

Lab7

一、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

二、实验过程

1. 实现信息量结构初始化

① 在lab7/src/include目录下新建prt_sem_external.h头文件

下面对该板块进行分析：

这个代码板块主要实现了一个信号量管理模块的接口，包含信号量的状态、类型、模式的宏定义、控制块结构体、全局变量和操作信号量的函数。

首先对宏定义进行分析：

- OS_SEM_UNUSED 和 OS_SEM_USED 用来标识信号量是否在使用。
- SEM_PROTOCOL_PRIO_INHERIT 表示优先级继承协议。
- SEM_TYPE_BIT_WIDTH 和 SEM_PROTOCOL_BIT_WIDTH 定义了信号量类型和协议的位宽。
- MAX_POSIX_SEMAPHORE_NAME_LEN 定义了POSIX信号量名称的最大长度。
- GET_SEM_LIST 等宏用来操作信号量列表和获取信号量相关信息。

最后定义了一些变量，同时声明了一些外部定义的函数。

② 在lab7/src/kernel/sem目录下新建prt_sem_init.c文件

这段代码主要是定义了信号量的初始化函数，分别分析这三个函数的功能：

- OsSemInit：初始化信号量管理模块，分配内存并初始化空闲信号量列表。
- OsSemCreate：创建信号量，检查参数并初始化信号量控制块。
- PRT_SemCreate：封装函数，调用OsSemCreate进行信号量创建。

③ 在src/bsp目录中的os_cpu_armv8_external.h文件添加定义

④ 在src/kernel/sem目录下新建prt_sem.c文件

这一部分主要定义了一系列检查等辅助函数，其中：

- OsSemPendListPut函数：将当前运行任务挂接到信号量的等待链表上。
- OsSemPendListGet函数：从信号量的等待列表中取出第一个任务并放入就绪队列。

重点核心函数分析：

- PRT_SemPend：实现等待信号响应操作，检查参数并挂起任务。
- PRT_SemPost：释放信号量，从阻塞队列移除信号并调度任务。

⑤ 在src/include目录下的prt_task_external.h文件中加入OsTskReadyAddBgd()

⑥ 在src/kernel/task目录中的prt_task.c文件加入OsTskScheduleFastPs()

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

⑧ 在src/include目录中加入prt_sem.h文件

⑨ 最后将所有新文件加入构建系统

至此，信号系统已构建完毕。

三、测试及分析

在main函数中执行任务1与任务2，运行结果符合预期。



```
~/os/lab7
> ./runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
MINIEULER BY SMILE
ctr-a h: print help of qemu emulator. ctr-a x: quit emulator.

task 2 run ...
task 1 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 1 run ...
task 1 run ...
task 1 run ...
task 1 run ...
task 1 run ...
QEMU: Terminated
```

四、作业

各种并发问题模拟，至少3种。

1.死锁

C

```
static SemHandle sem_lock1, sem_lock2;

void Test1TaskEntry()
{
    PRT_SemPend(sem_lock1, OS_WAIT_FOREVER);
    PRT_Printf("task 1 run ...\n");
    OsTskSchedule();
    PRT_SemPend(sem_lock2, OS_WAIT_FOREVER) ;
    U32 cnt = 5;
    while (cnt > 0) {
        // PRT_TaskDelay(200);
        PRT_Printf("task 1 run ...\n");
        cnt--;
    }
}

void Test2TaskEntry()
{
    PRT_SemPend(sem_lock2, OS_WAIT_FOREVER);
    PRT_Printf("task 2 run ...\n");
    PRT_SemPend(sem_lock1, OS_WAIT_FOREVER) ;
    U32 cnt = 5;
    while (cnt > 0) {
        // PRT_TaskDelay(200);
        PRT_Printf("task 2 run ...\n");
        cnt--;
    }
}
```

解释：

先让任务1持有锁1，然后进行任务切换，让任务2先持有锁2。

此时，不管是执行任务1还是任务2，任务1会等待任务2的锁2，任务2会等待任务1的锁1。形成死锁的局面。

运行结果：

```
./runMiniEuler.sh
> ./runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
MINIEULER BY SMILE
ctr-a h: print help of qemu emulator. ctr-a x: quit emulator.

task 1 run ...
task 2 run ...
█
```

2. 生产者-消费者问题

C

```
static SemHandle mutex; // 用于保护临界区的互斥信号量
static SemHandle empty; // 表示可用的空间数量
static SemHandle full; // 表示可用产品的数量

#define BUFFER_SIZE 5

static int buffer[BUFFER_SIZE]; // 缓冲区
static int in = 0; // 生产者插入的位置
static int out = 0; // 消费者读取的位置

void producer() {
    int item = 0;
    while (1) {
        // 生产一个产品
        item++;
        // 等待空闲空间
        PRT_SemPend(empty, OS_WAIT_FOREVER);
        // 获取互斥锁
        PRT_SemPend(mutex, OS_WAIT_FOREVER);
        // 将产品放入缓冲区
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        PRT_Printf("Producer produce item: %d\n", item);
        // 释放互斥锁
        PRT_SemPost(mutex);
        // 发送产品信号
        PRT_SemPost(full);
        // 模拟生产过程的延时
        for (volatile int j = 0; j < 1000000; j++) {
            // 空循环
        }
    }
}

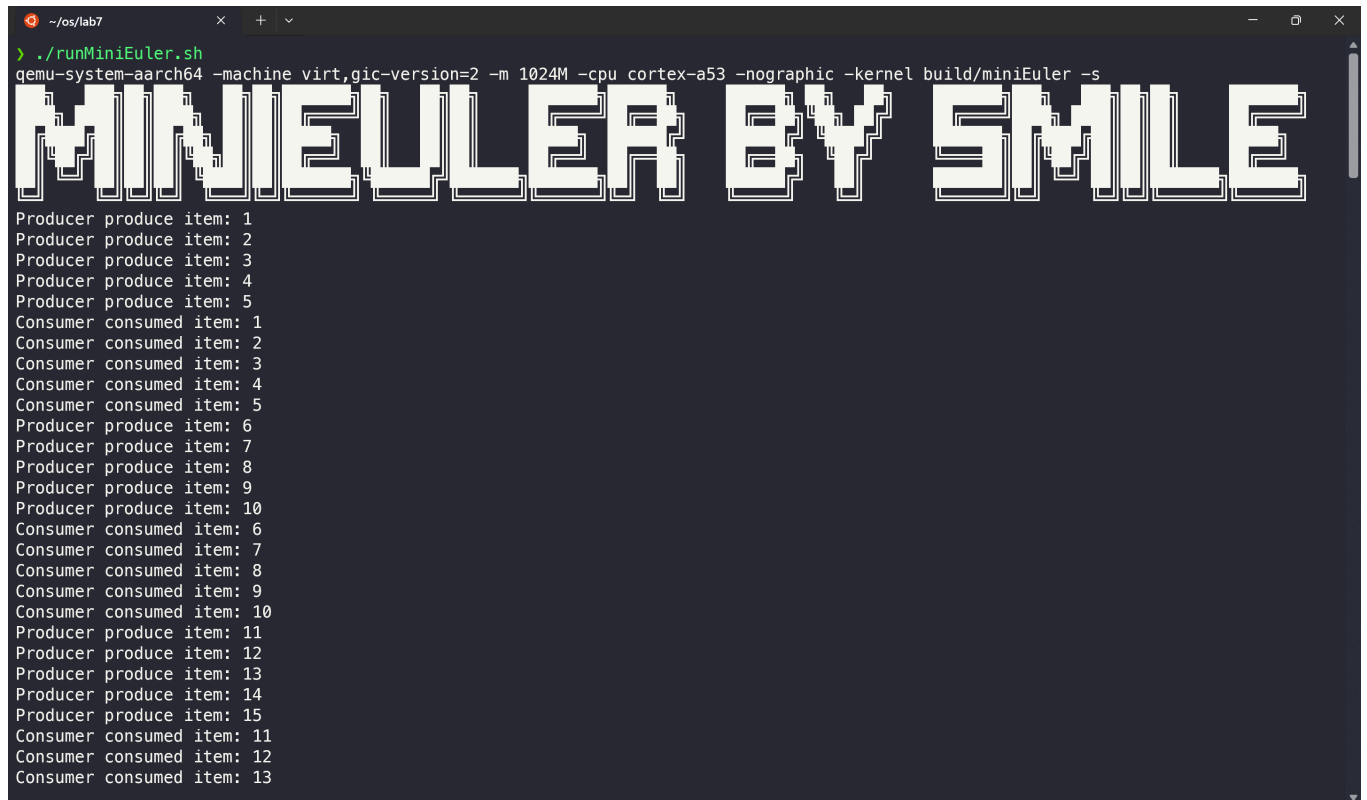
void consumer() {
    int item = 0;
    while (1) {
        // 等待可用产品
```

```

PRT_SemPend(full, OS_WAIT_FOREVER);
// 获取互斥锁
PRT_SemPend(mutex, OS_WAIT_FOREVER);
// 从缓冲区中取出产品
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
PRT_Printf("Consumer consumed item: %d\n", item);
// 释放互斥锁
PRT_SemPost(mutex);
// 发送空闲空间信号
PRT_SemPost(empty);
// 消费产品
// 模拟消费过程的延时
for (volatile int j = 0; j < 1000000; j++) {
    // 空循环
}
}
}

```

运行结果如下：



```

~/os/lab7
> ./runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
MINIEULER BY SMILE
Producer produce item: 1
Producer produce item: 2
Producer produce item: 3
Producer produce item: 4
Producer produce item: 5
Consumer consumed item: 1
Consumer consumed item: 2
Consumer consumed item: 3
Consumer consumed item: 4
Consumer consumed item: 5
Producer produce item: 6
Producer produce item: 7
Producer produce item: 8
Producer produce item: 9
Producer produce item: 10
Consumer consumed item: 6
Consumer consumed item: 7
Consumer consumed item: 8
Consumer consumed item: 9
Consumer consumed item: 10
Producer produce item: 11
Producer produce item: 12
Producer produce item: 13
Producer produce item: 14
Producer produce item: 15
Consumer consumed item: 11
Consumer consumed item: 12
Consumer consumed item: 13

```

3.哲学家进餐问题

C

```
#define PHILO_NUM 5
static SemHandle forks[PHILO_NUM];    // 叉子信号量数组
// static SemHandle table;            // 餐桌信号量 (限制同时就餐
// 人数)

void philosopher(U32 id) {
    U32 left = id;
    U32 right = (id + 1) % PHILO_NUM;

    // 调整拿叉子顺序避免死锁 (最后一个哲学家反向拿取)
    // if (id == PHILO_NUM - 1) {
    //     U32 temp = left;
    //     left = right;
    //     right = temp;
    // }

    while (1) {
        // 思考阶段
        PRT_Printf("Philosopher %d is thinking\n", id);
        for(volatile int i = 0; i < 1000000; i++); // 模拟思考时间
        // 请求进入餐桌
        // PRT_SemPend(table, OS_WAIT_FOREVER);

        // 拿取左边叉子
        PRT_SemPend(forks[left], OS_WAIT_FOREVER);
        PRT_Printf("Philosopher %d took left fork %d\n", id,
left);

        // 拿取右边叉子
        PRT_SemPend(forks[right], OS_WAIT_FOREVER);
        PRT_Printf("Philosopher %d took right fork %d\n", id,
right);

        // 进餐阶段
        PRT_Printf("Philosopher %d is EATING\n", id);
        for(volatile int i = 0; i < 1000000; i++); // 模拟吃饭时间
```

```

        // 放下右边叉子
        PRT_SemPost(forks[right]);
        PRT_Printf("Philosopher %d released right fork %d\n", id,
right);

        // 放下左边叉子
        PRT_SemPost(forks[left]);
        PRT_Printf("Philosopher %d released left fork %d\n", id,
left);

        // 离开餐桌
        // PRT_SemPost(table);

    }
}

```

运行结果如下：

```

~/os/lab7
> ./runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
PHILOEARS
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 1 took left fork 1
Philosopher 1 took right fork 2
Philosopher 1 is EATING
Philosopher 3 took left fork 3
Philosopher 3 took right fork 4
Philosopher 3 is EATING
Philosopher 1 released right fork 2
Philosopher 1 released left fork 1
Philosopher 2 took left fork 2
Philosopher 1 is thinking
Philosopher 3 released right fork 4
Philosopher 3 released left fork 3
Philosopher 3 is thinking
Philosopher 4 took left fork 4
Philosopher 4 took right fork 0
Philosopher 4 is EATING
Philosopher 2 took right fork 3
Philosopher 2 is EATING
Philosopher 1 took left fork 1
Philosopher 4 released right fork 0
Philosopher 4 released left fork 4
Philosopher 4 is thinking
Philosopher 2 released right fork 3
Philosopher 2 released left fork 2

```

死锁的结果：


```
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
```

PHILOERS

```
Philosopher 0 is thinking  
Philosopher 1 is thinking  
Philosopher 2 is thinking  
Philosopher 3 is thinking  
Philosopher 4 is thinking  
Philosopher 0 took left fork 0  
Philosopher 1 took left fork 1  
Philosopher 2 took left fork 2  
Philosopher 3 took left fork 3  
Philosopher 4 took left fork 4
```

五、心得体会

1. 通过这个实验我更为深入地理解了信号量的相关原理
2. 我理解了信号量及信号量相关函数实现的底层原理
3. 通过自己模拟教材中的并发问题，我对并发有了更深刻的理解，也使得我将来能够更好地解决这些问题