











内容导航:

-  3.1 处理机调度概述
-  3.2 调度算法
-  **3.3 实时调度**
-  3.4 Linux进程调度
-  3.5 死锁概述
-  3.6 预防死锁
-  3.7 避免死锁
-  3.8 死锁的检测与解除

第3章 处理机调度与死锁

实时调度是针对实时任务的调度

实时任务，都联系着一个截止时间

- 硬实时HRT任务
- 软实时SRT任务

实时调度应具备一定的条件



1. 提供必要的信息（向调度程序提供）

- (1) 就绪时间（任务转为就绪状态的起始时间）
- (2) 开始截止时间和完成截止时间（某个时间点必须开始、必须结束）
- (3) 处理时间（执行时间）
- (4) 资源要求（任务执行时所需的一组资源）
- (5) 优先级

2. 系统处理能力强

- 实时系统中通常都有着多个实时任务
- 若处理机的处理能力不够强，则有可能因处理机忙不过来而使某些实时任务不能得到及时处理，从而导致发生难以预料的后果
- 假定系统中有 m 个周期性的硬实时任务，它们的处理时间可表示为 C_i ，周期时间表示为 P_i ，则在单处理机情况下，必须满足下面的限制条件：

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1 \quad (\text{可调度})$$

2. 系统处理能力强

一个软实时系统处理三个事件流，其周期分别为 100 ms, 200 ms 和500 ms。如果事件处理时间分别为 50 ms, 30 ms 和100 ms, 因为 $0.5+0.15+0.2<1$ 故此系统是可调度的。

如果加入周期为1 s的第四个事件，则只要其处理时间不超过 150ms, 该系统仍将是可调度的。

这个运算的隐含条件是上下文切换的开销很小，可以忽略。

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

3. 采用抢占式调度机制

- 当一个优先权更高的任务到达时，允许将当前任务暂时挂起，而令高优先权任务立即投入运行，这样便可满足该硬实时任务对截止时间的要求。

这种调度机制比较复杂

- 对于一些小的实时系统，如果能预知任务的开始截止时间，则对实时任务的调度可采用非抢占调度机制，简化调度程序和对任务调度时所花费的系统开销。

但在设计这种调度机制时，应使所有的实时任务都比较小，并在执行完关键性程序和临界区后，能及时地将自己阻塞起来，以便释放出处理机，供调度程序去调度那种开始截止时间即将到达的任务

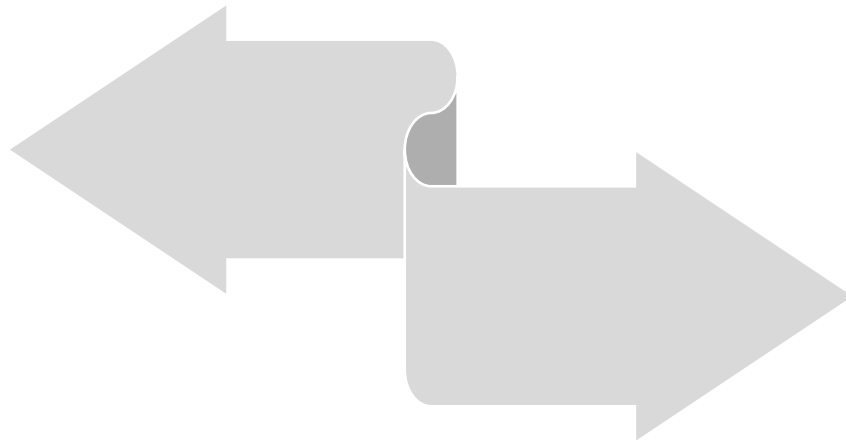
4. 具有快速切换机制

- **对中断的快速响应能力**。对紧迫的外部事件请求中断能及时响应，要求系统具有快速硬件中断机构，还应使禁止中断的时间间隔尽量短，以免耽误时机(其它紧迫任务)
- **快速的任務分派能力**。为了提高分派程序进行任务切换时的速度，应使系统中的每个运行功能单位适当的小，以减少任务切换的时间开销



根据实时任务性质

- HRT调度算法
- SRT调度算法



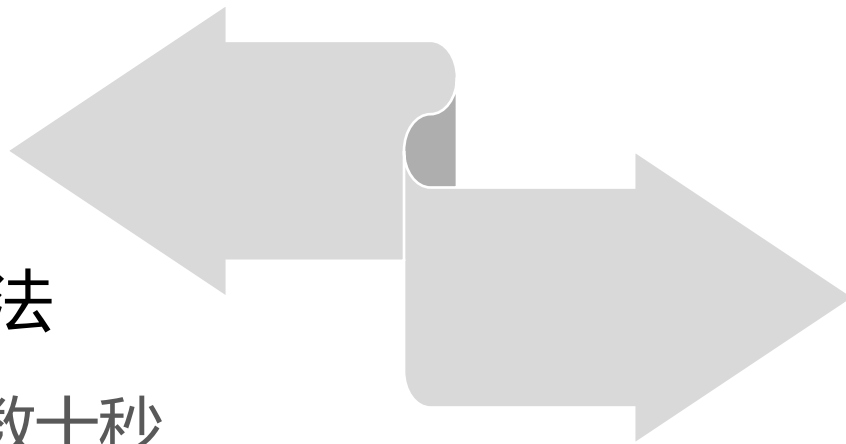
根据调度方式

- 非抢占式调度算法
- 抢占式调度算法



非抢占式轮转调度算法

- 响应时间：数秒至数十秒
- 可用于要求不太严格的实时控制系统



抢占式优先调度算法

- 响应时间：数秒至数百毫秒
- 可用于有一定要求的实时控制系统



基于时钟中断的抢占式优先级调度

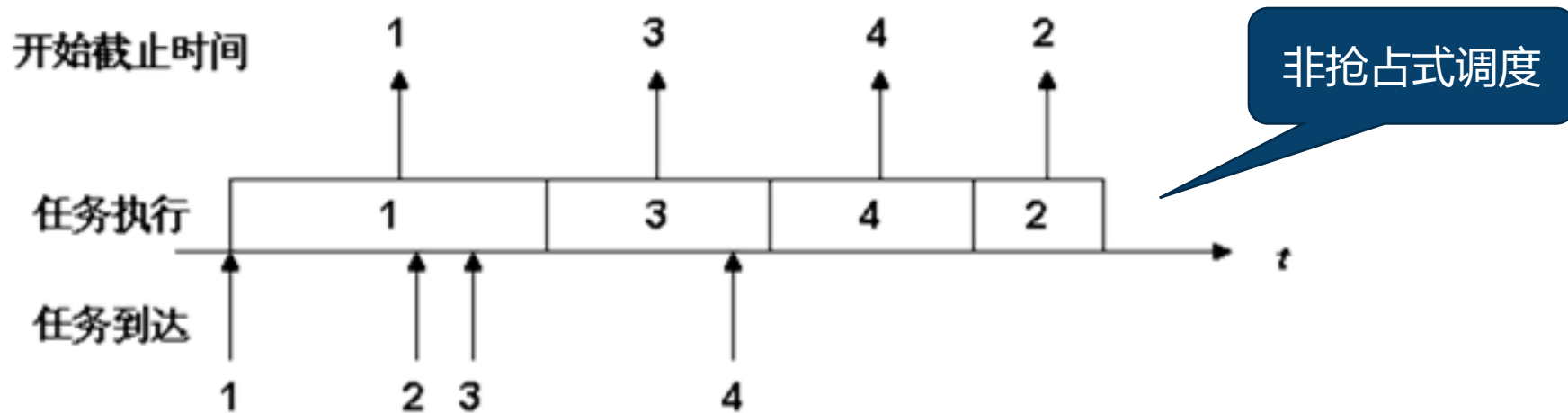
- 响应时间：几十毫秒至几毫秒
- 可用于大多数实时系统



立即抢占的优先级调度

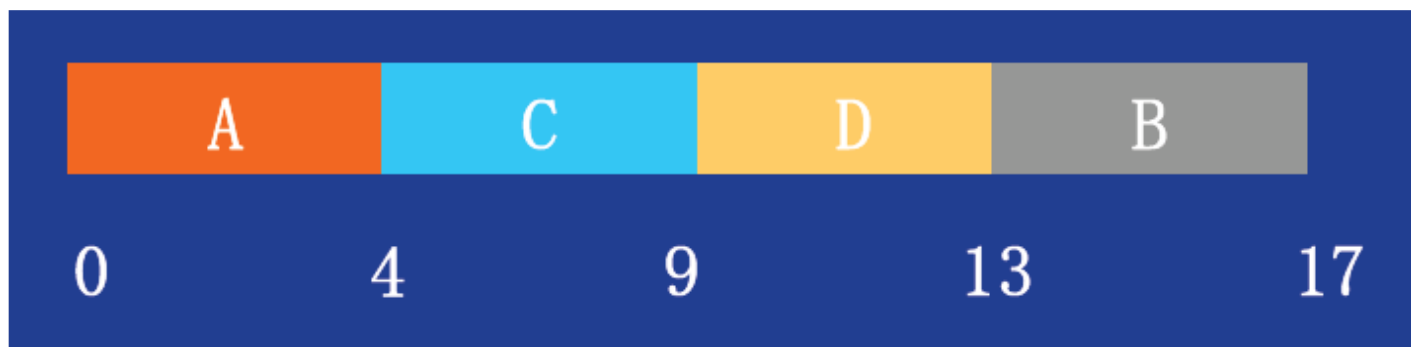
- 响应时间：几毫秒至几百微秒
- 可用于有严格时间要求的实时系统

- OS EDF根据任务的截止时间确定优先级，截止时间越早，优先级越高
- OS 既可用于抢占式调度，也可用于非抢占式调度
- OS 非抢占式调度用于非周期实时任务
- OS 抢占式调度用于周期实时任务



非抢占式调度方式用于非周期实时任务

任务	到达时间	开始截止时间	执行时间
A	0	2	4
B	2	15	5
C	3	6	5
D	6	10	4

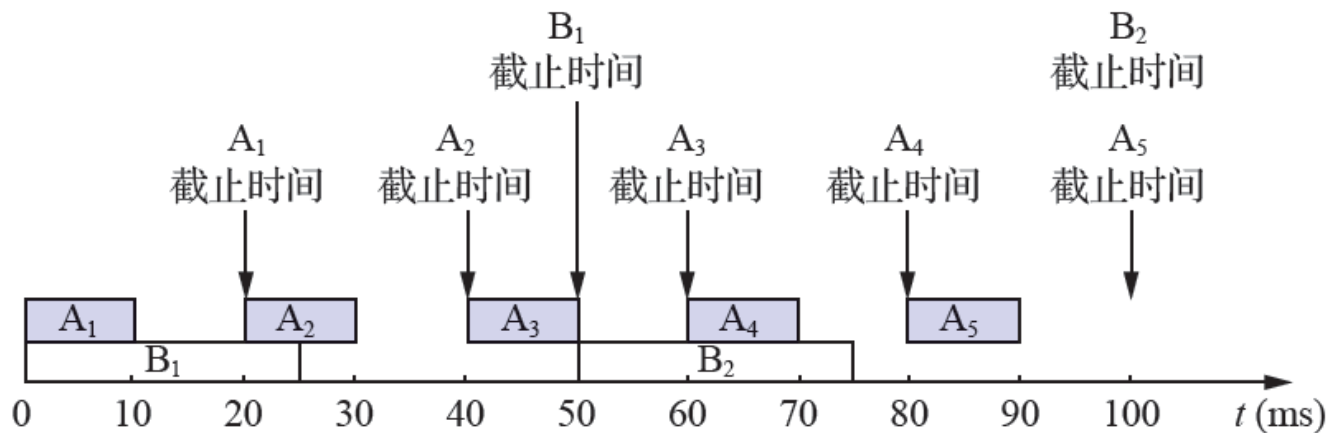




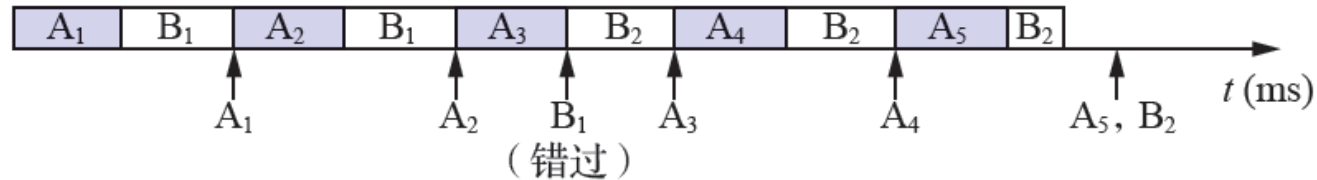
抢占式最早截止时间优先（周期实时任务）

	周期时间	执行时间
A	20 ms	10 ms
B	50 ms	25 ms

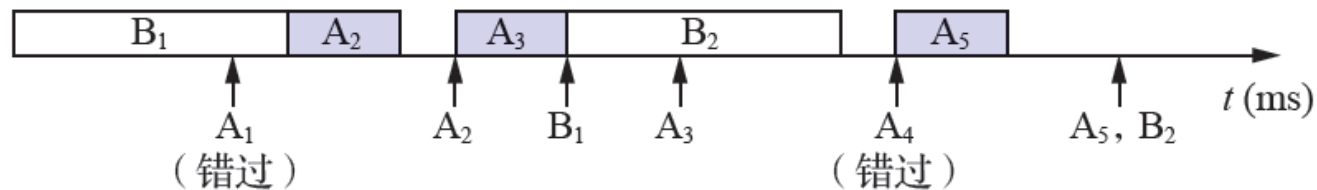
(条件) 到达时间、执行时间和最后截止时间



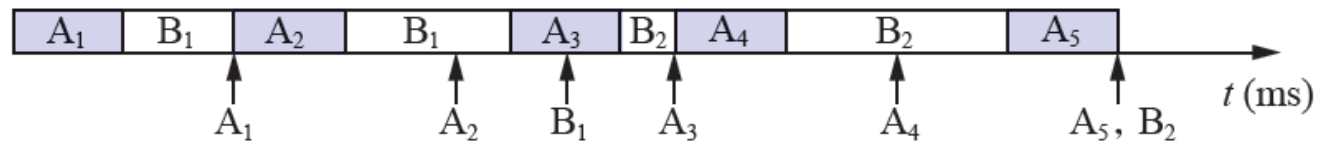
($A > B$) 固定优先级调度



($B > A$) 固定优先级调度



(EDF) 使用完成截止时间最早和最后截止时间调度





根据任务的紧急程度（**松弛度**）确定任务优先级

- 紧急程度越高（松弛度越低），优先级越高
- 松弛度slack time=必须完成时间 - 其本身的运行时间 - 当前时间

如果已经运行一段时间

- 松弛度=必须完成时间 - 剩余运行时间 - 当前时间

剩余运行时间 = 其本身的运行时间 - 已经运行多少时间



主要用在抢占式调度方式中

1. 该算法主要用于可抢占调度方式中，**当一任务的最低松弛度减为0时，它必须立即抢占CPU**，以保证按截止时间的要求完成任务。
2. 计算关键时间点的各进程周期的松弛度，当进程在当前周期截止时间前完成了任务，**则在该进程进入下个周期前，无需计算它的松弛度。**
3. 当出现多个进程松弛度相同且为最小时，按照“**最近最久未调度**”（Least Recently Used，LRU）的原则进行进程调度。



根据任务的紧急程度（**松弛度**）确定任务优先级

- 紧急程度越高（松弛度越低），优先级越高
- $\text{松弛度} = \text{必须完成时间} - \text{其本身的运行时间} - \text{当前时间}$



主要用在抢占式调度方式中



例子：

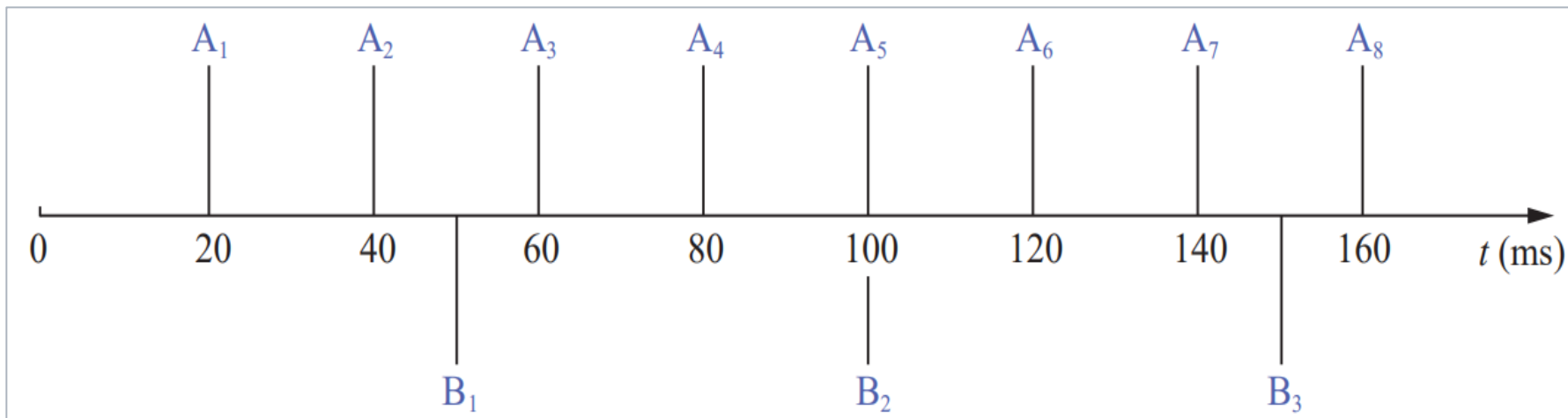
- 两个周期性实时任务A和B，任务A要求每20 ms执行一次，执行时间为10 ms，任务B要求每50 ms执行一次，执行时间为25 ms



最低松弛度优先 (周期实时任务)

	周期时间	执行时间
A	20 ms	10 ms
B	50 ms	25 ms

(条件)



t1 (0 ms)

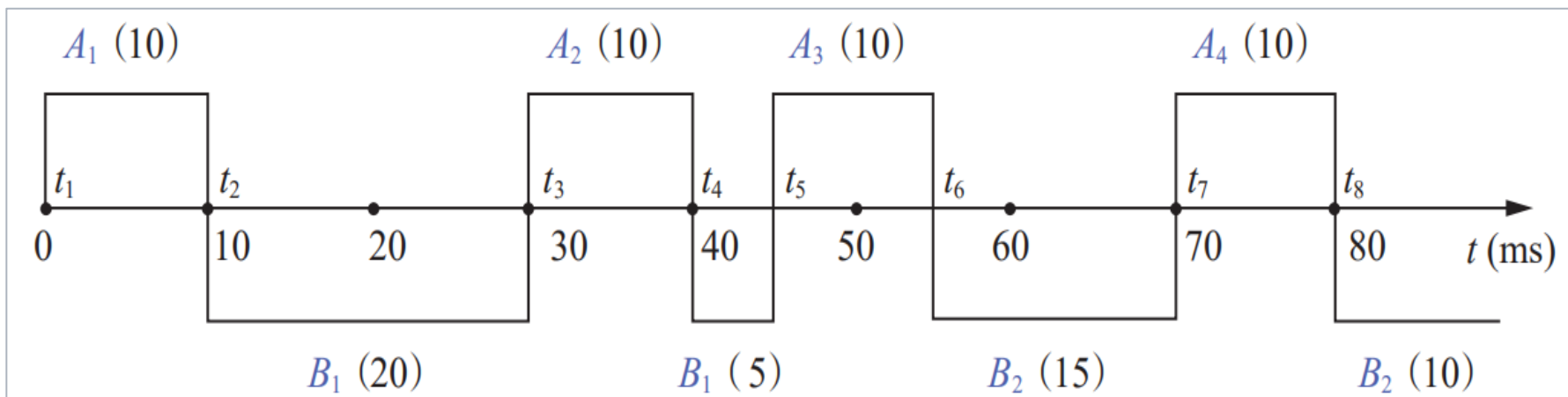
A1: $20 - 10 - 0 = 10$ (小)

B1: $50 - 25 - 0 = 25$

t2 (10 ms)

A2: $40 - 10 - 10 = 20$

B1: $50 - 25 - 10 = 15$ (小)



t2.5 (20 ms)

A2: $40 - 10 - 20 = 10$ (小, 但不抢占, 什么时候抢占?)

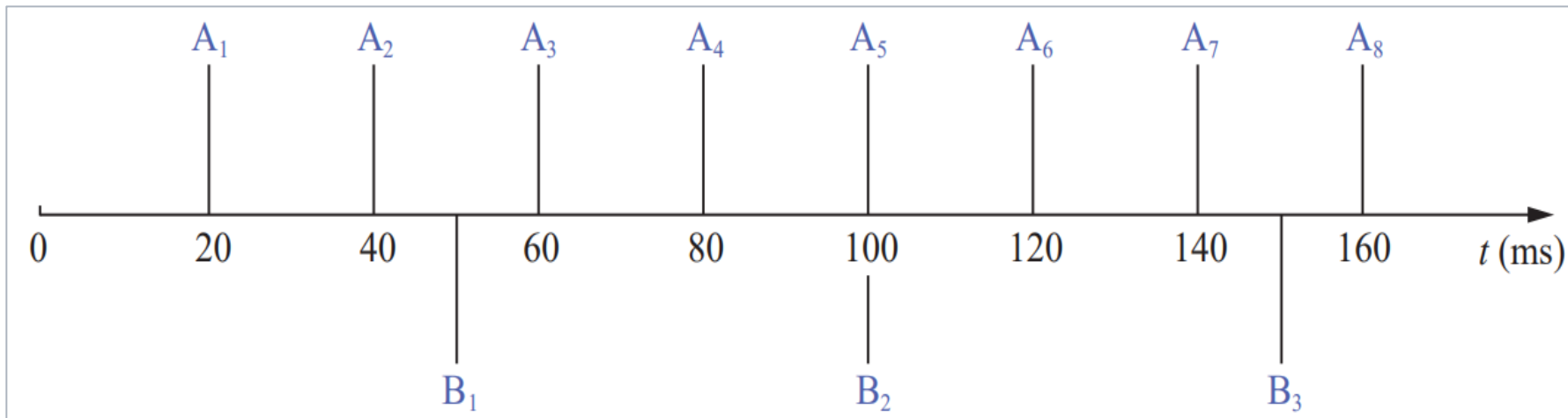
B1: $50 - (25 - 10) - 20 = 15$



最低松弛度优先（周期实时任务）

	周期时间	执行时间
A	20 ms	10 ms
B	50 ms	25 ms

(条件)



t₃ (30 ms)

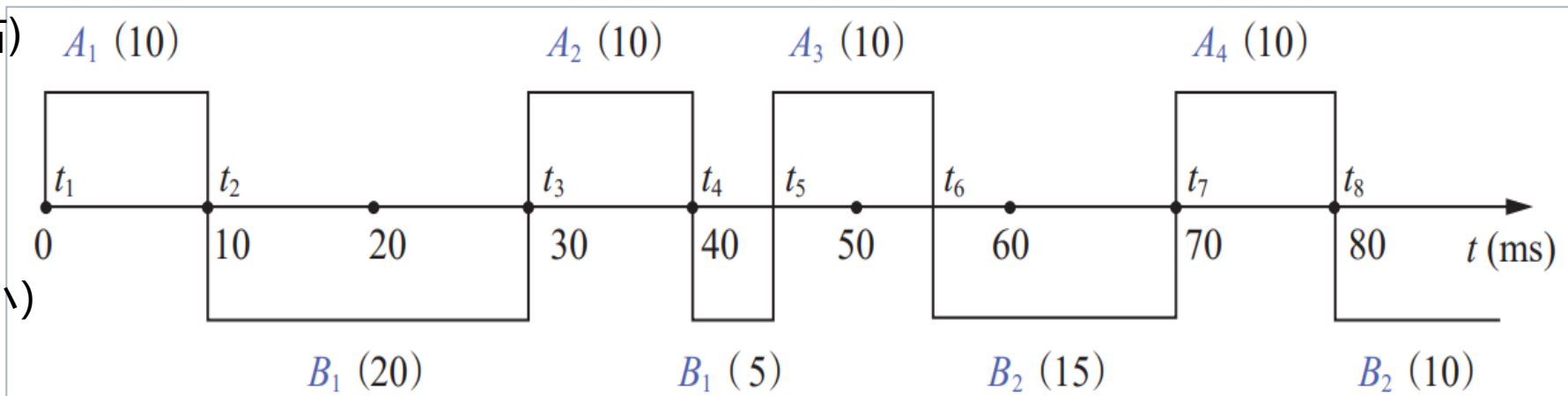
A₂: 40 - 10 - 30 = 0 (0, 抢占)

B₁: 50 - (25-20) - 30 = 15

t₄ (40 ms)

A₃: 60 - 10 - 40 = 10

B₁: 50 - (25-20) - 40 = 5 (小)



t₅ (45 ms)

A₃: 60 - 10 - 45 = 5 (小)

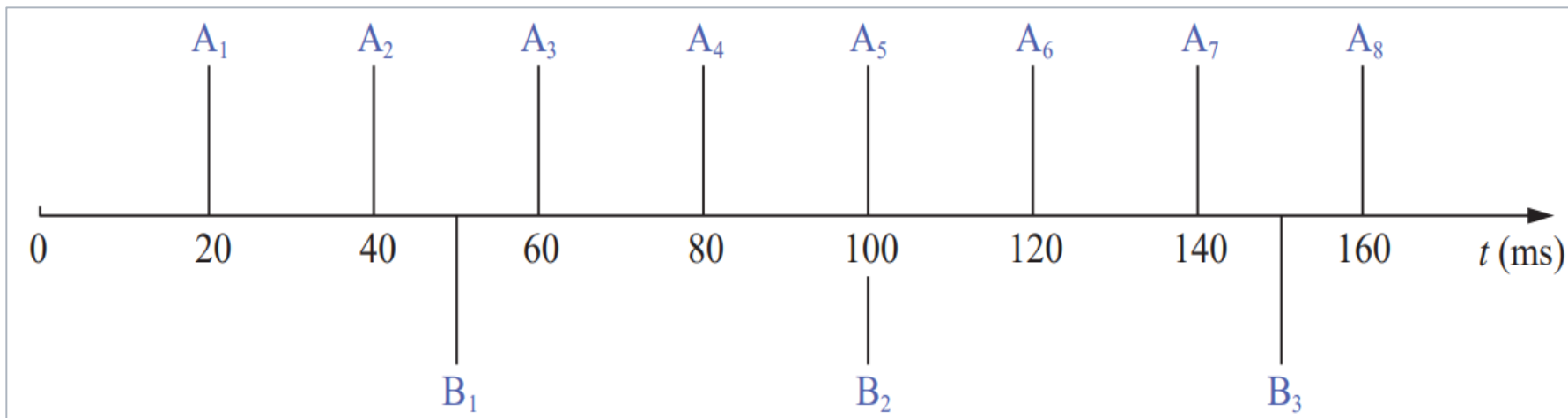
B₂: 100 - 25 - 45 = 30



最低松弛度优先（周期实时任务）

	周期时间	执行时间
A	20 ms	10 ms
B	50 ms	25 ms

(条件)



t₆ (55 ms)

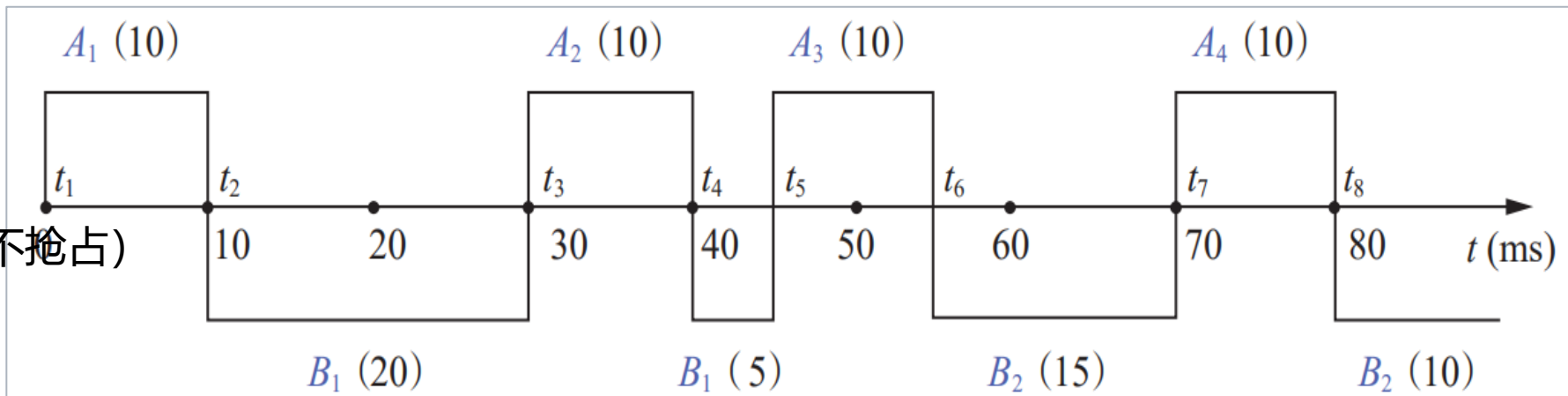
A₄:

B₂: $100 - 25 - 55 = 20$

t_{6.5} (60 ms)

A₄: $80 - 10 - 60 = 10$ (小, 不抢占)

B₂: $100 - (25 - 5) - 60 = 20$



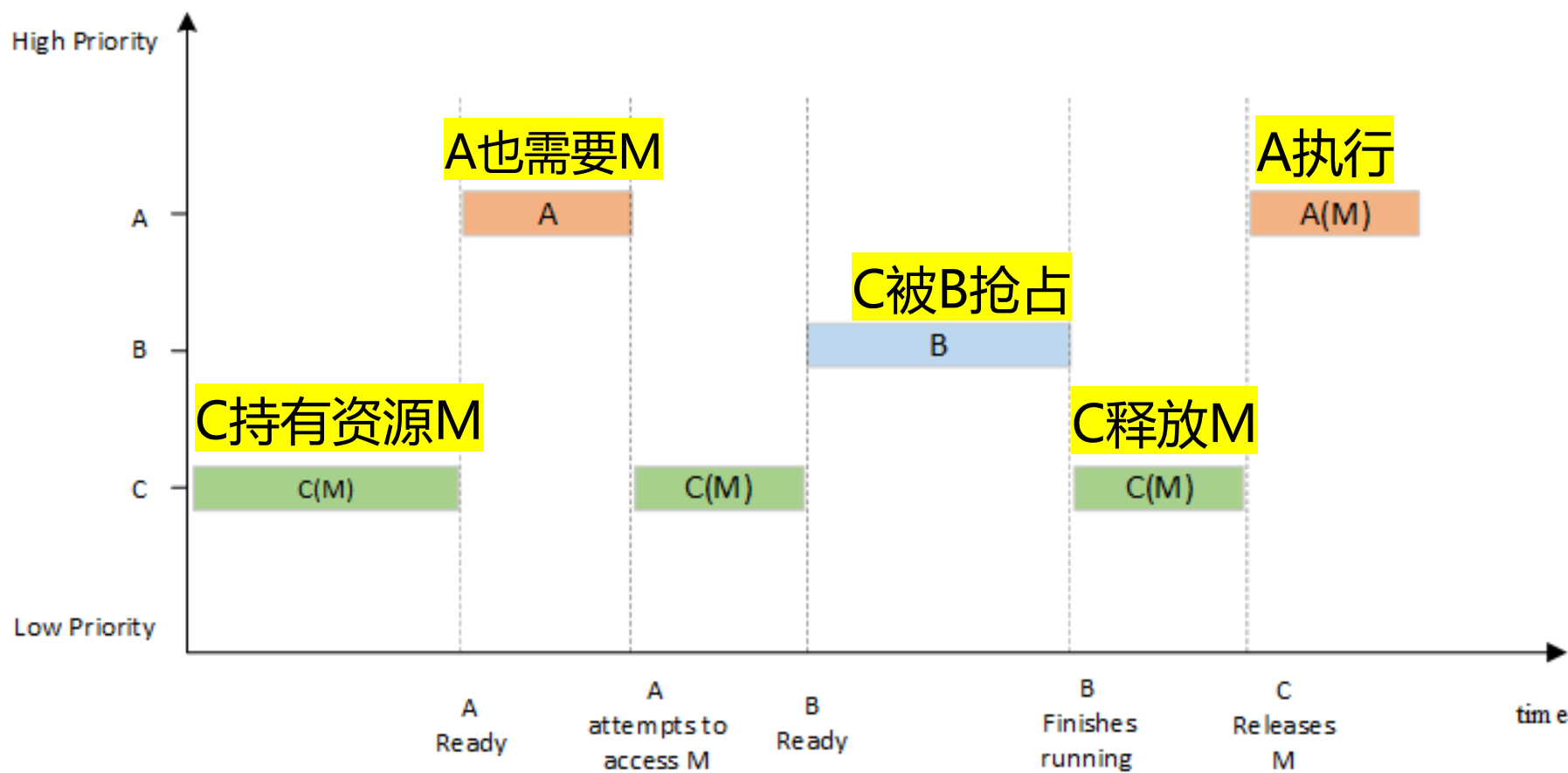
t₇ (70 ms)

A₄: $80 - 10 - 70 = 0$ (立即抢占)


B₂: $100 - (25 - 15) - 70 = 20$



采用优先级调度和抢占方式，可能产生**优先级倒置**。现象：高优先级进程被低优先级进程延迟或阻塞。



C/B 的
优先级
比 A 低，
但却被
优先执
行

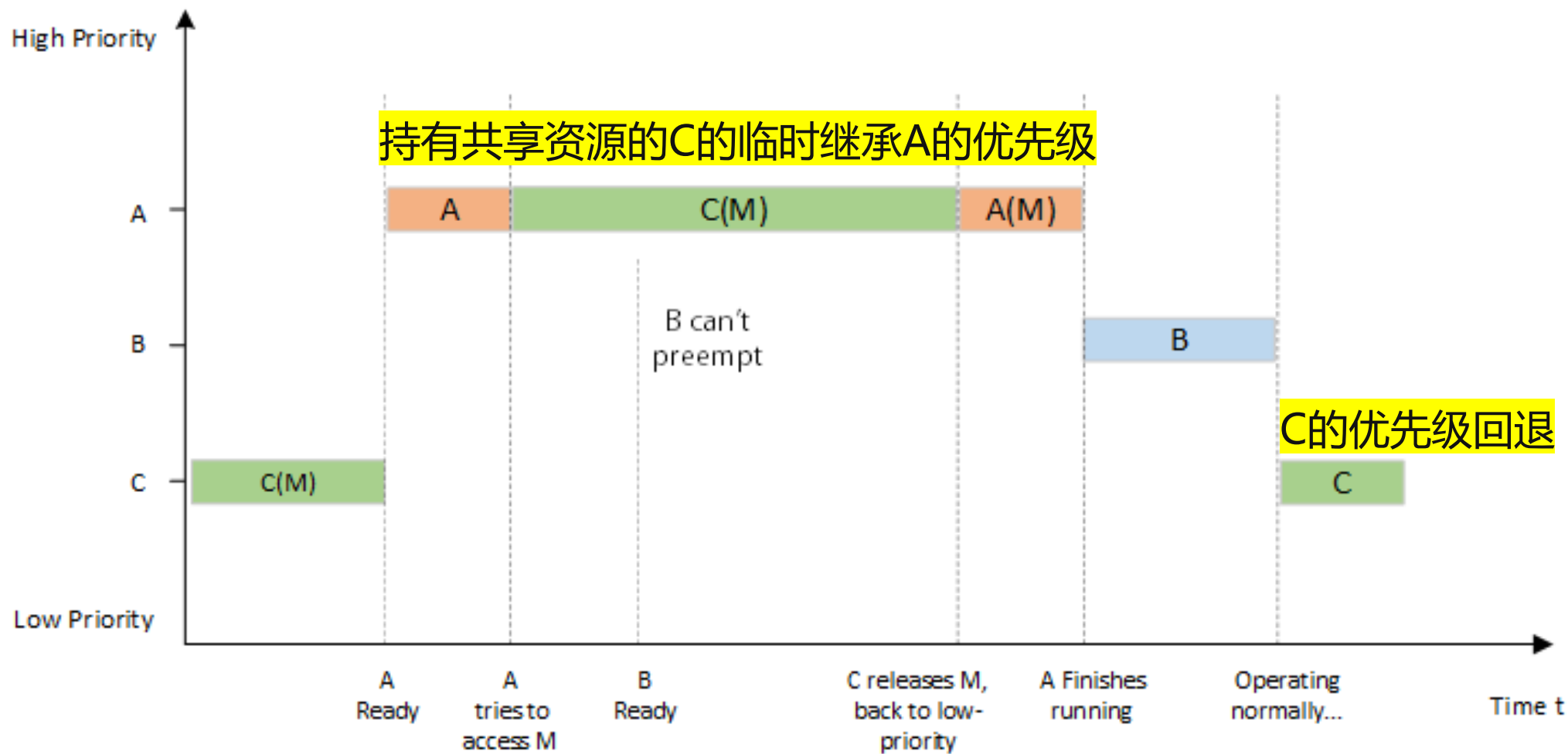
 采用优先级调度和抢占方式，可能产生**优先级倒置**。现象：高优先级进程被低优先级进程延迟或阻塞。

 **解决方法：**

- 制定一些规定，如规定低优先级进程执行后，其所占用的处理机不允许被抢占（问题：高优先级进程可能会等待很长时间）；
- 建立动态**优先级继承**。








优先级继承 (Priority Inheritance)





内容导航:

-  3.1 处理机调度概述
-  3.2 调度算法
-  3.3 实时调度
-  **3.4 Linux进程调度**
-  3.5 死锁概述
-  3.6 预防死锁
-  3.7 避免死锁
-  3.8 死锁的检测与解除

第3章 处理机调度与死锁

在 CPU 的角度看进程行为的话，可以分为两类：

- **CPU 消耗型**：此类进程就是一直占用 CPU 计算，CPU 利用率很高
- **IO 消耗型**：此类进程会涉及到 IO，需要和用户交互，比如键盘输入，占用 CPU 不是很高，只需要 CPU 的一部分计算，大多数时间是在等待 IO

CPU 消耗型进程需要高的吞吐率，IO 消耗型进程需要强的响应性，这两点都是调度器需要考虑的。

为了更快响应 IO 消耗型进程，内核提供了一个抢占(preempt)机制，使优先级更高的进程，去抢占优先级低的进程运行。

- 2.5版本之前，Linux采用传统的UNIX调度算法。由于没有考虑到SMP，所以SMP不支持。性能表现就非常不好
- 2.5系列内核中，调度程序做了大手术，采用了O(1)调度程序，引入了静态时间片算法和每一处理器的运行队列（运行时间是常量，与系统的任务数量是无关的）。O(1)调度器在多处理器环境下表现近乎完美，但对于响应时间敏感的程序有先天不足，如交互进程（interactive processes）
- 2.6开发初期引入了“反转楼梯最后期限调度算法”（RSDL），吸取了队列理论，将公平调度的概念引入Linux，最终在2.6.23替代了O(1)调度算法，并有了一个新名字，“完全公平调度算法”（CFS）

基于调度器类：允许不同的可动态添加的调度算法并存，每个类都有一个特定的优先级。



总调度器：根据调度器类的优先顺序，依次对调度器类中的进程进行调度。



调度器类：使用所选的调度器类算法（调度策略）进行内部的调度。



调度器类的默认优先级顺序为：Stop_Task > Real_Time > Fair > Idle_Task



Linux进程调度器

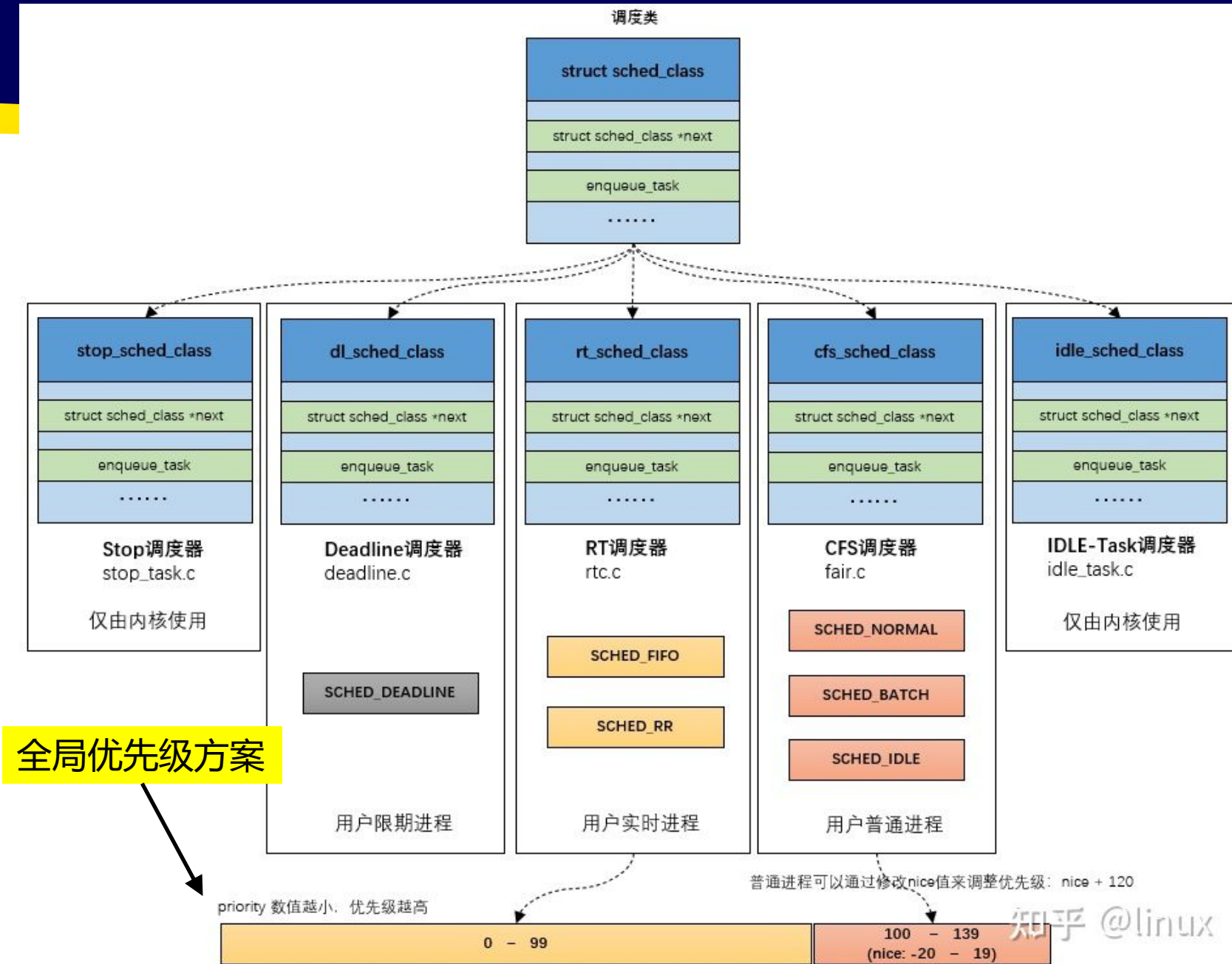
1.Stop调度器：优先级最高的调度类，可以抢占其他所有进程，不能被其他进程抢占

2.Deadline调度器：把进程按照绝对截止期限进行排序，选择最小进程进行调度

3.RT调度器：为每个优先级维护一个队列；

4.CFS调度器：采用CFS算法，引入虚拟运行时间概念

5.IDLE-Task调度器：每个CPU都会有一个idle线程，当没有其他进程可以调度时，调度运行idle线程



- SCHED_NORMAL调度策略
- CFS 调度器和以往的调度器不同之处在于没有固定时间片的概念，而是公平分配 CPU 使用的时间。比如：2个优先级相同的任务在一个 CPU 上运行，那么每个任务都将会分配一半的 CPU 运行时间，这就是要实现的公平。（实际上是根据nice值(-20,19)分配时间， nice越大对其他进程更友好)
- CFS 没有直接分配优先级，每次调度原则是：总是选择 vriture_runtime 最小的任务来调度（任务运行了多久）





- “实时进程” 优先级 高于 “普通（正常）进程”



- 有两种调度策略： SCHED_FIFO 和 SCHED_RR

1、**SCHED_FIFO 调度策略**：该策略不涉及CPU 时间片机制（分时复用机制），被调度器调度运行后的进程，其**运行时长不受限制**，可以运行任意长的时间，在没有高优先级进程的前提下，只能**等待进程主动释放 CPU 资源**

2、**SCHED_RR 调度策略**：**进程使用完 CPU 时间片后**，会加入到与进程优先级相应的执行队列的末尾；同时，释放 CPU 资源，**CPU 时间片会被轮转给相同进程优先级的其它进程**



普通进程调度：

- 采用SCHED_NORMAL调度策略。
- 分配优先级、挑选进程并允许、计算使其运行多久。
- CPU运行时间与友好值（-20~+19）有关，数值越低优先级越高。



实时进程调度：

- 实时调度的进程比普通进程具有更高的优先级。
- SCHED_FIFO：进程若处于可执行的状态，就会一直执行，直到它自己被阻塞或者主动放弃CPU。
- SCHED_RR：与SCHED_FIFO大致相同，只是进程在耗尽其时间片后，不能再执行，而是需要接受CPU的调度。

进程调度的考量标准

- **等待时间**

- 进程自进入就绪队列开始至进程占用CPU之间的时间间隔

- **周转时间**

- 进程自进入就绪队列开始至进程结束之间的时间间隔

- **CPU吞吐量**

- 单位时间内运行结束的进程个数

进程调度的原则

- **公平性原则：**
 - 应保证每个进程获得合理的CPU份额
- **有效性原则：**
 - CPU资源应得到最大限度的利用
- **友好性原则：响应时间快**
 - 与用户（人）交互的时间应尽可能的短
- **快捷性原则：周转时间短**
 - 批处理作业的处理时间尽可能的短
- **广泛性原则：吞吐量大**
 - 单位时间内完成的作业尽可能的多

时间片轮转调度

- **核心思想：**
 - 每个进程运行固定的时间片，然后调入下一个进程
- **实现机理：**
 - 维护就绪进程队列，采用FIFO方式一次读取
- **特殊控制：**
 - 时间片内发生阻塞或结束，则立即放弃时间片
- **优缺点分析**
 - 优点：绝对公平
 - 缺点：公平即合理吗？时间片如何设计才能保证效率？

优先级调度

- **核心思想：**
 - 为每个进程赋予不同级别的优先级，越高越优先
- **实现机理：**
 - 维护一个优先级队列，自顶向下依次读取
- **特殊控制：**
 - 静态优先级与动态优先级概念
- **优缺点分析**
 - 优点：响应时间快，易于调整。最通用的方法
 - 缺点：死规则，如何保证周转时间和吞吐量？

最短作业优先

- 核心思想：
 - 保证响应时间最快、平均周转时间最短
- 实现机理：
 - 依据**先验信息**，将进程按照运行时间增序调度
- 特殊控制：
 - 如何确定最短作业？（老化算法）
- 优缺点分析
 - 优点：保证了CPU的利用效率
 - 缺点：无法通用，约束条件多

■ 实时调度

- 针对于专用领域和专用应用目的
- 必须具备前提条件才能进行实时调度
- 特点：系统规模小、中断时间短、进程切换快、OS管理深度高

- **FCFS（先来先服务）**：只考虑等待时间；
- **SJF（短作业优先算法）**：只考虑执行时间；
- **HRRN（高响应比优先算法）**

$$\text{优先权} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}} = \frac{\text{响应时间}}{\text{要求服务时间}}$$

- **RR（时间片轮转算法）**：机会均等，都能享有；
- **FB（多级反馈队列调度算法）**
- **EDF（最早截止时间优先算法）**
- **LLF（最低松弛度优先算法）**

反思：如何实现进程调度？

■ 调度机制

- 不同调度算法适用于不同环境和不同目的
- 调度算法一旦固定，则其最优、最坏情况均无法避免
- 如能根据具体情况动态调整，则效果更佳

■ 调度策略

- 为用户提供改变调整调度机制的渠道
- 实现方法——提供系统调用，能够改变调度机制

简答题

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18						

计算题
综合应用题

19	20	21	22
----	----	----	----

23	24	25
----	----	----

标黄色为本次作业