进程调度（Scheduling）

- 进程（Linux中称任务）定义：是一个可并发执行的具有独立功能的程序关于某个数据集合的一次执行过程，也是操作系统进行资源分配和保护的基本单位。

- 描述进程的三个方面：
  - 程序的一次运行活动；
  - 进程的运行活动是建立在某个数据集合之上的；
  - 进程在获得资源的基础上从事自己的运行活动。

Before Scheduling

- How to switch from one process to another
- how to understand interruption
- How interruption works

Context switch and Interruption

- User process is computing
  - Count = read(file, buf, sizeof[buf])
- User process stops running
- OS issues disk read
- .....
- OS copies data to user buffer
- User process starts running again

Context switch and Interruption

- User process A: read()
  - trap to kernel mode

- Kernel: tell disk to read sector 346761
  - uses IVT(interrupt vector table) to handle the interrupt
  - Disk interrupt => invoke disk driver

唤醒

- Kernel switch to user process B
  - return to user mode, but not for A
  - A is blocked in a system call
    - unable to execute more instructions

- User process B: computes whatever it wants

Context switch and Interruption

- Disk: done!
  - asserts an interrupt signal
  - CPU stops running B
  - interrupts to kernel mode
  - run "disk interrupt handler" code

- Kernel: switch to user process A
  - return from interrupt, to user mode, for A, not for B
  - B is able to execute, but not executing now
    - B is not running
    - B is not blocked
    - B is ready

进程调度

- 程序与进程之间的区别：

1.“进程”是一个动态的概念:进程强调的是程序的一次“执行”过程，程序是一组有序指令的集合，在多道程序设计环境下，它不涉及“执行”，因此,是一个静态的概念；

2.不同的进程可以执行同一个程序:即使多个进程执行同一个程序，只要它们运行在不同的数据集合上，它们就是不同的进程；

3.每一个进程都有自己的生命期:当系统要完成某一项工作时，它就“创建”一个进程，程序执行完毕，系统就“撤销”这个进程，收回它所占用的资源。

进程调度

- 进程的特征：

1.进程之间具有并发性:在一个系统中，同时会存在多个进程。于是与它们对应的多个程序同时在系统中运行，轮流占用CPU和各种资源。

2.进程间会相互制约:由于进程是系统中资源分配和运行调度的单位，因此在对资源共享和竞争中，必然会相互制约，影响了各自向前推进的速度。

- 进程调度算法的原则：

（1）公平性。

（2）资源利用率（特别是CPU利用率）。

（3）响应时间 - 交互式系统情况。

（4）系统吞吐量 - 批处理系统。

（5）周转时间 – 从进程提交到进程完成的时间间隔。

（6）等待时间 – 在就绪队列中等待所花费的时间之和

抢占还是非抢占?

两种占用CPU的方式:

1.可剥夺式（可抢占式Preemptive）：当有比正在运行的进程优先级更高的进程就绪时，系统可强行剥夺正在运行进程的CPU，提供给具有更高优先级的进程使用。

2.不可剥夺式（不可抢占式Nonpreemptive ）：某一进程被调度运行后，除非由于它自身的原因不能运行，否则一直运行下去。

- CPU调度决策(调度时机)可在如下四种环境下发生，当一个进程：
  1. 从运行状态切换到等待状态
  2. 从运行状态切换到就绪状态
  3. 从等待状态切换到就绪状态

  终止时

  - 如，在进程通信中，执行中的进程执行了某种原语操作（P操作，阻塞原语，唤醒原语），时间片时间到，等待I/O…

  - 当调度只能发生在 1 和 4 两种情况时，称调度方案是非抢占式调度。
    - 如Microsoft Windows 3.1, Apple Macintosh

  - 否则，称为可抢占式调度
    - 如UNIX, Linux, Windows NT/2000

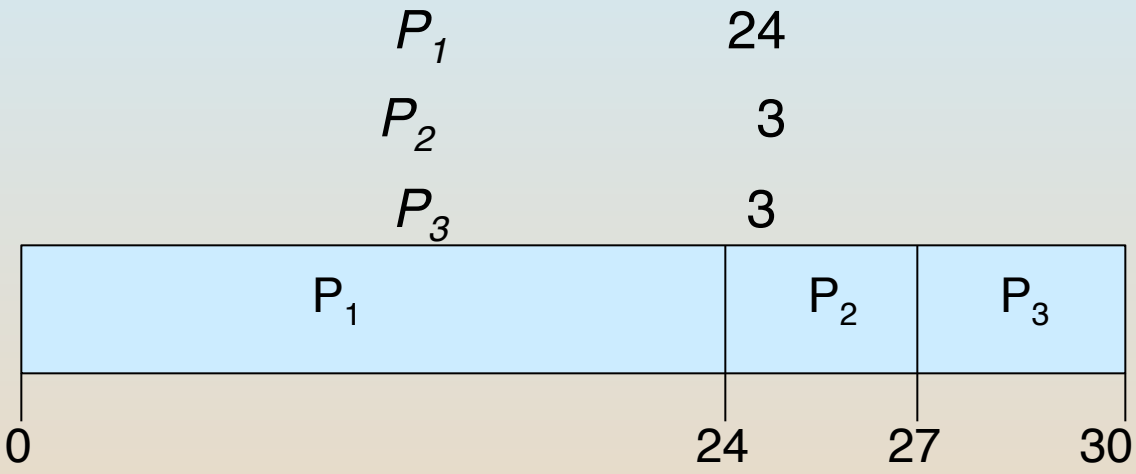
- 各种进程调度算法:

先进先出进程调度算法（FIFO）：按照进程就绪的先后次序来调度进程。

优点:实现简单。

缺点:没考虑进程的优先级。

例子



- 等待时间 P1 = 0; P2 = 24; P3 = 27
- 平均等待时间: (0 + 24 + 27)/3 = 17

- 短作业优先调度算法
  - schedule the process with the shortest time cost
  - 两种占用CPU的方式:
    - nonpreemptive
    - preemptive
  - SJF is optimal – gives minimum average waiting time for a given set of processes

基于优先级的调度算法：优先选择就绪队列中优先级最高的进程投入运行。优先级根据优先数来决定。

静态优先数法：在进程创建时指定优先数，在进程运行时优先数不变。

动态优先数法：在进程创建时创立一个优先数，但在其生命周期内优先数可以动态变化。如等待时间长优先数可改变。

优先级定义

- 内部因素
  - 时间极限
  - 内存要求
  - 打开文件的数量
  - 平均I/O时间区间与平均CPU区间之比
- 外部因素
  - 进程重要性
  - 使用计算机支付的费用
  - 赞助工作的单位
  - 其它因素

基于优先级的调度算法:

潜在问题?

Starvation – low priority processes?

解决方法

Aging - as time progresses increase the priority of the process

时间片轮转调度算法：

1. 专门为分时系统而设计，类似于FCFS调度，但增加了抢占以在进程间切换
   - 每个进程获得一小段CPU时间 (时间量, 或 时间片)
     - 通常是 10-100 ms
   - 时间片用完后，CPU被抢占，进程排在就绪队列末尾
     - 就绪队列可看作循环的 FIFO 队列
     - 新进程放在就绪队列末尾
   - 如果就绪队列中有n个进程且时间片为q，那么每个进程会得到1/n的CPU时间，每个长度不超过q时间单元
   - 每个进程必须等待CPU的时间不会超过 (n-1)*q 个时间单元

2.时间片选择问题：

固定时间片；

可变时间片。

3.与时间片大小有关的因素：系统响应时间；就绪进程个数； CPU能力。

时间片轮转调度算法的性能

- 性能很大程度上取决于时间片的大小
  - 如果时间片非常大 ⇒ FIFO（FCFS）
  - 如果时间片很小 ⇒ 响应时间短，但如果时间片过小，上下文切换开销过大

多级队列反馈调度算法：

- 允许进程在队列之间移动，以此实现老化
- 多级反馈队列调度定义了如下的参数：
  - 队列数量
  - 每个队列的调度算法
  - 用以确定进程何时升级到较高优先权队列的方法
  - 用以确定进程何时降级到较低优先权队列的方法
  - 用以确定进程在需要服务时应进入那个队列的方法

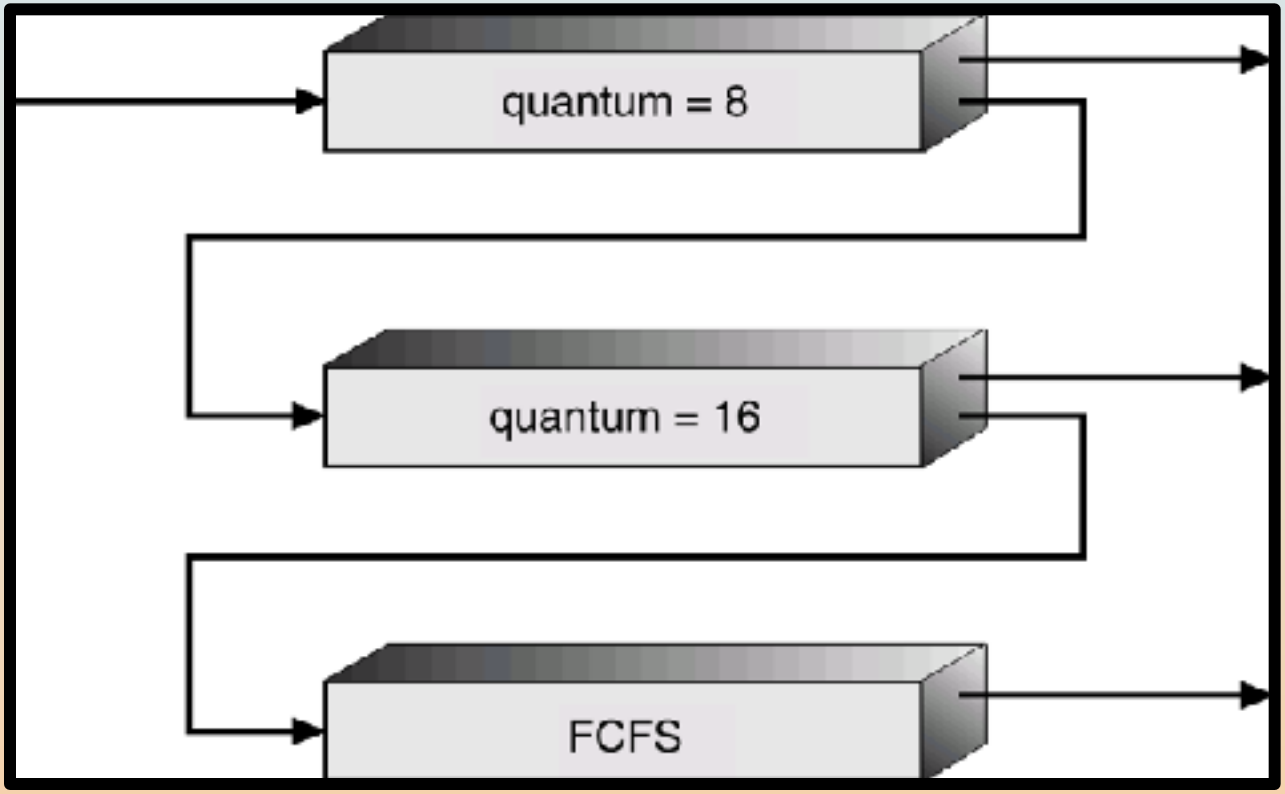
多级反馈队列举例

- 三个队列:
  - Q0 – 8ms的时间片
  - Q1 – 16ms的时间片
  - Q2 – FCFS

- 调度:
  - 新进程进入就绪队列时被放在队列0内，队列0的每个进程都有8ms的时间片。如果一个进程不能在这一个时间内完成，那么它就被移到队列1的尾部。

  - 队列1按照FCFS调度，当进程获得调度时，将获得16ms的时间片，如果没有完成，将被抢占并被移动到队列2。

多级反馈队列举例



多级队列反馈调度算法:

* 系统中设置多个就绪队列。
* 每个就绪队列分配给不同时间片，优先级高的为第一级队列，时间片最小，随着队列级别的降低，时间片加大。

* 各队列按照先进先出调度算法。
* 一个新进程就绪后进入第一级队列。
* 进程由于等待而放弃CPU后，进入等待队列，一旦等待的事件发生，则回到原来的就绪队列。
* 当有一个优先级更高的进程就绪时，可以抢占CPU，被抢占进程回到原来一级就绪队列末尾。
* 当第一级队列空时，就去调度第二级队列，如此类推

当时间片到后，进程放弃CPU，回到下一级队列。

Discussion

- SJF/SRTF are the best you can do at minimizing average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF

- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - SRTF (and RR): short jobs not stuck behind long ones

算法评估

- 问题：
  - 如何为特定系统选择CPU调度算法?
  - 那个算法较好?
- 回答：
  - 定义准则
    - CPU利用率、响应时间、吞吐量等
    - 定义这些度量值的相对重要性
    - 基于权重的平均值
  - 基于准则评估性能

确定性建模

- 采用特定预先确定的系统负荷，定义在给定负荷下每个算法的性能。
  - example P130
    - 给定负荷
    - 准则:最小的平均等待时间
    - 比较FCFS, SJF和RR (时间片为10)
- 简单快速
- 要求输入精确数字
- 主要用途在于描述调度算法和提供例子

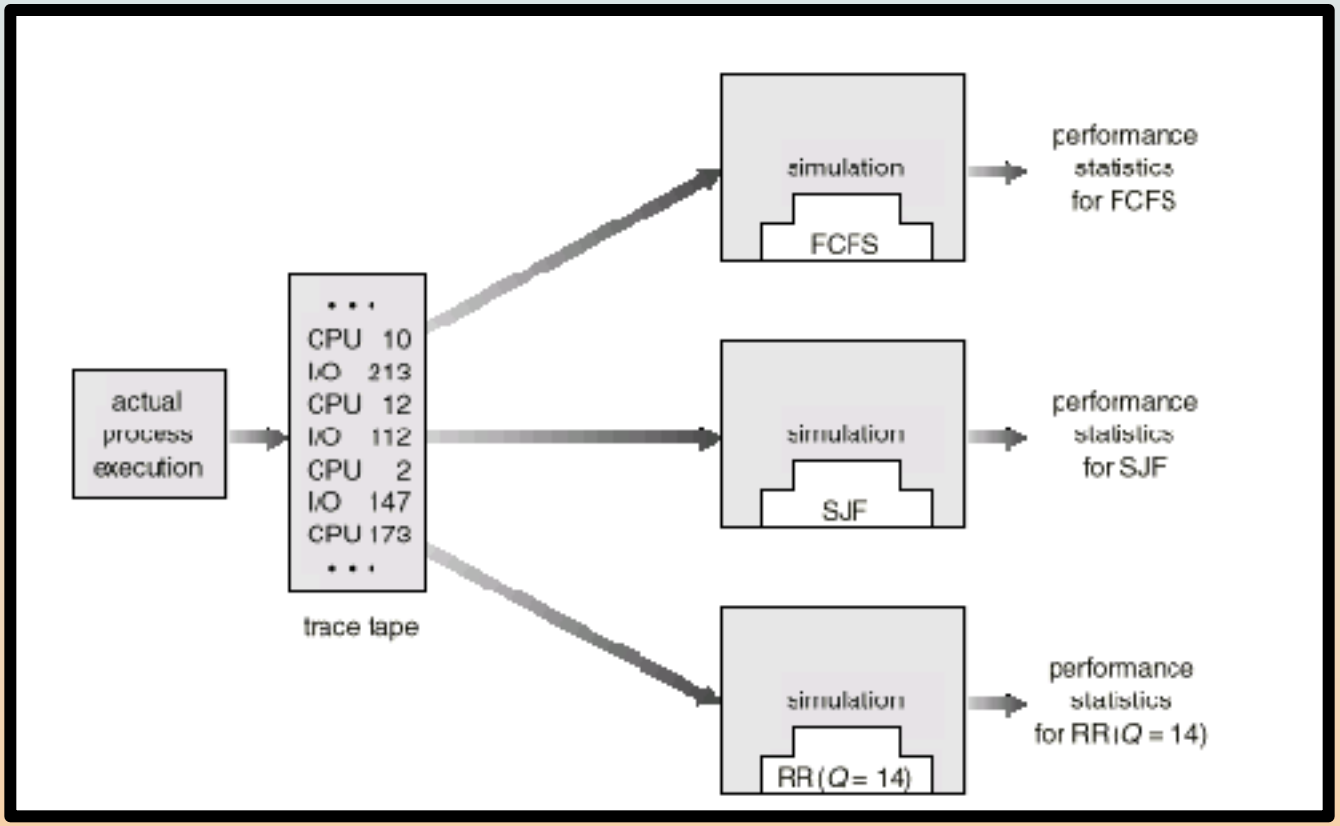
排队模型

- 系统负荷每时每刻都在变化
- CPU和I/O区间的分布却是可以确定。
  - 通常这种分布是指数的
- 到达率和服务率 => 计算使用率、平均队列长度、平均等待时间等
- 然而，排队模型仅仅是真实系统的近似模拟。

Simulation (模拟)

- 模拟涉及对计算机系统模型进行程序设计
- 模拟程序
  - 通常是事件驱动
  - 内部时钟
- 随机数生成器
  - 根据概率分布生成进程、CPU区间时间、到达时间、离开时间等
- 可采用跟踪磁带纠正驱动模拟不够精确的问题

通过模拟来评估CPU调度算法



Implementation (实现)

- 精确的评估方法是具体实现调度算法，并将其放在操作系统内，并观测它如何工作。
  - 将真实的算法放进实际系统然后在真实操作系统条件下对它进行评估
- 最大的困难是代价：需要编码实现
- 其它的困难
  - 变化的环境
  - 根据用户调整调度策略

Java 线程调度简介

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm
- FIFO Queue is Used if There Are Multiple Threads With the Same Priority

Java 线程调度简介

JVM Schedules a Thread to Run When:

1. The Currently Running Thread Exits the Runnable State
2. A Higher Priority Thread Enters the Runnable State

* Note – the JVM Does Not Specify Whether Threads are Time-Sliced or Not

小结

- CPU调度：从就绪队列中选择一个等待进程，并为其分配CPU。

- 调度算法
  - FIFO (FCFS)
  - 短作业优先shortest job first (SJF)
    - 被证明最优，但难以确定下一个CPU区间的长度
  - 优先权调度 priority based scheduling
    - 饥饿和老化
  - 时间片轮转round-robin (RR)
    - 多用于分时、交互式系统
    - 问题是，如何选择时间片大小
  - 多级反馈队列
    - 允许进程从一个队列移动到另外一个

小结

- 调度算法：抢占还是不抢占?
- 调度时机和执行过程
- 分析和选择算法
  - 确定性模型
  - 排队模型
  - 模拟
  - 实现