**操作系统知识点总结**

**1. 基础知识**

**1.1 内核态和用户态**

多数计算机有两种运行模式，用户态和内核态。软件中最基础的部分是操作系统，它运行在内核态。在这个模式中，**操作系统具有对所有硬件的完全访问权，可以执行机器能够运行的任何指令。**软件的其余部分运行在用户态，在用户态下，只使用了机器指令中的一个子集。

核心态和用户态各有优势：运行在核心态的程序可以访问更多资源，但可靠性、安全性要求高，维护管理都较复杂；用户态程序访问资源受限，但可靠性、安全性要求低，编写维护较简单。

为了从操作系统中获得服务，用户程序必须使用**系统调用**陷入内核并调用操作系统。TRAP指令把用户态切换成内核态，并启用操作系统。当有关工作完成之后，在系统调用后面的指令把控制权返回给用户程序。

**1.2 什么是操作系统**

操作系统是一中运行在内核态的软件，执行以下两个基本任务：**为应用程序提供一个资源集的清晰抽象，并管理这些硬件资源。**

**1.3 计算机硬件**

计算机硬件包括：**CPU、内存以及IO设备。**

CPU负责从内存中取出指令并执行。由于用来访问内存以及得到指令或数据的时间要比执行指令花费的时间长得多，因此所有的CPU内都有一些用来保存关键变量和临时数据的寄存器。除此之外，计算机还有一些对程序员可见的专用寄存器：

* 程序计数器：用于保存下一条指令的内存地址。
* 堆栈指针：指向内存中当前栈的顶端。
* PSW程序状态字寄存器：包含了条件码位，CPU优先级，模式（用户态或内核态），以及各种控制位。

**2. 进程与线程**

**2.1 进程与线程的区别**

**进程：进程是一个正在执行程序的示例，拥有自己的程序计数器和内部状态，是系统进行资源分配和调度的一个独立单位**（具有动态、并发、独立、异步的特性，**以及就绪、执行、阻塞3种状态**，资源拥有单位等属性）；引入进程是为了使多个程序可以并发的执行，以提高系统的资源利用率和吞吐量。

线程：**是比进程更小的可独立运行的基本单位，**可以看做是轻量级的进程（具有轻型实体，独立调度分派单位，可并发执行，共享进程资源等属性）；引入目的是为了减少程序在并发执行过程中的开销，使OS的并发效率更高。

两者的对比：

1. 调度方面：**在引入线程的OS中，线程是独立的调度和分派单位，而进程作为资源的拥有单位**(相当于把未引入线程的传统OS中的进程的两个属性分开了)。由于线程不拥有资源，因此可以显著的提高并发度以及减少切换开销。
2. 并发性：**引入了线程的OS中，进程间可以并发，而且一个进程内部的多个线程之间也是可以并发的，**这就使OS具有更好的并发性，有效的提高了系统资源利用率和吞吐量。
3. 拥有资源：处于安全和方便管理的因素，一个进程往往会独占一些资源，如地址空间、全局变量、打开的文件、子进程、信号和账户信息等；而为了处理各自的任务，**线程也会独占一些资源，如栈、寄存器、程序计数器和状态等。**
4. 系统开销：**创建或者撤销进程的时候，系统要为之创建或回收PCB，系统资源等，切换时也需要保存和恢复CPU环境。**而线程的切换只需要保存和恢复少量的寄存器，不涉及存储器管理方面的工作，所以开销较小。此外，**统一进程中的多个线程由于共享地址空间，所以通信同步等都比较方便。**

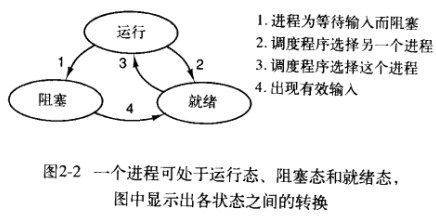
**例题：同一个进程中的线程不共享的部分是（F）**。（2017阿里巴巴实习生笔试题）

A、信号 B、堆 C、文件描述符 D、进程组id E、代码段 F、栈空间

**2.2 进程的三种状态**

1. **就绪状态：**进程获得了除了CPU之外的所有的必要资源，只要获得CPU就可以立即执行，此时的进程处于就绪态。
2. **运行状态：**进程已经获得CPU，正在运行，在多处理其系统中，会有多个进程同时处于运行状态。
3. **阻塞状态：**处于执行状态的进程由于发生某些事件而暂时无法继续执行，放弃处理机而处于暂停状态，此时进程就处于阻塞（执行受到阻塞）状态。

进程的三种状态之间有4种可能的转换关系：



**2.3 线程的实现方式**

**在用户空间中实现线程**  
**用户级线程指不需要内核支持而在用户程序中实现的线程，其不依赖于操作系统核心，**应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。**不需要用户态/核心态切换，**速度快，操作系统内核不知道多线程的存在，**因此一个线程阻塞将使得整个进程（包括它的所有线程）阻塞。**由于这里的处理器时间片分配是以进程为基本单位，所以每个线程执行的时间相对减少。

**在内核中实现线程**  
内核级线程由操作系统内核创建和撤销。内核维护进程及线程的上下文信息以及线程切换。**一个内核线程由于I/O操作而阻塞，不会影响其它线程的运行。**

**在用户空间中实现线程的优势**

* 可以在**不支持线程的操作系统**中实现。
* 创建和销毁线程、线程切换代价等线程管理的代价比内核线程少得多。
* 允许每**个进程定制自己的调度算法，线程管理比较灵活。**
* 线程能够利用的表空间和堆栈空间比内核级线程多。

**在用户空间中实现线程的缺点**

* 同一进程中只能同时有一个线程在运行，如果有一个线程使用了系统调用而阻塞，那么整个进程都会被挂起。
* 页面失效也会导致整个进程都会被挂起。

内核线程的优缺点刚好跟用户线程相反。实际上，**操作系统可以使用混合的方式来实现线程。**

**2.4 线程同步**

**竞争条件**：**两个或多个线程读写某些共享数据，而最后的结果取决于线程运行的精确时序。**为避免竞争条件，需要找到某种途径组织多个线程同时读写共享的数据。这里，我们把**对共享数据（共享内存）进行访问的程序片段称之为临界区，**只要我们能够使两个线程不可能同时处于临界区，就能够避免竞争条件。

同步机制需要遵循的原则：

1. 空闲让进：当没有线程处于临界区的时候，应该许可其他线程进入临界区的申请。
2. 忙则等待：当前如果有线程处于临界区，如果有其他线程申请进入，则必须等待，保证对临界区的互斥访问。
3. 有限等待：对要求访问临界资源的线程，需要在有限时间内进入临界区，防止出现死等。
4. 让权等待：当线程无法进入临界区的时候，需要释放处理机，边陷入忙等。

线程同步的常用方法：**互斥锁，条件变量，信号量。**

**互斥锁**：同一时刻只允许一个线程进入临界区。互斥锁有两个基本操作，**加锁和解锁。**一个线程如果想要进入临界区，它首先需要尝试锁住相关的互斥量。如果互斥量没有加锁，那么这个线程可以立即进入，并对互斥量进行加锁以防止其他线程进入。如果互斥量已经被加锁，则调用线程被阻塞，直至该互斥量被解锁。

**条件变量**：允许线程由于一些未达到的条件而阻塞，通常与互斥锁配合使用。条件变量的基本操作有：**触发条件(当条件变为 true 时)；等待条件，挂起线程直到其他线程触发条件。**

* 在等待进程中，需要等待该条件，即需要\_cond.wait()；wait()过程将会把调用线程放到等待条件的线程列表上，然后对该互斥量解锁；此时在互斥量解锁期间，又有新的线程进入该临界区，条件尚未发生，wait()会继续这一过程。
* 在唤醒进程中，首先会进行条件检查（已经被同一个互斥量锁住，睡眠的线程不可能错过）；如果条件成立，则唤醒等待进程。
* 需要使用while (\_count > 0)，而不是if (\_count > 0)，原因为当线程从\_cond.wait()唤醒时，此时互斥量会继续被锁住（此时多个线程对互斥量争用的问题），很有可能此时的条件会被其他线程修改，造成\_count > 0的条件不成立，因此需要继续判断的。

**信号量**：为控制一个具有有限数量的用户资源而设计。它**允许多个线程在同一个时刻去访问同一个资源，但一般需要限制同一时刻访问此资源的最大线程数目。**

**Pthread中的函数调用**

创建互斥锁：int pthread\_mutex\_init(pthread\_mutex\_t \*mutex,const pthread\_mutexattr\_t \*restrict attr)

加锁：int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex)

解锁：int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex)

销毁互斥锁：pthread\_mutex\_destroy ()

初始化条件变量：pthread\_cond\_init(pthread\_cond\_t \*cond,pthread\_condattr\_t \*cond\_attr)

无条件等待：pthread\_cond\_wait(pthread\_cond\_t \*cond,pthread\_mutex\_t \*mutex)

计时等待：pthread\_cond\_timewait(pthread\_cond\_t \*cond,pthread\_mutex \*mutex,const timespec \*abstime)

激活一个等待该条件的线程：pthread\_cond\_signal(pthread\_cond\_t \*cond)

激活所有等待线程：pthread\_cond\_broadcast(pthread\_cond\_t \*cond)

销毁条件变量：pthread\_cond\_destroy(pthread\_cond\_t \*cond)

经典的进程同步问题：**生产者-消费者问题**；哲学家进餐问题；**读者-写者问题**。

**2.5 进程间通信**

进程间通信的方式有**管道、消息队列、共享内存、信号、信号量和套接字**等。

1. **管道**管道是连接两个**一个读进程和一个写进程**之间用于实现数据交换的一个共享文件。为了协调管道通信双方，需要管道机制实现如下功能：1）互斥：**同一时刻只能有一个进程对管道进行读写；（半双工）**2）同步：当**读端发现管道为空的时候需要睡眠等待，直到有数据时候被唤醒（空了没得读）**，相应的**写端也是在管道已满的时候等待直到被唤醒（满了写不进去）**；3）确定对方的存在性：**只有同时有读端和写端，管道才有存在意义**。它包括**无名管道和有名管道**两种，**前者用于父进程和子进程间的通信，后者用于运行于同一台机器上的任意两个进程间的通信。**
2. **信号量**是一个计数器，可以用来控制多个进程对共享资源的访问。它**常作为一种锁机制，**防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
3. **消息队列**是由**消息存放在内核中并由消息队列标识符标识。**消息队列克服了**信号传递信息少、**管道**只能承载无格式字节流以及缓冲区大小受限**等缺点。其基本思想是根据**”生产者-消费者”原理**，利用内存中**公用消息缓冲区实现进程之间的信息交换**。每当一个进程向另一个进程发送消息时，便**申请一个消息缓冲区**，并把已准备好的消息送到缓冲区，然后把该消息缓冲区**插入到接收进程的消息队列中**，最后通知接收进程。接收进程收到发送里程发来的通知后，**从本进程的消息队列中摘下一消息缓冲区，取出所需的信息，然后把消息缓冲区不定期给系统。系统负责管理公用消息缓冲区以及消息的传递。**
4. **信号**是一种**比较复杂的通信方式，用于通知接收进程某个事件已经发生。**
5. **共享内存**就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。**共享内存是最快的IPC方式**，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，**如信号量，配合使用，**来实现进程间的同步和通信。
6. **套接字**也是一种进程间通信机制，可以实现**不同主机间的进程通信。**一个套接口可以看做是进程间通信的端点，每个套接口的名字是唯一的，其他进程可以访问，连接和进行数据通信。

**2.6 进程调度算法**

**基本调度算法**：

1. **先来先服务FCFS**：既可以作为作业调度算法也可以作为进程调度算法；按作业或者**进程到达的先后顺序依次调度**；因此对于长作业比较有利。
2. **短作业优先SPF**：作业调度算法，**算法从就绪队列中选择估计时间最短的作业进行处理，**直到得出结果或者无法继续执行；缺点：不利于长作业；未考虑作业的重要性；运行时间是预估的，并不靠谱。
3. \*\* 高优先权优先HRRF\*\*：既可以作为作业调度也可以作为进程调度算法；调度作业时，从**就绪队列中选择优先级最高的作业进行处理**；由于涉及到了优先级，因此可以分为抢占式和非抢占式；而且优先级的确定也可以分为静态优先级（事先根据进程类型，进程对资源的需求，用户要求等方面确定一个固定值）；动态优先级（随进程的推进或者等待时间而增加或者减少）。
4. **最高响应比优先HRN**：FCFS可能造成短作业用户不满，SPF可能使得长作业用户不满，于是提出HRN，**选择响应比最高的作业运行。（考虑等待和处理时间因素）**响应比=1+作业等待时间/作业处理时间。
5. 时间片轮转：按到达的先后对进程放入队列中，然后给队首进程分配CPU时间片，时间片用完之后计时器发出中断，暂停当前进程并将其放到队列尾部，循环。
6. **多级反馈队列：**目前公认较好的调度算法；设置多个就绪队列并为每个队列设置不同的优先级，第一个队列优先级最高，其余依次递减**。优先级越高的队列分配的时间片越短，进程到达之后按FCFS放入第一个队列，**如果调度执行后没有完成，那么放到第二个队列尾部等待调度，如果第二次调度仍然没有完成，放入第三队列尾部…。只有当前一个队列为空的时候才会去调度下一个队列的进程。

**3. 死锁**

死锁是指多个进程在运行过程中，因为争夺资源而造成的一种僵局，如果没有外力推进，处于僵局中的进程就无法继续执行。

**3.1 死锁原因：**

1. **竞争资源**：请求同一有限资源的进程数多于可用资源数
2. **进程推进顺序非法**：进程执行中，请求和释放资源顺序不合理，如资源等待链

**3.2 死锁产生的必要条件：**

1. **互斥条件**:进程对所分配的资源进行排他性的使用
2. **请求和保持条件**：进程被阻塞的时候并不释放锁申请到的资源
3. **不可剥夺条件**：进程对于已经申请到的资源在使用完成之前不可以被剥夺
4. **环路等待条件**：发生死锁的时候存在的一个 进程-资源 环形等待链

**3.3 死锁处理：**

1. **预防死锁**：破坏产生死锁的4个必要条件中的一个或者多个；实现起来比较简单，但是如果限制过于严格会降低系统资源利用率以及吞吐量
2. **避免死锁**：在资源的动态分配中，防止系统进入不安全状态(可能产生死锁的状态)-如银行家算法
3. **检测死锁**：允许系统运行过程中产生死锁，在死锁发生之后，采用一定的算法进行检测，并确定与死锁相关的资源和进程，采取相关方法清除检测到的死锁。实现难度大
4. **解除死锁**：与死锁检测配合，将系统从死锁中解脱出来（撤销进程或者剥夺资源）。对检测到的和死锁相关的进程以及资源，通过撤销或者挂起的方式，释放一些资源并将其分配给处于阻塞状态的进程，使其转变为就绪态。实现难度大

**例题：列举一种死锁发生的场景，并给出解决方案。**（PPS2013校园招聘笔试题）  
答：**最经典的场景就是生产者/消费者，生产者线程生产物品，**然后将物品放置在一个空缓冲区中供消费者线程消费。消费者线程从缓冲区中获得物品，然后释放缓冲区。由于生产者/消费者都在操作缓冲区，容易导致死锁的发生。可以通过添加锁的保护来对缓冲区进行互斥的访问，保证某一时刻只有一个线程对缓冲区进行操作，当缓冲区满的时候，生产者线程就会挂起，同时通知消费者线程。而缓冲区空的时候，消费者线程就会挂起，同时通知生产者线程。

**4. 存储管理**

**4.1 内存管理方式**

由于**连续内存分配方式**(单一连续分配，固定分区分配，动态分区分配，动态重定位分区分配)导致的内存利用率偏低以及内存碎片的问题，进而引出**离散的内存分配方式**。离散内存分配可以从OS的**内存管理角度引出页式**(离散分配的基本单位是页)管理，也可以从程序编制角度引出段式(离散分配的基本单位是段)管理。

**基本分页存储管理**

基本分页存储管理中不具备页面置换功能(即没有实现虚拟内存的功能)，因此需要整个程序的所有页面都装入内存之后才可以运行。因为程序数据存储在不同的页面中，而页面又离散的分布在内存中，因此需要一个页表来记录逻辑地址和实际存储地址之间的映射关系，以实现从页号到物理块号的映射。由于页表也是存储在内存中的，因此和不适用分页管理的存储方式相比，访问分页系统中内存数据需要两次的内存访问(一次是从内存中访问页表，从中找到指定的物理块号，加上页内偏移得到实际物理地址；第二次就是根据第一次得到的物理地址访问内存取出数据)。

为了减少两次访问内存导致的效率影响，分页管理中引入了快表(或者联想寄存器)机制，包含快表机制的内存管理中，当要访问内存数据的时候，首先将页号在快表中查询，如果查找到说明要访问的页表项在快表中，那么直接从快表中读取相应的物理块号；如果没有找到，那么访问内存中的页表，从页表中得到物理地址，同时将页表中的该映射表项添加到快表中(可能存在快表换出算法)。

**基本分段存储管理方式**

分页是为了提高内存利用率，而分段是为了满足程序员在编写代码的时候的一些逻辑需求(比如数据共享，数据保护，动态链接等)。

分段内存管理当中，地址是二维的，一维是段号，一维是段内地址；其中每个段的长度是不一样的，而且每个段内部都是从0开始编址的。由于分段管理中，每个段内部是连续内存分配，但是段和段之间是离散分配的，因此也存在一个逻辑地址到物理地址的映射关系，相应的就是段表机制。段表中的每一个表项记录了该段在内存中的起始地址和该段的长度。段表可以放在内存中也可以放在寄存器中。

访问内存的时候根据段号和段表项的长度计算当前访问段在段表中的位置，然后访问段表，得到该段的物理地址，根据该物理地址以及段内偏移量就可以得到需要访问的内存。由于也是两次内存访问，所以分段管理中同样引入了联想寄存器。

**分段和分页的对比**：

1. 页是信息的物理单位，是出于系统内存利用率的角度提出的离散分配机制；段是信息的逻辑单位，每个段含有一组意义完整的信息，是出于用户角度提出的内存管理机制。
2. 页的大小是固定的，由系统决定；段的大小是不确定的，由用户决定。
3. 页地址空间是一维的，段地址空间是二维的。

**段页式存储管理**

先将用户程序分为若干个段，然后再把每个段分成若干个页，并且为每一个段赋予一个段名称。这样在段页式管理中，一个内存地址就由段号，段内页号以及页内地址三个部分组成。

段页式内存访问：系统中设置了一个段表寄存器，存放段表的起始地址和段表的长度。地址变换时，根据给定的段号（还需要将段号和寄存器中的段表长度进行比较防止越界）以及寄存器中的段表起始地址，就可以得到该段对应的段表项，从段表项中得到该段对应的页表的起始地址，然后利用逻辑地址中的段内页号从页表中找到页表项，从该页表项中的物理块地址以及逻辑地址中的页内地址拼接出物理地址，最后用这个物理地址访问得到所需数据。由于**访问一个数据需要三次内存访问**，所以段页式管理中也引入了高速缓冲寄存器。

**4.2 虚拟内存**

如果存在一个程序，所需内存空间超过了计算机可以提供的实际内存，那么由于该程序无法装入内存所以也就无法运行。单纯的增加物理内存只能解决一部分问题，但是仍然会出现无法装入单个或者无法同时装入多个程序的问题。

虚拟内存就是具有请求调入功能和置换功能，可以从逻辑上对内存容量加以扩充的一种存储器系统。它使得应用程序认为它拥有连续可用的内存（一个连续完整的地址空间），允许运行比实际系统拥有的内存大得多的程序。而实际上，它通常被分割成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

**虚拟内存的基本思想**：每个程序拥有自己的地址空间，这个空间被分割成很多块，每一块称作一页或页面。每一页有连续的地址范围。这些也被映射到物理内存，但并不是所有的页都必须在内存中才能运行程序。当程序引用到一部分在物理内存中的地址空间时，由硬件立刻执行必要的映射。当程序引用到一部分不再物理内存中的地址空间时，由操作系统负责将缺失部分装入物理内存并重新执行失败的指令，这个过程或陷阱称为**缺页中断（或页错误）**。

**虚存的代价**：

* 虚存的管理需要建立很多数据结构，占用额外内存。
* 虚拟地址到物理地址的转换，增加了指令执行时间。
* 页式的换入换出需要磁盘I/O，耗费时间。
* 如果一页中只有部分数据，浪费内存。

**例题：计算缺页中断次数。**（2014淘宝笔试题）  
有一虚拟存储系统，若进程在内存中占3页（开始时内存为空），若采用先进先出页面置换算法，当执行如下访页页号序列后1，2，3，4，1，2，5，1，2，3，4，5，会产生（10）次缺页？

**4.3 页面置换算法**

1. **最优页面置换算法**：只具有理论意义的算法，用来评价其他页面置换算法。置换策略是将当前页面中在未来最长时间内不会被访问的页置换出去。
2. **先进先出置换算法**：由操作系统维护一个所有当前在内存中的页面的链表，最新进入的页面放在表尾，最早进入的页面放在表头；当发生缺页中断时，淘汰表头的页面并将新调入的页面追加到表尾。简单粗暴的一种置换算法，没有考虑页面访问频率信息。
3. **最近最少使用算法LRU**：算法赋予每个页面一个访问字段，用来记录上次页面被访问到现在所经历的时间t，每次置换的时候把t值最大的页面置换出去(实现方面可以采用寄存器或者栈的方式实现)。
4. **时钟置换算法**(也被称为最近未使用算法NRU)：页面设置一个访问位，页面被访问的时候访问位设置为1；并将所有页面保存在一个循环队列中，表针指向最老的页面。页面置换的时候，如果当前指针所指页面访问为为0，那么置换，否则将其置为0，循环直到遇到一个访问为位0的页面。
5. **改进型Clock算法**：在Clock算法的基础上添加一个修改位，替换时根究访问位和修改位综合判断。优先替换访问为何修改位都是0的页面，其次是访问位为0修改位为1的页面。
6. **最少使用算法LFU**：设置寄存器记录页面被访问次数，每次置换的时候置换当前访问次数最少的。存在问题是该访问寄存器并不能真正反映当前页面访问次数，因为访问速度比较快，所以在更新寄存器的时间间隔内访问1次和访问100次都是一样的。另外，LFU和LRU是很类似的，支持硬件也是一样的，但是区分两者的关键在于一个以时间为标准，一个以次数为标准(例如对于寄存器 pa 001111 和pb 111000，两个页面，如果采用LRU，那么被淘汰的是pa，如果采用LFU那么被淘汰的是pb)。
7. **页面缓冲算法PBA**：置换的时候，页面无论是否被修改过，都不被置换到磁盘，而是先暂留在内存中的页面链表(已修改页面链表和未修改页面链表，也可以不区分)里面，当其再次被访问的时候可以直接从这些链表中取出而不必进行磁盘IO，当链表中已修改也难数目达到一定数量之后，进行依次写磁盘操作(相当于将多次IO合并为一次)

**5. 链接**

\*\*静态链接库的优点 \*\*

* 代码装载速度快，执行速度略比动态链接库快；
* 只需保证在开发者的计算机中有正确的.LIB文件，在以二进制形式发布程序时不需考虑在用户的计算机上.LIB文件是否存在及版本问题，可避免DLL地狱等问题。

**动态链接库的优点**

* 更加节省内存并减少页面交换；
* DLL文件与EXE文件独立，只要输出接口不变（即名称、参数、返回值类型和调用约定不变），更换DLL文件不会对EXE文件造成任何影响，因而极大地提高了可维护性和可扩展性；
* 不同编程语言编写的程序只要按照函数调用约定就可以调用同一个DLL函数；
* 适用于大规模的软件开发，使开发过程独立、耦合度小，便于不同开发者和开发组织之间进行开发和测试。

**不足之处**

* 使用静态链接生成的可执行文件体积较大，包含相同的公共代码，造成浪费；
* 使用动态链接库的应用程序不是自完备的，它依赖的DLL模块也要存在，如果使用载入时动态链接，程序启动时发现DLL不存在，系统将终止程序并给出错误信息。而使用运行时动态链接，系统不会终止，但由于DLL中的导出函数不可用，程序会加载失败；速度比静态链接慢。当某个模块更新后，如果新模块与旧的模块不兼容，那么那些需要该模块才能运行的软件，统统撕掉。这在早期Windows中很常见。