第7章 进程间通信

7.1 进程间通信概述
7.2 管道通信
7.3 信号通信
7.4 信号量
7.5 共享内存
7.6 消息队列

7.1 进程间通信概述

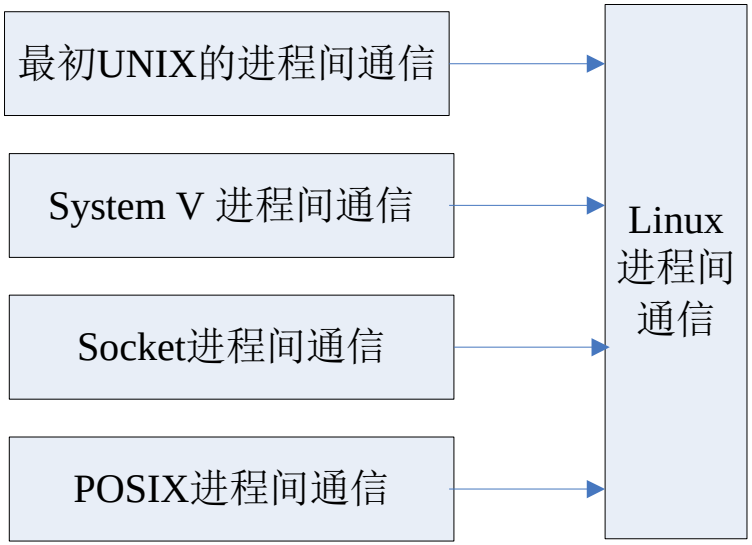
- 为什么进程间需要通信?
- 数据传输
  - 一个进程需要将它的数据发送给另一个进程
- 资源共享
  - 多个进程之间共享同样的资源
- 通知事件
  - 一个进程需要向另一个或一组进程发送消息，通知它们发生了某种事件
- 进程控制
  - 有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另一个进程的所有操作，并能够及时知道它的状态改变。

7.1.1 Linux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）在进程间的通信方面的侧重点有所不同。前者是对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”，其通信进程主要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。而Linux则把两者的优势都继承了下来。

7.1.1 Linux进程间通信

- UNIX进程间通信（IPC）方式包括管道、FIFO以及信号。
- System V进程间通信（IPC）包括System V消息队列、System V信号量以及System V共享内存区。
- Posix 进程间通信（IPC）包括Posix消息队列、Posix信号量以及Posix共享内存区。



POSIX：Portable Operating System Interface of UNIX
可移植操作系统接口。

7.1.2 Linux进程间通信方式

- 现在Linux使用的进程间通信方式主要包括:
  - 管道（pipe）和有名管道(FIFO)
  - 信号（Signal)
  - 消息队列
  - 共享内存
  - 信号量
  - 套接字（socket)

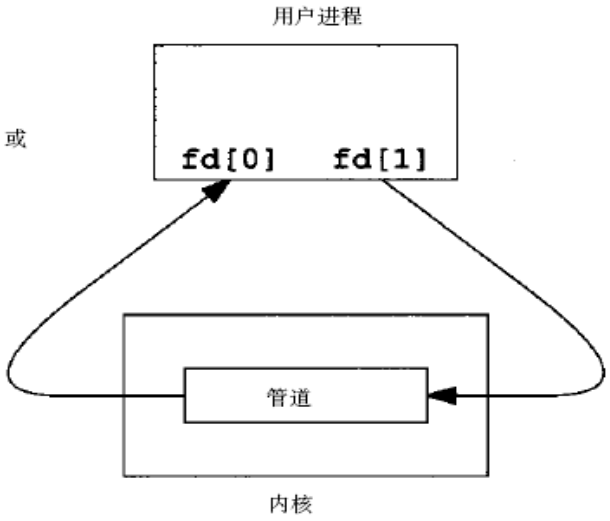
7.2 管道通信

- 管道是单向的、先进先出的，它把一个进程的输出和另一个进程的输入连接在一起。一个进程（写进程）在管道的尾部写入数据，另一个进程（读进程）从管道的头部读出数据。
- 数据被一个进程读出后，将从管道中删除。其它读进程将不能再读到这些数据。管道提供了简单的流控制机制，进程试图读空管道时，进程将阻塞。同样，管道已经满时，进程再试图向管道写入数据，进程将阻塞。
- 管道包括**无名管道和命名管道**，前者用于父进程和子进程间的通信，后者可用于运行于同一系统中任意两个进程间的通信。

7.2.1 无名管道

- 创建
  - 无名管道由pipe()函数创建：

    int pipe(int filedis[2]);
  - 当一个管道建立时，它会创建两个文件描述符：filedis[0]用于读管道，filedis[1]用于写管道。



fd[0] 用于读取管道；fd[1] 用于写入管道。

7.2.1 无名管道

- **管道关闭**
  - 关闭管道只需要将这两个文件描述符关闭即可。可以使用普通的close函数逐个关闭。

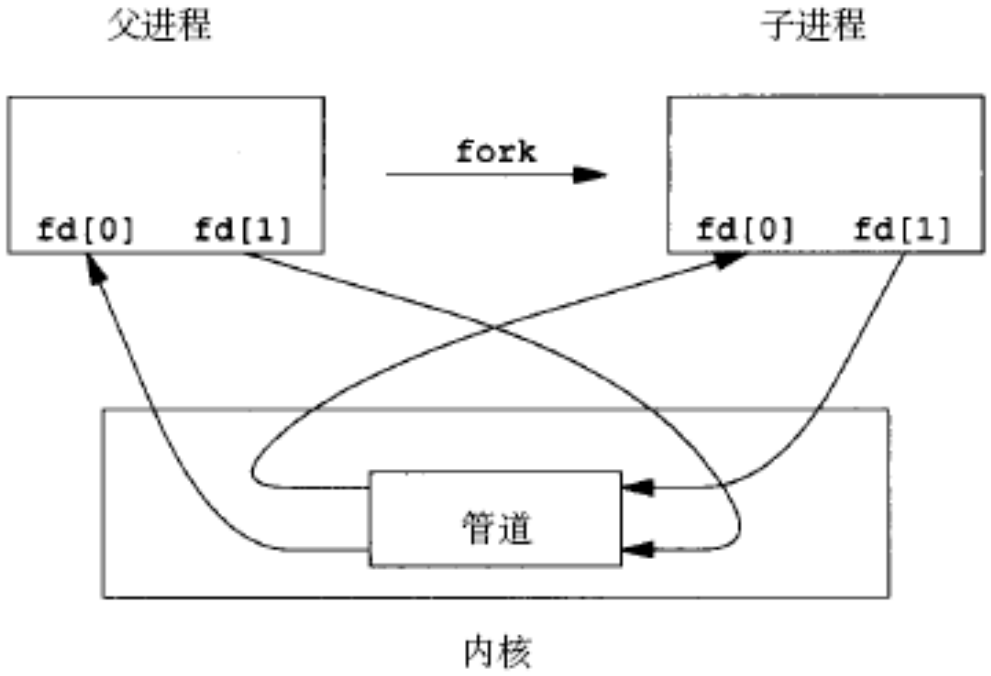
- **无名管道使用特点：**
  - 它只能用于具有亲缘关系的进程间通信(如父子进程)
  - 它是半双工的通信模式，具有固定的读端和写端
  - 管道可以看做是一种特殊的文件，对于它的读写可以使用普通的read和write等函数。但它不是普通的文件，只存在于内核的内存空间中。

无名管道实例：

```
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int pipe_fd[2];
	if(pipe(pipe_fd)<0)
	{
		printf("pipe create error\n");
		return -1;
	}
	else
		printf("pipe create success\n");
	close(pipe_fd[0]);
	close(pipe_fd[1]);
}
```

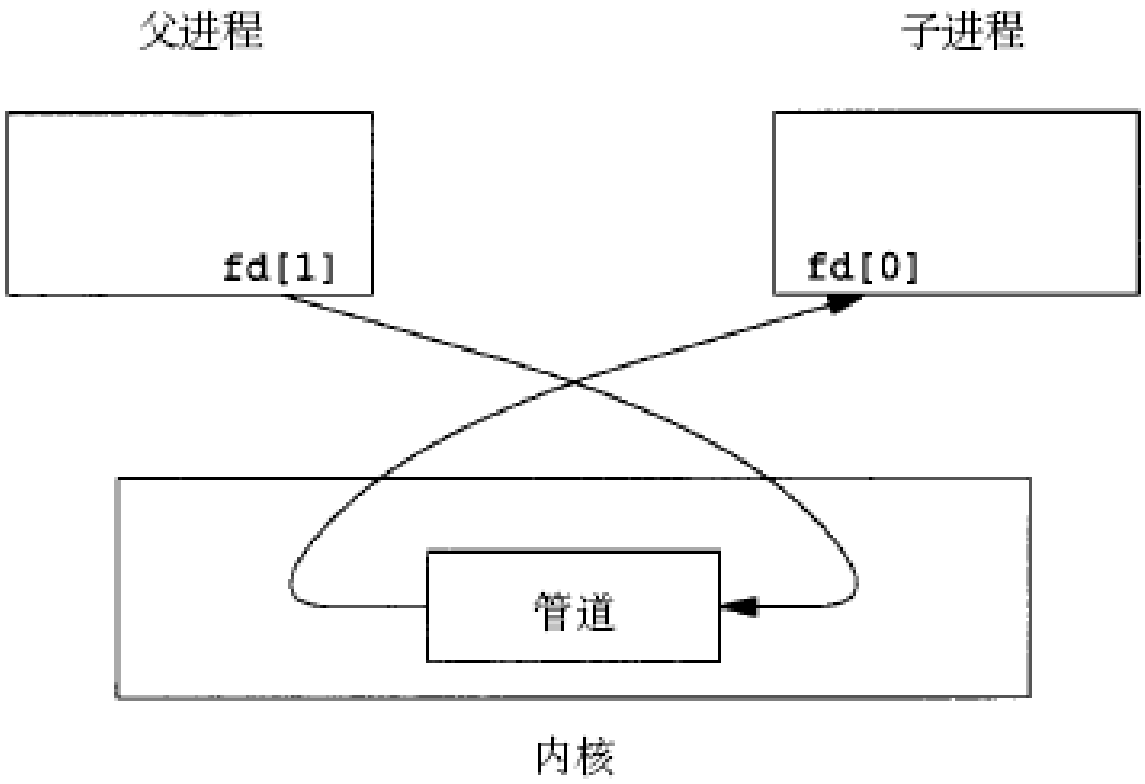
7.2.2 无名管道读写

- 无名管道用于有亲缘关系的进程间通信，通常先创建一个管道，再通过fork函数创建一个子进程，该子进程会继承父进程所创建的管道。



7.2.2 无名管道读写

- 父进程写入和子进程读的命名管道：



管道读写注意事项

- 只有在管道的读端存在时，向管道写入数据才有意义，否则，向管道写入数据的进程将收到内核传来的SIGPIPE信号。
- 向管道写入数据时，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，如果读进程不读取管道缓冲区中的数据，写操作将会阻塞。
- 父子进程在运行时，它们的先后顺序不能保证，为了保证父子进程关闭了相应的文件描述符，可以使用sleep()解决。

管道读写注意事项

- 可以通过打开两个管道来创建一个双向的管道。但需要在子进程中正确地设置文件描述符。
- 必须在系统调用fork( )前调用pipe( )，否则子进程将不会继承文件描述符。
- 当使用半双工管道时，任何关联的进程都必须共享一个相关的祖先进程。因为管道存在于系统内核之中，所以任何不在创建管道的进程的祖先进程之中的进程都将无法寻址它。而在命名管道中却不是这样。
- 程序分析：pipe_rw.c

作业：仿照半双工管道程序写一个全双工管道程序

7.2.3 命名管道FIFO

- 命名管道和无名管道基本相同，但也有不同点：无名管道只能有父子进程使用；但是通过命名管道，不相关的进程也能交换数据。
- 命名管道创建：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char* pathname,mode_t mode)
```

  - pathname:FIFO文件名
  - mode：属性（见文件操作章节）

一旦创建了一个FIFO，就可用open打开它，一般的文件访问函数（close、read、write等）都可以用于FIFO。

7.2.3 命名管道FIFO

- 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。

所需头文件	#include <sys/types.h> #include <sys/state.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	

FIFO操作

- 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。即如果FIFO内没有数据，则读函数将立刻返回0。

FIFO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，则写操作进行部分写入或者调用失败。
- 类似于管道，若写一个尚无进程为读而打开的FIFO，则产生信号SIGPIPE。若某个FIFO的最后一个写进程关闭了该FIFO，则将为该FIFO的读进程产生一个文件结束标志。
- 管道删除：命名管道的删除可以用unlink函数实现。
- 实例分析：fifo_write.c、fifo_read.c

FIFO出错信息

- FIFO相关出错信息：
  - EACCES (无存取权限)
  - EEXIST (指定文件不存在)
  - ENAMETOOLONG (路径名太长)
  - ENOENT (包含的目录不存在)
  - ENOSPC (文件系统剩余空间不足)
  - ENOTDIR (文件路径无效)
  - EROFS (指定的文件存在于只读文件系统中)

7.3 信号通信

7.3.1 信号的定义

- 信号是软件中断。它可以作为进程间通信的一种机制，更重要的是，信号总是中断一个进程的正常运行，它更多地被用于处理一些非正常情况。
  - 信号是异步的，进程并不知道信号什么时候到达。
  - 进程既可以处理信号，也可以发送信号给特定进程。
  - 每个信号都有一个名字，这些名字都以SIG开头。例如：SIGABRT是进程异常终止信号。

7.3.2 信号来源

- 很多条件可以产生一个信号：
  - 当用户按某些终端键时，产生信号。在终端上按Ctrl+c键通常产生中断信号（SIGINT）。这是停止一个已失去控制程序的方法。
  - 硬件异常产生信号：除数为0、无效的存储访问等等。这些条件通常由硬件检测到，并将其通知内核。然后内核为该条件发生时正在运行的进程产生适当的信号。例如，对执行一个无效存储访问的进程产生一个SIGSEGV。
  - 进程用kill(2)函数可将信号发送给另一个进程或进程组。自然有些限制：接收信号进程和发送信号进程的所有者必须相同，或发送信号进程的所有者必须是超级用户。

7.3.2 信号来源

- 很多条件可以产生一个信号：
  - 用户可用kill(1)命令将信号发送给其他进程。此程序是kill函数的接口。常用此命令终止一个失控的后台进程。
  - 当检测到某种软件条件已经发生，并将其通知有关进程时也产生信号。这里并不是指硬件产生条件（如被0除），而是软件条件。例如SIGURG (在网络连接上传来非规定波特率的数据)、SIGPIPE (在管道的读进程已终止后一个进程写此管道)，以及SIGALRM(进程所设置的闹钟时间已经超时)。

7.3.3 信号的种类

- 不可靠的信号：Linux信号机制基本上是从Unix系统中继承过来的。早期Unix系统中的信号机制比较简单和原始，后来在实践中暴露出一些问题，因此，把那些建立在早期机制上的信号叫做“不可靠信号”，信号值小于SIGRTMIN的叫不可靠信号(1~31)。
  - 每次信号处理后，该信号对应的处理函数会恢复到默认值。但现代的Linux已经对其进行了改进，信号处理函数一直是用户指定的或者是系统默认的。
  - 信号可能丢失。
  - 不可靠信号不支持信号排队，同一个信号产生多次，只要程序还未处理该信号，那么实际只处理此信号一次。

7.3.3 信号的种类

- 可靠信号：信号值位于SIGRTMIN和SIGRTMAX之间的信号都是可靠信号，可靠信号克服了信号可能丢失的问题 。
- 实时信号与非实时信号：Linux目前定义了64种信号（将来可能会扩展），前面32种为非实时信号，后32种为实时信号。非实时信号都不支持排队，都是不可靠信号，实时信号都支持排队，都是可靠信号。
  - 信号排队意味着无论产生多少次信号，信号处理函数就会被调用同样的次数。

7.3.3 信号的种类

信号名称	信号说明	默认处理
SIGABRT	由程序调用 abort时产生该信号。程序异常结束。	进程终止并且产生core文件
SIGALRM	timer到期，有alarm或者setitimer	进程终止
SIGBUS	总线错误，地址没对齐等。取决于具体硬件。	结束终止并产生core文件
SIGCHLD	子进程停止或者终止时，父进程会收到该信号。	忽略该信号
SIGCONT	让停止的进程继续执行	继续执行或者忽略
SIGFPE	算术运算异常，除0等。	进程终止并且产生core文件
SIGHUP	终端关闭时产生这个信号	进程终止
SIGILL	代码中有非法指令	进程终止并产生core文件
SIGINT	终端输入了中断字符ctrl+c	进程终止

7.3.3 信号的种类

SIGIO	异步I/O,跟SIGPOLL一样。	进程终止
SIGIOT	执行I/O时产生硬件错误	进程终止并且产生core文件
SIGKILL	这个信号用户不能去捕捉它。	进程终止
SIGPIPE	往管道写时，读者已经不在了	进程终止
SIGPOLL	异步I/O，跟SIGIO一样。	进程终止
SIGPROF	由setitimer设置的timer到期引发	进程终止
SIGPWR	Ups电源切换时	进程终止
SIGQUIT	Ctrl+\，不同于SIGINT，这个是会产生core dump文件的。	进程终止并且产生core文件
SIGSEGV	内存非法访问，segment fault	进程终止并且产生core文件
SIGSTOP	某个进程停止执行，该信号不能被用户捕捉。	进程暂停执行

7.3.3 信号的种类

SIGSYS	调用操作系统不认识的系统调用	进程终止并且产生core文件
SIGTERM	由kill函数调用产生	进程终止
SIGTRAP	调试器使用，gdb	进程终止并且产生core文件
SIGTSTP	Ctrl+z，挂起进程	进程暂停
SIGTTIN	后台程序要从终端读取成数据时	进程暂停
SIGTTOU	后台终端要把数据写到终端时	进程暂停
SIGURG	一些紧急的事件，比如从网络收到带外数据	忽略
SIGUSR1	用户自定义信号	进程终止
SIGUSR2	用户自定义信号	进程终止
SIGVTALRM	有setitimer产生	进程终止

7.3.4 信号的处理

- 可以要求系统在某个信号出现时按照下列三种方式中的一种进行操作。
  - (1) 忽略此信号。大多数信号都可使用这种方式进行处理，但有两种信号却决不能被忽略。它们是：SIGKILL和SIGSTOP。这两种信号不能被忽略的原因是：它们向超级用户提供一种使进程终止或停止的可靠方法。另外，如果忽略某些由硬件异常产生的信号（例如非法存储访问或除以0），则进程的行为是未定义的。
  - (2) 捕捉信号。为了做到这一点要通知内核在某种信号发生时，调用一个用户函数。在用户函数中，可执行用户希望对这种事件进行的处理。如果捕捉到SIGCHLD信号，则表示子进程已经终止，所以此信号的捕捉函数可以调用waitpid以取得该子进程的进程ID以及它的终止状态。
  - (3) 执行系统默认动作。对大多数信号的系统默认动作是终止该进程。

7.3.4.1 信号的处理—缺省

- 每一个信号都有一个缺省动作，它是当进程没有给这个信号指定处理程序时，内核对信号的处理。有5种缺省的动作：
  - 异常终止（abort）：在进程的当前目录下，把进程的地址空间内容、寄存器内容保存到一个叫做core的文件中，而后终止进程。
  - 退出（exit）：不产生core文件，直接终止进程。
  - 忽略（ignore）：忽略该信号。
  - 停止（stop）：挂起该进程。
  - 继续（continue）：如果进程被挂起，则恢复进程的运行。否则，忽略信号。

7.3.4.2 信号的处理—捕捉

- 当系统捕捉到某个信号时，可以忽略该信号或是使用指定的处理函数来处理该信号，或者使用系统默认的方式。
- 信号处理的主要方法有两种，一种是使用简单的signal函数，另一种是使用信号集函数组。

（1）Signal处理机制

#include <signal.h>

void (*signal (int signo, void (*func)(int)))(int)

返回：成功则返回信号以前的处理配置，若出错则为SIG_ERR

typedef void (*sighandler_t)(int)

sighandler_t signal(int signum,sighandler_t handler)

func的值是：

- (a)常数SIG_IGN：向内核表示忽略此信号（有两个信号SIGKILL和SIGSTOP不能忽略）
- (b)常数SIG_DFL：接到此信号后的动作是系统默认动作。
- (c)当接到此信号后要调用的函数的地址：我们称此为捕捉此信号。我们称此函数为信号处理程序（signal handler）或信号捕捉函数（signal-catching function）。

实例分析

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
void my_func(int sign_no)
{
    if(sign_no==SIGINT)
        printf("I have get SIGINT\n");
    else if(sign_no==SIGQUIT)
        printf("I have get SIGQUIT\n");
}
int main()
{
    printf("Waiting for signal SIGINT or SIGQUIT \n");
    signal(SIGINT,my_func);
    signal(SIGQUIT,my_func);
    pause();
    exit(0);
}
```

（1）Signal处理机制

- 忽略掉终端CTRL+C产生的信号：

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
int main()
{
    signal(SIGINT,SIG_IGN);
    while(1)
    sleep(1);
    return 0;
}
//程序运行后，将Ctrl+c产生的SIGINT信号忽略掉了，则CTRL+c将不能终止该进程
```

（1）Signal处理机制

- 接受信号的默认处理，接受默认处理相当于没有写信号处理程序

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
int main()
{
    signal(SIGINT,SIG_DFL);
    while(1)
            sleep(1);
    return 0;
}
```

（1）Signal处理机制

- **注意：**
  - Signal主要用于前32种非实时信号的处理
  - Signal不能传递附加数据
  - 采用signal处理机制，一些特殊情况的考虑
    - 注册一个信号处理函数，处理完毕之后，是否需要重新注册，才能捕捉下一个信号
    - 如果信号处理函数正在处理，并且没有处理完毕时，又发生了一个同类型的信号，这时该怎么处理
    - 如果信号处理函数正在处理，并且没有处理完毕时，又发生了一个不同类型的信号，这时该怎么处理
    - ……

(2) sigaction

- **Linux支持一个更健壮、更新的信号处理函数 sigaction**

  #include <signal.h>

  int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

  - 第一个signum指定需要处理(捕捉)的信号（除SIGKILL和SIGSTOP）。
  - 第二个参数act(结构体)设定信号的处理方式，act可以为NULL。之前设定的信号处理方式会保存到第三个参数oldact，oldact为NULL时不保存。
  - 返回0成功，返回-1失败。

(2) sigaction

```
struct sigaction {
        union{
            void (*sa_handler)(int);
            void (*sa_sigaction)(int, siginfo_t *, void *)}_u;
            sigset_t  sa_mask;
            int         sa_flags;
            void (*sa_restorer)(void);//保留，不使用
         }
```

- sa_handler和signal函数的第二个参数类型一样，当信号递送给进程时会调用这个sa_handler.
- sa_sigaction也是信号处理的函数指针，它只会在sa_flags包含SA_SIGINFO时才会被调用, siginfo_t 包含了信号产生的原因。sa_flags的值为0 或其它任何值都将按默认方式处理，即调用sa_handler捕捉函数。
- 不用同时赋值给sa_handler和sa_sigaction，因为它们是一个union。

(2) sigaction

- sa_mask信号屏蔽字，当执行sa_handler信号处理函数时，sa_mask指定的信号会被阻塞，直到该信号处理函数返回。
- 针对sigset_t结构体，有一组专门的函数对它进行处理：
  - int sigemptyset(sigset_t *set) //清空信号结合set
  - int sigfillset(sigset_t *set) //将所有信号填充进set中
  - int sigaddset(sigset_t *set,int signum) //添加信号
  - int sigdelset(sigset_t *set,int signum) //删除信号
  - int sigismember (const sigset_t *set,int signum) //判断某信号是否在set中

举例：在处理SIGINT时，阻塞SIGQUIT信号

```
struct sigaction act;
sigemptyset(&act.sa_mask);
sigaddset(&act.sa_mask,SIGQUIT) ;
sigaction(SIGINT,&act,NULL);
```

(2) sigaction

- sa_flags用来改变信号处理时的行为。当sa_flags包含SA_RESTART时，被中断的系统调用在信号处理完后会被自动启动。

sa_flags	说明
SA_NOCLDSTOP	若**signum**是**SIGCHLD,**当一子进程停止时，不通知父进程
SA_NOMASK/SA_NODEFER	**在处理此信号结束前允许此信号再次递送，相当于中断嵌套**
SA_RESTART	由此信号中断的系统调用自动重启
SA_NOCLDWAIT	若**signum**是**SIGCHLD,**则当调用进程的子进程终止时，不创建僵尸进程。若调用进程在后面调用**wait**，则阻塞到它所有子进程都终止，此时返回**-1**
SA_NODEFER	当捕捉到此信号时，在执行其信号捕捉函数时，系统不自动阻塞此信号。
SA_ONESHOT/SA_RESETHAND	当调用新的信号处理函数前，将此信号处理方式改为系统预设（**SIG_DFL**）的方式
SA_SIGINFO	此选项对信号处理程序提供了附加信息。

举例说明

```
int main()
{
    struct sigaction action;
    printf("Waiting for signal SIGINT or SIGQUIT...\n");

    action.sa_handler = my_func;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;

    sigaction(SIGINT, &action, 0);
    sigaction(SIGQUIT, &action, 0);
    pause();
    exit(0);
}
```

Sigprocmask信号阻塞

- 函数segaction中设置的被阻塞信号集合只是针对于要处理的信号，比如在处理SIGINT时才阻塞SIGQUIT信号；
- 函数sigprocmask是全程阻塞，在sigprocmask中设置了阻塞集合后，被阻塞的信号将不能再被信号处理函数捕捉，直到重新设置阻塞信号集合。

  int sigprocmask(int how,sigset_t *set,sigset_t *oldset)
  - how：(1):SIG_BLOCK,将参数2的信号集添加到进程原有的阻塞信号集中，(2):SIG_UNBLOCK,从进程原有的阻塞信号集中移除参数2中包含的信号，(3):SIG_SET，重新设置进程阻塞信号集为参数2的信号集
  - set是阻塞信号集，olsset存放原有的信号集

7.3.5 信号发送

7.3.5.1 kill和raise

- Linux中发送信号的主要函数有kill和raise

区别：kill()不仅可以中止进程，也可以向进程发送其他信号。与kill函数不同的是，raise()函数运行向进程自身发送信号。

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signo) ;
int raise(int signo) ;
```

两个函数返回：若成功则为0，若出错则为-1。

7.3.5.1 kill和raise

- kill的pid参数有四种不同的情况：
  - pid>0 将信号发送给进程ID为pid的进程。
  - pid = 0 将信号发送给其进程组ID等于发送进程的进程组ID，而且发送进程有许可权向其发送信号的所有进程。
  - pid < 0 将信号发送给其进程组ID等于pid绝对值，而且发送进程有许可权向其发送信号的所有进程。如上所述一样，“所有进程”并不包括系统进程集中的进程。
  - pid = -1 将信号发送给所有进程

举例：kill_raise.c

7.3.5.2 sigqueue信号发送函数

- sigqueue也可以发送信号，并能传递附加的信息。

```
#include <signal.h>
int sigqueue(pid_t pid,int sig,const union sigval value)
```

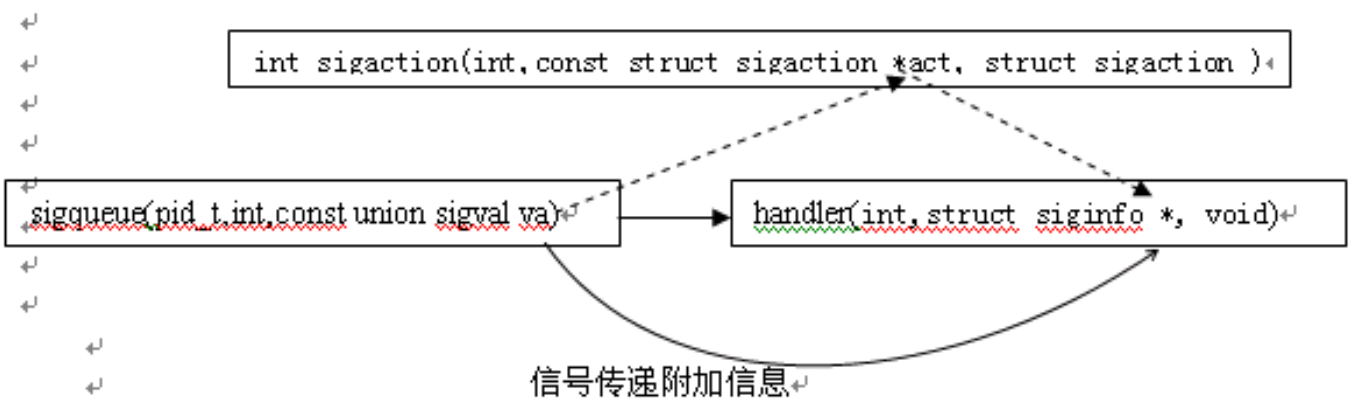
  - pid：接收信号的进程
  - sig：发送的信号
  - value：是一个整型与指针类型的联合体：

```
union sigval {
        int sival_int;
        void *sival_ptr;
}//4字节值
```

- 由sigqueue函数发送的信号的第3个参数value，可以被进程信号处理函数的第2个参数info->si_ptr接收到

sigqueue信号发送函数

信号参数的传递过程可图示如下：




```
typedef struct {
int si_signo;
int si_errno;
int si_code;
union sigval si_value;
} siginfo_t;
```

```
union sigval {
        int sival_int;
        void *sival_ptr;
}
```

sigqueue信号发送函数

- 举例：
  - 进程给自己发送信号，并带上附加数据sigqueue.c
  - 一个进程向另一个进程发送信号：sigqueue_send.c，sigqueue_rec.c

7.3.5.3 alarm和pause函数

- alarm
- 使用alarm函数可以设置一个时间值(闹钟时间)，在将来的某个时刻该时间值会被超过。当所设置的时间值被超过后，产生SIGALRM信号。如果不忽略或不捕捉此信号，则其默认动作是终止该进程。

  #include <unistd.h>

  unsigned int alarm(unsigned int seconds) ;

  返回：0或以前设置的闹钟时间的余留秒数

alarm

- 参数seconds的值是秒数，经过了指定的seconds秒后会产生信号SIGALRM。
- 每个进程只能有一个闹钟时间。如果在调用alarm时，以前已为该进程设置过闹钟时间，而且它还没有超时，则该闹钟时间的余留值作为本次alarm函数调用的值返回。以前登记的闹钟时间则被新值替换。
- 如果有以前登记的尚未超过的闹钟时间，而且seconds值是0，则取消以前的闹钟时间，其余留值仍作为函数的返回值。

pause

- pause函数使调用进程挂起直至捕捉到一个信号。

  #include <unistd.h>

  int pause(void);

  返回：-1，errno设置为EINTR

  只有执行了一个信号处理程序并从其返回时，pause才返回。

  举例：模拟sleep(5)

  作业：实现一个定时任务，每隔10S钟打印一下当前时间的秒值。

信号通信总结

- linux下的信号应用并没有想象的那么恐怖，程序员所要做的最多只有三件事情：
  - 安装信号：使用signal或sigaction（推荐使用）
  - 实现三参数信号处理函数，handler(int signal,struct siginfo *info, void *);
  - 发送信号，推荐使用sigqueue()。
- 实际上，对有些信号来说，只要安装信号就足够了（信号处理方式采用缺省或忽略）。其他可能要做的无非是与信号集相关的几种操作。

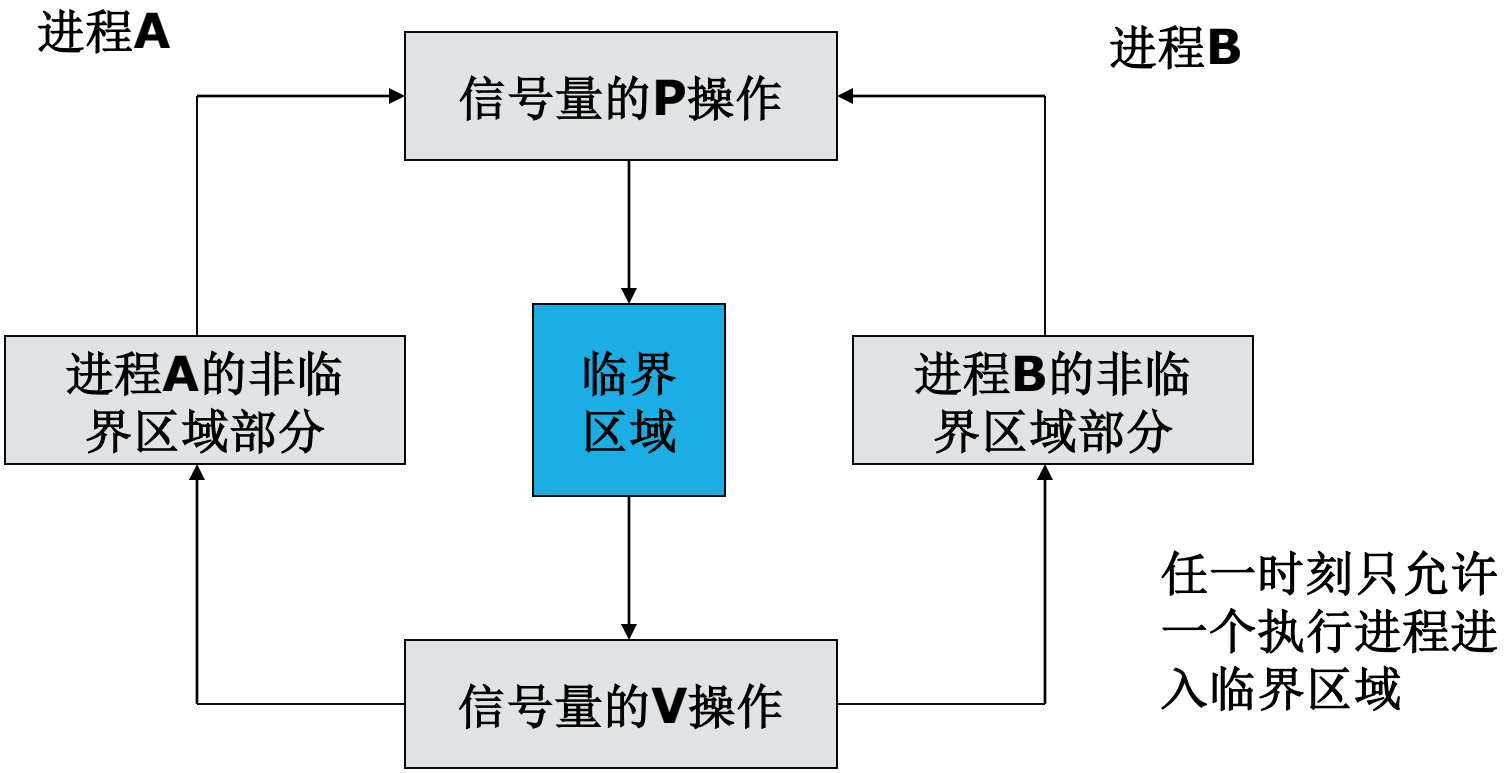
7.4 信号量通信

- 在多任务操作系统环境下，多个进程会同时运行，并且一些进程之间可能存在一定的关联。多个进程可能为了完成同一个任务会相互协作，这样形成进程之间的同步关系。而且在不同进程之间，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。
- 进程之间的互斥与同步关系存在的根源在于临界资源。临界资源是在同一个时刻只允许有限个（通常只有一个）进程可以访问（读）或修改（写）的资源，通常包括硬件资源（处理器、内存、存储器以及其他外围设备等）和软件资源（共享代码段，共享结构和变量等）。访问临界资源的代码叫做临界区，临界区本身也会成为临界资源。

7.4.1 信号量概述

- 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，包括一个称为信号量的变量和在该信号量下等待资源的进程等待队列，以及对信号量进行的两个原子操作（PV操作）。其中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）。

信号量概述



PV操作如何保护临界区

7.4.2 信号量的使用

使用信号量访问临界区的伪代码所下所示：

```
{
    /* 设R为某种资源，S为资源R的信号量*/
    INIT_VAL(S);  /* 对信号量S进行初始化 */
    非临界区代码;
    P(S);                /* 进行P操作 */
    临界区（使用资源R）；/* 只有有限个（通常只有一
                            个）进程被允许进入该区*/
    V(S);                /* 进行V操作 */
    非临界区代码;
}
```

7.4.2 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。当使用二进制信号量时，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semclt()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。

(1)信号量的使用-创建/获取

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)
函数传入值	key：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 IPC_PRIVATE。它用于创建当前进程的私有信号量。
	nsems：需要创建的信号量数目，通常取值为 1。
	semflg：同 open()函数的权限位，也可以用八进制表示法，其中使用 IPC_CREAT 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 IPC_EXCL 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。
函数返回值	成功：信号量标识符，在信号量的其他函数中都会使用该值。
	出错：-1

(2)信号量的使用-控制信号量

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)
函数传入值	semid：semget()函数返回的信号量标识符。
	semnum：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）。
	cmd：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： IPC_STAT：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。 IPC_SETVAL：将信号量值设置为 arg 的 val 值。 IPC_GETVAL：返回信号量的当前值。 IPC_RMID：从系统中，删除信号量（或者信号量集）

	arg：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。
	union semun
	{
	int val;
	struct semid_ds *buf;
	unsigned short *array;
	}

函数返回值	成功：根据 cmd 值的不同而返回不同的值。 IPC_STAT、IPC_SETVAL、IPC_RMID：返回 0 IPC_GETVAL：返回信号量的当前值

(3)信号量的使用-改变信号量值

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 *sops, size_t nsops)
函数传入值	semid：semget()函数返回的信号量标识符。

	sops：指向信号量操作数组，一个数组包括以下成员：
	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 */
	}

	nsops：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）
函数返回值	成功：信号量标识符，在信号量的其他函数中都会使用该值。
	出错：-1

举例说明

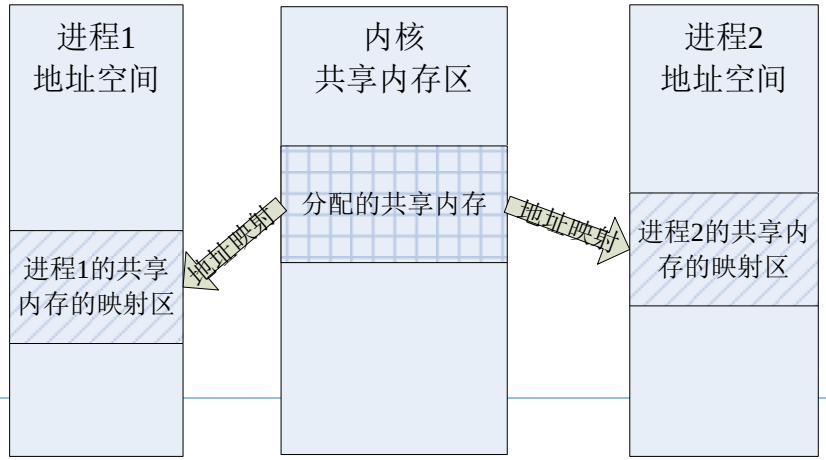
- 由于信号量的操作调用接口比较复杂，我们封装几个函数：
  - 信号量初始化函数set_semvalue()
  - P操作函数sem_p()
  - V操作函数sem_v()
  - 删除信号量函数del_semvalue()

程序举例sem1.c，进程互斥

sem_fork.c，进程同步

7.5 共享内存

- 共享内存是一种最为高效的进程间通信方式。因为进程可以直接读写内存，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。这段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。因此，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，从而大大提高了效率。当然，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 。



7.5 共享内存

- 共享内存实现分为两个步骤：

  一、创建共享内存，使用shmget函数。也就是从内存中获得一段共享内存区域。

  二、映射共享内存，将这段创建的共享内存映射到具体的进程空间去，使用shmat函数。

- 到这里，就可以使用这段共享内存了，也就是可以使用不带缓冲的I/O读写命令对其进行操作。除此之外，当然还有撤销映射的操作，其函数为shmdt()。

7.5.1 共享内存的创建

int shmget (key_t key, int size, int shmflg);

- key：标识共享内存的键值，可以使用IPC_PRIVATE
  - IPC_PRIVATE：则函数shmget()将创建一块只属于该进程私有的共享内存
- size：以字节为单位指定所需的共享内存大小
- shmflg：同open函数的权限，使用IPC_CREAT，创建共享内存
- 返回值：如果成功，返回共享内存段标识符
  如果失败，则返回-1

7.5.2 映射

- 原型：int shmat (int shmid, char *shmaddr, int shmflg);
  - shmid：shmget函数返回的共享内存标识符
  - shmaddr：将共享内存映射到指定地址，(若为0：表示系统自动分配)
  - flag：决定以什么方式来确定映射地址，通常为0，表示共享内存可读写
  - 返回值：如果成功，则返回共享内存段连接到进程中的地址。如果失败，则返回- 1：
    - **errno = EINVAL** (无效的IPC ID 值或者无效的地址)
    - errno = ENOMEM (没有足够的内存)
    - errno = EACCES (存取权限不够)

7.5.3 解除映射

- 当一个进程不在需要共享的内存段时，它将会把内存段从其地址空间中脱离。

系统调用：shmdt();

调用原型：int shmdt (char *shmaddr);

返回值：如果失败，则返回- 1：

errno = EINVAL (无效的连接地址)

- 脱离不等于删除，删除共享内存需要调用：

shmctl(int shm_id,int command,struct shmid_ds *buf)

Command：IPC_RMID

实例分析：shm1.c

7.6 消息队列

- 消息队列就是一些消息的列表。用户可以从消息队列中添加消息和读取消息等。从这点上看，消息队列具有一定的FIFO特性，但是它可以实现消息的随机查询，比FIFO具有更大的优势。同时，这些消息又是存在于内核中的，由“队列ID”来标识。
- 消息队列的实现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这四种操作。其中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()函数，它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()，它把消息从消息队列中取走，与FIFO不同的是，这里可以指定取走某一条消息；最后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

- 特点：
  - 持续性：系统V消息队列是随内核持续的，只有在内核重启或者人工删除时，该消息队列才会被删除
  - 键值：消息队列的内核持续性要求每个消息队列都在系统范围内对应唯一的键值，所以要获得一个消息队列的描述字，必须提供该消息队列的键值。

    key_t ftok(char *pathname,char proj)

    返回文件名对应的键值

    pathname:文件名，proj:项目名，不为0即可。
  - 消息队列相比命名管道的优势在于它独立于发送和接收进程而存在，消除了同步命名管道的打开和关闭产生的问题。

7.6.1 消息队列-打开/创建

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int msgget(key_t key, int msgflg)
函数传入值	key：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 IPC_PRIVATE。它用于创建当前进程的私有消息队列。
	msgflg：权限标志位
函数返回值	成功：消息队列 ID
	出错：-1

7.6.2 消息队列-发送消息

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 *msgp, size_t msgsz, int msgflg)

函数传入值	msqid：消息队列
	msgp：指向消息
	struct msgbuf
	{
	long mtype; /* 消息类型，该结构必须从这个域开始 */
	char mtext[1]; /* 消息正文 */
	}
	msgsz：消息正文的字节数（不包括消息类型指针变量）
	msgflg：IPC_NOWAIT 若消息无法立即发送（比如：当前消息队列已满），函数会立即返回。
	0：msgsnd 调用阻塞直到发送成功为止。

函数返回值	成功：0 出错：-1

每个数据块都被认为含有一个类型，接收进程可以独立的接收含有不同类型值的数据块,该值必须大于0.

7.6.3 消息队列-接收消息

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgrcv(int msgid, void *msgp, size_t msgsz, long int msgtyp, int msgflg)	
函数传入值	msqid：消息队列的队列 ID	
	msgp：消息缓冲区，同于 msgsnd()函数的 msgp	
	msgsz：消息正文的字节数（不包括消息类型指针变量）	
	msgtyp：	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
	msgflg：	MSG_NOERROR：若返回的消息比 msgsz 字节多，则消息就会截短到 msgsz 字节，且不通知消息发送进程
		IPC_NOWAIT 若在消息队列中并没有相应类型的消息可以接收，则函数立即返回
		0：msgsnd()调用阻塞直到接收一条相应类型的消息为止
函数返回值	成功：0	
	出错：−1	

7.6.4 消息队列-控制

```
struct msqid_ds{
    uid_t msg_perm.uid;
    uid_t msg_perm.gid;
    mode_t msg_perm.mo
};
```

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgctl (int msgqid, int cmd, struct msqid_ds *buf)	
函数传入值	msqid：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 msgqid_ds 结构类型变量	
函数返回值	成功：0	
	出错：−1	

消息对列举例

- 程序分析msg1.c msg2.c