**进程间通信之信号灯集**

1. **原理讲解**

**<1>system信号量和posix信号量**

**说到信号灯集，大家一定会首先，想到我们前面学习的信号量。我们之前学习的信号量是我们的posix信号量，常用于我们的线程之间的通讯。今天我们学习的信号量（也叫信号灯集），是针对System V而言的，常用于进程之间的通信。我们先来了解一个概念-----计数信号量（set of counting semaphores），它表示一个或者多个信号量组成了一个集合，我们也称之后为信号灯集。我们而System V信号量就是计数信号量，而posix信号量就是单个的计数信号量。**

**<2>信号灯概述**

**首先，信号灯的出现主要是为了解决多任务程序并发执行时，对资源的访问出现的竞争同步的问题。信号量最初是对资源的一种抽象和模拟，最初的信号量(整型信号量)，就是通过一个数值，来表示某种资源的个数，在你这个抽象的基础上，他的发明者Dijkstra(迪杰斯塔拉)为其制定了两个原子性(一次性执行完的，不可被中断的)的操作，P操作(passeren（申请通过）)，V操作（vrijgeven释放)来实现对互斥资源的访问，后来处于对(1).多个临界资源的访问，(2)和资源使用数量的限制(在有些情况下，当资源数量低于某一下限值时，便不予分配).信号量的概念得到进一步发展，形成了所谓的”信号灯集”.**

**<3>、信号灯的两种类型**

**A、二值信号灯**

**最简单的信号灯形式，信号灯的值只能取0或1,类似互斥锁。**

**注意：虽然二值信号灯能够实现互斥锁的功能，但两者的关注内容不同。信号灯强调共享资源，只要共享资源可用，其他进程同样可以修改信号灯的值；互斥锁更强调进程，占用资源的进程使用完资源后，必须由进程本身来解锁。**

**(理解：**

**信号量相当于是代码周围的卫兵,当卫兵发现共享代码段正在被执行,则卫兵不让后续调用者前去执行;当共享代码段为空闲时,卫兵允许后续调用者去执行;)**

**[信号量:强调的是请人给自己当保镖。]**

**B.计数信号灯**

**信号灯的值可以取任意非负值(当然受内核本省的约束)，用来统计资源，其值就代表可用资源的个数。**

**注意：通常所说的系统V信号灯指的是计数信号灯集**

**总结：**

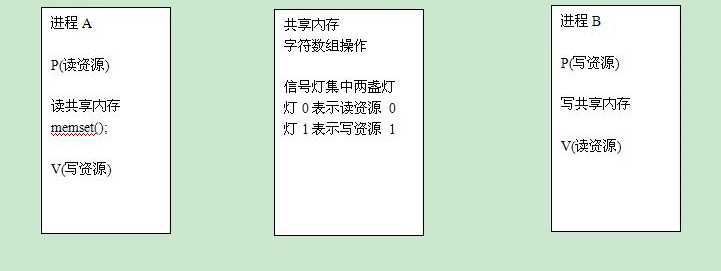
**从上面的描述可知，信号量的大致的发展，”信号灯集”是在”(简单)信号量”的基础上发展过来的。是为了解决以下问题：**

**<1>.多个临界资源的访问**

**<2>.资源保护的目的**

1. **信号灯集和共享内存**

**一般来说，我们的信号灯集和我们的共享内存联合起来使用的。就是我们所谓的P,V操作。一般是实现，我们进程之间的同步。**



**大致的操作流程我们可以分为以下几步：**

**<1>获得一个指定的key值。**

1. **指定参数IPC\_PRIVATE ----- 亲缘进程间的通信。**
2. **通过ftok来获得一个key值 ---- 任意进程间的通信。**

**<2>.创建IPC对象**

**<3>初始化信号灯集中信号灯的值**

**<4>操作IPC对象-----申请资源(P)，释放资源(V)**

**<4>.删除IPC对象**

1. **获得一个指定的key值。**

|  |
| --- |
| key\_t ftok( **const char** \*pathname, **int** proj\_id);  功能 :  获得 key 值  参数 :  @pathname 一个已经存在的文件路径 ,( 避免传 ".",因为”.”在不同的目录下，代表的是不同目录。 )  @proj\_id 只会使用它的低八位  返回值：  成功返回 key值  失败返回 -1并置errno  例 :  key\_t key = ftok( "/linux/ubuntu" , 'k' ); |

**2）创建IPC对象**

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h>  **int** semget(key\_t key, **int** nsems, **int** semflg);  功能: 获得一个“信号量集”ID  参数: @key IPC对象对应的键值  取值:  <1>.ftok 生成的 . 任意进程间使用  <2>.IPC\_PRIVATE. 亲缘进程间间使用  @nsems 要创建的信号灯的个数（即资源类型的个数，注不是某一类资源的个数）  @semflg 类似open中的打开模式的标志位。  可以是 IPC\_CRATE IPC\_EXCL的或的形式  如: IPC\_CREAT | 0666 表示不存在则创建  IPC\_CREAT |IPC\_EXCL一起使用时可确定要创建的信号灯集是否存在。类似文件中的操作。  返回值:  成功 信号量的id值  返回 -1 并 置errno |

**3）初始化信号灯集中信号灯的值**

**翻译如下：**

**Semctl()函数 由cmd参数来指定控制操作，并且由semid参数来设置信号灯的标识符号。**

**Semnum表示我们的信号灯集的个数(信号灯集的个数默认从0开始的)。**

**Semctl()函数可以有3个或者4个参数，它是有cmd参数来决定的。当有四个参数的时候，**

**第四个参数一定是一个union semun类型。被调用的进程必须先设置如下类型的共用体：**

**union semun{**

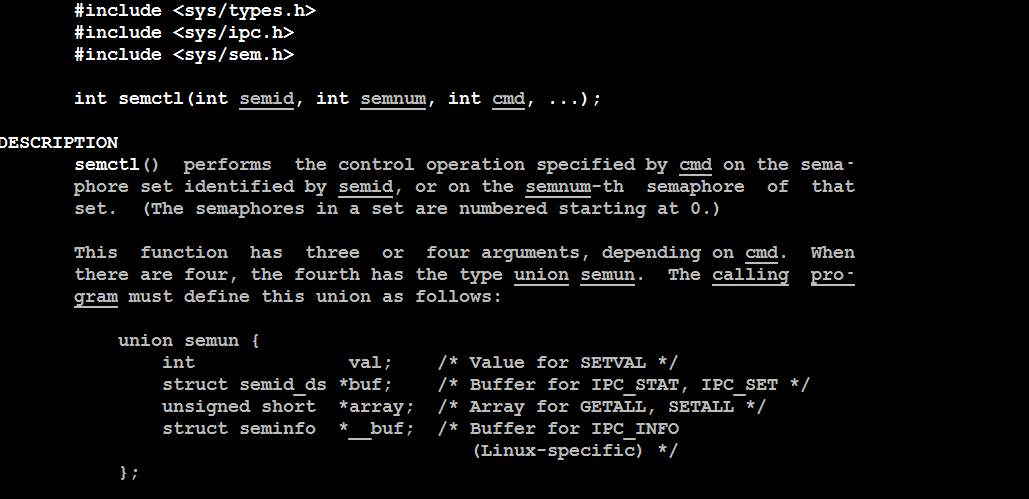
**int val; // 信号量的初始值。**

**struct semid\_ds \*buf; //cmd参数为IPC\_STAT,IPC\_SET设置的缓冲区**

**unsigned short \*arrary; //cmd参数为GETALL,SETALL设置的指针首地址**

**struct seminfo \*\_\_buf; //cmd 参数为IPC\_INFO，设置的缓冲区.**

**}**



**总结如下：**

|  |
| --- |
| **int** semctl(**int** semid,**int** semnum,**int** cmd, .....);  功能: 对信号灯集中信号灯进行控制  参数:  @semid 信号灯集的ID  @semnum 信号灯在信号灯集中的编号，创建信号灯集的时候，编号从0开始  @cmd : 操作命令  **SETVAL : 设置信号灯的值，此时要用到可变参数，添加第四个共用体参数**  **GETVAL: 获得信号灯的值，此时要用到可变参数，添加第四个共用体参数**  **IPC\_RMID:立即删除信号灯集，不需要额外用到第四个参数**  **@.... :可变参数，根据cmd的参数而定。**  返回值:  成功 SETVAL 被设置成功返回0  SETVAL 被设置成功返回semval的值  IPC\_RMID 被设置成功返回0  失败 返回-1 并置 errno |

**例如： 假如我们创建了0号信号灯集，我们把0号信号灯的值设置为1.**

**//注:以下共用体在代码中需要先定义**

**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(Linux-specific) \*/**

**};**

**union semum sem\_var;**

**sem\_varl.var = 1; //给结构体变量赋值为1**

**//把0号信号灯集合的值赋与为1**

**if(semctl(semid,0,SETVAL,sem\_val) < 0)**

**{**

**perror（“Fail to semctl”）**

**}**

**4）操作IPC对象---封装P和V操作**

|  |
| --- |
| **int** semop(**int** semid, **struct** sembuf \*sops, **unsigned** nsops);  参数:  @semid 信号量对应的id  @sops 对应的要操作的结构体首地址(信号灯集 实际上 是有信号灯组成的数组)  @nsops 操作信号灯的个数    返回值：  　　　　成功 返回0  失败 返回-1，并置errno |
| **相关数据结构的说明:**  struct sem\_buf 是操作信号量的一组操作的集合  这个集合中包含了三个成员  <1>.要操作的信号量  <2>.要执行的操作 (请求资源（P操作）还是 释放资源（v操作）)  <3>.操作的标志位（IPC\_NOWAIT 或 是SEM\_UNDO）  Struct sem\_buf  {  **unsigned short** sem\_num; /\* semaphore number \*/  **short** sem\_op; /\* semaphore operation \*/  **short** sem\_flg; /\* operation flags \*/  };  结构体成员说明:  @sem\_op的使用说明:    sem\_op >0 释放sem\_op数目的共享资源；    sem\_op = 0可以用于对共享资源是否已用完的测试；  sem\_op < 0 申请-sem\_op个共享资源。  注：   1. sem\_op = 0,调用者阻塞等待，直到信号灯的值等于0时返回。   可以用来测试共享资源是否已用完。  b. sem\_op = 1,释放资源，V操作  c. sem\_op = -1,分配资源，P操作  @sem\_flg  可以有三种值  sem\_flg =0;  正常操作，即默认资源为未满足时，进行阻塞。  sem\_flg = IPC\_NOWAIT;  使对信号量的操作非阻塞。  即:该标志被指定时，调用进程在信号量的值不满足的条件下，不会阻塞。  而是直接返回-1，并置errno=EAGAIN.  sem\_flg = SEM\_UNDO;  进程结束时自动释放申请的资源。 |

**常用P和V操作的封装如下：**

**//申请资源 P操作**

**void P(int sem\_id,int sem\_num)**

**{**

**struct sembuf sem; //填充信号灯操作的结构体**

**sem.sem\_num = sem\_num;**

**sem.sem\_op = -1; //申请一个资源**

**sem.sem\_flg = 0;**

**if (semop(sem\_id,&sem,1) < 0)**

**{**

**perror("fail to seop - p");**

**exit(EXIT\_FAILURE);**

**}**

**return；**

**}**

**//释放资源 V 操作**

**void V(int sem\_id,int sem\_num)**

**{**

**struct sembuf sem; //填充信号灯操作的结构体**

**sem.sem\_num = sem\_num;**

**sem.sem\_op = 1; //释放一个资源**

**sem.sem\_flg = 0;**

**if (semop(sem\_id,&sem,1) < 0)**

**{**

**perror("fail to seop - p");**

**exit(EXIT\_FAILURE);**

**}**

**return;**

**}**

**5）删除信号灯集**

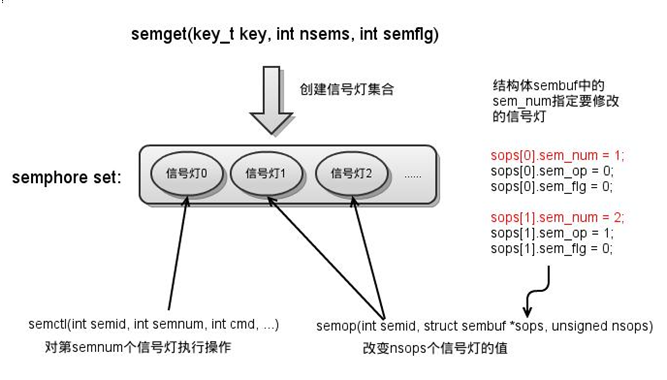
**if(semctl(semid,0,IPC\_RMID) < 0)**

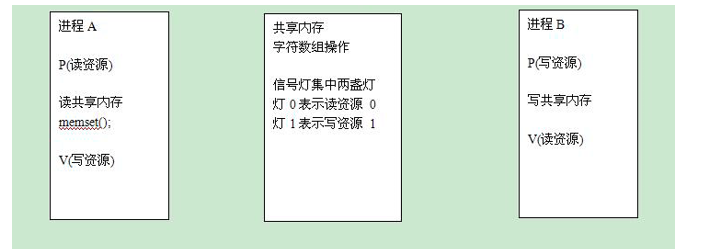
**{**

**perror(“Fail to semctl IPC\_RMID”);**

**exit(EXIT\_FAILURE);**

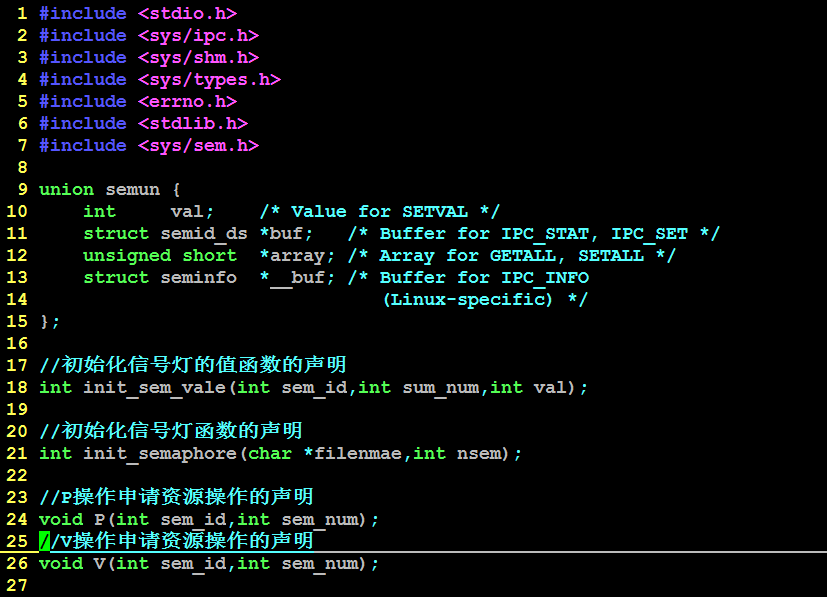
**}**

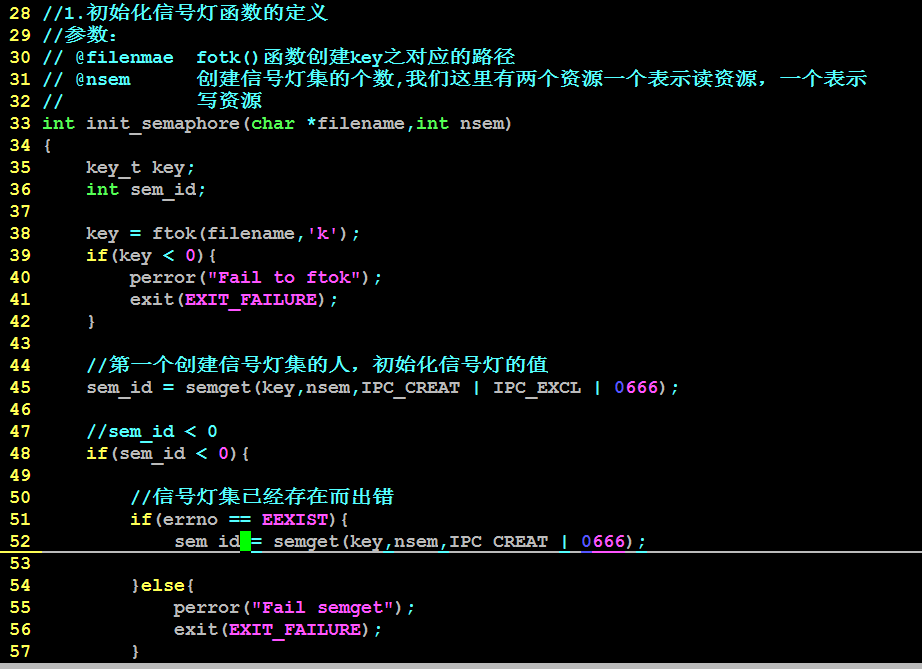


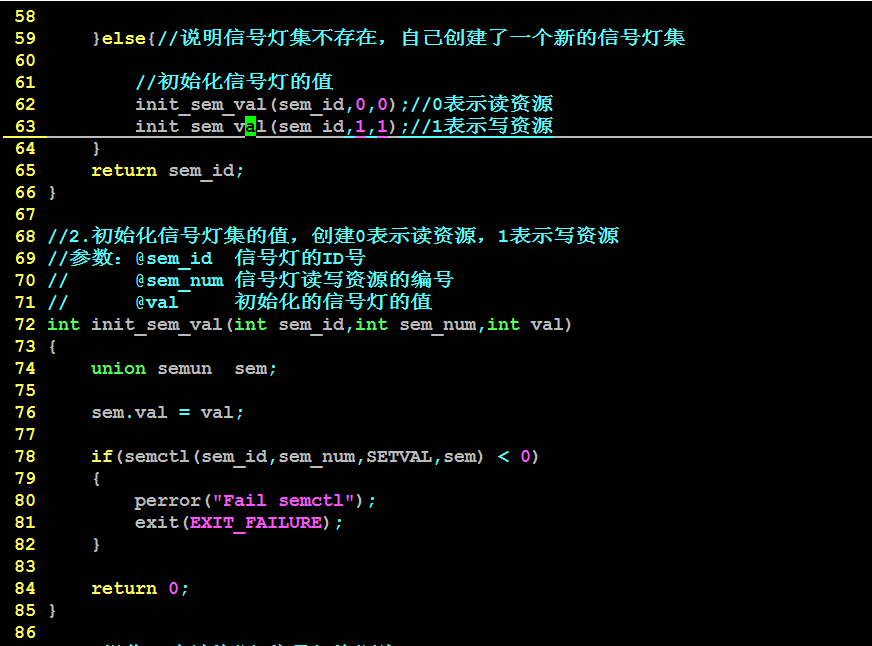


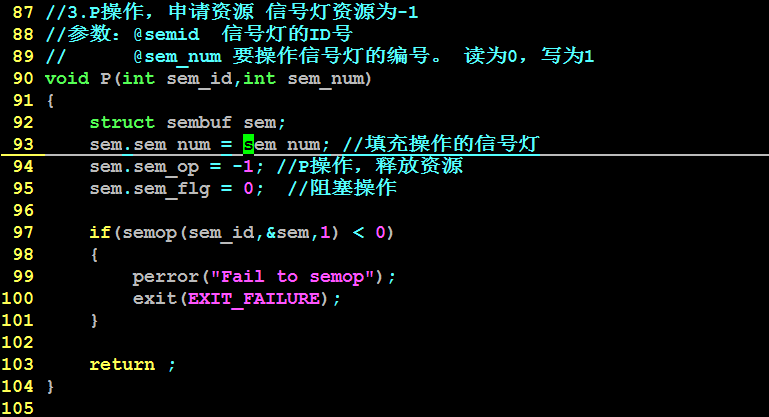
**实验代码：对上述图片的实现代码如下：**

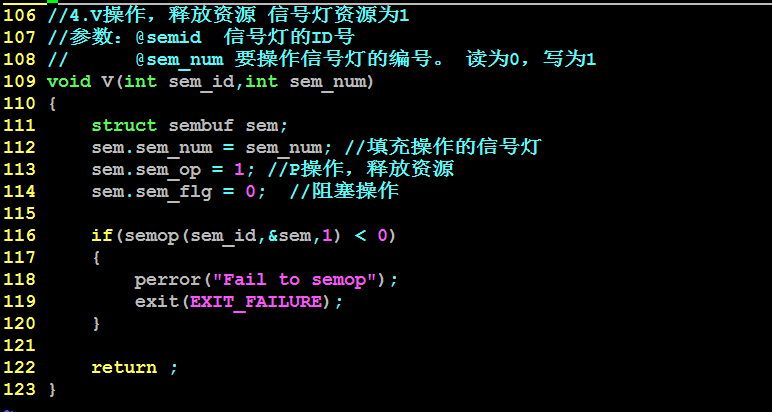
**sem.c 🡺信号量各个函数的封装**



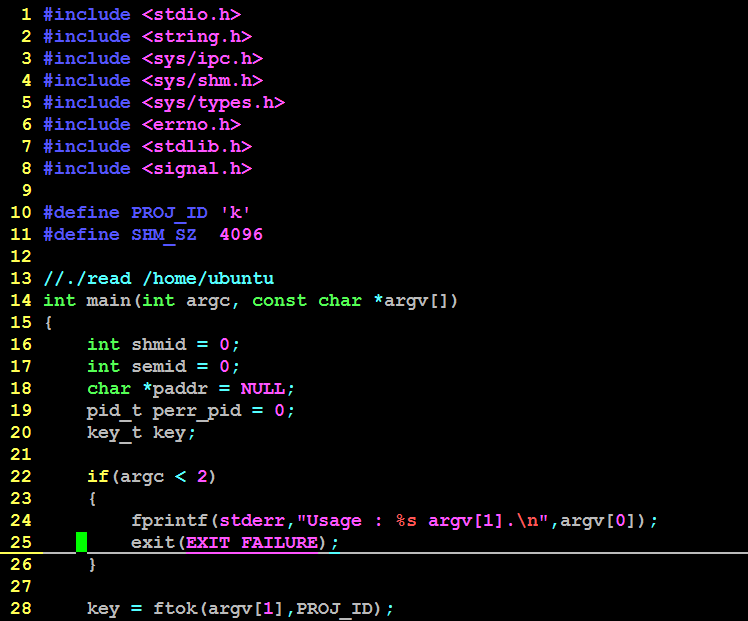


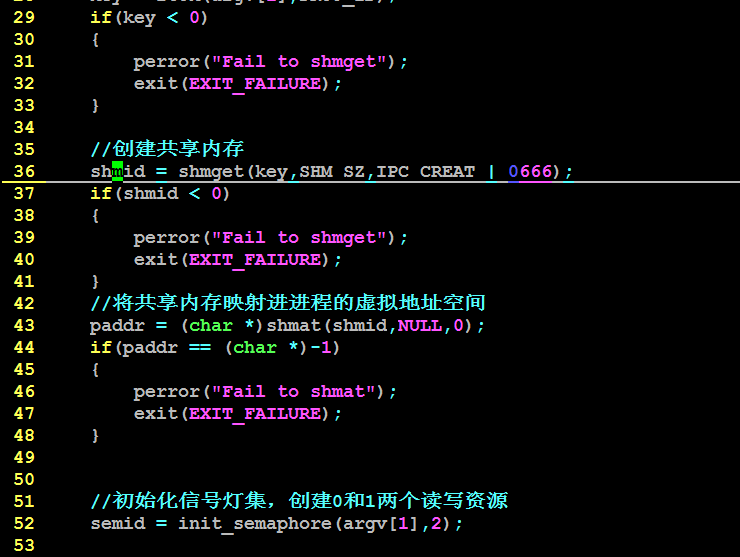


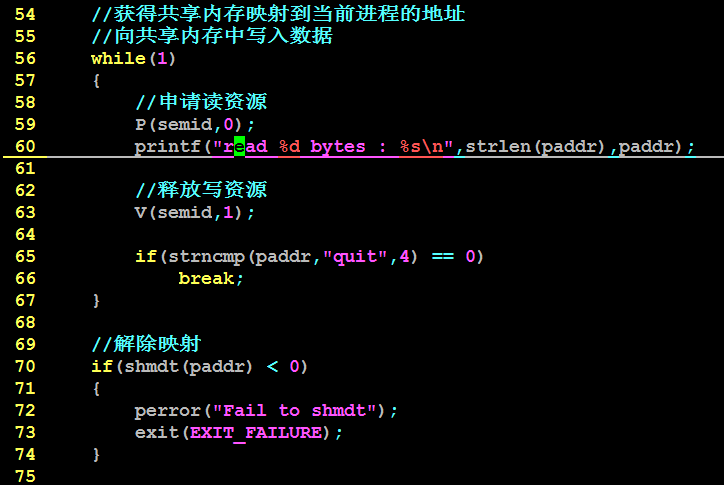


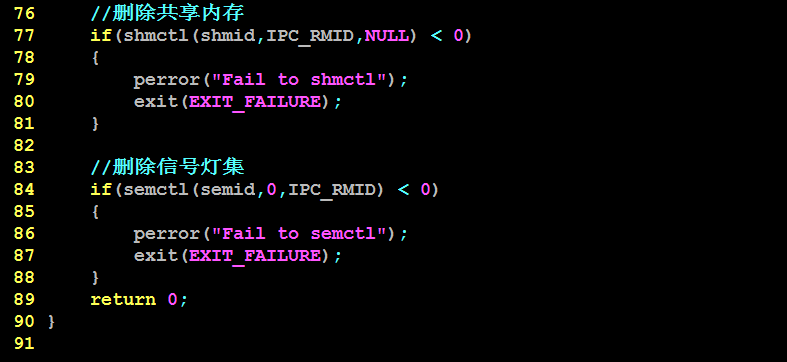


**shm\_read.c**

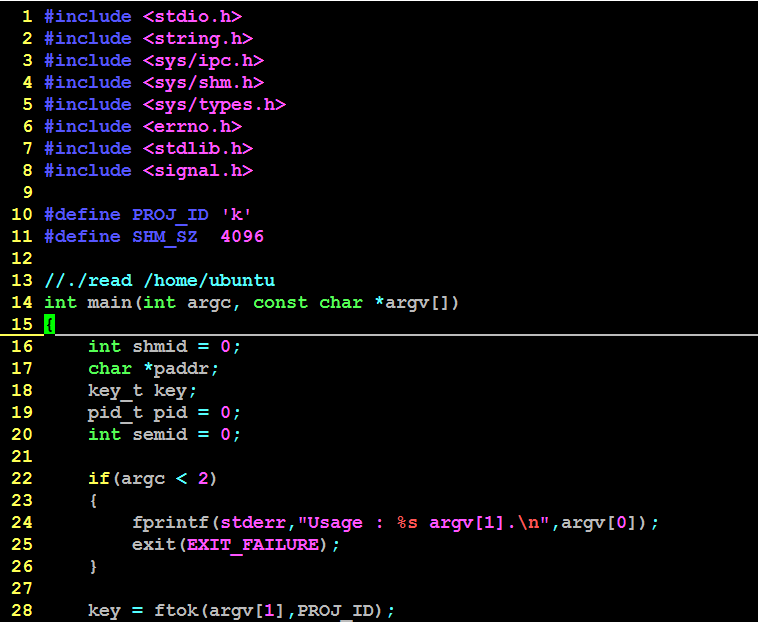


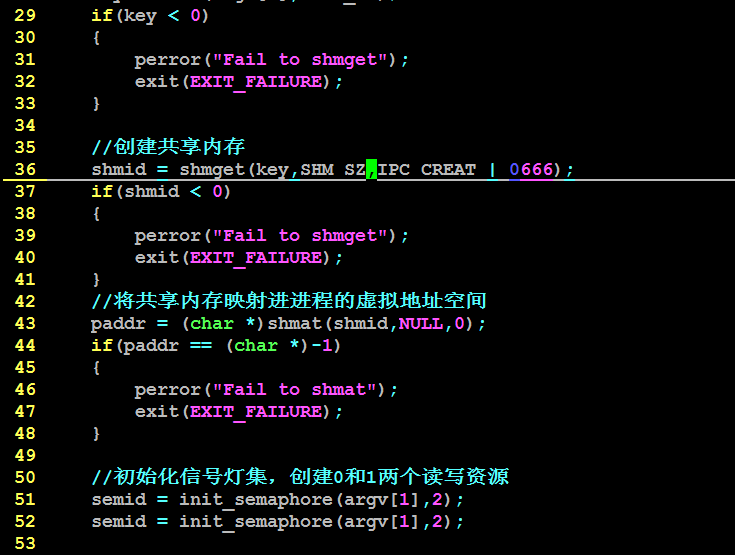


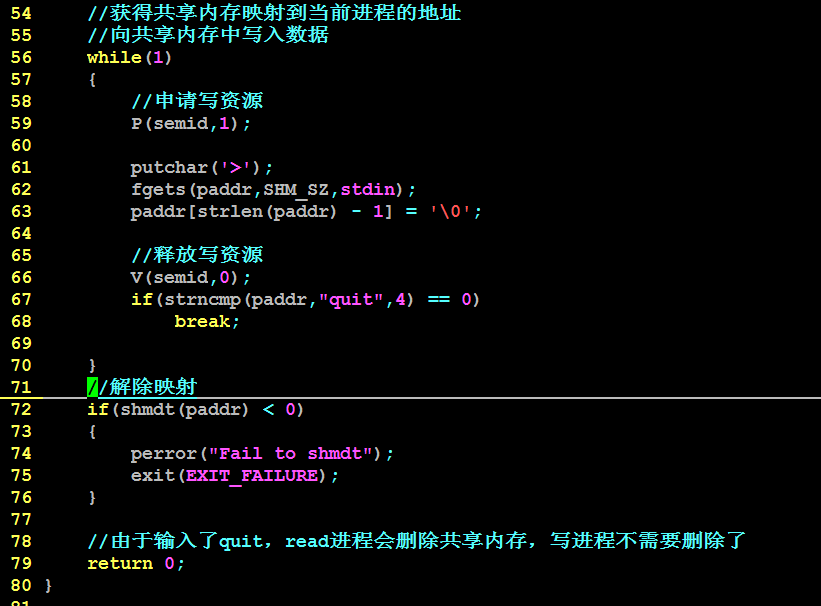




**shm\_write.c**







**运行结果:**

