**操作系统**

**1.1.1操作系统基本概念，功能**

1.操作系统（OS）是指控制和管理整个计算机系统的硬件和软件资源，并合理地组织调度计算机的工作和资源的分配：以提供给用户和其他软件方便的接口和环境：它是计算机系统中最基本的系统软件。

含义：（1）操作系统是系统资源的管理者

（2）向上层提供方便易用的服务

（3）是最接近硬件的一层软件

2.操作系统的功能和目标

操作系统作为系统资源的管理者，应该具有以下几方面的功能：处理机管理、存储器管理、文件管理、设备管理。为了方便用户使用操作系统，还必须向用户提供接口。同时，操作系统可用来扩充机器，以提供方便的服务、更高的资源利用率。

操作系统要向上层提供方便易用的服务。硬件只能明白二进制指令，对外暴露的是丑陋不友好的交互接口。但在硬件之上安装了操作系统，操作系统对外暴露了美丽友好的交互接口。这体现了“封装”的思想：操作系统把一些丑陋的硬件功能封装成简单易用的服务，使用户能更方便地使用计算机，用户无需关心底层硬件的原理，只需要对操作系统发出命令即可。

服务的例子：

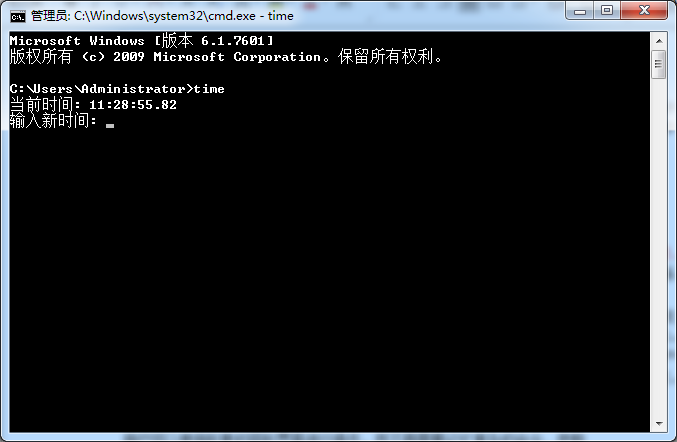
（1）GUI：图形化用户接口

用户可以使用形象的图形界面进行操作，而不再需要记忆复杂的命令、参数

例子：在Windows操作系统中，删除一个文件只需要把文件“拖拽”到回收站即可。

（2）联机命令接口=交互式命令接口

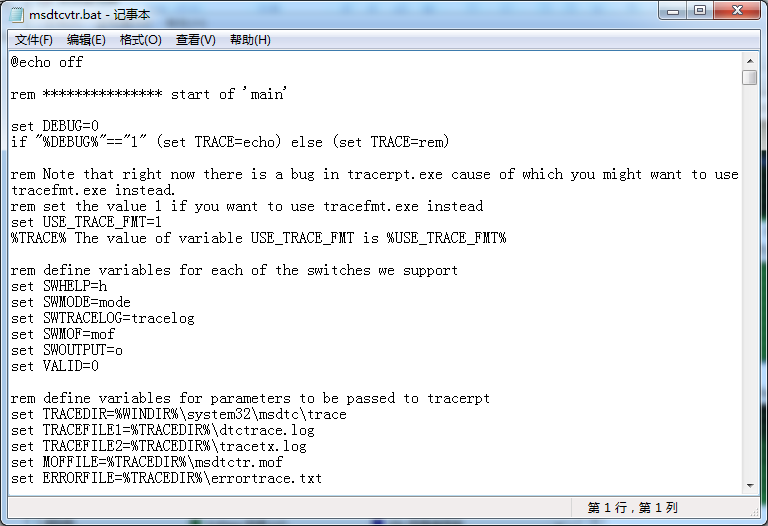
用win+R键打开“运行”，然后输入cmd，按回车，打开命令解释器，可以在命令行窗口里使用各种各样的命令。例如使用time命令：



其特点是用户说一句，系统跟着做一句

（3）脱机命令接口=批处理命令接口

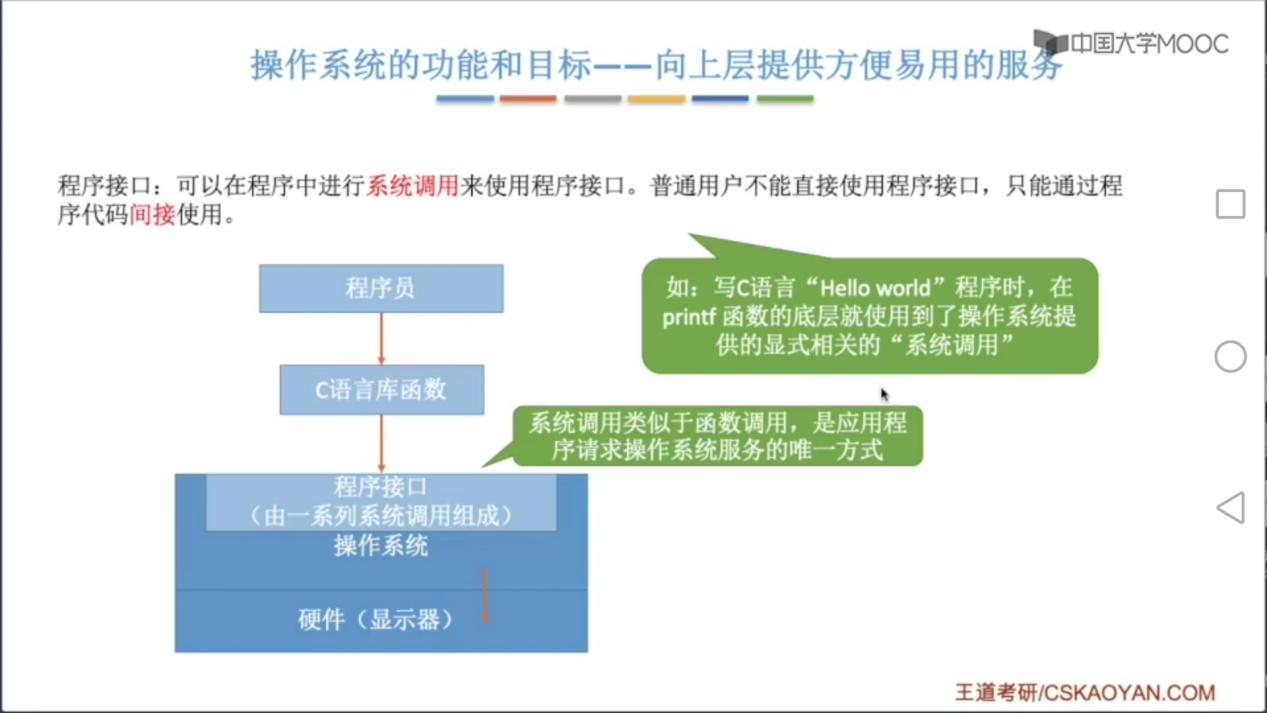
使用windows系统的搜索功能，搜索C盘中的\*.bat文件，用记事本打开任意一个



特点是用户说一堆，系统跟着做一堆

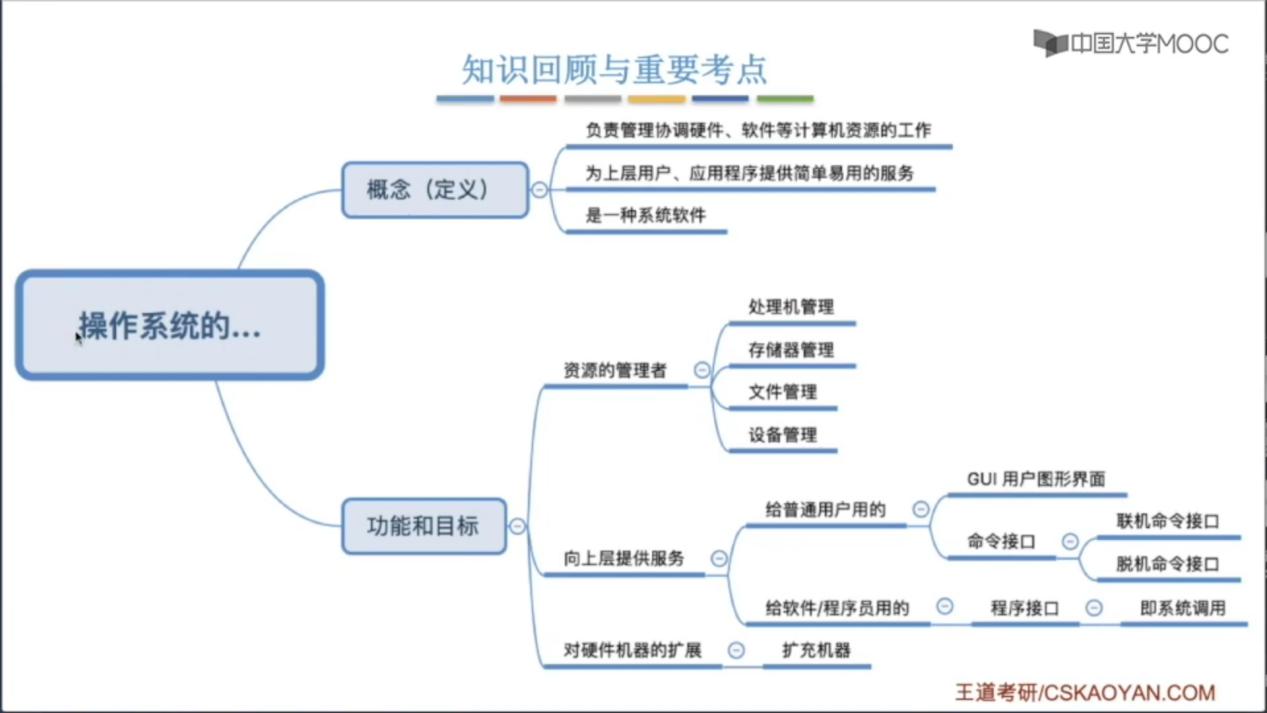
（4）程序接口：可以在程序中进行系统调用来使用程序接口。普通用户不能直接使用程序接口，只能通过程序代码间接使用。

例子：写C语言语句printf(“%s”,”Hello world”)，在printf函数的底层就使用到了操作系统提供的显示相关的“系统调用”



操作系统作为最接近硬件的层次，需要实现对硬件机器的拓展。没有任何软件支持的计算机称为裸机。在裸机上安装的操作系统，可以提供资源管理功能和方便用户的服务功能，将裸机改造成功能更强、使用更方便的机器。通常把覆盖了软件的机器称为扩充机器，又称为之虚拟机。操作系统将CPU、内存、磁盘、显示器、键盘等硬件合理地组织起来，让各种硬件能够相互协调配合，实现更多更复杂的功能。

本节小结



**1.1.2操作系统的特征**

**操作系统有并发、共享、虚拟、异步四个特征。其中，并发和共享是两个最基本特征，二者互为存在条件。**

1.并发：指两个或多个事件在同一时间间隔内发生。这些事件宏观上是同时发生的，但微观上是交替发生的。操作系统的并发性是指计算机系统中“同时”存在多个运行的程序（这些程序宏观上看是同时运行着的，而微观上看是交替运行的），因此它具有处理和调度多个程序同时执行的能力。

操作系统就是伴随着“多道程序技术”而出现的。因此，操作系统和程序并发是一起诞生的。

而并行和并发是不同的，并行是指两个或多个事件在同一时刻同时发生。

单核CPU同一时刻只能执行一个程序，各个程序只能并发地执行。

多核CPU同一时刻可以同时执行多个程序，多个程序可以并行地执行

比如Intel的第八代i3处理器就是4核CPU，可以并行地执行4个程序。即使对于4核CPU来说，只要有4个以上的程序需要“同时”运行，那么并发性依然是必不可少的。因此并发性是操作系统一个最基本的特性。

2.共享：指资源共享，系统中的资源可供内存中多个并发执行的**进程**共同使用。

计算机系统中的资源有两种共享方式：（1）互斥共享方式，是指系统中的某些资源，虽然可以提供给多个进程使用，但一个时间段内只允许一个进程访问该资源（2）同时共享方式，是指系统中的某些资源，允许一个时间段内由多个进程“同时”对它们进行访问。这里的“同时”往往是宏观上的，而在微观上，这些进程可能是交替地对该资源进行访问的（分时共享）

互斥共享方式：使用QQ和微信视频。同一时间段内摄像头只能分配给其中一个进程。

同时共享方式：使用QQ发送文件A，同时使用微信发送文件B。宏观上看，两边都在同时读取并发送文件，说明两个进程都在访问硬盘资源。微观上看，两个进程是交替着访问硬盘的。但同时共享方式也能允许两个进程在微观上也是同时访问资源，比如扬声器，同时播放两个进程发生的声音，微观上看，两个进程是同时使用扬声器的。

并发和共享是OS两个最基本的特征，两者互为存在的条件：（1）资源共享是以程序的并发为条件的，若系统不允许程序并发执行，则系统中只能有一个程序正在运行，则共享性失去存在的意义。（2）若系统不能对资源共享实施有效的管理，则必将影响到程序的并发执行，甚至根本无法并发执行。

3.虚拟：把一物理上的实体变为若干个逻辑上的对应物。物理实体（前者）是实际存在的，而逻辑上对应物（后者）是用户感受到的。

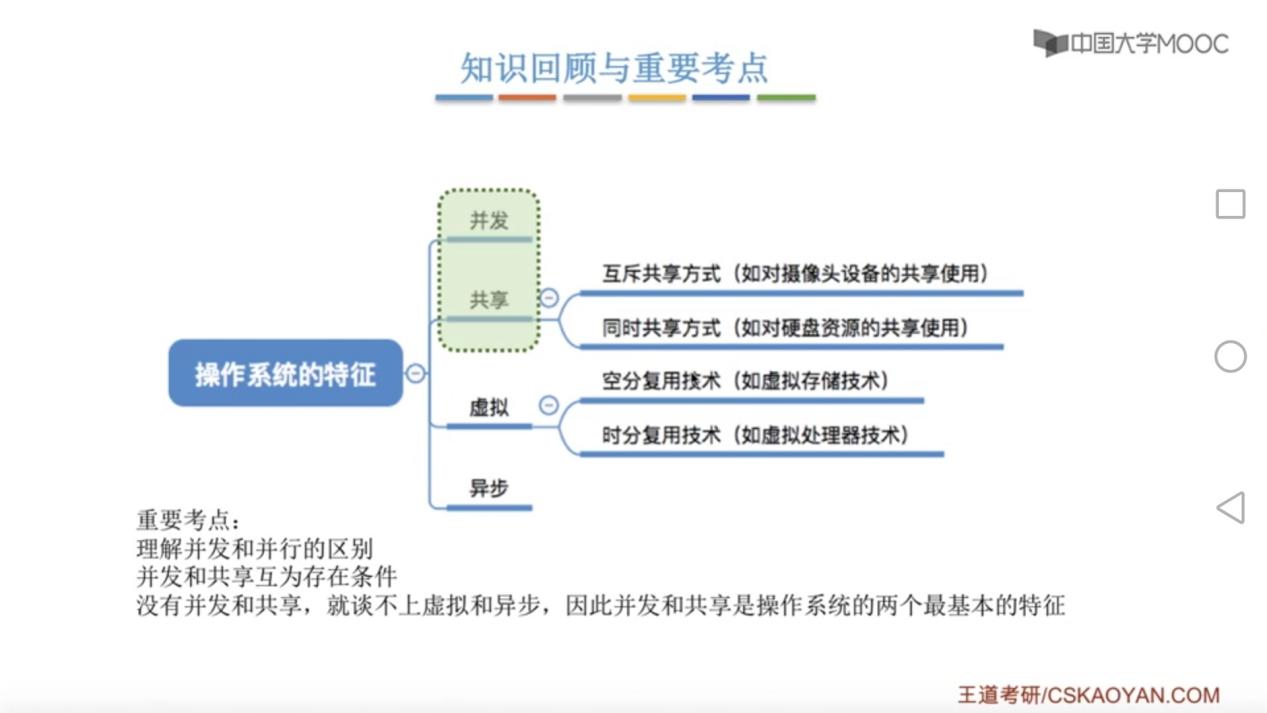
例如虚拟存储器技术，可以让实际内存只有4GB的计算机同时运行若干所需内存大于4GB的进程，在用户看来似乎内存远远大于4GB。（空分复用技术）

再如虚拟处理器技术，一个程序需要被分配CPU才能正常执行，但利用虚拟处理器技术，可以让实际上只有一个单核CPU的计算机，在用户看来似乎有多个CPU在同时为每个用户服务。（时分复用技术）

如果失去了并发性，则一个时间段内只允许一道程序运行，虚拟性就会失去意义。

4.异步：在多道程序环境下，允许多个程序并发执行，但由于资源有限，进程的执行不是一贯到底的，而是走走停停，以不可预知的速度向前推进，这就是进程的异步性。

如果失去了并发性，即系统只能串行执行各个程序，那么每个程序的执行会一贯到底。所以只有系统具有并发性，才会有异步性。



**1.2操作系统的发展与分类**

1.手工操作阶段

主要缺点：用户独占全机、人机速度矛盾导致资源利用率极低

2.批处理阶段

（1）单道批处理系统

引入脱机输入/输出技术（用外围机+磁带完成），并由监督程序负责控制作业的输入、输出

各个程序员将自己的程序放在纸带机上，由外围机把各个纸带机里的程序存到磁带上。之后计算机从磁带里读取程序和数据。由监督程序控制中磁带中输入输出作业，这里的监督程序就是操作系统的雏形。

主要特征：自动性、顺序性、单道性

自动性：顺利情况下磁带上的一批作业无需人工干预就能自动地逐个依次运行

顺序性：磁带上各道作业顺序地进入内存，完成顺序与进入内存顺序在正常情况下相同。

单道性： 内存中仅有一道程序运行，即监督程序每次从磁带上只调入一道程序进入内存运行，当该程序完成或发生异常时，才换入其后继程序进入内存运行。

主要优点：缓解了一定程度的人机速度矛盾，资源利用率有所提升。

主要缺点：内存中仅能有一道程序运行，只有在该程序运行结束之后才能调入下一道程序。CPU有大量的时间是在空闲等待I/O完成。资源利用率依然很低。

（2）多道批处理系统

在此阶段，操作系统正式诞生，用于支持多道程序并发运行。多道程序设计技术允许多个程序同时进入内存并允许它们在CPU中交替地运行。当一道程序因I/O请求而暂停运行时，CPU便立即转去运行另一道程序。

主要特征：多道、宏观并行、微观串行

多道：计算机内存中同时存放多道相互独立的程序

宏观并行：同时进入系统的多道程序都处于运行过程中，即它们先后开始各自的运行，但都未运行完毕

微观串行：内存中的多道程序轮流占有CPU，交替执行

主要优点：多道程序并发执行，共享计算机资源。资源利用率大幅提升，CPU和其他资源更能保持“忙碌”状态，系统吞吐量增大。

主要缺点：用户响应时间长，没有人机交互功能

3.分时操作系统

在分时操作系统中，计算机以时间片为单位轮流为各个用户/作业服务，各个用户可通过终端与计算机进行交互。

主要特征：同时性、交互性、独立性、及时性

同时性：允许多个终端用户同时使用一台计算机

交互性：用户能够方便地与系统进行人机对话，用户通过终端采用人机对话的方式直接控制程序运行，同程序进行交互

独立性：系统中多个用户可以彼此独立进行操作，互不干扰，感觉不到别人存在。

及时性：用户请求能在很短时间内获得响应，采用时间片轮转方式使一台计算机同时为多个终端服务，使用户能够对系统的及时响应感到满意。

主要优点：用户请求可以被即时响应，解决了人机交互问题。允许多个用户同时使用一台计算机，并且用户对计算机的操作相互独立，感受不到别人存在

主要缺点：不能优先处理一些紧急任务。操作系统对各个用户/作业都是完全公平的，循环地为每个用户/作业服务一个时间片，不区分任务的紧急性。

4.实时操作系统

主要优点：能够优先响应一些紧急任务，某些紧急任务不需时间片排队

在实时操作系统的控制下，计算机系统接收到外部信号后及时进行处理，并且要在严格的时限内处理完事件。实时操作系统的主要特点是及时性和可靠性。

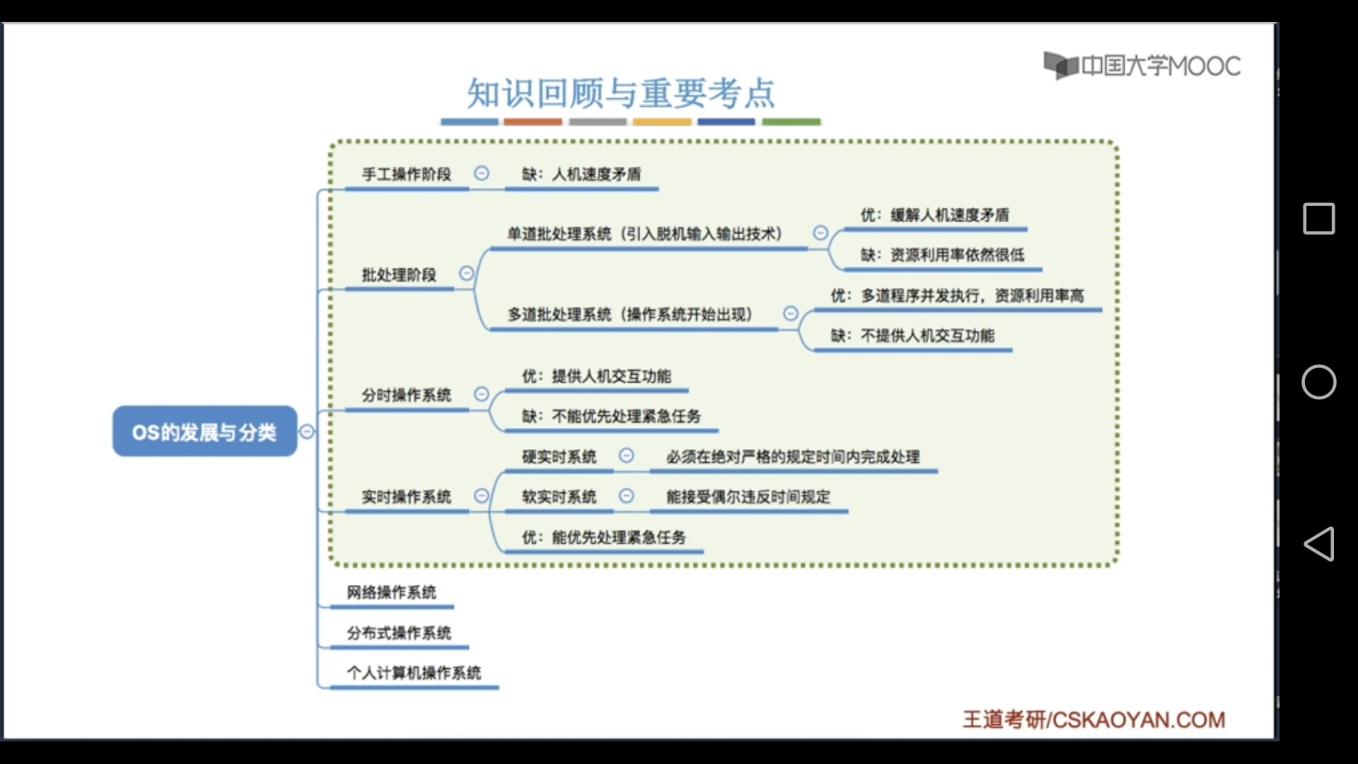
实时操作系统分为硬实时系统和软实时系统。

硬实时系统必须在绝对严格的规定时间内完成处理，否则会产生严重后果。如导弹控制系统、自动驾驶系统

软实时系统能接受偶尔违反时间规定且不会引起任何永久性的损害，如飞机订票系统、银行管理系统、12306火车订票系统。

除上述几种操作系统外，还有许多种各式各样的操作系统，如网络操作系统、分布式操作系统、个人计算机操作系统。

本节小结



**1.3.1操作系统的运行机制**

“指令”是处理器(CPU)能识别、能执行的最基本命令

1.内核

内核是操作系统最重要最核心的部分，也是最接近硬件的部分。由很多内核程序组成了“操作系统内核”，或简称“内核”。操作系统的功能未必都在内核中，如图形化用户界面GUI。

操作系统内核作为“管理者”，有时会让CPU执行一些“特权指令”，如：内存清零指令。这些指令影响重大，只允许“管理者”--即操作系统内核来使用。应用程序只能使用“非特权指令”，如加法指令等。在CPU设计和生产的时候就划分了特权指令和非特权指令，因此CPU执行一条指令前就能判断其类型。

为了让CPU能区分某时刻正在运行的是内核程序还是应用程序，CPU被划分为两种状态：“内核态”和“用户态”。处于内核态时，正在运行的是内核程序，此时可以执行特权指令。处于用户态时，正在运行的是应用程序，此时只能执行非特权指令。CPU中有一个寄存器叫程序状态字寄存器(PSW)，其中有个二进制位，用0和1来区分内核态和用户态。

用户态也叫目态，内核态也叫管态

2.内核态、用户态的切换

（1）刚开机时，CPU为内核态，操作系统内核程序先在CPU上运行

（2）开机完成后，用户可以启动某个应用程序

（3）操作系统内核程序在合适的时候主动让出CPU，让该应用程序在CPU上运行。

操作系统内核在让出CPU之前，会用一条特权指令把PSW的标志位设置为“用户态”

（4）应用程序运行在“用户态”

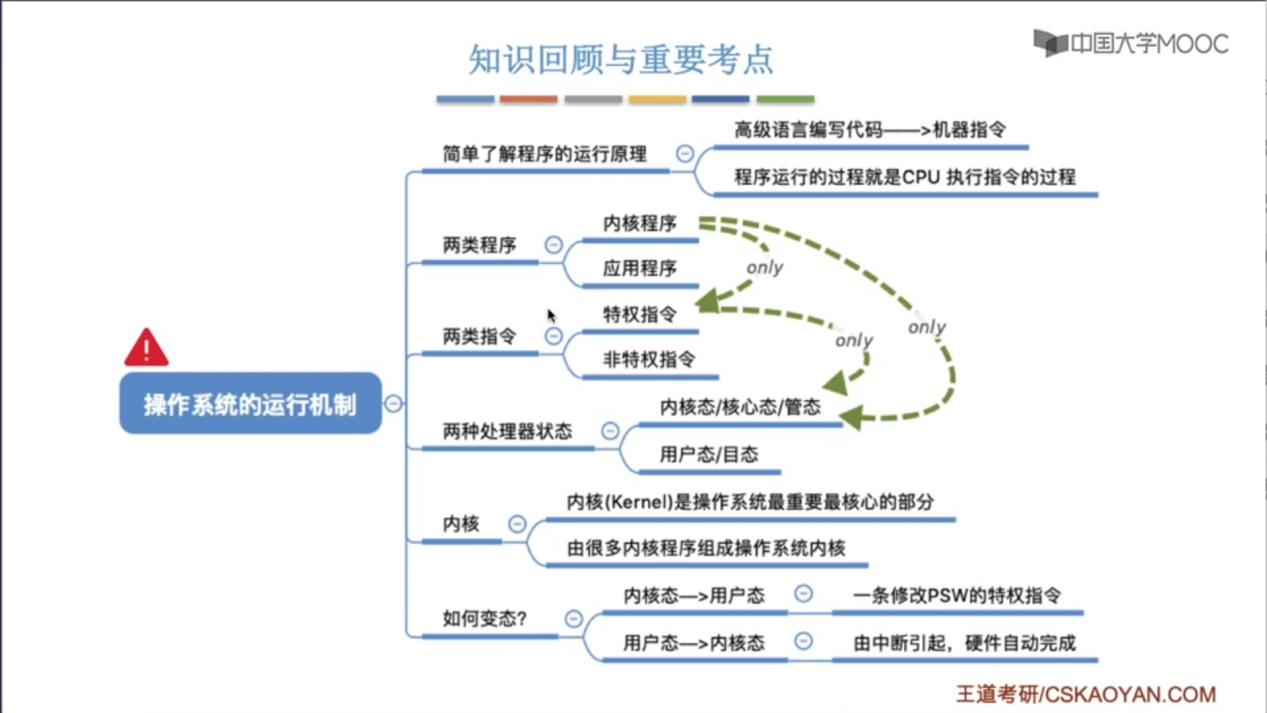
（5）若应用程序中植入了特权指令，CPU发现接下来要执行的指令是特权指令，但又正处于“用户态”，会拒绝执行这条特权指令。这个非法事件会引发一个中断信号，CPU检测到中断信号后会立即变为“内核态”，并停止运行当前的应用程序，转而运行处理中断信号的内核程序。

（6）中断使操作系统再次夺回CPU的控制权

（7）操作系统会对引发中断的事件进行处理，处理完成后再把CPU使用权交给别的应用程序

**CPU从内核态切换到用户态，是通过执行一条特权指令--修改PSW的标志位为“用户态”，这意味着操作系统将主动让出CPU使用权**

**CPU从用户态切换到内核态，是由“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将强行夺回CPU的使用权。**



**1.3.2中断和异常**

CPU上会运行两种程序，一种是OS的内核程序，一种是应用程序，前者是系统的管理者。

在合适的情况下，操作系统内核会把CPU的使用权主动让给应用程序。“中断”是让操作系统内核夺回CPU使用权的唯一途径，如果没有“中断”机制，那么一旦应用程序在CPU上运行了，CPU就会一直运行这个应用程序。这样就无法实现“并发”。

中断可分为两种类型：内中断和外中断

1.内中断

内中断与当前执行的指令有关，中断信号来源于CPU内部。也称“异常”。

例子：1.试图在用户态下执行特权指令

2.非特权指令也可能引发内中断，如除法指令的除数为0

若当前执行的指令是非法的，则会引发一个中断信号

3.应用程序想请求操作系统内核的服务，此时会执行一条特殊的指令--陷入指令，该指令会引发一个内部中断信号。系统调用就是通过陷入指令来完成的，执行陷入指令，意味着应用程序主动地将CPU控制权还给操作系统内核，陷入指令不是特权指令。

2.外中断

外中断与当前执行的指令无关，中断信号来源于CPU外部

例子：1.时钟中断--由时钟部件发来的中断信号

程序的并发执行就离不开时钟部件提供的中断信号。时钟部件每隔一个时间片（如50ms）会给CPU发送一个时钟中断信号。当正在CPU上运行的程序运行满一个时间片的时间后，CPU接到时钟部件传来的中断信号，会暂停正在运行的程序并立即转入与时钟中断相应的处理程序（同时变成内核态），处理完毕后，内核程序会将CPU的使用权交给另一个正在等待的程序（同时切换成用户态），另一个程序开始执行。这样的过程不断重复，每个应用程序就这样轮流使用CPU，每次使用CPU长达一个时间片的时间，做到了程序的并发执行。

2.I/O中断--由输入输出设备发来的中断信号，当输入输出任务完成时向CPU发送中断信号，通知CPU任务完成，之后CPU会转入相应的处理I/O中断的内核程序。

CPU在每一条指令执行结束后，会例行检查是否有外部中断信号需要处理。

3.内中断（异常）的分类

内中断由引发原因可分为三类：陷入、故障、终止。

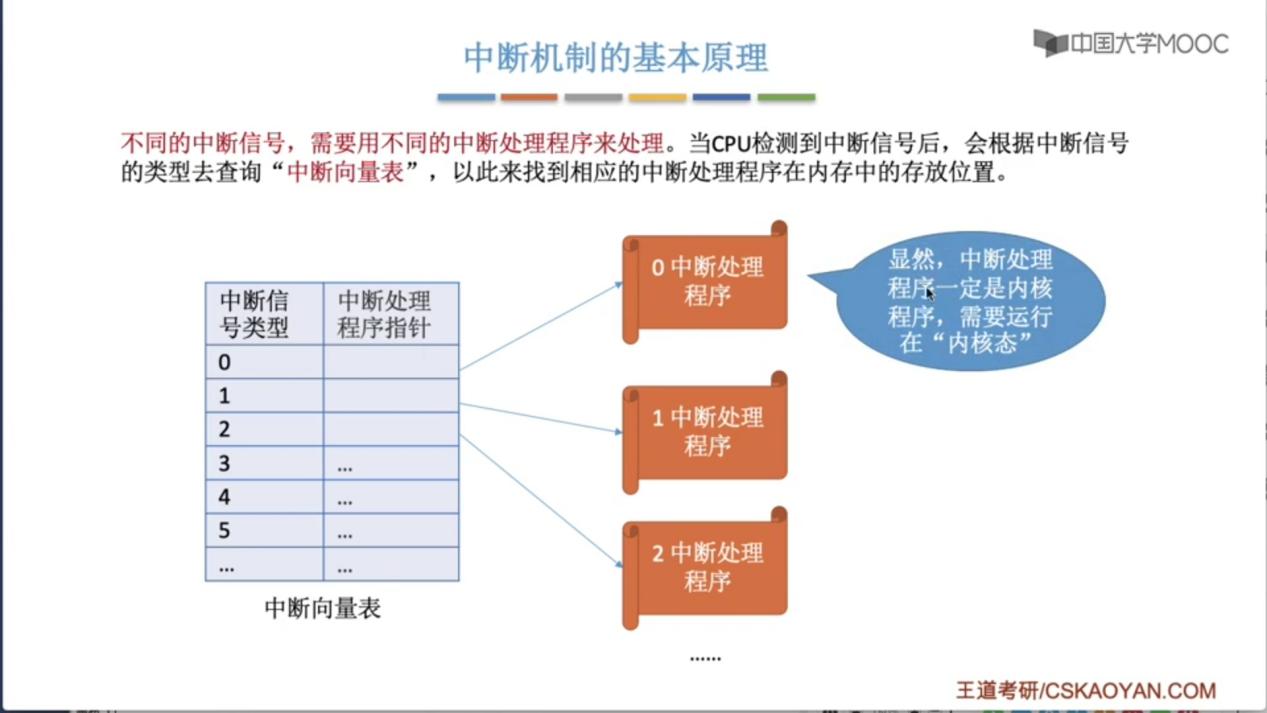
陷入（trap），由陷入指令引发的异常，是应用程序故意引发的。

故障（fault），由错误条件引起的，可能被内核程序修复的问题。内核程序修复故障后会把CPU使用权还给应用程序，让它继续执行下去。如：缺页故障。

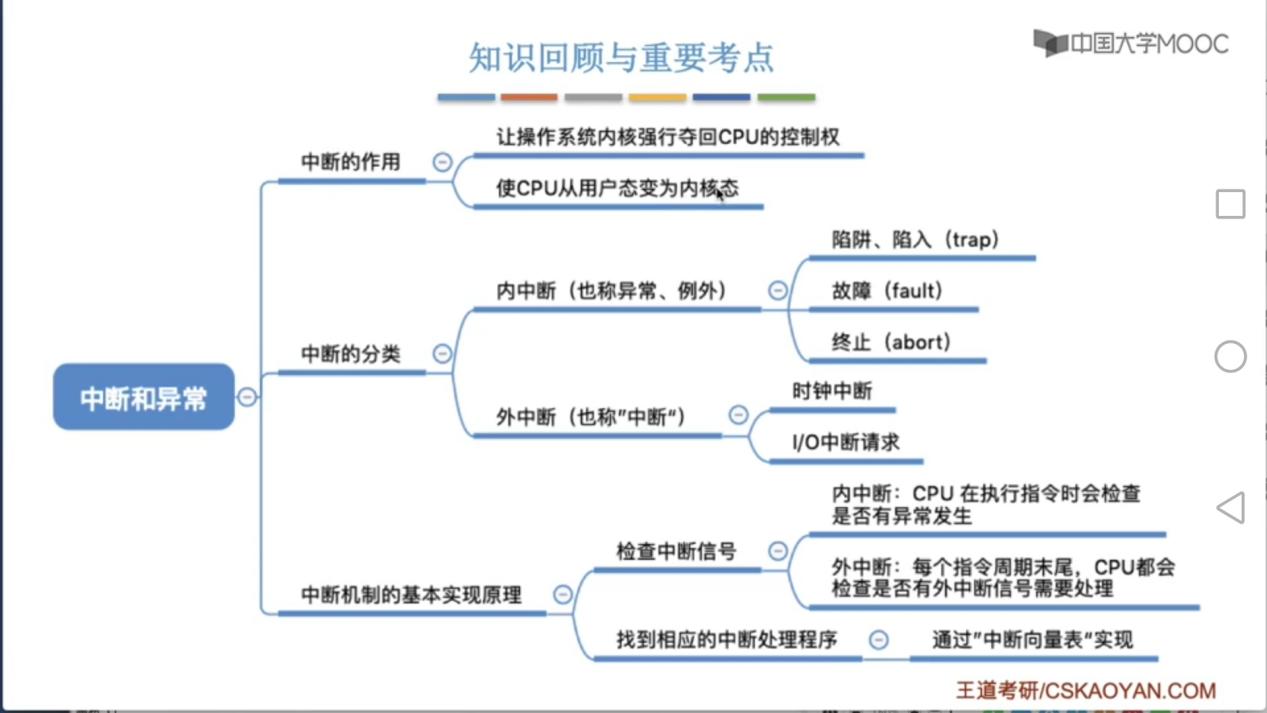
终止（abort），由致命错误引起，内核程序无法修复该错误，因此一般不再将CPU使用权还给引发终止的应用程序，而是直接终止该应用程序。如：整数除以0（这是程序自身的bug，操作系统也无法修复）、非法使用特权指令等。

4.中断机制的基本原理

不同的中断信号，需要用不同的中断处理程序来处理。当CPU检测到中断信号后，会根据中断信号的类型去查询“中断向量表”，以此来找到相应的中断处理程序在内存中的存放位置。中断处理程序一定是内核程序，需要运行在“内核态”。



本节小结



**1.3.3系统调用**

操作系统作为用户和计算机硬件之间的接口，需要向上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。

“系统调用”是操作系统提供给应用程序使用的接口，可以理解为一种可供应用程序调用的特殊函数，应用程序可以通过系统调用来请示获得操作系统内核的服务。

1.与库函数的区别

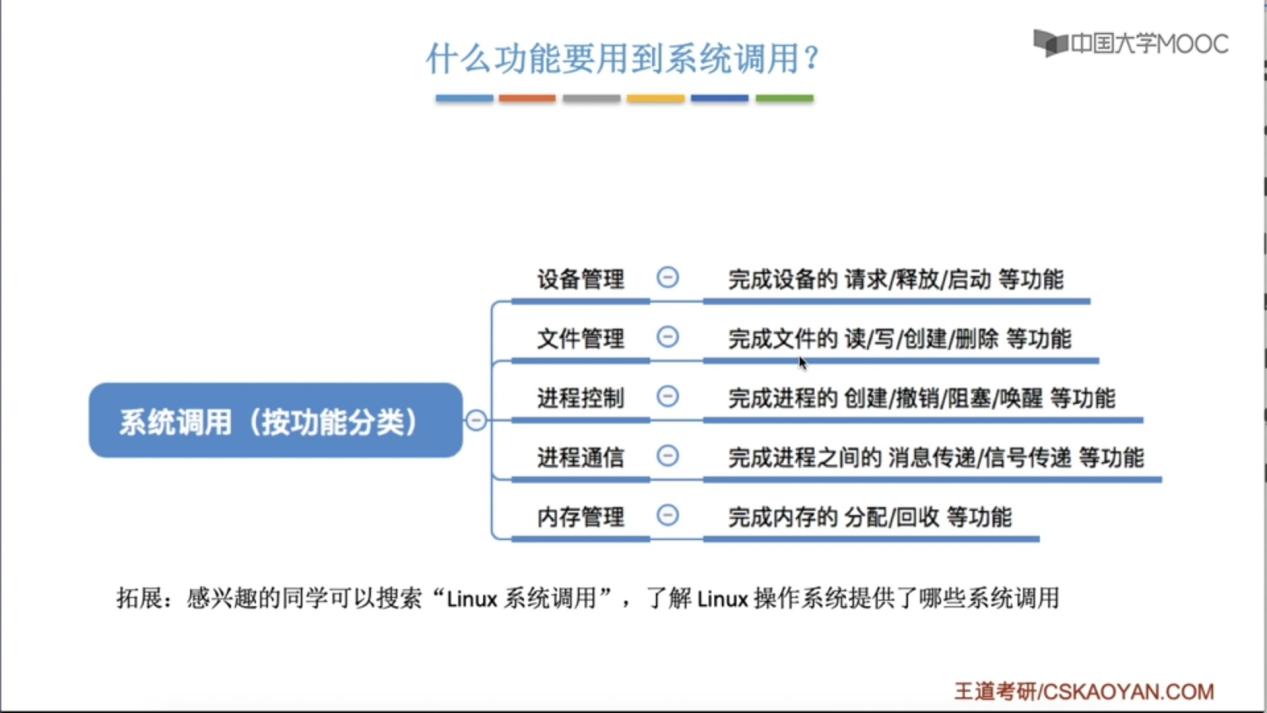
操作系统向上提供系统调用，使得上层程序能请求内核的服务。在此之上，有各种高级编程语言，向上提供库函数。有时会将系统调用封装成库函数，以隐藏系统调用的一些细节，使编程更加方便。对于普通的应用程序，可直接进行系统调用，也可使用库函数。有的库函数涉及系统调用（如“创建新文件”函数），有的库函数不涉及（如“取绝对值”函数）。

2.系统调用的必要性

当系统存在的两个进程同时向打印机发出打印请求时，如果可以随意地、并发地共享打印机资源，那么打印机就会将两个进程打印的内容混在一起。因此，解决方法就是由操作系统内核对共享资源进行统一的管理，并向上提供“系统调用”，用户进程想要使用打印机这种共享资源，只能通过系统调用向操作系统内核发出请求。内核会对各个请求进行协调处理。

3.系统调用的功能

系统调用按功能大致可分为如下几类：设备管理、文件管理、进程控制、进程通信、内存管理



凡是与共享资源有关的操作（如存储分配、I/O操作、文件管理等），都必须通过系统调用的方式向操作系统内核提出服务请求，由操作系统内核代为完成。这样可以保证系统的稳定性和安全性，防止用户进行非法操作。

3.系统调用过程

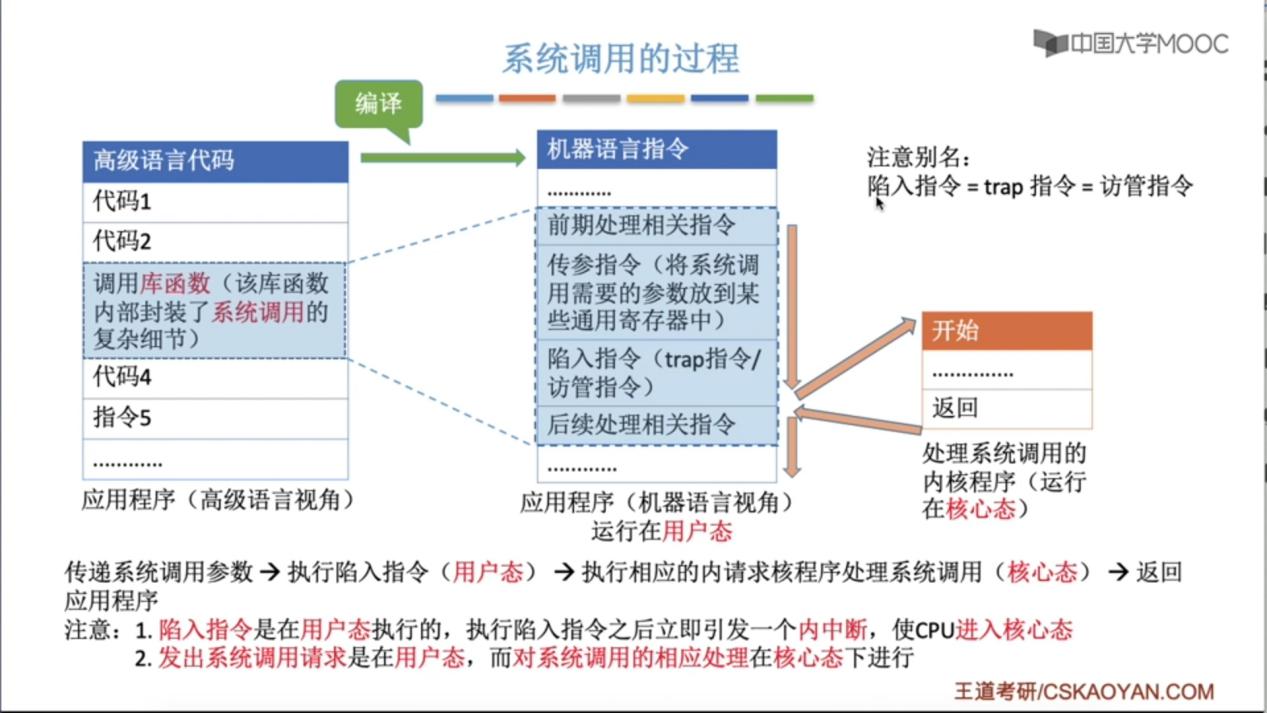
当应用程序想要进行系统调用时，先通过传参指令给CPU向CPU寄存器中传递一些必要的参数。比如指明系统调用的类型，如fork。参数全部进入寄存器后，应用程序会执行“陷入指令”，陷入指令的执行会引发内中断，CPU检测到内中断信号后，会暂停执行应用程序，转入相应的中断处理程序--即系统调用的入口程序。这个程序是内核程序，CPU切换到内核态来执行系统调用入口程序的指令。系统调用入口程序会检查寄存器中的参数，判断用户需要哪种系统调用服务。然后入口程序会调用对应种类的系统调用的处理程序，如fork系统调用的处理程序，让这种系统调用的处理程序在CPU上执行。执行完毕后，CPU转回用户态然后接着执行之前的应用程序。

整个过程可以表示成几个步骤：

传递系统调用参数->执行陷入指令（用户态）->执行相应的内核程序处理系统调用（核心态）->返回应用程序

注意：1.陷入指令是非特权指令，是在用户态执行的，执行陷入指令之后立即引发一个内中断，使CPU进入核心态。

2.发出系统调用请求是在用户态，而对系统调用的相应处理在核心态下进行。



陷入指令又叫trap指令，访管指令。

**1.4操作系统的体系结构**

计算机系统的层次结构从下至上依次是裸机、操作系统、应用程序和用户。操作系统还可以进一步划分成内核部分、非内核部分。

操作系统最核心的功能需要放在内核中，如时钟管理、中断处理、原语（设备驱动、CPU切换）。

时钟管理就是利用时钟中断实现计时功能。

原语是一种特殊的程序，具有原子性，也就是说，这段程序的运行必须一气呵成，不可被“中断”，在执行原语的过程中，中断信号送到CPU，CPU也必须把原语执行完成才能去处理中断。原语处于操作系统最底层，是最接近硬件的部分。其运行时间较短，调用频繁。

Ubuntu、CentOS的开发团队其主要工作是实现非内核功能，其内核都是用了Linux内核。

内核是操作系统最基本最核心的部分，实现操作系统内核功能的那些程序就是内核程序。由于时钟管理、中断处理、原语三个模块是与硬件关联较紧密的模块，是必须放在内核中的。

而进程管理、存储器管理、设备管理更多的是对数据结构的操作，不直接涉及硬件。因此有的操作系统不把这些管理功能放在内核当中，只保留与硬件接触最紧密的功能。这就引出两种不同的操作系统设计方法，把所有这些功能都包含在内核当中的结构叫大内核。而只保留与硬件关系最紧密的部分的结构叫微内核。

如果采用微内核结构的话，不属于内核的进程管理、存储器管理、设备管理等的这部分功能就要运行在用户态。这样会对性能造成影响。当一个应用程序想要请求操作系统的服务，同时涉及到进程管理、存储器管理与设备管理。则这三个管理模块每次需要单独地请求内核服务，使CPU进行两次状态的切换，所以CPU共转换状态6次。而采用大内核结构，由于进程管理模块、存储器管理模块、设备管理模块都划在内核中，所以应用程序总共只需2次切换CPU状态。而切换状态的过程是有成本的，要消耗不少的时间，频繁地切换状态会降低系统性能。所以这是微内核结构的一个缺点--需频繁切换CPU的状态，性能低。

大内核结构的优点在于高性能，而缺点在于内核代码庞大，结构混乱，难以维护。相比之下，微内核的内核功能少，结构清晰，更加稳定，方便维护。

典型的大内核操作系统有Linux、Unix。典型的微内核操作系统有Windows NT。

**2.1.1进程的概念、组成、特征**

程序：是静态的，是存放在磁盘里的可执行文件，是一系列的指令集合

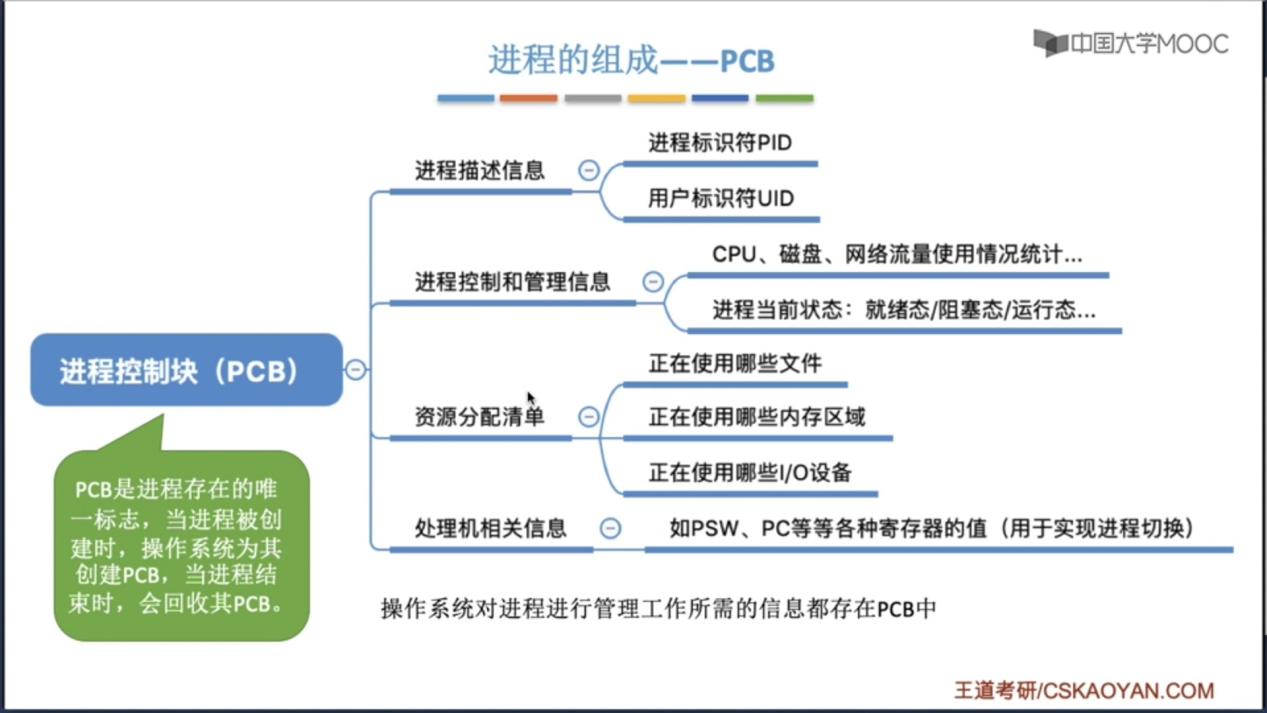
进程：是**动态**的，是程序的一次执行过程

同一个程序多次执行会对应多个进程。

1.进程控制块PCB

当进程被创建时，操作系统会为该进程分配一个唯一的、不重复的“身份证号”---PID（进程ID）。操作系统要记录PID,进程所属用户ID(UID)。还要记录给进程分配了哪些资源（如分配了多少内存、正在使用哪些I/O设备、正在使用哪些文件），这些信息可用于实现操作系统对资源的管理。还要记录进程的运行情况（如：CPU使用时间、磁盘使用情况、网络流量使用情况等），可用于实现操作系统对进程的控制、调度。这些信息都保存在一个数据结构PCB中，即进程控制块。

操作系统需要对各个并发运行的进程进行管理，但凡管理时所需要的信息，都会被放在PCB中。其中的信息大致可分为四类：进程描述信息（如进程标识符PID、用户标识符UID）、进程控制和管理信息（如CPU、磁盘、网络流量使用情况统计，以及进程当前状态等）、资源分配清单（正在使用哪些文件、哪些内存区域、哪些I/O设备）、处理机相关信息（如PSW、PC等待各种寄存器的值，**用以实现进程切换**）**PCB是进程存在的唯一标志**，当进程被创建时，操作系统为其创建PCB，当进程结束时，会回收其PCB。



Linux里的进程控制块部分字段源码：

struct task\_struct{

volatile long state; //进程状态：就绪态？运行态？阻塞态？

struct files\_struct \*files; //表示进程当前打开的文件

kuid\_t loginuid; //用户ID

struct io\_context \*io\_context; //I/O管理需要使用到的信息

...};

2.程序段和数据段

PCB是给操作系统用的，而每个进程还有程序段和数据段是给进程自己用的。程序段

写完程序之后，经过编译、链接会得到一个可执行文件，平时存放在硬盘中。程序运行前需将其读入内存中。并且操作系统创建对应的进程，也就要创建相应的PCB。程序中的指令序列也需要读入内存中，这些指令序列称为程序段。进程执行的过程就是CPU从程序段中取出指令，一条条执行的过程。过程执行过程中，产生的各种数据（如各种变量的值）放在内存的数据段中。

3.进程和进程实体

所以一个进程实体（即进程映像）由PCB、程序段、数据段组成。进程是动态的，进程实体（进程映像）是静态的。进程实体可看成是进程在某时刻的一张照片，反映了进程在某一时刻的状态。

引入了进程实体的概念后，可把进程定义为：进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位。

如果一台计算机同时挂三个QQ号，会对应三个QQ进程，它们的PCB和数据段各不相同，但程序段的内容都是相同的（相同的QQ程序）。

4.进程的特征

程序是静态的，进程是动态的。相比于程序，进程拥有以下特征：

（1）**动态性，是进程的最基本特征**。进程是程序的一次执行过程，是动态地产生、变化和消亡的。

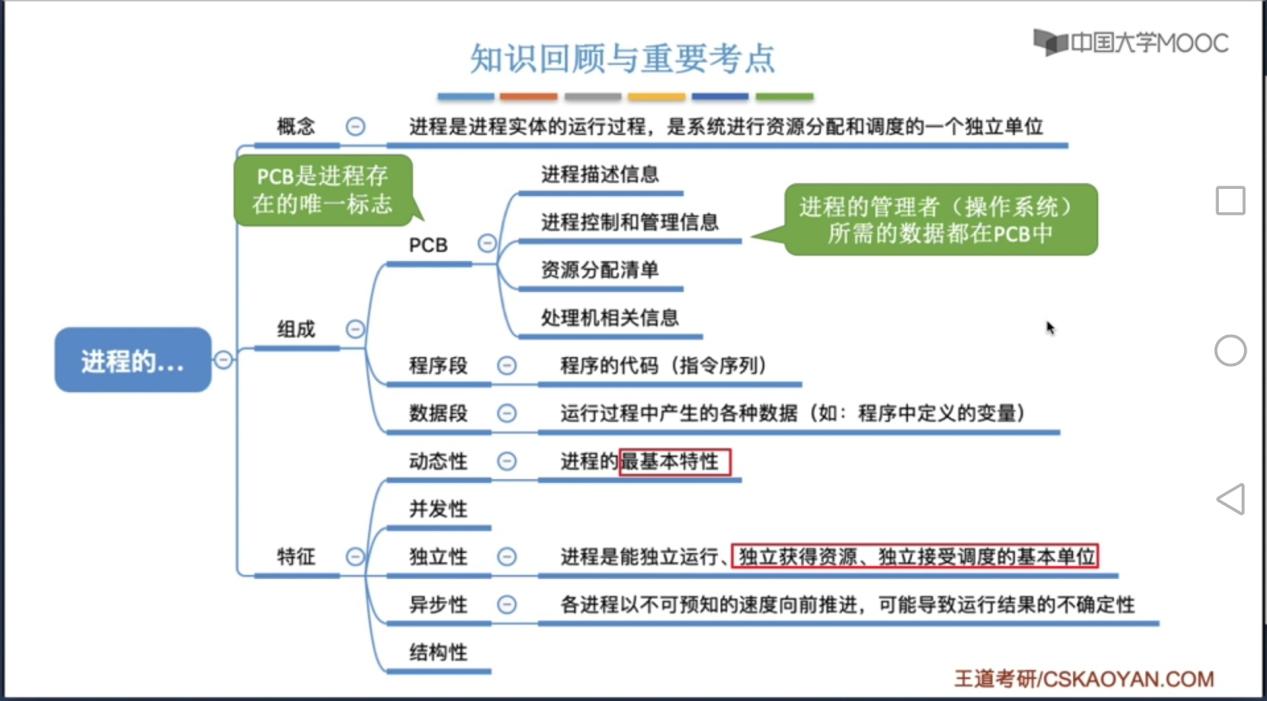
（2）并发性，内存中有多个进程实体，各进行可并发执行

（3）独立性，进程是能独立运行、独立获得资源、独立接受调度的基本单位

（4）异步性，各进程按各自独立、不可预知的速度向前推进，操作系统要提供“同步机制”来解决异步问题

（5）结构性，每个进程都会配置PCB，结构上看，进程由程序段、数据段、PCB组成

本节小结



**2.1.2进程的状态与组织**

**1.进程的状态**

进程正在被创建时，它的状态是“创建态”，在这个阶段操作系统会为进程分配除CPU外的资源、初始化PCB。

当进程创建完成后，便进入“就绪态”，处于就绪态的进程已经具备运行条件，但由于没有空闲的CPU，就暂时不能运行。

系统中可能会有多个进程处于就绪态，当CPU空闲时，操作系统就会选择一个就绪进程，让它在处理机上运行。假设正在运行的进程有一条指令发出系统调用，请求使用打印机资源，但打印机设备很“忙”，正在为别的进程服务，则正在运行的进程不能继续执行。

在进程运行的过程中，可能会请求等待某个事件的发生（如等待分配资源或等待其它进程响应），事件发生前，进程无法继续往下执行，此时操作系统会让这个进程放弃CPU，并让它进入“阻塞态”。当CPU再次空闲时，又会选择另一个“就绪态”进程在CPU上运行。

当打印机为其它的进程服务结束后，可以分配给刚才申请打印机而阻塞的进程，而该进程等待的事件发生了，此时操作系统会让进程从阻塞态回到就绪态。

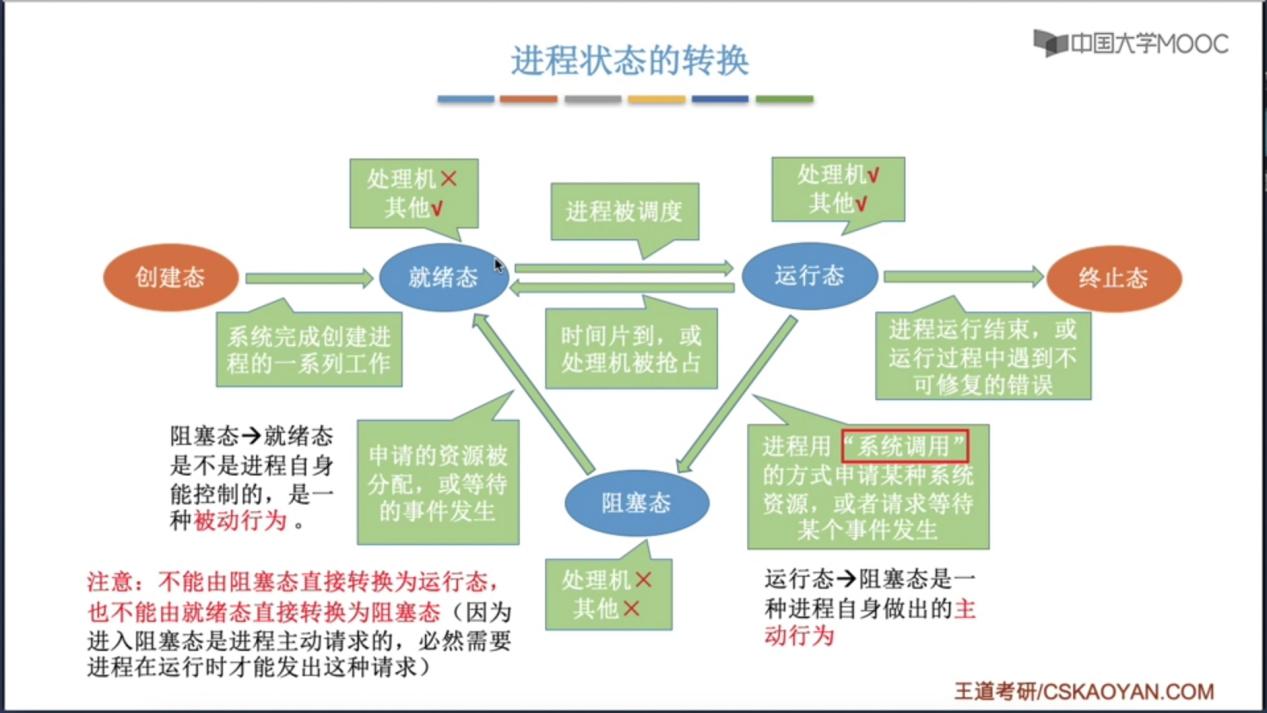
运行结束的进程会发生exit系统调用，请求操作系统终止该进程。此时该进程会进入“终止态”，操作系统会让该进程下CPU，并回收内存空间等资源，最后还要回收该进程的PCB。当终止进程的工作完成后，这个进程就彻底消失了。

进程在创建态下由系统完成创建进程的一系列工作，包括创建PCB，分配资源。当工作完成后，进程就具备了上CPU执行的条件，此时进程处于就绪态，就绪态的进程只差处理机这一种资源，其它资源全都具备。当就绪态的进程被调度后，进程就进入“运行态”，运行态的进程拥有处理机资源和其它所有所需的资源。有时，正在运行的进程可能请求某个事件发生，在事件发生前，无法继续执行的进程应该被剥夺处理机资源。除了处理机资源，进程还在等待其它的某种资源或事件，进入阻塞态。当其等待的其它的资源或事件再次具备后，进程回到就绪态。

运行态到阻塞态是一种进程自身做出的主动行为。阻塞态到就绪态不是进程自身执行的，是一种被动行为。进程不能从阻塞态转换为运行态，也不能从就绪态转换为阻塞态。因为进入阻塞态是进程主动请求的，必然需要进程在运行时才能发出这种请求。

正在运行的进程运行结束后，或遇到不可修改的错误，系统会为其做相应的善后工作，此时系统进入终止态，系统回收其资源撤销其PCB。

当正在执行的进程的时间片用完了，或者处理机被抢占。进程会从运行态转换为就绪态。这种情况下，进程只是失去了处理机，而不需等待其它事件的发生。



进程整个生命周期中，大部分时间都处于就绪态、运行态、阻塞态这三种基本状态。

单CPU情况下，同一时刻只会有一个进程处于运行态，多核CPU情况下，可能有多个进程处于运行态。

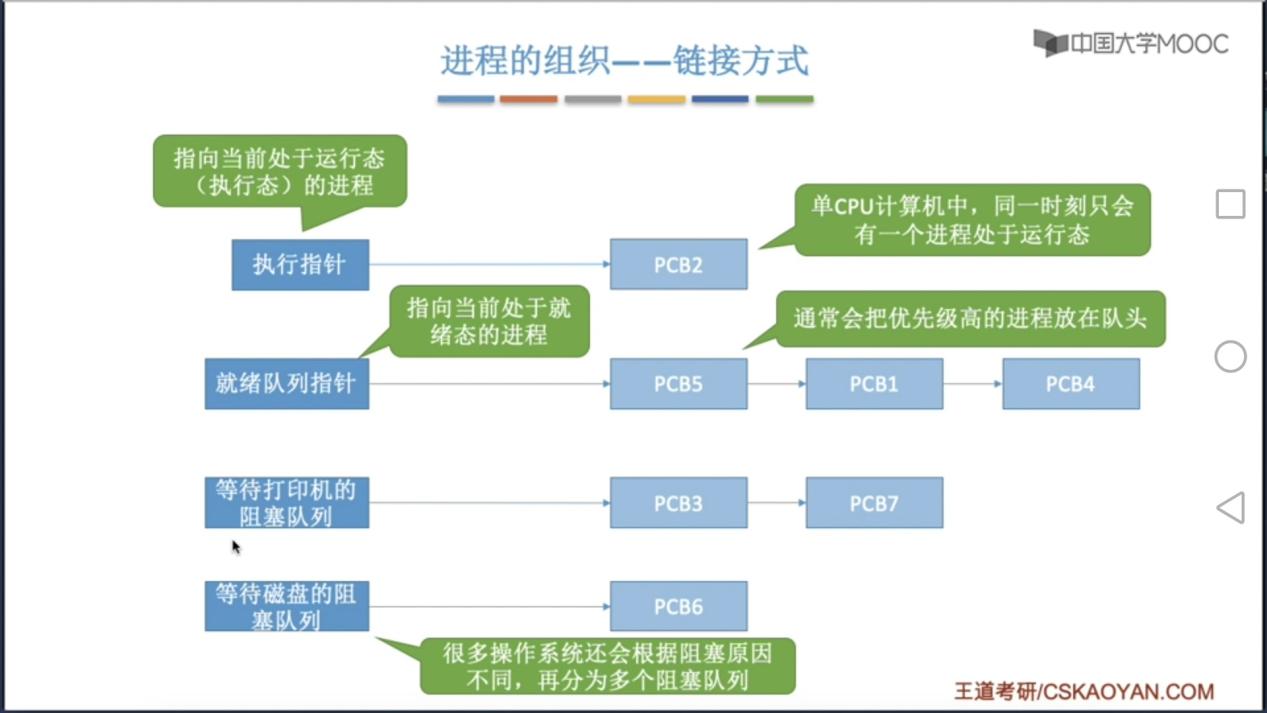
进程的PCB中，会有一个变量state来表示进程的当前状态。

为了对同一个状态下的各进程进行统一管理，操作系统会将各进程PCB组织起来。

**2.组织方式**

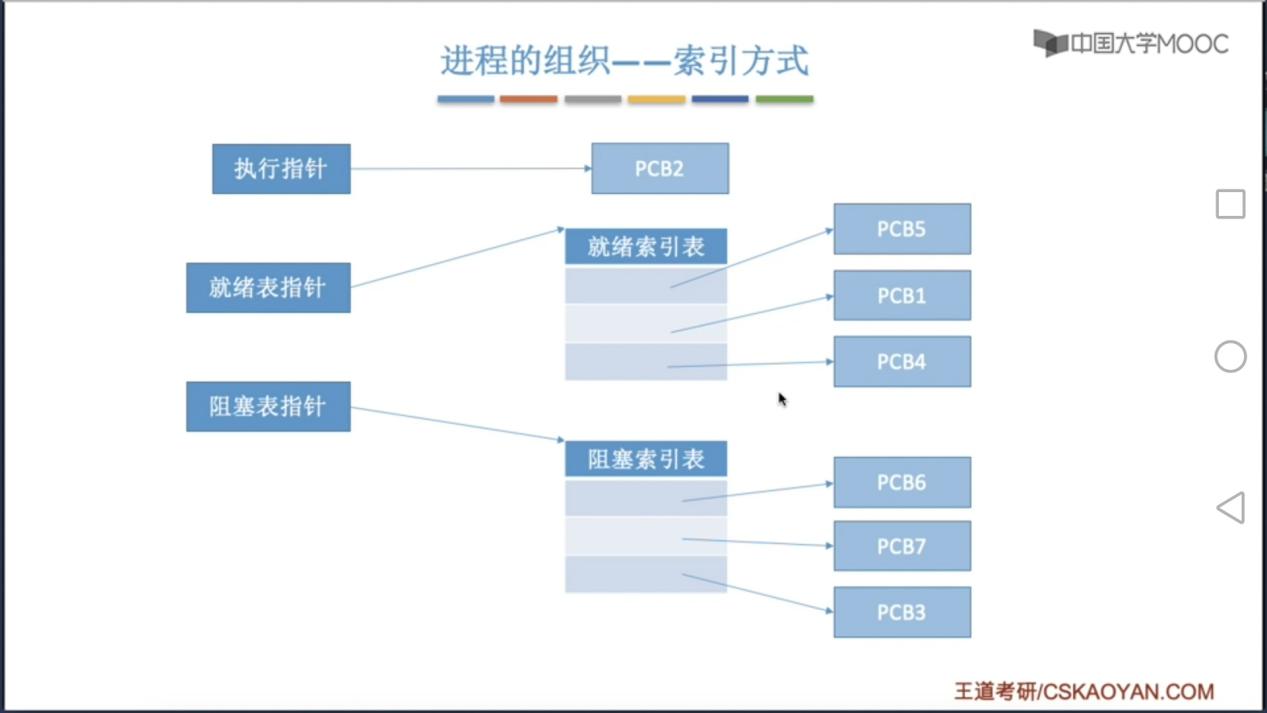
1.链接方式。

这种方式中，操作系统管理一系列的队列。每个队列都会指向相应状态的PCB，例如执行指针指向处于运行态的PCB。就绪队列指针指向当前处于就绪态的进程的PCB。通常会把优先级高的进程放在就绪队列队头。阻塞队列指针指向当前处于阻塞态的进程的PCB，很多操作系统还会区别不同阻塞原因，将阻塞队列分成多个。

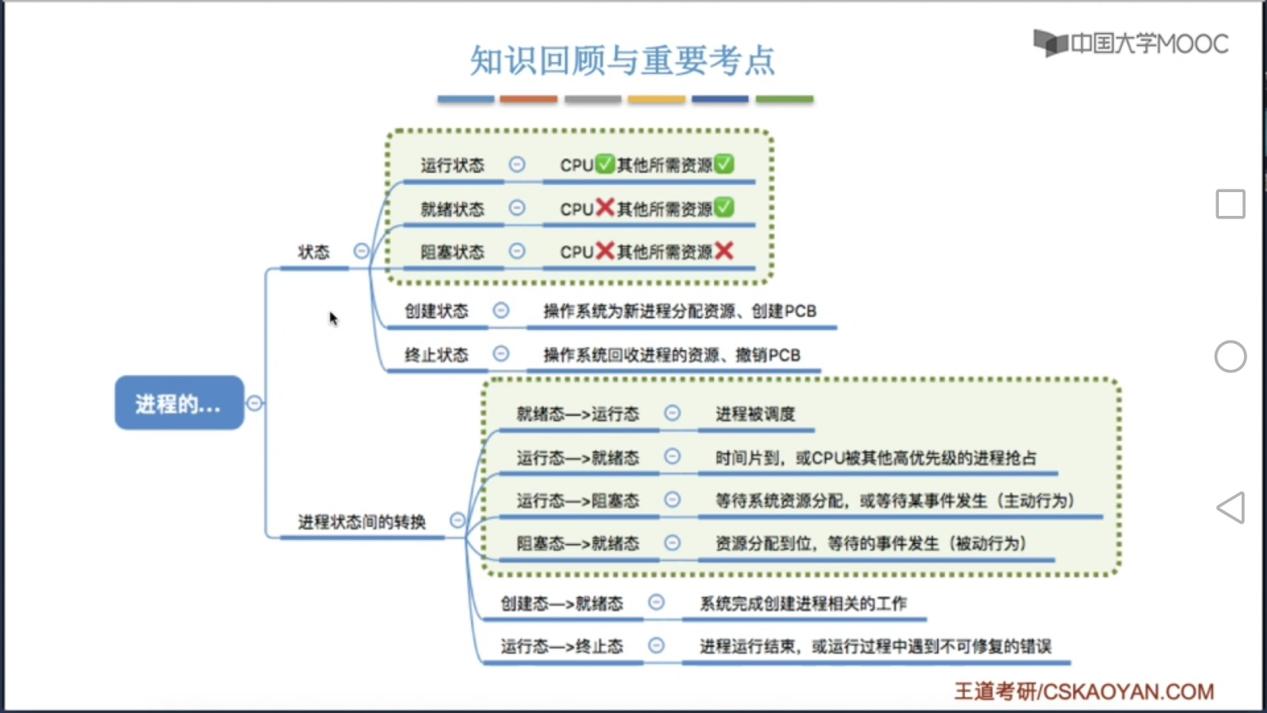


2.索引方式

这种方式中，操作系统给各种状态的进程建立索引表。每个索引表表项又指向相应的PCB。



本节小结



**2.1.3进程控制**

进程控制的主要功能是对系统中的所有进程实施有效的管理，它具有创建新进程、撤销已有进程、实现进程状态转换等功能。

进程的控制需要用“原语”来实现，原语具有“原子性”，不能被中断。

假设PCB中的变量state表示进程状态，1表示就绪态、2表示阻塞态

某时一阻塞态进程所等待的事件发生，则操作系统中，负责进程控制的内核程序至少要做这两件事：

（1）将进程的state设为1

（2）将进程的PCB从阻塞队列移出，插入就绪队列

CPU执行（1）后就收到中断信号，那么state为1的PCB还插在阻塞队列里CPU就去处理其它中断，这样会产生错误，所以进程控制要一气呵成，不能中断，否则有可能导致操作系统中的某些关键数据结构信息不统一，影响操作系统进行的其它管理工作。

可以用“关中断指令”和“开中断指令”这两个特权指令实现原子性。

正常情况下，CPU每执行一条指令后，会例行检查是否有外部中断信号需要处理。如果有，则暂停运行当前程序，转去执行相应的中断处理程序。

当CPU执行了关中断指令之后，就不再例行检查中断信号，直到执行开中断指令之后才会恢复检查。这样，关中断、开中断之间的这些指令序列就是不可被中断的，这就实现了“原子性”。

1.创建原语

操作系统创建一个进程时使用的原语是创建原语，创建原语会做如下几件事情：

（1）申请空白PCB

（2）为新进程分配所需资源

（3）初始化PCB（设置PID,UID等等）

（4）将PCB插入就绪队列

引起进程创建的事件有用户登录、作业调度、提供服务（用户向OS提出某些请求时，会新建一个进程处理该请求）、应用请求（由用户进程主动请求创建子进程）

2.撤销原语

操作系统要终止一个进程时使用的原语是撤销原语，撤销原语所做的事情如下：

（1）从PCB集合中找到终止进程的PCB

（2）若进程正在运行，立即剥夺CPU，将CPU分配给其他进程

（3）终止其所有子进程

（4）将该进程拥有的所有资源归还给父进程或操作系统

（5）删除PCB

引起进程终止的事件：正常结束（进程自己使用exit系统调用请求终止）、异常结束（整数除以0、非法使用特权指令等）、外界干预（用户打开任务管理器杀掉进程）

3.阻塞原语

将进程的状态由运行态变为阻塞态，需使用阻塞原语

（1）找到要阻塞的进程对应的PCB

（2）保护进程运行现场，将PCB状态信息设置为“阻塞态”，暂时停止进程运行

（3）将PCB插入相应事件的等待队列

引起进程阻塞的事件：等待系统分配某种资源、等待相互配合的其它进程完成工作

4.唤醒原语

将进程的状态由阻塞态变为就绪态，需使用唤醒原语

（1）在事件等待队列中找到PCB

（2）将PCB从等待队列中移除，设置进程为就绪态

（3）将PCB插入就绪队列，等待被调度

引起进程唤醒的事件：阻塞进程所等待的事件发生

5.切换原语

切换原语就让一个正在CPU上运行的进程从运行态切换到就绪态（换下CPU），然后将另一个处于就绪态的进程切换到运行态（换上CPU）

（1）将换下进程的运行环境信息（进程上下文）存入PCB

（2）PCB移入相应队列

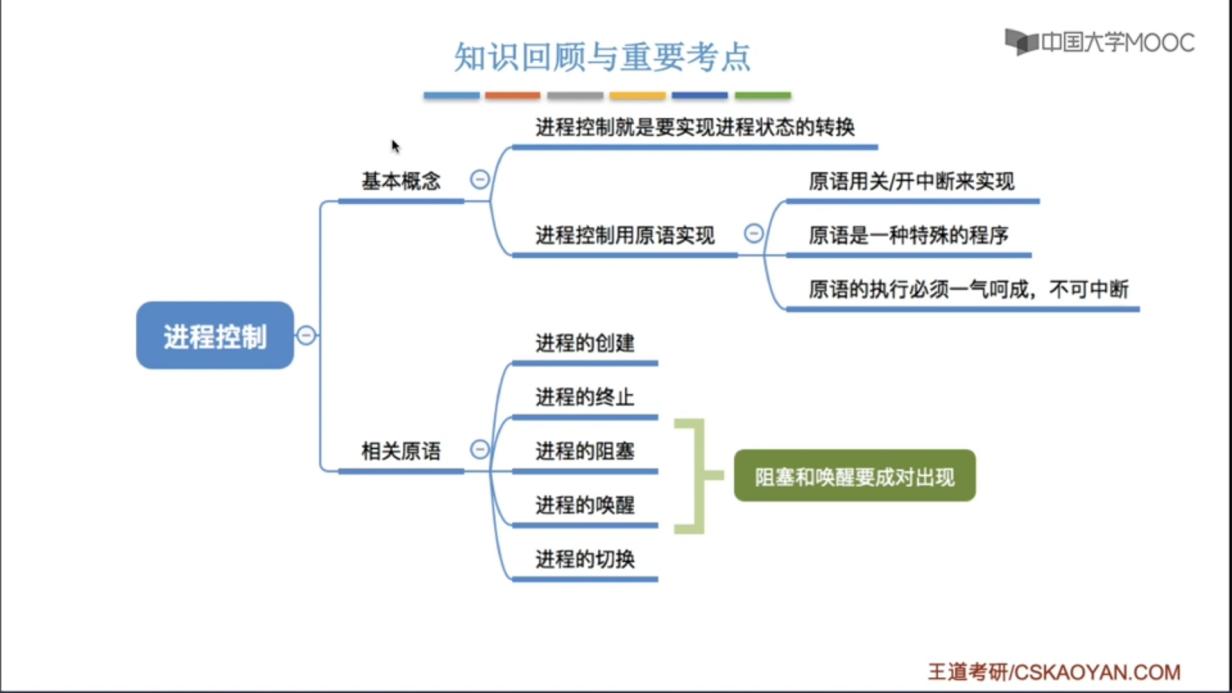
（3）选择另一个进程执行，并更新其PCB

（4）根据PCB恢复新进程所需的运行环境

引起进程切换的事件：当前进程时间片到、有更高优先级的进程到达、当前进程主动阻塞、当前进程终止

CPU中会设置很多寄存器，来存放程序运行过程中所需的某些数据。如PSW程序状态字寄存器，PC程序计数器存放下一条指令的地址，IR指令寄存器，存放当前正在执行的指令，还有一些通用寄存器。

在进程切换时，先在PCB中保存这个进程的运行环境（一些必要的寄存器信息，如PSW，PC，通用寄存器等）。当原来的进程再次投入运行时，可通过PCB恢复它的运行环境。



**2.1.4进程通信**

进程通信就是进程间的信息交换

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

1.共享存储

操作系统为两个进程分配一个共享空间，二者的通信通过共享空间完成。

两个进程对共享空间的访问必须是互斥的（互斥访问通过操作系统提供的工具实现）。操作系统只负责提供共享空间和同步互斥工具（如PV操作）。

有关共享空间的申请，使用，撤销方法：<https://blog.csdn.net/gnosed/article/details/82491040>

共享存储分为两种：1.基于数据结构的共享2.基于存储区的共享

基于数据结构的共享这种方式要求诸进程公用某些数据结构，借以实现诸进程间的信息交换。这种方式共享速度慢、限制多，是一种低级通信方式。

基于存储区的共享：在内存中画出一块共享存储区，数据的形式、存放位置都由进程控制，而不是操作系统。相比之下，这种共享方式速度更快，是一种高级通信方式。

2.管道通信

“管道”是指用于连接读写进程中的一个共享文件，又名pipe文件。其实就是在内存中开辟一个大小固定的缓冲区。

（1）管道只能采用半双工通信，某一时间段内只能实现单向的传输。如果要实现双向同时通信，则需要设置两个管道。

（2）各进程要互斥地访问管道

（3）数据以字符流的形式写入管道。当管道写满时，写进程的write()系统调用将被阻塞，等待读进程将数据取走。当读进程将数据全部取走后，管道变空，此时读进程的read()系统调用将被阻塞

（4）如果没写满，就不允许读。如果没读空，就不允许写

（5）数据一旦被读出，就从管道中被抛弃，这就意味着读进程最多只能有一个，否则可能会有读错数据的情况

3.消息传递

进程间的数据交换以格式化的消息为单位。进程通过OS提供的“发送消息/接收消息”两个原语进行数据交换。

消息的结构包括消息头和消息体两部分。消息头包括：发送进程ID、接收进程ID、消息类型、消息长度等格式化的信息（计算机网络中发送的“报文”其实就是一种格式化的消息）

消息传递有两种方式：①直接通信方式，即消息直接挂到接收进程的消息缓冲队列上

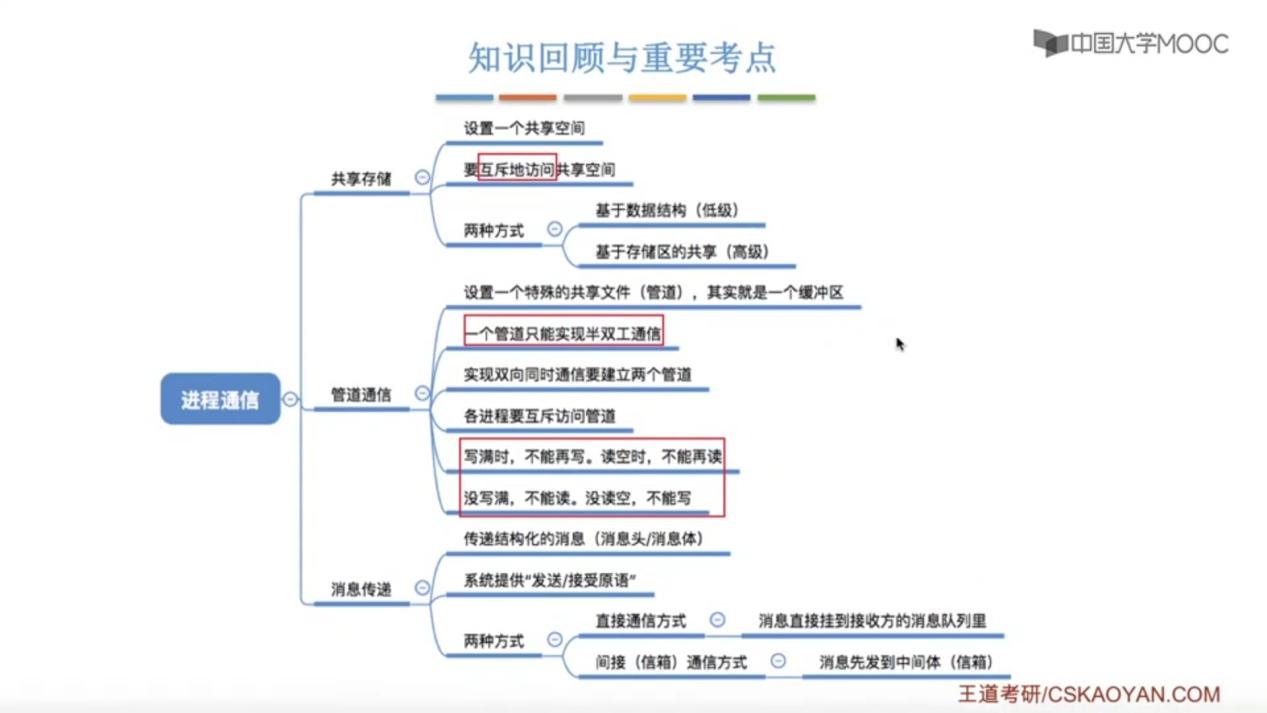
每个进程有一个消息缓冲队列。发送进程先创建格式化的消息体，然后使用发送原语发送给接收进程，消息就被挂到接收进程的消息队列的队尾。接收进程又用接收原语从消息队列队头依次取走消息。

消息队列创建、发送和接收：<https://blog.csdn.net/qq_27664167/article/details/81712887>

②间接通信方式，消息要先发送到中间实体（信箱）中，因此也称“信箱通信方式”。例如计算机网络中的电子邮件系统。

系统为信箱通信提供了若干条原语，用于信箱创建、撤销和消息的发送、接收

发送进程用发送原语将消息发送到信箱。接收进程再用接收原语将消息从信箱接收过来。



**2.1.5线程的概念**

**1.线程**

在没引入进程之前，系统中各个程序只能串行执行。引入了进程之后，进程之间可并发执行。但有的进程可能需要“同时”做很多事，而传统的进程只能串行执行一系列程序。为此，引入了“线程”，来增加并发度。

在传统进程机制中，进程是程序执行流的最小单位。引入了线程后，线程成为了程序执行流的最小单位。每个进程可有多个线程，线程间执行的代码不同，可并发地执行。线程是一个基本的CPU执行单元，也是程序执行流的最小单位。引入线程之后，不仅是进程之间可以并发，进程内的各线程之间也可以并发，从而进一步提升了系统的并发度。使得一个进程内也可以并发处理各种任务（如QQ进程同时并发执行视频聊天、文字聊天、文件传送等功能）

引入线程后，进程只作为除CPU之外的系统资源的分配单元。

**2.引入线程机制后带来的变化**

从资源分配、调度的角度来看：传统进程机制中，进程既是资源分配的基本单位，也是调度的基本单位。引入线程后，进程只是资源分配的基本单位，线程是调度的基本单位。

从并发性的角度来看：传统进程机制中，只能进程间并发。引入线程后，各线程间也能并发，提升了并发度。

从系统开销的角度来看：传统的进程间并发，需要切换进程的运行环境，系统开销很大。引入线程后，如果是同一进程内的线程切换，则不需要切换进程环境，系统开销小。所以并发所带来的系统开销减少。

**3.线程的属性**

1.线程是处理机调度的单位

2.多CPU计算机中，各个线程可占用不同的CPU

3.每个线程都有一个线程ID，和线程控制块TCB（用于管理线程的数据结构）

4.线程也有就绪、阻塞、运行三种基本状态

5.线程几乎不拥有系统资源

6.同一进程的不同线程间共享进程的资源

7.由于共享地址空间，同一进程的线程间通信甚至无需系统干预

8.同一进程中的线程切换，不会引起进程切换

9.不同进程中的线程切换，会引起进程切换

10.切换同进程内的线程，系统开销很小

11.切换进程，系统开销很大

**2.1.6多线程模型**

1.用户级线程

早期的操作系统如早期的Unix只支持进程，不支持线程，当时的“线程”是由线程库实现的，操作系统只能看到进程。

例如QQ进程如果想同时视频聊天，文字聊天和文件传输并发运行。在不支持线程的情况下，可以分别建立三个进程分别处理三件事情。

//进程1处理视频聊天 //进程2处理文字聊天 //进程3处理文件传输

int main(){ int main(){ int main(){

while(true){ while(true){ while(true){

处理视频聊天的代码； 处理文字聊天的代码； 处理文件传输的代码；

} } }

} } }

三段代码都是用while循环来实现持续处理。也可以用下面的方式将它们合在一起并发运行

int main(){

int i = 0;

while(true){

if (i==0) {处理视频聊天的代码；}

if (i==1) {处理文字聊天的代码；}

if (i==2) {处理文件传输的代码；}

i = (i+1) % 3;

}

}

从代码的角度，线程其实就是一段代码逻辑。上述三段代码逻辑上可以看作三个“线程”，while循环就是一个最简单的“线程库”，线程库完成了对线程的管理工作。

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。

这种用户级线程的管理是由应用程序通过线程库完成的，所有的线程管理工作都由应用程序负责。这种线程的切换不需要CPU由用户态切换到内核态，无需操作系统干预，在用户态下即可完成。操作系统不能意识到用户级线程的存在，只能看到一个进程。

“用户线线程”就是“从用户视角能看到的线程”

这种方式的优点：用户级线程的切换在用户空间即可完成，不需要切换到核心态，线程管理的系统开销小，效率高

缺点：当一个用户级线程被阻塞后，整个进程都会阻塞，并发度不高。多个线程也不可能在多核处理机上并行运行，不能发挥多核的优势。

2.内核级线程

大多数现代操作系统都实现了内核级进程，如Windows、Linux。

内核级线程的管理工作由操作系统内核完成。线程调用、切换等工作都由内核负责，因此内核级线程的切换必然需要在核心态下才能完成。操作系统会为每个内核级线程建立相应的TCB。通过TCB对线程进行管理。“内核级线程”就是“从操作系统内核视角能看到的线程”。

这种方式的优点：当一个线程被阻塞后，别的线程还可以继续执行，并发能力强。多线程可在多核处理机上并行执行，能充分发挥多核的优势。

缺点：一个用户进程会占用多个内核级线程，线程切换由操作系统内核完成，需要切换到核心态。因此线程管理的成本高，开销大。

3.多线程模型

在支持内核级线程的系统中，如果再引入线程库，可以实现若干用户级线程映射到若干内核级线程。根据用户级线程和内核级线程的映射关系，可以划分为几种多线程模型

（1）一对一模型：一个用户级线程映射到一个内核级线程。每个用户进程有与用户级线程同数量的内核级线程。

优点：当一个线程被阻塞后别的线程还可以继续执行，并发能力强，多线程可在多核处理机上并行执行

缺点：一个用户进程会占用多个内核级线程，线程切换由操作系统内核完成，需要切换到核心态，因此线程管理的成本高，开销大。

（2）多对一模型：多个用户级线程映射到一个内核级线程，且一个进程只被分配一个内核级线程。

优点：用户级线程的切换在用户空间即可完成，线程管理的系统开销小，效率高。

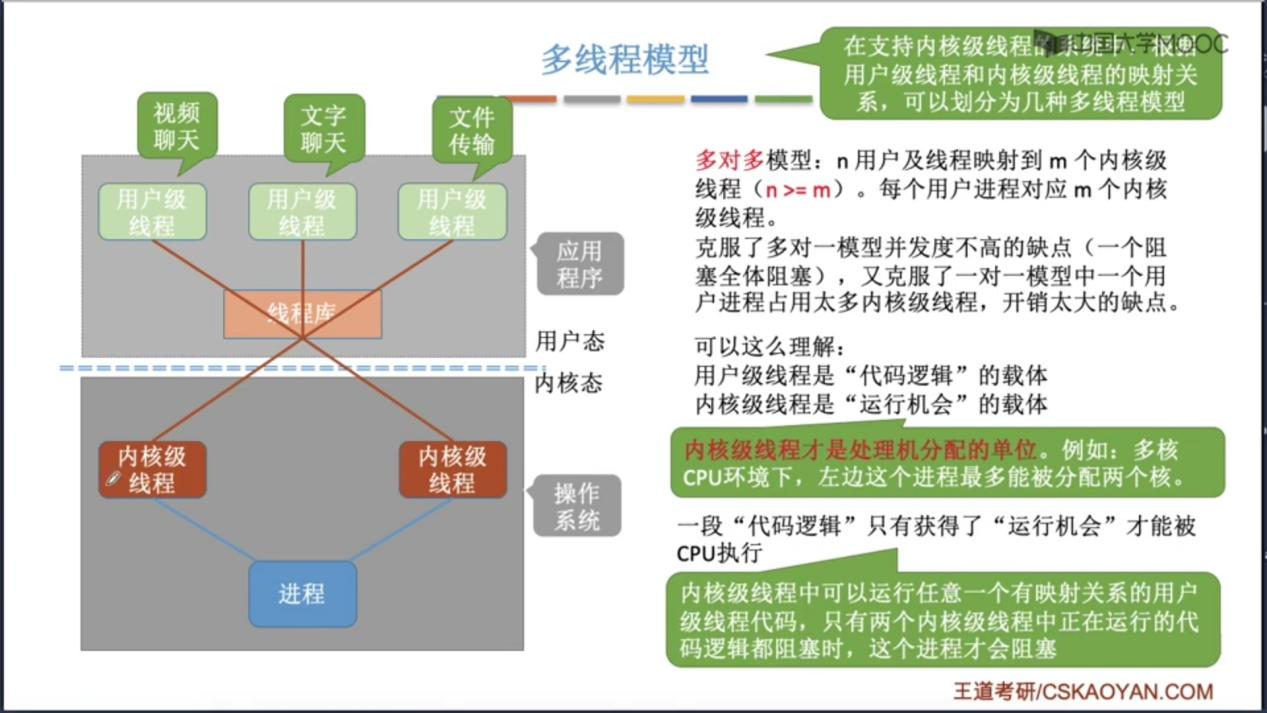
缺点：当一个用户级线程被阻塞后，整个进程都会被阻塞，并发度不高。多个线程不可在多核处理机上并行运行。

**操作系统只“看得见”内核级线程，因此只有内核级线程都是处理机分配的单位。**

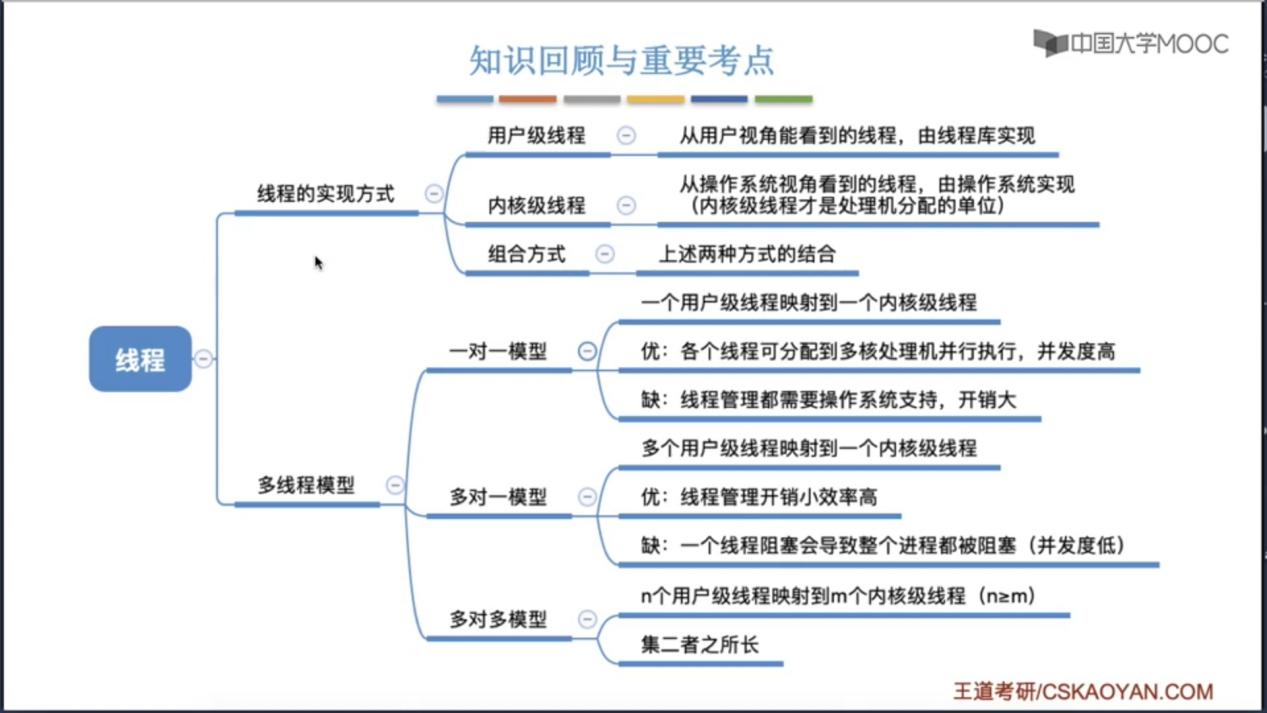
（3）多对多模型：n用户级线程映射到m内核级线程上(n>=m)，每个用户进程对应m个内核级线程

克服了多对一模型并发度不高的缺点，又克服了一对一模型中一个用户进程占用太多内核级线程，开销太大的缺点。

用户级线程可以理解为“代码逻辑”的载体，内核级线程是“运行机会”的载体。一段代码逻辑只有获得了运行机会才能被CPU执行。内核级线程中可以运行任意一个有映射关系的用户级线程代码，只有进程映射的所有内核线程中正在运行的代码逻辑都阻塞时，整个进程才会阻塞。



本节小结



**2.2.1调度的概念、层次**

**1.调用的三个层次**

（1）高级调度

作业的概念：一个具体的任务

内存空间是有限的，有时无法将用户提交的作业全部放入内存。高级调度（作业调度）是指按一定的原则从外在的作业后备队列中挑选一个作业调入内存，并创建进程。每个作业只调入一次，调出一次。作业调入时会建立PCB，调出时才撤销PCB。

（2）低级调度（进程调度/处理机调度）

按某种策略从就绪队列中选取一个进程，将处理机分配给它。

进程调度是操作系统中最基本的一种调度，在一般的操作系统中都必须配置进程调度。进程调度的频率很高，一般几十毫秒一次。

（3）中级调度（内存调度）

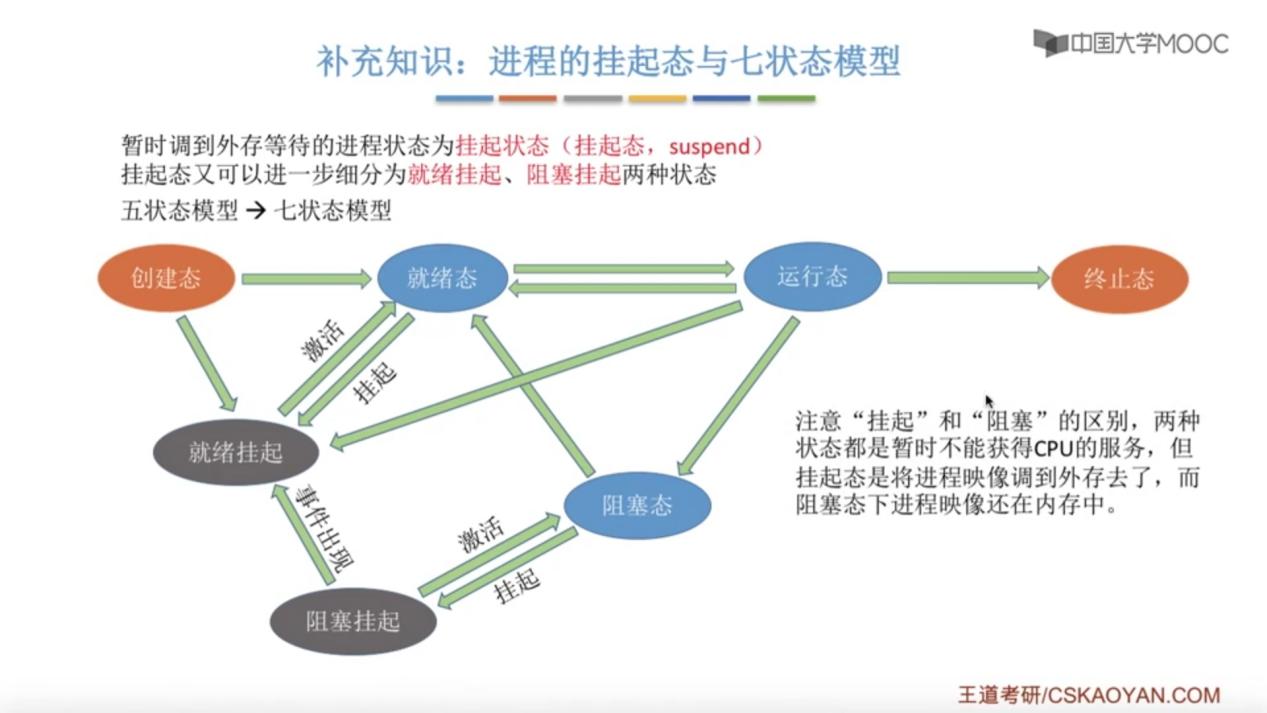
内存不够时，可将某些进程的数据调出外存。等内存空闲或进程需要运行时再重新调入内存。

暂调到外存等待的进程为挂起状态。被挂机的进程PCB会被组织成挂机队列。

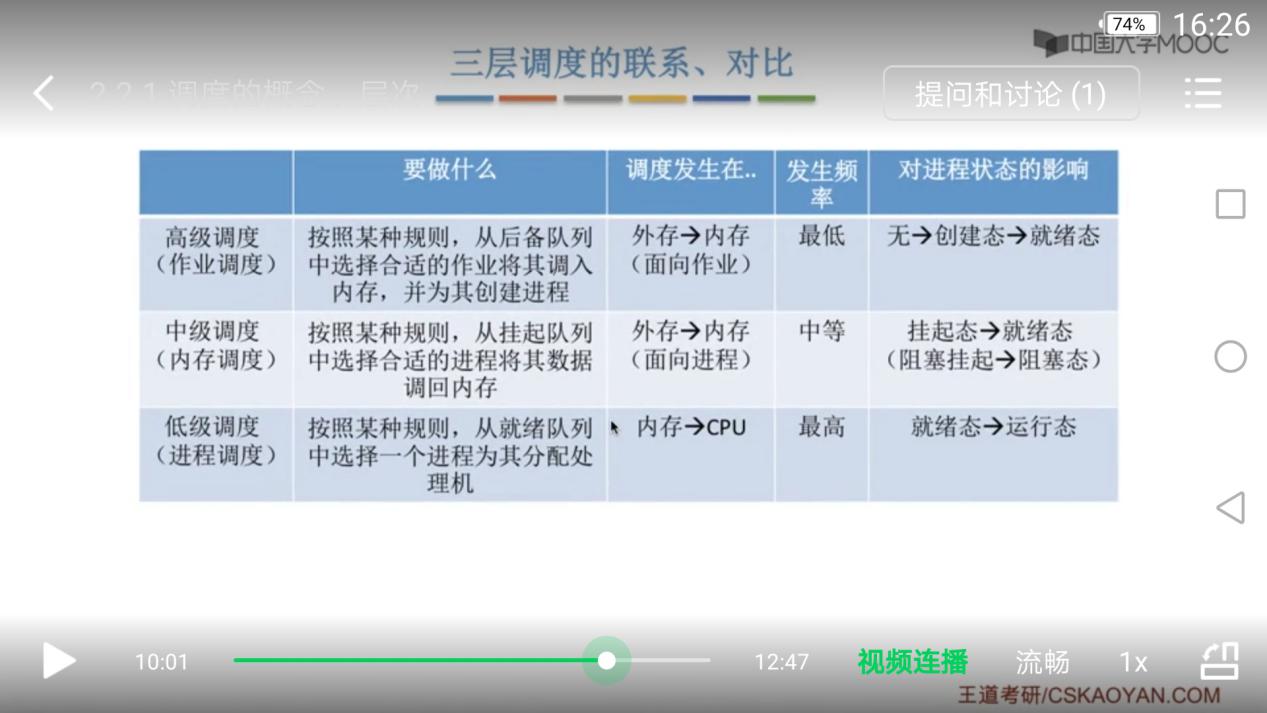
中级调度是指按某种策略决定将哪个处于挂机状态的进程重新调入内存。一个进程可能会多次调出，调入内存。因此中级调度发生的频率要比高级调度更高。

**2.七状态模型**

暂时调到外存等待的进程状态为挂起状态。挂起态又可以进一步细分为就绪挂起、阻塞挂机状态。“挂起”和“阻塞”两种状态都是暂时不能获得CPU服务。但挂起态是将进程映像调到外存去了，而阻塞态下进程映像还在内存中。



3.三层调度的联系、对比



**2.2.2 进程调度的时机、切换与过程、进程调度方式**

1.进程调度的时机

需要进行进程调度与切换的情况分为两大类：

（1）当前运行的进程主动放弃处理机

进程正常终止、运行过程中发生异常而终止、进程主动请求阻塞（如等待I/O）

（2）当前运行的进程被动放弃处理机

分给进程的时间片用完、有更紧急的事需要处理（如I/O中断）、有更高优先级的进程进入就绪队列

不能进行进程调度与切换的情况

（1）在处理中断的过程中。中断处理过程复杂，与硬件密切相关，很难做到在中断处理过程中进行进程切换。

（2）进程在操作系统内核程序临界区中

临界资源：一个时间段内只允许一个进程使用的资源，各进程要互斥地访问临界资源。

临界区：访问临界资源的那段代码。

内核程序临界区一般是用来访问某种内核数据结构的，比如进程的就绪队列。当进程处于内核程序临界区访问就绪队列时，就绪队列要上锁。如果进程还没退出临界区就进行调度，但调度程序也需要访问就绪队列，就无法顺利进行进程调度。所以内核程序临界区访问的临界资源如果不尽快释放的话，极有可能影响到操作系统内核的其他管理工作。因此在访问内核程序临界区期间不能进行调度与切换。

但进程在访问普通的临界区时是可以进行调度的。比如进程使用了打印机，在打印完成之前进程一直处于临界区内，临界资源不会解锁。此时应允许进程调度使CPU不会一直空闲。普通临界区访问的临界资源不会直接影响操作系统内核的管理工作。

（3）在原语（原子操作）中，原子操作不可中断，要一气呵成。

有的系统中，只允许进程主动放弃处理机。有的系统中，进程可以主动放弃处理机，也会在有更紧急任务时被剥夺处理机。

2.进程调度方式

非剥夺调度方式，又称非抢占方式。即，只允许进程主动放弃处理机。在运行过程中即使有更紧迫的任务到达，当前进程依然会继续使用处理机，直到其终止或主动要求进入阻塞态。

剥夺调度方式，又称抢占方式。当一个进程正在处理机上执行时，若有更重要的进程需要使用处理机，则立即暂停正在执行的进行，将处理机分配给更重要紧迫的那个进程。

非抢占方式的系统实现简单，系统开销小，但无法及时处理紧急任务。适合于早期的批处理系统。

抢占方式的系统可以优先处理更紧急的进程，也可实现让各进程按时间片轮转的功能。适合于分时操作系统、实时操作系统。

3.进程的切换

狭义的进程调度指的是从就绪队列中选一个要运行的进程（可以是刚暂停的，也可以是另一个进程）

进程切换是指一个进程让出处理机，由另一个进程占用处理机的过程。

广义的进程调度包含了选择一个进程和进程切换两个步骤。

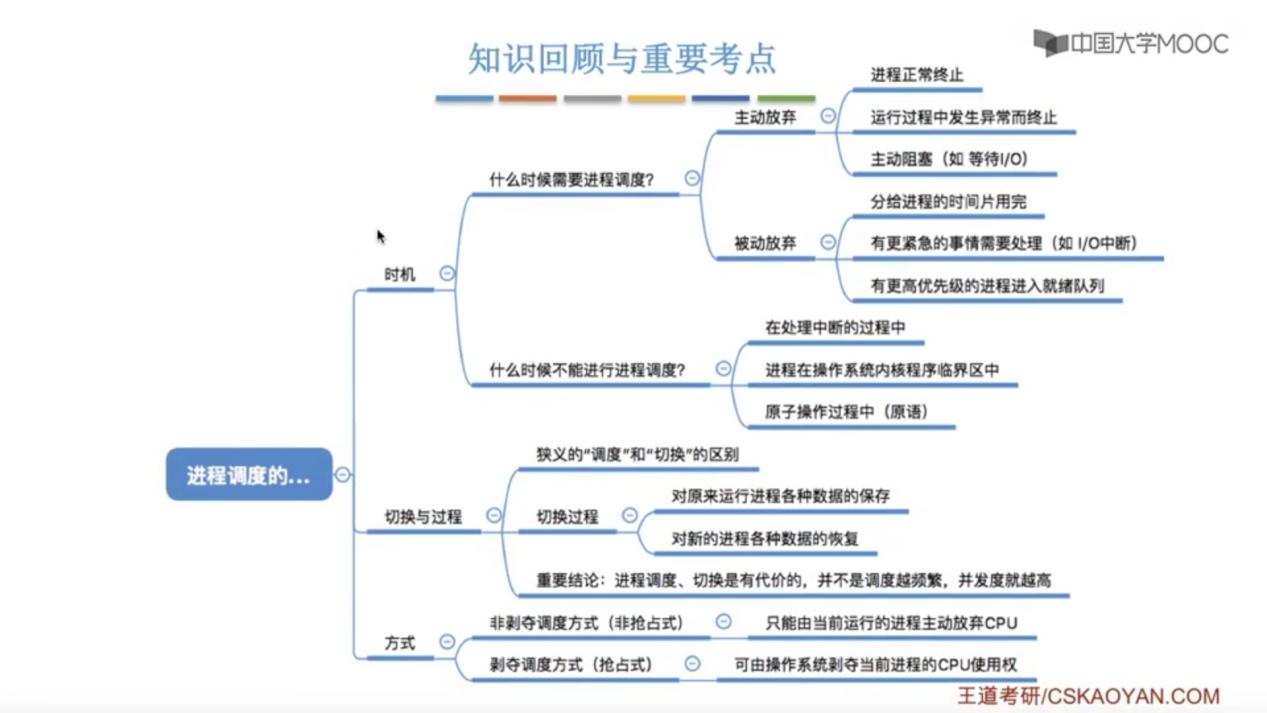
进程切换的过程主要完成了：

1.对原来运行进程各种数据的保存

2.对新的进程各种数据的恢复

如程序计数器、程序状态字、各种数据寄存器等处理机现场信息，这些信息保存在PCB中

进程的切换是有代价的，因此过于频繁的进行进程调度切换，必然会使整个系统的效率降低。



**2.2.3调度算法的评价指标**

1.CPU利用率：指CPU“忙碌”的时间占总时间的比例。利用率=忙碌的时间/总时间

多道程序并发执行的情况，可用甘特图求解。

2.系统吞吐量：单位时间内完成作业的数量。系统吞吐量=总共完成了多少道作业/总时间

3.周转时间：从作业被提交给系统开始，到作业完成为止的时间间隔。包括四部分：

（1）作业在外存后备队列上等待作业调度的时间

（2）进程在就绪队列上等待进程调度的时间

（3）进程在CPU上执行的时间

（4）进程等待I/O操作完成的时间

后三项在一个作业的整个处理过程中，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

对于操作系统来说，更关心系统的整体表现，因此更关心所有作业周转时间的平均值

带权周转时间 = 作业周转时间/作业实际运行的时间 （带权周转时间必然大于等于1）

带权周转时间更小，用户满意度更高

平均带权周转时间 = 各作业带权周转时间之和/作业数

4.等待时间

指进程或作业处于等待处理机状态时间之和。对于进程来说，等待时间就是指进程建立后等待被服务时间之和，在等待I/O完成的期间其实进程也是在被服务的，因此不计入等待时间。

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。

一个作业总共需被CPU、I/O设备服务多久一般是确定不变的，因此调度算法只会影响作业/进程的等待时间。当然，也有“平均等待时间”来评价整体性能。

5.响应时间

对于计算机用户来说，会希望自己的提交的请求（比如通过键盘输入了一个调试命令）尽早地开始被系统服务、回应。响应时间指从用户提交请求到首次产生响应所用的时间。

**2.2.4调度算法FCFS、SJF(SPF)、HRRN**

1.FCFS算法

算法思想：主要从“公平”角度考虑

算法规则：按作业或进程到达的先后顺序进行服务

用于作业调度时，考虑的是哪个作业先到达后备队列；用于进程调度时，考虑的是哪个进程先到达就绪队列

FCFS一般是非抢占式的算法

在计算等待时间的时候，如果一个进程又有计算又有I/O操作，那么等待时间就是周转时间-运行时间-I/O操作时间

优点：公平、实现简单

缺点：排在长作业后面的短作业需等待很长时间，对长作业有利，对短作业不利。

不会导致饥饿。（饥饿：某进程或作业长期得不到服务）

2.SJF(SPF)算法

算法思想：追求最少的平均等待时间，最少的平均周转时间，最少的平均带权周转时间

算法规则：最短的作业或进程先得到服务

可用于作业调度或进程调度。对于作业调度，称为SJF算法，对于进程调度，称为SPF算法。

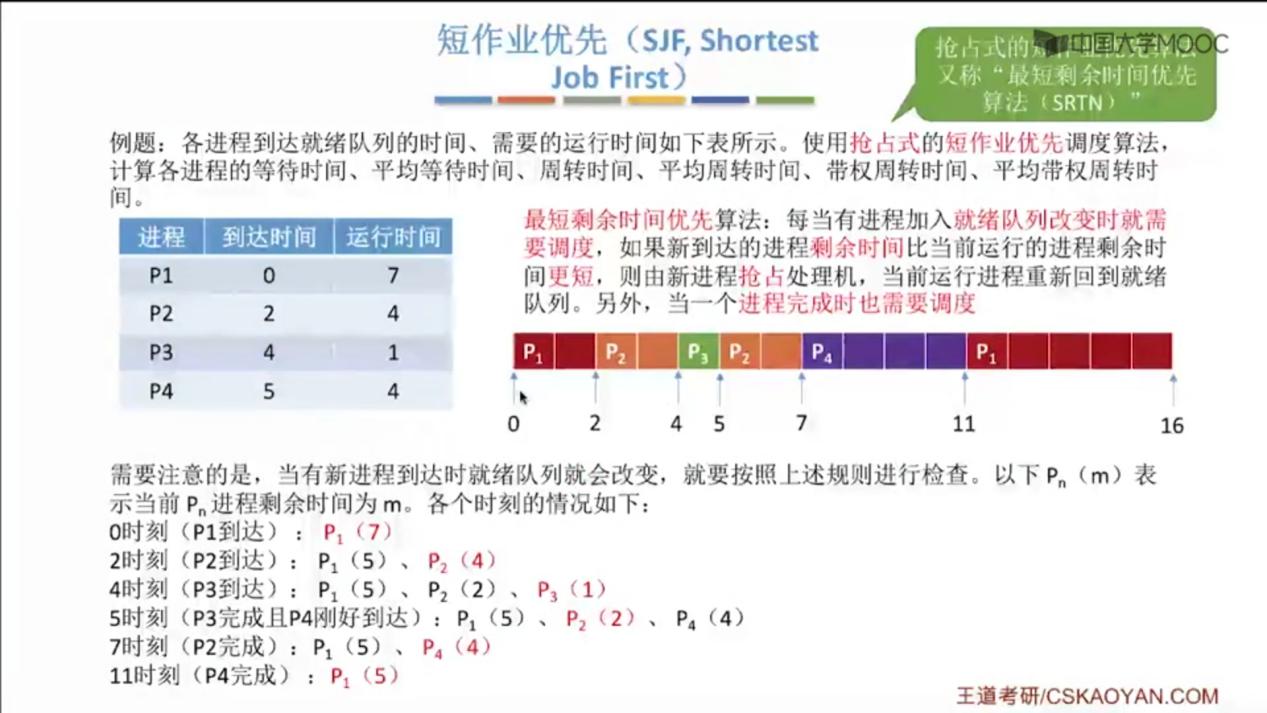
SJF(SPF)是非抢占式的算法，但也有抢占式的版本，即最短剩余时间优先算法(SRTN)

每次调度时选择当前已到达且运行时间最短的作业或进程

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

优点：“最短的”平均等待时间、平均周转时间

缺点：不公平，对短作业有利，对长作业有利，可能产生饥饿现象。另外，运行时间是由用户提供的，并不一定真实。



如题目中未特别说明，SJF算法默认是非抢占的。但如果说“SJF算法的平均等待时间、平均周转时间最少”，这个说法是错误的不严谨的。SRTN算法的平均等待时间平均周转时间更少。

但相比于其它算法，SJF算法依然可以获得较少的平均等待时间，平均周转时间。所以在选择题中如果遇到“SJF算法的平均等待时间、平均周转时间最少”的选项，如果没有更合适的选项，可以选择该选项。

3.高响应比优先算法

响应比=（已等待时间+要求服务的时间）/要求服务的时间

算法思想：要综合考虑作业或进程的等待时间和要求服务的时间

算法规则：在每次调度时，先计算各个作业的响应比，选择响应比最高的作业为其服务

既可以用于作业调度，也可以用于进程调度。

一般来说是非抢占式的，只有进程主动放弃处理机时才需调度，才需计算响应比。

优点：综合考虑了等待时间和运行时间。等待时间相同时，要求服务时间短的优先。要求服务时间相同时，等待时间长的优先。对于长作业来说，随着等待时间越来越久，其响应比越来越大，从而避免了长作业饥饿的问题。

这几种算法主要关心评价系统整体性能的指标，但不关心“响应时间”，也不区分任务的紧急度。因此交互性很糟糕，适用于早期的批处理系统。

**2.2.5时间片轮转算法、优先级调度算法、多级反馈队列调度算法**

1.时间片轮转算法

算法思想是公平轮流地为各进程服务，让每个进程都能在一定时间时隔内得到响应

算法规则：按各进程到达就绪队列的顺序、轮流让各个进程执行一个时间片，若进程没在一个时间片内执行完，则会剥夺处理机，将进程重新放在就绪队列队尾重新排队。

用于进程调度，是属于抢占式的算法。由时钟装置发出时钟中断来通知CPU时间已到。常用于分时系统，更注意“响应时间”。

如果同一时间有进程时间片用完，另一新进程刚好到达，除非特殊说明，默认新进程先进入就绪队列。如果一个进程的时间片没用完就执行完毕，会立即主动放弃处理机引发调度。

当时间片太大时，使每个进程都能在一个时间片内完成，则算法退化为FCFS算法，并且增大了进程的响应时间，因此时间片不能太大。

当时间片太小，进程的切换又过于频繁，系统会花大量时间进行切换。可见时间片也不能太小。

优点：公平，响应快。适用于分时操作系统。不会引起饥饿。

缺点：高频切换会有一定开销，也不区分任务紧急度。

2.优先级调度算法

算法思想：实时操作系统出现后，越来越多的应用场景需要根据任务的紧急程度来决定处理顺序

算法规则：每个作业或进程有各自的优先级，调度时选择优先级最高的作业或进程

可用于作业调度和进程调度，甚至可用于I/O调度。

抢占式和非抢占式都有。非抢占式只在进程主动放弃处理机时调度，抢占式还需在就绪队列变化时检查是否会发生抢占。

根据优先级是否可以动态改变，可将优先级分为静态优先级和动态优先级两种。

静态优先级：创建进程时确定，之后一直不变

动态优先级：创建进程时有一个初始值，之后会根据情况动态地调整优先级

通常，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繁忙型进程（可以更早让I/O设备投入工作，I/O设备可与CPU并行工作，有助于提升资源利用率和系统吞吐量）

如果某进程在就绪队列中等待了很长时间，可适当提升其优先级。如果某进程占用处理机运行了很长时间，可适当降低其优先级。

优点：用优先级区分紧急程度，适用于实时操作系统，可灵活地调整对各作业的偏好程度

缺点：若源源不断地有高优先级进程到来，则可能导致饥饿

3**.多级反馈队列调度算法**

**算法规则：（1）设置多级就绪队列，各级队列优先级从高到低，时间片从小到大**

**（2）新进程到达时先进入第1级队列，按FCFS原则排队等待被分配时间片，若用完时间片进程还未结束，则进程进入下一级队列队尾。如果已经在最下级的队列，则重新放回该队列队尾。**

**（3）只有第k级队列为空时，才会为k+1级队头的进程分配时间片**

该算法用于进程调度，是抢占式的算法。在k级队列的进程运行过程中，若更上级的队列中进入了一个新进程，则由于新进程处于优先级更高的队列中，新进程会抢占处理机。但**此时需将被抢占处理机的进程放回k级队列队尾**。

优点：对各类型进程相对公平，每个新到达进程都可以很快得到响应，短进程只用较少时间就可完成，不必事先估计进程运行时间，可灵活地调整对各类进程的偏好程度（例如可将I/O阻塞的进程重新放回原队列，这样I/O型进程就可保持较高优先级）

这种算法会导致饥饿。

这三种算法更注重系统的响应时间，适合用于交互式系统。UNIX使用的就是多级反馈队列调度算法。

**2.3.1进程同步、进程互斥**

1.进程同步

同步亦称直接制约关系，它是指为完成某种任务而建立的两个或多个进程，这些进程因为需要在某些位置上协调它们的工作次序而产生的制约关系。进程间的直接制约关系就是源于它们之间的相互合作。

例子：管道通信的实际应用中应保证写操作在前，读操作在后

2.进程互斥

我们把一个时间段内只允许一个进程使用的资源称为临界资源。许多物理设备（如摄像头、打印机）都属于临界资源。此外还有许多变量、数据、内存缓冲区等都属于临界资源。

对临界资源的访问，必须互斥地进行。互斥，亦称间接制约关系。进程互斥，指当一个进程访问某临界资源时，另一个想要访问该临界资源的进程必须等待。当前访问临界资源的进程访问结束，释放该资源之后，另一个进程才能去访问临界资源。

对临界资源的互斥访问的代码，可以在逻辑上分为如下四部分：

do{

entry section; //进入区，检查进程能不能现在进入临界区，如果能，则设置正在访问临界资源的标志（对资源上锁）阻止其它进程同时进入临界区

critical section; //临界区 ，访问临界资源的那段代码

exit section; //退出区 ，负责解除正在访问临界资源的标志（对资源解锁）

remainder section; //剩余区，做其它处理

}while(True)

临界区，也称临界段，是进程中访问临界资源的代码段。

进入区和退出区是负责实现互斥的代码段。

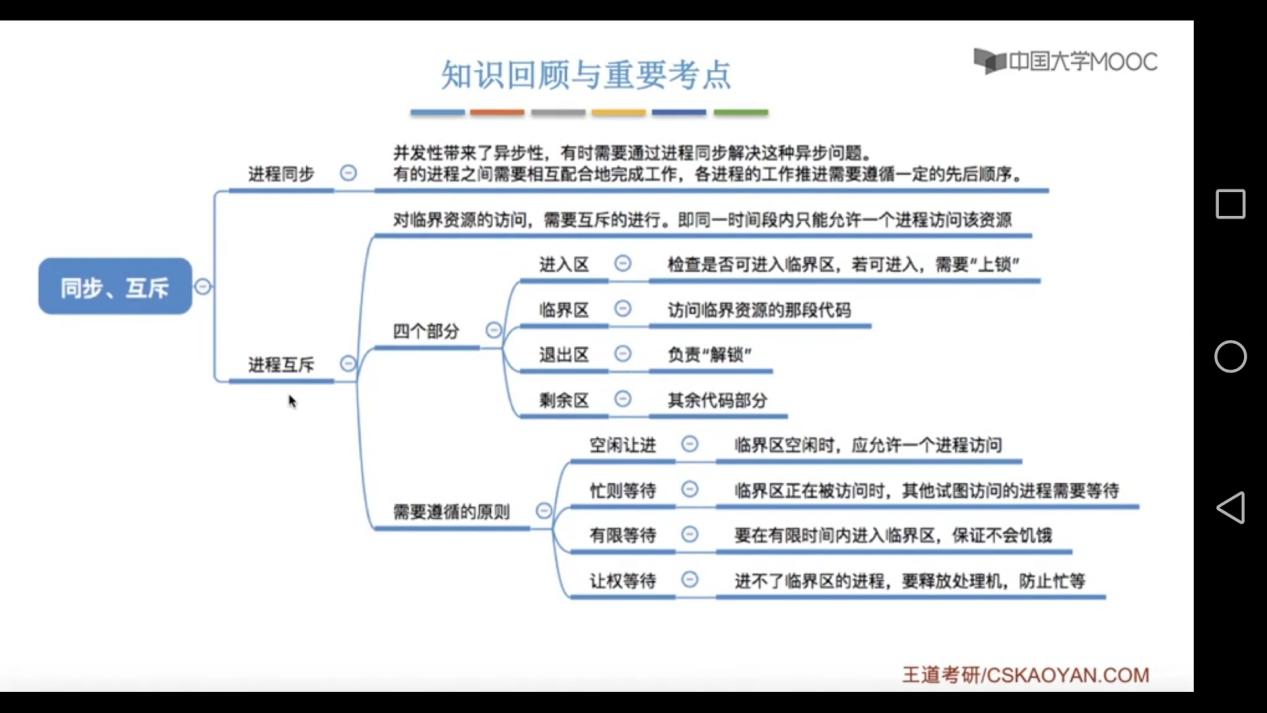
3.实现临界资源互斥访问的原则

（1）空闲让进：临界区空闲时，可以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：当已有进程进入临界区时，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（不会饥饿）

（4）让权等待：当进程不能进入临界区时，应立即释放处理机，防止进程忙等待



**2.3.2进程互斥的软件实现方法**

1.单标志法

算法思想：两个进程在访问完临界区后会把使用临界区的权限转交给另一个进程。也就是说每个进程进入临界区的权限只能被另一个进程赋予。

int turn = 0;//表示当前允许进入临界区的进程号

P0进程：

while(turn!=0);

critical section;

turn = 1;

remainder section;

P1进程：

while(turn!=1);

critical section;

turn = 0;

remainder section;

注：turn的初值为0，若P1先上处理机运行，会一直卡在while（turn!=1）这一步，直到其时间片用完，发生调度，切换P0上处理机运行。P0不会卡住，在P0访问临界区期间即时切换回P1，P1依然会卡在while(turn!=1)这一步。只有P0在退出区将turn改为1后，P1才能进入临界区。

因此该算法可以实现“同一时刻最多只允许一个进程访问临界区”。

但该算法只能按p0->p1->p0->p1这样轮流访问，这种必须轮流访问带来的问题是，如果此时允许进入临界区的进程是P0，而P0一直不访问临界区，那么虽然此时临界区空闲，也不允许P1访问。因此，单标志法的最大问题是：违反了“空闲让进”的原则。

2.双标志先检查法

算法思想：设置一个布尔型数组flag[]，数组中各个元素用来标记各进程想进入临界区的意愿，比如“flag[0]=true”意味着0号进程P0现在想要进入临界区。每个进程在进入临界区之前先检查当前有没有别的进程想进入临界区，如果没有，则把自身对应的标志flag[i]设为true，之后开始访问临界区。

bool flag[2];

flag[0]=false; flag[1]=false;//刚开始两个进程都不想进入临界区

P0进程：

while(flag[1]); ① //如果P1想进入临界区，P0就一直循环等待

flag[0]=true; ② //标记P0想进入临界区

critical section; ③

flag[0]=false; ④ //临界区访问完，标记P0不想使用临界区

remainder section;

P1进程：

while(flag[0]);⑤

flag[1]=true;⑥

critical section;⑦

flag[1]=false;⑧

remainder section;

这个算法的主要问题是违反了“忙则等待”的原则：如果P0和P1并发执行，当P0执行完①跳出循环后（此时P0的PC寄存器内容是②）时间片刚好用完，立刻切换到了P1，P1执行了⑤⑥⑦，然后时间片用完又切回P0,P0执行②③，则P0和P1同时访问了临界区。

原因在于，进入区的“检查”和“上锁”两个处理不是一气呵成的。“检查”后，“上锁”前可能发生进程切换。

3.双标志后检查法片

算法思想：双标志先检查法的改版。前一个算法的问题是先“检查”后“上锁”，但这两个操作又无法一气呵成，因此导致两个进程同时进入临界区的问题。因此，人们又想到先“上锁”后“检查”的方法来避免上述问题。

bool flag[2];

flag[0]=false;

flag[1]=false;

P0进程：

flag[0]=true;①

while(flag[1]);②

critical section;

flag[0]=false;

remainder section;

P1进程：

flag[1]=true;⑤

while(flag[0]);⑥

critical section;

flag[1]=false;

remainder section;

这个先上锁后检查的方法虽然解决了“忙则等待”的问题，但又违背了“空闲让进”和“有限等待”原则，会因各进程都长期无法访问临界资源而产生“饥饿”现象：如果按①⑤②⑥的顺序执行，则P0和P1将都无法进入临界区，都被while循环卡死。

4.Peterson算法

算法思想：结合双标志法、单标志法的思想。如果双方都争着想进入临界区，那可让进程尝试“谦让”。

bool flag[2]; //用来表达访问临界区的“意愿”

int turn = 0; //用来表达“谦让”

P0进程：

flag[0]=true; ① //表达P0想进入临界区

turn = 1; ② //试图谦让，表达可以让对方P1先使用临界资源

while(flag[1] && turn == 1); ③//检查对方是否有意愿且最后是自己谦让了，如果是就//卡在这里

critical section; ④

flag[0]=false; ⑤

remainder section;

P1进程：

flag[1]=true; ⑥

turn = 0; ⑦

while (flag[0] && turn == 0); ⑧

critical section; ⑨

flag[1]=false; ⑩

remainder section;

如果按①⑥②⑦⑧③的顺序执行，将由于⑦在②之后执行，turn的值为0，即是P1最后表达了谦让，则P0一定能跳过③的循环进入临界区，同时P1会卡在⑧的循环。反之如果按①⑥⑦②⑧③的顺序执行，即turn的值在②执行后值为1，即P0最后表达了谦让，则P1能跳过循环而P0被循环卡住。如果按①⑥⑦⑧②③的顺序执行，虽然第一次P1会被⑧卡住，P0会被③卡住，但当P1再被调度时由于已执行了②，turn最后是1，所以第二次P1可以跳过循环⑧进入临界区，到达⑩之后P0又可以跳过循环③。每个进程的进入区做了三件事：

（1）主动争取（2）主动谦让（3）检查对方意愿和最后是否是自己谦让。谁最后谦让，谁就失去了行动的优先权。

因此Peterson算法满足了“空闲让进”、“忙则等待”、“有限等待”的原则。但未满足“让权等待”的原则。因为不能进入临界区的进程是卡在while循环里，并未让出CPU。所以它相较于之前的三个算法是最好的，但还不够好。

**2.3.3进程互斥的硬件实现方法**

1.中断屏蔽方法

利用“开/关中断指令”实现。与原语的实现思想相同，在某进程开始访问临界区到结束访问为止都不允许被中断，也就不能发生进程切换，因此也不可能发生两个进程同时访问临界区的情况。

伪代码：

...

关中断；

临界区；

开中断；

...

优点：简单高效

缺点：不适用于多处理机（关中断指令只对执行了关中断指令的那个处理机有用，其他处理机还是会正常切换进程，可能导致两个处理机上的两个进程同时进入临界区），只适用于操作系统内核进程，不适用于用户进程（开关中断两条指令是特权指令，只能运行在内核态）

2.TestAndSet指令，简称TS指令，TSL指令

TSL指令是用硬件实现的，执行过程不许被中断，只能一气呵成。用一个共享变量lock表示当前临界区是否被加锁（lock存在一个寄存器中），每次调用TSL指令时，都将lock置为true并返回lock之前的值。这样就可以使用该指令实现互斥的逻辑：

while(TestAndSet(&lock));

critical section;

lock = false;

remainder section;

当lock初始为false时，TestAndSet返回的结果是false并将lock置为true(这两件事是一气呵成的），所以进程可以跳过while循环进入临界区。当此时再有进程试图访问，lock为true,TS指令返回true,便被卡在while循环里，直到正在访问临界资源的进程访问完毕，将lock重置为false。此时其他进程可以跳过while循环进入临界区。

优点：实现简单，无需像软件实现方法那样严格检查是否会有逻辑漏洞，适用于多处理机环境。

缺点：不满足“让权等待”原则，暂时无法进入临界区的进程会占用CPU并循环执行TSL指令，从而导致“忙等”

3.Swap指令，有的地方也叫Exchange指令，XCHG指令

Swap指令是用硬件实现的，执行过程不许被中断，只能一气呵成。

bool old = true;

while (old == true)

Swap(&lock,&old);

critical section;

lock = false;

remainder section;

逻辑上来看Swap和TSL指令并无太大区别，都是先记录下此时临界区是否已经被上锁，再将上锁标记lock设置为true，最后检查old,如果old为false则说明之前没有别的进程对临界区上锁，即可跳出循环，进入临界区

优点和缺点均和TSL指令相同。实现简单，无需检查漏洞，适用于多处理机环境。但会导致“忙等”。

**2.3.4信号量机制**

之前解决进程互斥访问的方法中,单标志法无法实现“空闲让进”，双标志后检查法无法实现“空闲让进”和“有限等待”，双标志先检查法由于检查和上锁不能一气呵成，导致两个进程都能同时进入临界区。而Peterson算法、中断屏蔽法、TS指令和Swap指令法虽能实现互斥访问，但无法实现“让权等待”。

1965年，Dijkstra提出了有效实现进程互斥、同步的方法--信号量机制

1.信号量机制

信号量机制分为整型信号量和记录型信号量

用户进程可通过操作系统提供的一对原语来对信号量进行操作，从而很方便的实现了进程互斥、进程同步。

信号量其实就是一个变量（可以是一个整数，也可以是更复杂的记录型变量），可以用一个信号量来表示系统中某种资源的数量。比如系统中只有一台打印机，就可以设置一个初值为1的信号量。

原语是一种特殊的程序段，其执行不可中断。因此如果能把进入区、退出区的操作都用“原语”实现，这些操作能一气呵成就能避免问题。

这对原语是wait(s)原语和signal(s)原语，括号里的信号量s其实就是一个参数。

wait和signal原语常简称为P、V操作。可以把wait(s)写为P(s),signal(s)写为V(s).

2.整型信号量

用一个整数型变量作为信号量，用来表示系统中某种资源的数量

int S=1;

void wait(int S){

while(S<=0); //如果资源数量不够，就卡在此处等待

S = S-1; //如果资源数量足够，就占用一个资源

}

void signal(int S){

S = S+1;

}

P0进程：

...

wait(S);

使用资源；

signal(S);

...

因为是用原语进行检查和上锁，避免了并发、异步导致的问题。

存在的问题：不满足“让权等待”原则，会发生“忙等”。

3.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); //如果剩余资源数不足，使用block原语使进程阻塞

//插入信号量S的等待队列中

}

进程使用资源完毕后，通过signal原语释放资源

void signal(semaphore S){

S.value++；

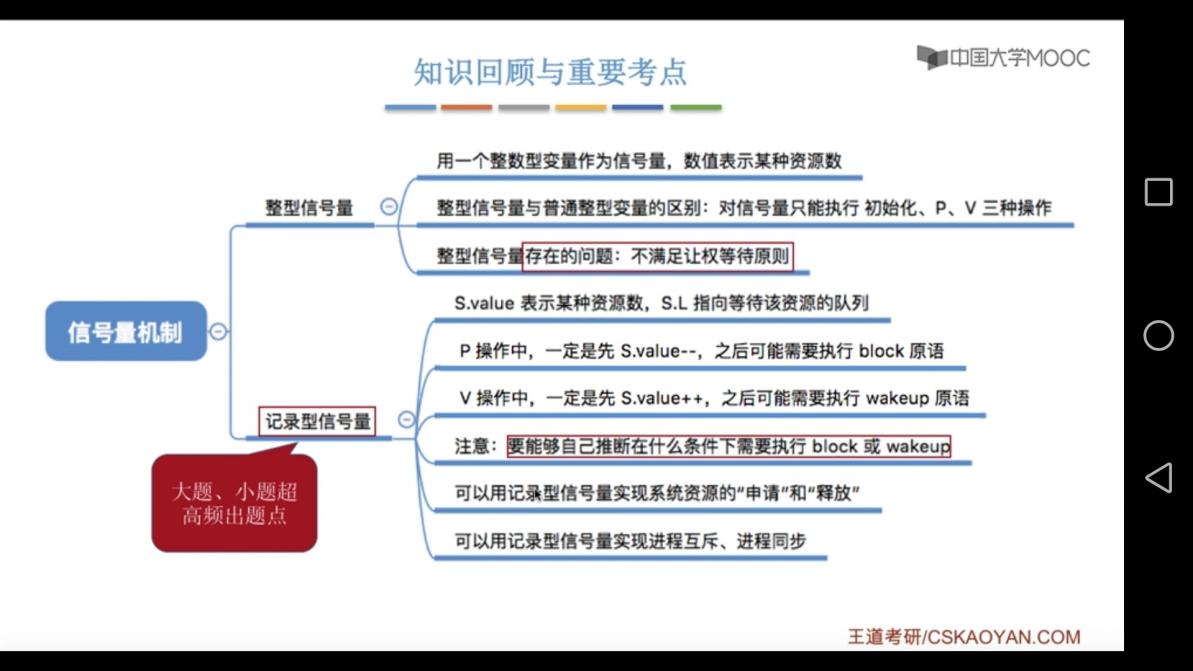
if (S.value<=0) wakeup(S.L);//释放资源后若有别的进程等待资源，使用wakeup原语唤醒一个进程,从阻塞态变成就绪态，再把资源分配给它

}

当S.value=0时，资源恰好分配完。此时若再有进程使用wait原语，S.value的值将变为-1，表明有1个进程在等待。若还有其它进程继续使用wait原语，S.value的值将变为-2,表明有2个进程在等待。因此S.value的相反数表示正在等待资源的进程数。

每次使用资源完毕后的进程释放资源时，会使S.value加1，若S.value仍小于等于0，就把等待队列中的一个进程唤醒，并把资源分配给这个进程。若S.value大于0了，说明等待队列中已经没有进程了。

在题目中，wait(S)、signal(S)也可记为P(S)、V(S),这对原语可用于实现系统资源的“申请”和“释放”。S.value的初值表示系统中某种资源的数目。对S的一次P操作表示进程请求一个单位的资源，因此需将S.value减1。当S.value<0时，表示该类资源已经分配完毕，进程应调用block原语进行自我阻塞，主动放弃处理机并插入该资源的等待队列。由此可见，该机制实现了“让权等待”原则，不会出现“忙等”现象。对S的一次V操作表示进程释放一个单位的资源，因此需将S.value加1。若加1后仍是S.value<=0，表示依然有进程在等待资源，因此应调用wakeup原语唤醒等待队列中的第一个进程。



**2.3.5信号量机制实现互斥、同步、前驱关系**

1.信号量机制实现互斥

方法：

（1）分析并发进程的关键活动，划定临界区

（2）设置互斥信号量mutex，**初值为1**

（3）在进入区使用P操作申请资源，在退出区使用V操作释放资源

semaphore mutex = 1;//mutex表示剩余的进入临界区的名额数，无特殊说明可以这样简写

P1(){

P(mutex);

critical section;

V(mutex)

}

P2(){

P(mutex);

critical section;

V(mutex);

}

对不同的临界资源需要不同的互斥信号量。PV操作必须成对出现。缺少P操作不能保证临界资源被互斥访问，缺少V操作会导致资源永不被释放，等待进程永不被唤醒

2.信号量机制实现同步

进程同步:要让各并发进程按要求有序推进

方法：

（1）分析什么地方需要实现“同步关系”，即保证一前一后执行的两个操作

（2）设置同步信号量S,**初值为0**

（3）在“前面的操作”之后执行V(S),在“后面的操作”之前执行P(S)。**前V后P。**

semaphore S=0;//初始化同步信号量，初始为0

P1(){

...

代码A;

V(S);

...

}

P2(){

...

P(S);

代码B;

...

}

S初始化为0，意味着这种“资源”刚开始是没有的。要想P2进程执行代码B,必须先由P1进程执行代码A完毕后用V操作释放一个“资源”。这就实现了P1的代码A先于P2的代码B。若先执行到P1的V操作，则S++后S=1,之后执行到P2的P(S)操作时，由于S=1表示有可用“资源”，会执行S--且不会阻塞P2。若先执行到P2的P(S)操作，由于初始时S=0，S--后S=-1，因此P2中P操作会执行block原语主动请求阻塞。当执行完P1的V操作，S++使S又变回0，由于此时有进程P2在S对应的阻塞队列中，V操作会执行wakeup原语唤醒P2。P2再次被调度时就可以继续执行代码B。

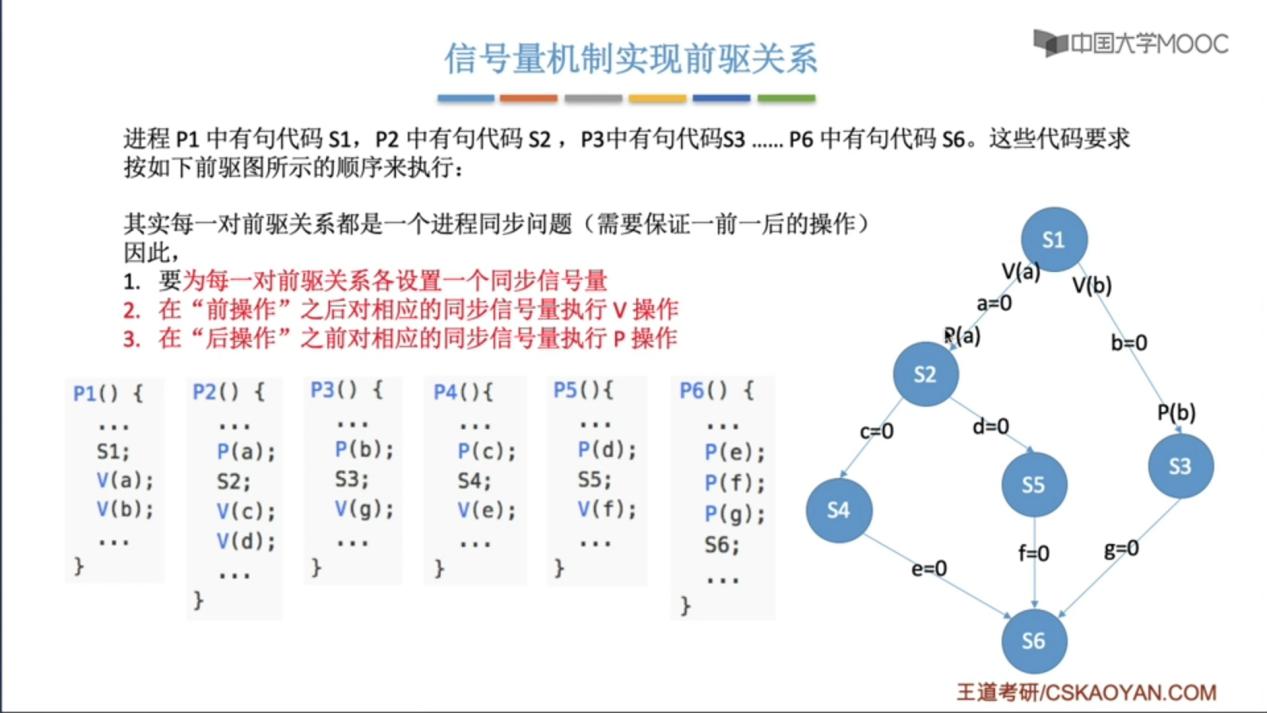
3.信号量机制实现前驱关系

利用上面的前V后P原则可实现多个进程之间的同步问题。解决的方法如下：

（1）为**每一对**前驱关系各设置一个同步信号量，**初值为0**

（2）在“前面的操作”后面对相应的同步信号量执行V操作，在“后面的操作”前面对相应的同步信号量执行P操作

如下例：



本节小结



**2.3.6生产者-消费者问题**

问题：系统中有一组生产者进程和一组消费者进程，生产者进程每次生产一个产品放入缓冲区，消费者进程每次从缓冲区中取出一个产品并使用。

生产者、消费者共享一个初始为空，大小为n的缓冲区。只有缓冲区没满时，生产者才能把产品放入缓冲区，否则必须等待。只有缓冲区不空时，消费者才能从中取出产品，否则必须等待。缓冲区是临界资源，各进程必须互斥访问缓冲区。（否则两个生产者生产的产品，如写入的数据，可能互相覆盖）

首先要**分析关系**：找出题目中描述的各个进程，分析它们之间的同步、互斥关系

本问题中有两对同步关系，即

（1）缓冲区有产品，消费者才能消费

（2）缓冲区没满（有空位），生产者才能生产

为此，需要为这两对**同步关系各定义一个信号量**:(1)的信号量为full,(2)的信号量为empty

在缓冲区有产品后加V(full)操作，在消费者从缓冲区取产品之前加P(full)操作。在缓冲区没满后加V(empty)操作，在生产者将产品放入缓冲区之前加P(empty)操作。

然后**确定信号量初值**：由于消费者申请的资源是“产品”，而最开始没有产品，所以full的初值为0。由于生产者申请的资源是“缓冲区的空位”，而最开始有n个空位，所以empty的初值为n。由于要实现对缓冲区的互斥访问，所以要设置信号量mutex=1来实现对缓冲区的互斥访问

semaphore mutex=1,empty=n,full=0;

producer(){

while(1){

生产一个产品；

P(empty)；//消耗一个空闲区

把产品放入缓冲区；

V(full);//增加一个产品

}

}

consumer(){

while(1){

P(full);//消耗一个产品（非空区）

从缓冲区取出一个产品；

V(empty);//产生一个空位

使用产品;

}

}

如果再加上对缓冲区的互斥访问的PV操作，则只需在每次访问缓冲区前加P(mutex),访问缓冲区后加V(mutex)。

producer(){

while(1){

生产一个产品；

P(empty);

P(mutex);

把产品放入缓冲区；

V(mutex);

V(full);

}

}

consumer(){

while(1){

P(full);

P(mutex);

从缓冲区取出一个产品；

V(mutex);

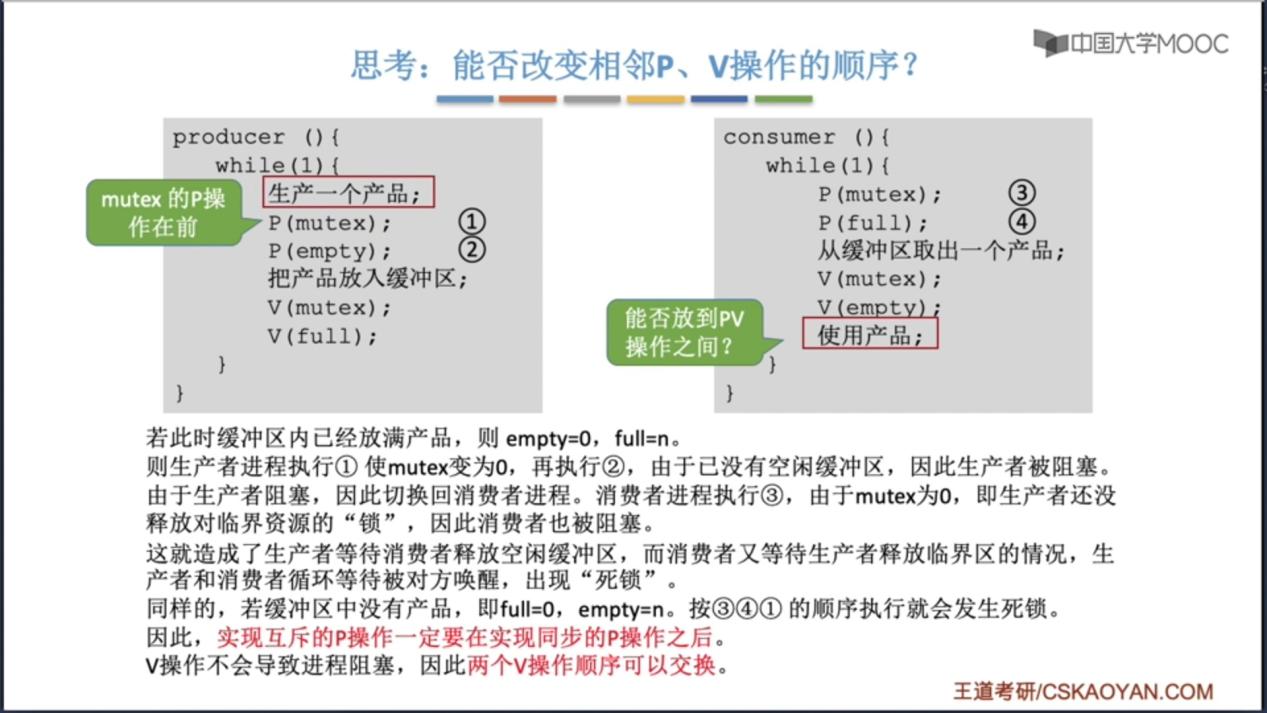
V(empty);

使用产品；

}

}

那么能不能将每个进程里对mutex的P操作和对empty或full的P操作交换顺序呢？



如果交换了顺序，在缓冲区放满产品时，empty=0,full=n。若进程间调度导致按①②③的顺序执行，则①会让producer拿到缓冲区的使用权，②会将producer进程阻塞在信号empty的等待队列里。而③会让consumer阻塞在缓冲区的等待队列里。这就造成了生产者等待消费者释放空闲区，而消费者又等待生产者释放临界区。双方都等待被对方“唤醒”，结果双双卡死。这个现象叫“死锁”。同样，当缓冲区空时,empty=n,full=0。若按③④①的顺序执行，也会导致死锁。因此实现互斥的P操作一定要在实现同步的P操作之后。

而V操作不会导致进程阻塞，因此两个V操作的顺序可以交换。

**2.3.7多生产者多消费者问题**

问题：桌子上有一只盘子每次只能向其中放入一个水果。爸爸向盘子中放苹果，妈妈向盘子中放橘子。女儿只吃盘子中的苹果，儿子只吃盘子中的橘子。只有盘子空时爸爸或妈妈才可以放一个水果，仅当盘子中有自己想吃的水果时，儿子或女儿才可以从盘子中取出水果。用PV操作实现上述过程。

**关系分析**

盘子看作是大小为1的缓冲区。爸爸妈妈看作生产者进程，儿子女儿看作消费者进程。但这个问题中生产者生产的产品类别不同，和消费者消费的产品类别不同。

互斥关系：要实现对盘子（缓冲区）的互斥访问

同步关系：（1）父亲将苹果放入盘子后，女儿才能取苹果

（2）母亲将橘子放入盘子后，儿子才能取橘子

（3）只有盘子为空时，父亲或母亲才能放入水果。盘子为空这个事件可以由儿子或女儿触发。事件发生后才允许父亲或母亲放水果

**整理思路**

实现互斥要在临界区前后分别执行PV操作。实现同步要在前事件后执行V操作，后事件前执行P操作。对于互斥关系，要设置一个初值为1的信号量mutex。对于每一对同步关系，应各设置一个信号量，然后根据问题给的初始情况确定信号量的初值。

关系（1）设置信号量apple，开始时盘子没有苹果，故apple初值为0

关系（2）设置信号量orange,开始时盘子没有橘子，故orange初值为0

关系（3）设置信号量plate,开始时盘子中是允许放水果的，故plate初值为1

由上面的分析，可以得到各进程的操作如下：

semaphore mutex=1,apple=0,orange=0,plate=1;

dad(){

while(1){

P(plate);

P(mutex);

在盘子中放入苹果；

V(mutex);

V(apple);

}

}

mom(){

while(1){

P(plate);

P(mutex);

在盘子中放入橘子；

V(mutex);

V(orange);

}

}

daughter(){

while(1){

P(apple);

P(mutex);

取出盘子中的苹果；

V(mutex);

V(plate);

}

}

son(){

while(1){

P(orange);

P(mutex);

取出盘子中的橘子；

V(mutex);

V(plate);

}

}

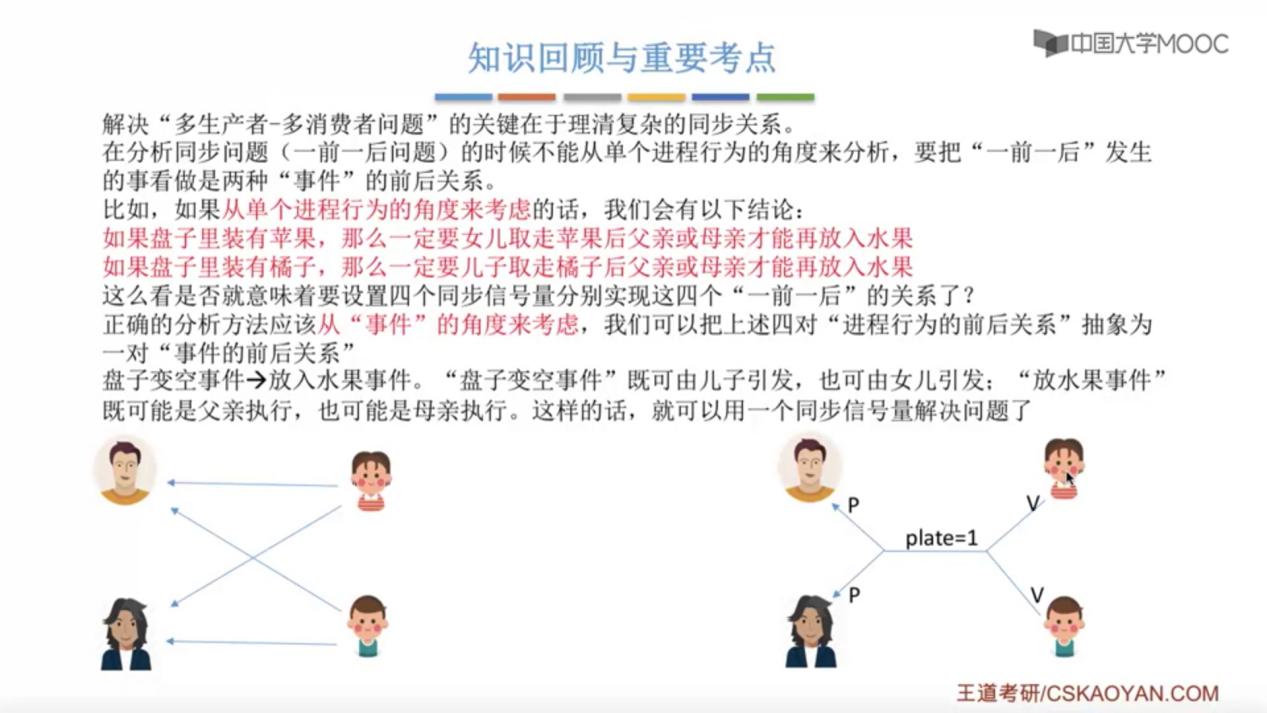
那么是否可以去掉对盘子进行互斥访问的信号量mutex和对mutex进行的所有PV操作呢？如果去掉所有的mutex的操作。刚开始时，即使儿子女儿先上处理机运行也会因为P(水果)而阻塞，只有父亲或母亲中的一个可以通过P(plate)操作。如果是父亲先通过了P(plate)操作，因为plate的值为1，所以母亲不能再通过P(plate)而阻塞，此时只有父亲进程能继续运行下去，直到父亲的V(apple)操作唤醒女儿进程。此时女儿从盘中取出苹果，执行V(plate)操作，唤醒母亲进程。然后母亲通过P(plate),在盘中放入橘子，执行V(orange)操作，又唤醒儿子进程。儿子最后从盘中取走橘子。因此，即使没有对盘子互斥访问的信号量mutex，只依靠apple和orange两个信号量依然能实现对盘子的互斥访问。mutex信号量是可以去掉的。

原因在于本题中的缓冲区大小为1，在任何时刻，apple,orange,plate三个信号量中最多只有一个是1。因此在任何时刻，最多只有一个进程的P操作不会被阻塞并顺利进入临界区。其它进程的P操作均会被阻塞。

如果盘子的容量是2的话，即plate的初值为2。则刚开始父亲进程可以通过P(plate),父亲通过后母亲也可以通过P(plate)。即父亲母亲可以同时进入临界区，这样两个生产者写入缓冲区的数据有可能相互覆盖。因为一个生产者写入缓冲后，不待缓冲区的指针后移就让另一生产者写入，写的数据可能相互覆盖。所以将产品放入缓冲区指针所指位置，和缓冲区指针后移的两个操作必须一气呵成，这就要求必须定义初值为1的信号量mutex来保证两个生产者对缓冲区进行互斥访问。

**总结：在生产者-消费者问题中，如果缓冲区大小为1，那么有可能不需要设置互斥信号量就可以实现互斥访问缓冲区的功能。当然，这不是绝对的，要具体问题具体分析。**考试中如果来不及分析，可以加上互斥信号量保证各进程一定会互斥访问缓冲区。但要注意实现互斥的P操作一定要在实现同步的P操作之后，否则可能引起“死锁”。

另外，在分析同步关系的时候，一定要从“事件”的角度进行分析，而不能从“进程的行为”的角度进行分析。前后指的是两个事件的前后，而不是进程行为的前后。此问题中如果从进程的行为角度来分析，那么就会得出“如果盘中有苹果，女儿必须取走苹果父亲才能再放入苹果”、“如果盘中有苹果，女儿必须取走苹果母亲才能再放入橘子”，“如果盘中有橘子，儿子必须取走橘子父亲才能再放入苹果”，“如果盘中有橘子，儿子必须取走橘子母亲才能再放入橘子”这四种“行为的前后关系”，这并不意味着需要有4个信号量来实现这个同步。正确的想法应该是从事件的角度分析，即“盘子变空”的事件可以导致“生产者放入水果”这一事件。只需一个信号plate就可以实现这一同步条件。



**2.3.8吸烟者问题**

问题：假设一个系统有三个抽烟者进程和一个供应者进程。每个抽烟者不停地卷烟并抽掉它，但要卷起并抽掉一支烟，抽烟者需要有三种材料：烟草、纸和胶水。三个抽烟者中，第一个拥有烟草，第二个拥有纸，第三个拥有胶水。供应者进程无限地提供三种材料，供应者每次将两种材料放在桌子上，拥有剩下那种材料的抽烟者可以卷一根烟并抽掉它，并给供应者进程发一个信号告诉供应者他抽完了。供应者就会放另外两种材料在桌上，这个过程一直重复（让三个抽烟者轮流抽烟）

这个问题是生产者和消费者问题的变种。不同点在于是“可生产多种产品的单生产者”对“多个消费者”

**关系分析**

桌子看作是容量为1初值为空的缓冲区。同一时刻桌子上会放三种产品组合中的一种：

组合1：纸+胶水， 组合2：烟草+胶水， 组合3：烟草+纸。

同步关系（从事件的角度分析）：

（1）当桌子上有组合1这件事完成后才能使抽烟者1取走纸和胶水并卷烟抽掉

（2）当桌子上有组合2这件事完成后才能使抽烟者2取走烟草和胶水并卷烟抽掉

（3）当桌子上有组合3这件事完成后才能使抽烟者3取走烟草和纸并卷烟抽掉

（4）某抽烟者发出抽烟完成信号这件事完成后供应者才能将下一个组合放在桌子上

**整理思路**

对于互斥关系，由于缓冲区的大小为1，所以可以不设置对桌子互斥访问的信号量

对于上面四个同步关系，需各设置一个信号量：（1）的信号量为offer1，（2）的信号量为offer2,(3)的信号量为offer3，（4）的信号量为finish。

由于刚才开始时桌子上没有东西，所以offer1,offer2,offer3的初值应设为0。最开始时没有抽烟者发出抽烟完成的信号，所以finish的初值也设为0。为了实现三个抽烟者轮流抽烟，可以设置一个整型变量i,每次循环结束使i加1再模3。根据i的值决定每次循环要为哪个抽烟者提供材料。由此，得到的各进程操作如下：

semaphore offer1=0,offer2=0,offer3=0,finish=0;

int i = 0;

producer(){

while(1){

if(i==0){

将组合1放在桌上；

V(offer1);

}else if(i==1){

将组合2放在桌上；

V(offer2);

}else if(i==2){

将组合3放在桌子上；

V(offer3);

}

i = (i+1)%3;

P(finish);

}

}

smoker1(){

while(1){

P(offer1);

从桌上拿走组合1卷烟抽掉；

V(finish);

}

}

smoker2(){

while(1){

P(offer2);

从桌上拿走组合2卷烟抽掉；

V(finish);

}

}

smoker3(){

while(1){

P(offer3);

从桌上拿走组合3卷烟抽掉；

V(finish);

}

}

对上面的代码经分析后，不需设一个专门的信号量实现对桌子的互斥访问。因为任一时刻，四个信号量里最多只有一个是1。

若一个生产者要生产多种产品（或者说会引发多种前驱事件），那么各个V操作应该放在各自对应的“事件”发生之后的位置。

**2.3.9读者-写者问题**

问题：有读者和写者两组并发进程，共享一个文件，当两个或两个以上的读进程同时访问共享数据时不会产生副作用，但若某个写进程和其它进程（读进程或另一写进程）同时访问共享数据时则可能导致数据不一致的错误。因此要求：（1）允许多个读者可以同时对文件执行读操作（2）只允许一个写者往文件中写信息（3）任一写者在完成写操作前不允许其它读者或写者工作（4）写者执行写操作前，应让已有的读者和写者全部退出

**关系分析**：

互斥关系：读进程与读进程不存在互斥问题，写进程-读进程，写进程-写进程存在互斥问题。

本题中没有同步关系。

**整理思路：**

为文件互斥访问设置信号量rw，初始时允许进程访问，故rw初值为1。由于写者与读者要互斥访问文件，所以写文件和读文件的前后都需要P(rw)和V(rw)保证读者和写者，写者和写者互斥。

writer(){ reader(){

while(1){ while(1){

P(rw); P(rw);

写文件； 读文件；

V(rw); V(rw);

} }

} }

但这样又会导致读者与读者之间不能同时访问文件。于是我们用一个整型变量count来记录某时刻正在访问文件的读进程的数目。每次对文件的加锁都由第一个读文件的进程执行，每次对文件的解锁都由最后一个读完文件的进程执行。count的初值应设为0。读进程变成如下：

reader(){

while(1){

if(count==0) P(rw); //如果已有读进程为文件加锁，则后来的读进程不必再加锁

count++; //读进程数目加1

读文件；

count--;

if(count==0) V(rw);

}

}

但这样又会导致问题，若两个读进程并发执行，则初始时刻读者1来访问时检查了count=0，正要执行P(rw)操作时突然发生进程切换，此切换到读者2，则读者2来访问时count依然是0，于是执行P(rw)操作，使rw的值变为0。若此时再切回读者1，则其执行P(rw)操作时会发现rw的值已经是0，于是读者1就被阻塞。这样就不能实现多个读者对文件的同时访问。这个问题的原因在于，检查count是否为0和count加一的操作没有一气呵成。

因此，需要设置另一个互斥信号量mutex来保证各读进程对count变量的访问是互斥的。由于最开始是可以对count进行访问的，所以mutex的初值为1。在执行if(count==0)前要执行P(mutex),在执行完count++后需要立即执行V(mutex)然后再读文件：

semaphore rw=1,mutex=1;

int count = 0;

reader(){

while(1){

P(mutex);

if(count ==0) P(rw);

count++;

V(mutex);

读文件；

P(mutex);

count--;

if(count==0) V(rw);

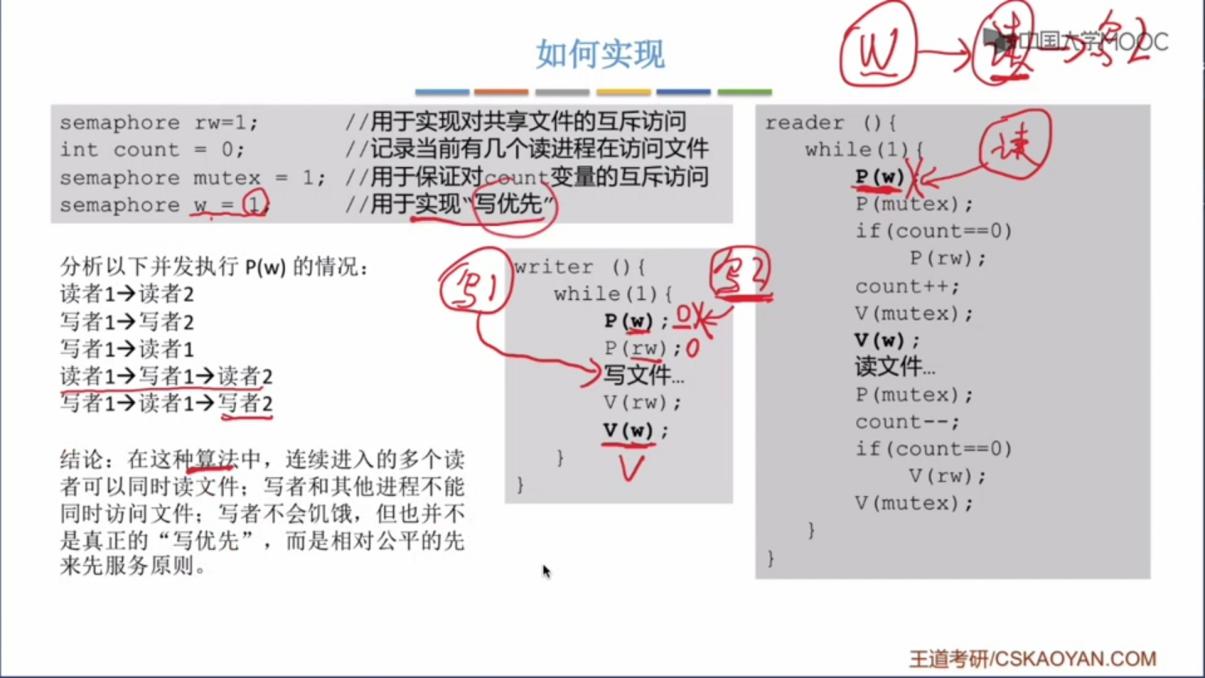
V(mutex);

}

}

到这里，问题已经解决了。但依然会有潜在的问题：只要有读进程一直来读，写进程就要一直阻塞等待，可能会“饿死”。因此这种算法中，读进程是优先的。

为了保证读进程和写进程之间的公平，我们还可以设置一个信号量w来为的各个写进程和读进程排队。



**2.3.10哲学家进餐问题**

问题：一张圆桌上坐着5名哲学家，每两个哲学家之间的桌上摆一根筷子。哲学家每时刻只做两件事中的一件：思考或进餐。哲学家在思考时并不影响他人，只有当哲学家饥饿时才试图拿走左右两根筷子（一根一根地拿起）。如果筷子已经在别人手上，则需要等待。饥饿的哲学家只有同时拿起两根筷子才可以开始进餐，当进餐完毕后，放下筷子继续思考。

**关系分析：**

系统中有5个哲学家进程，5位哲学家与左右邻居对其中间的筷子的访问是互斥关系。

**整理思路：**

这个问题中只有互斥关系，但与之前问题不同的是，每个哲学家进程需要同时持有两个临界资源才能开始吃饭。如何避免临界资源分配不当造成的死锁现象，是哲学家问题的精髓。

为了实现对五个筷子资源的互斥访问，我们定义一个互斥信号量数组chopstick[5]={1,1,1,1,1}并对哲学家按0-4编号，哲学家i左边的筷子编号为i，右边的筷子编号为(i+1)%5。

如果按直接的想法，每个哲学家吃饭前用两次P操作依次拿起自己左边的筷子和右边的筷子，然后吃饭，再依次用再次V操作放下自己左边的筷子和右边的筷子，就像如下算法：

semaphore chopstick[5]={1,1,1,1,1};

P(int i){//i是哲学家的编号

while(1){

P(chopstick[i]); ①//拿起左边的筷子

P(chopstick[(i+1)%5]); ② //拿起右边的筷子

吃饭；

V(chopstick[i]); //放下左边的筷子

V(chopstick[(i+1)%5]); //放下右边的筷子

思考；

}

}

那么如果五个哲学家并发执行，上边的算法会造成死锁：哲学家0首先拿起筷子0，然后进程切换到哲学家1进程，哲学家1又拿起筷子1，然后进程切换到哲学家2拿起筷子2，依此类推，哲学家3拿起筷子3，哲学家4拿起筷子4。这样，当进程再次切换到任何一个哲学家时，均已无法拿起自己右边的筷子：哲学家0右边的筷子1在哲学家1手中，哲学家1右边的筷子2在哲学家2手中。这样每个人手里都有一支筷子且都等待右边的人放下他手中的筷子，从而造成死锁。

解决上述问题的思路有三种：

1.外加一个同时最多只能有4位哲学家同时进餐的限制。这样，可以保证至少有一个哲学家是可以拿到左右两只筷子的。为此设置一个初值为4的信号量k:

semaphore k = 4;

P (int i){

while(1){

P(k);

P(chopstick[i]);

P(chopstick[(i+1)%5]);

吃饭；

V(chopstick[i]);

V(chopstick[(i+1)%5]);

V(k);

思考；

}

}

2.死锁的原因在于每位哲学家都是先拿左边的筷子再拿右边的筷子，如果规定奇数号哲学家先拿左边的筷子，偶数号哲学家先拿右边的筷子。用这种方法可以保证如果相邻的两个奇偶号哲学家都想吃饭，他们必须先竞争他们中间的筷子再去拿别的筷子。这样只会有其中一个可以拿起他们中间的筷子，另一个只能“空手”阻塞。这就避免了占有一支后等待另一支的情况。

P (int i){

while(1){

if (i%2==1){

P(chopstick[i]);

P(chopstick[(i+1)%5]);

}else{

P(chopstick[(i+1)%5]);

P(chopstick[i]);

}

吃饭；

V(chopstick[i]);

V(chopstick[(i+1)%5]);

思考；

}

}

3.仅当一个哲学家左右两支筷子都可以使用时才能让他拿起筷子,为此设置一个让拿两支筷子一气呵成的互斥信号量mutex：

semaphore mutex=1;

P(int i){

while(1){

P(mutex);

P(chopstick[i]);

P(chopstick[(i+1)%5]);

V(mutex);

吃饭；

V(chopstick[i]);

V(chopstickp[(i+1)%5]);

思考；

}

}

最开始当一个哲学家拿起了自己左边的筷子但还未拿起右边的筷子时，mutex的值为0。此时其它哲学家试图拿筷子时只能阻塞。等到此哲学家拿起了两支筷子并执行V(mutex)后，其它的哲学家才能来拿筷子。

实际上，这种算法会有可能出现一名哲学家在两边的筷子不能都拿起的情况下拿起暂时能用的那一支，但此后该哲学家进程会阻塞而无法释放mutex。这就导致其它再试图拿筷子的哲学家阻塞在P(mutex)这一步。因此保证了各哲学家拿筷子这件事是互斥执行的。即使有一个哲学家在拿筷子拿到一半时被阻塞，也不会有别的哲学家继续尝试拿筷子，因而可以解决问题。

**2.3.11管程**

信号量机制存在编写程序困难，易出错的问题。为了让程序员写程序不再关注复杂的PV操作，1973年Hansen首次在Pascal语言中引入了“管程”这种高级同步机制。

1.管程的定义

管程是一种特殊的软件模块，由以下部分组成：

1.局部于管程的共享数据结构说明；

2.对该数据结构进行操作的一组过程；

3.对局部于管程的共享数据设置初始值的语句

4.管程的名字

管理的定义类似于“类”。

2.管程的基本特征

1.局部于管程的数据只能被局部于管程的过程所访问（类似于类的私有成员变量，只能由类的成员函数访问）

2.一个进程只有通过调用管程内的过程才能进入管程访问共享数据（类似于类的外部程序只用调用类的公有成员函数才能访问类的私有成员变量）

**3.每次仅允许一个进程在管程内执行某个内部过程**

由于只允许一个进程在管程内执行内部过程，所以进程在释放管程前，其它进程都不能访问管程内的共享数据结构，从而实现了对它的互斥访问。

例：用管程解决生产者消费者问题（伪代码）

monitor ProducerConsumer

condition full,empty;//条件变量，有wait操作和signal操作。类似于信号量

int count = 0;//缓冲区中产品数

void insert(Item item){

if(count==N) full.wait();//条件变量full有缓冲区满时将调用管程的进程挂在自己的队列上并让进程释放管程；

count++;

insert\_item(item);//在缓冲区插入产品

if(count==1) empty.signal();//条件变量empty将一个在自己队列上等待的进程唤醒

}

Item remove(){

if(count==0) empty.wait();

count--;

if(count==N-1) full.signal();//消费了一个产品可以唤醒一个等待的生产者

return remove\_item();

}

end monitor;

生产者进程就可以调用管程中的insert过程将产品放入缓冲区

producer(){

while(1){

item=生产一个产品；

ProducerConsumer.insert(item);

}

}

consumer(){

while(1){

item=ProducerConsumer.remove();

消费产品item;

}

}

每次仅允许一个进程在管程内执行某个过程，由编译器负责实现各进程互斥进入管程中的过程。管程中设置条件变量和等待/唤醒操作，以解决同步问题。

引入管理的目的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”--其实就是一些过程函数

3.只有通过这些特定的“入口”才能访问共享数据

4.管理中有很多“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，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，该进程应先释放管理的使用权，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

**2.4.1死锁的概念**

1.死锁的概念

死锁：各进程互相等待对方手里的资源，导致各进程都阻塞，无法向前推进的现象。

死锁有别于饥饿，饥饿是指由于长期得不到想要的资源，某进程无法向前推进的现象。比如SJF算法中若有短进程源源不断到来，则长进程将一直得不到处理机。死循环是指执行过程中一直跳不出某个循环的现象。

区别：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。

饥饿现象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，也可能是就绪态

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。死锁和饥饿是操作系统的问题，死循环是被管理者的问题。

2.死锁的必要条件：

产生死锁必须满足以下四个条件，只要其中任一条件不成立，死锁就不会发生。

（1）互斥条件：只有对**必须互斥使用的资源的争抢**才会导致死锁

像内存、扬声器这样可以同时让多个进程使用的资源是不会导致死锁的。进程不用阻塞等待。

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

（3）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。

（4）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。

发生死锁时一定有循环等待，但发生循环等待时未必死锁。如果同类资源数大于1，则即使有循环等待，也未必发生死锁。但如果系统中每类资源都只有一个，那么循环等待就是死锁的充分必要条件。

3.什么时候会发生死锁

（1）.对系统资源的竞争。各进程对不可剥夺的资源的竞争可能引起死锁，对可剥夺的资源的竞争是不会引起死锁的（如CPU）

（2）.进程推进顺序非法。请求和释放资源的顺序不当，也同样会导致死锁。例如并发执行的进程P1占有资源R1，进程P2占有资源R2。之后进程P1又紧接着申请资源R2，而进程P2又申请资源R1。两者会因为申请的资源被对方占有而阻塞，从而发生死锁。

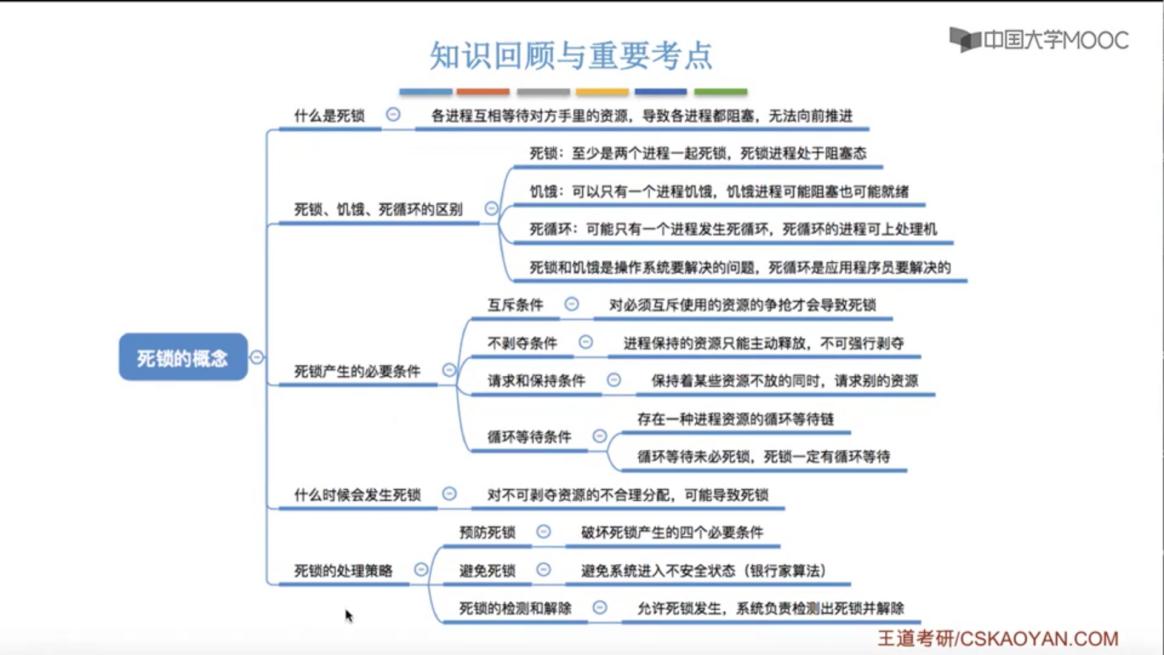
（3）.信号量的使用不当也会造成死锁。如生产者-消费者问题中，如果实现互斥的P操作在实现同步的P之前，就有可能导致死锁。

4.死锁的处理策略

（1）预防死锁。破坏死锁产生的四个必要条件中的一个或几个

（2）避免死锁。用某种算法防止系统进入不安全状态，从而避免死锁

（3）死锁的检测和解除。允许死锁发生但操作系统会负责检测出死锁的发生，然后采取某种措施解除死锁



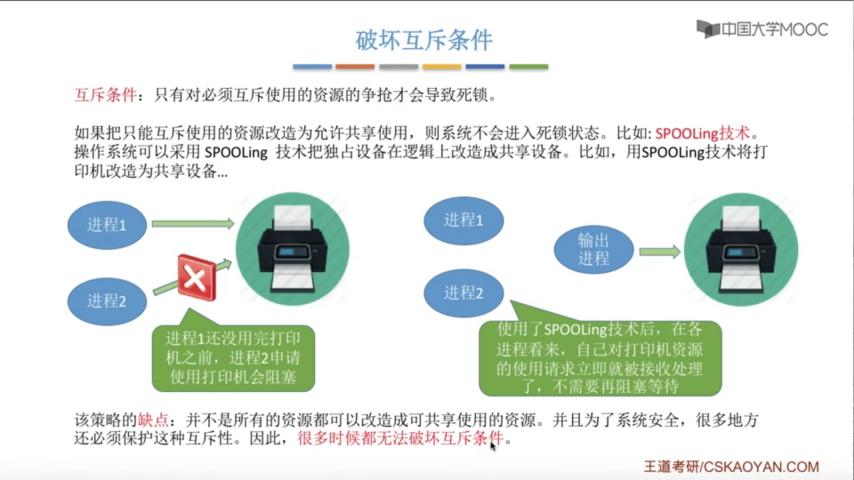
**2.4.2预防死锁**

预防死锁就是指通过破坏死锁产生的必要条件来防止死锁

1.破坏互斥条件

如果把只能互斥使用的资源改造为允许共享使用，则系统不会进入死锁状态。比如：SPOOLing技术。操作系统可以采用SPOOLing技术把独占设备在逻辑上改造成共享设备。比如用SPOOLing技术将打印机改造为共享设备。使用了SPOOLing技术后，在各进程看来，自己对打印机资源的使用请求立即被一个“输出进程”接收处理了，就不需要自己再阻塞等待了。

但并不是所有的资源都可以改造成可共享使用的资源。并且为了系统安全，很多地方还必须保护这种互斥性。因此，很多时候都无法破坏互斥条件。



2.破坏不剥夺条件

方案一：当进程请求新资源得不到满足时，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。

方案二：当进程需要的资源被其它进程所占有的时候，可以由操作系统协助，将想要的资源强行夺来。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。

缺点：

（1）实现比较复杂

（2）释放已获得的资源可能造成前一阶段工作的失效，因此只适用于易保存和恢复状态的资源

（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就要把之前获得的那些资源全部放弃，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。

3.破坏请求和保持条件

可以采用静态分配方法，即进程在运行前，一次申请完所需要的全部资源，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，该进程不会再请求别的任何资源了。

该策略实现起来简单，但缺点在于：

（1）有些资源可能很少时间在使用，因此进程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。

（2）另外，该策略也有可能导致某些进程饥饿。（需要资源多的进程很难一时凑全所有所需资源，因而长期间得不到执行）

4.破坏循环等待条件

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。在任何一时刻，总有一个进程拥有的资源编号是最大的，那这个进程申请之后的资源必然畅通无阻。因此不可能出现所有进程都阻塞的死锁现象。

该策略的缺点：

（1）不方便增加新设备，因为可能需要重新分配所有编号

（2）进程实际使用资源的顺序可能和编号递增的顺序不一致，导致其长时间持有小号资源却一直不用，浪费了资源

（3）必须按规定次序申请资源，用户编程麻烦，限制了用户简单自主地编程

**2.4.3死锁的处理策略--避免死锁**

1.安全序列

如果系统按照一种序列分配资源，每个进程都有顺利完成。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，不到我们在分配资源之前总是要考虑到最坏的情况。

如果系统处于安全状态，就一定不会发生死锁。如果系统进入不安全状态，就可能发生死锁（处于不安全状态未必就是发生了死锁，但发生死锁时一定是在不安全状态）

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

2.银行家算法

判断系统是否处于安全状态可以用安全性算法。思路是尝试找出一个安全序列。

依次检查剩余可用资源是否能满足各进程要求，假设能满足某进程要求，就将它加入安全序列，然后把资源分配给它让它运行到底最后回收它已分配的资源。将回收后的资源加上原来剩余的资源就是新的剩余资源。然后再找到一个新的能满足要求的进程，重复上述过程，直到所有进程都能加入安全序列，则系统是处于安全状态的。否则在某一时刻剩余资源不能满足任何一个进程的要求了，则系统是不安全的，有可能发生死锁。

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，可用一个n\*m的矩阵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Requesti表示本次申请的各种资源量。

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Requesti[j]<=Need[i,j]便转向（2），否则认为出错

（2）如果Requesti[j]<=Available[j]便转向（3），否则表示尚无足够资源，Pi必须等待

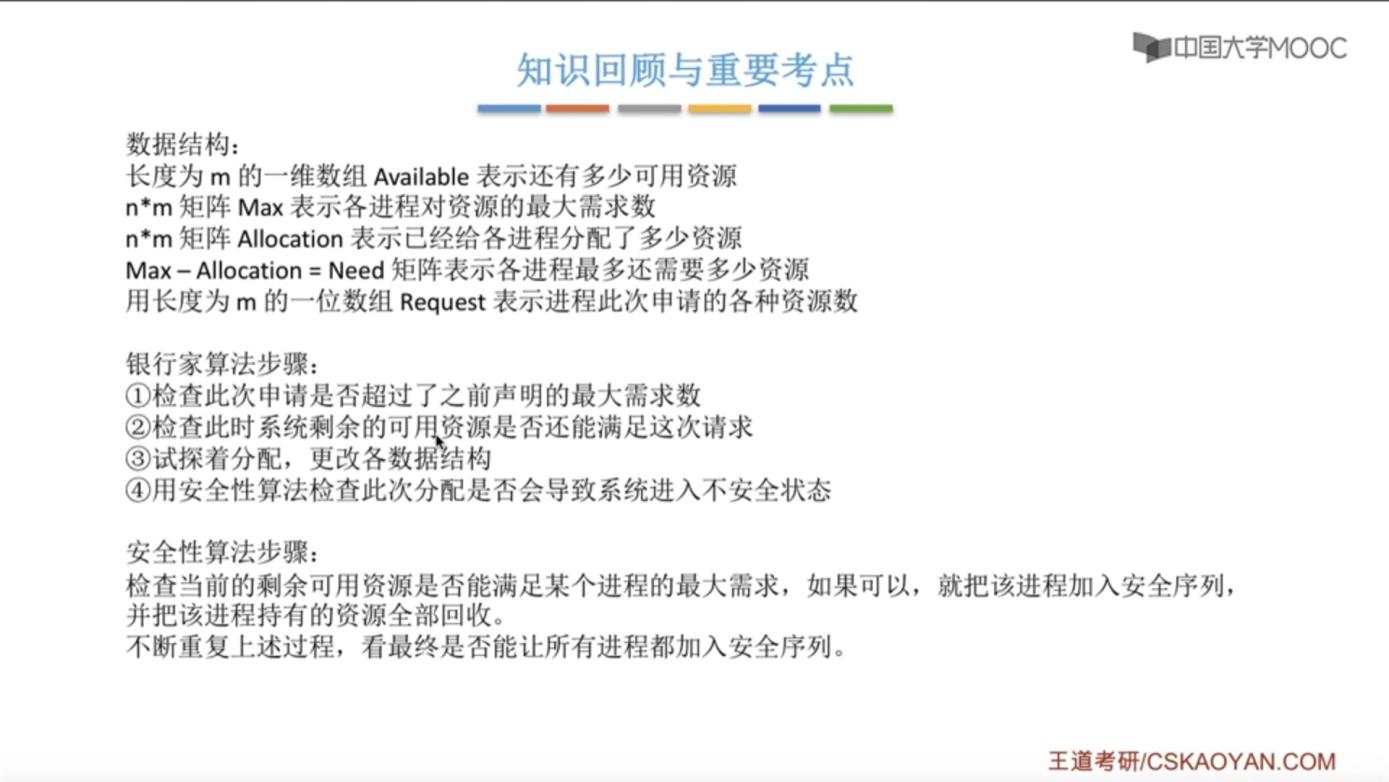
（3）系统试探着把资源分配给进程Pi，并修改相应数据（并非真的分配，修改数据只是为了做预判）

Available = Available - Requesti

Allocation[i,j]=Allocation[i,j]+Requesti[j];//假设答应分配请求，则该进程已分配资源数要加上本次分配数

Need[i,j]=Need[i,j]-Requesti[j];//进程还需要的资源数减去本次分配数

（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，恢复相应的数据，让进程阻塞等待。



**2.4.4死锁的检测与解除**

如果系统中既不采取预防死锁的措施，也不采取避免死锁的措施，系统就很可能发生死锁。在这种情况下，系统就提供两个算法：死锁检测算法、死锁解除算法

1.死锁的检测

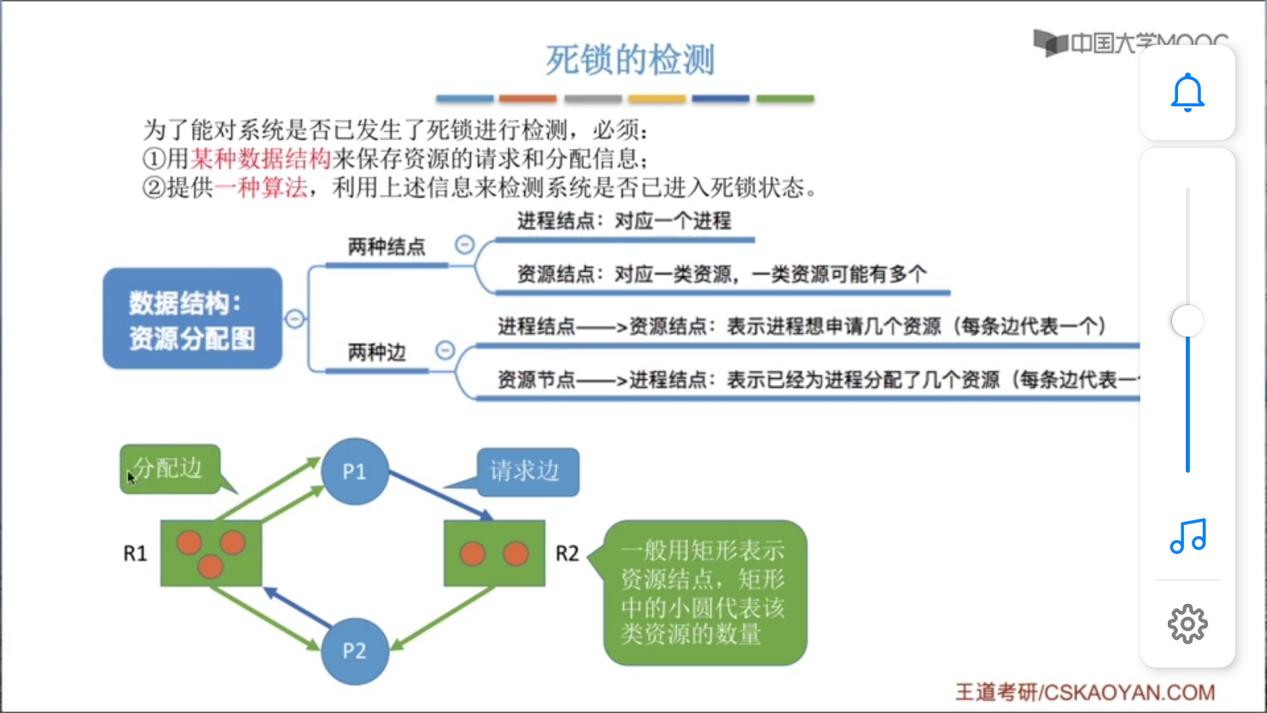
我们用资源分配图来保存资源的请求和分配信息。资源分配图中有两种结点：进程结点、资源结点。有两种边：请求边、分配边。每条边连接一个进程结点和一个资源结点。各结点各边的含义如下：

进程结点：表示一个进程

资源结点：对应一类资源，一个资源可能有多个

请求边：表示进程还想申请多少个资源（每条边代表想申请一个资源）

分配边：表示该资源已有多个分配给了某进程（每条边代表已分配一个资源）



如上的资源分配图表示系统中有P1和P2两个进程，R1资源有3个，R2资源有2个。P1有一条请求边指向R2，说明P1想请求1个R2资源。R1有两条分配边指向P1，说明R1资源已有2个分配给P1。

如果系统中剩余的可用资源足够满足进程的需求，那么这个进程暂时是不会阻塞的，可以顺利地执行下去。检测死锁的算法如下：

（1）在资源分配图中，找出既不阻塞又不是孤点的进程（即找出一条有边与它相连，且该有向边对应资源的申请数量小于系统中已有空闲资源数量）消去它所有的请求边和分配边，使它成为孤立的结点。在上图中，P1满足这一条件，于是消去P1所有边。

（2）进程所释放的资源可以唤醒某些因等待这些资源而阻塞的进程，原来的阻塞进程可能变成非阻塞进程。上图中P2就满足这样的条件：P1释放2个R1资源后可满足P2对1个R1资源的请求。根据（1）中方法进行一系列简化后，若能消去图中所有的边，则称该图是**可完全简化**的。否则称为资源分配图是不可完全简化的。

死锁定理：如果某时刻系统的资源分配图是不可完全简化的，那么此时系统死锁。

系统死锁也不意味着所有进程都是死锁的，用死锁检测算法化简资源分配图后，还连着边的那些进程才是死锁进程。

2.死锁的解除

主要的方法有：

（1）资源剥夺法。挂起某些死锁进程，并抢占它的资源，将这些资源分配给其它的死锁进程。但是应防止被挂起的进程长时间得不到资源而饥饿。

（2）撤销进程法。强制撤销部分甚至全部死锁进程，并剥夺这些进程的资源。这种方式的优点是实现简单，但所付出的代价可能会很大。因为有些进程可能已经运行了很长时间了，已经接近结束了，一旦被终止可谓功亏一篑，以后还得从头再来。

（3）进程回退法。让一个或多个死锁进程回退到足以避免死锁的地步。这就要求系统要记录进程的历史信息，设置还原点。这种方法实现很复杂。

决定剥夺哪个进程的资源或撤销哪个进程，可以从以下角度考虑：

1.进程优先级 2.已经执行的时间（越长的话撤销代价越大，尽量不剥夺）3.还要多久能完成（剩余时间少的尽量不剥夺） 4.进程已经使用了多少资源（撤销使用了很多的资源的进程可释放出很多资源，优先剥夺）5.进程是交互式的还是批处理式的（交互式的尽量不剥夺）

**3.1.1内存的基础知识**

**1.内存**

内存可存放数据。程序执行前需要先放到内存中才能被CPU处理，这是为了缓和CPU和硬盘之间的速度矛盾。

内存通过给存储单元编址的方式区分各个程序的数据的存放位置。内存地址从0开始，每个地址对应一个存储单元。如果计算机“按字节编址”，则每个存储单元大小为1字节，即1B。即8个二进制位。如果字长为16位的计算机“按字编址”，则每个存储单元大小为1个字，每个字的大小为16个二进制位。

有的系统中，寄存器和内存可能统一编址。指令的工作基于“地址”，每个地址对应一个数据的存储单元。我们写的代码要翻译成CPU能识别的指令。这些指令会告诉CPU应该去内存的哪个地址读/写数据，这个数据应该做什么样的处理。

C语言程序经过编译、链接处理后，生成一个可执行文件（也叫装入模块）。程序经过编译、链接后生成的指令中指明的是逻辑地址，即：相对于进程的起始地址而言的地址

比如一条C语言的语句int x=10编译后的结果是“往地址为79的存储单元中写入10”，这个79是一个相对地址。假设程序装入内存时，是从地址为100的存储单元开始的。那么变量x的物理地址就应该是100+79=179，执行这条语句就是在物理地址为179的存储单元里写入10。

程序经编译链接后指明的是逻辑地址，而程序在实际执行时被装入内存的什么位置是不确定的。所以内存管理的一个很重要的问题是如何将指令中的逻辑地址转换为物理地址？

在装入程序时，有三种不同的装入方式的策略解决这个问题：

1.绝对装入

在编译时，如果知道程序将放到内存中的哪个位置，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，将程序和数据装入内存。

例，如果知道装入要从地址100的地方开始存放，那么编译、链接后得到的指令直接就可以使用绝对地址。在上面的例子中，指令被编译的结果就是“往绝对地址为179的存储单元中写入10”。这种方式的前提是需要提前知道装入模块被装入到内存的什么位置，所以灵活性很差，只适用于单道程序环境。程序拿到另一台计算机中，装入的位置又不一定是100了，这个绝对地址就没有意义了。

2.静态重定位：又称可重定位装入

编译链接后的装入模块的地址都是从0开始的，指令中使用的地址、数据存放的地址都是相对于起始地址而言的逻辑地址。可根据内存的当前情况，将装入模块装入到内存的适当位置。装入时对地址进行“重定位”，将逻辑地址变换为物理地址，地址变换是在**装入时**一次完成的。

在上面的例子中，装入模块里指令的内容依旧是“地址为79的存储单元中写入10”。若在装入时，起始物理地址为100，则所有地址相关的参数都加100。内存里的指令内容就变成“绝对地址为179的存储单元中写入10”。

静态重定位的特点是

（1）在一个作业装入内存时，必须分配其要求的全部内存空间，如果没有足够的内存，就不能装入该作业。

（2）作业一旦进入内存后，在运行期间就不能再移动，也不能再申请内存空间。

3.动态重定位：又称动态运行时装入

编译链接后的装入模块的地址也是从0开始的，装入程序将装入模块装入内存后，并不会立即把逻辑地址转换为物理地址，而是把地址转换推迟到程序真正要执行时才进行。因此装入内存后所有的地址依然是逻辑地址。这种方式需要一个重定位寄存器的支持。

在上面的例子中，装入模块指令的内容和内存里指令的内容都是“地址为79的存储单元中写入10”，重定位寄存器中存放装入模块存放的起始位置，如100。CPU在执行此条指令时，CPU取出逻辑地址“79”，然后和重定位寄存器中的内容相加，相加得到的结果（179）才是最终访问的地址。

采用动态重定位时允许程序在内存中发生移动。并且可将程序分配到不连续的存储区中：在程序运行前只需装入它的部分代码即可投入运行，然后在运行期间，根据需要动态申请分配内存。此外，便于程序段的共享，可以向用户提供一个比存储空间在得多的地址空间。（虚拟存储管理）

**2.从写程序到程序运行的步骤**

1.程序员用IDE软件写出程序的源文件。比如在源代码文件1里定义了main()函数，又在源代码文件2里定义了函数a()，在源代码文件3里定义了函数b()。C语言里，这些文件的后缀是“.c”，C++语言里，这些文件的后缀是“.cpp”。

2.经过编译，三个源代码文件会形成三个与它们一一对应的目标模块文件。目标模块中包含了代码所对应的指令，指令中的地址是逻辑地址，每个模块的逻辑地址都是相互独立从0开始的。如源文件1编译成的目标模块1的逻辑地址是0-99，源文件2编译成的目标模块2的逻辑地址是0-49，源文件3编译成的目标模块3的逻辑地址是0-29。C语言里，这些文件的后缀是“.o”。

编译：由编译器将用户源代码编译成若干个目标模块（编译就是把高级语言翻译为机器语言）

3.把所有的目标模块组装起来，形成一个完整的装入模块。在Windows系统中，装入模块就是.exe文件。把目标模块链接起来后形成的装入模块，有一个完整的逻辑地址。上例中，三个模块的地址空间大小之和是100+50+30=180，链接成的装入模块的完整逻辑地址就是0-179。当然，除了用户编写的源文件需要链接，用户调用的库函数的文件，如printf所在的文件也要一起编译链接起来。

链接：由链接程序将编译后形成的一组目标模块，以及所需库函数链接在一起，形成一个完整的装入模块。

4.把装入模块装入内存当中，装入后就确定了逻辑地址对应的实际物理地址，上例中就是N到N+179。然后就可以让程序运行了。

总之，程序从编写到执行，一共需要进行四步：编辑、编译、链接、装入。

上面介绍了三种装入方式，其实链接这一步也有三种方式：

（1）静态链接

程序运行前先将各目标模块及它们所需的库函数连接成一个完整的可执行文件（装入模块），之后不再拆开。

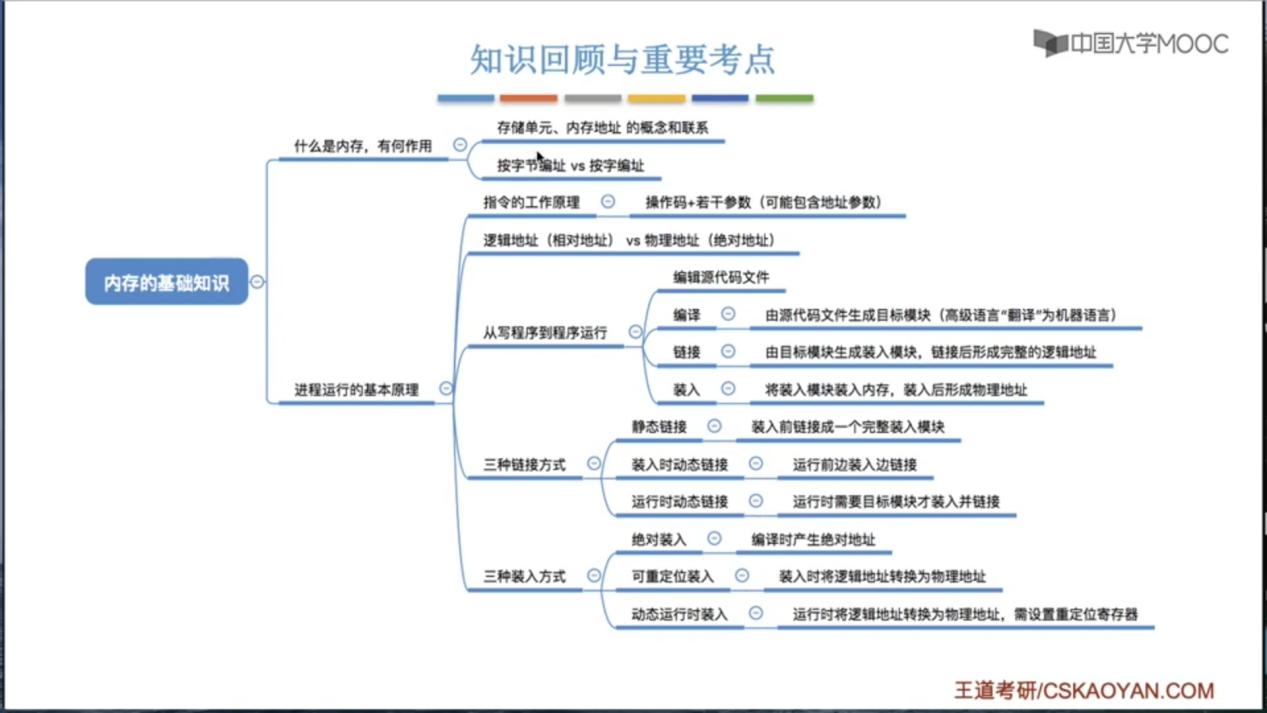
（2）装入时动态链接

将各目标模块装入内存时，边装入边链接的链接方式。程序的完整逻辑地址是一边装入一边形成的。

（3）运行时动态链接

在程序中执行中需要该目标模块时，才对它进行链接。其优点是便于修改和更新，便于实现对目标模块的共享。比如刚开始是main()函数运行，就先把目标模块1装入内存，执行main的过程中发现main函数要调用a函数，于是再把目标模块2装入内存中，并且装入时同时进行链接。如果b函数在整个过程中都用不到的话，那目标模块3就可以不装入内存。

所以这种方式的优点是便于修改和更新，便于实现对目标模块的共享。灵活性更高，提升内存利用率。



**3.1.2内存管理的概念**

操作系统是系统资源的管理者，需要对内存进行管理。管理的工作主要有以下四种：

1.操作系统负责内存空间的分配与回收

2.操作系统需要提供某种技术从逻辑上对内存空间进行扩充，即实现虚拟性

3.操作系统需要提供地址转换功能，负责程序的逻辑地址与物理地址的转换（重定位）

为了使编程更方便，程序员写程序时应该只需要关注指令、数据的逻辑地址。而逻辑地址到物理地址的转换（这个过程称为重定位）应该由操作系统负责，这样就保证了程序员写程序时不需要关注物理内存的实际情况。

绝对装入方式在编译时就产生绝对地址，只适用于单道程序阶段，那时还没有产生操作系统。这个地址转换由编译器完成。

静态重定位（可重定位装入）在装入时将逻辑地址转换为物理地址，装入程序是操作系统的一部分，这种方式用于早期的多道批处理操作系统

动态重定位（动态运行时装入）运行时将逻辑地址转换为物理地址，需设置重定位寄存器，这种方式是现代操作系统采用的。

4.操作系统需要提供内存保护功能。保证各进程在各自存储空间内运行，互不干扰。

假设某进程的逻辑地址空间为0-179，实际物理地址空间为100-279

内存保护可采取两种方法：

（1）在CPU中设置一对上、下限寄存器，存放进程的上、下限地址。进程的指令要访问某个地址时，CPU检查是否越界。

上例中，上限寄存器中存放100，下限寄存器中存放279。CPU访问某个地址时，分别与这两个寄存器对比，在两个寄存器之间的地址才可以访问

（2）采用重定位寄存器（又称基址寄存器）和界地址寄存器（又称限长寄存器）进行越界检查。重定位寄存器中存放的是进程的起始物理地址。界地址寄存器中存放的是进程的最大逻辑地址。

上例中，重定位寄存器中存放100，界地址寄存器中存放179。若CPU要访问逻辑地址为80的存储单元，首先将逻辑地址与界地址寄存器中的值对比。没有超过才能认为地址合法，如果超过了就抛出一个越界异常。如果没有超过，则将逻辑地址与重定位寄存器中的地址相加，得到实际的物理地址80+100=180。

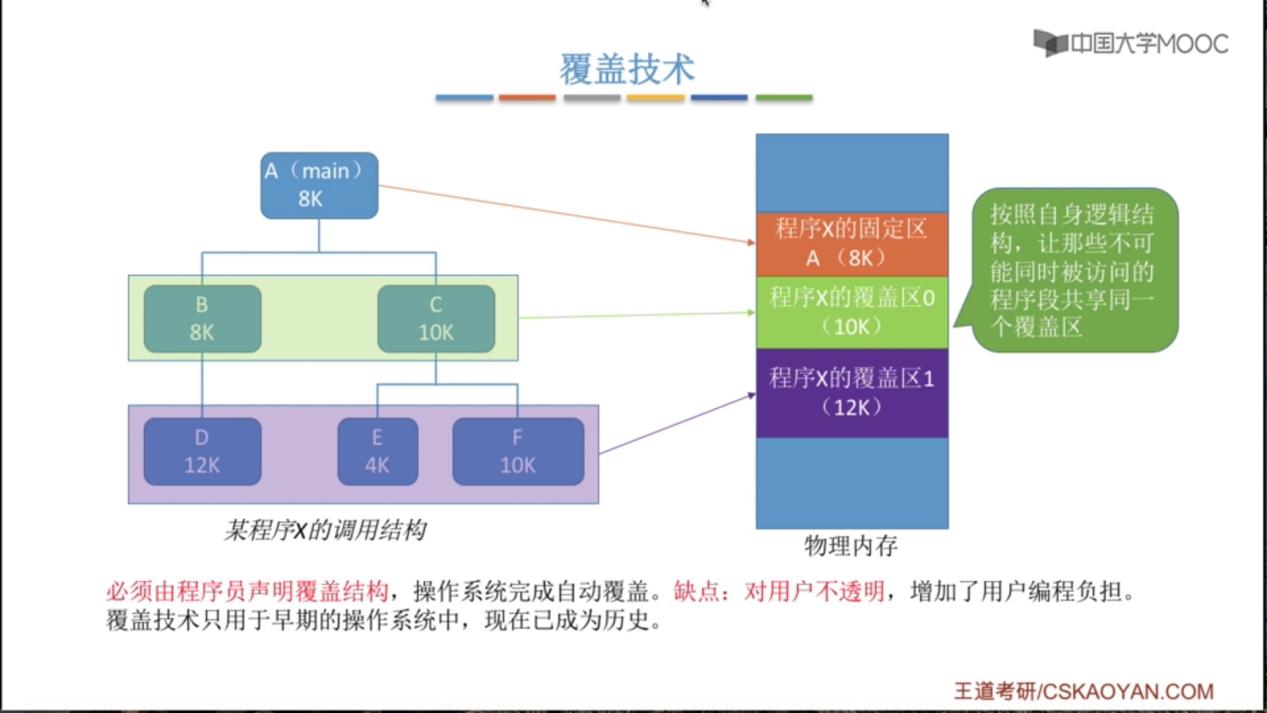
**3.1.3覆盖与交换**

覆盖和交换是两种实现内存空间扩充的技术

1.覆盖技术

思想：将程序分为多个模块。常用的模块常驻内存，不常用的模块在需要时调入内存。

内存分为一个“固定区”和若干个“覆盖区”。需要常驻内存的段放在“固定区”中，调入后不再调出（除非运行结束），不常用的段放在覆盖区，需要用到时调入内存，用不到时调出内存。



假设某程序X的调用结构如上，A模块调用B模块（8K）和C模块（10K）。但又不同时调用B和C模块。于是就可以让这两个模块共享同一个大小为10K的覆盖区。A模块经常使用，故放在固定区中。

这种方式必须由程序员声明覆盖结构，由操作系统完成自动覆盖。缺点在于对用户不透明，增加了用户编程负担。因此覆盖技术已被淘汰。

2.交换技术

思想：内存空间紧张时，系统将内存中某些进程暂时换出外存，把外存中某些已具备运行条件的进程换入内存（进程在内存与磁盘间动态调度）

中级调度是为了实现交换技术而实现的一种调度策略，要决定将哪个处于挂起状态的进程重新调回内存。

问题：

（1）应该在外存的什么位置保存被换出的进程

具有对换功能的操作系统中，通常把磁盘空间分为文件区和对换区两部分。文件区主要用于存放文件，主要追求存储空间的利用率，因此对文件区空间的管理采用离散分配方式；对换区空间只占磁盘空间的小部分，被换出的进程数据就存放在对换区。由于对换的速度直接影响到系统的整体速度，因此对换区空间的管理主要追求换入换出速度，因此通常对换区采用连续分配方式。总之，对换区的I/O速度比文件区更快。

（2）什么时候应该交换

交换通常在许多进程运行且内存吃紧时进行，而系统负荷降低就暂停。例如：在发现许多进程运行时经常发生缺页，就说明内存紧张，此时可以换出一些进程；如果缺页率明显下降，就可以暂停换出。

（3）应该换出哪些进程

可优先换出阻塞进程；可换出优先级低的进程；为了防止饥饿有的系统还会考虑进程在内存中的驻留时间

PCB会常驻内存，不会被换出外存。

**3.1.4连续分配管理方式**

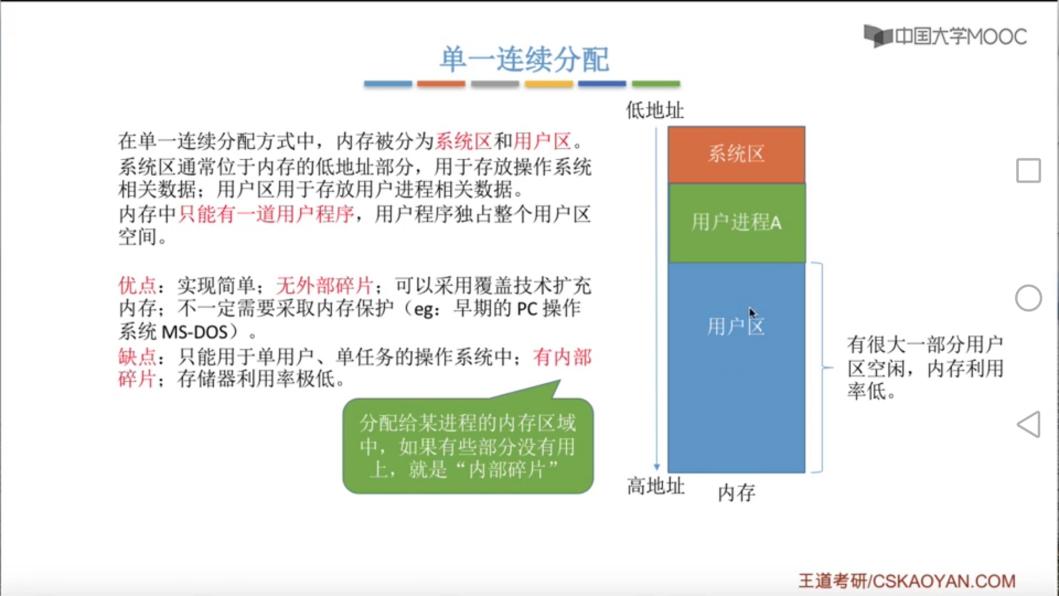
内存空间的分配与回收有两种方式：连续分配管理方式和非连续分配管理方式。连续分配，指为用户进程分配的必须是一个连续的内存空间。

连续分配管理方式又分为三种：单一连续分配、固定分区分配、动态分区分配

**1.单一连续分配**

在单一连续分配方式中，内存被分为系统区和用户区。系统区通常位于内存的低地址部分，用于存放操作系统的相关数据；用户区用于存放用户进程相关数据。

内存中只能有一道用户程序，用户程序独占整个用户区空间。



优点：实现简单，无外部碎片。可以采用覆盖技术扩充内存；不一定需要采取内存保护。（有的也有内存保护机制。但可以没有内存保护，如早期的微软公司的PC操作系统MS-DOS就没有）。用户程序只有一个，即使出问题也只影响本身。或者就算用户程序的数据越界把操作系统的数据损坏了，只需重启计算机来恢复数据。

缺点：只能用于单用户、单任务的操作系统中；有内部碎片，存储器利用率极低，有可能有很大一部分用户区空闲

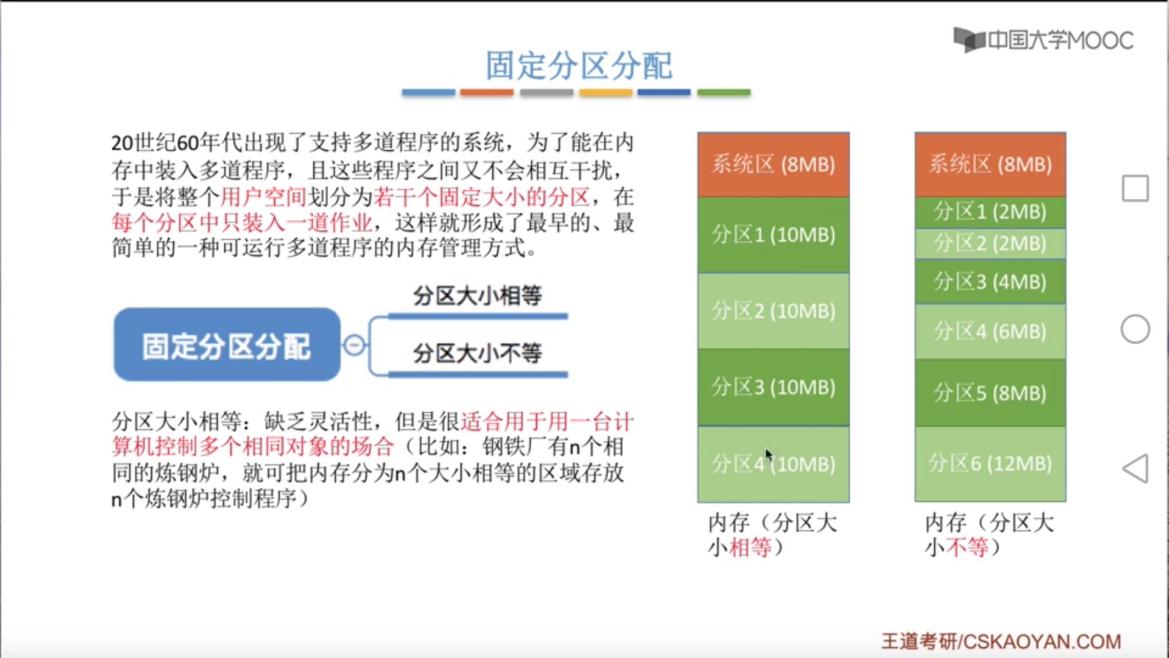
内部碎片：分配给进程的内存区域中，如果有些部分没有用上，就是“内部碎片”

外部碎片：内存中的某些空闲分区由于太小而难以利用

**2.固定分区分配**

将整个用户空间划分为若干个固定大小的分区，在每个分区中只装入一道作业，这样就形成了最早的、最简单的可运行多道程序的内存管理方式

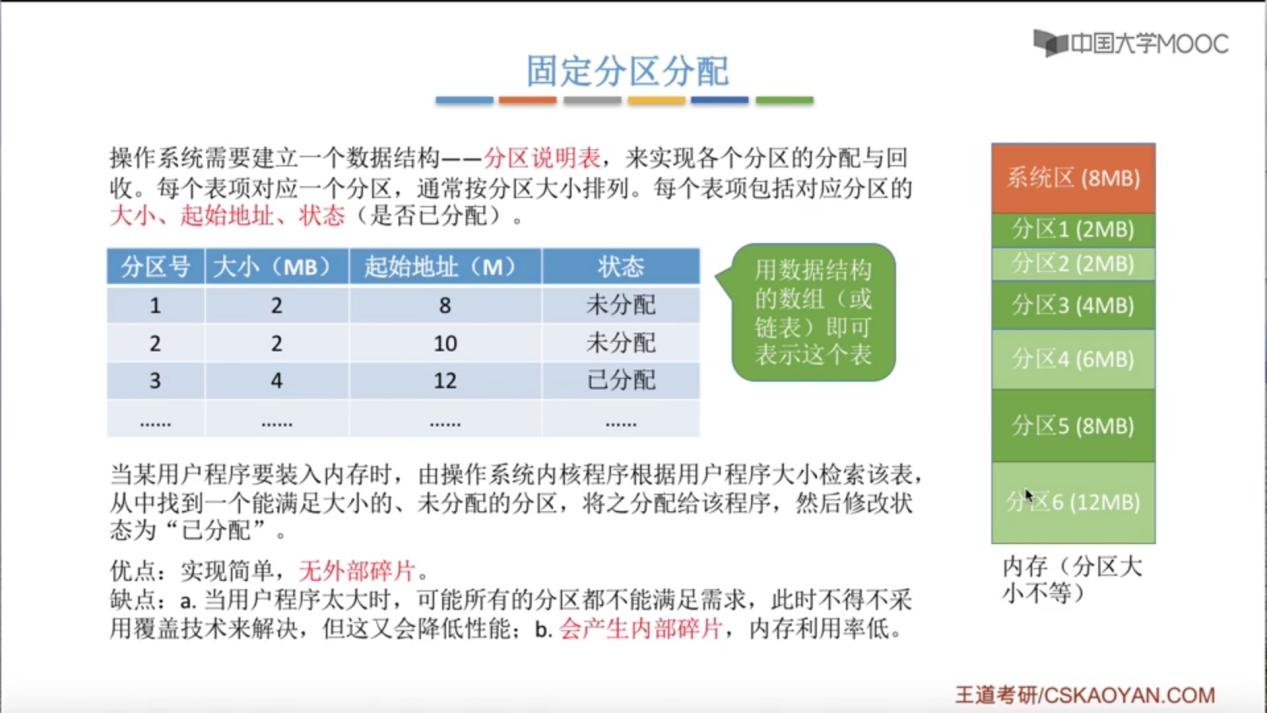
分区的划分也有两种方式：分区大小相等、分区大小不等

分区大小相等的情况下：缺乏灵活性，但很适合用于一台计算机控制多个相同对象的场合

分区大小不等的情况下：增加了灵活性，可以满足不同大小的进程需求。根据常在系统中运行的作业大小情况进行划分

（比如：划分多个小分区、适量中等分区、少量大分区）

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。每个表项包括对应分区的大小、起始地址、状态（是否已分配）。



当某用户程序要装入内存时，由操作系统内核程序根据用户程序大小检查该表，从中找到一个能满足大小的、未分配的分区，将之分配给该程序，然后修改状态为“已分配”。

优点：实现简单，无外部碎片。

缺点：1.当用户程序太大时，可能所有分区都不能满足要求，此时不得不采用覆盖技术解决，但这又会降低性能。2.会产生内部碎片，内存利用率低。

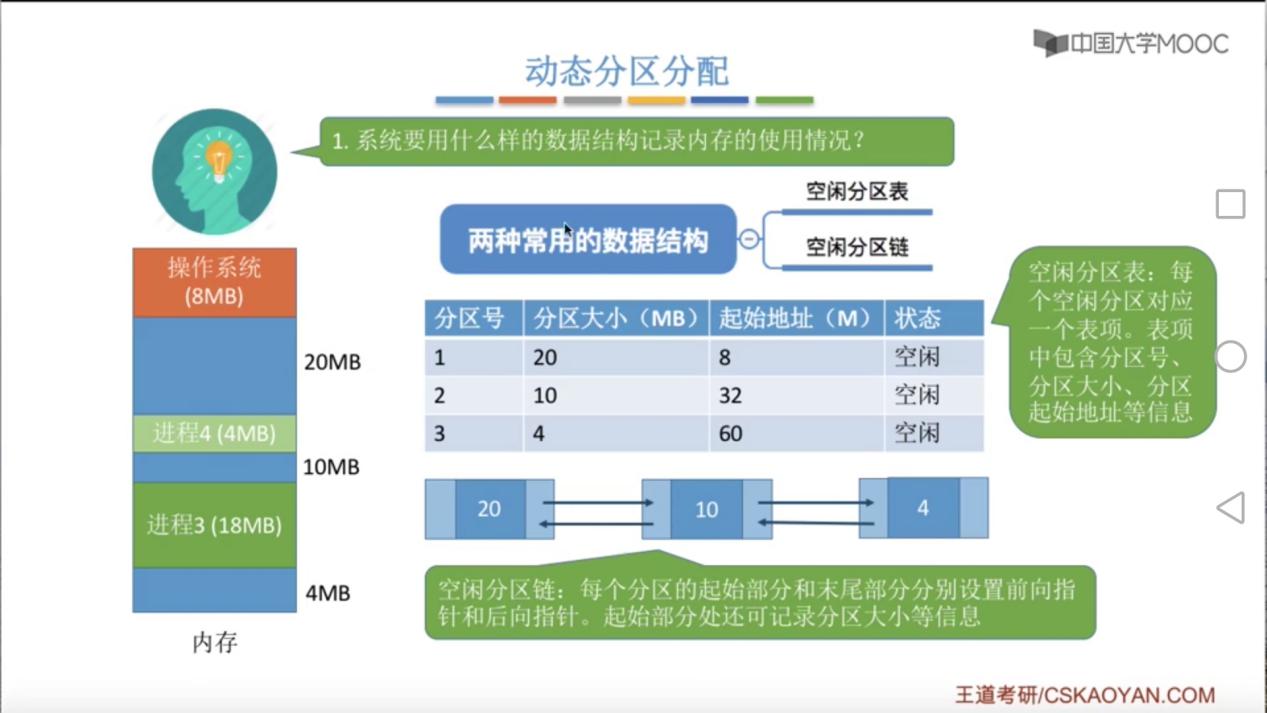
**3.动态分区分配**

动态分区分配又称为可变分区分配。这种分配方式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建立分区，并使分区的大小正好适合进程的需要。因此系统分区的大小和数目是可变的。

这种方式用来记录内存的使用情况的数据结构是空闲分区表或空闲分区链

空闲分区表：每个空闲分区对应一个表项。表项中包含分区号、分区大小、分区起始地址等信息

空闲分区链：每个分区的起始部分和末尾部分分别设置前向指针和后向指针。起始部分处还可记录分区大小等信息



在分配和回收时，需对该数据结构做调整:

1.分配一块空闲分区的一部分

如上例中，某进程需4MB的内存，若分配算法决定将空闲分区1中的4MB分给它。由于该分区大小为20MB，大于所需的4MB，因此只需修改该空闲分区的起始地址和大小即可。

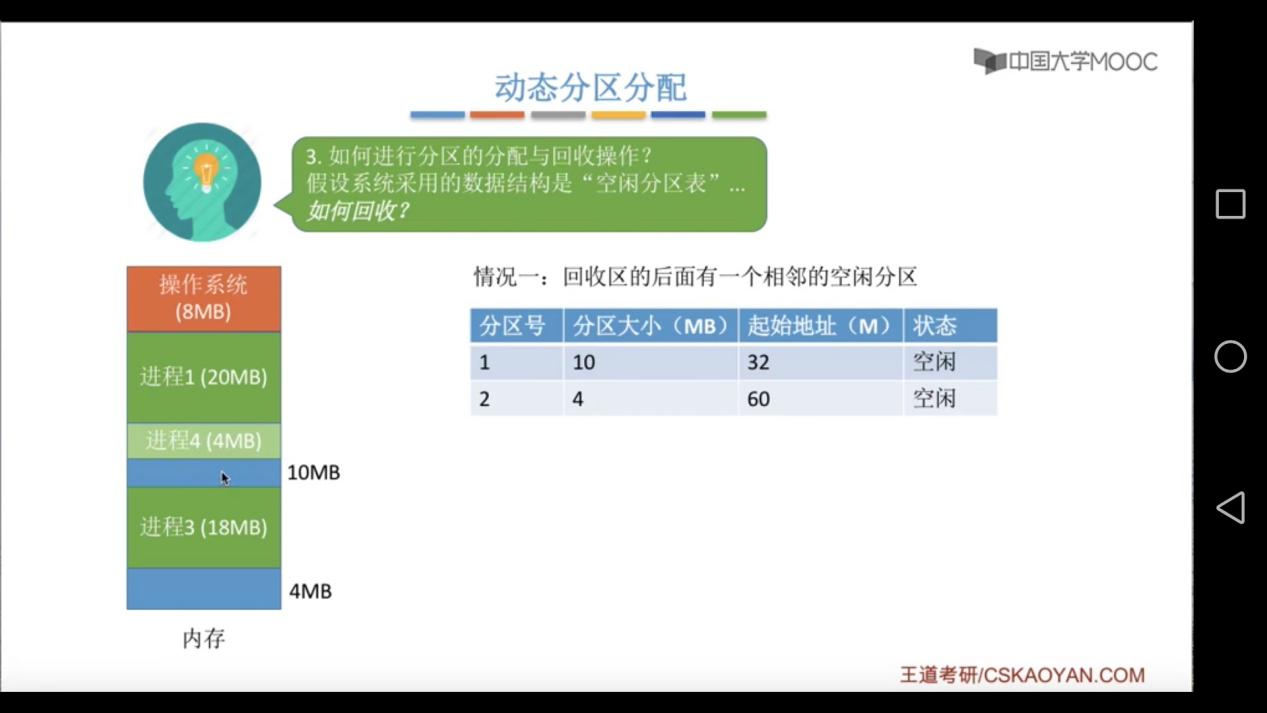
|  |  |  |  |
| --- | --- | --- | --- |
| 分区号 | 分区大小 | 起始地址 | 状态 |
| 1 | 16 | 12 | 空闲 |
| 2 | 10 | 32 | 空闲 |
| 3 | 4 | 60 | 空闲 |

2.分配一块空闲分区的全部

如上例中，若分配算法决定将空闲分区3中的4MB分给它。由于该分区大小正好是4MB。需要把该空闲分区表的表项删除（若采用空闲分区链，需将链中的结点删除）

|  |  |  |  |
| --- | --- | --- | --- |
| 分区号 | 分区大小 | 起始地址 | 状态 |
| 1 | 20 | 8 | 空闲 |
| 2 | 4 | 60 | 空闲 |

3.回收区的后面有一个相邻的空闲分区



在此例中，若要回收进程4的所有4MB内存，由于它后面有10MB的空闲分区，需要将回收的4MB与后面的10MB合并。即修改分区1的分区大小和起始地址。

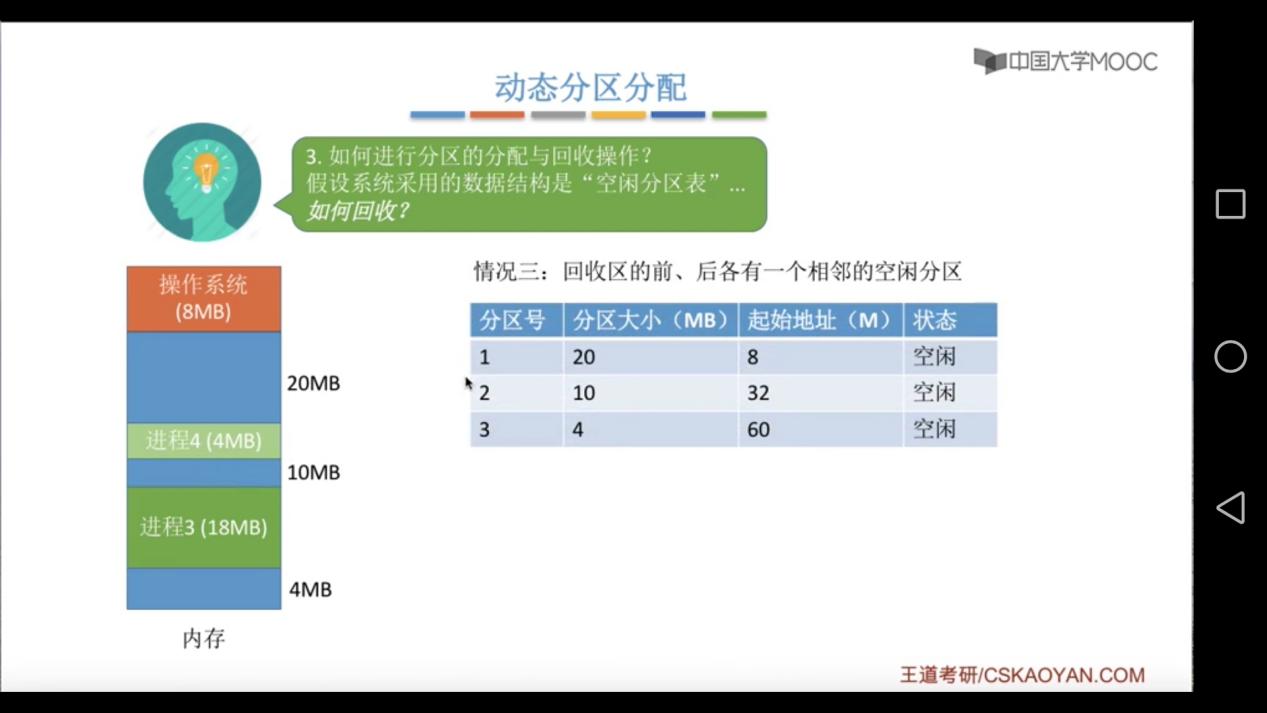
|  |  |  |  |
| --- | --- | --- | --- |
| 分区号 | 分区大小 | 起始地址 | 状态 |
| 1 | 14 | 28 | 空闲 |
| 2 | 4 | 60 | 空闲 |

4.回收区的前面有一个相邻的空闲分区

和上例类似，需将回收区与其前面的空闲区合并。需修改空闲区的分区大小。

5.回收区的前后各有一个相邻的空闲分区

需要将这三个分区合并，即将空闲分区表中的两项再加上回收区合并成一个空闲分区。

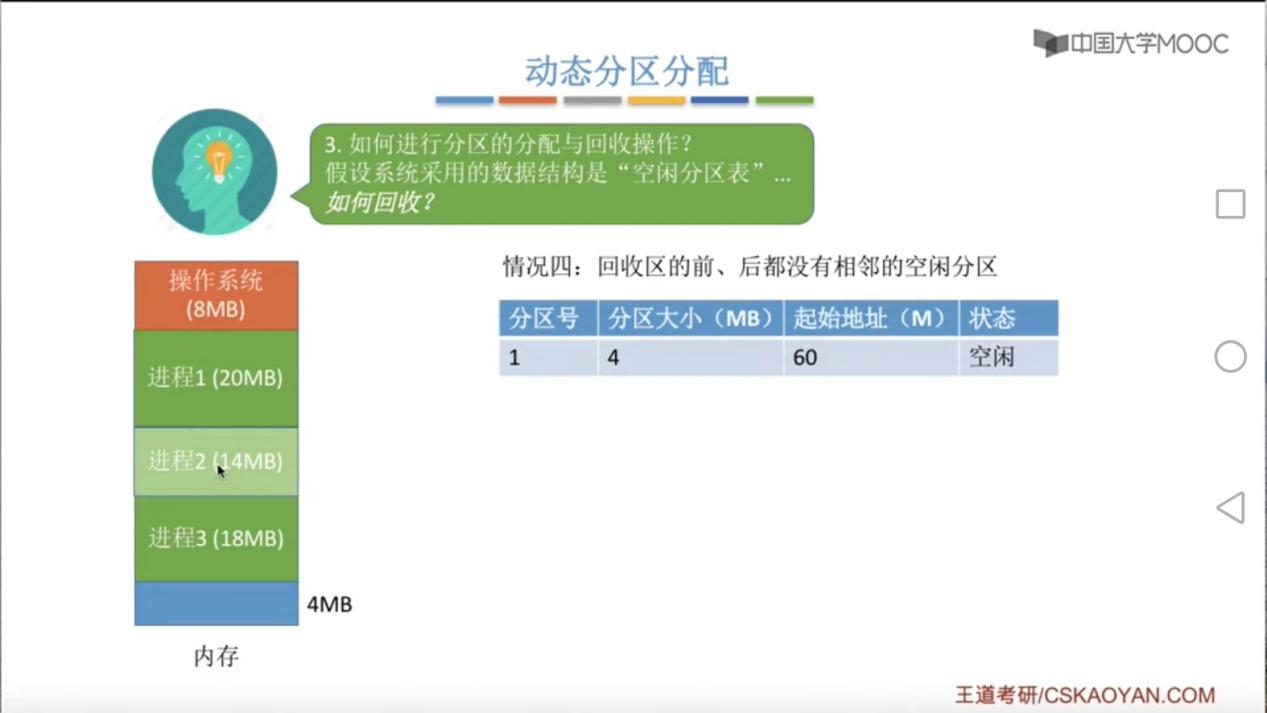


本例中回收进程4的4MB内存空间，要将相邻的前后空闲分区表项合并

|  |  |  |  |
| --- | --- | --- | --- |
| 分区号 | 分区大小 | 起始地址 | 状态 |
| 1 | 34 | 8 | 空闲 |
| 2 | 4 | 60 | 空闲 |

6.回收区的前后都没有相邻的空闲分区

此时要增加一个空闲分区的表项



本例中回收进程2的14MB内存空间，要插入一个新空闲分区表项

|  |  |  |  |
| --- | --- | --- | --- |
| 分区号 | 分区大小 | 起始地址 | 状态 |
| 1 | 14 | 28 | 空闲 |
| 2 | 4 | 60 | 空闲 |

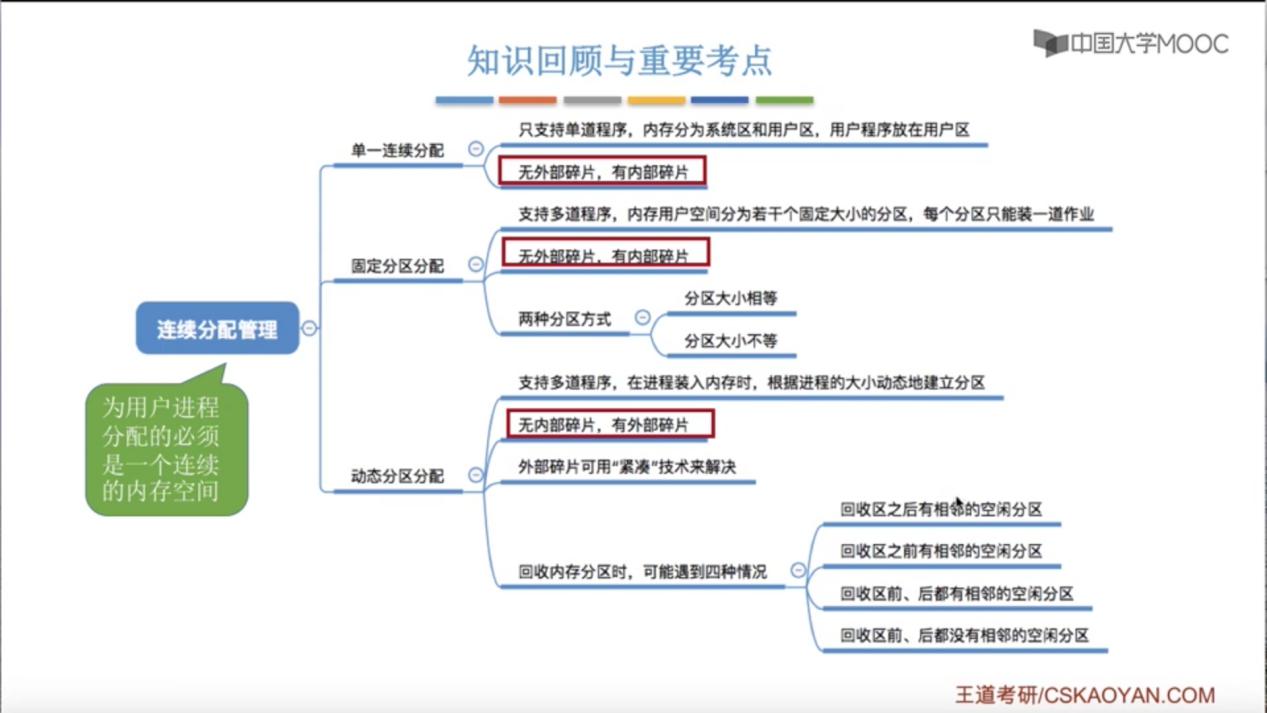
注:各表项的顺序不一定按照地址递增顺序排列，具体的排列方式需要依据动态分区分配算法来确定。

总之，回收后如果发现有些空闲分区是相连的，那么就需合并。

动态分区分配没有内部碎片，但是有外部碎片，内存中某些空闲分区可能因为大小而难以利用。

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“碎片”不能满足进程的需求，可以通过紧凑（Compaction）技术来解决外部碎片：把进程挪位攒到一起，从而使碎片合并起来。

动态分区分配方式应采用动态重定位的装入方式。可重定位装入和绝对装入都不允许进程挪动。挪动后应修改进程的起始位置（在PCB中），在进程上CPU运行前，会把PCB中进程的起始地址写入重定位寄存器（基址寄存器）中。



**3.1.5动态分区分配算法**

动态分区分配算法解决的问题是在动态分区分配方式中，当很多个空闲分区都能满足需求时，应选择哪个分区进行分配。有四种算法：首次适应算法(first fit)，最佳适应算法(best fit)，最坏适应算法(worst fit)，邻近适应算法(next fit)。

1-首次适应算法

算法思想：每次都从低地址开始查找，找到第一个能满足大小的空闲分区

如何实现：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（或空闲分区表），找到大小能满足要求的第一个空闲分区。

2-最佳适应算法

算法思想：由于动态分区分配是一种连续分配方式，为各进程分配的空间必须是连续的一整片区域。因此为了保证当“大进程”到来时能有连续的大片空间，可以尽可能多地留下大片的空闲区，优先使用更小的空闲区。

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，找到大小能满足要求的第一个空闲分区。

每次分配后，需将分配出去的空闲分区按剩余的空间大小重新排列。

缺点：每次都选最小的分区进行分配，会留下越来越多的、很小的难以利用的内存块。因此这种方法会产生很多的外部碎片。

3-最坏适应算法（又称最大适应算法）

算法思想：为了解决最佳适应算法的问题-太多的小碎片。可以在每次分配时优先使用最大的连续空闲区，这样分配后剩余的空闲区就不会太小，更方便使用。

如何实现：空闲分区按容量递减次序链接。每次分配内存时顺序查找空闲分区链，找到大小能满足要求的第一个空闲分区。

缺点：每次都选最大的分区进行分配，虽然可以让分配后留下的空闲区更大更可用，但这种方式会导致较大的连续空闲区被迅速用完。之后有“大进程”到达就没有内存分区可用了。

4-邻近适应算法

算法思想：首次适应算法每次都从链头开始查找，这可能会导致低地址部分出现很多小的空闲分区，而每次分配查找时都要经过这些分区，增加了查找的开销。如果每次都从上次查找结束的位置开始检索，就能解决上述问题。

如何实现：空闲分区以地址递增的顺序排列（可排成一个循环链表）。每次分配内存时从上次查找结束的位置开始查找空闲分区链，找到大小能满足要求的第一个空闲分区。

首次适应算法每次都要从头查找，每次都需要检索低地址的小分区。但这种规则也决定了当低地址部分有更小的分区可以满足需求时更有可能用到低地址部分的小分区，也更有可能把高地址部分的大分区保留下来，首次适应算法也具有最佳适应算法的优点。

与此相对，邻近适应算法的规则会导致无论高地址低地址，空闲分区都有相同的概率被使用，也就导致了高地址部分的大分区更可能被划分为小分区，最后导致无大分区可用。所以，邻近适应算法也具有最坏适应算法的缺点。

综合来看，首次适应算法的性能最好，算法开销小，回收分区后一般不需要对空闲分区队列重新排序

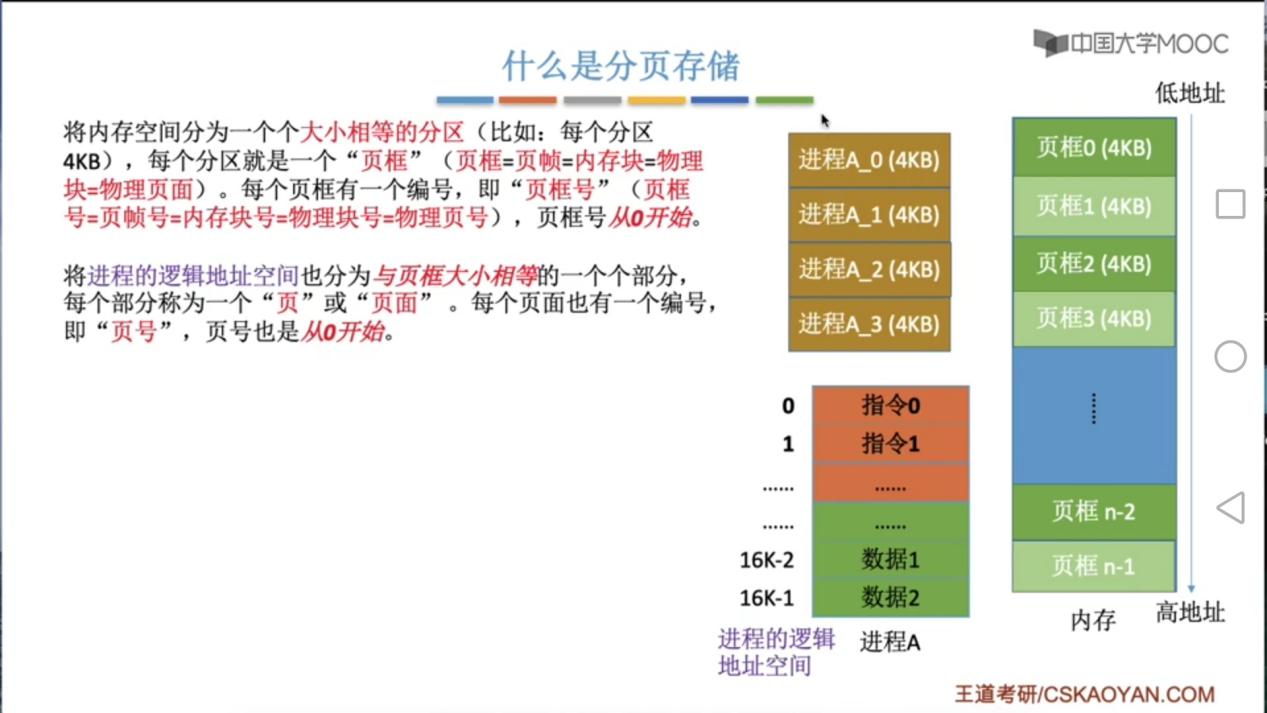
**3.1.6基本分页存储管理的基本概念（重点内容）**

内存的分配有两种方式，连续分配管理方式和非连续分配管理方式。非连续分配方式，就是为用户进程分配的可以是一些分散的内存空间。

非连续分配管理方式分成三种：基本分页存储管理、基本分段存储管理、段页式存储管理

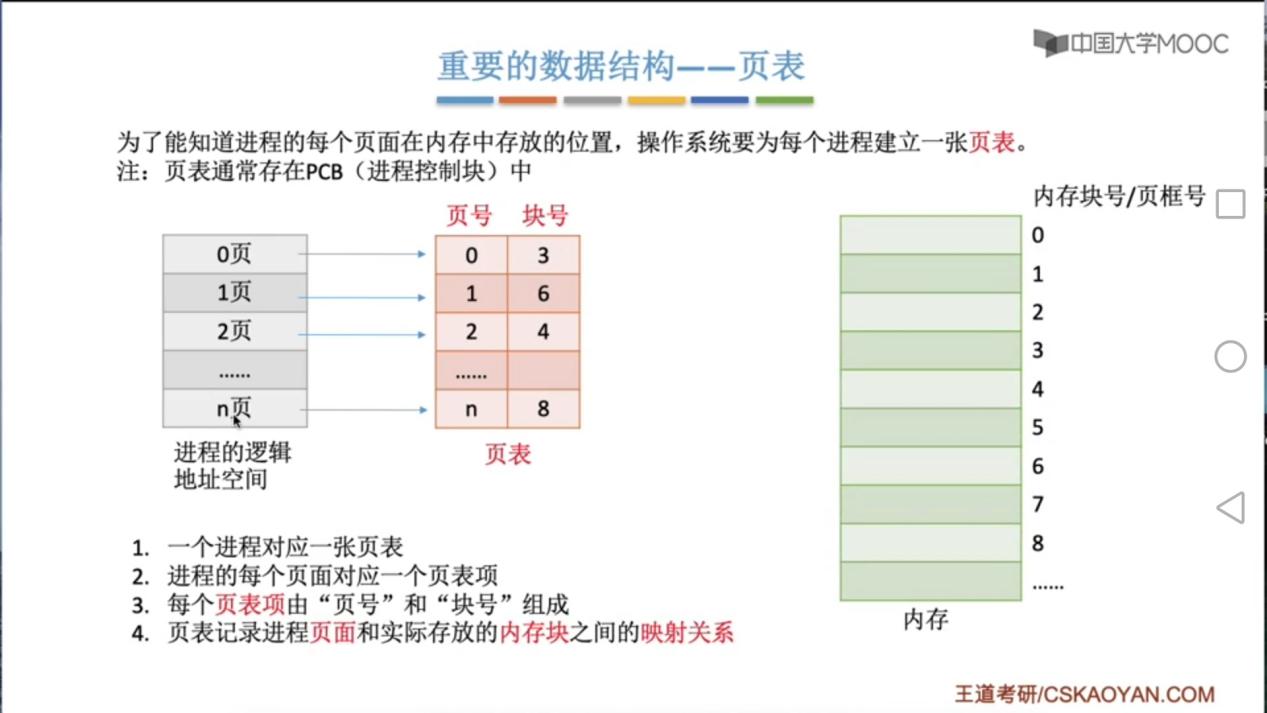
**1.分页存储**

分页存储方式中，系统把内存分成一个个大小相等的分区（比如每个分区4KB），每个分区就是一个“页框”（也叫页帧，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。

同时将进程的*逻辑地址空间*也分为与页框大小相等的一个个部分，每个部分称为一个“页”或“页面”。每个页面也有一个编号，即“页号”，页号也是从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。

各个页面不必连续存放，可以放到不相邻的各个页框中。

为了能知道进程的每个页面在内存中存放的位置，操作系统要为每个进程建立一张页表。页表通常存在PCB中。

1.一个进程对应一张页表

2.进程的每个页面对应一个页表项

3.每个页表项由“页号”和“块号”组成

4.页表记录进程页面和实际存放的内存块之间的映射关系

例：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？

内存块的大小=页面大小=4KB=2^12B

4GB的内存总共被分为2^32/2^12=2^20个内存块，内存块号的范围是0~2^20-1。故内存块号占20个bit。由于系统分配内存都是以字节为单位，故20个bit至少要用3B来表示块号。

页表项连续存放，因此页号是隐含的，不占存储空间，相当于数组的下标。

若每个页表项占3B，页表中各页表项从内存地址为X的地方开始连续存放，则页号为i的页表项所在的地址为X+3i。

页表记录的只是内存块号，而不是内存块的起始地址。内存块起始地址=内存块号\*块的大小

**2.地址转换**

进程在内存中连续存放时，只需将目标逻辑地址与重定位寄存器中的内容（进程在内存中起始位置）相加就能得到物理地址。逻辑地址可理解为相对于起始地址的“偏移量”。

但在分页存储中，虽然进程的各个页面是离散存放的，但页面内部是连续的。

如果要访问逻辑地址A，则需要

（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，这是通过查询页表实现的

（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

设页面大小为S，则P = A/S（整数除法，只保留整数部分），W=A%S（W为余数）

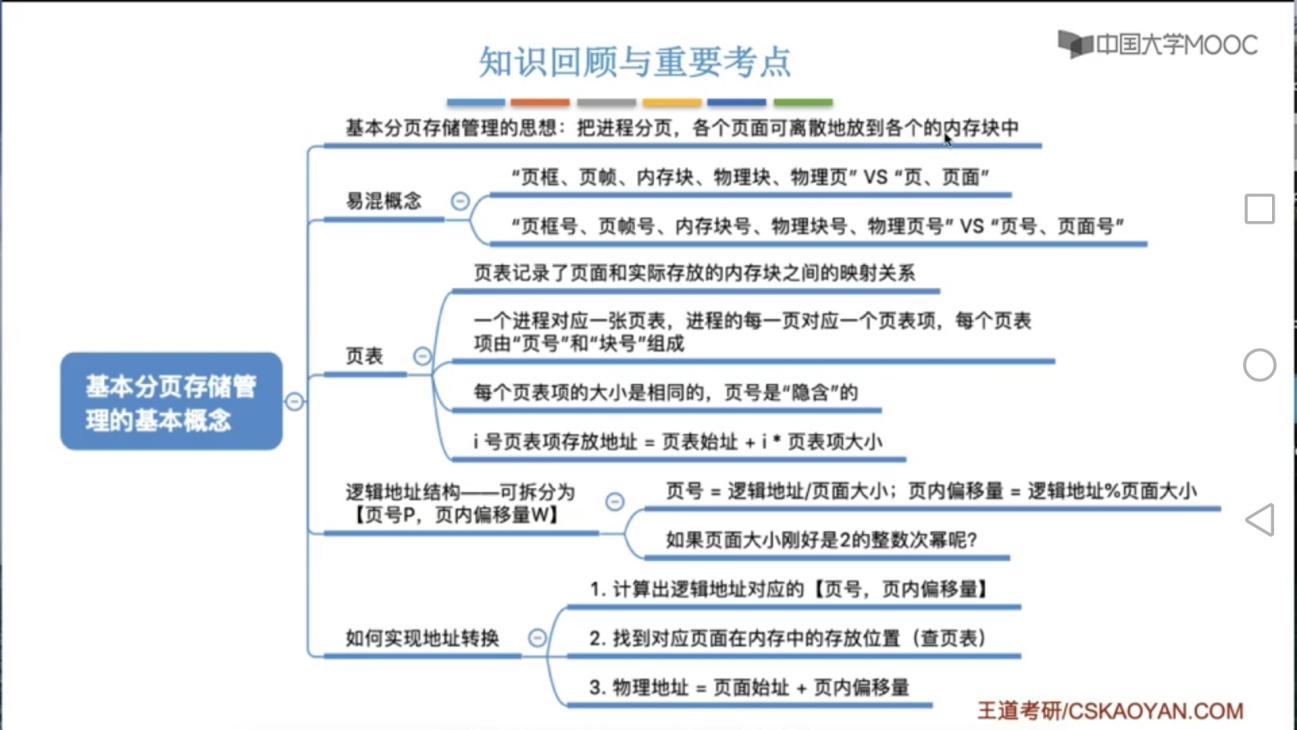
计算机内，地址是用二进制表示的，如果页面大小刚好是2的整数幂。则硬件可以很快速的把逻辑地址拆分成页号+页内偏移量

假设某计算机用32个二进制位表示逻辑地址，页面大小为4KB=2^12B。则逻辑地址的后12位就是页内偏移量，前20位就是页号。求它们的值的操作可以通过无符号左移、无符号右移实现。

结论就是：如果每个页面大小为2^K个字节，用二进制数表示逻辑地址，则末尾K位即为页内偏移量，其余部分就是页号。页面大小刚好是2的整数幂可以使逻辑地址拆分更加迅速，物理地址的计算也更加迅速。

如果有K位表示页内偏移量，则说明系统中一个页面的大小是2^K个内存单元

如果有M位表示页号，则说明在该系统中，一个进程最多允许有2^M个页面



**3.1.7基本地址变换机构**

基本地址变换机构可以借助进程的页表将逻辑地址转换为物理地址

通常会在系统中设置一个页表寄存器（PTR），存放页表在内存中起始地址F和页表长度M。进程未执行时，页表的始址和页表长度放在PCB中，当进程被调度时，操作系统内核会把它们放到页表寄存器中。

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）由逻辑地址A计算页号P和页内偏移量W

（2）检查页号P是否超过页表长度M。如果P>=M，则产生越界中断，否则继续执行

（3）找对应页表项的地址：页表项地址=页表始址F+页号P\*页表项长度，取出该页表内容b，即为内存块号

（4）计算E=b\*L+W，用得到的物理地址E去访问内存（若内存块号、页内偏移量是用二进制表示的，那么二者拼接起来就是最终的物理地址）

例：若页面大小为1KB，页号2对应的内存块号b=8，将逻辑地址A=2500转换为物理地址

A=2500的二进制表示方法为100111000100，页面大小为1KB=2^10B，故页内偏移量占10位。A的页内偏移量是二进制表示法的后10位，即0111000100。从而页号是前2位“10”，在十进制中是2。由于页号2对应的内存块号b=8，用二进制表示是1000。所以实际的物理地址就是1000拼上0111000100，即10000111000100，十进制表示是8644。

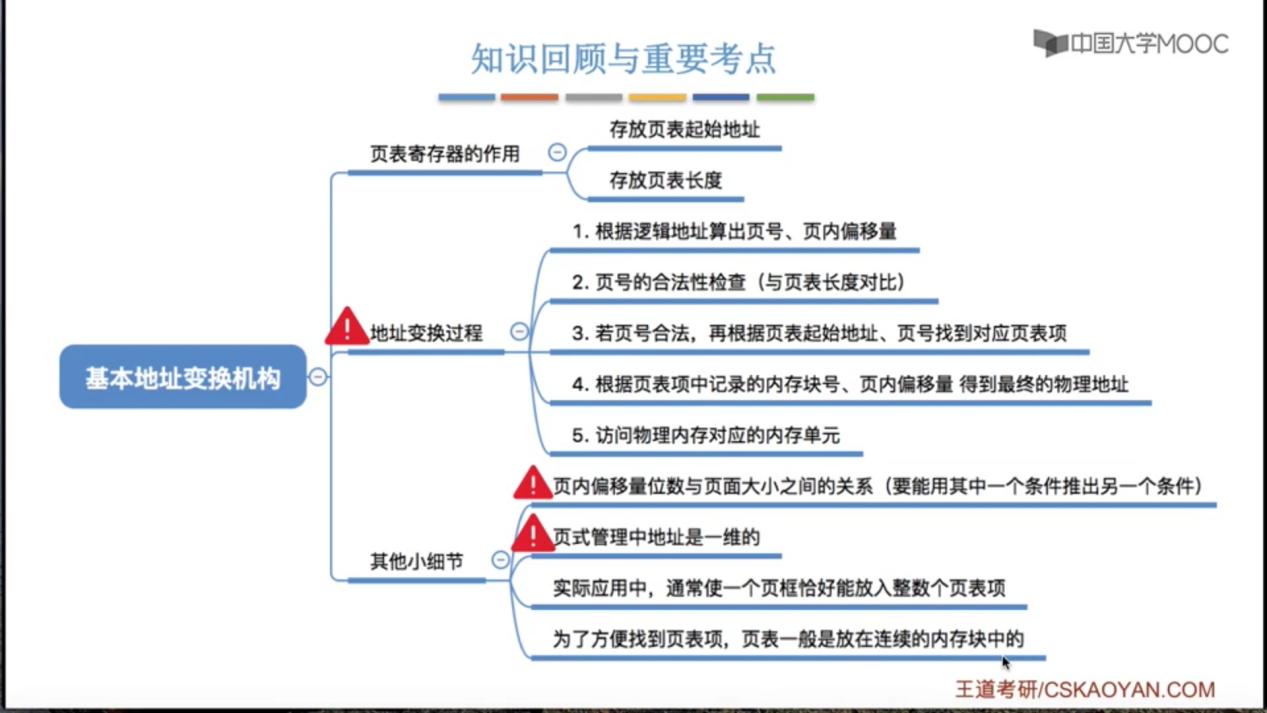
在分页存储管理系统中，只要确定了每个页面大小，逻辑地址结构就确定了。因此页式管理中地址是一维的。即只要给一个逻辑地址，系统就可以自动算出页号、页内偏移量。

在页表中，每个页表项长度是相同的，页号是“隐含”的，各页表项会按顺序连续地存放在内存中。在上面举的例子中，系统物理内存4GB，页面大小4KB，则每个页表项的大小是3B。

这样，一个页框能存放4096/3=1365个页面项（第一页存放页号为0-1364），而且会有1B的页内碎片。所以查询页号1365的页表项被放入第2页且前一页有1B的页内碎片。该页表项所在的物理地址就是页表始址+3\*1365+1，这样做很不方便。

实际应用中，为了方便页表的查询，常常会让一个页表项占更多的字节，使得每个页面恰好可以装得下整数个页表项。所以上例中，每个页表项最好占用4字节。页表的第一个内存块能存放4KB/4B=1024个页表项，存放的页号是从0到1023。

为了方便找到页表项，进程页表通常是装在连续的内存块中。



**CPU从得到一个逻辑地址A到实际访问它对应的物理地址E的存储内容的过程中，需要访问2次内存。第一次访问是查页表，第二次访问是查地址为E的存储单元。**

**3.1.8具有快表（TLB）的地址变换机构**

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

当进程切换的时候，快表的内容需要被清空。

引入快表中的地址变换过程：

（1）CPU给出逻辑地址，由硬件算得页号、页内偏移量，将页号与页表长度对比判定是否越界，如果越界则发出一个越界中断，如果没有则将页号与快表中所有页号进行比较

（2）如果找到匹配的页号，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。在内存中找到页表项后，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。

基于局部性原理，一般来说快表的命中率可达90%以上。

例：某系统使用基本分页存储管理，并采用了具有快表的地址变换机构。访问一次快表耗时1us，访问一次内存耗时100us，若快表命中率为90%。那么访问一个逻辑地址的平均耗时是多少？

答：（1+100）\*0.9+（1+100+100）\*0.1=111us

有的系统支持快表和慢表同时查找，如果是这样，平均耗时应该是(1+100)\*0.9+(100+100)\*0.1=110.9us

**局部性原理**

时间局部性：如果执行了程序中的某条指令，那么不久后这条指令很有可能再次执行；如果某个数据被访问过，不久之后该数据很可能再次被访问（因为程序中存在大量的循环）

空间局部性：一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也很有可能被访问。（因为很多数据在内存中都是连续存放的）

每访问一个逻辑地址都要查询内存中的页表。由于局部性原理，可能连续很多次查到的都是同一个页表项

TLB只有页表项的副本，而普通Cache中可能会有其他各种数据的副本

**3.1.9两级页表**

单级页表存在两个问题：

a.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，失去了离散分配存储的优点

b.由局部性原理，没有必要让整个页表常驻内存，因为进程一段时间内可能只需要访问某几个特定的页面

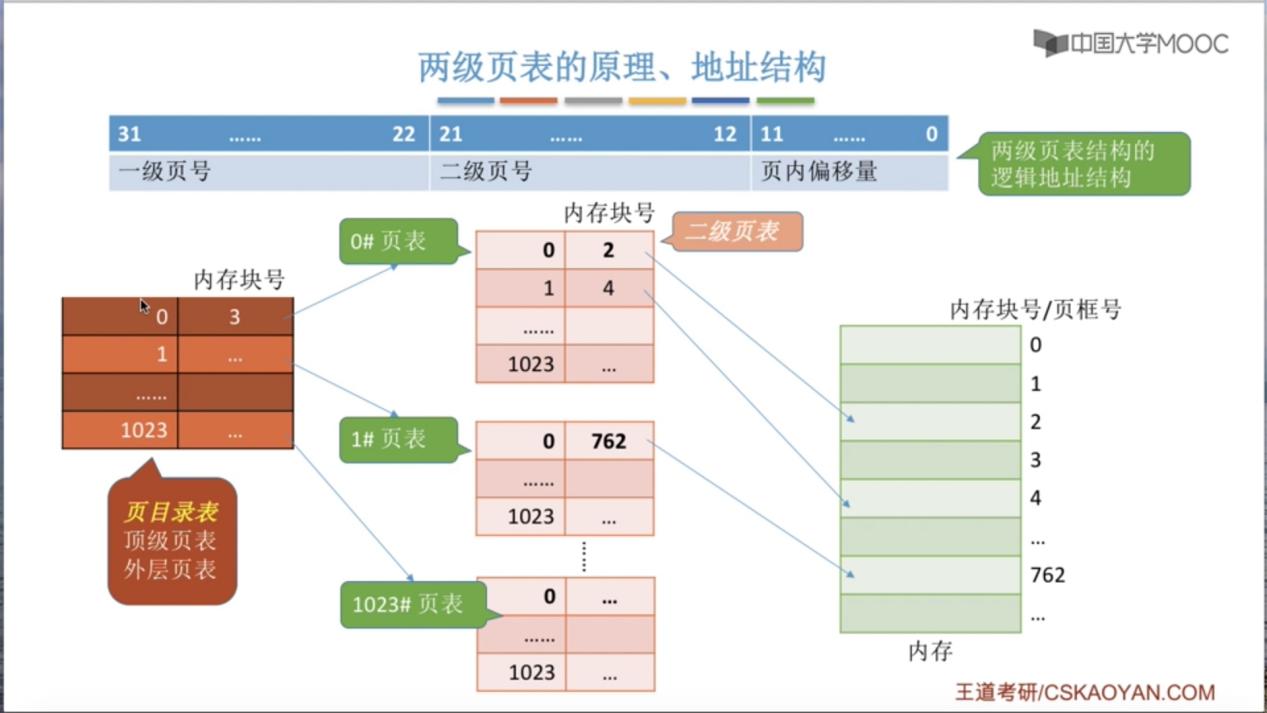
可将页表进行分组，使每个内存块刚好可以放入一个分组

例：某计算机系统按字节寻址，支持32位的逻辑地址，采用分页存储管理，页面大小为4KB，页表项长度为4B。

若没有多级页表机制，页内地址要用12位表示，剩余20位表示页号。因此该系统中用户进程最多有2^20页。一个进程的页表中最多有2^20个页表项，这些页表项要占据的空间大小为2^22B。所以要分配2^22B/(2^12B)=1024个连续的页框给这个页表。

但引入了两级页表机制后，页面大小为4KB，页表项长度为4B。则每个页框可存放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，再将各级离散地放到各个内存块中。每一块又可以叫二级页表。

这样，这要为离散分配的页表再建立一张目录，称为页目录表，也叫外层页表或顶层页表

这样32位的逻辑地址就可以拆分为10位一级页号+10位2级页号+12位页内偏移量

例：在上图示例中，将逻辑地址（0000000000，0000000001，111111111111）转换为物理地址，用十进制表示

（1）把逻辑地址拆成三部分

一级页号0000000000，二级页号0000000001，页内偏移量111111111111

（2）从PCB中读出页目录表始址，再根据一级页号查页目录表，找到下一级页表在内存中的存放位置。由一级页号0可知，对应的二级页表在内存块号为3的内存块中，可从中读出0#页表。

（3）根据二级页号查二级页表，找到最终想访问的内存块号。由二级页号1可知，对应的进程页面在内存块号为4的内存块中。

（4）给合页内偏移量得到物理地址：4\*4096+1023=17407

所以对于问题a，可以采用二级页表的方法解决。对于问题b，可以在需要访问页面时才把页面调入内存（虚拟存储技术）。可以在页表项中增加一个标志位，用于表示该页面是否已调入内存。若想访问的页面不在内存中，则产生缺页中断（内中断），然后将目标页面从外存调入内存。

注意：1.若采用多级页表机制，则各级页表的大小不能超过一个页面

例：某系统按字节编址，采用40位逻辑地址，页面大小为4KB，页表项大小为4B，假设采用纯页式存储，则要采用（）级页表，页内偏移量为（）位？

答：页内偏移量为12位，剩余28位是页号。页面大小4KB，页表项大小为4B，故每页因此面可存放2^10个页表项。因此每级页表对应页号应为10位。总共28位的页号至少要分为三级：

|  |  |  |  |
| --- | --- | --- | --- |
| 一级页号8位 | 二级页号10位 | 三级页号10位 | 页内偏移量12位 |

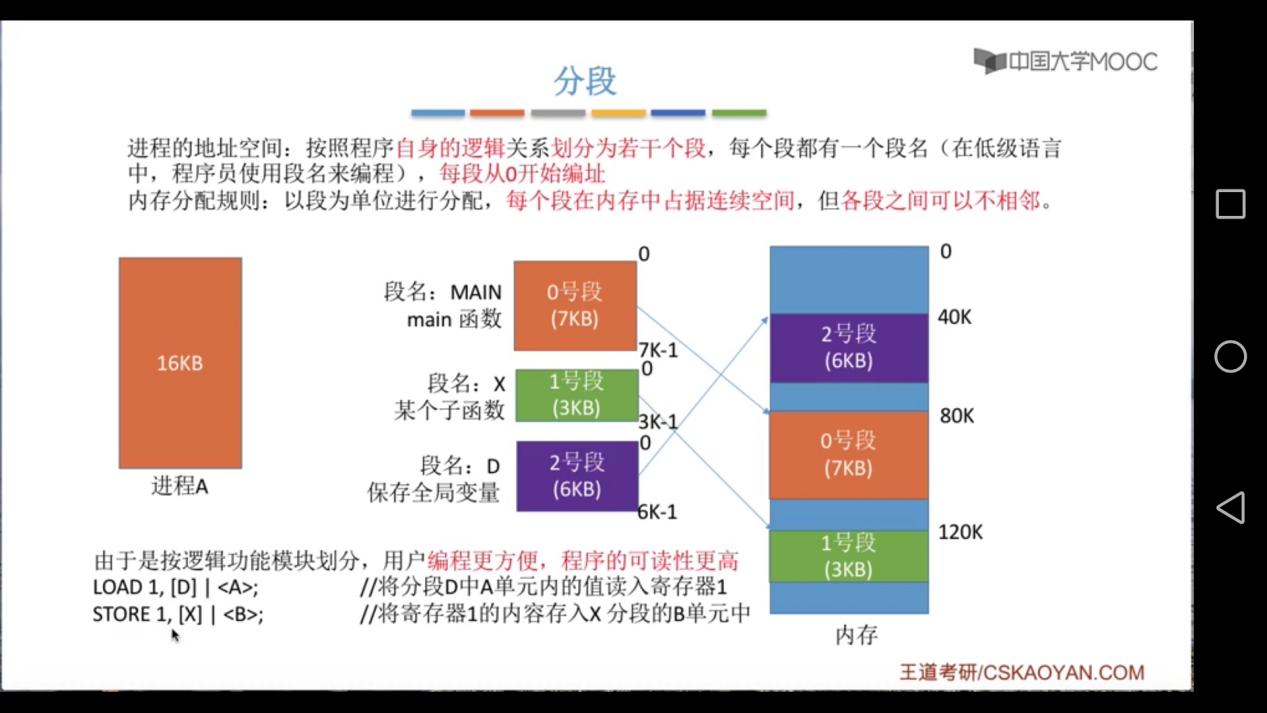
2.假设没有快表机构，两级页表的访存次数为3：第一次访问页目录表，第二次访问二级页表，第三次访问目标存储单元。总之，N级页表机制的访存次数为N+1。

**3.1.10基本分段存储管理**

**1.分段**

分段：进程的地址空间按程序自身的逻辑关系划分为若干个段，每个段都有一个段名（在低级语言中，程序员使用段名来编程），每段从0开始编址。

内存分配的规则：以段为单位进行分配，每个段在内存中占据连续空间，但各段之间可以不相邻



优点：由于按逻辑功能模块划分，用户编程更方便，程序的可读性更高。

编译程序会将段名转换为段号

分段系统的逻辑地址结构由段号和段内地址所组成。如：

|  |  |
| --- | --- |
| 31 ....... 16 | 15...... 0 |
| 段号 | 段内地址 |

段号的位数决定了每个进程最多可以分几个段

段内地址位数决定了每个段的最大长度是多少

在上例中，若系统按字节寻址，则段号占16位，每个进程最多有2^16=64K个段

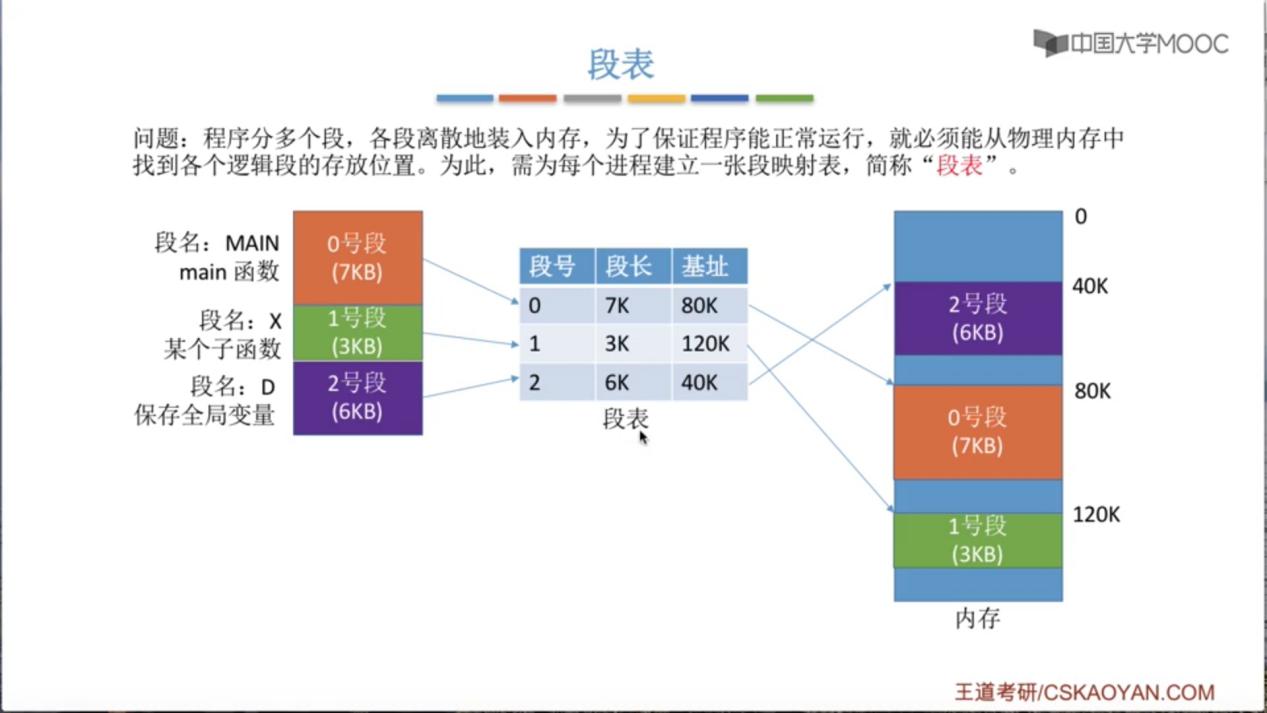
段内地址占16位，因此每个段的最大长度是2^16=64KB

汇编指令如LOAD 1,[D]|<A>; STORE 1,[X]|<B>

写程序时使用的段名[D]、[X]会被编译程序翻译成对应段号，<A>单元、<B>单元会被编译程序翻译成段内地址。

**2.段表**

程序分多个段，各段离散地装入内存，为了保证程序能正常运行，就必须能从物理内存中找到各个逻辑段的存放位置。为此，需为每个进程建立一张段映射表，简称“段表”。



1.每个段对应一个段表项，其中记录了该段在内存中的起始位置（又称“基址”）和段的长度

2.各个段表项的长度是相同的。例如：

某系统按字节寻址，采用分段存储管理，逻辑地址结构为段号16位+段内地址16位，因此用16位即可表示最大段长。物理内存大小为4GB（可用32位表示整个物理内存地址空间）。因此可以让每个段表项占16+32=48位，即6B。由于段表项长度相同，因此段号可以是隐含的，不占存储空间。若段表存放的起始地址为M，则K号段对应的段表项存放的地址为M+6\*K

**3.地址变换**

如指令LOAD 1,[D]|<A> 经过编译程序编译后，形成等价的机器指令：

“取出段号为2，段内地址为1024”的内存单元中的内容，放到寄存器1中

地址变换的步骤是：

（1）进程切换相关的内核程序负责恢复进程运行环境，包括从PCB中取出数据恢复段表寄存器，段表寄存器中包含了段表始址F，段表长度M。

（2）根据逻辑地址得到段号S、段内地址W

（3）判断段号是否越界：若S>=M，则产生越界中断，否则继续执行。段号是从0开始的。

（4）查询段表，找到对应的段表项。段表项的存放地址为F+S\*段表项长度

**（5）检查段内地址是否超过段长。若W>=C，则产生越界中断，否则继续执行。**（这一步是分页存储管理方式中没有的，因为页面大小是统一的，页内地址位数不能大于页面大小以2为底的对数，但段的大小是人为规定的，段内地址有可能越界）

（6）计算得到物理地址：用段基址b+段内地址W得到物理地址E

（7）访问目标内存单元E

**4.分段与分页管理的对比**

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

段是信息的逻辑单位。分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。

分段比分页更容易实现信息的共享和保护。如果多个进程有共享的代码，则共享的代码可以作为一个功能段，允许各个共享的进程访问，只需让各进程的段表项指向同一个段即可实现共享。只有不能被修改的代码（称为纯代码或可重入代码）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）。分页不容易实现这种信息的共享，因为页的大小固定且由系统决定，因此给进程分页可能会切断共享的代码，使共享的代码和私有的代码混在一起成为一页。总之，页面不是按逻辑模块划分的，这就很难实现共享。

在段表中加一项“是否允许其它进程访问”，即可实现进程间共享同一段

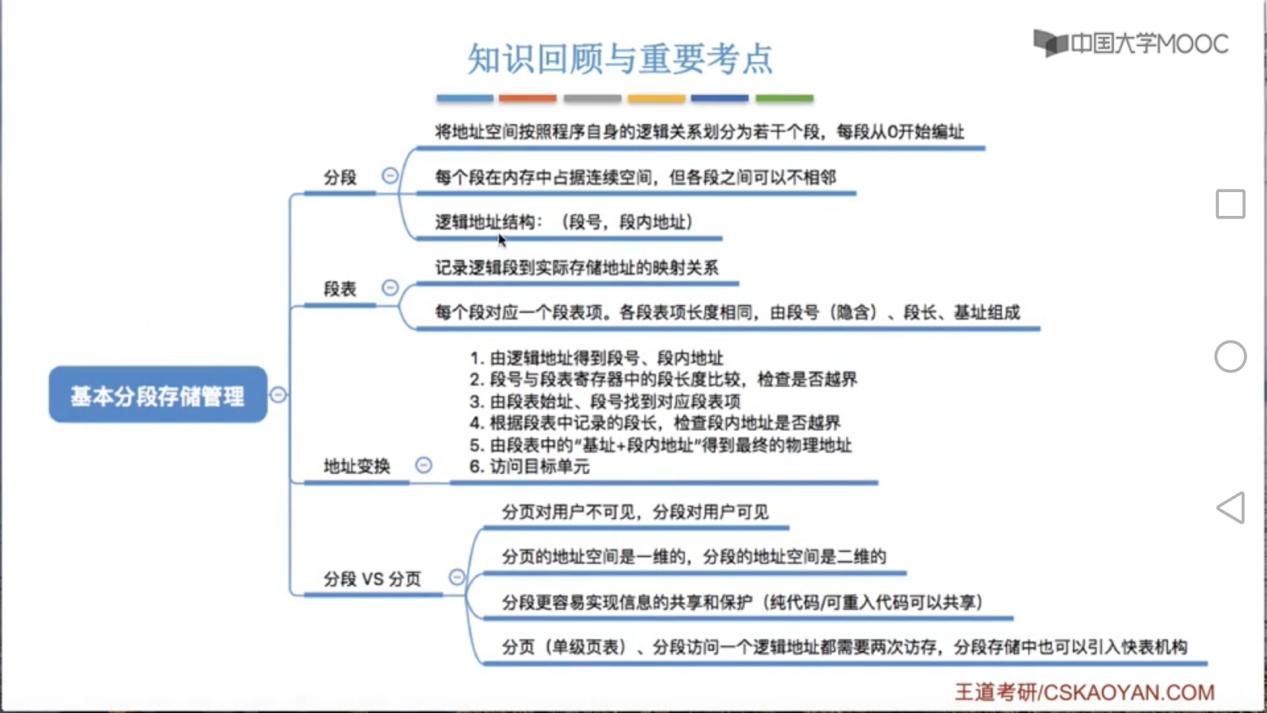
|  |  |  |
| --- | --- | --- |
| 段长 | 基址 | 是否允许其它进程访问 |
|  |  |  |

访问一个逻辑地址需要几次访存？

分页（单级页表）：2次

分段：第1次查内存中的段表，第2次访问目标内存单元。总共两次访存。

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。



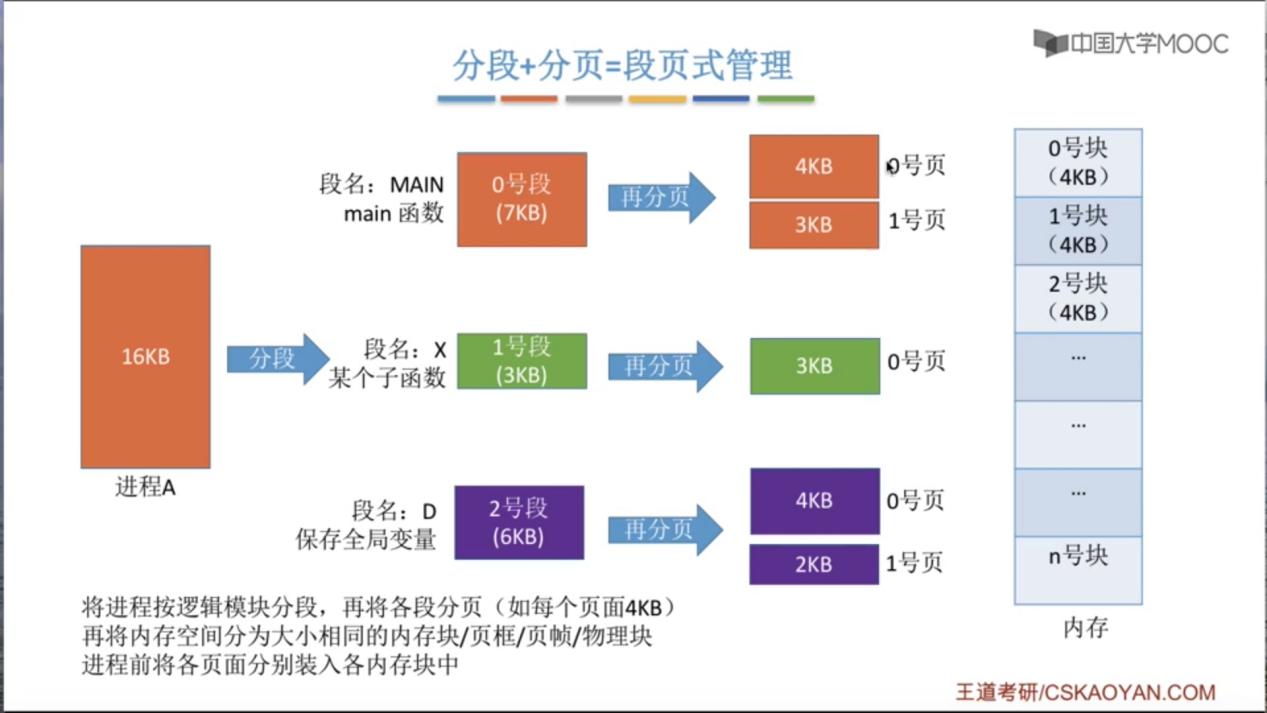
**3.1.11段页式管理方式**

|  |  |  |
| --- | --- | --- |
|  | 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。 |

1.段页式管理

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

进程运行前将各页面分别装入各内存块中



段页式管理的逻辑地址结构由段号、页号、页内偏移量组成：

|  |  |  |
| --- | --- | --- |
| 31..... 16 | 15...... 12 | 11...... 0 |
| 段号 | 页号 | 页内偏移量 |

段号的位数决定了进程最多可以分几个段

页号位数决定了每个段最多有多少页

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，每个进程最多有64K个段

页号占4位，每个段最多有16页。页内偏移量占12位，每个页面大小为4KB。

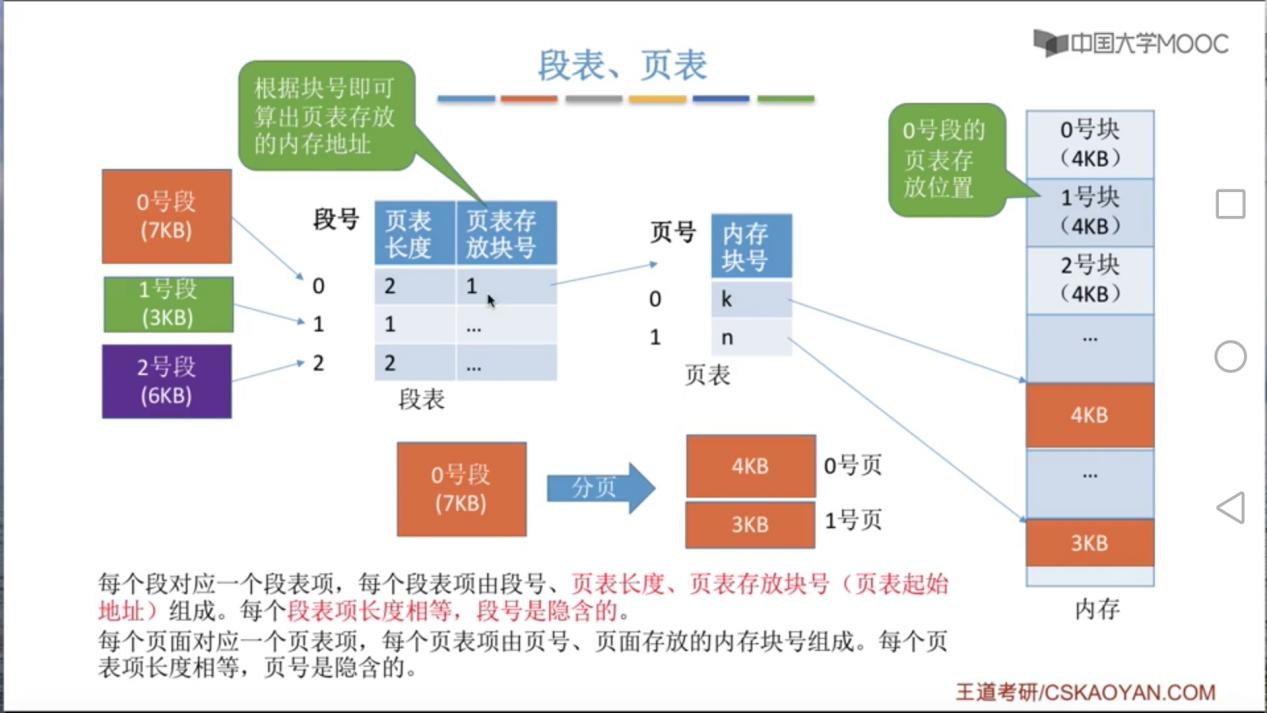
“分段”对用户是可见的，程序员编程时需要显式地给出段号、段内地址。而将各段“分页”对用户是不可见的。系统会根据段内地址自动划分页号和页内偏移量。因此段页式管理的地址结构是二维的。

2.段表、页表

每个段对应一个段表项，每个段表项由段号、页表长度、页表存放块号（页表起始地址）组成。每个段表项长度相同，段号是隐含的。根据块号即可算出页表存放的内存地址。

|  |  |  |
| --- | --- | --- |
| 段号 | 页表长度 | 页表存放块号 |
|  |  |  |

每个页面对应一个页表项，每个页表项由页号、页面存放的内存块号组成。每个页表项长度相等，页号是隐含的。



每个进程只对应一个段表。每个段对应一个页表。因此一个进程会对应多个页表。

**3.地址变换**

进程上CPU运行前，从PCB中读出段表始址F和段表长度M放入段表寄存器中。

（1）根据逻辑地址得到段号、页号、页内偏移量

（2）判断段号是否越界。若S>=M，则产生越界中断，否则继续执行

（3）查询段表，找到对应的段表项，段表项的存放地址为F+S\*段表项长度

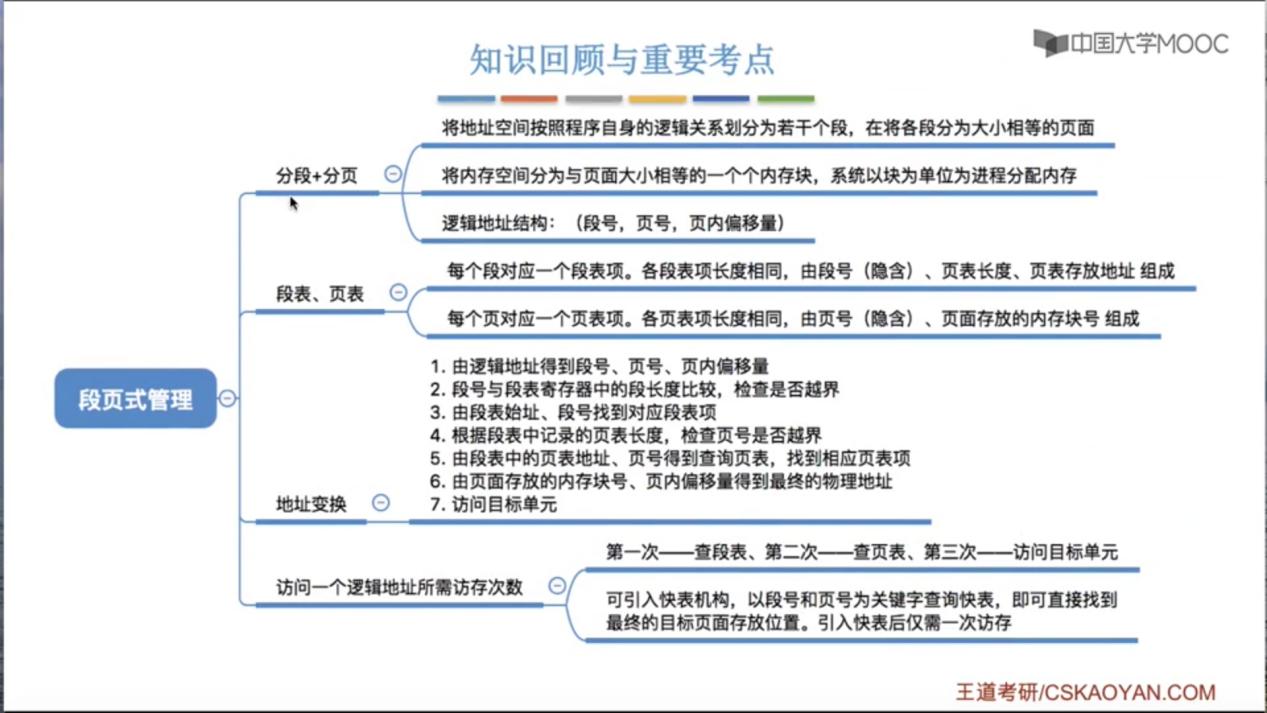
（4）检查页号是否越界，若页号>=页表长度，则发生越界中断，否则继续执行

（5）根据页表存放块号，页号查询页表，找到对应页表项

（6）根据内存块号、页内偏移量拼接得到最终物理地址E

（7）访问目标内存单元

共需要进行三次访存，第一次访问段表，第二次访问页表。第三次访问内存单元E。也可引入快表机制，用段号和页号作为查询快表的关键字。若快表命中，只需访存一次。



**3.2.1虚拟内存的基本概念**

传统存储管理方式有单一连续分配、固定分区分配、动态分区分配、基本分页存储管理、基本分段存储管理、基本段页式存储管理方式。很多暂时用不到的数据也会长期占用内存，导致内存利用率不高。

传统存储管理方式具有以下缺点：

（1）一次性：作业必须一次性全部装入内存后才能开始运行。这会造成两个问题：①作业很大时，不能全部装入内存，导致大作业无法运行。②当大量作业要求运行时，由于内存无法容纳所有作业，因此只有少量作业能运行，导致多道程序并发度下降

（2）驻留性：一旦作业被装入内存，就会一直驻留在内存中，直到作业运行结束。事实上，在一个时间段内，只需要访问作业的一小部分数据即可正常运行，这就导致了内存中会驻留大量的、暂时用不到的数据，浪费了宝贵的内存资源。

**局部性原理**

时间局部性：如果执行了程序中的某条指令，那么不久后这条指令很有可能再次执行。如果某个数据被访问过，不久之后该数据很可能再次被访问。（程序中有大量循环）

空间局部性：一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也很有可能被访问（因为很多数据在内存中是连续存放的，并且程序的指令也是顺序地在内存中存放的）

**1.虚拟内存**

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。**

虚拟内存是操作系统虚拟性的体现。

虚拟内存有以下三个主要特征：

多次性：无需在作业运行时一次性全部装入内存，而是允许被分成多次调入内存

对换性：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容量，使用户看到的内存容量远大于实际的容量

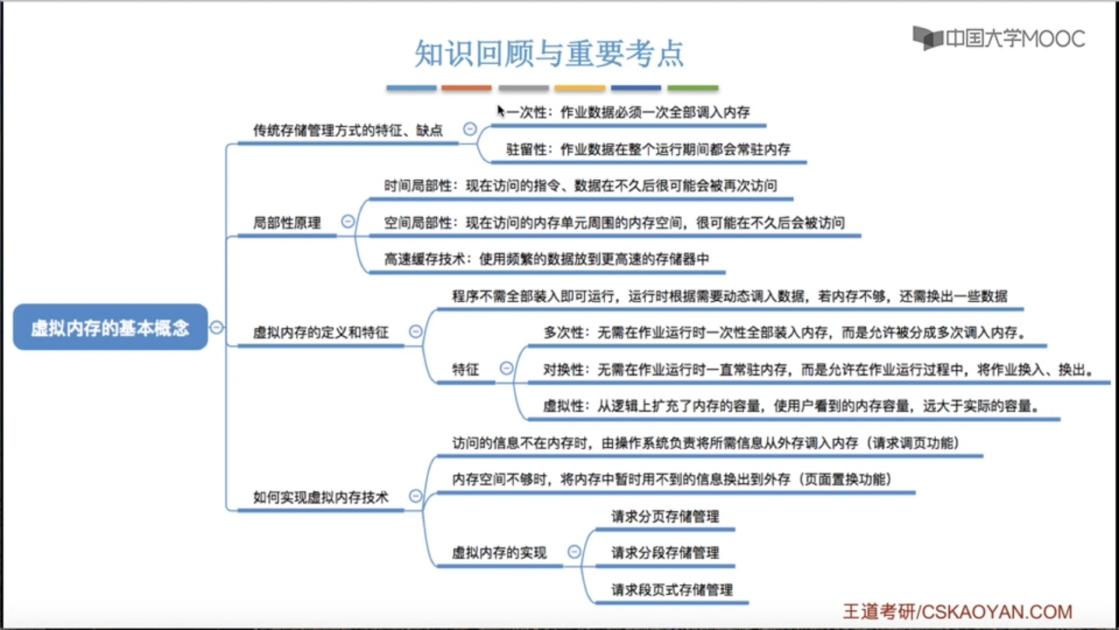
2.如何实现虚拟内存技术

虚拟内存技术允许一个作业多次调入内存。如果采用连续分配方式，会不方便实现。因此，虚拟内存的实现需要建立在离散分配的内存管理方式基础上。

对应于传统非连续分配存储管理的三种方式：基本分页、基本分段、基本段页式。虚拟内存的实现也有三种方式：请求分页存储管理、请求分段存储管理、请求段页式存储管理

主要区别在于：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。

为此，操作系统需再增加两个功能：1.请求调页（调段）功能 2.页面置换（段置换）功能



**3.2.2请求分页管理方式**

相比于基本分页管理，操作系统要提供请求调页功能和页面置换功能。

1.请求分页管理方式的页表机制

与基本分页管理相比，请求分页管理为了实现“请求调页”，操作系统需要知道每个页面是否已经调入内存，如果还没调入，也需要知道该页面在外存中存放的位置。当内存空间不够时，要实现“页面置换”，操作系统需要通过某些指标来决定到底换出哪个页面；有的页面没有被修改过，换出时就不用再浪费时间写回外存。有的页面修改过，就需要将外存中的旧数据覆盖，因此操作系统也需要记录各个页面是否被修改的信息。

为此，请求分页存储管理的页表增加四个字段：

（1）状态位：记录页面是否已调入内存

（2）访问字段：可记录最近被访问过几次，或记录上次访问的时间，供置换算法选择换出页面时参考

（3）修改位：页面调入内存后是否被修改过

（4）外存地址：页面在外存中存放位置

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 页号（隐含） | 内存块号 | 状态位 | 访问字段 | 修改位 | 外存地址 |
| 0 | 无 | 0 | 0 | 0 | X |
| 1 | b | 1 | 10 | 0 | Y |

2.缺页中断机构

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。

调入需要I/O操作系统，故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。

缺页中断是因为当前执行的指令想要访问的目标页面未调入内存产生的，因此属于内中断（故障）。

一条指令在执行期间，可能产生多次缺页中断。（如：copy A to B，即将逻辑地址A中的数据复制到逻辑地址B，而A、B属于不同的页面，则有可能产生两次中断）

3.地址变换机构

与基本分页存储管理方式相比，请求分页存储管理方式需要额外执行以下步骤：

（1）查找页号对应的页表项时，需要对页面是否在内存中进行判断

（2）地址变换过程中，若发现页面没有调入内存而内存中没有空闲块时，需要进行页面置换来换出某些页面，腾出内存空间

（3）页面被调入、调出或访问的时候需要修改请求页表中新增的表项

请求分页存储管理也需要检查页号合法性，查询快表中是否有对应页表项。如果快表命中，则直接得到最终物理地址。若快表没有命中，就要查询慢表。在慢表中找到对应页表项后，若对应页面未调入内存，则产生缺页中断，之后由操作系统的缺页中断处理程序进行处理。

快表中有的页面一定是在内存中的。若某页面被换出外存，则快表中相应表项也要被删除，否则可能访问错误的页面。

整个过程可以用如下的流程图表示：



其中的一些细节：

（1）只有“写指令”才需要修改“修改位”。并且一般来说只需修改快表中的数据，只有要将快表项删除时才需要写回内存中的慢表。这样可以减少访存次数。

（2）和普通中断处理一样，缺页中断处理依然需要保留CPU现场

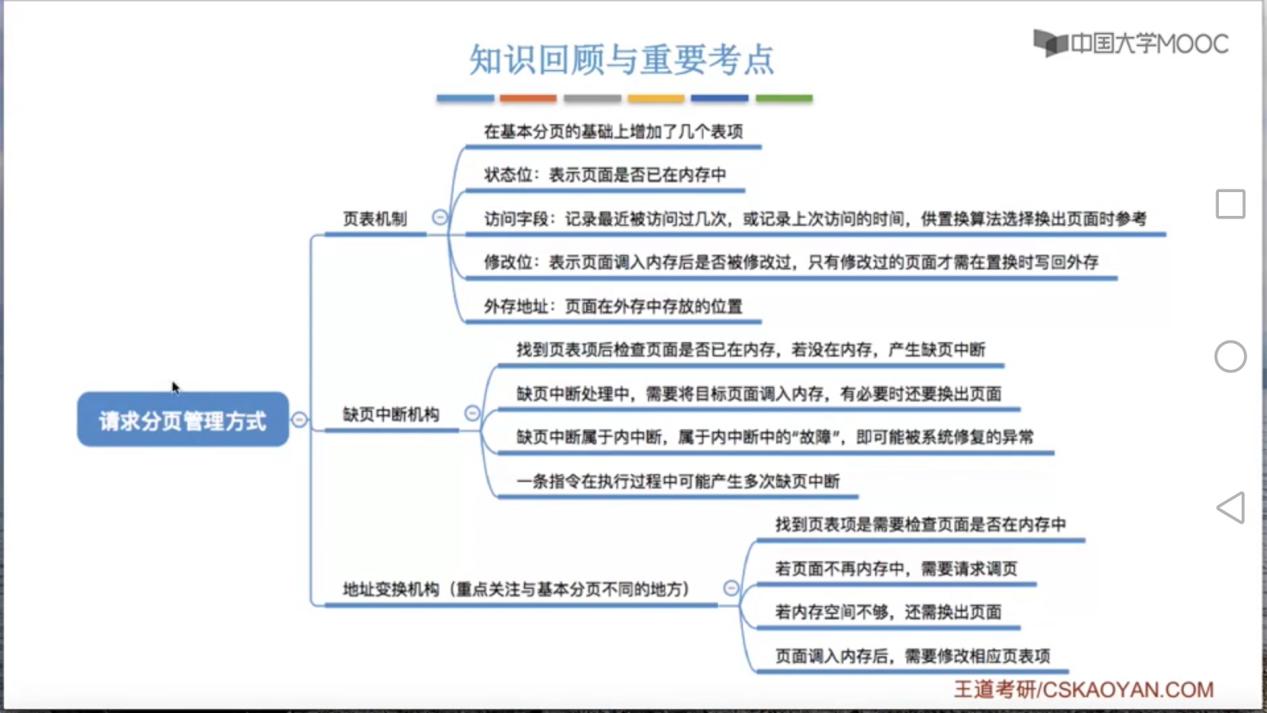
（3）需要用某种“页面置换算法”来决定哪个页面换出

（4）换入/换出页面都需要启动慢速的I/O操作，如果换入换出太频繁，会有很大的开销

（5）页面调入内存后，需要修改慢表，同时也需要将表项复制到快表中。

因此在具有快表机构的请求分页系统中，访问一个逻辑地址时，若发生缺页，则地址变换步骤是：

查快表（未命中）->查慢表（发现未调入内存）->调页（调入的页面对应的表项会直接加入快表）->查快表(命中，这一步一定会命中）->访问目标内存单元



**3.2.3页面置换算法**

页面的换入换出需要磁盘I/O，会有较大的开销，因此好的页面置换算法应该追求更少的缺页率

1.最佳置换算法

每次选择淘汰的页面将是以后永不使用，或者在最长时间内不再被访问的页面，这样可以保证最低的缺页率。

例：假设系统为某进程分配了三个内存块，并考虑到有以下页面号引用串：7，0，1，2，0，3，0，4，2，3，0，3，2，1，2，0，1，7，0，1。那么按最佳置换算法的规则，每当要发生页面置换时，就向后寻找，目前在内存中的三页在以后的访问序列中最后一个出现的页号，即是要淘汰的页面

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 |
| 块1 | 7 | 7 | 7 | 2 |  | 2 |  | 2 |  |  | 2 |  |  | 2 |  |
| 块2 |  | 0 | 0 | 0 |  | 0 |  | 4 |  |  | 0 |  |  | 0 |  |
| 块3 |  |  | 1 | 1 |  | 3 |  | 3 |  |  | 3 |  |  | 1 |  |
| 缺页 | √ | √ | √ | √ |  | √ |  | √ |  |  | √ |  |  | √ |  |
|  | 0 | 1 | 7 | 0 | 1 |  |  |  |  |  |  |  |  |  |  |
| 块1 |  |  | 7 |  |  |  |  |  |  |  |  |  |  |  |  |
| 块2 |  |  | 0 |  |  |  |  |  |  |  |  |  |  |  |  |
| 块3 |  |  | 1 |  |  |  |  |  |  |  |  |  |  |  |  |
| 缺页 |  |  | √ |  |  |  |  |  |  |  |  |  |  |  |  |

第一次访问2时，内存中已有0，1，7三个页面。由于7号页面在接下来的序列中出现得最晚，所以选择7号页面换出

整个过程缺页中断发生了9次，页面置换发生了6次。缺页时未必发生页面置换。若还有可用的空闲内存块，就不用进行页面置换。

缺页率=9/20=45%

最佳置换算法执行的前提是提前知道以后访问的序列，但实际上只有在进程执行的过程中才能知道接下来会访问到哪个页面。操作系统无法提前预判页面访问序列。因此，**最佳置换算法是无法实现的。**

2.先进先出置换算法（FIFO）

每次选择淘汰的页面是最早进入内存的页面

实现方法：把调入内存的页面根据调入的先后顺序排成一个队列，需要换出页面时选择队头页面即可。队列的最大长度取决于系统为进程分配了多少个内存块。

例：假设系统为某进程分配了三个内存块，并考虑到有以下页面引用串：

3，2，1，0，3，2，4，3，2，1，0，4

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3 | 2 | 1 | 0 | 3 | 2 | 4 | 3 | 2 | 1 | 0 | 4 |
| 块1 | 3 | 3 | 3 | 0 | 0 | 0 | 4 |  |  | 4 | 4 |  |
| 块2 |  | 2 | 2 | 2 | 3 | 3 | 3 |  |  | 1 | 1 |  |
| 块3 |  |  | 1 | 1 | 1 | 2 | 2 |  |  | 2 | 0 |  |
| 缺页 | √ | √ | √ | √ | √ | √ | √ |  |  | √ | √ |  |

第一次访问0时，内存中已有3，2，1三个页面。由于3号页面进入内存是最早的，所以选择3号页面换出

整个过程缺页中断发生了9次

如果是分配了四个内存块，那么相同的访问序列下，缺页次数是10次。也就是说，当为进程分配的物理块数增大时，缺页次数不减反增的异常现象，这种现象叫Belady异常。

只有FIFO算法会产生Belady异常。另外，FIFO算法虽然实现简单，但是该算法与进程实际运行时的规律不适应，因为先进入的页面也有可能最经常被访问。因此，算法性能差。

3.最近最久未使用置换算法（LRU）

每次淘汰的页面是最近最久未使用的页面

实现方法：赋予每个页面对应的页表项中，用访问字段记录该页面自上次被访问以来所经历的时间t。当需要淘汰一个页面时，选择现有页面中t值最大的，即最近最久未使用的页面

例：假设系统为某进程分配了四个内存块，并考虑到有以下页面号引用串：

1，8，1，7，8，2，7，2，1，8，3，8，2，1，3，1，7，1，3，7

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 8 | 1 | 7 | 8 | 2 | 7 | 2 | 1 | 8 |
| 块1 | 1 | 1 |  | 1 |  | 1 |  |  |  |  |
| 块2 |  | 8 |  | 8 |  | 8 |  |  |  |  |
| 块3 |  |  |  | 7 |  | 7 |  |  |  |  |
| 块4 |  |  |  |  |  | 2 |  |  |  |  |
| 缺页 | √ | √ |  | √ |  | √ |  |  |  |  |
|  | 3 | 8 | 2 | 1 | 3 | 1 | 7 | 1 | 3 | 7 |
| 块1 | 1 |  |  |  |  |  | 1 |  |  |  |
| 块2 | 8 |  |  |  |  |  | 7 |  |  |  |
| 块3 | 3 |  |  |  |  |  | 3 |  |  |  |
| 块4 | 2 |  |  |  |  |  | 2 |  |  |  |
| 缺页 | √ |  |  |  |  |  | √ |  |  |  |

第一次访问页面3时，内存中已有1，2，7，8四个页面，由于最近三个被访问的页面是8，1，2。所以选择淘汰7号页面。

该算法的实现需要专门的硬件支持，虽然算法性能好，但实现困难，开销大

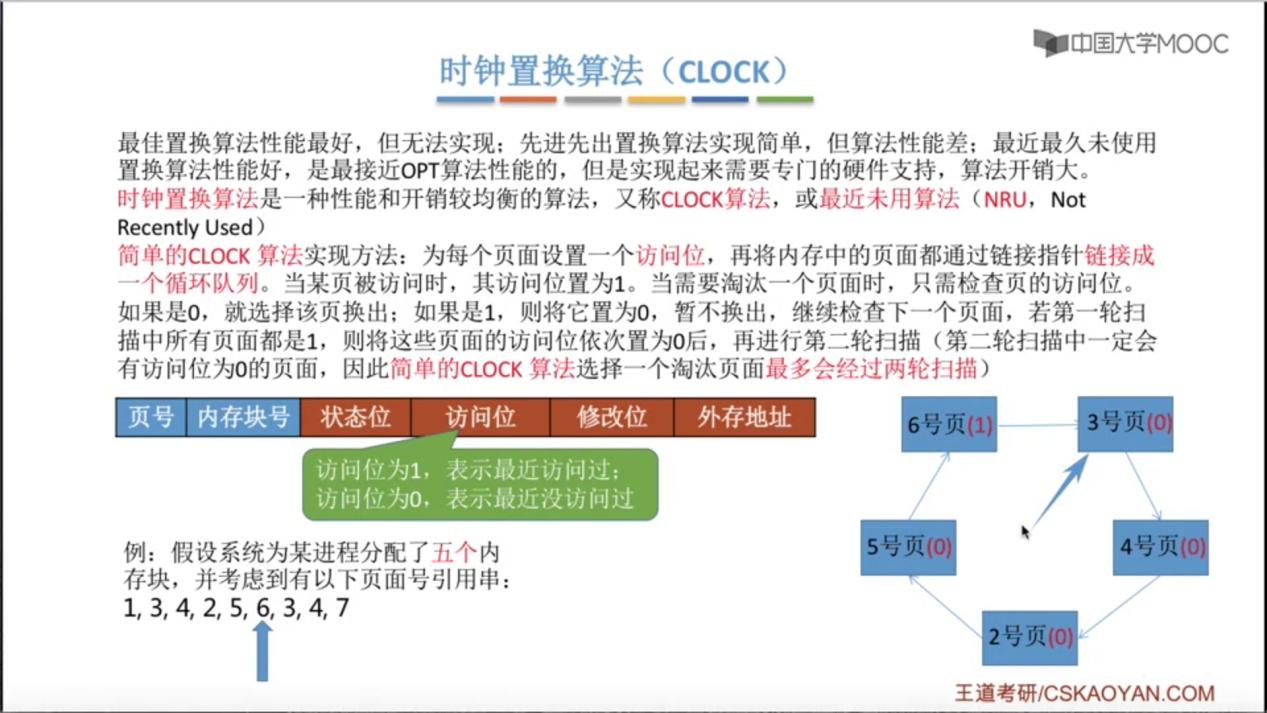
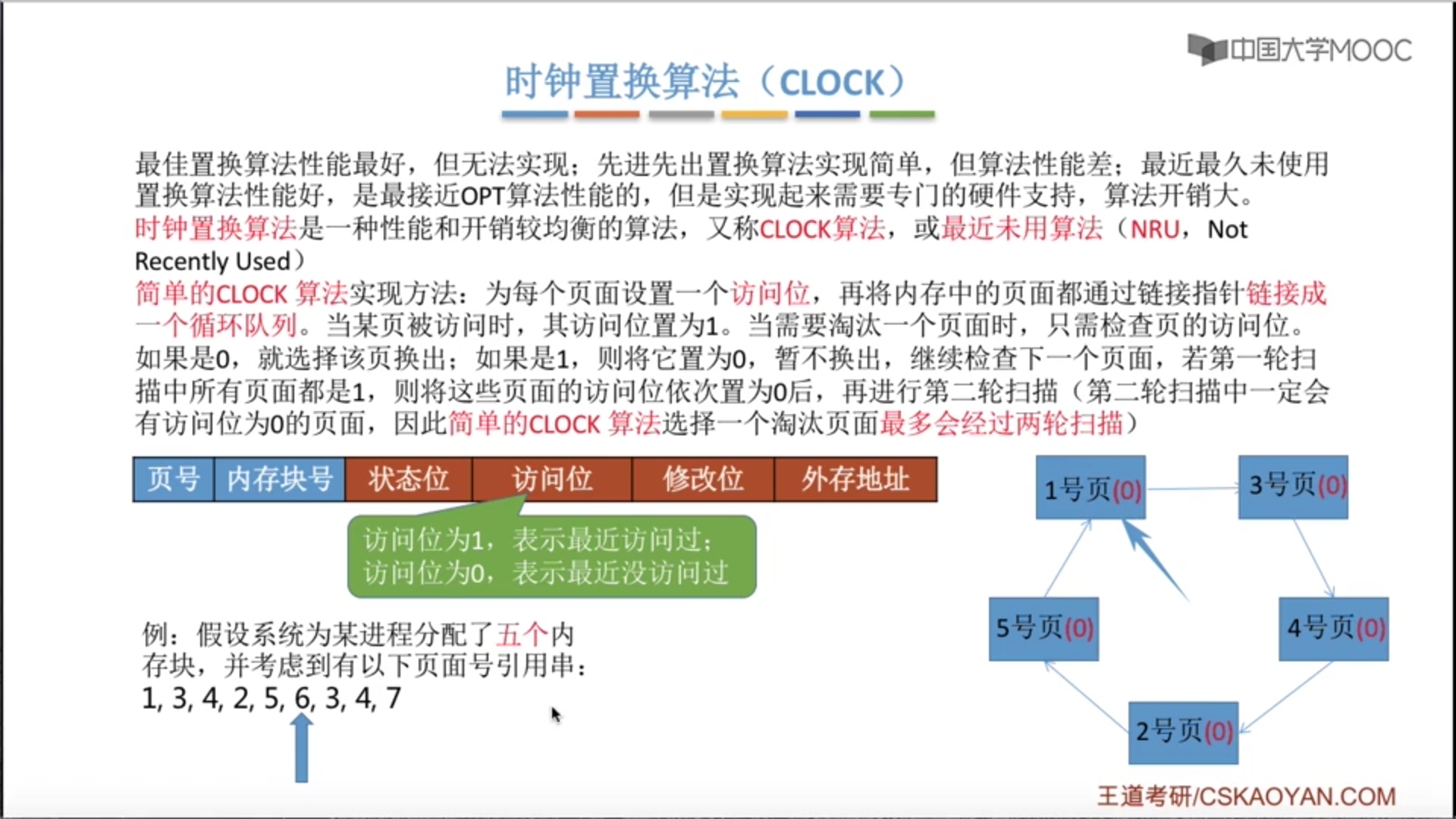
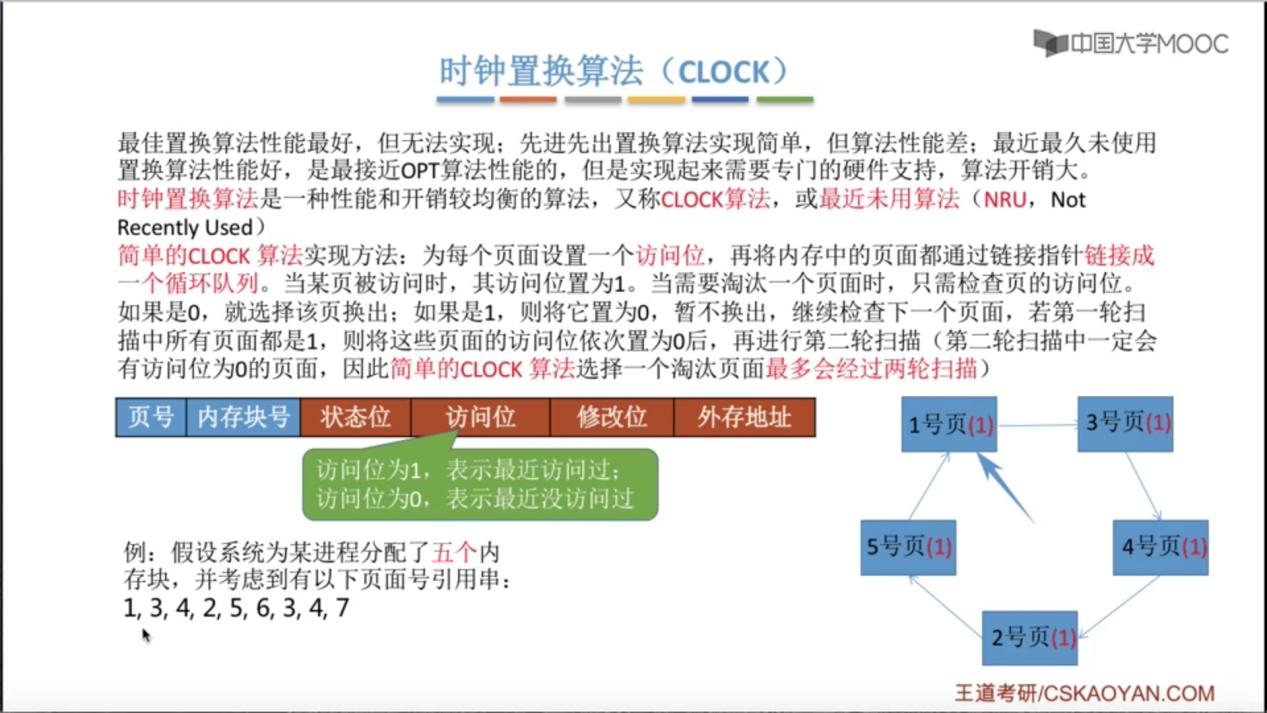
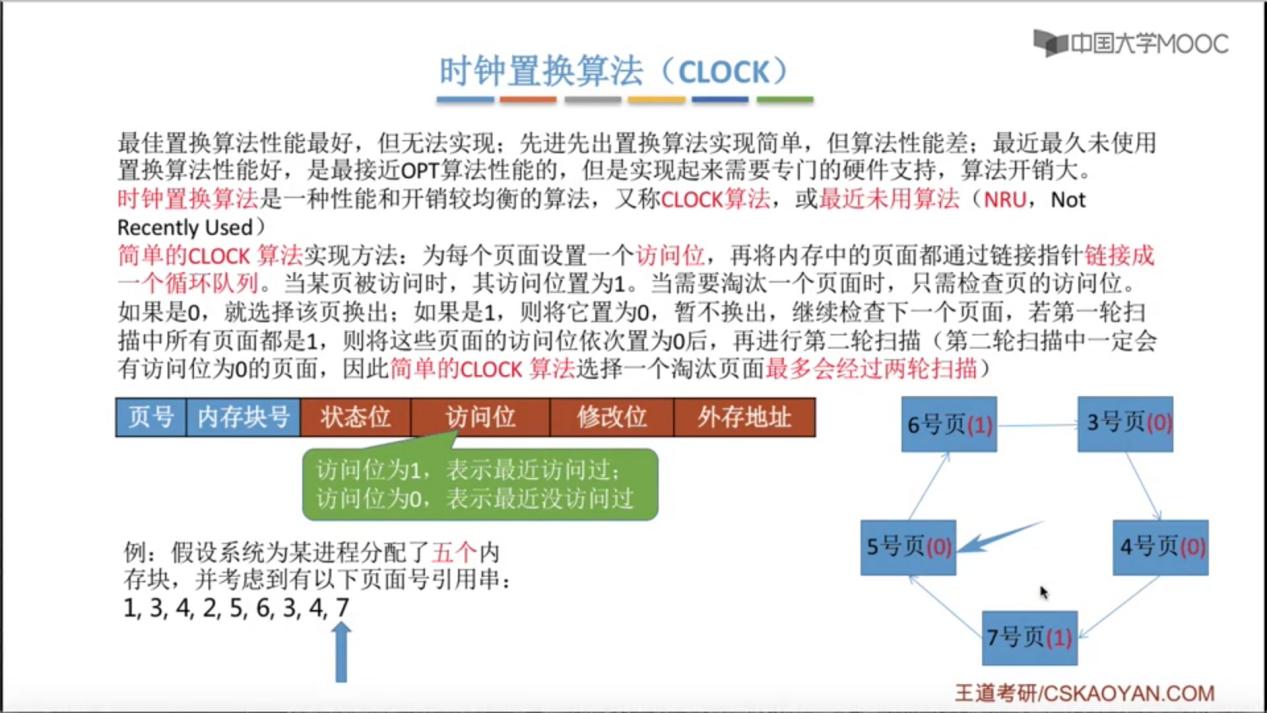
4.时钟置换算法(CLOCK),也叫最近未用算法(NRU)

(1)简单时钟算法

为每个页面设置一个访问位，再将内存中的页面都通过链接指针链接成一个循环队列。当某页被访问时，其访问位置为1。当需要淘汰一个页面时，只需检查页的访问位。如果是0，就选择该页换出，如果是1，则将它置为0，暂不换出，继续检查下一个页面。若第一轮扫描中所有页面者是1，则将这些页面的访问位依次置为0后，再进行第二轮扫描（第二轮扫描中一定会有访问位为0的页面，因此简单的CLOCK算法选择一个淘汰页面最多会经过两轮扫描）

例：假设系统为某进程分配了5个内存块，并考虑到有以下页面号引用串：

1，3，4，2，5，6，3，4，7



第一次访问页面6时，扫描整个由页面1，3，4，2，5组成的循环队列，因为访问都是1，所以一圈扫描后全置为0。然后指针停在页面1且页面1访问位为0。故换出页面1。

换入页面6后，指针要向后移一位指向页面3。

访问页面3，4后，页面3，4的标志位又置为1.

第一次访问页面7时，从上次指针停下的位置，也就是页面3开始扫描。将3，4的访问位都置0，然后选择换出页面2。

换入页面7后，指针要再向后移一位指向页面5。

扫描的过程很像时钟指针转动扫描，故算法叫做时钟置换算法

（2）改进时钟算法

除了考虑一个页面最近有没有被访问过之外，操作系统还应考虑页面有没有被修改过。在其他条件都相同时，应优先淘汰没有修改过的页面，避免I/O操作。这就是改进弄的时钟置换算法的思想

为此，需要使用到页表中的修改位。修改位=0表示页面没有被修改过，修改位=1表示页面有被修改过。在此，用（访问位，修改位）的形式表示各页面状态，即(1,1)表示一个页面近期被访问过，且被修改过。

算法规则：将所有可能被置换的页面排成一个循环队列

第一轮：从当前位置开始扫描到第一个(0,0)的帧用于替换。本轮扫描不修改任何标志位

第二轮：若第一轮扫描失败，则重新扫描，查找第一个(0,1)的帧用于替换。本轮将所有扫描过的帧访问位设为0

第三轮：若第二轮扫描失败，则重新扫描，查找第一个(0,0)的帧用于替换。本轮扫描不修改任何标志位

第四轮：若第三轮扫描失败，则重新扫描，查找第一个(0,1)的帧用于替换。

由于第二轮已将所有帧的访问位置为0，因此经第三轮、第四轮扫描一定会有一个帧被选中，因此改进型CLOCK置换算法选择一个淘汰页面最多会进行四轮扫描

**3.2.4页面分配策略、抖动、工作集**

驻留集：指请求分页存储管理中给进程分配的物理块的集合

在采用了虚拟存储技术的系统中，驻留集大小一般小于进程的总大小

若驻留集太小，会导致缺页频繁，系统要花大量时间来处理缺页，实际用于进程推进的时间很少

驻留集太大，又会导致多道程序并发度下降，资源利用率低。所以应该选择一个合适的驻留集大小。

对于驻留集大小是否可变，又有两种分配策略：

（1）固定分配：操作系统为每个进程分配一组固定数目的物理块，在进程运行期间不再改变，即驻留集大小不变

（2）可变分配：先为每个进程分配一定数目的物理块，在进程运行期间，可根据情况做适当的增加或减少，即驻留集大小可变

对于页面置换时，被置换页面的范围是什么，又有两种置换策略：

（1）局部置换：发生缺页时只能选进程自己的物理块进行置换

（2）全局置换：可以将操作系统保留的空闲物理块分配给缺页进程，也可以将别的进程持有的物理块置换到外存，再分配给缺页进程

这两种分配策略和置换策略两两组合，总共有三种分配置换策略：

|  |  |  |
| --- | --- | --- |
|  | 局部置换 | 全局置换 |
| 固定分配 | √ | × |
| 可变分配 | √ | √ |

固定分配-全局置换是不存在的，因为采用全局置换的置换策略意味着进程拥有的物理块数必然会改变，因此不可能是固定分配

1.固定分配局部置换：系统为每个进程分配一定数量的物理块，在整个运行期间都不改变。若进程在运行中发生缺页，则只能从该进程在内存中的页面中选出一页换出，然后再调入需要的页面。

这种策略的缺点是：很难在刚开始时就确定应为每个进程分配多少个物理块才合理。

2.可变分配全局置换：刚开始会为每个进程分配一定数量的物理块。操作系统会保持一个空闲物理块队列。当某进程发生缺页时，从空闲物理块中取出一块分配给该进程；若已无空闲物理块，则可选择一个未锁定的页面换出外存，再将该物理块分配给缺页的进程。采用这种策略时，只要某进程发生缺页，都将获得新的物理块，仅当空闲物理块用完时，系统才选择一个未锁定的页面调出。被选择调出的页可能是系统中任何一个进程中的页，因此这个被选中的进程拥有的物理块会减少，缺页率会增加。

系统会锁定一些重要的页面，这些页面中的内容不能置换出外存。

3.可变分配局部置换：刚开始会为每个进程分配一定数量的物理块。当某进程发生缺页时，只允许从该进程自己的物理块中选出一个进行换出外存。如果进程在运行中频繁地缺页，系统会为该进程多分配几个物理块，直到该进程缺页率趋势适当适度；反之，如果进程在运行中缺页率特别低，则可适当减少分配给该进程的物理块

可变分配全局置换和可变分配局部置换的区别在于：

可变分配全局置换是只要缺页就给分配新物理块

可变分配局部置换要根据发生缺页的频率来动态地增加或减少进程的物理块

对于何时调入页面，有两种策略：

（1）预调页策略：根据局部性原理，一次调入若干个相邻的页面可能比一次调入一个页面更高效。但如果提前调入的页面大多数都没被访问过，则又是低效的。因此，可以预测不久之后可能访问到的页面，将它们预先调入内存，但目前预测成功率只有50%左右。故这种策略主要用于进程的首次调入，由程序员指出应该先调入哪些部分。

（2）请求调页策略：进程在运行期间发现缺页时才将所缺页面调入内存。由于这种策略调入的页面一定会被访问到，但由于每次只能调入一页，而每次调页都要磁盘I/O操作，因此I/O开销较大。

系统拥有足够的对换区空间：页面的调入、调出都是在内存与对换区之间进行，这样可以保证页面的调入、调出速度很快。在进程运行前，需将进程相关的数据从文件区复制到对换区。对换区的读/写速度更快，采用连续分配方式。