可

磁盘访问时间！

- 减少磁臂移动 e.g. 多个相邻块为一个分配单位；从磁盘中部开始分配（减少寻道时间）



- 寻道时间：定位磁道（最耗时）
- 旋转延迟：从0扇区开始；平均为旋转周期/2；旋转周期=1/r
- 传输时间
- 增加磁盘个数
- cache
- 块预读

访问权限

- *read/write/delete*
- *execute*: 可由OS读出文件内容，作为代码执行
- *change protection*: 修改属主/访问权限

用户范围：指定用户/用户组/任意用户

建立矩阵

VFS——虚拟文件系统：解决FS的识别问题，对不同FS抽象

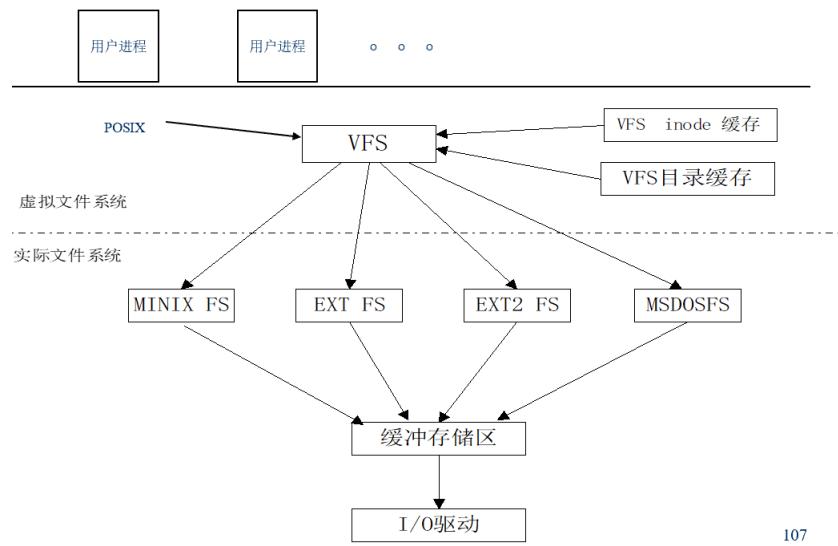
- 在内存建立一个解释FS的抽象软件 VFS
- 用VFS建立物理设备与FS服务的接口

- VFS对每个FS细节抽象，使不同FS在系统内部被**管理进程**看成相似的FS
- 系统启动时建立，关闭时消失

功能：

- 记录（可用FS）类型
- 建立（设备&FS）关联
- 提供（面向文件级的）通用性操作
- 把特定FS的操作**映射**到物理FS中
- 高效查询

V F S 与实际文件系统的逻辑关系



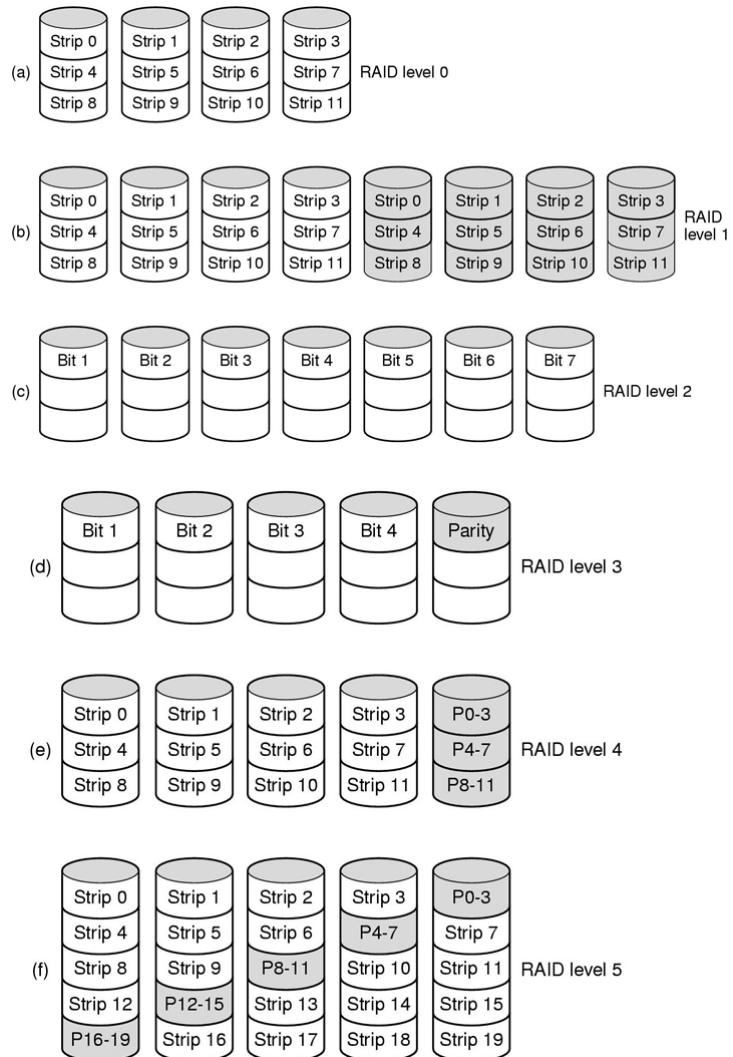
107

磁盘调度算法：优化磁盘访问请求顺序【7】

- FIFO
- SSTF（最短服务时间优先）：选从当前出发移动最少的
- SCAN（电梯算法）：一个方向，到头
- C-SCAN（循环扫描）：一个方向 e.g. 左边出去右边回来
- C-LOOK：一个方向，不到头
- N-step-SCAN：分子队列，队列间FIFO，队列内扫描
- FSCAN（双队列）

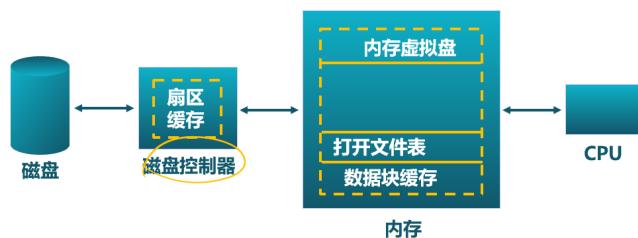
RAID——Redundant Array Of Inexpensive Disks

- 目标
 - 通过——并行组件（磁盘驱动器）
 - 获得——额外**磁盘访问性能**
 - 实现——独立**IO并行处理**



- 共同点
 - 是一组物理磁盘驱动器，OS将它们看作一个整体
 - 数据分布在物理驱动器阵列中
 - 用冗余的磁盘保存奇偶校验信息，用于数据恢复
- 分层特性
 - RAID0没有冗余数据提高性能，只是放在磁盘阵列中

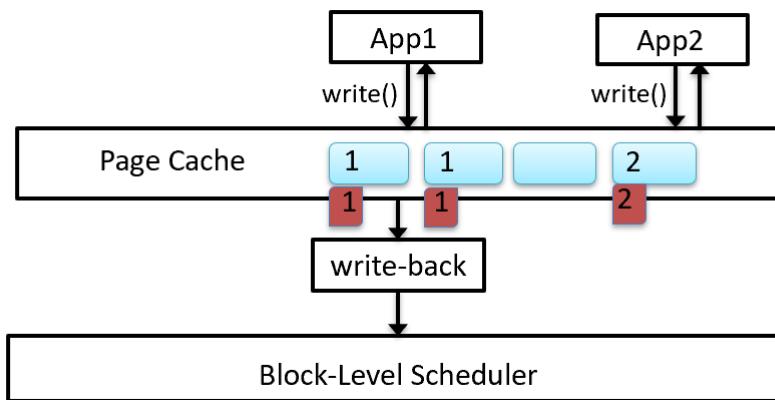
磁盘缓存



数据块缓存（用后/预读/延迟写出）

页缓存：统一缓存数据块和内存页。需要置换算法协调

标签 (dirty)：跨层次追踪 IO请求，识别操作的app&进程。



内存管理

是否真的掌握进程的内存布局?

4G虚拟内存空间



```
#include <stdio.h> #include <stdlib.h>
int max_length = 128; bool isDebug;
char * generate(int length){ 静态和动态链接库的区别?
    int i;
    char * buffer = (char*) malloc (length+1);
    if (buffer == NULL) return NULL;
    for (i=0; i<length; i++) buffer[i]=rand()%26+'a';
    buffer[length]='\0'; 用户态只能用3G内存, 内核态1G内存
    return buffer; Lib在windows里属于静态文件, 默认属于自己写的代码,
} 加载到代码段, 动态库 (.dll/.so) 是运行时自动加载的,
int main(int argc, char *argv[]){
    int num; char * buffer;
    printf ("Input the string length : "); .data—int; .bss—bool; .rodata—数组字符串
    scanf ("%d", &num); 数据段 (.data, .bss, .rodata)
    if(num > max_length) num = max_length; 栈区、堆区 new malloc放在堆区
    buffer = generate(num); 内核态使用的不是物理地址,
    printf ("Random string is: %s\n", buffer); 进程的内核态? 也是虚拟地址
    free (buffer); 内核态使用的是物理地址?
    return 0; malloc动态分配内存, 使用物理资源, 内核态代码存在哪?
}
```

保护机制 (e.g., 空指针)

0x00000000 0xffffffff

内核态内存
堆栈分界线?
动态链接库
用户态内存

虚拟内存：进程的内存布局

虚拟地址从低到高：(32位操作系统) 用户态3G, 内核态1G

- 寻址空间：32位/64位除去低12位偏移量，高20/48位得到 $2^{20}/2^{48}$ 大小的寻址空间

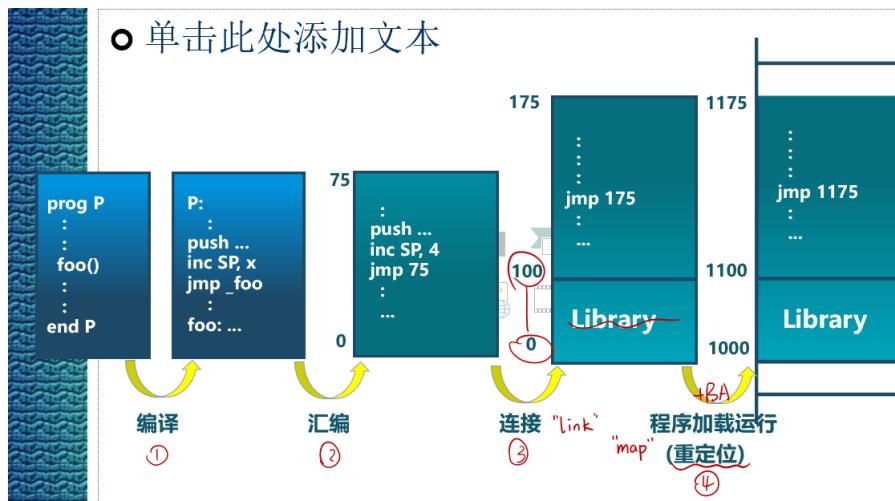
- 地址空间：

CPU中的PC寄存器，存储下一条指令的地址

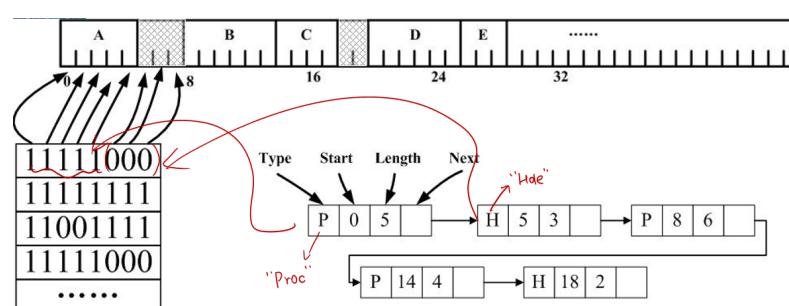
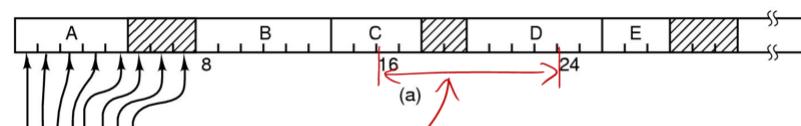
线性地址空间：数据按序排列

- “内存管理”
 - 物理&逻辑地址映射
 - Pakinson's law: 程序扩展以填充内存
 - 特点: 效率 (更多可用空间)、合理性 (分配、重定位、保护)、方便 (加载、动态调整)
 - 类型: 简单 (静态&固定)、复杂 (swapping&paging)
 - 目标: 分配&追踪、地址转换、共享&保护、空间扩展
- 单道编程: 内存中只有一个程序
- 多道编程: 把内存分为多个**固定分区**
 - 地址转换&内存保护方案:
 - 加载进内存时修改指令
 - 给每个内存块设置PWD(当前工作目录), 存入PSW
 - 基址&界限寄存器

compile>>assemble>>LA>>link(.lib)>>relocate(+BA)>>PA



位图&链表——管理内存空洞



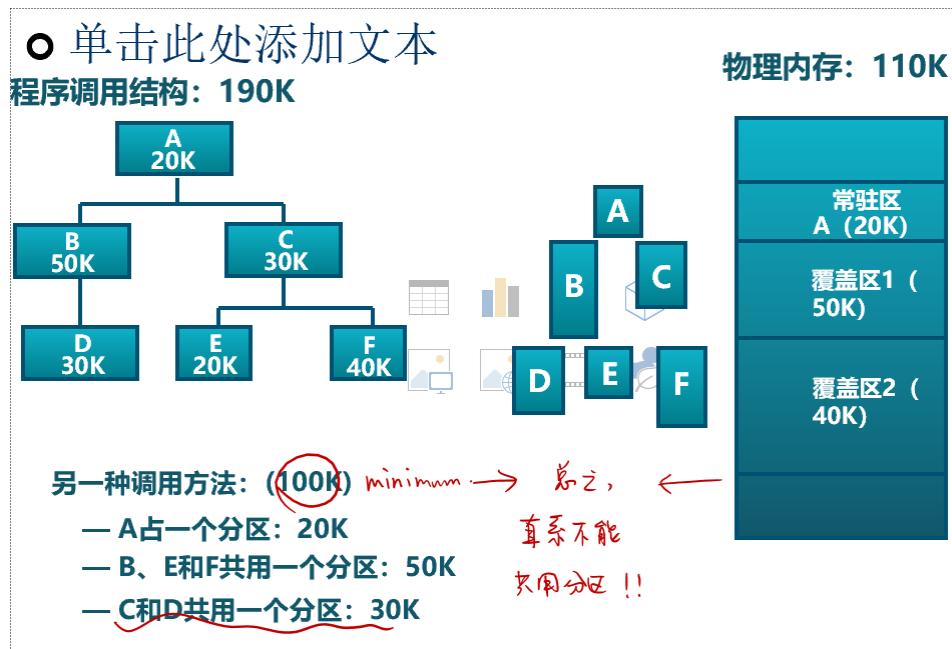
分区分配算法

- 最先匹配 (First-Fit)
 - 分配&释放时间性能，内存高端交大
 - 低端碎片，查找开销大
- 下次匹配 (Next-Fit)
 - 时间性能，分布均匀
 - 碎片
- 最佳匹配 (Best-Fit) : 小到大，第一个
 - 个别外碎片小，保留大分区
 - 整体外碎片多
- 最坏匹配 (Worst-Fit) : 大到小，第一个
 - 小分区少
 - 大分区也少

碎片整理

- 紧凑 (条件：所有应用程序可动态重定位)
- 分区对换 (swap) : 抢占并回收处于等待状态进程的分区

覆盖技术



- 方法：划分模块功能>>覆盖关系>>执行前预先加载&交换
- drawbacks:
 - 难以划分
 - 执行路径难以预判
 - hard to program
- 虚拟内存：机器自动完成覆盖&交换

虚拟存储

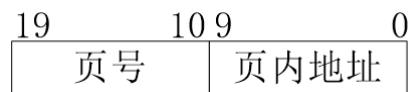
- 特征：
 - 不连续性 (物理内存分配&虚拟地址)
 - 虚拟内存 > 物理内存
 - 部分交换 (虚拟存储只对部分虚拟地址空间进行换入换出)
- 进程的各段不必完全装入内存就可以启动运行
- 定时换出、按需换入
- 方法：分页&分段

内存管理单元——MMU (Memory Management Unit)

- 一个硬件设备
- 将虚拟页 (page) 映射为物理页 (frame)
- 存储分段的段寄存器
- 存储分页的：
 - 页表寄存器：指PT物理地址，叠加页号找到对应Page；页表项总数：防止越界
 - 高速缓冲存储器 (TLB)
- 进程被激活时用页表配置MMU

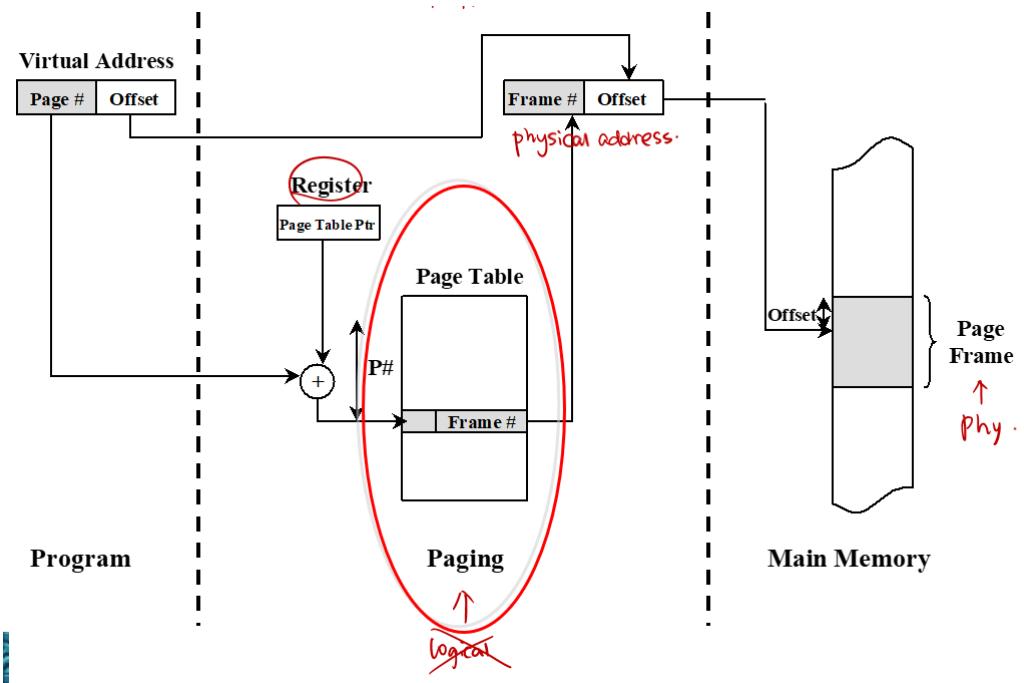
分页

- 内存以字节编址
- Page (虚拟页) —— 描述进程逻辑空间 vs. Frame (物理页) —— 描述内存物理空间
- 大小上：Page size == Frame size == 磁盘块大小 == 4KB
- 页表用作 Page <==> Frame 映射
- 分页后进程的逻辑地址 = 页号 | 页内地址 (页偏移)
e.g. 10位页偏移 —— 页长 (Page size) 1K; 10位页号 —— 逻辑地址共 1024 页



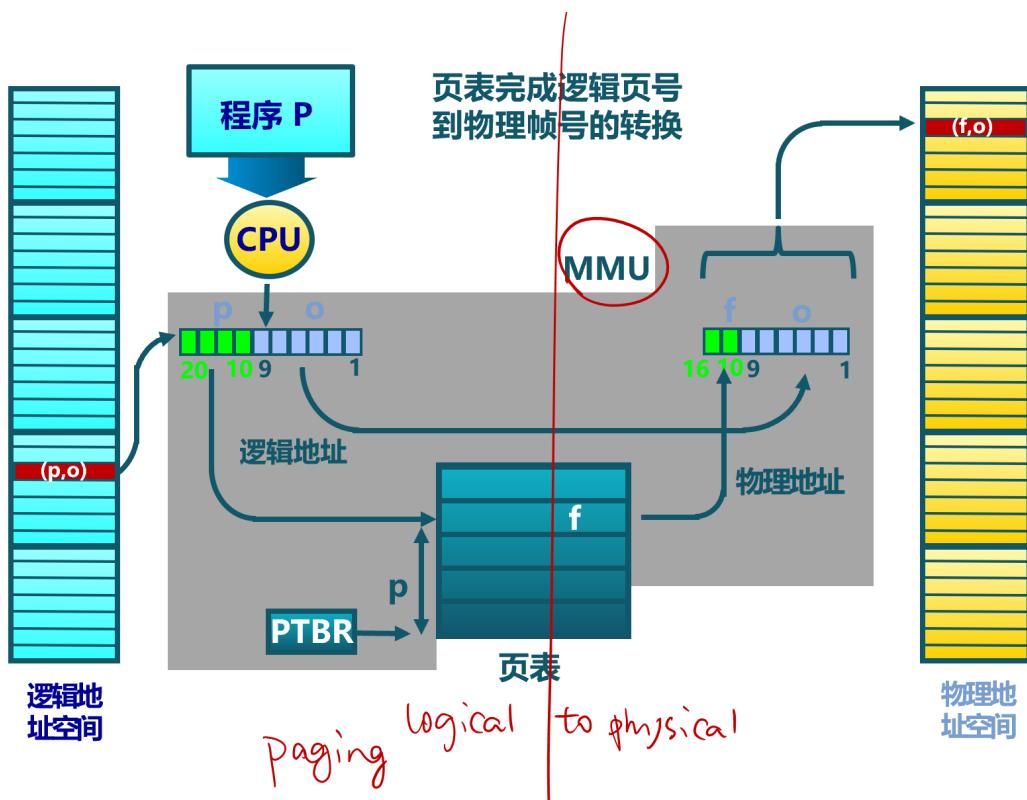
如果：虚拟内存地址为32位，则VA=20位页号+12位页偏移 (20位页号—— 2^{20} 个页表项——1M; 12位页偏移——Page size=4KB ==> i.e. 需要12位偏移描述全部内容)

- 一个页表项至少 4B：页号 3×8 bits > 20 bits > 2×8 bits，以字节寻址，页表项至少需要装下 3×8 bits = 4B 的页号。
- 分配时，进程在一个Frame内是连续的，Frame间可以不连续



页表：以页号+PTE为索引&Frame (的物理) 地址为内容

- 为了索引性能而用数组存储
- drawbacks:
 - 查页表要访存！！！ i.e. 需要两次访存！！ ==> 所以用了TLB
 - 页表太大=>内存利用率降低 ==> 所以有了多级页表



多级页表——解决页表太大导致内存利用率低

- 一级页表的物理地址存在 cr3 里，可以直接载入
- 二级页表可以不存在/不在主存中

一个40MB的程序

就是说在磁盘里有40MB
磁盘里一个frame是4KB
一共有10K个frame
磁盘里frame和内存里page一样大

需要10K个page

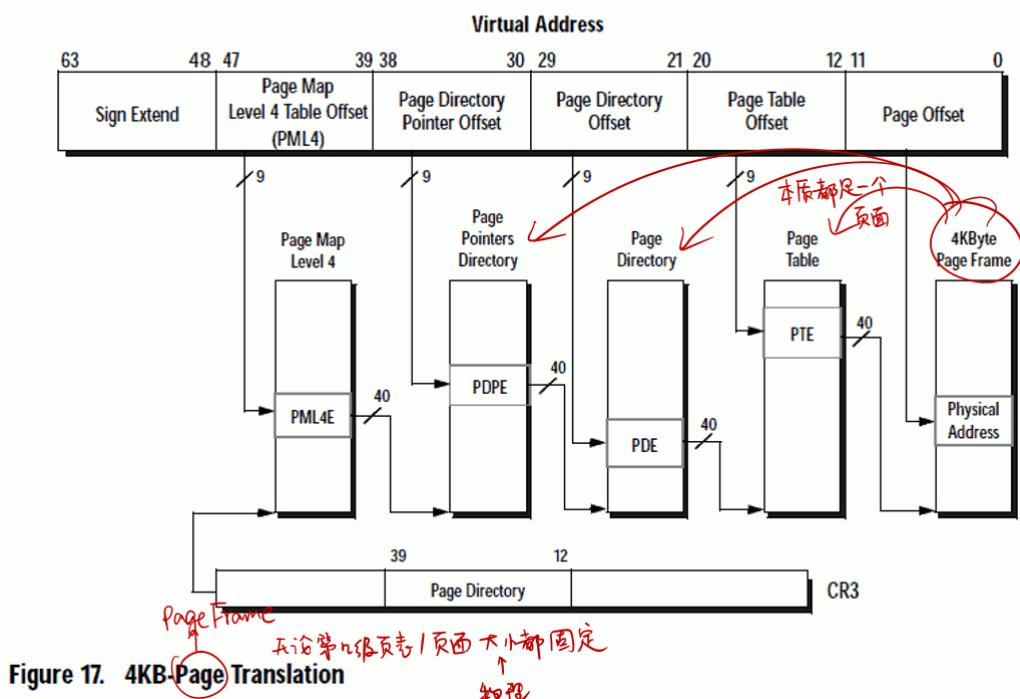
一个页表项4B

整个页表就是40KB
把页表转换成物理页的个数，就是10个物理页

但是由于页号有20位也就是 $2^{20}=4M$ 个页表项
整个页表大小4MB

对一个40MB的程序，只需要40KB的页表，却由于虚拟地址格式限定分配了4MB的页表

X64的页表是什么样的

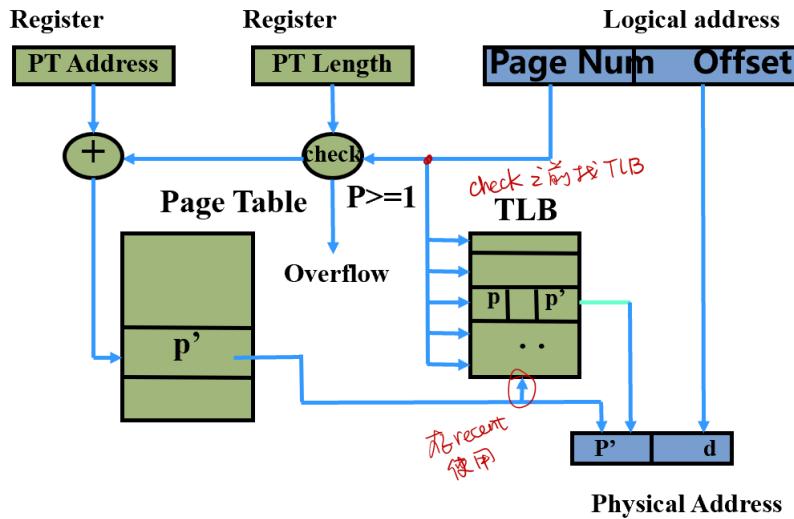


X64的页表是什么样的

- 每个页表里面有512项
 - $4k/8\text{字节}(64\text{bit}) = 512$
 - 因此用于页表项的寻址部分仅有9位
- 类似的，页目录项也只有512项
 - 页目录项也只有9位
- 页表扩展为4级，每级都是9位地址
 - 寻址空间最大为256T
 - 最高的16位目前没有使用
- 实际使用的内存远没有这么多

除[63...48]，地址低48位构成 $2^{48} = 256$ T的寻址空间

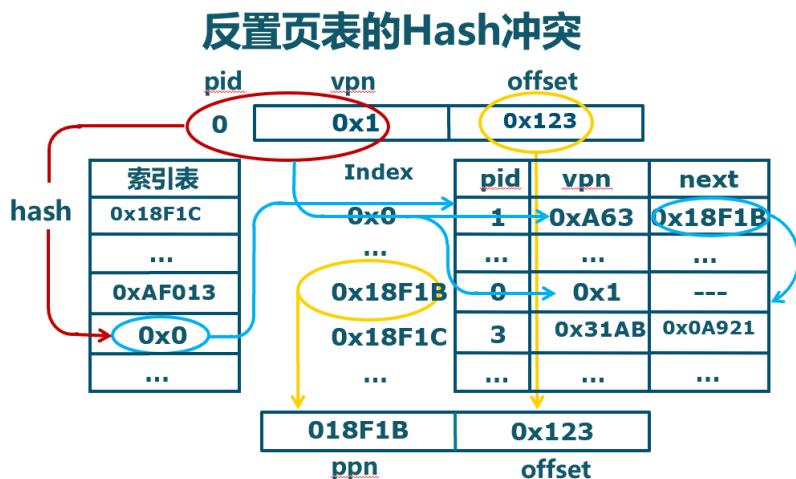
TLB (Translation Look-aside Buffer)——减少二次访存



- 是硬件技术，存在MMU里，用“关联存储” ($\langle \text{key}, \text{value} \rangle$)
- 保存近期使用过的映射关系（页表项）
- 功耗up！

反置页表：页表项以Frame号为索引，Page号为内容

- 节约内存，搜索差
- 全局唯一
- 正常的页表有 2^{52} 项 (4KB)，反置页表只有 2^{16} 项 (hash)，每一项包含一个链表，是引用这个Frame的一串Page
- 用hash处理虚拟地址，得到索引结果（物理地址）。用索引结果和进程id确认反置页表里的虚拟地址是否正确，如果正确，这个索引结果就是物理地址（高位）；如果虚拟地址不match，就用next里的作为索引继续去确认。直到虚拟地址match，当时使用的索引值就是物理地址（高位）。



分页系统工作流程

- Step1：全局页表初始化（创建+填页号）

- Step2: 创建进程
 - 创建进程的页表
 - 填充页号
- Step3: 取第一条指令
 - 缺页中断
 - 加载
 - 更新全局&进程页表
 - 把页表项存到TLB
- Step4: 页替换
 - 缺页中断
 - 选页写出到磁盘
 - 加载
 - 更新页表&TLB

缺页中断

情景

- 访问权限=>终止进程
- 未加载进内存=>换入
- 对bss段修改=>分配写值

.bss 段是存放程序中未初始化&默认初始化为0的全局变量的内存区域。属于静态内存分配。不占用程序空间，但占用运行时内存。

- COW=>拷贝更新

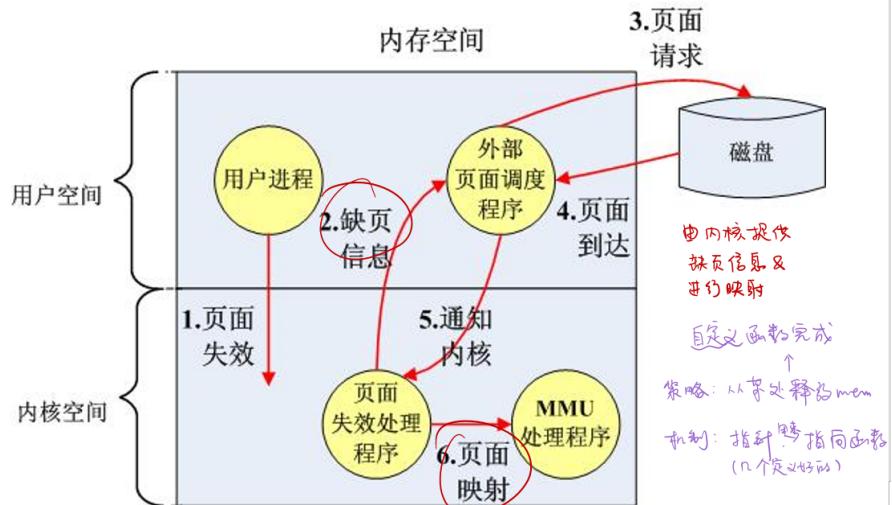
页式存储管理性能度量——EAT (Effective Memory Access Time) : 有效存储访问时间

$$EAT = \text{访存时间} * (1 - p) + \text{缺页异常处理时间} * p * (1 + q)$$

p: 缺页率; q: 页修改率; 缺页异常处理时间: 磁盘访问时间

页置换算法

Page replacement in user space



- 分类:
 - 局部 (*i.e.* 当前进程) : OPT、FIFO、LRU、Clock、LFU、NFU、NRU
 - 全局: 工作集、缺页率算法
- 最优 (OPT) : 未来最长时间不访问
- FIFO (没有second chance, 根据第一次进来的时间排序)
 - 简单
 - 性能; *Belady*

Belady 现象: 分配的物理页面数增加, 缺页次数反而升高

原因: FIFO的置换特征与进程访问内存的动态特征矛盾。被置换出的页面不一定是近期不会访问的

- LRU——Second chance: 换最早上次被引用的页【时间】
 - 实现:
 - 链表: 访问时页面移到头, 缺页时置换尾
 - 栈: 访问时压栈&抽出相同页号, 缺页时置换栈底
 - 特征: 开销大
 - 矩阵: 访问时先行置1后列置0, 缺页时置换最小二进制数值的行
- LFU: 换最少访问次数的页【次数】
 - 特征:
 - 开销大
 - 开始频繁使用, 后续不使用的难置换
 - Solution: 计数定期右移
- NRU: 换最近未使用=>最小RM
 - R——read: 每次时钟中断时重置
 - M——written
- Clock (AD) : 在second chance基础上循环链表

原始的方案: 在寻找页面是否存在的时候不改变驻留位。页面存在时, 驻留位置1; 不存在时, 指针开始移动, 判断指向位置驻留位是否为0: 为0则替换, 指针停止; 不为0则指针离开后清零。

时钟页面置换示例

时间	0	1	2	3	4	5	6	7	8	9	10
访问请求	c	a	d	b	e	b	a	b	c	d	
物理帧号	0	a	a	a	a	e	e	e	e	d	
物理帧号	1	b	b	b	b	b	b	b	b	b	
物理帧号	2	c	c	c	c	c	a	a	a	a	
物理帧号	3	d	d	d	d	d	d	d	c	c	
缺页状态					●		●		●	●	
驻留页面的页表项	0 a	0 a	1 a	1 a	1 a	1 e	1 e	1 e	1 e	1 d	
驻留页面的页表项	0 b	0 b	0 b	0 b	1 b	0 b	1 b	0 b	1 b	0 b	
驻留页面的页表项	0 c	1 c	1 c	1 c	1 c	0 c	0 c	1 a	1 a	0 a	
驻留页面的页表项	0 d	0 d	0 d	1 d	1 d	0 d	0 d	0 d	1 c	0 c	

改进的方案：减少修改页的缺页处理开销，缺页时写回并跳过有修改的页面。（一轮没找到可替换的继续循环直到成功替换）

case AD:

- 00: 换出
- 01: 写出, 置00
- 10: 置00
- 11: 置01

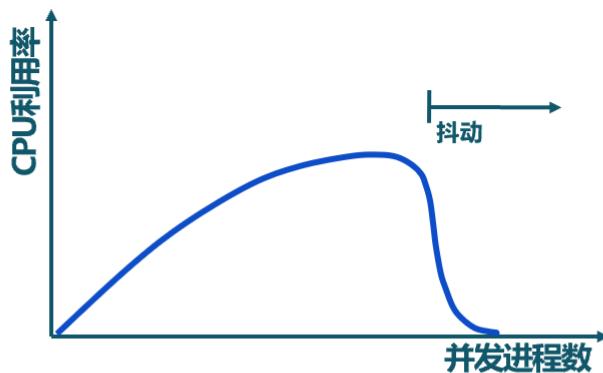
改进的Clock算法

时间	0	1	2	3	4	5	6	7	8	9	10
访问请求	c	a ^w	d	b ^w	e	b	a ^w	b	c	d	
物理帧号	0 1 2 3	a b c d	a b c d	a b c d	a b c d	a b e d	a b e d	a b e d	a b e d	a b e d	a b e d
缺页状态						●			●	●	

驻留页面的页表项	00 a	00 a	11 a	11 a	11 a	00 a	00 a	11 a	11 a	11 a	00 a*
	00 b	00 b	00 b	00 b	11 b	00 b	10 d				
	00 c	10 e	00 e								
	00 d	00 d	00 d	10 d	10 d	00 d	00 d	00 d	00 d	10 c	00 c

- NFU: 最不常用
 - 每个页面维持一个计数器，每次时钟中断扫描所有页面，把计数器加上当前的R位更新，计数器的值反映被访问的频繁程度。

抖动——CPU利用率&进程并发数的关系



■ CPU利用率与并发进程数存在相互促进和制约的关系

- ▣ 进程数少时，提高并发进程数，可提高CPU利用率
- ▣ 并发进程导致内存访问增加
- ▣ 并发进程的内存访问会降低了访存的局部性特征
- ▣ 局部性特征的下降会导致缺页率上升和CPU利用率下降

- 物理页面太少，不能包含工作集 >> 大量缺页，频繁置换 >> 进程运行速度慢

- 原因：内存中进程数目增加，每个分到的页面数减少，缺页率上升

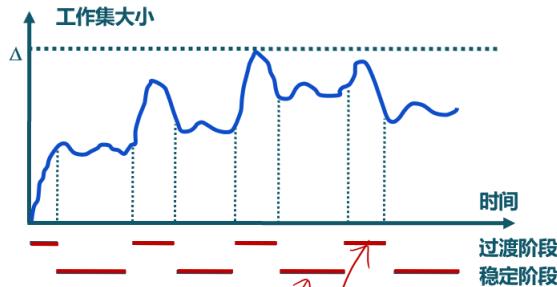
负载控制：调节进程并发数 (MPL)

工作集算法

工作集：一个进程当前正在使用的逻辑页面集合

- $W(t, \Delta)$ ：在当前 t 时刻以前 Δ 时间内被访问过的页面组成的集合

工作集的变化



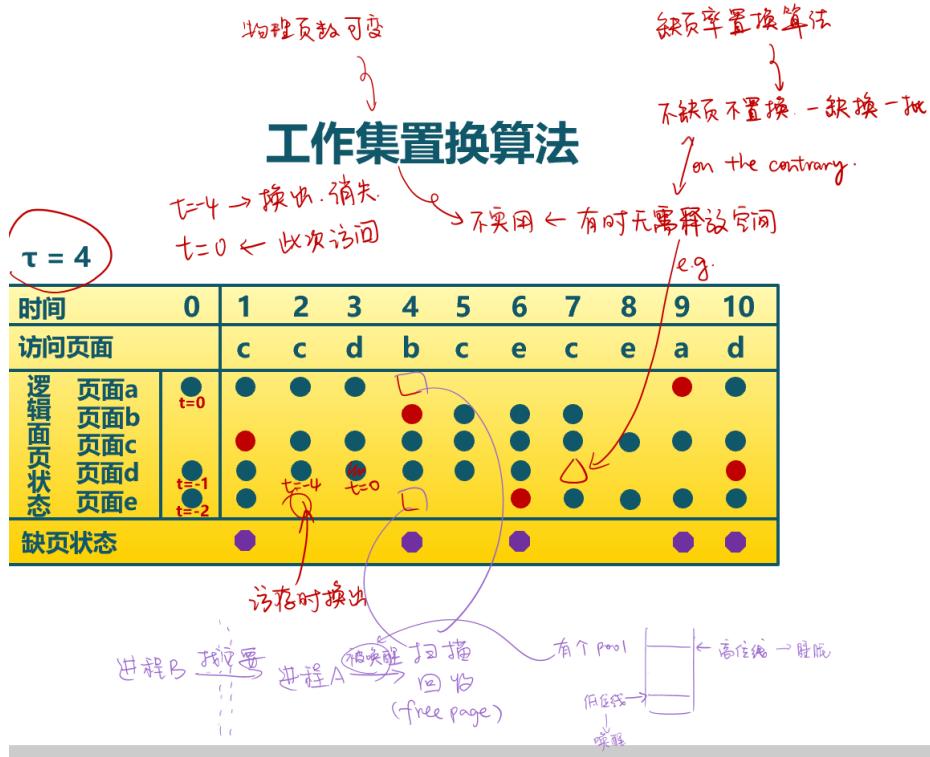
- 进程开始执行后，随着访问新页面逐步建立较稳定的工作集
- 当内存访问的局部性区域的位置大致稳定时，工作集大小也大致稳定
- 局部性区域的位置改变时，工作集快速扩张和收缩过渡到下一个稳定值

常驻集：当前时刻，进程实际驻留在内存中的页面集合。取决于OS分配给进程的物理页面数和页面置换算法。（工作集是固有性质）

缺页率：常驻集 > 工作集时缺页较少；工作集过渡时缺页较多；常驻集大小达到一定数目后，缺页率也不会明显下降。

改善缺页率的出发点一般是增大常驻集。

算法实现：对窗口大小 τ ，维护链表。访存时换出不在 W 内的页面，更新链表；缺页时换入页面，更新链表。



缺页率置换算法 (PFF: Page Fault Frequency)

- 缺页率：缺页/访问次数 or 缺页平均时间间隔倒数
 - 影响因素：置换算法、分配物理页面数、页面大小、编程
- 通过调整常驻集大小，使每个进程的缺页率保持在合理范围内

缺页率置换算法的实现

- 访存时，设置引用位标志
 - 缺页时，计算从上次缺页时间 t_{last} 到现在 $t_{current}$ 的时间间隔
 - 如果 $t_{current} - t_{last} > T$ ，则置换所有在 $[t_{last}, t_{current}]$ 时间内没有被引用的页
 - 如果 $t_{current} - t_{last} \leq T$ ，则增加缺失页到工作集中
- 访存设 A
缺页时刻
距上次缺页
 at
超过 T 置换页
内全部
未 A.
在 T 内，增加页
不换出.

页置换算法比较！

	Principle	Performance
Most optimal	Running flow is known	Perfect but can't be realized
NRU	"RM" classification	Not bad, time-consuming
FIFO	Time sorting	Not reasonable
LRU & NFU	Locally optimal	Approach "perfect"
Working set	Prepaging	Stable and efficiency

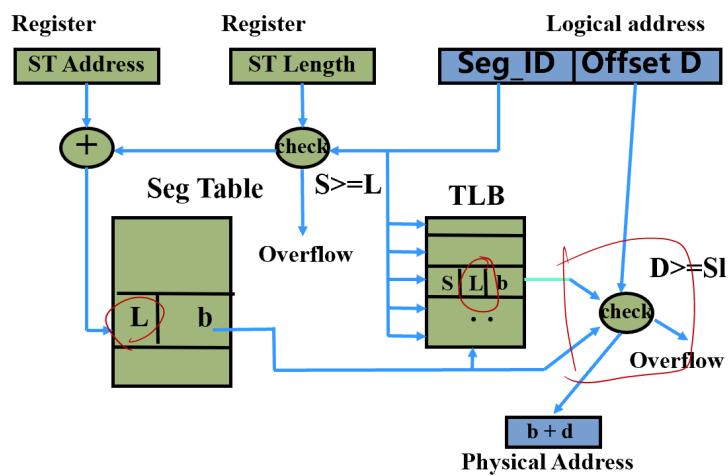
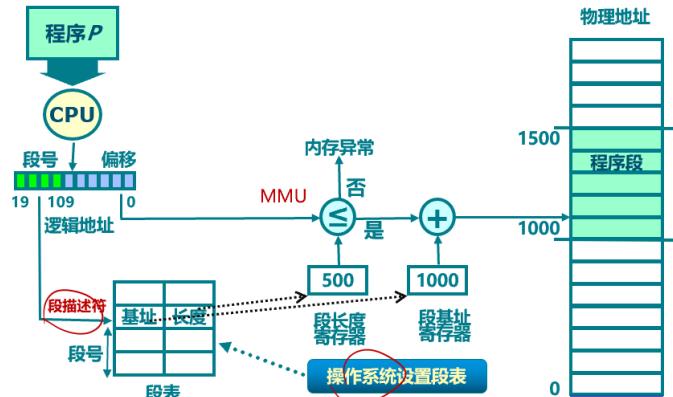
段式

“段”：表示访问数据和存储数据等属性相同的一段地址空间，对应一个连续的内存块，是程序的逻辑分区。

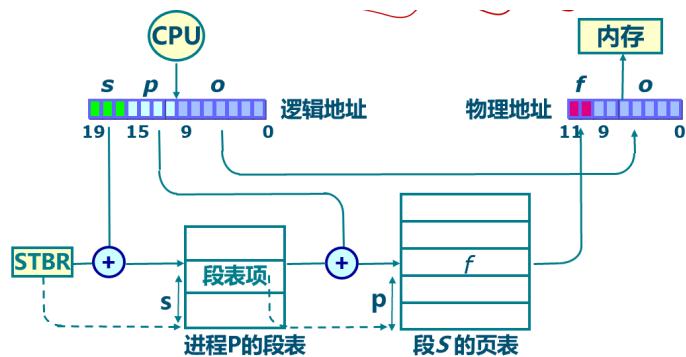
目的：更细粒度和灵活的分离与共享

- 段表中存了基址&长度：先检查是否越界，后叠加偏移量。

段访问的硬件实现



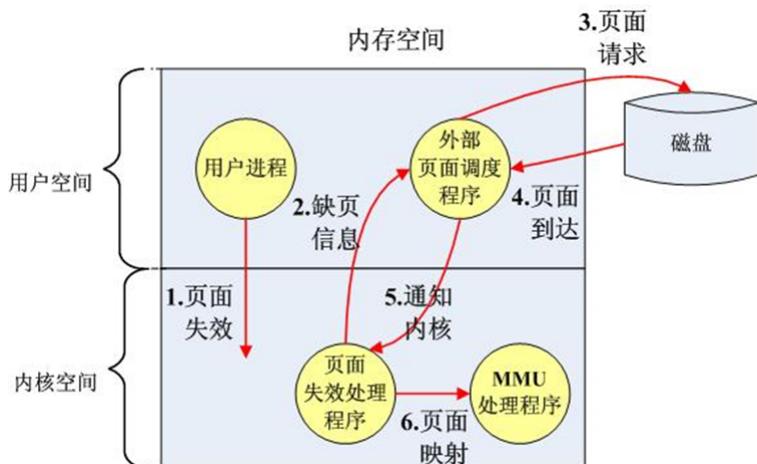
段页式：用段技术组织程序内容，页技术组织物理内存 e.g. 给每个段加一级页表



- 段式：内存保护（属性）
- 页式：内存利用、优化转移、后备存储（灵活）
- 内存共享：一个段一个页表，所有进程都指向这个页表

页&段式对比！

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can procedure and data be distinguished and separately protected	No	Yes
Can tables whose size fluctuates be accommodated easily	No	Yes
Is sharing of procedures between user facilitated	No	Yes



page_free_list: 找页分配

- 进程告知OS需要空白Page
- OS找到并将Page给进程，修改进程PT
- 为了定位空白Page，OS维护一个供系统整体使用的page_free_list (Bitmap)
- page_free_list中记录每个Page的使用情况：对内存页的申请与释放、对进程的销毁
- page_free_list管理的是物理地址

进程管理

- 创建

- 分配一级页表 *i.e.* 目录表
- 存入cr3
- 程序加载，在虚拟地址中载入**分段信息**和部分数据、指令
 - 根据编译链接的结果，将信息放在程序二进制头
 - 建立VA与文件物理偏移量的对应关系
- 创建完成，**PC转去main执行**
- 缺页：**OS加载新数据**

- 调度

- 占据CPU的进程：
 - 载入PDE
 - 重新配置MMU，TLB失效
 - cache
- 未占据CPU的进程：
 - 释放D=0内存，修改页表
 - 写出D=1的内容，释放内存
 - 多级页表中，如果整个二级都失效也可释放
 - 意外杀死

进程

进程：程序运行的过程，包括输入输出、程序和状态

进程	程序
动态	静态
暂时	永久
程序+数据+PCB	语句+指令

程序可对应多个进程，进程可调用多个程序

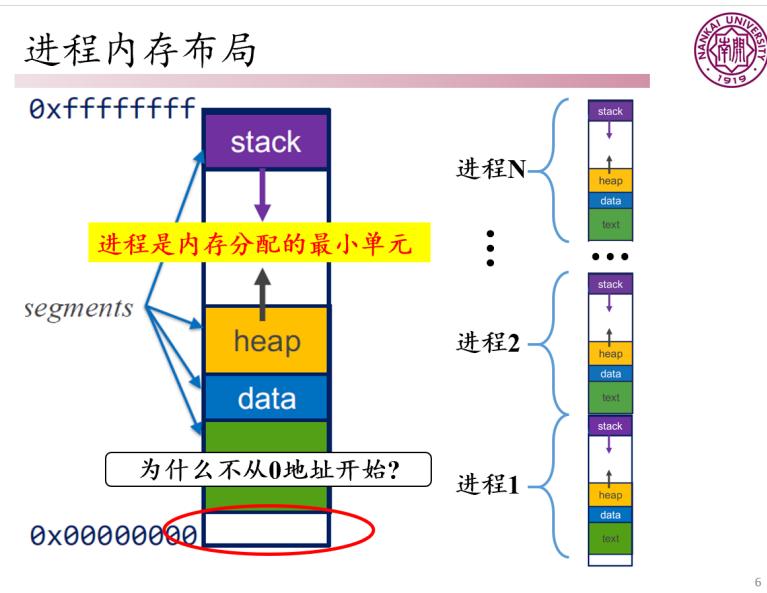
是从程序到指令间的数据结构

PCB/进程描述符/属性

- 对于任意一个进程，OS维护一个PCB描述进程的基本信息和运行状态，进而控制和管理进程。是进程存在的唯一标志。

- OS由内核维护，用户不可直接修改。
- PCB的**大小决定并发程度**。不同状态的进程存在不同PCB表（e.g. 就绪/等待/僵尸队列）里。
- 进程的创建和撤销本质是对PCB的。
- 包括：三部分----进程、存储、文件
 - 进程：寄存器、程序计数器、程序状态字、堆栈指针、进程状态、优先级、调度参数 pid 、父进程、信号……内存分配状况（进程在内存中的位置 i.e. Page table）、磁盘位置、调度信息、中断栈
 - 存储：正文段、数据段、堆栈段指针
 - 文件：根目录、文件描述符、 uid ……打开文件状态

进程内存布局



进程切换——上下文切换

进程暂停，调度另一个从就绪到运行

要求——快：

- 切换内存（堆栈）
- 切换CPU寄存器&指令指针寄存器（i.e. 保存&恢复上下文（PCB中））：存X用户寄存器>>存X内核寄存器>>恢复Y内核寄存器>>恢复Y用户寄存器
- 生命周期信息：寄存器（PC、SP…）、CPU状态、内存地址空间

TSS：保存各种寄存器。CPU中存在TR寄存器，修改TR会触发当前TSS内容保存在当前任务中，用新进程给TSS赋值，实现CPU寄存器的切换。替代汇编代码。

switch_to

```
.text
.globl switch_to
switch_to:                      # switch_to(from, to)
```

```
# 【进入函数的时候栈自上而下（小到大）是： eip->from's context addr->to's context addr,  
esp指向eip
```

```
# save from's registers  
movl 4(%esp), %eax          # eax points to from      【让eax指向from's context  
地址】  
popl 0(%eax)                # save eip !popl  
#【把栈top的eip pop给from's context的top】  
movl %esp, 4(%eax)  
movl %ebx, 8(%eax)  
movl %ecx, 12(%eax)  
movl %edx, 16(%eax)  
movl %esi, 20(%eax)  
movl %edi, 24(%eax)  
movl %ebp, 28(%eax)          # save ebp::context of from  
  
# restore to's registers  
movl 4(%esp), %eax          # not 8(%esp): popped return address already  
# eax now points to to  
#【esp+4现在是to's context（原先在esp+8）】  
movl 28(%eax), %ebp  
movl 24(%eax), %edi  
movl 20(%eax), %esi  
movl 16(%eax), %edx  
movl 12(%eax), %ecx  
movl 8(%eax), %ebx  
movl 4(%eax), %esp          # restore esp::context of to  
  
pushl 0(%eax)                # push eip 【把eax, 即to's context的top位置的eip压  
入esp】  
  
ret
```

进程镜像：对进程整个生命周期的描述

由代码段、数据段、PCB组成。

- 用户层上下文：用户空间 (.text、.stack、.data)
- 寄存器层：PC、PSW、IR、堆栈指针等
- 系统层：PCB、资源、动态内核栈

进程状态

- 三状态模型



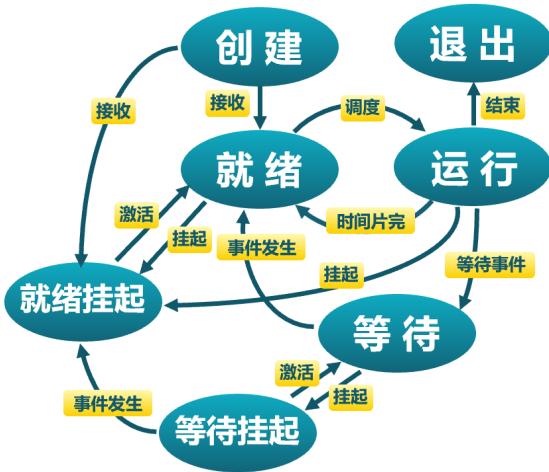
状态	特征
运行	正在使用CPU
就绪	可运行，在等待
等待	不能运行，等待外部事件发生（在内存）
创建	DS已被创建，但还未创建运行镜像。过程中分配pid、申请PCB、分配空间、初始化PCB（优先级）、加入就绪队列
退出	完成了全部任务，但DS还未被删除——“僵尸”：子进程归还了资源但遗留DS

- 状态迁移
 - 就绪不能直接等待，等待不能直接运行
 - 就绪>>运行：获得使用CPU的优先权

进程挂起：对进程做分级处理，引入优先级会使某进程等待时间过长而被换至外存（运行镜像被换出到硬盘）

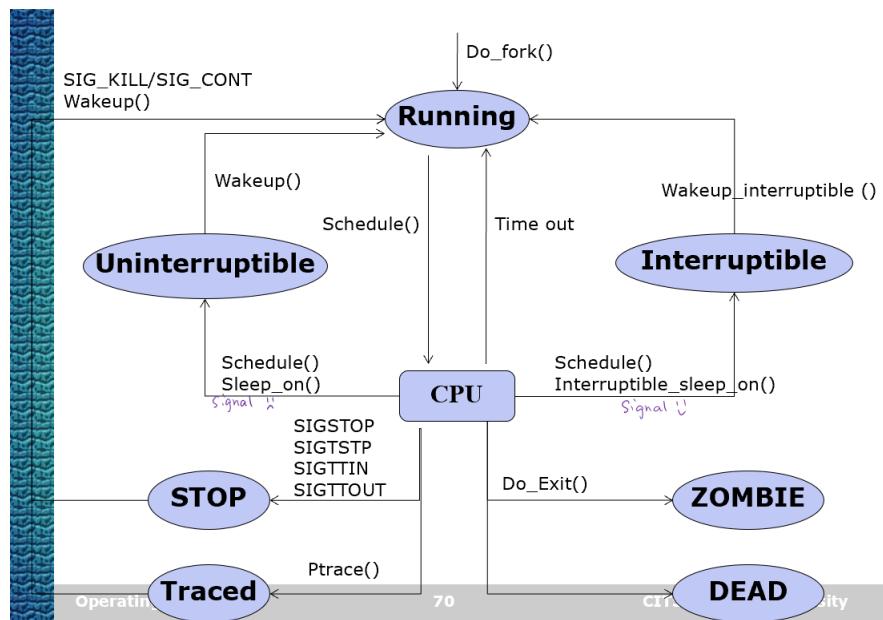
目的：

- 提高处理机效率：就绪进程表空时，提交新进程，双挂起模式下直接进入就绪挂起
- 为运行进程提供足够内存
- 便于调试：挂起被调试进程时才能对其地址空间读写



- 单挂起——只存在阻塞挂起 vs. 双挂起——就绪(e.g. 新进程创建)&阻塞态都可挂起
- 就绪挂起：进程在外存，但只要进入内存即可运行。
- 状态转移：
 - 阻塞>>阻塞挂起：没有进程处于就绪状态或就绪状态要求更多的资源 (i.e. 没有多余的分给阻塞进程)
 - 阻塞挂起>>阻塞：进程释放内存，有高优先级阻塞挂起进程
 - 就绪>>就绪挂起：有更高优先级就绪
 - 运行>>就绪挂起：(抢分式分时系统) 有高优先级阻塞挂起进入了就绪挂起
 - 阻塞挂起>>就绪挂起：事件发生
- “激活”：把进程从外存换入

Is Linux Process States needed?



进程调度：CPU资源的时分复用

时机：进程创建/退出（运行>>阻塞）、IO阻塞、中断

原因：做合理决策、保证效率

How：根据策略选择进程、switch

注意：调度频率、是否抢占式算法、CPU/IO密集进程

调度程序：一个挑选就绪进程的内核函数

- 非抢占系统：OS不能轻易将进程的某些资源收回，只能等待主动放弃
- 可抢占系统：OS可以将进程的某些资源收回，用于其他进程，并且过段时间可以将相等的资源归还
 - 抢占时机：中断请求被服务例程响应完成；时间片用完；利用任何进程被打断的机会（e.g. 中断、异常；时钟中断）

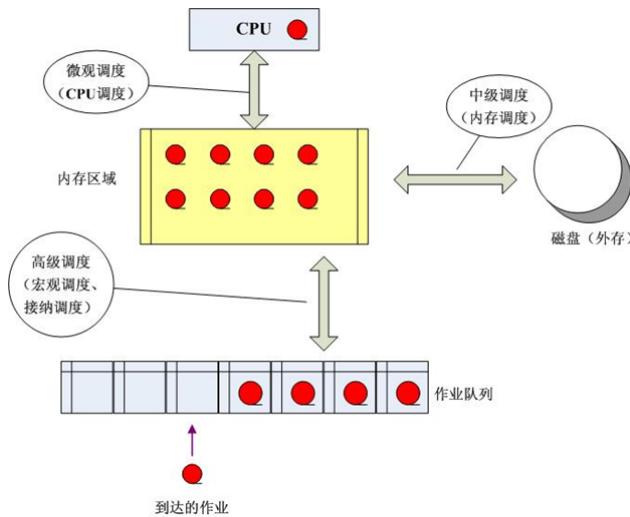
“资源调度”：进程拥有资源的时间，服务的顺序；抢占式

“资源分配”：进程获得资源的数量；非抢占式

- 不同操作系统有不同目标：
 - For all——公平、强制、平衡；
 - 批处理——吞吐率、周转时间、CPU利用率；
 - 交互式——响应时间、恰当性；
 - 实时——防止数据丢失、可预测性。
- 决策指标：
 - CPU使用率：CPU处于忙状态的时间比
 - 吞吐量：单位时间完成的进程数—>OS的计算带宽
 - 减少开销：操作系统开销、上下文切换开销
 - 提高资源利用：CPU、IO
 - 必须保证吞吐量不受交互影响
 - 周转时间：从初始化到结束包括等待的总时间
 - 等待时间：在就绪队列中的时间（减少每个）
 - 初始等待：第一次被调度 - 到达时间
 - 总体等待：在就绪态队列中等待的总时间
 - 响应时间：用户态程序完成某些功能所需的总时间；从提交请求到产生响应所花费的总时间—>OS的计算延迟
 - 减少时间：及时处理输入请求，尽快反馈输出
 - 减少平均波动：交互式系统的可预测性比高差异低平均更重要
 - 系统开销：完成任务所需的额外资源量（时间，空间）
 - 公平性：每个任务使用CPU的时间或资源量
 - 饥饿度：任务两次被调度之间的等待时间

调度算法：改变响应时间、等待时间

三层调度：接纳调度(高级)、内存调度(中级)、CPU调度(微观)



- 非抢占式

- **FCFS**
 - 平均等待时间波动大 (短的排在长的后面)
 - 资源利用率 (CPU&IO) 低 (CPU密集型程序会导致IO空闲时IO密集型程序也在等待)
- **短任务优先SPF**
 - 最优平均周转时间、可能可以最小化平均响应
 - 实际不好: 饥饿 (连续短进程) 、无法保证同时到达、无法估计执行时间
- **最高响应比优先HRRN:** $R=(w+s)/s$ ——关注等待时间, 防止饥饿
 - w: 等待时间
 - s: 执行时间

- 抢占式

- **短剩余时间优先SRF**——SPF的改进
- **时间片轮换RR:** 阻塞/完成/超时时switch (FCFS)
 - 时间片: 分配处理机资源的最短时间单位
 - 额外切换
 - 时间片长度
 - 太大: 等待时间过长/退化为FCFS
 - 太小: 切换频繁, 影响吞吐量

基于优先级

- **多级队列调度MQ:** 不同就绪队列采用不同调度
 - 队列间固定优先级+RR (不同大小)
- **多级反馈队列MLFQ:** 进程可在不同队列间移动的MQ
 - 新来的进入最高级
 - 时间片随优先级降低而增加
 - 当前时间片未完成则下降
 - CPU密集型下降快, IO密集型停留在高级
- **多优先级队列优化方案:**
 - IO (交互) 型: 进入最高级, 处理完一次转入阻塞

- 计算 (CPU) 型：每次都执行完时间片进入更低级。最终用最大时间片执行，减少调度。
- IO不多，CPU为主：IO完成后放回IO请求时离开的队列，以免逐次下降。
- IO完成时提高优先级，时间片用完时降低。
- **优先级算法：**MQ的改进，给任务&队列分配优先级。根据等待时间提升优先级，平衡进程对响应时间的要求。适用作业和进程调度，分成抢先和非抢先式。
- 静态优先级：创建时确定，终止前不变。
 - 依据：进程类型（系统/用户）；资源需求（少的高）；用户要求
 - 动态：
 - 修改：等待时间长-->提高；时间片耗尽-->降低
 - 线性优先级调度算法 (SRR, Selfish Round Robin)
- 调度机制：参数化以决定优先级

进程通信方式

- 直接通信
 - send&receive函数
 - 链路属性：自动建立，一条链路<=>一对进程
- 间接通信：OS维护消息队列实现
 - 收发：消息队列有唯一标识；共享队列=>可以通信
 - 链路属性：共享队列才能建立连接，可“多对多”
- 阻塞——同步：S：发送后等待，直到被接收；R：等待直到接收。
- 非阻塞——异步：R：没有消息发送时，在请求接收消息后接收不到（不等）

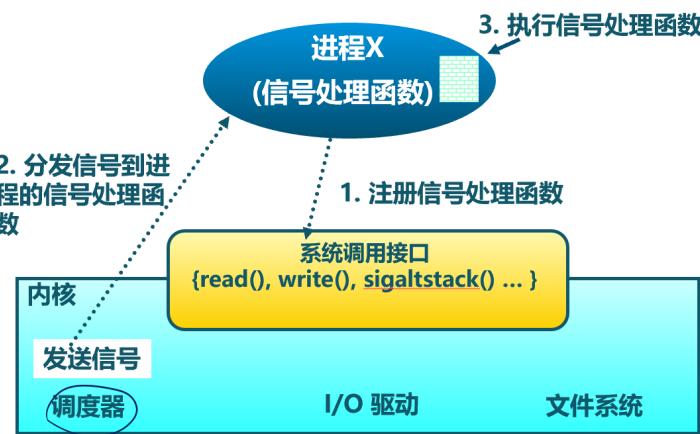
通信链路缓冲

几种常用方式

- | 信号和共享内存全局可见
- **信号：**进程间的软件中断通知和处理机制。信号事件发生时不传递信息，只绑定函数。
 - 触发 (OS发送信号)：
 - 进程主动请求OS发送信号 e.g. kill, 调试、挂起、恢复、超时
 - OS发现一个系统事件 e.g. 除0错误
 - **kill(pid[i], SIGINT); 子进程收到信号，就终止本进程 (kill)**
 - 捕获：用一个**信号处理函数**（用户态的捕获函数），以信号为参数传入，另一个参数是处理函数


```
signal(SIGINT, int handler); //register handler for SIGINT
```
 - 忽略：执行OS指定的缺省处理 e.g. 进程终止/挂起
 - 屏蔽：禁止进程接收&处理信号
 - :((信息量小

信号的实现



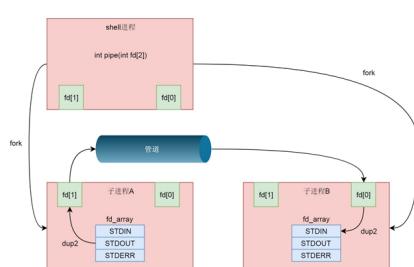
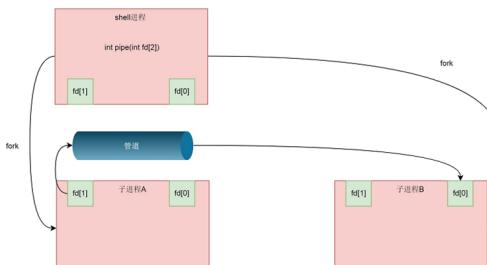
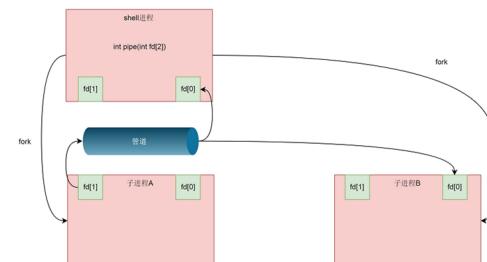
- 管道（串行）：基于内存文件——0 stdin, 1 stdout, 2 stderr

- 子进程从父进程继承文件描述符
- 读read, 写write (scanf基于read, printf基于write)
- 创建pipe(rfd[2])——建立匿名管道的函数原型：
 - rfd[0]——读(的)文件描述符
 - rfd[1]——写文件描述符
- 创建过程: shell创建pipe, 分别拷贝出子进程进程设置stdout(写)和stdin

上一阶段的输出是下一个阶段的输入

- cmd1 argv1 | cmd2 argv2 | cmd3 argv3...
叫做匿名管道, 用完即销毁 (最常用于Shell) Linux系统中很常用

命令2接到命令1的输出和自己的输入一起作为输入, 然后得到命令2输出, 再将命令2的输出与命令3的输入一起作为输入



- 消息队列（串行）：OS维护，以字节序列为基本单位的间接通信

- 消息——字节序列，有标识。相同标识的消息FIFO组成消息队列。
- **共享内存（传输效率高）（全局变量<--线程通信）**：把同一个物理内存区域同时映射到多个进程的内存地址空间（“零拷贝”）
 - 零拷贝：CPU不执行存储区域之间的拷贝。实现应用程序可以绕开内核直接获得硬件存储。
 - 举例：
 - DMA：用户进程内存和IO设备之间直接拷贝。
 - mmap：将文件映射进内核缓冲区，用户进程可以共享。
 - 进程需要明确设置共享内存段
 - 快速方便；但需要额外的**同步机制**（读写/访问冲突——信号量协调）

Meltdown

```

1 做一些事把缓存清掉，比如说读2GB电影；
2 呵呵；
3 秘密 = 读内存[666]；
4 如果什么什么条件就跳转到第2行；
5 其实没读到值 = 读内存[1万 + 秘密]；
```

```

本来第3行会死，但即使死了，由于CPU内部异步偷偷执行大段代码，在CPU执行“告诉操作系统第3行出了个非法操作”这个任务期间，其实4、5行都被分别执行了。  
等CPU终于根据操作系统返回的结果确定4、5不应该被运行，它会撤销这两行对外界的影响。【其实没读到值】不会被赋值。  
然而这个撤销并不能撤销第5行产生的副作用：CPU其实已经将“读内存地址 1万+秘密”这个指令传给内存/缓存系统了。

哈哈哈哈哈哈哈哈哈哈 vs. 正经的ppt：

本来应该死在4，但在4执行完但死之前执行了7（因为分支预测啥的），[rbx+rax]就已经被访问过了，之后再访问就会很快了（rbx有没有被赋值不重要）

```

1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx] 非法地址，产生异常
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax] 会被乱序执行

```

乱序执行的结果，不会影响体系结构状态，但会影响“微体系结构状态”

Probe数组中的其中一项（rax索引的）进入了cache

攻击者通过遍历数组会知道哪一项访问时间最短，从而反推出rax的值

- 危害：attacker使用普通权限可以读取到内核内容。
- 抵抗：PHP；用户&内核空间分表；不把全部物理地址映射进内核

**Hammer Attack——KSM: Kernel Samepage Merging**（默认用后出现的页面替代新出现的）：申请三个连续页面，中间的复制密码，等待合并

## 线程：进程内一个相对独立的、具有可调度特性的指令执行流的最小单元，是CPU调度的基本单位

引入线程后，进程将只作为除CPU外的系统资源的分配单元。线程则作为CPU调度单元(同一地址空间内的上下文切换代价较小)

进程：OS为正在运行的程序提供的抽象。

线程：并发性抽象，可以被理解为轻量级进程。

### why 进程 need 线程？

- 充分利用**多核**
- 呈现出程序结构：描述属于同一程序下的并发任务
- 高响应性：**分离交互与功能性线程**
- **隐藏IO等待**

**线程库：** (1) 多线程编程的接口； (2) 记录线程状态和调度各线程的运行机制。

**讨论出发点：** (1) 管理者； (2) 系统调用（权限）； (3) 创建； (4) 多核

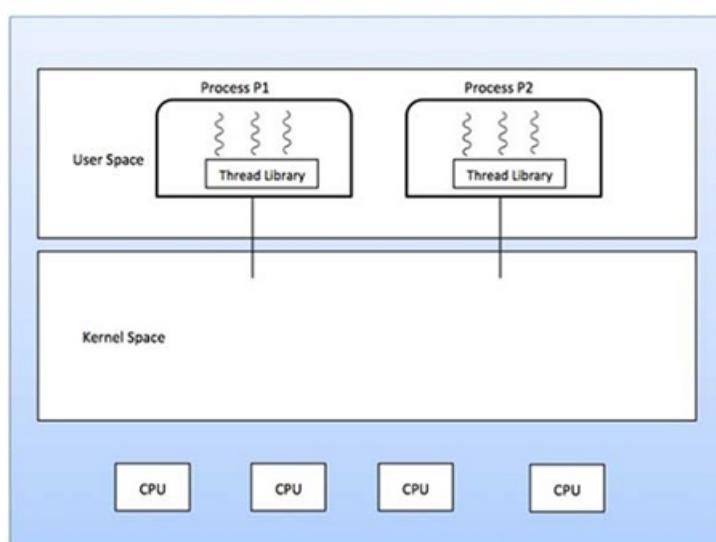
- **纯用户级 (ULT)**：线程的管理全部由用户程序完成，核心部分只对进程管理。

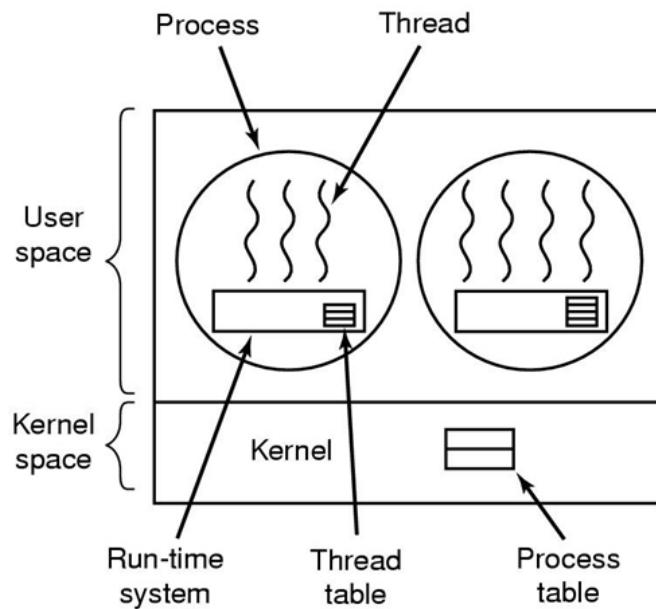
◦ :)

- 不用切换特权
- 根据用户需求调用
- 不需要修改内核
- 创建快、分布式（用户）管理

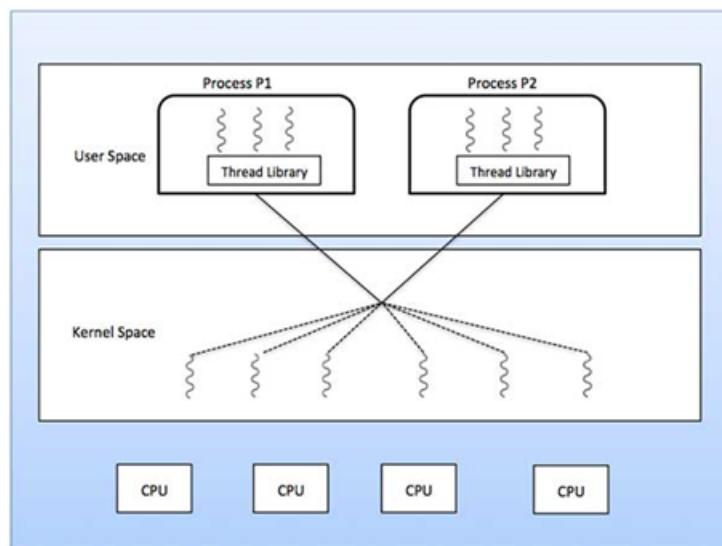
◦ :(

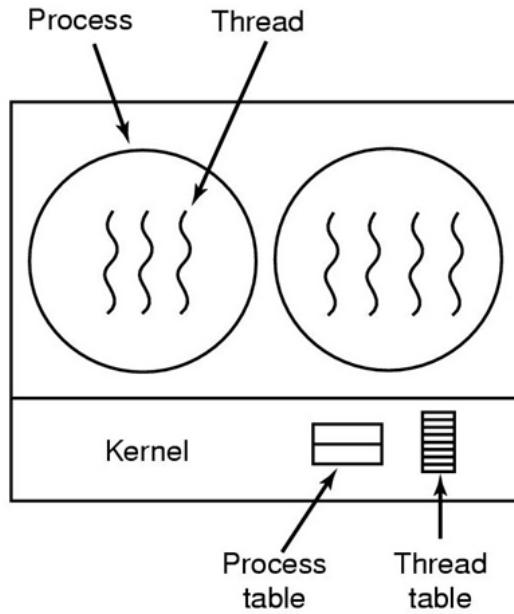
- 系统调用会阻塞
- 多线程只能使用同一个核



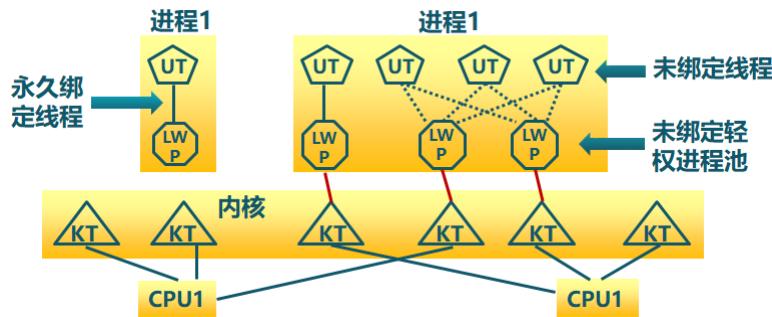


- **内核级 (KLT)** : 线程由内核管理，内核给应用程序级提高系统调用，实现对线程的使用。
  - ;)
    - OS可以调度多线程到多核
    - 如果进程的一个线程被阻塞，可以调度其他进程
  - :(
    - 创建慢（权限、安全）
    - 同进程的线程切换需要内核模式





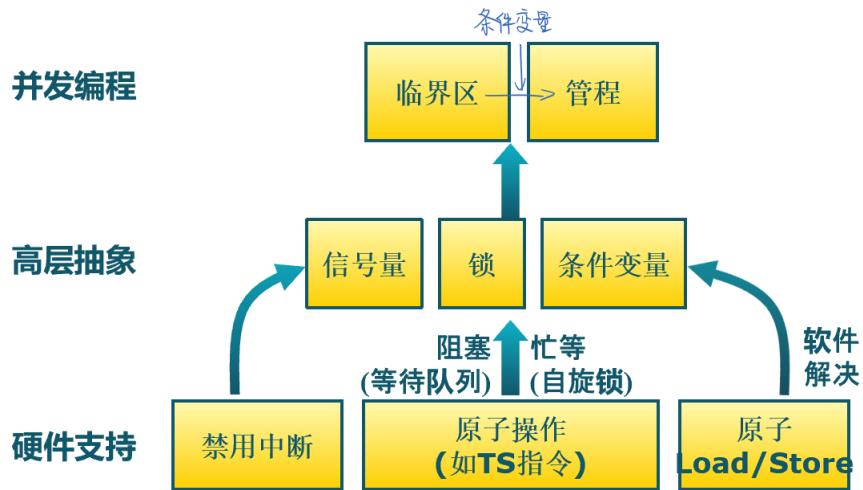
**轻权进程：内核支持的用户线程**（通过从属于内核线程支持的用户进程）。一个进程可有多个线程，每个轻权进程由一个单独的内核线程来支持。*i.e.* 一个内核线程 >> 一个用户进程 >> 多个用户线程（是轻权进程）



## 同步和互斥

- 经典问题：生产者消费者——固定缓存
  - “同步”：多线程需要协调工作而等待、传递消息，产生制约。
  - “互斥”：一个线程进入临界区，其他想进入临界区的线程必须等待。
    - 临界资源：e.g. 物理设备（内核）、全局变量
    - 临界区：线程访问临界资源的代码
    - 竞争条件：多进程同时进入临界区产生的不可预知的状态。造成不可预知的原因是多线程代码可能交叉执行 (*i.e.* 调度顺序不可知)
    - Good solution (临界区访问规则)
      - 安全性：满足至多一个进程在CR内
      - 通用性：不能有CPU速度和个数的前提
      - 没有处于CR外的进程可以阻塞其他进程
      - 公平性：不能永久等待
    - “异步”：互斥是为了防止竞争；异步是为了实现正确的逻辑顺序。
- 临界区访问规则：空闲则入，忙则等待，有限等待（，让权等待——不能进入CR的进程应释放CPU，如转换到阻塞状态）
- 中断屏蔽：一定互斥。但低效、不安全（用户操作）、对多核无用

## 基本同步方法



- 忙等待：全局变量——记录&检查

- 解决两个进程互斥
- 耗时、编程困难、优先权
- 趋势：原子操作
- 锁变量（检查）
  - 使用超过信号量
  - 消极机制 改进：事务型内存
- 严格轮换：
  - 耗时
  - 单个变量：CR外的进程也有阻塞作用（在出CR但恢复变量值（让位）前会阻塞其他入CR进程）
  - bool数组

第二种尝试：每个进程用一个布尔量标征自己的状态，类似自旋锁功能。

|                                                                                                                                                                      |                                                                                                                                                                        |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>/* PROCESS 0 */<br/>•<br/>•<br/><b>while</b> (flag[1])<br/>    /* do nothing */;<br/>flag[0] = true;<br/>/*critical section*/;<br/>flag[0] = false;<br/>•</pre> | <pre>/* PROCESS 1 */<br/>•<br/>•<br/><b>while</b> (flag[0])<br/>    /* do nothing */;<br/>flag[1] = true;<br/>/* critical section */;<br/>flag[1] = false;<br/>•</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

同时

缺点：一进程在临界区内失败会阻塞另一进程；临界区会进入多进程。

第三次尝试：把**flag[\*]=true**换到前面

|                                                                                                                                                                        |                                                                                                                                                                        |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>/* PROCESS 0 */<br/>•<br/>•<br/>flag[0] = true;<br/><b>while</b> (flag[1])<br/>    /* do nothing */;<br/>/* critical section */;<br/>flag[0] = false;<br/>•</pre> | <pre>/* PROCESS 1 */<br/>•<br/>•<br/>flag[1] = true;<br/><b>while</b> (flag[0])<br/>    /* do nothing */;<br/>/* critical section */;<br/>flag[1] = false;<br/>•</pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

问题：若两进程**flag**同时被置成**true**，又没有进入临界区，将产生死锁。

- Peterson (Dekkers)

- 两个变量：
  - turn——该谁进入CR
  - flag[]——ready or not
- 先把turn设为另一方，让对方先完成

## Busy-Waiting Correct Solution(Peterson's)

```
boolean flag [2];
int turn;
void P0()
{
 while (true)
 {
 flag [0] = true;
 turn = 1;
 while (flag [1] && turn == 1)
 /* do nothing */;
 /* critical section */;
 flag [0] = false;
 /* remainder */;
 }
}
void P1()
{
 while (true)
 {
 flag [1] = true;
 turn = 0;
 while (flag [0] && turn == 0)
 /* do nothing */;
 /* critical section */;
 flag [1] = false;
 /* remainder */;
 }
}
void main()
{
 flag [0] = false;
 flag [1] = false;
 parbegin (P0, P1);
}
```

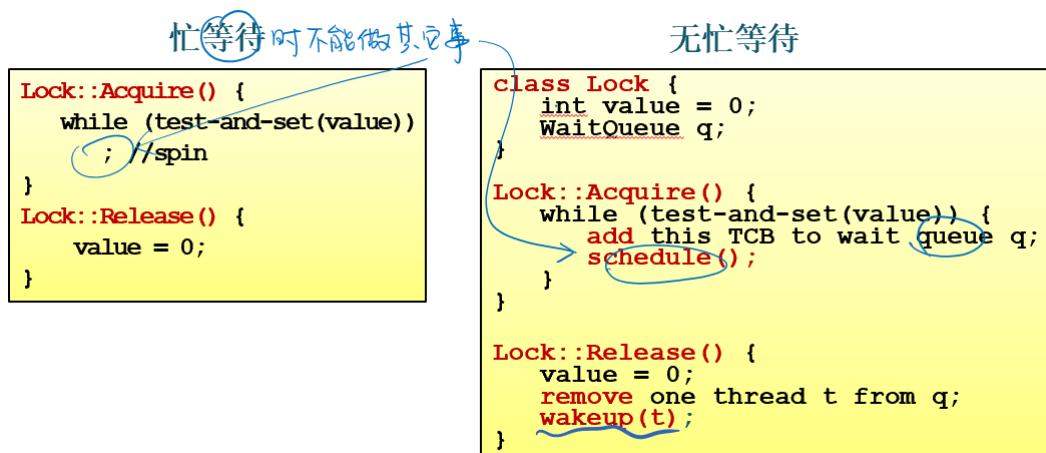
- Dekkers: 分开判断对方flag&turn。不是自己turn时，先取消自己的flag，等对方完成，再重新设置自己的flag。

## Busy-Waiting Correct Solution(Dekker's)

```
boolean flag [2];
int turn;
void P0()
{
 while (true)
 {
 flag [0] = true;
 while (flag [1])
 if (turn == 1)
 {
 flag [0] = false;
 while (turn == 1)
 /* do nothing */;
 flag [0] = true;
 }
 /* critical section */;
 turn = 1;
 flag [0] = false;
 /* remainder */;
 }
}
void P1()
{
 while (true)
 {
 flag [1] = true;
 while (flag [0])
 if (turn == 0)
 {
 flag [1] = false;
 while (turn == 0)
 /* do nothing */;
 flag [1] = true;
 }
 /* critical section */;
 turn = 0;
 flag [1] = false;
 /* remainder */;
 }
}
void main ()
{
 flag [0] = false;
 flag [1] = false;
 turn = 1;
 parbegin (P0, P1);
}
```

- TSL: Test and Set Lock

- “TSL”: 硬件原语
- 自旋锁: 无论test是否释放都把锁置1



- Sleep and wakeup: 基于原子操作 (一种系统调用)

- `schedule()` & `wakeup()`
  - :) 可扩展性好
  - Simple——`sleep()`&`wakeup()`
    - 风险：唤醒信号可能丢失，需要DS维护稳定性
    - :( 多进程不通用&低效
  - 信号量：追踪记录竞争条件的数据结构
    - 两个标准原语：P（可能阻塞）、V（不会阻塞）过程中不会中断/调度
      - P: sem--&check --> may block&add to queue
      - V: sem++&check --> may remove someone from queue&wakeup
    - 一个Semaphore：累计唤醒次数
    - queue (optional) : 等待进入的线程
- 对不同需求：
- 同步需求：sem set 0，使用前check P，提供后check V (Possible separated)
    - 生产者消费者——有限缓存
      - 外层：empty/full（缓存）；内层：mutex（锁——CR）
      - 内外层顺序不可变：否则先占用mutex（CR），后consumer由于缓存为空而休眠 => 死锁
    - 互斥：sem set 1，访问前check P，访问后check V
      - 要求：PV成对，次序无误
  - 监管者：一系列过程、变量、D
    - 管程
 

"管程" 可以看做一个软件模块，将共享的变量和对于这些共享变量的操作封装起来，形成一个具有一定接口的功能模块，进程可以调用管程来实现进程级别的并发控制。

      - 锁：代码的互斥访问
      - 条件变量：共享数据的并发访问
        - (是资源) 是等待机制
        - 每个变量对应一个等待队列
        - `wait()`/`schedule()`: 将自己阻塞，唤醒别的等待者/释放互斥访问
        - `signal()`: 唤醒一个等待
      - Hansen (reality) vs. Hoare
        - Hansen: 唤醒进程`signal`后先`release`再回到被唤醒进程
        - `release`仅仅是“提示”，需要重新检查变量
          - :) 高效
      - Hoare: 唤醒进程`signal`被唤醒进程，等待被唤醒`release`后再`release`自己
        - `release`表示放弃
        - :( 低效
  - 哲学家晚餐——同步互斥
    - 死锁：环形等待资源
      - 形成前提：(选择破坏)
        - 互斥资源—>不用锁（资源够多）
        - 获得资源前不释放现有资源—>一次分配
        - 非抢占—>可保存恢复

- 环形等待—>只用一个锁(?)
- 饥饿: 线程无休止等待CPU调度
- 死锁会饿死, 饿死不一定因为死锁
- 写者读者—多进程同步
  - 如果读者持续到达, 读者优先: (一个写者)

## 读者—写者问题



### • 关系分析

- 读者和写者是互斥的
- 写者和写者是互斥的
- 读者和读者不互斥

### • 整理思路

- 确定线程数: 读者和写者
- 写者比较简单: 它与任何进程都互斥 (一组PV操作)
- 读者比较复杂: 它与其它读者不互斥 (需要计数器参与)  
计数器不为0说明还有人读文件

### • 确定信号量

- 写者需要一个二值信号量保护写操作, 读者一个计数器 count以及一个二值信号量保护count

## 读者—写者问题



```

int count = 0
Semaphore mutex = 1
Semaphore rw = 1

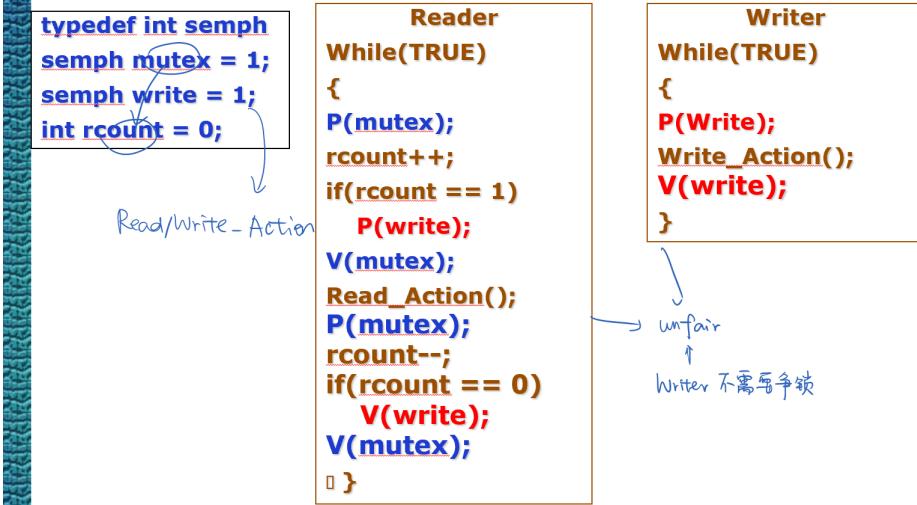
Reader(){
 while(1){
 mutex.P(); //为了count
 保护其他读者 if(count == 0)
 rw.P(); 互斥写者
 count++;
 mutex.V();
 读操作;
 mutex.P();
 count--;
 if(count==0)
 rw.V();
 mutex.V();}}
}

Writer(){
 while(1){
 rw.P();
 写操作;
 rw.V();}
}

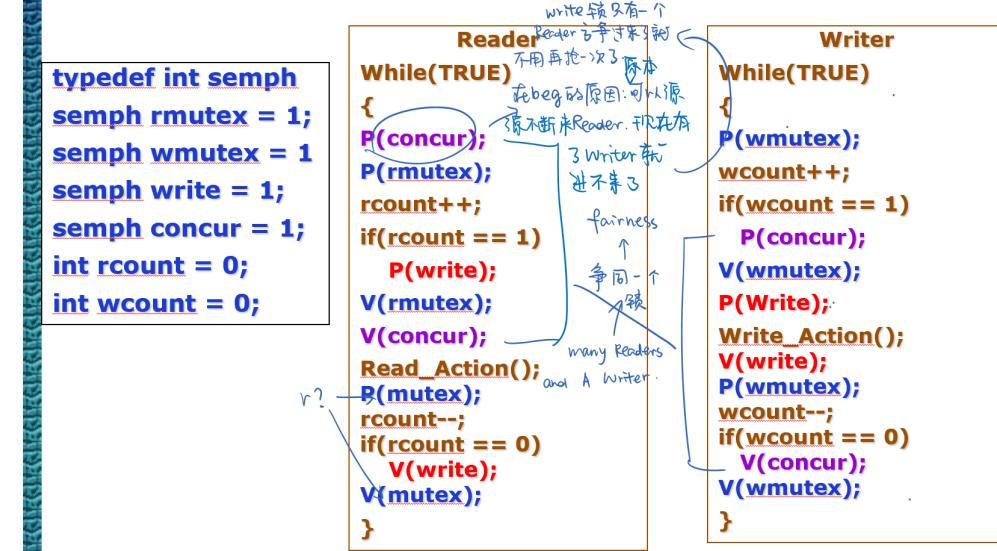
```

有没有别的写法?

# Give priority to Reader



## Keep fairness: reader-writer



- 理发师—复杂背景 (椅子是生产者消费者问题)
- 消息传递: 操作前传递消息

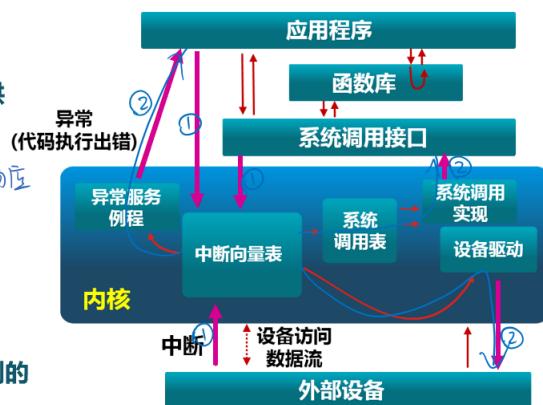
- Simple
- 邮箱
- Rendezvous

## 用户与内核交互

# 中断、异常和系统调用比较

## 源头①

- 中断：外设
  - 异常：应用程序意想不到的行为
  - 系统调用：应用程序请求操作提供服务
- ②
- 响应方式
- 中断：异步：不需要等待，随时响应
  - 异常：同步“并发”
  - 系统调用：异步或同步
- 处理机制
- 中断：持续，对用户应用程序是透明的
  - 异常：杀死或者重新执行意想不到的应用程序指令
  - 系统调用：等待和持续



- 系统调用 (trap) : 程序向OS发出服务请求, OS为程序提供接口

- 实现
  - 每个系统调用对应一个调用号，系统调用接口根据调用号维护表索引
  - 系统调用接口调用内核态的系统调用功能，返回调用状态和结果
- vs. 函数调用
  - 用户可以通过系统调用提升权限
  - 接口行为固定
  - 系统调用——int&iret，进行堆栈&特权转换
  - 函数调用——call&ret，常规调用不切换堆栈

- 中断 (interrupt) : 来自硬件设备的处理请求 (信号)

- 硬件：在CPU初始化时设置中断使能标志
  - 依据内部或外部事件设置中断标志
  - 依据中断向量调用响应中断服务例程
- 软件
  - 保存现场（编译器）
  - 中断服务处理（服务例程）
  - 清除中断标记
  - 恢复现场
- 嵌套
  - 硬件中断服务例程可被打断
  - 异常服务例程可被打断（硬件中断），可嵌套（服务例程缺页）

- 异常 (exception) : 硬件发现的、由程序产生的非法指令或其他原因导致当前指令执行失败