第2章 实时系统概念

实时系统的特点是，如果逻辑和时序出现偏差将会引起严重后果的系统。有两种类型的实时系统：软实时系统和硬实时系统。在软实时系统中系统的宗旨是使各个任务运行得越快越好，并不要求限定某一任务必须在多长时间内完成。

在硬实时系统中，各任务不仅要执行无误而且要做到准时。大多数实时系统是二者的结合。实时系统的应用涵盖广泛的领域，而多数实时系统又是嵌入式的。这意味着计算机建在系统内部，用户看不到有个计算机在系统里面。以下是一些嵌入式系统的例子：

过程控制
- 食品加工
- 化工厂

汽车业
- 发动机控制
- 防抱死系统(ABS)

办公自动化
- 传真机
- 复印机

计算机外设
- 打印机
- 计算机终端
- 扫描仪
- 调制解调器

通讯类
- Switch Hurb
- 路由器

机器人

航空航天
- 飞机管理系统
- 武器系统
- 喷气发动机控制

民用消费品
- 微波炉
- 洗碗机
- 洗依机
- 稳温调节器

实时应用软件的设计一般比非实时应用软件设计难一些。本章讲述实时系统概念。

2.0 前后台系统（Foreground/Background System）

不复杂的小系统一般设计成如图 2.1 所示的样子。这种系统可称为前后台系统或超循环系统(Super-Loops)。应用程序是一个无限的循环，循环中调用相应的函数完成相应的操作，这部分可以看成后台行为(background)。中断服务程序处理异步事件，这部分可以看成前台行为（foreground）。后台也可以叫做任务级。前台也叫中断级。时间相关性很强的关键操作(Critical operation)一定是靠中断服务来保证的。因为中断服务提供的信息一直要等到后台程序走到该处理这个信息这一步时才能得到处理，这种系统在处理信息的及时性上，比实际可以做到的要差。这个指标称作任务级响应时间。最坏情况下的任务级响应时间取决于整个循环的执行时间。因为循环的执行时间不是常数，程序经过某一特定部分的准确时间也是不能确定的。进而，如果程序修改了，循环的时序也会受到影响。

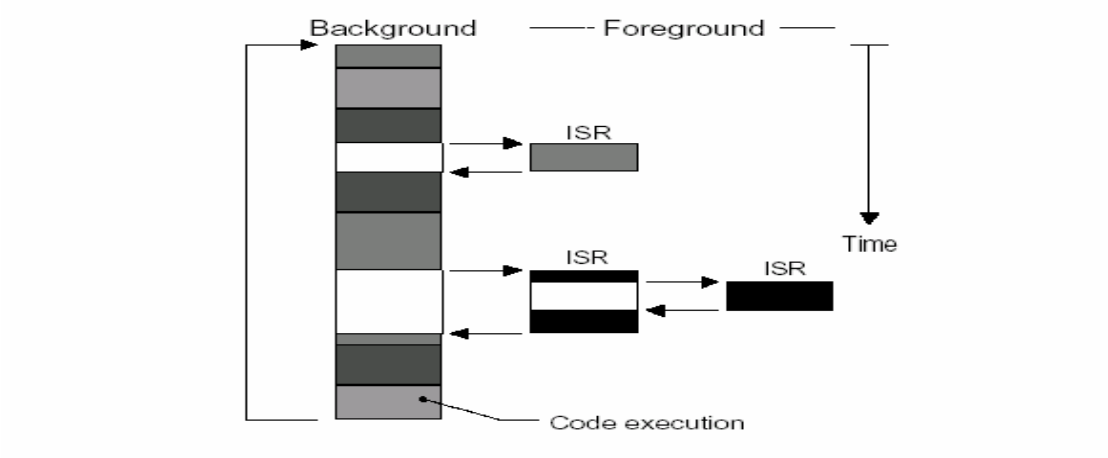

图 2-1 前后台系统

很多基于微处理器的产品采用前后台系统设计，例如微波炉、电话机、玩具等。在另外一些基于微处理器的应用中，从省电的角度出发，平时微处理器处在停机状态(halt)，所有的事都靠中断服务来完成。

2.1 代码的临界段

代码的临界段也称为临界区，指处理时不可分割的代码。一旦这部分代码开始执行，则不允许任何中断打入。为确保临界段代码的执行，在进入临界段之前要关中断，而临界段代码执行完以后要立即开中断。(参阅 2.03 共享资源)

2.2 资源

任何为任务所占用的实体都可称为资源。资源可以是输入输出设备，例如打印机、键盘、显示器，资源也可以是一个变量，一个结构或一个数组等。

2.3 共享资源

可以被一个以上任务使用的资源叫做共享资源。为了防止数据被破坏，每个任务在与共享资源打交道时，必须独占该资源。这叫做互斥（*mutual exclusion*）。在 2.18 节“互斥”中，将对技术上如何保证互斥条件做进一步讨论。

2.4 多任务

多任务运行的实现实际上是靠 CPU(中央处理单元)在许多任务之间转换、调度。CPU

只有一个，轮番服务于一系列任务中的某一个。多任务运行很像前后台系统，但后台任务有多个。多任务运行使 CPU 的利用率得到最大的发挥，并使应用程序模块化。在实时应用中，多任务化的最大特点是，开发人员可以将很复杂的应用程序层次化。使用多任务，应用程序将更容易设计与维护。

2.5 任务

一个任务，也称作一个线程，是一个简单的程序，该程序可以认为 CPU 完全只属该程序自己。实时应用程序的设计过程，包括如何把问题分割成多个任务，每个任务都是整个应用的某一部分，每个任务被赋予一定的优先级，有它自己的一套 CPU 寄存器和自己的栈空间(如图 2.2 所示)。

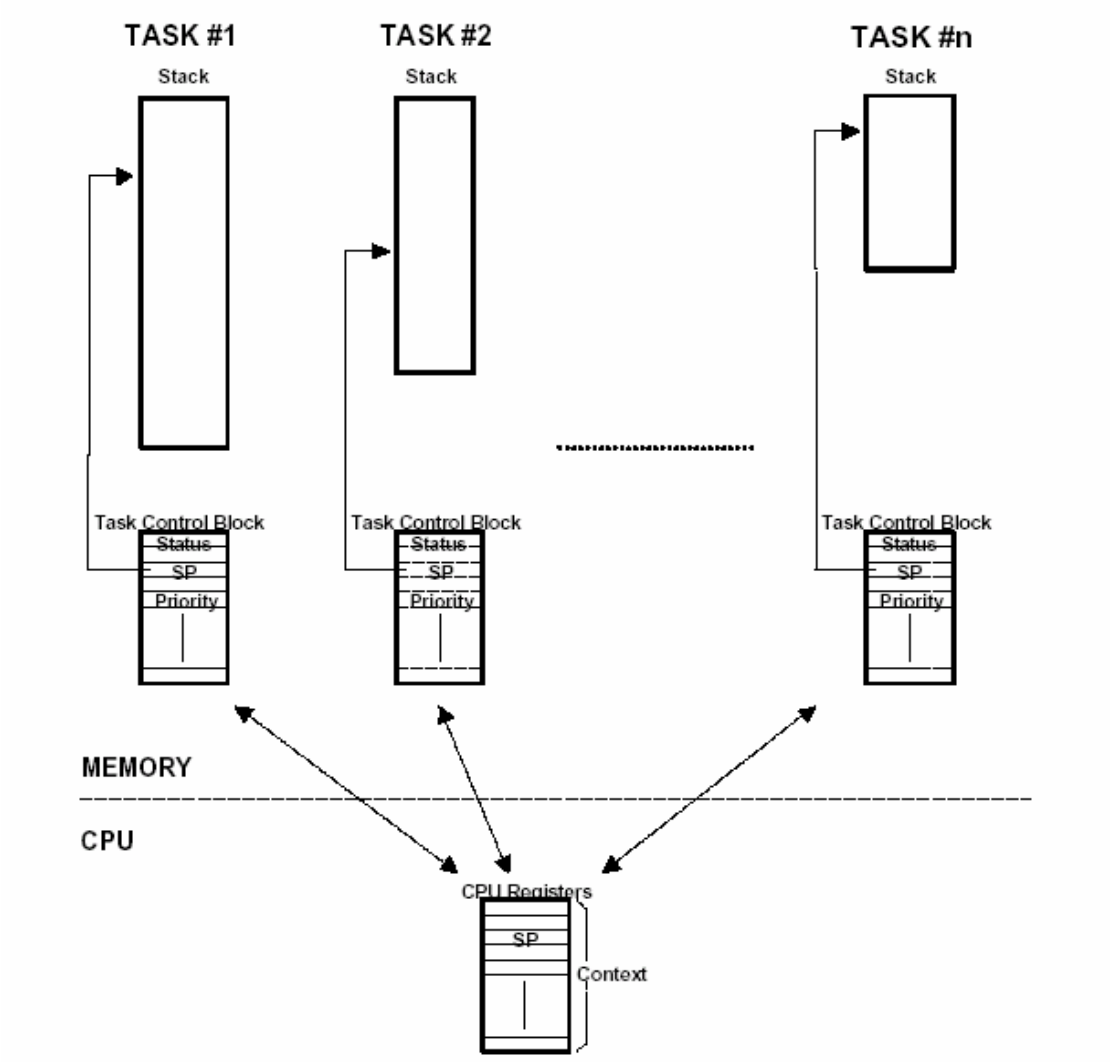

图 2.2 多任务。

典型地、每个任务都是一个无限的循环。每个任务都处在以下 5 种状态之一的状态下，这 5 种状态是休眠态，就绪态、运行态、挂起态(等待某一事件发生)和被中断态（参见图 2.3）休眠态相当于该任务驻留在内存中，但并不被多任务内核所调度。就绪意味着该任务已经准备好，可以运行了，但由于该任务的优先级比正在运行的任务的优先级低，还暂时不能运行。运行态的任务是指该任务掌握了 CPU 的控制权，正在运行中。挂起状态也可以叫做等待事件态 WAITING，指该任务在等待，等待某一事件的发生，（例如等待某外设的 I/O 操作，等待某共享资源由暂不能使用变成能使用状态，等待定时脉冲的到来或等待超时信号的到来以结束目前的等待，等等）。最后，发生中断时，CPU 提供相应的中断服务，原来正在运行的任务暂不能运行，就进入了被中断状态。图 2.3 表示μC/OS-Ⅱ中一些函数提供的服务，这些函数使任务从一种状态变到另一种状态。

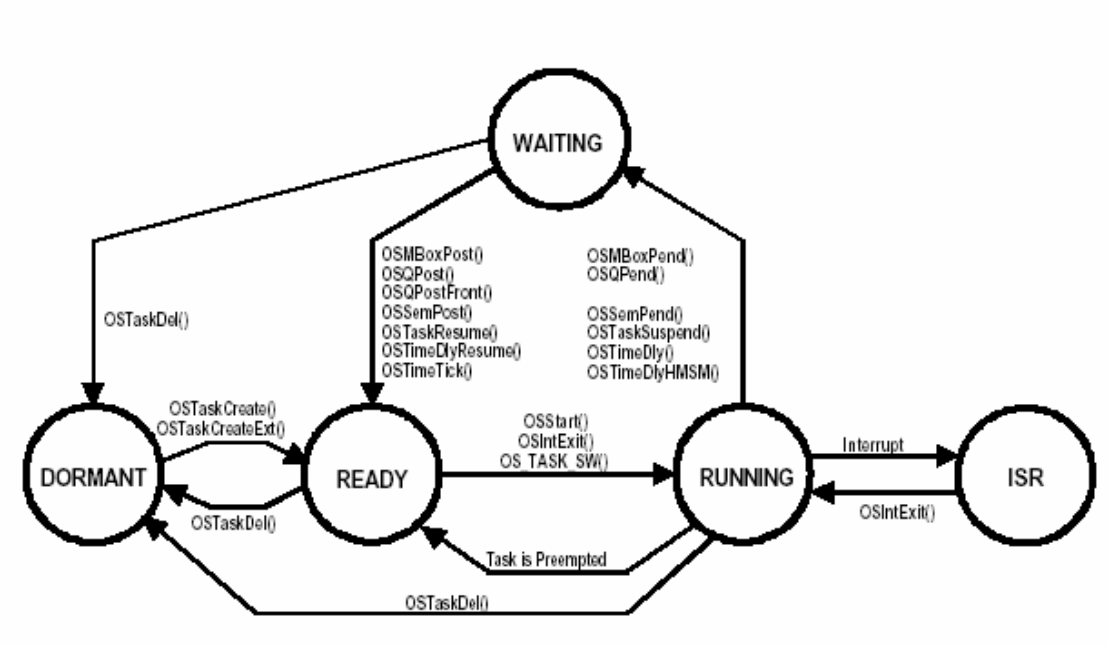

图 2.3 任务的状态

2.6 任务切换*(Context Switch or Task Switch)*

Context Switch 在有的书中翻译成上下文切换，实际含义是任务切换，或 CPU 寄存器内容切换。当多任务内核决定运行另外的任务时，它保存正在运行任务的当前状态（Context），即 CPU 寄存器中的全部内容。这些内容保存在任务的当前状况保存区（Task's Context Storage area），也就是任务自己的栈区之中。（见图 2.2）。入栈工作完成以后，就是把下一个将要运行的任务的当前状况从该任务的栈中重新装入 CPU 的寄存器，并开始下一个任务的运行。这个过程叫做任务切换。任务切换过程增加了应用程序的额外负荷。CPU 的内部寄存器越多，额外负荷就越重。做任务切换所需要的时间取决于 CPU 有多少寄存器要入栈。实时内核的性能不应该以每秒钟能做多少次任务切换来评价。

2.7 内核（Kernel）

多任务系统中，内核负责管理各个任务，或者说为每个任务分配 CPU 时间，并且负责任务之间的通讯。内核提供的基本服务是任务切换。之所以使用实时内核可以大大简化应用系统的设计，是因为实时内核允许将应用分成若干个任务，由实时内核来管理它们。内核本身也增加了应用程序的额外负荷，代码空间增加 ROM 的用量，内核本身的数据结构增加了 RAM 的用量。但更主要的是，每个任务要有自己的栈空间，这一块吃起内存来是相当厉害的。内核本身对 CPU 的占用时间一般在 2 到 5 个百分点之间。

单片机一般不能运行实时内核，因为单片机的 RAM 很有限。通过提供必不可缺少 的系统服务，诸如信号量管理，邮箱、消息队列、延时等，实时内核使得 CPU 的利用更为有效。一旦读者用实时内核做过系统设计，将决不再想返回到前后台系统。

2.8 调度（*Scheduler*）

调度（Scheduler）,英文还有一词叫 dispatcher，也是调度的意思。这是内核的主要职责之一，就是要决定该轮到哪个任务运行了。多数实时内核是基于优先级调度法的。每个任务根据其重要程度的不同被赋予一定的优先级。基于优先级的调度法指，CPU 总是让处在就绪态的优先级最高的任务先运行。然而，究竟何时让高优先级任务掌握 CPU 的使用权，有两种不同的情况，这要看用的是什么类型的内核，是不可剥夺型的还是可剥夺型内核。

2.9 不可剥夺型内核（Non-Preemptive Kernel）

不可剥夺型内核要求每个任务自我放弃 CPU 的所有权。不可剥夺型调度法也称作合作型多任务，各个任务彼此合作共享一个 CPU。异步事件还是由中断服务来处理。中断服务可以使一个高优先级的任务由挂起状态变为就绪状态。但中断服务以后控制权还是回到原来被中断了的那个任务，直到该任务主动放弃 CPU 的使用权时，那个高优先级的任务才能获得 CPU 的使用权。

不可剥夺型内核的一个优点是响应中断快。在讨论中断响应时会进一步涉及这个问题。在任务级，不可剥夺型内核允许使用不可重入函数。函数的可重入性以后会讨论。每个任务都可以调用非可重入性函数，而不必担心其它任务可能正在使用该函数，从而造成数据的破坏。因为每个任务要运行到完成时才释放 CPU 的控制权。当然该不可重入型函数本身不得有放弃 CPU 控制权的企图。

使用不可剥夺型内核时，任务级响应时间比前后台系统快得多。此时的任务级响应时间取决于最长的任务执行时间。

不可剥夺型内核的另一个优点是，几乎不需要使用信号量保护共享数据。运行着的任务占有 CPU，而不必担心被别的任务抢占。但这也不是绝对的，在某种情况下，信号量还是用得着的。处理共享 I/O 设备时仍需要使用互斥型信号量。例如，在打印机的使用上，仍需要满足互斥条件。图 2.4 示意不可剥夺型内核的运行情况，任务在运行过程之中，[L2.4(1)]

中断来了，如果此时中断是开着的，CPU 由中断向量[F2.4(2)]进入中断服务子程序，中断服务子程序做事件处理[F2.4(3)]，使一个有更高级的任务进入就绪态。中断服务完成以后，中断返回指令[F2.4(4)]，使 CPU 回到原来被中断的任务，接着执行该任务的代码[F2.4(5)]直到该任务完成，调用一个内核服务函数以释放 CPU 控制权，由内核将控制权交给那个优先级更高的、并已进入就绪态的任务[F2.4(6)]，这个优先级更高的任务才开始处理中断服务程序标识的事件[F2.4(7)]。

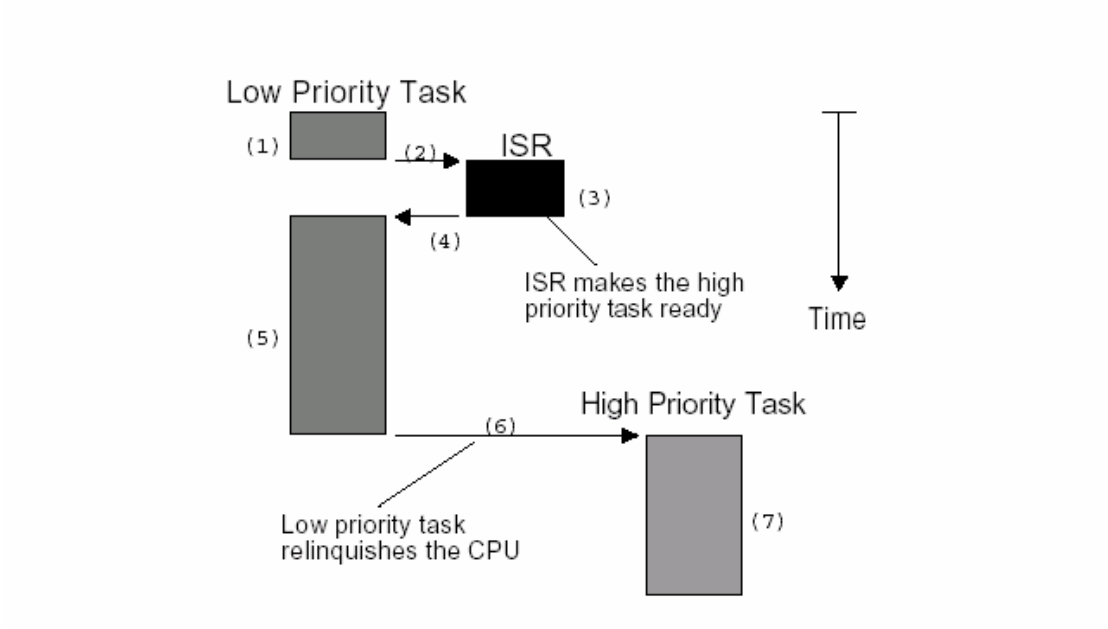

图 2.4 不可剥夺型内核

不可剥夺型内核的最大缺陷在于其响应时间。高优先级的任务已经进入就绪态，但还不能运行，要等，也许要等很长时间，直到当前运行着的任务释放 CPU。与前后系统一样，不可剥夺型内核的任务级响应时间是不确定的，不知道什么时候最高优先级的任务才能拿到 CPU 的控制权，完全取决于应用程序什么时候释放 CPU。

总之，不可剥夺型内核允许每个任务运行，直到该任务自愿放弃 CPU 的控制权。中断可以打入运行着的任务。中断服务完成以后将 CPU 控制权还给被中断了的任务。任务级响应时间要大大好于前后系统，但仍是不可知的，商业软件几乎没有不可剥夺型内核。

2.10 可剥夺型内核

当系统响应时间很重要时，要使用可剥夺型内核。因此，μC/OS-Ⅱ以及绝大多数商业上销售的实时内核都是可剥夺型内核。最高优先级的任务一旦就绪，总能得到 CPU 的控制权。当一个运行着的任务使一个比它优先级高的任务进入了就绪态，当前任务的 CPU 使用权就被剥夺了，或者说被挂起了，那个高优先级的任务立刻得到了 CPU 的控制权。如果是中断服务子程序使一个高优先级的任务进入就绪态，中断完成时，中断了的任务被挂起，优先级高的

那个任务开始运行。如图 2.5 所示。

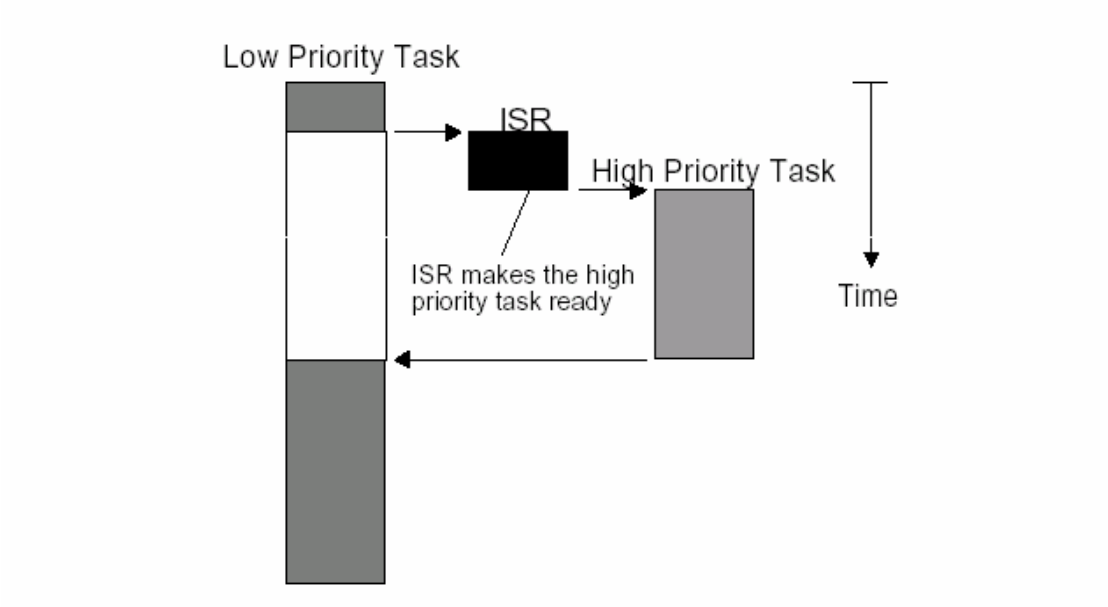

图 2.5 可剥夺型内核

使用可剥夺型内核，最高优先级的任务什么时候可以执行，可以得到 CPU 的控制权是可知的。使用可剥夺型内核使得任务级响应时间得以最优化。

使用可剥夺型内核时，应用程序不应直接使用不可重入型函数。调用不可重入型函数时，要满足互斥条件，这一点可以用互斥型信号量来实现。如果调用不可重入型函数时，低优先级的任务 CPU 的使用权被高优先级任务剥夺，不可重入型函数中的数据有可能被破坏。综上所述，可剥夺型内核总是让就绪态的高优先级的任务先运行，中断服务程序可以抢占 CPU，到中断服务完成时，内核让此时优先级最高的任务运行（不一定是那个被中断了的任务）。任务级系统响应时间得到了最优化，且是可知的。**μC/OS-Ⅱ属于可剥夺型内核**。

2.11 可重入性（*Reentrancy*）

可重入型函数可以被一个以上的任务调用，而不必担心数据的破坏。可重入型函数任何时候都可以被中断，一段时间以后又可以运行，而相应数据不会丢失。可重入型函数或者只使用局部变量，即变量保存在 CPU 寄存器中或堆栈中。如果使用全局变量，则要对全局变量予以保护。程序 2.1 是一个可重入型函数的例子。

程序清单 2.1 可重入型函数

```
void strcpy(char *dest, char *src)
{
    while (*dest++ = *src++) {
        ;
```

```
    }
    *dest = NUL;
}
```

函数 Strcpy()做字符串复制。因为参数是存在堆栈中的，故函数 Strcpy()可以被多个任务调用，而不必担心各任务调用函数期间会互相破坏对方的指针。

不可重入型函数的例子如程序 2.2 所示。Swap()是一个简单函数，它使函数的两个形式变量的值互换。为便于讨论，假定使用的是可剥夺型内核，中断是开着的，Temp 定义为整数全程变量。

程序清单 2.2　　不可重入型函数

```
int Temp;

void swap(int *x, int *y)
{
    Temp = *x;
    *x   = *y;
    *y   = Temp;
}
```

程序员打算让 Swap() 函数可以为任何任务所调用，如果一个低优先级的任务正在执行 Swap()函数，而此时中断发生了，于是可能发生的事情如图 2.6 所示。[F2.6(1)]表示中断发生时 Temp 已被赋值 1，中断服务子程序使更优先级的任务就绪，当中断完成时[F2.6(2)]，内核（假定使用的是μC/OS-Ⅱ）使高优先级的那个任务得以运行[F2.6(3)],高优先级的任务调用 Swap()函数是 Temp 赋值为 3。这对该任务本身来说，实现两个变量的交换是没有问题的，交换后 Z 的值是 4，X 的值是 3。然后高优先级的任务通过调用内核服务函数中的延迟一个时钟节拍[F2.6(4)],释放了 CPU 的使用权，低优先级任务得以继续运行[F2.6(5)].注意，此时 Temp 的值仍为 3！在低优先级任务接着运行时，Y 的值被错误地赋为 3，而不是正确值 1。

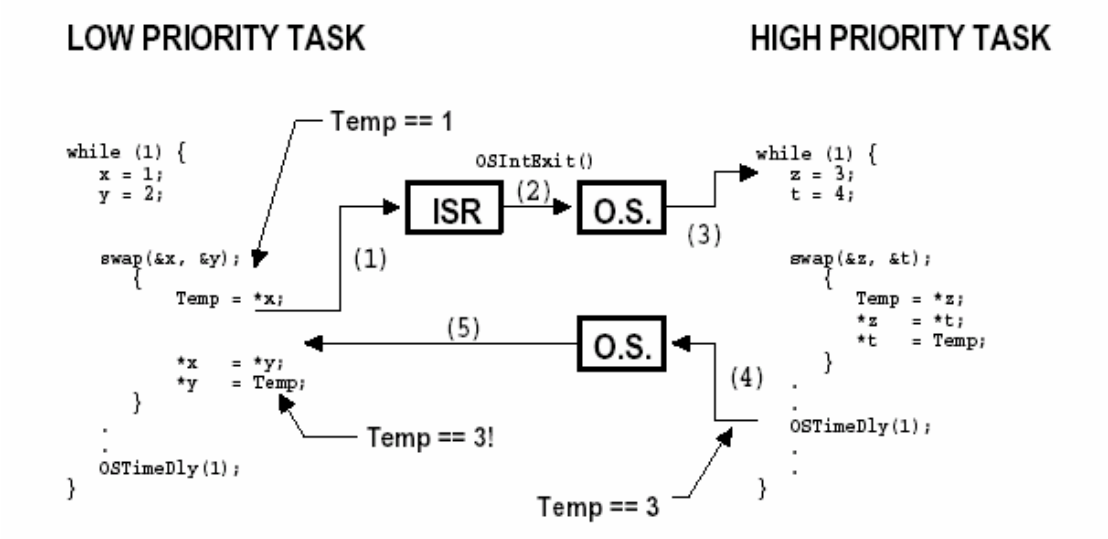

图 2.6 不可重入性函数

请注意，这只是一个简单的例子，如何能使代码具有可重入性一看就明白。然而有些情况下，问题并非那么易解。应用程序中的不可重入函数引起的错误很可能在测试时发现不了，直到产品到了现场问题才出现。如果在多任务上您还是把新手，使用不可重入型函数时，千万要当心。

使用以下技术之一即可使 Swap()函数具有可重入性：

- **把 Temp 定义为局部变量**
- **调用 Swap()函数之前关中断，调动后再开中断**
- **用信号量禁止该函数在使用过程中被再次调用**

如果中断发生在 Swap()函数调用之前或调用之后，两个任务中的 X，Y 值都会是正确的。

2.12 时间片轮番调度法

当两个或两个以上任务有同样优先级，内核允许一个任务运行事先确定的一段时间，叫做时间额度（*quantum*），然后切换给另一个任务。也叫做时间片调度。内核在满足以下条件时，把 CPU 控制权交给下一个任务就绪态的任务：

- 当前任务已无事可做
- 当前任务在时间片还没结束时已经完成了。

目前，μC/OS-Ⅱ不支持时间片轮番调度法。应用程序中各任务的优先级必须互不相同。

2.13 任务优先级

每个任务都有其优先级。任务越重要，赋予的优先级应越高。

2.14 静态优先级

应用程序执行过程中诸任务优先级不变，则称之为静态优先级。在静态优先级系统中，诸任务以及它们的时间约束在程序编译时是已知的。

2.15 动态优先级

应用程序执行过程中，任务的优先级是可变的，则称之为动态优先级。实时内核应当避免出现优先级反转问题。

2.16 优先级反转

使用实时内核,优先级反转问题是实时系统中出现得最多的问题。图 2.7 解释优先级反转是如何出现的。如图，任务 1 优先级高于任务 2，任务 2 优先级高于任务 3。任务 1 和任务 2 处于挂起状态，等待某一事件的发生，任务 3 正在运行如[图 2.7(1)]。此时，任务 3 要使用其共享资源。使用共享资源之前，首先必须得到该资源的信号量(Semaphore)(见 2.18.04 信号量)。任务 3 得到了该信号量，并开始使用该共享资源[图 2.7（2）]。由于任务 1 优先级高，它等待的事件到来之后剥夺了任务 3 的 CPU 使用权[图 2.7(3)]，任务 1 开始运行[图 2.7(4)]。运行过程中任务 1 也要使用那个任务 3 正在使用着的资源，由于该资源的信号量还被任务 3 占用着,任务 1 只能进入挂起状态,等待任务 3 释放该信号量[图 2.7(5)]。任务 3 得以继续运行[图 2.7(6)]。由于任务 2 的优先级高于任务 3，当任务 2 等待的事件发生后,任务 2 剥夺了任务 3 的 CPU 的使用权[图 2.7(7)]并开始运行。处理它该处理的事件[图 2.7(8)],直到处理完之后将 CPU 控制权还给任 3[图 2.7(9)]。任务 3 接着运行[图 2.7(10)],直到释放那个共享资源的信号量[图 27(11)]。直到此时,由于实时内核知道有个高优先级的任务在等待这个信号量，内核做任务切换，使任务 1 得到该信号量并接着运行[图 2.7(12)]。

在这种情况下,任务 1 优先级实际上降到了任务 3 的优先级水平。因为任务 1 要等,直等到任务 3 释放占有的那个共享资源。由于任务 2 剥夺任务 3 的 CPU 使用权,使任务 1 的状况更加恶化,任务 2 使任务 1 增加了额外的延迟时间。任务 1 和任务 2 的优先级发生了反转。

纠正的方法可以是,在任务 3 使用共享资源时,提升任务 3 的优先级。任务完成时予以恢复。任务 3 的优先级必须升至最高,高于允许使用该资源的任何任务。多任务内核应允许动态改变任务的优先级以避免发生优先级反转现象。然而改变任务的优先级是很花时间的。如果任务 3 并没有先被任务 1 剥夺 CPU 使用权，又被任务 2 抢走了 CPU 使用权,花很多时间在共享资源使用前提升任务 3 的优先级,然后又在资源使用后花时间恢复任务 3 的优先级,则无形中浪费了很多 CPU 时间。真正需要的是,为防止发生优先级反转,内核能自动变换任务的优

先级,这叫做优先级继承(Priority inheritance)但μC/OS-II不支持优先级继承,一些商业内核有优先级继承功能。

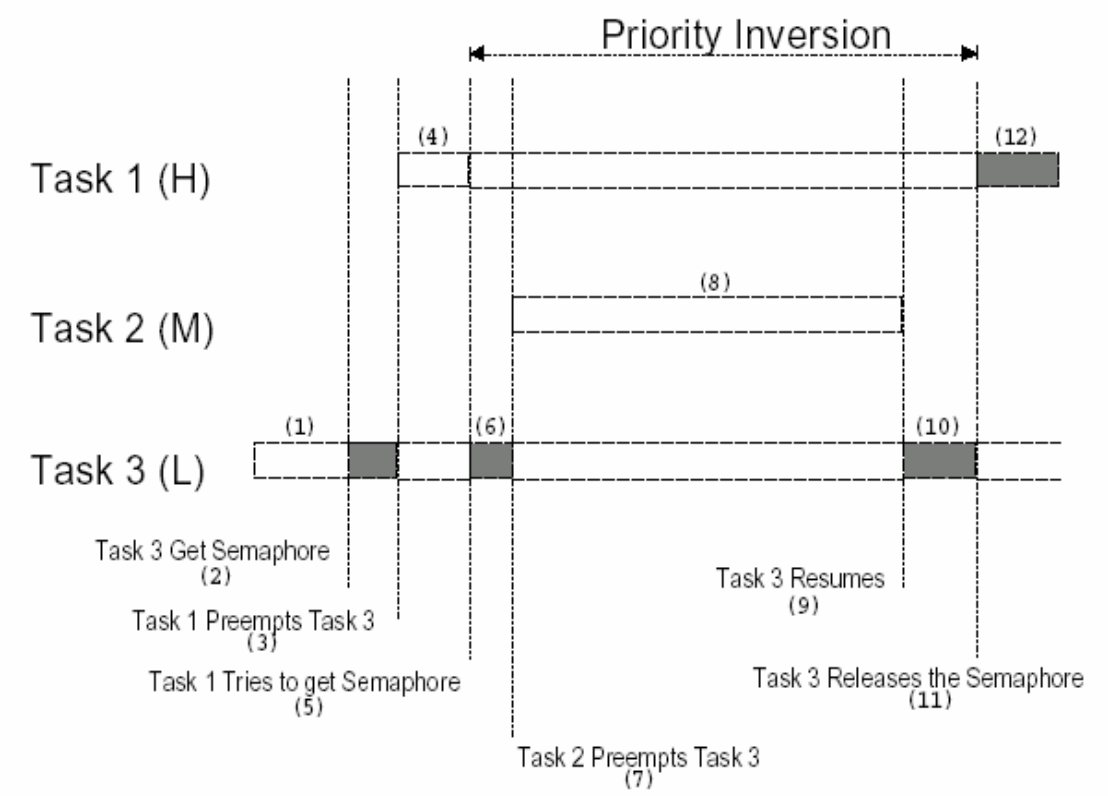

图 2.7 优先级反转问题

图 2.8 解释如果内核支持优先级继承的话,在上述例子中会是怎样一个过程。任务 3 在运行[图 2.8(1)],任务 3 申请信号量以获得共享资源使用权[图 2.8(2)],任务 3 得到并开始使用共享资源[图 2.8(3)]。后来 CPU 使用权被任务 1 剥夺[图 2.8(4)],任务 1 开始运行[图 2.8(5)],任务 1 申请共享资源信号量[图 2.8(6)]。此时,内核知道该信号量被任务 3 占用了,而任务 3 的优先级比任务 1 低,内核于是将任务 3 的优先级升至与任务 1 一样,,然而回到任务 3 继续运行,使用该共享资源[图 2.7(7)],直到任务 3 释放共享资源信号量[图 2。8(8)]。这时,内核恢复任务 3 本来的优先级并把信号量交给任务 1,任务 1 得以顺利运行。 [图 2.8(9)],任务 1 完成以后[图 2.8(10)]那些任务优先级在任务 1 与任务 3 之间的任务例如任务 2 才能得到 CPU 使用权,并开始运行 [图 2.8(11)]。注意,任务 2 在从[图 2.8(3)]到[图 2.8(10)]的任何一刻都有可能进入就绪态,并不影响任务 1、任务 3 的完成过程。在某种程度上,任务 2 和任务 3 之间也还是有不可避免的优先级反转。

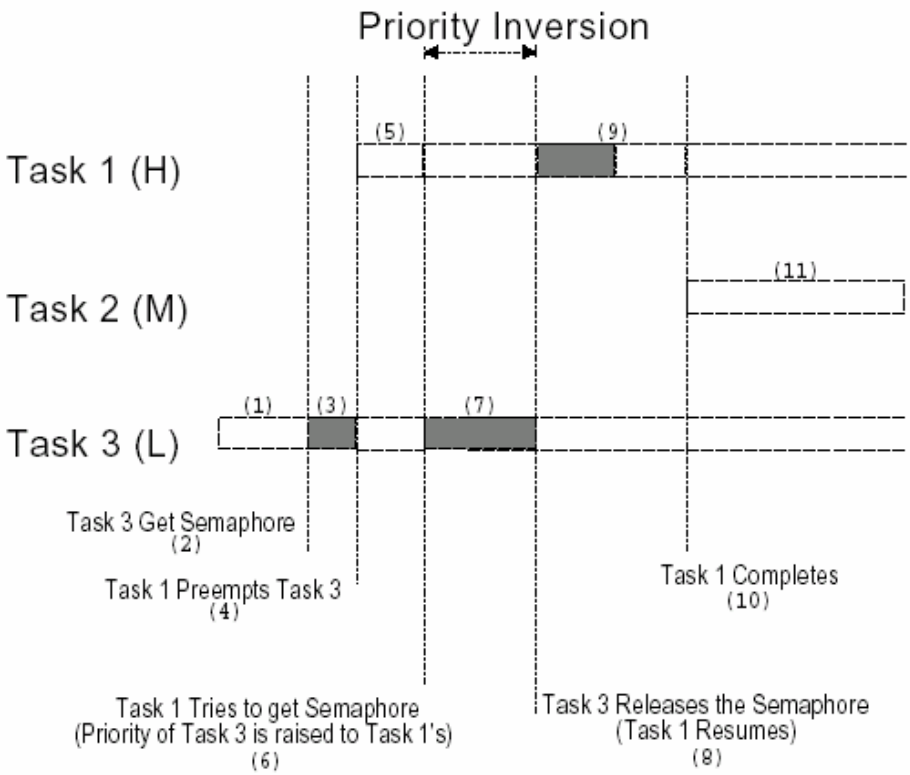

Figure 2-8, Kernel that supports priority inheritance.

图 2.8

2.17 任务优先级分配

给任务定优先级可不是件小事，因为实时系统相当复杂。许多系统中，并非所有的任务都至关重要。不重要的任务自然优先级可以低一些。实时系统大多综合了软实时和硬实时这两种需求。软实时系统只是要求任务执行得尽量快，并不要求在某一特定时间内完成。硬实时系统中，任务不但要执行无误，还要准时完成。

一项有意思的技术可称之为单调执行率调度法 RMS(Rate Monotonic Scheduling),用于分配任务优先级。这种方法基于哪个任务执行的次数最频繁,执行最频繁的任务优先级最高。见图 2.9。

图 2.9 基于任务执行频繁度的优先级分配法

任务执行频繁度（Hz）

RMS 做了一系列假设：

- 所有任务都是周期性的
- 任务间不需要同步，没有共享资源,没有任务间数据交换等问题
- CPU 必须总是执行那个优先级最高且处于就绪态的任务。换句话说，要使用可剥夺型调度法。

给出一系列 n 值表示系统中的不同任务数，要使所有的任务满足硬实时条件，必须使不等式[2.1]成立，这就是 RMS 定理：

[2.1] $$\sum_i \frac{Ei}{Ti} \le n(2^{1/n} - 1)$$

这里 E_i 是任务 i 最长执行时间，T_i 是任务 i 的执行周期。换句话说，E_i/T_i 是任务 i 所需的 CPU 时间。表 2.1 给出 $n(2^{1/n}-1)$ 的值，n 是系统中的任务数。对于无穷多个任务，极限值是 $\lambda n(2)$ 或 0.693。这就意味着，基于 RMS，要任务都满足硬实时条件，所有有时间条件要求的任务 i 总的 CPU 利用时间应小于 70%！请注意，这是指有时间条件要求的任务，系统中当然还可以有对时间没有什么要求的任务，使得 CPU 的利用率达到 100%。使 CPU 利用率达到 100%并不好，因为那样的话程序就没有了修改的余地，也没法增加新功能了。作为系统设计的一条原则，CPU 利用率应小于 60%到 70%。

RMS 认为最高执行率的任务具有最高的优先级，但最某些情况下，最高执行率的任务并非是最重要的任务。如果实际应用都真的像 RMS 说的那样，也就没有什么优先级分配可讨论了。然而讨论优先级分配问题，RMS 无疑是一个有意思的起点。

表 2.1 基于任务到CPU 最高允许使用率

任务数	$n(2^{1/n} - 1)$
1	1.000
2	0.828
3	0.779
4	0.756
5	0.743
.	.
.	.
.	.
∞	0.693

2.18 互斥条件

实现任务间通讯最简便到办法是使用共享数据结构。特别是当所有到任务都在一个单一

地址空间下，能使用全程变量、指针、缓冲区、链表、循环缓冲区等，使用共享数据结构通讯就更为容易。虽然共享数据区法简化了任务间的信息交换，但是必须保证每个任务在处理共享数据时的排它性，以避免竞争和数据的破坏。与共享资源打交道时，使之满足互斥条件最一般的方法有：

- 关中断
- 使用测试并置位指令
- 禁止做任务切换
- 利用信号量

2.18.1 关中断和开中断

处理共享数据时保证互斥，最简便快捷的办法是关中断和开中断。如示意性代码程序 2.3 所示：

程序清单 2.3　　关中断和开中断

```
Disable interrupts;                                    /*关中断*/
Access the resource (read/write from/to variables);    /*读/写变量*/
Reenable interrupts;                                   /*重新允许中断*/
```

μC/OS-II 在处理内部变量和数据结构时就是使用的这种手段，即使不是全部，也是绝大部分。实际上μC/OS-II提供两个宏调用，允许用户在应用程序的 C 代码中关中断然后再开中断：OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()[参见 8.03.02 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICALL()],这两个宏调用的使用法见程序 2.4

程序清单 2.4 利用μC/OS_Ⅱ 宏调用关中断和开中断

```
void Function (void)
{
    OS_ENTER_CRITICAL();
    .
   .    /*在这里处理共享数据*/
    .
    OS_EXIT_CRITICAL();
}
```

可是，必须十分小心，关中断的时间不能太长。因为它影响整个系统的中断响应时间，

即中断延迟时间。当改变或复制某几个变量的值时，应想到用这种方法来做。这也是在中断服务子程序中处理共享变量或共享数据结构的唯一方法。在任何情况下，关中断的时间都要尽量短。

如果使用某种实时内核，一般地说，关中断的最长时间不超过内核本身的关中断时间，就不会影响系统中断延迟。当然得知道内核里中断关了多久。凡好的实时内核，厂商都提供这方面的数据。总而言之，要想出售实时内核，时间特性最重要。

2.18.2 测试并置位

如果不使用实时内核，当两个任务共享一个资源时，一定要约定好，先测试某一全程变量，如果该变量是 0，允许该任务与共享资源打交道。为防止另一任务也要使用该资源，前者只要简单地将全程变量置为 1，这通常称作测试并置位(Test-And-Set),或称作 TAS。TAS 操作可能是微处理器的单独一条不会被中断的指令,或者是在程序中关中断做 TAS 操作再开中断，如程序清单 2.5 所示。

程序清单 2.5 利用测试并置位处理共享资源

```
Disable interrupts;                                关中断
if ('Access Variable' is 0) {                      如果资源不可用，标志为0
    Set variable to 1;                             置资源不可用，标志为1
    Reenable interrupts;                           重开中断
    Access the resource;                           处理该资源
    Disable interrupts;                            关中断
    Set the 'Access Variable' back to 0;           清资源不可使用，标志为0
    Reenable interrupts;                           重新开中断
} else {                                           否则
    Reenable interrupts;                           开中断
   /* You don't have access to the resource, try back later; */
   /* 资源不可使用，以后再试; */
}
```

有的微处理器有硬件的 TAS 指令(如 Motorola 68000 系列,就有这条指令)

2.18.3 禁止,然后允许任务切换

如果任务不与中断服务子程序共享变量或数据结构,可以使用禁止、然后允许任务切换。(参见 3.06 给任务切换上锁和开锁)。如程序清单 2.6 所示，以 μC/OS-Ⅱ的使用为例，两个或两个以上的任务可以共享数据而不发生竞争。注意，此时虽然任务切换是禁止了，但中断

还是开着的。如果这时中断来了，中断服务子程序会在这一临界区内立即执行。中断服务子程序结束时，尽管有优先级高的任务已经进入就绪态，内核还是返回到原来被中断了的任务。直到执行完给任务切换开锁函数 OSSchedUnlock (),内核再看有没有优先级更高的任务被中断服务子程序激活而进入就绪态，如果有，则做任务切换。虽然这种方法是可行的，但应该尽量避免禁止任务切换之类操作，因为内核最主要的功能就是做任务的调度与协调。禁止任务切换显然与内核的初衷相违。应该使用下述方法。

程序清单2.6 用给任务切换上锁，然后开锁的方法实现数据共享

```
void Function (void)
{
    OSSchedLock();
    .
    .    /* You can access shared data in here (interrupts are
recognized) */
    .    /*在这里处理共享数据(中断是开着的)这里的中断是与此函数部共享数据结
构的中断*/
    OSSchedUnlock();
}
```

2.18.4 信号量(Semaphores)

信号量是 60 年代中期 Edgser Dijkstra 发明的。信号量实际上是一种约定机制，在多任务内核中普遍使用.信号量用于：

- 控制共享资源的使用权(满足互斥条件)
- 标志某事件的发生
- 使两个任务的行为同步

(译者注：信号与信号量在英文中都叫做 Semaphore，并不加以区分，而说它有两种类型，二进制型(binary)和计数器型(counting)。本书中的二进制型信号量实际上是只取两个值 0 和 1 的信号量。实际上 这个信号量只有一位，这种信号量翻译为信号更为贴切。而二进制信号量通常指若干位的组合。而本书中解释为事件标志的置位与清除(见 2.21))。

信号像是一把钥匙，任务要运行下去，得先拿到这把钥匙。如果信号已被别的任务占用，该任务只得被挂起，直到信号被当前使用者释放。换句话说，申请信号的任务是在说：“把钥匙给我，如果谁正在用着，我只好等!”信号是只有两个值的变量，信号量是计数式的。只取两个值的信号是只有两个值 0 和 1 的量，因此也称之为信号量。计数式信号量的值可以是 0 到 255 或 0 到 65535，或 0 到 4294967295，取决于信号量规约机制使用的是 8 位、16 位还是 32 位。到底是几位，实际上是取决于用的哪种内核。根据信号量的值，内核跟踪那些等待信号量的任务。

一般地说，对信号量只能实施三种操作：初始化(INITIALIZE)，也可称作建立(CREATE)；

等信号(WAIT)也可称作挂起(PEND)；给信号(SIGNAL)或发信号(POST)。信号量初始化时要给信号量赋初值，等待信号量的任务表(Waiting list)应清为空。

想要得到信号量的任务执行等待(WAIT)操作。如果该信号量有效(即信号量值大于 0)，则信号量值减 1，任务得以继续运行。如果信号量的值为 0，等待信号量的任务就被列入等待信号量任务表。多数内核允许用户定义等待超时，如果等待时间超过了某一设定值时，该信号量还是无效，则等待信号量的任务进入就绪态准备运行，并返回出错代码(指出发生了等待超时错误)。

任务以发信号操作(SIGNAL)释放信号量。如果没有任务在等待信号量，信号量的值仅仅是简单地加 1。如果有任务在等待该信号量，那么就会有一个任务进入就绪态，信号量的值也就不加 1。于是钥匙给了等待信号量的诸任务中的一个任务。至于给了那个任务，要看内核是如何调度的。收到信号量的任务可能是以下两者之一。

- 等待信号量任务中优先级最高的，或者是
- 最早开始等待信号量的那个任务，即按先进先出的原则(First In First Out，FIFO)

有的内核有选择项，允许用户在信号量初始化时选定上述两种方法中的一种。但μC/OS-Ⅱ只支持优先级法。如果进入就绪态的任务比当前运行的任务优先级高(假设，是当前任务释放的信号量激活了比自己优先级高的任务)。则内核做任务切换(假设，使用的是可剥夺型内核)，高优先级的任务开始运行。当前任务被挂起。直到又变成就绪态中优先级最高任务。

程序清单 2.7 示意在μC/OS-Ⅱ中如何用信号量处理共享数据。要与同一共享数据打交道的任务调用等待信号量函数 OSSemPend()。处理完共享数据以后再调用释放信号量函数 OSSemPost()。这两个函数将在以后的章节中描述。要注意的是，在使用信号量之前，一定要对该信号量做初始化。作为互斥条件，信号量初始化为 1。使用信号量处理共享数据不增加中断延迟时间，如果中断服务程序或当前任务激活了一个高优先级的任务，高优先级的任务立即开始执行。

程序清单 2.7　通过获得信号量处理共享数据

```
OS_EVENT *SharedDataSem;
void Function (void)
{
    INT8U err;
    OSSemPend(SharedDataSem, 0, &err);
    .
    .    /* You can access shared data in here (interrupts are
recognized) */
    .    /*共享数据的处理在此进行，(中断是开着的)*/
    OSSemPost(SharedDataSem);
}
```

当诸任务共享输入输出设备时，信号量特别有用。可以想象，如果允许两个任务同时给

打印机送数据时会出现什么现象。打印机会打出相互交叉的两个任务的数据。例如任务 1 要打印“I am Task!”，而任务 2 要打印“I am Task2!”可能打印出来的结果是：“I Ia amm T Tasask k1!2!”

在这种情况下，使用信号量并给信号量赋初值 1(用二进制信号量)。规则很简单，要想使用打印机的任务，先要得到该资源的信号量。图 2.10 两个任务竞争得到排它性打印机使用权，图中信号量用一把钥匙表示，想使用打印机先要得到这把钥匙。

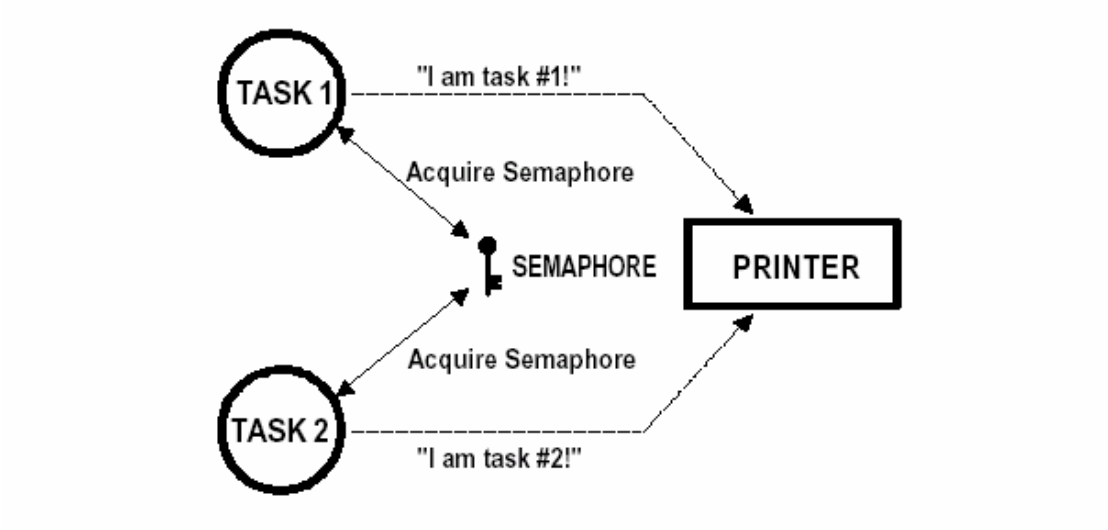

图 2.10 用获取信号量来得到打印机使用权

上例中，每个任务都知道有个信号表示资源可不可以使用。要想使用该资源，要先得到这个信号。然而有些情况下，最好把信号量藏起来，各个任务在同某一资源打交道时，并不知道实际上是在申请得到一个信号量。例如，多任务共享一个 RS-232C 外设接口，各任务要送命令给接口另一端的设备并接收该设备的回应。如图 2.11 所示。

调用向串行口发送命令的函数 CommSendCmd()，该函数有三个形式参数：Cmd 指向送出的 ASCII 码字符串命令。Response 指向外设回应的字符串。timeout 指设定的时间间隔。如果超过这段时间外设还不响应,则返回超时错误信息。函数的示意代码如程序清单 2.8 所示。

程序清单 2.8　隐含的信号量。

```
INT8U CommSendCmd(char *cmd, char *response, INT16U timeout)
{
    Acquire port's semaphore;
    Send command to device;
    Wait for response (with timeout);
    if (timed out) {
        Release semaphore;
        return (error code);
```

```
    } else {
        Release semaphore;
        return (no error);
    }
}
```

要向外设发送命令的任务得调用上述函数。设信号量初值为 1，表示允许使用。初始化是在通讯口驱动程序的初始化部分完成的。第一个调用 CommSendCmd()函数的任务申请并得到了信号量，开始向外设发送命令并等待响应。而另一个任务也要送命令，此时外设正“忙”，则第二个任务被挂起，直到该信号量重新被释放。第二个任务看起来同调用了一个普通函数一样，只不过这个函数在没有完成其相应功能时不返回。当第一个任务释放了那个信号量，第二个任务得到了该信号量，第二个任务才能使用 RS-232 口。

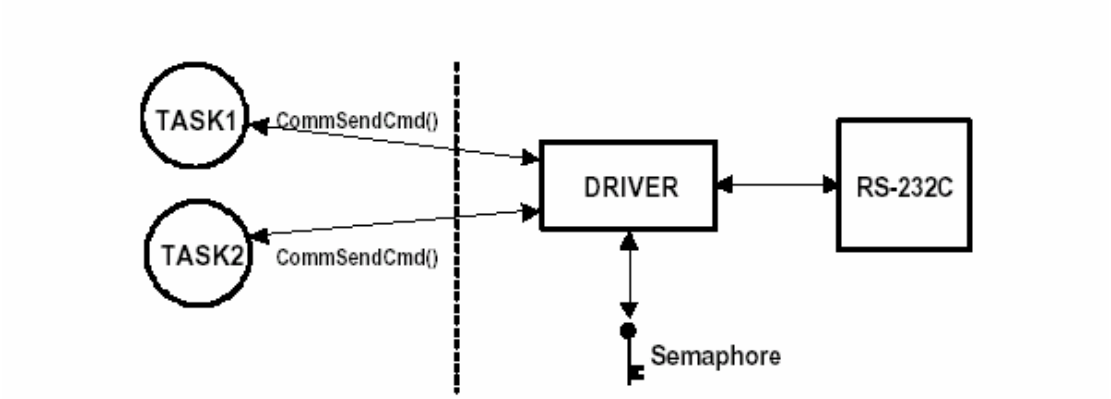

图 2.11 在任务级看不到隐含的信号量

计数式信号量用于某资源可以同时为几个任务所用。例如，用信号量管理缓冲区阵列(buffer pool)，如图 2.12 所示。缓冲区阵列中共有 10 个缓冲区，任务通过调用申请缓冲区函数 BufReq()向缓冲区管理方申请得到缓冲区使用权。当缓冲区使用权还不再需要时，通过调用释放缓冲区函数 BufRel()将缓冲区还给管方。函数示意码如程序清单 2.9 所示

程序清单 2.9　用信号量管理缓冲区。

```
BUF *BufReq(void)
{
   BUF *ptr;

   Acquire a semaphore;
   Disable interrupts;
   ptr        = BufFreeList;
```

```
   BufFreeList = ptr->BufNext;
   Enable interrupts;
   return (ptr);
}


void BufRel(BUF *ptr)
{
   Disable interrupts;
   ptr->BufNext = BufFreeList;
   BufFreeList  = ptr;
   Enable interrupts;
   Release semaphore;
}
```

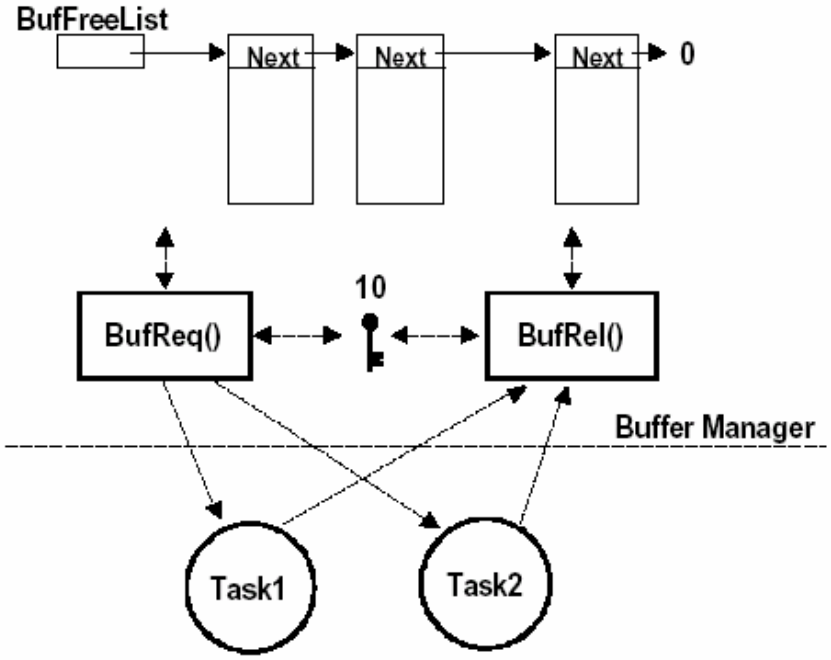

图 2.12 计数式信号量的用法

缓冲区阵列管理方满足前十个申请缓冲区的任务，就好像有 10 把钥匙可以发给诸任务。当所有的钥匙都用完了，申请缓冲区的任务被挂起，直到信号量重新变为有效。缓冲区管理程序在处理链表指针时，为满足互斥条件，中断是关掉的(这一操作非常快)。任务使用完某一缓冲区，通过调用缓冲区释放函数 BufRel()将缓冲区还给系统。系统先将该缓冲区指针

插入到空闲缓冲区链表中(Linked list)然后再给信号量加 1 或释放该信号量。这一过程隐含在缓冲区管理程序 BufReq()和 BufRel()之中，调用这两个函数的任务不用管函数内部的详细过程。

信号量常被用过了头。处理简单的共享变量也使用信号量则是多余的。请求和释放信号量的过程是要花相当的时间的。有时这种额外的负荷是不必要的。用户可能只需要关中断、开中断来处理简单共享变量，以提高效率。(参见 2.18.0.1 关中断和开中断)。假如两个任务共享一个 32 位的整数变量，一个任务给这个变量加 1，另一个任务给这个变量清 0。如果注意到不管哪种操作，对微处理器来说，只花极短的时间，就不会使用信号量来满足互斥条件了。每个任务只需操作这个任务前关中断，之后再开中断就可以了。然而，如果这个变量是浮点数，而相应微处理器又没有硬件的浮点协处理器，浮点运算的时间相当长，关中断时间长了会影响中断延迟时间，这种情况下就有必要使用信号量了。

2.19 死锁(或抱死)（Deadlock （or Deadly Embrace)）

死锁也称作抱死，指两个任务无限期地互相等待对方控制着的资源。设任务 T1 正独享资源 R1，任务 T2 在独享资源 T2，而此时 T1 又要独享 R2，T2 也要独享 R1，于是哪个任务都没法继续执行了，发生了死锁。最简单的防止发生死锁的方法是让每个任务都：

- 先得到全部需要的资源再做下一步的工作
- 用同样的顺序去申请多个资源
- 释放资源时使用相反的顺序

内核大多允许用户在申请信号量时定义等待超时，以此化解死锁。当等待时间超过了某一确定值，信号量还是无效状态，就会返回某种形式的出现超时错误的代码，这个出错代码告知该任务，不是得到了资源使用权，而是系统错误。死锁一般发生在大型多任务系统中，在嵌入式系统中不易出现。

2.20 同步

可以利用信号量使某任务与中断服务同步(或者是与另一个任务同步，这两个任务间没有数据交换)。如图 2.13 所示。注意，图中用一面旗帜，或称作一个标志表示信号量。这个标志表示某一事件的发生(不再是一把用来保证互斥条件的钥匙)。用来实现同步机制的信号量初始化成 0，信号量用于这种类型同步的称作单向同步(unilateral rendezvous)。一个任务做 I/O 操作，然后等信号回应。当 I/O 操作完成，中断服务程序(或另外一个任务)发出信号，该任务得到信号后继续往下执行。

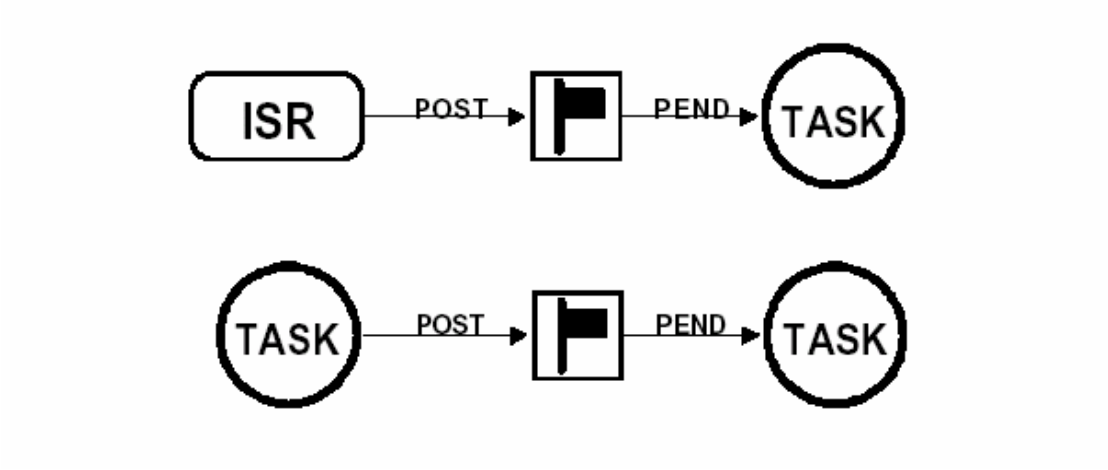

图 2.13 用信号量使任务与中断服务同步

如果内核支持计数式信号量，信号量的值表示尚未得到处理的事件数。请注意，可能会有一个以上的任务在等待同一事件的发生，则这种情况下内核会根据以下原则之一发信号给相应的任务：

- 发信号给等待事件发生的任务中优先级最高的任务，或者
- 发信号给最先开始等待事件发生的那个任务

根据不同的应用，发信号以标识事件发生的中断服务或任务也可以是多个。

两个任务可以用两个信号量同步它们的行为。如图 2.14 所示。这叫做双向同步(bilateral rendezvous)。双向同步同单向同步类似，只是两个任务要相互同步。

例如则程序清单 2.10 中，运行到某一处的第一个任务发信号给第二个任务[L22.10(1)]，然后等待信号返回[L2.10(2)]。同样，当第二个任务运行到某一处时发信号给第一个任务[2.10(3)]等待返回信号[L2.10(4)]。至此，两个任务实现了互相同步。在任务与中断服务之间不能使用双向同步，因为在中断服务中不可能等一个信号量。

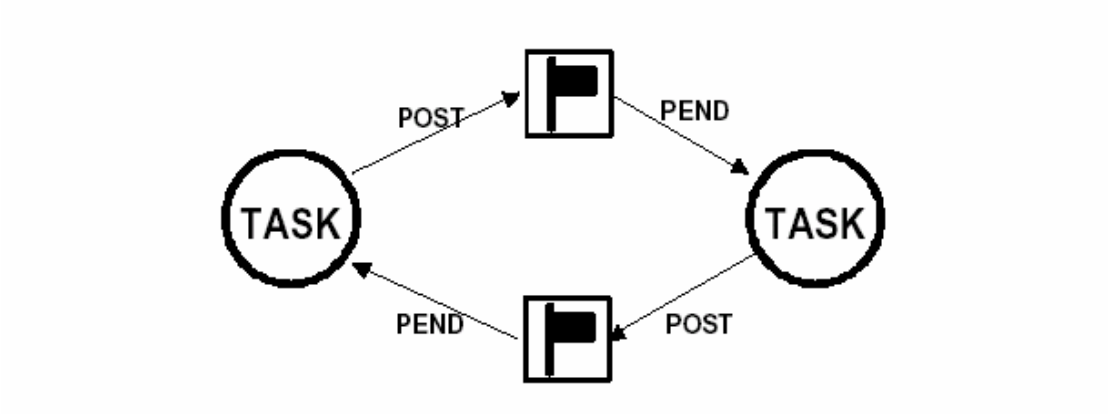

图 2.14 两个任务用信号量同步彼此的行为

程序清单2.10 双向同步

```
Task1()
{
   for (;;) {
      Perform operation;
      Signal task #2;                                    (1)
      Wait for signal from task #2;                      (2)
      Continue operation;
   }
}

Task2()
{
   for (;;) {
      Perform operation;
      Signal task #1;                                    (3)
      Wait for signal from task #1;                      (4)
      Continue operation;
   }
}
```

2.21 事件标志(Event Flags)

当某任务要与多个事件同步时,要使用事件标志。若任务需要与任何事件之一发生同步,可称为独立型同步(即逻辑或关系)。任务也可以与若干事件都发生了同步,称之为关联型(逻辑与关系)。独立型及关联型同步如图2.15所示。

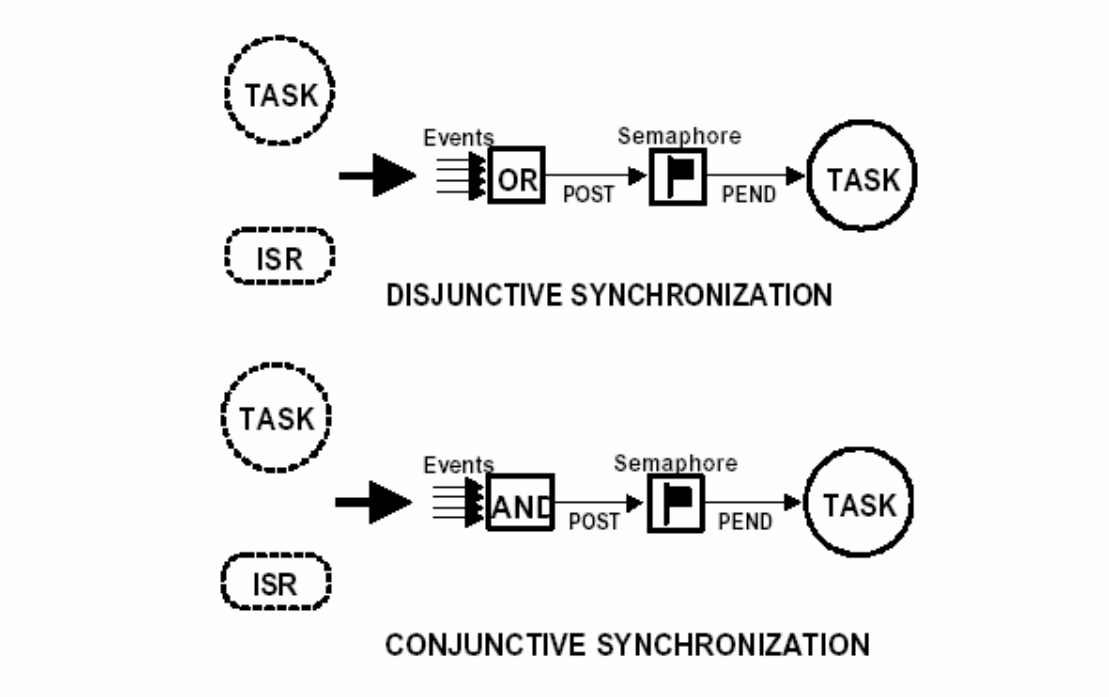

图 2.15 独立型及关联型同步

可以用多个事件的组合发信号给多个任务。如图 2.16 所示，典型地，8 个、16 个或 32 个事件可以组合在一起，取决于用的哪种内核。每个事件占一位(bit)，以 32 位的情况为多。任务或中断服务可以给某一位置位或复位，当任务所需的事件都发生了，该任务继续执行，至于哪个任务该继续执行了，是在一组新的事件发生时辨定的。也就是在事件位置位时做辨断。

内核支持事件标志，提供事件标志置位、事件标志清零和等待事件标志等服务。事件标志可以是独立型或组合型。μC/OS-Ⅱ目前不支持事件标志.

2.22 任务间的通讯(Intertask Communication)

有时很需要任务间的或中断服务与任务间的通讯。这种信息传递称为任务间的通讯。任务间信息的传递有两个途径：通过全程变量或发消息给另一个任务。

用全程变量时，必须保证每个任务或中断服务程序独享该变量。中断服务中保证独享的唯一办法是关中断。如果两个任务共享某变量，各任务实现独享该变量的办法可以是关中断再开中断，或使用信号量(如前面提到的那样)。请注意，任务只能通过全程变量与中断服务程序通讯，而任务并不知道什么时候全程变量被中断服务程序修改了，除非中断程序以信号量方式向任务发信号或者是该任务以查询方式不断周期性地查询变量的值。要避免这种情况，用户可以考虑使用邮箱或消息队列。

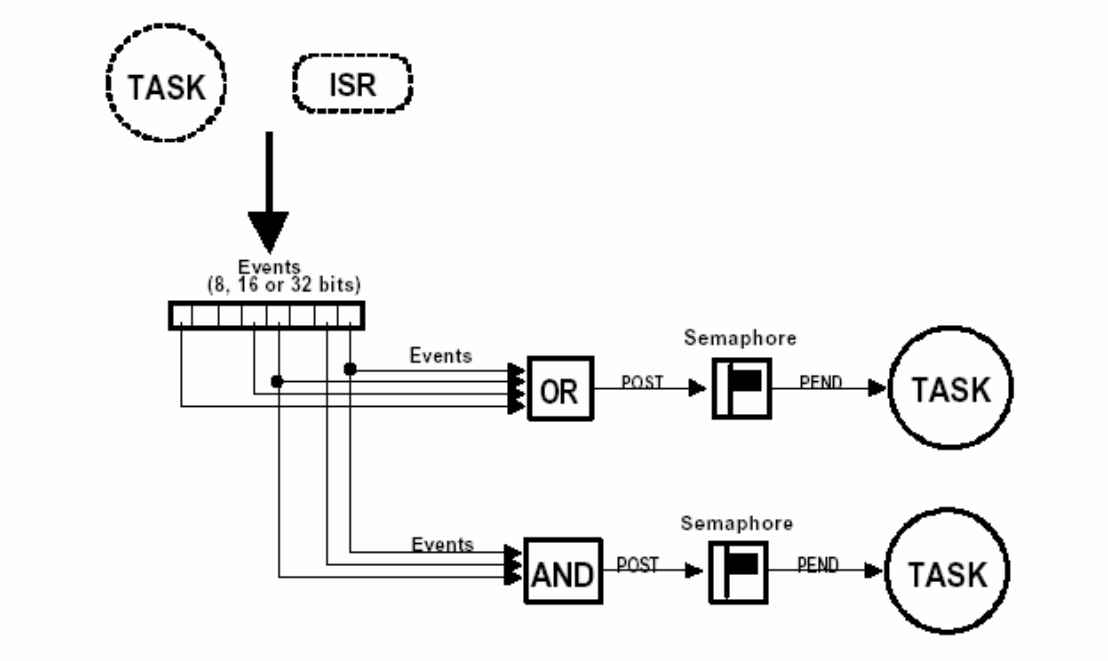

图 2.16 事件标志

2.23 消息邮箱(Message Mail boxes)

通过内核服务可以给任务发送消息。典型的消息邮箱也称作交换消息，是用一个指针型变量，通过内核服务，一个任务或一个中断服务程序可以把一则消息(即一个指针)放到邮箱里去。同样，一个或多个任务可以通过内核服务接收这则消息。发送消息的任务和接收消息的任务约定，该指针指向的内容就是那则消息。

每个邮箱有相应的正在等待消息的任务列表，要得到消息的任务会因为邮箱是空的而被挂起，且被记录到等待消息的任务表中，直到收到消息。一般地说，内核允许用户定义等待超时，等待消息的时间超过了，仍然没有收到该消息，这任务进入就绪态，并返回出错信息，报告等待超时错误。消息放入邮箱后，或者是把消息传给等待消息的任务表中优先级最高的那个任务(基于优先级)，或者是将消息传给最先开始等待消息的任务(基于先进先出)。图 2.17 示意把消息放入邮箱。用一个 I 字表示邮箱，旁边的小砂漏表示超时计时器，计时器旁边的数字表示定时器设定值，即任务最长可以等多少个时钟节拍(Clock Ticks)，关于时钟节拍以后会讲到。

内核一般提供以下邮箱服务：

- 邮箱内消息的内容初始化，邮箱里最初可以有，也可以没有消息
- 将消息放入邮箱(POST)
- 等待有消息进入邮箱(PEND)

- 如果邮箱内有消息，就接受这则消息。如果邮箱里没有消息，则任务并不被挂起(ACCEPT)，用返回代码表示调用结果，是收到了消息还是没有收到消息。

消息邮箱也可以当作只取两个值的信号量来用。邮箱里有消息，表示资源可以使用，而空邮箱表示资源已被其它任务占用。

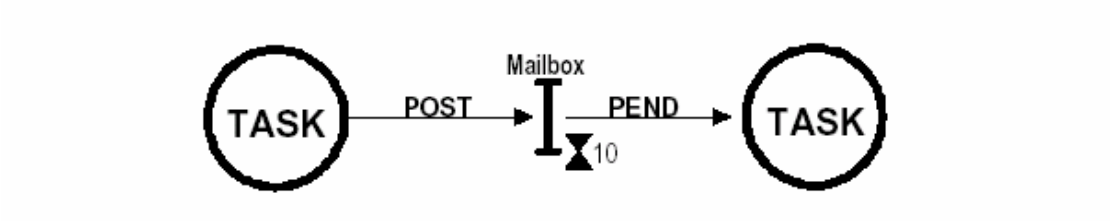

图 2.17 消息邮箱

2.24 消息队列(Message Queue)

消息队列用于给任务发消息。消息队列实际上是邮箱阵列。通过内核提供的服务，任务或中断服务子程序可以将一条消息(该消息的指针)放入消息队列。同样，一个或多个任务可以通过内核服务从消息队列中得到消息。发送和接收消息的任务约定，传递的消息实际上是传递的指针指向的内容。通常，先进入消息队列的消息先传给任务，也就是说，任务先得到的是最先进入消息队列的消息，即先进先出原则(FIFO)。然而μC/OS-II也允许使用后进先出方式(LIFO)。

像使用邮箱那样，当一个以上的任务要从消息队列接收消息时，每个消息队列有一张等待消息任务的等待列表(Waiting List)。如果消息队列中没有消息，即消息队列是空，等待消息的任务就被挂起并放入等待消息任务列表中，直到有消息到来。通常，内核允许等待消息的任务定义等待超时的时间。如果限定时间内任务没有收到消息，该任务就进入就绪态并开始运行，同时返回出错代码，指出出现等待超时错误。一旦一则消息放入消息队列，该消息将传给等待消息的任务中优先级最高的那个任务，或是最先进入等待消息任务列表的任务。图 2.18 示意中断服务子程序如何将消息放入消息队列。图中两个大写的 I 表示消息队列，“10”表示消息队列最多可以放 10 条消息，沙漏旁边的 0 表示任务没有定义超时，将永远等下去，直至消息的到来。

典型地，内核提供的消息队列服务如下：

- 消息队列初始化。队列初始化时总是清为空。
- 放一则消息到队列中去(Post)
- 等待一则消息的到来(Pend)
- 如果队列中有消息则任务可以得到消息，但如果此时队列为空，内核并不将该任务挂起(Accept)。如果有消息，则消息从队列中取走。没有消息则用特别的返回代码

通知调用者，队列中没有消息。

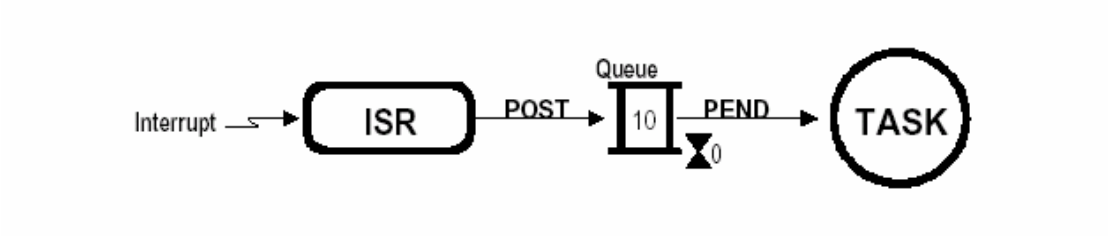

图 2.18 消息队列

2.25 中断

中断是一种硬件机制，用于通知 CPU 有个异步事件发生了。中断一旦被识别，CPU 保存部分(或全部)现场(Context)即部分或全部寄存器的值，跳转到专门的子程序，称为中断服务子程序(ISR)。中断服务子程序做事件处理，处理完成后，程序回到：

- 在前后台系统中，程序回到后台程序
- 对不可剥夺型内核而言，程序回到被中断了的任务
- 对可剥夺型内核而言，让进入就绪态的优先级最高的任务开始运行

中断使得 CPU 可以在事件发生时才予以处理，而不必让微处理器连续不断地查询(Polling)是否有事件发生。通过两条特殊指令：关中断(Disable interrupt)和开中断(Enable interrupt)可以让微处理器不响应或响应中断。在实时环境中，关中断的时间应尽量的短。关中断影响中断延迟时间(见 2.26 中断延迟)。关中断时间太长可能会引起中断丢失。微处理器一般允许中断嵌套，也就是说在中断服务期间，微处理器可以识别另一个更重要的中断，并服务于那个更重要的中断，如图 2.19 所示。

2.26 中断延迟

可能实时内核最重要的指标就是中断关了多长时间。所有实时系统在进入临界区代码段之前都要关中断，执行完临界代码之后再开中断。关中断的时间越长，中断延迟就越长。中断延迟由表达式[2.2]给出。

[2.2] 中断延迟 = 关中断的最长时间 + 开始执行中断服务子程序的第一条指令的时间

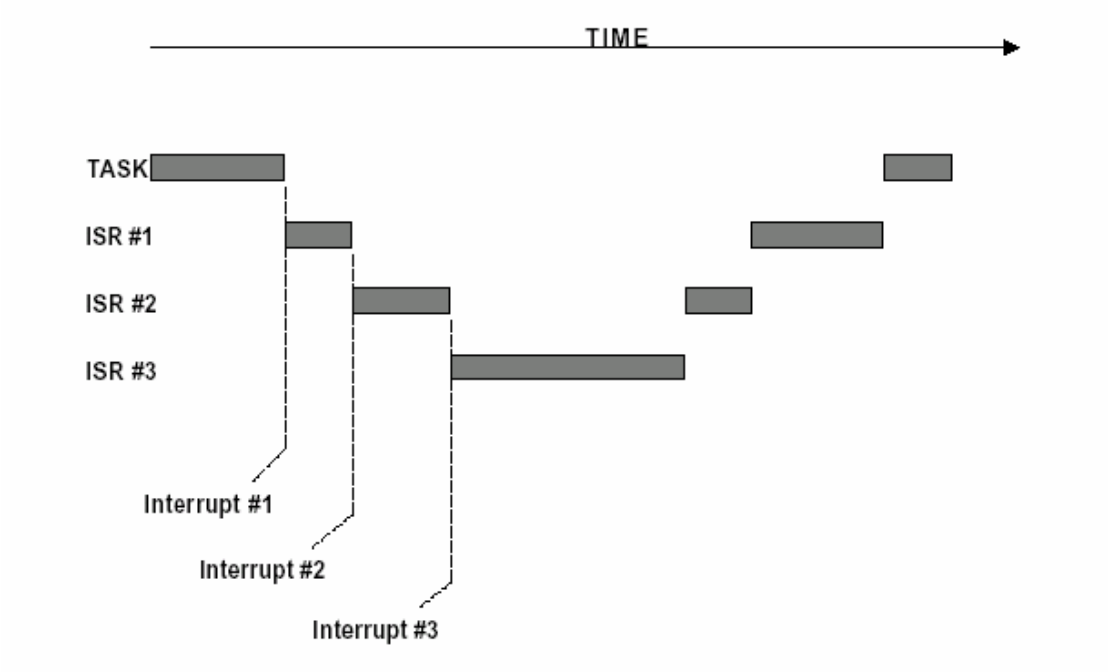

图 2.19 中断嵌套

2.27 中断响应

中断响应定义为从中断发生到开始执行用户的中断服务子程序代码来处理这个中断的时间。中断响应时间包括开始处理这个中断前的全部开销。典型地，执行用户代码之前要保护现场，将 CPU 的各寄存器推入堆栈。这段时间将被记作中断响应时间。

对前后台系统，保存寄存器以后立即执行用户代码，中断响应时间由[2.3]给出。

[2.3] 中断响应时间 = 中断延迟 + 保存 CPU 内部寄存器的时间

对于不可剥夺型内核，微处理器保存内部寄存器以后，用户的中断服务子程序代码全立即得到执行。不可剥夺型内核的中断响应时间由表达式[2.4]给出。

[2.4] 中断响应时间 = 中断延迟 + 保存 CPU 内部寄存器的时间

对于可剥夺型内核，则要先调用一个特定的函数，该函数通知内核即将进行中断服务，使得内核可以跟踪中断的嵌套。对于 μC/OS-Ⅱ说来，这个函数是 OSIntEnter()，可剥夺型内核的中断响应时间由表达式[2.5]给出：

[2.5] 中断响应 = 中断延迟 + 保存 CPU 内部寄存器的时间 + 内核的进入中断服务

函数的执行时间

中断响应是系统在最坏情况下的响应中断的时间，某系统 100 次中有 99 次在 50μs 之内响应中断，只有一次响应中断的时间是 250μs，只能认为中断响应时间是 250μs。

2.28 中断恢复时间(Interrupt Recovery)

中断恢复时间定义为微处理器返回到被中断了的程序代码所需要的时间。在前后台系统中，中断恢复时间很简单，只包括恢复 CPU 内部寄存器值的时间和执行中断返回指令的时间。中断恢复时间由[2.6]式给出。

[2.6] 中断恢复时间 = 恢复 CPU 内部寄存器值的时间 + 执行中断返回指令的时间

和前后台系统一样，不可剥夺型内核的中断恢复时间也很简单，只包括恢复 CPU 内部寄存器值的时间和执行中断返回指令的时间，如表达式[2.7]所示。

[2.7] 中断恢复时间 = 恢复 CPU 内部寄存器值的时间 + 执行中断返回指令的时间

对于可剥夺型内核，中断的恢复要复杂一些。典型地，在中断服务子程序的末尾，要调用一个由实时内核提供的函数。在μC/OS-II 中，这个函数叫做 OSIntExit()，这个函数用于辨定中断是否脱离了所有的中断嵌套。如果脱离了嵌套(即已经可以返回到被中断了的任务级时)，内核要辨定，由于中断服务子程序 ISR 的执行，是否使得一个优先级更高的任务进入了就绪态。如果是，则要让这个优先级更高的任务开始运行。在这种情况下，被中断了的任务只有重新成为优先级最高的任务而进入就绪态时才能继续运行。对于可剥夺型内核，中断恢复时间由表达式[2.8]给出。

[2.8] 中断恢复时间 = 判定是否有优先级更高的任务进入了就绪态的时间 + 恢复那个优先级更高任务的 CPU 内部寄存器的时间 + 执行中断返回指令的时间

2.29 中断延迟、响应和恢复

图 2.20 到图 2.22 示意前后台系统、不可剥夺性内核、可剥夺性内核相应的中断延迟、响应和恢复过程。

注意，对于可剥夺型实时内核，中断返回函数将决定是返回到被中断的任务[图 2.22A]，还是让那个优先级最高任务运行。是中断服务子程序使那个优先级更高的任务进入了就绪态[图 2.22B]。在后一种情况下，恢复中断的时间要稍长一些，因为内核要做任务切换。在本书中，我做了一张执行时间表，此表多少可以衡量执行时间的不同，假定μC/OS-II 是在

33MHZ Intel 80186 微处理器上运行的。此表可以使读者看到做任务切换的时间开销。(见表 9.3，在 33MHZ 80186 上μC/OS-Ⅱ服务的执行时间).

2.30 中断处理时间

虽然中断服务的处理时间应该尽可能的短，但是对处理时间并没有绝对的限制。不能说中断服务必须全部小于 100μS，500μS 或 1mS。如果中断服务是在任何给定的时间开始，且中断服务程序代码是应用程序中最重要的代码，则中断服务需要多长时间就应该给它多长时间。然而在大多数情况下，中断服务子程序应识别中断来源，从叫中断的设备取得数据或状态，并通知真正做该事件处理的那个任务。当然应该考虑到是否通知一个任务去做事件处理所花的时间比处理这个事件所花的时间还多。在中断服务中通知一个任务做时间处理(通过信号量、邮箱或消息队列)是需要一定时间的，如果事件处理需花的时间短于给一个任务发通知的时间，就应该考虑在中断服务子程序中做事件处理并在中断服务子程序中开中断，以允许优先级更高的中断打入并优先得到服务。

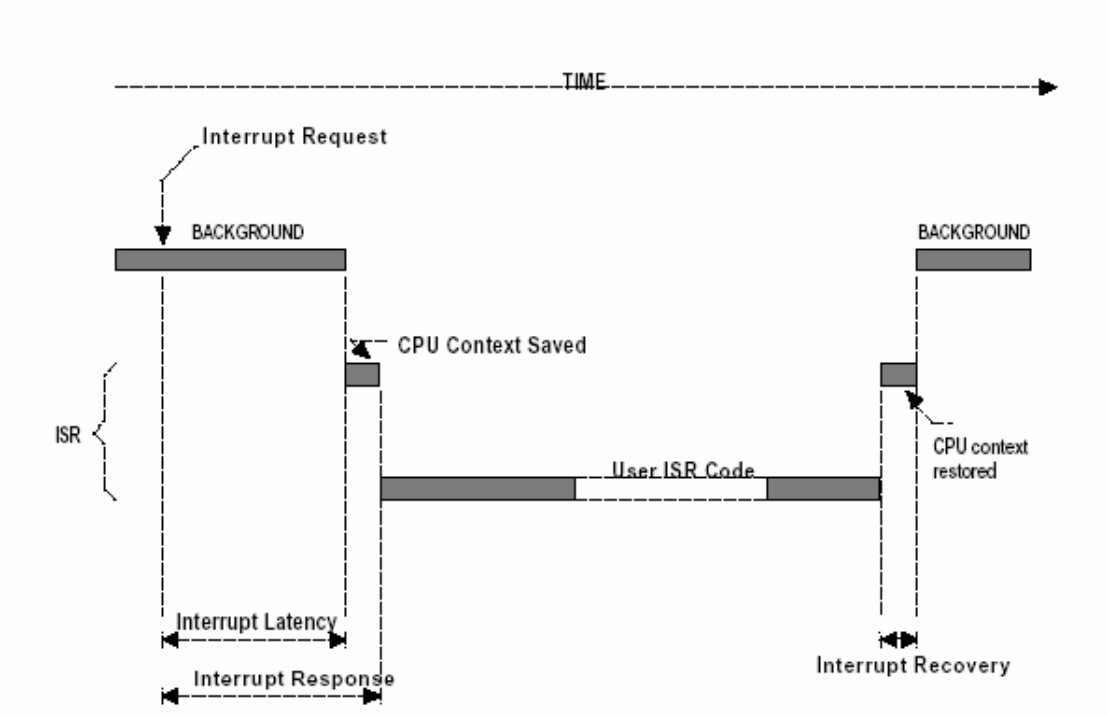

图 2.20 中断延迟、响应和恢复(前后台模式)

2.31 非屏蔽中断(NMI)

有时，中断服务必须来得尽可能地快，内核引起的延时变得不可忍受。在这种情况下可以使用非屏蔽中断，绝大多数微处理器有非屏蔽中断功能。通常非屏蔽中断留做紧急处理用，如断电时保存重要的信息。然而，如果应用程序没有这方面的要求，非屏蔽中断可用于时间要求最苛刻的中断服务。下列表达式给出如何确定中断延迟、中断响应时间和中断恢复时间。

[2.9] 中断延迟时间 = 指令执行时间中最长的那个时间 + 开始做非屏蔽中断服务的时间

[2.10] 中断响应时间 = 中断延迟时间 + 保存 CPU 寄存器花的时间

[2.11] 中断恢复时间 = 恢复 CPU 寄存器的时间 + 执行中断返回指令的时间。

在一项应用中，我将非屏蔽中断用于可能每 150μS 发生一次的中断。中断处理时间在 80 至 125μS 之间。所使用的内核的关中断时间是 45μS。可以看出，如果使用可屏蔽中断的话，中断响应会推迟 20μS。

在非屏蔽中断的中断服务子程序中，不能使用内核提供的服务，因为非屏蔽中断是关不掉的，故不能在非屏蔽中断处理中处理临界区代码。然而向非屏蔽中断传送参数或从非屏蔽中断获取参数还是可以进行的。参数的传递必须使用全程变量，全程变量的位数必须是一次读或写能完成的，即不应该是两个分离的字节，要两次读或写才能完成。

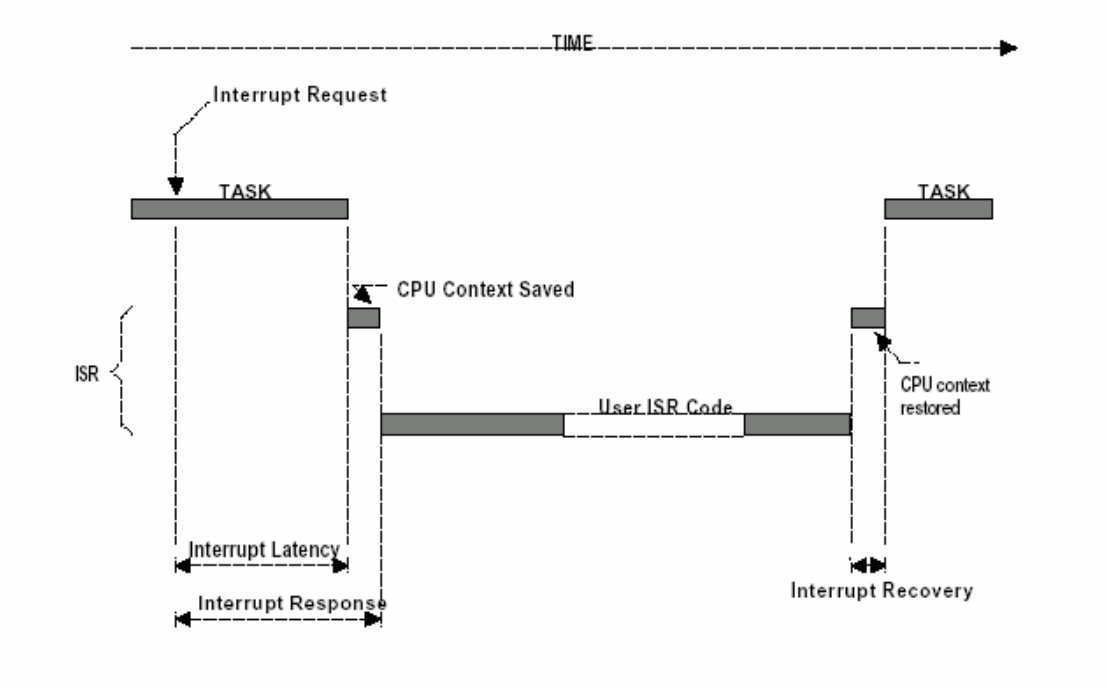

图 2.21 中断延迟、响应和恢复(不可剥夺型内核)

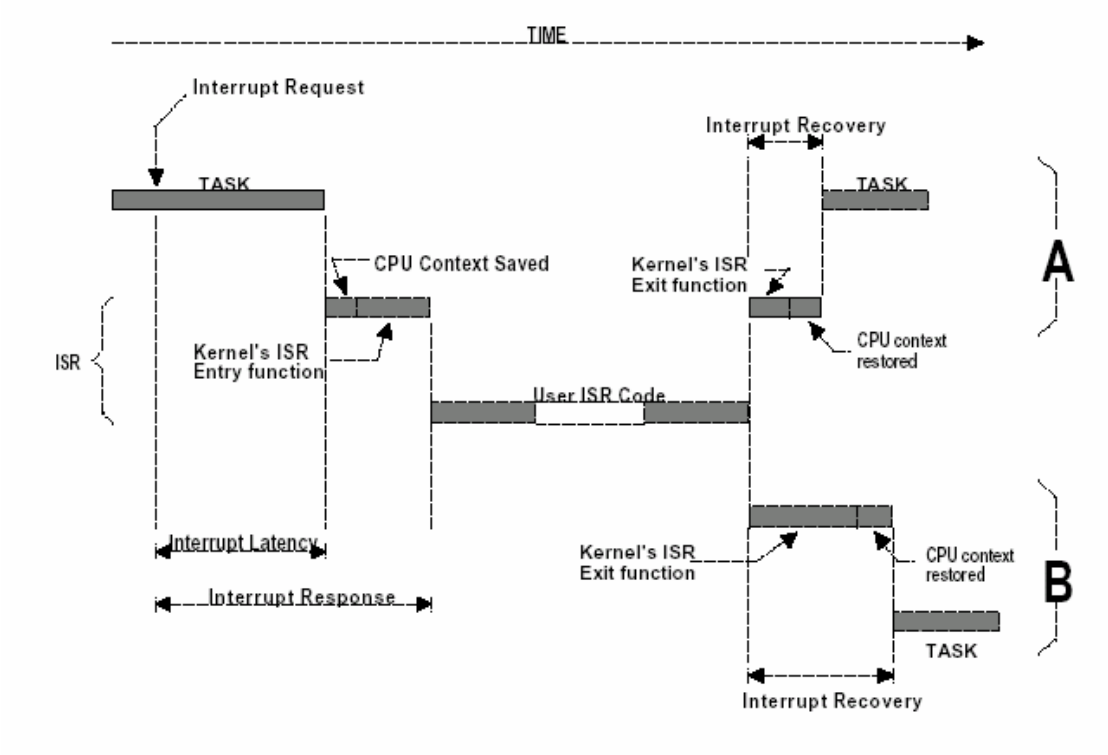

图 2.22 中断延迟、响应和恢复(可剥夺型内核)

非屏蔽中断可以用增加外部电路的方法禁止掉，如图 2.23 所示。假定中断源和非屏蔽中断都是正逻辑，用一个简单的“与”门插在中断源和微处理器的非屏蔽中断输入端之间。向输出口(Output Port)写 0 就将中断关了。不一定要以这种关中断方式来使用内核服务，但可以用这种方式在中断服务子程序和任务之间传递参数(大的、多字节的，一次读写不能完成的变量)。

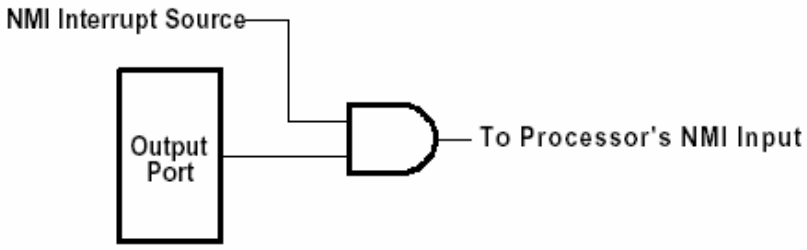

图 2.23 非屏蔽中断的禁止

假定非屏蔽中断服务子程序每 40 次执行中有一次要给任务发信号，如果非屏蔽中断 150

μS 执行一次，则每 6mS(40*150μS)给任务发一次信号。在非屏蔽中断服务子程序中，不能使用内核服务给任务发信号，但可以使用如图 2.24 所示的中断机制。即用非屏蔽中断产生普通可屏蔽中断的机制。在这种情况下，非屏蔽中断通过某一输出口产生硬件中断(置输出口为有效电平)。由于非屏蔽中断服务通常具有最高的优先级，在非屏蔽中断服务过程中不允许中断嵌套，普通中断一直要等到非屏蔽中断服务子程序运行结束后才能被识别。在非屏蔽中断服务子程序完成以后，微处理器开始响应这个硬件中断。在这个中断服务子程序中，要清除中断源(置输出口为无效电平)，然后用信号量去唤醒那个需要唤醒的任务。任务本身的运行时间和信号量的有效时间都接近 6mS，实时性得到了满足。

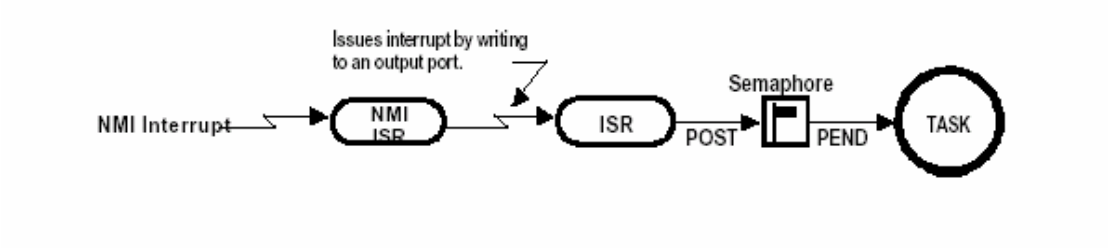

图 2.24 非屏蔽中断产生普通可屏蔽中断

2.32 时钟节拍(Clock Tick)

时钟节拍是特定的周期性中断。这个中断可以看作是系统心脏的脉动。中断之间的时间间隔取决于不同的应用，一般在 10mS 到 200mS 之间。时钟的节拍式中断使得内核可以将任务延时若干个整数时钟节拍，以及当任务等待事件发生时，提供等待超时的依据。时钟节拍率越快，系统的额外开销就越大。

各种实时内核都有将任务延时若干个时钟节拍的功能。然而这并不意味着延时的精度是 1 个时钟节拍，只是在每个时钟节拍中断到来时对任务延时做一次裁决而已。

图 2.25 到 图 2.27 示意任务将自身延迟一个时钟节拍的时序。阴影部分是各部分程序的执行时间。请注意，相应的程序运行时间是长短不一的，这反映了程序中含有循环和条件转移语句(即 if/else, switch, ? : 等语句)的典型情况。时间节拍中断服务子程序的运行时间也是不一样的。尽管在图中画得有所夸大。

第一种情况如图 2.25 所示，优先级高的任务和中断服务超前于要求延时一个时钟节拍的任务运行。可以看出，虽然该任务想要延时 20mS，但由于其优先级的缘故，实际上每次延时多少是变化的，这就引起了任务执行时间的抖动。

第二种情况，如图 2.26 所示，所有高优先级的任务和中断服务的执行时间略微小于一个时钟节拍。如果任务将自己延时一个时钟节拍的请求刚好发生在下一个时钟节拍之前，这个任务的再次执行几乎是立即开始的。因此，如果要求任务的延迟至少为一个时钟节拍的话，则要多定义一个延时时钟节拍。换句话说，如果想要将一个任务至少延迟 5 个时钟节拍的话，得在程序中延时 6 个时钟节拍。

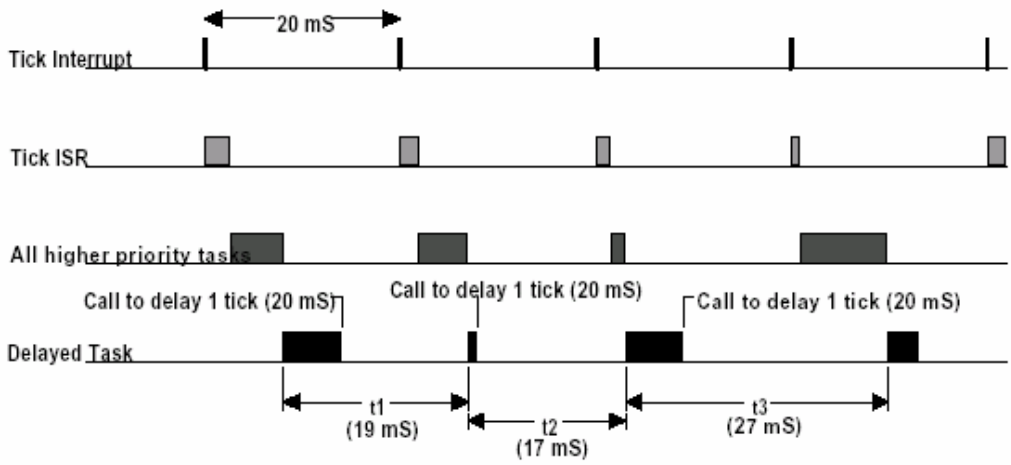

图 2.25 将任务延迟一个时钟节拍(第一种情况)

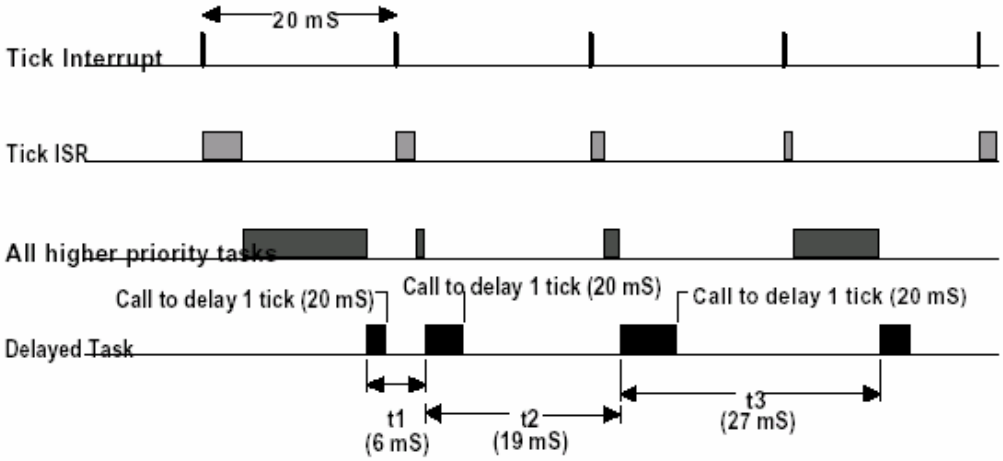

图 2.26 将任务延迟一个时钟节拍(第二种情况)

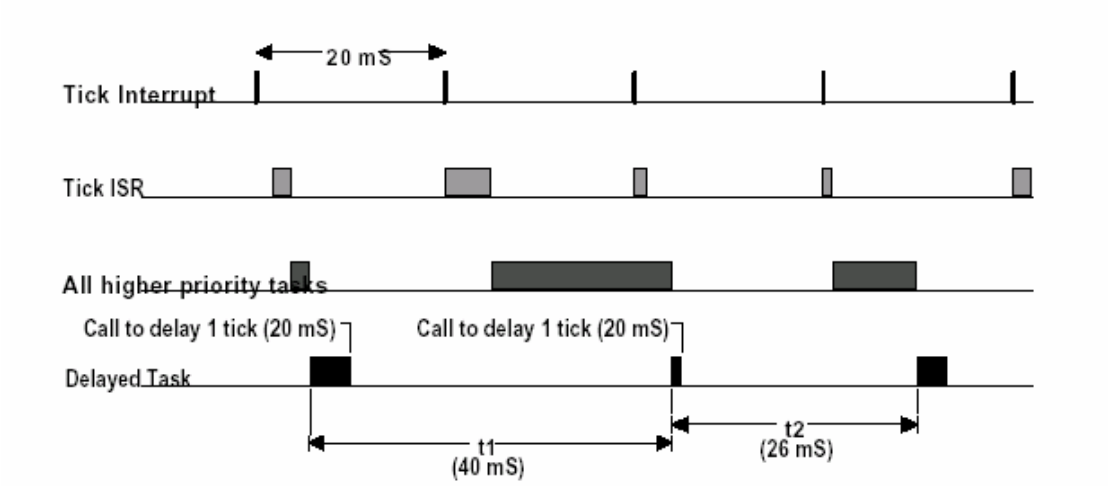

图 2.27 将任务延迟一个时钟节拍(第三种情况)

第三种情况，如图 2.27 所示，所有高优先级的任务加上中断服务的执行时间长于一个时钟节拍。在这种情况下，拟延迟一个时钟节拍的任务实际上在两个时钟节拍后开始运行，引起了延迟时间超差。这在某些应用中或许是可以的，而在多数情况下是不可接受的。

上述情况在所有的实时内核中都会出现，这与 CPU 负荷有关，也可能与系统设计不正确有关。以下是这类问题可能的解决方案：

- 增加微处理器的时钟频率
- 增加时钟节拍的频率
- 重新安排任务的优先级
- 避免使用浮点运算(如果非使用不可，尽量用单精度数)
- 使用能较好地优化程序代码的编译器
- 时间要求苛刻的代码用汇编语言写
- 如果可能，用同一家族的更快的微处理器做系统升级。如从 8086 向 80186 升级，从 68000 向 68020 升级等
-

不管怎么样，抖动总是存在的。

2.33 对存储器的需求

如果设计是前后台系统，对存储器容量的需求仅仅取决于应用程序代码。而使用多任务内核时的情况则很不一样。内核本身需要额外的代码空间(ROM)。内核的大小取决于多种因素，取决于内核的特性，从 1K 到 100K 字节都是可能的。8 位 CPU 用的最小内核只提供任务调度、任务切换、信号量处理、延时及超时服务约需要 1K 到 3K 代码空间。代码空间总需求量由表达式[2.12]给出。

[2.12] 总代码量 = 应用程序代码 + 内核代码

因为每个任务都是独立运行的，必须给每个任务提供单独的栈空间(RAM)。应用程序设计人员决定分配给每个任务多少栈空间时，应该尽可能使之接近实际需求量 (有时，这是相当困难的一件事)。栈空间的大小不仅仅要计算任务本身的需求 (局部变量、函数调用等等)，还需要计算最多中断嵌套层数(保存寄存器、中断服务程序中的局部变量等)。根据不同的目标微处理器和内核的类型，任务栈和系统栈可以是分开的。系统栈专门用于处理中断级代码。这样做有许多好处，每个任务需要的栈空间可以大大减少。内核的另一个应该具有的性能是，每个任务所需的栈空间大小可以分别定义(μC/OS－II 可以做到)。相反，有些内核要求每个任务所需的栈空间都相同。所有内核都需要额外的栈空间以保证内部变量、数据结构、队列等。如果内核不支持单独的中断用栈，总的 RAM 需求由表达式[2.13]给出。

[2.13] RAM 总需求 = 应用程序的 RAM 需求 + (任务栈需求 + 最大中断嵌套栈需求) * 任务数

如果内核支持中断用栈分离，总 RAM 需求量由表达式[2.14]给出

[2.14]RAM 总需求 = 应用程序的 RAM 需求 + 内核数据区的 RAM 需求 + 各任务栈需求之总和 + 最多中断嵌套之栈需求

除非有特别大的 RAM 空间可以所用，对栈空间的分配与使用要非常小心。为减少应用程序需要的 RAM 空间，对每个任务栈空间的使用都要非常小心，特别要注意以下几点：

- 定义函数和中断服务子程序中的局部变量，特别是定义大型数组和数据结构
- 函数(即子程序)的嵌套
- 中断嵌套
- 库函数需要的栈空间
- 多变元的函数调用

综上所述，多任务系统比前后台系统需要更多的代码空间(ROM)和数据空间(RAM)。额外的代码空间取决于内核的大小，而 RAM 的用量取决于系统中的任务数。

2.34 使用实时内核的优缺点

实时内核也称为实时操作系统或 RTOS。它的使用使得实时应用程序的设计和扩展变得容易，不需要大的改动就可以增加新的功能。通过将应用程序分割成若干独立的任务，RTOS 使得应用程序的设计过程大为减化。使用可剥夺性内核时，所有时间要求苛刻的事件都得到

了尽可能快捷、有效的处理。通过有效的服务，如信号量、邮箱、队列、延时、超时等，RTOS 使得资源得到更好的利用。

如果应用项目对额外的需求可以承受，应该考虑使用实时内核。这些额外的需求是：内核的价格，额外的 ROM/RAM 开销，2 到 4 百分点的 CPU 额外负荷。

还没有提到的一个因素是使用实时内核增加的价格成本。在一些应用中，价格就是一切，以至于对使用 RTOS 连想都不敢想。

当今有 80 个以上的 RTOS 商家，生产面向 8 位、16 位、32 位、甚至是 64 位的微处理器的 RTOS 产品。一些软件包是完整的操作系统，不仅包括实时内核，还包括输入输出管理、视窗系统(用于显示)、文件系统、网络、语言接口库、调试软件、交叉平台编译(Cross-Platform compilers)。RTOS 的价格从 70 美元到 30,000 美元。RTOS 制造商还可能索取每个目标系统的版权使用费。就像从 RTOS 商家那买一个芯片安装到每一个产品上，然后一同出售。RTOS 商家称之为硅片软件(Silicon Software)。每个产品的版权费从 5 美元到 250 美元不等。同如今的其它软件包一样，还得考虑软件维护费，这部分开销为每年还得花 100 到 5,000 美元！

2.35 实时系统小结

三种类型的实时系统归纳于表 2.2 中，这三种实时系统是：前后台系统，不可剥夺型内核和可剥夺型内核。

. 表 2.2 实时系统小结

	Foreground/ Background	***Non-Preemptive Kernel***	***Preemptive Kernel***
Interrupt latency (Time)	MAX(Longest instruction, User int. disable) + Vector to ISR	MAX(Longest instruction, User int. disable, Kernel int. disable) + Vector to ISR	MAX(Longest instruction, User int. disable, Kernel int. disable) + Vector to ISR
Interrupt response (Time)	Int. latency + Save CPU's context	Int. latency + Save CPU's context	Interrupt latency + Save CPU's context + Kernel ISR entry function
Interrupt recovery (Time)	Restore background's context + Return from int.	Restore task's context + Return from int.	Find highest priority task + Restore highest priority task's context + Return from interrupt
Task response (Time)	Background	Longest task + Find highest priority task	Find highest priority task + Context switch

+ Context switch

ROM size	Application code	Application code + Kernel code	Application code + Kernel code
RAM size	Application code	Application code + Kernel RAM + SUM(Task stacks + MAX(ISR stack))	Application code + Kernel RAM + SUM(Task stacks + MAX(ISR stack))
Services available ?	Application code must provide	Yes	Yes