

第一部分、嵌入式系统概念

Concepts of Embedded Systems

1、嵌入式系统概念(Concepts of Embedded Systems)

1-1、嵌入式系统的特性(Characteristics of Embedded Systems)

1-1-1、嵌入式系统与通用系统(Embedded Systems vs. General Purpose Systems)

1、什么是嵌入式系统

定义1：嵌入在较大的设备中的信息处理系统

定义2：集成了物理处理器的软件

定义3：应用在特定领域，为特定环境下的应用进行优化的信息处理系统

2、嵌入式系统主要做什么

感知现实世界（输入信号）

消息处理

实时反馈（输出信号）

3、嵌入式系统的特性

可靠性： $R(t)$ ：系统在t时刻可以正常工作的概率

可维护性： $M(d)$ ：系统在t时刻可以从错误中恢复工作的概率

可利用性： $A(t)$ ：系统在t时刻可以工作的概率

安全性：系统故障时无伤害

私密性：授权与保密

高效性：能源高效，程序高效，运行高效，轻便，低耗

能源高效：大部分嵌入式系统靠电池供电，但电池跟不上电子技术发展的步伐，因此需要低功耗

程序高效：嵌入式系统内存和硬盘空间限制大

运行高效：尽量快地解决问题以缩短硬件通电时间

注1：即使再高级的系统也总有故障的时候

注2：这些特性在一开始设计时就要进行考量，而不能事后补救

同时大部分的嵌入式系统必须满足实时性要求

系统需要对现实环境做出及时的反映

对于嵌入式系统来说反应太慢使得正确的操作不适当

设计时需要对实时性进行验证

嵌入式系统通过传感器和执行器与现实世界进行交互

嵌入式系统一般为混合系统：既要处理模拟信息又要处理数字信息

不少嵌入式系统为响应型系统：能感知现实环境并根据现实情况做出及时的反映

重要的是：输入和环境（就好比种花，有种子还不够还要有好的环境）

所有嵌入式系统都是专用系统

专门为一种特定应用而生：因为这样可以通过特定优化以减少资源并增大鲁棒性

专门的人机交互设计：比如说洗衣机有人机交互面板，而不是鼠标键盘...

最后下一个定义：具备上述性质的系统称为嵌入式系统

4、嵌入式系统与通用系统的比较

嵌入式系统	通用系统
专门为某种应用而生	通用
终端用户不可编程	终端用户可编程
限制运行环境的性能评估	越快越好
关键在于资源资金的消耗，能耗，可预见性	关键在于资源资金的消耗，平均运行速度

1-1-2、嵌入式系统与网络物理系统(Embedded Systems vs. Cyber Physical systems)

1、什么是网络物理系统

网络物理系统并不是一个单独的设备，而是集物理信息感知，信息处理与执行的一整套网络

网络物理系统既包含了集中式的计算又包含了与物理环境的交互

 网络物理系统是嵌入式系统和物理环境的融合

网络物理系统是计算系统和物理器件的结合

 必须满足鲁棒性和响应性的要求

 可能是协同处理的，分布式的，集中式的

2、嵌入式系统与网络物理系统

网络物理系统比嵌入式系统在安全性上要求更高

嵌入式系统是网络物理系统的子系统

网络物理系统除了要进行计算（信息处理）外还要进行控制和交互

1-2、嵌入式系统的趋势

1、传统的嵌入式系统

传统的嵌入式系统的别称为嵌入式控制器

 简单的单核处理器，专门的时序电路设计，一般应用于控制领域

2、嵌入式系统的趋势

趋势1：多处理器系统

 单处理器系统只能满足性能要求低得场合

 对于性能和安全性要求高的场合（车辆，飞机等）必须采用多处理器

 对于能源效率、运行效率要求高的场合（移动设备）需要采用异构处理器设计

趋势2：集成度的提升

 摩尔定律：每两年单位面积集成的晶体管数目翻一番

 微处理器，微控制器，片上系统，片上多处理器系统(MPSoC)

趋势3：软件升级

 单纯依靠硬件实现嵌入式系统可扩展性不强而且制造维护成本高

 越来越多的功能通过软件进行实现

1-3、嵌入式系统设计

1、嵌入式系统设计是一个整体，不等于软件设计加硬件设计

嵌入式系统需要与现实世界进行交互，因此必须满足一些约束

 相互作用约束：截止期限，吞吐量，抖动

 处理约束：可利用资源量，功耗，故障率

 既有功能性要求，又有非功能性要求（性能，功耗...）

由于需要与现实世界进行交互并且需要满足非功能性约束，单纯靠软件建模和硬件设计无法很好地解决问题，因此必须进行软硬件协同设计

第二部分、嵌入式设计方法论

Design Methodologies

2、嵌入式设计方法论(Design Methodologies)

2-1、回顾：嵌入式系统设计趋势(Design Trend Recap)

1、嵌入式系统设计是一个整体

嵌入式系统需要与现实世界进行交互，因此必须满足一些约束

 相互作用约束：截止期限，吞吐量，抖动

 处理约束：可利用资源量，功耗，故障率

 既有功能性要求，又有非功能性要求（性能，功耗...）

2、嵌入式系统设计趋势

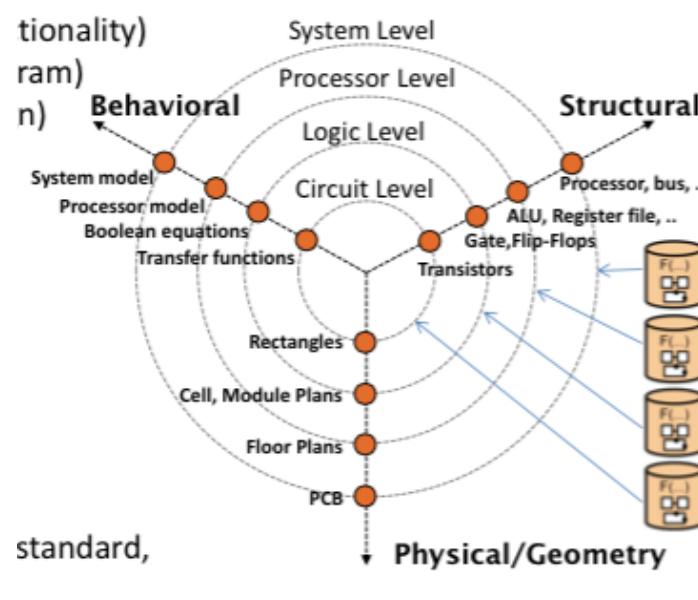
- 趋势1：多处理器系统
- 趋势2：集成度的提升
- 趋势3：软件升级

2-2、Gajski的Y-图(Gajski's Y-Chart)

1、产生背景

由于嵌入式系统设计的复杂性，不论单纯采用硬件中RTL的方法，还是采用用任何一种软件设计语言都不能很好地对嵌入式系统设计进行建模

2、Gajski的Y-图



三个设计视角：

- 视角1：行为：说明设计模型（规范、功能）：传递函数 -> 布尔表达 -> 处理器模型 -> 系统模型
- 视角2：结构：说明设计架构（网表、框图）：晶体管 -> 门与触发器 -> ALU与寄存器文件 -> 处理器与总线
- 视角3：物理：说明设计基础（板图、布局）：电路元件 -> 标准单元 -> 平面板图 -> PCB

四个抽象层级：

- 层级1：电路层
- 层级2：逻辑层
- 层级3：处理器（RTL）层
- 层级4：系统层

四个组成库（分别对应四个抽象层级）：

- 晶体管
- 逻辑（标准单元）
- RTL（ALU, 寄存器文件）
- 处理器/通信

3、综合

定义：在任意一个抽象层级中将行为设计视角转换为结构设计视角
分类：电路层综合、逻辑层综合、处理器层综合、系统层综合

4、处理器层综合

输入：行为设计视角中的处理器层：处理器模型
带数据通路的有限状态机、指令流程图等

输出：结构设计视角中的处理器层：处理器结构模型

数据通路模块：

存储模块（寄存器、寄存器文件、内存等）

功能模块（ALU、移位器、乘法器等）

连接模块（总线、选择器等）

控制模块：

寄存器（程序计数器(PC)、控制字、指令寄存器等）

存放控制的存储器

处理器结构：

流水线、用于处理hazard的转发、多时钟周期等

综合过程：输入 -> 选择需要的组件和结构中采用的连接方式 -> 明确周期，分析处理器模型中的调度 -> 绑定变量 -> 绑定操作 -> 绑定传输方式 -> 开始综合 -> 优化模型 -> 输出

综合要素：在不同的模型，不同的库，不同的特性，不同的结构，不同的工具，不同的度量方式，不同的质量下的综合结果会有所不同

5、系统层综合

输入：行为设计视角中的系统层：系统行为模型

MoC：计算模型

输出：结构设计视角中的系统层：系统结构模型

计算元件集：处理器、IP核、用户自定义的硬件元件、内存

通信元件集：总线、桥、仲裁器、片上网络系统(NoC)

综合过程：输入 -> 行为描述与分析 -> 分配元件 -> 元件连接 -> 处理器与信道的绑定 -> 定义软硬件获取指令的方式 -> 设计调度方式 -> 优化模型

综合要素：在不同的元对象编译器，不同的库，不同的特性，不同的平台，不同的工具，不同的度量方式，不同的质量下的综合结果会有所不同

6、设计方法论（仅作了解）

设计方法论定义了用来设计一个产品的模型、组成、工具设计，同时还融合了技术、复杂度等因素的分析

设计方法论为具体的设计提供了标准的参照

类型：

方法1、自底而上设计方法

从底层的行为视角开始，每个底层生成底层库交予上层

电路层：生成逻辑层的标准单元

逻辑层：生成处理器层的RTL元件

处理器层：生成系统层的处理、通信元件

系统层：生成特定应用的嵌入式系统平台

每层的设计遵循：行为功能定义 -> 结构网表定义 -> 硬件板图定义

优势：

1、每个抽象层级生成自己的库，不同抽象层级的库相分离

2、每一层可以做到较为精确的度量（因为每一层都有自己的硬件板图）

3、允许全局分布式开发（每层做自己的，最后再合起来）

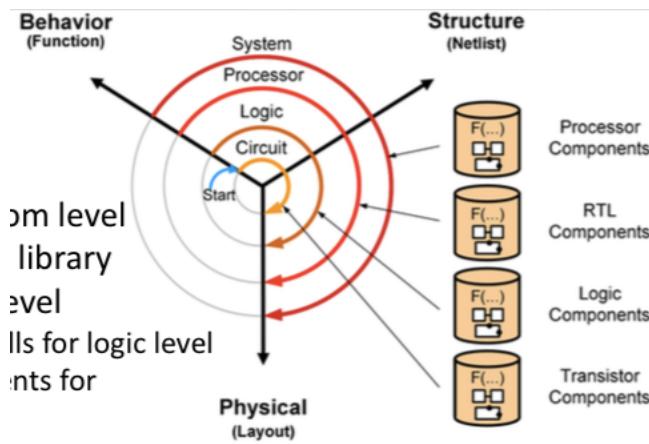
4、易于管理（每层做自己的，可以分配）

劣势：

1、很难进行优化（因为每层单独做自己的，而且不知道上层需要什么）

2、定制的库没办法加入到设计中（因为层与层之间相分离）

3、每层一个板图，缺乏整体性



方法2、自顶而下设计方法

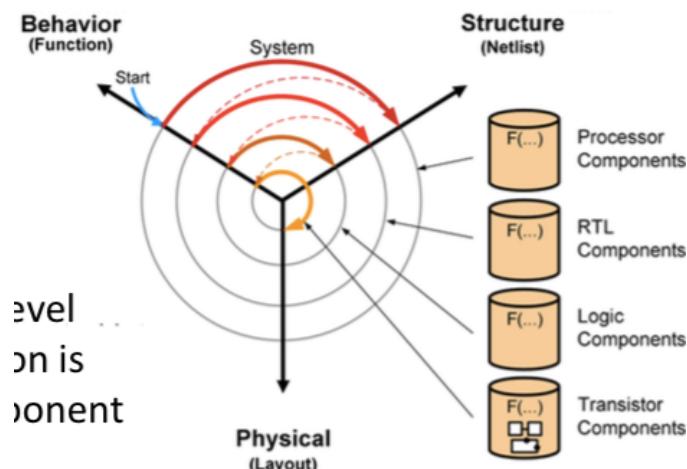
从顶层的行为视角开始，除最底层外进行行为功能定义 -> 结构网表定义，生成结构网表
上层将结构网表交给下层进行行为功能定义

底层进行行为功能定义 -> 结构网表定义 -> 硬件板图定义（只有底层才进行硬件板图定义）
优势：

- 1、可以进行用户自定义设计，上层用户自定义的要求可以传到下面各层
- 2、最后只需要一个小的晶体管库（只有底层需要进行具体设计，上层只管交给底层）
- 3、最后只生成一个硬件板图

劣势：

- 1、很难对顶层进行度量（因为顶层不进行具体的设计，只有底层才进行）
- 2、顶层的任务很难具体化
- 3、热点去除困难（顶层设计不好的整个底层都要重做）
- 4、导致底层和顶层设计的来回迭代（因为顶层改变影响底层，顶层又很难进行具体定义）



方法3、中间相遇设计方法

中间相遇设计方法融合了自底而上设计方法和自顶而下设计方法

先设定某一层，在此之上采用自顶而下设计方法，在此之下采用自底而上设计方法
最终在这一层相遇

一种情况：处理器层及其下层采用自底而上设计方法，系统层采用自顶而下设计方法
最终在处理器层相遇

此时系统层将MoC综合成处理器结构最后再综合成RTL，通过RTL进行实现
会产生三个板图

另一种情况：逻辑层及其下层采用自底而上设计方法，处理器层、系统层采用自顶而下设计方法
最终在逻辑层相遇

此时系统层将MoC综合成处理器结构，然后再综合成RTL结构，最后再综合成标准单元，

通过标准单元进行实现

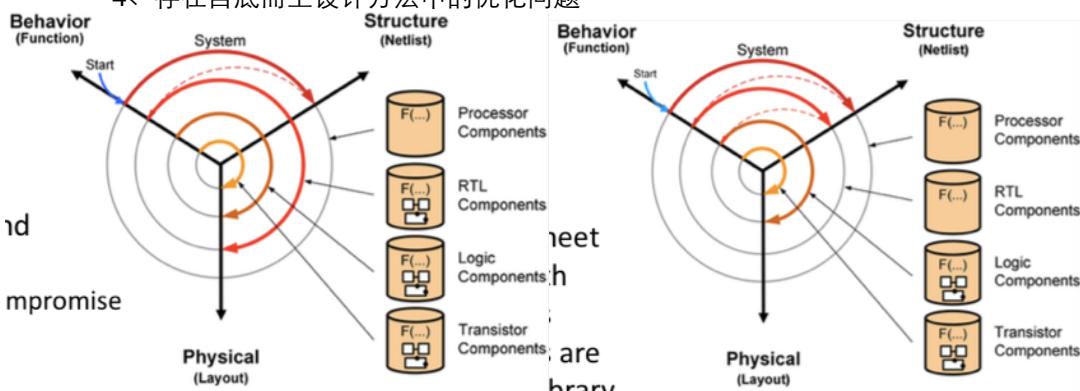
会产生两个板图

优势：

- 1、比自顶而下设计方法综合时间短（自顶而下一次要做四层综合）
- 2、比自底而上设计方法生成的板图少
- 3、比自底而上设计方法使用的库少
- 4、比自顶而下设计方法的度量精确

劣势：

- 1、比自顶而下设计方法使用的库多
- 2、比自顶而下设计方法生成的板图多
- 3、不如自底而上设计方法的度量精确
- 4、存在自底而上设计方法中的优化问题



方法4、平台设计方法

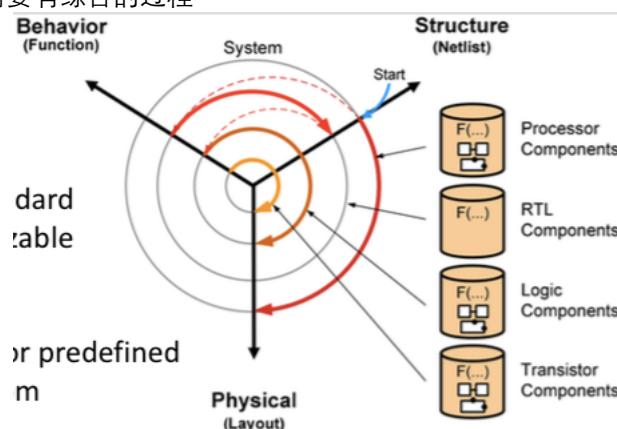
平台设计方法从系统层结构视角开始，一方面生成系统层板图(PCB)，另一方面交给下层进行行为功能定义然后再进行处理器层的综合形成处理器层结构，再交给下层。下层再使用自底而上设计方法。这种设计方法和上面提到的中间相遇设计方法的第二种情况类似，不同之处在于一开始的出发点在于系统层架构（因此这个方法称为平台设计方法），并通过这个架构另外生成系统层板图（这也就是平台）

优势：

- 1、兼容底层标准单元和系统层平台
- 2、留下了处理器层的实现，从而可以插入一个既能兼容平台又能兼容底层库的处理器
- 3、留下了可供用户进行优化的接口（用户在优化底层库时不影响系统层平台）

劣势：

- 1、还是需要对平台进行人为的定义（因为设计方法论是作为标准的，人参与偶然性会大）
- 2、还是需要有综合的过程



方法5、系统设计方法

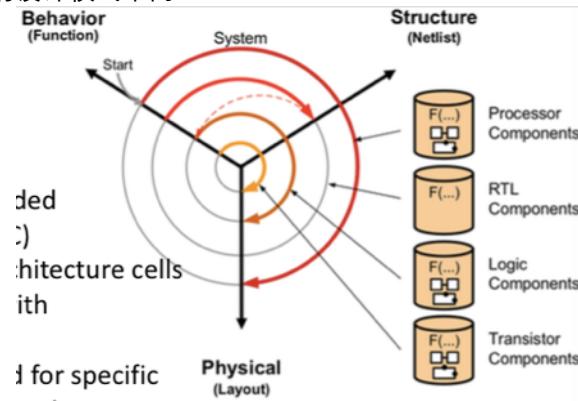
系统设计方法从系统层行为视角开始，一方面生成系统层板图(PCI)，另一方面交给下层进行行为功能定义然后再进行处理器层的综合形成处理器层结构，再交给下层。下层再使用自底而上设计方法。这种设计方法和上面提到的平台设计方法类似，不同之处在于一开始的出发点在于系统层行为视角（因为系统是基于功能的，和平台是基于架构的不同，因此称为系统设计）

优势：

- 1、只留下了处理器层的实现，模型得到简化，适合具体应用设计，不需理会底层
- 2、底层只需要一个简单的可重定向编译器就可以实现所有的架构单元（因为自底向上）

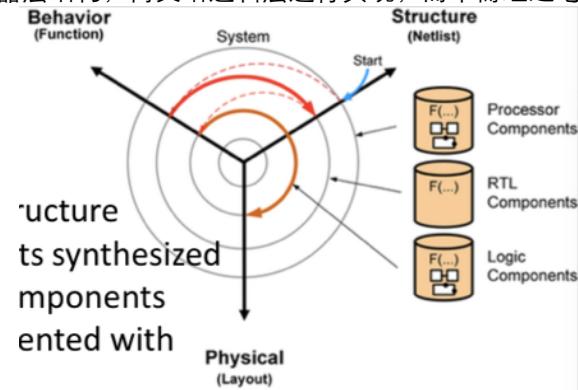
劣势：

- 1、还是需要进行人为的定义
- 2、与传统的设计模式不同



方法6：FPGA设计方法

FPGA设计是一种特殊的平台设计，它从系统层结构视角开始交给下层进行行为功能定义然后再进行处理器层的综合形成处理器层结构，再交给逻辑层进行实现，而不需经过电路层。



7、设计流程

1960s~1980s的设计流程：捕捉与仿真

全手动设计，通过仿真进行模型验证

1980s~1990s的设计流程：描述与综合

系统功能越发复杂，安全性要求越来越高，直接上的方法已不能满足要求，需要划分功能模块

先对系统进行功能性描述，再进行综合，最后仿真

2000s~的设计流程：说明、尝试与完善

单层设计也不能满足要求，因此要进行多层协同设计

首先定义模型

然后探索不同的设计模式

最后再根据具体情况做优化

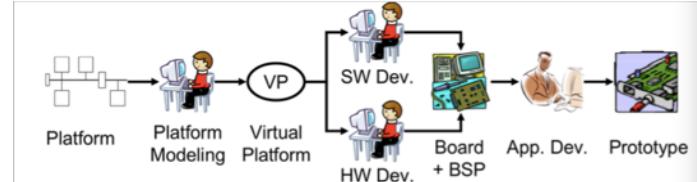
8、系统设计模式

传统系统设计模式：严格遵循硬件 -> 软件 -> APP的设计模式



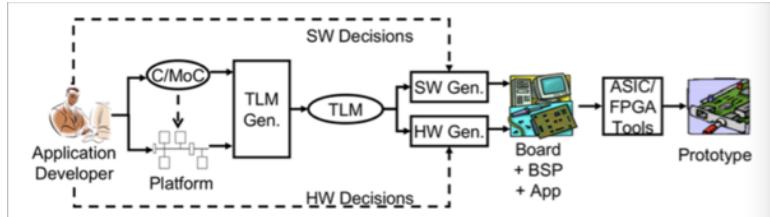
平台通过架构师进行定义（架构师打好框架），硬件设计在这个框架下进行：设计慢，易出错
软件设计基于硬件设计得到的板进行：只能发现板上软件的错误，而很难发现板这个硬件的错误

基于虚拟平台的系统设计模式



在虚拟平台上可以同时进行软件硬件设计，从而可以相互补充
虚拟平台上提供的一些工具可以提升软硬件开发的速度

基于模型的系统设计模式



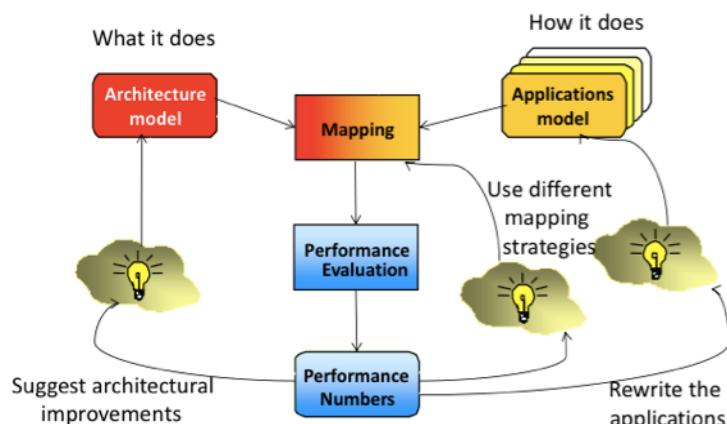
在这种设计模式中只需要应用程序开发者定义数学模型
平台，底层软件，底层硬件都可以自动综合出来（就像编译器一样将函数转换成可执行代码）
借助事务级模型(TLM)对设计进行验证和评估，并生成软硬件
例如：FPGA，只用写好上层的软件和硬核，系统就会自动进行分析综合

设计模式三要素：

- 建模：说明这个系统是干什么的
- 设计：说明这个系统是怎么进行工作的（优化也包含在里面）
- 分析：说明这个系统为什么会这样做

2-3、Kienhuis的Y-图(Kienhuis's Y-Chart)

1、Kienhuis的Y图



2、与Gajski的Y-图的比较

Gajski的Y-图更注重综合过程（注重行为、结构、物理三个视角的相互关系）

Kienhuis的Y-图更注重质量评价过程（中间有很大一块是性能的分析评估）

第三部分、嵌入式功能定义与建模

Specification and Modeling

3、计算模型(Model of Computation (MoC))

3-1、绪论(Introduction)

1、为什么需要考虑功能定义？

设计嵌入式系统的第一步就是定义这个系统是做什么的功能定义就是对系统行为的准确，清晰，无歧义的描述

但要做到这一点不容易

 嵌入式系统的复杂度不断提升

 很难在一开始就得到准确的描述

如果一开始功能定义出了差错，后续设计也会出现问题

2、基于模型的功能定义

模型是抽象了某项特定任务的实体的特征和属性而生成的最简化的实体
现在普遍将各层的设计抽象成模型进行考量

 抽象成模型降低了功能定义的难度

 模型可以帮助我们发现并改正功能定义中的瑕疵

3、嵌入式系统中建模与功能定义的要求

要求1：模块化：系统是对象的组合

要求2：层次表达：在多对象下用户很难理解整个系统（只见树木，不见森林）

 因此必须进行层次化的表达

 行为层次化：例如：语句、函数、程序

 结构层次化：例如：晶体管、门、处理器、印刷电路板

要求3：并发性、同步性、交互性

要求4：能正确表示时序行为和要求

 1、能正确测量程序执行的时间

 2、能正确定义延迟的处理方法

 3、能正确定义超时的行为

 4、能明确指定截止时间

要求5：表示面向状态的行为（在响应性系统中需要这么做）

要求6：表示面向数据流的行为（说明不同组件间数据是如何交互的，是怎么处理这些数据的）

要求7：为高效实现提供可能（不能是空谈的模型，必须要是可以实现的）

3-2、计算模型(Model of Computation (MoC))

1、背景

组件和执行模型定义了计算的组件和操作

通信模型定义了不同组件间的数据交互

但是还没有一个模型能同时满足上述七个要求

因此出现了计算模型

2、传统的计算模型

种类1：冯诺依曼模型

 组成：指令集、内存、程序计数器...

局限性：

只能线性执行指令，不能进行并发执行，不满足要求3

没有考虑到时序行为，不满足要求4

不能描述时序：指令不能被延迟，也不能规定指令在某个特定时刻执行

截止时间不能被定义

超时也不能被定义

冯诺依曼模型并不适合于嵌入式系统

种类2：基于线程的并发模型

基于线程的并发会出现对于共享变量的访问问题

访问共享变量会导致竞态条件

为避免竞态条件需要定义互斥变量（临界区）

临界区又会导致死锁产生

进行死锁避免会产生极高的代价

基于线程的并发模型也不适合于嵌入式系统

3、嵌入式系统需要一个什么样的计算模型

不能基于线程，又不能使用冯诺依曼模型

对于以控制为主导的或是响应型的系统而言：采用基于状态的模型，定义状态输入输出关系

对于以数据为主导的系统而言：采用数据流系统，定义输入输出数据流关系

4、数据流模型(Data-Flow Models)

4-1、Kahn进程网络(Kahn Process Network)

1、数据流语言

数据流语言模型：进程间通过FIFO的缓冲区进行数据交互

数据流语言不同于命令式语言

命令式语言以程序计数器为中心

数据流语言以数据的移动（交互）为中心

由系统负责进行调度，而不是程序员进行调度（不加人工干预）

基本特征

所有进程同步执行

进程可以通过命令式语言进行描述

进程间只能通过缓冲区进行通信

读写等价：读什么写什么

适宜基于数据流的应用

从根本上进行并发：很容易从图中看出实际的并行的硬件是什么样的

尤其适合基于框图的功能定义（比如控制系统、信号处理系统）

适合多媒体应用（多媒体基于数据流）

表达方式

通过C、C++、Java等进行进程功能描述

通过xml等进行网络描述

2、Kahn进程网络

特征：

读：可能发生阻塞（阻塞读空信道的操作直到原空信道中出现数据）

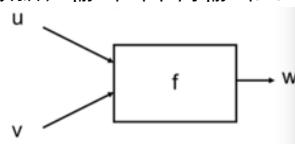
写：不阻塞

信道采用FIFO的调度，容量无限

是确定性的：输出不受输入到来顺序的影响

模块：

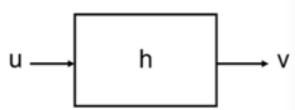
f模块：两个输入（u、v），一个输出（w）
进程交替地从u、v中拿数据，输出出来再输出至w



g模块：一个输入（u），两个输出（v、w）
进程从u中拿数据，再交替输出到v、w



h模块：一个输入（u），一个输出（v）
进程从u中拿数据，再输出到v



3、确定性分析

随机系统：在给定各信道输入序列的情况下，在输入序列到来顺序不同的情况下可能出现不同输出序列的系统称为随机系统

确定性系统：在给定各信道输入序列的情况下，即使输入序列的到来顺序不同，输出序列也随之确定的系统称为确定性系统

重要性：在确定性系统中系统的行为和时序无关，从而可以将两者分离开来考虑

满足单调取值的Kahn进程网络是确定性系统

g、h模块只有一个输入且遵循FIFO，因此一定是确定性输出的

f模块中有两个输入，但由于是交替读取，而且当读取到一个空信道时会阻塞读取操作。因此最后读取到的结果一定是从信道1读第一个，从信道2读第一个，从信道1读第二个，从信道2读第二个...这种读取结果是单调的，因此f模块也是一个确定性模块

因此Kahn进程网络是确定性系统

以下行为将会使系统变得不确定

- 1、允许进程从任一非空信道中读取数据
- 2、允许多个进程从同个信道中读取数据或往同个信道中写入数据
- 3、允许进程间共享变量

4、Kahn进程网络的调度问题

上面我们假设的是缓冲区大小不限，但这并不现实，于是出现了下面这个调度问题

f模块中从两个信道读取数据，一个信道以较快的速度地往里进数据，另一个信道以较慢的速度往里面进数据。但由于保证确定性，因此那个以较快的速度地往里进数据的信道就会堆积大量数据最终导致前面的模块对信道的写入操作被阻塞

同样地，如果g、h模块输出数据过快，但后续模块不能及时将数据取出同样会导致g、h输出阻塞

Tom Park算法

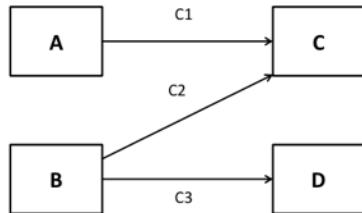
解决在有界存储空间中的Kahn进程网络的调度问题

方法：

- 1、设定一个缓冲区大小数值，阻塞往一个剩余空间不足的缓冲区的写入
- 2、正常进行进程的操作直至出现由于阻塞写入操作而造成的死锁
- 3、一旦出现死锁就说明需要扩大缓冲区的容量

例子：假设下图中缓冲区C1、C2、C3的大小均为1，进程的执行顺序为A、B、C、D

注：这里B是一个复制器，一次会往两个信道同时输出



最后的结果为：

	A	B	C	D	A	C	A
C1	1	1	0	0	1	0	...
C2	0	1	1	1	1	1	...
C3	0	1	1	0	0	0	...

解释：首先运行A，发现A要写入的信道C1有剩余空间，因此往C1中写入；然后轮到B，发现B要写入的信道C2、C3都有剩余空间，因此往两边都写入；然后轮到C，发现C1有值，因此把C1的值拿走；然后是D，D发现C3有值，因此把C3的值拿走；又回头轮到A，发现A要写入的信道C1有剩余空间，因此往C1中写入；然后B发现C2有值，那就阻塞操作直接跳到C；C发现C1有值，因此把C1的值拿走；到D的时候发现C3没有值，阻塞操作；最后的结果是AC两个进程在那里循环

在有界缓冲区下的KPN表达

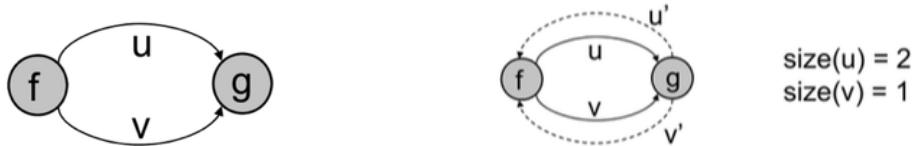
在原信道的情况下反向加入虚拟信道，虚拟信道的大小和原信道的大小相同

对于原信道的读操作会导致对虚拟信道的写操作

对于原信道的写操作会导致对虚拟信道的读操作

原信道和虚拟信道的数据总量保持一定

在有界缓冲区下可能出现死锁



5、Kahn进程网络的评价

优势：

- 1、将调度和进程的具体内部功能相分离
- 2、基于数据流
- 3、体现了并行性
- 4、很容易将图映射到实际物理器件上

劣势：

- 1、很难安排每个进程的处理速度（因此缓冲区里的数据不好估计）
- 2、调度困难，而且必须要在运行时调度（运行前无法估计调度情况）

4-2、同步数据流(Synchronous Dataflow)

1、什么是同步数据流

同步数据流是另一种数据流模型，里面的每个进程在每次启动时都会读写一定数量的数据

每次启动都是原子操作，不可以被中断

2、SDF调度

在系统运行前（编译时）就可以完成调度工作

步骤：

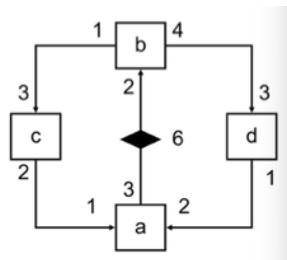
- 1、计算相对运行率：解线性方程组
- 2、确定调度周期：通过模拟得到初始时每个缓冲区里的数据量

结果：得到一种可持续的调度，而且不会出现缓冲区数据溢出的问题

3、相对运行率的计算

步骤1：将进程看成变量，每条边一个方程，出边为加，入边为减

例子：假设给定这样一幅SDF图



由图即可得到方程组：

$$\begin{array}{l} 3a - 2b = 0 \\ 4b - 3d = 0 \\ b - 3c = 0 \\ 2c - a = 0 \\ d - 2a = 0 \end{array} \quad \left[\begin{array}{ccccc} 3 & -2 & 0 & 0 & 0 \\ 0 & 4 & 0 & -3 & 0 \\ 0 & 1 & -3 & 0 & 0 \\ -1 & 0 & 2 & 0 & 0 \\ -2 & 0 & 0 & 1 & 0 \end{array} \right] \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = 0$$

步骤2：判断是否可以调度

当第一步得到的线性方程组的列空间的维数（矩阵的秩）为变量数n - 1时说明存在周期性调度

此时存在唯一的最小正整数解

如果第一步得到的线性方程组的列空间的维数为变量数n说明这个系统不相容

如果第一步得到的线性方程组的列空间的维数小于变量数n - 1说明这个系统不能连通，需要划分

注：如何计算列空间的维数（矩阵的秩）

将方程进行初等行变换，然后看主元的数目。有多少个主元列空间维数就是多少

注：如何得到最小正整数解

将方程进行初等行变换后用一个变量表示其他变量，然后令最小的那个为1，最后再看有没有非整数的，将所有变量扩大K倍变成整数

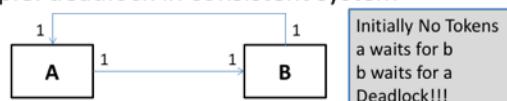
$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 4 \end{bmatrix} \Rightarrow \left[\begin{array}{ccccc} 3 & -2 & 0 & 0 & 0 \\ 0 & 4 & 0 & -3 & 0 \\ 0 & 1 & -3 & 0 & 0 \\ -1 & 0 & 2 & 0 & 0 \\ -2 & 0 & 0 & 1 & 0 \end{array} \right] \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = 0$$

最后得到的最小正整数解就是我们需要的相对运行率

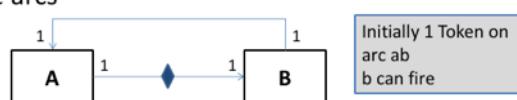
4、确定调度周期

得到最小正整数解并不代表存在调度，如果没有激励输入到系统，系统会发生死锁

Example: deadlock in consistent system

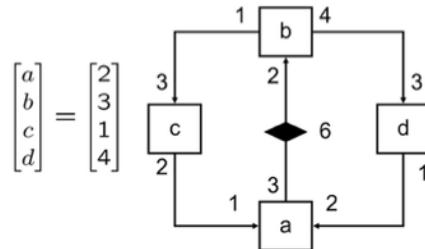


Solution here: add an initial token (delay) on one of the arcs



解释：上面那幅图A需要一个数据来启动，B需要一个数据来启动，但B在等A的数据，A在等B的数据，因此发生了死锁。解决办法很简单，强行插入一个激励，激活整个系统。下图中往B初始化输入一个数据，B得以启动，进而A启动，之后系统就正常运行了

调度周期是不唯一的，要想得到调度周期只能跟着图走一遍，需要具体例子具体分析
比如说下面这个例子：



- 1、从相对运行率得出，在一个周期里面a需要运行2次，b需要运行3次，c需要运行1次，d需要运行4次
- 2、从输入激励的地方出发进行运行：

在这里激励给到进程B有6个数据，B需要有两个数据启动，因此B可以正常运行3次

在B运行了3次以后

BC之间的缓冲区里面就有了3个数据，因此C可以正常运行1次

BD之间的缓冲区里面就有了12个数据，因此D可以正常运行4次

在C运行了1次以后

AC之间的缓冲区里面就有了2个数据

AD之间的缓冲区里面就有了4个数据

而A需要C给的2个数据，D给的4个数据，因此此时可以正常启动2次

- 3、最后得到调度结果BBCDDDDAA

同样地以下调度结果也是合理的，感兴趣的可以自己按照以上的办法模拟一遍

BBBCDDDDAA

BDBDBCADDAA

BBDBBDDCAA

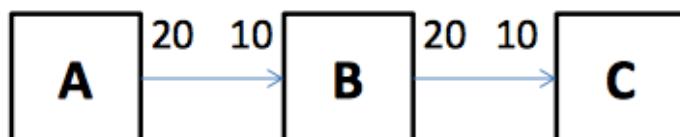
但一般情况下尽量将同个进程放在一起运行（比如说例子中的BBCDDDDAA），这样看上去简洁一些

这样运行的代码会比较简洁，高效

但并不是所有模型都存在这样的调度

与此同时还有一种可以考虑的因素，就是考虑各个信道容量，取那个信道容量最小的调度

例子：假设存在下面这个SDF图



从这个SDF图中我们可以看出，进程A每做一次操作会往AB两个进程间的信道写入20个数据；进程B每做一次操作会从AB两个进程间的信道读走10个数据并同时往BC两个进程间的信道写入20个数据；进程C每做一次操作会从BC两个进程间的信道读走10个数据

现在我们得到下面三个调度方式：

调度1：ABCBCCC

调度2：A(2B)(4C)

调度3：A(2(B(2C)))

首先看调度1：

A: AB信道含有20个数据，BC信道没有数据

B: AB信道含有10个数据，BC信道含有20个数据

C: AB信道含有10个数据，BC信道含有10个数据

B: AB信道没有数据，BC信道含有30个数据

C: AB信道没有数据，BC信道含有20个数据

C: AB信道没有数据，BC信道含有10个数据

C: AB信道没有数据，BC信道没有数据

从这里可以看出AB信道最多有20个数据，BC信道最多有30个数据

因此AB信道容量至少为20，BC信道容量至少为30，因此在调度1的情况下总体信道容量为50

然后看调度2：

调度2相当于：ABBCCCC

A: AB信道含有20个数据，BC信道没有数据

B: AB信道含有10个数据，BC信道含有20个数据

B: AB信道没有数据，BC信道含有40个数据

C: AB信道没有数据，BC信道含有30个数据

C: AB信道没有数据，BC信道含有20个数据

C: AB信道没有数据，BC信道含有10个数据

C: AB信道没有数据，BC信道没有数据

从这里可以看出AB信道最多有20个数据，BC信道最多有40个数据

因此AB信道容量至少为20，BC信道容量至少为40，因此在调度1的情况下总体信道容量为60

最后看调度3：

调度3相当于：ABCCBCC

A: AB信道含有20个数据，BC信道没有数据

B: AB信道含有10个数据，BC信道含有20个数据

C: AB信道含有10个数据，BC信道含有10个数据

C: AB信道含有10个数据，BC信道没有数据

B: AB信道没有数据，BC信道含有20个数据

C: AB信道没有数据，BC信道含有10个数据

C: AB信道没有数据，BC信道没有数据

从这里可以看出AB信道最多有20个数据，BC信道最多有20个数据

因此AB信道容量至少为20，BC信道容量至少为20，因此在调度1的情况下总体信道容量为40

因此调度3是在这种标准下的最佳选择

5、SDF的评价

优势：可以在编译时确定调度

劣势：

- 1、过程限制多（进程启动过程中不能中断，而且定量输入输出）
- 2、对于非功能性属性无法建模（比如能量，可靠性等等）

4-3、例题(Exercises)

1、判断下面两个算法的确定性和公平性

Algorithm 1

```
if L1[X], L2[Y] then
    del(X), del(Y), out[X,Y]
else if L1[X], L2[∅] then
    del(X), out[X]
else if L1[∅], L2[Y] then
    del(Y), out[Y]
else if L1[∅], L2[∅] then
    no operation
end if
```

Algorithm 2

```
if L1[X] = L2[Y] then  
    del(X), del(Y), out[X, Y]  
else if L1[X] < L2[Y] then  
    del(X), out[X]  
else if L1[X] > L2[Y] then  
    del(Y), out[Y]  
end if
```

分析：

确定性：在给定各信道输入序列的情况下，即使输入序列的到到来顺序不同，输出序列也随之确定

公平性：一个信道的输入情况不造成其它信道的饥饿

对于算法1来说如果两个信道都有值就需要取两个信道的值，如果只有一个信道有值就只取那个信道的值，如果两个信道都没有值就不做操作

确定性分析：算法1是不确定的，反例很好举。假设信道X有两个数据X1、X2，信道Y有一个数据Y1，那么如果按X1, X2, Y1的顺序进入，最后的输出顺序就是X1, X2, Y1；如果按照X1, Y1, X2的顺序进入，最后的输出顺序就是X1, Y1, X2。违背确定性；

公平性分析：算法1是公平的，因为只要信道有值，每次循环就会取出信道中的一个值，因此不会产生信道饥饿的现象。

对于算法2来说只有当两个信道都有值的时候才会发生取值。而具体取值是通过值的序列号确定的，每次只输出序列号小的那个值。如果两个信道的值的序列号一样小就都取。

确定性分析：算法2是确定的。因为每次都取序列号小的那个值，也就保证了X_k必定在X_i(i > k)的值之前被取走，也必定在Y_i(i > k)的值之前被取走。从而保证了取值的单调性，从而保证了确定性

公平性分析：算法2是不公平的，因为如果其中一个信道一直没值，另一个信道的值也不能被取出，因此会出现信道饥饿现象

2、根据下列基本进程画出输出序列为n(n+1)/2的KPN图

加法器：两路输入，一路输出，输出为两路输入相加的结果

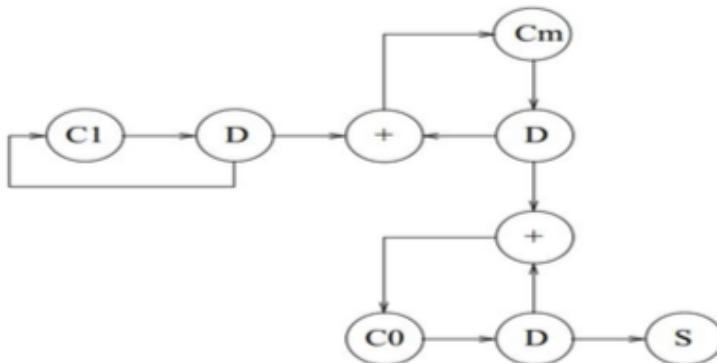
乘法器：两路输入，一路输出，输出为两路输入相乘的结果

复制器：一路输入，两路输出，两路输出与输入完全相同

常量生成器：一路输入，一路输出，初始输出为预设的初始值，在此之后输出值等于输入值

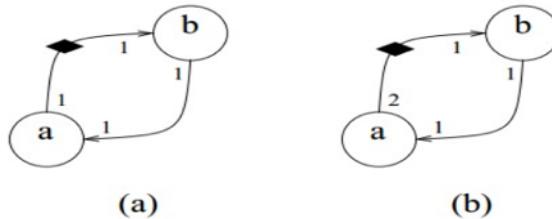
汇聚：将输入抛出以显示最终结果

分析：一开始先生成全1序列，然后通过这个再生成1, 2, 3...的等差数列，最后再生成数列相加后的结果



(其中m等于1)

3、给定下面两幅SDF图



- (1) 请根据SDF图写出对应的计算相对运行率的拓扑矩阵
- (2) 根据第一问得到的矩阵判断这两个系统是否相容
- (3) 如果第二问判断为“是”，请得出相对运行率

分析：

第一问很简单，将进程看成变量，每条边一个方程，出边为加，入边为减，可得

$$\begin{cases} a - b = 0 \\ b - a = 0 \end{cases} \rightarrow \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = 0$$

$$\begin{cases} 2a - b = 0 \\ b - a = 0 \end{cases} \rightarrow \begin{bmatrix} 2 & -1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = 0$$

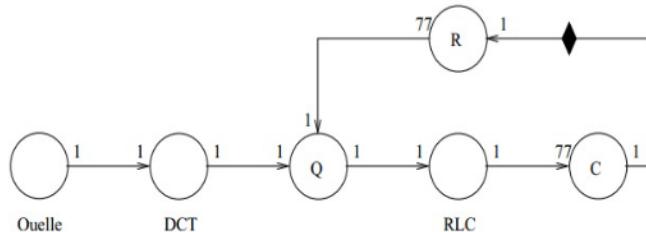
第二问就是看着两个矩阵的列空间维数是多少，如果是变量数n，就说明不相容，否则相容。对上面两个矩阵进行初等行变换后可得（a）中有一个主元，因此列空间维数为1，说明相容；（b）中有两个主元，因此列空间维数为2，说明不相容。

第三问只需要对（a）进行分析，此时我们用一个变量表示另一个变量

$$\begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \rightarrow a = b$$

因此a在一个周期中应该运行一次，b在一个周期中应该运行一次

4、给定下面一幅SDF图



- (1) 请根据SDF图写出对应的计算相对运行率的拓扑矩阵
- (2) 根据第一问得到的矩阵判断这个系统是否相容
- (3) 如果第二问判断为“是”，请得出相对运行率

分析

topological matrix:

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -77 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 & -77 \end{pmatrix}$$

Therefore it is consistent. Fire number: Quelle:77; DCT:77; Q:77; RLC:77; C:1; R:1

5、状态图(StateCharts)

5-1、状态图(StateCharts)

1、什么是状态图

产生背景：经典的自动机不适用于复杂系统（画出来的图过于复杂）

定义：状态图就是显示各个状态及状态转移关系的图，其主要由以下要素组成

活动状态(active state)：当前处于的状态

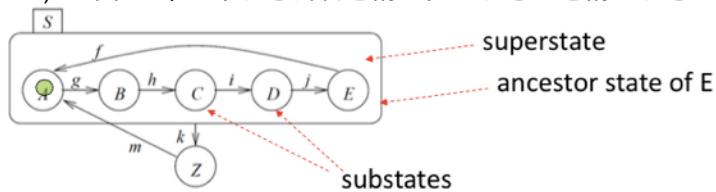
基本状态(basic state)：不包含子状态的状态，用圆形表示

超状态(super state)：包含子状态的状态，用圆角矩形表示

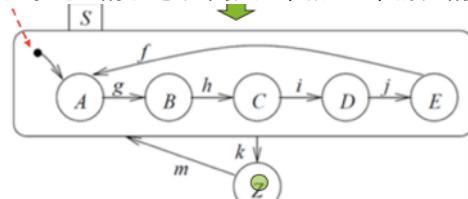
或类型超状态(OR super state)：只要这个超状态中的任意一个子状态处于活动状态，这个超状态就处于活动状态

与类型超状态(AND super state)：只有这个超状态中的全部一个子状态处于活动状态，这个超状态才处于活动状态

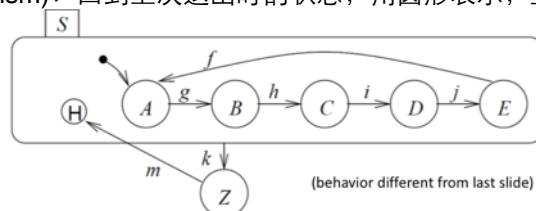
父状态(ancestor state)：对于一个基本状态而言它的直接超状态为它的父状态



默认状态(default state)：初始时进入的状态，用黑色圆点加带箭头的线指向的状态就是默认状态



历史机制(history mechanism)：回到上次退出时的状态，用圆形表示，里面标注H



2、状态集的树状表达

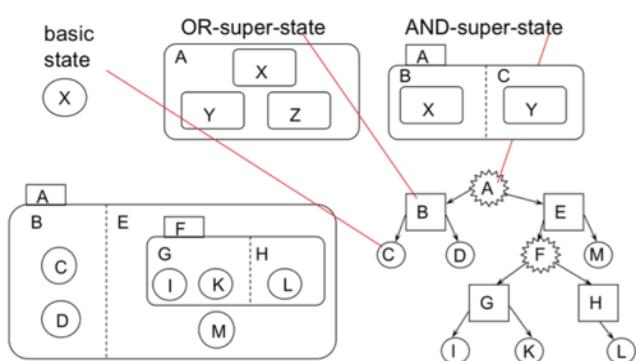
三种基本状态集的树状表达

基本状态：用圆形表示

或类型超状态：用矩形表示

与类型超状态：用星形表示

例子：



3、通过树状表达提取状态集

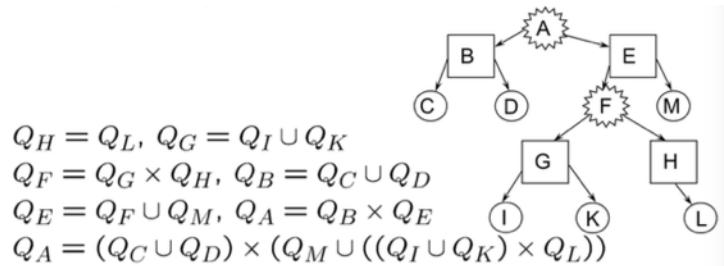
三种基本状态集的转换方法

基本状态的状态集就是基本状态本身

或类型超状态的状态集是它所有子状态的并集

与类型超状态的状态集是它所有子状态的笛卡尔积

例子：



4、状态类型

在状态图中的状态只会是基本状态、或类型超状态、与类型超状态三者之一

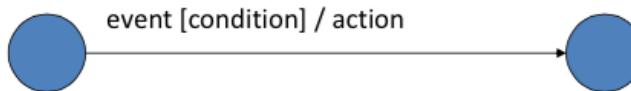
满足下列情况之一的状态不是稳定状态

- 1、状态无法到达
- 2、状态复合转换（存在半个状态的情况）
- 3、静态行为（无法转换为其他的状态）

5、状态转换

状态转换三要素

- 1、事件(event): 外部触发事件，可以是单个事件，也可以是事件的交集、并集、补集
- 2、状态(condition): 状态机的内部状态
- 3、动作(action): 在状态跳转过程中伴随的行为，可以是单个事件，也可以是同时并发的多个事件



只有当事件和状态同时满足时才会发生状态的转移，在转移过程中会伴随执行相应的动作
外部事件只能在系统稳定（上述步骤完成）后才可以进行触发

6、状态图转换为有限状态机

步骤1、画好所有的基本状态

步骤2、对所有的基本状态考虑所有的外部触发事件

步骤3、去除无法到达的基本状态

7、评价

优势：

- 1、层次结构，可以很清晰地表达事件的层次性
- 2、理解简单，状态关系直观
- 3、实现简单，有很多工具可以帮助实现

劣势：

- 1、效率低
- 2、不适合分布式应用（因为无法体现分布式）
- 3、不能描述非功能性行为
- 4、不是面向对象的（面向过程）
- 5、对层次结构的描述不足

5-2、例题(Exercises)

1、对状态图进行评价

优势：

- 1、层次结构，可以很清晰地表达事件的层次性
- 2、理解简单，状态关系直观
- 3、实现简单，有很多工具可以帮助实现

劣势：

- 1、效率低
- 2、不适合分布式应用（因为无法体现分布式）
- 3、不能描述非功能性行为
- 4、不是面向对象的（面向过程）
- 5、对层次结构的描述不足

2、请根据下面的状态图完成下列任务

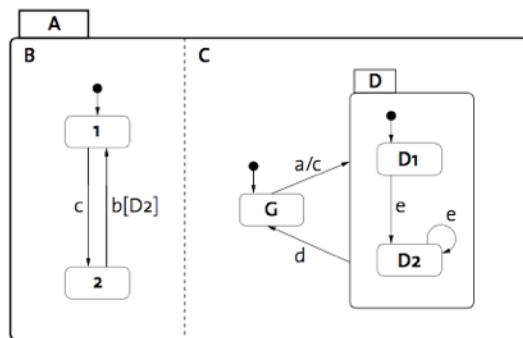


Figure 1: StateChart

- (a)、转换为树状表达
- (b)、转换为状态集表达
- (c)、转换为有限状态机表达

分析

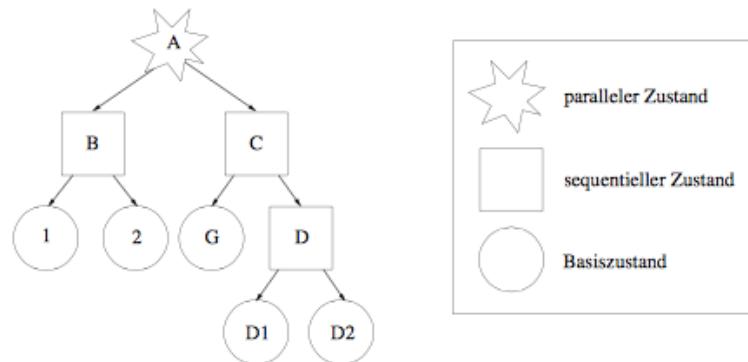
- (a)、

三种基本状态集的树状表达

基本状态：用圆形表示

或类型超状态：用矩形表示

与类型超状态：用星形表示



(b) 、
三种基本状态集的转换方法

基本状态的状态集就是基本状态本身
或类型超状态的状态集是它所有子状态的并集
与类型超状态的状态集是它所有子状态的笛卡尔积

$$\begin{aligned} Z_A &= Z_B \times Z_C \\ &= (Z_1 \cup Z_2) \times (Z_G \cup Z_D) \\ &= (Z_1 \cup Z_2) \times (Z_G \cup (Z_{D1} \cup Z_{D2})) \\ &= (Z_1, Z_G) \cup (Z_1, Z_{D1}) \cup (Z_1, Z_{D2}) \cup (Z_2, Z_G) \cup (Z_2, Z_{D1}) \cup (Z_2, Z_{D2}) \end{aligned}$$

(c) 、转换为有限状态机时遵循下面几步

步骤1、画好所有的基本状态

步骤2、对所有的基本状态考虑所有的外部触发事件

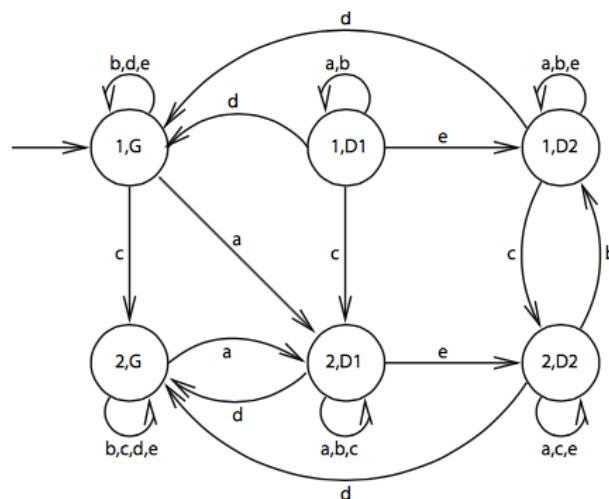


Figure 3: Equivalent FSM (not minimized)

步骤3、去除无法到达的基本状态

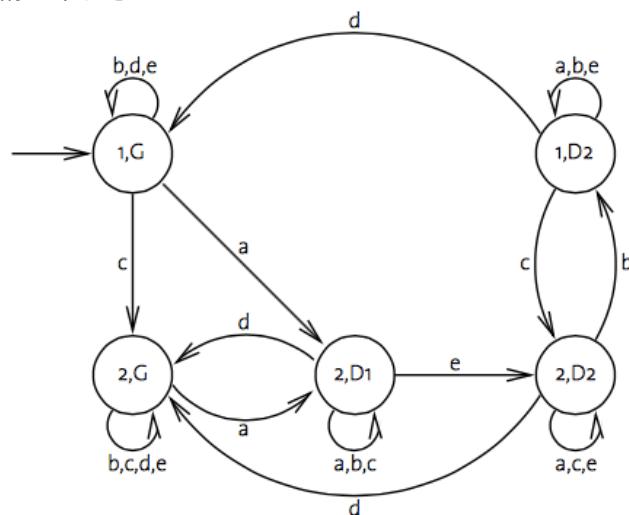


Figure 4: Equivalent FSM (minimized)

第四部分、嵌入式软件 Embedded Software

6、嵌入式软件(Embedded Software)

6-1、嵌入式操作系统(Embedded Operating System)

1、意义：为什么需要操作系统？

这个问题的答案就如同问为什么我们的电脑需要操作系统

单纯的硬件设备并不能满足各种服务要求

存在多种多样的服务，但不存在万能的硬件设备

因此需要软件层的操作系统负责软硬件管理与交互

2、为什么不能用桌面型操作系统

三种操作系统架构

宏内核：将所有的系统相关功能封装在内核中

微内核：只将必要的功能封装在内核中

混合内核：内核封装了介于上述两者之间的功能

桌面型操作系统用的是宏内核，它将所有的系统相关功能封装在内核中

这样的内核已经不能模块化、容错性不强、不可配置、不可修改（这种内核已经做死了，不能动了）

占据太多空间

没有能源优化

也不适合关键任务（因为容错性不强，很容易死机、崩溃）

3、嵌入式操作系统设计要素

可配置性：没有一个单独的实时系统可以满足所有需要，因此需要一个可配置的系统

最简形式：将无用的功能全部去除，只保留最基本的功能

条件编译：使用#if和#endif命令进行条件编译

用动态数据替代静态数据（但这样可能导致严重后果）

设备驱动：嵌入式系统中的设备驱动是基于特定任务的，而不是一开始就集成好的

提升可预见性与驱动的适应性（驱动跟着软件走，不用考虑操作系统的版本问题）

最大限度减少冗余

中断：任何进程都可以发起中断

在标准的操作系统中，这是不可靠性的主要来源

然而在嵌入式系统中可以尝试这样做

加入中断会比进程调用更有效率

但是反过来中断会带来可组合性问题

一个绑定了中断的任务很难再与其他任务发生关联

如果关联可能导致其他任务的不确定性

4、实时操作系统：一种嵌入式操作系统

定义：实时操作系统是支持实时系统架构的操作系统

要素：

1、能预测时序行为

操作系统下的所有服务必须在规定时间之前完成

必须要是一个确定性系统

只能在规定时间内运行

几乎所有的活动都在调度器的控制之下（完全控制）

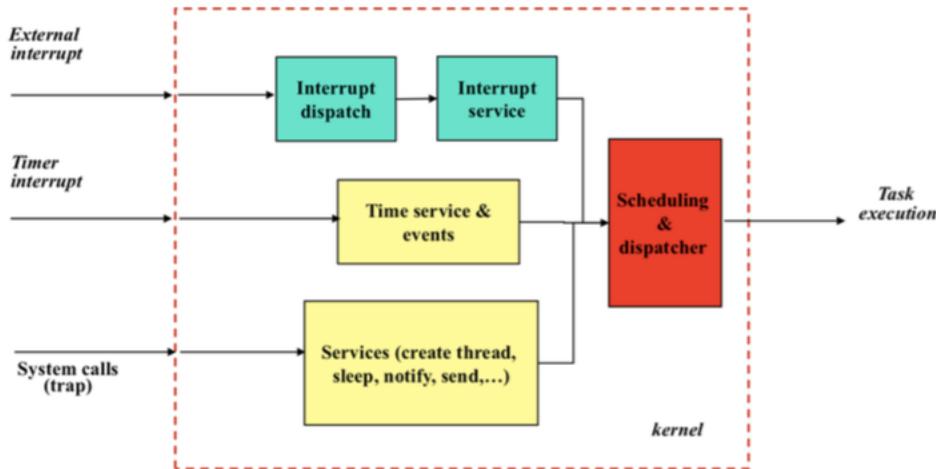
2、能正确管理时序和调度

能够意识到任务的截止时间并给予合理的调度

操作系统的时钟必须精确

3、速度要快

结构：



实时系统内核的主要功能：进程管理

通过进程或线程在处理器上执行准并行任务

维护进程状态、进程队列

任务抢占、中断处理

CPU调度（保证在截止时间前完成任务，最小化进程等待时间，平等使用系统资源）

进程同步（临界区处理、信号量、管程、互斥资源）

进程间通信

实时时钟

5、复习：操作系统的一些知识

进程状态：

运行：正式进入CPU中开始运行

就绪：具备执行的能力但还没分配到CPU时间

等待：由于等待事件发生而处于阻塞状态

线程：

线程是程序执行流的最小单元

线程与进程的最根本区别是：线程间会共享一部分变量（比如：内存）

线程的优势和性质

线程间切换速度快

一个应用程序会为每个不同的活动分配一个单独的线程

线程执行的信息会被保存在线程控制块（TCB）中

进程同步：

在传统的操作系统中进程同步依靠信号量和管程实现

在实时操作系统中特殊的信号量和调度方法会被用来解决进程同步问题

进程间通信：

同步通信

两进程必须事先约定好消息的传输方式

并且要双方都准备好了以后才可以通信

在实时系统中会出现进程同步问题

在静态实时系统中可以通过预先设定优先级解决进程同步问题

异步通信

不需等到对方准备好以后再通信

发送端将消息扔到邮箱里面

接收方从邮箱里面收取信息

比同步通信更加适合实时系统

缓冲区有限，有可能阻塞写入或读取操作

6-2、资源访问协议(Resource Access Protocols)

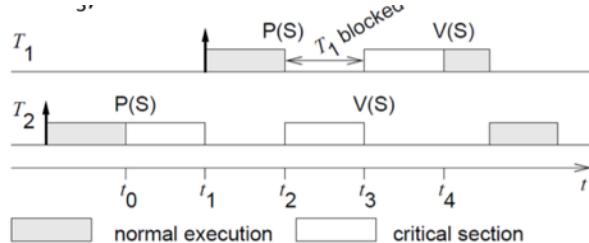
1、为什么需要定义资源访问协议？

系统中很多资源都是共享的，对这些资源进行访问都要在临界区中进行
一般情况下我们通过信号量解决对共享资源的访问问题

获得信号量的进程会处于就绪状态，等待信号量的进程会处于等待状态
所有等待信号量的进程会被放入一个等待队列中等待获得信号量的线程进行P操作

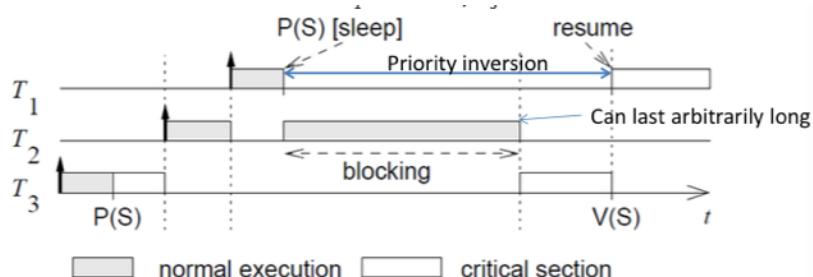
2、优先级反转

例子1：



在这个例子中进程T1的优先级高于进程T2，但T1在T2进入临界区以后才进来，这就导致了T1的P操作被阻塞，必须等到T2把临界区里面的任务做完以后用V操作释放临界区资源后T1才可以进入临界区。因此此时T1的优先级降到了T2，使得T2可以正常执行。这样做看似合情合理，T1被拥有临界区的进程所阻塞理应把高优先级给到哪个进程。但是如果是下面这个例子的情况就不会这样认为了。

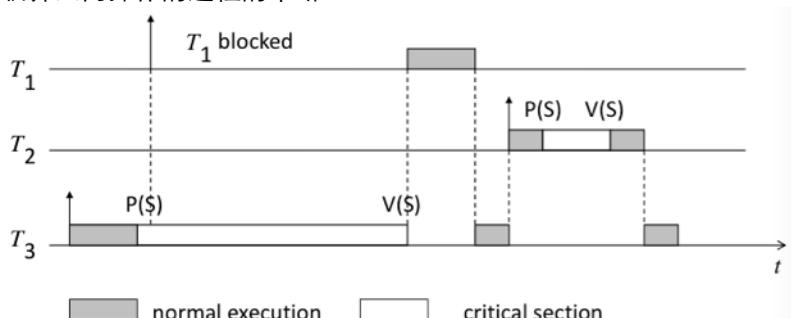
例子2：



在这个例子中进程T1的优先级高于进程T2，进程T2的优先级高于进程T3.T3先进入临界区，然后T2、T1进来。但是这个时候T1也要访问临界区，而临界区被T3所占着。此时T1只好把自己的优先级降低到T3。但这时T2的优先级比T3高，而T2还有一个很长的非临界区任务未完成。实际上还得得到这个任务完成后优先级较低的T3才可以进行临界区操作。因此最后造成了T1实际上是被T2所阻塞的异常情况

3、优先级反转问题的解决方法

方法1、屏蔽所有在临界区内操作的进程的中断



在这个例子中T3处于临界区里面，虽然在临界区执行的过程中优先级较高的T1进来，但是T3继续它的临界区操作。但这样又带来了另一个问题，这里的T1是不需要临界区的，本来第一时间就可以完成的却拖了很久等到T3从临界区中出来才可以执行。这种解决方法并不是一个很好的方法。

方法2、资源访问协议

4、资源访问协议

基本思想：修改导致阻塞的进程的优先级。当一个优先级较低的线程阻塞了一个优先级较高的线程时会将这个优先级较低的线程的优先级提高

方法

- 1、优先级继承协议(PIP)，适用于静态优先级的情况
- 2、优先级上限协议(PCP)，适用于静态优先级的情况
- 3、堆栈资源策略(SRP)，适用于动态静态优先级的情况

5、优先级继承协议(PIP)

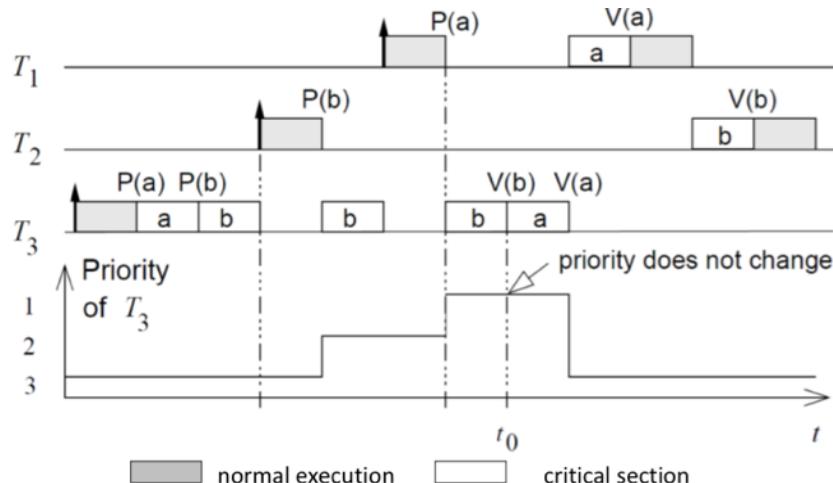
前提：优先级固定

基本思想：当一个优先级较低的线程阻塞了一个优先级较高的线程时会将这个优先级较低的线程的优先级提高至被阻塞的进程的优先级

算法描述：

- 1、如果进程 T_i 试图进入临界区会出现下列两种情况
如果临界区资源被一个较高优先级的进程占用就会被阻塞，不进行后续操作
如果临界区资源被一个较低优先级的进程占用就会被阻塞，并进行第二步
否则就立即进入临界区
- 2、当进程 T_i 被阻塞时，它将自己的优先级给到当前拥有信号量的进程 T_k 。在当前拥有信号量的进程 T_k 离开临界区之前会保持这个优先级
- 3、当进程 T_k 离开临界区以后，会使用 V 操作将信号量释放，然后这个信号量会给出到当前阻塞队列中优先级最高的进程。如果只有一个进程被阻塞，那么 T_k 会立即恢复为它原本的优先级，否则会改变为现在仍在阻塞队列中的进程的最高优先级

例子1：



在前面三个时间段 T_3 分别完成了一次临界区外操作，一次进入临界区 a 后的操作，一次从临界区 a 再进入临界区 b 后的操作；

第四个时间段 T_2 进来，由于此时 T_2 的优先级最高，因此这个时段 T_2 完成了一次临界区外操作；

到了第五个时间段 T_2 想进入临界区 b ，但发现这个已经被较低优先级的 T_3 占着，根据算法， T_2 被阻塞，同时将 T_3 的优先级抬升至 T_2 ，从而第五个时段是进行 T_3 临界区内的操作；

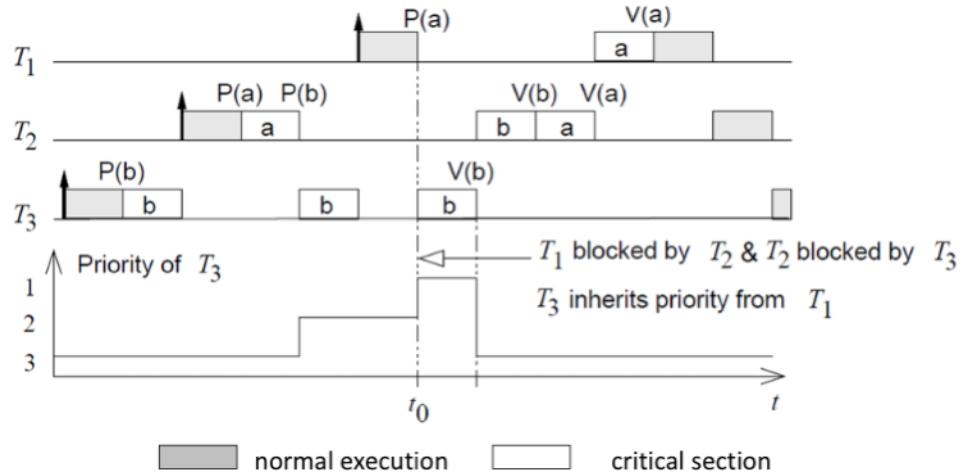
第六个时段 T_1 进来，由于此时 T_1 的优先级最高，因此这个时段 T_1 完成了一次临界区外操作；

到了第七个时间段 T_1 想进入临界区 a ，但发现这个已经被较低优先级的 T_3 占着，根据算法， T_1 被阻塞，同时将 T_3 的优先级抬升至 T_1 ，从而第七个时段是进行 T_3 临界区内的操作。在 T_3 操作完成后退出临界区 b ，根据算法这个释放的信号量会给出到当前等待这个信号量的最高优先级进程 T_2 ，并把自己的优先级改变为当前仍被阻塞的进程的最高优先级。而当前被阻塞的最高优先级是 T_1 的优先级，因此 T_3 仍保持原来 T_1 的优先级不变；

第八个时段，此时 T_1 被阻塞， T_2 的优先级不如 T_3 ，因此继续做的是 T_3 的操作。 T_3 操作完成后退出临界区 a ，根据算法这个释放的信号量会给出到当前等待这个信号量的最高优先级进程 T_1 。而此时已经没有被阻塞的进程了，因此 T_3 恢复为原来的优先级 T_3 ；

后面的时段，已经没有进程被阻塞，因此总是执行最高优先级的进程

例子2：



在前面两个时间段 T_3 分别完成了一次临界区外操作，一次进入临界区 b 后的操作；

第三、四个时间段 T_2 进来，由于此时 T_2 的优先级最高，因此这个时段 T_2 完成了一次临界区外操作，一次进入临界区 a 后的操作；

到了第五个时间段 T_2 想进入临界区 b ，但发现这个已经被较低优先级的 T_3 占着，根据算法， T_2 被阻塞，同时将 T_3 的优先级抬升至 T_2 ，从而第五个时段是进行 T_3 临界区内的操作；

第六个时段 T_1 进来，由于此时 T_1 的优先级最高，因此这个时段 T_1 完成了一次临界区外操作；

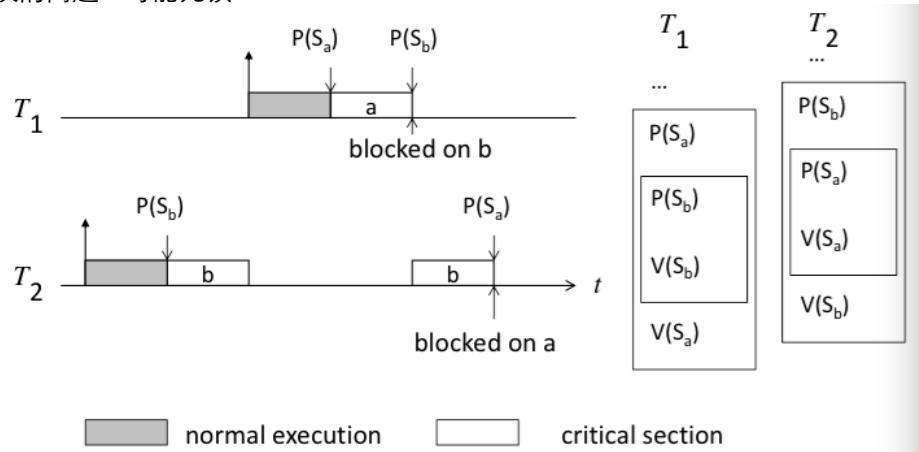
到了第七个时间段 T_1 想进入临界区 a ，但发现这个已经被较低优先级的 T_2 占着，根据算法， T_1 被阻塞，同时将 T_2 的优先级抬升至 T_1 。但是此时最高优先级的进程 T_2 正被比它更低优先级的进程 T_3 所阻塞，于是 T_2 又把 T_1 进程的优先级给到 T_3 。从而第七个时段是进行 T_3 临界区内的操作。在 T_3 操作完成后退出临界区 b ，根据算法这个释放的信号量会给予当前等待这个信号量的最高优先级进程 T_2 ，并把自己的优先级改变为当前仍被阻塞的进程的最高优先级。而当前已经没有进程被 T_3 所阻塞了（ T_1 是被 T_2 所阻塞的，跟 T_3 没有关系），因此 T_3 恢复为原来的优先级 T_3 ；

第八个时段，此时 T_1 被阻塞， T_2 的优先级最高，因此做的是 T_2 的操作。 T_2 操作完成后退出临界区 b ，根据算法这个释放的信号量会给予当前等待这个信号量的最高优先级进程。而此时虽然已经没有等待这个信号量的进程，但 T_1 仍然被进程 T_2 所阻塞，因此 T_2 的优先级仍为 T_1 ；

第九个时段，此时 T_1 被阻塞， T_2 的优先级最高，因此做的是 T_2 的操作。 T_2 操作完成后退出临界区 a ，根据算法这个释放的信号量会给予当前等待这个信号量的最高优先级进程 T_1 。而此时虽然已经没有进程被进程 T_2 所阻塞，因此 T_2 的优先级恢复为 T_2 ；

后面的时段，已经没有进程被阻塞，因此总是执行最高优先级的进程

优先级继承协议的问题：可能死锁



在上面这个例子中前面两个时间段T2分别完成了一次临界区外操作，一次进入临界区b后的操作；

第三、四个时间段T1进来，由于此时T1的优先级最高，因此这个时段T1完成了一次临界区外操作，一次进入临界区a后的操作。然后T1想进入临界区b，但此时临界区b被T2占据，因此T1被阻塞并将它的优先级给到T2；

第五个时段由于T1被阻塞，因此T2执行。然后T1想进入临界区b，但此时临界区b被T1占据，因此T2被阻塞。最终导致T1被T2所阻塞，T2被T1所阻塞的死锁局面

不足之处：

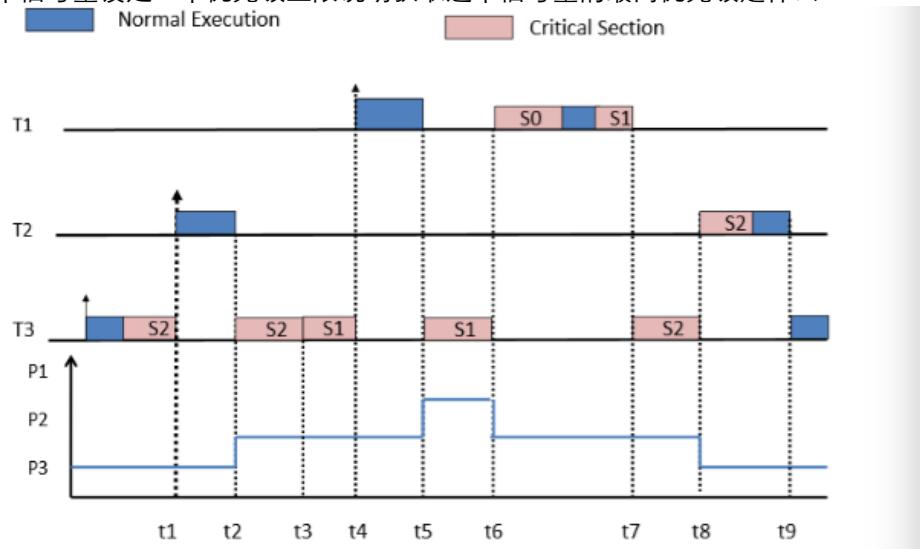
- 1、可能导致很多进程的优先级都很高
- 2、可能导致死锁
- 3、必须固定优先级

6、优先级上限协议(PCP)

前提：优先级固定

基本思想：一个进入临界区的进程不会被低优先级进程所阻塞

为每个信号量设定一个优先级上限说明获取这个信号量的最高优先级是什么



比如在这个例子中，申请信号量S0的进程有T1，因此信号量S0的优先级上限为T1的优先级；

申请S1的进程有T1、T3，因此信号量S1的优先级上限为T1的优先级；

申请S2的进程有T2、T3，因此信号量S2的优先级上限为T2的优先级；

算法描述：

- 1、当一个进程T想要得到信号量Sk时，Sk会分配给进程T当且仅当进程T的优先级大于信号量S*的优先级上限

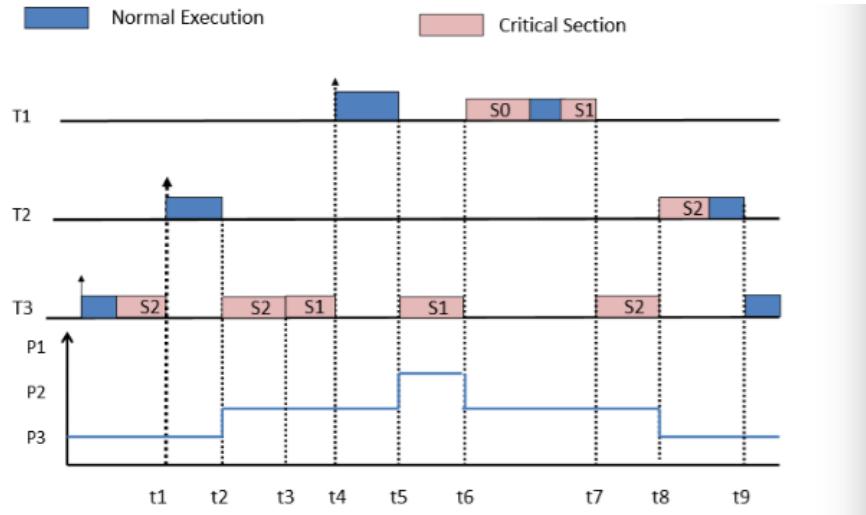
S*为被进程T以外的进程所拥有的信号量中具有最高优先级上限的的信号量

如果进程T不满足拥有信号量Sk的条件则认为进程T被信号量S*所阻塞

如果进程T被信号量S*所阻塞，那么进程T的优先级将会给到当前拥有信号量S*的进程

- 2、当一个进程T*释放信号量S*时，系统将信号量S*给到当前被信号量S*所阻塞的最高优先级进程
T*的优先级将设为它还拥有的信号量所阻塞的进程的最大优先级
如果T*已经不拥有任何进程了，就恢复为初始时的优先级

例子：



在t1以前，进程T3想要得到信号量S2。根据算法，S2会分配给进程T3当且仅当进程T3的优先级大于信号量S*的优先级上限，而S*为被进程T3以外的进程所拥有的信号量中具有最高优先级上限的的信号量。但是T3以外并没有其他进程，因此满足这个条件，S2分配给了T3；

t2时刻，T2也想要得到信号量S2。根据算法，S2会分配给进程T2当且仅当进程T2的优先级大于信号量S*的优先级上限，而S*为被进程T2以外的进程所拥有的信号量中具有最高优先级上限的的信号量。在这里易知进程T2以外的进程指的是T3，而S*指的是S2。S2的最高优先级上限就是T2的优先级，因此并不满足大于的条件，T2被S2所阻塞，因此T2的优先级会给到当前拥有S2的优先级T3；

t3时刻，进程T3想要得到信号量S1。根据算法，S1会分配给进程T3当且仅当进程T3的优先级大于信号量S*的优先级上限，而S*为被进程T3以外的进程所拥有的信号量中具有最高优先级上限的的信号量。但是T3以外并没有其他进程拥有信号量（进程T2没有拿到信号量），因此满足这个条件，S1分配给了T3；

t5时刻，T1想要得到信号量S0。根据算法，S1会分配给进程T1当且仅当进程T1的优先级大于信号量S*的优先级上限，而S*为被进程T1以外的进程所拥有的信号量中具有最高优先级上限的的信号量。在这里易知进程T1以外的进程指的是T3，而S*指的是S1。S1的最高优先级上限就是T1的优先级，因此并不满足大于的条件，T1被S1所阻塞，因此T1的优先级会给到当前拥有S1的优先级T3；

t6时刻T3释放信号量S1，由于信号量S1没有阻塞任何的进程，因此直接释放。然后T3的优先级将设为它还拥有的信号量所阻塞的进程的最大优先级，即T2的优先级；

t8时刻T3释放信号量S2，此时系统将信号量S2给到当前被信号量S2所阻塞的最高优先级进程T2。然后T3已经不拥有信号量了，因此恢复为初始时的优先级。

评价

这个算法比PIP的好处在于

- 1、不会死锁
- 2、一个进程最多只会被每个低优先级进程阻塞一次

但还是不能应对优先级改变的情况

7、堆栈资源策略(SRP)（仅作了解）

背景：PCP的进化版本，可以应对优先级改变的情况

基本思想：阻止一些情况下进程的抢占

进程*i*抢占级数*li*：一个与截止时间相关的单调减函数

截止时间越长越容易被抢占

资源上限：访问某个临界区的所有进程的最大抢占级数

系统上限：所有被阻塞的资源中的最大资源上限（这个是一个动态值）

算法描述：一个进程可以抢占另一个进程当且仅当

- 1、它拥有最高优先级
- 2、它的抢占级数大于系统上限

6-3、例题(Exercises)

1、说明私有资源、共享资源和互斥资源三者间的区别。然后用例子说明互斥资源会遇到的一个问题

私有资源只能被一个进程所使用，共享资源可以被多个进程所使用

共享资源可以同时被多个进程访问，互斥资源在某一时刻最多只能被一个进程所访问

互斥资源会遇到临界区问题（共享变量的访问问题）

例子：

在生产者消费者问题中：

生产者执行：

```
register 1 = count  
register 1 = register 1 + 1  
count = register1
```

消费者执行：

```
register 2 = count  
register 2 = register 2 - 1  
count = register2
```

假设一开始的count值为5

在一般情况下生产一个再消费一个最后得到的count应该是和原来相同，还是5

但是如果是下面这种情况：

```
register 1 = count  
register 1 = register 1 + 1
```

生产者执行完这两步就被消费者中断掉了

```
register 2 = count  
register 2 = register 2 - 1
```

消费者执行完这两步又被生产者中断掉了

```
count = register1  
count = register2
```

最终得到的count值就不是5了，而是4

这个也就是临界区问题：对共享数据的并发访问可能导致数据的不一致

2、请说出解决临界区问题的三种方法

设定进程非抢占，硬件上屏蔽中断，静态调度（一开始就认为规定好调度方式），信号量

3、请解释信号量是怎样解决临界区问题的

在信号量中定义了两种原子操作

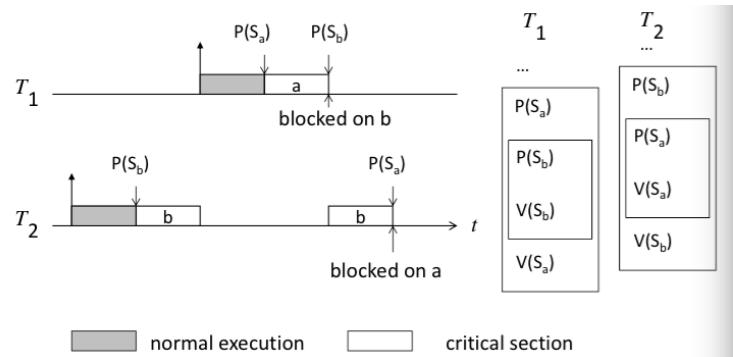
P操作：请求进入临界区的锁

V操作：释放临界区的锁

一个进程A要想进入到临界区中必须先通过P操作请求进入临界区的锁，如果临界区里面有其他进程，那么进程A就必须等到哪个进程从临界区里面出来（被放入这个信号量的阻塞队列中），通过V操作释放锁以后进程A才可以进入临界区；如果临界区里面没有其他进程，那么进程A就可以立即进入到临界区。符合有空让进，无空等待，择一而入，算法可行的临界区问题解决原则。

4、PIP是否会遇到死锁问题，请举例说明

PIP会有死锁问题：



在上面这个例子中前面两个时间段T2分别完成了一次临界区外操作，一次进入临界区b后的操作；

第三、四个时间段T1进来，由于此时T1的优先级最高，因此这个时段T1完成了一次临界区外操作，一次进入临界区a后的操作。然后T1想进入临界区b，但此时临界区b被T2占据，因此T1被阻塞并将它的优先级给到T2；

第五个时段由于T1被阻塞，因此T2执行。然后T1想进入临界区b，但此时临界区b被T1占据，因此T2被阻塞。最终导致T1被T2所阻塞，T2被T1所阻塞的死锁局面

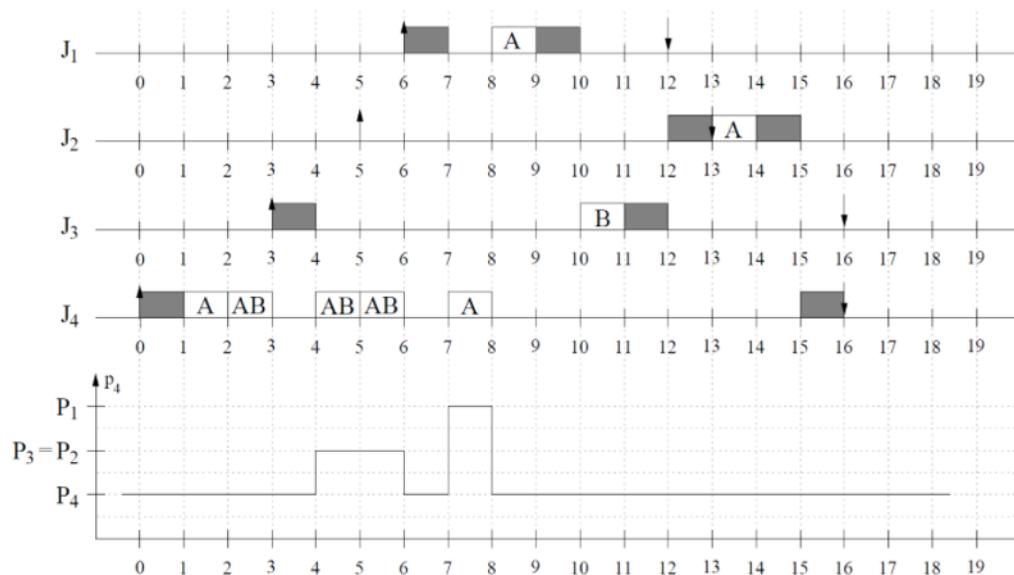
5、请使用PIP的调度方式对下列四个任务J1、J2、J3、J4进行调度

	J1	J2	J3	J4
Check in	6	5	3	0
Deadline	12	13	16	16
Execution time	3	3	3	7
Nominal priority	highest	medium	medium	lowest
Structure of tasks	A	A	B	A AB AB AB A

请画出调度的过程和J4优先级的变化

分析：

调度过程：



解释：

0~3时只有J4这一个任务，因此J4申请到了临界区A和临界区B的资源；

3时刻J3任务到来，此时J3的优先级最高，因此抢占任务J4，开始运行J3；

4时刻J3想进去临界区B，但此时临界区B被J4所占用，因此J3被J4阻塞，将它的优先级给到J4；

5时刻J2进来，此时没被阻塞的任务有J2和J4。此时由于J4继承了J3的优先级，因此具有和J2一样的优先级。优先级相同根据先到先得的原则因此运行的还是J4；

6时刻J4释放临界区B，此时被阻塞的任务J3被激活。此时J4所占据的临界区已经没有阻塞其他进程，因此J4恢复为原本的优先级。同时J1任务到来，此时J1的优先级最高，因此抢占任务J3（如果J1没过来现在运行的就是任务J3），开始运行J1；

7时刻J1想进去临界区A，但此时临界区A被J4所占用，因此J1被J4阻塞，将它的优先级给到J4；

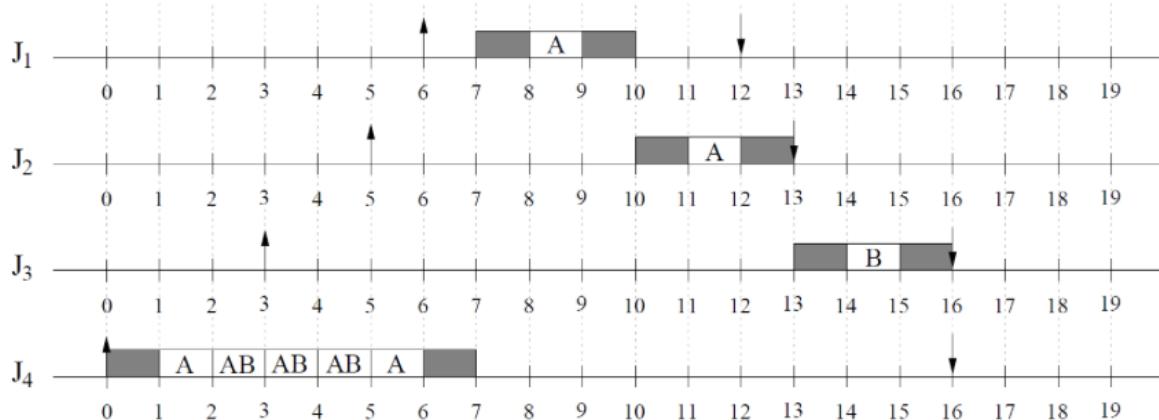
8时刻J4释放临界区A，此时被阻塞的任务J1被激活（J1和J2都被阻塞，但由于J1的优先级比较高，因此激活的是J1）。此时J4所占据的临界区已经没有阻塞其他进程，因此J4恢复为原本的优先级；

9时刻J1释放临界区A，此时被阻塞的任务J2被激活。由于J1的优先级本来就比J2的高，因此J1、J2的优先级还是初始时的优先级；

后面的调度就根据优先级进行了，在优先级相同时遵循的是先到先得的原则

6、请自己设计一种调度方式对下列四个任务J1、J2、J3、J4进行调度，要求使得这四个任务都可以在截止时间前完成

一种方法是把四个进程的优先级改变一下，变成J4 > J1 > J2 > J3

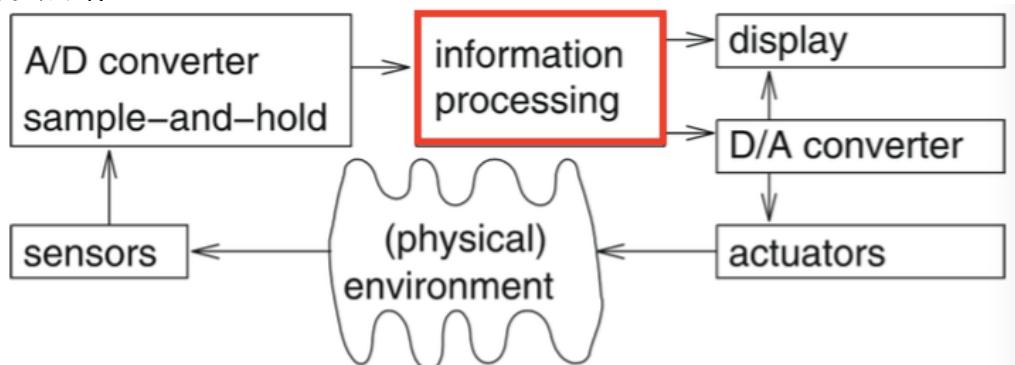


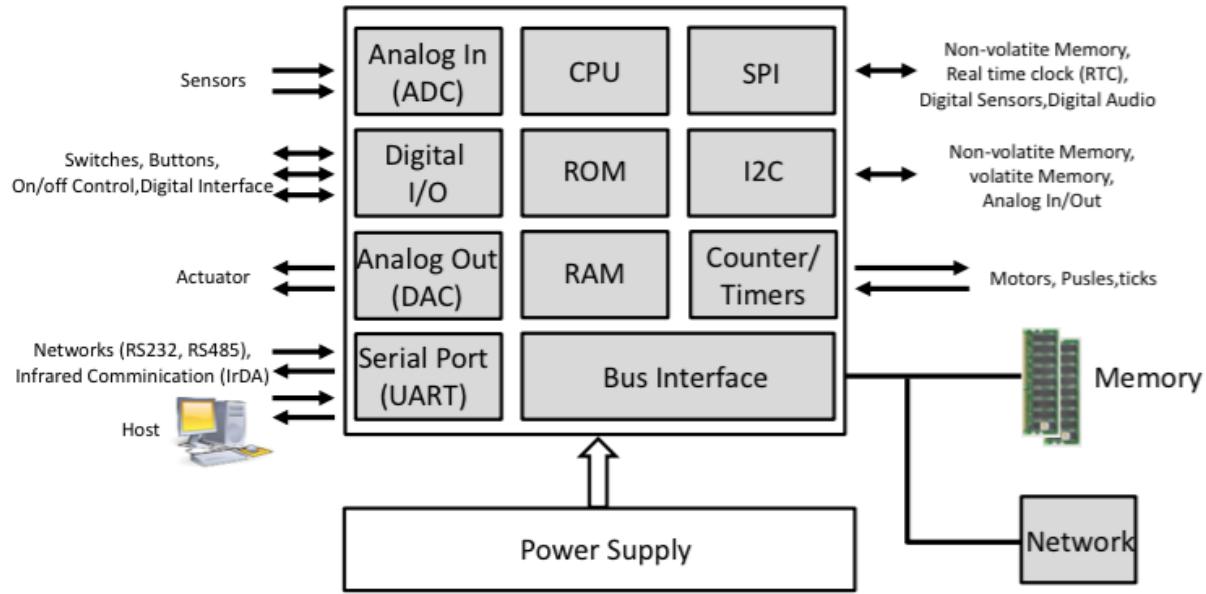
第五部分、嵌入式系统硬件 Embedded System Hardware

7、嵌入式系统硬件基础(Embedded System Hardware Basics)

7-1、嵌入式计算系统(Embedded Computing System)

1、嵌入式系统硬件





7-2、电路基础(Electronics Basics)

1、电阻、电容、电感、二极管

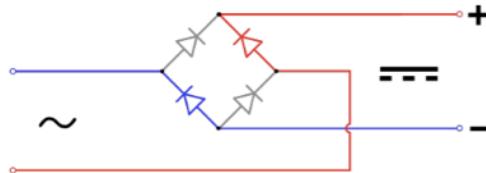
电阻在电路中起到阻碍作用

电容在电路中起到储能（电池）、镇流（镇流器）、去除高频噪声的作用

电感在电路中起到阻碍电流变化的作用

二极管在电路中起到不对称的电导作用（只能单向导通）

二极管桥起到转交流为直流的作用



2、运算放大器

特性：无限大的输入阻抗，零输出阻抗，无限大开环增益，无限大的共模抑制比

铁律：虚短（输入的电势相同），虚短（输入的电流为0）

3、滤波器分析方法

步骤1：将电容、电感、电阻转换为阻抗的表达形式（其中s为拉普拉斯变换）

电容：

$$\frac{1}{sC}$$

电感：

$$sL$$

电阻：

$$R$$

步骤2：通过基尔霍夫电压定律和基尔霍夫电流定律对电路进行分析即可得出传递函数

步骤3：通过传递函数表达式可以得出拐点频率

对于一阶系统来说（ s 的最高次幂为1）时的传递函数表达式和拐点频率间的关系为

$$H = \frac{I}{I + Ks} \rightarrow f_c = \frac{I}{2\pi K}$$

对于二阶系统来说（ s 的最高次幂为2）时的传递函数表达式和拐点频率间的关系为

$$H = \frac{I}{\omega_0^2 + 2\beta s + s^2} \rightarrow f_c = \frac{\omega_0}{2\pi}$$

7-3、供电系统的设计(Power System Design)

1、常用元件（仅作了解）

低压差调节阀：用于降压

开关调节阀：用于启用或关断电源

2、电源设计准则

清楚需要多大的输入输出电压

提供的能量要能满足系统的需求

高效性

减少电源噪声

7-4、外设连接：SPI和I2C(Peripherals Connections: SPI and I2C)

1、SPI（串行外设接口）

SPI的特点：

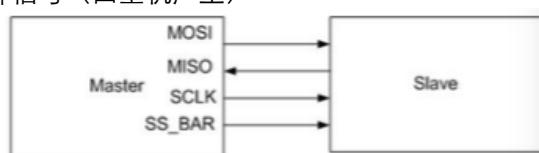
四线总线（每条总线里面包含四条线）

MOSI：主机输出，从机输入

MISO：主机输入，从机输出

SCLK：时钟信号（由主机产生）

SS：从机选择信号（由主机产生）

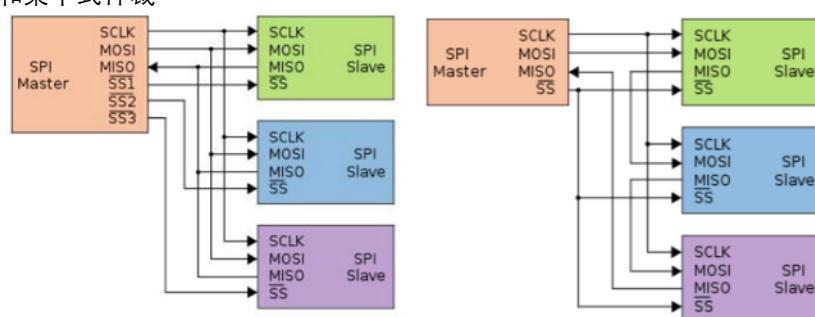


用于短距离传输数据

单主多从

全双工同步传输

SPI的分布式仲裁和集中式仲裁



Master and multiple independent slaves

Master and multiple daisy-chained slaves

时钟相位

两个参数对应了四个模式

CPOL: 时钟优先级

0: 初始时钟为0

1: 初始时钟为1

CPHA: 时钟相位

0: 上升沿收数据, 下降沿传数据

1: 下降沿收数据, 上升沿传数据

进行交互的主从机的这两个参数必须相同, 否则会出现数据冲突 (主机必须调整参数与从机匹配)

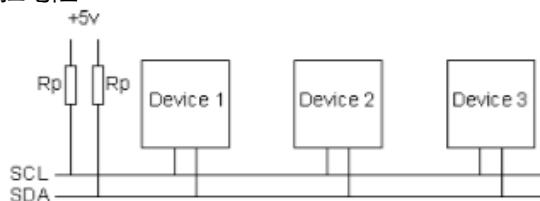
2、I2C总线

两线总线 (每条总线里面包含两条线)

SCL: 传输时钟信号

SDA: 传输数据信号

两线总线共地, 同时需要有上拉电阻



3、UART (通用异步收发传输器)

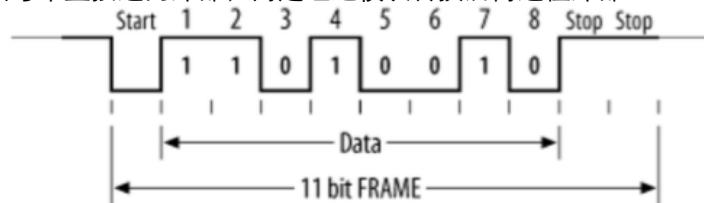
计算机串行通信子系统的关键部件

可以双向传输 (以数据流的方式进行传输)

没有时钟信息



在传输过程中传输的信号不直接通向外部, 而是经过模块转换后再通往外部



在中间传输的串行数据以低电平作为数据的开始, 以高电平作为数据的结束

4、RS485元件 (仅作了解)

工业上主要用于低功耗网络

半双工的主从架构

可以传输多达1200米的距离

5、比较: SPI、I2C、UART

	SPI (串行外设接口)	I2C总线	UART (通用异步收发传输器)
总线中包含线路数	4	2	2
主机数	单主机	多主机	单主机
速度	最快	慢	快
时序	同步	同步	异步

7-5、微控制器：Atmel AVR芯片(Microcontrollers: Atmel AVR Chips)

1、微控制器的最小系统

微控制器的最小系统是能满足最基本控制功能的集成电路
一般包含：振荡电路、复位电路、电源、调试电路、存储器电路

2、振荡电路

用于产生稳定的时钟信号
整个微控制器的运行都依赖于这个时钟频率
更高的时钟频率意味着更高的性能，同时也会带来更高的功耗

8、嵌入式处理器(Embedded Processor)

8-1、处理器类型(Processor types)

1、通用处理器 (GPP)

通用处理器特性：

1、高性能

对电路做了深度优化
利用并行性
超标量体系结构：动态调度指令的执行顺序
超级流水线：分支预测，hazard处理
复杂的存储器架构

2、不适合实时应用

因为出现动态调度因此没办法对时间进行分析

3、其他特性

对于大型应用有着很好的平均性能
但也有着很高的功耗

通用处理器架构：

冯诺依曼架构：将程序存储器和数据存储器合二为一
哈佛架构：分开程序存储器和数据存储器

2、指令集类型

复杂指令集 (CISC)

直接将复杂指令映射成机器语言
每个复杂指令都由多个简单指令组合而成
可能会导致时序不一致的问题（每个复杂指令的执行时间不一定相同）
一般用于通用计算

简单指令集 (RISC)

只有简单的指令，因此已是需要将高级程序设计语言转换为简单的指令
每条指令执行时间几乎完全相同
一般用于嵌入式系统

3、专用指令集处理器

微控制器 (MicroCtrl)

用于基于控制的系统
系统由事件所驱动
例子：车载控制系统、消费类电子系统

数字信号处理器 (DSP)

用于基于数据的系统
是一个面向数据流的周期性行为系统
例子：信号处理

超长指令字处理器 (VLIW)

用于基于数据的系统
例子：图像处理

4、微控制器（仅作了解）

用于控制领域的应用

支持进程的调度与同步

支持抢占

比较短的延迟

低功耗

集成了一些外围单元（比如说AD/DA转换器、PWM控制器）

适合实时应用

5、数字信号处理器（仅作了解）

为数据流应用做了特定优化

适合简单的控制流

并行的硬件单元

特殊的指令集

高数据吞吐量

零开销循环

特殊的存储器

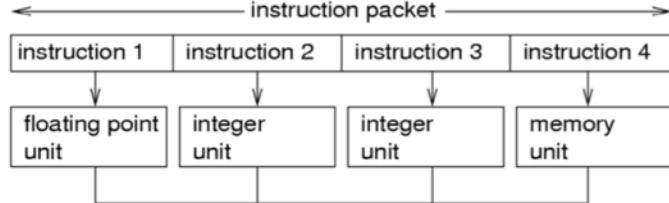
适合实时应用

6、超长指令字处理器（仅作了解）

设计思想：应该由编译器进行并行检测，而不应该在运行时由硬件进行并行检测

做法：将并行操作的指令合并到一个长句里面，每个长句表示一个功能单元

因此超长指令字处理器也被称为是显式并行指令处理器



7、专用集成电路（ASIC）

用户自己定制的专用集成电路

一般用于对某些属性有较高要求的，产量大的电路

但这种需要很长的设计时间，而且一旦流片就无法更改，每次流片都是一笔不小的开销

8、可重构处理单元（RPU）

设计思想：

完全的硬件芯片灵活性太差（一旦做成硬件就没法更改），成本也很高

完全的软件仿真速度又很慢

可重构处理单元意在既结合硬件的速度，又结合软件的灵活性

硬件可编程：硬件的功能和接口可以编程

硬件可以动态重构

典型产品：现场可编程门阵列（FPGA）

使用RPU的目前最先进的技术

有着非常广泛的应用

9、专业化系统

通用微处理器和嵌入式系统处理的最主要区别在于是通用的还是专业化的

但专业化也不能太专，也要兼顾一定的灵活性

需要附加一些相关的多余功能，以备日后增加功能时使用

10、为什么有这么多的处理器类型

不同功能不同应用领域的嵌入式系统对性能、功耗的要求有所不同

8-2、片上多处理器系统(Multi Processor System on Chip (MPSoC))

8-2-1、概念(Concepts)

1、什么是片上多处理器系统

定义：在一个芯片上互连了处理单元、存储单元、IO模块的异构处理系统

背景：意在既满足于性能要求的情况下尽量降低功耗

2、为什么要使用片上多处理器系统

原因1：摩尔定律

每两年进程在单位面积上的晶体管数翻一番

原因2：功耗墙

处理核的运行速度越快，功耗越大，产热越多

因此意图增多处理核进行处理，把每个核的运行速度降下来（用数量换速度）

3、片上多处理器系统的问题

问题1：存储器墙

存储器的发展远远跟不上处理速度

存储器的带宽瓶颈直接影响到处理速度不能太快（因为需要储存处理出来的数据）

问题2：指令级并行 (ILP) 墙

指令级并行程度越高（流水线级数越多），功耗可能越大，hazard问题更严重

问题3：功耗仍是一个主要因素

每个处理器的功耗下去了，但是总体功耗并没有下去

芯片集成度越高，漏电流变得很大（很容易被击穿）

8-2-2、big.LITTLE技术(ARM's big.LITTLE technology)（仅作了解）

1、什么是big.LITTLE技术

ARM研究出的big.LITTLE技术是基于同步自适应对称多核处理器芯片的技术

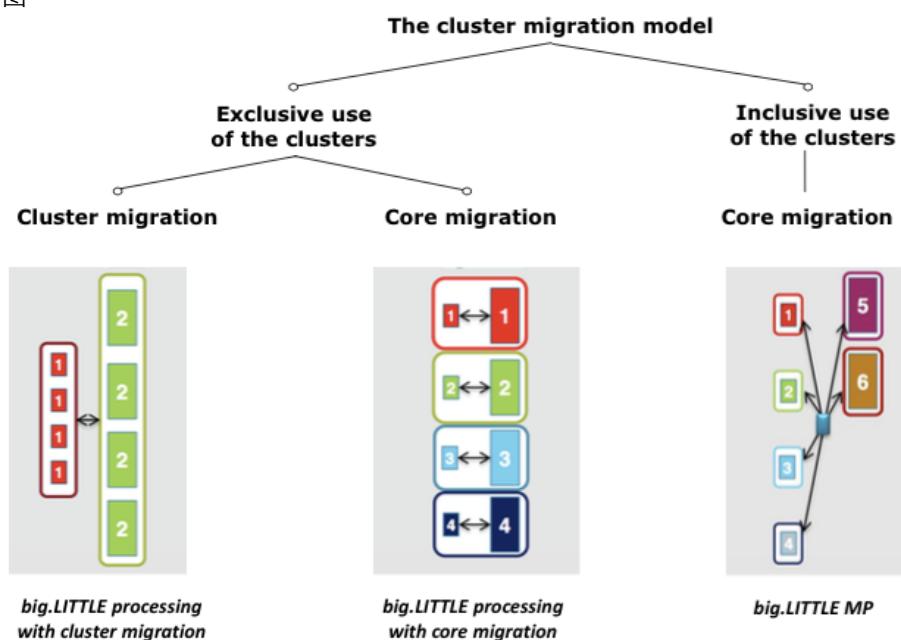
对称多核处理器：每个处理器核都以相同的方式访问主存

在这项技术中有两种处理器核

低功耗处理器核

高性能处理器核

2、技术结构图



分支技术1：聚簇融合技术：任务在某一时刻只能在一种处理器核中运行

某一时刻只激活一种处理器核

对于低负载的任务放在低功耗处理器核中运行

对于高负载的任务放在高性能处理器核中运行

具体在哪个核中运行由操作系统决定

分支技术2：处理器核融合技术：将一个低功耗处理器核和一个高性能处理器核打包组合

对于每个组只激活其中一个处理器核

对于低负载的任务放在低功耗处理器核中运行

对于高负载的任务放在高性能处理器核中运行

组与组之间独立选择激活哪个处理器核

具体在哪个核中运行由操作系统决定

分支技术3：另一种处理器核融合技术：将所有处理器核都给到操作系统进行管理

操作系统根据任务的实际情况给任务分配处理器核

9、信号处理(Signal Processing)

9-1、模数转换(AD)

1、模数转换的过程

步骤1：抗混叠滤波器

用于去除高频噪声

步骤2：采样保持

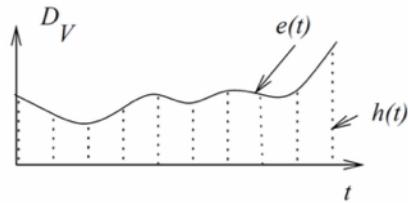
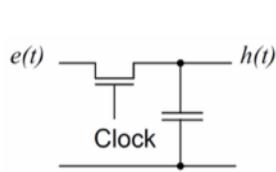
意义：计算机不能处理模拟信号

器件：由时钟驱动的晶体管 + 电容

晶体管由时钟驱动起到开关电路的作用，也就是对信号进行采样

电容可以保存采样后的电平，也就是起到保持的作用

电路：



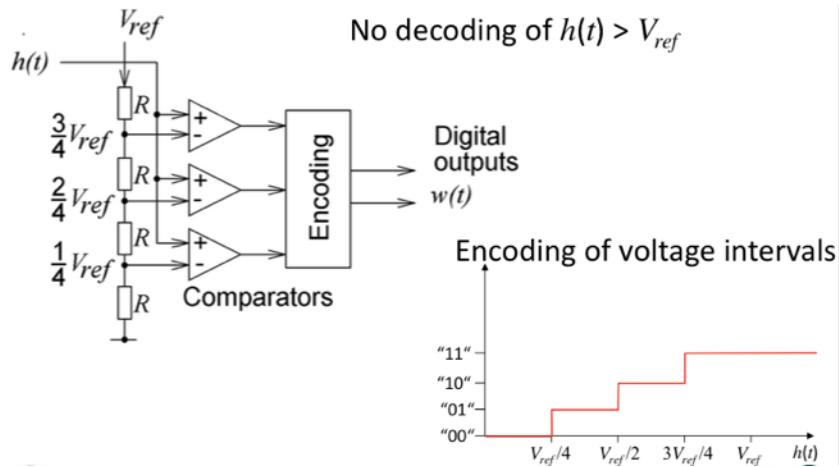
备注：根据奈奎斯特采样定理采样频率应大于原信号最大频率的两倍，否则不可恢复

步骤3：模数转换

2、模数转换的硬件实现

方法1：通过电阻，比较器和编码器进行实现

电路：



参数：

- 1、比特分辨率：最终编码出来的位数，单位：bit
- 2、电平分辨率(Q)：最后得到的一个电压区间上界和下界的差值
- 3、转换区间(V_{FSR})：可转换的电压区间
- 4、转换区间数(n)：转换后的电平区间数

$$Q = \frac{V_{FSR}}{n}$$

分析：

时间复杂度： $O(1)$

硬件复杂度： $O(n)$

方法2：利用二分的思想进行实现

从高位开始一位一位地往低位试值

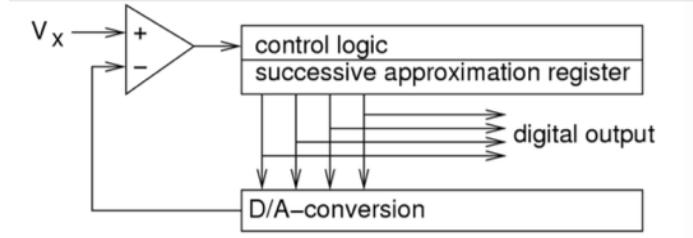
设置当前最高位为1

比较当前电平和输入电平

如果当前电平大于输入电平，将这一位设为0

将下一位看成最高位继续试值直到试到最低位

电路：



分析：

时间复杂度： $O(\log_2(n))$

硬件复杂度： $O(\log_2(n))$

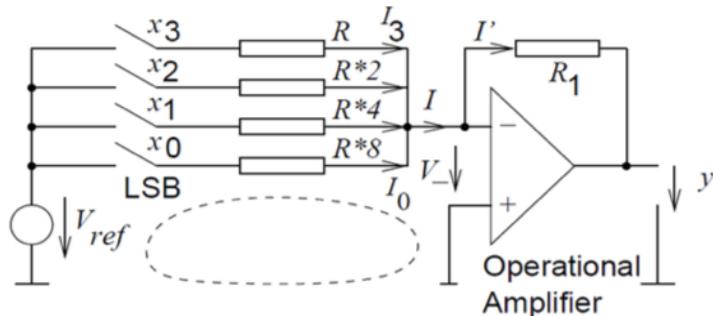
3、信噪比计算

$$\text{signal to noise ratio(SNR)} = 20 * \log_{10} \left(\frac{\text{effective signal voltage}}{\text{effective noise voltage}} \right) (\text{dB})$$

9-2、数模转换(DA)

1、数模转换的过程

步骤1：数模转换



步骤2：通过低通滤波器恢复模拟波形

第六部分、嵌入式系统分析

Embedded System Analysis

10、实时调度(Real-Time Scheduling)

10-1、实时模型(Real-Time Model)

1、硬实时系统与软实时系统

硬实时系统：在硬实时系统中如果一个任务在截止时间内没有完成就会产生灾难性后果

软实时系统：在软实时系统中如果一个任务在截止时间内没有完成会有些许影响，但不会对系统的正常行为与外界环境造成严重危害

2、调度

调度：调度是指规划给到各个任务的处理器时间以使得每个任务可以被执行到完成

调度可以用函数 $\sigma(t)$ 表示，函数 $\sigma(t)$ 说明在 t 时刻处理器正在执行的是哪个任务

当 $\sigma(t)=0$ 时，说明处理器没有在执行任务，处于空闲状态

当 σ 的值发生改变时说明处理器正在变更正在处理的任务，说明正在进行上下文切换

在处理器时间片中 σ 的值不发生改变

抢占型调度：在抢占型调度中正在处理器内执行的任务随时可能被挂起（中断在处理器内的执行），然后分派另一个任务到处理器中

可行调度：一个可行的调度是指所有任务都能在满足给定的约束条件下执行完毕

任务可调度：对于一个可调度的任务集来说必定存在一种算法可以产生至少一种可行调度

3、时序约束

常见的时序约束

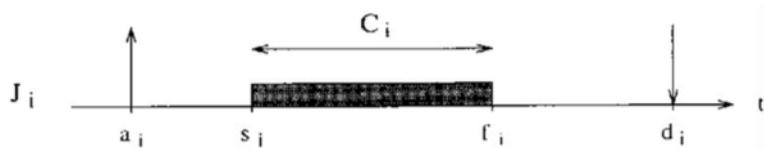
到来时间 a_i （释放时间 r ）：说明任务什么时候就绪执行（任务已经创建好，具备执行能力）

计算时间 C_i ：一个任务从开始执行到完成真正占有处理器的时间

截止时间 d_i ：限定任务必须在这个时刻前完成

开始时间 s_i ：任务第一次进入处理器执行的时间

结束时间 f_i ：任务结束执行的时间



根据上面的时序约束我们还可以得出下面的结论

截止时间 d_i 小于释放时间 r 和计算时间 C_i 之和

延迟时间 L_i ：为结束时间 f_i 和截止时间 d_i 的差，表示任务的延迟

如果在截止时间前结束任务延迟时间为负值

溢出时间 E_i ：为延迟时间 L_i 和0的最大值，同样也有表示延迟的意思

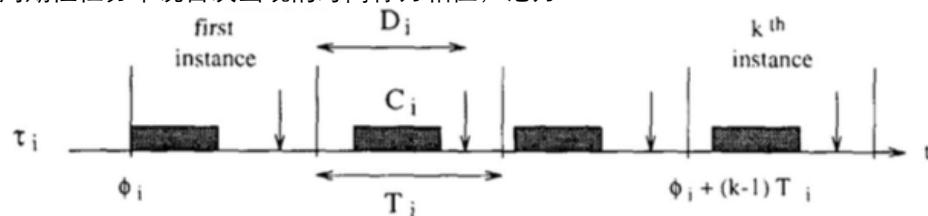
如果在截止时间前结束任务延迟时间为0

松弛时间 X_i ：为截止时间 d_i 减去到来时间 a_i 后的结果再减去计算时间 C_i ，说明要求任务在截止时间前完成的情况下，这个任务一开始可以拖延多长时间开始执行（反过来说，如果任务在松弛时间 X_i 之内还没有进入过处理器，那这个任务就不可能在截止时间前完成）

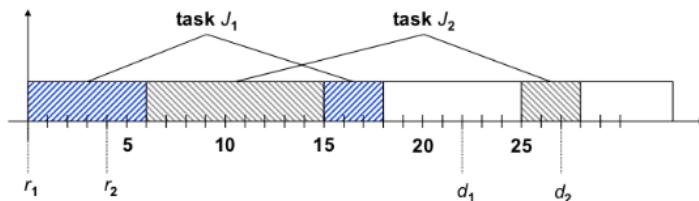
4、周期性任务

周期性任务 τ_i ：每隔一个时间周期 T_i 就会出现一次的任务

对于周期性任务来说首次出现的时间称为相位，记为 ϕ_i



例子：有关时序约束的计算



Computation times: $C_1 = 9, C_2 = 12$

Start times: $s_1 = 0, s_2 = 6$

Finishing times: $f_1 = 18, f_2 = 22$

Lateness: $L_1 = -4, L_2 = 1$

Tardiness: $E_1 = 0, E_2 = 1$

Laxity: $X_1 = 13, X_2 = 11$

5、优先级约束

优先级关系可以通过有向图加以表示

节点表示事件

依赖关系用带箭头的线表达

$A \rightarrow B$: 说明A应该先于B被执行

最后通过拓扑排序就可以得到对应的事件执行顺序

6、调度算法的分类

抢占型调度算法：在抢占型调度中正在处理器内执行的任务随时可能被挂起（中断在处理器内的执行），然后分派另一个任务到处理器中

非抢占型调度算法：在非抢占型调度中任务一旦进入处理器中执行就不会被中断，直至其执行完毕才会分派另一个任务到处理器中

静态调度算法：基于固定参数的调度决策（调度决策不能根据任务的实际情况进行自适应调节）

动态调度算法：基于非固定参数的调度决策（调度决策能根据任务的实际情况进行自适应调节）

最优调度算法：在某一指标下为最优的调度算法

启发式调度算法：随着迭代次数的增多会不断趋近某一指标（达不到最优，只能趋近）下最优的调度算法

7、调度算法的常见衡量指标

平均响应时间：每个任务的响应时间的平均值

响应时间：结束时间 f_i 和释放时间 r_i 的差

$$\bar{t}_r = \frac{1}{n} \sum_{i=1}^n (f_i - r_i)$$

总完成时间：系统为完成这些任务总共启动了多长时间

总完成时间为最晚完成的任务的完成时间和最早到来的任务的释放时间的差

$$t_c = \max_i(f_i) - \min_i(r_i)$$

响应时间的加权和：每个任务的响应时间经加权后的值

$$t_w = \frac{\sum_{i=1}^n w_i (f_i - r_i)}{\sum_{i=1}^n w_i}$$

最长延迟：所有任务的延迟时间的最大值

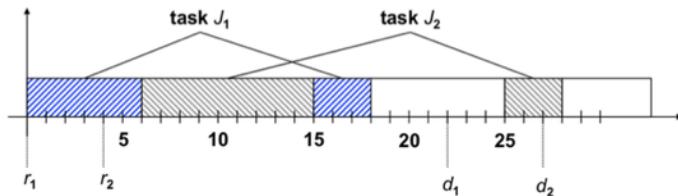
$$L_{\max} = \max_i(f_i - d_i)$$

延迟任务数：说明有多少个任务被延迟（没有在截止时间前完成）

$$N_{late} = \sum_{i=1}^n miss(f_i)$$

$$miss(f_i) = \begin{cases} 0 & \text{if } f_i < d_i \\ 1 & \text{otherwise} \end{cases}$$

例子：有关常见衡量指标的计算



$$\text{Average response time: } \bar{t}_r = \frac{1}{2}(18+24) = 21$$

$$\text{Total completion time: } t_c = 28 - 0 = 28$$

$$\text{Weighted sum of response times: } w_1 = 2, w_2 = 1: t_w = \frac{2 \cdot 18 + 24}{3} = 20$$

$$\text{Number of late tasks: } N_{late} = 1$$

$$\text{Maximum lateness: } L_{max} = 1$$

10-2、周期性任务的经典实时调度(Classical real-time scheduling for Periodic Tasks)

10-2-1、固定周期速率调度(Periodic Rate Monotonic (RM))

1、对周期性任务建模

对于周期性任务来说需要满足下面的前提

前提1：任务*i*每隔T*i*时间到来一次

因此任务*i*第j次的到来时间为：

$$r_{ij} = \Phi_i + (j - 1)T_i$$

前提2：每个周期里面的任务*i*的计算时间C*i*完全一致

前提3：每个周期里面的任务*i*的相对截止时间D*i*完全一致（每个周期的任务*i*到来后必须要在时间D*i*内完成）

因此任务*i*第j次的截止时间为：

$$d_{ij} = \Phi_i + (j - 1)T_i + D_i$$

而在很多情况下周期T和相对截止时间D相同，因此又有

$$d_{ij} = \Phi_i + jT_i$$

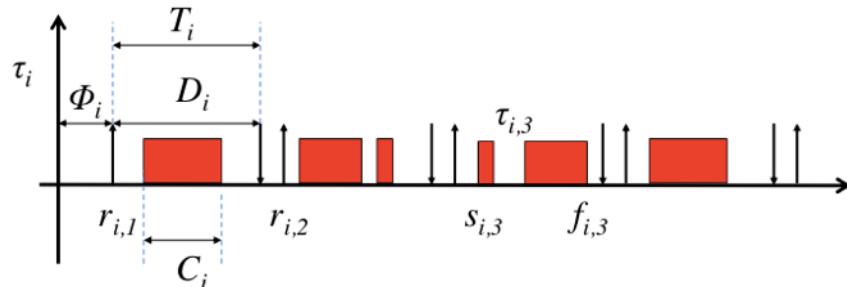
前提4：每个周期相互独立（上一个周期的任务的执行不会对下一个周期任务的执行造成影响）

前提5：没有一个任务会阻塞自己（不会进入等待状态等待事件发生）

前提6：任务到来后立即释放（到来时间a*i*和释放时间r*i*完全一致）

前提7：在操作系统内核中所有的开销都被假定为零（不考虑上下文切换等等的其他开销）

例子：



2、固定周期速率调度

前提假设：

静态优先级调度：每个任务的优先级已经事先指定好，不会发生更改

抢占型调度：高优先级任务会立即抢占低优先级任务

周期 T_i 和相对截止时间 D_i 相同

算法描述：

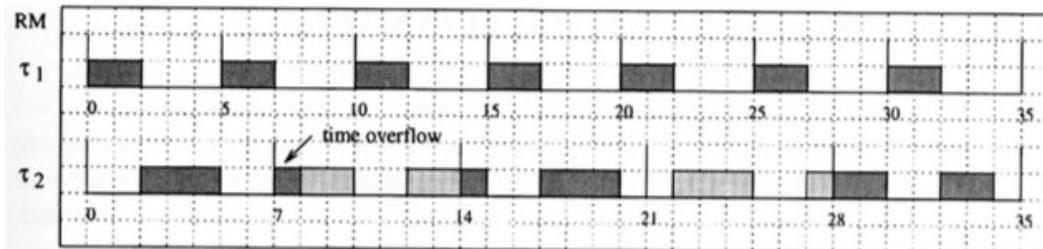
每个任务都被事先赋予一个优先级

优先级不需要人为制定，而是系统根据每个任务的周期 T_i 进行确定

周期 T_i 越短的任务优先级越高

高优先级的任务总是抢占低优先级的任务

例子：



任务1的周期是5，任务2的周期是7，因此任务1的优先级高于任务2的优先级，任务1总是抢占任务2的执行
评价：

这个算法是最优的：对于固定优先级的情况下如果这个算法不能调度其他算法也不能（证明略）

3、固定周期速率调度的可调度性分析

处理器利用率(U)：说明处理器用于执行这个任务集的用时占整个处理器时间的比重

计算方法：所有任务单周期内的计算时间除以所有任务的周期

$$U = \sum_{i=1}^n \frac{C_i}{T_i}$$

引理：当满足下列式子时任务集是可调度的（证明略）

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

注：不满足这个式子不表明任务集不可调度

10-2-2、最早截止时间优先调度(Earliest Deadline First (EDF))

1、最早截止时间优先调度

前提假设：

动态优先级调度：每个任务的优先级会根据实际情况自适应变化

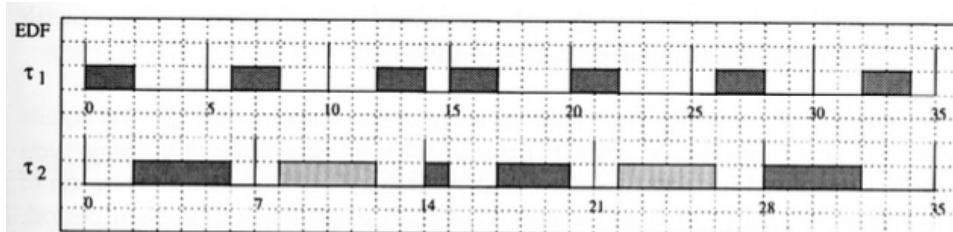
抢占型调度：高优先级任务会立即抢占低优先级任务

相对截止时间 D_i 小于等于周期 T_i

算法描述：

当前执行的任务会被更早截止的任务所抢占（每次在处理器中运行的总是当前最早截止的任务）

例子：



每次在处理器中运行的都是当前最早截止的任务

评价：

这个算法是最优的：对于周期性任务集的情况下如果这个算法不能调度其他算法也不能（证明略）

2、最早截止时间优先调度的可调度性分析

平均处理器利用率(U)：说明处理器用于执行这个任务集的用时占整个处理器时间的比重

计算方法：所有任务单周期内的计算时间除以所有任务的周期

$$U = \sum_{i=1}^n \frac{C_i}{T_i}$$

引理：当满足下列式子时任务集是可调度的（证明略）

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$

10-3、混合任务集(Mixed Task Sets)

10-3-1、固定周期速率调度下的轮询服务器(RM - Polling Server)

1、什么是混合任务集

在很多应用中不全是周期性任务集，实际上有很多的非周期性任务

周期性任务：由时间所驱动，通过硬件保证固定的周期速率

非周期性任务：由事件所驱动，根据具体应用同样会有硬、软、非实时性的要求

散发的任务：

对于非周期任务来说要想对时序进行约束只能通过对环境加以假设

也就是说我们需要把非周期性任务转变为“周期性任务”

我们可以用非周期性任务的最短时间间隔作为它的周期

另一种说法是：用非周期性任务的最大出现频率作为它的频率

经过这样处理以后得到的“周期性任务”称为散发的任务

2、后台调度：一种解决非周期性任务调度问题的方法

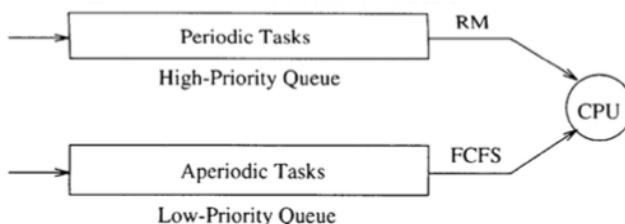
背景：RM调度方法和EDF调度方法只针对周期性调度

思想：利用处理器的间隙（处理周期性调度以外的空闲时间）来处理非周期性任务

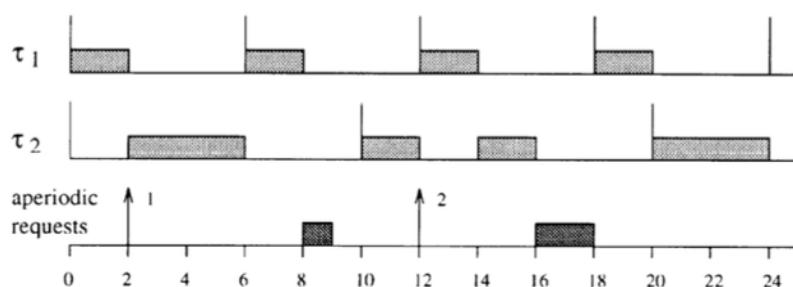
不影响周期性任务的正常调度

但非周期性任务的响应时间可能非常长，而且无法通过给它们赋予优先级解决

为不影响周期性任务，只能把非周期性任务的优先级调到最低，并采用FIFO的调度方法



例子：



3. 固定周期速率调度下的轮询服务器

背景：RM调度方法和EDF调度方法只针对周期性调度

思想：另外引入一个周期性任务，这个周期性任务的工作是检查是否有非周期性的任务请求

这个新引入的周期性任务和其它周期性任务一样具有周期 T_s 和计算时间 C_s

这个新引入的周期性任务的调度方式也和其它周期性任务一样

不过当轮到这个新引入的周期性任务时实际执行的却是非周期性的任务请求

非周期性任务的优先级可以通过这个新引入的周期性任务完成定义

由于在固定周期速率调度下优先级根据周期大小确定

因此我们通过设定这个新引入的周期性任务的周期即可完成非周期性任务优先级的定义

这个新引入的周期性任务称为轮询服务器

轮询服务器的功能：

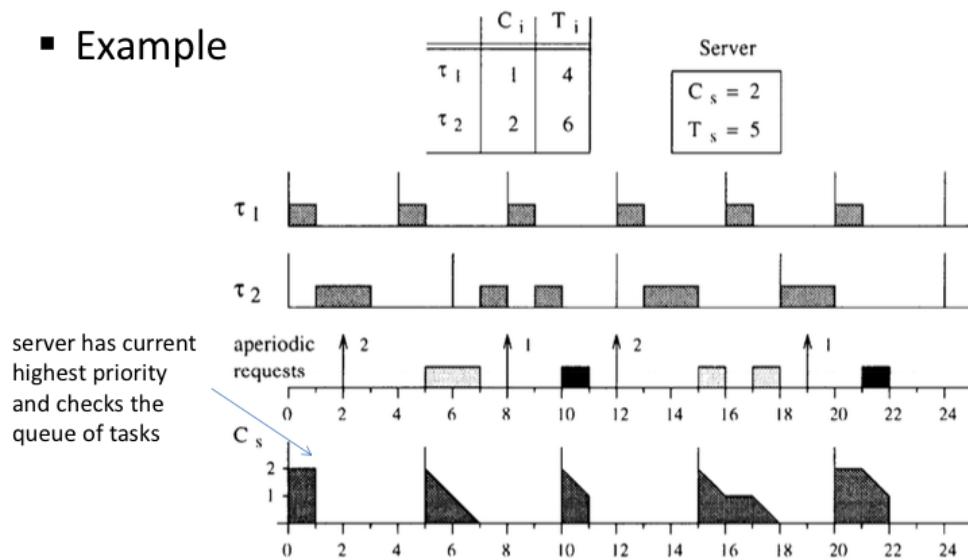
1、对于一个轮询服务器来说有两个参数最重要，一个是周期 T_s ，另一个是计算时间 C_s 。当我们进行轮询服务器的实例化时需要指定这两个参数。周期 T_s 的实际含义是轮询的频率，这个频率同时也与非周期性任务的优先级密切相关；而计算时间 C_s 实际上代表的是最多可容纳的非周期性任务的计算时间

2、当处理器分派到轮询服务器时，轮询服务器会检查任务队列里面有没非周期性任务，如果有就立即将那个任务调度进处理器中执行；如果没有，轮询服务器会暂时将自己切换到等待状态，等到下一个周期的时候再唤醒进入就绪队列中等待被分配处理器（因此在轮询服务器切换到等待状态的这段时间内即使有非周期性任务也不能执行，必须至少要等到下一个周期才可以执行）

轮询服务器的不足之处：在轮询服务器切换到等待状态的这段时间内即使有非周期性任务也不能执行，必须至少要等到下一个周期才可以执行

例子：

■ Example



在这个例子中周期 T_s 的值为5，因此在途中可以看到每隔5个时间片轮询服务器就重新启动进入就绪队列（重新启动的含义是将计算时间 C_s 重置（计算时间 C_s 实际上代表的是最多可容纳的非周期性任务的计算时间，在这例子中的意思是每5个时间片最多分配2个时间片给非周期性任务），并进入就绪队列中等待被分配处理器）。

此时的调度方式仍是根据周期进行优先级设置，周期越短的优先级越高。因此优先执行 τ_1 ，然后是非周期性任务，最后是 τ_2 。

当处理器分派到轮询服务器时，轮询服务器会检查任务队列里面有没非周期性任务，如果有就立即将那个任务调度进处理器中执行；如果没有，轮询服务器会暂时将自己切换到等待状态，等到下一个周期的时候再唤醒进入就绪队列中等待被分配处理器。比如说1时刻，处理器分派到轮询服务器了，此时发现并没有非周期性任务未处理，因此将自己切换到等待状态，直到下一个周期（5时刻）的时候再唤醒进入就绪队列中等待被分配处理器。因此虽然2时刻有非周期性任务进来，但由于此时轮询服务器已经处于等待状态，因此这个任务必须等到轮询服务器切换到就绪状态（5时刻）以后才可能被执行

4、引入轮询服务器后的固定周期速率调度的可调度性分析

这个的分析方法和引入轮询服务器前的固定周期速率调度的可调度性分析相类似
当满足下列式子时任务集是可调度的（证明略）

$$U = \frac{C_s}{T_s} + \sum_{i=1}^n \frac{C_i}{T_i} \leq (n + I)(2^{I/(n+I)} - 1)$$

注：不满足这个式子不表明任务集不可调度

5、非周期性分析

非周期性任务必须在轮询服务器处于就绪状态时才可能被调度

下面这个式子将保证一定能够在非周期性任务的截止时间到来前完成这项非周期性任务

$$\left(I + \left\lceil \frac{C_a}{C_s} \right\rceil \right) T_s \leq D_a$$

注：这个式子基于一个前提：在上一个非周期性任务完成前下一个不会进来（非周期性任务不会积累）

注：不满足这个式子不表明非周期性任务的截止时间到来前不能完成这项非周期性任务

10-3-2、最早截止时间优先调度下的总带宽服务器(EDF – Total Bandwidth Server)

1、什么是总带宽服务器

总带宽服务器负责给到来的非周期性任务分配优先级

假设第k个非周期性任务在时间t = rk时刻到来，它将得到一个截止时间（因为这个算法是根据截止时间进行调度优先级分配的）

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

其中Ck为这个非周期性任务的计算时间，Us为这个服务器的处理器利用率，而且默认d0为0
由上式可以推出Us为：

$$U_s = \frac{C_k}{d_k - \max(r_k, d_{k-1})}$$

当非周期性任务拿到截止时间以后就会进入就绪队列中等待被调度了

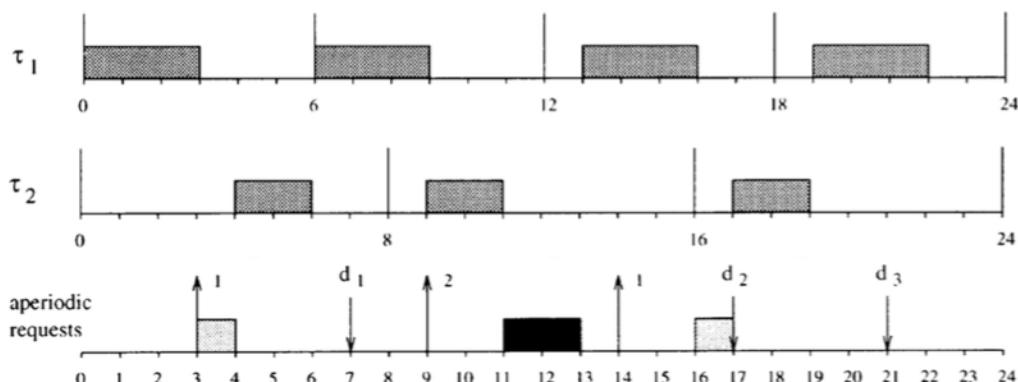
2、引入总带宽服务器后的最早截止时间优先调度的可调度性分析

这个的分析方法和引入总带宽服务器前的最早截止时间优先调度的可调度性分析相类似

当满足下列式子时任务集是可调度的（证明略）

$$U_p + U_s \leq 1$$

例子（给定Us为0.25）：



首先我们分析一下可调度性

$$U_p + U_s = \sum_{i=1}^n \frac{C_i}{T_i} + U_s = \frac{3}{6} + \frac{2}{8} + 0.25 \leq 1$$

因此是可以调度的

第一个任务在 $t = r_3$ 时刻到来时，它的截止时间计算为

$$d_1 = \max(r_1, d_0) + \frac{C_1}{U_s} = \max(3, 0) + \frac{1}{0.25} = 7$$

第二个任务在 $t = r_9$ 时刻到来时，它的截止时间计算为

$$d_2 = \max(r_2, d_1) + \frac{C_2}{U_s} = \max(9, 7) + \frac{2}{0.25} = 17$$

第三个任务在 $t = r_{14}$ 时刻到来时，它的截止时间计算为

$$d_3 = \max(r_3, d_2) + \frac{C_3}{U_s} = \max(14, 17) + \frac{1}{0.25} = 21$$

而且保证每次在处理器中运行的都是当前最早截止的任务

10-4、例题(Exercises)

1、最早截止时间优先调度的可调度性分析

假设给定下面这个任务集 T_1 、 T_2 、 T_3 的计算时间 C_i 和周期 T_i ，请解决下列问题：

	τ_1	τ_2	τ_3
C_i	6	5	1
T_i	9	15	5

(1)、验证这个任务集在最早截止时间优先调度下的可调度性

(2)、假设周期性任务 T_1 、 T_2 、 T_3 的第一个任务都在 0 时刻到来。请用图表表示这些任务在最早截止时间优先调度下的调度情况。并说明其中出现的可能的延迟现象

(3)、如果延迟现象在某一时刻出现在某一任务上，那么这个任务在之后的时间是否一直会出现延迟现象。请说明并验证你的观点。

分析：

(1)、对于最早截止时间调度来说当满足下列式子时任务集是可调度的

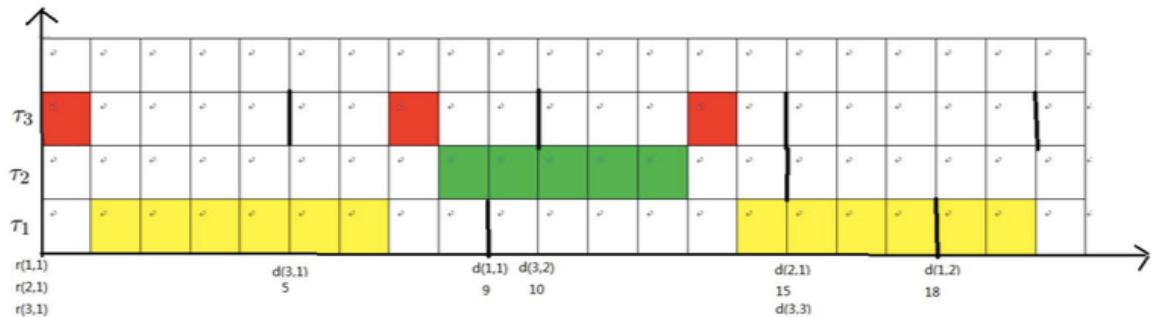
$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$

在这一题目中平均处理器利用率满足

$$U = \sum_{i=1}^n \frac{C_i}{T_i} = \frac{6}{9} + \frac{5}{15} + \frac{1}{5} = \frac{6}{5} > 1$$

由于这个条件是充要条件，因此这个任务集在最早截止时间优先调度下是不可调度的

(2) 在最早截止时间调度中每次在处理器中运行的都是当前最早截止的任务



从图中可以看出任务 τ_1 和任务 τ_3 都出现了延迟

(3) 如果延迟现象在某一时刻出现在某一任务上，那么这个任务在之后的时间并不一直会出现延迟现象

就拿任务 τ_3 作为反例，虽然在 $t=15$ 时到来的周期性任务 τ_3 出现了延迟（在 $t=20$ 时还没有完成）

但是在 $t=20$ 时，此时任务 τ_1 在 $t=27$ 时截止，任务 τ_2 在 $t=30$ 时截止，任务 τ_3 在 $t=20$ 时截止

因此 $t=20 \sim t=21$ 这段时间执行的是任务 τ_3

在 $t=21$ 时，此时任务 τ_1 在 $t=27$ 时截止，任务 τ_2 在 $t=30$ 时截止，任务 τ_3 在 $t=25$ 时截止

因此 $t=21 \sim t=22$ 这段时间执行的还是任务 τ_3

这就保证了在 $t=20$ 时到来的任务 τ_3 不会出现延迟

2、固定周期速率调度的可调度性分析

假设给定下面这个任务集 T_1 、 T_2 、 T_3 的计算时间 C_i 和周期 T_i ，请解决下列问题：

	τ_1	τ_2	τ_3
C_i	1	3	2
T_i	3	8	9

(1) 验证这个任务集在固定周期速率调度下的可调度性

(2) 假设周期性任务 τ_1 、 τ_2 、 τ_3 的第一个任务都在0时刻到来。请用图表表示这些任务在固定周期速率调度下的调度情况。并说明其中可能出现的可能的延迟现象

分析：

(1) 对于固定周期速率调度来说当满足下列式子时任务集是可调度的

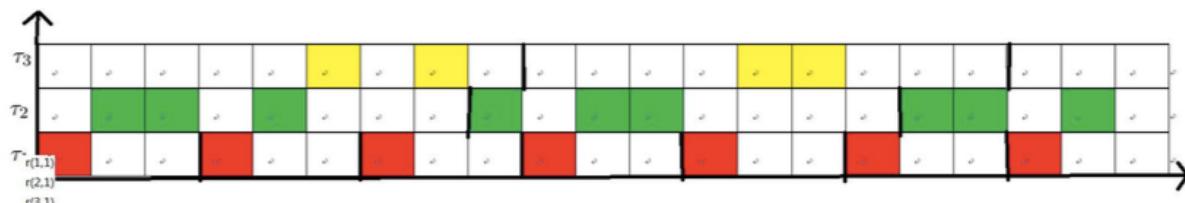
$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$

在这一题目中平均处理器利用率满足

$$U = \sum_{i=1}^n \frac{C_i}{T_i} = \frac{1}{3} + \frac{3}{8} + \frac{2}{9} = \frac{67}{72} > 3(\sqrt[3]{2} - 1)$$

由于这个条件是充分条件，并不是必要条件，因此无法通过判断可调度性

(2) 在固定周期速率调度中每次优先执行周期短的任务（在这里优先执行任务 τ_1 ，其次 τ_2 ，再次 τ_3 ）



在这里并没有出现延迟现象

3、引入轮询服务器的固定周期速率调度的可调度性分析

假设给定下面这个任务集 T_1 、 T_2 、 T_3 的计算时间 C_i ，截止时间 D_i 和周期 T_i

	τ_1	τ_2	τ_3
C_i	2	2	2
D_i	6	8	16
T_i	6	8	16

在上述的周期性任务外我们还引入了一个非周期性任务 J_a ，任务 J_a 的计算时间 C_a 为1，并用 D_a 表示它的截止时间。假设我们引入了轮询服务器来解决非周期性任务的调度问题，并以固定周期速率调度的调度方式进行调度。请基于上述条件解决下列问题：

- (1)、假设轮询服务器的周期 T_s 为25，计算时间 C_s 为1。请说明当这个非周期性任务的截止时间 D_a 满足什么条件时能保证它一定不出现延迟
- (2)、在(1)的基础上验证引入轮询服务器的固定周期速率调度的可调度性
- (3)、确定轮询服务器的周期 T_s 和计算时间 C_s ，以保证可调度性的前提下使得任务 J_a 的截止时间 D_a 尽量提前
- (4)、在(3)的基础上假设非周期性任务的截止时间 D_a 为32，说明这个非周期性任务是否会出现延迟

分析（我对这道例题的后三问的答案不是很认同，下面说的是我的想法...）

(1)、对于非周期性任务来说下面这个式子将保证一定能够在非周期性任务的截止时间到来前完成这项非周期性任务

$$\left(1 + \left\lceil \frac{C_a}{C_s} \right\rceil\right)T_s \leq D_a$$

因此对于这个问题为

$$\left(1 + \left\lceil \frac{C_a}{C_s} \right\rceil\right)T_s = \left(1 + \left\lceil \frac{1}{1} \right\rceil\right) \times 25 = 50 \leq D_a$$

(2)、当满足下列式子时任务集是可调度的

$$U = \frac{C_s}{T_s} + \sum_{i=1}^n \frac{C_i}{T_i} \leq (n + I)(2^{I/(n+I)} - 1)$$

因此对于这个问题为

$$U = \frac{C_s}{T_s} + \sum_{i=1}^n \frac{C_i}{T_i} = \frac{1}{25} + \frac{2}{6} + \frac{2}{8} + \frac{2}{16} \leq 4 \times (2^{1/4} - 1)$$

因此这个任务集是确定可以调度的

(3)、因为如果要保证非周期性任务的截止时间到来前完成就必须满足

$$\left(1 + \left\lceil \frac{C_a}{C_s} \right\rceil\right)T_s \leq D_a$$

而其中 J_a 的计算时间 C_a 是不可改变的，因此我们要想办法提升轮询服务器的计算时间 C_s 和减少周期 T_s 。其中由于轮询服务器的计算时间 C_s 必须要是大于0的正值，而计算时间 C_a 是1，因此不论计算时间 C_s 为何值，下式一定成立

$$\left\lceil \frac{C_a}{C_s} \right\rceil = 1 \Rightarrow 2T_s \leq D_a$$

而轮询服务器的周期 T_s 又受到下面式子的限制

$$U = \frac{C_s}{T_s} + \sum_{i=1}^n \frac{C_i}{T_i} \leq (n+1)(2^{1/(n+1)} - 1)$$

其中又需要轮询服务器的计算时间 C_s 尽可能小以换取尽可能小的轮询服务器的周期 T_s

因此 C_s 取最小值1，此时可知：

$$U = \frac{C_s}{T_s} + \sum_{i=1}^n \frac{C_i}{T_i} = \frac{1}{T_s} + \frac{2}{6} + \frac{2}{8} + \frac{2}{16} \leq 4 \times (2^{1/4} - 1)$$

解得：

$$T_s > 20.62$$

因此取周期 T_s 为21，此时可以得到最小的截止时间 D_a 为42

(4)、由于上面我们是基于充分条件进行分析的，因此截止时间 D_a 小于42时不一定不可以调度
在这里假设的是截止时间 D_a 为32，因此我们查看前面32个周期的调度情况

发现在32个周期内，任务 τ_1 出现6次，因此需要计算时间为12；任务 τ_2 出现4次，因此需要计算时间为8；任务 τ_3 出现1次，因此需要计算时间为2，总共的计算时间为22，留下了10个计算时间的空隙。由于我们这个任务只需要1个计算时间，因此可以满足要求

4、引入总带宽服务器后的最早截止时间优先调度的可调度性分析

假设引入了一个计算时间 C_a 为2，相对截止时间 d_a 为7的非周期性任务。问在保证调度性的前提下周期性任务平均处理器利用率最大为多少

分析：

假设第 k 个非周期性任务在时间 $t = r_k$ 时刻到来，它将得到一个截止时间（因为这个算法是根据截止时间进行调度优先级分配的）

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

其中 C_k 为这个非周期性任务的计算时间， U_s 为这个服务器的处理器利用率，而且默认 d_0 为0
由上式可以推出 U_s 为：

$$U_s = \frac{C_k}{d_k - \max(r_k, d_{k-1})}$$

而在这个例子中时间 $t = r_0$ 时刻到来，默认 d_0 为0，因此可得：

$$U_s = \frac{2}{7 - \max(0, 0)} = \frac{2}{7}$$

由于满足下列式子时任务集是可调度的

$$U_p + U_s \leq 1$$

因此可得：

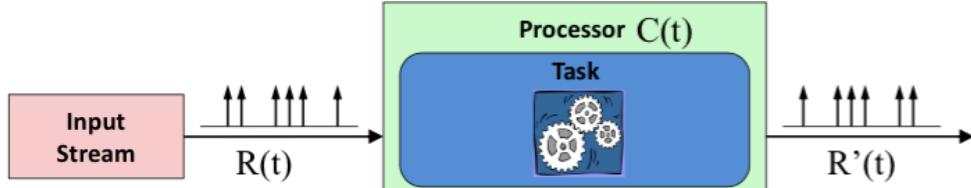
$$U_p \leq 1 - U_s = \frac{5}{7} \Rightarrow \max(U_p) = \frac{5}{7}$$

11、对于实时性的数学分析(Mathematical Analysis on Real-Time)

11-1、实时性演算(Real-Time Calculus)

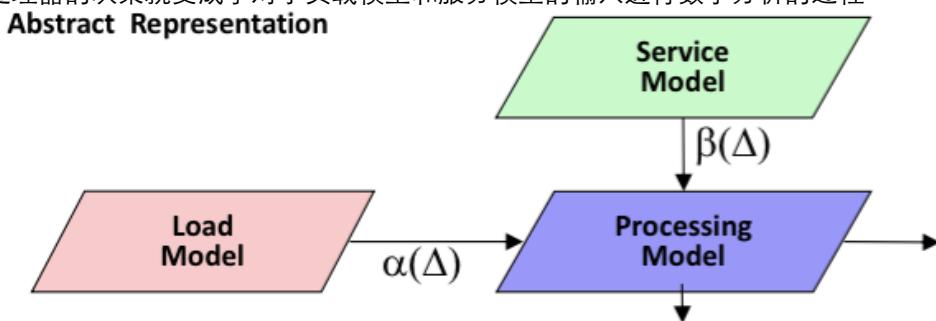
1、性能分析的数学模型

在具体的实例中我们将事件流交到处理器中进行调度决策最终得到实际的任务执行序列



而现在我们需要对实时性能进行数学上的分析就必须要进行分析建模

因此我们将事件流抽取出来成为负载模型，将系统资源也抽象出来成为服务模型，处理器也抽象出来成为处理模型。处理器的决策就变成了对于负载模型和服务模型的输入进行数学分析的过程



2、事件流模型：对于输入事件的建模

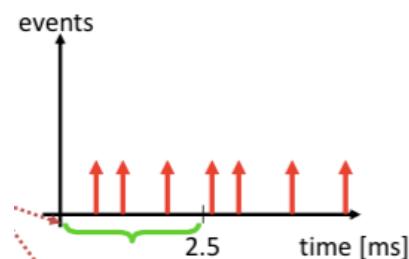
建模目标：能尽量全方位地表示事件流的特征

建模方法：

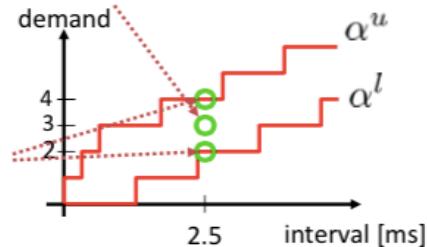
- 1、设定一个时间的滑动窗口，这个窗口范围为 $[0, \infty)$
- 2、将这个滑动窗口在事件流的时域范围 $[0, \infty)$ 上扫过，记录下滑动窗口扫描范围下的任务数量
 记录下滑动窗口的扫描到的任务数目最大值 α^u
 记录下滑动窗口的扫描到的任务数目最小值 α^l

- 3、绘制滑动窗口—任务数目曲线

例子：假设有下面这个事件流



我们将时间窗从0开始变化直至 ∞ ，对于其中每个时间窗都在 $[0, \infty)$ 上扫过整个时间轴，并记录下每次扫描范围下的任务数量，最后得出最大值和最小值并绘制在滑动窗口—任务数目曲线上即可得到：



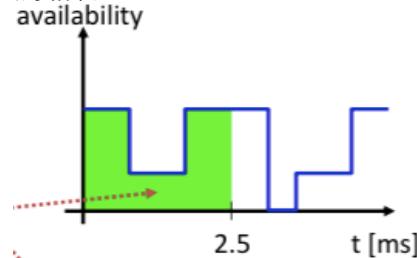
3、服务模型：对于系统资源的建模

建模目标：能尽量全方位地表示系统资源的特征
建模方法：

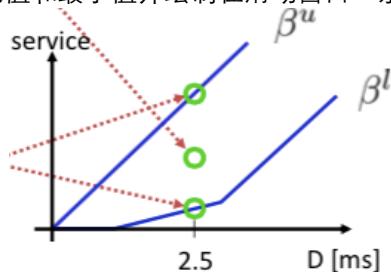
- 1、设定一个时间的滑动窗口，这个窗口范围为 $[0, \infty)$
- 2、将这个滑动窗口在事件流的时域范围 $[0, \infty)$ 上扫过，记录下扫描范围下的系统资源量
记录下扫描到的系统资源量最大值 β^u
记录下扫描到的系统资源量最小值 β^l

3、绘制滑动窗口—系统资源曲线

例子：假设有下面这个可利用系统资源情况



我们将时间窗从0开始变化直至 ∞ ，对于其中每个时间窗都在 $[0, \infty)$ 上扫过整个时间轴，并记录下每次扫描范围下的系统资源量，最后得出最大值和最小值并绘制在滑动窗口—系统资源曲线上即可得到：



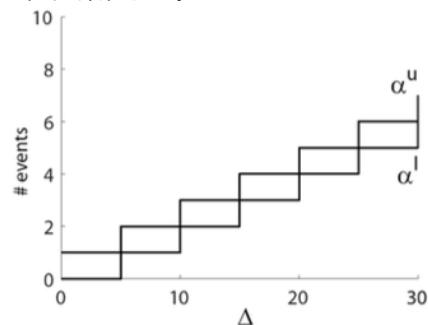
4、对周期性事件流建模： (p, j, d) 模型

对于周期性事件流来说最重要的就是周期 p ，按照上面的事件流建模方法，对于周期性事件来说

当滑动窗口大小为 $0+$ 时：最小任务数 α^l 为0，最大任务数 α^u 为1

滑动窗口每增大周期 p 的大小，最小任务数 α^l 和最大任务数 α^u 均加上1

下图是周期 p 为5的时候的滑动窗口—任务数目曲线：



但是在实际情况下往往不会这样，因为种种因素，任务到来时会出现一定程度的抖动现象。一般来说我们假设这个抖动时间为 j ，第 n 个任务实际到来时刻为 $(n-1)p \pm (j/2)$

因此对于存在抖动的周期性事件来说和原来的无抖动的周期性事件的滑动窗口—任务数目曲线差异在于：

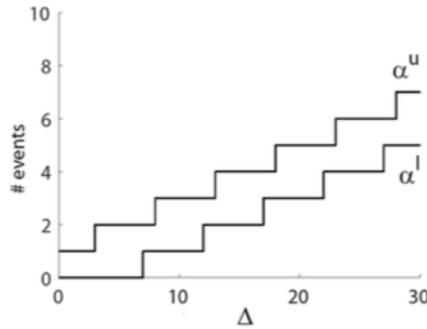
最小任务数 α^l 曲线沿着时间轴向右平移 j

时间窗口小于 $(np - j)$ 时都不能保证一定有 n 个任务进来

最大任务数 α^u 曲线沿着时间轴向左平移 j

时间窗口大于 $((n-1)p - j)$ 时就可能有 n 个任务进来

下图是周期 p 为5、抖动时间 j 为2的时候的滑动窗口—任务数目曲线：

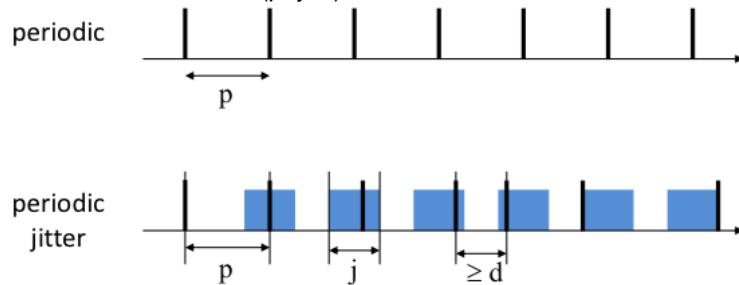


但是引进了抖动时间 j 以后又引发了另一个问题，如果任务周期 p 比抖动时间 j 要小会出现任务重叠的现象。由于我们只有一个处理器，因此在某一时刻只能执行一个任务，而且每个任务都是存在执行时间，因此相邻两个任务之间必须留下一段时间间隙。在这里我们假设这个时间间隙为 d ，而且保证这个时间间隙小于周期 p （否则就会产生任务的积累）。引入时间间隙 d 以后滑动窗口—任务数目曲线出现了一处小改动

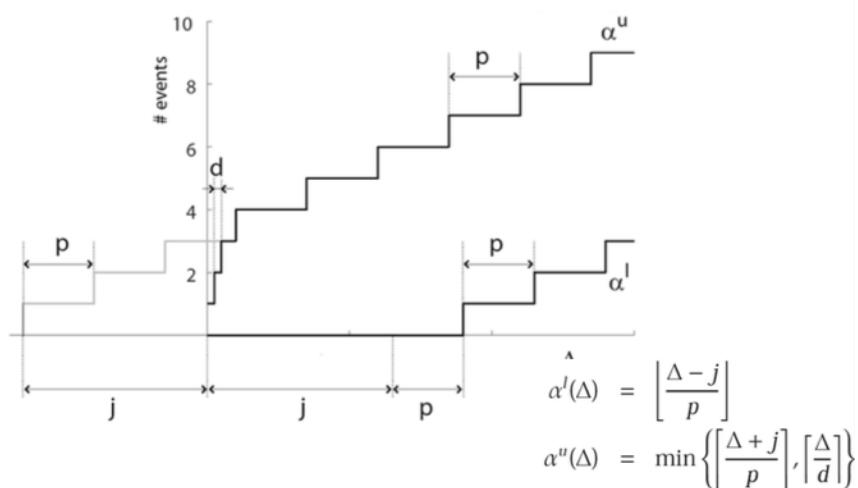
最大任务数 α^u 曲线初始时不得超过1

最大任务数 α^u 每次变化的时间间隔都不得小于 d

最终我们就得到了对于周期性事件流建模的 (p, j, d) 模型：



和最终的滑动窗口—任务数目曲线：



和最终的最小任务数 α^l 和最大任务数 α^u 的表达式

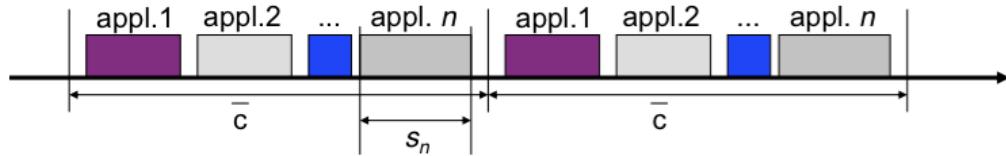
$$\alpha^l(\Delta) = \left\lfloor \frac{\Delta - j}{p} \right\rfloor$$

$$\alpha^u(\Delta) = \min \left\{ \left\lceil \frac{\Delta + j}{p} \right\rceil, \left\lceil \frac{\Delta}{d} \right\rceil \right\}$$

5、一种对资源的建模方法：TDMA时分复用模型

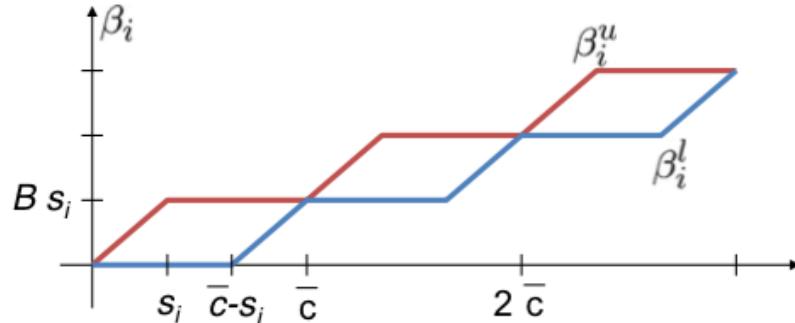
TDMA时分复用的含义：

假设有n个任务，这n个任务一共要用的计算时间设为c，每单位时间用到的资源量为B。在TDMA时分复用中采用轮转法周期性调度的方式，只在属于进程的时段，进程才可以利用系统资源，在其他时段这个进程是没有可利用的资源的。具体原理如下图所示：



因此在TDMA时分复用下对于某一任务i来说滑动窗口—系统资源曲线存在如下规律

最小可利用系统资源 β^l 曲线在 $(c - s_i)$ 之前为0， $(c - s_i) \sim c$ 时线性增至 Bs_i ，后一周期比前一周期增加1
最大可利用系统资源 β^u 曲线在 (s_i) 之前线性增至 Bs_i ， $(s_i) \sim c$ 时为 Bs_i ，后一周期比前一周期增加1



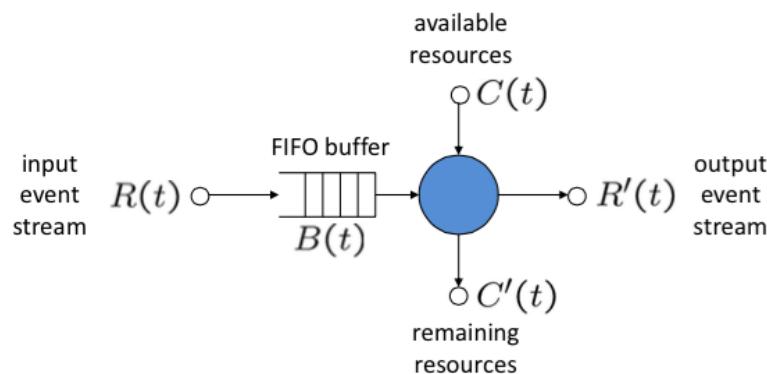
和最终的最小可利用系统资源 β^l 和最大可利用系统资源 β^u 的表达式

$$\beta_i^l(\Delta) = B \times \max \left\{ \left\lfloor \frac{\Delta}{c} \right\rfloor s_i, \Delta - \left\lfloor \frac{\Delta}{c} \right\rfloor (c - s_i) \right\}$$

$$\beta_i^u(\Delta) = B \times \min \left\{ \left\lceil \frac{\Delta}{c} \right\rceil s_i, \Delta - \left\lceil \frac{\Delta}{c} \right\rceil (c - s_i) \right\}$$

6、贪心调度处理（仅作了解）

调度行为：



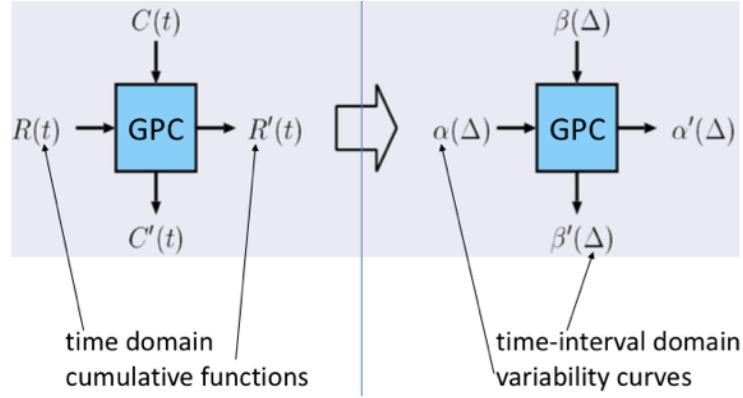
调度由输入事件驱动；每当一个输入事件进来都会实例化成为一个任务，任务通过贪心的原则进行调度（先到先服务：FIFO）；每个任务的执行通过可利用资源量进行限制。并且满足下列式子：

$$C(t) = C'(t) + R'(t)$$

$$B(t) = R(t) + R'(t)$$

$$R'(t) = \inf_{0 \leq u \leq t} \{R(u) + C(t) - C(u)\}$$

将输入的事件流和实际的系统资源变成我们上面可以分析的任务数曲线和可利用系统资源曲线就需要用到卷积等数学运算了



先定义一些用到的卷积运算：

最小加卷积：

$$(f \otimes g)(t) = \inf_{0 \leq u \leq t} \{f(t - u) + g(u)\}$$

最小加反卷积：

$$(f \oslash g)(t) = \sup_{0 \leq u \leq t} \{f(t + u) - g(u)\}$$

最大加卷积：

$$(f \overline{\otimes} g)(t) = \sup_{0 < u < t} \{f(t - u) + g(u)\}$$

最大加反卷积：

$$(f \overline{\oslash} g)(t) = \inf_{0 \leq u \leq t} \{f(t + u) - g(u)\}$$

任务数曲线和可利用系统资源曲线可以通过下列关系得到（证明略）

$$\alpha^l(t - s) \leq R(t) - R(s) \leq \alpha^u(t - s) \quad \forall s \leq t$$

$$\beta^l(t - s) \leq C(t) - C(s) \leq \beta^u(t - s) \quad \forall s \leq t$$

$$\alpha^u = R \oslash R; \quad \alpha^l = R \overline{\oslash} R; \quad \beta^u = C \oslash C; \quad \beta^l = C \overline{\oslash} C$$

同时还可以通过数学运算计算出决策后的剩余资源（证明略）

决策后事件流满足：

$$R'(t) \geq (R \otimes \beta^l)(t)$$

决策后最大任务数曲线满足：

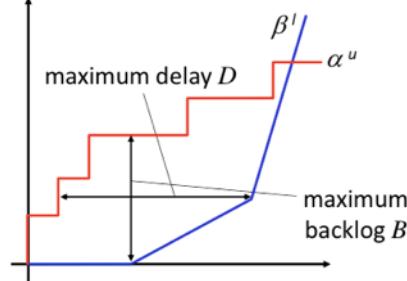
$$\alpha^{u'} = (\alpha^u \oslash \beta^l)$$

决策后最大可利用系统资源曲线满足：

$$\beta^{l'}(\Delta) = \sup_{0 \leq \lambda \leq \Delta} (\beta^l(\lambda) - \alpha^u(\lambda))$$

7、时序分析

进行时序分析时我们需要考虑最坏情况，因此我们需要考虑最大任务数曲线和最小可利用系统资源曲线



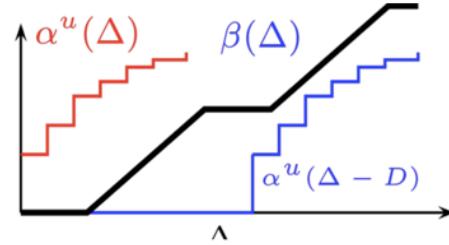
在这两条曲线中我们可以看出下面两点：

- D: 任务出现到处理需要延迟的时间
- B: 某一时刻任务还另外需要的资源量

其数学定义为：

$$\begin{aligned} B &= \sup_{t \geq 0} \{R(t) - R'(t)\} \leq \sup_{\lambda \geq 0} \{\alpha^u(\lambda) - \beta^l(\lambda)\} \\ D &= \sup_{t \geq 0} \{\inf\{\tau \geq 0 : R(t) \leq R(t + \tau)\}\} \\ &= \sup_{\Delta \geq 0} \{\inf\{\tau \geq 0 : \alpha^u(\Delta) \leq \beta^l(\Delta + \tau)\}\} \end{aligned}$$

当满足最小可利用系统资源曲线在延迟后的最大任务数曲线左侧时是可以调度的，比如下面这种情况

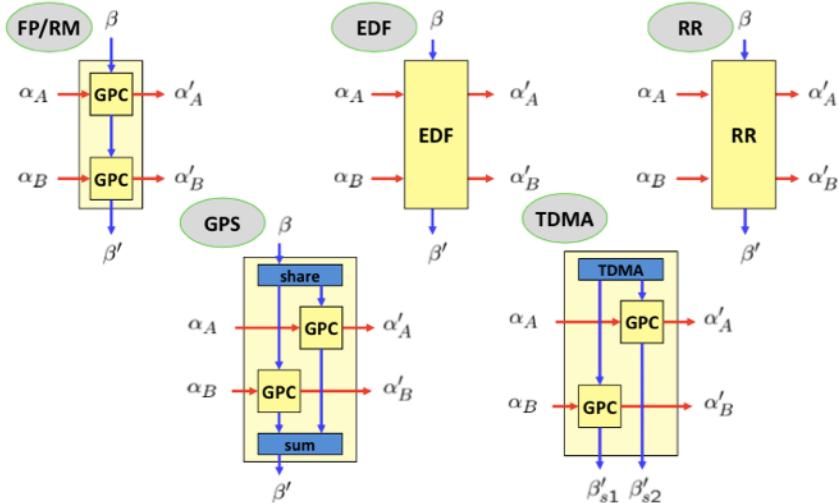


如果不满足就可能会出现任务没在截止时间前完成

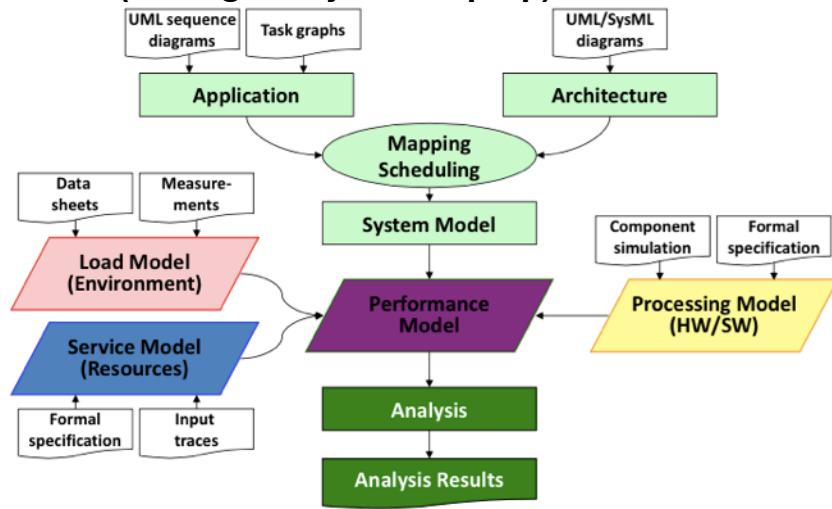
11-2、组成分析(Compositional Analysis)

1、组成分析（仅作了解）

方法：将各个功能模块变成一个个的贪心调度处理模块，进行数学分析



11-3、时序分析总结(Timing Analysis Wrap-up)



12、低功耗设计(Low Power Design)

12-1、总论(General Remarks)

1、名家谈功耗

功耗是当今嵌入式系统中的最重要考虑因素
由于电池技术的发展远不及晶体管技术的发展，因此降低功耗势在必行

12-2、功率和能量(Power and Energy)

1、功率和能量相关

功率和能量满足下列关系式：

$$E = \int P(t) dt$$

从这个式子中可以看出在功率一定的情况下处理速度越快最后消耗的能量越小
但是处理速度越快有时候反而代表着功率的提升（这一点在后面会有所解释）

2、低功耗设计的意义

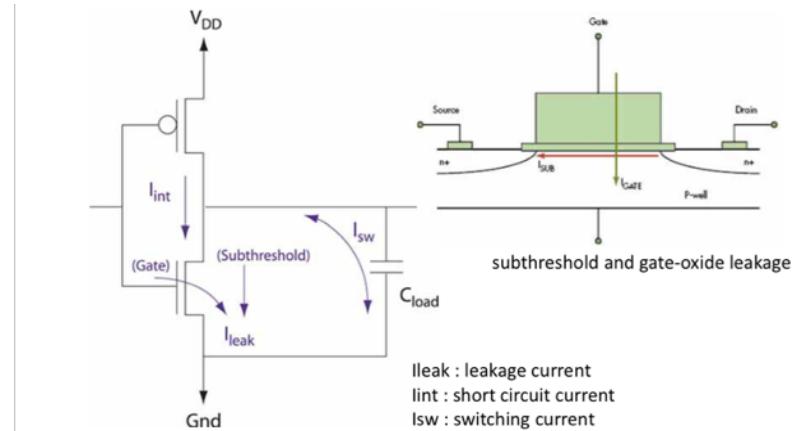
功耗高将导致移动设备的使用不可持续（电池容量有限）

功耗高将导致对电费高企

低功耗可以降低设备的运行温度，延长设备的使用寿命

3、功耗的来源

对于一个MOS管来说功耗大致有如下来源：



动态功耗：电容充电放电的功耗

短路功耗：在MOS管发生0、1切换时发生的短路功耗

漏电功耗：MOS管在静态的情况下产生的漏电（已经成为最主要因素）

动态功耗可以通过下面这个表达式进行估计

$$P \sim \alpha C_L V_{dd}^2 f$$

其中 V_{dd} 为对MOS管施加的电压， α 为MOS管发生0、1切换的频率， C_L 为MOS管的负载电容的电容值， f 为时钟频率

而 V_{dd} 又对电路的运行效率存在影响，对于MOS管来说电路的延迟满足下面这个表达式

$$\tau = kC_L \frac{V_{dd}}{(V_{dd} - V_T)^2}$$

其中 V_{dd} 为对MOS管施加的电压， V_T 为MOS管发生0、1翻转的阈值电压。由于现在阈值电压值已经远小于对MOS管施加的电压，因此其主要作用的还是 V_{dd} ，从此可以看出运行效率和低功耗有时并不可兼得

一种降低功耗的方法是设置电压域，及时切断不需要的元件的供电系统

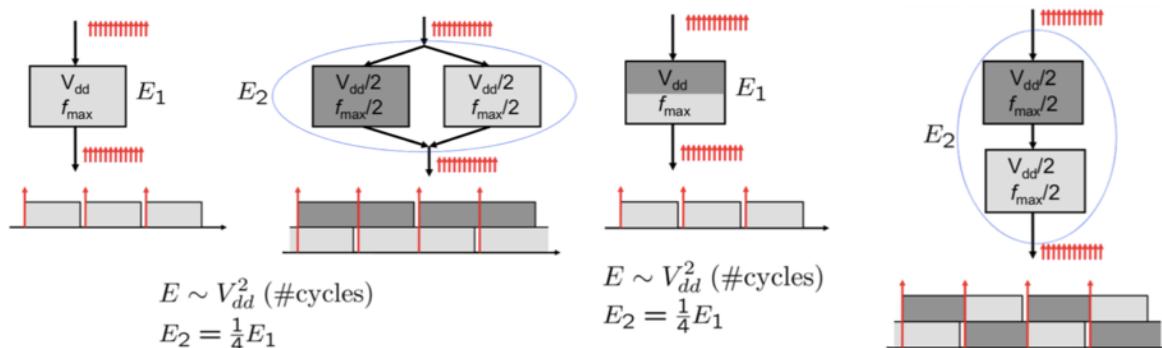
12-3、基本技术(Basic Techniques)

12-3-1、并行性(Parallelism)

由于动态功耗可以通过下面这个表达式进行估计

$$P \sim \alpha C_L V_{dd}^2 f$$

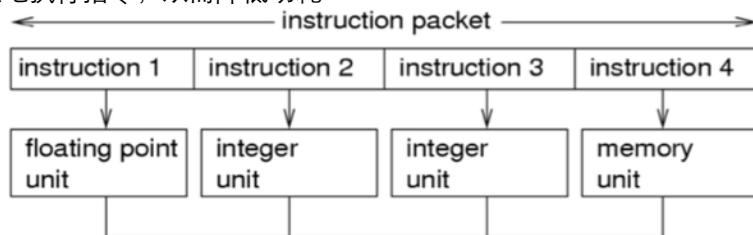
在并行性里面可以将任务分配到不同的处理器去做，最终每个处理器的电路延迟就比原来放宽一些，因此不需要对MOS管施加那么大的电压，从而可以极大地降低功耗



12-3-2、超长指令字处理器（并行性和减少开销）(VLIW (parallelism and reduced overhead))

在超长指令字处理器中由编译器进行并行检测，而不是运行时由硬件进行并行检测

从而可以更具并行性地执行指令，从而降低功耗



12-3-3、动态电压调节(Dynamic Voltage Scaling)

1、能量的计算

由于动态功耗可以通过下面这个表达式进行估计

$$P \sim \alpha C_L V_{dd}^2 f$$

同样地，能量也可以进行估计计算

$$E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd}^2 (\#cycles)$$

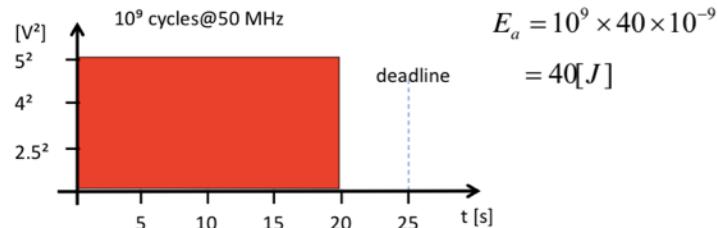
例子：假设对于任务有下面三种处理器的电压可供调节

V_{dd} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
Cycle time [ns]	20	25	40

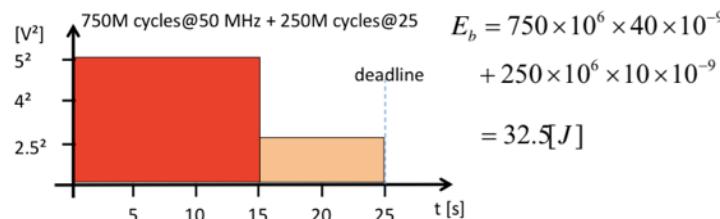
需要在25秒的截止时间之前完成 10^9 的时钟周期，给出下面三种调度方式：

- (1)、对MOS管施加5V电压运行
- (2)、对MOS管施加5V或者2.5V电压运行
- (3)、对MOS管施加4V电压运行

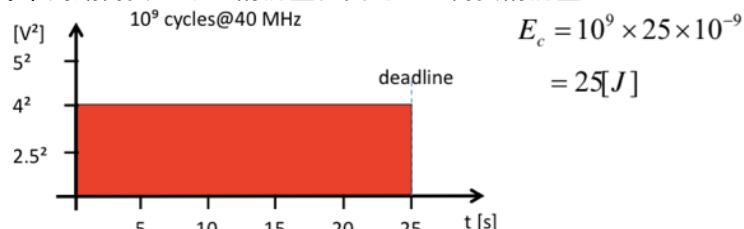
在第一种情况下由于每20纳秒运行一个时钟周期，因此要运行完全部 10^9 的时钟周期就需要20秒，因此不会出现超时现象。由于每个周期需要40纳焦的能量，因此总共需要的能量为



在第二种情况下在5V时每20纳秒运行一个时钟周期，在2.5V时每40纳秒运行一个时钟周期，因此如果要在截止时间25秒前完成，5V时的运行时间不得小于15秒（完成 $0.75 * 10^9$ 的时钟周期），在这种情况下总共需要的能量为



在第三种情况下由于每25纳秒运行一个时钟周期，因此要运行完全部 10^9 的时钟周期就需要25秒，因此不会出现超时现象。由于每个周期需要25纳焦的能量，因此总共需要的能量为



因此第三种调度方法是最优的

2、YDS调度方法

意义：提出一种在给定截止时间下通过降低电压（会导致处理器的运行速度放缓）尽可能节省功耗的算法
条件：假设如下的参数：

在处理器的相对运行频率为1时任务的计算时间 c_i

任务的到来时间 a_i

任务的截止时间 d_i

方法：

- 将所有的任务的到来时间和截止时间都放在时间轴上然后计算所有任务开始时间到所有任务结束时间的任务强度G

任务强度的计算方法：区间内的运算时间除以区间长度

- 找到任务强度最大的区间，并在那个区间内执行最早截止时间优先调度。并以任务强度作为此时的相对运行频率（要注意此时每个任务的运算时间并不是原来的相对运行频率为1时任务的计算时间 c_i ，而是要对 c_i 进行放缩，放缩的大小根据此时的相对运行频率与1的比值）

- 将第二步中找到的任务强度最大的区间删除并且更新此时的

任务的到来时间 a_i

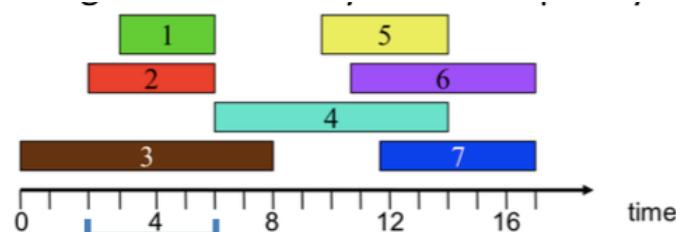
任务的截止时间 d_i

- 重新回到第一步计算新一轮任务强度G直到运行完成，将每个任务的调度结果表示出来

例子：假设我们存在如下的任务

	相对运行频率为1时任务的计算时间 c_i	任务的到来时间 a_i	任务的截止时间 d_i
任务1	5	3	6
任务2	3	2	6
任务3	2	0	8
任务4	6	6	14
任务5	6	10	14
任务6	2	11	17
任务7	2	12	17

首先计算所有任务开始时间到所有任务结束时间的任务强度G



$$G([0,6]) = (5+3)/6 = 8/6, G([0,8]) = (5+3+2)/(8-0) = 10/8,$$

$$G([0,14]) = (5+3+2+6+6)/14 = 11/7, G([0,17]) = (5+3+2+6+6+2+2)/17 = 26/17$$

$$G([2, 6]) = (5+3)/(6-2) = 2, G([2, 14]) = (5+3+6+6)/(14-2) = 5/3,$$

$$G([2, 17]) = (5+3+6+6+2+2)/15 = 26/15$$

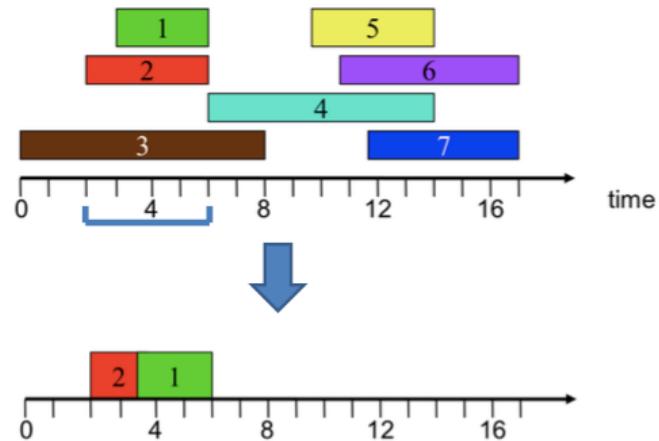
$$G([3, 6]) = 5/3, G([3, 14]) = (5+6+6)/(14-3) = 17/11,$$

$$G([3, 17]) = (5+6+6+2+2)/14 = 21/14$$

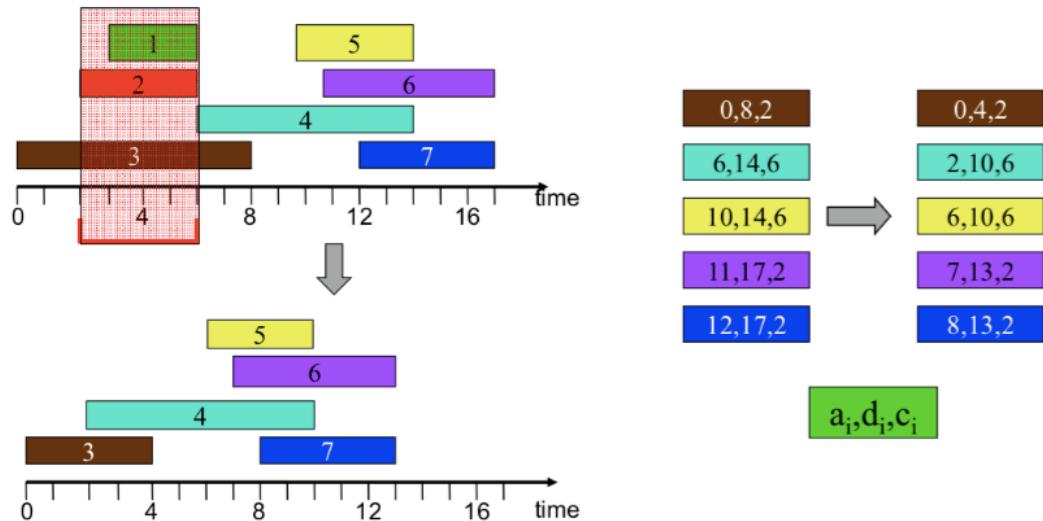
$$G([6, 14]) = 12/(14-6) = 12/8, G([6, 17]) = (6+6+2+2)/(17-6) = 16/11$$

$$G([10, 14]) = 6/4, G([10, 17]) = 10/7, G([11, 17]) = 4/6, G([12, 17]) = 2/5$$

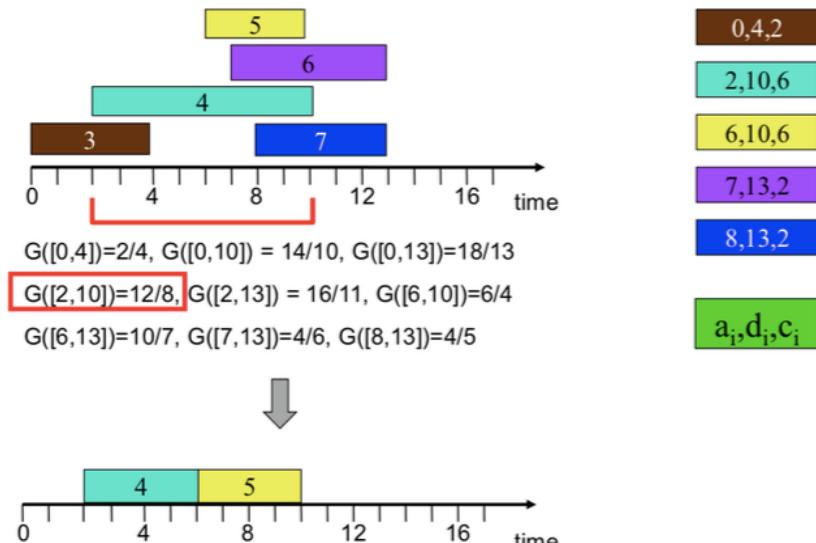
然后找到任务强度任务强度最大的区间，并在那个区间内执行最早截止时间优先调度。并以任务强度作为此时的相对运行频率（要注意此时每个任务的运算时间并不是原来的相对运行频率为1时任务的计算时间 c_i ，而是要对 c_i 进行放缩，放缩的大小根据此时的相对运行频率与1的比值）

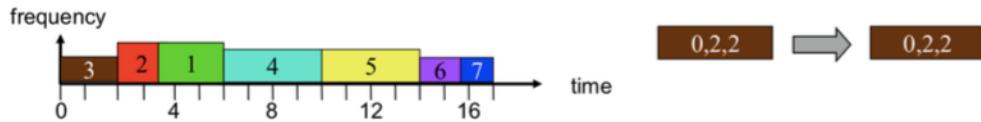
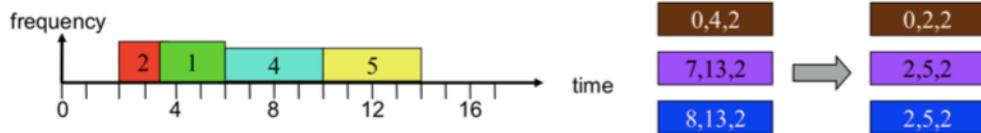


在此之后将第上一步中找到的任务强度最大的区间删除并更新此时的任务的到来时间 a_i 和任务的截止时间 d_i



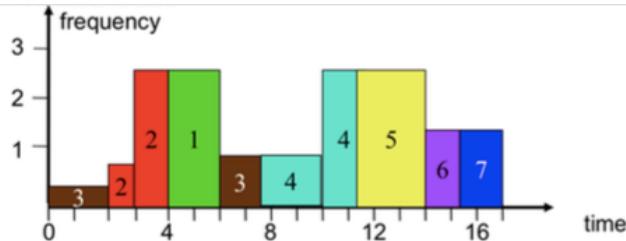
再重新计算任务强度 G ，进行下一次的循环





	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Freq.	2	2	1	1.5	1.5	4/3	4/3

最终将结果表示出来



- Continuously update to the best schedule for all arrived tasks
 - Time 0: task v_3 is executed at 2/8
 - Time 2: task v_2 arrives
 - $G([2,6]) = \frac{3}{4}$, $G([2,8]) = 4.5/6 = 3/4 \Rightarrow$ execute v_2 at $\frac{3}{4}$
 - Time 3: task v_1 arrives
 - $G([3,6]) = (5+3-3/4)/3 = 29/12$, $G([3,8]) < G([3,6]) \Rightarrow$ execute v_2 and v_1 at $29/12$
 - Time 6: task v_4 arrives
 - $G([6,8]) = 1.5/2$, $G([6,14]) = 7.5/8 \Rightarrow$ execute v_3 and v_4 at $15/16$
 - Time 10: task v_5 arrives
 - $G([10,14]) = 39/16 \Rightarrow$ execute v_4 and v_5 at $39/16$
 - Time 11 and Time 12
 - The arrival of v_6 and v_7 does not change the critical interval
 - Time 14:
 - $G([14,17]) = 4/3 \Rightarrow$ execute v_6 and v_7 at $4/3$

评价：

算法的时间复杂度： $O(N^3)$ ，其中N为任务的数量

12-3-4、动态功耗管理(Dynamic Power Management)