# 第 3 章

# 共享资源使用

## 3.1 实验 5 访问竞争问题分析

实验目的：展示在多任务系统中，使用共享资源时面临的访问竞争问题。

### 3.1.1 竞争问题介绍

如果你已经认真地完成了前面章节的实验，那现在你应该对如何实现多任务设计有了深层次的理解。这些实验虽然很重要，但是提供的知识还是十分有限的。它们只是处理了一些相互独立的、没有交互的任务，而且，在现实世界中，这种应用场景非常罕见。在大多数实际设计中，任务间通常基于共用的软件和/或硬件对象实现交互。因此，不幸的是，任务之间可能会因为对共享对象的“同时”访问而无意间相互干扰。

本实验的目的是说明竞争问题是真正存在的，而不仅仅是一个抽象的学术问题。实验旨在表明，如果不使用保护措施，将会产生共享资源使用冲突。

### 3.1.2 竞争问题概述

图 3.1 展示了描述竞争问题实现的软件任务框图，软件中包含两个用户任务和一个共享数据对象。共享数据对象用来保存应用中使用的数据。

两个任务，操作同一个数据对象，使任务间以一个简单直接的方式交换信息。通常，数据对象的数据可以直接写入/读取。该数据对象有如下几个特点：

(1) 是一个定义良好的软件组件。

(2) 对整个系统任务设计可见。

(3) 替代使用全局变量进行信息交换。

但是，何时执行读取和写入操作是由各个任务单独控制的，不存在全局控制策略。因此，始终存在多个任务同时操作数据对象的情况。例如，一个任务在写数据时，另一个任务在执行相应的读操作，此时这种资源争用可能会导致数据损坏，从而引起任务行为异常。

只有在多处理器及多核系统中才会产生真正的同时访问；单处理器设计中不会发生该

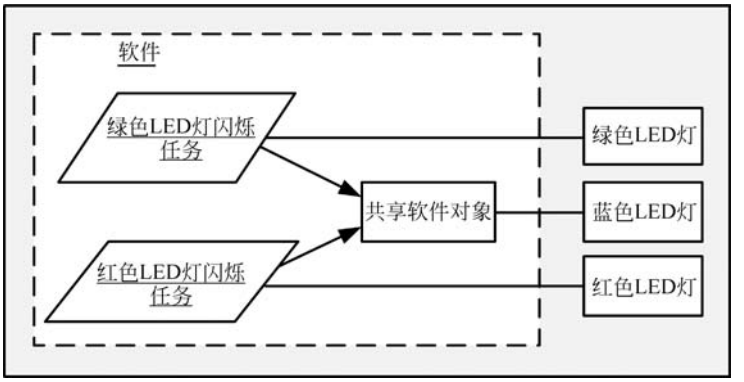


图 3.1 软件任务框图

情况。在本实验的设计中,它貌似是一种同时访问形式,但仍然会有任务使用导致损坏数据的可能。

### 3.1.3 实验细节

**1. 基本实现**

在本实验中,共享数据对象旨在模仿可读写数据存储空间的行为,该数据空间特点如下:

(1) 保存多个数据项;

(2) 所有任务都可以访问,执行读取和写入操作;

(3) 需要访问时间(处理器时间),必须考虑在系统时间负荷中。

模拟此类操作的有效方法是使用数据对象将消息发送到终端设备。任务之间的交互将通过消息发送过程,以可视化方式展示出来。不幸的是,使用 STM 开发板实现终端访问太复杂。通常,为了实现这一点,你可以:

(1) 使用微控制器提供的简单串行通信特性(但这需要额外的硬件转换接口);

(2) 或使用微控制器的 USB OTG 功能(但这是一项颇具挑战性的软件编写工作)。

因此,本实验中使用一个非常简单的方法:直接使用软件延时方式模拟读/写操作过程。当然,该方法中,我们需要手动检测"同时访问"是否发生。一个简单的方法是在访问函数中使用访问指示器——Start flag 标志。这是一个二进制标志位,具有两个值,Up(1)和 Down(0),初始化为 Up。Up 表示资源未被使用;Down 表示任务正在执行共享数据对象相关的代码。

因此,当任务访问临界区域时,Start flag 标志可能的状态如下。

(1) Start flag 标志为 Up:一切良好;

(2) Start flag 标志为 Down:另一个任务在运行共享代码,但该任务现在已被抢占。

在本实验中,我们将通过点亮蓝色 LED 灯来检测资源争用。因此,访问函数(伪代码)的程序设计如下:

**注意**：Start flag 标志初始化为 Up。

(1) 检查 Start flag 标志是否为 Up。

(2) 如果 Start flag 标志为 Up，将其改为 Down；否则给出争用警告——点亮蓝色 LED 灯。

(3) 运行软件延时，模拟读写操作。

(4) 设置 Start flag 标志为 Up。

如果你愿意，可以轻松修改此参数，以便对任务的冲突次数进行计数。在这里，关注点只是为了检测冲突已经发生。

**2. 具体实现**

1) 任务 1：闪烁绿色 LED 灯

实现代码结构如下：

```
循环开始
  点亮绿色 LED 灯
  访问共享数据
  关闭绿色 LED 灯
  延时 0.5s
循环结束
```

2) 任务 2：闪烁红色 LED 灯

实现代码结构如下：

```
循环开始
  点亮红色 LED 灯
  访问共享数据
  关闭红色 LED 灯
  延时 0.1s
循环结束
```

**注意**：在系统中，将红色 LED 灯闪烁任务的优先级设置为较高的级别。

3) 共享数据访问函数

建议实现代码：

(1) 检查 Start flag 是否为 Up，如果为 Up，则将 Start flag 标志设置 Down，否则点亮蓝色 LED 灯。

(2) 模拟执行读/写操作 500ms。

(3) 将 Start flag 标志设置为 Up。

编译、下载和运行单个任务，以检查时间是否正确。一旦对两个任务的独立运行状态满意，使能两个任务同时运行。如果使用了推荐的时间配置，则应用运行后，蓝色 LED 灯很快会点亮，表明任务冲突已经发生。

**3. 附加实验(可选)**

如果你已经理解了实验的软件实现，就会明白，当首次检测到争用时，蓝色 LED 灯点

亮,并且永远不会熄灭。现在,我们修改 LED 灯常亮的方式,通过闪烁 LED 灯,对争用的频率得到一个“粗糙但直观”的感觉。显然,点亮的时间必须足够长,以便你能够看到 LED 灯何时亮起。因此,修改访问函数,使蓝色 LED 灯保持点亮状态约 0.1s,然后熄灭。

正如前所述,这个实验有点粗糙和简单,不能提供准确的争用次数,但它很有启发性,值得去做。

### 3.1.4 实验回顾

你现在应该:

(1) 认识到当任务共享项目时,始终存在“同时访问”的可能性。

(2) 了解争用会产生的问题,问题严重程度完全取决于共享的内容及其使用方式。

(3) 认识到除非采取特殊的检测措施,否则此类争用可能完全不会被发现。

(4) 认识到我们在程序中使用全局变量时,可能也会有类似的问题。

(5) 推断出解决争用问题的唯一安全、可靠的方法是设计代码时避免同时访问。

(6) 意识到典型的多任务模型由并发单元(任务或活动对象)和被动对象(在本例中为共享数据对象)组成。

(7) 认识到非并发项是顺序代码单元,仅在被任务调用时执行,它们在调用任务的上下文中运行。

## 3.2 实验 6 通过挂起调度器消除资源竞争

实验目的:演示消除资源争用的一种简单方法——挂起调度器。

### 3.2.1 方法介绍

如果我们可以确保,任何时候有且只有一个任务在使用共享对象,那么资源访问冲突就不可能发生。实现这个最简单的方法是在共享资源访问期间挂起任务调度器。实际上,该方法使用了 RTOS 提供的接口函数,通过在访问过程中关闭中断来实现。

### 3.2.2 实验细节

我们将使用以下两个 FreeRTOS 函数来禁用和启用中断。

禁用中断:taskENTER_CRITICAL()。

启用中断:taskEXIT_CRITICAL()。

函数的详细信息,请访问 http://www.freertos.org/taskENTER_CRITICAL_taskEXIT_CRITICAL.html。

修改实验 5 每个任务中的共享资源访问实现:禁用中断,访问共享资源,启用中断,任务其他部分代码保持不变。

重新编译、下载并运行代码,你会发现,蓝色 LED 灯永远不会点亮。

### 3.2.3　实验回顾

你现在应该：

(1) 了解为什么在访问临界代码段时停止调度可以防止产生不良影响。

(2) 认识到该技术的实现非常简单。

现在，我们来更深入地探讨一下这种技术。这种技术的好处是它的使用非常简单，绝对保证有效。缺点是停止调度会停止并发单元的执行，会暂停多任务处理，并且中断禁用的时间越长，对系统的影响越大。所以黄金法则是：如果使用关中断的方式保护共享资源，进入和退出临界段的时间必须非常快。

## 3.3　实验 7　演示系统性能的降低

实验目的：演示在多任务设计中，共享资源的使用会导致系统性能下降。

### 3.3.1　介绍

在设计多任务系统时，必须为系统运行时行为开发一个良好的模型。首先需要确定各个任务的执行时间，然后对它们的整体行为建模。当任务相互独立时，任务整体行为非常简单直观；当任务之间存在交互时，系统行为会变得比较复杂。如果不考虑任务的交互，与预期相比，最好的情况是只降低了系统的响应能力；在最坏的情况下，它们可能会导致灾难性的系统故障。

本实验的目的是展示资源共享使用对三个任务系统的性能影响。该设计中，每个任务的功能是闪烁指定的 LED 灯；任何系统性能的变化将导致 LED 灯的闪烁模式改变。为了可以清楚地看到 LED 灯的状态变化，在实验中，我们需要适当延长访问时间。

这是一个非常有启发性的实验，所以请认真完成本实验所有的工作。请使用推荐的时间参数，以产生可观察的效果。

### 3.3.2　实验细节

#### 1. 概述及关键代码

图 3.2 展示了本实验的任务实现框图，软件包含三个用户任务和一个共享软件对象。实验的目标是证明在多任务设计中使用共享资源会降低系统的整体性能。

将绿色和红色 LED 灯闪烁任务的优先级设置为 normal(注：CubeMX 中 normal 对应的优先级值为 3)，橙色 LED 灯闪烁任务优先级设为更高级别 above normal(注：above normal 对应的优先级值为 4)。

绿色 LED 灯和红色 LED 灯闪烁任务的代码实现，如下所示：

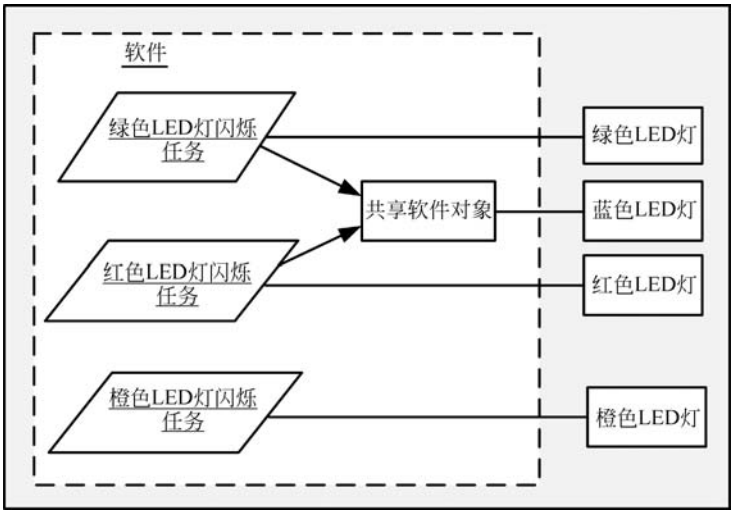


图 3.2　任务实现框图

```
for(;;)
{
  HAL_GPIO_WritePin(GPIOD, GPIO_PIN_xx, GPIO_PIN_SET);
  AccessSharedData();
  osDelay(DelayTimeMsec);
  HAL_GPIO_WritePin(GPIOD, GPIO_PIN_xx, GPIO_PIN_RESET);
  osDelay(DelayTimeMsec);
} // for 循环结束
```

闪烁红色 LED 灯时：DelayTimeMsec＝550ms(注意：延时使用 osDelay 实现，而非 osDelayUntil)。

橙色 LED 灯闪烁任务代码实现如下：

```
for(;;)
{
  HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_13);
  osDelay(50);
} // for 循环结束
```

橙色 LED 灯将以 10Hz 的频率闪烁。

共享数据函数访问代码如下：

(1) 检查 Start flag 标志是否为 Up，如果为 Up，将 Start flag 标志更新为 Down；否则点亮蓝色 LED 灯。

(2) 模拟读/写操作，执行 1s。

(3) 关闭蓝色 LED 灯(检测到冲突后,如果希望关闭蓝色 LED 灯,插入此代码)。

(4) 设置 Start flag 标志为 Up。

实验 7 部分实现中,要求使用资源控制机制访问共享资源。本例中通过关中断的方式实现,需要时修改绿色 LED 灯和红色 LED 灯任务中的相关代码,如下所示:

- 禁用中断 taskENTER_CRITICAL();
- 访问共享资源;
- 启用中断 taskEXIT_CRITICAL()。

任务其余部分代码实现保持不变。

图 3.3 给出了每个任务运行时大概的时间数据。

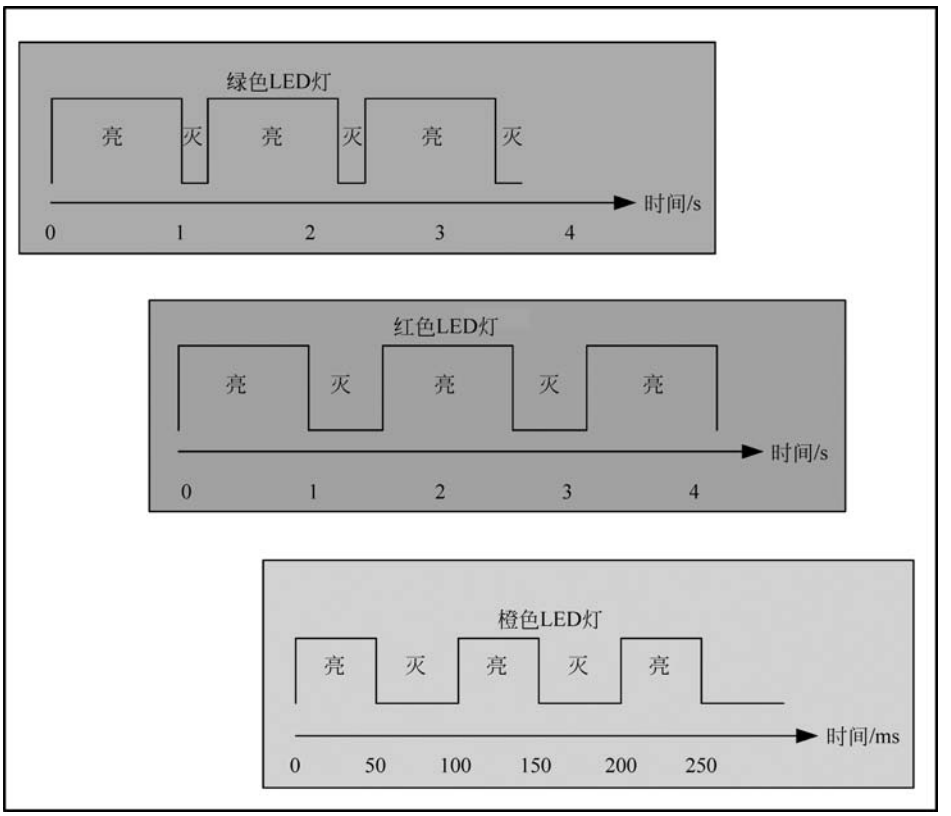


图 3.3　LED 灯闪烁行为时序

**2. 实验实现**

本实验实现分为以下三步:

(1) 单独运行每个任务,确认单个任务的运行时间;

(2) 激活所有任务,不使用互斥机制访问共享资源,演示系统行为;

(3) 使用控制机制访问共享资源,演示系统行为。

实验 7.1：每次仅运行一个任务，检查实际运行行为与预测值是否相符。

实验 7.2：所有任务处于活动状态，共享资源代码调用不使用互斥机制，运行系统，与实验 7.1 比较运行时行为。

实验 7.3：使用 taskENTER_CRITICAL()和 taskEXIT_CRITICAL()函数保护共享资源代码，重复实验 7.2，比较运行时行为。

### 3.3.3 实验回顾

前面提到，我们人为地延长了共享资源访问时间，扩大了其对系统性能的影响，以便通过可视的方式展示。我们可以通过减少处理器利用率，将其对性能的影响降低至合理水平，但影响仍会存在。此外，观察这种交互行为非常困难，我们还需要使用专门的运行时分析工具。

完成本实验后，你现在应该：

(1) 认识到在多任务实现中执行任务时，可能会产生与任务单独运行时完全不同的结果。

(2) 明确当任务间共享资源时，通常肯定会发生争用。

(3) 意识到准确地预测何时将发生访问争用非常困难，甚至几乎是不可能的。

(4) 理解要准确预测资源争用产生的影响同样困难。

(5) 领会到可以通过禁止任务切换(禁用中断)在临界资源访问期间消除争用。

(6) 能看到禁用中断对运行时行为的影响。

(7) 推断出在多任务编程时，需要非常小心地使用中断禁用机制。

(8) 理解为什么中断不应长时间禁用。

(9) 意识到禁用中断的任务将成为系统中最高优先级的任务。

这个简单的实验应该可以帮助你理清设计重点，以实现时间关键的实时多任务系统设计：

(1) 设计之前，需要彻底评估系统的时间性能。

(2) 必须尽量减少时间不确定性，可以最小化资源共享来降低时间不确定性的影响。一个重要影响因素是任务数量。一般来说，任务越多，共享的资源越多，任务运行时行为变化越大。所以黄金规则是：最小化设计中的任务数量。

(3) 将共享资源的使用时间保持最短，不在共享代码中执行冗长的处理。

## 3.4 实验 8 使用信号量保护临界代码

实验目的：演示如何使用信号量机制来保护临界段代码，消除资源冲突。

### 3.4.1 背景介绍

实验 7 演示了使用中断禁用作为共享资源访问控制机制，该机制实现非常简单，但可能

导致多任务系统的运行时行为遭到严重破坏，不适合作为一个通用解决方案。

使用关中断机制面临的首要问题是，中断禁用会影响整个系统，一般来说是不可取的。我们需要的是一种对系统干扰最小的方法。幸运的是，有三种互斥的访问控制机制可用：信号量、互斥信号量和微型监视器（受保护对象）。

本实验的目的是介绍和展示信号量机制的使用。

## 3.4.2　实验细节

### 1. 信号量介绍及其创建

本实验代码基于实验 7.3，做了简单修改：将关中断调用替换为使用信号量。信号量可被视为 RTOS 定义的程序变量。与所有变量一样，信号量必须先创建才能使用。信号量创建过程包含下列步骤：

（1）定义信号量“句柄”。

（2）定义信号量。

（3）创建信号量。

以上步骤所需的所有代码都可以由 CubeMX 自动生成。只需加载 Cube 项目，在 FreeRTOS 配置选项卡中选择 Timers and Semaphores，如图 3.4 所示。

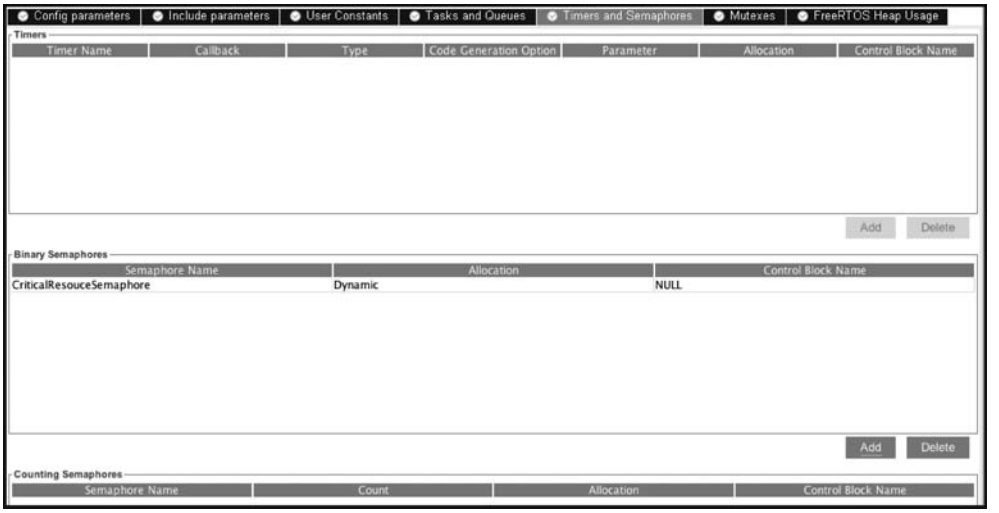


图 3.4　CubeMX 信号量配置界面

添加二进制信号量并命名（CriticalResourceSemaphore）。现在生成并检查项目源代码。与实验 7 相比，你会发现 main.c 文件中增加了信号量相关的代码，变化总结如下：

```
/* 私有变量 ------------------------------------------------------------ */
osSemaphoreId CriticalResourceSemaphoreHandle;
/* 创建信号量 */
/* 定义并创建 CriticalResourceSemaphore */
osSemaphoreDef(CriticalResourceSemaphore);
CriticalResourceSemaphoreHandle =
            osSemaphoreCreate(osSemaphore(CriticalResourceSemaphore),1);
```

**注意**：函数 osSemaphoreCreate 的第二个参数是信号量创建等待超时值，其单位为毫秒，1 为自动生成的默认值。

**2. 信号量使用**

信号量是一种任务流控制机制，信号量有以下两种状态：

- 发布态(released，同义词 free、available、unlocked)。
- 锁定态(locked，同义词 taken、unavailable、acquired)。

当信号量锁定时，申请信号量的任务将停止运行；信号量处于发布状态时，允许申请信号量的任务继续执行。

任务可以通过查询信号量的状态，确定是否停止或继续执行，使用如下所示 API：

```
osSemaphoreWait(CriticalResourceSemaphoreHandle, WaitTimeMilliseconds);
```

参数 WaitTimeMilliseconds 决定任务继续执行代码前等待的时间。此值取决于设计目标，由用户定义。例如，当任务只想检查信号量的当前状态时，可以将调用等待的时间参数值设置为 0。

如果任务需要释放信号量，它可以调用下列 API 向信号量发送消息：

```
osSemaphoreRelease(CriticalResourceSemaphoreHandle);
```

API 使用示例如下：

```
//申请信号量：等待信号量就绪
osSemaphoreWait(CriticalResourceSemaphoreHandle, WaitTimeMilliseconds);
访问共享(临界)资源
//发信号：释放信号量
osSemaphoreRelease(CriticalResourceSemaphoreHandle);
```

使用 CubueMX 工具创建信号量时，信号量默认设置为发布态。

**译者注**：使用 CubeMX 工具创建信号量时，CMSIS v1 API 创建信号量函数 osSemaporeCreate()调用的是 vSemaphoreCreateBinary()。在 FreeRTOS v7.6.0 版本中，增加了 xSemaphoreCreateBinary()，使用该 API 创建信号量时，其默认为阻塞态。

**3. 实验内容**

本实验分为两步实现。

实验 8.1：修改实验 7.3 代码(3.3.2 节)，修改如下所示。

(1) 将 taskENTER_CRITICAL()更改为 osSemaphoreWait(CriticalResourceSemaphoreHandle，WaitTimeMilliseconds)。

(2) 将 taskEXIT_CRITICAL()替换为 osSemaphoreRelease(CriticalResourceSemaphoreHandle)。

构建、下载和运行软件。观察 LED 灯的闪烁行为，并与实验 7.3 中的结果进行比较。

实验 8.2：将 WaitTimeMilliseconds 参数值设置为 0，重新执行生成的应用，观察 LED 灯的闪烁行为。

### 3.4.3　实验回顾

如果实验 8.1 正确运行，LED 灯的闪烁行为将产生非常显著的变化：由于橙色 LED 灯闪烁任务是系统中优先级最高的任务，其行为将完全按照预期方式执行，原因是信号量机制作用范围有限，它仅影响共享临界资源的任务。

观察实验 8.1 的运行结果，还将看到蓝色 LED 灯不会点亮，即系统运行过程中没有资源争用发生。

运行实验 8.2 的代码，你会发现蓝色 LED 灯会点亮，发生了资源争用。该现象符合预期，在信号量等待时间为 0 时，即使信号量不可用时，任务还将继续执行，会访问共享资源，从而导致访问冲突。

## 3.5　实验 9　使用互斥信号量保护临界代码

实验目的：演示使用互斥信号量保护临界代码段，消除资源竞争。

### 3.5.1　实现细节

本实验实现基于实验 8 的代码做了简单修改：用互斥信号量替换信号量。

**1. 创建互斥信号量**

创建互斥信号量所需的代码可以由 CubeMX 自动生成。只需加载 Cube 项目，在 FreeRTOS 配置选项卡中选择 Mutexes，如图 3.5 所示。

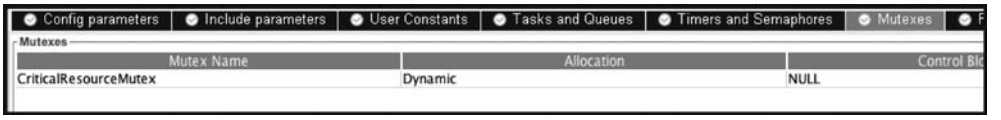


图 3.5　CubeMX 互斥信号量配置

添加互斥信号量并命名(CriticalResourceMutex)，生成并检查项目源代码，代码实现类似于实验 8，但信号量被互斥信号量替换，互斥信号量相关代码如下：

```
/* 私有变量 ------------------------------------------------------------ */
osMutexId CriticalResourceMutexHandle;
/* 创建互斥信号量 */
/* 定义并创建 CriticalResourceMutex */
osMutexDef(CriticalResourceMutex);
CriticalResourceMutexHandle
```

**2. 互斥信号量使用**

任务可以通过查询互斥信号量的状态,确定是否停止或继续执行,API 如下所示:

```
osMutexWait(CriticalResourceMutexHandle, WaitTimeMilliseconds);
```

参数 WaitTimeMilliseconds 决定任务继续执行代码前等待的时间,与信号量机制完全相同。

如果任务希望释放资源,它会向互斥信号量发送以下消息:

```
osMutexRelease(CriticalResourceMutexHandle);
```

需要访问共享资源的任务,按下列顺序使用互斥信号量 API:

```
//申请互斥信号量: 等待互斥信号量
osMutexWait(CriticalResourceMutexHandle, WaitTimeMilliseconds)
使用共享(临界)资源
//发布互斥信号量: 互斥信号量释放
osMutexRelease(CriticalResourceMutexHandle);
```

**注意**:*互斥信号量创建后,其初始状态为已释放,表示共享资源可用。*

构建并运行实验代码,观察运行结果。

### 3.5.2 实验回顾

通过观察运行时的现象发现,实验 9 的运行时行为与实验 8.1 的行为完全相同。因此,我们可以推断出互斥信号量实现的功能与信号量完全相同。但除此之外,互斥信号量的使用还包含下列特性:

(1) 只有锁定互斥信号量的任务才能释放它(所有权属性)。

(2) FreeRTOS 互斥信号量支持优先级继承机制。

本实验没有展示互斥信号量的这两个属性,我们将在后面的实验中演示这些功能。

## 3.6 实验 10 使用封装机制提升系统安全

实验目的:演示通过信号量封装对象可以提升软件的功能安全和信息安全。

### 3.6.1　机制介绍

通过前面的实验，我们已经了解如何消除多任务处理中的资源争用问题。但这些也反映了一个重要的信息：如果你希望生成可靠的软件，临界资源应始终受到保护。生成高质量的软件应该是设计师的默认职责。

信号量和互斥信号量都可以用作共享资源保护机制。然而使用它们必须非常小心，否则可能会导致其他问题。以信号量为例，思考以下问题：

(1) 信号量不会自动与特定的受保护对象关联。然而在实际应用中，正确地将信号量与保护对象进行关联至关重要。

(2) 信号量等待和释放操作必须成对使用。遗憾的是，信号量机制并没有强制执行配对。因此任务如果只调用其中之一，也会被视为有效的代码，这可能会导致非常不正常的运行时行为。

(3) 没有机制阻止等待信号量之前，先调用发信号操作，可能导致奇怪的系统行为。

(4) 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通过调用发信号操作释放信号量，即使这是一个编程错误。

(5) 资源与信号量的关联不能保证安全，如果存在绕过保护区的“后门”，就可以绕过安全措施，如全局变量。

互斥信号量比信号量可靠，但仍然具备上述的大部分缺点。因此，关键问题是如何改进？有两件事情很清楚，需要：

(1) 将受保护对象(临界段)与保护机制(信号量或互斥信号量)关联起来。

(2) 确保任务不能直接访问受保护对象。

能满足需求的一种简单方法是将受保护对象及其保护机制封装在一个程序单元中，这种结构我们称为“简易监视器”。

### 3.6.2　实现概述

本质上，简易监视器机制可防止任务直接访问共享资源：

(1) 将共享资源(临界代码段)及其保护信号量封装在一个程序单元内。

(2) 将所有信号量/互斥信号量的相关操作保持在封装单元内。

(3) 对外部程序隐藏操作细节。

(4) 防止直接访问信号量/互斥信号量和临界代码段。

(5) 提供共享资源的间接使用方法。

从概念的角度来看，简易监视器的模型如图 3.6 所示。

该模型中，信号量和临界代码段封装在一个程序单元中。与前面的实验一样，对临界代码段的访问由信号量控制，封装对象对外部软件不可见；对封装对象的所有访问必须通过接口函数实现。

C 语言中，封装的基本实现形式是函数。本实验中，我们将使用函数形式构建一个简易

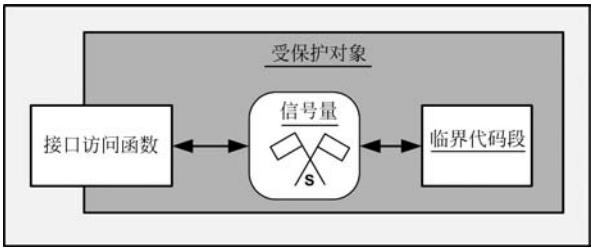


图 3.6　简易监视器(受保护对象)

监视器。

虽然在本实验中使用了信号量,但我们也可以使用互斥信号量实现,并且互斥信号量是FreeRTOS中实现互斥保护的首选方法,稍后我们将解释原因。就目前而言,使用哪种方式并不重要,如果你能用信号量解决该问题,你也可以毫不费力地使用互斥信号量实现。

### 3.6.3　实验实现

图 3.7 展示了此实验的任务实现框图,实验代码基于实验 8,使用相同的时间参数。在用户代码中,需要实现一个 AccessSharedData 函数,简易监视器函数的伪代码实现如下:

```
锁定信号量(osSemaphoreWait)
//临界段代码开始
  检查并基于 Start flag 的状态操作
  模拟读/写操作
  设置 Start flag 标志为 Up
//临界段代码结束
解锁(释放)信号量(osSemaphoreRelease)
```

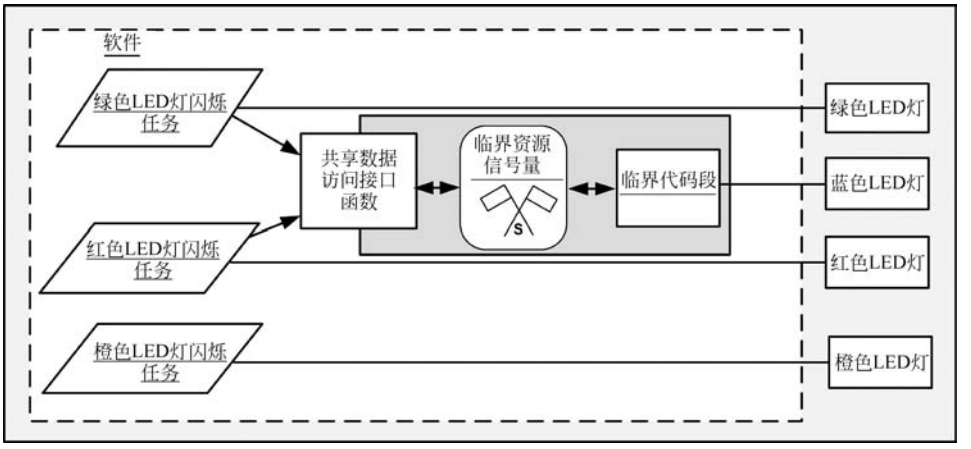


图 3.7　系统任务框图

当绿色或红色 LED 灯任务希望访问临界代码时，它只需调用此函数。

构建、下载并运行软件，观察 LED 灯的闪烁行为；并与实验 8.1 的运行结果进行比较，你会发现，最终得到的行为与实验 8 中使用信号量时的运行行为相同。然而，本实验的实现方法更安全、更健壮。

### 3.6.4　实验回顾

通过本实验，我们实现了如何以可控的方式处理资源争用。在实现中，可以看到：

(1) 任务代码中没有信号量调用。

(2) 受保护代码是私有的。

(3) 应用程序不能直接使用受保护的代码。

(4) 应用程序访问受保护代码的唯一方法是调用监视器函数，此调用(全局对象)作为监视器的公共函数接口。

(5) 与实验 8 相比，此方法提供了更安全、更可靠的程序实现。

毫无疑问，与早期工作相比，本节描述的监视器方法显著地提高了代码的质量、鲁棒性等。不幸的是，它有一个弱点，CubeMX 生成的信号量本身是一个全局对象，因此，信号量不受保护，它可以被任何任务滥用或误用。在后面的实验中，我们将介绍如何克服这个弱点，以产生一个真正健壮的、可移植的简易监视器。

## 3.7　实验 11　优先级反转影响演示

实验目的：演示多任务设计中，优先级反转问题对系统的影响。

### 3.7.1　介绍

通过前面的实验，我们已经得出结论，多任务系统的时间行为很难预测。仅在简单的情况下，如低处理器利用率的独立周期任务系统中，我们的预测才可能是正确的。然而，当任务间共享软件项时，就很难产生确定的时序，特别是在使用优先级抢占调度策略时，问题可能会更加复杂。这些方面已在 *Real-Time Operating Systems Book1—The Theory* 一书中讨论过，说明优先级反转问题如何产生额外的延迟。

本实验旨在生成完成一个优先级反转效果的实际运行示例。

### 3.7.2　实现及关键代码

图 3.8 展示了实验实现的系统任务框图。系统中包括三个用户任务和一个非并发保护对象(与实验 10 完全相同)。然而，代码实现与前面的实验存在差异，因此在重用软件时要小心。

本实验分为以下四步实现：

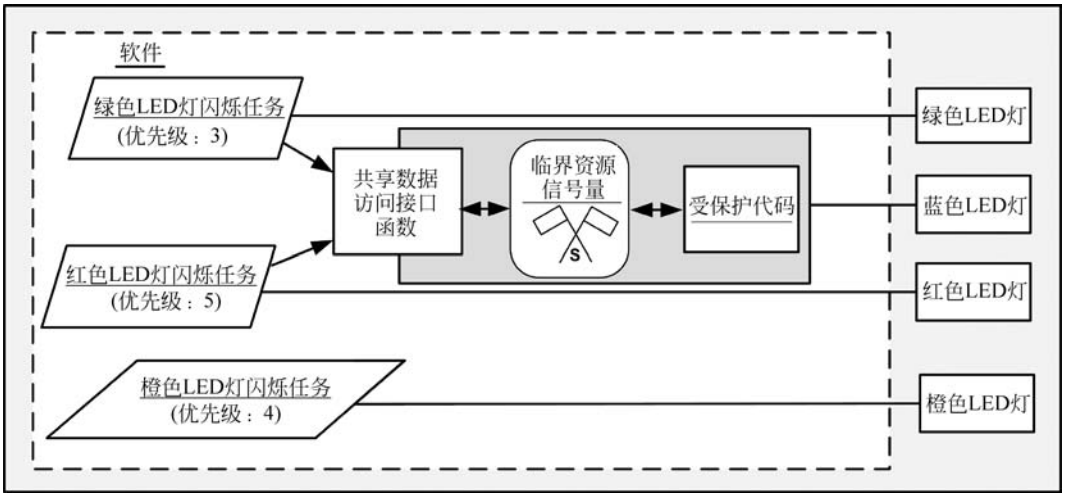


图 3.8 系统任务框图

(1) 实验 11.1：演示没有资源争用，也没有任务交互引起延迟的情况下，所有任务的运行行为。

(2) 实验 11.2：演示使用互斥信号量防止资源争用，为后面的优先级反转演示提供时间数据。

(3) 实验 11.3：演示经典优先级反转问题。

(4) 实验 11.4：重复步骤(3)，增加任务执行相关的可视化信息。

可以通过观察 LED 灯闪烁模式来推断代码的运行行为。对于本实验，请使用建议的时间参数，方便比较实际运行结果与书中给出的结果。

实验完成后，我们将解释实现这组特定实验的原因。

### 3.7.3 实验实现

本实验最重要的一点是任务不能作为周期任务运行。任务以运行至完成方式执行一次，以便观察、记录运行行为，并与理论预测进行比较，为此，每个任务在任务代码结束时，执行一条任务挂起操作。在 FreeRTOS 中，挂起操作实现 API 如下所示：

```
vTaskSuspend(NULL);
```

该函数的更多信息，请参阅 http://www.freertos.org/a00130.html。

各个任务的功能行为非常相似。橙色 LED 灯闪烁任务的程序结构如下：

```
代码开始
  以 10Hz 速率闪烁橙色 LED 灯 4s
  挂起任务
代码结束
```

绿色和红色 LED 灯闪烁任务的基本结构如下：

```
代码开始
  访问共享数据(注意:必须是第一个操作)
  以 10Hz 的速率闪烁 LED 灯 4s
  挂起任务
代码结束
```

受保护对象的程序结构如下：

```
代码开始
  获取信号量
  模拟读/写操作; 以 10Hz 的速度闪烁蓝色 LED 灯 2s
  释放信号量
代码结束
```

不用担心计时是否精确，代码中没有使用 osDelay 或 osDelayUntil 延时 API；所有计时操作由软件实现，原因是这些延时 API 会导致任务重新调度，将完全改变任务运行时行为。

**注意**：任务的优先级设置如图 3.8 所示，用户任务中，红色 LED 灯闪烁任务优先级最高，其次是橙色 LED 灯任务，绿色 LED 灯闪烁任务优先级最低。优先级设置可以使用 CubeMX 配置功能实现。

**1. 实验 11.1**

本实验的主要目的是让用户思考共享资源是如何影响系统响应能力的。

应用运行后，所有任务同时启动。观察 LED 灯的闪烁顺序，结果应与图 3.9 类似(实际结果可能会有所不同，具体取决于软件延时值)。

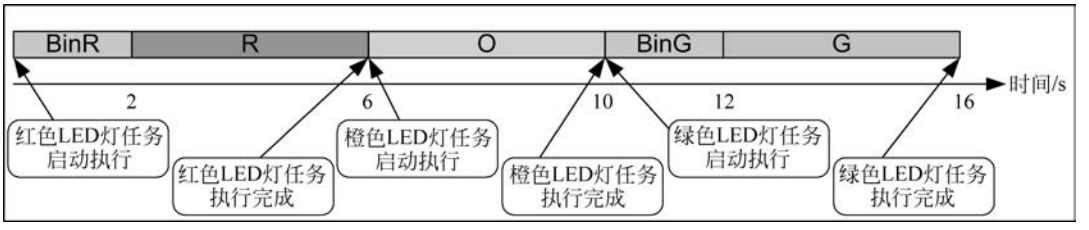


图 3.9　实验 11.1 的 LED 灯闪烁时序

临界代码段的代码将在绿色或红色 LED 灯任务中执行，软件总运行时间为

$$(3 \times 4s) + (2 \times 2s) = 16s$$

**注意**：BinG 表示蓝色 LED 灯闪烁，但相关代码在绿色 LED 灯任务上下文中执行。同理，BinR 表示蓝色 LED 灯闪烁，但相关代码在红色 LED 灯任务上下文中执行。

理解图 3.9 中时间的计算过程，它告诉我们一个简单的事实，增加共享资源的使用会导致系统响应能力降低。

这个实验给我们的提示是，要提高系统响应能力需最大限度地减少资源共享。为了实

现该目的,可以尽量减少设计中的任务数。

**2. 实验 11.2**

此实验实质上是一个使用共享资源访问控制机制消除资源争用的重复操作。

实验实现:

(1) 仅运行红色和绿色 LED 灯任务。

(2) 红色 LED 灯任务启动后,挂起一段时间,以允许优先级较低的绿色 LED 灯任务运行,直至其被抢占。

本实验中,红色 LED 灯闪烁任务的程序结构如下:

```
代码开始
  挂起 1s,使绿色 LED 灯任务运行,可以使用 osDelay(1000)实现
  访问共享数据
  以 10Hz 的速率闪烁 LED 灯 4s
  挂起任务
代码结束
```

重新运行代码,观察实际 LED 灯闪烁顺序,结果应如图 3.10 所示。

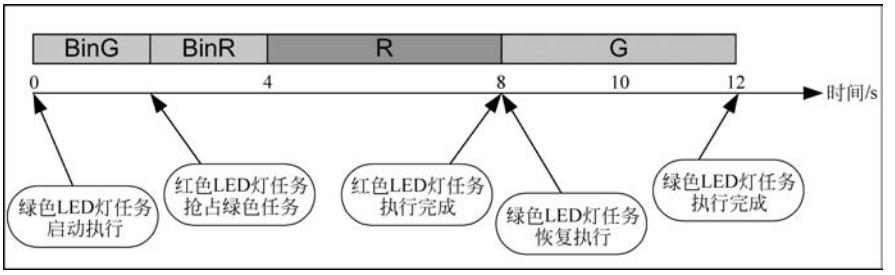


图 3.10 实验 11.2 的 LED 灯闪烁时序

图 3.10 的时序准确地解释了目标系统的运行行为。

**3. 实验 11.3**

本实验非常清楚地说明了什么是优先级反转。实现要点如下:

(1) 运行实验中的三个任务。

(2) 对于红色和橙色 LED 灯任务,启动后挂起 1s,使最低优先级的绿色 LED 灯任务能够运行,并在其被抢占之前锁定信号量(受保护对象)。

重新运行代码,观察 LED 灯闪烁顺序,闪烁结果应与图 3.11 相同,分析并了解目标系统中发生的行为,尤其是在时间段 0～1s、5～6s 和 6～8s 内的行为。与图 3.9 进行比较,分析此结果对红色 LED 灯任务的时间性能影响。

**4. 实验 11.4**

图 3.11 中显示了 LED 灯何时闪烁,可以看到蓝色 LED 灯的闪烁变化。然而,我们实际上无法从闪烁结果判断蓝色 LED 灯是在绿色或红色 LED 灯任务上下文中闪烁。因此,现在修改代码,以显示蓝色 LED 灯运行在哪个任务上下文中。

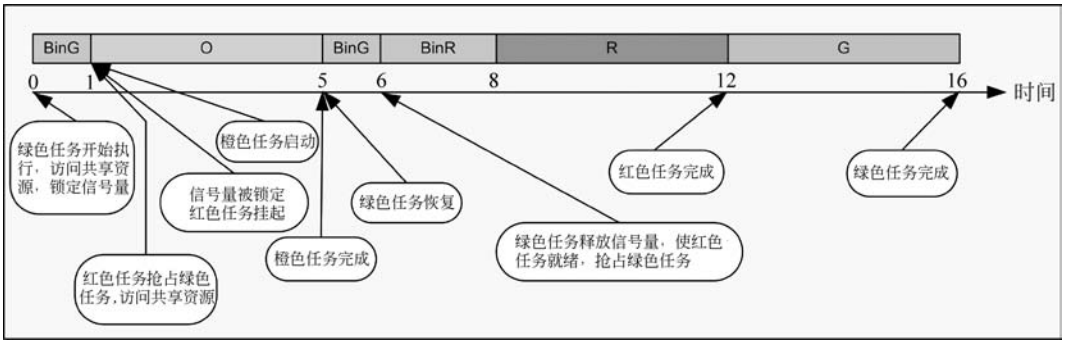


图 3.11　实验 11.3 的 LED 灯闪烁时序

当红色任务启动时，它抢占绿色任务，绿色任务会加载到就绪队列中。抢占发生时，蓝色 LED 灯可能为点亮或熄灭状态，取决于抢占发生的时刻，该状态将一直持续到红色任务再次执行。

### 3.7.4　实验讨论及回顾

通过前面的实验，我们可以看到：

(1) 在实验 11.1 和实验 11.2 中，红色任务完成所用的时间(从开始运行到完成)为 6s。

(2) 在实验 11.3 中，红色任务从开始运行($t=1$)到完成($t=12$)用时 11s。

(3) 在三个任务设计中，绿色任务的完成时间为：

实验 11.1，执行一次需 6s，运行完成需 16s。

实验 11.3，作为三个非连续执行周期完成，运行完成需 16s。

(4) 实验 11.3 清楚地显示了在时间 1～6s 范围内执行较低优先级任务(橙色任务)，即使红色(最高优先级)任务在 $t=1$s 时已经激活。这是典型的优先级反转现象。很显然，这意味着在 1～6s 范围内，红色任务不能处于就绪状态。

你应该能够推断出：

(1) 互斥和任务切换之间没有联系。系统调度策略确定何时以及为什么就绪/调度任务；共享资源争用取决于任务代码及任务何时使用这些资源。

(2) 共享项是被动代码单元(对象)，仅在任务调用时执行。因此，其执行过程中可以被视为调用任务的一部分。

(3) 多任务设计中的所有对象都是共享单元。

(4) 共享资源的使用通常会影响多任务设计的时间行为。

(5) 我们只能准确预测简单多任务系统的时间行为(确定性)。

(6) 由周期和非周期任务组成的多任务系统的时间行为几乎是不可预测或非确定的。

(7) 增加多任务设计中的任务和对象数量，会增大任务行为和响应能力的不确定性。

(8) 对于快速硬实时系统，我们应尽量减少任务和对象数量。

## 3.8 实验 12 使用优先级继承机制消除优先级反转

实验目的：演示使用优先级继承技术消除任务的优先级反转问题。

### 3.8.1 实验介绍

通过前面的实验了解到，任务不能同时访问共享资源，对共享资源的使用需以互斥的方式完成。在实验中，我们已使用过信号量保护共享资源的方法。如果两个任务争用一个资源，则仅一个任务可以使用它，另一个任务将挂起在信号量等待队列中。

不幸的是，任务挂起也破坏了其执行过程和运行时间(为安全性和稳健性付出的代价)，这是不可避免的，所以在设计系统时，必须考虑潜在的干扰。幸运的是，在这样的情况下，我们通常可以计算最坏情况的性能不可预测性的上限。然而，当任务优先级反转发生时，事情就不那么顺利。正如实验 11 中所看到的，反转会进一步破坏任务执行模式和时间。此外，这种破坏很难预测，因为它完全取决于系统的运行时状态。因此，为了生产可靠和稳健的软件系统，这个问题必须消除。

*Real-Time Operating System Book 1—The Theory* 一书从原理的角度描述了优先级继承技术如何防止优先级反转，现在，我们将通过实验了解如何在实践中实现此操作。

### 3.8.2 问题概述

FreeRTOS 的互斥信号量构造中提供了优先级继承机制，优先级继承在互斥信号量调用时自动实现，不需要任何其他编程激活。互斥信号量的符号及其在受保护对象中的应用如图 3.12 所示。

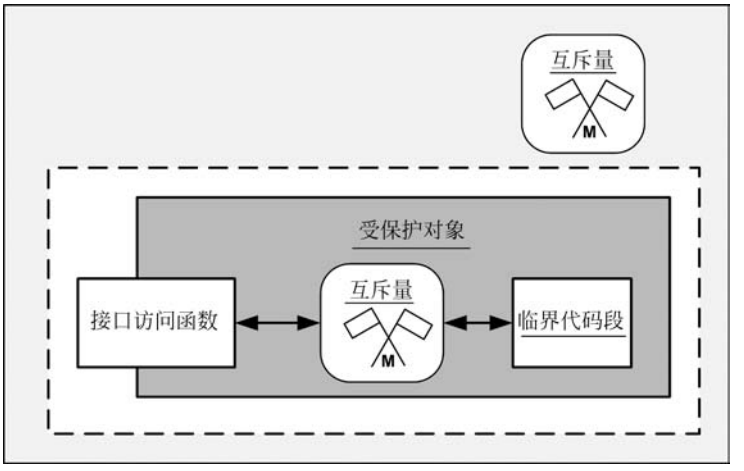


图 3.12 受保护对象中的互斥信号量应用及表示符号

在本实验中，我们将重复实验 11.4 的工作，使用互斥信号量代替信号量，如图 3.13 所示。互斥信号量的 API 及其使用可以参考实验 9。并且，在代码运行之前，预测 LED 灯闪烁行为。

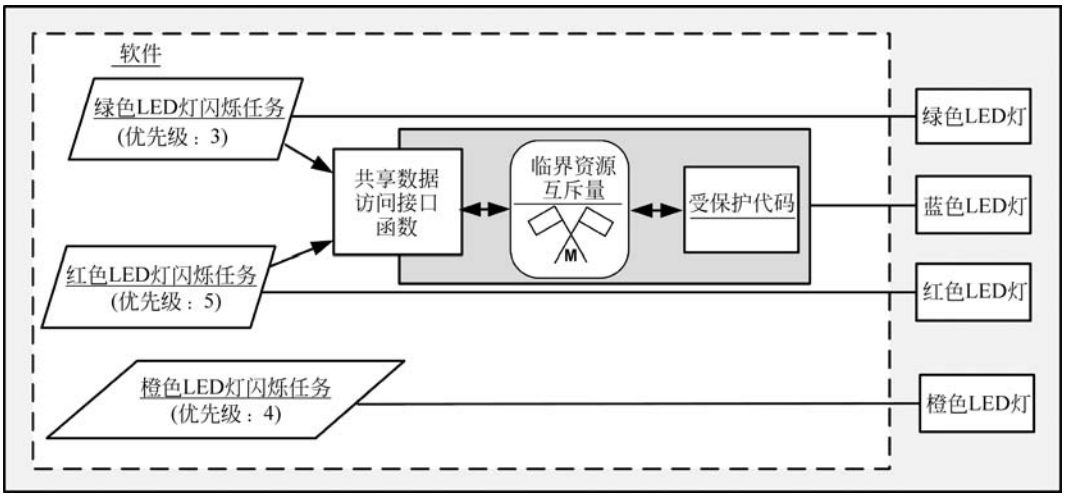


图 3.13　系统任务框图

更新实验 11 的程序，将信号量替换为互斥信号量，运行系统。如果操作正确，你将看到如图 3.14 所示的 LED 灯闪烁模式。

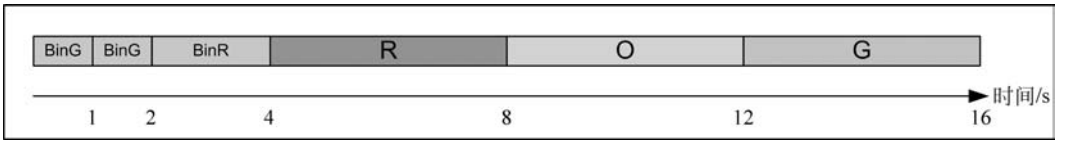


图 3.14　实验 12 的 LED 灯闪烁时序

参考图 3.14，准确了解目标系统中发生的情况。

### 3.8.3　实验回顾

比较图 3.11(无优先级继承)和图 3.14(使用优先级继承)可以看到，如果没有优先级继承，红色 LED 灯任务将在 $t=12$s 时完成；具有优先级继承机制时，任务在 $t=8$s 时刻完成。鉴于我们需要控制对共享资源的访问，所以一次只有一个任务能使用它，是我们可以做到的最好结果。

鉴于此，你应该始终使用互斥信号量而非信号量作为访问控制(互斥)机制。稍后，我们将了解在多任务设计中如何有效地利用信号量。