作者：曾任斯

##

1. 基本概念

信号量SEM全称Semaphore，中文也翻译为信号灯。信号量跟MSG和SHM有极大的不同，SEM不是用来传输数据的，而是作为“旗语”，用来协调各进程或者线程工作的。

信号量本质上是一个数字，用来表征一种资源的数量，当多个进程或者线程争夺这些稀缺资源的时候，信号量用来保证他们合理地、秩序地使用这些资源，而不会陷入逻辑谬误之中。

1.1 红绿灯

在一个繁忙的十字路口中，路口的通过权就是稀缺资源，并且是排他的，南北向的车辆用了，东西向的车辆就不能使用，就是通过十字路口的权限，为了避免撞车，一个方向通行时另外方向的车必须停下来等待，直到红绿灯变换为止。这就是最基本的资源协同。

信号灯

1.2 信号量的分类

在Unix/Linux系统中常用的信号量有三种：

- **IPC信号量组**
- POSIX具名信号量
- POSIX匿名信号量

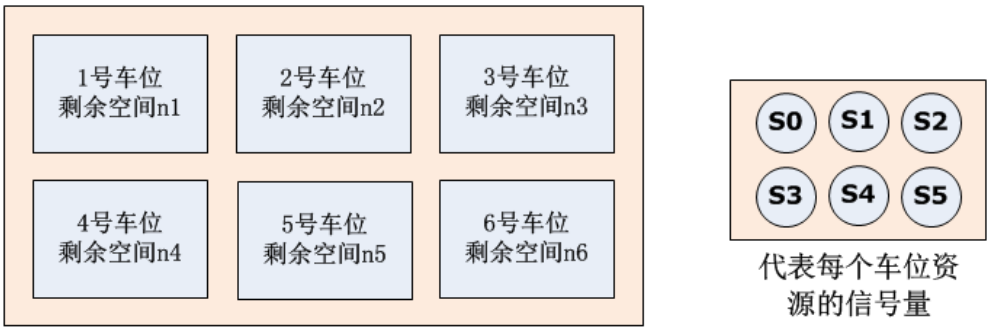
本节课讲解的是第一种IPC信号量组，既然说到它是一个“组”，说明这种机制可以一次性在其内部设置多个信号量，实际上在其内部实现中，IPC信号量组是一个数组，里面包含N个信号量元素，每个元素相当于一个POSIX信号量。

1.3 基本概念

- 临界资源（critical resources）
  - 多个进程或线程有可能同时访问的资源（变量、链表、文件等等）
- 临界区（critical zone）
  - 访问这些资源的代码称为临界代码，这些代码区域称为临界区
- **P操作**
  - 程序进入临界区之前必须要对资源进行申请，这个动作被称为P操作，这就像你要把车开进停车场之前，先要向保安申请一张停车卡一样，P操作就是申请资源，如果申请成功，资源数将会减少。如果申请失败，要不在门口等，要不走人。
- **V操作**

- 程序离开临界区之后必须要释放相应的资源，这个动作被称为V操作，这就像你把车开出停车场之后，要将停车卡归还给保安一样，V操作就是释放资源，释放资源就是让资源数增加。

信号量组非常类似于停车场的卡牌，想象一个有N个车位的停车场，每个车位是立体的可升降的，能停n辆车，那么我们可以用一个拥有N个信号量元素，每个信号量元素的初始值等于n的信号量来代表这个停车场的车位资源——某位车主要把他的m辆车开进停车场，如果需要1个车位，那么必须对代表这个车位的信号量元素申请资源，如果n大于等于m，则申请成功，否则不能把车开进去。



停车场和信号量组

3. 函数接口

3.1 创建（或打开）SEM

与其他IPC对象类似，首先需要创建（或打开）SEM对象，接口如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

接口说明：

- key：SEM对象键值
- nsems：信号量组内的信号量元素个数
- semflg：创建选项
  - IPC_CREAT：如果key对应的信号量不存在，则创建之
  - IPC_EXCL：如果该key对应的信号量已存在，则报错
  - mode：信号量的访问权限（八进制，如0644）

创建信号量时，还受到以下系统信息的影响：

1. SEMMNI：系统中信号量的总数最大值。
2. SEMMSL：每个信号量中信号量元素的个数最大值。
3. SEMMNS：系统中所有信号量中的信号量元素的总数最大值。

Linux中，以上信息在 `/proc/sys/kernel/sem` 中可查看。
示例代码：

```c
int main()
{
     key_t key = ftok(".", 1);

     // 创建（若已有则打开）一个包含2个元素的信号量组
     int id = semget(key, 2, IPC_CREAT|0666);
}
```

3.2 PV操作

对于信号量而言，最重要的作用就是用来表征对应资源的数量，所谓的P/V操作就是对资源数量进行+n/-n 操作，既然只是个加减法，那么为什么不使用普通的整型数据呢？原因是：

- 整型数据的加减操作不具有原子性，即操作可能被中断
- 普通加减法无法提供阻塞特性，而申请资源不可得时应进入阻塞

对于原子性再做个简单的解释，即这种资源数量的加减法不能有中间过程，不管是成功还是失败都必须一次性完成。加减法看似简单，但在硬件层面上并非一个原子性操作，而是包含了多个寄存器操作步骤，因此不能作为P/V操作的手段。

在SEM对象中，P/V操作的函数接口如下：

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
```

接口说明：

- 参数
    - semid：SEM对象ID
    - sops：P/V操作结构体sembuf数组
    - nsops：P/V操作结构体数组元素个数
- 返回值
    - 成功：0
    - 失败：-1

其中，所谓P/V操作结构体定义如下：

```c
struct sembuf
{
    unsigned short sem_num;  /* 信号量元素序号（数组下标） */
    short          sem_op;   /* 操作参数 */
    short          sem_flg;  /* 操作选项 */
};
```

举例说明，假设有一个信号量组SEM对象，包含两个元素，现在要对第0个元素进程P操作（即减操作），对第1个元素进程V操作（即加操作），则代码如下：

```c
int main()
{
     key_t key = ftok(".", 1);
```

```
    // 创建（若已有则打开）一个包含2个元素的信号量组
    int id = semget(key, 2, IPC_CREAT|0666);

    // 定义包含两个P/V操作的结构体数组
    struct sembuf op[2];
    op[0].sem_num = 0;  // 信号量元素序号
    op[0].sem_op  = -2; // P操作
    op[0].sem_num = 0;  // 选项默认0

    op[1].sem_num = 1;  // 信号量元素序号
    op[1].sem_op  = +3; // V操作
    op[1].sem_num = 0;  // 选项默认0

    // 同时对第0、1号信号量元素分别进行P、V操作
    semop(id, op, 2);
}
```

注意：

1. P操作是申请资源，因此如果资源数不够的话会导致进程阻塞，这正是我们想要的效果，因为资源数不可为负数。
2. V操作是释放资源，永远不会阻塞。
3. SEM对象的一大特色就是可以对多个信号量元素同时进行P/V操作，这也是跟POSIX单个信号量的区别。

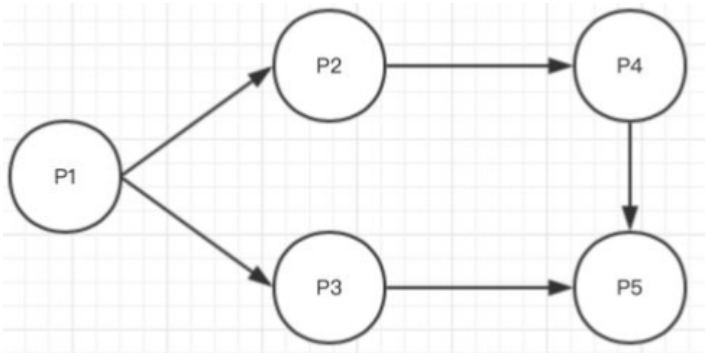
等零操作

当操作结构体sembuf中的sem_op为0时，称为等零操作，即阻塞等待直到对应的信号量元素的值为零，示例代码如下：

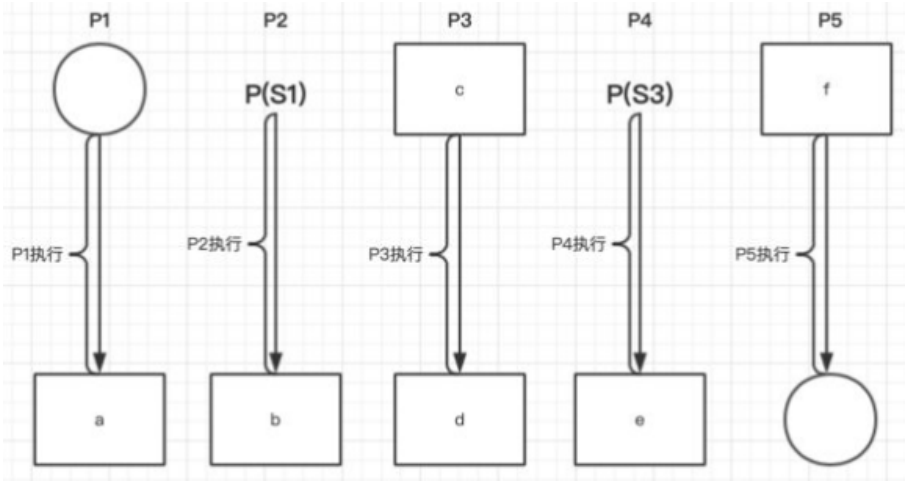
```
struct sembuf op[1];
op[0].sem_num = 0;
op[0].sem_op  = 0; // 等零操作
op[0].sem_flg = 0;

// 若此时信号量组中的第0个元素的值不为零，则阻塞直到恢复零为止
semop(id, op, 1);
```

「课堂练习」

进程P1、P2、P3、P4和P5的**前趋图**如下图所示：



若用PV操作控制进程P1、P2、P3、P4和P5的并发执行过程，则需要设置5个信号量S1、S2、S3、S4和S5，且信号量S1～S5的初值都等于零。。



填空：
a：___
b：___
c：___
d：___
e：___
f：___

3.3 其余操作

system-V的IPC对象都有类似的操作接口，SEM对象也有control函数，接口如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

接口说明：

- semid：信号量组的ID
- semnum：信号量组内的元素序号（从0开始）
- cmd：操作命令字：
    - IPC_STAT：获取信号量组的一些信息，放入semid_ds{ }中
    - IPC_SET：将 semid_ds{ } 中指定的信息，设置到信号量组中
    - **IPC_RMID**：删除指定的信号量组。
    - GETALL：获取所有的信号量元素的值。
    - SETALL：设置所有的信号量元素的值。
    - GETVAL：获取第semnum个信号量元素的值。
    - **SETVAL**：设置第semnum个信号量元素的值。

在上述常见控制命令中，以 **IPC_RMID** 和 **SETVAL** 用的最多。

1. IPC_RMID
其中，当命令字为IPC_RMID时，意为删除SEM对象，这操作与其他两种IPC对象一样，示例代码如下：

```
// 删除SEM对象
semctl(id, 0/*该参数将被忽略*/, IPC_RMID);
```

2. SETVAL

一般而言，SEM对象在刚被创建出来的时候需要进行初始化，该命令字可以执行初始化的操作，示例代码如下：

```
// 操作联合体
union semun
{
     int              val;    /* Value for SETVAL */
     struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
     unsigned short  *array;  /* Array for GETALL, SETALL */
     struct seminfo  *__buf;  /* Buffer for IPC_INFO */
};

int main()
{
    // 创建（或打开）SEM对象
    // 注意需要指定 IPC_EXCL 模式，因为要区分是否是新建的SEM对象
    int semid = semget(key, 1, IPC_EXCL|IPC_CREAT|0666);

    // 1，若SEM对象已存在，则重新打开即可
      if(semid == -1 && errno == EEXIST)
            semid = semget(key, 1, 0);

    // 2，若SEM对象为当前新建，则需进行初始化
      else if(semid > 0)
    {
        union semun a;
        a.val = 1; // 假设将信号量元素初始值定为1

        semctl(semid, 0, SETVAL, a);
    }
}
```

注意：

1. semun是一个联合体，各成员对应不同的命令字。
2. 该联合体须由调用者自己定义。

2. IPC_SET / IPC_STAT

这两个命令字用来设置、获取SEM对象的相关属性信息，这些信息用如下结构体表达：

```
struct semid_ds
{
     struct ipc_perm sem_perm;  /* 所有者、权限*/
     time_t          sem_otime; /* 最后一次PV操作的时间 */
     time_t          sem_ctime; /* 最后一次IPC_SET的时间 */
     unsigned long   sem_nsems; /* 信号量元素个数 */
};
```