操作系统概述

计算机系统

完整的计算机由硬件和软件两部分组成
计算机硬件是指计算机系统中由电子、机械、光电组件组成的各种计算机部件和设备
计算机软件是指计算机系统中的程序、数据和有关的文档。
根据软件的作业可以将其分为系统软件、支撑软件和应用软件三类。
操作系统：操作系统是一组控制和管理计算机硬件和软件资源，合理地对各类作业进行调度、以及方便用户使用计算机的程序的集合
任务：管理计算机的资源
操作系统的目标：
方便性
有效性
3. OS 的作用
- OS 是用户与计算机硬件之间的接口；
- OS 是计算机系统资源的管理者；
- OS 是扩充裸机功能的软件，它实现了对计算机资源的抽象；
- OS 是计算机系统工作流程的组织者

计算机硬件

中央处理器、存储器、各种输入设备。

计算机软件

系统软件、应用软件和支撑软件三类。
系统软件支持和管理硬件你，它创立的是一个平台，如编译程序、装配程序、操作系统
应用软件是某个特定应用领域专用的软件
支撑软件是支撑其他软件的编址和维护，如中间件
计算机系统自上而下可分为四部分：硬件、操作系统、应用程序和用户。操作系统管理计算机硬件，为应用程序提供基础

操作系统的发展与分类

手工操作

单道批处理

多道批处理

1、 多道
2、 宏观上并行
3、 微观上串行
▲批处理首先考虑资源利用率和系统吞吐量，分时系统首先考虑交互能力和响应时间，实时系统首先考虑实时性和可靠性。
▲批处理系统的特点：资源利用率高，系统吞吐量大
无交互能力，作业平均周转时间长
▲多道程序设计技术：多道程序同时装入内存，允许他们并发运行。优点：提高 CPU、存储器、设备的资源利用率，增加系统吞吐量。

分时操作

按照时间片轮流将处理器分配给各联机作业使用。
1、 同时性
2、 交互性
3、 独立性
4、 及时性
分时系统的关键问题：人机交互
作业提交直接进入内存，引入时间片分时共享 CPU
响应时间 ≈ 进程数目 ×时间片大小
▲分时系统和实时系统的特征比较
交互能力： 分时系统强于实时系统
实时性： 实时系统优于分时系统
可靠性： 实时系统优于分时系统

实时操作系统

必须在决定的世家安内完成该事件的处理

微机操作系统

网络操作系统

分布式操作系统

嵌入式操作系统

操作系统的特性

并发性

两个或多个事件在同一时间间隔内同时发生，宏观上由多个程序在同时执行，微观上在单处理机系统中这多个程序时交替运行的。
并行性是同时刻发生的。
▲并发与并行的区别
并发性：指两个或多个事件在同一时间间隔内发生。如单处理器中的两个进程（宏观上同时，微观上交替）（只并发，但不并行）
并行性：两个或多个事件在同一时刻同时发生。如计算机中的 CPU 与 I/O 设备、I/O 设备与 I/O 设备的工作
一个程序的一次运行过程，每个进程设置一个 PCB，存放下一条要执行的指令地址。

共享性

虚拟性

异步性

操作系统的功能

处理器管理

即进程管理，对处理器进行分配调度

存储器管理

设备管理

文件管理

作业管理

提供用户接口

命令接口（联机命令、脱机命令）
应用程序接口（即系统调用，是 OS 提供的一组实现特殊功能的子程序，以供应用程序 取得 OS 的服务）。

图形接口

操作系统的内核结构

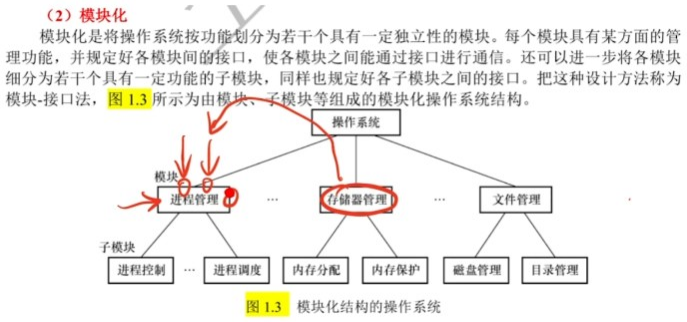
	特性、思想	优点	缺点
分层结构	内核分多层，每层可单向调用更低一层提供的接口	1. 便于调试和验证，自底向上逐层调试验证	1. 仅可调用相邻低层，难以合理定义各层的边界
		2. 易扩充和易维护，各层之间调用接口清晰固定	2. 效率低，不可跨层调用，系统调用执行时间长
模块化	将内核划分为多个模块，各模块之间相互协作。 内核 = 主模块+可加载内核模块 主模块：只负责核心功能，如进程调度、内存管理 可加载内核模块：可以动态加载新模块到内核，而无需重新编译整个内核	1. 模块间逻辑清晰易于维护，确定模块间接口后即可多模块同时开发	1. 模块间的接口定义未必合理、实用
		2. 支持动态加载新的内核模块（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	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	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. 内核外的某个功能模块出错不会导致整个系统崩溃	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	1. 降低了系统的一致性
		2. 减少了虚拟硬件资源的"映射层"，提升效率	2. 使系统变得更复杂

整体结构

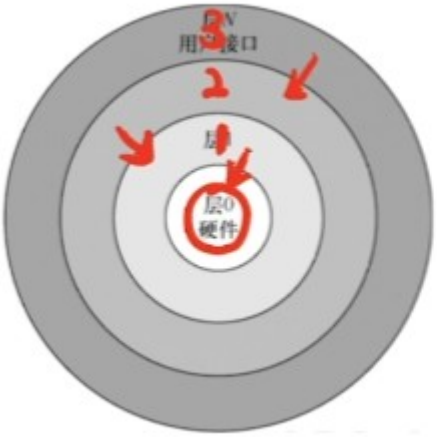
（实内核、单体结构模型、五结构模型）

模块结构

内核=主模块+可加载内核模块



层次结构



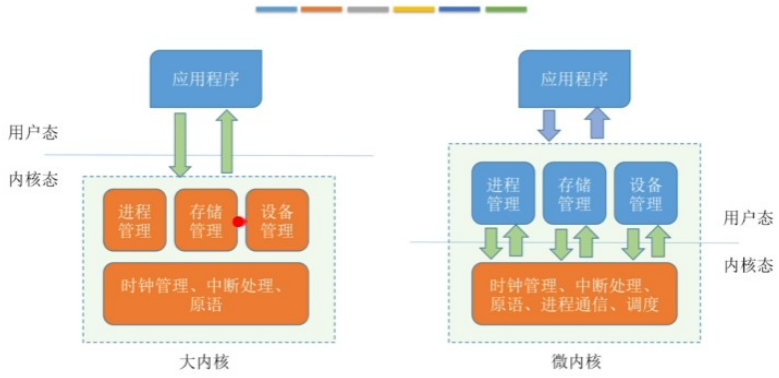
最底层是硬件，最高层是用户接口

每层可调用更低一层

微内核结构

（Client/Server 模型 ）

宏内核结构



OS 的其他分类方法

– 用户数量：单用户 OS，多用户 OS
– 任务数量：单任务 OS，多任务 OS
单用户单任务 OS，单用户多任务 OS，多用户多任务 OS
– 计算机体系结构：微机 OS，网络 OS，多处理机 OS，分布式 OS 等

操作系统的硬件基础

处理器

处理器指令

计算机的所有操作都是由处理器指令（机器指令或计算机指令）所决定的
每条处理器指令包含处理器执行所需的信息：操作码、源操作数、目的操作数和下一条指令地址
寻址方式：指令寻址和数据寻址

寄存器

CPU 的控制部件中，包含的寄存器有指令寄存器 IR 和程序计数器 PC、算术逻辑单元有累加器 ACC。包括通用寄存器、专用寄存器和控制寄存器。

处理器特权级

计算机系统中运行的程序可以分为两大类：操作系统的管理程序和用户程序。
即，CPU 上会运行两种程序，一种是操作系统内核程序，一种是应用程序。
多数系统将处理器特权级划分为管态和目态。
管态又称为系统态、核心态等，是操作系统管理程序运行时处理器所处状态。
所谓特权指令（清内存、外部设备输入输出、修改特殊寄存器、改变机器状态等）。使用系统中所有资源等等权限。
目态又称为用户态，时用户程序执行时处理器其所处的状态。

存储器

磁盘的工作原理。

1、 盘面。每个盘片又两个盘面。每一个这样的有效盘面都由一个盘面号，按顺序从上至下、从 0 开始依次编号

磁面号又叫磁头号，因为每一个有效盘面都有对应的读写磁头。

2、 磁道。磁盘在低级格式化时被划分成许多同心圆。这些痛惜暖的轨迹叫做磁道。信息以脉冲串的形式记录再写轨迹中。磁道由外向内，从 0 开始顺序编号。

每条磁道不是连续记录数据，而是划分成一段段圆弧。这些圆弧的角速度一样，线速度不一样。

每一段圆弧叫做一个删去，删去从 1 开始编号。每个删去中断数据作为一个单元读出或写入。

3、 柱面。所有盘面上的同一磁道构成一个圆柱，称作柱面。每个援助上的磁头由上而下从 0 开始编号 。数据的读写按柱面进行，即从 0 磁头开始进行从左。只有同一柱面上所有磁头全部读写完毕后，磁头才转移到下一柱面。

选取磁头只需通过电子切换，但是柱面必须通过急切切换。所以数据的读写按柱面进行，而不是盘面进行。

4．扇区。操作系统一扇区的形式将新抄袭存储在硬磁盘上。每个扇区包含两个主要部分：扇区标识符和存储数据的数据段。512B

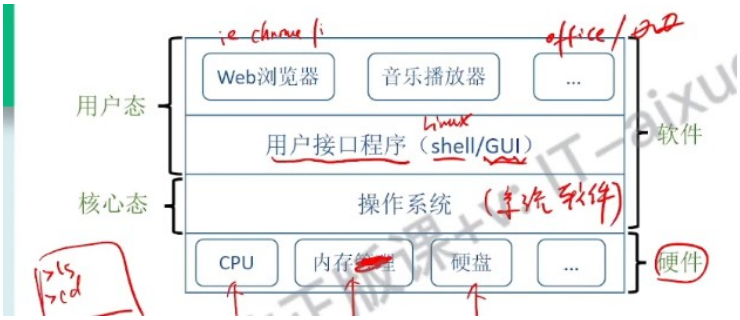
扇区标识符，又称为扇区头标，包括组成山区三维地质的三个数字：1、盘面好 2 柱面号 3 扇区号，也叫做块号

还有其余字段

CPU——进程

内存（主存）——存储器

磁盘/硬盘——文件系统

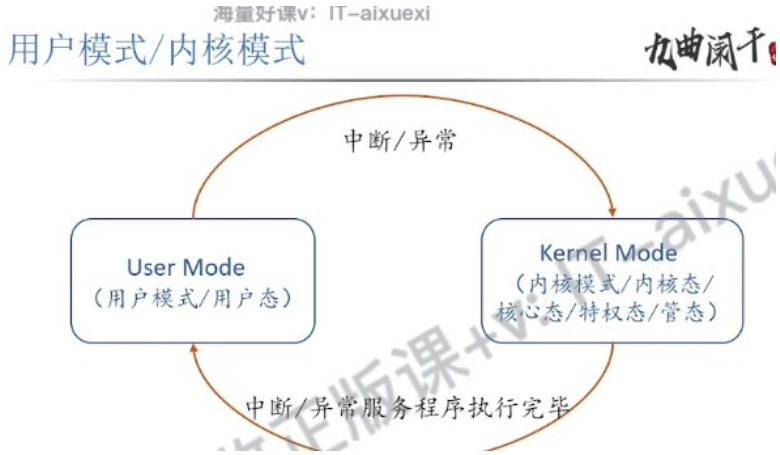


中断和时钟

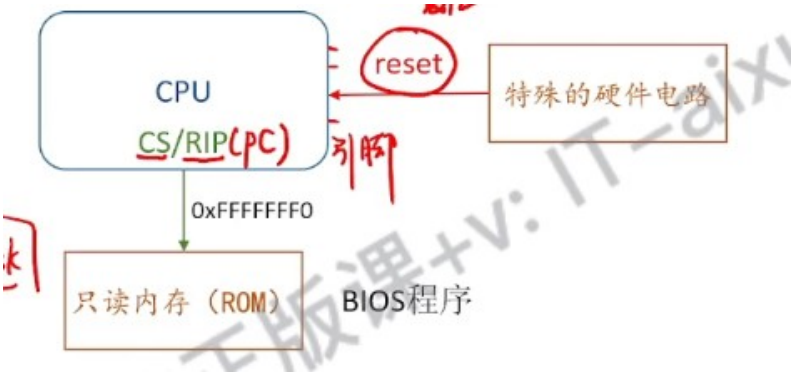
中断会使得 CPU 由用户态变为内核态，使操作系统重新夺回对 CPU 控制权。

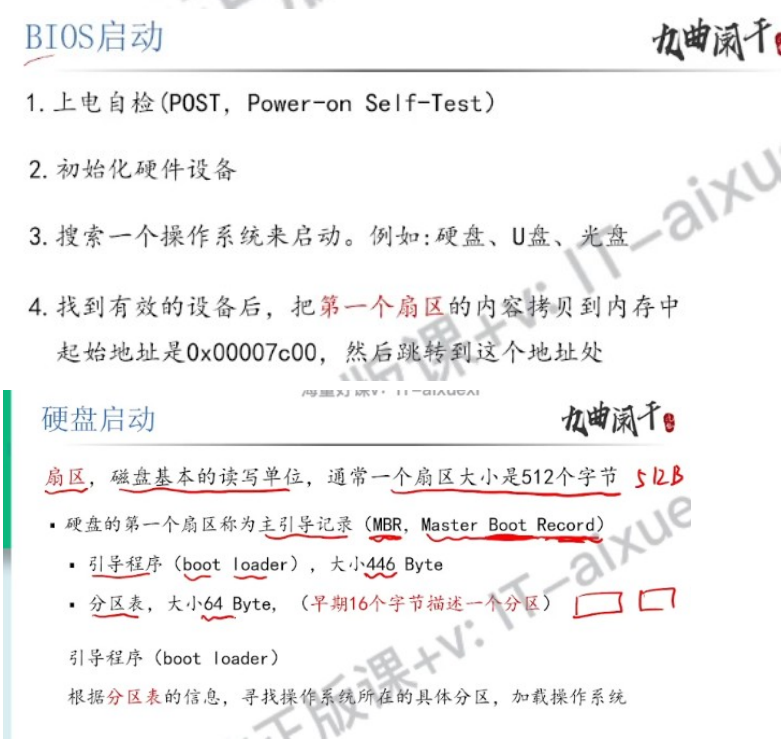
内核态->用户态，执行一条特权指令修改 PSW 的标志位为用户态。这个动作意味着操作系

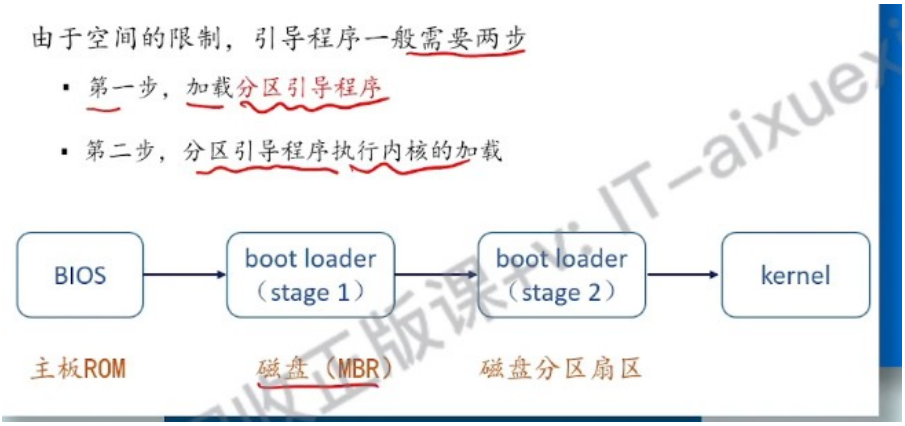
统将主动让出 CPU。
用户态->内核态：由中断引发，硬件自动完成变态过程，触发中断信号意味着操作系统夺取 CPU 的使用权。
响应中断时，保存到主存中。
硬件 PS 和 PC
中断/异常处理程序
所有通用寄存器
需要的其他信息：中断/异常、错误码
SS ： 段地址
ESP： 端内偏移量
按中断功能分类
1、 输入输出中断 I/O 传输结束或出错终端
2、 外中断：时钟中断、操作员控制中断、通信中断
3、 机器的故障中断：电源故障、主存取指令等
4、 程序性中断：（？）溢出、用户态下用核心态指令
5、 访管中断
要么是执行用户态中的代码——
要么执行核心态的代码——



系统调用
用陷入指令
操作系统的启动
操作系统存储在硬盘上
操作系统从硬盘加载到内存里去运行
CPU 执行的第一条指令来自哪里?
将 CS 和 RIP 复位
从 ROM（BIOS 程序）中读取 basic input/output system

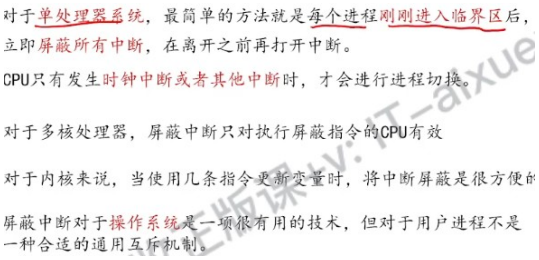




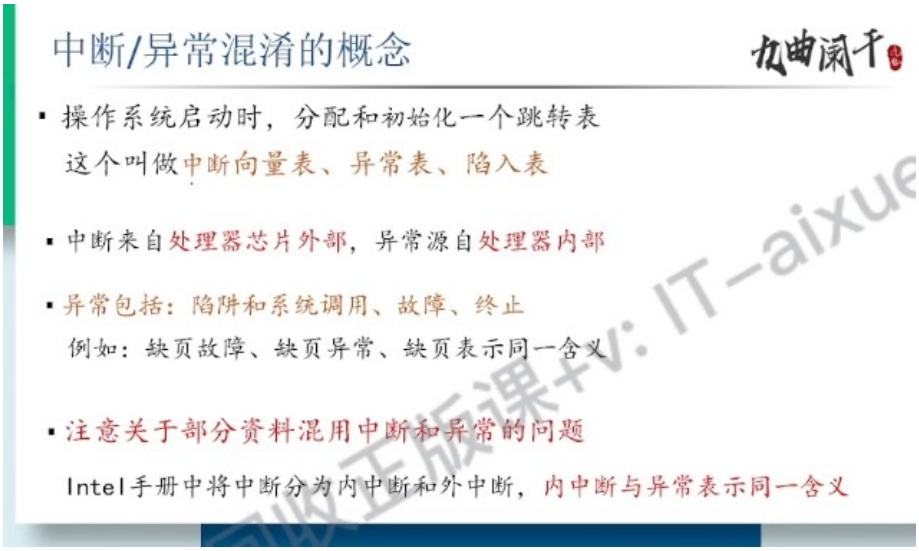


中断一般是异步的，由硬件随机产生，中断信号来自 CPU 外部。
异常一般是同步的，在特殊的或出错指令执行时由 CPU 控制单元产生，中断信号来源于 CPU 内部，内中断。
中断进一步可分为屏蔽中断和非屏蔽中断。可屏蔽中断是由程序控制器屏蔽行的中断，处

于屏蔽状态时，处理器忽略该类中断信号。
非屏蔽中断，不能由程序控制其屏蔽性。
异常又包括：
1、 处理器异常
故障
陷阱
异常终止
2、 编程异常
屏蔽中断



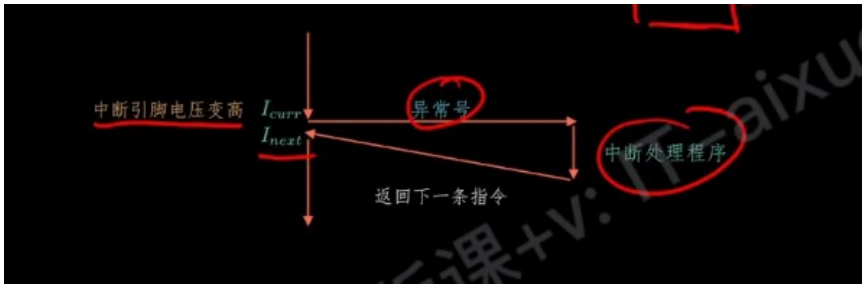
用户获得中断的权限会有系统风险。一个进程禁止中断后，一直没有开放中断，会影响系统的正常运行。



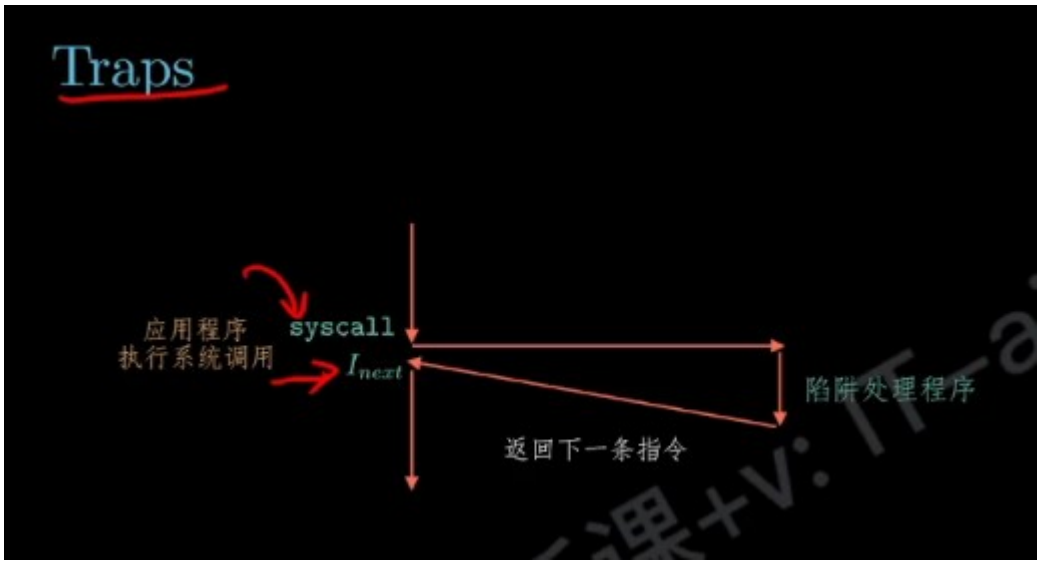
Class	Cause	Async/sync	Return behav
Interrupt	Signal from I/O device	Async	Always returns to nex
Trap	Intentional exception	Sync	Always returns to nex
Fault	Potentially recoverable error	Sync	Might return to curre
Abort	Nonrecoverable error	Sync	Never returns

中断

当前 CPU 执行完指令，中断引脚电压变高说明有中断
然后去响应中断
时钟中断
I/O 中断请求
不同中断信号
中断向量表
中断处理程序处于内核态

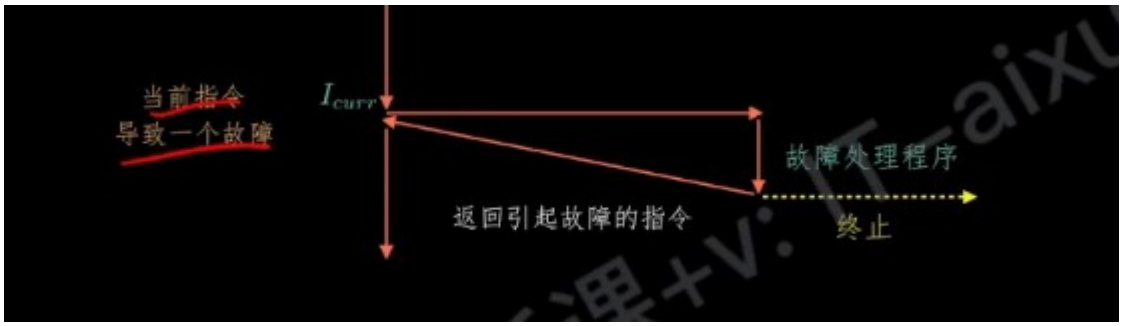


陷阱

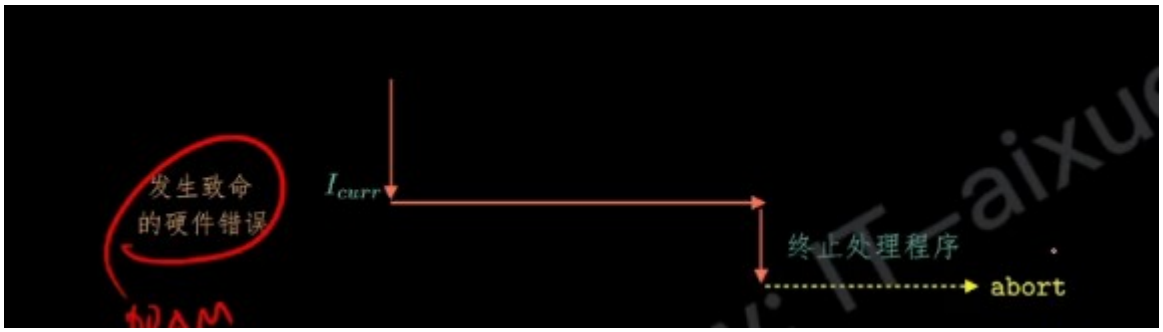


故障

缺页故障、缺页异常——地址不在内存中
保护异常——地址不合法
因此需要再次取一次故障指令

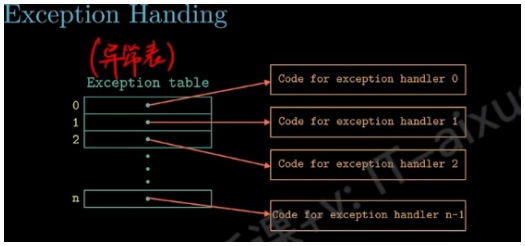


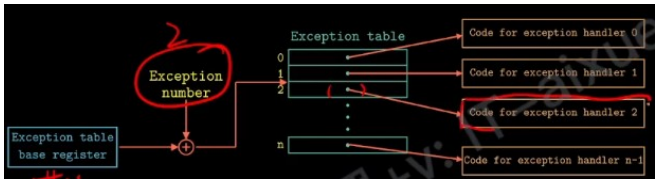
中止



异常表

存储在内存中
异常号相当于偏移量
找到异常处理程序





异常号	描述	异常类别
0	除法错误	故障
13	一般保护故障	故障
14	缺页	故障
18	机器检查	终止
32~255	操作系统定义的异常	中断或陷阱

图 8-9　x86-64 系统中的异常示例

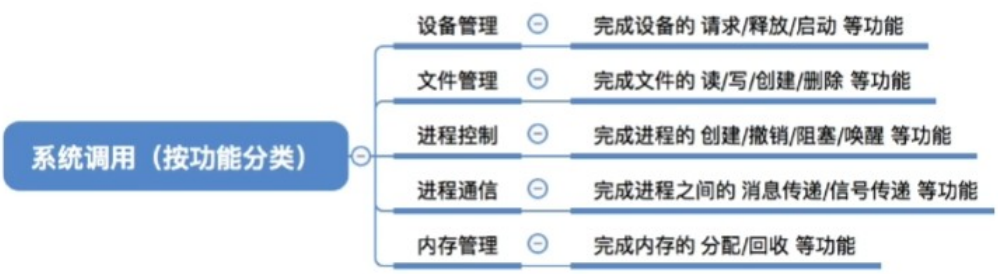
陷阱和系统调用

系统调用

操作系统作为用户和计算机硬件之间的接口，为方便用户调用这些内核服务功能需要提供一些简单易用的服务。主要包括命令接口和程序接口，其中程序接口由一组系统调用组成

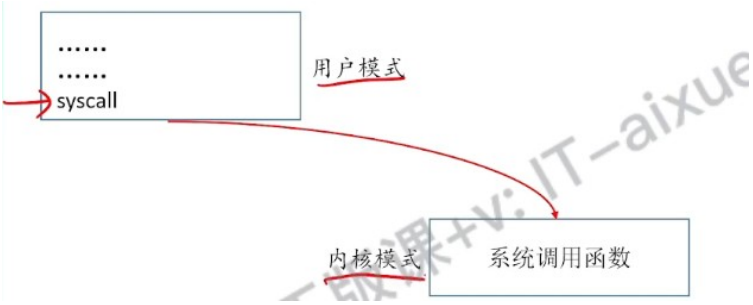
用户态和核心（内核）态
用户不能直接访问磁盘
只能通过操作系统提供的接口
操作系统替用户访问磁盘
执行系统调用（请求操作系统内核的服务），程序必须执行特殊的陷阱指令（syscall x86）
可以理解为一种可供程序调用的特殊函数，应用程序可以通过系统调用来获得操作系统内核的服务

普通应用程序	可直接进行系统调用，也可使用库函数。有的库函数涉及系统调用，有的不涉及
编程语言	向上提供库函数。有时会将系统调用封装成库函数，以隐藏系统调用的一些细节，使程序员编程更加方便。
操作系统	向上提供系统调用，使得上层程序能请求内核的服务
裸机	



由于应用程序运行在用户态，而系统调用服务例程运行在内核态，因此应用程序布恩那个直接调用内核发服务例程，而是通过执行一条叫做“访管指令”（陷入指令/trap 指令）的机器指令来实现调用的，这条指令的功能时引发一个编程异常，促使 CPU 从用户态切换到内核态，即系统调用入口程序
根据向量号查找中断描述表，得到该异常的中断处理程序入口地址
执行 system_call()函数，找到这个系统调用服务例程的入口地址

程序的特权级从用户模型提升至内核级
一旦进入内核，系统可以执行任何需要的特权操作
当系统完成后，执行从陷阱返回（return-from-trap）指令
因此陷阱指令是用户使用系统调用的窗口。
用户->read->访问磁盘



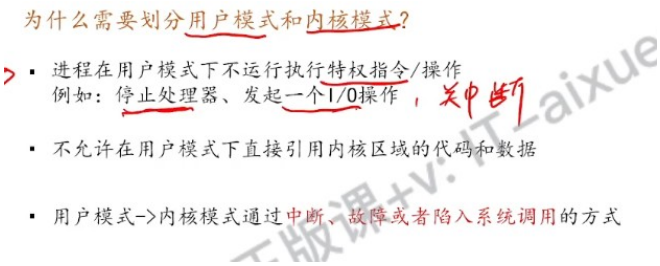
Sys_call 是在内核态
系统调用函数在内核态
执行哪一个系统调用？——系统调用号

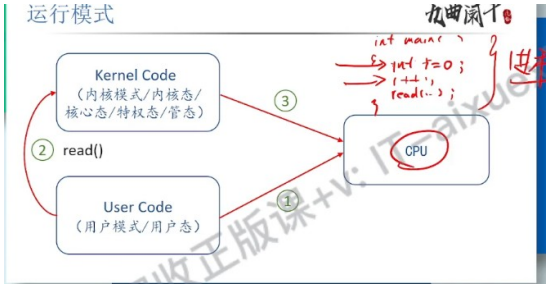
```
8  main:
9      movq   $1, %rax        write is system call 1
10     movq   $1, %rdi        Arg 1 : stdout is 1
11     movq   $string, %rsi   Arg 2 : hello world
12     movq   $len, %rdx      Arg 3 : string length
13     syscall  write )
14     movq   $60, %rax       _exit is system call 60
15     movq   $0, %rdi
16     syscall
```

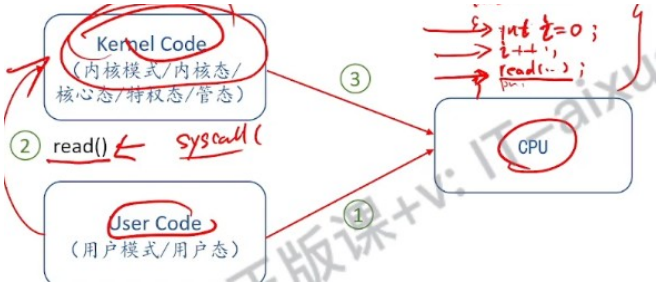
编号	名字	描述	编号	名字	描述
0	read	读文件	33	pause	挂起进程直到信号到达
1	write	写文件	37	alarm	调度告警信号的传送
2	open	打开文件	39	getpid	获得进程 ID
3	close	关闭文件	57	fork	创建进程
4	stat	获得文件信息	59	execve	执行一个程序
9	mmap	将内存页映射到文件	60	_exit	终止进程
12	brk	重置堆顶	61	wait4	等待一个进程终止
32	dup2	复制文件描述符	62	kill	发送信号到一个进程

图 8-10　Linux x86-64 系统中常用的系统调用示例

控制寄存器有一位记录了当前模式是用户模式还是内核模式







进程管理

引入进程的原因

1. 引入进程的原因

- 为了提高资源利用率必须让多个程序并发运行。
- 程序的并发运行出现了新的特征：间断性、 失去封闭性、 不可再现性。
- 为了让程序能正确并发运行，引入进程概念。

操作系统的代码是所有进程的公共代码

运行模式和进程切换

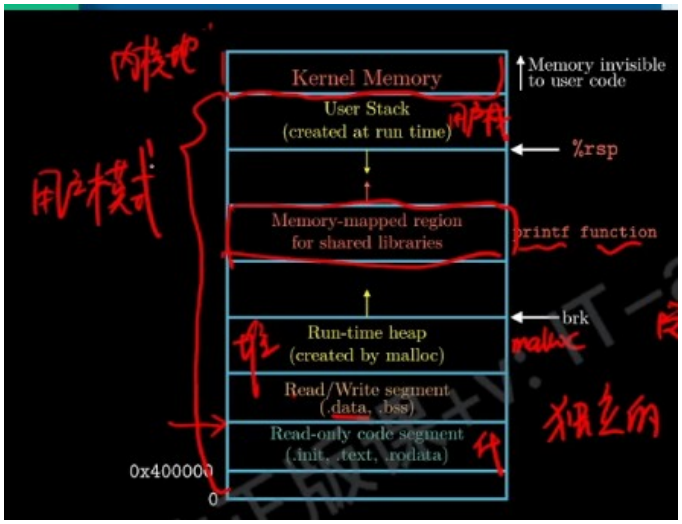
如何切换进程?

要从用户模式切换到内核模式
访问了内核区域，触发一般保护故障

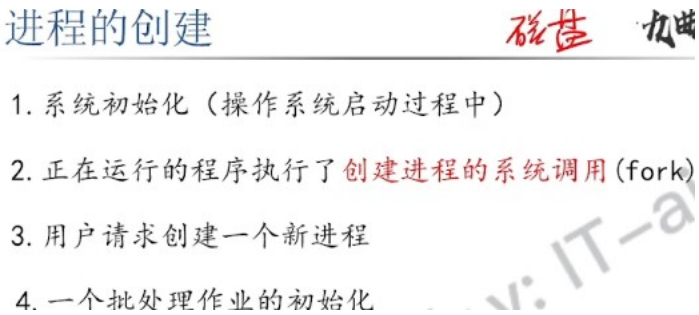
进程：一个正在运行程序的抽象，程序的一次执行。
程序：指令的有序集合。
–

- Linux：创建进程使用系统调用 fork()+ execve()
- Windows：CreateProcess()
- 父进程 —— shell
- 子进程 —— hello

在./hello->shell 中 fork()+execute()
父进程和子进程具有独立的但是相同的虚拟地址空间
Init——最初的父进程
树形的关系
进程运行时有独立的虚拟地址空间



进程的创建



程序是静态的
但是一个程序对应一个进程
进程是具有一定独立功能的程序关于讴歌数据集合的一次运行过程，是系统进行资源分配和调度的一个独立单位。
进程的特征
结构特征
动态性
并发性
独立性
异步性
结构性

程序的并发执行：多个程序共享资源，并发运行
1、 间断性。并发程序并不是一气呵成的，中间总会因此彼此间的各种制约关系出现暂停，因为系统只有一个 CPU
2、 失去封闭性而导致程序运行结果不可再现性，即对没有对资源的互斥共享
3、 静态程序结构不能支持并发运行的实现
进程控制
分配内存资源、回收内存资源、控制状态转换
进程互斥
互斥方式：多个进程载访问某些共享资源（临界资源）应采用互斥的方式访问
同步方式：多个进程相互合作完成一些共同任务，前驱满足后继
进程通信近九成之间的信息交换
调度
对资源或人物进行合理分配和管理
能够后背队列中按照一定的算法选择若干个作业调入内存，为他们创建进程，分配必要资源，插入就绪队列

4. 进程与程序的区别
(1)从定义上看，程序是一组指令的有序集合；进程是程序的运行过程；
(2)从结构上看，进程不仅包含程序段，还包含数据段和 PCB；
(3)进程是动态性，而程序是静态的；

(4)进程可独立地、并发地执行，程序则不能独立、并发执行

5. 进程与程序的对应关系

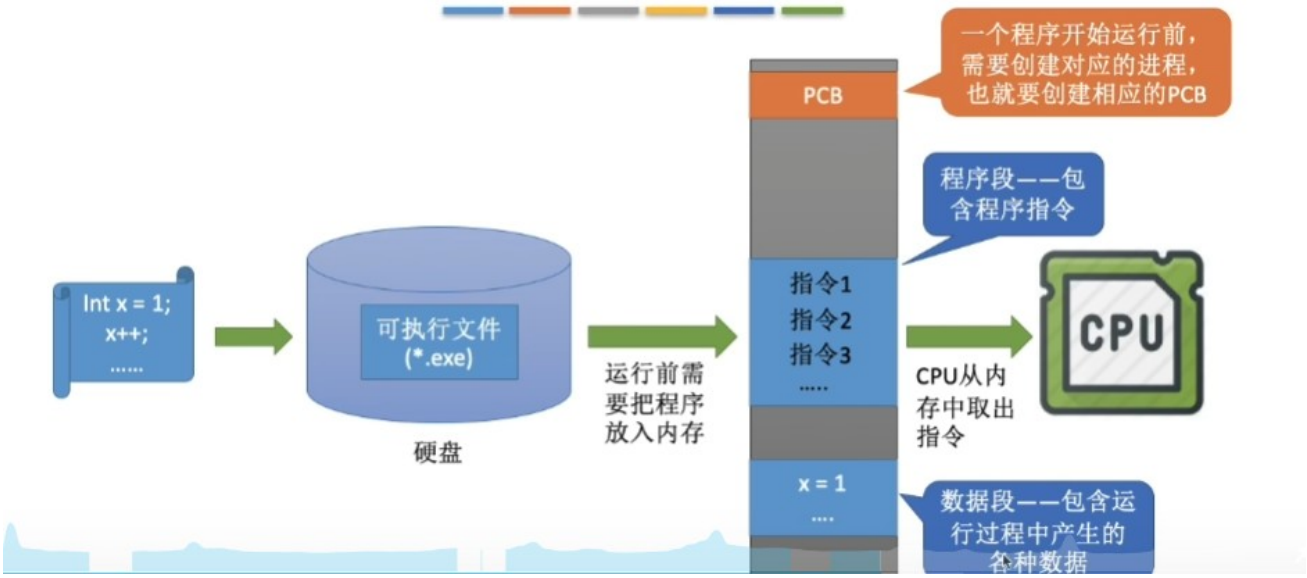
- 在某个时刻一个进程对应于一个程序；
- 在整个生命周期中，进程可执行多个程序；(fork+exec)
- 一个程序多次执行则将对应多个进程；

进程控制块/进程表

PCB（struct）

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

进程映像是指进程实体的组成。
程序（段）正文段——代码段 CS
数据集数据段 DS
（有的包含栈堆栈段 SP
）
PCB 进程描述信息
进程是动态的，进程是进程实体（进程映像）的组成



进程状态

进程被创建后，进入就绪队列，等待被调度执行

8. 内核态与用户态

CPU 指令（特权指令，非特权指令）

▲特权指令：关机指令、清主存、启动外设指令、设置系统时钟时间、关中断、修改存储器管理寄存器等

▲非特权指令：通用寄存器清 0 指令，访问内存指令，算术运算指令等

CPU 的执行状态

内核态（核心态、系统态、管态）：能访问所有的内存空间和 I/O 端口，能执行特权和非特权指令。

用户态（目态）只能访问分配给自己的内存空间，只能执行非特权指令。

进程的三（五）种状态：

运行

就绪

阻塞

创建状态和终止状态

进程状态

进程被创建后，进入就绪队列
等待被调度执行

进程的三种状态：

- 运行
- 就绪
- 阻塞

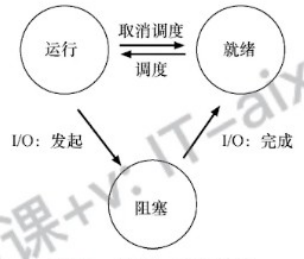
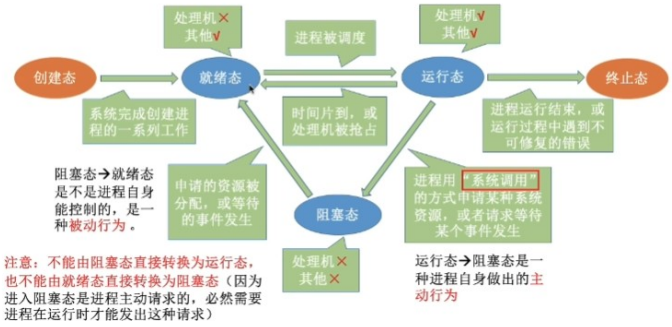
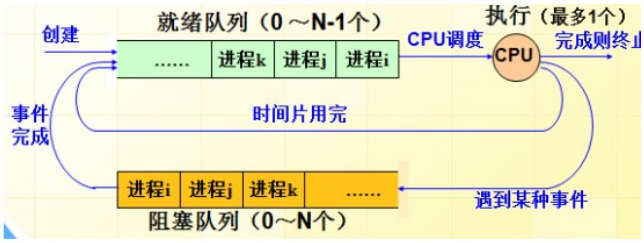


图 4.2　进程：状态转换

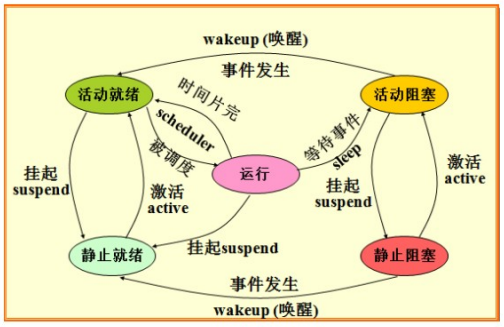


7. 单 CPU 中 N 个进程的情况



具有挂起状态的进程状态转换

具有挂起状态的进程状态转换

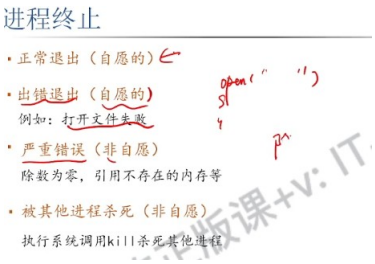


进程终止

跟踪进程状态

表 4.2　跟踪进程状态：CPU 和 I/O

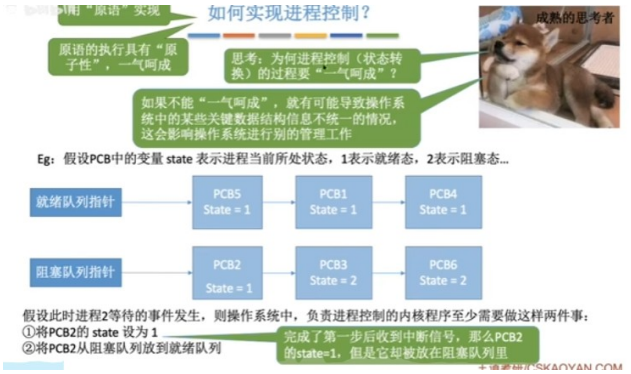
时间	Process0	Process1	注
1	运行	就绪	
2	运行	就绪	
3	运行	就绪	Process0 发起 I/O
4	阻塞	运行	Process0 被阻塞
5	阻塞	运行	所以 Process1 运行
6	阻塞	运行	
7	就绪	运行	I/O 完成
8	就绪	运行	Process1 现在完成
9	运行	—	
10	运行	—	Process0 现在完成



单 CPU，所以一次只能有一个进程运行（交给 CPU）

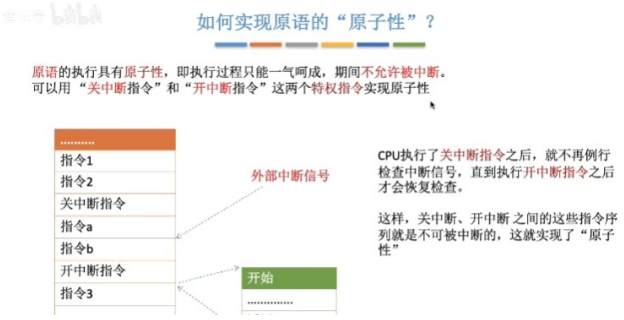
$$T_{周转时间} = T_{完成时间} - T_{到达时间}$$

$$T_{周转时间} = T_{等待时间} + T_{执行时间}$$



但是就绪队列还是可以有多个进

周转时间：一个是性能指标，另外一个是指标时公平
性能和公平中往往是矛盾的
如何实现原语的原子性？



OS 的内核运行于核心态，应用程序则运行于用户态。（进程控制的大量原语）

进程通信

1、 共享存储器系统通信

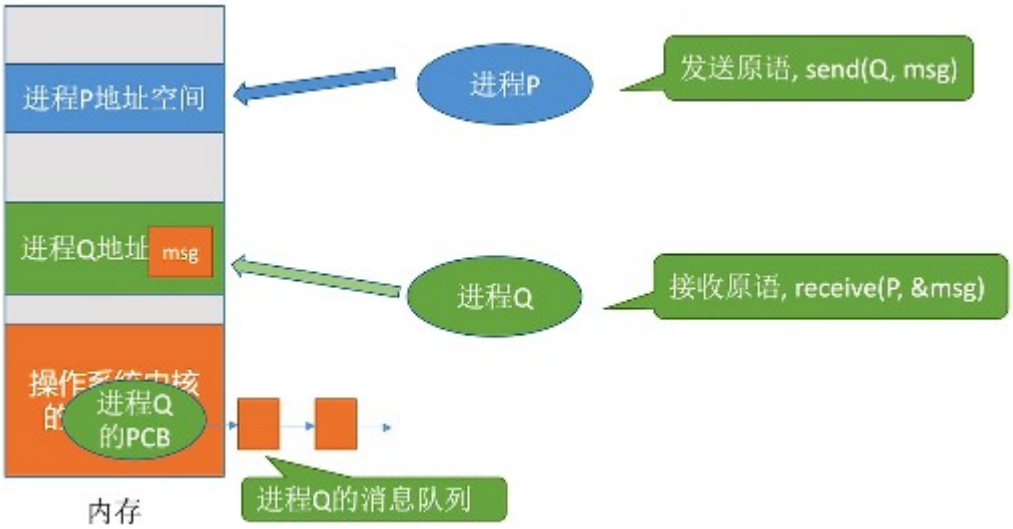
在存储器中划分出一块共享存储区，诸进程通过对共享存储区的读写操作来实现通信

Shm_open()　　通过系统调用，申请一片共享内存区
Void * map()　　通过 mmap 系统调用，将共享内存区映射到自己的地址空间

同步互斥工作
基于数据的共享
基于存储区的共享
高级通信方式

2、 消息传递系统通信

以格式化的信息为单位
直接通信方式
由一对通信原语 send(),receive()
间接通信方式
进程 Q 的消息队列
如果其他进程要发送给进程 Q 的消息都在 Q 的消息队列中
进程 P 建立消息 msg，发送原语,send(Q,msg)
被复制到了 Q 的消息队列中
接受原语（receive (p,&msg)）



间接通信方式，以“信箱中作为中间实体进行消息传递
Send (A,msg)发送到哪个信箱

3、 管道通信

管道式一个特殊的共享问题件，又名为 pipe 文件。就是在内存中开辟一个大小固定的内存缓冲区
先进先出的
单向通信的
管道文件被读取的部分会消失
无名管道
有名管道

4、 客户服务通信
消息缓冲队列通信机制

进程控制

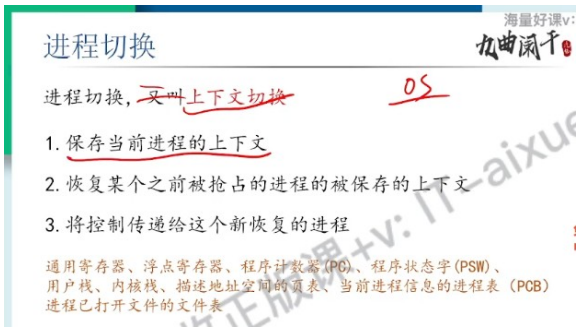
进程创建

作业调度
用户登录
提供特定服务
应用请求

进程创建原语

进程撤销

进程切换



进程阻塞与唤醒

进程调度

调度的层次

高级：负责将进程调入内存，分配资源。从外存的后被队列中选择若干个作业调入内存，创建进程，并将新创建的进程插入就绪队列，准备执行；此外，当作业执行完毕后回收进程。
中级：提高内存利用率和 CPU 吞吐量。将进程换出到外存，挂起状态。
低级：负责分配 CPU 资源
调度频率：低级>中级>高级

进程调度是操作系统在内核模式中进行

忙等待

While(1)
用户态一直占用 CPU，无法切换到内核模式，就无法进行进程的切换。
时钟中断
CPU 响应中断，处理中断处理程序
从用户态切换到内核态
用户能关中断的话，就可以一直占用，这是不行的。

进程调度方式

非抢占式
抢占式
内核完全不可抢占
内核部分抢占
内核完全可抢占

考虑的目标

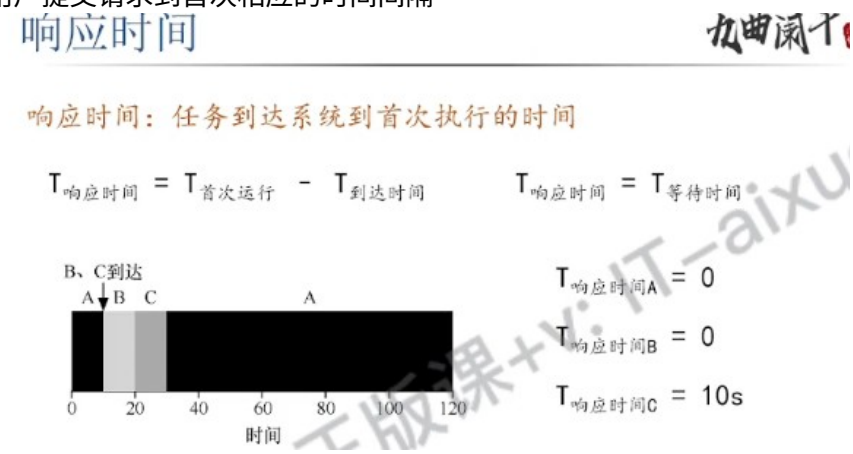
系统设计目标
调度的公平性
资源的均衡利用
合理的系统开销

评价指标

CPU 利用率 忙碌时间/总时间
甘特图
系统吞吐量

总共完成了多少道作业/总共花了多少时间
周转时间和带权周转时间
周转时间——从作业提交给系统开始，到作业完成为止的这段时间间隔
它包括四部分：作业在外存后备队列上等待作业调度（进入到内存并创建进程）的时间、进程在就绪队列上等待（调入到 CPU）进程调度、进程在 CPU 上执行的时间，进程等待 I/O 操作完成的时间。后三项可能在一个作业处理过程中可能发生多次。
周转时间=作业完成时间-作业提交时间
平均周转时间——各作业周转时间之和/作业数
带权周转时间——作业周转时间/实际运行的时间=作业完成时间-作业提交时间/作业实际运行的时间
平均带权周转时间
等待时间——进程/作业处于等待处理机时间之和
对于进程来说，等待时间就是指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
响应时间

用户提交请求到首次相应的时间间隔



对截止时间的保证

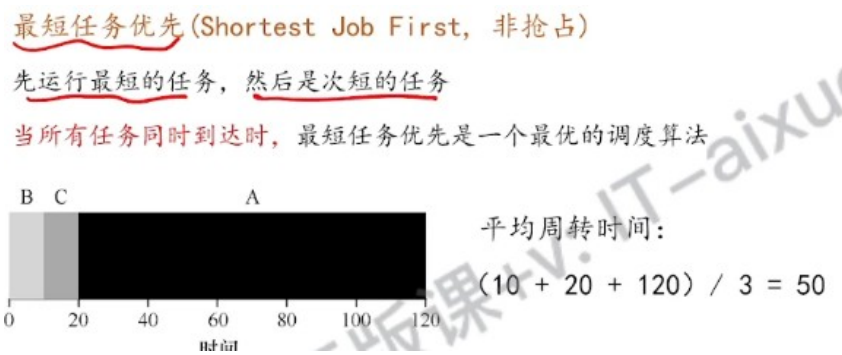
调度算法

周转时间=完成时间-提交时间
带权周转时间 周转时间/要求执行时间

先来先服务 FCFS

ABC 同时到达就绪队列
缺点：有些任务先到达到时间非常长，系统平均周转时间比较高

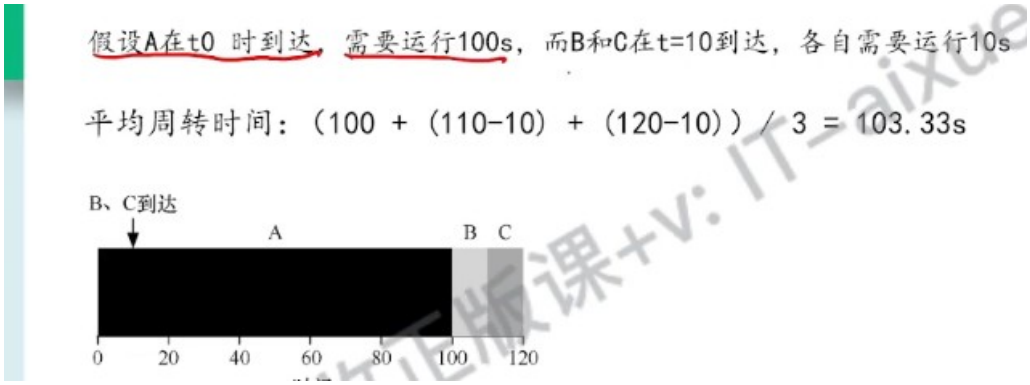
短作业优先



不会产生饥饿现象

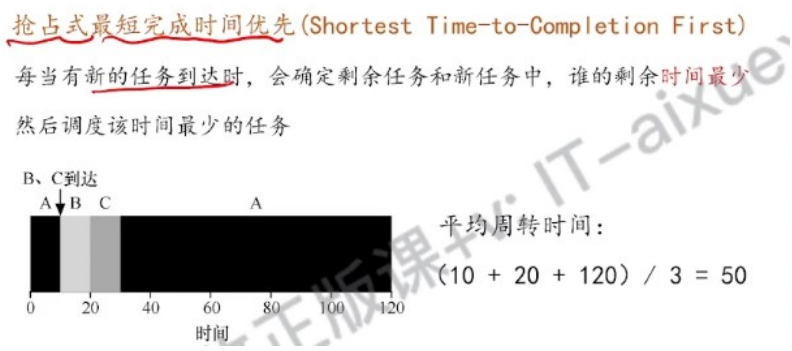
非抢占式

短作业优先 SJF



抢占式

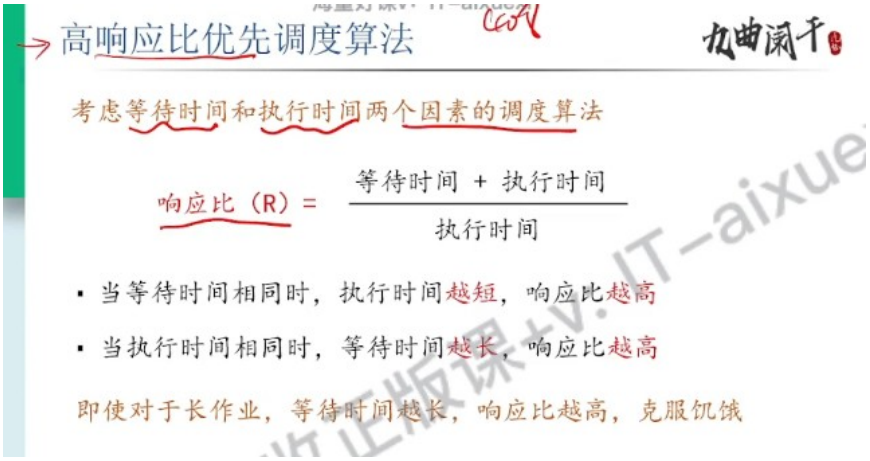
最短剩余时间优先算法 SRTN
需要比较该进程的下一次运行时间是否比当前运行进程剩余运行时间段，如果是，则抢占当前运行进程的 CPU



产生饥饿现象

高响应比优先调度算法

等待时间和执行时间两个因素



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

优先级调度算法
抢占式、非抢占都有
非抢占式，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
抢占式还需在就绪队列变化时，检查是否发生抢占

P2 先到达，先上处理机

确定优先级?
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
与 I/O 型进程相对的时计算型进程（CPU）
动态提升?
线程 I/O 操作结束提升优先级
在就绪队列中随等待时间延长而提升
随占用 CPU 时间延长而降低
随剩余运行时间缩短而提升
完成 I/O 操作后提升

时间片调度（轮转调度 RR）

时间片调度（轮转调度，round robin）

每个进程被分配一个时间段，称为时间片，允许进程在该时间段内执行

如果在时间片内结束时，该进程没有执行完，接下来会将CPU分配另外进程执行

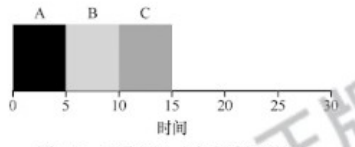


图 7.6 又是 SJF（响应时间不好）

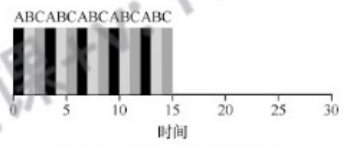


图 7.7 轮转（响应时间好）

时间片大小为 2（注：以下括号内表示当前时刻就绪队列中的进程、进程的剩余运行时间）

0时刻（P1(5)）:0时刻只有P1到达就绪队列，让P1上处理机运行一个时间片

2时刻（P2(4) → P1(3)）:2时刻P2到达就绪队列，P1运行完一个时间片，被剥夺处理机，重新放到队

此时P2排在队头，因此让P2上处理机。（注意：2时刻，P1下处理机，同一时刻新进程P2到达，如

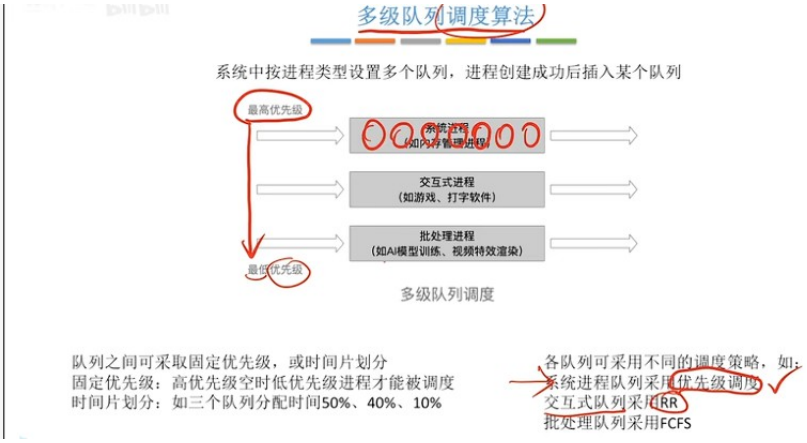
题目中遇到这种情况，默认新到达的进程先进入就绪队列）

7时刻（P2(2) → P4(6) → P1(1)）：虽然P3的时间片没用完，但是由于P3只需运行1个单位的时间，

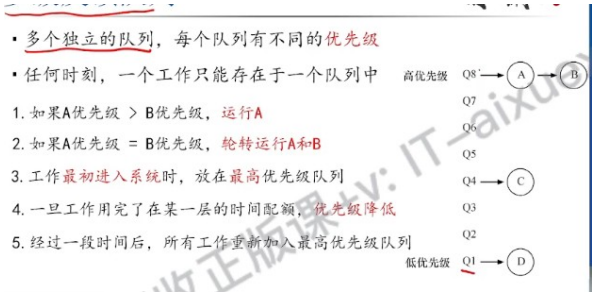
完了会主动放弃处理机，因此也会发生调度。队头进程P2上处理机。

如果每个进程都在一个时间片内完成，则轮转算法退化为 FCFS 算法
如何确定时间片的长度?
1、 系统响应时间
2、 就绪进程的数量
3、 进程调度以及上下文切换的时间开销
4、 CPU 指令的速度

多级队列调度



多级反馈队列调度



优先级从高到低，时间片从小到大

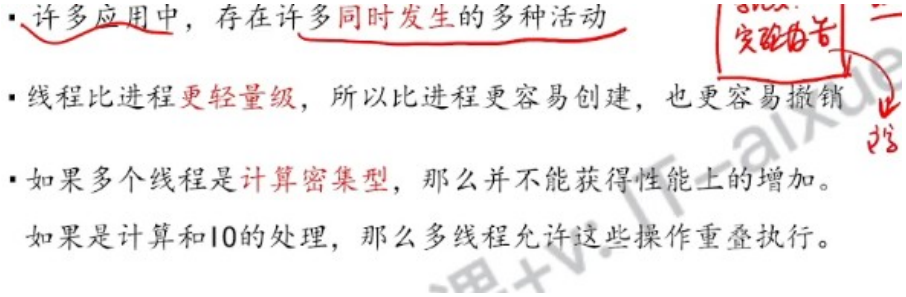
各队列之间采用抢占式优先级算法调度
当 CPU 正在运行第 i 个队列中的某个进程时，又有进程而进入优先级较高的队列，则系统立即调度高优先级的进程进行

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	

多处理器调度

除了调度算法决定让哪个进程上 CPU，还要确定上哪个 CPU
负载均衡
处理机亲和性
XPU 数据共享
缓存一致性
1、公共就绪队列

线程



轻量级：不需要保存那么多信息
因为 CPU 只有一个，多个线程还是会竞争 CPU
如果是计算和 IO 的处理，那么多线程允许这些操作的重叠执行。会提升性能。
多线程≠并行
提高系统的并发程度，同一个进程中两个线程
线程作为调度和执行到基本单位，把进程作为资源分配和拥有的基本单位

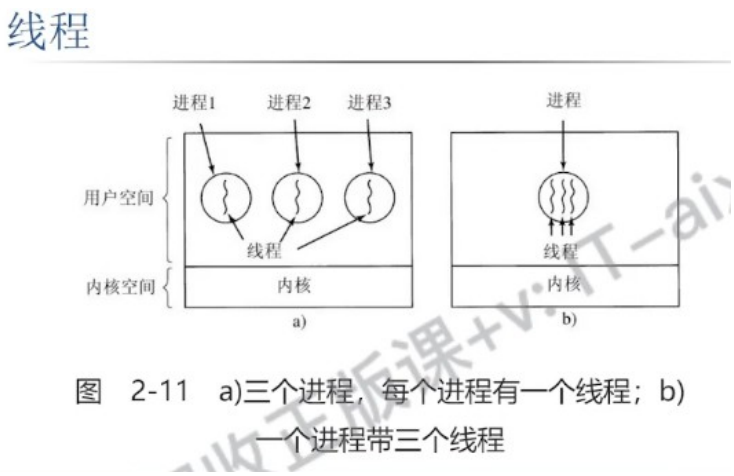
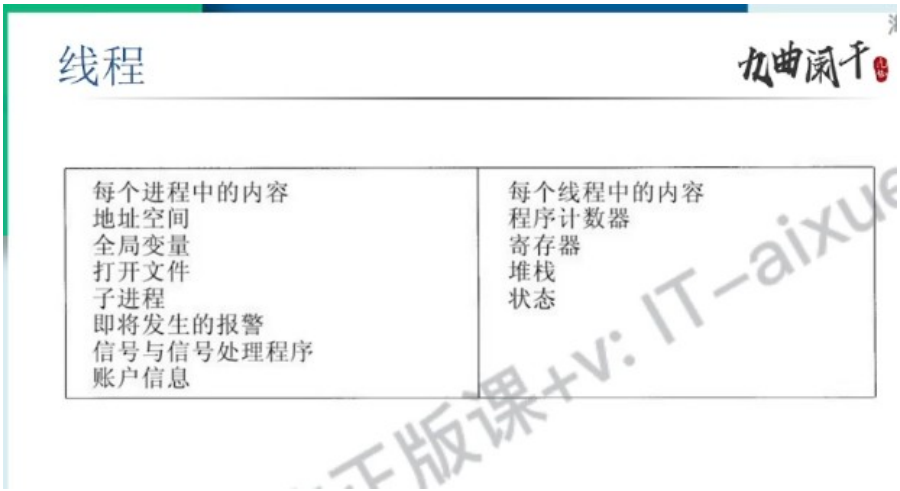


图 2-11 a)三个进程，每个进程有一个线程；b)一个进程带三个线程

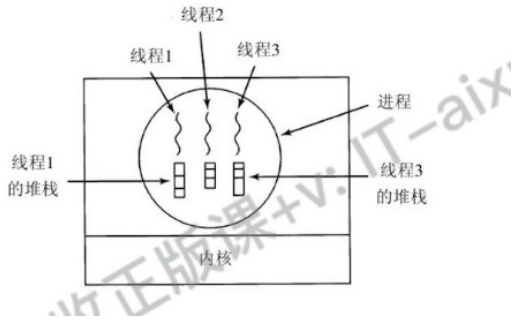


用户级线程

运行在用户态，由支撑线程一组应用程序代码完成， 该组代码称为线程库，运行在用户空间，
操作系统并不知道线程的单位，仍以进程为单位
很多编程语言提供了强大的线程库，可以实现线程的创建、销毁、调度等功能
19. 线程的引入
- 将拥有资源的实体和执行的实体分开，使执行的实体具有较少的资源，从而减少并发执行的开销，从而提高系统的并发程度。
- 拥有资源的基本单位——进程；
- 执行的基本单位（即 CPU 调度和分派的单位)——线程。
▲线程是进程内一个相对独立的运行单位，一个进程可以有一个或多个线程（至少有一个），这些线程共享这个进程的代码、数据及大部分管理信息，但每个线程有自己的程序计数器、堆栈和线程控制块。
▲但对用户级线程而言，内核进行 CPU 调度仍然以进程（而不是用户级线程）为单位。
内核级线程（KLT）
组合方式

线程表 TCB

左边是线程与进程共享的内容



线程包的实现

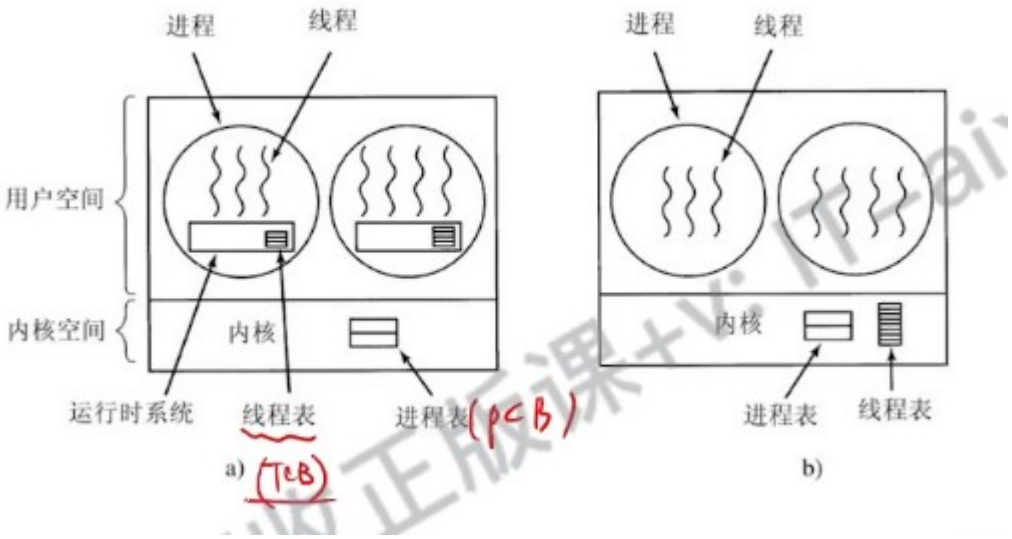
1、 用户进程管理

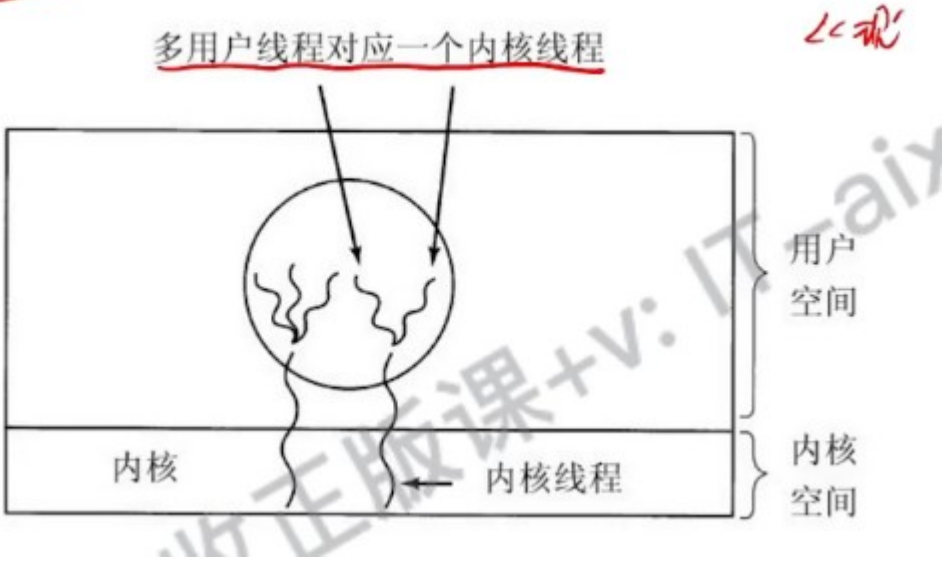
先放置在用户空间中，内核的角度按正常的方式管理，即单线程进程

2、 内核管理线程

但是内核开销较大

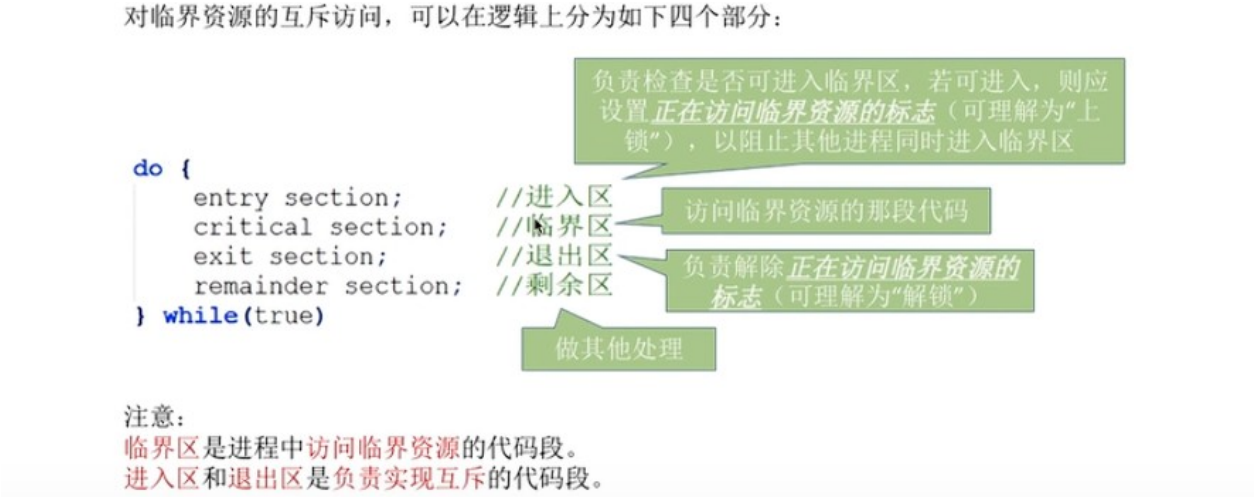
3、 混合



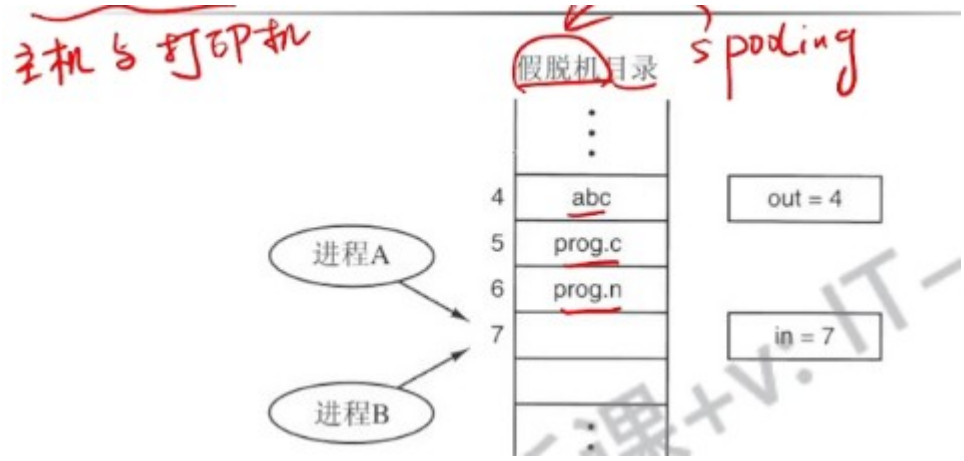


同步和异步

临界资源
系统中某些资源一次只允许一个进程使用。这类自愿成为临界资源
如物理设备打印机、软件资源共享变量、文件、表格等
必须以互斥的方式共享
每个进程中访问临界资源的那段代码叫做临界区
每个进程在进入临界区之前应该先对欲访问的临界资源进行检查，看它是否正被其他进程访问，如果临界资源未被其它进程访问，则该进程便可进入临界区访问该临界资源，并把临界源的状态设置为“忙”；
通常把这段置于临界区之前的用于检查临界资源使用状态的代码称为进入区，相应地，当进程访问完临界资源退出临界区时，将临界资源状态恢复为“空闲”。完成该项工作的代码称为“退出区”。其余无关的称为“剩余区”



互斥
互斥共享
称之为临界资源。必须以互斥的方式访问
同时共享
同步
直接制约关系，某些位置上协调，相互合作的进程按一定的先后顺序
进程的制约关系
直接制约：源于进程合作
间接制约：源于资源共享
10. 同步
为了保证进程正确的并发执行，对多个相关进程在执行的次序上进行协调的过程。
竞争条件



Out：打印机即将要打印的位置

In：打印机需要把文件送入的位置

进程 A，进程 B 几乎在同一时刻向打印机发送文件

进程 A 还没有完，时间片用完，切换进程 B，此时 in=7 是空的，覆盖了。

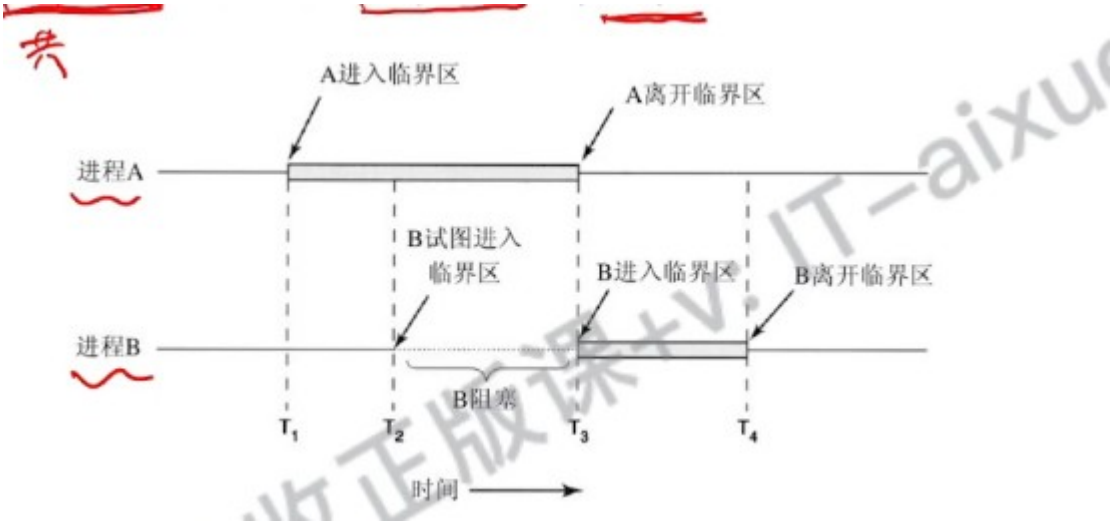
临界区：

对共享内存进行访问的程序片段称为临界区。

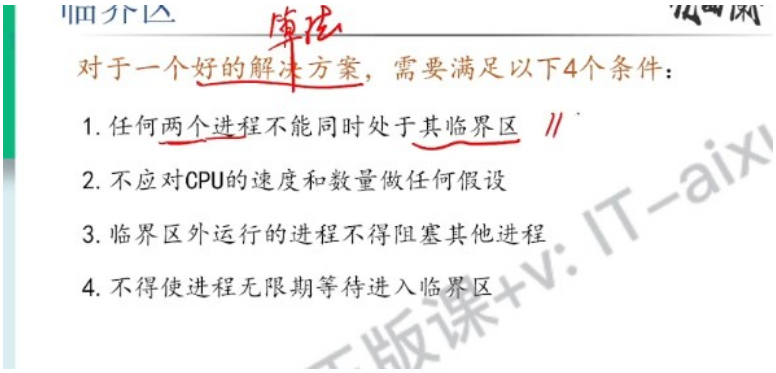
每个进程中访问临界资源的那段代码为临界区。

临界资源：系统中某些资源一次只允许一个进程使用，多个进程共享临界资源时，必须互斥方式共享。

临界区：访问临界资源的那段代码



同步机制满足原则：



1、 空闲让进，临界区空闲时，允许一个请求进入临界区的进程立即进入临界区
2、 忙则等待，当已有进程接入临界区，其它试图进入临界区的进程必须等待
3、 有限等待，对请求访问的进程，应保证能在有限时间内进入临界区（保证不会饥饿）
4、 让权等待，当进程不能进入临界区，应立即释放 CPU

Peterson 算法

```
#define FALSE 0
#define TRUE  1
#define N     2                                   /* 进程数量 */

int turn;                                         /* 现在轮到谁？ */
int interested[N];                                /* 所有值初始化为0（FALSE） */

void enter_region(int process);                   /* 进程是0或1 */
{
    int other;                                    /* 其他进程号 */

    other = 1 - process;                          /* 另一方进程*/
    interested[process] = TRUE;                   /* 表明所感兴趣的 */
    turn = process;                               /* 设置标志 */
    while (turn == process && interested[other] == TRUE); /*空语句 */
}

void leave_region(int process)                    /* 进程：谁离开？ */
{
    interested[process] = FALSE;                  /* 表示离开临界区 */
}
```

硬件方式

均存在让权等待

禁止中断

利用专用机器指令

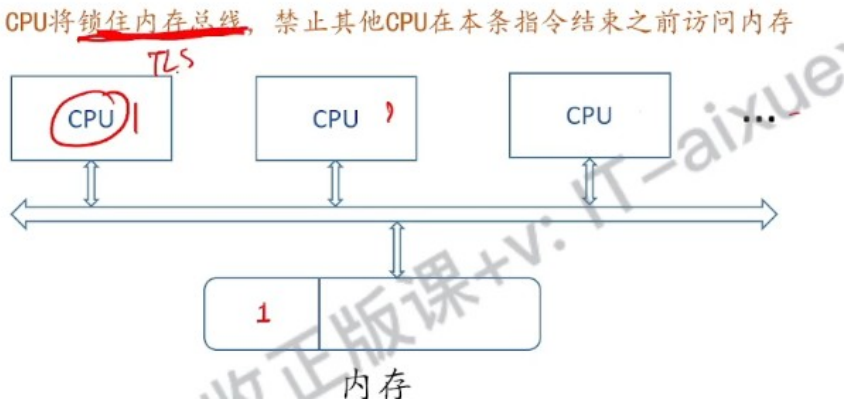
解决 Swap 指令/XCHG 指令/TSL 指令
执行的过程不允许中断，利用硬件的方式变成了原子操作

TSL 指令是用硬件实现的，执行的过程不允许被中断，只能一气呵成。以下是用C语言描述的

```
//布尔型共享变量 lock 表示当前临界区是否被加锁
//true 表示已加锁, false 表示未加锁
bool TestAndSet (bool *lock){
  bool old;
  old = *lock;  //old用来存放lock 原来的值
  *lock = true; //无论之前是否已加锁, 都将lock设为true
  return old;   //返回lock原来的值
}
```

```
//以下是使用 TSL 指令实现互斥的算法
while (TestAndSet (&lock)); //"
临界区代码段...
lock = false;       //"解锁"
剩余区代码段...
```

测试并加锁- Test and set lock



测试并加锁- Test and set lock

```
enter_region:
    TSL REGISTER,LOCK        | 复制锁到寄存器并将锁设为1
    CMP REGISTER,#0          | 锁是零吗？
    JNE enter_region         | 若不是零，说明锁已被设置，所以循环
    RET                      | 返回调用者，进入了临界区

leave_region:
    MOVE LOCK,#0             | 在锁中存入0
    RET                      | 返回调用者
```

利用软件

单标志、双标志先检查、双标志后检查
进入区的检查、上锁操作无法一气呵成——锁

Peterson 算法
有 while 循环，未遵循让权等待

信号量机制

锁：mutexlock
只有 true 和 false 两种表示锁是否可用
Aacquire 获得锁。Release 释放锁
每个互斥锁都有一个布尔变量

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 acquire()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

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

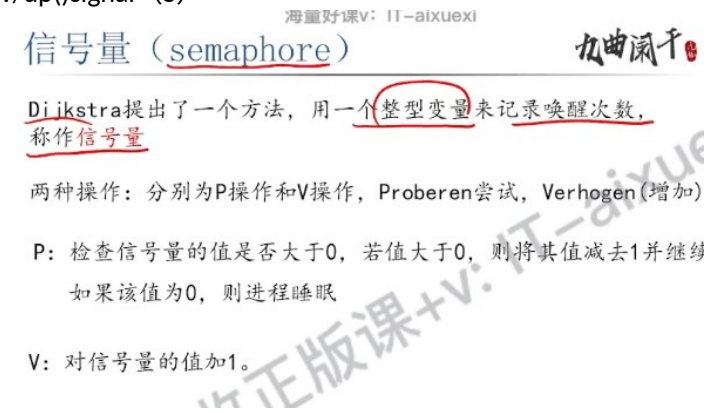
```
acquire()
    while(!available)
        ;                       //忙等待
    available = false;          //获得锁
}
```

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

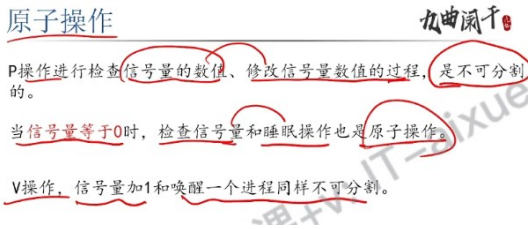
信号量

P/down()wait（S）

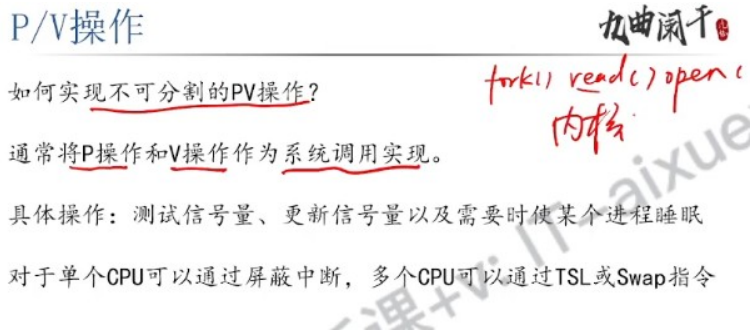
V/up()signal（S）



原子操作：不可分割的操作



原语是一种特殊的程序段，不可被中断

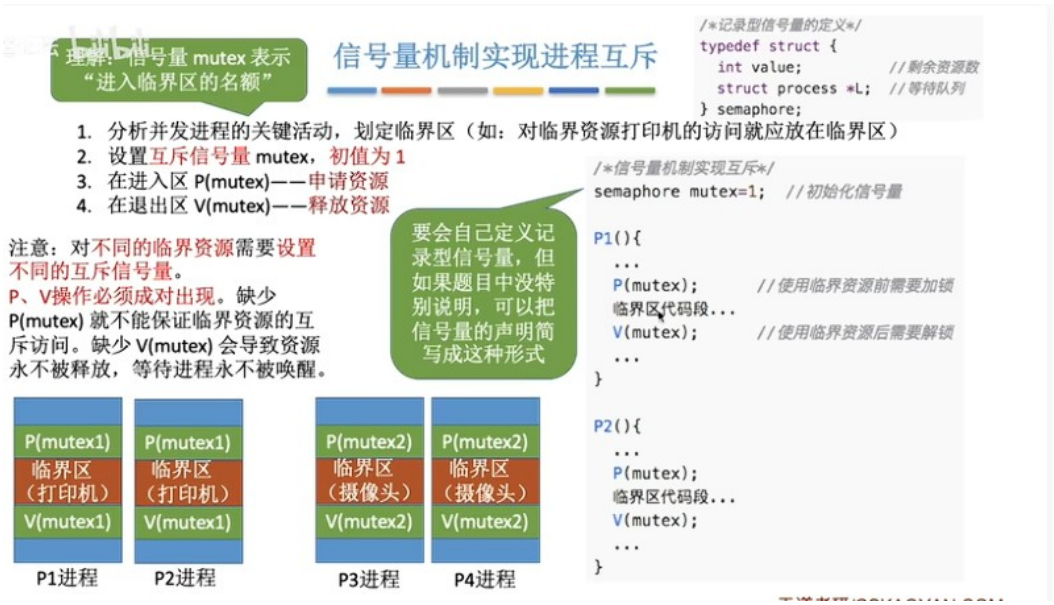


1. 分析并发进程的关键活动，划定临界区（如：对临界资源打印机的访问就应放在临界区）
2. 设置互斥信号量 mutex，初值为 1

```
/*信号量机制实现互斥*/
semaphore mutex=1;  //初始化信号量

P1(){
  ...
  P(mutex);       //使用临界资源前需要加锁
  临界区代码段...
  V(mutex);       //使用临界资源后需要解锁
  ...
}

P2(){
  ...
  P(mutex);
  临界区代码段...
  V(mutex);
  ...
}
```



信号量背后的含义，一个信号量对应一种资源
用户可以通过操作系统提供的一对原语来对信号量进行操作
信号量可以是整型、记录型、AND 型、信号量集
信号量的值=这种资源的剩余数量
P(S)/down()/wait()——申请一个资源 S，如果资源不够就阻塞等待
V(S)/up()/signal——释放一个资源 S，如果有进程在等待该资源则唤醒一个进程
一对原语 wait(S)/signal(S)

整形信号量：只有三种，初始化、P、V
一个整数型的变量作为信号量，用来表示系统中某种资源的数量

```
int S = 1;  //初始化整型信号量s，表示当前系统中可用的打印机资源数

void wait (int S) {   //wait 原语，相当于“进入区”
  while (S <= 0);     //如果资源数不够，就一直循环等待
  S=S-1;              //如果资源数够，则占用一个资源
}

void signal (int S) { //signal 原语，相当于“退出区”
  S=S+1;        //使用完资源后，在退出区释放资源
}
```

```
进程P0:
...
wait(S);          //进入区，申请资源
使用打印机资源...  //临界区，访问资源
signal(S);        //退出区，释放资源
...
```

用原语实现了，“检查”和“上锁”解决了并发、异步的问题

Wait 原语又不能被中断。难道 CPU 卡在哪里了？

记录型信号量

```
/*记录型信号量的定义*/
typedef struct {
  int value;           //剩余资源数
  struct process *L;   //等待队列
} semaphore;
```

```
/*某进程需要使用资源时，通过 wait 原语申请*/
void wait (semaphore S) {
  S.value--;
  if (S.value < 0 ) {
    block (S.L);
  }
}
```

如果剩余资源数不够，使用block原语使进程从运行态进入阻塞态，并把挂到信号量 S 的等待队列（即阻塞队列）中

```
/*进程使用完资源后，通过 signal 原语释放*/
void signal (semaphore S) {
  s.value++;
  if (S.value <= 0) {
    wakeup(S.L);
  }
}
```


wait 原语

某进程需要使用资源 S

S--表示请求分配一个资源

如果 S 值≥0，则表示可以为进程分配资源，该进程进入临界区继续执行，

如果 S 的值＜0，表示-1 之前已没有资源可供分配（S=0）

执行 block 原语，将其进程阻塞起来，插入到 S 的阻塞队列中，然后执行另一进程

Signal

S++表明释放一个资源

如果 S＞0，则表示阻塞队列为空，该进程继续执行；；

如果 S≤0，表示阻塞队列中有阻塞的进程，wakeup 原语唤醒队首进程从阻塞状态变为就绪状态

不管怎么说，先--或者++，然后判断，决定阻塞 or 释放

S>0，其值表示当前可供分配的资源数目

S<0，其绝对值表示阻塞队列中的进程数目

信号量集机制

信号量机制实现进程互斥

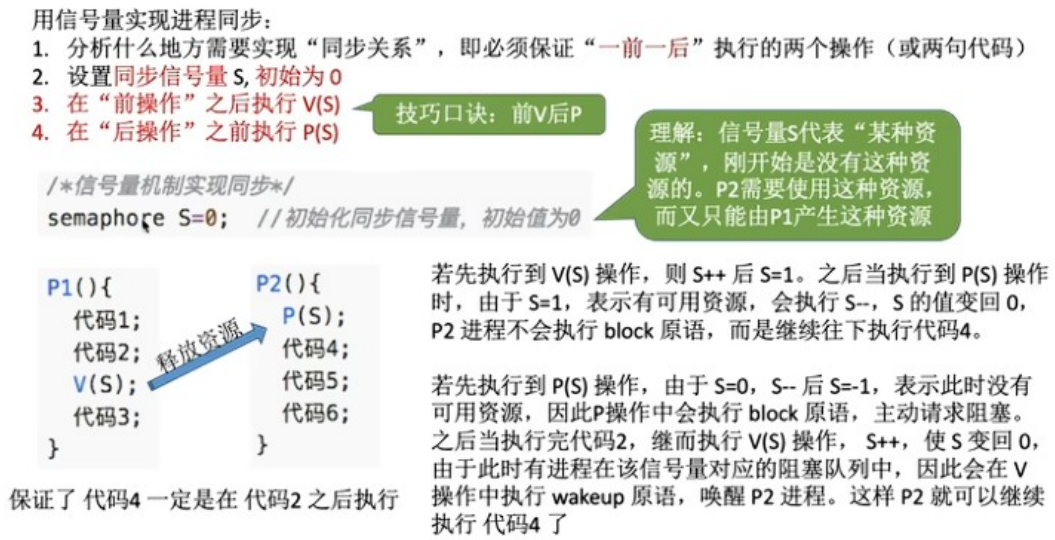
如果题目没有特别说明，对于一个信号量的定义只需要 semaphore

smaphore 这个信号量不是整型信号量而是记录型信号量，带有排队阻塞的信号量，并不会忙等
对于不同的临界资源需要设置不同的信号量
利用信号量机制可以方便地解决多个进程互斥使用临界资源的问题

生产者/操作	必须在谁之后执行	依赖的信号量
PB 读取 buf1	PA 写完 buf1	full1
PC 打印 buf2	PB 写完 buf2	full2
PA 写 buf1	上一次 PB 读取完 buf1	empty1
PB 写 buf2	上一次 PC 打印完 buf2	empty2

信号量机制实现进程同步

要让各并发进程按要求有序地推进
设置同步信号量 S，初始为 0



首先可以定义一个初值为 1 的信号量 S（互斥信号量）
当进程想要进入临界区访问临界资源时，wait（）操作申请资源
退出临界区时执行 signal 操作（）释放资源
只要把进程临界区置于 wait（）和 siganl()之间，就可以实现互斥

该方案使用了三个信号量

1. 一个称为full，用来记录充满的缓冲区的数量
2. 一个称为empty，记录空的缓冲区数目
3. 一个称为mutex，用来确保生产者和消费者不会同时访问缓冲区

```
#define N 100                          /* 缓冲区中的槽数目 */
typedef int semaphore;                 /* 信号量是一种特殊的整型数据 */
semaphore mutex = 1;                   /* 控制对临界区的访问 */
semaphore empty = N;                   /* 计数缓冲区的空槽数目 */
semaphore full = 0;                    /* 计数缓冲区的满槽数目 */
```

为什么 empty 和 full 也要设为信号量?

```
void producer(void)
{
    int item;

    while (TRUE) {                     /* TRUE是常量1 */
        item = produce_item();         /* 产生放在缓冲区中的一些数据 */
        down(&empty);                  /* 将空槽数目减1 */
        down(&mutex);                  /* 进入临界区 */
        insert_item(item);             /* 将新数据项放到缓冲区中 */
        up(&mutex);                    /* 离开临界区 */
        up(&full);                     /* 将满槽的数目加1 */
    }
}
```

```
void consumer(void)
{
    int item;

    while (TRUE) {                     /* 无限循环 */
        down(&full);                   /* 将满槽数目减1 */
        down(&mutex);                  /* 进入临界区 */
        item = remove_item();          /* 从缓冲区中取出数据项 */
        up(&mutex);                    /* 离开临界区 */
        up(&empty);                    /* 将空槽数目加1 */
        consume_item(item);            /* 处理数据项 */
    }
}
```

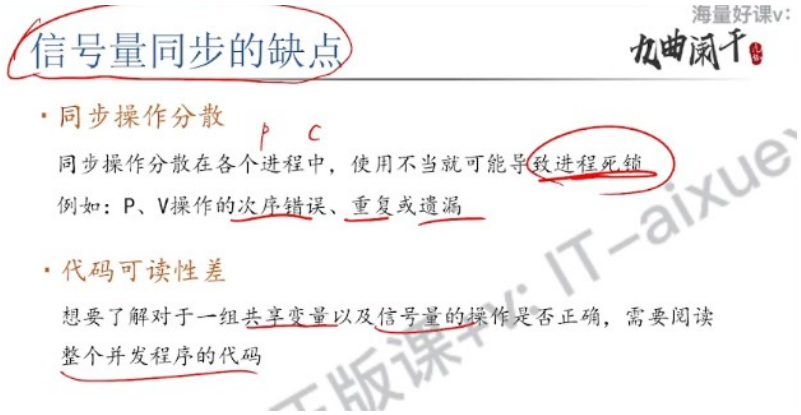
不能相反

会导致死锁

Mutex=0；

Producer 会被阻塞；

Consumer 发现 mutex=0，也会被阻塞。



经典进程同步问题

生产者消费者问题

M 个生产者
N 个消费者
共享一个具有 k 个缓冲区的循环缓冲池
生产者不断生产产品，将每个产品依次放入缓冲区中（一个缓冲区正好放一个产品）
消费者依次从缓冲区取出产品并进行消费。规定消费者不能从一个空缓冲区中取产品，生产者不能向一个已装满产品且尚未被取走的缓冲区中投放产品。
什么时候用数字 out in=0，1
什么时候用信号量 mutex ，Empty，full?

```
#define N 100                                   /* 缓冲区中的槽数目 */
int count = 0;                                  /* 缓冲区中的数据项数目 */

void producer(void)
{
     int item;

     while (TRUE) {                             /* 无限循环 */
          item = produce_item();                /* 产生下一新数据项 */
          if (count == N) sleep();              /* 如果缓冲区满了，就进入休眠状态 */
          insert_item(item);                    /* 将（新）数据项放入缓冲区中 */
          count = count + 1;                    /* 将缓冲区的数据项计数器增1 */
          if (count == 1) wakeup(consumer);     /* 缓冲区空吗？ */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* 无限循环 */
          if (count == 0) sleep();              /* 如果缓冲区空，则进入休眠状态 */
          item = remove_item();                 /* 从缓冲区中取出一个数据项 */
          count = count - 1;                    /* 将缓冲区的数据项计数器减1 */
          if (count == N - 1) wakeup(producer); /* 缓冲区满吗？ */
          consume_item(item);                   /* 打印数据项 */
     }
}
```

1、 count == 1 时唤醒消费者，保证至少有一个元素可消费；

2、 count == N-1 时唤醒生产者，保证有一个空位可写；

当前缓冲区已空

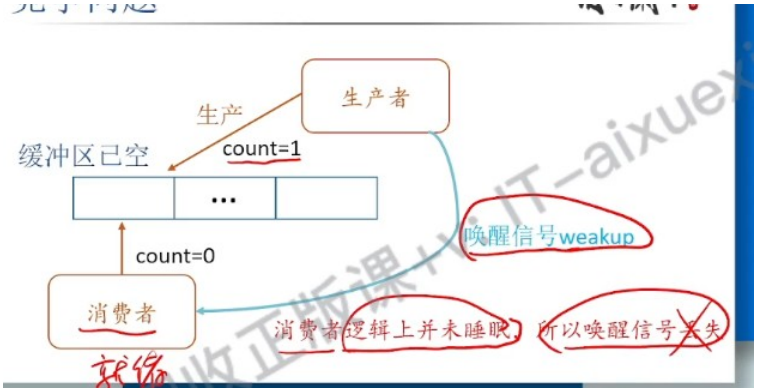
消费者去执行的时候发现 count==0

但是调度程序切换到了生产者，发现 count=1

生产者向消费者发送 wakeup

消费者还没来得及 sleep 却 wakeup

丢失唤醒（lost wakeup）问题



最后消费者 sleep

生产者也 sleep 了

可以设置等待位

但是麻烦

读者-写者问题

多个进程同时读数据库是可以的
当一个进程正在写数据库，其它进程既不能读也不能写
任意时刻只能由一个写数据库

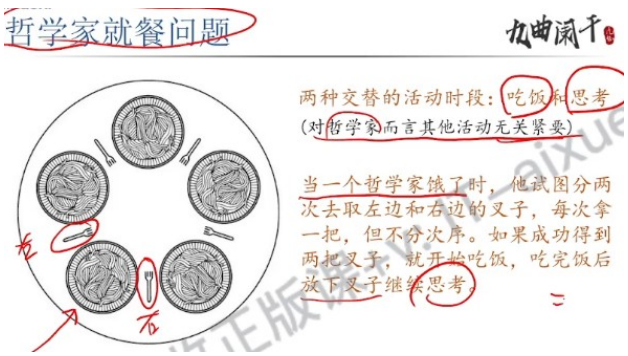
```
semaphore mutex = 1;                /* 控制对rc的访问 */
semaphore db = 1;                   /* 控制对数据库的访问 */
int rc = 0;                         /* 正在读或者即将读的进程数目 */

void writer(void)
{
    while (TRUE) {                  /* 无限循环 */
        think_up_data();            /* 非临界区 */
        down(&db);                  /* 获取互斥访问 */
        write_data_base();          /* 更新数据 */
        up(&db);                    /* 释放互斥访问 */
    }
}
```

```
void reader(void)
{
    while (TRUE) {                  /* 无限循环 */
        down(&mutex);               /* 获得对rc的互斥访问权 */
        rc = rc + 1;                /* 现在又多了一个读者 */
        if (rc == 1) down(&db);     /* 如果这是第一个读者…… */
        up(&mutex);                 /* 释放对rc的互斥访问 */
        read_data_base();           /* 访问数据 */
        down(&mutex);               /* 获取对rc的互斥访问 */
        rc = rc - 1;                /* 现在减少了一个读者 */
        if (rc == 0) up(&db);       /* 如果这是最后一个读者…… */
        up(&mutex);                 /* 释放对rc的互斥访问 */
        use_data_read();            /* 非临界区 */
    }
}
```

读者优先——有写者想写则必须等待

哲学家就餐问题



```
#define N 5                                     /* 哲学家的数目 */

void philosopher(int i)                         /* i：哲学家编号，从0到4 */
{
    while (TRUE) {
        think( );                               /* 哲学家在思考 */
        take_fork(i);                           /* 拿起左边叉子 */
        take_fork((i+1) % N);                   /* 拿起右边叉子，%是模运算 */
        eat( );                                 /* 进食 */
        put_fork(i);                            /* 将左叉放回桌上 */
        put_fork((i+1) % N);                    /* 将右叉放回桌上 */
    }
}
```

同时成功取到左侧叉子，再取右侧筷子失效。死锁

如果五位哲学家同时拿起了左边的叉子，那么就没有人能够拿到他们右边的叉子，于是发生死锁

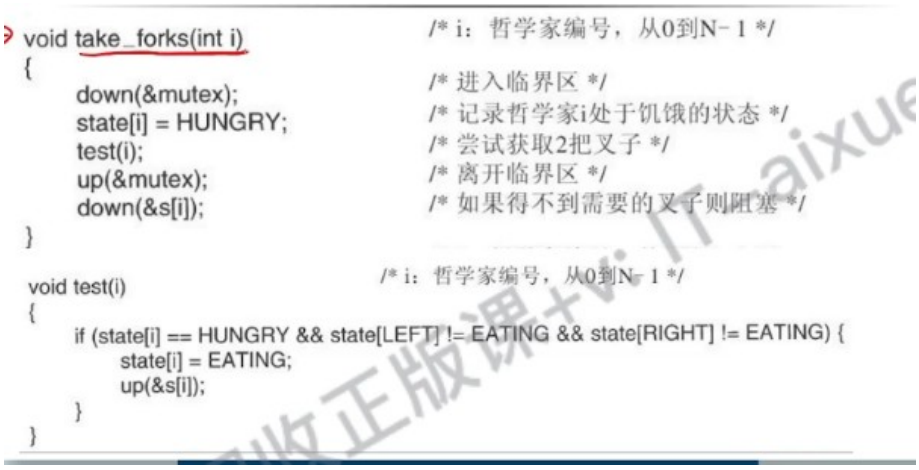
修改上述解决方法：拿到左叉子后，检查右叉子是否可用，如果不可用，放下左叉子，等待一段时间后，重复这个过程。

可能在某个瞬间，所有哲学家同时拿起了左叉子，然后检查右叉子不可用，又放下左叉子。如此永远重复下去。
所有程序都在不停的运行，但是都无法取得进展，称为饥饿

可能想到的解决方法：随机等待一段时候，再拿右边的叉子

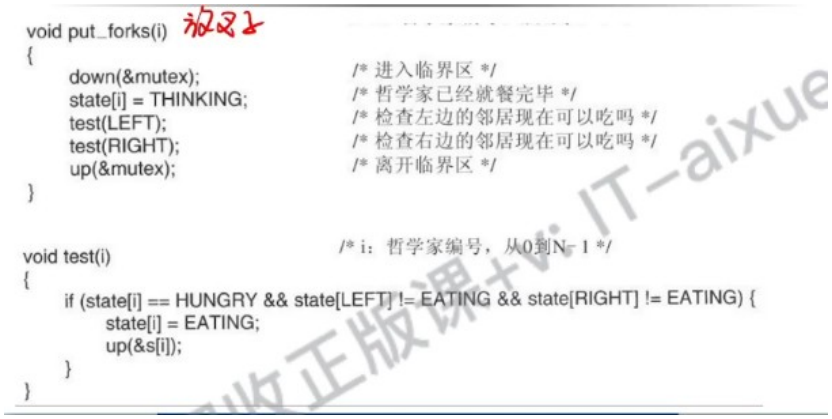
哲学家就餐问题

```
#define N           5                   /* 哲学家数目 */
#define LEFT        (i+N-1)%N           /* i 的左邻居编号 */
#define RIGHT       (i+1)%N             /* i 的右邻居编号 */
#define THINKING    0                   /* 哲学家在思考 */
#define HUNGRY      1                   /* 哲学家试图拿起叉子 */
#define EATING      2                   /* 哲学家进餐 */
typedef int semaphore;                  /* 信号量是一种特殊的整型数据 */
int state[N];                           /* 数组用来跟踪记录每位哲学家的状态 */
semaphore mutex = 1;                    /* 临界区的互斥 */
semaphore s[N];                         /* 每个哲学家一个信号量 */
```



```
void take_forks(int i)          /* i：哲学家编号，从0到N－1 */
{
    down(&mutex);               /* 进入临界区 */
    state[i] = HUNGRY;          /* 记录哲学家i处于饥饿的状态 */
    test(i);                    /* 尝试获取2把叉子 */
    up(&mutex);                 /* 离开临界区 */
    down(&s[i]);                /* 如果得不到需要的叉子则阻塞 */
}

void test(i)                    /* i：哲学家编号，从0到N－1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```



```
void put_forks(i)
{
    down(&mutex);               /* 进入临界区 */
    state[i] = THINKING;        /* 哲学家已经就餐完毕 */
    test(LEFT);                 /* 检查左边的邻居现在可以吃吗 */
    test(RIGHT);                /* 检查右边的邻居现在可以吃吗 */
    up(&mutex);                 /* 离开临界区 */
}

void test(i)                    /* i：哲学家编号，从0到N－1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

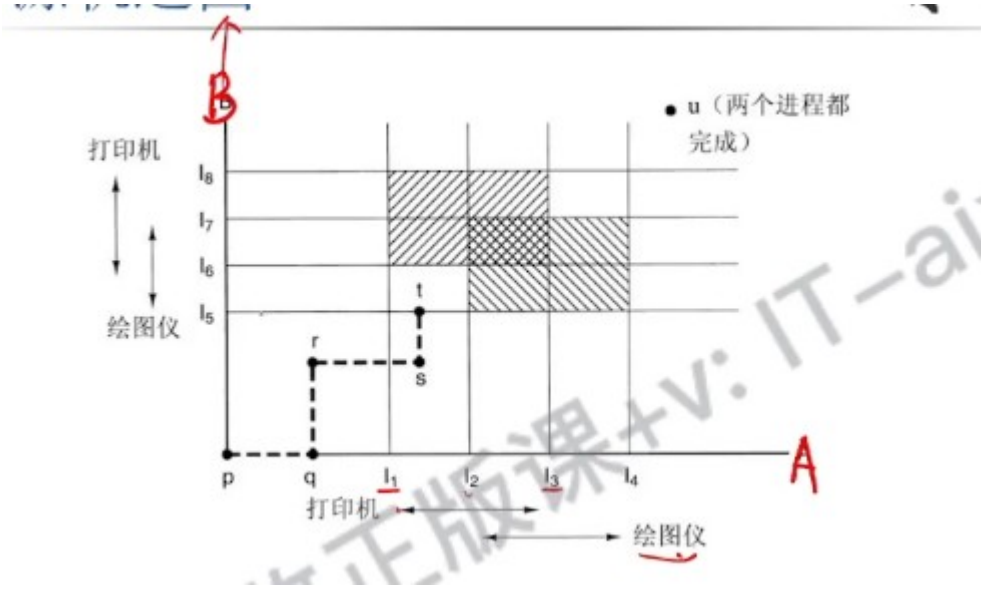
1/

死锁

若系统中存在一组进程（两个或两个以上），且它们都无限等待该组进程中另一进程所占有的无法释放的资源，无法向前推进

1、 竞争资源

2、 进程推进顺序不当

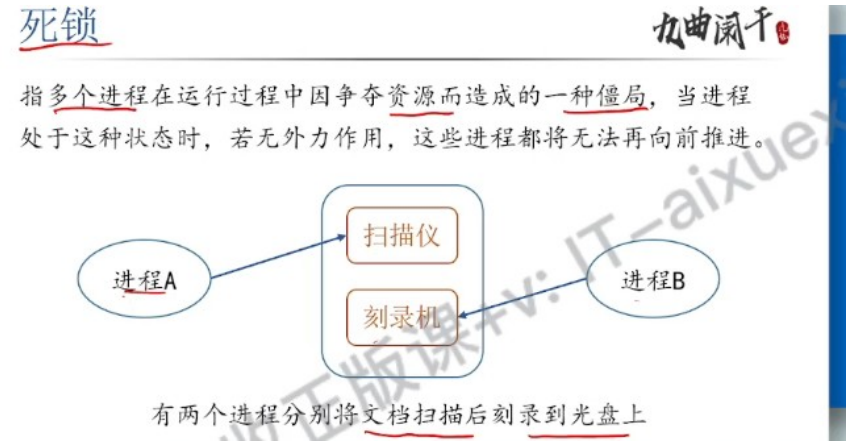


3、 信号量使用不当（顺序不当，其实可以把互斥信号量、同步信号量也看成一种竞争资源）

饥饿：长进程一直得不到处理机

死循环：跳不出循环

	共同点	区别
死锁	都是进程无法顺利向前推进的现象（故意设计的死循环除外）	死锁一定是“循环等待对方手里的资源”导致的，因此如果有死锁现象，那至少有两个或两个以上的进程同时发生死锁。另外，发生死锁的进程一定处于阻塞态。
饥饿		可能只有一个进程发生饥饿。发生饥饿的进程既可能是阻塞态(如长期得不到需要的I/O设备)，也可能是就绪态(长期得不到处理机)
死循环		可能只有一个进程发生死循环。死循环的进程可以上处理机运行（可以是运行态），只不过无法像期待的那样顺利推进。死锁和饥饿问题是由于操作系统分配资源的策略不合理导致的，而死循环是由代码逻辑的错误导致的。死锁和饥饿是管理者（操作系统）的问题，死循环是被管理者的问题。



我们把这一类需要排他性使用的对象称为资源

资源可以是硬件设备(打印机等)或者是一组信息(数据库中加锁记录)

- 可抢占资源

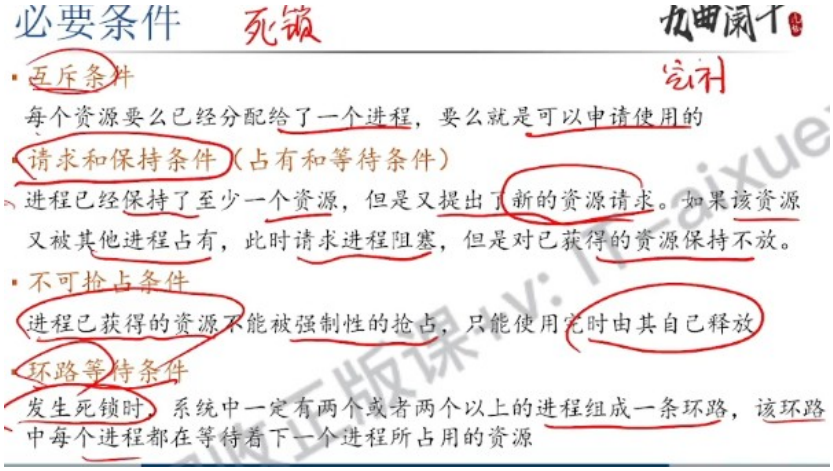
  可以从拥有它的进程中抢断而不会产生副作用，例如内存

- 不可抢占资源

  无法从拥有它的进程处抢夺过来，例如光盘刻录机

- 产生死锁的必要条件：互斥条件、占有且等待条件、不可剥夺条件、循环等待条件。
  必要条件：这四个条件必须**同时存在**才会导致死锁。

1. **充分条件（A → B）**：
   - A 是 B 的“充分”条件，即 **A 成立足以保证 B 成立**，但 B 可能由其他原因导致。
2. **必要条件（B ← A）**：
   - B 是 A 的“必要”条件，即 **A 成立必须依赖 B 成立**（或无 B 必无 A），但 B 成立不一定导致 A。
3. **关键区别**：
   - 充分条件关注“A 能否保证 B”，必要条件关注“B 是否是 A 的前提”。



死锁避免

安全状态

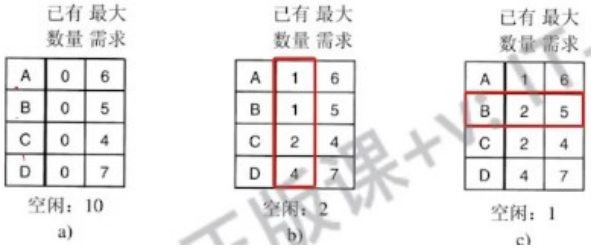
安全序列，就是指如果系统按照
安全序列可能有多个
找不出任何一个安全序列，就进入了不安全状态
进入不安全状态，就可能发生死锁

不安全状态并不是死锁

银行家算法

判断对请求的满足是否会导致进入不安全状态

如果会导致不安全状态，就拒绝该请求，否则就满足该请求



安全序列

	最大需求	已借走	最多还会借
B	70	20+30=60	50-30=20
A	40	10	30
T	50	30	20

给B借30亿是不安全的...之后手里只剩10亿，如果BAT都提出再借20亿的请求，那么任何一个企业的需求都得不到满足...

	最大需求	已借走	最多还会借
B	70	20	50
A	40	10+20=30	30-20=10
T	50	30	20

给A借 20 亿是安全的，因为存在 T→B→A 这样的*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。

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，如果有进程提前归还了一些资源，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。

如果导致不安全状态，就拒绝该请求，否则就满足该请求

死锁避免从本质上说是不可能的，因为很少由进程在运行之前就知道运行所需的最大资源数，进程数也是不固定的

进程	最大需求	已分配	最多还需要
P0	(8, 5, 3)	(0, 1, 0)	(8, 4, 3)
P2	(9, 5 ,2)	(3, 0, 2)	(6, 5, 0)
P4	(4, 3, 6)	(0, 0, 2)	(4, 3, 4)

资源总数 (10, 5, 7)，剩余可用资源 (7, 4, 3)

再看一个找不到安全序列的例子：
经对比发现，（3, 3, 2）可满足 P1、P3，说明无论如何，这两个进程的资源需求一定是可以依次被满足的，因此P1、P3 一定可以顺利的执行完，并归还资源。 可把 P1、P3 先加入安全序列。
(2, 0, 0) + (2, 1, 1) + (3, 3, 2) = (7, 4, 3)

银行家算法

Available = (1, 2, 1)　　$Request_0$ = (2, 1, 1)

假设系统中有 n 个进程，m 种资源
每个进程在运行前先声明对各种资源的最大需求数，则可用一个 n*m 的矩阵（可用二维数组实现）表示所有进程对各种资源的最大需求数。不妨称为**最大需求矩阵 Max**，Max[i, j]=K 表示进程 Pi 最多需要 K 个资源 Rj。同理，系统可以用一个 n*m 的**分配矩阵 Allocation** 表示对所有进程的资源分配情况。Max – Allocation = **Need 矩阵**，表示各进程最多还需要多少各类资源。
另外，还要用一个**长度为m的一维数组 Available** 表示当前系统中还有多少可用资源。
某进程Pi向系统申请资源，可用一个长度为m的一维数组 $Request_i$ 表示本次申请的各种资源量。

进程	最大需求	已分配	最多还需要
P0	(7, 5, 3)	(2, 2, 1)	(5, 3, 2)
P1	(3, 2, 2)	(2, 0, 0)	(1, 2, 2)
P2	(9, 0 ,2)	(3, 0, 2)	(6, 0, 0)
P3	(2, 2, 2)	(2, 1, 1)	(0, 1, 1)
P4	(4, 3, 3)	(0, 0, 2)	(4, 3, 1)

Max 矩阵　　Allocation 矩阵　　Need 矩阵

可用银行家算法预判本次分配是否会导致系统进入不安全状态：
①如果 $Request_i[j] \le Need[i, j]$ $(0 \le j \le m)$便转向②；否则认为出错。（因为它所需要的资源数已超过它所宣布的最大值。）
②如果 $Request_i[j] \le Available[j]$ $(0 \le j \le m)$，便转向③；否则表示尚无足够资源，Pi必须等待。
③系统试探着把资源分配给进程Pi，并修改相应的数据（并非真的分配，修改数值只是为了做预判）：

$Available = Available - Request_i$;
$Allocation[i, j] = Allocation[i, j] + Request_i[j]$;
$Need[i, j] = Need[i, j] – Request_i[j]$

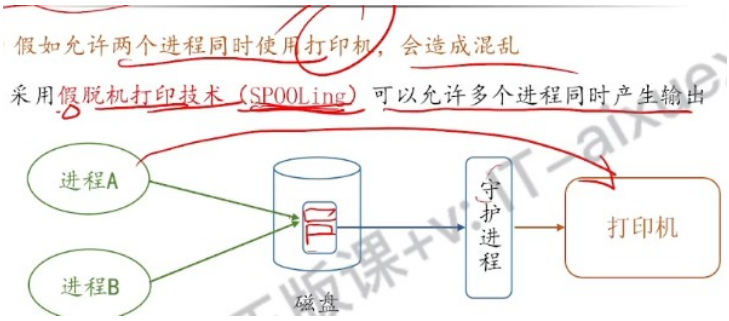
④操作系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。若安全，才正式分配；否则，恢复相应数据，让进程阻塞等待。

数据结构：
长度为 m 的一维数组 Available 表示还有多少可用资源
n*m 矩阵 Max 表示各进程对资源的最大需求数
n*m 矩阵 Allocation 表示已经给各进程分配了多少资源
Max – Allocation = Need 矩阵表示各进程最多还需要多少资源
用长度为 m 的一位数组 Request 表示进程此次申请的各种资源数

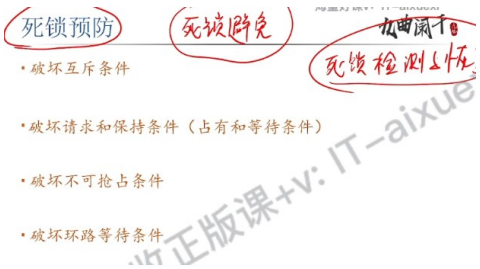
银行家算法步骤：
①检查此次申请是否超过了之前声明的最大需求数
②检查此时系统剩余的可用资源是否还能满足这次请求
③试探着分配，更改各数据结构
④用安全性算法检查此次分配是否会导致系统进入不安全状态

安全性算法步骤：
检查当前的剩余可用资源是否能满足某个进程的最大需求，如果可以，就把该进程加入安全序列，并把该进程持有的资源全部回收。

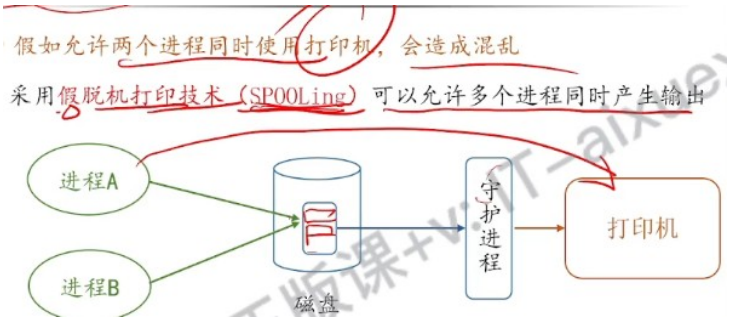
鸵鸟算法


假如允许两个进程同时使用打印机，会造成混乱
采用假脱机打印技术（SPOOLing）可以允许多个进程同时产生输出
进程A
进程B
磁盘
守护进程
打印机

预防死锁



破坏互斥条件



操作系统采用 SPOOLing 技术把独占设备在逻辑上改造成共享设备
在各进程看来，自己对打印机资源的使用请求立即被接受处理了，不需要再阻塞等待
在磁盘上开一个缓冲区实现共享

破坏请求和保持条件

- 规定所有进程在开始执行前请求所需的全部资源

  如果所需的全部资源可用，那么进程肯定可以运行结束

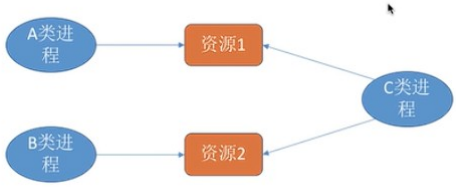
  如果有一个或者多个资源正被使用，那么就不进行分配，进程等待

- 当一个进程请求资源时，先暂停释放其当前占用的所有资源
  然后再尝试一次获得所需的全部资源

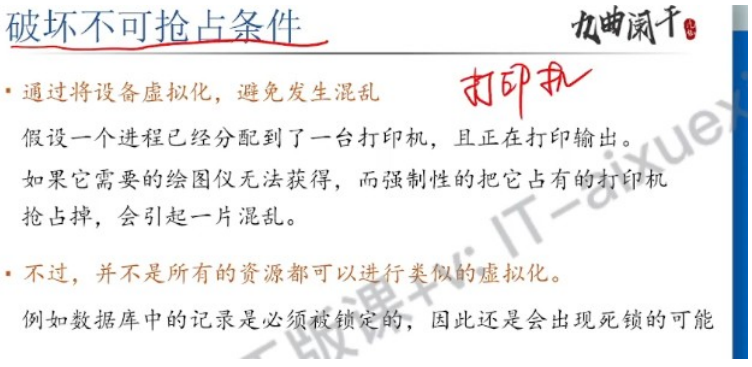
请求和保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源又被其他进程占有，此时请求进程被阻塞，但又对自己已有的资源保持不放。

可以采用静态分配方法，即进程在运行前一次申请完它所需要的全部资源，在它的资源未满足前，不让它投入运行。一旦投入运行后，这些资源就一直归它所有，该进程就不会再请求别的任何资源了。

该策略实现起来简单，但也有明显的缺点：
有些资源可能只需要用很短的时间，因此如果进程的整个运行期间都一直保持着所有资源，就会造成严重的资源浪费，资源利用率极低。另外，该策略也有可能导致某些进程饥饿。



破坏不剥夺条件



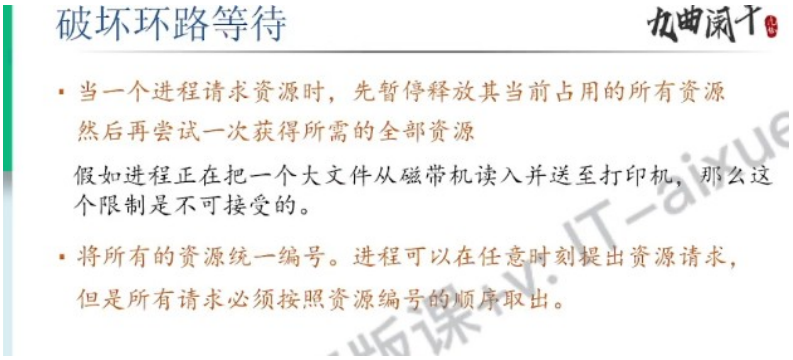
不剥夺条件：进程所获得的资源在未使用完之前，不能由其他进程强行夺走，只能主动释放。

破坏不剥夺条件：
方案一：当某个进程请求新的资源得不到满足时，它必须立即释放保持的所有资源，待以后需要时再重新申请。也就是说，即使某些资源尚未使用完，也需要主动释放，从而破坏了不可剥夺条件。

方案二：当某个进程需要的资源被其他进程所占有的时候，可以由操作系统协助，将想要的资源强行剥夺。这种方式一般需要考虑各进程的优先级（比如：剥夺调度方式，就是将处理机资源强行剥夺给优先级更高的进程使用）

该策略的缺点：
1. 实现起来比较复杂。
2. 释放已获得的资源可能造成前一阶段工作的失效。因此这种方法一般只适用于易保存和恢复状态的资源，如CPU。
3. 反复地申请和释放资源会增加系统开销，降低系统吞吐量。
4. 若采用方案一，意味着只要暂时得不到某个资源，之前获得的那些资源就都需要放弃，以后再重新申请。如果一直发生这样的情况，就会导致进程饥饿。

破坏环路等待



循环等待条件：存在一种进程资源的循环等待链，链中的每一个进程已获得的资源同时被下一个进程所请求。

可采用顺序资源分配法。首先给系统中的资源编号，规定每个进程必须按编号递增的顺序请求资源，同类资源（即编号相同的资源）一次申请完。

原理分析：一个进程只有已占有小编号的资源时，才有资格申请更大编号的资源。按此规则，已持有大编号资源的进程不可能逆向地回来申请小编号的资源，从而就不会产生循环等待的现象。

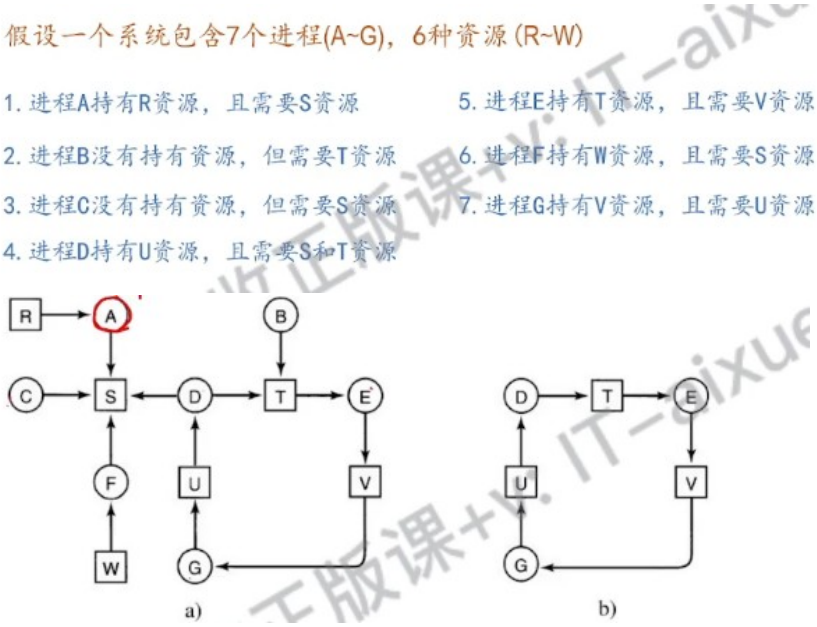
假设系统中共有10个资源，编号为 1, 2, ... 10

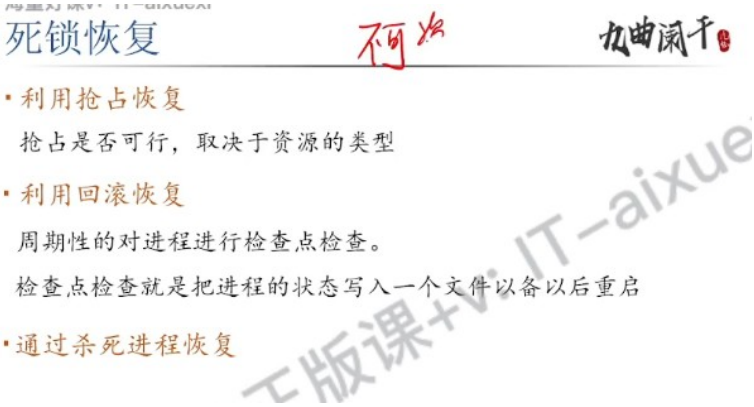
该策略的缺点：
1. 不方便增加新的设备，因为可能需要重新分配所有的编号；
2. 进程实际使用资源的顺序可能和编号递增顺序不一致，会导致资源浪费；
3. 必须按规定次序申请资源，用户编程麻烦。

死锁的检测和恢复

允许死锁发生，当检测到死锁发生后，采取措施进行恢复

资源分配图





1、 撤销进程
2、 抢占资源
    1、 利用抢占恢复
    2、 利用回滚恢复

死锁的数学公式

1. 死锁条件公式推导

系统参数：

N：进程数

W：每个进程最大资源需求

M：系统资源总数

最小安全资源数：$M \geq N \times (W-1)+1$

逻辑证明：

每个进程获得$(W-1)$个资源（共占用 $N \times (W-1)$个）

系统至少剩 1 个资源

该剩余资源可分配给任一进程使其完成

完成进程释放 W 个资源，保障其他进程执行

管程

和之前的 PV 操作一样，实现进程间的互斥和同步

1、 局部于管程的共享数据结构
2、 对于该数据结构进行操作的一组过程
3、 对于局部于共享数据设置初始值的语句
4、 管程的名字

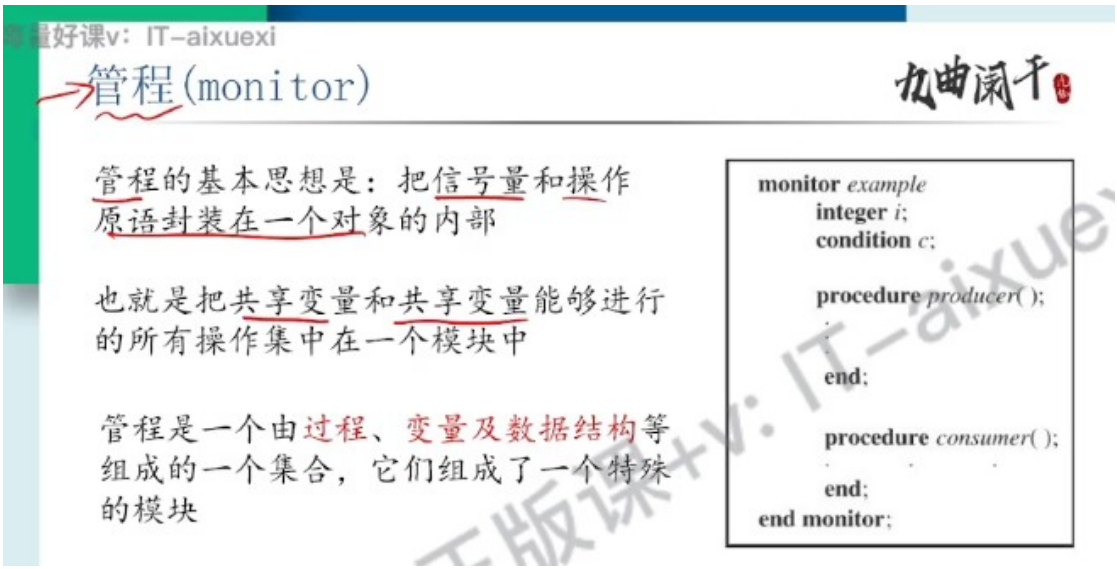
基本特征：

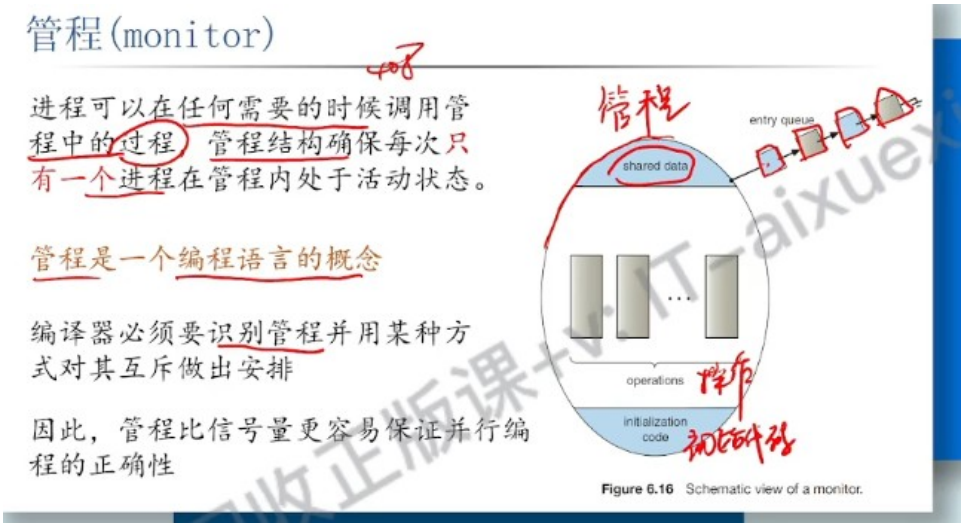
1、 局部于管程的数据结构只能被管城内的过程访问，任何外部过程不能访问，而管程内的过程也只能访问该管程内的数据结构
2、 一个进程若想访问管程内的数据结构（共享资源）们只能通过调用管程内地某个过程实现间接访问
3、 任意时刻，管程中只能有一个进程在管程中执行管程的某个过程，其它任何调用管程的进程都将被阻塞，直到管程变成可用，这一特性使管程能有效地实现互斥

引入管程的目的无非就是要更方便地实现进程互斥和同步。

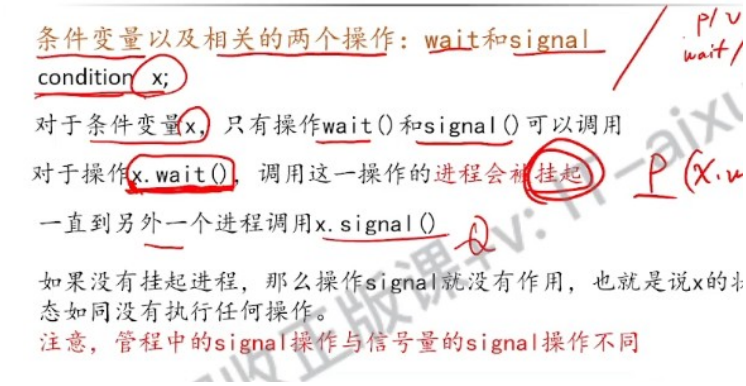
1. 需要在管程中定义共享数据（如生产者消费者问题的缓冲区）
2. 需要在管程中定义用于访问这些共享数据的“入口”——其实就是一些函数（如生产者消费者问题中，可以定义一个函数用于将产品放入缓冲区，再定义一个函数用于从缓冲区取出产品）
3. 只有通过这些特定的“入口”才能访问共享数据
4. 管程中有很多“入口”，但是每次只能开放其中一个“入口”，并且只能让一个进程或线程进入（如生产者消费者问题中，各进程需要互斥地访问共享缓冲区。管程的这种特性即可保证一个时间段内最多只会有一个进程在访问缓冲区。注意：这种***互斥特性是由编译器负责实现***的，程序员不用关心）
5. 可在管程中设置条件变量及等待/唤醒操作以解决同步问题。可以让一个进程或线程在条件变量上等待（此时，该进程应先释放管程的使用权，也就是让出“入口”）；可以通过唤醒操作将等待在条件变量上的进程或线程唤醒。

程序员可以用某种特殊的语法定义一个管程（比如: monitor ProducerConsumer ..... end monitor;），之后其他程序员就可以使用这个管程提供的特定“入口”很方便地使用实现进程同步/互斥了。





管程里的 wait 和 signal 和之前的 wait 和 signal 并不一样



内存管理

内存的分配与回收
地址映射
编译器在对程序进行编译的时候，通常从 0 开始为程序代码编址，程序中设计的所有地址都是相对起始地址 0 确定的，这种地址称为虚地址、相对地址或逻辑地址。相应地，这些地址构成的地址空间称为虚地址空间、程序空间或者逻辑地址空间。
当程序加载到内存中时，通常不是从 0 开始的内存空间，程序在物理内存中的空间称为实地址、绝对地址或者物理地址，构成的地址空间称为是地址空间、内存空间或物理空间。虚地址空间可能是一维的连续空间，也可能是二维的非线性空间，这是存储器管理方式所决定的。而实地址空间总是一维线性的。
程序运行过程中使用的地址都是虚地址，而程序加载到物理内存的实际地址往往与虚地址不同，因此虚地址不能直接用于访存。
这个从虚地址映射到实际物理地址的地址转换功能称为地址映射，又称为地址重定位。
这个过程应该由操作系统负责，这样程序员只需要关注指令、数据的逻辑地址
程序运行过程中
内存的共享和保护

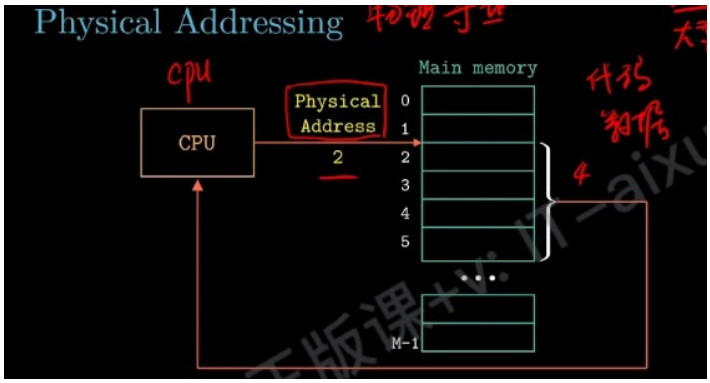
内存扩充

32 位机器的 4GB 限制主要指虚拟地址空间，这是程序能直接“看到”的最大范围。

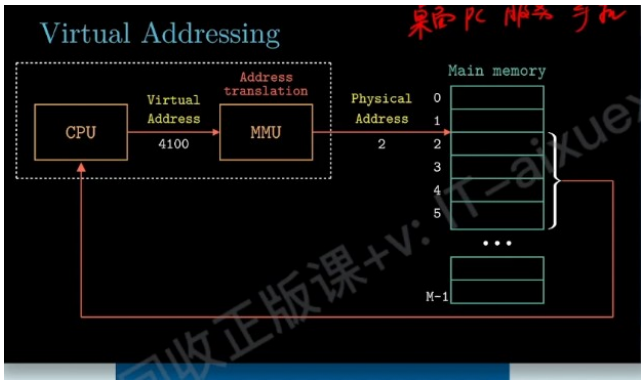
什么是内存

内存中保存代码的指令和相关数据

根据物理地址



基于虚拟寻址



CPU 发出的地址是逻辑/虚拟地址

但对于主存希望得到访存的地址依然是物理地址

将虚拟地址转换成物理地址，翻译的部件叫做 MMU 内存管理单元

（虚拟）地址空间

{0，1，2....}

虚拟地址空间最大值，取决于虚拟地址的位数

虚拟地址的位数取决于什么呢?

处理器设计时定义了地址总线的位数（或虚拟地址的位数）。

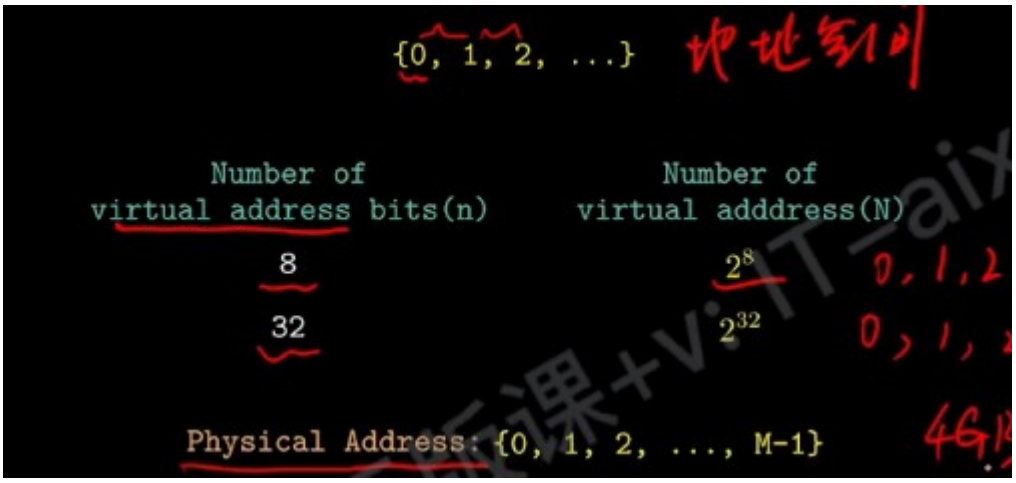
32 位机器通常指处理器具有 32 位的通用寄存器、数据总线和地址总线。

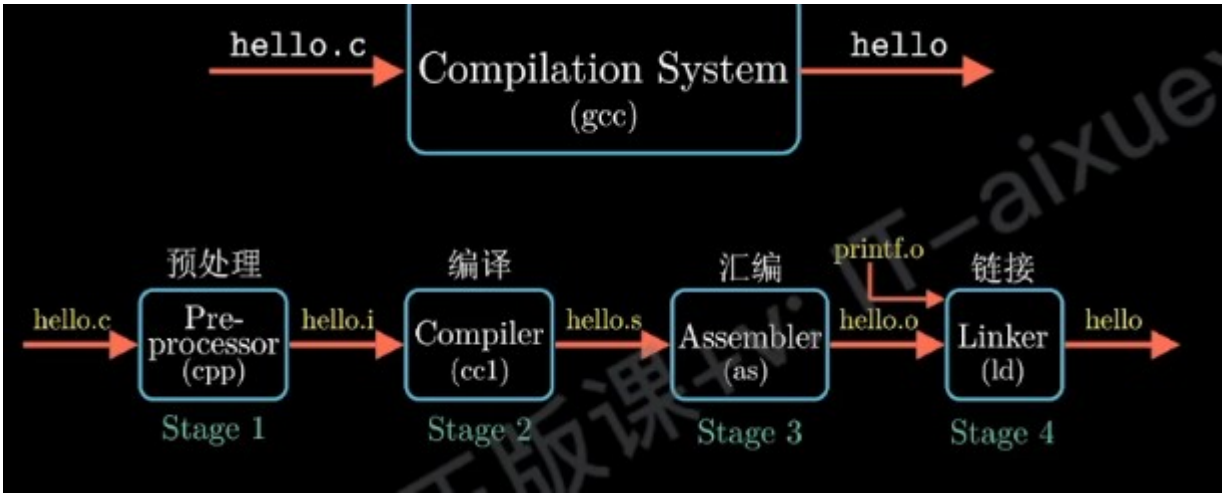
32 位机器（处理器），最大地址 2^32-1

物理地址空间大小 取决于内存的大小

4G 内存条——就有 4G 的内存空间

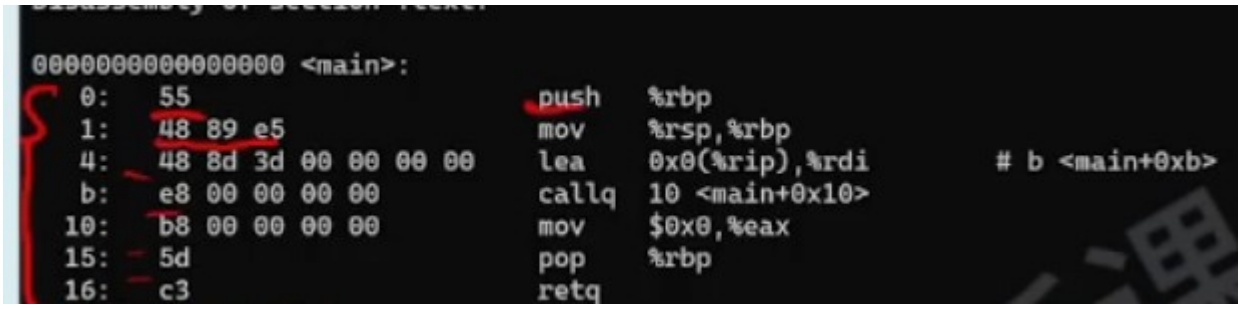
但绝不是操作系统使用的就是 4GB（并不是最大支持 4GB）





在编译的过程中会有虚拟地址的概念

这是 hello.o 的反汇编，并没有赋予地址的概念



```
0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # b <main+0xb>
   b:   e8 00 00 00 00          callq  10 <main+0x10>
  10:   b8 00 00 00 00          mov    $0x0,%eax
  15:   5d                      pop    %rbp
  16:   c3                      retq
```

然而 hello 的反汇编，赋予了地址

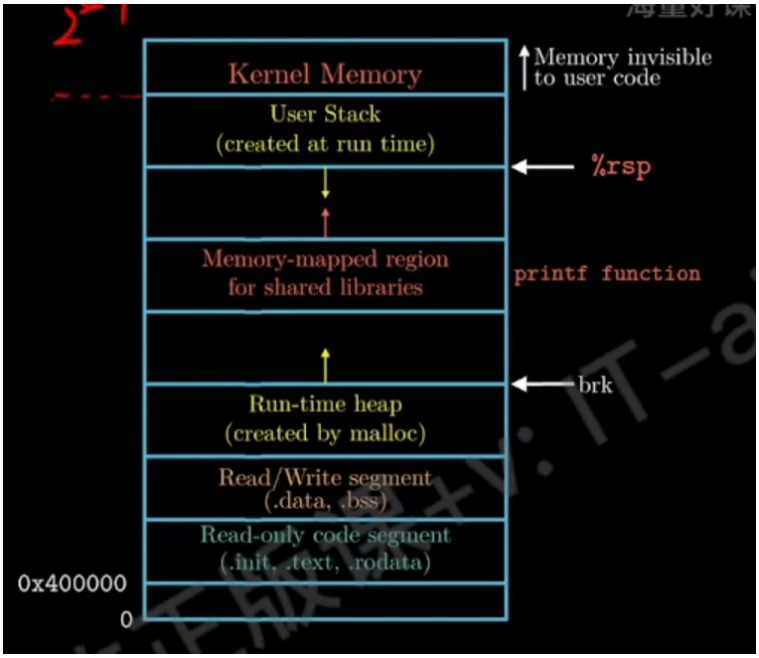
```
00000000000004e8 <_init>:
 4e8:   48 83 ec 08             sub    $0x8,%rsp
 4ec:   48 8b 05 f5 0a 20 00    mov    0x200af5(%rip),%rax        # 200fe8 <__gmon_start__>
 4f3:   48 85 c0                test   %rax,%rax
 4f6:   74 02                   je     4fa <_init+0x12>
 4f8:   ff d0                   callq  *%rax
 4fa:   48 83 c4 08             add    $0x8,%rsp
 4fe:   c3                      retq

Disassembly of section .plt:

0000000000000500 <.plt>:
 500:   ff 35 ba 0a 20 00       pushq  0x200aba(%rip)        # 200fc0 <_GLOBAL_OFFSET_TABLE_+0x8>
 506:   ff 25 bc 0a 20 00       jmpq   *0x200abc(%rip)       # 200fc8 <_GLOBAL_OFFSET_TABLE_+0x10>
 50c:   0f 1f 40 00             nopl   0x0(%rax)

0000000000000510 <puts@plt>:
 510:   ff 25 ba 0a 20 00       jmpq   *0x200aba(%rip)       # 200fd0 <puts@GLIBC_2.2.5>
 516:   68 00 00 00 00          pushq  $0x0
```

好像是一个地址对应一个字节
CPU 在执行这些指令，读写的都是虚拟地址
0x200af5(%rip)，得到的是虚拟地址，还要经过 MMU 转换成物理地址
我就不需要考虑 4G/8G 等等内存大小
进程独占一个虚拟地址空间



程序的装入和链接

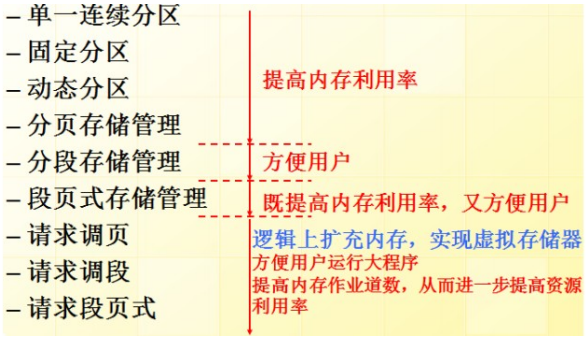
如何把逻辑地址转换为物理地址?

1、 绝对装入
在编译时，编译程序将产生绝对地址的目标代码，装入程序按照装入模块中的地址

2、 静态重定位
把地址重定位放在装入模块到内存的适当位置时
不能移动位置
3、 动态重定位
重定位寄存器
真正执行指令时，再将指令本身那个或指令中操作数的逻辑地址转换为物理地址
允许程序在内存中发生移动

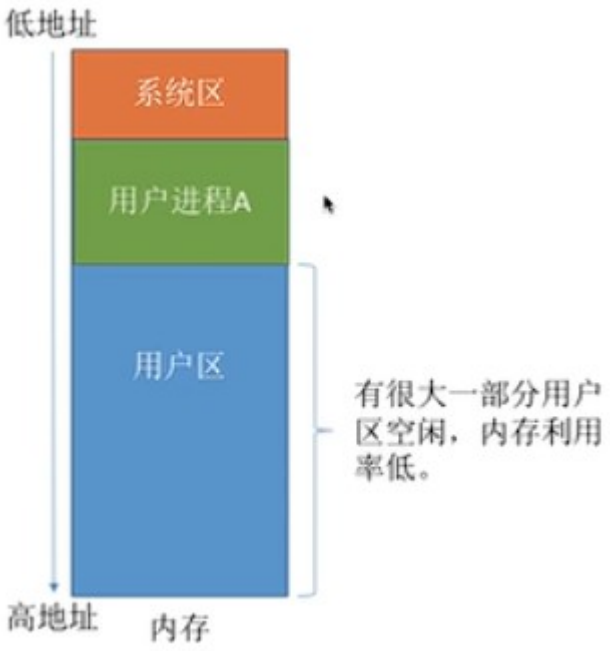
需要一个数据结构将虚拟地址和物理地址对应关系
页表，放在内存里，记录虚拟页和物理页的映射关系
内存空间的分配与回收
从逻辑上对内存空间进行扩充
负责逻辑地址和物理地址转换——三种装入方式页式存储/段式存储 动态重定位
内存保护 设置上下限寄存器 2、重定位寄存器（基址寄存器）和界地址寄存器（限长寄存器）进行越界检查
重定位寄存器存放到时进程的实际物理地址
界地址寄存器中存放的是最大物理地址

连续存储器管理方式



单一连续分区

内存被分为系统区和用户区，内存中只能有一道用户程序



固定分区

分区大小相等、分区大小不等
每个分区只装入一道程序

操作系统需要建立一个数据结构——分区说明表，来实现各个分区的分配与回收。每个表项对应一个分区，通常按分区大小排列。每个表项包括对应分区的大小、起始地址、状态（是否已分配）。

分区号	大小（MB）	起始地址（M）	状态
1	2	8	未分配
2	2	10	未分配
3	4	12	已分配
……	……	……	……

用数据结构的数组（或链表）即可表示这个表

内存（分区大小不等）

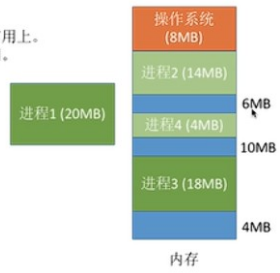
无外部碎片
会产生内部碎片
外部碎片：系统进行分区后产生的碎片
内部碎片：进程无法将系统分配的分区全部使用

分区分配表和相对应的分配回收算法实现

可变分区方式

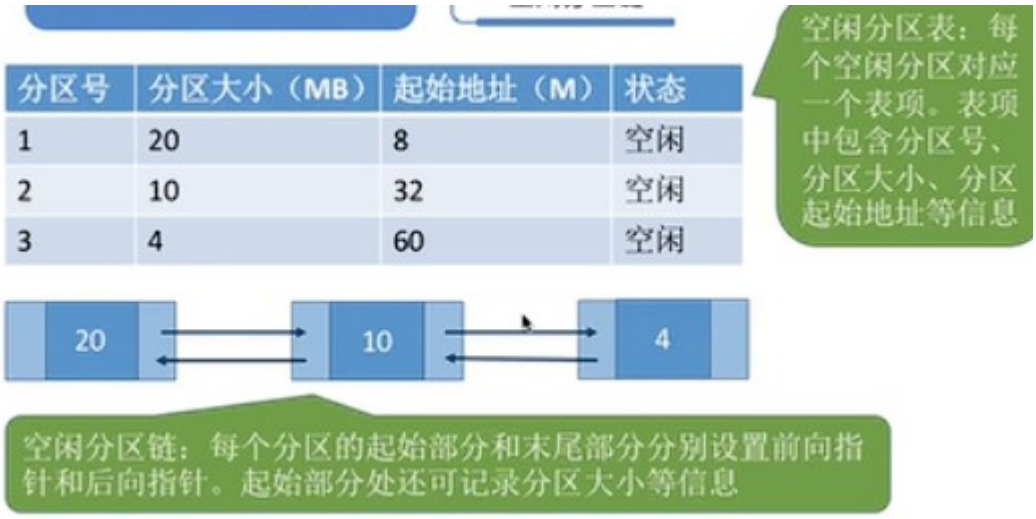
动态分区分配没有内部碎片，但是有外部碎片。
内部碎片，分配给某进程的内存区域中，如果有些部分没有用上。
外部碎片，是指内存中的某些空闲分区由于太小而难以利用。

如果内存中空闲空间的总和本来可以满足某进程的要求，但由于进程需要的是一整块连续的内存空间，因此这些“碎片”不能满足进程的需求。



内存

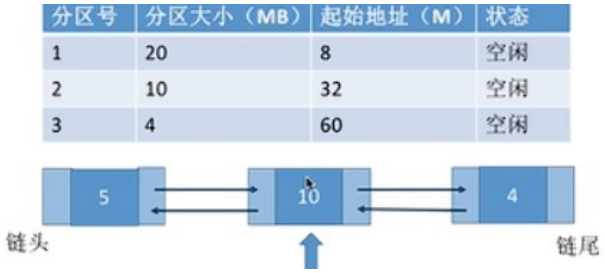
没有内部碎片，但是有外部碎片
不会预先划分内存分区，而是在进程装入内存时，根据进程的大小动态地建立分区
空闲分区表
空闲分区链



分配算法

1、 首次适应算法

空闲分区以地址递增的次序排列，每次分配内存时顺序查找空闲分区链，从第一个空闲分区开始查找，找到第一个可以满足需求的分区就进行必要的划分和分配



2、 最佳适应算法

为了保证当“大进程”到来时能有连续的大片空间，可以尽可能多地流线大片空闲区

如何实现：空闲分区按容量递增次序链接。每次分配内存时顺序查找空闲分区链

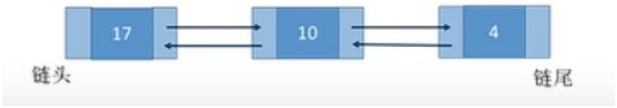
容易产生外部碎片

3、 最坏适应算法

根据分区链中根据分区查找与请求相差最大的分区

按容量递减的次序，找到大小能满足要求的一个空闲分区

分区号	分区大小（MB）	起始地址（M）	状态
1	20	8	空闲
2	10	32	空闲
3	4	60	空闲

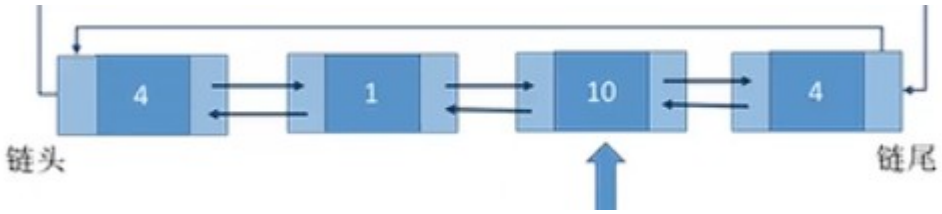


大进程到达，就没有内存分区可用了

4、邻近适应算法

首次适应算法很想，每次都从上次查找结束的位置开始检索

构造一个循环链表，每次分配内存时从上次查找的结束位置开始查找



邻近适应算法无论低地址和高地址部分的空闲分区的概率使用，也就导致了无大分区可用

回收算法

1、 上邻接
2、 下邻接
3、 上下邻接
4、 无邻接

非连续的分配方式

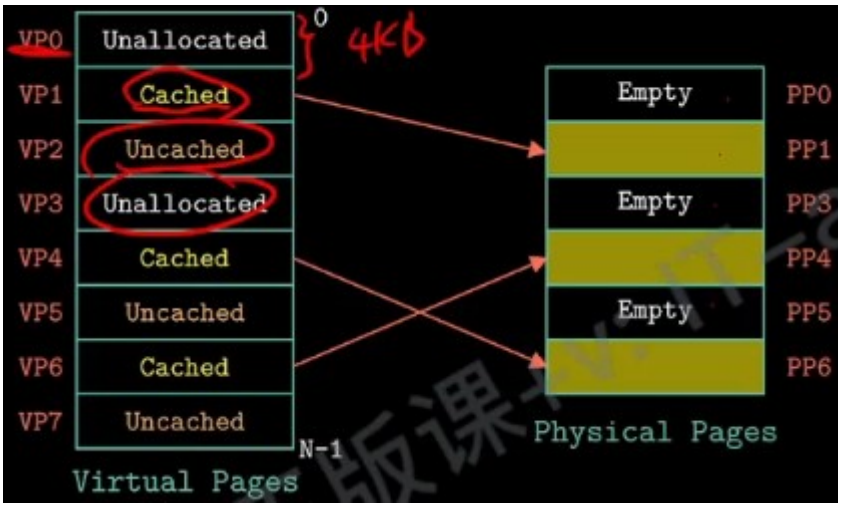
将内存的物理空间和程序逻辑空间划分为大小相等的块。划分为大小和数量都固定的基本分区

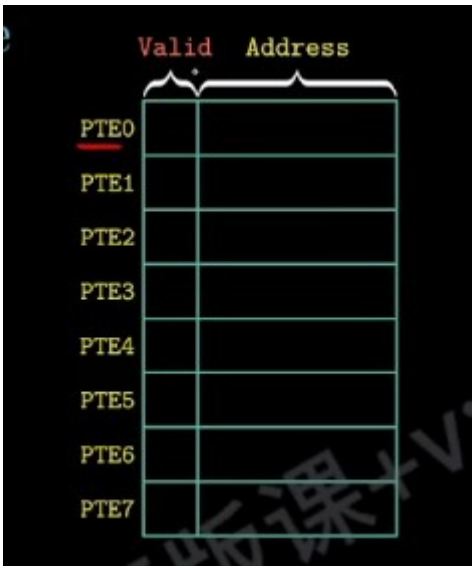
基本地址变换机构（用于实现逻辑地址到物理地址转换的一组硬件机构）

分页存储管理的基本原理

将物理内存划分成一块块（页）大小为 4KB，叫做页框-页帧=内存块=物理块=物理页面

每个页框有一个编号，即“页框号”（页帧号=内存块号=物理块号=物理页号）
页框号从 0 开始
将进程的逻辑地址空间也分为页框大小相等的一个个部分
每个部分称为一个“页”（页面），每个页面也有一个编号，即页号，页号从 0 开始
操作系统以页框为单位为各个进程分配内存空间，进程的每个页面分别放入一个页框中。
也就是说，进程的页面与内存的页框有一一对应的关系。
不必连续存放，也无前后次序要求，只要求足够容纳所有的物理块即可。
将进程的逻辑地址空间
将物理地址空间也划分成一页页
通常大小和虚拟地址空间一样
物理地址空间的每一页也叫做页框
在物理地址空间已有的部分是已缓存的 cache 的
只要确定了每个页面的大小，逻辑地址结构就确定了。因此，页式管理中地址是一维的
Uncache 的，在磁盘上，还没在物理地址空间
Unallocated，在磁盘上也没有



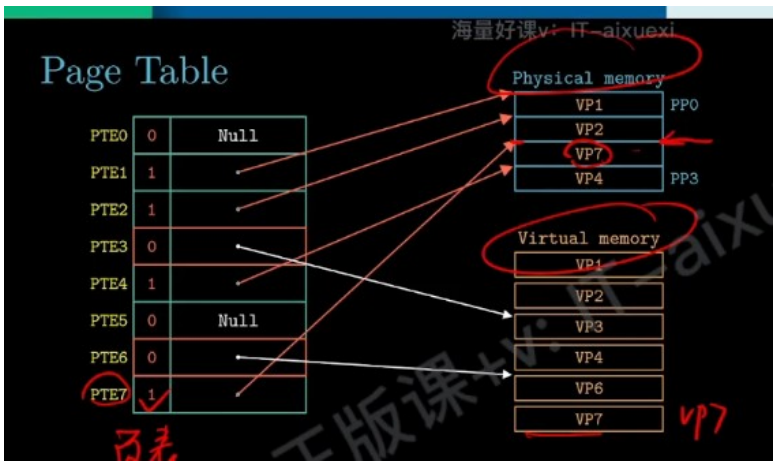


PTE 页表项

PTED0 valid=1 虚拟页 0 已经缓存在物理内存 address，根据 address 来在物理内存获取数据

=0，不在物理内存中，发生了缺页，进行缺页处理程序，从磁盘上调到虚拟内存里来

虚拟虚拟，其实并不存在，存在的是物理页



分页系统的逻辑地址空间分为两部分：页号 P 和页内地址 d（又称为页内偏移量）
页面的大小决定页内地址的位数，页号位数决定了逻辑地址空间中页面的总数。

页表——为了直到每个页面在内存中存放到位置，要为每个进程创建一个页表
通常在 PCB 中
每一个页面对应一个页表项
每个页表项由页号和块号组成
页表记录进程页面和实际存放的内存块之间的映射关系
页表存储在内存中,只存储物理块号，页号不占用存储空间（因为是按页号是按顺序排列的，只有块号）
然后将页表的起始地址及长度保存在进程的 PCB 中，当以后调度进程到 CPU 上运行时，再将 PCB 保存到页表起始地址及长度写入 CPU 的页表寄存器中
虽然进程的各个页面是离散存放的，但是页面内部是连续存放的

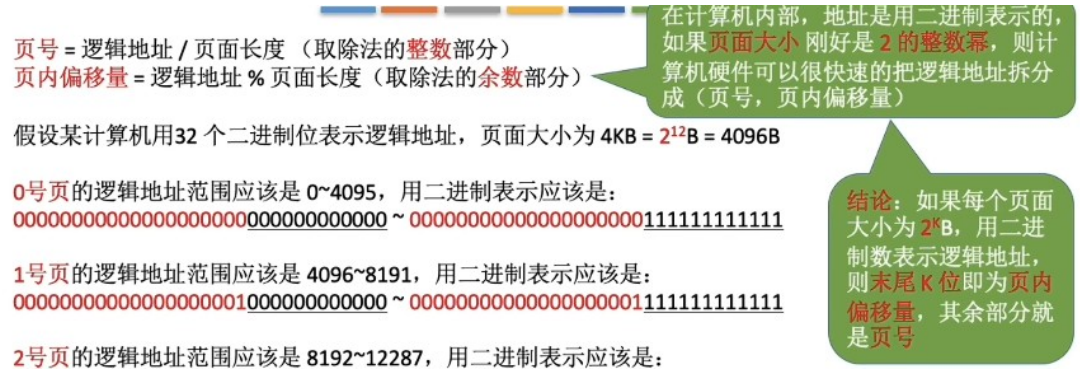
如果要访问逻辑地址 A，则
①确定逻辑地址A 对应的“页号” P
②找到P号页面在内存中的起始地址（需要查页表）
③确定逻辑地址A 的“页内偏移量” W

逻辑地址A 对应的物理地址 = P号页面在内存中的起始地址+页内偏移量W

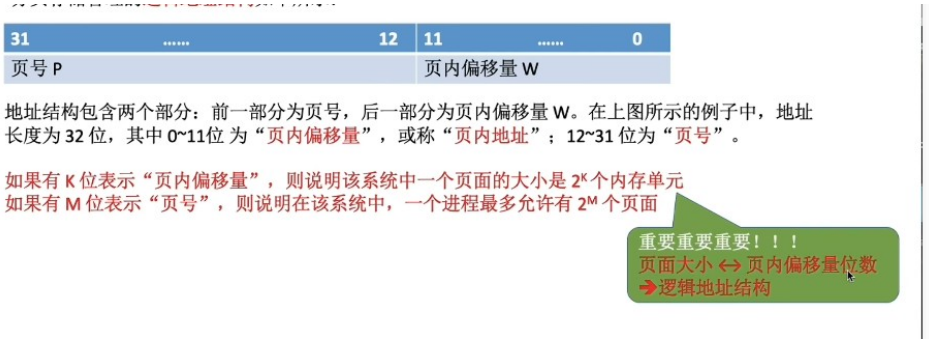
如何计算：
页号 = 逻辑地址 / 页面长度 （取除法的整数部分）
页内偏移量 = 逻辑地址 % 页面长度（取除法的余数部分）

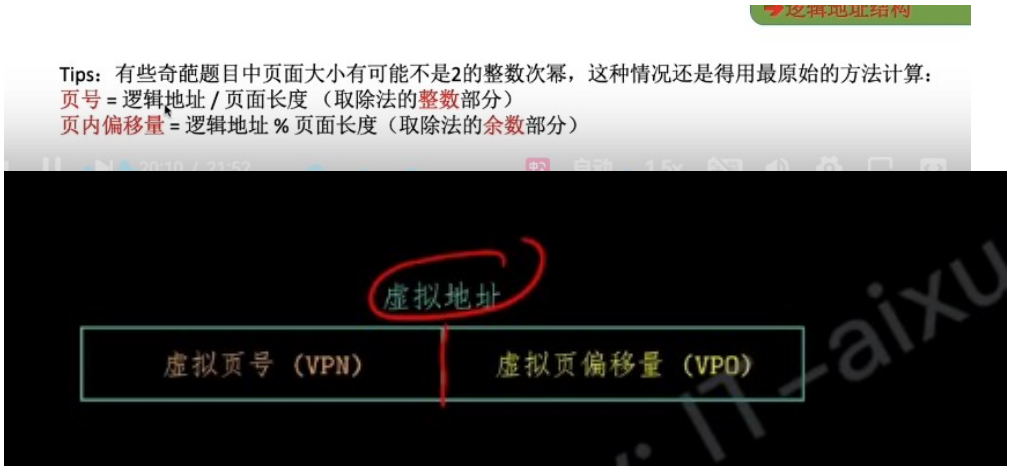
页号 = 110 / 50 = 2
页内偏移量 = 110 % 50 = 10

逻辑地址 可以拆分为（页号，页内偏移量）

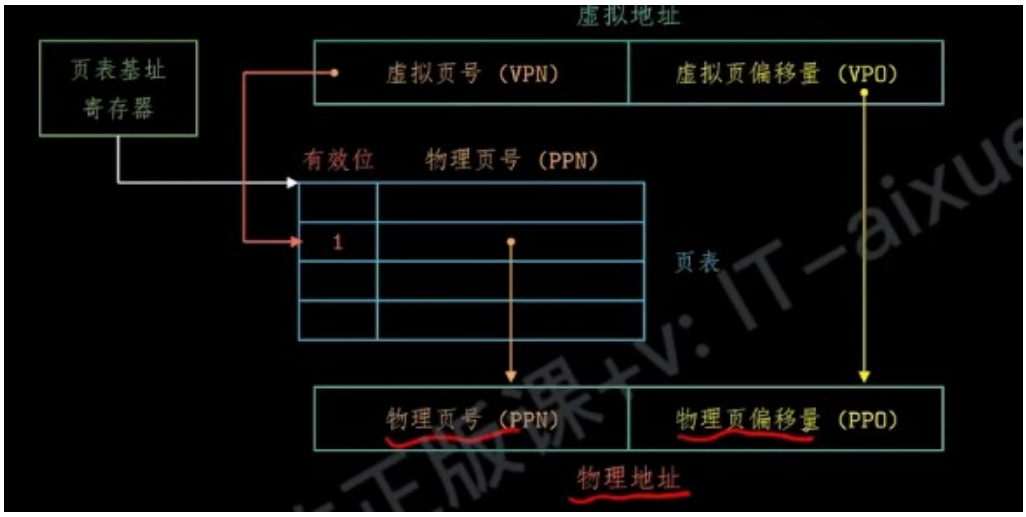


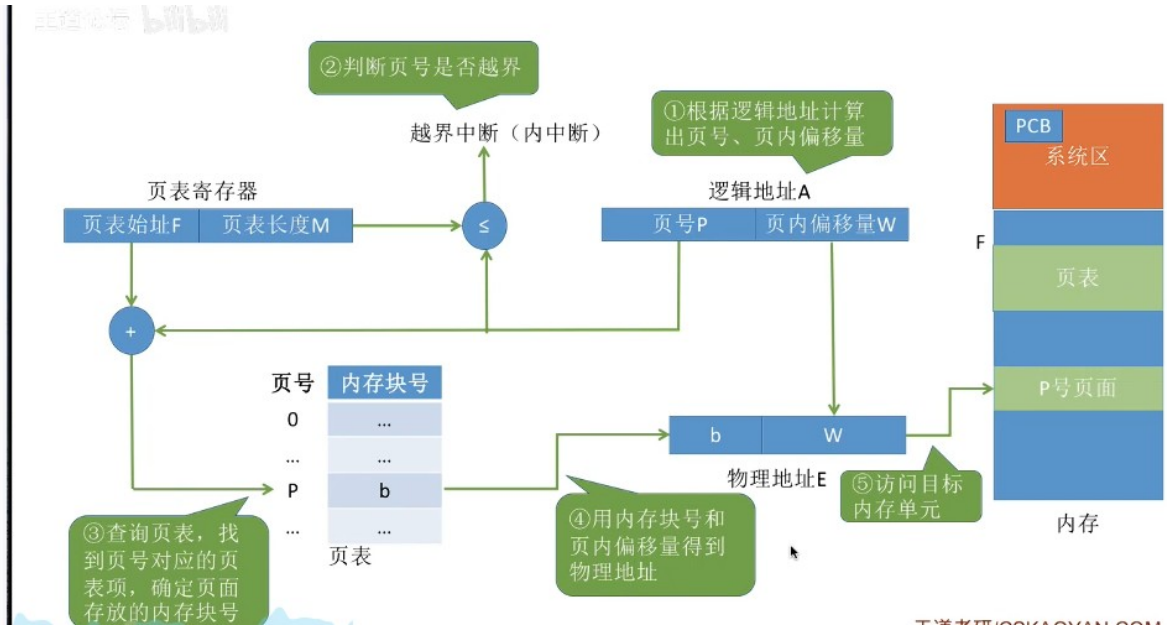
假设通过查询页表得知1号页面存放的内存块号是9（1001），则
9号内存块的起始地址 = 9*4096 = 00000000000000001001000000000000
则逻辑地址4097对应的物理地址 = 页面在内存中存放的起始地址 + 页内偏移量
=（00000000000000001001000000000001）





VP0:根据虚拟页大小 4KB，长度为 12 位，页内地址
在一个页 4kb 中的起始位置
虚拟页根据地址位数，32 位，32-12=20 位，虚拟页号 20 位，有 2^20 个虚拟页。
分页系统的逻辑地址空间是一维线性地址空间





注意：页面大小是2的整数幂

设页面大小为L，逻辑地址A到物理地址E的变换过程如下：

①计算页号 P 和页内偏移量W（ 如果用十进制数手算，则 P=A/L，W=A%L；但是在计运行时，逻辑地址结构是固定不变的，因此计算机硬件可以更快地得到二进制表示的内偏移量）

②比较页号P 和页表长度M，若 P≥M，则产生越界中断，否则继续执行。（注意：页始的，而页表长度至少是1，因此 P=M 时也会越界）

③页表中页号P对应的页表项地址 = 页表起始地址F + 页号P * 页表项长度，取出该页即为内存块号。（注意区分页表项长度、页表长度、页面大小的区别。页表长度指的表中总共有几个页表项，即总共有几个页；页表项长度指的是每个页表项占多大的存页面大小指的是一个页面占多大的存储空间）

④计算 E = b * L + W，用得到的物理地址E 去访存。（如果内存块号、页面偏移量是示的，那么把二者拼接起来就是最终的物理地址了）

动手验证：假设页面大小 L = 1KB，最终要访问的内存块号b = 2，页内偏移量W= 1023
①尝试用 E = b * L + W 计算目标物理地址。
②尝试把内存块号、页内偏移量用二进制表示，并把它们拼接起来得到物理地址。
对比①②的结果是否一致

Eg：假设某系统物理内存大小为 4GB，页面大小为 4KB，则每个页表项至少应该为多少字节？

内存块大小=页面大小=4KB= 2^{12} B
→4GB 的内存总共会被分为 $2^{32} / 2^{12} = 2^{20}$ 个内存块
→内存块号的范围应该是 0 ~ 2^{20} -1
→内存块号至少要用 20 bit 来表示

每个页表项的长度是相同的，页号是“隐含”的

Eg：假设某系统物理内存大小为 4GB，页面大小为 4KB，的内存总共会被分为 $2^{32} / 2^{12} = 2^{20}$ 个内存块，因此内存块号的范围应该是 0 ~ 2^{20} -1
因此至少要 20 个二进制位才能表示这么多的内存块号，因此至少要 3个字节才够（每个字节 8 个二进制位，3个字节共 24 个二进制位）

各页表项会按顺序连续地存放在内存中
如果该页表在内存中存放的起始地址为 X，则
M 号页对应的页表项是存放在内存地址为 X + 3*M

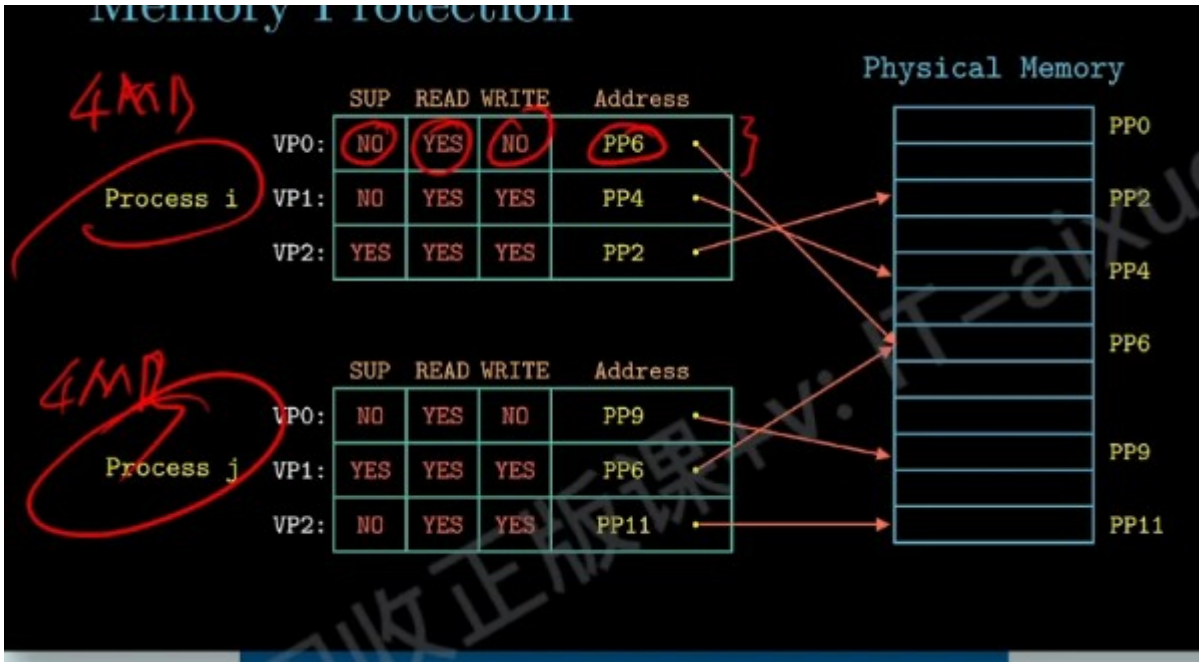
一个页面为 4KB，则每个页框可以存放 4096/3 = 1365 个页表项，但是这个页框会剩余 4096 % 3 = 1 B 页内碎片
因此，1365 号页表项存放的地址为 X + 3*1365 + 1
如果每个页表项占 4字节，则每个页框刚好可存放 1024 个页表项

X
剩余 1B

页表项时连续地存放在内存中！
理论上来说，3B 即可表示内存块号的范围
但是为了方便页表的查询，常常让每个页面恰好可以装得下整数个页表项

物理内存大小为 4GB，页面大小为 4KB
内存块大小=页面大小=4KB
内存一共有 2^20 内存块
由于以字节编址，24bit
至少要用 24 位（3B）来表示块号

虚拟页偏移量=物理页偏移量
如何快速找到页表?
页表基址寄存器
每个页表项大小相同，4B
页表项是顺序的，0-n
假设虚拟地址 32 位
虚拟页大小是 4KB（即页内偏移量为 12 位）
有多少个虚拟页？2^32/2^12=2^20,即 2^20 个虚拟页
每一个虚拟页就有一个虚拟页号
再假设每一个页表项 4B
那么就有 2^20x4B 即占 4MB 大小
每一个进程都有一个独立的页表（4MB）
页表占用了大量的内存空间



注意：页面大小是2的整数幂

设页面大小为L，逻辑地址A到物理地址E的变换过程如下：

①计算页号 P 和页内偏移量W（如果用十进制数手算，则 P=A/L，W=A%L；但运行时，逻辑地址结构是固定不变的，因此计算机硬件可以更快地得到二进制内偏移量）

②比较页号P 和页表长度M，若 P≥M，则产生越界中断，否则继续执行。（注始的，而页表长度至少是1，因此 P=M 时也会越界）

③页表中页号P对应的页表项地址 = 页表起始地址F + 页号P * 页表项长度，取出即为内存块号。（注意区分页表项长度、页表长度、页面大小的区别。页表长表中总共有几个页表项，即总共有几个页；页表项长度指的是每个页表项占多页面大小指的是一个页面占多大的存储空间）

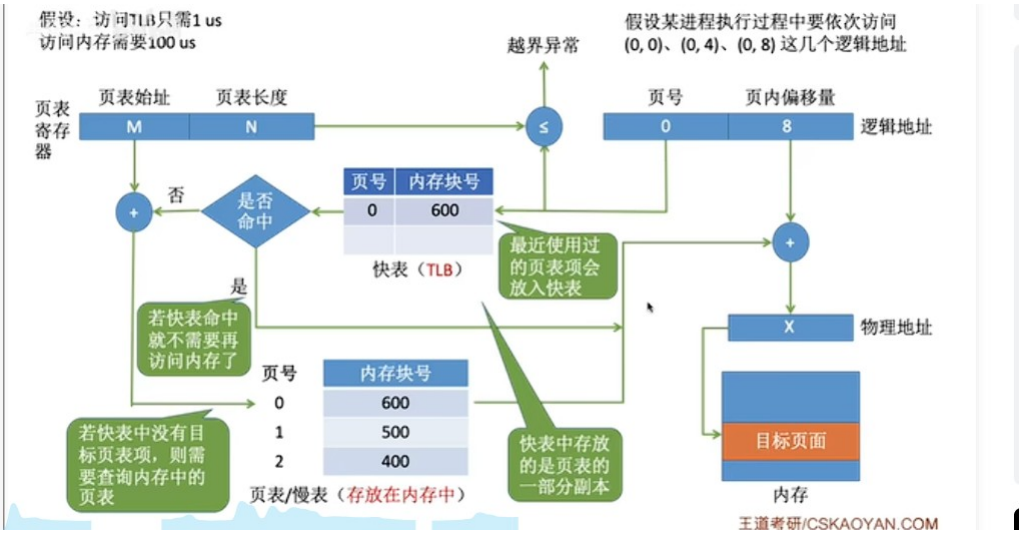
④计算 E = b * L + W，用得到的物理地址E 去访存。（如果内存块号、页面偏移示的，那么把二者拼接起来就是最终的物理地址了）

快表 TLB

不是内存，是一种高速缓存

与此对应，内存中的页表常称为慢表

用来存放最近访问的页表项的副本



①CPU给出逻辑地址，由某个硬件算得页号、页内偏移量，将页号与快表中的所有页号进行比较。

②如果找到匹配的页号，说明要访问的页表项在快表中有副本，则直接从中取出该页对应的内存块号，再将内存块号与页内偏移量拼接形成物理地址，最后，访问该物理地址对应的内存单元。因此，若快表命中，则访问某个逻辑地址仅需一次访存即可。

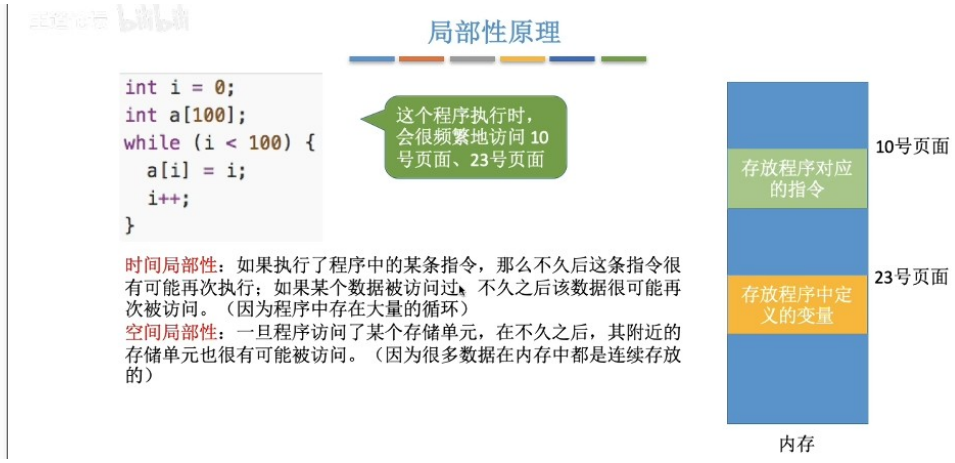
③如果没有找到匹配的页号，则需要访问内存中的页表，找到对应页表项，得到页面存放的内存块号，再将内存块号与页内偏移量拼接形成物理地址，最后，访问该物理地址对应的内存单元。因此，若快表未命中，则访问某个逻辑地址需要两次访存（注意：在找到页表项后，应同时将其存入快表，以便后面可能的再次访问。但若快表已满，则必须按照一定的算法对旧的页表项进行替换）

由于查询快表的速度比查询页表的速度快很多，因此只要快表命中，就可以节省很多时间。

、

问一次内存耗时 100us。若快表的命中率为 90%，那么访问一个逻辑地址的平均耗
(1+100) * 0.9 + (1+100+100) * 0.1 = 111 us

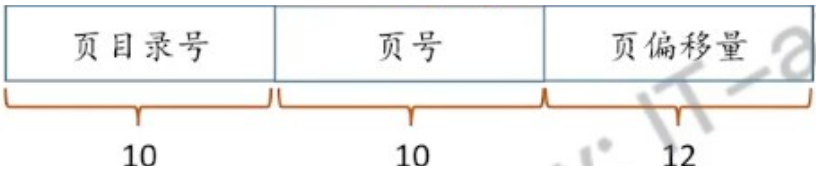
局部性原理

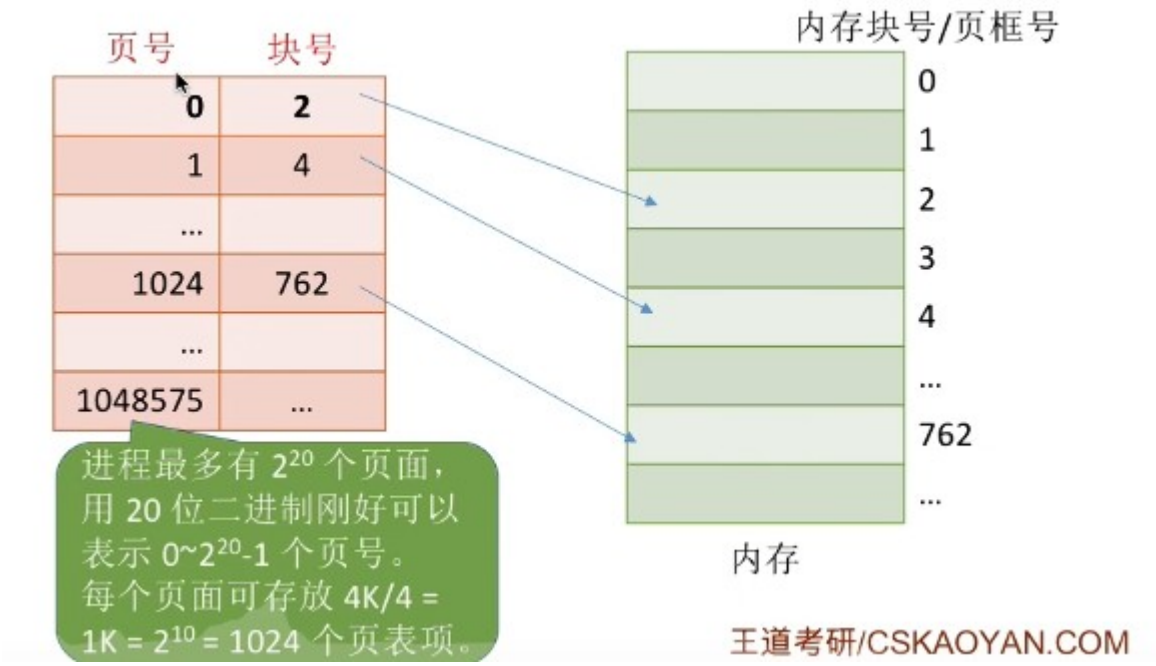


因此我们要建立多级页表，减少内存存储页表的空间
多级页表
采用两级页表的形式
页表是连续存放在进程当中

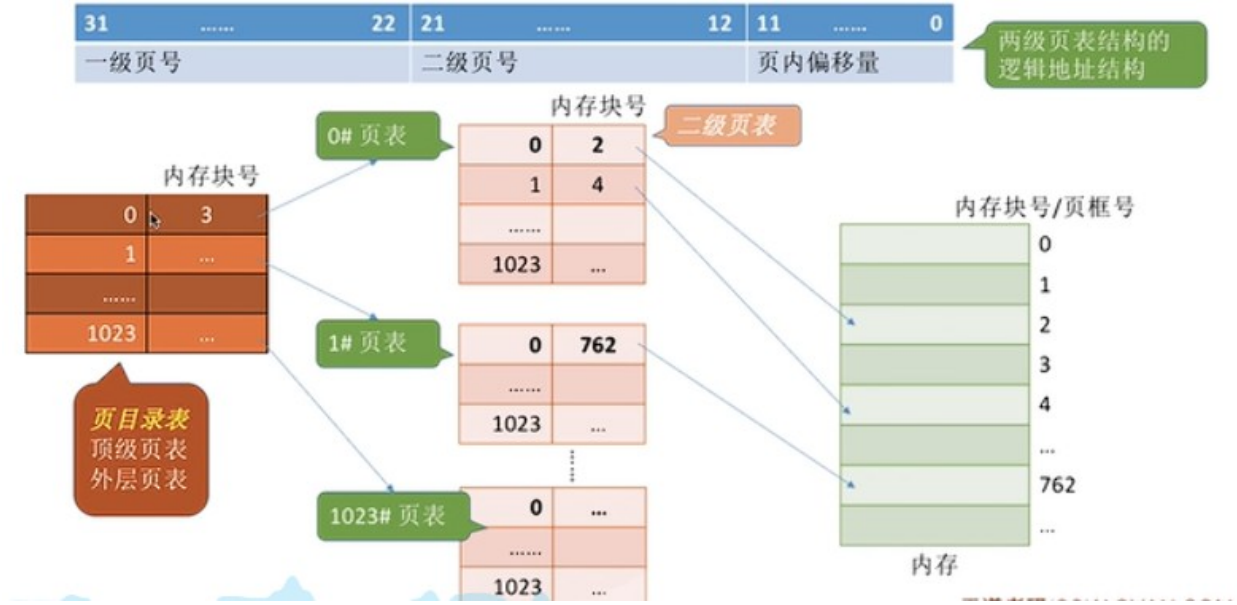
问题一：页表必须连续存放，因此当页表很大时，需要占用很多个连续的页框。
问题二：没有必要让整个页表常驻内存，因为进程在一段时间内可能只需要访问

因此，我们可以参照进程在内存中必须连续存储的问题，我们通过了页号+页内偏移量+页表的机制
同样的可以解决页表必须连续存放的问题，把必须连续存放的页表再分页
如页面大小 4KB，每个表项 4B，每个页面可以存放 1K 个页表项，因此每 1K 个连续的页表项为一组，每组刚好占一个内存块，再将各组离散地存放到各个内存块中
要为离散分配的页表再建立一张页表，称为页目录表，或称外层页表、顶层页表





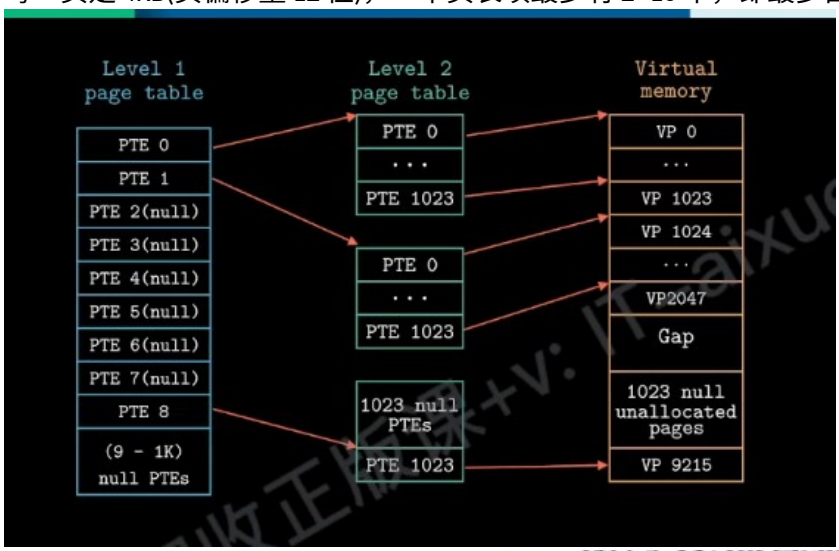
一个页面可以存放 1024 个页表

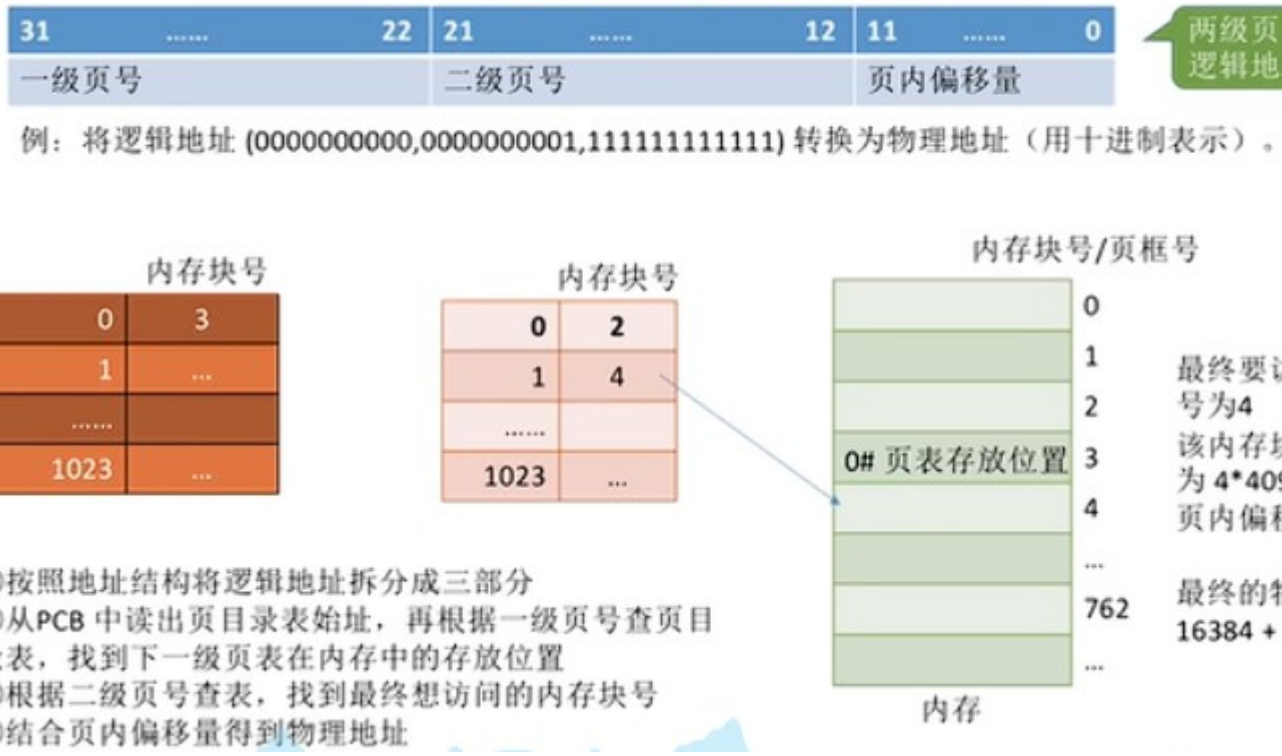


就要为这些小页表再建立一个表，叫做页目录表

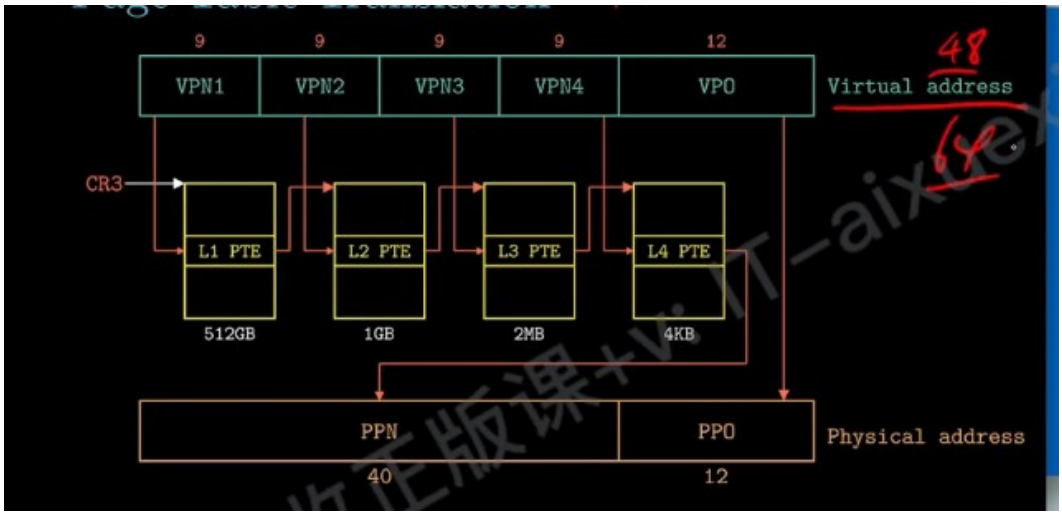
10 位一级页号表示一共有多少个页面来存放每个小页表项（2^22B/2^12B=1024）
10 位二级页号表示每个页面（4KB）可以存放多少个页表项（4KB/4B=1024）
一共加起来 20 位表示一共可以有多少个页表
小页表叫做二级页表
页目录号：表示 0-2^10-1
页目录号对应页目录项（页表项的一种）
每一个页目录号可以寻址一个二级的页表

0-2^10-1 个
每一页是 4KB(页偏移量 12 位)，一个页表项最多有 2^10 个，即最多占用的物理内存为 4MB





而一共有 2^10 个页目录项，共有 4MBx2^10=4GB
则可以通过来记录整个虚拟内存 4GB 的映射关系。



48 位的虚拟地址

对于问题二，可以在需要访问页面时才把也秒调入内存（虚拟存储技术）
属于内中断

可以在页表项中增加一个标志位，表示该页面是否已经调入内存

7. 没有快表时访问一个数据需要访问内存的次数

- 1 次（连续分配）

HDU 编程营：936217564

6/16/21

万物皆有裂痕，那是光进来的地方。

8- 2 次（一级分页存储管理、分段存储管理）

- 3 次（二级分页存储管理、段页式存储管理）

8. 动态分区分配

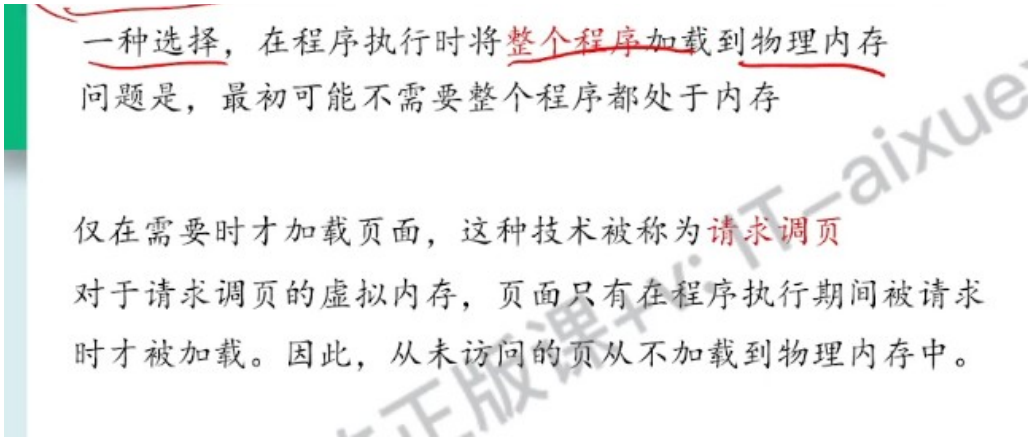
- 首次适应算法（空闲区按起始地址递增的次序拉链）
- 最佳适应算法（空闲区按分区大小递增的次序拉链）

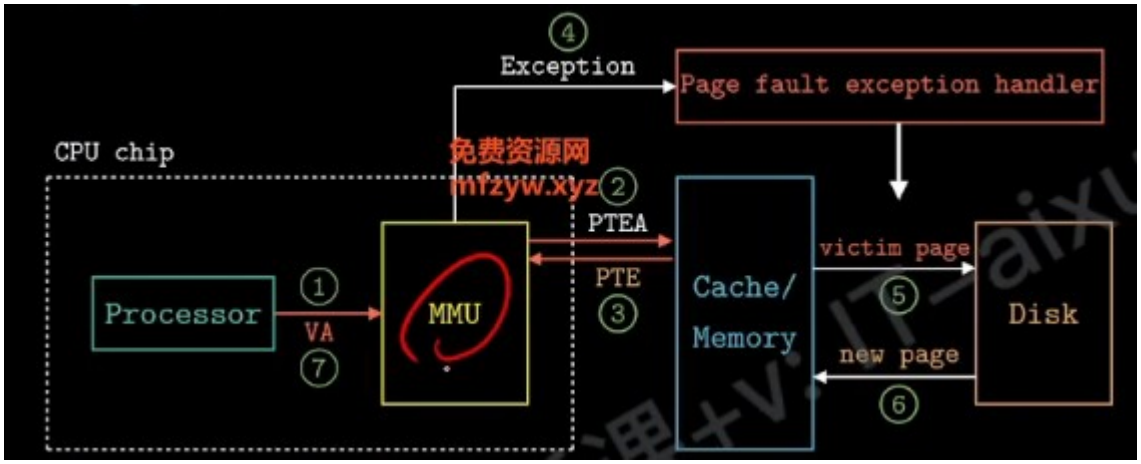
- 回收时要进行分区的合并（具体有前后都没有空闲分区、只是前面有空闲区、只是后面有空闲区、前后都是空闲区这四种情况）
- 碎片问题可采用紧凑技术加以解决
- 采用紧凑技术后的动态分区分配方式也叫可重定位分区分配方式（因为它需要得到动态重定位技术的支持）

9. 对换

- 所谓“对换”，是指把内存中暂时不能运行的进程或暂时不用的程序或数据，调出到外存上，以便腾出足够的内存空间，再把具备运行条件的进程或进程所需要的程序和数据，调入内存。
- 整体对换：以进程为单位的对换。（但进程的 PCB 常驻内存不应该被换出；进程的程序段如果正在被其他进程共享，也不应该被换出内存）
- 部分对换：以“页”或“段”为单位的对换

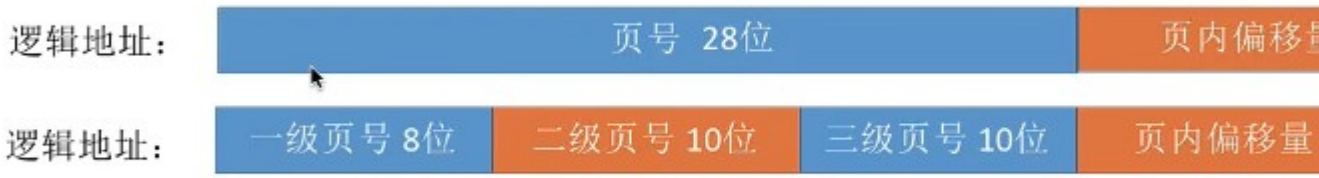
请求调页





1、 各级页表的大小不能超过一个页面（4KB）

表对应页号应为10位。总共28位的页号至少要分为三级

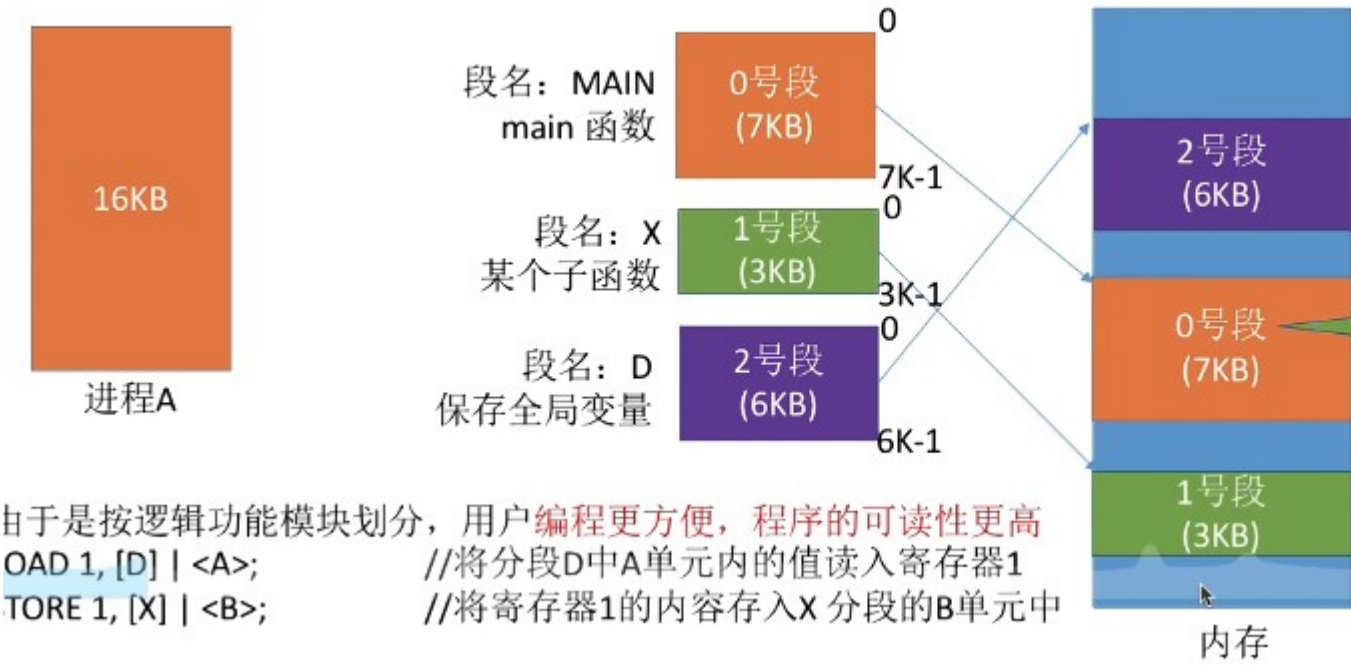


2、 时间上的增多

分段存储管理

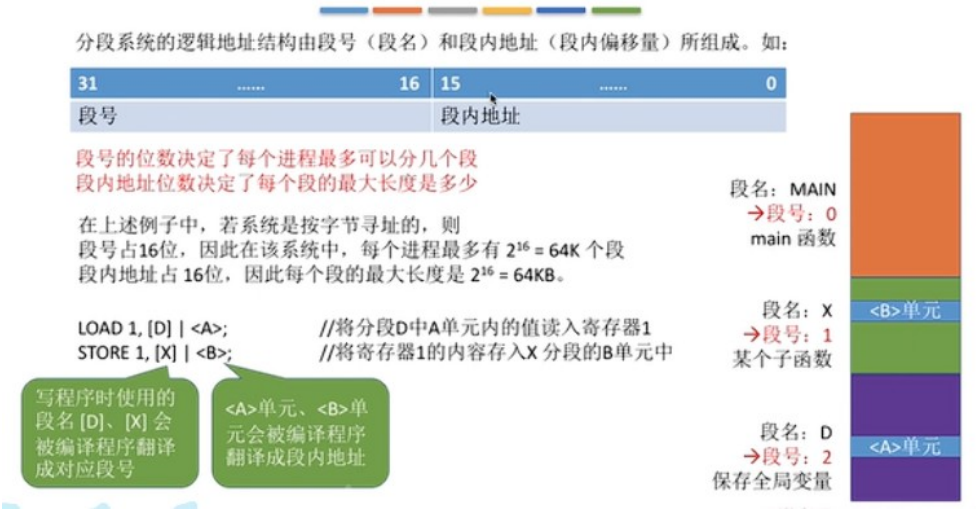
与分页最大的区别就是分配的地址空间不同

例如一个进程可以分为一个主程序段、子程序段、数据段等等



由于是按逻辑功能模块划分，用户编程更方便，程序的可读性更高

```
LOAD 1, [D] | <A>;          //将分段D中A单元内的值读入寄存器1
STORE 1, [X] | <B>;         //将寄存器1的内容存入X 分段的B单元中
```



段表

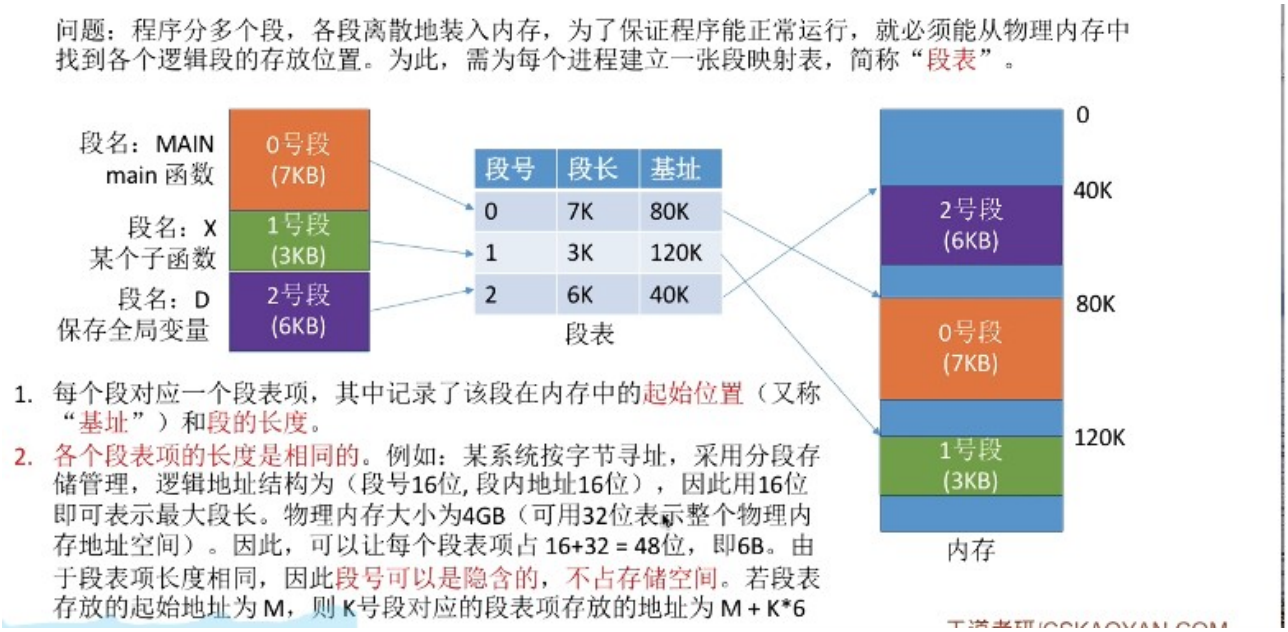
段号、段长和段基址

每个段对应一个段表项。因为在分页式管理当中，每个页面大小是相同的 4KB

每个段表项的长度相同

所以要对段内偏移量进行检查

这里不同的是，基址为整个物理地址（32 位）



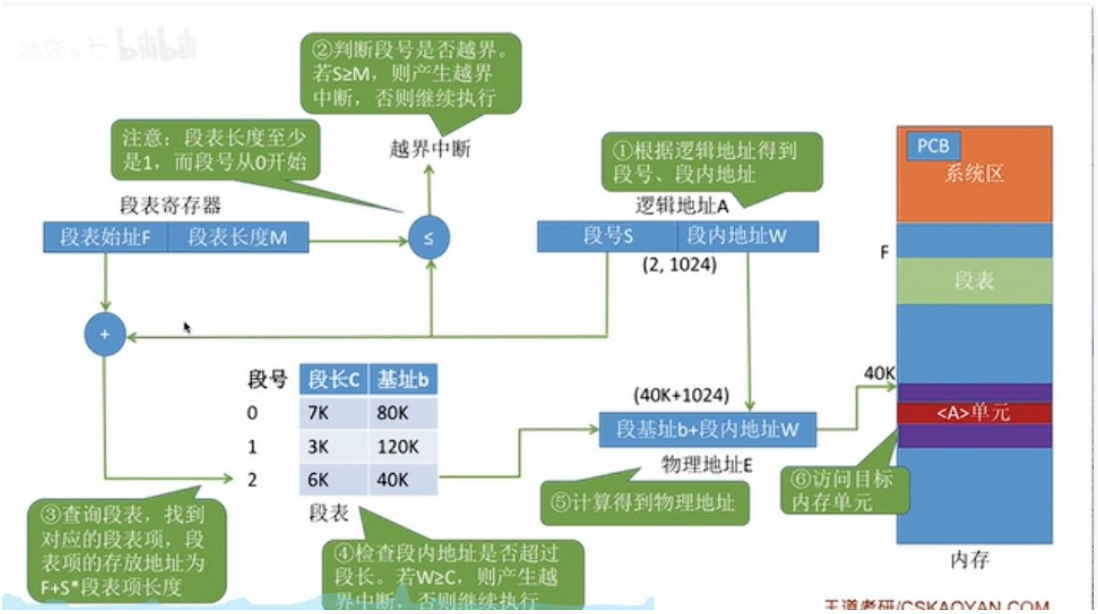
页是信息的物理单位。分页的主要目的是为了实现离散分配，提高内存利用率。分页仅仅是系统管理上的需要，完全是系统行为，对用户是不可见的。

段是信息的逻辑单位。分段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。

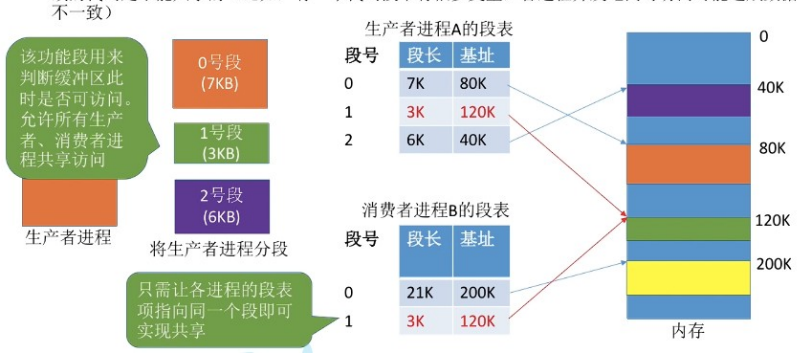


分段、分页管理的对比

比如，有一个代码段只是简单的输出“Hello World！”

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（比如，有一个代码段中有很多变量，各进程并发地同时访问可能造成数据不一致）



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。

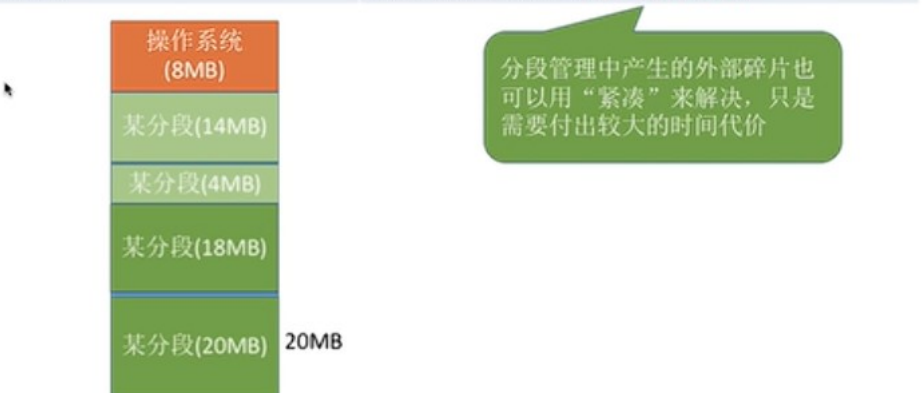
分页的用户进程地址空间是一维的，程序员只需给出一个记忆符即可表示一个地址。
分段的用户进程地址空间是二维的，程序员在标识一个地址时，既要给出段名，也要给出段内地址。

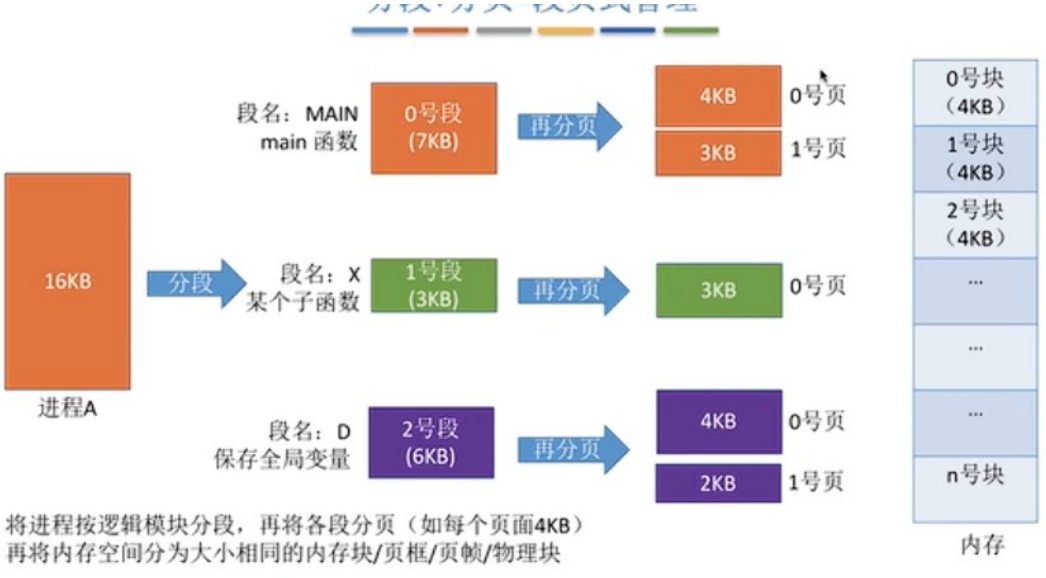
分段比分页更容易实现信息的共享和保护。不能被修改的代码称为纯代码或可重入代码（不属于临界资源），这样的代码是可以共享的。可修改的代码是不能共享的

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

	优点	缺点
分页管理	内存空间利用率高，不会产生外部碎片，只会有少量的页内碎片	不方便按照逻辑模块实现信息的共享和保护
分段管理	很方便按照逻辑模块实现信息的共享和保护	如果段长过大，为其分配很大的连续空间会很不方便。另外，段式管理会产生外部碎片





将进程按逻辑模块分段，再将各段分页（如每个页面4KB）
再将内存空间分为大小相同的内存块/页框/页帧/物理块

分段系统的逻辑地址结构由段号和段内地址（段内偏移量）组成。如：

31 16	15 0
段号	段内地址

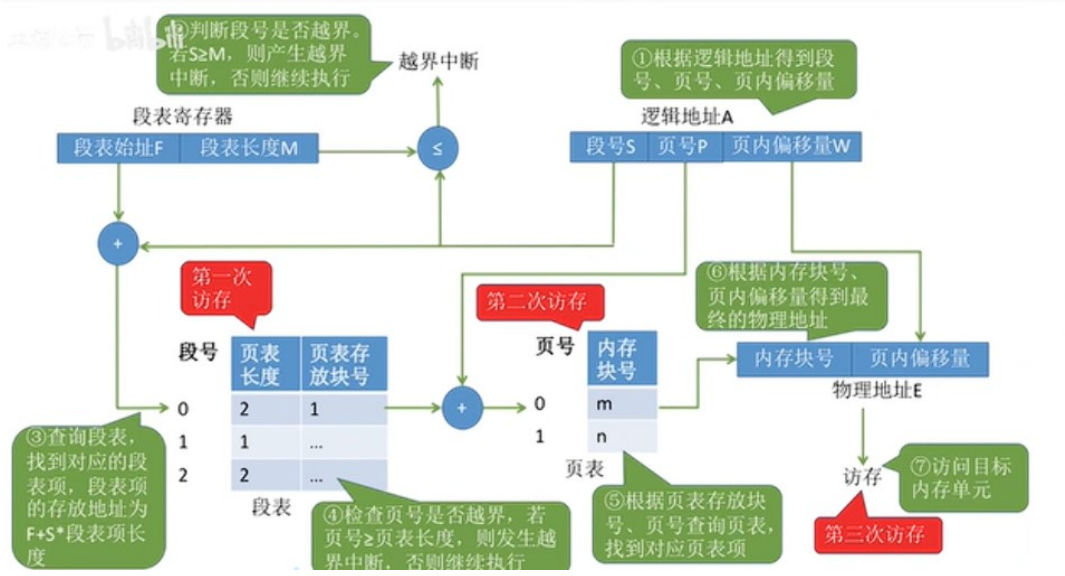
段页式系统的逻辑地址结构由段号、页号、页内地址（页内偏移量）组成。如：

31 16	15 12	11 0
段号	页号	页内偏移量

段号的位数决定了每个进程最多可以分几个段
页号位数决定了每个段最大有多少页
页内偏移量决定了页面大小、内存块大小是多少

在上述例子中，若系统是按字节寻址的，则
段号占16位，因此在该系统中，每个进程最多有 2^{16} = 64K 个段
页号占4位，因此每个段最多有 2^4 = 16页
页内偏移量占12位，因此每个页面\每个内存块大小为 2^{12} = 4096 = 4KB

“分段”对用户是可见的，程序员编程时需要显式地给出段号、段内地址。而将各段“分页”对用户是不可见的。系统会根据段内地址自动划分页号和页内偏移量。
因此段页式管理的地址结构是二维的。



与此对应，段页式的段表存储的是段号（隐藏）、页表长度、页表存放块号组成
每个段表项长度相等
每个页面对应一个页表项，每个页表项由页号（隐含）、页面存放的内存块号组成，每个页表项长度相等
虚拟存储系统
虚拟存储系统的基本概念
将进程装入的一次性或者整体性改为多次性：改变进程必须全部装入内存才能开始运行的方式 1、作业很大时，并不能全部装入内存，导致大作业无法运行。
当大量作业要求运行时，会只有少量作业能运行，导致多道程序并发度下降
将进程的驻留性改为置换性：当作业被装入内存，就会一直驻留在内存中。那么在需要时将暂时不用的部分换出到外存储器。

基于局部性原理，在程序装入时，可以将程序中很快会用到的部分装入内存，暂时用不到的部分留在外存，就可以让程序开始执行。

在程序执行过程中，当所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，然后继续执行程序。

若内存空间不够，由操作系统负责将内存中暂时用不到的信息换出到外存。

在操作系统的管理下，在用户看来似乎有一个比实际内存大得多的内存，这就是虚拟内存

操作系统虚拟性的一个体现，实际的物理内存大小没有变，只是在逻辑上进行了扩充。

多次性：

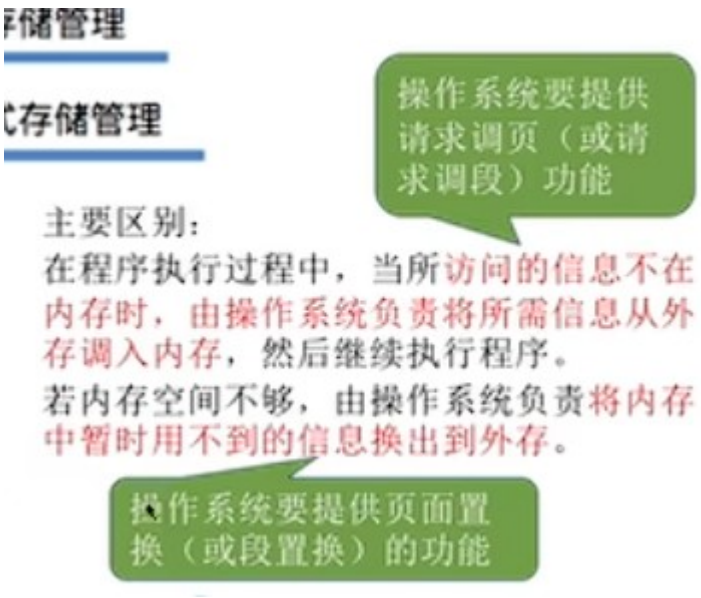
对换性：

驻留性：

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：在作业运行时无需一直常驻内存，而是允许在作业运行过程中，将作业换入、换出。

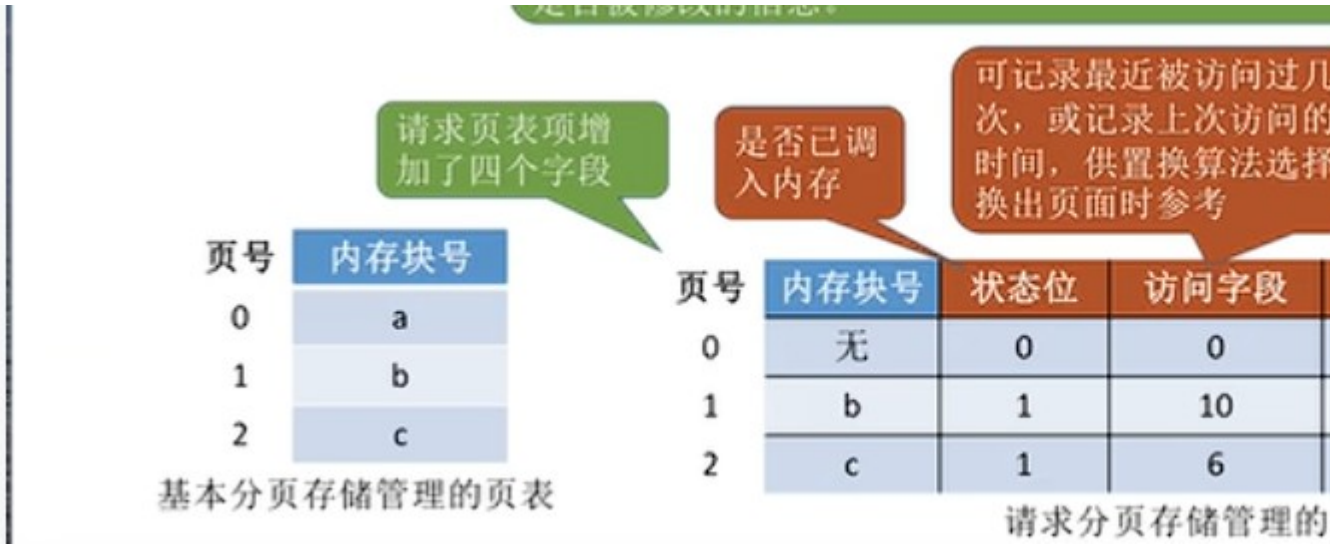
虚拟性：从逻辑上扩充了内存的容量，使用户看到的内存容量，远大于实际的容量。



局部性原理：时间局部性：数据可能再次被访问
空间局部性：邻近的存储单元也有可能被访问到

请求分页存储管理方式

页面置换
页面置换策略



缺页中断机构

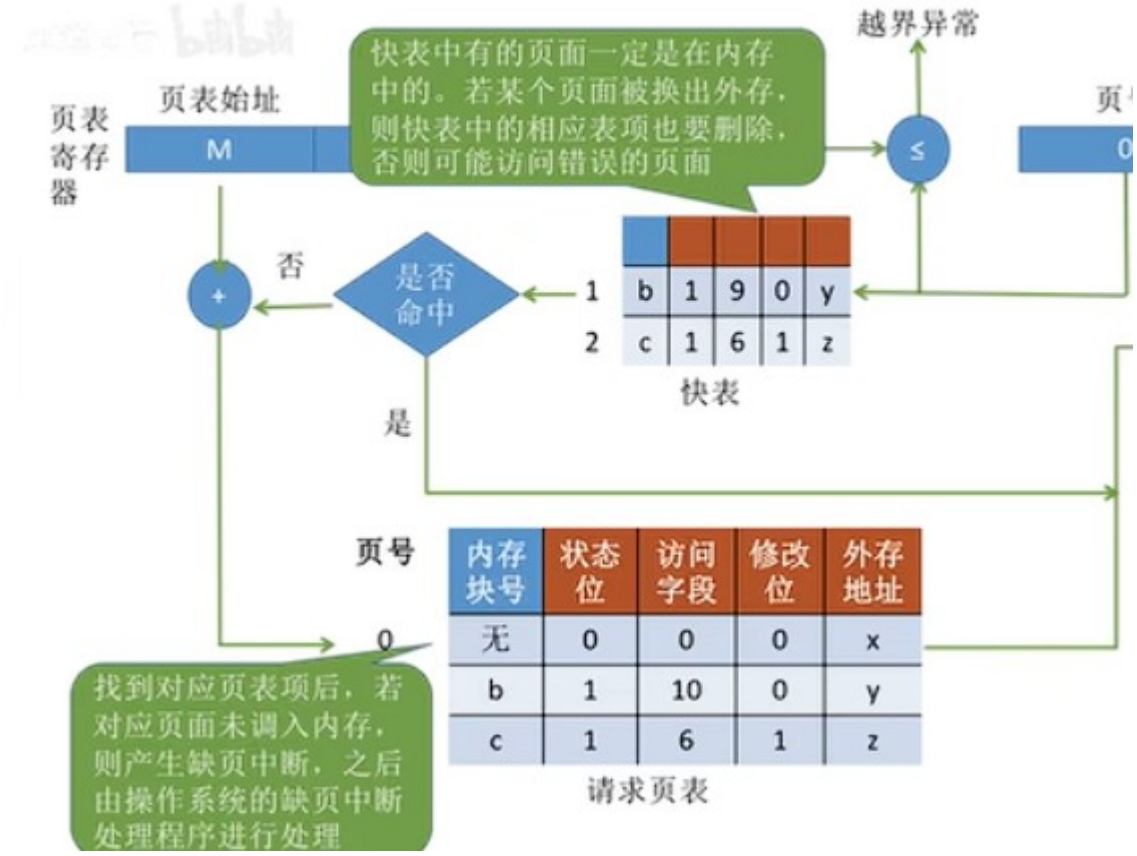
缺页中断机构

页号	内存块号	状态位	访问字段	修改位	外存地址
0	c	1	0	0	x
1	b	1	10	0	y
2	无	0	0	0	z

假设此时要访问逻辑地址=（页号，页内偏移量）=（0, 1024）

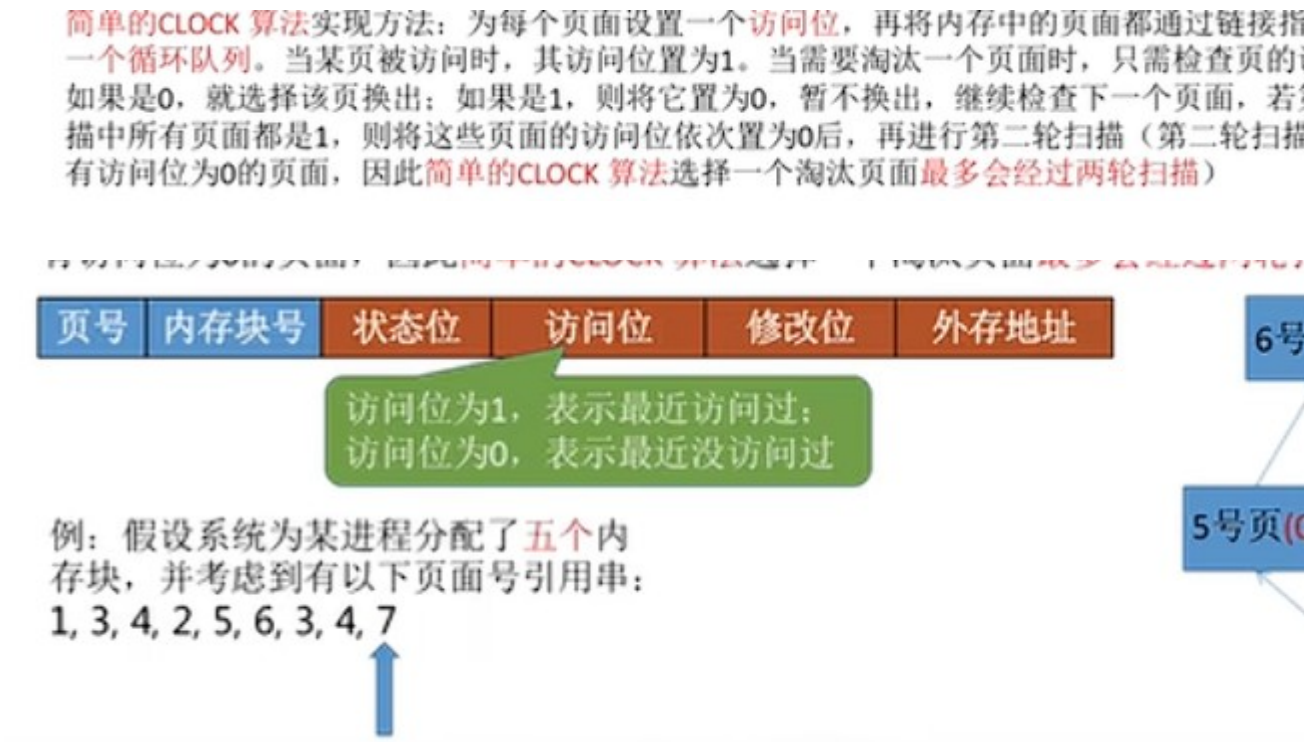
在请求分页系统中，每当要访问的页面不在内存时，便产生一个缺页中断，然后由操作系统的缺页中断处理程序处理中断。
此时缺页的进程阻塞，放入阻塞队列，调页完成后再将其唤醒，放回就绪队列。

如果内存中有空闲块，则为进程分配一个空闲块，将所缺页面装入该块，并修改页表中相应的页表项。
如果内存中没有空闲块，则由页面置换算法选择一个页面淘汰，若该页面在内存期间被修改过，则要将其写回外存。未修改过的页面不用写回外存。



1、请求调页
2、页面置换
3、修改数据

页面置换算法
1、 最佳置换算法 OPT
将来不在访问的页面或者长时间内不会访问的页面
2、 先进先出置换算法 FIFO
最先调入内存的页面，或者在也内存中驻留时间最久的页面算法
3、 最近最久未使用置换算法 LRU
最近一段时间内最长没有被访问的页面
4、 最近最少使用置换算法 LFU
被访问的频次而不是 LRU 中的事件
淘汰过去一段话时间里访问次数最少的页面
5、 时钟置换算法（CLOCK）
简单的时钟置换算法



改进型

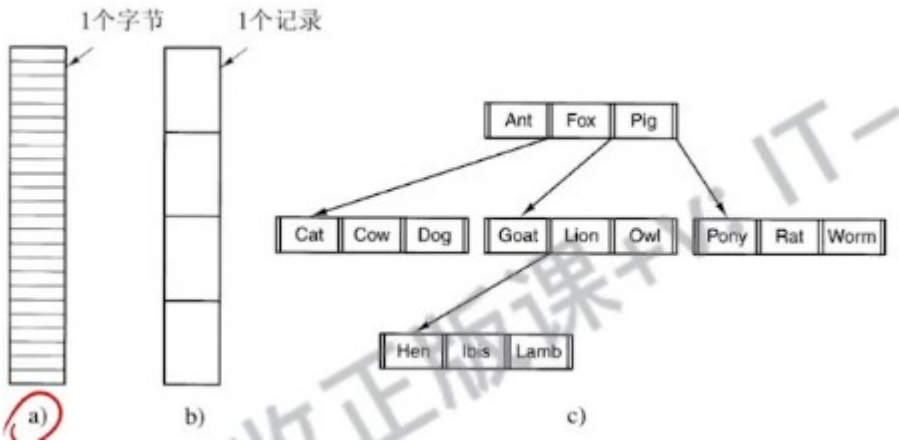
文件系统

文件命名

扩展名	含　义
file.bak	备份文件
file.c	C源程序文件
file.gif	符合图形交换格式的图像文件
file.hlp	帮助文件
file.html	WWW超文本标记语言文档
file.jpg	符合JPEG编码标准的静态图片
file.mp3	符合MP3音频编码格式的音乐文件
file.mpg	符合MPEG编码标准的电影
file.o	目标文件（编译器输出格式，尚未连接）
file.pdf	pdf格式的文件
file.ps	PostScript文件
file.tex	为TEX格式化程序准备的输入文件
file.txt	一般正文文件
file.zip	压缩文件

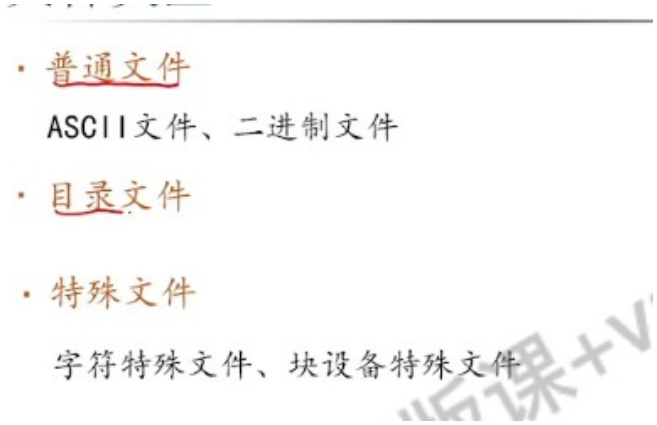
文件结构：

字节序列、记录序列、树
1、无结构的字节序列



3、 记录序列 有固定格式 基于
树形结构
由一棵记录树构成

文件类型



特殊文件：前者鼠标键盘后者磁盘
Gcc -o hello hello.c 可执行的文件就是二进制文件 ELF 格式
目录文件~文件夹
文件访问
顺序访问
从文件开始按顺序读取文件的全部字节或记录，不能跳过某一些内容
随机访问

能够以任何次序读取其中字节或记录的文件称为随机访问文件

文件属性（元数据）

属　性	含　义
保护	谁可以存取文件，以什么方式存取文件
口令	存取文件需要的口令
创建者	创建文件者的ID
所有者	当前所有者
只读标志	0表示读/写；1表示只读
隐藏标志	0表示正常；1表示不在列表中显示
系统标志	0表示普通文件；1表示系统文件
存档标志	0表示已经备份；1表示需要备份
ASCII/二进制标志	0表示ASCII码文件；1表示二进制文件
随机存取标志	0表示只允许顺序存取；1表示随机存取
临时标志	0表示正常；1表示进程退出时删除该文件
加锁标志	0表示未加锁；非零表示加锁
记录长度	一个记录中的字节数
键的位置	每个记录中键的偏移量
键的长度	键字段的字节数
创建时间	文件创建的日期和时间
最后一次存取时间	文件上一次存取的日期和时间
最后一次修改时间	文件上一次修改的日期和时间
当前大小	文件的字节数
最大长度	文件可能增长到的字节数

1、 文件名：由创建文件的用户决定文件名

2、 标识符:标识符是操作系统用于区分各个文件的一种内部名称

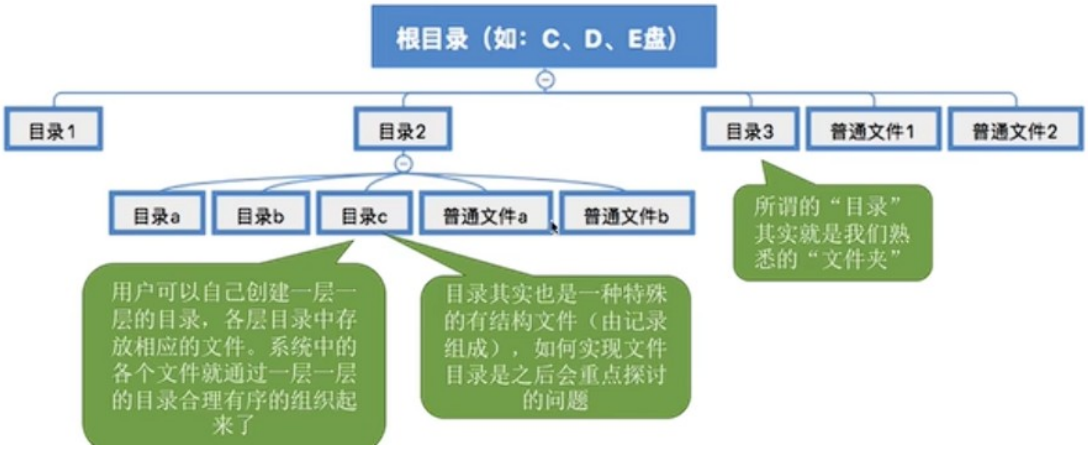
3、 类型：文件的类型

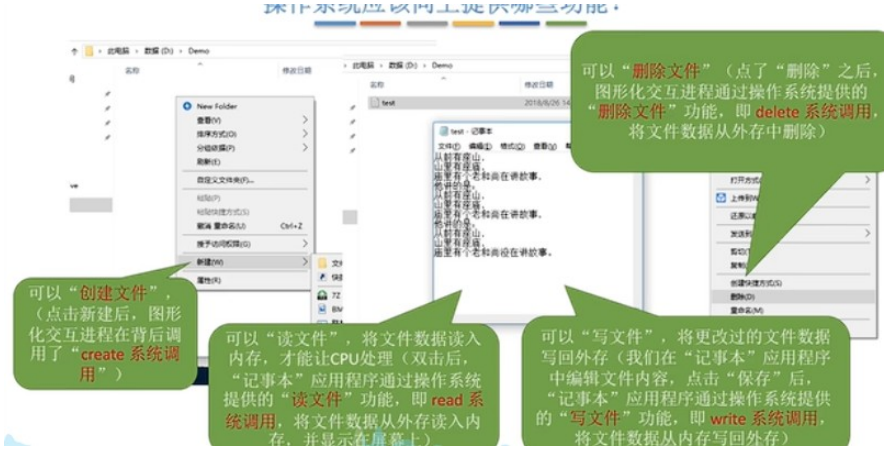
4、 位置：存放的路径

无结构文件：由一些二进制或字符流组成，又称“流式文件“

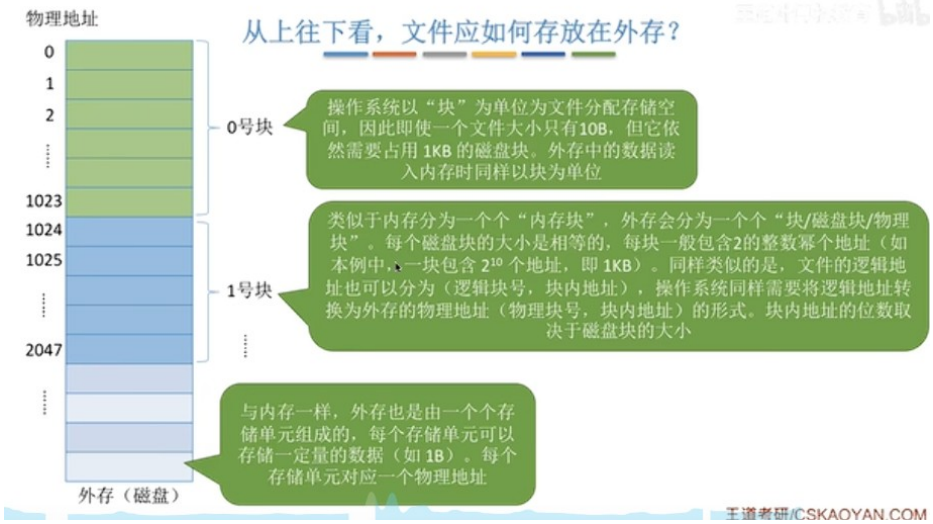
有结构文件：由一个个记录组成

文件之间应该怎样组织起来?




操作系统应该向上提供哪些功能？
可以“删除文件”（点了“删除”之后，图形化交互进程通过操作系统提供的“删除文件”功能，即 delete 系统调用，将文件数据从外存中删除）
可以“创建文件”，（点击新建后，图形化交互进程在背后调用了“create 系统调用”）
可以“读文件”，将文件数据读入内存，才能让CPU处理（双击后，“记事本”应用程序通过操作系统提供的“读文件”功能，即 read 系统调用，将文件数据从外存读入内存，并显示在屏幕上）
可以“写文件”，将更改过的文件数据写回外存（我们在“记事本”应用程序中编辑文件内容，点击“保存”后，“记事本”应用程序通过操作系统提供的“写文件”功能，即 write 系统调用，将文件数据从内存写回外存）


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）


物理地址
从上往下看，文件应如何存放在外存？
0
1
2
1023
1024
1025
2047
0号块
1号块
操作系统以“块”为单位为文件分配存储空间，因此即使一个文件大小只有10B，但它依然需要占用 1KB 的磁盘块。外存中的数据读入内存时同样以块为单位
类似于内存分为一个个“内存块”，外存会分为一个个“块/磁盘块/物理块”。每个磁盘块的大小是相等的，每块一般包含2的整数幂个地址（如本例中，一块包含 2^{10} 个地址，即 1KB）。同样类似的是，文件的逻辑地址也可以分为（逻辑块号，块内地址），操作系统同样需要将逻辑地址转换为外存的物理地址（物理块号，块内地址）的形式。块内地址的位数取决于磁盘块的大小
与内存一样，外存也是由一个个存储单元组成的，每个存储单元可以存储一定量的数据（如 1B）。每个存储单元对应一个物理地址
外存（磁盘）
王道考研/CSKAOYAN.COM

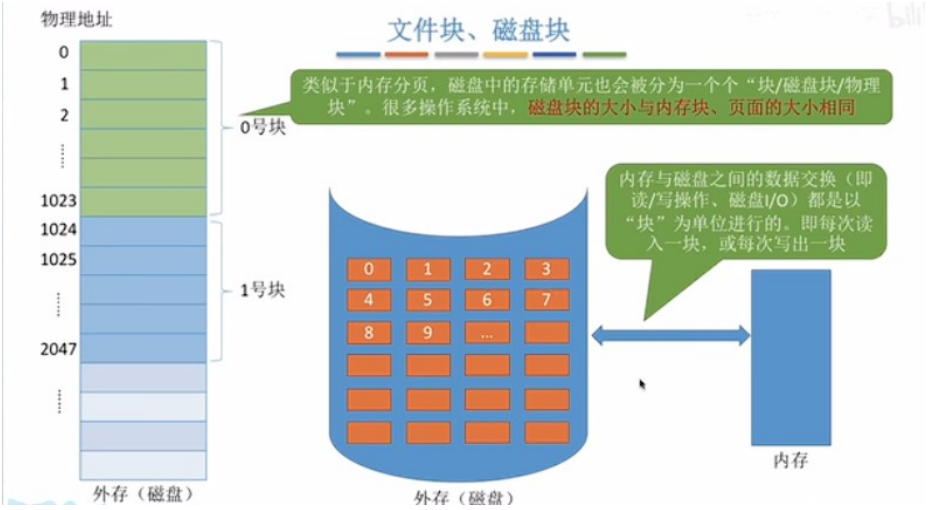
文件的逻辑结构

逻辑上可以相邻，物理结构可以不相邻

无结构文件

文件的物理结构

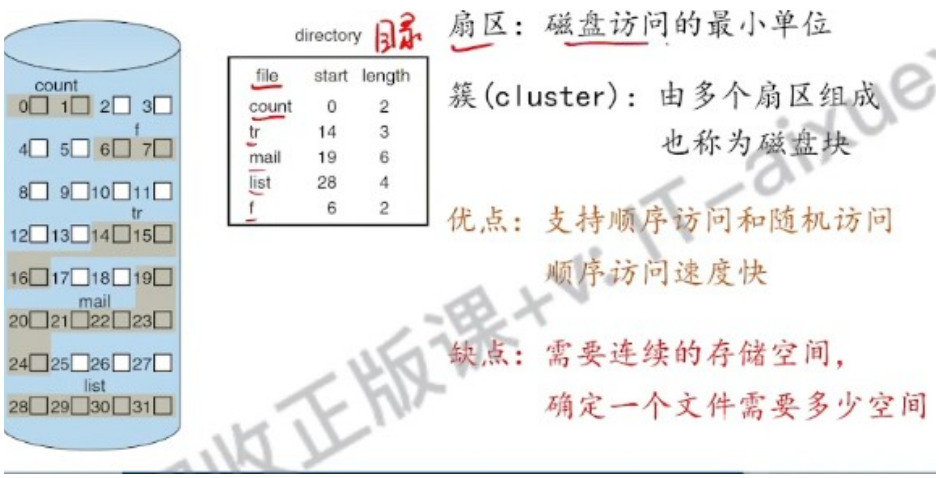
文件最后时存储在磁盘上，要读到内存里
磁盘分配空间
连续分配
链接分配
索引分配
连续分配
连续文件又称为顺序文件，它把逻辑文件中的信息顺序地存放到一组相邻接的磁盘块
内存与外存以块位单位传递数据

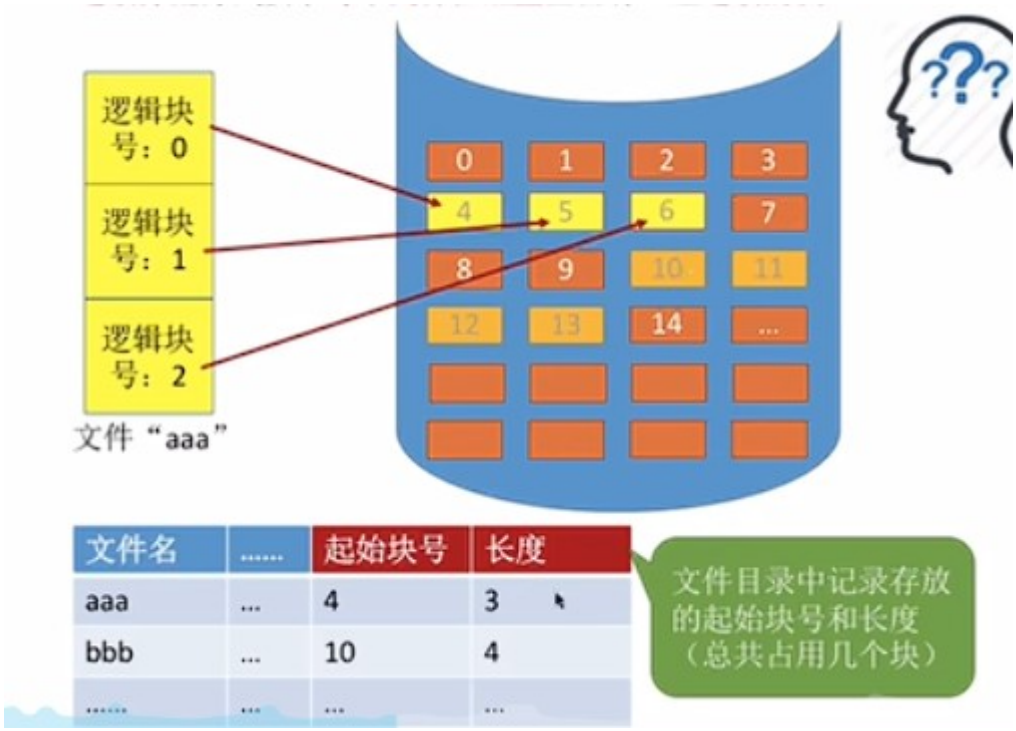


文件的逻辑地址也可以表示为（逻辑块号，块内地址）的形式

连续分配

要实现逻辑地址到物理地址的映射

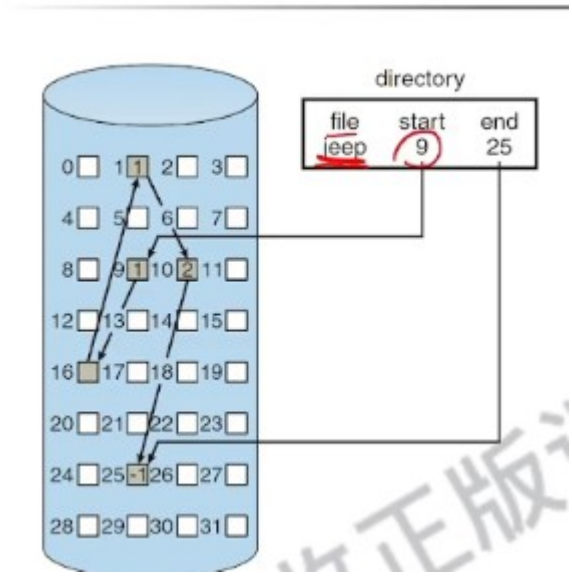




CD-ROM

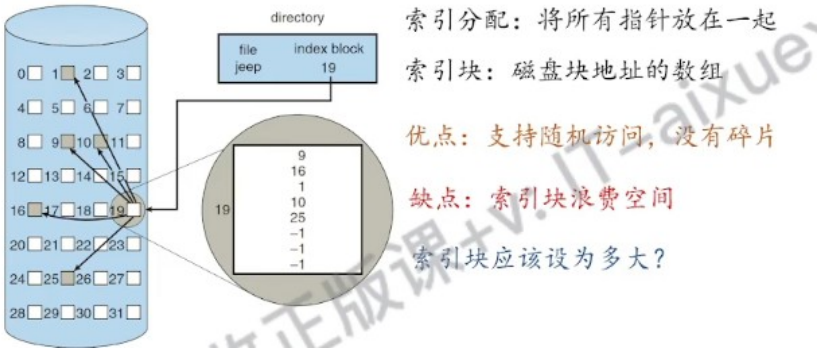
扇区：512B

链式分配



索引分配

隐式链接

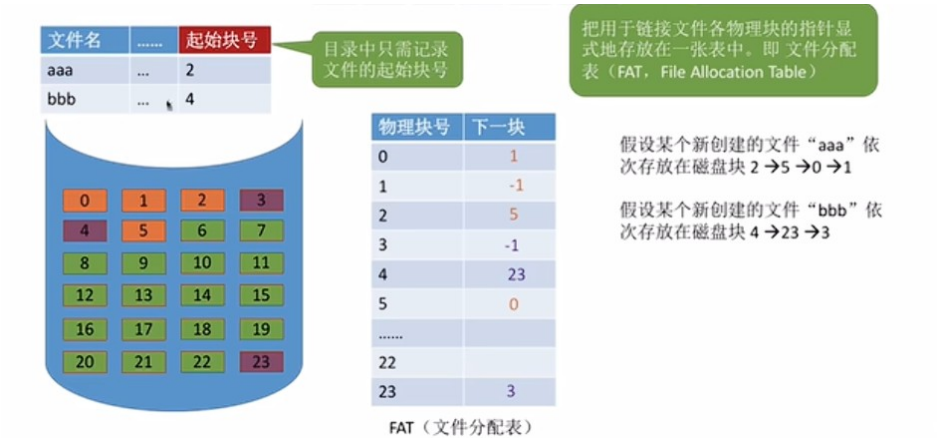


显式链接

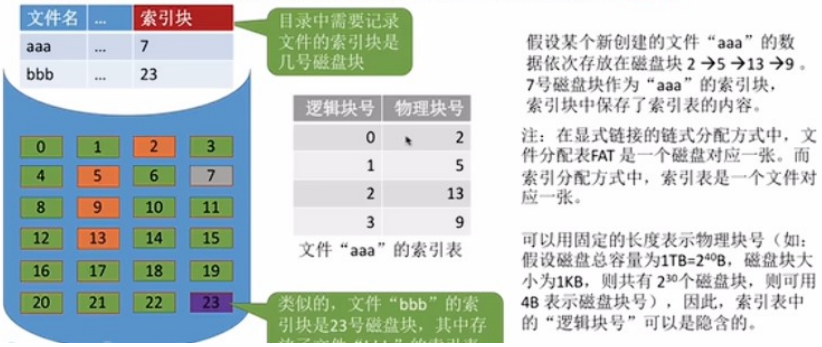
把物理块的指针显示地存放在一张表中，即文件分配表

一个磁盘仅设置一张 FAT，常驻内存

支持随机访问
不需要访问磁盘

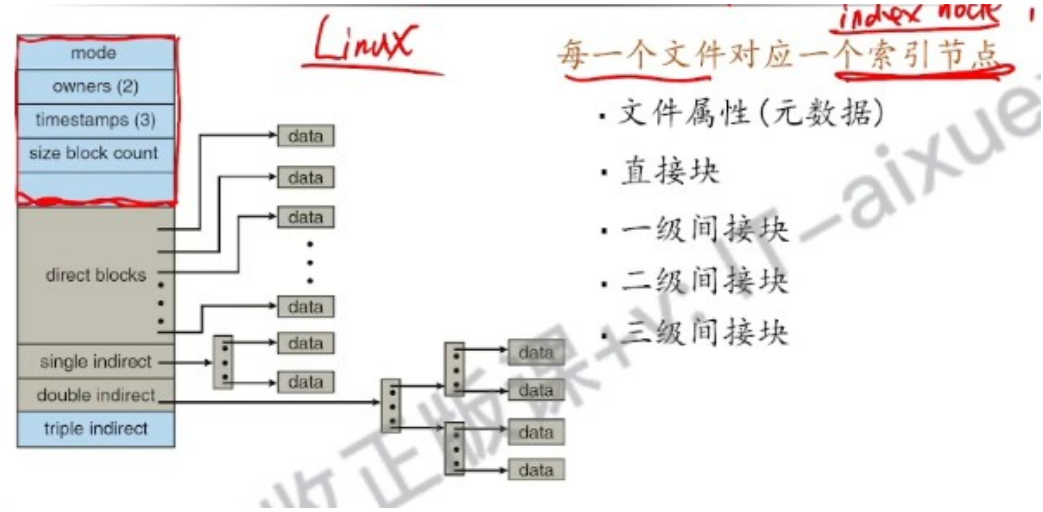


索引分配允许文件离散地分配在各个磁盘块中，系统会为每个文件建立一张索引表，索引表中记录了文件的各个逻辑块对应的物理块（索引表的功能类似于内存管理中的页表——建立逻辑页面到物理页之间的映射关系）。索引表存放的磁盘块称为索引块。文件数据存放的磁盘块称为数据块。



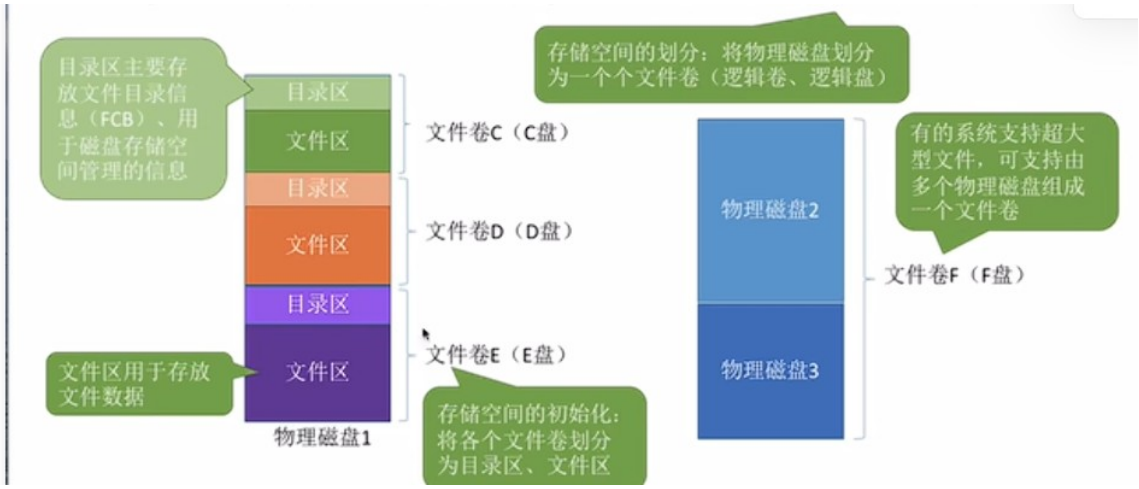
指针：磁盘块地址
解决方案：1 链接方案：如果索引表太大，那么可以将多个索引块链接起来存放
2、

多级索引

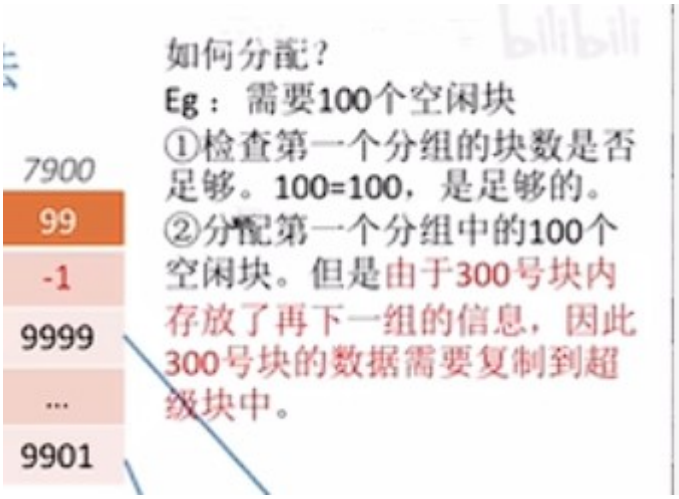


建立多层索引，类似于多级页表，使第一层索引块指向第二层的索引块。

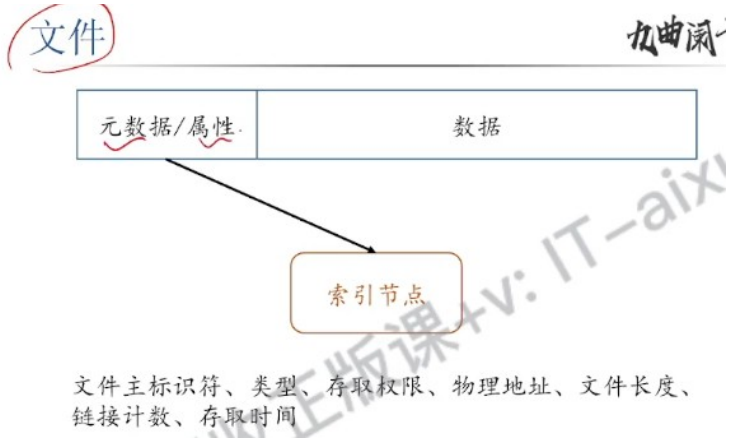
混合索引



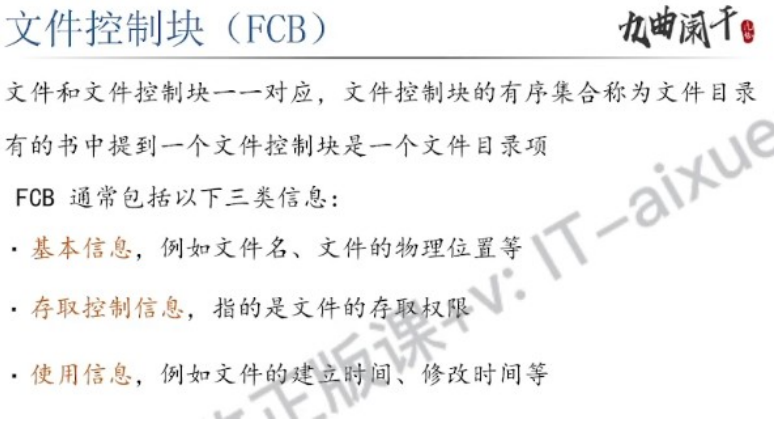
成组链接法



文件

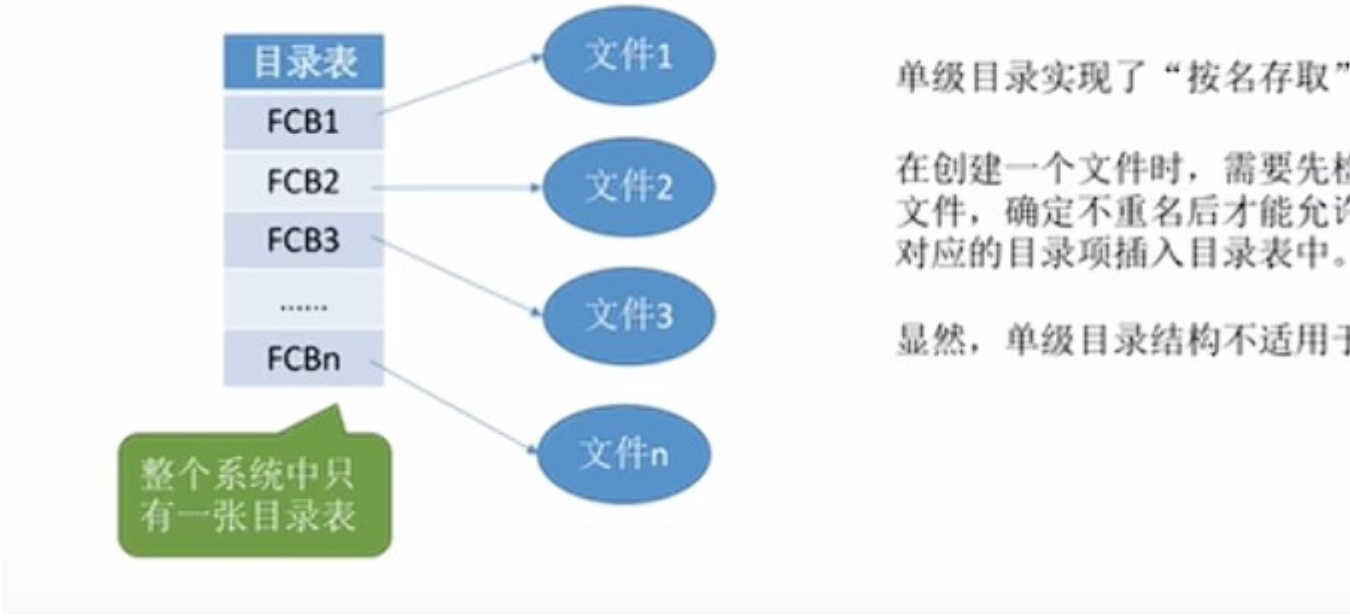


文件控制块 FCB



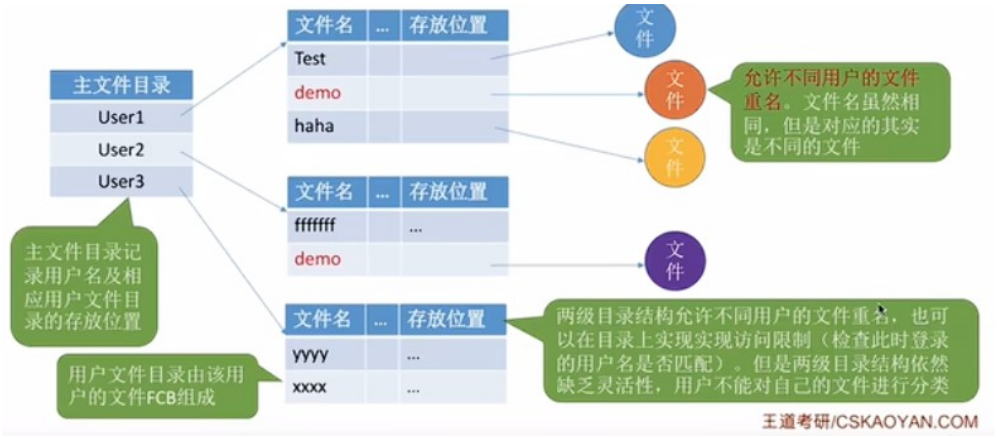
FCB 的有序集合称为文件目录（目录文件），一个 FCB 就是一个文件目录项
目录文件中的一条记录就是一个 FCB
也叫做文件控制块

单级目录结构



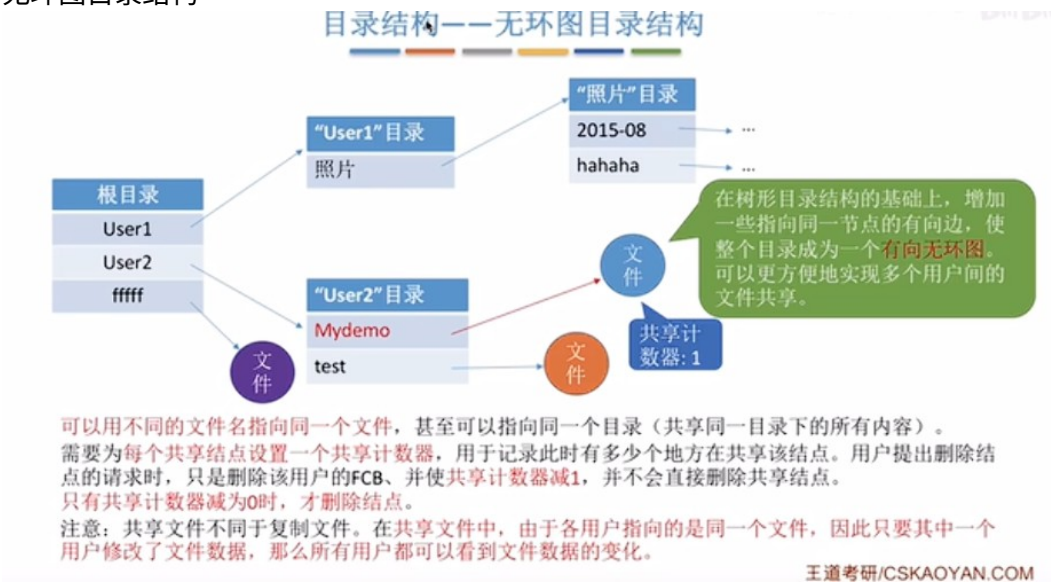
两级目录结构

MFD 主文件目录
UFD 用户文件目录

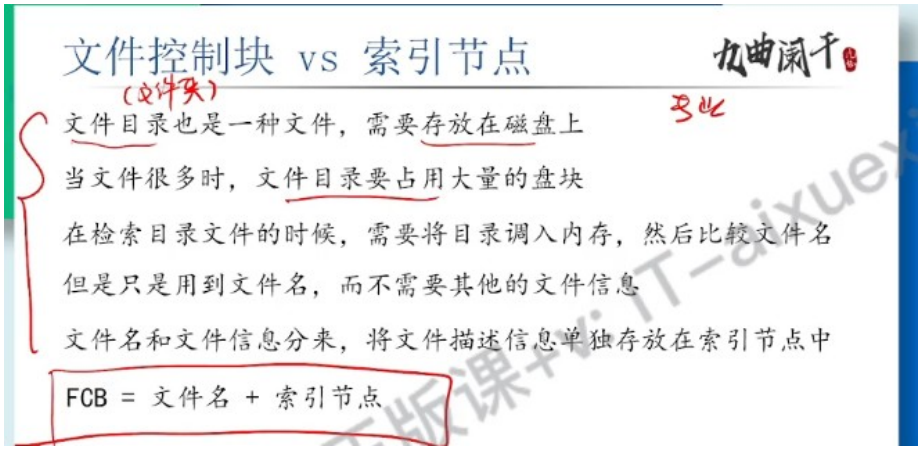


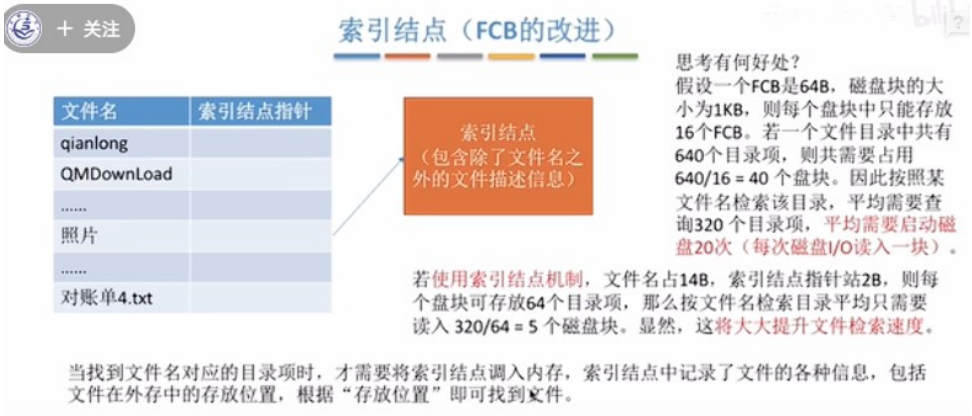
多级目录，又称树形目录

要访问某个文件时要用文件路径名标识文件，文件路径名是个字符串
无环图目录结构



索引节点


文件控制块 vs 索引节点
文件目录也是一种文件，需要存放在磁盘上
当文件很多时，文件目录要占用大量的盘块
在检索目录文件的时候，需要将目录调入内存，然后比较文件名
但是只是用到文件名，而不需要其他的文件信息
文件名和文件信息分来，将文件描述信息单独存放在索引节点中
FCB = 文件名 + 索引节点


+ 关注
索引结点（FCB的改进）
文件名
索引结点指针
qianlong
QMDownLoad
.....
照片
.....
对账单4.txt
索引结点
（包含除了文件名之外的文件描述信息）
思考有何好处？
假设一个FCB是64B，磁盘块的大小为1KB，则每个盘块中只能存放16个FCB。若一个文件目录中共有640个目录项，则共需要占用640/16 = 40 个盘块。因此按照某文件名检索该目录，平均需要查询320 个目录项，平均需要启动磁盘20次（每次磁盘I/O读入一块）。
若使用索引结点机制，文件名占14B，索引结点指针站2B，则每个盘块可存放64个目录项，那么按文件名检索目录平均只需要读入 320/64 = 5 个磁盘块。显然，这将大大提升文件检索速度。
当找到文件名对应的目录项时，才需要将索引结点调入内存，索引结点中记录了文件的各种信息，包括文件在外存中的存放位置，根据“存放位置”即可找到文件。

文件操作

- 创建文件（creat open+特定标记）
- 读文件（read）
- 写文件（write）
- 删除文件（unlink）
- 截断文件（truncate）

word excle txt touch → Disk 设备

创建文件

创建文件

Unix

创建一个文件，可以通过系统调用open实现 creat

```
int fd = open("path", O_CREAT | O_WRONLY | O_TRUNC);
```

系统调用open返回值 —— 文件描述符(file descriptor)

- 文件描述符是一个整数，是每个进程私有的
- 文件描述符可以理解为一种权限，允许执行某些操作
- 将文件描述符看成指向文件类型对象的指针
- 每一个进程通过一个文件描述符表记录打开的文件

对文件的读写通过文件描述符来访问 fd8 read("

```
int open(char *filename, int flags, mode_t mode);
ssize_t read(int fd, void *buf, size_t n);
ssize_t write(int fd, const void *buf, size_t n);
int close(int fd);
```

关闭后，该文件描述符从文件描述符表中删除

目录文件——文件夹，目录文件的位置

文件系统通常使用目录(文件夹)记录文件的位置

目录包含一个文件名的列表，每个文件名对应一个inode编号

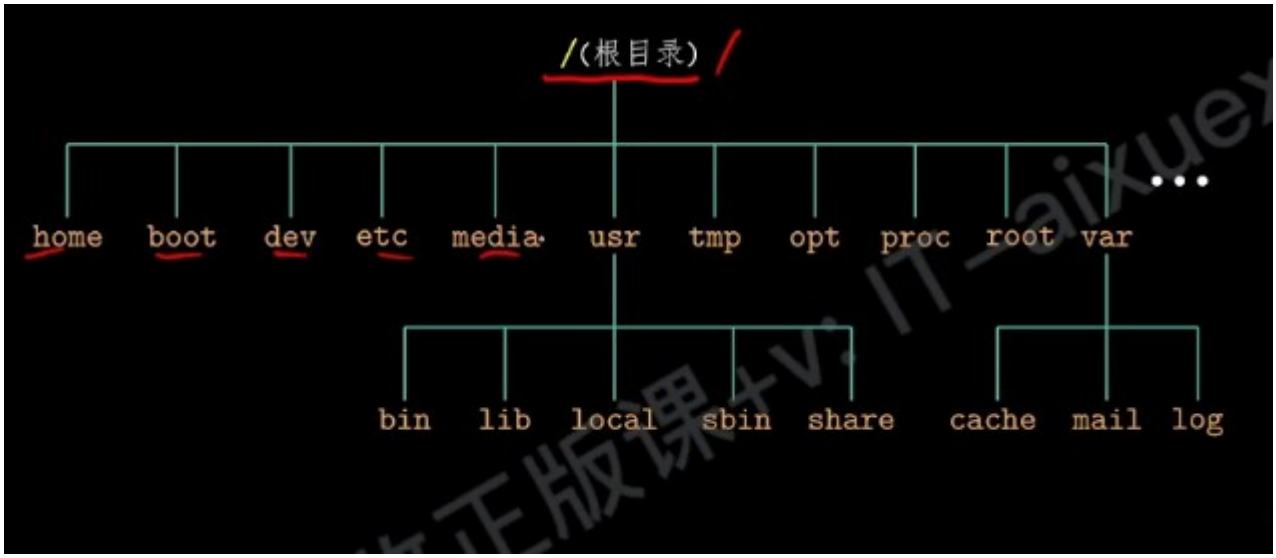
每个文件名称为目录项，每个名字到inode的映射称为链接

```
struct dirent {
    ino_t d_ino;       /* inode number */
    char  d_name[256]; /* Filename */
};
```

指向一个索引节点

FCB=文件名+索引编号

文件目录结构



- 创建目录
- 删除目录
- 打开目录和关闭目录
- 读目录项
- 文件链接

Linux系统编程

目录中每个名字到索引节点的映射称为链接

链接的本质就是目录中一个指向索引节点(inode)的名字

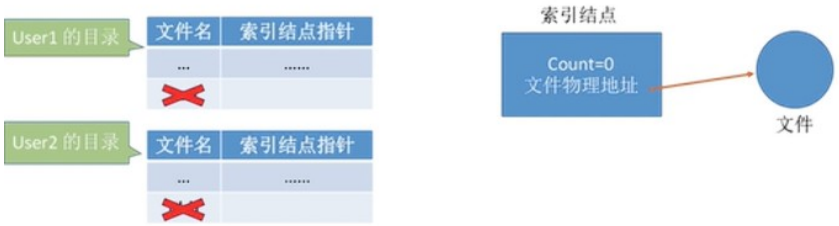
所有的链接中，没有一个链接是“原始”或者“初始”状态

硬链接

```
> ln  /oldpath  /newpath
```

```
int link(const char *oldpath, const char*newpath);
```

知识回顾：索引结点，是一种文件目录瘦身策略。由于检索文件时只需用到文件名，因此可以将除了文件名之外的其他信息放到索引结点中。这样目录项就只需要包含文件名、索引结点指针。



索引结点中设置一个链接计数变量 count，用于表示链接到本索引结点上的用户目录项数。
若 count = 2，说明此时有两个用户目录项链接到该索引结点上，或者说是有两个用户在共享此文件。
若某个用户决定“删除”该文件，则只是要把用户目录中与该文件对应的目录项删除，且索引结点的 count值减 1。
若 count>0，说明还有别的用户要使用该文件，暂时不能把文件数据删除，否则会导致指针悬空。
当 count = 0 时系统负责删除文件。

同一个索引节点

共享文件

Copy 和硬链接的区别?
Copy 是两份
硬链接只有一份，映射了同一索引节点
软链接（符号链接）

```
> ln -s /oldpath  /newpath
```

符号链接本身是一个不同类型的文件

普通文件、目录文件、符号链接

快捷方式

当前工作目录

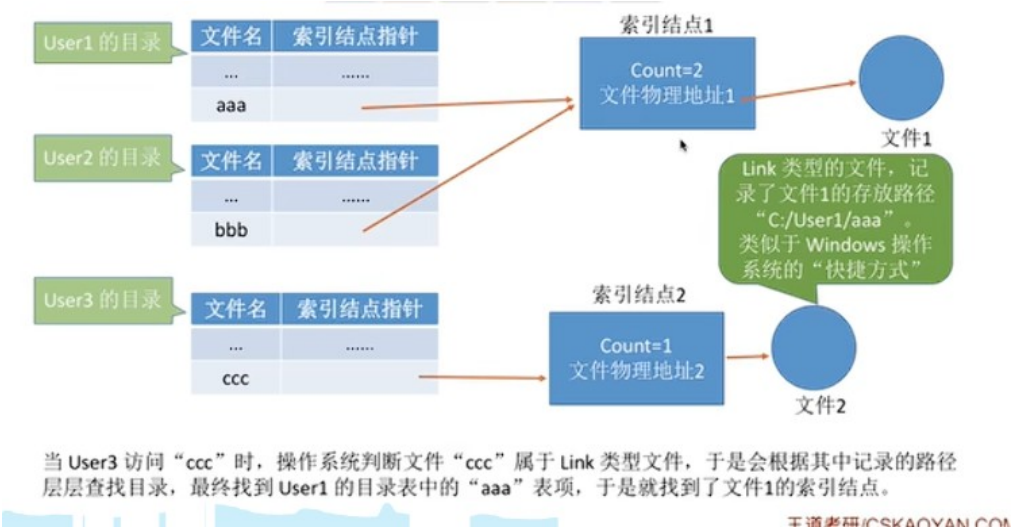
每个进程都有一个当前目录，一般是创建时从父进程继承的

内核解析相对路径时，会把当前工作目录作为起点

获得当前工作目录使用系统调用getcwd

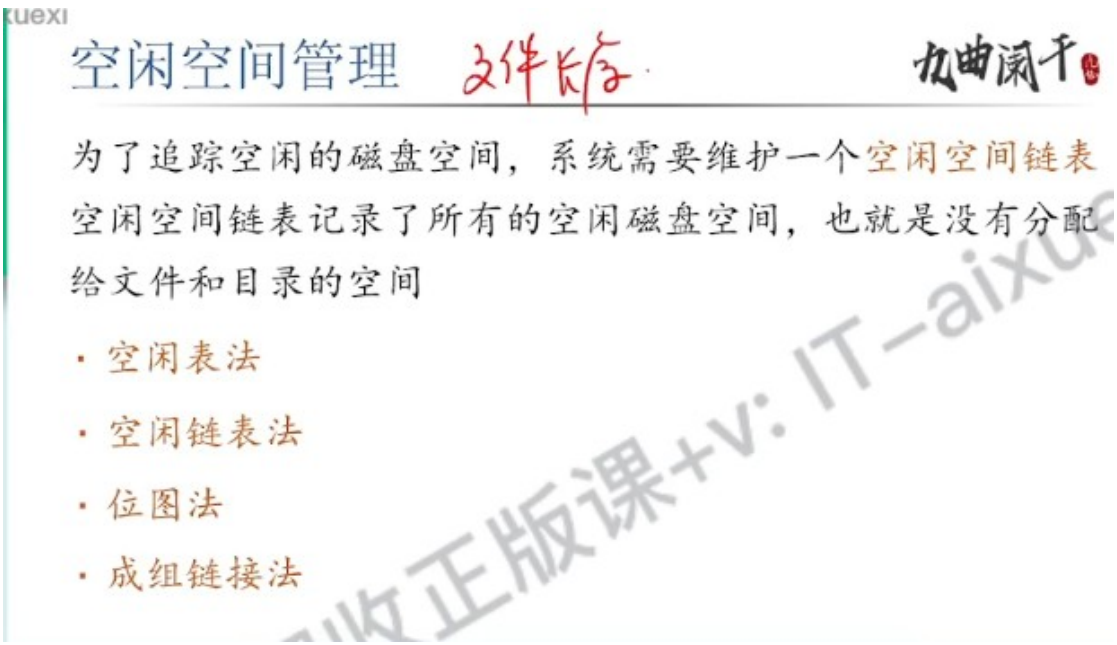
```
char *getcwd(char *buf, size_t size);
```

当前工作目录也可以修改



当 User3 访问“ccc”时，操作系统判断文件“ccc”属于 Link 类型文件，于是会根据其中记录的路径层层查找目录，最终找到 User1 的目录表中的“aaa”表项，于是就找到了文件1的索引结点。

存储空间管理



设备管理

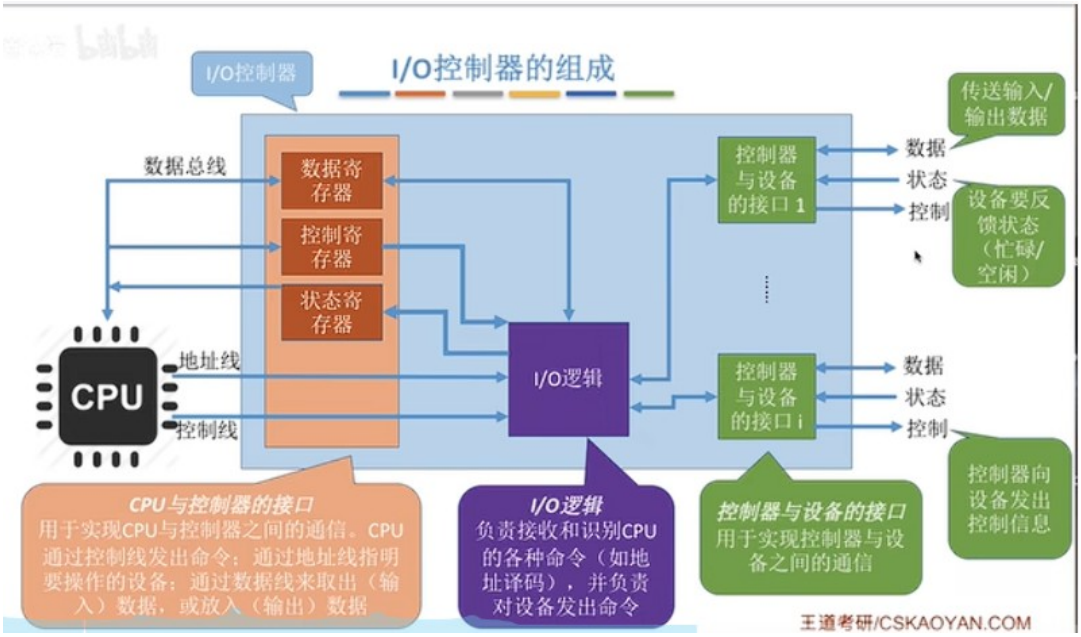
设备的分类

1、 按数据传输速率
低速：鼠标、键盘
中速：打印机、扫描仪
高速：磁盘

2、 按信息交换单位
块设备：磁盘 可寻址
字符设备：不可寻址 中断驱动 交互式终端
3、 按设备共享属性
4、 按工作特性
存储设备
I/O 设备
网络通信设备

I/O 控制器/设备控制器

接受和识别命令
数据交换
地址识别
数据缓冲
识别和报告设备状态
差错控制

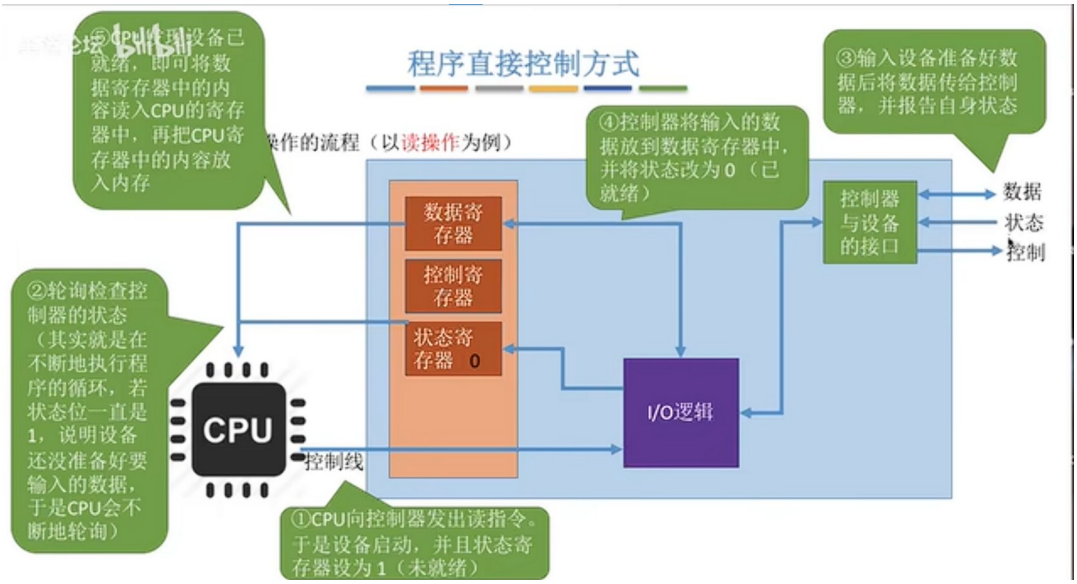


1、 一个 I/O 控制器可能会对应多个设备
2、 内存映像 I/O/寄存器独立编址

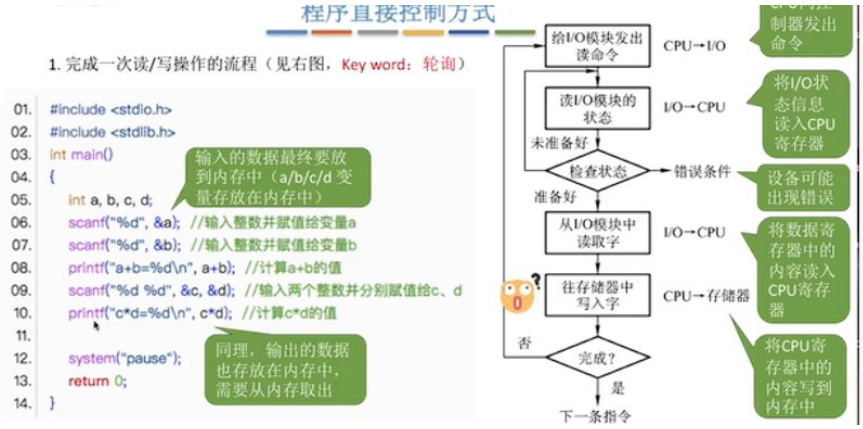
I/O 控制方式

1、 直接控制方式
2、 中断驱动控制方式
3、 直接存储器存取

直接控制

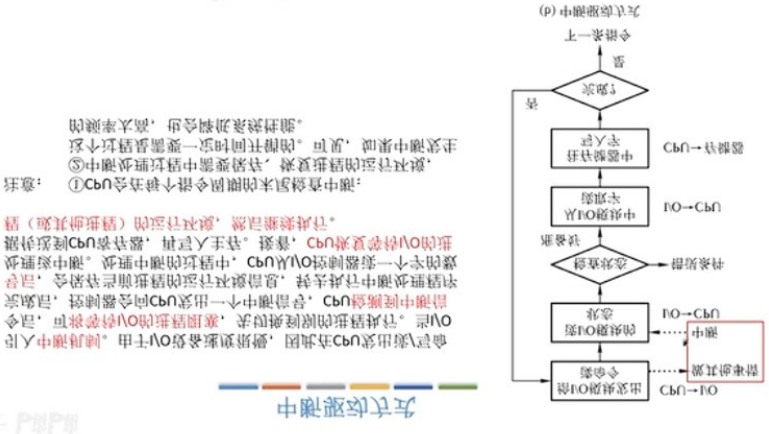


1、先经过 CPU，再到内存
2、CPU 干预的频率
3、每次读写一个字
4、缺点：CPU/I/O 设备只能串行工作，CPU 一直处于轮询状态

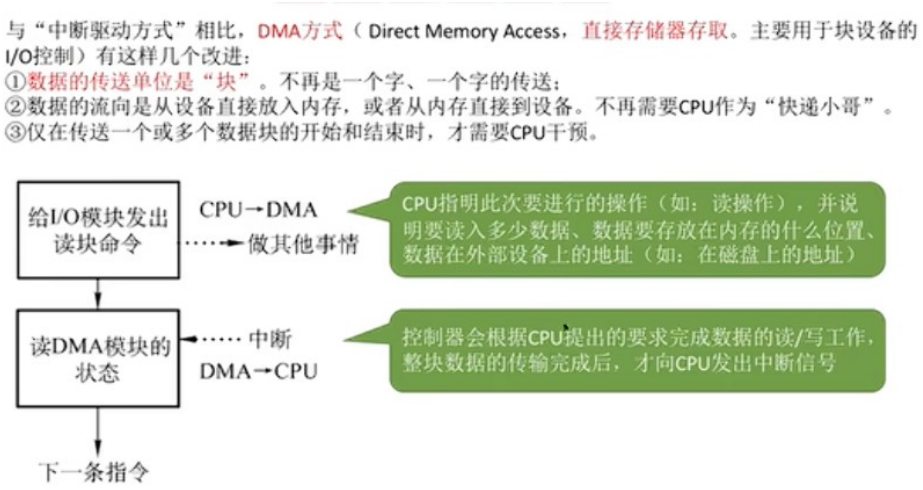


中断驱动方式

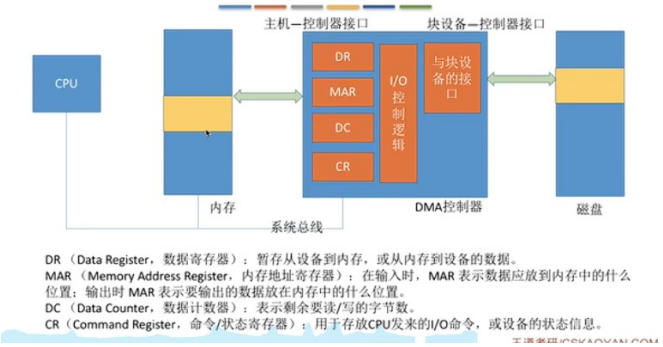
由于 I/O 设备速度很慢，可将 I/O 等待的进程阻塞，现金切换到别的进程执行



DMA 方式



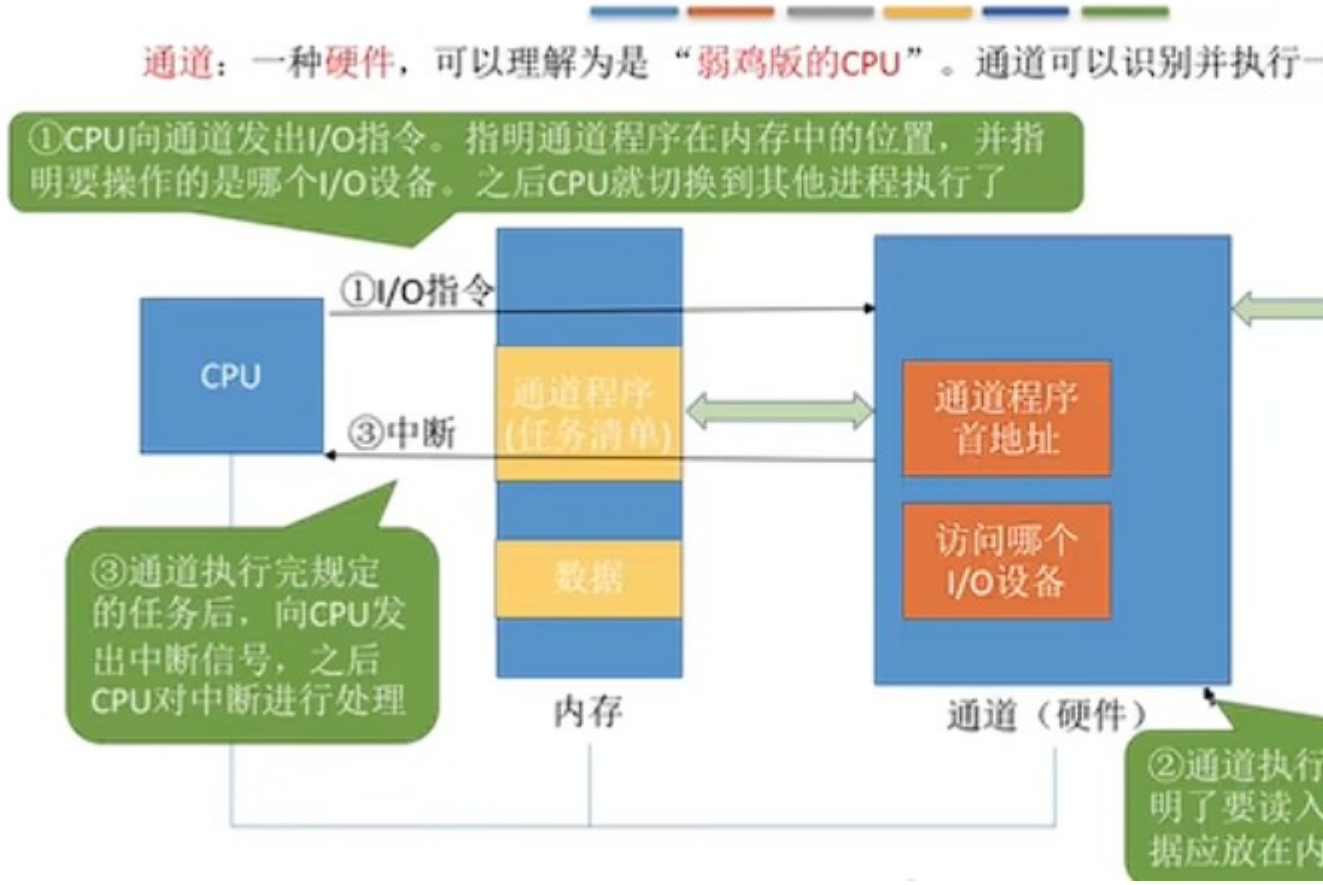
读取的时候是按字读取，然后放入内存以块为单位
仅在数据传送开始和结束时 caixuCPU 干预



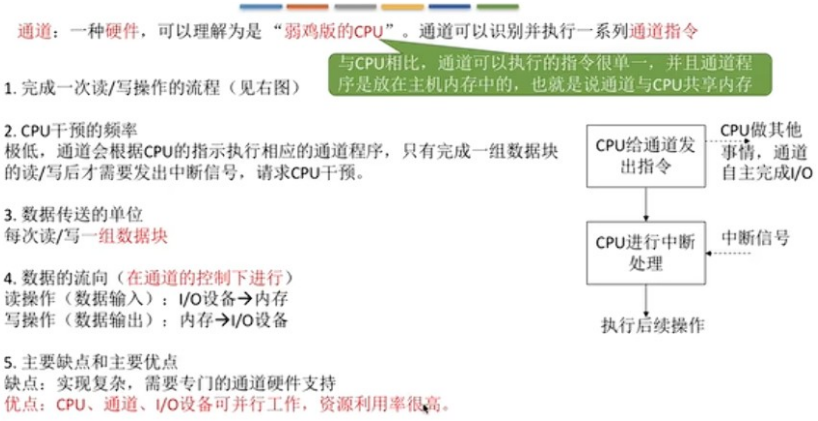
缺点：只能处理离散的

通道控制方式：

一种硬件，识别并执行一系列通道指令


通道：一种硬件，可以理解为是“弱鸡版的CPU”。通道可以识别并执行
①CPU向通道发出I/O指令。指明通道程序在内存中的位置，并指明要操作的是哪个I/O设备。之后CPU就切换到其他进程执行了
①I/O指令
CPU
③中断
通道程序
(任务清单)
数据
内存
通道程序首地址
访问哪个I/O设备
通道（硬件）
③通道执行完规定的任务后，向CPU发出中断信号，之后CPU对中断进行处理
②通道执行
明了要读入
据应放在内

通道与 CPU 共享内存


通道：一种硬件，可以理解为是“弱鸡版的CPU”。通道可以识别并执行一系列通道指令
与CPU相比，通道可以执行的指令很单一，并且通道程序是放在主机内存中的，也就是说通道与CPU共享内存
1. 完成一次读/写操作的流程（见右图）
2. CPU干预的频率
极低，通道会根据CPU的指示执行相应的通道程序，只有完成一组数据块的读/写后才需要发出中断信号，请求CPU干预。
3. 数据传送的单位
每次读/写一组数据块
4. 数据的流向（在通道的控制下进行）
读操作（数据输入）：I/O设备→内存
写操作（数据输出）：内存→I/O设备
5. 主要缺点和主要优点
缺点：实现复杂，需要专门的通道硬件支持
优点：CPU、通道、I/O设备可并行工作，资源利用率很高。
CPU给通道发出指令
CPU做其他事情，通道自主完成I/O
CPU进行中断处理
中断信号
执行后续操作

	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	低	一组块	设备→内存 内存→设备	

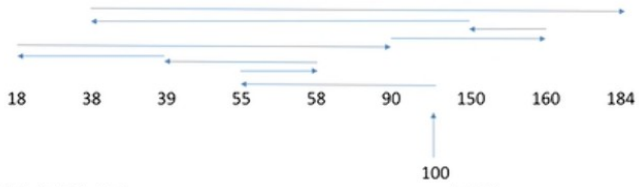
磁盘调度算法

FCFS

根据进程请求访问磁盘的先后顺序进行调度。

假设磁头的初始位置是100号磁道，有多个进程先后陆续地请求访问 55、58、39、18、90、160、150、38、184 号磁道

按照 FCFS 的规则，按照请求到达的顺序，磁头需要依次移动到 55、58、39、18、90、160、150、38、184 号磁道



磁头总共移动了 45+3+19+21+72+70+10+112+146 = 498 个磁道

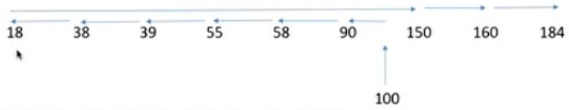
响应一个请求平均需要移动 498/9 = 55.3 个磁道（平均寻找长度）

优点：公平；如果请求访问的磁道比较集中的话，算法性能还算过的去

缺点：如果有大量进程竞争使用磁盘，请求访问的磁道很分散，则FCFS在性能上很差，寻道时间长。

SSTF 算法会优先处理的磁道是与当前磁头最近的磁道。可以保证每次的寻道时间最短，但是并不能保证总的寻道时间最短。（其实就是贪心算法的思想，只是选择眼前最优，但是总体未必最优）

假设磁头的初始位置是100号磁道，有多个进程先后陆续地请求访问 55、58、39、18、90、160、150、38、184 号磁道



磁头总共移动了 (100-18) + (184-18) = 248 个磁道
响应一个请求平均需要移动 248/9 = 27.5 个磁道（平均寻找长度）
优点：性能较好，平均寻道时间短
缺点：可能产生“饥饿”现象
Eg：本例中，如果在处理18号磁道的访问请求时又来了一个38号磁道的访问请求，处理38号磁道的访问请求时又来了一个18号磁道的访问请求。如果有源源不断的 18号、38号磁道的访问请求到来的话，150、160、184 号磁道的访问请求就永远得不到满足，从而产生“饥饿”现象。

扫描算法（SCAN）

SSTF 算法会产生饥饿的原因在于：磁头有可能在一个小区域内来回来去地移动。为了防止这个问题，可以规定，只有磁头移动到最外侧磁道的时候才能往内移动，移动到最内侧磁道的时候才能往外移动。这就是扫描算法（SCAN）的思想。由于磁头移动的方式很像电梯，因此也叫电梯算法。

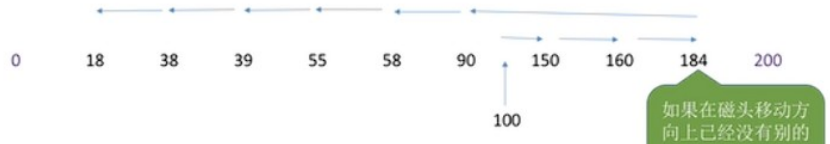
假设某磁盘的磁道为 0~200号，磁头的初始位置是100号磁道，且此时磁头正在往磁道号增大的方向移动，有多个进程先后陆续地请求访问 55、58、39、18、90、160、150、38、184 号磁道

磁头总共移动了 (200-100) + (200-18) = 282 个磁道
响应一个请求平均需要移动 282/9 = 31.3 个磁道（平均寻找长度）
优点：性能较好，平均寻道时间较短，不会产生饥饿现象
缺点：①只有到达最边上的磁道时才能改变磁头移动方向，事实上，处理了184号磁道的访问请求之后就不需要再往右移动磁头了。
②SCAN算法对于各个位置磁道的响应频率不平均（如：假设此时磁头正在往右移动，且刚处理过90号磁道，那么下次处理90号磁道的请求就需要等磁头移动很长一段距离；而响应了184号磁道的请求之后，很快又可以再次响应 184 号磁道的请求了）

注意接下来的 LOOK 调度算法才是教材里的 SCAN 算法

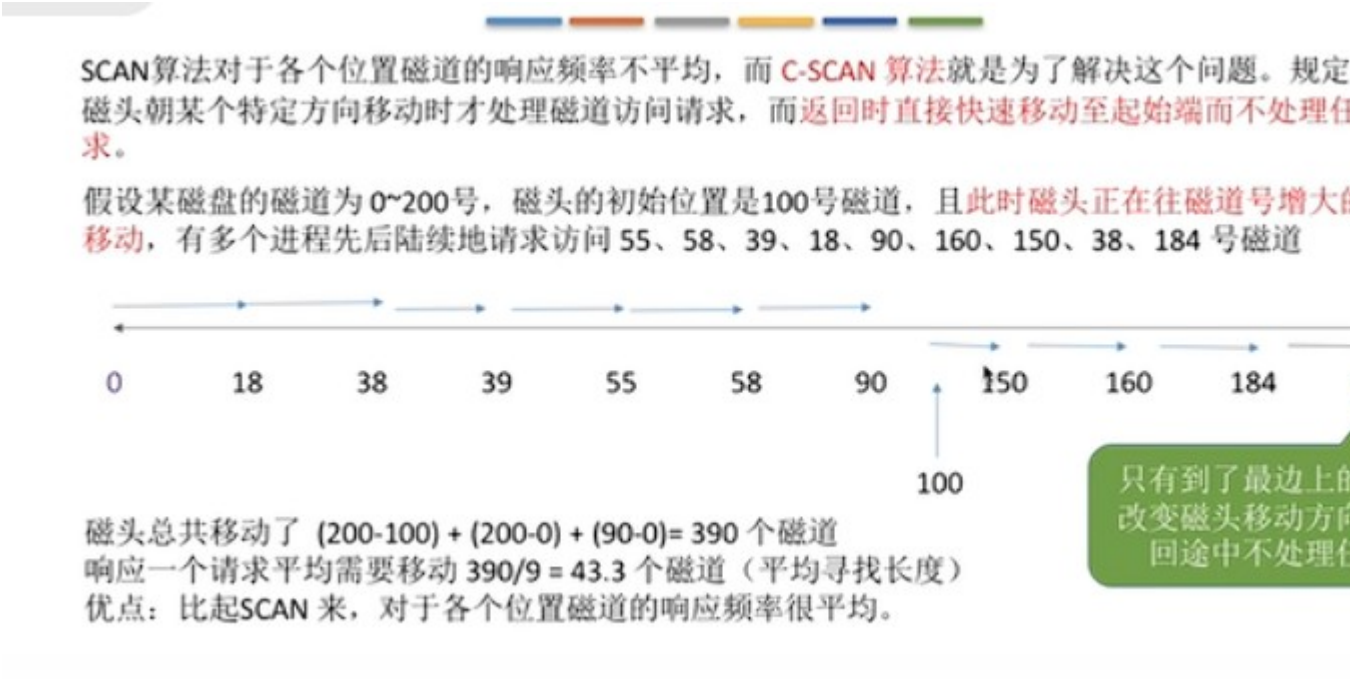
扫描算法（SCAN）中，只有到达最边上的磁道时才能改变磁头移动方向，事实上，处理了184号磁道的访问请求之后就不需要再往右移动磁头了。LOOK 调度算法就是为了解决这个问题，如果在磁头移动方向上已经没有别的请求，就可以立即改变磁头移动方向。（边移动边观察，因此叫 LOOK）

假设某磁盘的磁道为 0~200号，磁头的初始位置是100号磁道，且此时磁头正在往磁道号增大的方向移动，有多个进程先后陆续地请求访问 55、58、39、18、90、160、150、38、184 号磁道

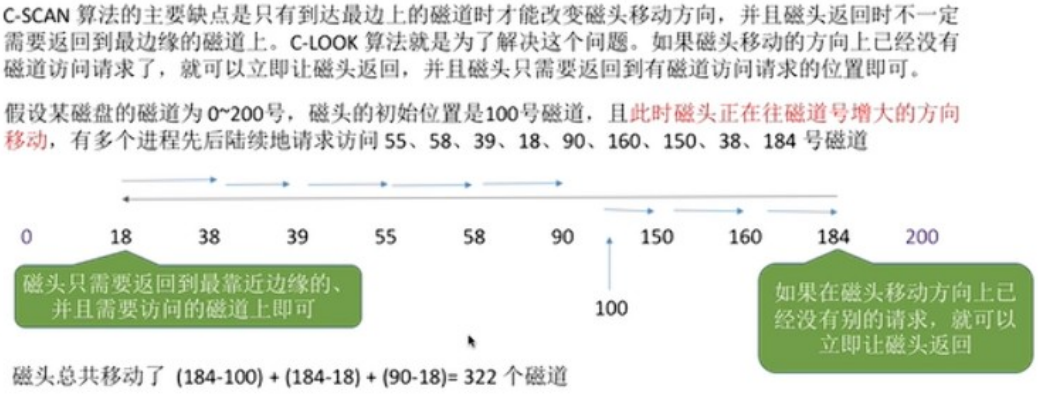


磁头总共移动了 (184-100) + (184-18) = 250 个磁道
响应一个请求平均需要移动 250/9 = 27.5 个磁道（平均寻找长度）
优点：比起 SCAN 算法来，不需要每次都移动到最外侧或最内侧才改变磁头方向，使寻道时间进一步缩短

CSCAN 算法



注意接下来的 C-LOOK 调度算法才是我们教材中的的 CSCAN 算法



假脱机技术：用 ru
脱离主机的控制进行的输入/输出操作
在磁盘上开辟两个存储区域——输入井和输出井
外围控制机