第 3 章

使用互斥机制控制资源共享

本章目标

- 解释在多任务设计中使用共享资源时所面临的问题。
- 描述什么是互斥。
- 展示如何使用程序标志实现互斥。
- 讲解二值信号量和计数信号量的概念及其使用。
- 描述互斥量并展示如何使用互斥量改进信号量性能。
- 解析信号量和互斥量的缺陷。
- 展示通过使用简单的监视器结构来克服信号量和互斥量的弱点。

3.1 共享资源使用中的问题

在单 CPU 系统中,处理器是一个共享资源。在多个进程之间共享处理器时,处理器的使用由调度程序控制,不存在竞争问题。但对于系统的其他资源而言,情况并非如此。不同的任务可能需要同时使用同一硬件外设或内存区域。如果不控制这些公共资源的访问,系统中很快就会出现资源争用问题。例如,在图 3.1 所示场景中,控制算法由中断驱动的定时

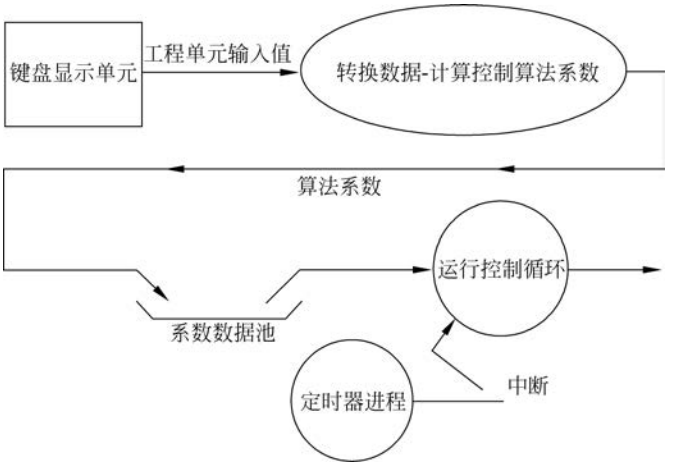

图 3.1　进程间共享数据

器进程以恒定的间隔执行，此应用场景中系统可能发生什么情况？

运行控制循环进程使用的部分数据来自系数的共享读写数据库（内存数据池）。系数值源自工程单元的键盘显示输入。现在面临的问题很简单，如果系数更新过程中，系统激活了运行控制循环，会是什么结果？每个系数最长占用 8 字节，但一次修改过程可能仅改变 1 或 2 字节。因此，当发生任务切换时，系数值可能仅部分被更新。如果发生这种情况，可能导致灾难性的结果。

如何解决这个问题？解决方案非常简单，确保一个共享资源在任何时候只能被一个进程访问即可，即实施一种互斥策略，但困难的是如何制定具体的方法。

3.2 使用单个标志实现互斥

为了控制对共享（或“公共”）对象的访问，可以假设将它放在一个特定的房间中，见图 3.2。

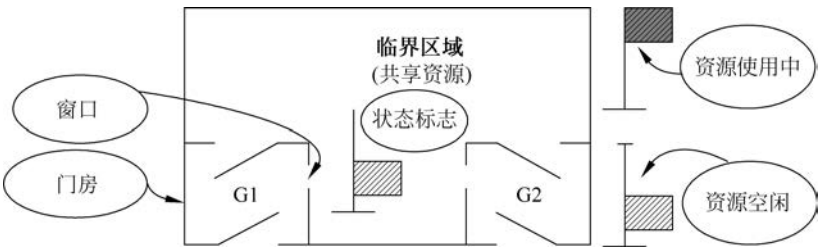

图 3.2 单个标志方法

为需要使用资源的每个任务提供一个门房，作为其使用资源的手段。通过标志指示器指示任务是否在房间内（临界区域）。每个任务从门房内只能看到标志升起或降下。

假设最初临界区域为空，标志被降下。希望使用该资源的用户 1（即任务 1）进入相应的门房，它首先检查标志的位置，发现标志为降下状态（表示资源空闲），它将升起标志，进入临界区。此时用户 2（即任务 2）到达现场，也想访问共享对象，见图 3.3，它进入其门房并检查标志状态。由于标志处于升起状态（资源正在使用），故用户 2 将在此等待，不断检查标志状态。最终用户 1 完成工作离开，它的最后一份工作是降下标志，表示资源被释放。因此，当用户 2 再次检查资源状态时，发现资源可用，于是它升起标志并进入临界区域，成为共享资源的唯一拥有者。

整个工作流程看起来非常顺畅，通过相当简单的机制成功实现了互斥。事实确实如此吗？考虑以下应用场景。当用户 1 进入门房 1 时，资源为空闲状态，它首先检查标志，发现其为降下状态，然后任务将升起标志。此时，用户 2 也进入其门房并检查标志，发现标志为降下状态。所以，在它看来，自己也可以进入临界区。它并没有意识到，用户 1 也在做同样的事情，从而发生了冲突。

由于用户 1 在检查标志状态并改变其状态的过程中存在时间差，导致保护机制失效。在计算机语言中，对于单处理器系统，该操作等价于：

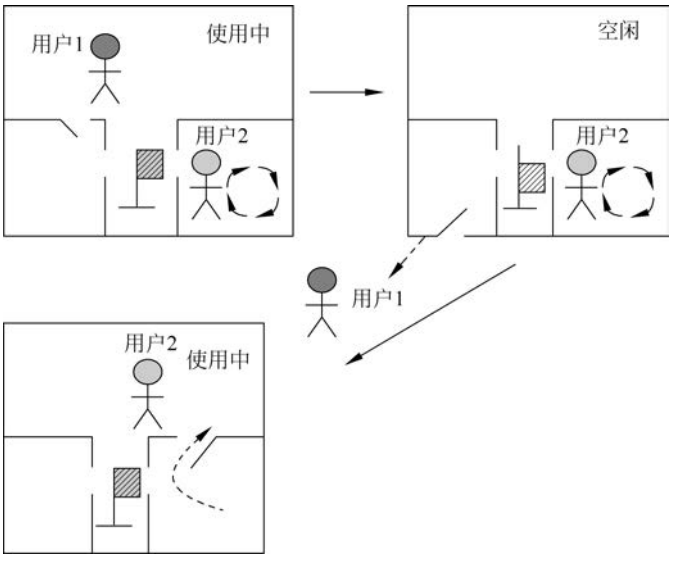

图 3.3　互斥实现(单标志方法)

(1) 将内存中的状态变量加载到处理器寄存器中。

(2) 检查变量的状态。

(3) 如果变量为“空闲”,将其值更改为“使用中”,并且将新的状态值写入内存;否则重新执行检查。

在此序列中,用户 1(即任务 1)可能被抢占。如果抢占发生在变量加载后但其状态改变之前,可能会遇到冲突问题。假设抢占任务的为用户 2,它检查标志状态,发现资源可用,用户 2 进入临界区。当任务 1 恢复运行时,它也认为资源可用,也要进入临界区,所以互斥机制失效。

幸运的是,大多数现代微处理器能在单条指令中完成位/字节的设置和测试工作。这意味着检查和设置操作是不可分割的原子操作,从而保证了操作的安全性。然而,在多处理器系统中情况则不同,因为这些操作具有真正的并发性,这部分内容稍后讨论。

单个标志技术易于实现,使用简单,通过适当的设计可以安全地工作,但它的效率不高。如上所述,当发现标志被置位后,任务会进入检查循环。任务不能改变标志状态,它将在其执行时间片期间保持“忙等待”模式,导致处理器时间浪费和处理器性能降低(即利用率降低)。在实时系统(尤其是硬实时)中,这种低效率方式是不可接受的,需要采用另一种互斥技术。

很明显,当任务发现它的动作被阻塞时,它需要放弃处理器,即任务挂起,从而允许另一个任务使用处理器。这种方法称为“挂起-等待”,可以通过多种方式实现。在某些情况下可以使用信号,在程序控制下完成。然而更好的技术是使用专门设计的结构,用于支持挂起-等待操作,如信号量、互斥信号量和监视器。

注意:在多核/多处理器中以本节描述的方式使用标志时,标志也称为自旋锁。

3.3 信号量

3.3.1 二值信号量

信号量本质上是一个程序数据项，用于决定任务继续运行还是挂起。信号量类型有两种，二值信号量和通用型/计数信号量。两者的工作原理相同，信号量原语最初由 Edsger Dijkstra 在 1965 年提出。

首先分析二值信号量。本质上，二值信号量是一个任务流控制机制，可以将其比作铁路信号，见图 3.4。火车将根据信号位置情况，决定通过该点还是必须停止。如果火车停下来，它们会保持在该位置直到信号变为“允许通行”。以类似的方式，信号量可以允许任务继续执行其代码或挂起。一旦任务挂起，它将保持在此状态，直到某些程序操作使该任务重新就绪。

图 3.4 信号量类比——铁路信号
（来源：Simon Howden，FreeDigitalPhotos.net）

在铁路系统中，信号是一种安全机制，用于控制列车运行，从而防止碰撞、损坏或人身事故。实际的铁路网络中还有很多信号，可根据需要使用这些信号。同样，多任务设计中也可能使用许多信号量，每个信号量都相当于一个特定的信号。

在并发软件中，信号量用于两个截然不同的目的。本节描述的功能是作为互斥（消除争用）机制，每个共享资源分配一个信号量。信号量也用于同步，实现任务交互，该功能将在 5.2 节描述。

用于访问控制时信号量的概念见图 3.5。该类比是停车场入口控制，信号量相当于控制机制。这里的共享资源是一个单独的停车位，用于上下客。需要确保有且只有一辆车可以进入停车位。因此，用户在尝试停车之前，必须首先检查停车位是否空闲。为此使用了一

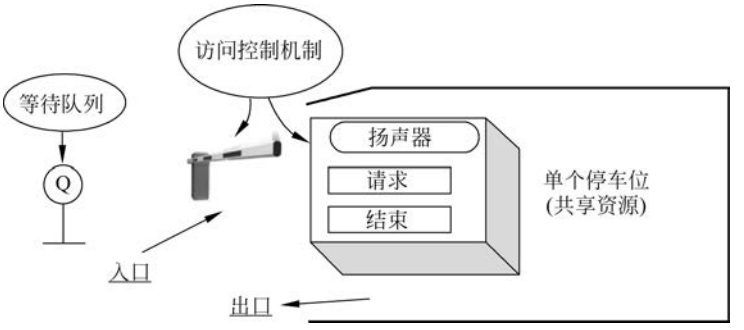

图 3.5 二值信号量用于互斥—概念

个访问控制接口，其包括：

（1）“请求”按钮。用户按下此按钮以向停车场服务员发出需要进入停车场的信号。

（2）“结束”按钮。用户在退出时，按此按钮通知服务员，停车位再次空闲。

（3）扬声器。停车场服务员用它来回答用户的请求。

在此类比中，二进制信号量等价于访问控制机制的软件实现，这里用户指的是任务。

假设最初停车场资源处于空闲状态。第一个操作是向停车场服务员提供此状态信息（假设从控制室看不到停车位）。其对应的软件操作将初始化该信号量。同样，服务员功能由操作系统软件提供。

当用户需要使用停车资源时，它靠近屏障并按下请求按钮，在信号量术语中，该行为被定义为信号等待（wait）操作。由于资源处于空闲状态，故服务员抬起屏障并回答可以通过，用户随即进入保护区域，然后屏障关闭。

某个时刻用户离开并腾出停车位，在退出时按下结束按钮，发出信号给访问控制机制，该行为被定义为信号发布（signal）操作。操作结果将更新服务员看到的状态信息，显示停车场再次空闲。

现在考虑另一种情况，当另一辆车到达时资源正在使用，请求服务的用户被放置到等待队列（对应于任务挂起）。当前资源占用者，一旦完成工作，在离开停车位时将生成一个信号，控制机制收到该信号，标记资源空闲。但随后的事件遵循了不同的模式，没有更新服务员的状态信息。取而代之的是抬起屏障并发送“通过”消息给等待的用户（相当于一个任务唤醒另一个任务），授权用户进入保护区域；后续的事件处理过程与前面一致。

需要考虑的一个重点是，当任务 1 已经处于等待队列中时，一个更高优先级的任务（比如任务 3）到达，系统如何处理。实际上，结果取决于使用的排队策略。通常队列使用两种排队策略，先进先出（FIFO）策略和优先级抢占策略。

使用先进先出策略排队时，任务 3 排在任务 1 的后面。因此条件允许（即空间可用）时，任务 1 可以立即执行。此方式虽然安全，但导致较低优先级的任务延迟了较高优先级任务的执行，可能导致严重的系统问题（优先级翻转，见 4.4 节）。

如果使用优先级抢占策略，任务 3 优先并排在队列前面，因此它将第一个就绪。但该方式延迟了任务 1 的执行，也会导致潜在的性能问题（任务饥饿）。

由设计者来决定使用哪种方法以及在何处使用它，但无论哪种情况，任务行为建模都如图 3.6 所示。

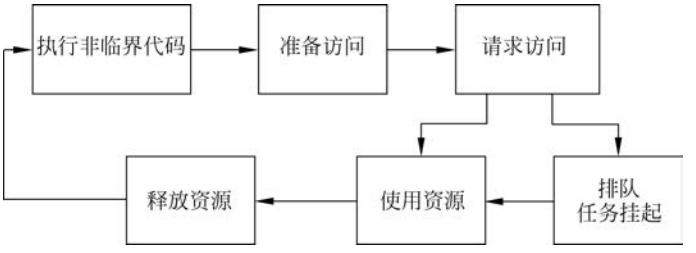

图 3.6　任务行为建模

如前所述，为每个受控的资源创建一个信号量，在编程术语中，信号量被看作一个命名的数据项。

下面来看一个简单的应用程序，即保护图 3.2 中的系数数据池。首先创建一个信号量，命名为 CoefficientsSemaphore，可执行的操作包括等待信号量 Wait(CoefficientsSemaphore)和发布信号量 Signal(CoefficientsSemaphore)。

二进制信号量只有两个值，“0”或“1”。“0”表示资源正在使用中，“1”表示资源当前空闲。在其初始形式中，信号量操作见代码清单 3.1 和代码清单 3.2。

代码清单 3.1

```
/* Wait(CoefficientsSemaphore)程序代码片段 */
if(CoefficientsSemaphore == 1)
{
  CoefficientsSemaphore = 0;
} /* end if */
else
{
```

代码清单 3.2

```
/* Signal(CoefficientsSemaphore) 程序代码片段 */
if(Task Waiting)
{
  WakeupTask();
}
else
{
```

在程序中，将在需要的位置使用上述代码段，如代码清单 3.3 所示。

代码清单 3.3

```
/* 示例片段 - 包含代码和伪代码 */
UnprotectedProgramStatements;
Wait(CoefficientsSemaphore);
UseSharedResource;
Signal(CoefficientsSemaphore);
UnprotectedProgramStatements;
```

wait 和 signal 操作被定义为“原语”类型，即每个操作都是不可分割的。换言之，一旦 wait 或 signal 处理开始，其对应的机器指令序列不能被中断。这是必不可少的，否则会遇到和单一标志互斥机制相同的问题。提供原子性操作不是一件轻松的任务，它可能会带来实现上的困难，但系统必须克服困难实现信号量。此外，这些原语操作必须由操作系统而非程序员保护。

二进制信号量可以实现为单字节，甚至是字节中的一位，使用一条“位设置和测试”指令

实现。但是，如果测试和检查涉及多条处理器指令时，该方法行不通。在这种情况下，通过在信号量操作执行之前禁用系统中断，确保操作的原子性。资源操作完成后，重新启用中断。

注意：此技术通常不适用于多处理器系统，多处理器系统需要一种硬件锁定机制。

wait 和 signal 也称为 P 操作和 V 操作，源自荷兰语词汇。关于它们实际指代的词存在一些分歧，最流行的是 prolaag 和 verhogen。

【译者注】 原语是为完成特定的功能而编写的一段程序，它在执行时不可分割、不可中断。

3.3.2　通用或计数信号量

我们重新构造单个共享资源，使其包含一组相同的共享对象。每个对象提供一个指定的服务。例如，共享对象可以是一组局域网队列，用于存储发送的消息。鉴于这种安排，在任务不使用同一队列的前提下，让多个任务访问资源是安全的。为了支持这一点，信号量结构改变为：

(1) 信号量有一个范围值(比如 0～4)，其初始值被设置为最大值(4)。

(2) 每个值对应于所提供资源的一个实例，0 指示所有资源都在使用。

(3) 当用户想要访问数据仓库时，它首先检查资源是否可用(不为 0 值)，见代码清单 3.4。如果允许访问，它会将信号量值减 1，并使用资源。

代码清单 3.4

```
/* 等待 CAN 队列 程序代码片段 */
if (CanQueue)  > 0)
{
   -- CanQueue;
}
else
{
  SuspendTask();
}
/* end if */;
```

当用户完成任务后，它将信号量值加 1 并退出数据仓库，如代码清单 3.5 所示。

代码清单 3.5

```
/* 发布 CAN 队列 程序代码片段 */
if (TaskWaiting)
{
  WakeupTask();
}
else
{
```

```
    ++CanQueue;
} /* end if */
CoefficientsSemaphore = 1;
```

CanQueue 的值控制对资源的访问，并定义可以使用的对象数；不允许为负值。

二值信号量可以被视为计数值为 1 的特殊的计数信号量。代码清单 3.6 的演示示例通过使用 ThreadX RTOS 的信号量结构进行了演示。

代码清单 3.6

```
/* 代码示例: RTOS -- ThreadX
             Wait 等价于: tx_semaphore_get
             Signal 等价于: tx_semaphore_put
             Semaphore 数据类型: TX_SEMAPHORE
*/
/* 创建值为 1 的计数信号量,位于文件范围 */
TX_SEMAPHORE ADCsemaphore;
int SemaphoreStatus;
SemaphoreStatus = tx_semaphore_create
                    (&ADCsemaphore, "ADCsema", 1);
/* 使用信号量控制共享对象的访问,函数范围 */
/******* 受保护代码段开始 *******/
/* 等待信号量 */
tx_semaphore_get (&ADCsemaphore);
/* 使用受保护资源 */
GetAnalogueInput (&RotorSpeed);
/* 发布信号量 */
 tx_semaphore_put (&ADCsemaphore);
/****** 受保护代码段结束 ******/
```

3.3.3 信号量的限制和缺陷

信号量已被广泛用于执行互斥策略，它易于理解、使用简单且易于实施。遗憾的是，以下其局限性及相关问题并没有受到重视。

(1) 信号量不会自动与特定的受保护对象相关联。然而，在实践中，正确配对它们至关重要。

(2) 在到目前为止描述的操作中，没有“看到”信号量状态的概念。请求者真正做的是问“我可以使用资源吗?”，如果答案是否定的，那么请求任务会自动挂起。

(3) 信号 wait(等待)和 signal(发布)是一对操作。遗憾的是，基本机制并没有强制执行这种配对。因此，一个任务可以单独调用任何一个操作，这会被认为是有效的源代码，结果可能导致非常不寻常的运行时行为。

(4) 没有操作限制信号量发布操作在等待操作之前调用，这也可能是奇怪运行时行为

的来源。

(5) 信号量必须对共享受保护资源的所有任务可见。这意味着任何任务都可以“释放”信号量(通过调用发布操作),即使这是一个编程错误。

(6) 仅使资源与信号量关联并不能保证其得到保护。如果存在进入保护区的“后门”路线(例如,使用声明为程序全局的资源),则可以绕过保护措施。

(7) 信号量还有一个更重要的问题,这与它的使用而不是构造有关。大多数程序在需要使用信号量时才实现它们,因此它们往往分散在代码中,通常很难找到。在小型设计中这是可以处理的,但对于大型设计则不然。因此,设计人员必须跟踪所有互斥活动,否则调试可能非常具有挑战性。而且,在后期设计中(在维护阶段),分散的信号量会让维护变得非常困难。通常,进行“简单”程序修改的结果会使软件带有非常奇怪的(出乎意料的)运行时行为。

3.4　互斥量

互斥量与信号量非常相似,但互斥量专门用于控制对共享资源的访问,即互斥(请记住,信号量本质上是一种流控制机制)。为避免混淆,信号量和互斥操作使用不同的操作名称。使用锁定(lock,又名等待)和解锁(unlock,又名发布)指示互斥操作。

互斥量与信号量的一个关键区别:释放(解锁)互斥量的任务必须是锁定它的任务。因此可以认为锁定任务拥有互斥量。代码清单 3.7 给出了一个使用互斥量的应用示例。

代码清单 3.7

```
/* 代码示例: RTOS 标准 -- Pthreads
     Lock 等价于: pthread_mutex_lock
     Unlock 等价于: pthread_mutex_unlock
     Mutex 数据类型: pthread_mutex_t
*/
/* 在文件范围内创建并初始化 mutex */
pthread_mutex_t ADCmutex;
pthread_mutex_init (&ADCmutex, NULL);
/* 使用 mutex 控制共享对象访问,函数范围 */
/********** 受保护代码段开始 **********/
/* 锁定互斥量 */
pthread_mutex_lock (&ADCmutex);
/* 使用受保护资源 */
GetAnalogueInput (&RotorSpeed);
/* 解锁互斥量 */
pthread_mutex_unlock (&ADCmutex);
/********** 受保护代码段结束 **********/
```

3.5 简单监视器

前面已经指出，信号量构造方式存在许多限制和问题（互斥量也存在类似问题），它们不是健壮的编程结构。我们想要的是一个替代品，在程序方面：

（1）为临界区域的代码提供保护。

（2）将数据与适用于该数据的操作一起封装。

（3）具备高可见性。

（4）易于使用。

（5）难以误用。

（6）简化证明程序正确性的工作。

满足这些标准的最重要和广泛使用的构造方式是监视器（其起源于 Dijkstra、Brinch Hansen 和 Hoare 的工作）。此处描述的构造使用原始监视器的简化版本，因此称为简单监视器。从根本上说，它通过以下方式防止任务直接访问共享资源。

（1）将资源（临界代码段）及其保护信号量或互斥信号量封装在一个程序单元内。

（2）将信号量/互斥信号量的所有操作限制在封装单元内部。

（3）将操作对“外部”世界隐藏，即它们对程序单元私有。

（4）防止直接访问信号量/互斥信号量操作和临界代码部分。

（5）提供间接使用共享资源的方法。

图 3.7 概念性地展示了这些信息，是早期信号量访问控制技术的改编版本。首先，所有组成部分都被封装，封装单元具有单个入口/出口点（相当于停车场的入口和出口通道）。输入与输出分离，输入被路由到访问控制机制。请注意，在这种安排中，入口驱动程序只能请求进入（相当于等待）。除此之外，访问行为如前所述。请注意，所有排队都是在封装单元内

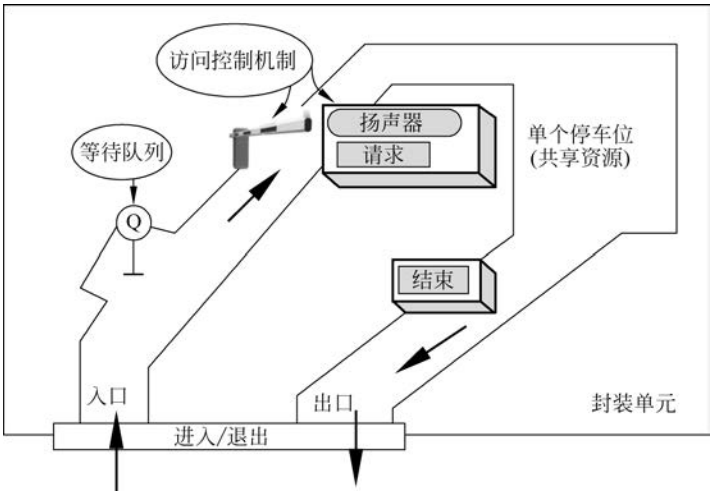

图 3.7 简单监视器概念图

完成的。

结束按钮(即发布)位于出口通道上,因此只能由离开的驾驶员操作。

一些编程语言提供相同或类似的构造方式(例如 Ada 中的受保护对象),然而 RTOS 中通常不会提供简单监视器机制。在这种情况下,用户必须自己构建它,其中一个关键特性是封装单元。如果使用 C++编程,那么显而易见的选择就是类。类为我们提供了所有必需的封装、信息隐藏和公共访问机制。在 C 中,我们可以通过将监控软件在一个“.c”文件中实现,并在相应的“.h”文件中提供其公共接口来模仿类的实现。

代码清单 3.8、代码清单 3.9 和代码清单 3.10 展示了一个实际的例子。

代码清单 3.8

```
/* 这是一个 .c 文件 */
/* 二值信号量 ADCsemaphore 已创建并初始化,在本文件范围内可见 */
int AnalogueInputMeasurement (int ChannelNumber)
{
int AnalogueValue = 0;
/********** 受保护代码段开始 **********/
/* 等待信号量 */
tx_semaphore_get (&ADCsemaphore);
/* 使用受保护资源 */
AnalogueValue = Convert (ChannelNumber);
/* 发布信号量 */
```

代码清单 3.9

```
/* 这是一个.h 文件,提供公共访问接口 */
int AnalogueInputMeasurement (int ChannelNumber);
```

代码清单 3.10

```
/* 任务代码 */
const int RotorSpeedChannel = 0;
const int RotorPositionChannel = 1;
int RotorSpeed = 0;
int RotorPosition = 0;
void main (void)
{
  …
  RotorSpeed = AnalogueInputMeasurement (RotorSpeedChannel);
  RotorPosition = AnalogueInputMeasurement (RotorPositionChannel);
  …
} /* main 结束 */
```

示例中,“.c”和“.h”文件的组合提供了函数 AnalogueInputMeasurement()的私有代码的整体封装和公共访问接口。函数封装了受保护的资源及其保护信号量;在“.c”文件之外无法等待或发布信号量 ADCsemaphore。此外,代码实现保证:

(1) get(等待)和 put(发布)以正确的顺序成对使用。

(2) 调用函数 AnalogueInputMeasurement()的任务拥有信号量。

(3) 当函数执行完成时,资源必须可用(即不能保持锁定状态)。

也可以用互斥信号量来代替信号量。但是,它不会为设计增加任何价值,因为两种机制的整体行为是相同的。总之,简单监视器用于控制对资源的访问,其比信号量强大得多,且易于使用、难以误用。

请注意,实际应用中对监视器的调用会嵌入各个任务的代码中。

3.6 互斥机制综述

互斥机制的主要目的是控制对共享资源的访问。共享资源包括硬件、共享数据和系统软件(例如 RTOS),见图 3.8。

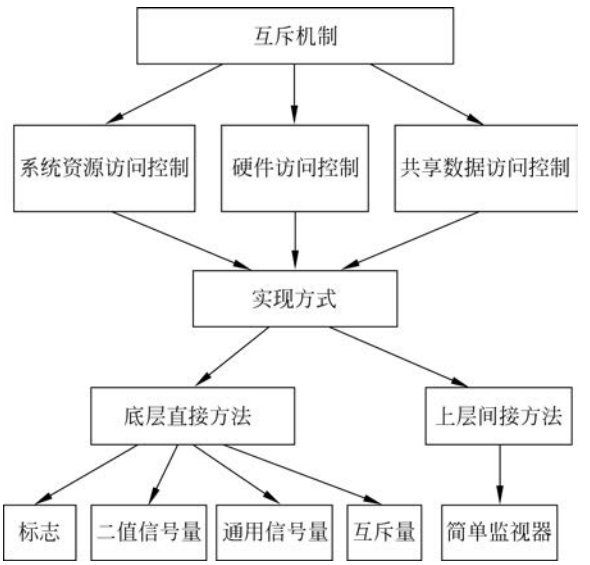

图 3.8 应用中的互斥机制

针对资源争用已提出了许多解决方案,但在实践中仅少数被使用。对于小的程序,或者并行性很少的应用,只要小心,底层直接方法即可满足需求。底层直接方法也适用于不使用操作系统的设计(对于大多数小型嵌入式功能而言是典型的)。在实时系统中,二值信号量往往比通用(计数)信号量得到更广泛的使用,两者没有根本性的区别,但对于外部设备的监控,二值信号量做得更好。即便如此,信号量和互斥量的特性也有可能使它们变得不安全,尤其是在以下情况下:

- 程序很大。
- 软件被构造为许多协作的并行任务。

在这种情况下,简单监视器是一种更好的选择。

3.7　回顾

通过本章的学习，应该能够达到以下目标。

- 完全理解为什么任务不能不受控制地使用共享资源。
- 了解什么是互斥以及它的作用。
- 清楚忙-等待(busy-wait)和挂起-等待(suspend-wait)两种互斥方法的区别。
- 认识到挂起等待方法的优势。
- 清楚如何使用单个标志来实现忙等待互斥机制。
- 理解为什么某些操作必须是原子性的。
- 清楚二值信号量和计数信号量的概念及使用。
- 了解互斥信号量如何改进信号量的缺陷。
- 明白为什么信号量和互斥量不是健壮的程序结构。
- 了解什么是简单监视器以及它如何克服信号量和互斥信号量的缺点。
- 理解简单监视器的代码结构及其使用。

下面可以基于本节的理论知识开始实践工作，即《嵌入式实时操作系统——基于STM32Cube、FreeRTOS和Tracealyzer》一书中实验5～实验10中涉及的内容。

【译者注】《嵌入式实时操作系统——基于STM32Cube、FreeRTOS和Tracealyzer》已经由清华大学出版社于2021年5月出版。