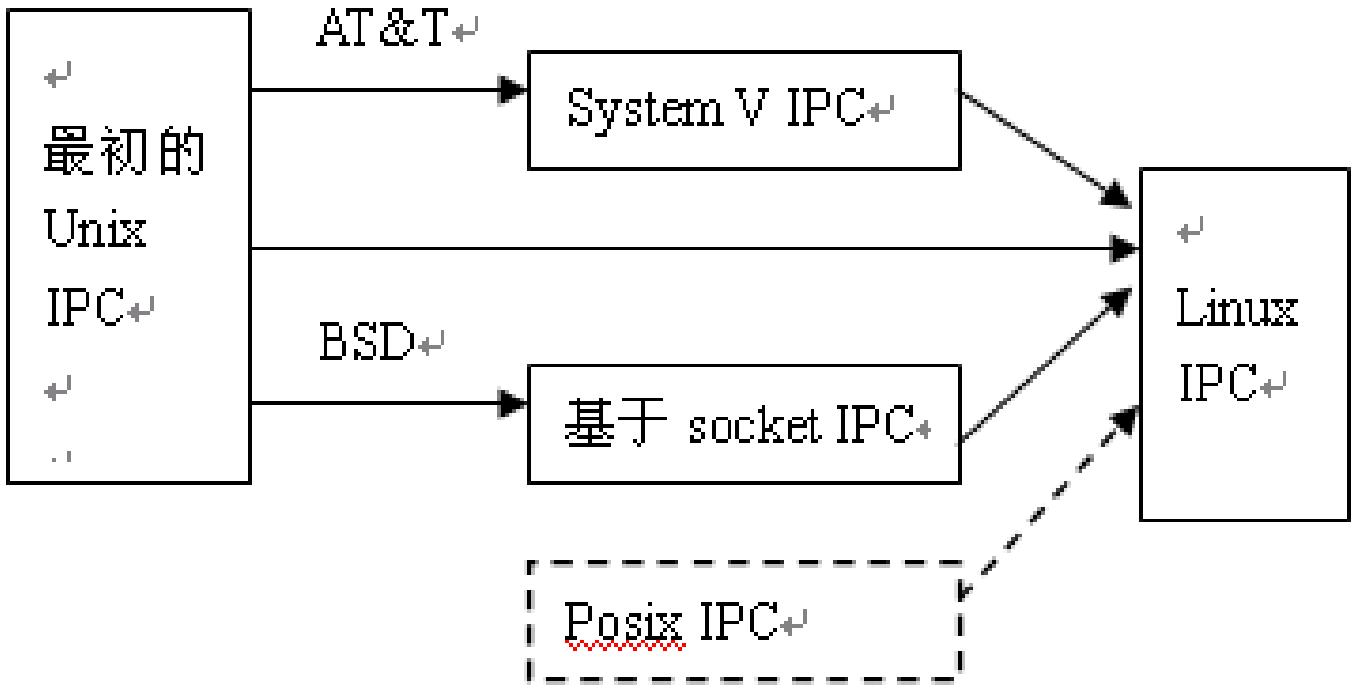
进程间通信

理解进程间通信

- 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的。而对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。前者对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；后者则跳过了该限制，形成了基于套接口（socket）的进程间通信机制。Linux则把两者继承了下来

理解进程间通信

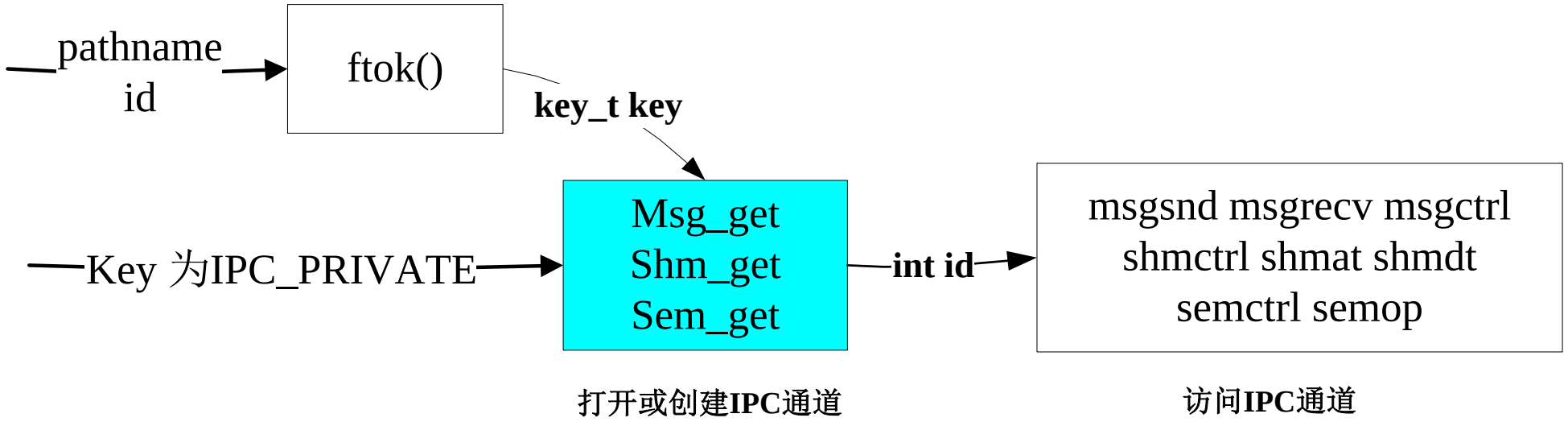


图一 Linux 所继承的进程间通信

理解进程间通信

- 最初Unix IPC包括：
  - 管道、FIFO、信号；
- System V IPC包括：
  - System V消息队列、System V信号灯、System V共享内存区；
- Posix IPC包括：
  - Posix消息队列、Posix信号灯、Posix共享内存区。

System V IPC对象

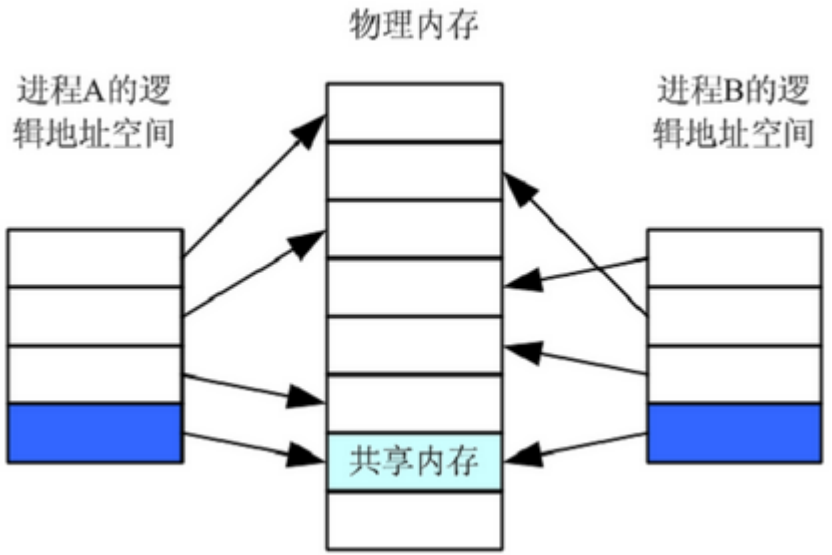


进程间通信之

共享内存

共享内存

- 原理：
  - 由操作系统提供一块物理内存，映射到多个进程空间中。
  - 在两个进程的虚拟内存映射表中，多个进程的虚拟内存都指向同一块物理内存。
- 注意：
  - 多进程操作同一块内存，
  - 要注意依靠进程间同步机制，保护临界资源。



思考：多进程的共享内存和多线程的全局变量有那些相似和区别？

共享内存

- 系统在内核在预申请了一部分内存，用作共享内存。使用共享内存，就像多线程操作全局变量一样，非常高效；所以共享内存是进程间通信最高效的一种方法。
- 优点：
  - 速度快，高效，进程间大量快速数据交互时使用。
  - 无需复杂操作即可修改目标进程中的内存数据。
- 缺点：
  - 需要注意保护临界区资源。
  - 需要完全自定义数据格式。
  - 实用起来复杂。

共享内存

- 共享内存的使用，主要有以下几个API：
- ftok()　　：生成一个共享内存的标识符。
- shmget()　　：申请一段共享内存。
- shmat()　　：将共享内存映射到本进程的虚拟空间中。
- shmdt()　　：删除本进程对某段共享内存的使用权。
- shmctl()。　　：控制共享内存，删除共内存区，得到状态。

ftok

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

- key_t ftok(const char *pathname, int proj_id);
- pathname：文件路径
- proj_id：　　工程的id
- 用来将文件路径和工程的id组合生成一个key_t，这个key_t后来作为IPC对象的唯一标识符。
- IPC对象的唯一标识符为整数，不好记忆，所以将一个文件和一个工程id与这个key_t对应起来，起到翻译的作用（域名和IP）。

共享内存

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int shmget(key_t key, int size, int shmflg);
函数参数	key：IPC_PRIVATE、自定义的值 或 ftok的返回值
	size：共享内存区大小
	shmflg：同open函数的权限位，也可以用8进制表示法
函数返回值	成功：共享内存段标识符，如果后两个参数都为0，用来测试共享内存是否存在。
	出错：-1

共享内存

- key为shm产生前的索引值，一般可以设为：
  - ftok的返回值。只要其他进程知道ftok的参数，即文件和工程号就可以生成相同的key，使用它找到、使用这块共享内存。
  - 自定义的值。自己随便定义一个，只要其他进程知道这个值就可以找到并使用这块共享内存。
  - IPC_PRIVATE
- size：要申请的共享内存区大小，注意共享内存不能无限大，每个系统共享内存区大小时固定的，但在系统中可以配置，默认都大于32MB。
- shmflg为标识位，可由以下常用标记组成
  - IPC_CREAT　　shmid标识的共享内存区不存在则创建，存在则打开。
  - IPC_EXCL　　和IPC_CREAT同时使用，shmid标识的共享内存区存在时返回错误。
  - 0600　　如果共享内存时新创建的，标识该共享内存的用户权限。

共享内存

- ipcs –m 显示共享内存的信息（root权限才能显示所有的共享内存信息）
- ipcrm –m shmid 删除指定的共享内存（没人用时才被删掉）

- ipcs –a 显示所有的ipc对象信息，包括共享内存、消息队列、信号量。

```
root@ubuntu:/home/ziyang/Documents/5-4# ipcs -m

------------ 共享内存段 --------------
键         shmid      拥有者   权限     字节      nattch     状态
0x00000000 0          ziyang     600        393216     2          目标
0x00000000 32769      ziyang     600        393216     2          目标
0x00000000 65538      ziyang     600        393216     2          目标
0x00000000 98307      ziyang     600        393216     2          目标
0x00000000 131076     ziyang     600        393216     2          目标
```

实验1

- 1、用指令ipcs查看共享内存的状态，期间申请一段共享内存，对比前后的变化。
- 2、判断与1.1相同的key对应的共享内存是否存在，观察返回，然后用ipcrm删掉这段共享内存。

共享内存

所需头文件	**#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>**	
函数原型	**void *shmat(int shmid, const void *shmaddr, int shmflg);**	
函数参数	**shmid：要映射的共享内存区标识符**	
	shmaddr：将共享内存映射到指定地址(若为NULL，则表示由系统自动完成映射)	
	shmflg :	**SHM_RDONLY：共享内存只读**
		默认0：共享内存可读写
函数返回值	成功：映射后的地址	
	出错：-1	

实验2

- 1、申请或打开一段共享内存，将内存映射到进程空间中，一个进程写入字符串“this is a message from shm”，并给另一个进程发信号，另一个进程收到信号后打印该字符串。

共享内存

所需头文件	**#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>**
函数原型	**int shmdt(const void *shmaddr);**
函数参数	**shmaddr：共享内存映射后的地址**
函数返回值	**成功：0**
	出错：-1

共享内存

- shmdt将进程中共享 内存的映射删除掉，之后该进程将不能使用这段共享内存，但共享内存对应的物理内存并不会发生变化，如果被别的进程映射，依旧可用。
- 解除映射后共享内存的nattch减1。
- 进程结束，系统会自动解除内存映射， nattch减1。

实验3

- 1、申请一块共享内存，映射到当前进程中，在共享内存中写入“this is a message from shm”，然后解除共享内存映射； 进程B将这块共享内存映射到自己的地址空间，打印内容，并将内存中的内容改为“Process （pid） toke the message away！”。

共享内存

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int shmctl(int shmid, int cmd, struct shmid_ds *buf);
函数参数	shmid：要操作的共享内存标识符 cmd： IPC_STAT (获取对象属性) IPC_SET (设置对象属性) IPC_RMID (删除对象) buf：指定IPC_STAT/IPC_SET时用以保存/设置属性
函数返回值	成功：0 出错：-1

共享内存

The buf argument is a pointer to a shmid_ds structure, defined in <sys/shm.h> as follows:

```
struct shmid_ds {
    struct ipc_perm shm_perm;    /* Ownership and permissions */
    size_t          shm_segsz;   /* Size of segment (bytes) */
    time_t          shm_atime;   /* Last attach time */
    time_t          shm_dtime;   /* Last detach time */
    time_t          shm_ctime;   /* Last change time */
    pid_t           shm_cpid;    /* PID of creator */
    pid_t           shm_lpid;    /* PID of last shmat(2)/shmdt(2) */
    shmatt_t        shm_nattch;  /* No. of current attaches */
    ...
};
```

The ipc_perm structure is defined in <sys/ipc.h> as follows (the highlighted fields are settable using **IPC_SET**):

```
struct ipc_perm {
    key_t          __key;    /* Key supplied to shmget(2) */
    uid_t          uid;      /* Effective UID of owner */
    gid_t          gid;      /* Effective GID of owner */
    uid_t          cuid;     /* Effective UID of creator */
    gid_t          cgid;     /* Effective GID of creator */
    unsigned short mode;     /* Permissions + SHM_DEST and
                                SHM_LOCKED flags */
    unsigned short __seq;    /* Sequence number */
};
```

实验4

- 1、将实验3.1改为取走消息后，删除共享内存。

进程间通信之

消息队列

消息队列

- 消息队列是IPC对象的一种。
- 消息队列由消息队列ID来唯一标识，（还记得ftok吗？）。
- 消息队列就是一个消息的列表。用户可以在消息队列中添加消息、读取消息等。
- 消息队列可以按照类型来发送/接收消息。
- 消息队列是应用最用最广泛的一种进程通信方式，是本节**重点**。

消息队列

- 消息队列的操作包括创建或打开消息队列、添加消息、读取消息和控制消息队列。
- 创建或打开消息队列使用的函数是msgget，这里创建的消息队列的数量会受到系统消息队列数量的限制。
- 添加消息使用的函数是msgsnd，按照类型把消息添加到已打开的消息队列末尾。
- 读取消息使用的函数是msgrcv，可以按照类型把消息从消息队列中取走。
- 控制消息队列使用的函数是msgctl，它可以完成多项功能。

理解消息队列

- 消息队列的本质是内核中的一个链表，这个链表开放了链表操作的一部分功能，并且这个链表对所有的进程是可见的，不同进程操作同一个链表完成通信。
- msgget 是创建并初始化这个链表。
- msgsnd发送消息是在链表的尾部插入一个结点。
- msgrcv是遍历链表，找到第一个符合要求的结点，把结点摘掉。
- msgctl完成链表操作的其他开放功能。

消息队列

所需头文件	**#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>**
函数原型	**int msgget(key_t key, int flag);**
函数参数	**key：和消息队列关联的key值（同shmget说明）**
	flag：消息队列的访问权限（同shmget说明）
函数返回值	成功：消息队列ID
	出错：-1

实验5

- 1、创建一个消息队列，然后用ipcs –q查看。（参考shmget独立完成）

消息队列

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>
函数原型	int msgsnd(int msqid, const void *msgp, size_t size, int flag);
函数参数	msqid：消息队列的ID

	msgp：指向消息的指针。常用消息结构msgbuf如下：
	struct msgbuf{
	long mtype; //消息类型
	char mtext[N]}; //消息正文

	size：发送的消息正文的字节数
	flag：IPC_NOWAIT 消息没有发送完成函数也会立即返回。
	0：直到发送完成函数才返回
函数返回值	成功：0
	出错：-1

消息队列

- msqid：消息队列的id，msgget的返回值
- msgp：必须是这样的结构：

```
struct msgbuf{
    long mtype;       //消息类型，必须大于0。（思考：为什么？）
    char mtext[N];    //消息正文
};
```

- Size：消息正文mtext的大小。
- flag：
  - 默认0，阻塞知道消息发送完。
  - IPC_NOWAIT ，非阻塞，消息没有发送完成函数也会立即返回，（非阻塞发送没有完成，会返回失败，error为EAGAIN）。

消息队列

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>	
函数原型	int msgrcv(int msgid, void* msgp, size_t size, long msgtype, int flag);	
函数参数	msqid：消息队列的ID	
	msgp：接收消息的缓冲区	
	size：要接收的消息的字节数	
	msgtype：	0：接收消息队列中第一个消息。
		大于0：接收消息队列中第一个类型为msgtyp的消息.
		小于0：接收消息队列中类型值不小于msgtyp的绝对值且类型值又最小的消息。
	flag：	0：若无消息函数会一直阻塞
		IPC_NOWAIT：若没有消息，进程会立即返回ENOMSG。
函数返回值	成功：接收到的消息的长度（msgsnd中mtext的长度）	
	出错：-1	

消息队列

- 消息队列是消息机制在共享内存上的一种实现，是对共享内存的一种链表性的封装。
- 其不同的消息类型可对应不同的进程间通信，也可对应不同的数据格式。
- 消息队列和信号机制结合，构成实时处理模型。
- 消息队列和定时器结合，构成轮询处理模型。
- 消息队列是实际工作中处理进程间通信最常用的方法。

实验6

- 1、设计一个消息队列，一个进程发送消息，一个进程接收消息。
- 2、假定一个门锁的设备、一个进程兼容不同门锁设备，收到开门消息后开门；一个进程用于WIFI开门验证，验证通过后向门锁设备发送开门消息。
- 进程A收到一个信号，就从队列中取出消息，进行相关的操作。
- 进程B验证通过就发个消息到消息队列中，指明开门还是关门、开关哪个门。

消息队列

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>
函数原型	int msgctl (int msgqid, int cmd, struct msqid_ds *buf);
函数参数	msqid：消息队列的队列ID
	cmd： IPC_STAT：读取消息队列的属性，并将其保存在buf指向的缓冲区中。
	IPC_SET：设置消息队列的属性。这个值取自buf参数。
	IPC_RMID：从系统中删除消息队列。
	buf：消息队列缓冲区
函数返回值	成功：0
	出错：-1

消息队列

- struct msqid_ds

```
 {
   struct ipc_perm msg_perm;
   struct msg *msg_first;                /* first message on queue,unused  */
   struct msg *msg_last;                 /* last message in queue,unused */
   __kernel_time_t msg_stime;            /* last msgsnd time */
   __kernel_time_t msg_rtime;            /* last msgrcv time */
   __kernel_time_t msg_ctime;            /* last change time */
   unsigned long  msg_lcbytes;           /* Reuse junk fields for 32 bit */
   unsigned long  msg_lqbytes;           /* ditto */
   unsigned short msg_cbytes;            /* current number of bytes on queue */
   unsigned short msg_qnum;              /* number of messages in queue */
   unsigned short msg_qbytes;            /* max number of bytes on queue */
   __kernel_ipc_pid_t msg_lspid;         /* pid of last msgsnd */
   __kernel_ipc_pid_t msg_lrpid;         /* last receive pid */
}
```

消息队列

- 消息队列跟命名管道 、共享内存有不少的相同之处，和它们一样，消息队列进行通信的进程可以是不相关的进程，同时它们都是通过发送和接收的方式来传递数据的。在命名管道中，发送数据用write，接收数据用read，则在消息队列中，发送数据用msgsnd，接收数据用msgrcv。而且它们对每个数据都有一个最大长度的限制。

- 与命名管道、共享内存相比，消息队列的优势在于：
- 1、通过消息队列可以避免共享内存的临界区保护问题，不需要由进程自己来考虑临界区保护。
- 2、接收程序可以通过消息类型有选择地接收数据，而不是像命名管道中那样，只能默认地接收，所以一个队列可以很多进程使用，使用消息类型区分，不会错乱。
- 3、消息队列提供原子消息，保证消息的完整性，而管道读写由于管道读写的异步，不能保证一次写和一次读操作的对应；共享内存甚至不能保证读写操作的完整性，需要加临界区保护机制和同步机制。

实验7

- 1、向消息队列中发一些消息，得到队列中消息的个数。
- 2、建立一个消息队列，用ipcs –m查看，再删除消息队列，观察变化。

进程间System V IPC通信之

信号量

信号量

- 信号量是一种system v提供的进程间同步的机制。
- 信号量是线程间信号量机制在进程间的扩展实现
  - 之前学的posix线程信号机制：sem_init、sem_wait、sem_post、sem_destory
  - 进程间system v信号机制：
    - semget：初始化，对应sem_init
    - semop： P、V操作，对应sem_wait、sem_post
    - semctl： 删除信息，对应sem_destroy
- 不同点：
  - 进程间信号一次可操作多个信号灯，更复杂一点。

信号量

所需头文件	**#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>**
函数原型	**int semget(key_t key, int nsems, int semflg);**
函数参数	**key：和信号灯集关联的key值**
	nsems: 信号灯集中包含的信号灯数目
	semflg：信号灯集的访问权限，通常为IPC_CREAT | 0666
函数返回值	成功：信号灯集ID
	出错：-1

信号量

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semop (int semid, struct sembuf *opsptr, size_t nops);

函数参数	semid：信号灯集ID
	opsptr：要操作的信号灯数组
	struct sembuf {
	short sem_num; // 要操作的信号灯的编号
	short sem_op; // 0：等待，直到信号灯的值变成0
	// 1：释放资源，V操作
	// -1：分配资源，P操作
	short sem_flg; // 0, IPC_NOWAIT, SEM_UNDO
	};
	
	nops: 要操作的信号灯的个数

函数返回值	成功：0
	出错：-1

信号量

- struct sembuf **sops[2];**
- int semid;

- /* Code to set semid omitted */

- sops[0].sem_num = 0; /* Operate on semaphore 0 */
- sops[0].sem_op = 0; /* Wait for value to equal 0 */
- sops[0].sem_flg = 0;

- sops[1].sem_num = 0; /* Operate on semaphore 0 */
- sops[1].sem_op = 1; /* Increment value by one */
- sops[1].sem_flg = 0;

- if (semop(semid, **sops**, 2) == -1) {//第二个参数是标识要操作的信号灯，如果多于一个，使用数组存放，第三个参数标识数组中元素个数
- perror("semop");
- exit(EXIT_FAILURE);
- }

信号量

- 信号量被占用之后，一般不会自动释放，如果程序意外崩溃，可能导致信号量得不到释放而死锁。
- 设置sops的SEM_UNDO标识位，可在程序终止时自动释放申请的信号量。
- sops[1].sem_num = 0;　　/* Operate on semaphore 0 */
- sops[1].sem_op = 1;　　/* Increment value by one */
- sops[1].sem_flg = SEM_UNDO;

- 思考：如何设计一个程序，每次开机后只能运行一次。

信号量

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semctl (int semid, int semnum, int cmd…/*union semun arg*/);
函数参数	semid：信号灯集ID semnum: 要修改的信号灯编号 cmd：　GETVAL：获取信号灯的值 SETVAL：设置信号灯的值 IPC_RMID：从系统中删除信号灯集合
函数返回值	成功：0 出错：-1

信号量

```
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           */
};
```

- semctl(semid, 0, SETVAL, 3);//设置第0个信号的初始值是3

实验8

- 1、设计一个安装程序，使程序只能有一个运行实例。

进程间通信

- pipe: 具有亲缘关系的进程间，单工，数据在内存中
- fifo: 可用于任意进程间，双工，有文件名，数据在内存
- signal: 唯一的异步通信方式
- msg：常用于cs模式中， 按消息类型访问 ，可有优先级
- shm：效率最高(直接访问内存)，需要同步、互斥机制
- sem：配合使用，用以实现同步和互斥