

## qinxiongxu的专栏

目录视图

摘要视图

RSS 订阅

## 个人资料



qinxiongxu

访问: 114707次

积分: 1038

等级: BLOG &gt; 4

排名: 千里之外

原创: 17篇

转载: 28篇

译文: 0篇

评论: 39条

## 文章搜索

## 文章分类

生活点滴 (5)

嵌入式linux开发 (29)

Android开发 (3)

工作点滴 (1)

旗舰版stm32专区 (2)

uCOSII专区 (1)

u-boot专区 (2)

ARM汇编专区 (1)

S5PV210专区 (1)

## 文章存档

2014年01月 (1)

2013年09月 (3)

2013年08月 (3)

2013年07月 (1)

2013年06月 (2)

展开

## 阅读排行

最全面的linux信号量解析

linux强大的游戏模拟器-- (53161)  
(5559)

SD初始化过程以及Cmd (4893)

基于GPL329xx linux平台 (3859)

ARM Scatter File详解 (3488)

[聚焦行业最佳实践，BDTC 2016完整议程公布](#) [微信小程序实战项目——点餐系统](#) [程序员11月书讯，评论得书啦](#) [Get IT技能知识库，50个领域一键直达](#)

## 最全面的linux信号量解析

标签: linux semaphore thread struct system null

2016-11-25 19:42

53215人阅读

评论(9)

收藏

举报

分类: 嵌入式linux开发 (28)

2012-06-28 15:08 285人阅读 评论(0) 收藏 编辑 删除

## 信号量

## 一. 什么是信号量

信号量的使用主要是用来保护共享资源，使得资源在一个时刻只有一个进程（线程）所拥有。

信号量的值为正的时候，说明它空闲。所测试的线程可以锁定而使用它。若为0，说明它被占用，测试的线程要进入睡眠队列中，等待被唤醒。

## 二. 信号量的分类

在学习信号量之前，我们必须先知道——Linux提供两种信号量：

(1) 内核信号量，由内核控制路径使用

(2) 用户态进程使用的信号量，这种信号量又分为POSIX信号量和SYSTEM

V信号量。

POSIX信号量又分为有名信号量和无名信号量。

有名信号量，其值保存在文件中，所以它可以用于线程也可以用于进程间的同步。无名信号量，其值保存在内存中。

倘若对信号量没有以上的全面认识的话，你就会很快发现自己在信号量的森林里迷失了方向。

## 三. 内核信号量

## 1. 内核信号量的构成

内核信号量类似于自旋锁，因为当锁关闭着时，它不允许内核控制路径继续进行。然而，当内核控制路径试图获取内核信号量锁保护的忙资源时，相应的进程就被挂起。只有在资源被释放时，进程才再次变为可运行。

只有可以睡眠的函数才能获取内核信号量；中断处理程序和可延迟函数都不能使用内核信号量。

内核信号量是struct semaphore类型的对象，它在<asm/semaphore.h>中定义：

```
struct semaphore {
    atomic_t count;
    int sleepers;
    wait_queue_head_t wait;
}
```

count: 相当于信号量的值，大于0，资源空闲；等于0，资源忙，但没有进程等待这个保护的资源；小于0，资源不可用，并至少有一个进程等待资源。

wait: 存放等待队列链表的地址，当前等待资源的所有睡眠进程都会放在这个链表中。

sleepers: 存放一个标志，表示是否有一些进程在信号量上睡眠。

## 2. 内核信号量中的等待队列（删除，没有联系）

linux时间函数gettimeofday: (3326)
Linux下RTC时间的读写: (3154)
ucos ii 46个系统API函数 (2865)
旗舰版stm32开发板介绍: (2394)
Windows下用fastboot烧: (2372)

## 评论排行

基于Android平台下的科: (17)
最全面的linux信号量解析 (9)
基于GPL329xx linux平台 (4)
旗舰版stm32开发板介绍: (3)
2012 (2)
基于嵌入式linux2.6平台: (2)
ucos ii 46个系统API函数 (2)
windows下安装SVN (0)
Android驱动 (0)
HQ2416 BASE终于诞生 (0)

## 推荐文章

- \* RxJava详解，由浅入深
- \* 倍升工作效率的小策略
- \* Android热修复框架AndFix原理解析及使用
- \* “区块链”究竟是什么鬼
- \* 架构设计：系统存储-MySQL主从方案业务连接透明化（中）

## 最新评论

- 最全面的linux信号量解析  
wujinting007: 最后的例子中  
empty\_sem\_mutex  
(p\_sem\_mutex)  
full\_sem\_mutex...
- 最全面的linux信号量解析  
wujinting007: 最后的例子中  
empty\_sem\_mutex  
(p\_sem\_mutex)  
full\_sem\_mutex...
- 最全面的linux信号量解析  
wujinting007: 最后的例子中  
empty\_sem\_mutex  
(p\_sem\_mutex)  
full\_sem\_mutex...
- 最全面的linux信号量解析  
wujinting007: 最后的例子中  
empty\_sem\_mutex  
(p\_sem\_mutex)  
full\_sem\_mutex...
- 最全面的linux信号量解析  
hn\_lgc: 好像不是在一个时刻只  
有一个进程（线程）所拥有啊，  
信号量是一个整数，比如从3递  
减到0，应该是同时可以被...
- ucos ii 46个系统API函数解析 .  
li1373740149: {\^o^}/~
- 最全面的linux信号量解析  
elikang: @u011388486:没有  
吧，你再仔细理解一下哈
- 最全面的linux信号量解析  
appleckie: 请问第一句话是不是  
解释反了
- 最全面的linux信号量解析  
Kevin\_Smart: 学习了
- ucos ii 46个系统API函数解析 .  
xiahui45: 很详细,非常适合我们  
UCOS入门

上面已经提到了内核信号量使用了等待队列wait\_queue来实现阻塞操作。

当某任务由于没有某种条件没有得到满足时，它就被挂到等待队列中睡眠。当条件得到满足时，该任务就被移出等待队列，此时并不意味着该任务就被马上执行，因为它又被移进工作队列中等待CPU资源，在适当的时机被调度。

内核信号量是在内部使用等待队列的，也就是说该等待队列对用户是隐藏的，无须用户干涉。由用户真正使用的等待队列我们将在另外的篇章进行详解。

## 3. 内核信号量的相关函数

（1）初始化：

void sema\_init (struct semaphore \*sem, int val);

void init\_MUTEX (struct semaphore \*sem); //将sem的值为1，表示资源空闲

void init\_MUTEX\_LOCKED (struct semaphore \*sem); //将sem的值为0，表示资源忙

（2）申请内核信号量所保护的资源：

void down(struct semaphore \* sem); // 可引起睡眠

int down\_interruptible(struct semaphore \* sem); // down\_interruptible能被信号打断

int down\_trylock(struct semaphore \* sem); // 非阻塞函数，不会睡眠。无法锁定资源则马上返回

（3）释放内核信号量所保护的资源：

void up(struct semaphore \* sem);

## 4. 内核信号量的使用例程

在驱动程序中，当多个线程同时访问相同的资源时（驱动中的全局变量时一种典型的共享资源），可能会引发“竞态”，因此我们必须对共享资源进行并发控制。Linux内核中解决并发控制的最常用方法是自旋锁与信号量（绝大多数时候作为互斥锁使用）。

ssize\_t globalvar\_write(struct file \*filp, const char \*buf, size\_t len, loff\_t \*off)

```
{  
    //获得信号量  
    if (down_interruptible(&sem))  
    {  
        return - ERESTARTSYS;  
    }  
    //将用户空间的数据复制到内核空间的global_var  
    if (copy_from_user(&global_var, buf, sizeof(int)))  
    {  
        up(&sem);  
        return - EFAULT;  
    }  
    //释放信号量  
    up(&sem);  
    return sizeof(int);  
}
```

## 四. POSIX 信号量与SYSTEM V信号量的比较

1. 对POSIX来说，信号量是个非负整数。常用于线程间同步。

而SYSTEM V信号量则是一个或多个信号量的集合，它对应的是一个信号量结构体，这个结构体是为SYSTEM V IPC服务的，信号量只不过是它的一部分。常用于进程间同步。

2. POSIX信号量的引用头文件是“<semaphore.h>”，而SYSTEM V信号量的引用头文件是“<sys/sem.h>”。

3. 从使用的角度，System V信号量是复杂的，而Posix信号量是简单。比如，POSIX信号量的创建和初始化或PV操作就很非常方便。

## 五. POSIX信号量详解

### 1. 无名信号量

无名信号量的创建就像声明一般的变量一样简单，例如：`sem_t sem_id`。然后再初始化该无名信号量，之后就可以放心使用了。

无名信号量常用于多线程间的同步，同时也用于相关进程间的同步。也就是说，无名信号量必须是多个进程（线程）的共享变量，无名信号量要保护的变量也必须是多个进程（线程）的共享变量，这两个条件是缺一不可的。

常见的无名信号量相关函数：`sem_destroy`

`int sem_init(sem_t *sem, int pshared, unsigned int value);`

1)pshared==0 用于同一多线程的同步；



2)若`pshared>0` 用于多个相关进程间的同步（即由`fork`产生的）

```
int sem_getvalue(sem_t *sem, int *sval);
```

取回信号量`sem`的当前值，把该值保存到`sval`中。

若有1个或更多的线程或进程调用`sem_wait`阻塞在该信号量上，该函数返回两种值：

1) 返回0

2) 返回阻塞在该信号量上的进程或线程数目

linux采用返回的第一种策略。

`sem_wait`(或`sem_trywait`)相当于P操作，即申请资源。

```
int sem_wait(sem_t *sem); // 这是一个阻塞的函数
```

测试所指定信号量的值,它的操作是原子的。

若`sem>0`，那么它减1并立即返回。

若`sem==0`，则睡眠直到`sem>0`，此时立即减1，然后返回。

```
int sem_trywait(sem_t *sem); // 非阻塞的函数
```

其他的行为和`sem_wait`一样，除了：

若`sem==0`，不是睡眠，而是返回一个错误EAGAIN。

`sem_post`相当于V操作，释放资源。

```
int sem_post(sem_t *sem);
```

把指定的信号量`sem`的值加1；

唤醒正在等待该信号量的任意线程。

注意：在这些函数中，只有`sem_post`是信号安全的函数，它是可重入函数

（a）无名信号量在多线程间的同步

无名信号量的常见用法是将要保护的变量放在`sem_wait`和`sem_post`中间所形成的临界区内，这样该变量就会被保护起来，例如：

```
#include <pthread.h>
#include <semaphore.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int number; // 被保护的全局变量
sem_t sem_id;

void* thread_one_fun(void *arg)
{
    sem_wait(&sem_id);
    printf("thread_one have the semaphore\n");
    number++;
    printf("number = %d\n",number);
    sem_post(&sem_id);
}

void* thread_two_fun(void *arg)
{
    sem_wait(&sem_id);
    printf("thread_two have the semaphore\n");
    number--;
    printf("number = %d\n",number);
    sem_post(&sem_id);
}

int main(int argc,char *argv[])
{
    number = 1;
    pthread_t id1, id2;
    sem_init(&sem_id, 0, 1);
    pthread_create(&id1,NULL,thread_one_fun, NULL);
    pthread_create(&id2,NULL,thread_two_fun, NULL);
    pthread_join(id1,NULL);
    pthread_join(id2,NULL);
    printf("main,,, \n");
    return 0;
}
```

```
}
```

上面的例程，到底哪个线程先申请到信号量资源，这是随机的。如果想要某个特定的顺序的话，可以用2个信号量来实现。例如下面的例程是线程1先执行完，然后线程2才继续执行，直至结束。

```
int number; // 被保护的全局变量
sem_t sem_id1, sem_id2;
void* thread_one_fun(void *arg)
{
    sem_wait(&sem_id1);
    printf("thread_one have the semaphore\n");
    number++;
    printf("number = %d\n",number);
    sem_post(&sem_id2);
}
void* thread_two_fun(void *arg)
{
    sem_wait(&sem_id2);
    printf("thread_two have the semaphore\n");
    number--;
    printf("number = %d\n",number);
    sem_post(&sem_id1);
}
int main(int argc,char *argv[])
{
    number = 1;
    pthread_t id1, id2;
    sem_init(&sem_id1, 0, 1); // 空闲的
    sem_init(&sem_id2, 0, 0); // 忙的
    pthread_create(&id1,NULL,thread_one_fun, NULL);
    pthread_create(&id2,NULL,thread_two_fun, NULL);
    pthread_join(id1,NULL);
    pthread_join(id2,NULL);
    printf("main,,,\\n");
    return 0;
}
```

（b）无名信号量在相关进程间的同步

说是相关进程，是因为本程序中共有2个进程，其中一个是另外一个的子进程（由fork产生）的。

本来对于fork来说，子进程只继承了父进程的代码副本，mutex理应在父子进程中是相互独立的两个变量，但由于在初始化mutex的时候，由pshared = 1指定了mutex处于共享内存区域，所以此时mutex变成了父子进程共享的一个变量。此时，mutex就可以用来同步相关进程了。

```
#include <semaphore.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
int main(int argc, char **argv)
{
    int fd, i, count=0, nloop=10, zero=0, *ptr;
    sem_t mutex;
    //open a file and map it into memory
```

```

fd = open("log.txt",O_RDWR|O_CREAT,S_IRWXU);
write(fd,&zero,sizeof(int));
ptr = mmap( NULL,sizeof(int),PROT_READ |
PROT_WRITE,MAP_SHARED,fd,0 );
close(fd);
/* create, initialize semaphore */
if( sem_init(&mutex,1,1) < 0 )//
{
perror("semaphore initialization");
exit(0);
}
if (fork() == 0)
{ /* child process*/
for (i = 0; i < nloop; i++)
{
sem_wait(&mutex);
printf("child: %d\n", (*ptr)++);
sem_post(&mutex);
}
exit(0);
}
/* back to parent process */
for (i = 0; i < nloop; i++)
{
sem_wait(&mutex);
printf("parent: %d\n", (*ptr)++);
sem_post(&mutex);
}
exit(0);
}

```

## 2. 有名信号量

有名信号量的特点是把信号量的值保存在文件中。

这决定了它的用途非常广：既可以用于线程，也可以用于相关进程间，甚至是不相关进程。

（a）有名信号量能在进程间共享的原因

由于有名信号量的值是保存在文件中的，所以对于相关进程来说，子进程是继承了父进程的文件描述符，那么子进程所继承的文件描述符所指向的文件是和父进程一样的，当然文件里面保存的有名信号量值就共享了。

（b）有名信号量相关函数说明

有名信号量在使用的时候，和无名信号量共享sem\_wait和sem\_post函数。

区别是有名信号量使用sem\_open代替sem\_init，另外在结束的时候要像关闭文件

一样去关闭这个有名信号量。

(1)打开一个已存在的有名信号量，或创建并初始化一个有名信号量。一个单一的调用就完成了信号量的创建、初始化和权限的设置。

```
sem_t *sem_open(const char *name, int oflag, mode_t mode , int value);
```

name是文件的路径名；

Oflag 有O\_CREAT或O\_CREAT|EXCL两个取值；

mode\_t控制新的信号量的访问权限；

Value指定信号量的初始化值。

注意：

这里的name不能写成/tmp/aaa.sem这样的格式，因为在linux下，sem都是创建

在/dev/shm目录下。你可以将name写成"/mysem"或"mysem"，创建出来的文件都是

"/dev/shm/sem.mysem"，千万不要写路径。也千万不要写"/tmp/mysem"之类的。

当oflag = O\_CREAT时，若name指定的信号量不存在时，则会创建一个，而且后

面的mode和value参数必须有效。若name指定的信号量已存在，则直接打开该信号量，同时忽略mode和value参数。

当oflag = O\_CREAT|O\_EXCL时，若name指定的信号量已存在，该函数会直接返

回error。

(2) 一旦你使用了一信号量，销毁它们就变得很重要。

在做这个之前，要确定所有对这个有名信号量的引用都已经通过sem\_close()函数关闭了，然后只需在退出或是退出处理函数中调用sem\_unlink()去删除系统中的信号量，注意如果有任何的处理器或是线程引用这个信号量，sem\_unlink()函数不会起到任何的作用。

也就是说，必须是最后一个使用该信号量的进程来执行sem\_unlink才有效。因为每个信号灯有一个引用计数器记录当前的打开次数，sem\_unlink必须等待这个数为0时才能把name所指的信号灯从文件系统中删除。也就是要等待最后一个sem\_close发生。

(c) 有名信号量在无相关进程间的同步

前面已经说过，有名信号量是位于共享内存区的，那么它要保护的资源也必须是位于共享内存区，只有这样才能被无相关的进程所共享。

在下面这个例子中，服务进程和客户进程都使用shmget和shmat来获取一块共享内存资源。然后利用有名信号量来对这块共享内存资源进行互斥保护。

```
< u>File1: server.c </u>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define SHMSZ 27
char SEM_NAME[] = "vik";
int main()
{
    char ch;
    int shmid;
    key_t key;
    char *shm,*s;
    sem_t *mutex;

    //name the shared memory segment
    key = 1000;

    //create & initialize semaphore
    mutex = sem_open(SEM_NAME,O_CREAT,0644,1);
    if(mutex == SEM_FAILED)
    {
        perror("unable to create semaphore");
        sem_unlink(SEM_NAME);
        exit(-1);
    }

    //create the shared memory segment with this key
    shmid = shmget(key,SHMSZ,IPC_CREAT|0666);
    if(shmid<0)
    {
        perror("failure in shmget");
        exit(-1);
    }

    //attach this segment to virtual memory
    shm = shmat(shmid,NULL,0);

    //start writing into memory
    s = shm;
    for(ch='A';ch<='Z';ch++)
    {
        sem_wait(mutex);
        *s++ = ch;
```

```

sem_post(mutex);
}

//the below loop could be replaced by binary semaphore
while(*shm != "")
{
sleep(1);
}

sem_close(mutex);
sem_unlink(SEM_NAME);
shmctl(shmid, IPC_RMID, 0);
exit(0);
}

< u>File 2: client.c</u>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define SHMSZ 27
char SEM_NAME[] = "vik";
int main()
{
char ch;
int shmid;
key_t key;
char *shm,*s;
sem_t *mutex;
//name the shared memory segment
key = 1000;
//create & initialize existing semaphore
mutex = sem_open(SEM_NAME,0,0644,0);
if(mutex == SEM_FAILED)
{
perror("reader:unable to execute semaphore");
sem_close(mutex);
exit(-1);
}
//create the shared memory segment with this key
shmid = shmget(key,SHMSZ,0666);
if(shmid<0)
{
perror("reader:failure in shmget");
exit(-1);
}
//attach this segment to virtual memory
shm = shmat(shmid,NULL,0);
//start reading
s = shm;
for(s=shm;*s!=NULL;s++)
{
sem_wait(mutex);
putchar(*s);
sem_post(mutex);
}

```

```
//once done signal exiting of reader:This can be replaced by
another semaphore
*shm = "";
sem_close(mutex);
shmctl(shmid, IPC_RMID, 0);
exit(0);
}
```

## 六. SYSTEM V信号量

这是信号量值的集合，而不是单个信号量。相关的信号量操作函数由<sys/ipc.h>引用。

### 1. 信号量结构体

内核为每个信号量集维护一个信号量结构体，可在<sys/sem.h>找到该定义：

```
struct semid_ds {
struct ipc_perm sem_perm; /* 信号量集的操作许可权限 */
struct sem *sem_base; /* 某个信号量sem结构数组的指针，当前信号量集
中的每个信号量对应其中一个数组元素 */
ushort sem_nsems; /* sem_base 数组的个数 */
time_t sem_otime; /* 最后一次成功修改信号量数组的时间 */
time_t sem_ctime; /* 成功创建时间 */
};
struct sem {
ushort semval; /* 信号量的当前值 */
short sempid; /* 最后一次返回该信号量的进程ID 号 */
ushort semncnt; /* 等待semval大于当前值的进程个数 */
ushort semzcnt; /* 等待semval变成0的进程个数 */
};
```

### 2. 常见的SYSTEM V信号量函数

#### （a）关键字和描述符

SYSTEM V信号量是SYSTEM V IPC（即SYSTEM V进程间通信）的组成部分，其他的有SYSTEM V消息队列，SYSTEM V共享内存。而关键字和IPC描述符无疑是它们的共同点，也使用它们，就不得不先对它们进行熟悉。这里只对SYSTEM V信号量进行讨论。IPC描述符相当于引用ID号，要想使用SYSTEM V信号量（或MSG、SHM），就必须用IPC描述符来调用信号量。而IPC描述符是内核动态提供的（通过semget来获取），用户无法让服务器和客户事先认可共同使用哪个描述符，所以有时候就需要到关键字KEY来定位描述符。

某个KEY只会固定对应一个描述符（这项转换工作由内核完成），这样假如服务器和客户事先认可共同使用某个KEY，那么大家就都能定位到同一个描述符，也就能定位到同一个信号量，这样就达到了SYSTEM V信号量在进程间共享的目的。

#### （b）创建和打开信号量

```
int semget(key_t key, int nsems, int oflag)
```

(1) nsems>0：创建一个信号量集，指定集合中信号量的数量，一旦创建就不能更改。

(2) nsems==0：访问一个已存在的集合

(3) 返回的是一个称为信号量标识符的整数，semop和semctl函数将使用它。

(4) 创建成功后信号量结构被设置：

.sem\_perm 的uid和gid成员被设置成的调用进程的有效用户ID和有效组ID

.oflag 参数中的读写权限位存入sem\_perm.mode

.sem\_otime 被置为0,sem\_ctime被设置为当前时间

.sem\_nsems 被置为nsems参数的值

该集合中的每个信号量不初始化，这些结构是在semctl，用参数SET\_VAL，SETALL初始化的。

semget函数执行成功后，就产生了一个由内核维持的类型为semid\_ds结构体的信号量集，返回semid就是指向该信号量集的索引。

#### （c）关键字的获取

有多种方法使客户机和服务器在同一IPC结构上会合：

(1) 服务器可以指定关键字IPC\_PRIVATE创建一个新IPC结构，将返回的标识符存放在某处（例如一个文件）以便客户机取用。关键字 IPC\_PRIVATE保证服务器创建一个新IPC结构。这种技术的缺点是：服务器要将整型标识符写到文件中，然后客户机在此后又要读文件取得此标识符。



IPC\_PRIVATE关键字也可用于父、子关系进程。父进程指定 IPC\_PRIVATE创建一个新IPC结构，所返回的标识符在fork后可由子进程使用。子进程可将此标识符作为exec函数的一个参数传给一个新程序。

(2) 在一个公用头文件中定义一个客户机和服务器都认可的关键字。然后服务器指定此关键字创建一个新的IPC结构。这种方法的问题是关键字可能已与一个IPC结构相结合，在此情况下，get函数（msgget、semget或shmget）出错返回。服务器必须处理这一错误，删除已存在的IPC结构，然后试着再创建它。当然，这个关键字不能被别的程序所占用。

(3) 客户机和服务器认同一个路径名和课题ID（课题ID是0~255之间的字符值），然后调用函数ftok将这两个值变换为一个关键字。这样就避免了使用一个已被占用的关键字的问题。

使用ftok并非高枕无忧。有这样一种例外：服务器使用ftok获取得到一个关键字后，该文件就被删除了，然后重建。此时客户端以此重建后的文件来ftok所获取的关键字就和服务器的关键字不一样了。所以一般商用的软件都不怎么用ftok。

一般来说，客户机和服务器至少共享一个头文件，所以一个比较简单的方法是避免使用ftok，而只是在该头文件中存放一个大家都知道的关键字。

（d）设置信号量的值（PV操作）

```
int semop(int semid, struct sembuf *opsptr, size_t nops);
```

(1) semid: 是semget返回的semid

(2) opsptr: 指向信号量操作结构数组

(3) nops: opsptr所指向的数组中的sembuf结构体的个数

```
struct sembuf {
    short sem_num; // 要操作的信号量在信号量集里的编号，
    short sem_op; // 信号量操作
    short sem_flg; // 操作表示符
};
```

(4) 若sem\_op 是正数，其值就加到semval上，即释放信号量控制的资源

若sem\_op 是0，那么调用者希望等到semval变为0，如果semval是0就返回；

若sem\_op 是负数，那么调用者希望等待semval变为大于或等于sem\_op的绝对值

例如，当前semval为2，而sem\_op = -3，那么怎么办？

注意：semval是指semid\_ds中的信号量集中的某个信号量的值

(5) sem\_flg

SEM\_UNDO 由进程自动释放信号量

IPC\_NOWAIT 不阻塞

到这里，读者肯定有个疑惑：semop希望改变的semval到底在哪里？我们怎么没看到有它的痕迹？其实，前面已经说明了，当使用semget时，就产生了一个由内核维护的信号量集（当然每个信号量值即semval也是只由内核才能看得到了），用户能看到的就是返回的semid。内核通过semop 函数的参数，知道应该去改变semid 所指向的信号量的哪个semval。

（e）对信号集实行控制操作（semval的赋值等）

```
int semctl(int semid, int semum, int cmd, .../* union semun arg */);
```

semid是信号量集合；

semum是信号在集合中的序号；

semum是一个必须由用户自定义的结构体，在这里我们务必弄清楚该结构体的组成：

```
union semun
{
    int val; // cmd == SETVAL
    struct semid_ds *buf // cmd == IPC_SET或者 cmd == IPC_STAT
    ushort *array; // cmd == SETALL, 或 cmd = GETALL
};
```

val只有cmd ==SETVAL时才有用，此时指定的semval = arg.val。

注意：当cmd == GETVAL时，semctl函数返回的值就是我们想要的semval。千万不要以为指定的semval被返回到arg.val中。

array指向一个数组，当cmd==SETALL时，就根据arg.array来将信号量集的所有值都赋值；当cmd ==GETALL时，就将信号量集的所有值返回到arg.array指定的数组中。

buf 指针只在cmd==IPC\_STAT 或IPC\_SET 时有用，作用是semid 所指向的信号量集（semid\_ds机构体）。一般情况下不常用，这里不做谈论。

另外，cmd == IPC\_RMID还是比较有用的。

(f) 例码

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

static int nsems;
static int semflg;
static int semid;
int errno=0;
union semun {
int val;
struct semid_ds *buf;
unsigned short *array;
}arg;

int main()
{
struct sembuf sops[2]; //要用到两个信号量，所以要定义两个操作数组
int rslt;
unsigned short argarray[80];
arg.array = argarray;
semid = semget(IPC_PRIVATE, 2, 0666);
if(semid < 0 )
{
printf("semget failed. errno: %d\n", errno);
exit(0);
}
//获取0th信号量的原始值
rslt = semctl(semid, 0, GETVAL);
printf("val = %d\n",rslt);
//初始化0th信号量，然后再读取，检查初始化有没有成功
arg.val = 1; // 同一时间只允许一个占有者
semctl(semid, 0, SETVAL, arg);
rslt = semctl(semid, 0, GETVAL);
printf("val = %d\n",rslt);
sops[0].sem_num = 0;
sops[0].sem_op = -1;
sops[0].sem_flg = 0;
sops[1].sem_num = 1;
sops[1].sem_op = 1;
sops[1].sem_flg = 0;
rslt=semop(semid, sops, 1); //申请0th信号量，尝试锁定
if (rslt < 0 )
{
printf("semop failed. errno: %d\n", errno);
exit(0);
}
//可以在这里对资源进行锁定
sops[0].sem_op = 1;
semop(semid, sops, 1); //释放0th信号量
rslt = semctl(semid, 0, GETVAL);
printf("val = %d\n",rslt);
rslt=semctl(semid, 0, GETALL, arg);
if (rslt < 0)
{
printf("semctl failed. errno: %d\n", errno);
exit(0);
}
```

```
printf("val1:%d val2: %d\n", (unsigned int)jargarray[0], (unsigned int)jargarray[1]);
if(semctl(semid, 1, IPC_RMID) == -1)
{
    perror("semctl failure while clearing reason");
}
return(0);
}
```

## 七. 信号量的牛刀小试——生产者与消费者问题

### 1. 问题描述:

有一个长度为N的缓冲池为生产者和消费者所共有，只要缓冲池未满，生产者便可将消息送入缓冲池；只要缓冲池未空，消费者便可从缓冲池中取走一个消息。生产者往缓冲池放信息的时候，消费者不可操作缓冲池，反之亦然。

### 2. 使用多线程和信号量解决该经典问题的互斥

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#define BUFF_SIZE 10
char buffer[BUFF_SIZE];
char count; // 缓冲池里的信息数目
sem_t sem_mutex; // 生产者和消费者的互斥锁
sem_t p_sem_mutex; // 空的时候，对消费者不可进
sem_t c_sem_mutex; // 满的时候，对生产者不可进
void * Producer()
{
    while(1)
    {
        sem_wait(&p_sem_mutex); //当缓冲池未满时
        sem_wait(&sem_mutex); //等待缓冲池空闲
        count++;
        sem_post(&sem_mutex);
        if(count < BUFF_SIZE) //缓冲池未满
            sem_post(&p_sem_mutex);
        if(count > 0) //缓冲池不为空
            sem_post(&c_sem_mutex);
    }
}
void * Consumer()
{
    while(1)
    {
        sem_wait(&c_sem_mutex); //缓冲池未空时
        sem_wait(&sem_mutex); //等待缓冲池空闲
        count--;
        sem_post(&sem_mutex);
        if(count > 0)
            sem_post(&c_sem_mutex);
    }
}
int main()
{
    pthread_t ptid, ctid;
    //initialize the semaphores
    sem_init(&empty_sem_mutex, 0, 1);
    sem_init(&full_sem_mutex, 0, 0);
    //creating producer and consumer threads
    if(pthread_create(&ptid, NULL, Producer, NULL))
    {
```

```
printf("\n ERROR creating thread 1");
exit(1);
}
if(pthread_create(&ctid, NULL,Consumer, NULL))
{
printf("\n ERROR creating thread 2");
exit(1);
}
if(pthread_join(ptid, NULL)) /* wait for the producer to finish */
{
printf("\n ERROR joining thread");
exit(1);
}
if(pthread_join(ctid, NULL)) /* wait for consumer to finish */
{
printf("\n ERROR joining thread");
exit(1);
}
sem_destroy(&empty_sem_mutex);
sem_destroy(&full_sem_mutex);
//exit the main thread
pthread_exit(NULL);
return 1;
}
```

顶

3

踩

0

下一篇 2012

我的同类文章

嵌入式linux开发（28）

• windows下安装SVN

2014-01-04

阅读 449

• 嵌入式linux交叉编译jrtplib库

2013-09-22

阅读 564

• MINI2440启动配置文件/etc...

2013-08-25

阅读 496

• linux 調試工具Strace 編譯...

2013-07-27

阅读 416

• linux zImage啟動流程分析

2013-05-24

阅读 1608

• 基于GPL329xx linux平台电...

2016-11-25

阅读 3861

• 基于GPL329XXB IPC开发I...

2013-09-28

阅读 904

• Linux下RTC时间的读写分析

2013-09-07

阅读 3154

• microwinsows资源

2013-08-07

阅读 383

• ubuntu 安装SSH服务器

2013-06-26

阅读 431

• LVDS接口定义详解

2013-05-11

阅读 1860

更多文章

参考知识库



Linux知识库

7374 关注 | 3378 收录



软件测试知识库

2626 关注 | 310 收录

猜你在找

- 话说Linux内核-uboot和系统移植第14部分
- 阿里云ECS Linux服务器项目部署实战视频课程
- 搞定Ftp上传，远程管理Linux服务器

- 最全面的linux信号量解析
- 最全面的linux信号量解析
- 最全面的linux信号量解析




**办公电脑买不如租  
租不如找凌雄**

IT设备租赁热线: 400-678-5432  
专业团队随时为您上门服务


查看评论

8楼 wujinting007 2016-10-12 15:40发表




最后的例子中 empty\_sem\_mutex (p\_sem\_mutex)  
full\_sem\_mutex ( c\_sem\_mutex)是不是应该改为 ( ) 里的啊，还有sem\_mutex没有初始化吧

7楼 wujinting007 2016-10-12 15:39发表




最后的例子中 empty\_sem\_mutex (p\_sem\_mutex)  
full\_sem\_mutex ( c\_sem\_mutex)是不是应该改为 ( ) 里的啊，还有sem\_mutex没有初始化吧

6楼 wujinting007 2016-10-12 15:28发表




最后的例子中 empty\_sem\_mutex (p\_sem\_mutex)  
full\_sem\_mutex ( c\_sem\_mutex)是不是应该改为 ( ) 里的啊，还有sem\_mutex没有初始化吧

5楼 wujinting007 2016-10-12 15:21发表




最后的例子中 empty\_sem\_mutex (p\_sem\_mutex)  
full\_sem\_mutex ( c\_sem\_mutex)是不是应该改为 ( ) 里的啊，还有sem\_mutex没有初始化吧

4楼 hn\_lgc 2016-09-24 23:47发表




好像不是在一个时刻只有一个进程（线程）所拥有啊，  
信号量是一个整数，比如从3递减到0，应该是同时可以被多个的进程(线程)拥有吧。

3楼 applekie 2016-05-07 09:14发表



请问第一句话是不是解释反了

Re: elikang 2016-07-12 17:57发表



回复u011388486：没有吧，你再仔细理解一下哈

2楼 Kevin\_Smart 2016-04-26 18:16发表



学习了

1楼 anqiang12 2014-06-03 17:23发表



确实挺全面

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

\* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题	Hadoop	AWS	移动游戏	Java	Android	iOS	Swift	智能硬件	Docker	OpenStack
VPN	Spark	ERP	IE10	Eclipse	CRM	JavaScript	数据库	Ubuntu	NFC	WAP
jQuery	BI	HTML5	Spring	Apache	.NET	API	HTML	SDK	IIS	Fedora
XML	LBS	Unity	Splashtop	UML	components	Windows Mobile	Rails	QEMU	KDE	Cassandra
CloudStack	FTC	coremail	OPhone	CouchBase	云计算	iOS6	Rackspace	Web App	SpringSide	Maemo
Compuware	大数据	aptech	Perl	Tornado	Ruby	Hibernate	ThinkPHP	HBase	Pure	Solr
Angular	Cloud Foundry	Redis	Scala	Django	Bootstrap					