中国矿业大学计算机学院

2019-2020(2)本科生 Linux 操作系统课程作业

内容范围 进程通信

指 标 点 1.2 占 比 50%

学生姓名 袁孝健 学 号 06172151

专业班级 信息安全 2017-01 班

任课教师 杨东平

课程基础理论掌握程度	熟练 □	较熟练 □	一般 □	不熟练 □
综合知识应用能力	强 □	较强 □	一般 □	差 □
作业内容	完整 □	较完整 □	一般 □	不完整 □
作业格式	规范 □	较规范 □	一般 □	不规范 □
作业完成状况	好 □	较好 □	一般 □	差 □
工作量	饱满 □	适中 □	一般 □	欠缺 □
学习、工作态度	好 □	较好 □	一般 □	差 □
抄袭现象	无 □	有 □	姓名:	
存在问题				
总体评价				

综合成绩: 任课教师签字:

年 月 日

目　　录

1 进程通信

1.1 进程的概念

进程是操作系统的概念，每当我们执行一个程序时，对于操作系统来讲就创建了一个进程,在这个过程中，伴随着资源的分配和释放。可以认为进程是一个程序的一次执行过程。

1.2 进程通信的概念

进程用户空间是相互独立的，一般而言是不能相互访问的。但很多情况下进程间需要互相通信，来完成系统的某项功能。进程通过与内核及其它进程之间的互相通信来协调它们的行为。通信的概念

1.3 进程通信的应用场景

（1）数据传输：一个进程需要将它的数据发送给另一个进程，发送的数据量在一个字节到几兆字节之间。

（2）共享数据：多个进程想要操作共享数据，一个进程对共享数据的修改，别的进程应该立刻看到。

（3）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（它们）发生了某种事件（如进程终止时要通知父进程）。

（4）资源共享：多个进程之间共享同样的资源。为了作到这一点，需要内核提供锁和同步机制。

（5）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陷入和异常，并能够及时知道它的状态改变。

1.4 进程通信的方式

（1）管道: 无名管道(pipe)、命名管道(named_pipe)、流管道(s_pipe)

（2）信号(signal)

（3）消息队列(message queue)

（4）共享内存(shared memory)

（5）信号量(semophore)

（6）套接字（socket)

2 管道

2.1 无名管道

2.1.1 概述

无名管道是一种半双工的通信方式,数据只能单向流动,而且只能在具有亲缘关系的进程间使用。进程的亲缘关系一般指的是父子关系，无名管道一般用于两个不同进程之间的通信。当一个进程创建了一个管道,并调用 fork 创建自己的一个子进程后,父进程关闭

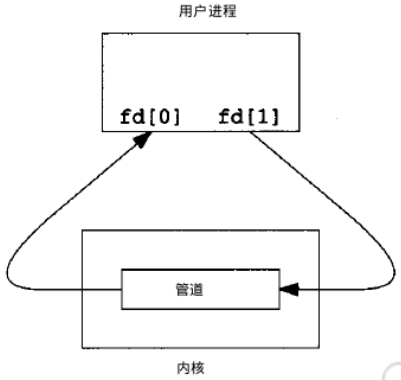
读管道端,子进程关闭写管道端,这样提供了两个进程之间数据流动的一种方式。

2.1.2 原型

（1）函数原型

```
int pipe(int fd[2]);
```

当一个管道建立时，它会创建两个文件描述符：fd[0]为读而打开，fd[1]为写而打开，要关闭管道只需将这两个文件描述符关闭即可，如下：



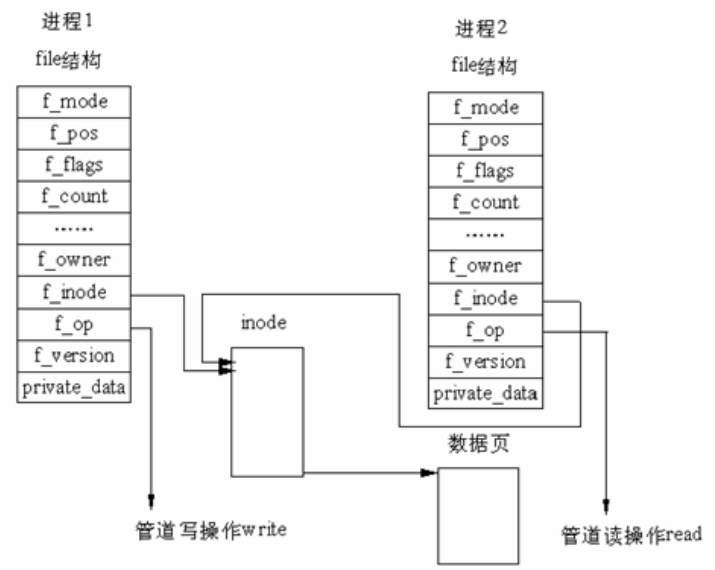
（2）头文件

```
include <unistd.h>
```

（3）返回值：若成功返回 0，失败返回-1

2.1.3 实现原理

在 Linux 中，管道的实现并没有使用专门的数据结构，而是借助了文件系统的 file 结构和 VFS 的索引节点 inode。通过将两个 file 结构指向同一个临时的 VFS 索引节点，而这个 VFS 索引节点又指向一个物理页面而实现的。如下图：



有两个 file 数据结构，但它们定义文件操作进程地址是不同的，其中一个是向管道中写入数据的进程地址，而另一个是从管道中读出数据的进程地址。这样，用户程序的系统

调用仍然是通常的文件操作，而内核却利用这种抽象机制实现了管道这一特殊操作。

2.2 命名管道

2.2.1 概述

由于基于 fork 机制，所以管道只能用于父进程和子进程之间，或者拥有相同祖先的两个子进程之间（有亲缘关系的进程之间）。为了解决这一问题，Linux 提供了 FIFO 方式连接进程。FIFO 又叫做命名管道(named PIPE)。

2.2.2 原型

（1）函数原型

```
int mkfifo(const char *filename, mode_t mode);
int mknode(const char *filename, mode_t mode | S_IFIFO, (dev_t) 0 );
```

其中 filename 是被创建的文件名称，mode 表示将在该文件上设置的权限位和将被创建的文件类型(在此情况下为 S_IFIFO)，dev 是当创建设备特殊文件时使用的一个值。因此，对于先进先出文件它的值为 0。

（2）头文件

```
#include <sys/types.h>
#include <sys/stat.h>
```

2.2.3 实现原理

FIFO (First in, First out)为一种特殊的文件类型，它在文件系统中有对应的路径。当一个进程以读(r)的方式打开该文件，而另一个进程以写(w)的方式打开该文件，那么内核就会在这两个进程之间建立管道，所以 FIFO 实际上也由内核管理，不与硬盘打交道。之所以叫 FIFO，是因为管道本质上是一个先进先出的队列数据结构，最早放入的数据被最先读出来，从而保证信息交流的顺序。FIFO 只是借用了文件系统(file system,命名管道是一种特殊类型的文件，因为 Linux 中所有事物都是文件，它在文件系统中以文件名的形式存在。)来为管道命名。写模式的进程向 FIFO 文件中写入，而读模式的进程从 FIFO 文件中读出。当删除 FIFO 文件时，管道连接也随之消失。FIFO 的好处在于我们可以通过文件的路径来识别管道，从而让没有亲缘关系的进程之间建立连接。

2.3 流管道

2.3.1 概述

流管道(s_pipe)是一种基于文件流的管道，例如用户执行 ls -l 或./pipe 这类很常见的操作，其实都是把一些列操作合并到 popen 函数中完成。

2.3.2 原型

（1）函数原型

```
FILE *popen(const char *cmd, const char *type)
```

cmd 是一个字符串，包含一个 shell 命令，会被发送到 /bin/sh 中以 -c 参数执行；

type 分为 r（使该命令的结果产生输出）和 w（使该命令的结果产生输入）。

（2）头文件

```
#include <unistd.h>
```

（3）返回值

成功返回文件流指针，出错返回-1。

3 信号量

3.1 概述

为了防止出现因多个程序同时访问一个共享资源而引发的一系列问题，我们需要一种方法。比如在任一时刻只能有一个执行线程访问代码的临界区域。临界区域是指执行数据更新的代码需要独占式地执行。而信号量就可以提供这样的一种访问机制，让一个临界区同一时间只有一个线程在访问它，也就是说信号量是用来调协进程对共享资源的访问的。

信号量是一个特殊的变量，程序对其访问都是原子操作，且只允许对它进行等待（即 P(信号变量)）和发送（即 V(信号变量)）信息操作。最简单的信号量是只能取 0 和 1 的变量，这也是信号量最常见的一种形式，叫做二进制信号量。而可以取多个正整数的信号量被称为通用信号量。

3.2 原型

3.2.1 创建信号量

（1）函数原型

```
int semget (key_t key, int nsem, int oflag);
```

（2）头文件

```
#include <sys/sem.h>
```

（3）返回值

成功时返回信号量集的 IPC 标识符(一个正整数)，失败，则返回 -1，errno 被设定成以下的某个值:

- EACCES: 没有访问该信号量集的权限
- EEXIST: 信号量集已经存在，无法创建
- EINVAL: 参数 nsems 的值小于 0 或者大于该信号量集的限制；或者是该 key 关联的信号量集已存在，并且 nsems 大于该信号量集的信号量数
- ENOENT: 信号量集不存在，同时没有使用 IPC_CREAT

- ENOMEM: 没有足够的内存创建新的信号量集
- ENOSPC: 超出系统限制

（4）参数

① key: 所创建或打开信号量集的键值，需要是唯一的非零整数

② nsem: 创建的信号量集中的信号量的个数，该参数只在创建信号量集时有效，一般为 1。若用于访问一个已存在的集合，那么就可以把该参数指定为 0

③ oflag:

- 调用函数的操作类型，有两个值: 1) IPC_CREATE：若信号量已存在，返回该信号量标识符;2)IPC_EXCL：若信号量已存在，返回错误。
- 也可用于设置信号量集的访问权限: SEM_R(read)和 SEM_A(alter)。
- 两者通过 or 表示。

3.2.2 打开信号量

（1）函数原型

```
int semop (int semid, struct sembuf * opsptr, size_t nops);
```

（2）头文件

```
#include <sys/sem.h>
```

（3）参数:

① emid: 信号量标识符

② opsptr：指向一个信号量操作数组，信号量操作由 sembuf 结构表示:

```
struct sembuf {
    short sem_num; // 除非使用一组信号量，否则它为 0
    short sem_op; /*信号量在一次操作中需要改变的数据，通常是两个数，一个是 -
    1，即 P (等     待)操作，一个是 +1，即 V (发送信号)操作*/
    short sem_flg;
    /*说明函数 semop 的行为。通常为 SEM_UNDO，使操作系统跟踪
    信号，并在进程没有释放该信号量而终止时，
    操作系统自动释放该进程持有的信号量/*
};
```

③ nops:规定 opsptr 数组中元素个数

- sem_op>0:进程释放占用的资源数，该值加到信号量的值上(V 操作)。
- sem_op<0:要获取该信号量控制的资源数，信号量值减去该值的绝对值(P 操作)。
- sem_op=0:调用进程希望等待到该信号量值变成 0。

3.2.3 信号量操作

（1）函数原型

```
int semctl (int semid, int semnum, int cmd, [union semun sem_union]);
```

（2）头文件

```
#include <sys/sem.h>
```

（3）返回值

成功，则为一个正数；失败，则为 -1，同时置 errno 为以下值之一:

- EACCESS: 权限不够。
- EFAULT: arg 指向的地址无效。
- EIDRM: 信号量集已经删除。
- EINVAL: 信号量集不存在，或者 semid 无效。
- EPERM: EUID 没有 cmd 的权利。
- ERANGE: 信号量值超出范围。

（4）参数

① semnum：指定信号集中的哪个信号(操作对象)

② cmd：指定将执行的命令:

- IPC_STAT:读取一个信号量集的数据结构 semid_ds，并将其存储在 semun 中的 buf 参数中。
- IPC_SET: 设置信号量集的数据结构 semid_ds 中的元素 ipc_perm,其值取自 semun 中的 buf 参数。
- IPC_RMID: 删除不再使用的信号量。
- GETALL: 用于读取信号量集中的所有信号量的值。
- GETNCNT: 返回正在等待资源的进程数目。
- GETPID: 返回最后一个执行 semop 操作的进程的 PID。
- GETVAL: 返回信号量集中的一个单个的信号量的值。
- GETZCNT:返回这在等待完全空闲的资源的进程数目。
- SETALL: 设置信号量集中的所有的信号量的值。
- SETVAL: 设置信号量集中的一个单独的信号量的值。

③ 第三个参数是可选项，它取决于参数 cmd，其结构如下：

```
union semun{
    int val; // 用于 SETVAL，信号量的初始值
    struct semid_ds *buf; // 用于 IPC_STAT 和 IPC_SET 的缓冲区
    unsigned short *array; // 用于 GETALL 和 SETALL 的数组
};
```

3.3 工作原理

由于信号量只能进行两种操作等待和发送信号，即 P(sv)和 V(sv),他们的行为是这样

的：

- P(sv)：如果 sv 的值大于零，就给它减 1；如果它的值为零，就挂起该进程的执行。
- V(sv)：如果有其他进程因等待 sv 而被挂起，就让它恢复运行，如果没有进程因等待 sv 而挂起，就给它加 1。

例如两个进程共享信号量 sv，一旦其中一个进程执行了 P(sv)操作，它将得到信号量，并可以进入临界区，使 sv 减 1。而第二个进程将被阻止进入临界区，因为当它试图执行 P(sv)时，sv 为 0，它会被挂起以等待第一个进程离开临界区域并执行 V(sv)释放信号量，这时第二个进程就可以恢复执行。

4 消息队列

4.1 概述

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。消息队列是随内核持续的。

每个消息队列都有一个队列头，用结构 struct msg_queue 来描述。队列头中包含了该消息队列的大量信息，包括消息队列键值、用户 ID、组 ID、消息队列中消息数目等等，甚至记录了最近对消息队列读写进程的 ID。读者可以访问这些信息，也可以设置其中的某些信息。

4.2 原型

4.2.1 消息队列中的数据结构

msqid_ds 内核数据结构: Linux 内核维护每个消息队列的结构体，它保存着消息队列当前状态信息。

```
struct msqid_ds{
    struct ipc_perm msg_perm; // 所有者和权限
    time_t msg_stime; // 最后一次调用 msgsnd 的时间
    time_t msg_rtime; // 最后一次调用 msgrcv 的时间
    time_t msg_ctime; // 队列最后一次变动的时间
    unsigned long __msg_cbytes; // 当前队列中字节数( 不标准)
    msgqnum_t msg_qnum; // 当前队列中消息数
    msglen_t msg_qbytes; // 队列中允许的最大字节数
    pid_t msg_lspid; // 最后一次调用 msgsnd 的 PID
    pid_t msg_lrpid; // 最后一次调用 msgrcv 的 PID
};
```

其中的 ipc_perm 保存消息队列的一些重要的信息，如消息队列关联的键值、消息队列的用户 id、组 id 等。

```
struct ipc_perm{
    key_t key; // 消息队列键值
    uid_t uid; // 有效的拥有者 UID
    gid_t gid; // 有效的拥有者 GID
    uid_t cuid; // 有效的创建者 UID
    gid_t cgid; // 有效的创建者 GID
    unsigned short mode; // 权限
    unsigned short seq; // 队列号
};
```

4.2.2 创建消息队列

（1）函数原型

```
int msgget(key_t key,int msgflg);
```

（2）头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

（3）返回值

成功，返回以 key 命名的消息队列的标识符(非零正整数)；失败，返回-1。

（4）参数

① key: 命名消息队列的键，一般用 ftok 函数获取。

② msgflg: 消息队列的访问权限，可以与以下键或操作: IPC_CREAT：不存在则创建，存在则返回已有的 qid。

4.2.3 发送消息

（1）函数原型

```
int semctl (int semid, int semnum, int cmd, [union semun sem_union]);
```

（2）返回值

成功，消息数据的一份副本将被放到消息队列中，并返回 0；失败，返回-1。

（3）参数

① msgid: 由 msgget 函数返回的消息队列标识符。

② msgp: 将发往消息队列的消息结构体指针。

③ msgsz: 消息长度，是消息结构体中待传递数据的大小(不是整个结构体的大小)

④ msgflg:

- IPC_NOWAIT: 消息队列满时返回-1。
- 0: 消息队列满时阻塞。

4.2.4 接收消息

（1）函数原型

```
ssize_t msgrcv(int qid,void *msgp,size_t msgsz,long msgtype,int
msgflg);
```

（2）返回值

成功,返回放到接收缓存区中的字节数，消息被复制到由 msgp 指向的用户分配的缓存区中，然后删除消息队列中的对应消息；失败，返回-1。

（3）参数

① msgid、msgp、magsz、msgflg 的作用同函数 msgsnd。

② msgtype：可以实现一种简单的接收优先级。如果 msgtyp 为 0，就获取队列中的第一个消息。如果它的值大于零，将获取具有相同消息类型的第一个信息。如果它小于零，就获取类型等于或小于 msgtype 的绝对值的第一个消息。

4.2.5 控制消息队列

（1）函数原型

```
int msgctl(int msgid,int cmd,struct msgid_ds *buf);
```

（2）参数

① msgid：由 msgget 函数返回的消息队列标识符

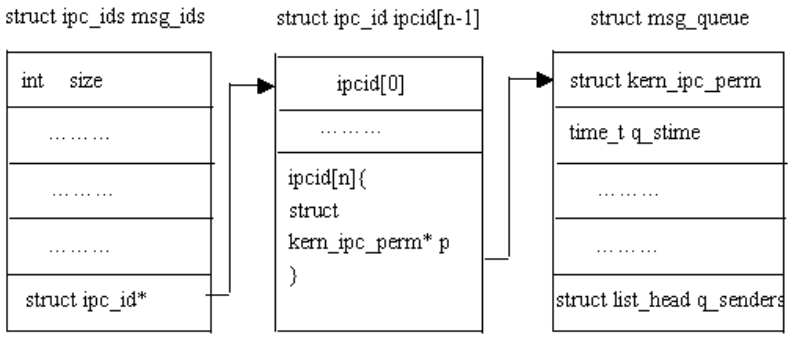
② cmd：将要采取的动作，它可以取 3 个值之一：

- IPC_STAT: 用来获取消息队列信息，并存储在 buf 指向的 msgid_ds 结构。
- IPC_SET: 用来设置消息队列的属性，要设置的属性存储在 buf 指向的 msgid_ds 结构中。
- IPC_RMID: 删除 msqid 标识的消息队列。

③ buf：指向 msgid_ds 权限结构。

4.3 实现原理

下图说明了内核与消息队列是怎样建立起联系的:



从上图可以看出，全局数据结构 struct ipc_ids msg_ids 可以访问到每个消息队列头的第一个成员：struct kern_ipc_perm；而每个 struct kern_ipc_perm 能够与具体的消息队列对应起来是因为在该结构中，有一个 key_t 类型成员 key，而 key 则唯一确定一个消息队列。

5 信号

5.1 概述

（1）什么是信号

信号用于通知接收进程某个事件已经发生。除了用于进程间通信外，还可以发送信号给进程本身。

（2）信号的产生

① 由硬件产生，如从键盘输入 Ctrl+C 可以终止进程

② 由其他进程发送，如 shell 下用命令 kill -信号标号 PID 可以向指定进程发送信号

③ 异常，进程异常时会发送信号

（3）信号的通信

信号（signal）是进程之间相互传递消息的一种方法，分为“软中断信号”和“硬件中断信号”，用来通知进程发生了异步事件。只是用来通知某进程发生了什么事件，并不传递任何数据。比如在 shell 中使用 Ctrl+ C 停止执行某个命令。每种信号用一个整型常量宏表示，以 SIG 开头，比如 SIGCHLD、SIGINT 等，它们在系统头文件中定义。

使用命令 kill -l 可以查看 Linux 所有信号信息:

```
ubuntu@VM-0-14-ubuntu:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

5.2 原型

5.2.1 信号处理

（1）函数原型

```
void (*signal(int sig, void (*func)(int))) (int);

int sigaction(int sig,const struct sigaction *act,struct sigaction
*oact);
```

（2）头文件

```
#include <signal.h>
```

（3）功能

① 用于处理指定的信号，主要通过忽略和恢复其默认行为来工作。

② 设置与信号 sig 关联的动作。

（4）参数

① sig: 信号值。

② func: 信号处理函数指针，参数为信号值。

③ act: 指定信号的动作。

④ oact: 保存原信号的动作。

5.2.2 信号阻塞

（1）函数原型

```
void (*signal(int sig, void (*func)(int))) (int);
```

（2）功能

阻塞是阻止进程收到该信号，此时信号处于未决状态，放入进程的未决信号表中，当解除对该信号的阻塞时，未决信号会被进程接收。

（3）参数

① how: 设置 block 阻塞表的方式。

- SIG_BLOCK: 将信号集添加到 block 表中。
- SIG_UNBLOCK: 将信号集从 block 表中删除。
- SIG_SETMASK: 将信号集设置为 block 表。

② set: 要设置的集合。

③ oset: 设置前保存之前 block 表信息。

5.2.3 信号发送

（1）函数原型

```
int kill(pid_t pid, int signo);
int raise(int sig);
void abort(void);
int pause(void);
unsigned int alarm(unsigned int seconds);
```

（2）功能

① kill: 发送信号给进程或进程组，可以是本身或其它进程。

② raise: 能向进程自身发信号。

③ abort: 发送 SIGABRT 信号，让进程异常终止，发生转储(core)。

④ pause：将调用进程挂起直至捕捉到信号为止，通常可以用于判断信号是否已到。

⑤ alarm：发送 SIGALRM 闹钟信号。

（3）参数

① pid：

- >0：发送给进程 ID。
- 0：发送给所有和当前进程在同一个进程组的进程。
- -1：发送给所有的进程表中的进程(进程号最大的除外)。
- <-1：发送给进程组号为 -PID 的每一个进程。

② signo：信号值。

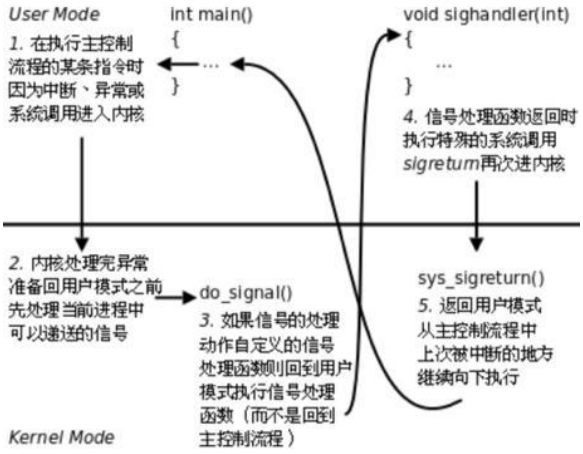
③ seconds：系统经过 seconds 秒后向进程发送 SIGALRM 信号。

5.3 实现原理

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。

信号是由操作系统来处理的，说明信号的处理在内核态信号不一定会立即被处理，此时会储存在信号的信号表中，处理过程如下：



6 共享内存

6.1 概述

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。是针对其他通信机制运行效率较低而设计的。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。

6.2 原型

6.2.1 shmget

（1）函数原型

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

（2）头文件

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

（3）返回值

成功，非负整数，即该共享内存段的标识码；失败，-1。

（4）参数

① key: 共享内存段的名字，通常用 ftok() 函数获取。

② size: 以字节为单位的共享内存大小。内核以页为单位分配内存，但最后一页的剩余部分内存不可用。

③ shmflg: 九个比特的权限标志(其作用与文件 mode 模式标志相同)，并与 IPC_CREAT 或时创建共享内存段。

（5）功能

创建/获取共享内存。

6.2.2 shmat

（1）函数原型

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

（2）返回值

成功，指向共享内存第一个节的指针；失败，-1。

（3）参数

① wshmid: shmget 返回的共享内存标识。

② shmaddr: 指定共享内存连接到当前进程中的地址位置。

③ shmflg: 它有两个可能取值， 用来控制共享内存连接的地址。

（4）功能

将共享内存段连接到进程。共享内存段刚创建时不能被任何进程访问，必须将其连接到一个进程的地址空间才能使用。

6.2.3 shmd

（1）函数原型

```
int shmdt(const void *shmaddr);
```

（2）返回值

成功返回 0；失败返回-1。

（3）参数

① shmaddr: shmat 所返回的指针。

（4）功能

将共享内存从当前进程中分离。共享内存分离并未删除它，只是使得该共享内存对当前进程不再可用。

6.2.4 shmctl

（1）函数原型

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

（2）返回值

成功返回 0，否则返回-1。

（3）参数

① shmid: 由 shmget 返回的共享内存标识码。

② cmd: 将要采取的动作，有三个可取值。

- IPC_STAT: 把 shmid_ds 结构中的数据设置为共享内存的当前关联值。
- IPC_SET: 如果有足够的权限，把共享内存的当前值设置为 shmid_ds 数据结构中给出的值。
- IPC_RMID: 删除共享内存段。

③ buf: 用于保存共享内存模式状态和访问权限，至少包含以下成员:

```
struct shmid_ds {
    uid_t shm_perm.uid;
    uid_t shm_perm.gid;
    mode_t shm_perm.mode;
}
```

（4）功能

共享内存控制。

6.3 实现原理

进程间需要共享的数据被放在一个叫做 IPC 共享内存区域的地方，所有需要访问该共享区域的进程都要把该共享区域映射到本进程的地址空间中去。系统 V 共享内存通过 shmget 获得或创建一个 IPC 共享内存区域，并返回相应的标识符。内核在保证 shmget 获得或创建一个共享内存区，初始化该共享内存区相应的 shmid_kernel 结构体的同时，还将在特殊文件系统 shm 中，创建并打开一个同名文件，并在内存中建立起该文件的相应 dentry 及 inode 结构，新打开的文件不属于任何一个进程（任何进程都可以访问该共享

内存区)。所有这一切都是系统调用 shmget 完成的。

shmget()用来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。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。shmdt()调用用来解除进程对共享内存区域的映射。shmctl 实现对共享内存区域的控制操作。

7 套接字

7.1 概述

两个基本概念：客户方和服务方。当两个应用之间需要采用 SOCKET 通信时，首先需要在两个应用之间（可能位于同一台机器，也可能位于不同的机器）建立 SOCKET 连接。

（1）发起呼叫连接请求的一方为客户方

在客户方呼叫连接请求之前，它必须知道服务方在哪里。所以需要知道服务方所在机器的 IP 地址或机器名称，如果客户方和服务方事前有一个约定就好了，这个约定就是 PORT（端口号)。也就是说，客户方可以通过服务方所在机器的 IP 地址或机器名称和端口号唯一的确定方式来呼叫服务方。

（2）接受呼叫连接请求的一方成为服务方。

在客户方呼叫之前，服务方必须处于侦听状态，侦听是否有客户要求建立连接。一旦接到连接请求，服务方可以根据情况建立或拒绝连接。当客户方的消息到达服务方端口时，会自动触发一个事件（event)，服务方只要接管该事件，就可以接受来自客户方的消息了。

7.2 原型

7.2.1 socket

（1）函数原型

```
int socket(int domain, int type, int protocol);
```

（2）头文件

```
#include <sys/types.h>
#include <sys/socket.h>
```

（3）返回值

成功返回 Socket 描述符；失败返回-1，可用 errno 查看出错的详细情况

（4）参数

① domain: 程序采用的通讯协族。

② type: 采用的通讯协议, SOCK_STREAM 表示 TCP，SOCK_DGRAM 表示 UDP。

③ protocol: 由于指定了 type，此值一般为 0。

（5）功能

用于创建一个 socket。

7.2.2 bind

（1）函数原型

```
int bind(int sockfd, struct sockaddr* servaddr, int addrlen);
```

（2）返回值

成功返回 0，失败返回-1，相应地设定全局变量 errno，最常见的错误是该端口已经被其他程序绑定。

（3）参数

① sockfd：由 socket 调用返回的文件描述符。

② servaddr：出于兼容性，一般使用 sockaddr_in 结构。

③ addrlen：servaddr 结构的长度。

（4）功能

用于 Server 程序，绑定被侦听的端口。

7.2.3 listen

（1）函数原型

```
int listen(int sockfd, int backlog);
```

（2）返回值

成功返回 0，失败返回-1.

（3）参数

① sockfd：被 bind 的文件描述符(socket()建立的)。

② backlog：设置 Server 端请求队列的最大长度。

（4）功能

用于 Server 程序，侦听 bind 绑定的套接字。

7.2.4 accept

（1）函数原型

```
int accept(int sockfd, struct sockaddr* addr, int *addrlen);
```

（2）返回值

成功，则返回 Server 用于与 Client 进行数据传输的文件描述符；失败，则返回-1，相应地设定全局变量 errno。

（3）参数

① sockfd：listen 后的文件描述符(socket()建立的)。

② addr：返回 Client 的 IP、端口等信息，确切格式由套接字的地址类别(如 TCP 或 UDP)决定；若 addr 为 NULL，则 addrlen 应置为 NULL。

③ addrlen：返回真实的 addr 所指向结构的大小，只要传递指针就可以，但必须先初

始化为 addr 所指向结构的大小。

（4）功能

Server 用它响应连接请求，建立与 Client 连接。accept 是阻塞函数，服务器端会一直阻塞到有一个客户程序发出了连接。

7.2.5 recv

（1）函数原型

```
ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
```

（2）返回值

成功,则返回实际接收的字节数；失败，则返回-1，相应地设定全局变量 errno；为 0，则表示对端已经关闭。

（3）参数

① sockfd：接收端套接字描述符。

② buf：指向容纳接收信息的缓冲区的指针。

③ nbytes：buf 缓冲区的大小。

③ flags：接收标志。

（4）功能

用于 TCP 协议中接收信息，recv 缺省是阻塞函数，直到接收到信息或出错。

7.2.6 send

（1）函数原型

```
ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);
```

（2）返回值

成功，则返回已发送的字节数；失败，则返回-1，相应地设定全局变量 errno。

（3）参数

① sockfd：指定发送端套接字描述符。

② buf：存放要发送数据的缓冲区。

③ nbytes：实际要发送的数据的字节数。

③ flags：发送标志。

（4）功能

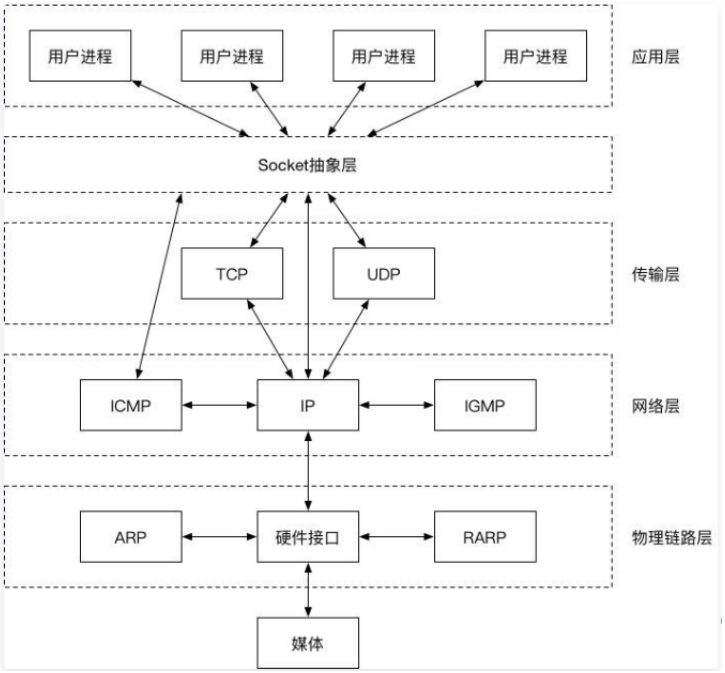
用于 TCP 协议中发送信息，send 缺省是阻塞函数，直到发送完毕或出错。

7.3 实现原理

（1）Socket

Socket 是在应用层和传输层之间的一个抽象层，它把 TCP/IP 层复杂的操作抽象为几个简单的接口，供应用层调用实现进程在网络中的通信。Socket 起源于 UNIX，在 Unix 一切皆文件的思想下，进程间通信就被冠名为文件描述符（file desciptor），Socket

是一种“打开—读/写—关闭”模式的实现，服务器和客户端各自维护一个“文件”，在建立连接打开后，可以向文件写入内容供对方读取或者读取对方内容，通讯结束时关闭文件。



（2）Socket 通信

Socket 保证了不同计算机之间的通信，也就是网络通信。对于网站，通信模型是服务器与客户端之间的通信。两端都建立了一个 Socket 对象，然后通过 Socket 对象对数据进行传输。通常服务器处于一个无限循环，等待客户端的连接，大致流程如下：