第5讲

多处理器与分布式系统中的实时任务调度

安吉尧

多处理器系统

- ***Multiprocessor Systems***指包含两台或多台功能相近的处理器，处理器之间彼此可以交换数据，所有处理器共享内存，I/O设备，控制器，及外部设备，整个硬件系统由统一的操作系统控制，在处理器和程序之间实现作业、任务、程序、数组及其元素各级的全面并行

分布式系统

- **Distributed System是建立在网络之上的软件系统**
- 与网络相比，其区别不在硬件而在高层软件（尤其是OS）
- 分布式系统具有高度的内聚性和透明性
  - 内聚性是指每一个数据库分布节点高度自治，有本地的数据库管理系统
  - 透明性是指每一个数据库分布节点对用户的应用来说都是透明的，看不出是本地还是远程

多处理器和分布式系统

- 多处理器系统是紧密耦合（tightly coupled）的，共享物理内存，即进程间通信的开销和任务运行时间相比可以忽略不计，因为通信是由向共享内存中的读写完成的
- 分布式系统是松散耦合（loosely coupled）的，不共享物理内存，进程间通信的开销和任务运行时间相当
- 多处理器系统可以用集权调度而分布式系统不可以。集权调度可以把系统中各种任务的状态维护在一个集权式的数据结构里。当任务状态变化时，系统中各个处理器需要更新系统，这可能导致过高的通信开销

多处理器和分布式系统（续）

- 价格下降
- 普及性
- 快速响应能力
- 容错能力
- 调度更复杂
  - 单处理器最佳实时调度算法呈多项式复杂度（即P问题）
  - 多处理器和分布式系统上确定最佳实时调度却是一个NP难问题

实时任务调度的变化

- 在多处理器和分布式系统上的实时任务调度由两个子问题组成
  - 处理器任务分配
    - 如何把一系列任务分开，并分配给处理器
    - 动态or静态
      - 静态算法中，任务被分配到子系统中，每一个子系统和一个独立的处理器相关
      - 动态算法中，待执行的任务被放在一个普通的优先级队列中，并在处理器空闲时把任务分配出去
  - 单个处理器上的任务调度

多处理器任务静态分配

- 在静态算法中，任务在运行前就被分配，且这种分配在系统的整个运行过程中始终有效
- 对于多处理器的任务分配没有尝试去减小程序通信的开销，因为多处理器中通信时间等同于内存访问时间
  - 1、使用率平衡（Utilization Balancing）算法
  - 2、RMA的循环首次适应（Next-Fit）算法
  - 3、EDF的封箱（Bin Packing）算法

1、Utilization Balancing

- 首先将任务按照使用率的递增顺序进行排序
- 将队列头部的任务分配给使用率最低的处理器，以平衡不同处理器之间使用率
  - 在一个理想的平衡系统中，每个处理器的使用率u_i等于系统中所有处理器的均值
- 分配给处理器Pi的任务的使用率总和 $u_i=\sum_{j\in STi} u_{tj}$

  其中STi是分配给处理器Pi的所有任务，u_{tj}是任务Tj的使用率
- 在使用不同的处理器来处理不确定任务的系统中，本算法较难实现使用率的平衡，即难以在每个处理器Pi上实现 $u_i = \bar{u}$
- 所有好的使用率平衡算法的目的都是减小 $\sum_{i=1}^{n}\left|(u_i-\bar{u})\right|$

  其中n是系统中的处理器个数，$\bar{u}$ 是处理器的平均使用率，u_i是处理器i的使用率
- 本算法适合处理器个数确定的情况，在单个处理器上则使用EDF

2、Next-Fit

- 循环首次适应算法(Next-Fit)
- 内存管理中的算法

Q：什么是Next-Fit算法？

2、Next-Fit

- 根据任务的使用，将一系列任务分为多组，每组任务使用**RMA**分配到一个处理器上
- 本算法旨在尽量减少处理器的使用个数，无需提前知晓处理器的个数
- 一个或多个处理器被指定处理一类任务，即实质上是使具有相近使用的任务被分配给相同的处理器
- 假设任务分为m类，任务t_j属于类j,0<j<m,当且仅当 $(2^{\frac{1}{j+1}}-1) < e_i / p_i \leq (2^{\frac{1}{j}}-1)$

- 假设我们希望把一个系统中的任务分为四类，根据上式，由任务的使用可计算出各个类：
  - 类1 $(2^{\frac{1}{2}}-1) < C_1 \le (2^{\frac{1}{1}}-1)$
  - 类2 $(2^{\frac{1}{3}}-1) < C_2 \le (2^{\frac{1}{2}}-1)$
  - 类3 $(2^{\frac{1}{4}}-1) < C_3 \le (2^{\frac{1}{3}}-1)$
  - 类4 $0 < C_3 \le (2^{\frac{1}{4}}-1)$

  由此可以得到各个类的使用范围（执行域）：

  类1：（0.41 ,1]，类2：(0.26,0.41]，

  类3(0.19，0.26]，类4（0，0.19]

  注意：对执行时间久的任务划分的域比较大

  仿真研究表明：

  使用本算法所需的处理器个数是理想个数的2.34倍

Next-Fit算法举例

- 下面的表格显示了10个周期性实时任务的执行时间(毫秒计)和周期时间(毫秒计)。假设这些任务在有4个处理器的系统上执行。使用Next-Fit分配它们，在每个处理器上使用RMA算法调度任务。

Task	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
ei	5	7	3	1	10	16	1	3	9	17
pi	10	21	22	24	30	40	50	55	70	100

课堂练习：

运用Next-Fit算法，上述任务集该如何分配到这4个处理器中？

根据Next-Fit算法公式计算各个各个任务的使用率和分类如下表所示：

Task	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
ei	5	6	3	1	10	16	1	3	9	17
Pi	10	21	22	24	30	40	50	55	70	100
Ui	0.5	0.33	0.14	0.04	0.33	0.4	0.02	0.05	0.13	0.17
Class	1	2	4	4	2	2	4	4	4	3

3、结合EDF的Bin Packing算法

课堂练习：

什么是Bin Packing算法？

3、结合EDF的Bin Packing算法

- 该算法将任务分配给处理器，使得每个处理器可以使用**EDF**来调度分配给它的任务
- 分配任务时，任意处理器的使用率不能超过**1**
- 将任务分配问题当做装箱问题来解决
  - 装箱问题（**Bin Packing**）是一个经典的组合优化问题
  - 设有许多具有同样结构和负荷的箱子 **B1，B2，…，**其数量足够供所达到目的之用。每个箱子的负荷（可为长度、重量等等.）为 **C**，今有 **n** 个负荷为 **wj，0 < wj < C，j = 1，2，…，n** 的物品 **J1，J2，…，Jn** 需要装入箱内：装箱问题就是指寻找一种方法，使得能以最小数量的箱子数将**J1，J2，…，Jn** 全部装入箱内

Bin Packing

- 有n个周期性的实时任务待分配。当每个处理器使用EDF算法调度时，需要的“箱包”的数目为$\left\lceil \sum_{i=1}^{n} u_i \right\rceil$
- 若按照第一优先随机装箱算法，任务被随机选取并随机分配给处理器，只要处理器的使用不超过1

在这种方法中，处理器个数最多是理想个数的1.7倍。

- 若按照第一优先减少的装箱算法，按任务的处理器使用率的非递增顺序构建一个顺序表（即最高处理器使用率的任务处于第一个位置），从列表中依次选取任务并分配给处理器，只要保证处理器的使用不超过1 即可，任务被分配给第一个满足条件的处理器

此时，处理器个数最多是理想个数的1.22倍。

任务的动态分配

- 前面讨论的三种方法都是任务的静态分配算法
- 但在很多应用中，不同节点上的任务出现是异步的，此时场景需要动态算法来处理任务

- 动态算法假设任意处理器可以处理任意任务
- 很多动态解决方案本身就是分布式的，而且没有假设处理器上会有集权式的调度规则
- 在动态算法中，任务不是被预分配给处理器，而是在任务出现的时候才进行分配

- 在任务加载的节点上可以立即完成分配，可完成的调度使用要优于静态的方法
- 但由于每个节点上的分配单元需要跟踪其他节点上的加载位置，动态方法可能会导致过高的运行时间
  - 在静态分配算法中，在系统的初始化阶段任务就固定分配给处理器，运行过程中就不会产生过多的计算开销
  - 若一些任务需要指定处理器来运行，动态分配会更低效

集中处理（*Focussed Addressing*）和竞争（*Bidding*）方法

- 每个处理器维护两张表，状态表和系统负载表
  - *状态表*包含了该处理器要处理的任务的信息，包括任务的执行时间和周期
  - *系统负载表*包含了系统中其他处理器的最新负载信息，由此可以确定其他处理器的可用计算能力
- 时间轴被分成若干窗口（表示固定的时间段），在每个窗口的结尾，处理器向其他处理器发送自己下一个窗口中的空余段，即没有任务需要被执行的时间段，收到该消息的处理器更新自己的系统负载表

- 当任务在一个节点上出现时，该节点首先检查任务是否可以在本地处理，若可以，节点更新自己的状态表；否则它会查找可以转交任务的其他节点
- 当查找合适的节点时，处理器查询自己的系统负载表以确定系统中最空闲的、可以处理该任务的处理器，并向其发出**RFBs（Request for bids）**信息
- 在查找处理器的过程中，一个过载的处理器会检查它的剩余信息（**surplus information**），选择一个处理器（称为集中（**focussed**）处理器）

- 但系统负载表中的信息可能是过期的，因此，在过载处理器将任务转交给集中处理器的时候，存在集中处理器已经变为负载状态的可能
- 所以一个处理器不能简单地根据自己系统负载表中的信息转交任务，一个处理器只有在它可以确定任务可以被及时完成的情况下才发出RFB请求
  - 这包括获得其它处理器响应和移交任务的时间
  - 选择处理器的标准可以依据处理器的相似性、准确的负载信息等因素

- 缺陷：在集中处理和竞争的方法中，为了在各个处理器上维护系统负载表，可能会导致过高的通信开销
- 时间轴上的窗口尺寸是一个重要的参数，它决定了过高的开销是否会发生
  - 如果时间窗口增加，则高开销的发生会减少。但是这会导致各个处理器上的信息过期，进而出现因为时间窗口中状态改变而没有节点应答的情景
  - 如果时间窗口缩短，处理器上的信息会更新快一些，维护信息表的通信开销又会很高。

同伴（Buddy）算法

- 本算法致力于解决集中处理和竞争算法中的高通信开销，差别主要在发现目标处理器上的方法不同。
- 一个处理器总处于下面两种状态之一：未超载（**underloaded**）和超载
  - 如果一个节点的使用小于某个临界值，那么它的状态是未超载，即 $u_i < Th$，P_i 未超载；否则即处于超载状态

- 在同伴算法中，广播并不是在时间窗口的结尾发生的，而是在状态切换的时候才发送广播，
- 广播只针对一个处于同伴集（**buddy set**）的部分处理器发送
- 在设计同伴集的时候有几个标准：
  - 首先，集合不应过大或过小；
  - 在多点网络中，处理器的同伴集通常是其邻接节点

任务调度的容错

- 所谓容错：在出现故障时，功能单元继续执行所要求功能的能力
- 使用任务调度的技巧可以在实时系统中达到有效的容错能力，而且对硬件资源的要求不高
- 在任务的最初拷贝上，通过调度另外的副拷贝就可以达到容错 （冗余技术）
- 副拷贝可以和原始拷贝不同，是一个可以在更短时间内被执行完的短小版本
- 不同任务的副拷贝可以在同样的节点中过载，在初始拷贝成功执行完后，相应的备份可以被丢弃

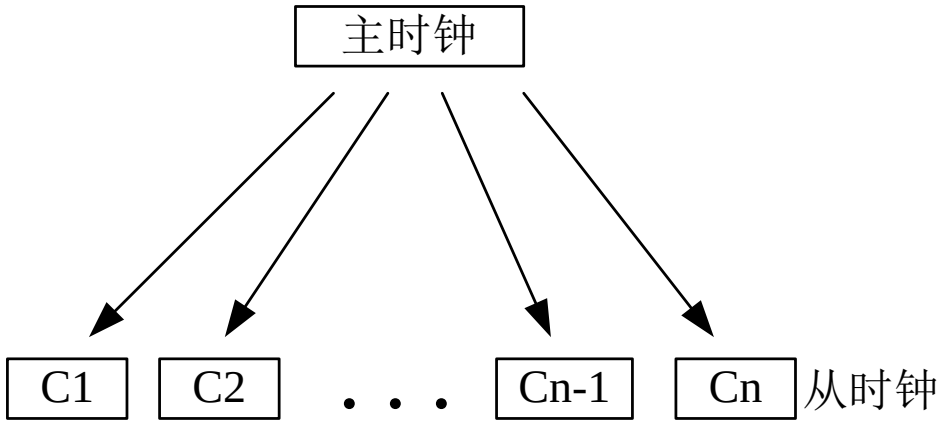
分布式实时系统中的时钟

- 电脑中的时钟除了传统的应用之外，还有两个主要目的
  - 确定超时：
    - 例如根据截止时间确定一个任务的是否失败
    - 对于分布式系统中的发送端和接收端的通信，超时可用于指示可能的传输错误或延迟，或者不存在接收端
  - 加盖时间戳
    - 如任务间的消息通信：消息发送方会在消息中包含当前时间
    - 时间戳不仅赋给接收端关于消息的时间信息，也可以用来确定顺序。良好的实时时钟服务是时间戳的基础

- 一个典型的分布式系统在每个终端上都有时钟
- 由于几乎不可能让两个时钟运行在完全一致的速度上，系统中的时钟可能会不同，这种时钟上的异步称为时钟侧滑（clock slew），其决定了时钟的误差
- 时钟间异步和误差将使得分布式系统中的时间戳和超时变得没有意义
- 为了在多节点上实现有效的超时和时间戳，时钟需要被同步

- 时钟同步的目标是使网络中的所有时钟在时间值上得到统一
- 一个系统中的时钟根据系统中的某个时钟同步时，被称为*内时钟同步*
  - 集权式时钟同步
  - 分布式时钟同步。
- 使用外部时钟同步一些时钟时，被称为*外同步*

集权式时钟同步

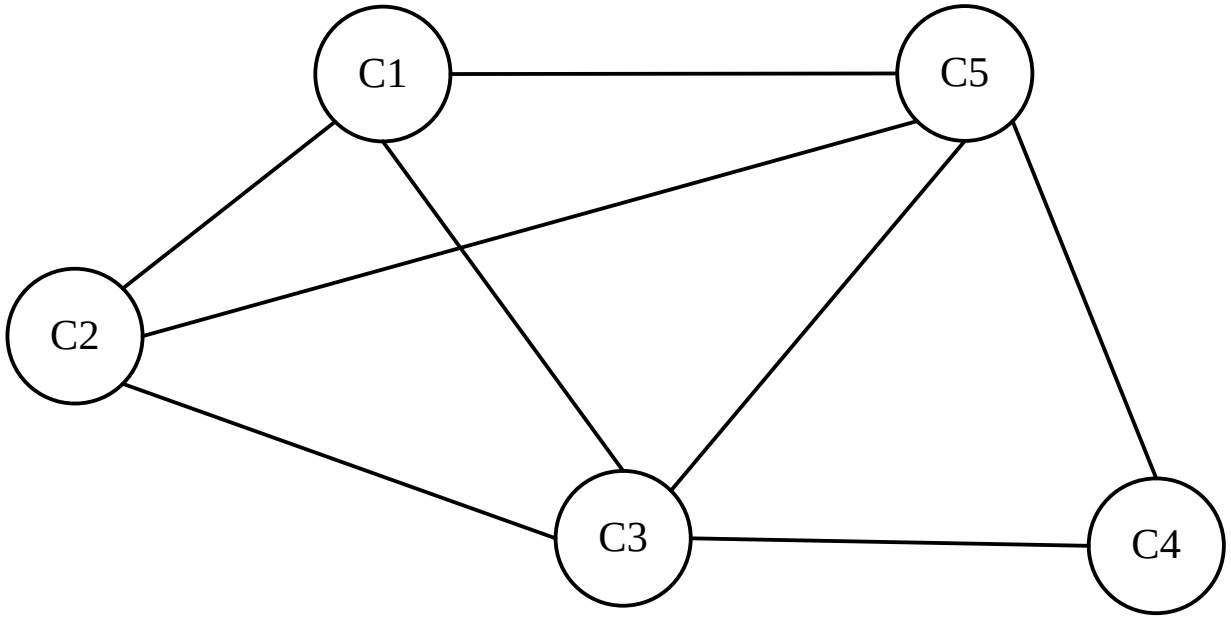


- 一个时钟被设计为主时钟（时钟服务器），其余时钟称为从时钟并与主时钟保持同步
- 时钟服务器每隔ΔT时间向其它时钟发出时间信息以达到同步，从时钟接收到时间信息后设置自己的时间
- ΔT 参数需要仔细的选择：过小广播通信开销过大；过大同步效果差

- 假设两个时钟间的最大差异要被限制在ρ内。
- 通常把可以确定两个任意时钟之间的最大差异作为一个时钟的特定参数。因为参数ρ表示单位时间的偏差，所以它是小于单位时间的
- 假设每ΔT个间隔同步一次时钟，则任意和主时钟之间的时间偏移将被乘以ρΔT。由此可知，两个时钟之间的最大偏移被限制在2ρΔT。

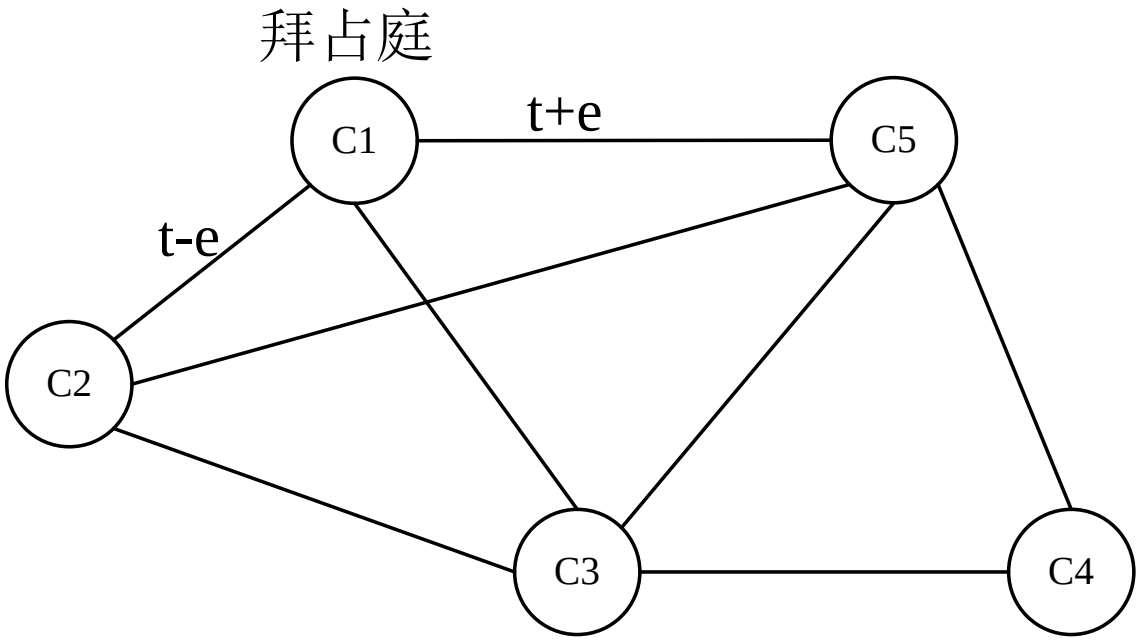
分布式时钟同步

- 集权式时钟同步系统的主要问题是任何发生在主时钟上的错误都会导致整个同步的崩溃
- 分布式时钟同步克服了这个问题，它没有主时钟，而是时钟之间周期性地交换时间信息；每个时钟计算同步时间并相应的重设时间

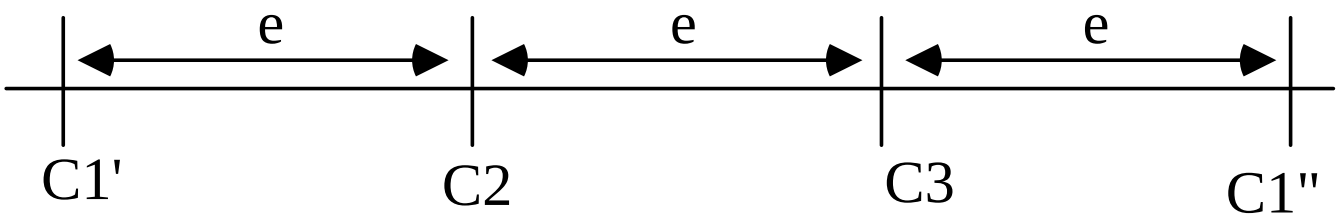


- 但在系统运行中，一些时钟可能会出错，出错的时钟带来的时间偏差大于预先设定的容错，甚至出错时钟会停止计时
- 通过丢弃那些明显大于设定界限的时间信息，可以很容易的识别坏掉的时钟，并在同步过程中处理

- 拜占庭（Byzantine）时钟提出了一个更隐蔽的问题。拜占庭时钟是双面的，它可以对不同时钟发送不同值。
- 如下图所示，C1是一个拜占庭时钟，发送t+e给时钟C5，同时发送t-e给时钟C2



- 存在拜占庭式时钟时两个时钟的偏差



- 已经证明如果三分之一的时钟是错误的或者拜占庭的，仍然可以使正确的时钟很好地同步
- 同步设计：假设系统中有n个时钟，每个时钟周期性的在时间间隔的末端广播自己的时间信息。系统中的时钟要求同步在 ε 时间内，如果一个时钟收到的时间信息和自己的时间信息之差大于 ε ，那么就可以确定发送端是错误的，并且丢弃收到的消息。每个时钟在广播之后求出自己的收到的有效信息的平均值，并重置时间

- 分布式时钟同步过程伪码表示：

```
good-clocks=n;
for(j=1 ; j<n : j++){
        if(|ci-cj|) good-clocks--;        //错误时钟
        else total-time = total-time+cj ;
        ci = total-time/good-clocks;
         //把自己的时间设为计算得到的时间。
}
```

- 定理：一个分布式系统有n个时钟，一个拜占庭时钟可以使系统中两个正常工作时钟的计算平均时间之间的最大偏差（3ε）/n，ε表示两个时钟之间允许的最大偏差值
- 即，系统每个时钟独立地执行前述相同的一系列同步步骤。如果一个分布式系统中的所有n个时钟执行了以上步骤，而且n个时钟里最多检查出m个时钟坏掉，并且n>3*m，那么正常的时钟会在（3ε）m/n的偏差内达到重新同步

Homework 3

Q1: 实时计算环境下，有哪些时钟同步策略与算法？各有什么优缺点？
Q2：试针对其中2种，说明应用场景。

本讲小结

- 首先讨论了在分布式和多处理器系统中实时任务是如何分配的。分布式和多处理器系统中的任务调度是一个比单处理器上复杂的多的问题。
- 在多处理器和分布式系统上的任务调度问题由两个子问题组成：对各个处理器进行任务分配和在处理器上的任务调度。任务分配问题是一个NP难问题。
- 在一个分布式实时系统中，让所有时钟同步在一个可以接受的范围内是很重要的。围绕集权式同步和分布式同步的设计。
- 集权式时钟同步容易受到单点错误的影响。另一方面，在分布式系统中，当不超过25%的时钟坏掉或者处于拜占庭状态时，那些正常工作的时钟可以实现（再）同步