中国科学院大学
University of Chinese Academy of Sciences

操作系统课程 **B0911010Y-01**

CPU调度

中国科学院大学计算机与控制学院
中国科学院计算技术研究所
2019-09-23

内容提要

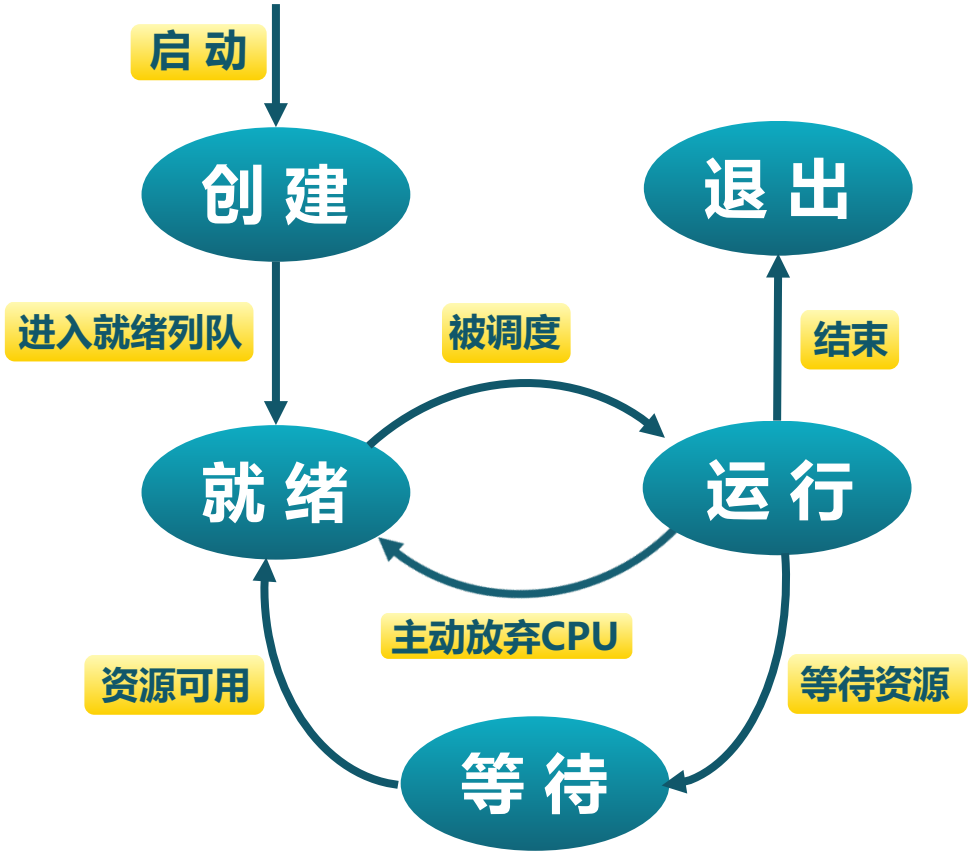
- CPU调度基础
- CPU调度算法

进程调度

- 进程切换与调度
  - CPU是共享资源，进程按照一定策略使用CPU资源
  - 进程切换
    - 进程A离开不再使用CPU，进程B占据使用CPU
  - 进程调度
    - 进程A离开不再使用CPU，哪个进程占据使用CPU
  - 进程切换场景
    - 进程主动放弃CPU使用权（非抢占式）
    - 进程等待IO，等待资源或特定事件（非抢占式）
    - 内核不让进程使用CPU，例如有更高优先级进程要运行（抢占式）

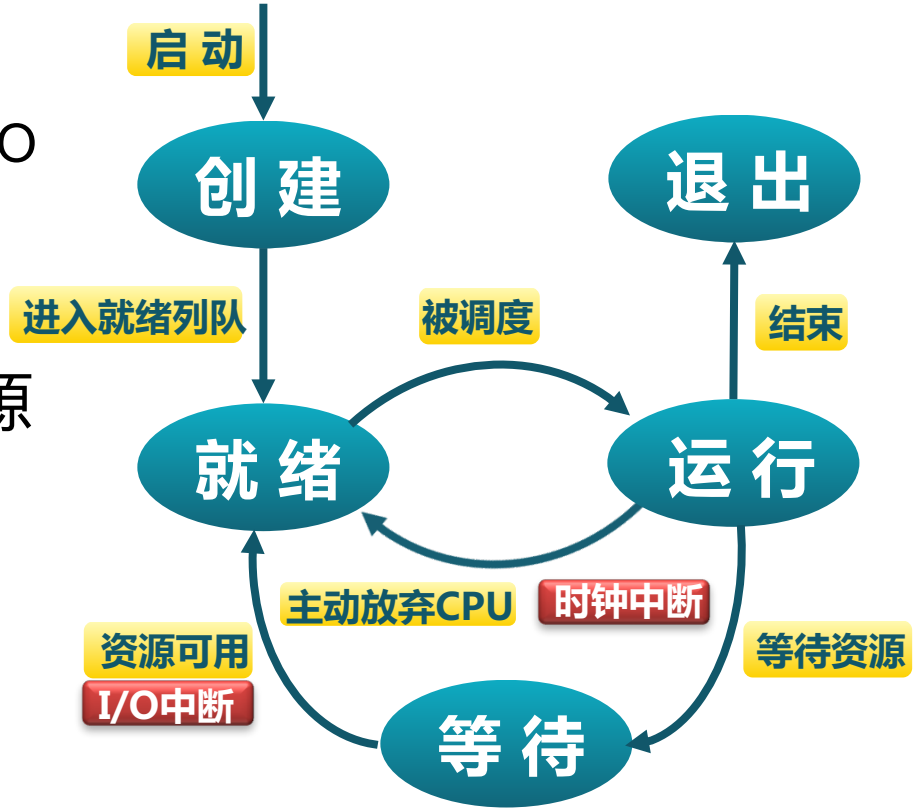
非抢占式调度

- 非抢占式调度
  - 进程主动放弃CPU使用权：yield()
  - 进程等待IO，等待资源或特定事件:block()
  - 主动调用调度器进行调度进程：schedule()



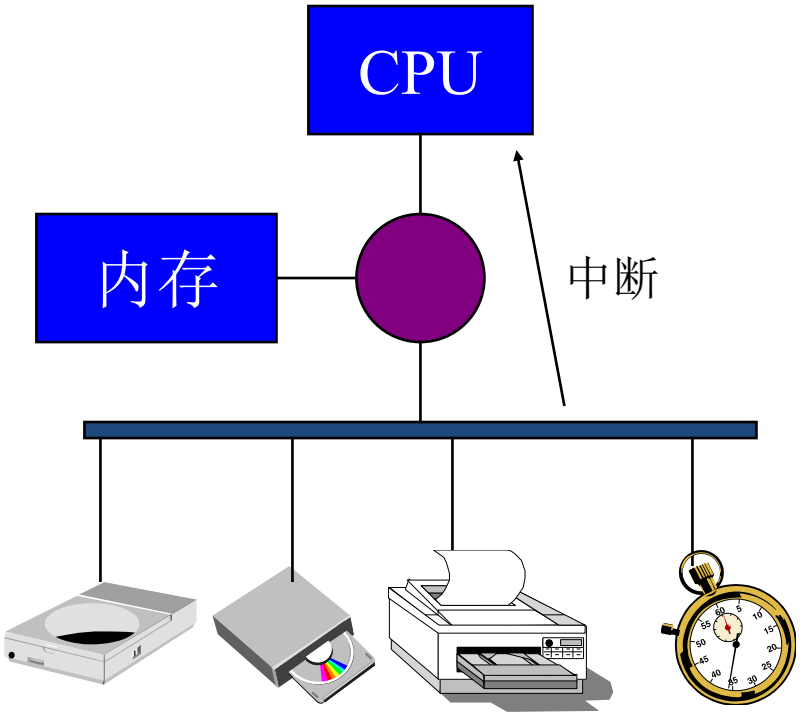
抢占式调度

- 抢占式调度
  - 有更高优先级进程要执行
  - 内核强制切换进程
    - 基于中断，例如时钟中断、IO中断
- 为什么要抢占
  - 利用时间中断管理CPU资源
  - 异步I/O和计算交叠在一起



抢占式调度

- 中断
  - 同步中断
    - 指令执行时由CPU控制单元产生，一条指令执行完才产生，发生在指令之间
  - 异步中断
    - 其他硬件设备产生，可以发生在指令执行期间
    - SMP结构下，非原子操作
- 中断类型
  - 可屏蔽中断
    - 关闭/开启中断
  - 不可屏蔽中断（Non-Masking Interrupts ，NMI)
    - 例如，无法恢复的硬件错误



抢占式调度的中断处理

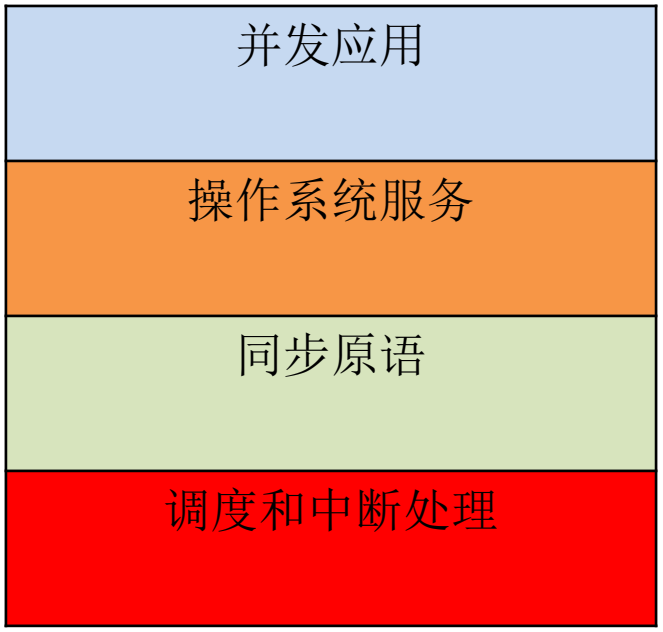
- 中断处理基本流程
  - CPU检查中断条件是否满足
    - 有中断请求
    - CPU允许中断
  - 如果CPU允许中断，关中断，不再响应中断
  - 保存被中断的现场
  - 判断中断类型，调用中断处理程序
  - 执行中断处理程序
  - 恢复现场
  - 开中断
- 中断上半部（top-half）和下半部（bottom-half）

抢占式调度的中断处理

- I/O中断处理
  - 保存当前进程/线程到它们的PCB/TCB
  - 进行I/O
  - 调用调度器
- 时间中断处理
  - 保存当前进程/线程到它们的PCB/TCB
  - 更新系统时间，递减进程时间片
  - 调用调度器

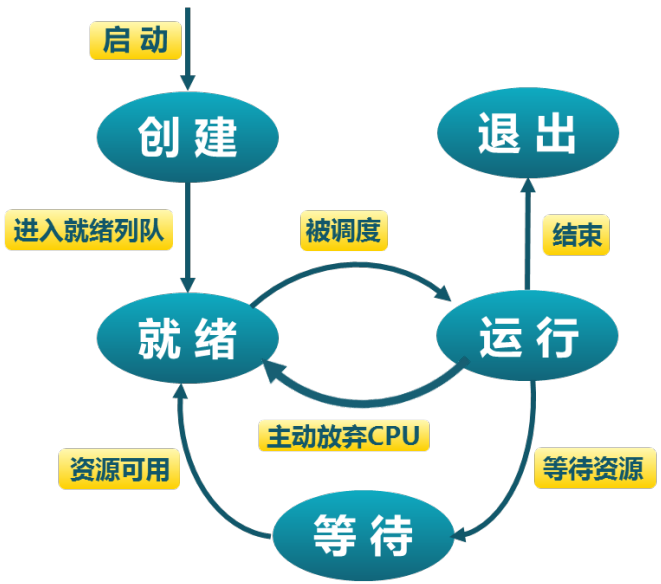
抢占式调度下的共享资源访问

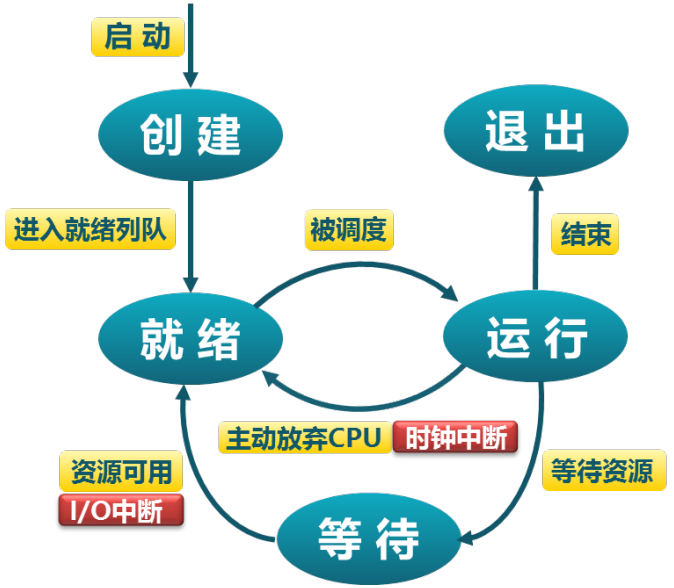
- 问题
  - 中断随时随地都可能发生
  - 如何保证对共享资源的访问
- 简单方法
  - 应用时刻关注是否发生中断和抢占
- 目标
  - 不要时刻关注中断和抢占
  - 底层行为被封装在“原语”中
  - 同步“原语”关注抢占
  - OS和应用使用同步原语



调度器

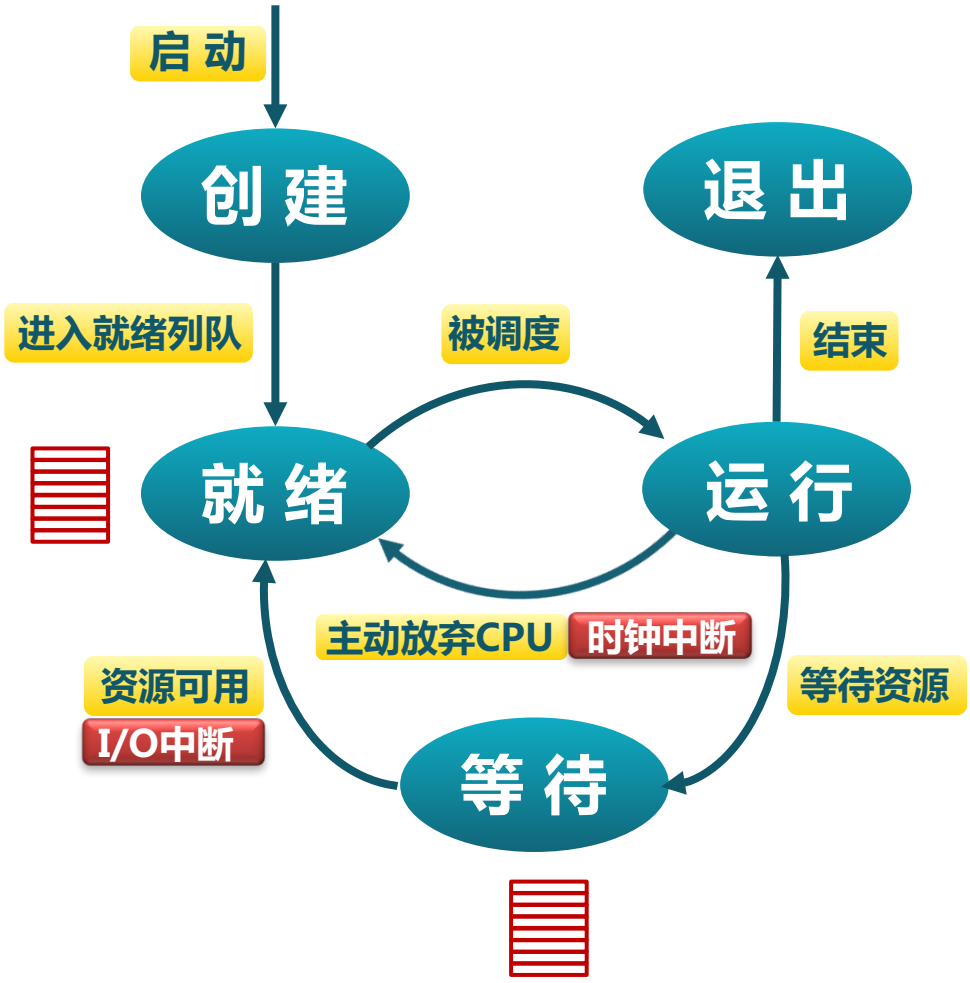
- **调度器如何工作，三个步骤：**
  - 保存当前进程/线程状态（PCB/TCB）
  - 选择下一个待运行的进程/线程
  - 分派（加载并跳转到相应PCB/TCB）
- 非抢占式调度与抢占式调度的区别





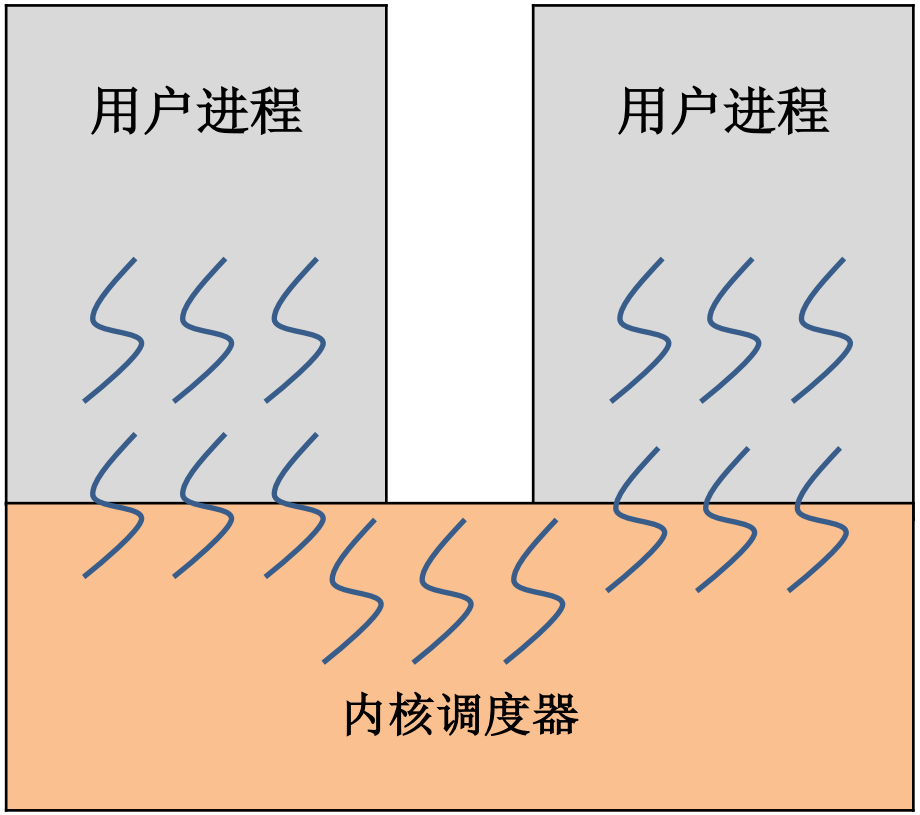
什么时候调度？

- 进程/线程创建
  - 父进程、子进程谁先执行
- 进程/线程退出
  - 下一个进程/线程是谁
- I/O阻塞、同步
  - 主动让出CPU，哪个进程/线程执行
- I/O中断
  - 中断处理完，哪个进程/线程执行
- 时间中断
  - 时间片用完，哪个进程/线程执行



进程调度 vs. 线程调度

- 内核拥有自己的地址空间，并与所有的进程共享
- **内核包含**
  - 引导加载程序
  - BIOS
  - 核心驱动
  - 线程
  - 调度器
- **调度器**
  - 使用就绪队列来存放所有的就绪线程
  - 在相同的地址空间进行调度（thread context switch）
  - 在新的地址空间进行调度（process context switch）



内容提要

- CPU调度基础
- CPU调度算法

调度算法

- 调度器的工作
  - 保存当前进程/线程状态（PCB/TCB）
  - **选择下一个待运行的进程/线程**
  - 分派（加载并跳转到相应PCB/TCB）

调度准则

- 假设
  - 一个用户运行一个程序，一个程序创建一个线程
  - 程序之间是独立的
- 常用指标
  - 吞吐率：每秒处理请求数
  - 响应时间（等待时间）：从提交一个请求到产生响应所用时间
  - 周转时间：从作业提交到作业完成的时间间隔
  - 公平性：每个程序是否都有执行机会，防止饥饿

调度准则

- 批处理和实时交互系统设计目标
  - 保证公平性
    - 每个作业都有机会运行； 没有人会“饥饿”
  - 最大化CPU资源利用率
    - 不包括idle进程
  - 最大化吞吐率
    - 操作数/秒（最小化开销，最大化资源利用率）
  - 最小化周转时间
    - 批处理作业：执行时间（从提交到完成）
  - 缩短响应时间
    - 交互式作业：响应时间（例如，键盘打字）
  - 均衡性
    - 满足用户需求
    - 提升计算机系统各部件利用率

调度准则

- 不同类型计算机系统的需求不一样
  - 服务器
    - 吞吐率
    - 响应时间
    - 公平性
  - 个人计算机
    - 响应时间
  - 工业控制计算机
    - 实时性

先到先服务（FCFS）算法

- 什么是先到先服务？
  - 一直运行到结束
  - 一直运行到阻塞或者主动放弃CPU
  - 用于非抢占式调度
- 例子1
  - P1 = 24s, P2 = 3s, 且P3 = 3s，同时提交，顺序运行
  - 平均周转时间 =（24 + 27 + 30）/ 3 = 27

- 例子2
  - 同样的进程，但是以不同的顺序运行：P2，P3，P1
  - 平均周转时间 =（3 + 6 + 30）/ 3 = 13

先到先服务算法分析

- 优点
  - 实现简单
- 缺点
  - 平均周转时间波动较大
    - 短进程（作业）排到长进程（作业）后面
    - 平均响应时间如何？
  - I/O资源利用率较低
    - CPU密集型应用导致I/O设备空闲，I/O资源利用率低
    - 如果I/O密集是否会导致CPU利用率低？

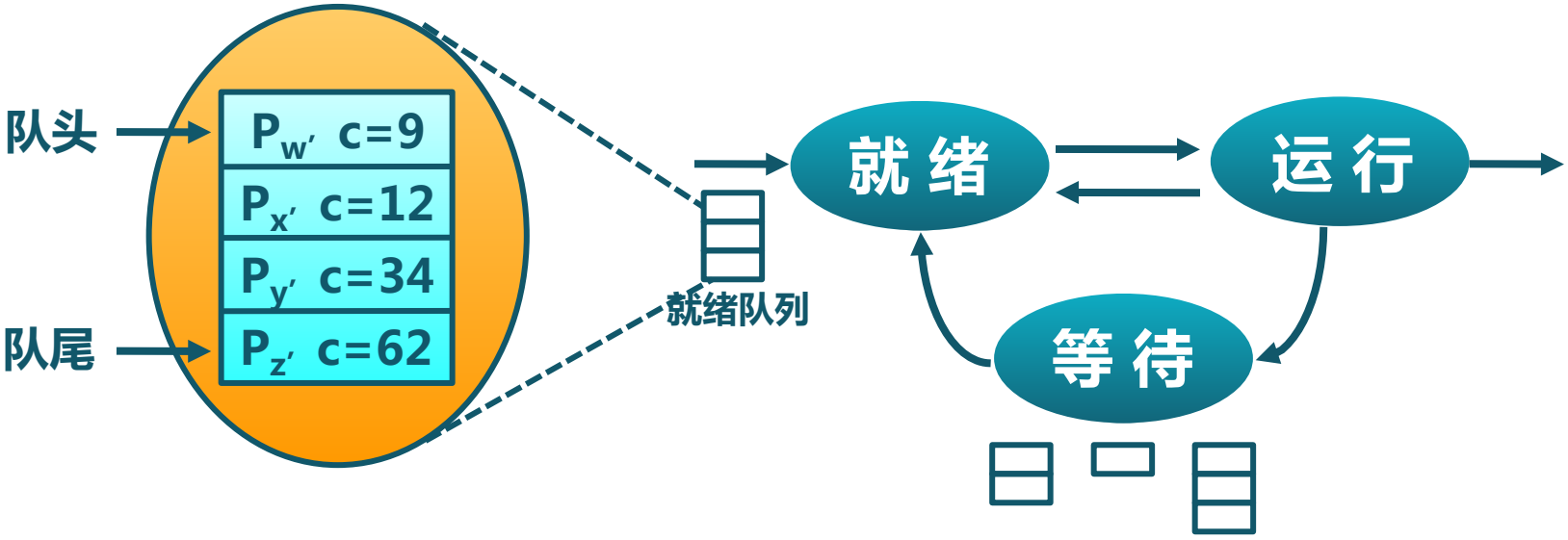
最短时间优先

- 最短时间优先（Short Time to Complete First, STCF）
  - 非抢占
- 例子
  - P1 = 6s, P2 = 8s, P3 = 7s, P4 = 3s
  - 所有作业同时到达
  - 平均响应时间 =（0 + 3 + 9 + 16）/ 4 = 7

FCFS算法平均响应时间 =（0 + 6 + 14 + 21）/ 4 = 10.25

最短剩余时间优先

- 最短剩余时间优先（Short Remaining Time to Complete First, SRTCF）
  - 抢占式调度
  - 选择就绪队列中剩余时间最短进程占用CPU进入运行状态
  - 就绪队列按剩余时间来排序

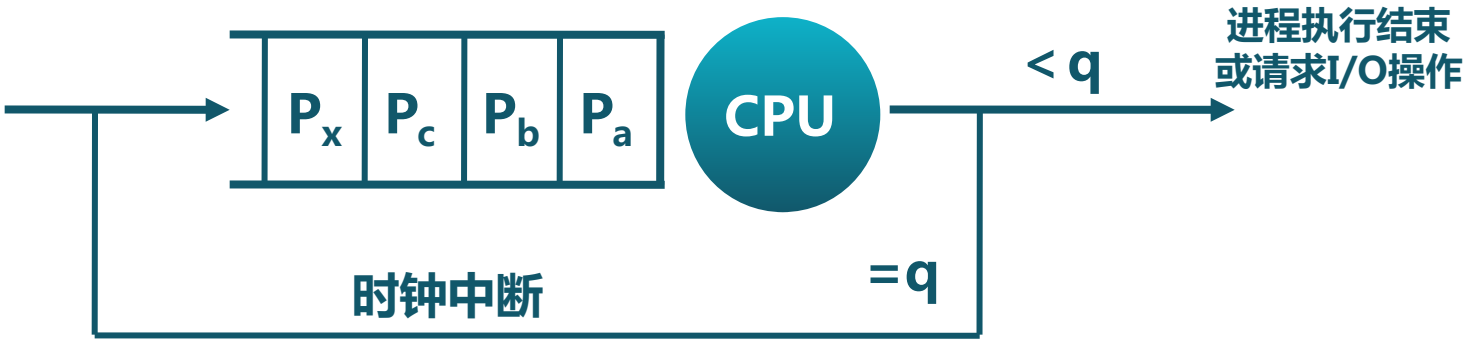


STCF和SRTCF分析

- 优点
  - 平均响应时间短
- 可能会造成饥饿
  - 连续的短进程流会使长进程无法获得CPU资源
- 挑战
  - 需要预知未来：如何预估下一个CPU计算的持续时间？
  - 简单的解决办法：询问用户

时间片轮转算法（RR，Round Robin）

- 和FCFS算法类似，但是增加了时间片
- 时间片结束时，调度器按FCFS算法切换到下一个就绪进程
- 轮转调度是抢占式调度
- 多用于交互式系统

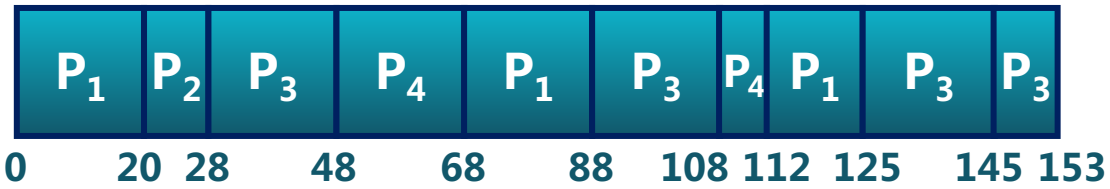


时间片为20的轮转算法示例

- 示例: 4个进程的执行时间如下

P1	53
P2	8
P3	68
P4	24

- 甘特图



等待时间 $P_1=(68-20)+(112-88)=72$

$P_2=(20-0)=20$

$P_3=(28-0)+(88-48)+(125-108)=85$

$P_4=(48-0)+(108-68)=88$

平均等待时间 = (72+20+85+88)/4=66.25

时间片长度选择

- 大时间片
  - 等待时间过长
  - 极端情况下退化为FCFS
- 小时间片
  - 响应时间快
  - 产生大量上下文切换，影响系统吞吐
- 经验规则
  - 选择一个合适的时间片，使上下文切换开销处于1%以内

不同时间片的调度效果

- 示例: 4个进程的执行时间如下

P1	53
P2	8
P3	68
P4	24

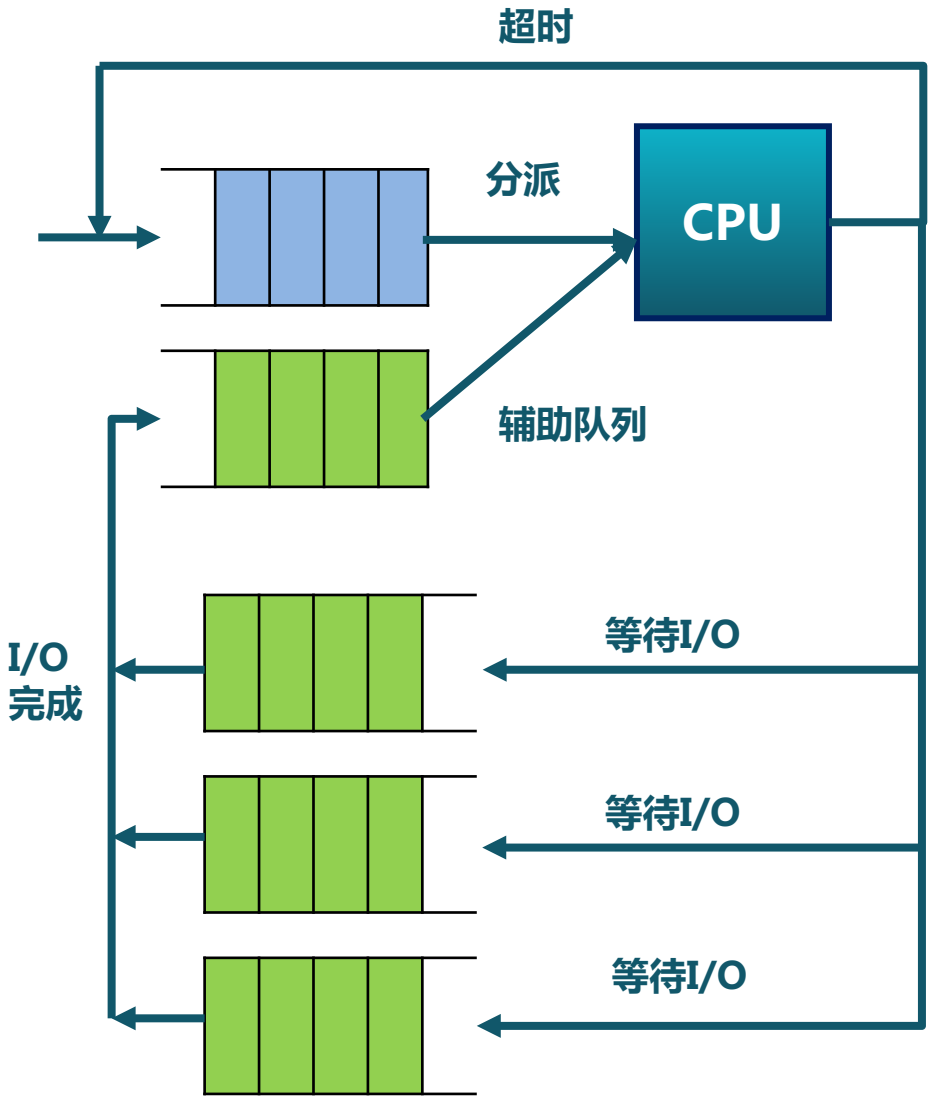
- 假设上下文切换开销为0，不同时间片以及FCFS对应的调度效果

时间片	P_1	P_2	P_3	P_4	平均等待时间
RR(q=1)	84	22	85	57	62
RR(q=5)	82	20	85	58	61.25
RR(q=8)	80	8	85	56	57.25
RR(q=10)	82	10	85	68	61.25
RR(q=20)	72	20	85	88	66.25
BestFCFS	32	0	85	8	31.25
WorstFCFS	68	145	0	121	83.5

=最短时间优先

虚拟轮转算法（Virtual Round Robin）

- 引入辅助队列
  - FIFO（先入先出）
- I/O密集型进程
  - 进入辅助队列（而不是就绪队列）以备调度
- 引入优先级
  - 辅助队列比就绪队列有更高的优先级

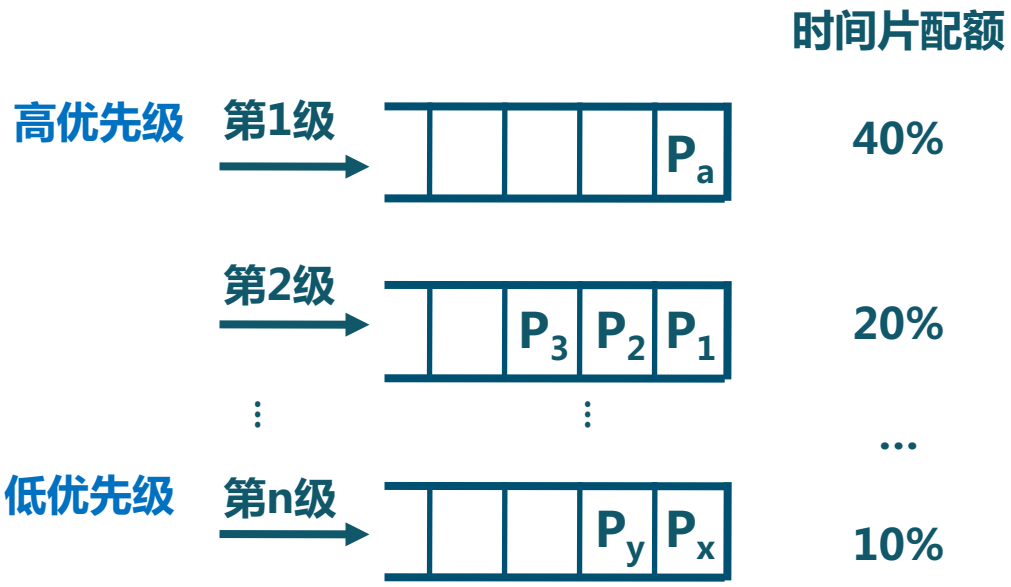


多级队列（MQ）与优先级

- 将就绪队列分为多个独立的子队列，每个队列可有自己的调度算法
  - 前台任务（交互式）-RR，后台任务（批处理）-FCFS
- 队列之间
  - 最高优先级优先
    - 固定优先级
    - 先调度高优先级，再调度低优先级
    - 可能导致饥饿
  - 潜在问题：优先级反转
    - 高优先级进程所需资源被低优先级进程持有，等待低优先级执行
    - 中优先级进程优先执行
    - 解决方法：提升低优先级进程的优先级

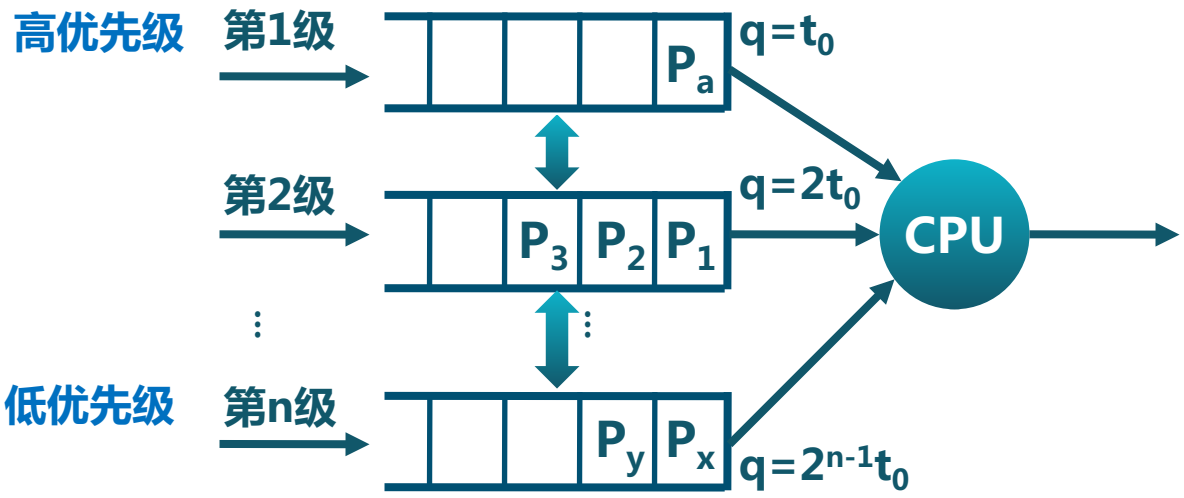
多级队列（MQ）与优先级

- 将就绪队列分为多个独立的子队列，每个队列可有自己的调度算法
  - 前台任务（交互式）-RR，后台任务（批处理）-FCFS
- 队列之间
  - 时间片轮转：每个队列得到一个确定能调度其进程的CPU总时间，队列间按照时间片调度



多级反馈队列(MLFQ)算法

- 进程可在不同队列间移动的多级队列算法
- 特征
  - 时间片大小随优先级级别增加而减小
  - 进程在当前的时间片没有完成，则降到下一个优先级
  - CPU密集型进程的优先级下降很快，I/O密集型进程停留在高优先级

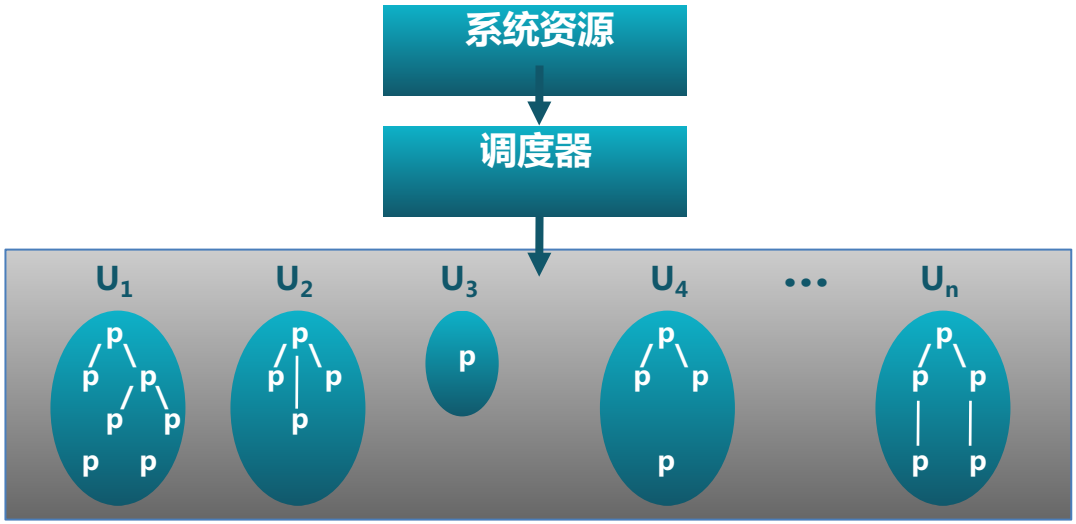


彩票调度

- 动机
  - SRTCF可以保证平均响应延迟，但是对长进程不公平
- 彩票方法
  - 给每个作业一定数量的彩票
  - 随机抽取一张中奖彩票（winning ticket），中奖的作业获得CPU
  - 为了近似SRTCF，给短作业更多的彩票
  - 为了避免“饥饿”，给每个作业至少一张彩票
  - 相互合作的进程可以交换彩票

公平共享调度(FSS, Fair Share Scheduling)

- FSS控制用户对系统资源的访问
  - 一些用户组比其他用户组更重要
  - 配额管理
    - 保证不重要的组无法垄断资源（使用Limit）
    - 重要的组保证资源（使用reservation）
    - 未使用的资源按比例分配（使用proportional）
  - 动态调度
    - 没有达到资源使用率目标的组获得更高的优先级

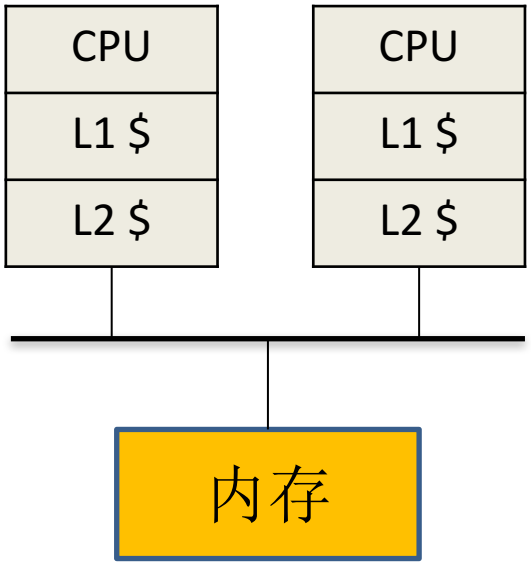


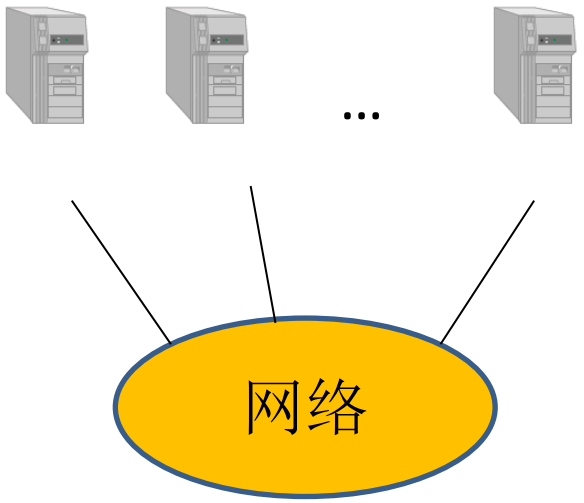
调度算法总结

- 调度算法比较

算法	适用系统	平均响应时间	公平性	潜在问题
FCFS	批处理	长	可能造成饥饿	
STCF	批处理	短	长进程可能饥饿	需要预测作业执行时间
SRTCF	批处理	短	长进程可能饥饿	需要预测作业执行时间
轮询算法	交互式	短，IO进程响应时间较长	公平对待，	时间片小会导致吞吐率低
虚拟轮询算法	交互式	短	对IO密集进程友好	
多级队列优先级调度	交互式	低优先级队列长	可能造成饥饿	优先级反转
多级反馈队列	交互式	短	兼顾长短作业	
彩票算法	交互式	短	兼顾长短作业	

多处理器和集群





- **多处理器架构**
  - Cache一致性
  - 单独一个OS
- **集群**
  - 分布式内存
  - 每个“盒子”各一个OS

多处理器/集群调度

- 设计问题
  - 进程/线程到处理器分配
- 协同调度（co-scheduling）
  - 一个进程的多个线程共同运行
  - 一个应用的多个进程共同运行
- 处理器分配
  - 静态分配
    - 线程会在一个专用的处理器上运行直到完成，线程绑定
    - 调度开销小
    - 负载容易不均衡
  - 动态分配
    - 进程可以运行在任一空闲的处理器
    - 调度开销大
    - 负载容易均衡

实时调度

- 两种类型的实时
  - 硬实时（hard deadline）
    - 必须满足，否则会导致错误
  - 软实时（soft deadline）
    - 大多时候满足，没有强制性
- 实时任务

- 周期性实时任务

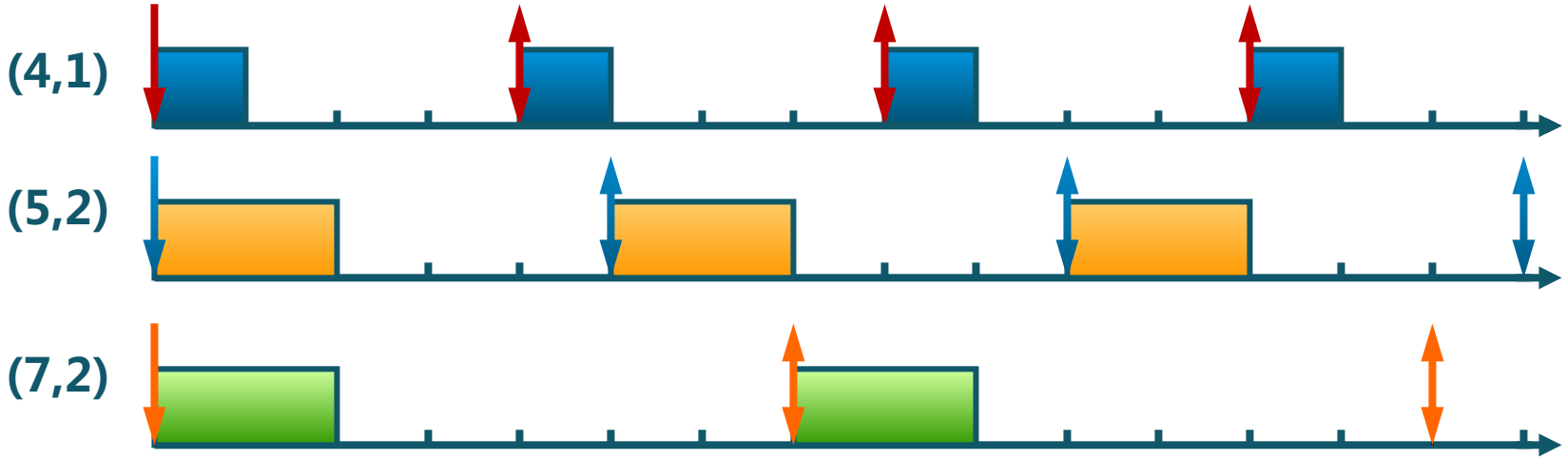
实时调度

- 接纳控制/准入控制（Admission Control）
  - 只有当系统能够保证所有进程的实时性的前提下，新的实时进程才会被接纳/准入
  - 如果满足下面的条件，作业就是可调度的：

$$\sum \frac{C_i}{T_i} \leq 1$$

其中C_i = 计算时间，T_i = 周期

- **示例：1/4 + 2/5 + 2/7 = 131/140 < 1**



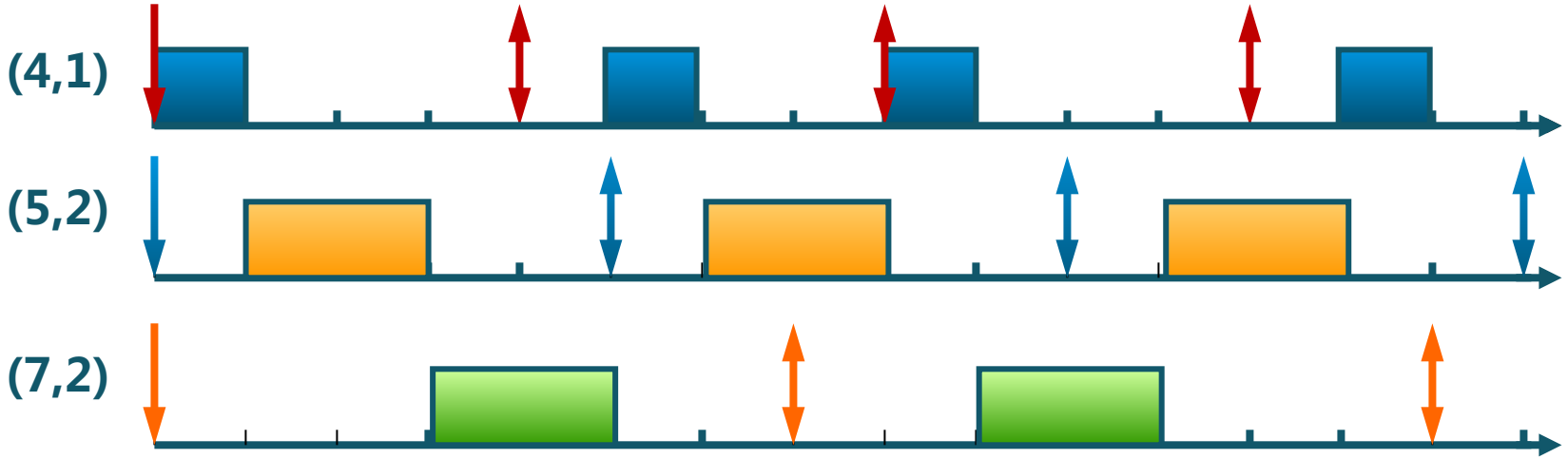
实时调度

- 接纳控制/准入控制（Admission Control）
  - 只有当系统能够保证所有进程的实时性的前提下，新的实时进程才会被接纳/准入
  - 如果满足下面的条件，作业就是可调度的：

$$\sum \frac{C_i}{T_i} \leq 1$$

其中C_i = 计算时间，T_i = 周期

- **示例：1/4 + 2/5 + 2/7 = 131/140 < 1**



速率单调调度

- 假设
  - 每个周期性进程必须在其周期内完成
  - 进程之间没有依赖关系
  - 每个进程在每个周期内需要的CPU时间相同
  - 非周期性进程没有截止日期
  - 进程抢占瞬时发生（没有开销）
- 基本思想
  - 给每个进程分配一个固定的优先级=出现频率
  - 运行最高优先级的进程
  - 证明是最优的
- 例子
  - P1每30ms运行一次，所以优先级是33（33次/秒）
  - P2每50ms运行一次，所以优先级是20（20次/秒）

最早最终时限优先调度（Earliest Deadline First Scheduling, EDF）

- 假设
  - 当进程需要CPU时间时，它会宣布其最终时限，需要的CPU时间可以变化
  - 不一定是周期性进程
- EDF的基本思想
  - 根据最终时限对就绪的进程进行排序
  - 运行列表中的第一个进程（最早最终时限优先）
  - 当新的进程就绪时，并且其最终时限快来临时，它会抢占当前进程
  - 和SRTCF的区别？
- 例子
  - P1需要在第30s之前结束，P2需要在第40s之前结束，P3需要在第50s之前结束
  - P1先运行

4.3 BSD多队列调度

- “1秒钟”抢占
  - 进程如果在1秒内没有阻塞或者完成，则会被抢占
- 优先级每秒都会重新计算
  - P_i = base + CPU_i / 2 + nice, where CPU_i = (U_i + CPU_{i-1}) / 2
  - base是进程的基础优先级
  - U_i是进程在第i段时间间隔的利用率
- 优先级
  - 交换器（swapper）
  - 块I/O设备控制
  - 文件操作
  - 字符I/O设备控制
  - 用户进程

Linux中的调度

- 分时共享调度
  - 每个进程都会有优先级和credits
  - I/O事件会提升优先级
  - 拥有最多credits的进程会优先运行
  - 时间中断会减少进程的credits
  - 如果所有进程的credits都耗尽了，内核会重新给进程分配：credits = credits/2 + priority
- 实时调度
  - 软实时
  - 内核不会被用户代码抢占

Windows中的调度

- 分类和优先级
  - 实时类： 16个静态优先级
  - 可变类： 16个变化的优先级
    - 如果进程用完了所分配的额度，降低其优先级
    - 如果进程在等待I/O，增高其优先级
- 优先级驱动的调度器
  - 对于实时类，用轮转算法进行调度（within each priority）
  - 对于可变类，用多队列进行调度
- 多处理器调度
  - 有N个处理器，将N-1个最高优先级的线程运行在N-1个处理器上，剩下的线程运行在1个处理器上
  - 线程会等待属于它亲和力集合（affinity set）的处理器

总结

- 调度基础
  - 抢占式调度与非抢占式调度
  - 中断在调度中的作用，中断的处理
- 调度算法
  - 优化问题，不同的系统决定了不同的调度目标
  - 批处理系统
    - FCFS简单易实现
    - STCF和SRTCF可以获得最小的平均响应时间
  - 交互式系统
    - 轮询算法及改进算法有利于公平性，较小的时间片有利于提高I/O利用率
    - 多队列与优先级调度及其变种，普遍存在于多个系统中
    - 彩票调度的灵活性很好
  - 实时调度依赖于接纳/准入控制（admission control）