OPERATING SYSTEMS

CPU Scheduling

Operating Systems

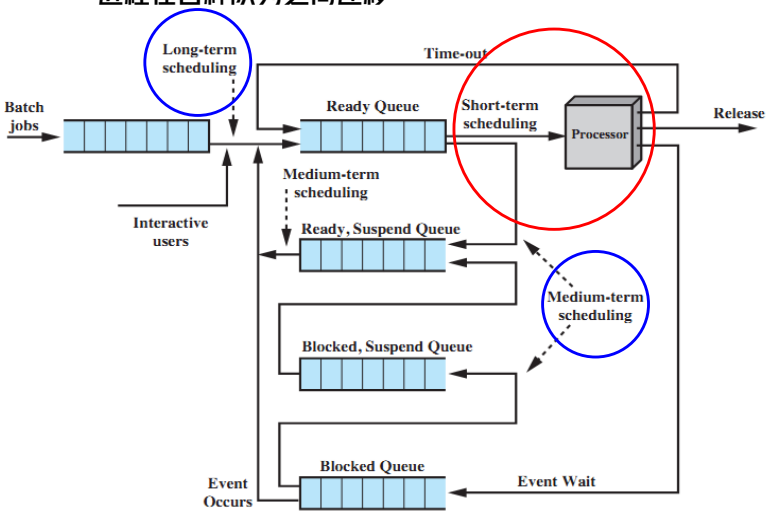
郑贵锋 博士
中山大学计算机学院
zhenggf@mail.sysu.edu.cn
https://gitee.com/code_sysu

中山大學 SUN YAT-SEN UNIVERSITY

- **目录**
  - 基本概念
  - 调度标准
  - 简单调度算法
    - 先到先得FCFS
    - 最短作业优先SJFS
    - 优先级调度
  - 高级调度算法
    - 轮转调度
    - 多优先级队列调度MPQS
    - 多级反馈队列调度MFQS
    - 线程调度
  - 多处理器调度
  - 实时CPU调度
  - 算法评估

Basic Concepts

- **概述**
  - 回顾：进程调度的排队图
    - 进程在各种队列之间迁移



Basic Concepts

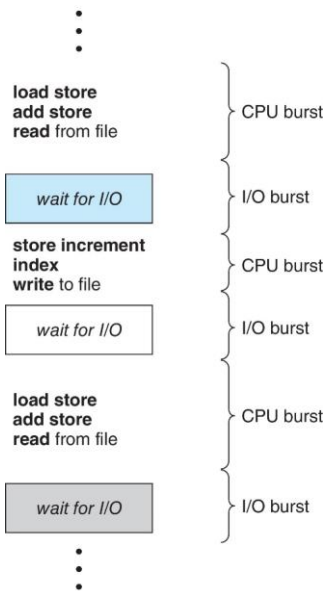
- **概述**
  - 在单CPU情况下，多道程序设计的目标是让进程始终运行，以最大限度地提高CPU利用率。
    - 一次在内存中保留多个进程
    - 每当一个进程必须等待时，另一个进程就可以接管CPU的使用
  - CPU是主要的计算机资源之一。CPU调度是操作系统的一项基本功能，是操作系统设计的核心。
  - 进程调度和线程调度
    - 内核级线程（不是进程）由操作系统调度。
    - 术语“进程调度”和“线程调度”经常互换使用。
      - 在讨论一般调度概念时，我们使用“进程调度”。
      - 我们使用“线程调度”来指代特定于线程的思想。

Basic Concepts

- **CPU-I/O执行周期**
  - 进程执行包括一个CPU执行和I/O等待的周期。进程在这两种状态之间交替。
    - 进程执行开始于一个CPU执行期（burst，突发脉冲）
    - 然后是一个（通常较长的）I/O执行期，然后是另一个CPU执行期，然后是另一个I/O执行期，依此类推。
    - 进程通常在一个CPU执行期结束。
  - CPU执行期分布是主要关注的问题。

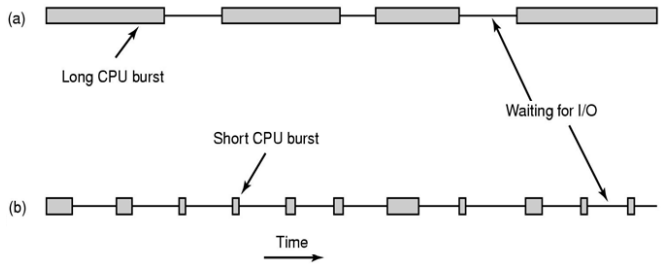
Basic Concepts

- **CPU-I/O执行周期**
  - 一个CPU-I/O执行周期可能由许多次CPU执行和I/O执行组成。
  - CPU执行和I/O执行以交替顺序执行。



Basic Concepts

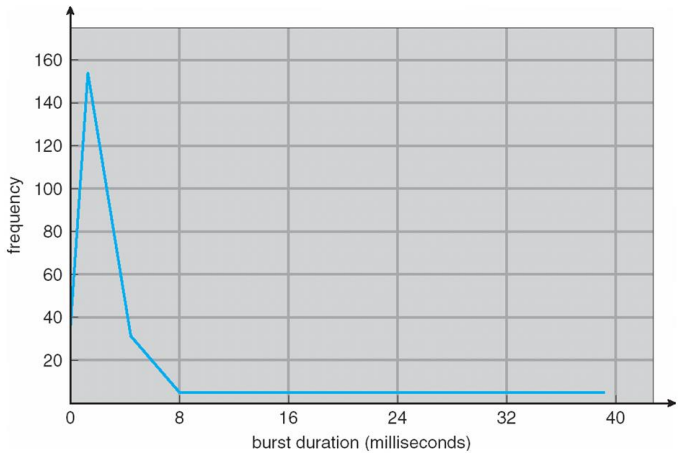
- **CPU-I/O执行周期**
  - CPU执行时间
    - *CPU执行时间*（或服务时间）是一个CPU-I/O执行周期所需的处理器总时间。
  - CPU执行时间长的作业是CPU密集作业，也称为“长作业”。他们可能会有一些长CPU执行。
  - CPU执行时间短的作业是I/O密集作业，也称为“短作业”。它们通常有许多短CPU执行。



长作业(a) vs. 短作业(b)

Basic Concepts

- **CPU-I/O执行周期**
  - CPU执行时间统计直方图
    - 具有大量短CPU执行（小于5ms）和少量长CPU执行



Basic Concepts

- **CPU调度程序Scheduler**
  - 短期调度程序或CPU调度程序决定
    - 接下来选择内存中就绪队列中的哪个进程执行，以及
    - 当CPU空闲时，将CPU分配给该进程。
  - 就绪队列可以实现为FIFO队列、优先级队列、树或简单的无序链表。队列中的记录通常是进程的进程控制块（PCB）。
  - CPU调度决策可能在以下情况下发生：
    1. 从运行状态切换到阻塞状态
    2. 从运行状态切换到就绪状态
    3. 从阻塞状态切换到就绪状态
    4. 终止

Basic Concepts

- **CPU调度程序 Scheduler**
  - 在这四种情况下，有两种决策模式：
    - 非抢占的或合作的
    - 抢占的
  - 非抢占的或合作的
    - 一旦CPU分配给进程，进程将占用CPU，直到它终止或阻塞自身进行I/O。
  - 抢占
    - 当前正在运行的进程可能会被中断并移动到就绪状态。
    - 允许更好的服务，因为任何一个进程都不能长期独占处理器
  - 当前大多数操作系统都采用抢占式调度决策模式。它可能导致竞争条件，在多个进程之间共享数据时引起数据不一致性。

抢占式调度

- 系统调用期间的抢占
  - 在处理系统调用期间，内核可能正忙于代表进程的活动。这些活动可能涉及更改重要的内核数据（例如，I/O队列）。
  - 如果进程在这些更改中间被抢占，且内核（或设备驱动程序）需要读取或修改相同的结构，会发生什么情况？
    - 某些操作系统（包括大多数UNIX版本）通过等待系统调用完成或I/O阻塞发生后再进行上下文切换来处理此问题。
      - 该方案保证了内核结构的简单性，因为当内核数据结构处于不一致状态时，内核不会抢占进程。
      - 它不适合支持实时计算，实时计划要求任务必须在给定的时间范围内完成执行。

抢占式调度

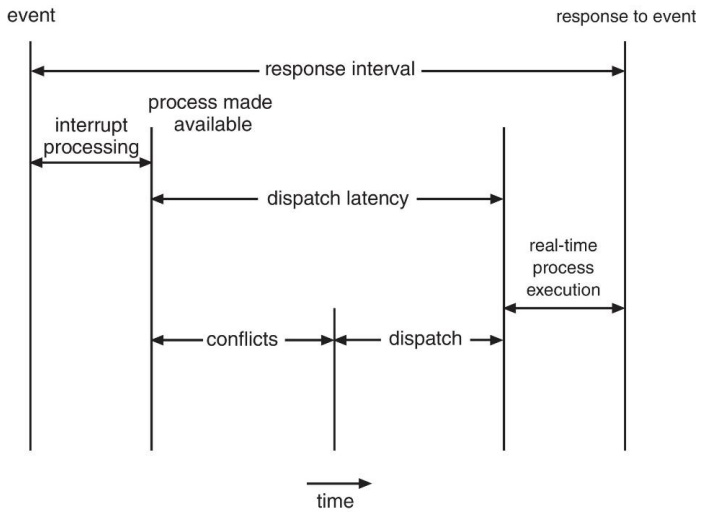
- 中断期间的抢占
  - 根据其定义，中断可以在任何时候发生。
  - 操作系统几乎在任何时候都需要接受中断。
    - 否则，可能会丢失输入或覆盖输出
  - 受中断影响的代码段必须防止同时使用。
    - 在进入时禁用中断和在退出时重新启用中断不会同时访问这些代码段。
    - 需要注意的是，禁用中断的代码段并不经常出现，通常包含很少的指令。

Basic Concepts

- **分发程序Dispatcher**
  - Dispatcher是CPU调度功能中涉及的另一个组件。该模块将CPU控制权交给短期调度程序选择的进程。
  - 分发程序的功能包括：
    - 切换上下文
    - 切换到用户模式
    - 跳转到用户程序中的正确位置以恢复运行该程序
  - 分发延时
    - 分发程序应尽可能快，因为它在每个进程切换期间都会被调用
    - 分发程序停止一个进程并开始另一个进程运行所需的时间称为分发延时。

Basic Concepts

- **分发程序Dispatcher**
  - 分发延时



Scheduling Criteria

- **不同系统的调度目标**
  - 所有系统
    - 公平性
      - 为每个进程分配公平的CPU份额
    - 策略执行能力
      - 确保政策得到执行
    - 平衡能力
      - 使系统的所有部分保持忙碌
  - 批处理系统
    - 吞吐量
      - 最大化每小时的作业数
    - 周转时间
      - 缩短提交和终止之间的时间
    - CPU利用率
      - 让CPU一直处于忙碌状态

Scheduling Criteria

- **不同系统的调度目标**
  - 交互系统
    - 响应时间
      - 快速响应请求
    - 相当性/平衡性
      - 满足用户的期望
  - 实时系统
    - 及时性/截止期限前完成
      - 避免丢失数据
    - 预见性
      - 避免多媒体系统的质量退化

Scheduling Criteria

- **CPU调度标准**
  - 已经提出了许多比较CPU调度算法的标准。使用哪些特征进行比较会对判断哪种算法最好产生重大影响。这些标准包括：
    - CPU利用率
      - 使CPU尽可能忙
    - 吞吐量
      - 每个时间单位完成其执行的进程数
    - 周转时间
      - 从开始到结束执行特定进程的时间量
    - 等待时间
      - 进程在就绪队列中等待的时间量
    - 响应时间
      - 从提交请求到生成第一个响应所需的时间量（对于分时环境）

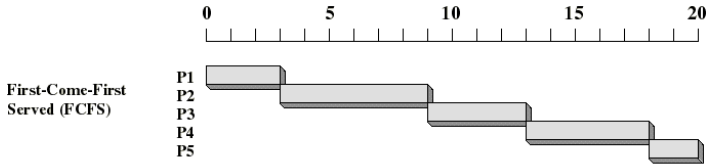
Scheduling Criteria

- **CPU调度标准**
  - 面向用户：
    - 响应时间
    - 周转时间
  - 面向系统：
    - CPU利用率
    - 吞吐量
    - 公平
  - 优化准则
    - 最大化CPU利用率
    - 最大化吞吐量
    - 尽量减少周转时间
    - 尽量减少等待时间
    - 最小化响应时间

Simple Scheduling Algorithms

- **先到先得**
  - **选择函数：在就绪队列中等待时间最长的进程，因此称为先到先服务（FCFS）**
  - **实现：FIFO就绪队列**
  - **决策模式：非抢占**
  - **实例**

进程	到达时间	执行时间
P_1	0	3
P_2	2	6
P_3	4	4
P_4	6	5
P_5	8	2

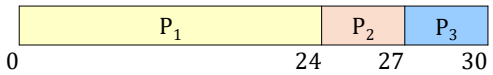


Simple Scheduling Algorithms

- **先到先得**
  - **一个更简单的FCFS示例。**

进程	执行时间
P_1	24
P_2	3
P_3	3

  - **假设进程按顺序到达：** P_1, P_2, P_3. **调度的甘特图为：**



  - **等待时间：** P_1=0； P_2=24； P_3=27.
  - **平均等待时间：**

```
(0 + 24 + 27)/3 = 17
```

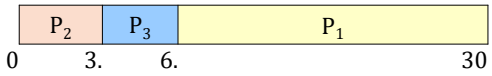
  - ***护航效应：短进程在长进程后。***
    - **考虑一个CPU密集进程和多个I/O密集进程**

Simple Scheduling Algorithms

- **先到先得**
  - 一个更简单的FCFS示例。

进程	执行时间
P_1	24
P_2	3
P_3	3

  - 现在假设进程到达顺序为： P_2, P_3, P_1. 调度的甘特图为：



  - 等待时间：P_1=6；P_2=0；P_3=3。
  - 平均等待时间：

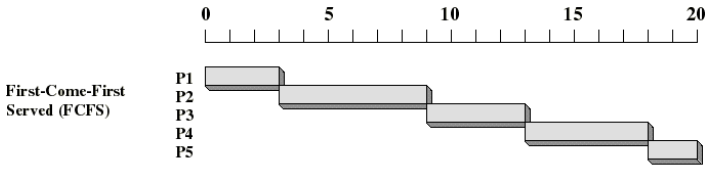
$$(6 + 0 + 3)/3 = 3$$

  - 比刚才的情况好多了

Simple Scheduling Algorithms

- **先到先得**
  - 另一个FCFS示例。

进程	到达时间	执行时间
P_1	0	3
P_2	2	6
P_3	4	4
P_4	6	5
P_5	8	2



  - 等待时间： $P_1 = 0, P_2 = 1, P_3 = 3, P_4 = 7, P_5 = 10.$
  - 平均等待时间：

$$(0 + 1 + 3 + 7 + 10)/5 = 21/5 = 4.2$$

Simple Scheduling Algorithms

- **先到先得**
  - FCFS的缺点
    - 不执行任何I/O的进程将独占处理器。所有其他进程等待（一个大的）进程离开CPU（护航效应）。
    - 对CPU密集的进程有利：
      - I/O密集进程必须等待CPU密集进程完成。
      - 即使I/O完成，I/O设备也可能必须等待（设备利用率低）。
  - 我们本可以给I/O密集的进程多一点优先级，让I/O设备保持忙碌。

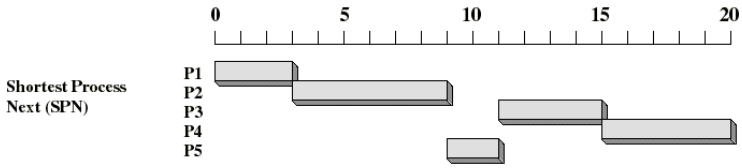
Simple Scheduling Algorithms

- **最短作业优先调度**
  - 选择函数：就绪队列中预期CPU执行时间最短的进程–最短作业优先（SJF），也称为最短时间优先（STF）和最短进程下一个（SPN）
    - 更恰当地说，最短的下一个CPU执行
    - 需要关联（以某种方式估计）每个进程所需的处理时间（下一个CPU执行时间）
    - 如果两个进程的下一个CPU执行相同，则使用FCFS。
  - 决策模式：非抢占或抢占
  - 将首先选择I/O密集的进程
  - SJF是最优的
    - 它给出了一组给定进程的最小平均等待时间。
  - 我们如何知道下一个CPU执行时间？
    - SJF调度经常用于长期调度，其中用户在向批处理系统提交作业时指定处理时间。
    - 对于短期调度，无法知道下一个CPU执行的长度。我们可能必须通过选择具有最短预测下一个CPU执行的进程来近似SJF调度。

Simple Scheduling Algorithms

- **最短作业优先调度**
  - 非抢占的示例

进程	到达时间	执行时间
P_1	0	3
P_2	2	6
P_3	4	4
P_4	6	5
P_5	8	2

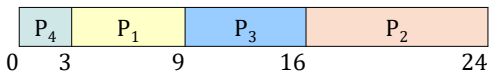


Simple Scheduling Algorithms

- **最短作业优先调度**
  - 一个简单的SJF示例：假设所有进程同时到达

进程	执行时间
P_1	6
P_2	8
P_3	7
P_4	3

  - SJF调度图（决策模式：非抢占）



  - 平均等待时间：

```
(3 + 16 + 9 + 0)/4 = 7.
```

  - 平均周转时间：

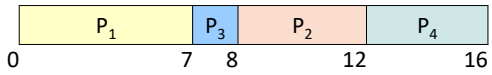
```
(9 + 24 + 16 + 3)/4 = 13.
```

Simple Scheduling Algorithms

- **最短作业优先调度**
  - 另一个SJF示例：具有不同到达时间的进程

进程	到达时间	执行时间
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

  - SJF调度图（决策模式：非抢占）



  - 平均等待时间：

$$[(0-0)+(8-2)+(7-4)+(12-5)]/4 = 4.$$

  - 平均周转时间：

$$[(7-0)+(12-2)+(8-4)+(16-5)]/4 = 8.$$

Simple Scheduling Algorithms

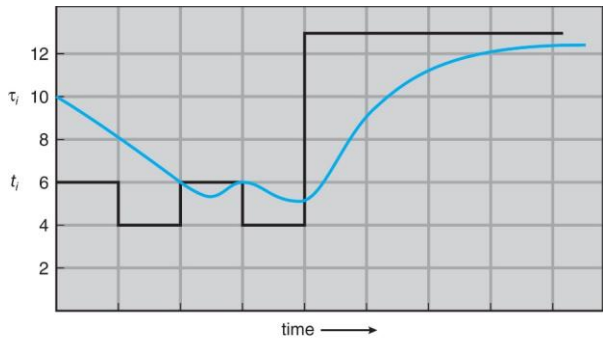
- **最短作业优先调度**
  - 预测下一个CPU执行的长度
    - 进程的下一个CPU执行通常被预测为该进程先前CPU执行的测量长度的**指数平均值**：

(1) t_n=第n个CPU执行的实际长度
(2) τ_{n+1}=下一个CPU执行的预测值
(3) α, $0 \le \alpha \le 1$.
(4) 定义$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

    - 如何设置α？
      - $\alpha = 0$
        - $\tau_{n+1} = \tau_n$
        - 最近的历史记录无效
      - $\alpha = 1$.
        - $\tau_{n+1} = t_n$
        - 只有实际的最后一次CPU执行有效
      - 保持平衡，设置 $\alpha = 0.5$和$\tau_0 = 10$.

Simple Scheduling Algorithms

- **最短作业优先调度**
  - 预测下一个CPU执行的长度



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Simple Scheduling Algorithms

- **最短作业优先调度**
  - 指数平均的思想
    - 展开公式 $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$，得：

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots$$
$$+(1 - \alpha)^j \alpha t_{n-j} + \ldots$$
$$+(1 - \alpha)^{n+1} \tau_0$$

    - 因为 α 与 $(1-\alpha)$ 都小于或等于1，则每个后续项的权重小于其前一项；因此，权重呈指数递减。
    - 这里的指数平均比简单平均好。
  - SJF调度可以是抢占式的，也可以是非抢占式的。
    - 当一个新进程到达就绪队列而前一个进程仍在执行时，就会出现这种选择。新到达的进程的下一个CPU执行可能比当前正在执行的进程的剩余CPU执行短。
    - 抢占式SJF调度有时称为最短剩余时间优先（SRTF）调度。

Simple Scheduling Algorithms

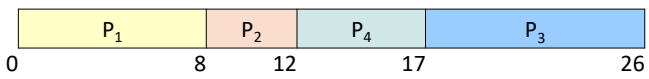
- **最短作业优先调度**
  - 最短工作优先的缺点
    - 只要有较短进程的持续到达，较长进程有饥饿的可能性
    - 非抢占不适合分时环境：
      - CPU密集的进程的优先级较低（这是应该的），但如果一个进程不执行I/O操作，那么它仍然可以独占CPU，前提是它是第一个进入系统的进程。
    - SJF隐式地包含优先级：最短的作业被赋予了优先权。(优先级隐含)

Simple Scheduling Algorithms

- **最短作业优先调度**
  - SJF示例：非抢占式调度

进程	到达时间	执行时间
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

  - SJF调度图（决策模式：非抢占）



  - 平均等待时间：

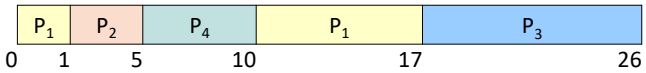
$$[(0-0)+(8-1)+(17-2)+(12-3)]/4 = 7.75$$

Simple Scheduling Algorithms

- **最短作业优先调度**
  - SRTF示例：SJF的抢占式调度

进程	到达时间	执行时间
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

  - SRTF调度图（决策模式：抢占）



  - 平均等待时间：

$$[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 6.5$$

Simple Scheduling Algorithms

- **优先级调度**
  - 优先级编号与每个进程相关联。
  - CPU分配给具有最高优先级的进程。
    - 最小整数=最高优先级
    - SJF是一种优先级调度，其优先级是预测的下一个CPU执行时间
  - 优先级可以在内部或外部定义。
    - 内部定义的优先级使用一些可测量的数量来计算进程的优先级
      - 例如，时间限制、内存要求、打开文件的数量、平均I/O执行与平均CPU执行的比率等。
    - 外部优先级由操作系统之外的标准确定，例如进程的重要性、为计算机使用支付的资金类型和金额、赞助工作的部门以及其他（通常是政治）因素。

Simple Scheduling Algorithms

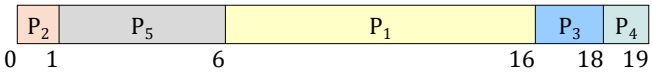
- **优先级调度**
  - 决策模式：抢占或非抢占。
    - 当一个进程到达就绪队列并且这个新到达的进程的优先级高于当前正在运行的进程的优先级时，接下来会发生什么？
      - 抢占式优先级调度将让新到达的进程抢占CPU，将正在运行的进程置于就绪队列的最前面。
      - 非抢占式优先级调度算法将使新进程置于就绪队列的最前面。
  - 饥饿-无限期阻塞
    - 低优先级进程可能永远不会执行。
    - 解决方案：老化策略
      - 逐渐提高系统中等待时间较长的进程的优先级

Simple Scheduling Algorithms

- **优先级调度**
  - 优先级调度示例：所有进程同时到达

进程	执行时间	优先级
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

  - 优先级调度甘特图。



  - 平均等待时间：

$$(6 + 0 + 16 + 18 + 1)/5 = 8.2$$

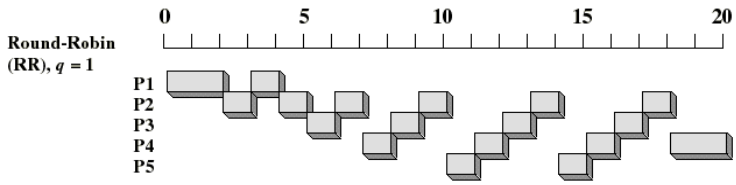
Advanced Scheduling Algorithms

- **轮转调度**
  - 轮转(Round-Robin, RR)调度算法是专门为分时系统设计的。它类似于FCFS调度，但添加了抢占以使系统能够在进程之间切换。
    - 每个进程分配一个小的CPU时间单位（时间量）
    - 一个进程允许运行的，直到达到设置的时间片周期（时间量）。时钟中断发生
    - 正在运行的进程被抢占并添加到就绪队列的末尾，同时将执行上下文切换
  - 计时器在每次时间量都会中断，以调度下一个进程。就绪队列被视为循环队列。CPU调度器依序遍历就绪队列，将CPU分配给每个进程，时间间隔最多为1个时间量。
  - 选择函数：（最初）与FCFS相同
  - 决策模式：抢占

Advanced Scheduling Algorithms

- **轮转调度**
  - 时间量q=1的RR示例

进程	到达时间	执行时间
P_1	0	3
P_2	2	6
P_3	4	4
P_4	6	5
P_5	8	2

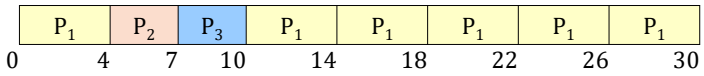


Advanced Scheduling Algorithms

- **轮转调度**
  - 时间量q=4的RR示例

进程	执行时间
P_1	24
P_2	3
P_3	3

- 甘特图是



- 通常，平均等待时间越长，响应越好

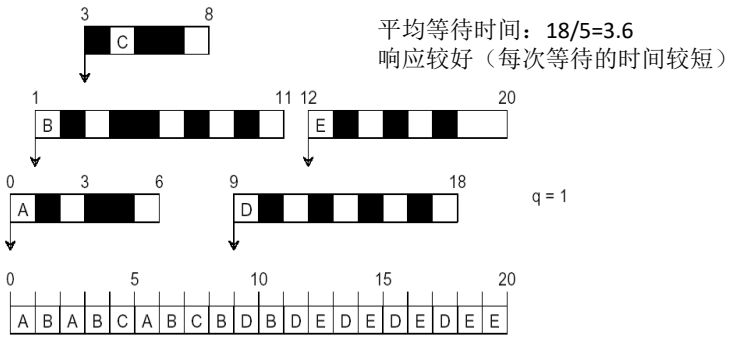
  (6 + 4 + 7)/3 = 5.66

- 与上下文切换时间相比，时间量q应该较大，否则会出现高开销。
  - q值通常在10ms到100ms之间，上下文切换<10μs
  - 极大的q值将使RR调度降级为FCFS

Advanced Scheduling Algorithms

- **轮转调度**
  - 时间量q=1的RR示例

进程	到达时间	执行时间	**等待时间**
A	0	3	3
B	1	5	5
C	3	2	3
D	9	5	4
E	12	5	3

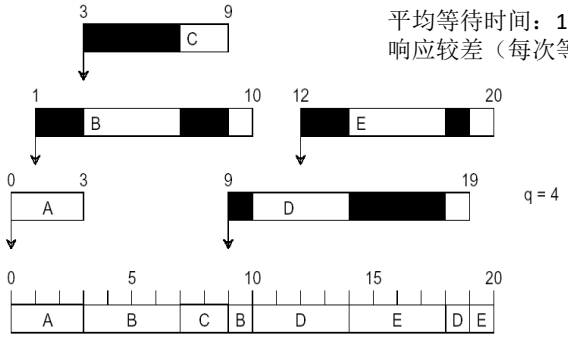


平均等待时间：18/5=3.6
响应较好（每次等待的时间较短）

Advanced Scheduling Algorithms

- **轮转调度**
  - 时间量q=4的RR示例

进程	到达时间	执行时间	等待时间
A	0	3	0
B	1	5	4
C	3	2	4
D	9	5	5
E	12	5	3



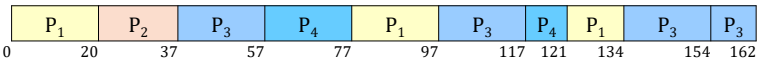
平均等待时间：16/5=3.2
响应较差（每次等待的时间较长）

Advanced Scheduling Algorithms

- **轮转调度**
  - 时间量q=20的RR示例。

进程	执行时间
P_1	53
P_2	17
P_3	68
P_4	24

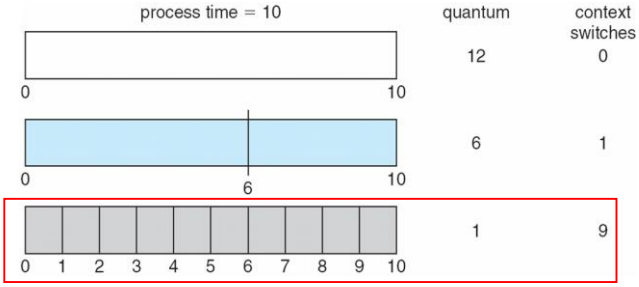
  - 甘特图是



  - 如果就绪队列中有n个进程，且时间量为q，则没有进程等待时间超过 $(n-1) \times q$ 时间单位
    - 最坏的情况是一个进程只需1个时间量，开始就来了，但最后才轮到它执行

Advanced Scheduling Algorithms

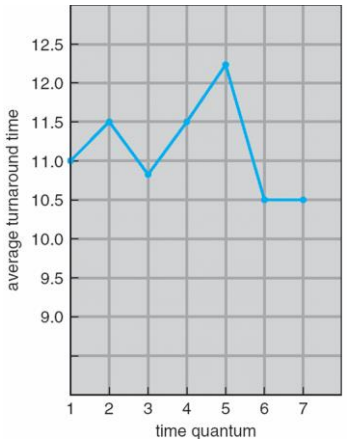
- **轮转调度**
  - 示例：小时间量需更多上下文切换
    - 假设我们只有一个10个时间单位的进程。



    - 如果上下文切换时间约为时间量的10%，则大约10%的CPU时间将用于上下文切换。
    - 实际上，大多数现代系统的时间量从10ms到100ms不等。上下文切换时间通常小于10μs，小于时间量的1/1000.

Advanced Scheduling Algorithms

- **轮转调度**
  - 示例：周转时间随时间量的变化而变化

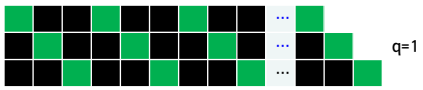


process	time
P_1	6
P_2	3
P_3	1
P_4	7

    - 一组进程的平均周转时间不一定随着时间量大小的增加而增加。
      - 周转时间：从开始到结束执行的时间

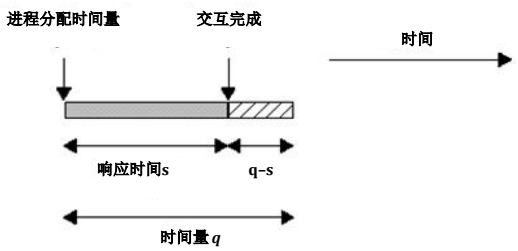
Advanced Scheduling Algorithms

- **轮转调度**
  - 示例：周转时间随时间量的变化而变化
    - 周转时间还取决于时间量的大小。
      - 一组进程的平均周转时间不一定随着时间量大小的增加而增加。
    - 通常，如果大多数进程在单个时间段内完成下一个CPU执行，则平均周转时间可以得到改善。
      - 例如，给定3个进程，每个进程有10个时间单位，每个时间量q=1个时间单位，平均周转时间为29. 但是，如果时间量q=10，则平均周转时间将降至20.
    - 此外，如果计算上下文切换时间，则平均周转时间会在较小的时间量内增加更多，因为需要更多上下文切换。



Advanced Scheduling Algorithms

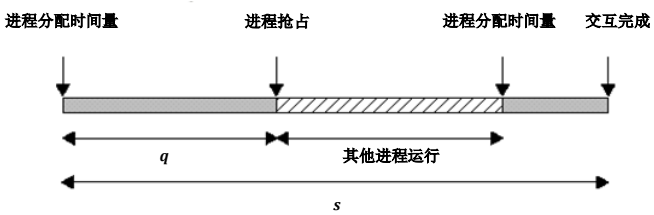
- **轮转调度**
  - 时间量的更多讨论
    - 时间量必须显著大于处理时钟中断和调度所需的时间
      - 它应比典型的交互更大（但不要太大，以避免对I/O密集的进程不利）



时间量大于
典型交互

Advanced Scheduling Algorithms

- **轮转调度**
  - 时间量的更多讨论
    - 时间量必须显著大于处理时钟中断和调度所需的时间
      - 它应比典型的交互更大（但不要太大，以避免对I/O密集的进程不利）



时间量小于
典型交互

Advanced Scheduling Algorithms

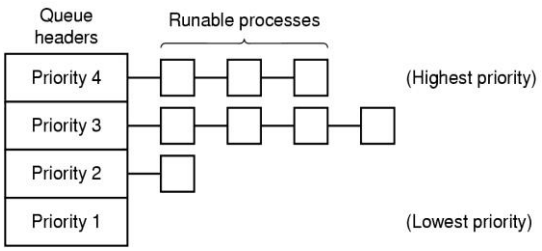
- **轮转调度**
  - 轮转的缺点
    - 仍然对CPU密集的进程有利
      - I/O密集进程使用CPU的时间小于时间量，然后阻塞等待I/O
      - CPU密集的进程在其所有时间量内运行，并被放回就绪队列（从而在阻塞的进程之前）。
    - 一种解决方案——虚拟轮转
      - 当I/O完成时，被阻塞的进程被移动到一个辅助队列，该队列优先于主就绪队列。
      - 从辅助队列调度的进程的运行时间，加上自从主就绪队列中选择该进程以来的运行时间，不得超过基本时间量。

Advanced Scheduling Algorithms

- **多优先级队列调度**
  - 在优先级调度中，每个进程都有一个优先级编号。考虑进程的数量远大于优先级的情况。
  - 多优先级队列调度(MPQS)通过具有多个就绪队列来表示每个优先级级别来实现。
    - 调度程序将始终选择优先级较高的进程，而不是优先级较低的进程。
    - 对于同一优先级就绪队列中的进程，FCFS调度用于选择下一个要分配CPU的进程。
    - 低优先级可能会遭受饥饿。然后允许进程根据其时间或执行历史使用所谓的动态多优先级机制更改其优先级。

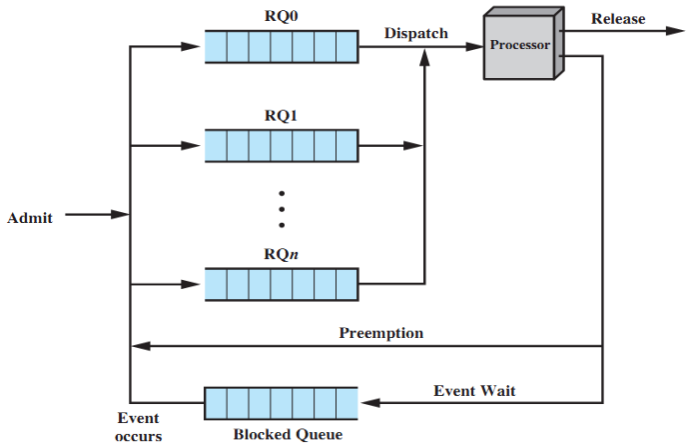
Advanced Scheduling Algorithms

- **多优先级队列调度**
  - 具有队列的优先级调度。



Advanced Scheduling Algorithms

- **多优先级队列调度**
  - 优先级队列

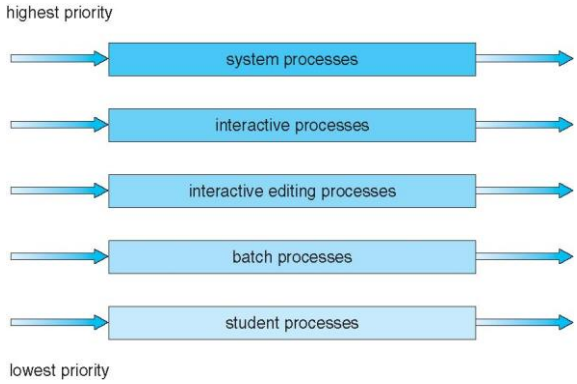


Advanced Scheduling Algorithms

- **多级队列调度**
  - 多级队列调度(MQS)算法将就绪队列划分为多个单独的队列。
    - 进程永久地位于给定队列中，通常基于进程的某些属性，例如内存大小、进程优先级或进程类型。
    - 每个队列都有自己的调度算法。
    - 例如，就绪队列被划分为单独的队列：
      - 由RR算法调度的前台（交互式）进程
      - 由FCFS算法调度的后台（批处理）进程
  - 必须在队列之间进行调度。
    - 固定优先级抢占式调度
      - 例如，前台队列可能比后台队列具有绝对优先级，后台队列可能会饿死。
    - 时间片——每个队列获得一定数量的CPU时间，可以在其各个进程之间进行调度
      - 例如，使用RR的前台队列占80%时间；使用FCFS的后台队列占20%时间

Advanced Scheduling Algorithms

- **多级队列调度**
  - 示例：具有5个队列的多级队列调度算法，以下按优先级顺序列出



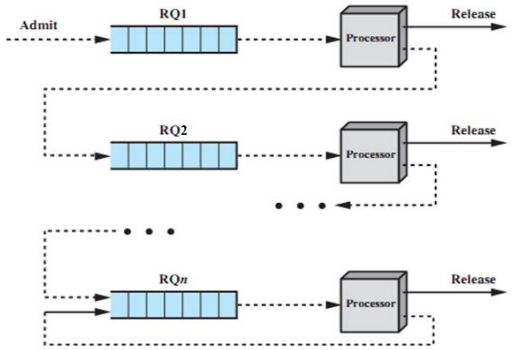
  - 如果交互式编辑进程在批处理进程运行时进入就绪队列，则批处理进程将被抢占。

Advanced Scheduling Algorithms

- **多级反馈队列调度**
  - 多级反馈队列调度MFQS算法允许进程在不同级别的队列之间移动
    - 其思想是根据进程的CPU执行特征来分离进程。如果一个进程占用了太多的CPU时间，它将被移动到优先级较低的队列中。该方案将I/O密集和交互进程保留在高优先级队列中。
    - 此外，在低优先级队列中等待时间过长的进程可能会移动到高优先级队列。这种形式的老化可以防止饥饿。
    - 多级反馈队列调度程序由以下参数定义：
      - 队列的数量
      - 每个队列的调度算法
      - 当进程需要服务时，用于确定其将进入哪个队列的方法
      - 用于确定何时升级进程的方法
      - 用于确定何时降级进程的方法

Advanced Scheduling Algorithms

- **多级反馈队列调度**
  - 实例



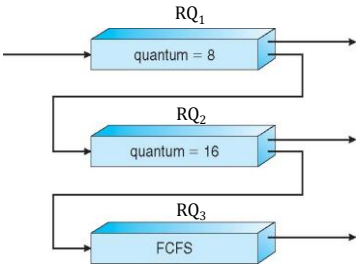
Advanced Scheduling Algorithms

- **多级反馈队列调度**
  - 实例
    - 几个优先级降低的就绪队列 RQ_1，RQ_2，…，RQ_n：

      $P(RQ_1) > P(RQ_2) > \ldots > P(RQ_n)$
    - 新进程被放置在RQ_1中
    - 当它们到达时间量点时，被放置在RQ_2中。如果它们再次到达，将被放置在RQ_3，…，直到它们到达RQ_n
    - I/O密集的进程将倾向于停留在优先级较高的队列中。CPU密集的作业将向下迁移
    - 仅当RQ_1到RQ_{i-1}为空时，调度程序Dispatcher才会在RQ_i中选择一个进程执行
    - 因此，长作业可能会饥饿

Advanced Scheduling Algorithms

- **多级反馈队列调度**
  - 实例
    - 考虑三个就绪队列：
      - RQ_1–具有8ms时间量和FCFS的RR
      - RQ_2–具有16ms时间量和FCFS的RR
      - RQ_3–FCFS
    - 调度：
      - 新作业P进入队列RQ_1，队列RQ_1由FCFS提供服务。当它得到CPU时，作业P执行8ms. 如果作业P未在8ms内完成，则移动到队列RQ_2的尾部
      - 在RQ_2，作业P再次被FCFS服务，并执行16ms. 如果它仍然没有完成，它将被抢占并移动到队列RQ_3的尾部



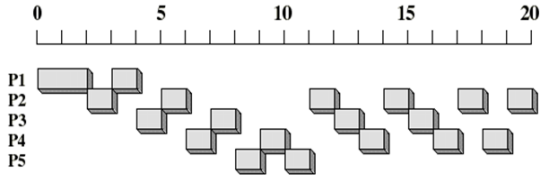
Advanced Scheduling Algorithms

- **多级反馈队列调度**
  - 反馈调度的时间量
    - 在固定的时间量下，较长进程的周转时间可能会令人担忧地延长。
    - 为了补偿这一点，我们可以根据队列的深度增加时间量：
      - 示例：将第i个就绪队列的时间量定义为

        $q_i=2^{i-1}$
    - 有关示例，请参见下一张幻灯片
    - 更长的进程可能仍然会遭受饥饿
      - 可能的解决方法：一段时间后将进程提升到更高的优先级

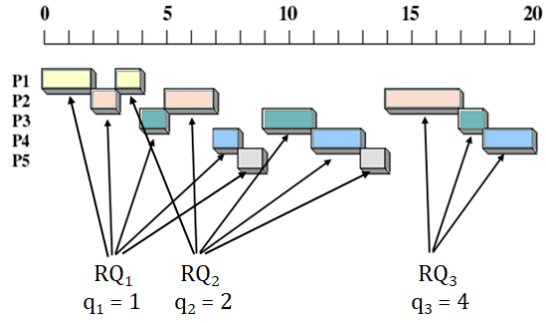
Advanced Scheduling Algorithms

- **多级反馈队列调度**
  - 示例：q=1的反馈调度



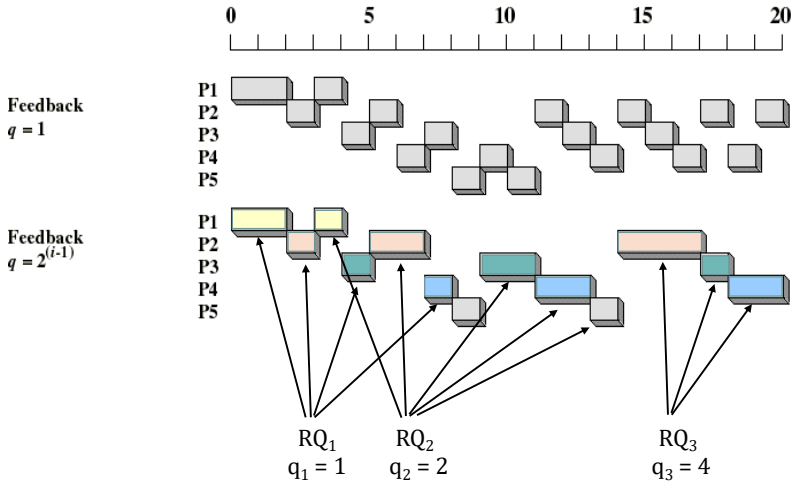
Advanced Scheduling Algorithms

- **多级反馈队列调度**
  - 示例：$q_i=2^{i-1}$的反馈调度



Advanced Scheduling Algorithms

- **多级反馈队列调度**
  - 示例：$q=1$和$q_i=2^{i-1}$的反馈调度



Threads Scheduling

- **线程调度**
  - 在支持用户级线程ULTs和内核级线程KLTs的操作系统上，由操作系统调度的是KLTs, 而不是进程。ULTs由线程库管理。要在CPU上运行，ULTs最终必须映射到相关的内核级线程，尽管此映射可能是间接的，可能使用轻量级进程(LWP)
  - 进程竞争域
    - 在实现多对一和多对多模型的系统上，线程库调度ULTs在可用LWP上运行。
    - 此方案称为进程竞争域(PCS，process-contention scope), 因为CPU的竞争发生在同一进程中的线程之间。
      - 当我们说线程库将ULTs(逻辑上)调度到可用的LWP上时，并不意味着线程实际上是在CPU上运行的。这需要操作系统将内核线程调度到物理CPU上。

Threads Scheduling

- **线程调度**
  - 进程竞争域（续）
    - 通常，PCS是根据优先级完成的——调度程序选择具有最高优先级的可运行线程来运行。ULT优先级由程序员设置，不由线程库调整，尽管某些线程库可能允许程序员更改线程的优先级。需要注意的是，PCS通常会抢占当前运行的线程，以支持更高优先级的线程；然而，不能保证在具有相同优先级的线程之间进行时间切片。
  - 系统竞争域
    - 内核使用系统竞争域（SCS，system-contention scope）决定将哪个KLT调度到CPU上。使用SCS调度的CPU竞争发生在系统中的所有线程之间。
      - 使用一对一模式的系统，如Windows、Linux和Solaris，只使用SCS调度线程。

Threads Scheduling

- **线程调度**
  - Pthreads调度
    - Pthreads使用以下值来标识竞争域

      PTHREAD_SCOPE_PROCESS
        - 使用PCS调度来调度线程

      PTHREAD_SCOPE_SYSTEM
        - 使用SCS调度来调度线程
    - Pthread IPC提供两个函数来获取和设置竞争域策略

```
pthread_attr_setscope(pthread_attr t *attr, int scope)
pthread_attr_getscope(pthread_attr t *attr, int *scope)
```

    - 下一张幻灯片展示了Pthread调度API的例子：它首先确定现有的竞争域，将其设置为PTHREAD_SCOPE_SYSTEM. 然后，它创建5个单独的线程，这些线程将使用SCS调度策略运行。
    - 注意，在某些系统上，只允许某些竞争域值。例如，Linux和Mac OS系统只允许PTHREAD_SCOPE_SYSTEM.

Threads Scheduling

- **线程调度**
  - **示例：** Pthreads调度API
    - pthread_scope.c

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *runner(void *);
int main(int argc, char *argv[])
{
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
      /* get the default attributes */
    pthread_attr_init(&attr);
      /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("\nscope = PTHREAD_SCOPE_PROCESS\n");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("\nscope = PTHREAD_SCOPE_SYSTEM\n");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
```

Threads Scheduling

- **线程调度**
  - **示例：** Pthreads调度API
    - pthread_scope.c (2)

```
      /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
      /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, &runner, NULL);
      /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

  /* Each thread will begin control in this function */
void *runner(void *param)
{
      /* do some work ... */
    pthread_exit(0);
}
```