

# Principles of Cyber-Physical Systems

## Chapter 10: Real-Time System

Instructor: Lanshun Nie

# Lecture Content

## ❑ 实时系统定义

- ❑ 实时系统 (real-time system)
- ❑ 实时操作系统 (RTOS)

## ❑ 经典实时系统模型及相关基础概念

- ❑ 偶发任务模型 (Sporadic Task Model)
- ❑ 软实时和硬实时 (HRT and SRT)
- ❑ 资源模型 (Resource Model)

## ❑ 实时调度算法 (Real-Time Scheduling Algorithms)

- ❑ Uniprocessor Scheduling
- ❑ Multiprocessor Scheduling
- ❑ Parallel Scheduling

## ❑ 资源访问

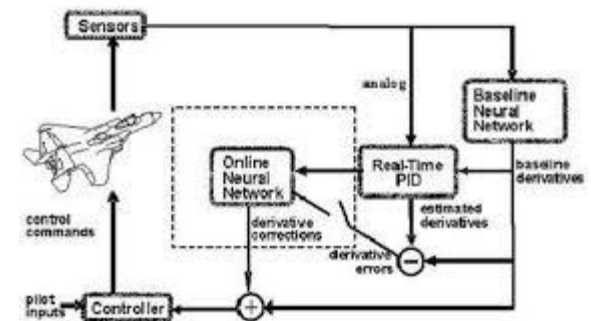
- ❑ 资源与资源访问控制
- ❑ 优先级翻转 (Priority Inversion)
  - ❑ 优先级继承
  - ❑ 优先级天花板

# Real-Time System (RTS)

- ❑ A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period
  - ❑ 实时系统的正确性不仅取决于系统产生的逻辑结果的正确性，还取决于产生这些结果的时间。
  - ❑ 没有及时响应就和错误的响应一样糟糕！

## ❑ Example

- ❑ Heart rate monitors
- ❑ Traffic lights controller
- ❑ Flight control system
- ❑ .....



# Real-Time System (RTS)



# RTOS

- ❑ A real-time operating system (RTOS) is any operating system (OS) intended to serve real-time applications that process data as it comes in, typically without buffer delays.
- ❑ 常用的有：LynxOS、pSOSystem、QNX、VRTX、VxWorks
- ❑ 特点：
  - ❑ 顺应标准
  - ❑ 模块性与可裁剪性
  - ❑ 速度与效率
  - ❑ 系统调用
  - ❑ 分阶段的中断处理
  - ❑ 调度
  - ❑ 优先级逆转的控制
  - ❑ 时钟和定时器的精度
  - ❑ 内存管理
  - ❑ 网络

## RTOS: 共同点

- 顺应标准：操作系统兼容或部分兼容**POSIX API**标准，**POSIX**标准意在期望获得源代码级别的软件可移植性。
- 模块性与可裁剪性：内核小，微内核，操作系统是可配置的。特别是，操作系统可以缩减来适应嵌入式系统的小**ROM**特点。通过为内核添加可选的组件，可以将操作系统配置成能提供**I/O**、文件以及网络服务的系统。

## RTOS: 共同点

- ❑ 速度与效率：这些操作系统大部分都是微内核，与老的微内核系统不同，他们的系统开销很低，例如任务切换时间，中断延迟，信号量的获得/释放延迟等，都是很小的，通常只有几个微秒。
- ❑ 系统调用：对于内核函数中需要互斥量的不可抢占部分优化程度很高，这些部分尽可能地短并且是可确定的
- ❑ 调度：所有实时操作系统都至少提供了**32**个优先级数，这是与实时**POSIX**兼容所需的最小数目，大多数提供了**128**甚至**256**个优先级数。

## RTOS: 共同点

- ❑ 优先级逆转的控制：提供了优先级继承和最高限度优先级协议。
- ❑ 时钟和定时器的精度：可以提供小到纳秒级别的定时器精度
- ❑ 内存管理：提供虚拟地址到物理地址的映射，但可能没有分页，也可能不提供内存保护
- ❑ 网络：可以被配置成支持TCP/IP、流等网络服务。



# LynxOS

- 从单片式体系结构到微内核体系结构
- 主要用于实时嵌入式系统，航空电子，航空航天，军事，工业过程控制和电信应用。
- 最近具有Linux兼容性
- 28KB
- 提供调度，中断分派，同步基本服务
- 提供的其他服务是轻量级服务模块，成为内核插件程序 (Kernel Plug-In, 简称KPI)，通过向微内核里添加KPI，可以将系统配置成支持I/O和文件系统、TCP/IP、流、套接字等
- KPI是多线程的，每个KPI可以根据需要创建很多线程来执行例程，KPI之间的通信只需要很少的指令。
- <https://en.wikipedia.org/wiki/LynxOS>

# pSOSystem

- 由**Alfred Chao**在大约**1982**年研发
- 面向对象的操作系统
- 也是模块化的
- 对象类包括任务，内存区，划分，消息队列以及信号量
- 系统执行的基本单元是任务，任务有自己的虚拟环境
- **2000**年**2**月，被竞争对手**VxWorks**的公司**Wind River Systems**收购。尽管初步报告称**pSOS**支持将继续，但开发工作仍停止。**Wind River**宣布计划推出支持**pSOS**系统调用的**VxWorks**“融合”版本，并且不会进一步发布**pSOS**本身。
- [https://en.wikipedia.org/wiki/PSOS\\_\(real-time\\_operating\\_system\)](https://en.wikipedia.org/wiki/PSOS_(real-time_operating_system))

# QNX/Neutrino

- 起源于滑铁卢大学的两名学生**Gordon Bell**和**Dan Dodge**的一个实时操作系统课程设计，后成立公司**Quantum Software Systems**继续维护，最终于**2010**年被**BlackBerry**收购。
- 是首批商业上成功的微内核操作系统之一，适合于多种设备，包括汽车和移动电话
- 是个消息传递的操作系统，消息是所有线程之间通信的基本方法。
- 其微内核提供了基本的线程和实时服务，其他操作系统服务由成为资源管理器的可选组件来提供
- 由于运行时可以不包括可选组件，所以可被缩减成很小的规模。（**QNX4.x**的微内核只有**12KB**大小。）
- <https://en.wikipedia.org/wiki/QNX>

# VxWorks

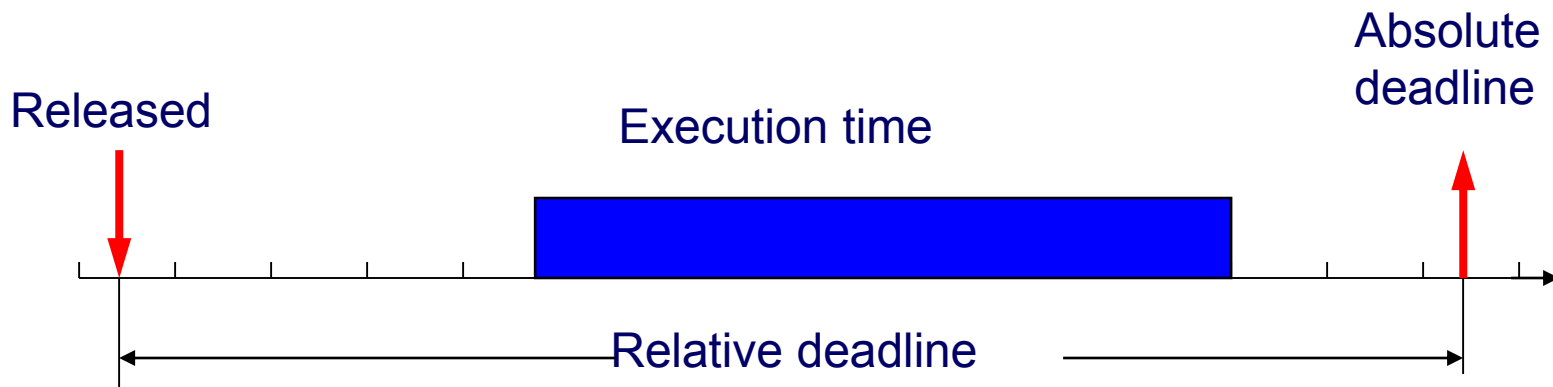
- 是由美国加利福尼亚州阿拉米达的英特尔子公司**Wind River Systems**开发的专有软件实时操作系统（**RTOS**）。
- **1987**年首次发布，专为需要实时，确定性能的嵌入式系统而设计。
- 在许多情况下还用于安全和安全认证，适用于航空航天和国防，医疗设备，工业设备，机器人，能源等行业。运输，网络基础设施，汽车和消费电子产品。
- 在其最新版本**VxWorks 7**中，**RTOS**经过重新设计，具有模块化和可升级性，因此操作系统内核与中间件，应用程序和其他软件包分开。
- 可扩展性，安全性，安全性，连接性和图形性已得到改进，以满足物联网（**IoT**）的需求。
- <https://en.wikipedia.org/wiki/VxWorks>

# 什么是RTOS所需要的

- ❑ 快速上下文切换?
  - ❑ should be fast anyway
- ❑ **Small size?**
  - ❑ should be small anyway
- ❑ 对外部触发事件的快速响应?
  - ❑ 不一定快但可预测
- ❑ 多任务?
  - ❑ 经常使用，但非必须
- ❑ **“Low Level” programming interfaces?**
  - ❑ 可能与其他嵌入式系统一样需要底层编程接口
- ❑ 较高的处理器利用率?
  - ❑ 在任何系统中都是可取的（但要避免超载）

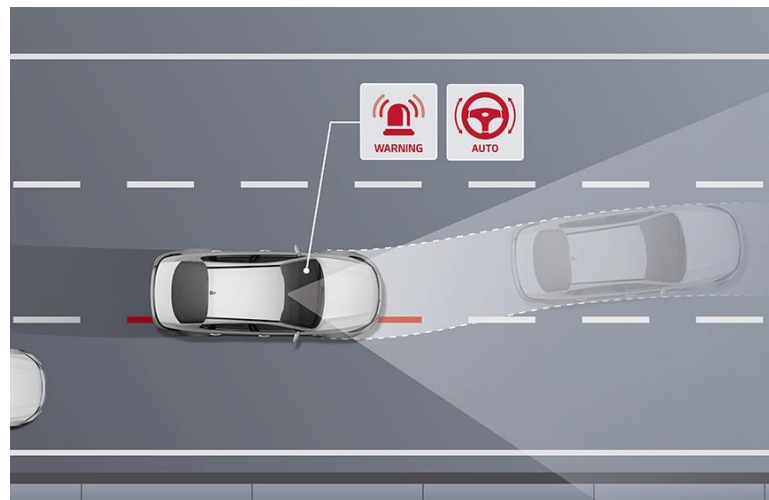
## RTS中的术语

- **作业 (Job)** :带时序约束的计算任务
  - 执行时间 (Execution time, work)
  - 释放时间(release time):作业就绪的时刻
  - 响应时间 $R_i$  (Response time ) = 完成时间-释放时间
  - 绝对截止时间(Absolute deadline ) = 释放时间+ 相对截止时间 $D_i$

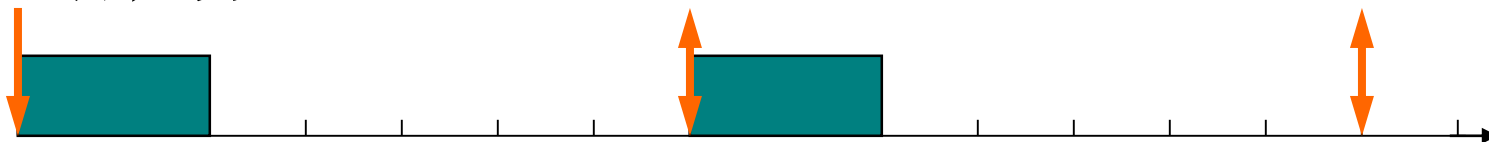


# RTS中的术语

- **任务(task)**: 某种应用
  - 一系列类似的作业
  - 可能会多次释放，每次释放都是产生一个作业
  - **作业**: 任务的实例
  - **最坏执行时间(work)**  $C_i$
  - 相对截止期限  $D_i$



- **周期任务**
  - Inter-arrival time = Period  $P_i$
  - 释放间隔时间 = 周期  $P_i$
  - 隐含  $D_i =$  周期  $P_i$
- **非周期任务**



# Classical RTS Model : Sporadic Task Model

- 是一个已经被充分研究且简单的任务模型
- 每个**task**都是一个顺序程序，它响应外部事件（如设备中断）而被重复调用，一次调用被看成释放一个**job**，由于可以无限次调用某个复发任务，一个复发任务可以生成任意多个**job**
- 在此模型下，**RTS**被建模为**n**个偶发任务的集合 $T = \{T_1, T_2, T_3 \dots, T_n\}$ 
  - 每个偶发任务用三元组 $(e_i, d_i, p_i)$ 来描述
  - $e_i$ 代表最坏执行时间（WCET），一个任务中，所有作业最坏执行时间
  - $d_i$ 代表相对截止时间, 相对于发布时间来说的, 平台无关
  - $p_i$ 代表两个作业发布时间的至少间隔, 平台无关

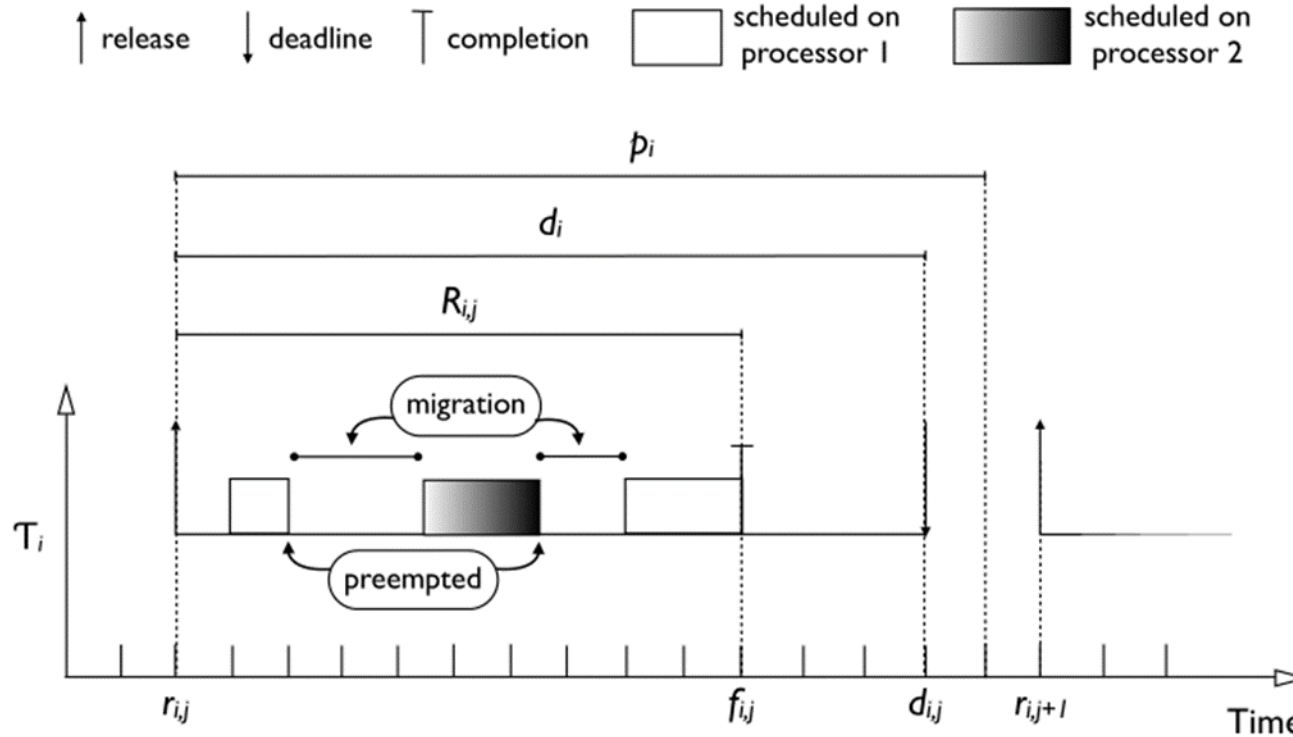


# Classical RTS Model : Sporadic Task Model

- 是一个已经被充分研究且简单的任务模型
- 对于周期任务， $p_i$ 参数是固定的，即两个连续作业的释放时间间隔是固定的
- 第 $i$ 个任务 $T_i$ 释放的第 $j$ 个作业表示为 $T_{i,j}$ ，释放时间为 $r_{i,j}$ ，由于任务周期限制，必须满足 $r_{i,j+1} \geq r_{i,j} + p_i$ ， $T_{i,j}$ 最多执行 $e_i$ 个时间单元，完成时间表示为 $f_{i,j}$ ，响应时间表示为 $R_{i,j} = f_{i,j} - r_{i,j}$ ，任务 $T_i$ 的响应时间 $R_i$ 是该任务释放的作业中响应时间最大的那个。
- 偶发任务是顺序的， $T_{i,j+1}$ 只有在 $f_{i,j}$ 之后才能执行，即使 $r_{i,j+1} \leq f_{i,j}$ ，
- $T_{i,j}$ 的绝对截止时间为 $d_{i,j} = r_{i,j} + d_i$ ，若完成时间晚于该值，则称它是 **tardy** 的，迟缓程度 **tardiness** 等于  $\max(0, f_{i,j} - d_{i,j})$ 。
- 若 $d_i = p_i$ ，隐式截止期限偶发任务； $d_i \leq p_i$ ，约束截止期限偶发任务； $d_i$  任意，任意偶发任务

# Classical RTS Model : Sporadic Task Model

- 一个偶发任务 $T_i$ 的图示：作业 $T_{i,j}$ 被更高优先级的作业抢占了两次（图中未标出），因此 $T_{i,j}$ 迁移了两次：第一次从核心1跳到核心2，第二次再跳回到核心1执行

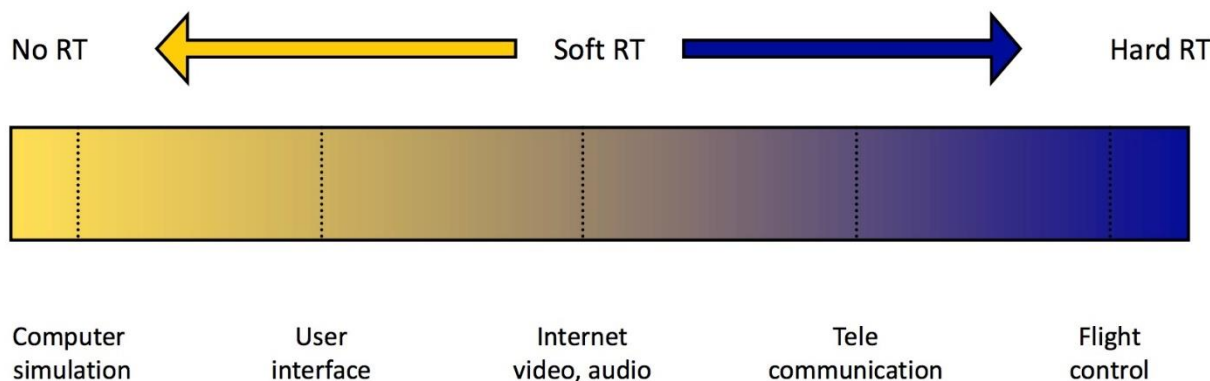


# Classical RTS Model : Sporadic Task Model

- $T_i$ 的利用率表示为 $u_i = \frac{e_i}{p_i}$ ，代表了任务需要的处理器资源
- 一个任务集合 $T$ 的总利用率 $U_{sum} = \sum_{\tau_i \in \tau} u_i$ ，若总利用率超过了核心个数 $m$ ，则说明系统超载
- 对于 $d_i \leq p_i$ ，即约束截止期限偶发任务而言，密度也是一个重要概念，表示为 $\delta_i = \frac{e_i}{d_i}$ ，总密度表示为 $\delta_{sum} = \sum_{\tau_i \in \tau} \delta_i$ ，

# HRT and SRT

- **HRT**: 系统中所有任务释放的作业都不能错过自己的截止期限，即对于所有的作业  $f_{i,j} \leq d_{i,j}$  都成立
- **SRT**: 多种定义，但以第一种为主
  - 定义1: 允许作业超过截止期限，但是对于任意作业，需要满足  $f_{i,j} - d_{i,j} \leq C$ ，其中  $C$  是常数，同时也可以用来衡量 **soft** 程度，当  $C=0$  时就是 **HRT**
  - 定义2: 系统中固定百分比的作业的截止期限需要保证被满足
  - 定义3: 每连续  $y$  个作业中必须至少有  $x$  个作业的截止期限要保证满足
- **Example**:
  - **HRT**: 自动驾驶控制系统，空间站任务调度系统，飞行控制系统。。。
  - **SRT**: 多媒体系统，计算机视觉和图像处理系统，移动计算系统。。。



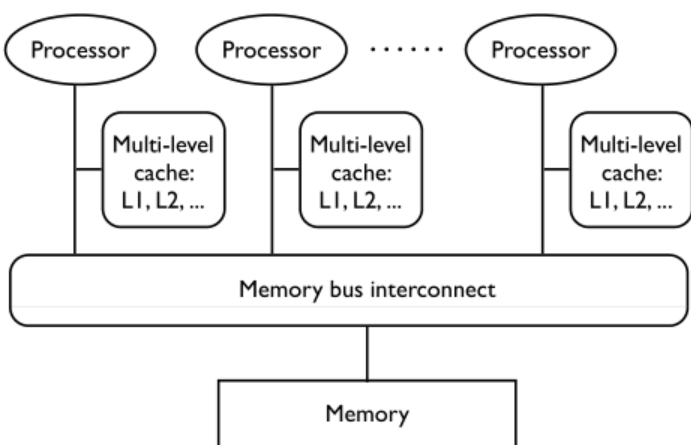
# HRT and SRT: Temporal Correctness



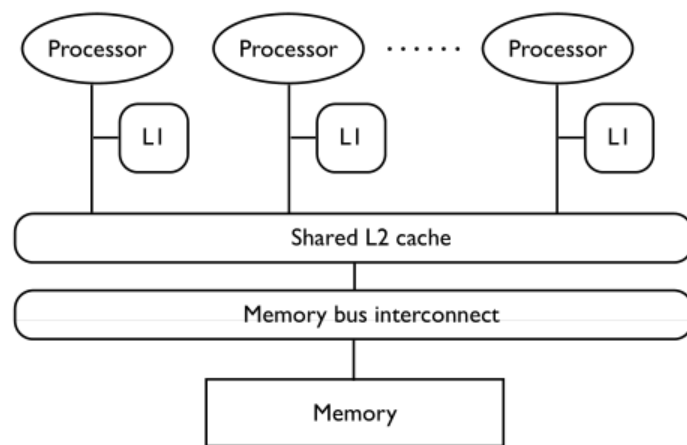
- 根据**HRT**和**SRT**要求限制，实时任务的时序正确性本质上由最大响应时间决定（**maximum response time**），而响应时间取决于底层的调度算法，因此**HRT**和**SRT**可调度性可形式化定义如下：
  - 定义**1.1**：一个任务集合**T**在一个调度算法**A**下是**HRT**可调度的，当且仅当  $R_i \leq d_i$  对于每个任务都成立。
  - 定义**1.2**：一个任务集合**T**在一个调度算法**A**下是**SRT**可调度的，当且仅当存在一个常数**C**使得  $R_i \leq d_i + C$  对于每个任务都成立，这里**C**是一个常数。

# Resource Model

- ❑ 为了实现统一的内存访问时间，通常采取具有集中式内存的多处理器体系结构（**SMP**）。
- ❑ 如下图(a)表示的是不共享缓存（**without a shared cache**）的**SMP**架构，图(b)表示的是共享缓存（**with a shared cache**）的**SMP**架构。



(a)



(b)

# Resource Model

- ❑ 允许每个作业在任何处理器上执行，在任何时间点占用最多一个处理器（并行任务可占用多个）。
- ❑ 作业（任务）在不同的处理器上执行，我们称之为迁移（**migrate**）
- ❑ 由于将与作业相关的指令和数据重新加载到本地缓存中需要时间开销，因此迁移会产生迁移开销。
- ❑ 降低或消除迁移开销的技术
  - ❑ 例如，将任务或作业的执行限制在一个或一个处理器子集中（强行限制迁移）
  - ❑ 此外，对于现在的多核架构，同一芯片上的不同内核通常在某个级别共享缓存，如图（b）所示。
  - ❑ 在许多情况下，这样的共享高速缓存能够减少迁移开销

# Resource Model: 计算开销

- ❑ 开销来源:
  - ❑ 迁移开销
  - ❑ 任务抢占
  - ❑ 上下文切换
  - ❑ 调度算法本身的调度行为。
- ❑ 为了确保时限正确性，任何开销都要被考虑进去，通常是将每一任务的WCET扩大到这些外部活动所需的最大总时间来实现的，即考虑最坏情况下的时间开销。
- ❑ 假定WCET就代表系统总时间开销



# 评价指标

- 两个重要概念：可行性（**feasibility**）和最优性（**optimality**）。
  - 定义1.3：任务集 $\mathbf{T}$ 是HRT（SRT）可行的，当且仅当存在至少一个调度算法A，在该算法下T是HRT（SRT）可调度的。
  - 定义1.4：算法A是最优的，当且仅当每个HRT（SRT）可行的任务集合T，在该算法下是HRT（SRT）可调度的。
- 为了检查给定的一个任务系统是否是可行的，一个最直接的方法是看利用率，利用率代表了对处理器核心的需求，因此，这意味着任务集的总利用率不能超过处理器的总数，否则总处理器需求可能超过可用核心数，这导致任务违反其时间约束。
  - 引理1.1：仅当 $U_{\text{sum}} \leq m$ 时， $m$ 个处理器上的实时偶发任务集才是可行的。（必要性）
  - 引理1.2：一个隐式截止期限的实时偶发任务集是HRT可行的，当且仅当 $U_{\text{sum}} \leq m$

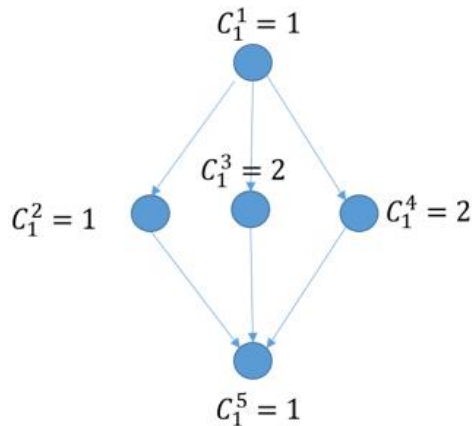
# The Parallel Task Model

- 基于偶发任务系统，相关概念一致，针对并行任务对其进行修改
- 任务集合里的每一个任务都是一个并行任务，而且由一个有向无环图（directed acyclic graph DAG）表示。
  - DAG里的每一个节点都是一个子任务，节点之间的边代表着依赖关系。
  - 一个子任务只有当它的前驱任务（前驱节点）执行完毕之后才能开始执行（但也不是立即就能执行，受调度算法的影响），该并行任务的DAG模型有几个非常重要的参数，定义如下：

- 总执行时间（total execution time/work）：任务 $T_i$ 的总执行时间是它的DAG表现形式下所有节点（子任务）执行时间的加和，同样用 $e_i$ 表

- 关键路径长度（critical-path length）：即给定任务DAG的关键路径的表示意义代表在无线多核心下执行该任务的一个实例花费的

时间，可以在线性时间内获得它的总执行时间 $e_i$ 和关键路径其他相关的利用率和总利用率等的定义是一致的。



# Lecture Content

## ❑ 实时系统定义

- ❑ 实时系统 (real-time system)
- ❑ 实时操作系统 (RTOS)

## ❑ 经典实时系统模型及相关基础概念

- ❑ 偶发任务模型 (SporadicTask Model)
- ❑ 软实时和硬实时 (HRT and SRT)
- ❑ 资源模型 (Resource Model)

## ❑ 实时调度算法 (Real-Time Scheduling Algorithms)

- ❑ Uniprocessor Scheduling
- ❑ Multiprocessor Scheduling
- ❑ Parallel Scheduling

## ❑ 资源访问

- ❑ 资源与资源访问控制
- ❑ 优先级翻转 (Priority Inversion)
  - ❑ 优先级继承
  - ❑ 优先级天花板

# 实时调度算法

## ☐ 单核调度:

- ☐ JFP调度: job-level fixed-priority scheduling
- ☐ JDP调度: job-level dynamic-priority scheduling
- ☐ TFP调度: task-level fixed-priority scheduling

## ☐ 多核调度:

- ☐ Partition
- ☐ Global
- ☐ Cluster

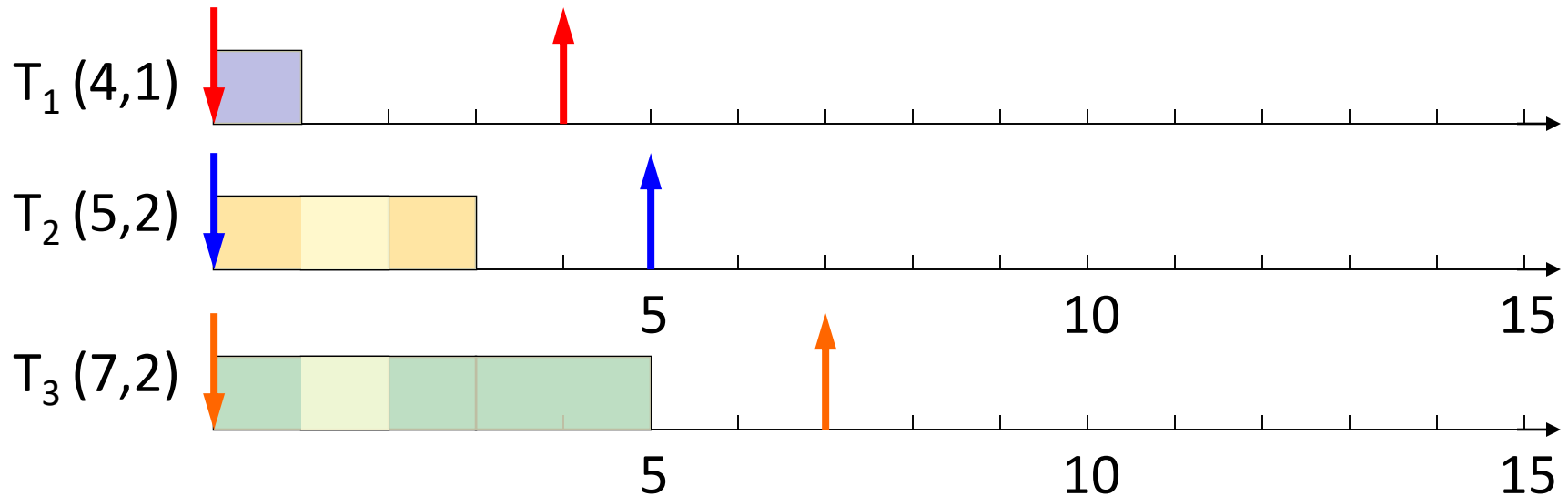
## ☐ 并行调度

## 单核调度：JFP调度

- JFP调度算法下，每个作业在运行时拥有自己独有的优先级，这也是job-level的意义所在，在作业层面固定优先级
- 最重要的JFP调度算法是earliest-deadline-first (EDF)，最早截止时间优先
  - 绝对截止时间早的作业拥有更高的优先级
  - 如果两个作业的截止时间相同，任务ID小的那个优先级别高
  - 具有单核调度最优性，即任何可行的偶发任务系统都能被EDF调度
  - 对于隐式截止时间偶发任务集来说，当且仅当 $U_{sum} \leq 1$ 时，所有作业的截止时间都能被满足，相反，大于1时，延迟度将会无限增大（响应时间也是如此）

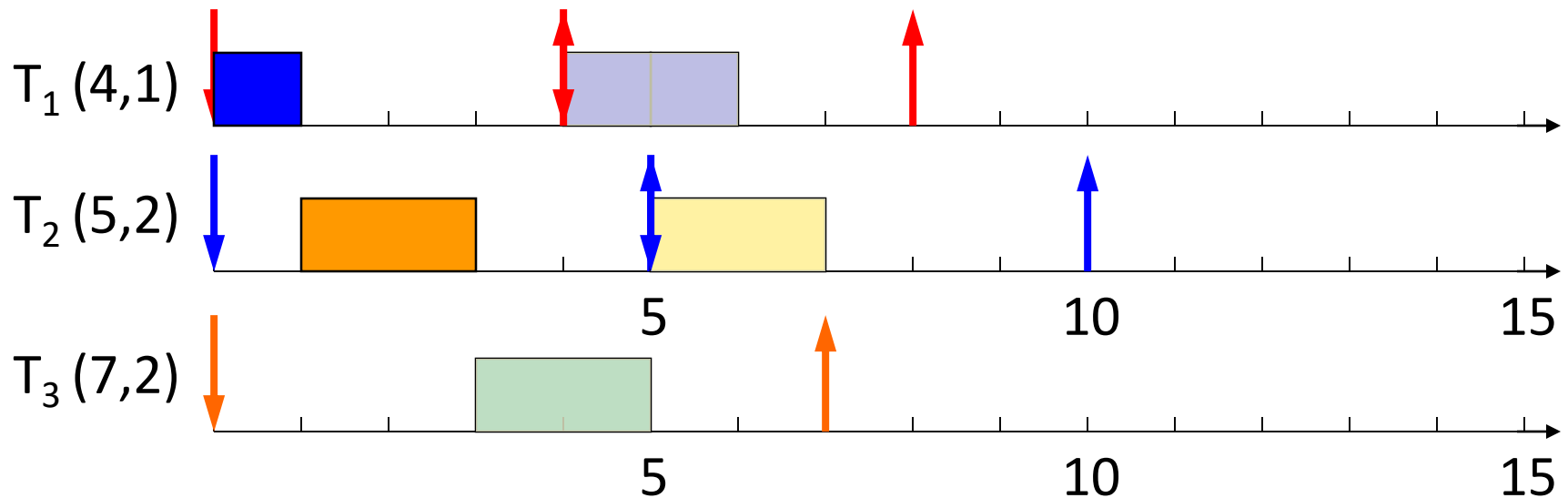
# EDF (Earliest Deadline First)

- 绝对截止时间早的作业优先级高
- 每次选择执行优先级最高的作业



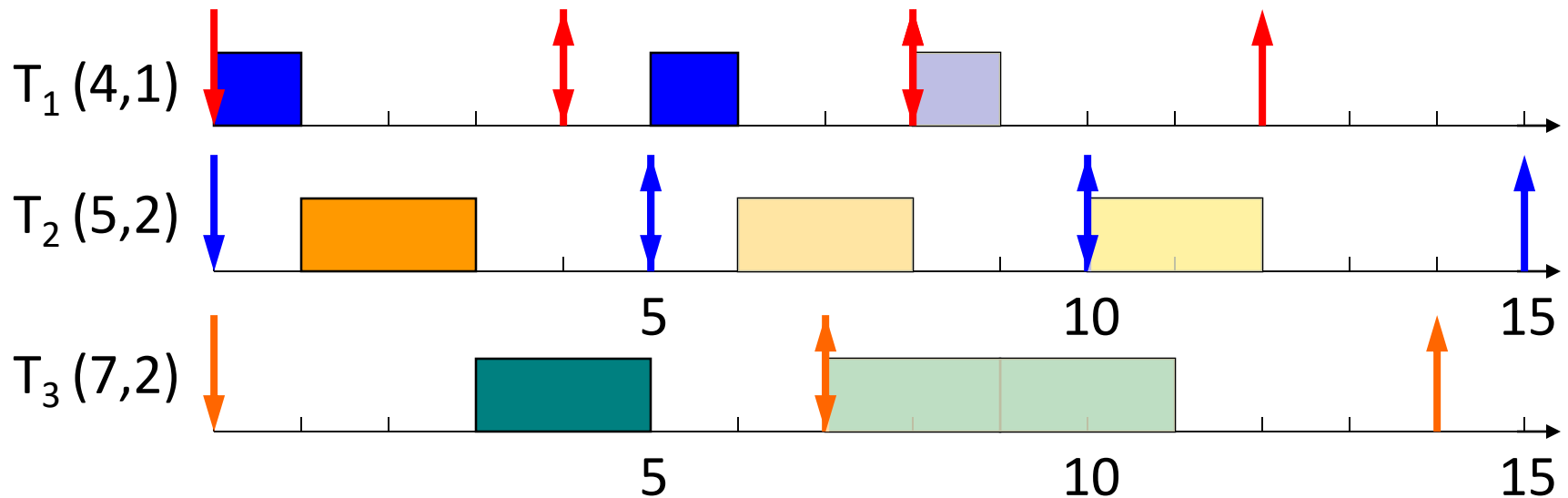
# EDF (Earliest Deadline First)

- 绝对截止时间早的作业优先级高
- 每次选择执行优先级最高的作业



# EDF (Earliest Deadline First)

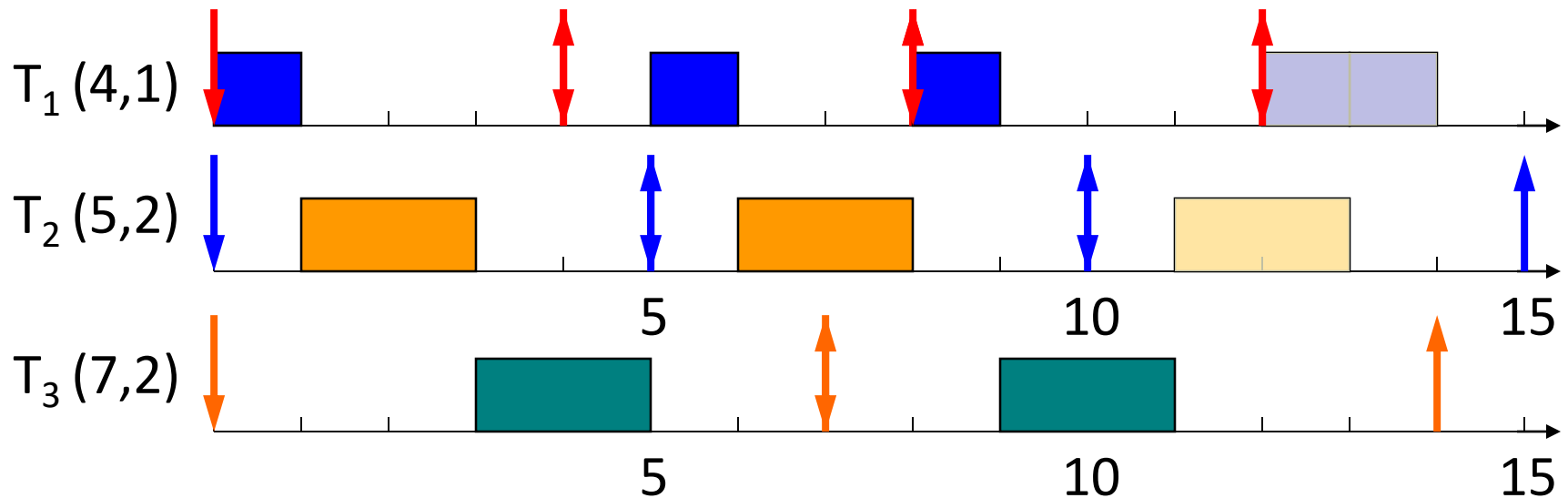
- 绝对截止时间早的作业优先级高
- 每次选择执行优先级最高的作业





# EDF (Earliest Deadline First)

- 绝对截止时间早的作业优先级高
- 每次选择执行优先级最高的作业



# Schedulable Utilization Bound(可调度利用率界)

□ 总利用率:

$$U = \sum_{i=1}^n \frac{e_i}{p_i}$$

n:任务的个数

□ 利用率界 ( Utilization Bound )  $U_b$

□ 如果  $U \leq U_b$ , 所有的任务都能按时完成

□  $U > m$ 时, 不存在调度算法可以调度一个任务集使其满足所有时限

□ m是核心数

□ 如果一个调度算法的  $U_b = m$ , 那它就是最优的

## EDF调度的性质(单核下)

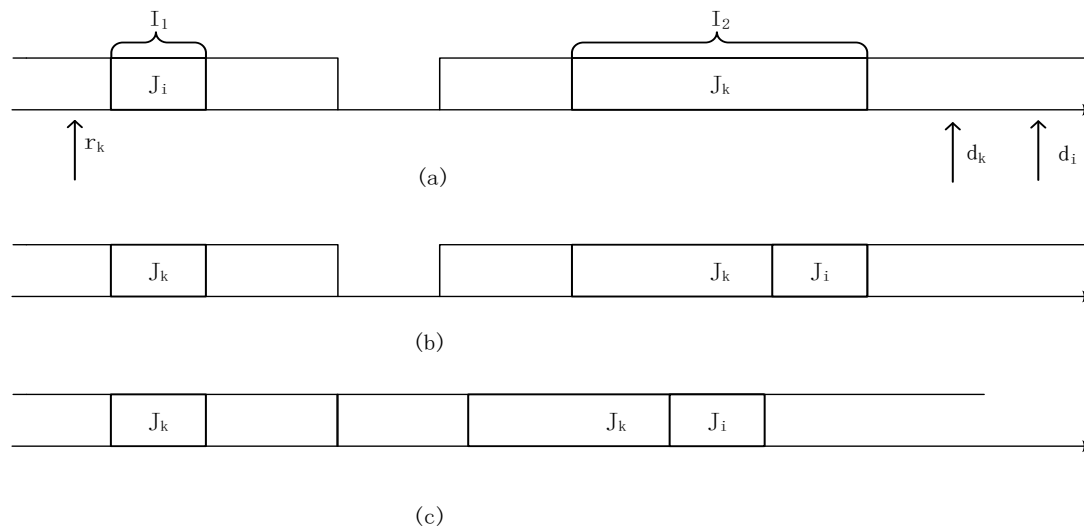
□  $U_b = 1$

□  $U \leq 1$ : 可调度性的充分必要条件

□ 系统只要不超载，就能确保可调度性

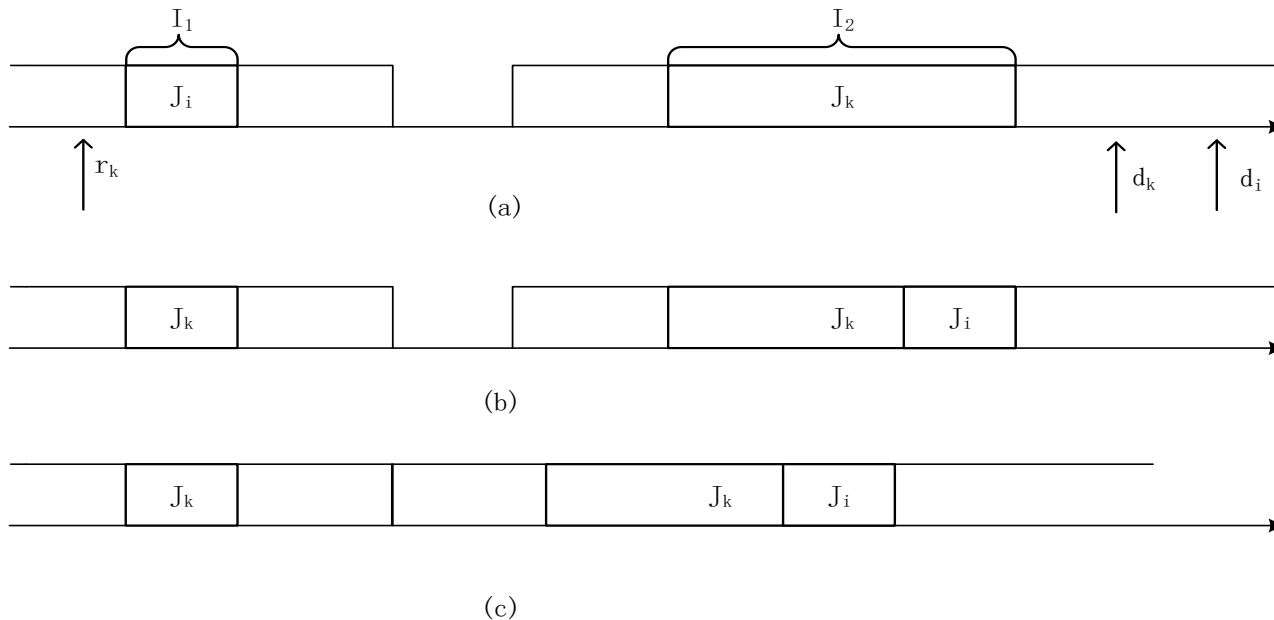
# EDF调度的性质：单核可抢占下的最优性

- 对于具有任意释放时间和截止时间的任务集 $T$ ，当作业允许抢占，但是并不竞争资源时，当且仅当 $T$ 存在可行的调度时，EDF调度算法总能在单处理器上产生一个可行的调度
- 证明：这个证明基于以下事实： $T$ 的任意可行调度可以系统地转换成一个EDF调度（即EDF调度算法产生的调度）。为了证明这一点，不失一般性，考虑这样的作业对，假设某一可行调度下，在时间区间 $I_1$ 和 $I_2$ 中分别调度作业 $J_i$ 和 $J_k$ 的一部分，此外，假设 $J_i$ 的截止时间 $d_i$ 迟于 $J_k$ 的截止时间 $d_k$ ，但是 $I_1$ 却早于 $I_2$ ，如图(a)所示：



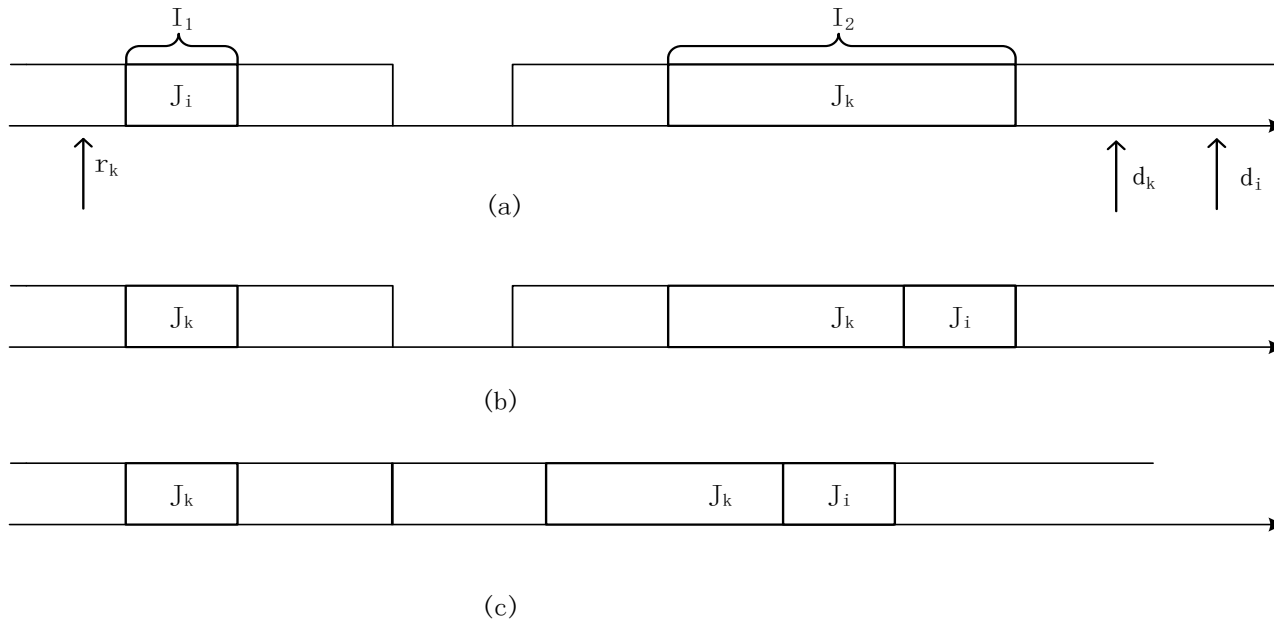
## EDF调度的性质：单核可抢占下的最优性

- 存在两种情况，分别考虑如下：第一种情况， $J_k$ 的释放时间可能迟于 $I_1$ 的结束时间，在这种情况下， $J_k$ 不能在 $I_1$ 中被调度，但是在这些时间区间内，这两个作业已经是基于EDF算法被调度的，因此不需要被转换，所以只需要考虑第二种情况了，即 $J_k$ 的释放时间 $r_k$ 早于 $I_1$ 的结束时间，为了不失一般性，假设 $r_k$ 不迟于 $I_1$ 的开始时间。



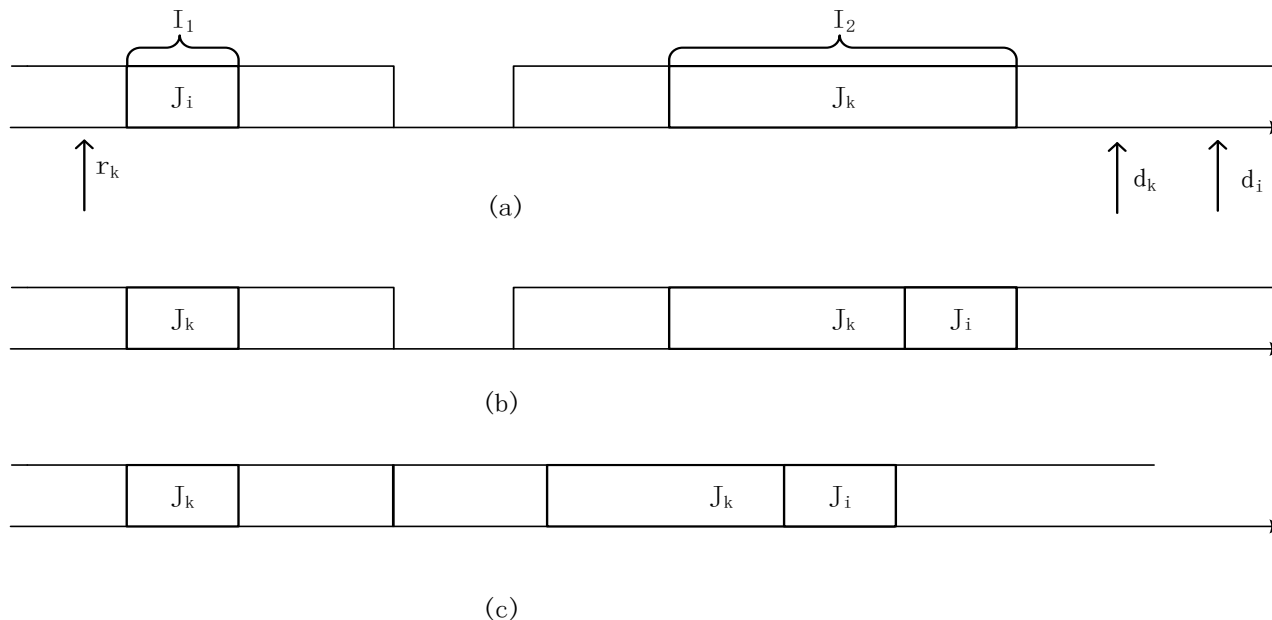
## EDF调度的性质：单核可抢占下的最优性

- 为了将给定的调度转换成EDF算法产生的调度，我们交换 $J_i$ 和 $J_k$ ，同样根据时间区间 $I_1$ 和 $I_2$ 的大小关系分为两种情况，第一种情况， $I_1$ 长度小于 $I_2$ ，如图所示，就将 $J_k$ 中截取适合 $I_1$ 长度的部分向前移至 $I_1$ ，将 $I_1$ 中原本被调度的 $J_i$ 的全部向后移至 $I_2$ 中，并放在 $J_k$ 的后面，结果如图b所示，显然，这种交换总是可行的。



## EDF调度的性质：单核可抢占下的最优性

- 第二种情况， $I_1$ 长度大于 $I_2$ ，也可以做类似的交换，将在 $I_2$ 中被调度的 $J_k$ 的全部前移至 $I_1$ 中，并且放在 $J_i$ 的前面，再将 $J_i$ 中截取适合 $I_2$ 长度的部分后移至 $I_2$ 中。这两种情况处理完毕后，这两个作业的调度顺序就是基于EDF的调度。
- 按照上述给定的方法，对于给定的非EDF调度，针对任何一对不是基于EDF调度的作业，重复以上的转换过程，直到这种作业对不再存在为止。



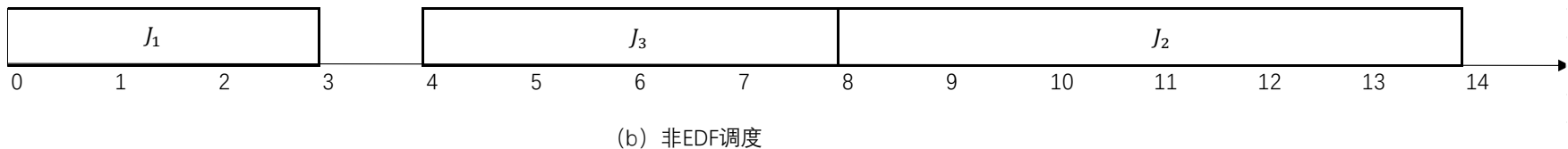
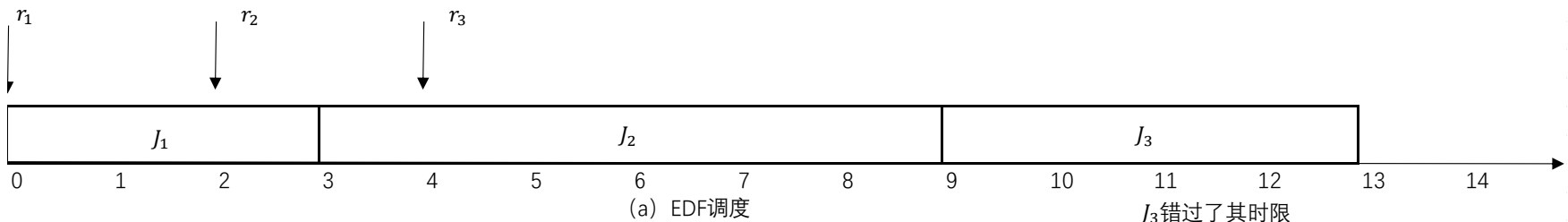
## EDF调度的性质：单核可抢占下的最优性

- 此外如果留有一些空闲区间，以至于虽然有就绪可执行的作业但是要在稍后的区间中才能得到调度，那么通过这种转换的调度可能仍然不是EDF调度，因为EDF调度是贪婪的调度算法，处理器在有作业就绪可执行的时候永远不会空闲。
- 这个时候对转换后的调度稍加处理即可，具体操作是将这些作业中的一个或者若干个向前移至处理器核心空闲的区间，使得这些区间不再空闲，而使原先调度作业的区间成为空闲的，显然这样的处理总是有可能的，如果有必要的话，可以重复这一过程，直到在有作业准备就绪可执行的状态下，处理器核心永远不会空闲。



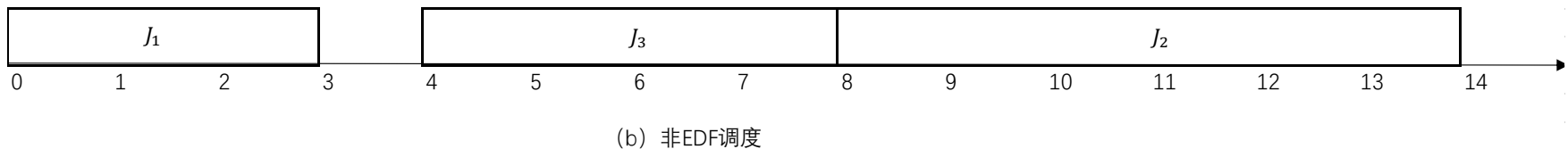
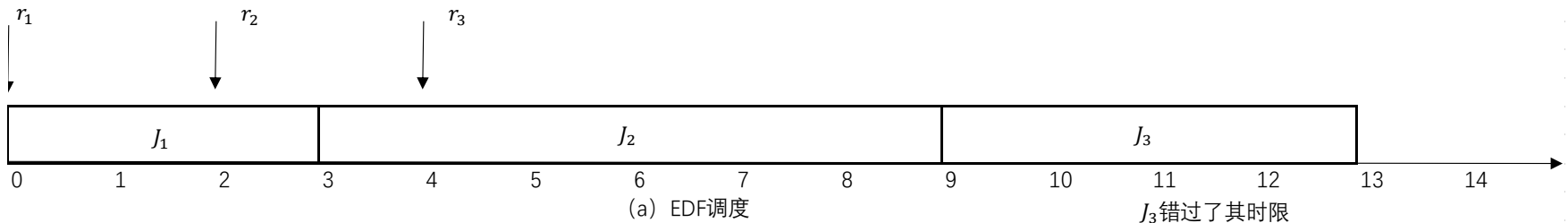
# EDF调度的性质：单核不可抢占下的非最优性

- 上面EDF算法最优性的证明的前提条件是允许抢占和单核心，但是一个很自然的问题就会出现，如果不允许抢占或者有多个处理器核心的情况下，EDF算法是否是最优的，结论是否定的，对于否定性的结论通常只需要举反例即可，下面是证明过程。
- 在单核不允许抢占的情况下，EDF算法不是最优的，进而所有基于优先级的调度算法都不是最优的
- 证明：考虑三个独立的，不可抢占的作业 $J_1$ ， $J_2$ ， $J_3$ ，他们的释放时间分别为0，2，4，由调度图中的箭头所指示，他们的执行时间分别为3，6，4，截止期分别是10，14和12。图（a）展示了按照EDF调度算法产生的执行过程。



# EDF调度的性质：单核不可抢占下的非最优性

- 我们可以注意到作业1在时刻3完成执行，此时作业2已经被释放，作业3还未释放，因此，作业2被调度执行，当时刻到达4时，作业3也被释放，虽然作业3相对于作业2具有更高的优先级（截止时间更早），但是由于作业2正在被执行，不能被抢占，只能等到作业2执行完毕才能开始执行，结果作业3错过了它的截止时限，这是按照EDF调度算法产生的调度结果，是个不可行的调度
- 实际上，如果按照图（b）进行调度，则同样的三个作业都能满足截止期限。那就是在时刻3，作业1完成的时候，不立刻调度作业2，而是等到时刻4，作业3释放后，先调度作业3，作业3完成后，再调度作业2，这样一来就能使三个作业均满足时限要求。



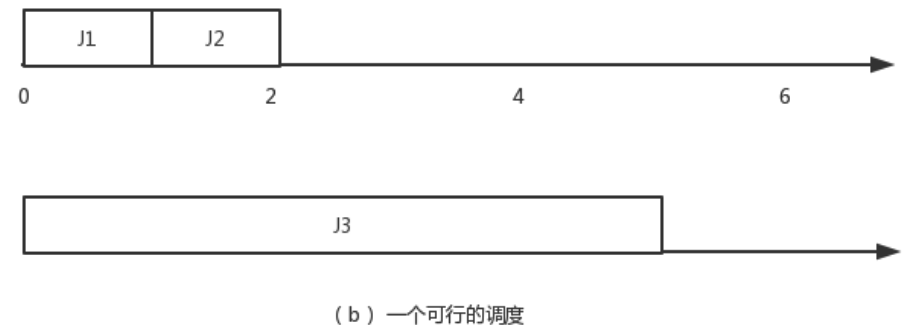
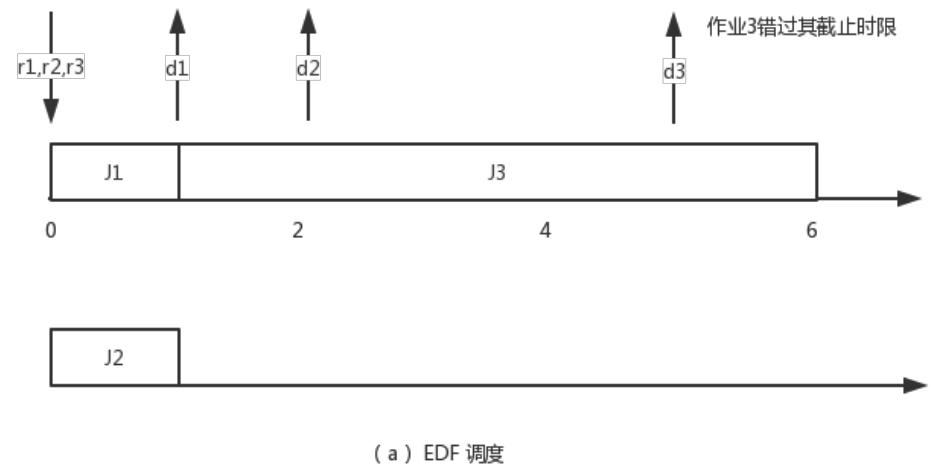
## EDF调度的性质：单核不可抢占下的非最优性

- 更进一步的，可以看出这种调度结果绝不是基于优先级的调度算法的。这是因为，根据定义，使用优先级驱动的调度算法在有作业释放且就绪的情况下，不能使处理器核心处于空闲的状态。而在图（b）中，在[3,4]时间段，作业2处于就绪状态，但处理器却处于空闲状态。
- 这一个简单的例子也说明了这样一个重要的事实，当作业具有任意的释放时间，执行时间和截止时限时，不但非抢占的EDF调度算法不是最优的，所有的基于优先级调度的算法也都不是最优的，得证。

# EDF调度的性质：多核可抢占下的非最优性

□ 多核调度允许抢占中，EDF调度算法不是最优的

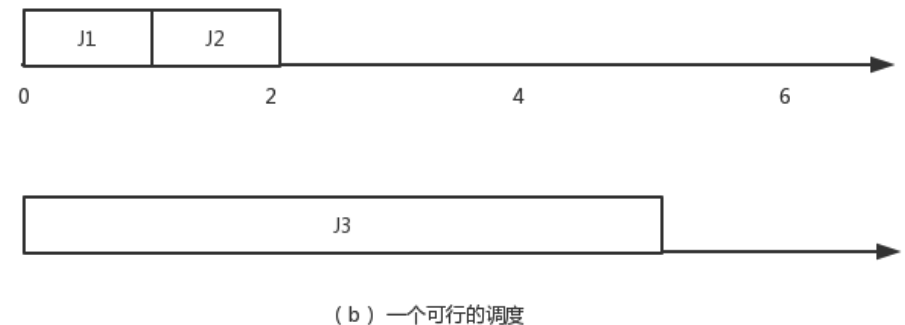
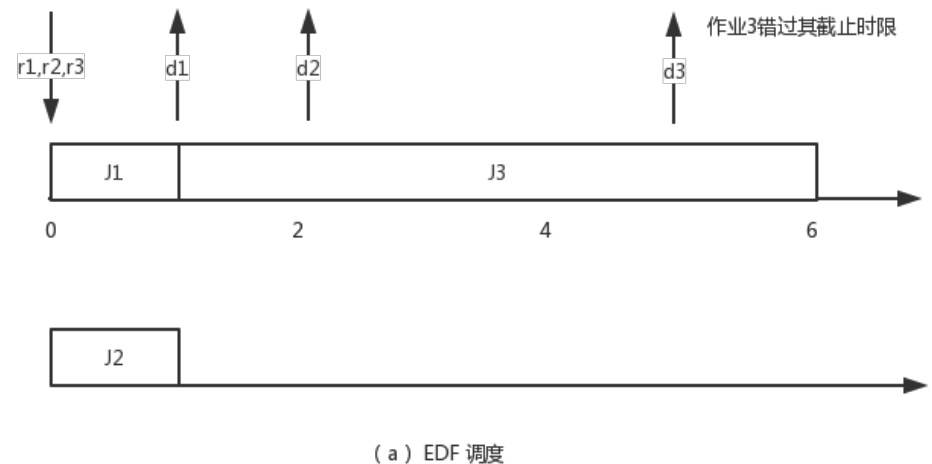
□ 证：如右图所示，该系统包括三个作业 $J_1$ ， $J_2$ ， $J_3$ 。执行时间分别是1，1，5，截止时间分别是1，2，5，释放时间全都是0时刻，系统中有两个核心，按照EDF的调度规则，在时刻0处，作业1和2被分别调度在两个核心上，这是因为他俩的优先级比作业3高，一个单位时间过后，开始执行作业3，无论作业3放到哪个核心上执行，都会错过自己的截止时间。



# EDF调度的性质：多核可抢占下的非最优性

□ 多核调度允许抢占中，EDF调度算法不是最优的

□ 证：而另一方面，在这种情况下，如果在两个核心上先分别执行作业3和作业1，然后在作业1完成执行后，在空闲出来的核心上执行作业2，这样就能保证三个作业全都能满足各自的截止时限。这种非edf调度反而比上述的edf调度更优，因此命题得证。



# 单核调度：JDP调度

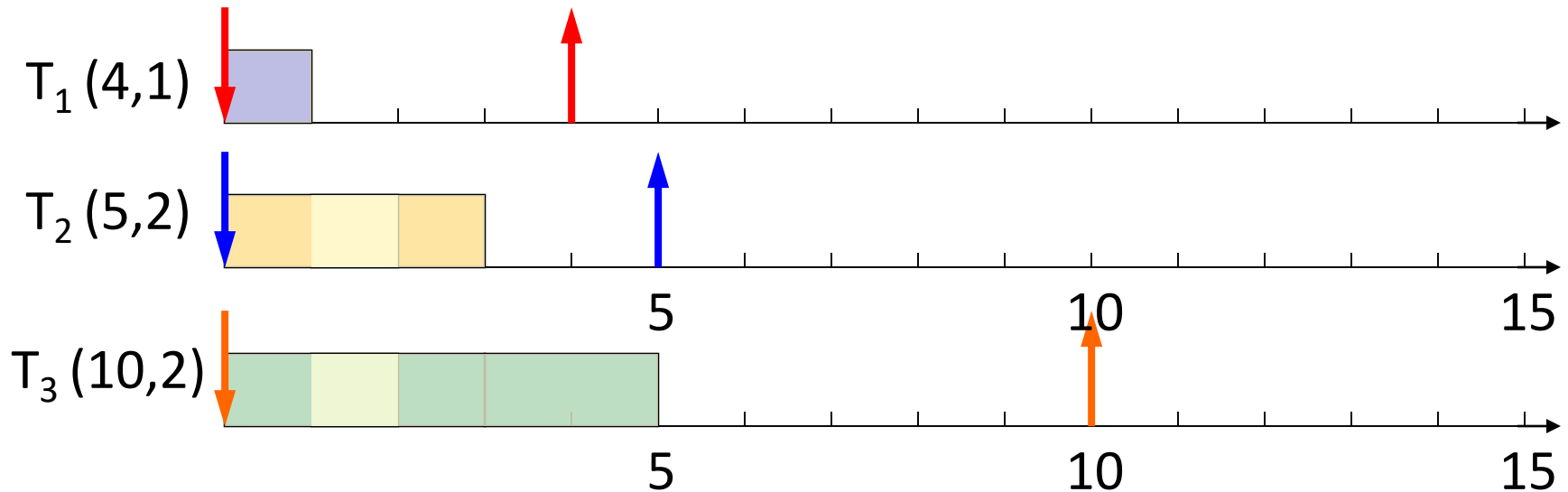
- ❑ 和JFP调度不同，JDP类调度算法给作业分配优先级是随时间动态变化的。
- ❑ 最小松弛度优先（least-laxity-first）是其中著名的一个JDP调度算法
  - ❑ 松弛度越小，优先级别越高
  - ❑ 松弛度=到截止时间还剩的时间-当前剩余所需执行时间
  - ❑ 和EDF一样，LLF在单核系统上对偶发任务同样是最优的
- ❑ 和JFP以及TFP的不同：
  - ❑ 多数JFP和TFP是事件驱动（event-driven）的调度算法，例如作业释放或者完成，当需要时立马重新调度
  - ❑ LLF是因子时间驱动（quantum-driven）的，只有时间是调度因子Q的整数倍时进行调度，即每Q个时间单元调度器执行一次调度

## 单核调度：TFP调度

- TFP调度算法下，每个任务释放的所有作业在运行时拥有相同的优先级，这也是Task-level的意义所在，在任务层面固定优先级，使一个任务释放的所有作业优先级相同
- 最出名和广为使用的TFP调度是RM调度算法（速率单调）

# RM (Rate Monotonic)

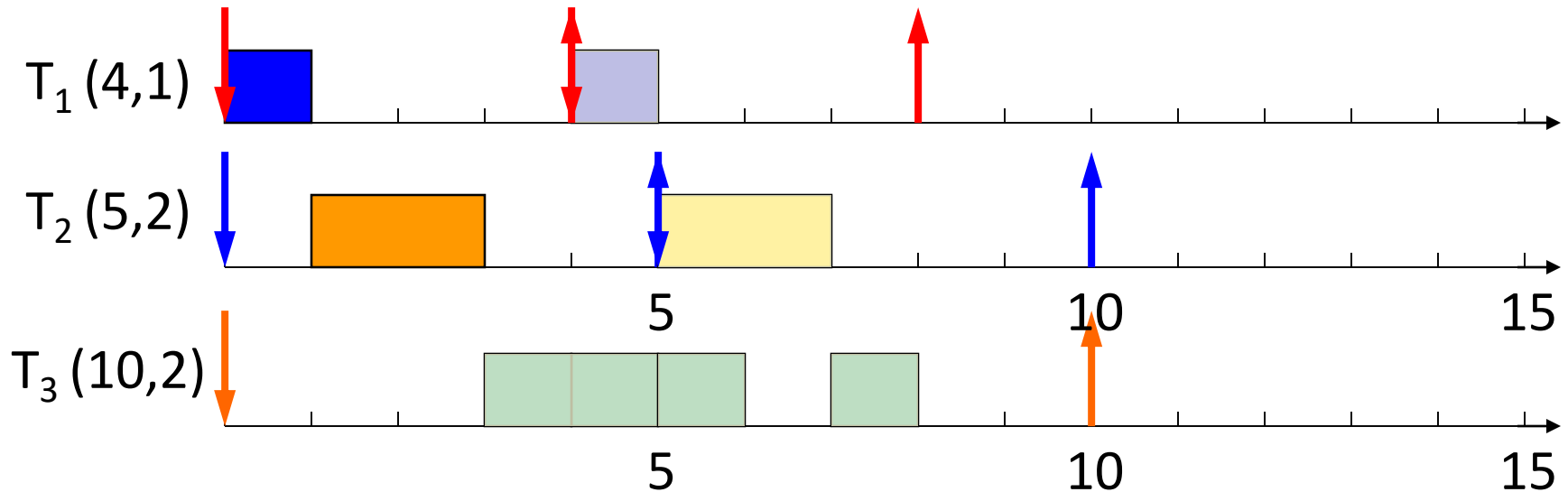
- TFP调度
- 根据任务的周期来分配优先级
- 周期更短的任务优先级更高，同一任务的作业优先级一致





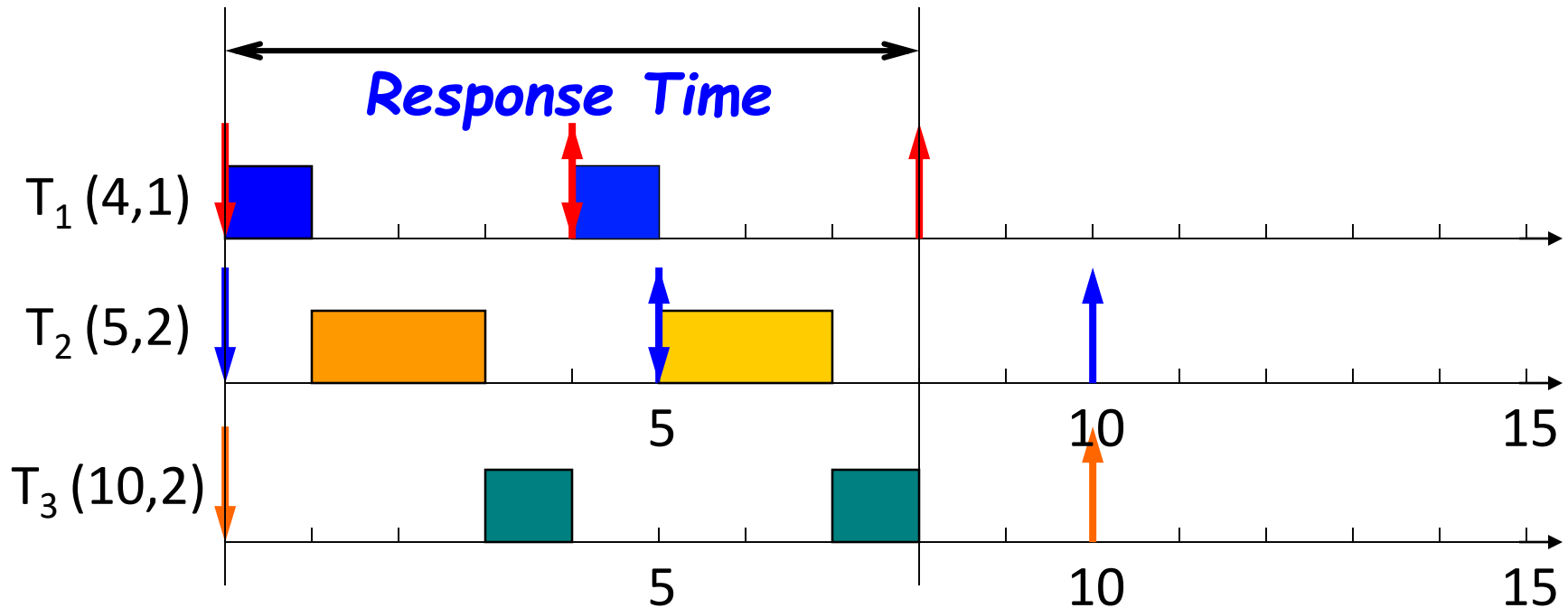
# RM (Rate Monotonic)

- TFP调度
- 根据任务的周期来分配优先级
- 周期更短的任务优先级更高，同一任务的作业优先级一致



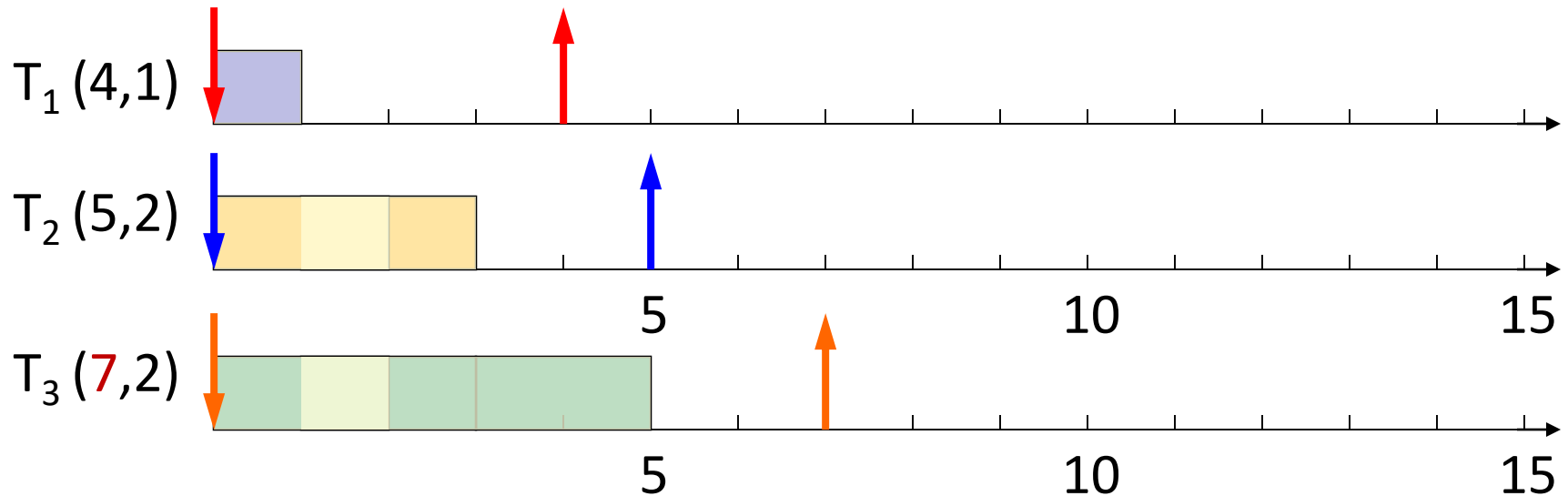
# RM (Rate Monotonic)

- 响应时间 ( response time )



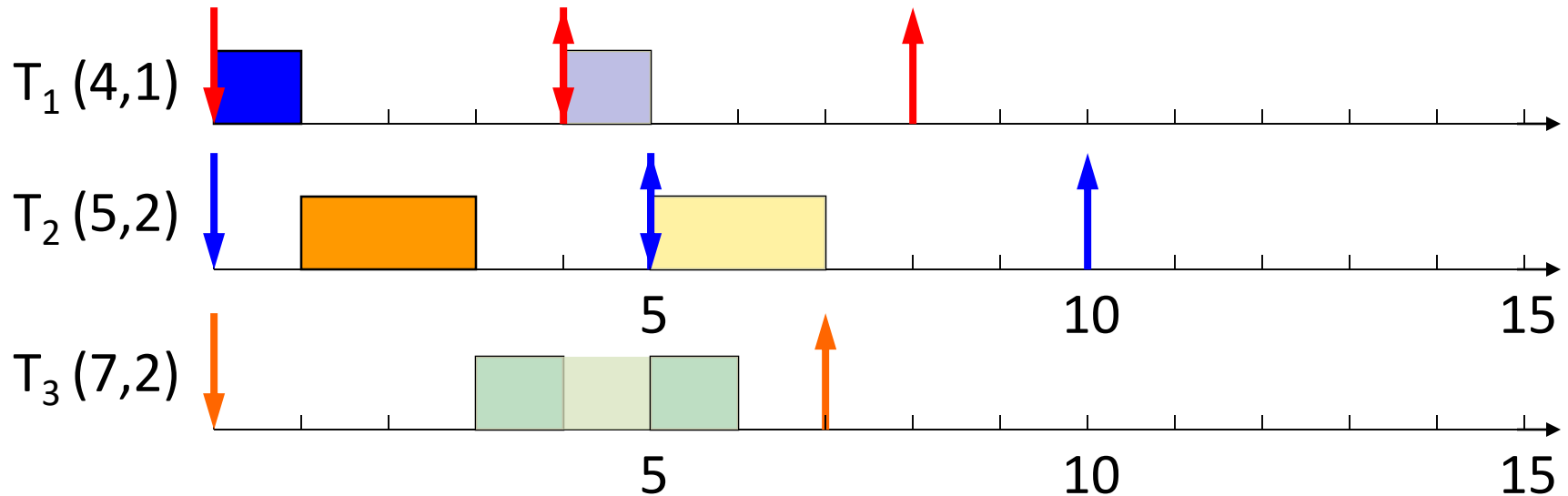
# RM (Rate Monotonic) – Another Task Set

- TFP调度
- 根据任务的周期来分配优先级
- 周期更短的任务优先级更高，同一任务的作业优先级一致



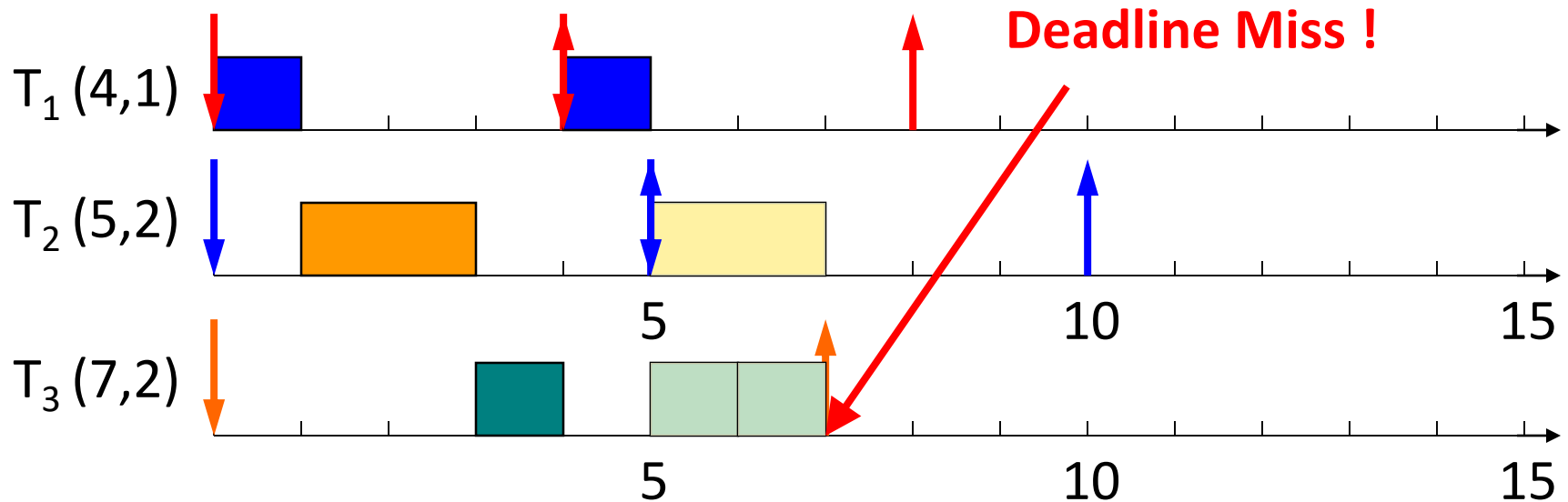
# RM (Rate Monotonic) – Another Task Set

- TFP调度
- 根据任务的周期来分配优先级
- 周期更短的任务优先级更高，同一任务的作业优先级一致



# RM (Rate Monotonic) – Another Task Set

- TFP调度
- 根据任务的周期来分配优先级
- 周期更短的任务优先级更高，同一任务的作业优先级一致



## RM调度算法（速率单调）性质

- ❑ TFP调度算法下，每个任务释放的所有作业在运行时拥有相同的优先级，这也是Task-level的意义所在，在任务层面固定优先级，使一个任务释放的所有作业优先级相同
- ❑ 最出名和广为使用的TFP调度是RM调度算法（速率单调）
  - ❑ 该调度下，按照任务的周期长短的顺序来赋予优先级，周期短的优先级高，周期相同时，看ID，ID小的优先级高
  - ❑ RM调度在单核情况下，即使对于隐式截止期限的周期任务而言，也不是最优的

## RM调度算法（速率单调）性质

$$\square U_b(n) = n * (2^{\frac{1}{n}} - 1)$$

$\square n$ : 任务的个数

$$\square U_b(n) = 0.828$$

$$\square U_b(n) \geq U_b(\infty) = \ln 2 = 0.693$$

$\square U \leq U_b(n)$ 是一个充分条件，但不是必要条件

$\square$ 即使CPU没有满载，也不能确保可调度性

$\square$ 算法开销小：一旦确定了任务集，任务的优先级就固定下来，不再改变

$\square$ 简单易实现

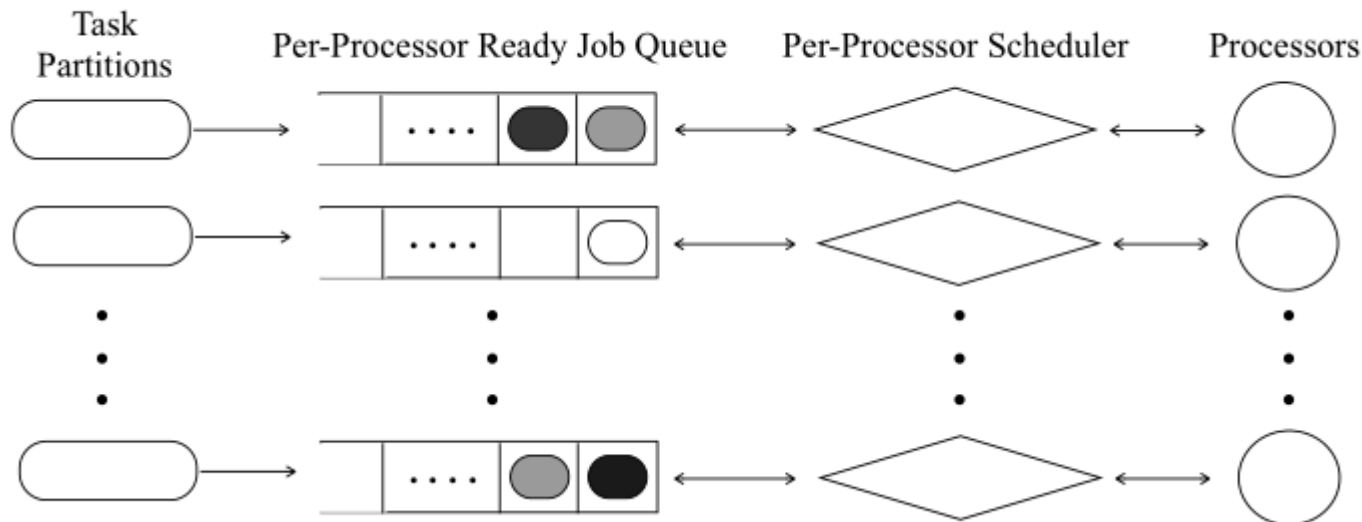
# 多核调度

- ❑ 多核调度算法可以分为三个大类：
  - ❑ 基于划分 (partition) 的多核调度
  - ❑ 基于全局 (global scheduling) 的多核调度
  - ❑ 基于簇 (cluster) 的多核调度
- ❑ 其中基于全局的多核调度还可以和单核调度类似，划分为三个更小的子类
  - ❑ 全局TFP调度
  - ❑ 全局JFP调度
  - ❑ 全局JDP调度



# 多核调度: Partition

- ❑ 任务被静态划分到不同的核心上，即每个任务被绑定到专有的核心上，不会迁移到其他核心执行
- ❑ 不同核心上可以采用不同的调度算法
- ❑ 需要确保分配给核心的任务的总利用率满足该核心上选用算法的可调度性条件
  - ❑ 例如周期任务系统有1, 2, 3, 4, 5号任务核心数为2，其中123划分为核心1, 45划分为核心2，其中核心1上运行EDF调度器，核心2上运行RM调度器
  - ❑ 则需要满足123的利用率之和小于1, 45任务利用率之和小于0.693

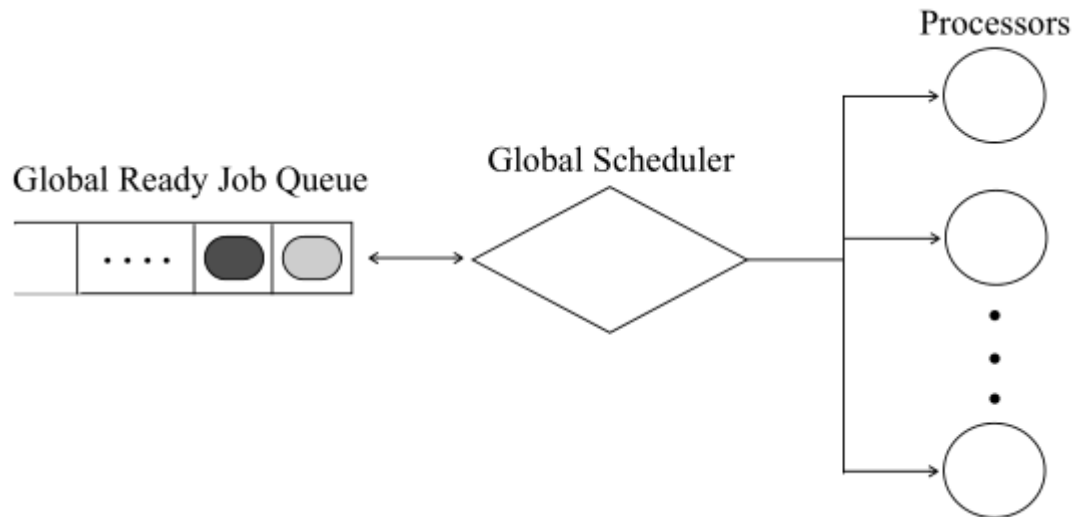


# 多核调度: Partition

- ❑ 将任务静态划分到不同的核心上可以采用各种启发式装箱算法 (bin-packing heuristics)
  - ❑ First-fit
  - ❑ Best-fit
  - ❑ Worst-fit
- ❑ First-Fit: 检查所有的核心, 找到第一个剩余capacity能够放下当前任务的核心并将该任务绑定到该核心, 否则则开启新的核心。
- ❑ Best-fit: 从全部核心中找出能满足任务利用率要求的, 且剩余capacity最小的核心并将该任务绑定到该核心
- ❑ Worst-fit: 从全部核心中找出能满足任务利用率要求的, 且剩余capacity最大的核心并将该任务绑定到该核心

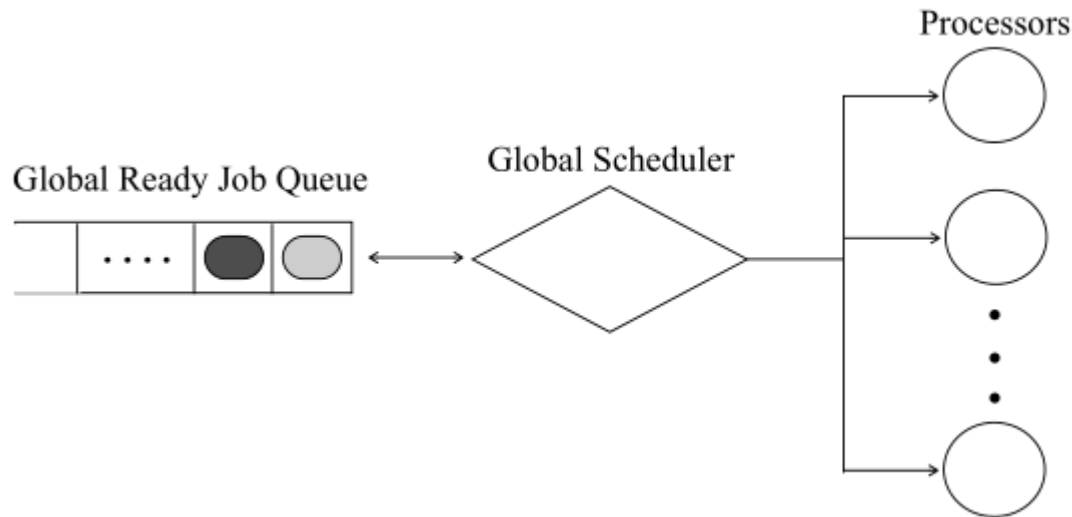
## 多核调度：全局调度

- 和基于划分的调度不同，在全局调度下，一个全局预备队列用来存放就绪的作业，允许作业在任何时刻从一个核心迁移到另一个核心执行



## 多核调度：全局TFP调度

- ❑ 在任意时刻，就绪队列中最多有 $m$ 个最高优先级的就绪作业被放在 $m$ 个核心上执行
- ❑ 在GTFP调度下，任务被分配固定的优先级，作业从释放它的任务中继承优先级

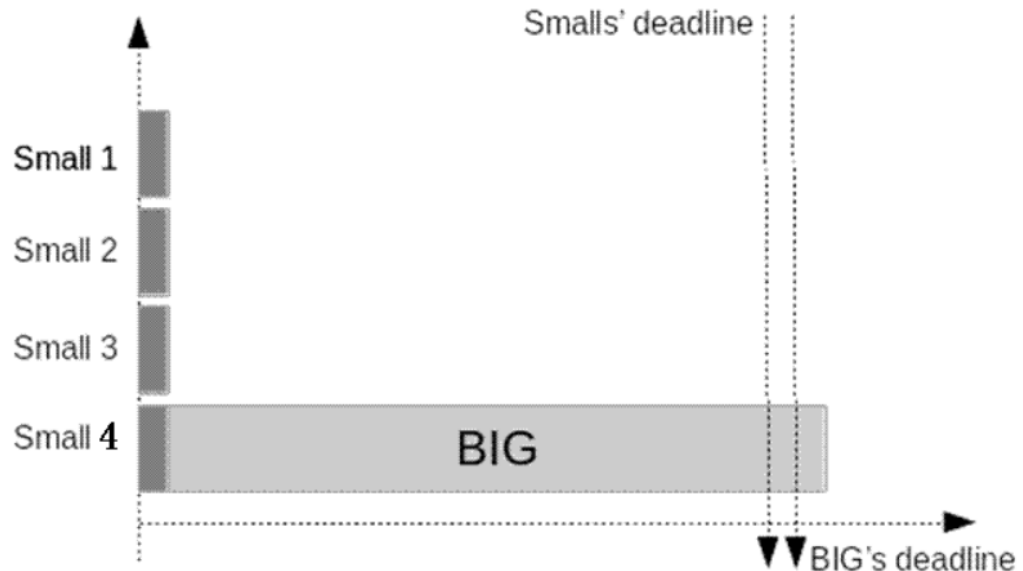


## 多核调度：全局TFP调度——GRM

- Dhall 和 Liu首先研究并证明了RM调度算法在多核系统上调度HRT周期任务不是最优的，而多核系统下的RM调度就属于GTFP
- 他们举出一个硬实时周期任务系统的例子，其总利用率接近于1，但是在 $m$ 核心上却不能被RM调度，这被称作Dhall效应，这代表RM可以造成接近 $m-1$ 的性能损失（capacity loss）。

# Dhall效应

- 构造一个轻负载：调度器要面对的是4个“小活”和一个“大活”，“小活”的运行时间是1ms，周期是999ms，“大活”的运行时间和周期都是1000ms。在这种场景下，系统的C利用率是1.004。
- 我们注意到1.004是远小于4的，因此直观上感觉调度器是可以很好的调度这四个任务，但实际上不管是RM还是单核上表现最优的EDF调度器，在多核上都会出现问题。
- 如果所有任务同时释放，4个小活将会被调度在4个CPU上，这时候，“大活”只能在“小活”运行完毕之后才开始执行。因此“大活”的deadline不能得到保障，从而产生Dhall效应（Dhall's effect）。



## 多核调度：GJFP调度

- JFP调度算法的global版本，其中GEDF算法是最为知名的一个，研究的也最多。
- 在GEDF调度下，作业按照edf标准赋予优先级，然后选择 $\min(m, \alpha)$ 个拥有最高优先级的作业分配执行，其中 $\alpha$ 是就绪的作业个数。
- 一些基于GEDF的针对SRT和HRT的可调度性测试已经被推导出来

# GEDF的SRT可调度性测试

- 2005年, Devi和Anderson: 任何总利用率小于等于 $m$ 的零星任务都可以被抢占式和非抢占式GEDF下以有限的延迟调度在 $m$ 个核心上, 因此, GEDF是SRT最优的。
- 可调度性测试: 总利用率 $\sum u_i \leq m$
- 更进一步, 在2008年, Devi和Anderson证明了:

- 非抢占GEDF: tardiness bounds =  $\frac{\sum_{T_k \in \varepsilon_{\max}(m)} e_k - e_{\min}}{m - \sum_{T_k \in U_{\max}(m-1)} u_k} + e_i$

- GEDF: tardiness bounds =  $\frac{\sum_{T_k \in \varepsilon_{\max}(m-1)} e_k - e_{\min}}{m - \sum_{T_k \in U_{\max}(m-1)} u_k} + e_i$



# GEDF的HRT可调度性测试

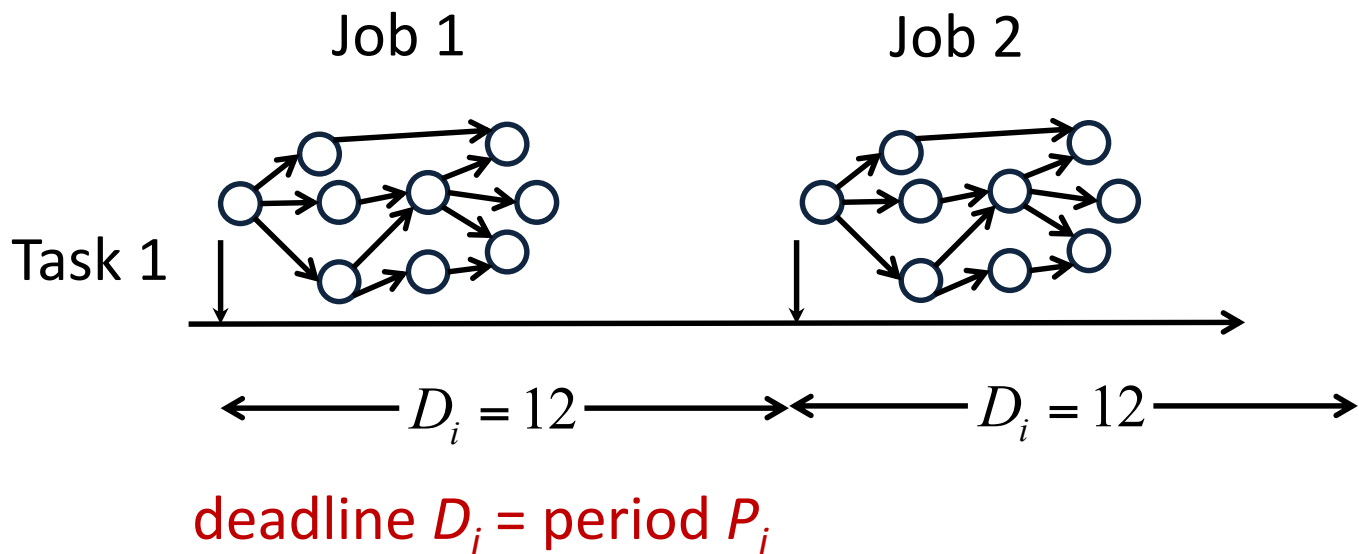
- 2003, Goossens et al: 如果  $U_{max} \leq m - (m - 1) * u_{max}$  则隐式期限偶发任务集  $T$  在GEDF下HRT可调度的。
- 2005, 进一步扩展之: 如果  $\delta_{max} \leq m - (m - 1) * \delta_{max}$  则任意期限偶发任务集  $T$  在GEDF下HRT可调度的

# GJDP调度

- ❑ 1996年, Baruah等人提出了第一个全局多核实时调度算法公平调度 proportionate fair (pfair) 算法
- ❑ Pfair对于隐式截止期限的周期任务是HRT最优的
- ❑ Pfair算法是一个时间片驱动的调度算法, 区别于事件驱动算法, 例如EDF和RM
- ❑ Pfair算法将任务被划分成多个子任务, 每个子任务的执行时间长度即一个时间片的长度
- ❑ 所有等待调度的子任务按照调度优先级排列组成全局等待队列, 系统中有新的子任务到来时, 按其调度优先级插入到等到队列中, 在每个时间片到来时, 处理器检查等待队列, 选择优先级最高的若干个子任务调度。

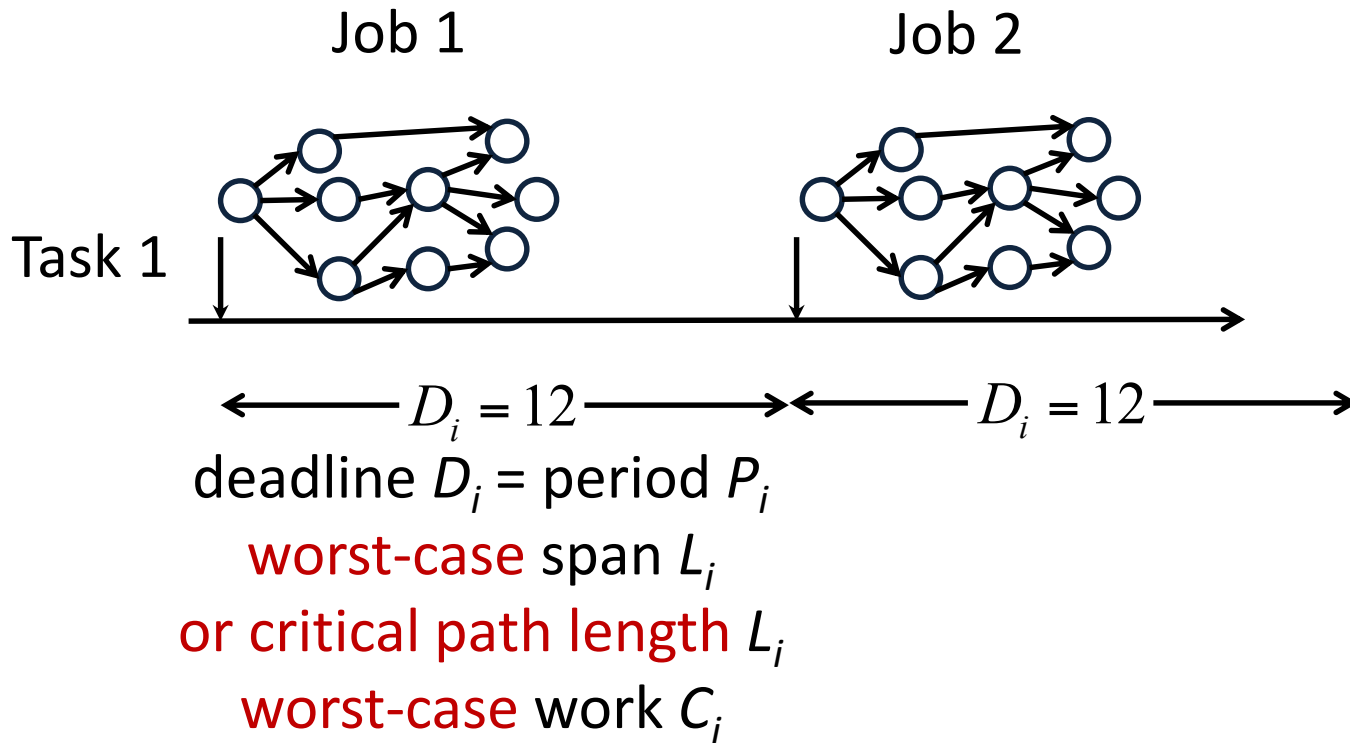
# 并行调度 (Parallel Scheduling)

□ 任务 (task) 周期性释放以DAG图表示的作业 (job)



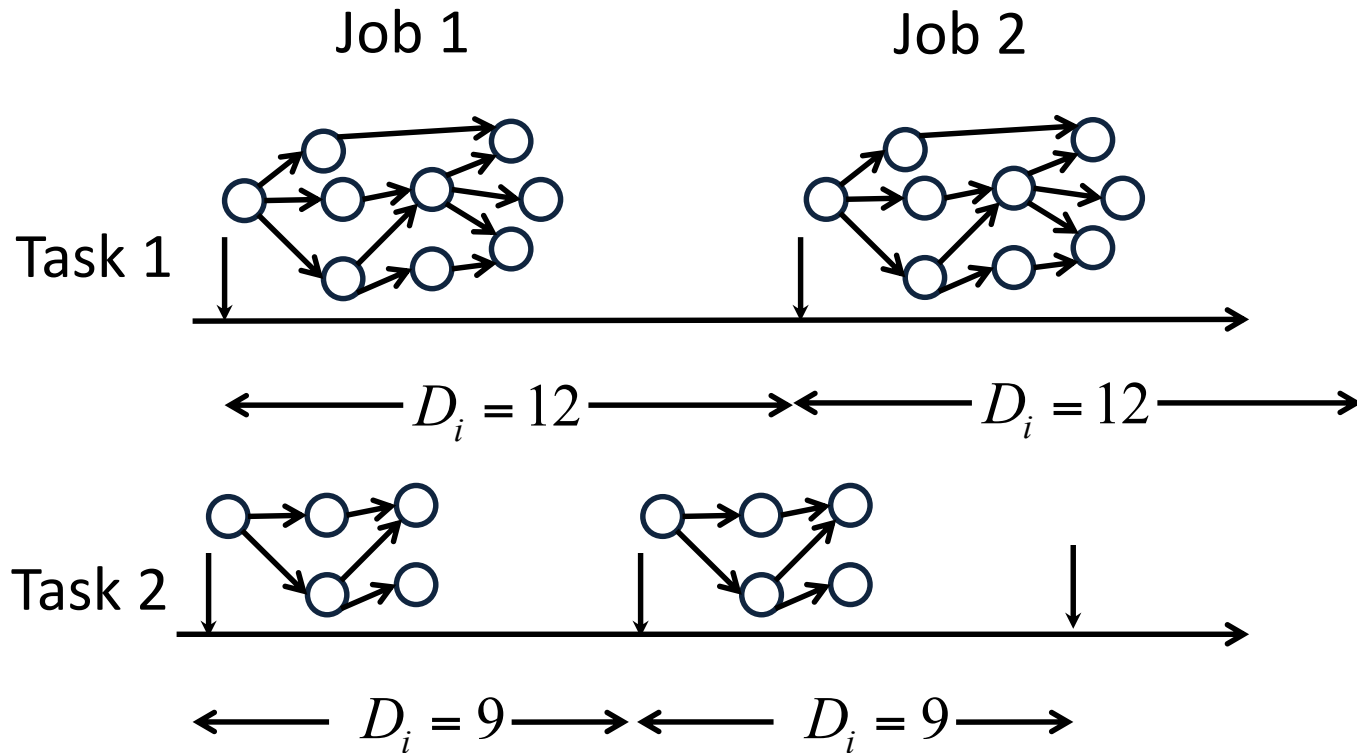
# 并行调度 (Parallel Scheduling)

□ 任务 (task) 周期性释放以DAG图表示的作业 (job)



# 并行调度 (Parallel Scheduling)

- ❑ 任务 (task) 周期性释放以DAG图表示的作业 (job)
- ❑ 在多核系统上被调度

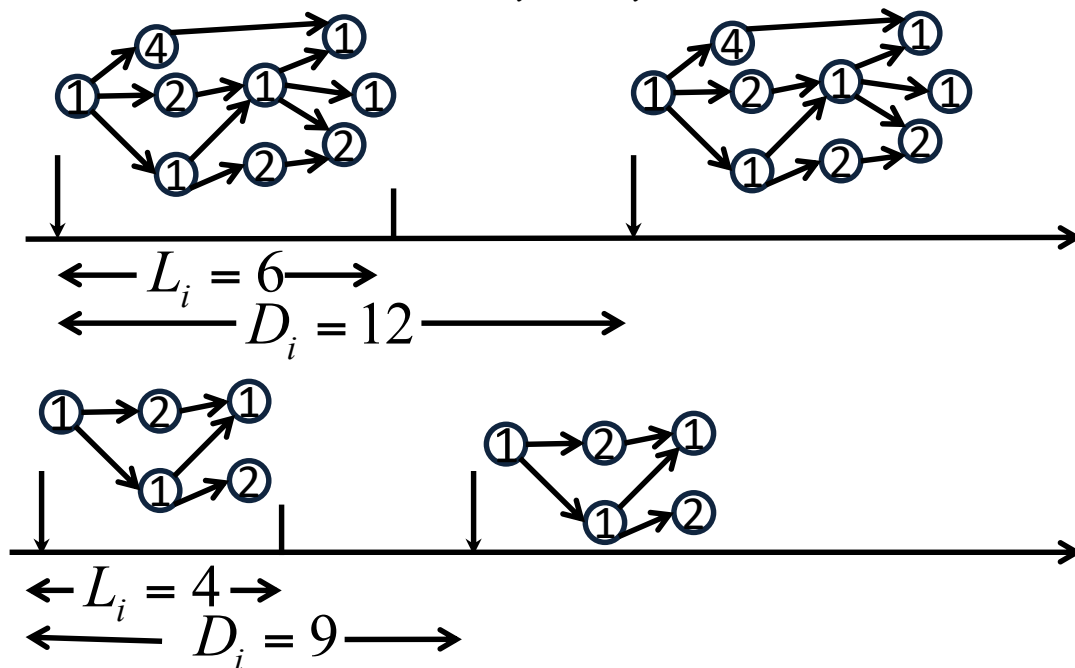


- ❑ 目的是满足各个时限约束

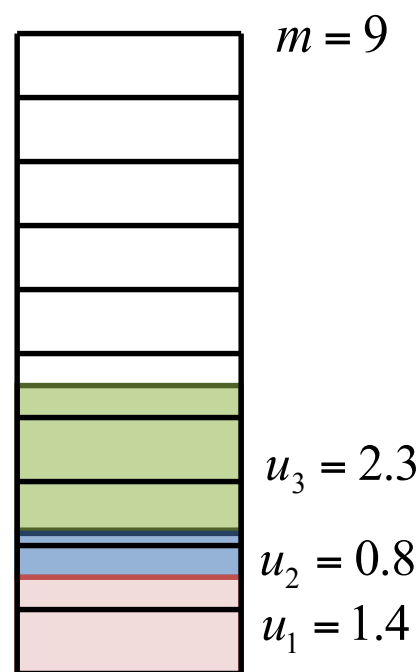
## capacity augmentation bound of $\alpha$

□ 一个调度器的 **capacity augmentation bound** 是  $\alpha$ ，如果他总能在  $m$  个核心上成功调度一个任务集合  $T$ ，并满足：

(a) For each task  $L_i \leq D_i / \alpha$



(b)  $U \leq m / \alpha$

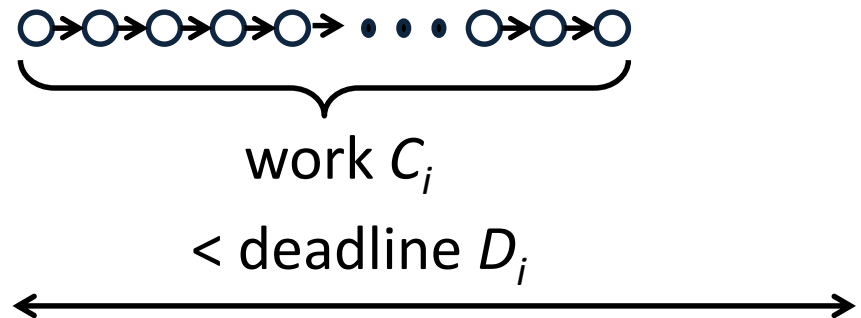


注意：该bound越小，说明调度算法表现越好。

# Federated Scheduling Intuition

Calculate the minimum #cores needed for a task to meet its deadline, if it is the only task in the system.

- Low-utilization ( $u_i < 1$ ,  $C_i < D_i$ )
  - Need at most 1 core to complete all work within the deadline



- High-utilization ( $u_i \geq 1$ ,  $C_i \geq D_i$ )
  - Need more than 1 core

# Federated Scheduling Algorithm

## HU Tasks

$C_1 = 31$	$C_2 = 22$
$L_1 = 6$	$L_2 = 3$
$D_1 = 18$	$D_2 = 7$
$u_1 = 1.72$	$u_2 = 3.14$

## LU Tasks

$C_3 = 15$	$C_4 = 30$
$L_3 = 4$	$L_4 = 30$
$D_3 = 17$	$D_4 = 40$
$u_3 = 0.88$	$u_4 = 0.75$



$$m = 12$$



# Federated Scheduling Algorithm


All low-utilization tasks share some cores together.

## HU Tasks

$C_1 = 31$	$C_2 = 22$
$L_1 = 6$	$L_2 = 3$
$D_1 = 18$	$D_2 = 7$
$u_1 = 1.72$	$u_2 = 3.14$

## LU Tasks

$C_3 = 15$	$C_4 = 30$
$L_3 = 4$	$L_4 = 30$
$D_3 = 17$	$D_4 = 40$
$u_3 = 0.88$	$u_4 = 0.75$



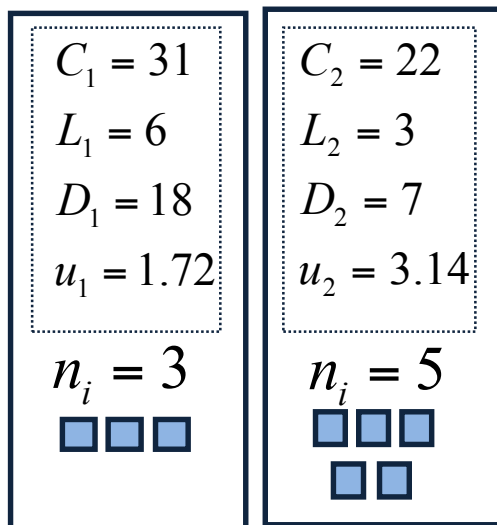
Treat as sequential tasks  
and use scheduler for  
sequential tasks.

# Federated Scheduling Algorithm

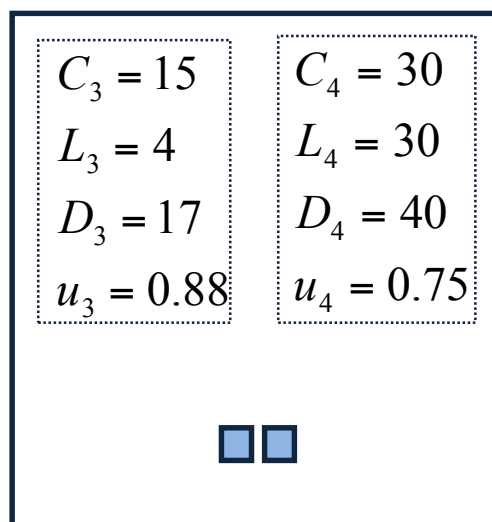
All low-utilization tasks share some cores together.

Assign dedicated cores to each high-utilization task.

HU Tasks



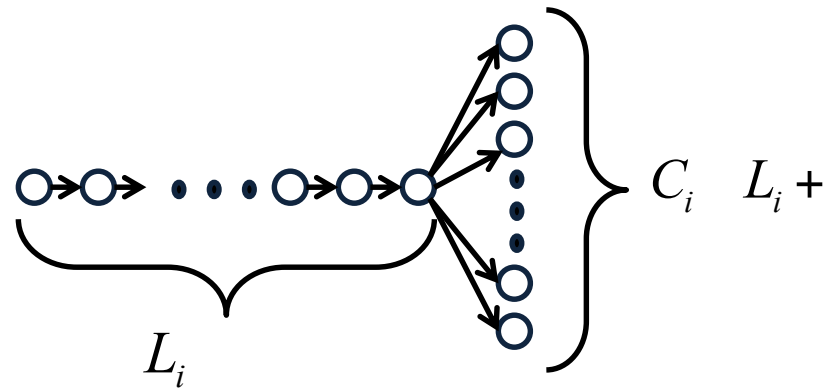
LU Tasks



$$n = \frac{C_i}{D_i} \frac{L_i}{L_i}$$

# Canonical DAG of High-Utilization Tasks

A chain of  $(L_i - \varepsilon)$  nodes +  $(C_i - L_i + \varepsilon)$  parallel nodes



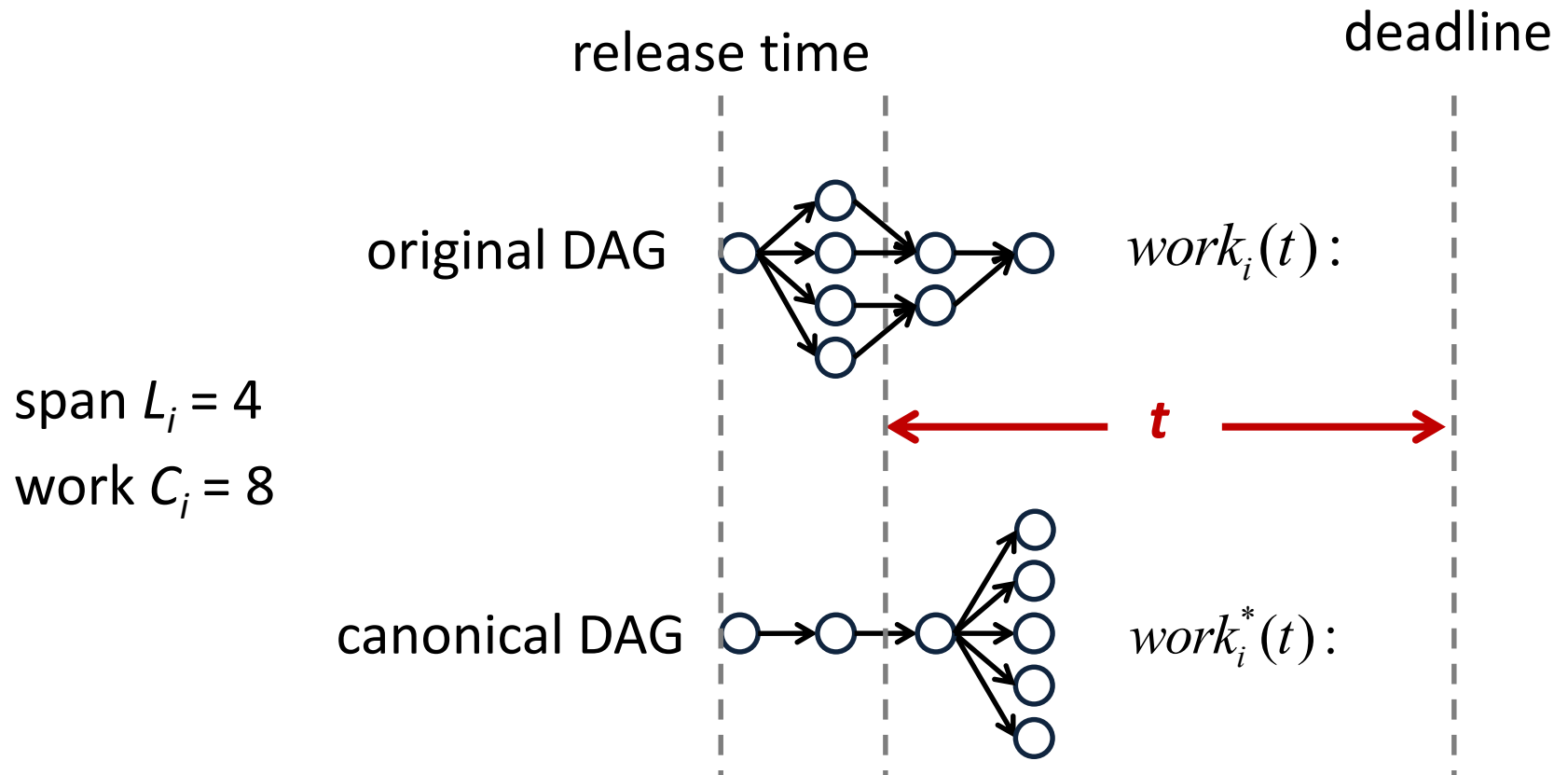
Same span  $L_i$  and work  $C_i$   
with the original DAG

# Canonical DAG of High-Utilization Tasks

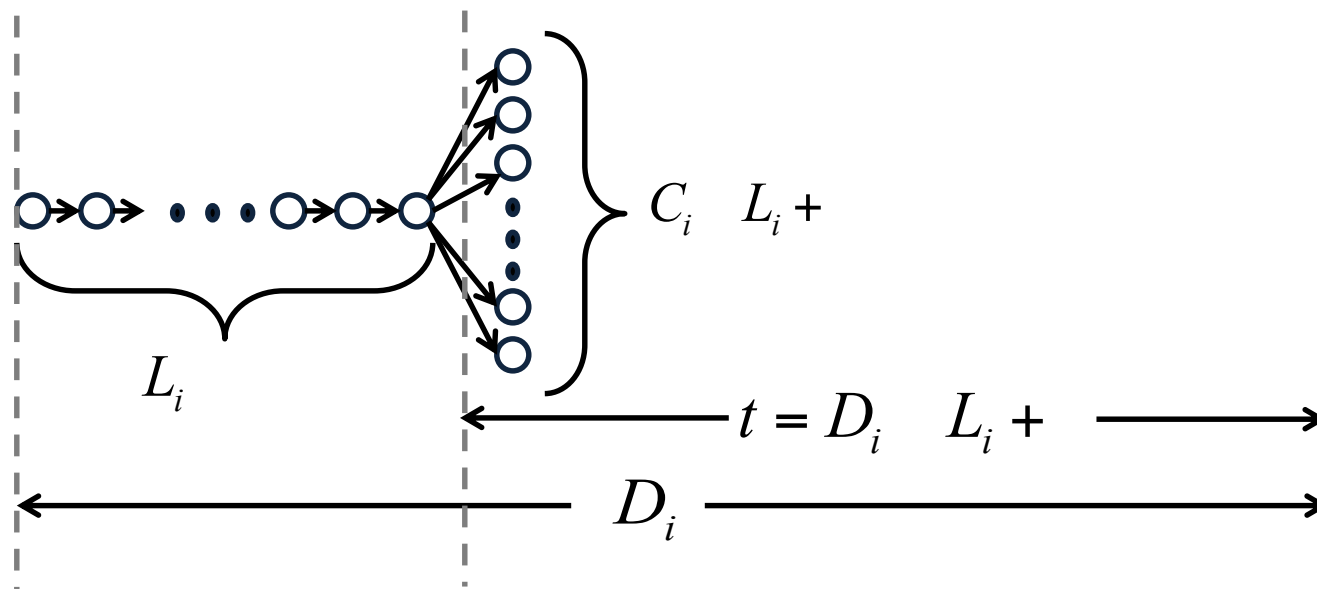
A chain of  $(L_i - \varepsilon)$  nodes +  $(C_i - L_i + \varepsilon)$  parallel nodes

The “Worst-Case” DAG – leave most work by the end

We can prove that for all  $t$ ,  $work_i^*(t) \geq work_i(t)$



# Cores Assignment to High-Utilization Tasks



For  $t = D_i L_i +$ ,  $work_i^*(t) = C_i L_i +$ .

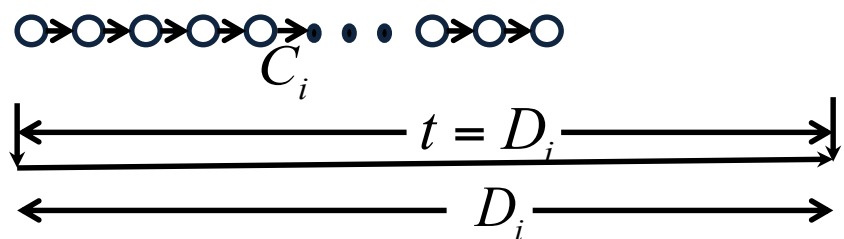
It needs  $n_i = \frac{C_i L_i +}{D_i L_i +} \leq \left\lceil \frac{C_i L_i}{D_i L_i} \right\rceil$  (for small enough  $\varepsilon$ )

We can prove that on  $n_i$  dedicated cores and using a *greedy* (work-conserving) scheduler, HU tasks never miss a deadline.

# Bound The Total Load Of Canonical Tasks

For all tasks, we bound  $\frac{work_i^*(t)}{t}$

For LU tasks,  $\frac{work_i^*(t)}{t} \quad C_i / D_i = u_i$

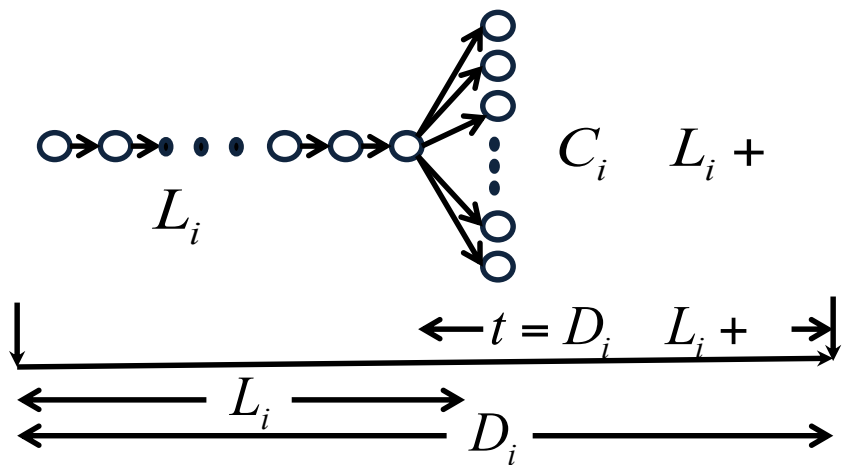


For HU tasks,

$$\frac{work_i^*(t)}{t} \quad \frac{C_i}{D_i} \frac{L_i +}{L_i +} \quad \frac{C_i}{D_i} \frac{L_i}{L_i}$$

If  $D_i \quad L_i,$

$$\frac{work_i^*(t)}{t} \quad \frac{C_i}{D_i} \frac{L_i +}{D_i /} = \frac{u_i}{1} \frac{L_i}{1 /} = \frac{u_i}{1}$$



Over all tasks,  $work_i^*(t) \quad \frac{U}{1} t$

# GEDF Has Capacity Augmentation Bound $\alpha \leq 2.618$

1. Bonifaci et. al [BMSW13] proved that  $\tau$  is schedulable by GEDF on  $m$  processors if

$$L_i \leq D_i \text{ and } \frac{work_i(t)}{m} \leq \frac{m+1}{m} t$$

2. We know that  $work_i(t) \leq work_i^*(t) \leq \frac{U}{1} t$

3. Therefore, the task set is schedulable if  $\frac{U}{1} \leq \frac{m}{m+1}$

4. We substitute  $\frac{U}{1} = \frac{m}{m+1}$  and solve for  $\alpha$  to get  $\frac{3 + 1/m + \sqrt{5}}{2} \leq \frac{2 + 1/m + 1/m^2}{2} \leq \frac{3 + \sqrt{5}}{2}$

# Capacity Augmentation Bound of Different Schedulers

Scheduler	Result
Any Scheduler	Lower bound: $> 2 \quad 1/m$
Federated	Upper bound: $2$ Optimal for large $m$
Global EDF	Upper bound: $(3 + \sqrt{5})/2$ Lower bound: $(3 + \sqrt{5})/2 \quad 2.618$
Global RM	Upper bound: $2 + \sqrt{3} = 3.73$



# 资源访问

- ❑ 我们需要知道，作业除了处理器之外，还会需要其他类型的资源用于执行
- ❑ 但迄今为止，我们只关注了处理器调度问题而忽略了这些需求
- ❑ 资源访问如何控制？
- ❑ 会产生哪些问题？

# 资源访问

- ❑ 资源与资源访问控制
- ❑ 优先级翻转 (Priority Inversion)
  - ❑ 优先级继承
  - ❑ 优先级天花板

# 资源与资源访问控制

- ❑ 假设系统包含若干类可重用的资源，分别为 $R_1, R_2, \dots, R_n$ 。资源 $R_i$ 有 $v_i$ 个无差别的部件。
- ❑ 通常，连续可重用的资源基于不可抢占原则赋予作业，并以互斥的方式使用
  - ❑ 当资源 $R_i$ 的一个部件被分配给作业后，在该作业释放这一部件之前，这个部件便不能被其他作业使用。
  - ❑ 这类资源包括互斥量，读/写锁，连接套接字，打印机和远程服务器等
  - ❑ 一个二进制信号量是一种仅包含一个部件的资源，而计数信号量则是包含多个部件的资源，一个带有五台打印机的系统具有五个部件的打印机资源，而一个互斥的写锁只有一个部件
- ❑ 有些资源可以同时供多个作业使用。将这种资源建模为拥有许多部件，部件之间互斥使用的资源。
  - ❑ 例如，将最多能被5个用户同时读的文件建模为拥有5个互斥部件的资源
  - ❑ 通过以这种方式建模共享资源，就可以对他们一视同仁

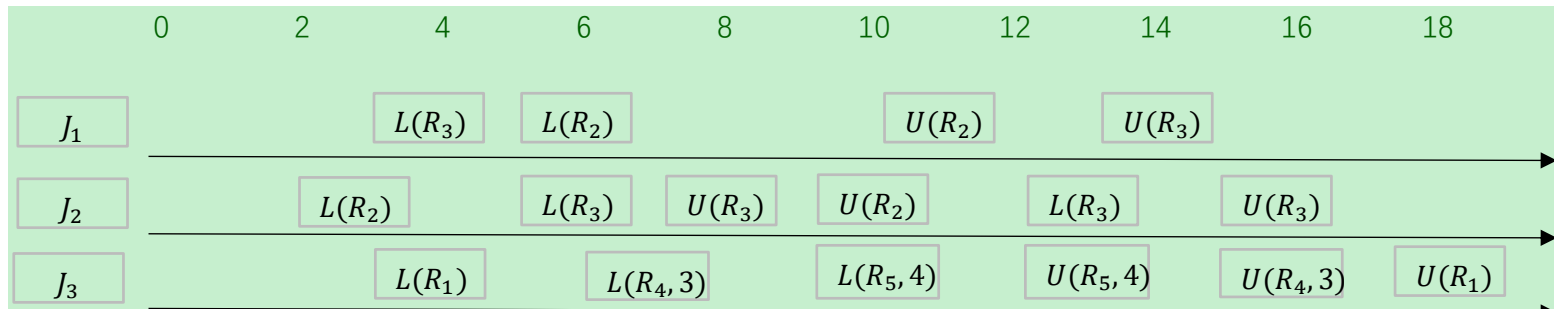
# 资源冲突和阻塞

- ❑ 假设使用基于锁的并发控制机制使作业对资源进行互斥访问，当一个作业要使用资源 $R_i$ 的 $B_i$ 个部件时，它会执行加锁（lock）来请求资源，用 $L(R_i, B_i)$ 来表示这个锁需求。
- ❑ 如果作业被给予了它请求的资源，作业就继续执行。如果作业不再需要该资源，它会执行解锁（unlock  $U(R_i, B_i)$ ）来释放资源
- ❑ 如果资源 $R_i$ 只有一个部件，那么使用更简单的 $L(R_i)$ 和 $U(R_i)$ 来分别表示加锁和解锁
- ❑ 根据约定俗成，称一个起始于加锁而结束于相匹配解锁的作业段为一个临界区，此外，资源释放按照后进先出的顺序进行，因此，重叠的临界区是严格嵌套的。

# 资源冲突和阻塞

□ 如下图所示：

- 给出三个作业，以及当作业单独从时刻0开始运行后执行加锁和解锁的时刻，资源123都仅有一个部件，资源45有多个部件
- 作业3有三个严格嵌套的重叠临界区。不包含在其他临界区内的临界区称之为最外临界区，例如作业3以 $L(R_1)$ 和 $U(R_1)$ 分界的临界区
- 使用 $[R, B; e]$ 来表示任一临界区，例如作业3中起始于 $L(R_5, 4)$ 的临界区可以表示为 $[R_5, 4; 3]$ ，特别地使用嵌套的方式来表示嵌套临界区，例如作业3的嵌套临界区可以表示为 $[R_1; 14[R_4, 3; 9[R_5, 4; 3]]]$



## 资源冲突和阻塞

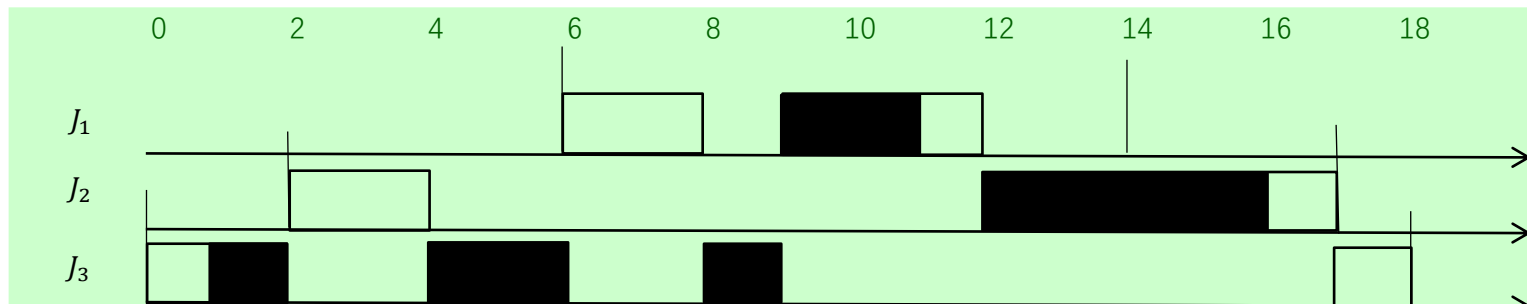
- ❑ 如果作业请求的某些资源是相同类型的，那么两个作业就会相互冲突
- ❑ 当某个作业请求的资源已为其他作业所占有时，作业会竞争同一资源
- ❑ 调度程序在资源的空闲部件不足以满足请求时通常会拒绝该请求。
- ❑ 如后所述，调度程序有时即使在被请求的资源部件空闲时，也会拒绝该请求，以阻止某些不希望的执行动作发生。
- ❑ 如果调度程序没有将资源 $R_i$ 的 $B_i$ 个部件分配给请求他们的作业，作业的加锁请求 $L(R_i, B_i)$ 就会失败
- ❑ 如果作业的加锁请求失败，作业将被阻塞并失去对处理器的占用，被阻塞的作业就会被移出就绪队列，直到调度程序将资源 $R_i$ 的 $B_i$ 个部件分配给它，这时作业变为非阻塞，被移回就绪队列，等待调度

# 资源冲突和阻塞

□ 下图说明了资源竞争的结果

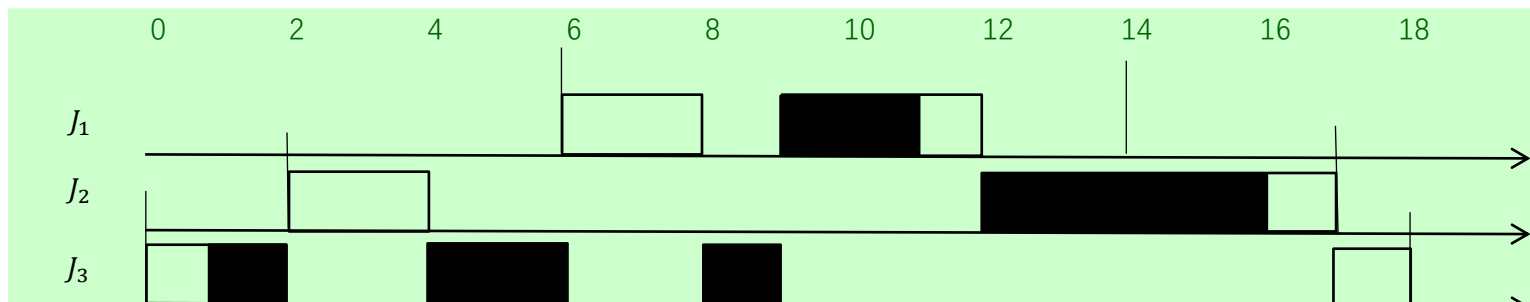
- 该例中有三个作业123，可行区间分别是  $(6, 14]$ ,  $(2, 17]$ ,  $(0, 18]$ ，每条时间线上的竖线代表释放时间和截止时间
- 基于EDF来调度，因此作业1优先级最高，其次是作业2作业3
- 所有三个作业只请求仅有一个部件的同一资源R，临界区间分别是  $[R; 2]$ ,  $[R; 4]$ ,  $[R; 4]$

□ 下面是对该调度段的描述，黑色代表处于其临界区



## 资源冲突和阻塞

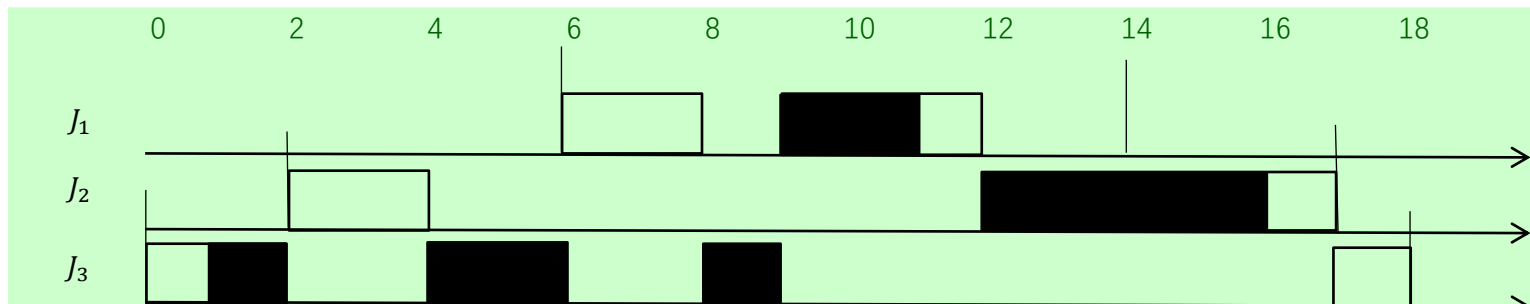
- ❑ 1. 在时刻0，只有作业3就绪，因而执行作业3。
- ❑ 2. 在时刻1，当作业3执行 $L(R)$ 时得到资源R。
- ❑ 3. 作业2在时刻2被释放，抢占了作业3并开始执行。
- ❑ 4. 在时刻4，作业2试图对R加锁，可由于作业3正在使用R，加锁请求失败，作业2被阻塞，而作业3重新获得处理器并开始执行。
- ❑ 5. 在时刻6，作业1就绪，抢占了作业3并开始执行
- ❑ 6. 当作业1执行到时刻8时，它要执行 $L(R)$ 来请求资源R，由于作业3仍然拥有该资源，因而作业1被阻塞，目前只有作业3就绪可执行，因而再次获得处理器并执行。





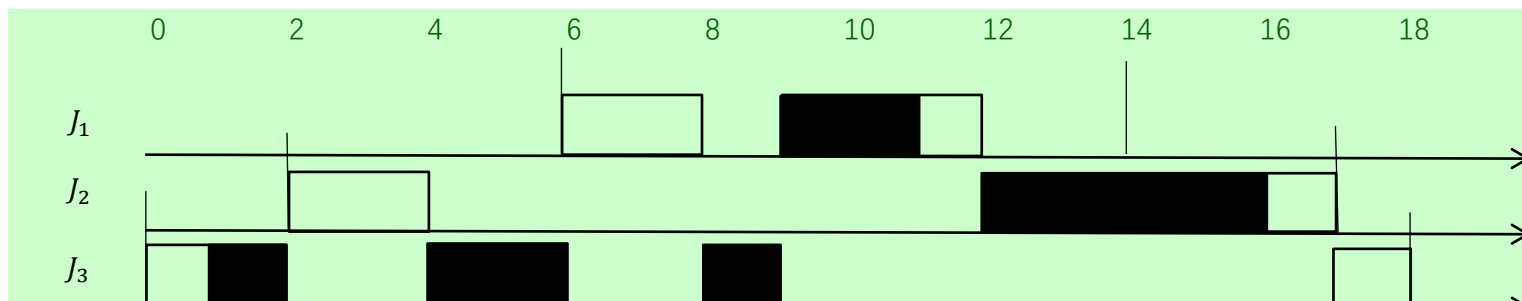
## 资源冲突和阻塞

- ❑ 7. 作业3的临界区在时刻9完成，当作业3执行  $U(R)$  时资源变为空闲。此时作业1和2都在等待该资源，但是前者优先级高，因而资源分配给作业1，作业1得以继续执行。
- ❑ 8. 作业1在时刻11释放该资源R，此时作业2解除阻塞，由于作业1拥有最高优先级，因而它得以继续执行。
- ❑ 9. 作业1在时刻12完成，由于作业2不再被阻塞并且比作业3优先级高，因而作业2获得处理器，并占有该资源开始执行，当它在时刻17完成时，作业3恢复执行至时刻18完成。



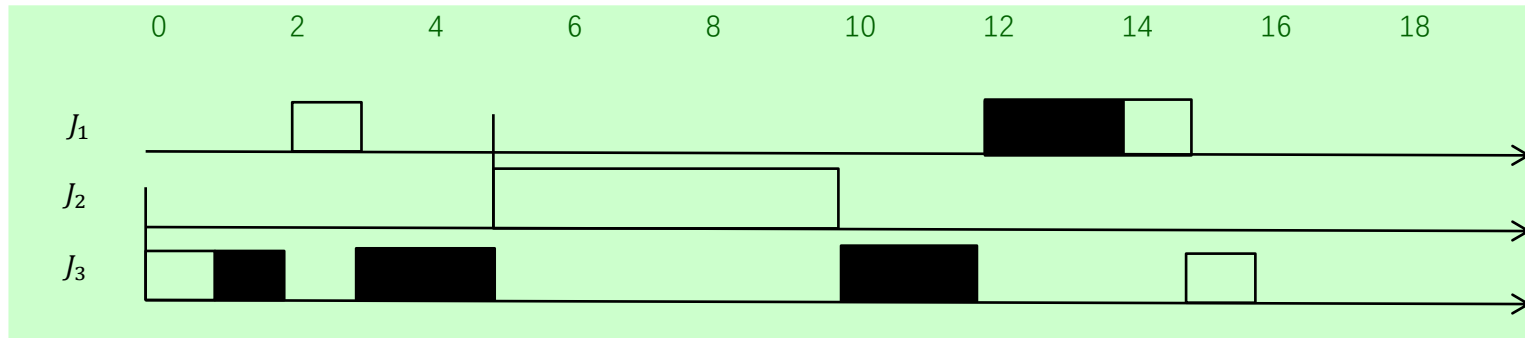
# 优先级翻转

- ❑ 我们知道，如果某些作业或部分作业的执行是不可抢占的，那么就可能发生较高优先级的就绪作业被阻塞直到较低优先级作业的不可抢占部分完成，这就是优先级翻转问题。
- ❑ 作业间的资源竞争也会导致优先级翻转，由于资源是基于不可抢占策略分配给作业，因而当作业发生冲突时，即使两个作业都是可抢占的，较高优先级的作业也可能被较低优先级作业阻塞。
- ❑ 阻塞造成的延迟可能使较高优先级的作业错过它的时限。
- ❑ 上述例子中，优先级最低的作业3获得资源R时，先是阻塞了作业2，然后阻塞了作业1，因此优先级逆转发生的区间是(4, 6]和(8, 9]
- ❑ 更糟糕的是，如果没有良好的资源访问控制，那么优先级翻转持续的时间将会无限期延长。



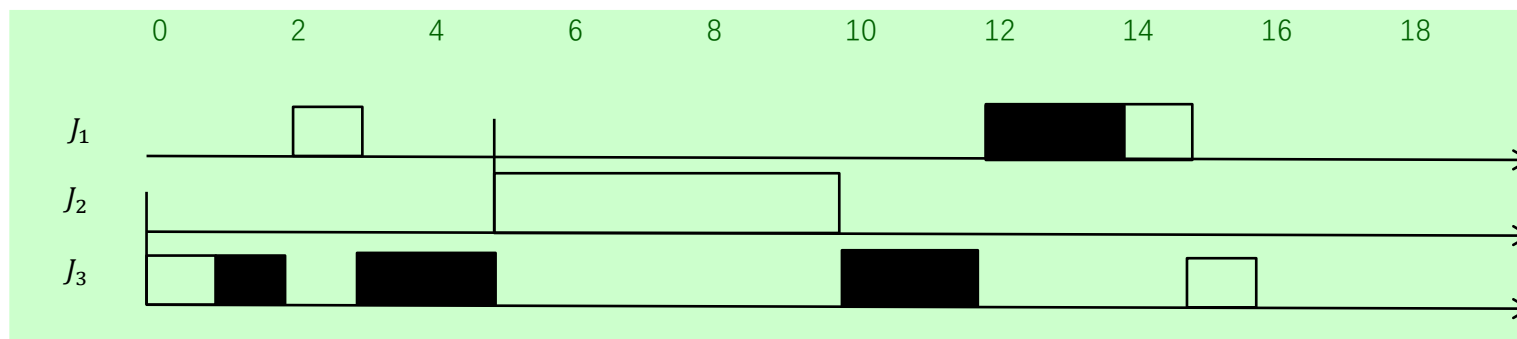
# 优先级翻转

- ❑ 如果没有良好的资源访问控制，那么优先级翻转持续的时间将会无限期延长，一个例子如下：
- ❑ 作业123中，作业1的优先级最高，作业3的优先级最低，其中作业2不申请资源R，作业1的临界区是 $[R; 2]$ ，作业3的临界区是 $[R; 5]$



# 优先级翻转

- ❑ 在时刻0，作业3就绪并执行，紧接着它请求资源R并继续执行
- ❑ 将资源R分配给作业3后，在时刻2，作业1变为就绪，作业1抢占了作业3并执行到时刻3，此时它请求资源R，但由于该资源正在被作业3使用，因而作业1被阻塞并开始了优先级逆转
- ❑ 当时刻5，比作业3优先级高但是比作业1优先级低的作业2被释放，由于作业2没有请求资源R，因而此作业抢占作业3并执行至完成
- ❑ 这样，作业2延长了优先级逆转的持续时间，在这种情况下，优先级逆转被认为是失控的
- ❑ 由于可以有任意多个优先级比作业1低，比作业3高的作业同时被释放，因而会进一步延长优先级逆转持续的时间，事实上，当优先级逆转失控时，作业可能被无限期阻塞。



# 优先级翻转：火星探路者事故

- ❑ 火星探路者（Mars Pathfinder, MPF）于1997年07月04日在火星表面着陆。几天后，开始出现系统复位、数据丢失的现象。
- ❑ 经过研发人员的分析，最后得出结论，就是因为系统里发生了优先级反转的问题。
- ❑ 其中有如下两个任务需要互斥访问共享资源“信息总线”：
  - ❑ T1：总线管理任务，高优先级（这里用T1表示），负责在总线上放入或者取出各种数据，频繁进行总线数据I/O，它被设计为最重要的任务
  - ❑ T6：数据收集任务，优先级低（这里用T6表示），它运行频度不高，只向总线写数据，并通过互斥信号量将数据发布到“信息总线”。
- ❑ 还有一个不需要访问该共享资源的任务T3，它是需要较长时间运行的通信任务，其优先级比T6高，比T1低
- ❑ 在很少情况下，如果通信任务被中断程序激活，并且刚好在总线管理任务（T1）等待数据收集任务（T6）完成期间就绪，这样T3将被系统调度，从而比它低优先级的数据收集任务T6得不到运行，因而使最高优先级的总线管理任务（T1）也无法运行，一直被阻塞在那里。在经历一定的时间后，看门狗观测到“总线”没有活动，将其解释为严重错误，并使系统复位。

# 优先级翻转：火星探路者事故解决方案

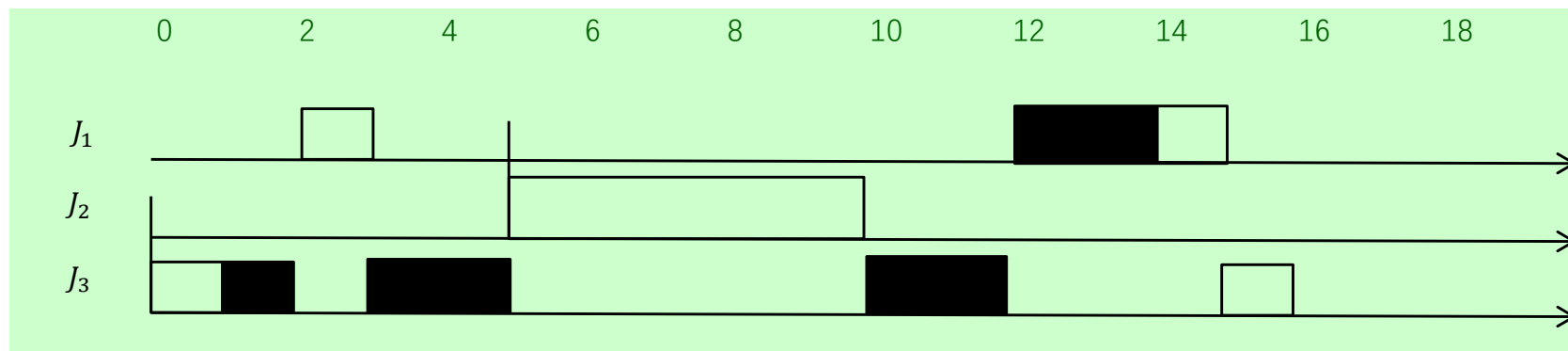
- ❑ 探路者采用的操作系统是风河（WindRiver）公司出品的一个实时操作系统VxWorks，我国的军工、自动化、航天等军工行业的实时控制系统基本上完全依赖VxWorks。
- ❑ VxWorks中采用了优先级继承协议来解决优先级翻转问题，VxWorks中，默认情况下该功能被关闭，所以就有可能发生优先级反转，也就导致了“火星探路者”中问题的发生。
- ❑ 另外还有一个防止优先级反转的是优先级天花板协议。
- ❑ 下面将介绍这两种资源共享访问协议。

# 优先级翻转 (Priority Inversion)

- ❑ 解决优先级翻转提出的两个资源共享访问协议
  - ❑ 优先级继承协议
  - ❑ 优先级天花板协议

# 优先级继承 (Priority Inheritance) 协议

- ❑ 优先级继承是指将低优先级任务的优先级提升到等待它所占有的资源的最高优先级任务的优先级. 当高优先级任务由于等待资源而被阻塞时, 此时资源的拥有者的优先级将会自动被提升。
- ❑ 下图是上面举过的一个优先级翻转的例子, 让我们来看一下如何使用优先级继承协议来解决优先级翻转这个问题

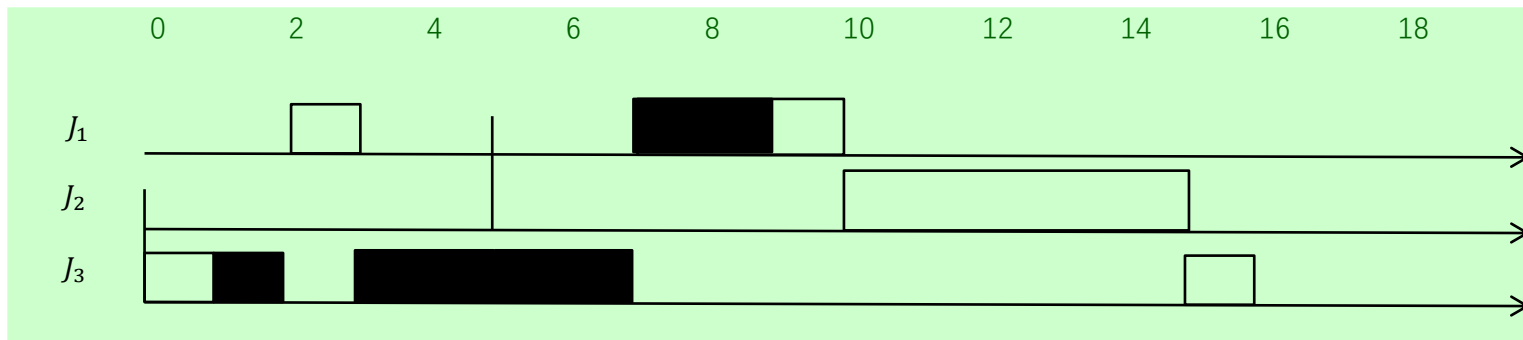




# 优先级继承 (Priority Inheritance) 协议

- ❑ 当作业1在时刻3请求资源R且被作业3阻塞时，作业3继承了作业1的优先级P1，当作业2在时刻5就绪时，由于他的优先级P2比作业3的优先级P1要低，因而无法抢占作业3的执行。
- ❑ 结果是作业3尽快地结束其临界区的执行后作业1也立即开始执行并尽快结束其临界区
- ❑ 这样就很大程度上减少了优先级翻转持续的时间

<https://en.wikipedia.org/wiki/LynxOS>



# 优先级天花板（Priority Ceilings）协议

- ❑ 优先级继承协议无法防止死锁，且无法将作业的阻塞时间降低至最少。
- ❑ 优先级天花板协议扩展了优先级继承协议，以防止死锁并进一步降低阻塞时间
  - ❑ 将申请某资源的任务的优先级提升到可能访问该资源的所有任务中最高优先级任务的优先级
- ❑ 事实上，有个定理如下表述：
  - ❑ 如果某处理器上可抢占的，优先级驱动的作业的系统对资源的访问由优先级天花板协议控制，那么永远不会发生死锁。
  - ❑ 正规证明可以参阅Sha L, Rajkumar R, Lehoczky J P. Priority inheritance protocols: An approach to real-time synchronization[J]. IEEE Transactions on computers, 1990, 39(9): 1175-1185.
- ❑ 感兴趣的同学可以进一步查阅相关资料