# Linux编程思路

1. 与系统打交道的库函数、系统调用一定要做返回值检查
2. 常用的库要理解原理和倒背如流
3. 资源的竞争与释放，调用顺序也会导致竞争
4. Linux为提高效率，会延时写脏页，修改文件时，磁盘存储同步,调用fsync()函数或sync命令
5. 写函数，开关配对，申请内存与释放内存配对,谁new谁delete
6. 用函数时考虑阻塞和非阻塞
7. 释放cpu
8. 资源在多个线程使用要避免竞争，锁一释放，就会存在竞争。
9. 绘制并重审所修改bug所在函数的程序流程图，写代码前写流程图，仔细推敲分支并发情况。
10. 可重入
11. 考虑程序产生的信号，和信号的处理
12. 搜代码：google，github，csdn，pudn，apt-get source code，https://sourceforge.net

技巧：#define sigfillset(ptr) (\*(ptr) = ˜(sigset\_t)0, 0)——宏返回逗号后的值

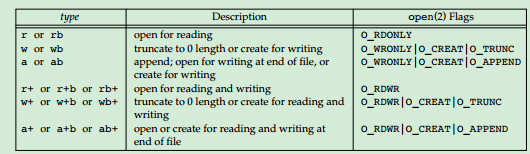
原子操作思想:atomicoperation:If the operation is performed atomically, either all the s teps are performed, or none are performed.

1. 【变量值】赋值与复位都要有,初值在函数中赋值，不能使用初值，无法保证调用时使用的是初值
2. 线程中调用的子函数中不能含有sleep，否则会有多处阻塞
3. 应用程序中不要出现死循环
4. 有交互的就放一个线程处理，没有交互的就放多个线程处理
5. 代码架构，分层只是因为上一层数据解析的方式至少有2种以上，即最少有2个子层，要清晰的界定每一层该做的事，层与层之间的接口只留一个全局的结构体即可（简洁）
6. 函数返回前统一修改全局变量，达到出错一个不该，成功全更新的目的，方便管理
7. 分层思想和面对对象的思想并进，降低程序的耦合度和易于理解
8. 有交互的就放一个线程处理，没有交互的就放多个线程处理
9. 对接口（全局变量）应在同一个地方同时更新，避免只更新了一部分，导致程序状态错乱
10. 多线程架构设计：实时性的生产数据和消费数据分层两个线程，避免消费数据影响实时性；线程内处理数据不能等待源数据。

# 文件I/O——#include <fcntl.h>、#include <unistd.h>

输入/出fd: STDIN\_FILENO、STDOUT\_FILENO

开：int open(const char \*pathname, intoflag, ... /\* mode\_t mode \*/ );



Mode: O\_WRONLY|O\_CREAT|O\_APPEND| O\_RDWR|O\_CREAT|O\_APPEND| O\_RDONLY

Returns: file descriptor if OK, –1 on error

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

Returns: file descriptor opened for write-only if OK, –1 on error

关：int close(intfiledes);

Returns: 0 if OK, –1 on error

偏移：off\_t lseek(intfiledes, off\_t offset, int whence);

Returns: new file offset if OK, –1 on error;whence: SEEK\_SET, SEEK\_CUR, SEEK\_END

读：ssize\_t read(intfiledes, void \*buf, size\_tnbytes) ;

Returns: number of bytes read, 0 if end of file, –1 on error

写：ssize\_t write(intfiledes, const void \*buf, size\_tnbytes);

Returns: number of bytes written if OK, –1 on error

atom写：ssize\_t p write (int filedes, void \*buf, size\_t nbytes , off\_t offset);

Returns: number of bytes read, 0 if end of file, –1 on error

atom读：ssize\_t pread(int filedes, void \*buf, size\_t nbytes , off\_t offset);

Returns: number of bytes written if OK, –1 on error

复制描述符：int dup(int filedes); int dup2(int filedes, int filedes2);

Both return: new file descriptor if OK, –1 on error

同步：int fsync(int filedes);int fdatasync(int filedes); void sync(void)//写入磁盘

Returns: 0 if OK, –1 on error

c库缓冲-----fflush---------〉内核缓冲--------fsync-----〉磁盘

void sync(void);

读/写属性:int fcntl(int filedes, intcmd, ... /\* intarg \*/ ) ;

Returns: depends on cmd if OK (see following), –1 on error

# 文件和目录——#include <sys/stat.h>

读属性：int stat(const char \*restrict pathname, struct stat \*restrict buf);

Int fstat(intfiledes, struct stat \*buf);

Int lstat(const char \*restrict pathname, struct stat \*restrict buf);

All three return: 0 if OK, –1 on error

struct stat

{

mode\_t st\_mode; /\* file type & mode (permissions) \*/

ino\_t st\_ino; /\* i-node number (serial number) \*/

dev\_t st\_dev; /\* device number (file system) \*/

dev\_t st\_rdev; /\* device number for special files \*/

nlink\_t st\_nlink; /\* number of links \*/

uid\_t st\_uid; /\* user ID of owner \*/

gid\_t st\_gid; /\* group ID of owner \*/

off\_t st\_size; /\* size in bytes, for regular files \*/

struct timespec st\_atim; /\* time of last access \*/

struct timespec st\_mtim; /\* time of last modification \*/

struct timespec st\_ctim; /\* time of last file status change \*/

blksize\_t st\_blksize; /\* best I/O block size \*/

blkcnt\_t st\_blocks; /\* number of disk blocks allocated \*/

};

读类型：stat.mode& S\_IFMT

S\_ISREG(struct stat \*restrict buf) ——regular file

S\_ISDIR(struct stat \*restrict buf) ——directory file

S\_ISCHR(struct stat \*restrict buf) ——character special file

S\_ISBLK(struct stat \*restrict buf) ——block special file

S\_ISFIFO(struct stat \*restrict buf) ——pipe or FIFO

S\_ISLNK(struct stat \*restrict buf) ——symbolic link

S\_ISSOCK(struct stat \*restrict buf) ——socket

S\_TYPEISMQ(struct stat \*restrict buf) ——message queue

S\_TYPEISSEM(struct stat \*restrict buf) ——semaphore

S\_TYPEISSHM(struct stat \*restrict buf)—— shared memory object

Allreturn: 1 if ture, 1 if false

读访问权限：int access(const char \*pathname, int mode);

Returns: 0 if OK, –1 on error；mode: R\_OK, W\_OK, X\_OK, F\_OK

设置访问权限：mode\_t umask(mode\_t cmask);——用于程序创建文件时的默认权限

Any bits that are on in the file mode creation mask are turned off in the file’s mode. anyone can read a file, we should set the umask to 0, the umask value that is in effect when our process is running can cause permission bits to be turned off.

Returns: previous file mode creation mask

设置访问权限：int chmod(const char \*pathname, mode\_t mode);

Int fchmod(intfiledes, mode\_t mode);

Both return: 0 if OK, –1 on erro

设置分组：int chown(const char \*pathname, uid\_t owner, gid\_t group);

int fchown(intfiledes, uid\_t owner, gid\_t group);

int lchown(const char \*pathname, uid\_towner,gid\_t group);

All three return: 0 if OK, –1 on error

设置文件大小：int truncate(const char \*pathname, off\_t length);

int ftruncate(int filedes, off\_t length);

Both return: 0 if OK, –1 on error

访问权限标志位：Figure 4.6. The nine file access permission bits, from <sys/stat.h>

st\_mode mask Meaning

S\_IRUSR user-read

S\_IWUSR user-write

S\_IXUSR user-execute

S\_IRGRP group-read

S\_IWGRP group-write

S\_IXGRP group-execute

S\_IROTH other-read

S\_IWOTH other-write

S\_IXOTH other-execute

新建链接符：int link(const char \*existingpath, const char \*newpath);

Int linkat(int fd, const char \*existingpath, intnfd, const char \*newpath,int flag);

flag: AT\_SYMLINK\_NOFOLLOW

Both return: 0 if OK, −1 on error

删除连接符：int unlink(const char \*pathname);

Int unlinkat(intfd, const char \*pathname, int flag);

flag: AT\_REMOVEDIR

Returns: 0 if OK, –1 on error

删除文件/夹：int remove(const char \*pathname);

Returns: 0 if OK, –1 on error

重命名文件/夹:int rename(const char \*oldname, const char \*newname);

Returns: 0 if OK, –1 on error

创建符合链接：int symlink(const char \*actualpath, const char \*sympath);

Returns: 0 if OK, –1 on error

读符号链接：ssize\_t readlink(const char\* restrict pathname,char \*restrict buf,size\_tbufsize);

Returns: number of bytes read if OK, –1 on error

改变文件时间：int futimens(intfd, conststructtimespec times[2]);

int utimensat(intfd, const char \*path, conststructtimespec times[2],int flag);

int utimes(const char \*pathname, conststructtimeval times[2]);

Both return: 0 if OK, −1 on error

返回目录信息：DIR \*opendir(const char \*pathname);pathname:directory name

DIR \*fdopendir(intfd);

Struct DIR{

ino\_td\_ino; /\* i-node number \*/

char d\_name[]; /\* null-terminated filename \*/

}

Both return: pointer if OK, NULL on erro

读目录的文件/目录：struct dirent \*readdir(DIR \*dp);

Returns: pointer if OK, NULL at end of directory or error

关闭目录：void rewinddir(DIR \*dp);

int closedir(DIR \*dp);

Returns: 0 if OK, −1 on error

Long telldir(DIR \*dp);

Returns: current location in directory associated with dp

Void seekdir(DIR \*dp, long loc);

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Description | Example | ls(1) option |
| st\_atim st\_mtim st\_ctim | last-access time of file data  last-modification time of file data  last-change time of i-node status | read  write  chmod, chown | -u default -c |

改变当前工作目录：int chdir(const char \*pathname);

Int fchdir(intfd);

Both return: 0 if OK, −1 on error

获取当前工作目录：char \*getcwd(char \*buf, size\_t size);

Returns: buf if OK, NULL on error

获取设备号：stat.st\_dev:containing that filename and its corresponding i-node

stat.st\_rdev:contains the device number for the actual device

major(stat.st\_rdev) minor(stat.st\_rdev)

# 标准I/O——#include <wchar.h><stdio,h><stdarg.h>（不可重入）

设置字节模式：int fwide(FILE \*fp, int mode);

Returns: positive if stream is wide oriented,negative if stream is byte oriented,or 0 if stream has no orientation

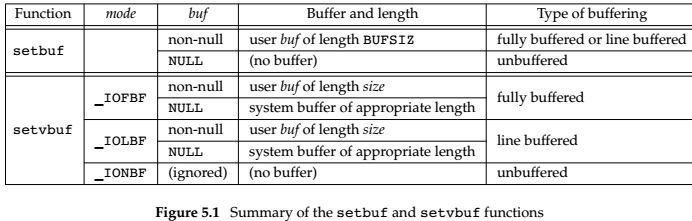
自定义buf：void setbuf(FILE \*restrict fp, char \*restrict buf );

Int setvbuf(FILE \*restrict fp, char \*restrict buf, intmode,size\_t size);

use only before we do any other operation on the stream

mode:\_IOFBF,\_IOLBF,\_IONBF,null size:BUFSIZ

Returns: 0 if OK, nonzero on error



将buf写入内核：int fflush(FILE \*fp); ——只是将buffer写到内核内存，要保证写到磁盘，调用fsync（）；any unflushed standard I/O buffers in memory are discarded

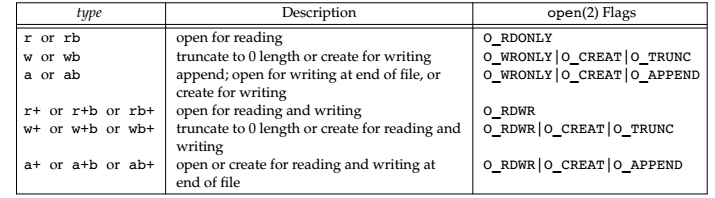
fflush(NULL); /\* all output streams to be flushed \*/

Returns: 0 if OK, EOF on error

创：FILE \*fopen(const char \*restrict pathname, const char \*restrict type);

FILE \*freopen(const char \*restrict pathname, const char \*restrict type,FILE \*restrict fp);

FILE \*fdopen(intfd, const char \*type);



Standard error is always unbuffered； All other streams are line buffered if they refer to a terminal device; otherwise,they are fully buffered.

All three return: file pointer if OK, NULL on error

关：int fclose(FILE \*fp);

Returns: 0 if OK, EOF on error

exit function directly orby returning from the main function, all standard I/O streams with unwritten buffereddata are flushed and all open standard I/O streams are closed

读byte：int getc(FILE \*fp);

int fgetc(FILE \*fp);

int getchar(void); = getc(stdin)

All three return: next character if OK, EOF(-1) on end of file or error

写buffer byte：int ungetc(int c, FILE \*fp);

they are not written back to the underlying fileor device

Returns: c if OK, EOF on error

写文件byte：int putc(int c, FILE \*fp);

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

int putchar(int c);

All three return: c if OK, EOF on error

区分EOF:int ferror(FILE \*fp);

int feof(FILE \*fp);

Both return: nonzero (true) if condition is true, 0 (false) otherwise

清除EOF标志：void clearerr(FILE \*fp);

读line：char \*fgets(char \*restrict buf, int n, FILE \*restrict fp);buf:buf[MAXLINE] n:MAXLINE

char \*gets(char \*buf );--gets function readsfrom standard input

the bufferno more than n − 1 characters， is terminated with a null byte

Both return: buf if OK, NULL on end of file or error

写文件line：int fputs(const char \*restrict str, FILE \*restrict fp);

Int puts(const char \*str);

Both return: non-negative value if OK, EOF on error

读/写结构体：size\_t fread(void \*restrict ptr, size\_t size, size\_tnobj,FILE \*restrict fp);

size\_t fwrite(const void \*restrict ptr, size\_t size, size\_tnobj,FILE \*restrict fp);

size:the size of object ;nobj:number of object

Both return: number of objects read or written

读写位置：long ftell(FILE \*fp);

Returns: current file position indicator if OK, −1L on error

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

Returns: 0 if OK, −1 on error；whence :SEEK\_SET, SEEK\_CUR

void rewind(FILE \*fp);

be set to the beginning ofthe file with the rewind function

off\_t ftello(FILE \*fp);

Returns: current file position indicator if OK, (off\_t)−1 on error

Int fseeko(FILE \*fp, off\_t offset, int whence);

Returns: 0 if OK, −1 on error

Int fgetpos(FILE \*restrict fp, fpos\_t \*restrict pos);

Int fsetpos(FILE \*fp, constfpos\_t \*pos);

格式化输出：

Int printf(const char \*restrict format, ...);

Int fprintf(FILE \*restrict fp, const char \*restrict format, ...);

Int dprintf(intfd, const char \*restrict format, ...);

All three return: number of characters output if OK, negative value if output error

Int sprintf(char \*restrict buf, const char \*restrict format, ...);

Returns: number of characters stored in array if OK, negative value if encoding error

sprintf function automatically appends anull byte at the end of the array, but this null byte is not included in the return value.

Int snprintf(char \*restrict buf, size\_t n,const char \*restrict format, ...);

Returns: number of characters that would have been stored in array, if buffer was large enough, negative value if encoding error

格式转换：%[flags][fldwidth][precision][lenmodifier]convtype——the sequence %n$

represents the nth argument

int vprintf(const char \*restrict format, va\_list arg);

int vfprintf(FILE \*restrict fp, const char \*restrict format,va\_list arg);

int vdprintf(intfd, const char \*restrict format, va\_list arg);

All three return: number of characters output if OK, negative value if output error

Int vsprintf(char \*restrict buf, const char \*restrict format,va\_listarg);

Returns: number of characters stored in array if OK, negative value if encoding error

Int vsnprintf(char \*restrict buf, size\_tn,const char \*restrict format, va\_listarg);

Returns: number of characters that would have been stored in arrayif buffer was large enough, negative value if encoding error

**变参数函数解析：**

void va\_start(va\_list ap, last); //The va\_start() macro initializes ap for subsequent use by va\_arg() and va\_end(), and must be called first.

type va\_arg(va\_list ap, type);

void va\_end(va\_list ap);

void va\_copy(va\_list dest, va\_list src);

**代买实现原理：**

Typedef char \*\_\_builtin\_va\_list va\_list

#define va\_start \_crt\_va\_start

#define va\_arg \_crt\_va\_arg

#define va\_end \_crt\_va\_end

#define \_ADDRESSOF(v) ( &(v) )

#define \_INTSIZEOF(n) ( (sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1) )

#define \_crt\_va\_start(ap,v) ( ap = (va\_list)\_ADDRESSOF(v) + \_INTSIZEOF(v) )

#define \_crt\_va\_arg(ap,t) ( \*(t \*)((ap += \_INTSIZEOF(t)) - \_INTSIZEOF(t)) )

#define \_crt\_va\_end(ap) ( ap = (va\_list)0 )

**解释：**

1. 宏\_ADDRESSOF(v)作用：取参数v的地址。
2. b. 宏\_INTSIZEOF(n)作用：返回参数n的size并保证4字节对齐（32-bits平台）。这个宏应用了一个小技巧来实现字节对齐：~(sizeof(int) - 1)的值对应的2进制值的低k位一定是0，其中sizeof(int) = 2^k，因此，在IA x86\_32下，k=2。理解了这一点，那么(sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1)的作用就很直观了，它保证了sizeof(n)的值按sizeof(int)的值做对齐，例如在32-bits平台下，就是按4字节对齐；在64-bits平台下，按8字节对齐。至于为什么要保证对齐，与编译器的底层实现有关，这里不再展开。
3. \_crt\_va\_start(ap,v)作用：通过v的内存地址来计算ap的起始地址，其中，v是可变参数函数的参数中，最后一个类型已知的参数，执行的结果是ap指向可变参数列表的第1个参数。以int snprintf(char \*str, size\_t size, const char \*format, ...)为例，其函数参数列表中最后一个已知类型的参数是const char \*format，因此，参数format对应的就是\_crt\_va\_start(ap, v)中的v, 而ap则指向传入的第1个可变参数。特别需要理解的是：为什么**ap = address(v) + sizeof(v)，这与函数栈从高地址向低地址的增长方向 及函数调用时参数从右向左的压栈顺序有关**\_crt\_va\_arg(ap,t)作用：更新指针ap后，取类型为t的变量的值并返回该值。
4. \_crt\_va\_end(ap)作用：指针ap置0，防止野指针。

**概括来说，可变参数函数的实现原理是：**

1）根据函数参数列表中最后一个已知类型的参数地址，得到可变参数列表的第一个可变参数

2）根据程序员指定的每个可变参数的类型，通过地址及参数类型的size获取该参数值

3）遍历，直到访问完所有的可变参数

从上面的实现过程可以注意到，可变参数的函数实现严重依赖于函数栈及函数调用约定（主要是参数压栈顺序），同时，依赖于程序员指定的可变参数类型。因此，若指定的参数类型与实际提供的参数类型不符时，程序出core简直就是一定的。

void func(int iAgr1,int iAgr2,unsigned char \*Data)

{

va\_list ap;

va\_start(ap, Data);

s = va\_arg(ap, const char \*); //获取参数

va\_end(ap);

}

可变参数函数：err\_sys(const char \*fmt, ...)

可变参数宏函数：#define my\_printf(….) err\_sys(const char \*fmt, ...)

格式化输入：

Int scanf(const char \*restrict format, ...);

Int fscanf(FILE \*restrict fp, const char \*restrict format, ...);

Int sscanf(const char \*restrict buf, const char \*restrict format, ...);

All three return: number of input items assigned,EOF if input error or end of file before any conversion

格式转换：%[\*][fldwidth][m][lenmodifier]convtype——the sequence %n$

\* 亦可用于格式中, (即 %\*d 和 %\*s) 加了星号 \* 表示跳过此数据不读入(%\*s:不读取字符串)

Such as:

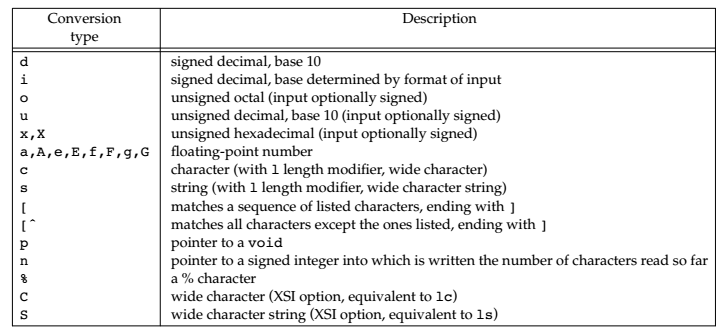
printf（“%05.2f”，i）；

5是一共5的宽度

0是不足的话在左边补0

2是小数点后保留两位

represents the nth argument



Int vscanf(const char \*restrict format, va\_listarg);

Int vfscanf(FILE \*restrict fp, const char \*restrict format,va\_listarg);

Int vsscanf(const char \*restrict buf, const char \*restrict format,va\_listarg);

All three return: number of input items assigned,EOF if input error or end of file before any conversion

转换为fd：int fileno(FILE \*fp);

Returns: the file descriptor associated with the stream，We need this function if we want to call the dup or fcntl functions,

建临时文件：char \*tmpname(char \*ptr);

Ptr: null or char name[L\_tmpnam];

Returns: pointer to unique pathname

FILE \*tmpfile(void);

Returns: file pointer if OK, NULL on error

创建memory stream：FILE \*fmemopen(void \*restrict buf, size\_tsize,const char \*restrict type);

Returns: stream pointer if OK, NULL on error

FILE \*open\_memstream(char \*\*bufp, size\_t \*sizep);byte oriented

FILE \*open\_wmemstream(wchar\_t \*\*bufp, size\_t \*sizep);wide oriented

Both return: stream pointer if OK, NULL on error

# 高级I/O——<fcntl.h>、<sys/select.h>、<poll.h>、<aio.h>、<sys/uio.h>、<sys/mman.h>

记录的锁：int fcntl(int fd, int cmd, ... /\* struct flock \*flockptr \*/ );

cmd :F\_GETLK, F\_SETLK, or F\_SETLKW

Returns: depends on cmd if OK (see following)

struct flock {

short l\_type; /\* F\_RDLCK, F\_WRLCK, or F\_UNLCK \*/

short l\_whence; /\* SEEK\_SET, SEEK\_CUR, or SEEK\_END \*/

off\_t l\_start; /\* offset in bytes, relative to l\_whence \*/

off\_t l\_len; /\* length, in bytes; 0 means lock to EOF \*/

pid\_t l\_pid; /\* returned with F\_GETLK \*/

};

监听I/O：int select(int maxfdp1, fd\_set \*restrict readfds, fd\_set \*restrict writefds, fd\_set \*restrict exceptfds, struct timeval \*restrict tvptr);

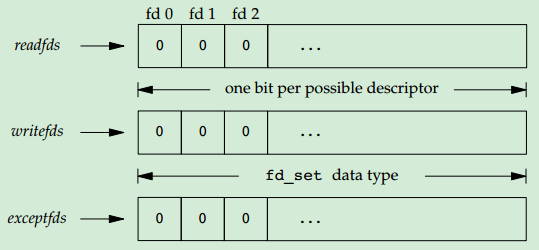
int pselect(int maxfdp1, fd\_set \*restrict readfds,fd\_set \*restrict writefds, fd\_set \*restrict exceptfds,const struct timespec \*restrict tsptr,const sigset\_t \*restrict sigmask);

maxfdp1：maximum file descriptor plus 1,value is fd+1——a count of the number of descriptors to check (starting with descriptor 0).

Tvptr：NULL(Wait forever, be interrupted if we catch a signal); tvptr−>tv\_sec == 0 && tvptr−>tv\_usec == 0 (Don’t wait at all); tvptr−>tv\_sec != 0 || tvptr−>tv\_usec != 0 (Wait the specified number of seconds and microseconds )

Sigmask: points to a signal mask that is atomically installed

Returns: count of ready descriptors, 0 on timeout, −1 on error



设置fd\_set: int FD\_ISSET(int fd, fd\_set \*fdset);——test whether a given bit in the set is still on

Returns: nonzero if fd is in set, 0 otherwise

void FD\_CLR(int fd, fd\_set \*fdset);——clear a single bit

void FD\_SET(int fd, fd\_set \*fdset);——To turn on a single

void FD\_ZERO(fd\_set \*fdset);——set to all zero bits

监听I/O：int poll(struct pollfd fdarray[], nfds\_t nfds, int timeout);

struct pollfd

{

int fd; /\* file descriptor to check, or <0 to ignore \*/

short events; /\* events of interest on fd \*/

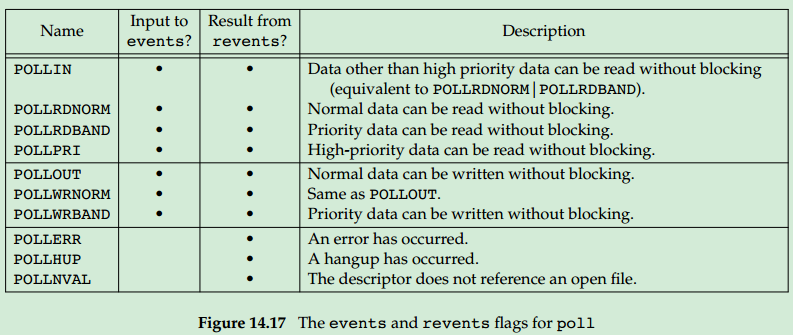
short revents; /\* events that occurred on fd \*/

};

Nfds: The number of elements in the fdarray array

Timeout: −1, Wait forever;0, Don’t wait; > 0, Wait timeout milliseconds

Returns: count of ready descriptors, 0 on timeout, −1 on error



异步I/O读写：int aio\_read(struct aiocb \*aiocb);

int aio\_write(struct aiocb \*aiocb);

struct aiocb

{

int aio\_fildes; /\* file descriptor \*/

off\_t aio\_offset; /\* file offset for I/O \*/

volatile void \*aio\_buf; /\* buffer for I/O \*/

size\_t aio\_nbytes; /\* number of bytes to transfer \*/

int aio\_reqprio; /\* priority \*/

struct sigevent aio\_sigevent; /\* signal information \*/

int aio\_lio\_opcode; /\* operation for list I/O \*/

};

struct sigevent

{

int sigev\_notify; /\* notify type \*/

int sigev\_signo; /\* signal number \*/

union sigval sigev\_value; /\* notify argument \*/

void (\*sigev\_notify\_function)(union sigval); /\* notify function \*/

pthread\_attr\_t \*sigev\_notify\_attributes; /\* notify attrs \*/

};

Both return: 0 if OK, −1 on error

异步I/O同步：int aio\_fsync(int op, struct aiocb \*aiocb);——阻塞

op:O\_DSYNC,behaves like a call to fdatasync;O\_SYNC ,like fsync

return: 0 if OK, −1 on error

读取读/写/同步错误: int aio\_error(const struct aiocb \*aiocb);

Returns: 0 , The asynchronous operation completed successfully;-1 on error; EINPROGRESS,operate still pending

读取读/写/同步结果：ssize\_t aio\_return(const struct aiocb \*aiocb);——only one time per asynchronous I/O operation

return: −1 and set errno if aio\_return itself fails; whatever read, write, or fsync would have returned on success

等待异步操作完成：int aio\_suspend(const struct aiocb \*const list[], int nent, const struct timespec \*timeout);

Returns: 0 if OK, −1 on error

取消等待异步操作完成：int aio\_cancel(int fd, struct aiocb \*aiocb);

Aiocb:NULL,all asynchronous I/O operations on the file;

Returns: AIO\_ALLDONE, AIO\_CANCELED, AIO\_NOTCANCELED,-1 on error

多个异步读/写：int lio\_listio(int mode, struct aiocb \*restrict const list[restrict],

int nent, struct sigevent \*restrict sigev);

mode: LIO\_WAIT or LIO\_NOWAIT; sigev: NULL,not notify,else notify

Returns: 0 if OK, −1 on error

读/写不连续内存：ssize\_t readv(int fd, const struct iovec \*iov, int iovcnt);

ssize\_t writev(int fd, const struct iovec \*iov, int iovcnt);

struct iovec

{

void \*iov\_base; /\* starting address of buffer \*/

size\_t iov\_len; /\* size of buffer \*/

};

Iovcnt：The number of elements in the iov array

Both return: number of bytes read or written, −1 on error

映射文件到内存：void \*mmap(void \*addr, size\_t len, int prot, int flag, int fd, off\_t off );——we cannot append to a file

addr: 0 to allow the system to choose the starting address;len: number of bytes to map;

off :the starting offset in the file of the bytes;

prot: protection of the mapped region, PROT\_READ, PROT\_WRITE, PROT\_EXEC, PROT\_NONE

flag: MAP\_FIXED(The return value must equal addr), MAP\_SHARED(a store operation is equivalent to a write to the file), MAP\_PRIVATE(Any

modifications affect the copy, not the original program file.)

Returns: starting address of mapped region if OK, MAP\_FAILED on error

改变映射内存权限：int mprotect(void \*addr, size\_t len, int prot);

Returns: 0 if OK, −1 on error

同步映射内存到磁盘：int msync(void \*addr, size\_t len, int flags);——is similar to fsync

flags: MS\_ASYNC or MS\_SYNC(wait for completing before returning)

Returns: 0 if OK, −1 on error

取消映射：int munmap(void \*addr, size\_t len);

The updating of the disk file for a MAP\_SHARED region happens automatically by the kernel’s virtual memory algorithm; Modifications to memory in a MAP\_PRIVATE region are discarded when the region is unmapped

Returns: 0 if OK, −1 on error

# 进程环境——<stdlib.h>, <unistd.h>

exit(main(argc, argv));——if the main function returns, the exit function is called

退出：void exit(int status);

void \_Exit(int status);—— prevent any standard I/O buffers

void \_exit(int status);——prevent any standard I/O buffers

注册退出函数：int atexit(void (\*func)(void));

Returns: 0 if OK, nonzero on error

内存分配heap：void \*malloc(size\_t size); ——The initial value of the memory is indeterminate

void \*calloc(size\_t nobj, size\_t size); ——the space is initialized to all 0 bits

void \*realloc(void \*ptr, size\_t newsize); ——if ptr is a null pointer, realloc behaves like malloc and allocates a region of the specified newsize.

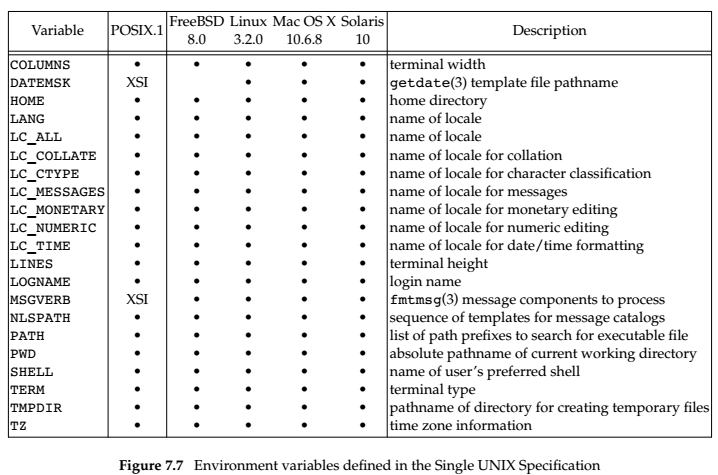
All three return: non-null pointer if OK, NULL on error

void free(void \*ptr);

内存分配stack：void alloca(size\_t size)——the memory is allocated from the stack frame of the current function，we don’t have to free the space

获取环境变量：char \*getenv(const char \*name);

Returns: pointer to value associated with name, NULL if not found



设置环境变量：int putenv(char \*str);——putenv is free to place the string passed to it directly into the environment. it would be an error to pass putenv a string allocated on the stack

Returns: 0 if OK, nonzero on error

int setenv(const char \*name, const char \*value, int rewrite); ——setenv must allocate memory to，create the name=value string from its arguments,

change the current environment and the environment of any subsequent child processes, but we couldn’t affect the environment of the parent process.

Both return: 0 if OK, −1 on error

清除环境变量：int unsetenv(const char \*name);

Both return: 0 if OK, −1 on error

跳转：int setjmp(jmp\_buf env);

Returns: 0 if called directly, nonzero if returning from a call to longjmp

void longjmp(jmp\_buf env, int val);

env: be equal to the setjmp’s env ,val: the val of the setjmp return . Variables that are declared as global , volatile variables or static are left alone when longjmp is executed. when we call longjmp, we abort the other signal handler. The longjmp function is often called from a signal handler to return to the main loop of a program, instead of returning from the handler, which do not save and restore the signal mask.

读写限制：int getrlimit(int resource, struct rlimit \*rlptr);

int setrlimit(int resource, const struct rlimit \*rlptr);

Both return: 0 if OK, −1 on error

string-creation operator (#)

# 信号——<signal.h>、<setjmp.h>、<stdlib.h>、<unistd.h>、<time.h>

原理：when a signal handler is invoked, the signal being caught is added to the current signal mask of the process. The original mask is restored when the signal handler returns.

捕获信号：void ( \*signal(int signo, void (\*func)(int)))(int);

Signo：the name of the signal；func：SIG\_IGN、SIG\_DFL、the address of a function

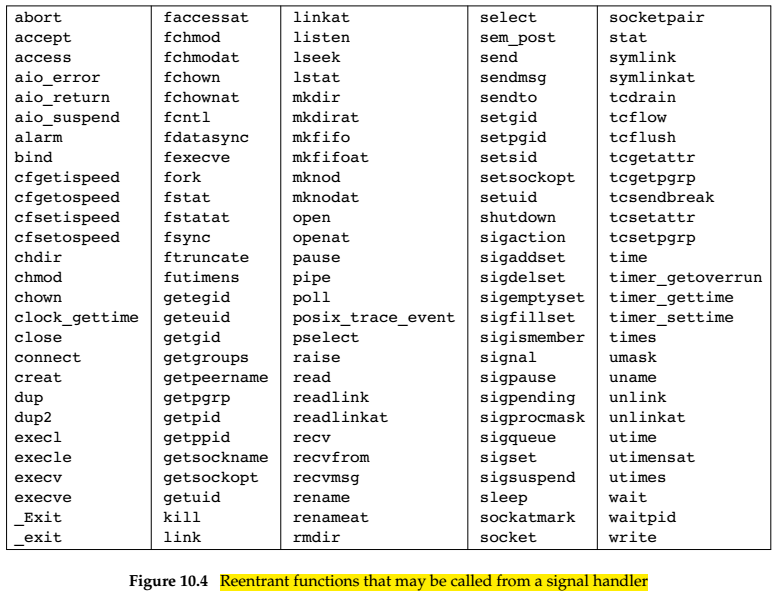
Returns: previous disposition of signal (see following) if OK, SIG\_ERR on error

中断: sig\_int()

{

signal(SIGINT, sig\_int); /\* reestablish handler for next time \*/

. . /\* process the signal ... \*/}



信号处理函数必须是可重入的。

**signal mask**：defines the set of signals currently blocked from delivery to that process.

发送信号：int kill(pid\_t pid, int signo);

int raise(int signo); = kill(getpid(), signo);

signo:0 as the null signal, unblocked signal is delivered to the process before kill returns

Both return: 0 if OK, −1 on error

定时器：unsigned int alarm(unsigned int seconds);

Seconds: 0, the previous alarm clock is canceled; SIGALRM: terminate the process

Returns: 0 or number of seconds until previously set alarm

等待信号：int pause(void);——suspends the calling process until a signal is caught

Returns: −1 with errno set to EINTR

创建信号集：int sigemptyset(sigset\_t \*set);——zeros all bits，unblock all signal

int sigfillset(sigset\_t \*set);——turns on all the bits in the integer，block all signal

All return: 0 if OK, −1 on error

加/减信号：int sigaddset(sigset\_t \*set, int signo);

int sigdelset(sigset\_t \*set, int signo);

All return: 0 if OK, −1 on error

查询信号：int sigismember(const sigset\_t \*set, int signo);

Returns: 1 if true, 0 if false, −1 on error

设置mask：int sigprocmask(int how, const sigset\_t \*restrict set,sigset\_t \*restrict oset);——only for single-threaded processes

how: SIG\_BLOCK——after the signal occur,the handle function cannot be called;, SIG\_UNBLOCK, SIG\_SETMASK；set: The new signal mask for the process, NULL, how is ignored:no change;oset:the old set of signal

Returns: 0 if OK, −1 on error

查询挂起的信号：int sigpending(sigset\_t \*set);

set: The set of signals is returned through the set argument

Returns: 0 if OK, −1 on error

查/改信号属性：int sigaction(int signo, const struct sigaction \*restrict act, struct sigaction \*restrict oact);

Returns: 0 if OK, −1 on error

struct sigaction {

void (\*sa\_handler)(int); /\* addr of signal handler, \*/

/\* or SIG\_IGN, or SIG\_DFL \*/

sigset\_t sa\_mask; /\* additional signals to block \*/

int sa\_flags; /\* signal options, Figure 10.16 \*/

/\* alternate handler \*/

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);

};

struct siginfo\_t {

int si\_signo; /\* signal number \*/

int si\_errno; /\* if nonzero, errno value from errno.h \*/

int si\_code; /\* additional info (depends on signal) \*/

pid\_t si\_pid; /\* sending process ID \*/

uid\_t si\_uid; /\* sending process real user ID \*/

void \*si\_addr; /\* address that caused the fault \*/

int si\_status; /\* exit value or signal number \*/

union sigval si\_value; /\* application-specific value \*/

/\* possibly other fields also \*/};

跳转：int sigsetjmp(sigjmp\_buf env, int savemask);

Savemask: nonzero, then sigsetjmp also saves the current signal mask of the process in env

Returns: 0 if called directly, nonzero if returning from a call to siglongjmp

void siglongjmp(sigjmp\_buf env, int val);

原子变量类型：sig\_atomic\_t——can be written without being interrupted

临界代码：

sigset\_t newmask, oldmask;

sigemptyset(&newmask);

sigaddset(&newmask, SIGINT);

/\* block SIGINT and save current signal mask \*/

if (sigprocmask(SIG\_BLOCK, &newmask, &oldmask) < 0)

err\_sys("SIG\_BLOCK error");

/\* critical region of code \*/

/\* restore signal mask, which unblocks SIGINT \*/

if (sigsuspend(&waitmask) != -1)

err\_sys("sigsuspend error");

/\*

\* Reset signal mask which unblocks SIGINT.

\*/

if (sigprocmask(SIG\_SETMASK, &oldmask, NULL) < 0)

err\_sys("SIG\_SETMASK error");

设置并等待信号：int sigsuspend(const sigset\_t \*sigmask);——restore the signal mask and put the

process to sleep in a single atomic operation，the process is suspended until a signal is caught or until a signal occurs that terminates the process. If a signal is caught and if the signal handler returns, then sigsuspend returns, and the signal mask of the process is set to its value before the call to sigsuspend（mask被设置为调用sissupend之前的值）

sigmask：The signal mask of the process will be set

Returns: −1 with errno set to EINTR——indicating an interrupted system call

唤醒程序：while (quitflag == 0) //wake up the main routine only when the quit signal is caught

sigsuspend(&zeromask);

quitflag = 0;

sig\_int(int signo) /\* one signal handler for SIGINT and SIGQUIT \*/

{

if (signo == SIGQUIT)

quitflag = 1; /\* set flag for main loop \*/

}

终止程序：void abort(void);——This function never returns，and send SIGABRT

kill(getpid(), SIGABRT);

挂起秒：unsigned int sleep(unsigned int seconds);

Returns: 0 or number of unslept seconds

挂起纳秒：int nanosleep(const struct timespec \*reqtp, struct timespec \*remtp);

Reqtp:timr to sleep in seconds and nanoseconds; remtp: time left in the sleep interval, NULL if we are uninterested in the time unslept.be equal: clock\_nanosleep(CLOCK\_REALTIME, 0, reqtp, remtp);

Returns: 0 if slept for requested time or −1 on error

挂起：int clock\_nanosleep(clockid\_t clock\_id, int flags, const struct timespec \*reqtp, struct timespec \*remtp);

Flags：0，the sleep time is relative;TIMER\_ABSTIME, the sleep time is

absolute

Returns: 0 if slept for requested time or error number on failure

发送信号队列：int sigqueue(pid\_t pid, int signo, const union sigval value)

Returns: 0 if OK, −1 on error

job-control signals： (SIGTSTP, SIGTTIN, and SIGTTOU)

打印错误信息：void psignal(int signo, const char \*msg);——be similar to perror

void psiginfo(const siginfo\_t \*info, const char \*msg); ——be similar to perror

char \*strsignal(int signo);——be similar to strerror

Returns: a pointer to a string describing the signal

Msg：output to the standard error, followed by a colon and a space, followed by a description of the signal,followed by a newline;

映射信号码与信号名：int sig2str(int signo, char \*str);

int str2sig(const char \*str, int \*signop);

The string consists of the signal name without the ‘‘SIG’’ prefix. For example, translating SIGKILL would result in the string ‘‘KILL’’ being stored in the str memory buffer[ SIG2STR\_MAX ].

Both return: 0 if OK, −1 on error, don’t set errno when they fail.

# 线程——<pthread.h>、<time.h>

线程号相等：int pthread\_equal(pthread\_t tid1, pthread\_t tid2);

Returns: nonzero if equal, 0 otherwise

获取线程id：pthread\_t pthread\_self(void);

Returns: the thread ID of the calling thread

创建线程：int pthread\_create(pthread\_t \*restrict tidp,const pthread\_attr\_t \*restrict attr,void \*(\*start\_rtn)(void \*), void \*restrict arg);

Returns: 0 if OK, error number on failure

退出线程：void pthread\_exit(void \*rval\_ptr);

rval\_ptr：return value

结束线程：int pthread\_cancel(pthread\_t tid);

Returns: 0 if OK, error number on failure

等待线程结束：int pthread\_join(pthread\_t thread, void \*\*rval\_ptr);

rval\_ptr ：contain the return code

Returns: 0 if OK, error number on failure

结束线程时的回掉函数：void pthread\_cleanup\_push(void (\*rtn)(void \*), void \*arg);

When call pthread\_exit or pthread\_cancel or o pthread\_cleanup\_pop with a nonzero execute argument will do the function

Rtn: cleanup function, arg:the arg of the cleanup function

删除回掉函数：void pthread\_cleanup\_pop(int execute);

execute : zero, the cleanup function is not called; otherwise, call the cleanup function

卸载线程：int pthread\_detach(pthread\_t tid);

Returns: 0 if OK, error number on failure

创建互斥锁：int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex,const pthread\_mutexattr\_t \*restrict attr);

pthread\_mutex\_t f\_lock = PTHREAD\_MUTEX\_INITIALIZER;静态创建，不需要销毁

销毁互斥锁：int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

Both return: 0 if OK, error number on failure

获取互斥锁：int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex); ——block

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);——nonblack

int pthread\_mutex\_timedlock(pthread\_mutex\_t \*restrict mutex, const struct timespec \*restrict tsptr); ——timeout

All return: 0 if OK, error number on failure

释放互斥锁：int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

return: 0 if OK, error number on failure

创建读写锁：int pthread\_rwlock\_init(pthread\_rwlock\_t \*restrict rwlock, const pthread\_rwlockattr\_t \*restrict attr);

return: 0 if OK, error number on failure

销毁读写锁：int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock);

return: 0 if OK, error number on failure

获取读/写锁: int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rwlock);——block

int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*rwlock); ——block

int pthread\_rwlock\_tryrdlock(pthread\_rwlock\_t \*rwlock); ——nonblack

int pthread\_rwlock\_trywrlock(pthread\_rwlock\_t \*rwlock); ——nonblack

int pthread\_rwlock\_timedrdlock(pthread\_rwlock\_t \*restrict rwlock,

const struct timespec \*restrict tsptr);——timeout

int pthread\_rwlock\_timedwrlock(pthread\_rwlock\_t \*restrict rwlock,

const struct timespec \*restrict tsptr); ——timeout

return: 0 if OK, error number on failure

释放读/写锁：int pthread\_rwlock\_unlock(pthread\_rwlock\_t \*rwlock)

return: 0 if OK, error number on failure

创建条件变量：int pthread\_cond\_init(pthread\_cond\_t \*restrict cond, const pthread\_condattr\_t \*restrict attr);

销毁条件变量：int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

Both return: 0 if OK, error number on failure

获取条件变量：int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond,pthread\_mutex\_t \*restrict mutex);

int pthread\_cond\_timedwait(pthread\_cond\_t \*restrict cond,pthread\_mutex\_t \*restrict mutex,const struct timespec \*restrict tsptr);

Both return: 0 if OK, error number on failure

发送条件变量：int pthread\_cond\_signal(pthread\_cond\_t \*cond);

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

Both return: 0 if OK, error number on failure

创建自旋锁：int pthread\_spin\_init(pthread\_spinlock\_t \*lock, int pshared);

销毁自旋锁：int pthread\_spin\_destroy(pthread\_spinlock\_t \*lock);

Both return: 0 if OK, error number on failure

获取自旋锁：int pthread\_spin\_lock(pthread\_spinlock\_t \*lock);

int pthread\_spin\_trylock(pthread\_spinlock\_t \*lock);

All return: 0 if OK, error number on failure

释放自旋锁：int pthread\_spin\_unlock(pthread\_spinlock\_t \*lock);

return: 0 if OK, error number on failure

创建障碍：int pthread\_barrier\_init(pthread\_barrier\_t \*restrict barrier,const pthread\_barrierattr\_t \*restrict attr,unsigned int count);

return: 0 if OK, error number on failure

销毁障碍：int pthread\_barrier\_destroy(pthread\_barrier\_t \*barrier);

获得障碍：int pthread\_barrier\_wait(pthread\_barrier\_t \*barrier);

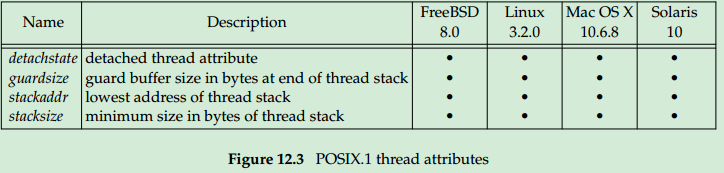
Returns: 0 or PTHREAD\_BARRIER\_SERIAL\_THREAD if OK, error number on failure

# 线程控制——<pthread.h>

创/删属性：int pthread\_attr\_init(pthread\_attr\_t \*attr);

int pthread\_attr\_destroy(pthread\_attr\_t \*attr);

return: 0 if OK , error number on failure



读/写析构：int pthread\_attr\_getdetachstate(const pthread\_attr\_t \*restrict attr,

int \*detachstate);

int pthread\_attr\_setdetachstate(pthread\_attr\_t \*attr, int detachstate);

detachstate ：PTHREAD\_CREATE\_DETACHED or THREAD\_CREATE\_JOINABLE

Both return: 0 if OK, error number on failure

读/写堆栈：int pthread\_attr\_getstack(const pthread\_attr\_t \*restrict attr,void \*\*restrict stackaddr,size\_t \*restrict stacksize);

int pthread\_attr\_setstack(pthread\_attr\_t \*attr,void \*stackaddr, size\_t tacksize);

Both return: 0 if OK, error number on failure

读/写堆栈大小：int pthread\_attr\_getstacksize(const pthread\_attr\_t \*restrict attr,size\_t \*restrict stacksize);

int pthread\_attr\_setstacksize(pthread\_attr\_t \*attr, size\_t stacksize);

Both return: 0 if OK, error number on failure

读/写保护区：int pthread\_attr\_getguardsize(const pthread\_attr\_t \*restrict attr,size\_t \*restrict guardsize);

int pthread\_attr\_setguardsize(pthread\_attr\_t \*attr, size\_t guardsize);

Both return: 0 if OK, error number on failure

创/删mutex属性：int pthread\_mutexattr\_init(pthread\_mutexattr\_t \*attr);

int pthread\_mutexattr\_destroy(pthread\_mutexattr\_t \*attr);

Both return: 0 if OK, error number on failure

读/写share属性:int pthread\_mutexattr\_getpshared(const pthread\_mutexattr\_t \*restrict attr,int \*restrict pshared);

int pthread\_mutexattr\_setpshared(pthread\_mutexattr\_t \*attr,int pshared);

pshared: PTHREAD\_PROCESS\_PRIVATE

Both return: 0 if OK, error number on failure

读/写robust属性：int pthread\_mutexattr\_getrobust(const pthread\_mutexattr\_t \*restrict attr,

int \*restrict robust);

int pthread\_mutexattr\_setrobust(pthread\_mutexattr\_t \*attr,int robust);

robust：PTHREAD\_MUTEX\_STALLED or PTHREAD\_MUTEX\_ROBUST

Both return: 0 if OK, error number on failure

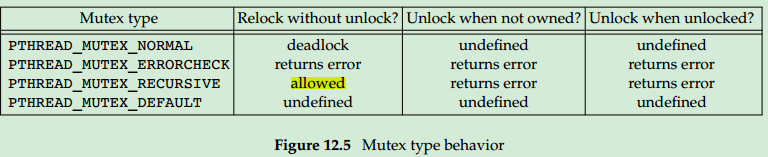
写死锁属性：int pthread\_mutex\_consistent(pthread\_mutex\_t \* mutex);

Returns: 0 if OK, error number on failure

读/写type属性：int pthread\_mutexattr\_gettype(const pthread\_mutexattr\_t \*restrict attr, int \*restrict type);

int pthread\_mutexattr\_settype(pthread\_mutexattr\_t \*attr, int type);

Both return: 0 if OK, error number on failure



创/删读写锁属性：int pthread\_rwlockattr\_init(pthread\_rwlockattr\_t \*attr);

int pthread\_rwlockattr\_destroy(pthread\_rwlockattr\_t \*attr);

Both return: 0 if OK, error number on failure

读/写share属性：int pthread\_rwlockattr\_getpshared(const pthread\_rwlockattr\_t \*restrict attr,int \*restrict pshared);

int pthread\_rwlockattr\_setpshared(pthread\_rwlockattr\_t \*attr,int pshared);

Both return: 0 if OK, error number on failure

创/删Condition属性：int pthread\_condattr\_init(pthread\_condattr\_t \*attr);

int pthread\_condattr\_destroy(pthread\_condattr\_t \*attr);

Both return: 0 if OK, error number on failure

读/写share属性：int pthread\_condattr\_getpshared(const pthread\_condattr\_t \*restrict attr,

int \*restrict pshared);

int pthread\_condattr\_setpshared(pthread\_condattr\_t \*attr,int pshared);

Both return: 0 if OK, error number on failure

读/写超时属性：int pthread\_condattr\_getclock(const pthread\_condattr\_t \*restrict attr,

clockid\_t \*restrict clock\_id);

int pthread\_condattr\_setclock(pthread\_condattr\_t \*attr,clockid\_t clock\_id);

Both return: 0 if OK, error number on failure

创/删障碍属性：int pthread\_barrierattr\_init(pthread\_barrierattr\_t \*attr);

int pthread\_barrierattr\_destroy(pthread\_barrierattr\_t \*attr);

Both return: 0 if OK, error number on failure

读/写share属性：int pthread\_barrierattr\_getpshared(const pthread\_barrierattr\_t \*restrict attr,

int \*restrict pshared);

int pthread\_barrierattr\_setpshared(pthread\_barrierattr\_t \*attr,int pshared);

Both return: 0 if OK, error number on failure

获得文件锁：int ftrylockfile(FILE \*fp);

Returns: 0 if OK, nonzero if lock can’t be acquired

void flockfile(FILE \*fp);

释放文件锁：void funlockfile(FILE \*fp);

字符锁：int getchar\_unlocked(void);

int getc\_unlocked(FILE \*fp);

Both return: the next character if OK, EOF on end of file or error

int putchar\_unlocked(int c);

int putc\_unlocked(int c, FILE \*fp);

Both return: c if OK, EOF on error

创建指定数据的key：int pthread\_key\_create(pthread\_key\_t \*keyp, void (\*destructor)(void \*));

Destructor：When the thread exits, if the data address has been set

to a non-null value, the destructor function is called

Returns: 0 if OK, error number on failure

删除key：int pthread\_key\_delete(pthread\_key\_t key);

Returns: 0 if OK, error number on failure

避免线程竞争：pthread\_once\_t initflag = PTHREAD\_ONCE\_INIT;

int pthread\_once(pthread\_once\_t \*initflag, void (\*initfn)(void));

Returns: 0 if OK, error number on failure

读/写线程数据：void \*pthread\_getspecific(pthread\_key\_t key);

Returns: thread-specific data value or NULL if no value has been associated with the key

int pthread\_setspecific(pthread\_key\_t key, const void \*value);

Returns: 0 if OK, error number on failure

使能取消点：int pthread\_setcancelstate(int state, int \*oldstate);

cancelability state：PTHREAD\_CANCEL\_ENABLE、PTHREAD\_CANCEL\_DISABLE

Returns: 0 if OK, error number on failure

设置取消点：void pthread\_testcancel(void);

设置取消同/异步：int pthread\_setcanceltype(int type, int \*oldtype);

Type: PTHREAD\_CANCEL\_DEFERRED or PTHREAD\_CANCEL\_ASYNCHRONOUS

Returns: 0 if OK, error number on failure

绑定信号：int pthread\_sigmask(int how, const sigset\_t \*restrict set,sigset\_t \*restrict oset);

how：SIG\_BLOCK、SIG\_SETMASK、SIG\_UNBLOCK

Returns: 0 if OK, error number on failure

获取信号：int sigwait(const sigset\_t \*restrict set, int \*restrict signop);

Returns: 0 if OK, error number on failure

发送信号：int pthread\_kill(pthread\_t thread, int signo);

Returns: 0 if OK, error number on failure

fork清除锁的状态：int pthread\_atfork(void (\*prepare)(void), void (\*parent)(void),void (\*child)(void));

prepare：acquire all locks；parent：unlock all the locks；child：unlock all the locks

Returns: 0 if OK, error number on failure

# 字符串——<stdlib.h>、<string.h>、<sys/types.h>、<regex.h>

str转换为数字：double atof(const char \*nptr);int atoi(const char \*nptr);long atol(const char \*nptr);——有符号数

unsigned long strtoul (const char\* str, NULL, 0);——无符号数，

Return the number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

数字转换为str：char \*gcvt(double number，size\_tndigits，char \*buf);inttoascii(int c)

Ndigits:显示的位数 ; int c:为有符号数

大小写：string\* tolower(string str); string\* toupper(string str);

比较内存：int memcmp ( const void \*s1,const void \* s2,int n);

Return : 0:s1=s2 >0:s1>s2 ; <0:s1<s2

清0：void bzero(void \*s,int n)；所指的内存区域前n个字节，全部设为零值

查找字符：char \* index( const char \*s, int c);

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

char \*strstr(const char \*source,const char \*destination);

Return :该字符c/destination首次出现的地址

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

Return : 该字符c最后一次出现的地址

拷贝n个字符：void \* memcpy (void \* dest ,const void \*src, size\_t n);指针src和dest所指的内存区域不可重叠

char \*strncpy(char \*dest, const char \*src, int n)

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

void \* memmove(void \*dest,const void \*src,size\_t n);指针src和dest所指的内存区域可以重叠。

All return：指向dest的指针

内存置为c：void \* memset (void \*s ,int c, size\_t n);

比较大小：int strcasecmp (const char \*s1, const char \*s2);忽略大小写

Int strcmp(const char \*s1,const char \*s2);不忽略大小写

Return : 0:s1=s2 >0:s1>s2 ; <0:s1<s2

连接字符串：char \*strcat (char \*dest,const char \*src);

Return:dest的字符串起始地址

复制：char \* strdup( const char \*s);

Return:指向复制后的新字符串地址 or null;最后可以利用free()来释放

计算长度：size\_t strlen (const char \*s);

分割：1 . char \* strtok(char \*s,const char \*delim);

Return:the address of the string or NULL;用‘\0’替代delim，第一次用:char s=strtok(“wmh11wmh”,”wmh”);以后用：:char s = strtok (NULL,”wmh”)

2. pTmp = strchr(content, '=');\*pTmp = 0; //content分为pTmp和pTmp+1两字符串

建立正则表达式：int regcomp (regex\_t \*compiled, const char \*pattern, int cflags)

Cflags: REG\_EXTENDED 、REG\_ICASE 匹配字母时忽略大小写、

REG\_NOSUB 不用存储匹配后的结果、REG\_NEWLINE 识别换行符，这样'$'就可以从行尾开始匹配，'^'就可以从行的开头开始匹配。

Return: 0 success;-1 failed

匹配正则表达式：int regexec (regex\_t \*compiled, char \*string, size\_t nmatch, regmatch\_t matchptr [], int eflags)

typedef struct

{

regoff\_t rm\_so; //存放匹配文本串在目标串中的开始位置

regoff\_t rm\_eo; //存放结束位置

} regmatch\_t;

释放正则表达式：void regfree (regex\_t \*compiled)

输出错误：size\_t regerror (int errcode, regex\_t \*compiled, char \*buffer, size\_t length)

# 系统文件和信息——<pwd.h>、<shadow.h>、<grp.h>、<sys/utsname.h>、<sys/time.h>

关键字获取密码：struct passwd \*getpwuid(uid\_t uid);

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

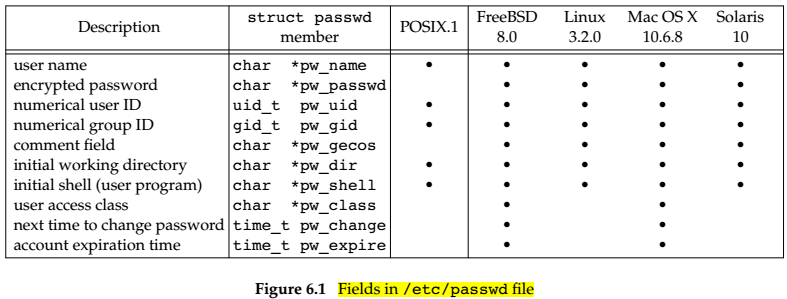
Both return: pointer if OK, NULL on error

轮询获取密码：void setpwent(void);

struct passwd \*getpwent(void);

Returns: pointer if OK, NULL on error or end of file

void endpwent(void);



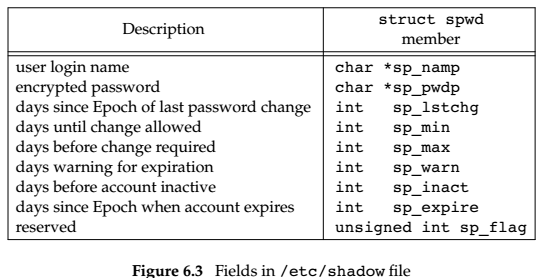
关键字获取加密密码：struct spwd \*getspnam(const char \*name);

轮询获取加密密码：void setspent(void);

struct spwd \*getspent(void);

Both return: pointer if OK, NULL on error

void endspent(void);



关键字获取组：struct group \*getgrgid(gid\_t gid);

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

Both return: pointer if OK, NULL on error

轮询获取组：struct group \*getgrent(void);

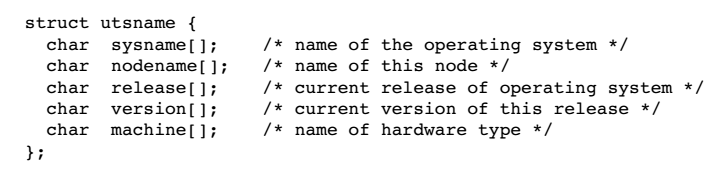
Returns: pointer if OK, NULL on error or end of file

void setgrent(void);

void endgrent(void);

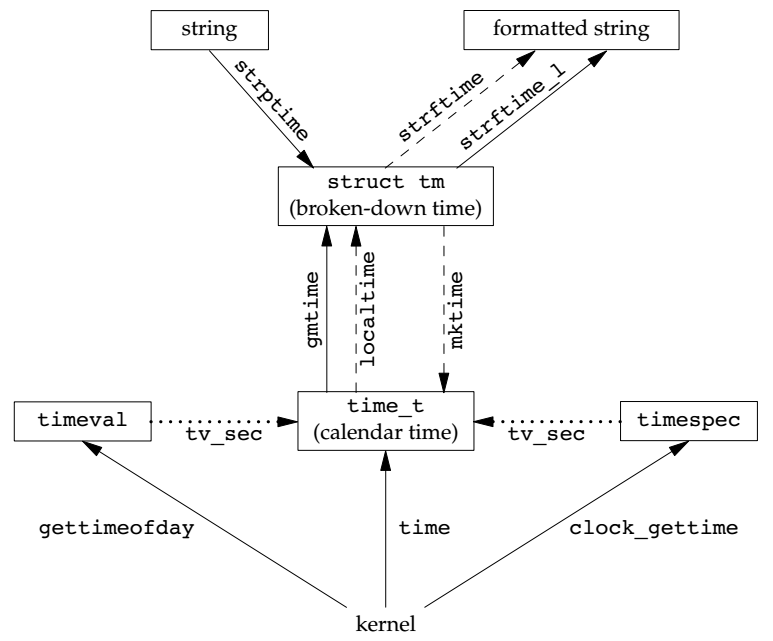
查询操作系统：int uname(struct utsname \*name);

Returns: non-negative value if OK, −1 on error



获取TCP/IP主机名：int gethostname(char \*name, int namelen);

Returns: 0 if OK, −1 on error



获取时间（秒）：time\_t time(time\_t \*calptr);

Returns: value of time if OK, −1 on error

获取不同类型时间：int clock\_gettime(clockid\_t clock\_id, struct timespec \*tsp);

struct timespec

{

\_\_time\_t tv\_sec; /\* Seconds. \*/

long int tv\_nsec; /\* Nanoseconds. \*/

};

Returns: 0 if OK, −1 on error

时间转换：int clock\_getres(clockid\_t clock\_id, struct timespec \*tsp);

Returns: 0 if OK, −1 on error

设置时间：int clock\_settime(clockid\_t clock\_id, const struct timespec \*tsp);

Returns: 0 if OK, −1 on error

获取时间（秒+ms）：int gettimeofday(struct timeval \*restrict tp, void \*restrict tzp);

tzp is NULL

struct timeval

{

\_\_time\_t tv\_sec; /\* Seconds. \*/

\_\_suseconds\_t tv\_usec; /\* Microseconds. \*/

};

Returns: 0 always

秒转为当地时间: struct tm \*gmtime(const time\_t \*calptr);

——expressed in Coordinated Universal Time (UTC)

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

——expressed relative to the user's specified timezone

Both return: pointer to broken-down time, NULL on error

struct tm { /\* a broken-down time \*/

int tm\_sec; /\* seconds after the minute: [0 - 60] \*/

int tm\_min; /\* minutes after the hour: [0 - 59] \*/

int tm\_hour; /\* hours after midnight: [0 - 23] \*/

int tm\_mday; /\* day of the month: [1 - 31] \*/

int tm\_mon; /\* months since January: [0 - 11] \*/

int tm\_year; /\* years since 1900 \*/

int tm\_wday; /\* days since Sunday: [0 - 6] \*/

int tm\_yday; /\* days since January 1: [0 - 365] \*/

int tm\_isdst; /\* daylight saving time flag: <0, 0, >0 \*/

}

当地时间转为秒：time\_t mktime(struct tm \*tmptr);

Returns: calendar time if OK, −1 on error

当地时间转为字符串：size\_t strftime(char \*restrict buf, size\_t maxsize, const char \*restrict format, const struct tm \*restrict tmptr);

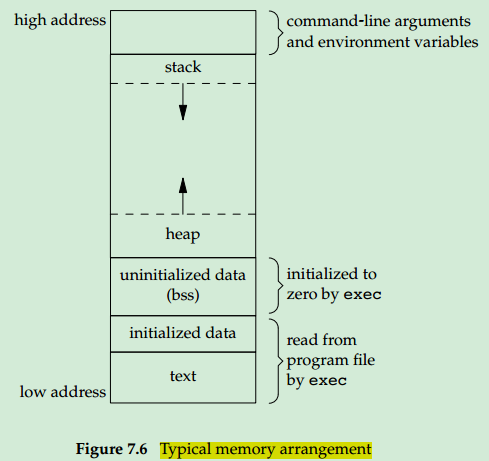
size\_t strftime\_l(char \*restrict buf, size\_t maxsize,const char \*restrict format,

const struct tm \*restrict tmptr, locale\_t locale);

Both return: number of characters stored in array if room, 0 otherwise

char \*asctime(const struct tm \*tm);——输出字符串：Local time is Fri Oct 27 14:48:29 2017

# 进程控制——<sys/wait.h>、<sys/resource.h>、<sys/time.h>



获取ID: pid\_t getpid(void);

Returns: process ID of calling process

pid\_t getppid(void);

Returns: parent process ID of calling process

uid\_t getuid(void);

Returns: real user ID of calling process

uid\_t geteuid(void);

Returns: effective user ID of calling process

gid\_t getgid(void);

Returns: real group ID of calling process

gid\_t getegid(void);

Returns: effective group ID of calling process

创建子进程：pid\_t fork(void);

Returns: 0 in child, process ID of child in parent, −1 on erro

pid\_t fork(void);

Returns: 0 in child, process ID of child in parent, −1 on erro，the child runs in the address space of the parent，vfork guarantees that the child runs first, until the child calls exec or exit。

等待子进程退出：pid\_t wait(int \*statloc);——wait for the child that terminates first

pid\_t waitpid(pid\_t pid, int \*statloc, int options);

statloc：store the the termination status of the terminated process；pid:the pid of a specific process.

Options: either is 0 or is constructed from the bitwise OR of the constants

Both return: process ID if OK, 0 (see later), or −1 on error

int waitid(idtype\_t idtype, id\_t id, siginfo\_t \*infop, int options);

Returns: 0 if OK, −1 on error

pid\_t wait3(int \*statloc, int options, struct rusage \*rusage);

pid\_t wait4(pid\_t pid, int \*statloc, int options, struct rusage \*rusage);

Both return: process ID if OK, 0, or −1 on error

等待父进程退出：while (getppid() != 1) sleep(1);

执行新程序：int execl(const char \*pathname, const char \*arg0, ... /\* (char \*)0 \*/ );

int execv(const char \*pathname, char \*const argv[]);

int execle(const char \*pathname, const char \*arg0, .../\* (char \*)0, char \*const envp[] \*/ );

int execve(const char \*pathname, char \*const argv[], char \*const envp[]);

If either execlp or execvp finds an executable file using one of the path prefixes,

int execlp(const char \*filename, const char \*arg0, ... /\* (char \*)0 \*/ );

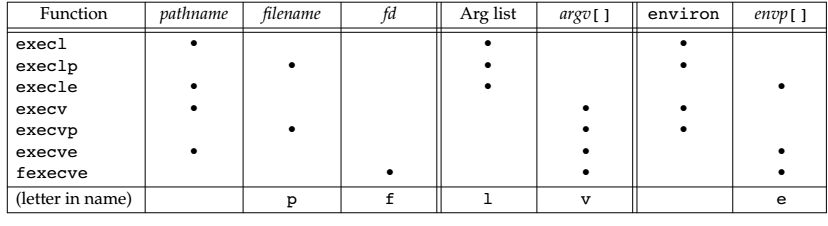
int execvp(const char \*filename, char \*const argv[]);

int fexecve(int fd, char \*const argv[], char \*const envp[]);

All seven return: −1 on error, no return on success

l stands for list null pointer is specified by the constant 0, we must cast it to a pointer and v stands for vector , e stands for environment parameter. p means that the function takes a

filename argument and uses the PATH environment variable to find the executable file, 



设置程序的ID: int setuid(uid\_t uid);——set real user ID and effective user ID

int setgid(gid\_t gid);——set real group ID and effective group ID

Both return: 0 if OK, −1 on error

int setreuid(uid\_t ruid, uid\_t euid);——set real user ID

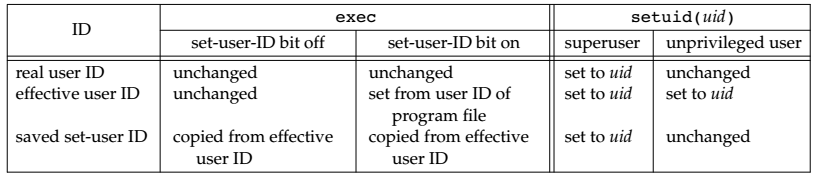
int setregid(gid\_t rgid, gid\_t egid); ;——set real group ID

Both return: 0 if OK, −1 on error

int seteuid(uid\_t uid); ——set effective user ID

int setegid(gid\_t gid); ——set effective group ID

Both return: 0 if OK, −1 on error



文件解析器：#!—and the interpreter

执行命令：int system(const char \*cmdstring);

Returns: (see below)

execl("/bin/sh", "sh", "-c", cmdstring, (char \*)0);

获取登录用户名：char \*getlogin(void);

Returns: pointer to string giving login name if OK, NULL on error

获取登录密码： char \* getpwnam(const char \* login)

Returns: pointer to string giving login name if OK, NULL on error

设置nice值：int nice(int incr);

Incr：is added to the nice value of the calling process

Returns: new nice value − NZERO if OK, −1 on error

int setpriority(int which, id\_t who, int value);

Returns: 0 if OK, −1 on error，value：is added to NZERO and this becomes the new nice value

读取nice值：int getpriority(int which, id\_t who);

Returns: nice value between −NZERO and NZERO−1 if OK, −1 on error

获取时间值：clock\_t times(struct tms \*buf );

Returns: elapsed wall clock time in clock ticks if OK, −1 on error

每秒的tick数：clock\_t sysconf(\_SC\_CLK\_TCK))

# 进程关系——<unistd.h>

获取组ID:pid\_t getpgrp(void);

Returns: process group ID of calling process

pid\_t getpgid(pid\_t pid);

Returns: process group ID if OK, −1 on error

设置组ID：int setpgid(pid\_t pid, pid\_t pgid);

Pgid: process group id ; pid: process id

Returns: 0 if OK, −1 on error

建立session：pid\_t setsid(void); ——A session is a collection of one or more process groups

Returns: process group ID if OK, −1 on error

获取session ID: pid\_t getsid(pid\_t pid);

Returns: session leader’s process group ID if OK, −1 on error

# 套接字——#include <sys/socket.h>,include <netdb.h>

创：int socket(int domain, int type, int protocol);

Domain：AF\_INET,AF\_INET6，AF\_UNIX,AF\_UNSPEC

type: SOCK\_STREAM, SOCK\_DGRAM, SOCK\_SEQPACKET

Returns: file (socket) descriptor if OK, −1 on error

禁止读/写：int shutdown(intsockfd, int how);

how: SHUT\_RD, SHUT\_WR, SHUT\_RDWR

Returns: 0 if OK, −1 on error

大小端转换：uint32\_t htonl(uint32\_t hostint32);

Returns: 32-bit integer in network byte order

uint16\_t htons(uint16\_t hostint16);

Returns: 16-bit integer in network byte order

uint32\_t ntohl(uint32\_t netint32);

Returns: 32-bit integer in host byte order

uint16\_t ntohs(uint16\_t netint16);

Returns: 16-bit integer in host byte order

地址转换：const char \*inet\_ntop(int domain, const void \*restrict addr,char \*restrict str, socklen\_t size);

Size：INET\_ADDRSTRLEN,INET6\_ADDRSTRLEN

Returns: pointer to address string on success, NULL on error

Int inet\_pton(int domain, const char \*restrict str,void \*restrict addr);

Returns: 1 on success, 0 if the format is invalid, or −1 on error

查看属性：

主机：struct hostent \*gethostent(void);

Returns: pointer if OK, NULL on error

Void sethostent(intstayopen);

Void endhostent(void);

网络：struct netent \*getnetbyaddr(uint32\_t net, int type);

Struct netent \*getnetbyname(const char \*name);

Struct netent \*getnetent(void);

All return: pointer if OK, NULL on error

voidsetnetent(intstayopen);

voidendnetent(void

协议：structprotoent \*getprotobyname(const char \*name);

structprotoent \*getprotobynumber(int proto);

structprotoent \*getprotoent(void);

All return: pointer if OK, NULL on error

voidsetprotoent(intstayopen);

voidendprotoent(void);

服务：structservent \*getservbyname(const char \*name, const char \*proto);

structservent \*getservbyport(int port, const char \*proto);

structservent \*getservent(void);

All return: pointer if OK, NULL on error

Void setservent(intstayopen);

voidendservent(void);

地址：int getaddrinfo(const char \*restrict host,const char \*restrict service,const struct addrinfo \*restrict hint,structaddrinfo \*\*restrict res);

Returns: 0 if OK, nonzero error code on error

Void freeaddrinfo(structaddrinfo \*ai);

const char \*gai\_strerror(int error);

convert the error code returnedinto an error message.

Returns: a pointer to a string describing the error；

地址转为主机和服务：int getnameinfo(conststructsockaddr \*restrict addr, socklen\_talen,char \*restrict host, socklen\_thostlen,char \*restrict service, socklen\_tservlen, int flags);

Returns: 0 if OK, nonzero on error

设置socket：int bind(intsockfd, conststructsockaddr \*addr, socklen\_tlen);

Returns: 0 if OK, −1 on error

查找socket：int getsockname(intsockfd, structsockaddr \*restrict addr,socklen\_t \*restrict alenp);

Alenp:an integer containing the size ofthe sockaddr buffer.

Returns: 0 if OK, −1 on error

intgetpeername(intsockfd, structsockaddr \*restrict addr,socklen\_t \*restrict alenp);

Returns: 0 if OK, −1 on error

连接socket：int connect(int sockfd, const struct sockaddr \*addr, socklen\_tlen);

Addr：the address of the server with which we wishto communicate

Returns: 0 if OK, −1 on error

server允许连接：int listen(int sockfd, int backlog);

Backlog:the number of outstanding connect requests.

Returns: 0 if OK, −1 on error

server建立连接：int accept(int sockfd, struct sockaddr \*restrict addr,socklen\_t \*restrict len);

Addr：a bufferlarge enough to hold the addresslen：the size of the address.

Returns: new file (socket) descriptor if OK, −1 on error

发数据：ssize\_t send(intsockfd, const void \*buf, size\_tnbytes, int flags);

Returns: number of bytes sent if OK, −1 on error

ssize\_tsendto(intsockfd, const void \*buf, size\_tnbytes, intflags,conststructsockaddr \*destaddr, socklen\_tdestlen);

destaddr：a destination address to be used with connectionless sockets

Returns: number of bytes sent if OK, −1 on error

ssize\_tsendmsg(intsockfd, conststructmsghdr \*msg, int flags);

Returns: number of bytes sent if OK, −1 on error

收数据：ssize\_trecv(intsockfd, void \*buf, size\_tnbytes, int flags);

ssize\_trecvfrom(intsockfd, void \*restrict buf, size\_tlen, intflags,structsockaddr \*restrict addr,socklen\_t \*restrict addrlen);

ssize\_trecvmsg(intsockfd, structmsghdr \*msg, int flags);

Returns: length of message in bytes,0 if no messages are available and peer has done an orderly shutdown,or −1 on erro

设置socket：intsetsockopt(intsockfd, int level, int option, const void \*val,socklen\_tlen)

level :the number of the protocol that controls the option ;Len：the size ofthe object to which valpoints;val: a data structure or an integer,

Returns: 0 if OK, −1 on error

查看socket:int getsockopt(intsockfd, int level, int option, void \*restrict val,socklen\_t \*restrict lenp);

Returns: 0 if OK, −1 on error

Server： socket——>bind——>listen——>accept

Client: socket——>bind——>connect

# 压缩并删除原文件夹（递归）

zip -rvmq ./test.zip test 中 –r 递归

tar -cvf test.tar ./test --remove-files 打包 最大

tar -zcf test.tar.gz ./test --remove-files 小

tar -jcf test.tar.bz2 ./test --remove-files 大

tar –tf test.tar ——参考文件列表

# 守护进程——<syslog.h>

void openlog(const char \*ident, int option, int facility);

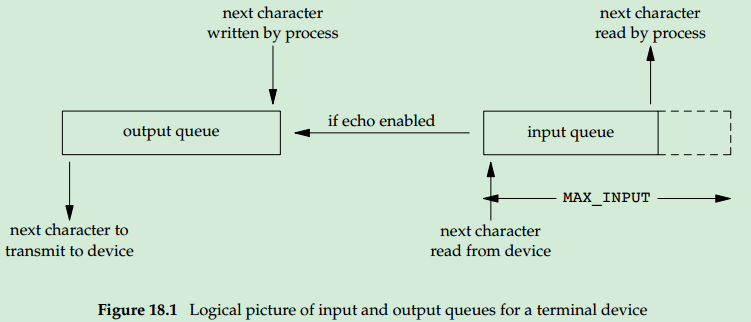
void syslog(int priority, const char \*format, ...);

void closelog(void);

int setlogmask(int maskpri);

Returns: previous log priority mask value

# 终端I/O——<termios.h>、<stdio.h>、<unistd.h>、<sys/ioctl.h>



struct termios

{

tcflag\_t c\_iflag; /\* input flags \*/

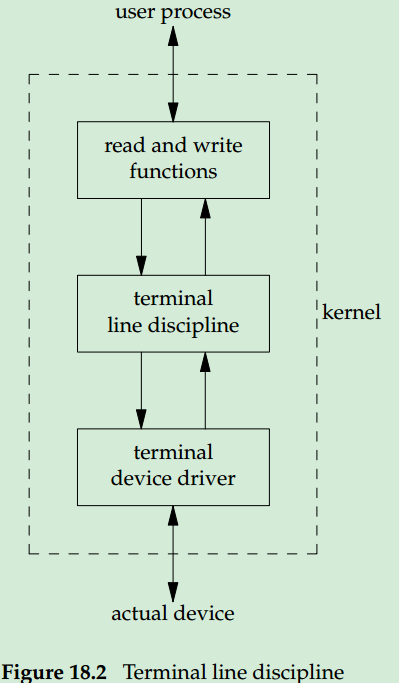
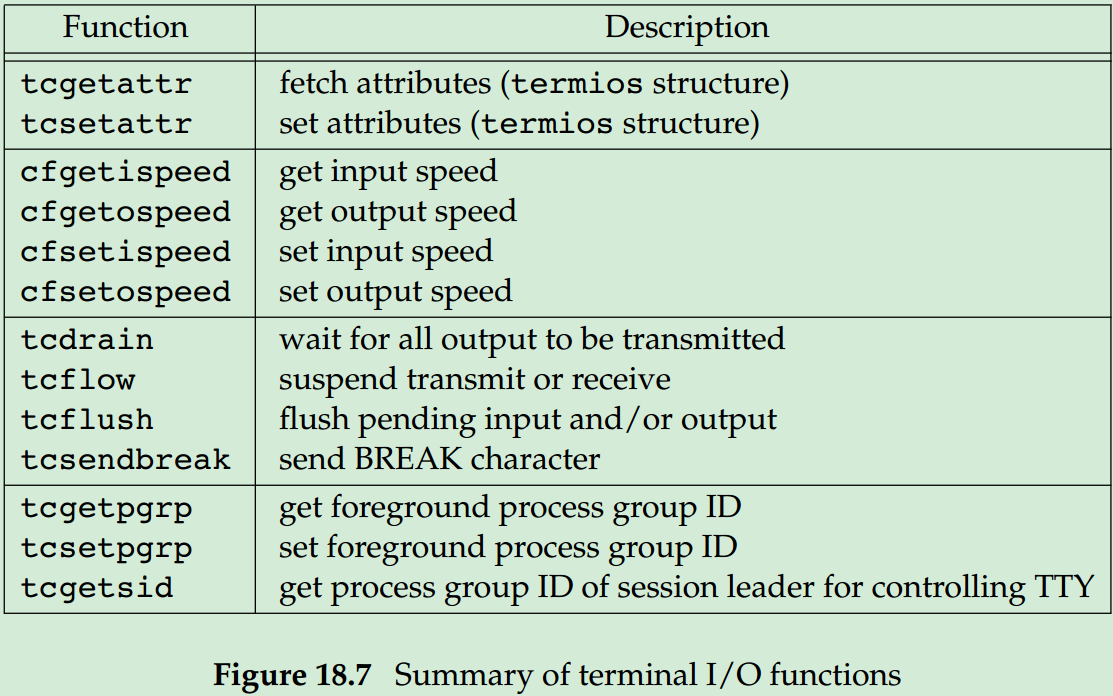
tcflag\_t c\_oflag; /\* output flags \*/

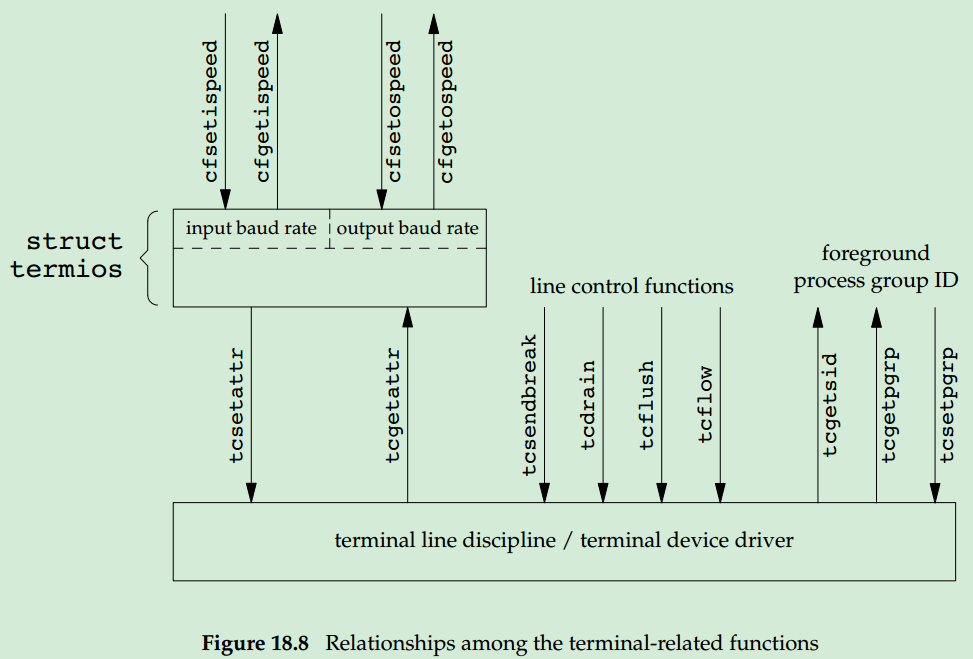
tcflag\_t c\_cflag; /\* control flags \*/

tcflag\_t c\_lflag; /\* local flags \*/

cc\_t c\_cc[NCCS]; /\* control characters \*/

};



读/写属性：int tcgetattr(int fd, struct termios \*termptr);

int tcsetattr(int fd, int opt, const struct termios \*termptr);

opt: TCSANOW, TCSADRAIN, TCSAFLUSH

Both return: 0 if OK, −1 on error

读/写波特率：speed\_t cfgetispeed(const struct termios \*termptr);

speed\_t cfgetospeed(const struct termios \*termptr);

Both return: baud rate value

int cfsetispeed(struct termios \*termptr, speed\_t speed);

int cfsetospeed(struct termios \*termptr, speed\_t speed);

Both return: 0 if OK, −1 on error

输入/出控制函数：int tcdrain(int fd);——waits for all output to be transmitted

int tcflow(int fd, int action);——control over both input and output flow control

int tcflush(int fd, int queue);——throw away input/output buffer

int tcsendbreak(int fd, int duration);——transmits a continuous stream of zero bits for a specified duration .

action: TCOOFF, TCOON, TCIOFF, TCION;queue: TCIFLUSH, TCOFLUSH, TCIOFLUSH; duration:0,0.25~0.5s

All four return: 0 if OK, −1 on error

读终端名：char \*ctermid(char \*ptr);

Ptr：point to an array

Returns: pointer to name of controlling terminal，on success, pointer to empty string on error

判断为终端：int isatty(int fd);

Returns: 1 (true) if terminal device, 0 (false) otherwise

终端路径：char \*ttyname(int fd);

Returns: pointer to pathname of terminal, NULL on error

写窗口大小：ioctl(fd, TIOCGWINSZ, (char \*) & winsize)

Return:0 if OK, -1 on error

struct winsize

{

unsigned short ws\_row; /\* rows, in characters \*/

unsigned short ws\_col; /\* columns, in characters \*/

unsigned short ws\_xpixel; /\* horizontal size, pixels (unused) \*/

unsigned short ws\_ypixel; /\* vertical size, pixels (unused) \*/

};

# 伪终端——<fcntl.h>、<stdlib.h>

建主终端：int posix\_openpt(int oflag);

oflag:similar to open function

Returns: file descriptor of next available PTY master if OK, −1 on error

写权限: int grantpt(int fd);——0x0620

int unlockpt(int fd);——grant access to the slave pseudo terminal device

Both return: 0 on success, ?1 on error

建从终端：char \*ptsname(int fd);

Returns: pointer to name of PTY slave if OK, NULL on error

设置终端原则：ioctl(fds, I\_FIND, "ldterm")——make the terminal act like a real terminal

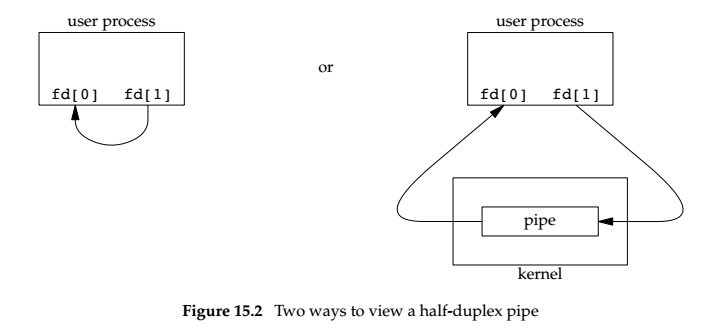
Return :>0 if ok,-1 on error ,0 not set up

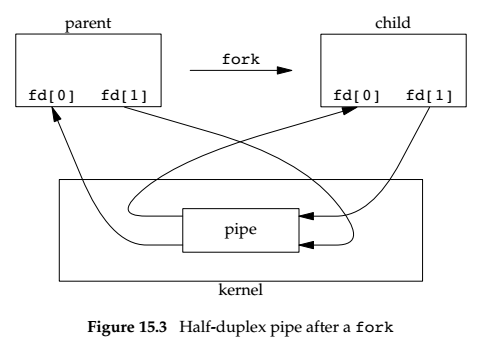
# 进程间通信——<unistd.h>、<sys/ipc.h>、<sys/msg.h>、 <sys/sem.h>、<sys/shm.h>、<semaphore.h>

建立pipe：int pipe(int fd[2]);

fd[0] is open for reading，fd[1] is open for writing

Returns: 0 if OK, −1 on error





开/关命令pipe：FILE \*popen(const char \*cmdstring, const char \*type);

type:”r” the file pointer is connected to the standard output of cmdstring ,“w” stdin

Returns: file pointer if OK, NULL on error

int pclose(FILE \*fp);

Returns: termination status of cmdstring, or −1 on error

创建fifo: int mkfifo(const char \*path, mode\_t mode);——the fifo without O\_NONBLOCK

int mkfifoat(int fd, const char \*path, mode\_t mode);

mode: the same as for the open function

Both return: 0 if OK, −1 on error

创建key：key\_t ftok(const char \*path, int id);

Returns: key if OK, (key\_t)−1 on error,key\_t = IPC\_PRIVATE, guarantees that the server creates a new IPC

struct ipc\_perm

{

uid\_t uid; /\* owner’s effective user ID \*/

gid\_t gid; /\* owner’s effective group ID \*/

uid\_t cuid; /\* creator’s effective user ID \*/

gid\_t cgid; /\* creator’s effective group ID \*/

mode\_t mode; /\* access modes \*/

...

};

创建MQ：int msgget(key\_t key, int flag);

Returns: message queue ID if OK, −1 on error

struct msqid\_ds

{

struct ipc\_perm msg\_perm; /\* see Section 15.6.2 \*/

msgqnum\_t msg\_qnum; /\* # of messages on queue \*/

msglen\_t msg\_qbytes; /\* max # of bytes on queue \*/

pid\_t msg\_lspid; /\* pid of last msgsnd() \*/

pid\_t msg\_lrpid; /\* pid of last msgrcv() \*/

time\_t msg\_stime; /\* last-msgsnd() time \*/

time\_t msg\_rtime; /\* last-msgrcv() time \*/

time\_t msg\_ctime; /\* last-change time \*/

...

};

设置MQ: int msgctl(int msqid, int cmd, struct msqid\_ds \*buf );

Returns: 0 if OK, −1 on error

发送消息：int msgsnd(int msqid, const void \*ptr, size\_t nbytes, int flag);

Ptr:points to mymesg type;flag: IPC\_NOWAIT

Returns: 0 if OK, −1 on error

struct mymesg

{

long mtype; /\* positive message type \*/

char mtext[512]; /\* message data, of length nbytes \*/

};

接收消息：ssize\_t msgrcv(int msqid, void \*ptr, size\_t nbytes, long type, int flag);

Type: specify which message we want; flag: IPC\_NOWAIT

Returns: size of data portion of message if OK, −1 on error

建立信号集：int semget(key\_t key, int nsems, int flag);

Returns: semaphore ID if OK, −1 on error

struct semid\_ds

{

struct ipc\_perm sem\_perm; /\* see Section 15.6.2 \*/

unsigned short sem\_nsems; /\* # of semaphores in set \*/

time\_t sem\_otime; /\* last-semop() time \*/

time\_t sem\_ctime; /\* last-change time \*/

...

};

Struct semaphore

{

unsigned short semval; /\* semaphore value, always >= 0 \*/

pid\_t sempid; /\* pid for last operation \*/

unsigned short semncnt; /\* # processes awaiting semval>curval \*/

unsigned short semzcnt; /\* # processes awaiting semval==0 \*/

...

};

操作信号集：int semctl(int semid, int semnum, int cmd, ... /\* union semun arg \*/ );

Returns: (see following)

union semun

{

int val; /\* for SETVAL \*/

struct semid\_ds \*buf; /\* for IPC\_STAT and IPC\_SET \*/

unsigned short \*array; /\* for GETALL and SETALL \*/

};

操作多个信号：int semop(int semid, struct sembuf semoparray[], size\_t nops);

Returns: 0 if OK, −1 on error

struct sembuf

{

unsigned short sem\_num; /\* member # in set (0, 1, ..., nsems-1) \*/

short sem\_op; /\* operation (negative, 0, or positive) \*/

short sem\_flg; /\* IPC\_NOWAIT, SEM\_UNDO \*/

}

建立共享内存：int shmget(key\_t key, size\_t size, int flag);

size:size of bytes;

Returns: shared memory ID if OK, −1 on error

struct shmid\_ds

{

struct ipc\_perm shm\_perm; /\* see Section 15.6.2 \*/

size\_t shm\_segsz; /\* size of segment in bytes \*/

pid\_t shm\_lpid; /\* pid of last shmop() \*/

pid\_t shm\_cpid; /\* pid of creator \*/

shmatt\_t shm\_nattch; /\* number of current attaches \*/

time\_t shm\_atime; /\* last-attach time \*/

time\_t shm\_dtime; /\* last-detach time \*/

time\_t shm\_ctime; /\* last-change time \*/

...

};

操作共享内存：int shmctl(int shmid, int cmd, struct shmid\_ds \*buf );

Returns: 0 if OK, −1 on error

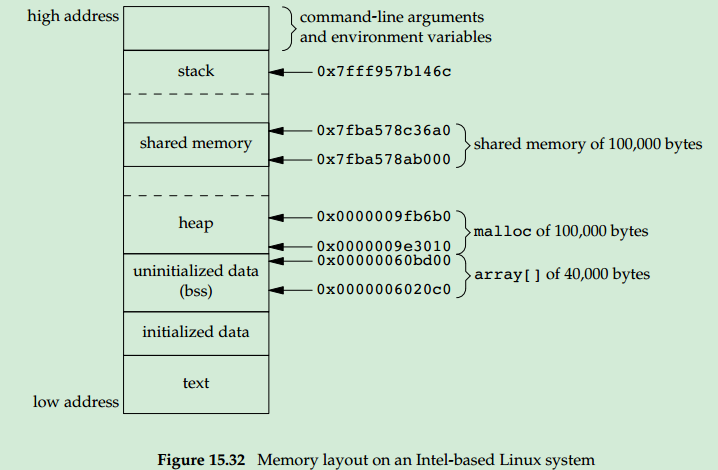
获得共享内存地址：void \*shmat(int shmid, const void \*addr, int flag);

Addr:0

Returns: pointer to shared memory segment if OK, −1 on error

释放共享内存：int shmdt(const void \*addr);

Returns: 0 if OK, −1 on error



开/关信号： sem\_t \*sem\_open(const char \*name, int oflag, ... /\* mode\_t mode, unsigned int value \*/ );

Returns: Pointer to semaphore if OK, SEM\_FAILED on error

int sem\_close(sem\_t \*sem);

Returns: 0 if OK, −1 on error

销毁信号：int sem\_unlink(const char \*name);

Returns: 0 if OK, −1 on error

等待信号：int sem\_trywait(sem\_t \*sem);——no block if the semaphore count is 0

int sem\_wait(sem\_t \*sem);——block if the semaphore count is 0

Both return: 0 if OK, −1 on error

# POSIX 消息队列——<mqueue.h>

需要注意以下几点：

1、消息队列的名字只能以一个 '/'开头，名字中不能包含其他的'/'

2、mq\_receive() 的第三个参数表示读取消息的长度，不能小于能写入队列中消息的最大大小，即一定要大于等于该队列的 mq\_attr 结构中 mq\_msgsize 的大小。

3、消息的优先级：它是一个小于 MQ\_PRIO\_MAX 的数，数值越大，优先级越高。 POSIX 消息队列在调用 mq\_receive 时总是返回队列中最高优先级的最早消息。如果消息不需要设定优先级，那么可以在 mq\_send 是置 msg\_prio 为 0， mq\_receive 的 msg\_prio 置为 NULL。

4、默认情况下mq\_send和mq\_receive是阻塞进行调用，可以通过mq\_setattr来设置为O\_NONBLOCK，如：

struct mq\_attr new\_attr;

mq\_getattr(mqID, &new\_attr);//获取当前属性

new\_attr.mq\_flags = O\_NONBLOCK;//设置为非阻塞

mq\_setattr(mqID, &new\_attr, NULL)//设置属性

**创建**：mqd\_t mq\_open(const char \*name,int oflag,int mode,mq\_addr \*attr);

Name：消息队列的名字字符串，必须以’/’开头，否则会出错。

Oflag: 表示打开的方式，

1．首先必须说明读写方式，可以使以下的值之一：

O\_RDONLY：建立的队列是只读的

O\_WRONLY： 建立的队列是只写的

O\_RDWR：建立的队列是可读可写

2．必须有O\_CREATE,说明是创建消息队列。

3．还有可选的选项：

O\_NONBLOCK：说明在创建的队列上发送和接收消息时，如果没有资源，不会等待，之间返回，如果不设置这个选项，缺省是会等待。

O\_EXCL：在创建队列时，检测要创建的队列的名字是否已经存在了，如果已存在，函数会返回出错

可以以或的方式形成Oflag，例如：O\_RDWR|O\_CREAT|O\_EXCL

Mode：是一个可选参数，在 oflag 中含有 O\_CREAT 标志且消息队列不存在时，才需要提供该参数。表示默认的访问权限，这个权限和文件访问的权限是相同的，取值也相同。

    mode取值如下：

|  |  |
| --- | --- |
| **mode取值** | **含义** |
| **S\_IRUSR** | 文件所有者的读权限 |
| **S\_IWUSR** | 文件安所有者的写权限 |
| **S\_IXUSR** | 文件所有者的执行权限 |
| **S\_IRGRP** | 文件所有者同组用户的读权限 |
| **S\_IWGRP** | 文件所有者同组用户的写权限 |
| **S\_IXGRP** | 文件所有者同组用户的执行权限 |
| **S\_IROTH** | 其他用户的读权限 |
| **S\_IWOTH** | 其他用户的写权限 |
| **S\_IXOTH** | 其他用户的执行权限 |

Mode可以由多个值组合而成，如：S\_IRUSR|S\_IWUSR，队列的所有者有读和写的权限。

Attr：指向结构struct mq\_attr的指针。我们可以在创建队列时通过这个结构设置队列的最大消息数和每个消息的最大长度。

struct mq\_attr

{

long mq\_flags; // 0或者O\_NONBLOCK，说明是否等待

long mq\_maxmsg; //队列中包含的消息数的最大限制数

long mq\_msgsize; //每个消息大小的最大限制数

long mq\_curmsgs; //当前队列中的消息数

}

mq\_maxmsg 和 mq\_msgsize 属性只能在创建消息队列时通过 mq\_open 来设置。 mq\_open 只会设置该两个属性，忽略另外两个属性 。 mq\_curmsgs 属性只能被获取而不能被设置。

当attr参数为NULL时则使用linux的默认值msg\_max ,msgsize\_max 。

而使用mq\_open创建一个新的队列时，attr只能给它指定mq\_maxmsg,mq\_msgsize这两个属性。mq\_open忽略 attr结构中的另外两个成员。

在创建过程中需要注意的是，指定的这两个属性都必须小于等于msg\_max或者msgsize\_max的。查看系统中消息队列的这两个限制值的方法是：

cat /proc/sys/fs/mqueue/msg\_max

cat /proc/sys/fs/mqueue/msgsize\_max

系统默认值有些时候不够大，需要我们对这个限制数进行修改，方法如下：

修改/etc/sysctl.conf 在这个文件中添加 #mqueue max fs.mqueue.msg\_max=100 fs.mqueue.msgsize\_max=9000 fs.mqueue.queues\_max=520 保存好后重启系统就好了。

返回值：函数的返回值是mqd\_t类型，是消息队列的描述符。这实际是一个整数。当函数执行出错时，会返回-1.

要注意的是，描述符对于不同的进程/线程是不一样的，因此不同的进程/线程不能使用其他进程/线程的描述符。

**开**：mqd\_t mq\_open(const char \*name,int oflag,);

Name：消息队列的名字字符串，必须以’/’开头，否则会出错。

Oflag: 表示打开的方式，O\_RDONLY、O\_WRONLY、O\_RDWR

Return：函数的返回值是mqd\_t类型，是消息队列的描述符。在发送和接收消息队列的函数中需要这个描述符来指定是哪个消息队列。这个值实际是一个整数。当函数执行出错时，会返回-1.

要注意的是，描述符对于不同的进程/线程是不一样的，因此不同的进程/线程不能使用其他进程/线程的描述符。

**读属性**：mqd\_t mq\_getattr(mqd\_t mqdes, struct mq\_attr \*attr);

**写属性**：mqd\_t mq\_setattr(mqd\_t mqdes, struct mq\_attr \*newattr, struct mq\_attr \*oldattr);

**发消息：**int mq\_send(mqd\_t mqdes, const char \*ptr, size\_t len, unsigned int prio);

mqdes: 打开消息队列时获得的描述符。

ptr: 指向发送缓冲区的指针，发送缓冲区存放了要发送的数据。

Len: 要发送的数据的长度。

prio ：消息的优先级；它是一个小于 MQ\_PRIO\_MAX 的数，数值越大，优先级越高。 POSIX 消息队列在调用 mq\_receive 时总是返回队列中 最高优先级的最早消息 。如果消息不需要设定优先级，那么可以在 mq\_send 是置 prio 为 0 ， mq\_receive 的 prio 置为 NULL 。

return：0 if OK, -1 on error.

**收消息**：ssize\_t mq\_receive(mqd\_t mqdes, char \*ptr, size\_t len, unsigned int \*prio);

mqdes: 打开消息队列时获得的描述符。

ptr: 指向接收缓冲区的指针。接收缓冲区用来存放收到的消息。

Len: 接收缓冲区的长度。 len不能小于mq\_msgsize，否则会返回EMSGSIZE

prio ：消息的优先级；它是一个小于 MQ\_PRIO\_MAX 的数，数值越大，优先级越高。 POSIX 消息队列在调用 mq\_receive 时总是返回队列中 最高优先级的最早消息 。如果消息不需要设定优先级，那么可以在 mq\_send 是置 prio 为 0 ， mq\_receive 的 prio 置为 NULL 。

返回值：

return：0 if OK, -1 on error.

**关闭：**mqd\_t mq\_close(mqd\_t mqdes);——关闭消息队列，但不能删除它

return：0 if OK, -1 on error.

**删**：mqd\_t mq\_unlink(const char \*name);

return：0 if OK, -1 on error.

# 正则表达式——<regex.h>

**一、行定位符（^和$）**

　　行定位符就是用来描述字串的边界。“^”表示行的开始；“$”表示行的结尾。如：

　　^tm : 该表达式表示要匹配字串tm的开始位置是行头，如tm equal Tomorrow Moon就可以匹配

　　tm$ : 该表达式表示要匹配字串tm的位置是行尾，Tomorrow Moon equal tm匹配。

　　如果要匹配的字串可以出现在字符串的任意部分，那么可以直接   写成 ：tm

二、单词定界符（\b、\B）

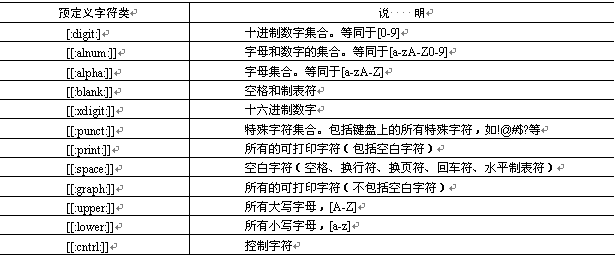
　　单词分界符\b，表示要查找的字串为一个完整的单词。如：\btm\b

　　还有一个大写的\B，意思和\b相反。它匹配的字串不能是一个完整的单词，而是其他单词或字串的一部分。如：\Btm\B

**三、字符类（[ ]）**

　　正则表达式是区分大小写的，如果要忽略大小写可使用方括号表达式“[]”。只要匹配的字符出现在方括号内，即可表示匹配成功。但要注意：一个方括号只能匹配一个字符。例如，要匹配的字串tm不区分大小写，那么该表达式应该写作如下格式：[Tt][Mm]

　　POSIX风格的预定义字符类如表所示：



**四、选择字符（|）**

 　　还有一种方法可以实现上面的匹配模式，就是使用选择字符（|）。该字符可以理解为“或”，如上例也可以写成 (T|t)(M|m)，该表达式的意思是以字母T或t开头，后面接一个字母M或m。

　　使用“[]”和使用“|”的区别在于“[]”只能匹配单个字符，而“|”可以匹配任意长度的字串。如果不怕麻烦，上例还可以写为 ：TM|tm|Tm|tM

**五、连字符（-）**

　　变量的命名规则是只能以字母和下划线开头。但这样一来，如果要使用正则表达式来匹配变量名的第一个字母，要写为 ：[a,b,c,d…A,B,C,D…]

　　这无疑是非常麻烦的，正则表达式提供了连字符“-”来解决这个问题。连字符可以表示字符的范围。如上例可以写成 ：[a-zA-Z]

**六、排除字符（[^]）**

　　上面的例子是匹配符合命名规则的变量。现在反过来，匹配不符合命名规则的变量，正则表达式提供了“^”字符。这个元字符在前面出现过，表示行的开始。而这里将会放到方括号中，表示排除的意思。

　　例如：[^a-zA-Z]，该表达式匹配的就是不以字母和下划线开头的变量名。

**七、限定符（? \* + {n,m}）**

　　对于重复出现字母或字串，可以使用限定符来实现匹配。限定符主要有6种，如表所示：



**八、点号字符（.）**

　　点字符（.）可以匹配出换行符外的任意一个字符。

　　注意：是除了换行符外的、任意的一个字符。如匹配以s开头、t结尾、中间包含一个字母的单词。

　　格式如下： ^s.t$，匹配的单词包括：sat、set、sit等。

　　再举一个实例，匹配一个单词，它的第一个字母为r，第3个字母为s，最后一个字母为t。能匹配该单词的正则表达式为：^r.s.\*t$

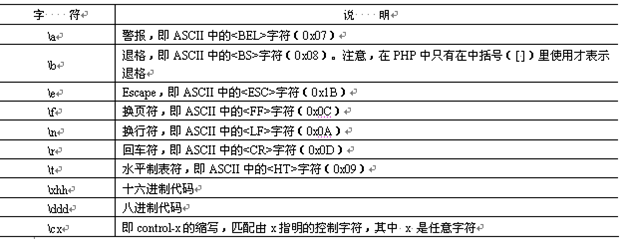
**九、转义字符（\）**

　　正则表达式中的转移字符（\）和PHP中的大同小异，都是将特殊字符（如“.”、“?”、“\”等）变为普通的字符。举一个IP地址的实例，用正则表达式匹配诸如127.0.0.1这样格式的IP地址。如果直接使用点字符，格式为：[0-9]{1,3}(.[0-9]{1,3}){3}

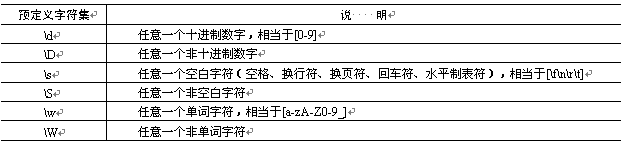
　　这显然不对，因为“.”可以匹配一个任意字符。这时，不仅是127.0.0.1这样的IP，连127101011这样的字串也会被匹配出来。所以在使用“.”时，需要使用转义字符（\）。修改后上面的正则表达式格式为： [0-9]{1,3}(\.[0-9]{1,3}){3}

**十、反斜线（\）**

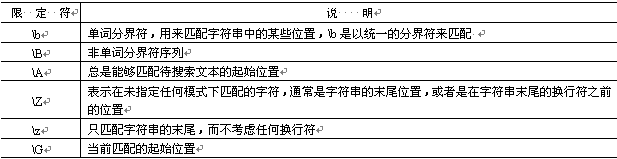
　　除了可以做转义字符外，反斜线还有其他一些功能。反斜线可以将一些不可打印的字符显示出来，如表所示：



　　还可以指定预定义字符集，如表所示：



　　反斜线还有一种功能，就是定义断言，其中已经了解过了\b、\B，其他如表所示：



**十一、括号字符（()）**

　　小括号字符的第一个作用就是可以改变限定符的作用范围，如“|”、“\*”、“^”等。来看下面的一个表达式。

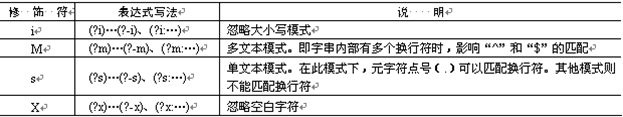
　　(thir|four)th，这个表达式的意思是匹配单词thirth或fourth，如果不使用小括号，那么就变成了匹配单词thir和fourth了。

　　小括号的第二个作用是分组，也就是子表达式。如(\.[0-9]{1,3}){3}，就是对分组(\.[0-9]{1,3})进行重复操作。后面要学到的反向引用和分组有着直接的关系。

**十二、反向引用**

**十三、模式修饰符**

　　模式修饰符的作用是设定模式。也就是规定正则表达式应该如何解释和应用。

　　不同的语言都有自己的模式设置，PHP中的主要模式如表所示：　　

# Ubuntu 命令

**Apt-xxx**

apt-cache search package 搜索包

apt-cache show package 获取包的相关信息，如说明、大小、版本等

sudo apt-get install package 安装包

sudo apt-get install package - - reinstall 重新安装包

sudo apt-get -f install 修复安装"-f = ——fix-missing"

sudo apt-get remove package 删除包

sudo apt-get remove package - - purge 删除包，包括删除配置文件等

sudo apt-get update 更新源

sudo apt-get upgrade 更新已安装的包

sudo apt-get dist-upgrade 升级系统

sudo apt-get dselect-upgrade 使用 dselect 升级

apt-cache depends package 了解使用依赖

apt-cache rdepends package 是查看该包被哪些包依赖

sudo apt-get build-dep package 安装相关的编译环境

apt-get source package 下载该包的源代码：如内核：apt-get source linux-image-$(uname -r)

sudo apt-get clean && sudo apt-get autoclean 清理无用的包

sudo apt-get check 检查是否有损坏的依赖

**tar: 压缩解压**

-c ：建立一个压缩文件的参数指令(create 的意思)；

-x ：解开一个压缩文件的参数指令！

-t ：查看 tarfile 里面的文件！

特别注意，在参数的下达中， c/x/t 仅能存在一个！不可同时存在！

tar -cvf /tmp/etc.tar /etc <==仅打包，不压缩！

tar -zcvf /tmp/etc.tar.gz /etc <==打包后，以 gzip 压缩

tar -jcvf /tmp/etc.tar.bz2 /etc <==打包后，以 bzip2 压缩

du:查看文件/夹大小

-h: print sizes in human readable format (e.g., 1K 234M 2G)

-s: summarize, display only a total for each argument

统计文件数量

ls -lR|grep "^-"|wc –l //文件数，递归

ls -lR|grep "^d"|wc –l // 目录总数，递归

wc: 查看文件print newline, word, and byte counts for each file

-c, --bytes -l, --lines -L --max-line-length -w, --words -m, --chars

**Dmesg命令**——对应log：/var/log/dmesg

dmesg | grep sda ：列出所有被检测到的硬件

dmesg | head -20：只输出dmesg命令的前20行日志

dmesg | tail -20：只输出dmesg命令最后20行日志

dmesg | grep -i usb ：grep 命令 的‘-i’选项表示忽略大小写，搜索包含特定字符串的被检测到的硬件

shell命令的输出追加到文件中（>>）:ls >> file

shell命令的覆盖输出到文件中(>): ls >file

获取系统位数：getconf LONG\_BIT

**网络命令**

断开/连接网卡: sudo nmcli nm wifi off/on or ifconfig wlan0 down/up

查看网卡已保存的连接：nmcli con

删除网卡已保存的连接：nmcli con delete uuid cecee8d9-4dad-49bb-8f5d-f2d59499b65e

查看硬件ap：nmcli dev wifi

查找所有可用的wifi接入点，记录接入点名称（essid）：iwlist scan|egrep 'ESSID|Link|Signal'

**抓包tcpdump**

显示通过网络传输到本系统的 TCP/IP 以及其他网络的数据包

-w: save the packet data to a file 🡪format:\*.pcap

-r: read from a saved packet file 🡪format:\*.pcap

-c: continue capturing packets

-i :Listen on interface,如：-i wlan0

-tttt: Print a timestamp in default format proceeded by date on each dump line

-vv: Even more verbose output

src:源ip dst:目的ip host:ip net:网络/端口 port:端口 portrange:端口范围 ——使用and(与)、or（或）、not（非），如：tcpdump src 192.168.1.100 && port 22 -w ssh\_packets

查看整个网络的数据包：tcpdump net 192.168.1.0/24

根据 IP 地址查看报文：tcpdump host 192.168.1.100 、tcpdump src 192.168.1.100、tcpdump dst 192.168.1.100

查看端口收到、发出的数据:tcpdump udp -i wlan0 -vv -n |grep 3001

CAN命令：

设置位速率：ip link set can0 type can bitrate 125000

读设置：ip -details link show can0

开：ifconfig can0 up

关：ifconfig can0 down

查启动状态：ip -details -statistics link show can0

**用户密码**

1.创建账户：sudo useradd username

2修改密码：sudo passwd username

3.删除账户：userdel -r newuser

**文件传输命令**

Scp

**服务命令**

**路径命令**

1.当前命令：pwd

2.命令的绝对命令：which ls

**查看linux命令源代码**

Cscope，ls命令

1. 搜索命令所在包， which ls—> /bin/ls
2. 搜索该软件所在包，dpkg -S /bin/ls—> coreutils: /bin/ls(实现在包coreutils中)
3. 获取软件包，sudo apt-get source coreutils
4. 获取该包的源代码然后解压，sudo apt-get source coreutils ; cd /usr/src/coreutils-XXX #XXX表示版本号 ; sudo tar zxvf coreutils-XXX.tar.gz
5. 或者只下载源码，然后手动打补丁再解压: sudo apt-get -d source coreutils ;cd /usr/src ;tar zxvf coreutils-XXX.tar.gz ;gzip -d coreutils-XXX.diff.gz #这一步会生成coreutils-XXX.diff文件 ;patch -p0 < coreutils-XXX.diff ;cd coreutils-XXX ;tar zxvf coreutils-XXX.tar.gz

读网络适配器及统计信息：cat proc/net/dev

* bytes: The total number of bytes of data transmitted or received by the interface.（接口发送或接收的数据的总字节数）
* packets: The total number of packets of data transmitted or received by the interface.（接口发送或接收的数据包总数）
* errs: The total number of transmit or receive errors detected by the device driver.（由设备驱动程序检测到的发送或接收错误的总数）
* drop: The total number of packets dropped by the device driver.（设备驱动程序丢弃的数据包总数）
* fifo: The number of FIFO buffer errors.（FIFO缓冲区错误的数量）
* frame: The number of packet framing errors.（分组帧错误的数量）
* colls: The number of collisions detected on the interface.（接口上检测到的冲突数）
* compressed: The number of compressed packets transmitted or received by the device driver. (This appears to be unused in the 2.2.15 kernel.)（设备驱动程序发送或接收的压缩数据包数）
* carrier: The number of carrier losses detected by the device driver.（由设备驱动程序检测到的载波损耗的数量）
* multicast: The number of multicast frames transmitted or received by the device driver.（设备驱动程序发送或接收的多播帧数）

**查看程序占用内存**：

top | grep AppName查看程序的情况

ps -aux | grep process\_name

cat /proc/pid/status

# 工具函数——<math.h>、 <stdlib.h>

int abs(int i); // 处理int类型的取绝对值

double fabs(double i); //处理double类型的取绝对值

float fabsf(float i); /处理float类型的取绝对值

# 小知识

**问题1：可重入与线程安全的区别？**

**解决1：**可重入：任何时候都可以重复调用，即,**在并发和自己调用自己都能得到正确的结果。**如果一个函数只访问自己的局部变量或参数，称为可重入函数。

线程安全：仅当被多个并发线程反复调用时，它会一直产生正确的结果。**仅仅支持并发就行。**

**不可重入特点：**

如果一个函数符合以下条件之一的，则是不可重入的：

（1）调用了malloc/free函数，因为malloc函数是用全局链表来管理堆的。

（2）调用了标准I/O库函数，标准I/O库的很多实现都以不可重入的方式使用全局数据结构。

（3）可重入体内使用了静态的数据结构。

# End