一、实验名称

信号量与同步

二、实验目的

1. 理解信号量的基本概念，以及其在进程同步中的作用
2. 学习使用信号量来实现进程之间的同步
3. 理解并解决多种并发时可能出现的问题

三、实验任务

信号量结构初始化

1.新建 lab7/src/include/prt_sem_external.h 头文件

```
// 防止头文件重复包含
#ifndef PRT_SEM_EXTERNAL_H
#define PRT_SEM_EXTERNAL_H

// 包含基础信号量定义头文件
#include "prt_sem.h"
// 包含任务相关的外部接口定义
#include "prt_task_external.h"
// 如果定义了POSIX选项，则包含POSIX信号量类型定义
#if defined(OS_OPTION_POSIX)
#include "bits/semaphore_types.h"
#endif

// 信号量状态定义
#define OS_SEM_UNUSED 0       // 信号量未被使用
#define OS_SEM_USED   1       // 信号量正在被使用

// 信号量协议类型定义
#define SEM_PROTOCOL_PRIO_INHERIT 1  // 优先级继承协议

// 位宽定义（用于信号量类型编码）
#define SEM_TYPE_BIT_WIDTH         0x4U    // 类型字段占4位
#define SEM_PROTOCOL_BIT_WIDTH     0x8U    // 协议字段占8位

// 信号量标志定义
```

```
#define OS_SEM_WITH_LOCK_FLAG    1        // 使用锁机制的标志
#define OS_SEM_WITHOUT_LOCK_FLAG 0        // 不使用锁机制的标志

// POSIX信号量最大名称长度
#define MAX_POSIX_SEMAPHORE_NAME_LEN    31

// 宏定义：从链表组件获取信号量控制块指针
#define GET_SEM_LIST(ptr) LIST_COMPONENT(ptr, struct TagSemCb, semList)
// 宏定义：通过信号量ID获取信号量控制块指针
#define GET_SEM(semid) (((struct TagSemCb *)g_allSem) + (semid))
// 宏定义：通过信号量ID获取关联任务控制块指针
#define GET_SEM_TSK(semid) (((SEM_TSK_S *)g_semTsk) + (semid))
// 宏定义：通过任务ID获取关联的信号量指针
#define GET_TSK_SEM(tskid) (((TSK_SEM_S *)g_tskSem) + (tskid))

// 宏定义：提取信号量类型的基础类型（低4位）
#define GET_SEM_TYPE(semType) (U32)((semType) & ((1U << SEM_TYPE_BIT_WIDTH)
- 1))
// 宏定义：提取信号量互斥体类型（中间4位）
#define GET_MUTEX_TYPE(semType) (U32)(((semType) >> SEM_TYPE_BIT_WIDTH) &
((1U << SEM_TYPE_BIT_WIDTH) - 1))
// 宏定义：提取信号量协议类型（高8位）
#define GET_SEM_PROTOCOL(semType) (U32)((semType) >> SEM_PROTOCOL_BIT_WIDTH)

// 信号量控制块结构定义
struct TagSemCb {
    U16 semStat;          // 信号量状态（OS_SEM_UNUSED/OS_SEM_USED）
    U16 semId;            // 内核信号量索引ID
#if defined(OS_OPTION_SEM_RECUR_PV)
    U32 recurCount;       // 递归计数（仅用于可重入互斥信号量）
#endif
    U32 semCount;         // 信号量计数值
    struct TagListObject semList;    // 阻塞在该信号量的任务链表
    struct TagListObject semBList;   // 互斥信号量持有任务链表（仅计数型无效）

    U32 semOwner;         // 当前持有该信号量的任务ID
    enum SemMode semMode; // 信号量操作模式（如阻塞/非阻塞）
    U32 semType;          // 信号量类型编码（类型+协议）

#if defined(OS_OPTION_POSIX)
    char name[MAX_POSIX_SEMAPHORE_NAME_LEN + 1]; // POSIX信号量名称
    sem_t handle;                                // POSIX信号量句柄
#endif
```

```
};

// 全局变量声明
extern U16 g_maxSem;                    // 系统支持的最大信号量数量
extern struct TagSemCb *g_allSem;    // 全局信号量控制块数组指针

// 函数声明
extern U32 OsSemCreate(U32 count, U32 semType, enum SemMode semMode,
SemHandle *semHandle, U32 cookie);
// 功能：创建信号量
// 参数：count-初始计数值；semType-信号量类型；semMode-操作模式；
//        semHandle-输出参数返回句柄；cookie-上下文信息
// 返回：操作结果状态码

extern bool OsSemBusy(SemHandle semHandle);
// 功能：检查信号量是否正被占用
// 参数：semHandle-信号量句柄
// 返回：true表示被占用，false表示可用
#endif /* PRT_SEM_EXTERNAL_H */
```

2.新建 src/kernel/sem/prt_sem_init.c 文件

```
#include "prt_sem_external.h"   // 信号量相关外部接口声明
#include "os_attr_armv8_external.h" // ARMv8架构属性相关定义
#include "os_cpu_armv8_external.h"  // ARMv8 CPU相关定义

// 全局变量定义（BSS段，系统初始化时自动清零）
OS_SEC_BSS struct TagListObject g_unusedSemList;  // 未使用信号量链表
OS_SEC_BSS struct TagSemCb *g_allSem;             // 所有信号量控制块数组指针

// 外部内存分配函数声明
extern void *OsMemAllocAlign(U32 mid, U8 ptNo, U32 size, U8 alignPow);

/*
* 函数名：OsSemInit
* 功能：信号量模块初始化
* 参数：无
* 返回：OS_OK 成功，其他错误码
*/
OS_SEC_L4_TEXT U32 OsSemInit(void)
{
    struct TagSemCb *semNode = NULL;  // 临时信号量控制块指针
    U32 idx;                          // 循环索引
```

```
    U32 ret = OS_OK;                    // 返回值初始化

    /* 分配信号量控制块内存池 */
    g_allSem = (struct TagSemCb *)OsMemAllocAlign(
        (U32)OS_MID_SEM,    // 内存管理标识
        0,                  // 物理页号（不适用）
        4096,               // 分配4KB内存
        OS_SEM_ADDR_ALLOC_ALIGN); // 地址对齐要求

    if (g_allSem == NULL) {          // 内存分配失败处理
        return OS_ERRNO_SEM_NO_MEMORY;
    }

    // 计算最大信号量数量（假设每个控制块1字节，实际需按结构体大小计算）
    g_maxSem = 4096 / sizeof(struct TagSemCb);

    // 内存初始化（清零）
    char *cg_allSem = (char *)g_allSem;
    for(int i = 0; i < 4096; i++) {
        cg_allSem[i] = 0;
    }

    // 初始化未使用链表
    INIT_LIST_OBJECT(&g_unusedSemList);

    // 遍历所有信号量控制块并加入空闲链表
    for (idx = 0; idx < g_maxSem; idx++) {
        semNode = ((struct TagSemCb *)g_allSem) + idx;  // 计算当前控制块地址
        semNode->semId = (U16)idx;                      // 设置信号量ID
        // 将控制块加入空闲链表尾部
        ListTailAdd(&semNode->semList, &g_unusedSemList);
    }

    return ret;
}

/*
* 函数名：OsSemCreate
* 功能：创建信号量
* 参数：
*   count     - 初始计数值
*   semType   - 信号量类型（二进制/计数）
*   semMode   - 信号量模式
```

```
*   semHandle - 输出参数，返回信号量句柄
*   cookie    - 保留参数
* 返回：OS_OK 成功，其他错误码
*/
OS_SEC_L4_TEXT U32 OsSemCreate(U32 count, U32 semType, enum SemMode semMode,
                               SemHandle *semHandle, U32 cookie)
{
    uintptr_t intSave;                      // 中断保存变量
    struct TagSemCb *semCreated = NULL;     // 新创建的信号量控制块
    struct TagListObject *unusedSem = NULL;// 空闲链表节点指针
    (void)cookie;                           // 显式标记未使用参数

    // 参数校验
    if (semHandle == NULL) {
        return OS_ERRNO_SEM_PTR_NULL;
    }

    intSave = OsIntLock();                  // 关中断

    // 检查空闲链表是否已满
    if (ListEmpty(&g_unusedSemList)) {
        OsIntRestore(intSave);              // 恢复中断
        return OS_ERRNO_SEM_ALL_BUSY;
    }

    // 从空闲链表头部获取节点
    unusedSem = OS_LIST_FIRST(&(g_unusedSemList));
    ListDelete(unusedSem);                  // 从链表中移除

    // 转换为信号量控制块指针
    semCreated = (GET_SEM_LIST(unusedSem));

    // 初始化控制块成员
    semCreated->semCount = count;           // 设置初始计数值
    semCreated->semStat = OS_SEM_USED;      // 标记为已使用
    semCreated->semMode = semMode;          // 设置信号量模式
    semCreated->semType = semType;          // 设置信号量类型
    semCreated->semOwner = OS_INVALID_OWNER_ID; // 清除所有者

    // 类型特异性初始化
    if (GET_SEM_TYPE(semType) == SEM_TYPE_BIN) {  // 二进制信号量
        INIT_LIST_OBJECT(&semCreated->semBList);  // 初始化等待队列
#if defined(OS_OPTION_SEM_RECUR_PV)
```

```
        // 递归互斥锁支持
        if (GET_MUTEX_TYPE(semType) == PTHREAD_MUTEX_RECURSIVE) {
            semCreated->recurCount = 0;  // 初始化递归计数器
        }
#endif
    }

    // 初始化通用等待队列
    INIT_LIST_OBJECT(&semCreated->semList);

    *semHandle = (SemHandle)semCreated->semId; // 返回句柄（即信号量ID）

    OsIntRestore(intSave);  // 恢复中断
    return OS_OK;
}

/*
* 函数名：PRT_SemCreate
* 功能：创建计数型信号量（公有接口）
* 参数：
*   count      - 初始计数值
*   semHandle  - 输出参数，返回信号量句柄
* 返回：同OsSemCreate
*/
OS_SEC_L4_TEXT U32 PRT_SemCreate(U32 count, SemHandle *semHandle)
{
    U32 ret;

    // 参数有效性检查（最大值保护）
    if (count > OS_SEM_COUNT_MAX) {
        return OS_ERRNO_SEM_OVERFLOW;
    }

    // 调用底层创建函数，指定为计数型信号量
    ret = OsSemCreate(
        count,
        SEM_TYPE_COUNT,       // 信号量类型：计数型
        SEM_MODE_FIFO,        // 调度模式：FIFO
        semHandle,
        (U32)(uintptr_t)semHandle  //
    );

    return ret;
```

```
}
```

3.在 src/bsp/os_cpu_armv8_external.h 加入 定义

```
#define OS_SEM_ADDR_ALLOC_ALIGN 2U //为信号量相关的内存分配指定对齐方式为2的幂次方对齐，即2^2=4字节
```

4.新建 src/kernel/sem/prt_sem.c 文件

```
#include "prt_sem_external.h"
#include "prt_asm_cpu_external.h"
#include "os_attr_armv8_external.h"
#include "os_cpu_armv8_external.h"

/* 核内信号量最大个数 */
OS_SEC_BSS U16 g_maxSem;

/*
 * 描述：信号量发布操作的错误检查
 * 参数：semPosted - 信号量控制块指针
 *       semHandle - 信号量句柄
 * 返回：错误码
 */
OS_SEC_ALW_INLINE INLINE U32 OsSemPostErrorCheck(struct TagSemCb *semPosted,
SemHandle semHandle)
{
    (void)semHandle;  // 标记未使用的参数
    /* 检查信号量是否处于未使用状态 */
    if (semPosted->semStat == OS_SEM_UNUSED) {
        return OS_ERRNO_SEM_INVALID;  // 返回无效信号量错误
    }

    /* 检查计数型信号量溢出 */
    if ((semPosted)->semCount >= OS_SEM_COUNT_MAX) {
        return OS_ERRNO_SEM_OVERFLOW;  // 返回计数器溢出错误
    }

    return OS_OK;  // 验证通过返回成功
}

/*
 * 描述：将当前任务挂载到信号量等待链表
```

```
 * 参数：semPended - 被等待的信号量控制块
 *       timeOut   - 超时时间
 */
OS_SEC_L0_TEXT void OsSemPendListPut(struct TagSemCb *semPended, U32
timeOut)
{
    struct TagTskCb *curTskCb = NULL;
    struct TagTskCb *runTsk = RUNNING_TASK;  // 获取当前运行任务
    struct TagListObject *pendObj = &runTsk->pendList;  // 任务等待链表头

    OsTskReadyDel((struct TagTskCb *)runTsk);  // 将任务从就绪队列移除

    runTsk->taskSem = (void *)semPended;  // 记录任务关联的信号量

    TSK_STATUS_SET(runTsk, OS_TSK_PEND);  // 设置任务为等待状态

    /* 根据信号量模式插入等待队列 */
    if (semPended->semMode == SEM_MODE_PRIOR) {  // 优先级模式
        /* 遍历查找第一个优先级低于当前任务的节点 */
        LIST_FOR_EACH(curTskCb, &semPended->semList, struct TagTskCb,
pendList) {
            if (curTskCb->priority > runTsk->priority) {
                ListTailAdd(pendObj, &curTskCb->pendList);  // 插入到该节点前
                return;
            }
        }
    }

    /* FIFO模式或优先级模式下无更高优先级任务 */
    ListTailAdd(pendObj, &semPended->semList);  // 添加到等待队列尾部
}

/*
 * 描述：从信号量等待链表中取出最高优先级任务
 * 参数：semPended - 被等待的信号量控制块
 * 返回：被唤醒的任务控制块指针
 */
OS_SEC_L0_TEXT struct TagTskCb *OsSemPendListGet(struct TagSemCb *semPended)
{
    struct TagTskCb *taskCb = GET_TCB_PEND(LIST_FIRST(&(semPended-
>semList)));  // 获取队列首任务

    ListDelete(LIST_FIRST(&(semPended->semList)));  // 从等待队列移除该节点
```

```
    /* 清除定时等待标志 */
    if (TSK_STATUS_TST(taskCb, OS_TSK_TIMEOUT)) {
        OS_TSK_DELAY_LOCKED_DETACH(taskCb);  // 分离定时等待任务
    }

    TSK_STATUS_CLEAR(taskCb, OS_TSK_TIMEOUT | OS_TSK_PEND);  // 清除等待状态
    taskCb->taskSem = NULL;  // 解除任务与信号量的关联

    /* 若任务未被挂起则加入就绪队列 */
    if (!TSK_STATUS_TST(taskCb, OS_TSK_SUSPEND)) {
        OsTskReadyAddBgd(taskCb);  // 后台加入就绪队列
    }

    return taskCb;
}

/*
 * 描述：检查信号量等待参数有效性
 * 参数：timeout - 等待超时时间
 * 返回：错误码
 */
OS_SEC_L0_TEXT U32 OsSemPendParaCheck(U32 timeout)
{
    if (timeout == 0) {  // 不允许零超时等待
        return OS_ERRNO_SEM_UNAVAILABLE;
    }

    if (OS_TASK_LOCK_DATA != 0) {  // 检查任务锁状态
        return OS_ERRNO_SEM_PEND_IN_LOCK;  // 在任务锁定的情况下不允许等待
    }

    return OS_OK;
}

/*
 * 描述：判断是否需要触发调度
 * 参数：semPended - 被等待的信号量控制块
 *       runTsk    - 当前运行任务
 * 返回：true-无需调度/false-需要调度
 */
OS_SEC_L0_TEXT bool OsSemPendNotNeedSche(struct TagSemCb *semPended, struct
TagTskCb *runTsk)
```

```
{
    if (semPended->semCount > 0) {  // 信号量可用
        semPended->semCount--;      // 减少计数
        semPended->semOwner = runTsk->taskPid;  // 更新信号量持有者
        return true;
    }
    return false;
}

/*
 * 描述：信号量P操作（等待）
 * 参数：semHandle - 信号量句柄
 *       timeout   - 超时时间
 * 返回：错误码
 */
OS_SEC_L0_TEXT U32 PRT_SemPend(SemHandle semHandle, U32 timeout)
{
    uintptr_t intSave;
    U32 ret;
    struct TagTskCb *runTsk = NULL;
    struct TagSemCb *semPended = NULL;

    if (semHandle >= (SemHandle)g_maxSem) {  // 校验句柄有效性
        return OS_ERRNO_SEM_INVALID;
    }

    semPended = GET_SEM(semHandle);  // 获取信号量控制块
    intSave = OsIntLock();  // 关闭中断

    /* 检查信号量状态 */
    if (semPended->semStat == OS_SEM_UNUSED) {
        OsIntRestore(intSave);
        return OS_ERRNO_SEM_INVALID;
    }

    if (OS_INT_ACTIVE) {  // 中断上下文中不允许等待
        OsIntRestore(intSave);
        return OS_ERRNO_SEM_PEND_INTERR;
    }

    runTsk = (struct TagTskCb *)RUNNING_TASK;

    /* 尝试立即获取信号量 */
```

```
    if (OsSemPendNotNeedSche(semPended, runTsk)) {
        OsIntRestore(intSave);
        return OS_OK;
    }

    /* 参数有效性检查 */
    ret = OsSemPendParaCheck(timeout);
    if (ret != OS_OK) {
        OsIntRestore(intSave);
        return ret;
    }

    /* 将任务加入等待队列 */
    OsSemPendListPut(semPended, timeout);

    if (timeout != OS_WAIT_FOREVER) {
        OsIntRestore(intSave);
        return OS_ERRNO_SEM_FUNC_NOT_SUPPORT;  // 仅支持无限等待
    } else {
        OsTskScheduleFastPs(intSave);  // 触发任务调度
    }

    OsIntRestore(intSave);
    return OS_OK;
}

/*
 * 描述：信号量V操作（释放）
 * 参数：semHandle - 信号量句柄
 * 返回：错误码
 */
OS_SEC_L0_TEXT U32 PRT_SemPost(SemHandle semHandle)
{
    U32 ret;
    uintptr_t intSave;
    struct TagSemCb *semPosted = NULL;

    if (semHandle >= (SemHandle)g_maxSem) {  // 校验句柄有效性
        return OS_ERRNO_SEM_INVALID;
    }

    semPosted = GET_SEM(semHandle);
    intSave = OsIntLock();
```

```
    /* 执行发布前的错误检查 */
    ret = OsSemPostErrorCheck(semPosted, semHandle);
    if (ret != OS_OK) {
        OsIntRestore(intSave);
        return ret;
    }

    /* 判断发布操作是否无效 */
    if (OsSemPostIsInvalid(semPosted)) {
        OsIntRestore(intSave);
        return OS_OK;
    }

    /* 处理等待队列 */
    if (!ListEmpty(&semPosted->semList)) {
    // 存在等待任务时直接唤醒
        OsSemPostSchePre(semPosted);  // 激活等待任务
        OsTskScheduleFastPs(intSave); // 快速调度
    } else {
    // 无等待任务时增加计数器
        semPosted->semCount++;        // 增加信号量计数
        semPosted->semOwner = OS_INVALID_OWNER_ID;  // 清除所有者
    }

    OsIntRestore(intSave);
    return OS_OK;
}
```

5.src/include/prt_task_external.h 加入 OsTskReadyAddBgd()

```
// 更新任务状态为 OS_TASK_STATUS_READY，将任务插入就绪队列的双向链表
/*


* - 该函数为内联函数（OS_SEC_ALW_INLINE INLINE），直接嵌入调用位置以减少函数调用开销
* - 实际的任务队列操作由底层函数 OsTskReadyAdd 完成

*/
OS_SEC_ALW_INLINE INLINE void OsTskReadyAddBgd(struct TagTskCb *task)
{
```

```
    OsTskReadyAdd(task);
}
```

6.src/kernel/task/prt_task.c 加入 OsTskScheduleFastPs()

```
/*
 * 描述：如果快速切换后只有中断恢复，使用此接口
 */
OS_SEC_TEXT void OsTskScheduleFastPs(uintptr_t intSave)
{
    /* 查找并设置当前系统中最高优先级的就绪任务 */
    OsTskHighestSet();

    /* 检查当前运行态任务是否不是最高优先级任务，且全局任务锁未被占用 */
    if ((g_highestTask != RUNNING_TASK) && (g_uniTaskLock == 0)) {
        /* 设置统一调度请求标志位，通知系统需要进行任务切换 */
        UNI_FLAG |= OS_FLG_TSK_REQ;

        /* 仅当中断处于非激活状态（无硬件中断或时钟中断）时执行快速调度陷阱 */
        if (OS_INT_INACTIVE) {
            /* 执行快速上下文切换陷阱处理，传入当前中断保存状态 */
            OsTaskTrapFastPs(intSave);
        }
    }
}
```

7.src/bsp/os_cpu_armv8_external.h 加入 OsTaskTrapFastPs()

```
// 定义一个始终内联的安全级别函数，用于快速处理任务陷阱（Task Trap）
// 参数intSave用于保存中断上下文，具体类型uintptr_t适配指针操作
OS_SEC_ALW_INLINE INLINE void OsTaskTrapFastPs(uintptr_t intSave)
{
    // 显式忽略未使用的参数，消除编译器警告
    (void)intSave;

    // 调用核心任务陷阱处理函数，执行实际的任务状态保存与调度
    OsTaskTrap();
}
```

8.加入 src/include/prt_sem.h [下载]，该头文件主要是信号量相关的函数声明和宏定义。

提示：将新增文件加入构建系统

验证

```
#include "prt_typedef.h"

#include "prt_tick.h"

#include "prt_task.h"

#include "prt_sem.h"



extern U32 PRT_Printf(const char *format, ...);

extern void PRT_UartInit(void);

extern U32 OsActivate(void);

extern U32 OsTskInit(void);

extern U32 OsSemInit(void);



static SemHandle sem_sync;



void Test1TaskEntry()

{

    PRT_Printf("task 1 run ...\n");

    PRT_SemPost(sem_sync);

    U32 cnt = 5;

    while (cnt > 0) {

        // PRT_TaskDelay(200);
```

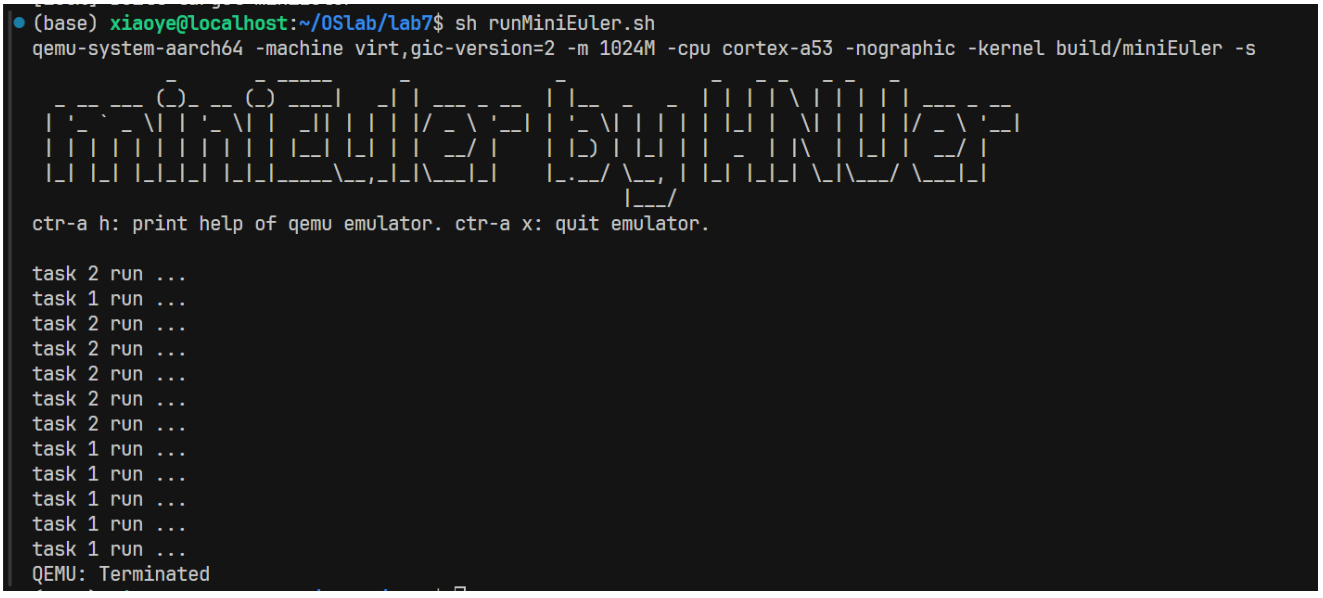
```
        PRT_Printf("task 1 run ...\n");
        cnt--;
    }
}

void Test2TaskEntry()
{
    PRT_Printf("task 2 run ...\n");

    PRT_SemPend(sem_sync, OS_WAIT_FOREVER);
    U32 cnt = 5;
    while (cnt > 0) {
        // PRT_TaskDelay(100);
        PRT_Printf("task 2 run ...\n");
        cnt--;
    }
}

S32 main(void)
{
    // 任务模块初始化
```

```
    OsTskInit();

    OsSemInit(); // 参见demos/ascend310b/config/prt_config.c 系统初始化注册表



    PRT_UartInit();



    PRT_Printf("            _       _ _____      _             _
_   _ _   _ _   _           \n");

    PRT_Printf("  _ __ ___ (_)_ __ (_) ____|   _| | ___ _ __  | |__  _   _
 | | | | \\ | | | | | ___ _ __ \n");

    PRT_Printf(" | '_ ` _ \\| | '_ \\| |  _|| | | | |/ _ \\ '__| | '_ \\| |
| | | |_| |  \\| | | | |/ _ \\ '__|\n");

    PRT_Printf(" | | | | | | | | | | | |__| |_| | |  __/ |    | |_) | |_| |
|  _  | |\\  | |_| |  __/ |   \n");

    PRT_Printf(" |_| |_| |_|_|_| |_|_|_____\\__,_|_|\\___|_|    |_.__/ \\__,
| |_| |_|_| \\_|\\___/ \\___|_|   \n");

    PRT_Printf("                                                    |___/
                        \n");



    PRT_Printf("ctr-a h: print help of qemu emulator. ctr-a x: quit
emulator.\n\n");



    U32 ret;

    ret = PRT_SemCreate(0, &sem_sync);

    if (ret != OS_OK) {

        PRT_Printf("failed to create synchronization sem\n");
```

```
        return 1;
    }

    struct TskInitParam param = {0};

    // task 1
    // param.stackAddr = 0;
    param.taskEntry = (TskEntryFunc)Test1TaskEntry;
    param.taskPrio = 35;
    // param.name = "Test1Task";
    param.stackSize = 0x1000; //固定4096，参见prt_task_init.c的
OsMemAllocAlign

    TskHandle tskHandle1;
    ret = PRT_TaskCreate(&tskHandle1, &param);
    if (ret) {
        return ret;
    }

    ret = PRT_TaskResume(tskHandle1);
    if (ret) {
        return ret;
```

```
    }



    // task 2

    // param.stackAddr = 0;

    param.taskEntry = (TskEntryFunc)Test2TaskEntry;

    param.taskPrio = 30;

    // param.name = "Test2Task";

    param.stackSize = 0x1000; //固定4096，参见prt_task_init.c的
OsMemAllocAlign



    TskHandle tskHandle2;

    ret = PRT_TaskCreate(&tskHandle2, &param);

    if (ret) {

        return ret;

    }



    ret = PRT_TaskResume(tskHandle2);

    if (ret) {

        return ret;

    }



    // 启动调度
```

```
    OsActivate();



    // while(1);

    return 0;



}
```



作业

作业1

各种并发问题模拟，至少3种。

竞态条件

```
static U32 shared_counter=0;//共享变量 临界区



void Test1TaskEntry()

{

    U32 cnt = 5;
```

```
    while (cnt > 0) {
        U32 temp=shared_counter;//task1获取临界区资源
        temp++;//加1
        PRT_Printf("Task 1 increments shared_counter to %u\n", temp);
        shared_counter=temp;//改写临界区资源
        cnt--;
    }
}

void Test2TaskEntry()
{
    U32 cnt = 5;
    while (cnt > 0) {
        U32 temp=shared_counter;//task2获取临界区资源
        temp++;//加1
        PRT_Printf("Task 2 increments shared_counter to %u\n", temp);
        shared_counter=temp;//改写临界区资源
        cnt--;
    }
}
```

竞态条件发生在多个任务同时访问和修改共享资源时，如果缺乏适当的同步机制，导致最终结果依赖于任务的执行顺序。换句话说，函数指令的原子性被破坏了。上述代码，

shared_counter 是一个共享资源，两个任务在没有同步机制的情况下修改它。两个任务可能会同时读取相同的值，分别增加 1，然后写回共享变量。这会导致一些更新丢失。
假设两个任务（Task1和Task2）同时执行，且系统调度导致以下交错执行时：

1. **初始状态**：`shared_counter = 0`
2. **Task1执行**：
   - 读取 `temp = shared_counter` → `temp = 0`
   - `temp++` → `temp = 1`
   - **未立即写回**，此时被系统调度暂停
3. **Task2执行**：
   - 读取 `temp = shared_counter` → `temp = 0`
   - `temp++` → `temp = 1`
   - 写回 `shared_counter = 1`
4. **Task1恢复执行**：
   - 写回 `shared_counter = 1`（覆盖Task2的更新）

最终结果：
经过一次循环后，`shared_counter`仅增加1，而非预期的2。

所以，我们可以联想到最终的错误输出可能是有多种情况的（当然也不排除它碰巧就是正确执行咯）。

解决策略：添加一个互斥锁来保持每次循环的原子性

```
static U32 shared_counter=0;

static SemHandle mutex=1;//二值信号量（互斥锁）



void Test1TaskEntry()

{

    U32 cnt = 5;

    while (cnt > 0) {

        PRT_SemPend(mutex,OS_WAIT_FOREVER);//

        U32 temp=shared_counter;

        temp++;
```

```
        PRT_Printf("Task 1 increments shared_counter to %u\n", temp);
        shared_counter=temp;
        PRT_SemPost(mutex);
        cnt--;
    }
}


void Test2TaskEntry()
{
    U32 cnt = 5;
    while (cnt > 0) {
        PRT_SemPend(mutex,OS_WAIT_FOREVER);
        U32 temp=shared_counter;
        temp++;
        PRT_Printf("Task 2 increments shared_counter to %u\n", temp);
        shared_counter=temp;
        PRT_SemPost(mutex);
        cnt--;
    }
}
```

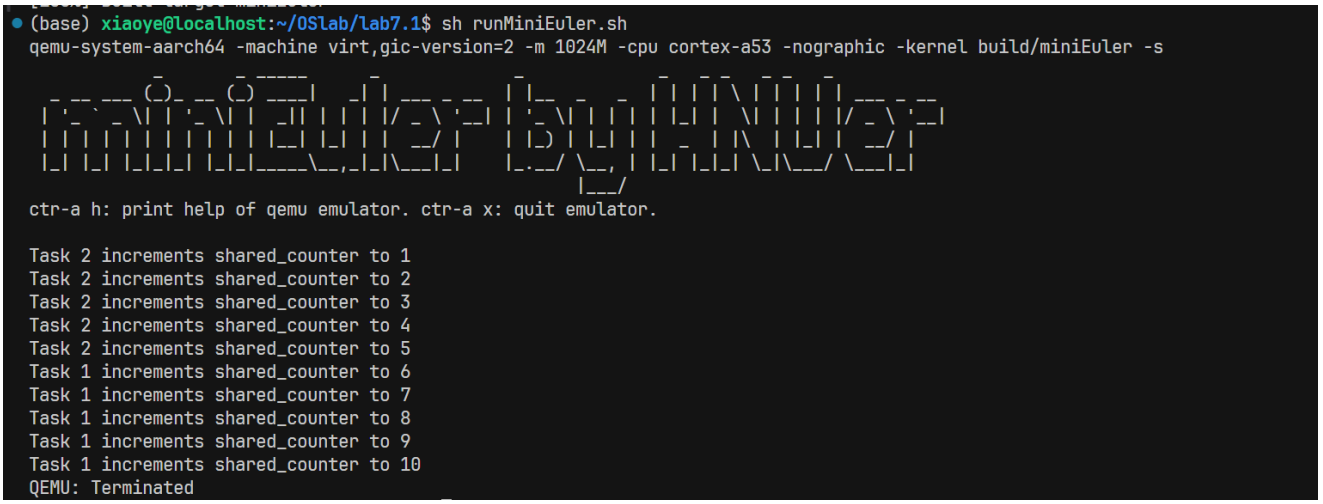
解释

二值信号量（互斥锁）初始化为1，表示资源可用。

`PRT_SemPend(mutex, OS_WAIT_FOREVER)`，尝试尝试获取信号量 `mutex`，若成功则进入临界区。如果信号量值大于0，递减信号量值，表示资源被占有，任务继续执行；如果等于0，任务被挂起，加入信号量等待序列。任务进入阻塞状态，系统调度其他任务运行。

`PRT_SemPost(mutex)`，释放信号量mutex，允许其他任务进入临界区。将信号量值+1，表示资源被释放。若有任务在等待队列中，系统唤醒优先级最高的任务，使其尝试获取信号量。

当我们每次循环都要先进行锁的获取，后续再执行本次循环的指令时，不会被其他任务打断，这样可以保证每次循环的原子性特点，确保结果输出正确。



死锁

```
static SemHandle sem1;//锁1

static SemHandle sem2;//锁2



void Test1TaskEntry()

{

    PRT_Printf("Task 1 trying to lock sem1 \n");

    PRT_SemPend(sem1,OS_WAIT_FOREVER);

    PRT_Printf("Task 1 locked sem1 \n");
```

```
    PRT_Printf("Task 1 trying to lock sem2 \n");
    PRT_SemPend(sem2,OS_WAIT_FOREVER);
    PRT_Printf("Task 1 locked sem2 \n");


    PRT_SemPost(sem2);
    PRT_SemPost(sem1);
}


void Test2TaskEntry()
{
    PRT_Printf("Task 2 trying to lock sem2 \n");
    PRT_SemPend(sem2,OS_WAIT_FOREVER);
    PRT_Printf("Task 2 locked sem2 \n");


    PRT_Printf("Task 2 trying to lock sem1 \n");
    PRT_SemPend(sem1,OS_WAIT_FOREVER);
    PRT_Printf("Task 2 locked sem1 \n");


    PRT_SemPost(sem1);
    PRT_SemPost(sem2);
}
```

上述代码，任务 1 先锁定 sem1，然后试图锁定 sem2；任务 2 先锁定 sem2，然后试图锁定 sem1。如果任务 1 和任务 2 同时运行，这将导致一个死锁情况，因为每个任务都在等待对方释放一个它已经持有的信号量。

解决策略：添加时间锁

```
#define TIMEOUT 1000



void Test1TaskEntry()

{

    while(1){

    PRT_Printf("Task 1 trying to lock sem1 \n");

    if(PRT_SemPend(sem1,TIMEOUT)!=OS_OK){

        PRT_Printf("Task 1 failed to lock sem1,retrying \n");

        continue;

    }

    PRT_Printf("Task 1 locked sem1 \n");

    PRT_Printf("Task 1 trying to lock sem2 \n");

    if(PRT_SemPend(sem2,TIMEOUT)!=OS_OK){

        PRT_Printf("Task 1 failed to lock sem2,releasing sem1 and retrying
\n");

        PRT_SemPost(sem1);//释放第一个锁

        continue;

    }

    PRT_Printf("Task 1 locked sem2 \n");
```

```
    PRT_SemPost(sem2);

    PRT_SemPost(sem1);

    break;

    }

}
```

解释说明

首先每个任务尝试获取第一个信号量（sem1 或 sem2），如果在限制时间内获取失败，则重试。如果成功获取第一个信号量，则尝试获取第二个信号量。如果在限制时间内获取第二个信号量失败，那就释放第一个信号量并重试。成功获取两个信号量后，进入临界区进行处理，处理完成后，释放两个信号量。如此就可以解决死锁的问题。
另外，还可以让任务一和任务二按照相同的顺序进行锁的获取，这样可以避免死锁问题，但是我认为效率不高，如果其中一个任务抢先占有资源，那么另一个任务的等待时间可能会较长一点。

饥饿问题

```
static SemHandle sem_sync;



void Test1TaskEntry()

{

    PRT_Printf("任务1执行 \n");

    PRT_SemPost(sem_sync);

    U32 cnt=5;

    while(cnt>0){

        //增加延迟以模拟较长的执行时间
```

```
        PRT_TaskDelay(500);//增加延迟

        PRT_Printf("任务1执行 \n");

        cnt--;

    }

}



void Test2TaskEntry()

{

    PRT_Printf("任务2执行 \n");

    PRT_SemPost(sem_sync,OS_WAIT_FOREVER);//wait

    U32 cnt=5;

    while(cnt>0){

        //减少延迟以模拟较短的执行时间

        PRT_TaskDelay(100);//减少延迟

        PRT_Printf("任务2执行 \n");

        cnt--;

    }

}
```

任务一先执行，执行结束给任务二发信号，任务二收到信号后才执行，任务一占用资源时间较长，如果是高频率的循环，饥饿问题会更显著。

解决策略：时间片轮转调度

```
#define TIME_SLICE 100//时间片大小
```

```
void Test1TaskEntry()

{

    PRT_Printf("任务1执行 \n");

    PRT_SemPost(sem_sync);

    while(1){

        PRT_TaskDelay(TIME_SLICE);//时间片轮转

        PRT_Printf("任务1执行 \n");

    }

}



void Test2TaskEntry()

{

    PRT_Printf("任务2执行 \n");

    PRT_SemPost(sem_sync,OS_WAIT_FOREVER);

    while(1){

        PRT_TaskDelay(TIME_SLICE));//时间片轮转

        PRT_Printf("任务2执行 \n");

    }

}
```

加入了时间片轮转调度策略，对任务一任务二进行分时调度，每隔固定的时间就让正在运行的任务终止，并决定下一个时间片要执行的任务。任务调度策略可采用就绪队列实现，任务执行结束，将任务放到队列末尾，然后下一次执行队首任务。

🆗分析完啦✴️✴️✴️