
2025第12次课堂小测试


2025第13次课堂小测试

操作系统 Operating System

第四章 进程管理

4.3 调度

原仓周

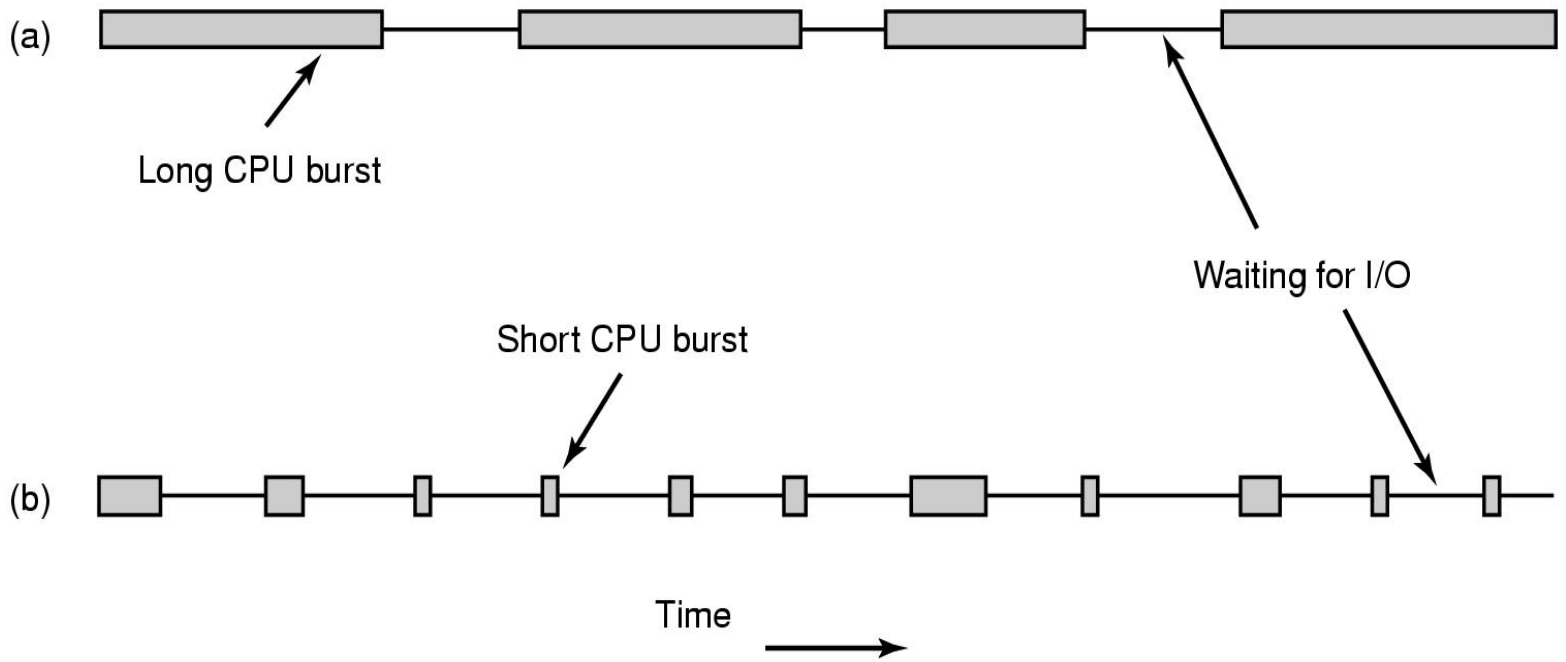
yuancz@buaa.edu.cn

内容提要

- 基本概念
  - 调度（选择，分派），调度的时机
  - 调度的分类，调度的性能准则（用户，系统）
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度

为什么要调度

- 处理机管理的工作是对CPU资源进行合理的分配使用，以提高处理机利用率，并使各用户公平地得到处理机资源
- 这里的主要问题是处理机调度算法和调度算法特征分析



CPU调度

什么是CPU调度?

- CPU 调度的任务是控制、 协调多个进程对 CPU 的竞争
  - 也就是按照一定的策略（调度算法），从就绪队列中 选择一个进程，并把 CPU 的控制权交给被选中的进程

CPU调度的场景

- N 个进程就绪，等待上 CPU 运行
- M个CPU，M≥1
- OS需要决策，给哪个进程分配哪个 CPU

要解决的问题

- **WHEN:**
  - 何时需要调度
    - 进程调度的时机
- **HOW:**
  - 如何分派到CPU上
    - CPU切换过程（进程的上下文切换）
- **WHAT:**
  - 按什么原则选择下一个要执行的进程
    - 进程调度算法

何时调度（When）

- 当一个新的进程被创建时
  - 是执行新进程还是继续执行父进程?
- 当一个进程运行完毕时
- 当一个进程由于I/O、信号量或其他的某个原因被阻塞时
- 当一个I/O中断发生时，表明某个I/O操作已经完成，而等待该I/O操作的进程转入就绪状态
- 在分时系统中，当一个时钟中断发生时

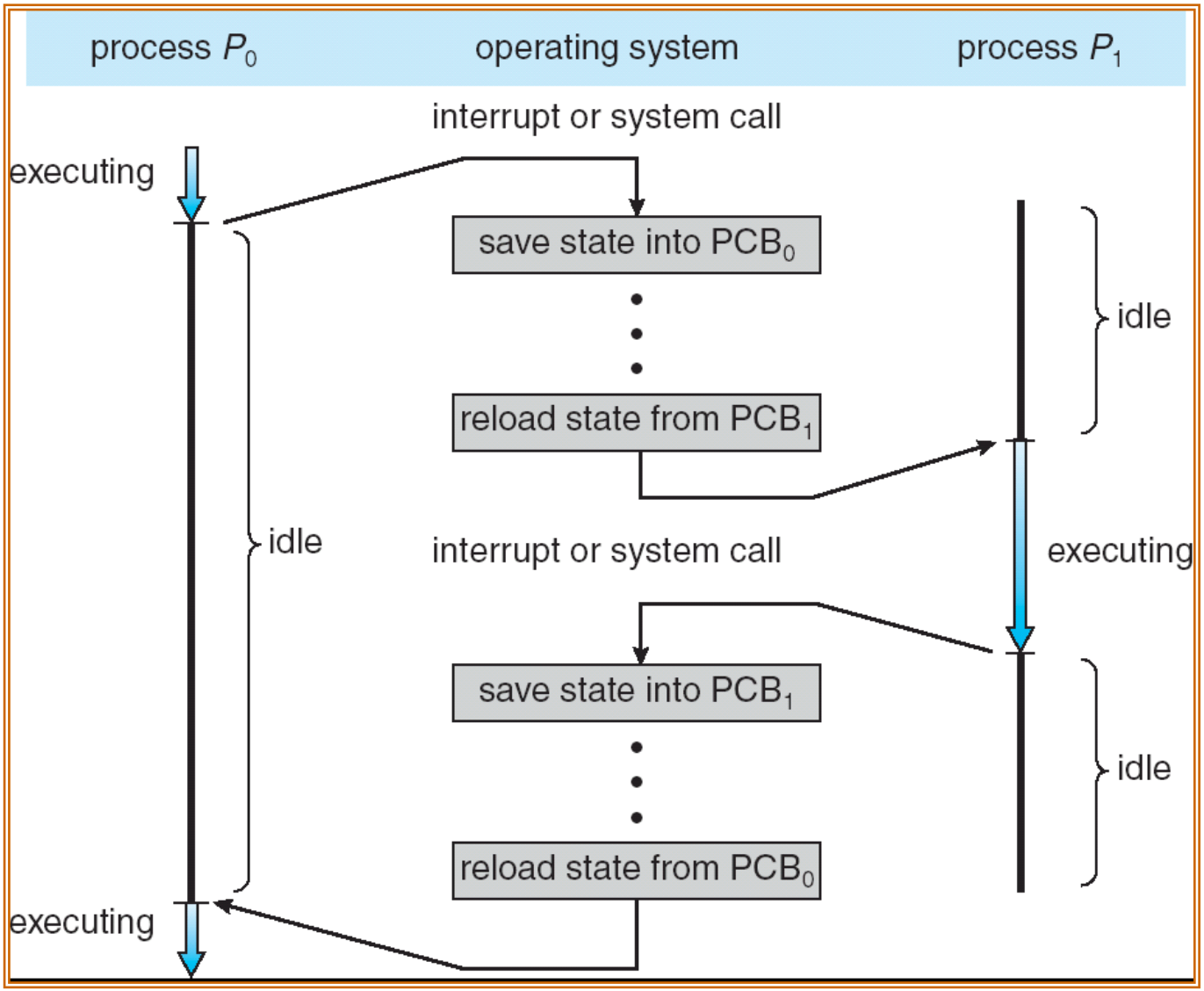
何时切换（When）

- 只要OS取得对CPU的控制，进程切换就可能发生:
  - 用户调用：来自程序的显式请求(如：打开文件)，该进程多半会被阻塞
  - 陷阱：最末一条指令导致出错，会引起进程移至退出状态
  - 中断：外部因素影响当前指令的执行，控制被转移至中断处理程序

CPU切换过程（How）

- 在进程（上下文）中切换的步骤
  - 保存处理器的上下文，包括程序计数器和其它寄存器
  - 用新状态和其它相关信息更新正在运行进程的PCB
  - 把进程移至合适的队列-就绪、阻塞
  - 选择另一个要执行的进程
  - 更新被选中进程的PCB
  - 从被选中进程中重装入CPU 上下文

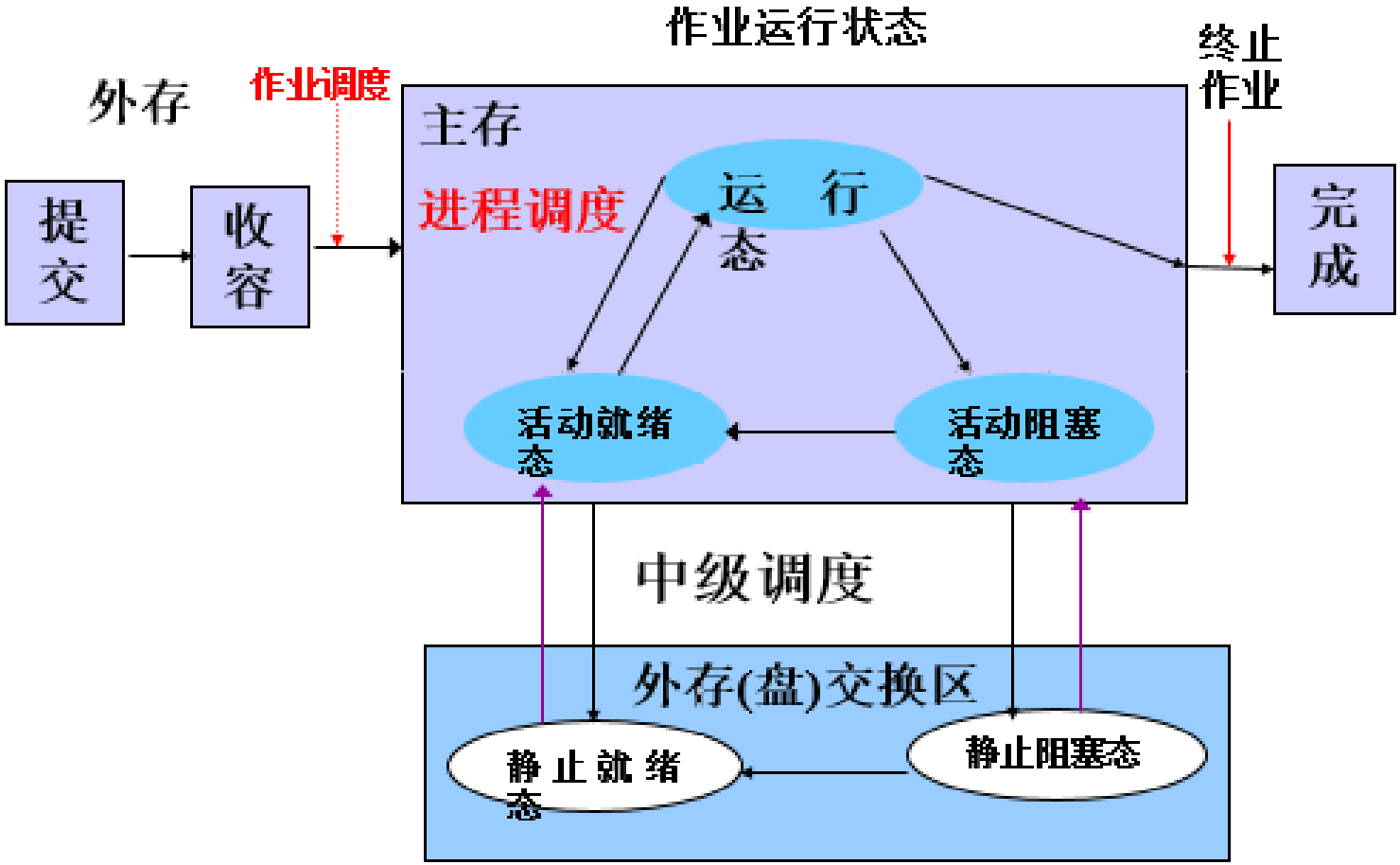
CPU切换过程（How）


process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
idle
reload state from PCB_1
idle
interrupt or system call
executing
save state into PCB_1
idle
reload state from PCB_0
executing

调度的分类

- 高级调度
- 中级调度
- 低级调度

CPU三级调度


作业运行状态
外存
作业调度
主存
进程调度
运 行 态
终止作业
提交
收容
完成
活动就绪态
活动阻塞态
中级调度
外存(盘)交换区
静止就绪态
静止阻塞态

高级调度和中级调度

- 高级调度：又称为“宏观调度”、“作业调度”
  - 从用户工作流程的角度，一次提交的若干个作业，对每个作业进行调度
  - 时间上通常是分钟、小时或天
    - 接纳多少个作业
    - 接纳那些作业
- 中级调度：又称为“内外存交换”
  - 从存储器资源的角度。将进程的部分或全部换出到外存上，将当前所需部分换入到内存
  - 指令和数据必须在内存里才能被CPU直接访问

低级调度

- 低级调度：又称为“微观调度”“进程或线程调度”
  - 从CPU资源的角度，执行的单位。时间上通常是毫秒
  - 因为执行频繁，要求在实现时达到高效率
- 非抢占式
- 抢占式
  - 时间片原则
  - 优先权原则
  - 短作业（进程）优先

调度的性能准则

- 从不同的角度来判断处理机调度算法的性能，如：
  - 用户的角度
  - 处理机的角度
  - 算法实现的角度
- 实际的处理机调度算法选择是一个综合的判断结果

面向用户的调度性能准则(1/2)

- 周转时间：作业从提交到完成（得到结果）所经历的时间
  - 包括：在收容队列中等待，CPU上执行，就绪队列和阻塞队列中等待，结果输出等待——批处理系统。
  - 外存等待时间、就绪等待时间、CPU执行时间、I/O操作时间
  - 平均周转时间、带权平均周转时间（T/Ts)
- 响应时间：用户输入一个请求（如击键）到系统给出首次响应（如屏幕显示）的时间——分时系统

面向用户的调度性能准则(2/2)

- 截止时间：开始截止时间和完成截止时间——实时系统，与周转时间有些相似
- 优先级：可以使关键任务达到更好的指标
- 公平性：不因作业或进程本身的特性而使上述指标过分恶化。如长作业等待很长时间

面向系统的调度性能准则

- 吞吐量：单位时间内所完成的作业数，跟作业本身特性和调度算法都有关系——批处理系统
  - 平均周转时间不是吞吐量的倒数，因为并发执行的作业在时间上可以重叠。如：在2小时内完成4个作业，而平均周转时间是1.25小时，则吞吐量是2个作业/小时
- 处理机利用率：——大中型主机
- 各种资源的均衡利用：如CPU繁忙的作业和I/O繁忙（指次数多，每次时间短）的作业搭配——大中型主机

调度算法本身的调度性能准则

- 易于实现
- 执行开销比小

内容提要

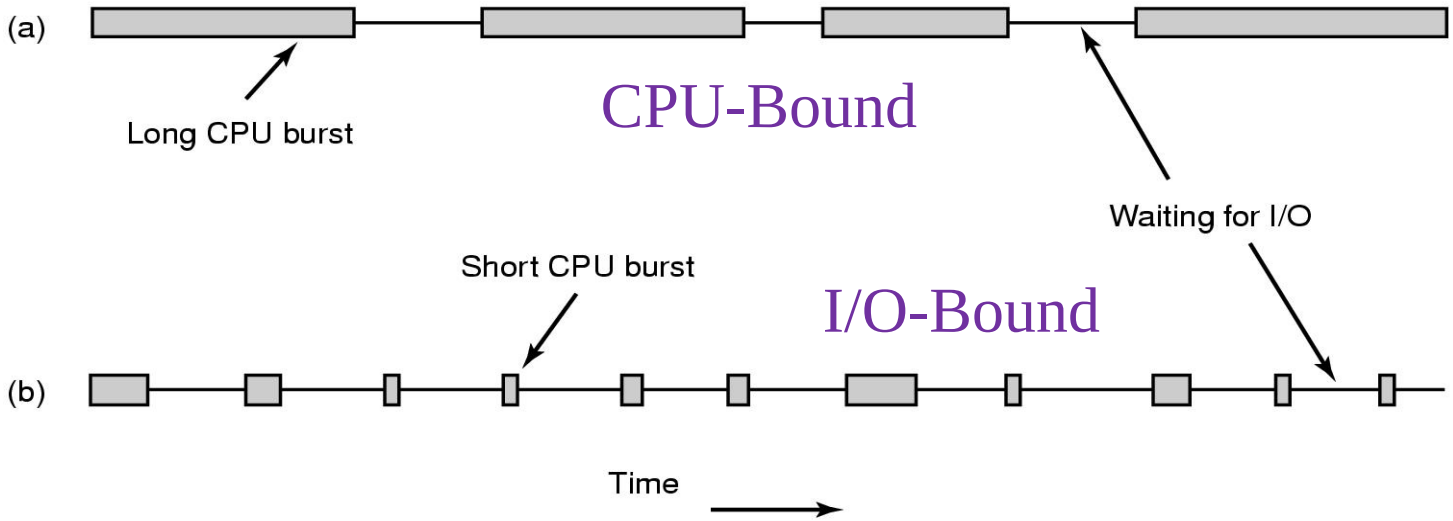
- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度

设计调度算法要点

- 进程的分类
- 进程优先级（数）
- 进程优先级就绪队列的组织
- 抢占式调度与非抢占式调度
- 时间片

进程的分类（第一种）

- I/O Bound（I/O密集型）
  - 频繁的进行I/O，通常会花费很多时间等待I/O操作完成
- CPU bound（CPU密集型）
  - 计算量大，需要大量的CPU时间。



进程的分类（第二种）

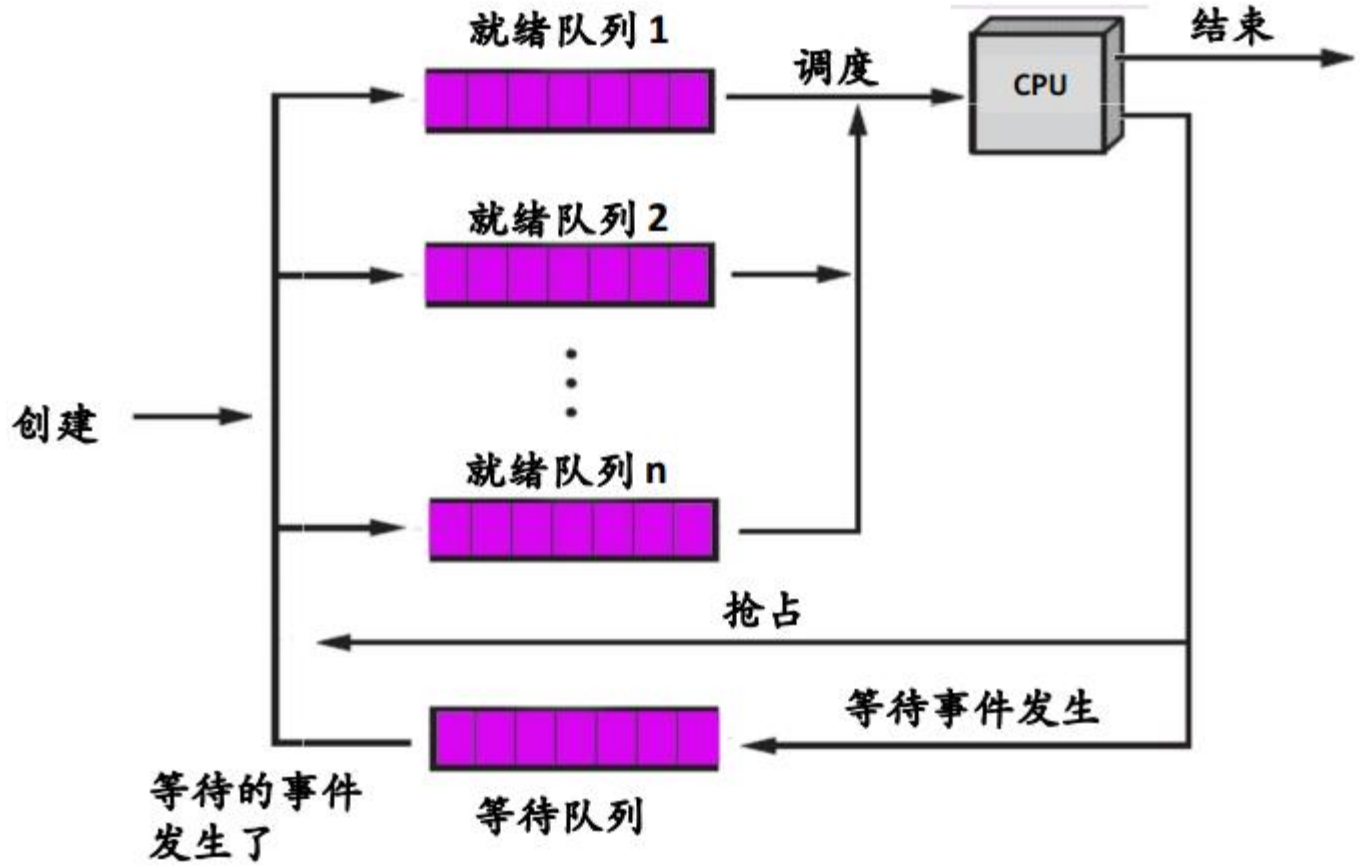
- 批处理进程（Batch Process)
  - 无需与用户交互，通常在后台运行
  - 不需很快的响应
  - 典型的批处理程序：编译器、科学计算
- 交互式进程（Interactive Process)
  - 与用户交互频繁，因此要花很多时间等待用户输入
  - 响应时间要快，平均延迟要低于50~150ms
  - 典型的交互式进程：Word、触控型GUI
- 实时进程（Real-time Process)
  - 有实时要求，不能被低优先级进程阻塞
  - 响应时间要短且要稳定
  - 典型的实时进程：视频/音频、控制类

进程优先级（数）

- 优先级和优先数是不同的，优先级表现了进程的重要性和紧迫性，优先数实际上是一个数值，反映了某个优先级
- 静态优先级
  - 进程创建时指定，运行过程中不再改变
- 动态优先级
  - 进程创建时指定了一个优先级，运行过程中可以动态变化。如：等待时间较长的进程可提升其优先级

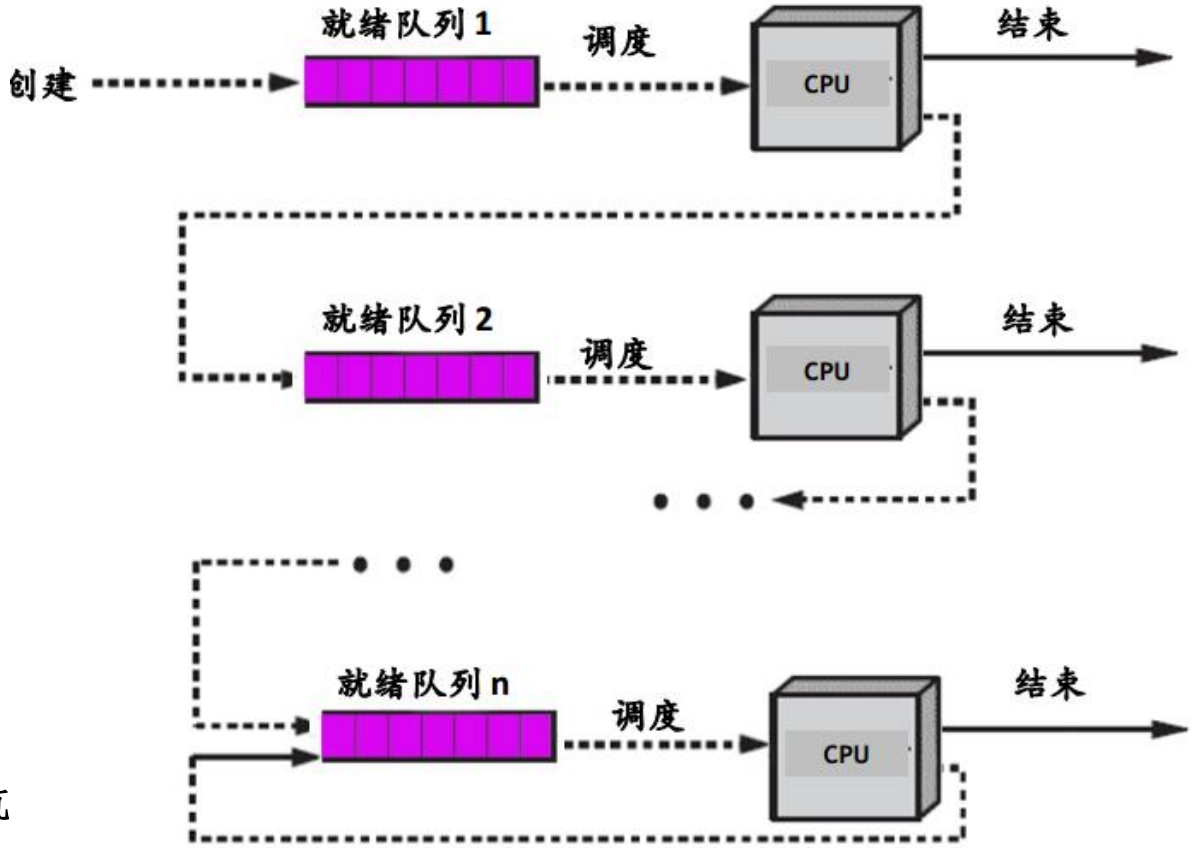
进程就绪队列组织

- 按优先级排队方式
  - 创建多个进程后按照不同的优先级进行排列，CPU调度优先级较高的进程进行执行



进程就绪队列组织

- 另一种方式
  - 所有进程创建之后都进入到第一级就绪队列，随着进程的运行，可能会降低某些进程的优先级，如某些进程的时间片用完了，那么就会将其降级



占用CPU的方式

- 不可抢占式方式
  - 一旦处理器分配给一个进程，它就一直占用处理器，直到该进程自己因调用原语操作或等待I/O等原因而进入阻塞状态，或时间片用完时才让出处理器，重新进行
- 抢占式方式
  - 就绪队列中一旦有优先级高于当前运行进程优先级的进程存在时，便立即进行进程调度，把处理器转给优先级高的进程

时间片（Time slice或quantum）

- 一个时间段，分配给调度上CPU的进程，确定了允许该进程运行的时间长度
- 那么如何选择时间片？有一下需要考虑的因素
  - 进程切换的开销
  - 对响应时间的要求
  - 就绪进程个数
  - CPU能力
  - 进程的行为

内容提要

- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度

吞吐量、平均等待时间和平均周转时间

- 吞吐量 $=\frac{\text{作业数}}{\text{总执行时间}}$，即单位时间CPU完成的作业数量
- 周转时间(Turnover Time)=完成时刻 – 提交时刻
- 带权周转时间=周转时间/服务时间（执行时间）
- 平均周转时间 $=\frac{\text{作业周转时间之和}}{\text{作业数}}$
- 平均带权周转时间 $=\frac{\text{作业带权周转时间之和}}{\text{作业数}}$

批处理系统中常用的调度算法

- 先来先服务
  - (FCFS：First Come First Serve)
- 最短作业优先
  - (SJF：Shortest Job First)
- 最短剩余时间优先
  - (FRTF：Shortest Remaining Time First)
- 最高响应比优先
  - (HRRF：Response Ratio First)

先来先服务FCFS

- 这是最简单的调度算法，按先后顺序调度
  - 按照作业提交或进程变为就绪状态的先后次序，分派CPU
  - 当前作业或进程占用CPU，直到执行完或阻塞，才出让CPU（非抢占方式）
  - 在作业或进程唤醒后（如I/O完成），并不立即恢复执行，通常等到当前作业或进程出让CPU
  - 最简单的算法
- FCFS的特点
  - 比较有利于长作业，而不利于短作业
  - 有利于CPU繁忙的作业，不利于I/O繁忙的作业

短作业优先SJF

- 又称为“短进程优先”SPN(Shortest Process Next)；这是对FCFS算法的改进，其目标是减少平均周转时间
  - 对预计执行时间短的作业（进程）优先分派处理机。通常后来的短作业不抢先正在执行的作业

SJF的特点

- 优点：
  - 比FCFS改善平均周转时间和平均带权周转时间，缩短作业的等待时间
  - 提高系统的吞吐量
- 缺点：
  - 对长作业非常不利，可能长时间得不到执行
  - 未能依据作业的紧迫程度来划分执行的优先级
  - 难以准确估计作业（进程）的执行时间，从而影响调度性能

示例

有三道作业，它们的提交时间和运行时间见下表

作业号	提交时间/时	运行时间/h
1	10:00	2
2	10:10	1
3	10:25	0.25

试给出在下面两种调度算法下，作业的执行顺序、平均周转时间和带权周转时间

（1）先来先服务FCFS调度算法

（2）短作业优先SJF调度算法

示例

采用FCFS调度算法时，作业的执行顺序是作业1 ->作业2 ->作业3。由此可得到运行表

作业号	提交时刻/时	运行时间/h	开始时刻/时	完成时刻/时
1	10:00	2	10:00	12:00
2	10:10	1	12:00	13:00
3	10:25	0.25	13:00	13:15

那么，平均周转时间为

$T=(\sum Ti)/3=[(12-10)+(13-10:10)+(13:15-10:25)]/3$
$=[2+2.83+2.83]/3 =2.55h$

带权平均周转时间为

$W=[\sum(Ti/Tir)]/3=（2/2+2.83/1+2.83/0.25）/3=5.05h$

示例

在SJF调度算法下，作业的执行顺序是作业1 -> 作业3-> 作业2；由此得运行表

作业号	提交时刻/时	运行时间/h	开始时刻/时	完成时刻/时
1	10:00	2	10:00	12.00
2	10:10	1	12:15	13:15
3	10:25	0.25	12:00	12:15

那么，平均周转时间为

T=（∑Ti）/3=[(12-10)+(13:15-10:10)+(12:15-10:25)]/3
=[2+3.08+1.83]/3=2.3h

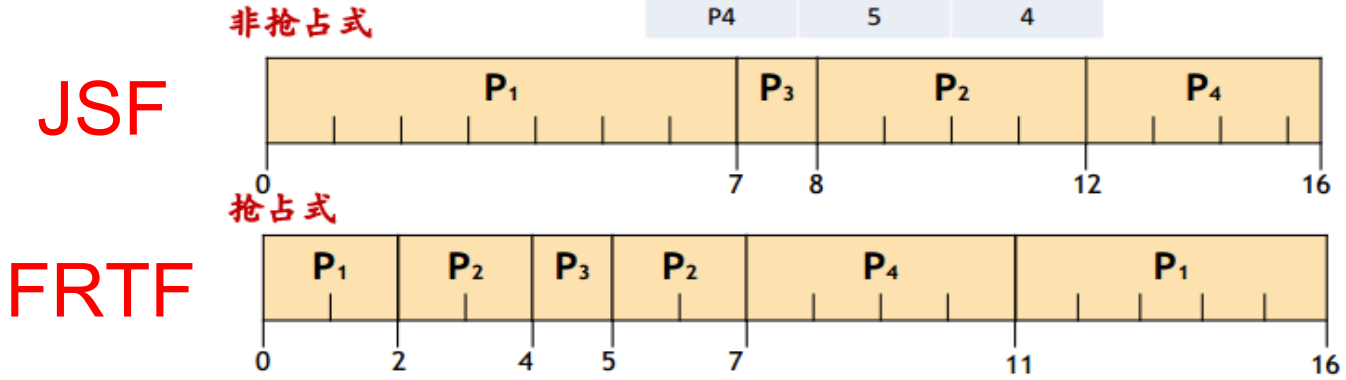
带权平均周转时间为

W=[∑（Ti/Tir）]/3=(2/2+3.08/1+1.83/0.25)/3=3.8h

最短剩余时间优先FRTF

- 将短作业优先进行改进，改进为抢占式，这就是最短剩余时间优先算法了，即一个新就绪的进程比当前运行进程具有更短的完成时间，系统抢占当前进程，选择新就绪的进程执行

进程	到达时刻	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4



缺点：源源不断的短任务到来，可能使长的任务长时间得不到运行，导致产生“饥饿”现象

最高响应比优先HRRF

- HRRF算法实际上是FCFS算法和SJF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的等待时间过长，改善了调度性能

- 在每次选择作业投入运行时，先计算后备作业队列中每个作业的响应比RP，然后选择其值最大的作业投入运行

- RP值定义为：$RP = \dfrac{\text{已等待时间} + \text{要求运行时间}}{\text{要求运行时间}}$

$$= 1 + \frac{\text{已等待时间}}{\text{要求运行时间}}$$

最高响应比优先HRRF

- 响应比的计算时机：
  - 每当调度一个作业运行时，都要计算后备作业队列中每个作业的响应比，选择响应比最高者投入运行
- 响应比最高优先（HRRF）算法效果：
  - 短作业容易得到较高的响应比
  - 长作业等待时间足够长后，也将获得足够高的响应比
  - 饥饿现象不会发生
- 缺点：
  - 每次计算各道作业的响应比会有一定的时间开销，性能比SJF略差

内容提要

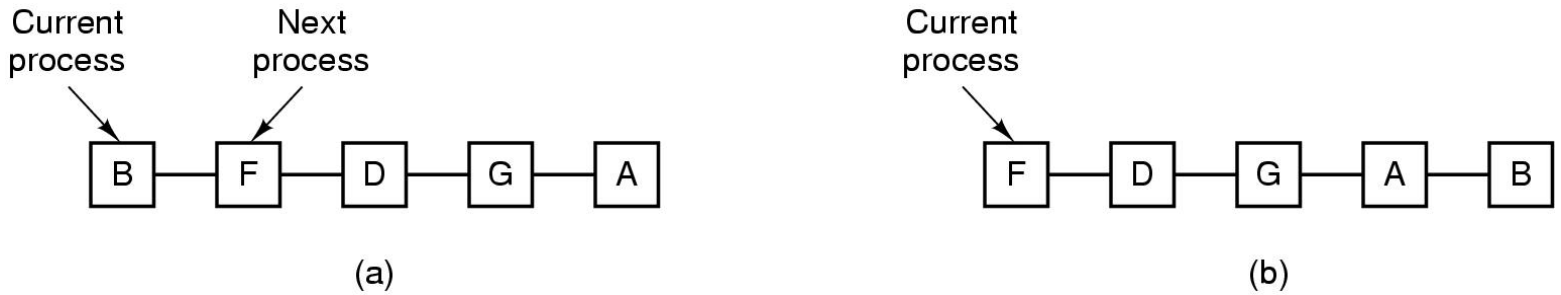
- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度

交互式系统的调度算法

- 时间片轮转(RR：Round Robin)
- 多级队列（MQ：Multi-level Queue）
- 多级反馈队列（MFQ： Multi-level Feedback Queue ）

时间片轮转(Round Robin)算法

- 本算法主要用于微观调度，设计目标是提高资源利用率。其基本思路是通过时间片轮转，提高进程并发性和响应时间特性，从而提高资源利用率

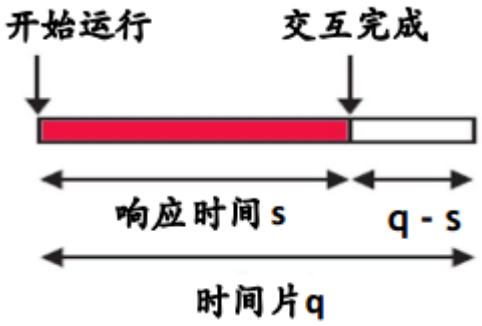


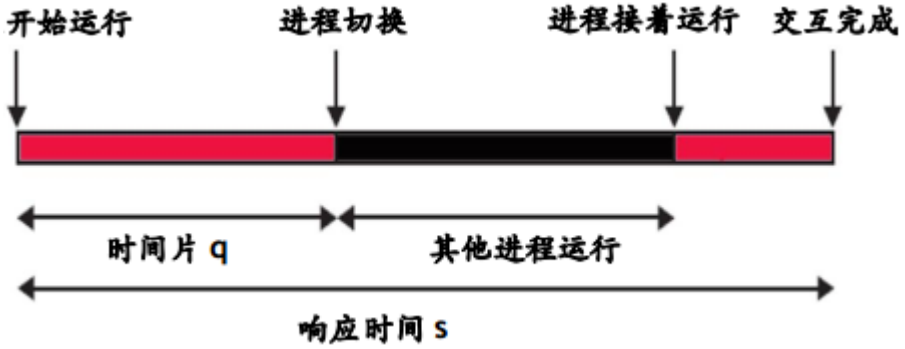
时间片轮转算法

- 将系统中所有的就绪进程按照FCFS原则，排成一个队列
- 每次调度时将CPU分派给队首进程，让其执行一个时间片。时间片的长度从几个ms到几百ms
- 在一个时间片结束时，发生时钟中断
- 调度程序据此暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程
- 进程可以未使用完一个时间片，就出让CPU（如阻塞）

时间片长度的确定

- 时间片长度变化的影响
  - 过长—>退化为FCFS算法，进程在一个时间片内都执行完，响应时间长
  - 过短—>用户的一次请求需要多个时间片才能处理完，上下文切换次数增加，响应时间长





确定时间片长度要考虑的因素

- 对响应时间的要求：

  T(响应时间) = N(进程数目) * q(时间片)

- 就绪进程的数目：
  - 数目越多，时间片越小
- 系统的处理能力：应当使用户输入通常在一个时间片内能处理完，否则使响应时间，平均周转时间和平均带权周转时间延长

CPU执行速度

10ms能够执行多少条指令？It depends …

- 流水线的工作频率
- 普通流水线、超流水、超标量、乱序执行
- Hyper-threading
- 多核、多CPU、SMP、NUMA

- Linux根据用户优先级设置时间片，5-800ms，优先级动态调整
- 现在x86 CPU上Linux 2.6内核上下文切换通常不超过10us。回顾：线程切换和进程切换的开销？

作业调度举例

作业	到达时间	作业时长
A	0	4
B	1	3
C	2	5
D	3	2
E	4	4

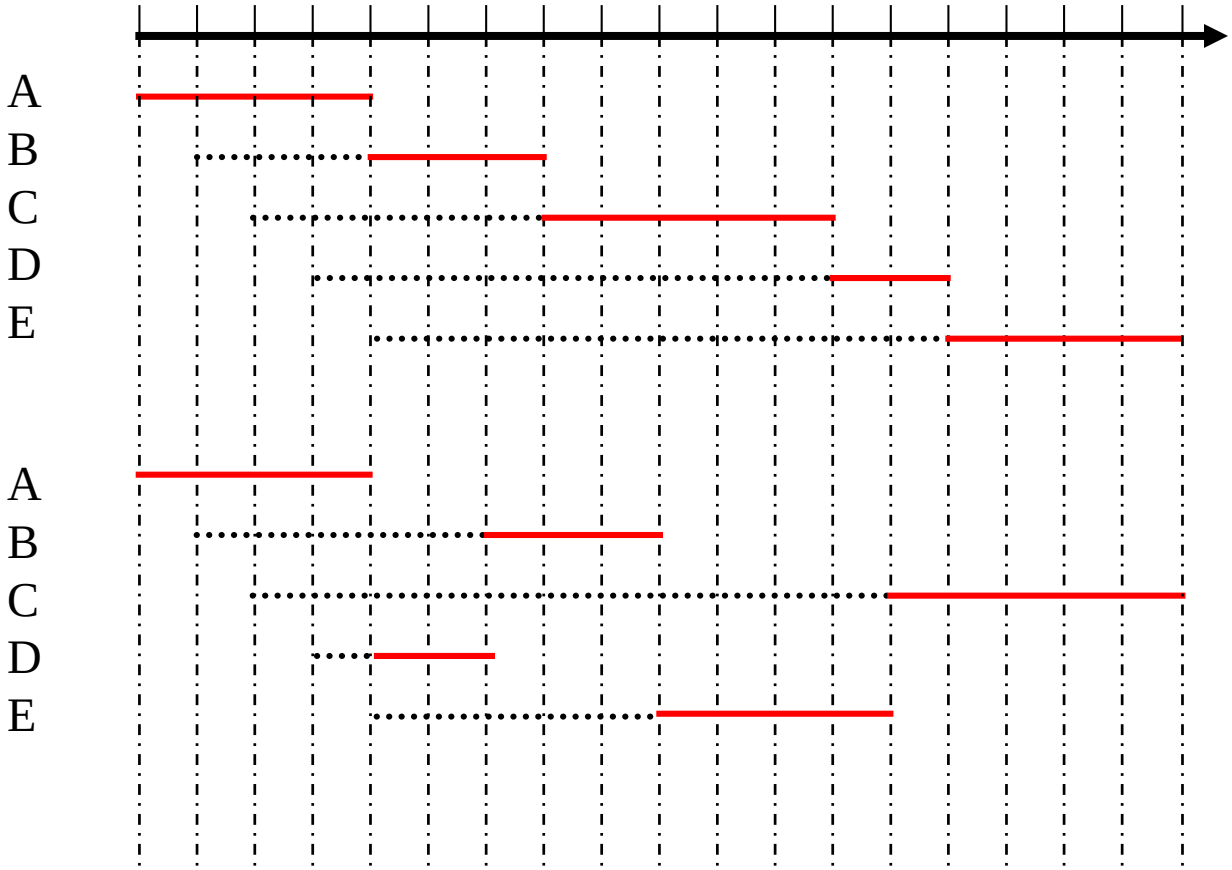
A B C D E

0 1 2 3 4

4 3 5 2 4

作业到达时间

作业执行长度



FCFS

平均周转时间 ＝ 9

带权平均周转时间 ＝2.8

SJF

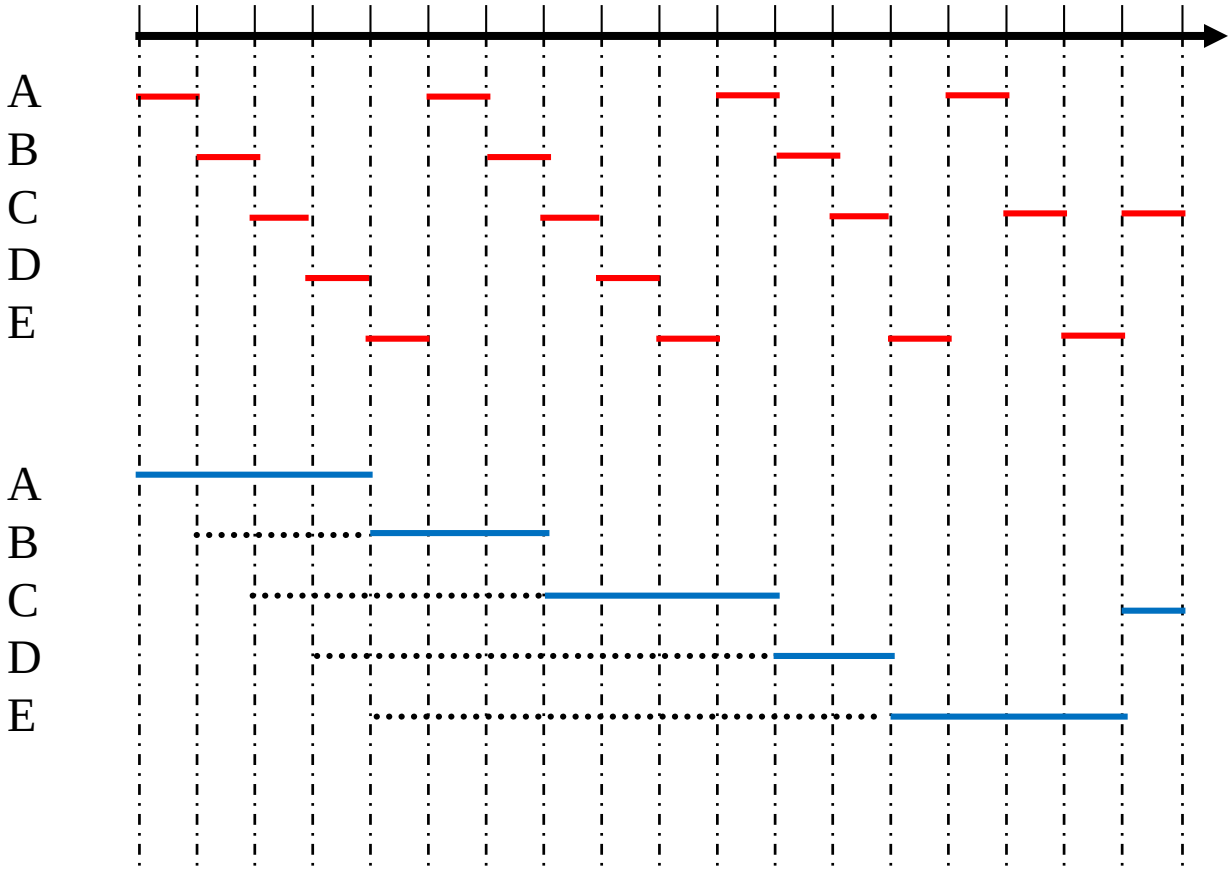
平均周转时间 ＝ 8

带权平均周转时间 ＝2.1

A B C D E
0 1 2 3 4
4 3 5 2 4

作业到达时间

作业执行长度



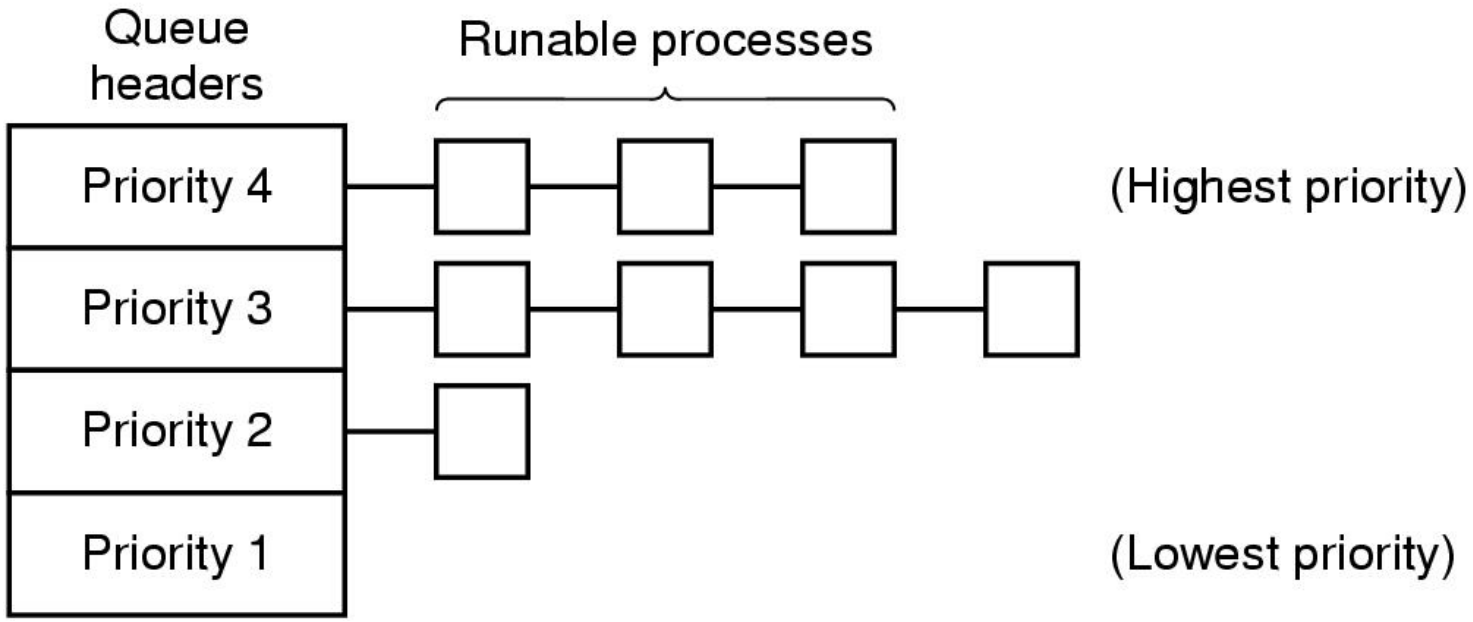
RR

1单位时间片
15次切换

4单位时间片
5次切换

优先级算法(Priority Scheduling)

- 本算法是平衡各进程对响应时间的要求
  - 适用于作业调度和进程调度
  - 可分成抢先式和非抢先式



静态优先级

- 创建进程时就确定，直到进程终止前都不改变。通常是一个整数。依据：
  - 进程类型（系统进程优先级较高）
  - 对资源的需求（对CPU和内存需求较少的进程，优先级较高）
  - 用户要求（紧迫程度和付费多少）

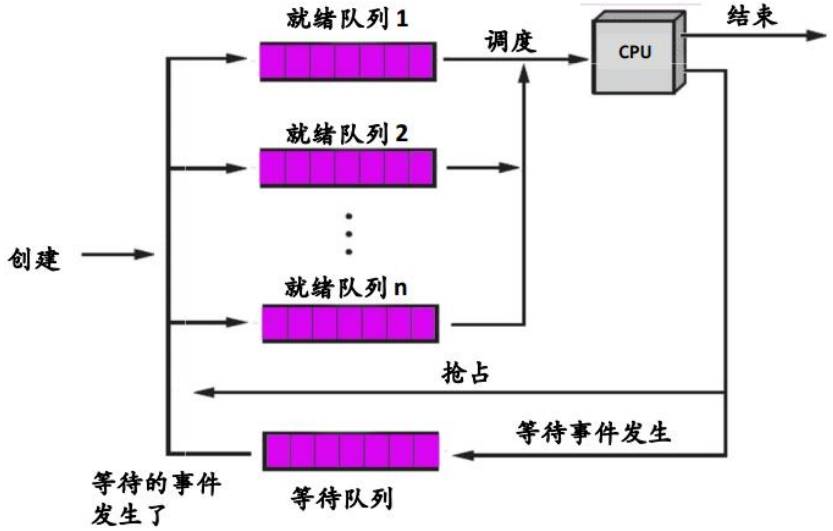
动态优先级

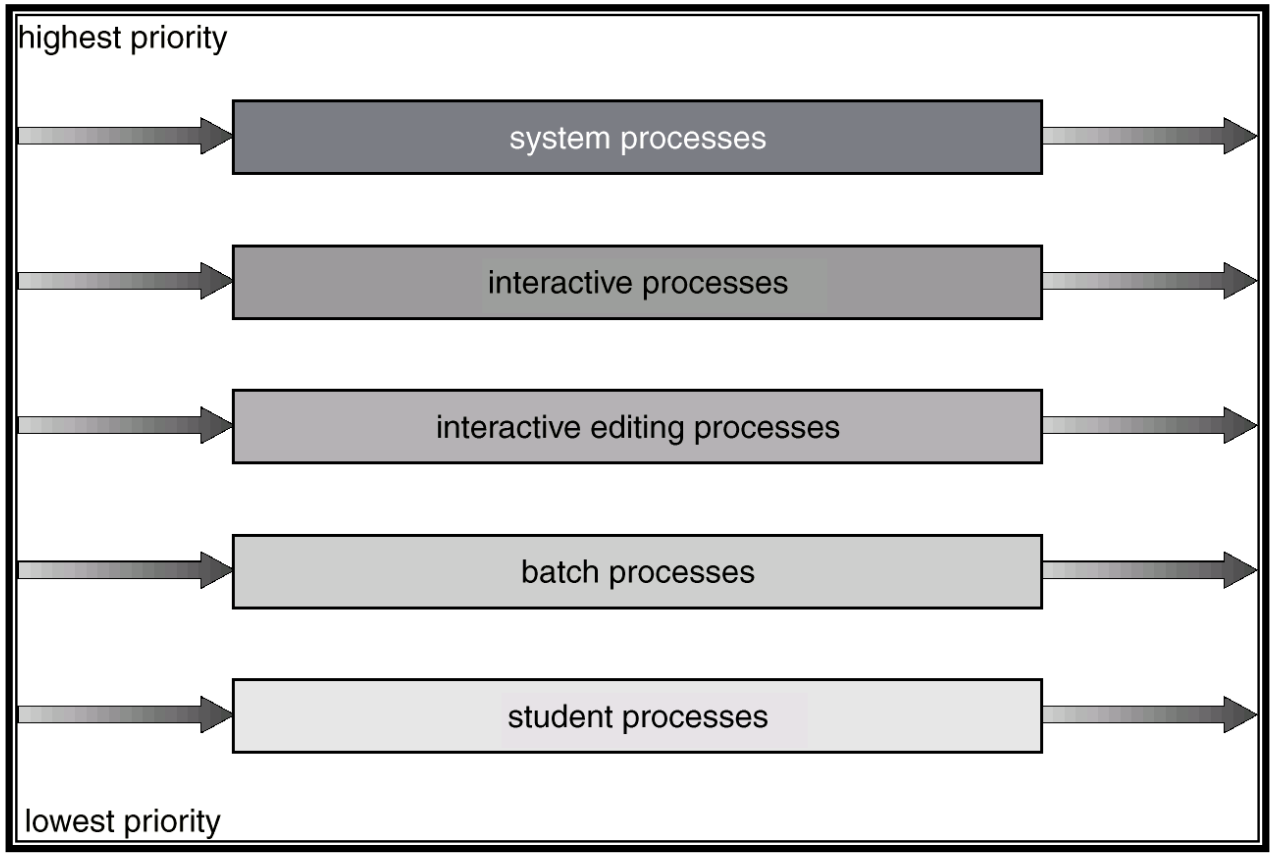
- 在创建进程时赋予的优先级，在进程运行过程中可以自动改变，以便获得更好的调度性能。如：
  - 在就绪队列中，等待时间延长则优先级提高，从而使优先级较低的进程在等待足够的时间后，其优先级提高到可被调度执行；
  - 进程每执行一个时间片，就降低其优先级，从而一个进程持续执行时，其优先级降低到出让CPU

多级队列算法MQ

- 本算法引入多个就绪队列，通过各队列的区别对待，达到一个综合的调度目标；
  - 根据作业或进程的性质或类型的不同，将就绪队列再分为若干个子队列
  - 每个作业固定归入一个队列
- 不同队列可有不同的优先级、时间片长度、调度策略等；在运行过程中还可改变进程所在队列。
  - 如：系统进程、用户交互进程、批处理进程等

多级队列算法MQ





OS教学组

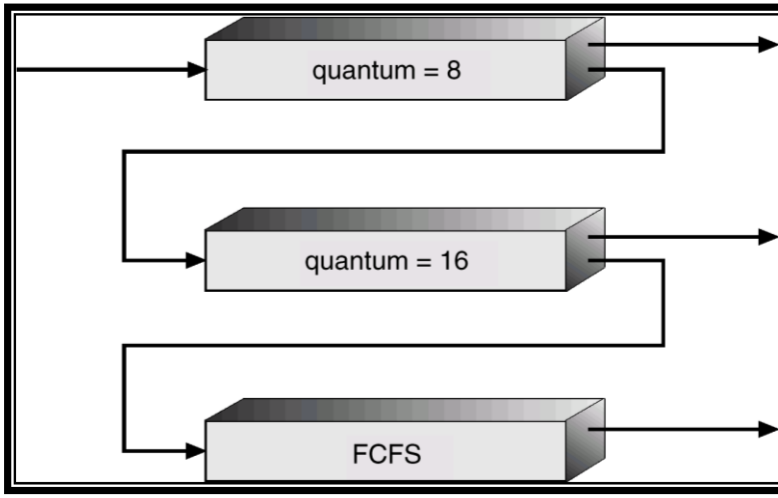
多级反馈队列算法MFQ

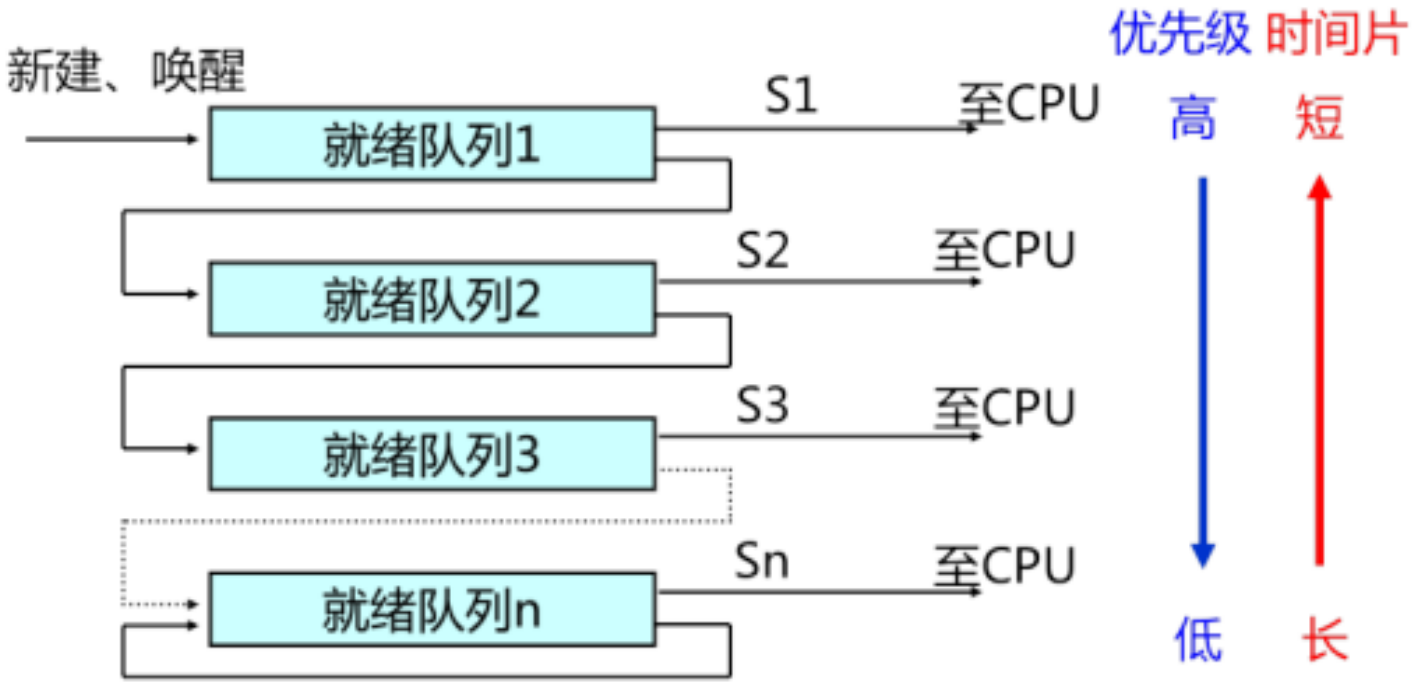
- 多级反馈队列算法时间片轮转算法和优先级算法的综合和发展。优点：
  - 为提高系统吞吐量和缩短平均周转时间而照顾短进程
  - 为获得较好的I/O设备利用率和缩短响应时间而照顾I/O型进程
  - 不必估计进程的执行时间，动态调节

多级反馈队列算法

- 设置多个就绪队列，分别赋予不同的优先级（如逐级降低），队列1的优先级最高。每个队列执行时间片的长度也不同，规定优先级越低则时间片越长（如逐级加倍）
- 新进程进入内存后，先投入队列1的末尾，按FCFS算法调度；若按队列1一个时间片未能执行完，则降低投入到队列2的末尾，同样按FCFS算法调度；如此下去，降低到最后的队列，则按“时间片轮转”算法调度直到完成
- 仅当较高优先级的队列为空，才调度较低优先级的队列中的进程执行。如果进程执行时有新进程进入较高优先级的队列，则抢先执行新进程，并把被抢先的进程投入原队列的末尾

多级反馈队列算法





时间片：S1<S2<S3

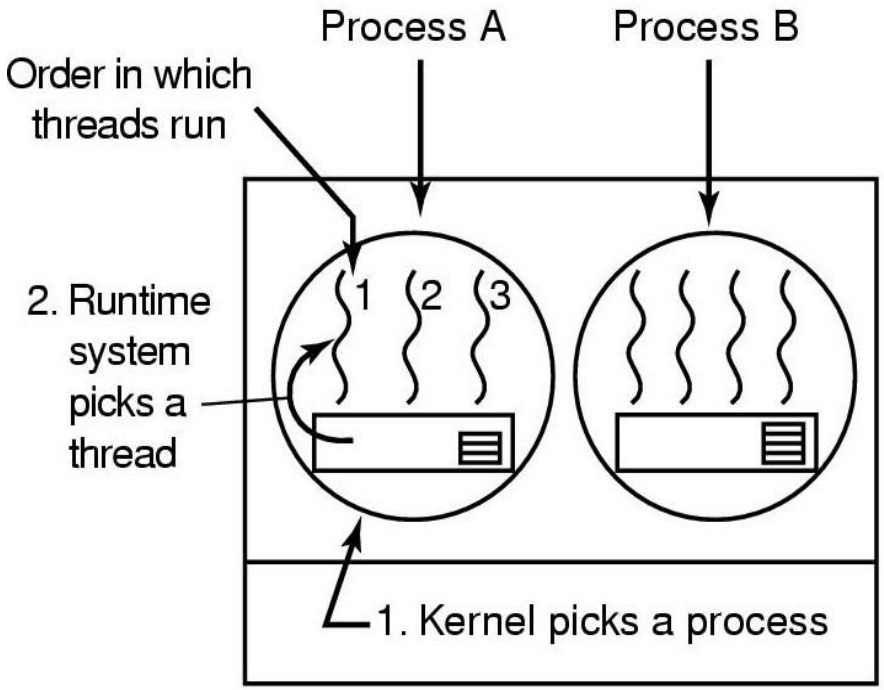
教学组

几点说明

- I/O型进程：让其进入最高优先级队列，以及时响应I/O交互。通常执行一个小时间片，要求可处理完一次I/O请求的数据，然后转入到阻塞队列
- 计算型进程：每次都执行完时间片，进入更低级队列。最终采用最大时间片来执行，减少调度次数
- I/O次数不多，而主要是CPU处理的进程：在I/O完成后，放回I/O请求时离开的队列，以免每次都回到最高优先级队列后再逐次下降
- 为适应一个进程在不同时间段的运行特点
  - I/O完成时，提高优先级
  - 时间片用完时，降低优先级

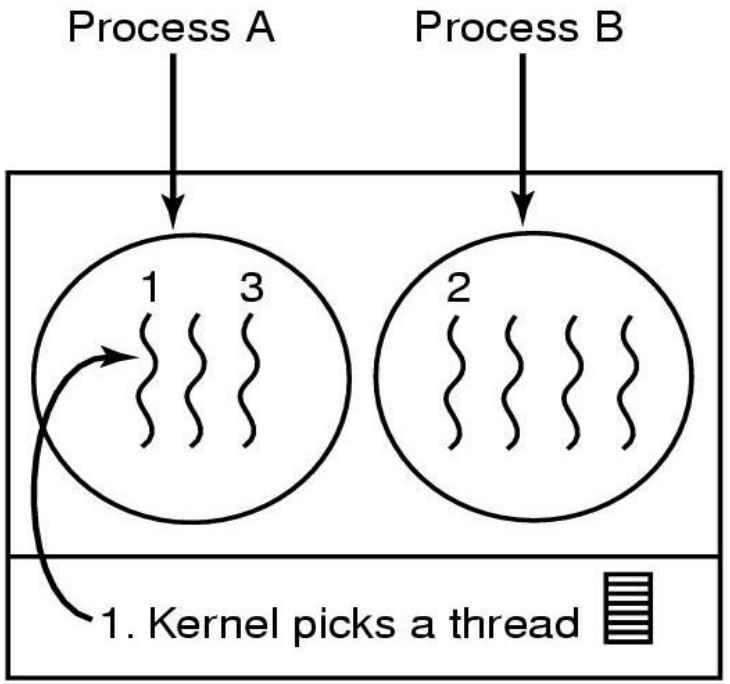
线程调度

用户级线程级



Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3

内核级线程级

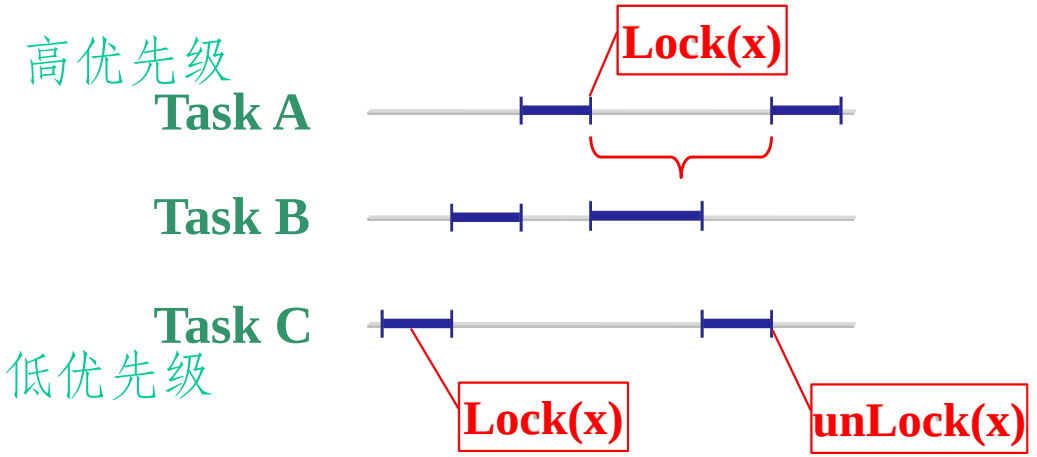


Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

优先级倒置

优先级倒置现象

- 高优先级进程（或线程）被低优先级进程（或线程）延迟或阻塞
  - 例如：有三个完全独立的进程 Task A、Task B 和 TaskC，Task A 的优先级最高，Task B 次之，Task C 最低。Task A 和 Task C 共享同一个临界资源 X

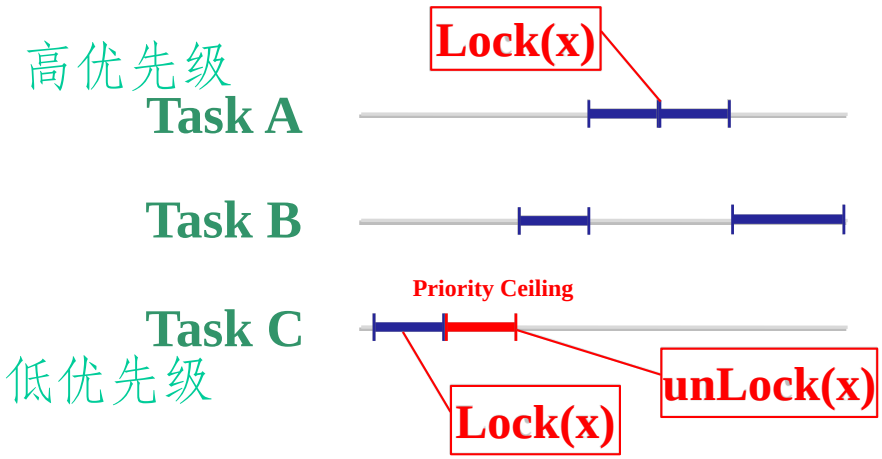


Task A 和 Task C 共享同一个临界资源，高优先级进程 Task A 因低优先进程 Task C 被阻塞，又因为低优先进程 Task B 的存在延长了被阻塞的时间

解决方法——优先级置顶

优先级置顶（Priority Ceiling）

- 进程 Task C 在进入临界区后，Task C 所占用的处理机就不允许被抢占。这种情况下，Task C 具有最高优先级（Priority Ceiling）

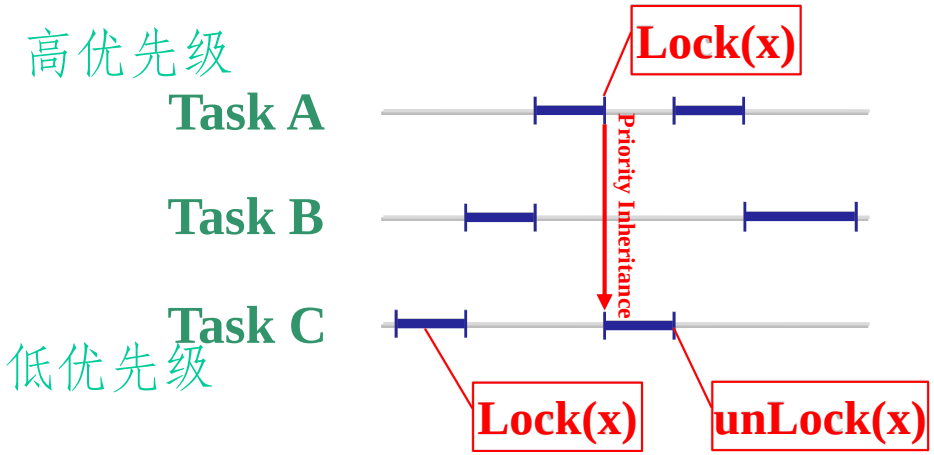


如果系统中的临界区都较短且不多，该方法是可行的。反之，如果 Task C 临界区非常长，则高优先级进程Task A 仍会等待很长的时间，其效果无法令人满意

解决方法——优先级继承

优先级继承（Priority Inheritance）

- 当高优先级进程 Task A 要进入临界区使用临界资源 X时，如果已经有一个低优先级进程 Task C 正在使用该资源，可以采用优先级继（Priority Inheritance）的方法



此时一方面 Task A 被阻塞，另一方面由 Task C 继承Task A 的优先级，并一直保持到 Task C 退出临界区

内容提要

- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度

实时系统

- 实时系统是一种时间起着主导作用的系统
  - 当外部的物理设备给了计算机一个刺激，计算机必须在一个确定的时间范围内恰当地做出反应
  - 对于这种系统来说，正确的但是迟到的应答往往比没有还要糟糕
- 实时系统被分为硬实时系统和软实时系统
  - 硬实时要求绝对满足截止时间要求，如：汽车和飞机的控制系统
  - 而软实时倒是可以偶尔不满足，如视频/音频程序
- 实时系统通常将对不同刺激的响应指派给不同的进程（任务），并且每个进程的行为是可提前预测的

实时调度

问题描述：

- 假设一任务集S={t_1,t_2,t_3,..., t_n}，周期分别是T_1,T_2,...,T_n，执行时间为c_1,c_2,..., c_n，截至周期(deadline)为D_1,D_2,..., D_n，通常$D_i = T_i$。CPU利用率：用$U = \sum_1^n (ci/Ti)$ 表示
- 前提条件
  - 任务集（S）是已知的
  - 所有任务都是周期性（T）的，必须在限定的时限（D）内完成
  - 任务之间都是独立的，每个任务不依赖于其他任务
  - 每个任务的运行时间（c）是不变的
  - 调度,任务切换的时间忽略不计

实时调度算法

- 静态表调度**Static table-driven scheduling**
- 单调速率调度**RMS：Rate Monotonic Scheduling**
  - 任务集可调度，**if**，$\sum_{i=1}^{n}\frac{C_i}{T_i} \le \mathbf{n}(\sqrt[n]{2}-1)$

$$\lim_{n\to\infty} n(\sqrt[n]{2}-1) = \ln 2 \approx 0.693147\ldots$$

- 最早截止时间优先算法**EDF：Earliest Deadline First**
  - 任务集可调度，**iff**，$\sum_{i=1}^{n}\frac{C_i}{T_i} \le 1$

静态表调度算法

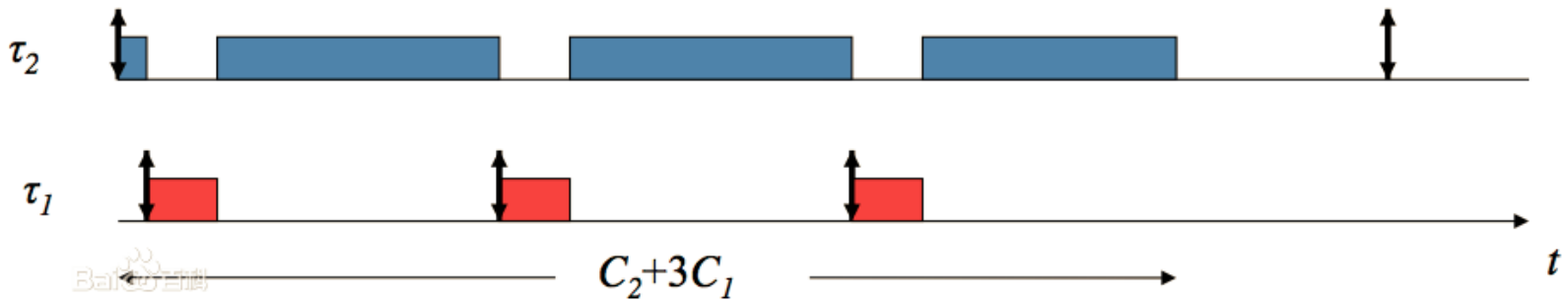
- 通过对所有周期性任务的分析预测（到达时间、运行时间、结束时间、任务间的优先关系），事先确定一个固定的调度方案
- 特点：
- 无任何计算，按固定方案进行，开销最小；
- 无灵活性，只适用于完全固定的任务场景

单调速率调度RMS

- RMS是单处理器下的最优静态调度算法
  - 1973年Liu和Layland发表的这篇文章的前半部分首次提出了RM调度算法在静态调度中的最优性
  - 它的一个特点是可通过对系统资源利用率的计算来进行任务可调度性分析，算法简单、有效，便于实现
- 不仅如此，他们还把系统的利用系数(utilization factor)和系统可调度性联系起来，推导出用RM调度所能达到的最小系统利用率公式

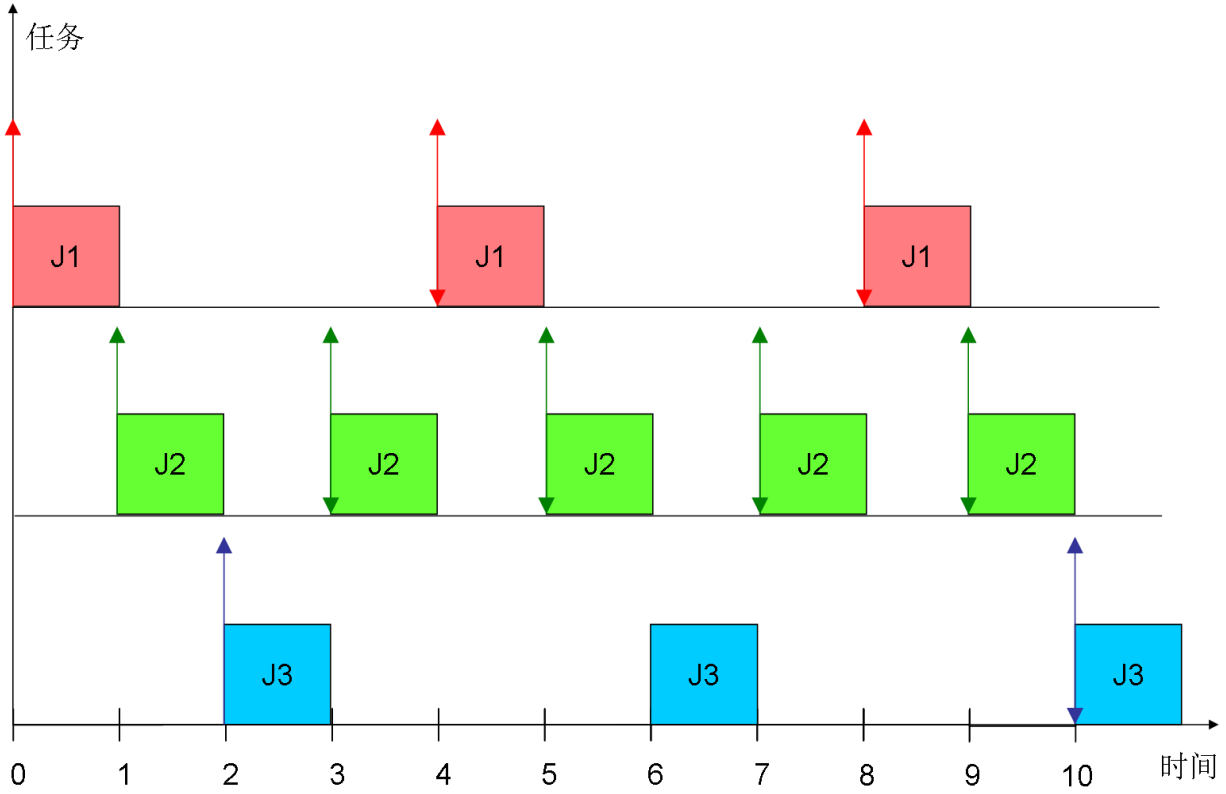
单调速率调度RMS

- RMS已被证明是静态最优调度算法，开销小，灵活性好，是实时调度的基础性理论
- 特点
  - 任务的周期越小，其优先级越高，优先级最高的任务最先被调度
  - 如果两个任务的优先级一样，当调度它们时，RM算法将随机选择一个调度
- 静态、抢先式调度



http://dl.acm.org/citation.cfm?doid=321738.321743

任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	1	4	4
J2	1	1	2	2
J3	2	2	8	8

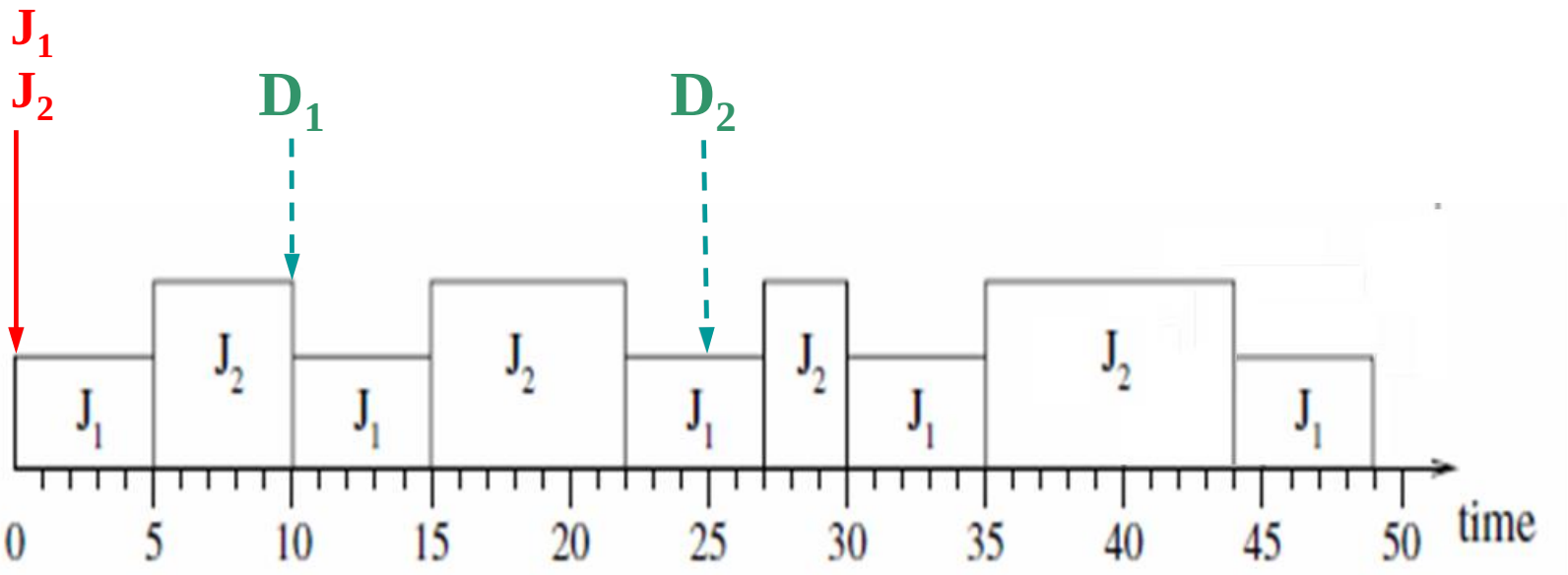


最早截止期优先EDF

- 任务的绝对截止时间越早，其优先级越高，优先级最高的任务最先被调度
- 如果两个任务的优先级一样，当调度它们时，EDF算法将随机选择一个调度

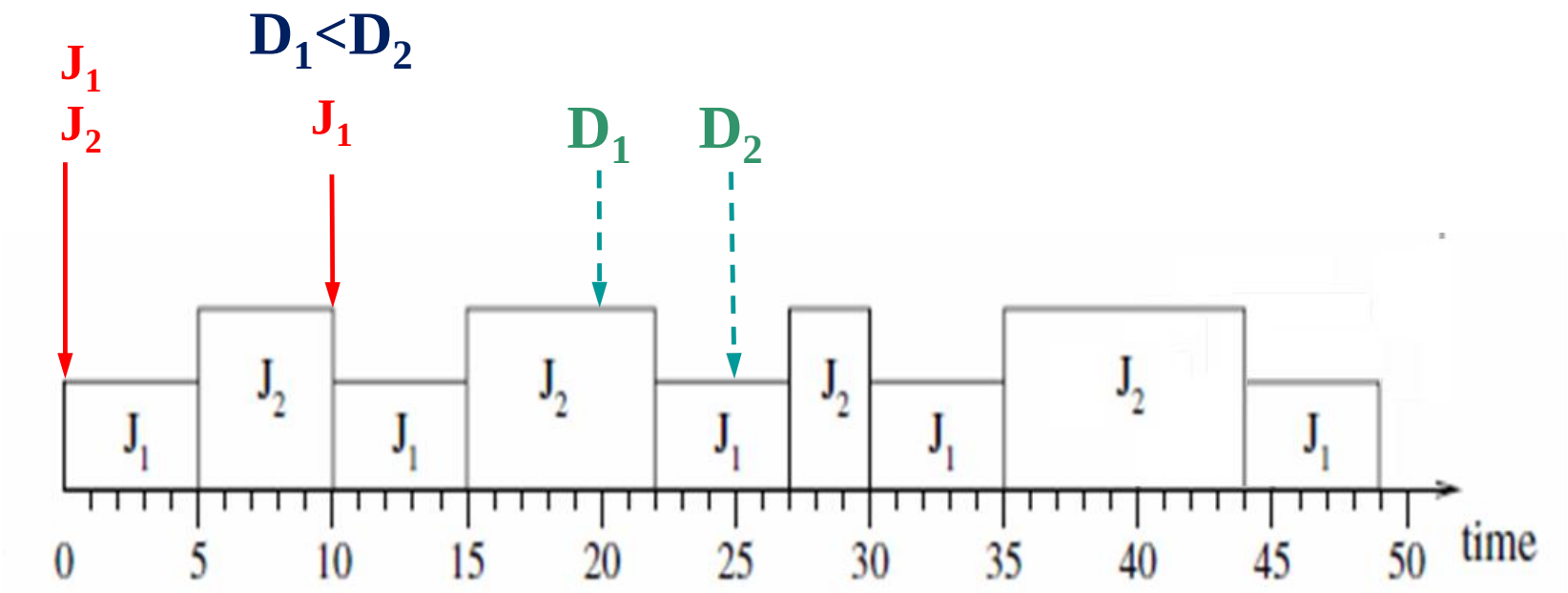
任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25

$D_1<D_2$



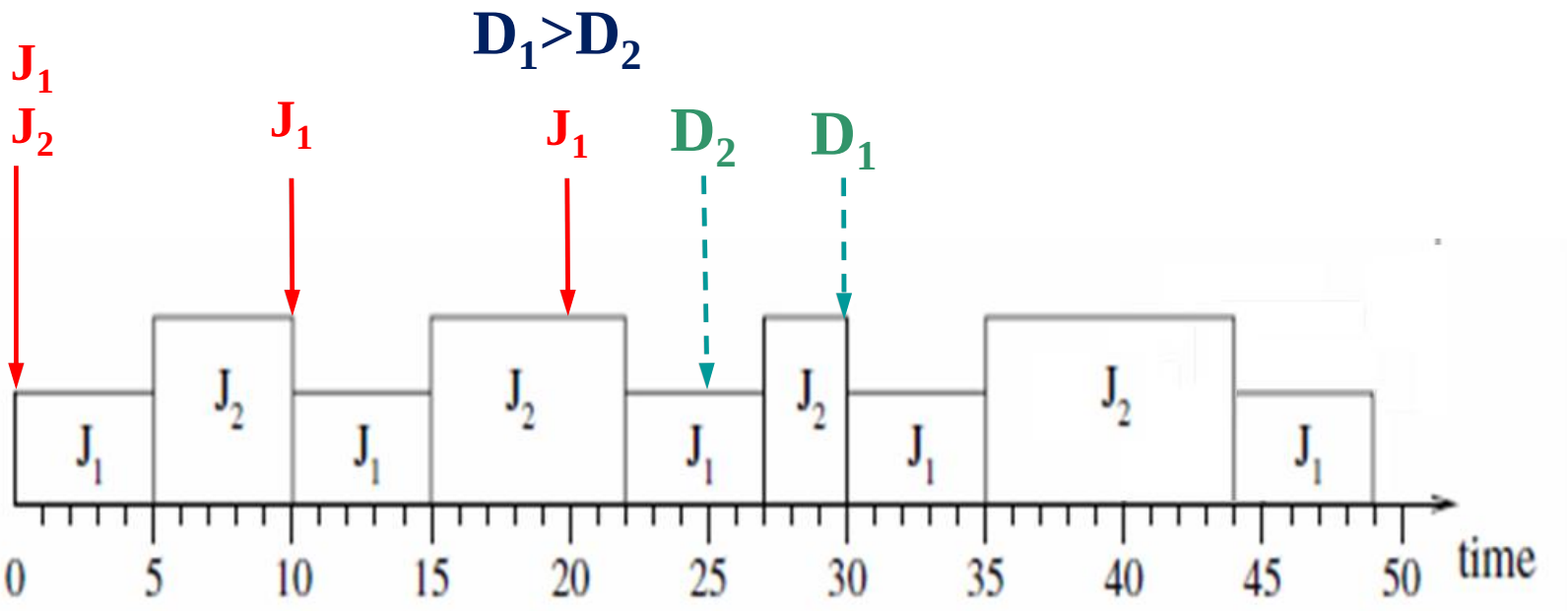
在0时刻，由于$D_1<D_2$选择J_1运行，之后J_2运行

任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25



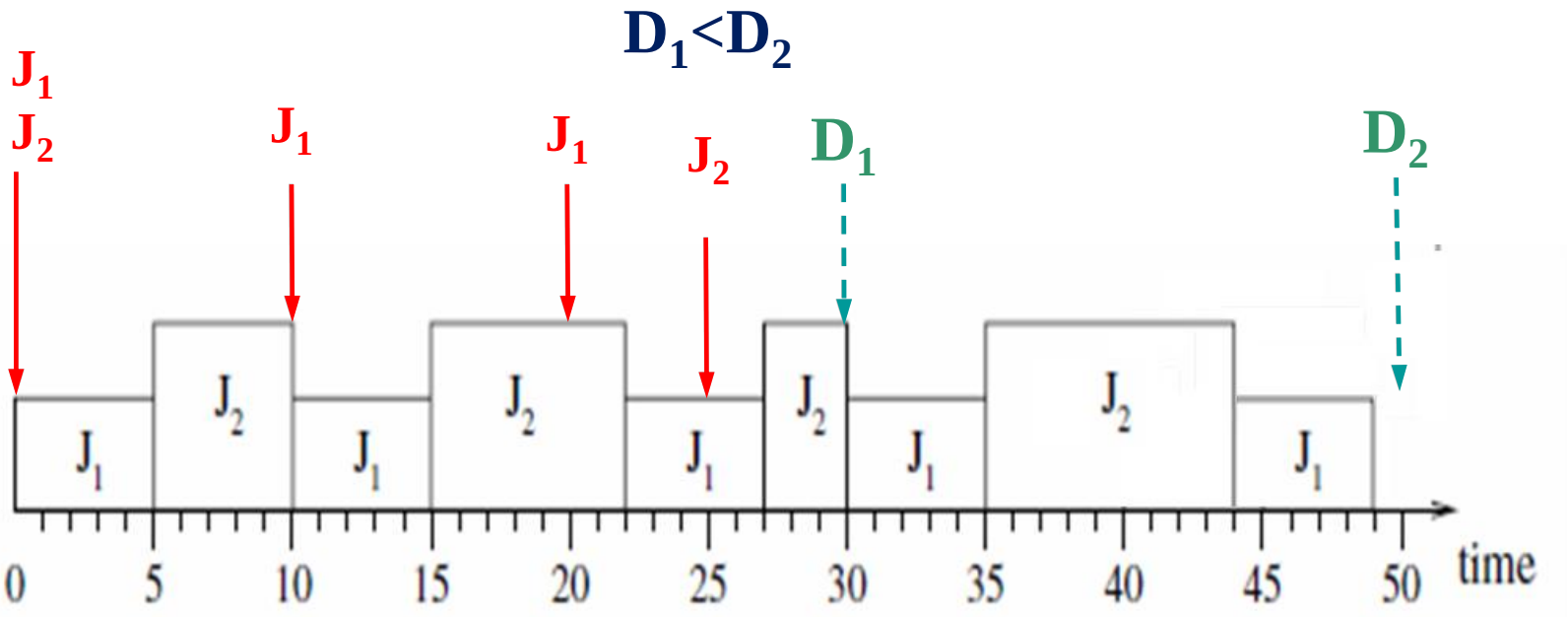
在时刻10，J_1的新周期开始，切换到J_1运行，之后J_2运行

任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25



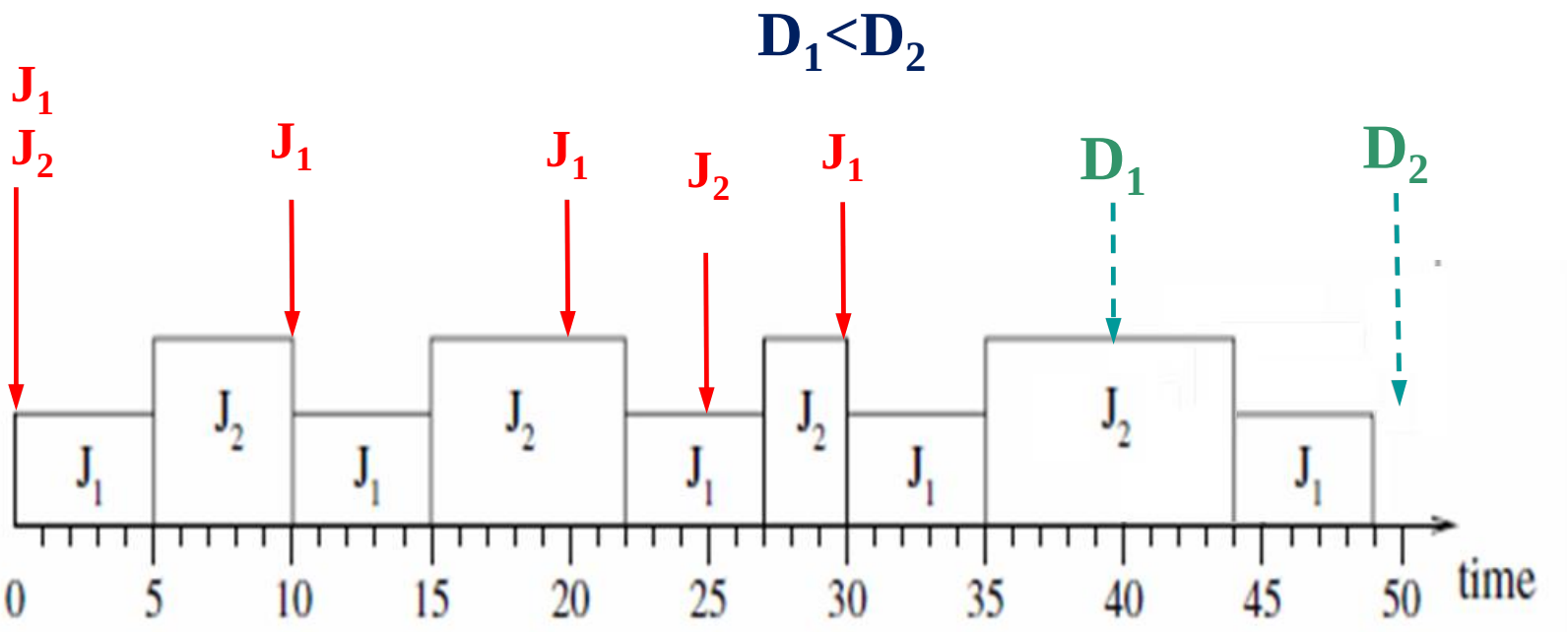
在时刻20，J_1新周期开始，但由于$D_1>D_2$，保持J_2运行，之后J_1运行

任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25



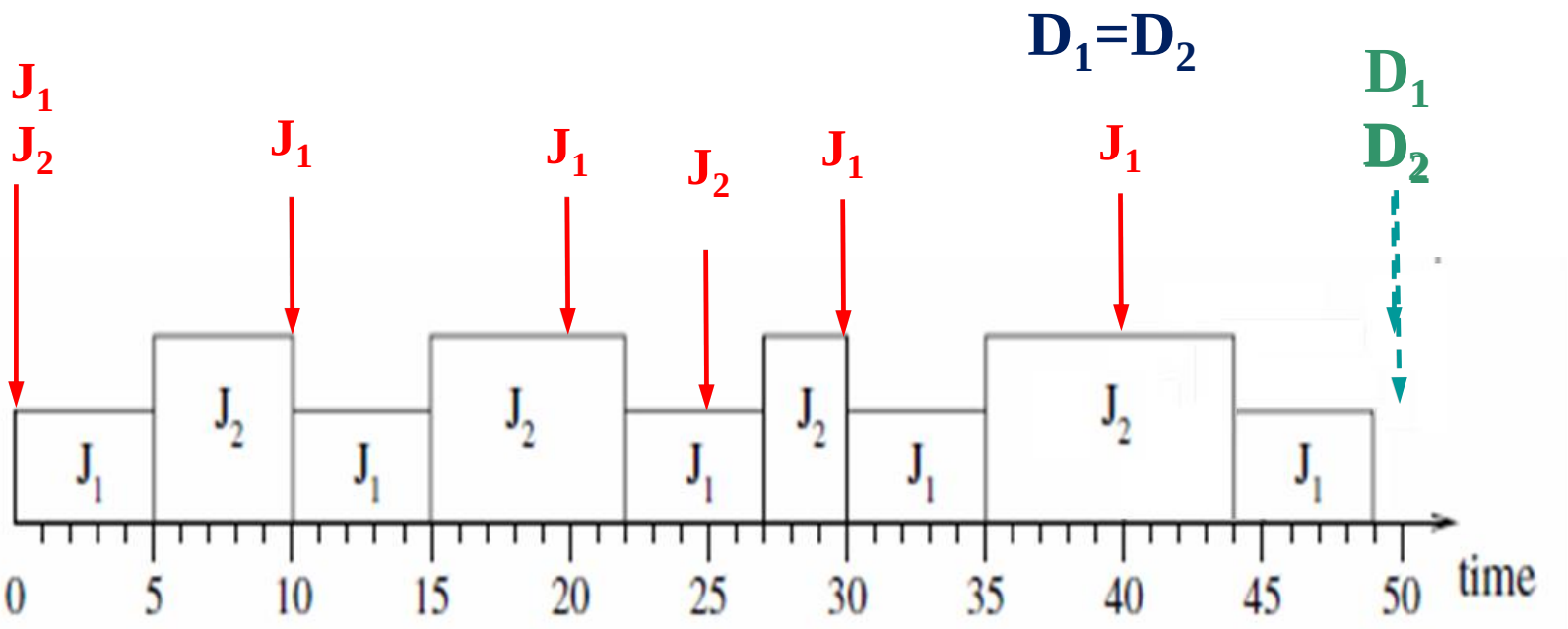
在时刻25，J_2新周期开始，但由于$D_1<D_2$，保持J_1运行，之后J_2运行

任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25



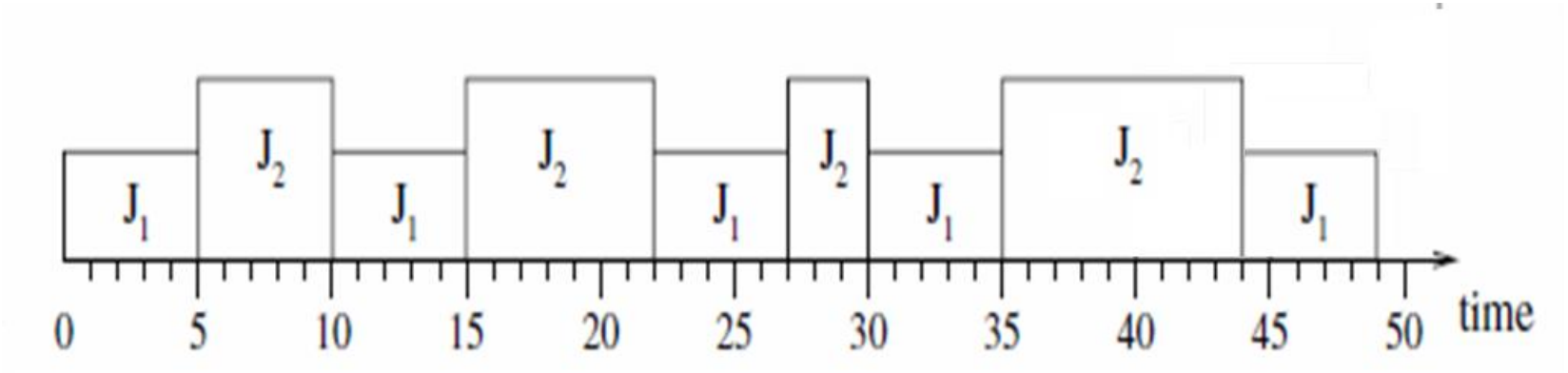
在时刻30，J_1的新周期开始，切换到J_1运行，之后J_2运行

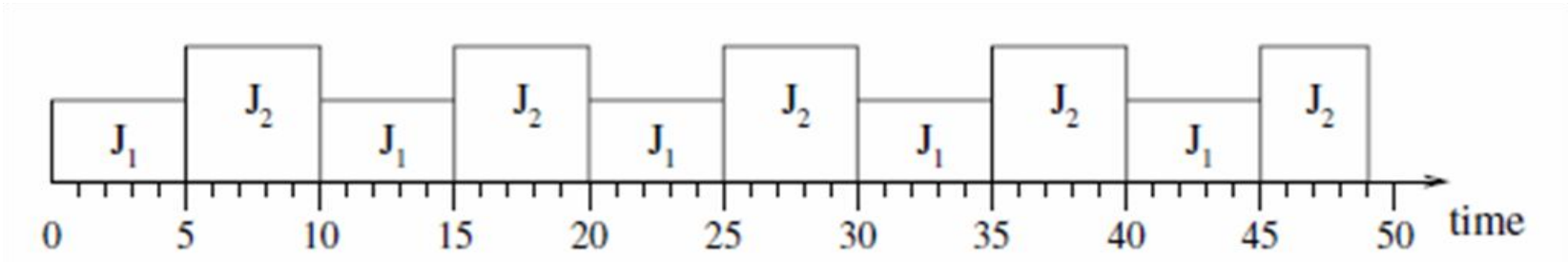
任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25



在时刻40，J_1新周期开始，$D_1=D_2$，为减少一次切换保持J_2运行，之后J_1运行

任务Ji	起始时间Si	执行时间Ci	周期Pi	截止时间Di
J1	0	5	10	10
J2	0	12	25	25





内容提要

- 基本概念
- 设计调度算法要考虑的问题
- 批处理系统的调度算法
- 交互式系统的调度算法
- 实时系统的调度算法
- 多处理机调度

多处理机调度

- 与单处理机调度的区别：
  - 注重整体运行效率
    - 而不是个别处理机的利用率
  - 更多样的调度算法
  - 多处理机访问OS数据结构时的互斥
    - 对于享内存系统
- 调度单位广泛采用线程

非对称式多处理系统(AMP)

- AMP：Asymmetric Multi-Processor，指多处理器系统中，各个处理器的地位不同
- 主－从处理机系统
  - 由主处理机管理一个公共就绪队列，并分派进程给从处理机执行
- 各个处理机有固定分工
  - 如执行OS的系统功能，I/O处理，应用程序
- 有潜在的不可靠性
  - 主机故障造成系统崩溃

对称式多处理系统(SMP)

- SMP：Symmetric Multi-Processor，指多处理器系统中，各个处理器的地位相同
- 按控制方式，SMP调度算法可分为集中控制和分散控制
- 下面所述
  - 静态和动态调度都是集中控制
  - 而自调度是分散控制

对称式多处理系统(SMP)

- 静态分配(static assignment)：每个CPU设立一个就绪队列，进程从开始执行到完成，都在同一个CPU上
  - 优点：调度算法开销小
  - 缺点：容易出现忙闲不均
- 动态分配(dynamic assignment)：所有CPU采用一个公共就绪队列，队首进程每次分派到当前空闲的CPU上执行
  - 可防止系统中多个处理器 忙闲不均

对称式多处理系统(SMP)

- 自调度(self-scheduling)：各个CPU采用一个公共就绪队列，每个处理机都可以从队列中选择适当进程来执行
- 需要对就绪队列的数据结构进行互斥访问控制。是最常用的算法，实现时易于移植，采用单处理机的调度技术
  - 变型：Mach OS中局部和全局就绪队列相结合，其中局部就绪队列中的线程优先调度

自调度（Self Scheduling）

自调度：各个处理机自行在就绪队列中取任务

- 优点：不需要专门的处理机从事任务分派工作
- 缺点: 当处理机个数较多（如十几个或上百个）时，对就绪队列的访问可能成为系统的瓶颈
- 低效问题?
- 线程切换问题?

成组调度(gang scheduling)

- 将一个进程中的一组线程，每次分派时同时到一组处理机上执行，在剥夺处理机时也同时对这一组线程进行
- 优点
  - 通常这样的一组线程在应用逻辑上相互合作，成组调度提高了这些线程的执行并行度，有利于减少阻塞和加快推进速度，最终提高系统吞吐量
  - 每次调度可以完成多个线程的分派，在系统内线程总数相同时能够减少调度次数，从而减少调度算法的开销

两种成组调度

	应用程序A	应用程序B
Cpu1	线程1	线程1
Cpu2	线程2	空闲
Cpu3	线程3	空闲
Cpu4	线程4	空闲
时间	1/2	1/2

面向程序：浪费3/8的处理机时间

	应用程序A	应用程序B
Cpu1	线程1	线程1
Cpu2	线程2	空闲
Cpu3	线程3	空闲
Cpu4	线程4	空闲
时间	4/5	1/5

面向线程：浪费3/20的处理机时间

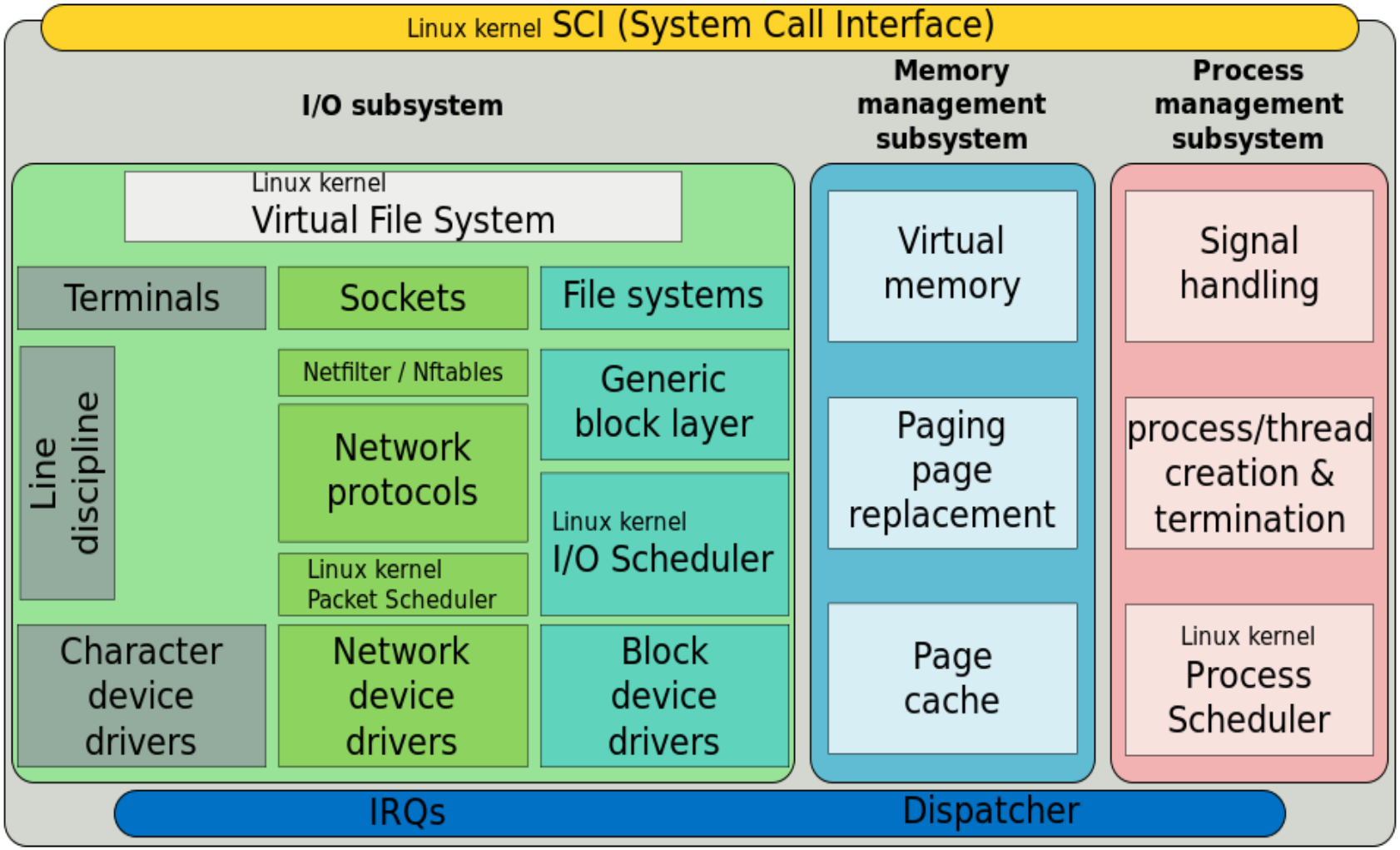
面向所有程序（进程）平分处理机时间

面向所有线程平分处理机时间

专用处理机调度(dedicated processor assignment)

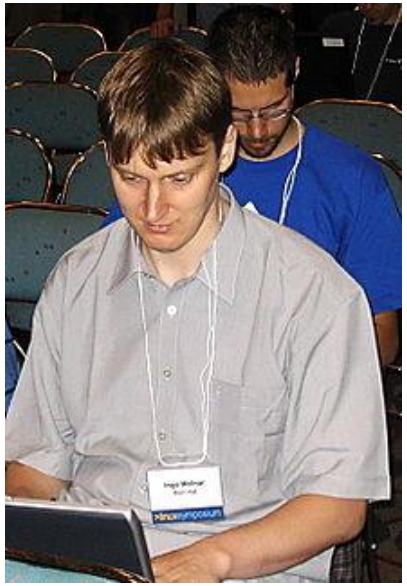
- 为进程中的每个线程都固定分配一个CPU，直到该线程执行完成
- 缺点：线程阻塞时，造成CPU的闲置。优点：线程执行时不需切换，相应的开销可以大大减小，推进速度更快
- 适用场合：CPU数量众多的高度并行系统，单个CPU利用率已不太重要

Linux的调度



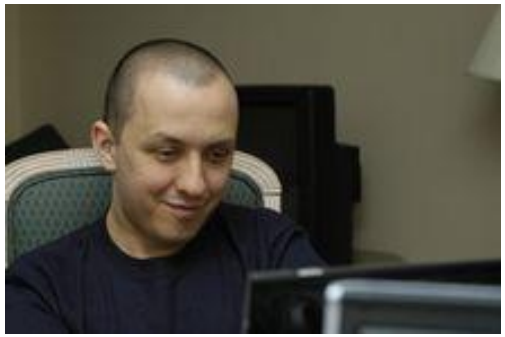
Linux调度的历史变迁

- Linux 2.4：O(n)调度器 (Linus)
- Linux 2.6.0：O(1)调度器
- Linux 2.6.23：CFS（Completely Fair Scheduler，完全公平调度器）(Ingo Molnár)



- 其他：(Con Kolivas)
  - SD(staircase scheduler)
  - RSDL(The Rotating Staircase Deadline Schedule)
  - BFS

https://www.eet-china.com/mp/a111242.html
万字长文，锤它！揭秘Linux进程调度器



Linux的调度策略

- Linux中实现了三种进程调度策略：
  - SCHED_OTHER。一般进程
  - SCHED_FIFO。先进先出（First In First Out）的实时进程
  - SCHED_RR。轮转（Round Robin）方式执行的实时进程

2.4以前内核的单队列

- Linux并不为这三种调度策略的进程分别设置一个运行队列，而是通过权重的不同计算以及其他的一些队列操作，在一个运行队列中实现这三种不同的调度
- 发生进程调度时，调度程序要在运行队列中选择一个最值得运行的进程来执行，这个进程便是通过在运行队列中一一比较各个可运行进程的权重来选择的
- 权重越大的进程越优先，而对于相同权重的进程，在运行队列中的位置越靠前越优

调度策略的排队方式

- 调度策略为SCHED_RR的实时进程，在分配的时间片到期后，插入到运行队列的队尾
- 调度策略为SCHED_FIFO的进程，在时间片到期后，调度程序并不改变该进程在运行队列中的位置

具体的实现函数

- sys_sched_setschedule，sys_sched_setparam，sys_sched_getscheduler，sys_sched_getparam，sys_sched_get_priority_max，sys_sched_get_priority_min

优先级priority

- 进程的优先级反映了进程相对于其他进程的可选择度，其实就是系统每次允许进程运行的时间。子进程继承了父进程的优先级
- priority也可以通过系统调用sys_setpriority（sys_nice已被sys_setpriority取代）设置
- 系统为每个进程预定的priority为DEF_PRIORITY（include/linux/sched.h），200ms

相对优先级rt_priority

- 对于实时进程，除了用priority来反映其优先级（可执行时间）外，还有相对优先级用于同类进程之间的比较选择。
- 实时进程的rt_priority取值1 ~ 99，一般进程的rt_priority值只能取0。
- 进程的rt_priority 可通过setscheduler函数而改变。

计数器counter

- counter用以反映进程所剩余的可运行时间，在进程运行期间，每次发生时钟中断时，其值减1，直至0。由于时钟中断为快中断，在其底半处理过程中，才刷新当前进程的counter值。因此，也可能在发生了好几次时钟中断后才集中进行处理。

counter的修改

- 计数器counter是衡量一般进程权重的重要指标，主要因为如下几种事件而改变：
  - task 0的counter初值为DEF_PRIORITY，在执行sys_idle()时，将counter值置为 -100。
  - 在创建子进程时，父进程的counter变为原值的一半，并将该值赋予子进程。
  - 在进程运行期间，每次发生时钟中断时，counter值减1，直至为0。
  - 若所有的可运行进程的counter值都为0，则需要为所有的进程都重新赋counter值。

权重

- 权重通过调用函数goodness来计算，对于实时进程，其权重为1000+rt_priority；否则，权重为counter。
- 对于当前进程，可以得到比其他进程稍高的权重，为counter+1。这样处理是为了在某个进程与当前进程权重相同时可选择当前进程继续执行，以减少进程切换的开销。

传统Linux调度器的问题

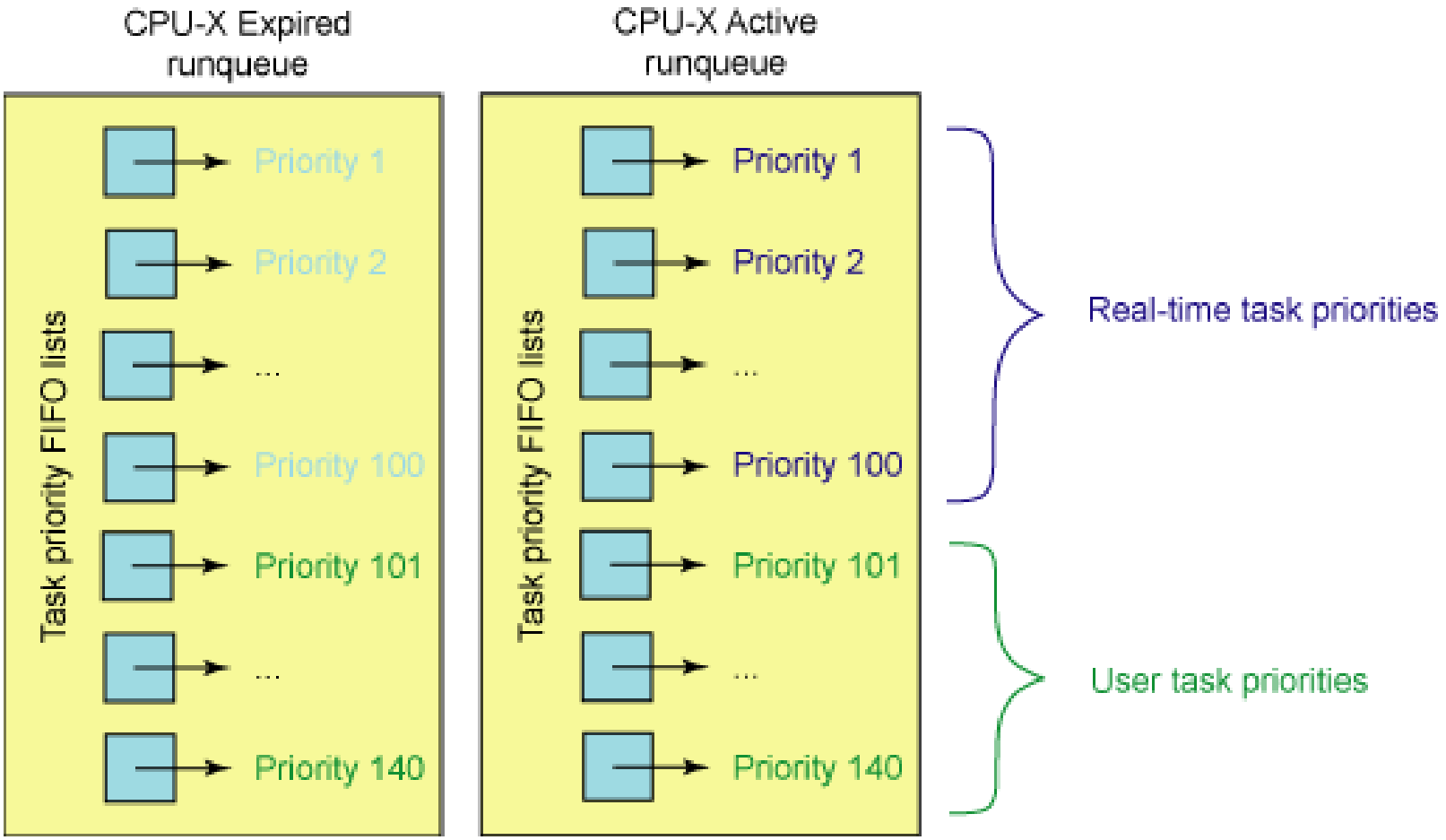
- 可扩展性不好
  - 调度器选择进程时需要遍历整个 runqueue。O(n)
- 高负载系统上的调度性能比较低
  - 调度器预分配给每个进程的时间片比较大，因此在高负载的服务器上，该调度器的效率比较低，因为平均每个进程的等待时间于该时间片的大小成正比。
- 交互式进程的优化并不完善
  - Linux2.4 识别交互式进程的原理基于以下假设，即交互式进程比批处理进程更频繁地处于SUSPENDED状态。然而现实情况往往并非如此：I/O进程vs交互式进程。
- 对实时进程的支持不够
  - Linux2.4内核是非抢占的，当进程处于内核态时不会发生抢占，这对于真正的实时应用是不能接受的。

Case Study: Linux O(1) Scheduler

- Priority-based scheduler: 140 priorities
  - 40 for "user tasks" (set by "nice"), 100 for "Realtime/Kernel"
  - Lower priority value ⇒ higher priority (for nice values)
  - Highest priority value ⇒ Lower priority (for realtime values)
  - All algorithms O(1)
    - Timeslices/priorities/interactivity credits all computed when job finishes time slice
    - 140-bit bit mask indicates presence or absence of job at given priority level
- Two separate priority queues: "active" and "expired"
  - All tasks in the active queue use up their timeslices and get placed on the expired queue, after which queues swapped
- Timeslice depends on priority – linearly mapped onto timeslice range
  - Like a multi-level queue (one queue per priority) with different timeslice at each level
  - Execution split into "Timeslice Granularity" chunks – round robin through priority

O(1)调度器

- Per-core running queue



O(1) Scheduler Continued

- Heuristics
  - User-task priority adjusted ±5 based on heuristics
    - p->sleep_avg = sleep_time − run_time
    - Higher sleep_avg ⇒ more I/O bound the task, more reward (and vice versa)
  - Interactive Credit
    - Earned when a task sleeps for a "long" time
    - Spend when a task runs for a "long" time
    - IC is used to provide hysteresis to avoid changing interactivity for temporary changes in behavior
  - However, "interactive tasks" get special dispensation
    - To try to maintain interactivity
    - Placed back into active queue, unless some other task has been starved for too long…
- Real-Time Tasks
  - Always preempt non-RT tasks
  - No dynamic adjustment of priorities
  - Scheduling schemes:
    - SCHED_FIFO: preempts other tasks, no timeslice limit
    - SCHED_RR: preempts normal tasks, RR scheduling amongst tasks of same priority

O(1)调度器问题

- O(1)调度器区分交互式进程和批处理进程的算法与以前虽大有改进，但仍然在很多情况下会失效
- O(1)调度器对NUMA支持也不完善
- 代码复杂，难以阅读和维护

传统UNIX的进程调度

- 未设置作业调度，进程调度采用基于时间片的多级反馈队列算法，进程优先级分为核心优先级和用户优先级。

调度时机

- 调度由0号进程完成（始终在核心态执行）。时机：
  - 进程由核心态转入用户态时：在每次执行核心代码之后返回用户态之前，检查各就绪进程的优先级并进行调度。如中断——进程回到就绪队列
  - 进程主动放弃处理机时：进程申请系统资源而未得到满足（如read），或进行进程间同步而暂停（如wait或pause），或进程退出（如exit）——进程进入阻塞队列或exit状态。

调度标志

- UNIX System V中有三个与调度有关的标志：
  - runrun：表示要求进行调度，当发现有就绪进程优先级高于当前进程时，设置该标识。在wakeup，setrun, setpri（设置优先级）过程和时钟中断处理例程进行设置。
  - runin：表示内存中没有适当的进程可以换出或内存无足够空间换入一个外存就绪进程。
  - runout：表示外存交换区中没有适当的进程可以换入。

用户优先级

- 进程在用户态和核心态的优先级是不同的，这里说的是用户态进程的优先级。它是基于执行时间的动态优先级，进程优先级可为0~127之间的任一整数。优先数越大，优先级越低。0~49之间的优先级为系统内核保留，用户态下的进程优先级为50~127之间。

- 在UNIX System V中：进程优先数：P_pri = P_CPU / 2 + PUSER + P_nice + NZERO
  - 系统设置部分：PUSER和NZERO是基本用户优先数的阈值，分别为25和20
  - CPU使用时间部分：P_CPU表示该进程最近一次CPU使用时间。每次时钟中断则该值加1（最多可达80）。如果时钟中断的周期为16.6ms，则每秒钟过后将该值为60。
    - 新创建进程的P_CPU值为0，因而具有较高的优先级。
  - 用户设置部分：P_nice是用户可以通过系统调用设置的一个优先级偏移值。默认为20。超级用户可以设置其在0到39之间，而普通用户只能增大该值（即降低优先级）。

核心优先级

- 内核把进程阻塞事件与一个睡眠优先级（0~49）联系起来；当进程从阻塞中醒来时，可及时进行处理。核心优先级分为可中断和不可中断两类优先级。当一个软中断信号到达时，若进程正在可中断优先级上阻塞，则进程立即被唤醒；若正在不可中断优先级上，则继续阻塞。其中：
  - 不可中断优先级：对换，等待磁盘I/O，等待缓冲区，等待文件索引结点——关键操作，应该很快完成
  - 可中断优先级：等待tty（虚终端）I/O，等待子进程退出

调度的实现

- 分三个阶段
  - 检查是否作上下文切换（runrun标志）和核心是否允许作上下文切换（对核心的各种数据结构的操作都已经完成，核心处于正确的状态）。如果允许作上下文切换，则保存当前进程的上下文。
  - 恢复0号进程的上下文，然后执行0号进程，寻找最高优先级的就绪进程，如果没有这样的进程存在，则执行idle过程。如果有这样的进程存在，则该进程作为当前进程分派处理机，保存0号进程的上下文。
  - 恢复当前进程的上下文，执行该进程。

小结

- 调度的类型（如调度单位的不同级别，时间周期，不同的OS），性能准则
- 调度算法：FCFS, SJF, RR, 多级队列，优先级，多级反馈队列
- 调度算法的性能分析：周转时间和作业长短的关系
- 实时调度：调度算法
- 多处理机调度：自调度，成组调度，专用处理机调度

北京航空航天大 学组

2025第11次课堂小测试

北京航空航天大学 OS教学组