2.6 进程调度

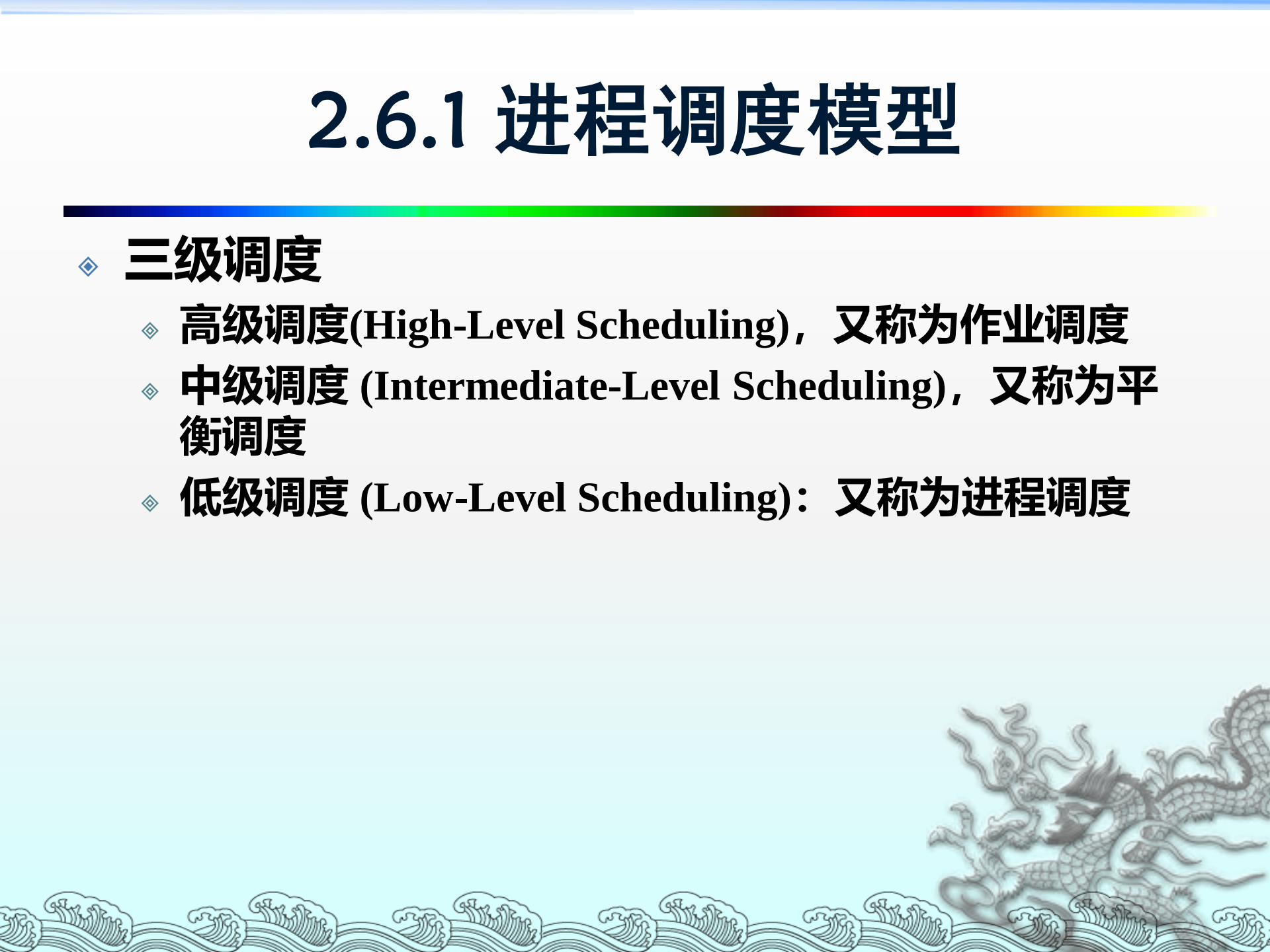
- 由于进程总数一般多于CPU数，必然会出现竞争CPU的情况。进程调度的功能就是按一定策略、动态地把CPU分配给处于就绪队列中的某一进程执行
- 两种基本的进程调度方式，抢占方式和非抢占方式，也称剥夺式（preemptive）和非剥夺式（non_preemptive）调度
- 剥夺原则有：优先权原则、短进程优先原则、时间片原则

可能引发进程调度的时机：

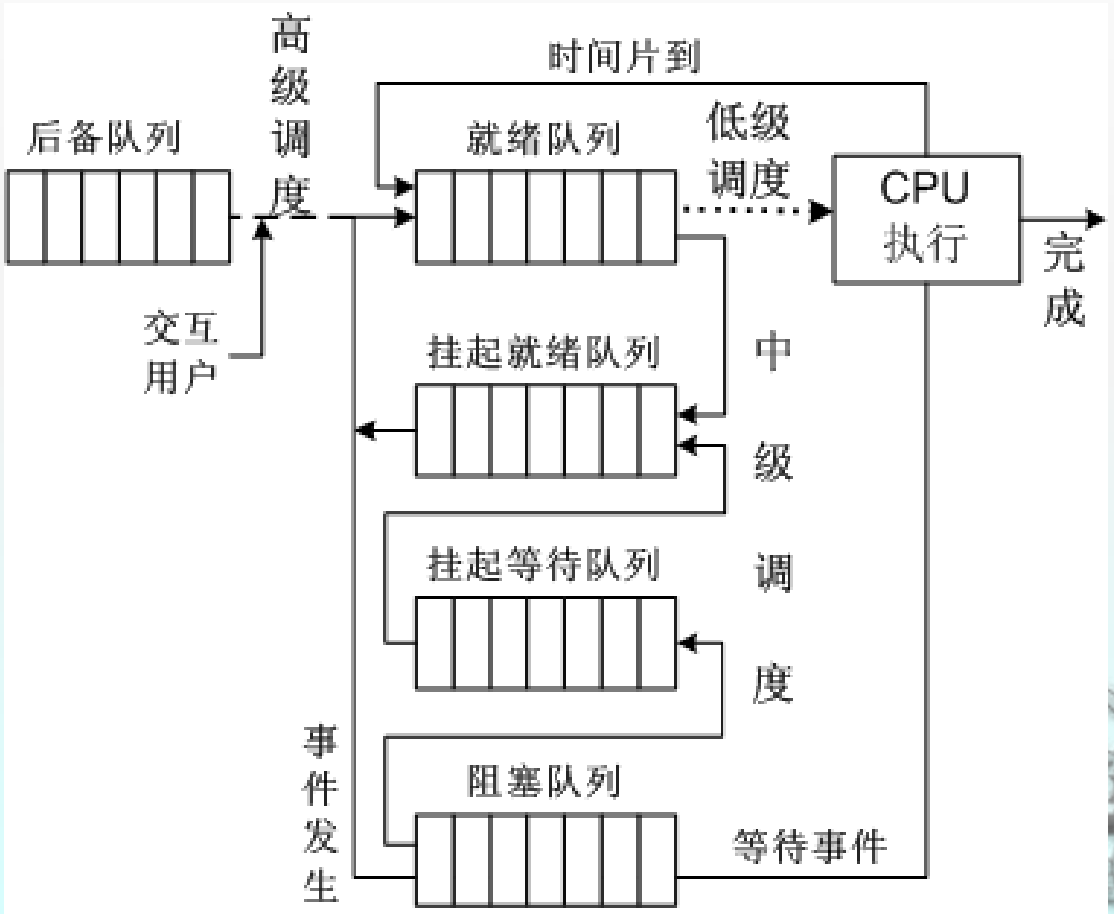
- 正在运行的进程运行完毕；
- 运行中的进程要求I/O操作；
- 执行某种原语操作(如P操作)导致进程阻塞；
- 比正在运行的进程优先级更高的进程进入就绪队列；
- 分配给运行进程的时间片已经用完

2.6.1 进程调度模型

- 三级调度
  - 高级调度(High-Level Scheduling)，又称为作业调度
  - 中级调度 (Intermediate-Level Scheduling)，又称为平衡调度
  - 低级调度 (Low-Level Scheduling)：又称为进程调度



三级调度模型


高级调度
时间片到
后备队列
就绪队列
低级调度
CPU
执行
完成
交互用户
挂起就绪队列
中级调度
挂起等待队列
事件发生
阻塞队列
等待事件

2.6.2 调度算法选择/评价准则

- 调度算法也称为调度策略
- 评价调度算法的准则如下：
  - 处理器利用率（CPU utilization）

    = CPU有效工作时间 / CPU总的运行时间
  - 响应时间（response time）：交互环境下用户从键盘提交请求开始，到系统首次产生响应为止的时间
  - 周转时间（turnaround time）Ti = Tf – Ts，即：周转时间 = 完成时刻 - 提交时刻

- 带权周转时间—— Wi = 作业的周转时间Ti / 系统为作业提供的服务时间Tsi，显然带权周转时间总大于1
- 平均作业周转时间 T = (ΣTi) / n
- 平均作业带权周转时间W = (ΣWi) / n
- 系统吞吐量（throughput）——单位时间内完成的进程数目
- 公平性——不出现饥饿情况

2.6.3 调度算法

- 操作系统中存在多种调度算法，有的适用于高级调度，有的适用于低级调度，但大多数算法既适用于高级调度，也适用于低级调度

1．先来先服务（First-Come First-Served，FCFS）——按进程就绪的先后顺序来调度，到达得越早，就越先执行

- 获得CPU的进程，未遇到其它情况时，一直运行下去
- 是一种非抢占式算法
- 没有考虑执行时间长短、运行特性和资源的要求

- FCFS调度算法适用性
  - 对长作业非常有利，对短作业不利
  - 对CPU繁忙型的作业非常有利，对I/O繁忙型作业非常不利
  - 非抢占式算法，对响应时间要求高的进程不利
  - 平均作业周转时间与作业提交的顺序有关

- **【例2-1】系统中现有5 个作业A、B、C、D、E同时提交（到达顺序也为ABCDE），其预计运行时间分别10、1、2、1、5个时间单位，如表所示，计算FCFS调度下作业的平均周转时间和平均带权周转时间**

作业ID	预计需运行时间
A	10
B	1
C	2
D	1
E	5

- **设作业到达时刻为0，根据定义计算，系统运行情况**

作业ID	运行时间	等待时间	开始时间	完成时间	周转时间	带权周转时间
A	10	0	0	10	10	1
B	1	10	10	11	11	11
C	2	11	11	13	13	6.5
D	1	13	13	14	14	14
E	5	14	14	19	19	3.8
平均周转时间 平均带权周转时间	T =（10+11+13+14+19）/ 5 = 13.4 W =（1+11+6.5+14+3.8）/ 5 = 7.26					

- 2．短作业优先（Shortest-Job-First，SJF）——以进入系统的作业所要求的CPU服务时间为标准，总选取估计所需CPU时间最短的作业优先投入运行。
- SJF调度算法适用性：
  - 算法易于实现，效率不高，主要弱点是忽视了作业等待时间。
  - 对长作业不利，如果系统不断接收短作业，可能会出现饥饿现象。
  - 非抢占式算法，对响应时间要求高的进程不利。
  - SJF的平均作业周转时间比FCFS要小，故它的调度性能比FCFS好。
  - 实现SJF调度算法需要知道作业所需运行时间，而要精确知道一个作业的运行时间是办不到的。

【例2-2】同例2-1，但采用SJF 算法调度作业

运行次序	运行时间	等待时间	开始时间	完成时间	周转时间	带权周转时间
B	1	0	0	1	1	1
D	1	1	1	2	2	2
C	2	2	2	4	4	2
E	5	4	4	9	9	1.8
A	10	9	9	19	19	1.9
平均周转时间 平均带权周转时间	T =（1+2+4+9+19）/ 5 = 7 W =（1+2+2+1.8+1.9）/ 5 = 1.74					

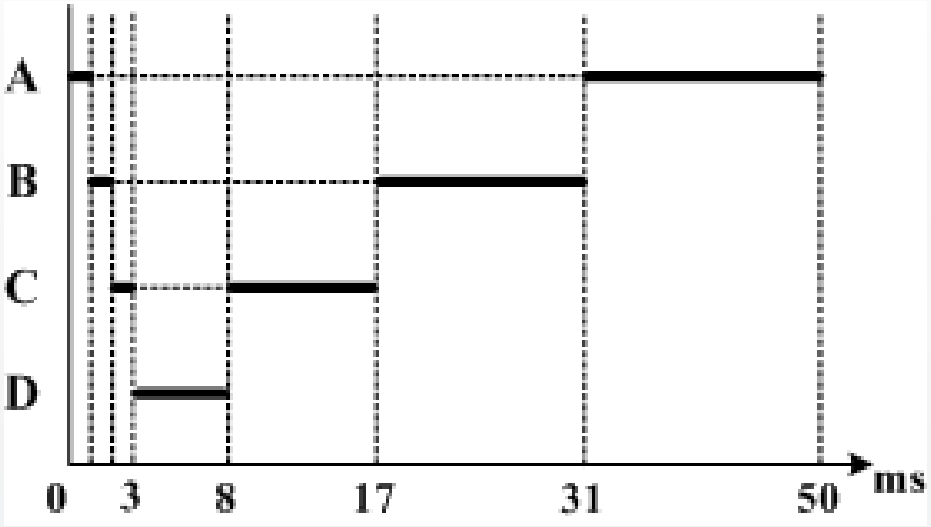
3．最短剩余时间优先（Shortest Remaining Time First，SRTF）

- 若一就绪状态的新作业所需的CPU时间比当前正在执行的作业剩余任务所需CPU时间还短，SRTF将打断正在执行作业，将执行权分配给新作业
- SRTF将SJF算法改为抢占式，因此只要有新作业进入就绪队列，就可能会引发进程切换。

SRTF调度算法适用性：

- 长进程仍有可能出现饥饿现象
- 必须计算运行、剩余时间，系统开销增大
- 因抢占式调度，系统性能会比SJF要好

- 【例2-3】作业A、B、C、D需要运行的时间分别为20ms、15ms、10ms、5ms。A作业在0ms到达，B作业在1ms到达，C作业在2ms到达，D作业在3ms到达。计算SRTF调度下作业的平均周转时间和平均带权周转时间



- A、B、C、D作业的周转时间分别为50ms、30ms、15ms、5ms。
- 平均周转时间为（50 + 30 + 15 + 5）/ 4 = 25.00 ms。
- A、B、C、D作业的带权周转时间为分别为2.5、2、1.5、1。
- 平均带权周转时间为（2.5 + 2 + 1.5 + 1）/ 4 = 1.75

4．高响应比优先（Highest Response Ratio First，HRRF）

——是FCFS与SJF两种算法的折衷——既考虑作业等待时间，又考虑作业的运行时间，既照顾短作业又不使长作业等待过久，改善了调度性能，**仍属于非抢占式算法**

- 响应比为作业的响应时间与作业所需运行时间之比，简化为：

  响应比 = 1 + （已等待的时间 / 估计运行时间）

HRRF算法适用性：

- 由定义可知，短作业容易得到较高响应比，长作业在等待了足够长的时间后，也将获得足够高的响应比，因此不会发生饥饿现象
- 需要经常计算作业的响应比，导致额外的开销
- HRRF算法的平均周转时间和平均带权周转时间都介于FCFS与SJF算法之间，比SJF算法差，比FCFS算法优
- 虽然HRRF算法的平均周转时间和平均带权周转时间不及SJF算法，但是，在现实中其可以实现，结果也比较可靠
- 如果在算法中引入抢占调度，则算法过程会更复杂。因为所有作业的响应比是动态变化的，抢占时间的计算需要解多个方程得到

- **【例2-4】系统中现有3 个作业A、B、C先后提交（到达），其参数如表所示，计算HRRF调度下作业的平均周转时间和平均带权周转时间**

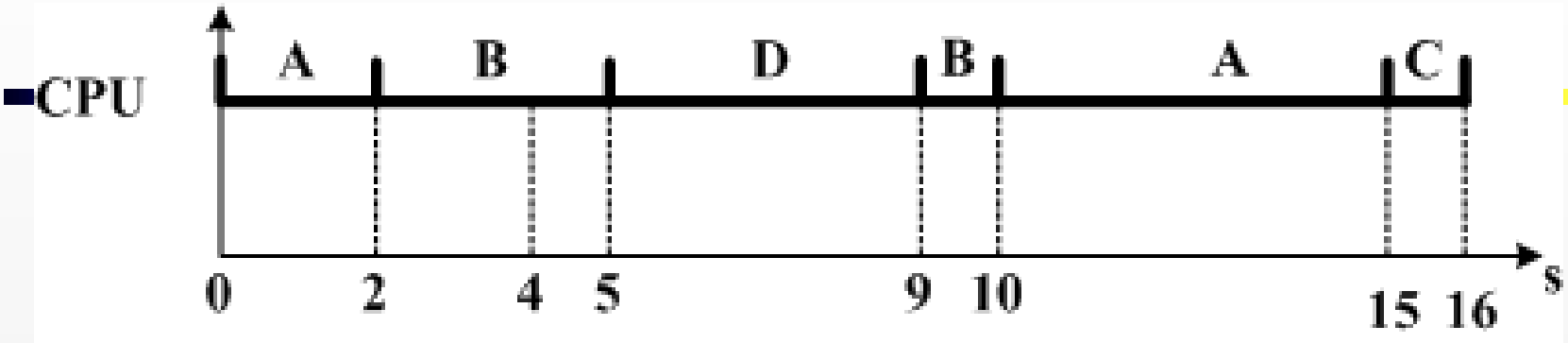
作业ID	提交（到达）时间	预计需运行时间
A	00:00	2:00
B	00:10	1:00
C	00:25	0:25

作业ID	提交时间	运行时间	开始时间	完成时间	周转时间	带权周转时间
A	00:00	2:00	00:00	02:00	2	1
B	00:10	1:00	02:25	03:25	3.25	3.25
C	00:25	0:25	02:00	02:25	2	4.8
平均周转时间 平均带权周转时间		T =（2+3.25+2）/ 3 = 2.4 W =（1+3.25+4.8）/ 3 = 3				

- 5．优先权（Highest-Priority-First，HPF）——根据进程的优先权进行进程调度，每次总是选取优先权高的进程调度，也称优先级调度算法，**一般是抢占式调度。**
- 优先权通常用一个整数表示，也叫优先数
  - Windows系统中有0~31共32个优先级，31最高
  - Unix系统中，使用数值-20~+19来表示优先级，-20优先级最高
- 优先权可由系统或用户给定
  - 静态优先权
  - 动态优先权

- **【例2-5】系统的进程调度采用抢占式优先权调度算法，优先数越小优先级越高，其参数如表所示，求平均周转时间和平均等待时间**

作业ID	提交（到达）时间	预计需运行时间	优先数
A	0	7	3
B	2	4	2
C	4	1	4
D	5	4	1



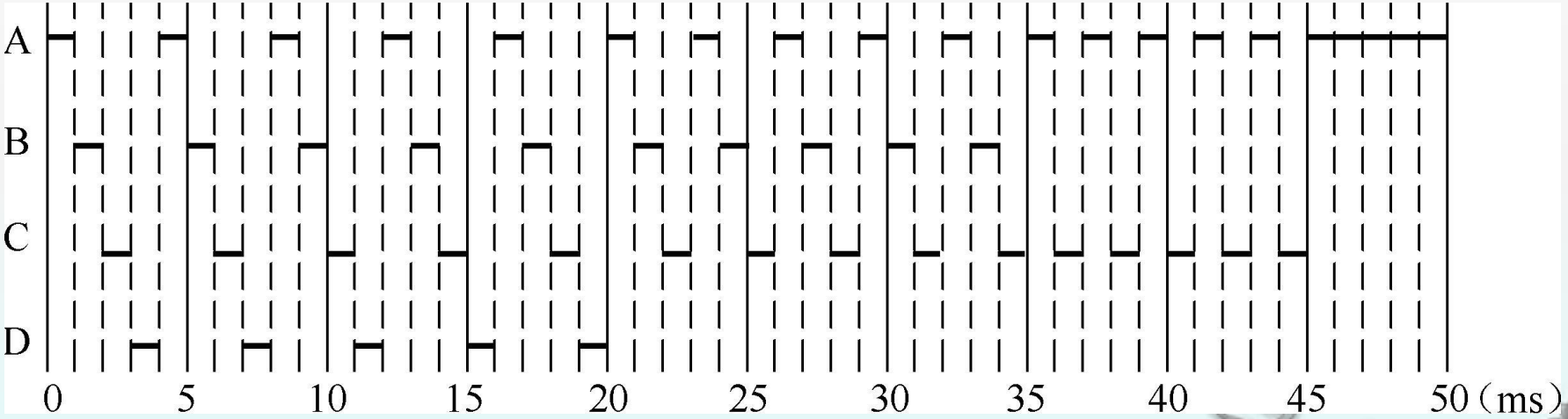
作业ID	提交（到达）时间	运行结束时间
A	0	15
B	2	10
C	4	16
D	5	9

平均周转时间T =（15+8+12+4）/ 4 = 9.75

平均等待时间Tw =（8+4+11+0）/ 4 = 5.75

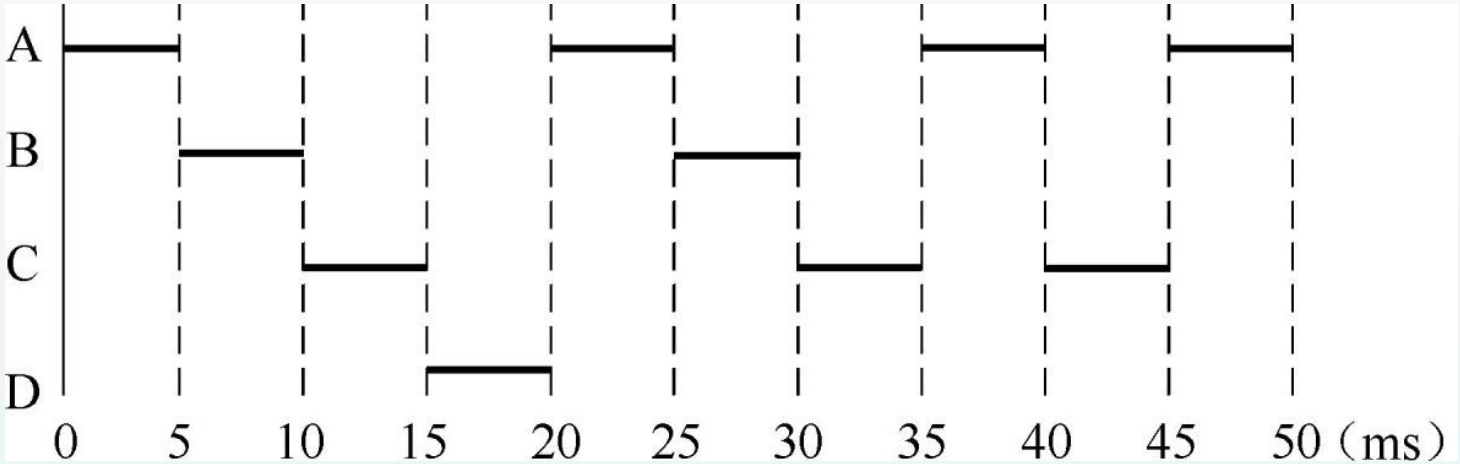
- 6．时间片轮转（Round-Ribon，RR）——调度程序把CPU分配给进程使用一个规定的时段，称为一个时间片（如100ms），就绪队列中的进程轮流获得CPU的一个时间片。当一个时间片结束时，系统剥夺该进程执行权，等候下一轮调度，属于抢占式调度
- 时间片的长短，影响进程的进度
  - 需要从进程数、切换开销、系统效率和响应时间等方面综合考虑，确定时间片大小

例:进程A、B、C、D需要处理的时间分别为20ms、10ms、15ms、5ms，在0时间达到。达到的先后顺序为A→B→C→D。如果时间片为1，则调度情况如图所示。



进程A、B、C、D周转时间分别为50ms、34ms、45ms、20ms，平均周转时间为37.25ms。

如果时间片为5ms，则调度情况如图所示。



进程A、B、C、D周转时间分别为50ms、30ms、45ms、20ms，平均周转时间为38.25ms。

如果时间片为10ms，则调度情况如图所示。

图　时间片为10的TRR调度

表 2.10 时间片与进程周转时间的关系

周转时间	A (ms)	B (ms)	C (ms)	D (ms)	平均周转时间 (ms)
时间片1	50	34	45	20	37.5
时间片2	50	33	46	23	38.0
时间片3	50	36	45	23	38.5
时间片4	50	35	46	29	40.0
时间片5	50	30	45	20	38.25
时间片6	50	33	48	23	39.75
时间片7	50	36	50	26	40.25
时间片8	50	39	46	29	41.00
时间片9	50	42	48	32	43.50
时间片10	45	20	50	35	40.00
时间片11	46	32	50	37	41.25
时间片12	47	22	50	39	39.50

- 时间片调度算法是一种抢占式的算法。
- 系统的花销主要体现在进程切换上。
- 当时间片大到每个进程足以完成时，时间片调度算法便退化为FCFS算法。

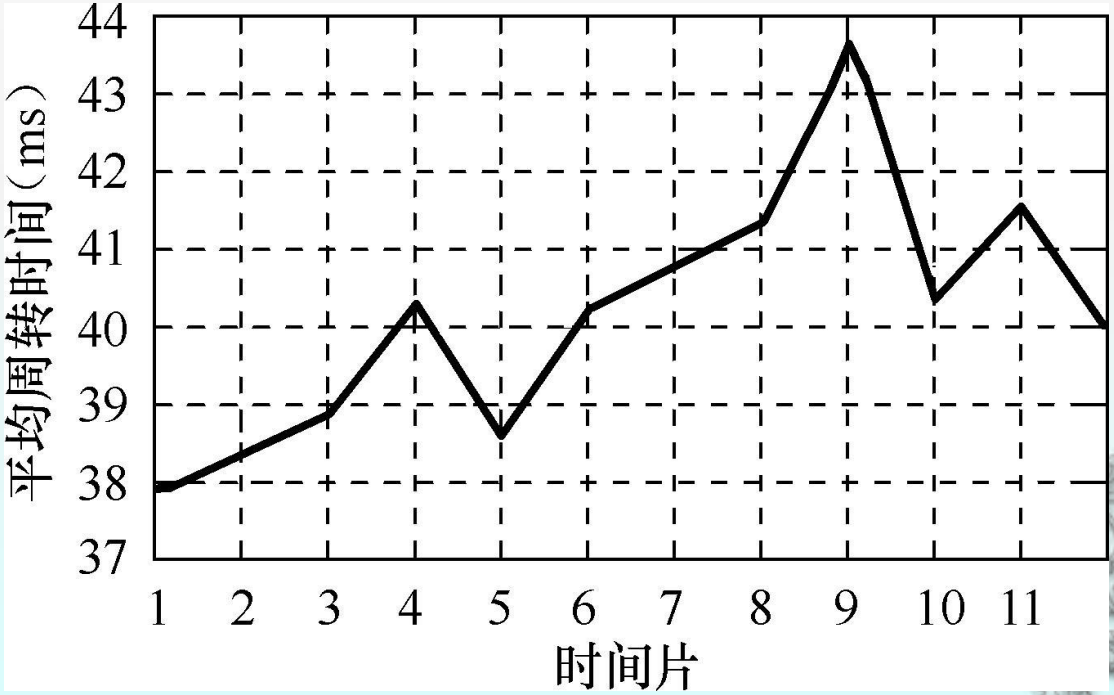


图3.16　时间片和平均周转时间关系

- 7．多级反馈队列（Multilevel-Feed-Queue，MFQ）——又称反馈循环队列，是一种基于时间片的进程多级队列调度算法的改进算法。系统设置多个就绪队列，最高级就绪队列的优先级最高，随着就绪队列级别的降低优先级依次下降，较高级就绪队列的进程获得较短的时间片
- 不需事先知道各进程所需运行时间，因而可行性较高，同时综合考虑了进程的时间和优先权因素，既照顾了短进程，又照顾了长进程，是一种综合调度算法，被广泛应用于各种操作系统中

多级反馈调度队列如图所示。

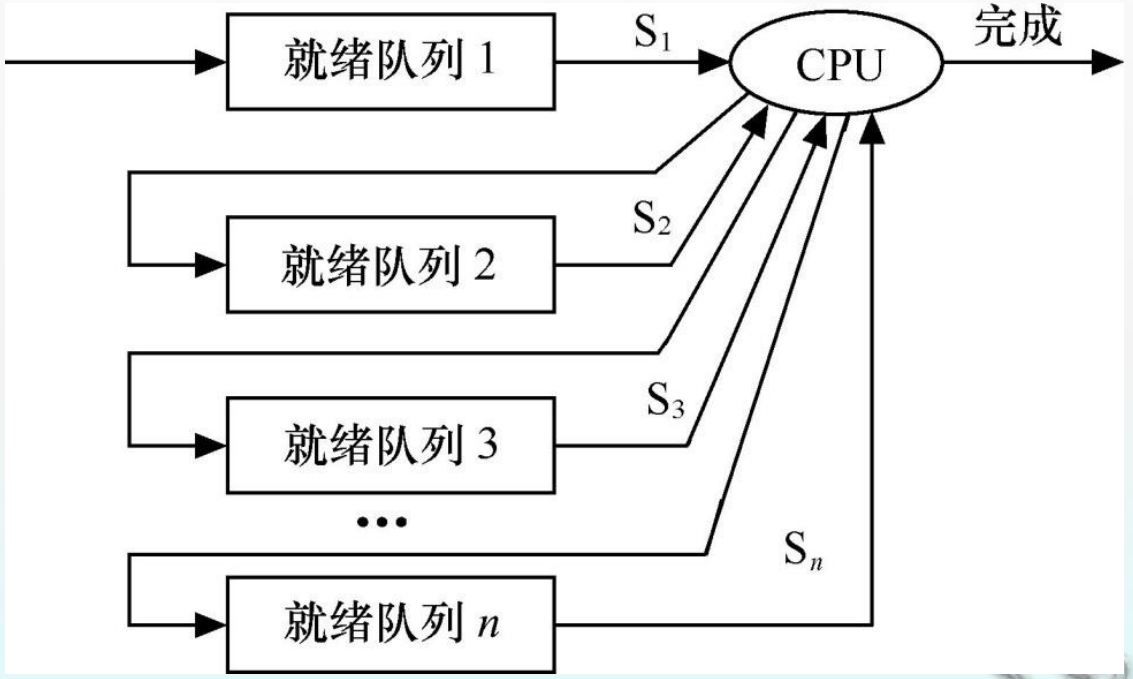


图2.19　多级反馈队列调度算法

- 多级反馈队列调度算法既照顾了短进程，又照顾了长进程，是一种综合调度算法。
- 多级反馈队列调度算法在许多操作系统中得到了应用，如分时操作系统中的UNIX操作系统和Linux操作系统。

在BSD UNIX操作系统中，进程的就绪队列有32个，按照0-31号的顺序，优先级逐渐降低。其中0-7号就绪队列用于系统进程，8-31号就绪队列用于用户进程。

系统进程的优先级高于用户进程。在每个就绪队列内部，采用的是时间片轮转调度，时间片是变化的，最大时间片不能超过100ms。

2.6.4 多CPU系统中的调度

- 多处理器系统的作用是利用系统内的多个CPU来并行执行用户进程，以提高系统的吞吐量或用来进行冗余操作以提高系统的可靠性
- 系统的多个处理器在物理上处于同一机壳中，有一个单一的系统物理地址空间，多个处理器共享系统内存、外设等资源

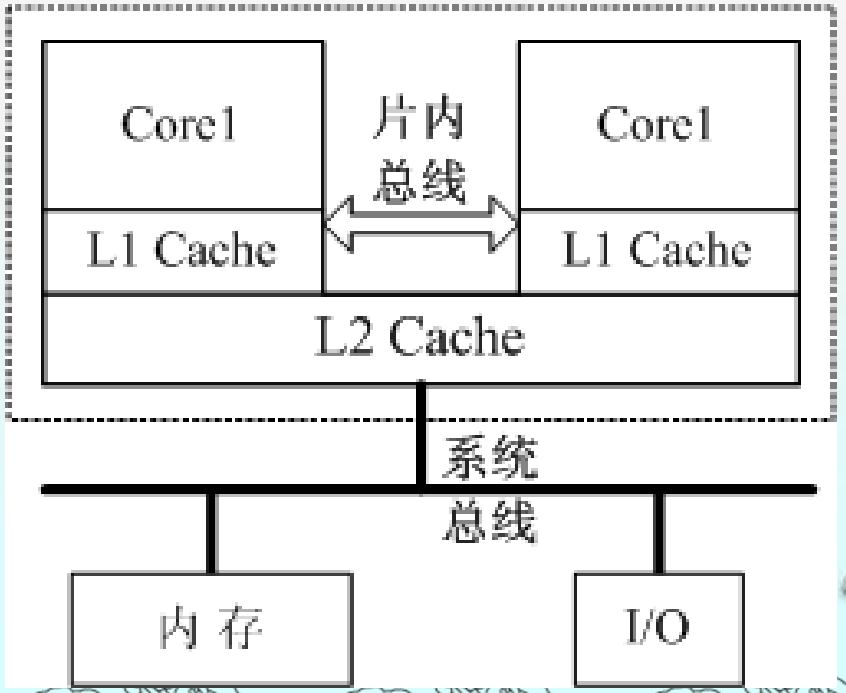
多CPU系统的类型主要有两种：

- 主-从模式，只有一个主处理器，运行操作系统，管理整个系统资源，并负责为各从处理器分配任务，从处理器有多个，执行预先规定的任务及由主处理器分配的任务。这种类型的系统无法做到负载平衡，可靠性不高，很少使用。
- 对称处理器模式SMP（Symmetric MultiProcessor），所有处理器都是相同的、平等的，共享一个操作系统，每个处理器都可以运行操作系统代码，管理系统资源。是目前比较常见的多CPU系统类型。

- 多处理器系统中，比较有代表性的进（线）程调度方式有以下几种方式：
  - 1）自调度
  - 2）组调度/群调度（Gang Scheduling）
  - 3）专用处理器分配
  - 4）动态调度

2.6.5 多核CPU中的调度

- 多核处理器是指在一枚处理器中集成两个或多个计算引擎（内核/core），称为CMP（Chip multiprocessors）结构，可在特定的时钟周期内执行更多任务，通常采用共享二级Cache的结构



- 对于多核CPU，优化操作系统任务调度算法是保证效率的关键
  - 全局队列调度——（多数系统）
  - 局部队列调度
- 中断处理——全局中断控制器也需要封装在芯片内部
- 系统提供同步与互斥机制——需要利用硬件提供的“读 - 修改 - 写”的原子操作或其他同步互斥机制

2.7 死锁

- 一个进程集合中的每个进程都在等待只能由该集合中的其它一个进程才能引发的事件，则称一组进程或系统此时发生了死锁
- 死锁一旦发生，会使整个系统瘫痪而无法工作。因此，必须想办法解决死锁问题

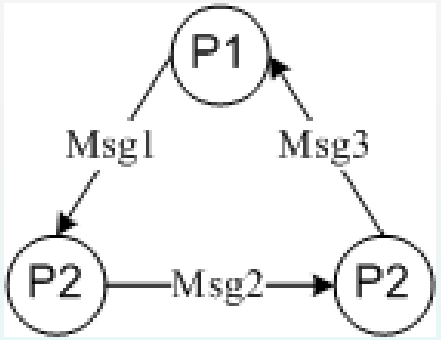
2.7.1死锁的产生原因

- 死锁产生的原因主要有两个：
  - **并发进程对临界资源的竞争**
  - **并发进程推进顺序不当**
    - 如果进程间推进顺序如下：

      P1：receive（S3），send（S1）；

      P2：receive（S1），send（S2）；

      P3：receive（S2），send（S3）；
    - 三个进程永远都不能接收到所需要的信息，不能继续运行，发生了死锁。



2.7.2 死锁产生的必要条件

- 1971年，Coffman等人总结出死锁发生的四个必要条件。
- 1）互斥条件（Mutual exclusion）——资源的使用是互斥的
- 2）占有并等待条件（Hold and wait）——已经得到某些资源的进程申请新资源时，已得到的资料不释放。
- 3）不剥夺条件（No pre-emption）——系统或其它进程不能剥夺进程已经获得的资源。
- 4）环路等待条件（Circular wait）——系统中若干进程间形成等待环路

- 只要破坏上述几个条件之一，即可防止死锁
  - 破坏第1个条件，使资源可同时访问而不是互斥使用，可行性较差
  - 可重入程序，只读数据文件，时钟、磁盘等可以采用这种方法，但许多资源如打印机，可写文件等由于其特殊性质决定只能互斥地占有。因此，对大多数资源来讲，破坏互斥条件根本不现实。

- 只要破坏上述几个条件之一，即可防止死锁
  - 破坏第2个条件，进程必须获得所需的所有资源才能运行——静态分配，严重降低资源利用效率。
  - 静态分配策略：一个进程所需要的全部资源必须在该进程执行前申请并得到后，进程才能执行。如果进程不能得到全部所需要的资源，则系统不对进程进行资源分配，进程必须等待。会严重降低资源利用率，因为在每个进程所有占有的资源中，有些资源在运行后期才用甚至在例外情况下才用。

- 只要破坏上述几个条件之一，即可防止死锁
  - 破坏第3个条件，采用剥夺式调度方法，只适用于CPU和内存分配
  - 实现起来复杂，且要付出很大的代价。而且反复申请和释放资源，延长进程周转时间，增加系统开销，同时降低了系统的吞吐量。

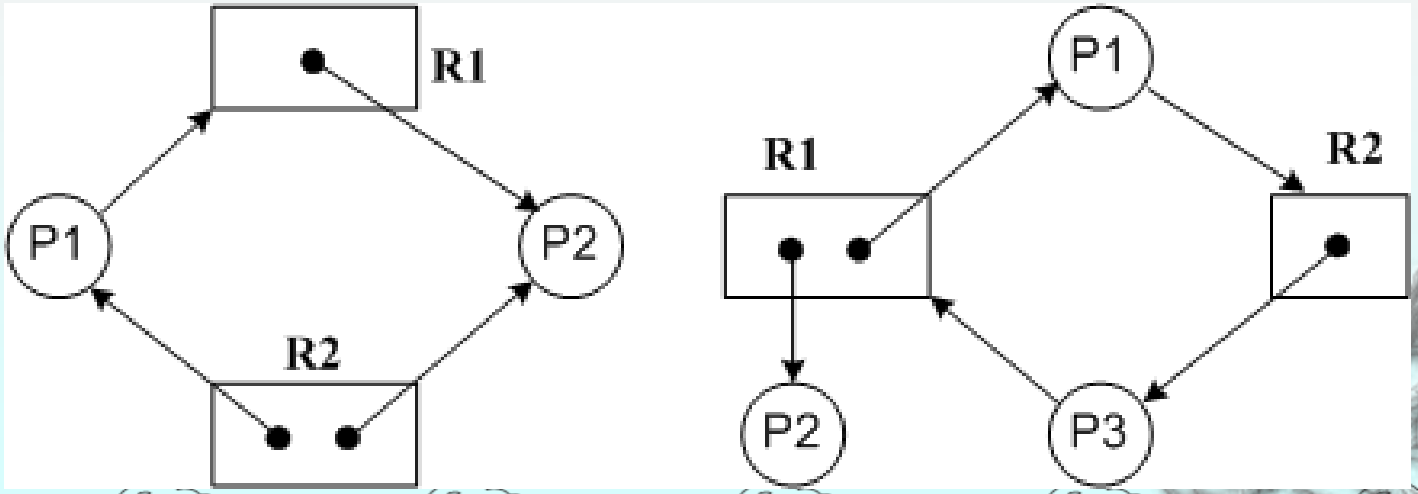
- 只要破坏上述几个条件之一，即可防止死锁
  - 破坏条件4，采用层次分配策略——按此策略分配资源时系统不会发生死锁
  - 阻止环路等待：

    采用层次分配策略，先定义资源类型的线性顺序，如果一个进程已经分配到了R类型的资源，那么他接下来请求的资源只能是排在R类型后面的资源。释放资源时，先释放更高层资源。
  - 证明其正确性：

    i<j,假设两个进程A和B死锁，原因是A获得Ri并请求Rj, 而B获得Rj并请求Ri,则这种情况是不可能的，因为这意为着i<j 且j<i。
- 这种方法是低效的，会限制新设备类型的增加，会使执行速度变慢，并可能在没有必要的情况下拒绝资源访问

系统资源分配图

- 圆圈表示进程，资源类用方框表示，框中的圆点代表单个该类资源，有向边连接进程和资源
- 申请边从进程指向资源类方框，表示进程正在等待资源；分配边从单个资源圆点指向进程，表示进程已经获得资源



- 根据进程-资源分配图定义得出如下结论：
  - 1）如果进程-资源分配图中无环路，则此时系统没有发生死锁
  - 2）如果进程-资源分配图中有环路，且每个资源类中仅有一个资源，则系统中发生了死锁，此时，环路是系统发生死锁的充要条件，环路中的进程便为死锁进程
  - 3）如果进程-资源分配图中有环路，且涉及的资源类中有多个资源，则环路的存在只是产生死锁的必要条件而不是充分条件

- 进程-资源分配图可用以下步骤化简：
  - 1）在资源分配图中，找出一个既非等待又非孤立的进程结点Pi，由于Pi可获得它所需要的全部资源，且运行完后西方它所占有的全部资源，故可在资源分配图中消去Pi所有的申请边和分配边，使之成为既无申请边又无分配边的孤立结点；
  - 2）将Pi所释放的资源分配给申请它们的进程，即在资源分配图中将这些进程对资源的申请边改为分配边；
  - 3）重复前两步骤，直到找不到符合条件的进程结点。
- 经过化简后，若能消去资源分配图中的所有边，使所有进程都成为孤立结点，则称该图是可完全化简的，否则为不可化简的
- 系统为死锁状态的充分条件是：当且仅当该状态的进程-资源分配图是不可完全简化的——死锁定理。

2.7.3 死锁的避免

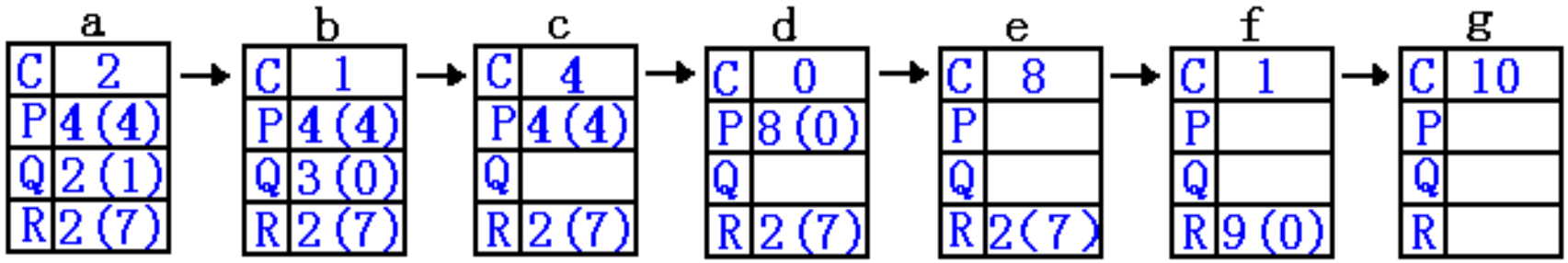
- 死锁避免法是通过资源分配算法分析系统是否存在一个并发进程的状态序列，在确定不会进程循环等待的情况下，才将资源真正分配给进程，以保证并发进程不会产生死锁。
- 死锁避免法能支持更高的进程并发度，动态地决定是否给进程分配资源——如果进程的资源请求方案会导致死锁，系统拒绝执行，反之，如果一个资源的分配会导致死锁，系统拒绝分配。

- Dijkstra在1965年提出了避免死锁的银行家调度算法，该算法是以银行系统所采用的借贷策略（尽可能放贷、尽快回收资金）为基础而建立的算法模型。

- 在此模型中，进程相当于贷款客户，系统资源相当于资金，调度程序相当于银行家（贷款经理）。

- 假定银行有可供借用资金10法郎，现有三个用户P、Q、R来银行办理借款业务，他们最大借款总额分别为8法郎，3法郎和9法郎。又假定在某时刻状态为：P已借4法郎尚需4法郎；Q已借2法郎尚需1法郎；R已借2法郎尚需7法郎。目前银行余款为2法郎。

按下图可以让所有顾客完成借款任务



在上图中，由状态a转入状态b，若先将1法郎借给R用户，使R变为3(6)状态，系统便会进入死锁。银行家算法是根据以上例子得出的。

银行家算法的思路：

- 1）在某一时刻，各进程已获得所需的部分资源。有一进程提出新的资源请求，系统将剩余资源试探性地分配给该进程。
- 2）如果此时剩余资源能够满足余下的某些进程的需求，则将剩余资源分配给能充分满足的、资源需求缺口最大的进程，运行结束后释放的资源再并入系统的剩余资源集合。
- 3）反复执行第2步，直到所有的进程都能够获得所需而运行结束。说明第1步的进程请求是可行的，系统处于安全状态，相应的进程执行序列称为系统的安全序列。如果所有的进程都试探过而不能将资源分配给进程，即不存在安全序列，则系统是不安全的。

银行家算法所需的数据结构：

- 假设系统中有n个进程，m类资源；
- 系统当前资源剩余量向量　Available[m] = { R_1，R_2，……，R_m }；
- n个进程对m类资源的需求声明矩阵
  Claim[n][m]，Claim[i][j]的值表示进程i对j类资源的总需求量；
- n个进程已获得的各类资源数量矩阵
  Possession[n][m]，Possession[i][j]的值k表示进程i已获得k个j类资源；

- n个进程的各类资源需求缺口矩阵

  Shortage[n][m] = Claim – Possession，Shortage[i][j]的值s表示进程i还需要（缺）s个j类资源；
- 某进程i在某时刻发出的资源请求向量

  Request[m]，取值随具体情况而定。
- 前4个数据结构及其取值确定了系统在某一时刻的状态，如果算法尝试资源分配方案能够使得所有进程安全运行完毕，则说明**该状态安全，资源分配方案可行。**

银行家算法的算法细化说明：

1）判断请求向量Request的有效性——超过相应进程总需求量则报错，超过系统目前剩余量则阻塞；

2）就系统资源剩余量对Request进行**试分配**：

Available[*] = Available[*] - Request[*];

Possession[i][*] = Possession[i][*] + Request[*];

Shortage[i][*] = Shortage[i][*] - Request[*];

3）执行安全性测试算法，若安全则确认试分配方案，否则进程i阻塞；

执行安全性测试算法细化说明：

- ① 定义工作向量Rest[*] = Available[*]，进程集合Running{*}，布尔量possible= true；
- ② 从Running集合中找出P_k，满足条件Shortage[k][*] < Rest[*]；
- ③ 找到合格的进程P_k，则释放其占用资源（Rest[*] = Rest[*] + Possession[k][*]），将其从Running集合中去掉，重复步骤②；

- ④ 找不到合格的进程P_k，possible为false，退出安全性测试算法；

- ⑤ 最终检查Running集合，为空则返回安全，非空则不安全。

银行家算法示例：

- 假设系统中有5个进程{ P_0，P_1，P_2，P_3，P_4}，3类系统资源{ A，B，C}，各拥有资源数{10，5，7}。T_0时刻系统状态如表2.8 所示。

表2.8 T_0时刻系统状态

进程ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P_0	7，5，3	0，1，0	7，4，3	3，3，2
P_1	3，2，2	2，0，0	1，2，2	
P_2	9，0，2	3，0，2	6，0，0	
P_3	2，2，2	2，1，1	0，1，1	
P_4	4，3，3	0，0，2	4，3，1	

- 可以断言目前系统处于安全状态，因为存在一个执行序列{P_1，P_3，P_4，P_2，P_0}，能满足安全性条件。

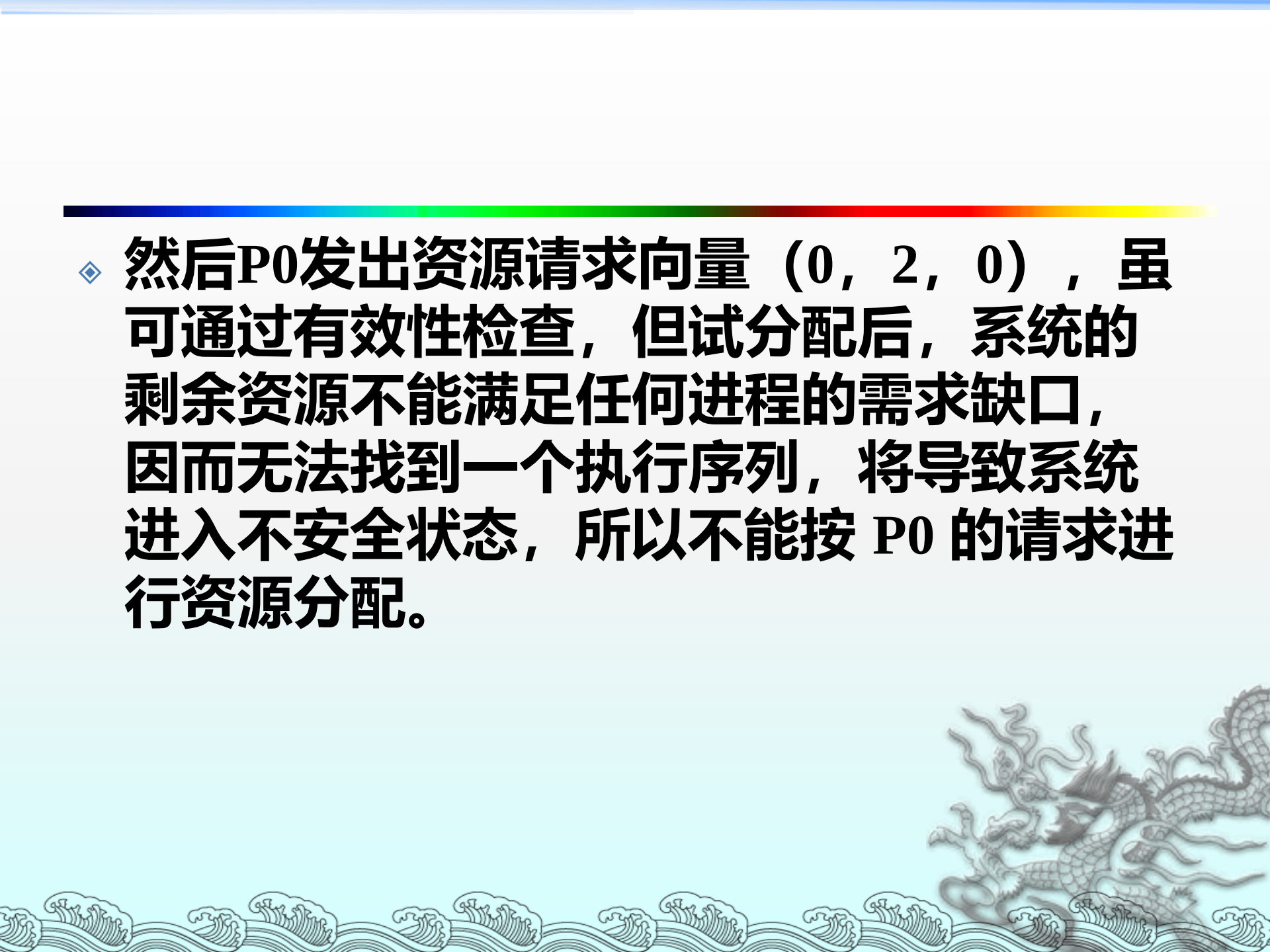
- 现有进程P_1发出资源请求Request = {1，0，2}。系统为了确定能否满足该请求，采用银行家算法中的安全性测试算法，首先检查Request的有效性：
- Request（1，0，2）≤Shortage[1]（1，2，2）；
- Request（1，0，2）≤Available（3，3，2）；

- **通过有效性检查后，系统按P1的请求进行试分配，此时（T1时刻）系统状态如表2.9 所示。**
- 表2. 9 T1时刻系统状态

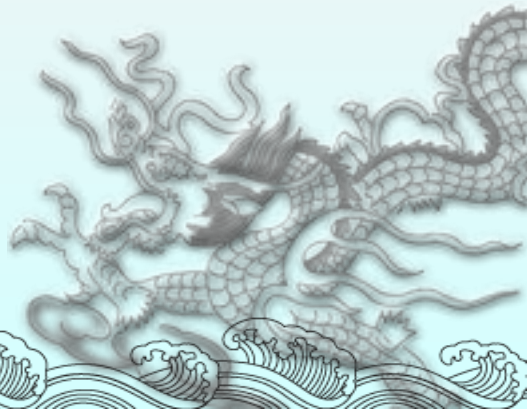
进程ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P0	7，5，3	0，1，0	7，4，3	2，3，0
P1	3，2，2	3，0，2	0，2，0	
P2	9，0，2	3，0，2	6，0，0	
P3	2，2，2	2，1，1	0，1，1	
P4	4，3，3	0，0，2	4，3，1	

- 经分析，仍存在一个执行序列{P1，P3，P4，P0，P2}，能满足安全性条件，因此刚才的试分配方案是可行的。
- 现又有进程P4发出资源请求向量（3，3，0），因系统剩余资源向量Available（2，3，0）小于该请求向量，所以无法通过有效性检查，P4进程阻塞。

- 然后P0发出资源请求向量（0，2，0），虽可通过有效性检查，但试分配后，系统的剩余资源不能满足任何进程的需求缺口，因而无法找到一个执行序列，将导致系统进入不安全状态，所以不能按 P0 的请求进行资源分配。



- 银行家算法是一个很经典的死锁避免算法，理论性很强，看起来似乎很完美，但其实现要求进程不相关，并且事先要知道进程总数和各进程所需资源情况，所以可行性并不高。



习题：设系统中有3种类型资源（A，B，C）和5个进程（P1，P2，P3，P4，P5），A资源数量为17，B资源数量为5，C资源数量为20，

在t0时刻系统状态如下表，剩余资源数为：2，3，3。系统采用银行算法实施死锁避免策略。请问：

（1）t_0时刻是否安全状态？若是，请计算出安全序列。

（2）在t_0时刻若进程P_4请求资源（0，3，4），是否能实施资源分配？为什么？

（3）在（2）的基础上，若进程P4请求资源（2，0，1），是否能实施资源分配？为什么？

（4）在（3）的基础上，若进程P1请求资源（0，2，0），是否能实施资源分配？为什么？

进程	最大资源需求量			已分配资源数量		
	A	B	C	A	B	C
P1	5	5	9	2	1	2
P2	5	3	6	4	0	2
P3	4	0	11	4	0	5
P4	4	2	5	2	0	4
P5	4	2	4	3	1	4

2.7.4 检测与解除

- 死锁的检测算法可以采用基于死锁定理的检测，也可以采用类似于银行家算法中的安全性测试算法，，如果算法退出时仍有未结束的进程，则系统不安全，那些未结束的进程就是死锁的进程。
- 只不过死锁检测的不是试分配之后的系统状态，而是系统当前状态，需要考虑检查每个进程还需要的所有资源能否满足要求。

由死锁发生的必要条件可知，环路等待是死锁发生时必须具备的现象。**有死锁，则一定有环路存在，但有环路存在不一定发生死锁。**

1．单个资源的死锁检测

当系统中有多类资源，但每类资源只有一个时，如只有一台打印机或一台磁带机，对于这样的简单情况，如果从资源分配图中去检测，有死锁，一定会有环路存在。如图所示。

资源R_1、R_2、R_3都只有一个，进程P_2、P_4、P_5构成环路，因此，进程P_2、P_4、P_5发生死锁。对于进程P_1、P_3，虽然没有在环上，但是，由于所需要的资源R_1、R_2都已经分配，受其他进程死锁的影响，也发生了死锁。

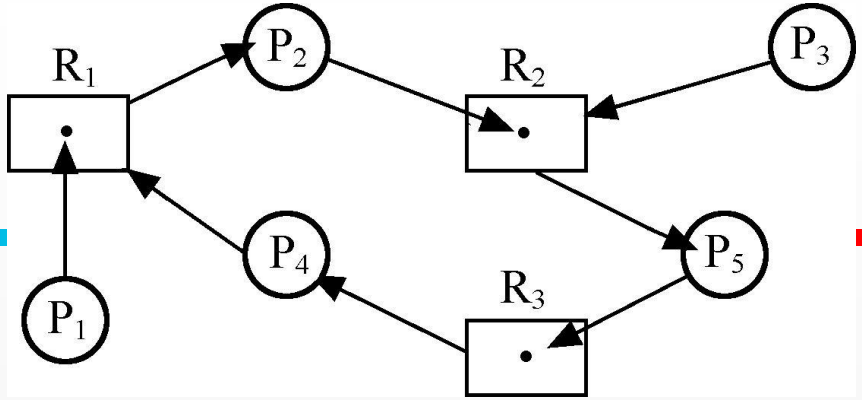


图　有环路有死锁的情况

针对资源分配图，可以采用将资源分配图中的每个节点看作为一棵树的树根，用深度优先搜索法搜索每一个节点，看该节点是否是原来已经搜索过的节点。如果是，则表示存在环路。如果从给定节点出发的有向边都已经搜索过了，则从当前节点出发的子图中不包含环。如果所有节点都是这样，则整个图不存在环，系统不存在死锁。

下面是针对只有一个资源的简单情况所作的死锁检测算法实现步骤的描述：

(1) 用L表示节点集合，L的初始值为空。

(2) 在资源分配图中，将图中的任意一个节点N作为初始节点。

(3) 将所选的初始节点添加到L的尾部，并检查该节点是否已经在L中出现过两次，如果是，则该图包含了一个环。算法结束。

(4) 从给定的节点开始，检测是否存在没有标记的从该节点出发的有向边，如果存在，向下进入第（5）步，否则跳到（6）步。

(5) 随机选取一条没有标记的从该节点出发的有向边，然后顺着这条有向边找到新的当前节点，返回到第（3）步。

(6) 移走该节点，返回到当前节点前面的一个节点，并将其作为新的当前节点，转到第（3）步。如果这一节点是起始节点，则表明该资源分配图不存在任何环。算法结束。

将上图作为实例，检测环，步骤如下：

(1) 首先，初始化L为空；

(2) 从节点P_1开始，将节点P_1添加到L中$L = \{ P_1 \}$，移动到唯一可以移动的节点R_1，将其添加到$L = \{ P_1, R_1 \}$，没有节点在L中出现过两次。

(3) 沿方向线从R_1到达节点P_2，使$L = \{ P_1, R_1, P_2 \}$，没有节点在L中出现过两次。沿方向线从P_2到达节点R_2，使$L = \{ P_1, R_1, P_2, R_2 \}$，没有节点在$L$中出现过两次。

(4) 沿方向线从R_2到达节点P_5，使$L = \{ P_1, R_1, P_2, R_2, P_5 \}$，没有节点在$L$中出现过两次。沿方向线从$P_5$到达节点$R_3$，使$L = \{ P_1, R_1, P_2, R_2, P_5, R_3 \}$，没有节点在$L$中出现过两次。

(5) 沿方向线从R_3到达节点P_4，使$L = \{ P_1, R_1, P_2, R_2, P_5, R_3, P_4 \}$，没有节点在$L$中出现过两次。沿方向线从$P_4$到达节点$R_1$，使$L = \{ P_1, R_1, P_2, R_2, P_5, R_3, P_4, R_1 \}$，节点$R_1$在$L$中出现过两次。则得到环$\{ R_1, P_2, R_2, P_5, R_3, P_4, R_1 \}$。

也可以选其他节点为起始节点，同样也可以得到该环。

这种算法只是针对每类资源只有一个的情况而言，不是一种最佳算法。

2．多个资源的死锁检测

如果相同的资源有多个，则需要用矩阵来表示每类资源的个数。用E_1，…，E_m分别表示每类资源的个数，用P_1，…，P_n表示检测的进程，用A_1，…，A_m表示每类资源可用的个数，用C_{11}，…，C_{nm}表示每类资源已经分配给进程P_1，…，P_n的个数，用R_{11}，…，R_{nm}表示进程P_1，…，P_n请求每类资源可用的个数。则对任意一类资源j，存在所有进程当前已经分配得到的资源加上可用资源为现有资源，即：

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

检测算法首先查找某个进程，其请求资源个数小于等于系统当前能够提供的资源个数，则能够将资源分配给该进程，该进程能够运行完成，将该进程标记为完成。将完成后释放的资源加入可用资源中，再分配给下一进程，依次类推，直到系统的所有进程都做过一次标记。再在没有标记完成的进程中检测是否此时系统能够提供的资源可以满足进程的请求，如果能够满足，则进程标记完成。这样，反复对不能标记完成的进程进行标记，最后，如果存在进程无法标记为完成，则这些进程为死锁进程。

多个资源的死锁检测与银行家算法的思想非常相似，只是银行家算法是在进程分配资源之前针对进程资源分配的一个规划，而死锁检测算法是在进程已经分配资源后进行的检查。

在系统中有进程P_1、P_2、P_3；有4台磁带机、2台绘图仪、4台扫描仪、1台打印机，

即现有资源$E=\{4,2,4,1\}$；当前已经分配给进程P_1、P_2、P_3的资源分别为：

$$C=\begin{bmatrix}0,0,1,0\\2,0,0,1\\0,1,2,0\end{bmatrix}$$

当前进程请求的资源数为

$$R=\begin{bmatrix}2,0,0,1\\1,0,1,0\\2,1,0,0\end{bmatrix}$$

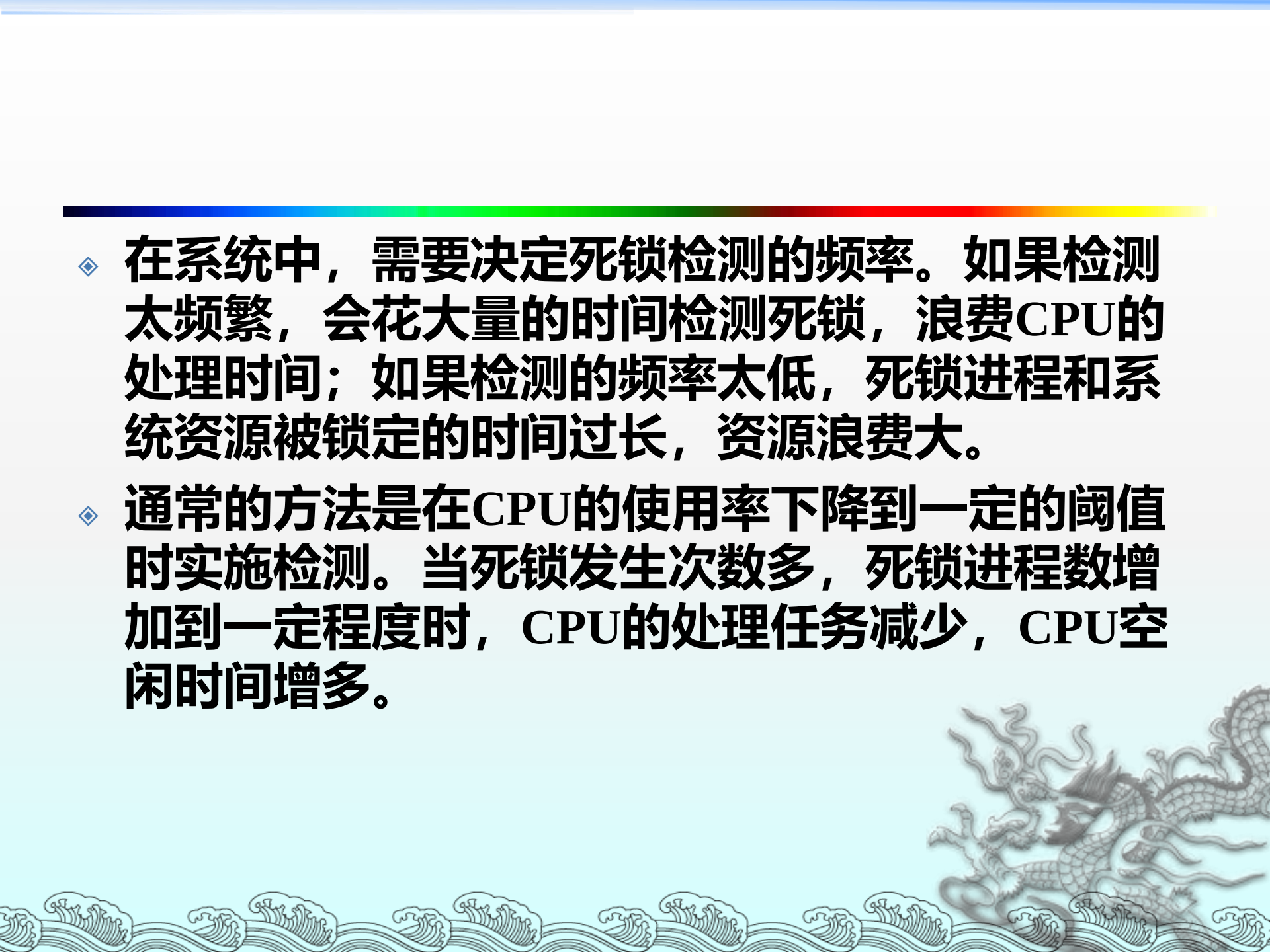
现在检测是否存在进程死锁。

首先算出当前能够提供的资源$R = \{2,1,1,0\}$；

然后找标记为完成的进程：

- 如果P_1进程请求为$\{2,0,0,1\}$，不能满足；
- 如果P_2请求资源$\{1,0,1,0\}$，能够得到满足，P_2标记为完成。P_2完成，并释放资源。释放的资源加上系统原来分配剩余的资源，总共能够提供的资源为$\{4,1,1,1\}$，能够满足进程P_3的请求$\{2,1,0,0\}$，P_3进程标记为完成。
- P3完成后释放的资源加上系统原来剩余的资源，能够提供的资源为$\{4,2,3,1\}$。此时系统没有标记完成的进程为P1，P1的请求为$\{2,0,0,1\}$能够满足，P1完成。
- 系统中所有的进程都标记完成，系统没有进程死锁。

- 在系统中，需要决定死锁检测的频率。如果检测太频繁，会花大量的时间检测死锁，浪费CPU的处理时间；如果检测的频率太低，死锁进程和系统资源被锁定的时间过长，资源浪费大。
- 通常的方法是在CPU的使用率下降到一定的阈值时实施检测。当死锁发生次数多，死锁进程数增加到一定程度时，CPU的处理任务减少，CPU空闲时间增多。



- 在系统成功地检测到死锁后，常用的死锁解除方法有以下几种：

- 重启：重新启动死锁进程，甚至重启操作系统。
- 撤销：撤销死锁进程，回收资源，优先选择占用资源最多或者撤销代价最小的，撤销一个不行就继续选择撤销进程，直至解除死锁。
- 剥夺：剥夺死锁进程资源再分配，选择原则同上。
- 回滚：根据系统保存的检查点，使进程或系统回退到死锁前的状态。

虽然大多数操作系统都潜在地受到死锁的威胁，但是，从解决死锁的角度出发，需要了解死锁发生的频率、系统受死锁影响的程度、系统崩溃的原因和发生的次数。如果死锁发生的频率极高，对系统的影响很大，甚至导致系统崩溃，则应该采取有力的措施去预防死锁、避免死锁，不惜代价检测死锁并解除死锁。但是，如果死锁几年发生一次，或者几个月会发生一次死锁，则通常不会选择牺牲系统性能和可用性去检测死锁、解除死锁。

2.8 线程的基本概念

- 80年代中期，人们提出了比进程更小的运行的基本单位——线程，用来提高系统内程序并发执行的速度，减少系统开销，从而可进一步提高系统的吞吐量。
- 近几年，线程概念已得到了广泛应用，不仅在新推出的操作系统中，大都已引入了线程概念，而且在新推出的数据库管理系统和其它应用软件中，也都采用多线程技术来改善系统的性能。

2.8.1 线程的引入

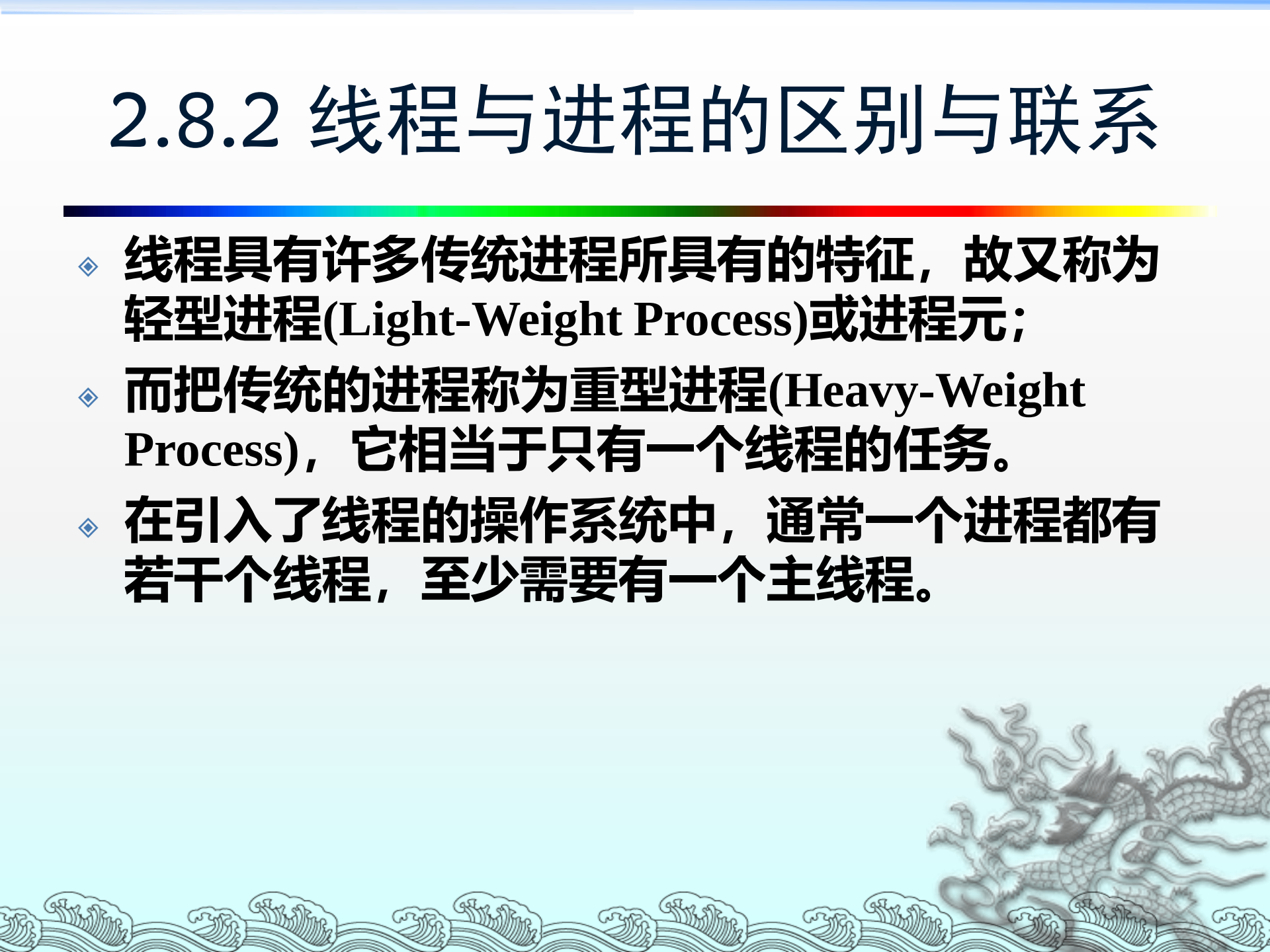
- 在操作系统中引入**进程**的目的，是为了使多个程序并发执行，以改善资源利用率及提高系统的吞吐量。
- 而在操作系统中再引入**线程**则是为了减少程序并发执行时所付出的时空开销，使操作系统并发粒度更小、并发性更好。

- **线程是操作系统进程中能够独立执行的实体（控制流），是处理器调度和分派的基本单位。**
- **线程是进程的组成部分，线程只拥有一些在运行中必不可少的资源（如程序计数器、一组寄存器和栈），与同属一个进程的其它线程共享进程所拥有的全部资源。**
- **一个线程可以创建和撤消另一个线程；同一进程中的多个线程之间可以并发执行。由于线程之间的相互制约，致使线程在运行中也呈现出间断性。**

- 线程有就绪、阻塞和执行三种基本状态，有的系统中线程还有终止状态。
- 挂起状态对线程是没有意义的，如果进程挂起后被对换出主存，则其所有线程因共享了进程的地址空间，也必须全部对换出去。
- 线程也有一一对应的描述和控制数据结构TCB（Thread Control Block）。不同进程间的线程互不可见，同一进程内的线程间通信主要基于全局变量。

2.8.2 线程与进程的区别与联系

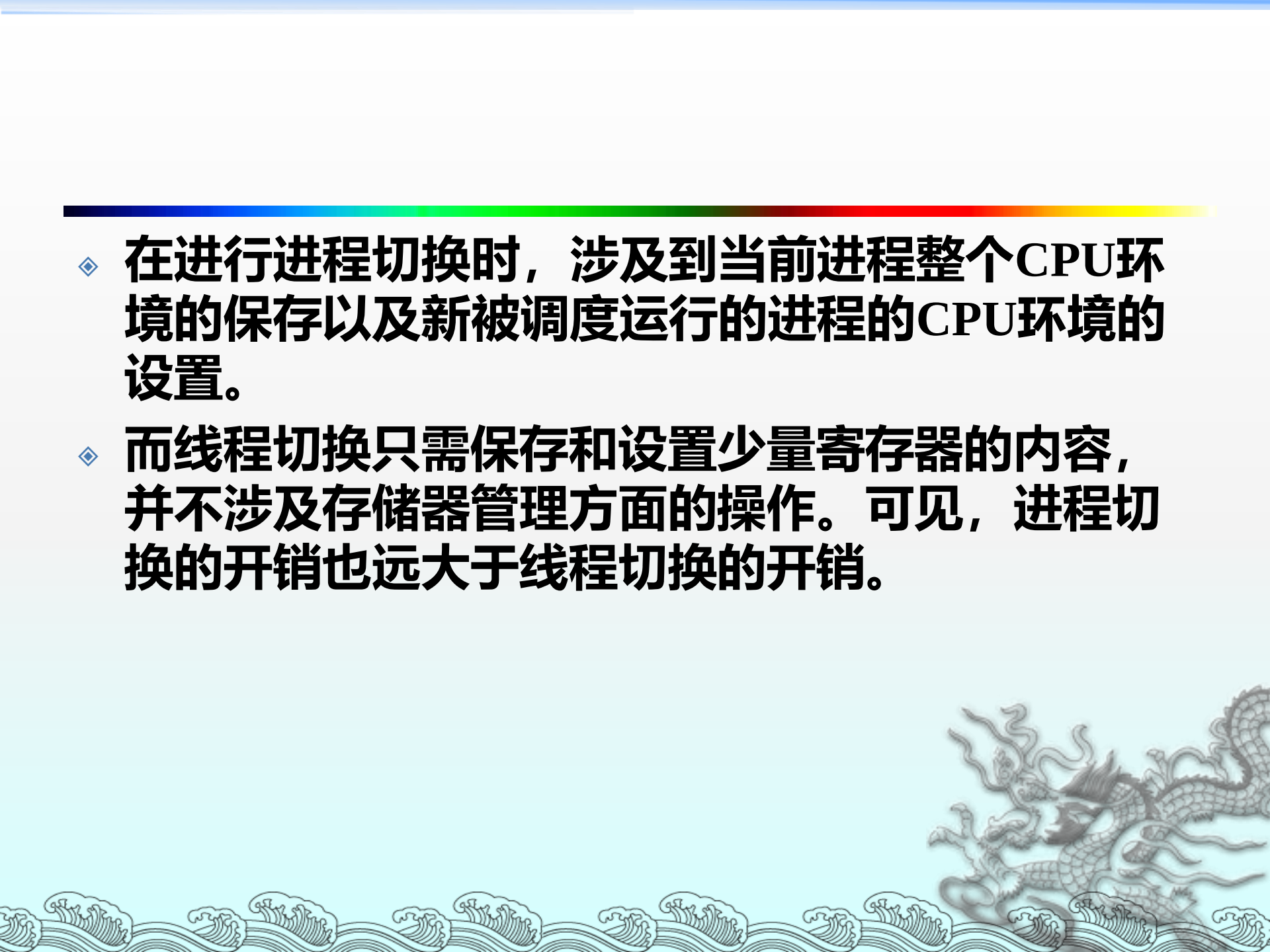
- 线程具有许多传统进程所具有的特征，故又称为轻型进程(Light-Weight Process)或进程元；
- 而把传统的进程称为重型进程(Heavy-Weight Process)，它相当于只有一个线程的任务。
- 在引入了线程的操作系统中，通常一个进程都有若干个线程，至少需要有一个主线程。



- 在传统的操作系统中，拥有资源的基本单位和独立调度的基本单位都是进程。而在引入线程的操作系统中，则把线程作为调度和分配的基本单位，而把进程作为拥有资源的基本单位，使传统进程的两个属性分开，线程便能轻装运行，从而可显著地提高系统的并发程度。
- 在同一进程中，线程的切换不会引起进程的切换，在由一个进程中的线程切换到另一个进程中的线程时，将会引起进程的切换。

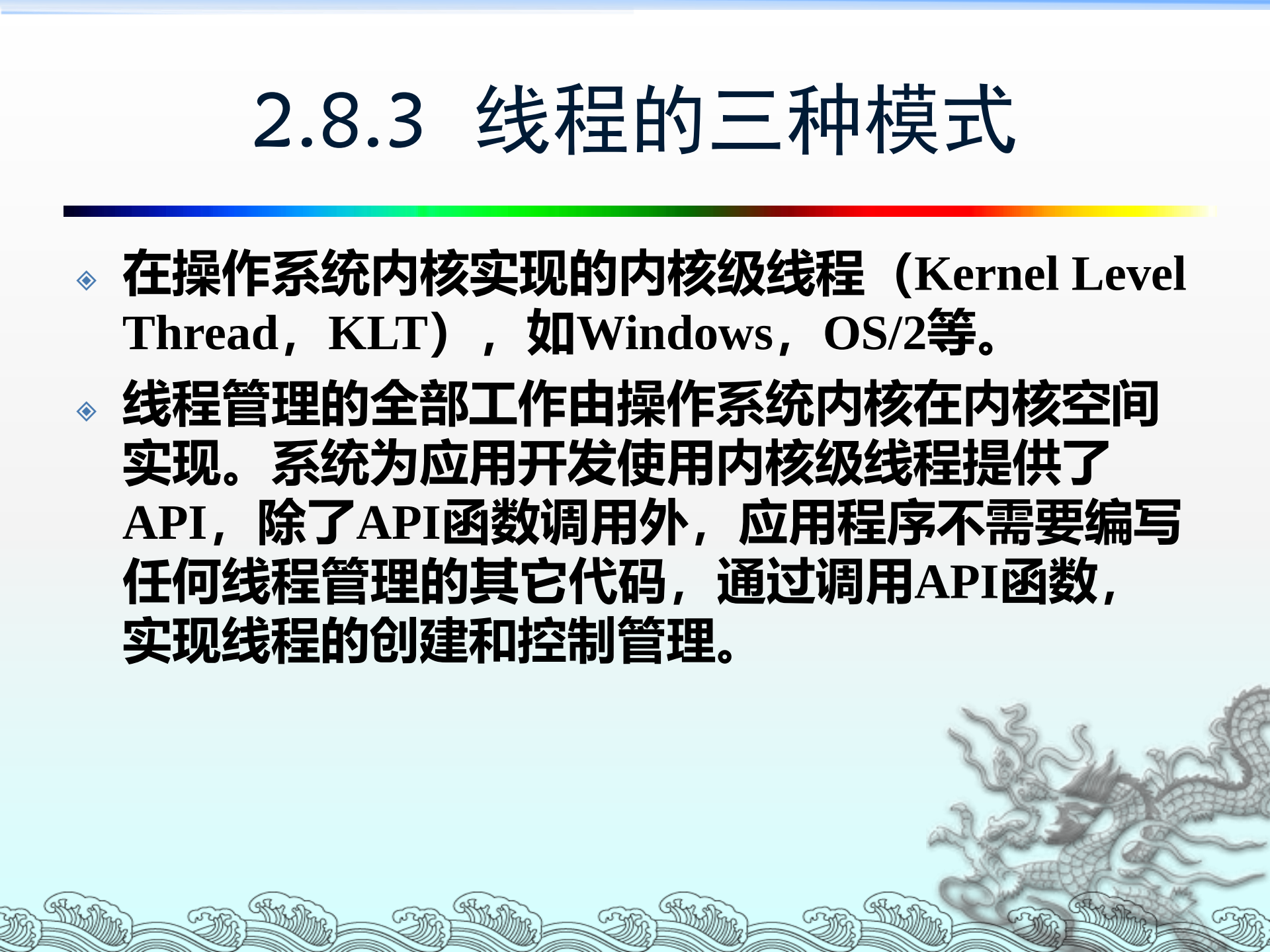
- 不论是传统的操作系统，还是支持线程的操作系统，进程都是拥有资源的独立单位，拥有自己的资源。
- 一般地说，线程自己不拥有系统资源(只有一些必不可少的资源)，但可以访问、共享其隶属进程的资源——进程的代码段、数据段以及打开的文件、I/O设备等。

- 在进行进程切换时，涉及到当前进程整个CPU环境的保存以及新被调度运行的进程的CPU环境的设置。
- 而线程切换只需保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作。可见，进程切换的开销也远大于线程切换的开销。



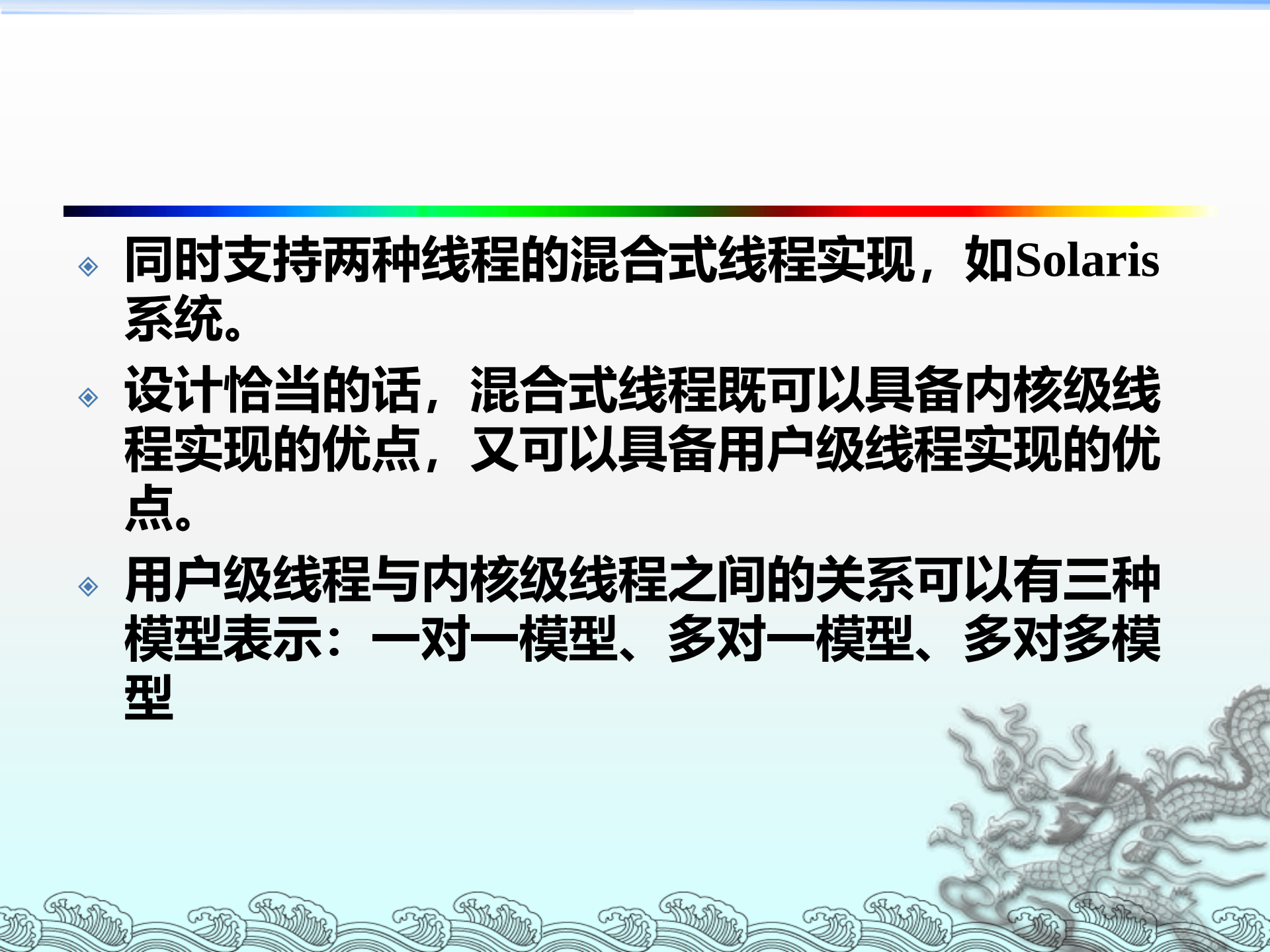
2.8.3 线程的三种模式

- 在操作系统内核实现的内核级线程（Kernel Level Thread，KLT），如Windows，OS/2等。
- 线程管理的全部工作由操作系统内核在内核空间实现。系统为应用开发使用内核级线程提供了API，除了API函数调用外，应用程序不需要编写任何线程管理的其它代码，通过调用API函数，实现线程的创建和控制管理。



- 在用户空间实现的用户级线程（User Level Thread，ULT），如Java的线程库等。
- 线程管理的全部工作由应用程序在用户空间实现，内核不知道线程的存在。用户级线程由用户空间运行的用户级线程库实现，应用开发通过线程库进行程序设计，用户级线程库是线程运行的支撑环境

- 同时支持两种线程的混合式线程实现，如Solaris系统。
- 设计恰当的话，混合式线程既可以具备内核级线程实现的优点，又可以具备用户级线程实现的优点。
- 用户级线程与内核级线程之间的关系可以有三种模型表示：一对一模型、多对一模型、多对多模型

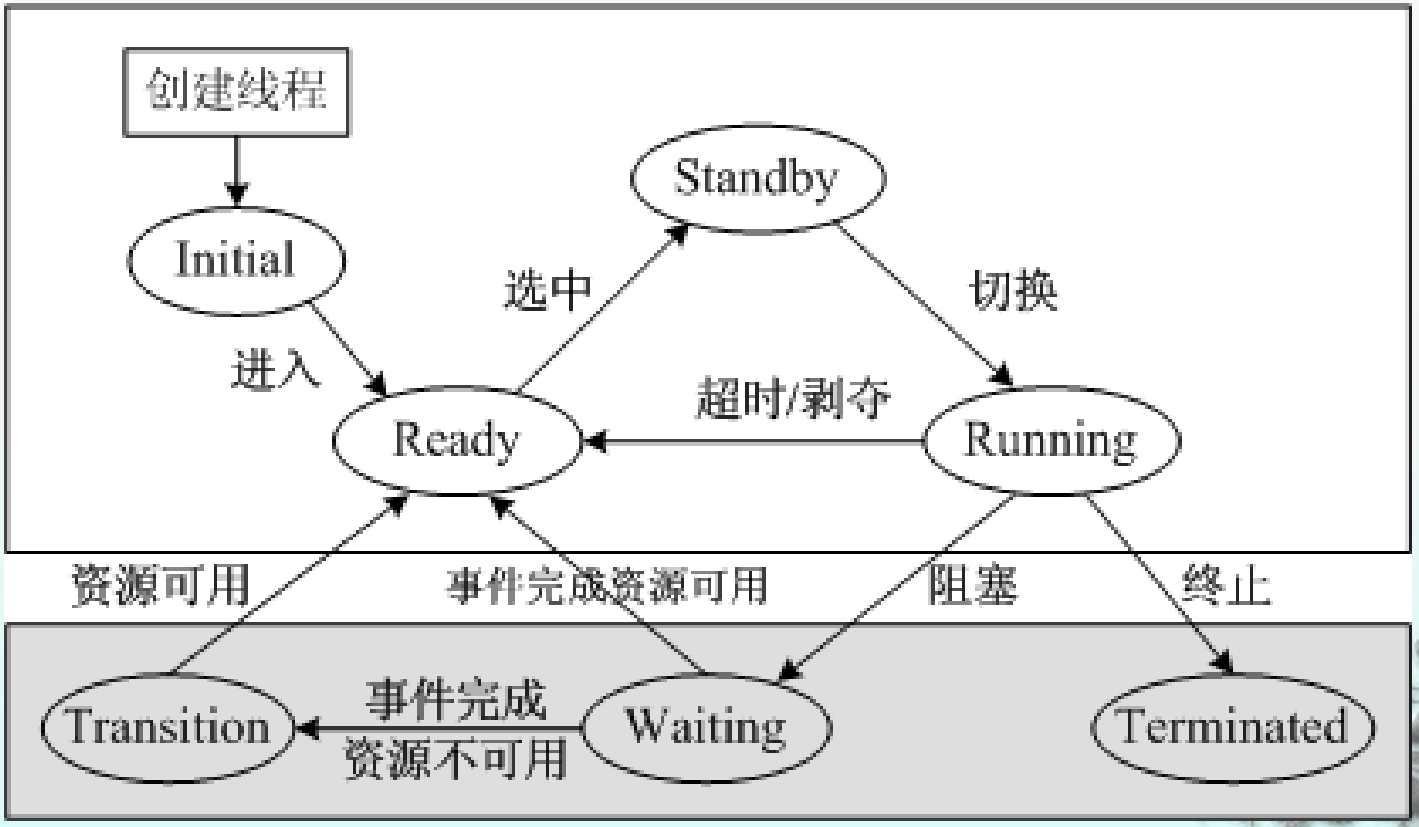


2.9 Windows 7中的进程与线程

- Windows系统中，进程被简单地划分为可执行和不可执行两种状态，线程是处理器调度的进步单位，状态包括以下七种：
- 初始状态（initial）：线程在创建过程中所处状态，创建完成后即进入就绪态。
- 就绪状态（ready）：线程已获得除处理器外的所需资源，正等待调度执行。
- 备用状态（standby）：已选择线程的执行处理器，正等待进入运行状态。每处理器上只能有一个线程处于该状态。

- 运行状态（running）：已完成描述表切换，线程进入运行状态，直至被抢先、时间片用完、线程终止或进入等待状态。
- 等待状态（waiting）：正等待某对象以同步线程的执行，待事件出现后，将根据优先级进入运行或就绪状态。
- 转换状态（transition）：与就绪状态相似，但线程的内核堆栈位于外存。
- 终止状态（terminated）：线程执行完毕进入该状态。

- Windows的线程状态转移如图2.21所示。



- 图2.21 Windows 线程状态及转换

2.9.1 进程

- Windows7的进程（process）是系统资源分配的基本单位，线程是被调度的运行实体。

- 在Windows中，进程和线程均是作为对象加以管理的，进程的属性包括进程标识、资源访问令牌、进程基本优先级和默认的亲和处理器集合（Processor Affinity）等，可通过其句柄（handle）加以引用。

- 1．内核对象
- Windows是个基于对象（object-based）的操作系统，用对象表示所有资源。在Windows操作系统中常见的有三种对象类型：
- Windows内核对象、 Windows GDI对象、Windows USER对象。

2．进程的创建与终止

- Windows系统调用函数CreateProcess创建进程的主要阶段：
- 打开将在进程中被执行的映像文件（.EXE）；
- 创建Windows内核进程对象；
- 创建初始线程（堆栈，环境和内核线程对象）；
- 向Win32子系统通知新进程以便它设置新的进程和线程；

- 启动初始线程的执行（除非CREATE-SUSPENDED标记被指定）；
- 在新进程和线程的环境中，完成地址空间的初始化（例如加载所需DLL）并开始程序的执行。

- 进程可以通过以下4种方式终止：
  - 主线程的入口点函数返回（强烈推荐的方式）；
  - 进程中的一个线程调用ExitProcess函数（要避免这种方式）；
  - 另一个进程中的线城调用TerminateProcess函数（要避免这种方式）；
  - 进程中的所有线程都“自然死亡”（这种情况几乎从来不会发生）。

- 当进程终止运行时系统会执行以下操作：
  - 终止进程中遗留的任何线程；
  - 释放进程分配的所有USER对象和GDI对象，关闭所有内核对象（如果使用计数减为0，销毁内核对象）；
  - 进程的退出代码从STILE_ACTIVE变为传给ExitProcess或TerminateProcess函数的代码；
  - 进程内核对象的状态变为已触发状态；
  - 进程内核对象的使用计数递减1。

- 3．进程间共享内核对象
  - 利用文件映射对象，可在同一台机器上运行的两个不同进程之间共享数据块；
  - 借助邮件槽和命名管道，网络中的不同计算机上的进程可以相互发送数据块；
  - 互斥量，信号量和事件允许不同进程中的线程同步执行。
- 有三种不同机制允许进程共享对象：使用对象句柄继承，为对象命名和复制对象句柄。

2.9.2 线程

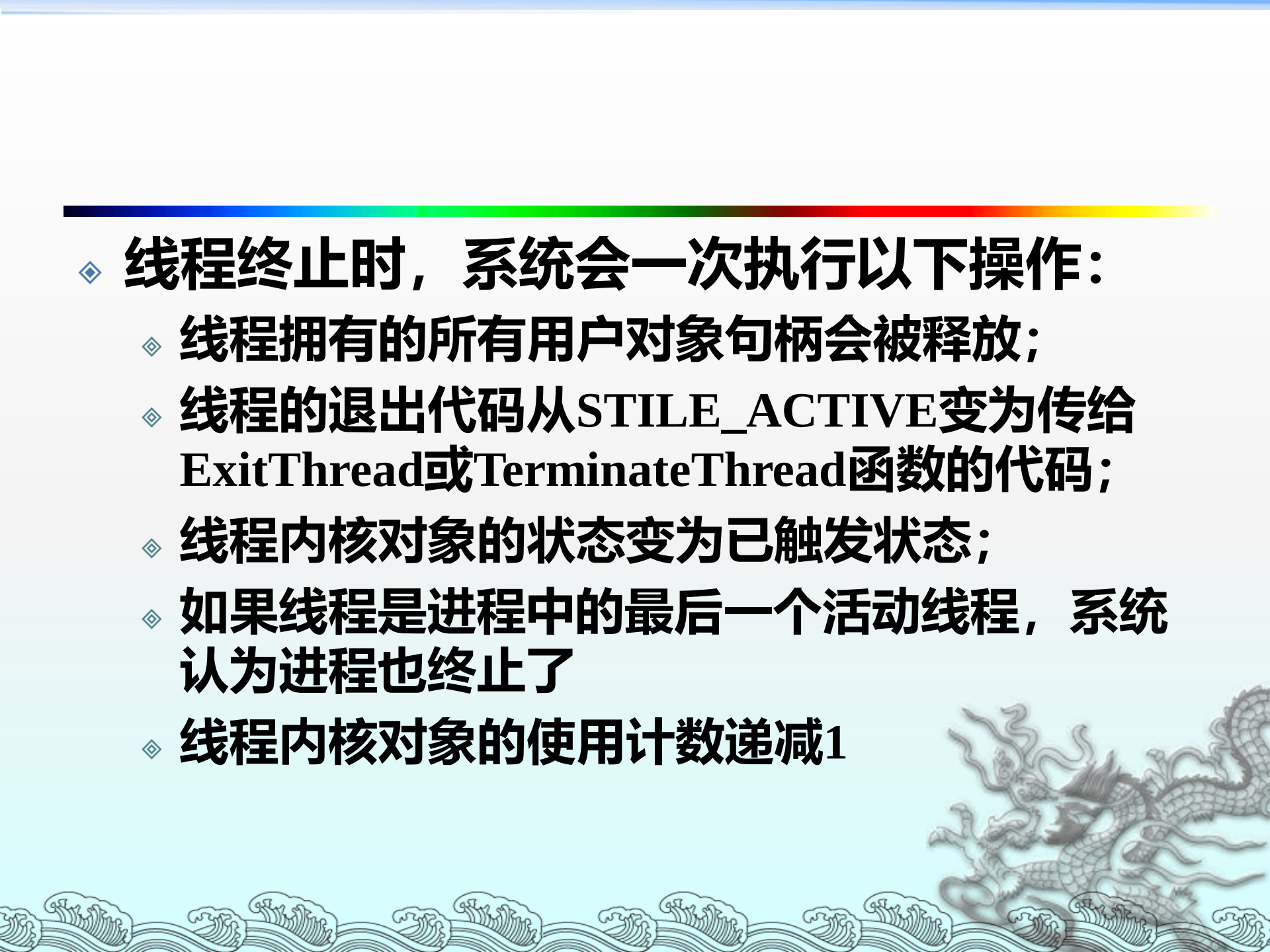
- Windows NT是基于内核线程模型的，所有线程都由内核管理调度，这个调度由内核的分发器（Dispatcher，也称调度器Scheduler）的模块完成进行。
- Windows NT一直提供对用户级线程（UMS）的支持，它被称为纤程（Fiber），纤程是比线程更细小的运行实体，一个线程可以分成多个纤程来运行。

- 1．线程的创建与终止
- Windows提供了一组系统API用于线程控制（如CreateThread、SuspendThread、ResumeThread、ExitThread、TerminateThread、SetThreadPriority等）。
- 创建线程时，应用程序调用CreateThread，CreateThread 函数的一个调用 导致 系统创建一个线程内核对象，该对象最初的使用计数为2。（创建线程内核对象加1，返回线程内核对象句柄加1

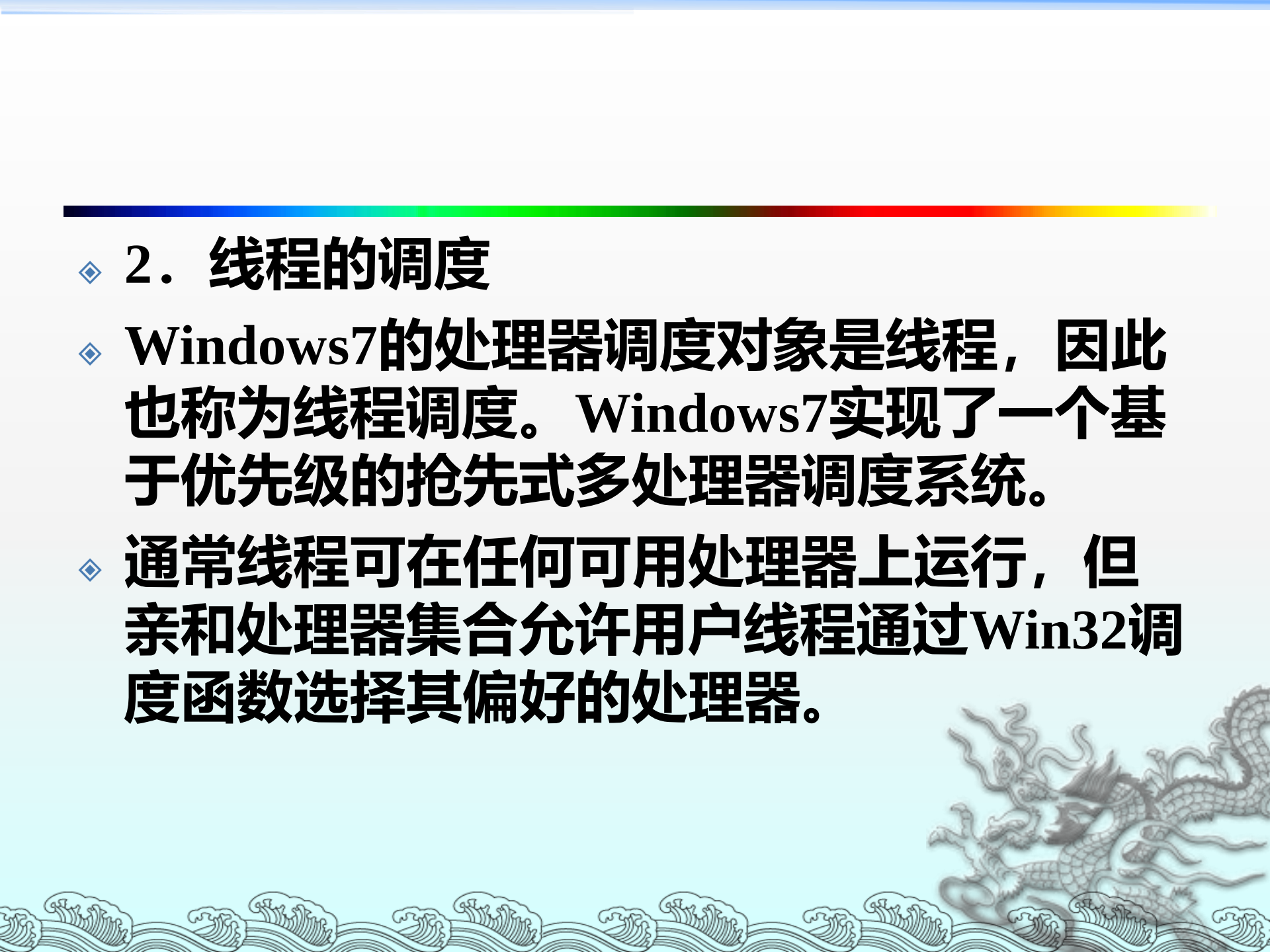
- 创建了内核对象，系统就分配内存，供线程的堆栈使用。此内存是从进程的地址空间分配的，因为线程没有自己的地址空间。
- 每个线程都有自己的一组CPU寄存器，称为线程的上下文（context），线程上下文主要包括寄存器、线程环境块、核心栈和用户栈。上下文反映了当线程上一次执行时，线程CPU寄存器的状态。Context结构保存在线程的内核对象中。

- 线程可以通过以下4种方法来终止运行：
  - 线程函数返回（推荐）；
  - 线程通过ExitThread 函数“杀死”自己（不推荐）；
  - 同一个进程或另一个进程中的线程调用TerminateThread 函数（不推荐）；
  - 包含线程的进程终止运行（不推荐）。

- 线程终止时，系统会一次执行以下操作：
  - 线程拥有的所有用户对象句柄会被释放；
  - 线程的退出代码从STILE_ACTIVE变为传给ExitThread或TerminateThread函数的代码；
  - 线程内核对象的状态变为已触发状态；
  - 如果线程是进程中的最后一个活动线程，系统认为进程也终止了
  - 线程内核对象的使用计数递减1



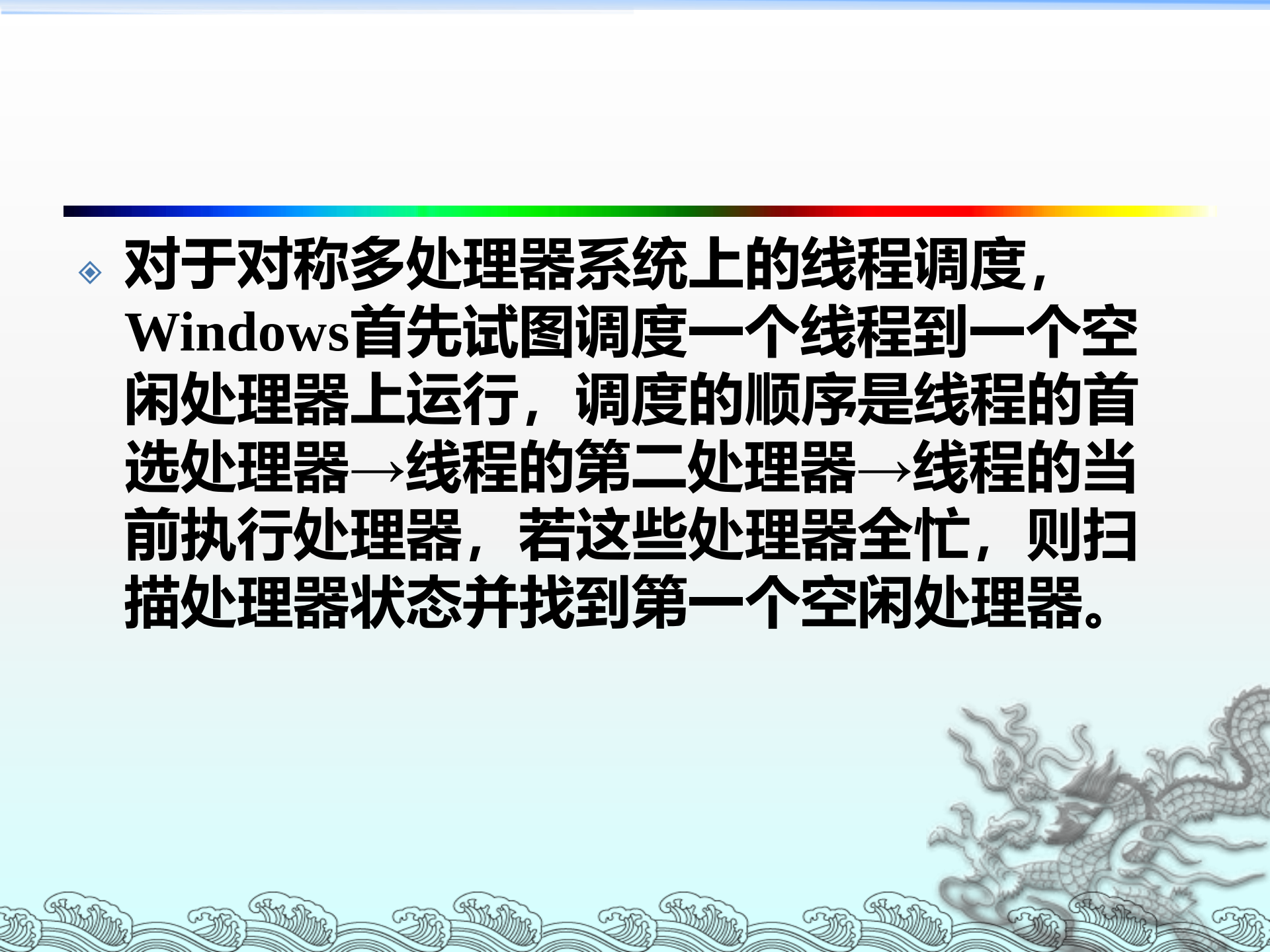
- 2．线程的调度
- Windows7的处理器调度对象是线程，因此也称为线程调度。Windows7实现了一个基于优先级的抢先式多处理器调度系统。
- 通常线程可在任何可用处理器上运行，但亲和处理器集合允许用户线程通过Win32调度函数选择其偏好的处理器。



- 线程调度的触发事件包括：
  - 一个线程进入就绪状态；
  - 一个线程由于时间配额使用完毕而从运行状态转入退出或等待状态；
  - 一个线程由于调用系统服务而改变优先级或被操作系统本身调整其优先级；
  - 一个正在运行的线程改变了其亲和处理器集合。

- Windows 7在单处理器系统中的线程调度策略：
  - 主动切换：一个线程可能因进入等待状态而主动放弃处理器的使用。
  - 抢先：当一个高优先级线程进入就绪状态时，正处于运行状态的低优先级线程被抢先。当抢先线程完成运行后，被抢先的线程可继续使用剩余的时间配额。
  - 时间配额用完：当一个处于运行状态的线程用完其时间配额时，Windows首先确定是否需要降低其优先级，然后确定是否需要调度另一个线程进入运行状态。
  - 结束：当线程完成运行时，转移到终止状态。

- 对于对称多处理器系统上的线程调度，Windows首先试图调度一个线程到一个空闲处理器上运行，调度的顺序是线程的首选处理器→线程的第二处理器→线程的当前执行处理器，若这些处理器全忙，则扫描处理器状态并找到第一个空闲处理器。



- 若线程进入就绪状态时所有处理器均忙，Windows将检查其是否可抢先执行，检查的顺序为线程的首选处理器→线程的第二处理器，若这两个处理器均不在线程的亲和掩码（Affinity mask）中，选择该线程可运行的最大编号的处理器。对于选中的处理器，Windows仅对运行的线程和备用线程检查优先级以确定能否抢先。根据这样的调度机制，可能出现最高优先级就绪线程不处于运行状态的情况。

小结

- 本章引入进程、线程、管程、并发、调度、同步、死锁等概念。介绍了进程状态与转换，进程调度算法、进程同步机制与通信机制、银行家算法等经典理论，最后介绍了Windows7的进程与线程管理。
- 本章是操作系统课程的核心和难点所在。