

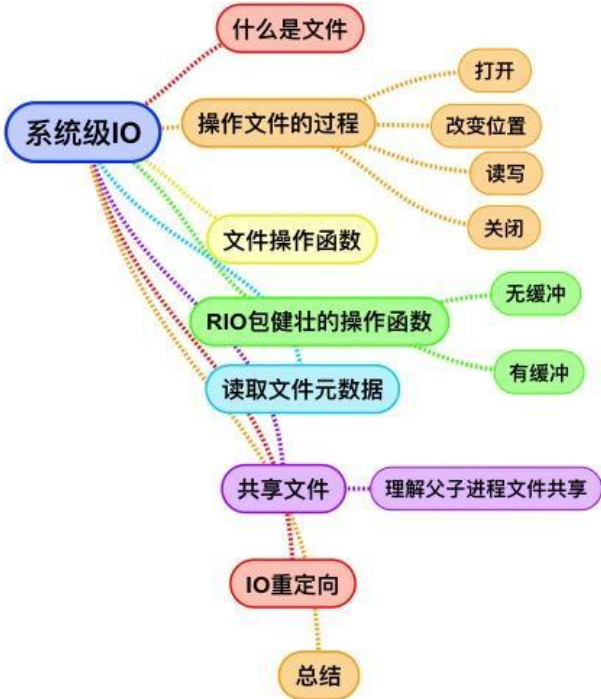
《深入理解计算机系统》| 系统级IO



唐鱼的学习探索

关注

2018.05.02 21:41:57 字数 1,504 阅读 1,099



目 录

小鹅通

小鹅通商家数突破100万

在线直播课堂系统,最高直减5888元

立即注册

广告



唐鱼的学习探索

关注

总资产 29 (约2.85元)

如何高效的准备一次考试

阅读 1,737

都9102年了，你还不知道anki是什么

阅读 102

Input是指从设备拷贝数据到内存，而Output是从内存拷贝数据到外部设备的过程，我们平时使用的都是语言提供的标准IO库，如printf和scanf，这些是通过内核提供的系统级IO函数来实现的。我们学习系统级的IO，有助于我们理解其他概念，在读取元数据的时候也需要用到系统级的IO。这一章的内容很简单，来不及解释了，开车了：

1.1 什么是Unix文件

一个Unix文件是一个m个字节的序列，所有的IO设备（网络、磁盘、终端）都被映射为文件，内核提供一个简单的接口，使得对所有这些设备的访问都是以文件的方式进行。

1.2 操作文件的一般过程

打开文件：一个应用程序通过要求内核打开相应的文件，内核将返回一个非负整数，称为描述符，记录打开文件的所有信息：标准输入（描述符0）、标准输出（描述符1）、标准错误（描述符2）

改变当前文件位置：内核保持一个文件的位置k，初始为0，表示从文件开始处偏移的字节数。

通过seek操作。

读写文件：读操作就是从文件拷贝n个字节到存储器，如果是从k处开始，就是拷贝k+n为止。文件的大小为m，如果k>m就会触发（EOF），所有就不需要明确的EOF字符了。写操作就是从存储器拷贝n个字节到文件当前位置k处。

关闭文件：内核释放打开文件时创建的数据结构，释放所有的存储器资源。

1.3 文件操作函数

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

int open(char *filename, int flags, mode_t mode);
```

打开文件

filename：是文件名

flags： O_RDONLY|O_WRONLY|O_RDWR; (O_CREAT|O_TRUNC|O_APPEND) ；

mode： 访问权限

掩码	描述
S_IRUSR S_IWUSR S_IXUSR	使用者（拥有者）能够读这个文件 使用者（拥有者）能够写这个文件 使用者（拥有者）能够执行这个文件
S_IRGRP S_IWGRP S_IXGRP	拥有者所在组的成员能够读这个文件 拥有者所在组的成员能够写这个文件 拥有者所在组的成员能够执行这个文件
S_IROTH S_IWOTH S_IXOTH	其他人（任何人）能够读这个文件 其他人（任何人）能够写这个文件 其他人（任何人）能够执行这个文件

返回值：成功为描述符，失败为-1。

```
#include <unistd.h>

int close(int fd);
```

返回：若成功则为 0，若出错则为 -1。

关闭文件：

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t n);
ssize_t write(int fd, const void *buf, size_t n);
```

unsigned int

int 可能返回负值

返回：若成功则为读的字节数，若 EOF 则为 0，若出错为 -1。

返回：若成功则为写的字节数，若出错则为 -1。

读写文件：

fd: 描述符fd的当前位置; buf: 存储器位置; n: 拷贝大小

(注: 当读取时遇到EOF、从终端读取文本行或者读写网络套接字的时候返回不足值)

1.4 对文件操作函数的封装: RIO (Robust健壮的)

RIO之所以称之为健壮的IO包, 是因为他提供了方便高效的IO访问, 你可以从一个描述符中读一些文本行, 然后读二进制, 最后再读文本行。有两类输入输出函数:

```
#include "csapp.h"

ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

返回: 若成功则为传送的字节数, 若EOF则为0 (只对rio_readn而言), 若出错则为-1。

① 无缓冲输入输出: 二进制与网络的直接读写

对于同一个描述表, 可以任意的交错调用rio_readn和rio_writen

```
1 // 健壮的R包, 无缓冲, 对read的一次封装
2 ssize_t rio_readn(int fd, void *usrbuf, size_t n)
3 {
4     size_t nleft = n;
5     ssize_t nread; // 读取的字节数
6     char *bufp = usrbuf; // 存储器位置
7
8     while(nleft > 0) // 大于0的时候开始读取
9     {
10         if((nread = read(fd, bufp, nleft)) < 0) // 从bufp位置读取nleft的数据
11         {
12             if(errno == EINTR) // 被信号处理程序中断
13                 nread = 0;
14             else
15                 return -1;
16         }
17         else if(nread == 0) // 遇到EOF文件结尾或者中断
18             break;
19         nleft -= nread; // 如果被中断了, 更新目前要读的大小 (总读-已读)
20         bufp += nread; // 如果被中断了, 更新存储器位置 (位置+偏移nread)
21     }
22     return (n - nleft); // 返回值 >= 0 已成功读取的字节数
23 }
```

```
1 ssize_t rio_write(int fd, void *usrbuf, size_t n)
2 {
3     size_t nleft = n;
4     ssize_t nwritten;
5     char *bufp = usrbuf;
6
7     while(nleft > 0)
8     {
9         if((nwritten = write(fd, bufp, nleft)) <= 0)
10         {
11             if(errno == EINTR)
12                 nwritten = 0;
13             else
14                 return -1;
15         }
16         nleft -= nwritten;
17         bufp += nwritten;
18     }
19
20     return n;
21 }
```

这是如何实现的呢：

从上面的代码不难看出，如果程序的信号处理程序返回中断，这个函数会手动重启read或者write。

```
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
```

返回：若成功则为读的字节数，若 EOF 则为 0，若出错则为 -1。

② 带缓冲的输入函数

这个函数有一个好处是，它从内部读缓冲区拷贝的一行，当缓冲区为空的时候，自动调用read填满缓冲区，效率很高。

在调用这两个函数以前，是通过rio_readinitb函数来完成一些初始化的，主要是将fd与一块缓冲区联系起来：

```
1  #define RIO_BUFSIZE 8192
2  typedef struct
3  {
4      int rio_fd;
5      int rio_cnt;
6      char *rio_bufptr;
7      char rio_buf[RIO_BUFSIZE];
8  } rio_t;
9
10 void rio_readinitb(rio_t *rp, int fd)
11 {
12     rp->rio_fd = fd;
13     rp->rio_cnt = 0;
14     rp->rio_bufptr = rp->rio_buf;
15 }
16
```

```
1  ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen)
2  {
3      int n, rc;
4      char c, *bufp = userbuf; 指向用户存储区
5
6      for (n = 1; n < maxlen; n++)
7      {
8          if ((rc = rio_read(rp, &c, 1)) == 1) 读取1个字符成功
9          {
10             *bufp++ = c; 更新后指向下一个 将读取的字符放到第一个字节处
11             if (c == '\n')
12                 break;
13          }
14          else if (rc == 0) 遇到了EOF
15          {
16             if (n == 1)
17                 return 0;
18             else
19                 break; 如果是中断，重启
20          }
21          else
22             return -1;
23      }
24      *bufp = 0; 最后一个字符赋值为0
25      return n;
26  }
27
```

```

1  ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n)
2  {
3      size_t nleft = n;
4      ssize_t nread; → 已读字节数
5      char *bufp = usrbuf; → 指向用户区域
6
7      while(nleft > 0)
8      {
9          if((nread = rio_read(rp, bufp, nleft)) < 0)
10         {
11             if(errno == ENINTR)
12                 nread = 0;
13             else
14                 return -1;
15         }
16         else if (nread == 0) → 中断重启
17             break;
18         nleft -= nread; → 更新已读数量
19         bufp += nread; → 更新指向位置
20     }
21     return (n - nleft);
22 }

```

有了数据上的这个结构，我们来看看readnb和readlineb函数的具体实现：

这里面都用到了一个带缓冲的读函数，rio_read，如下：

```

1  // 带缓冲区的read函数
2  static ssize_t rio_read(rio_t *rp, char *usrbuf, size_t n)
3  {
4      int cnt;
5
6      while(rp->rio_cnt <= 0) // 如果为空填满整个缓冲区
7      {
8          rp->rio_cnt = read(rp->rio_fd, rp->rio_buf, sizeof(rp->rio_buf));
9
10         if(rp->rio_cnt < 0)
11         {
12             if(errno != EINTR)
13                 return -1;
14         }
15         else if(rp->rio_cnt == 0) // 文件结尾
16             return 0;
17         else
18             rp->rio_bufptr = rp->rio_buf; // 重设缓冲区
19     }
20     cnt = n;
21     if(rp->rio_cnt < n) // 填满的数据小于需要读取的n
22         cnt = rp->rio_cnt;
23     memcpy(usrbuf, rp->rio_bufptr, cnt); // 将用户存储空间usrbuf一次拷贝
24     rp->rio_bufptr += cnt; // 读取成功更新指针
25     rp->rio_cnt -= cnt; // 将cnt复位
26     return cnt;
27 }

```

我们再来看一个应用：从标准输入中读取一行并显示

```

GNU nano 2.2.6      File: cpfile.c

#include "csapp.h"

int main(int argc, char **argv)
{
    int n;
    rio_t rio;
    char buf[MAXLINE];

    Rio_readinitb(&rio, STDIN_FILENO);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
        Rio_writen(STDOUT_FILENO, buf, n);
}

```

运行结果如下：

```
pi@raspberrypi:~/code $ ./cpfile
I am Neo, the one.
I am Neo, the one.
Ok
Ok
```

1.4 读取文件元数据

```
struct stat {
    dev_t      st_dev;      /* Device */
    ino_t      st_ino;      /* inode */
    mode_t     st_mode;     /* Protection and file type */
    nlink_t    st_nlink;    /* Number of hard links */
    uid_t      st_uid;      /* User ID of owner */
    gid_t      st_gid;      /* Group ID of owner */
    dev_t      st_rdev;     /* Device type (if inode device) */
    off_t      st_size;     /* Total size, in bytes */
    unsigned long st_blksize; /* Blocksize for filesystem I/O */
    unsigned long st_blocks; /* Number of blocks allocated */
    time_t     st_atime;    /* Time of last access */
    time_t     st_mtime;    /* Time of last modification */
    time_t     st_ctime;    /* Time of last change */
};
```

文件元数据是指文件本身的一些信息，包含：访问模式、大小和创建时间：

宏指令	描述
S_ISREG()	这是一个普通文件吗？
S_ISDIR()	这是一个目录文件吗？
S_ISSOCK()	这是一个网络套接字吗？

我们只讲解其中的st_mode和st_size字段，其中模式设定中包含三种文件类型：

```
GNU nano 2.2.6      File: statcheck.c

#include "csapp.h"

int main(int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

    Stat(argv[1], &stat);
    if(S_ISREG(stat.st_mode))
        type = "regular";
    else if(S_ISDIR(stat.st_mode))
        type = "directory";
    else
        type = "other";

    if((stat.st_mode & S_IRUSR))
        readok = "yes";
    else
        readok = "no";

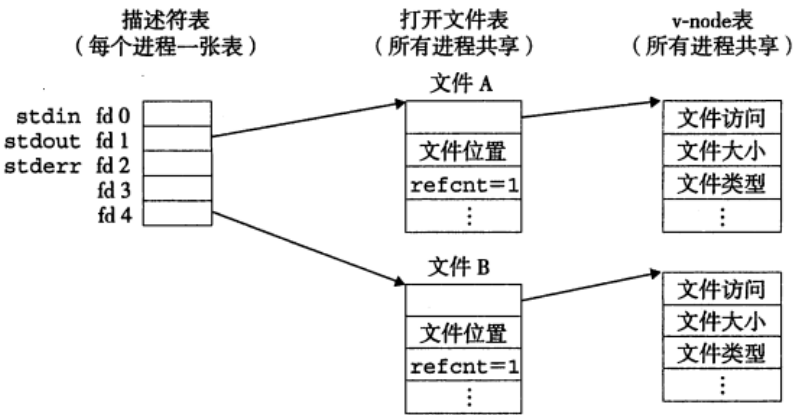
    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

我们来写一个应用程序，展示文件的读取模式：

运行结果如下，我们读取根目录元信息：

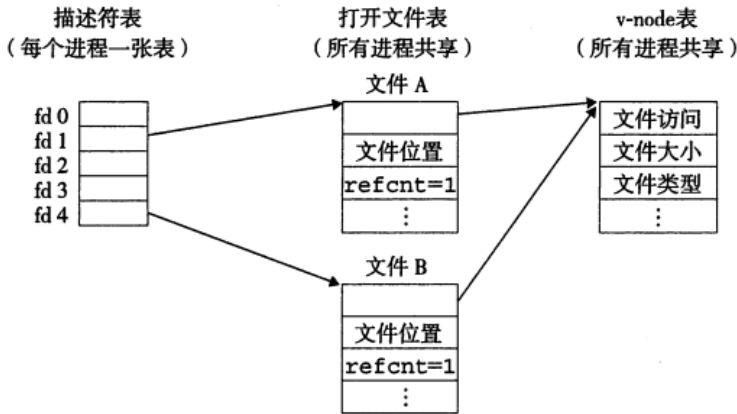
```
pi@raspberrypi:~/code $ ./statcheck ./
type: directory, read: yes
pi@raspberrypi:~/code $
```

1.5 共享文件



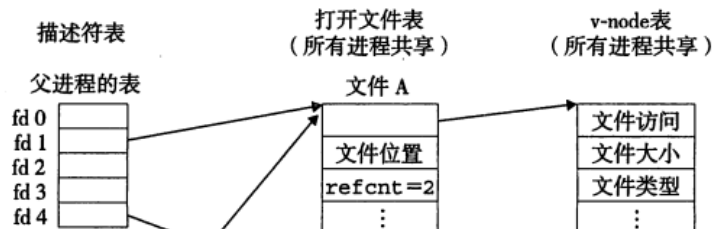
不理解文件是如何打开的，理解共享都是耍流氓。内核通过三个数据结构表示打开的文件：

- 描述符表：每个独立的进程1张，指向一打开的文件表；
- 文件表：包括打开文件位置，引用数量，以及一个指向元数据的v-node指针；
- v-node表：包含stat结构的大部分信息；



共享文件：

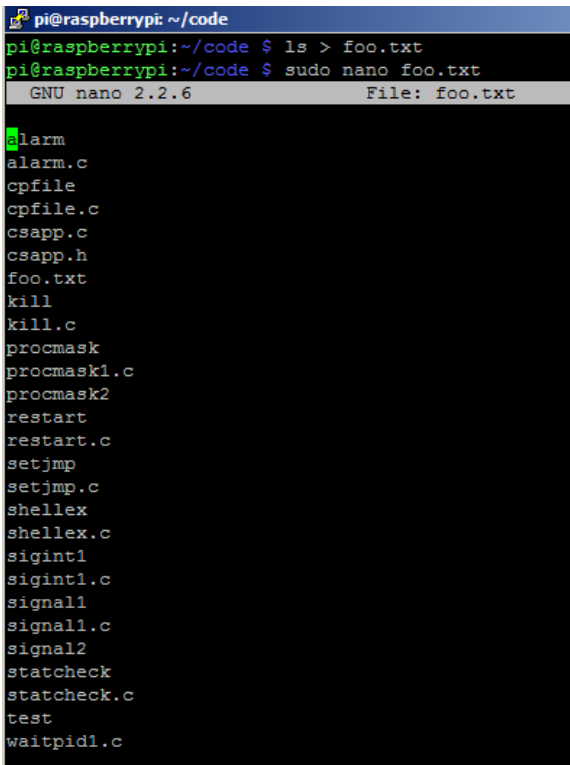
同一个进程的不同表项，通过文件表指向了同一个位置



理解父子进程共享文件：

子进程只需要将它的描述符表指向，文件表中同样的位置就行了。

1.6 IO重定向



重定向允许我们把本来输出到终端的内容，从新定位到磁盘文本中去，效果如下图：

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

返回：若成功则为非负的描述符，若出错则为 -1。

将当前目录列表，从终端重定位到foo.txt文本中，这个过程使用的是dup2函数

描述符表
(每个进程一张表)

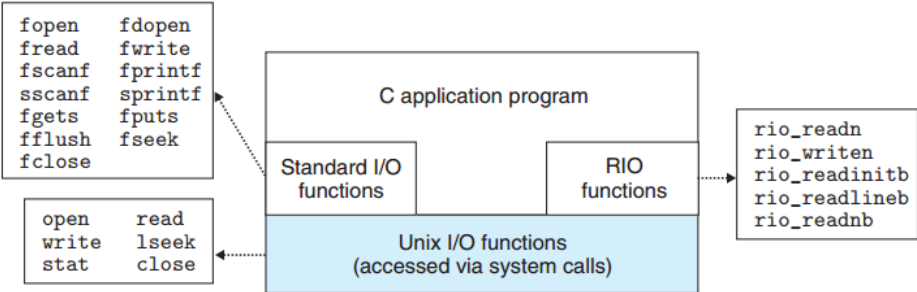
打开文件表
(所有进程共享)

v-node表
(所有进程共享)

将新的fd加到老的fd上面，删除掉newfd以前的内容，如果newfd已打开还会被关闭。

1.7 总结：什么时候用什么

赞赏



我们这一章讨论了标准IO函数、各种IO包，以及系统级的IO。他们之间的关系可以用下图来表示：

建议：

Unix IO 读取文件元信息

标准IO：在磁盘和终端中输入输出

RIO：网络套接字首选，如果要格式化先调用sprintf再调用rio

阅读 15,043

三生三世枕上书 续 8. 我疼你
阅读 5,613

枕上书续(十八)你我算是门当户对
阅读 8,593

女人要永远有养活自己的能力
阅读 7,076

枕上书创意篇
阅读 2,096



2017年5月2日 完

👍 4人点赞 >

👎

📖 《深入理解计算机系统》

⋮

"小礼物走一走，来简书关注我"

赞赏支持

还没有人赞赏，支持一下



唐鱼的学习探索 如果我像一般人一样读那么多书，我就跟他们一样愚蠢了。



写下你的评论...


评论0

赞4



ExpressVPN® Best-Rated VPN

With a VPN You Can Surf the Internet with No Censorship. Blazing-Fast Speeds!







写下你的评论...

全部评论 0

只看作者

按时间倒序 按时间正序

被以下专题收入，发现更多相似内容


-  《深入理解计算...
-  程序员
-  操作系统
-  其他

推荐阅读

更多精彩内容>

聊聊Linux 五种IO模型

上一篇《聊聊同步、异步、阻塞与非阻塞》已经通俗的讲解了，要理解同步、异步、阻塞与非阻塞重要的两个概念点了，没有看过...

 猿码架构

阅读 74,947

评论 64

赞 326

linux文件复用

本文摘抄自linux基础编程 IO概念 Linux的内核将所有外部设备都可以看做一个文件来操作。那么我们对与外部设...

 lintong

阅读 625

评论 0

赞 3

linux 异步非异步阻塞非阻塞(转载)

转自: <http://www.jianshu.com/p/486b0965c296> <http://www.jia...>

 demop

阅读 1,773


评论 1

赞 18

	Blocking	Non-blocking
synchronous	Read/write	Read/write (O_NONBLOCK)
asynchronous	I/O multiplexing (select/poll)	AIO

linux资料总章

linux资料总章2.1 1.0写的不好抱歉 但是2.0已经改了很多 但是错误还是无法避免 以后资料会慢慢更新 大...

 O感悟人生O

阅读 7,981

评论 2

赞 27

盈盈风荷举

和尘的花儿 [002] 荷的夏，开始渐行渐远。有点惋惜，些许不舍。最后，稍事整理，盖上印章。差点忘了枝干上的小刺...

 和尘之尘

阅读 741

评论 26

赞 53

