

**SEAWAY Edge**

**API文档**

中科海微（北京）科技有限公司

Seaway Technologies Co. Ltd

## **目录**

[目录 1](#_Toc1179293470)

[文档修订目录 1](#_Toc1258772127)

[文档版本 1](#_Toc1321731892)

[Version3.1 1](#_Toc777064276)

[第一章 字符测试篇 3](#_Toc1271229325)

[1.1isalnum 3](#_Toc236461272)

[1.2 isalpha 4](#_Toc1359851554)

[1.3 isascii 5](#_Toc1504096704)

[1.4 iscntrl 7](#_Toc1323295291)

[1.5 isdigit 8](#_Toc1283307505)

[1.6 isgraph 9](#_Toc1370969714)

[1.7 islower 10](#_Toc1535934535)

[1.8 isprint 11](#_Toc1698292805)

[1.9 isspace 12](#_Toc1002021358)

[1.10 ispunct 14](#_Toc406204132)

[1.11 isupper 15](#_Toc222332711)

[1.12 isxdigit 16](#_Toc124327997)

[第二章 字符串转换篇 17](#_Toc79057048)

[2.1 atof 17](#_Toc1566911890)

[2.2 atoi 18](#_Toc496172069)

[2.3 atol 20](#_Toc484962382)

[2.4 gcvt 21](#_Toc1062313909)

[2.5 strtod 22](#_Toc130827405)

[2.6 strtol 23](#_Toc1940424954)

[2.7 strtoul 25](#_Toc1035538536)

[2.8 toascii 26](#_Toc1088699264)

[2.9 tolower 27](#_Toc1207857608)

[2.10 toupper 28](#_Toc299902565)

[第三章 内存控制篇 29](#_Toc318290446)

[3.1 calloc 29](#_Toc125761245)

[3.2 free 31](#_Toc545336067)

[3.3 getpagesize 32](#_Toc3072673)

[3.4 malloc 33](#_Toc102807583)

[3.5 mmap 34](#_Toc1310195293)

[3.6 munmap 36](#_Toc154973113)

[第四章 日期时间篇 37](#_Toc1882930027)

[4.1 asctime 37](#_Toc1085941597)

[4.2 ctime 39](#_Toc2104388573)

[4.3 gettimeofday 40](#_Toc1550563968)

[4.4 gmtime 41](#_Toc614553831)

[4.5 localtime 43](#_Toc1557379194)

[4.6 mktime 44](#_Toc1341423922)

[4.7 settimeofday 45](#_Toc1028530848)

[4.8 time 46](#_Toc1422087633)

[第五章 内存及字符串操作篇 47](#_Toc1681340368)

[5.1 bcmp 47](#_Toc1697737750)

[5.2 bcopy 48](#_Toc263146561)

[5.3 bzero 49](#_Toc1035421554)

[5.4 index 50](#_Toc1270066437)

[5.6 memccpy 51](#_Toc19155479)

[5.6 memchr 52](#_Toc1971072150)

[5.7 memcmp 53](#_Toc726886428)

[5.8 memcpy 54](#_Toc1893211260)

[5.9 memmove 56](#_Toc2083932868)

[5.10 memset 58](#_Toc1348913553)

[5.11 rindex 59](#_Toc205223892)

[5.12 strcasecmp 60](#_Toc339215762)

[5.13 strcat 61](#_Toc1777712796)

[5.14 strchr 62](#_Toc78981661)

[5.15 strcmp 63](#_Toc299882581)

[5.16 strcoll 64](#_Toc2129903005)

[5.17 strcpy 65](#_Toc874893192)

[5.18 strcspn 67](#_Toc509346935)

[5.19 strdup 68](#_Toc724119603)

[5.20 strlen 69](#_Toc488340072)

[5.21 strncasecmp 70](#_Toc1996574917)

[5.22 strncat 71](#_Toc2002645644)

[5.23 strncpy 72](#_Toc954139277)

[5.24 strpbrk 73](#_Toc958436390)

[5.25 strrchr 74](#_Toc165570583)

[5.26 strspn 75](#_Toc1753465616)

[5.27 strstr 76](#_Toc578520331)

[5.28 strtok 77](#_Toc1532733148)

[第六章 常用数学函数篇 79](#_Toc1579672671)

[6.1 abs 79](#_Toc218253636)

[6.2 acos 80](#_Toc286791176)

[6.3 asin 81](#_Toc1145991164)

[6.4 atan 82](#_Toc2040147052)

[6.5 atan2 84](#_Toc2027594962)

[6.6 ceil 85](#_Toc1518015738)

[6.7 cos 87](#_Toc1184782206)

[6.8 cosh 88](#_Toc1166161258)

[6.9 exp 89](#_Toc1736500684)

[6.10 frexp 90](#_Toc1064233258)

[6.11 ldexp 92](#_Toc177071343)

[6.12 log 93](#_Toc1773210706)

[6.13 log10 95](#_Toc1721766323)

[6.14 pow 96](#_Toc384447336)

[6.15 sin 98](#_Toc1775565976)

[6.16 sinh 99](#_Toc504599920)

[6.17 sqrt 100](#_Toc397933437)

[6.18 acos 102](#_Toc803198901)

[6.19 tanh 103](#_Toc281724065)

[第七章 用户组篇 104](#_Toc1882402467)

[7.1 endgrent 104](#_Toc809175265)

[7.2 endpwent 105](#_Toc1942226051)

[7.3 endutent 106](#_Toc1241804757)

[7.4 fgetgrent 106](#_Toc1766469353)

[7.5 fgetpwent 108](#_Toc88996096)

[7.6 getegid 110](#_Toc1108767160)

[7.7 geteuid 110](#_Toc1334053101)

[7.8 getgid 111](#_Toc1701193827)

[7.9 getgrent 112](#_Toc367374231)

[7.10 getgrgid 114](#_Toc443215292)

[7.11 getgrnam 115](#_Toc1646124848)

[7.12 getgroups 116](#_Toc388496035)

[7.13 getpw 118](#_Toc1102573365)

[7.14 getpwent 119](#_Toc314155592)

[7.15 getpwnam 121](#_Toc1498230418)

[7.16 getpwuid 122](#_Toc1512874251)

[7.17 getuid 123](#_Toc671156077)

[7.18 getutent 124](#_Toc1536072095)

[7.19 getutid 126](#_Toc1862780078)

[7.20 getutline 127](#_Toc1728164980)

[7.21 initgroups 128](#_Toc552493185)

[7.22 pututline 129](#_Toc33670667)

[7.23 seteuid 130](#_Toc1114701108)

[7.24 setfsgid 131](#_Toc134185728)

[7.25 setfsuid 132](#_Toc401701146)

[7.26 setgid 133](#_Toc1850058301)

[7.27 setgrent 134](#_Toc514139994)

[7.28 setgroups 135](#_Toc1824167277)

[7.29 setpwent 136](#_Toc1302879967)

[7.30 setregid 137](#_Toc1760340557)

[7.31 setreuid 138](#_Toc161536780)

[7.32 setuid 139](#_Toc529331652)

[7.33 setutent 140](#_Toc1599809290)

[7.34 utmpname 140](#_Toc1499476590)

[第八章 数据结构及算法篇 141](#_Toc982450585)

[8.1 crypt 141](#_Toc45220312)

[8.2 bsearch 142](#_Toc1956056393)

[8.3 lfind 144](#_Toc1760128875)

[8.4 lsearch 145](#_Toc898764700)

[8.5 qsort 147](#_Toc138339902)

[8.6 rand 148](#_Toc1501426860)

[8.7 srand 149](#_Toc1548383770)

[第九章 文件操作篇 150](#_Toc479188044)

[9.1 close 150](#_Toc649779258)

[9.2 creat 151](#_Toc885644211)

[9.3 dup 153](#_Toc813096920)

[9.4 dup2 153](#_Toc1281488579)

[9.5 fcntl 154](#_Toc865051490)

[9.6 flock 156](#_Toc456102240)

[9.7 fsync 157](#_Toc1341211537)

[9.8 lseek 157](#_Toc1753943447)

[9.9 mkstemp 158](#_Toc19491360)

[9.10 open 160](#_Toc1173773176)

[9.11 read 162](#_Toc820987690)

[9.12 sync 163](#_Toc757673855)

[9.13 write 164](#_Toc1793937922)

[第十章 文件内容操作篇 165](#_Toc44251174)

[10.1 clearerr 165](#_Toc700139556)

[10.2 fclose 166](#_Toc1182615779)

[10.3 fdopen 168](#_Toc1262244668)

[10.4 feof 169](#_Toc1702670010)

[10.5 fflush 170](#_Toc1555261795)

[10.6 fgetc 171](#_Toc114037281)

[10.7 fgets 172](#_Toc1069168643)

[10.8 fileno 173](#_Toc1521708452)

[10.9 fopen 174](#_Toc971200641)

[10.10 fputc 175](#_Toc2093456087)

[10.11 fputs 177](#_Toc344381761)

[10.12 fread 178](#_Toc555828462)

[10.13 freopen 179](#_Toc255096384)

[10.14 fseek 180](#_Toc1027566476)

[10.15 ftell 182](#_Toc246272958)

[10.16 fwrite 184](#_Toc908617337)

[10.17 fclose 185](#_Toc375369142)

[10.18 getchar 186](#_Toc1669698355)

[10.19 gets 188](#_Toc1451437136)

[10.20 mktemp 189](#_Toc1037198479)

[10.21 putc 190](#_Toc1070073854)

[10.22 putchar 191](#_Toc1703204200)

[10.23 rewind 192](#_Toc1943458537)

[10.24 setbuf 194](#_Toc481360489)

[10.25 setbuffer 195](#_Toc654840374)

[10.26 setlinebuf 195](#_Toc48474943)

[10.27 setvbuf 196](#_Toc822064788)

[10.29 ungetc 197](#_Toc1680590765)

[第十一章 进程操作篇 198](#_Toc1984062011)

[11.1 atexit 198](#_Toc4148261)

[11.2 execl 200](#_Toc1000345923)

[11.3 execlp 201](#_Toc164455498)

[11.4 execv 202](#_Toc192101197)

[11.5 execve 203](#_Toc976896538)

[11.6 execvp 204](#_Toc1187632851)

[11.7 exit 205](#_Toc1832311539)

[11.8 \_exit 206](#_Toc744537993)

[11.9 vfork 207](#_Toc62837282)

[11.10 getpgid 208](#_Toc1691727897)

[11.11 getpgrp 209](#_Toc187278599)

[11.12 getpid 210](#_Toc1527870538)

[11.13 getppid 211](#_Toc1458164987)

[11.14 getpriority 212](#_Toc295556945)

[11.15 nice 214](#_Toc295899104)

[11.16 on\_exit 214](#_Toc1751598123)

[11.17 setpgid 216](#_Toc1403820185)

[11.18 setpgrp 216](#_Toc1750503353)

[11.19 setpriority 217](#_Toc183889971)

[11.20 system 218](#_Toc409774564)

[11.21 wait 219](#_Toc101041219)

[11.22 waitpid 221](#_Toc1687686603)

[第十二章 格式化输入输出篇 222](#_Toc984727045)

[12.1 fprintf 222](#_Toc1798461533)

[12.2 fscanf 223](#_Toc910653606)

[12.3 printf 224](#_Toc239203873)

[12.4 scanf 226](#_Toc210106327)

[12.5 sprintf 227](#_Toc793922221)

[12.6 sscanf 228](#_Toc1134869536)

[12.7 vfprintf 229](#_Toc1950022545)

[12.8 vfscanf 230](#_Toc1280976948)

[12.9 vprintf 231](#_Toc856003861)

[12.10 vscanf 232](#_Toc863940574)

[12.11 vsprintf 233](#_Toc1112289851)

[12.12 vsscanf 234](#_Toc410378622)

[第十三章 文件权限控制篇 234](#_Toc1663509437)

[13.1 access 234](#_Toc513507366)

[13.2 alphasort 236](#_Toc1929006716)

[13.3 chdir 237](#_Toc255257053)

[13.4 chmod 238](#_Toc1580446712)

[13.5 chown 240](#_Toc342658841)

[13.6 chroot 241](#_Toc1663483080)

[13.7 closedir 242](#_Toc70525267)

[13.8 fchdir 243](#_Toc2054672972)

[13.9 fchmod 244](#_Toc1351596644)

[13.10 fchown 246](#_Toc202777742)

[13.11 fstat 247](#_Toc28962005)

[13.12 ftruncate 248](#_Toc1433113813)

[13.13 getcwd 249](#_Toc167270339)

[13.14 link 250](#_Toc256493650)

[13.15 lstat 251](#_Toc889096021)

[13.16 opendir 252](#_Toc845609121)

[13.17 readdir 253](#_Toc105720801)

[13.18 readlink 255](#_Toc880526338)

[13.19 remove 255](#_Toc696351289)

[13.20 rename 256](#_Toc1937721720)

[13.21 rewinddir 258](#_Toc699441285)

[13.22 seekdir 260](#_Toc184193317)

[13.23 stat 262](#_Toc1213143492)

[13.24 symlink 264](#_Toc1092925426)

[13.25 telldir 265](#_Toc1370001991)

[13.26 truncate 267](#_Toc303799603)

[13.27 umask 267](#_Toc1391298702)

[13.28 unlink 268](#_Toc1755335978)

[13.29 utime 269](#_Toc1948923407)

[13.30 utimes 270](#_Toc2135117405)

[第十四章 信号处理篇 271](#_Toc466484465)

[14.1 alarm 271](#_Toc1889091705)

[14.2 kill 272](#_Toc1566048687)

[14.3 pause 274](#_Toc1020704777)

[14.4 sigaction 275](#_Toc885814803)

[14.5 sigaddset 277](#_Toc1532753017)

[14.6 sigdelset 278](#_Toc1913610954)

[14.7 sigemptyset 278](#_Toc1344206406)

[14.8 sigfillset 279](#_Toc549099202)

[14.9 sigismember 280](#_Toc973056855)

[14.10 signal 281](#_Toc1078590080)

[14.11 sigpending 282](#_Toc954010233)

[14.12 sigprocmask 282](#_Toc937077529)

[14.13 sleep 283](#_Toc1964446452)

[14.14 ferror 284](#_Toc1037929786)

[14.15 perror 285](#_Toc476248721)

[14.16 strerror 286](#_Toc640701478)

[14.17 mkfifo 287](#_Toc786734688)

[14.18 pclose 289](#_Toc593086637)

[14.19 pipe 290](#_Toc1535502332)

[14.20 popen 291](#_Toc876707925)

[第十五章 接口处理篇 293](#_Toc944793408)

[15.1 accept 293](#_Toc648722338)

[15.2 bind 294](#_Toc301858947)

[15.3 connect 295](#_Toc986948015)

[15.4 endprotoent 298](#_Toc471598677)

[15.5 endservent 298](#_Toc1944306909)

[15.6 getsockopt 299](#_Toc1855046811)

[15.7 htonl 301](#_Toc604165331)

[15.8 htons 301](#_Toc904035101)

[15.9 inet\_addr 302](#_Toc671139982)

[15.10 inet\_aton 303](#_Toc1265563430)

[15.11 inet\_ntoa 304](#_Toc1646528122)

[15.12 listen 304](#_Toc723871212)

[15.13 ntohl 307](#_Toc608599829)

[15.14 ntohs 308](#_Toc272715342)

[15.15 recv 308](#_Toc796650296)

[15.16 recvfrom 309](#_Toc1888469474)

[15.17 recvmsg 311](#_Toc1845630505)

[15.18 send 312](#_Toc1258100267)

[15.19 sendmsg 313](#_Toc767199107)

[15.20 sendto 314](#_Toc823574761)

[15.21 setprotoent 316](#_Toc1288903212)

[15.22 setservent 317](#_Toc928736795)

[15.23 setsockopt 317](#_Toc1368167169)

[15.24 shutdown 319](#_Toc1678200954)

[15.25 socket 319](#_Toc473214180)

[第十六章 环境变量篇 321](#_Toc1178778419)

[16.1 getenv 321](#_Toc1192244558)

[16.2 putenv 322](#_Toc2031859796)

[16.3 setenv 323](#_Toc182636778)

[第十七章 终端控制篇 325](#_Toc822196283)

[17.1 getopt 325](#_Toc1743143583)

[17.2 isatty 326](#_Toc1042287107)

[17.3 select 327](#_Toc695298770)

[17.4 ttyname 328](#_Toc1427904063)

# 文档修订目录

## 文档版本

## Version3.1

|  |  |  |  |
| --- | --- | --- | --- |
| 文档版本号 | 修订日期 | 修订内容 | 修订人 |
| V3.1 | 2023/04/29 | 初订 | 陈海鑫 |

**前言**

在使用本手册之前，请您认真阅读以下使用许可协议，只有在同意以下使用许可协议的情况下方能使用本手册中介绍的产品。

**版权声明**

中科海微（北京）科技有限公司版权所有，并保留对本文档及本声明的最终解释权和修改权。本文档中出现的任何文字叙述、文档格式、插图、照片、方法、过程等内容，除另有特别注明外，其著作权或其他相关权利均属于中科海微（北京）科技有限公司。未经中科海微（北京）科技有限公司书面同意，任何人不得以任何方式或形式对本手册内的任何部分进行复制、摘录、备份、修改、传播、翻译成其它语言、将其全部或部分用于商业用途**。**

**免责条款**

本文档依据现有信息制作，其内容如有更改，恕不另行通知。中科海微（北京）科技有限公司在编写该文档的时候已尽最大努力保证其内容准确可靠，但中科海微（北京）科技有限公司不对本文档中的遗漏、不准确、或错误导致的损失和损害承担责任。

**技术支持与信息反馈**

如果您在使用我们的产品时遇到问题,或者您认为我们的产品有某些功能缺陷,请访问我们的官网www.haiwei.tech联系我们的客服,我们将为您解决问题和反馈;或者需要技术支持指导以及有任何宝贵意见，也请您通过官网或者电话联系我们：

**API文档**

本文档主要适用于使用Linux的用户。用户需要具备以下经验和技能：

* 熟悉Linux基本操作。
* 对Linux有一定了解

# 字符测试篇

## 1.1isalnum

​测试字符是否为英文或数字。

* 相关函数

isalpha，isdigit，islower，isupper

* 表头文件

#include<ctype.h>

* 定义函数

int isalnum(int c)

* 函数说明

检查参数c是否为英文字母或阿拉伯数字，在标准c中相当于使用“isalpha(c) || isdigit(c)”做测试。在默认本地环境中，下列字符为字母数字：数字（ 0123456789 ），大写字母（ ABCDEFGHIJKLMNOPQRSTUVWXYZ ），小写字母（ abcdefghijklmnopqrstuvwxyz ）。若 c 的值不能表示为 unsigned char 且不等于 EOF ，则行为未定义。

* 返回值

若参数c为字母或数字，则返回TRUE，否则返回NULL(0)。

* 附加说明

此为宏定义，非真正函数。

* 范例

/\* 找出str 字符串中为英文字母或数字的字符\*/#include < ctype.h>main(){char str[]=”123c@#FDsP[e?”;int i;for (i=0;str[i]!=0;i++ )if ( isalnum(str[i])) printf(“%c is an alphanumeric character\n”,str[i]);}

* 执行

1 is an apphabetic character2 is an apphabetic character3 is an apphabetic characterc is an apphabetic characterF is an apphabetic characterD is an apphabetic characters is an apphabetic characterP is an apphabetic charactere is an apphabetic character

## 1.2 isalpha

测试字符是否为英文字母。

* 相关函数

isalnum，islower，isupper

* 表头文件

#include<ctype.h>

* 定义函数

int isalnum(int c)

* 函数说明

检查参数c是否为英文字母，即是大写字母（ ABCDEFGHIJKLMNOPQRSTUVWXYZ ）或小写字母（ abcdefghijklmnopqrstuvwxyz ）。在标准c中相当于使用“isupper(c)||islower(c)”做测试。若 c 的值不能表示为 unsigned char 且不等于 EOF ，则行为未定义。

* 返回值

若参数c为英文字母，则返回TRUE，否则返回NULL(0)。

* 附加说明

此为宏定义，非真正函数。

* 范例

/\* 找出str 字符串中为英文字母的字符\*/#include <ctype.h>main(){char str[]=”123c@#FDsP[e?”;int i;for (i=0;str[i]!=0;i++)if(isalpha(str[i])) printf(“%c is an alphanumeric character\n”,str[i]);}

* 执行

c is an apphabetic characterF is an apphabetic characterD is an apphabetic characters is an apphabetic characterP is an apphabetic charactere is an apphabetic character

## 1.3 isascii

测试字符是否为ASCII 码字符。

* 相关函数

Iscntrl

* 表头文件

#include<ctype.h>

* 定义函数

int isascii(int c);

* 函数说明

检查参数c是否为ASCII码字符，也就是判断c的范围是否在0到127之间。

* 返回值

若参数c为ASCII码字符，则返回TRUE，否则返回NULL(0)。

* 附加说明

此为宏定义，非真正函数。

* 范例

/\* 判断int i是否具有对映的ASCII码字符\*/#include<ctype.h>main(){int i;for(i=125;i<130;i++)if(isascii(i))printf("%d is an ascii character:%c\n",i,i);elseprintf("%d is not an ascii character\n",i);}

* 执行

125 is an ascii character:}126 is an ascii character:~127 is an ascii character:128 is not an ascii character129 is not an ascii character

## 1.4 iscntrl

测试字符是否为ASCII 码的控制字符。

* 相关函数

isascii

* 表头文件

#include<ctype.h>

* 定义函数

int iscntrl(int c)；

* 函数说明

检查给定字符是否为控制字符，即编码 0x00-0x1F 及 0x7F 。若 c 的值不能表示为 unsigned char 且不等于 EOF ，则行为未定义。

* 返回值

若参数c为ASCII码字符，则返回TRUE，否则返回NULL(0)。

* 附加说明

此为宏定义，非真正函数。

* 范例

/\* 判断控制字符\*/#include <stdio.h>

#include <ctype.h>

#include <locale.h>

int main(void)

{

unsigned char c = '\x94'; // ISO-8859-1 的控制码 CCH

printf("In the default C locale, \\x94 is %sa control character\n",iscntrl(c) ? "" : "not " );

setlocale(LC\_ALL, "en\_GB.iso88591");

printf("In ISO-8859-1 locale, \\x94 is %sa control character\n",iscntrl(c) ? "" : "not " );

}

* 执行

In the default C locale, \x94 is not a control character

In ISO-8859-1 locale, \x94 is a control character

## 1.5 isdigit

测试字符是否为阿拉伯数字。

* 相关函数

Iscntrl

* 表头文件

#include<ctype.h>

* 定义函数

int isdigit(int c)

* 函数说明

检查参数c是否为阿拉伯数字0到9。若 c 的值不能表示为 unsigned char 且不等于 EOF ，则行为未定义。

* 返回值

若参数c为阿拉伯数字，则返回TRUE，否则返回NULL(0)。

* 附加说明

此为宏定义，非真正函数。isdigit 与 isxdigit 是仅有的不受当前安装的 C 本地环境影响的标准窄字符分类函数，尽管一些实现可能将额外的单字节字符分类为数字。

* 范例

/\* 找出str字符串中为阿拉伯数字的字符\*/#include<ctype.h>main(){char str[]="123@#FDsP[e?";int i;for(i=0;str[i]!=0;i++)if(isdigit(str[i])) printf("%c is an digit character\n",str[i]);}

* 执行

1 is an digit character2 is an digit character3 is an digit character

## 1.6 isgraph

测试字符是否为可打印字符。

* 相关函数

isprint

* 表头文件

#include<ctype.h>

* 定义函数

int isgraph(int c)

* 函数说明

检查参数c是否为可打印字符，若c所对映的ASCII码可打印，且非空格字符则返回TRUE。

* 返回值

若参数c为可打印字符，则返回TRUE，否则返回NULL(0)。

* 附加说明

此为宏定义，非真正函数。

* 范例

/\* 判断str字符串中哪些为可打印字符\*/#include<ctype.h>main(){char str[]="a5 @;";int i;for(i=0;str[i]!=0;i++)if(isgraph(str[i])) printf("str[%d] is printable character:%d\n",i,str[i]);}

* 执行

str[0] is printable character:astr[1] is printable character:5str[3] is printable character:@str[4] is printable character:;

## 1.7 islower

测试字符是否为小写字母。

* 相关函数

isalpha，isupper

* 表头文件

#include<ctype.h>

* 定义函数

int islower(int c)

* 函数说明

检查参数c是否为小写英文字母。默认 "C" 本地环境中， islower 仅对小写字母（ abcdefghijklmnopqrstuvwxyz ）返回非零值。若 islower 返回非零值，则保证同一 C 本地环境中 iscntrl 、 isdigit 、 ispunct 和 isspace 对同一字符返回零。若 c 不可表示为 unsigned char 且不等于 EOF 则行为未定义。

* 返回值

若参数c为小写英文字母，则返回TRUE，否则返回NULL(0)。

* 附加说明

此为宏定义，非真正函数。

* 范例

#include<ctype.h>main(){char str[]="123@#FDsP[e?";int i;for(i=0;str[i]!=0;i++)if(islower(str[i])) printf("%c is a lower-case character\n",str[i]);}

* 执行

c is a lower-case characters is a lower-case charactere is a lower-case character

## 1.8 isprint

测试字符是否为可打印字符。

* 相关函数

Isgraph

* 表头文件

#include<ctype.h>

* 定义函数

int isprint(int c);

* 函数说明

检查参数c是否为可打印字符，即为数字（ 0123456789 ）、大写字母（ ABCDEFGHIJKLMNOPQRSTUVWXYZ ）、小写字母（ abcdefghijklmnopqrstuvwxyz ）、标点字符（ !"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~ ）或空格之一，或任何当前 C 本地环境分类为可打印的字符。若c所对映的ASCII码可打印，其中包含空格字符，则返回TRUE。若 c 的值不能表示为 unsigned char 且不等于 EOF ，则行为未定义。

* 返回值

若参数c为可打印字符，则返回TRUE，否则返回NULL(0)。

* 附加说明

此为宏定义，非真正函数。

* 范例

/\* 判断str字符串中哪些为可打印字符包含空格字符\*/#include<ctype.h>main(){char str[]="a5 @;";int i;for(i=0;str[i]!=0;i++)if(isprint(str[i])) printf("str[%d] is printable character:%d\n",i,str[i]);}

* 执行

str[0] is printable character:astr[1] is printable character:5str[2] is printable character:str[3] is printable character:@str[4] is printable character:;

## 1.9 isspace

测试字符是否为空格字符。

* 相关函数

Isgraph

* 表头文件

#include<ctype.h>

* 定义函数

int isspace(int c)

* 函数说明

检查参数c是否为空白字符，即空格（ 0x20 ）、换行（ 0x0a ）、回车（ 0x0d ）、水平制表符（ 0x09 ）或垂直制表符（ 0x0b ）之一，也就是判断是否为空格('')、定位字符('\t')、CR('\r')、换行('\n')、垂直定位字符('\v')或翻页('\f')的情况。若 c 的值不能表示为 unsigned char 且不等于 EOF ，则行为未定义。

* 返回值

若参数c为空格字符，则返回TRUE，否则返回NULL(0)。

* 附加说明

此为宏定义，非真正函数。

* 范例

/\*将字符串str[]中内含的空格字符找出，并显示空格字符的ASCII码\*/#include <ctype.h>main(){char str="123c @# FD\tsP[e?\n";int i;for(i=0;str[i]!=0;i++)if(isspace(str[i]))printf("str[%d] is a white-space character:%d\n",i,str[i]);}

* 执行

str[4] is a white-space character:32str[7] is a white-space character:32str[10] is a white-space character:9 /\* \t \*/str[16] is a white-space character:10 /\* \t \*/

## 1.10 ispunct

测试字符是否为标点符号或特殊符号。

* 相关函数

isspace，isdigit，isalpha

* 表头文件

#include<ctype.h>

* 定义函数

int ispunct(int c)

* 函数说明

检查参数c是否为当前 C 本地环境分类为标点字符，默认 C 本地环境分类字符 !"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~ 为标点。返回TRUE也就是代表参数c为非空格、非数字和非英文字母。若 c 的值不能表示为 unsigned char 且不等于 EOF ，则行为未定义。

* 返回值

若参数c为标点符号或特殊符号，则返回TRUE，否则返回NULL(0)。

* 附加说明

此为宏定义，非真正函数。

* 范例

/\*列出字符串str中的标点符号或特殊符号\*/#include <ctype.h>main(){char str[]="123c@ #FDsP[e?";int i;for(i=0;str[i]!=0;i++)if(ispunct(str[i])) printf("%c\n",str[i]);}

* 执行

@#[?

## 1.11 isupper

测试字符是否为大写英文字母。

* 相关函数

isalpha，islower

* 表头文件

#include<ctype.h>

* 定义函数

int isupper(int c)

* 函数说明

检查参数c是否为大写英文字母。根据当前C本地环境检查给定字符是否大写字母。在默认 "C" 本地环境中， isupper 仅对大写拉丁字母（ ABCDEFGHIJKLMNOPQRSTUVWXYZ ）返回非零值。若 isupper返回非零，则保证 iscntrl 、 isdigit、 ispunct 及isspace 在同一 C 本地环境中对同一字符返回零。若 c 的值不能表示为 unsigned char 且不等于 EOF ，则行为未定义。

* 返回值

若参数c为大写英文字母，则返回TRUE，否则返回NULL(0)。

* 附加说明

此为宏定义，非真正函数。

* 范例

/\*找出字符串str中为大写英文字母的字符\*/#include <ctype.h>main(){char str[]="123c@#FDsP[e?";int i;for(i=0;str[i]!=0;i++)if(isupper(str[i])) printf("%c is an uppercase character\n",str[i]);}

* 执行

F is an uppercase characterD is an uppercase characterP is an uppercase character

## 1.12 isxdigit

测试字符是否为16进制数字。

* 相关函数

isalnum，isdigit

* 表头文件

#include<ctype.h>

* 定义函数

int isxdigit(int c)

* 函数说明

检查给定的字符是否十六进制数字字符（ 0123456789abcdefABCDEF ）或被分类为十六进制数字字符。若 c 的值不能表示为 unsigned char 且不等于 EOF ，则行为未定义。

* 返回值

若参数c为16进制数字，则返回TRUE，否则返回NULL(0)。

* 附加说明

此为宏定义，非真正函数。isdigit 和 isxdigit 是仅有的不受当前安装的 C 本地环境影响的标准窄字符分类函数。尽管某些实现可能分类另外的单字节字符为数字。

* 范例

/\*找出字符串str中为十六进制数字的字符\*/#include <ctype.h>main(){char str[]="123c@#FDsP[e?";int i;for(i=0;str[i]!=0;i++)if(isxdigit(str[i])) printf("%c is a hexadecimal digits\n",str[i]);}

* 执行

1 is a hexadecimal digits2 is a hexadecimal digits3 is a hexadecimal digitsc is a hexadecimal digitsF is a hexadecimal digitsD is a hexadecimal digitse is a hexadecimal digits

# 第二章 字符串转换篇

## 2.1 atof

​将字符串转换成浮点型数。

* 相关函数

atoi，atol，strtod，strtol，strtoul

* 表头文件

#include <stdlib.h>

* 定义函数

double atof(const char \*nptr);

* 函数说明

函数会舍弃任何空白符（由 std::isspace() 确定），直至找到首个非空白符。然后它会取用尽可能多的字符，以构成合法的浮点数表示，并将它们转换成浮点值。合法的浮点值可以为下列之一：十进制浮点数表达式。十六进制浮点数表达式。无穷大表达式。非数（NaN）表达式。任何其他可由当前 C 本地环境接受的表达式。

参数nptr字符串可包含正负号、小数点或E(e)来表示指数部分，如123.456或123e-2。

* 返回值

成功时返回代表 str 内容的 double 值。若转换的值在返回值范围外，则返回值未定义。若无可进行的转换，则返回 0.0 。

* 附加说明

atof()与使用strtod(nptr,(char\*\*)NULL)结果相同。

* 范例

/\* 将字符串a 与字符串b转换成数字后相加\*/#include<stdlib.h>main(){char \*a=”-100.23”;char \*b=”200e-2”;float c;c=atof(a)+atof(b);printf(“c=%.2f\n”,c);}

* 执行

c=-98.23

## 2.2 atoi

​将字符串转换成整型数。

* 相关函数

atof，atol，atrtod，strtol，strtoul

* 表头文件

#include <stdlib.h>

* 定义函数

int atoi(const char \*nptr);

* 函数说明

atoi()会扫描参数nptr字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时('\0')才结束转换，并将结果返回。

* 返回值

成功时为对应 nptr 内容的整数值。若转换的值落在对应的返回类型范围外，则返回值未定义。若无法进行转换，则返回 ​0​ 。

* 附加说明

atoi()与使用strtol(nptr，(char\*\*)NULL，10)；结果相同。

* 范例

/\* 将字符串a 与字符串b转换成数字后相加\*/#include<stdlib.h>mian(){char a[]=”-100”;char b[]=”456”;int c;c=atoi(a)+atoi(b);printf(c=%d\n”,c);}

* 执行

c=356

## 2.3 atol

​将字符串转换成长整型数。

* 相关函数

atof，atoi，strtod，strtol，strtoul

* 表头文件

#include <stdlib.h>

* 定义函数

long atol(const char \*nptr);

* 函数说明

atol()会扫描参数nptr字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时('\0')才结束转换，并将结果返回。

* 返回值

成功时为对应 nptr内容的长整数值。若转换的值落在对应的返回类型范围外，则返回值未定义。若无法进行转换，则返回 ​0​ 。

* 附加说明

atol()与使用strtol(nptr,(char\*\*)NULL,10)；结果相同。

* 范例

/\*将字符串a与字符串b转换成数字后相加\*/#include<stdlib.h>main(){char a[]=”1000000000”;char b[]=” 234567890”;long c;c=atol(a)+atol(b);printf(“c=%d\n”,c);}

* 执行

c=1234567890

## 2.4 gcvt

​将浮点型数转换为字符串，取四舍五入。

* 相关函数

ecvt，fcvt，sprintf

* 表头文件

#include <stdlib.h>

* 定义函数

char \*gcvt(double number，size\_t ndigits，char \*buf);

* 函数说明

gcvt()用来将参数number转换成ASCII码字符串，参数ndigits表示显示的位数。gcvt()与ecvt()和fcvt()不同的地方在于，gcvt()所转换后的字符串包含小数点或正负符号。若转换成功，转换后的字符串会放在参数buf指针所指的空间。

* 返回值

返回一字符串指针，此地址即为buf指针。

* 范例

#include<stdlib.h>main(){double a=123.45;double b=-1234.56;char \*ptr;int decpt,sign;gcvt(a,5,ptr);printf(“a value=%s\n”,ptr);ptr=gcvt(b,6,ptr);printf(“b value=%s\n”,ptr);}

* 执行

a value=123.45b value=-1234.56

## 2.5 strtod

​将字符串转换成浮点数。

* 相关函数

atoi，atol，strtod，strtol，strtoul

* 表头文件

#include <stdlib.h>

* 定义函数

double strtod(const char \*nptr,char \*\*endptr);

* 函数说明

strtod()会扫描参数nptr字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，到出现非数字或字符串结束时('\0')才结束转换，并将结果返回。若endptr不为NULL，则会将遇到不合条件而终止的nptr中的字符指针由endptr传回。参数nptr字符串可包含正负号、小数点或E(e)来表示指数部分。如123.456或123e-2。

* 返回值

成功时为对应 nptr内容的浮点值。若转换出的值落在对应返回类型的范围外，则发生值域错误并返回 HUGE\_VAL 、 HUGE\_VALF 或 HUGE\_VALL 。若无法进行转换，则返回 ​0​ 并将 \*str\_end 设为 nptr。

* 附加说明

参考atof()。函数设置 str\_end 所指向的指针指向最后被转译字符的后一字符，若 str\_end 为空指针，则忽略它。

* 范例

/\*将字符串a，b，c 分别采用10，2，16 进制转换成数字\*/#include<stdlib.h>mian(){char a[]=”1000000000”;char b[]=”1000000000”;char c[]=”ffff”;printf(“a=%d\n”,strtod(a,NULL,10));printf(“b=%d\n”,strtod(b,NULL,2));printf(“c=%d\n”,strtod(c,NULL,16));}

* 执行

a=1000000000b=512c=65535

## 2.6 strtol

​将字符串转换成长整型数。

* 相关函数

atof，atoi，atol，strtod，strtoul

* 表头文件

#include <stdlib.h>

* 定义函数

long int strtol(const char \*nptr, char \*\*endptr, int base);

* 函数说明

strtol()会将参数nptr字符串根据参数base来转换成长整型数。参数base范围从2至36，或0。参数base代表采用的进制方式，如base值为10则采用10进制，若base值为16则采用16进制等。当base值为0时则是采用10进制做转换，但遇到如'0x'前置字符则会使用16进制做转换。一开始strtol()会扫描参数nptr字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，再遇到非数字或字符串结束时('\0')结束转换，并将结果返回。若参数endptr不为NULL，则会将遇到不合条件而终止的nptr中的字符指针由endptr返回。函数设置 str\_end 所指向的指针指向最后被转译字符的后一字符。若 str\_end 为空指针，则忽略它。若 str 为空或无期待的形式，则不进行转换，并（若 str\_end 不是空指针）将 nptr的值存储于 str\_end 所指的对象。

* 返回值

若成功，则返回对应 nptr内容的整数值。

若被转换值落在对应返回类型的范围外，则发生值域错误（设 errno 为 ERANGE ）并返回 LONG\_MAX 、 LONG\_MIN 、 LLONG\_MAX 或 LLONG\_MIN 。

若无法进行转换，则返回 ​0​ 。

* 附加说明

ERANGE指定的转换字符串超出合法范围。

* 范例

/\* 将字符串a，b，c 分别采用10，2，16进制转换成数字\*/#include<stdlib.h>main(){char a[]=”1000000000”;char b[]=”1000000000”;char c[]=”ffff”;printf(“a=%d\n”,strtol(a,NULL,10));printf(“b=%d\n”,strtol(b,NULL,2));printf(“c=%d\n”,strtol(c,NULL,16));}

* 执行

a=1000000000b=512c=65535

## 2.7 strtoul

​将字符串转换成无符号长整型数。

* 相关函数

atof，atoi，atol，strtod，strtol

* 表头文件

#include <stdlib.h>

* 定义函数

unsigned long int strtoul(const char \*nptr, char \*\*endptr, int base);

* 函数说明

strtoul()会将参数nptr字符串根据参数base来转换成无符号的长整型数。参数base范围从2至36，或0。参数base代表采用的进制方式，如base值为10则采用10进制，若base值为16则采用16进制数等。当base值为0时则是采用10进制做转换，但遇到如'0x'前置字符则会使用16进制做转换。一开始strtoul()会扫描参数nptr字符串，跳过前面的空格字符串，直到遇上数字或正负符号才开始做转换，再遇到非数字或字符串结束时('\0')结束转换，并将结果返回。若参数endptr不为NULL，则会将遇到不合条件而终止的nptr中的字符指针由endptr返回。

* 返回值

成功时为对应 nptr内容的整数值。若被转换值落在对应返回类型的范围外，则发生值域错误（ errno 被设为ERANGE ）并返回 ULONG\_MAX 或 ULLONG\_MAX 。若无转换可进行，则返回 ​0​ 。

* 附加说明

ERANGE指定的转换字符串超出合法范围。

* 范例

#include <stdio.h>

#include <errno.h>

#include <stdlib.h>

int main(void)

{

const char \*p = "10 200000000000000000000000000000 30 -40";

printf("Parsing '%s':\n", p);

char \*end;

for (unsigned long i = strtoul(p, &end, 10);

p != end;

i = strtoul(p, &end, 10))

{

printf("'%.\*s' -> ", (int)(end-p), p);

p = end;

if (errno == ERANGE){

printf("range error, got ");

errno = 0;

}

printf("%lu\n", i);

}

}

* 执行

Parsing '10 200000000000000000000000000000 30 -40':

'10' -> 10

' 200000000000000000000000000000' -> range error, got 18446744073709551615

' 30' -> 30

' -40' -> 18446744073709551576

## 2.8 toascii

​将整型数转换成合法的ASCII 码字符。

* 相关函数

isascii，toupper，tolower

* 表头文件

#include <ctype.h>

* 定义函数

int toascii(int c)

* 函数说明

toascii()会将参数c转换成7位的unsigned char值，第八位则会被清除，此字符即会被转成ASCII码字符。

* 返回值

将转换成功的ASCII码字符值返回。

* 范例

#include<stdlib.h>main(){int a=217;char b;printf(“before toascii () : a value =%d(%c)\n”,a,a);b=toascii(a);printf(“after toascii() : a value =%d(%c)\n”,b,b);}

* 执行

before toascii() : a value =217()after toascii() : a value =89(Y)

## 2.9 tolower

​将大写字母转换成小写字母。

* 相关函数

isalpha，toupper

* 表头文件

#include <ctype.h>

* 定义函数

int tolower(int c);

* 函数说明

按照当前安装的 C 本地环境所定义的规则，转换给定字符为小写。默认 "C" 本地环境中，以分别小写字母 ABCDEFGHIJKLMNOPQRSTUVWXYZ 替换大写字母abcdefghijklmnopqrstuvwxyz 。

* 返回值

c 的小写版本，或若当前 C 本地环境中无小写版本，则返回不修改的 c 。

* 范例

/\* 将s字符串内的大写字母转换成小写字母\*/#include<ctype.h>main(){char s[]=”aBcDeFgH12345;!#$”;int i;printf(“before tolower() : %s\n”,s);for(i=0;I<sizeof(s);i++)s[i]=tolower(s[i]);printf(“after tolower() : %s\n”,s);}

* 执行

before tolower() : aBcDeFgH12345;!#$after tolower() : abcdefgh12345;!#$

## 2.10 toupper

​将小写字母转换成大写字母。

* 相关函数

isalpha，tolower

* 表头文件

#include <ctype.h>

* 定义函数

int toupper(int c);

* 函数说明

按照当前安装的 C 本地环境所定义的字符转换规则，转换给定字符为大写。默认 "C" 本地环境中，分别以大写字母 ABCDEFGHIJKLMNOPQRSTUVWXYZ 替换下列小写字母 abcdefghijklmnopqrstuvwxyz 。

* 返回值

被转换的字符，或若无大写版本列于当前 C 本地环境，则为不修改的 c 。

* 范例

/\* 将s字符串内的小写字母转换成大写字母\*/#include<ctype.h>main(){char s[]=”aBcDeFgH12345;!#$”;int i;printf(“before toupper() : %s\n”,s);for(i=0;I<sizeof(s);i++)s[i]=toupper(s[i]);printf(“after toupper() : %s\n”,s);}

* 执行

before toupper() : aBcDeFgH12345;!#$after toupper() : ABCDEFGH12345;!#$

# 第三章 内存控制篇

## 3.1 calloc

​配置内存空间。

* 相关函数

malloc，free，realloc，brk

* 表头文件

#include <stdlib.h>

* 定义函数

void \*calloc(size\_t nmemb，size\_t size);

* 函数说明

为 nmemb个对象的数组分配内存，并初始化所有分配存储中的字节为零。若分配成功，会返回指向分配内存块最低位（首位）字节的指针，它为任何类型适当地对齐。

若 size 为零，则行为是实现定义的（可返回空指针，或返回不可用于访问存储的非空指针）。

C11起calloc 是线程安全的：它表现得如同只访问通过其参数可见的内存区域，而非任何静态存储。

令 free 或 realloc 解分配一块内存区域的先前调用，同步于令 calloc 分配相同或部分相同的内存区域的调用。这种同步出现于任何解分配函数所做的内存访问之后，和任何 calloc 所做的内存访问之前。所有操作每块特定内存区域的分配及解分配函数拥有单独全序。

* 返回值

成功时，返回指向新分配内存的指针。为避免内存泄漏，必须用 free() 或 realloc() 解分配返回的指针。失败时，返回空指针。因为对齐需求的缘故，分配的字节数不必等于 num\*size 。初始化所有位为零不保证浮点数或指针被各种初始化为 0.0 或空指针（尽管这在所有常见平台上为真）。

* 范例

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

int \*p1 = calloc(4, sizeof(int)); // 分配并清零 4 个 int 的数组

int \*p2 = calloc(1, sizeof(int[4])); // 等价，直接命名数组类型

int \*p3 = calloc(4, sizeof \*p3); // 等价，免去重复类型名

if(p2) {

for(int n=0; n<4; ++n) // 打印数组

printf("p2[%d] == %d\n", n, p2[n]);

}

free(p1);

free(p2);

free(p3);

}

* 执行

p2[0] == 0

p2[1] == 0

p2[2] == 0

p2[3] == 0

## 3.2 free

​释放原先配置的内存。

* 相关函数

malloc，calloc，realloc，brk

* 表头文件

#include <stdlib.h>

* 定义函数

void free(void \*ptr);

* 函数说明

解分配之前由 malloc() 、 calloc() 、 aligned\_alloc() (C11 起) 或 realloc() 分配的空间。

若 ptr 为空指针，则函数不进行操作。

若 ptr 的值不等于之前从 malloc() 、 calloc() 、 realloc() 或 aligned\_alloc() (C11 起) 返回的值，则行为未定义。

若 ptr 所指代的内存区域已经被解分配，则行为未定义，即是说已经以ptr 为参数调用 free() 或 realloc() ，而且没有后继的 malloc() 、 calloc() 或 realloc() 调用以 ptr 为结果。

若ptr为指向先前由malloc()、calloc()或realloc()所返回的内存指针。调用free()后ptr所指的内存空间便会被收回。

此函数接收空指针（并对其不处理）以减少特例的数量。不管分配成功与否，分配函数返回的指针都能传递给 free() 。

* 返回值

（无）

* 范例

#include <stdlib.h>

int main(void)

{

int \*p1 = malloc(10\*sizeof \*p1);

free(p1); // 每一个分配的指针都要释放

int \*p2 = calloc(10, sizeof \*p2);

int \*p3 = realloc(p2, 1000\*sizeof \*p3);

if(p3) // p3 非空表示 p2 被 realloc 释放

free(p3);

else // p3 为空表示 p2 未被释放

free(p2);

}

## 3.3 getpagesize

​取得内存分页大小。

* 相关函数

Sbrk

* 表头文件

#include <unistd.h>

* 定义函数

size\_t getpagesize(void);

* 函数说明

返回一分页的大小，单位为字节（byte）。此为系统的分页大小，不一定会和硬件分页大小相同。

* 返回值

内存分页大小。附加说明在Intel x86 上其返回值应为4096bytes。

* 范例

#include <unistd.h>main(){printf(“page size = %d\n”,getpagesize( ) );}

## 3.4 malloc

​配置内存空间。

* 相关函数

calloc，free，realloc，brk

* 表头文件

#include <stdlib.h>

* 定义函数

void \*malloc(size\_t size);

* 函数说明

分配 size 字节的未初始化内存。

若分配成功，则返回为任何拥有基础对齐的对象类型对齐的指针。

若 size 为零，则 malloc 的行为是实现定义的。例如可返回空指针。亦可返回非空指针；但不应当解引用这种指针，而且应将它传递给 free 以避免内存泄漏。

C11起malloc 是线程安全的：它表现得如同只访问通过其参数可见的内存区域，而非任何静态存储。

令 free 或 realloc 归还一块内存区域的先前调用，同步于令 malloc 分配相同或部分相同的内存区域的调用。此同步出现于任何通过解分配函数所作的内存访问后，和任何 malloc 所作的内存访问前。所有操作每块特定内存区域的分配和解分配函数有单独全序。

* 返回值

成功时，返回指向新分配内存的指针。为避免内存泄漏，必须用 free() 或 realloc() 解分配返回的指针。失败时，返回空指针。

* 范例

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

int \*p1 = malloc(4\*sizeof(int)); // 足以分配 4 个 int 的数组

int \*p2 = malloc(sizeof(int[4])); // 等价，直接命名数组类型

int \*p3 = malloc(4\*sizeof \*p3); // 等价，免去重复类型名

if(p1) {

for(int n=0; n<4; ++n) // 置入数组

p1[n] = n\*n;

for(int n=0; n<4; ++n) // 打印出来

printf("p1[%d] == %d\n", n, p1[n]);

}

free(p1);

free(p2);

free(p3);

}

* 执行

p1[0] == 0

p1[1] == 1

p1[2] == 4

p1[3] == 9

## 3.5 mmap

​建立内存映射。

* 相关函数

munmap，open

* 表头文件

#include <unistd.h>#include <sys/mman.h>

* 定义函数

void \*mmap(void \*start, size\_t length,int prot, int flags, int fd, off\_t offsize);

* 函数说明

mmap()用来将某个文件内容映射到内存中，对该内存区域的存取即是直接对该文件内容的读写。参数start指向欲对应的内存起始地址，通常设为NULL，代表让系统自动选定地址，对应成功后该地址会返回。参数length代表将文件中多大的部分对应到内存。

* 参数

prot代表映射区域的保护方式有下列组合 PROT\_EXEC 映射区域可被执行 PROT\_READ 映射区域可被读取 PROT\_WRITE 映射区域可被写入 PROT\_NONE 映射区域不能存取

* 参数

flags会影响映射区域的各种特性 MAP\_FIXED 如果参数start所指的地址无法成功建立映射时，则放弃映射，不对地址做修正。通常不鼓励用此旗标。 MAP\_SHARED对映射区域的写入数据会复制回文件内，而且允许其他映射该文件的进程共享。 MAP\_PRIVATE 对映射区域的写入操作会产生一个映射文件的复制，即私人的“写入时复制”（copy on write）对此区域作的任何修改都不会写回原来的文件内容。 MAP\_ANONYMOUS建立匿名映射。此时会忽略参数fd，不涉及文件，而且映射区域无法和其他进程共享。 MAP\_DENYWRITE只允许对映射区域的写入操作，其他对文件直接写入的操作将会被拒绝。 MAP\_LOCKED 将映射区域锁定住，这表示该区域不会被置换（swap）。 在调用mmap()时必须要指定MAP\_SHARED 或MAP\_PRIVATE。参数fd为open()返回的文件描述词，代表欲映射到内存的文件。参数offset为文件映射的偏移量，通常设置为0，代表从文件最前方开始对应，offset必须是分页大小的整数倍。

* 返回值

若映射成功则返回映射区的内存起始地址，否则返回MAP\_FAILED(－1)，错误原因存于errno 中。

* 错误代码

EBADF 参数fd 不是有效的文件描述词 EACCES 存取权限有误。如果是MAP\_PRIVATE 情况下文件必须可读，使用MAP\_SHARED则要有PROT\_WRITE以及该文件要能写入。 EINVAL 参数start、length 或offset有一个不合法。 EAGAIN 文件被锁住，或是有太多内存被锁住。 ENOMEM 内存不足。

* 范例

/\* 利用mmap()来读取/etc/passwd 文件内容\*/#include<sys/types.h>#include<sys/stat.h>#include<fcntl.h>#include<unistd.h>#include<sys/mman.h>main(){int fd;void \*start;struct stat sb;fd=open(“/etc/passwd”,O\_RDONLY); /\*打开/etc/passwd\*/fstat(fd,&sb); /\*取得文件大小\*/start=mmap(NULL,sb.st\_size,PROT\_READ,MAP\_PRIVATE,fd,0);if(start= = MAP\_FAILED) /\*判断是否映射成功\*/return;printf(“%s”,start);munma(start,sb.st\_size); /\*解除映射\*/closed(fd);}

* 执行

root : x : 0 : root : /root : /bin/bashbin : x : 1 : 1 : bin : /bin :daemon : x : 2 : 2 :daemon : /sbinadm : x : 3 : 4 : adm : /var/adm :lp : x :4 :7 : lp : /var/spool/lpd :sync : x : 5 : 0 : sync : /sbin : bin/sync :shutdown : x : 6 : 0 : shutdown : /sbin : /sbin/shutdownhalt : x : 7 : 0 : halt : /sbin : /sbin/haltmail : x : 8 : 12 : mail : /var/spool/mail :news : x :9 :13 : news : /var/spool/news :uucp : x :10 :14 : uucp : /var/spool/uucp :operator : x : 11 : 0 :operator : /root:games : x : 12 :100 : games :/usr/games:gopher : x : 13 : 30 : gopher : /usr/lib/gopher-data:ftp : x : 14 : 50 : FTP User : /home/ftp:nobody : x :99: 99: Nobody : /:xfs :x :100 :101 : X Font Server : /etc/xll/fs : /bin/falsegdm : x : 42 :42 : : /home/gdm: /bin/bashkids : x : 500 :500 :/home/kids : /bin/bash

## 3.6 munmap

​解除内存映射。

* 相关函数

Mmap

* 表头文件

#include<unistd.h>#include<sys/mman.h>

* 定义函数

int munmap(void \*start, size\_t length);

* 函数说明

munmap()用来取消参数start所指的映射内存起始地址，参数length则是欲取消的内存大小。当进程结束或利用exec相关函数来执行其他程序时，映射内存会自动解除，但关闭对应的文件描述词时不会解除映射。

* 返回值

如果解除映射成功则返回0，否则返回－1，错误原因存于errno中错误代码EINVAL。

* 参数

start或length 不合法。

* 范例

参考mmap（）

# 第四章 日期时间篇

## 4.1 asctime

​将时间和日期以字符串格式表示。

* 相关函数

time，ctime，gmtime，localtime

* 表头文件

#include<time.h>

* 定义函数

char \*asctime(const struct tm \*timeptr);

* 函数说明

asctime()将参数timeptr所指的tm结构中的信息转换成真实世界所使用的时间日期表示方法，然后将结果以字符串形态返回。此函数已经由时区转换成当地时间，字符串格式为:“Wed Jun 30 21:49:08 1993\n”

转换日历时间 tm 为以下固定的 25 字符表示形式： Www Mmm dd hh:mm:ss yyyy\n

Www ——来自 time\_ptr->tm\_wday 的星期之日的三字母英文缩写， Mon 、 Tue 、 Wed 、 Thu 、 Fri 、 Sat 、 Sun 之一。

Mmm ——来自 time\_ptr->tm\_mon 的月名的三字母英文缩写， Jan 、 Feb 、 Mar 、 Apr 、 May 、 Jun 、 Jul 、 Aug 、 Sep 、 Oct 、 Nov 、 Dec 之一。

dd ——来自 timeptr->tm\_mday 的 2 位月之日，如同由 sprintf 以 %2d 打印

hh ——来自 timeptr->tm\_hour 的 2 位时，如同由 sprintf 以 %.2d 打印

mm ——来自 timeptr->tm\_min 的 2 位分，如同由 sprintf 以 %.2d 打印

ss ——来自 timeptr->tm\_sec 的 2 位秒，如同由 sprintf 以 %.2d 打印

yyyy ——来自 timeptr->tm\_year + 1900 的 4 位年，如同由 sprintf 以 %4d 打印

若 \*time\_ptr 的任何成员在其正常范围外则行为未定义

若 time\_ptr->tm\_year 所指示的历年拥有多于 4 位数或小于 1000 年则行为未定义。

函数不支持本地化，且不能移除换行符。

函数修改静态存储且非线程安全。

asctime 返回指向静态数据的指针从而不是线程安全的。 POSIX 标记此函数为过时并推荐用 strftime 代替。 C 标准亦推荐用 strftime 代替 asctime 、 asctime\_r 及 asctime\_s 因为 strftime 更灵活且为本地环境相关。

* 返回值

指向保有如上描述的日期与时间的静态空终止字符串指针。该字符串可能在 asctime 与 ctime 间共享，且可能在每次调用这些函数时被重写。

若再调用相关的时间日期函数，此字符串可能会被破坏。此函数与ctime不同处在于传入的参数是不同的结构。

* 附加值

返回一字符串表示目前当地的时间日期。

* 范例

#include <time.h>main(){time\_t timep;time (&timep);printf(“%s”,asctime(gmtime(&timep)));}

* 执行

Sat Oct 28 02:10:06 2000

## 4.2 ctime

​将时间和日期以字符串格式表示。

* 相关函数

time，ctime，gmtime，localtime

* 表头文件

#include<time.h>

* 定义函数

char \*ctime(const time\_t \*timep);

* 函数说明

ctime()将参数timep所指的time\_t结构中的信息转换成真实世界所使用的时间日期表示方法，然后将结果以字符串形态返回。此函数已经由时区转换成当地时间，字符串格式为“Wed Jun 30 21 :49 :08 1993\n”。若再调用相关的时间日期函数，此字符串可能会被破坏。

* 返回值

返回一字符串表示目前当地的时间日期。

指向保有日期与时间的文本表示的静态空终止字符串的指针。该字符串可能在 asctime 与 ctime 之间共享，且可能在每次调用任何这些函数时被重写。

* 范例

#include<time.h>main(){time\_t timep;time (&timep);printf(“%s”,ctime(&timep));}

* 执行

Sat Oct 28 10 : 12 : 05 2000

## 4.3 gettimeofday

​取得目前的时间。

* 相关函数

time，ctime，ftime，settimeofday

* 表头文件

#include <sys/time.h>#include <unistd.h>

* 定义函数

int gettimeofday(struct timeval \*tv, struct timezone \*tz);

* 函数说明

gettimeofday()会把目前的时间有tv所指的结构返回，当地时区的信息则放到tz所指的结构中。 timeval结构定义为:

struct timeval{long tv\_sec; /\*秒\*/long tv\_usec; /\*微秒\*/};

timezone 结构定义为:

struct timezone{int tz\_minuteswest; /\*和Greenwich 时间差了多少分钟\*/int tz\_dsttime; /\*日光节约时间的状态\*/};

上述两个结构都定义在/usr/include/sys/time.h。tz\_dsttime 所代表的状态如下

DST\_NONE /\*不使用\*/DST\_USA /\*美国\*/DST\_AUST /\*澳洲\*/DST\_WET /\*西欧\*/DST\_MET /\*中欧\*/DST\_EET /\*东欧\*/DST\_CAN /\*加拿大\*/DST\_GB /\*大不列颠\*/DST\_RUM /\*罗马尼亚\*/DST\_TUR /\*土耳其\*/DST\_AUSTALT /\*澳洲（1986年以后）\*/

* 返回值

成功则返回0，失败返回－1，错误代码存于errno。附加说明EFAULT指针tv和tz所指的内存空间超出存取权限。

* 范例

#include<sys/time.h>#include<unistd.h>main(){struct timeval tv;struct timezone tz;gettimeofday (&tv , &tz);printf(“tv\_sec; %d\n”, tv,.tv\_sec) ;printf(“tv\_usec; %d\n”,tv.tv\_usec);printf(“tz\_minuteswest; %d\n”, tz.tz\_minuteswest);printf(“tz\_dsttime, %d\n”,tz.tz\_dsttime);}

* 执行

tv\_sec: 974857339tv\_usec:136996tz\_minuteswest:-540tz\_dsttime:0

## 4.4 gmtime

​取得目前时间和日期。

* 相关函数

time,asctime,ctime,localtime

* 表头文件

#include<time.h>

* 定义函数

struct tm \*gmtime(const time\_t \*timep);

* 函数说明

gmtime()将参数timep 所指的time\_t 结构中的信息转换成真实世界所使用的时间日期表示方法，然后将结果由结构tm返回。 结构tm的定义为

struct tm{int tm\_sec;int tm\_min;int tm\_hour;int tm\_mday;int tm\_mon;int tm\_year;int tm\_wday;int tm\_yday;int tm\_isdst;};

int tm\_sec 代表目前秒数，正常范围为0-59，但允许至61秒 int tm\_min 代表目前分数，范围0-59 int tm\_hour 从午夜算起的时数，范围为0-23 int tm\_mday 目前月份的日数，范围01-31 int tm\_mon 代表目前月份，从一月算起，范围从0-11 int tm\_year 从1900 年算起至今的年数 int tm\_wday 一星期的日数，从星期一算起，范围为0-6 int tm\_yday 从今年1月1日算起至今的天数，范围为0-365 int tm\_isdst 日光节约时间的旗标 此函数返回的时间日期未经时区转换，而是UTC时间。

* 返回值

成功时返回指向静态的内部 struct tm 对象的指针，失败时返回空指针。该结构体可能会为 gmtime 、 localtime 以及 ctime 所共享，并于每次调用时被覆盖。

* 范例

#include <time.h>main(){char \*wday[]={"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};time\_t timep;struct tm \*p;time(&timep);p=gmtime(&timep);printf(“%d%d%d”,(1900+p->tm\_year), (1+p->tm\_mon),p->tm\_mday);printf(“%s%d;%d;%d\n”, wday[p->tm\_wday], p->tm\_hour, p->tm\_min, p->tm\_sec);}

* 执行

2000/10/28 Sat 8:15:38

## 4.5 localtime

​取得当地目前时间和日期。

* 相关函数

time, asctime, ctime, gmtime

* 表头文件

#include<time.h>

* 定义函数

struct tm \*localtime(const time\_t \*timep);

* 函数说明

localtime()将参数timep所指的time\_t结构中的信息转换成真实世界所使用的时间日期表示方法，然后将结果由结构tm返回。结构tm的定义请参考gmtime()。此函数返回的时间日期已经转换成当地时区。

* 返回值

成功时为指向静态内部 tm 对象的指针，否则为空指针。该结构体可能在 gmtime 、 localtime 及 ctime 间共享并可能在每次调用时被重写。

* 范例

#include<time.h>main(){char \*wday[]={“Sun”,”Mon”,”Tue”,”Wed”,”Thu”,”Fri”,”Sat”};time\_t timep;struct tm \*p;time(&timep);p=localtime(&timep); /\*取得当地时间\*/printf (“%d%d%d ”, (1900+p->tm\_year),( l+p->tm\_mon), p->tm\_mday);printf(“%s%d:%d:%d\n”, wday[p->tm\_wday],p->tm\_hour, p->tm\_min, p->tm\_sec);}

* 执行

2000/10/28 Sat 11:12:22

## 4.6 mktime

​将时间结构数据转换成经过的秒数。

* 相关函数

time，asctime，gmtime，localtime

* 表头文件

#include<time.h>

* 定义函数

time\_t mktime(struct tm \*timeptr);

* 函数说明

mktime()用来将参数timeptr所指的tm结构数据转换成从公元1970年1月1日0时0分0 秒算起至今的UTC时间所经过的秒数。

忽略timeptr ->tm\_wday 与 timeptr ->tm\_yday 。容许 timeptr中的值在其正常范围外。

timeptr ->tm\_isdst 的负值会导致 mktime 尝试确定在指定时间夏时令是否有效。

若转换到 time\_t 成功，则修改 timeptr会被修改。更新 timeptr的所有域为符合其正确范围的值。用可用于其他域的信息重新计算 timeptr ->tm\_wday 与 timeptr ->tm\_yday 。

* 返回值

返回经过的秒数。

* 范例

/\* 用time()取得时间（秒数），利用localtime()转换成struct tm 再利用mktine（）将struct tm转换成原来的秒数\*/#include<time.h>main(){time\_t timep;struct tm \*p;time(&timep);printf(“time() : %d \n”,timep);p=localtime(&timep);timep = mktime(p);printf(“time()->localtime()->mktime():%d\n”,timep);}

* 执行

time():974943297time()->localtime()->mktime():974943297

## 4.7 settimeofday

​设置目前时间。

* 相关函数

time，ctime，ftime，gettimeofday

* 表头文件

#include<sys/time.h>#include<unistd.h>

* 定义函数

int settimeofday(const struct timeval \*tv, const struct timezone \*tz);

* 函数说明

settimeofday()会把目前时间设成由tv所指的结构信息，当地时区信息则设成tz所指的结构。详细的说明请参考gettimeofday()。注意，只有root权限才能使用此函数修改时间。

* 返回值

成功则返回0，失败返回－1，错误代码存于errno。

* 错误代码

EPERM 并非由root权限调用settimeofday（），权限不够。 EINVAL 时区或某个数据是不正确的，无法正确设置时间。

## 4.8 time

​取得目前的时间。

* 相关函数

ctime，ftime，gettimeofday

* 表头文件

#include<time.h>

* 定义函数

time\_t time(time\_t \*t);

* 函数说明

此函数会返回从公元1970年1月1日的UTC时间从0时0分0秒算起到现在所经过的秒数。如果t 并非空指针的话，此函数也会将返回值存到t指针所指的内存。

* 返回值

成功则返回秒数，失败则返回((time\_t)-1)值，错误原因存于errno中。若t不是空指针，则返回值也会存储于 t 所指向的对象。

* 范例

#include<time.h>main(){int seconds= time((time\_t\*)NULL);printf(“%d\n”,seconds);}

* 执行

9.73E+08

# 第五章 内存及字符串操作篇

## 5.1 bcmp

​比较内存内容。

* 相关函数

bcmp，strcasecmp，strcmp，strcoll，strncmp，strncasecmp

* 表头文件

#include<string.h>

* 定义函数

int bcmp(const void \*s1, const void \*s2, int n);

* 函数说明

bcmp()用来比较s1和s2所指的内存区间前n个字节，若参数n为0，则返回0。

* 返回值

若参数s1 和s2 所指的内存内容都完全相同则返回0 值，否则返回非零值。

* 附加值

建议使用memcmp()取代。

* 范例

参考memcmp()。

## 5.2 bcopy

​拷贝内存内容。

* 相关函数

memccpy，memcpy，memmove，strcpy，ctrncpy

* 表头文件

#include<string.h>

* 定义函数

void bcopy(const void \*src, void \*dest, int n);

* 函数说明

bcopy()与memcpy()一样都是用来拷贝src所指的内存内容前n个字节到dest所指的地址，不过参数src与dest在传给函数时是相反的位置。

* 附加值

建议使用memcpy()取代

* 范例

#include<string.h>main(){char dest[30]=”string(a)”;char src[30]=”string\0string”;int i;bcopy(src,dest,30);/\* src指针放在前\*/printf(bcopy(): “)for(i=0;i<30;i++)printf(“%c”,dest[i]);memcpy(dest src,30); /\*dest指针放在钱\*/printf(‘\nmemcpy() : “);for(i=0;i<30;i++)printf(“%c”,dest[i]);

* 执行

bcopy() : string stringmemcpy() :string sring

## 5.3 bzero

​将一段内存内容全清为零。

* 相关函数

memset，swab

* 表头文件

#include<string.h>

* 定义函数

void bzero(void \*s,int n)；

* 函数说明

bzero() 函数从 s 所指向的位置开始，通过向该区域写零（包含“\0 ”的字节），擦除内存中 n 个字节的数据。

指向的 n 个字节中的数据，向该区域写入零（包含“\0 ”的字节）。相当于调用memset((void\*)s,0,size\_tn);

* 附加值

建议使用memset取代。

* 范例

参考memset()。

## 5.4 index

​查找字符串中第一个出现的指定字符。

* 相关函数

rindex，srechr，strrchr

* 表头文件

#include<string.h>

* 定义函数

char \*index(const char \*s, int c);

* 函数说明

index()用来找出参数s字符串中第一个出现的参数c地址，然后将该字符出现的地址返回。字符串结束字符(NULL)也视为字符串一部分。

* 返回值

如果找到指定的字符则返回该字符所在地址，否则返回0。。

* 范例

#include<string.h>main(){char \*s =”0123456789012345678901234567890”;char \*p;p =index(s,’5’);printf(%s\n”,p);}

* 执行

5.68E+25

## 5.6 memccpy

​拷贝内存内容

* 相关函数

bcopy，memcpy，memmove，strcpy，strncpy

* 表头文件

#include<string.h>

* 定义函数

void \*memccpy(void \*dest, const void \*src, int c, size\_t n);

* 函数说明

从 src 所指向的对象复制字符到 dest 所指向的对象，在满足接下来的任何两个条件之一后停止：

复制了 count 个字符

找到（并复制）了字符 (unsigned char)c 。

转译 src 与 dest 对象为 unsigned char 的数组。

若符合任何以下条件则行为未定义：

出现越过 dest 数组结尾的访问

对象重叠（这违反 restrict 契约）

dest 或 src 为非法或空指针值

与memcpy()不同的是，memccpy()会在复制时检查参数c是否出现，若是则返回dest中值为c的下一个字节地址。

memccpy(dest, src, 0, count) 表现类似 [strncpy](mk:@MSITStore:E:\\编程学习\\cpp\\CPP_Tools\\18%20C++\\cppreference_docs\\C++.chm::/chmhelp/c-string-byte-strncpy.html)(dest, src, count) ，除了前者返回指向被写入缓冲区的*末尾*的指针，并且不会以零填充目标数组。从而 memccpy 对高效连接多个字符串有用。

* 返回值

返回指向dest中值为c的下一个字节指针。返回值为0表示在src所指内存前n个字节中没有值为c的字节。

* 范例

#include<string.h>main(){char a[]="string[a]";char b[]="string(b)";memccpy(a,b,'B',sizeof(b));printf("memccpy():%s\n",a);}

* 执行

memccpy():string(b)

## 5.6 memchr

​在某一内存范围中查找一特定字符。

* 相关函数

index，rindex，strchr，strpbrk，strrchr，strsep，strspn，strstr

* 表头文件

#include<string.h>

* 定义函数

void \*memchr(const void \*s, int c, size\_t n);

* 函数说明

在 s 所指向对象的首 n 个字符（每个都转译成 unsigned char ）中寻找 c （在如同以 (unsigned char)c 转换到 unsigned char 后）的首次出现。

若访问出现于被搜索的数组结尾后，则行为未定义。若 ptr 为空指针则行为未定义。

此函数表现如同它按顺序读取字符，并立即于找到匹配的字符时停止：若 ptr 所指向的字符数组小于 count ，但在数组中找到匹配，则行为良好定义。

memchr()从头开始搜寻s所指的内存内容前n个字节，直到发现第一个值为c的字节，则返回指向该字节的指针。

* 返回值

如果找到指定的字节则返回该字节的指针，否则返回空指针。

* 范例

#include <string.h>main(){char \*s="0123456789012345678901234567890";char \*p;p=memchr(s,'5',10);printf("%s\n",p);}

* 执行

56789012345678901234567890

## 5.7 memcmp

​比较内存内容。

* 相关函数

bcmp，strcasecmp，strcmp，strcoll，strncmp，strncasecmp

* 表头文件

#include<string.h>

* 定义函数

int memcmp(const void \*s1, const void \*s2, size\_t n);

* 函数说明

memcmp()用来比较s1和s2所指的内存区间前n个字符。字符串大小的比较是以ASCII码表上的顺序来决定，次顺序亦为字符的值。memcmp()首先将s1第一个字符值减去s2第一个字符的值，若差为0则再继续比较下个字符，若差值不为0则将差值返回。例如，字符串"Ac"和"ba"比较则会返回字符'A'(65)和'b'(98)的差值(－33)。若在 s1和 s2 所指向的任一对象结尾后出现访问，则行为未定义。若 s1 或 s2 为空指针则行为未定义。

* 返回值

若参数s1和s2所指的内存内容都完全相同则返回0值。s1若大于s2则返回大于0的值。s1若小于s2则返回小于0的值。此函数读取对象表示，而非对象值，而且典型地只对字节数组有意义：结构体可以含有填充字节而其值不确定，存储于联合体最近存储成员后的任何字节的值是不确定的，且一个类型可以对相同值拥有二种或多种表示（对于 +0 和 -0 或 +0.0 和 –0.0 的相异编码、类型中不确定填充位）。

* 范例

#include<string.h>main(){char \*a ="aBcDeF";char \*b="AbCdEf";char \*c="aacdef";char \*d="aBcDeF";printf("memcmp(a,b):%d\n",memcmp((void\*)a,(void\*) b,6));printf("memcmp(a,c):%d\n",memcmp((void\*)a,(void\*) c,6));printf("memcmp(a,d):%d\n",memcmp((void\*)a,(void\*) d,6));}

* 执行

memcmp(a,b):1 /\*字符串a>字符串b，返回1\*/memcmp(a,c):-1 /\* 字符串a<字符串c,返回－1\*/memcmp(a,d):0 /\*字符串a＝字符串d，返回0\*/

## 5.8 memcpy

​拷贝内存内容。

* 相关函数

bcopy，memccpy，memcpy，memmove，strcpy，strncpy

* 表头文件

#include<string.h>

* 定义函数

void \*memcpy(void \*dest, const void \*src, size\_t n);

* 函数说明

从 src 所指向的对象复制 n 个字符到 dest 所指向的对象。两个对象都被转译成 unsigned char 的数组。

若访问发生在 dest 数组结尾后则行为未定义。若对象重叠（这违背 restrict 契约） (C99 起)，则行为未定义。若 dest 或 src 为非法或空指针则行为未定义。

与strcpy()不同的是，memcpy()会完整的复制n个字节，不会因为遇到字符串结束'\0'而结束。

* 返回值

返回 dest 的副本，本质为更底层操作的临时内存地址，在实际操作中不建议直接使用此地址，操作完成以后，真正有意义的地址是 dest 本身。

* 附加值

指针src和dest所指的内存区域不可重叠。memcpy 可用于设置分配函数所获得对象的有效类型。

memcpy 是最快的内存到内存复制子程序。它通常比必须扫描其所复制数据的 strcpy ，或必须预防以处理重叠输入的 memmove 更高效。

许多 C 编译器将适合的内存复制循环变换为 memcpy 调用。

在严格别名使用禁止检验同一内存为二个不同类型的值处，可用 memcpy 转换值。

* 范例

#include<string.h>main(){char a[30]="string (a)";char b[30]="string\0string";int i;strcpy(a,b);printf("strcpy():");for(i=0;i<30;i++)printf("%c",a[i]);memcpy(a,b,30);printf("\nmemcpy() :");for(i=0;i<30;i++)printf("%c",a[i]);}

* 执行

strcpy() : string a )memcpy() : string string

## 5.9 memmove

​拷贝内存内容。

* 相关函数

bcopy，memccpy，memcpy，strcpy，strncpy

* 表头文件

#include<string.h>

* 定义函数

void \*memmove(void \*dest, const void \*src, size\_t n);

* 函数说明

从 src 所指向的对象复制 n 个字节到 dest 所指向的对象。两个对象都被转译成 unsigned char 的数组。对象可以重叠：如同复制字符到临时数组，再从该数组到 dest 一般发生复制。

若出现 dest 数组末尾后的访问则行为未定义。若 dest 或 src 为非法或空指针则行为未定义。

memmove()与memcpy()一样都是用来拷贝src所指的内存内容前n个字节到dest所指的地址上。不同的是，当src和dest所指的内存区域重叠时，memmove()仍然可以正确的处理，不过执行效率上会比使用memcpy()略慢些。

* 返回值

返回 dest 的副本，本质为更底层操作的临时内存地址，在实际操作中不建议直接使用此地址，操作完成以后，真正有意义的地址是dest本身。

* 附加值

指针src和dest所指的内存区域可以重叠。

memmove 可用于设置由分配函数获得的对象的有效类型。

尽管说明了“如同”使用临时缓冲区，此函数的实际实现不会带来二次复制或额外内存的开销。常用方法（ glibc 和 bsd libc ）是若目标在源之前开始，则从缓冲区开始正向复制，否则从末尾反向复制，完全无重叠时回落到更高效的 memcpy 。

在严格别名时用禁止检验同一内存为二个不同类型的值时，可使用 memmove 转换值。

* 范例

#define \_\_STDC\_WANT\_LIB\_EXT1\_\_ 1

#include <stdio.h>

#include <stdint.h>

#include <inttypes.h>

#include <string.h>

#include <stdlib.h>

int main(void)

{

char str[] = "1234567890";

puts(str);

memmove(str+4, str+3, 3); // 从 [4,5,6] 复制到 [5,6,7]

puts(str);

// 设置分配的内存的有效类型为 int

int \*p = malloc(3\*sizeof(int)); // 分配的内存无有效类型

int arr[3] = {1,2,3};

memmove(p,arr,3\*sizeof(int)); // 分配的内存现在拥有有效类型

// 转译数据

double d = 0.1;

// int64\_t n = \*(int64\_t\*)(&d); // 严格别名使用违规

int64\_t n;

memmove(&n, &d, sizeof d); // OK

printf("%a is %" PRIx64 " as an int64\_t\n", d, n);

#ifdef \_\_STDC\_LIB\_EXT1\_\_

set\_constraint\_handler\_s(ignore\_handler\_s);

char src[] = "aaaaaaaaaa";

char dst[] = "xyxyxyxyxy";

int r = memmove\_s(dst,sizeof dst,src,5);

printf("dst = \"%s\", r = %d\n", dst,r);

r = memmove\_s(dst,5,src,10); // count 大于 destsz

printf("dst = \"");

for(size\_t ndx=0; ndx<sizeof dst; ++ndx) {

char c = dst[ndx];

c ? printf("%c", c) : printf("\\0");

}

printf("\", r = %d\n", r);

#endif

}

* 结果

1234567890

1234456890

0x1.999999999999ap-4 is 3fb999999999999a as an int64\_t

dst = "aaaaayxyxy", r = 0

dst = "\0\0\0\0\0yxyxy", r = 22

## 5.10 memset

​将一段内存空间填入某值。

* 相关函数

bzero，swab

* 表头文件

#include<string.h>

* 定义函数

void \*memset(void \*s, int c, size\_t n);

* 函数说明

memset()会将参数s所指的内存区域前n个字节以参数c填入，然后返回指向s的指针。在编写程序时，若需要将某一数组作初始化，memset()会相当方便。

若出现 dest 数组结尾后的访问则行为未定义。若 dest 为空指针则行为未定义。

* 返回值

dest 的副本，本质为更底层操作的临时内存地址，在实际操作中不建议直接使用此地址，操作完成以后，真正有意义的地址是dest本身。

* 附加值

参数c虽声明为int， 但必须是unsigned char ，所以范围在0到255之间。

若 memset 所修改的对象在其生存期的剩余部分不再被访问，则此函数可以被优化掉。为此，此函数不能用于擦洗内存（例如以令填充存储密码的数组）。对 memset\_s 禁止此优化：保证进行内存写。该问题的第三方解决方案包含 FreeBSD explicit\_bzero 或 Microsoft SecureZeroMemory 。

* 范例

#include <string.h>main(){char s[30];memset (s,'A',sizeof(s));s[30]='\0';printf("%s\n",s);}

* 执行

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

## 5.11 rindex

​查找字符串中最后一个出现的指定字符。

* 相关函数

index，memchr，strchr，strrchr

* 表头文件

#include<strings.h>

* 定义函数

char \*rindex(const char \*s, int c);

* 函数说明

rindex()用来找出参数s字符串中最后一个出现的参数c地址，然后将该字符出现的地址返回。字符串结束字符(NULL)也视为字符串一部分。

* 返回值

如果找到指定的字符则返回该字符所在的地址，否则返回0。

* 范例

#include <strings.h>main(){char \*s ="0123456789012345678901234567890";char \*p;p=rindex(s,'5');printf("%s\n",p);}

* 执行

567890

## 5.12 strcasecmp

​比较两个忽略大小写的字符串。

* 相关函数

bcmp，memcmp，strcmp，strcoll，strncmp

* 表头文件

#include<strings.h>

* 定义函数

int strcasecmp(const char \*s1, const char \*s2);

* 函数说明

strcasecmp() 函数对字符串 s1 和 s2 进行逐字节比较、忽略字符的大小写。

如果发现 s1 小于、等于或大于零，则返回一个小于、等于或大于零的整数。

strncasecmp() 函数与之类似，但它比较的 s1 和 s2 的字节数不超过 n 个。

* 返回值

若参数s1和s2字符串相同则返回0。s1长度大于s2长度则返回大于0 的值，s1 长度若小于s2 长度则返回小于0的值。

* 范例

#include <strings.h>main(){char \*a="aBcDeF";char \*b="AbCdEf";if(!strcasecmp(a,b))printf("%s=%s\n",a,b);}

* 执行

aBcDeF=AbCdEf

## 5.13 strcat

​连接两字符串。

* 相关函数

bcopy，memccpy，memcpy，strcpy，strncpy

* 表头文件

#include<string.h>

* 定义函数

char \*strcat(char \*dest, const char \*src);

* 函数说明

strcat()会将参数src字符串拷贝到参数dest所指的字符串尾,字符 src[0] 替换 dest 末尾的空终止符。产生的字节字符串是空终止的。第一个参数dest要有足够的空间来容纳要拷贝的字符串。

因为 strcat 需要在每次调用时找到 dest 的结尾，故用 strcat 将多个字符串连接成一体是低效的。

为提升效率，允许 strcat\_s 从最后被写入字符到 destsz 破坏目标数组：它可以用多字节块复制，然后检查空字节。

尽管 strcat\_s 由于潜在安全风险禁止截断，还可以用带边界检查的 strncat\_s 截断字符串。

* 返回值

返回参数dest的字符串起始地址。

* 范例

#include <string.h>main(){char a[30]="string(1)";char b[]="string(2)";printf("before strcat() : %s\n",a);printf("after strcat() : %s\n",strcat(a,b));}

* 执行

before strcat () : string(1)after strcat () : string(1)string(2)

## 5.14 strchr

​查找字符串中第一个出现的指定字符。

* 相关函数

index，memchr，rinex，strbrk，strsep，strspn，strstr，strtok

* 表头文件

#include<string.h>

* 定义函数

char \*strchr(const char \*s, int c);

* 函数说明

strchr()用来找出参数s字符串中第一个出现的参数c地址，然后将该字符出现的地址返回。

寻找 c （如同用 (char)c 转换成 char 后）在 s 所指向的空终止字节字符串（转译每个字符为 unsigned char ）中的首次出现位置。终止空字符被认为是字符串的一部分，并且能在寻找 '\0' 时找到。

若 str 不是指向空终止字节字符串的指针，则行为未定义。

* 返回值

如果找到指定的字符则返回该字符所在地址，否则返回0。

* 范例

#include<string.h>main(){char \*s=0123456789012345678901234567890”;char \*p;p=strchr(s,'5');printf("%s\n",p);}

* 执行

5.68E+25

## 5.15 strcmp

​比较字符串。

* 相关函数

bcmp，memcmp，strcasecmp，strncasecmp，strcoll

* 表头文件

#include<string.h>

* 定义函数

int strcmp(const char \*s1, const char \*s2);

* 函数说明

strcmp()用来比较参数s1和s2字符串。字符串大小的比较是以ASCII 码表上的顺序来决定，此顺序亦为字符的值。strcmp()首先将s1第一个字符值减去s2第一个字符值，若差值为0则再继续比较下个字符，若差值不为0则将差值返回。例如字符串"Ac"和"ba"比较则会返回字符"A"(65)和'b'(98)的差值(－33)。

* 返回值

若参数s1和s2字符串相同则返回0。s1若大于s2则返回大于0的值。s1若小于s2则返回小于0 的值。

* 范例

#include<string.h>main(){char \*a="aBcDeF";char \*b="AbCdEf";char \*c="aacdef";char \*d="aBcDeF";printf("strcmp(a,b) : %d\n",strcmp(a,b));printf("strcmp(a,c) : %d\n",strcmp(a,c));printf("strcmp(a,d) : %d\n",strcmp(a,d));}

* 执行

strcmp(a,b) : 32strcmp(a,c) :-31strcmp(a,d) : 0

## 5.16 strcoll

​采用目前区域的字符排列次序来比较字符串。

* 相关函数

strcmp，bcmp，memcmp，strcasecmp，strncasecmp

* 表头文件

#include<string.h>

* 定义函数

int strcoll(const char \*s1, const char \*s2);

* 函数说明

strcoll() 函数比较两个字符串 s1 和 s2。如果发现 s1 分别小于、等于或大于 s2，它将返回一个小于、等于或大于零的整数。比较的基础是字符串，这些字符串被解释为适用于程序当前的 LC\_COLLATE 类别的本地语言。

* 返回值

若参数s1和s2字符串相同则返回0。s1若大于s2则返回大于0的值。s1若小于s2则返回小于0 的值。

* 附加值

若LC\_COLLATE为"POSIX"或"C"，则strcoll()与strcmp()作用完全相同。

对照顺序为字典顺序：国家字母表（其等价类）中字母的位置拥有高于其大小写或变体的优先级。在等价类内，小写字符先于其大写等价物对照，而且本地环境限定的顺序可能应用到有发音符号的字符。一些本地环境中，字符组作为单个对照单元比较。例如， "ch" 在捷克语中后随 "h" 而前趋 "i" ， "dzs" 在匈牙利语中后随 "dz" 而前趋 "g" 。

* 范例

#include <stdio.h>

#include <string.h>

#include <locale.h>

int main(void)

{

setlocale(LC\_COLLATE, "cs\_CZ.iso88592");

const char\* s1 = "hrnec";

const char\* s2 = "chrt";

printf("In the Czech locale: ");

if(strcoll(s1, s2) < 0)

printf("%s before %s\n", s1, s2);

else

printf("%s before %s\n", s2, s1);

printf("In lexicographical comparison: ");

if(strcmp(s1, s2) < 0)

printf("%s before %s\n", s1, s2);

else

printf("%s before %s\n", s2, s1);

}

* 输出

In the Czech locale: hrnec before chrt

In lexicographical comparison: chrt before hrnec

## 5.17 strcpy

​拷贝字符串。

* 相关函数

bcopy，memcpy，memccpy，memmove

* 表头文件

#include<string.h>

* 定义函数

char \*strcpy(char \*dest, const char \*src);

* 函数说明

复制 src 所指向的空终止字节字符串，包含空终止符，到首元素为 dest 所指的字符数组。

若 dest 数组长度不足则行为未定义。若字符串覆盖则行为未定义。若 dest 不是指向字符数组的指针或 src 不是指向空终止字节字符串的指针则行为未定义。

* 返回值

返回参数dest的字符串起始地址。

* 附加值

如果参数dest所指的内存空间不够大，可能会造成缓冲溢出(buffer Overflow)的错误情况，在编写程序时请特别留意，或者用strncpy()来取代。

为提升效率，允许 strcpy\_s 破坏至多 destsz 个目标数组上次写入的字符：它可能先复制多字节块再检查空字节。

尽管 strcpy\_s 因潜在的安全风险禁止截断，也还可以用使用边界检查的 strncpy\_s 替而进行截断字符串。

* 范例

#include<string.h>main(){char a[30]="string(1)";char b[]="string(2)";printf("before strcpy() :%s\n",a);printf("after strcpy() :%s\n",strcpy(a,b));}

* 执行

before strcpy() :string(1)after strcpy() :string(2)

## 5.18 strcspn

​返回字符串中连续不含指定字符串内容的字符数。

* 相关函数

strspn

* 表头文件

#include<string.h>

* 定义函数

size\_t strcspn(const char \*s, const char \*reject);

* 函数说明

strcspn()从参数s字符串的开头计算连续的字符，而这些字符都完全不在参数reject 所指的字符串中。简单地说，若strcspn()返回的数值为n，则代表字符串s开头连续有n个字符都不含字符串reject内的字符。

若 dest 或 src 不是指向空终止字节字符串的指针，则行为未定义。

* 返回值

返回字符串s开头连续不含字符串reject内的字符数目。

* 范例

#include <string.h>main(){char \*str="Linux was first developed for 386/486-based pcs.";printf("%d\n",strcspn(str," "));printf("%d\n",strcspn(str,"/-"));printf("%d\n",strcspn(str,"1234567890"));}

* 执行

5 /\*只计算到“ ”的出现，所以返回“Linux”的长度\*/33 /\*计算到出现“/”或“－”，所以返回到“6”的长度\*/30 /\* 计算到出现数字字符为止，所以返回“3”出现前的长度\*/

## 5.19 strdup

​复制字符串。

* 相关函数

calloc，malloc，realloc，free

* 表头文件

#include<string.h>

* 定义函数

char \*strdup(const char \*s);

* 函数说明

strdup()会先用maolloc()配置与参数s字符串相同的空间大小，然后将参数s字符串的内容复制到该内存地址，然后把该地址返回。该地址最后可以利用free()来释放。

若错误发生，则返回空指针，并可能设置 errno 。

同所有来自动态内存 TR 的函数， strdup 仅若实现定义了 \_\_STDC\_ALLOC\_LIB\_\_ 且用户在包含 string.h 前定义 \_\_STDC\_WANT\_LIB\_EXT2\_\_ 为整数常量 1 才保证可用。

* 返回值

返回一字符串指针，该指针指向复制后的新字符串地址。若返回NULL表示内存不足。

* 范例

#include<string.h>main(){char a[]="strdup";char \*b;b=strdup(a);printf("b[ ]=\"%s\"\n",b);}

* 执行

b[ ]="strdup"

## 5.20 strlen

​返回字符串长度。

* 表头文件

#include<string.h>

* 定义函数

size\_t strlen(const char \*s);

* 函数说明

返回给定空终止字符串的长度，即首元素为 str 所指，且不包含首个空字符的字符数组中的字符数。若 str 不是指向空终止字节字符串的指针则行为未定义。

* 返回值

返回字符串s的字符数。

* 范例

/\*取得字符串str的长度\*/#include<string.h>main(){char \*str = "12345678";printf("str length = %d\n", strlen(str));}

* 执行

str length = 8

## 5.21 strncasecmp

​忽略大小写比较字符串。

* 相关函数

bcmp，memcmp，strcmp，strcoll，strncmp

* 表头文件

#include<strings.h>

* 定义函数

int strncasecmp(const char \*s1, const char \*s2, size\_t n);

* 函数说明

strncasecmp()用来比较参数s1和s2字符串前n个字符，比较时会自动忽略大小写的差异。

* 返回值

若参数s1和s2 字符串相同则返回0。s1 若大于s2则返回大于0的值，s1若小于s2则返回小于0 的值。

* 范例

#include<strings.h>main(){char \*a="aBcDeF";char \*b="AbCdEf";if(!strncasecmp(a,b))printf("%s =%s\n",a,b);}

* 执行

aBcDef=AbCdEf

## 5.22 strncat

​连接两字符串。

* 相关函数

bcopy，memccpy，memecpy，strcpy，strncpy

* 表头文件

#include<string.h>

* 定义函数

char \*strncat(char \*dest, const char \*src, size\_t n);

* 函数说明

拷贝来自 src 所指向的字符数组的至多 n个字符，到 dest 所指向的空终止字节字符串的末尾，若找到空字符则停止。字符 src[0] 替换位于 dest 末尾的空终止符。始终后附终止空字符到末尾（故函数可写入的最大字节数是 count+1 ）。第一个参数dest要有足够的空间来容纳要拷贝的字符串。

若目标数组没有对于 dest 和 src 的首 count 个字符加上终止空字符的足够空间，则行为未定义。若源与目标对象重叠，则行为未定义。若 dest 不是指向空终止字节字符串的指针，或 src 不是指向字符数组的指针，则行为未定义。

因为 strncat 需要在每次调用时找到 dest 的结尾，故用 strncat 将多个字符串连接成一体是低效的。

尽管截断以适应目标缓冲区是安全风险，从而是 strncat\_s 的运行时制约违规，还是可以通过指定 n 等于目标数组大小减一获取截断行为：这将一如往常地复制首 n 个字节并后附空终止符： strncat\_s(dst, sizeof dst, src, (sizeof dst)-strnlen\_s(dst, sizeof dst)-1); 。

* 返回值

返回参数dest的字符串起始地址。

* 范例

#include <string.h>main(){char a[30]="string(1)";char b[]="string(2)";printf("before strnact() :%s\n", a);printf("after strncat() :%s\n", strncat(a,b,6));}

* 执行

before strnact() : string(1)after strncat() : string(1) string

## 5.23 strncpy

​拷贝字符串。

* 相关函数

bcopy，memccpy，memcpy，memmove

* 表头文件

#include<string.h>

* 定义函数

char \*strncpy(char \*dest, const char \*src, size\_t n);

* 函数说明

复制 src 所指向的字符数组的至多 n 个字符（包含空终止字符，但不包含后随空字符的任何字符）到 dest 所指向的字符数组。

若在完全复制整个 src 数组前抵达 n ，则结果的字符数组不是空终止的。

若在复制来自 src 的空终止字符后未抵达 n ，则写入额外的空字符到 dest ，直至写入总共 n 个字符。

若字符数组重叠，若 dest 或 src 不是指向字符数组的指针（包含若 dest 或 src 为空指针），若 dest 所指向的数组大小小于 n ，或若 src 所指向的数组大小小于 n 且它不含空字符，则行为未定义。

尽管适合目标缓冲区的截断是安全风险，从而是 strncpy\_s 的运行时制约违规，还是可通过指定 n 等于目标数组大小减一以获取截断行为：它会复制首 n 个字节，并照常添加空终止符： strncpy\_s(dst, sizeof dst, src, (sizeof dst)-1);

* 返回值

返回参数dest的字符串起始地址。

* 范例

#include <string.h>main(){char a[30]="string(1)";char b[]="string(2)";printf("before strncpy() : %s\n",a);printf("after strncpy() : %s\n",strncpy(a,b,6));}

* 执行

before strncpy() : string(1)after strncpy() : string(1)

## 5.24 strpbrk

​查找字符串中第一个出现的指定字符。

* 相关函数

index，memchr，rindex，strpbrk，strsep，strspn，strstr，strtok

* 表头文件

#include<string.h>

* 定义函数

char \*strpbrk(const char \*s, const char \*accept);

* 函数说明

strpbrk()用来找出参数s 字符串中最先出现存在参数accept 字符串中的任意字符。

在 s 所指向的空终止字节串中，扫描来自 accept所指向的空终止字节串的任何字符，并返回指向该字符的指针。

若 s 或 accept不是指向空终止字节字符串的指针则行为未定义。

* 返回值

如果找到指定的字符则返回该字符所在地址，否则返回0。

* 范例

#include <string.h>main(){char \*s="0123456789012345678901234567890";char \*p;p=strpbrk(s,"a1 839"); /\*1会最先在s字符串中找到\*/printf("%s\n",p);p=strprk(s,"4398");/\*3 会最先在s 字符串中找到\*/printf("%s\n",p);}

* 执行

1.23E+29

## 5.25 strrchr

​查找字符串中最后出现的指定字符。

* 相关函数

index，memchr，rindex，strpbrk，strsep，strspn，strstr，strtok

* 表头文件

#include<string.h>

* 定义函数

char \*strrchr(const char \*s, int c);

* 函数说明

strrchr()用来找出参数s字符串中最后一个出现的参数c地址，然后将该字符出现的地址返回。

寻找 c （如同用 (char)c 转换到 char 后）在 s 所指向的空终止字节串中（将每个字符转译成 unsigned char ）的最后出现。若搜索 '\0' ，则认为终止空字符为字符串的一部分，而且能找到。

若 s 不是指向空终止字节串的指针，则行为未定义。

* 返回值

如果找到指定的字符则返回该字符所在地址，否则返回0。

* 范例

#include<string.h>main(){char \*s="0123456789012345678901234567890";char \*p;p=strrchr(s,'5');printf("%s\n",p);}

* 执行

567890

## 5.26 strspn

​返回字符串中连续不含指定字符串内容的字符数。

* 相关函数

strcspn，strchr，strpbrk，strsep，strstr

* 表头文件

#include<string.h>

* 定义函数

size\_t strspn(const char \*s, const char \*accept);

* 函数说明

strspn()从参数s 字符串的开头计算连续的字符，而这些字符都完全是accept 所指字符串中的字符。简单的说，若strspn()返回的数值为n，则代表字符串s 开头连续有n 个字符都是属于字符串accept内的字符。

若 s 或 accept不是指向空终止字节字符串的指针，则行为未定义。

* 返回值

返回字符串s开头连续包含字符串accept内的字符数目。

* 范例

#include <string.h>

#include <stdio.h>

int main(void)

{

const char \*string = "abcde312$#@";

const char \*low\_alpha = "qwertyuiopasdfghjklzxcvbnm";

size\_t spnsz = strspn(string, low\_alpha);

printf("After skipping initial lowercase letters from '%s'\n"

"The remainder is '%s'\n", string, string+spnsz);

}

* 执行

After skipping initial lowercase letters from 'abcde312$#@'

The remainder is '312$#@'

## 5.27 strstr

​在一字符串中查找指定的字符串。

* 相关函数

index，memchr，rindex，strchr，strpbrk，strsep，strspn，strtok

* 表头文件

#include<string.h>

* 定义函数

char \*strstr(const char \*haystack, const char \*needle);

* 函数说明

strstr()会从字符串haystack 中搜寻字符串needle，并将第一次出现的地址返回。

若 haystack或 needle不是指向空终止字节字符串的指针，则行为未定义。

* 返回值

指向于 haystack中找到的子串首字符的指针，或若找不到该子串则为空指针。若 needle指向空字符串，则返回 haystack。

* 范例

#include<string.h>main(){char \* s="012345678901234567890123456789";char \*p;p= strstr(s,"901");printf("%s\n",p);}

* 执行

901234567890123456789

## 5.28 strtok

​分割字符串。

* 相关函数

index，memchr，rindex，strpbrk，strsep，strspn，strstr

* 表头文件

#include<string.h>

* 定义函数

char \*strtok(char \*s, const char \*delim);

* 函数说明

strtok()用来将字符串分割成一个个片段。参数s指向欲分割的字符串，参数delim则为分割字符串，当strtok()在参数s的字符串中发现到参数delim的分割字符时则会将该字符改为\0 字符。在第一次调用时，strtok()必需给予参数s字符串，往后的调用则将参数s设置成NULL。每次调用成功则返回下一个分割后的字符串指针。

寻找 s 所指向的空终止字节字符串中的下个记号。由 delim 所指向的空终止字节字符串鉴别分隔字符。

若 s 不是空指针，则调用被当做 strtok 对此特定字符串的首次调用。函数搜索首个不含于 delim 的字符。

若找不到这种字符，则 s 中完全没有记号，而函数返回空指针。

若找到这种字符，则它是记号的起始。然后函数从该点搜索首个含于 delim 的字符。

若找不到这种字符，则 s 只有一个记号，而将来对 strtok 的调用将返回空指针

若找到这种字符，则用空字符 '\0' 替换它，并将指向下个字符的指针存储于静态位置，以为后继调用所用。

然后函数返回指向记号起始的指针

若 s 为空指针，则将调用当做对 strtok 的后继调用，函数从先前调用中它剩下的位置开始。行为如同将先前存储的指针作为 s 传递。

若s 或 delim 不是指向空终止字节字符串的指针，则行为未定义。

此函数是破坏性的：它写入 '\0' 字符到字符串 str 的元素。特别是，不能以字符串字面量为 strtok 的首参数。

每次对 strtok 的调用都会修改静态对象：它不是线程安全的。

不同于大多数其他记号化器， strtok 中的分隔符能对于后继记号不同，而且甚至能依赖于先前记号的内容。

strtok\_s 函数异于 POSIX strtok\_r 函数，前者通过在被记号化的字符串外部存储，和检查运行时制约来防护。

* 返回值

返回下一个分割后的字符串指针，如果已无从分割则返回NULL。

* 范例

#include<string.h>main(){char s[]="ab-cd : ef;gh :i-jkl;mnop;qrs-tu: vwx-y;z";char \*delim="-: ";char \*p;printf("%s ";strtok(s,delim));while((p=strtok(NULL,delim)))printf("%s ",p);printf("\n");}

* 执行

ab cd ef;gh i jkl;mnop;qrs tu vwx y;z /\*－与:字符已经被\0 字符取代\*/

# 第六章 常用数学函数篇

## 6.1 abs

​计算整型数的绝对值。

* 相关函数

labs, fabs

* 表头文件

#include<stdlib.h>

* 定义函数

int abs(int j)

* 函数说明

abs()用来计算参数j的绝对值，然后将结果返回。若返回类型无法表示结果，则行为未定义。

在补码中，最负的值的绝对值处于对应整数范围外，例如对于 32 位补码类型整数， INT\_MIN 为 -2147483648 ，但其绝对值应有的结果是 2147483648 ，大于 INT\_MAX ，其值为 2147483647 。

* 返回值

返回参数j的绝对值结果。

* 范例

#ingclude <stdlib.h>main(){int ansert;answer = abs(-12);printf("|-12| = %d\n", answer);}

* 执行

|-12| = 12

## 6.2 acos

​取反余弦函数数值。

* 相关函数

asin , atan , atan2 , cos , sin , tan

* 表头文件

#include<math.h>

* 定义函数

double acos(double x);

* 函数说明

acos()用来计算参数x的反余弦值，然后将结果返回。参数x范围为－1至1之间，超过此范围则会失败。

* 返回值

若不出现错误，则返回 x 于范围 [0 ; π] 中的弧（反）余弦（ arccos(arg) ）。

若出现定义域错误，则返回实现定义值（受支持平台上为 NaN ）。

若出现下溢所致的值域错误，则返回（舍入后的）正确结果。

单位为弧度，在函数库中角度均以弧度来表示。

* 错误代码

EDOM参数x超出范围。

报告 math\_errhandling 中指定的错误。

若 x 在范围 [-1.0; 1.0] 外则出现定义域错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

◦若参数为 +1 ，则返回值 +0 。

◦若 |x| > 1 ，则返回定义域错误并返回 NaN 。

◦若参数为 NaN ，则返回 NaN 。

* 附加说明

使用GCC编译时请加入－lm。

* 范例

#include <math.h>main (){double angle;angle = acos(0.5);printf("angle = %f\n", angle);}

* 执行

angle = 1.047198

## 6.3 asin

​取反正弦函数值。

* 相关函数

acos , atan , atan2 , cos , sin , tan

* 表头文件

#include<math.h>

* 定义函数

double asin(double x)

* 函数说明

asin()用来计算参数x的反正弦值，然后将结果返回。参数x范围为－1至1之间，超过此范围则会失败。

* 返回值

若不出现错误，返回－PI/2之PI/2之间的计算结果。若出现定义域错误，则返回实现定义值（受支持平台上为 NaN ）。若出现下溢所致的值域错误，则返回（舍入后的）正确结果。

* 错误代码

EDOM参数x超出范围

报告 math\_errhandling 中指定的错误。

若 x 在范围 [-1.0; 1.0] 外则出现定义域错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

◦若参数为 ±0 ，则返回不修改的参数。

◦若 |x| > 1 ，则出现定义域错误并返回 NaN 。

◦若参数为 NaN ，则返回 NaN 。

* 附加说明

使用GCC编译时请加入－lm。

* 范例

#include<math.h>main(){double angle;angle = asin (0.5);printf("angle = %f\n",angle);}

* 执行

angle = 0.523599

## 6.4 atan

​取反正切函数值。

* 相关函数

asin , atan , atan2 , cos , sin , tan

* 表头文件

#include<math.h>

* 定义函数

double atan(double x);

* 函数说明

若不出现错误，atan()用来计算参数x的反正切值，然后将结果返回。

若出现下溢所致的值域错误，则返回（舍入后的）正确结果。

* 返回值

返回-PI/2至PI/2之间的计算结果。

* 错误代码

报告 math\_errhandling 中指定的错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

若参数为 ±0 ，则返回不修改的参数

若参数为 +∞ ，则返回 +π/2

若参数为 -∞ ，则返回 -π/2

若参数为 NaN ，则返回 NaN

* 附加说明

使用GCC编译时请加入－lm

POSIX 指定在下溢情况下，返回不修改的 x ，而若不支持如此，则返回不大于 DBL\_MIN 、 FLT\_MIN 和 LDBL\_MIN 的实现定义值。

* 范例

#include<math.h>main(){double angle;angle =atan(1);printf("angle = %f\n",angle);}

* 执行

angle = 1.570796

## 6.5 atan2

​取反余弦函数数值。

* 相关函数

asin , atan , atan2 , cos , sin , tan

* 表头文件

#include<math.h>

* 定义函数

double atan2(double y, double x);

* 函数说明

atan2()用来计算参数y/x的反正切值，然后将结果返回。

* 返回值

若不出现错误，返回-PI 至PI 之间的计算结果。

若出现定义域错误，则返回实现定义值。

若出现下溢所致的值域错误，则返回（舍入后的）正确结果。

* 错误代码

报告 math\_errhandling 中指定的错误。

若 x 与 y 均为零则可能出现定义域错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

若 x 与 y 均为零，则定义域错误不出现

若 x 与 y 均为零，则也不出现值域错误

若 y 为零，则不出现极点错误

若 y 为 ±0 且 x 为负或 -0 ，则返回 ±π

若 y 为 ±0 且 x 为正或 +0 ，则返回 ±0

若 y 为 ±∞ 且 x 有限，则返回 ±π/2

若 y 为 ±∞ 且 x 为 -∞ ，则返回 ±3π/4

若 y 为 ±∞ 且 x 为 +∞ ，则返回 ±π/4

若 x 为 ±0 且 y 为负，则返回 -π/2

若 x 为 ±0 且 y 为正，则返回 +π/2

若 x 为 -∞ 且 y 为正有限，则返回 +π

若 x 为 -∞ 且 y 为负有限，则返回 -π

若 x 为 +∞ 且 y 为正有限，则返回 +0

若 x 为 +∞ 且 y 为负有限，则返回 -0

若 x 为 NaN 或 y 为 NaN ，则返回 NaN

* 附加说明

使用GCC编译时请加入－lm。atan2(y, x) 等价于 carg(x + I\*y) 。

POSIX 指定在下溢情况下，返回不修改的 arg ，而若不支持如此，则返回不大于 DBL\_MIN 、 FLT\_MIN 和 LDBL\_MIN 的实现定义值。

* 范例

#include<math.h>main(){double angle;angle = atan2(1,2);printf("angle = %f\n", angle);}

* 执行

angle = 0.463648

## 6.6 ceil

​取不小于参数的最小整型数。

* 相关函数

Fabs

* 表头文件

#include<math.h>

* 定义函数

double ceil(double x);

* 函数说明

ceil()会返回不小于参数x的最小整数值，结果以double形态返回。

* 返回值

若不出现错误，则返回不小于 x 的最小整数值，即 ⌈arg⌉ 。

* 附加说明

使用GCC编译时请加入－lm。舍入非整数有限值时，可以（但不要求）引发 FE\_INEXACT 。

所有标准浮点格式中的最大可表示浮点值都是准确的整数，故此函数自身决不上溢；然而存储于整数对象时，结果可以溢出任何整数类型（包含 intmax\_t ）。

* 错误代码

报告 math\_errhandling 中指定的错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

◦当前舍入模式无效。

若 x 为 ±∞ ，则返回不修改的参数

若 x 为 ±0 ，则返回不修改的参数

若 x 为 NaN ，则返回 NaN

* 范例

#include<math.h>main(){double value[ ]={4.8,1.12,-2.2,0};int i;for (i=0;value[i]!=0;i++)printf("%f=>%f\n",value[i],ceil(value[i]));}

* 执行

4.800000=>5.0000001.120000=>2.000000-2.200000=>-2.000000

## 6.7 cos

​取余弦函数数值。

* 相关函数

asin , atan , atan2 , cos , sin , tan

* 表头文件

#include<math.h>

* 定义函数

double cos(double x);

* 函数说明

cos()用来计算参数x 的余玄值，然后将结果返回。

* 返回值

若不出现错误，则返回 arg 的余弦（ cos(arg) ），在范围 [-1 ; +1] 中。

若 arg 的绝对值很大，则结果可能拥有少量或无有效数字。 (C99 前)

若出现定义域错误，则返回实现定义值（受支持平台上为 NaN ）。

若出现下溢所致的值域错误，则返回（舍入后的）正确的结果。

* 附加说明

使用GCC编译时请加入－lm。参数为无穷大的情况不被指定为 C 中的定义域错误，但被定义为 POSIX 中的定义域错误。

* 错误代码

报告 math\_errhandling 中指定的错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），

则若参数为 ±0 ，则结果为 1.0

若参数为 ±∞ ，则返回 NaN 并引发 FE\_INVALID

若参数为 NaN ，则返回 NaN

* 范例

#include<math.h>main(){double answer = cos(0.5);printf("cos (0.5) = %f\n",answer);}

* 执行

cos(0.5) = 0.877583

## 6.8 cosh

​取双曲线余玄函数值。

* 相关函数

asin , atan , atan2 , cos , sin , tan

* 表头文件

#include<math.h>

* 定义函数

double cosh(double x);

* 函数说明

cosh()用来计算参数x的双曲线余玄值，然后将结果返回。数学定义式为:(exp(x)+exp(-x))/2。

* 返回值

若不出现错误，则返回 arg 的双曲余弦cosh(arg)。

若出现上溢所致的值域错误，则返回 +HUGE\_VAL 、+HUGE\_VALF 或 +HUGE\_VALL 。

* 附加说明

使用GCC编译时请加入－lm。

对于 IEEE 兼容的 double 类型，若 |x| > 710.5 ，则 cosh(x) 上溢。

* 错误代码

报告 math\_errhandling 中指定的错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

◦若参数为 ±0 ，则返回 1

◦若参数为 ±∞ ，则返回 +∞

◦若参数为 NaN ，则返回 NaN

* 范例

#include<math.h>main(){double answer = cosh(0.5);printf("cosh(0.5) = %f\n",answer);}

* 执行

cosh(0.5) = 1.127626

## 6.9 exp

​计算指数。

* 相关函数

log，log10，pow

* 表头文件

#include<math.h>

* 定义函数

double exp(double x);

* 函数说明

exp()用来计算以e为底的x次方值，即ex值，然后将结果返回。

* 返回值

若不出现错误，则返回 arg 的底 e 指数。

若出现上溢所致的值域错误，则返回 +HUGE\_VAL 、 +HUGE\_VALF 或 +HUGE\_VALL 。

若出现下溢所致的值域错误，则返回（舍入后的）正确结果。

* 附加说明

使用GCC编译时请加入－lm。对于 IEEE 兼容的 double 类型，若 709.8 < x 则保证上溢，而若 x < -708.4 则保证下溢.

* 错误代码

报告 math\_errhandling 中指定的错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

若参数为 ±0 ，则返回 1

若参数为 -∞ ，则返回 +0

若参数为 +∞ ，则返回 +∞

若参数为 NaN ，则返回 NaN

* 范例

#include<math.h>main(){double answer;answer = exp (10);printf("e^10 =%f\n", answer);}

* 执行

e^10 = 22026.465795

## 6.10 frexp

​将浮点型数分为底数与指数。

* 相关函数

ldexp，modf

* 表头文件

#include<math.h>

* 定义函数

double frexp(double x, int \*exp);

* 函数说明

frexp()用来将参数x 的浮点型数切割成底数和指数。底数部分直接返回，指数部分则借参数exp 指针返回，将返回值乘以2 的exp次方即为x的值。

* 返回值

返回参数x的底数部分，指数部分则存于exp指针所指的地址。

若 x 为零，则返回零并存储零于 \*exp 。

否则（若 x 非零），若不出现错误，则返回范围 (-1;-0.5], [0.5; 1) 中的值 y ，并存储整数值为 \*exp ，满足 y×2(\*exp)

=x 。

若存储于 \*exp 的值在 int 范围外，则行为未指定。

若 x 不是浮点数，则行为未指定。

* 附加说明

使用GCC编译时请加入－lm。

函数 frexp 与其对偶 ldexp 能一起用于操纵浮点数的表示，而无需直接的位操作。

* 错误代码

此函数不受制于任何指定于 math\_errhandling 的错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

若 x 为 ±0 ，则返回不修改的参数，并存储 0 于 \*exp 。

若 x 为 ±∞ ，则返回它，并存储未指定值于 \*exp 。

若 x 为 NaN ，则返回 NaN ，并存储未指定值于 \*exp 。

不引发浮点异常。

若 FLT\_RADIX 为 2 （或 2 的幂），则返回值准确，忽略当前舍入模式。

* 范例

#include <math.h>main(){int exp;double fraction;fraction = frexp (1024,&exp);printf("exp = %d\n",exp);printf("fraction = %f\n", fraction);}

* 执行

exp = 11fraction = 0.500000 /\* 0.5\*(2^11)=1024\*/

## 6.11 ldexp

​计算2的次方值。

* 相关函数

Frexp

* 表头文件

#include<math.h>

* 定义函数

double ldexp(double x, int exp);

* 函数说明

ldexp()用来将参数x乘上2的exp次方值，即x\*2exp。

* 返回值

若不出现错误，则返回 x 乘 2 的 exp 次幂。

若出现上溢所致的值域错误，则返回 ±HUGE\_VAL 、 ±HUGE\_VALF 或 ±HUGE\_VALL 。

若出现下溢所致的值域错误，则返回（舍入后的）正确结果。

* 错误代码

EDOM参数x超出范围。

报告 math\_errhandling 中指定的错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

决不引发 FE\_INEXACT ，除非出现值域错误（结果准确）

忽略当前舍入模式，除非出现值域错误

若 x 为 ±0 ，则返回不修改的参数

若 x 为 ±∞ ，则返回不修改的参数

若 exp 为 0 ，则返回不修改的 arg

若 x 为 NaN ，则返回 NaN

* 附加说明

使用GCC编译时请加入－lm。二进制系统上（其中 FLT\_RADIX 为 2 ）， ldexp 等价于 scalbn 。

函数 ldexp （“加载指数”）与其对偶 frexp 能一同用于操纵浮点数的表示，而无需直接的位操作。

多数实现上， ldexp 效率低于用通常算术运算符乘或除以二的幂。

* 范例

/\* 计算3\*(2^2)＝12 \*/#include<math.h>main(){int exp;double x,answer;answer = ldexp(3,2);printf("3\*2^(2) = %f\n",answer);}

* 执行

3\*2^(2) = 12.000000

## 6.12 log

​计算以e 为底的对数值。

* 相关函数

exp，log10，pow

* 表头文件

#include<math.h>

* 定义函数

double log(double x);

* 函数说明

log（）用来计算以e为底的x 对数值，然后将结果返回。

* 返回值

若不出现错误，则返回 x 的自然（底 e ）对数（ ln(x) 或loge(x) ）。

若出现定义域错误，则返回实现定义值（支持的平台上为 NaN ）。

若出现极点错误，则返回 -HUGE\_VAL 、 -HUGE\_VALF 或 -HUGE\_VALL 。

* 错误代码

EDOM 参数x为负数，ERANGE 参数x为零值，零的对数值无定义。

报告 math\_errhandling 中指定的错误。

若 x 小于零则出现定义域错误。

若 x 为零则可能出现极点错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

若参数为 ±0 ，则返回 -∞ 并引发 FE\_DIVBYZERO 。

若参数为 1 ，则返回 +0 。

若参数为负数，则返回 NaN 并引发 FE\_INVALID 。

若参数为 +∞ ，则返回 +∞ 。

若参数为 NaN ，则返回 NaN 。

* 附加说明

使用GCC编译时请加入－lm。

* 范例

#include<math.h>main(){double answer;answer = log (100);printf("log(100) = %f\n",answer);}

* 执行

log(100) = 4.605170

## 6.13 log10

​计算以10 为底的对数值。

* 相关函数

exp，log，pow

* 表头文件

#include<math.h>

* 定义函数

double log10(double x);

* 函数说明

log10()用来计算以10为底的x对数值，然后将结果返回。

* 返回值

若不出现错误，则返回 x 的常用（底 10 ）对数（ log10(x) 或 lg(x) ）。

若出现定义域错误，则返回实现定义值（支持的平台上为 NaN ）。

若出现极点错误，则返回 -HUGE\_VAL 、 -HUGE\_VALF 或 -HUGE\_VALL 。

* 错误代码

EDOM参数x超出范围。

* 附加说明

使用GCC编译时请加入－lm。报告 math\_errhandling 中指定的错误。

若 x 小于零则出现定义域错误。

若 x 为零则可能出现极点错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

若参数为 ±0 ，则返回 -∞ 并引发 FE\_DIVBYZERO 。

若参数为 1 ，则返回 +0 。

若参数为负数，则返回 NaN 并引发 FE\_INVALID 。

若参数为 +∞ ，则返回 +∞ 。

若参数为 NaN ，则返回 NaN 。

* 范例

#include<math.h>main(){double answer;answer = log10(100);printf("log10(100) = %f\n",answer);}

* 执行

log10(100) = 2.000000

## 6.14 pow

​计算次方值。

* 相关函数

exp，log，log10

* 表头文件

#include<math.h>

* 定义函数

double pow(double x, double y);

* 函数说明

pow()用来计算以x为底的y次方值，即xy值，然后将结果返回。

* 返回值

若不出现错误，则返回 x 的 y 次幂。

若出现定义域错误，则返回实现定义值（支持的平台上为 NaN ）。

若出现极点错误或上溢所致的值域错误，则返回 ±HUGE\_VAL 、 ±HUGE\_VALF 或 ±HUGE\_VALL 。

若出现下溢所致的值域错误，则返回（舍入后的）正确结果。

* 错误代码

EDOM参数x超出范围。

报告 math\_errhandling 中指定的错误。

若 x 有限且为负，且 y 有限且为非整数，则出现定义域错误，并可能出现值域错误。

若 x 为零且 y 为零，则可能出现定义域错误。

若 x 为零且 y 为负，则可能出现定义域错误或极点错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

pow(+0, y) ，其中 y 为负奇数，返回 +∞ 并引发 FE\_DIVBYZERO

pow(-0, y) ，其中 y 为负奇数，返回 -∞ 并引发 FE\_DIVBYZERO

pow(±0, y) ，其中 y 为有限负数，且为偶数或非整数，则返回 +∞ 并引发 FE\_DIVBYZERO

pow(±0, -∞) 返回 +∞ 并可能引发 FE\_DIVBYZERO (C23 前)

pow(+0, y) ，其中 y 为正奇数，返回 +0

pow(-0, y) ，其中 y 为正奇数，返回 -0

pow(±0, y) ，其中 y 为正非整数或正偶数，返回 +0

pow(-1, ±∞) returns 1

pow(+1, y) 对于任何 y 返回 1 ，即使 y 为 NaN

pow(x, ±0) 对于任何 x 返回 1 ，即使 x 为 NaN

pow(x, y) 返回 NaN 并引发 FE\_INVALID ，若 x 为有限负数且 y 为有限非整数。

pow(x, -∞) 对任何 |x|<1 返回 +∞

pow(x, -∞) 对任何 |x|>1 返回 +0

pow(x, +∞) 对任何 |x|<1 返回 +0

pow(x, +∞) 对任何 |x|>1 返回 +∞

pow(-∞, y) 返回 -0 ，若 y 为负奇整数

pow(-∞, y) 返回 +0 ，若 y 为负非整数或负偶数

pow(-∞, y) 返回 -∞ ，若 y 为正奇整数

pow(-∞, y) 返回 +∞ ，若 y 为正非整数或正偶数

pow(+∞, y) 对任何 y 返回 +0

pow(+∞, y) 对任何 y 返回 +∞

除了指定于上处，若任何参数为 NaN ，则返回 NaN

* 附加说明

使用GCC编译时请加入－lm。尽管 pow 不能获得负数的开方根，也为 y 为 1/3 的常用情况提供了 cbrt 。

* 范例

#include <math.h>main(){double answer;answer =pow(2,10);printf("2^10 = %f\n", answer);}

* 执行

2^10 = 1024.000000

## 6.15 sin

​取正玄函数值。

* 相关函数

acos，asin，atan，atan2，cos，tan

* 表头文件

#include<math.h>

* 定义函数

double sin(double x);

* 函数说明

sin（）用来计算参数x的正玄值，然后将结果返回。

* 返回值

若不发生错误，则返回 x 的正弦（ sin(arg) ），于范围 [-1 ; +1] 中。

若 x 的绝对值很大，结果可能拥有少量或无有效数字。 (C99 前)

若发生定义域错误，则返回实现定义的值（受支持的平台上为 NaN ）。

若发生下溢所致的值域错误，则返回（舍入后的）正确结果。

* 附加说明

使用GCC编译时请加入－lm。参数为无穷大的情况不指定为 C 中的定义域错误，但它被指定为 POSIX 中的定义域错误。

POSIX 亦指定在溢出的情况下，返回不修改的 x ，而且若不支持如此，则返回实现定义的不大于 DBL\_MIN 、 FLT\_MIN 及 LDBL\_MIN 的值。

* 错误代码

报告 math\_errhandling 中指定的错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

◦若参数是 ±0 ，则返回不修改的参数

◦若参数是 ±∞ ，则返回 NaN 并引发 FE\_INVALID

◦若参数是 NaN ，则返回 NaN

* 范例

#include<math.h>main(){double answer = sin (0.5);printf("sin(0.5) = %f\n",answer);}

* 执行

sin(0.5) = 0.479426

## 6.16 sinh

​取双曲线正玄函数值。

* 相关函数

cosh，tanh

* 表头文件

#include<math.h>

* 定义函数

double sinh(double x);

* 函数说明

sinh()用来计算参数x的双曲线正玄值，然后将结果返回。数学定义式为:(exp(x)-exp(-x))/2。

* 返回值

若不出现错误，则返回 arg 的双曲正弦sinh(x)。

若出现上溢所致的值域错误，则返回 ±HUGE\_VAL 、 ±HUGE\_VALF 或 ±HUGE\_VALL 。

若出现下溢所致的值域错误，则返回（舍入后的）正确结果。

* 错误代码

EDOM参数x超出范围。

报告 math\_errhandling 中指定的错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

若参数为 ±0 或 ±∞ ，则返回不修改的参数

若参数为 NaN ，则返回 NaN

* 附加说明

使用GCC编译时请加入－lm。POSIX 指定在下溢情况下，返回不修改的 x ，而若不支持如此，则返回不大于 DBL\_MIN 、 FLT\_MIN 和 LDBL\_MIN 的实现定义值。

* 范例

#include<math.h>main(){double answer = sinh (0.5);printf("sinh(0.5) = %f\n",answer);}

* 执行

sinh(0.5) = 0.521095

## 6.17 sqrt

​计算平方根值。

* 相关函数

Hypotq

* 表头文件

#include<math.h>

* 定义函数

double sqrt(double x);

* 函数说明

sqrt()用来计算参数x的平方根，然后将结果返回。参数x必须为正数。

* 返回值

返回参数x的平方根值。

若不出现错误，则返回 x 的平方根。

若出现定义域错误，则返回实现定义值（支持的平台上为 NaN ）。

若出现下溢所致的值域错误，则返回（舍入后的）正确结果。

* 错误代码

EDOM 参数x为负数。

报告 math\_errhandling 中指定的错误。

若 x 小于零则出现定义域错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

若参数小于 -0 ，则引发 FE\_INVALID 并返回 NaN 。

若参数为 +∞ 或 ±0 ，则返回不修改的参数。

若参数为 NaN ，则返回 NaN 。

* 附加说明

使用GCC编译时请加入－lm。IEEE 标准要求 sqrt 为准确。其他要求为准确的运算只有算术运算符和函数 fma 。舍入到返回类型后（用默认舍入模式）， sqrt 的结果与无限精度结果不可辨别。换言之，误差小于 0.5 ulp 。其他函数，含 pow ，不受这种制约。

* 范例

/\* 计算200的平方根值\*/#include<math.h>main(){double root;root = sqrt (200);printf("answer is %f\n",root);}

* 执行

answer is 14.142136

## 6.18 acos

​取正切函数值。

* 相关函数

atan，atan2，cos，sin

* 表头文件

#include <math.h>

* 定义函数

double tan(double x);

* 函数说明

tan()用来计算参数x的正切值，然后将结果返回。

* 返回值

若不出现错误，则返回 x 于范围 [0 ; π] 中的弧（反）余弦（ arccos(arg) ）。

若出现定义域错误，则返回实现定义值（受支持平台上为 NaN ）。

若出现下溢所致的值域错误，则返回（舍入后的）正确结果。

* 错误代码

EDOM参数x超出范围。

报告 math\_errhandling 中指定的错误。

若 x 在范围 [-1.0; 1.0] 外则出现定义域错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

若参数为 +1 ，则返回值 +0 。

若 |x| > 1 ，则返回定义域错误并返回 NaN 。

若参数为 NaN ，则返回 NaN 。

* 附加说明

使用GCC编译时请加入－lm。

* 范例

#include<math.h>main(){double answer = tan(0.5);printf("tan (0.5) = %f\n",answer);}

* 执行

tan(0.5) = 0.546302

## 6.19 tanh

​取双曲线正切函数值。

* 相关函数

cosh，sinh

* 表头文件

#include<math.h>

* 定义函数

double tanh(double x);

* 函数说明

tanh()用来计算参数x的双曲线正切值，然后将结果返回。数学定义式为:sinh(x)/cosh(x)。

* 返回值

若不出现错误，则返回 x 的双曲正切tanh(arg)。

若发生下溢所致的错误，则返回（舍入后的）正确结果。

* 错误代码

EDOM参数x超出范围。

报告 math\_errhandling 中指定的错误。

若实现支持 IEEE 浮点算术（ IEC 60559 ），则

若参数为 ±0 ，则返回 ±0

若参数为 ±∞ ，则返回 ±1

若参数为 NaN ，则返回 NaN

* 附加说明

使用GCC编译时请加入－lm。POSIX 指定在下溢的情况中，返回不修改的 x ，而且若不支持这么做，则返回不大于 DBL\_MIN 、 FLT\_MIN 和 LDBL\_MIN 的实现定义值。

* 范例

#include<math.h>main(){double answer = tanh(0.5);printf("tanh(0.5) = %f\n",answer);}

* 执行

tanh(0.5) = 0.462117

# 第七章 用户组篇

## 7.1 endgrent

​关闭组文件。

* 相关函数

getgrent，setgrent

* 表头文件

#include<grp.h>#include<sys/types.h>

* 定义函数

void endgrent(void);

* 函数说明

endgrent()用来关闭由getgrent()所打开的密码文件。

* 范例

请参考getgrent()与setgrent()。

## 7.2 endpwent

​关闭密码文件。

* 相关函数

getpwent，setpwent

* 表头文件

#include<pwd.h>#include<sys/types.h>

* 定义函数

void endpwent(void);

* 函数说明

endpwent()用来关闭由getpwent()所打开的密码文件。

* 范例

请参考getpwent()与setpwent()。

## 7.3 endutent

​关闭utmp 文件。

* 相关函数

getutent，setutent

* 表头文件

#include<utmp.h>

* 定义函数

void endutent(void);

* 函数说明

endutent()用来关闭由getutent所打开的utmp文件。当用户代码使用其他函数访问完文件后，应调用该函数。

* 范例

请参考getutent()。

## 7.4 fgetgrent

​从指定的文件来读取组格式。

* 相关函数

getpwent，setpwent

* 表头文件

#include<grp.h>#include<stdio.h>#include<sys/types.h>

* 定义函数

struct group \*getgrent(FILE \*stream);

* 函数说明

fgetgrent()会从参数stream指定的文件读取一行数据，然后以group结构将该数据返回。参数stream所指定的文件必须和、etc/group相同的格式。group结构定义请参考getgrent()。

* 返回值

函数 fgetgrent() 返回一个指向组结构的指针，如果没有更多数据或发生错误，则返回 NULL。如果出现错误，则设置 Errno 以指示错误。

* 范例

#include <grp.h>#include<sys/types.h>#include<stdio.h>main(){struct group \*data;FILE \*stream;int i;stream = fopen("/etc/group", "r");while((data = fgetgrent(stream))!=0){i=0;printf("%s :%s:%d :", data->gr\_name,data->gr\_passwd,data->gr\_gid);while (data->gr\_mem[i])printf("%s,",data->gr\_mem[i++]);printf("\n");}fclose(stream);}

* 执行

root:x:0:root,bin:x:1:root,bin,daemondaemon:x:2:root,bin,daemonsys:x:3:root,bin,admadm:x:4:root,adm,daemontty:x:5disk:x:6:rootlp:x:7:daemon,lpmem:x:8kmem:x:9wheel:x:10:rootmail:x:12:mailnews:x:13:newsuucp:x:14:uucpman:x:15games:x:20gopher:x:30dip:x:40:ftp:x:50nobody:x:99:

## 7.5 fgetpwent

​从指定的文件来读取密码格式。

* 相关函数

getpwent，setpwent

* 表头文件

#include<pwd.h>#include<stdio.h>#include<sys/types.h>

* 定义函数

struct passwd \*fgetpwent(FILE \*stream);

* 函数说明

fgetpwent()会从参数stream指定的文件读取一行数据，然后以passwd结构将该数据返回。参数stream所指定的文件必须和/etc/passwd相同的格式。passwd结构定义请参考getpwent()。

* 返回值

函数 fgetpwent() 返回一个指向 passwd 结构的指针，如果没有更多数据或发生错误，则返回 NULL。如果出现错误，则设置 errno 以指示错误信息

* 范例

#include<pwd.h>#include<sys/types.h>main(){struct passwd \*user;FILE \*stream;stream = fopen("/etc/passwd", "r");while((user = fgetpwent(stream))!=0){printf("%s:%d:%d:%s:%s:%s\n",user->pw\_name,user->pw\_uid,user->pw\_gid,user->pw\_gecos,user->pw\_dir,user->pw\_shell);}}

* 执行

root:0:0:root:/root:/bin/bashbin:1:1:bin:/bin:daemon:2:2:daemon:/sbin:adm:3:4:adm:/var/adm:lp:4:7:lp:/var/spool/lpd:sync:5:0:sync:/sbin:/bin/syncshutdown:6:0:shutdown:/sbin:/sbin/shutdownhalt:7:0:halt:/sbin:/sbin/haltmail:8:12:mail:/var/spool/mail:news:9:13:news:var/spool/newsuucp:10:14:uucp:/var/spool/uucp:operator:11:0:operator :/root:games:12:100:games:/usr/games:gopher:13:30:gopher:/usr/lib/gopher-data:ftp:14:50:FTP User:/home/ftp:nobody:99:99:Nobody:/:xfs:100:101:X Font Server: /etc/Xll/fs:/bin/falsegdm:42:42:/home/gdm:/bin/bashkids:500:500: : /home/kids:/bin/bash

## 7.6 getegid

​取得有效的组识别码。

* 相关函数

getgid，setgid，setregid

* 表头文件

#include<unistd.h>#include<sys/types.h>

* 定义函数

gid\_t getegid(void);

* 函数说明

getegid()用来取得执行目前进程有效组识别码。有效的组识别码用来决定进程执行时组的权限。返回值返回有效的组识别码。

* 范例

main(){ printf("egid is %d\n",getegid());}

* 执行

egid is 0 /\*当使用root身份执行范例程序时\*/

## 7.7 geteuid

​取得有效的用户识别码。

* 相关函数

getuid，setreuid，setuid

* 表头文件

#include<unistd.h>#include<sys/types.h>

* 定义函数

uid\_t geteuid(void)

* 函数说明

geteuid()用来取得执行目前进程有效的用户识别码。有效的用户识别码用来决定进程执行的权限，借由此改变此值，进程可以获得额外的权限。倘若执行文件的setID位已被设置，该文件执行时，其进程的euid值便会设成该文件所有者的uid。例如，执行文件/usr/bin/passwd的权限为-r-s--x--x，其s 位即为setID(SUID)位，而当任何用户在执行passwd 时其有效的用户识别码会被设成passwd 所有者的uid 值，即root的uid 值(0)。

* 返回值

返回有效的用户识别码。

* 范例

main(){ printf ("euid is %d \n",geteuid());})

* 执行

euid is 0 /\*当使用root身份执行范例程序时\*/

## 7.8 getgid

​取得真实的组识别码。

* 相关函数

getpwent，setpwent

* 表头文件

#include<unistd.h>#include<sys/types.h>

* 定义函数

gid\_t getgid(void);

* 函数说明

getgid()用来取得执行目前进程的组识别码。

* 返回值

返回组识别码

* 范例

main(){printf(“gid is %d\n”,getgid());}

* 执行

gid is 0 /\*当使用root身份执行范例程序时\*/

## 7.9 getgrent

​从组文件中取得账号的数据。

* 相关函数

getpwent，setpwent

* 表头文件

#include<grp.h>#include <sys/types.h>

* 定义函数

struct group \*getgrent(void);

* 函数说明

getgrent()用来从组文件(/etc/group)中读取一项组数据，该数据以group 结构返回。第一次调用时会取得第一项组数据，之后每调用一次就会返回下一项数据，直到已无任何数据时返回NULL。

struct group{char \*gr\_name; /\*组名称\*/char \*gr\_passwd; /\* 组密码\*/gid\_t gr\_gid; /\*组识别码\*/char \*\*gr\_mem; /\*组成员账号\*/}

* 返回值

返回group结构数据，如果返回NULL则表示已无数据，或有错误发生。

* 附加说明

getgrent()在第一次调用时会打开组文件，读取数据完毕后可使用endgrent()来关闭该组文件。

* 错误代码

ENOMEM 内存不足，无法配置group结构。

* 范例

#include<grp.h>#include<sys/types.h>main(){struct group \*data;int i;while((data= getgrent())!=0){i=0;printf(“%s:%s:%d:”,data->gr\_name,data->gr\_passwd,data->gr\_gid);while(data->gr\_mem[i])printf(“%s,”,data->gr\_mem[i++]);printf(“\n”);}endgrent();}

* 执行

root:x:0:root,bin:x:1:root,bin,daemon,daemon:x:2:root,bin,daemon,sys:x:3:root,bin,adm,adm:x:4:root,adm,daemontty:x:5disk:x:6:rootlp:x:7:daemon,lpmem:x:8kmem:x:9:wheel:x:10:rootmail:x:12:mailnews:x:13:newsuucp:x:14:uucpman:x:15:games:x:20gopher:x:30dip:x:40ftp:x:50nobody:x:99

## 7.10 getgrgid

​从组文件中取得指定gid 的数据。

* 相关函数

fgetgrent，getgrent，getgrnam

* 表头文件

#include<grp.h>#include<sys/types.h>

* 定义函数

struct group \*getgrgid(gid\_t gid);

* 函数说明

getgrgid（）用来依参数gid指定的组识别码逐一搜索组文件，找到时便将该组的数据以group结构返回。group结构请参考getgrent（）。

* 返回值

getgrgid() 函数返回一个指向结构体的指针，该结构体包含组数据库中与组 ID gid 匹配的记录的细分字段。如果找不到匹配的数据或发生错误，则返回 NULL。如果发生错误，则设置 errno 以指示错误。如果要在调用后检查 errno，则应在调用前将其设置为 0

* 范例

/\* 取得gid＝3的组数据\*/#include<grp.h>#include<sys/types.h>main(){struct group \*data;int i=0;data = getgrgid(3);printf(“%s:%s:%d:”,data->gr\_name,data->gr\_passwd,data->gr\_gid);while(data->gr\_mem[i])printf(“%s ,”,data->mem[i++]);printf(“\n”);}

* 执行

sys:x:3:root,bin,adm

## 7.11 getgrnam

​从组文件中取得指定组的数据。

* 相关函数

fgetgrent，getgrent，getgrnam

* 表头文件

#include<grp.h>#include<sys/types.h>

* 定义函数

struct group \*getgrnam(const char \*name);

* 函数说明

getgrnam() 函数返回一个指向结构体的指针，该结构包含组数据库（如本地组文件 /etc/group、NIS 和LDAP）中与组名称相匹配的记录字段的结构指针。group 结构请参考getgrent（）。

* 返回值

getgrnam()函数返回指向group结构体的指针，如果找不到匹配的数据或发生错误，则返回 NULL。如果发生错误，则设置 errno 以指示错误。如果要在调用后检查 errno，则应在调用前将其设置为 0。

* 范例

/\* 取得adm的组数据\*/#include<grp.h>#include<sys/types.h>main(){struct group \* data;int i=0;data = getgrnam(“adm”);printf(“%s:%s:%d:”,data->gr\_name,data->gr\_passwd,data->gr\_gid);while(data->gr\_mem[i])printf(“%s,”,data->gr\_mem[i++]);printf(“\n”);}

* 执行

adm:x:4:root,adm,daemon

## 7.12 getgroups

​取得组代码。

* 相关函数

initgroups，setgroup，getgid，setgid

* 表头文件

#include<grp.h>#include<unistd.h>

* 定义函数

int getgroups(int size,gid\_t list[]);

* 函数说明

getgroups() 会以列表形式返回调用进程的补充组 ID。参数 size 应设置为 list 指向的缓冲区中可存储的最大数据数。

如果调用进程是超过 size 的补充组的成员，则会出错。至于调用进程的有效组 ID 是否包含在返回的列表中，目前还没有明确说明。(因此，应用程序还应调用 getegid(2)，并添加或删除生成的值）。

如果 size 为零，则不修改列表，但会返回进程的补充组 ID 总数。这样，调用者就可以确定动态分配列表的大小，以便在进一步调用 getgroups() 时使用。

* 返回值

成功时，getgroups() 返回补充组 ID 的数量。如果出错，则返回 -1 并设置 errno 表示错误

* 错误代码

EFAULT 参数list数组地址不合法。EINVAL 参数size值不足以容纳所有的组。

* 范例

#include<unistd.h>#include<sys/types.h>main(){gid\_t list[500];int x,i;x = getgroups(0.list);getgroups(x,list);for(i=0;i<x;i++)printf(“%d:%d\n”,i,list[i]);}

* 执行

0:001:012:023:034:045:066:10

## 7.13 getpw

​取得指定用户的密码文件数据。

* 相关函数

fgetgrent，getgrent，getgrnam

* 表头文件

#include<pwd.h>#include<sys/types.h>

* 定义函数

int getpw(uid\_t uid,char \*buf);

* 函数说明

getpw()会从/etc/passwd中查找符合参数uid所指定的用户账号数据，找不到相关数据就返回-1。所返回的buf字符串格式如下:账号:密码:用户识别码(uid):组识别码(gid):全名:根目录:shell

* 返回值

getpw() 函数成功时返回 0；错误时返回-1，并设置 errno 以指示错误。

如果在/etc/passwd中找不到 uid，getpw() 返回-1，将 errno 设为 0，并保持 buf 不变。

* 附加说明

getpw()会有潜在的安全性问题，请尽量使用别的函数取代。使用shadow的系统已把用户密码抽出/etc/passwd，因此使用getpw()取得的密码将为“x”。

* 范例

#include<pwd.h>#include<sys/types.h>main(){char buffer[80];getpw(0,buffer);printf(“%s\n”,buffer);}

* 执行

root:x:0:0:root:/root:/bin/bash

## 7.14 getpwent

​从密码文件中取得账号的数据。

* 相关函数

getpw，fgetpwent，getpwnam，getpwuid，setpwent，endpwent

* 表头文件

#include<pwd.h>#include<sys/types.h>

* 定义函数

struct passwd \*getpwent(void);

* 函数说明

getpwent（）用来从密码文件（/etc/passwd）中读取一项用户数据，该用户的数据以passwd 结构返回。第一次调用时会取得第一位用户数据，之后每调用一次就会返回下一项数据，直到已无任何数据时返回NULL。 passwd 结构定义如下

struct passwd{char \* pw\_name; /\*用户账号\*/char \* pw\_passwd; /\*用户密码\*/uid\_t pw\_uid; /\*用户识别码\*/gid\_t pw\_gid; /\*组识别码\*/char \* pw\_gecos; /\*用户全名\*/char \* pw\_dir; /\*家目录\*/char \* pw\_shell; /\* 所使用的shell路径\*/};

* 返回值

getpwent() 函数返回一个指向 passwd 结构的指针，如果没有更多条目或发生错误，则返回 NULL。如果发生错误，则设置 errno 以指示错误。

如果要在调用后检查 errno，则应在调用前将其设置为零。返回值可能指向静态区域，并可能被后续调用 getpwent()、getpwnam(3) 或 getpwuid(3) 所覆盖。(不要将返回指针传递给 free(3)）。

* 附加说明

getpwent()在第一次调用时会打开密码文件，读取数据完毕后可使用endpwent()来关闭该密码文件。错误代码ENOMEM 内存不足，无法配置passwd结构。

* 范例

#include<pwd.h>#include<sys/types.h>main(){struct passwd \*user;while((user = getpwent())!=0){printf(“%s:%d:%d:%s:%s:%s\n”,user->pw\_name,user->pw\_uid,user->pw\_gid,user->pw\_gecos,user->pw\_dir,user->pw\_shell);}endpwent();}

* 执行

root:0:0:root:/root:/bin/bashbin:1:1:bin:/bin:daemon:2:2:daemon:/sbin:adm:3:4:adm:/var/adm:lp:4:7:lp:/var/spool/lpd:sync:5:0:sync:/sbin:/bin/syncshutdown:6:0:shutdown:/sbin:/sbin/shutdownhalt:7:0:halt:/sbin:/sbin/haltmail:8:12:mail:/var/spool/mail:news:9:13:news:var/spool/newsuucp:10:14:uucp:/var/spool/uucp:operator:11:0:operator :/root:games:12:100:games:/usr/games:gopher:13:30:gopher:/usr/lib/gopher-data:ftp:14:50:FTP User:/home/ftp:nobody:99:99:Nobody:/:xfs:100:101:X Font Server: /etc/Xll/fs:/bin/falsegdm:42:42:/home/gdm:/bin/bashkids:500:500: : /home/kids:/bin/bash

## 7.15 getpwnam

​从密码文件中取得指定账号的数据。

* 相关函数

getpw，fgetpwent，getpwent，getpwuid

* 表头文件

#include<pwd.h>#include<sys/types.h>

* 定义函数

struct passwd \*getpwnam(const char \*name);

* 函数说明

getpwnam()用来逐一搜索参数name 指定的账号名称，找到时便将该用户的数据以passwd结构返回。passwd结构请参考getpwent()。

* 返回值

getpwnam()函数返回指向 passwd 结构的指针，如果未找到匹配条目或发生错误，则返回 NULL。如果发生错误，则会设置 errno 以指示错误。如果想在调用后检查 errno，则应在调用前将其设置为零。

返回值可能指向静态区域，并可能被后续调用 getpwent(3)、getpwnam() 或 getpwuid() 所覆盖。(不要将返回指针传递给 free(3)）。

* 范例

/\*取得root账号的识别码和根目录\*/#include<pwd.h>#include<sys/types.h>main(){struct passwd \*user;user = getpwnam(“root”);printf(“name:%s\n”,user->pw\_name);printf(“uid:%d\n”,user->pw\_uid);printf(“home:%s\n”,user->pw\_dir);}

* 执行

name:rootuid:0home:/root

## 7.16 getpwuid

​从密码文件中取得指定uid 的数据。

* 相关函数

getpw，fgetpwent，getpwent，getpwnam

* 表头文件

#include<pwd.h>#include<sys/types.h>

* 定义函数

struct passwd \*getpwuid(uid\_t uid);

* 函数说明

getpwuid()用来逐一搜索参数uid 指定的用户识别码，找到时便将该用户的数据以结构返回结构请参考将该用户的数据以passwd 结构返回。passwd 结构请参考getpwent()。

* 返回值

getpwuid() 函数返回指向 passwd 结构的指针，如果未找到匹配条目或发生错误，则返回 NULL。如果发生错误，则会设置 errno 以指示错误。如果想在调用后检查 errno，则应在调用前将其设置为零。

返回值可能指向静态区域，并可能被后续调用 getpwent(3)、getpwnam() 或 getpwuid() 所覆盖。(不要将返回指针传递给 free(3)）。

* 范例

#include<pwd.h>#include<sys/types.h>main(){struct passwd \*user;user= getpwuid(6);printf(“name:%s\n”,user->pw\_name);printf(“uid:%d\n”,user->pw\_uid);printf(“home:%s\n”,user->pw\_dir);}

* 执行

name:shutdownuid:6home:/sbin

## 7.17 getuid

​取得真实的用户识别码。

* 相关函数

geteuid，setreuid，setuid

* 表头文件

#include<unistd.h>

* 定义函数

uid\_t getuid(void);

* 函数说明

getuid()用来取得执行目前进程的用户识别码。

* 返回值

用户识别码

* 范例

main(){ printf(“uid is %d\n”,getuid());}

* 执行

uid is 0 /\*当使用root身份执行范例程序时\*/

## 7.18 getutent

​从utmp 文件中取得账号登录数据。

* 相关函数

getutent，getutid，getutline，setutent，endutent，pututline，utmpname

* 表头文件

#include<utmp.h>

* 定义函数

struct utmp \*getutent(void);

* 函数说明

getutent()用来从utmp 文件(/var/run/utmp)中读取一项登录数据，该数据以utmp 结构返回。第一次调用时会取得第一位用户数据，之后每调用一次就会返回下一项数据，直到已无任何数据时返回NULL。utmp结构定义如下

struct utmp{short int ut\_type; /\*登录类型\*/pid\_t ut\_pid; /\*login进程的pid\*/char ut\_line[UT\_LINESIZE];/\*登录装置名，省略了“/dev/”\*/char ut\_id[4]; /\* Inittab ID\*/char ut\_user[UT\_NAMESIZE];/\*登录账号\*/char ut\_host[UT\_HOSTSIZE];/\*登录账号的远程主机名称\*/struxt exit\_status ut\_exit;/\* 当类型为DEAD\_PROCESS时进程的结束状态\*/long int ut\_session; /\*Sessioc ID\*/struct timeval ut\_tv; /\*时间记录\*/int32\_t ut\_addr\_v6[4]; /\*远程主机的网络地址\*/char \_\_unused[20]; /\* 保留未使用\*/};

ut\_type有以下几种类型: EMPTY 此为空的记录。 RUN\_LVL 记录系统run－level的改变 BOOT\_TIME 记录系统开机时间 NEW\_TIME 记录系统时间改变后的时间 OLD\_TINE 记录当改变系统时间时的时间。 INIT\_PROCESS 记录一个由init衍生出来的进程。 LOGIN\_PROCESS 记录login进程。 USER\_PROCESS 记录一般进程。 DEAD\_PROCESS 记录一结束的进程。 ACCOUNTING 目前尚未使用。

exit\_status结构定义

struct exit\_status{short int e\_termination; /\*进程结束状态\*/short int e\_exit; /\*进程退出状态\*/};

timeval的结构定义请参考gettimeofday（）。相关常数定义如下: UT\_LINESIZE 32 UT\_NAMESIZE 32 UT\_HOSTSIZE 256

* 返回值

返回utmp 结构数据，如果返回NULL 则表示已无数据，或有错误发生。

getutent()调用成功后会返回一个指向 struct utmp 的指针，调用失败（包括 “未找到记录 ”的情况）则返回 NULL。该结构 utmp 分配在静态存储中，可能会被后续调用覆盖。

* 附加说明

getutent()在第一次调用时会打开utmp 文件，读取数据完毕后可使用endutent()来关闭该utmp文件。

* 范例

#include<utmp.h>main(){struct utmp \*u;while((u=getutent())){if(u->ut\_type = = USER\_PROCESS)printf(“%d %s %s %s \n”,u->ut\_type,u->ut\_user,u->ut\_line,u->ut\_host);}endutent();}

* 执行

/\* 表示有三个root账号分别登录/dev/pts/0，/dev/pts/1，/dev/pts/2 \*/7 root pts/07 root pts/17 root pts/2

## 7.19 getutid

​从utmp 文件中查找特定的记录。

* 相关函数

getutent，getutline

* 表头文件

#include<utmp.h>

* 定义函数

struct utmp \*getutid(struct utmp \*ut);

* 函数说明

getutid()用来从目前utmp 文件的读写位置逐一往后搜索参数ut指定的记录，如果ut->ut\_type 为RUN\_LVL，BOOT\_TIME，NEW\_TIME，OLD\_TIME 其中之一则查找与ut->ut\_type 相符的记录；若ut->ut\_type 为INIT\_PROCESS，LOGIN\_PROCESS，USER\_PROCESS或DEAD\_PROCESS其中之一，则查找与ut->ut\_id相符的记录。找到相符的记录便将该数据以utmp 结构返回。utmp结构请参考getutent()。

* 返回值

getutid()在成功时返回指向 struct utmp 的指针，在失败时（包括 “未找到记录 ”的情况）返回 NULL。该结构 utmp 分配在静态存储中，可能会被后续调用覆盖。

如果失败，这些函数将设置 errno 以指示错误。

* 范例

#include<utmp.h>main(){struct utmp ut,\*u;ut.ut\_type=RUN\_LVL;while((u= getutid(&ut))){printf(“%d %s %s %s\n”,u->ut\_type,u->ut\_user,u->ut\_line,u->ut\_host);}}

* 执行

1 runlevel –

## 7.20 getutline

​从utmp 文件中查找特定的记录。

* 相关函数

getutent，getutid，pututline

* 表头文件

#include<utmp.h>

* 定义函数

struct utmp \*getutline (struct utmp \*ut);

* 函数说明

getutline()用来从目前utmp文件的读写位置逐一往后搜索ut\_type为USER\_PROCESS 或LOGIN\_PROCESS 的记录，而且ut\_line 和ut->ut\_line 相符。找到相符的记录便将该数据以utmp 结构返回，utmp结构请参考getutent()。

* 返回值

getutline ()在成功时返回指向 struct utmp 的指针，在失败时（包括 “未找到记录 ”的情况）返回 NULL。该结构 utmp 分配在静态存储中，可能会被后续调用覆盖。

如果失败，这些函数将设置 errno 以指示错误。

* 范例

#include<utmp.h>main(){struct utmp ut,\*u;strcpy (ut.ut\_line,”pts/1”);while ((u=getutline(&ut))){printf(“%d %s %s %s \n”,u->ut\_type,u->ut\_user,u->ut\_line,u->ut\_host);}}

* 执行

7 root pts/1

## 7.21 initgroups

​初始化组清单。

* 相关函数

setgrent，endgrent

* 表头文件

#include<grp.h>#include<sys/types.h>

* 定义函数

int initgroups(const char \*user, gid\_t group);

* 函数说明

initgroups（）用来从组文件（/etc/group）中读取一项组数据，若该组数据的成员中有参数user时，便将参数group组识别码加入到此数据中。

user 参数必须为非空。

* 返回值

initgroups() 函数成功时返回 0。如果出错，则返回 -1，并设置 errno 表示出错。

## 7.22 pututline

​将utmp 记录写入文件。

* 相关函数

getutent，getutid，getutline

* 表头文件

#include<utmp.h>

* 定义函数

struct utmp \*pututline(struct utmp \*ut);

* 函数说明

pututline()用来将参数ut的utmp结构记录到utmp文件中。此函数会先用getutid()来取得正确的写入位置，如果没有找到相符的记录则会加入到utmp文件尾，utmp结构请参考getutent()。

* 返回值

成功时 pututline() 返回 ut，失败时返回 NULL。

如果失败，这些函数会设置 errno 以指示错误。

* 附加说明

需要有写入/var/run/utmp 的权限

* 范例

#include<utmp.h>main(){struct utmp ut;ut.ut\_type =USER\_PROCESS;ut.ut\_pid=getpid();strcpy(ut.ut\_user,”kids”);strcpy(ut.ut\_line,”pts/1”);strcpy(ut.ut\_host,”www.gnu.org”);pututline(&ut);}

* 执行

/\*执行范例后用指令who -l 观察\*/root pts/0 dec9 19:20kids pts/1 dec12 10:31(www.gnu.org)root pts/2 dec12 13:33

## 7.23 seteuid

​设置有效的用户识别码。

* 相关函数

setuid，setreuid，setfsuid

* 表头文件

#include<unistd.h>

* 定义函数

int seteuid(uid\_t euid);

* 函数说明

seteuid() 设置调用进程的有效用户 ID。非特权进程只能将有效用户 ID 设置为真实用户 ID、有效用户 ID 或保存的 setuser-ID。使用 “组 ”代替 “用户 ”的 setegid() 也是如此。在Linux下，seteuid(euid)相当于setreuid(-1,euid)。

* 返回值

成功时，返回 0。错误时，返回-1，并设置 errno 表示错误。

注意：在某些情况下，即使调用者是 UID 0，seteuid() 也会失败；忽略检查 seteuid() 的失败返回是一个严重的安全错误。

如果不检查 seteuid() 的失败返回，将是一个严重的安全错误。

## 7.24 setfsgid

​设置文件系统的组识别码。

* 相关函数

setuid，setreuid，seteuid，setfsuid

* 表头文件

#include< sys/fsuid.h>

* 定义函数

int setfsgid(uid\_t fsgid);

* 函数说明

setfsgid()用来重新设置目前进程的文件系统的组识别码。一般情况下，文件系统的组识别码(fsgid)与有效的组识别码(egid)是相同的。如果是超级用户调用此函数，参数fsgid 可以为任何值，否则参数fsgid必须为real/effective/saved的组识别码之一。

通常情况下，进程的文件系统组 ID 值与其有效组 ID 值相同。之所以如此，是因为每当进程的有效组 ID 发生变化时，内核也会更改文件系统组 ID，使其与有效组 ID 的新值相同。进程可以使用 setfsgid() 将其文件系统组 ID 更改为 fsgid 中给出的值，从而使其文件系统组 ID 的值偏离其有效组 ID。

只有当调用者是超级用户或 fsgid 与调用者的实际组 ID、有效组 ID、保存的设置组 ID 或当前文件系统用户 ID 一致时，setfsgid() 才会成功。

* 返回值

无论成功或失败，该调用都会返回调用者之前的文件系统组 ID。

* 附加说明

此函数为Linux特有。

* 错误代码

EPERM 权限不够，无法完成设置。

## 7.25 setfsuid

​设置文件系统的用户识别码。

* 相关函数

setuid，setreuid，seteuid，setfsgid

* 表头文件

#include <sys/fsuid.h>

* 定义函数

int setfsuid(uid\_t fsuid);

* 函数说明

setfsuid()用来重新设置目前进程的文件系统的用户识别码。一般情况下，文件系统的用户识别码(fsuid)与有效的用户识别码(euid)是相同的。如果是超级用户调用此函数，参数fsuid可以为任何值，否则参数fsuid必须为real/effective/saved的用户识别码之一。

通常情况下，进程的文件系统用户 ID 值与其有效用户 ID 值相同。之所以如此，是因为每当进程的有效用户 ID 发生变化时，内核也会改变文件系统用户 ID，使其与有效用户 ID 的新值相同。进程可以通过使用 setfsuid() 将其文件系统用户 ID 更改为 fsuid 中给出的值，从而使其文件系统用户 ID 的值偏离其有效用户 ID。

对 setfsuid() 和 setfsgid(2) 的明确调用通常只在 Linux NFS 服务器等程序中使用，这些程序需要更改文件访问时使用的用户和组 ID，而实际和有效的用户和组 ID 并没有相应的改变。更改 NFS 服务器等程序的正常用户 ID 是（曾经）一个安全漏洞，会使其受到不必要的信号干扰。(不过，这个问题已经成为历史，见下文）。

只有当调用者是超级用户，或者 fsuid 与调用者的真实用户 ID、有效用户 ID、保存的 set-user-ID 或当前文件系统用户 ID 一致时，setfsuid() 才会成功。。

* 返回值

无论成功或失败，该调用都会返回调用者之前的文件系统用户 ID。

* 附加说明

此函数为Linux特有

* 错误代码

EPERM 权限不够，无法完成设置。

## 7.26 setgid

​设置真实的组识别码。

* 相关函数

getgid，setregid，getegid，setegid

* 表头文件

#include<unistd.h>

* 定义函数

int setgid(gid\_t gid);

* 函数说明

setgid()用来将目前进程的真实组识别码(real gid)设成参数gid值。如果是以超级用户身份执行此调用，则real、effective与savedgid都会设成参数gid。

setgid() 设置调用进程的有效组 ID。如果调用进程是特权进程（更准确地说：在其用户名空间中具有 CAP\_SETGID 功能），则也会设置实际 GID 和保存的 set-group-ID。

在 Linux 下，setgid() 与 POSIX 版本的 \_POSIX\_SAVED\_IDS 功能一样。这允许非 set-userID-root 的 set-group-ID 程序放弃所有组权限，执行一些无权限的工作，然后以安全的方式重新启用原始有效组 ID

* 返回值

成功时，返回 0。如果出错，则返回-1，并设置 errno 表示错误。

* 错误代码

EPERM 并非以超级用户身份调用，而且参数gid 并非进程的effective gid或saved gid值之一。

## 7.27 setgrent

​从头读取组文件中的组数据。

* 相关函数

getgrent，endgrent

* 表头文件

#include<grp.h>#include<sys/types.h>

* 定义函数

void setgrent(void);

* 函数说明

getgrent() 函数返回一个指向结构体的指针，该结构体包含组数据库（如本地组文件 /etc/group、NIS 和 LDAP）中记录的细分字段。

第一次调用 getgrent()时，它会返回第一个条目；此后，它会返回连续的条目。

setgrent() 函数会倒退到组数据库的起始位置，以便重复扫描。endgrent() 函数用于在完成所有处理后关闭分组数据库。

* 附加说明

请参考setpwent()。

## 7.28 setgroups

​设置组代码。

* 相关函数

initgroups，getgroup，getgid，setgid

* 表头文件

#include<grp.h>

* 定义函数

int setgroups(size\_t size, const gid\_t \*list);

* 函数说明

setgroups()用来将list 数组中所标明的组加入到目前进程的组设置中。参数size为list()的gid\_t数目，最大值为NGROUP(32)。

setgroups() 为调用进程设置补充组 ID。需要适当的权限（见下文 EPERM 错误的说明）。size参数指定 list 指向的缓冲区中补充组 ID 的数量。进程可以通过调用 setgroups(0, NULL) 删除所有辅助组；

* 返回值

成功时，setgroups() 返回 0。错误时，返回-1，并设置 errno 表示错误。

* 错误代码

EFAULT 参数list数组地址不合法。 EPERM 权限不足，必须是root权限 EINVAL 参数size值大于NGROUP(32)。

## 7.29 setpwent

​从头读取密码文件中的账号数据。

* 相关函数

getpwent，endpwent

* 表头文件

#include<pwd.h>#include<sys/types.h>

* 定义函数

void setpwent(void);

* 函数说明

setpwent()用来将getpwent()的读写地址指回密码文件开头。

* 范例

#include<pwd.h>#include<sys/types.h>main(){struct passwd \*user;int i;for(i=0;i<4;i++){user=getpwent();printf(“%s :%d :%d :%s:%s:%s\n”,user->pw\_name,user->pw\_uid,user->pw\_gid,user->pw\_gecos,user->pw\_dir,user->pw\_shell);}setpwent();user=getpwent();printf(“%s :%d :%d :%s:%s:%s\n”,user->pw\_name,user->pw\_uid,user->pw\_gid,user->pw\_gecos,user->pw\_dir,user->pw\_shell);endpwent();}

* 执行

root:0:0:root:/root:/bin/bashbin:1:1:bin:/bindaemon:2:2:daemon:/sbinadm:3:4:adm:/var/admroot:0:0:root:/root:/bin/bash

## 7.30 setregid

​设置真实及有效的组识别码。

* 相关函数

setgid，setegid，setfsgid

* 表头文件

#include<unistd.h>

* 定义函数

int setregid(gid\_t rgid, gid\_t egid);

* 函数说明

setregid()用来将参数rgid设为目前进程的真实组识别码，将参数egid设置为目前进程的有效组识别码。如果参数rgid或egid值为-1，则对应的识别码不会改变。

setreuid() 设置调用进程的实际和有效用户 ID。

如果为真实用户 ID 或有效用户 ID 提供-1 的值，系统会强制保持该 ID 不变。

无权限进程只能将有效用户 ID 设置为真实用户 ID、有效用户 ID 或保存的 set-user-ID。

非特权用户只能将真实用户 ID 设置为真实用户 ID 或有效用户 ID。

如果设置了真实用户 ID（即 ruid 不为-1）或有效用户 ID 被设置为不等于先前真实用户 ID 的值，则保存的 set-user-ID 将被设置为新的有效用户 ID。

与此完全类似，setregid() 也会设置调用进程的真实和有效组 ID，而且上述所有操作都适用于 “组 ”而非 “用户”。

* 返回值

成功时，返回 0。错误时，返回-1，并设置 errno 表示错误。

注意：在某些情况下，即使调用者是 UID 0，setreuid() 也可能失败；如果不检查 setreuid() 的失败返回，则是严重的安全错误。

## 7.31 setreuid

​设置真实及有效的用户识别码。

* 相关函数

setuid，seteuid，setfsuid

* 表头文件

#include<unistd.h>

* 定义函数

int setreuid(uid\_t ruid, uid\_t euid);

* 函数说明

setreuid()用来将参数ruid 设为目前进程的真实用户识别码，将参数euid 设置为目前进程的有效用户识别码。如果参数ruid 或euid值为-1，则对应的识别码不会改变。

setreuid() 设置调用进程的实际和有效用户 ID。

如果为真实用户 ID 或有效用户 ID 提供-1 的值，系统会强制保持该 ID 不变。

无权限进程只能将有效用户 ID 设置为真实用户 ID、有效用户 ID 或保存的 set-user-ID。

非特权用户只能将真实用户 ID 设置为真实用户 ID 或有效用户 ID。

如果设置了真实用户 ID（即 ruid 不为-1）或有效用户 ID 被设置为不等于先前真实用户 ID 的值，则保存的 set-user-ID 将被设置为新的有效用户 ID。

* 返回值

成功时，返回 0。错误时，返回-1，并设置 errno 表示错误。

注意：在某些情况下，即使调用者是 UID 0，setreuid() 也可能失败；如果不检查 setreuid() 的失败返回，则是严重的安全错误。

* 附加说明

请参考setuid（）。

## 7.32 setuid

​设置真实的用户识别码。

* 相关函数

getuid，setreuid，seteuid，setfsuid

* 表头文件

#include<unistd.h>

* 定义函数

int setuid(uid\_t uid);

* 函数说明

setuid() 设置调用进程的有效用户 ID。如果调用进程是特权进程（更准确地说：如果进程在其用户名空间中具有CAP\_SETUID 功能），则还会设置实际 UID 和保存的 set-user-ID。

在 Linux 下，setuid() 的实现方式与 POSIX 版本一样，使用\_POSIX\_SAVED\_IDS 功能。这允许设置用户 ID（root 以外）的程序放弃所有用户权限，执行一些无权限的工作，然后以安全的方式重新使用原来的有效用户 ID。

如果用户是 root 或程序是 set-user-ID-root，则必须特别小心：setuid() 会检查调用者的有效用户 ID，如果是超级用户，则所有与进程相关的用户 ID 都会被设置为 uid。如果是超级用户，所有与进程相关的用户 ID 都会被设置为 uid。这样做之后，程序就不可能重新获得 root 权限。

因此，set-user-ID-root 程序如果想暂时放弃 root 权限，以非特权用户的身份运行，然后重新获得 root 权限，就不能使用 setuid()。您可以使用 seteuid(2) 来实现这一目的。

* 返回值

成功时，返回 0。错误时，返回-1，并设置 errno 表示错误。

注意：在某些情况下，即使调用者是 UID 0，setuid() 也会失败；如果不检查 setuid() 的失败返回，则是严重的安全错误。

* 附加说明

一般在编写具setuid root的程序时，为减少此类程序带来的系统安全风险，在使用完root权限后建议马上执行setuid(getuid());来抛弃root权限。此外，进程uid和euid不一致时Linux系统将不会产生core dump。

## 7.33 setutent

​从头读取utmp 文件中的登录数据。

* 相关函数

getutent，endutent

* 表头文件

#include<utmp.h>

* 定义函数

void setutent(void);

* 函数说明

setutent()用来将getutent()的读写地址指回utmp文件开头。一般来说，最好在调用其他函数之前调用该函数。

* 附加说明

请参考setpwent()或setgrent()。

## 7.34 utmpname

​设置utmp 文件路径。

* 相关函数

getutent，getutid，getutline，setutent，endutent，pututline

* 表头文件

#include<utmp.h>

* 定义函数

void utmpname(const char \*file);

* 函数说明

utmpname()用来设置utmp文件的路径，以提供utmp相关函数的存取路径。如果没有使用utmpname()则默认utmp文件路径为/var/run/utmp。

# 第八章 数据结构及算法篇

## 8.1 crypt

​将密码或数据编码。

* 相关函数

Getpass

* 表头文件

#define \_XOPEN\_SOURCE#include<unistd.h>

* 定义函数

char \*crypt(const char \*key, const char \*salt);

* 函数说明

crypt() 是密码散列函数。它以数据加密标准算法为基础，并有所变化，目的之一是阻止使用硬件实现密钥搜索。key 是用户输入的密码。salt 是一个从 [a-zA-Z0-9./] 中选择的双字符串。该字符串用于以 4096 种不同方式之一扰乱算法。

取密钥前 8 个字符中每个字符的最低 7 位，就得到了 56 位密钥。这个 56 位密钥用于重复加密一个常量字符串（通常是由全零组成的字符串）。返回值指向散列密码，即一系列 13 个可打印的 ASCII 字符（前两个字符代表盐本身）。返回值指向静态数据，每次调用都会覆盖其内容。

警告：密钥空间由 256 个相等于 7.2e16 的可能值组成。使用大规模并行计算机可以对该密钥空间进行穷举搜索。破解等软件可以搜索人类通常用于密码的那部分密钥空间。因此，在选择密码时至少应避免使用常用词和名称。建议使用passwd 程序，在选择过程中检查可破解的密码。

* 返回值

成功时，返回指向散列密码的指针。如果出错，则返回 NULL。

* 附加说明

使用GCC编译时需加-lcrypt。

* 范例

#include<unistd.h>main(){char passwd[13];char \*key;char slat[2];key= getpass(“Input First Password:”);slat[0]=key[0];slat[1]=key[1];strcpy(passwd,crypt(key slat));key=getpass(“Input Second Password:”);slat[0]=passwd[0];slat[1]=passwd[1];printf(“After crypt(),1st passwd :%s\n”,passwd);printf(“After crypt(),2nd passwd:%s \n”,crypt(key slat));}

* 执行

Input First Password: /\* 输入test，编码后存于passwd[ ] \*/Input Second Password /\*输入test，密码相同编码后也会相同\*/After crypt () 1st Passwd : teH0wLIpW0gyQAfter crypt () 2nd Passwd : teH0wLIpW0gyQ

## 8.2 bsearch

​二元搜索。

* 相关函数

Qsort

* 表头文件

#include<stdlib.h>

* 定义函数

void \*bsearch(const void \*key, const void \*base, size\_t nmemb, size\_t size, int (\*compar)(const void \*, const void \*));

* 函数说明

bsearch()利用二元搜索从排序好的数组中查找数据, 也就是说，所有比较小于关键目标的元素必须出现于所有比较等于的元素之前，而且所有比较等于关键目标的元素要出现于所有比较大于关键目标的元素之前。参数key指向欲查找的关键数据，参数base指向要被搜索的数组开头地址，参数nmemb 代表数组中的元素数量，每一元素的大小则由参数size决定，最后一项参数compar 为一函数指针，这个函数用来判断两个元素之间的大小关系，若传给compar 的第一个参数所指的元素数据大于第二个参数所指的元素数据则必须回传大于0 的值，两个元素数据相等则回传0。用 compar所指向的函数比较元素。若数组未依照与 compar标准相同的相对于 \*key 的升序划分，则行为未定义。

* 附加说明

找到关键数据则返回找到的地址，如果在数组中找不到关键数据则返回NULL。

* 范例

#include<stdio.h>#include<stdlib.h>#define NMEMB 5#define SIZE 10int compar(const void \*a,const void \*b){return (strcmp((char \*)a,(char \*)b));}main(){char data[50][size]={“linux”,”freebsd”,”solaris”,”sunos”,”windows”};char key[80],\*base ,\*offset;int i, nmemb=NMEMB,size=SIZE;while(1){printf(“>”);fgets(key,sizeof(key),stdin);key[strlen(key)-1]=’\0’;if(!strcmp(key,”exit”))break;if(!strcmp(key,”list”)){for(i=0;i<nmemb;i++)printf(“%s\n”,data[i]);continue;}base = data[0];qsort(base,nmemb,size,compar);offset = (char \*) bsearch(key,base,nmemb,size,compar);if( offset = =NULL){printf(“%s not found!\n”,key);strcpy(data[nmemb++],key);printf(“Add %s to data array\n”,key);}else{printf(“found: %s \n”,offset);}}}

* 执行

>hello /\*输入hello字符串\*/hello not found! /\*找不到hello 字符串\*/add hello to data array /\*将hello字符串加入\*/>.list /\*列出所有数据\*/freebsdlinuxsolarissunoswindowshello>hellofound: hello

## 8.3 lfind

​线性搜索。

* 相关函数

lsearch

* 表头文件

#include<stdlib.h>

* 定义函数

void \*lfind(const void \*key, const void \*base, size\_t \*nmemb, size\_t size, int (\*compar)(const void \* , const void \*));

* 函数说明

lfind()利用线性搜索在数组中从头至尾一项项查找数据。参数key指向欲查找的关键数据，参数base指向要被搜索的数组开头地址，参数nmemb代表数组中的元素数量，每一元素的大小则由参数size决定，最后一项参数compar为一函数指针，这个函数用来判断两个元素是否相同，若传给compar 的异地个参数所指的元素数据和第二个参数所指的元素数据相同时则返回0，两个元素数据不相同则返回非0值。Lfind()与lsearch()不同点在于，当找不到关键数据时lfind()仅会返回NULL，而不会主动把该笔数据加入数组尾端。

* 返回值

找到关键数据则返回找到的该笔元素的地址，如果在数组中找不到关键数据则返回空指针(NULL)。

* 附加说明

使用GCC编译时需加-lcrypt。

* 范例

参考lsearch()。

## 8.4 lsearch

​线性搜索。

* 相关函数

lfind

* 表头文件

#include<stdlib.h>

* 定义函数

void \*lsearch(const void \* key, const void \*base, size\_t \*nmemb, size\_t size, int (\*compar)(const void \*, const void \*));

* 函数说明

lsearch()利用线性搜索在数组中从头至尾一项项查找数据。参数key指向欲查找的关键数据，参数base指向要被搜索的数组开头地址，参数nmemb 代表数组中的元素数量，每一元素的大小则由参数size 决定，最后一项参数compar 为一函数指针，这个函数用来判断两个元素是否相同，若传给compar 的第一个参数所指的元素数据和第二个参数所指的元素数据相同时则返回0，两个元素数据不相同则返回非0 值。如果lsearch（）找不到关键数据时会主动把该项数据加入数组里。

* 返回值

找到关键数据则返回找到的该笔元素的四肢，如果在数组中找不到关键数据则将此关键数据加入数组，再把加入数组后的地址返回。

* 范例

#include<stdio.h>#include<stdlib.h>#define NMEMB 50#define SIZE 10int compar (comst void \*a,const void \*b){return (strcmp((char \*) a, (char \*) b));}main(){char data[NMEMB][SIZE]={“Linux”,”freebsd”,”solzris”,”sunos”,”windows”};char key[80],\*base,\*offset;int i, nmemb=NMEMB,size=SIZE;for(i=1;i<5;i++){fgets(key,sizeof9key),stdin);key[strlen(key)-1]=’\0’;base = data[0];offset = (char \*)lfind(key,base,&nmemb,size,compar);if(offset ==NULL){printf(“%s not found!\n”,key);offset=(char \*) lsearch(key,base,&nmemb,size,compar);printf(“Add %s to data array\n”,offset);}else{printf(“found : %s \n”,offset);}}}

* 执行

linuxfound:linuxos/2os/2 not found!add os/2 to data arrayos/2found:os/2

## 8.5 qsort

​利用快速排序法排列数组。

* 相关函数

Bsearch

* 表头文件

#include<stdlib.h>

* 定义函数

void qsort(void \*base, size\_t nmemb, size\_t size, int (\*compar)(const void \*, const void \*));

* 函数说明

参数base指向欲排序的数组开头地址，参数nmemb代表数组中的元素数量，每一元素的大小则由参数size决定，最后一项参数compar为一函数指针，这个函数用来判断两个元素间的大小关系，若传给compar的第一个参数所指的元素数据大于第二个参数所指的元素数据则必须回传大于零的值，两个元素数据相等则回传0。

* 范例

#define nmemb 7#include <stdlib.h>int compar (const void \*a ,const void \*b){int \*aa=(int \* ) a,\*bb = (int \* )b;if( \* aa >\* bb)return 1;if( \* aa == \* bb) return 0;if( \* aa < \*bb) return -1;}main( ){int base[nmemb]={ 3,102,5,-2,98,52,18};int i;for ( i=0; i<nmemb;i++)printf(“%d “,base[i]);printf(“\n”);qsort(base,nmemb,sizeof(int),compar);for(i=0;i<nmemb;i++)printf(“%d”base[i]);printf(“\n”);}

* 执行

3 102 5 -2 98 52 18-2 3 5 18 52 98 102

## 8.6 rand

​产生随机数。

* 相关函数

srand，random，srandom

* 表头文件

#include<stdlib.h>

* 定义函数

int rand(void);

* 函数说明

rand() 函数返回一个伪随机整数，其范围在 0 到 RAND\_MAX 之间（包括 RAND\_MAX）（即数学范围 [0, RAND\_MAX]）。

srand() 函数将其参数设置为一个新的伪随机整数序列的种子。将由 rand() 返回的新伪随机整数序列的种子。使用相同的种子值调用 srand() 可以重复这些序列。如果没有提供种子值，rand() 函数将自动以 1 作为种子值。

函数 rand() 不是可重入的，因为它使用的隐藏状态在每次调用时都会被修改。这可能只是下一次调用时使用的种子值，也可能是更复杂的内容。为了在线程应用程序中获得可重现的行为，必须将这种状态显式化；

* 返回值

返回0至RAND\_MAX之间的随机数值，RAND\_MAX定义在stdlib.h，其值为2147483647。

* 范例

/\* 产生介于1 到10 间的随机数值，此范例未设随机数种子，完整的随机数产生请参考srand（）\*/#include<stdlib.h>main(){int i,j;for(i=0;i<10;i++){j=1+(int)(10.0\*rand()/(RAND\_MAX+1.0));printf(“%d “,j);}}

* 执行

9 4 8 8 10 2 4 8 3 69 4 8 8 10 2 4 8 3 6

## 8.7 srand

​设置随机数种子。

* 相关函数

rand，random srandom

* 表头文件

#include<stdlib.h>

* 定义函数

void srand(unsigned int seed);

* 函数说明

srand()用来设置rand()产生随机数时的随机数种子。参数seed必须是个整数，通常可以利用geypid()或time(0)的返回值来当做seed。如果每次seed都设相同值，rand()所产生的随机数值每次就会一样。

* 范例

/\* 产生介于1 到10 间的随机数值，此范例与执行结果可与rand（）参照\*/#include<time.h>#include<stdlib.h>main(){int i,j;srand((int)time(0));for(i=0;i<10;i++){j=1+(int)(10.0\*rand()/(RAND\_MAX+1.0));printf(“ %d “,j);}}

* 执行

5 8 8 8 10 2 10 8 9 92 9 7 4 10 3 2 10 8 7

# 第九章 文件操作篇

## 9.1 close

​关闭文件。

* 相关函数

open，fcntl，shutdown，unlink，fclose

* 表头文件

#include<unistd.h>

* 定义函数

int close(int fd);

* 函数说明

close() 关闭文件描述符，使其不再指向任何文件，并可被重新使用。

无论使用哪个文件描述符获得锁，与该文件关联并为进程所有的任何记录锁（参见 fcntl(2)）都会被移除。

这会带来一些不幸的后果，因此在使用咨询记录锁时应格外小心。请参见 fcntl(2)，了解有关风险和后果的讨论，以及开放文件描述符锁（可能是首选）。

如果 fd 是最后一个引用底层打开文件描述符的文件描述符（参见 open(2)），那么与打开文件描述符相关的资源将被释放；如果该文件描述符是最后一个引用已被 unlink(2) 删除的文件的文件描述符，那么该文件将被删除。

* 返回值

close() 成功时返回 0。如果出错，则返回-1，并设置 errno 表示错误。

* 错误代码

EBADF 参数fd 非有效的文件描述词或该文件已关闭。

* 附加说明

虽然在进程结束时，系统会自动关闭已打开的文件，但仍建议自行关闭文件，并确实检查返回值。

* 范例

参考open()

## 9.2 creat

​建立文件。

* 相关函数

read，write，fcntl，close，link，stat，umask，unlink，fopen

* 表头文件

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

* 定义函数

int creat(const char \*pathname, mode\_t mode);

* 函数说明

参数pathname指向欲建立的文件路径字符串。Creat()相当于使用下列的调用方式调用open() open(const char \* pathname ,(O\_CREAT|O\_WRONLY|O\_TRUNC));

* 返回值

关于参数mode请参考open（）函数。

* 错误代码

creat()会返回新的文件描述词，若有错误发生则会返回-1，并把错误代码设给errno。EEXIST 参数pathname所指的文件已存在。 EACCESS 参数pathname 所指定的文件不符合所要求测试的权限 EROFS 欲打开写入权限的文件存在于只读文件系统内 EFAULT 参数pathname 指针超出可存取的内存空间 EINVAL 参数mode 不正确。 ENAMETOOLONG 参数pathname太长。 ENOTDIR 参数pathname为一目录 ENOMEM 核心内存不足 ELOOP 参数pathname有过多符号连接问题。 EMFILE 已达到进程可同时打开的文件数上限 ENFILE 已达到系统可同时打开的文件数上限

* 附加说明

creat()无法建立特别的装置文件，如果需要请使用mknod()。

* 范例

参考open()

## 9.3 dup

​复制文件描述词。

* 相关函数

open，close，fcntl，dup2

* 表头文件

#include<unistd.h>

* 定义函数

int dup(int oldfd);

* 函数说明

dup()用来复制参数oldfd所指的文件描述词，并将它返回。例如，当利用lseek()对某个文件描述词作用时，另一个文件描述词的读写位置也会随着改变。不过，文件描述词之间并不共享close-on-exec旗标。

系统调用 dup() 会分配一个新的文件描述符，该描述符指向与 oldfd 描述符相同的打开文件描述。此新的文件描述词和参数oldfd指的是同一个文件，共享所有的锁定、读写位置和各项权限或旗标。(有关打开文件描述的解释，请参阅 open(2)）。新文件描述符的编号保证是调用进程中未使用的最低编号文件描述符。成功返回后，新旧文件描述符可以互换使用。

例如，如果在其中一个文件描述符上使用 lseek(2) 修改了文件偏移量，另一个文件描述符的偏移量也会随之改变。

两个文件描述符不共享文件描述符标志（执行时关闭标志）。复制描述符的执行时关闭标志 (FD\_CLOEXEC; 参见 fcntl(2))处于关闭状态。

* 返回值

当复制成功时，则返回最小及尚未使用的文件描述词。若有错误则返回-1，errno会存放错误代码。错误代码EBADF参数fd非有效的文件描述词，或该文件已关闭。

## 9.4 dup2

​复制文件描述词。

* 相关函数

open，close，fcntl，dup

* 表头文件

#include<unistd.h>

* 定义函数

int dup2(int odlfd, int newfd);

* 函数说明

dup2() 系统调用执行与 dup() 相同的任务，但它不使用编号最低的未使用文件描述符，而是使用 newfd 中指定的文件描述符编号。换句话说，文件描述符 newfd 会进行调整，使其现在指向与 oldfd 相同的打开文件描述。

如果文件描述符 newfd 之前是打开的，那么在重新使用之前会关闭它；关闭过程是静默进行的（也就是说，关闭过程中的任何错误都不会被 dup2() 报告）。

关闭和重新使用文件描述符 newfd 的步骤是以原子方式执行的。这一点很重要，因为如果试图使用 close(2) 和 dup() 实现等效功能，就会出现竞赛条件，即 newfd 可能会在这两个步骤之间被重复使用。发生这种重复使用的原因可能是主程序被分配文件描述符的信号处理程序中断，也可能是并行线程分配了文件描述符。

请注意以下几点：

- 如果 oldfd 不是有效的文件描述符，则调用失败，newfd 也不会关闭。

- 如果 oldfd 是一个有效的文件描述符，且 newfd 的值与 oldfd 相同，则 dup2() 不会执行任何操作，而是返回 newfd。

* 返回值

当复制成功时，则返回最小及尚未使用的文件描述词。若有错误则返回-1，errno会存放错误代码。

* 错误代码

EBADF 参数fd 非有效的文件描述词，或该文件已关闭

* 附加说明

dup2()相当于调用fcntl(oldfd，F\_DUPFD，newfd)；请参考fcntl()。

## 9.5 fcntl

​文件描述词操作。

* 相关函数

open，flock

* 表头文件

#include<unistd.h>#include<fcntl.h>

* 定义函数

int fcntl(int fd, int cmd);int fcntl(int fd, int cmd, long arg);int fcntl(int fd, int cmd, struct flock \*lock);

* 函数说明

fcntl()用来操作文件描述词的一些特性。参数fd代表欲设置的文件描述词，参数cmd代表欲操作的指令。 有以下几种情况:F\_DUPFD用来查找大于或等于参数arg的最小且仍未使用的文件描述词，并且复制参数fd的文件描述词。执行成功则返回新复制的文件描述词。请参考dup2()。F\_GETFD取得close-on-exec旗标。若此旗标的FD\_CLOEXEC位为0，代表在调用exec()相关函数时文件将不会关闭。 F\_SETFD 设置close-on-exec 旗标。该旗标以参数arg 的FD\_CLOEXEC位决定。 F\_GETFL 取得文件描述词状态旗标，此旗标为open（）的参数flags。 F\_SETFL 设置文件描述词状态旗标，参数arg为新旗标，但只允许O\_APPEND、O\_NONBLOCK和O\_ASYNC位的改变，其他位的改变将不受影响。 F\_GETLK 取得文件锁定的状态。 F\_SETLK 设置文件锁定的状态。此时flcok 结构的l\_type 值必须是F\_RDLCK、F\_WRLCK或F\_UNLCK。如果无法建立锁定，则返回-1，错误代码为EACCES 或EAGAIN。 F\_SETLKW F\_SETLK 作用相同，但是无法建立锁定时，此调用会一直等到锁定动作成功为止。若在等待锁定的过程中被信号中断时，会立即返回-1，错误代码为EINTR。参数lock指针为flock 结构指针，定义如下

struct flcok{short int l\_type; /\* 锁定的状态\*/short int l\_whence;/\*决定l\_start位置\*/off\_t l\_start; /\*锁定区域的开头位置\*/off\_t l\_len; /\*锁定区域的大小\*/pid\_t l\_pid; /\*锁定动作的进程\*/};

l\_type 有三种状态: F\_RDLCK 建立一个供读取用的锁定 F\_WRLCK 建立一个供写入用的锁定 F\_UNLCK 删除之前建立的锁定 l\_whence 也有三种方式: SEEK\_SET 以文件开头为锁定的起始位置。 SEEK\_CUR 以目前文件读写位置为锁定的起始位置 SEEK\_END 以文件结尾为锁定的起始位置。

* 返回值

成功则返回0，若有错误则返回-1，错误原因存于errno.

## 9.6 flock

​锁定文件或解除锁定。

* 相关函数

open,fcntl

* 表头文件

#include<sys/file.h>

* 定义函数

int flock(int fd, int operation);

* 函数说明

flock()会依参数operation所指定的方式对参数fd所指的文件做各种锁定或解除锁定的动作。此函数只能锁定整个文件，无法锁定文件的某一区域。

如果其他进程持有不兼容的锁，调用 flock() 可能会阻塞。

要提出非阻塞请求，请在上述任何操作中加入 LOCK\_NB（通过 ORing）。一个文件不能同时拥有共享锁和独占锁。由 flock() 创建的锁与打开的文件描述相关联（参见 open(2)）。这意味着重复的文件描述符（例如通过 fork(2) 或 dup(2) 创建的文件描述符）会指向同一个锁，并且可以使用这些文件描述符修改或释放该锁。此外，在这些重复文件描述符中的任何一个上进行显式 LOCK\_UN 操作，或者关闭所有此类文件描述符后，锁就会被释放。如果进程使用 open(2)（或类似方法）为同一个文件获取了多个文件描述符，那么 flock() 会独立处理这些文件描述符。如果试图使用其中一个文件描述符锁定文件，可能会被调用进程已经通过另一个文件描述符设置的锁所拒绝。一个进程只能在一个文件上持有一种锁（共享锁或独占锁）。对已锁定文件的后续 flock() 调用将把现有锁转换为新锁模式。通过 flock() 创建的锁会在执行 execve(2) 时保留。

无论文件以何种模式打开，都可以在文件上设置共享锁或独占锁

* 返回值

返回0表示成功，若有错误则返回-1，错误代码存于errno。

* 参数

operation有下列四种情况: LOCK\_SH 建立共享锁定。多个进程可同时对同一个文件作共享锁定。 LOCK\_EX 建立互斥锁定。一个文件同时只有一个互斥锁定。 LOCK\_UN 解除文件锁定状态。 LOCK\_NB 无法建立锁定时，此操作可不被阻断，马上返回进程。通常与LOCK\_SH或LOCK\_EX 做OR(|)组合。 单一文件无法同时建立共享锁定和互斥锁定，而当使用dup()或fork()时文件描述词不会继承此种锁定。

## 9.7 fsync

​将缓冲区数据写回磁盘。

* 相关函数

Sync

* 表头文件

#include<unistd.h>

* 定义函数

int fsync(int fd);

* 函数说明

fsync()负责将参数fd所指的文件数据，由系统缓冲区写回磁盘，以确保数据同步。

fsync()会将文件描述符 fd 所引用文件的所有已修改内核数据（即已修改的缓冲缓存页）传输（“刷新”）到磁盘设备（或其他永久存储设备），这样即使系统崩溃或重启，也能检索到所有已更改的信息。这包括写入或刷新磁盘缓存（如果存在）。

调用会阻塞，直到设备报告传输已完成。

除了刷新文件数据外，fsync() 还会刷新与文件相关的元数据信息（参见 inode(7)）。

调用 fsync() 不一定能确保包含文件的目录中的条目也已到达磁盘。为此，还需要在目录的文件描述符上明确调用 fsync()。

* 返回值

成功时，这些系统调用返回 0。如果出错，则返回-1，并设置 errno 表示错误。

## 9.8 lseek

​移动文件的读写位置。

* 相关函数

dup，open，fseek

* 表头文件

#include<sys/types.h>#include<unistd.h>

* 定义函数

off\_t lseek(int fildes, off\_t offset, int whence);

* 函数说明

每一个已打开的文件都有一个读写位置，当打开文件时通常其读写位置是指向文件开头，若是以附加的方式打开文件(如O\_APPEND)，则读写位置会指向文件尾。当read()或write()时，读写位置会随之增加，lseek()便是用来控制该文件的读写位置。参数fildes 为已打开的文件描述词，参数offset 为根据参数whence来移动读写位置的位移数。

* 返回值

成功完成后，lseek() 返回结果偏移位置，以从文件开头算起的字节数为单位。如果出错，则返回值 (off\_t) -1, 并设置 errno 以指示错误。

* 参数

whence为下列其中一种: SEEK\_SET 参数offset即为新的读写位置。 SEEK\_CUR 以目前的读写位置往后增加offset个位移量。 SEEK\_END 将读写位置指向文件尾后再增加offset个位移量。 当whence 值为SEEK\_CUR 或SEEK\_END时，参数offet允许负值的出现。 下列是教特别的使用方式: 1) 欲将读写位置移到文件开头时:lseek（int fildes,0,SEEK\_SET）； 2) 欲将读写位置移到文件尾时:lseek（int fildes，0,SEEK\_END）； 3) 想要取得目前文件位置时:lseek（int fildes，0,SEEK\_CUR）；

* 附加说明

Linux系统不允许lseek（）对tty装置作用，此项动作会令lseek（）返回ESPIPE。

* 范例

参考本函数说明

## 9.9 mkstemp

​建立唯一的临时文件。

* 相关函数

mktemp

* 表头文件

#include<stdlib.h>

* 定义函数

int mkstemp(char \*template);

* 函数说明

mkstemp()用来建立唯一的临时文件。参数template 所指的文件名称字符串中最后六个字符必须是XXXXXX, 这些字符将被替换为一个字符串，使文件名具有唯一性。由于模板会被修改，因此它不能是字符串常量，而应声明为字符数组。Mkstemp()会以可读写模式和0600 权限来打开该文件，如果该文件不存在则会建立该文件。打开该文件后其文件描述词会返回。文件顺利打开后返回可读写的文件描述词。若果文件打开失败则返回NULL，并把错误代码存在errno 中。

文件创建时的权限为 0600，即只允许所有者读写。返回的文件描述符提供对文件的读写访问权限。文件使用 open(2) O\_EXCL 标志打开，保证调用者就是创建文件的进程。

mkostemp() 函数类似于 mkstemp()，不同之处在于可以在标志中指定以下与 open(2) 意义相同的位： O\_APPEND、O\_CLOEXEC 和 O\_SYNC。请注意，在创建文件时，mkostemp() 会在给 open(2) 的 flags 参数中包含 O\_RDWR、O\_CREAT 和 O\_EXCL；在给 mkostemp() 的 flags 参数中包含这些值是不必要的，在某些系统上会产生错误。

mkstemps() 函数与 mkstemp() 类似，只是模板中的字符串包含 suffixlen 字符的后缀。因此，模板的形式为 prefixXXXXXXsuffix，字符串 XXXXXX 的修改与 mkstemp() 相同。

* 错误代码

EINVAL 参数template 字符串最后六个字符非XXXXXX。EEXIST 无法建立临时文件。

* 附加说明

参数template所指的文件名称字符串必须声明为数组，如:char template[ ] =”template-XXXXXX”;千万不可以使用下列的表达方式char \*template = “template-XXXXXX”;

* 范例

#include<stdlib.h>main( ){int fd;char template[ ]=”template-XXXXXX”;fd=mkstemp(template);printf(“template = %s\n”,template);close(fd);}

* 执行

template = template-lgZcbo

## 9.10 open

​打开文件。

* 相关函数

read，write，fcntl，close，link，stat，umask，unlink，fopen

* 表头文件

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

* 定义函数

int open( const char \*pathname, int flags);int open( const char \*pathname, int flags, mode\_t mode);

* 函数说明

参数pathname 指向欲打开的文件路径字符串。

下列是参数flags 所能使用的旗标:

O\_RDONLY 以只读方式打开文件

O\_WRONLY 以只写方式打开文件

O\_RDWR 以可读写方式打开文件。

上述三种旗标是互斥的，也就是不可同时使用，但可与下列的旗标利用OR(|)运算符组合。

O\_CREAT 若欲打开的文件不存在则自动建立该文件。 O\_EXCL 如果O\_CREAT 也被设置，此指令会去检查文件是否存在。文件若不存在则建立该文件，否则将导致打开文件错误。此外，若O\_CREAT与O\_EXCL同时设置，并且欲打开的文件为符号连接，则会打开文件失败。 O\_NOCTTY 如果欲打开的文件为终端机设备时，则不会将该终端机当成进程控制终端机。 O\_TRUNC 若文件存在并且以可写的方式打开时，此旗标会令文件长度清为0，而原来存于该文件的资料也会消失。 O\_APPEND 当读写文件时会从文件尾开始移动，也就是所写入的数据会以附加的方式加入到文件后面。 O\_NONBLOCK 以不可阻断的方式打开文件，也就是无论有无数据读取或等待，都会立即返回进程之中。 O\_NDELAY 同O\_NONBLOCK。 O\_SYNC 以同步的方式打开文件。 O\_NOFOLLOW 如果参数pathname 所指的文件为一符号连接，则会令打开文件失败。 O\_DIRECTORY 如果参数pathname 所指的文件并非为一目录，则会令打开文件失败。 此为Linux2.2以后特有的旗标，以避免一些系统安全问题。

参数mode 则有下列数种组合，只有在建立新文件时才会生效，此外真正建文件时的权限会受到umask值所影响，因此该文件权限应该为（mode-umaks）。

S\_IRWXU 00700 权限，代表该文件所有者具有可读、可写及可执行的权限。

S\_IRUSR 或S\_IREAD 00400权限，代表该文件所有者具有可读取的权限。

S\_IWUSR 或S\_IWRITE 00200 权限，代表该文件所有者具有可写入的权限。

S\_IXUSR 或S\_IEXEC 00100 权限，代表该文件所有者具有可执行的权限。

S\_IRWXG 00070权限，代表该文件用户组具有可读、可写及可执行的权限。

S\_IRGRP 00040 权限，代表该文件用户组具有可读的权限。

S\_IWGRP 00020权限，代表该文件用户组具有可写入的权限。

S\_IXGRP 00010 权限，代表该文件用户组具有可执行的权限。

S\_IRWXO 00007权限，代表其他用户具有可读、可写及可执行的权限。

S\_IROTH 00004 权限，代表其他用户具有可读的权限

S\_IWOTH 00002权限，代表其他用户具有可写入的权限。

S\_IXOTH 00001 权限，代表其他用户具有可执行的权限。

* 返回值

成功时，open()返回新文件描述符（一个非负整数）。如果出错，则返回 -1，并设置 errno 表示错误。

* 错误代码

EEXIST 参数pathname 所指的文件已存在，却使用了O\_CREAT和O\_EXCL旗标。 EACCESS 参数pathname所指的文件不符合所要求测试的权限。 EROFS 欲测试写入权限的文件存在于只读文件系统内。 EFAULT 参数pathname指针超出可存取内存空间。 EINVAL 参数mode 不正确。 ENAMETOOLONG 参数pathname太长。 ENOTDIR 参数pathname不是目录。 ENOMEM 核心内存不足。 ELOOP 参数pathname有过多符号连接问题。 EIO I/O 存取错误。

* 附加说明

使用access()作用户认证方面的判断要特别小心，例如在access()后再作open()空文件可能会造成系统安全上的问题。

* 范例

#include<unistd.h>#include<sys/types.h>#include<sys/stat.h>#include<fcntl.h>main(){int fd,size;char s [ ]=”Linux Programmer!\n”,buffer[80];fd=open(“/tmp/temp”,O\_WRONLY|O\_CREAT);write(fd,s,sizeof(s));close(fd);fd=open(“/tmp/temp”,O\_RDONLY);size=read(fd,buffer,sizeof(buffer));close(fd);printf(“%s”,buffer);}

* 执行

Linux Programmer!

## 9.11 read

​由已打开的文件读取数据。

* 相关函数

readdir，write，fcntl，close，lseek，readlink，fread

* 表头文件

#include<unistd.h>

* 定义函数

ssize\_t read(int fd, void \*buf, size\_t count);

* 返回值

成功后，将返回读取的字节数（0 表示文件结束），文件位置也将按此数前进。如果这个数字小于请求的

例如，这可能是因为现在实际可用的字节数较少（可能是因为我们已接近文件末尾，或因为我们是从管道或终端读取的

从管道或终端读取），或者 read() 被信号中断。

如果出错，将返回-1，并设置 errno 表示错误。在这种情况下，文件位置（如果有的话）是否发生变化没有明确说明。

* 函数说明

read() 尝试从文件描述符 fd 向缓冲区 buf 开始读取最多count字节。

对于支持寻道的文件，读取操作从文件偏移量开始，文件偏移量按读取的字节数递增。如果文件偏移量位于或超过文件末尾，则不会读取任何字节，read() 返回 0。

如果count为零，read() 可能会检测到下面描述的错误。如果没有任何错误，或者 read() 没有检测错误，那么计数为 0 的 read() 将返回 0，并且没有其他影响。

根据 POSIX.1，如果 count 大于 SSIZE\_MAX，结果将由实现定义；Linux 上的上限参见 NOTES。

* 附加说明

如果顺利read()会返回实际读到的字节数，最好能将返回值与参数count 作比较，若返回的字节数比要求读取的字节数少，则有可能读到了文件尾、从管道(pipe)或终端机读取，或者是read()被信号中断了读取动作。当有错误发生时则返回-1，错误代码存入errno中，而文件读写位置则无法预期。

* 错误代码

EINTR 此调用被信号所中断。 EAGAIN 当使用不可阻断I/O 时（O\_NONBLOCK），若无数据可读取则返回此值。 EBADF 参数fd 非有效的文件描述词，或该文件已关闭。

* 范例

参考open()

## 9.12 sync

​将缓冲区数据写回磁盘。

* 相关函数

Fsync

* 表头文件

#include<unistd.h>

* 定义函数

void sync(void);

int syncfs(int fd);

* 函数说明

sync() 会将对文件系统元数据和缓存文件数据的所有待处理修改写入底层文件系统。

syncfs() 与 sync() 类似，但只同步打开的文件描述符 fd 所引用的包含文件的文件系统。

* 返回值

syncfs() 成功时返回 0；出错时返回 -1 并设置 errno 以指示错误。

## 9.13 write

​将数据写入已打开的文件内。

* 相关函数

open，read，fcntl，close，lseek，sync，fsync，fwrite

* 表头文件

#include<unistd.h>

* 定义函数

ssize\_t write(int fd, const void \*buf, size\_t count);

* 函数说明

write() 从 buf 开始的缓冲区向文件描述符 fd 指向的文件写入最多 count 字节。

例如，如果底层物理介质空间不足，或遇到 RLIMIT\_FSIZE 资源限制（参见 setrlimit(2)），或在写入少于 count 字节后调用被信号处理程序中断，写入的字节数可能少于 count。(另请参见 pipe(7)）。

对于可寻址文件（即可以使用 lseek(2) 的文件，例如普通文件），写入是在文件偏移量处进行的，文件偏移量按实际写入的字节数递增。如果文件是用 O\_APPEND 打开(2)的，则在写入之前首先将文件偏移量设置为文件末尾。文件偏移量的调整和写入操作是作为一个原子步骤执行的。POSIX 要求，在写入()返回后进行的读取(2)操作将返回新数据。请注意，并非所有文件系统都符合 POSIX 标准。

根据 POSIX.1，如果 count 大于 SSIZE\_MAX，结果将由实现定义；Linux 上的上限参见 NOTES。

* 返回值

成功时，返回写入的字节数。如果出错，则返回-1，并设置 errno 表示出错。

请注意，成功写入的字节数可能少于计数字节。发生这种部分写入的原因有很多，例如，磁盘设备上没有足够的空间来写入所有请求的字节，或者向套接字、管道或类似设备阻塞的写入()在传输了部分字节后，但在传输所有请求的字节前被信号处理器中断。在部分写入的情况下，调用者可以再调用一次 write() 来传输剩余的字节。后续调用将传输更多字节，或者导致错误（例如，磁盘已满）。

如果 count 为零，且 fd 指向普通文件，那么如果检测到以下错误之一，write() 可能会返回失败状态。如果没有检测到错误，或者没有进行错误检测，则返回 0，不会产生任何其他影响。如果 count 为零，且 fd 指向普通文件以外的文件，则不会指定结果。

* 错误代码

EINTR 此调用被信号所中断。 EAGAIN 当使用不可阻断I/O 时（O\_NONBLOCK），若无数据可读取则返回此值。 EADF 参数fd非有效的文件描述词，或该文件已关闭。

* 范例

参考open()

# 第十章 文件内容操作篇

## 10.1 clearerr

​重置给定文件流的错误标志和 EOF 指示器。

* 相关函数

Feof

* 表头文件

#include<stdio.h>

* 定义函数

void clearerr(FILE \*stream);

* 函数说明

函数 clearerr() 清除数据流指向的文件结束和错误指示符。

函数 feof() 测试 stream 所指向数据流的文件结束指示符，如果设置为非零，则返回非零值。文件结束指示符只能通过函数clearerr()清除文件结束指示符。

函数 ferror() 测试 stream 所指向数据流的错误指示符，如果设置了错误指示符，则返回非零值。错误指示器只能通过函数 clearerr() 重置。

关于非锁定对应函数，请参阅 unlocked\_stdio(3)。

* 范例

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

int main(void)

{

FILE\* tmpf = tmpfile();

fputs("abcde\n", tmpf);

rewind(tmpf);

int ch;

while ((ch=fgetc(tmpf)) != EOF)

printf("%c", ch);

assert(feof(tmpf)); // 此循环期待以 eof 终止

puts("End of file reached");

clearerr(tmpf); // 清除 eof

if (feof(tmpf))

puts("EOF indicator set");

else

puts("EOF indicator cleared\n");

}

* 输出

abcde

End of file reached

EOF indicator cleared

## 10.2 fclose

​关闭文件。

* 相关函数

close，fflush，fopen，setbuf

* 表头文件

#include<stdio.h>

* 定义函数

int fclose(FILE \*stream);

* 函数说明

fclose()用来关闭先前fopen()打开的文件。此动作会让缓冲区内的数据写入文件中，并释放系统所提供的文件资源。

函数 fclose() 会刷新 stream 指向的数据流（使用 fflush(3) 写入任何缓冲输出数据），并关闭底层文件描述符。

关闭给定的文件流。冲入任何未写入的缓冲数据到 OS 。舍弃任何未读取的缓冲数据。

无论操作是否成功，流都不再关联到文件，且由 setbuf 或 setvbuf 分配的缓冲区若存在，则亦被解除关联，并且若使用自动分配则被解分配。

若在 fclose 返回后使用指针 stream 的值则行为未定义。

* 返回值

成功完成后，将返回 0。否则，将返回 EOF，并设置 errno 表示错误。在这两种情况下，对流的任何进一步访问（包括再次调用 fclose()）都会导致未定义的行为。

* 错误代码

EBADF表示参数stream非已打开的文件。

* 范例

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

FILE\* fp = fopen("test.txt", "r");

if(!fp) {

perror("File opening failed");

return EXIT\_FAILURE;

}

int c; // 注意：int，非char，要求处理EOF

while ((c = fgetc(fp)) != EOF) { // 标准C I/O读取文件循环

putchar(c);

}

if (ferror(fp))

puts("I/O error when reading");

else if (feof(fp))

puts("End of file reached successfully");

fclose(fp);

}

## 10.3 fdopen

​将文件描述词转为文件指针。

* 相关函数

fopen，open，fclose

* 表头文件

#include<stdio.h>

* 定义函数

FILE \*fdopen(int fildes, const char \*mode);

* 函数说明

fdopen()会将参数fildes 的文件描述词，转换为对应的文件指针后返回。参数mode 字符串则代表着文件指针的流形态，此形态必须和原先文件描述词读写模式相同。关于mode 字符串格式请参考fopen()。

fdopen() 函数将流与现有的文件描述符 fd 关联。流的模式（“r”、“r+”、“w”、“w+”、“a”、“a+”中的一种）必须与文件描述符的模式一致。新数据流的文件位置指示符被设置为属于 fd 的指示符，错误和文件结束指示符被清除。模式 “w ”或 “w+”不会导致文件截断。文件描述符不会被复制，并将在关闭 fdopen() 创建的流时关闭。对共享内存对象应用 fdopen() 的结果未定义。

* 返回值

转换成功时返回指向该流的文件指针。失败则返回NULL，并把错误代码存在errno中。

* 范例

#include<stdio.h>main(){FILE \* fp =fdopen(0,”w+”);fprintf(fp,”%s\n”,”hello!”);fclose(fp);}

* 执行

hello!

## 10.4 feof

​检查文件流是否读到了文件尾。

* 相关函数

fopen，fgetc，fgets，fread

* 表头文件

#include<stdio.h>

* 定义函数

int feof(FILE \*stream);

* 函数说明

feof()用来侦测是否读取到了文件尾，尾数stream为fopen（）所返回之文件指针。如果已到文件尾则返回非零值，其他情况返回0。

此函数仅报告最近一次 I/O 操作所报告的流状态，而不检验关联的数据源。例如，若最近一次 I/O 是抵达文件最后字节的 fgetc ，则 feof 返回零。下个 fgetc 失败并更改流状态为文件尾。然后 feof 才返回非零。

典型用法中，输入流处理在任何错误时停止；而 feof 和 ferror 用于区别不同错误条件。

函数 feof() 测试 stream 所指向数据流的文件结束指示符，如果设置为非零，则返回非零值。文件结束指示符只能通过函数clearerr()函数清除文件结束指示符。

* 返回值

如果为数据流设置了文件结束指示符，则 feof() 函数返回非零值；否则返回零值。

* 范例

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

FILE\* fp = fopen("test.txt", "r");

if(!fp) {

perror("File opening failed");

return EXIT\_FAILURE;

}

int c; // 注意：int，非char，要求处理EOF

while ((c = fgetc(fp)) != EOF) { // 标准C I/O读取文件循环

putchar(c);

}

if (ferror(fp))

puts("I/O error when reading");

else if (feof(fp))

puts("End of file reached successfully");

fclose(fp);

}

## 10.5 fflush

​更新缓冲区。

* 相关函数

write，fopen，fclose，setbuf

* 表头文件

#include<stdio.h>

* 定义函数

int fflush(FILE \*stream);

* 函数说明

fflush()会强迫将缓冲区内的数据写回参数stream指定的文件中。如果参数stream为NULL，fflush()会将所有打开的文件数据更新。

对于输出流（及最后操作为输出的更新流），从 stream 的缓冲区写入未写的数据到关联的输出设备。

对于输入流（及最后操作为输入的更新流），行为未定义。

对于输出流，fflush() 会通过流的底层写函数强制写入给定输出或更新流的所有用户空间缓冲数据。

对于与可寻文件（如磁盘文件，但不包括管道或终端）相关的输入流，fflush() 会丢弃任何已从底层文件获取但尚未被应用程序使用的缓冲数据。

如果流参数为 NULL，fflush() 会刷新所有打开的输出流。

有关非锁定对应程序，请参阅 unlocked\_stdio(3)。

* 返回值

成功返回0，失败返回EOF，错误代码存于errno中。

* 错误代码

EBADF 参数stream 指定的文件未被打开，或打开状态为只读。其它错误代码参考write（）。

## 10.6 fgetc

​由文件中读取一个字符。

* 相关函数

open，fread，fscanf，getc

* 表头文件

#include<stdio.h>

* 定义函数

int fgetc(FILE \* stream);

* 函数说明

fgetc()从参数stream所指的文件中读取一个字符。若读到文件尾而无数据时便返回EOF。

* 返回值

成功时为作为 unsigned char 获得并转换为 int 的字符，失败时为 EOF 。

若文件尾条件导致失败，则另外设置 stream 上的文件尾指示器（见 feof() ）。若某些其他错误导致失败，则设置 stream 上的错误指示器（见 ferror() ）。

* 范例

#include<stdio.h>main(){FILE \*fp;int c;fp=fopen(“exist”,”r”);while((c=fgetc(fp))!=EOF)printf(“%c”,c);fclose(fp);}

## 10.7 fgets

​由文件中读取一字符串。

* 相关函数

open，fread，fscanf，getc

* 表头文件

#include<stdio.h>

* 定义函数

char \*fgets(char \*s, int size, FILE \*stream);

* 函数说明

fgets()用来从参数stream所指的文件内读入字符并存到参数s所指的内存空间，直到出现换行字符、读到文件尾或是已读了size-1个字符为止，最后会加上NULL作为字符串结束。

* 返回值

成功时为 s，失败时为空指针。

若遇到文件尾条件导致了失败，则设置 stream 上的文件尾指示器（见 feof() ）。这仅若它导致未读取字符才是失败，该情况下返回空指针且不改变 s 所指向数组的内容（即不以空字符覆写首字节）。

若某些其他错误导致了失败，则设置 stream 上的错误指示器（见 ferror() ）。 s 所指向的数组内容是不确定的（甚至可以不是空终止）。

POSIX 额外要求若 fgets 遇到异于文件尾条件的失败则设置 errno 。

尽管标准规范在 count<=1 的情况下不明，常见的实现

若 count < 1 则不做任何事并报告错误

若 count == 1 ，则某些实现不做任何事并报告错误，

其他实现不读内容，存储零于 str[0] 并报告成功

* 范例

#include<stdio.h>main(){char s[80];fputs(fgets(s,80,stdin),stdout);}

* 执行

this is a test /\*输入\*/this is a test /\*输出\*/

## 10.8 fileno

​返回文件流所使用的文件描述词。

* 相关函数

open，fopen

* 表头文件

#include<stdio.h>

* 定义函数

int fileno(FILE \*stream);

* 函数说明

函数 fileno() 检查参数流，并返回用于实现该流的整数文件描述符。该文件描述符仍为 stream 所有，并将在调用 fclose(3) 时关闭。

在将文件描述符传递给可能关闭它的代码之前，请使用 dup(2) 复制该文件描述符。

* 返回值

成功时，fileno() 返回与数据流相关的文件描述符。如果失败，则返回 -1，并设置 errno 表示错误。

* 范例

#include<stdio.h>main(){FILE \* fp;int fd;fp=fopen(“/etc/passwd”,”r”);fd=fileno(fp);printf(“fd=%d\n”,fd);fclose(fp);}

* 执行

fd=3

## 10.9 fopen

​打开文件。

* 相关函数

close，fflush，fopen，setbuf

* 表头文件

#include<stdio.h>

* 定义函数

FILE \*fopen(const char \*path, const char \*mode);

* 函数说明

参数path字符串包含欲打开的文件路径及文件名，参数mode字符串则代表着流形态。

mode有下列几种形态字符串: r 打开只读文件，该文件必须存在。 r+ 打开可读写的文件，该文件必须存在。 w 打开只写文件，若文件存在则文件长度清为0，即该文件内容会消失。若文件不存在则建立该文件。 w+ 打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。 a 以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。 a+ 以附加方式打开可读写的文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留。

上述的形态字符串都可以再加一个b字符，如rb、w+b或ab＋等组合，加入b 字符用来告诉函数库打开的文件为二进制文件，而非纯文字文件。不过在POSIX系统，包含Linux都会忽略该字符。由fopen()所建立的新文件会具有S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IWGRP|S\_IROTH|S\_IWOTH(0666)权限，此文件权限也会参考umask值。

* 返回值

文件顺利打开后，指向该流的文件指针就会被返回。若果文件打开失败则返回NULL，并把错误代码存在errno 中。

* 附加说明

一般而言，开文件后会作一些文件读取或写入的动作，若开文件失败，接下来的读写动作也无法顺利进行，所以在fopen()后请作错误判断及处理。

path的格式是实现定义的，而且不需要表示一个文件（譬如可以是控制台或另一能通过文件系统 API 访问的设备）。在支持的平台上， path可以包含绝对或相对路径。

* 错误代码

EBADF表示参数stream非已打开的文件。

* 范例

#include<stdio.h>main(){FILE \* fp;fp=fopen(“noexist”,”a+”);if(fp= =NULL) return;fclose(fp);}

## 10.10 fputc

​将一指定字符写入文件流中。

* 相关函数

fopen，fwrite，fscanf，putc

* 表头文件

#include<stdio.h>

* 定义函数

int fputc(int c, FILE \*stream);

* 函数说明

写入字符 c 到给定输出流 stream 。 putc() 可以实现为宏并对 stream 求值超过一次，故对应的参数决不应是有副效应的表达式。

在内部，在写入前将字符转换为 unsigned char 。

* 返回值

成功时，返回被写入字符。

失败时，返回 EOF 并设置 stream 上的错误指示器（见 ferror() ）。

* 范例

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

int ret\_code = 0;

for (char c = 'a'; (ret\_code != EOF) && (c != 'z'); c++)

ret\_code = putc(c, stdout);

/\* 测试是否抵达 EOF 。 \*/

if (ret\_code == EOF)

if (ferror(stdout))

{

perror("putc()");

fprintf(stderr,"putc() failed in file %s at line # %d\n", \_\_FILE\_\_,\_\_LINE\_\_-7);

exit(EXIT\_FAILURE);

}

putc('\n', stdout);

return EXIT\_SUCCESS;

}

* 输出

abcdefghijklmnopqrstuvwxy

## 10.11 fputs

​将一指定的字符串写入文件内。

* 相关函数

fopen，fwrite，fscanf，fputc，putc

* 表头文件

#include<stdio.h>

* 定义函数

int fputs(const char \*s, FILE \*stream);

* 函数说明

将以NULL结尾的字符串 str 的每个字符写入到输出流 stream ，如同通过重复执行 fputc 。

不将 str 的空字符写入。

相关函数 puts 后附新换行符到输出，而 fputs 写入不修改的字符串。

不同的实现返回不同的非负数：一些返回最后写入的字符，一些返回写入的字符数（或若字符串长于 INT\_MAX 则为该值），一些简单地非负常量，例如零。

* 返回值

成功时，返回非负值。

失败时，返回 EOF 并设置 stream 上的错误指示器（见 ferror ）。

* 错误代码

EBADF表示参数stream非已打开的文件。

* 范例

#include <stdio.h>

int main(void)

{

int rc = fputs("Hello World", stdout);

if (rc == EOF)

perror("fputs()"); // POSIX 要求设置 errno

}

* 输出

Hello World

## 10.12 fread

​从文件流读取数据。

* 相关函数

fopen，fwrite，fseek，fscanf

* 表头文件

#include<stdio.h>

* 定义函数

size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

* 函数说明

fread()用来从文件流中读取数据。参数stream为已打开的文件指针，参数ptr 指向欲存放读取进来的数据空间，读取的字符数以参数size\*nmemb来决定。Fread()会返回实际读取到的nmemb数目，如果此值比参数nmemb 来得小，则代表可能读到了文件尾或有错误发生，这时必须用feof()或ferror()来决定发生什么情况。

* 返回值

成功时，fread() 和 fwrite() 会返回读取或写入的条目数。只有当 size 为 1 时，这个数字才等于传输的字节数。

如果出现错误或到达文件末尾，返回值将是短条目数（或零）。

数据流的文件位置指示器将根据成功读取或写入的字节数前进。

fread() 不区分文件结束和错误，调用者必须使用 feof(3)和 ferror(3) 来确定发生了哪种情况。

* 范例

#include<stdio.h>#define nmemb 3struct test{char name[20];int size;}s[nmemb];main(){FILE \* stream;int i;stream = fopen(“/tmp/fwrite”,”r”);fread(s,sizeof(struct test),nmemb,stream);fclose(stream);for(i=0;i<nmemb;i++)printf(“name[%d]=%-20s:size[%d]=%d\n”,i,s[i].name,i,s[i].size);}

* 执行

name[0]=Linux! size[0]=6name[1]=FreeBSD! size[1]=8name[2]=Windows2000 size[2]=11

## 10.13 freopen

​打开文件。

* 相关函数

fopen，fclose

* 表头文件

#include<stdio.h>

* 定义函数

FILE \*freopen(const char \*path, const char \*mode, FILE \*stream);

* 函数说明

freopen() 函数打开文件名为 path指向的字符串的文件，并将 stream 指向的流与之关联。原始流（如果存在）将被关闭。模式参数的使用与 fopen() 函数相同。

如果 path参数是空指针，freopen() 会将流的模式更改为 mode 中指定的模式；也就是说，freopen() 会重新打开与流关联的路径名。C99 标准中增加了对这一行为的规范，其中规定：在这种情况下，如果调用 freopen() 成功，则不必关闭与流相关联的文件描述符。至于在什么情况下允许改变模式（如果有的话），则由实现定义。

freopen() 函数的主要用途是更改与标准文本流（stderr、stdin 或 stdout）相关的文件。

* 返回值

文件顺利打开后，指向该流的文件指针就会被返回。如果文件打开失败则返回NULL，并把错误代码存在errno 中。

* 错误代码

EBADF表示参数stream非已打开的文件。

* 范例

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

puts("stdout is printed to console");

if (freopen("redir.txt", "w", stdout) == NULL)

{

perror("freopen() failed");

return EXIT\_FAILURE;

}

puts("stdout is redirected to a file"); // 写入 redir.txt

fclose(stdout);

}

* 输出

stdout is printed to console

## 10.14 fseek

​移动文件流的读写位置。

* 相关函数

rewind，ftell，fgetpos，fsetpos，lseek

* 表头文件

#include<stdio.h>

* 定义函数

int fseek(FILE \*stream, long offset, int whence);

* 函数说明

函数 fseek() 为 stream 所指向的数据流设置文件位置指示器。

若 stream 以二进制模式打开，则新位置准确地是文件起始后（若 origin 为 SEEK\_SET ）或当前文件位置后（若 origin 为 SEEK\_CUR ），或文件结尾后（若 origin 为 SEEK\_END ）的 offset 字节。不要求二进制流支持 SEEK\_END ，尤其是是否输出附加的空字节。

若 stream 以文本模式打开，则仅有的受支持 offset 值为零（可用于任何 origin ）和先前在关联到同一个文件的流上对 ftell 的调用的返回值（仅可用于 SEEK\_SET 的 origin ）。

若 stream 为宽面向，则一同应用对文本和二进制流的限制（允许 ftell 的结果与 SEEK\_SET 一同使用，并允许零 offset 以 SEEK\_SET 和 SEEK\_CUR 但非 SEEK\_END 为基准）。

除了更改文件位置指示器， fseek 还撤销 ungetc 的效果并清除文件尾状态，若可应用。

若发生读或写错误，则设置流的错误指示器（ ferror ）而不影响文件位置。

* 返回值

当调用成功时则返回0，若有错误则返回-1，errno会存放错误代码。

* 参数

whence为下列其中一种: SEEK\_SET从距文件开头offset位移量为新的读写位置。SEEK\_CUR 以目前的读写位置往后增加offset个位移量。 SEEK\_END将读写位置指向文件尾后再增加offset个位移量。 当whence值为SEEK\_CUR 或SEEK\_END时，参数offset允许负值的出现。 下列是较特别的使用方式: 1) 欲将读写位置移动到文件开头时:fseek(FILE stream,0,SEEK\_SET); 2) 欲将读写位置移动到文件尾时:fseek(FILE stream,0,0SEEK\_END);

* 附加说明

fseek()不像lseek()会返回读写位置，因此必须使用ftell()来取得目前读写的位置。

* 范例

#include<stdio.h>main(){FILE \* stream;long offset;fpos\_t pos;stream=fopen(“/etc/passwd”,”r”);fseek(stream,5,SEEK\_SET);printf(“offset=%d\n”,ftell(stream));rewind(stream);fgetpos(stream,&pos);printf(“offset=%d\n”,pos);pos=10;fsetpos(stream,&pos);printf(“offset = %d\n”,ftell(stream));fclose(stream);}

* 执行

offset = 5offset =0offset=10

## 10.15 ftell

​取得文件流的读取位置。

* 相关函数

fseek，rewind，fgetpos，fsetpos

* 表头文件

#include<stdio.h>

* 定义函数

long ftell(FILE \*stream);

* 函数说明

ftell()用来取得文件流目前的读写位置。参数stream为已打开的文件指针。

若流以二进制模式打开，则由此函数获得的值是从文件开始的字节数。

若流以文本模式打开，则由此函数返回的值未指定，且仅若作为 fseek() 的输入才有意义。

* 返回值

当调用成功时则返回目前的读写位置，若有错误则返回-1，errno会存放错误代码。

* 错误代码

EBADF 参数stream无效或可移动读写位置的文件流。

* 范例

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

/\* 准备浮点值的数组。 \*/

#define SIZE 5

double A[SIZE] = {1.,2.,3.,4.,5.};

/\* 写数组到文件。 \*/

FILE \* fp = fopen("test.bin", "wb");

fwrite(A,sizeof(double),SIZE,fp);

fclose (fp);

/\* 读取浮点值到数组 B。 \*/

double B[SIZE];

fp = fopen("test.bin","rb");

long int pos = ftell(fp); /\* 位置指示器在文件起始 \*/

if (pos == -1L)

{

perror("ftell()");

fprintf(stderr,"ftell() failed in file %s at line # %d\n", \_\_FILE\_\_,\_\_LINE\_\_-4);

exit(EXIT\_FAILURE);

}

printf("%ld\n", pos);

int ret\_code = fread(B,sizeof(double),1,fp); /\* 读取一个浮点值 \*/

pos = ftell(fp); /\* 读取一个浮点值后的位置指示器 \*/

if (pos == -1L)

{

perror("ftell()");

fprintf(stderr,"ftell() failed in file %s at line # %d\n", \_\_FILE\_\_,\_\_LINE\_\_-4);

exit(EXIT\_FAILURE);

}

printf("%ld\n", pos);

printf("%.1f\n", B[0]); /\* 打印浮点值 \*/

return EXIT\_SUCCESS;

}

* 输出

0

8

1.0

## 10.16 fwrite

​将数据写至文件流。

* 相关函数

fopen，fread，fseek，fscanf

* 表头文件

#include<stdio.h>

* 定义函数

size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

* 函数说明

fwrite()用来将数据写入文件流中。参数stream为已打开的文件指针，参数ptr 指向欲写入的数据地址，总共写入的字符数以参数size\*nmemb来决定。Fwrite()会返回实际写入的nmemb数目。

* 返回值

成功时， fwrite() 会返回读取或写入的条目数。只有当 size 为 1 时，这个数字才等于传输的字节数。

如果出现错误或到达文件末尾，返回值将是短条目数（或零）。

数据流的文件位置指示器将根据成功读取或写入的字节数前进。

* 范例

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

enum { SIZE = 5 };

int main(void)

{

double a[SIZE] = {1, 2, 3, 4, 5};

FILE \*f1 = fopen("file.bin", "wb");

assert(f1);

size\_t r1 = fwrite(a, sizeof a[0], SIZE, f1);

printf("wrote %zu elements out of %d requested\n", r1, SIZE);

fclose(f1);

double b[SIZE];

FILE \*f2 = fopen("file.bin", "rb");

size\_t r2 = fread(b, sizeof b[0], SIZE, f2);

fclose(f2);

printf("read back: ");

for(size\_t i = 0; i < r2; i++)

printf("%0.2f ", b[i]);

}

* 执行

wrote 5 elements out of 5 requested

read back: 1.00 2.00 3.00 4.00 5.00

## 10.17 fclose

​由文件中读取一个字符。

* 相关函数

read，fopen，fread，fgetc

* 表头文件

#include<stdio.h>

* 定义函数

int fclose(FILE \*stream);

* 函数说明

关闭给定的文件流。冲入任何未写入的缓冲数据到 OS 。舍弃任何未读取的缓冲数据。

无论操作是否成功，流都不再关联到文件，且由 setbuf 或 setvbuf 分配的缓冲区若存在，则亦被解除关联，并且若使用自动分配则被解分配。

若在 fclose 返回后使用指针 stream 的值则行为未定义。

* 返回值

getc()会返回读取到的字符，若返回EOF则表示到了文件尾。

* 范例

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

FILE\* fp = fopen("test.txt", "r");

if(!fp) {

perror("File opening failed");

return EXIT\_FAILURE;

}

int c; // 注意：int，非char，要求处理EOF

while ((c = fgetc(fp)) != EOF) { // 标准C I/O读取文件循环

putchar(c);

}

if (ferror(fp))

puts("I/O error when reading");

else if (feof(fp))

puts("End of file reached successfully");

fclose(fp);

}

## 10.18 getchar

​由标准输入设备内读进一字符。

* 相关函数

fopen，fread，fscanf，getc

* 表头文件

#include<stdio.h>

* 定义函数

int getchar(void);

* 函数说明

getchar()用来从标准输入设备中读取一个字符。然后将该字符从unsigned char转换成int后返回。等价于 getc(stdin) 。

* 返回值

成功时为获得的字符，失败时为 EOF 。

若失败由文件尾条件产生，则另外设置 stdin 上的文件尾指示器（见 feof() ）。若失败由某些其他错误产生，则设置 stdin 上的错误指示器（见 ferror() ）。

* 附加说明

getchar()非真正函数，而是getc(stdin)宏定义。

* 范例

#include<stdio.h>main(){FILE \* fp;int c,i;for(i=0li<5;i++){c=getchar();putchar(c);}}

* 执行

1234 /\*输入\*/1234 /\*输出\*/

## 10.19 gets

​关闭文件。

* 相关函数

fopen，fread，fscanf，fgets

* 表头文件

#include<stdio.h>

* 定义函数

char \*gets(char \*s);

* 函数说明

gets()用来从标准设备读入字符并存到参数s所指的内存空间，直到出现换行字符或读到文件尾为止，最后加上NULL作为字符串结束。

* 返回值

成功时为 str ，失败时为空指针。

若文件尾条件导致了失败，则附加设置 stdin 的文件尾指示器（见 feof() ）。若其他某些原因导致了失败，则设置 stdin 的错误指示器（见 ferror() ）。

* 附加说明

gets() 函数不进行边界检查，从而此函数对缓冲区溢出攻击极度脆弱。无法安全使用它（除非程序运行的环境限定能出现在 stdin 上的内容）。因此，此函数在 C99 的第三次勘误中被弃用，而在 C11 标准发布时被移除。推荐的替代品是 fgets() 和 gets\_s() 。

* 范例

参考fgets()

## 10.20 mktemp

​产生唯一的临时文件名。

* 相关函数

Tmpfile

* 表头文件

#include<stdlib.h>

* 定义函数

char \*mktemp(char \*template);

* 函数说明

mktemp() 函数从模板中生成一个唯一的临时文件名。模板的最后六个字符必须是 XXXXXX，这些字符将被替换为一个字符串，使文件名具有唯一性。由于 template 会被修改，因此它不能是字符串常量，而应声明为字符数组。产生后的文件名会借字符串指针返回。

* 返回值

文件顺利打开后，指向该流的文件指针就会被返回。如果文件打开失败则返回NULL，并把错误代码存在errno中。

mktemp() 函数总是返回模板。如果创建了独一无二的名称，则模板的最后六个字节将被修改，从而使生成的名称独一无二（即不存在）。 如果无法创建独一无二的名称，则模板将变为空字符串，并设置 errno 表示错误。

* 附加说明

参数template所指的文件名称字符串必须声明为数组，如:char template[ ]=”template-XXXXXX”；不可用char \* template=”template-XXXXXX”；

* 范例

#include<stdlib.h>main(){char template[ ]=”template-XXXXXX”;mktemp(template);printf(“template=%s\n”,template);}

## 10.21 putc

​将一指定字符写入文件中。

* 相关函数

fopen，fwrite，fscanf，fputc

* 表头文件

#include<stdio.h>

* 定义函数

int putc(int c, FILE \*stream);

* 函数说明

putc()会将参数c转为unsigned char后写入参数stream指定的文件中。虽然putc()与fputc()作用相同，但putc()为宏定义，非真正的函数调用。

putc() 可以实现为宏并对 stream 求值超过一次，故对应的参数决不应是有副效应的表达式。

在内部，在写入前将字符转换为 unsigned char 。

* 返回值

成功时，返回被写入字符。

失败时，返回 EOF 并设置 stream 上的错误指示器（见 ferror() ）。

* 范例

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

int ret\_code = 0;

for (char c = 'a'; (ret\_code != EOF) && (c != 'z'); c++)

ret\_code = putc(c, stdout);

/\* 测试是否抵达 EOF 。 \*/

if (ret\_code == EOF)

if (ferror(stdout))

{

perror("putc()");

fprintf(stderr,"putc() failed in file %s at line # %d\n", \_\_FILE\_\_,\_\_LINE\_\_-7);

exit(EXIT\_FAILURE);

}

putc('\n', stdout);

return EXIT\_SUCCESS;

}

* 输出

abcdefghijklmnopqrstuvwxy

## 10.22 putchar

​将指定的字符写到标准输出设备。

* 相关函数

fopen，fwrite，fscanf，fputc

* 表头文件

#include<stdio.h>

* 定义函数

int putchar(int c);

* 函数说明

putchar()用来将参数c字符写到标准输出设备。

写字符 ch 到 stdout 。在内部，字符于写入前被转换到 unsigned char 。等价于 putc(ch, stdout) 。

* 返回值

成功时返回写入的字符。

失败时返回 EOF 并设置 stdout 上的错误指示器（见 ferror() ）。

* 附加说明

putchar()非真正函数，而是putc(c，stdout)宏定义。

* 范例

参考getchar()

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

int ret\_code = 0;

for (char c = 'a'; (ret\_code != EOF) && (c != 'z'); c++)

ret\_code = putchar(c);

/\* 测试是否抵达 EOF 。 \*/

if (ret\_code == EOF)

if (ferror(stdout))

{

fprintf(stderr,"putchar() failed in file %s at line # %d\n", \_\_FILE\_\_,\_\_LINE\_\_-6);

perror("putchar()");

exit(EXIT\_FAILURE);

}

putchar('\n');

// putchar 返回值不等于参数

int r = 0x1070;

printf("\n0x%x\n", r);

r = putchar(r);

printf("\n0x%x\n", r);

}

* 输出

abcdefghijklmnopqrstuvwxy

0x1070

p

0x70

## 10.23 rewind

​重设文件流的读写位置为文件开头。

* 相关函数

fseek，ftell，fgetpos，fsetpos

* 表头文件

#include<stdio.h>

* 定义函数

void rewind(FILE \*stream);

* 函数说明

rewind()用来把文件流的读写位置移至文件开头。参数stream为已打开的文件指针。此函数相当于调用fseek(stream,0,SEEK\_SET)，除了它清除文件尾和错误指示器。

此函数丢弃任何来自先前对 ungetc 调用的效果。

* 范例

此例演示如何读文件二次

#include <stdio.h>

char str[20];

int main(void)

{

FILE \*f;

char ch;

f = fopen("file.txt", "w");

for (ch = '0'; ch <= '9'; ch++) {

fputc(ch, f);

}

fclose(f);

f = fopen("file.txt", "r");

fread(str, 1, 10, f);

puts(str);

rewind(f);

fread(str, 1, 10, f);

puts(str);

fclose(f);

return 0;

}

* 输出

0123456789

0123456789

## 10.24 setbuf

​设置文件流的缓冲区。

* 相关函数

setbuffer，setlinebuf，setvbuf

* 表头文件

#include<stdio.h>

* 定义函数

void setbuf(FILE \*stream, char \*buf);

* 函数说明

在打开文件流后，读取内容之前，调用setbuf()可以用来设置文件流的缓冲区。

参数stream为指定的文件流，参数buf指向自定的缓冲区起始地址。如果参数buf为NULL指针，则为无缓冲IO。

Setbuf()相当于调用:setvbuf(stream,buf,buf?\_IOFBF:\_IONBF,BUFSIZ)

设置用于流操作的内部缓冲区。其长度至少应该为 BUFSIZ 个字符。

若 buffer 非空，则等价于 setvbuf(stream, buffer, \_IOFBF, BUFSIZ) 。

若 buffer 为空，则等价于 setvbuf(stream, NULL, \_IONBF, 0) ，这会关闭缓冲。

若 BUFSIZ 不是适合的缓冲区大小，则能用 setvbuf 更改它。

setvbuf 亦应当用于检测错误，因为 setbuf 不指示成功或失败。

此函数仅可在已将 stream 关联到打开的文件后，但要在任何其他操作（除了对 setbuf/setvbuf 的失败调用）前使用。

一个常见错误是设置 stdin 或 stdout 的缓冲区为生存期在程序终止前结束的数组。

* 范例

/\*setbuf 可用于禁用要求立即输出的流的缓冲。\*/

#include <stdio.h>

#include <threads.h>

int main(void)

{

setbuf(stdout, NULL); // 无缓冲的 stdout

putchar('a'); // 若 stdout 无缓冲，则 'a' 立即出现

thrd\_sleep(&(struct timespec){.tv\_sec=1}, NULL); // 休眠 1 秒

putchar('b');

}

* 输出

ab

## 10.25 setbuffer

​设置文件流的缓冲区。

* 相关函数

setlinebuf，setbuf，setvbuf

* 表头文件

#include<stdio.h>

* 定义函数

void setbuffer(FILE \*stream, char \*buf, size\_t size);

* 函数说明

在打开文件流后，读取内容之前，调用setbuffer()可用来设置文件流的缓冲区。

参数stream为指定的文件流，参数buf指向自定的缓冲区起始地址，参数size为缓冲区大小。

缓冲有三种类型：非缓冲、块缓冲和行缓冲。

当输出流为非缓冲流时，信息一经写入就会出现在目标文件或终端上；当输出流为块缓冲流时，许多字符会被保存起来并作为一个块写入；当输出流为行缓冲流时，字符会被保存起来，直到输出换行符或从连接到终端设备的任何流（通常是 stdin）读取输入为止。

可以使用函数 fflush(3) 强制提前删除块（参见 fclose(3)）。(参见 fclose(3)）。

通常情况下，所有文件都是块缓冲的。如果数据流指向终端（通常是 stdout），则是行缓冲。默认情况下，标准错误流 stderr 始终是非缓冲的。

可以在任何打开的流上使用 setvbuf() 函数来更改其缓冲区。模式参数必须是以下三个宏之一：

## 10.26 setlinebuf

​设置文件流为线性缓冲区。

* 相关函数

setbuffer，setbuf，setvbuf

* 表头文件

#include<stdio.h>

* 定义函数

void setlinebuf(FILE \*stream);

* 函数说明

setlinebuf()用来设置文件流以换行为依据的无缓冲IO。相当于调用:setvbuf(stream,(char \* )NULL,\_IOLBF,0);请参考setvbuf()。

## 10.27 setvbuf

​设置文件流的缓冲区。

* 相关函数

setbuffer，setlinebuf，setbuf

* 表头文件

#include<stdio.h>

* 定义函数

int setvbuf(FILE \*stream, char \*buf, int mode, size\_t size);

* 函数说明

在打开文件流后，读取内容之前，调用setvbuf()可以用来设置文件流的缓冲区。参数stream为指定的文件流，参数buf指向自定的缓冲区起始地址，参数size为缓冲区大小，参数mode有下列几种 \_IONBF 无缓冲IO \_IOLBF 以换行为依据的无缓冲IO \_IOFBF 完全无缓冲IO。如果参数buf为NULL指针，则为无缓冲IO。

setlinebuf() 函数与以下调用完全等价：setvbuf(stream, NULL, \_IOLBF, 0)；

## 10.29 ungetc

​将指定字符写回文件流中。

* 相关函数

fputc，getchar，getc

* 表头文件

#include<stdio.h>

* 定义函数

int ungetc(int c, FILE \*stream);

* 函数说明

ungetc()将参数c字符写回参数stream所指定的文件流。这个写回的字符会由下一个读取文件流的函数取得。

若 c不等于 EOF ，则推入字符 c（转译为 unsigned char ）到与流 stream 关联的输入缓冲区，方式满足从 stream 的后继读取操作将取得该字符。不修改与流关联的外部设备。

流重寻位操作 fseek 、 fsetpos 和 rewind 弃去 ungetc 的效果。

若调用 ungetc 多于一次，而无中间读取或重寻位，则可能失败（换言之，保证大小为 1 的回放缓冲区，但任何更大的缓冲区是实现定义的）。若成功进行多次 ungetc ，则读取操作以 ungetc 的逆序取得回放的字符。

若 c 等于 EOF ，则操作失败而不影响流。

对 ungetc 的成功调用清除文件尾状态标志 feof 。

在二进制流上对 ungetc 的成功调用将流位置指示器减少一（若流位置指示器为零，则行为不确定）。

在文本流上对 ungetc 的成功调用以未指定方式修改流位置指示器，但保证在以读取操作取得所有回放字符后，流位置指示器等于其在 ungetc 之前的值。

实践中，回放缓冲区的大小会在 4k （ Linux 、 MacOS ）和 4 （ Solaris ）或保证的最小值 1 （ HPUX 、 AIX ）间变化。

若回放的字符等于存在于外部字符序列中该位置的字符，则回放缓冲区的表观大小可以更大（实现可以简单地自减读取的文件位置指示器，并避免维护回放缓冲区）。

* 返回值

成功时返回 ch 。

失败时返回 EOF ，而给定的流保持不变。

* 范例

/\*setbuf 可用于禁用要求立即输出的流的缓冲。\*/

#include <ctype.h>

#include <stdio.h>

void demo\_scanf(const char\* fmt, FILE\* s) {

if(\*fmt == '%') {

int c;

switch(\*++fmt) {

case 'u': while(isspace(c=getc(s))) {} // 跳过空白符

unsigned int num = 0;

while(isdigit(c)) {

num = num\*10 + c-'0';

c = getc(s);

}

printf("%%u scanned %u\n", num);

ungetc(c, s); // 重处理非数字

case 'c': c = getc(s);

printf("%%c scanned '%c'\n", c);

}

}

}

int main(void)

{

FILE\* f = fopen("input.txt", "w+");

fputs("123x", f);

rewind(f);

demo\_scanf("%u%c", f);

fclose(f);

}

* 输出

%u scanned 123

%c scanned 'x'

# 第十一章 进程操作篇

## 11.1 atexit

​设置程序正常结束前调用的函数。

* 相关函数

\_exit,exit,on\_exit

* 表头文件

#include<stdlib.h>

* 定义函数

int atexit(void (\*function)(void));

* 函数说明

注册 func 所指向的函数，使它在程序正常终止（通过 exit() 或从 main() 返回）时得到调用。将以注册顺序的逆序调用这些函数，即首先执行最后调用的函数。

可以注册同一函数多于一次。

atexit 是线程安全的：多个线程调用此函数不会引入数据竞争。

标准保证至少支持注册 32 个函数。准确极限是实现定义的。

* 返回值

如果执行成功则返回0，否则返回-1，失败原因存于errno中。

* 范例

#include <stdlib.h>

#include <stdio.h>

void f1(void)

{

puts("pushed first");

}

void f2(void)

{

puts("pushed second");

}

int main(void)

{

atexit(f1);

atexit(f2);

}

* 执行

pushed second

pushed first

## 11.2 execl

​执行文件。

* 相关函数

fork，execle，execlp，execv，execve，execvp

* 表头文件

#include<unistd.h>

* 定义函数

int execl(const char \*path, const char \*arg, ....);

* 函数说明

execl()用来执行参数path字符串所代表的文件路径，接下来的参数代表执行该文件时传递过去的argv(0)、argv[1]……，最后一个参数必须用空指针(NULL)作结束。 返回值 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno中。

const char \*arg 及其后的省略号可以看作 arg0、arg1、...、argn。它们共同描述了一个或多个指向空尾字符串的指针列表，这些字符串代表了可供执行程序使用的参数列表。按照惯例，第一个参数应指向与被执行文件相关的文件名。参数列表必须以一个空指针结束，由于这些都是可变函数，因此该指针必须被转为 (char \*) NULL。

* 范例

#include<unistd.h>main(){execl(“/bin/ls”,”ls”,”-al”,”/etc/passwd”,(char \* )0);}

* 执行

/\*执行/bin/ls -al /etc/passwd \*/-rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

## 11.3 execlp

​从PATH 环境变量中查找文件并执行。

* 相关函数

fork，execl，execle，execv，execve，execvp

* 表头文件

#include<unistd.h>

* 定义函数

int execlp(const char \*file, const char \*arg, ...);

* 函数说明

execlp()会从PATH 环境变量所指的目录中查找符合参数file的文件名，找到后便执行该文件，然后将第二个以后的参数当做该文件的argv[0]、argv[1]……，最后一个参数必须用空指针(NULL)作结束。

* 返回值

如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno 中。

* 错误代码

参考execve()。

* 范例

/\* 执行ls -al /etc/passwd execlp()会依PATH 变量中的/bin找到/bin/ls \*/#include<unistd.h>main(){execlp(“ls”,”ls”,”-al”,”/etc/passwd”,(char \*)0);}

* 执行

-rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

## 11.4 execv

​执行文件。

* 相关函数

fork，execl，execle，execlp，execve，execvp

* 表头文件

#include<unistd.h>

* 定义函数

int execv(const char \*path, char \*const argv[]);

* 函数说明

execv()用来执行参数path字符串所代表的文件路径，与execl()不同的地方在于execve()只需两个参数，第二个参数利用数组指针来传递给执行文件。

char \*const argv[] 参数是一个指向空尾字符串的数组，表示新程序可用的参数列表。按照惯例，第一个参数应指向与正在执行的文件相关的文件名。指针数组必须以空指针结束。

* 返回值

如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno 中。

* 错误代码

请参考execve（）。

* 范例

/\* 执行/bin/ls -al /etc/passwd \*/#include<unistd.h>main(){char \* argv[ ]={“ls”,”-al”,”/etc/passwd”,(char\*) }};execv(“/bin/ls”,argv);}

* 执行

-rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

## 11.5 execve

​执行文件。

* 相关函数

fork，execl，execle，execlp，execv，execvp

* 表头文件

#include<unistd.h>

* 定义函数

int execve(const char \*filename, char \*const argv[], char \*const envp[]);

* 函数说明

execve()用来执行参数filename字符串所代表的文件路径，第二个参数系利用数组指针来传递给执行文件，最后一个参数则为传递给执行文件的新环境变量数组。

新进程映像的环境由参数 envp 指定。envp 参数是一个指向空端字符串的指针数组，必须以空指针结束。

所有其他 exec() 函数（后缀中不包含 “e”）都从调用进程的外部变量 environ 中获取新进程映像的环境。

* 返回值

如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno 中。

* 错误代码

EACCES欲执行的文件不具有用户可执行的权限。欲执行的文件所属的文件系统是以noexec 方式挂上。欲执行的文件或script翻译器非一般文件。EPERM进程处于被追踪模式，执行者并不具有root权限，欲执行的文件具有SUID 或SGID 位。欲执行的文件所属的文件系统是以nosuid方式挂上，欲执行的文件具有SUID 或SGID 位元，但执行者并不具有root权限。E2BIG 参数数组过大 ENOEXEC 无法判断欲执行文件的执行文件格式，有可能是格式错误或无法在此平台执行。 EFAULT 参数filename所指的字符串地址超出可存取空间范围。 ENAMETOOLONG 参数filename所指的字符串太长。 ENOENT 参数filename字符串所指定的文件不存在。 ENOMEM 核心内存不足 ENOTDIR 参数filename字符串所包含的目录路径并非有效目录 EACCES 参数filename字符串所包含的目录路径无法存取，权限不足 ELOOP 过多的符号连接 ETXTBUSY 欲执行的文件已被其他进程打开而且正把数据写入该文件中 EIO I/O 存取错误 ENFILE 已达到系统所允许的打开文件总数。 EMFILE 已达到系统所允许单一进程所能打开的文件总数。 EINVAL 欲执行文件的ELF执行格式不只一个PT\_INTERP节区 EISDIR ELF翻译器为一目录 ELIBBAD ELF翻译器有问题。

* 范例

#include<unistd.h>main(){char \* argv[ ]={“ls”,”-al”,”/etc/passwd”,(char \*)0};char \* envp[ ]={“PATH=/bin”,0}execve(“/bin/ls”,argv,envp);}

* 执行

-rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

## 11.6 execvp

​执行文件。

* 相关函数

fork，execl，execle，execlp，execv，execve

* 表头文件

#include<unistd.h>

* 定义函数

int execvp(const char \*file, char \*const argv[]);

* 函数说明

execvp()会从PATH 环境变量所指的目录中查找符合参数file 的文件名，找到后便执行该文件，然后将第二个参数argv传给该欲执行的文件。

* 返回值

如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno中。

* 错误代码

请参考execve（）。

* 范例

/\*请与execlp（）范例对照\*/#include<unistd.h>main(){char \* argv[ ] ={ “ls”,”-al”,”/etc/passwd”,0};execvp(“ls”,argv);}

* 执行

-rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd

## 11.7 exit

​正常结束进程。

* 相关函数

\_exit，atexit，on\_exit

* 表头文件

#include<stdlib.h>

* 定义函数

void exit(int status);

* 函数说明

exit() 函数会导致进程正常终止，并将状态的最小有效字节（即 status & 0xFF）返回给父进程（参见 wait(2)）。

所有通过 atexit(3) 和 on\_exit(3) 注册的函数都会被调用，调用顺序与其注册顺序相反。(其中一个函数可以使用 atexit(3) 或 on\_exit(3)注册一个额外的函数，以便在退出处理过程中执行；新注册的函数将被添加到待调用函数列表的前面）。如果其中一个函数没有返回（例如，它调用了 \_exit(2)，或用信号杀死了自己），那么其余函数都不会被调用，进一步的退出处理（尤其是对 stdio(3) 流的刷新）也会被放弃。如果使用 atexit(3) 或 on\_exit(3)多次注册了某个函数，那么它被调用的次数与注册的次数相同。

所有打开的 stdio(3) 流都会被刷新和关闭。由 tmpfile(3) 创建的文件将被删除。

C 标准规定了两个常量 EXIT\_SUCCESS 和 EXIT\_FAILURE，可以传递给 exit()，分别表示成功或失败终止。

* 返回值

如果执行成功则返回0，否则返回-1，失败原因存于errno中。

* 范例

参考wait（）

## 11.8 \_exit

​结束进程执行。

* 相关函数

exit，wait，abort

* 表头文件

#include<unistd.h>

* 定义函数

void \_exit(int status);

* 函数说明

\_exit()用来立刻结束目前进程的执行，并把参数status返回给父进程，并关闭未关闭的文件。此函数调用后不会返回，并且会传递SIGCHLD信号给父进程。属于该进程的任何打开的文件描述符都会被关闭。进程的任何子进程都将被 init(1)（或通过使用 prctl(2) PR\_SET\_CHILD\_SUBREAPER 操作定义的最近的 “subreaper ”进程）继承。进程的父进程会收到 SIGCHLD 信号。返回给父进程的值 status & 0xFF 将作为进程的退出状态，父进程可以使用 wait(2) 系列调用之一来收集该值。

函数 \_Exit() 等同于 \_exit()。

* 附加说明

\_exit（）不会处理标准I/O 缓冲区，如要更新缓冲区请使用exit（）。

## 11.9 vfork

​建立一个新的进程

* 相关函数

wait，execve

* 表头文件

#include<unistd.h>

* 定义函数

pid\_t vfork(void);

* 函数说明

vfork()会产生一个新的子进程，其子进程会复制父进程的数据与堆栈空间，并继承父进程的用户代码，组代码，环境变量、已打开的文件代码、工作目录和资源限制等。

Linux 使用copy-on-write(COW)技术，只有当其中一进程试图修改欲复制的空间时才会做真正的复制动作，由于这些继承的信息是复制而来，并非指相同的内存空间，因此子进程对这些变量的修改和父进程并不会同步。此外，子进程不会继承父进程的文件锁定和未处理的信号。注意，Linux不保证子进程会比父进程先执行或晚执行，因此编写程序时要留意 死锁或竞争条件的发生。

vfork() 函数的作用与 fork(2) 相同，但如果 vfork() 创建的进程修改了用于存储 vfork() 返回值的 pid\_t 类型变量以外的任何数据，或者从调用 vfork() 的函数中返回，或者在成功调用 \_exit(2) 或 exec(3) 系列函数之一之前调用了任何其他函数，那么其行为将是未定义的。

* 返回值

如果vfork()成功则在父进程会返回新建立的子进程代码(PID)，而在新建立的子进程中则返回0。如果vfork 失败则直接返回-1，失败原因存于errno中。

* 错误代码

EAGAIN 内存不足。ENOMEM 内存不足，无法配置核心所需的数据结构空间。

* 范例

#include<unistd.h>main(){if(vfork() = =0){printf(“This is the child process\n”);}else{printf(“This is the parent process\n”);}}

* 执行

this is the parent processthis is the child process

## 11.10 getpgid

​取得进程组识别码。

* 相关函数

setpgid，setpgrp，getpgrp

* 表头文件

#include<unistd.h>

* 定义函数

pid\_t getpgid(pid\_t pid);

* 函数说明

getpgid()用来取得参数pid 指定进程所属的组识别码。如果参数pid为0，则会取得目前进程的组识别码。

* 返回值

执行成功则返回组识别码，如果有错误则返回-1，错误原因存于errno中。

* 错误代码

ESRCH 找不到符合参数pid 指定的进程。

* 范例

/\*取得init 进程（pid＝1）的组识别码\*/#include<unistd.h>mian(){printf(“init gid = %d\n”,getpgid(1));}

* 执行

init gid = 0

## 11.11 getpgrp

​取得进程组识别码。

* 相关函数

setpgid，getpgid，getpgrp

* 表头文件

#include<unistd.h>

* 定义函数

pid\_t getpgrp(void);

* 函数说明

getpgrp()用来取得目前进程所属的组识别码。此函数相当于调用getpgid(0)；

* 返回值

返回目前进程所属的组识别码。

* 范例

#include<unistd.h>main(){printf(“my gid =%d\n”,getpgrp());}

* 执行

my gid =29546

## 11.12 getpid

​取得进程识别码。

* 相关函数

fork，kill，getpid

* 表头文件

#include<unistd.h>

* 定义函数

pid\_t getpid(void);

* 函数说明

getpid（）用来取得目前进程的进程识别码，许多程序利用取到的此值来建立临时文件，以避免临时文件相同带来的问题。

* 返回值

目前进程的进程识别码

* 范例

#include<unistd.h>main(){printf(“pid=%d\n”,getpid());}

* 执行

pid=1494 /\*每次执行结果都不一定相同\*/

## 11.13 getppid

​取得父进程的进程识别码。

* 相关函数

fork，kill，getpid

* 表头文件

#include<unistd.h>

* 定义函数

pid\_t getppid(void);

* 函数说明

getppid() 返回调用进程的父进程 ID。这要么是使用 fork() 创建此进程的进程的 ID，要么是（如果该进程已终止）此进程被重新分配给的进程的 ID（要么是（如果该进程已终止）此进程被重新分配给的进程的 ID）。

如果该进程已经终止，则返回该进程的父进程 ID（init(1) 或通过 prctl(2) PR\_SET\_CHILD\_SUBREAPER 操作定义的 “subreaper ”进程）。

* 返回值

目前进程的父进程识别码。

* 范例

#include<unistd.h>main(){printf(“My parent ‘pid =%d\n”,getppid());}

* 执行

My parent pid =463

## 11.14 getpriority

​取得程序进程执行优先权。

* 相关函数

setpriority，nice

* 表头文件

#include<sys/time.h>#include<sys/resource.h>

* 定义函数

int getpriority(int which, int who);

* 函数说明

getpriority()可用来取得进程、进程组和用户的进程执行优先权。

进程、进程组或用户的调度优先级，由 which 和 who 表示，可通过 getpriority() 调用获得，也可通过 setpriority() 调用设置。这些系统调用处理的进程属性与 nice(2) 处理的属性（也称为 “nice ”值）相同。

值是 PRIO\_PROCESS、PRIO\_PGRP 或 PRIO\_USER 中的一个，并且是相对于 which（PRIO\_PROCESS 的进程标识符、PRIO\_PGRP 的进程组标识符和 PRIO\_USER 的用户 ID）而言的。如果 who 的值为零，则分别表示调用进程、调用进程的进程组或调用进程的真实用户 ID。

传统上，只有拥有特权的进程才能降低好数值（即设置更高的优先级）。不过，自 Linux 2.6.12 起，非特权进程可以降低具有合适的 RLIMIT\_NICE 软限制的目标进程的 nice 值；详情请参阅 getrlimit(2)。

* 参数

which有三种数值，参数who 则依which值有不同定义 which who 代表的意义 PRIO\_PROCESS who 为进程识别码 PRIO\_PGRP who 为进程的组识别码 PRIO\_USER who 为用户识别码 此函数返回的数值介于-20 至20之间，代表进程执行优先权，数值越低代表有较高的优先次序，执行会较频繁。

* 返回值

成功时，getpriority() 返回调用线程的nice值，该值可能是负数。如果出错，则返回 -1，并设置 errno 表示出错。

由于成功调用 getpriority() 可以合法返回值-1，因此有必要在调用前清除 errno，然后检查 errno，以确定-1 是错误还是合法值。

* 附加说明

由于返回值有可能是-1，因此要同时检查errno是否存有错误原因。最好在调用次函数前先清除errno变量。

* 错误代码

ESRCH 参数which或who 可能有错，而找不到符合的进程。EINVAL 参数which 值错误。

## 11.15 nice

​改变进程优先顺序。

* 相关函数

setpriority，getpriority

* 表头文件

#include<unistd.h>

* 定义函数

int nice(int inc);

* 函数说明

nice()用来改变进程的进程执行优先顺序。参数inc数值越大则优先顺序排在越后面，即表示进程执行会越慢。只有超级用户才能使用负的inc 值，代表优先顺序排在前面，进程执行会较快。

nice() 会为调用线程的 nice 值加上 inc。(nice 值越高，优先级越低）。

nice 值的范围是 +19（低优先级）到 -20（高优先级）。如果试图设置超出范围的 nice 值，则会被箝制在该范围内。

传统上，只有特权进程才能降低 nice 值（即设置更高的优先级）。不过，自 Linux 2.6.12 起，非特权进程可以降低目标进程的 nice 值，只要目标进程有合适的 RLIMIT\_NICE 软限制；详情请参阅 getrlimit(2)。

* 返回值

成功时，将返回新的 nice 值（但请参阅下面的注释）。如果出错，则返回 -1，并设置 errno 表示出错。

成功调用可以合法地返回-1。要检测错误，可在调用前将 errno 设为 0，并在 nice() 返回-1 后检查它是否为非 0。

* 错误代码

EPERM 一般用户企图转用负的参数inc值改变进程优先顺序。

## 11.16 on\_exit

​设置程序正常结束前调用的函数。

* 相关函数

\_exit，atexit，exit

* 表头文件

#include<stdlib.h>

* 定义函数

int on\_exit(void (\*function)(int, void \*), void \*arg);

* 函数说明

on\_exit()用来设置一个程序正常结束前调用的函数。当程序通过调用exit()或从main中返回时，参数function所指定的函数会先被调用，然后才真正由exit()结束程序。参数arg指针会传给参数function函数，详细情况请见范例。

同一个函数可以注册多次：每次注册都会调用一次。

通过 fork(2) 创建子进程时，它会继承父进程的注册副本。

在成功调用 exec(3) 函数后，所有注册信息都会被删除。

* 返回值

如果执行成功则返回0，否则返回-1，失败原因存于errno中。

* 范例

#include<stdlib.h>void my\_exit(int status,void \*arg){printf(“before exit()!\n”);printf(“exit (%d)\n”,status);printf(“arg = %s\n”,(char\*)arg);}main(){char \* str=”test”;on\_exit(my\_exit,(void \*)str);exit(1234);}

* 执行

before exit()!exit (1234)arg = test

## 11.17 setpgid

​设置进程组识别码。

* 相关函数

getpgid，setpgrp，getpgrp

* 表头文件

#include<unistd.h>

* 定义函数

int setpgid(pid\_t pid, pid\_t pgid);

* 函数说明

setpgid()将参数pid 指定进程所属的组识别码设为参数pgid 指定的组识别码。如果参数pid 为0，则会用来设置目前进程的组识别码，如果参数pgid为0，则会以目前进程的进程识别码来取代。

* 返回值

执行成功则返回组识别码，如果有错误则返回-1，错误原因存于errno中。

* 错误代码

EINVAL 参数pgid小于0。 EPERM 进程权限不足，无法完成调用。 ESRCH 找不到符合参数pid指定的进程。

## 11.18 setpgrp

​设置进程组识别码。

* 相关函数

getpgid，setpgid，getpgrp

* 表头文件

#include<unistd.h>

* 定义函数

int setpgrp(void);

* 函数说明

setpgrp()将目前进程所属的组识别码设为目前进程的进程识别码。此函数相当于调用setpgid(0,0)。

* 返回值

执行成功则返回组识别码，如果有错误则返回-1，错误原因存于errno中。

## 11.19 setpriority

​设置程序进程执行优先权。

* 相关函数

getpriority，nice

* 表头文件

#include<sys/time.h>#include<sys/resource.h>

* 定义函数

int setpriority(int which, int who, int prio);

* 函数说明

进程、进程组或用户的调度优先级，由 which 和 who 表示，可通过 getpriority() 调用获得，也可通过 setpriority() 调用设置。这些系统调用处理的进程属性与 nice(2) 处理的属性（也称为 "nice "值）相同。

which是 PRIO\_PROCESS、PRIO\_PGRP 或 PRIO\_USER 中的一个，并且who是相对于 which（PRIO\_PROCESS 的进程标识符、PRIO\_PGRP 的进程组标识符和 PRIO\_USER 的用户 ID）而言的。如果 who 的值为零，则分别表示调用进程、调用进程的进程组或调用进程的真实用户 ID。

prio 参数的取值范围为 -20 至 19（但请参阅下面的注释），其中 -20 为最高优先级，19 为最低优先级。如果试图在此范围之外设置优先级，则会被静默箝制在此范围内。默认的优先级为 0；数值越小，进程的调度优先级越高。

调用 getpriority() 会返回指定进程享有的最高优先级（最低数值）。setpriority() 调用会将所有指定进程的优先级设置为指定值。

传统上，只有拥有特权的进程才能降低好数值（即设置更高的优先级）。不过，自 Linux 2.6.12 起，非特权进程可以降低具有合适的 RLIMIT\_NICE 软限制的目标进程的 nice 值；详情请参阅 getrlimit(2)。

* 返回值

执行成功则返回0，如果有错误发生返回值则为-1，错误原因存于errno。 ESRCH 参数which或who 可能有错，而找不到符合的进程 EINVAL 参数which值错误。 EPERM 权限不够，无法完成设置 EACCES 一般用户无法降低优先权

## 11.20 system

​执行shell 命令。

* 相关函数

fork，execve，waitpid，popen

* 表头文件

#include<stdlib.h>

* 定义函数

int system(const char \*string);

* 函数说明

system()会调用fork()产生子进程，由子进程来调用/bin/sh-c string来执行参数string字符串所代表的命令，此命令执行完后随即返回原调用的进程。在调用system()期间SIGCHLD 信号会被暂时搁置，SIGINT和SIGQUIT 信号则会被忽略。

* 返回值

如果system()在调用/bin/sh时失败则返回127，其他失败原因返回-1。若参数string为空指针(NULL)，则返回非零值。如果system()调用成功则最后会返回执行shell命令后的返回值，但是此返回值也有可能为system()调用/bin/sh失败所返回的127，因此最好能再检查errno 来确认执行成功。

* 附加说明

在编写具有SUID/SGID权限的程序时请勿使用system()，system()会继承环境变量，通过环境变量可能会造成系统安全的问题。

* 范例

#include<stdlib.h>main(){system(“ls -al /etc/passwd /etc/shadow”);}

* 执行

-rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd-r--------- 1 root root 572 Sep 2 15 :34 /etc/shadow

## 11.21 wait

​等待子进程中断或结束。

* 相关函数

waitpid，fork

* 表头文件

#include<sys/types.h>#include<sys/wait.h>

* 定义函数

pid\_t wait(int \*status);

* 函数说明

wait()会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用wait()时子进程已经结束，则wait()会立即返回子进程结束状态值。子进程的结束状态值会由参数status 返回，而子进程的进程识别码也会一快返回。

所有这些系统调用都用于等待调用进程的子进程发生状态变化，并获取状态发生变化的子进程的信息。状态变化被认为是：子进程终止；子进程被信号停止；或子进程被信号恢复。在子进程终止的情况下，执行等待可让系统释放与子进程相关的资源；如果不执行等待，则已终止的子进程将保持 “僵尸 ”状态（见下文注释）。

如果子进程已经改变了状态，那么这些调用会立即返回。否则，这些调用将阻塞，直到子进程改变状态或信号处理程序中断调用（假设系统调用没有使用 sigaction(2) 的 SA\_RESTART 标志自动重启）。在本页的其余部分，状态已发生变化但尚未被这些系统调用等待的子进程被称为可等待子进程。

* 参数

status可以设成NULL。子进程的结束状态值请参考waitpid()。

* 返回值

如果执行成功则返回子进程识别码(PID)，如果有错误发生则返回-1。失败原因存于errno中。

* 范例

#include<stdlib.h>#include<unistd.h>#include<sys/types.h>#include<sys/wait.h>main(){pid\_t pid;int status,i;if(fork()= =0){printf(“This is the child process .pid =%d\n”,getpid());exit(5);}else{sleep(1);printf(“This is the parent process ,wait for child...\n”;pid=wait(&status);i=WEXITSTATUS(status);printf(“child’s pid =%d .exit status=^d\n”,pid,i);}}

* 执行

This is the child process.pid=1501This is the parent process .wait for child...child’s pid =1501,exit status =5

## 11.22 waitpid

​等待子进程中断或结束。

* 相关函数

wait，fork

* 表头文件

#include<sys/types.h>#include<sys/wait.h>

* 定义函数

pid\_t waitpid(pid\_t pid, int \*status, int options);

* 函数说明

系统调用 wait() 会暂停调用线程的执行，直到其一个子线程终止。wait(&wstatus)调用等价于：waitpid(-1, &wstatus, 0); waitpid() 系统调用会暂停调用线程的执行，直到 pid 参数指定的子线程改变状态。默认情况下，waitpid() 只等待已终止的子线程，但这一行为可通过选项参数修改，如下所述。

pid 的值可以是

< -1 表示等待进程组 ID 等于 pid 绝对值的任何子进程。

Linux 手册 6.8 2024-05-02 1177

系统调用手册 wait(2)

-1表示等待任何子进程。

0 表示等待进程组 ID 与调用进程时的进程组 ID 相同的任何子进程。

0 表示等待调用 waitpid() 时进程组 ID 等于调用进程组 ID 的任何子进程。

> 0 表示等待进程 ID 等于 pid 值的子进程。

参数option可以为0 或下面的OR 组合 WNOHANG 如果没有任何已经结束的子进程则马上返回，不予以等待。

WUNTRACED 如果子进程进入暂停执行情况则马上返回，但结束状态不予以理会。

子进程的结束状态返回后存于status，底下有几个宏可判别结束情况

WIFEXITED(status)如果子进程正常结束则为非0值。

WEXITSTATUS(status)取得子进程exit()返回的结束代码，一般会先用WIFEXITED 来判断是否正常结束才能使用此宏。 WIFSIGNALED(status)如果子进程是因为信号而结束则此宏值为真

WTERMSIG(status)取得子进程因信号而中止的信号代码，一般会先用WIFSIGNALED 来判断后才使用此宏。 WIFSTOPPED(status)如果子进程处于暂停执行情况则此宏值为真。一般只有使用WUNTRACED 时才会有此情况。 WSTOPSIG(status)取得引发子进程暂停的信号代码，一般会先用WIFSTOPPED 来判断后才使用此宏。

* 返回值

如果执行成功则返回子进程识别码(PID)，如果有错误发生则返回-1。失败原因存于errno中。

* 范例

参考wait()。

# 第十二章 格式化输入输出篇

## 12.1 fprintf

​格式化输出数据至文件。

* 相关函数

printf，fscanf，vfprintf

* 表头文件

#include<stdio.h>

* 定义函数

int fprintf(FILE \*stream, const char \*format, ...);

* 函数说明

fprintf()会根据参数format字符串来转换并格式化数据，然后将结果输出到参数stream指定的文件中，直到出现字符串结束('\0')为止。成功返回后，这些函数将返回打印的字节数（不包括用于结束字符串输出的空字节）。如果遇到输出错误，则返回负值。

* 返回值

关于参数format字符串的格式请参考printf()。成功则返回实际输出的字符数，失败则返回-1，错误原因存于errno中。

* 范例

#include<stdio.h>main(){int i = 150;int j = -100;double k = 3.14159;fprintf(stdout,”%d %f %x \n”,j,k,i);fprintf(stdout,”%2d %\*d\n”,i,2,i);}

* 执行

-100 3.141590 96150 150

## 12.2 fscanf

​格式化字符串输入。

* 相关函数

scanf，sscanf

* 表头文件

#include<stdio.h>

* 定义函数

int fscanf(FILE \*stream, const char \*format, ...);

* 函数说明

fscanf()会自参数stream的文件流中读取字符串，再根据参数format字符串来转换并格式化数据。格式转换形式请参考scanf()。转换后的结构存于对应的参数内。

* 返回值

成功则返回参数数目，失败则返回-1，错误原因存于errno中。

* 范例

#include<stdio.h>main(){int i;unsigned int j;char s[5];fscanf(stdin,”%d %x %5[a-z] %\*s %f”,&i,&j,s,s);printf(“%d %d %s \n”,i,j,s);}

* 执行

10 0x1b aaaaaaaaa bbbbbbbbbb /\*从键盘输入\*/10 27 aaaaa

## 12.3 printf

​格式化输出数据。

* 相关函数

scanf，snprintf

* 表头文件

#include<stdio.h>

* 定义函数

int printf(const char \*format, ...);

* 函数说明

printf()会根据参数format字符串来转换并格式化数据，然后将结果写出到标准输出设备，直到出现字符串结束('\0')为止。参数format字符串可包含下列三种字符类型一般文本，伴随直接输出。ASCII控制字符，如\t、\n等。格式转换字符。格式转换为一个百分比符号(％)及其后的格式字符所组成。一般而言，每个％符号在其后都必需有一printf()的参数与之相呼应（只有当％％转换字符出现时会直接输出％字符），而欲输出的数据类型必须与其相对应的转换字符类型相同。Printf()格式转换的一般形式如下％(flags)(width)(.prec)type以中括号括起来的参数为选择性参数，而％与type则是必要的。底下先介绍type的几种形式整数％d 整数的参数会被转成一有符号的十进制数字 ％u 整数的参数会被转成一无符号的十进制数字 ％o 整数的参数会被转成一无符号的八进制数字 ％x 整数的参数会被转成一无符号的十六进制数字，并以小写abcdef表示 ％X 整数的参数会被转成一无符号的十六进制数字，并以大写ABCDEF表示浮点型数 ％f double 型的参数会被转成十进制数字，并取到小数点以下六位，四舍五入。 ％e double型的参数以指数形式打印，有一个数字会在小数点前，六位数字在小数点后，而在指数部分会以小写的e来表示。 ％E 与％e作用相同，唯一区别是指数部分将以大写的E 来表示。 ％g double 型的参数会自动选择以％f 或％e 的格式来打印，其标准是根据欲打印的数值及所设置的有效位数来决定。 ％G 与％g 作用相同，唯一区别在以指数形态打印时会选择％E 格式。字符及字符串％c 整型数的参数会被转成unsigned char型打印出。 ％s 指向字符串的参数会被逐字输出，直到出现NULL字符为止 ％p 如果是参数是“void \*”型指针则使用十六进制格式显示。prec 有几种情况正整数的最小位数。在浮点型数中代表小数位数在％g 格式代表有效位数的最大值。在％s格式代表字符串的最大长度。若为×符号则代表下个参数值为最大长度。width为参数的最小长度，若此栏并非数值，而是\*符号，则表示以下一个参数当做参数长度。flags 有下列几种情况#NAME?+ 一般在打印负数时，printf（）会加印一个负号，整数则不加任何负号。此旗标会使得在打印正数前多一个正号（+）。 # 此旗标会根据其后转换字符的不同而有不同含义。当在类型为o 之前（如％#o），则会在打印八进制数值前多印一个o。 而在类型为x 之前（％#x）则会在打印十六进制数前多印’0x’，在型态为e、E、f、g或G 之前则会强迫数值打印小数点。在类型为g 或G之前时则同时保留小数点及小数位数末尾的零。 0 当有指定参数时，无数字的参数将补上0。默认是关闭此旗标，所以一般会打印出空白字符。

* 返回值

成功则返回实际输出的字符数，失败则返回-1，错误原因存于errno中。

* 范例

#include<stdio.h>main(){int i = 150;int j = -100;double k = 3.14159;printf(“%d %f %x\n”,j,k,i);printf(“%2d %\*d\n”,i,2,i); /\*参数2 会代入格式\*中，而与%2d同意义\*/}

* 执行

-100 3.14159 96150 150

## 12.4 scanf

​格式化字符串输入。

* 相关函数

fscanf，snprintf

* 表头文件

#include<stdio.h>

* 定义函数

int scanf(const char \*format, ...);

* 函数说明

scanf()会将输入的数据根据参数format字符串来转换并格式化数据。Scanf()格式转换的一般形式如下％[\*][size][l][h]type以中括号括起来的参数为选择性参数，而％与type则是必要的。\* 代表该对应的参数数据忽略不保存。 size 为允许参数输入的数据长度。 l 输入的数据数值以long int 或double型保存。 h 输入的数据数值以short int 型保存。底下介绍type的几种形式％d 输入的数据会被转成一有符号的十进制数字（int）。 ％i 输入的数据会被转成一有符号的十进制数字，若输入数据以“0x”或“0X”开头代表转换十六进制数字，若以“0”开头则转换八进制数字，其他情况代表十进制。 ％0 输入的数据会被转换成一无符号的八进制数字。 ％u 输入的数据会被转换成一无符号的正整数。 ％x 输入的数据为无符号的十六进制数字，转换后存于unsigned int型变量。 ％X 同％x ％f 输入的数据为有符号的浮点型数，转换后存于float型变量。 ％e 同％f ％E 同％f ％g 同％f ％s 输入数据为以空格字符为终止的字符串。 ％c 输入数据为单一字符。 [] 读取数据但只允许括号内的字符。如[a-z]。 读取数据但不允许中括号的^符号后的字符出现，如0-9.

* 返回值

成功则返回参数数目，失败则返回-1，错误原因存于errno中。

* 范例

#include <stdio.h>main(){int i;unsigned int j;char s[5];scanf(“%d %x %5[a-z] %\*s %f”,&i,&j,s,s);printf(“%d %d %s\n”,i,j,s);}

* 执行

10 0x1b aaaaaaaaaa bbbbbbbbbb10 27 aaaaa

## 12.5 sprintf

​格式化字符串复制。

* 相关函数

printf，sprintf

* 表头文件

#include<stdio.h>

* 定义函数

int sprintf(char \*str, const char \*format, ...);

* 函数说明

sprintf()会根据参数format字符串来转换并格式化数据，然后将结果复制到参数str所指的字符串数组，直到出现字符串结束(’\0’)为止。关于参数format字符串的格式请参考printf()。

* 返回值

成功则返回参数str字符串长度，失败则返回-1，错误原因存于errno中。

* 附加说明

使用此函数得留意堆栈溢出，或改用snprintf（）。

* 范例

#include<stdio.h>main(){char \* a=”This is string A!”;char buf[80];sprintf(buf,”>>> %s<<<\n”,a);printf(“%s”.buf);}

* 执行

>>>This is string A!<<<

## 12.6 sscanf

​格式化字符串输入。

* 相关函数

scanf，fscanf

* 表头文件

#include<stdio.h>

* 定义函数

int sscanf(const char \*str, const char \*format, ...);

* 函数说明

sscanf()会将参数str的字符串根据参数format字符串来转换并格式化数据。格式转换形式请参考scanf()。转换后的结果存于对应的参数内。

* 返回值

成功则返回参数数目，失败则返回-1，错误原因存于errno中。

* 范例

#include<stdio.h>main(){int i;unsigned int j;char input[ ]=”10 0x1b aaaaaaaa bbbbbbbb”;char s[5];sscanf(input,”%d %x %5[a-z] %\*s %f”,&i,&j,s,s);printf(“%d %d %s\n”,i,j,s);}

* 执行

10 27 aaaaa

## 12.7 vfprintf

​格式化输出数据至文件。

* 相关函数

printf，fscanf，fprintf

* 表头文件

#include<stdio.h>#include<stdarg.h>

* 定义函数

int vfprintf(FILE \*stream, const char \*format, va\_list ap);

* 函数说明

vfprintf()会根据参数format字符串来转换并格式化数据，然后将结果输出到参数stream指定的文件中，直到出现字符串结束(’\0’)为止。关于参数format字符串的格式请参考printf()。va\_list用法请参考附录C或vprintf()范例。

* 返回值

成功则返回实际输出的字符数，失败则返回-1，错误原因存于errno中。

* 范例

参考fprintf()及vprintf()。

## 12.8 vfscanf

​格式化字符串输入。

* 相关函数

scanf，sscanf，fscanf

* 表头文件

#include<stdio.h>

* 定义函数

int vfscanf(FILE \*stream, const char \*format, va\_list ap);

* 函数说明

vfscanf()会自参数stream 的文件流中读取字符串，再根据参数format字符串来转换并格式化数据。格式转换形式请参考scanf()。转换后的结果存于对应的参数内。va\_list用法请参考附录C 或vprintf()。

* 返回值

成功则返回参数数目，失败则返回-1，错误原因存于errno中。

* 范例

参考fscanf()及vprintf()。

## 12.9 vprintf

​格式化输出数据。

* 相关函数

printf，vfprintf，vsprintf

* 表头文件

#include<stdio.h>#include<stdarg.h>

* 定义函数

int vprintf(const char \*format, va\_list ap);

* 函数说明

vprintf()作用和printf()相同，参数format格式也相同。va\_list为不定个数的参数列，用法及范例请参考附录C。

* 返回值

成功则返回实际输出的字符数，失败则返回-1，错误原因存于errno中。

* 范例

#include<stdio.h>#include<stdarg.h>int my\_printf( const char \*format,……){va\_list ap;int retval;va\_start(ap,format);printf(“my\_printf( ):”);retval = vprintf(format,ap);va\_end(ap);return retval;}main(){int i = 150,j = -100;double k = 3.14159;my\_printf(“%d %f %x\n”,j,k,i);my\_printf(“%2d %\*d\n”,i,2,i);}

* 执行

my\_printf() : -100 3.14159 96my\_printf() : 150 150

## 12.10 vscanf

​格式化字符串输入。

* 相关函数

vsscanf，vfscanf

* 表头文件

#include<stdio.h>#include<stdarg.h>

* 定义函数

int vscanf(const char \*format, va\_list ap);

* 函数说明

vscanf()会将输入的数据根据参数format字符串来转换并格式化数据。格式转换形式请参考scanf()。转换后的结果存于对应的参数内。va\_list用法请参考附录C或vprintf()范例。

* 返回值

成功则返回参数数目，失败则返回-1，错误原因存于errno中。

* 范例

请参考scanf()及vprintf()。

## 12.11 vsprintf

​格式化字符串复制。

* 相关函数

vnsprintf，vprintf，snprintf

* 表头文件

#include<stdio.h>

* 定义函数

int vsprintf(char \*str, const char \*format, va\_list ap);

* 函数说明

vsprintf()会根据参数format字符串来转换并格式化数据，然后将结果复制到参数str所指的字符串数组，直到出现字符串结束(’\0’)为止。关于参数format字符串的格式请参考printf()。va\_list用法请参考附录C或vprintf()范例。

* 返回值

成功则返回参数str字符串长度，失败则返回-1，错误原因存于errno中。

* 范例

请参考vprintf()及vsprintf()。

## 12.12 vsscanf

​格式化字符串输入

* 相关函数

vscanf，vfscanf

* 表头文件

#include<stdio.h>

* 定义函数

int vsscanf(const char \*str, const char \*format, va\_list ap);

* 函数说明

vsscanf()会将参数str的字符串根据参数format字符串来转换并格式化数据。格式转换形式请参考附录C 或vprintf()范例。

* 返回值

成功则返回参数数目，失败则返回-1，错误原因存于errno中。

* 范例

请参考sscanf()及vprintf()。

# 第十三章 文件权限控制篇

## 13.1 access

​判断是否具有存取文件的权限。

* 相关函数

stat，open，chmod，chown，setuid，setgid

* 表头文件

#include<unistd.h>

* 定义函数

int access(const char \*pathname, int mode);

* 函数说明

access()会检查是否可以读/写某一已存在的文件。参数mode有几种情况组合，R\_OK，W\_OK，X\_OK 和F\_OK。R\_OK，W\_OK与X\_OK用来检查文件是否具有读取、写入和执行的权限。F\_OK则是用来判断该文件是否存在。由于access()只作权限的核查，并不理会文件形态或文件内容，因此，如果一目录表示为“可写入”，表示可以在该目录中建立新文件等操作，而非意味此目录可以被当做文件处理。例如，你会发现DOS的文件都具有“可执行”权限，但用execve()执行时则会失败。

* 返回值

若所有欲查核的权限都通过了检查则返回0值，表示成功，只要有一权限被禁止则返回-1。

* 错误代码

EACCESS 参数pathname 所指定的文件不符合所要求测试的权限。 EROFS 欲测试写入权限的文件存在于只读文件系统内。 EFAULT 参数pathname指针超出可存取内存空间。 EINVAL 参数mode 不正确。 ENAMETOOLONG 参数pathname太长。 ENOTDIR 参数pathname为一目录。 ENOMEM 核心内存不足 ELOOP 参数pathname有过多符号连接问题。 EIO I/O 存取错误。

* 附加说明

使用access()作用户认证方面的判断要特别小心，例如在access()后再做open()的空文件可能会造成系统安全上的问题。

* 范例

/\* 判断是否允许读取/etc/passwd \*/#include<unistd.h>int main(){if (access(“/etc/passwd”,R\_OK) = =0)printf(“/etc/passwd can be read\n”);}

* 执行

/etc/passwd can be read

## 13.2 alphasort

​依字母顺序排序目录结构。

* 相关函数

scandir，qsort

* 表头文件

#include<dirent.h>

* 定义函数

int alphasort(const struct dirent \*\*a, const struct dirent \*\*b);

* 函数说明

alphasort()为scandir()最后调用qsort()函数时传给qsort()作为判断的函数，详细说明请参考scandir()及qsort()。

alphasort() 和 versionsort() 函数可用作比较函数 compar()。前者使用 strcoll(3) 对目录条目进行排序，后者使用 strverscmp(3) 对字符串 (\*a)->d\_name 和 (\*b)->d\_name 进行排序。

* 返回值

如果第一个参数被认为分别小于、等于或大于第二个参数，则 alphasort() 和 versionsort() 函数返回一个小于、等于或大于零的整数。

* 范例

/\* 读取/目录下所有的目录结构，并依字母顺序排列\*/main(){struct dirent \*\*namelist;int i,total;total = scandir(“/”,&namelist ,0,alphasort);if(total <0)perror(“scandir”);else{for(i=0;i<total;i++)printf(“%s\n”,namelist[i]->d\_name);printf(“total = %d\n”,total);}}

* 执行

...gnome.gnome\_privateErrorLogWeblogbinbootdevdoscdosdetchomeliblost+foundmiscmntoptprocrootsbintmpusrvartotal = 24

## 13.3 chdir

​改变当前的工作（目录）。

* 相关函数

getcwd，chroot

* 表头文件

#include<unistd.h>

* 定义函数

int chdir(const char \*path);

* 函数说明

chdir()用来将当前的工作目录改变成以参数path所指的目录。

* 返回值

执行成功则返回0，失败返回-1，errno为错误代码。

* 范例

#include<unistd.h>main(){chdir(“/tmp”);printf(“current working directory: %s\n”,getcwd(NULL,NULL));}

* 执行

current working directory :/tmp

## 13.4 chmod

​改变文件的权限。

* 相关函数

fchmod，stat，open，chown

* 表头文件

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

* 定义函数

int chmod(const char \*path, mode\_t mode);

* 函数说明

chmod()会依参数mode 权限来更改参数path 指定文件的权限。

chmod()系统调用可以更改文件的模式位（文件模式由文件权限位和设置用户 ID、设置组 ID 和粘性位组成）。(文件模式由文件权限位加上设置用户 ID、设置组 ID 和粘性位组成）。

- chmod() 更改指定文件的模式，其路径名在 path中给出，如果是符号链接，则取消引用。

新的文件模式在 mode 中指定，它是由以下零个或多个值组成的位掩码：

* 返回值

权限改变成功返回0，失败返回-1，错误原因存于errno。

* 错误代码

EPERM 进程的有效用户识别码与欲修改权限的文件拥有者不同，而且也不具root权限。

EACCESS 参数path所指定的文件无法存取。

EROFS 欲写入权限的文件存在于只读文件系统内。

EFAULT 参数path指针超出可存取内存空间。

EINVAL 参数mode不正确

ENAMETOOLONG 参数path太长

ENOENT 指定的文件不存在

ENOTDIR 参数path路径并非一目录

ENOMEM 核心内存不足

ELOOP 参数path有过多符号连接问题。

EIO I/O 存取错误

* 参数

mode 有下列数种组合

S\_ISUID 04000 文件的（set user-id on execution）位

S\_ISGID 02000 文件的（set group-id on execution）位

S\_ISVTX 01000 文件的sticky位

S\_IRUSR（S\_IREAD） 00400 文件所有者具可读取权限

S\_IWUSR（S\_IWRITE）00200 文件所有者具可写入权限

S\_IXUSR（S\_IEXEC） 00100 文件所有者具可执行权限

S\_IRGRP 00040 用户组具可读取权限

S\_IWGRP 00020 用户组具可写入权限

S\_IXGRP 00010 用户组具可执行权限

S\_IROTH 00004 其他用户具可读取权限

S\_IWOTH 00002 其他用户具可写入权限

S\_IXOTH 00001 其他用户具可执行权限 只有该文件的所有者或有效用户识别码为0，才可以修改该文件权限。

基于系统安全，如果欲将数据写入一执行文件，而该执行文件具有S\_ISUID 或S\_ISGID 权限，则这两个位会被清除。如果一目录具有S\_ISUID 位权限，表示在此目录下只有该文件的所有者或root可以删除该文件。

* 范例

/\* 将/etc/passwd 文件权限设成S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH \*/#include<sys/types.h>#include<sys/stat.h>main(){chmod(“/etc/passwd”,S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);}

## 13.5 chown

​改变文件的所有者。

* 相关函数

fchown，lchown，chmod

* 表头文件

#include<sys/types.h>#include<unistd.h>

* 定义函数

int chown(const char \*path, uid\_t owner, gid\_t group);

* 函数说明

chown()会将参数path指定文件的所有者变更为参数owner代表的用户，而将该文件的组变更为参数group组。

如果参数owner或group为-1，对应的所有者或组不会有所改变。root与文件所有者皆可改变文件组，但所有者必须是参数group组的成员。

当root用chown()改变文件所有者或组时，该文件若具有S\_ISUID或S\_ISGID权限，则会清除此权限位，此外如果具有S\_ISGID权限但不具S\_IXGRP位，则该文件会被强制锁定，文件模式会保留。

* 返回值

成功则返回0，失败返回-1，错误原因存于errno。

* 错误代码

参考chmod（）。

* 范例

/\* 将/etc/passwd 的所有者和组都设为root \*/#include<sys/types.h>#include<unistd.h>main(){chown(“/etc/passwd”,0,0);}

## 13.6 chroot

​改变根目录。

* 相关函数

Chdir

* 表头文件

#include<unistd.h>

* 定义函数

int chroot(const char \*path);

* 函数说明

chroot() 会将调用进程的根目录更改为 path 中指定的目录。调用进程的所有子进程都将继承根目录。

只有特权进程（Linux：在其用户名空间中具有 CAP\_SYS\_CHROOT 功能的进程）才能调用 chroot()。

该调用会改变路径名解析过程中的一个成分，而不会做其他任何事情。尤其是，它并不打算用于任何类型的安全目的，既不是为了完全沙箱进程，也不是为了限制文件系统的系统调用。过去，守护进程在将不信任用户提供的路径传递给open(2) 等系统调用之前，会使用 chroot() 对自身进行限制。但是，如果文件夹被移出 chroot 目录，攻击者就可以利用这一点将文件夹也移出 chroot 目录。

最简单的方法是 chdir(2)到要移动的目录，等待它被移出，然后打开类似 ../../../etc/passwd这样的路径。

如果不允许使用 chdir(2)，在某些情况下还可以使用一个稍微麻烦一点的变种。如果守护进程允许指定 “chroot 目录”，这通常意味着如果要防止远程用户访问 chroot 目录之外的文件，就必须确保文件夹永远不会被移出该目录。

此调用不会关闭打开的文件描述符，而此类文件描述符可能允许访问 chroot 树之外的文件。

* 返回值

调用成功则返回0，失败则返-1，错误代码存于errno。

* 错误代码

EPERM 权限不足，无法改变根目录。 EFAULT 参数path指针超出可存取内存空间。 ENAMETOOLONG 参数path太长。 ENOTDIR 路径中的目录存在但却非真正的目录。 EACCESS 存取目录时被拒绝 ENOMEM 核心内存不足。 ELOOP 参数path有过多符号连接问题。 EIO I/O 存取错误。

* 范例

/\* 将根目录改为/tmp ,并将工作目录切换至/tmp \*/#include<unistd.h>main(){chroot(“/tmp”);chdir(“/”);}

## 13.7 closedir

​关闭目录。

* 相关函数

Opendir

* 表头文件

#include<sys/types.h>#include<dirent.h>

* 定义函数

int closedir(DIR \*dir);

* 函数说明

函数 closedir() 关闭了与 dir 相关的目录流。成功调用 closedir() 也会关闭与 dir 相关的底层文件描述符。调用此函数后，目录流描述符 dir 将不可用。

* 返回值

关闭成功则返回0，失败返回-1，错误原因存于errno 中。

* 错误代码

EBADF 参数dir为无效的目录流

* 附加说明

使用access()作用户认证方面的判断要特别小心，例如在access()后再做open()的空文件可能会造成系统安全上的问题。

* 范例

参考readir()。

## 13.8 fchdir

​改变当前的工作目录。

* 相关函数

getcwd，chroot

* 表头文件

#include<unistd.h>

* 定义函数

int fchdir(int fd);

* 函数说明

fchdir()用来将当前的工作目录改变成以参数fd 所指的文件描述词。

fchdir()与 chdir()完全相同，唯一的区别是目录是以开放文件描述符的形式给出的。

* 返回值

执行成功则返回0，失败返回-1，errno为错误代码。

* 范例

#include<sys/types.h>#include<sys/stat.h>#include<fcntl.h>#include<unistd.h>main(){int fd;fd = open(“/tmp”,O\_RDONLY);fchdir(fd);printf(“current working directory : %s \n”,getcwd(NULL,NULL));close(fd);}

* 执行

current working directory : /tmp

## 13.9 fchmod

​改变文件的权限。

* 相关函数

chmod，stat，open，chown

* 表头文件

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

* 定义函数

int fchmod(int fildes, mode\_t mode);

* 函数说明

fchmod()会依参数mode权限来更改参数fildes所指文件的权限。参数fildes为已打开文件的文件描述词。

fchmod() 系统调用可以更改文件的模式位（文件模式由文件权限位和设置用户 ID、设置组 ID 和粘性位组成）。(文件模式由文件权限位加上设置用户 ID、设置组 ID 和粘性位组成）。

fchmod() 更改打开的文件描述符 fd 所引用文件的模式。

新的文件模式在 mode 中指定，它是由以下零个或多个值组成的位掩码：

* 返回值

权限改变成功则返回0，失败返回-1，错误原因存于errno。

* 错误代码

EBADF 参数fildes为无效的文件描述词。 EPERM 进程的有效用户识别码与欲修改权限的文件所有者不同，而且也不具root权限。 EROFS 欲写入权限的文件存在于只读文件系统内。 EIO I/O 存取错误。

* 范例

#include<sys/stat.h>#include<fcntl.h>main(){int fd;fd = open (“/etc/passwd”,O\_RDONLY);fchmod(fd,S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);close(fd);}

## 13.10 fchown

​改变文件的所有者。

* 相关函数

chown，lchown，chmod

* 表头文件

#include<sys/types.h>#include<unistd.h>

* 定义函数

int fchown(int fd, uid\_t owner, gid\_t group);

* 函数说明

fchown()会将参数fd指定文件的所有者变更为参数owner代表的用户，而将该文件的组变更为参数group组。如果参数owner或group为-1，对映的所有者或组有所改变。参数fd 为已打开的文件描述词。当root用fchown()改变文件所有者或组时，该文件若具S\_ISUID或S\_ISGID权限，则会清除此权限位。

* 返回值

成功则返回0，失败则返回-1，错误原因存于errno。

* 错误代码

EBADF 参数fd文件描述词为无效的或该文件已关闭。 EPERM 进程的有效用户识别码与欲修改权限的文件所有者不同，而且也不具root权限，或是参数owner、group不正确。 EROFS 欲写入的文件存在于只读文件系统内。 ENOENT 指定的文件不存在 EIO I/O存取错误

* 范例

#include<sys/types.h>#include<unistd.h>#include<fcntl.h>main(){int fd;fd = open (“/etc/passwd”,O\_RDONLY);chown(fd,0,0);close(fd);}

## 13.11 fstat

​由文件描述词取得文件状态。

* 相关函数

stat，lstat，chmod，chown，readlink，utime

* 表头文件

#include<sys/stat.h>#include<unistd.h>

* 定义函数

int fstat(int fildes, struct stat \*buf);

* 函数说明

fstat()用来将参数fildes所指的文件状态，复制到参数buf所指的结构中(struct stat)。Fstat()与stat()作用完全相同，不同处在于传入的参数为已打开的文件描述词。详细内容请参考stat()。

* 返回值

执行成功则返回0，失败返回-1，错误代码存于errno。

* 范例

#include<sys/stat.h>#include<unistd.h>#include<fcntk.h>main(){struct stat buf;int fd;fd = open (“/etc/passwd”,O\_RDONLY);fstat(fd,&buf);printf(“/etc/passwd file size +%d\n “,buf.st\_size);}

* 执行

/etc/passwd file size = 705

## 13.12 ftruncate

​改变文件大小

* 相关函数

open，truncate

* 表头文件

#include<unistd.h>

* 定义函数

int ftruncate(int fd, off\_t length);

* 函数说明

truncate() 和 ftruncate() 函数会将以 path 命名或以 fd 引用的常规文件截断到长度为精确字节的大小。

如果文件先前大于这个长度，多余的数据就会丢失。如果文件先前较短，则会被扩展，扩展部分读取为空字节（'\0'）。

文件偏移量不会改变。

如果文件大小改变了，那么文件的 st\_ctime 和 st\_mtime 字段（分别是上次状态改变的时间和上次修改的时间；参见 inode(7)）会被更新，而set-user-ID 和 set-group-ID 模式位可能会被清除。

使用 ftruncate()，文件必须可以写入；使用 truncate()，文件必须可以写入。

* 返回值

执行成功则返回0，失败返回-1，错误原因存于errno。

* 错误代码

EBADF 参数fd文件描述词为无效的或该文件已关闭。 EINVAL 参数fd 为一socket 并非文件，或是该文件并非以写入模式打开。

## 13.13 getcwd

​取得当前的工作目录。

* 相关函数

get\_current\_dir\_name，getwd，chdir

* 表头文件

#include<unistd.h>

* 定义函数

char \*getcwd(char \*buf, size\_t size);

* 函数说明

getcwd()会将当前的工作目录绝对路径复制到参数buf所指的内存空间，参数size为buf的空间大小。在调用此函数时，buf所指的内存空间要足够大，若工作目录绝对路径的字符串长度超过参数size大小，则回值NULL，errno的值则为ERANGE，应用程序应检查此错误，并在必要时分配更大的缓冲区。倘若参数buf为NULL，getcwd()会依参数size的大小自动配置内存(使用malloc())，如果参数size也为0，则getcwd()会依工作目录绝对路径的字符串程度来决定所配置的内存大小，进程可以在使用完此字符串后利用free()来释放此空间。

这些函数返回一个空尾字符串，其中包含一个绝对路径名，即调用进程的当前工作目录。路径名作为函数结果返回，并通过参数 buf（如果存在）返回。

作为 POSIX.1-2001 标准的扩展，如果 buf 为 NULL，glibc 的 getcwd() 会使用 malloc(3) 动态分配缓冲区。在这种情况下，除非 size 为零，否则分配的缓冲区长度为 size，此时 buf 将根据需要分配尽可能大的长度。调用者应 free(3) 返回的缓冲区。

* 返回值

成功后，这些函数会返回一个指向字符串的指针，字符串中包含当前工作目录的路径名。在 getcwd() 和 getwd() 的情况下，这个指针的值与buf 的值相同。

如果函数执行失败，则返回 NULL，并设置 errno 表示错误。出错时，buf 指向的数组内容未定义。

* 范例

#include<unistd.h>main(){char buf[80];getcwd(buf,sizeof(buf));printf(“current working directory : %s\n”,buf);}

* 执行

current working directory :/tmp

## 13.14 link

​建立文件连接。

* 相关函数

symlink，unlink

* 表头文件

#include<unistd.h>

* 定义函数

int link(const char \*oldpath, const char \*newpath);

* 函数说明

link()以参数newpath指定的名称来建立一个新的连接(硬连接)到参数oldpath所指定的已存在文件。如果参数newpath指定的名称为一已存在的文件则不会建立连接。

在任何操作中，这个新名称都可以与旧名称一样使用；两个名称都指向同一个文件（因此具有相同的权限和所有权），无法分辨哪个名称是 “原始名称”。

* 返回值

成功则返回0，失败返回-1，错误原因存于errno。

* 错误代码

EXDEV 参数oldpath与newpath不是建立在同一文件系统。 EPERM 参数oldpath与newpath所指的文件系统不支持硬连接 EROFS 文件存在于只读文件系统内 EFAULT 参数oldpath或newpath 指针超出可存取内存空间。 ENAMETOLLONG 参数oldpath或newpath太长 ENOMEM 核心内存不足 EEXIST 参数newpath所指的文件名已存在。 EMLINK 参数oldpath所指的文件已达最大连接数目。 ELOOP 参数pathname有过多符号连接问题 ENOSPC 文件系统的剩余空间不足。 EIO I/O 存取错误。

* 附加说明

link()所建立的硬连接无法跨越不同文件系统，如果需要请改用symlink()。

* 范例

/\* 建立/etc/passwd 的硬连接为pass \*/#include<unistd.h>main(){link(“/etc/passwd”,”pass”);}

## 13.15 lstat

​由文件描述词取得文件状态。

* 相关函数

stat，fstat，chmod，chown，readlink，utime

* 表头文件

#include<sys/stat.h>#include<unistd.h>

* 定义函数

int lstat (const char \*file\_name, struct stat \*buf);

* 函数说明

lstat()与stat()作用完全相同，都是取得参数file\_name所指的文件状态，其差别在于，当文件为符号连接时，lstat()会返回该link本身的状态。详细内容请参考stat()。

这些函数在 statbuf 指向的缓冲区中返回文件信息。文件本身不需要权限，但对于 stat()、fstatat() 和 lstat()，则需要路径名中指向该文件的所有目录的执行（搜索）权限。

stat() 和 fstatat() 可检索路径名指向的文件的信息；fstatat() 的不同之处如下所述。

lstat() 与 stat() 相同，但如果 pathname 是一个符号链接，则返回链接本身的信息，而不是链接指向的文件。

fstat() 与 stat() 完全相同，只是要获取信息的文件由文件描述符 fd 指定。

* 返回值

执行成功则返回0，失败返回-1，错误代码存于errno。

* 范例

参考stat()。

## 13.16 opendir

​打开目录。

* 相关函数

open，readdir，closedir，rewinddir，seekdir，telldir，scandir

* 表头文件

#include<sys/types.h>#include<dirent.h>

* 定义函数

DIR \*opendir(const char \*name);

* 函数说明

opendir()用来打开参数name指定的目录，并返回DIR\*形态的目录流，和open()类似，接下来对目录的读取和搜索都要使用此返回值。目录流的位置是目录中的第一个条目。

* 返回值

成功则返回DIR\* 型态的目录流，打开失败则返回NULL。

* 错误代码

EACCESS 权限不足 EMFILE 已达到进程可同时打开的文件数上限。 ENFILE 已达到系统可同时打开的文件数上限。 ENOTDIR 参数name非真正的目录 ENOENT 参数name 指定的目录不存在，或是参数name 为一空字符串。 ENOMEM 核心内存不足。

## 13.17 readdir

​读取目录。

* 相关函数

open，opendir，closedir，rewinddir，seekdir，telldir，scandir

* 表头文件

#include<sys/types.h>#include<dirent.h>

* 定义函数

struct dirent \*readdir(DIR \*dir);

* 函数说明

readdir()返回参数dir目录流的下个目录进入点。 结构dirent定义如下struct dirent{ino\_t d\_ino;ff\_t d\_off;signed short int d\_reclen;unsigned char d\_type;har d\_name[256;};d\_ino 此目录进入点的inode d\_off 目录文件开头至此目录进入点的位移 d\_reclen \_name的长度，不包含NULL字符 d\_type d\_name 所指的文件类型 d\_name 文件名

readdir() 函数返回一个指向 dirent 结构的指针，该结构代表 dir指向的目录流中的下一个目录条目。如果到达目录流的末尾或发生错误，则返回 NULL。

* 返回值

成功后，readdir() 会返回一个指向 dirent 结构的指针。(该结构可能是静态分配的，请不要尝试释放（3）它）。

如果到达目录流的末端，则返回 NULL，errno 不会改变。如果发生错误，则返回 NULL，并设置 errno 以指示错误。要区分流结束和错误，可在调用 readdir() 之前将 errno 设为零，然后在返回 NULL 时检查 errno 的值。

* 附加说明

EBADF参数dir为无效的目录流。

* 范例

#include<sys/types.h>#include<dirent.h>#include<unistd.h>main(){DIR \* dir;struct dirent \* ptr;int i;dir =opendir(“/etc/rc.d”);while((ptr = readdir(dir))!=NULL){printf(“d\_name: %s\n”,ptr->d\_name);}closedir(dir);}

* 执行

d\_name:.d\_name:..d\_name:init.dd\_name:rc0.dd\_name:rc1.dd\_name:rc2.dd\_name:rc3.dd\_name:rc4.dd\_name:rc5.dd\_name:rc6.dd\_name:rcd\_name:rc.locald\_name:rc.sysinit

## 13.18 readlink

​取得符号连接所指的文件。

* 相关函数

stat，lstat，symlink

* 表头文件

#include<unistd.h>

* 定义函数

int readlink(const char \*path, char \*buf, size\_t bufsiz);

* 函数说明

readlink()会将参数path的符号连接内容存到参数buf所指的内存空间，返回的内容不是以NULL作字符串结尾，但会将字符串的字符数返回。若参数bufsiz小于符号连接的内容长度，过长的内容会被截断。

* 返回值

调用成功后，将返回 buf 中的字节数。(如果返回值等于 bufsiz，则可能发生了截断。）如果出错，则返回 -1，并设置 errno 表示出错。

* 错误代码

EACCESS 取文件时被拒绝，权限不够 EINVAL 参数bufsiz 为负数 EIO I/O 存取错误。 ELOOP 欲打开的文件有过多符号连接问题。 ENAMETOOLONG 参数path的路径名称太长 ENOENT 参数path所指定的文件不存在 ENOMEM 核心内存不足 ENOTDIR 参数path路径中的目录存在但却非真正的目录。

## 13.19 remove

​删除文件。

* 相关函数

link，rename，unlink

* 表头文件

#include<stdio.h>

* 定义函数

int remove(const char \*pathname);

* 函数说明

remove()会删除参数pathname指定的文件。

remove() 会从文件系统中删除一个名称。文件会调用 unlink(2)，目录会调用 rmdir(2)。

如果被删除的名称是文件的最后一个链接，并且没有进程打开该文件，那么该文件将被删除，其占用的空间将被重新使用。

如果该名称是指向文件的最后一个链接，但仍有进程打开该文件，则该文件将继续存在，直到指向它的最后一个文件描述符关闭为止。

如果名称指向一个符号链接，该链接将被删除。

如果名称指向套接字、FIFO 或设备，则名称会被删除，但打开对象的进程仍可继续使用该对象。

* 返回值

成功则返回0，失败则返回-1，错误原因存于errno。

* 错误代码

EROFS 欲写入的文件存在于只读文件系统内 EFAULT 参数pathname指针超出可存取内存空间 ENAMETOOLONG 参数pathname太长 ENOMEM 核心内存不足 ELOOP 参数pathname有过多符号连接问题 EIO I/O 存取错误。

## 13.20 rename

​更改文件名称或位置。

* 相关函数

link，unlink，symlink

* 表头文件

#include<stdio.h>

* 定义函数

int rename(const char \*oldpath, const char \*newpath);

* 函数说明

rename()会将参数oldpath 所指定的文件名称改为参数newpath所指的文件名称。若newpath所指定的文件已存在，则会被删除。

rename() 重命名文件，并根据需要在目录之间移动文件。文件的任何其他硬链接（使用 link(2) 创建的链接）都不会受到影响。oldpath 的打开文件描述符也不受影响。

重命名操作是否成功取决于各种限制条件：请参阅下面的错误。

如果 newpath 已经存在，它将被原子替换，这样其他试图访问 newpath 的进程就不会发现它丢失了。不过，可能会有一个窗口，其中 oldpath 和 newpath 都指向正在重命名的文件。

如果 oldpath 和 newpath 是指向同一文件的现有硬链接，那么 rename() 不会做任何操作，而是返回成功状态。

如果 newpath 已经存在，但由于某种原因操作失败，rename( ) 会保证在原处保留 newpath 的实例。

oldpath 可以指定一个目录。在这种情况下，newpath 要么不存在，要么指定一个空目录。

如果 oldpath 指向一个符号链接，该链接将被重命名；如果 newpath 指向一个符号链接，该链接将被覆盖。

* 返回值

执行成功则返回0，失败返回-1，错误原因存于errno

* 范例

/\* 设计一个DOS下的rename指令rename 旧文件名新文件名\*/#include <stdio.h>void main(int argc,char \*\*argv){if(argc<3){printf(“Usage: %s old\_name new\_name\n”,argv[0]);return;}printf(“%s=>%s”,argc[1],argv[2]);if(rename(argv[1],argv[2]<0)printf(“error!\n”);elseprintf(“ok!\n”);}

## 13.21 rewinddir

​重设读取目录的位置为开头位置。

* 相关函数

open，opendir，closedir，telldir，seekdir，readdir，scandir

* 表头文件

#include<sys/types.h>#include<dirent.h>

* 定义函数

void rewinddir(DIR \*dir);

* 函数说明

rewinddir()用来设置参数dir 目录流目前的读取位置为原来开头的读取位置。

* 错误代码

EBADF dir为无效的目录流

* 附加说明

使用access()作用户认证方面的判断要特别小心，例如在access()后再做open()的空文件可能会造成系统安全上的问题。

* 范例

#include<sys/types.h>#include<dirent.h>#include<unistd.h>main(){DIR \* dir;struct dirent \*ptr;dir = opendir(“/etc/rc.d”);while((ptr = readdir(dir))!=NULL){printf(“d\_name :%s\n”,ptr->d\_name);}rewinddir(dir);printf(“readdir again!\n”);while((ptr = readdir(dir))!=NULL){printf(“d\_name: %s\n”,ptr->d\_name);}closedir(dir);}

* 执行

d\_name:.d\_name:..d\_name:init.dd\_name:rc0.dd\_name:rc1.dd\_name:rc2.dd\_name:rc3.dd\_name:rc4.dd\_name:rc5.dd\_name:rc6.dd\_name:rcd\_name:rc.locald\_name:rc.sysinitreaddir again!d\_name:.d\_name:..d\_name:init.dd\_name:rc0.dd\_name:rc1.dd\_name:rc2.dd\_name:rc3.dd\_name:rc4.dd\_name:rc5.dd\_name:rc6.dd\_name:rcd\_name:rc.locald\_name:rc.sysinit

## 13.22 seekdir

​设置下回读取目录的位置。

* 相关函数

open，opendir，closedir，rewinddir，telldir，readdir，scandir

* 表头文件

#include<dirent.h>

* 定义函数

void seekdir(DIR \*dir, off\_t offset);

* 函数说明

seekdir()用来设置参数dir目录流目前的读取位置，在调用readdir()时便从此新位置开始读取。参数offset 代表距离目录文件开头的偏移量。

* 错误代码

EBADF 参数dir为无效的目录流

* 附加说明

使用access()作用户认证方面的判断要特别小心，例如在access()后再做open()的空文件可能会造成系统安全上的问题。

* 范例

#include<sys/types.h>#include<dirent.h>#include<unistd.h>main(){DIR \* dir;struct dirent \* ptr;int offset,offset\_5,i=0;dir=opendir(“/etc/rc.d”);while((ptr = readdir(dir))!=NULL){offset = telldir(dir);if(++i = =5) offset\_5 =offset;printf(“d\_name :%s offset :%d \n”,ptr->d\_name,offset);}seekdir(dir offset\_5);printf(“Readdir again!\n”);while((ptr = readdir(dir))!=NULL){offset = telldir(dir);printf(“d\_name :%s offset :%d\n”,ptr->d\_name.offset);}closedir(dir);}

* 执行

d\_name : . offset :12d\_name : .. offset:24d\_name : init.d offset 40d\_name : rc0.d offset :56d\_name :rc1.d offset :72d\_name:rc2.d offset :88d\_name:rc3.d offset 104d\_name:rc4.d offset:120d\_name:rc5.d offset:136d\_name:rc6.d offset:152d\_name:rc offset 164d\_name:rc.local offset :180d\_name:rc.sysinit offset :4096readdir again!d\_name:rc2.d offset :88d\_name:rc3.d offset 104d\_name:rc4.d offset:120d\_name:rc5.d offset:136d\_name:rc6.d offset:152d\_name:rc offset 164d\_name:rc.local offset :180d\_name:rc.sysinit offset :4096

## 13.23 stat

​取得文件状态。

* 相关函数

fstat，lstat，chmod，chown，readlink，utime

* 表头文件

#include<sys/stat.h>#include<unistd.h>

* 定义函数

int stat(const char \*file\_name, struct stat \*buf);

* 函数说明

stat()用来将参数file\_name所指的文件状态，复制到参数buf所指的结构中。 下面是struct stat内各参数的说明struct stat{dev\_t st\_dev; /\*device\*/ino\_t st\_ino; /\*inode\*/mode\_t st\_mode; /\*protection\*/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 \*/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 lastaccess\*/time\_t st\_mtime; /\* time of last modification \*/time\_t st\_ctime; /\* time of last change \*/};st\_dev 文件的设备编号

st\_ino 文件的i-node

st\_mode 文件的类型和存取的权限

st\_nlink 连到该文件的硬连接数目，刚建立的文件值为1。

st\_uid 文件所有者的用户识别码

st\_gid 文件所有者的组识别码

st\_rdev 若此文件为装置设备文件，则为其设备编号

st\_size 文件大小，以字节计算

st\_blksize 文件系统的I/O 缓冲区大小。

st\_blcoks 占用文件区块的个数，每一区块大小为512 个字节。

st\_atime 文件最近一次被存取或被执行的时间，一般只有在用mknod、utime、read、write与tructate时改变。 st\_mtime 文件最后一次被修改的时间，一般只有在用mknod、utime和write时才会改变

st\_ctime i-node最近一次被更改的时间，此参数会在文件所有者、组、权限被更改时更新先前所描述的

st\_mode 则定义了下列数种情况

S\_IFMT 0170000 文件类型的位遮罩

S\_IFSOCK 0140000 scoket

S\_IFLNK 0120000 符号连接

S\_IFREG 0100000 一般文件

S\_IFBLK 0060000 区块装置

S\_IFDIR 0040000 目录

S\_IFCHR 0020000 字符装置

S\_IFIFO 0010000 先进先出

S\_ISUID 04000 文件的（set user-id on execution）位

S\_ISGID 02000 文件的（set group-id on execution）位

S\_ISVTX 01000 文件的sticky位

S\_IRUSR（S\_IREAD） 00400 文件所有者具可读取权限

S\_IWUSR（S\_IWRITE）00200 文件所有者具可写入权限

S\_IXUSR（S\_IEXEC） 00100 文件所有者具可执行权限

S\_IRGRP 00040 用户组具可读取权限

S\_IWGRP 00020 用户组具可写入权限

S\_IXGRP 00010 用户组具可执行权限

S\_IROTH 00004 其他用户具可读取权限

S\_IWOTH 00002 其他用户具可写入权限

S\_IXOTH 00001 其他用户具可执行权限

上述的文件类型在POSIX 中定义了检查这些类型的宏定义 S\_ISLNK （st\_mode） 判断是否为符号连接 S\_ISREG （st\_mode） 是否为一般文件 S\_ISDIR （st\_mode）是否为目录 S\_ISCHR （st\_mode）是否为字符装置文件 S\_ISBLK （s3e） 是否为先进先出 S\_ISSOCK （st\_mode） 是否为socket 若一目录具有sticky 位（S\_ISVTX），则表示在此目录下的文件只能被该文件所有者、此目录所有者或root来删除或改名。

* 返回值

执行成功则返回0，失败返回-1，错误代码存于errno

* 错误代码

ENOENT 参数file\_name指定的文件不存在 ENOTDIR 路径中的目录存在但却非真正的目录 ELOOP 欲打开的文件有过多符号连接问题，上限为16符号连接 EFAULT 参数buf为无效指针，指向无法存在的内存空间 EACCESS 存取文件时被拒绝 ENOMEM 核心内存不足 ENAMETOOLONG 参数file\_name的路径名称太长

* 范例

#include<sys/stat.h>#include<unistd.h>mian(){struct stat buf;stat (“/etc/passwd”,&buf);printf(“/etc/passwd file size = %d \n”,buf.st\_size);}

* 执行

/etc/passwd file size = 705

## 13.24 symlink

​建立文件符号连接

* 相关函数

link，unlink

* 表头文件

#include<unistd.h>

* 定义函数

int symlink(const char \*oldpath, const char \*newpath);

* 函数说明

symlink()以参数newpath指定的名称来建立一个新的连接(符号连接)到参数oldpath所指定的已存在文件。参数oldpath指定的文件不一定要存在，如果参数newpath指定的名称为一已存在的文件则不会建立连接。

符号链接在运行时被解释为链接内容已被替换到查找文件或目录的路径中。

符号链接可能包含...路径组件，这些组件（如果在链接开始时使用）指向链接所在目录的父目录。

符号链接（也称为软链接）可能指向一个现有文件，也可能指向一个不存在的文件；后一种情况称为悬挂链接。

符号链接的权限无关紧要；在跟踪链接时，其所有权会被忽略（启用了 protected\_symlinks 功能的情况除外，详见 proc(5) 中的说明），但在请求删除或重命名链接，且链接位于设置了粘性位 (S\_ISVTX) 的目录中时，所有权会被检查。

如果 linkpath 已存在，则不会被覆盖。

* 返回值

成功则返回0，失败返回-1，错误原因存于errno。

* 错误代码

EPERM 参数oldpath与newpath所指的文件系统不支持符号连接 EROFS 欲测试写入权限的文件存在于只读文件系统内 EFAULT 参数oldpath或newpath指针超出可存取内存空间。 ENAMETOOLONG 参数oldpath或newpath太长 ENOMEM 核心内存不足 EEXIST 参数newpath所指的文件名已存在。 EMLINK 参数oldpath所指的文件已达到最大连接数目 ELOOP 参数pathname有过多符号连接问题 ENOSPC 文件系统的剩余空间不足 EIO I/O 存取错误

* 范例

#include<unistd.h>main(){symlink(“/etc/passwd”,”pass”);}

## 13.25 telldir

​取得目录流的读取位置。

* 相关函数

open，opendir，closedir，rewinddir，seekdir，readdir，scandir

* 表头文件

#include<dirent.h>

* 定义函数

off\_t telldir(DIR \*dir)

* 函数说明

telldir()返回参数dir目录流目前的读取位置。此返回值代表距离目录文件开头的偏移量返回值返回下个读取位置，有错误发生时返回-1。

* 错误代码

EBADF参数dir为无效的目录流。

* 范例

#include<sys/types.h>#include<dirent.h>#include<unistd.h>main(){DIR \*dir;struct dirent \*ptr;int offset;dir = opendir(“/etc/rc.d”);while((ptr = readdir(dir))!=NULL){offset = telldir (dir);printf(“d\_name : %s offset :%d\n”, ptr->d\_name,offset);}closedir(dir);}

* 执行

d\_name : . offset :12d\_name : .. offset:24d\_name : init.d offset 40d\_name : rc0.d offset :56d\_name :rc1.d offset :72d\_name:rc2.d offset :88d\_name:rc3.d offset 104d\_name:rc4.d offset:120d\_name:rc5.d offset:136d\_name:rc6.d offset:152d\_name:rc offset 164d\_name:rc.local offset :180d\_name:rc.sysinit offset :4096

## 13.26 truncate

​改变文件大小。

* 相关函数

open，ftruncate

* 表头文件

#include<unistd.h>

* 定义函数

int truncate(const char \*path, off\_t length);

* 函数说明

truncate()函数会将以 path 命名或以 fd 引用的常规文件截断到长度为精确字节的大小。

如果文件先前大于这个长度，多余的数据就会丢失。如果文件先前较短，则会被扩展，扩展部分读取为空字节（'\0'）。

文件偏移量不会改变。

如果文件大小改变了，那么文件的 st\_ctime 和 st\_mtime 字段（分别是上次状态改变的时间和上次修改的时间；参见 inode(7)）会被更新，set-user-ID 和 set-group-ID 模式位可能会被清除。

使用 truncate()，文件必须可以写入。

* 返回值

执行成功则返回0，失败返回-1，错误原因存于errno。

* 错误代码

EACCESS 参数path所指定的文件无法存取。 EROFS 欲写入的文件存在于只读文件系统内 EFAULT 参数path指针超出可存取内存空间 EINVAL 参数path包含不合法字符 ENAMETOOLONG 参数path太长 ENOTDIR 参数path路径并非一目录 EISDIR 参数path 指向一目录 ETXTBUSY 参数path所指的文件为共享程序，而且正被执行中 ELOOP 参数path’有过多符号连接问题 EIO I/O 存取错误。

## 13.27 umask

​设置建立新文件时的权限遮罩。

* 相关函数

creat，open

* 表头文件

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

* 定义函数

mode\_t umask(mode\_t mask);

* 函数说明

umask() 将调用进程的文件模式创建掩码（umask）设置为 mask & 0777（即只使用掩码中的文件权限位），并返回掩码的前一个值。

umask 被 open(2)、mkdir(2) 和其他创建文件的系统调用用于修改新创建文件或目录的权限。具体来说，open(2) 和 mkdir(2) 的模式参数会关闭 umask 中的权限。在使用open()建立新文件时，该参数mode并非真正建立文件的权限，而是(mode&~umask)的权限值。例如，在建立文件时指定文件权限为0666，通常umask值默认为022，则该文件的真正权限则为0666&～022＝0644，也就是rw-r--r--返回值此调用不会有错误值返回。返回值为原先系统的umask值。

另外，如果父目录有默认的 ACL（参见 acl(5)），则忽略 umask，继承默认的 ACL，根据继承的 ACL 设置权限位，并关闭模式参数中不存在的权限位。例如，以下默认 ACL 相当于 022:u::rwx,g::r-x,o::r-x 的 umask

将此默认 ACL 的效果与 0666（rw-rw-rw-）模式参数相结合，文件权限将变为 0644（rw-r--r--）。

用于指定掩码的常量在 inode(7) 中有描述。

进程 umask 的典型默认值是 S\_IWGRP | S\_IWOTH（八进制 022）。

## 13.28 unlink

​删除文件。

* 相关函数

link，rename，remove

* 表头文件

#include<unistd.h>

* 定义函数

int unlink(const char \*pathname);

* 函数说明

unlink()会删除参数pathname指定的文件。如果该pathname是文件的最后一个链接，且没有进程打开该文件，则该文件将被删除，其使用的空间将被重新使用。

如果该名称是指向文件的最后一个链接，但仍有进程打开该文件，则该文件将继续存在，直到指向它的最后一个文件描述符关闭为止。

如果名称指向一个符号链接，该链接将被删除。

如果名称指向套接字、FIFO 或设备，则其名称将被删除，但打开该对象的进程仍可继续使用它。

* 返回值

成功则返回0，失败返回-1，错误原因存于errno

* 错误代码

EROFS 文件存在于只读文件系统内 EFAULT 参数pathname指针超出可存取内存空间 ENAMETOOLONG 参数pathname太长 ENOMEM 核心内存不足 ELOOP 参数pathname 有过多符号连接问题 EIO I/O 存取错误

## 13.29 utime

​修改文件的存取时间和更改时间。

* 相关函数

utimes，stat

* 表头文件

#include<sys/types.h>#include<utime.h>

* 定义函数

int utime(const char \*filename, struct utimbuf \*buf);

* 函数说明

utime() 系统调用会将文件名指定的 inode 的访问和修改时间分别改为时间的 actime 和 modtime 字段。状态更改时间 (ctime) 将被设置为当前时间，即使其他时间戳没有实际更改。

如果 times 为空，文件的访问时间和修改时间将被设置为当前时间。

允许更改时间戳的条件是：进程拥有适当的权限，或有效用户 ID 等于文件的用户 ID，或 times 为空且进程拥有文件的写入权限。

utimbuf 结构如下

struct utimbuf {

time\_t actime; /\* 访问时间 \*/

time\_t modtime; /\* 修改时间 \*/

};

utime() 系统调用允许指定分辨率为 1 秒的时间戳

* 返回值

如果参数buf为空指针(NULL)，则该文件的存取时间和更改时间全部会设为目前时间。 执行成功则返回0，失败返回-1，错误代码存于errno。

* 错误代码

EACCESS 存取文件时被拒绝，权限不足 ENOENT 指定的文件不存在。

## 13.30 utimes

​修改文件的存取时间和更改时间。

* 相关函数

utime，stat

* 表头文件

#include<sys/types.h>#include<utime.h>

* 定义函数

int utimes(char \*filename, struct timeval \*tvp);

* 函数说明

utimes() 系统调用与utime()类似，但 times 参数指向的是一个数组而非结构体。该数组的元素是 timeval 结构，允许以 1 微秒的精度指定时间戳。timeval 结构如下

struct timeval {

long tv\_sec; /\* 秒 \*/

long tv\_usec; /\* 微秒 \*/

};

times[0] 指定新的访问时间，times[1] 指定新的修改时间。如果 times 为 NULL，则与 utime() 类似，文件的访问时间和修改时间都将设置为当前时间。

* 返回值

参数tvp 指向两个timeval 结构空间，和utime（）使用的utimebuf结构比较，tvp[0].tc\_sec 则为utimbuf.actime，tvp]1].tv\_sec 为utimbuf.modtime。 执行成功则返回0。失败返回-1，错误代码存于errno。

* 错误代码

EACCESS 存取文件时被拒绝，权限不足 ENOENT 指定的文件不存在

# 第十四章 信号处理篇

## 14.1 alarm

​设置信号传送闹钟。

* 相关函数

signal，sleep

* 表头文件

#include<unistd.h>

* 定义函数

unsigned int alarm(unsigned int seconds);

* 函数说明

alarm()用来设置信号SIGALRM在经过参数seconds指定的秒数后传送给目前的进程。如果参数seconds 为0，则之前设置的闹钟会被取消，并将剩下的时间返回。

alarm() 会以秒为单位向调用进程发送 SIGALRM 信号。

如果秒数为零，则取消任何待处理的警报。

在任何情况下，先前设置的 alarm() 都会被取消。

* 返回值

返回之前闹钟的剩余秒数，如果之前未设闹钟则返回0。

* 范例

#include<unistd.h>#include<signal.h>void handler() {printf(“hello\n”);}main(){int i;signal(SIGALRM,handler);alarm(5);for(i=1;i<7;i++){printf(“sleep %d ...\n”,i);sleep(1);}}

* 执行

sleep 1 ...sleep 2 ...sleep 3 ...sleep 4 ...sleep 5 ...hellosleep 6 ...

## 14.2 kill

​传送信号给指定的进程。

* 相关函数

raise，signal

* 表头文件

#include<sys/types.h>#include<signal.h>

* 定义函数

int kill(pid\_t pid, int sig);

* 函数说明

kill() 系统调用可用于向任何进程组或进程发送任何信号。

如果 pid 为正数，那么信号 sig 将发送给具有 pid 指定 ID 的进程。

如果 pid 为 0，则向调用进程的进程组中的每个进程发送信号 sig。

如果 pid 等于-1，那么除了进程 1（init）外，sig 会发送给调用进程有权限发送信号的每个进程，但请参阅下文。

如果 pid 小于-1，则向进程组中 ID 为-pid 的每个进程发送 sig。

如果 sig 为 0，则不会发送信号，但仍会进行存在和权限检查；这可用于检查是否存在允许调用者发送信号的进程 ID 或进程组 ID。

要使一个进程有权限发送信号，它必须具有特权（在 Linux 下：在目标进程的用户名空间中具有 CAP\_KILL 功能），或者发送进程的真实或有效用户 ID 必须等于目标进程的真实或保存的设置用户 ID。就 SIGCONT 而言，发送进程和接收进程属于同一会话就足够了。

* 返回值

执行成功则返回0，如果有错误则返回-1。

* 错误代码

EINVAL 参数sig 不合法 ESRCH 参数pid 所指定的进程或进程组不存在 EPERM 权限不够无法传送信号给指定进程

* 范例

#include<unistd.h>#include<signal.h>#include<sys/types.h>#include<sys/wait.h>main(){pid\_t pid;int status;if(!(pid= fork())){printf(“Hi I am child process!\n”);sleep(10);return;}else{printf(“send signal to child process (%d) \n”,pid);sleep(1);kill(pid ,SIGABRT);wait(&status);if(WIFSIGNALED(status))printf(“chile process receive signal %d\n”,WTERMSIG(status));}}

* 执行

sen signal to child process(3170)Hi I am child process!child process receive signal 6

## 14.3 pause

​让进程暂停直到信号出现。

* 相关函数

kill，signal，sleep

* 表头文件

#include<unistd.h>

* 定义函数

int pause(void);

* 函数说明

pause()会令目前的进程暂停（进入睡眠状态），直到被信号(signal)所中断。

* 返回值

只有当信号被捕获且信号捕获函数返回时，pause() 才会返回。在这种情况下，pause() 返回-1，errno 设置为 EINTR

* 错误代码

EINTR 有信号到达中断了此函数。

## 14.4 sigaction

​查询或设置信号处理方式。

* 相关函数

signal，sigprocmask，sigpending，sigsuspend

* 表头文件

#include<signal.h>

* 定义函数

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

* 函数说明

sigaction()会依参数signum指定的信号编号来设置该信号的处理函数。参数signum可以指定SIGKILL和SIGSTOP以外的所有信号。 如参数结构sigaction定义如下struct sigaction{void (\*sa\_handler) (int);sigset\_t sa\_mask;int sa\_flags;void (\*sa\_restorer) (void);}sa\_handler此参数和signal()的参数handler相同，代表新的信号处理函数，其他意义请参考signal()。

sa\_mask 用来设置在处理该信号时暂时将sa\_mask 指定的信号搁置。

sa\_restorer 此参数没有使用。

sa\_flags 用来设置信号处理的其他相关操作，下列的数值可用。

OR 运算（|）组合 A\_NOCLDSTOP :

如果参数signum为SIGCHLD，则当子进程暂停时并不会通知父进程

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的方式。

SA\_RESTART:被信号中断的系统调用会自行重启

SA\_NOMASK/SA\_NODEFER:在处理此信号未结束前不理会此信号的再次到来。

如果参数oldact不是NULL指针，则原来的信号处理方式会由此结构sigaction 返回。

* 返回值

执行成功则返回0，如果有错误则返回-1。

* 错误代码

EINVAL 参数signum 不合法， 或是企图拦截SIGKILL/SIGSTOPSIGKILL信号 EFAULT 参数act，oldact指针地址无法存取。 EINTR 此调用被中断

* 范例

#include<unistd.h>#include<signal.h>void show\_handler(struct sigaction \* act){switch (act->sa\_flags){case SIG\_DFL:printf(“Default action\n”);break;case SIG\_IGN:printf(“Ignore the signal\n”);break;default: printf(“0x%x\n”,act->sa\_handler);}}main(){int i;struct sigaction act,oldact;act.sa\_handler = show\_handler;act.sa\_flags = SA\_ONESHOT|SA\_NOMASK;sigaction(SIGUSR1,&act,&oldact);for(i=5;i<15;i++){printf(“sa\_handler of signal %2d =”.i);sigaction(i,NULL,&oldact);}}

* 执行

sa\_handler of signal 5 = Default actionsa\_handler of signal 6= Default actionsa\_handler of signal 7 = Default actionsa\_handler of signal 8 = Default actionsa\_handler of signal 9 = Default actionsa\_handler of signal 10 = 0x8048400sa\_handler of signal 11 = Default actionsa\_handler of signal 12 = Default actionsa\_handler of signal 13 = Default actionsa\_handler of signal 14 = Default action

## 14.5 sigaddset

​增加一个信号至信号集。

* 相关函数

sigemptyset，sigfillset，sigdelset，sigismember

* 表头文件

#include<signal.h>

* 定义函数

int sigaddset(sigset\_t \*set, int signum);

* 函数说明

sigaddset()用来将参数signum 代表的信号加入至参数set 信号集里。

* 返回值

执行成功则返回0，如果有错误则返回-1。

* 错误代码

EFAULT 参数set指针地址无法存取 EINVAL 参数signum非合法的信号编号

## 14.6 sigdelset

​从信号集里删除一个信号。

* 相关函数

sigemptyset，sigfillset，sigaddset，sigismember

* 表头文件

#include<signal.h>

* 定义函数

int sigdelset(sigset\_t \*set, int signum);

* 函数说明

sigdelset()用来将参数signum代表的信号从参数set信号集里删除。

* 返回值

执行成功则返回0，如果有错误则返回-1。

* 错误代码

EFAULT 参数set指针地址无法存取 EINVAL 参数signum非合法的信号编号

## 14.7 sigemptyset

​初始化信号集。

* 相关函数

sigaddset，sigfillset，sigdelset，sigismember

* 表头文件

#include<signal.h>

* 定义函数

int sigemptyset(sigset\_t \*set);

* 函数说明

sigemptyset()用来将参数set信号集初始化并清空。

* 返回值

执行成功则返回0，如果有错误则返回-1。

* 错误代码

EFAULT 参数set指针地址无法存取

## 14.8 sigfillset

​将所有信号加入至信号集。

* 相关函数

sigempty，sigaddset，sigdelset，sigismember

* 表头文件

#include<signal.h>

* 定义函数

int sigfillset(sigset\_t \*set);

* 函数说明

sigfillset()用来将参数set信号集初始化，然后把所有的信号加入到此信号集里。

* 返回值

执行成功则返回0，如果有错误则返回-1。

* 附加说明

EFAULT 参数set指针地址无法存取

## 14.9 sigismember

​测试某个信号是否已加入至信号集里。

* 相关函数

sigemptyset，sigfillset，sigaddset，sigdelset

* 表头文件

#include<signal.h>

* 定义函数

int sigismember(const sigset\_t \*set, int signum);

* 函数说明

sigismember()用来测试参数signum 代表的信号是否已加入至参数set信号集里。如果信号集里已有该信号则返回1，否则返回0。

* 返回值

信号集已有该信号则返回1，没有则返回0。如果有错误则返回-1。

* 错误代码

EFAULT 参数set指针地址无法存取 EINVAL 参数signum 非合法的信号编号

## 14.10 signal

​设置信号处理方式。

* 相关函数

sigaction，kill，raise

* 表头文件

#include<signal.h>

* 定义函数

void (\*signal(int signum, void (\*handler)(int)))(int);或者

typedef void (\*sig\_t)(int);sig\_t signal(int signum, sig\_t handler);

* 函数说明

signal()会依参数signum 指定的信号编号来设置该信号的处理函数。当指定的信号到达时就会跳转到参数handler指定的函数执行。

警告：在不同的 Linux 版本中，它的行为也会有所不同。请避免使用：改用 sigaction(2)。

请参阅下面的可移植性。

signal()设置处理程序对信号符号的处置，即 SIG\_IGN、 SIG\_DFL 或程序员定义的函数地址（“信号处理程序”）。

如果信号音被发送到进程，则会发生以下情况之一：

\* 如果处置设置为 SIG\_IGN，则信号被忽略。

\* 如果处置设置为 SIG\_DFL，则会执行与信号相关的默认操作（参见 signal(7)）。

\* 如果处置设置为函数，则首先将处置重置为 SIG\_DFL，或者阻塞信号（参见下面的可移植性），然后调用带有参数 signum 的处理程序。如果调用处理程序导致信号被阻塞，那么从处理程序返回后信号将被解除阻塞。

SIGKILL 和 SIGSTOP 信号不能被捕获或忽略。

* 返回值

返回先前的信号处理函数指针，如果有错误则返回SIG\_ERR(-1),并设置 errno 表示错误。

* 附加说明

在信号发生跳转到自定的handler处理函数执行后，系统会自动将此处理函数换回原来系统预设的处理方式，如果要改变此操作请改用sigaction()。

* 范例

参考alarm()或raise()。

## 14.11 sigpending

​查询被搁置的信号。

* 相关函数

signal，sigaction，sigprocmask，sigsuspend

* 表头文件

#include<signal.h>

* 定义函数

int sigpending(sigset\_t \*set);

* 函数说明

sigpending()会将被搁置的信号集合由参数set指针返回。

* 返回值

执行成功则返回0，如果有错误则返回-1

* 错误代码

EFAULT 参数set指针地址无法存取 EINTR 此调用被中断。

## 14.12 sigprocmask

​查询或设置信号遮罩。

* 相关函数

signal，sigaction，sigpending，sigsuspend

* 表头文件

#include<signal.h>

* 定义函数

int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset);

* 函数说明

sigprocmask()可以用来改变目前的信号遮罩，其操作依参数how来决定

SIG\_BLOCK 新的信号遮罩由目前的信号遮罩和参数set 指定的信号遮罩作联集

SIG\_UNBLOCK 将目前的信号遮罩删除掉参数set指定的信号遮罩

SIG\_SETMASK 将目前的信号遮罩设成参数set指定的信号遮罩。

如果参数oldset不是NULL指针，那么目前的信号遮罩会由此指针返回。

sigprocmask() 用于获取和/或更改调用线程的信号屏蔽。信号屏蔽

* 返回值

sigprocmask() 成功时返回 0。如果失败，则返回 -1 并设置 errno 以指示错误。

* 错误代码

EFAULT 参数set，oldset指针地址无法存取。 EINTR 此调用被中断

## 14.13 sleep

​让进程暂停执行一段时间。

* 相关函数

signal，alarm

* 表头文件

#include<unistd.h>

* 定义函数

unsigned int sleep(unsigned int seconds);

* 函数说明

sleep()会令目前的进程暂停，直到达到参数seconds 所指定的时间，或是被信号所中断。

* 返回值

若进程暂停到参数seconds 所指定的时间则返回0，若有信号中断则返回剩余秒数。

## 14.14 ferror

​检查文件流是否有错误发生。

* 相关函数

clearerr，perror

* 表头文件

#include<stdio.h>

* 定义函数

int ferror(FILE \*stream);

* 函数说明

函数 ferror() 测试 stream 所指向数据流的错误指示符，如果设置了错误指示符，则返回非零值。错误指示器只能通过函数 clearerr() 重置。

* 返回值

如果文件流有错误发生则返回非0值。

## 14.15 perror

​打印出错误原因信息字符串

* 相关函数

Strerror

* 表头文件

#include<stdio.h>

* 定义函数

void perror(const char \*s);

* 函数说明

perror() 函数在标准错误中生成一条信息，描述在调用系统或库函数时遇到的最后一个错误。

首先（如果 s 不是 NULL 且 \*s 不是空字节（'\0'）），打印参数字符串 s，然后是冒号和空格。然后是与 errno 当前值相对应的错误信息和换行符。

为发挥最大作用，参数字符串应包括导致错误的函数名称。

全局错误列表 sys\_errlist[]（可按 errno 索引）可用于获取不带换行的错误信息。表中提供的最大信息编号是 sys\_nerr-1。直接访问该列表时要小心，因为新的错误值可能尚未添加到 sys\_errlist[]。如今，sys\_errlist[] 已被弃用；请使用 strerror(3) 代替。

当系统调用失败时，通常会返回-1，并将变量 errno 设置为说明出错原因的值。(许多库函数也会这样做。函数 perror() 的作用是将错误代码转换为人类可读的形式。请注意，在系统调用或库函数调用成功后，errno 是未定义的：这次调用即使成功，也可能会改变这个变量，例如因为它在内部使用了其他库函数而失败了。因此，如果调用失败后没有立即调用 perror()，则应保存 errno 的值。

* 范例

#include<stdio.h>main(){FILE \*fp;fp = fopen(“/tmp/noexist”,”r+”);if(fp = =NULL) perror(“fopen”);}

* 执行

$ ./perrorfopen : No such file or directory

## 14.16 strerror

​返回错误原因的描述字符串。

* 相关函数

Perror

* 表头文件

#include<string.h>

* 定义函数

char \*strerror(int errnum);

* 函数说明

strerror() 函数返回一个指向字符串的指针，该字符串描述了参数 errnum 传递的错误代码，可能使用当前本地语言的 LC\_MESSAGES 部分来选择适当的语言。(例如，如果 errnum 是 EINVAL，返回的描述将是 “无效参数”）。应用程序不得修改该字符串，在后续调用 strerror() 或 strerror\_l()，或获取该字符串的线程退出时，返回的指针将失效。包括 perror(3) 在内的任何其他库函数都不会修改该字符串。

* 返回值

返回描述错误原因的字符串指针。

* 范例

/\* 显示错误代码0 至9 的错误原因描述\*/#include<string.h>main(){int i;for(i=0;i<10;i++)printf(“%d : %s\n”,i,strerror(i));}

* 执行

0 : Success1 : Operation not permitted2 : No such file or directory3 : No such process4 : Interrupted system call5 : Input/output error6 : Device not configured7 : Argument list too long8 : Exec format error9 : Bad file descriptor

## 14.17 mkfifo

​建立具名管道。

* 相关函数

pipe，popen，open，umask

* 表头文件

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

* 定义函数

int mkfifo(const char \*pathname, mode\_t mode);

* 函数说明

mkfifo()会依参数pathname建立特殊的FIFO文件，该文件必须不存在，而参数mode为该文件的权限（mode&~umask），因此umask值也会影响到FIFO文件的权限。Mkfifo()建立的FIFO文件其他进程都可以用读写一般文件的方式存取。当使用open()来打开FIFO文件时，O\_NONBLOCK旗标会有影响 1、当使用O\_NONBLOCK 旗标时，打开FIFO 文件来读取的操作会立刻返回，但是若还没有其他进程打开FIFO 文件来读取，则写入的操作会返回ENXIO 错误代码。 2、没有使用O\_NONBLOCK 旗标时，打开FIFO 来读取的操作会等到其他进程打开FIFO文件来写入才正常返回。同样地，打开FIFO文件来写入的操作会等到其他进程打开FIFO 文件来读取后才正常返回。

FIFO 特殊文件与管道类似，只是创建方式不同。FIFO 特殊文件不是匿名通信通道，而是通过调用 mkfifo() 进入文件系统的。一旦以这种方式创建了先进先出特殊文件，任何进程都可以像普通文件一样打开它进行读写。不过，在对它进行任何输入或输出操作之前，必须同时在两端打开它。

为读取而打开 FIFO 通常会阻塞，直到其他进程为写入而打开同一个 FIFO，反之亦然。有关 FIFO 特殊文件的非阻塞处理，请参阅 fifo(7)。

* 返回值

若成功则返回0，否则返回-1，错误原因存于errno中。

* 错误代码

EACCESS 参数pathname所指定的目录路径无可执行的权限 EEXIST 参数pathname所指定的文件已存在。 ENAMETOOLONG 参数pathname的路径名称太长。 ENOENT 参数pathname包含的目录不存在 ENOSPC 文件系统的剩余空间不足 ENOTDIR 参数pathname路径中的目录存在但却非真正的目录。 EROFS 参数pathname指定的文件存在于只读文件系统内。

* 范例

#include<sys/types.h>#include<sys/stat.h>#include<fcntl.h>main(){char buffer[80];int fd;unlink(FIFO);mkfifo(FIFO,0666);if(fork()>0){char s[ ] = “hello!\n”;fd = open (FIFO,O\_WRONLY);write(fd,s,sizeof(s));close(fd);}else{fd= open(FIFO,O\_RDONLY);read(fd,buffer,80);printf(“%s”,buffer);close(fd);}}

* 执行

hello!

## 14.18 pclose

​关闭管道I/O

* 相关函数

Popen

* 表头文件

#include<stdio.h>

* 定义函数

int pclose(FILE \*stream);

* 函数说明

pclose()用来关闭由popen所建立的管道及文件指针。参数stream为先前由popen()所返回的文件指针。

* 返回值

返回子进程的结束状态。如果有错误则返回-1，错误原因存于errno中。

* 范例

参考popen()。

* 错误代码

ECHILD pclose()无法取得子进程的结束状态。

## 14.19 pipe

​建立管道。

* 相关函数

mkfifo，popen，read，write，fork

* 表头文件

#include<unistd.h>

* 定义函数

int pipe(int filedes[2]);

* 函数说明

pipe()会建立管道，这是一个可用于进程间通信的单向数据通道,并将文件描述词由参数filedes数组返回。pipefd[0] 指管道的读端，pipefd[1] 指管道的写端。写入管道写入端的数据会被内核缓冲，直到从管道读取端读取为止。更多详情，请参阅 pipe(7)。

如果 flags 为 0，则 pipe2() 与 pipe() 相同。可以将 flags 中的下列值按位 OR，以获得不同的行为.

* 返回值

成功时，返回 0。出错时，返回-1，设置 errno 表示出错，pipefd 保持不变。

在 Linux（和其他系统）上，pipe() 不会在失败时修改 pipefd。POSIX.1-2008 TC2 中增加了规范这一行为的要求。Linux 特有的 pipe2() 系统调用同样不会在失败时修改 pipefd。

* 错误代码

EMFILE 进程已用完文件描述词最大量。 ENFILE 系统已无文件描述词可用。 EFAULT 参数filedes数组地址不合法。

* 范例

/\* 父进程借管道将字符串“hello!\n”传给子进程并显示\*/#include <unistd.h>main(){int filedes[2];char buffer[80];pipe(filedes);if(fork()>0){/\* 父进程\*/char s[ ] = “hello!\n”;write(filedes[1],s,sizeof(s));}else{/\*子进程\*/read(filedes[0],buffer,80);printf(“%s”,buffer);}}

* 执行

hello!

## 14.20 popen

​建立管道I/O。

* 相关函数

pipe，mkfifo，pclose，fork，system，fopen

* 表头文件

#include<stdio.h>

* 定义函数

FILE \*popen(const char \*command, const char \*type);

* 函数说明

popen()会调用fork()产生子进程，然后从子进程中调用/bin/sh -c来执行参数command的指令。由于管道的定义是单向的，因此类型参数只能指定读取或写入，而不能同时指定读取和写入。

命令参数是一个指向包含 shell 命令行的空尾字符串的指针。该命令使用 -c 标志传递给 /bin/sh；解释（如有）由 shell 执行。

type 参数是一个指向空尾字符串的指针，读取时必须包含字母 “r”，写入时必须包含字母 “w”。自 glibc 2.9 起，该参数还可以包含字母 “e”，这将导致在底层文件描述符上设置执行时关闭标志 (FD\_CLOEXEC)；请参阅 open(2) 中关于 O\_CLOEXEC 标志的说明，了解为什么这可能有用。依照此type值，popen()会建立管道连到子进程的标准输出设备或标准输入设备，然后返回一个文件指针。随后进程便可利用此文件指针来读取子进程的输出设备或是写入到子进程的标准输入设备中。

除了必须使用 pclose() 而不是 fclose(3) 关闭之外，popen() 的返回值在所有方面都是一个正常的标准 I/O 流。向这样的数据流写入时，写入的是命令的标准输入；命令的标准输出与调用 popen() 的进程的标准输出相同，除非命令本身对其进行了修改。相反，从流中读取命令的标准输出，命令的标准输入与调用 popen() 的进程的标准输入相同。

* 返回值

成功时，返回一个指向可用于读取或写入管道的开放流的指针；如果 fork(2) 或 pipe(2) 调用失败，或者函数无法分配内存，则返回 NULL。一旦失败，这函数会设置 errno 以显示错误。

* 错误代码

EINVAL参数type不合法。

* 注意事项

在编写具SUID/SGID权限的程序时请尽量避免使用popen()，popen()会继承环境变量，通过环境变量可能会造成系统安全的问题。

* 范例

#include<stdio.h>main(){FILE \* fp;char buffer[80];fp=popen(“cat /etc/passwd”,”r”);fgets(buffer,sizeof(buffer),fp);printf(“%s”,buffer);pclose(fp);}

* 执行

root :x:0 0: root: /root: /bin/bash

# 第十五章 接口处理篇

## 15.1 accept

​接受socket连线。

* 相关函数

socket，bind，listen，connect

* 表头文件

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

* 定义函数

int accept(int s, struct sockaddr \*addr, int \*addrlen);

* 函数说明

accept()用来接受参数s的socket连线。参数s的socket必需先经bind()、listen()函数处理过，当有连线进来时accept()会返回一个新的socket处理代码，往后的数据传送与读取就是经由新的socket处理，而原来参数s的socket能继续使用accept()来接受新的连线要求。连线成功时，参数addr所指的结构会被系统填入远程主机的地址数据，参数addrlen为scokaddr的结构长度。关于结构sockaddr的定义请参考bind()。

参数 sockfd 是用 socket(2) 创建的套接字，用 bind(2) 绑定到本地地址，并在 listen(2) 之后监听连接。

参数 addr 是指向 sockaddr 结构的指针。该结构由通信层已知的对等套接字地址填充。addr 返回地址的具体格式由套接字的地址系列决定（参见 socket(2) 和相关协议的手册）。当 addr 为 NULL 时，将不填写任何内容；在这种情况下，addrlen 将不被使用，也应为 NULL。

addrlen 参数是一个值-结果参数：调用者必须将其初始化为包含 addr 指向的结构的大小（以字节为单位）；返回时，它将包含对等地址的实际大小。如果提供的缓冲区太小，返回的地址将被截断；在这种情况下，addrlen 返回的值将大于调用时提供的值。

如果队列中没有待处理的连接，且套接字未标记为非阻塞，accept() 会阻塞调用者，直到出现连接。如果套接字被标记为非阻塞，队列中也没有待处理的连接，accept() 就会以错误 EAGAIN 或 EWOULDBLOCK 失败。

要想知道套接字上是否有新连接，可以使用 select(2)、poll(2) 或 epoll(7)。当有新连接尝试时，系统将发送一个可读事件，然后您可以调用 accept() 为该连接获取一个套接字。或者，也可以将套接字设置为在套接字上发生活动时发送 SIGIO；详情请参阅 socket(7)。

* 返回值

成功时，这些系统调用会返回所接受套接字的文件描述符（一个非负整数）。如果出错，则返回-1，设置 errno 表示出错，addrlen 保持不变。

* 错误代码

EBADF 参数s 非合法socket处理代码。 EFAULT 参数addr指针指向无法存取的内存空间。 ENOTSOCK 参数s为一文件描述词，非socket。 EOPNOTSUPP 指定的socket并非SOCK\_STREAM。 EPERM 防火墙拒绝此连线。 ENOBUFS 系统的缓冲内存不足。 ENOMEM 核心内存不足。

* 范例

参考listen()。

## 15.2 bind

​对socket定位。

* 相关函数

socket，accept，connect，listen

* 表头文件

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

* 定义函数

int bind(int sockfd, struct sockaddr \*my\_addr, int addrlen);

* 函数说明

使用 socket(2) 创建套接字时，套接字存在于名称空间（地址族）中，但没有分配地址。传统上，这一操作被称为 “为套接字分配名称”。通常，在 SOCK\_STREAM 套接字接收连接（参见 accept(2)）之前，必须使用 bind() 为其分配一个本地地址。名称绑定所使用的规则因地址系列而异。

bind()用来设置给参数sockfd的socket一个名称。此名称由参数my\_addr指向一sockaddr结构，对于不同的socket domain定义了一个通用的数据结构struct sockaddr{unsigned short int sa\_family;char sa\_data[14];};sa\_family 为调用socket（）时的domain参数，即AF\_xxxx值。 sa\_data 最多使用14个字符长度。 此sockaddr结构会因使用不同的socket domain而有不同结构定义，例如使用AF\_INET domain，其socketaddr结构定义便为struct socketaddr\_in{unsigned short int sin\_family;uint16\_t sin\_port;struct in\_addr sin\_addr;unsigned char sin\_zero[8];};struct in\_addr{uint32\_t s\_addr;};sin\_family 即为sa\_family sin\_port 为使用的port编号 sin\_addr.s\_addr 为IP 地址 sin\_zero 未使用。

* 返回值

成功则返回0，失败返回-1，错误原因存于errno中。

* 错误代码

EBADF 参数sockfd 非合法socket处理代码。 EACCESS 权限不足 ENOTSOCK 参数sockfd为一文件描述词，非socket。

* 参数

addrlen为sockaddr的结构长度。

* 范例

参考listen()

## 15.3 connect

​建立socket连线。

* 相关函数

socket，bind，listen，

* 表头文件

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

* 定义函数

int connect(int sockfd, struct sockaddr \*serv\_addr, int addrlen);

* 函数说明

connect() 系统调用将文件描述符 sockfd 所指向的套接字连接到 serv\_addr指定的地址。addrlen 参数指定 serv\_addr的大小。serv\_addr中地址的格式由套接字 sockfd 的地址空间决定；详情请参见 socket(2)。

如果 socket sockfd 的类型是 SOCK\_DGRAM，那么 serv\_addr默认是发送数据报的地址，也是接收数据报的唯一地址。如果套接字的类型是 SOCK\_STREAM 或 SOCK\_SEQPACKET，则该调用会尝试与绑定到 serv\_addr指定地址的套接字建立连接。

某些协议套接字（如 UNIX 域流套接字）只能成功连接一次。

有些协议套接字（如 UNIX 和 Internet 域中的数据报套接字）可能会多次使用 connect() 来改变其关联。

某些协议套接字（例如 UNIX 和 Internet 域中的 TCP 套接字和数据报套接字）可以通过连接到 sockaddr 的 sa\_family 成员设置为 AF\_UNSPEC 的地址来解除关联；此后，套接字可以连接到另一个地址。(从 Linux 2.2 开始支持 AF\_UNSPEC）。

* 返回值

成功则返回0，失败返回-1，错误原因存于errno中。

* 错误代码

EBADF 参数sockfd 非合法socket处理代码 EFAULT 参数serv\_addr指针指向无法存取的内存空间 ENOTSOCK 参数sockfd为一文件描述词，非socket。 EISCONN 参数sockfd的socket已是连线状态 ECONNREFUSED 连线要求被server端拒绝。 ETIMEDOUT 企图连线的操作超过限定时间仍未有响应。 ENETUNREACH 无法传送数据包至指定的主机。 EAFNOSUPPORT sockaddr结构的sa\_family不正确。 EALREADY socket为不可阻断且先前的连线操作还未完成。

* 范例

/\* 利用socket的TCP client此程序会连线TCP server，并将键盘输入的字符串传送给server。TCP server范例请参考listen（）。\*/#include<sys/stat.h>#include<fcntl.h>#include<unistd.h>#include<sys/types.h>#include<sys/socket.h>#include<netinet/in.h>#include<arpa/inet.h>#define PORT 1234#define SERVER\_IP “127.0.0.1”main(){int s;struct sockaddr\_in addr;char buffer[256];if((s = socket(AF\_INET,SOCK\_STREAM,0))<0){perror(“socket”);exit(1);}/\* 填写sockaddr\_in结构\*/bzero(&addr,sizeof(addr));addr.sin\_family = AF\_INET;addr.sin\_port=htons(PORT);addr.sin\_addr.s\_addr = inet\_addr(SERVER\_IP);/\* 尝试连线\*/if(connect(s,&addr,sizeof(addr))<0){perror(“connect”);exit(1);}/\* 接收由server端传来的信息\*/recv(s,buffer,sizeof(buffer),0);printf(“%s\n”,buffer);while(1){bzero(buffer,sizeof(buffer));/\* 从标准输入设备取得字符串\*/read(STDIN\_FILENO,buffer,sizeof(buffer));/\* 将字符串传给server端\*/if(send(s,buffer,sizeof(buffer),0)<0){perror(“send”);exit(1);}}}

* 执行

$ ./connectWelcome to server!hi I am client! /\*键盘输入\*//\*<Ctrl+C>中断程序\*/

## 15.4 endprotoent

​结束网络协议数据的读取。

* 相关函数

getprotoent，getprotobyname，getprotobynumber，setprotoent

* 表头文件

#include<netdb.h>

* 定义函数

void endprotoent(void);

* 函数说明

endprotoent()用来关闭由getprotoent()打开的文件。

* 返回值

成功则返回新的socket处理代码，失败返回-1，错误原因存于errno中。

* 范例

参考getprotoent()

## 15.5 endservent

​结束网络服务数据的读取。

* 相关函数

getservent，getservbyname，getservbyport，setservent

* 表头文件

#include<netdb.h>

* 定义函数

void endservent(void);

* 函数说明

endservent()用来关闭由getservent()所打开的文件。

* 范例

参考getservent()。

## 15.6 getsockopt

​取得socket状态

* 相关函数

setsockopt

* 表头文件

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

* 定义函数

int getsockopt(int s, int level, int optname, void \*optval, socklen\_t \*optlen);

* 函数说明

getsockopt()操作文件描述符 s 所指套接字的选项。选项可能存在于多个协议级别，但它们总是存在于最上层的套接字级别。

操作套接字选项时，必须指定选项所在的层级和选项名称。要在套接字应用程序接口级别操作选项，必须指定级别为 SOL\_SOCKET。要在任何其他级别操作选项，必须提供控制该选项的相应协议的协议号。例如，要表示一个选项将由 TCP 协议解释，应将 level 设置为 TCP 的协议号；参见 getprotoent(3)。

参数 optval 和 optlen 用于访问 setsockopt() 的选项值。对于 getsockopt()，参数 optval 和 optlen 标识了一个缓冲区，请求的选项值将在该缓冲区中返回。对于 getsockopt()，optlen 是一个值-结果参数，最初包含 optval 指向的缓冲区的大小，返回时会被修改，以指示返回值的实际大小。如果不提供或不返回选项值，optval 可以为 NULL。

Optname 和任何指定的选项将不经解释传递给相应的协议模块进行解释。包含文件 <sys/socket.h> 包含套接字级选项的定义，如下所述。

大多数套接字级选项使用 int 参数作为 optval。对于 setsockopt()，如果要启用布尔选项，参数应为非零；如果要禁用选项，参数应为零。

* 返回值

成功时，标准选项返回 0。如果出错，则返回-1，并设置 errno 表示出错。

Netfilter 允许程序员定义带有相关处理程序的自定义套接字选项；对于此类选项，成功时的返回值就是处理程序返回的值。

* 错误代码

EBADF 参数s 并非合法的socket处理代码 ENOTSOCK 参数s为一文件描述词，非socket ENOPROTOOPT 参数optname指定的选项不正确 EFAULT 参数optval指针指向无法存取的内存空间

* 范例

#include<sys/types.h>#include<sys/socket.h>main(){int s,optval,optlen = sizeof(int);if((s = socket(AF\_INET,SOCK\_STREAM,0))<0) perror(“socket”);getsockopt(s,SOL\_SOCKET,SO\_TYPE,&optval,&optlen);printf(“optval = %d\n”,optval);close(s);}

* 执行

optval = 1 /\*SOCK\_STREAM的定义正是此值\*/

## 15.7 htonl

​将32位主机字符顺序转换成网络字符顺序。

* 相关函数

htons，ntohl，ntohs

* 表头文件

#include<netinet/in.h>

* 定义函数

unsigned long int htonl(unsigned long int hostlong);

* 函数说明

htonl（）用来将参数指定的32位hostlong 转换成网络字符顺序。

* 返回值

返回对应的网络字符顺序。

* 范例

参考getservbyport()或connect()。

## 15.8 htons

​将16位主机字符顺序转换成网络字符顺序。

* 相关函数

htonl，ntohl，ntohs

* 表头文件

#include<netinet/in.h>

* 定义函数

unsigned short int htons(unsigned short int hostshort);

* 函数说明

htons()用来将参数指定的16位hostshort转换成网络字符顺序。

* 返回值

返回对应的网络字符顺序。

* 范例

参考connect()。

## 15.9 inet\_addr

​将网络地址转成二进制的数字。

* 相关函数

inet\_aton，inet\_ntoa

* 表头文件

#include<sys/socket.h>#include<netinet/in.h>#include<arpa/inet.h>

* 定义函数

unsigned long int inet\_addr(const char \*cp);

* 函数说明

inet\_addr()用来将参数cp所指的网络地址字符串转换成网络所使用的二进制数字。网络地址字符串是以数字和点组成的字符串，例如:“163.13.132.68”。

* 返回值

成功则返回对应的网络二进制的数字，失败返回-1。

## 15.10 inet\_aton

​将网络地址转成网络二进制的数字。

* 相关函数

inet\_addr，inet\_ntoa

* 表头文件

#include<sys/scoket.h>#include<netinet/in.h>#include<arpa/inet.h>

* 定义函数

int inet\_aton(const char \*cp, struct in\_addr \*inp);

* 函数说明

inet\_aton()用来将参数cp所指的网络地址字符串转换成网络使用的二进制的数字，然后存于参数inp所指的in\_addr结构中。 结构in\_addr定义如下struct in\_addr{unsigned long int s\_addr;};

* 返回值

成功则返回非0值，失败则返回0。

## 15.11 inet\_ntoa

​将网络二进制的数字转换成网络地址。

* 相关函数

inet\_addr，inet\_aton

* 表头文件

#include<sys/socket.h>#include<netinet/in.h>#include<arpa/inet.h>

* 定义函数

char \*inet\_ntoa(struct in\_addr in);

* 函数说明

inet\_ntoa()用来将参数in所指的网络二进制的数字转换成网络地址，然后将指向此网络地址字符串的指针返回。

* 返回值

成功则返回字符串指针，失败则返回NULL。

## 15.12 listen

​等待连接。

* 相关函数

socket，bind，accept，connect

* 表头文件

#include<sys/socket.h>

* 定义函数

int listen(int s, int backlog);

* 函数说明

listen()用来等待参数s 的socket连线。参数backlog指定同时能处理的最大连接要求，如果连接数目达此上限则client端将收到ECONNREFUSED的错误。Listen()并未开始接收连线，只是设置socket为listen模式，真正接收client端连线的是accept()。通常listen()会在socket()，bind()之后调用，接着才调用accept()。

listen() 会将 s 所指向的套接字标记为被动套接字，即使用 accept(2) 接受传入连接请求的套接字。

s 参数是一个文件描述符，指向 SOCK\_STREAM 或 SOCK\_SEQPACKET 类型的套接字。

backlog 参数定义了 s 的待处理连接队列的最大长度。如果连接请求在队列已满时到达，客户端可能会收到一个错误信息，提示连接已满。或者，如果底层协议支持重传，该请求可能会被忽略，以便以后重新尝试连接时获得成功。

* 返回值

成功则返回0，失败返回-1，错误原因存于errno

* 错误代码

EBADF 参数sockfd非合法socket处理代码 EACCESS 权限不足 EOPNOTSUPP 指定的socket并未支援listen模式。

* 附加说明

listen()只适用SOCK\_STREAM或SOCK\_SEQPACKET的socket类型。如果socket为AF\_INET则参数backlog 最大值可设至128。

* 范例

#include<sys/types.h>#include<sys/socket.h>#include<netinet/in.h>#include<arpa/inet.h>#include<unistd.h>#define PORT 1234#define MAXSOCKFD 10main(){int sockfd,newsockfd,is\_connected[MAXSOCKFD],fd;struct sockaddr\_in addr;int addr\_len = sizeof(struct sockaddr\_in);fd\_set readfds;char buffer[256];char msg[ ] =”Welcome to server!”;if ((sockfd = socket(AF\_INET,SOCK\_STREAM,0))<0){perror(“socket”);exit(1);}bzero(&addr,sizeof(addr));addr.sin\_family =AF\_INET;addr.sin\_port = htons(PORT);addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);if(bind(sockfd,&addr,sizeof(addr))<0){perror(“connect”);exit(1);}if(listen(sockfd,3)<0){perror(“listen”);exit(1);}for(fd=0;fd<MAXSOCKFD;fd++)is\_connected[fd]=0;while(1){FD\_ZERO(&readfds);FD\_SET(sockfd,&readfds);for(fd=0;fd<MAXSOCKFD;fd++)if(is\_connected[fd]) FD\_SET(fd,&readfds);if(!select(MAXSOCKFD,&readfds,NULL,NULL,NULL))continue;for(fd=0;fd<MAXSOCKFD;fd++)if(FD\_ISSET(fd,&readfds)){if(sockfd = =fd){if((newsockfd = accept (sockfd,&addr,&addr\_len))<0)perror(“accept”);write(newsockfd,msg,sizeof(msg));is\_connected[newsockfd] =1;printf(“cnnect from %s\n”,inet\_ntoa(addr.sin\_addr));}else{bzero(buffer,sizeof(buffer));if(read(fd,buffer,sizeof(buffer))<=0){printf(“connect closed.\n”);is\_connected[fd]=0;close(fd);}elseprintf(“%s”,buffer);}}}}

* 执行

$ ./listenconnect from 127.0.0.1hi I am clientconnected closed.

## 15.13 ntohl

​将32位网络字符顺序转换成主机字符顺序。

* 相关函数

htonl，htons，ntohs

* 表头文件

#include<netinet/in.h>

* 定义函数

unsigned long int ntohl(unsigned long int netlong);

* 函数说明

ntohl()用来将参数指定的32位netlong转换成主机字符顺序。

* 返回值

返回对应的主机字符顺序。

* 范例

参考getservent()。

## 15.14 ntohs

​将16位网络字符顺序转换成主机字符顺序。

* 相关函数

htonl，htons，ntohl

* 表头文件

#include<netinet/in.h>

* 定义函数

unsigned short int ntohs(unsigned short int netshort);

* 函数说明

ntohs()用来将参数指定的16位netshort转换成主机字符顺序。

* 返回值

返回对应的主机顺序。

* 范例

参考getservent()。

## 15.15 recv

​经socket接收数据。

* 相关函数

recvfrom，recvmsg，send，sendto，socket

* 表头文件

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

* 定义函数

int recv(int s, void \*buf, int len, unsigned int flags);

* 函数说明

recv()用来接收远端主机经指定的socket传来的数据，并把数据存到由参数buf 指向的内存空间，参数len为可接收数据的最大长度。

* 参数

flags一般设0。其他数值定义如下: MSG\_OOB 接收以out-of-band 送出的数据。 MSG\_PEEK 返回来的数据并不会在系统内删除，如果再调用recv()会返回相同的数据内容。 MSG\_WAITALL强迫接收到len大小的数据后才能返回，除非有错误或信号产生。 MSG\_NOSIGNAL此操作不愿被SIGPIPE信号中断返回值成功则返回接收到的字符数，失败返回-1，错误原因存于errno中。

* 错误代码

EBADF 参数s非合法的socket处理代码 EFAULT 参数中有一指针指向无法存取的内存空间 ENOTSOCK 参数s为一文件描述词，非socket。 EINTR 被信号所中断 EAGAIN 此动作会令进程阻断，但参数s的socket为不可阻断 ENOBUFS 系统的缓冲内存不足。 ENOMEM 核心内存不足 EINVAL 传给系统调用的参数不正确。

* 范例

参考listen()。

## 15.16 recvfrom

​经socket接收数据。

* 相关函数

recv，recvmsg，send，sendto，socket

* 表头文件

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

* 定义函数

int recvfrom(int s, void \*buf, int len, unsigned int flags, struct sockaddr \*from, int \*fromlen);

* 函数说明

recv()用来接收远程主机经指定的socket 传来的数据，并把数据存到由参数buf 指向的内存空间，参数len 为可接收数据的最大长度。参数flags 一般设0，其他数值定义请参考recv()。参数from用来指定欲传送的网络地址，结构sockaddr 请参考bind()。参数fromlen为sockaddr的结构长度。

* 返回值

成功则返回接收到的字符数，失败则返回-1，错误原因存于errno中。

* 错误代码

EBADF 参数s非合法的socket处理代码 EFAULT 参数中有一指针指向无法存取的内存空间。 ENOTSOCK 参数s为一文件描述词，非socket。 EINTR 被信号所中断。 EAGAIN 此动作会令进程阻断，但参数s的socket为不可阻断。 ENOBUFS 系统的缓冲内存不足 ENOMEM 核心内存不足 EINVAL 传给系统调用的参数不正确。

* 范例

/\*利用socket的UDP client此程序会连线UDP server，并将键盘输入的字符串传给server。UDP server 范例请参考sendto（）。\*/#include<sys/stat.h>#include<fcntl.h>#include<unistd.h>#include<sys/typs.h>#include<sys/socket.h>#include<netinet/in.h>#include<arpa/inet.h>#define PORT 2345#define SERVER\_IP “127.0.0.1”main(){int s,len;struct sockaddr\_in addr;int addr\_len =sizeof(struct sockaddr\_in);char buffer[256];/\* 建立socket\*/if((s = socket(AF\_INET,SOCK\_DGRAM,0))<0){perror(“socket”);exit(1);}/\* 填写sockaddr\_in\*/bzero(&addr,sizeof(addr));addr.sin\_family = AF\_INET;addr.sin\_port = htons(PORT);addr.sin\_addr.s\_addr = inet\_addr(SERVER\_IP);while(1){bzero(buffer,sizeof(buffer));/\* 从标准输入设备取得字符串\*/len =read(STDIN\_FILENO,buffer,sizeof(buffer));/\* 将字符串传送给server端\*/sendto(s,buffer,len,0,&addr,addr\_len);/\* 接收server端返回的字符串\*/len = recvfrom(s,buffer,sizeof(buffer),0,&addr,&addr\_len);printf(“receive: %s”,buffer);}}

* 执行

(先执行udp server 再执行udp client)hello /\*从键盘输入字符串\*/receive: hello /\*server端返回来的字符串\*/

## 15.17 recvmsg

​经socket接收数据。

* 相关函数

recv，recvfrom，send，sendto，sendmsg，socket

* 表头文件

#include<sys/types.h>#include<sys/socktet.h>

* 定义函数

int recvmsg(int s, struct msghdr \*msg, unsigned int flags);

* 函数说明

recvmsg()用来接收远程主机经指定的socket传来的数据。参数s为已建立好连线的socket，如果利用UDP协议则不需经过连线操作。参数msg指向欲连线的数据结构内容，参数flags一般设0，详细描述请参考send()。关于结构msghdr的定义请参考sendmsg()。

* 返回值

成功则返回接收到的字符数，失败则返回-1，错误原因存于errno中。

* 错误代码

EBADF 参数s非合法的socket处理代码。 EFAULT 参数中有一指针指向无法存取的内存空间 ENOTSOCK 参数s为一文件描述词，非socket。 EINTR 被信号所中断。 EAGAIN 此操作会令进程阻断，但参数s的socket为不可阻断。 ENOBUFS 系统的缓冲内存不足 ENOMEM 核心内存不足 EINVAL 传给系统调用的参数不正确。

* 范例

参考recvfrom()。

## 15.18 send

​经socket传送数据

* 相关函数

sendto，sendmsg，recv，recvfrom，socket

* 表头文件

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

* 定义函数

int send(int s, const void \*msg, int len, unsigned int flags);

* 函数说明

send()用来将数据由指定的socket 传给对方主机。参数s为已建立好连接的socket。参数msg指向欲连线的数据内容，参数len则为数据长度。参数flags一般设0，其他数值定义如下 MSG\_OOB 传送的数据以out-of-band 送出。 MSG\_DONTROUTE 取消路由表查询 MSG\_DONTWAIT 设置为不可阻断运作 MSG\_NOSIGNAL 此动作不愿被SIGPIPE 信号中断。

* 返回值

成功则返回实际传送出去的字符数，失败返回-1。错误原因存于errno

* 错误代码

EBADF 参数s 非合法的socket处理代码。 EFAULT 参数中有一指针指向无法存取的内存空间 ENOTSOCK 参数s为一文件描述词，非socket。 EINTR 被信号所中断。 EAGAIN 此操作会令进程阻断，但参数s的socket为不可阻断。 ENOBUFS 系统的缓冲内存不足 ENOMEM 核心内存不足 EINVAL 传给系统调用的参数不正确。

* 范例

参考connect()

## 15.19 sendmsg

​经socket传送数据。

* 相关函数

send，sendto，recv，recvfrom，recvmsg，socket

* 表头文件

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

* 定义函数

int sendmsg(int s, const struct msghdr \*msg, unsigned int flags);

* 函数说明

sendmsg()用来将数据由指定的socket传给对方主机。参数s为已建立好连线的socket，如果利用UDP协议则不需经过连线操作。参数msg 指向欲连线的数据结构内容，参数flags一般默认为0，详细描述请参考send()。 结构msghdr定义如下struct msghdr{void \*msg\_name; /\*Address to send to /receive from . \*/socklen\_t msg\_namelen; /\* Length of addres data \*/struct iovec \* msg\_iov; /\* Vector of data to send/receive into \*/size\_t msg\_iovlen; /\* Number of elements in the vector \*/void \* msg\_control; /\* Ancillary dat \*/size\_t msg\_controllen; /\* Ancillary data buffer length \*/int msg\_flags; /\* Flags on received message \*/};

* 返回值

成功则返回实际传送出去的字符数，失败返回-1，错误原因存于errno

* 错误代码

EBADF 参数s 非合法的socket处理代码。 EFAULT 参数中有一指针指向无法存取的内存空间 ENOTSOCK 参数s为一文件描述词，非socket。 EINTR 被信号所中断。 EAGAIN 此操作会令进程阻断，但参数s的socket为不可阻断。 ENOBUFS 系统的缓冲内存不足 ENOMEM 核心内存不足 EINVAL 传给系统调用的参数不正确。

* 范例

参考sendto()。

## 15.20 sendto

​经socket传送数据。

* 相关函数

send , sendmsg,recv , recvfrom , socket

* 表头文件

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

* 定义函数

int sendto(int s, const void \*msg, int len, unsigned int flags, const struct sockaddr \*to, int tolen);

* 函数说明

sendto() 用来将数据由指定的socket传给对方主机。参数s为已建好连线的socket,如果利用UDP协议则不需经过连线操作。参数msg指向欲连线的数据内容，参数flags 一般设0，详细描述请参考send()。参数to用来指定欲传送的网络地址，结构sockaddr请参考bind()。参数tolen为sockaddr的结果长度。

* 返回值

成功则返回实际传送出去的字符数，失败返回－1，错误原因存于errno 中。

* 错误代码

EBADF 参数s非法的socket处理代码。 EFAULT 参数中有一指针指向无法存取的内存空间。 WNOTSOCK canshu s为一文件描述词，非socket。 EINTR 被信号所中断。 EAGAIN 此动作会令进程阻断，但参数s的soket为补课阻断的。 ENOBUFS 系统的缓冲内存不足。 EINVAL 传给系统调用的参数不正确。

* 范例

#include <sys/types.h>#include <sys/socket.h>#include <netinet.in.h>#include <arpa.inet.h>#define PORT 2345 /\*使用的port\*/main(){int sockfd,len;struct sockaddr\_in addr;char buffer[256];/\*建立socket\*/if(sockfd=socket (AF\_INET,SOCK\_DGRAM,0))<0){perror (“socket”);exit(1);}/\*填写sockaddr\_in 结构\*/bzero ( &addr, sizeof(addr) );addr.sin\_family=AF\_INET;addr.sin\_port=htons(PORT);addr.sin\_addr=hton1(INADDR\_ANY) ;if (bind(sockfd, &addr, sizeof(addr))<0){perror(“connect”);exit(1);}while(1){bezro(buffer,sizeof(buffer));len = recvfrom(socket,buffer,sizeof(buffer), 0 , &addr &addr\_len);/\*显示client端的网络地址\*/printf(“receive from %s\n “ , inet\_ntoa( addr.sin\_addr));/\*将字串返回给client端\*/sendto(sockfd,buffer,len,0,&addr,addr\_len);”}}

* 执行

请参考recvfrom()

## 15.21 setprotoent

​打开网络协议的数据文件。

* 相关函数

getprotobyname, getprotobynumber, endprotoent

* 表头文件

#include <netdb.h>

* 定义函数

void setprotoent(int stayopen);

* 函数说明

setprotoent()用来打开/etc/protocols， 如果参数stayopen值为1，则接下来的getprotobyname()或getprotobynumber()将不会自动关闭此文件。

setprotoent() 函数打开与数据库的连接，并将下一个条目设置为第一个条目。如果 stayopen 非零，则在调用 getproto\*() 函数之间不会关闭与数据库的连接。

## 15.22 setservent

​打开主机网络服务的数据文件。

* 相关函数

getservent, getservbyname, getservbyport, endservent

* 表头文件

#include <netdb.h>

* 定义函数

void setservent(int stayopen);

* 函数说明

setservent()用来打开/etc/services，如果参数stayopen值为1，则接下来的getservbyname()或getservbyport()将补回自动关闭文件。

setservent() 函数打开与数据库的连接，并将下一个条目设置为第一个条目。如果 stayopen 非零，则在调用 getserv\*() 函数之间不会关闭与数据库的连接。

## 15.23 setsockopt

​设置socket状态

* 相关函数

Getsockopt

* 表头文件

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

* 定义函数

int setsockopt(int s, int level, int optname, const void \*optval, socklen\_t optlen);

* 函数说明

setsockopt()用来设置参数s所指定的socket状态。参数level代表欲设置的网络层，一般设成SOL\_SOCKET以存取socket层。参数optname代表欲设置的选项，有下列几种数值: SO\_DEBUG 打开或关闭排错模式 SO\_REUSEADDR 允许在bind（）过程中本地地址可重复使用 SO\_TYPE 返回socket形态。 SO\_ERROR 返回socket已发生的错误原因 SO\_DONTROUTE 送出的数据包不要利用路由设备来传输。 SO\_BROADCAST 使用广播方式传送 SO\_SNDBUF 设置送出的暂存区大小 SO\_RCVBUF 设置接收的暂存区大小 SO\_KEEPALIVE 定期确定连线是否已终止。 SO\_OOBINLINE 当接收到OOB 数据时会马上送至标准输入设备 SO\_LINGER 确保数据安全且可靠的传送出去。

* 返回值

成功则返回0，若有错误则返回-1，错误原因存于errno。

* 参数

optval代表欲设置的值，参数optlen则为optval的长度。

* 附加说明

EBADF 参数s并非合法的socket处理代码 ENOTSOCK 参数s为一文件描述词，非socket ENOPROTOOPT 参数optname指定的选项不正确。 EFAULT 参数optval指针指向无法存取的内存空间。

* 范例

参考getsockopt()。

## 15.24 shutdown

​终止socket通信

* 相关函数

socket，connect

* 表头文件

#include<sys/socket.h>

* 定义函数

int shutdown(int s, int how);

* 函数说明

shutdown()用来终止参数s所指定的socket连线。参数s是连线中的socket处理代码，参数how有下列几种情况: how=0 终止读取操作。 how=1 终止传送操作 how=2 终止读取及传送操作

调用 shutdown() 会关闭与 sockfd 关联的套接字上的全部或部分全双工连接。如果 how 为 SHUT\_RD，将禁止继续接收。如果 how 为 SHUT\_WR，将禁止继续发送。如果 how 为 SHUT\_RDWR，将禁止继续接收和发送。

* 返回值

成功则返回0，失败返回-1，错误原因存于errno。

* 错误代码

EBADF 参数s不是有效的socket处理代码 ENOTSOCK 参数s为一文件描述词，非socket ENOTCONN 参数s指定的socket并未连线

## 15.25 socket

​建立一个socket通信。

* 相关函数

accept，bind，connect，listen

* 表头文件

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

* 定义函数

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

* 函数说明

socket() 创建一个通信端点，并返回一个指向该端点的文件描述符。调用成功后返回的文件描述符将是该进程当前未打开的最低文件描述符。

domain参数指定了一个通信域；它选择了用于通信的协议族。这些协议族在 <sys/socket.h> 中定义。目前 Linux 内核可以理解的格式包括

参数domain 指定使用何种的地址类型，完整的定义在/usr/include/bits/socket.h 内，底下是常见的协议:

PF\_UNIX/PF\_LOCAL/AF\_UNIX/AF\_LOCAL UNIX 进程通信协议

PF\_INET?AF\_INET Ipv4网络协议

PF\_INET6/AF\_INET6 Ipv6 网络协议

PF\_IPX/AF\_IPX IPX-Novell协议

PF\_NETLINK/AF\_NETLINK 核心用户接口装置

PF\_X25/AF\_X25 ITU-T X.25/ISO-8208 协议

PF\_AX25/AF\_AX25 业余无线AX.25协议

PF\_ATMPVC/AF\_ATMPVC 存取原始ATM PVCs

PF\_APPLETALK/AF\_APPLETALK appletalk（DDP）协议

PF\_PACKET/AF\_PACKET 初级封包接口

* 参数

type有下列几种数值: SOCK\_STREAM 提供双向连续且可信赖的数据流，即TCP。支持 OOB 机制，在所有数据传送前必须使用connect()来建立连线状态。 SOCK\_DGRAM 使用不连续不可信赖的数据包连接 SOCK\_SEQPACKET 提供连续可信赖的数据包连接 SOCK\_RAW 提供原始网络协议存取 SOCK\_RDM 提供可信赖的数据包连接 SOCK\_PACKET 提供和网络驱动程序直接通信。 protocol用来指定socket所使用的传输协议编号，通常此参考不用管它，设为0即可。

* 返回值

成功则返回socket处理代码，失败返回-1。

* 错误代码

EPROTONOSUPPORT 参数domain指定的类型不支持参数type或protocol指定的协议 ENFILE 核心内存不足，无法建立新的socket结构 EMFILE 进程文件表溢出，无法再建立新的socket EACCESS 权限不足，无法建立type或protocol指定的协议 ENOBUFS/ENOMEM 内存不足 EINVAL 参数domain/type/protocol不合法

* 范例

参考connect()。

# 第十六章 环境变量篇

## 16.1 getenv

​取得环境变量内容。

* 相关函数

putenv，setenv，unsetenv

* 表头文件

#include<stdlib.h>

* 定义函数

char \*getenv(const char \*name);

* 函数说明

getenv() 函数在环境列表中搜索环境变量名称，并返回指向相应值字符串的指针。参数name为环境变量的名称，如果该变量存在则会返回指向该内容的指针。环境变量的格式为name＝value。

GNU 特有的 secure\_getenv() 函数与 getenv() 函数类似，只是在需要 “安全执行 ”的情况下返回 NULL。如果在加载调用进程运行的程序时，以下条件之一为真，则需要安全执行：

- 进程的有效用户 ID 与其真实用户 ID 不匹配，或进程的有效组 ID 与其真实组 ID 不匹配（通常情况下，这是执行了

set-user-ID或set-group-ID程序的结果）；

- 可执行文件的有效能力位被设置；或

- 进程具有非空允许能力集。

如果某些 Linux 安全模块触发，也可能需要安全执行。

secure\_getenv() 函数用于通用程序库，以避免在 set-user-ID 或 set-group-ID 程序意外信任环境时出现漏洞。

* 返回值

执行成功则返回指向该内容的指针，找不到符合的环境变量名称则返回NULL。

* 范例

#include<stdlib.h>main(){char \*p;if((p = getenv(“USER”)))printf(“USER=%s\n”,p);}

* 执行

USER = root

## 16.2 putenv

​改变或增加环境变量。

* 相关函数

getenv，setenv，unsetenv

* 表头文件

#include<stdlib.h>

* 定义函数

int putenv(const char \*string);

* 函数说明

putenv() 函数用于添加或更改环境变量的值。参数字符串的形式为 name=value。如果 name 不存在于环境中，则字符串将被添加到环境中。如果 name 已经存在，则环境中 name 的值将改为 value。string 指向的字符串会成为环境的一部分，因此更改字符串会改变环境。

* 返回值

执行成功则返回0，有错误发生则返回-1。

* 错误代码

ENOMEM 内存不足，无法配置新的环境变量空间。

* 范例

#include<stdlib.h>main(){char \*p;if((p = getenv(“USER”)))printf(“USER =%s\n”,p);putenv(“USER=test”);printf(“USER+5s\n”,getenv(“USER”));}

* 执行

USER=rootUSER=root

## 16.3 setenv

​改变或增加环境变量。

* 相关函数

getenv，putenv，unsetenv

* 表头文件

#include<stdlib.h>

* 定义函数

int setenv(const char \*name, const char \*value, int overwrite);

* 函数说明

setenv()用来改变或增加环境变量的内容。参数name为环境变量名称字符串。

* 参数

如果 name 还不存在，函数 setenv() 会将变量 name 与 value 值一起添加到环境中。如果 name 确实存在于环境中，那么在overwrite非零的情况下，它的值将被改为 value。

如果overwrite为零，则不会更改 name 的值（setenv() 返回成功状态）。此函数复制 name 和 value 指向的字符串（与 putenv(3) 相反）。

* 返回值

执行成功则返回0，有错误发生时返回-1。

* 错误代码

ENOMEM 内存不足，无法配置新的环境变量空间

* 范例

#include<stdlib.h>main(){char \* p;if((p=getenv(“USER”)))printf(“USER =%s\n”,p);setenv(“USER”,”test”,1);printf(“USER=%s\n”,getenv(“USEr”));unsetenv(“USER”);printf(“USER=%s\n”,getenv(“USER”));}

* 执行

USER = rootUSER = testUSER = (null)

# 第十七章 终端控制篇

## 17.1 getopt

​分析命令行参数。

* 表头文件

#include<unistd.h>

* 定义函数

int getopt(int argc,char \* const argv[ ],const char \* optstring);

* 函数说明

getopt()用来分析命令行参数。参数argc和argv是由main()传递的参数个数和内容。参数optstring 则代表欲处理的选项字符串。此函数会返回在argv 中下一个的选项字母，此字母会对应参数optstring 中的字母。如果选项字符串里的字母后接着冒号“:”，则表示还有相关的参数，全域变量optarg 即会指向此额外参数。如果getopt()找不到符合的参数则会印出错信息，并将全域变量optopt设为“?”字符，如果不希望getopt()印出错信息，则只要将全域变量opterr设为0即可。

* 返回值

如果成功找到一个选项，则 getopt() 返回该选项字符。如果所有 命令行选项都已解析完毕，则 getopt() 返回-1。如果 getopt() 遇到 optstring 中没有的选项字符，则返回'? 如果 getopt() 遇到一个参数缺失的选项，那么返回值取决于 optstring 中的第一个

* 范例

#include<stdio.h>#include<unistd.h>int main(int argc,char \*\*argv){int ch;opterr = 0;while((ch = getopt(argc,argv,”a:bcde”))!= -1)switch(ch){case ‘a’:printf(“option a:’%s’\n”,optarg);break;case ‘b’:printf(“option b :b\n”);break;default:printf(“other option :%c\n”,ch);}printf(“optopt +%c\n”,optopt);}

* 执行

$./getopt –boption b:b$./getopt –cother option:c$./getopt –aother option :?$./getopt –a12345option a:’12345’

## 17.2 isatty

​判断文件描述词是否是为终端机

* 相关函数

Ttyname

* 表头文件

#include<unistd.h>

* 定义函数

int isatty(int desc);

* 函数说明

如果参数desc所代表的文件描述词为一终端机则返回1，否则返回0。

* 返回值

如果文件为终端机则返回1，否则返回0。

* 范例

参考ttyname()。

## 17.3 select

​I/O多工机制

* 表头文件

#include<sys/time.h>#include<sys/types.h>#include<unistd.h>

* 定义函数

int select(int n, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

* 函数说明

select()用来等待文件描述词状态的改变。参数n代表最大的文件描述词加1，参数readfds、writefds 和exceptfds 称为描述词组，是用来回传该描述词的读，写或例外的状况。底下的宏提供了处理这三种描述词组的方式:FD\_CLR(inr fd,fd\_set\* set)；用来清除描述词组set中相关fd 的位FD\_ISSET(int fd,fd\_set \*set)；用来测试描述词组set中相关fd 的位是否为真FD\_SET（int fd,fd\_set\*set）；用来设置描述词组set中相关fd的位FD\_ZERO（fd\_set \*set）； 用来清除描述词组set的全部位

警告：select() 只能监控小于 FD\_SETSIZE (1024) 的文件描述符编号，这对许多现代应用程序来说是一个不合理的低限制，而且这一限制不会改变。所有现代应用程序都应使用 poll(2) 或 epoll(7)，它们不会受到这种限制。

select()允许程序监控多个文件描述符，直到一个或多个文件描述符 “准备就绪”，可以进行某类 I/O 操作（如输入）。如果可以在不阻塞的情况下执行相应的 I/O 操作（例如读取 (2) 或足够小的写入 (2)），则该文件描述符被视为已准备就绪。

* 参数

timeout为结构timeval，用来设置select()的等待时间，其结构定义如下struct timeval{time\_t tv\_sec;time\_t tv\_usec;};

* 返回值

如果参数timeout设为NULL则表示select（）没有timeout。

* 错误代码

执行成功则返回文件描述词状态已改变的个数，如果返回0代表在描述词状态改变前已超过timeout时间，当有错误发生时则返回-1，错误原因存于errno，此时参数readfds，writefds，exceptfds和timeout的值变成不可预测。 EBADF 文件描述词为无效的或该文件已关闭 EINTR 此调用被信号所中断 EINVAL 参数n 为负值。 ENOMEM 核心内存不足

* 范例

fs\_set readset；FD\_ZERO(&readset);FD\_SET(fd,&readset);select(fd+1,&readset,NULL,NULL,NULL);if(FD\_ISSET(fd,readset){……}

## 17.4 ttyname

​返回一终端机名称。

* 相关函数

isatty

* 表头文件

#include<unistd.h>

* 定义函数

char \*ttyname(int desc);

* 函数说明

函数 ttyname() 返回一个指向在文件描述符 fd 上打开的终端设备空尾路径名的指针，如果出错（例如 fd 未连接到终端），则返回 NULL。返回值可能指向静态数据，可能被下一次调用覆盖。函数 ttyname\_r() 将路径名存储在长度为 buflen 的缓冲区 buf 中。

* 返回值

如果成功则返回指向终端机名称的字符串指针，有错误情况发生时则返回NULL。

* 范例

#include<unistd.h>#include<sys/types.h>#include <sys/stat.h>#include<fcntl.h>main(){int fd;char \* file = “/dev/tty”;fd = open (fiel,O\_RDONLY);printf(“%s”,file);if(isatty(fd)){printf(“is a tty.\n”);printf(“ttyname = %s \n”,ttyname(fd));}else printf(“ is not a tty\n”);close(fd);}

* 执行

/dev/tty is a ttyttyname = /dev/tty