操作系统

L07 CPU调度算法2

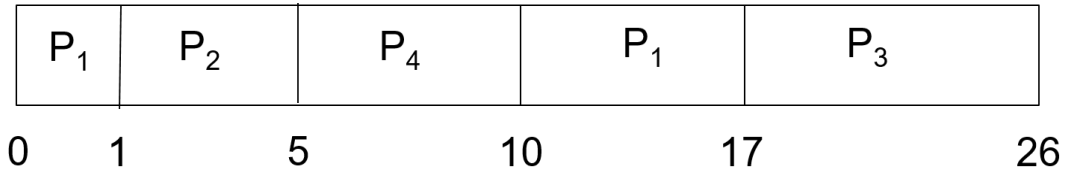
胡燕
大连理工大学 软件学院

0-CPU调度算法1-回顾

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

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- SRTF的调度甘特图



平均等待时间= [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5

SRTF调度可被视为SJF的抢占式版本

0-CPU调度算法1-回顾

- 优先级调度：
  - 每个进程被赋予一个优先数（Priority Number）
  - 每次调度时，优先级最高的任务被选中执行

概念辨析

优先级与优先数的关系视系统而定：
有的系统（如Linux）优先数越小，优先级越高；
有的系统优先数越大，优先级越高

优先级调度又分为：
- 抢占式优先级调度（Preemptive priority scheduling）
- 非抢占式优先级调度（Non-preemptive priority scheduling）

0-CPU调度算法1-回顾

- 问题：SJF算法存在的问题？

  长进程饥饿（Starvation）问题

- **最高响应比优先调度**（Highest Response Ratio Next,HRRN）

  选择就绪队列中相应比最高的进程投入接下来的运行

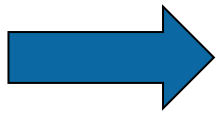
  Response Ratio = (W+T)/T

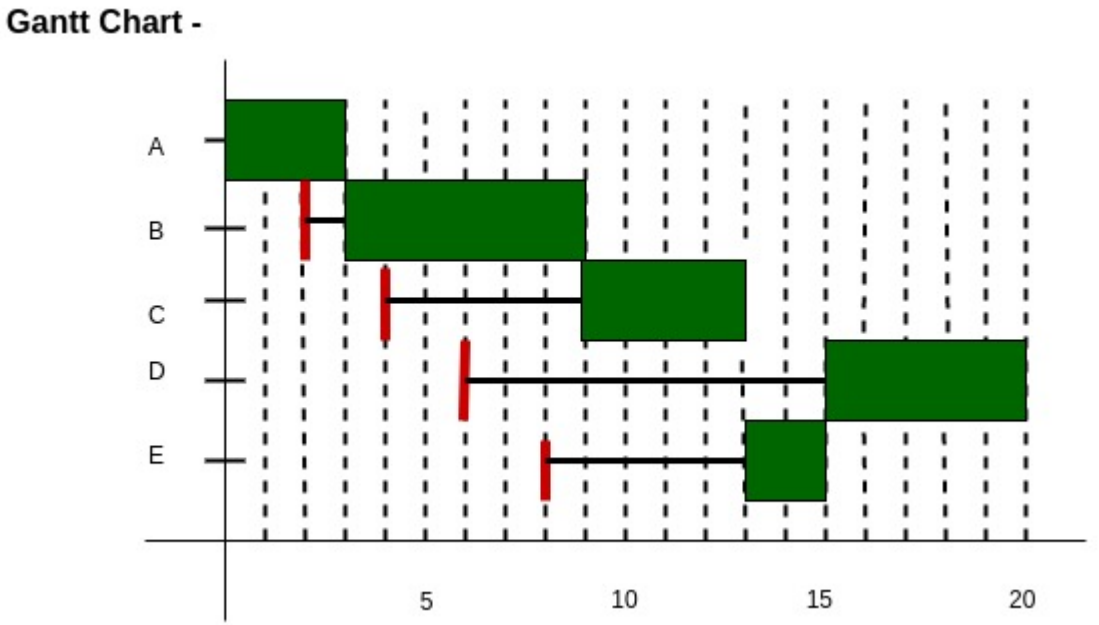
  W代表等待时间

  T代表进程将要执行时间长度（CPU Burst）

HRRN示例

Thread	Arrival TIme	CPU Burst Length
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2





0-CPU调度算法1-回顾

- 问题：我们认为可以将FCFS调度算法归入优先级调度算法这一类，为什么？

把进程的到达时间戳作为优先数，优先数越小，优先级越高

0-CPU调度算法1-回顾

- 问题：请简述FCFS调度算法中的Convoy Effect(护航效应)?

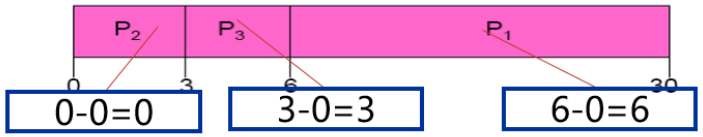
1-FCFS调度算法

- FCFS算法调度示例

进程	CPU Burst time
P1	24
P2	3
P3	3

假设3个进程的到达顺序改为P2,P3,P1(时刻0依次到达)

通过FCFS算法得到的Gantt图



平均等待时间: 6 + 0 + 3)/3 = 3

Convoy Effect:
FCFS算法不稳定，长进程先于短进程到达，会导致平均等待时间拉长

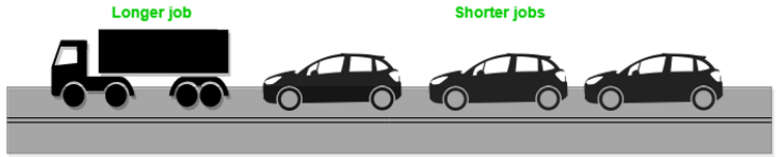
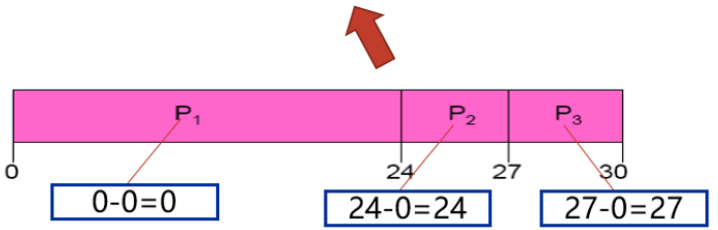


Figure - The Convey Effect, Visualized



平均等待时间: (0 + 24 + 27)/3 = 17

0-CPU调度算法1-回顾

- 问题：我们认为可以将SJF调度算法归入优先级调度算法这一类，为什么？

优先数是进程下一个CPU周期长度，优先数越小，优先级越高

0-CPU调度算法1-回顾

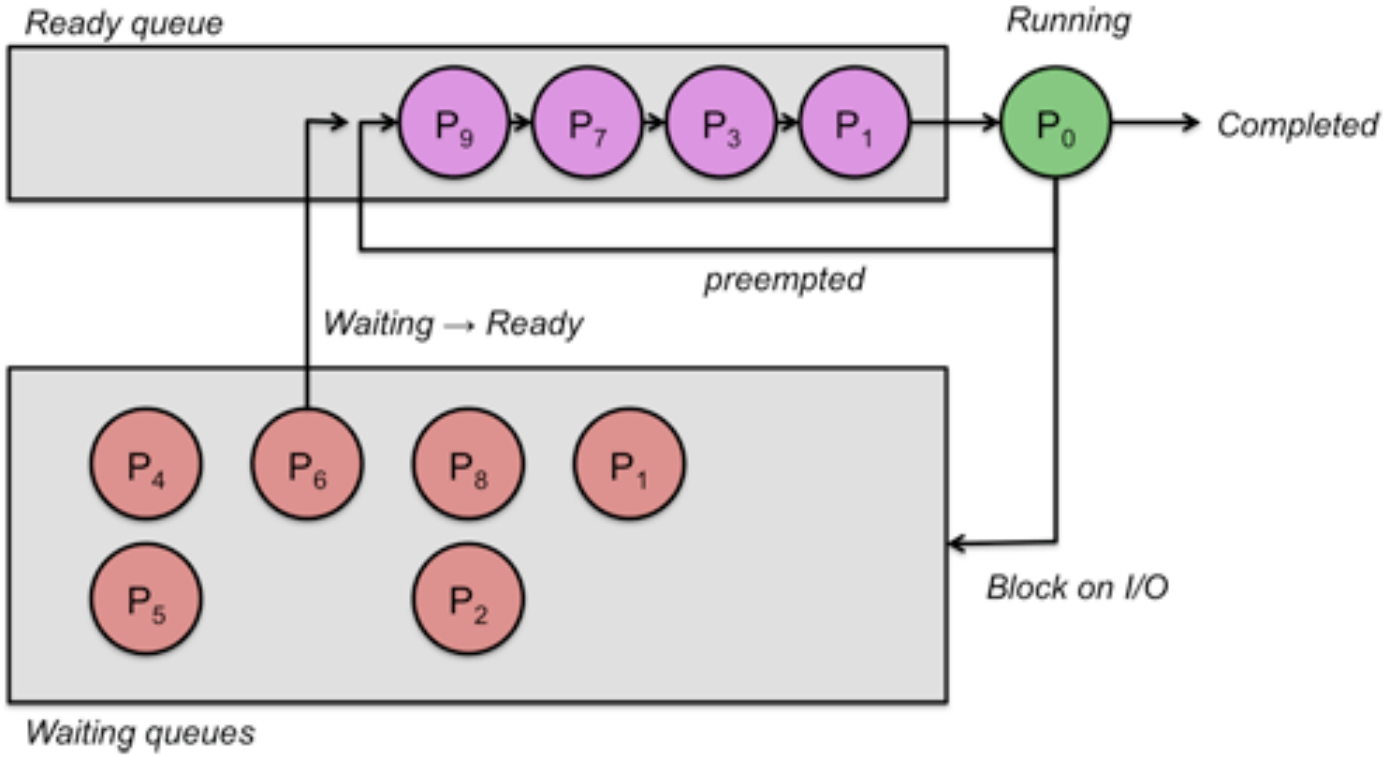
- 问题：优先级算法存在的问题？

低优先级进程可能被饿死（Starvation）

解决优先级调度中进程饥饿问题的方法：
采用老化机制（aging）

5.4-轮转调度

Round Robin调度



为什么需要RR调度

FIFO　　算法不稳定，无抢占特性

SJF　　响应时间长

SRTF　　存在Starvation问题

存在公平性问题

如何保证OS内核对进程的调度的公平性?

Round Robin

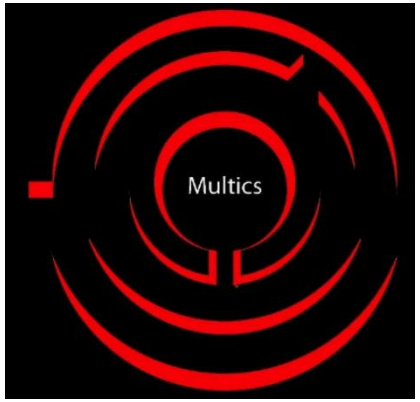
核心思想：进程轮流使用CPU

轮转调度是分时系统中的基础调度算法

分时系统背景

- IBM CTSS: 大型机上的分时系统 （1962）
- Multics (1965-1969)
- Unix (1969)

Prof. Fano using CTSS



Multics

Unix Workstation

RR基本思想

RR算法基本做法：

①系统将所有就绪进程按到达时间的先后次序排成一个队列

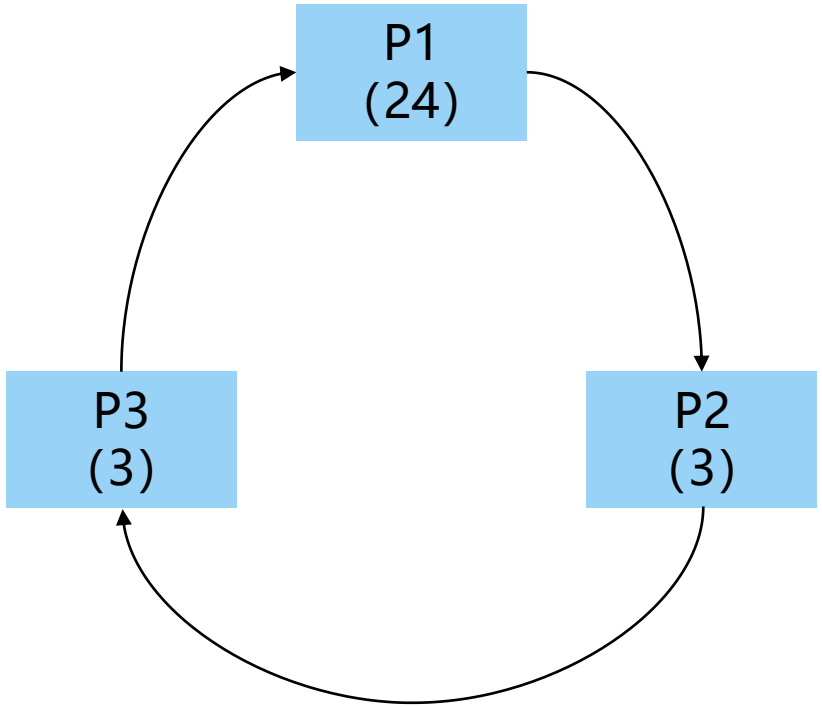
②进程调度程序总是选择就绪队列中第一个进程执行，即先来先服务的原则，但仅能运行一个时间片

③在使用完一个时间片后，即使进程并未完成其运行，它也必须释放出（被剥夺）处理机给下一个就绪的进程，而被剥夺的进程返回到就绪队列的末尾重新排队，等候再次运行

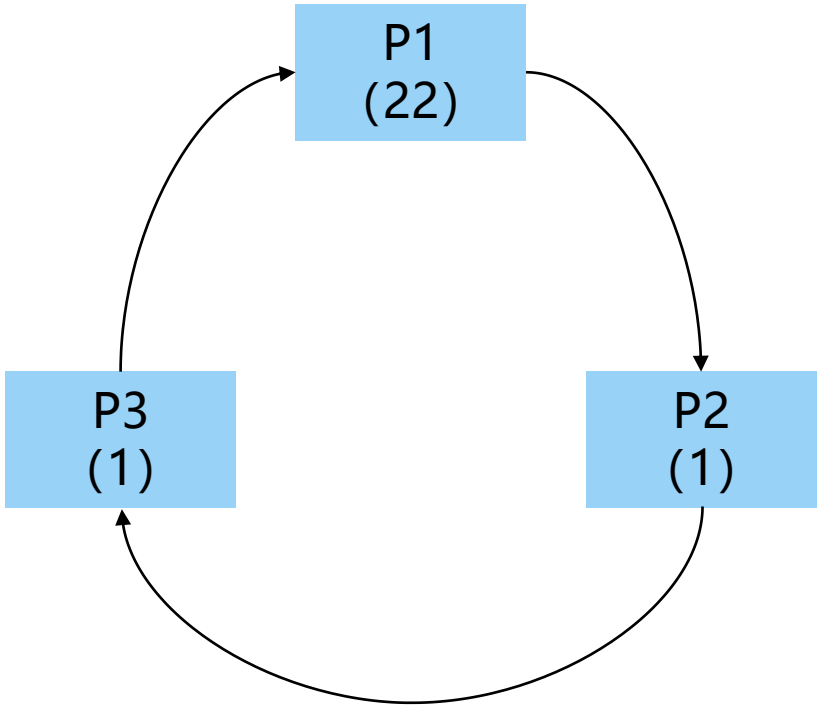
RR的目标：将CPU时间公平地分配给系统中的进程

核心思想：尽量将CPU时间等分，让各个进程轮流按需索取

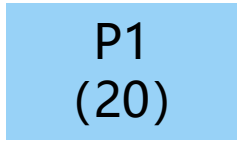
RR基本思想



初始状态



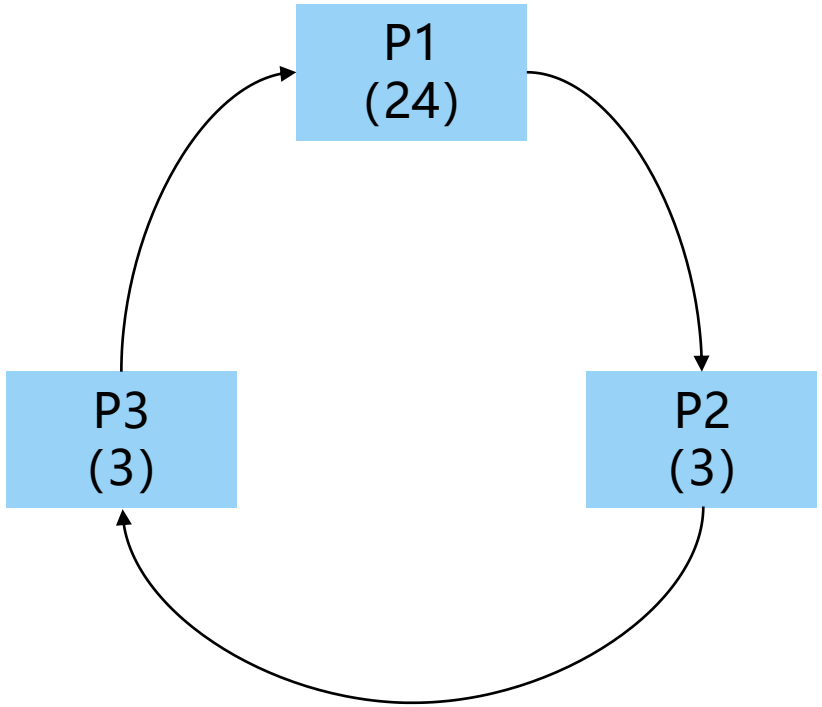
Round 1



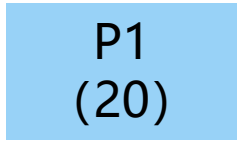
Round 2

时间片=2

RR基本思想



初始状态



Round 1

时间片=4

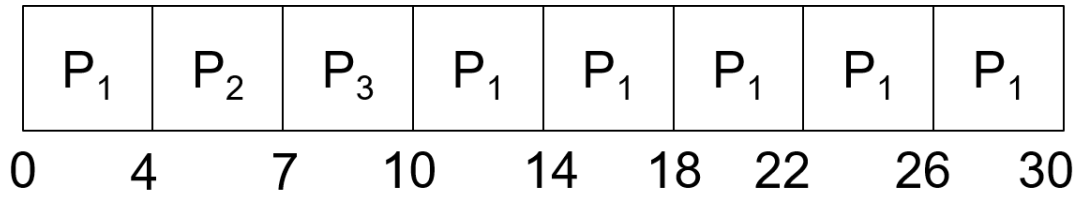
RR调度示例

RR算法：示例

Process	Burst Time
P1	24
P2	3
P3	3

(Time Quantum=4)

轮转调度作用于示例的甘特图



平均等待时间=(6+4+7)/3=$5\frac{2}{3}$

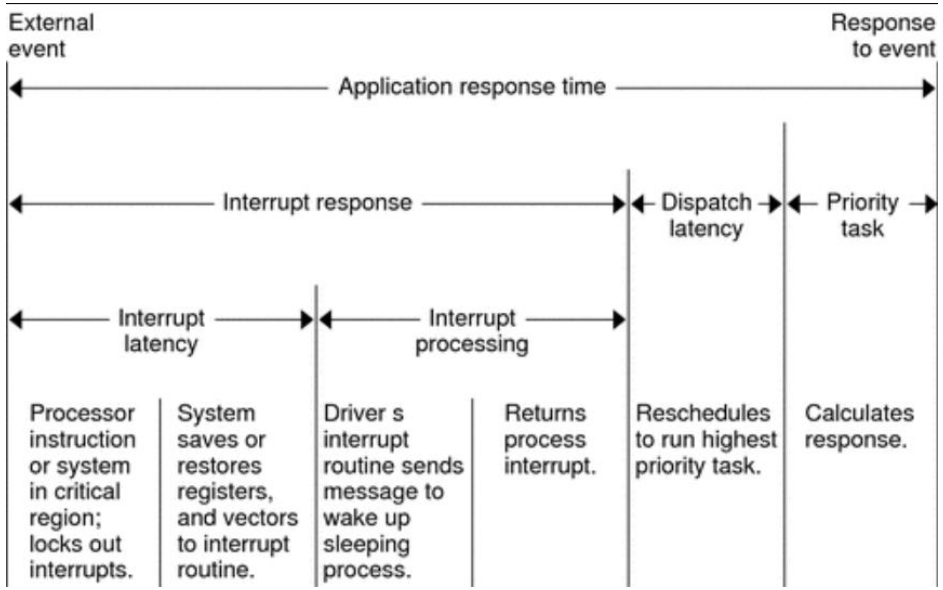
对比：FCFS算法，平均等待时间=17

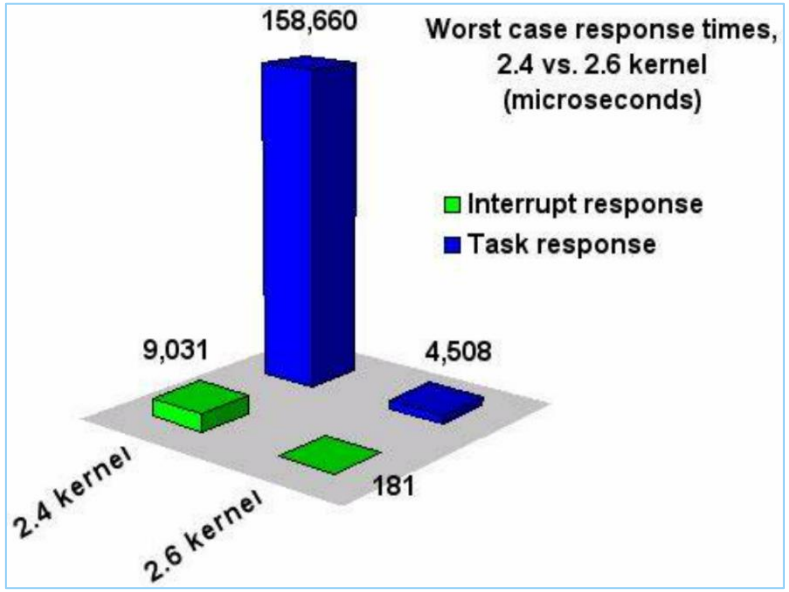
RR调度算法：响应时间分析

RR算法：worst-case response time

假设确定当前就绪队列中的进程个数为n，时间片=q，则可以预计当前运行进程的最坏响应时间
Rt = （n-1）*q

因此，为了保证交互性，时间片q的值不可设置得过大



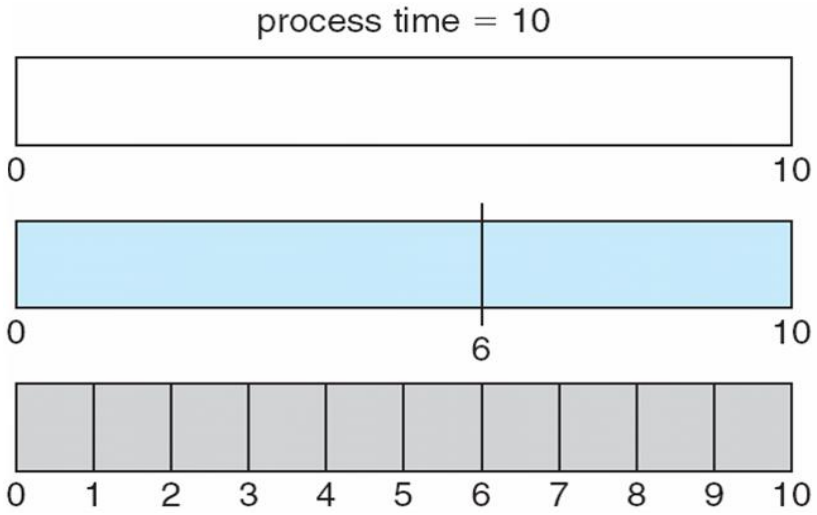


RR调度算法：响应时间分析

时间片大小设置

为了保证交互性，轮转算法的时间片设置的要相对较小。

问题：时间片是否设置得越小越好?



quantum	context switches
12	0
6	1
1	9

RR调度算法：响应时间分析

时间片大小设置

为了保证交互性，轮转算法的时间片设置的要相对较小。

例题：

进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

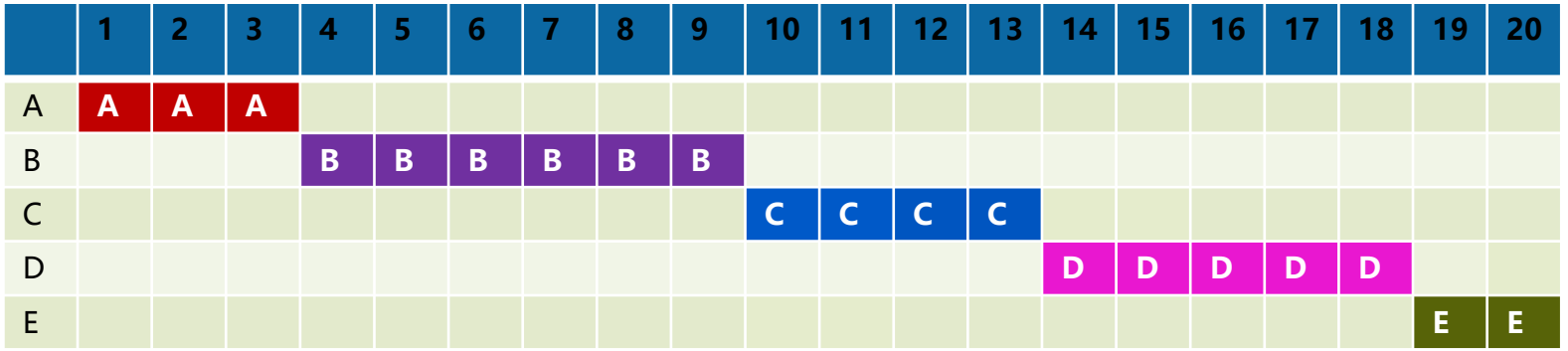
请同学们试着给出FCFS、RR（q=1)、RR(q=4)的调度。

RR调度算法：响应时间分析

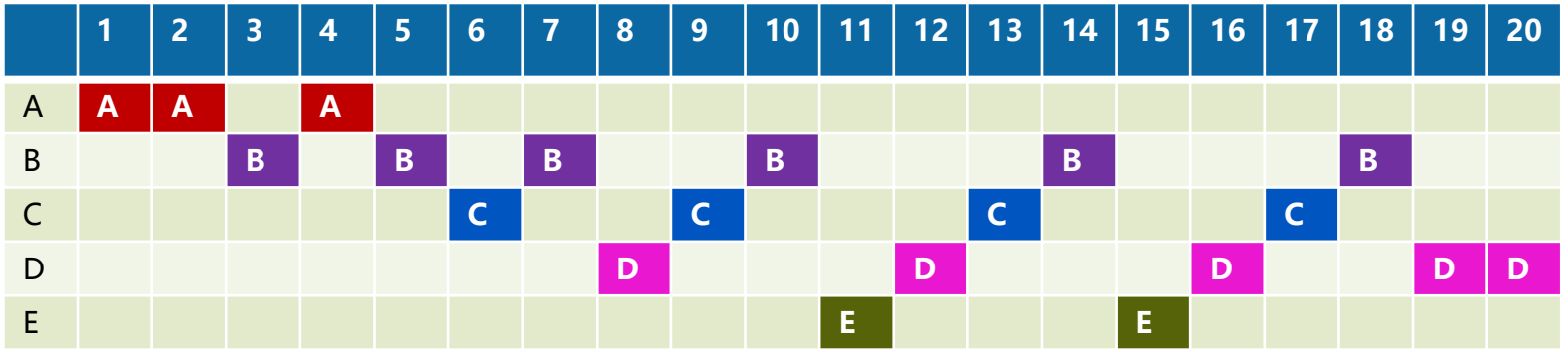
时间片大小设置

进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

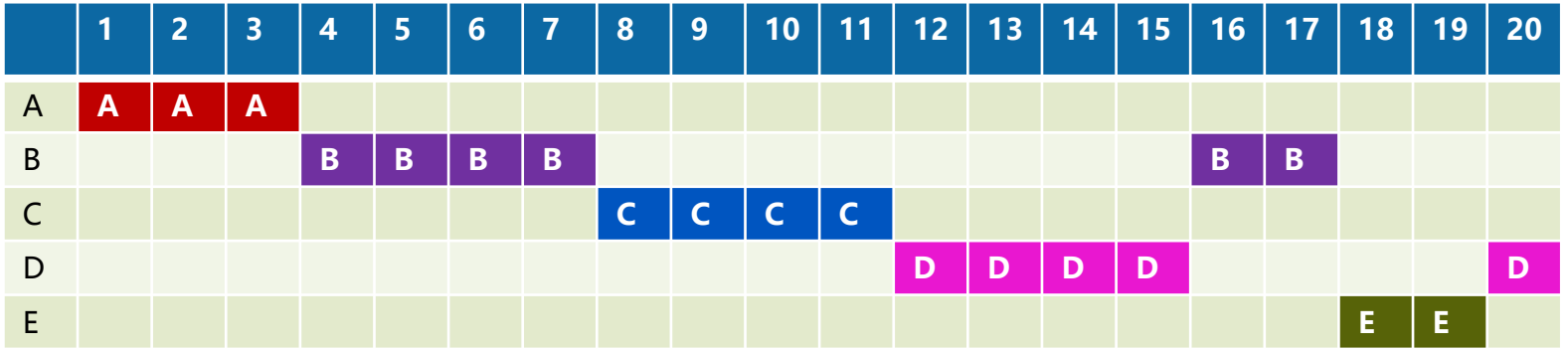
FCFS



RR (q=1)



RR (q=4)



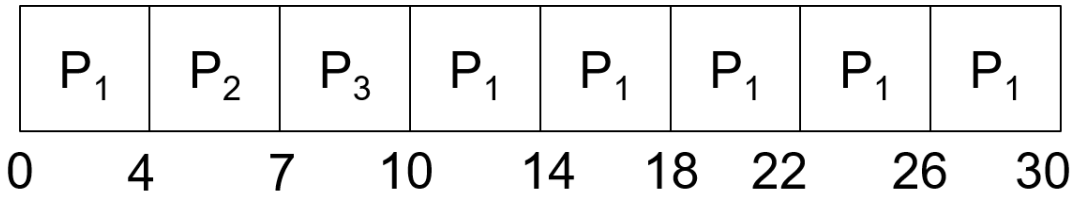
RR调度算法：就绪队列变化

RR算法：示例

Process	Burst Time
P1	24
P2	3
P3	3

(Time Quantum=4)

轮转调度作用于示例的甘特图



时刻0　　就绪队列：P1,P2,P3

选中P1去执行

就绪队列：P2,P3

时刻4　P1转至队尾，选中P2去执行

就绪队列：P3,P1

时刻7　P2终止，选中P3去执行

就绪队列：P1

时刻10 P3终止，选中P1去执行

就绪队列：为空

时刻14　P1继续执行

就绪队列：为空

…

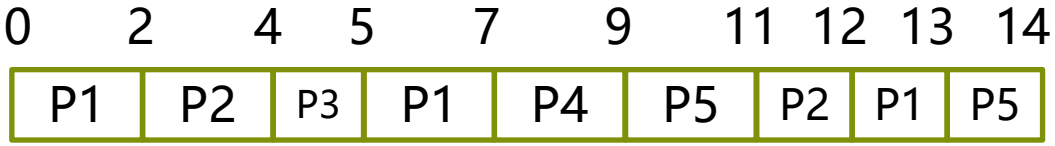
RR调度算法：就绪队列变化

RR算法：练习 (Time Quantum=2)

五个进程的到达时间和执行时间如下表，采用RR调度，时间片=2

PID	Arrival Time	Burst Time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

调度的Gantt图应该是怎样的?



	队列状态
时刻0	运行：P1,就绪：空
时刻1	运行：P1,就绪：P2
时刻2	运行：P2,就绪：P3,P1
时刻3	运行：P2,就绪：P3,P1,P4
时刻4	运行：P3,就绪：P1,P4,P5,P2
时刻5	运行：P1,就绪：P4,P5,P2
时刻7	运行：P4,就绪：P5,P2,P1
时刻9	运行：P5,就绪：P2,P1
时刻11	运行：P2,就绪：P1,P5
时刻12	运行：P1,就绪：P5
时刻13	运行：P5,就绪：空

背景

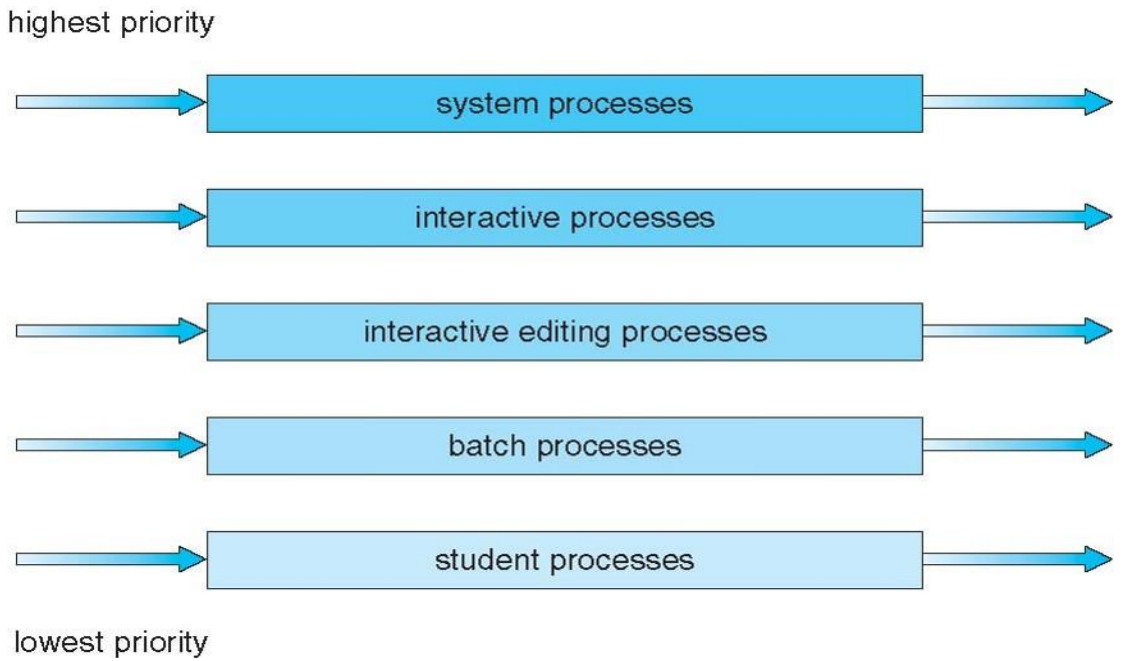
Priority Queue

优先级队列

操作系统中有很多任务(成百上千)

系统中设定优先级区间
- 例如：0~255，256个优先级

系统内会存在多个任务属于同一优先级的情况 => 通过优先级队列实现



MLQ调度实现思路

- **多级队列调度设计思路**

设计考虑1：不同队列可由不同的调度算法管理

例如: 将进程分为前台交互就绪队列和后台批处理就绪队列；前台进程调度使用RR算法，后台进程调度采用FCFS算法

MLQ调度实现思路

- 多级队列调度设计思路

设计考虑1：不同队列可由不同的调度算法管理

例如: 将进程分为前台交互就绪队列和后台批处理就绪队列；前台进程调度使用RR算法，后台进程调度采用FCFS算法

设计考虑2：为不同队列进程分配不同大小的时间配额

例如: 前台进程队列使用80%的CPU时间，后台进程队列享用20%的CPU时间

MLQ调度实现思路

- **MLQ示例：**

Process	Arrival Time	CPU Burst Time	Priority
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

MLQ调度实现思路

- 多级队列调度的优点与问题

优势

- 不同类型队列自由选择调度算法（调度策略灵活）
- 可以为不同类别的进程队列分配不同的时间配额（CPU时间分配策略灵活）

问题

- 进程固定于特定优先级队列（不便于动态调控）
- 高优先级队列中的进程具有优势，低优先级队列中的进程面临饥饿问题的概率高（Starvation）

多级反馈队列调度

MLQ

问题1：进程固定于特定队列（不灵活）

问题2：低优先级进程饥饿概率高（不公平）

MLFQ

策略1：令进程可在不同队列间移动（增加灵活性）

策略2：Aging（进程老化）：对等待过久的进程，提升其优先级（避免饥饿，保证公平）

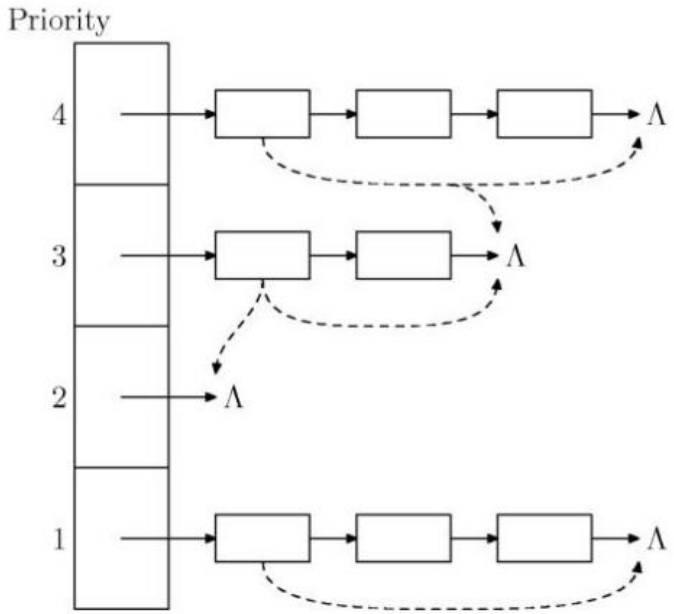
Multi-Level Feedback Queue

多级反馈队列调度

MLFQ调度算法：

MLFQ调度，设置多个不同优先级的就绪进程队列，并允许为每个优先级队列设置不同的调度算法或调度参数（与MLQ调度一致）。

与MLQ调度不同之处在于，MLFQ调度会持续分析进程的运行行为，并允许必要时对进程进行优先级调整，即允许进程在不同优先级队列间移动。



重要策略：
在执行完一个时间片后，任务是否会降低优先级（Demote）

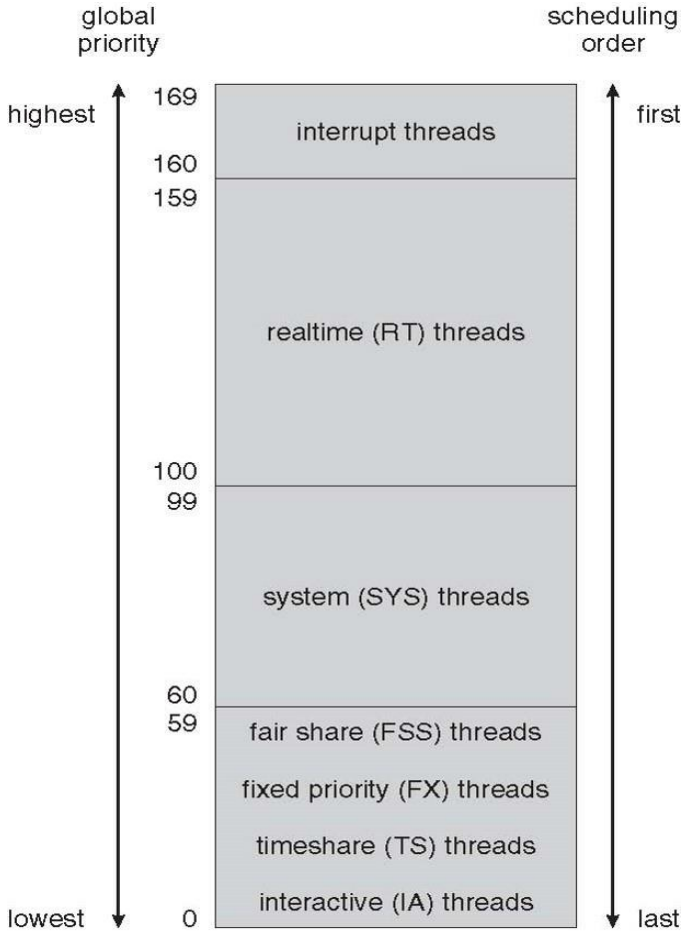
多级反馈队列调度

MLFQ调度算法：系统实例

主流操作系统都采纳MLFQ调度算法作为核心算法

多级反馈队列调度

MLFQ调度算法代表：Solaris操作系统的优先级调整策略表



Solaris优先级

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

MLFQ调度: Windows

MLFQ调度算法代表：CPU Scheduling in Windows (NT kernel)

Thread Relative Priority	Process Priority Classes					
	Realtime	High	Above Normal	Normal	Below Normal	Idle
Time-critical	31	15	15	15	15	15
Highest	26	15	12	10	8	6
Above-normal	25	14	11	9	7	5
Normal	24	13	10	8	6	4
Below-normal	23	12	9	7	5	3
Lowest	22	11	8	6	4	2
Idle	16	1	1	1	1	1

Priority Boost：

因I/O完成等被唤醒的线程，优先级可适当调整

基于优先级的抢占式调度

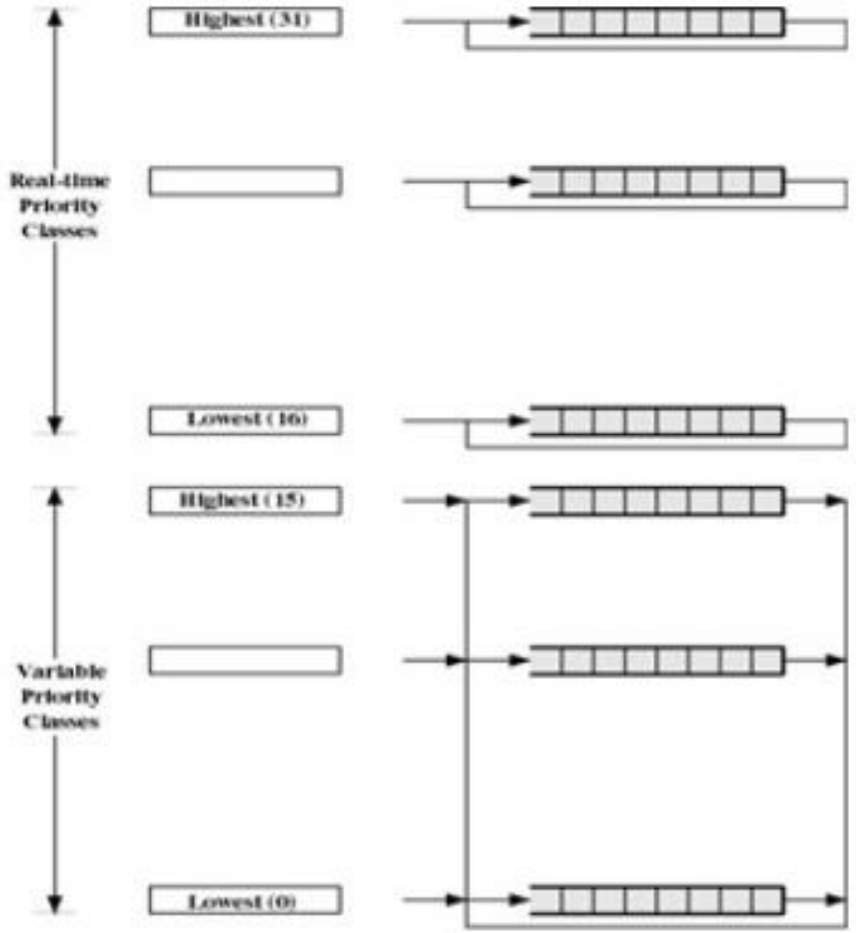
32个优先级：0-31

非实时优先级（1-15）可动态调整

MLFQ调度: Windows

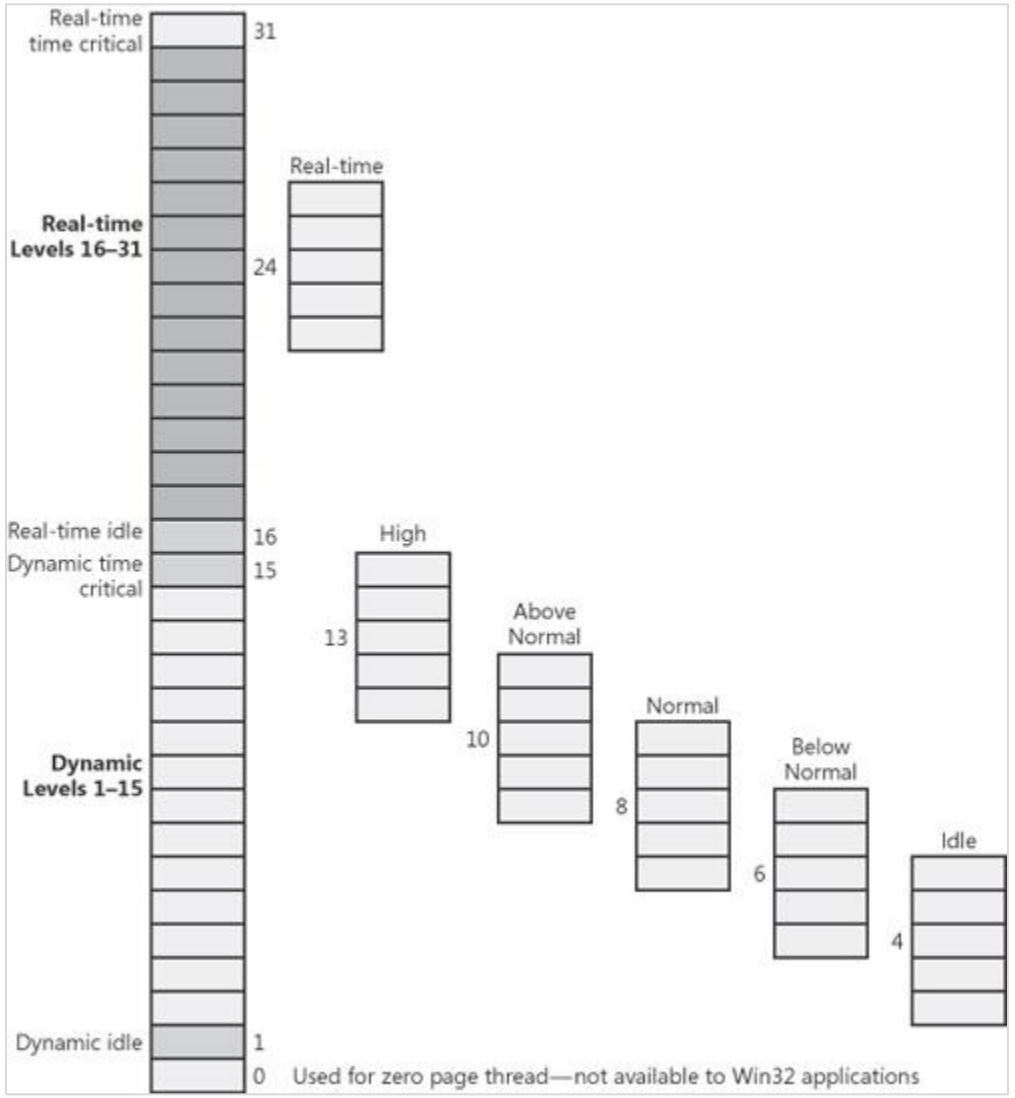
- **Windows NT-based OS use a multilevel feedback queue.**
  - 基于优先级的抢占式调度
  - 使用多级反馈队列

32个优先级:
（1-15：variable class, 16-31:real time)
优先级0：赋予memory-management thread



5.5-多级队列调度

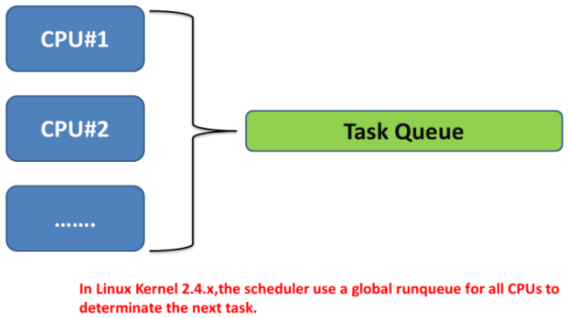
MLFQ调度: Windows



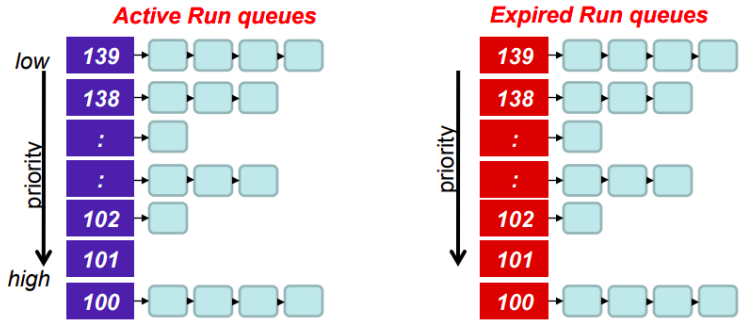
通过Relative Priority动态调整优先级范围

Linux中的调度算法

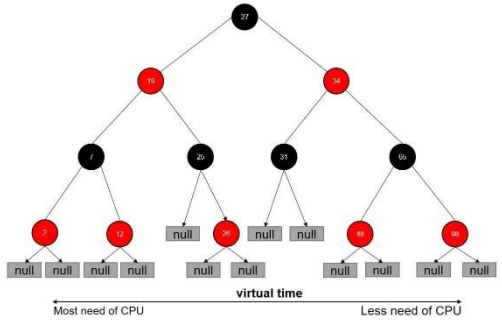
Linux调度算法的变迁



O(n)



O(1)

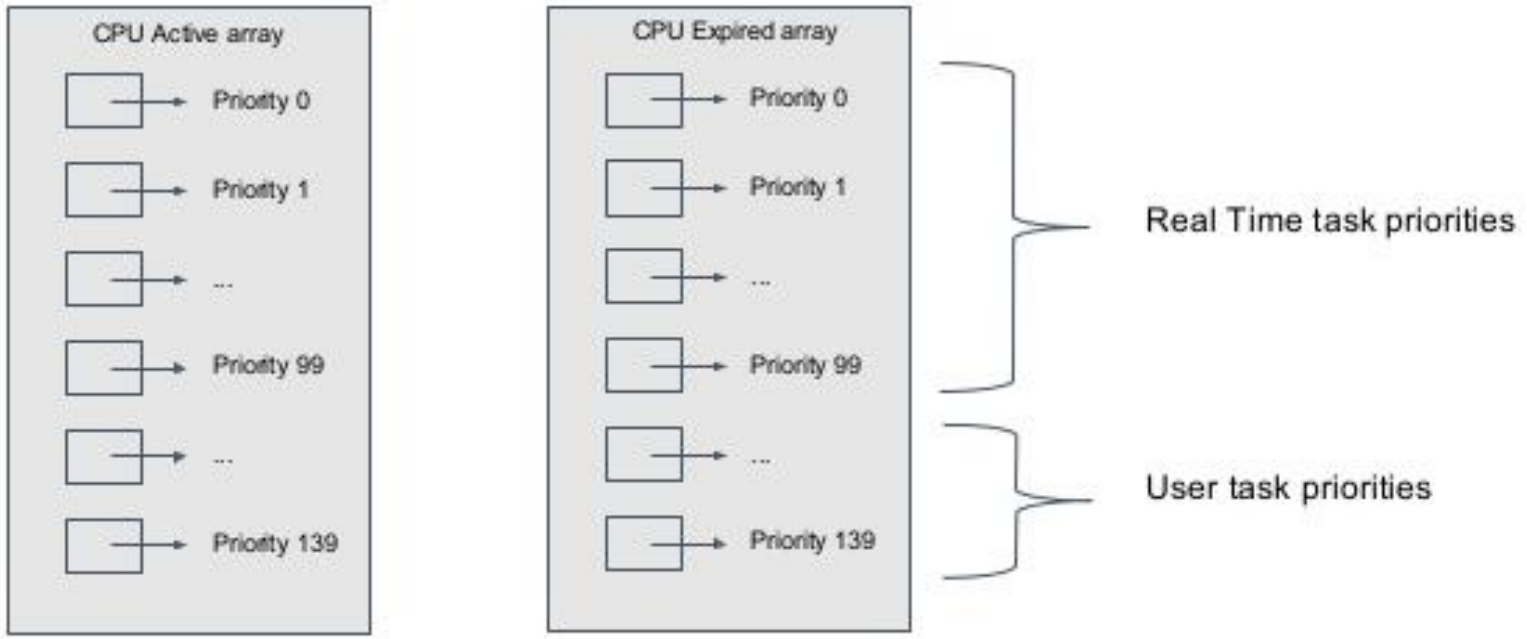


CFS

Linux中的调度算法

Linux O(1)调度器：

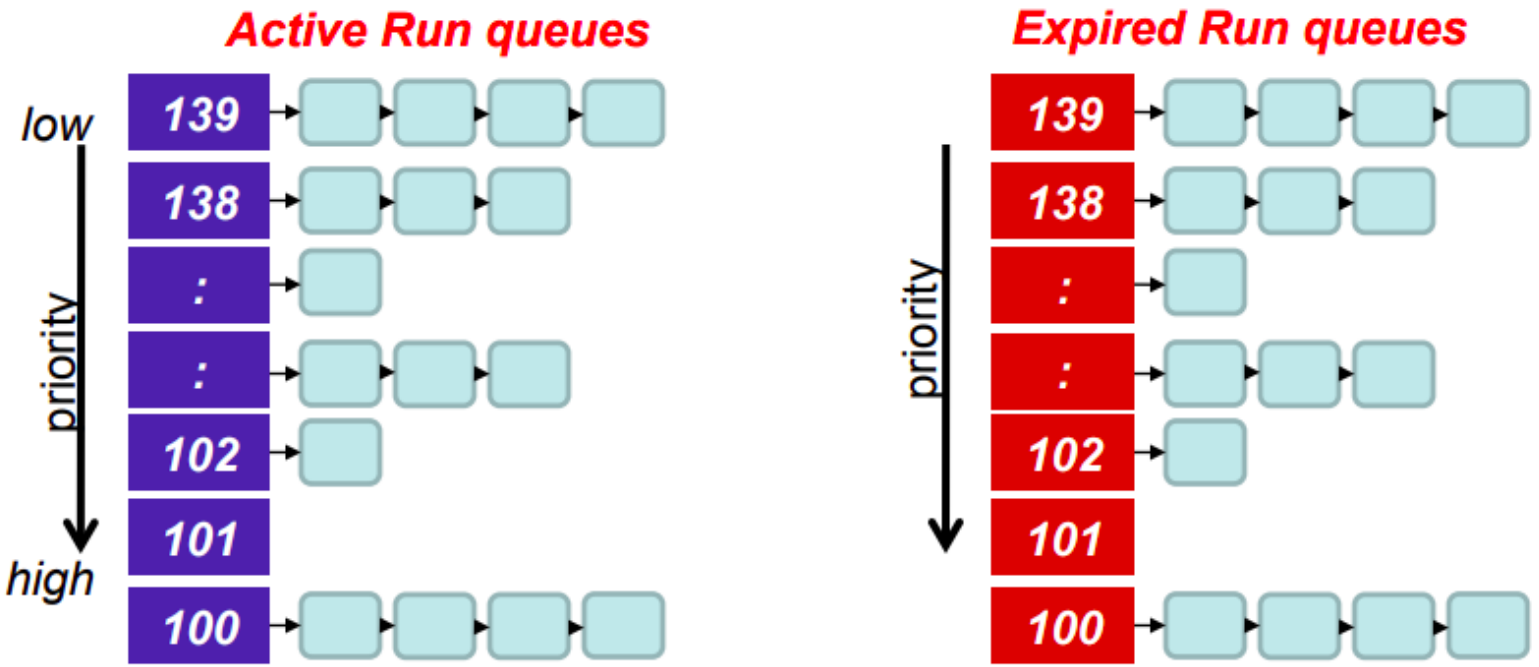
在Linux 2.6中首度引入
140个优先级（实时：0-99，普通进程：100-139）　注意:Linux中优先数越大，优先级越低



Linux中的调度算法

Linux O(1)调度器：

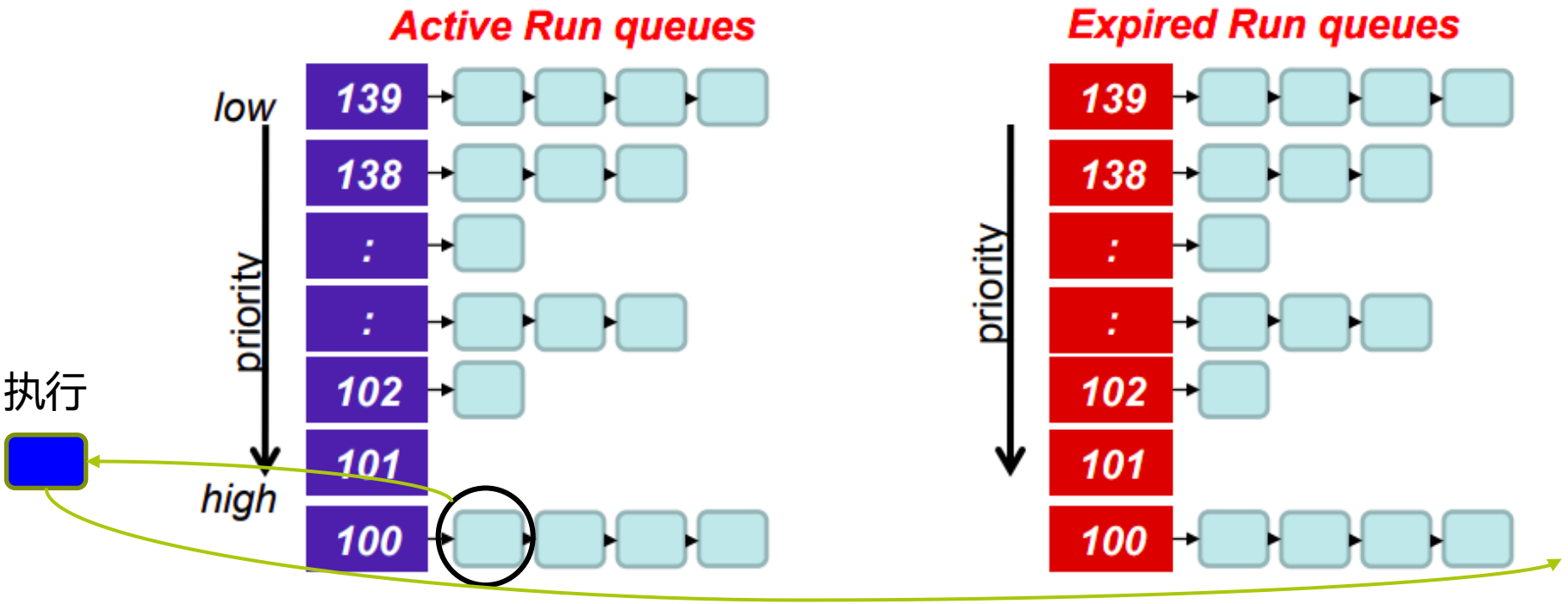
在Linux 2.6中首度引入
140个优先级（实时：0-99，普通进程：100-139）　　注意:Linux中优先数越大，优先级越低



Linux中的调度算法

Linux O(1)调度器：

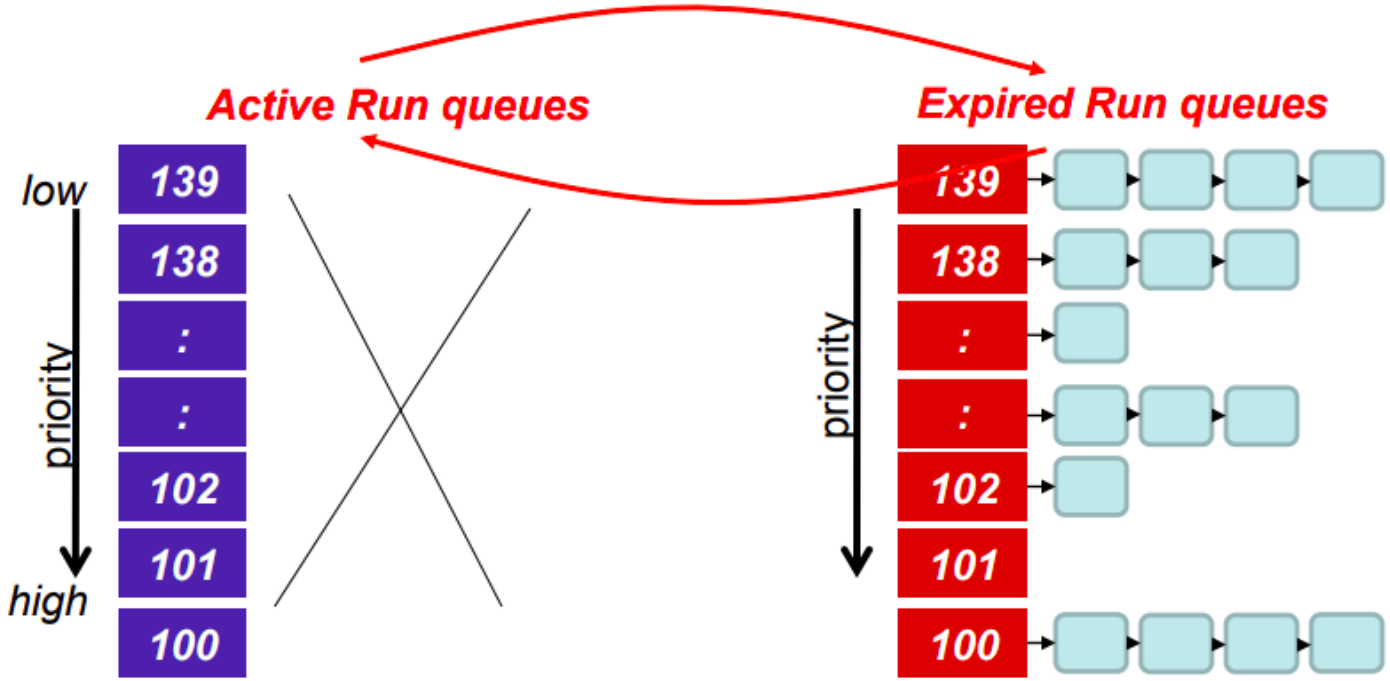
在Linux 2.6中首度引入
140个优先级（实时：0-99，普通进程：100-139）　　注意:Linux中优先数越大，优先级越低



Linux中的调度算法

Linux O(1)调度器：

在Linux 2.6中首度引入
140个优先级（实时：0-99，普通进程：100-139）　　注意:Linux中优先数越大，优先级越低



Linux CFS调度算法：

CFS调度的核心思想是维护为任务提供CPU时间方面的平衡。

当为任务的时间分配导致失去平衡时，应将CPU时间的分配向得到CPU时间少的任务倾斜

CFS通过vruntime来记录任务获取CPU时间的量

Vruntime = 实际运行时间 * 1024 / 任务权重

（在核心函数calc_delta_fair()内实现计算）

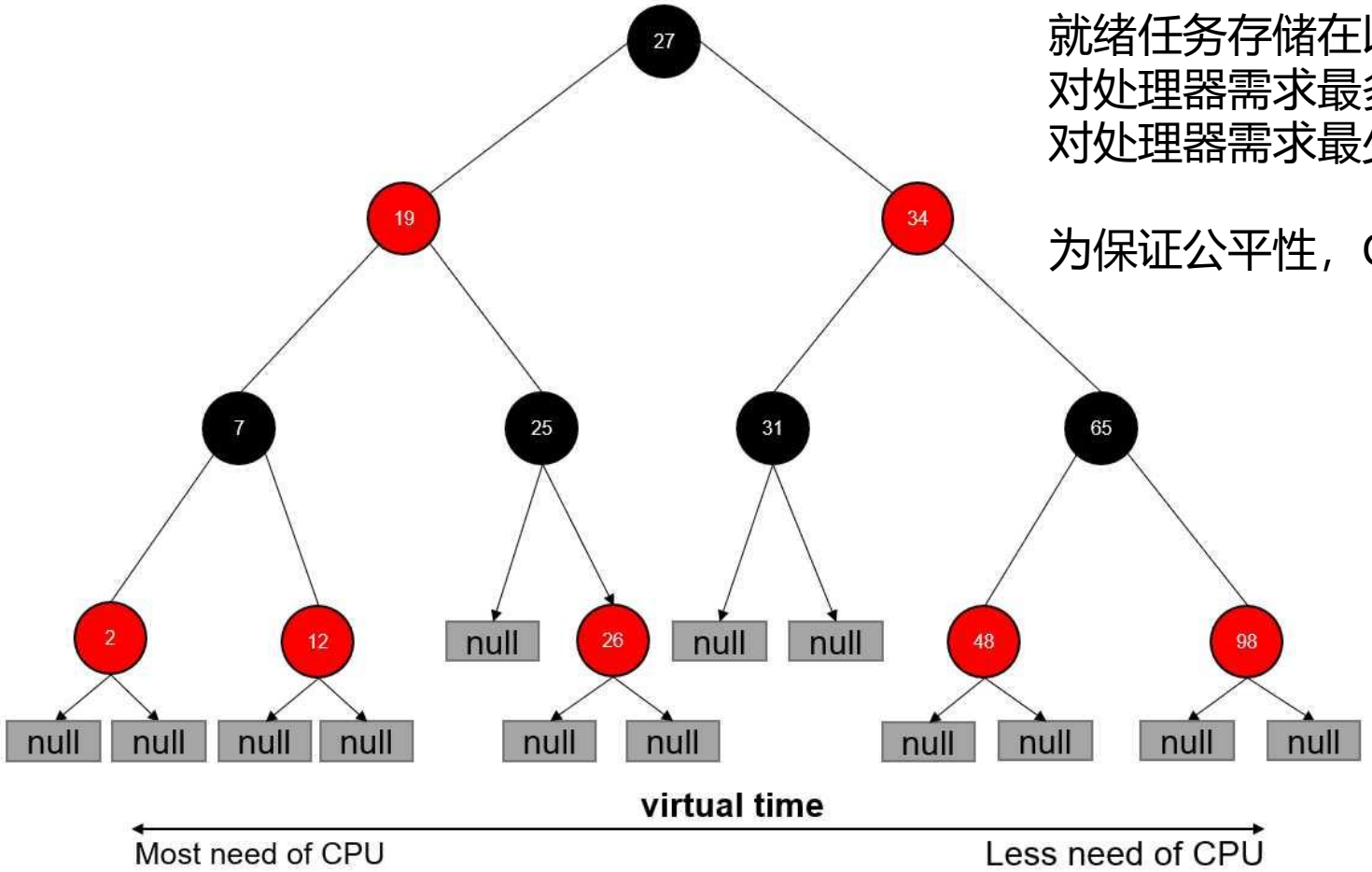
进程按照各自不同的速率在物理时钟节拍内推进

-优先级高的任务，其虚拟时钟比真实时钟跑得慢，但获得较多的运行时间

-优先级低的任务权重小，其虚拟时钟比真实时钟慢，反而获得较少的运行时间

CFS调度器总是选择虚拟时钟跑得最慢的任务来运行

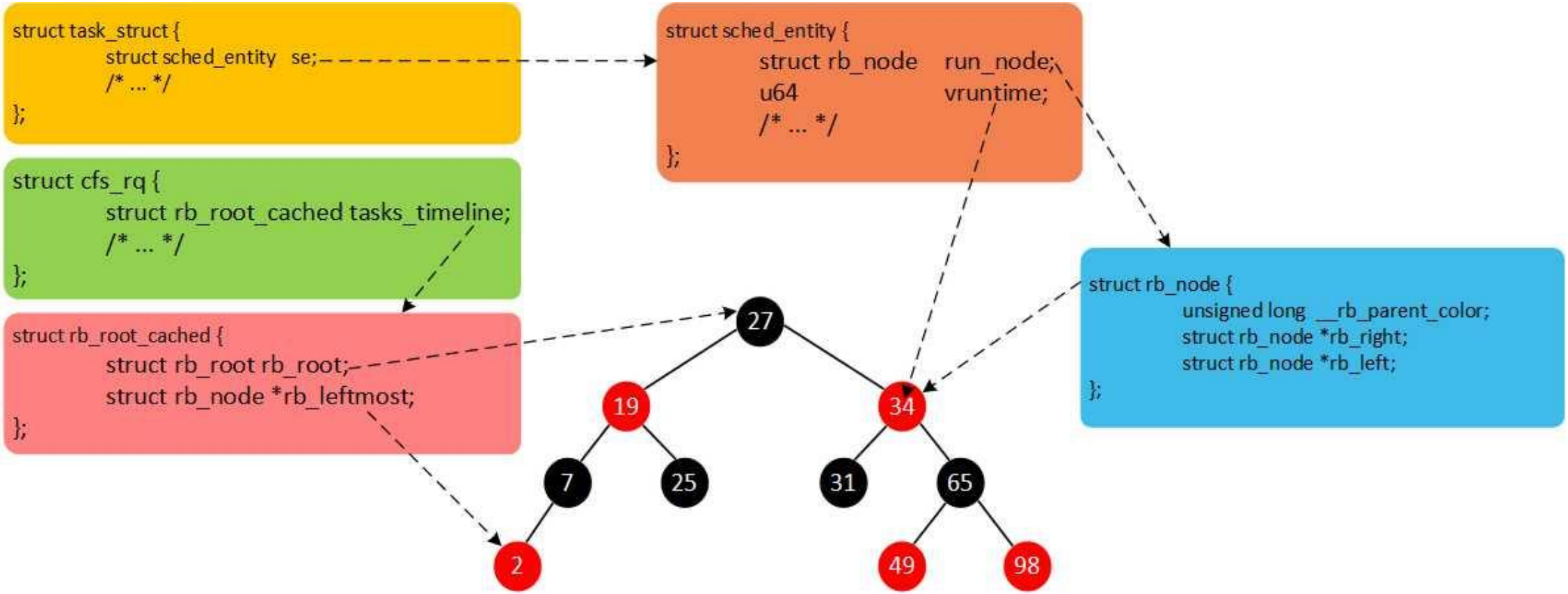
Linux CFS调度算法：示例说明



就绪任务存储在以vruntime排序的红黑树中
对处理器需求最多（vruntime最小）的任务在树的左侧
对处理器需求最少（vruntime最高）的任务在树的右侧

为保证公平性，CFS调度器每次选取最左端的任务去执行

Linux CFS调度算法：数据结构关系说明



Summary

小结：

- 轮转调度
- 多级队列调度
- 调度算法案例

计时器中断 （Timer Interrupt）

中断处理例程中实施调度

谢 谢!
Thank you!