1 概述

定义

- 负责管理协调硬件、软件等计算机资源的工作
- 为上层用户、应用程序提供简单易用的服务
- 是一种系统软件

功能和目标

资源的管理者

- 处理机管理
- 存储器管理
- 文件管理
- 设备管理

向上层提供服务

对于普通用户

- GUI图形界面
- 命令接口
    - 联机命令接口(说一句做一句)
    - 脱机命令接口(批处理)

对于软件和程序员

- 程序接口(系统调用)

对硬件机器的扩展

- 扩充机器(虚拟机)

特征

并发

- 并发
    - 指两个或多个事件在同一时间间隔内发生。
    - 这些事件宏观上是同时发生的，但微观上是交替发生的。
- 并行
    - 指两个或多个事件在同一时刻同时发生。
- 操作系统的并发性
    - 指计算机系统中"同时"运行着多个程序，这些程序宏观上看是同时运行着的，而微观上看是交替运行的。
    - 操作系统就是伴随着"多道程序技术"而出现的。因此，操作系统和程序并发是一起诞生的。

共享

- 共享即资源共享，是指系统中的资源可供内存中多个并发执行的进程共同使用。
- 并发和共享互为存在条件
- 资源共享方式
    - 互斥共享方式
        - 系统中的某些资源，虽然可以提供给多个进程使用，但一个时间段内只允许一个进程访问该资源。
    - 同时共享方式
        - 系统中的某些资源，允许一个时间段内由多个进程"同时"对它们进行访问。

虚拟

- 虚拟是指把一个物理上的实体变为若干个逻辑上的对应物。
- 物理实体(前者)是实际存在的，而逻辑上对应物(后者)是用户感受到的。
- 时分复用技术
    - 如虚拟存储器技术
- 空分复用技术
    - 如虚拟处理器、虚拟设备技术

异步

- 异步是指，在多道程序环境下，允许多个程序并发执行，但由于资源有限，进程的执行不是一贯到底的，而是走走停停，以不可预知的速度向前推进，这就是进程的异步性。
- 由于并发运行的程序会争抢着使用系统资源，而系统中的资源有限，因此进程的执行不是一贯到底的，而是走走停停的，以不可预知的速度向前推进。
- 如果失去了并发性，即系统只能串行地运行各个程序，那么每个程序的执行会一贯到底。只有系统拥有并发性，才有可能导致异步性。

发展与分类

- 手工操作阶段
    - 缺点：人机速度矛盾
- 批处理阶段
    - 单道批处理系统
        - 优点：缓解人机速度矛盾
        - 缺点：资源利用率低
    - 多道批处理系统
        - 优点：多道程序并发执行，资源利用率高
        - 缺点：不提供人机交互功能
- 分时操作系统
    - 优点：提供人机交互功能
    - 缺点：不能优先处理紧急任务
- 实时操作系统
    - 硬实时操作系统

        - 必须在绝对严格的规定时间内完成处理
    - 软实时操作系统
        - 能接受偶尔违反时间规定
- 网络操作系统
- 分布式操作系统
- 个人计算机操作系统

运行机制

程序运行原理

- 高级语言代码 -> 机器指令
- 程序运行的过程就是CPU执行指令的过程

两类程序

- 内核程序
- 应用程序

两类指令

- 特权指令
- 非特权指令

两种处理器状态

- 内核态/核心态/管态
- 用户态/目态
- CPU 中有一个寄存器叫 程序状态字寄存器(PSW)，其中有个二进制位，1表示"内核态"，0表示"用户态"

内核

- 内核(Kernel)是操作系统最核心最重要的部分
- 由很多内核程序组成操作系统的内核

如何变态

- 内核态 -> 用户态
    - 一条修改PSW的特权指令
- 用户态 -> 内核态
    - 由中断引起，硬件自动完成

中断和异常

中断的作用

- 让操作系统内核强行夺回CPU的控制权
- 使CPU从用户态变为内核态

中断的分类

- 内中断(异常)
    - 与当前执行的指令有关
    - 中断信号来自CPU内部
    - 陷阱、陷入(trap)
        - 由陷入指令引发，是应用程序故意引发的
    - 故障(fault)
        - 由错误条件引起的，可能被内核程序修复。内核程序修复故障后会把 CPU使用权还给应用程序，让它继续执行下去。
          如：缺页故障。
    - 终止(abort)
        - 由致命错误引起，内核程序无法修复该错误，因此一般不再将CPU使用权还给引发终止的应用程序，而是直接终止该应用程序。
          如：整数除0、非法使用特权指令
- 外中断
    - 与当前执行的指令无关
    - 中断信号来自CPU外部
    - 时钟中断
    - I/O中断

中断机制的基本实现原理

- 检查中断信号
    - 内中断：CPU在执行指令时会检查是否有异常发生
    - 外中断：每个指令周期末尾，CPU都会检查是否有外中断信号需要处理
- 找到相应的中断处理程序
    - 通过"中断向量表"实现

系统调用

定义

- 操作系统对应用程序/程序员提供的接口

系统调用和库函数的区别

- 有的库函数是对系统调用的进一步封装
- 有的库函数没有使用系统调用

按功能分类

- 设备管理
  - 完成设备的请求/释放/启动等功能
- 文件管理
  - 完成文件的读/写/创建/删除等功能
- 进程控制
  - 完成进程的创建/撤销/阻塞/唤醒等功能
- 进程通信
  - 完成进程之间的消息传递/信号传递等功能
- 内存管理
  - 完成内存的分配/回收等功能

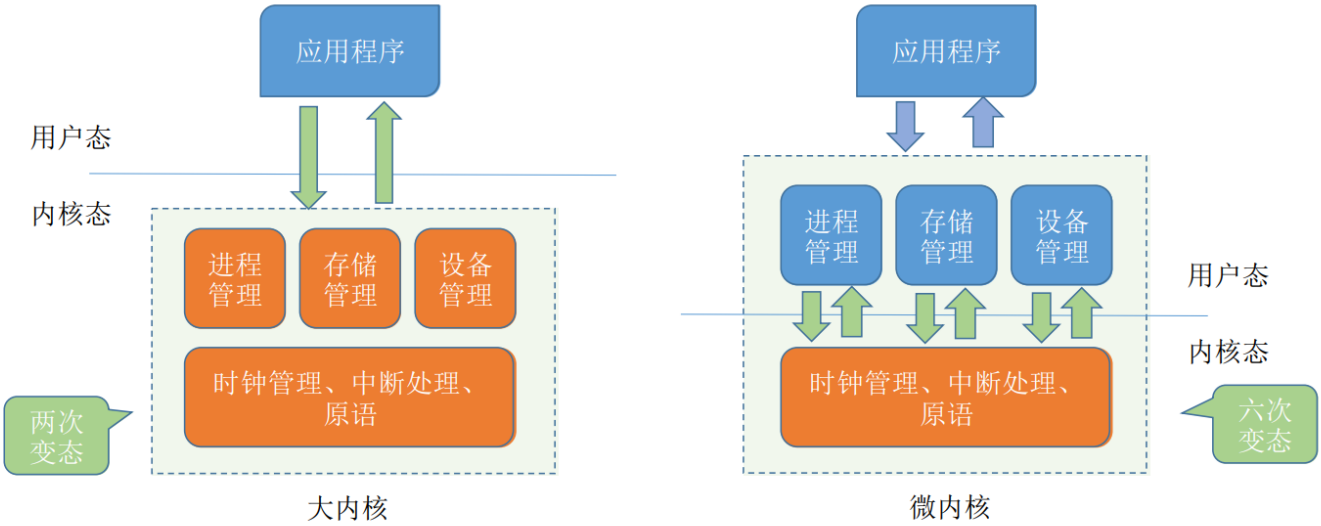
系统调用的过程

- 传参
- 陷入指令/Trap/访管
- 由操作系统内核程序处理系统调用请求
- 返回应用程序

其他

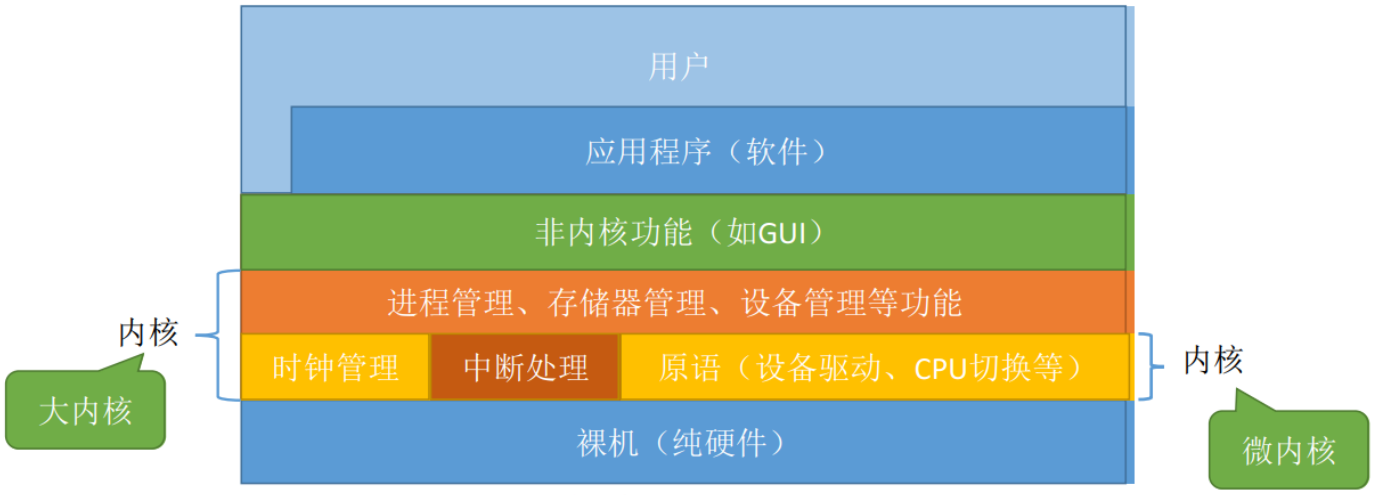
- 传递系统调用参数 -> 执行陷入指令(用户态) -> 执行相应的内请求核程序处理系统调用(核心态) -> 返回
- 陷入指令是在用户态执行的，执行陷入指令之后立即引发一个内中断，使CPU进入核心态
- 发出系统调用请求是在用户态，而对系统调用的相应处理在核心态下进行

操作系统的体系结构



一个故事：现在，应用程序想要请求操作系统的服务，这个服务的处理同时涉及到进程管理、存储管理、设备管理

注意：变态的过程是有成本的，要消耗不少时间，频繁地变态会降低系统性能



计算机系统的层次结构

注意：
操作系统内核需要运行在内核态
操作系统的非内核功能运行在用户态

操作系统结构			
	特性、思想	优点	缺点
分层结构	内核分多层，每层可单向调用更低一层提供的接口	1. 便于调试和验证，自底向上逐层调试验证 2. 易扩充和易维护，各层之间调用接口清晰固定	1. 仅可调用相邻低层，难以合理定义各层的边界 2. 效率低，不可跨层调用，系统调用执行时间长
模块化	将内核划分为多个模块，各模块之间相互协作。 内核 = 主模块+可加载内核模块 主模块：只负责核心功能，如进程调度、内存管理 可加载内核模块：可以动态加载新模块到内核，而无需重新编译整个内核	1. 模块间逻辑清晰易于维护，确定模块间接口后即可多模块同时开发 2. 支持动态加载新的内核模块（如：安装设备驱动程序、安装新的文件系统模块到内核），增强OS适应性 3. 任何模块都可以直接调用其他模块，无需采用消息传递进行通信，效率高	1. 模块间的接口定义未必合理、实用 2. 模块间相互依赖，更难调试和验证
宏内核（大内核）	所有的系统功能都放在内核里（大内核结构的OS通常也采用了"模块化"的设计思想）	1. 性能高，内核内部各种功能都可以直接相互调用	1. 内核庞大功能复杂，难以维护 2. 大内核中某个功能模块出错，就可能导致整个系统崩溃
微内核	只把中断、原语、进程通信等最核心的功能放入内核。进程管理、文件管理、设备管理等功能以用户进程的形式运行在用户态	1. 内核小功能少、易于维护，内核可靠性高 2. 内核外的某个功能模块出错不会导致整个系统崩溃	1. 性能低，需要频繁的切换 用户态/核心态。 用户态下的各功能模块不可以直接相互调用，只能通过内核的"消息传递"来间接通信 2. 用户态下的各功能模块不可以直接相互调用，只能通过内核的"消息传递"来间接通信
外核（exokernel）	内核负责进程调度、进程通信等功能，外核负责为用户进程分配未经抽象的硬件资源，且由外核负责保证资源使用安全	1. 外核可直接给用户进程分配"不虚拟、不抽象"的硬件资源，使用户进程可以更灵活的使用硬件资源 2. 减少了虚拟硬件资源的"映射层"，提升效率	1. 降低了系统的一致性 2. 使系统变得更复杂

大内核

- 将操作系统的主要功能模块都作为系统内核，运行在核心态
- 优点：高性能
- 缺点：内核代码庞大，结构混乱，难以维护

微内核

- 只把最基本的功能保留在内核
- 优点：内核功能少，结构清晰，方便维护
- 缺点：需要频繁地在核心态和用户态之间切换，性能低

操作系统内核

时钟管理

- 实现计时功能

中断处理

- 负责实现中断机制

原语

- 是一种特殊的程序
- 处于操作系统最底层，是最接近硬件的部分
- 这种程序的运行具有原子性，其运行只能一气呵成，不可中断
- 运行时间较短、调用频繁

对系统资源进行管理的功能

- 进程管理
- 存储器管理
- 设备管理

程序运行时内存映像与地址空间

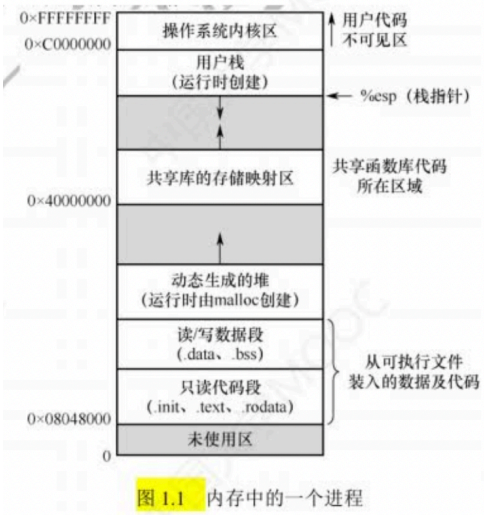


图 1.1 内存中的一个进程

1. 进程的内存映像与地址空间

不同于存放在硬盘上的可执行程序文件，当一个程序调入内存运行时就构成了进程的内存映像，一个进程的内存映像一般有几个要素：

- 代码段：即程序的二进制代码，代码段是只读的，可以被多个进程共享。
- 数据段：即程序运行时加工处理对象，包括全局变量和静态变量。
- 进程控制块（PCB）：存放在系统区。操作系统通过 PCB 来控制和管理进程。
- 堆：用来存放动态分配的变量。
- 栈：用来实现函数调用。

代码段和数据段在程序调入内存时就指定了大小，而堆和栈不一样。当调用像 malloc 和 free 这样的 C 标准库函数时，堆可以在运行时动态地扩展和收缩。用户栈在程序运行期间也可以动态地扩展和收缩，每次调用一个函数，栈就会增长；从一个函数返回时，栈就会收缩。

图 1.1 是一个进程在内存中的映像。其中，共享库用来存放进程用到的共享函数库代码，如 printf()函数等。在只读代码段中，.init 是程序初始化时调用的_init 函数；.text 是用户程序的机器代码；.rodata 是只读数据。在读/写数据段中，.data 是已初始化的全局变量和静态变量；.bss 是未初始化及所有初始化为 0 的全局变量和静态变量。

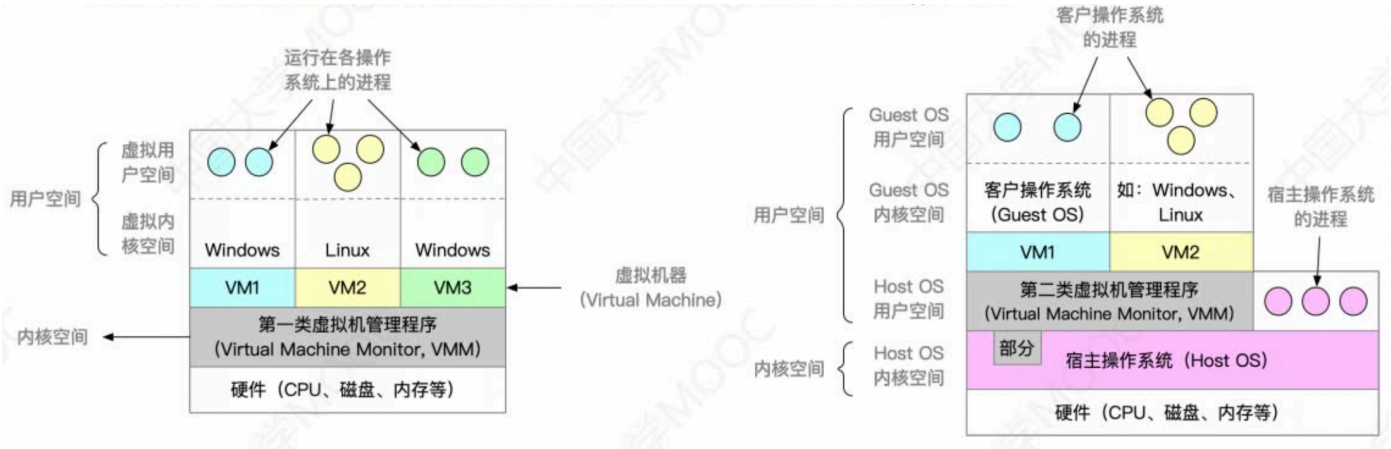
每个进程都有一个独立的地址空间，这个地址空间的地址为虚拟地址，对于 32 位系统，虚拟地址空间的范围为 $0 \sim 2^{32}-1$。进行在运行时，看到和使用的地址都是虚拟地址。

系统中还有一个物理地址空间，对应于系统中物理内存的所有可寻址单元。

操作系统通过内存管理部件（MMU）将进程使用的虚拟地址转换为物理地址。进程使用虚拟内存中的地址，操作系统在相关硬件的协助下，把它“转换”成真正的物理地址。虚拟地址通过页表映射到物理内存，页表由操作系统维护并被处理器引用。

虚拟机

使用虚拟化技术，将一台物理机器虚拟化为多台虚拟机器（VirtualMachine,VM），每个虚拟机器都可以独立运行一个操作系统。



两类虚拟机管理程序（VMM）的对比		
	第一类VMM	第二类VMM
对物理资源的控制权	直接运行在硬件之上，能直接控制和分配物理资源	运行在Host OS之上，依赖于Host OS为其分配物理资源
资源分配方式	在安装Guest OS时，VMM要在原本的硬盘上自行分配存储空间，类似于"外核"的分配方式，分配未经抽象的物理硬件	GuestOS 拥有自己的虚拟磁盘，该盘实际上是 Host OS 文件系统中的一个大文件。GuestOS分配到的内存是虚拟内存
性能	性能更好	性能更差，需要HostOS作为"中介"
可支持的虚拟机数量	更多，不需要和 Host OS 竞争资源，相同的硬件资源可以支持更多的虚拟机	更少，Host OS 本身需要使用物理资源，Host OS 上运行的其他进程也需要物理资源
虚拟机的可迁移性	更差	更好，只需导出虚拟机镜像文件即可迁移到另一台 HostOS 上，商业化应用更广泛
运行模式	第一类VMM运行在最高特权级（Ring 0），可以执行最高特权的指令。	第二类VMM部分运行在用户态、部分运行在内核态。GuestOS 发出的系统调用会被 VMM 截获，并转化为 VMM 对 HostOS 的系统调用

2 进程管理

基础

概念

- 进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位

进程控制块(PCB)

进程描述信息

- 进程标识符PID
- 用户标识符UID

进程控制和管理信息

- CPU、磁盘、网络流量使用情况统计...
- 进程当前状态：就绪态/阻塞态/运行态...

资源分配清单

- 正在使用哪些文件
- 正在使用哪些内存区域
- 正在使用哪些I/O设备

处理机相关信息

- 如PSW、PC等等各种寄存器的值(用于实现进程切换)

进程的组成

PCB

- 进程描述信息
- 进程控制和管理信息
- 资源分配清单
- 处理机相关信息

程序段

- 程序的代码(指令序列)

数据段

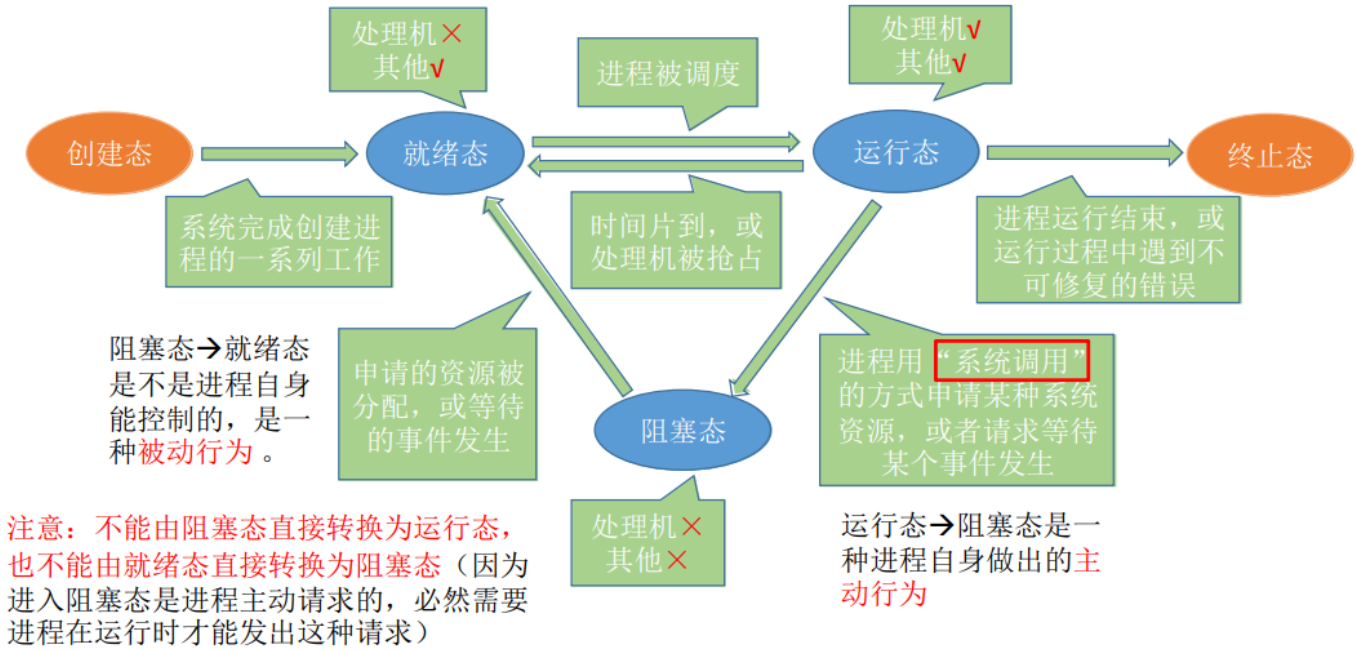
- 运行过程中产生的各种数据(如程序中定义的变量)

进程的特征

- 动态性
  - 进程是程序的一次执行过程，是动态地产生、变化消亡的
- 并发性
  - 内存中有多个进程实体，各进程可并发执行
- 独立性
  - 进程是能独立运行、独立获得资源、独立接受调度的基本单位
- 异步性
  - 各进程按各自独立的、不可预知的速度向前推进，操作系统要提供"进程同步机制"来解决异步问题
- 结构性
  - 每个进程都会配置一个PCB。结构上看，进程由程序段、数据段、PCB组成

进程的状态与转换

状态



- 运行态(Running)
  - CPU√ 其他所需资源√
  - 占有CPU，并在CPU运行
  - 单CPU情况下，同一时刻只会有一个进程处于运行态，多核CPU情况下，可能有多个进程处于运行态
- 就绪态(Ready)
  - CPU× 其他所需资源√
  - 已经具备运行条件，但由于没有空闲CPU而暂时不能运行
- 阻塞态(Waiting/Blocked，又称：等待态)
  - CPU× 其他所需资源×
  - 因等待某一事件而暂时不能运行

- 创建态(New，又称：新建态)
  - 进程正在被创建，操作系统为进程分配资源、初始化PCB
- 终止态(Terminated，又称：结束态)
  - 进程正在从系统中撤销，操作系统会回收进程拥有的资源、撤销PCB

进程的组织方式

- 链接方式
  - 按照进程状态将PCB分为多个队列
  - 操作系统持有指向各个队列的指针
- 索引方式
  - 根据进程状态的不同，建立几张索引表
  - 操作系统持有指向各个索引表的指针

进程控制

概念

- 进程控制就是要实现进程状态的转换
- 进程控制用原语实现
    - 原语用关/开中断实现
    - 原语是一种特殊的程序
    - 原语的执行具有原子性，执行过程必须一气呵成，不可中断

相关原语

进程的创建

- 操作系统创建一个进程时使用的原语

创建原语

- 申请空白PCB
- 为新进程分配所需资源
- 初始化PCB
- 将PCB插入就绪队列(创建态 -> 就绪态)

引起进程创建的事件

- 用户登录
    - 分时系统中，用户登录成功，系统会为其建立一个新的进程
- 作业调度
    - 多道批处理系统中，有新的作业放入内存时，会为其建立一个新的进程
- 提供服务
    - 用户向操作系统提出某些请求时，会新建一个进程处理该请求
- 应用请求
    - 由用户进程主动请求创建一个子进程

进程的终止

- 就绪态/阻塞态/运行态 -> 终止态 -> 无

撤销原语

- 从PCB集合中找到终止进程的PCB
- 若进程正在运行，立即剥夺CPU，将CPU分配给其他进程
- 终止其所有子进程(进程间的关系是树形结构)
- 将该进程拥有的所有资源归还给父进程或操作系统
- 删除PCB

引起进程终止的事件

- 正常结束
  - 进程自己请求终止(exit系统调用)
- 异常结束
  - 整数除以0、非法使用特权指令，然后被操作系统强行杀掉
- 外界干预
  - 任务管理器用户选择杀掉进程

进程的阻塞和唤醒

- 阻塞原语、唤醒原语必须成对使用

进程的阻塞

- 运行态 -> 阻塞态
- 阻塞原语
  - 找到要阻塞的进程对应的PCB
  - 保护进程运行现场，将PCB状态信息设置为阻塞态，暂时停止进程运行
  - 将PCB插入相应事件的等待队列
- 引起进程阻塞的事件
  - 需要等待系统分配某种资源
  - 需要等待相互合作的其他进程完成工作

进程的唤醒

- 阻塞态 -> 就绪态
- 唤醒原语
  - 在事件等待队列中找到PCB
  - 将PCB从等待队列移除，设置进程为就绪态
  - 将PCB插入就绪队列，等待被调度
- 引起进程唤醒的事件
  - 等待的事件发生
  - 因何事阻塞，就应由何事唤醒

进程的切换

- 运行态 -> 阻塞态/就绪态
- 就绪态 -> 运行态

切换原语

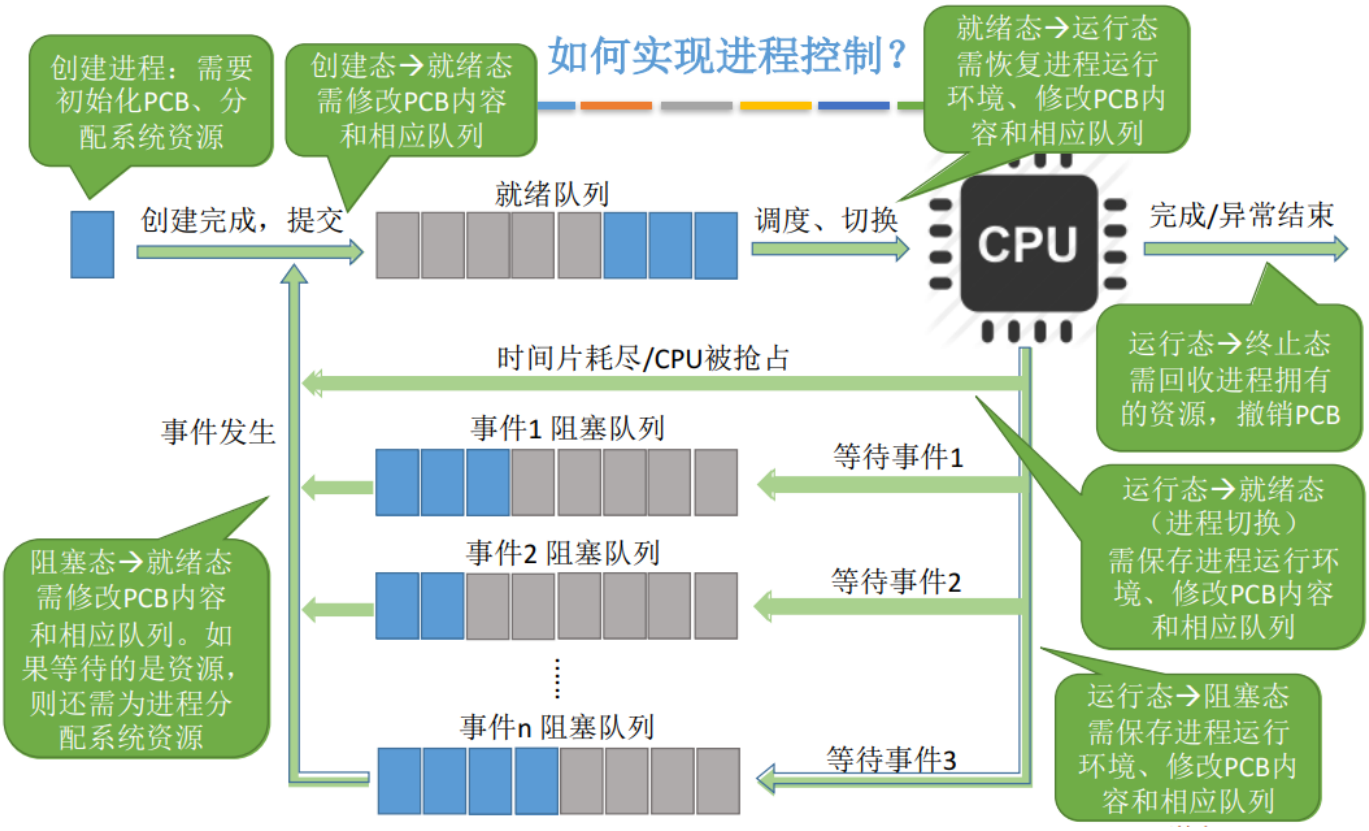
- 将运行环境信息存入PCB
- PCB移入相应队列
- 选择另一个进程执行，并更新其PCB
- 根据PCB恢复新进程所需的运行环境

引起进程切换的事件

- 当前进程时间片到
- 有更高优先级的进程到达
- 当前进程主动阻塞
- 当前进程终止

进程控制原语总结

- 更新PCB中的信息
  - 所有的进程控制原语一定都会修改进程状态标志
  - 剥夺当前运行进程的CPU使用权必然需要保存其运行环境
  - 某进程开始运行前必然要恢复期运行环境
- 将PCB插入合适的队列
- 分配/回收资源



进程通信

- 进程是分配系统资源的单位(包括内存地址空间)，因此各进程拥有的内存地址空间相互独立。
- 为了保证安全，一个进程不能直接访问另一个进程的地址空间。
- 但是进程之间的信息交换又是必须实现的。为了保证进程间的安全通信，操作系统提供了一些方法。

共享存储

- 设置一个共享空间
- 要互斥的访问共享空间
- 两种方式

- 基于数据结构的共享(低级)
- 基于存储区的共享(高级)

管道通信

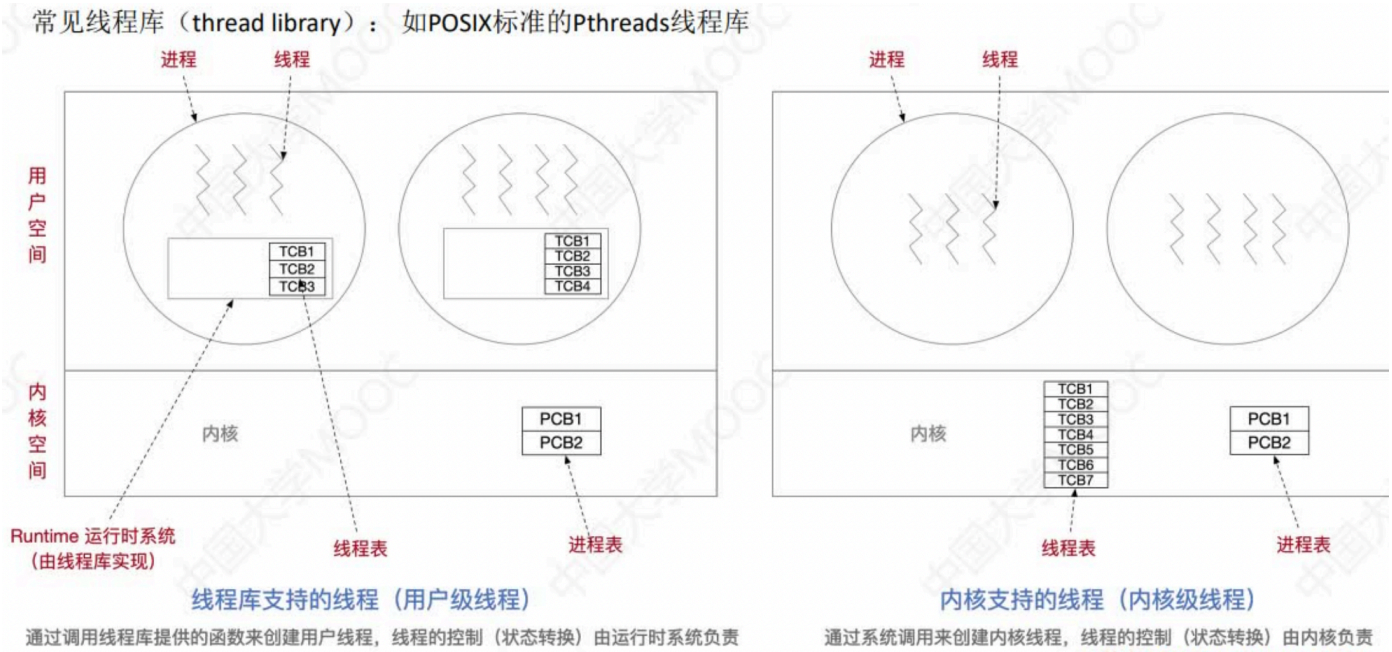
- 管道是指用于连接读写进程的一个共享文件，又名pipe文件。其实就是在内存中开辟一个大小固定的缓冲区
- 管道只能采用半双工通信，某一时间段内只能实现单向的传输。如果要实现双向同时通信，则需要设置两个管道。
- 各进程要互斥地访问管道。
- 数据以字符流的形式写入管道，当管道写满时，写进程的write()系统调用将被阻塞，等待读进程将数据取走。
  当读进程将数据全部取走后，管道变空，此时读进程的read()系统调用将被阻塞。
- 如果没写满，就不允许读。如果没读空，就不允许写。
- 数据一旦被读出，就从管道中被抛弃，这就意味着读进程最多只能有一个，否则可能会有读错数据的情况。

消息传递

- 进程间的数据交换以格式化的消息(Message)为单位。
- 进程通过操作系统提供的"发送消息/接收消息"两个原语进行数据交换。
- 消息头包括：发送进程ID、接受进程ID、消息类型、消息长度等格式化的信息
- 两种方式
  - 直接通信方式
    - 消息直接挂到接收方的消息队列里
  - 间接(信箱)通信方式
    - 消息先发到中间体(信箱)

线程/多线程模型

线程的实现



变化

- 资源分配、调度
  - 传统进程机制中，进程是资源分配、调度的基本单位
  - 引入线程后，进程是资源分配的基本单位，线程是调度的基本单位
- 并发性
  - 传统进程机制中，只能进程间并发
  - 引入线程后，各线程间也能并发，提升了并发度
- 系统开销
  - 传统的进程间并发，需要切换进程的运行环境，系统开销很大
  - 线程间并发，如果是同一进程内的线程切换，则不需要切换进程环境，系统开销小
  - 引入线程后，并发所带来的系统开销减小

线程的属性

- 线程是处理机调度的单位
- 多CPU计算机中，各个线程可占用不同的CPU
- 每个线程都有一个线程ID、线程控制块(TCB)
- 线程也有就绪、阻塞、运行三种基本状态
- 线程几乎不拥有系统资源
- 同一进程的不同线程间共享进程的资源
- 由于共享内存地址空间，同一进程中的线程间通信甚至无需系统干预
- 同一进程中的线程切换，不会引起进程切换
- 不同进程中的线程切换，会引起进程切换
- 切换同进程内的线程，系统开销很小
- 切换进程，系统开销较大

线程的实现方式

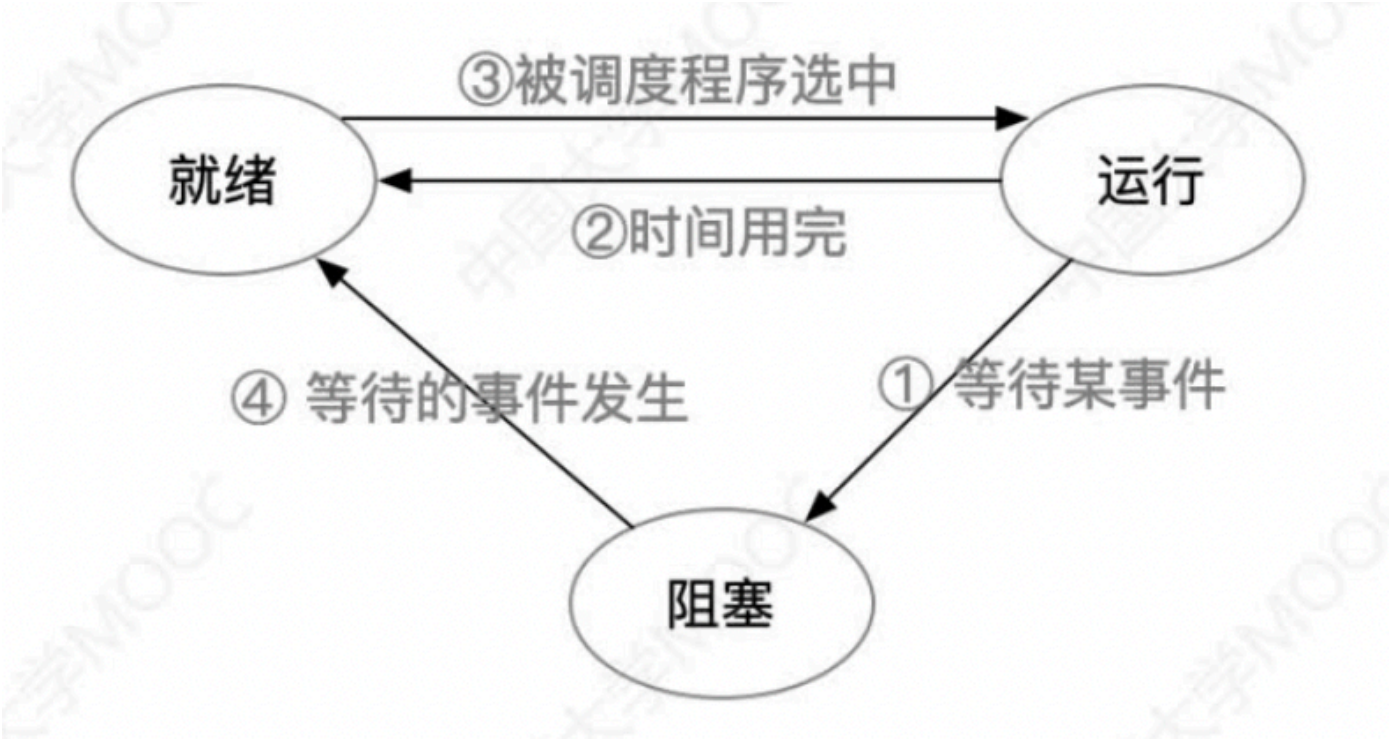
- 用户级线程
  - 从用户视角能看到的线程，由线程库实现
- 内核级线程
  - 从操作系统视角看到的线程
  - 由操作系统实现内核级线程才是处理机分配的单位
- 组合方式
  - 上述两种方式的结合

多线程模型

- 一对一模型
  - 一个用户级线程映射到一个内核级线程
  - 优点：各个线程可分配到多核处理机并行执行，并发度高
  - 缺点：线程管理都需要操作系统支持，开销大
- 多对一模型
  - 多个用户级线程映射到一个内核级线程
  - 优点：线程管理开销小效率高

- 缺点：一个线程阻塞会导致整个进程都被阻塞(并发度低)
- 多对多模型
  - n个用户级线程映射到m个内核级线程(n≥m)
  - 集二者之所长

线程的状态与转换



线程的组织与控制

处理机调度

概念

- 按某种算法选择一个进程将CPU分配给它

三个层次

	要做什么	调度发生在..	发生频率	对进程状态的影响
高级调度（作业调度）	按照某种规则，从后备队列中选择合适的作业将其调入内存，并为其创建进程	外存→内存（面向作业）	最低	无→创建态→就绪态
中级调度（内存调度）	按照某种规则，从挂起队列中选择合适的进程将其数据调回内存	外存→内存（面向进程）	中等	挂起态→就绪态（阻塞挂起→阻塞态）
低级调度（进程调度）	按照某种规则，从就绪队列中选择一个进程为其分配处理机	内存→CPU	最高	就绪态→运行态

- 高级调度(作业调度)
  - 按照某种规则，从后备队列中选择合适的作业将其调入内存，并为其创建进程
- 中级调度(内存调度)
  - 按照某种规则，从挂起队列中选择合适的进程将其数据调回内存
- 低级调度(进程调度)
  - 按照某种规则，从就绪队列中选择一个进程为其分配处理机

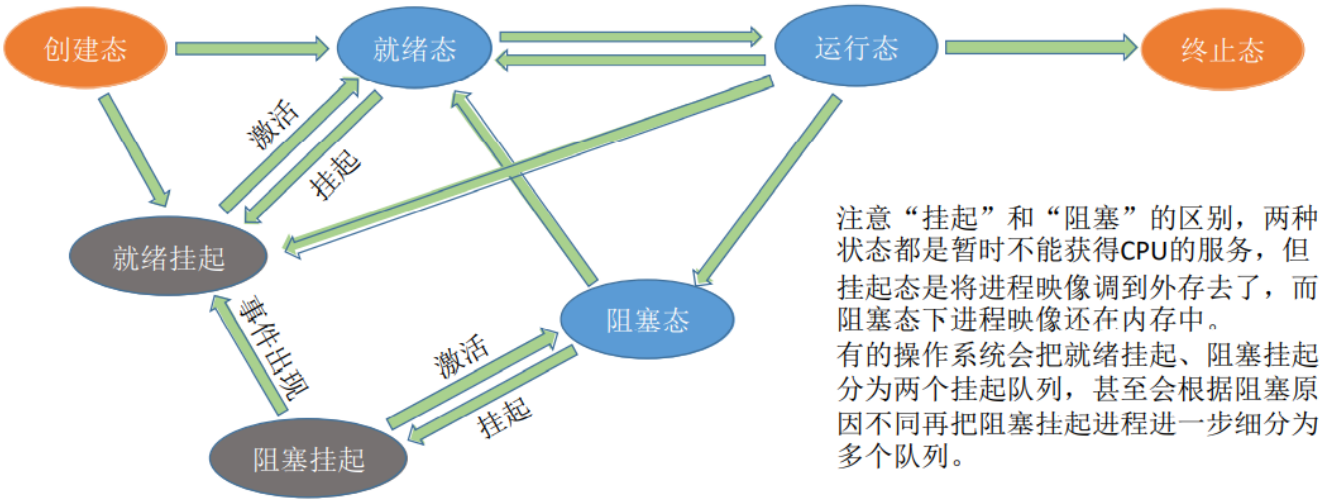
三层调度的联系、对比

- 高级调度
  - 外存 -> 内存(面向作业)
  - 发生频率：最低
- 中级调度
  - 外存 -> 内存(面向进程)
  - 发生频率：中等
- 低级调度
  - 内存 -> CPU
  - 发生频率：最高

补充

- 暂时调到外存等待的进程状态为挂起状态(挂起态，suspend)
- 挂起态又可以进一步细分为就绪挂起、阻塞挂起两种状态
- 七状态模型：在五状态模型的基础上加入了"就绪挂起"和"阻塞挂起"两种状态

五状态模型 → 七状态模型



注意“挂起”和“阻塞”的区别，两种状态都是暂时不能获得CPU的服务，但挂起态是将进程映像调到外存去了，而阻塞态下进程映像还在内存中。
有的操作系统会把就绪挂起、阻塞挂起分为两个挂起队列，甚至会根据阻塞原因不同再把阻塞挂起进程进一步细分为多个队列。

进程调度

时机

什么时候可以进行进程调度

- 主动放弃
  - 进程正常终止
  - 运行过程中发生异常而终止
  - 主动阻塞(如等待I/O)
- 被动放弃
  - 分给进程的时间片用完
  - 有更紧急的事情需要处理(如I/O中断)
  - 有更高优先级的进程进入就绪队列

什么时候不能进行进程调度

- 在处理中断的过程中
- 进程在**操作系统内核程序临界区**中
- 原子操作过程中(原语)

切换过程

- **狭义的进程调度**指的是从就绪队列中选中一个要运行的进程。
  (这个进程可以是刚刚被暂停执行的进程，也可能是**另一个进程**，后一种情况就需要**进程切换**)
- **进程切换**是指一个进程让出处理机，由另一个进程占用处理机的过程。
- **广义的进程调度**包含了选择一个进程和进程切换两个步骤。
- 切换过程
  - 对原来运行进程各种数据的保存
  - 对新的进程各种数据的恢复
- 进程调度、切换是有代价的，并不是调度越频繁，并发度就越高

方式

- 非剥夺调度方式(非抢占式)
    - 只能由当前运行的进程主动放弃CPU（阻塞或者退出时才会触发调度程序）
- 剥夺调度方式(抢占式)
    - 可由操作系统剥夺当前进程的CPU使用权（每个时钟中断或k个时钟中断会触发调度程序）

调度器/调度程序

就绪与运行的状态转换通过调度程序决定。

- 不支持内核级线程的操作系统，调度程序的处理对象是进程。
- 支持内核级线程的操作系统，调度程序的处理对象是内核线程。

闲逛进程

没有其他就绪进程时，运行闲逛进程。

闲逛进程的特性：

- 优先级最低
- 可以是0地址指令，占一个完整的指令周期（指令周期末尾例行检查中断）
- 能耗低

内核级线程与用户级线程调度

用户级线程

线程切换代价：低，不需要切换完整的上下文。

并发/并行度：低，一个阻塞整个进程阻塞。

内核级线程

线程切换代价：高，需转为内核态，切换完整的上下文。

并发/并行度：高，一个阻塞其他线程仍可运行。

上下文及切换机制

进程的内容（被该进程的各线程共享）	每个线程独有的内容
地址空间 全局变量 打开文件	程序计数器PC 寄存器 堆栈 状态

红字部分都是进程/线程切换时要保存/恢复的上下文

进程切换导致的地址空间切换代价巨大：
①保存/恢复页表寄存器
②TLB全部失效
③Cache全部失效，有可能要Cache写回（write-back）
④新进程运行初期可能缺页率高（需要磁盘I/O）

调度算法的评价指标

CPU利用率

- 或者计算某设备的利用率
- 利用率 $=\frac{\text{忙碌的时间}}{\text{总时间}}$

系统吞吐量

- 单位时间内完成作业的数量
- 系统吞吐量 $=\frac{\text{总共完成了多少道作业}}{\text{总共花了多少时间}}$

周转时间

- 周转时间 = 作业完成时间 - 作业提交时间
- 平均周转时间 $=\frac{\text{各作业周转时间之和}}{\text{作业数}}$
- 带权周转时间 $=\frac{\text{作业周转时间}}{\text{作业实际周转时间}}$
- 平均带权周转时间 $=\frac{\text{各作业带权周转时间之和}}{\text{作业数}}$

等待时间

- 对于进程来说，等待时间就是指进程建立后等待被服务的时间之和，在等待I/O完成的期间其实进程也是在被服务的，所以不计入等待时间。
- 对于作业来说，不仅要考虑建立进程后的等待时间，还要加上作业在外存后备队列中等待的时间。
- 进程/作业等待被服务的时间之和
- 平均等待时间即各个进程/作业等待时间的平均值

响应时间

- 从用户提交请求到首次产生响应所用的时间

调度算法

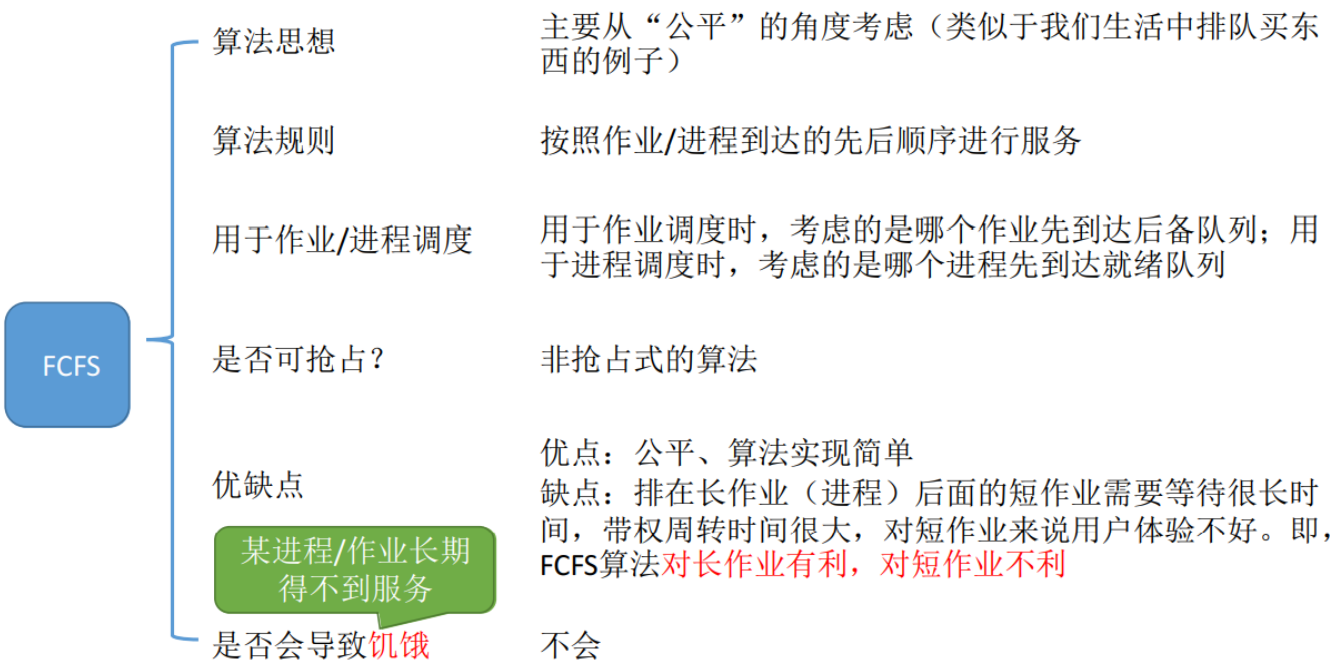
算法	思想&规则	可抢占？	优点	缺点	考虑到等待时间&运行时间？	会导致饥饿？
FCFS	自己回忆	非抢占式	公平；实现简单	对短作业不利	等待时间√ 运行时间×	不会
SJF/SPF	自己回忆	默认为非抢占式，也有SJF的抢占式版本最短剩余时间优先算法（SRTN）	“最短的”平均等待/周转时间；	对长作业不利，可能导致饥饿；难以做到真正的短作业优先	等待时间× 运行时间√	会
HRRN	自己回忆	非抢占式	上述两种算法的权衡折中，综合考虑的等待时间和运行时间		等待时间√ 运行时间√	不会

注：这几种算法主要关心对用户的公平性、平均周转时间、平均等待时间等评价系统整体性能的指标，但是不关心“响应时间”，也并不区分任务的紧急程度，因此对于用户来说，交互性很糟糕。因此这三种算法一般适合用于早期的批处理系统，当然，FCFS算法也常结合其他的算法使用，在现在也扮演着很重要的角色。

算法	思想&规则	可抢占？	优点	缺点	会导致饥饿？	补充
时间片轮转		抢占式	公平，适用于分时系统	频繁切换有开销，不区分优先级	不会	时间片太大或太小有何影响？
优先级调度		有抢占式的，也有非抢占式的。注意做题时的区别	区分优先级，适用于实时系统	可能导致饥饿	会	动态/静态优先级。 各类型进程如何设置优先级？如何调整优先级？
多级反馈队列	较复杂，注意理解	抢占式	平衡优秀666	一般不说它有缺点，不过可能导致饥饿	会	

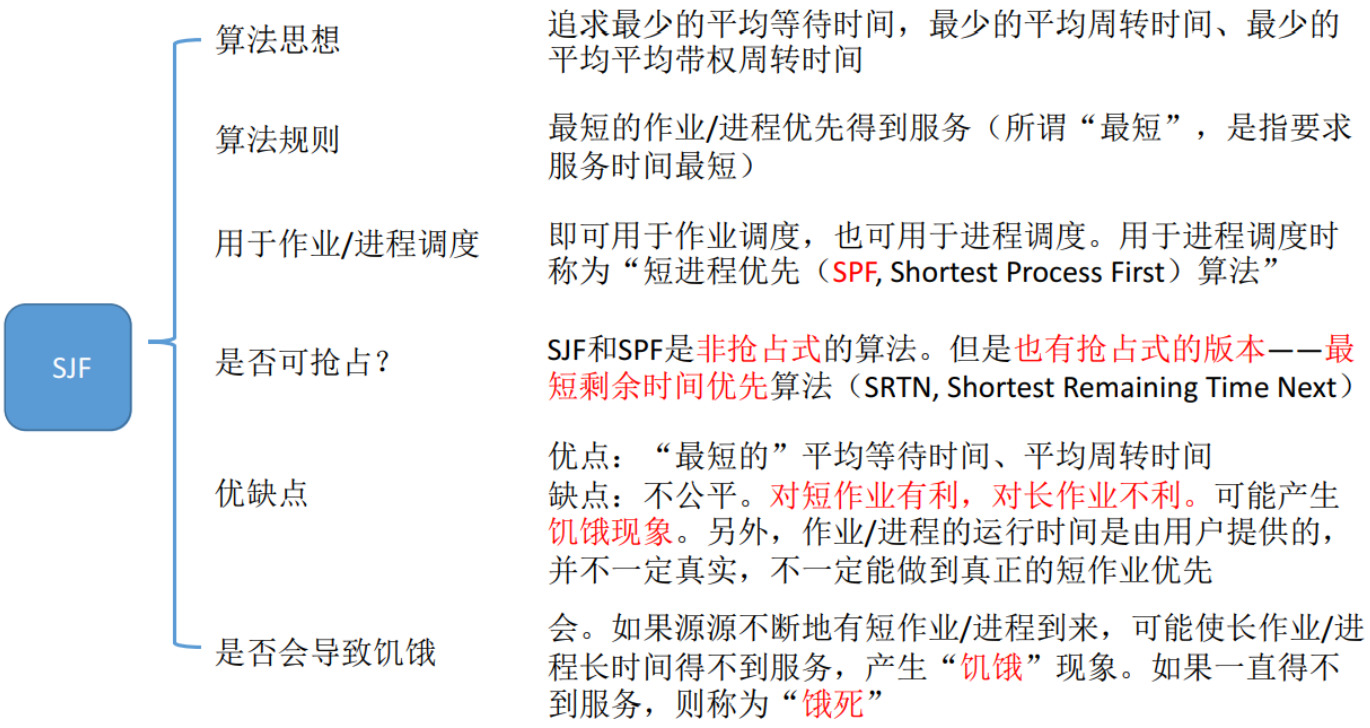
注：比起早期的批处理操作系统来说，由于计算机造价大幅降低，因此之后出现的交互式操作系统（包括分时操作系统、实时操作系统等）更注重系统的响应时间、公平性、平衡性等指标。而这几种算法恰好也能较好地满足交互式系统的需求。因此这三种算法适合用于交互式系统。（比如UNIX使用的就是多级反馈队列调度算法）

先来先服务(FCFS)



- 先来先服务调度算法：按照到达的先后顺序调度，事实上就是等待时间越久的越优先得到服务。

短作业优先(SJF)



- 短作业/进程优先调度算法：每次调度时选择当前已到达且运行时间最短的作业/进程。
- 最短剩余时间优先算法：每当有进程加入就绪队列改变时就需要调度，如果新到达的进程剩余时间比当前运行的进程剩余时间更短，则由新进程抢占处理机，当前运行进程重新回到就绪队列。另外，当一个进程完成时也需要调度

- 抢占式的短作业优先算法又称"最短剩余时间优先算法(SRTN)"

- 最短剩余时间优先算法：每当有进程加入就绪队列改变时就需要调度，如果新到达的进程剩余时间比当前运行的进程剩余时间更短，则由新进程抢占处理机，当前运行进程重新回到就绪队列。另外，当一个进程完成时也需要调度

注意细节

1. 如果题目中未特别说明，所提到的“短作业/进程优先算法”默认是非抢占式的

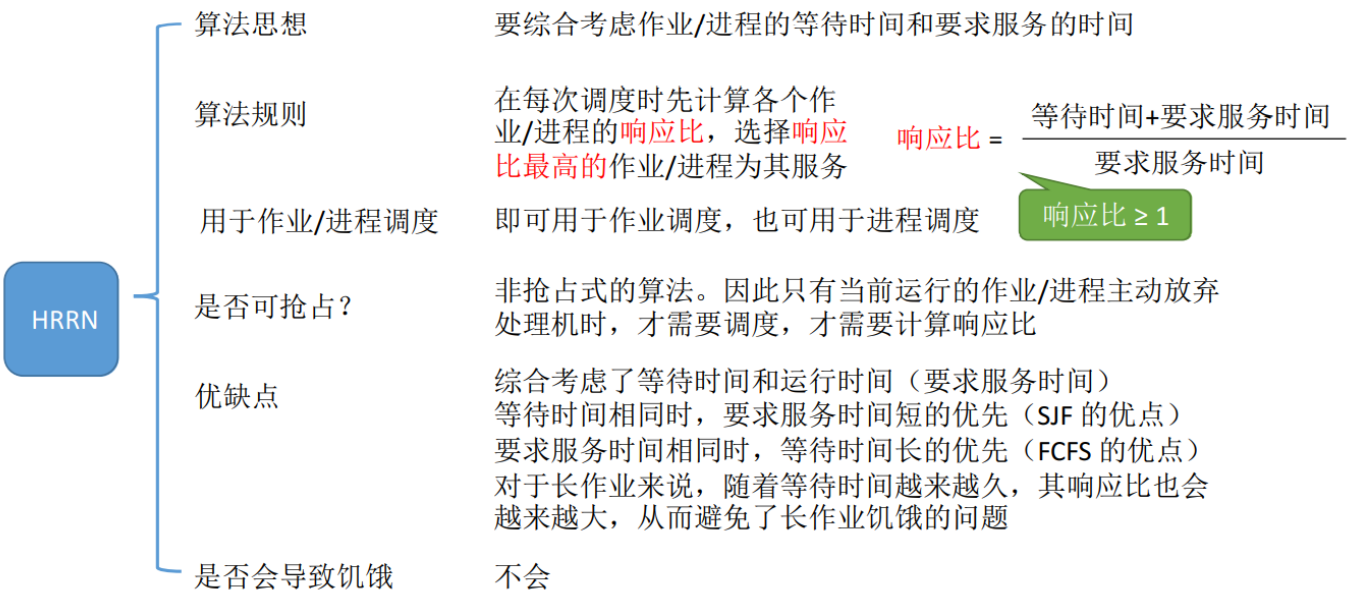
2. 很多书上都会说“SJF 调度算法的平均等待时间、平均周转时间最少”
严格来说，这个表述是错误的，不严谨的。之前的例子表明，最短剩余时间优先算法得到的平均等待时间、平均周转时间还要更少
应该加上一个条件“在所有进程同时可运行时，采用SJF调度算法的平均等待时间、平均周转时间最少”；
或者说“在所有进程都几乎同时到达时，采用SJF调度算法的平均等待时间、平均周转时间最少”；

如果不加上述前提条件，则应该说“抢占式的短作业/进程优先调度算法（最短剩余时间优先，SRNT算法）的平均等待时间、平均周转时间最少”

3. 虽然严格来说，SJF的平均等待时间、平均周转时间并不一定最少，但相比于其他算法（如 FCFS），SJF依然可以获得较少的平均等待时间、平均周转时间

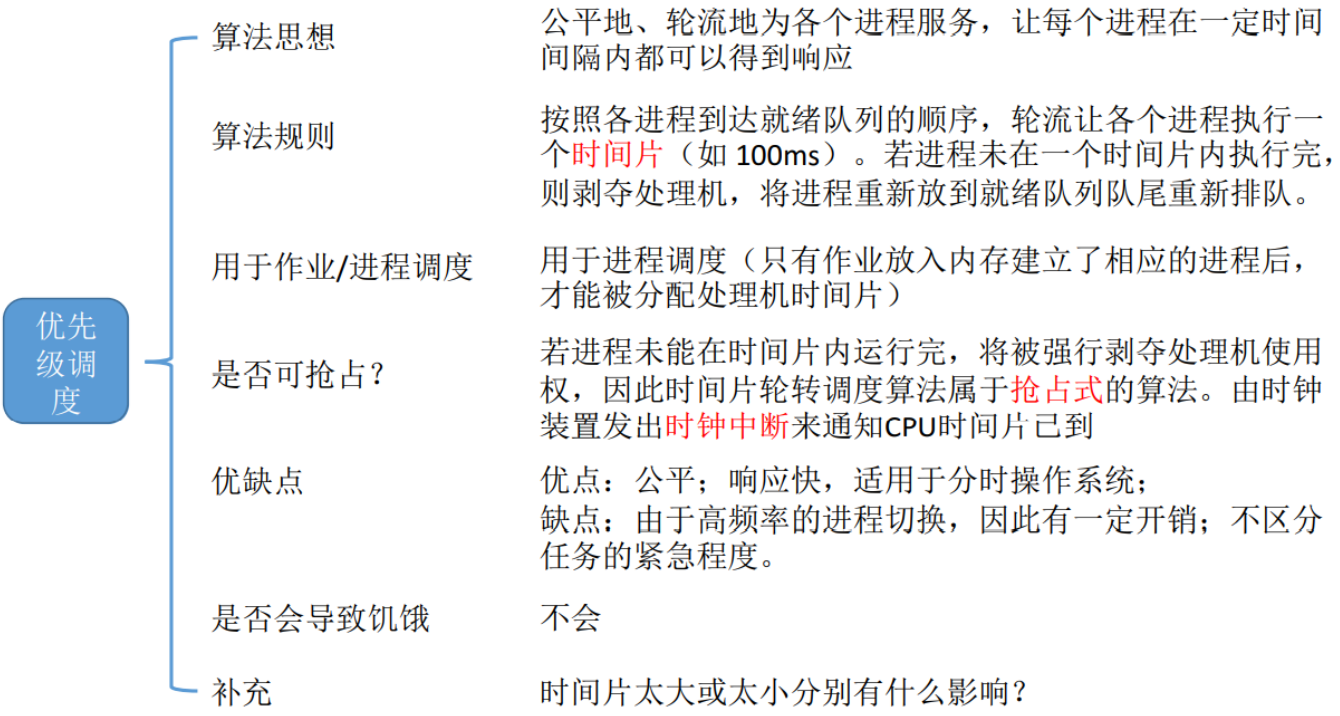
4. 如果选择题中遇到“SJF 算法的平均等待时间、平均周转时间最少”的选项，那最好判断其他选项是不是有很明显的错误，如果没有更合适的选项，那也应该选择该选项

高响应比优先(HRRN)



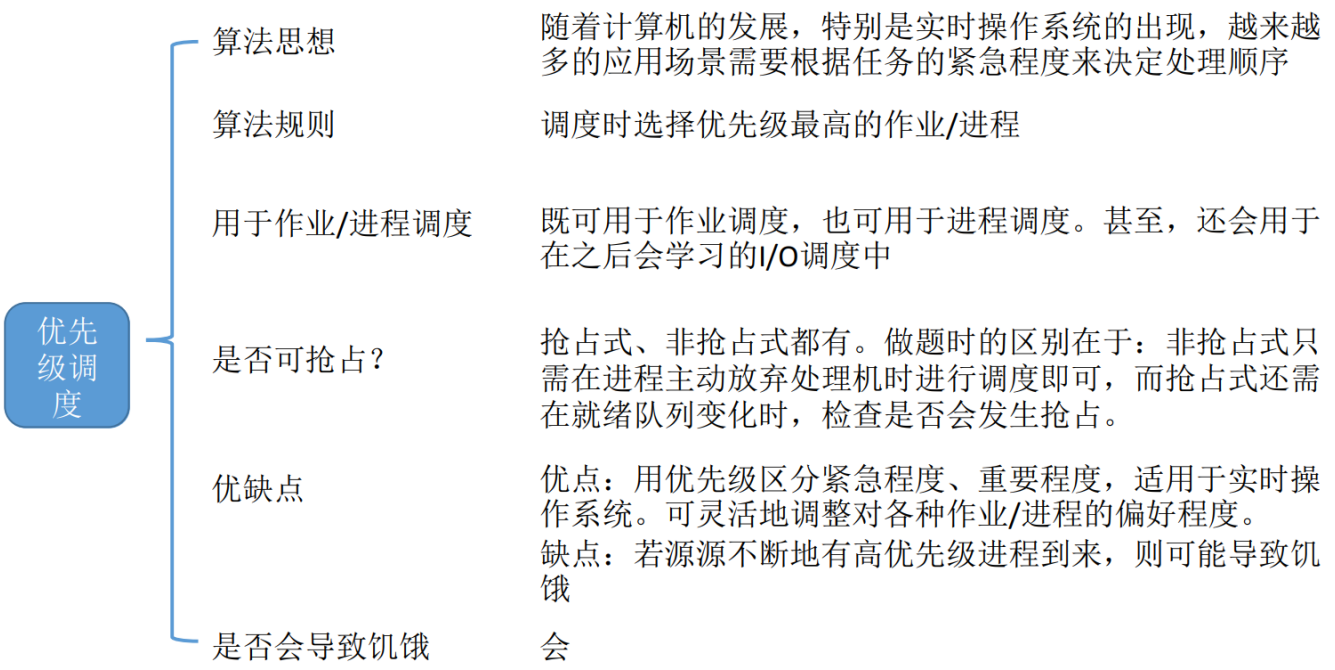
- 高响应比优先算法：非抢占式的调度算法，只有当前运行的进程主动放弃CPU时(正常/异常完成，或主动阻塞)，才需要进行调度，调度时计算所有就绪进程的响应比，选响应比最高的进程上处理机。

时间片轮转(RR)



- 时间片轮转调度算法：轮流让就绪队列中的进程依次执行一个时间片(每次选择的都是排在就绪队列队头的进程)
- 如果时间片太大，使得每个进程都可以在一个时间片内就完成，则时间片轮转调度算法退化为先来先服务调度算法，并且会增大进程响应时间。因此时间片不能太大。
- 另一方面，进程调度、切换是有时间代价的(保存、恢复运行环境)，因此如果时间片太小，会导致进程切换过于频繁，系统会花大量的时间来处理进程切换，从而导致实际用于进程执行的时间比例减少。可见时间片也不能太小。

优先级调度



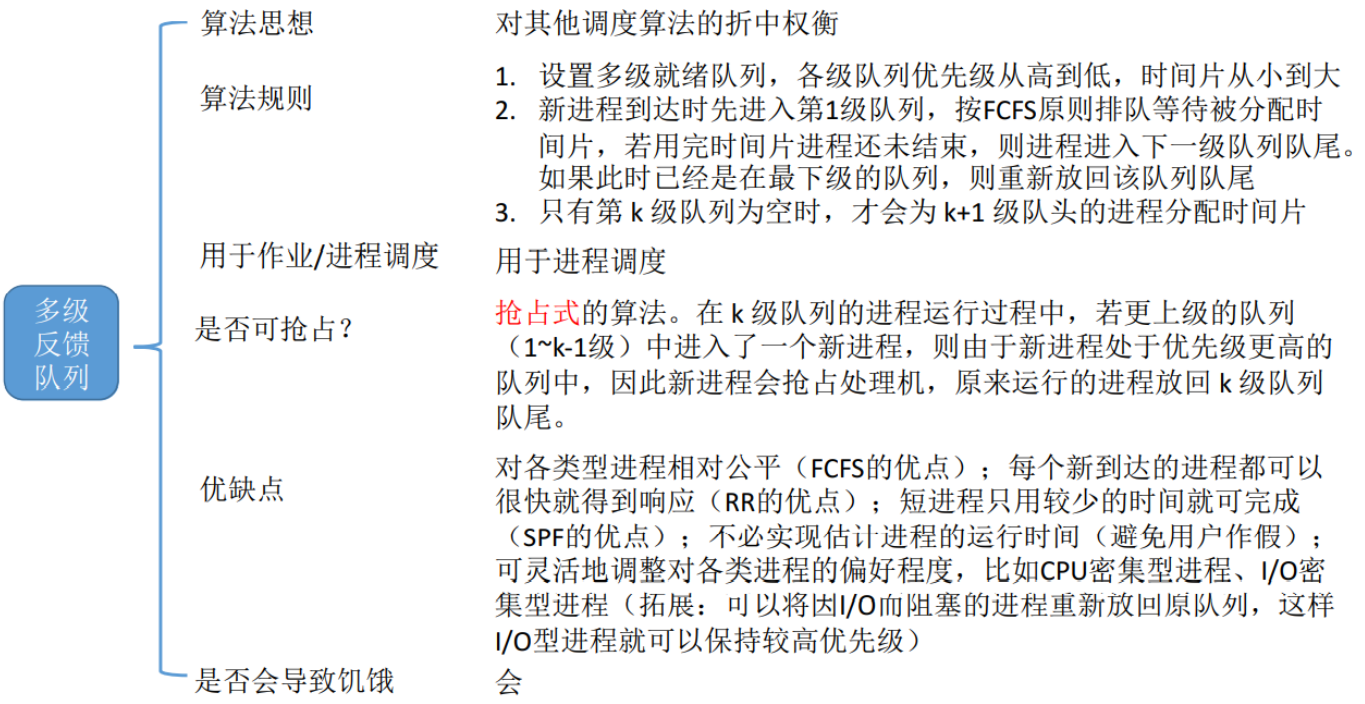
- 非抢占式的优先级调度算法：每次调度时选择当前已到达且优先级最高的进程。当前进程主动放弃处理机时发生调度。
- 抢占式的优先级调度算法：每次调度时选择当前已到达且优先级最高的进程。当前进程主动放弃处理机时发

生调度。另外，当就绪队列发生改变时也需要检查是会发生抢占。

- 就绪队列未必只有一个，可以按照不同优先级来组织。另外，也可以把优先级高的进程排在更靠近队头的位置
- 根据优先级是否可以动态改变，可将优先级分为静态优先级和动态优先级两种。
- 静态优先级：创建进程时确定，之后一直不变。
- 动态优先级：创建进程时有一个初始值，之后会根据情况动态地调整优先级。

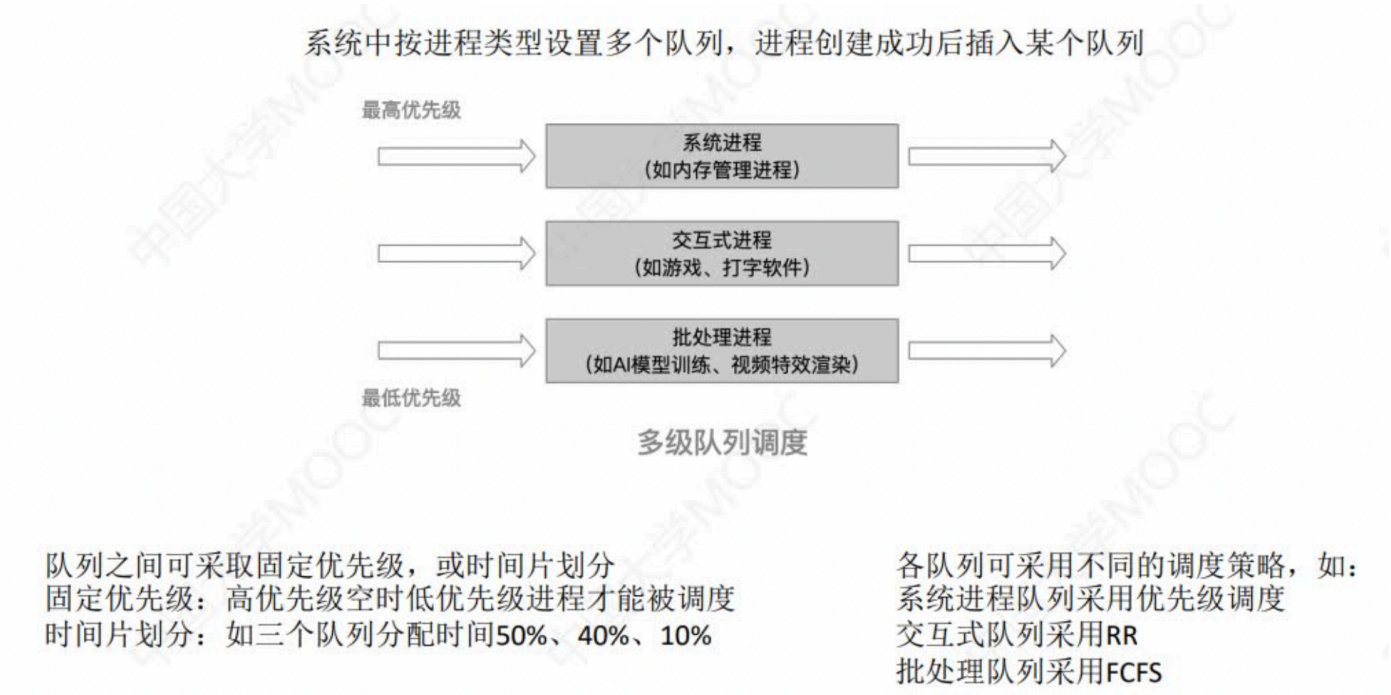
- 系统进程优先级 高于 用户进程
- 前台进程优先级 高于 后台进程
- 操作系统更偏好 I/O型进程(或称I/O繁忙型进程)
- 注：与I/O型进程相对的是计算型进程(或称CPU繁忙型进程)
- I/O设备和CPU可以并行工作。如果优先让I/O繁忙型进程优先运行的话，则越有可能让I/O设备尽早地投入工作，则资源利用率、系统吞吐量都会得到提升

多级反馈队列



- 设置多级就绪队列，各级队列优先级从高到低，时间片从小到大
- 新进程到达时先进入第1级队列，按FCFS原则排队等待被分配时间片。若用完时间片进程还未结束，则进程进入下一级队列队尾。如果此时已经在最下级的队列，则重新放回最下级队列队尾
- 只有第 k 级队列为空时，才会为 k+1 级队头的进程分配时间片
- 被抢占处理机的进程重新放回原队列队尾

多级队列调度算法



进程同步/进程互斥

进程同步

- **同步**亦称**直接制约关系**，它是指为完成某种任务而建立的两个或多个进程，这些进程因为需要在某些位置上**协调**它们的**工作次序**而产生的制约关系。进程间的直接制约关系就是源于它们之间的相互合作。
- 并发性带来了异步性，有时需要通过进程同步解决这种异步问题。

进程互斥

- 我们把**一个时间段内只允许一个进程使用**的资源称为**临界资源**。许多物理设备(比如摄像头、打印机)都属于临界资源。此外还有许多变量、数据、内存缓冲区等都属于临界资源。
- 对临界资源的访问，必须**互斥**地进行。互斥，亦称**间接制约关系**。**进程互斥**指当一个进程访问某临界资源时，另一个想要访问该临界资源的进程必须等待。当前访问临界资源的进程访问结束，释放该资源之后，另一个进程才能去访问临界资源
- 四个部分
  - 进入区
    - 检查是否可以进入临界区，若可进入，需要"上锁"
  - 临界区
    - 访问临界资源的代码
  - 退出区
    - 负责"解锁"
  - 剩余区
    - 其余代码部分
- 需要遵循的原则
  - 空闲让进

        - 临界区空闲时，应允许一个进程访问
    - 忙则等待
        - 临界区正在被访问时，其他试图访问的进程需要等待
    - 有限等待
        - 要在有限时间内进入临界区，保证不会饥饿
    - 让权等待
        - 进不了临界区的进程，要释放处理机，防止忙等

进程互斥的软件实现方法

单标志法

- 在进入区只做"检查"，不"上锁"
- 在退出区把临界区的使用权转交给另一个进程(相当于在退出区既给另一进程"解锁"，又给自己"上锁")
- 主要问题：不遵循"空闲让进"原则

双标志先检查

- 在进入区先"检查"后"上锁"，退出区"解锁"
- 主要问题：不遵循"忙则等待"原则

双标志后检查

- 在进入区先"加锁"后"检查"，退出区"解锁"
- 主要问题：不遵循"空闲让进、有限等待"原则，可能导致"饥饿"

Peterson算法

- 在进入区"主动争取，主动谦让，检查对方是否想进、己方是否谦让"
- 主要问题：不遵循"让权等待"原则，会发生"忙等"

进程互斥的硬件实现方法

中断屏蔽方法

- 使用"开/关中断"指令实现
- 优点：简单高效
- 缺点：只适用于单处理机，只适用于操作系统内核进程

TestAndSet(TS指令/TSL指令)

- 简称TS指令，也有地方称为TestAndSetLock指令，或TSL指令

- old记录是否已被上锁
- 再将lock设为true
- 检查临界区是否已被上锁
- (若已上锁，则循环重复前几步)

- 优点：实现简单，适用于多处理机环境
- 缺点：不满足"让权等待"

Swap指令(XCHG指令)

- 逻辑同TSL

锁

1. 互斥锁

解决临界区最简单的工具就是互斥锁（mutex lock）。一个进程在进入临界区时应获得锁；在退出临界区时释放锁。函数 acquire()获得锁，而函数 release()释放锁。

每个互斥锁有一个布尔变量 available，表示锁是否可用。如果锁是可用的，调用 acqiure()会成功，且锁不再可用。当一个进程试图获取不可用的锁时，会被阻塞，直到锁被释放。

```
acquire()
    while(!available)
        ;                       //忙等待
    available = false;          //获得锁
}
release(){
    available = true;           //释放锁
}
```

acquire()或 release()的执行必须是原子操作，因此互斥锁通常采用硬件机制来实现。

互斥锁的主要缺点是**忙等待**，当有一个进程在临界区中，任何其他进程在进入临界区时必须连续循环调用 acquire()。当多个进程共享同一 CPU 时，就浪费了 CPU 周期。因此，互斥锁通常用于多处理器系统，一个线程可以在一个处理器上等待，不影响其他线程的执行。

需要连续循环忙等的互斥锁，都可称为自旋锁（spin lock），如 TSL指令、swap指令、单标志法

特性：

- 需忙等，进程时间片用完才下处理机，违反“让权等待”
- 优点：等待期间不用切换进程上下文，多处理器系统中，若上锁的时间短，则等待代价很低
- 常用于多处理器系统，一个核忙等，其他核照常工作，并快速释放临界区
- 不太适用于单处理机系统，忙等的过程中不可能解锁

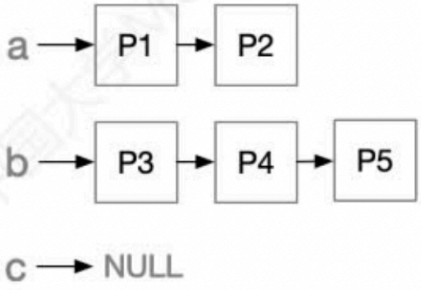
```
do {
    entry section;      //进入区
    critical section;   //临界区
    exit section;       //退出区
    remainder section;  //剩余区
} while(true)
```

```
acquire()
    while(!available)
        ;                       //忙等待
    available = false;          //获得锁
}
```

```
release(){
    available = true;           //释放锁
}
```

进程同步：条件变量

condition a,b,c;　　//定义三个条件变量



可简单理解为：等待队列

- 条件变量没有值，只有队列
- 对条件变量的操作：wait/signal
- 某进程调用 wait(a)：等待在a
- 某进程调用 signal(a)：唤醒一个等待a的进程

	信号量	条件变量
是否有“值”	有“值”，表示资源数，也有等待队列	没有值，就是个等待队列队列
操作	P、V	wait、signal
	V操作不会搞了个寂寞，会累计“值”	signal 可能会搞了个寂寞，如果一个条件变量上没有等待进程，则signal就完全无效

信号量机制

整型信号量

- 用一个整数型变量作为信号量，数值表示某种资源数
- 整型信号量与普通整型变量的区别：对信号量只能执行初始化、P、V三种操作
- 整型信号量存在的问题：不满足让权等待原则

记录型信号量

- S.value表示某种资源数，S.L指向等待该资源的队列
- P操作中，一定是先S.value--，之后可能需要执行block原语
- V操作中，一定是先S.value++，之后可能需要执行wakeup原语
- 注意：要能够自己推断在什么条件下需要执行block或wakeup
- 可以用记录型信号量实现系统资源的"申请"和"释放"
- 可以用记录型信号量实现进程互斥、进程同步

用信号量机制实现

进程互斥

- 分析问题，确定临界区
- 设置互斥信号量，初值为1
- 临界区之前对信号量执行P操作
- 临界区之后对信号量执行V操作

进程同步

- 分析问题，找出哪里需要实现"一前一后"的同步关系
- 设置同步信号量，初始值为0
- 在"前操作"之后执行V操作
- 在"后操作"之前执行P操作

进程的前驱关系

- 分析问题，画出前驱图，把每一对前驱关系都看成一个同步问题
- 为每一对前驱关系设置同步信号量，初值为0
- 在每个"前操作"之后执行V操作
- 在每个"后操作"之前执行P操作

生产者消费者问题

多生产者多消费者

吸烟者问题

读者写者问题

哲学家进餐问题

管程

引入目的

- 解决信号量机制编程繁琐的问题

组成

- 共享数据结构
- 对数据结构初始化的语句
- 一组用来访问数据结构的过程(函数)

基本特征

- 各外部进程/线程只能通过管程提供的特定"入口"才能访问共享数据
- 每次仅允许一 个进程在管程内执行某个内部过程

补充

- 各进程必须互斥访问管程的特性是由编译器实现的
- 可在管程中设置条件变量及等待/唤醒操作以解决同步问题

死锁

定义

- 各进程互相等待对方手里的资源，导致各进程都阻塞，无法向前推进

死锁/饥饿/死循环的区别

- 死锁：至少是两个进程一起死锁，死锁进程处于阻塞态
- 饥饿：可以只有一个进程饥饿，饥饿进程可能阻塞也可能就绪
- 死循环：可能只有一个进程发生死循环，死循环的进程可上处理机
- 死锁和饥饿是操作系统要解决的问题，死循环是应用程序员要解决的

死锁产生的必要条件

- 互斥条件
    - 对必须互斥使用的资源的争抢才会导致死锁
- 不剥夺条件
    - 进程保持的资源只能主动释放，不可强行剥夺
- 请求和保持条件
    - 保持着某些资源不放的同时，请求别的资源
- 循环等待条件
    - 存在一种进程资源的循环等待链
    - 循环等待未必死锁，死锁一定有循环等待

什么时候发生死锁

- 对不可剥夺资源的不合理分配，可能导致死锁

死锁的处理策略

- 预防死锁
    - 破坏死锁产生的四个必要条件
- 避免死锁
    - 避免系统进入不安全状态(银行家算法)
- 死锁的检测和解除
    - 允许死锁发生，系统负责检测出死锁并解除

预防死锁

破坏互斥条件

- 将临界资源改造为可共享使用的资源(如SPOOLing技术)
- 缺点：可行性不高，很多时候无法破坏互斥条件

破坏不剥夺条件

- 方案一：申请的资源得不到满足时，立即释放拥有的所有资源
- 方案二：申请的资源被其他进程占用时，由操作系统协助剥夺(考虑优先级)
- 缺点：实现复杂；剥夺资源可能导致部分工作失效；反复申请和释放导致系统开销大；可能导致饥饿

破坏请求和保持条件

- 运行前分配好所有需要的资源，之后一直保持
- 缺点：资源利用率低;可能导致饥饿

破坏循环等待条件

- 给资源编号，必须按编号从小到大的顺序申请资源
- 缺点：不方便增加新设备；会导致资源浪费；用户编程麻烦

避免死锁

安全序列

- 所谓**安全序列**，就是指如果系统按照这种序列分配资源，则每个进程都能顺利完成。只要能找出一个安全序列，系统就是**安全状态**。当然，**安全序列可能有多个**。

不安全序列

无法找到任何一个安全序列，说明此时系统处于不安全状态，有可能发生死锁

如果分配了资源之后，系统中找不出任何一个安全序列，系统就进入了**不安全状态**。这就意味着之后可能所有进程都无法顺利的执行下去。当然，如果有进程前归还了一些资源，那**系统也有可能重新回到安全状态**，不过我们在分配资源之前总是要考虑到最坏的情况。

如果系统处于**安全状态**，就**一定不会发生死锁**。如果系统进入不安全状态，就可能发生死锁(处于不安全状态未必就是发生了死锁，但发生死锁时一定是在不安全状态)

因此可以在资源分配之前预先判断这次分配是否会导致系统进入不安全状态，以此决定是否答应资源分配请求。这也是"银行家算法"的核心思想。

银行家算法

- 检查此次申请是否超过了之前声明的最大需求数
- 检查此时系统剩余的可用资源是否还能满足这次请求
- 试探着分配，更改各数据结构
- 用安全性算法检查此次分配是否会导致系统进入不安全状态

死锁的避免和解除

检测

数据结构：资源分配图

- 两种结点
  - 进程结点
  - 资源结点
- 两种边
  - 进程结点 -> 资源结点(请求边)
  - 资源结点 -> 进程结点(分配边)

死锁检测算法

依次消除与不阻塞进程相连的边，直到无边可消
注：所谓不阻塞进程是指其申请的资源数还足够的进程
死锁定理：若资源分配图是不可完全简化的，说明发生了死锁

解除

- 资源剥夺法
- 撤销进程法(终止进程法)
- 进程回退法

3 内存管理

内存的基础知识

定义

- 存储单元、内存地址
- 按字节编制、按字编址

进程运行的基本原理

指令的工作原理

- 操作码 + 若干参数(可能包含地址参数)

逻辑地址(相对地址)和物理地址(绝对地址)

写程序到程序运行

- 编辑源代码文件
- 编译
  - 由源代码文件生成目标模块(高级语言"翻译"为机器语言)
- 链接
  - 由目标模块生成装入模块，链接后形成完整的逻辑地址
- 装入

    - 将装入模块装入内存，装入后形成物理地址

链接方式

- 静态链接
    - 装入前链接成一个完整装入模块
- 装入时动态链接
    - 运行前边装入边链接
- 运行时动态链接
    - 运行时需要目标模块才装入并链接

装入方式

- 绝对装入
    - 编译时产生绝对地址
- 可重定位装入
    - 装入时将逻辑地址转换为物理地址
- 动态运行时装入
    - 运行时将逻辑地址转换为物理地址，需设置重定位寄存器

内存管理

内存空间的分配与回收

- 连续分配管理方式
    - 单一连续分配
    - 固定分区分配
    - 动态分区分配
- 非连续分配管理方式
    - 基本分页存储管理
    - 基本分段存储管理
    - 段页式存储管理

内存空间的扩充(实现虚拟性)

- 覆盖技术
- 交换技术
- 虚拟存储技术

地址转换

- 操作系统负责实现逻辑地址到物理地址的转换
- 三种方式
    - 绝对装入：编译器负责地址转换(单道程序阶段，无操作系统)
    - 可重定位装入：装入程序负责地址转换(早期多道批处理阶段)
    - 动态运行时装入：运行时才进行地址转换(现代操作系统)

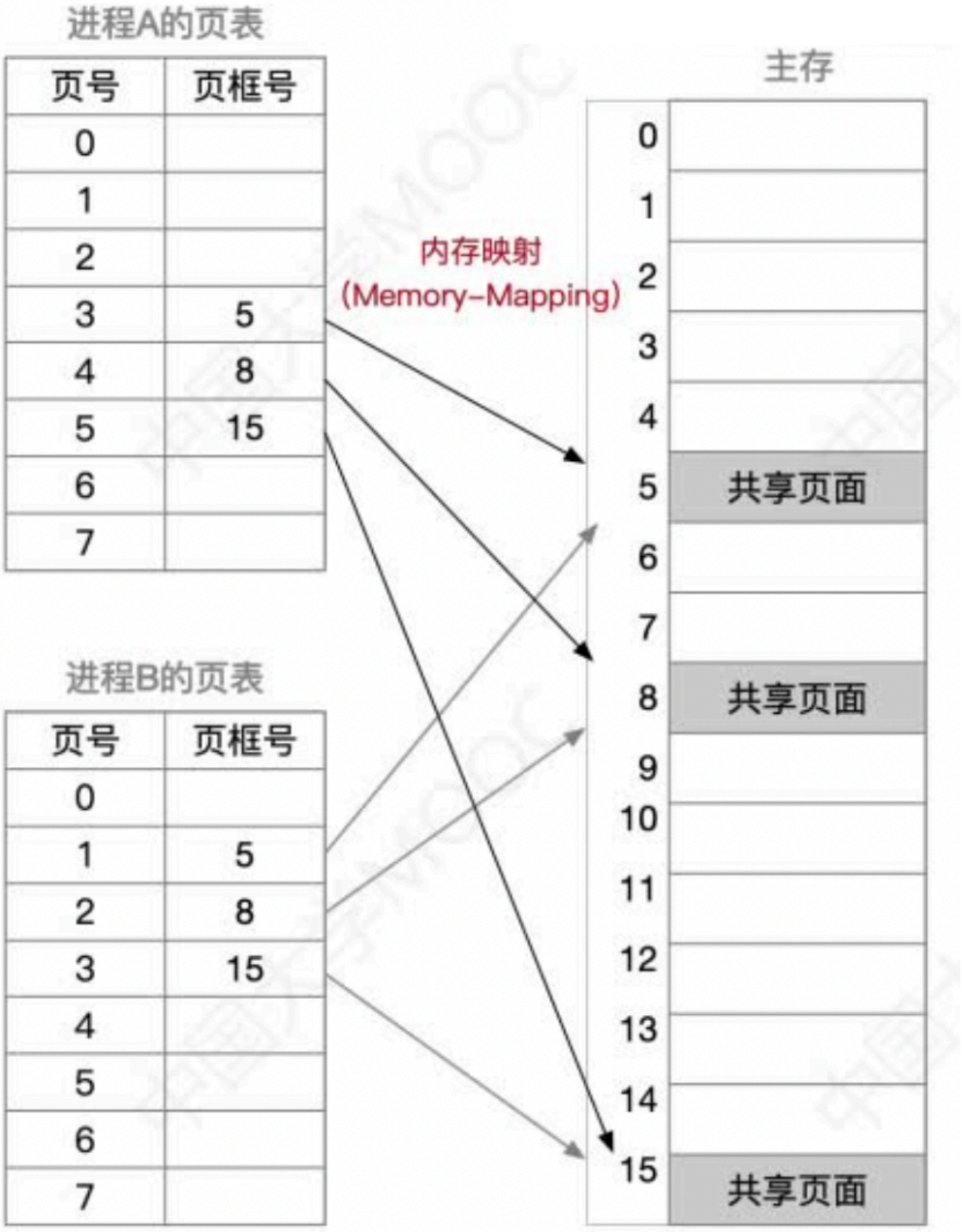
存储保护

- 保证各进程在自己的内存空间内运行，不会越界访问
- 两种方式
    - 设置上下限寄存器
    - 利用重定位寄存器、界地址寄存器进行判断

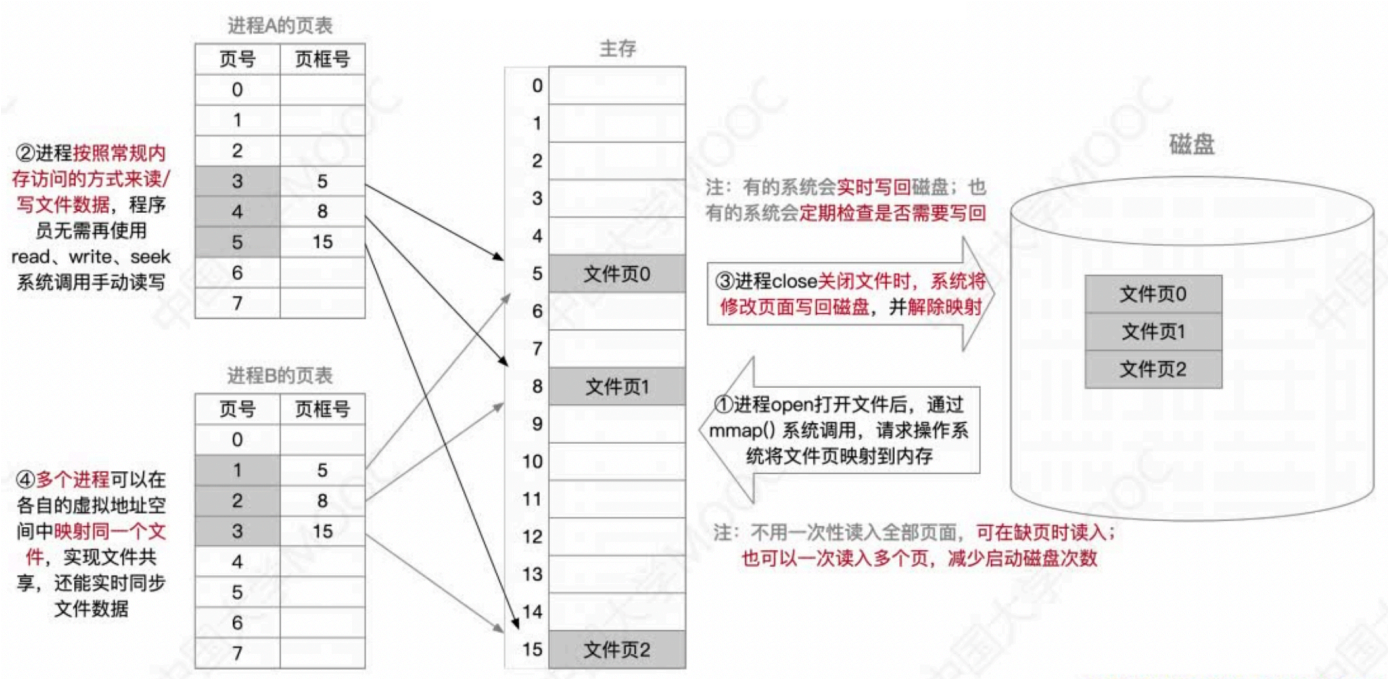
内存共享

内存共享通常是通过“内存映射”实现的，将多个进程的虚拟地址空间映射到同一片物理内存。

可以是“页”映射，也可以是“段”映射。


进程A的页表
页号
页框号
0
1
2
3
5
4
8
5
15
6
7
进程B的页表
页号
页框号
0
1
5
2
8
3
15
4
5
6
7
内存映射
(Memory-Mapping)
主存
0
1
2
3
4
5
共享页面
6
7
8
共享页面
9
10
11
12
13
14
15
共享页面

内存映射文件



覆盖与交换

覆盖技术

- 一个固定区
  - 存放最活跃的程序段
  - 固定区中的程序段在运行过程中不会调入调出
- 若干覆盖区
  - 不可能同时被访问程序段可共享一个覆盖区
  - 覆盖区中的程序段在运行过程中会根据需要调入调出
- 必须由程序员声明覆盖结构，操作系统完成自动覆盖
- 缺点：对用户不透明，增加了用户编程负担

交换技术

- 内存紧张时，换出某些进程以腾出内存空间，再换入某些进程
- 磁盘分为文件区和对换区，换出的进程放在对换区

覆盖与交换的区别

- 覆盖是在同一个程序或进程中的
- 交换是在不同进程(或作业)之间的

连续分配管理

- 内部碎片：分配给某进程的内存区域中，如果有些部分没有用上。
- 外部碎片：是指内存中的某些空闲分区由于太小而难以利用。
- 空闲分区表、空闲分区链

单一连续分配

- 只支持单道程序，内存分为系统区和用户区，用户程序放在用户区
- 无外部碎片，有内部碎片

固定分区分配

- 支持多道程序，内存用户空间分为若干个固定大小的分区，每个分区只能装一道作业
- 无外部碎片，有内部碎片
- 两种分区方式
    - 分区大小相等
    - 分区大小不等

动态分区分配

- 支持多道程序，在进程装入内存时，根据进程的大小动态地建立分区
- 无内部碎片，有外部碎片
- 外部碎片可用"紧凑"技术来解决
- 回收内存分区时，可能遇到四种情况
    - 回收区之后有相邻的空闲分区
    - 回收区之前有相邻的空闲分区
    - 回收区前、后都有相邻的空闲分区
    - 回收区前、后都没有相邻的空闲分区

动态分区分配算法

- 首次适应算法(First Fit)
- 最佳适应算法(Best Fit)
- 最坏适应算法(Worst Fit)
- 邻近适应算法(Next Fit)

算法	算法思想	分区排列顺序	优点	缺点
首次适应	从头到尾找适合的分区	空闲分区以地址递增次序排列	综合看性能最好。***算法开销小***，回收分区后一般不需要对空闲分区队列重新排序	
最佳适应	优先使用更小的分区，以保留更多大分区	空闲分区以容量递增次序排列	会有更多的大分区被保留下来，更能满足大进程需求	会产生很多太小的、难以利用的碎片；***算法开销大***，回收分区后可能需要对空闲分区队列重新排序
最坏适应	优先使用更大的分区，以防止产生太小的不可用的碎片	空闲分区以容量递减次序排列	可以减少难以利用的小碎片	大分区容易被用完，不利于大进程；***算法开销大***（原因同上）
邻近适应	由首次适应演变而来，每次从上次查找结束位置开始查找	空闲分区以地址递增次序排列（可排列成循环链表）	不用每次都从低地址的小分区开始检索。***算法开销小***（原因同首次适应算法）	会使高地址的大分区也被用完

基本分页存储管理

- 基本分页存储管理的思想：把进程分页，各个页面可离散地放到各个的内存块中
- 易混概念
  - "页框、页帧、内存块、物理块、物理页" 与 "页、页面"
  - "页框号、页帧号、内存块号、物理块号、物理页号" 与 "页号、页面号"

页表

- 页表记录了页面和实际存放的内存块之间的映射关系
- 一个进程对应一张页表，进程的每一页对应一个页表项，每个页表项由"页号"和"块号"组成
- 每个页表项的大小是相同的，页号是"隐含"的
- i号页表项存放地址 = 页表始址 + i * 页表项大小

逻辑地址结构

- 页号P = 逻辑地址 / 页面大小
- 页内偏移量W = 逻辑地址 % 页面大小

如何实现地址转换

- 计算出逻辑地址对应的页号与页内偏移量
- 找到对应页面在内存中的存放位置(查页表)
- 物理地址 = 页面始址 + 页内偏移量

基本地址变换机构

- 用于实现逻辑地址到物理地址转换的一组硬件机构

页表寄存器的作用

存放页表起始地址
存放页表长度

地址变换过程

- 根据逻辑地址算出页号、页内偏移量
- 页号的合法性检查(与页表长度对比)
- 若页号合法，再根据页表起始地址、页号找到对应页表项
- 根据页表项中记录的内存块号、页内偏移量得到最终的物理地址
- 访问物理内存对应的内存单元

其他

- 页内偏移量位数与页面大小之间的关系(要能用其中一个条件推出另一个条件)
- 页式管理中地址是一维的
- 实际应用中，通常使一个页框恰好能放入整数个页表项
- 为了方便找到页表项，页表一般是放在连续的内存块中的

具有快表的地址变换机构

定义

快表，又称**联想寄存器(TLB**， translation lookaside buffer)，是一种访问速度比内存快很多的高速缓存(TLB不是内存)，用来存放**最近访问的页表项的副本**，可以加速地址变换的速度。与此对应，内存中的页表常称为**慢表**。

地址变换

	地址变换过程	访问一个逻辑地址的访存次数
基本地址变换机构	①算页号、页内偏移量 ②检查页号合法性 ③查页表，找到页面存放的内存块号 ④根据内存块号与页内偏移量得到物理地址 ⑤访问目标内存单元	两次访存
具有快表的地址变换机构	①算页号、页内偏移量 ②检查页号合法性 ③查快表。若命中，即可知道页面存放的内存块号，可直接进行⑤；若未命中则进行④ ④查页表，找到页面存放的内存块号，并且将页表项复制到快表中 ⑤根据内存块号与页内偏移量得到物理地址 ⑥访问目标内存单元	快表命中，只需一次访存 快表未命中，需要两次访存

TLB和普通Cache的区别：TLB中只有页表项的副本，而普通Cache中可能会有其他各种数据的副本。

两级页表

单级页表存在的问题

- 所有页表项必须连续存放，页表过大时需要很大的连续空间
- 在一段时间内并非所有页面都用得到，因此没必要让整个页表常驻内存

两级页表

- 将长长的页表再分页
- 逻辑地址结构：(一级页号，二级页号，页内偏移量)
- 注意几个术语：页目录表/外层页表/顶级页表

如何实现地址变换

- 按照地址结构将逻辑地址拆分成三部分
- 从PCB中读出页目录表始址，根据一级页号查页目录表，找到下一级页表在内存中的存放位置
- 根据级页号查表，找到最终想访问的内存块号
- 结合页内偏移量得到物理地址

其他

- 多级页表中，各级页表的大小不能超过一个页面。若两级页表不够，可以分更多级
- 多级页表的访存次数(假设没有快表机构)
    - N级页表访问一个逻辑地址需要N + 1次访存

基本分段存储管理

分段

- 将地址空间按照程序自身的逻辑关系划分为若干个段，每段从0开始编址
- 每个段在内存中占据连续空间，但各段之间可以不相邻
- 逻辑地址结构：(段号，段内地址)

段表

- 记录逻辑段到实际存储地址的映射关系
- 每个段对应一个段表项。各段表项长度相同，由段号(隐含)、段长、基址组成

地址变换

- 由逻辑地址得到段号、段内地址
- 段号与段表寄存器中的段长度比较，检查是否越界
- 由段表始址、段号找到对应段表项
- 根据段表中记录的段长，检查段内地址是否越界
- 由段表中的"基址 + 段内地址"得到最终的物理地址
- 访问目标单元

分页与分段对比

页是信息的物理单位。分页的主要目的是为了实现离散分配，提高内存利用率。分页仅仅是系统管理上的需要，完全是系统行为，对用户是不可见的。
段是信息的逻辑单位。分页的主要目的是更好地满足用户需求。一个段通常包含着一组属于一个逻辑模块的信息。分段对用户是可见的，用户编程时需要显式地给出段名。

页的大小固定且由系统决定。段的长度却不固定，决定于用户编写的程序。

分页的用户进程地址空间是一维的，程序员只需给出一个记忆符即可表示一个地址。
分段的用户进程地址空间是二维的，程序员在标识一个地址时，既要给出段名，也要给出段内地址。

分段比分页更容易实现信息的共享和保护。不能被修改的代码称为纯代码或可重入代码（不属于临界资源），这样的代码是可以共享的。可修改的代码是不能共享的

访问一个逻辑地址需要几次访存？
分页（单级页表）：第一次访存——查内存中的页表，第二次访存——访问目标内存单元。总共两次访存
分段：第一次访存——查内存中的段表，第二次访存——访问目标内存单元。总共两次访存
与分页系统类似，分段系统中也可以引入***快表***机构，将近期访问过的段表项放到快表中，这样可以少一次访问，加快地址变换速度。

段页式管理方式

分段 + 分页

- 将地址空间按照程序自身的逻辑关系划分为若干个段，在将各段分为大小相等的页面
- 将内存空间分为与页面大小相等的一个个内存块，系统以块为单位为进程分配内存
- 逻辑地址结构：(段号，页号，页内偏移量)

段表、页表

- 每个段对应一个段表项。各段表项长度相同，由段号(隐含)、页表长度、页表存放地址组成
- 每个页对应一个页表项。各页表项长度相同，由页号(隐含)、页面存放的内存块号组成

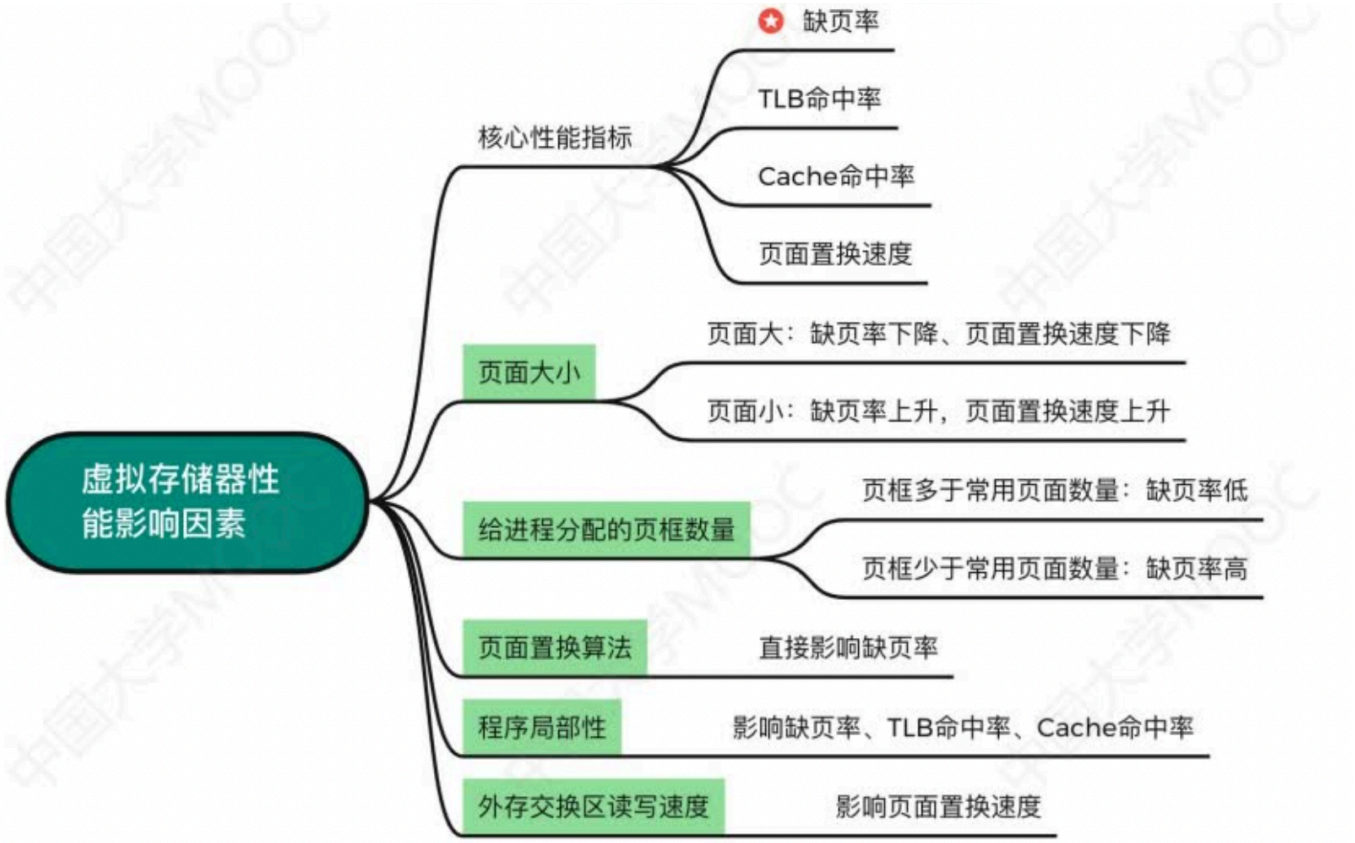
地址变换

- 由逻辑地址得到段号、页号、页内偏移量
- 段号与段表寄存器中的段长度比较，检查是否越界
- 由段表始址、段号找到对应段表项
- 根据段表中记录的页表长度，检查页号是否越界
- 由段表中的页表地址、页号得到查询页表，找到相应页表项
- 由页面存放的内存块号、页内偏移量得到最终的物理地址

访问一个逻辑地址所需的访存次数

- 第一次：查段表、第二次：查页表、第三次：访问目标单元
- 可引入快表机构，以段号和页号为关键字查询快表，即可直接找到最终的目标页面存放位置。引入快表后仅需一次访存

虚拟内存



传统存储管理方式的特征缺点

- 一次性：作业数据必须一次全部调入内存
- 驻留性：作业数据在整个运行期间都会常驻内存

局部性原理

- 时间局部性：现在访问的指令、数据在不久后很可能会被再次访问
- 空间局部性：现在访问的内存单元周围的内存空间，很可能在不久后会被访问
- 高速缓存技术：使用频繁的数据放到更高速的存储器中

虚拟内存的定义和特征

- 程序不需全部装入即可运行，运行时根据需要动态调入数据，若内存不够，还需换出一些数据
- 特征
    - 多次性：无需在作业运行时一次性全部装入内存，而是允许被分成多次调入内存。
    - 对换性：无需在作业运行时一直常驻内存，而是允许在作业运行过程中，将作业换入、换出。
    - 虚拟性：从逻辑上扩充了内存的容量，使用户看到的内存容量，远大于实际的容量。

如何实现虚拟内存技术

- 访问的信息不在内存时，由操作系统负责将所需信息从外存调入内存(请求调页功能)
- 内存空间不够时，将内存中暂时用不到的信息换出到外存(页面置换功能)
- 虚拟内存的实现

- 请求分页存储管理
- 请求分段存储管理
- 请求段页式存储管理

请求分页管理方式

页表机制

- 在基本分页的基础上增加了几个表项
- 状态位：表示页面是否已在内存中
- 访问字段：记录最近被访问过几次，或记录上次访问的时间，供置换算法选择换出页面时参考
- 修改位：表示页面调入内存后是否被修改过，只有修改过的页面才需在置换时写回外存
- 外存地址：页面在外存中存放的位置

缺页中断机构

- 找到页表项后检查页面是否已在内存，若没在内存，产生缺页中断
- 缺页中断处理中，需要将目标页面调入内存，有必要时还要换出页面
- 缺页中断属于内中断，属于内中断中的"故障"，即可能被系统修复的异常
- 一条指令在执行过程中可能产生多次缺页中断

地址变换机构

- 找到页表项是需要检查页面是否在内存中
- 若页面不再内存中，需要请求调页
- 若内存空间不够，还需换出页面
- 页面调入内存后，需要修改相应页表项

页面置换算法

- 最佳置换算法(OPT)
- 先进先出置换算法(FIFO)
- 最近最久未使用置换算法(LRU)
- 时钟置换算法(CLOCK)
- 改进型的时钟置换算法

	算法规则	优缺点
OPT	优先淘汰最长时间内不会被访问的页面	缺页率最小，性能最好；但无法实现
FIFO	优先淘汰最先进入内存的页面	实现简单；但性能很差，可能出现Belady异常
LRU	优先淘汰最近最久没访问的页面	性能很好；但需要硬件支持，算法开销大
CLOCK（NRU）	循环扫描各页面 第一轮淘汰访问位=0的，并将扫描过的页面访问位改为1。若第一轮没选中，则进行第二轮扫描。	实现简单，算法开销小；但未考虑页面是否被修改过。
改进型CLOCK（改进型NRU）	若用（访问位, 修改位）的形式表述，则 第一轮：淘汰（0, 0） 第二轮：淘汰（0, 1），并将扫描过的页面访问位都置为0 第三轮：淘汰（0, 0） 第四轮：淘汰（0, 1）	算法开销较小，性能也不错

页面分配策略

驻留集

- 指请求分页存储管理中给进程分配的内存块的集合

页面分配、置换策略

- 固定分配与可变分配：区别在于进程运行期间驻留集大小是否可变
- 局部置换与全局置换：区别在于发生缺页时是否只能从进程自己的页面中选择一个换出
- 固定分配局部置换：进程运行前就分配一定数量物理块，缺页时只能换出进程自己的某一页
- 可变分配全局置换：只要缺页就分配新物理块，可能来自空闲物理块，也可能需换出别的进程页面
- 可变分配局部置换：频繁缺页的进程，多分配一些物理块；缺页率很低的进程，回收一些物理块。直到缺页率合适

何时调入页面

- 预调页策略：一般用于进程运行前
- 请求调页策略：进程运行时，发现缺页再调页

从何处调页

- 对换区：采用连续存储方式，速度更快；文件区：采用离散存储方式，速度更慢
- 对换区足够大：运行将数据从文件区复制到对换区，之后所有的页面调入、调出都是在内存与对换区之间进行
- 对换区不够大：不会修改的数据每次都从文件区调入；会修改的数据调出到对换区，需要时再从对换区调入
- UNIX方式：第一次使用的页面都从文件区调入；调出的页面都写回对换区，再次使用时从对换区调入

抖动(颠簸)现象

- 页面频繁换入换出的现象。主要原因是分配给进程的物理块不够

工作集

- 在某段时间间隔里，进程实际访问页面的集合。驻留集大小一般不能小于工作集大小

4 文件管理

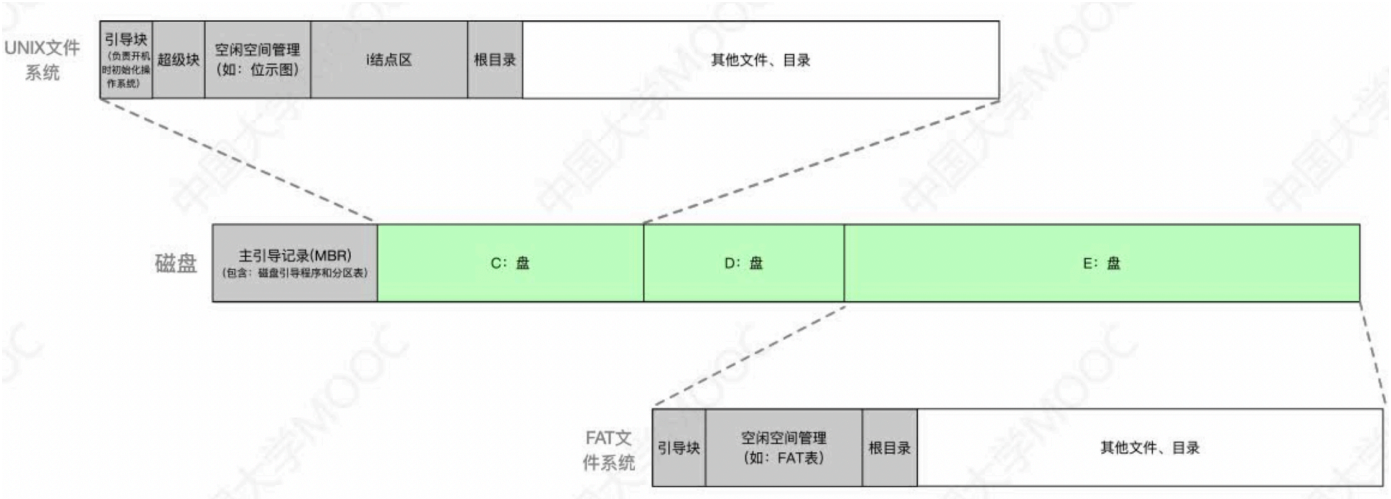
初识

- 文件的定义：一组有意义的信息的集合
- 文件的属性：文件名、标识符、类型、位置、大小、保护信息...
- 文件内部应该如何被组织起来(文件的逻辑结构)
- 文件之间应该如何被组织起来(目录结构)
- 操作系统应向上提供哪些功能
    - 创建文件(create系统调用)
    - 删除文件(delete系统调用)
    - 读文件(read系统调用)
    - 写文件(write系统调用)
    - 打开文件(open系统调用)
    - 关闭文件(close系统调用)
- 文件应如何存放在外存中(文件的物理结构)
- 操作系统如何管理外存中的空闲块(存储空间的管理)
- 操作系统需要提供的其他文件管理功能
    - 文件共享
    - 文件保护

文件系统的全局结构

物理格式化（低级格式化）：划分扇区，检测坏扇区，并用备用扇区替换坏扇区。

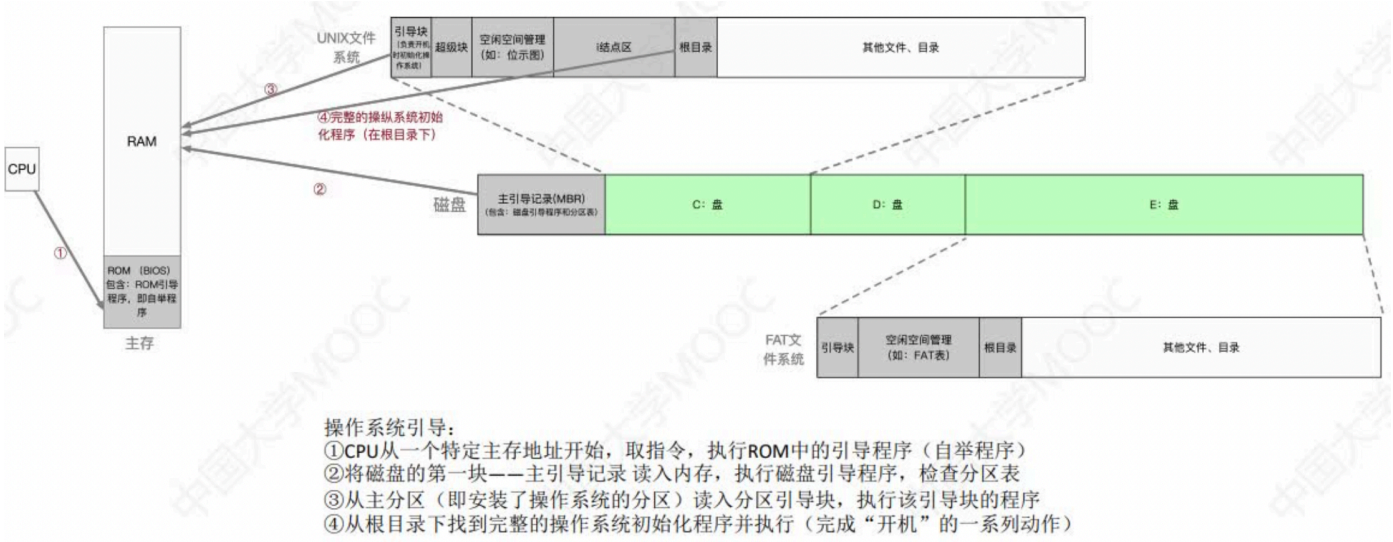
逻辑格式化：磁盘分区（分卷Volume），完成个分区的文件系统初始化。



每个分区可以是不同的文件系统，且每个分区都会有引导块，但未必都安装了操作系统。如果没安装操作系统，则该分区的引导块为空。

- 超级块：找到空闲块。
- i结点区：索引结点。

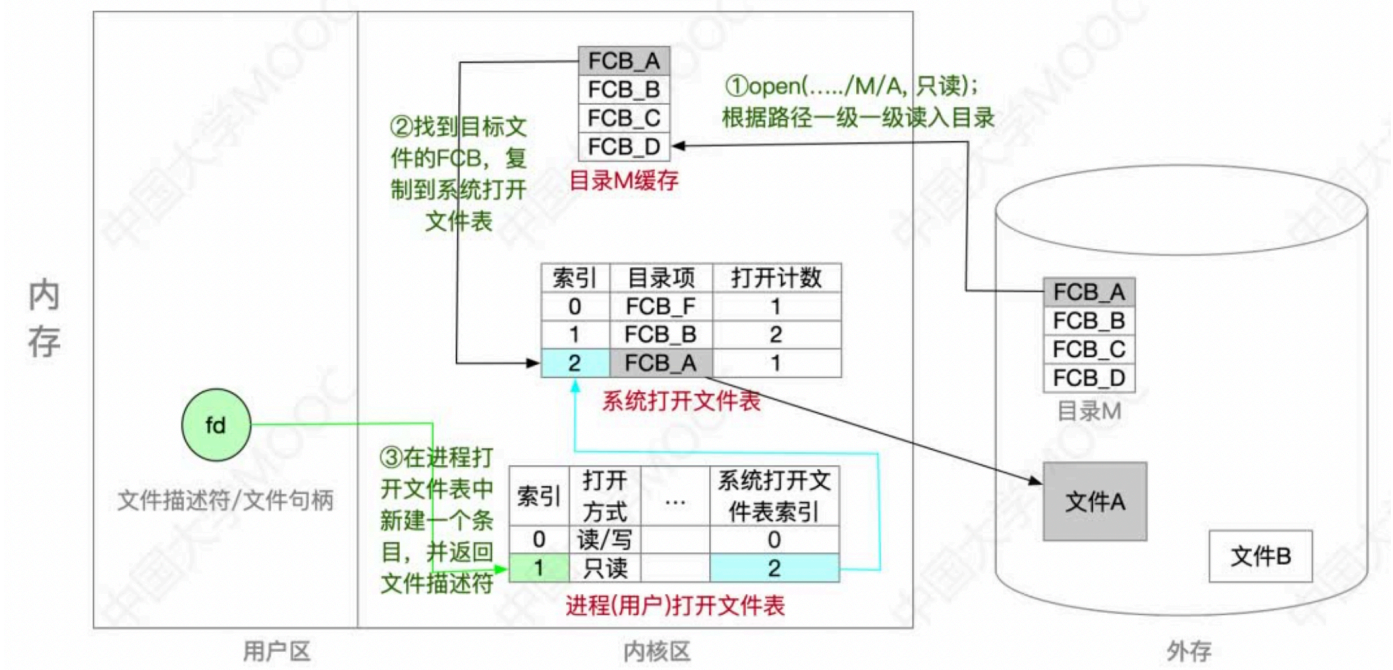
操作系统引导



文件系统在外存中的结构

每个分区就是一个独立的文件系统，可以采用不同的方案实现空闲分区管理、文件物理结构、目录实现方式。

文件系统在内存中的结构



近期访问过的目录文件会缓存在内存中，不用每次都从磁盘读入，这样可以加快目录检索速度。

文件元数据

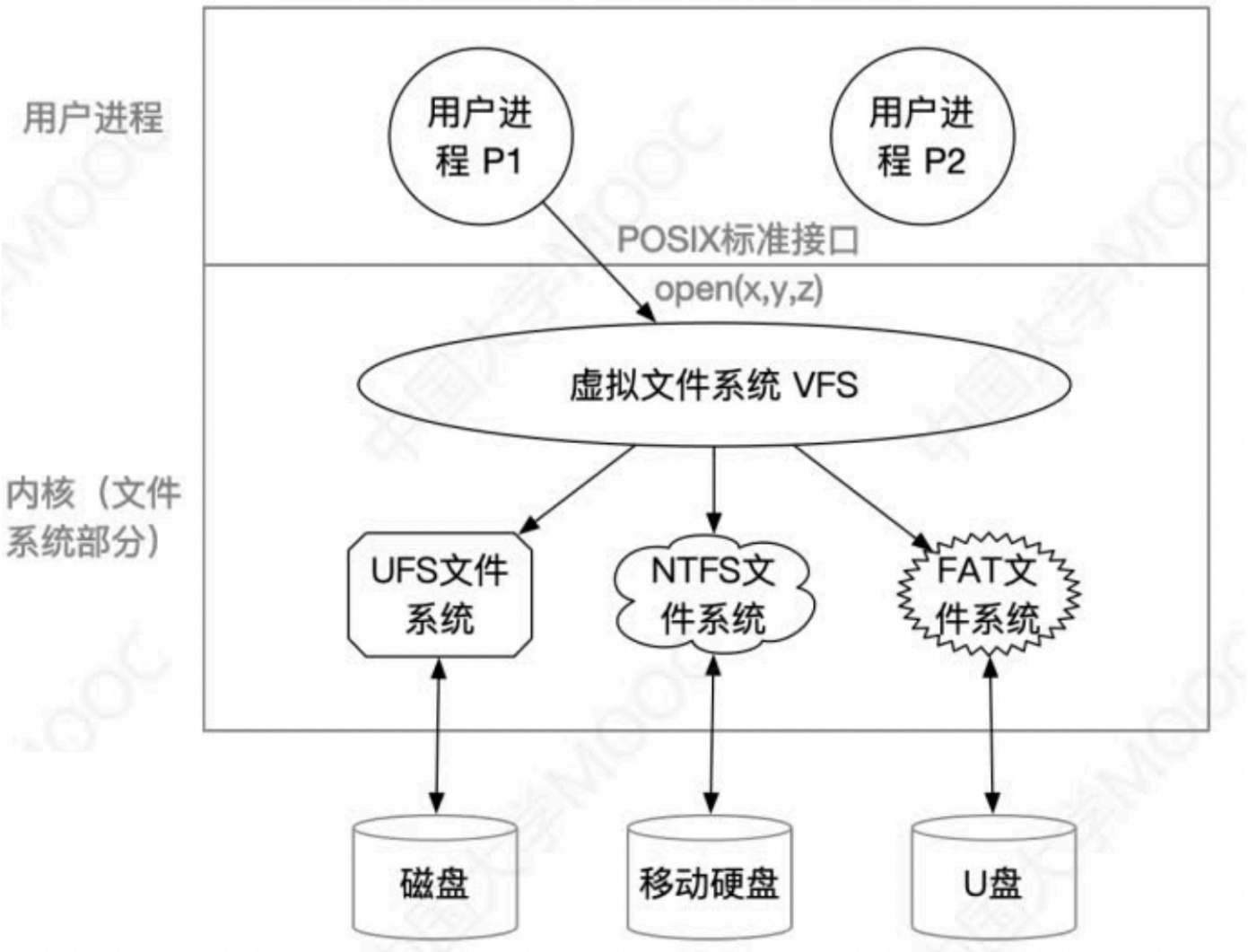
文件元数据（metadata）：即文件属性，记录在文件FCB中。

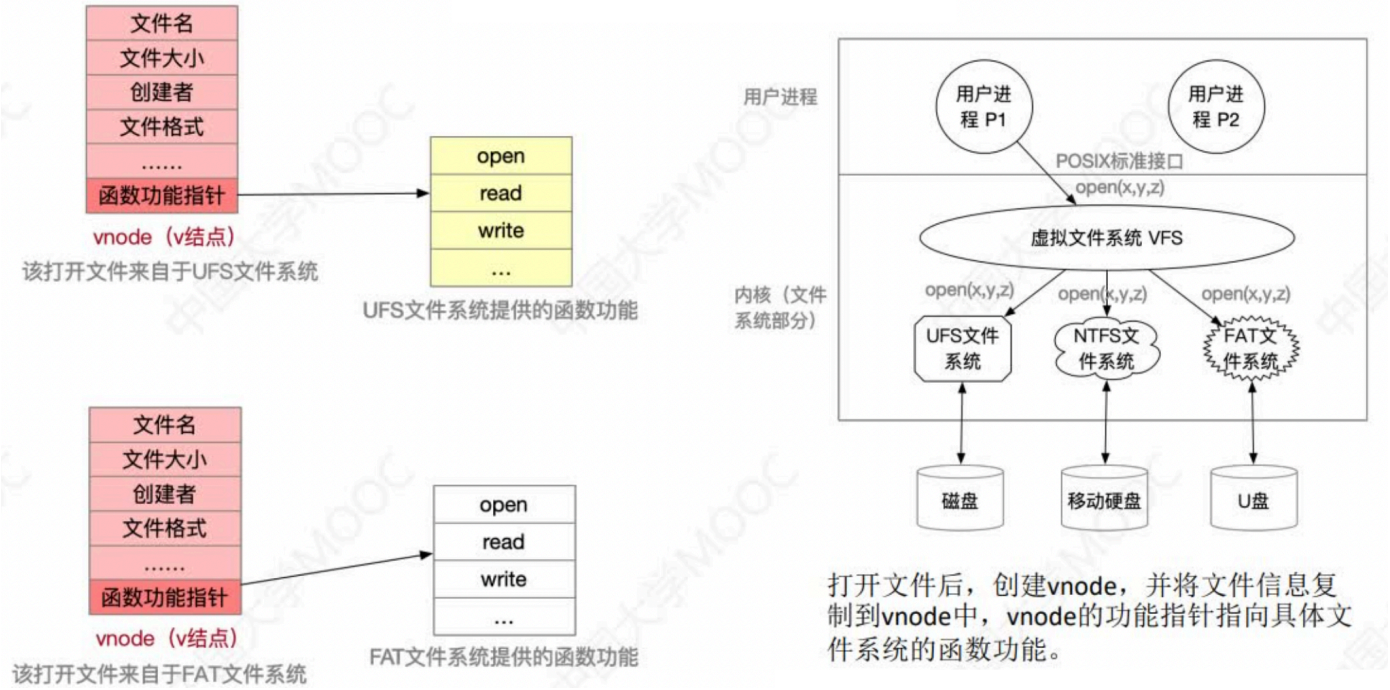
如： 文件创建者，创建时间，上一次修改时间，文件格式，隐藏标志，表示是否隐藏不显示？表示是否是系统文件？表示是否只读？文件大小等。

虚拟文件系统

虚拟文件系统的特点：

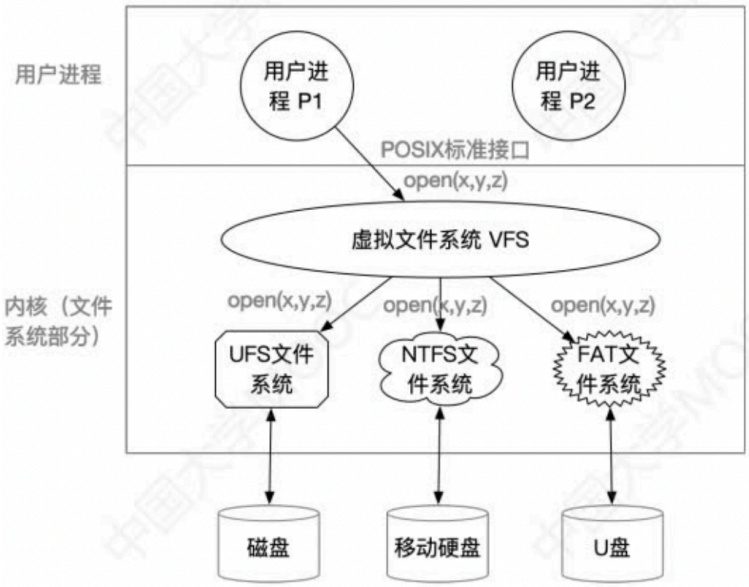
1. 向上层用户进程提供统一标准的系统调用接口，屏蔽底层具体文件系统的实现差异。
2. VFS要求下层的文件系统必须实现某些规定的函数功能，如：open/read/write。一个新的文件系统想要在某操作系统上被使用，就必须满足该操作系统VFS的要求。
3. 每打开一个文件，VFS就在主存中新建一个vnode，用统一的数据结构表示文件，无论该文件存储在哪个文件系统。





文件系统的挂载

文件系统挂载（mounting），即文件系统安装/装载——如何将一个文件系统挂载到操作系统中？



文件系统挂载要做的事：

①在VFS中注册新挂载的文件系统。内存中的挂载表（mount table）包含每个文件系统的相关信息，包括文件系统类型、容量大小等。

②新挂载的文件系统，要向VFS提供一个函数地址列表

③将新文件系统加到挂载点（mount point），也就是将新文件系统挂载在某个父目录下

目录

目录的操作

普通文件的操作：创建create、删除delete、打开open、关闭close、读read、写write

目录文件的操作：

- create系统调用，创建目录：创建一个新目录文件，为之分配磁盘空间
- delete系统调用，删除目录：只有空目录可以删除
- opendir系统调用，打开目录：和打开普通文件一样，只是将指向该目录文件的FCB读入内存而已，并不是把目录文件的数据全部读入内存
- readdir系统调用，读目录：必须在打开目录后才能读目录，返回目录文件中的下一个目录项
- rename系统调用，重命名：给目录重命名，本质上是修改指向该目录文件的FCB
- link系统调用，建立硬链接：将指定的文件与目录建立硬链接，该系统调用导致指定目录中多了一个FCB，而指定文件的inode共享计数器+1
- unlink系统调用，解除硬链接，指定文件的 inode共享计数器-1，若计数器=0，则删除文件数据

新建一个目录，会默认有两个FCB：'.'和'..'

文件的逻辑结构

无结构文件

- 由二进制流或字符流组成，无明显的逻辑结构

有结构文件

- 由记录组成，分为定长记录、可变长记录
- 逻辑结构
  - 顺序文件
  - 索引文件
  - 索引顺序文件

顺序文件

链式存储

- 无论是定长/可变长记录，都无法实现随机存取，每次只能从第一个记录开始依次往后查找

顺序存储

- 可变长记录
  - 无法实现随机存取。每次只能从第一个记录开始依次往后查找
- 定长记录
  - 可实现随机存取。记录长度为L，则第i个记录存放的相对位置是i*L
  - 若采用串结构，无法快速找到某关键字对应的记录
  - 若采用顺序结构，可以快速找到某关键字对应的记录(如折半查找)

有结构文件

顺序文件

- 串结构：记录顺序与关键字无关
- 顺序结构：记录按关键字顺序排列
- 可变长记录的顺序文件无法实现随机存取，定长记录可以
- 定长记录、顺序结构的顺序文件可以快速检索(根据关键字快速找到记录)
- 最大缺点：不方便增加/删除记录

索引文件

- 建立一张索引表，每个记录对应一个表项。各记录不用保持顺序，方便增加/删除记录
- 索引表本身就是定长记录的顺序文件，一个索引表项就是一条定长记录，因此索引文件可支持随机存取
- 若索引表按关键字顺序排列，则可支持快速检索
- 解决了顺序文件不方便增/删记录的问题，同时让不定长记录的文件实现了随机存取。但索引表可能占用很多空间

索引顺序文件

- 将记录分组，每组对应一个索引表项
- 检索记录时先顺序查索引表，找到分组，再顺序查找分组
- 当记录过多时，可建立多级索引表

文件目录

文件目录的实现

- 一个文件对应一个FCB，一个FCB就是一个目录项，多个FCB组成文件目录
- 对目录的操作：搜索、创建文件、删除文件、显示文件、修改文件

目录结构

- 单级目录结构
  - 一个系统只有一张目录表，不允许文件重名
- 两级目录结构
  - 不同用户的文件可以重名，但不能对文件进行分类
- 多级(树形)目录结构
  - 不同目录下的文件可以重名，可以对文件进行分类，不方便文件共享
  - 系统根据"文件路径"找到目标文件
  - 从根目录出发的路径是"绝对路径"
  - 从"当前目录"出发的路径是"相对路径"
- 无环图目录结构
  - 在树形目录结构的基础上，增加一些指向同一节点的有向边，使整个目录成为一个有向无环图
  - 为共享结点设置一个共享计数器，计数器为0时才真正删除该结点

索引结点

- 除了文件名之外的所有信息都放到索引结点中，每个文件对应一个索引结点
- 目录项中只包含文件名、索引结点指针，因此每个目录项的长度大幅减小
- 由于目录项长度减小，因此每个磁盘块可以存放更多个目录项，因此检索文件时磁盘I/O的次数就少了很多

文件的物理结构(文件分配方式)

		How?	目录项内容	优点	缺点
顺序分配		为文件分配的必须是连续的磁盘块	起始块号、文件长度	顺序存取速度快，支持随机访问	会产生碎片，不利于文件拓展
链接分配	隐式链接	除文件的最后一个盘块之外，每个盘块中都存有指向下一个盘块的指针	起始块号、结束块号	可解决碎片问题，外存利用率高，文件拓展实现方便	只能顺序访问，不能随机访问。
	显式链接	建立一张文件分配表(FAT)，显式记录盘块的先后关系（开机后FAT常驻内存）	起始块号	除了拥有隐式链接的优点之外，还可通过查询内存中的FAT实现随机访问	FAT需要占用一定的存储空间
索引分配		为文件数据块建立索引表。若文件太大，可采用链接方案、多层索引、混合索引	链接方案记录的是第一个索引块的块号，多层/混合索引记录的是顶级索引块的块号	支持随机访问，易于实现文件的拓展	索引表需占用一定的存储空间。访问数据块前需要先读入索引块。若采用链接方案，查找索引块时可能需要很多次读磁盘操作。

顺序分配

- 为文件分配的必须是连续的磁盘块

链接分配

- 链接分配采取离散分配的方式，可以为文件分配离散的磁盘块。分为隐式链接和显式链接两种。

隐式链接

- 除文件的最后一个盘块之外，每个盘块中都存有指向下一个盘块的指针。文件目录包括文件第一块的指针和最后一块的指针。
- 优点：很方便文件拓展，不会有碎片问题，外存利用率高。
- 缺点：只支持顺序访问，不支持随机访问，查找效率低，指向下一个盘块的指针也需要耗费少量的存储空间。

显式链接

- 把用于链接文件各物理块的指针显式地存放在一张表中，即文件分配表(FAT，FileAllocation Table)。一个磁盘只会建立一张文件分配表。开机时文件分配表放入内存，并常驻内存。
- 优点：很方便文件拓展，不会有碎片问题，外存利用率高，并且支持随机访问。相比于隐式链接来说，地址转换时不需要访问磁盘，因此文件的访问效率更高。
- 缺点：文件分配表的需要占用一定的存储空间。

索引分配

- 链接方案
- 多层索引
- 混合索引

索引分配允许文件离散地分配在各个磁盘块中，系统会为每个文件建立一张索引表，索引表中记录了文件的各个逻辑块对应的物理块（索引表的功能类似于内存管理中的页表——建立逻辑页面到物理页之间的映射关系）。索引表存放的磁盘块称为索引块。文件数据存放的磁盘块称为数据块。
若文件太大，索引表项太多，可以采取以下三种方法解决：
①链接方案：如果索引表太大，一个索引块装不下，那么可以将多个索引块链接起来存放。缺点：若文件很大，索引表很长，就需要将很多个索引块链接起来。想要找到 i 号索引块，必须先依次读入 0~i-1 号索引块，这就导致磁盘I/O次数过多，查找效率低下。

②多层索引：建立多层索引（原理类似于多级页表）。使第一层索引块指向第二层的索引块。还可根据文件大小的要求再建立第三层、第四层索引块。采用 K 层索引结构，且顶级索引表未调入内存，则访问一个数据块只需要 K + 1 次读磁盘操作。缺点：即使是小文件，访问一个数据块依然需要K+1次读磁盘。

③混合索引：多种索引分配方式的结合。例如，一个文件的顶级索引表中，既包含直接地址索引（直接指向数据块），又包含一级间接索引（指向单层索引表）、还包含两级间接索引（指向两层索引表）。优点：对于小文件来说，访问一个数据块所需的读磁盘次数更少。

超级超级超级重要考点：①要会根据多层索引、混合索引的结构计算出文件的最大长度（Key：各级索引表最大不能超过一个块）；②要能自己分析访问某个数据块所需要的读磁盘次数（Key：FCB中会存有指向顶级索引块的指针，因此可以根据FCB读入顶级索引块。每次读入下一级的索引块都需要一次读磁盘操作。另外，要注意题目条件——顶级索引块是否已调入内存）

逻辑结构和物理结构

逻辑结构

- 用户(文件创建者)的视角看到的样子
- 在用户看来，整个文件占用连续的逻辑地址空间
- 文件内部的信息组织完全由用户自己决定，操作系统并不关心

物理结构

- 由操作系统决定文件采用什么物理结构存储
- 操作系统负责将逻辑地址转变为(逻辑块号，块内偏移量)的形式，并负责实现逻辑块号到物理块号的映射

文件存储空间管理

存储空间的划分与初始化

- 文件卷(逻辑卷)，目录区，文件区的概念
- 目录区包含文件目录、空闲表、位示图、超级块等用于文件管理的数据

空闲表法

- 空闲表中记录每个连续空闲区的起始盘块号、盘块数
- 分配时可采用首次适应、最佳适应等策略；回收时注意表项的合并问题

空闲链表法

空闲盘块链

- 以盘块为单位组成一条空闲链
- 分配时从链头依次取出空闲块，回收时将空闲块插到链尾

空闲盘区链

- 以盘区为单位组成一条空闲链
- 分配时可采用首次适应、最佳适应等策略；回收时注意相邻空闲盘区合并的问题

位示图法

- 一个二进制位对应一个盘块。(字号，位号)或(行号，列号)与盘块号一一对应
- 重要考点：要能够自己推出盘块号 -> (字号，位号)之间的相互转换公式
- 需要注意的题目条件
  - 二进制位0/1到底哪个代表空闲，哪个代表不空闲
  - 字号、位号、盘块号到底是从0开始还是从1开始

成组链接法

- UNIX采用的策略，适合大型文件系统。理解即可，不方便用文字描述的知识点也很难作为考题

文件的基本操作

创建文件

- 分配外存空间，创建目录项

删除文件

- 回收外存空间，删除目录项

打开文件

- 将目录项中的信息复制到内存中的打开文件表中，并将打开文件表的索引号返回给用户
- (打开文件时并不会把文件数据直接读入内存。"索引号"也称"文件描述符")
- 打开文件之后，对文件的操作不再需要每次都查询目录，可以根据内存中的打开文件表进行操作
- 每个进程有自己的打开文件表，系统中也有一张总的打开文件表
- 进程打开文件表中特有的属性：读写指针、访问权限(只读、读写)
- 系统打开文件表中特有的属性：打开计数器(有多少个进程打开了该文件)

关闭文件

- 将进程打开文件表中的相应表项删除
- 系统打开文件表的打开计数器减1，若打开计数器为0，则删除系统表的表项

读文件

根据读指针、读入数据量、内存位置，将文件数据从外存读入内存

写文件

根据写指针、写出数据量、内存位置，将文件数据从内存写出外存

文件共享

硬链接

- 各个用户的目录项指向同一个索引结点
- 索引结点中需要有链接计数count
- 某用户想删除文件时，只是删除该用户的目录项，且count--
- 只有count == 0时才能真正删除文件数据和索引结点，否则会导致指针悬空

软链接(符号链接)

- 在一个Link型的文件中记录共享文件的存放路径(Windows快捷方式)
- 操作系统根据路径一层层查找目录，最终找到共享文件
- 即使软链接指向的共享文件已被删除，Link型文件依然存在，只是通过Link型文件中的路径去查找共享文件会失败
- 由于用软链接的方式访问共享文件时要查询多级目录，会有多次磁盘I/O，因此用软链接访问共享文件的速度要比硬链接更慢

文件保护

口令保护

- 为文件设置一个"口令"，用户想要访问文件时需要提供口令，由系统验证口令是否正确
- 实现开销小，但"口令"一般存放在FCB或索引结点中(也就是存放在系统中)因此不太安全

加密保护

- 用一个"密码"对文件加密，用户想要访问文件时，需要提供相同的"密码"才能正确的解密
- 安全性高，但加密/解密需要耗费一定的时间

访问控制

- 用一个访问控制表(ACL)记录各个用户(或各组用户)对文件的访问权限
- 对文件的访问类型可以分为：读/写/执行/删除等
- 实现灵活，可以实现复杂的文件保护功能

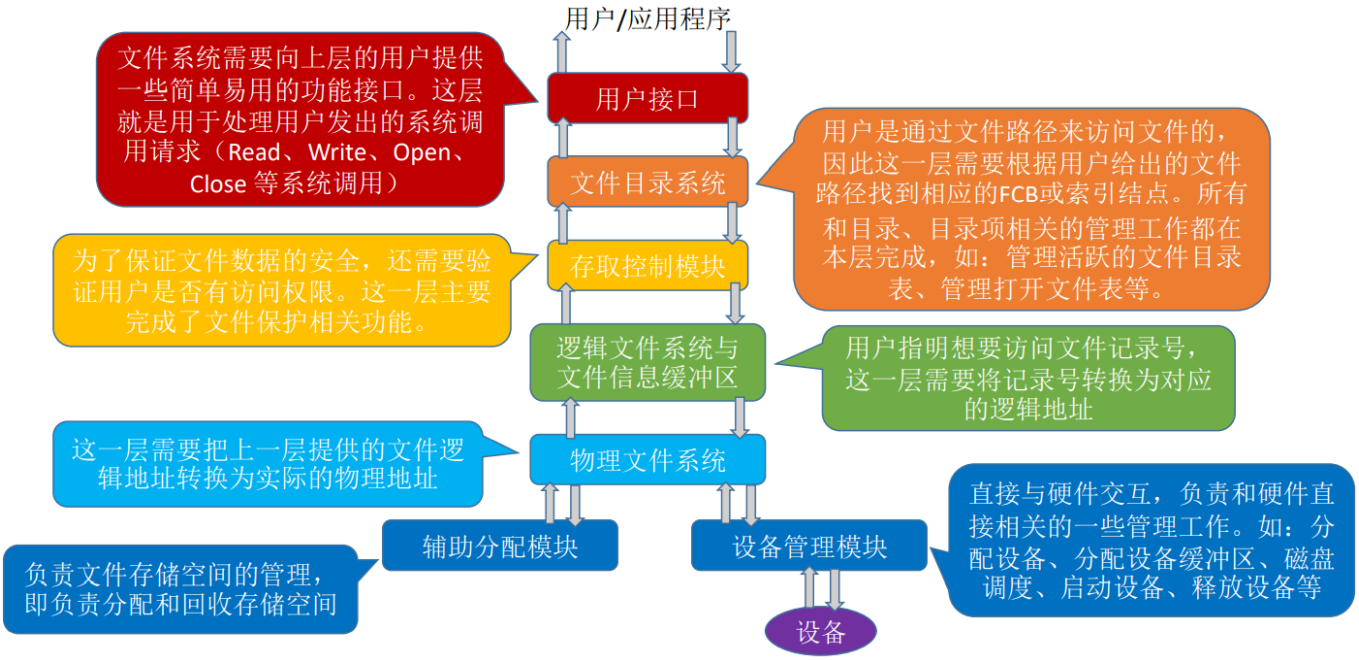
文件系统的层次结构

用一个例子来辅助记忆文件系统的层次结构：
假设某用户请求删除文件 “D:/工作目录/学生信息.xlsx” 的最后100条记录。

1. 用户需要通过操作系统提供的接口发出上述请求——用户接口
2. 由于用户提供的是文件的存放路径，因此需要操作系统一层一层地查找目录，找到对应的目录项——文件目录系统
3. 不同的用户对文件有不同的操作权限，因此为了保证安全，需要检查用户是否有访问权限——存取控制模块（存取控制验证层）
4. 验证了用户的访问权限之后，需要把用户提供的“记录号”转变为对应的逻辑地址——逻辑文件系统与文件信息缓冲区
5. 知道了目标记录对应的逻辑地址后，还需要转换成实际的物理地址——物理文件系统
6. 要删除这条记录，必定要对磁盘设备发出请求——设备管理程序模块
7. 删除这些记录后，会有一些盘块空闲，因此要将这些空闲盘块回收——辅助分配模块

用户接口

文件系统需要向上层的用户提供一些简单易用的功能接口。这层就是用于处理用户发出的系统调用请求(Read、Write、Open、Close等系统调用)



文件目录系统

用户是通过文件路径来访问文件的，因此这一层需要根据用户给出的文件路径找到相应的FCB或索引结点。所有和目录、目录项相关的管理工作都在本层完成，如：管理活跃的文件目录表、管理打开文件表等。

存取控制模块

为了保证文件数据的安全，还需要验证用户是否有访问权限。这一层主要完成了文件保护相关功能。

逻辑文件系统与文件信息缓冲区

用户指明想要访问文件记录号，这一层需要将记录号转换为对应的逻辑地址

物理文件系统

这一层需要把上一层提供的文件逻辑地址转换为实际的物理地址

辅助分配模块

负责文件存储空间的管理，即负责分配和回收存储空间

设备管理模块

直接与硬件交互，负责和硬件直接相关的一些管理工作。如：分配设备、分配设备缓冲区、磁盘调度、启动设备、释放设备

磁盘的结构

磁盘、磁道、扇区

- 磁盘由表面涂有磁性物质的圆形盘片组成
- 每个盘片被划分为一个个磁道，每个磁道又划分为一个个扇区

如何在磁盘中读/写数据

- 磁头移动到目标位置，盘片旋转，对应扇区划过磁道才能完成读/写

盘面、柱面

- 磁盘有多个盘片"摞"起来，每个盘片有两个盘面
- 所有盘面中相对位置相同的磁道组成柱面

磁盘的物理地址

- (柱面号，盘面号，扇区号)

磁盘的分类

- 根据磁头是否可移动
    - 固定头磁盘(每个磁道有一个磁头)
    - 移动头磁盘(每个盘面只有一个磁头)
- 根据盘片是否可更换
    - 固定盘磁盘
    - 可换盘磁盘

磁盘调度算法

一次磁盘读/写操作需要的时间

- 寻找时间(寻道时间)：启动磁臂、移动磁头所花的时间
- 延迟时间：将目标扇区转到磁头下面所花的时间
- 传输时间：读/写数据花费的时间

磁盘调度算法

先来先服务算法(FCFS)

- 按访问请求到达的先后顺序进行处理

最短寻找时间优先(SSTF)

- 每次都优先响应距离磁头最近的磁道访问请求
- 贪心算法的思想，能保证眼前最优，但无法保证总的寻道时间最短
- 缺点：可能导致饥饿

扫描算法(电梯算法、SCAN)

- 只有磁头移动到最边缘的磁道时才可以改变磁头移动方向
- 缺点：对各个位置磁道的响应频率不平均

循环扫描算法(C-SCAN)

- 只有磁头朝某个方向移动时才会响应请求，移动到边缘后立即让磁头返回起点，返回途中不响应任何请求

低频考点

- 若题目中无特别说明，则SCAN就是LOOK，C-SCAN就是C-LOOK

LOOK算法

- SCAN算法的改进，只要在磁头移动方向上不再有请求，就立即改变磁头方向

C-LOOK算法

- C-SCAN算法的改进，只要在磁头移动方向上不再有请求，就立即让磁头返回

减少延迟时间的方法

交替编号

- 具体做法：让编号相邻的扇区在物理上不相邻
- 原理：读取完一个扇区后需要一段时间处理才可以继续读入下一个扇区

错位命名

- 具体做法：让相邻盘面的扇区编号"错位"
- 原理：与"交替编号"的原理相同。"错位命名法"可降低延迟时间

磁盘地址结构的设计

- 理解为什么要用(柱面号，盘面号，扇区号)的结构
- 理解为什么不用(盘面号，柱面号，扇区号)的结构
- 原因：在读取地址连续的磁盘块时，前者更不需要移动磁头

磁盘的管理

磁盘初始化

- 低级格式化/物理格式化：划分扇区
- 磁盘分区(C盘、D盘、E盘)
- 逻辑格式化：建立文件系统(建立根目录文件、建立用于存储空间管理的数据结构)

引导块

- 计算机启动时需要运行初始化程序(自举程序)来完成初始化
- ROM中存放很小的自举装入程序
- 完整的自举程序存放在初始块(引导块)中

坏块的管理

- 简单的磁盘：逻辑格式化时将坏块标记出来
- 复杂的磁盘：磁盘控制器维护一个坏块链，并管理备用扇区

5 输入输出管理

概念和分类

I/O设备

- 将数据Input/Output(输入/输出)计算机的外部设备

按使用特性分类

- 人机交互类外部设备
- 存储设备
- 网络通信设备

按传输速率分类

- 低速设备
- 中速设备
- 高速设备

按信息交换的单位分类

- 块设备(传输快，可寻址)
- 字符设备(传输慢，不可寻址，常采用中断驱动方式)

I/O控制器

主要功能

- 接受和识别CPU发出的命令(要有 控制寄存器)
- 向CPU报告设备的状态(要有状态寄存器)
- 数据交换(要有数据寄存器，暂存输入输出的数据)
- 地址识别(由I/O逻辑实现)

组成

- CPU与控制器之间的接口(实现控制器与CPU之间的通信)
- I/O逻辑(负责识别CPU发出的命令，并向设备发出命令)
- 控制器与设备之间的接口(实现控制器与设备之间的通信)

两种寄存器编址方式

内存映射I/O

- 控制器中的寄存器与内存统一编制
- 可以采用对内存进行操作的指令来对控制器进行操作

寄存器独立编址

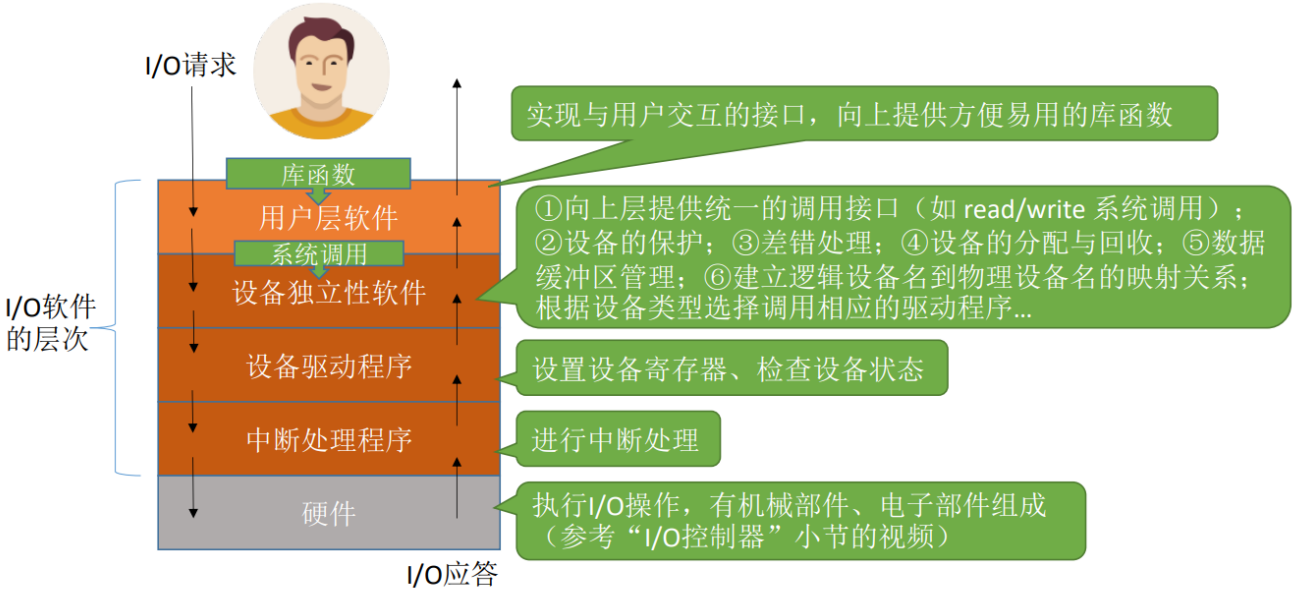
- 控制器中的寄存器独立编制
- 需要设置专门的指令来操作控制器

I/O控制方式

- 程序直接控制方式
- 中断驱动方式
- DMA方式
- 通道控制方式

	完成一次读/写的过程	CPU干预频率	每次I/O的数据传输单位	数据流向	优缺点
程序直接控制方式	CPU发出I/O命令后需要不断轮询	极高	字	设备→CPU→内存 内存→CPU→设备	每一个阶段的优点都是解决了上一阶段的最大缺点。 总体来说，整个发展过程就是要尽量减少CPU对I/O过程的干预，把CPU从繁杂的I/O控制事务中解脱出来，以便更多地去完成数据处理任务。
中断驱动方式	CPU发出I/O命令后可以做其他事，本次I/O完成后设备控制器发出中断信号	高	字	设备→CPU→内存 内存→CPU→设备	
DMA方式	CPU发出I/O命令后可以做其他事，本次I/O完成后DMA控制器发出中断信号	中	块	设备→内存 内存→设备	
通道控制方式	CPU发出I/O命令后可以做其他事。通道会执行通道程序以完成I/O，完成后通道向CPU发出中断信号	低	一组块	设备→内存 内存→设备	

I/O软件层次结构

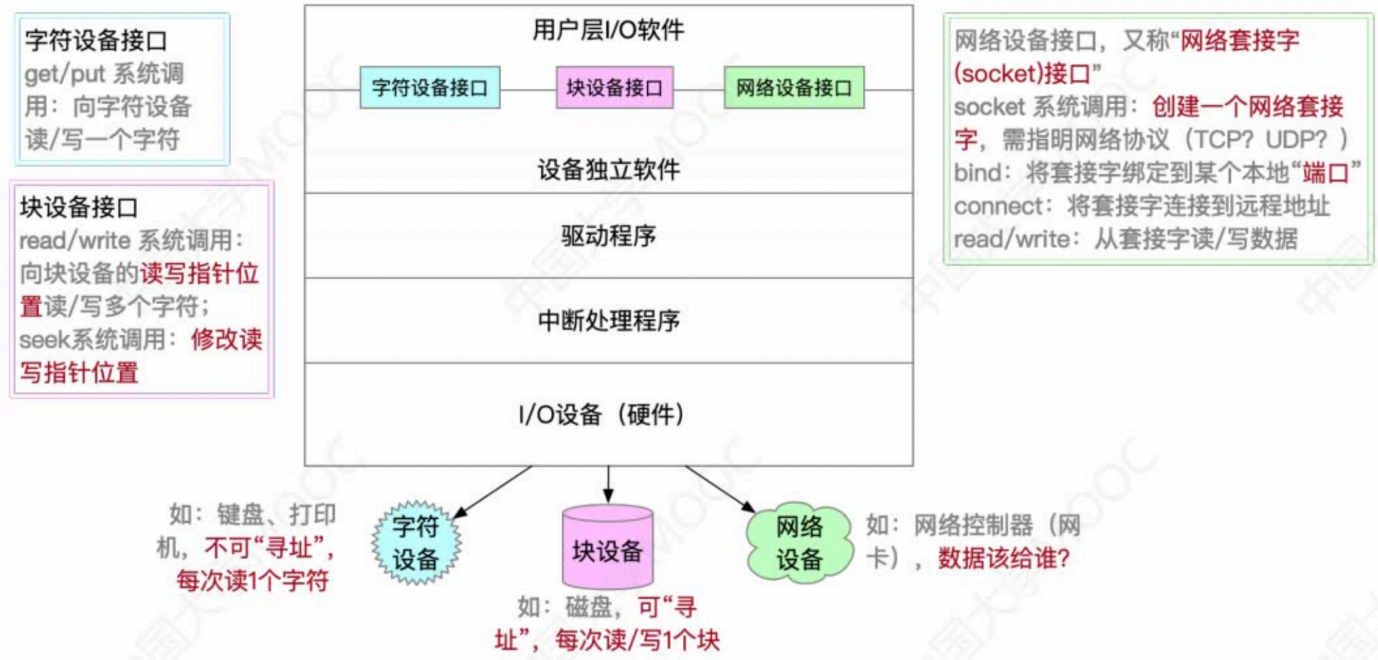


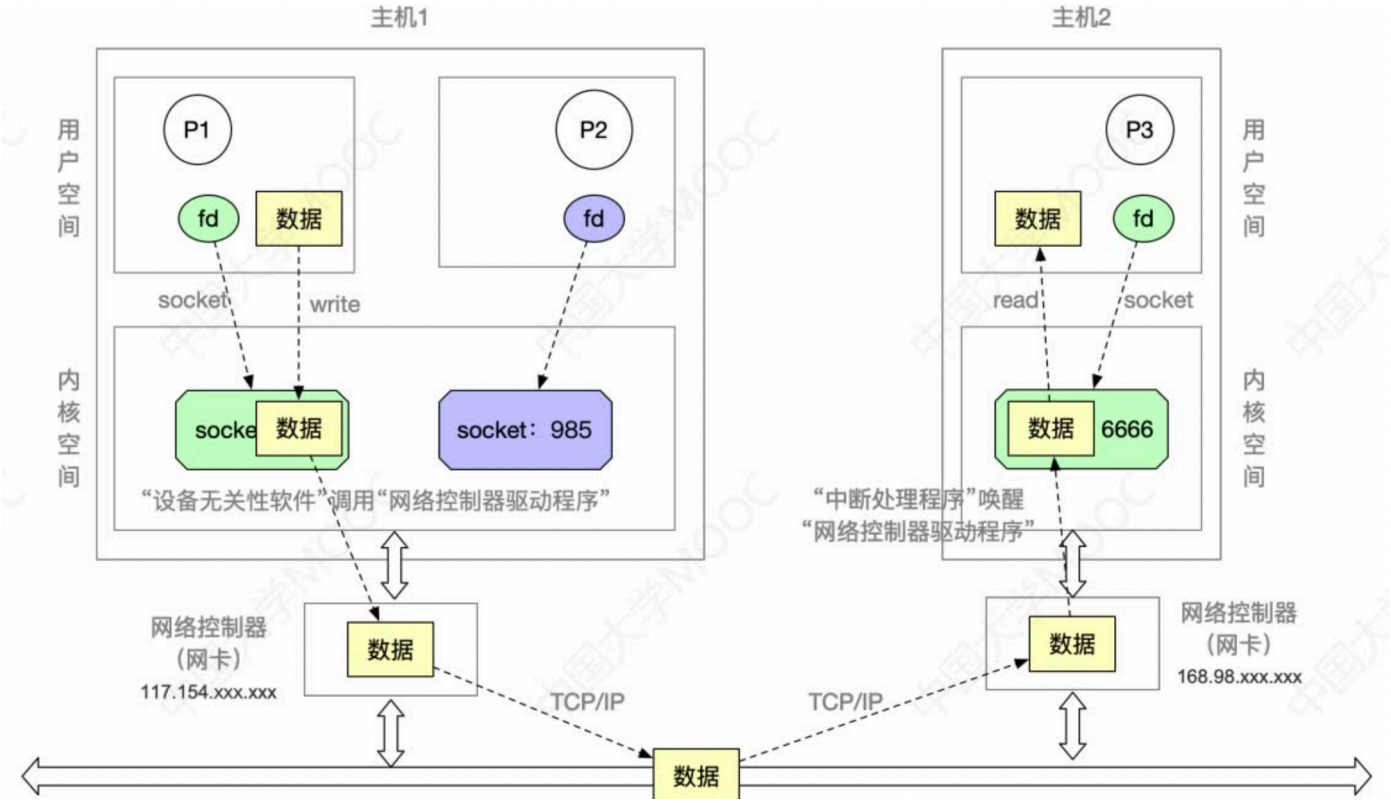
理解并记住I/O软件各个层次之间的顺序，要能够推理判断某个处理应该是在哪个层次完成的（最常考的是设备独立性软件、设备驱动程序这两层。只需理解一个特点即可：直接涉及到硬件具体细节、且与中断无关的操作肯定是在设备驱动程序层完成的；没有涉及硬件的、对各种设备都需要进行的管理工作都是在设备独立性软件层完成的）

- 用户层软件
- 设备独立性软件
- 设备驱动程序
- 中断处理程序
- 硬件

输入/输出应用程序接口

设备接口



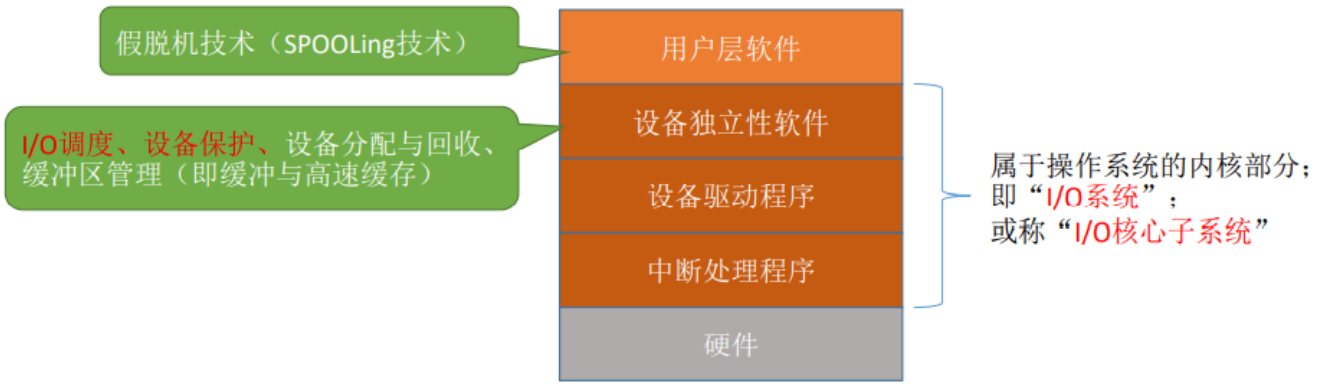


阻塞/非阻塞IO

阻塞I/O：应用程序发出I/O系统调用，进程需转为阻塞态等待。例如：字符设备接口：从键盘读一个字符。

非阻塞I/O：应用程序发出I/O系统调用，系统调用可迅速返回，进程无需阻塞等待。例如：块设备接口：往磁盘写数据write。

I/O核心子系统



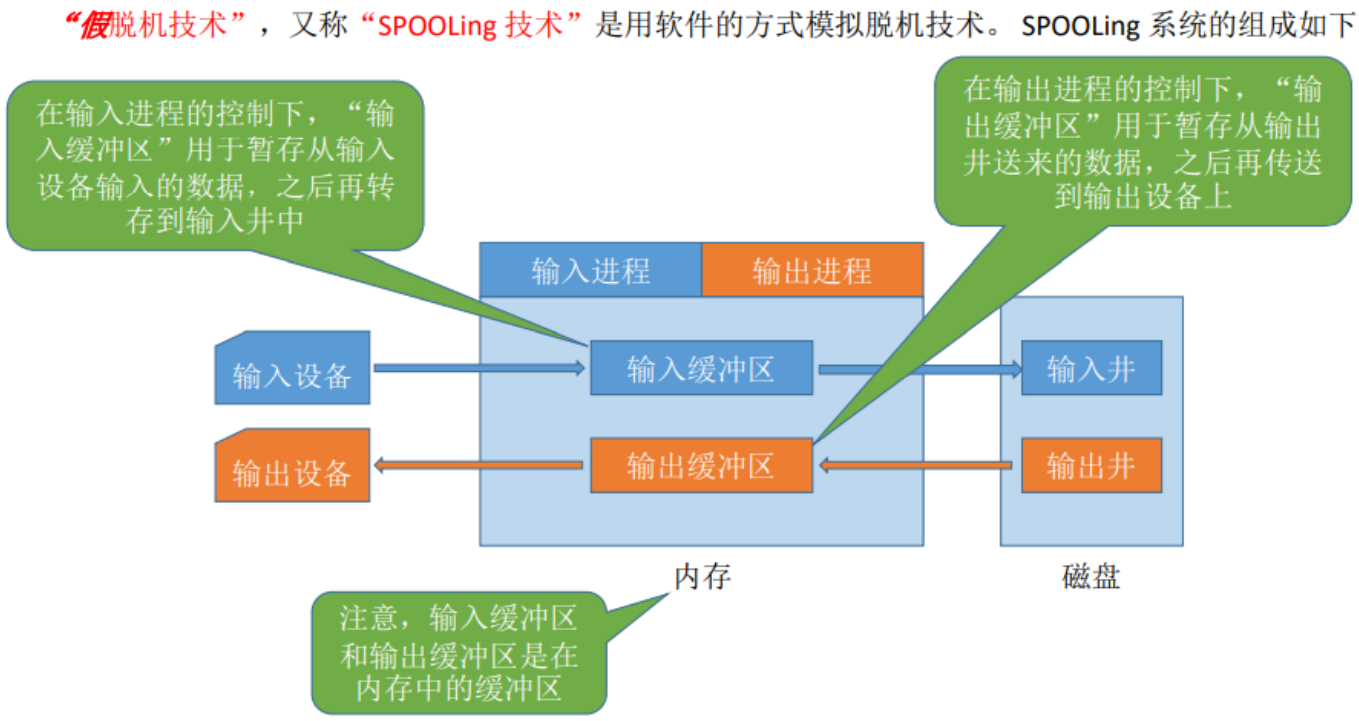
I/O调度

- 用某种算法确定一个好的顺序来处理各个I/O请求

设备保护

- 操作系统需要实现文件保护功能，不同的用户对各个文件有不同的访问权限(如：只读、读和写等)。

假脱机技术SPOOLing



脱机技术

外围控制机 + 更高速的设备：磁带
作用：缓解设备与CPU的速度矛盾，实现预输入、缓输出

假脱机技术

- 又叫SPOOLing技术，用软件的方式模拟脱机技术
- 输入井和输出井：模拟脱机输入/输出时的磁带
- 输入进程和输出进程：模拟脱机输入/输出时的外围控制机
- 输入缓冲区和输出缓冲区：内存中的缓冲区， 输入、输出时的"中转站"

共享打印机

- 用SPOOLing技术将独占式的打印机"虚拟"成共享打印机

设备驱动程序接口

操作系统规定好设备驱动程序的接口标准，各厂商必须按要求开发设备驱动程序

设备的分配与回收

考虑因素

- 固有属性
  - 独占设备
  - 共享设备
  - 虚拟设备(SPOOLing)
- 分配算法
  - 先来先服务、优先级高者优先、短任务优先等
- 安全性
  - 安全分配方式
    - 为进程分配一个设备后就将进程阻塞，本次I/O完成后才将进程唤醒。
  - 不安全分配方式
    - 进程发出I/O请求后，系统为其分配I/O设备，进程可继续执行，之后还可以发出新的I/O请求。只有某个I/O请求得不到满足时才将进程阻塞。

静态分配与动态分配

静态分配

- 进程运行前为其分配全部所需资源，运行结束后归还资源

动态分配

- 进程运行过程中动态申请设备资源

设备分配管理中的数据结构

设备控制表(DCT)

- 每个设备对应一张DCT，关键字段：类型/标识符/状态/指向COCT的指针/等待队列指针

控制器控制表(COCT)

- 每个控制器对应一张COCT，关键字段：状态/指向CHCT的指针/等待队列指针

通道控制表(CHCT)

- 每个控制器对应一张CHCT，关键字段：状态/等待队列指针

系统设备表(SDT)

- 记录整个系统中所有设备的情况，每个设备对应一个表目，关键字段：设备类型/标识符/DCT/驱动程序入口

设备分配的步骤

- ①根据进程请求的物理设备名查找SDT
- ②根据SDT找到DCT并分配设备
- ③根据DCT找到COCT并分配控制器
- ④根据COCT找到CHCT并分配通道
- 注：只有设备、控制器、通道三者都分配成功时，这次设备分配才算成功，之后便可启动I/O设备进行数据传送
- 缺点：用户编程时必须使用"物理设备名"，若换了一个物理设备，则程序无法运行。若进程请求的物理设备正在忙碌，则即使系统中还有同类型的设备，进程也必须阻塞等待

设备分配的步骤的改进

- 用户编程时使用逻辑设备名申请设备，操作系统负责实现从逻辑设备名到物理设备名的映射(通过LUT)
- 逻辑设备表的设置问题
    - 整个系统只有一张LUT：各用户所用的逻辑设备名不允许重复
    - 每个用户一张LUT：各个用户的逻辑设备名可重复

缓冲区管理

缓冲区的概念

- 缓冲区是一个存储区域，可以由专门的硬件寄存器组成，也可利用内存作为缓冲区。
- 作用
    - 缓和CPU与I/O设备之间速度不匹配的矛盾
    - 减少对CPU的中断频率，放宽对CPU中断响应时间的限制
    - 解决数据粒度不匹配的问题
    - 提高CPU与I/O设备之间的并行性

单缓冲

- 设备-(输入T)-缓冲区-(传送M)-工作区-(处理C)
- 处理一块数据平均耗时Max(C, T) + M
- 分析问题的初始状态：工作区满，缓冲区空

双缓冲

- 处理一块数据的平均耗时Max(T, C + M)
- 分析问题的初始状态：工作区空，一个缓冲区满，另一个缓冲区空

循环缓冲

- 多个缓冲区链接成循环队列，in指针指向第一个空缓冲区， out指针指向第一个满缓冲区

缓冲池

- 三个队列：空缓冲队列、输入队列、输出队列
- 四种工作缓冲区
    - 用于收容输入数据的工作缓冲区、用于提取输入数据的工作缓冲区
    - 用于收容输出数据的工作缓冲区、用于提取输出数据的工作缓冲区

固态硬盘