Linux进程间通讯（下）

Andrew Huang <bluedrum@163.com>

课程内容:

- IPC基本概念
- 管道
- 命名管道
- System V IPC 接口
  - System V 消息队列
  - System V 信号量
  - System V 共享内存

管道

1. 进程间通讯 (IPC)概念

- IPC是指能在两个进程间进行数据交换的机制.现代OS都对进程有保护机制.因此两个进程不能直接交换数据,必须通过一定机制来完成
- IPC的机制的作用如下
  - 因为IPC是标准机制,一个软件也能更容易跟第三方软件或内核进行配合的集成,或移植.
    - 如管道,在shell 下执行 ps –aux | grep bash
  - 简化软件结构，可以把一个软件划分多个进程或线程，通过**IPC**，集成在一起工作.如消息队列
  - 让操作系统各个模块交换数据，包括内核与应用程序机制
  - 提供进程之间或同一进程之间多线程的同步机制，如信号量

2. 管道(pipe)

- 管道是**Linux** 支持的最初**Unix IPC** 形式之一，具有以下特点
  - 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道
  - 只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）
  - 单独构成一种独立的文件系统：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在与内存中。
  - 数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。
- 管道最常用的应用就是在**Shell** 用来实现**|** 操作符的管道功能
- 管道的控制
  - 管道的创建
    - **int pipe(int fd[2]) ;**

- 管道的读写
  - 管道文件也是一种文件，用**write,read** 即可完成读写
  - 管道两端可分别用描述字**fd[0]** 以及**fd[1]** 来描述，需要注意的是，管道的两端是固定了任务的。即一端只能用于读，由描述字**fd[0]** 表示，称其为管道读端；另一端则只能用于写，由描述字**fd[1]** 来表示，称其为管道写端。如果试图从管道写端读取数据，或者向管道读端写入数据都将导致错误发生。
- 管道的关闭
  - 管道文件也是一种文件，因此用**close** 关闭即可

- 管道的示例 **test_pipe.c**
- 管道的局限
  - 只支持单向数据流；
  - 只能用于具有亲缘关系的进程之间；
  - 没有名字；
  - 管道的缓冲区是有限的（管道制存在于内存中，在管道创建时，为缓冲区分配一个页面大小）；
  - 管道所传送的是无格式字节流，这就要求管道的读出方和写入方必须事先约定好数据的格式，比如多少字节算作一个消息（或命令、或记录）等等；
- 管道的应用领域不广

2. 命名管道 (FIFO)

- 管道应用的一个重大限制是它没有名字，因此，只能用于具有亲缘关系的进程间通信，在有名管道（**named pipe**或**FIFO**）提出后，该限制得到了克服。可以在任意两个进程之间进行通讯
- **FIFO**不同于管道之处在于它提供一个路径名与之关联，以**FIFO**的文件形式存在于文件系统中。这样，即使与**FIFO**的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过**FIFO**相互通信
- **FIFO**严格遵循先进先出（**first in first out**），对管道及**FIFO**的读总是从开始处返回数据，对它们的写则把数据添加到末尾。它们不支持诸如**lseek()**等文件定位操作。
-

命名管道 (FIFO) 控制

- 命名管道的命名管道创建
  - **int mkfifo(const char * pathname, mode_t mode) ;**
- 命名管道的打开
  - 命名管道比管道多了一个打开操作：**open**
  - 在**open**时,用**O_NONBLOCK** 标志表示非阻塞模式,如
    fd=open(FIFO_SERVER,O_RDONLY|O_NONBLOCK,0);
- 命名管道的读入
  - **read** 读取管道数据
  - 读取分为阻塞和非阻塞模式,阻塞模式下,如果没有数据被入,进程会在**read**处停下来.直到有新数据被写入,或管道被关闭,才会继续
- 命名管道的写入
  - **write** 写入管道数据
  - **PIPE_BUF**表示一次触发管道读操作最大长度.如果每次写入数据长于**PIPE_BUF** ,**write**将会多次触发**read** 操作.

- 命名管道的关闭
  - 管道文件也是一种文件,因此用**close**关闭即可
- 参见**recv_fifo.c ,send_fifo.c**

System V IPC 接口

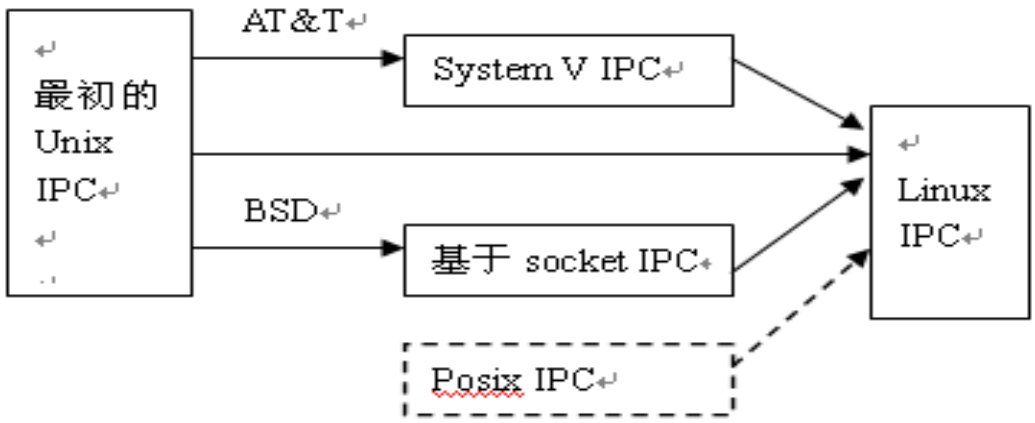
1. Linux IPC的实现

- Linux 常见6种通讯机制.管道,命名管道,信号,消息队列,信号量,共享内存.
- 其中管道,命名管道,信号是Unix沿革下来的传统的IPC机制
- 消息队列,信号量,共享内存.则是因为有System V和Posix两套实现

2. System V IPC 接口

- 由于历史原因, linux下的进程通信手段基本上是从Unix平台上的进程通信手段继承而来.
- Unix发展做出重大贡献的两大主力AT&T的贝尔实验室及BSD（加州大学伯克利分校的伯克利软件发布中心）在进程间通信方面的侧重点有所不同
  - 贝尔实验室对Unix早期的进程间通信手段进行了系统的改进和扩充，形成了“system V IPC”，通信进程局限在单个计算机内 .
  - BSD则形成了基于套接口（socket）的进程间通信机制 ,可以在跨机器进行通信.
- 由于Unix版本的多样性，电子电气工程协会（IEEE）开发了一个独立的Unix标准，这个新的ANSI Unix标准被称为计算机环境的可移植性操作系统界面（POSIX）。 他也发展出一套新的IPC接口.Linux 本身支持POSIX接口.因此也支持POSIX的IPC接口
- 最初Unix IPC包括：管道、FIFO、信号 , System V IPC包括：System V消息队列、System V信号量、System V共享内存区 , Posix IPC包括： Posix消息队列、Posix信号量、Posix共享内存区。
- System V IPC通常在多个操作系统均实现,包括一般的嵌入式Linux系统,Posix 需要测试才能通过

Linux 的IPC



图— Linux 所继承的进程间通信

- SystemV IPC 指以下三种类型的IPC ：

- SystemV 消息队列 sys/msg.h
- SystemV 信号灯 sys/sem.h
- **SystemV** 共享内存区 **sys/shm.h**

创建或打开函数
- msgget,semget,shmget

控制操作函数
- msgctl,semctl,shmctl

操作函数
- msgsnd,msgrcv,semop,shmat,shmdt

3. System V 关键字

- 每一个System V 对象(消息队列, 共享内存和信号量) 创建时, 需要的第一个参数是整数的Key 值,
  - 头文件<sys/types.h> 把key_t 定义为一个整数
- System V 创建对象时假设进行IPC通讯双方都取了相同的key值. 这样将双方关联起来
- 生成key 的方法有三种
  - 双方直接设置为一个相同的整数为key 值
  - 用IPC_PRIVA 让系统自动产生一个key 值,
  - 用ftok 函数将一个路径转换为key 值

ftok 函数

- ftok 函数把一个已存在的路径名和一个整数标识符转换成一个key_t 值，称为IPC 键(IPC key) ：
  - **#include <sys/ipc.h>**
  - **key_t ftok(const char *pahtname, int id);**
    - 如果pathname 不存在，或者对调用进程不可访问，ftok 返回-1
    - 不能保证两个不同的路径名与同一个id 值的组合产生不同的键。
    - 用于产生键的pahtname 不能是服务器存活期间由它反复创建并删除的文件，否则会导致ftok 多次调用返回不同的值

4. System V IPC 的类型

- 报文（Message ）队列（消息队列）：消息队列是消息的链接表，包括Posix 消息队列system V 消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。
- 共享内存：使得多个进程可以访问同一块内存空间，是最快的可用IPC 形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。
- 信号量（semaphore ）：主要作为进程间以及同一进程不同线程之间的同步手段。

System V 消息队列

1. 消息队列

- 消息队列就是一个消息的链表。可以把消息看作一个记录，具有特定的格式以及特定的优先级。对消息队列有写权限的进程可以向中按照一定的规则添加新消息；对消息队列有读权限的进程则可以从消息队列中读走消息。
- 消息队列能够克服早期unix 通信机制的一些缺点 ，如数据量小, 没有实时性
- 消息队列消息通常要以一个long mtype放在消息开始, mtype成员代表消息类型，从消息队列中读取消息的一个重要依据就是消息的类型
  - **struct msgbuf{ long mtype; char mtext[1]; };**
- 消息队列与管道以及有名管道相比，具有更大的灵活性
  - 它提供有格式字节流，有利于减少开发人员的工作量
  - 消息具有类型，在实际应用中，可作为优先级使用。这两点是管道以及有名管道所不能比的
  - 消息队列可以在几个进程间复用，而不管这几个进程是否具有亲缘关系，这一点与有名管道很相似；但消息队列是随内核持续的，与有名管道（随进程持续）相比，生命力更强，应用空间更大。

2. 消息队列编程

- 头文件
  - #include <sys/types.h>
  - #include <sys/ipc.h>
  - #include <sys/msg.h>
- msgget打开或创建消息队列
  - **int msgget(key_t key, int msgflg)**
- msgrcv从队列接收消息
  - **int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);**
- msgsnd 向队列发送消息
  - **int msgsnd(int msqid, struct msgbuf *msgp, int msgsz, int msgflg);**
- msgctl 发送队列控制命令
  - **msgctl(int msqid, int cmd, struct msqid_ds *buf);**
  - 共有三种cmd操作：IPC_STAT、IPC_SET 、IPC_RMID。

消息队列数据结构

- 对于系统中的每个System V消息队列，内核维护一个如下的结构：

```
struct msqid_ds {
    struct ipc_perm msg_perm; /* operation permission struct */
    struct msg *msg_first; /* ptr to first message on q */
    struct msg *msg_last; /* ptr to last message on q */
    unsigned short msg_cbytes; /* current # bytes on q */
    msgqnum_t msg_qnum; /* # of messages on q */
    msglen_t msg_qbytes; /* max # of bytes on q */
    pid_t msg_lspid; /* pid of last msgsnd */
    pid_t msg_lrpid; /* pid of last msgrcv */
    time_t msg_stime; /* last msgsnd time */
    time_t msg_rtime; /* last msgrcv time */
    time_t msg_ctime; /* last change time */
};
```

1)msgget 函数

- #include <sys/msg.h>
- **int msgget(key_t key, int oflag);**
  - 返回：成功时为非负标识符，出错时为-1
  - 用于创建一个新的SystemV消息队列或访问一个已经存在的消息队列。
- 参数key和oflag的说明见前。
- **Oflag :取值,IPC_CREAT** 创建新对象成功，IPC_EXCL检查新对象
- 返回值是一个整数标识符，其他三个msg函数用它来指代该队列。
- 当创建一个消息队列时，msqid_ds结构的如下成员被初始化：
  - msg_perm结构的uid和cuid被设置为当前进程的有效用户ID，gid和cgid被设置为当前用户的有效组ID；
  - oflag中的读写权限位存放在msg_perm.mode中；
  - msg_qnum、msg_lspid、msg_lrpid、msg_stime和msg_rtime被置为0；
  - msg_ctime被设置成当前时间；
  - msg_qbytes被设置为系统限制值。

2)msgsnd 函数

- #include <sys/msg.h>
- **int msgsnd(int msgid, const void *ptr, size_t length, int flag);**
- 返回：成功时为0，出错时为-1
- 该函数用于往消息队列上放置一个消息。
- msgid是msgget返回的标识符，ptr是一个结构指针，该结构有如下的模板：

```
struct msgbuf {
    long mtype; /* message type, must be > 0 */
    char mtext[ 1 ]; /* message data */
};
```

- 消息类型mtype必须大于0，因为非正消息类型有特殊的指示作用。
- length参数以字节为单位指定待发送消息的长度。这是位于长整数消息类型之后的用户

自定义数据的长度，该长度可以是0。

- flag参数可以是0，也可以是IPC_NOWAIT。IPC_NOWAIT标志使得msgsnd调用非阻塞。当有如下情形之一时：
  - 在指定的队列中已经有太多的字节(对应msqid_ds结构中的msg_qbytes值)；
  - 在系统范围存在太多的消息。
  - 若设置了IPC_NOWAIT，则msgsnd立即返回，返回一个EAGAIN错误。若未指定该标志，则msgsnd阻塞，直到具备存放新消息的空间；
  - 有msgid标识的消息队列被删除，此时返回EIDRM错误；
  - 被信号中断，此时返回EINTR错误。

3)msgrcv 函数

- #include <sys/msg.h>
- **ssize_t msgrcv(int msqid, void *ptr, size_t length, long type, int flag);**
- 返回：成功时为读入缓冲区中数据的字节数，出错时为-1
- 该函数从某个消息队列中读出一个消息。
- ptr参数指定所接收消息的存放位置。跟msgsnd一样，该指针指向紧挨在真正的消息数据之前返回的长整数类型字段。
- length指定由ptr指向的缓冲区中数据部分的大小。这是该函数能返回的最大数据量。该长度不包含长整数类型字段。
- type指定希望从所给定的队列中读出什么样的消息：
  - type为0，返回队列中第一个消息。每个消息队列是作为一个FIFO链表维护的，所以返回的是队列中最早的消息。
  - type大于0，返回其类型值为type的第一个消息。
  - type小于0，返回其类型值小于或等于type参数的绝对值的消息中类型值最小的第一个消息。
  - flag参数指定所请求的消息不在队列中时怎么办。在没有消息时，若设置了IPC_NOWAIT标志，则函数立即返回一个ENOMSG错误；否则，调用者阻塞直到如下某个时间发生：
    - 有一个所请求类型的消息可获取；
    - 由msqid标识的消息队列被删除，此时返回个EIDRM错误；
    - 被某个捕获的信号中断，此时返回EINTR错误。

4)msgctl 函数

- #include <sys/msg.h>
- **int msgctl(int msqid, int cmd, struct msqid_ds *buf);**
- 返回：成功时为0，出错时为-1
- 该函数提供在一个消息队列上的各种控制操作。
- msgctl提供三个命令：
  - IPC_RMID：从系统中删除由msqid指定的消息队列。当前在该队列上的任何消息都被丢弃。此时。第三个参数忽略不用。
  - IPC_SET：给指定的消息队列设置其msqid_ds结构的以下四个成员：msg_perm.uid、msg_perm.gid、msg_perm.mode和msg_perm.qbytes。它们的值来自buff指向的结构中的相应成员。
  - IPC_STAT：通过buff参数给调用者返回所指定消息队列中的当前msqid_ds结构。

消息队列打开

- 如果没有调用 msgctrl(semid,IPC_RMID,0)删除消息队列,则消息队列一直存在内核中,即便是创建进程已经退出也是如此,这个用ipcs可以看到
- 如果对一个已经创建的消息队列的路径再次创建消息队列,通常都会出错.因此可以采用一种保险的写法

```
/* 首先查询这个队列是否创建，如创建直接用它*/
if((msgid =msgget(key,IPC_EXCL|0666)) == -1)
{
        /*没有创建才去创建这个消息队列*/
        msgid=msgget(key,IPC_CREAT|IPC_EXCL|00666);
        if(msgid==-1)
        {
            printf("msg create error\n");
            return;
        }
}
```

- 其它对象也用这样打开方法

System V 信号量

1. 信号量

- 信号量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。
- 信号量相当是一个全局的整数变量,这个变量只能用原子操作来改变值
- 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。

2. 信号量集的数据结构

- SystemV信号灯是信号灯集的概念：一个或多个信号灯构成一个集合。对于系统每个信号灯集，内核维护如下的一个结构：

```
struct semid_ds {
    struct ipc_perm sem_perm; /* operation permission struct */
    struct sem *sem_base; /* ptr to first semaphore in set */
    unsigned short sem_nsems; /* # of semaphores in set */
    time_t sem_otime; /* last semop time */
    time_t sem_ctime; /* last change time */
};
```

- 当前信号灯集中的每个信号灯对应一个sem结构。定义如下：

```
struct sem {
    signed short semval; /* semaphore text map address */
    pid_t sempid; /* pid of last operation */
    unsigned short semncnt; /* # awaiting semval > cval */
    unsigned short semzcnt; /* # awaiting semval = 0 */
};
```

关于P原语

- **P原语：**
  - P是荷兰语Proberen（测试）的首字母。为阻塞原语，负责把当前进程由运行状态转换为阻塞状态，直到另外一个进程唤醒它。操作为：申请一个空闲资源（把信号量减1），若成功，则退出；若失败，则该进程被阻塞；
  - P原语的操作过程
    - sem减1；
    - 若sem减1后仍大于或等于零，则进程继续执行；
    - 若sem减1后小于零，则该进程被阻塞后进入与该信号相对应的队列中，然后转进程调度。
  - System V 实现代码

```
int p(int semid)
{
 struct sembuf sops={0,+1,IPC_NOWAIT};
 return (semop(semid,&sops,1));
}
```

关于V原语

- **V原语**
  - V是荷兰语Verhogen（增加）的首字母。为唤醒原语，负责把一个被阻塞的进程唤醒，它有一个参数表，存放着等待被唤醒的进程信息。操作为：释放一个被占用的资源（把信号量加1），如果发现有被阻塞的进程，则选择一个唤醒之
  - V原语的操作过程
    - sem加1；
    - 若相加结果大于零，则进程继续执行；
    - 若相加结果小于或等于零，则从该信号的等待队列中唤醒一等待进程，然后再返

回原进程继续执行或转进程调度。

– System V 实现代码

```
int v(int semid)
{
  struct sembuf sops={0,-1,IPC_NOWAIT};
  return (semop(semid,&sops,1));
}
```

3. 信号量的处理流程

- 用到头文件
  - #include <sys/types.h>
  - #include <sys/ipc.h>
  - #include <sys/sem.h>
- semget创建或打开一个信号量
  - **int semget(key_t key, int nsems, int semflg)**
- Semop()对信号量 +1 或 -1 或测试是否为0
  - **int semop(int semid, struct sembuf *sops, unsigned nsops);**
  - linux可以增加或减小信号量的值，相应于对共享资源的释放和占有
- semctl 对信号量进行各种控制
  - **int semctl(int semid，int semnum，int cmd，union semun arg)**

1)semget 函数

- #include <sys/sem.h>
- int semget(key_t key, int nsems, int oflag);
- 返回：成功时为非负标识符，出错时为-1
- 创建一个信号灯集或访问一个已存在的信号灯集。
- 返回值是信号灯标识符，供其他信号灯函数使用。
- nsems是集合中的信号灯数。如果不是创建一个信号灯集，而只是访问已存在的集合，则该参数可以指定为0。一旦创建完毕一个信号灯集，就不能改变其中的信号灯数。
- 当实际操作为创建一个新的信号灯集时，semid_ds结构的以下成员将被初始化：
  - sem_perm结构的uid和cuid被设置为调用进程的有效用户ID，gid和cgid被设置为调用进程的有效组ID；
  - oflag参数中的读写权限存入sem_perm.mode中；
  - sem_otime被设置为0，sem_ctime被置为当前时间；
  - sem_nsems被置为nsems参数的值；
  - 与该集合中每个信号灯关联的各个sem结构并不初始化。这些结构必须是在以SETVAL或SETALL命令调用semctl时初始化的。

信号量创建问题

- SystemV信号灯的创建和初始化需两次函数调用是一个致命的缺陷，这会导致竞争状态的出现。
- 解决竞争状态的方法是：当semget创建一个新的信号灯集时，其semid_ds结构的sem_otime成员保证被设置为0。该成员只是在semop调用成功时才被设置为当前值。在调用semget进行访问而不是创建时，以IPC_STAT命令调用semctl，然后等待sem_otime

变为非零值。到时就可断定该信号灯已经被初始化，而且对它初始化的进程已成功完成semop调用。所以，创建该信号灯集的进程必须初始化它的值，而且必须在任何其他进程可以使用该信号灯集之前调用semop。

- 这样将会造成程序相当复杂

2)semop 函数

- #include <sys/sem.h>
- **int semop(int semid, struct sembuf *opsptr, size_t nops);**
- 返回：成功时为0，出错时为-1
- 对一个或多个信号灯进行操作。
- opsptr指向如下结构模板的数组(该结构可能不止如下几个成员)：

```
struct sembuf {
    shrot sem_num; /* semaphore number:0,1,..,nsems-1 */
    short sem_op; /* semaphore operation: < 0,0, >0 */
    short sem_flg; /* operation flags:0,IPC_NOWAIT,SEM_UNDO */
};
```

- nops参数指出结构数组中元素的个数。每个元素给目标信号灯集中某个信号灯指定一个操作。特定的信号灯由sem_num指定；sem_op指定特定的操作；sem_flg指定非阻塞(IPC_NOWAIT)、恢复等标志。在阻塞、非阻塞情况下返回的错误情况与其他SystemV IPC相同。
- semop函数由内核保证原子的执行，内核或者完成所有操作，或者什么也不做。
- semop操作的具体描述：
  - 如果sem_op是正数，其值就加到semval(信号灯的当前值)上，这对应于释放由某个信号灯控制的资源。如果指定了SEM_UNDO标志，就从相应信号灯的semadj值中减掉sem_op的值。
  - 如果sem_op是0，那么调用者希望等待到semval变为0，如果semval已经是0，则立即返回；如果semval不为0，相应信号灯的semzcnt(等待semval变为0的线程数)值就加1，调用线程阻塞到semval变为0(那时semzcnt再减1)。若指定了IPC_NOWAIT，则调用线程不会睡眠，返回EAGAIN。
  - 如果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，这对应于分配资源。如果semval大于或等于sem_op的绝对值，则从semval中减掉sem_op的绝对值，如果指定了SEM_UNDO，那么sem_op的绝对值就加到相应信号灯的semadj值上。如果semval小于sem_op的绝对值，相应信号灯的semncnt值就加1，调用线程阻塞直到semval变为大于或等于sem_op的绝对值。若指定了IPC_NOWAIT，则调用线程不会睡眠，返回EAGAIN。
- semadj称为指定信号灯针对调用进程的调整值。当调用进程终止时，semadj的值就加到相应信号灯的semval上。若调用进程对某个信号灯的全部操作都指定SEM_UNDO标志，则该进程终止时，该信号灯的值就会变得像根本没有运行过该进程一样，这就是复旧(undo)的本意。

semop 的调用问题

- semop采用复杂的信号灯集做参数,因此造成程序调用变得复杂.
- 信号灯的值加1,或值减1都在semop完成,相当于加锁或解锁都是有一个函数完成,这也是

与其它互质量等其它同步机制不一样的地方

- 一般是将其封装成P,V原语的函数来操作

```
/* 申请资源,用 P 原语*/
int semaphore_wait_p(int sem_id) {
struct sembuf sb;
sb.sem_num = 0;
sb.sem_op = -1;
sb.sem_flg = SEM_UNDO;
if (semop(sem_id, &sb, 1) == -1) {
    fprintf(stderr, "semaphore_p failed\n");
                    return (-1);
            }
            return 0;
}
```

```
/* 释放或分配资源用，用 V 原语 */
int semaphore_signal_v(int sem_id) {
struct sembuf sb;
sb.sem_num = 0;
sb.sem_op = 1;
sb.sem_flg = SEM_UNDO;
if (semop(sem_id, &sb, 1) == -1) {
    fprintf(stderr, "semaphore_v failed\n");
                    return (-1);
            }
            return 0;
}
```

3)semctl 函数

- #include <sys/sem.h>
- **int semctl(int semid, int semnum, int cmd, ¡-/* union arg */);**
- 返回：成功时为非负值，出错时为-1
- 对一个信号灯执行各种控制操作。
- semnum标识某个信号灯，semnum仅仅用于GETVAL、SETVAL、GETNCNT、GETZCNT和GETPID命令
- 第四个参数是可选的，它依赖于第三个参数cmd。它是一个联合：

```
union semun {
        int val; /* used for SETVAL only */
        struct semid_ds *buf; /* used fro IPC_SET and IPC_STAT */
        ushort *array; /* used for GETALL and SETALL */
};
```

- 该联合没有出现在任何系统头文件中，由应用程序声明。而且它是以值传递的，而不是以引用传递的。

semop 命令选项

- GETVAL：把semval的当前值作为函数返回值返回。
- SETVAL：把semval设置为arg.val。如果操作成功，那么相应信号灯在所在进程中的调整值(semadj)将被置为0。
- GETPID：把sempid的当前值作为函数值返回。
- GETNCNT：把semncnt的当前值作为函数值返回。
- GETZCNT：把semzcnt的当前值作为函数值返回。
- GETALL：返回所指定信号灯集的每个成员的semval值。这些值通过arg.array指针返回。函数本身返回值为0。注意，调用者必须分配足够容纳所指定信号灯集中所有成员的semval值的一个unsigned short整数数组，然后把arg.array设置成指向这个数组。
- SETALL：设置所指定信号灯集中每个成员的semval值。这些值通过arg.array数组指定。

- IPC_RMID：把由semid指定的信号灯集从系统中删除。
- IPC_SET：设置semid_ds结构中的以下三个成员：sem_perm.uid、sem_perm.gid和sem_perm.mode。这些值来自由arg.buf参数指向的结构中相应成员。semid_ds中的sem_ctime成员也被设置为当前值。
- IPC_STAT：通过arg.buf参数返回当前的semid_ds结构。注意，调用者必须首先分配一个semid_ds结构，并把arg.buf设置为指向这个结构。

System V 共享内存

1. 共享内存数据表示

- 对于每个System V共享内存区，内核维护如下的信息结构：

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* operation permission struct */
    size_t shm_segsz; /* size of segment in bytes */
    pid_t shm_lpid; /* pid of last shmop */
    pid_t shm_cpid; /* pid of creator */
    shmatt_t shm_nattch; /* current # attached */
    shmat_t shm_cnattch; /* in-core # attached */
    time_t shm_atime; /* last shmat time */
    time_t shm_dtime; /* last shmdt time */
    time_t shm_ctime; /* last change time */
};
```

2. System V 共享内存使用流程

- 使用头文件
  - #include <sys/ipc.h>
  - #include <sys/shm.h>
- shmget() 获得共享内存区域的ID
  - 如果不存在指定的共享区域就创建相应的区域。
  - **int shmget(key_t key,int size,int shmflg);**
- shmat()把共享内存区域映射到调用进程的地址空间中去
  - 这样，进程就可以方便地对共享区域进行访问操作。
  - **void *shmat(int shmid,const void *shmaddr,int shmflg);**
- shmdt()调用用来解除进程对共享内存区域的映射
  - **int shmdt(const void *shmaddr);**
- Shmctl()实现对共享内存区域的控制操作
  - **int shmctl(int shmid,int cmd,struct shmid_ds *buf);**

1)shmget 函数

- #include <sys/shm.h>
- **int shmget(key_t key, size_t size, int oflag);**

- 返回：成功时为共享内存区对象，出错时为-1
- 函数创建一个尚未存在的共享内存区，或者访问一个已存在的共享内存区。
- 返回值是共享内存区标识符，供其他函数使用。
- size参数以字节为单位指定内存的大小。当实际操作为创建一个新的内存区时，必须指定一个不为0的size值；如果实际操作是访问一个已存在的共享内存区，则size应为0。
- 当实际操作为创建一个新的内存区时，该内存区被初始化为size个字节的0。

2)shmat 函数

- #include <sys/shm.h>
- **void * shmat(int shmid, const void *shmaddr, int flag);**
- 返回：成功时为映射区的其始地址，出错时为-1
- 调用shmat将共享内存区附接到调用进程的地址空间。
- shmid是shmget的返回值。shmat的返回值是所指定的共享内存区在调用进程内的起始地址。确定此地址的规则如下：
  - 如果shmaddr是空指针，则系统替调用者选择地址。这是推荐(也是可移植性最好的)方法。
  - 如果shmaddr非空，则返回地址取决于调用者是否给flag参数指定了SHM_RND值。如果SHM_RND没有指定，则共享内存区附接到由shmaddr指定的地址；若指定SHM_RND，则附接到由shmaddr指定的地址向下舍入一个SHMLBA常值。LBA代表 ”低端边界地址(lower boundary address)”。
- flag参数可以指定SHM_RDONLY值，它限定只读访问。

3)shmdt 函数

- #include <sys/shm.h>
- **int shmdt(const void *shmaddr);**
- 返回：成功时为0 ，出错时为-1
- 调用shmdt 断开与共享内存区的连接。
- 当一个进程终止时，它的所有当前附接着的共享内存区都自动断接掉。

4)shmctl 函数

- #include <sys/shm.h>
- **int shmctl(int shmid, int cmd, struct shmid_ds *buff);**
- 返回：成功时为0，出错时为-1
- 函数提供三个命令：
  - IPC_RMID：从系统中删除由shmid标识的共享内存区并拆除它。
  - IPC_SET：给所指定的共享内存区设置其shmid_ds结构的以下三个成员：shm_perm.uid、shm_perm.gid和shm_perm.mode，它们的值来自参数中的相应成员。shm_ctime的值用当前时间替换。
  - IPC_STAT：向调用者返回所指定共享内存区的当前shmid_ds结构。

3. 关于System V 的维护命令

- 可以用ipcs命令查看system V对象
- 用ipcrm可以删除system V 对象
  - ipcrm sem 196632

  - 删除semid　为196632的信号量
- ipcrm shm 12395
  - 删除shmid　为12395的共享内存
- ipcrm msg 234
  - 删除msgid为 234的消息队列

思考题

- 已知一个信号量初始值为5,如果信号量加锁为0,当前值为-1,请问现在被这个信号量阻塞的进程有多少个?

课堂练习

- 为守护进程加上信号处理函数
  - 当守护进程收到 SIGUSR1时,将重新读取配置文件
  - 动态分配一个块内存,在退出程序时,用信号处理函数来free
  - 要求用 signal 和sigaction 两种模式各做一次
  - 用kill命令,和自行开发程序发送信号,触发读配置文件操作