

---

# Linux进程间通信

广嵌



# Linux下进程间通信概述

---

- ▶ 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。

## UNIX平台进程通信方式

- ▶ 早期进程间通信方式
- ▶ AT&T的贝尔实验室，对Unix早期的进程间通信进行了改进和扩充，形成了“system V IPC”，其通信进程主要局限在单个计算机内
- ▶ BSD(加州大学伯克利分校的伯克利软件发布中心)，跳过了该限制，形成了基于套接字(socket)的进程间通信机制



# Linux下进程间通信概述

---

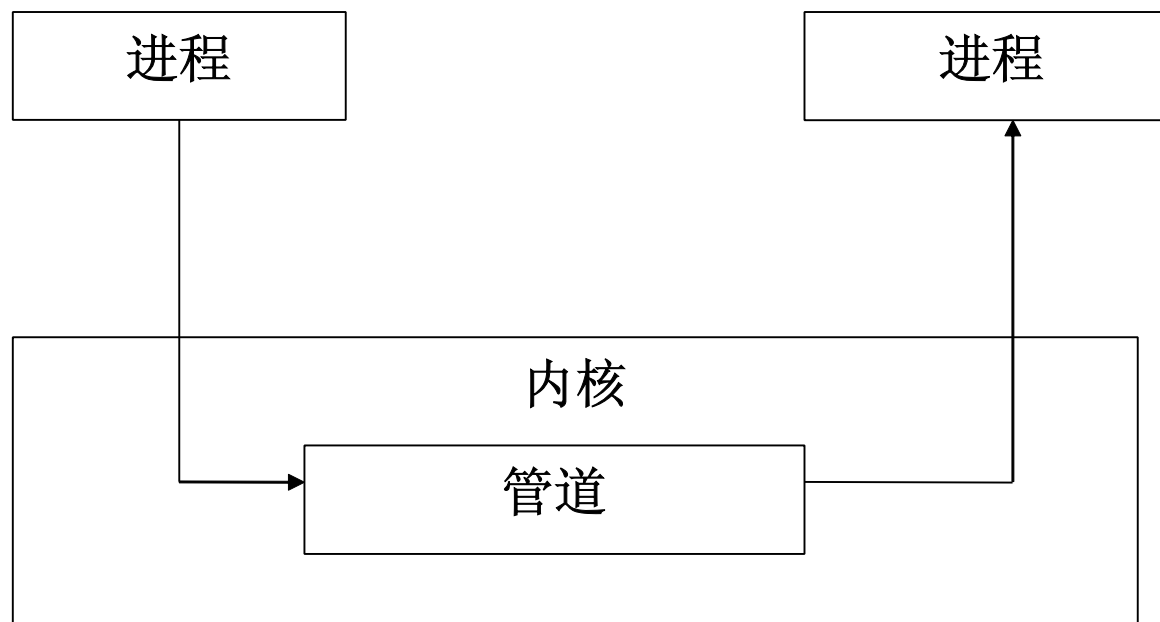
- ▶ 常用的进程间通信方式
  - ▶ 传统的进程间通信方式  
无名管道(pipe)、有名管道(fifo)和信号(signal)
  - ▶ **System V IPC**对象  
共享内存(share memory)、消息队列(message queue)和信号量(semaphore)
  - ▶ **BSD**  
套接字(socket)



# 无名管道

---

## ► 抽象



## 无名管道

---

- ▶ 无名管道是Linux中进程间通信的一种方式。
  - 它只能用于具有亲缘关系的进程之间的通信（也就是父子进程或者兄弟进程之间）。
  - 它是一个半双工的通信模式，具有固定的读端和写端。
  - 管道也可以看成是一种特殊的文件，对于它的读写也可以使用普通的read()和write()等函数。但是它不是普通的文件，并不属于其他任何文件系统，即只存在于内核的内存空间中。

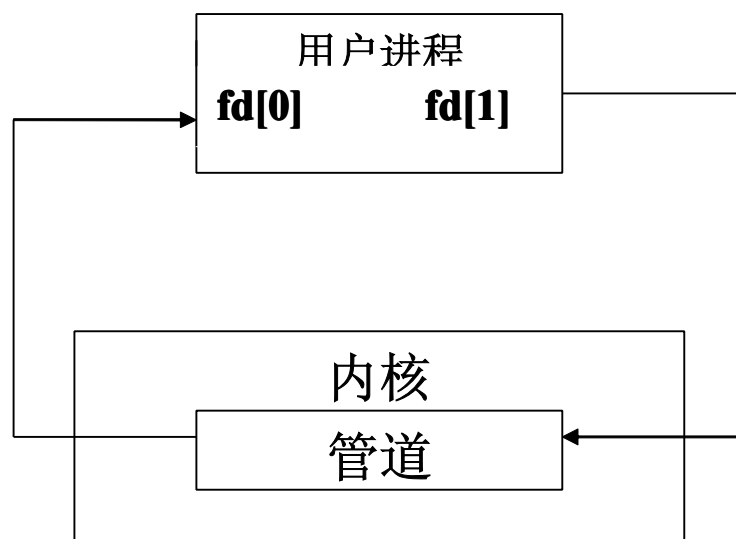


- 
- ▶ 管道是基于文件描述符的通信方式，当一个管道建立时，它会创建两个文件描述符`fds[0]`和`fds[1]`，其中
    - ◇ `fds[0]`固定用于读管道
    - ◇ `fd[1]`固定用于写管道这样就构成了一个半双工的通道。
  - ▶ 创建管道可以通过调用`pipe()`来实现。
  - ▶ 管道关闭时只需使用普通的`close()`函数逐个关闭各个文件描述符。
- 



所需头文件	<code>#include &lt;unistd.h&gt;</code>
函数原型	<code>int pipe(int fd[2])</code>
函数传入值	<code>fd[2]</code> : 管道的两个文件描述符, 之后就可以直接操作这两个文件描述符
函数返回值	成功: 0
	出错: -1

► 构成了一个半双工的通道。



---

## 示例

- ▶ `#include<unistd.h>`
- ▶ `#include<stdlib.h>`
- ▶ `#include<stdio.h>`
- ▶ `#include<string.h>`
- ▶ `int main()`
- ▶ `{`
- ▶ `int data_size;`
- ▶ `int pfd[2];`
- ▶ `char data[]="123";`
- ▶ `char buffer[1024];`
- ▶ `memset(buffer,'\0',sizeof(buffer));`
- ▶





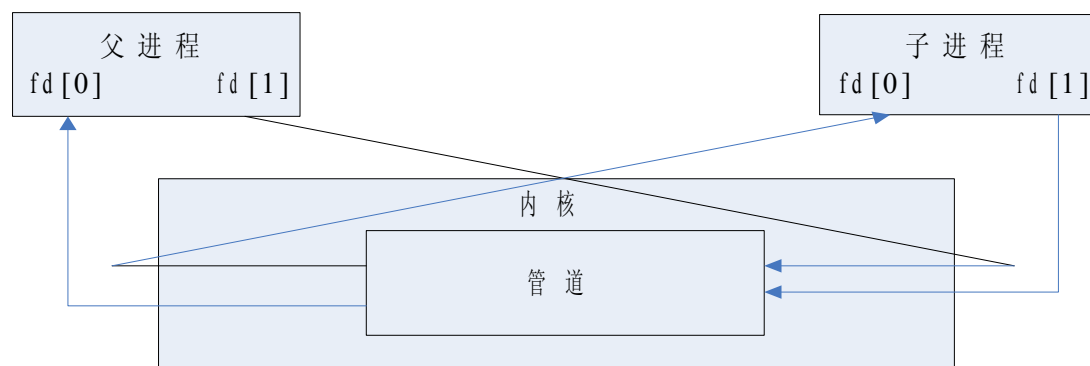
---

```
▶ if(pipe(pfd)<0)
▶ {
▶     printf("pipe create error\n");
▶ }
▶ else
▶ {
▶     data_size=write(pfd[1],data,strlen(data));
▶     printf("wrote %d bytes\n",data_size);
▶     data_size=read(pfd[0],buffer,1024);
▶     printf("read %d bytes: %s\n",data_size,buffer);
▶ }
▶ close(pfd[0]);
▶ close(pfd[1]);
▶ }
```

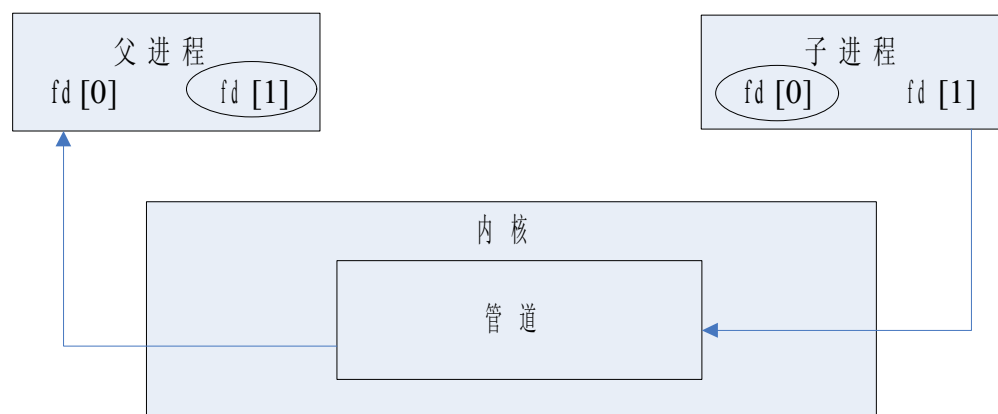


# 父子进程之间的管道通信

- ▶ 用pipe()函数创建的管道两端处于一个进程中，由于管道是主要用于在不同进程间通信的，因此这在实际应用中没有太大意义。实际上，通常先是创建一个管道，再通过fork()函数创建一子进程，该子进程会继承父进程所创建的管道。



- ▶ 父子进程分别拥有自己的读写通道，为了实现父子进程之间的读写，只需把无关的读端或写端的文件描述符关闭即可。此时，父子进程之间就建立起了一条“子进程写入父进程读取”的通道。



## 无名管道读写注意点

---

- ▶ 当管道中无数据时，读操作会阻塞
- ▶ 向管道中写入数据时，linux将不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读走管道缓冲区中的数据，那么写操作将会一直阻塞。
- ▶ 只有在管道的读端存在时，向管道中写入数据才有意义。否则，向管道中写入数据的进程将收到内核传来的SIGPIPE信号(通常Broken pipe错误)。
- ▶ 对一个关闭写端的管道做读操作时将返回0



---

▶ 练习

创建管道实现两进程的通信，一进程把数据写入管道，另一进程读出数据，打印。（`pipe_rw.c`）



## FIFO(有名管道)

---

- ▶ 无名管道只能用于具有亲缘关系的进程之间，这就限制了无名管道的使用范围
- ▶ 有名管道可以使互不相关的两个进程互相通信。
- ▶ 有名管道可以通过路径名来指出，并且在文件系统中可见
- ▶ 进程通过文件IO来操作有名管道
- ▶ 有名管道遵循先进先出规则
- ▶ 不支持如lseek() 操作



## ▶ 创建有名管道的函数原型

所需头文件	#include <sys/types.h>+ #include <sys/stat.h>	
函数原型	int mkfifo(const char *filename, mode_t mode)	
函数传入值	filename: 要创建的管道	
函数传入值	mode:	O_RDONLY: 读管道
		O_WRONLY: 写管道
		O_RDWR: 读写管道
		O_NONBLOCK: 非阻塞
		O_CREAT: 如果该文件不存在, 那么就创建一个新的文件, 并用第三个参数为其设置权限
		O_EXCL: 如果使用 O_CREAT 时文件存在, 那么可返回错误消息。这一参数可测试文件是否存在
函数返回值	成功: 0	
	出错: -1	

mkfifo("FIFO", O\_CREAT|O\_EXCL)<0 或 mkfifo("/tmp/myfifo", 0777)



## 有名管道

---

- ▶ 在创建管道成功之后，就可以使用`open()`、`read()`和`write()`这些函数了。
- ▶ 与普通文件的开发设置一样，对于为读而打开的管道可在`open()`中设置`O_RDONLY`，对于为写而打开的管道可在`open()`中设置`O_WRONLY`，在这里与普通文件不同的是阻塞问题。
- ▶ 由于普通文件的读写时不会出现阻塞问题，而在管道的读写中却有阻塞的可能，这里的非阻塞标志可以在`open()`函数中设定为`O_NONBLOCK`。





---

▶ 对于O\_RDONLY、O\_WRONLY、O\_NONBLOCK有4种组合方式:

①open(const char \*path,O\_RDONLY)

在这种情况下, open 调用将阻塞, 除非有一个 进程以写方式打开同一个 FIFO,否则它不会返回

②open(const char \*path,O\_RDONLY|O\_NONBLOCK)

即使没有其它进程以写方式打开FIFO, 这open调用也将成功并马上返回

③open(const char \*path,O\_WRONLY)

open调用将阻塞, 直到有一个进程以读方式打开同一个 FIFO为止。

④open(const char \*path,O\_WRONLY|O\_NONBLOCK)

open调用总是立刻返回, 便如果没有进程以读方式打开FIFO文件, open调用将返回一个错误 (-1) 并且FIFO也不会被打开。



# 有名管道

---

## ▶ 对于读进程

- 若该管道是阻塞打开，且当前FIFO内没有数据，则对读进程而言将一直阻塞到有数据写入。
- 若该管道是非阻塞打开，则不论FIFO内是否有数据，读进程都会立即执行读操作。

## ▶ 对于写进程

- 若该管道是阻塞打开，则写操作将一直阻塞到数据可以被写入。
- 若该管道是非阻塞打开而不能写入全部数据，则读操作进行部分写入或者调用失败。



- 
- ▶ `/* read*/`
  - ▶ `#define FIFO /tmp/my_fifo`
  - ▶ `mkfifo(FIFO,0777)`
  - ▶ `open(FIFO,O_RDONLY|O_NONBLOCK);`
  - ▶ `read`
  - ▶ `/*write*/`
  - ▶ `open(FIFO_SERVER,O_WRONLY|O_NONBLOCK)`
  - ▶ `write()`



---

# 共享内存

---

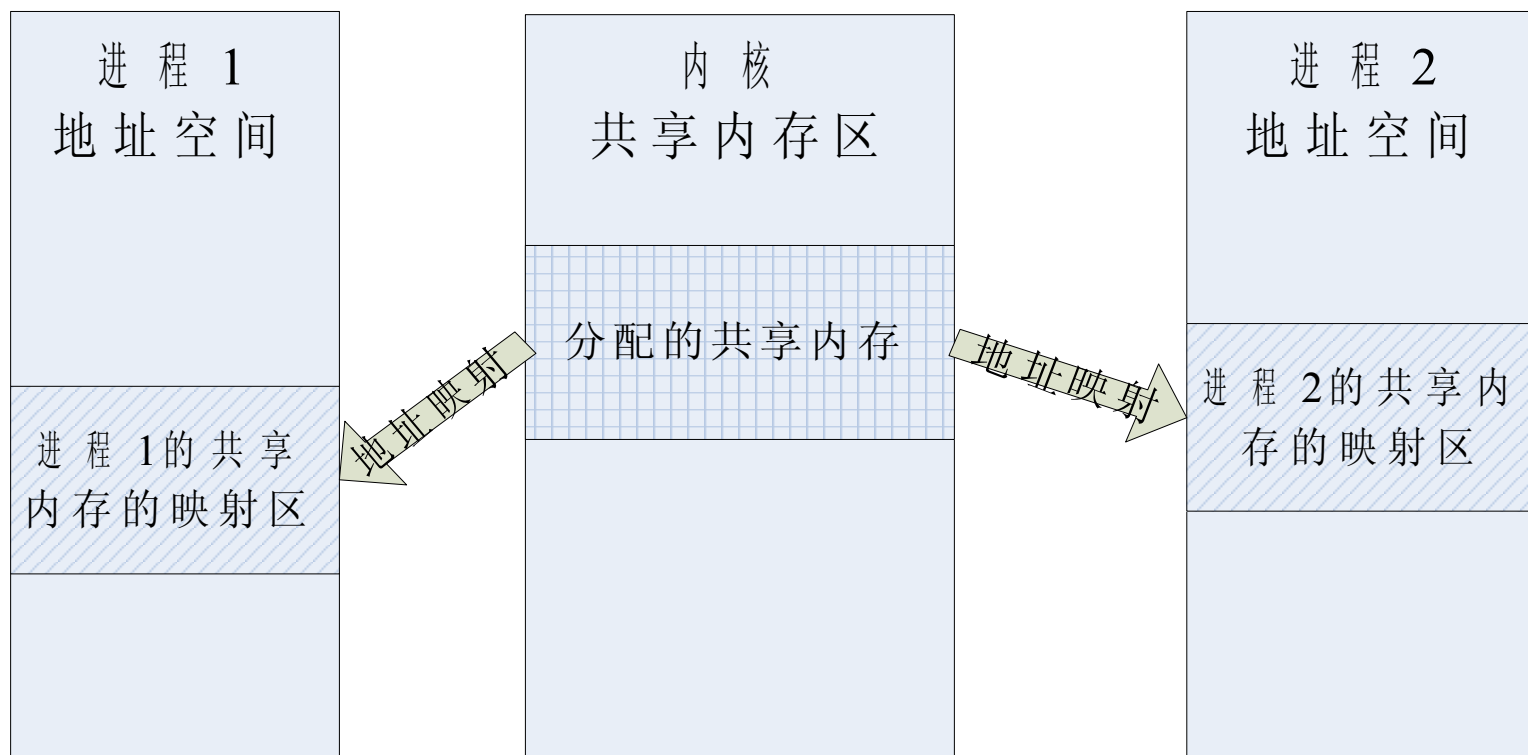


# 共享内存

---

- ▶ 共享内存是一种最为高效的进程间通信方式，进程可以直接读写内存，而不需要任何数据的拷贝
- ▶ 为了在多个进程间交换信息，内核专门留出了一块内存区，可以由需要访问的进程将其映射到自己的私有地址空间
- ▶ 进程就可以直接读写这一内存区而不需要进行数据的拷贝，从而大大提高的效率。
- ▶ 由于多个进程共享一段内存，因此也需要依靠某种同步机制，如互斥锁和信号量等





# 共享内存实现

---

- ▶ 共享内存的使用包括如下步骤：
  - ▶ 创建/打开共享内存
  - ▶ 映射共享内存，即把指定的共享内存映射到进程的地址空间用于访问
  - ▶ 撤销共享内存映射
  - ▶ 删除共享内存对象



## ► 创建/打开共享内存

所需头文件	<code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;sys/ipc.h&gt;</code> <code>#include &lt;sys/shm.h&gt;</code>
函数原型	<code>int shmget(key_t key, int size, int shmflg)</code>
函数传入值	<b>key:</b> 共享内存的键值，多个进程可以通过它访问同一个共享内存，其中有个特殊值 <code>IPC_PRIVATE</code> 。它用于创建当前进程的私有共享内存。
	<b>size:</b> 共享内存区大小
	<b>shmflg:</b> 同 <code>open()</code> 函数的权限位，也可以用八进制表示法
函数返回值	成功：共享内存段标识符
	出错：-1



---

## ► 映射共享内存

所需头文件	<code>#include &lt;sys/types.h&gt;↓</code> <code>#include &lt;sys/ipc.h&gt;↓</code> <code>#include &lt;sys/shm.h&gt;</code>	
函数原型	<code>char *shmat(int shmid, const void *shmaddr, int shmflg)</code>	
函数传入值	<b>shmid</b> : 要映射的共享内存区标识符	
	<b>shmaddr</b> : 将共享内存映射到指定地址（若为 0 则表示系统自动分配地址并把该段共享内存映射到调用进程的地址空间）	
	<b>shmflg</b>	<b>SHM_RDONLY</b> : 共享内存只读 默认 0: 共享内存可读写
函数返回值	成功: 被映射的段地址	
	出错: -1	



---

▸ 撤销共享内存映射

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



## ▶ 删除共享内存对象

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**

---

```
/*shm_write.c*?+  
#include<sys/ipc.h>+  
#include<sys/shm.h>+  
#include<sys/types.h>+  
#include<unistd.h>+  
#include <string.h>+  
#include<stdio.h>+  
+  
typedef struct+  
{+  
    char name[5];+  
    int age;+  
}people;+  
+  
main(int argc, char **argv)+  
{+
```



```
int shm_id,i;↵
char temp;↵
people *p_map;↵
char* shmname="/dev/shm/myshm";↵
key_t key=ftok(shmname,0);↵
shm_id=shmget(key,4096,IPC_CREAT);↵
if(shm_id==-1)↵
    perror("error shmget");↵
    return;↵
}↵
p_map=(people*)shmat(shm_id,NULL,0);↵
temp='a';↵
for(i=0;i<8;i++)↵
{
    ↵
    temp+=1;↵
    memcpy((*(p_map+i)).name,&temp,1);↵
    (*(p_map+i)).age=18+i;↵
}↵
if(shmdt(p_map)==-1)↵
    perror("error shmdt");↵
}↵
```



```
/*shm_read.c*/  
#include<sys/ipc.h>  
#include<sys/shm.h>  
#include<sys/types.h>  
#include<unistd.h>  
#include <string.h>  
#include<stdio.h>  
  
typedef struct  
{  
    char name[5];  
    int age;  
}people;  
  
main(int argc, char **argv)  
{  
    int shm_id,i;  
    char temp;
```



```

people *p_map;
char* shmname="/dev/shm/myshm";
key_t key=ftok(shmname, 0);
shm_id=shmget(key, 4096, IPC_CREAT);
if(shm_id==-1)
    perror("error shmget");
    return;

}

p_map=(people *)shmat(shm_id, NULL, 0);
temp='a';
for(i=0; i<8; i++)
{
    printf("name: %s                age: %d\n", (*p_map+i).name, (*p_map+i).age);
}

if(shmdt(p_map)==-1)
    perror("error shmdt");

}

```

编译后，首先运行程序一，再运行程序二。结果如下：

```
[root@localhost shm]# ./shm_write
[root@localhost shm]# ./shm_rread
name:b   age:18
name:c   age:19
name:d   age:20
name:e   age:21
name:f   age:22
name:g   age:23
name:h   age:24
name:i   age:25

```

程序运行结果：





---

# 信号量



## 信号量概述


---

- ▶ 在多任务操作系统环境下，在不同进程之间，为了争夺有限的系统资源（硬件或软件资源）会进入竞争状态，这就是进程之间的互斥关系。



- 
- ▶ 信号量是用来解决进程之间的同步与互斥问题的一种进程之间通信机制。
  - ▶ 信号量对应于某一种资源，取一个非负的整型值。
  - ▶ 信号量值指的是当前可用的该资源的数量，若它等于0则意味着目前没有可用的资源。



- 
- ▶ PV原子操作的具体定义为：
  - ▶ P操作：如果有可用的资源（信号量值 $>0$ ），则占用一个资源（给信号量值减去一，进入临界区代码）；如果没有可用的资源（信号量值等于0），则被阻塞到，直到系统将资源分配给该进程（进入等待队列，一直等到资源轮到该进程）。
  - ▶ V操作：如果在该信号量的等待队列中有进程在等待资源，则唤醒一个阻塞进程。如果没有进程等待它，则释放一个资源（给信号量值加一）。
- 
- 

## 信号量的使用

---

- ▶ 第一步：创建信号量或获得在系统已存在的信号量，此时需要调用`semget()`函数。不同进程通过使用同一个信号量键值来获得同一个信号量。
- ▶ 第二步：初始化信号量，此时使用`semctl()`函数的`SETVAL`操作。当使用二维信号量时，通常将信号量初始化为1。



- 
- ▶ 第三步：进行信号量的PV操作，此时调用semop()函数。这一步是实现进程之间的同步和互斥的核心工作部分。
  - ▶ 第四步：如果不需要信号量，则从系统中删除它，此时使用semctl()函数的IPC\_RMID操作。此时需要注意，在程序中不应该出现对已经被删除的信号量的操作。



所需头文件	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ipc.h&gt; #include &lt;sys/sem.h&gt;</pre>
函数原型	<pre>int semget(key_t key, int nsems, int semflg)</pre>
函数传入值	<p><b>key:</b> 信号量的键值，多个进程可以通过它访问同一个信号量，其中有个特殊值 <b>IPC_PRIVATE</b>。它用于创建当前进程的私有信号量。</p>
	<p><b>nsems:</b> 需要创建的信号量数目，通常取值为 <b>1</b>。</p>
	<p><b>semflg:</b> 同 <b>open()</b> 函数的权限位，也可以用八进制表示法，其中使用 <b>IPC_CREAT</b> 标志创建新的信号量，即使该信号量已经存在（具有同一个键值的信号量已在系统中存在），也不会出错。如果同时使用 <b>IPC_EXCL</b> 标志可以创建一个新的唯一的信号量，此时如果该信号量已经存在，该函数会返回出错。</p>
函数返回值	<p>成功：信号量标识符，在信号量的其他函数中都会使用该值。</p>
	<p>出错：<b>-1</b></p>



所需头文件	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ipc.h&gt; #include &lt;sys/sem.h&gt;</pre>
函数原型	<pre>int semctl(int semid, int semnum, int cmd, union semun arg)</pre>
函数传入值	<p><b>semid:</b> semget()函数返回的信号量标识符。</p> <p><b>semnum:</b> 信号量编号，当使用信号量集时才会被用到。通常取值为 0，就是使用单个信号量（也是第一个信号量）。</p> <p><b>cmd:</b> 指定对信号量的各种操作，当使用单个信号量（而不是信号量集）时，常用的有以下几种：</p> <p><b>IPC_STAT:</b> 获得该信号量（或者信号量集合）的 semid_ds 结构，并存放在由第四个参数 arg 的 buf 指向的 semid_ds 结构中。semid_ds 是在系统中描述信号量的数据结构。</p> <p><b>IPC_SETVAL:</b> 将信号量值设置为 arg 的 val 值。</p> <p><b>IPC_GETVAL:</b> 返回信号量的当前值。</p> <p><b>IPC_RMID:</b> 从系统中，删除信号量（或者信号量集）</p> <p><b>arg:</b> 是 union semun 结构，该结构可能在某些系统中并不给出定义，此时必须由程序员自己定义。</p> <pre>union semun {     int val;     struct semid_ds *buf;     unsigned short *array; }</pre>
函数返回值	<p>成功：根据 cmd 值的不同而返回不同的值。</p> <p><b>IPC_STAT、IPC_SETVAL、IPC_RMID:</b> 返回 0</p> <p><b>IPC_GETVAL:</b> 返回信号量的当前值</p>





所需头文件	<pre>#include &lt;sys/types.h&gt; #include &lt;sys/ipc.h&gt; #include &lt;sys/sem.h&gt;</pre>
函数原型	<pre>int semop(int semid, struct sembuf *sops, size_t nsops)</pre>
函数传入值	<p><b>semid:</b> semget()函数返回的信号量标识符。</p> <p><b>sops:</b> 指向信号量操作数组，一个数组包括以下成员：</p> <pre>struct sembuf {     short sem_num; /* 信号量编号，使用单个信号量时，通常取值为 0 */     short sem_op; /* 信号量操作：取值为-1 则表示 P 操作，取值为+1 则表示 V 操作 */     short sem_flg; /* 通常设置为 SEM_UNDO。这样在进程没释放信号量而退出时，系统自动释放该进程中未释放的信号量 */ }</pre> <p><b>nsops:</b> 操作数组 sops 中的操作个数（元素数目），通常取值为 1（一个操作）</p>
函数返回值	<p>成功：信号量标识符，在信号量的其他函数中都会使用该值。</p> <p>出错：-1</p>



---

▶ 例子：信号量例子



---

# 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
- ▶ 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

---

- ▶ 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。



## 消息队列（2）

---

所需头文件	<code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;sys/ipc.h&gt;</code> <code>#include &lt;sys/shm.h&gt;</code>
函数原型	<code>int msgget(key_t key, int msgflg)</code>
函数传入值	<b>key</b> : 消息队列的键值，多个进程可以通过它访问同一个消息队列，其中有个特殊值 <code>IPC_PRIVATE</code> 。它用于创建当前进程的私有消息队列。
	<b>msgflg</b> : 权限标志位
函数返回值	成功: 消息队列 ID
	出错: <code>-1</code>

在设置 `IPC_CREAT` 标志时，如果给出的是一个已有消息队列的键也不会产生错误，如果消息队列已有，则 `IPC_CREAT` 标志就被悄悄地忽略了。



## 消息队列（3）

表 8-7 msgsnd 函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int msgsnd(int msqid, const void *prt, size_t size, int flag)
函数传入值	msqid: 消息队列的队列 ID
	prt: 指向消息结构的指针。该消息结构 msgbuf 为: struct msgbuf{ long mtype;//消息类型 char mtext[1];//消息正文 }
	size: 消息的正文字节数
	flag: IPC_NOWAIT 若消息并没有立即发送而调用进程会立即返回 0: msgsnd 调用阻塞直到条件满足为止
函数返回值	成功: 0
	出错: -1

## 消息队列（4）

表 8-8 msgrcv 函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	iint msgrcv(int msgid,struct msgbuf *msgp,int size,long msgtype,int flag)	
函数传入值	msgid: 消息队列的队列 ID	
	msgp: 消息缓冲区	
	size: 消息的字节数，不要以 null 结尾	
	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: 接收消息队列中第一个类型值不小于 msgtyp 绝对值且类型 值又最小的消息
	flag:	MSG_NOERROR: 若返回的消息比 size 字节多，则消息就会截短到 size 字节，且不通知消息发送进程
IPC_NOWAIT 若消息并没有立即发送而调用进程会立即返回		
0: msgsnd 调用阻塞直到条件满足为止		
函数返回值	成功: 返回放到接收缓存区中的字节数	
	出错: -1	

成功: 返回放到接收器缓存区中的字节数，消息被复制后，消息队列中



# 消息队列（5）

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msqctl (int msgqid, int cmd, struct msqid_ds *buf)	
函数传入值	msgqid: 消息队列的队列 ID	
	cmd: 命令参数	IPC_STAT: 读取消息队列的数据结构 msqid_ds, 并将其存储在 buf 指定的地址中
		IPC_SET: 设置消息队列的数据结构 msqid_ds 中的 ipc_perm 域 (IPC 操作权限描述结构) 值。这个值取自 buf 参数
		IPC_RMID: 从系统内核中删除消息队列
	buf: 描述消息队列的 msqid_ds 结构类型变量	
函数返回值	成功: 0	
	出错: -1	



# 消息队列举例

---

```
#define BUFSZ 512
#define TYPE 100
struct msgbuf{
    long mtype;
    char mtext[BUFSZ];
};
int main()
{
    int qid, len;
    key_t key;
    struct msgbuf msg;

    /*根据不同的路径和关键字表示产生标准的key*/
    if ((key = ftok(".", 'a') == -1){
        perror("ftok");
        exit(1);
    }
```

---



---

`/*创建消息队列*/`

```
    if ((qid = msgget(key, IPC_CREAT|0666)) == -1){  
        perror("msgget");  
        exit(-1);  
    }
```

```
    printf("opened queue %d\n",qid);  
    puts("Please enter the message to queue:");
```

```
    if ((fgets(&msg->msg_text, BUFSZ, stdin)) == NULL){  
        puts("no message");  
        exit(-1);  
    }
```

```
    msg.mtype = TYPE;  
    len = strlen(msg.mtext) + 1;
```



---

/\*添加消息到消息队列\*/

```
if (msgsnd(qid, &msg, len, 0) < 0){  
    perror("msgsnd");  
    exit(-1);  
}
```

/\*从消息队列读取消息\*/

```
if (msgrcv(qid, &msg, BUFSZ, 0, 0) < 0){  
    perror("msgrcv");  
    exit(-1);  
}  
printf("message is:%s\n", (&msg)->mtext);
```

/\*从系统中删除消息队列。\*/



---

```
/*从系统中删除消息队列。*/  
if (msgctl(qid, IPC_RMID, NULL) < 0){  
    perror("msgctl");  
    exit(1);  
}  
return 0;  
}
```



- 
- ▶ 练习：写两个程序msgs.c和msgr.c，实现将msgs所在进程的进程号pid告诉msgr进程，msgr进程打印出pid(通过消息队列实现)

