**http://kenby.iteye.com/blog/1165042**

[**Linux进程通信 之 信号灯(semphore)**](http://kenby.iteye.com/blog/1165042)

[linux](http://www.iteye.com/blogs/tag/linux)[信号灯](http://www.iteye.com/blogs/tag/%E4%BF%A1%E5%8F%B7%E7%81%AF)[进程同步](http://www.iteye.com/blogs/tag/%E8%BF%9B%E7%A8%8B%E5%90%8C%E6%AD%A5)

**一. 信号灯简介**

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）。

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。

**二. 信号灯的基本操作**

对信号灯的操作无非有下面三种类型：

1、打开或创建信号灯

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

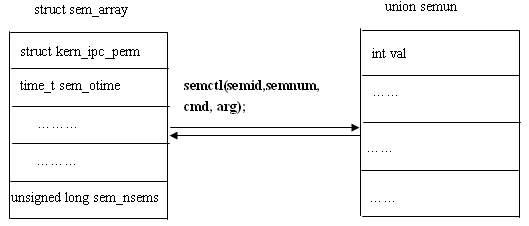
    semop系统调用。

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。



**三、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.

参数nsems指定信号灯集包含信号灯的数目；

semflg参数是一些标志位。

该调用返回与健值key相对应的信号灯集id

调用返回：成功返回信号灯集描述字，否则返回-1。

2）int semop(int semid, struct sembuf \*sops, unsigned nsops);

semid是信号灯集ID，sops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对应集合中的信号灯，0对应第一个信号灯, 以此类推...

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。

这里需要强调的是semop可以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.

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

semctl函数使用到的结构体:

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 \*/   //test!!

void \*\_\_pad;

};

struct  seminfo {

int semmap;

int semmni;

int semmns;

int semmnu;

int semmsl;

int semopm;

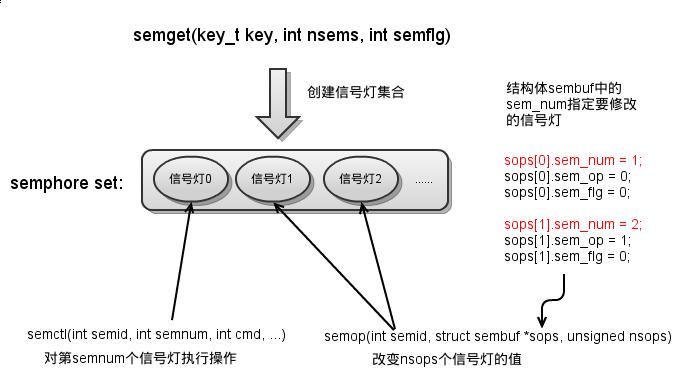
int semume;

int semusz;

int semvmx;

int semaem;

};



**四、范例**

这个范例使用信号灯来同步共享内存的操作, 程序创建一块共享内存, 然后父子进程共同

修改共享内存. 父子进程采用信号灯来同步操作.

C代码 [复制代码](http://kenby.iteye.com/blog/1165042) [[收藏代码http://kenby.iteye.com/images/spinner.gif](javascript:void())](javascript:void())

1. #include <stdio.h>
2. #include <sys/types.h>
3. #include <sys/ipc.h>
4. #include <sys/sem.h>
6. #define SHM\_KEY 0x33
7. #define SEM\_KEY 0x44
9. union semun {
10. **int** val;
11. struct semid\_ds \*buf;
12. unsigned **short** \*array;
13. };
15. **int** P(**int** semid)
16. {
17. struct sembuf sb;
18. sb.sem\_num = 0;
19. sb.sem\_op = -1;
20. sb.sem\_flg = SEM\_UNDO;
22. if(semop(semid, &sb, 1) == -1) {
23. perror("semop");
24. return -1;
25. }
26. return 0;
27. }
29. **int** V(**int** semid)
30. {
31. struct sembuf sb;
32. sb.sem\_num = 0;
33. sb.sem\_op = 1;
34. sb.sem\_flg = SEM\_UNDO;
36. if(semop(semid, &sb, 1) == -1) {
37. perror("semop");
38. return -1;
39. }
40. return 0;
41. }
43. **int** main(**int** argc, **char** \*\*argv)
44. {
45. pid\_t pid;
46. **int** i, shmid, semid;
47. **int** \*ptr;
48. union semun semopts;
50. /\* 创建一块共享内存, 存一个int变量 \*/
51. if ((shmid = shmget(SHM\_KEY, sizeof(**int**), IPC\_CREAT | 0600)) == -1) {
52. perror("msgget");
53. }
55. /\* 将共享内存映射到进程, fork后子进程可以继承映射 \*/
56. if ((ptr = (**int** \*)shmat(shmid, NULL, 0)) == (void \*)-1) {
57. perror("shmat");
58. }
59. \*ptr = 0;
61. /\* 创建一个信号量用来同步共享内存的操作 \*/
62. if ((semid = semget(SEM\_KEY, 1, IPC\_CREAT | 0600)) == -1) {
63. perror("semget");
64. }
66. /\* 初始化信号量 \*/
67. semopts.val = 1;
68. if (semctl(semid, 0, SETVAL, semopts) < 0) {
69. perror("semctl");
70. }
72. if ((pid = fork()) < 0) {
73. perror("fork");
74. } else if (pid == 0) {      /\* Child \*/
75. /\* 子进程对共享内存加1 \*/
76. for (i = 0; i < 100000; i++) {
77. P(semid);
78. (\*ptr)++;
79. V(semid);
80. printf("child: %d\n", \*ptr);
81. }
82. } else {                    /\* Parent \*/
83. /\* 父进程对共享内存减1 \*/
84. for (i = 0; i < 100000; i++) {
85. P(semid);
86. (\*ptr)--;
87. V(semid);
88. printf("parent: %d\n", \*ptr);
89. }
90. waitpid(pid);
91. /\* 如果同步成功, 共享内存的值为0 \*/
92. printf("finally: %d\n", \*ptr);
93. }
95. return 0;
96. }

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#define SHM\_KEY 0x33

#define SEM\_KEY 0x44

union semun {

int val;

struct semid\_ds \*buf;

unsigned short \*array;

};

int P(int semid)

{

struct sembuf sb;

sb.sem\_num = 0;

sb.sem\_op = -1;

sb.sem\_flg = SEM\_UNDO;

if(semop(semid, &sb, 1) == -1) {

perror("semop");

return -1;

}

return 0;

}

int V(int semid)

{

struct sembuf sb;

sb.sem\_num = 0;

sb.sem\_op = 1;

sb.sem\_flg = SEM\_UNDO;

if(semop(semid, &sb, 1) == -1) {

perror("semop");

return -1;

}

return 0;

}

int main(int argc, char \*\*argv)

{

pid\_t pid;

int i, shmid, semid;

int \*ptr;

union semun semopts;

/\* 创建一块共享内存, 存一个int变量 \*/

if ((shmid = shmget(SHM\_KEY, sizeof(int), IPC\_CREAT | 0600)) == -1) {

perror("msgget");

}

/\* 将共享内存映射到进程, fork后子进程可以继承映射 \*/

if ((ptr = (int \*)shmat(shmid, NULL, 0)) == (void \*)-1) {

perror("shmat");

}

\*ptr = 0;

/\* 创建一个信号量用来同步共享内存的操作 \*/

if ((semid = semget(SEM\_KEY, 1, IPC\_CREAT | 0600)) == -1) {

perror("semget");

}

/\* 初始化信号量 \*/

semopts.val = 1;

if (semctl(semid, 0, SETVAL, semopts) < 0) {

perror("semctl");

}

if ((pid = fork()) < 0) {

perror("fork");

} else if (pid == 0) { /\* Child \*/

/\* 子进程对共享内存加1 \*/

for (i = 0; i < 100000; i++) {

P(semid);

(\*ptr)++;

V(semid);

printf("child: %d\n", \*ptr);

}

} else { /\* Parent \*/

/\* 父进程对共享内存减1 \*/

for (i = 0; i < 100000; i++) {

P(semid);

(\*ptr)--;

V(semid);

printf("parent: %d\n", \*ptr);

}

waitpid(pid);

/\* 如果同步成功, 共享内存的值为0 \*/

printf("finally: %d\n", \*ptr);

}

return 0;

}

**五. 区别 System V信号量和Posix信号量**

信号量有两种实现：传统的System V信号量和新的POSIX信号量。它们所提供的函数很容易被区分：对于所有System V信号量函数，在它们的名字里面没有下划线。例如，应该是semget()而不是sem\_get()。然而，所有的的POSIX信号量函数都有一个下划线。下面列出了它们提供的所有函数清单：

|  |  |
| --- | --- |
| **Systm V** | **POSIX** |
| *semctl()* | sem\_getvalue() |
| *semget()* | sem\_post() |
| *semop()* | sem\_timedwait() |
|  | sem\_trywait() |
|  | sem\_wait() |
|  |  |
|  | sem\_destroy() |
|  | sem\_init() |
|  |  |
|  | sem\_close() |
|  | sem\_open() |
|  | sem\_unlink() |

另外一个区别是，对于POSIX信号量，你可以有命名的信号量，例如，信号量有一个文件

关联它们，

对于最后三个函数，被用来创建，关闭和删除这样一个命名的信号量。

而sem\_init()和sem\_destroy()仅仅供非命名信号量使用。

他们是有关信号量的两组程序设计接口函数。POSIX信号量来源于POSIX技术规范的实时

扩展方案(POSIX Realtime Extension)，常用于线程；system v信号量，常用于进程的同步。

这两者非常相近，但它们使用的函数调用各不相同。前一种的头文件为semaphore.h，函数

调用为sem\_init(),sem\_wait(),sem\_post(),sem\_destory()等等。后一种头文件为<sys/sem.h>,

函数调用为semctl(),semget(),semop()等函数。

更详细地请看 man sem\_overview

总结:

System V的信号量一般用于进程同步, 且是内核持续的, api为

semget

semctl

semop

Posix的有名信号量一般用于进程同步, 有名信号量是内核持续的. 有名信号量的api为

sem\_open

sem\_close

sem\_unlink

Posix的无名信号量一般用于线程同步, 无名信号量是进程持续的, 无名信号量的api为

sem\_init

sem\_destroy

下面一个范例使用Posix的有名信号量来同步父子进程的共享内存操作:

C代码 [复制代码](http://kenby.iteye.com/blog/1165042) [[收藏代码http://kenby.iteye.com/images/spinner.gif](javascript:void())](javascript:void())

1. #include <stdio.h>
2. #include <sys/types.h>
3. #include <sys/ipc.h>
4. #include <semaphore.h>
5. #include <fcntl.h>           /\* For O\_\* constants \*/
6. #include <sys/stat.h>        /\* For mode constants \*/
7. #include <stdlib.h>
9. #define SHM\_KEY 0x33
11. **int** main(**int** argc, **char** \*\*argv)
12. {
13. pid\_t pid;
14. **int** i, shmid;
15. **int** \*ptr;
16. sem\_t \*sem;
18. /\* 创建一块共享内存, 存一个int变量 \*/
19. if ((shmid = shmget(SHM\_KEY, sizeof(**int**), IPC\_CREAT | 0600)) == -1) {
20. perror("msgget");
21. }
23. /\* 将共享内存映射到进程, fork后子进程可以继承映射 \*/
24. if ((ptr = (**int** \*)shmat(shmid, NULL, 0)) == (void \*)-1) {
25. perror("shmat");
26. }
27. \*ptr = 0;
29. /\* posix的有名信号量是kernel persistent的
30. \* 调用sem\_unlink删除以前的信号量 \*/
31. sem\_unlink("/mysem");
33. /\* 创建新的信号量, 初值为1, sem\_open会创建共享内存
34. \* 所以信号量是内核持续的 \*/
35. if ((sem = sem\_open("/mysem", O\_CREAT, 0600, 1)) == SEM\_FAILED) {
36. perror("sem\_open");
37. }
39. if ((pid = fork()) < 0) {
40. perror("fork");
41. } else if (pid == 0) {      /\* Child \*/
42. /\* 子进程对共享内存加1 \*/
43. for (i = 0; i < 100000; i++) {
44. sem\_wait(sem);
45. (\*ptr)++;
46. sem\_post(sem);
47. printf("child: %d\n", \*ptr);
48. }
49. } else {                    /\* Parent \*/
50. /\* 父进程对共享内存减1 \*/
51. for (i = 0; i < 100000; i++) {
52. sem\_wait(sem);
53. (\*ptr)--;
54. sem\_post(sem);
55. printf("parent: %d\n", \*ptr);
56. }
57. waitpid(pid);
58. /\* 如果同步成功, 共享内存的值为0 \*/
59. printf("finally: %d\n", \*ptr);
60. sem\_unlink("/mysem");
61. }
63. return 0;
64. }

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <semaphore.h>

#include <fcntl.h> /\* For O\_\* constants \*/

#include <sys/stat.h> /\* For mode constants \*/

#include <stdlib.h>

#define SHM\_KEY 0x33

int main(int argc, char \*\*argv)

{

pid\_t pid;

int i, shmid;

int \*ptr;

sem\_t \*sem;

/\* 创建一块共享内存, 存一个int变量 \*/

if ((shmid = shmget(SHM\_KEY, sizeof(int), IPC\_CREAT | 0600)) == -1) {

perror("msgget");

}

/\* 将共享内存映射到进程, fork后子进程可以继承映射 \*/

if ((ptr = (int \*)shmat(shmid, NULL, 0)) == (void \*)-1) {

perror("shmat");

}

\*ptr = 0;

/\* posix的有名信号量是kernel persistent的

\* 调用sem\_unlink删除以前的信号量 \*/

sem\_unlink("/mysem");

/\* 创建新的信号量, 初值为1, sem\_open会创建共享内存

\* 所以信号量是内核持续的 \*/

if ((sem = sem\_open("/mysem", O\_CREAT, 0600, 1)) == SEM\_FAILED) {

perror("sem\_open");

}

if ((pid = fork()) < 0) {

perror("fork");

} else if (pid == 0) { /\* Child \*/

/\* 子进程对共享内存加1 \*/

for (i = 0; i < 100000; i++) {

sem\_wait(sem);

(\*ptr)++;

sem\_post(sem);

printf("child: %d\n", \*ptr);

}

} else { /\* Parent \*/

/\* 父进程对共享内存减1 \*/

for (i = 0; i < 100000; i++) {

sem\_wait(sem);

(\*ptr)--;

sem\_post(sem);

printf("parent: %d\n", \*ptr);

}

waitpid(pid);

/\* 如果同步成功, 共享内存的值为0 \*/

printf("finally: %d\n", \*ptr);

sem\_unlink("/mysem");

}

return 0;

}