# 基于小熊派IoT开发套件的LiteOS内核实战-互斥锁

# 1. LiteOS的互斥锁

## 1.1. 互斥锁

在多任务环境下，往往存在多个任务竞争同一共享资源的应用场景，互斥锁可被用于对共享资源的保护从而实现独占式访问。互斥锁(mutex)又称互斥型信号量，是一种特殊的二值信号量，用于实现对共享资源的独占式处理。另外，Huawei LiteOS提供的互斥锁通过优先级继承算法，解决了优先级翻转问题。

## 1.2. 互斥锁的使用方式

多任务环境下会存在多个任务访问同一公共资源的场景，而有些公共资源是非共享的，需要任务进行独占式处理。

互斥锁怎样来避免这种冲突呢？

在任意时刻，互斥锁的状态只有两种：开锁和闭锁。

当有任务持有时，互斥锁处于闭锁状态，这个任务获得该互斥锁的所有权。当该任务释放它时，该互斥锁被开锁，任务失去该互斥锁的所有权。当一个任务持有互斥锁时，其他任务将不能再对该互斥锁进行开锁或持有。

那么，当一个互斥锁为加锁状态时，此时其他任务如果想访问这个公共资源则会被阻塞，直到互斥锁被持有该锁的任务释放后，其他任务才能重新访问该公共资源，此时互斥锁再次上锁，如此确保同一时刻只有一个任务正在访问这个公共资源，保证了公共资源操作的完整性。

## 1.3. 互斥锁的使用场景

互斥锁可以提供任务之间的互斥机制，用来防止两个任务在同一时刻访问相同的共享资源。

除此之外，互斥锁还可以被用于防止多任务同步时造成优先级翻转的问题。

# 2. 互斥锁API

Huawei LiteOS 系统中的互斥锁模块为用户提供创建/删除互斥锁、获取/释放互斥锁的功能。

Huawei LiteOS 系统中提供的互斥锁 API 都是以 LOS 开头，但是这些 API 使用起来比较复杂，所以本文中我们使用 Huawei IoT Link SDK 提供的统一API接口进行实验，这些接口底层已经使用 LiteOS 提供的API实现，对用户而言更为简洁，API列表如下：

osal的api接口声明在中，使用相关的接口需要包含该头文件，关于函数的详细参数请参考该头文件的声明。

相关的接口定义在osal.c中，基于LiteOS的接口实现在 liteos\_imp.c文件中：

接口名 功能描述

osal\_mutex\_create 创建互斥锁

osal\_mutex\_del 删除互斥锁

osal\_mutex\_lock 获取互斥锁(上锁)

osal\_mutex\_unlock 释放互斥锁(解锁)

## 2.1. osal\_mutex\_create

osal\_mutex\_create接口用于创建一个互斥锁，其接口原型如下：

bool\_t osal\_mutex\_create(osal\_mutex\_t \*mutex)

{

bool\_t ret = false;

if((NULL != s\_os\_cb) &&(NULL != s\_os\_cb->ops) &&(NULL != s\_os\_cb->ops->mutex\_create))

{

ret = s\_os\_cb->ops->mutex\_create(mutex);

}

return ret;

}

该接口的参数说明如下表：

参数 描述

mutex 互斥锁索引ID的地址

返回值 false - 创建失败

返回值 true - 创建成功

## 2.2. osal\_mutex\_del

osal\_mutex\_del接口用于删除一个互斥锁，其接口原型如下：

bool\_t osal\_mutex\_del(osal\_mutex\_t mutex)

{

bool\_t ret = false;

if((NULL != s\_os\_cb) &&(NULL != s\_os\_cb->ops) &&(NULL != s\_os\_cb->ops->mutex\_del))

{

ret = s\_os\_cb->ops->mutex\_del(mutex);

}

return ret;

}

该接口的参数说明如下表：

参数 描述

mutex 互斥锁索引ID

返回值 false - 删除失败

返回值 true - 删除成功

## 2.3. osal\_mutex\_lock

osal\_mutex\_lock接口用于获取一个互斥锁，其接口原型如下：

bool\_t osal\_mutex\_lock(osal\_mutex\_t mutex)

{

bool\_t ret = false;

if((NULL != s\_os\_cb) &&(NULL != s\_os\_cb->ops) &&(NULL != s\_os\_cb->ops->mutex\_lock))

{

ret = s\_os\_cb->ops->mutex\_lock(mutex);

}

return ret;

}

参数 描述

mutex 互斥锁索引ID

返回值 false - 申请失败

返回值 true - 申请成功

## 2.4. osal\_mutex\_unlock

osal\_mutex\_unlock接口用于释放一个互斥锁，如果有任务阻塞等待该互斥锁，则唤醒最早被阻塞的任务，该任务进入就绪态，并进行调度。

其接口原型如下：

bool\_t osal\_mutex\_unlock(osal\_mutex\_t mutex)

{

bool\_t ret = false;

if((NULL != s\_os\_cb) &&(NULL != s\_os\_cb->ops) &&(NULL != s\_os\_cb->ops->mutex\_unlock))

{

ret = s\_os\_cb->ops->mutex\_unlock(mutex);

}

return ret;

}

该接口的参数说明如下表：

参数 描述

mutex 互斥锁索引ID

返回值 false - 释放失败

返回值 true - 释放成功

# 3. 动手实验 —— 使用互斥锁进行资源保护

实验内容

本实验中将创建两个任务，一个低优先级任务task1，一个高优先级任务task2，两个任务之间依次对共享资源上锁、操作、解锁，在串口终端中观察两个任务的运行情况。

实验代码

首先打开上一篇使用的 HelloWorld 工程，基于此工程进行实验。

在Demo文件夹右击，新建文件夹osal\_kernel\_demo用于存放内核的实验文件（如果已有请忽略这一步）。

接下来在此文件夹中新建一个实验文件 osal\_mutex\_demo.c，开始编写代码：

/\* 使用osal接口需要包含该头文件 \*/

#include <osal.h>

/\* 任务优先级宏定义（shell任务的优先级为10） \*/

#define USER\_TASK1\_PRI 12 //低优先级

#define USER\_TASK2\_PRI 11 //高优先级

/\* 共享资源 \*/

uint32\_t public\_value = 0;

/\* 互斥锁索引ID \*/

osal\_mutex\_t public\_value\_mutex;

/\* 任务task1入口函数 \*/

static int user\_task1\_entry()

{

while(1)

{

/\* 尝试获取互斥锁 \*/

if(true == osal\_mutex\_lock(public\_value\_mutex))

{

/\* 获取到互斥锁，对共享资源进行操作 \*/

printf("\r\ntask1: lock a mutex.\r\n");

public\_value += 10;

printf("task1: public\_value = %ld.\r\n", public\_value);

/\* 对共享资源操作完毕，释放互斥锁 \*/

printf("task1: unlock a mutex.\r\n\r\n");

osal\_mutex\_unlock(public\_value\_mutex);

/\* 满足条件则结束任务 \*/

if(public\_value > 100)

break;

}

}

/\* while(1)会执行结束，所以需要返回值 \*/

return 0;

}

/\* 任务task2入口函数 \*/

static int user\_task2\_entry()

{

while (1)

{

/\* 尝试获取互斥锁 \*/

if(true == osal\_mutex\_lock(public\_value\_mutex))

{

/\* 获取到互斥锁，对共享资源进行操作 \*/

printf("\r\ntask2: lock a mutex.\r\n");

public\_value += 5;

printf("task2: public\_value = %ld.\r\n", public\_value);

/\* 对共享资源操作完毕，释放互斥锁 \*/

printf("task2: unlock a mutex.\r\n\r\n");

osal\_mutex\_unlock(public\_value\_mutex);

/\* 满足条件则结束任务 \*/

if(public\_value > 90)

break;

/\* 优先级较高，需要挂起一下，让task1获取到互斥锁，否则task2再次上锁，形成死锁 \*/

osal\_task\_sleep(10);

}

}

/\* while(1)会执行结束，所以需要返回值 \*/

return 0;

}

/\* 标准demo启动函数，函数名不要修改，否则会影响下一步实验 \*/

int standard\_app\_demo\_main()

{

/\* 创建互斥锁public\_value\_mutex \*/

osal\_mutex\_create(&public\_value\_mutex);

/\* 创建任务task1 \*/

osal\_task\_create("user\_task1",user\_task1\_entry,NULL,0x400,NULL,USER\_TASK1\_PRI);

/\* 创建任务task2 \*/

osal\_task\_create("user\_task2",user\_task2\_entry,NULL,0x400,NULL,USER\_TASK2\_PRI);

return 0;

}

编写完成之后，要将我们编写的 osal\_mutex\_demo.c文件添加到makefile中，加入整个工程的编译：

这里有个较为简单的方法，直接修改Demo文件夹下的user\_demo.mk配置文件，添加如下代码：

#example for osal\_mutex\_demo

ifeq ($(CONFIG\_USER\_DEMO), "osal\_mutex\_demo")

user\_demo\_src = ${wildcard $(TOP\_DIR)/targets/STM32L431\_BearPi/Demos/osal\_kernel\_demo/osal\_mutex\_demo.c}

endif

添加位置如图：

这段代码的意思是：

如果 CONFIG\_USER\_DEMO 宏定义的值是osal\_mutex\_demo，则将osal\_mutex\_demo.c文件加入到makefile中进行编译。

那么，如何配置 CONFIG\_USER\_DEMO 宏定义呢？在工程根目录下的.sdkconfig文件中的末尾即可配置：

因为我们修改了mk配置文件，所以点击重新编译按钮进行编译，编译完成后点击下载按钮烧录程序。

实验现象

程序烧录之后，即可看到程序已经开始运行，在串口终端中可看到实验的输出内容：

\*\*Link\*\*main:V1.2.1 AT 11:30:59 ON Nov 28 2019

WELCOME TO IOT\_LINK SHELL

LiteOS:/>

task2: lock a mutex.

task2: public\_value = 5.

task2: unlock a mutex.

task1: lock a mutex.

task1: public\_value = 15.

task1: unlock a mutex.

task1: lock a mutex.

task1: public\_value = 25.

task1: unlock a mutex.

task2: lock a mutex.

task2: public\_value = 30.

task2: unlock a mutex.

task1: lock a mutex.

task1: public\_value = 40.

task1: unlock a mutex.

task1: lock a mutex.

task1: public\_value = 50.

task1: unlock a mutex.

task2: lock a mutex.

task2: public\_value = 55.

task2: unlock a mutex.

task1: lock a mutex.

task1: public\_value = 65.

task1: unlock a mutex.

task1: lock a mutex.

task1: public\_value = 75.

task1: unlock a mutex.

task2: lock a mutex.

task2: public\_value = 80.

task2: unlock a mutex.

task1: lock a mutex.

task1: public\_value = 90.

task1: unlock a mutex.

task1: lock a mutex.

task1: public\_value = 100.

task1: unlock a mutex.

task2: lock a mutex.

task2: public\_value = 105.

task2: unlock a mutex.

可以看到，系统启动后，首先打印版本号，串口shell的优先级为10，最先打印shell信息，接下来task1先创建，但是优先级较低，所以后创建的task2抢占执行，task2获取到互斥锁，对共享资源进行操作，操作完毕解锁，然后主动挂起，task1获取到互斥锁，对共享资源进行另一个操作，操作完毕解锁，在task1操作的时候，task2早已挂起完毕，但是获取不到互斥锁，所以挂起等待，在task1解锁后，堵塞的task2被唤醒开始执行。