Linux

Dy

2018 年 11 月 10 日

目录

1 第一章 IO 流

1.1 open

```
int open(const char *path, int flag, ...)
```

path:

flag:

O_RDONLY	只读
O_WRONLY	只写
O_RDWR	1
O_APPEND	2
O_CLOEXEC	3
O_CREAT	4
O_DIRECTORY	5

1.2 openat

```
int openat(int fd, const char *path, int oflag, ...);
```

fd: *fd 指向 filename 的目录，或者等于 AT_FDCWD，则 fd 等同于进程当前工作目录 fd*

flag:

O_RDONLY	只读
O_WRONLY	只写
O_RDWR	1
O_APPEND	2
O_CLOEXEC	3
O_CREAT	4
O_DIRECTORY	5

1.3 fcntl

```
fcntl(int fd,F_SETFL, flags|O_ASYNC)
```

int fd:

F_SETFL:

flags|O_ASYNC:

设置信号 IO，通过 fcntl(fd,F_SETOWN,pid) 设置接收 SIGIO 的进程，不能对终端设备使用

1.4 open

```
open(const char *path, int flag, int mode)
```

*path:

flag:

mode:

; flag = O_RDONLY|O_WRONLY|O_RDWR|O_APPEND|O_CLOEXEC|O_CREAT|O_DIRECTORY|O_E

1.5 openat

```
openat(fd|-1,filename|pathname,flags,mode)
```

fd|-1:

filename|pathname:

flags:

mode:

fd 指向 filename 的目录，或者等于 AT_FDCWD，则 fd 等同于进程当前工作目录 fd

1.6 close

```
close(fd)
```

fd:

只是进程文件描述符中对应的记录，并将 fd 所对应的全局文件打开表表项中的标记减一，当标记为 0 是文件关闭

1.7 read

read(fd, buf, len)

fd:

buf:

len:

文件空洞是可以读的，只不过读到的数据是 0，对于一般的文本文件读取，如果文件为空会返回 0，而对于一般的慢速设备比如 socket，如果当前 socket 缓冲区内无数据，则会阻塞，如果设置为非阻塞模式，当无数据时，则会返回-1，并且将错误码设置为 EAGAIN，如果另一段已经关闭，则返回 0，read 会读取换行符

1.8 aio_read

aio_read(aiocb *)

*:

1.9 write

write(fd, buf, len)

fd:

buf:

len:

可以对文件写大量的 0，并且会占用空间，而如果是偏移后写数据，中间的空洞不会占据空间 (视系统而定，mac 占用)，但是如果复制这个文件，复制的空洞部分会被 0 填充，因为 read 函数读取空洞部分读出的数据是 0，利用文件空洞可以实现多线程下载

1.10 lseek

lseek(fd, offset, position)

fd:

offset:

position:

1.11 fcntl

fcntl(fd, cmd, attr)

fd:

cmd:

attr:

cmd = F_GETFL|F_SETFL|F_GETFD|F_SETFD|F_DUPFD|F_F_DUPFD_CLOEXEC|F_GETOWN|F_S

F_SETFD 是设置文件描述符标志 (进程文件描述符表中,FD_CLOEXEC 标志)，F_SETFL 是设置文件状态标志 (全局文件打开表中，可设置 O_NONBLOCK)，F_DUPDF 是复制文件描述符但不会复制 F_CLOEXEC 标志，F_DUPDF_CLOEXEC 则会设置 F_CLOEXEC

1.12 dup2

dup2(oldfd, newfd)

oldfd:

newfd:

复制 oldfd 到 newfd，如果 newfd 以及被占用，则会先用 close 关闭它，复制后文件描述符设置的 close-on-exec 位会消失

1.13 pread

pread(fd, buf, len, offset)

fd:

buf:

len:

offset:

从 fd 的第 offset 的偏移量处读取 len 个字节的数据到 buf，不更新当前的文件偏移量，在多线程环境下有很大的作用

1.14 pwrite

pwrite(fd, buf, len, offset)

fd:

buf:

len:

offset:

从 fd 的第 offset 的偏移量处写入 buf 的前 len 个字节的数据，不更新当前的文件偏移量，在多线程环境下有很大的作用

1.15 readv

```
readv(int fd,struct iovec *,int len)
```

fd:

*:

len:

iovecvoid * iov_base,size_t iov_len iov_base 指向缓冲区，iov_len 表示缓冲区大小，iovec * 指向一个数组结构，len 表示数组长度，散布读，从 fd 读取数据，按 iovec 数组下标从小到大读取数据到数组元素中所指向的缓冲区中，也就是说如果 iovec[0] 指向的缓冲区装满后，然后存入 iovec[1] 指向的缓冲区

1.16 writev

```
writev(int fd,struct iovec *,int len)
```

fd:

*:

len:

聚集写，向 fd 写数据，按 iovec 数组下标从小到大写缓冲区的 iov_len 个数据到 fd 中，也就是说如果 iovec[0] 指向的缓冲区写到 fd 后，然后写入 iovec[1] 指向的缓冲区，如果自己要设置某种信息协议，比如发送的数据以某特定数据开头，特定数据结尾，则此时可以设置三个 iovec，分别用于头部数据，中间数据，尾部数据

1.17 truncate

```
truncate(path,len)
```

path:

len:

如果 len 大于文件大小，则会形成文件空洞

1.18 ftruncate

```
ftruncate(fd,len)
```

fd:

len:

可用于增大文件大小，然后用 memcpy 复制 mmap 的数据

1.19 mkstemp

mkstemp(**buf**[] = *"nameXXXXXX"*)

"nameXXXXXX":

不能传递静态分配的参数，程序结束后不会自动删除该文件

1.20 mkdtemp

mkdtemp(**buf**[] = *"nameXXXXXX"*)

"nameXXXXXX":

创建的是目录

1.21 tmpfile

tmpfile(**void**)

void:

返回创建的临时文件的流指针

2 环境

2.1 getenv

```
getenv("name")
```

"name":

返回真实地址，不是拷贝

2.2 putenv

```
putenv("name=value")
```

"name=value":

参数为真实的环境变量，不是副本，如果采用 char [] 保存字符串，则可以通过该指针进行后续更改，如使用静态分配，则不可更改

2.3 setenv

```
setenv("name","value",overwrite)
```

"name":

"value":

overwrite:

overwrite = TRUE|FALSE 等于 true 则覆盖，否则不覆盖

2.4 unsetenv

```
unsetenv("name")
```

"name":

删除环境变量

2.5 clearenv

clearenv(**void**)

void:

3 任意跳转

3.1 setjmp

setjmp(**jum_buf**)

jum_buf:

参数应该为全局变量，第一次调用返回 0，第二次调用会返回 longjmp 里的 val 参数

3.2 longjmp

longjmp(**jum_buf**, **val**)

jum_buf:

val:

调用时会跳转到 setjmp 函数处，不会回滚全局变量和自动变量的值，如果值保存在寄存器中，则值最后会回滚到调用第一次 setjmp 时的值，如果不想其回滚，则设置修饰符 volatile

4 堆操作

4.1 brk

brk(**void** ∗ **position**)

position:

4.2 sbrk

sbrk(**size**)

size:

4.3 brk

brk(**void** *)

*:

4.4 malloc

malloc(**size**)

size:

4.5 calloc

calloc(**num**, **size**)

num:

size:

4.6 realloc

realloc(ptr, size)

ptr:

size:

4.7 memalign

memalign(boundary, size)

boundary:

size:

4.8 alloc

alloc(size)

size:

5 系统用户文件处理

5.1 getpwnam

getpwnam(name)

name:

返回 passwd *，获取/etc/passwd 中匹配 name 的信息

5.2 getpwuid

getpwuid(**uid**)

uid:

通过 uid 进行匹配

5.3 getgrnam

getgrnam(**name**)

name:

返回 group *，获取/etc/group 中匹配 name 的信息

5.4 getgrgid

getgrgid(**uid**)

uid:

通过 uid 进行匹配

5.5 getpwent

getpwent(**void**)

void:

遍历/etc/passwd，每次调用后会指向下一条记录，会在第一次调用的时候打开/etc/passwd 文件

5.6 setpwent

setpwent(**void**)

void:

从初始处开始遍历，即重置指针，或者说将偏移量设为 0

5.7 endpwent

endpwent(**void**)

void:

关闭打开的/etc/passwd 文件

5.8 endgrent

endgrent(**void**)

void:

遍历/etc/group，每次调用后会指向下一条记录，会在第一次调用的时候打开/etc/group 文件

5.9 setgrent

setgrent(**void**)

void:

从初始处开始遍历，即重置指针，或者说将偏移量设为 0

5.10 getgrent

getgrent(**void**)

void:

关闭打开的/etc/group 文件

5.11 getspname

getspname(**name**)

name:

5.12 getspend

getspend(**name**)

name:

5.13 setspent

setspent(**void**)

void:

5.14 endspent

endspent(**void**)

void:

5.15 crypt

crypt(**pass**, **salt**)

pass:

salt:

6 进程用户信息

6.1 getlogin

getlogin()

返回登陆名，可以用 getpwuid(getuid()) 得到，不过如果一个用户有多个登录名 (多用登录名对应一个 uid)，则可能不会得到想要的结果

6.2 getuid

getuid()

返回实际用户 id

6.3 geteuid

geteuid()

返回有效用户 id

6.4 getgid

getgid()

返回实际组 id

6.5 getegid

getegid()

返回有效组 id

6.6 setuid

setuid(**uid**)

uid:

如果进程拥有特权进程权限，则可以将实际用户 id，有效用户 id 和保存的设置用户 id 改为 uid，如果非特权进程，而 uid 等于实际用户 id 或者保存的设置用户 id，则可以将有效 id 也改为 uid，比如某些设置了 set-user-id 为 root 的程序，当进程运行后，有效 id 为 root，保存的设置用户 id 也为 root，如果不进行某些操作，这个程序会一直以特权进程运行，此时可以用 setuid(getuid()) 将有效用户先设为实际用户 id，以用户权限运行，等到运行需要某些特权的命令时再用 setuid(程序刚运行时 geteuid() 的有效用户 id)，调用完命令后再将有效用户 id 恢复，保证最小权限原则

6.7 setgid

setgid(**uid**)

uid:

6.8 seteuid

seteuid(**euid**)

euid:

如果是特权进程，只更改有效用户 id 为 uid，非特权进程可以将有效用户 id 改为实际用户 id 或保存的设置用户 id

6.9 setegid

setegid(**egid**)

egid:

6.10 setreuid

setreuid(**uid**,**euid**)

uid:

euid:

如果是特权进程，将实际用户 id 设置为 uid，有效用户 id 和保存的设置用户 id 设置为 euid，参数取-1 对应 id 可保存不变

6.11 setregid

setregid(**gid**,**egid**)

gid:

egid:

6.12 getresuid

getresuid(&**uid**,&**euid**,&**suid**)

&uid:

&euid:

&suid:

mac 上无法使用

6.13 getresgid

getresgid(&**uid**,&**euid**,&**suid**)

&uid:

&euid:

&suid:

6.14 setfsuit

setfsuit(**fsuid**)

fsuid:

6.15 setfsgid

setfsgid(**fsgid**)

fsgid:

6.16 getgroups

getgroups(size ,gid_t [])

size:

[]:

将调用进程所属用户的各附属组 id 写入到数组中，最多写入 size 个附属组 id

6.17 setgroups

setgroups(size ,gid_t [])

size:

[]:

为调用进程设置附属组 id

7 时间

7.1 gettimeofday

gettimeofday(timeval *,NULL)

*:

NULL:

与 time() 功能相似，但是精度更高，timeval.tv_sec 和 time() 的返回值相同，而 timeval.tv_nsec 提供微秒级精度

7.2 time

time(time_t *)

*:

返回自 epoch 到现在 GMT 时间的秒数

7.3 ctime

ctime(time_t *)

*:

将日历时间 (time_t) 转换成人们可读取的时间日期字符串，会进行本地化处理

7.4 gmtime

gmtime(time_t *)

*:

将日历时间转换成分解的时间结构 tm，以格林时间 GMT 为标准，返回值为指针类型，说明该函数是不可重入的，tmtm_sec(0-60),tm_min(0-59),tm_hour(0-23),tm_mday(1-31),tm_mon(0-11),tm_year(>=1900),tm_wday(0-6),tm_yday(0-365),tm_isdst(夏令时标志)

7.5 localtime

localtime(time_t *)

*:

将日历时间转换成分解的时间结构 tm，以本地时间为标准，所以会在对日历时间处理时考虑本地时区和夏令时标志

7.6 mktime

mktime(**tm** ∗)

∗:

将 tm 结构中的年月日为参数，转化为 time_t 值

7.7 asctime

asctime(**tm** ∗)

∗:

将 tm 转换成人们可读取的时间日期字符串，不会进行本地化处理，所以可用 localtime() 的返回值作为参数

7.8 strftime

strftime(**buf**, **len** , **format** ,**tm** ∗)

buf:

len:

format:

∗:

将 tm 格式化输出

7.9 strptime

strptime(**buf**, **format** ,**tm** ∗)

buf:

format:

*:

strftime 的逆函数，将 buf 中的字符串根据 format 转换成相应的 tm 结构

8 进程优先级

8.1 nice

nice(**incr**)

incr:

将 insr 参数增加到进程的 nice 值上，nice 值越小，优先级越高

8.2 getpriority

getpriority(**which**,**who**)

which:

who:

which = PRIO_PROCESS|PRIO_PGRP|PRIO_USER 分别表示进程，进程组，用户，如果 who 为 0，则表示调用进程的相应 which

8.3 setpriority

setpriority(**which**,**who**)

which:

who:

8.4 times

```
times(struct tms *)
```

*:

tmstms_utime,tms_stime,tms_cutime,tms_cstime 分别为用户 CPU 时间，系统 CPU 时间，子进程中执行的用户 CPU 时间，子进程中执行的系统 CPU 时间，获取各值应该需要调用两次 times，用两个 tms 结构体中对应的各个数相减可得到，可用两次 times 的返回值相减获取进程实际生存时间，单位为 clock_t，可除以 _SC_CLK_TCK 的值获取实际秒数

8.5 clock

```
clock(void)
```

void:

9 系统信息

9.1 sysconf

```
sysconf(name)
```

name:

获取运行时的系统限制值，与文件和目录无关 name = _SC_* _SC_CLK_TCK 为每秒滴答数，可用于转换 clock_t(times 返回值)

9.2 pathconf

pathconf(**path** ,**name**)

path:

name:

获取运行时的系统限制值，与文件和目录有关 name = _PC_*

9.3 fpathconf

fpathconf(**fd** ,**name**)

fd:

name:

获取运行时的系统限制值，与文件和目录有关 name = _PC_*

9.4 uname

uname(**utsname** ∗)

*:

返回主机和操作系统的相关信息

9.5 gethostname

gethostname(**name**, **len**)

name:

len:

返回主机名，等于在终端运行 hostname

10 IO 缓冲区设置

10.1 setvbuf

setvbuf(**FILE** ∗, **buf**, **mode**, **size**)

*:

buf:

mode:

size:

mode = _IOFBF|_IOLBF|_IONBF 分别为全缓冲，行缓冲和不带缓冲

10.2 setbuf

setbuf(**FILE** ∗, **buf**)

*:

buf:

可以用来打开和关闭标准 io 缓冲，默认是全缓冲模式

10.3 setbuffer

setbuffer(**FILE** ∗, **buf**, **size**)

*:

buf:

size:

11 磁盘同步

11.1 sync

sync(**void**)

void:

将修改过的块缓冲区排入写队列，并不等待数据写会磁盘

11.2 fsync

fsync(**fd**)

fd:

将 fd 所指文件中被修改过的块缓冲区和被修改过的文件属性写入磁盘，等待写入完成后返回

11.3 fdatasync

fdatasync(**fd**)

fd:

将 fd 所指文件中被修改过的块缓冲区写入磁盘，等待写入完成后返回

11.4 posix_fadvise

posix_fadvise(**fd**,**offset**,**len**,**advice**)

fd:

offset:

len:

advice:

12 文件 IO

12.1 fopen

fopen(**FILE** ∗,**pathname**,**type**)

*:

pathname:

type:

type = "r+w+a"

12.2 freopen

freopen(**pathname**,**type**,**FILE** ∗)

pathname:

type:

*:

在一个指定的流上打开一个指定的文件

12.3 fmemopen

fmemopen(**buf**,**size**,**type**)

buf:

size:

type:

内存流，返回一个文件指针指向这个 buf 缓冲区，标准 io 对文件的操作实际上是对磁盘中数据的操作，而这个并不绑定实际的文件，可将 buf 视为一个文件

12.4 fileno

fileno(**FIEL** *)

*:

12.5 fdopen

fdopen(**fd** ,**mode**)

fd:

mode:

为 fd 返回一个流标识符

12.6 fclose

fclose(**FILE** *)

*:

关闭一个打开流，关闭之前冲刷缓冲区的数据

12.7 getc

getc(**FILE** ∗)

∗:

12.8 fgetc

fgetc(**FILE** ∗)

∗:

12.9 getchar

getchar(**void**)

void:

等同于 getc(stdin)

12.10 ungetc

ungetc(**char**,**FILE** ∗)

char:

∗:

将字符压送回流的最前端

12.11 putc

putc(**char**,**FILE** *)

char:

*:

12.12 fputc

fputc(**char**,**FILE** *)

char:

*:

12.13 putchar

putchar(**char**)

char:

等同于 putc(char,stdout)

12.14 fgets

fgets(**buf**, **size**,**FILE** *)

buf:

size:

*:

一次读取一行数据

12.15 gets

gets(**buf**)

buf:

已经废弃，从标志输入读取并且会在尾部删除换行符

12.16 fputs

fputs(**buf**,**FILE** *)

buf:

*:

将 buf 中的数据写到指定的流中，数据要以 null 字符结尾，但是不会向流写入空字符

12.17 puts

puts(**buf**)

buf:

将 buf 中的数据写到标准输出，并追加一个换行符

12.18 fread

fread(**addr**, **size** ,**num**,**FILE** *)

addr:

size:

num:

*:

12.19 fwrite

fwrite(**addr**, **size**, **num**, **FILE** *)

addr:

size:

num:

*:

将地址 addr 开始的 num 个 size 大小的数据写入 FILE * 流，可用 mmap(NULL,40,PROT_READ|PROT_WRITE

加上 memcpy 实现针对 fd 的版本

12.20 ftell

ftell(**FILE** *)

*:

返回当前偏移量

12.21 fseek

fseek(**fd**, **offset**, **whence**)

fd:

offset:

whence:

whence = SEEK_SET|SEEK_CUR|SEEK_END 改变当前偏移量

12.22 ferror

ferror(**FILE** *)

*:

12.23 feof

feof(**FILE** *)

*:

检测是否已读到文件结尾

12.24 clearerr

clearerr(**FILE** *)

*:

消除 FILE 中的两个错误标志

13 文件系统挂载

13.1 mount

mount(**path**, **target**)

path:

target:

13.2 umount

umount(**target**)

target:

13.3 umount2

umount2(**target**, **flag**)

target:

flag:

14 文件信息

14.1 stat

stat(**path**, **stat** ∗)

path:

*:

检查文件属性，类型，大小，链接数，用户 id，组 id，最后修改时间，最后访问时间，inode 号，占用字节块数量 (这里的字节块大小在 mac 上是 512 字节，因为一个数据块大小是 4K(根据文件系统决定)，所以一个一字节的文件也会占用 8 个字节块)

14.2 lstat

lstat(**path**, **stat** ∗)

path:

*:

//不会对符号链接解引用

14.3 fstat

fstat(**fd**, **stat** ∗)

fd:

:

14.4 fstatat

fstatat(**fd**|−1, **filename**|**pathname**, **stat** ∗, **flags**)

fd|-1:

filename|pathname:

*:

flags:

如果 flags = AT_SYMLINK_NOFOLLOW，等同于 lstat，如果 fd = AT_FDCWD，并且 filename 是相对路径名，则 fd 等同于进程当前工作目录 fd

15 文件时间属性更改

15.1 utime

utime(**path**, **utimbuf** ∗)

path:

*:

15.2 utimes

utimes(**path**, **timeval** [2])

path:

[2]:

更改访问时间 (st_atim) 和修改时间 (st_mtim), 不能改变 st.ctim，因为调用这个函数时，该字段就会自动更新

15.3 futimes

futimes(**fd**, **timeval** [2])

fd:

[2]:

15.4 lutimes

lutimes(**path**, **timeval** [2])

path:

[2]:

不会对符号链接解引用

15.5 utimensat

utimensat(**dirfd**, **path**, **timespec** [2], **flag**)

dirfd:

path:

[2]:

flag:

15.6 futimens

futimens(**fd**, **timespec** [2])

fd:

[2]:

16 文件权限及所有者

16.1 chown

chown(**pathname**, **uid**, **gid**)

pathname:

uid:

gid:

如果 uid 或者 gid 中的任意一个参数是-1，则对应的 id 不变

16.2 fchown

fchown(**pathfd**, **uid**, **gid**)

pathfd:

uid:

gid:

16.3 lchown

lchown(**pathname**, **uid** , **gid**)

pathname:

uid:

gid:

16.4 fchownat

fchownat(**fd** | −1 , **filename** | **pathname**, **uid** , **gid** , **flags**)

fd|-1:

filename|pathname:

uid:

gid:

flags:

flags = AT_SYMLINK_NOFOLLOW 设置该标签不会解引用，也就是直接改变符号链接的 uid 和 gid

16.5 chmod

chmod(**pathname** , **mode**)

pathname:

mode:

会对符号链接解引用

16.6 fchmod

fchmod(**fd**,**mode**)

fd:

mode:

16.7 fchmodat

fchmodat(**fd**|−1,**filename**|**pathname**,**mode**, **flags**)

fd|-1:

filename|pathname:

mode:

flags:

flsgs = AT_SYMLINK_NOFOLLOW 设置该标签不会解引用符号链接

16.8 access

access(**path**,**mode**)

path:

mode:

mode = F_OK|R_OK|W_OK|X_OK 以实际用户 id 和实际组 id 检测访问权限，四个选项分别为是否存在，是否有读权限，写权限，执行权限，会对符号链接解引用，使用 faccessat 并设置 AT_SYMLINK_NOFOLLOW 也无效

16.9 faccessat

faccessat(**fd**, **filename**, **mode**, **flags**)

fd:

filename:

mode:

flags:

flags = AT_EACCESS 如果设置这个标志则使用有效用户 id 和有效组 id 检测访问权限

16.10 setxattr

setxattr(**path**, **name**, **value**, **size**, **flag**)

path:

name:

value:

size:

flag:

//只能在 linux 下使用，下同

16.11 lsetxattr

lsetxattr(**path**, **name**, **value**, **size**, **flag**)

path:

name:

value:

size:

flag:

16.12 fsetxattr

fsetxattr(**fd**,**name**,**value**,**size**,**flag**)

fd:

name:

value:

size:

flag:

16.13 getxattr

getxattr(**path**,**name**,**value**,**size**,**flag**)

path:

name:

value:

size:

flag:

//只能在 linux 下使用，下同

16.14 lgetxattr

lgetxattr(path,name,value,size,flag)

path:

name:

value:

size:

flag:

16.15 fgetxattr

fgetxattr(fd,name,value,size,flag)

fd:

name:

value:

size:

flag:

16.16 removexatte

removexatte(path,name)

path:

name:

16.17 lremovexatte

lremovexatte(path,name)

path:

name:

16.18 fremovexatte

fremovexatte(fd,name)

fd:

name:

16.19 listxattr

listxattr(path,list,size)

path:

list:

size:

16.20 llistxattr

llistxattr(path,list,size)

path:

list:

size:

16.21 flistxattr

flistxattr(**fd**, **list** , **size**)

fd:

list:

size:

17 硬链接与软链接

17.1 link

link(**oldpath**, **newpath**)

oldpath:

newpath:

等同于 ln -n oldpath newpath 该函数会解引用符号链接

17.2 linkat

linkat(**ofd**, **oldpath**, **efd**, **newpath**, **flags**)

ofd:

oldpath:

efd:

newpath:

flags:

flags = AT_SYMLINK_FOLLOW 如果设置会对符号链接解引用，不设置会对软链接直接失败

17.3 unlink

unlink(**path**)

path:

如果 path 是符号链接，则直接对符号链接起作用

17.4 unlinkat

unlinkat(**fd**,**path**,**flags**)

fd:

path:

flags:

flags = AT_REMOVEDIR 设置该标签可以对目录进行操作

17.5 rename

rename(**oldpath**,**newpath**)

oldpath:

newpath:

不对符号链接解引用

17.6 renameat

renameat(**ofd**, **oldpath**, **nfd**, **newpath**)

ofd:

oldpath:

nfd:

newpath:

17.7 symlink

symlink(**filepath**, **linkpath**)

filepath:

linkpath:

17.8 symlinkat

symlinkat(**filepath**, **fd**, **linkpath**)

filepath:

fd:

linkpath:

17.9 readlink

readlink(**path**, **buf**, **size**)

path:

buf:

size:

直接读符号链接文件的数据，buf 中返回的符号链接内容不以 null 字符结尾

18 目录操作

18.1 mkdir

mkdir(**pathname**,**mode**)

pathname:

mode:

18.2 mkdirat

mkdirat(**fd** ,**pathname**, ,**pde**)

fd:

pathname:

:

pde:

18.3 rmdir

rmdir(**pathname**)

pathname:

18.4 remove

remove(**pathname**)

pathname:

//需要为绝对地址

18.5 opendir

opendir(**dirpath**)

dirpath:

//返回 DIR *

18.6 fdopendir

fdopendir(**fd**)

fd:

18.7 readdir

readdir(**DIR** *)

*:

//返回 dirent *

18.8 rewinddir

rewinddir(**DIR** *)

*:

18.9 closedir

closedir(**DIR** *)

*:

18.10 dirfd

dirfd(**DIR** *)

*:

18.11 nftw

nftw(**dirpath**, **function**, **num_of_fd**, **flags**)

dirpath:

function:

num_of_fd:

flags:

int funciton(pathname,stat*,typeflag,FTW *)

18.12 getcwd

getcwd(**buf**, **size**)

buf:

size:

18.13 chdir

chdir(**newpath**)

newpath:

改变当前工作目录，会对路径中的符号链接解引用

18.14 fchdir

fchdir(**fd**)

fd:

18.15 chroot

chroot(**pathname**)

pathname:

18.16 realpath

realpath(**pathname**)

pathname:

解析出绝对路径名

18.17 dirname

dirname(**pathname**)

pathname:

18.18 basename

basename(**pathname**)

pathname:

19 信号

19.1 signal

signal(**signum**,**function**)

signum:

function:

function = SIG_DEL | SIG_IGN | function 调用信号处理程序过程中将阻塞一切信号

19.2 kill

kill(**pid**,**signum**)

pid:

signum:

19.3 raise

raise(**signum**)

signum:

== kill(getpid(),signum)

19.4 killpg

killpg(**gid**,**signum**)

gid:

signum:

== kill(-gid,signum)

19.5 strsignal

strsignal(**signum**)

signum:

返回对该信号描述的字符串指针

19.6 psignal

psignal(**signum**,**meg**)

signum:

meg:

19.7 sigemptyset

sigemptyset(**sigset_t** *)

*:

将信号集设置为空

19.8 sigfillset

sigfillset(sigset_t *)

*:

将信号集填充入所有类型的信号

19.9 sigaddset

sigaddset(sigset_t *,signum)

*:

signum:

向信号集中增加一个信号

19.10 sigdelset

sigdelset(sigset_t *,signum)

*:

signum:

将信号集中的一个信号删除

19.11 sigismember

sigismember(sigset_t *,signum)

*:

signum:

19.12 sigandset

```
sigandset(sigset\_t * result,sigset\_t *,sigset\_t *)
```

result:

*:

*:

19.13 sigorset

```
sigorset(sigset\_t * result,sigset\_t *,sigset\_t *)
```

result:

*:

*:

19.14 sigisemptyset

```
sigisemptyset(sigset\_t *)
```

*:

19.15 sigprocmask

```
sigprocmask(flag,sigset\_t * new,sigset\_t * old)
```

flag:

new:

old:

flag = SIG_BLOCK|SIG_UNBLOCK|SIG_SETMASK SIG_BLOCK 是或操作,SIG_UNBLOCK 是 & 操作，SIG_SETMASK 是赋值操作，处于信号集中的信号会被阻塞

19.16 sigpending

```
sigpending(sigset\_t *)
```

*:

返回当前被阻塞的信号

19.17 sigaction

```
sigaction(signum, sigaction * new, sigaction * old)
```

signum:

new:

old:

切记初始化，将 sa_mask 设置为空，sa_flags 设置为 0，尤其是 SIGINT sigaction.sa_flags = SA_RESTART|SA_NODEFER|SA_SIGINFO|SA_INTERRUPT 设置 SA_RESTART 后可在部分文件 io 被信号中断后进行重启 (文件 io 只有对低速设备操作时才会被中断),SA_INTERRUPT 关闭自动重启 (有些操作系统默认中断后自动重启)，SA_SIGINFO 让信号处理函数变成 void (int,siginfo *,char *) 形式

19.18 pause

pause(**void**)

void:

阻塞直到接收到一个信号

19.19 sigsetjmp

sigsetjmp(**sigjmp_buf**,**mode**)

sigjmp_buf:

mode:

如果 mode 为 0，等价于 setjmp 在 mac os 下 setjmp 可以代替 sigsetjmp，会恢复信号掩码，linux 则不会

19.20 setlongjmp

setlongjmp(**sigjmp_buf**, **val**)

sigjmp_buf:

val:

19.21 abort

abort()

产生 SIGABRT 信号，设置了信号处理函数也会终止进程，阻塞和忽略该信号也无用

19.22 sigaltstack

```
sigaltstack(stack\_t * new,stack\_t * old)
```

new:

old:

19.23 sigqueue

```
sigqueue(pid,sig,union sigval value)
```

pid:

sig:

value:

mac 不支持

19.24 sigsuspend

```
sigsuspend(sigset\_t *)
```

*:

一般在两个 sigprocmask 之间调用，解除非 sigset 里所包含信号的阻塞和 pause(两个函数为原子操作)，然后函数阻塞直到接收到一个信号 (中断 sigsuspend 里的 pause，如果有经 sigsuspend 调用而解除阻塞的信号，则会立即返回)，之后将信号掩码该为调用 sigsuspend 之前的掩码，由于信号在被解除阻塞后会立即发送给进程，则如果在调用 sigprocmask 后想解除阻塞使信号被接收然后调用 pause，但这个信号并不会打断 pause，因为他在 pause 运行之前就已经被信号处理程序接收

19.25 sigwait

```
sigwait(sigset\_t * set,int * signop)
```

set:

signop:

如果 set 中的信号集包含有被阻塞的信号，移除那些被阻塞的信号，函数立刻返回，否则阻塞直到收到集合中的信号 (无论信号是否被阻塞)，返回后也不会取消对该信号的阻塞

19.26 signalfd

```
signalfd(fd,sigset\_t *,flags)
```

fd:

*:

flags:

19.27 sighold

```
sighold(sig)
```

sig:

19.28 sigrelse

```
sigrelse(sig)
```

sig:

19.29 sigignore

sigignore(sig)

sig:

19.30 sigblock

sigblock(mask)

mask:

写 mask 的就是要通过 sigmask 对信号进行转换

19.31 sigsetmask

sigsetmask(mask)

mask:

19.32 sigpause

sigpause(mask)

mask:

19.33 sigmask

sigmask(sig)

sig:

将信号值进行转换，变成可以" 与" 操作的形式

19.34 sigvec

sigvec(**sig**,**sigvec** ∗,**sigvec** ∗)

sig:

*:

*:

19.35 getitimer

getitimer(**which**,**itimerval** ∗)

which:

*:

which = ITIMER_REAL|ITIMER_VIRTUAL|ITIMER_PROF REAL 代表现实时间，VIRTUAL 代表用户态时间，PROF 为用户态加内核态时间

19.36 setitimer

setitimer(**which**,**itimerval** ∗ **new**,**itimerval** ∗ **old**)

which:

new:

old:

19.37 alarm

alarm(seconds)

seconds:

定时器超时后会产生 SIGALRM 信号，如果 seconds 为 0，并且此前有注册过且还未到期的闹钟，则取消该闹钟并返回剩余时间

19.38 sleep

sleep(seconds)

seconds:

休眠一段时间，可被信号中断

19.39 nanosleep

nanosleep(timespec *,timespec *)

*:

*:

更高精度的睡眠，如果被中断，则在第二个参数中返回未休眠完的时间

19.40 clock_nanosleep

clock_nanosleep(clockid_t,flags,timespec *,timespec *)

clockid_t:

flags:

*:

*:

flags = 0|TIMER_ABSTIME mac 上不可用

20 时钟

20.1 clock_gettime

clock_gettime(clockid_t, timespec *)

clockid_t:

*:

获取指定时钟的时间 clockid_t = CLOCK_REALTIME|CLOCK_MONOTONIC|CLOCK_PROCESS_CPUTIM

分别表示实时系统时间，不带负跳数的系统实时时间，调用进程的 CPU 时间，调用线程的 CPU 时间

20.2 clock_getres

clock_getres(clockid_t, timespec *)

clockid_t:

*:

将 timespec 结构体初始化为 clockid_t 参数对应的时钟精度，如果精度为 1 毫秒，则 tv_sec 字段就是 0，tv_nsec 字段就是 1000000

20.3 clock_settime

clock_settime(clockid_t, timespec *)

clockid_t:

*:

设置时钟值

21 进程生成

21.1 fork

fork(**void**)

void:

当执行 fork 之后，子进程会复制进程文件描述符表，但不会复制全局文件打开表，因为该表为内核级，当子进程或者父进程其一使用 close 关闭了该描述符后，另一个仍然可以进行 IO 操作，子进程一定要以 exit 退出，特别是在 socket 并发服务器里，很重要

21.2 vfork

vfork(**void**)

void:

因为 fork 会复制父进程的页表，如果 fork 后马上就执行 exec，那么这个复制是不必要的，所以 vfork 不会复制父进程的页表，如果 vfork 后没有立即执行 exec，那么子进程实际是在操作父进程的进程空间，vfork 会保证子进程先运行, 并且在它执行 exit 或者 _exit 后父进程才可以被调度

21.3 exit

exit(**status**)

status:

对每个打开流调用 fclose() 函数，并调用登记过的终止处理函数后终止

21.4 _exit

```
\_exit(status)
```

status:

丢弃缓冲区的 io 数据，直接终止

21.5 atexit

```
atexit(void (*func)
```

(*func:

(void)) 如果使用 _exit() 来退出不会执行被登记过的函数

21.6 wait

```
wait(status)
```

status:

调用时如果此时没有僵死进程，则会阻塞，如果有没回收的僵死进程，则立刻返回

21.7 waitpid

```
waitpid(int pid,int * status,options)
```

pid:

status:

options:

options = WUNTRACED|WCONTINUED|WNOHANG WCONTINUED 在 mac 上无效，由停止状态转变为运行态并不会使该系统调用返回

21.8 WIFEXITED

WIFEXITED(status)

status:

status 值应由 WEXITSTATUS(status) 处理，返回退出值

21.9 WIFSIGNALED

WIFSIGNALED(status)

status:

status 值应由 WTERMSIG(status) 处理，返回引起杀死的信号值

21.10 WIFSTOPPED

WIFSTOPPED(status)

status:

status 值应由 WSTOPSIG(status) 处理，返回引起停止的信号值

21.11 WIFCONTINUED

WIFCONTINUED(status)

status:

mac 上无效

21.12 waitid

waitid(**idtype_t**, **pid**, **siginfo_t** *, **options**)

idtype_t:

pid:

*:

options:

options = WEXITED|WSTOPED|WCONTINUED|WNOHANG|WNOWAIT

21.13 wait3

wait3(**status**, **options**, **rusage** *)

status:

options:

*:

21.14 wait4

wait4(**pid**, **status**, **options**, **rusage** *)

pid:

status:

options:

*:

21.15 execve

execve(**pathname**, **char** ** **argv**, **char** ** **env**)

pathname:

argv:

env:

倒数第二位为'v' 代表参数类型为数组，为'l' 则为列表，第一个参数一般设置为命令的文件名, 最后一位为'p' 则会通过路径列表查找文件，最后一位为'e' 允许带环境参数，带环境参数后不会继承原进程环境变量, 如果不带环境参数则继承原进程环境变量

21.16 execle

execle(**pathname**, **char** * **argv**, ..., **char** ** **env**)

pathname:

argv:

...:

env:

21.17 execvp

execvp(**filename**, **char** ** **argv**)

filename:

argv:

21.18 execlp

```
execlp(filename,char * argv,...)
```

filename:

argv:

...:

可以调用自己写的脚本，而且 filename 必须为完整路径

21.19 execv

```
execv(pathname,char ** argv)
```

pathname:

argv:

21.20 execl

```
execl(pathname,char * argv,...)
```

pathname:

argv:

...:

无法调用自己写的脚本

21.21 fexecve

```
fexecve(fd,char ** argv,char ** env)
```

fd:

argv:

env:

22 线程

22.1 pthread_create

```
pthread\_create(pthread\_t * tid,pthread\_attr\_t *,void *(*start)
```

tid:

*:

*(*start:

(void *),void * arg) 类似进程级的 fork，*tid 为线程创建成功后返回的线程 id，线程从 start 函数开始运行，如果有超过一个以上的参数，则可以将这些参数放入某个结构，将结构的地址用 arg 参数传入，新线程会继承调用线程的浮点环境 (文件描述符，环境变量，默认权限掩码等) 和信号屏蔽字，不会继承原线程挂起的信号

22.2 pthread_exit

```
pthread\_exit(void *)
```

*:

退出当前线程，返回值可以由 pthread_join 接收，或者用 return，如果用 exit 的三个函数退出，会直接终止整个进程，当 main 结束时，子线程结束运行

22.3 pthread_self

```
pthread\_self(void)
```

void:

类似进程级的 getpid，获取自身线程 id

22.4 pthread_equal

pthread_equal(pthread_t,pthread_t)

pthread_t:

pthread_t:

判断两个线程 id 是否相等

22.5 pthread_join

pthread_join(pthread_t tid,void * &rval)

tid:

&rval:

类似进程级的 waitpid，阻塞，直到 tid 指定的线程调用 pthread_exit(rval 等于 pthread_exit 的参数值)，通过 return 正常返回 (rval 会等于 tid 线程的返回值)，或者被取消 (rval 会等于 PTHREAD_CANCELED(1))

22.6 pthread_cancel

pthread_cancel(pthread_t)

pthread_t:

取消同一进程内的其他进程，仅仅提出请求，分离的线程也可取消

22.7 pthread_detach

pthread_detach(pthread_t)

pthread_t:

分离指定的线程，分离后线程占用的资源会在终止时立即释放，不能用 pthread_join 获取一个分离线程的退出状态

22.8 pthread_attr_init

pthread_attr_init(pthread_attr_t *)

*:

对属性对象进行初始化

22.9 pthread_attr_destroy

pthread_attr_destroy(pthread_attr_t *)

*:

对属性对象进行反初始化

22.10 pthread_attr_getdetachstate

pthread_attr_getdetachstate(pthread_attr_t *,int * detachstate)

*:

detachstate:

获取属性对象的 detachstat 值

22.11 pthread_attr_setdetachstate

pthread_attr_setdetachstate(pthread_attr_t *,int * detachstate)

*:

detachstate:

detachstate = PTHREAD_CREATE_DETACHED|PTHREAD_CREATE_JOINABLE, 设置 PTHREAD_CREATE_DETACHED 后,线程在运行时就会直接处理分离阶段,设置 PTHREAD_CREATE_JOIN 的线程会正常启动

22.12 pthread_attr_getstack

pthread_attr_getstack(pthread_attr_t *,void ** addr,int * size)

*:

addr:

size:

获取属性对象中线程栈的大小和地址

22.13 pthread_attr_setstack

pthread_attr_setstack(pthread_attr_t *,void * addr,int size)

*:

addr:

size:

设置属性对象中线程栈的大小和地址，地址需要与边界对齐

22.14 pthread_attr_getstacksize

```
pthread\_attr\_getstacksize(pthread\_attr\_t *,int * stacksize)
```

*:

stacksize:

获取属性对象中线程栈的大小

22.15 pthread_attr_setstacksize

```
pthread\_attr\_setstacksize(pthread\_attr\_t *,int stacksize)
```

*:

stacksize:

设置属性对象中线程栈的大小，不需要处理分配地址

22.16 pthread_attr_getguardsize

```
pthread\_attr\_getguardsize(pthread\_attr\_t *,int * guardsize)
```

*:

guardsize:

获取线程栈末尾之后避免栈溢出的拓展内存大小

22.17 pthread_attr_setguardsize

```
pthread\_attr\_setguardsize(pthread\_attr\_t *,int guardsize)
```

*:

guardsize:

设置线程栈末尾之后避免栈溢出的拓展内存大小，如果设置为 0，则取消此特征

22.18 pthread_mutex_init

pthread_mutex_init(pthread_mutex_t *,pthread_mutexattr_t *)

*:

*:

动态分配互斥量

22.19 pthread_mutex_destroy

pthread_mutex_destroy(pthread_mutex_t *)

*:

释放动态互斥量的内存

22.20 pthread_mutex_lock

pthread_mutex_lock(pthread_mutex_t *)

*:

对互斥量进行加锁，如果互斥量已经上锁，调用线程将阻塞直到互斥量被解锁

22.21 pthread_mutex_trylock

pthread_mutex_trylock(pthread_mutex_t *)

*:

如果互斥量已经上锁，不会阻塞，会直接返回失败，并且设置 EBUSY 错误码

22.22 pthread_mutex_unlock

pthread_mutex_unlock(pthread_mutex_t *)

*:

对互斥量解锁

22.23 pthread_mutex_timedlock

pthread_mutex_timedlock(pthread_mutex_t *,timespec *)

*:

*:

mac os 没有实现此函数，请求加锁，如果阻塞，则等待一定时间，如果在时间内未成功开锁，则返回错误

22.24 pthread_mutexattr_init

pthread_mutexattr_init(pthread_mutexattr_t *)

*:

初始化互斥量属性

22.25 pthread_mutexattr_destroy

pthread_mutexattr_destroy(pthread_mutexattr_t *)

*:

反初始化

22.26 pthread_mutexattr_getpshared

pthread_mutexattr_getpshared(pthread_mutexattr_t *,int * attr)

*:

attr:

获取互斥量进程共享属性

22.27 pthread_mutexattr_setpshared

pthread_mutexattr_setpshared(pthread_mutexattr_t *,int attr)

*:

attr:

attr = PTHREAD_PROCESS_PRIVATE|PTHREAD_PROCESS_SHARED PRIVATE 为默认行为，多个线程可以访问同一个互斥量，SHARED 为进程可以访问同一个互斥量

22.28 pthread_mutexattr_gettype

pthread_mutexattr_gettype(pthread_mutexattr_t *,int * type)

*:

type:

获取线程的互斥量锁定特性

22.29 pthread_mutexattr_settype

pthread_mutexattr_settype(pthread_mutexattr_t *, flag)

*:

flag:

flag = PTHREAD_MUTEX_NORMAL|PTHREAD_MUTEX_ERRORCHECK|PTHREAD_MUTEX_RECU

NORMAL 一种标准互斥量类型，不进行错误检查或死锁检测，ERRORCHECK 为互斥量提供死锁检测，RECURSIVE 运行同一线程在互斥量解锁之前对该互斥量多次加锁，在解锁状态和加锁状态不相同的情况下，不会释放该锁

22.30 pthread_rwlock_init

pthread_rwlock_init(pthread_rwlock_t *,pthread_rwlockattr_t *)

*:

*:

对读写锁进行初始化，如需要默认属性可以传递 NULL 给第二个参数

22.31 pthread_rwlock_destroy

pthread_rwlock_destroy(pthread_rwlock_t *)

*:

在使用完读写锁后需要进行释放

22.32 pthread_rwlock_rdlock

pthread_rwlock_rdlock(pthread_rwlock_t *)

*:

使用读模式锁定读写锁，可对一个锁同时上多个读锁，可能会因为系统实现而对读写锁有次数限制，所以应进行返回值检查

22.33 pthread_rwlock_wrlock

pthread_rwlock_wrlock(pthread_rwlock_t *)

*:

使用写模式锁定读写锁，一个锁上只能有一个写锁，申请上其他锁时都会产生阻塞

22.34 pthread_rwlock_unlock

pthread_rwlock_unlock(pthread_rwlock_t *)

*:

解锁读写锁

22.35 pthread_rwlock_tryrdlock

pthread_rwlock_tryrdlock(pthread_rwlock_t *)

*:

获取读锁成功时，返回 0，否则返回 EBUSY

22.36 pthread_rwlock_trywrlock

pthread_rwlock_trywrlock(pthread_rwlock_t *)

*:

获取写锁成功时，返回 0，否则返回 EBUSY

22.37 pthread_rwlockattr_getpshared

```
pthread\_rwlockattr\_getpshared(pthread\_rwlockattr\_t *,int * attr)
```

*:

attr:

获取读写锁进程共享属性

22.38 pthread_rwlockattr_setpshared

```
pthread\_rwlockattr\_setpshared(pthread\_rwlockattr\_t *,int attr)
```

*:

attr:

设置读写锁进程共享属性

22.39 pthread_cond_init

```
pthread\_cond\_init(pthread\_cond\_t *,pthread\_condattr\_t *)
```

*:

*:

动态分配条件变量

22.40 pthread_cond_destroy

```
pthread\_cond\_destroy(pthread\_cond\_t *)
```

*:

释放条件变量所在的内存空间前对条件变量进行反初始化

22.41 pthread_cond_wait

pthread_cond_wait(pthread_cond_t *,pthread_mutex_t *)

*:

*:

选定某个已上锁的互斥量,然后阻塞并等待条件变量变为真 (即等待其他线程运行 pthread_cond_signal 或者 pthread_cond_broadcast),运行期间会释放互斥锁,当满足条件返回时 (即被 pthread_cond_signal 或者 pthread_cond_broadcast 取消阻塞后) 会再次申请上锁，因为当它阻塞时释放了互斥锁

22.42 pthread_cond_timedwait

pthread_cond_timedwait(pthread_cond_t *,pthread_mutex_t *,timespec *)

*:

*:

*:

指定所需要等待的时间，当超出时间后还未满足条件则返回错误码，并且不会将释放掉的互斥锁再次上锁，这里的 timespec 是当前时间加成等待时间

22.43 pthread_cond_signal

pthread_cond_signal(pthread_cond_t *)

*:

将条件变为真，并至少唤醒一个等待该条件的线程 (即那些用 pthread_cond_wait 将 cond 绑定到互斥量的线程)

22.44 pthread_cond_broadcast

```
pthread\_cond\_broadcast(pthread\_cond\_t *)
```

*:

唤醒所有等待该条件的线程

22.45 pthread_once

```
pthread\_once(pthread\_once\_t *,void (*init)
```

*:

(*init:

(void))

22.46 pthread_condattr_getpshared

```
pthread\_condattr\_getpshared(pthread\_condattr\_t *,int * attr)
```

*:

attr:

获取条件变量的的进程同步属性

22.47 pthread_condattr_setpshared

```
pthread\_condattr\_setpshared(pthread\_condattr\_t *,int attr)
```

*:

attr:

控制着条件变量是可以被单个进程的多个线程使用，还是可以被多进程的线程使用

22.48 pthread_key_create

pthread_key_create(pthread_key_t *,void (*destructor)

*:

(*destructor:

(void *)) 创建一个键，用于获取对线程特定数据的访问

22.49 pthread_setspecific

pthread_setspecific(pthread_key_t,void *)

pthread_key_t:

*:

22.50 pthread_getspecific

pthread_getspecific(pthread_key_t)

pthread_key_t:

22.51 pthread_cleanup_push

pthread_cleanup_push(void (*func)

(*func:

(void *),void * arg) 设置线程退出处理程序，等同于进程的 atexit，但如果线程正常退出则不会调用 (return)

22.52 pthread_cleanup_pop

pthread_cleanup_pop(0|!0)

0|!0:

如果为 0，线程退出处理函数将不会被调用，如果非 0 则立即调用，mac 上这两个函数用宏实现，如果使用最好同时调用

22.53 pthread_setcancelstate

pthread_setcancelstate(**state**,**oldstate**)

state:

oldstate:

type = PTHREAD_CANCEL_DISABLE|PTHREAD_CANCEL_ENABLE 设置可取消状态，默认状态为 ENABLE，如果在 ENABLE 状态时接收到了取消请求，挂起请求，在到达取消点时线程会取消，如果为 DISABLE 状态，收到的取消请求会挂起，直到由 DISABLE 变为 ENABLE 时，才会在下一个取消点处理取消请求

22.54 pthread_setcanceltype

pthread_setcanceltype(**type**,**oldtype**)

type:

oldtype:

type = PTHREAD_CANCEL_ASYNCHRONOUS|PTHREAD_CANCEL_DEFERRED 设置可取消类型，type 分别为异步取消和推迟取消，默认为推迟取消，则需要到达取消点时才可取消，而设置异步取消时，线程可以在任意时间取消

22.55 pthread_testcancel

```
pthread\_testcancel(void)
```

void:

设置取消点，到达此函数时，如果有挂起的取消请求，并且取消状态不为 DISABLE，那么线程会被取消，否则此函数无效

22.56 pthread_sigmask

```
pthread\_sigmask(int how,sigset\_t * new,sigset\_t * old)
```

how:

new:

old:

线程级的 sigprocmask，失败时返回错误码，而不是设置 errno 并返回-1

22.57 sigwait

```
sigwait(sigset\_t * set,int * signop)
```

set:

signop:

先解除信号的阻塞状态，如果 set 中的信号集包含有被阻塞的信号，移除那些被阻塞的信号，函数立刻返回，否则阻塞直到收到集合中的信号 (无论信号是否被阻塞)，此函数返回后不会改变原来的信号掩码

22.58 pthread_kill

pthread_kill(pthread_t, signo)

pthread_t:

signo:

线程级的 kill，可以通过发送 0 查看线程是否存在，如果信号的默认处理动作是终止该进程，那么发送到任意一个线程都会终止整个进程

22.59 线程与 fork：当线程调用 fork 时，就为子进程创建了整个地址空间的副本，在子进程内部，只存在一个进程，它是由父进程中调用 fork 的线程的副本构成的，由于写时复制的原因，除非是 fork 后立即调用 exec，否则父进程和子进程还可以共享内存页，如果父进程中的线程占有锁，那么子进程也将同样占有这些锁，可是子线程并不包含那些占有锁线程的副本

线程与fork：当线程调用fork时，就为子进程创建了整个地址空间的副本，在子进程内部，只存在

包含线程代码的副本，但是这些线程代码并不会自动运行:

，所以子进程没有办法知道它占有了哪些锁，需要释放哪些锁

22.60 pthread_atfork

pthread_atfork(void (*prepare)

(*prepare:

(void),void(*parent)(void),void(*child)()) 锁清理函数，在线程 fork 时进行锁清理，prepare 用于在调用 fork 前获取父进程定义的所有锁，parent 用于在 fork 生成子进程后返回前释放父进程中 prepare 中获取的所有锁，child 函数同 parent 函数一样，不过是作用于子线程中

23 会话组

23.1 getpgrp

getpgrp()

获取调用进程的进程组 id

23.2 getpgid

getpgid(**pid**)

pid:

获取进程 id 为 pid 的进程组 id，如果 pid 为 0，则为调用进程

23.3 setpgrp

setpgrp()

将调用进程的进程组 id 设置为调用进程的进程 id

23.4 setpgid

setpgid(**pid**,**pgid**)

pid:

pgid:

如果 pid 为 0，则等价于 getpid()，如果 pgid 为 0，也等于 getpid()

23.5 getsid

getsid()

获取进程的会话 id

23.6 setsid

setsid(**pid**)

pid:

创建一个新会话

23.7 tcgetpgrp

tcgetpgrp(**fd**)

fd:

fd 为终端关联的文件描述符，返回前台进程组 id

23.8 tcsetpgrp

tcsetpgrp(**fd**, **pid**)

fd:

pid:

将 fd 关联终端的前台组 id 设置为会话中的另一个进程组 id

23.9 tcgetsid

tcgetsid(**fd**)

fd:

获取会话首进程的进程组 id

23.10 getpriority

getpriority(**which**,**who**)

which:

who:

which = PRIO_PROCESS|PRIO_PGRP|PRIO_USER

23.11 setpriority

setpriority(**which**,**who**,**value**)

which:

who:

value:

23.12 setutxent

setutxent(**void**)

void:

23.13 endutxent

endutxent(**void**)

void:

23.14 utmpxname

utmpxname(**filepath**)

filepath:

23.15 getutxent

getutxent(**void**)

void:

23.16 getutxid

getutxid(**utpmx** *)

*:

23.17 getutxline

getutxline(**utmpx** *)

*:

23.18 pututxline

pututxline(**utmpx** *)

*:

24 管道

24.1 pipe

pipe(**int** [2])

[2]:

如果某管道的写入端未关闭，且当前管道内无数据，此时进行读取会阻塞；即管道的写入端如果已关闭，此时进行读取且管道内无数据会直接返回 0，如果写一个读端已经关闭的管道，则产生信号 SIGPIPE，如果选择忽略此信号，则 write 函数返回-1，并且设置 errno 为 EPIPE，fork 会复制 pipe 产生的文件描述符，历史上，该管道是半双工的 (即同一时刻只能有一端发送，一端接受)，mac 上目前还是半双工的，某些系统支持全双工管道

24.2 popen

popen(**char** * **cmd**, **char** * **mode**)

cmd:

mode:

本质上是先创建一个 pipe，然后调用 fork，子进程调用 exec 运行 cmd，因为 cmd 命令有可能需要输入数据，所以 mode 可能是"r" 或者是"w"，如果返回的文件指针是可读的，那么使用"r"，如果使用的文件指针是可写的，那么使用"w"

24.3 pclose

pclose(FILE *)

*:

若成功则返回 cmd 的退出状态，否则返回-1

24.4 mkfifo

mkfifo(char * pathname,mode_t mode)

pathname:

mode:

命名管道，即该管道实际上为一文件，程序需要打开该文件进行通信，一端以只读方式打开，另一端以只写方式打开，先打开的一端会阻塞，直到另一端打开，或者设置 O_NONBLOCK 以非阻塞方式打开，设置 O_NONBLOCK 后需要先打开读取端，当以非阻塞方式打开管道后，如果写端已打开，但 read 时但无数据读取，则返回-1，如果写端已关闭，则返回 0，如果读取端已关闭，进行 write 操作时会触发 sigpipe 信号，对阻塞方式打开的读写管道即使另一端已关闭进行操作时也会阻塞

24.5

XSL IPC: 每个内核中的IPC结构都用一个非负整数的标示符加以引用，例如要向一个消息队列发送

uid(拥有者 id:

; gid_t gid; uid_t cuid(创建者 id); gid_t cgid; mode_t mode

24.6 ftok

```
ftok(char * pathname,int id)
```

pathname:

id:

使用路径名和一个项目 id 产生一个键

25 消息队列

25.1 msgget

```
msgget(key,flags)
```

key:

flags:

flags = IPC_CREAT|IPC_EXCL|0755 key |= IPC_PRIVATE 注意：创建时一定要指定权限，消息队列已经很少使用了，新程序尽量不要使用它，IPC_CREAT 创建一个新的消息队列或者打开一个现有队列，IPC_CREAT|IPC_EXCL 若已存在对应的消息队列，则退出，否则创建，返回一个消息队列 id

25.2 msgsnd

```
msgsnd(int msqid,void *,msgsize,flags)
```

msqid:

*:

msgsize:

flags:

flagss = IPC_NOWAIT 将新消息添加到队列尾端，发送的消息类型不能为 0，第三个参数为除了 type 项之外的数据大小之和，mac 上的管道容量为 2048 个字节

25.3 msgrcv

msgrcv(**int** **msqid**, **void** *, **maxmsgsize**, **msgtype**, **flags**)

msqid:

*:

maxmsgsize:

msgtype:

flags:

flags = IPC_NOWAIT|MSG_NOERROR IPC_NOWAIT 如果没有消息可读，则直接返回-1，MSG_NOERROR 用于当 maxsize 参数小于接收到的消息长度时，截断超过 maxsize 长度后的数据，如果不指定，返回-1，并且消息仍然留在队列当中，msgtype==0 则接受队列中第一条消息，大于 0 则接受队列中消息类型等于 msgtype 的消息，返回值为类似于 read，等于接收到数据的字节数

25.4 msgctl

msgctl(**int** **msqid**, **cmd**, **msqid_ds** *)

msqid:

cmd:

*:

cmd = IPC_RMID|IPC_STAT|IPC_SET 分别为删除消息队列以及其中的数据，获取 msqid 对应的 msqid_ds 属性，设置 msqid 对应的 msqid_ds 属性，msqid_dsipc_perm msg_perm; msgqnum_t msg_qnum(剩余消息数量); msglen_t msg_qbytes(队列容量);msglen_t cbytes(当前队列存在的数据量); pid_t msg_lspid(最后发送消息进程的 pid); pid_t msg_lrpid(最后接受消息进程的 pid); time_t msg_stime(最后发送消息的时间); time_t msg_rtime(最后接受消息的时间); time_t msg_ctime(队列最后改变的时间)

26 信号量

26.1 semget

semget(**key**, **nsems**, **flags**)

key:

nsems:

flags:

flags = IPC_CREAT|IPC_EXCL|O755 nsems 是该集合中的信号量数，如果是创建新集合，就必须指定 nsems，否则将其指定为 0，表示引用一个已经存在的集合，注意：创建时需要指定权限，信号量使用一个未命名结构体 structunsigned short semval(信号量的值); pid_t sempid(最后操作此信号量的 pid); semncnt(等待此信号量的值大于针对此信号调用 semop 时所指定 sem_op 绝对值的进程数量，可以直接理解为阻塞在该信号量的数量); semzcnt(等待此信号量变为 0 的进程数量)

26.2 semop

semop(**semid**, **sembuf** ∗, **flags**)

semid:

*:

flags:

sembufunsigned short sem_num(指定信号),short sem_op(进行的操作),short sem_flg sem_flg = IPC_NOWAIT|SEM_UNDO SEM_UNDO 用于如果某进程占用了信号量的资源，但是当它结束时，进程占用的信号量值并不会释放，指定 SEM_UNDO 可以解决这个问题，当进程结束时，将其占用的信号量恢复若 sem_op 为正值，则将此值加到对应的信号量上，若 sem_op 为负值，则表示要获取由该信号量控制的资源，如果该信号量的值大于等于 sem_op 的绝对值，则直接从信号量值中减去，否则，若指定了 IPC_NOWAIT，则直接出错返回 EAGIN，若没有指定，则该信号量的 semncnt 值 +1，然后调用进程被挂起知道以下行为发生，此信号量的值变成大于了 sem_op

的绝对值，则从该信号量值减去 sem_op 的绝对值，然后继续运行，收到信号，并从信号处理程序返回，semncnt 减 1，函数出错并设置 EINTR，或者此信号量被删除，出错返回 EIDRM，若 sem_op 等于 0，则表示调用进程希望等待该信号量变为 0，具体情况类似于 sem_op 小于 0，当该操作阻塞时，即减少信号量值导致信号量小于 0 时，此时如果被信号中断，该操作不会自动重启，

26.3 semctl

semctl(**semid**,**semnum**,**cmd**,. . . **union semun**)

semid:

semnum:

cmd:

semun:

注意：联合体参数不是指针类型，semnum 表示第几个信号量，从 0 开始，部分 cmd 操作对此参数没有要求，cmd = IPC_RMID|IPC_STAT|IPC_SET|GETVAL|SETVAL|GETALL|GETPIC|GETNCNT|GETZCN semunint val(用于 SETVAL); semid_ds * buf(用于 IPC_STAT 以及 IPC _SET); unsigned short * array(用于 GETALL 以及 SETALL) IPC_RMID 用于删除信号量集，IPC_STAT 用于获取获取该信号量集关联的数据结构，IPC_SET 用 semun->semid_ds 的属性更新该信号量集关联的数据结构，GETVAL 返回由 semid 指定的第 semnum 个信号量的值，SETVAL 将第 semnum 个信号量设置为 semnu->val，GETALL 将信号量集中的值设置为 semun->array[]，SETALL 为设置，semid_dssem_perm(权限信息); sem_nsems(信号量的个数); semds.sem_otime(最后一次 op 操作的时间); semds.sem_ctime(最后一次修改时间)

27 共享内存

27.1 shmget

shmget(**key**, **size** , **flags**)

key:

size:

flags:

flags = IPC_CREAT|IPC_EXCL|0755 当创建一个新段时，size 指定需要的大小，当引用一个已经存在的段时，则指定为 0

27.2 shmat

shmat(**shmid**, **addr**, **flags**)

shmid:

addr:

flags:

flags = SHM_RND|SHM_RDONLY 分别为将 addr 的值自动四舍五入到页面大小的倍数，将内存块以只读方式装载到调用进程的虚拟内存，如果 addr 为 0 则不需要用 SHM_RND，系统会自动将共享内存块映射到可用的地址上

27.3 shmdt

shmdt(**addr**)

addr:

接触对 addr 开始的内存共享段的映射，并将共享内存段的引用计数减一

27.4 shmctl

shmctl(**shmid**, **cmd**, **shmid_ds** *)

shmid:

cmd:

*:

cmd = IPC_RMID|IPC_STAT|IPC_SET IPC_RMID 用于删除此共享内存段，标示符会立即删除，所以不能再用 shmat 进行该段的连接，但是该内存段不会立即删除，只有当引用此共享内存段的计数变为 0 后才会真正删除该段，IPC_STAT 用于获取此段的属性，IPC_SET 为设置，shmid_dsshm_perm(权限设置) shm_segsz(共享存储的段大小); shm_lpid(最后进行 op 操作的 pid); shm_cpid(创建者的 pid?); shm_nattch(当前共享此区域的进程数量); shm_atime(最后一次访问的时间); shm_dtime(最后一次分离此内存段的时间); shm_ctime(最后一次改变的时间)

27.5 mmap

mmap(**addr**, **length**, **prot**, **flags**, **fd**|−1, **offset**)

addr:

length:

prot:

flags:

fd|-1:

offset:

prot = PROT_NONE|PROT_READ|PROT_WRITE|PROT_EXEC PROT_NONE 表示映射区不可访问,PROT_READ 表示映射区可读,PROT_READ 表示映射区可写,PROT_EXEC 表示映射区可执行，如果要写文件，则应该设置 PROT_READ|PROT_WRITR 并且打开文件时应该指定标记 O_RDWR，映射文件时 size 不能超过文件的大小 (可用 lseek 加 write 或者 ftruncate 增加文件大小) flags = MAP_PRIVATE|MAP_SHARED|MAP_ANONYMOUS|MAP_FIXED|MAP_NORESE PROT_NONE 表示区域无法访问，MAP_PRIVATE 表示创建私人映射，会创建一份副本，对数据的改变不会影响源文件，MAP_SHARED 表示创建共享映射，存储操作等于对文件调用 write，MAP_ANONYMOUS 表示创建匿名映射，私人匿名映射类似堆分配 (但是没有堆分配时块与块

之间的联系)，共享匿名映射就是共享内存分配，MAP_FIXED 表示不对 addr 参数进行处理，否则会将 addr 参数向上取整为分页大小的倍数，此时会对 addr 地址强行进行映射，还能覆盖该地址之前的映射

27.6 munmap

munmap(**addr**, **length**)

addr:

length:

解除映射区，如果是私人映射，那么映射区的数据会被丢弃

27.7 msync

msync(**addr**, **length**, **flags**)

addr:

length:

flags:

将页写回硬盘，flags = MS_SYNC|MS_ASYNC|MS_INVALIDATE 分别为同步更新，异步更新，通知系统丢弃那些与底层存储器没有同步的页

27.8 mprotect

mprotect(**addr**, **length**, **flags**)

addr:

length:

flags:

flags = prot = PROT_NONE|PROT_READ|PROT_WRITE|PROT_EXEC 用于更改保护位，addr 必须是系统页长的整数倍

28 内存锁

28.1 mlock

mlock(addr, length)

addr:

length:

当调用完成后，即使映射的地址区域当前不在区域内，也会在该函数返回前将该区域换进内存，而不需要等待发生缺页

28.2 munlock

munlock(addr, length)

addr:

length:

解锁以页为单位，当对同一页进行多次上锁也只会产生一次效果，某页的上锁属性应该保存在进程的该页的映射数据结构中，如果多个进程共享映射同一组分页时，只要还存在一个进程持有这些分页上的内存锁，那么这些分页就会保持被锁进内存的状态

28.3 mlockall

mlockall(flags)

flags:

flags = MCL_CURRENT|MCL_FUTURE MCL_CURRENT 将进程的虚拟内存中当前所有映射的分页全部锁进内存，MCL_FUTURE 将后续映射到虚拟内存中的所有分页锁进内存

28.4 munlockall

```
munlockall(void)
```

void:

28.5 mincore

```
mincore(addr,length,char vec[])
```

addr:

length:

vec[]:

无论是产生何种映射，包括私人匿名映射 (堆分配)，并不会立即为这些映射分配相应的内存，需要访问相应的虚拟内存产生缺页错误后才会进行分配

28.6 madvise

```
madvise(addr,length,flags)
```

addr:

length:

flags:

flags = MADV_NORMAL|MADV_RANDOM|MADV_SEQUENTIAL|MADV_WILLNEED|MADV_DONTNE

29 记录锁

29.1 flock

flock(fd, flags)

fd:

flags:

flags = LOCK_SH|LOCK_EX|LOCK_UN|LOCK_NB LOCK_SH 为设置共享锁,LOCK_EX 为设置互斥锁，LOCK_UN 为解锁，LOCK_NB 为执行非阻塞操作，无论对文件的访问模式是只读，只写或是读写都可以在上面放置共享锁和互斥锁，该函数的操作单位为整个文件，并且 flock 的锁转换非原子操作，它是先解锁，然后上锁，在解锁和上锁之间可能会有其他进程的上锁请求成功执行，此时该函数会阻塞，并且原本拥有的锁丢失

29.2 fcntl

fcntl(fd,cmd,flock *)

fd:

cmd:

*:

cmd = F_SETLK|F_SETLKW|F_GETLK 分别为设置锁，非阻塞操作设置锁，检测锁，flockl_type,l_whence,l_start,l_len,l_pid l_type = F_RDLCK|F_WRLCK|F_UNLCK 分别为设置读锁 (共享锁)，写锁 (互斥锁)，解锁，该函数放置锁需要与文件的打开模式相对应，即需要放置两种锁时，文件的打开模式应该为 O_RDWR l_start = SEEK_SET|SEEK_CUR|SEEK_END l_whence 为偏移量 l_len 为长度，，当 len 为 0 时，表示锁的范围可以拓展到最大可能偏移量 (无论此后追加写入了多少数据)，l_pid 当 cmd 为 F_GETLK 时有效，返回拥有该锁的进程 id，单个进程在某一时刻只能对一个文件区间拥有一把锁。多次加锁会覆盖上个锁

30 套接字

30.1 socket

socket(**domain**,**type**,**protocal**)

domain:

type:

protocal:

domain = AF_UNIX|AF_INET|AF_INET6 分别为 UNIX 域，Ipv4 因特网域，Ipv6 因特网域 type = SOCK_STREAM|SOCK_DGRAM|SOCK_SEQPACKET|SOCK_RAW 分别为流 (TCP), 报文 (UDP), 可靠传输的 UDP,IP 协议的数据报接口 protocal 通常为 0,INADDR_LOOPBACK(0x7f000001) 为 IPV4 回环地址,INADDR_ANY(0x0) 为 IPV4 通配地址,均为整形数据,IN6ADDR_LOOPBACK_INIT 为 IPV6 回环地址，IN6ADDR_ANY_INIT 为 IPV6 通配地址，为结构体类型

30.2 bind

bind(**int** **sockfd**,**sockaddr** ∗ **addr**,**int** **addrlen**)

sockfd:

addr:

addrlen:

sockaddr = sockaddr_un(AF_UNIX)|sockaddr_in(AF_INET)|sockaddr_in6(AF_INET6) addrlen 要根据使用的 sockaddr 来确定，不能使用 sizeof(struct sockaddr)，sockaddrunsigned char sa_len;sa_family_t sa_family; char sa_data[14] sockaddr_ununsigned char sun_len; sa_family_t sun_family; char sun_path[104](用于创建套接字的文件名，该文件仅用于向客户客户进程告示套接字名字,无法打开,也不能由应用程序进行通讯) 当 sun_path 指定的文件已存在时,bind 会失败,也就是说该文件是一次性的，程序结束时就应该删除该文件，每次 bind 时都要保证该文件不存在，sockaddr_inunsigned char sin_len; sa_family_t sin_family; in_port_t sin_port; struct in_addr sin_addr; unsigned char sin_zero[8] struct sin_addrin_addr_t(无符号 32 位整形) s_addr

30.3 listen

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

sockfd:

backlog:

将 sockfd 指定为监听套接字，此后此套接字能接收到连接请求，backlog 用于限制发起请求连接的数量，一旦未处理连接等于 backlog，系统就会拒绝多余的连接请求

30.4 accept

```
accept(int sockfd,sockaddr * addr,&addrlen)
```

sockfd:

addr:

&addrlen:

获得 sockfd 监听的连接请求并建立连接，返回一个套接字描述符，此描述符连接到客户端调用 connect 的进程，并将请求连接端的地址信息写入 addr 中，len 参数为缓冲区的大小，函数返回时，会将 len 改为向缓冲区写入的字节数，如果不关心对端机器的地址信息，可以将 addr 和 len 置为 NULL，如果 sockfd 是非阻塞且当前没有连接请求，accept 会退出并返回-1，否则将阻塞直到收到一个连接请求 (阻塞模式)

30.5 connect

```
connect(sockfd,sockaddr *,addrlen)
```

sockfd:

*:

addrlen:

如果要处理一个面向连接的网络服务 (SOCK_STREAM 或 SOCK_SEQPACKET)，那么在开始交换数据之前，需要在请求服务的进程套接字和提供服务的进程套接字之间建立一个连接，如果 sockfd 没有绑定到一个地址，connect 会给调用者绑定一个默认地址，如果 connect 失败，在部分系统上套接字会变成未定义的，最好是关闭套接字，新建一个套接字后再进行 connect 操作，当在一个数据报 socket 上使用 connect 后，可以使用 read 和 write 操作描述符

30.6 shutdown

shutdown(sockfd, flags)

sockfd:

flags:

flags = SHUT_RD|SHUT_WR|SHUT_RDWR SHUT_RD 为关闭读端，那么无法从套接字读取数据，SHUT_WR 为关闭写端，表示无法用套接字发送数据，SHUT_RDWR 则既无法读取也无法发送，由于套接字的 close 命令并不一定能直接关闭 socket(比如通过 dup 复制了描述符) 所以使用 shutdown 可以避免这个问题，而且使用 shutdown 能够使用单向通讯

30.7 send

```
send(int sockfd,void * buf,size\_t len,int flags)
```

sockfd:

buf:

len:

flags:

使用时套接字必须已经连接，类似于 write，但可以指定标志来改变处理传输数据的方式 flags = MSG_DONTWAIT，MSG_DONTWAIT 使用非阻塞操作

30.8 recv

```
recv(int sockfd,void * buf,size\_t len,int flags)
```

sockfd:

buf:

len:

flags:

flags = MSG_DONTWAIT|MSG_OOB|MSG_PEEK|MSG_WAITALL MSG_DONTWAIT 此次调用不会阻塞，MSG_PEEK 获取 sockfd 缓冲区中数据的一份副本，不会将数据从缓冲区移除，MSG_WAITALL 直到等待接受到 len 个字节后才会返回

30.9 sendto

```
sendto(int sockfd,void * buf,size\_t length,int flags,sockaddr * addr,addrlen)
```

sockfd:

buf:

length:

flags:

addr:

addrlen:

可用于发送报文，通过 addr 指定目标地址，如过 sockfd 有连接，那么无视 addr

30.10 recvfrom

```
recvfrom(int sockfd,void buf,size\_t length,int flags,sockaddr * addr,&addrlen)
```

sockfd:

buf:

length:

flags:

addr:

&addrlen:

带有获取发送者信息功能的 recv，将发送者的地址信息写入 addr，addrlen 表示缓冲区的大小，当函数返回时将 len 改为向缓冲区写入的字节数

30.11 socketpair

socketpair(**domain**, **type**, **protocal**, **int** [2])

domain:

type:

protocal:

[2]:

前三个参数类似 socket，第四个参数类似于 pipe，生成两个连接着的 unix 域的 socket 套接字，domain = AF_UNIX|AF_INET|AF_INET6 虽然结构足够通用，允许 socketpair 用于其他域，但一般来说操作系统仅对 unix 域提供支持，type = SOCK_STREAM|SOCK_DGRAM 分别为字节流和报文，unix 域的数据报是可靠的，既不会丢失报文也不会传递出错，unix 域套接字更像是套接字和管道的结合，一对相互连接的套接字可以起到全双工管道的作用，两端对读和写开放，由于创建的套接字没有名字，所以不能在无关进程中使用，如需要不同进程间通讯需要使用 socket 函数

30.12 htons

htons(**short**)

short:

h 代表主机 (host)，n 代表网络 (network)，l 代表 32 位，s 代表 16 位，表示在主机字节序与网络字节序之间进行转换

30.13 htonl

htonl(**int**)

int:

30.14 ntohs

ntohs(**short**)

short:

30.15 ntohl

ntohl(**int**)

int:

30.16 inet_aton

inet_aton(**in_addr**)

in_addr:

只能用于 IPv4，已过时

30.17 inet_ntoa

inet_ntoa(char *,in_addr *)

*:

*:

只能用于 IPv4，已过时

30.18 inet_pton

inet_pton(int domain,char * str ,in_addr|in6_addr *)

domain:

str:

*:

domain = AF_INET|AF_INET6 由表现形式转换成网络形式，即点分十进制字符串转换成二进制数字，注意：此函数转换时会考虑本机的大小端特性

30.19 inet_ntop

inet_ntop(int domain,in_addr|in6_addr *,char * str ,socklen_t addrlen)

domain:

*:

str:

addrlen:

->(char *) domain = AF_INET|AF_INET6 有网络形式转换成表现形式，即整形数字转换成点分十进制字符串，注意：此函数将参数视为网络字节序转换成字符串，所以对于小端法机器，如果你想要提供自己的参数给它，可以先使用 htonl 再进行传递

30.20 sethostent

sethostent(**int** **statopen**)

statopen:

打开文件网络配置信息文件，如果已打开文件，会将读取的偏移量置为 0，如果 statopen 非 0，调用 gethostent 后文件仍然保持打开状态，mac 上打开的是/etc/hosts 文件，返回的地址为网络字节序

30.21 gethostent

gethostent(**void**)

void:

->(struct hostent *) 返回文件中的下一个条目 hostentint h_addrtype;char * h_name;char ** h_addr_list;char ** h_aliases;int h_length

30.22 endhostent

endhostent(**void**)

void:

关闭网络配置信息文件

30.23 getnetbyaddr

getnetbyaddr(**uint32_t** **net**,**int** **type**)

net:

type:

->(struct netent *) netentchar * n_name; char ** n_aliases; int n_addrtype; uint32_t n_net(网络序) 以下五个函数应该是针对本机上 ip 地址名和 ip 地址，如 LOOPBACK 和 7f(这里是网络序)

30.24 getnetbyname

getnetbyname(**char** * **name**)

name:

->(struct netent *)

30.25 setnentent

setnentent(**int** **stayopen**)

stayopen:

30.26 getnetent

getnetent()

->(struct netent *)

30.27 endnetent

endnetent(**void**)

void:

30.28 getprotobyname

```
getprotobyname(char * name)
```

name:

->(struct protoent *) protoentchar * p_name; char ** p_aliases; int p_proto 根据协议名获取协议相关信息，如参数为"ip"

30.29 getprotobynumber

```
getprotobynumber(int proto)
```

proto:

->(struct protoent *) 根据协议号获取协议相关信息

30.30 setprotoent

```
setprotoent(int stayopen)
```

stayopen:

打开网络协议和网络号信息文件

30.31 getprotoent

```
getprotoent()
```

->(struct protoent *) 获取文件下一条目

30.32 endprotoent

endprotoent()

关闭文件

30.33 getservbyname

getservbyname(**char** * **name**,**char** * **proto**)

name:

proto:

->(struct servent *) servent(char * s_name; char ** s_aliases; int s_port(网络序); char * s_proto) proto 表示服务名，proto 表示协议名，根据服务名 (如 ssh) 和协议名 (如 tcp) 查询信息

30.34 getservbyport

getservbyport(**int** **port**,**char** * **proto**)

port:

proto:

->(struct servent *) port 表示端口名，proto 表示协议名，根据端口名 (如 23，需要使用网络序) 和协议名 (tcp) 查询信息

30.35 setservent

setservent(**int** **stayopen**)

stayopen:

打开端口绑定的服务名和端口号信息文件，mac 上即/etc/services 文件

30.36 getservent

```
getservent()
```

->(struct servent *) 获取文件下一条目

30.37 endservent

```
endservent()
```

关闭文件

30.38 getaddrinfo

```
getaddrinfo(char * host,char * service,addrinfo * hint,addrinfo ** res)
```

host:

service:

hint:

res:

需要提供主机名，服务名或者两者都提供，如果仅仅提供一个，另一个必须是一个空指针，主机名可以是一个节点名或者点分形式，addrinfoint ai_flags; int ai_family; int ai_socktype; int ai_protocol; int ai_addrlen; int ai_canonname; sockaddr * ai_addr,addrinfo * ai_next ai_family = AF_INET|AF_INET6|AF_UNSPEC 意义为获取哪种地址结构 ai_flags = AI_ADDRCONFIG|AI_ALL|AI_NUM AI_ADDRCONFIG 表示查询配置的地址类型，AI_ALL 表示查找 IPC4 和 IPV6(IPV6 需要指定 AI_V4MAPPED)，AI_NUMERICHOST 表示以数字格式指定主机地址，AI_NUMERICSERV 表示以数字形式 (端口号) 指定服务

30.39 freeaddrinfo

```
freeaddrinfo(addrinfo *)
```

*:

一般用于释放 getaddrinfo 第四个参数指向的 addrinfo 结构

30.40 gai_strerror

```
gai\_strerror(int error)
```

error:

如果 getaddrinfo 失败，使用此函数将 getaddrinfo 的返回值转换成错误信息

30.41 getnameinfo

```
getnameinfo(sockaddr *,addrlen,char * host,hostlen,char * service,servlen,flags)
```

*:

addrlen:

host:

hostlen:

service:

servlen:

flags:

flags = NI_DGRAM|NI_NAMEREQD|NI_NOFQDN|NI_NUMERICHOST|NI_NUMERICSERV

NI_DGRAM 服务基于流而非数据报，NI_NAMEREQD 如果找不到主机名，将其作为一个错误对待，NI_NUMERICHOST 返回主机地址的数字形式，NI_NUMERICSERV 返回服务地址的数字形式 (即端口号)

30.42 getservbyhost

```
getservbyhost(char * name,char * protocal)
```

name:

protocal:

30.43 gethostbyname

```
gethostbyname(port,char * protocal)
```

port:

protocal:

已过时

30.44 gethostbyaddr

```
gethostbyaddr()
```

已过时

30.45 sendfile

```
sendfile(fd,sockfd,&offset,len)
```

fd:

sockfd:

&offset:

len:

mac 上未成功，且 mac 上有六个参数

30.46 getsockname

```
getsockname(int sockfd,sockaddr * addr,&len)
```

sockfd:

addr:

&len:

获取套接字 socket 所绑定的地址信息并写到 addr 中，len 表示缓冲区的大小，函数返回后 len 的值会变为向 addr 写入的字节数

30.47 getpeername

```
getpeername(int sockfd,sockaddr * addr,&len)
```

sockfd:

addr:

&len:

获取套接字 socket 对端主机的地址信息并写到 addr 中，len 表示缓冲区的大小，函数返回后 len 的值会变为向 addr 写入的字节数

30.48 getsockopt

```
getsockopt(sockfd,level,optname,&optval,&len)
```

sockfd:

level:

optname:

&optval:

&len:

level = SOL_SOCKET optname = SO_REUSEADDR 获取套接字属性

30.49 setsockopt

```
setsockopt(int sockfd,int level,int optname,&optval,len)
```

sockfd:

level:

optname:

&optval:

len:

如果针对的是通用的套接字，将 level 指定为 SO_SOCKET，optname = SO_REUSEADDR 这里我只写了一个常用用法，地址复用，能让服务器重启时立即再次绑定同一个地址，optval 此时可以是一个指向整数的指针，len 表示 optval 指向数据的大小

30.50 sendmsg

```
sendmsg(int sockfd,msghdr * msg,int flag)
```

sockfd:

msg:

flag:

可以看作是使用套接字的 writev，msghdrvoid * msg_name(地址); socklen_t msg_namelen(地址字节数); iovec * msg_iov IO(缓冲数组); int msg_iovlen(数组中的元素个数); void * msg_control(指向控制信息头); socklen_t msg_controllen(控制信息的长度); int msg_flags(接受数据的标志)，msghdr.control 实际上是一个指向 cmsghdr 的指针，cmsghdrsocklen_t cmsg_len; int cmsg_level;

int cmsg_type 为了发送文件描述符，将 cmsg_len 设置为 cmsghdr 结构的长度加一个整形的长度(描述符的长度)，cmsg_level 字段设置为 SOL_SOCKET，cmsg_type 设置为 SCM_RIGHTS，用以表明在传送访问权 (SCM 是套接字级控制信息的缩写)，访问权限仅能通过 UNIX 域套接字发送，描述符仅随 cmsg_type 后存储

30.51 recvmsg

```
recvmsg(int sockfd,msghdr * msg,int flag)
```

sockfd:

msg:

flag:

可以看作是使用套接字的 readv，接受数据后，msghdr 中的 msg_flags 元素的可能值有 MSG_CTRUNC|MSG_EOR|MSG_ERRQUEUE|MSG_OOB|MSG_TRUNC MSG_CTRUNC 表示控制数据被截断，MSG_EOR 表示接受记录结束符，MSG_ERRQUEUE 表示接受错误信息作为辅助数据，MSG_OOB 表示接受带外数据，MSG_TRUNC 表示一般数据被截断

30.52 CMSG_LEN

```
CMSG\_LEN(unsigned int nbytes)
```

nbytes:

->(unsigned int) 返回为 nbytes 长的数据对象分配的空间大小，内部实现就是 sizeof(struct cmsghdr) + nbytes

30.53 CMSG_NXTHDR

```
CMSG\_NXTHDR(struct msghdr * mp, struct cmsghdr * cp)
```

mp:

cp:

->(struct cmsghdr *) 返回一个指针，指向与 msghdr 结构相关联的下一个 cmsghdr 结构，若当前的 cmsghdr 已是追后一个，返回 NULL

30.54 CMSG_FIRSTHDR

```
CMSG\_FIRSTHDR(struct msghdr * mp)
```

mp:

->(struct cmsghdr *) 返回一个指针，指向与 msghdr 结构相关联的第一个 cmsghdr 结构，若无这样的结构，返回 NULL

30.55 CMSG_DATA

```
CMSG\_DATA(struct cmsghdr * cp)
```

cp:

->(unsigned char *) 返回一个指针，指向与 cmsghdr 相关联的数据，内部实现就是 (unsigned char *)cp + sizeof(struct cmsghdr)

31 终端设置

31.1 ioctl

```
ioctl(fd,FIONREAD,&cnt)
```

fd:

FIONREAD:

&cnt:

获取终端输入队列中的未读取字节数

31.2 tcgetattr

tcgetattr(fd, termios *)

fd:

*:

31.3 tcsetattr

tcsetattr(fd, option, termios *)

fd:

option:

*:

option = TCSANOW|TCSADRAIN|TCSAFLUSH 分别为修改立即生效，等处理完终端输出缓冲区的数据后生效，抛弃终端输入缓冲区的数据然后生效

31.4 ioctl

ioctl(fd, TIOCGWINSZ, winsize *)

fd:

TIOCGWINSZ:

*:

31.5 ctermid

ctermid(buf)

buf:

返回进程控制终端的名称

31.6 isatty

isatty(fd)

fd:

判断文件描述符是否同一个终端相关联

31.7 ttyname

ttyname(fd)

fd:

返回这个文件描述符相关联的终端名称

32 IO 多路复用

32.1 select

```
select(int nfds,fd\_set * readset,fd\_set * writeset,fd\_set * errorset,timeval * int
```

nfds:

readset:

writeset:

errorset:

intval:

io 多路复用，等待一定长的时间，返回已准备好的文件描述符个数，nfds 为当前最大的文件描述符加 1，这样就只会检查小于 nfds 的文件描述符状态，intval 等于 NULL 时，永远等待，直到指定中的一个文件描述符已准备好或者捕捉到一个信号终端此进程，intval->tv_sec==0&&intval->tv_usec==0，不等待，测试所有文件描述符后立即返回，当 intval 有其他值时，等待指定的描述和微妙数，当指定描述符中的一个文件描述符准备好时，或者超过指定时间，则立即返回，readset，writeset，errorset 分别返回所关心描述符状态的结果，每一个位对应一个描述符，当调用完成后，若对应位为 1，则表示该下标对应的描述符为准备好状态，如设置为 NULL，则表示对该状态不关心

32.2 pselect

pselect(**nfds**, **fd_set** * **readset**, **fd_set** * **writeset**, **fd_set** * **errorset**, **timespec** * **intva**

nfds:

readset:

writeset:

errorset:

intval:

sigmask:

行为类似于 select，但提供了 sigmask 参数用于当函数调用期间设定的信号屏蔽字，当返回时恢复屏蔽信号字

32.3 FD_ISSET

FD_ISSET(**fd**, **fd_set** * **fdset**)

fd:

fdset:

若 fd 在 fdset 中，返回非 0，否则返回 0，可用于当 select 返回后判断 fd 的状态

32.4 FD_CLR

```
FD\_CLR(int fd,fdset * fdset)
```

fd:

fdset:

将 fd 从 fdset 中移除

32.5 FD_SET

```
FD\_SET(int fd,fdset * fdset)
```

fd:

fdset:

将 fd 加入 fdset

32.6 FD_ZERO

```
FD\_ZERO(fdset * fdset)
```

fdset:

将一个 fdset 的所有位置为 0

32.7 poll

```
poll(pollfd[],int nfds,int timeout)
```

pollfd[]:

nfds:

timeout:

pollfdint fd,short events,short revents events = POLLIN|POLLRDNORM|POLLRDBAND||POLLPRI|POLLOUT POLLERR|POLLHUP|POLLNVAL 这三个值即使不设置在 events 中，也可能出现在 revents ，nfds 指数组元素的个数，timeout 为-1 时永久等待，为 0 时测试后立即返回，其余值时为等待 timeout 毫秒，POLLIN 可以不阻塞的读取高优先级数据以外的数据，POLLRDNORM。可以不阻塞的读取普通数据，POLLRDBAND，可以不阻塞的读取优先级数据，可以不阻塞的读取高优先级的数据，POLLOUT，可以不阻塞的写普通数据，POLLWRNORM，同 POLLOUT，POLLWRBAND，可以不阻塞的写优先级数据，POLLERR，已出错，POLLHUP，已挂断，POLLNVAL，描述符没有引用一个打开文件

32.8 fcntl

```
fcntl(int fd,F\_SETFL,flags|O\_ASYNC)
```

fd:

F_SETFL:

flags|O_ASYNC:

设置信号 IO，通过 fcntl(fd,F_SETOWN,pid) 设置接收 SIGIO 的进程，不能对终端设备使用

32.9

```
((int)
```

(int:

&((struct sockaddr_un *)0)->sun_path) 可用于计算结构体内成员偏移量，等价于 offsetof