Linux进程间通讯

Andrew Huang <bluedrum@163.com>

课程内容:

- IPC基本概念
- 管道
- 信号(signal)
- System V IPC 接口
  - System V 消息队列
  - System V 信号量
  - System V 共享内存
- POSIX IPC 接口
  - POSIX 消息队列
  - POSIX 信号量
  - POSIX 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

在上一节,我们主要讲解最常用和重要的信号,POSIX 信号量.在下节我们讲解不太常用的管道和 Sysytem V 的消息队列,信量量和共享内存.

信号

1. 信号(Signal)

信号是在软件层次上对中断机制的一种模拟，在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信

号的到达，事实上，进程也不知道信号到底什么时候到达。

信号是进程间通信机制中唯一的异步通信机制，可以看作是异步通知，通知接收信号的进程有哪些事情发生了。信号机制经过POSIX实时扩展后，功能更加强大，除了基本通知功能外，还可以传递附加信息。

在 Linux 编程当中,经常看程序开头看对信号的使用.是属于比较常用机制了.

- **信号来源** ,信号事件的发生有两个来源:
  - 硬件来源(比如我们按下了键盘或者其它硬件故障),通常由内核进行通知
  - 程序可以用kill,sigqueue,alarm, raise ,settimer 来用触发信号
  - Shell的kill命令也可用来发信号给指定的进程
- 信号的类别和取值
  - Linux信号机制基本上是从Unix系统中继承过来的 ,早期只有SIGRTMIN 个用于通讯,(SIGRTMIN 一般为32)
  - 早期机制上的信号(<32)叫做“不可靠信号”，不可靠是信号有可能会丢失.
  - Linux 为解决可靠性问题,又发展出SIGRTMIN和SIGRTMAX 之间信号,SIGRTMAX 通常为64.这些信号是通过排队保证可靠的
  - 用kill –l 命令可以查看系统中的支持所有信号
- 信号处理函数
  - 早期Linux用signal()来注册处理函数,用kill()来发送信号,只能发信号编号,不能带参数
  - 后来Linux 新增了, 信号安装函数sigation（）以及信号发送函数sigqueue() ,新函数除了能发送信号编号,而且能在带参数发送
- 信号的用途及含义
  - 前32种信号已经有了预定义值，每个信号有了确定的用途及含义，并且每种信号都有各自的缺省动作。如按键盘的CTRL ^C时，会产生SIGINT信号，对该信号的默认反应就是进程终止。
  - 后32个信号表示实时信号，等同于前面阐述的可靠信号。这保证了发送的多个实时信号都被接收
  - 前**32**一般信号都分配特殊意义**,**用户自定义的消息**,**最好选择 **SIGUSR1,SIGUSR2**

名　字	说　明	ANSI C	POSIX.1	SVR4	4.3+BSD	缺省动作
SIGABRT	异常终止(abort)	•	•	•	•	终止w/core
SIGALRM	超时(alarm)		•	•	•	终止
SIGBUS	硬件故障			•	•	终止w/core
SIGCHLD	子进程状态改变		作业	•	•	忽略
SIGCONT	使暂停进程继续		作业	•	•	继续/忽略
SIGEMT	硬件故障			•	•	终止w/core
SIGFPE	算术异常	•	•	•	•	终止w/core
SIGHUP	连接断开		•	•	•	终止
SIGILL	非法硬件指令	•	•	•	•	终止w/core
SIGINFO	键盘状态请求				•	忽略
SIGINT	终端中断符	•	•	•	•	终止
SIGIO	异步I/O			•	•	终止/忽略
SIGIOT	硬件故障			•	•	终止w/core
SIGKILL	终止		•	•	•	终止
SIGPIPE	写至无读进程的管道		•	•	•	终止
SIGPOLL	可轮询事件(poll)			•		终止
SIGPROF	梗概时间超时(setitimer)			•	•	终止
SIGPWR	电源失效/再起动			•		忽略
SIGQUIT	终端退出符		•	•	•	终止w/core
SIGSEGV	无效存储访问	•	•	•	•	终止w/core
SIGSTOP	停止		作业	•	•	暂停进程

(续)

名　字	说　明	ANSI C	POSIX.1	SVR4	4.3+BSD	缺省动作
SIGSYS	无效系统调用			•	•	终止w/core
SIGTERM	终止	•	•	•	•	终止
SIGTRAP	硬件故障			•	•	终止w/core
SIGTSTP	终端挂起符		作业	•	•	停止进程
SIGTTIN	后台从控制tty读		作业	•	•	停止进程
SIGTTOU	后台向控制tty写		作业	•	•	停止进程
SIGURG	紧急情况			•	•	忽略
SIGUSR1	用户定义信号		•	•	•	终止
SIGUSR2	用户定义信号		•	•	•	终止
SIGVTALRM	虚拟时间闹钟(setitimer)			•	•	终止
SIGWINCH	终端窗口大小改变			•	•	忽略
SIGXCPU	超过CPU限制(setrlimit)			•	•	终止w/core
SIGXFSZ	超过文件长度限制(setrlimit)			•	•	终止w/core

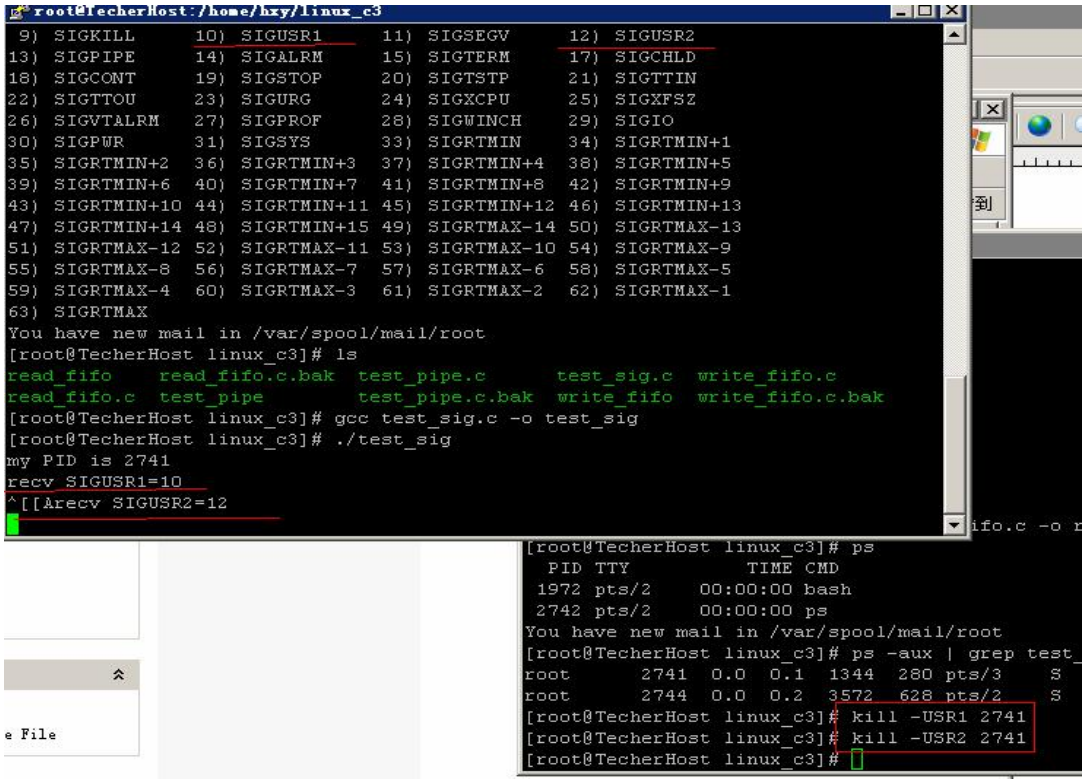
2. 涉及到signal的进程控制函数

- **pause**（让进程暂停直到信号出现）
  - **int pause(void);**
  - pause()会令目前的进程暂停（进入睡眠状态），直到被信号(signal)所中断。
- **sleep**（让进程暂停执行一段时间）
  - **unsigned int sleep(unsigned int seconds);**
  - sleep()会令目前的进程暂停，直到达到参数seconds 所指定的时间，或是被信号所中断。

3. 信号(Signal)的处理流程

- 进程可以通过三种方式来响应一个信号：
  - （1）忽略信号，即对信号不做任何处理，其中，有两个信号不能忽略：SIGKILL及SIGSTOP；
  - （2）捕捉信号。定义信号处理函数，当信号发生时，执行相应的处理函数；
  - （3）执行缺省操作，Linux对每种信号都规定了默认操作
- 信号是异步机制,因此必须提前在系统把信号处理函数注册,这样才能信号发来时,让程序调用处理函数
- 在注册后,可以在另外一个进程用信号发送函数进行信号触发
- 早期处理函数
  - 信号的注册函数 **signal**
  - 信号的发送
    - **int kill(pid_t pid,int signo)**
      对指进程发送信号signo
    - **int raise(int signo)** 对自己发送信号
    - **unsigned int alarm(unsigned int seconds)** 在seconds 后向进程本向发送SIGALRM 信号,一般用于定时处理
    - **int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue));** 它是alarm 的增加版本.
    - **void abort(void);**向进程发送SIGABORT信号，默认情况下进程会异常退出，当然可定义自己的信号处理函数。即使SIGABORT被进程设置为阻塞信号，调用abort()后，SIGABORT仍然能被进程接收。
    - 信号可以由kill命令发送
      - **Kill –USR1 2380** 向进程号为2380的进程发送 SIGUSR1信号
      - Kill 2380 等效于 kill –TERM 2380

- 参考实例 test_sig.c



老的信号的问题

- 老的信号接口signal是非队列的.换句话说,如果短时间大量信号被signal触发,可能一些信号会被丢弃.
- 解决办法之一就是为信号发送加一个缓存队列,所有发送的信号先发送队列之中.再由处理函数从队列取出.这种机制类似的Windows的消息队列.
- 新的信号处理函数
  - kill/alarm 被sigqueue代替
  - signal被sigaction被代替
- 由于历史原因,大量代码仍用老式信号处理函数居多
- 扩展的信号处理函数
  - 增强注册函数 sigaction
    - **int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);**
  - 增强发送信号函数sigqueue
    - **int sigqueue(pid_t pid, int sig, const union sigval val)**
    - 其中sigval 是一个typedef union sigval { int sival_int; void *sival_ptr; }sigval_t; ,
    - 这样可以将参数放入sigval 发送到sigaction 的注册的函数.
- 参见增加信号测试版本
  - recv_sig2.c 对应 send_sig2.c
  - self_sig3.c 自发送样例
  - recv_sig4.c 对send_sig4.c 带参数发送信号

POSIX IPC 接口

1. 两大类应用接口区别

历史上System V系统是Unix一个重要分支,因此LINUX从Unix继承下来一整套IPC通讯机制,但System V IPC存在时间比较老,许多系统都支持,而Posix IPC是新出的标准.很多嵌入式平台只支持System V 的接口

System V的接口相对复杂,而POSIX比较简单,优先选择后者.

- System V内置在glibc中,因此所有使用glibc库的环境都可以使用,POSIX IPC的使用必须链接librt.XXX库,(即使用-lrt参数)

2. POSIX 消息队列

接口名称	目的
mq_open(3RT)	连接到以及创建（可选）命名消息队列
mq_close(3RT)	结束到开放式消息队列的连接
mq_unlink(3RT)	结束到开放式消息队列的连接，并在最后一个进程关闭此队列时将其删除
mq_send(3RT)	将消息放入队列
mq_receive(3RT)	在队列中接收（删除）最早且优先级最高的消息
mq_notify(3RT)	通知进程或线程消息已存在于队列中
mq_setattr(3RT), mq_getattr(3RT)	设置或获取消息队列属性

3. POSIX 消息队列的使用

- 使用头文件 mqueue.h
- 队列数据结构 mqd_t
- 打开队列
  - mqd_t mq_open(const char *name, int oflag, /* unsigned long mode, mq_attr attr */ ...);
- 关闭队列

POSIX 信号量

1. 关于POSIX信号量

- posix 命名信号量可以用于Linux IPC通讯.而匿名信号量因为没有名字，其它进程无法通过名字来访问一个共同的信号量. 因此匿名信号量只用线程之间并发控制
- 相对于System V的信号量，posix 信号量不仅的编程上简单，在操作上也更加安全.
- 信号量主要用于进程（线程）的同步和互斥

2. 互斥与同步概念

- 互斥和同步是两个紧密相关而又容易混淆的概念。
- 互斥：是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。
- 同步：是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源，如“第一类读写者模型”。

posix 命名信号量

1. sem_open:打开命名信号量

- 函数sem_open创建一个新的有名信号量或打开一个已存在的有名信号量。有名信号量总是既可用于线程间的同步，又可以用于进程间的同步。

名称::	sem_open
功能：	创建并初始化有名信号量
头文件：	#include <semaphore.h>
函数原形：	sem_t *sem_open(const char *name,int oflag,/*mode_t mode,unsigned int value*/);
参数：	name 信号量的外部名字 oflag 选择创建或打开一个现有的信号量 mode 权限位 value 信号量初始值
返回值：	成功时返回指向信号量的指针，出错时为 SEM_FAILED

sem_open参数含意

- oflag参数可以是0、O_CREAT（创建一个信号量）或O_CREAT|O_EXCL（如果没有指定的信号量就创建），如果指定了O_CREAT，那么第三个和第四个参数是需要的；其中mode参数指定权限位，value参数指定信号量的初始值，通常用来指定共享资源的书面。该初始不能超过SEM_VALUE_MAX，这个常值必须低于为32767。二值信号量的初始值通常为1，计数信号量的初始值则往往大于1。
- 如果指定了O_CREAT（而没有指定O_EXCL），那么只有所需的信号量尚未存在时才初始化它。所需信号量已存在条件下指定O_CREAT不是一个错误。该标志的意思仅仅是“如果所需信号量尚未存在，那就创建并初始化它”。但是所需信号量等已存在条件下指定O_CREAT|O_EXCL却是一个错误。
- sem_open返回指向sem_t信号量的指针，该结构里记录着当前共享资源的数目。

测试sem_open

```
/*semopen.c*/
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc,char **argv)
{
sem_t *sem;

if(argc!=2)
{
    printf("please input a file name!\n");
    exit(1);
}
sem=sem_open(argv[1],O_CREAT,0644,1);
exit(0);
}
```

- 编译：
  - **gcc –lpthread –o semopen semopen.c**
  - 或者用gcc –lrt –o semopen Semopen.c
  - ./semopen
- 注意，因为信号量是有名的，如果第一次创建需要用O_CREAT参数,第二次创建同名的就会出错，比较保险是首先O_EXCL来查询一次，确认没有同名信号量打开再来创建

2. sem_close:关闭信号量

名称:：	sem_close
功能：	关闭有名信号量
头文件：	#include <semaphore.h>
函数原形：	int sem_close(sem_t *sem);
参数：	sem 指向信号量的指针
返回值：	若成功则返回 0，否则返回-1。

sem_close注意

- 一个进程终止时，内核还对其上仍然打开着的所有有名信号量自动执行这样的信号量关闭操作。不论该进程是自愿终止的还是非自愿终止的，这种自动关闭都会发生。
- 但应注意的是关闭一个信号量并没有将它从系统中删除。这就是说，Posix有名信号量至少是随内核持续的：即使当前没有进程打开着某个信号量，它的值仍然保持。

3. sem_unlink:删除信号量

名称::	sem_unlink
功能：	从系统中删除信号量
头文件：	#include <semaphore.h>
函数原形：	int sem_unlink(count char *name);
参数：	name 信号量的外部名字
返回值：	若成功则返回 0，否则返回-1。

- 命名信号量使用sem_unlink从系统中删除。
- 每个信号量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所指的信号量从文件系统中删除。也就是要等待最后一个sem_close发生。

sem_unlink:实例

```
/*semunlink.c*/
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
```

```
 /*接上一页*/
int main(int argc,char **argv)
{
sem_t *sem;
int val;
if(argc!=2)
{
    printf("please input a file name!\n");
    exit(1);
}
if((sem_unlink(argv[1]))!=0)
    perror("sem_unlink");
else
    printf("success");
exit(0);
}
```

4. sem_getvalue:取信号量值

名称:：	sem_getvalue
功能：	测试信号量
头文件：	#include <semaphore.h>
函数原形：	int sem_getvalue(sem_t *sem,int *valp);
参数：	sem 指向信号量的指针
返回值：	若成功则返回 0，否则返回-1。

- 在由 valp 指向的正数中返回所指定信号量的当前值。如果该信号量当前已上锁，那么返回值或为 0，或为某个负数，其绝对值就是等待该信号量解锁的线程数。

测试信号量的值

```
/*semgetvalue.c*/
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
```

```
/*接上一页*/
int main(int argc,char **argv)
{
sem_t *sem;
int val;

if(argc!=2)
{
    printf("please input a file name!\n");
    exit(1);
}
sem=sem_open(argv[1],0);
sem_getvalue(sem,&val);
printf("getvalue:value=%d\n",val);
exit(0);
}
```

5. sem_wait:等待信号量解锁

名称:：	sem_wait/sem_trywait
功能：	等待共享资源
头文件：	#include <semaphore.h>
函数原形：	int sem_wait(sem_t *sem); int sem_trywait(sem_t *sem);
参数：	sem 指向信号量的指针
返回值：	若成功则返回 0，否则返回-1。

- 我们可以用sem_wait来申请共享资源，sem_wait函数可以测试所指定信号量的值，如果该值大于0，那就将它减1并立即返回。我们就可以使用申请来的共享资源了。如果该值等于0，调用线程就被进入睡眠状态，直到该值变为大于0，这时再将它减1，函数随后返回。

sem_wait/sem_trywait区别

- sem_wait()是不可撤消的原子操作,即如果进程执行到sem_wait()时,信号量还有资源,则进程直接执行行.如果没有足够资源,则进程立刻被阻塞.直到获得资源或者信号量被删除为止.
- sem_trywait()只是尝试测试一下信号量是否有足够的资源.它立即返回并给一个值给当前进程.当前进程可以根据值来做下一步操作.

6. sem_post:释放信号量

名称::	sem_post
功能：	释放信号量资源
头文件：	#include <semaphore.h>
函 数 原形：	int sem_post(sem_t *sem); int sem_getvalue(sem_t *sem,int *valp);
参数：	sem 指向信号量的指针
返回值：	若成功则返回 0，否则返回-1。

- 一个线程使用完某个信号量时，它应该调用sem_post来告诉系统申请的资源已经用完。本函数和sem_wait函数的功能正好相反，它把所指定的信号量的值加1，然后唤醒正在等待该信号量值变为正数的任意线程。

7. 关于信号量的命名

- 必须是一个已经存在路径名或目录名.
- 在使用时不能用绝对路径,只能用相对路径
  - 这是Linux的BUG
- 当信号量打开时,不能删除,或覆盖对应文件名
  - 只有用sem_unlink才能真接删除信号量,解释对文件的控制

8. 使用posix命名信号量

- Posix有名信号量的值是随内核持续的。也就是说，一个进程创建了一个信号量，这个进程结束后，这个信号量还存在，并且信号量的值也不会改变。
- 当持有某个信号量锁的进程没有释放这个信号量就把进程终止时，内核并不给该信号量解锁。

posix匿名信号量

1. posix匿名信号量

- posix命名信号量。这些信号量由一个name参数标识，它通常指代文件系统中的某个文件。然而posix也提供基于内存的信号量，它们由应用程序分配信号量的内存空间，然后由系统初始化它们的值。
- 因为没有全局的名字，所以只能在同一进程中使用．多用于多线程的并发控制.

2. sem_init:初始化匿名信号量

名称::	sem_init/sem_destroy
功能：	初始化/关闭信号等
头文件：	#include <semaphore.h>
函 数 原形：	int sem_init(sem_t *sem,int shared,unsigned int value); int sem_getvalue(sem_t *sem);

参数：	sem　指向信号量的指针 shared　作用范围 value　信号量初始值
返回值：	若成功则返回 0，否则返回-1。

- 基于内存的信号量是由sem_init初始化的。sem参数指向必须由应用程序分配的sem_t变量。如果shared为0，那么待初始化的信号量是在同一进程的各个线程共享的，否则该信号量是在进程间共享的。当shared为零时，该信号量必须存放在即将使用它的所有进程都能访问的某种类型的共享内存中。跟sem_open一样，value参数是该信号量的初始值。
- 使用完一个基于内存的信号量后，我们调用sem_destroy关闭它。
- 除了sem_open和sem_close外，其它的poisx有名信号量函数都可以用于基于内存的信号量。

3. 命名信号量与匿名信号量区别

- sem_open不需要类型与shared的参数，有名信号量总是可以在不同进程间共享的。
- sem_init不使用任何类似于O_CREAT标志的东西，也就是说，sem_init总是初始化信号量的值。因此，对于一个给定的信号量，我们必须小心保证只调用一次sem_init。
- sem_open返回一个指向某个sem_t变量的指针，该变量由函数本身分配并初始化。但sem_init的第一个参数是一个指向某个sem_t变量的指针，该变量由调用者分配，然后由sem_init函数初始化。
- posix有名信号量是通过内核持续的，一个进程创建一个信号量，另外的进程可以通过该信号量的外部名（创建信号量使用的文件名）来访问它。posix基于内存的信号量的持续性却是不定的，如果基于内存的信号量是由单个进程内的各个线程共享的，那么该信号量就是随进程持续的，当该进程终止时它也会消失。如果某个基于内存的信号量是在不同进程间同步的，该信号量必须存放在共享内存区中，这要只要该共享内存区存在，该信号量就存在。

多线程加锁

```
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <pthread.h>
#incude <stdlib.h>

void *thread_function(void *arg); /*线程入口函数*/
void print(void); /*共享资源函数*/

sem_t bin_sem; /*定义信号灯*/
int value; /*定义信号量的灯*/
```

```
/*接上一页*/
int main()
{
int n=0;
pthread_t a_thread;

if((sem_init(&bin_sem,0,2))!=0) /*初始化信号灯，初始值为 2*/
{
    perror("sem_init");
    exit(1);
}
while(n++<5) /*循环创建 5 个线程*/
{
    if((pthread_create(&a_thread,NULL,thread_function,NULL))!=0)
{
    perror("Thread creation failed");
    exit(1);
}
}
pthread_join(a_thread,NULL);/*等待子线程返回*/
}

void *thread_function(void *arg)
{
sem_wait(&bin_sem); /*等待信号灯*/
print();
sleep(1);
sem_post(&bin_sem); /*挂出信号灯*/
printf("I finished,my pid is %d\n",pthread_self());
pthread_exit(arg);
}

void print()
{
printf("I get it,my tid is %d\n",pthread_self());
sem_getvalue(&bin_sem,&value); /*获取信号灯的值*/
printf("Now the value have %d\n",value);
}
```

POSIX 信号量实例

1. 信号量的使用流程

- 创建一个信号量
  - sem_open()
- 从信号量取一个资源
  - sem_wait()
- 释放一个信号量的资源
  - sem_post()
- 关闭一个信号量
  - sem_close()
- 删除一个信号量
  - sem_unlink()

2. 信号量使用实例

- 下面就是应用Posix命名信号量的一个小程序。用它来限制访问共享代码的进程数目

```
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>

void print(pid_t);
sem_t *sem; /*定义Posix有名信号量*/
int val; /*定义信号量当前值*/

int main(int argc,char *argv[])
{
int n=0;

if(argc!=2)
{
    printf("please input a file name!\n");
    exit(1);
}
sem=sem_open(argv[1],O_CREAT,0644,2); /*打开一个信号量，初值设为2*/

while(n++<5) /*循环创建5个子进程，使它们同步运行*/
{
```

```
/*接上一页
if(fork()==0)
   {
      sem_wait(sem); /*申请信号量*/
      print(getpid()); /*调用共享代码段*/
      sleep(1);
      sem_post(sem); /*释放信号量*/
      printf("I'm finished,my pid is %d\n",getpid());
      return 0;
   }
wait(); /*等待所有子进程结束*/
sem_close(sem);
sem_unlink(argv[1]);
exit(0);
}

void print(pid_t pid)
{
printf("I get it,my pid is %d\n",pid);
sem_getvalue(sem,&val);
printf("Now the value have %d\n",val);
}
```

POSIX 共享内存

1. 共享内存

- 共享内存可以说是最有用的进程间通信方式，也是最快的 IPC 形式。两个不同进程 A、B 共享内存的意思是，同一块物理内存被映射到进程 A、B 各自的进程地址空间。进程 A 可以即时看到进程 B 对共享内存中数据的更新，反之亦然。
- 由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。
- 共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据[1]：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。
- 共享内存广泛被应用数据库系统实现到驱动程序各种应用

2. 共享内存的实现

- Linux 有三种共享内存实现机制
  - mmap()
  - System V 共享内存
  - Posix 共享内存

- 前两者较为常用

- 系统调用mmap() 通过映射一个普通文件实现共享内存。系统V 则是通过映射特殊文件系统shm 中的文件实现进程间的共享内存通信。也就是说，每个共享内存区域对应特殊文件系统shm 中的一个文件（这是通过shmid_kernel 结构联系起来的)

1)mmap

- mmap 函数把一个文件或一个Posix 共享内存区对象映射到调用进程的地址空间。使用该函数有三个目的：
  - 1. 使用普通文件以提供内存映射I/O
  - 2. 使用特殊文件以提供匿名内存映射。
  - 3. 使用shm_open 以提供无亲缘关系进程间的Posix 共享内存区

mmap- 创建一个共享内存区

名称:：	mmap
功能：	把 I/O 文件映射到一个存储区域中
头文件：	#include <sys/mman.h>
函数原形：	void *mmap(void *addr,size_t len,int prot,int flag,int filedes,off_t off);
参数：	addr 指向映射存储区的起始地址,addr 参数用于指定映射存储区的起始地址。通常将其设置为 NULL，这表示由系统选择该映射区的起始地址。 len 映射的字节 prot 对映射存储区的保护要求 flag flag 标志位 filedes 要被映射文件的描述符，在映射该文件到一个地址空间之前，先要打开该文件。len 是映射的字节数。 off 要映射字节在文件中的起始偏移量,通常将其设置为 0。
返回值：	若成功则返回映射区的起始地址，若出错则返回 MAP_FAILED

mmap 参数说明

- prot参数说明对映射存储区的保护要求。可将prot参数指定为PROT_NONE,或者是PROT_READ(映射区可读),PROT_WRITE（映射区可写）,PROT_EXEC（映射区可执行）任意组合的按位或，也可以是PROT_NONE(映射区不可访问)。对指定映射存储区的保护要求不能超过文件open模式访问权限。
- flag参数影响映射区的多种属性：
  - MAP_FIXED 返回值必须等于addr.因为这不利于可移植性，所以不鼓励使用此标志。
  - MAP_SHARED 这一标志说明了本进程对映射区所进行的存储操作的配置。此标志指定存储操作修改映射文件。
  - MAP_PRIVATE 本标志导致对映射区建立一个该映射文件的一个私有副本。所有后来对该映射区的引用都是引用该副本，而不是原始文件。
  - 要注意的是必须指定MAP_FIXED或MAP_PRIVATE标志其中的一个，指定前者是对

存储映射文件本身的一个操作，而后者是对其副本进行操作。

2)unmmap 取消一个共享内存

名称:：	unmmap
功能：	解除存储映射
头文件：	#include <sys/mman.h>
函数原形：	int munmap(caddr_t addr,size_t len);
参数：	addr 指向映射存储区的起始地址,其中 addr 参数是由 mmap 返回的地址 ， len 映射的尺寸。再次访问这些地址导致向调用进程产生一个 SIGSEGV 信号。
返回值：	若成功则返回 0，若出错则返回-1

3)msyns 内存与数据同步

名称:：	msync
功能：	同步文件到存储器
头文件：	#include <sys/mman.h>
函数原形：	int msync(void *addr,size_t len,int flags);
参数：	addr 指向映射存储区的起始地址 len 映射的字节 prot flags
返回值：	若成功则返回 0，若出错则返回-1

3. posix 共享内存

- posix 共享内存区涉及两个步骤：
  - 1 、指定一个名字参数调用shm_open, 以创建一个新的共享内存区对象或打开一个以存在的共享内存区对象。
  - 2 、调用mmap 把这个共享内存区映射到调用进程的地址空间。传递给shm_open 的名字参数随后由希望共享该内存区的任何其他进程使用。

1)shm_open 打开共享内存

名称:：	shm_open
功能：	打开或创建一个共享内存区
头文件：	#include <sys/mman.h>
函数原形：	int shm_open(const char *name,int oflag,mode_t mode);

参数：	name　共享内存区的名字 cflag　标志位, oflag 参数必须含有 O_RDONLY 和 O_RDWR 标志，还可以指定如下标志：O_CREAT,O_EXCL 或 O_TRUNC. mode　权限位
返回值：	shm_open 的返回值是一个整数描述字，它随后用作 mmap 的第五个参数

2)shm_unlink删除一个共享内存区

名称::	shm_unlink
功能：	删除一个共享内存区
头文件：	#include <sys/mman.h>
函数原形：	int shm_unlink(const char *name);
参数：	name　共享内存区的名字
返回值：	成功返回 0，出错返回-1

共享内存的应用

- 共享内存是在进程之间开辟一个原始的内存buffer, 因此是进程之间传输大量数据最快的方式
  - 大量数据库在Unix/Linux 上实现内存分配, 都是采用共享内存自行分配
- 但共享内存没有同步和锁机制, 所以必须配合信号量或互斥锁进行加锁
- 参见共享内存实现服务器和客户端例子
- 采用共享内存+mmap 的例子
  - shm_server.c
  - shm_client.c

共享内存实例

- shm_open+mmap 实现服务器和客户端例子
  - shm_server.c
  - shm_client.c
  - shm_open 需要库rt 进行链接
- 文件+mmap 实现服务器和客户端例
  - map_server.c
    - map_client.c

```
gcc –lrt –o shm_server shm_server.c
gcc –lrt –o shm_client shm_client.c
./shm_server test&
./shm_client test
```

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