OSEK/VDX标准的产生

- 1993年5月，几家德国汽车制造商同意在适用于汽车且通用的实时分布式操作系统的规范化方面进行合作，产物就是OSEK(OSEK：Offense systeme und deren Schnit-stellen fur ide Elek im Kraftfahrzeug)。
- 与此同时，法国的PSA和Renault开发了一个类似的系统，该系统被成为VDX(VDX：Vehicle Distributed eXecu-tive)。
- 1994年，两项目合并，1995年，OSEK/VDX面世，译文大意是用于汽车电子的、带有接口的开放式系统。

OSEK/VDX是什么？

- OSEK/VDX是用于分布式实时结构的一组标准，它包含四个标准：操作系统（OS）、通信（COM）、网络管理（NM）和OSEK实现语言（OIL）。
- 虽然OSEK/VDX是欧洲汽车工业开发的，但它并不只是一个用于汽车的实时操作系统。基于这个标准的系统能够并且将要用于其他应用中，只要这些应用是被静态定义且需要一个紧凑的分布式实时系统。
- 我们所主要关心的是其中的操作系统标准部分。
- OSEK工作组于2000年11月推出OS Specification V2.1r1版本

OSEK的几个要点

- OS在单处理器上运行
- OS在启动时由用户配置指令生成，以后不支持任务的动态生成。
- OS提供的服务提供了标准接口，对于不同的处理器实现接口必须相同，即通常所说的OS的移植。
- 支持符合类（见后面详细介绍）和不同的调度策略。
- 几乎所有的API都返回一个StatusType类型，有几个例外。(StartOS()、ShutdownOS()、GetActiveApplicationMode()、EnterISR()、LeaveISR())
- 标准状态模式(API只返回E_OK)和扩展状态模式(可以返回错误码)，一般系统测试阶段采用扩展状态模式，发布的时候采用标准状态模式。
- 回调函数和应用程序模式。

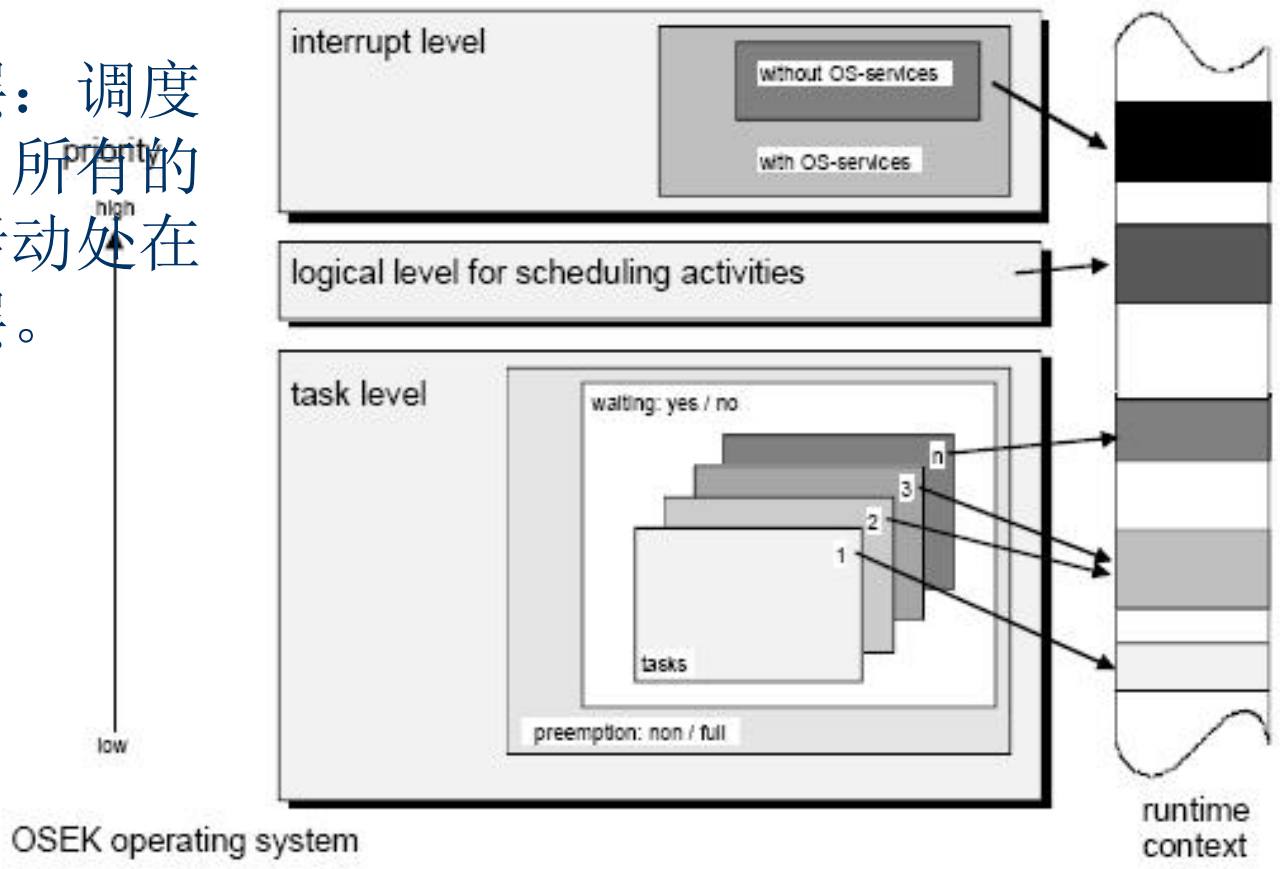
OSEK OS体系结构

- OS标准中定义的服务被两种实体来使用：中断服务程序(ISR)和任务(Task)
- 标准定义了三个处理级别：中断Level、调度Level、任务Level。以下分别翻译成：中断层、调度层、任务层。图示如下：

OSEK OS体系结构

- 优先级的划分必须满足下列条件，k……m分配给ISR，j分配给调度程序，0……i分配给任务，其中0≤i<j<k≤m

- 调度层：调度任务，所有的调度活动处在这一层。



优先级规则

- 中断的优先级要高于任务
- 中断处理层可以包含一个或多个中断优先级
- ISR的中断优先级是静态分配的
- ISR的优先级的分配与具体实现或硬件体系结构有关
- 对于任务的优先级和资源的天花板优先级来说，大的数字指较高的优先级。For task priorities and resource ceiling-priorities bigger numbers refer to higher priorities.
- 任务的优先级由用户（应用程序开发人员）指定

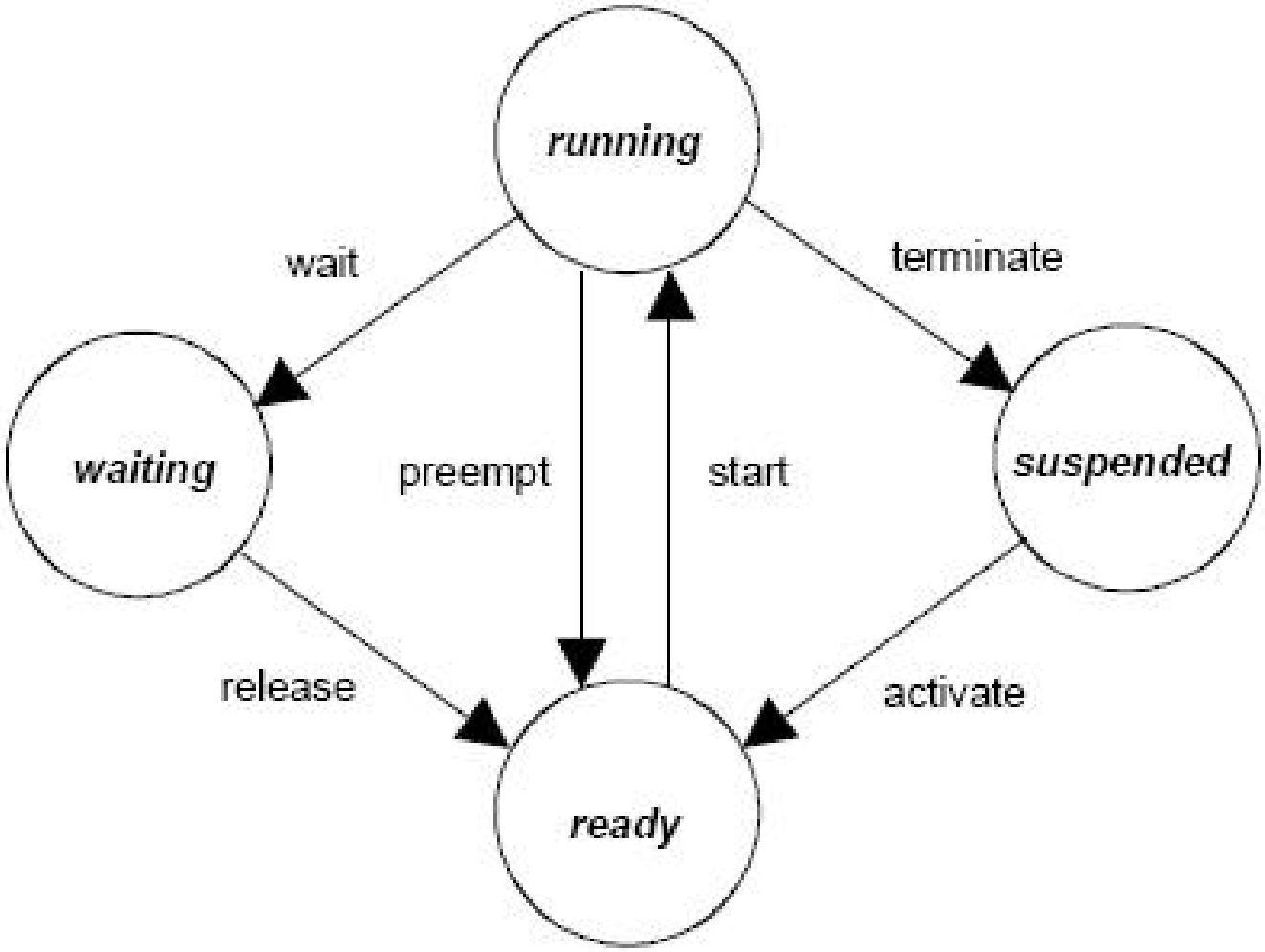
OSEK体系结构

- 在OS标准里，符合类也算作了体系结构的一部分
- 四个符合类：
  - BCC1（基本符合类1）
  - BCC2（基本符合类2）
  - ECC1（扩展符合类1）
  - ECC2（扩展符合类2）
- 因牵涉到很多任务的具体概念，每个符合类的定义放在任务之后再说。

任务—任务的类型

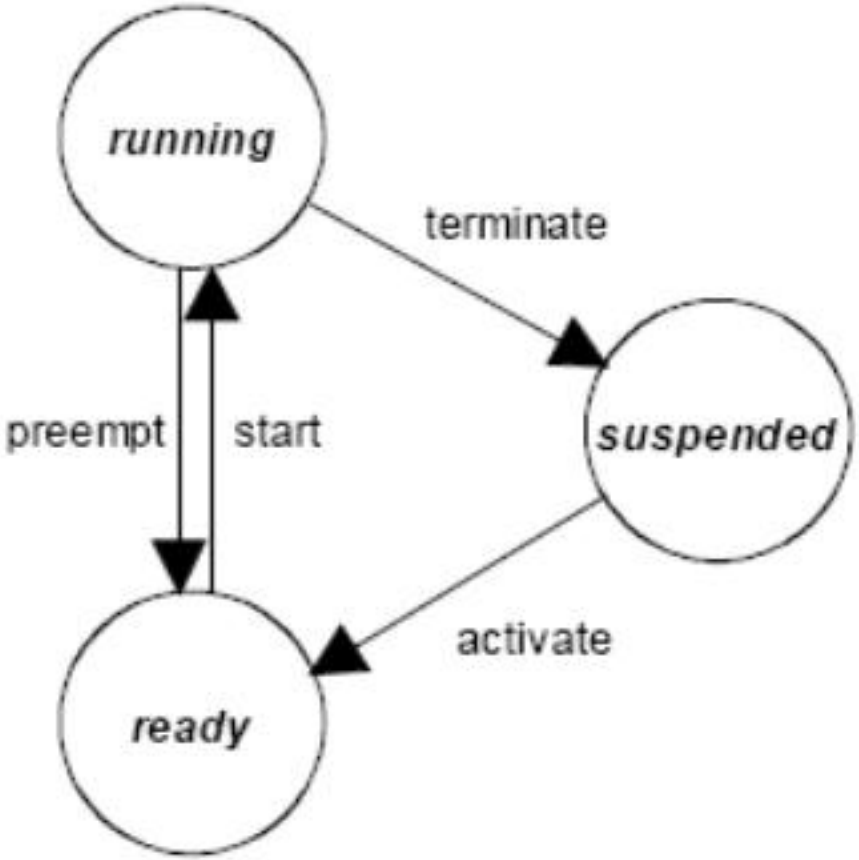
- 任务有两种类型：基本任务(BT)和扩展任务(ET)。
- 基本任务只在以下三种情况释放CPU：
  - 一、任务结束
  - 二、OS切换到高优先级的任务去运行
  - 三、产生中断，CPU去执行ISR。
- 区别于基本任务，扩展任务可调用WaitEvent()服务进入等待(Waitiing)状态，等待状态下的任务释放CPU，允许原本比它低优先级的任务去执行。

扩展任务状态转换


running
wait
terminate
waiting
preempt
start
suspended
release
activate
ready

Transition	Former state	New state	Description
activate	*suspended*	*ready*	A new task is set into the *ready* state by a system service. The OSEK operating system ensures that the execution of the task will start with the first instruction.
start	*ready*	*running*	A *ready* task selected by the scheduler is executed.
wait	*running*	*waiting*	The transition into the waiting state is caused by a system service. To be able to continue operation, the *waiting* task requires an event.
release	*waiting*	*ready*	At least one event has occurred which a task has *waited* for.
preempt	*running*	*ready*	The scheduler decides to start another task. The *running* task is put into the *ready* state.
terminate	*running*	*suspended*	The *running* task causes its transition into the *suspended* state by a system service.

基本任务状态转换


running
terminate
preempt
start
suspended
activate
ready

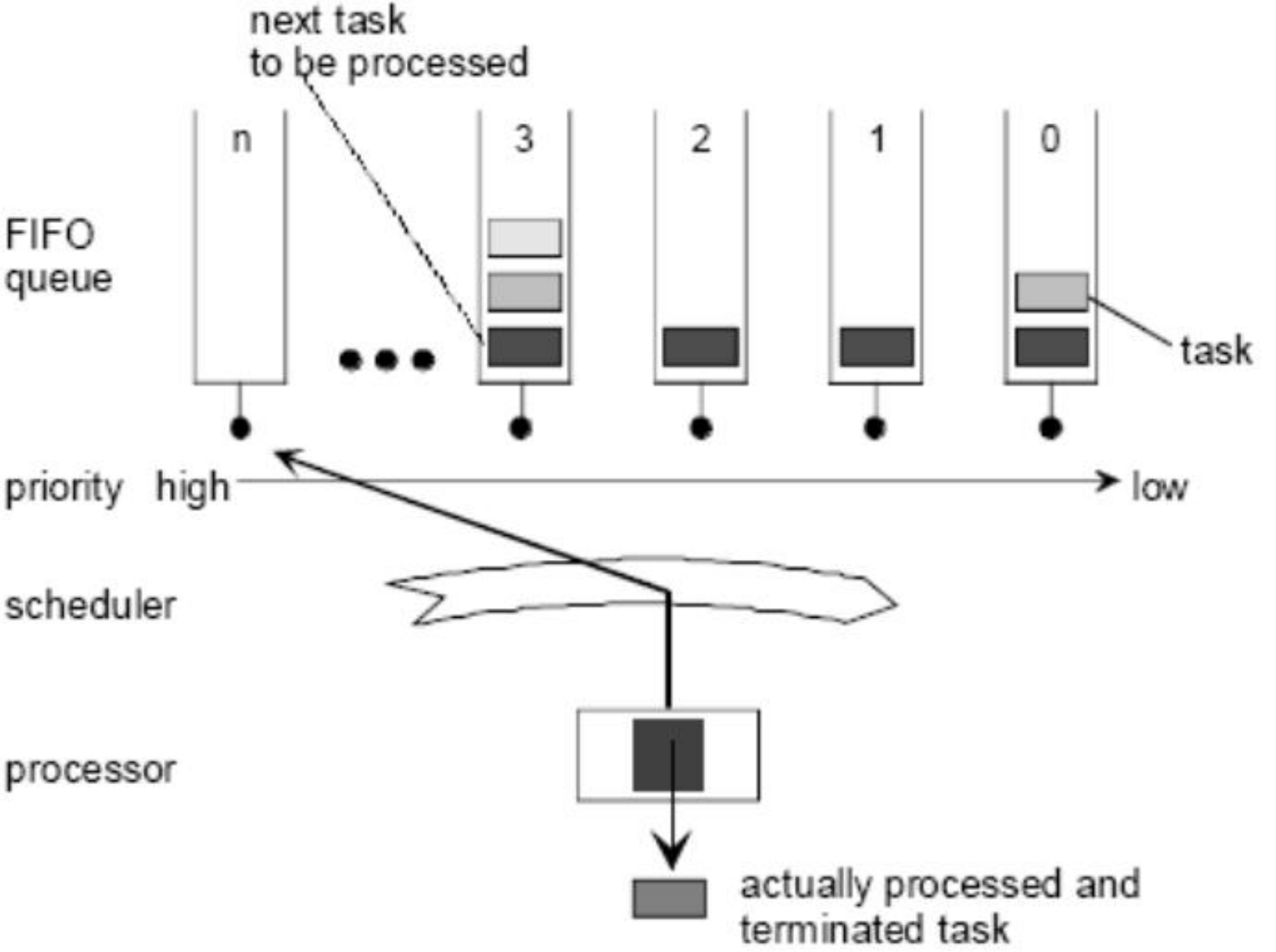
Transition	Former state	New state	Description
activate	*suspended*	*ready*[3]	A new task is set into the *ready* state by a system service. The OSEK operating system ensures that the execution of the task will start with the first instruction.
start	*ready*	*running*	A *ready* task selected by the scheduler is executed.
preempt	*running*	*ready*	The scheduler decides to start another task. The *running* task is put into the *ready* state.
terminate	*running*	*suspended*	The *running* task causes its transition into the *suspended* state by a system service.

两种任务类型的比较

- 基本任务没有等待状态，仅在任务开始和结束形成同步点(Synchronisation points)，如果应用程序需要内部同步点，可以用两个以上任务实现。
- 标准中有这样一句：An advantage of basic tasks is their moderate requirement regarding run time context (RAM).（？？）
- 扩展任务的有点是：可以由一个任务完成一个连贯的工作，即使有同步需求。当扩展任务缺少继续往下运行需要的信息时，便进入等待状态。当具有需要的信息（事件被设置或者数据被更新）时，脱离等待状态。

任务的优先级

- OS标准中的任务具有静态定义的优先级，它不能被应用程序修改。
- 有一种特殊情况，就是优先级天花板协议有效时，操作系统能改变一个任务的优先级。
- 标准中定义0是最低的优先级，没有定义最高优先级，定义太多的优先级将会影响应用程序的可移植性
- 如果允许多个任务具有相同的优先级，则需要多级任务队列。同优先级按照FIFO进行调度。
- 见下图：


next task
to be processed
n
3
2
1
0
FIFO
queue
task
priority
high
low
scheduler
processor
actually processed and
terminated task

任务的激活

- 激活将会使任务从挂起状态到就绪状态。
- 基本任务有一个独特的特性：多重激活，应用程序可以对基本任务提出多重激活请求(“Multiple requesting of task activation”)。意思是操作系统接收并记录已激活基本任务的并发激活请求。操作系统生成阶段会有一个激活次数的最大值。
- 基本任务状态转换图中的特殊情况：当一个基本任务处于非挂起状态时，激活并立即不进入就绪状态。
- 多重激活允许一个任务终止后然后立即在执行。
- 缺点：需要(??)一个包含所有优先级的多任务队列。

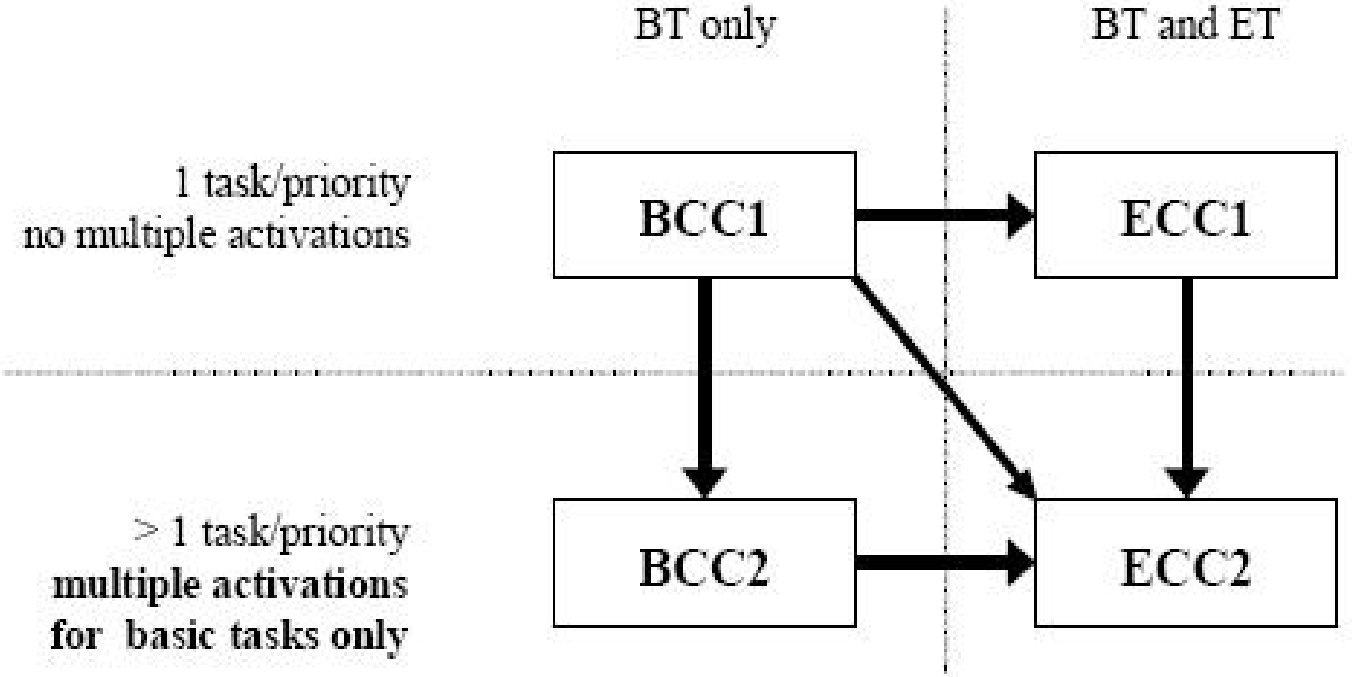
符合类

- 一个符合类被定义为操作系统要求的一个具体实现这样要求包括一个由应用指定的属性集。

属性	BCC1	BCC2	ECC1	ECC2
基本任务激活数	1	≥1	1	≥1
每个优先级任务数	1	≥1	1	≥1
基本任务	Yes	Yes	Yes	Yes
扩展任务	No	No	Yes	Yes

符合类

- 任务是向上兼容的：任何为BCCx符合类开发的任务可在可在一个ECCx符合类中使用，任何为xCC1符合类编写的任务可在xCC2符合类中使用。



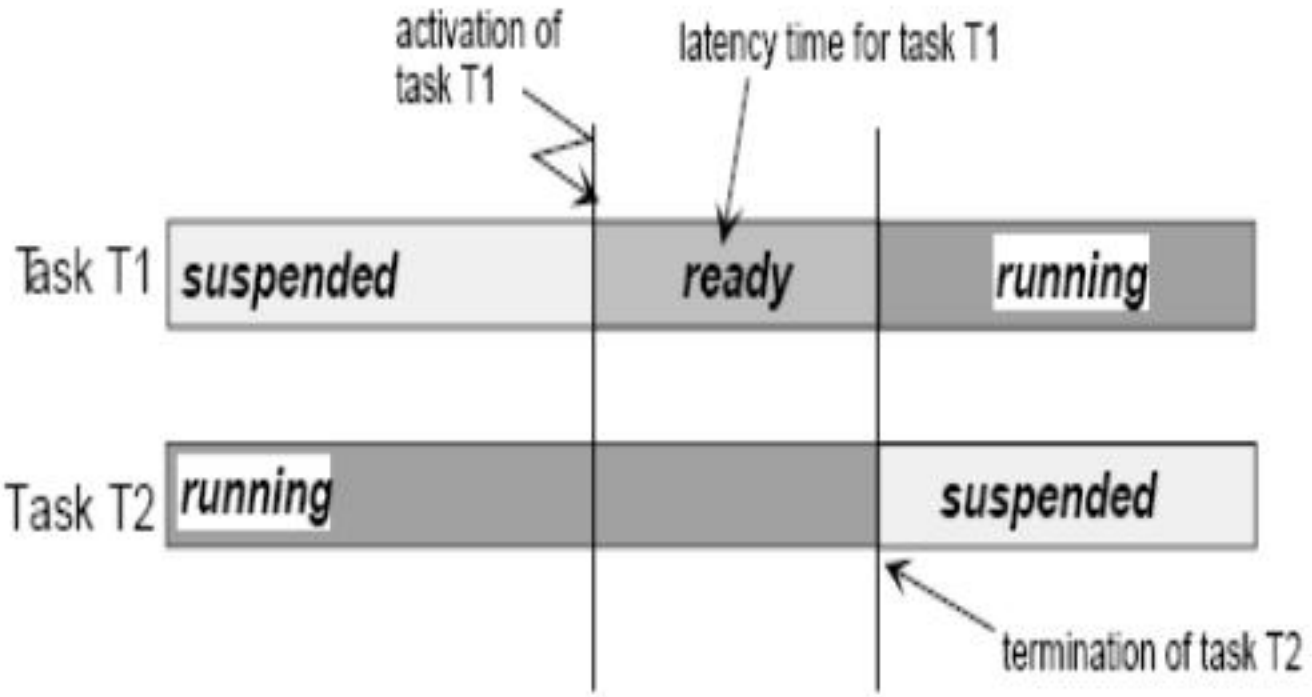
	BCC1	BCC2	ECC1	ECC2
Multiple requesting of task activation	no	yes	BT[2]: no ET: no	BT: yes ET: no
Number of tasks which are not in the *suspended* state	8		16 (any combination of BT/ET)	
More than one task per priority	no	yes	no (both BT/ET)	yes (both BT/ET)
Number of events per task	—		8	
Number of task priorities	8			
Resources	RES_SCHEDULER	8 (including RES_SCHEDULER)		
Alarm	1			
Application Mode	1			

- 具体开发OSEKOS，支持一个符合类即可。标准中为每个符合类的属性定义了一组最小的需求数值，超过这组数字将会影响可移植性。

调度策略

- 调度策略分三种：非抢占、全抢占、混合抢占
- 软件开发人员或者系统集成者通过给每个任务分配优先级并且把是否可抢占作为一个任务的属性来决定任务的执行顺序。
- 调度函数Schedule()检查就绪的最高优先级的任务，把处理器交给它。
- 如果OS API(或者称系统服务)正在运行，调度可能要推迟到这个API完成之后。

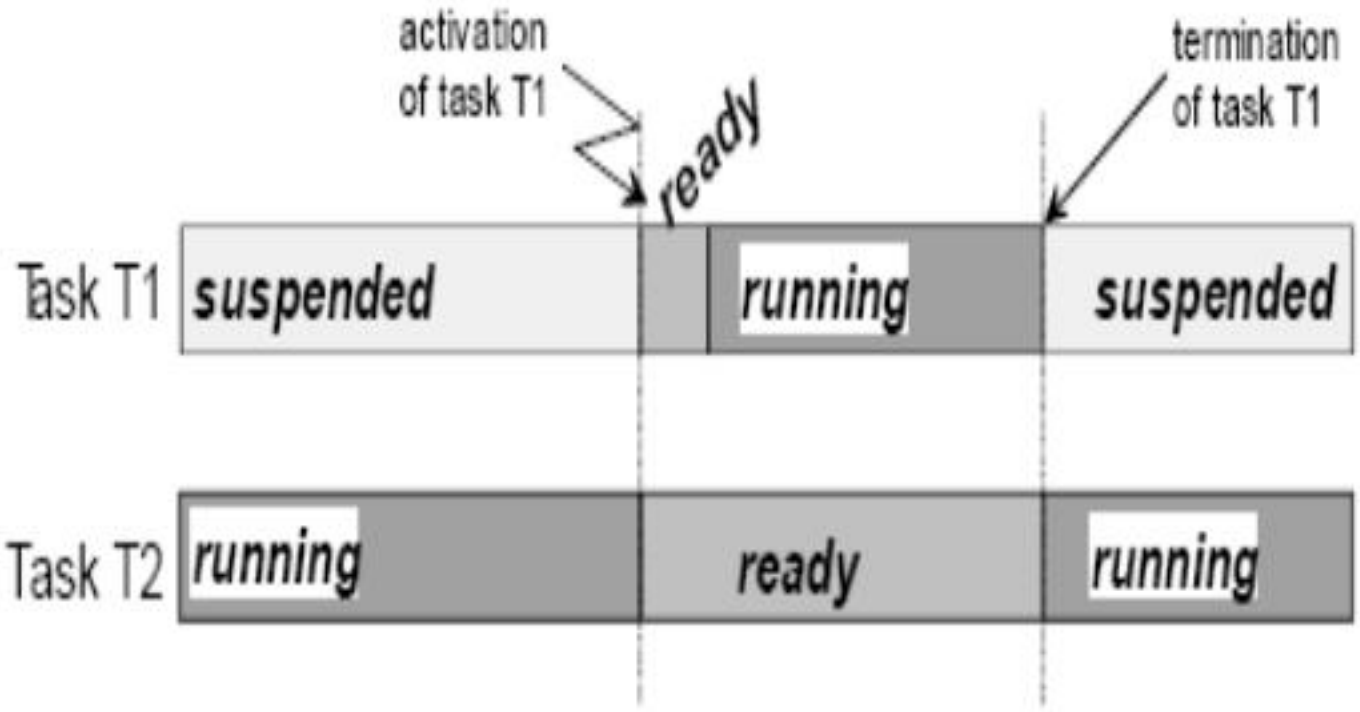
非抢占调度


activation of task T1
latency time for task T1
Task T1
suspended
ready
running
Task T2
running
suspended
termination of task T2

非抢占调度的调度点

- 非抢占调度的调度点：
  - 一个任务的成功结束（调用TerminateTask）
  - 一个任务成功结束并显示激活一个任务（当前任务调用ChainTask()）
  - 显示调用调度函数Schedule()
  - 一个任务调用WaitEvent()，并转入等待状态。

全抢占调度策略


activation
of task T1
ready
termination
of task T1
Task T1
suspended
running
suspended
Task T2
running
ready
running

全抢占调度的调度点

- 调度点：
  - 一个任务的成功结束（当前任务调用TerminateTask（）一个任务要结束必须调用它）
  - 一个任务成功结束并显示激活一个任务（当前任务调用ChainTask()）
  - 在任务层激活一个任务
  - 显示的调用WaitEvent()，并转入等待转态
  - 在任务层设置了某个任务正在等待的事件
  - 任务层资源的释放
  - 从中断层转到任务层运行

混合抢占任务调度

- 如果应用程序中的一些任务被指定为抢占任务，而另一些被指定为非抢占任务，则该程序采用混合抢占任务调度策略。
- 在这种策略下，操作系统根据当前运行任务允许的抢占类型决定是否启动调度程序。
- 非抢占任务在混合调度中的意义：
  - 如果任务的执行时间和上下文切换时间差不多
  - 如果为保存任务的上下文需耗费大量内存
  - 应用程序中某个任务被强制指定为非抢占

任务的终止

- 在OSEK操作系统中，一个任务只能被它自己终止。
- OSEK标准定义了一个API ChainTask()；允许在当前任务终止后立即再激活参数中指定的任务，当然可以指定任务自己。
- 每个任务在它的代码最后必须结束它自己。或者调用TermiateTask()；或者调用ChainTask(Task)；
  Ending the task without a call to *Terminate-Task* or *ChainTask* is strictly forbidden!

应用程序模式(APPMODE)

- 操作系统启动时需提供要运行的APPMODE
- 一种OSEK标准的实现至少要支持一种APPMODE
- 操作系统启动后不允许改变APPMODE
- APPMODE被应用程序用于定义当前的操作环境，The application mode is a means to structure the software running in the ECU according to those different conditions。用户可以按某种APPMODE来决定是否启动某个任务或ISR。就是说，某些任务和ISR只有在特定的APPMODE下才启动
- OSEK OS标准并未规定各个API和APPMODE的关系，纯粹是为应用程序提供另一种层次的信息和控制。

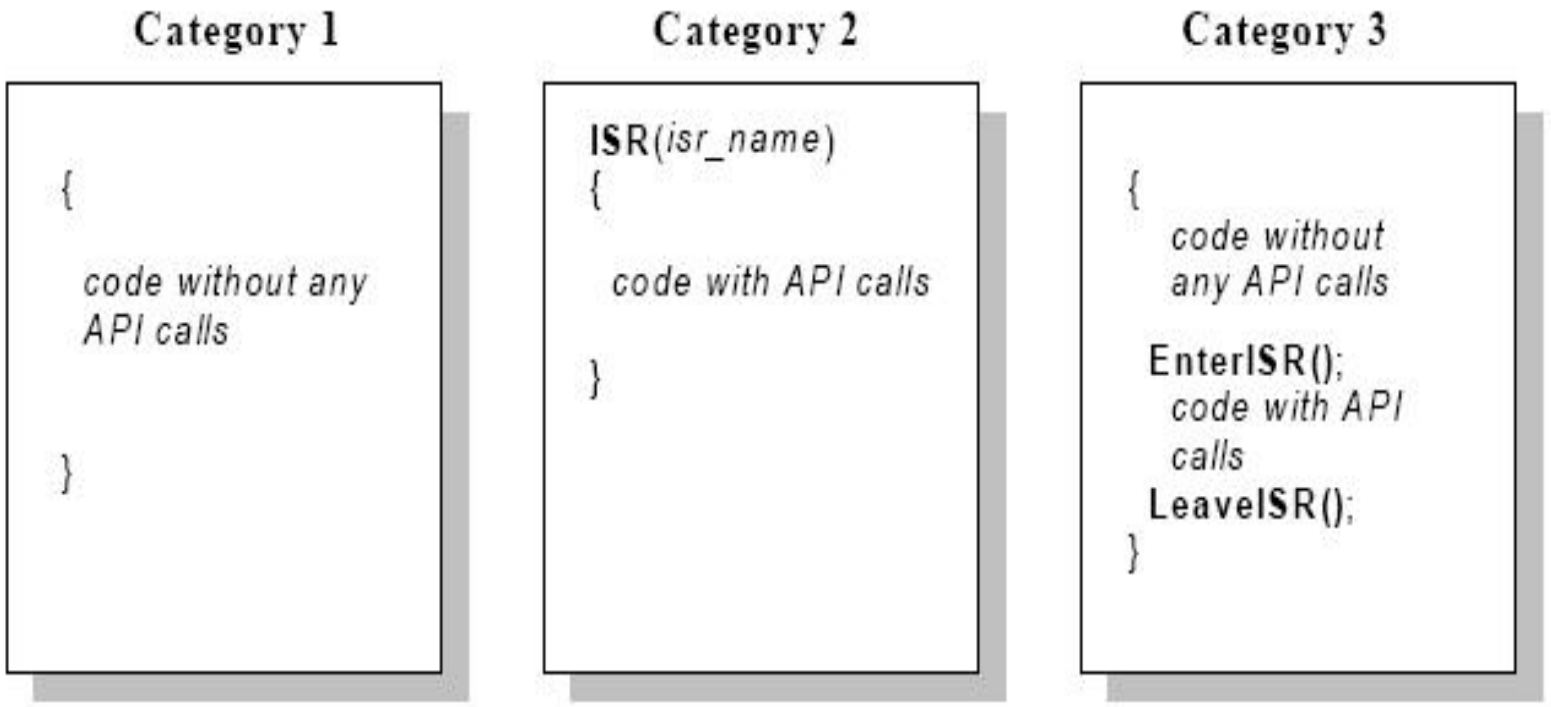
APPMODE的特性(characteristics)

- 标准中提到三个特性：Start up performance，Support of exclusive applications，Supported by all conformance classes。
- 因为检查当前APPMODE的开销很小，所有的符合类均支持APPMODE。
- 在启动OS之前，用户需决定要启动的APPMODE，然后将其作为一个参数传递给操作系统。（在StartOS()作为参数传递）。

中断分类

- 中断服务程序(ISR)分为三类：1类ISR，2类ISR和3类ISR。
- 1类ISR：此类中断不需要调用API。对任务管理没有影响。
- 2类ISR：需要调用API。典型地，这类中断需要增加计数器的值、激活任务，设置时间和发送消息。操作系统为这类ISR提供了准备其运行环境的框架，在框架内执行用户编写的中断处理程序。
- 3类ISR：这类中断在满足一定条件时会调用API，也有可能不调用API。但调用时必须调用EnterISR()和LeaveISR()。图示如下：

中断分类


Category 1
{
code without any API calls
}
Category 2
ISR(isr_name)
{
code with API calls
}
Category 3
{
code without any API calls
EnterISR();
code with API calls
LeaveISR();
}

中断

- 在ISR内部不会调度。
- 调度发生在2类ISR和3类ISR返回时，没有其它要处理的中断且该中断是在一个可抢占任务运行时发生的。
- 此时的调度应根据全抢占调度策略的调度点去调度任务。
- 中断的调度顺序是由具体硬件来决定的。
- 可被任务和ISR使用的中断服务如下图：

Service	called by Task	called by ISR category 2 and 3
ActivateTask	allowed	allowed
TerminateTask	allowed	--
ChainTask	allowed	--
Schedule	allowed	--
GetTaskID	allowed	allowed
GetTaskState	allowed	allowed
EnterISR	--	allowed[6]
LeaveISR	--	allowed[6]
EnableInterrupt	allowed	allowed
DisableInterrupt	allowed	allowed
GetInterruptDescriptor	allowed	allowed
DisableAllInterrupts	allowed	allowed
EnableAllInterrupts	allowed	allowed
SuspendOSInterrupts	allowed	allowed
ResumeOSInterrupts	allowed	allowed

Service	called by Task	called by ISR category 2 and 3
ResumeOSInterrupts	allowed	allowed
GetResource	allowed	allowed
ReleaseResource	allowed	allowed
SetEvent	allowed	allowed
ClearEvent	allowed	--
GetEvent	allowed	allowed
WaitEvent	allowed	--
GetAlarmBase	allowed	allowed
GetAlarm	allowed	allowed
SetRelAlarm	allowed	allowed
SetAbsAlarm	allowed	allowed
CancelAlarm	allowed	allowed
GetActiveApplicationMode	allowed	allowed
StartOS	--	--
ShutdownOS	allowed	allowed

事件机制

- 事件用于任务同步
- 操作系统只为扩展任务提供事件机制，每个事件均被分配给某个任务，每个任务可以拥有有限个事件。
- 事件机制激发任务进入或者脱离等待状态。
- 以下两图为全抢占和非抢占情况下设置事件对任务调度的影响：

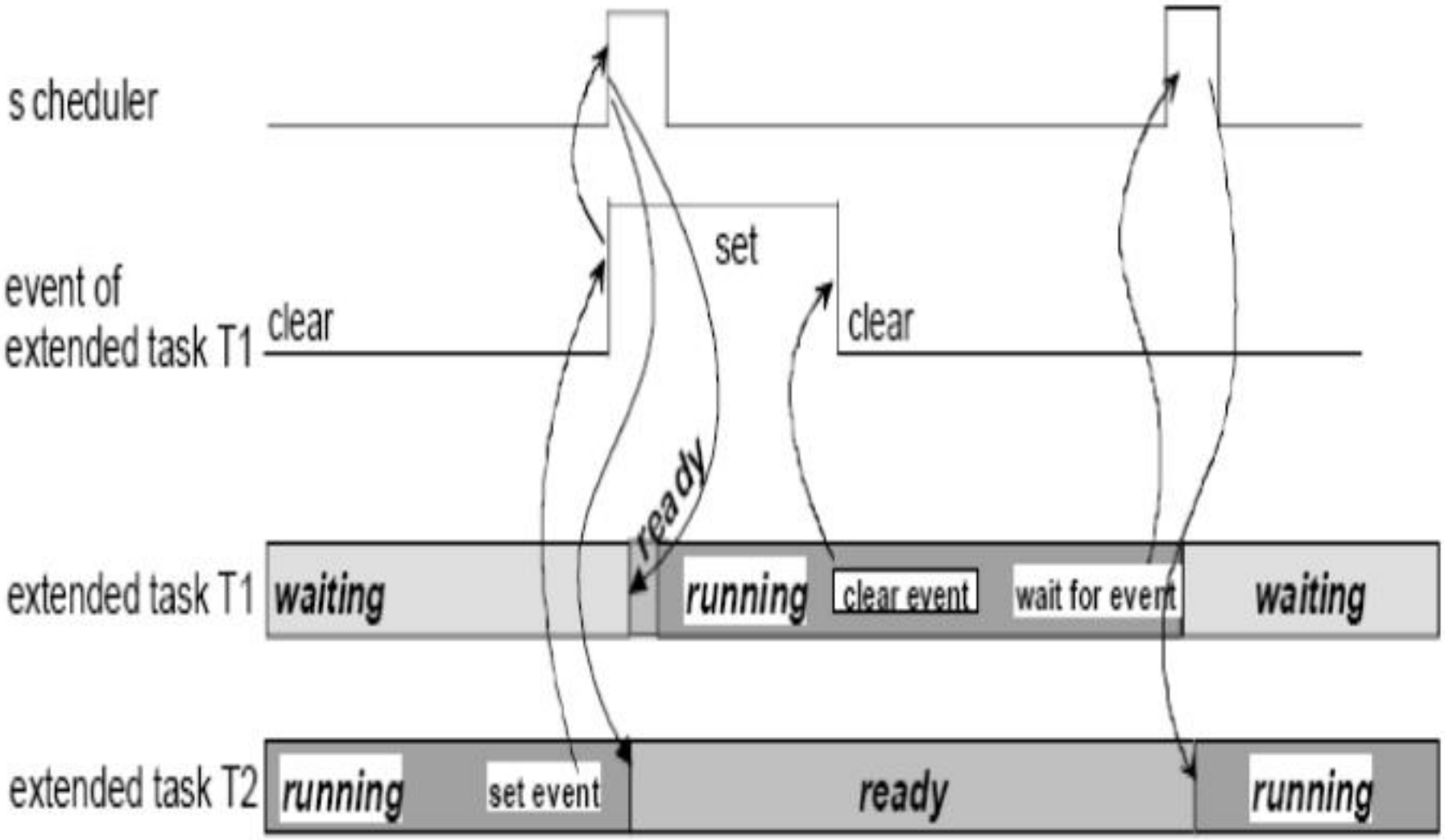


Figure 6–1 Full pre-emptive synchronisation of extended tasks

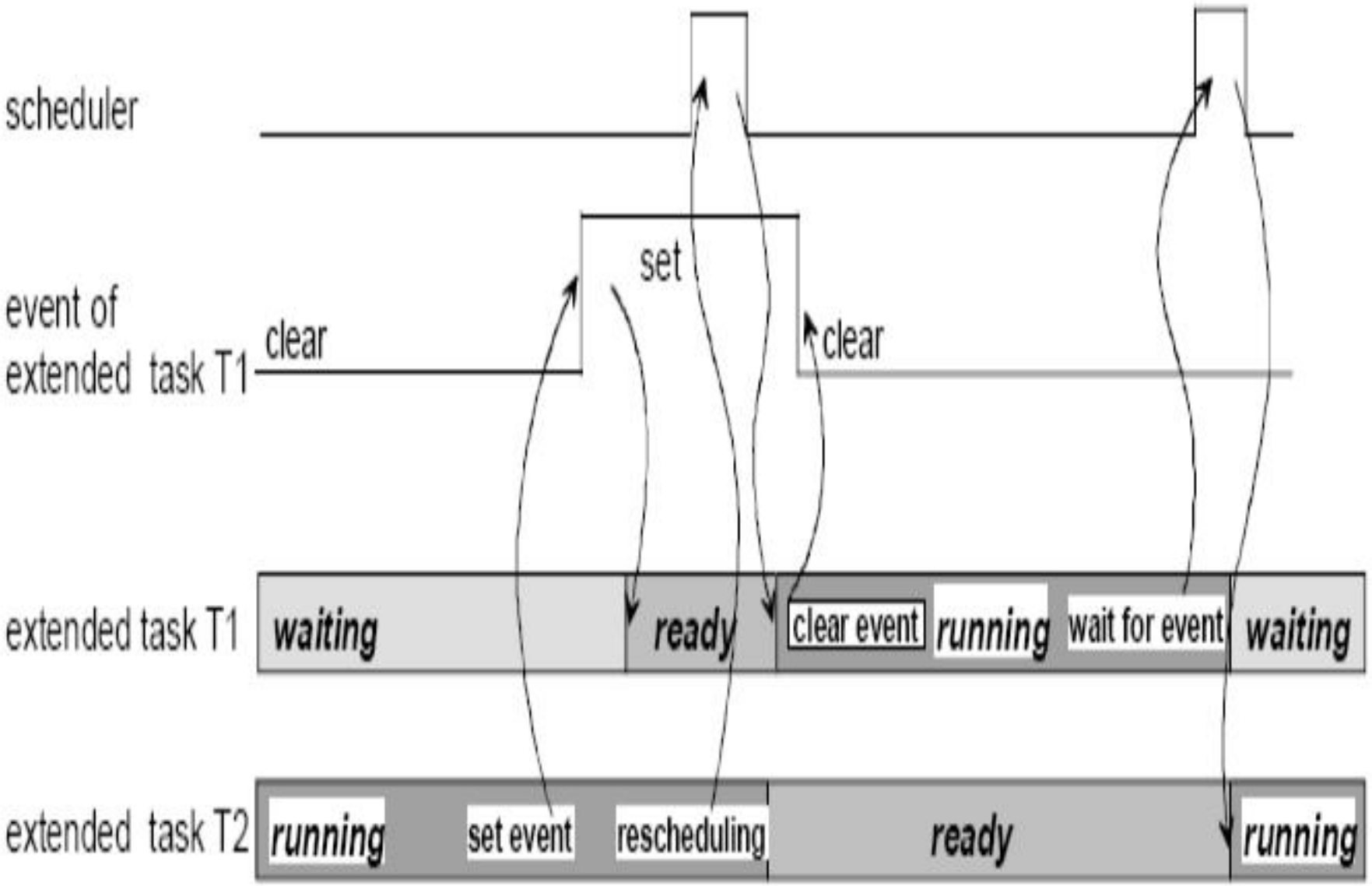


Figure 6–2 Non pre-emptive synchronisation of extended tasks

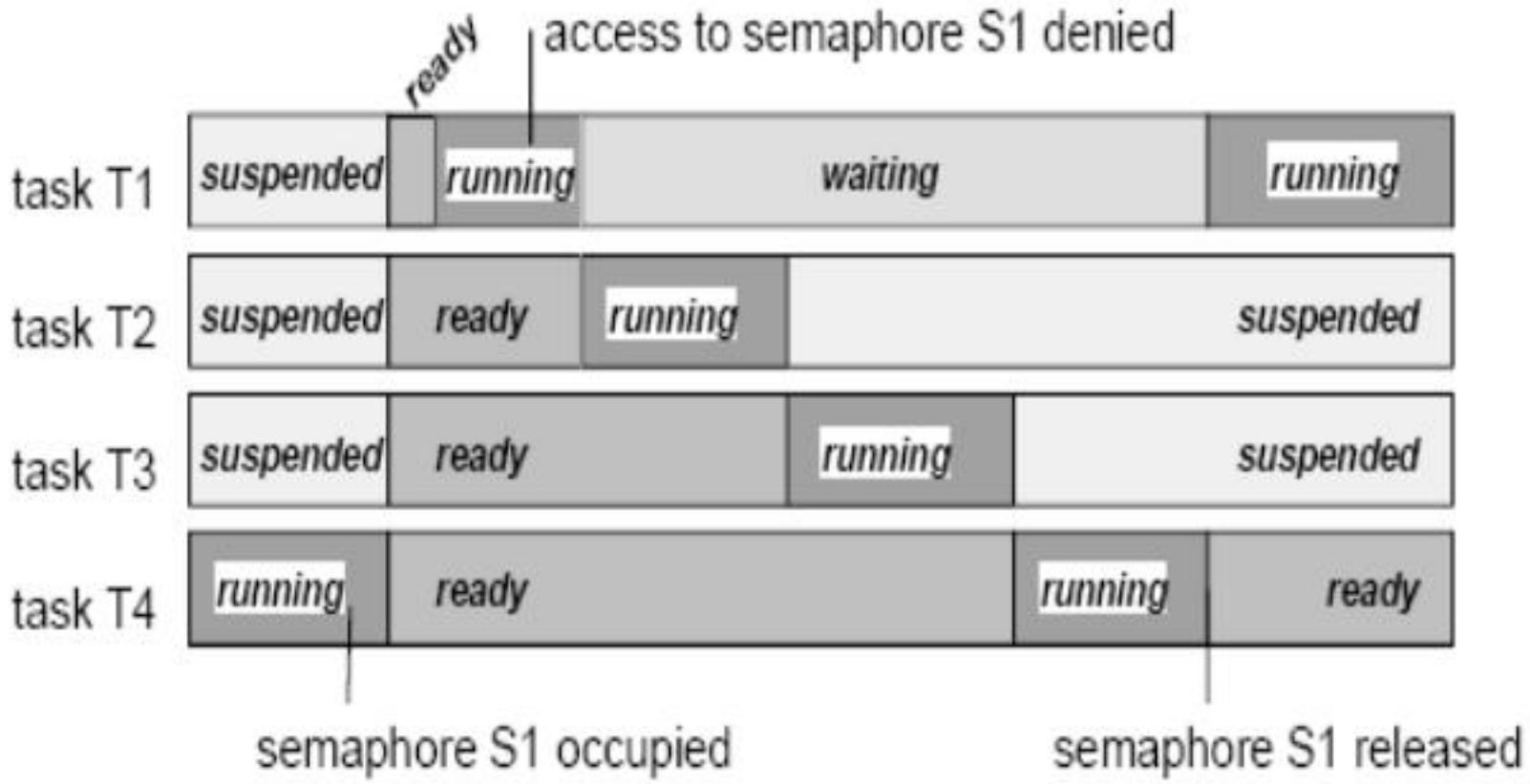
资源管理

- 资源管理对四个符合类都是必须的。
- 资源管理的目标：
  - 两个任务不能同时占有同一资源（互斥）。
  - 不出现优先级反转
  - 不能出现死锁
  - 访问资源的任务不能进入等待状态。
- ISR只有在其所需要的资源全部可用的时候才执行。The OSEK operating system ensures also that an interrupt service routine is only processed if all resources which might be occupied by that interrupt service routine during its execution have been released。

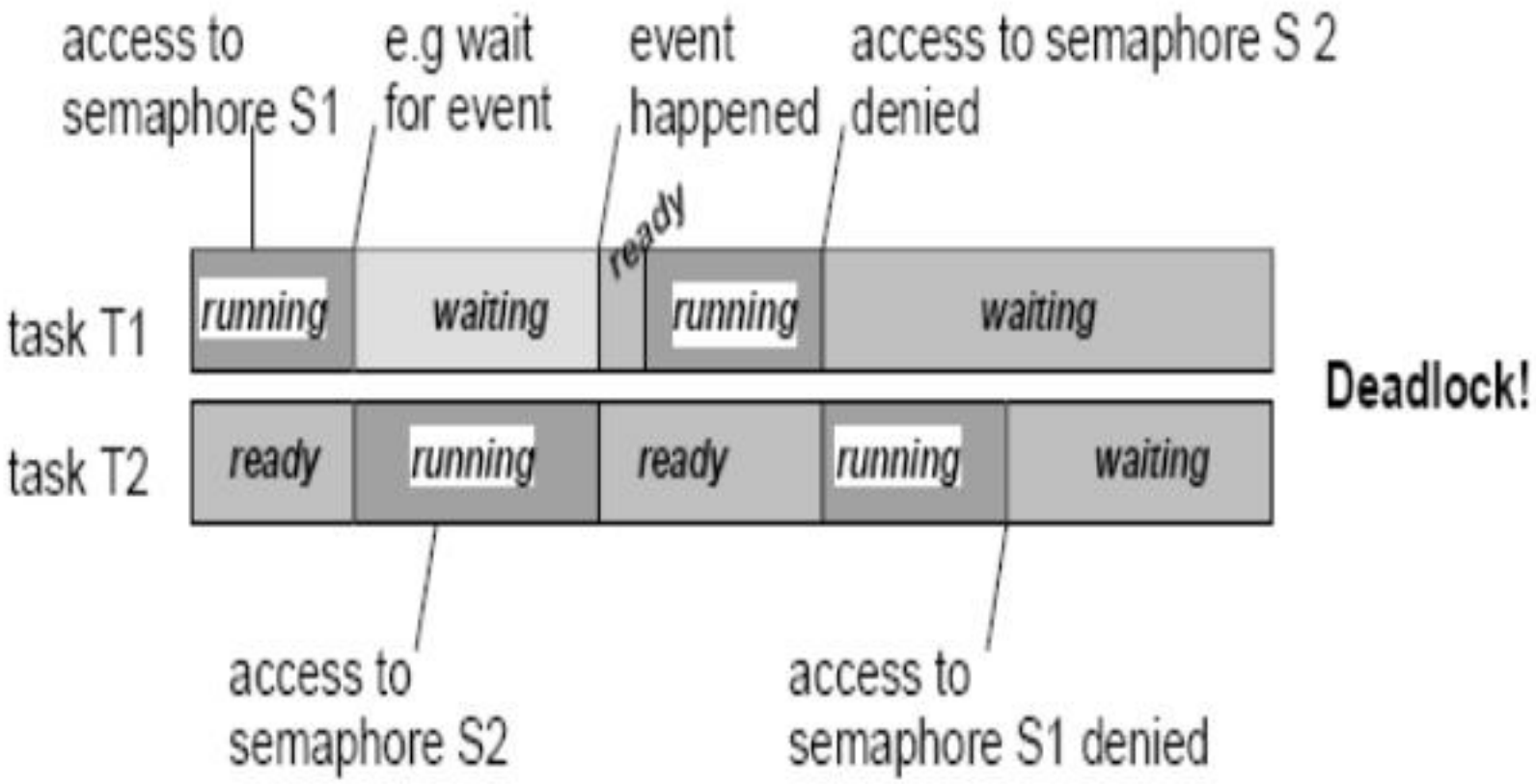
调度程序作为资源

- 如果一个任务在执行期间不想被打断，可以锁定调度程序。
- 在系统生成的时候，系统生成一个资源RES_SHEDULE。
- 由于中断层的优先级高于调度者，所以中断不会受到调度资源被上锁的影响。

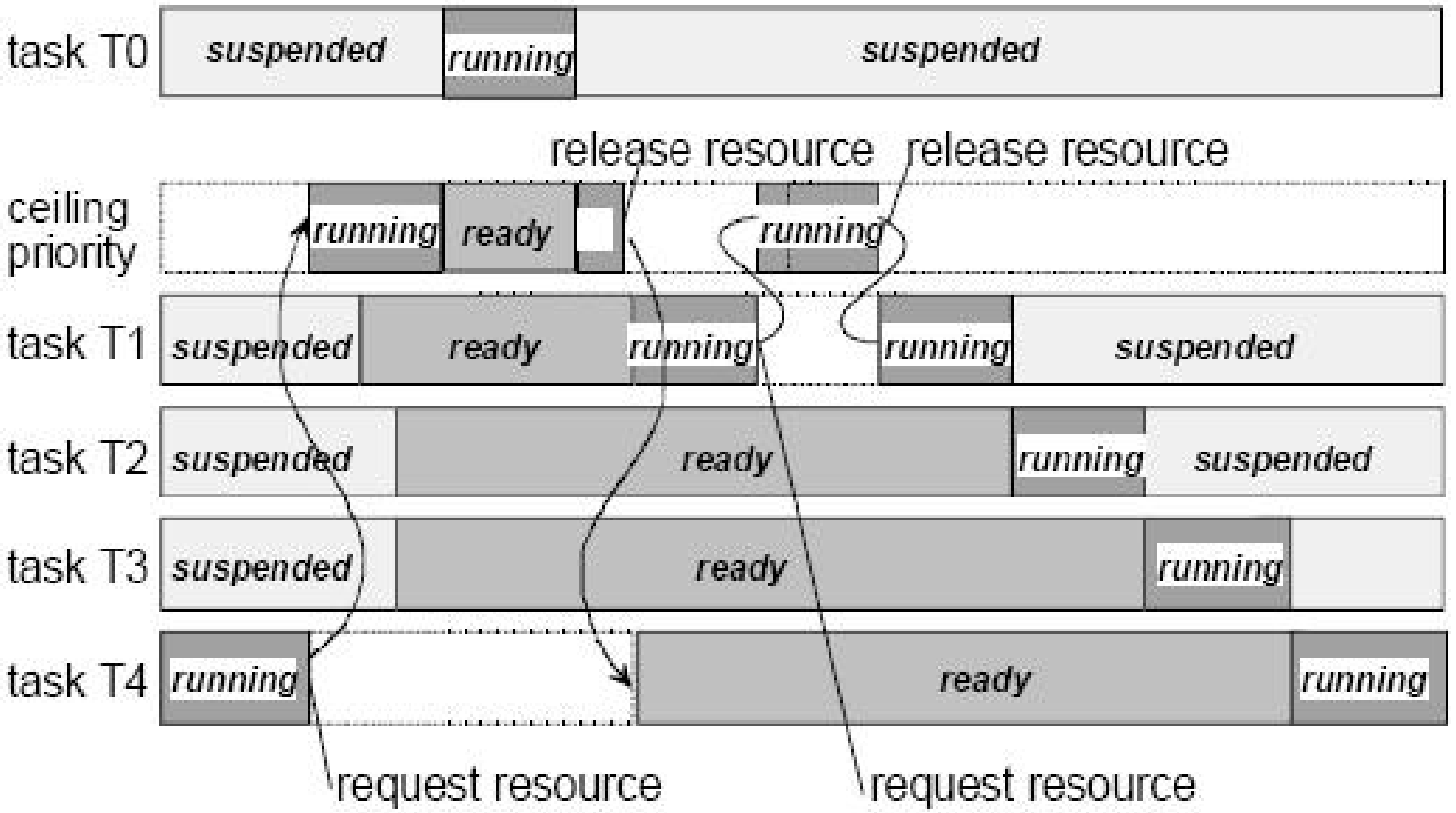
优先级反转


ready
access to semaphore S1 denied
task T1
suspended
running
waiting
running
task T2
suspended
ready
running
suspended
task T3
suspended
ready
running
suspended
task T4
running
ready
running
ready
semaphore S1 occupied
semaphore S1 released

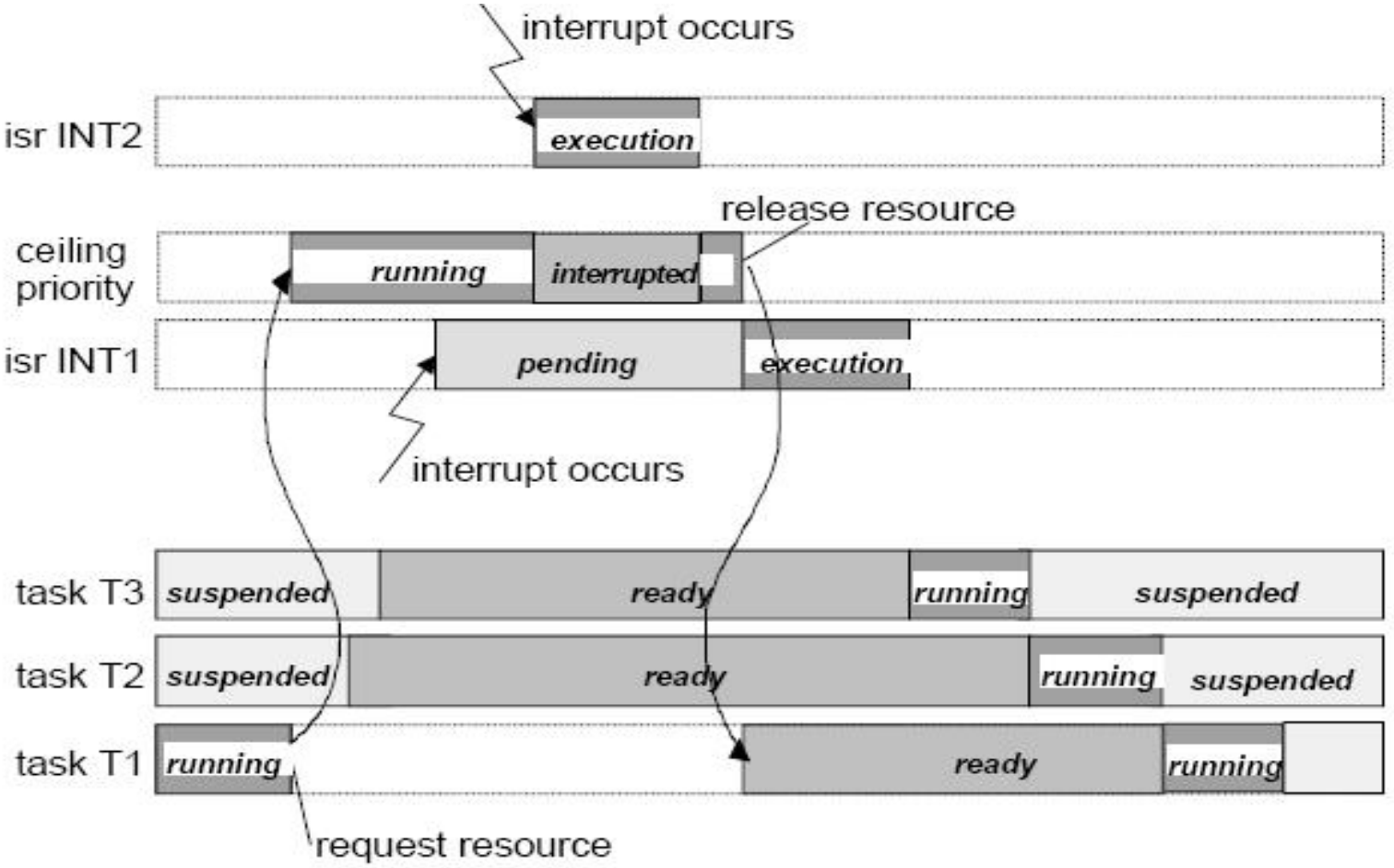
死锁


access to semaphore S1
e.g wait for event
event happened
access to semaphore S 2 denied
ready
task T1
running
waiting
running
waiting
Deadlock!
task T2
ready
running
ready
running
waiting
access to semaphore S2
access to semaphore S1 denied

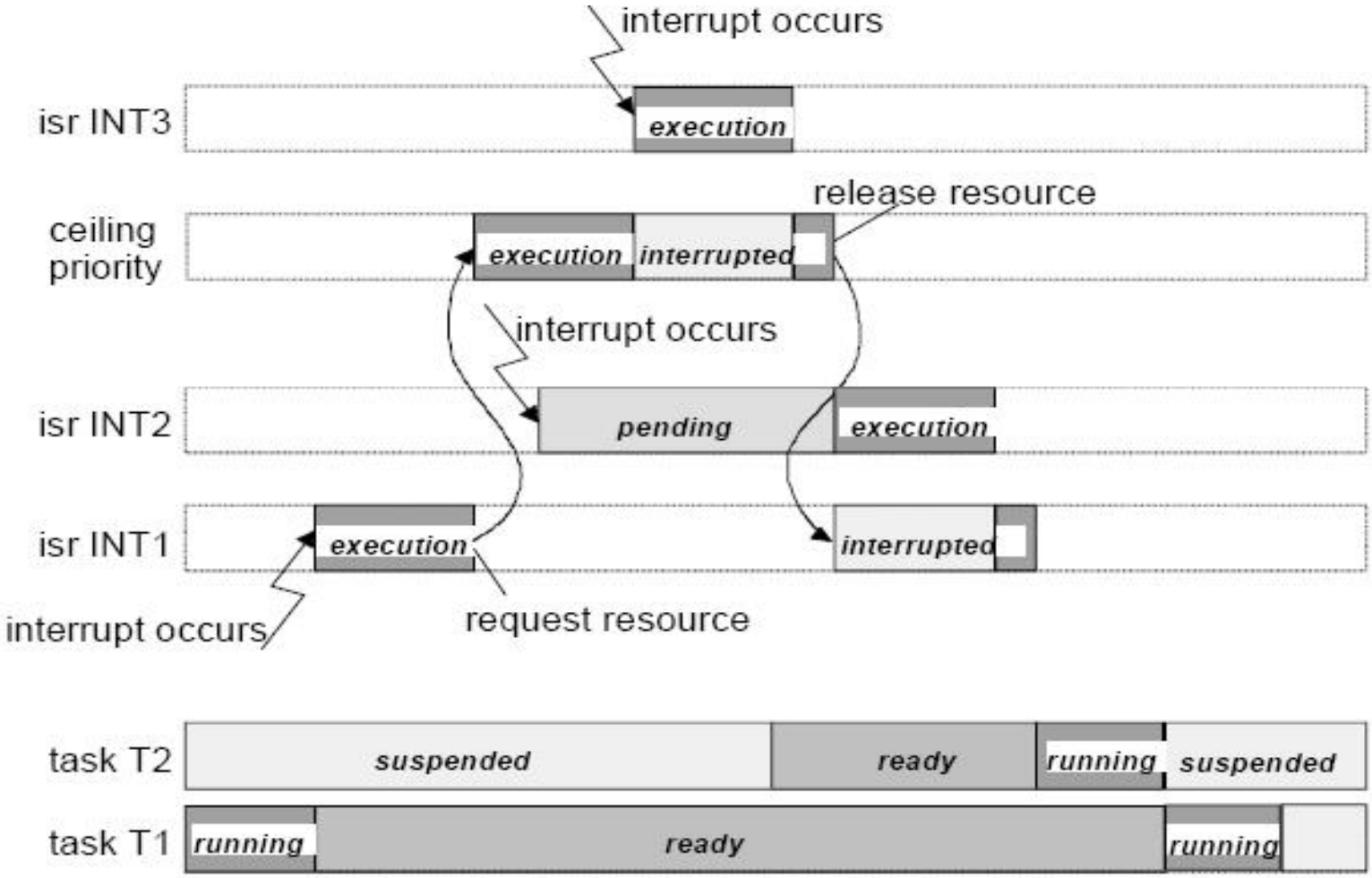
资源的优先级天花板协议



资源的优先级天花板协议



资源的优先级天花板协议



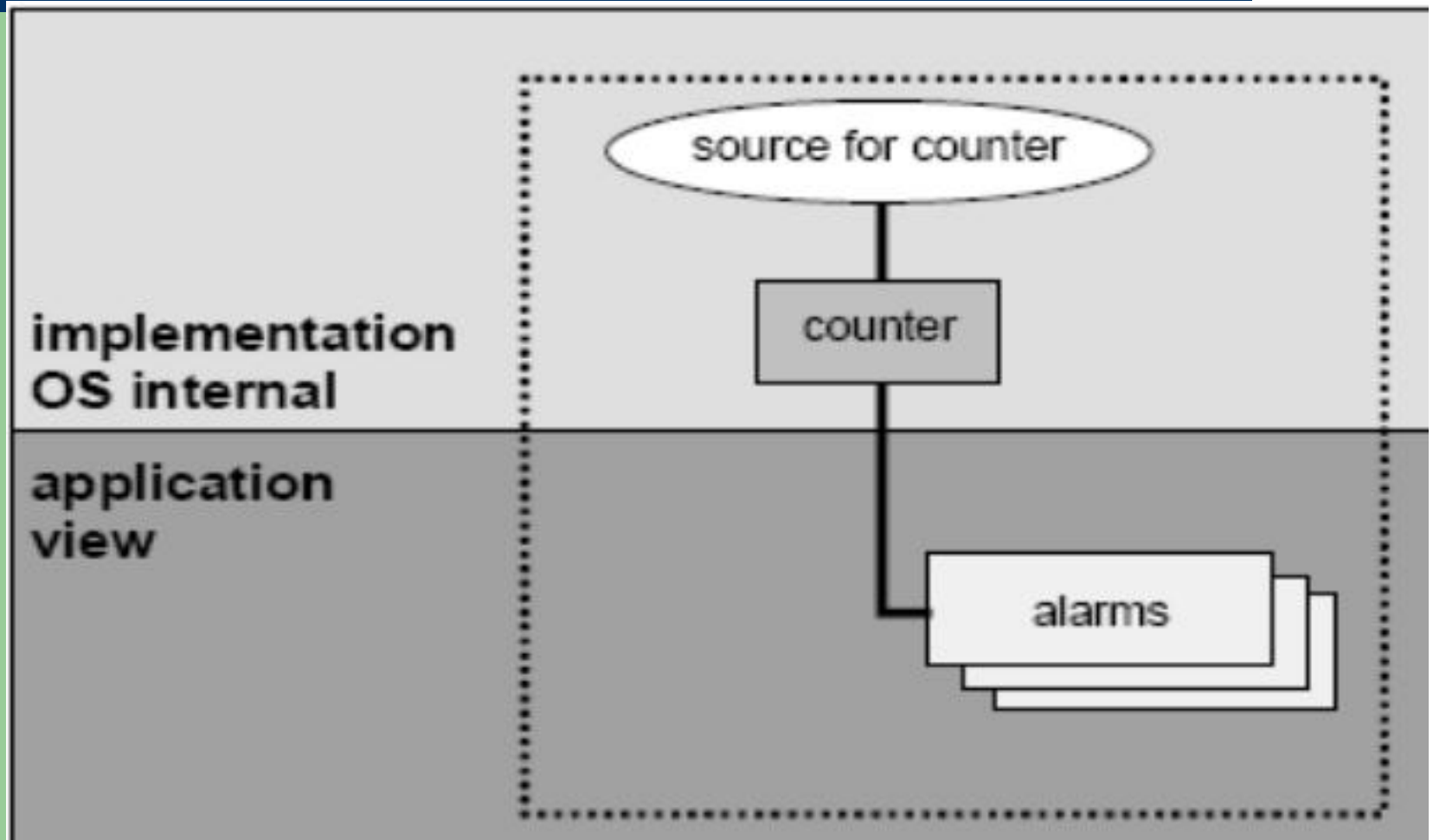
报警(ALARM)

- 计数器(Counter)：计数器由计数器的值（滴答数）来衡量，OSEK OS必须提供一个计数器（硬件或者软件定时器）。
- OSEK标准没有提供标准对计数器操作的API
- 报警会与一个计数器和一个任务/事件相关联，当计数器值达到预定义的数值时，就会激活相关联的任务或设置相应的事件。
- OSEK标准提供了设置报警何时触发的相关API
- 报警例子如下：

报警

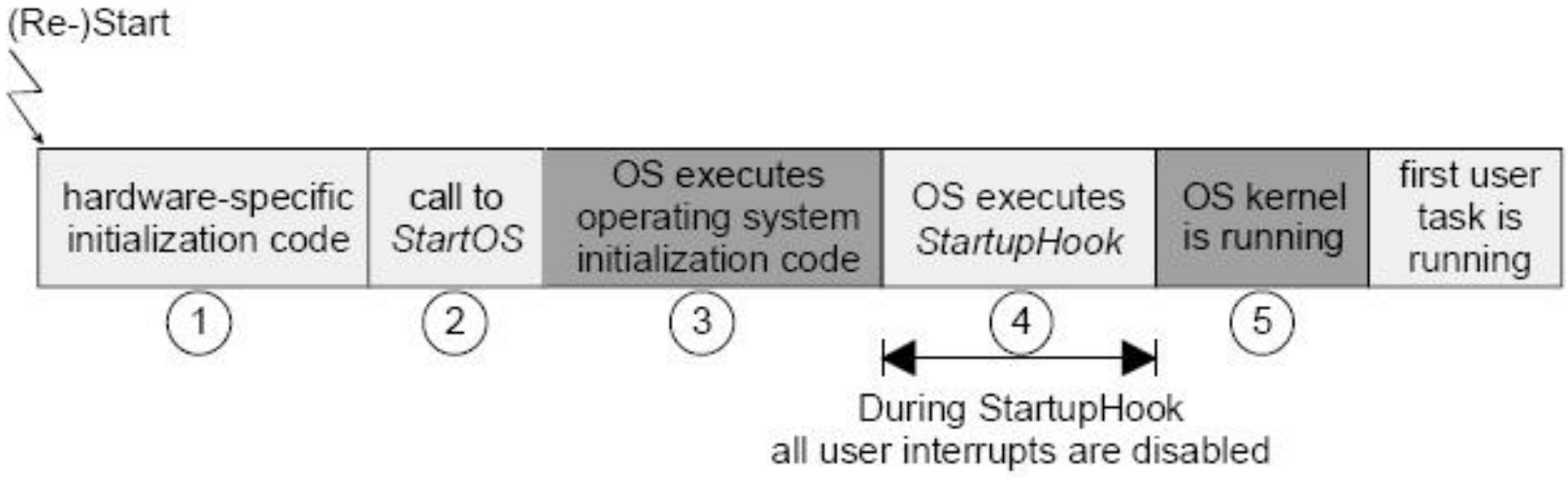
- 用户定义计数器和报警
  - Counter1 { MaxValue=500;MinCycle=10;}
  - Alarm1 { Counter＝Counter1; Task=Task1;}
  - SetAbsAlarm(alarm1,100,50);
  - SetRelAlarm(alarm1,80,50);
- Abs-绝对计数器值 Rel-相对计数器值，如果最后一个参数被设置为0，不是周期报警。

报警与OS


source for counter
counter
implementation
OS internal
application
view
alarms

系统的启动

- ①执行硬件相关代码，比如关掉2类和3类中断，到阶段⑤再开中断
- ②调用StratOS，APPMODE作为参数



- ③操作系统调用内部启动代码，比如找到优先级最高的任务置为就绪，但并不调度。
- ④调用StartupHook()

回调函数和错误处理

- 操作系统在合适的地方调用如下回调函数：
  - StartupHook()
  - ShutDownHook()
  - ErrorHook()
  - PreTaskHook()
  - PostTaskHook()
- 错误处理：应用错误、致命错误
  - 如果操作系统提供的服务结果不正确，但是内部数据没有破坏，则认为是应用错误。比如一个基本任务等待事件。
  - 认为内部数据不正确则认为是致命错误，此时要关闭OS。比如在调度新任务执行时，所有任务处于非就绪状态。

OS的关闭和调试

- OSEK标准中定义了一个关闭操作系统的系统调用ShutdownOS()，在其中调用ShutdownHook()。