**一、信号灯概述**

信号灯与其他进程间通信方式不大相同，它主要提供对进程间共享资源访问控制机制。相当于内存中的标志，进程可以根据它判定是否能够访问某些共享资源，同时，进程也可以修改该标志。除了用于访问控制外，还可用于进程同步。信号灯有以下两种类型：

* 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，类似于互斥锁。   
  注：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。
* 计算信号灯：信号灯的值可以取任意非负值（当然受内核本身的约束）。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part4/#ibm-pcon)

**二、Linux信号灯**

linux对信号灯的支持状况与消息队列一样，在red had 8.0发行版本中支持的是系统V的信号灯。因此，本文将主要介绍系统V信号灯及其相应API。在没有声明的情况下，以下讨论中指的都是系统V信号灯。

注意，通常所说的系统V信号灯指的是计数信号灯集。

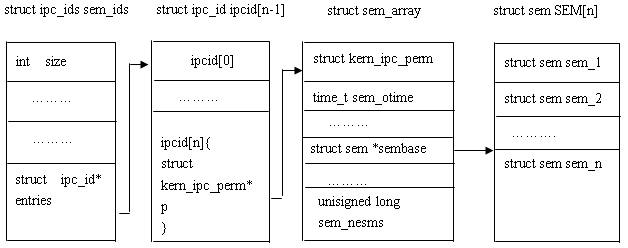
[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part4/#ibm-pcon)

**三、信号灯与内核**

1、系统V信号灯是随内核持续的，只有在内核重起或者显示删除一个信号灯集时，该信号灯集才会真正被删除。因此系统中记录信号灯的数据结构（struct ipc\_ids sem\_ids）位于内核中，系统中的所有信号灯都可以在结构sem\_ids中找到访问入口。

2、下图说明了内核与信号灯是怎样建立起联系的：

其中：struct ipc\_ids sem\_ids是内核中记录信号灯的全局数据结构；描述一个具体的信号灯及其相关信息。



其中，struct sem结构如下：

|  |
| --- |
| struct sem{  int semval; // current value  int sempid // pid of last operation  } |

从上图可以看出，全局数据结构struct ipc\_ids sem\_ids可以访问到struct kern\_ipc\_perm的第一个成员：struct kern\_ipc\_perm；而每个struct kern\_ipc\_perm能够与具体的信号灯对应起来是因为在该结构中，有一个key\_t类型成员key，而key则唯一确定一个信号灯集；同时，结构struct kern\_ipc\_perm的最后一个成员sem\_nsems确定了该信号灯在信号灯集中的顺序，这样内核就能够记录每个信号灯的信息了。kern\_ipc\_perm结构参见《Linux环境进程间通信（三）：消息队列》。struct sem\_array见附录1。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part4/#ibm-pcon)

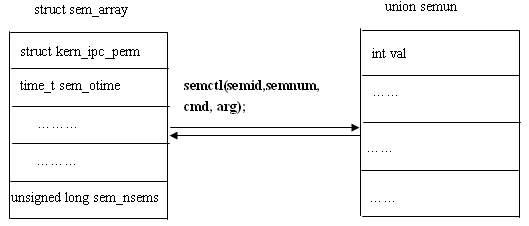
**四、操作信号灯**

**对消息队列的操作无非有下面三种类型：**

1、 打开或创建信号灯   
与消息队列的创建及打开基本相同，不再详述。

2、 信号灯值操作   
linux可以增加或减小信号灯的值，相应于对共享资源的释放和占有。具体参见后面的semop系统调用。

3、 获得或设置信号灯属性：   
系统中的每一个信号灯集都对应一个struct sem\_array结构，该结构记录了信号灯集的各种信息，存在于系统空间。为了设置、获得该信号灯集的各种信息及属性，在用户空间有一个重要的联合结构与之对应，即union semun。



联合semun数据结构各成员意义参见附录2

**信号灯API**

1、文件名到键值

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  key\_t ftok (char\*pathname, char proj)； |

它返回与路径pathname相对应的一个键值，具体用法请参考《Linux环境进程间通信（三）：消息队列》。

2、 linux特有的ipc()调用：

int ipc(unsigned int call, int first, int second, int third, void \*ptr, long fifth);

参数call取不同值时，对应信号灯的三个系统调用：   
当call为SEMOP时，对应int semop(int semid, struct sembuf \*sops, unsigned nsops)调用；   
当call为SEMGET时，对应int semget(key\_t key, int nsems, int semflg)调用；   
当call为SEMCTL时，对应int semctl(int semid，int semnum，int cmd，union semun arg)调用；   
这些调用将在后面阐述。

注：本人不主张采用系统调用ipc()，而更倾向于采用系统V或者POSIX进程间通信API。原因已在Linux环境进程间通信（三）：消息队列中给出。

3、系统V信号灯API

系统V消息队列API只有三个，使用时需要包括几个头文件：

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h> |

1）int semget(key\_t key, int nsems, int semflg)   
参数key是一个键值，由ftok获得，唯一标识一个信号灯集，用法与msgget()中的key相同；参数nsems指定打开或者新创建的信号灯集中将包含信号灯的数目；semflg参数是一些标志位。参数key和semflg的取值，以及何时打开已有信号灯集或者创建一个新的信号灯集与msgget()中的对应部分相同，不再祥述。   
该调用返回与健值key相对应的信号灯集描述字。   
调用返回：成功返回信号灯集描述字，否则返回-1。   
注：如果key所代表的信号灯已经存在，且semget指定了IPC\_CREAT|IPC\_EXCL标志，那么即使参数nsems与原来信号灯的数目不等，返回的也是EEXIST错误；如果semget只指定了IPC\_CREAT标志，那么参数nsems必须与原来的值一致，在后面程序实例中还要进一步说明。

2）int semop(int semid, struct sembuf \*sops, unsigned nsops);   
semid是信号灯集ID，sops指向数组的每一个sembuf结构都刻画一个在特定信号灯上的操作。nsops为sops指向数组的大小。   
sembuf结构如下：

|  |
| --- |
| struct sembuf {  unsigned short sem\_num; /\* semaphore index in array \*/  short sem\_op; /\* semaphore operation \*/  short sem\_flg; /\* operation flags \*/  }; |

sem\_num对应信号集中的信号灯，0对应第一个信号灯。sem\_flg可取IPC\_NOWAIT以及SEM\_UNDO两个标志。如果设置了SEM\_UNDO标志，那么在进程结束时，相应的操作将被取消，这是比较重要的一个标志位。如果设置了该标志位，那么在进程没有释放共享资源就退出时，内核将代为释放。如果为一个信号灯设置了该标志，内核都要分配一个sem\_undo结构来记录它，为的是确保以后资源能够安全释放。事实上，如果进程退出了，那么它所占用就释放了，但信号灯值却没有改变，此时，信号灯值反映的已经不是资源占有的实际情况，在这种情况下，问题的解决就靠内核来完成。这有点像僵尸进程，进程虽然退出了，资源也都释放了，但内核进程表中仍然有它的记录，此时就需要父进程调用waitpid来解决问题了。   
sem\_op的值大于0，等于0以及小于0确定了对sem\_num指定的信号灯进行的三种操作。具体请参考linux相应手册页。   
这里需要强调的是semop同时操作多个信号灯，在实际应用中，对应多种资源的申请或释放。semop保证操作的原子性，这一点尤为重要。尤其对于多种资源的申请来说，要么一次性获得所有资源，要么放弃申请，要么在不占有任何资源情况下继续等待，这样，一方面避免了资源的浪费；另一方面，避免了进程之间由于申请共享资源造成死锁。   
也许从实际含义上更好理解这些操作：信号灯的当前值记录相应资源目前可用数目；sem\_op>0对应相应进程要释放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相当于进程要申请-sem\_op个共享资源。再联想操作的原子性，更不难理解该系统调用何时正常返回，何时睡眠等待。   
调用返回：成功返回0，否则返回-1。

3) int semctl(int semid，int semnum，int cmd，union semun arg)   
该系统调用实现对信号灯的各种控制操作，参数semid指定信号灯集，参数cmd指定具体的操作类型；参数semnum指定对哪个信号灯操作，只对几个特殊的cmd操作有意义；arg用于设置或返回信号灯信息。   
该系统调用详细信息请参见其手册页，这里只给出参数cmd所能指定的操作。

|  |  |
| --- | --- |
| IPC\_STAT | 获取信号灯信息，信息由arg.buf返回； |
| IPC\_SET | 设置信号灯信息，待设置信息保存在arg.buf中（在manpage中给出了可以设置哪些信息）； |
| GETALL | 返回所有信号灯的值，结果保存在arg.array中，参数sennum被忽略； |
| GETNCNT | 返回等待semnum所代表信号灯的值增加的进程数，相当于目前有多少进程在等待semnum代表的信号灯所代表的共享资源； |
| GETPID | 返回最后一个对semnum所代表信号灯执行semop操作的进程ID； |
| GETVAL | 返回semnum所代表信号灯的值； |
| GETZCNT | 返回等待semnum所代表信号灯的值变成0的进程数； |
| SETALL | 通过arg.array更新所有信号灯的值；同时，更新与本信号集相关的semid\_ds结构的sem\_ctime成员； |
| SETVAL | 设置semnum所代表信号灯的值为arg.val； |

调用返回：调用失败返回-1，成功返回与cmd相关：

|  |  |
| --- | --- |
| Cmd | return value |
| GETNCNT | Semncnt |
| GETPID | Sempid |
| GETVAL | Semval |
| GETZCNT | Semzcnt |

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part4/#ibm-pcon)

**五、信号灯的限制**

1、 一次系统调用semop可同时操作的信号灯数目SEMOPM，semop中的参数nsops如果超过了这个数目，将返回E2BIG错误。SEMOPM的大小特定与系统，redhat 8.0为32。

2、 信号灯的最大数目：SEMVMX，当设置信号灯值超过这个限制时，会返回ERANGE错误。在redhat 8.0中该值为32767。

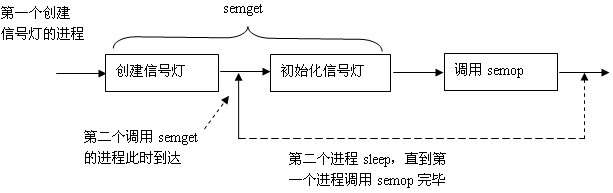
3、 系统范围内信号灯集的最大数目SEMMNI以及系统范围内信号灯的最大数目SEMMNS。超过这两个限制将返回ENOSPC错误。redhat 8.0中该值为32000。

4、 每个信号灯集中的最大信号灯数目SEMMSL，redhat 8.0中为250。 SEMOPM以及SEMVMX是使用semop调用时应该注意的；SEMMNI以及SEMMNS是调用semget时应该注意的。SEMVMX同时也是semctl调用应该注意的。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part4/#ibm-pcon)

**六、竞争问题**

第一个创建信号灯的进程同时也初始化信号灯，这样，系统调用semget包含了两个步骤：创建信号灯；初始化信号灯。由此可能导致一种竞争状态：第一个创建信号灯的进程在初始化信号灯时，第二个进程又调用semget，并且发现信号灯已经存在，此时，第二个进程必须具有判断是否有进程正在对信号灯进行初始化的能力。在参考文献[1]中，给出了绕过这种竞争状态的方法：当semget创建一个新的信号灯时，信号灯结构semid\_ds的sem\_otime成员初始化后的值为0。因此，第二个进程在成功调用semget后，可再次以IPC\_STAT命令调用semctl，等待sem\_otime变为非0值，此时可判断该信号灯已经初始化完毕。下图描述了竞争状态产生及解决方法：



实际上，这种解决方法也是基于这样一个假定：第一个创建信号灯的进程必须调用semop，这样sem\_otime才能变为非零值。另外，因为第一个进程可能不调用semop，或者semop操作需要很长时间，第二个进程可能无限期等待下去，或者等待很长时间。

[**回页首**](http://www.ibm.com/developerworks/cn/linux/l-ipc/part4/#ibm-pcon)

**七、信号灯应用实例**

本实例有两个目的：1、获取各种信号灯信息；2、利用信号灯实现共享资源的申请和释放。并在程序中给出了详细注释。

|  |
| --- |
| #include <linux/sem.h>  #include <stdio.h>  #include <errno.h>  #define SEM\_PATH "/unix/my\_sem"  #define max\_tries 3  int semid;  main()  {  int flag1,flag2,key,i,init\_ok,tmperrno;  struct semid\_ds sem\_info;  struct seminfo sem\_info2;  union semun arg; //union semun： 请参考附录2  struct sembuf askfor\_res, free\_res;  flag1=IPC\_CREAT|IPC\_EXCL|00666;  flag2=IPC\_CREAT|00666;  key=ftok(SEM\_PATH,'a');  //error handling for ftok here;  init\_ok=0;  semid=semget(key,1,flag1);  //create a semaphore set that only includes one semphore.  if(semid<0)  {  tmperrno=errno;  perror("semget");  if(tmperrno==EEXIST)  //errno is undefined after a successful library call( including perror call)  //so it is saved in tmperrno.  {  semid=semget(key,1,flag2);  //flag2 只包含了IPC\_CREAT标志, 参数nsems(这里为1)必须与原来的信号灯数目一致  arg.buf=&sem\_info;  for(i=0; i<max\_tries; i++)  {  if(semctl(semid, 0, IPC\_STAT, arg)==-1)  { perror("semctl error"); i=max\_tries;}  else  {  if(arg.buf->sem\_otime!=0){ i=max\_tries; init\_ok=1;}  else sleep(1);  }  }  if(!init\_ok)  // do some initializing, here we assume that the first process that creates the sem  // will finish initialize the sem and run semop in max\_tries\*1 seconds. else it will  // not run semop any more.  {  arg.val=1;  if(semctl(semid,0,SETVAL,arg)==-1) perror("semctl setval error");  }  }  else  {perror("semget error, process exit"); exit(); }  }  else //semid>=0; do some initializing  {  arg.val=1;  if(semctl(semid,0,SETVAL,arg)==-1)  perror("semctl setval error");  }  //get some information about the semaphore and the limit of semaphore in redhat8.0  arg.buf=&sem\_info;  if(semctl(semid, 0, IPC\_STAT, arg)==-1)  perror("semctl IPC STAT");  printf("owner's uid is %d\n", arg.buf->sem\_perm.uid);  printf("owner's gid is %d\n", arg.buf->sem\_perm.gid);  printf("creater's uid is %d\n", arg.buf->sem\_perm.cuid);  printf("creater's gid is %d\n", arg.buf->sem\_perm.cgid);  arg.\_\_buf=&sem\_info2;  if(semctl(semid,0,IPC\_INFO,arg)==-1)  perror("semctl IPC\_INFO");  printf("the number of entries in semaphore map is %d \n", arg.\_\_buf->semmap);  printf("max number of semaphore identifiers is %d \n", arg.\_\_buf->semmni);  printf("mas number of semaphores in system is %d \n", arg.\_\_buf->semmns);  printf("the number of undo structures system wide is %d \n", arg.\_\_buf->semmnu);  printf("max number of semaphores per semid is %d \n", arg.\_\_buf->semmsl);  printf("max number of ops per semop call is %d \n", arg.\_\_buf->semopm);  printf("max number of undo entries per process is %d \n", arg.\_\_buf->semume);  printf("the sizeof of struct sem\_undo is %d \n", arg.\_\_buf->semusz);  printf("the maximum semaphore value is %d \n", arg.\_\_buf->semvmx);    //now ask for available resource:  askfor\_res.sem\_num=0;  askfor\_res.sem\_op=-1;  askfor\_res.sem\_flg=SEM\_UNDO;    if(semop(semid,&askfor\_res,1)==-1)//ask for resource  perror("semop error");    sleep(3);  //do some handling on the sharing resource here, just sleep on it 3 seconds  printf("now free the resource\n");    //now free resource  free\_res.sem\_num=0;  free\_res.sem\_op=1;  free\_res.sem\_flg=SEM\_UNDO;  if(semop(semid,&free\_res,1)==-1)//free the resource.  if(errno==EIDRM)  printf("the semaphore set was removed\n");  //you can comment out the codes below to compile a different version:  if(semctl(semid, 0, IPC\_RMID)==-1)  perror("semctl IPC\_RMID");  else printf("remove sem ok\n");  } |

注：读者可以尝试一下注释掉初始化步骤，进程在运行时会出现何种情况（进程在申请资源时会睡眠），同时可以像程序结尾给出的注释那样，把该程序编译成两个不同版本。下面是本程序的运行结果（操作系统redhat8.0）：

|  |
| --- |
| owner's uid is 0  owner's gid is 0  creater's uid is 0  creater's gid is 0  the number of entries in semaphore map is 32000  max number of semaphore identifiers is 128  mas number of semaphores in system is 32000  the number of undo structures system wide is 32000  max number of semaphores per semid is 250  max number of ops per semop call is 32  max number of undo entries per process is 32  the sizeof of struct sem\_undo is 20  the maximum semaphore value is 32767  now free the resource  remove sem ok |

Summary：信号灯与其它进程间通信方式有所不同，它主要用于进程间同步。通常所说的系统V信号灯实际上是一个信号灯的集合，可用于多种共享资源的进程间同步。每个信号灯都有一个值，可以用来表示当前该信号灯代表的共享资源可用（available）数量，如果一个进程要申请共享资源，那么就从信号灯值中减去要申请的数目，如果当前没有足够的可用资源，进程可以睡眠等待，也可以立即返回。当进程要申请多种共享资源时，linux可以保证操作的原子性，即要么申请到所有的共享资源，要么放弃所有资源，这样能够保证多个进程不会造成互锁。Linux对信号灯有各种各样的限制，程序中给出了输出结果。另外，如果读者想对信号灯作进一步的理解，建议阅读sem.h源代码，该文件不长，但给出了信号灯相关的重要数据结构。

附录1： struct sem\_array如下：

|  |
| --- |
| /\*系统中的每个信号灯集对应一个sem\_array 结构 \*/  struct sem\_array {  struct kern\_ipc\_perm sem\_perm; /\* permissions .. see ipc.h \*/  time\_t sem\_otime; /\* last semop time \*/  time\_t sem\_ctime; /\* last change time \*/  struct sem \*sem\_base; /\* ptr to first semaphore in array \*/  struct sem\_queue \*sem\_pending; /\* pending operations to be processed \*/  struct sem\_queue \*\*sem\_pending\_last; /\* last pending operation \*/  struct sem\_undo \*undo; /\* undo requests on this array \*/  unsigned long sem\_nsems; /\* no. of semaphores in array \*/  }; |

其中，sem\_queue结构如下：

|  |
| --- |
| /\* 系统中每个因为信号灯而睡眠的进程，都对应一个sem\_queue结构\*/  struct sem\_queue {  struct sem\_queue \* next; /\* next entry in the queue \*/  struct sem\_queue \*\* prev;  /\* previous entry in the queue, \*(q->prev) == q \*/  struct task\_struct\* sleeper; /\* this process \*/  struct sem\_undo \* undo; /\* undo structure \*/  int pid; /\* process id of requesting process \*/  int status; /\* completion status of operation \*/  struct sem\_array \* sma; /\* semaphore array for operations \*/  int id; /\* internal sem id \*/  struct sembuf \* sops; /\* array of pending operations \*/  int nsops; /\* number of operations \*/  int alter; /\* operation will alter semaphore \*/  }; |

附录2：union semun是系统调用semctl中的重要参数：

|  |
| --- |
| 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 \*/ //test!!  void \*\_\_pad;  };  struct seminfo {  int semmap;  int semmni;  int semmns;  int semmnu;  int semmsl;  int semopm;  int semume;  int semusz;  int semvmx;  int semaem;  }; |

**参考资料**

[1] UNIX网络编程第二卷：进程间通信，作者：W.Richard Stevens，译者：杨继张，清华大学出版社。对POSIX以及系统V信号灯都有阐述，对Linux环境下的程序开发有极大的启发意义。

[2] linux内核源代码情景分析（上），毛德操、胡希明著，浙江大学出版社，给出了系统V信号灯相关的源代码分析，尤其在阐述保证操作原子性方面，以及阐述undo标志位时，讨论的很深刻。

[3]GNU/Linux编程指南，第二版，Kurt Wall等著，张辉译

[4]semget、semop、semctl手册