第四章 管道与信号

1、管道和命名管道

1.1、管道

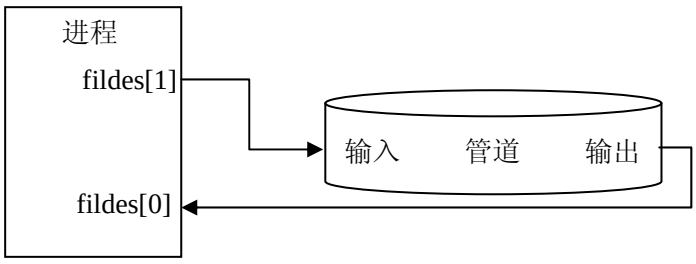
对那些需要相互配合来解决问题的进程来说，通信是一种必备的能力。最简单的 UNIX 进程间通信机制是管道，管道由特殊文件来表示。pipe 函数创建了一个被称为管道的通信缓冲区，其参数是一个由两个整数类型的文件描述符组成的数组，该函数在数组中填上两个新的文件描述符后返回。程序可以通过文件描述符 fildes[0]和 fildes[1]来访问这个缓冲区。写入 fildes[1]的数据可以按照先进先出的顺序从 fildes[0]中读出。

```
SYNOPSIS
#include <unistd.h>
int pipe(int fildes[2]);                                    POSIX
```

如果成功，pipe 返回 0，否则，返回- 1 并设置 errno。

下面语句创建了一个传统的单向通信缓冲区：

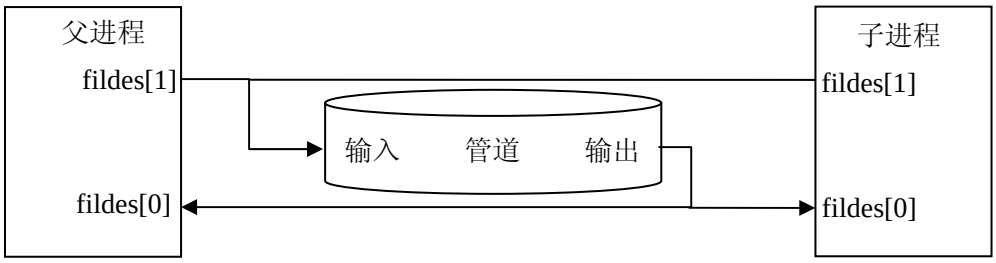
```
int fildes[2];
pipe(fildes);
```



管道是一个固定大小的缓冲区（如在 Linux 中管道大小为 4K 字节（1 页））。进程在管道上调用 read 时，如果管道非空，read 就立即返回。如果管道为空，那么只要有进程为写操作打开管道，read 就一直阻塞到有内容写入管道为止。另一方面，如果没有进程为写操作打开管道，对空管道进行的 read 调用就返回 0，表示遇到了文件结束。可以通过 fcntl 调用将对管道的读写设置为非阻塞型的。另外 lseek 调用不适用于管道。POSIX 标准也没有定义逆向操作会发生什么情况。

单个进程使用管道没有什么用处，通常父进程会用管道来与它的子进程进行通信。下面

是父进程创建管道后再创建子进程时的父子进程管道示例图：



也就是说，此时父子进程都可以往管道里写，也可以都从管道里读，这会导致不可知性。

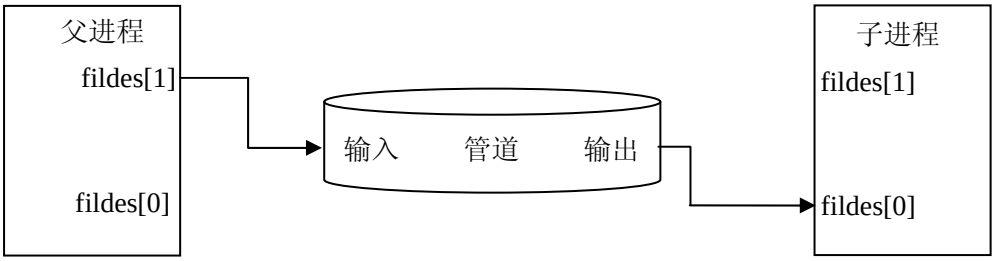
在父进程中关闭管道的输出端：

```
close(fildes[0]);
```

在子进程中关闭管道的输入端：

```
close(fildes[1]);
```

就会建立一个从父进程流向子进程的单向管道，如下图所示：



可以照猫画虎，建立一个从子进程流向父进程的单向管道。

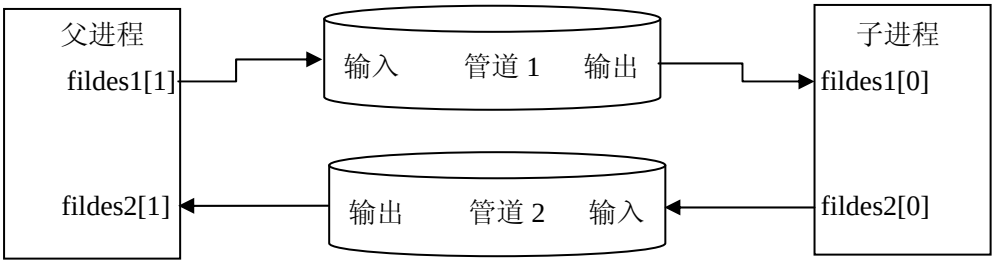
下面的程序简单演示了父子进程通过单向管道通信的过程：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>


int main(void)
{
    char buf[BUFSIZ] = "Hello world!";
    pid_t childpid;
    int fd[2];
    int readbytes;
```

```
        if (pipe(fd) == -1){
            perror("Failed to create pipe");
            return 1;
        }
        if ((childpid = fork()) == -1){
            perror("Failed to fork");
            return 1;
        }
        if (childpid == 0){
            close(fd[1]);
            if ((readbytes = read(fd[0], buf, BUFSIZ)) == -1){
                perror("Failed to read pipe");
                return 1;
            }
            fprintf(stderr, "I am child(pid=%d), what I read from pipe
is \"%s\"\n", getpid(), buf);
            return 0;
        }
        close(fd[0]);
        if ((readbytes = write(fd[1], buf, strlen(buf)+1)) == -1){
            perror("Failed to write pipe");
            return 1;
        }
        fprintf(stderr, "I am father(pid=%d), what I write to pipe is
\"%s\"\n", getpid(), buf);
        return 0;
    }
```

如果要实现父子进程之间的双向通信，就需要创建两个管道来实现：



下面的程序演示了父子进程通过两个管道进行双向通信的过程：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(void)
{
    char buf[BUFSIZ];
    pid_t childpid;
    int fd1[2],fd2[2];
    int readbytes;

    if ((pipe(fd1) == -1) || (pipe(fd2) == -1)){
        perror("Failed to create pipe one");
        return 1;
    }
    if ((childpid = fork()) == -1){
        perror("Failed to fork");
        return 1;
    }
    if (childpid == 0){
        close(fd1[1]);
        close(fd2[0]);
```

```
            for(;;){
                fprintf(stderr,  "I'm  child,please  input  string  to
father:");
                readbytes = read(STDIN_FILENO, buf, BUFSIZ);
                write(fd2[1], buf, readbytes-1);
                memset(buf, '\0', sizeof(buf));
                read(fd1[0], buf, BUFSIZ);
                fprintf(stderr, "I am child \"%s\" come from father\n",
buf);
            }
            return 0;
        }
        close(fd1[0]);
        close(fd2[1]);
        for(;;){
            memset(buf, '\0', sizeof(buf));
            read(fd2[0], buf, BUFSIZ);
            fprintf(stderr, "I am father \"%s\" come from child\n",
buf);
            fprintf(stderr, "NOw, please input string to child:");
            readbytes = read(STDIN_FILENO, buf, BUFSIZ);
            write(fd1[1], buf, readbytes-1);
            printf("now strlen is %d\n",strlen(buf));
        }
        return 0;
    }
```

还可以利用管道来实现进程间的同步。下面的程序演示了同步的进程扇。父进程在创建它的 n-1 个子进程之前创建了管道。在创建了所有的子进程之后，父进程向管道中写入 n 个

字符。包括父进程在内的每个进程在向标准错误输出信息之前，都要从管道中读出一个字符。由于来自管道中的读操作会保持阻塞，直到有内容可读为止，所以每个子进程都回等待，直到父进程向管道中写入为止，这样就提供了一种被称为路障（barrier）的同步点。在所有的进程创建好之前，没有哪个进程可以向标准错误进行任何的写操作。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

int main(int argc, char *argv[])
{
    char buf[] = "g";
    pid_t childpid = 0;
    int fd[2];
    int i,n,x;

    if (argc != 2){
        fprintf(stderr, "Usage: %s processes\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    if (pipe(fd) == -1){
        perror("Failed to create the synchronization pipe");
        return 1;
    }
    for (i = 1; i < n; i++){
        if ((childpid = fork()) <= 0)
            break;
    }
```

```
        if (childpid > 0){
            for (i = 0; i < n; i++){
                sleep(1);
                while ((x = write(fd[1], buf, 1) != 1) && (errno
==EINTR));
                if (x == -1)
                    perror("Failed    to    wtite    synchronization
characters");
            }
        }
        while ((x = read(fd[0], buf, 1) != 1) && (errno == EINTR));
        if (x == -1)
            perror("Failed to read synchronization characters");
        fprintf(stderr, "i: %d process ID: %ld parent ID: %ld child ID:
%ld\n ", i, (long)getpid(), (long)getppid(), (long)childpid);
        return (childpid == -1);
    }
```

1.2、命名管道

没有进程打开管道时管道就消失了，从这个意义上来说，管道是临时的。POSIX 用特殊文件来表示 FIFO 或命名管道（named pipe），这些文件在所有的进程都将其关闭之后仍然存在。FIFO 像普通文件一样，有名字和访问权限，而且会出现在 ls 列出的目录列表中，任何一个具有恰当权限的进程都可以访问 FIFO。FIFO 不支持诸如 lseek()等文件定位操作。

可以通过 mkfifo 命令或 mkfifo 系统调用来创建命名管道。

```
SYNOPSIS
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);          POSIX
```

mkfifo 函数创建新的 FIFO 特殊文件，创建的文件对应于 path 指定的路径名，参数 mode 为新创建的 FIFO 指定了权限。如果成功，mkfifo 就返回 0，否则返回-1 并设置 errno。

删除 FIFO 与删除普通文件的方法相同，即用 rm 命令或 unlink 函数。

1.3、管道与客户机-服务器模型

客户机-服务器模型是进程间交互的一种标准模式。一个被称为客户机（client）的进程，向另一个被称为服务器（server）的进程请求服务。主要有两种类型的客户机-服务器通信方式——简单-请求（simple-request）和请求-应答（request-reply）方式。在简单-请求方式中，客户机以单向传输的方式向服务器发送请求信息；在请求-应答方式中，客户机发送一个请求，服务器就会发送一个应答。

管道和 FIFO 都有一个很重要的特性，那就是它们能够保证不超过 PIPE_BUF（linux 中 4K）的写入操作是原子的，也就是说它们能够保证信息是作为一个单元写入的，不会插入其他写操作写入的字节。与之相反，fprintf 不是原子的，因此，来自多个客户机的消息片断可能会发生交错。但同时，对管道的读操作却不是原子的，这使得从管道中读取信息有可能发生错误，尽管这种可能性比较小，但仍需要采取措施。

管道在简单-请求方式中是一个不错的选择，而在请求-应答方式中就存在一些问题。下面程序演示了管道在简单-请求方式中的应用。客户机将信息（日志）写入命名管道，服务器将客户机的信息从命名管道中读出并写入日志文件。中间增加管道的作用使得各客户机的日志不会发生交错现象。

```
/*  Server  */
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include "mylib.h"
#define FIFOARG 1
#define FIFO_PERMS (S_IRWXU | S_IWGRP | S_IWOTH)

int main(int argc, char *argv[])
{
    int requestfd;
```

```
        if (argc != 2){
            fprintf(stderr, "Usage: %s fifoname >logfile\n", argv[0]);
            return 1;
        }
        if ((mkfifo(argv[FIFOARG], FIFO_PERMS) == -1) && (errno != 
EEXIST)){
            perror("Server failed to create a FIFO");
            return 1;
        }
        if ((requestfd = open(argv[FIFOARG], O_RDWR)) == -1){
            perror("Server failed to open its FIFO");
            return 1;
        }
        copyfile(requestfd, STDOUT_FILENO);
        return 1;
    }
    /* Client  */
    #include <errno.h>
    #include <fcntl.h>
    #include <limits.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <time.h>
    #include <unistd.h>
    #include <sys/stat.h>
    #define FIFOARG 1
```

```
int main(int argc, char *argv[])
{
    time_t curtime;
    int len,x;
    char requestbuf[PIPE_BUF];
    int requestfd;

    if (argc != 2){
        fprintf(stderr, "Usage: %s fifoname", argv[0]);
        return 1;
    }
    if ((requestfd = open(argv[FIFOARG], O_WRONLY)) == -1){
        perror("Client failed to open log fifo for writing");
        return 1;
    }
    curtime = time(NULL);
    snprintf(requestbuf,  PIPE_BUF,  "%d:  %s",  (int)getpid(),
ctime(&curtime));
    len = strlen(requestbuf);
    while (((x = write(requestfd, requestbuf, len)) != len) &&
(errno == EINTR));
    if (x == -1){
        perror("Client failed to write");
        return 1;
    }
    close(requestfd);
    return 0;
}
```

服务器进程从命令行接受命名管道，并把标准输出重定向到日志文件。如果命名管道不

存在，就创建一个命名管道。尽管服务器不会向管道写入，它也要为读操作和写操作打开管道，因为当为读操作打开管道时，open 会保持阻塞直到有进程为写操作打开管道。由于服务器为写操作打开了管道，因此 open 就不会阻塞了，同时，服务器为写操作打开管道，也使得 copyfile 永远也不会检测到文件尾，导致 read 阻塞，使服务器永远运行下去，满足了服务器进程的运行要求。

客户机进程只是简单的将自己的进程号和启动时间作为信息写入命名管道即退出。

练习 4-1：参照上述简单-请求管道模型，修改简单 shell 程序使其支持历史命令记录。

2、信号

2.1 信号的基本概念

尽管大多数进程之间的通信是计划好的，但也有一些进程间的通信问题是不可预知的。UNIX 操作系统处理这种异步事件的方法是信号机制。信号（signal）是传送给进程的事件通知，它可以完成进程间异步事件的通信。比如对一个正在运行的前台进程，当用户按“Ctrl_C”组合键时，shell 进程就接受到这个组合键，并根据正常设置产生程序终止信号 SIGINT，然后前台进程接受并响应该信号，从而终止进程。

UNIX 定义了一组异步事件，为每个事件分配了一个信号。每个信号都有一个以 SIG 开头的符号名。信号的名字都定义在 signal.h 中，任何一个使用了信号的 C 程序都要包含这个头文件。信号的名字表示的是大于 0 的小整数。下表列出了必需的 POSIX 信号及它们的默认行为。“kill -l”命令可以列举当前系统支持的信号名称，“man 7 signal”命令可以查看当前系统支持的信号的详细信息。

信号	描述	默认行为
SIGABRT	进程放弃	与实现有关
SIGALRM	告警时钟	非正常终止
SIGBUS	访问了内存对象中的未定义部分	与实现有关
SIGCHLD	子进程被终止、停止或继续	忽略
SIGCONT	如果进程被停止了，本信号使进程继续执行	继续
SIGFPE	算术计算中出现了被零除这样的错误	与实现有关
SIGHUP	在控制终端（进程）上挂起（死亡）	非正常终止
SIGILL	无效的硬件指令	与实现有关

SIGINT	交互终端提示信号（通常是 Ctrl-C）	非正常终止
SIGKILL	终止（不能被捕捉或忽略）	非正常终止
SIGPIPE	向一个没有读程序的管道写入	非正常终止
SIGQUIT	交互终端终止（通常为 Ctrl-|）	与实现有关
SIGSEGV	无效的内存引用	与实现有关
SIGSTOP	执行停止（不能被捕捉或忽略）	停止
SIGTERM	终止	非正常终止
SIGTSTP	终端停止	停止
SIGTTIN	后台进程试图进行读操作	停止
SIGTTOU	后台进程试图进行写操作	停止
SIGURG	在套接字上有高带宽数据	忽略
SIGUSR1	用户定义的信号 1	非正常终止
SIGUSR2	用户定义的信号 2	非正常终止

引发信号的事件发生时，信号就被生成（generate）了，进程根据信号采取行动时，信号就被传递（deliver）了，信号的寿命（lifetime）就是信号的生成和传递之间的时间间隔。已经生成但还未被传递的信号称为挂起（pending）的信号，在信号生成和信号传递之间可能会有相当长的时间。传递信号时，进程必须在处理器上运行。如果在传递信号时进程执行了信号处理程序（signal handler），那么进程就捕捉（catch）到了这个信号，如果将进程设置为忽略（ignore）某个信号，那么在传递时那个信号就会被丢弃，不会对进程产生影响。如果需要，进程可以阻塞（block）某个信号，这样当某个信号传递时，进程就会阻止其传递，从而挂起该信号。当进程解除对某信号的阻塞时，该信号就会被传递出去。阻塞信号不同于忽略信号，忽略信号就是丢弃信号，进程不做任何反应。

信号机制是在软件层次上对中断机制的模拟。从概念上讲，一个进程接受到一个信号与一个处理器接受到一个中断请求是一样的。一个进程所接收到的信号可以来自其他进程，可以来自外部事件，也可以来自进程自身。最重要的是，信号和中断都是“异步”的。处理器在执行一段程序时并不需要停下来等待中断的发生，也不知道中断会何时发生。信号也一样，一个进程并不需要通过一个什么样的操作来等待信号的到达，也不知道信号会什么时候到达。另外，本质上，信号仍然属于父子进程之间的通信，因为要发送信号时必须知道进程 ID，而某进程的 ID 只有其父进程才知道。

2.2 信号的产生

导致信号产生的原因很多，但总体说来有三种可能：1、程序错误：当硬件出现异常、除数为 0 或软件非法访问等情况时产生；2、外部事件：当定时器到达、用户按键中断或进程调用 abort 等信号发送函数时产生；3、显式请求：当进程调用 kill、raise 等信号发送函数或者用户执行 kill 命令传递信号时产生。

在命令解释程序上可以用 kill 命令产生信号，历史上很多信号的默认行为都是将进程终止，kill 名字就由此而来（kill 命令不一定杀死进程，但主要还是杀死），参数 signal_name 是一个信号的符号名，即删除 SIG 后的得到的名称。

```
SYNOPSIS
kill -s signal_name pid ...
kill -l [exit_status]
kill [-signal_name] pid ...
kill [-signal_number] pid ...     POSIX:Shell and Utilities
```

最后一种格式仅支持下列 signal_number 值：0 为信号 0，1 为信号 SIGHUP，2 为信号 SIGINT，3 为信号 SIGQUIT，6 为信号 SIGABRT，9 为信号 SIGKILL，14 为信号 SIGALRM，15 为信号 SIGTERM。

在一个程序中调用 kill 函数会向一个进程发送信号。kill 函数将进程 ID 和一个信号码作为参数。如果 pid 大于 0，kill 就向那个 ID 表示的进程发送信号。如果 pid 为 0，kill 就向调用程序的进程组成员发送信号。如果 pid 为-1，kill 就向所有它有权发送信息的进程发送信号。如果 pid 是其他负数值，kill 就将信号发送到组 ID 等于|pid|的进程组中去。

```
SYNOPSIS
include <signal.h>
int kill(pid_t pid, int sig);                    POSIX:CX
```

如果成功，kill 返回 0，否则返回-1 并设置 errno。

用户只能向其拥有的进程发送信号，对大多数信号来说，kill 通过比较进程和目标进程的用户 ID 来确定它是否有权发送信号。

进程可以用 raise 函数向自己发送一个信号。raise 函数只有一个参数，即信号码。

```
SYNOPSIS
include <signal.h>
```

```
int raise(int sig);                                    POSIX:CX
```

如果成功，raise 返回 0，否则返回一个非零的错误值并设置 errrno。

经过特定的秒数之后，alarm 函数会向调用者发送信号 SIGALRM。由于对 alarm 的请求不会被排队处理，因此在前一个定时器到时之前对 alarm 的调用会将 alarm 重置为一个新的值。调用 alarm 时，seconds 值为 0 就可以取消前一个告警请求。

```
SYNOPSIS
include <unistd.h>
unsigned alarm(unsigned seconds);                      POSIX
```

alarm 函数返回在重置之前告警值中剩余的秒数，或者以前如果没有设置的话，函数就返回 0。alarm 不报告错误。

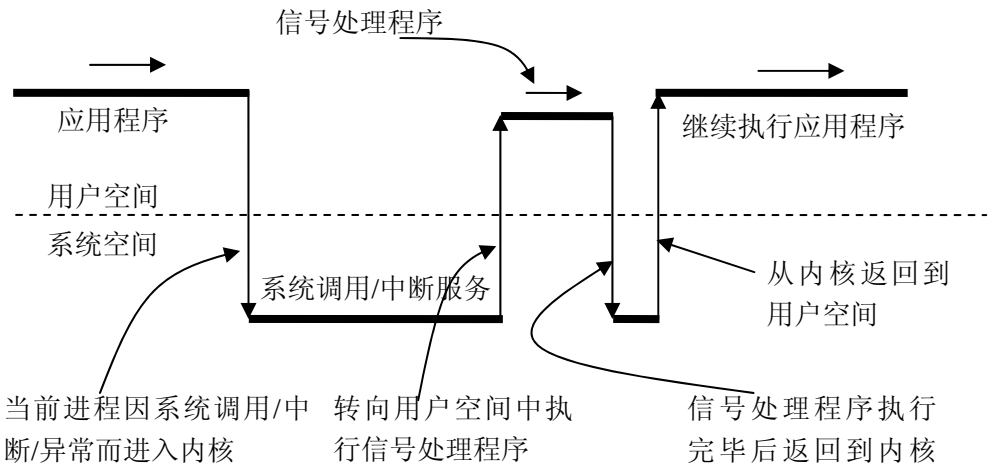
由于 SIGALRM 的默认动作是终止进程，所以下面的程序大约可以运行 10 秒：

```
#include <unistd.h>

int main(void)
{
   alarm(10);
   for( ; ; );
}
```

2.3 信号的处理

对信号的检测与响应总是发生在系统空间，通常发生在两种情况下：（1）当前进程由于系统调用、中断或异常而进入系统空间以后，从系统空间返回到用户空间的前夕；（2）当前进程在内核中睡眠状态刚被唤醒的时候，由于信号的存在而提前返回到用户空间。当有信号要响应时，处理器执行路线如下图所示。



当进程收到信号时有如下三种处理方式：1、系统默认：这是进程对信号的一般处理方式，由 UNIX 内核完成。系统对不同的信号采用不同的默认处理方式，包括终止、忽略、core-dump、挂起和继续等。其中挂起和继续处理方式主要应用于程序调试过程中，而 core-dump 则使进程退出，并将进程内存的数据和存储器状态以某种特殊的格式转存到文件系统 core 文件中。本类信号在程序异常时发生，产生 core 文件供程序员参考；2、忽略信号：信号接收后，立即丢弃。注意信号 SIGSTOP 和 SIGKILL 不能忽略，否则将会出现超级用户无法停止的进程，另外建议对某些信号不能忽略，比如硬件异常信号等；3、捕获信号：进程接收信号，并且调用自定义的代码响应之。

2.4 对信号掩码和信号集进行操作

进程可以通过阻塞信号暂时地阻止信号的传递。在传递之前，被阻塞的信号不会影响进程的行为。进程的信号掩码（signal mask）给出了当前被阻塞的信号的集合，信号掩码的类型为 sigset_t。

进程通过用 sigprocmask 函数来检查或修改它的进程信号掩码。参数 how 是一个整数，用来说明信号掩码的修改方式。参数 set 是指向一个信号集的指针，在修改中要用到这个信号集，如果 set 为 NULL，说明不需要进行修改。如果参数 oset 不为 NULL，sigprocmask 会将修改之前的信号集放在*oset 中返回。

```
SYNOPSIS
include <signal.h>
int sigprocmask(int how,  const sigset_t *restrict set,
    sigset_t *restrict oset);                    POSIX
```

如果成功，sigprocmask 返回 0，否则返回-1 并设置 errno。how 可以取下列某个值：

SIG_BLOCK　　向当前被阻塞的信号中添加一个信号集

SIG_UNBLOCK　　从当前被阻塞的信号中删除一个信号集

SIG_SETMASK　　将指定的信号集设置为被阻塞的信号

函数 sigprocmask 显示，对进程信号掩码的操作是通过信号集来进行的，顾名思义，信号集是几个信号的集合，它也是一个 sigset_t 类型的值，有 5 个基本的函数来操作信号集，每个函数的第一个参数都是一个指向 sigset_t 的指针。sigaddset 负责将 signo 加入信号集，而 sigdelset 则将 signo 从信号集中删除。sigemptyset 对一个 sigset_t 类型的信号集进行初始化，使其不包含任何信号（复位），而 sigfillset 则对一个 sigset_t 类型的信号集进行初始化，使其包含所有的信号（置位），在使用信号集之前，调用 sigemptyset 或 sigfillset 对其进行初始化。sigismember 负责报告 signo 是否在 sugset_t 中。

```
SYNOPSIS
include <signal.h>
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t, *set);
int sigismember(const sigset_t *set, int signo);     POSIX
```

下面的程序显示一条消息，在做一些无用工作的同时阻塞 SIGINT 信号，并解除对信号的阻塞，然后做更多无用的工作。程序在一个循环中不断的重复这个工作。如果用户在 SIGINT 被阻塞的情况下输入了 Ctrl-C，程序会在终止之前完成计算并打印一条消息。如果用户在 SIGINT 被解除了阻塞的情况下输入 Ctrl-C，程序就会立即终止。

```
#include <math.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
```

```
        int i;
        sigset_t intmask;
        int repeatfactor;
        double y = 0;

        if (argc != 2){
           fprintf(stderr, "Usage: %s repeatfactor\n", argv[0]);
           return 0;
        }
        repeatfactor = atoi(argv[1]);
        if ((sigemptyset(&intmask) == -1) ||
           (sigaddset(&intmask, SIGINT) == -1)){
           perror("Failed to initialize the signal mask");
           return 1;
        }
        for (;;){
           if (sigprocmask(SIG_BLOCK, &intmask, NULL) == -1)
              break;
           fprintf(stderr, "SIGINT signal blocked\n");
           for (i = 0; i < repeatfactor; i++)
              y += sin((double)i);
           fprintf(stderr, "Blocked calculation is finished, y=%f\n",
y);
           if (sigprocmask(SIG_UNBLOCK, &intmask, NULL) == -1)
              break;
           fprintf(stderr, "SIGINT signal unblocked\n");
           for (i = 0; i < repeatfactor; i++)
              y += sin((double)i);
           fprintf(stderr,  "Unblocked  calculation  is  finished,
```

```
y=%f\n", y); }
        perror("Failed to change signal mask");
        return 1;
    }
```

2.5 捕捉与忽略信号-sigaction

sigaction 函数允许调用程序检查或指定与特定信号相关的动作。参数 sig 用来指定动作的信号码，参数 act 是一个指向 struct sigaction 结构的指针，用来说明要采取的动作，参数 oact 也是一个指向 struct sigaction 结构的指针，用来接收与信号相关的前一个动作。如果 act 为 NULL，对 sigaction 的调用就不能改变与信号相关的动作，如果 oact 为 NULL，对 sigaction 的调用就不会返回与信号相关的前一个动作。

```
SYNOPSIS
include <signal.h>
int sigaction(int sig, const struct sigaction *restrict act,
struct sigaction *restrict oact);                    POSIX
```

如果成功，sigaction 返回 0，否则，返回-1 并设置 errno。

struct sigaction 结构必须至少包含下列成员：

```
struct aigaction{
    void (*sa_handle) (int);          /* SIG_DFL、SIGIGN 或者指向函数的指针 */
    sigset_t sa_mask;          /* 处理程序的执行过程中需要阻塞的额外的信号 */
    int sa_flags;                    /* 特殊的标志符合选项 */
    void (*sa_sigaction) (int, siginfo_t *, void *);      /* 实时处理程序 */
```

sa_handle 和 sa_sigaction 的存储可能会产生重叠，一个应用程序只能用这两个成员中的一个来指定它的行为，如果 sa_flags 字段的标志符 SA_SIGINFO 被清除了，sa_handle 就用来说明应该为特定信号采取的动作；如果设置了 sa_flags 字段的标志符 SA_SIGINFO，而且实现也支持 POSIX：RTS 或 POSIX：XSI 扩展，sa_sigaction 字段指定的就是一个信号捕捉函数。

struct sigaction 的成员 sa_handle 有两个特殊的值 SIG_DFL 和 SIG_IGN，前者指示 sigaction 应该恢复信号的默认行为，后者则指示进程应该忽略信号（丢弃它）。

下面的代码段将 SIGINT 的信号处理程序设置为 mysighand：

```
    struct sigaction newact;

    newact.sa_handle = mysighand; /* set the new handle */
    newact.sa_flags = 0;   /* no special options */
    if ((sigemptyset(&newact.sa_mask) == -1) || /* no block signal */
       (sigaction(SIG_INT, &newact, NULL) == -1))
       perror("Failed to install SIGINT signal handle");
```

如果是信号的默认行为在起作用，下面的代码段会使进程忽略 SININT 信号：

```
    struct sigaction act;

    if (sigaction(SIGINT, NULL, &act) == -1)  /* Find current SININT
handle */
       perror("Failed to get old handle for SIGINT");
    else if (act.sa_handle == SIG_DFL){  /*  if  SIGINT  handle  is
default */
       act.sa_handle == SIG_IGN;    /* set new handle to ignore */
       if (sigaction(SIGINT, &act, NULL) == -1)
          perror("Failed to ignore SIGINT");
    }
```

下面的程序在收到 Ctrl-C 后得体的终止：

```
    #include <math.h>
    #include <signal.h>
    #include <stdio.h>
    #include <stdlib.h>

    static volatile sig_atomic_t doneflag = 0;
    static void setdoneflag(int signo)
    {
       doneflag = 1;
```

```
}

int main(void)
{
    struct sigaction act;
    int count = 0;
    double sum = 0;
    double x;

    act.sa_handler = setdoneflag;
    act.sa_flags = 0;
    if ((sigemptyset(&act.sa_mask) == -1) ||
        (sigaction(SIGINT, &act, NULL) == -1)){
        perror("Failed to set SIGINT habdles");
        return 1;
    }
    while(!doneflag){
        x = (rand()+0.5)/(RAND_MAX+1.0);
        sum += sin(x);
        count++;
        printf("Count is %d and average is %f\n", count, sum/count);
    }
    printf("Program terminating ...\n");
    if(count == 0)
        printf("No values calculted yet\n");
    else
        printf("Count is %d and average is %f\n", count, sum/count);
    return 0;
}
```

由于信号处理程序可以在 main 函数检查 doneflag 时对这个变量进行修改，因此访问 doneflag 的代码是一个临界区，这里将其声明为 sig_atomic_t 来处理这个问题，sig_atomic_t 是一个小到可以实现原子访问的整数类型。

关于 volatile 限定符。

在 C 语言中，声明（declaration）用于说明每个标识符的含义，而并不需要为每个标识符预留存储空间，预留存储空间的声明称为定义（definition）。声明标识符时需要用存储类说明符（auto、register、static、extern 和 typedef 等），类型说明符（void、char、short、int、long、float、double、signed 和 unsigned 等），类型限定符（const 和 volatile）等。每个声明中最多只能有一个存储类说明符；类型说明符中除 long 和 short 可与 int 一起使用，signed 和 unsigned 可与 int、short 和 long 一起使用外，一个声明中也最多只能使用一个类型说明符；类型限定符可与任何类型说明符一起使用：const 告诉编译器初始化以后就不能再进行赋值，而 volatile 则告诉编译器该变量可能会被意想不到的改变，这样，编译器就不会去假设这个变量的值了。精确的说就是，优化器在每次用到这个变量时必须小心的重读取这个变量的值（from memory），而不是使用保存在寄存器里的备份。下面是 volatile 变量的几个例子：（1）并行设备的硬件寄存器；（2）一个中断服务子程序中会访问到的非自动变量（局部变量）；（3）多线程应用中被几个任务共享的变量。在嵌入式系统编程中经常会遇到 volatile 变量，注意，这只是告诉编译器，如果你自己直接写汇编，就不存在这些问题了。一个最经典的例子是对于一个设备状态寄存器的变量，它就既是 const 的（用户在程序中不能修改，只能读取）又是 volatile 的（在程序中要小心（是编译器应该小心）它可能被意想不到的修改）。

一个例子：

```
int square(volatile int *ptr)
{
    return *ptr * *ptr;
}
```

因为是 volatile 的，编译器应该将其编译为：

```
int square(volatile int *ptr)
{
    int a;
    a = *ptr;
```

```
    return a*a;
}
```

而不应该编译为：

```
int square(volatile int *ptr)


{
    int a, b;
    a = *ptr;
    b = *ptr;
    return a*b;
}
```

2.6 等待信号——pause、sigsuspend 和 sigwait

多进程完成任务时，经常需要相互之间同步、协调。

信号提供了一种不需要忙等（busy waiting）来等待事件的发生。忙等是指连续地使用 CPU 周期来检测事件的发生。通常，程序都通过在循环中测试一个变量的值来进行这种检测。更有效的方法是将进程挂起直到所等待的事件发生为止，这样，其他的进程就可以更有效的使用 CPU 了。

pause 函数将调用线程挂起，直到传递了一个信号为止，这个信号的动作如果是终止进程，pause 就不返回；如果信号的动作是执行用户定义的处理程序，pause 就会在信号处理程序返回之后返回。

```
SYNOPSIS
include <unistd.h>
int pause(void);                                    POSIX
```

pause 总是返回-1。如果被信号中断，pause 就将 errno 设置为 EINTR。

pause 的问题在于调用进程不知道是哪个信号使 pause 返回，因此需要在信号处理程序中设置一个标志符：

static volatile sig_stomic_t sigreceived = 0；

while(sigreceived == 0)

pause();

但是，如果在测试 sigreceived 和调用 pause 之间传递了信号，则该信号对 pause 不起作用，使得程序不能正常运行。正常的解决方案是必须在信号阻塞的情况下测试 received 的值，但如此，则 pause 又收不到信号（因为被阻塞了），如果程序在执行 pause 之前解除了对信号的阻塞，那么就有可能在解除阻塞和执行 pause 之间收到信号。实际上，这种情况比它看起来更容易发生，如果信号在进程将其阻塞的时候产生，那么在解除了对信号的阻塞之后，进程会立即收到信号。

函数 sigsuspend 提供了一种实现原子操作的办法来避免上述问题的出现。sigsuspend 函数用 sigmask 指向的那个掩码来设置信号掩码，并将进程挂起，直到进程捕捉到信号为止。被捕捉信号的信号处理程序返回时，sigsuspend 就返回。可以用参数 sigmask 来解除对程序正在寻找的那个信号的阻塞。当 suspend 返回时，信号掩码被重置为它在 sigsuspend 被调用之前的那个值。

```
SYNOPSIS
include <unistd.h>
int sigsuspend(const sigset_t, *sigmask);          POSIX
```

sigsuspend 函数总是返回-1 并设置 errno，如果被信号中断，则置 errno 为 EINTR。

下面的代码段是等待单个信号的正确方法：

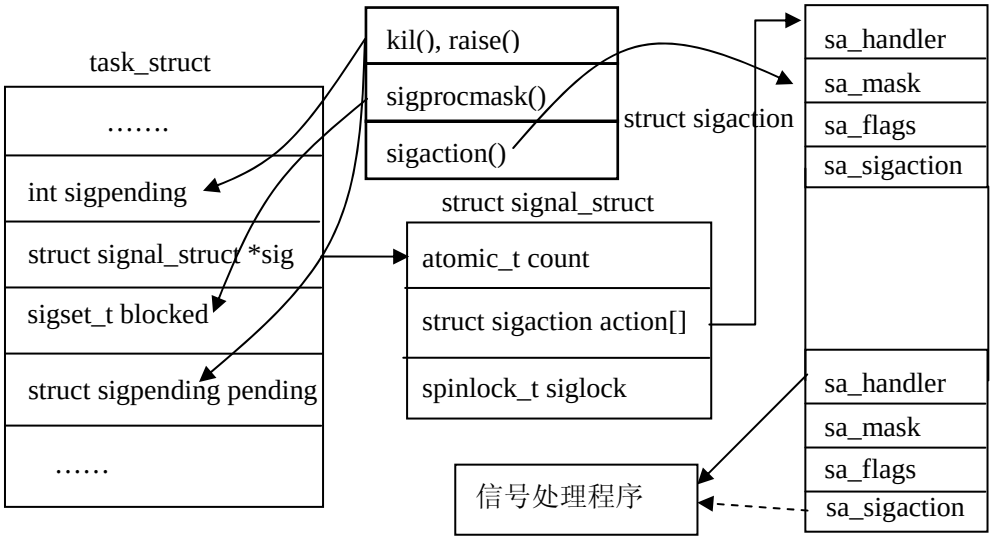
```
1  static volatile sig_atomic_t sigreceived = 0;
2
3  sigset_t maskall, maskmost, maskold;
4  int signum = SIGUSER1;
5
6  sigfillset(&maskall);
7  sigfillset(&maskmost);
8  sigdelset(&maskmost, signum);
9  sigprocmask(SIG_SETMASK, &maskall, &maskold);
10 if (sigreceived == 0);
11     sigsuspend(&maskmost);
12 sigprocmask(SIG_SETMASK, &maskold, NULL);
```

下面的代码允许在等待 signum 时处理其他的信号：

```
1  static volatile sig_atomic_t sigreceived = 0;
2
3  sigset_t maskblocked, maskunblocked, maskold;
4  int signum = SIGUSER1;
5
6  sigprocmask(SIG_SETMASK, NULL, &maskblocked);
7  sigprocmask(SIG_SETMASK, NULL, $maskunblocked);
8  sigaddset(&maskblocked, signum);
9  sigdelset(&maskunblocked, signum);
10 sigprocmask(SIG_BLOCK, &maskblocked, &maskold)
11 while (sigreceived == 0);
12    sigsuspend(&maskunblock);
13 sigprocmask(SIG_SETMASK, &maskold, NULL);
```

下面的代码更简洁，但功能相同：

```
1  static volatile sig_atomic_t sigreceived = 0;
2
3  sigset_t masknew, maskold;
4  int signum = SIGUSER1;
5
6  sigprocmask(SIG_SETMASK, NULL, &masknew);
7  sigaddset(&masknew, signum);
8  sigprocmask(SIG_SETMASK,&masknew, &maskold);
9  sigdelset(&masknew, signum);
10 while (sigreceived == 0);
11    sigsuspend(&masknew);
12 sigprocmask(SIG_SETMASK, &maskold, NULL);
```



下面的程序给出了在一个指定信号上阻塞的函数的实现模板。先将前一个程序用-c 参数编译，在编译后一个程序的时候将前面产生的.o 文件用 gcc 链接上。运行程序后，根据显示信息提供的进程 ID，在另一终端用命令“kill SIGUSR1 进程 ID 号”传递信号。注意初始化和恢复均指信号处理程序而非信号掩码。还要注意函数的编写，以及参数的传递，函数中变量被设置为 static，传递参数时只在初始化函数中传递信号值 signo，然后保存在 signum 静态变量中，因此调用其它操作该信号的函数时就不用传递了，当然应该先调用初始化函数。

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

static int isinitialized = 0;
static struct sigaction oact;
static int signum = 0;
static volatile sig_atomic_t sigreceived = 0;

/* ARGSUSED */
static void catcher(int signo)
{
   sigreceived = 1;
}
```

```
int initsuspend(int signo)
{
   struct sigaction act;

   if (isinitialized)
      return 0;
   act.sa_handler = catcher;
   act.sa_flags = 0;
   if ((sigfillset(&act.sa_mask) == -1) ||
      (sigaction(signo, &act,&oact) == -1))
      return -1;
   signum = signo;
   isinitialized = 1;
   return 0;
}

int restore(void)
{
   if (!isinitialized)
      return 0;
   if (sigaction(signum, &oact, NULL) == -1)
      return -1;
   isinitialized = 0;
   return 0;
}

int simplesuspend(void)
{
```

```
    sigset_t maskblocked, maskold, maskunblocked;

    if (!isinitialized){
        errno = EINVAL;
        return -1;
    }
    if ((sigprocmask(SIG_SETMASK, NULL, &maskblocked) == -1) ||
        (sigaddset(&maskblocked, signum) == -1) ||
        (sigprocmask(SIG_SETMASK, NULL, &maskunblocked) == -1) ||
        (sigdelset(&maskunblocked,signum) == -1) ||
        (sigprocmask(SIG_SETMASK, &maskblocked, &maskold) == -1))
        return -1;
    while(sigreceived == 0)
        sigsuspend(&maskunblocked);
    sigreceived == 0;
    return sigprocmask(SIG_SETMASK, &maskold, NULL);
}

#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int initsuspend(int signo);
int restore(void);
int simplesuspend(void);

int main(void)
{
    fprintf(stderr, "This is process %ld\n", (long)getpid);
```

```
    for(;;){
        if (initsuspend(SIGUSR1)){
            perror("Failed to setup handler for SIGUSR1");
            return 1;
        }
        fprintf(stderr, "Waiting for signal\n");
        if (simplesuspend()){
            perror("Failed to suspend for signal");
            return 1;
        }
        fprintf(stderr, "Got signal\n");
        if (restore()){
            perror("Failed to restore original handler");
            return 1;
        }
    }
    return 1;
}
```

下面的程序对上面的程序进行修改，增加 SIGQUIT 和 SIGINT 信号的信号处理程序，但是仍然只挂起 SIGUSR1 信号：

```
#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

static int isinitialized = 0;
static struct sigaction oact;
static int signum = 0;
static volatile sig_atomic_t sigreceived = 0;
```

```
    /* ARGSUSED */
    static void sigusr1(int signo)
    {
       sigreceived = 1;
       fprintf(stderr, "Now executing signal SIGUSR1 processing
program\n");
    }

    static void sigint(int signo)
    {
           fprintf(stderr, "Now executing signal SIGINT processing
program\n");
    }

    static void sigquit(int signo)
    {
           fprintf(stderr, "Now executing signal SIGQUIT processing
program\n");
    }

    int initsuspend(int signo)  /* set up handler for the pause */
    {
       struct sigaction act;

       if (isinitialized)
          return 0;
       if (signo == SIGINT)
          act.sa_handler = sigint;
```

```
        if (signo == SIGQUIT)
            act.sa_handler = sigquit;
        if (signo == SIGUSR1)
            act.sa_handler = sigusr1;
        act.sa_flags = 0;
        if  ((sigfillset(&act.sa_mask)  ==  -1)  ||(sigaction(signo,
&act,&oact) == -1))
            return -1;
        if (signo == SIGUSR1){
            signum = signo;
            isinitialized = 1;
        }
        return 0;
    }

    int restore(void)
    {
        if (!isinitialized)
            return 0;
        if (sigaction(signum, &oact, NULL) == -1)
            return -1;
        isinitialized = 0;
        return 0;
    }

    int simplesuspend(void)
    {
        sigset_t maskblocked, maskold, maskunblocked;
```

```
    if (!isinitialized){
        errno = EINVAL;
        return -1;
    }
    if ((sigprocmask(SIG_SETMASK, NULL, &maskblocked) == -1) ||
        (sigaddset(&maskblocked, signum) == -1) ||
        (sigprocmask(SIG_SETMASK, NULL, &maskunblocked) == -1) ||
        (sigdelset(&maskunblocked,signum) == -1) ||
        (sigprocmask(SIG_SETMASK, &maskblocked, &maskold) == -1))
        return -1;
    while(sigreceived == 0)
        sigsuspend(&maskunblocked);
    sigreceived == 0;
    return sigprocmask(SIG_SETMASK, &maskold, NULL);
}

#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int initsuspend(int signo);
int restore(void);
int simplesuspend(void);

int main(void)
{
    fprintf(stderr, "This is process %d\n", getpid());
    if (initsuspend(SIGQUIT)){
        perror("Failed to setup handler for SIGQUIT");
```

```
        return 1;
    }
    if (initsuspend(SIGINT)){
            perror("Failed to setup handler for SIGINT");
            return 1;
        }
    if (initsuspend(SIGUSR1)){
            perror("Failed to setup handler for SIGUSR1");
            return 1;
        }
    fprintf(stderr, "Waiting for signal\n");
    if (simplesuspend()){
        perror("Failed to suspend for signal");
        return 1;
    }
    fprintf(stderr, "Got signal\n");
    if (restore()){
        perror("Failed to restore original handler");
        return 1;
    }
    return 1;
}
```

2.7 信号处理：错误和信号异步安全

要意识到在信号与函数调用的交互中存在三个困难：被信号中断的 POSIX 函数是否应该重新启动；信号处理程序调用非可重入函数；对 errno 的错误处理。

如果进程在执行库函数时捕捉到了信号，会发生什么情况？答案取决于调用的类型。终端 I/O 可以将进程阻塞一段不确定的时间。对终端从键盘上获取键值或从管道中读入数据所用的时长则没有什么限制，执行这些操作的函数调用有时被称为“慢速”的。其它的操作，比如磁盘 I/O，可以阻塞很短的一段时间。还有其它一些操作，例如 getpid，根本不会阻塞。

最后这两种类型的操作都不会被认为是“慢速”的。

慢速的 POSIX 调用是那些被信号中断的调用。当信号被捕捉且信号处理程序返回时，这些调用就会返回。被中断的函数返回-1 并将 errno 置为 EINTR。查阅联机帮助页面的 ERRORS 小节，看看一个给定的函数是否能被信号中断。如果函数设置 errno，且其中一个值为 EINTR，就说明函数是可以被中断的，程序必须显式的处理这个错误，如果需要就重起这个函数。我们并不能从逻辑上判定哪些函数是属于这个范围的，因此一定要查看函数的联机帮助页面。

如果能够从信号处理程序中安全地调用一个函数，这个函数就是异步信号安全（async-signal safe）的。很多 POSIX 库函数都不是异步信号安全的，因为它们使用了静态数据结构，比如 malloc 和 free，或者是它们用一种不可重入的方式使用了全局数据结构。因此单个进程可能不能正确的执行对这些函数的并发调用。

通常这不是什么问题，但是信号向程序中引入了并发，因为信号是异步出现的，进程可能会在执行库函数的时候捕捉到一个信号，因此从信号处理程序内部调用库函数的时候，必须要特别小心。下表列出 POSIX 确保能够安全的从信号处理程序内部调用的函数。注意像 fprintf 这样来自 C 标准 I/O 库中的函数没有出现在表中。

_Exit	_exit	accept	access
aio_error	aio_return	aio_suspend	alarm
bind	cfgetispeed	cfgetospeed	cfsetispeed
cfsetospeed	chdir	chmod	chown
clock_gettime	close	connect	creat
dup	dup2	execle	execve
fchmod	fchown	fcntl	fdatasync
fork	fpathconf	fstat	fsync
ftruncate	getepid	geteuid	getgid
getgroups	getpeername	getpgrp	getpid
getppid	getsockname	getsockopt	getuid
kill	link	listen	lseek
lstat	mkdir	mkfifo	open
pathconf	pause	pipe	poll

posix_trace_event	pselect	raise	read
readlink	recv	recvfrom	recvmsg
rename	rmdir	select	sem_post
send	sendmsg	sendto	setgid
setpgid	setsid	setsockopt	setuid
shutdown	sigaction	sigaddsetq	sigdelset
sigfillset	sigismember	signal	sigpause
sigpending	sigprocmask	sigqueue	sigset
sigsuspend	sleep	socket	socketpair
stat	symlink	sysconf	tcdrain
tcflow	tcflush	tcgetattr	tcgetpgrp
tcsendbreak	tcsetattr	tcsetpgrp	time
timer_getoverrun	timer_gettime	timer_settime	times
umask	uname	unlink	wait
waitpid	write		

2.8 异步 I/O 编程

通常在执行读操作或写操作时，进程会一直阻塞直到 I/O 完成为止。对于注重性能的应用来说，需要允许 I/O 操作的处理异步（asynchronously）于程序的执行。POSIX：AIO 扩展对异步 I/O 的定义基于四个主要函数。函数 aio_read 允许进程对一个打开的文件描述符上的读操作请求进行排队，aio_write 对写操作请求进行排队，aio_return 在异步 I/O 操作执行完毕之后返回它的状态，aio_error 用来返回错误状态，aio_cancel 可以用来撤销已经在执行的异步 I/O 操作。

```
SYNOPSIS
include <aio.h>
int aio_read(struct aiocb *aiocbp);
int aio_write(struct aiocb *aiocbp);
ssize_t aio_return(struct aiocb *aiocbp);
int aio_error(const struct aiocb *aiocbp);
int aio_cancel(int fildes struct aiocb *aiocbp); POSIX:AIO
```

函数 aio_read 和 aio_write 都只有一个参数 aiocbp，它是一个指向异步 I/O 控制块的指针，结构 struct aiocb 至少包含下列成员：

```
int             aio_fildes          /*  文件描述符
volatile        *aio_buf            /*  缓冲区的位置
size_t          aio_nbytes          /*  传输的长度
off_t           aio_offset          /*  文件偏移量
int             aio_reqprio         /*  请求优先偏移量
struct sigevent aio_sigevent        /*  信号码和信号值
int             aio_lio_opcode      /* listio 操作
```

函数 aio_read 从与 aiocbp->aio_fildes 相关的文件中将 aiocbp->aio_bytes 字节读入一个由 aiocbp->aio_buff 指定的缓冲区中，请求被放入队列之后，函数就返回。aio_write 类似。如果请求被成功的放入队列，aio_read 和 aio_write 就返回 0，否则返回-1 并设置 errno。

函数 aio_error 和 aio_return 返回 aiocbp 指定的 I/O 操作的状态，用 aio_error 来监视异步 I/O 操作的进度。操作完成时，调用 aio_return 来检索读入或写出的字节数。I/O 操作成功的完成时，aio_error 返回 0，如果还在执行就返回 EINPROGRESS。如果操作失败，aio_error 返回与失败相关的代码。这个错误状态对应于可能已经由相应的 read 或 write 函数设置了的 errno 值。函数 aio_return 返回已经完成的底层 I/O 操作的状态，如果操作执行成功，返回值就是读入或写出的字节数。一旦调用了 aio_return，就不能为同一个 struct aiocb 调用 aio_return 或 aio_error 了，直到这个缓冲区开始了另一个异步操作为止。如果异步 I/O 还为完成，aio_return 的结果是未定义的。

POSIX 的异步 I/O 可以和信号一起使用，也可以不和信号一起使用，这取决于 struct aiocb 的 sigev_notify 字段的设置。