linux-应用函数

总共分为12部分分别是：进程、线程、消息队列、信号量集、共享内存、PGSQL编程、MYSQL编程、网络编程、文件访问、标准I/O、系统数据文件和信息、信号

（一）进程

1. 进程ID为0的进程通常是调度进程，常常被称为交换进程

进程ID为1的进程通常是init进程，在自举过程结束时由内核调用

进程ID为2的进程页守护进程，负责支持虚拟存储系统的分页操作

2. pid_t getpid(void); 返回值：调用进程的进程ID #include

3. pid_t getppid(void); 返回值：调用进程的父进程ID

4. uid_t getuid(void); 返回值：调用进程的实际用户ID

5. uid_t geteuid(void); 返回值：调用进程的有效用户ID

6. gid_t getgid(void); 返回值：调用进程的实际组ID

7. gid_t getegid(void); 返回值：调用进程的有效组ID

8. pid_t fork(void);创建子进程，返回值：子进程返回0，父进程返回子进程ID，出错-1

9. #include pid_t wait(int *statloc);//statloc 保存进程终止状态的指针

10. #includepid_t waitpid(pid_t pid,int *statloc,int options);

pid ==-1 等待任一子进程

pid >0 等待其子进程ID与pid相等的子进程

pid == 0 等待其组ID等于调用进程组ID的任一子进程

pid <-1 IDpidp>

options：

WCONTINUED 若实现支持作业控制，那么由pid指定的任一子进程在暂停后已经继续，但其状态尚未报告，则返回其状态

WNOHANG 若由pid指定的子进程并不是立即可用的，则waitpid阻塞，此时其返回0

WUNTRACED 若实现支持作业控制，而由pid指定的任一子进程已处于暂停状态，并且其状态自暂停以来还未报告过，则返回其状态

11.#include int setuid(uid_t uid); 设置实际实际用户ID和有效用户ID；

int setgid(gid_t gid); 设置实际组ID和有效组ID；成功返回0，错误-1

12.#includeint system(const char *cmdstring)

system返回值如下

-1出现错误

0调用成功但是没有出现子进程

>0 成功退出的子进程的id

（二）线程

1. #include int pthread_equal(pthread_t tid1, pthread_t tid2);

//相等返回非0，否则返回0

2. pthread_t pthread_self(void);返回调用线程的ID

3. int pthread_create(pthread_t *restrict tidp,

const pthread_attr_t *restrict attr, void *(*start_rtn)(void), void *restrict arg)；

创建线程：成功返回0，否则返回错误编号

4. void pthread_exit(void *rval_ptr);//终止线程

5. int pthread_join(pthread_t thread, void **rval_ptr);

//自动线程置于分离状态，以恢复资源。成功返回0，否则返回错误编号

6. int pthread_cancel(pthread_t tid);

//请求取消同一进程中的其他线程;成功返回0，否则返回错误编号

7. void pthread_cleanup_push(void (*rtn)(void *), void *arg);

//建立线程清理处理程序

8. void pthread_cle

anup_pop(int execute);//调用建立的清理处理程序

9. int pthread_detach(pthread_t tid);//使线程进入分离状态，已分离也不出错

10.int pthread_mutex_init(pthread_mutex_t *restrict mutex,

const pthread_nutexattr_t *restrict attr)//初始化互斥量;成功0，失败返回错误编号

11.int pthread_mutex_destroy(pthread_mutex_t *mutex);

//若有调用malloc动态分配内存则用该函数释放；成功0，失败返回错误编号

12.int pthread_mutex_lock(pthread_mutex_t *mutex);//锁住互斥量

int pthread_mutex_trylock(pthread_mutex_t *mutex);//尝试上锁

int pthread_mutex_unlock(pthread_mutex_t *mutex);//解锁

成功返回0，否则返回错误编号

13.int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr)//初始化读写锁

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);//释放资源，在释放内存之前使用

成功返回0，否则返回错误编号

14.int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);//在读模式下锁定读写锁

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);//在写模式下锁定读写锁

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);//锁住读写锁

15.int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);//尝试在读模式下锁定读写锁

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);//尝试在写模式下锁定读写锁

成功返回0，否则返回错误编号

16.int pthread_cond_init(pthread_cond_t *restrict cond, pthread_condattr_t * restrict attr)

//初始化条件变量

int pthread_cond_destroy(pthread_cond_t *cond);//去除初始化条件变量

成功返回0，否则返回错误编号

17.int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex)

int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex,

const struct timespec *restrict timeout);

//等待条件变为真，如果在给定的时间内条件不能满足，那么会生成一个代表出错码的返回变量 ；成功返回0，错误返回错误编号

18.int pthread_cond_signal(pthread_cond_t *cond);//唤醒等待该条件的某个线程

int pthread_cond_broadcast(pthread_cond_t *cond)//唤醒等待该条件的所有线程

19.int pthread_attr_init(pthread_attr_t *attr);//初始化线程属性

int pthread_attr_destroy(pthread_attr_t *attr);//释放内存空间（动态分配时调用）

成功返回0，否则返回错误编号

20.int pthread_attr_getdetachstate(const pthread_attr_t *restrict attr, int *detachstate);

//获取线程的分离状态

int pthread_attr_setdetachstate(const pthread_attr_t *restrict attr, int detachstate);

//设置分离状态 PTHREAD_CREATE_DETACHED：以分离状态启动线程

PTHREAD_CREATE_JOINABLE：正常启动线程，应用程序可以获取线程的终止状态

成功返回0，否则返回错误编号

21.int pthread_attr_getstack(const pthread_attr_t *restrict attr,void **rest

rict stackaddr, size_t *restrict stacksize);//获取线程的栈位置

int pthread_attr_setstack(const pthread_attr_t *attr, void *stackaddr, size_t *stacksize)

//设置新建线程的栈位置 ；成功返回0，否则返回错误编号

(三)消息队列

1.每个队列都有一个msqid_ds结构与之相关联：

struct msqid_ds{

struct ipc_perm msg_perm;

msgqnum_t msg_qnum; //消息的数量

msglen_t msg_qbytes; //最大消息的长度

pid_t msg_lspid; //最后一个发送到消息队列的进程ID

pid_t msg_lrpid; //最后一个读取消息的进程ID

time_t msg_stime; //最后一次发送到消息队列的时间

time_t msg_rtime; //最后一次读取消息的时间

time_t msg_ctime; //最后一次改变的时间

。

。

。

};

struct ipc_perm{

uid_t uid;//拥有者有效的用户ID

gid_t gid;//拥有者有效的组ID

uid_t cuid;//创建者有效的用户ID

uid_t cgid;//创建者有效的组ID

mode_t mode; //权限

。

。

}

2.#include int msgget(key_t key, int flag);

//打开一个现存的队列或创建一个新队列；成功返回0，出错返回-1

3.int msgctl(int msqid, int cmd, struct msqid_ds *buf);//对消息队列执行多种操作

cmd 可选：

IPC_STAT 取此消息队列的msqid_ds结构，并将它放在buf指向的结构

IPC_SET：按由buf指向结构中的值，设置与此队列相关结构中的下列四个字段：msg_perm.uid,msg_perm.gid,msg_perm.mode和msg_qbytes.此命令只有下列两种进程才能执行（1）其有效用户ID等于msg_perm.cuid或msg_perm.uid;（2）具有超级用户特权的进程

IPC_RMID：从系统中删除消息队列以及仍在该队列中的所有数据。

成功返回0，失败返回-1

4.int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag)//发送消息到消息队列中

成功返回0， 不成功返回-1并设置errno，错误码：

EACCES 对调用程序来说，调用被否定

EAGAIN 操作会阻塞进程，但(msgflg & IPC_NOWAIT) != 0

EIDRM msqid已经从系统中删除了

EINTR 函数被信号中断

EINVAL 参数msqid无效，消息类型<1 msgszp>

flag可以指定为IPC_NOWAIT 则不会阻塞直接返回EAGAIN

注：参数msgp指向用户定义的缓冲区，他是如下的结构

struct mymsg

{

long mtypes; 消息类型

char *mtext; 消息文本

}mymsg_t

5.ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);//读取消息

成功则返回消息的数据部分的长度，出错则返回-1

type: type==0返回队列中的第一个消息

type>0 返回队列中消息类型为type的第一个消息

type<0 typep>

(四) 信号量

1. 内核为每个信号量集合设置了一个semid_ds结构：

struct demid_ds{

struct ipc_perm se

m_perm;

unsigned short sem_nsems; //信号量的个数

time_t sem_otime; //上一次semop的时间

time_t sem_ctime;//上一次change的时间

。

。

};

2#include. int semget(key_t key, int nsems, int flag);//创建信号量

成功返回一个对应于信号量集标识符的非负整数，不成功返回-1并设置errno，错误码：

EACCES 存在key的信号量，但没有授予权限

EEXIST 存在key的信号量，但是

（ （ semflg & IPC_CREATE ） && (semflg & IPC_EXCL) ） != 0

EINVAL nsems < nsemsp>

ENOENT 不存在key的信号量，而且(semflg & IPC_CTEATE) == 0

ENOSPC 要超出系统范围内对信号量的限制了

功能：

函数返回与参数key相关的信号量集标识符。

如果键值为IPC_PRIVATE,或者semflg&IPC_CREAT非零且没有信号量集或标识符关联于key,那么函数就创建标识符及与之相关的信号量集。

参数nsems指定了集合中信号量元素的个数，可用0到nsems-1的整数来引用信号量集合中的单个信号量元素。

参数semflg指定信号量集的优先级，权限的设置与文件权限设置相同，并可以通过semclt来修改权限值，在使用信号量元素之前，应该用semctl对其进行初始化。

注意：

函数如果试图创建一个已经存在的信号量集，如果semflg值中包含了IPC_CREAT和IPC_EXCL，则失败并设置errno为EEXIST；否则返回一个已经存在的信号量集的句柄。

3.int semctl(int semid, int semnum, int cmd, ...);//信号量控制

如果成功返回一个非负的值，具体返回值取决于cmd的值。

cmd值GETVAL、GETPID、GETNCNT和GETZCNT使semctl返回与cmd相关的值。

如果成功，所有其它的cmd返回0。

如果不成功semctl返回-1并设置errno，必须检测的错误码：

EACCES 对调用程序来说，操作被否定

EINVAL semid的值或cmd的值无效，或者semnum的值为负或者太大

EPERM cmd的值为IPC_RMID或IPC_SET,且调用程序没有所要求的特权

ERANGE cmd为SETVAL或SETALL，而且要设置的值越界了

功能：

函数semctl为信号量集semid的semnum个元素提供了控制操作。参数cmd指定了操作类型，第四个参数arg可选，是否使用取决于cmd的值。

cmd的值：

GETALL 在arg.array中返回信号量集的值

GETVAL 返回一个特定信号量元素的值

GETPID 返回最后一个操纵元素的进程的进程ID

GETNCNT 返回等待元素增加的进程的个数

GETZCNT 返回等待元素变成零的进程的个数

IPC_RMID 删除semid标识的信号量集

IPC_SET 设置来之arg.buf的信号量集的权限

IPC_STAT 将信号量集semid的semid_ds结构成员拷贝到arg.buf中

SETALL 用arg.array来设置信号量集的值

SETVAL 将一个特定的信号量元素的值设定为arg.val

其中有几个命令需要一个arg参数来读取或存储结构

参数arg的类型为union semun,必须要定义这个类型的数据：

union semum

{

int val

struct semid_ds *buf;

unsigned short *array;

}arg;

4. int semop(int semid, struct sembuf *sops, size_t nsops);//信号量的操作

成功返回0，不成功返回-1并设置errno，必须检测的错误码：

E2BIG nsops的值太大

EACCES 对调用程序来说，操作被否定

EAGAIN 操作会阻塞进程但是（sem_flg&IPC_NOWAIT）!= 0

EFBIG 某一个sops条目的sem_num值小于0，或大于信号量集中元素的数目

EIDRM 信号量集标识符semid已经从系统中删除了

EINTR semop被信号中断

EINVAL semid的值无效，或者请求做SEM_UNDO操作的独立信号量集的数量

超出了限制

ENOSPC 已经超出了对请求SEM_UNDO的进程数的限制

ERANGE 操作会造成semval或semadj值得溢出

功能：

semop函数在单个信号量集上原子的执行sops数组中指定的所有操作。如果其中任何一个单独的元素

操作会使进程阻塞，进程就会阻塞而不会执行任何操作。

说明：

结构struct sembuf指定了一个信号量元素操作，包含下列成员。

short sem_num 信号量元素的数量（信号量元素在信号量集中的序号）

short sem_op 要执行的特定元素操作

short sem_flg 为操作指定选项的标志符

sem_op如果是大于零的整数，semop就将这个值与sem_num号信号量元素相加，并唤醒所有等待该元素增加的进程。

如果为零，且信号量元素值不为0，semop就会阻塞调用进程（进程在等待0），并增加等待那个信号量元素值变为零的进程计数。

如果sem_op为负数，那么，如果结果不能为负的话，semop就将sem_op值添加到相应的信号量元素值上去。如果操作可能会使元素值为负，semop就将进程阻塞在使信号量元素值增加的事件上。如果结果值为0，那么semop就唤醒等待0的进程。

（五）共享内存

1.内核为每个共享内存设置了一个 shmid_ds结构,它的成员如下：

Struct shmid_ds{

struct ipc_perm shm_perm; //操作权限结构

size_t shm_segsz; //用字节表示的段的长度

pid_t shm_lpid; //最后一个操作的进程ID

pid_t shm_cpid; //创建者的进程ID

shmatt_t shm_nattch //当前连接的进程数量

time_t shm_atime; //最后一次调用shmat的时间

time_t shm_dtime; //最后一次调用shmdt的时间

time_t shm_ctime; //最后一次调用shmtl的时间

。

。

}

2.include int shmget(key_t key, size_t size, int flag);//创建共享内存

size指的是共享内存段的格式n*sizeof(int)则共享内存段将用来存储int类型数据n个

成功返回一个对应于共享内存段标识符的非负整数，不成功返回-1并设置errno，错误码：

EACCES key的共享标识符存在，但没有授予相关的权限

EEXIST key的共享标识符存在，但((shmflg&IPC_CREAT) && (

shmflg&IPC_EXCL)!=0

EINVAL 要创建共享内存段，但size是无效的

EINVAL 没有共享内存段要创建，但size与系统设置的限制或与key所代表的共享段的长度不相符

ENOENT key的共享内存表示符不存在，但（shmflg&IPC_CREAT）== 0

ENOMEM 没有足够的内存空间来创建指定的共享内存段

ENOSPC 要超出系统范围内对共享标识符的限制了

功能：

shmget函数返回一个与参数key相关的共享内存段标识符。

如果键位IPC_CREAT或者shmflg&IPC_CREAT非零，而且没有共享内存段或标识符与key相关联，函数就创建这个段，共享内存段被初始化为零。

3. int shmctl(int shmid, int cmd, struct shmid_ds *buf);//共享内存的控制

成功返回0，不成功返回-1并设置errno，错误码：

EACCES cmd为IPC_STAT，但是调用程序没有读权限

EINVAL shmid或cmd的值无效

EPERM cmd为IPC_RMID或IPC_SET，调用程序没有正确的权限

cmd值：

IPC_RMID 删除共享内存段，并销毁相应的shmid_ds

IPC_SET 用buf中的值来设置共享内存段shmid的字段值

IPC_STAT 将共享内存段shmid中的当前值拷贝到buf中去

IPC_LOCK 将共享内存段锁定在内存中（只有超级用户可以执行）

IPC_UNLOCK 解锁共享内存

4.void *shmat(int shmid, const void *shmaddr, int shmflg);//共享内存段的连接

成功返回内存段的起始地址，不成功shmat返回-1并设置errno，必须检测的错误码：

EACCES 调用程序的操作权限别否定

EINVLA shmid和shmaddr的无效

EMFILE 连接到进程上的共享内存段的树木超出了限制

ENOMEM 进程数据空间不足以容纳共享内存段

功能：

函数将shmid指定的共享内存段连接到调用进程的地址空间，并为shmid增加shm_nattch的值。

如果shmaddr为0，则此段连接到由内核选择的第一个可用的地址上

如果shmaddr非0，并且没有指定SHM_RND,则此段连接到addr所指定的地址上

如果shmaddr非0，并且指定了SHM_RND，则此段连接到（addr-(addr mod ulus SHMLBA)）所表示的地址上。

5. int shmdt(const void *shmaddr);//分离共享内存

成功返回0，不成功返回-1并设置errno，错误码：

EINVAL shmaddr不对应于共享内存段的起始地址

功能：

用完一个共享内存，调用其来分离共享内存段，并对shm_nattch进行减操作。

最后一个分离共享内存段的进程应该通过调用shmctl来释放共享内存段

（六）PGSQL编程

1.PGconn *PQconnectdb(const char *conninfo);//与数据库服务器建立一个新的连接

conninfo可以包含的内容有：host,hostaddr, port, dbname, user, password, connect_timeout, options,tty,sslmode,requiressl,service

2.//与数据库服务器建立一个新的连接（PQconnectdb的前身）

PGconn *PQsetdbLogin(const char *pghost,//IP

const char *pgport,//断口

const char *pgoptions,// 发送给服务器的命

令行选项

const char *pgtty,//服务器日志的输出方向

const char *dbName,//数据库名

const char *login,//用户

const char *pwd);//密码

3. PGconn *PQsetdb(char *pghost,char *pgport, char *pgoptions,char *pgtty,char *dbName);//与数据库服务器建立一个新的连接

这是一个调用 PQsetdbLogin() 的宏，只是login和pwd参数用空（null ）代替。 提供这个函数是为了与非常老版本的程序兼容。

4. PGconn *PQconnectStart(const char *conninfo);

PostgreSQLPollingStatusType PQconnectPoll(PGconn *conn);

//与数据库服务器建立一个非阻塞的连接

5. PQconninfoOption *PQconndefaults(void);//返回缺省的连接选项

typedef struct PQconninfoOption

{

char *keyword; /* 选项的键字 */

char *envvar; /* 退守的环境变量名 */

char *compiled; /* 退守的编译时缺省值 */

char *val; /* 选项的当前值，或者 NULL */

char *label; /* 连接对话里字段的标识 */

char *dispchar; /* 在连接对话里为此字段显示的字符。

数值有：

"" 原样现实输入的数值

"*" 口令字段 - 隐藏数值

"D" 调试选项 - 缺省的时候不显示 */

int dispsize; /* 对话中字段的以字符计的大小 */

}PQconninfoOption;

6. void PQfinish(PGconn *conn);//关闭连接

7. void PQreset(PGconn *conn);//重建新的连接

8. //以非阻塞模式重置与服务器的通讯端口。

int PQresetStart(PGconn *conn);

PostgreSQLPollingStatusType PQresetPoll(PGconn *conn);

9. char *PQdb(const PGconn *conn);//返回连接的数据库名

10. char *PQuser(const PGconn *conn);//返回连接的用户名

11. char *PQpass(const PGconn *conn);//返回连接密码

12. char *PQhost(const PGconn *conn);//返回连接主机名

13. char *PQport(const PGconn *conn);//返回连接端口

14. char *PQtty(const PGconn *conn);//返回连接的调试控制台TTY

15. char *PQoptions(const PGconn *conn);//返回连接请求传递的命令行选项

16. ConnStatusType PQstatus(const PGconn *conn);//返回连接状态

这个状态可以是一系列值之一。 不过，我们在一个异步连接过程外面只能看到其中的两个：CONNECTION_OK 或 CONNECTION_BAD。一个与数据库的成功的连接返回状态 CONNECTION_OK。 一次失败的企图用状态 CONNECTION_BAD 标识。 通常，一个 OK 状态将保持到 PQfinish，但是一个通讯失败可能会导致状态过早地改变为 CONNECTION_BAD 。这时应用可以试着调用 PQreset 来恢复。

17. PGTransactionStatusType PQtransactionStatus(const PGconn *conn);

//返回当前服务器的事务内状态。

状态可以是 PQTRANS_IDLE （当前空闲）， PQTRANS_ACTIVE （正在处理一个命令），PQTRANS_INTRANS （空闲，在一个合法的事务块内），或者 PQTRANS_INERROR （空闲，在一个失败的事务块内）。如果连接有

问题，则返回 PQTRANS_UNKNOWN。只有在一个查询发送给了服务器并且还没有完成的时候才返回 PQTRANS_ACTIVE。

18. const char *PQparameterStatus(const PGconn *conn, const char *paramName);// 查找服务器的一个当前参数设置

19. int PQprotocolVersion(const PGconn *conn);查询所使用的前/后端协议。

20. int PQserverVersion(const PGconn *conn);//返回服务器的版本

21. char *PQerrorMessage(const PGconn *conn);// 返回连接中操作产生的最近的错误信息。

22. int PQsocket(const PGconn *conn);// 获取与服务器连接的套接字的文件描述符编号。 一个有效的描述符应该是大于或等于 0；结果为 -1 表示当前没有与服务器的连接打开。 （在正常的操作中，这个结果不会改变，但是可能在启动或者重置的过程中变化。）

23. int PQbackendPID(const PGconn *conn);// 返回处理此连接的服务器服务器的进程号 ID（PID）。

24. SSL *PQgetssl(const PGconn *conn);// 返回连接使用的 SSL 结构，或者如果没有使用 SSL 的话返回 NULL。

25. PGresult *PQexec(PGconn *conn, const char *command);// 给服务器提交一条命令并且等待结果。

26. PGresult *PQexecParams(PGconn *conn,

const char *command,

int nParams,

const Oid *paramTypes,

const char * const *paramValues,

const int *paramLengths,

const int *paramFormats,

int resultFormat);

//向服务器提交一条命令并且等待结果，还有额外的传递与 SQL 命令文本独立的参数的能力。

27. 用给定的参数提交请求，创建一个准备好的语句，然后等待结束。

PGresult *PQprepare(PGconn *conn,

const char *stmtName,

const char *query,

int nParams,

const Oid *paramTypes);

28. PGresult *PQexecPrepared(PGconn *conn,

const char *stmtName,

int nParams,

const char * const *paramValues,

const int *paramLengths,

const int *paramFormats,

int resultFormat);

//发送一个请求，执行一个带有给出参数的准备好的语句，并且等待结果。

29. 返回命令的结果状态。

ExecStatusType PQresultStatus(const PGresult *res);

PQresultStatus可以返回下面数值之一：

PGRES_EMPTY_QUERY 发送给服务器的字串是空的

PGRES_COMMAND_OK 成功完成一个不返回数据的命令

PGRES_TUPLES_OK 成功执行一个返回数据的查询查询（比如 SELECT 或者 SHOW）。

PGRES_COPY_OUT （从服务器）Copy Out （拷贝出）数据传输开始

PGRES_COPY_IN Copy In （拷贝入）（到服务器）数据传输开始

PGRES_BAD_RESPONSE 服务器的响应无法理解

PGRES_NONFATAL_ERROR 发生了一个非致命错误（通知或者警告）PGRES_FATAL_ERROR 发生了一个致命错误

30. char *PQresStatus(ExecStatusType st

atus);// PQresStatus 把PQresultStatus返回的枚举类型转换成一个描述状态码的字符串常量。 调用者不应该释放结果。

31. char *PQresultErrorMessage(const PGresult *res);

返回与查询关联的错误信息，或在没有错误时返回一个空字符串。

如果有错误，那么返回的字串将包括一个结尾的新行。调用者不应该直接释放结果。在相关的 PGresult 句柄传递给 PQclear 之后，它会自动释放。

32. char *PQresultErrorField(const PGresult *res, int fieldcode);

// 返回一个独立的错误报告字段。

33. void PQclear(PGresult *res);// PQclear 释放于PGresult相关联的存储空间。 任何不再需要的查询结果在不需要的时候都应该用PQclear释放掉。

34. PGresult* PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);

// 构造一个带有给出的状态的，空的PGresult对象。

35. int PQntuples(const PGresult *res)//返回查询结果里的行（元组）个数。 36. int PQnfields(const PGresult *res);

// 返回查询结果里数据行的数据域 （字段）的个数。

37. char *PQfname(const PGresult *res, int column_number);

// 返回与给出的数据域编号相关联的数据域（字段）的名称。数据域编号从 0 开始。调用者不应该直

接释放结果。在相关联的 PGresult 句柄传递给 PQclear 之后，结果会被自动释放。

38. int PQfnumber(const PGresult *res,const char *column_name);

//返回与给出的数据域名称相关联的数据域（字段）的编号。

39. Oid PQftable(const PGresult *res, int column_number);

//返回我们抓取的字段所在的表的 OID。字段编号从 0 开始。

40. int PQftablecol(const PGresult *res, int column_number);

//返回组成声明的查询结果字段的字段号（在它的表内部）。查询结果字段编号从 0 开始，但是表字段编号不会是 0。

41. int PQfformat(const PGresult *res, int column_number);

//返回说明给出字段的格式的格式代码。0开始

42. Oid PQftype(const PGresult *res, int column_number);

// 返回与给定数据域编号关联的数据域类型。 返回的整数是一个该类型的内部 OID 号。数据域编号从0 开始。

43. int PQfmod(const PGresult *res, int column_number);

// 返回与给定字段编号相关联的类型修饰词。 字段编号从 0 开始。

44. int PQfsize(const PGresult *res, int column_number);

//返回与给定字段编号关联的字段以字节计的大小。 字段编号从0 开始。

45. int PQbinaryTuples(const PGresult *res);

//如果PGresult包含二进制元组数据时返回 1如果包含 ASCII数据返回 0。

46. char *PQgetvalue(const PGresult *res,int row_number, int column_number);//返回一个 PGresult 里面一行的单独的一个字段的值。

47. int PQgetisnull(const PGresult *res,int row_number, int column_number);

//测试一个字段是否为空（NULL）。行和字段编号从 0 开始。

48. i

nt PQgetlength(const PGresult *res,int row_number,int column_number);//返回以字节计的字段的长度。行和字段编号从 0 开始。

49. char *PQcmdStatus(PGresult *res);

// 返回产生PGresult的 SQL 命令的命令状态字符串。

50. char *PQcmdTuples(PGresult *res);// 返回被 SQL 命令影响的行的数量。 51. Oid PQoidValue(const PGresult *res); // 返回一个插入的行的对象标识（OID）—— 如果SQL 命令是 INSERT，或者是一个包含合适 INSERT 语句的准 备好的 EXECUTE的时候。否则，函数返回 InvalidOid。如果受 INSERT 影响的 表不包含 OID， 也返回 InvalidOid。

52. char *PQoidStatus(const PGresult *res);//如果 SQL 命令是INSERT，或者包含合适 INSERT 的准备好语句 EXECUTE 了 。返回一个被插入的行的 OID 的字串。 （如果 INSERT 并非恰好插入一行，或者目标表没有OID，那么字串将是 0。 ） 如果命令不是INSERT，则返回一个空字串。

53. int PQsendQuery(PGconn *conn, const char *command);

//向服务器提交一个命令而不等待结果。 如果查询成功发送则返回 1，否则返回 0。（此时，可以用PQerrorMessage获取关于失败的信息）。

54. int PQsendQueryParams(PGconn *conn,

const char *command,

int nParams,

const Oid *paramTypes,

const char * const *paramValues,

const int *paramLengths,

const int *paramFormats,

int resultFormat);

// 给服务器提交一个命令和（命令需要的）分隔的参数，而不等待结果。

55. int PQsendPrepare(PGconn *conn, const char *stmtName, const char *query, int nParams, const Oid *paramTypes);

//发送一个请求，创建一个给定参数的准备好语句，而不等待结束。

56. int PQsendQueryPrepared(PGconn *conn,

const char *stmtName,

int nParams,

const char * const *paramValues,

const int *paramLengths,

const int *paramFormats,

int resultFormat);

//发送一个执行带有给出参数的准备好的语句的请求，不等待结果。

57. PGresult *PQgetResult(PGconn *conn);

//等待从前面 PQsendQuery，PQsendQueryParams， PQsendPrepare， 或者 PQsendQueryPrepared 调用返回的下一个结果， 然后返回之。当命令结束并且没有更多结果后返回 NULL。

58. int PQconsumeInput(PGconn *conn);

// 如果存在服务器来的输入可用，则使用之。

59. int PQisBusy(PGconn *conn);

// 在查询忙的时候返回 1 ，也就是说，PQgetResult 将阻塞住等待输入。一个 0 的返回表明这时调用 PQgetResult等以确保不阻塞

60. int PQsetnonblocking(PGconn *conn, int arg);// 把连接的状态设置为非阻塞。 如果arg为

1，把连接状态设置为非阻塞， 如果arg为 0， 把连接状态设置为阻塞。

如果 OK 返回 0，如果错误返回 -1。

61. int PQisnonblocking(const PGconn *conn);

// 返回数据库连接的阻塞状态。 如果连接设置为非阻塞状态，返回 1，如果是阻塞状态返回 0。

62. int PQflush(PGconn *conn);

// 试图把任何正在排队的数据冲刷到服务器，如果成功（或者发送队列为空 ）返回 0，如果因某种原因失败返回 -1，或者是在无法把发送队列中的所有数据都发送出去，返回 1。（这种情况只有在连接不为阻塞模式的时候才会出现）。

63. PGcancel *PQgetCancel(PGconn *conn);// 创建一个数据结构，这个数据结构包含通过特定数据库连接取消一个命令所需要的信息。

64. void PQfreeCancel(PGcancel *cancel);

// 释放 PQgetCancel 创建的数据结构。

65. int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);

// 要求服务器放弃处理当前命令。

66. int PQrequestCancel(PGconn *conn); //要求服务器放弃处理当前命令。

（七）MYSQL编程

1.数据类型：

MYSQL

這個結構表示對一個數據庫連接的句柄，它被用于幾乎所有的MySQL函數。

MYSQL_RES

這個結構代表返回行的一個查詢的(SELECT, SHOW, DESCRIBE, EXPLAIN)的結果。從查詢返回的信息在本章下文稱為結果集合。

MYSQL_ROW

這是一個行數據的類型安全(type-safe)的表示。當前它實現為一個計數字節的字符串數組。（如果字段值可能包含二進制數據，你不能將這些視為空終止串，因為這樣的值可以在內部包含空字節) 行通過調用mysql_fetch_row()獲得。

MYSQL_FIELD

這個結構包含字段信息，例如字段名、類型和大小。其成員在下面更詳細地描述。你可以通過重複調用mysql_fetch_field()對每一列獲得MYSQL_FIELD結構。字段值不是這個結構的部分；他們被包含在一個MYSQL_ROW結構中。

MYSQL_FIELD_OFFSET

這是一個相對一個MySQL字段表的偏移量的類型安全的表示。（由mysql_field_seek()使用。）偏移量是在一行以內的字段編號，從0開始。

my_ulonglong

該類型用于行編號和mysql_affected_rows()、mysql_num_rows()和mysql_insert_id()。這種類型

提供0到1.84e19的一個範圍。在一些系統上，試圖打印類型my_ulonglong的值將不工作。為了打印出這樣的值，將它變換到unsigned long並且使用一個%lu打印格式。例如：

printf (Number of rows: %lu\n", (unsigned long) mysql_num_rows(result));

MYSQL_FIELD結構包含列在下面的成員：

char * name

字段名，是一個空結尾的字符串。

char * table

包含該字段的表的名字，如果它不是可計算的字段。對可計算的字段，table值是一個空字符串。

char * def

這字段的缺省值，是一個空結尾的字符串。只要你使用，只有你使用mysql_list_fields()才可設置它。

enum enum_field_types type

字段類型。type值可以是下列之一

：

類型值 類型含義

FIELD_TYPE_TINY TINYINT字段

FIELD_TYPE_SHORT SMALLINT字段

FIELD_TYPE_LONG INTEGER字段

FIELD_TYPE_INT24 MEDIUMINT字段

FIELD_TYPE_LONGLONG BIGINT字段

FIELD_TYPE_DECIMAL DECIMAL或NUMERIC字段

FIELD_TYPE_FLOAT FLOAT字段

FIELD_TYPE_DOUBLE DOUBLE或REAL字段

FIELD_TYPE_TIMESTAMP TIMESTAMP字段

FIELD_TYPE_DATE DATE字段

FIELD_TYPE_TIME TIME字段

FIELD_TYPE_DATETIME DATETIME字段

FIELD_TYPE_YEAR YEAR字段

FIELD_TYPE_STRING 字符串(CHAR或VARCHAR)字段

FIELD_TYPE_BLOB BLOB或TEXT字段(使用max_length决定最大長度）

FIELD_TYPE_SET SET字段

FIELD_TYPE_ENUM ENUM字段

FIELD_TYPE_NULL NULL- 類型字段

FIELD_TYPE_CHAR 不推薦；使用FIELD_TYPE_TINY代替

你可以使用IS_NUM()宏來測試字段是否有一種數字類型。將type值傳給IS_NUM()並且如果字段是數字的，它將計算為TRUE：

if (IS_NUM(field->type))

printf("Field is numeric\n");

unsigned int length

字段寬度，在表定義中指定。

unsigned int max_length

對結果集合的字段的最大寬度(對實際在結果集合中的行的最長字段值的長度)。如果你使用mysql_store_result()或mysql_list_fields()，這包含字段最大長度。如果你使用mysql_use_result()，這個變量的值是零。

unsigned int flags

字段的不同位標志。flags值可以是零個或多個下列位設置：

標志值 標志含義

NOT_NULL_FLAG 字段不能是NULL

PRI_KEY_FLAG 字段是一個主鍵的一部分

UNIQUE_KEY_FLAG 字段是一個唯一鍵的一部分

MULTIPLE_KEY_FLAG 字段是一個非唯一鍵的一部分。

UNSIGNED_FLAG 字段有UNSIGNED屬性

ZEROFILL_FLAG 字段有ZEROFILL屬性

BINARY_FLAG 字段有BINARY屬性

AUTO_INCREMENT_FLAG 字段有AUTO_INCREMENT屬性

ENUM_FLAG 字段是一個ENUM（不推薦）

BLOB_FLAG 字段是一個BLOB或TEXT（不推薦）

TIMESTAMP_FLAG 字段是一個TIMESTAMP（不推薦）

BLOB_FLAG、ENUM_FLAG和TIMESTAMP_FLAG標志的使用是不推薦的，因為他們指出字段的類型而非它的類型屬性。對FIELD_TYPE_BLOB、FIELD_TYPE_ENUM或FIELD_TYPE_TIMESTAMP，最好是測試field->type。下面例子演示了一個典型的flags值用法：

if (field->flags & NOT_NULL_FLAG)

printf("Field can't be null\n");

你可以使用下列方便的宏決來確定flags值的布爾狀態：

IS_NOT_NULL(flags) 真，如果該字段被定義為NOT NULL

IS_PRI_KEY(flags) 真，如果該字段是一個主鍵

IS_BLOB(flags) 真，如果該字段是一個BLOB或TEXT（不推薦；相反測試field->type）

unsigned int decimals

對數字字段的小數位數。

2. my_ulonglong mysql_affected_rows(MYSQL *mysql) 返回被最新的UPDATE, DELETE或INSERT查詢影響的行數。

void mysql_close(MYSQL *mysql) 關閉一個服務器連接。

MYSQL *mysql_connect(MYSQL *mysql, const char *host, const char *user, const char *passwd) 連接一個MySQL服務器。該函數不推薦；使用mysql_real_connect()代替。

my_bool mysql_change_user(MYSQL *mysql, const char *user, const char *password, const char *db) 改變在一個打開的連接上的用戶和數據庫。

int mysql_create_db(MYSQL *mysql, const char *db) 創建一個數據庫。該函數不推薦；而使用SQL命令CREATE DATABASE。

void mysql_data_seek(MYSQL_RES *result, unsigned long long offset) 在一個查詢結果集合中搜尋一任意行。

void mysql_debug(char *debug) 用給定字符串做一個DBUG_PUSH。

int mysql_drop_db(MYSQL *mysql, const char *db) 拋棄一個數據庫。該函數不推薦；而使用SQL命令DROP DATABASE。

int mysql_dump_debug_info(MYSQL *mysql) 讓服務器將調試信息寫入日志文件。

my_bool mysql_eof(MYSQL_RES *result) 確定是否已經讀到一個結果集合的最後一行。這功能被反對; mysql_errno()或mysql_error()可以相反被使用。

unsigned int mysql_errno(MYSQL *mysql) 返回最近被調用的MySQL函數的出錯編號。

char *mysql_error(MYSQL *mysql) 返回最近被調用的MySQL函數的出錯消息。

unsigned int mysql_escape_string(char *to, const char *from, unsigned int length) 用在SQL語句中的字符串的轉義特殊字符。

MYSQL_FIELD *mysql_fetch_field(MYSQL_RES *result) 返回下一個表字段的類型。

MYSQL_FIELD *mysql_fetch_fields(MYSQL_RES *result) 返回一個表字段的類型，給出一個字段編號。

MYSQL_FIELD *mysql_fetch_field_direct(MYSQL_RES *result, unsigned int fieldnr) 返回一個所有字段結構的數組。

unsigned long *mysql_fetch_lengths(MYSQL_RES *result)返回當前行中所有列的長度。

MYSQL_ROW mysql_fetch_row(MYSQL_RES *result) 從結果集合中取得下一行。

unsigned int mysql_field_count(MYSQL *mysql) 把列光標放在一個指定的列上。

MYSQL_FIELD_OFFSET mysql_field_seek(MYSQL_RES *result, MYSQL_FIELD_OFFSET offset) 返回最近查詢的結果列的數量。

MYSQL_FIELD_OFFSET mysql_field_tell(MYSQL_RES *result)返回用于最後一個

mysql_fetch_field()的字段光標的位置。

void mysql_free_result(MYSQL_RES *result) 釋放一個結果集合使用的內存。

char *mysql_get_client_info(void) 返回客戶版本信息。

char *mysql_get_host_info(MYSQL *mysql) 返回一個描述連接的字符串。

unsigned int mysql_get_proto_info(MYSQL *mysql) 返回連接使用的協議版本。

char *mysql_get_server_info(MYSQL *mysql) 返回服務器版本號。

char *mysql_info(MYSQL *mysql) 返回關于最近執行得查詢的信息。

MYSQL *mysql_init(MYSQL *mysql) 獲得或初始化一個MYSQL結構。

my_ulonglong mysql_insert_id

(MYSQL *mysql)返回有前一個查詢為一個AUTO_INCREMENT列生成的ID。

int mysql_kill(MYSQL *mysql, unsigned long pid) 殺死一個給定的線程。

MYSQL_RES *mysql_list_dbs(MYSQL *mysql, const char *wild) 返回匹配一個簡單的正則表達式的數據庫名。

MYSQL_RES *mysql_list_fields(MYSQL *mysql, const char *table, const char *wild) 返回匹配一個簡單的正則表達式的列名。

MYSQL_RES *mysql_list_processes(MYSQL *mysql) 返回當前服務器線程的一張表。

MYSQL_RES *mysql_list_tables(MYSQL *mysql, const char *wild) 返回匹配一個簡單的正則表達式的表名。

unsigned int mysql_num_fields(MYSQL_RES *result) 返回一個結果集合重的列的數量。

my_ulonglong mysql_num_rows(MYSQL_RES *result) 返回一個結果集合中的行的數量。

int mysql_options(MYSQL *mysql, enum mysql_option option, const char *arg) 設置對mysql_connect()的連接選項。

int mysql_ping(MYSQL *mysql) 檢查對服務器的連接是否正在工作，必要時重新連接。

int mysql_query(MYSQL *mysql, const char *query)執行指定為一個空結尾的字符串的SQL查詢。

MYSQL *mysql_real_connect(MYSQL *mysql, const char *host, const char *user, const char *passwd, const char *db, unsigned int port, const char *unix_socket, unsigned int client_flag) 連接一個MySQL服務器。

int mysql_real_query(MYSQL *mysql, const char *query, unsigned int length) 執行指定為帶計數的字符串的SQL查詢。

int mysql_reload(MYSQL *mysql) 告訴服務器重裝授權表。

MYSQL_ROW_OFFSET mysql_row_seek(MYSQL_RES *result, MYSQL_ROW_OFFSET offset) 搜索在結果集合中的行，使用從mysql_row_tell()返回的值。

MYSQL_ROW_OFFSET mysql_row_tell(MYSQL_RES *result) 返回行光標位置。

int mysql_select_db(MYSQL *mysql, const char *db) 連接一個數據庫。

int mysql_shutdown(MYSQL *mysql) 關掉數據庫服務器。

char *mysql_stat(MYSQL *mysql) 返回作為字符串的服務器狀態。

MYSQL_RES *mysql_store_result(MYSQL *mysql) 檢索一個完整的結果集合給客戶。

unsigned long mysql_thread_id(MYSQL *mysql) 返回當前線程的ID。

MYSQL_RES *mysql_use_result(MYSQL *mysql) 初始化一個一行一行地結果集合的檢索。

（八）网络编程

1、socket函数：为了执行网络输入输出，一个进程必须做的第一件事就是调用socket函数获得一个文件描述符。

#include

int socket(int family, int type, int protocol);//成功返回非负描述字；失败返回-1

第一个参数指明了协议簇，目前支持5种协议簇，最常用的有AF_INET(IPv4协议)和AF_INET6(IPv6协议)；第二个参数指明 套接口类型，有三种类型可选：SOCK_STrEAM(字节流套接口)、SOCK_DGrAM(数据报套接口)和SOCK_rAW(原始套接口)；如果

套 接口类型不是原始套接口，那么第三个参数就为0。

2、connect函数：当用socket建立了套接口后，可以调用connect为这个套接字指明远程端的地址；如果是字节流套接口，connect就使用三次握手建立一个连接；如果是数据报套接口，connect仅指明远程端地址，而不向它发送任何数据。

int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);

//成功返回0；失败返回-1

第一个参数是socket函数返回的套接口描述字；第二和第三个参数分别是一个指向套接口地址结构的指针和该结构的大小。

这些地址结构的名字均已“sockaddr_”开头，并以对应每个协议族的唯一后缀结束。以IPv4套接口地址结构为例，它以“sockaddr_in”命名，定义在头文件；以下是结构体的内容：

struct in_addr {

in_addr_t s_addr; //IPV4地址

};

struct sockaddr_in {

uint8_t sin_len;//无符号的8位整数

sa_family_t sin_family;//套接口地址结构地址族，这里为AF_INET

in_port_t sin_port;//TCP或UDP端口

struct in_addr sin_addr;//地址

char sin_zero[8];

};

3、bind函数：为套接口分配一个本地IP和协议端口，对于网际协议，协议地址是32位IPv4地址

或128位IPv6地址与16位的 TCP或UDP端口号的组合；如指定端口为0，调用bind时内核将选择一个临时端口，如果指定一个通配IP地址，则要等到建立连接后内核才选择一个本地 IP地址。

int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);//成功0失败-1

第一个参数是socket函数返回的套接口描述字；第二和第第三个参数分别是一个指向特定于协议的地址结构的指针和该地址结构的长度。

4、listen函数：listen函数仅被TCP服务器调用，它的作用是将用sock创建的主动套接口转换成被动套接口，并等待来自客户端的连接请求。

int listen(int sockfd, int backlog);//成功返回0；失败-1

第一个参数是socket函数返回的套接口描述字；第二个参数规定了内核为此套接口排队的最大连接个数。由于listen函数第二个参数的原 因，内核要维护两个队列：以完成连接队列和未完成连接队列。未完成队列中存放的是TCP连接的三路握手为完成的连接，accept函数是从以连接队列中取 连接返回给进程；当以连接队列为空时，进程将进入睡眠状态。

5、accept函数：accept函数由TCP服务器调用，从已完成连接队列头返回一个已完成连接，如果完成连接队列为空，则进程进入睡眠状态。

int accept(int sockfd,struct sockaddr * cliaddr,socklen_t * addrlen);

// 返回：非负描述字 - - - 成功　　-1 - - - 失败

第一个参数是socket函数返回的套接口描述字；第二个和第三个参数分别是一个指向连接方的套接

口地址结构和该地址结构的长度；该函数返回的是一个全新的套接口描述字；如果对客户段的信息不感兴趣，可以将第二和第三个参数置为空。

6、inet_pton函数：将点分十进制串转换成网络字节序二进制值，此函数对IPv4地址和IPv6地址都能处理。

int inet_pton(int family,const char * strptr,void * addrptr);

//返回：1 - - - 成功　0 - - - 输入不是有效的表达格式　-1 - - - 失败

第一个参数可以是AF_INET或AF_INET6：第二个参数是一个指向点分十进制串的指针：第三个参数是一个指向转换后的网络字节序的二进制值的指针。

7、inet_ntop函数：和inet_pton函数正好相反，inet_ntop函数是将网络字节序二进制值转换成点分十进制串。

const char * inet_ntop(int family,const void *addrptr,char * strptr,size_t len);

//返回：指向结果的指针 - - - 成功　　NULL - - - 失败

第一个参数可以是AF_INET或AF_INET6：第二个参数是一个指向网络字节序的二进制值的指针；第三个参数是一个指向转换后的点分十进制串的指针；第四个参数是目标的大小，以免函数溢出其调用者的缓冲区。

8. int send(int sockfd, const void *msg, int len, int flags);

Sockfd是你想用来传输数据的socket描述符，msg是一个指向要发送数据的指针。

Len是以字节为单位的数据的长度。flags一般情况下置为0（关于该参数的用法可参照man手册）。

char *msg = "Beej was here!"; int len， bytes_sent; … …

len = strlen(msg); bytes_sent = send(sockfd, msg,len,0); … …

send()函数返回实际上发送出的字节数，可能会少于你希望发送的数据。所以需要对send()的返回值进行测量。当send()返回值与len不匹配时，应该对这种情况进行处理。

9.int recv(int sockfd,void *buf,int len,unsigned int flags);

Sockfd是接受数据的socket描述符；buf 是存放接收数据的缓冲区；len是缓冲的长度。Flags也被置为0。Recv()返回实际上接收的字节数，或当出现错误时，返回-1并置相应的errno值。

10. int sendto(int sockfd, const void *msg,int len,unsigned int flags,const struct sockaddr *to, int tolen);

该函数比send()函数多了两个参数，to表示目地机的IP地址和端口号信息，而tolen常常被赋值为sizeof (struct sockaddr)。Sendto 函数也返回实际发送的数据字节长度或在出现发送错误时返回-1。

11. int recvfrom(int sockfd,void *buf,int len,unsigned int flags,struct sockaddr *from,int *fromlen);

from是一个struct sockaddr类型的变量，该变量保存源机的IP地址及端口号。fromlen常置为sizeof(struct sockaddr)。当recvfrom()返回时，fromlen包含实际存入from中的数据字节数。Recvfrom()函数返回接收到的字节数或 当出

现错误时返回-1，并置相应的errno。

应注意的一点是，当你对于数据报socket调用了connect()函数时，你也可以利用send()和recv()进行数据传输，但该socket仍然是数据报socket，并且利用传输层的UDP服务。但在发送或接收数据报时，内核会自动为之加上目地和源地址信息。

12. struct hostent *gethostbyname(const char *name); //实现域名和IP的转换

函数返回一种名为hosten的结构类型，它的定义如下：

struct hostent {

char *h_name; /* 主机的官方域名 */

char **h_aliases; /* 一个以NULL结尾的主机别名数组 */

int h_addrtype; /* 返回的地址类型，在Internet环境下为AF-INET */

int h_length; /*地址的字节长度 */

char **h_addr_list; /* 一个以0结尾的数组，包含该主机的所有地址*/

};

13. int getsockopt(int sockfd,int level,int optname,void *optval,socklen_t *optlen)

int setsockopt(int sockfd,int level,int optname,const void *optval,socklen_t *optlen)

level指定控制套接字的层次.可以取三种值: 1)SOL_SOCKET:通用套接字选项. 2)IPPROTO_IP:IP选项. 3)IPPROTO_TCP:TCP选项.

optname指定控制的方式(选项的名称),我们下面详细解释

optval获得或者是设置套接字选项.根据选项名称的数据类型进行转换

选项名称 说明 数据类型

==========

SOL_SOCKET

SO_BROADCAST 允许发送广播数据 int

SO_DEBUG 允许调试 int

SO_DONTROUTE 不查找路由 int

SO_ERROR 获得套接字错误 int

SO_KEEPALIVE 保持连接 int

SO_LINGER 延迟关闭连接 struct linger

SO_OOBINLINE 带外数据放入正常数据流 int

SO_RCVBUF 接收缓冲区大小 int

SO_SNDBUF 发送缓冲区大小 int

SO_RCVLOWAT 接收缓冲区下限 int

SO_SNDLOWAT 发送缓冲区下限 int

SO_RCVTIMEO 接收超时 struct timeval

SO_SNDTIMEO 发送超时 struct timeval

SO_REUSERADDR 允许重用本地地址和端口 int

SO_TYPE 获得套接字类型 int

SO_BSDCOMPAT 与BSD系统兼容 int

==========

IPPROTO_IP

IP_HDRINCL 在数据包中包含IP首部 int

IP_OPTINOS IP首部选项 int

IP_TOS 服务类型

IP_TTL 生存时间 int

==========

IPPRO_TCP

TCP_MAXSEG TCP最大数据段的大小 int

TCP_NODELAY 不使用Nagle算法 int

==========

关于这些选项的详细情况请查看 Linux Programmer''s Manual

14. TCP客户端的过程

socket()->[*对所要连接的server对

应的sockaddr_in结构中的sin_family,sin_addr.s_addr,sin_port赋值*]->connect()-> { write()->read()-> } close()

TCP服务端的过程

socket()->[*对server所对应的sockaddr_in机构中的sin_family,sin_addr.s_addr,sin_port进行赋值*]->bind()->listen()->{ accept()-> read()->write()-> }close()

15.UDP客户的过程

[*对所要连接的server对应的sockaddr_in结构中的sin_family,sin_addr.s_addr,

sin_port赋值*]->socket()->{ sendto()->recvfrom()-> } close()

UDP服务端的过程

socket()->[对server对应的sockaddr_in结构中的sin_family,sin_addr.s_addr,

sin_port赋值] –>bind()->{ recvfrom()->sendto()-> }close()

（九）文件访问

1. #include

int open(const char *pathname, int oflag, ...)//打开或创建文件

若成功返回文件描述符，出错返回-1；参数说明：

pathname:要打开或创建文件的名字

oflag：以下三个只选其一：

O_RDONLY(只读打开)、O_WRONLY（只写打开）、O_RDWR（读写打开）

以下可选：

O_APPEND 每次写时都追加到文件的尾端

O_CREAT 若此文件不存在，则创建它（需第三个参数来指定权限）

O_EXCL 如果同时指定了CREAT而文件已经存在则出错。

O_TRUNC 如果此文件存在而且为只写或只读，则将其长度截为0

O_NOCTTY 如果pathname指的是终端设备，则不将该设备分配作为此进程的控制终端。

O_NONBLOCK 如果pathname指的是一个FIFO、一个块特殊文件或一个字符特殊文件，则此选项为文件的本次打开操作和后续的I/0操作设置非阻塞模式

O_DSYNC 使每次write等待物理I/0操作完成，但是如果写操作并不影响读取刚写入的数据，则不等待文件属性被更新。

O_RSYNC 使每一个以文件描述符作为参数的read操作等待，直至任何对文件同一部分进行的未决写操作都完成

O_SYNC 使每次write都等到物理I/O操作完成，包括由write操作引起的文件属性更新所需的I/0。

2.creat(const char *pathname, mode_t mode);//创建文件

若成功返回只写打开的文件描述符，失败返回-1

相当于open(pathname, O_WRONLY | O_CREAT | O_TRUNC,mode)

mode的选项：

S_IRUSR // user-read（文件所有者读）

S_IWUSR // user-write（文件所有者写）

S_IXUSR // user-execute（文件所有者执行）

S_IRGRP // group-read

S_IWGRP // group-write

S_IXGRP // group-execute

S_IROTH // other-read

S_IWOTH // other-write

S_IXOTH // other-execute

3.int close(int filedes)//关闭打开的文件；成功返回0，失败返回-1

4.off_t lseek(int filedes, off_t, int whence)

//为一个打开的文件设置其偏移量，成功返回新的文件偏移量 出错返回-1

若whence是SEEK_SET,则将该文件的偏移量设置为距文件开始处offset个字节

若whence是SEEK_CUR,则将该文件的偏移量设置为当前值加offset（可正、负）

若whence是SEEK_END,则

将该文件偏移量设置为文件长度加offset（可正、负）

5.ssize_t read(int filedes, void *buf, size_t nbytes);//从打开的文件中读数据；成功返回读到的字节数，若已到文件结尾则返回0，出错返回-1

6.ssize_t write(int filedes, const void *buf, size_t nbytes)

//向打开的文件写数据；若成功则返回已写的字节数，出错则返回-1

7.ssize_t pread(int filedes, void *buf, size_t nbytes, off_t offset);

//指定偏移量读文件；返回读到的字节数，已到文件结尾返回0，出错返回-1

8.ssize_t pwrite(int filedes, void *buf, size_t nbytes, off_t offset);

//指定偏移量写文件；返回写入的字节数，若出错则返回-1

9.int dup(int filedes);//复制一个现存的文件描述符

int dup2(int filedes, int filedes2)//将filedes复制到filedes2

//若成功则返回新的文件描述符，若出错则返回-1

10.#include

int fcntl(int filedes, int cmd, ...);//改变已打开的文件的性质

若成功则依赖cmd，若出错则返回-1，主要有5种功能：

(1)复制一个现有的描述符（cmd = F_DUPFD）

复制文件描述符filedes，返回新文件描述符

(2)获得/设置文件描述符标记（cmd=F_GETFD或F_SETFD）

返回文件描述符标记（F_GETFD）

(3)获得/设置文件状态标志（cmd=F_GETFL或F_SETFL）

返回文件状态标志（前面提到过，如O_RDONLY）

(4)获得/设置异步I/O所有权（cmd=F_GETOWN或F_SETOWN）

(5)获得/设置记录锁（cmd=F_GETLK、F_SETLK或F_SETLKW）

11.#include

int ioctl(int filedes, int request,...);//通用I/O操作

失败返回-1，成功返回其他值

12.#include

int stat(const char *restrict pathname, struct stat *restrict buf)

//返回与此命名有关的信息结构

int fstat(int filedes, struct stat *buf);//获取已在filedes上打开的文件的有关信息

int lstat(const char *restrict pathname, struct stat *restrict buf);//返回该符号链接的有关信息
三个函数的返回值：成功0失败-1

struct stat{

mode_t st_mode; //文件的类型和权限

ino_t st_ino;//i节点号

dev_t st_dev;//设备号

dev_t st_rdev;//特殊文件的设备号

nlink_t st_nlink;//链接号

uid_t st_uid;//拥有者的用户ID

gid_t st_gid;//拥有者的组ID

off_t st_size;//文件的大小

time_t st_atime;//上次访问的时间

time_t st_mtime;//上一次修改文件的时间

time_t st_ctime;//上一个改变文件状态的时间

blksize_t st_blksize;//最优I/O块大小

blkcnt_t st_blocks;//分配的硬盘块号码

};

13.int access(const char *pathname, int mode);//测试文件的访问权限

成功返回0，失败返回-1；mode有如下几个选项：

R_OK 测试读权限 W_OK测试写权限 X_OK测试执行权限 F_OK 测试文件是否存在

14.#include mode_t umask(mode

_t cmask); //返回原来的屏蔽字

15.#include int chmod(const char *pathname, mode_t mode);

int fchmod(int filedes, mode_t mode);//更改文件的访问权限，成功返回0，失败返回-1

mode的选项除了以上的9个权限之外，还有：S_ISUID 执行时设置用户ID S_ISGID 执行时设置组ID S_ISVTX 保存正本（粘住位） S_IRWXU 用户读写执行 S_IRWXG 组读写执行

16.int chown(const char *pathname, uid_t owner, gid_t group);

int fchown(int filedes, uid_t owner, gid_t group);

int lchown(const char *pathname ,uid_t owner, gid_t group);

//更改文件的用户ID和组ID；成功返回0，失败返回-1

17.int truncate(const char *pathname, off_t length);

int ftruncate(int filedes, off_t length);//截短文件 成功返回0出错返回-1

18.int link(const char *existingpath, const char *newpath);

//在目录existingpath下创建新目录newpath，成功返回0失败返回-1

19.#includeint unlink(const char *pathname)//删除目录，成功0失败返回-1

20.#include int remove(const char *pathname);//删除目录，成功返回0失败返回-1

21.int rename(const char *oldname, const char *newname);//更改文件名，成功返回0失败-1

22.#include int utime(const char *pathname, const struct utimbuf *times);

//更改文件的访问和修改时间；成功返回0，失败-1

struct utimbuf {

time_t actime; //访问时间

time_t modtime;//修改时间

}

23.#include int mkdir(const char *pathname, mode_t mode);//创建文件（0/-1）

24.int rmdir(const char *pathname);//删除文件 (0/-1)

25.#include int chdir(const char *pathname);

int fchdir(int filedes);//更改当前的工作目录 返回值：（0/-1）

26.char *getcwd(char *buf, size_t size);//获取当前工作目录的绝对路径名 返回值（buf/NULL）

（十）标准I/O

1.#include #include

int fwide(FILE *fp, int mode);//设置流定向（宽定向返回正/字节定向返回负/未定向返回0）

mode参数值为负：试图使指定的流是字节定向的；正：宽定向；0不试图设置流的定向

2.void setbuf(FILE *restrict fp, char *restrict buf);

int setvbuf(FILE *restrict fp, char *restrict buf, int mode, size_t size);// 返回(0/-1)

//更改缓冲类型（流被打开后也应该在对该流未执行任何操作时调用）

mode 参数的选项：_IOFBF (全缓冲) _IOLBF （行缓冲）_IONBF（不带缓冲，将忽略buf和size）

3.int fflush(FILE *fp);//冲洗流，将流中的数据传给内核，然后清空缓冲区。返回（0/EOF）

4.FILE *fopen(const char *restrict pathname, const char *restrict type);//打开指定的文件

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

//在一个指定的流上打开文件，若流已打开则先关闭流，若已定向则清除定向。

FILE *fdopen(int filedes, const char *type);

//获取一个现有的文件描述符，并使一个标准的I/O流与该描述

符相结合。

//成功返回文件指针，出错返回NULL

5.int fclose(FILE *fp);//关闭一个打开的流，返回（0/EOF）

6.int getc(FILE *fp);

int fgetc(FILE *fp);

int getchar(void); //读取一个字符；成功返回下一个字符，若已到文件结尾或出错返回EOF

7.int ferror(FILE *fp);

int feof(FILE *fp);//测试文件是否出错和到达结尾；为真返回非0，否则返回0

void clearerr(FILE *fp);//清除错误和结束标志

8.int ungetc(int c, FILE *fp);//将字符压回流中；成功返回c，出错返回EOF

9.int putc(int c, FILE *fp);

int fputc(int c, FILE *fp);

int putchar(int c);//输出函数，成功返回c，出错返回EOF

10.char *fgets(char *restrict buf, int n, FILE *restrict fp);

char *gets(char *buf);//每次输入一行；成功返回buf，出错或文件结尾则返回NULL

11.int fputs(const char *restrict str, FILE *restrict fp);

int puts(const char *str);//输出一行的函数；成功返回非负值，出错则返回EOF

12.size_t fread(void *restrict ptr, size_t size,size_t nobj,FILE *restrict fp);

size_t fwrite(const void *restrict ptr, size_t size, size_t nobj, FILE *restrict fp);

//二进制I/O操作，返回读或写的对象数（即nobj）

13.long ftell(FILE *fp);//返回当前的文件位置指示；成功返回当前文件位置指示，失败-1L

int fseek(FILE *fp, long offset, int whence);//为打开的文件设置新的偏移量（0/非0）

void rewind(FILE *fp);//将一个流设置到文件的起始位置

14.off_t ftello(FILE *fp);//成功返回当前文件的位置指示（与上就差返回类型），失败返回-1

int fseeko(FILE *fp, off_t offset, int whence);//成功返回0，失败返回非0

15.int fgetpos(FILE *restrict fp, fpos_t *restrict pos);

//将文件位置指示器的当前值存入指向的对象中

int fsetpos(FILE *fp, const fpos_t *pos);//用pos的值将流重新定位到该位置；成功0失败非0；

16.int printf(const char *restrict format, . . .)；

int fprintf(FILE *restrict fp, const char *restrict format,...);//成功返回输出字符数 / 负值

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

int snprintf(char *restrict buf, size_t n, const char *restrict format,...);

//成功返回存入数组的字符数，出错则返回负值

17.scanf(const char *restrict format, ...);

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

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

//返回指定的输入项数；若输入出错或在任意变换前已到达文件结尾则返回 E O F

18.int fileno(FILE *fp);//获取文件描述符；返回该流相关联的文件描述符

19.char *tmpnam(char *ptr);//创建临时文件 //返回指向唯一路径的指针

FILE *tmpfile(void); //若成功则返回文件指针,若出错则返回NULL

20.char *tempnam(const char *directory, const char *prefix);//指定目录和前缀创建临时文件

21.int mkstemp(char *template)//指定临时文件的名字创建；成功返回文件描述符，出错返回-1

（十一）系统数据文件和信息

1.include struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name);//获取口令文件返回passwd结构，出错返NULL

struct passwd {

char *pw_name; //用户名

char *pw_passwd; //加密口令

uid_t blog_uid; //数值用户ID

gid_t blog_gid; //数值组ID

char *pw_gecos; //注释字段

char *pw_dir; //出始工作目录

char *pw_shell;//初始shell（用户程序）

};

2.struct passwd *getpwent(void);//返回口令文件（/etc/passwd）中的下一项记录，调用用一定要使用endpwent()来关闭； 出错或者到达文件结尾则返回NULL

void setpwent(void);//打开文件（未打开时）然后反绕它所使用的文件（定位到开始处）

void endpwent(void);//关闭使用的文件

3.#include /etc/shadow

struct spwd *getspnam(const char *name);//获取阴影口令文件返回spwd结构

struct spwd *getspent(void);//获取阴影口令文件中的下一项并返回spwd结构；出错返回NULL

void setspent(void); //打开文件（未打开时）然后反绕文件

void endspent(void);//关闭文件

struct spwd {

char *sp_namp; //用户登录名

char *sp_pwdp;//加密口令

int sp_lstchg; //上次更改口令以来经过的时间

int sp_max; //经过多少天后允许被修改

int sp_warn; //到期警告天数

int sp_inact; //帐户不活动之前剩余天数

int sp_expire; //帐户到期天数

unsigned int sp_flag;//保留字

}

4.#include struct group *getgrgid(gid_t gid);//根据组ID返回相应的group结构

struct gourp *getgrnam(const char *name);根据组名获取组结构

//成功返回相对应的组结构，出错返回NULL

struct group {

char *gr_name;//组名

char *gr_passwd;//加密口令

int gr_gid;//组ID

char **gr_mem;//指向各用户名的指针的数组

};

5.struct group *getgrent(void);//返回/etc/group文件中的下一项并返回指针返回NULL

void setgrent(void);// 打开文件（未打开时）然后反绕文件

void endgrent(void);//关闭文件

6.int uname(struct utsname *name);

//返回与当前主机和操作系统有关的信息，成功返回非负，出错返回-1

struct utsname {

char sysname[]; //操作系统名

char nodename[]; //节点名

char release[]; //当前操作系统的版本

char version[]; //这个版本的序号

char machine[]; //主机名

}

7.int gethostname(char *name, int namlen);//获取主机名给name，成功返回0失败返回-1

8.#include time_t time(time_t *calptr);//返回当前的时间和日期并存于calptr，出错-1

9.#includeint gettimeofday(struct timeval *restrict tp, void *restrict tzp);

//获取精确到微秒的时间并存于tp中，返回值：总是0， tzp的唯一合法值是NULL

struct tmeval {

time_t tv_sec; //秒

long tv_usec ;//

微秒

}

10. struct tm *gmtime(const time_t *calptr);//将日历时间转成国际标准时间的年月日时分秒周日

struct tm *localtime(const time_t *calptr);// 将日历时间转换成本地时间（考虑到本地时间和夏时指标志）返回指向tm结构的指针

struct tm {

int tm_sec; //秒数 [0-60] 60可以表示润秒

int tm_min; //分钟 [0-59]

int tm_hour; //小时 [0-23]

int tm_mday; //日 [1-31]

int tm_mon; //月 [0-11]

int tm_year;//年 [1900-]

int tm_wday; //星期 [0-6]

int tm_yady;//一年的第几天 [0-365]

int tm_sidst; //是否为夏时制

};

11.time_t mktime(struct tm *tmptr);//以本地时间的信息转化成time_t的值，返回：日历时间/-1

12.char *asctime(const struct tm *tmptr);

char *ctime(const time_t *calptr);

//将各自的时间参数转化成如”Tue Feb 10 18:20:15 2007\n\0”的格式

13.size_t strftime(char *restrict buf, size_t maxsize, const char *restrict format, const struct tm *restrict tmptr);//将时间tmptr以指定的format格式存于buf中，返回存入的字符数/失败0

format中可以用以下符号表示要输出的格式：

%a 缩写的周日名 Tue %A 全周日名 Tuesday

%b 缩写的月名 Feb %B 全月名 February

%c 日期和时间 Tue Feb 10 13:02:15 2007 %C 年/100 [0-99] 7

%d 月日[0-31] 10 %D 日期[MM/DD/YY] 07/02/10

%e 月日[1-31]（一位数字加空格） 2 %F [YYYY-MM-DD] 2007-02-10

%g 基于周的年的最后两位数 07 %G 基于周的年 2007

%h 与%b相同 %H 小时（24小时制）：[00-23]

%I 小时（12小时制）[01-12] %j 年日[001-366]

%m 月[01-12] %M 分[00-59]

%n 换行符 %p AM/PM PM

%r 本地时间（12小时） 06：27：38 PM %R 与%H：%M相同 18：27

%S 秒[00-60] %t 水平制表符

%T 与“%H：%M：%S” 18：27：38 %u 周日[1-7]

%U 星期日周数[00-53] %V 周数[01-53]

%w 周日[0-6] %W 星期一周数

%x 日期 %X 时间

%y 年的最后两位数 07 %Y 年 2007

%z UTC偏移量 -0500 %Z 时区名 EST

%% 转换一个% %

（十二）信号

1.UNIX信号表 []内为默认处理动作 core表示动作的同时产生core jump，并生成一个core文件：

SIGABRT 由调用abort函数产生，进程非正常退出 [终止+core]

SIGALRM 用alarm函数设置的timer超时或setitimer函数设置的interval timer超时[终止]

SIGBUS 某种特定的硬件异常，通常由内存访问引起 [终止+core]

SIGCANCEL 由Solaris Thread Library内部使用，通常不会使用 [忽略]

SIGCHLD 进程Terminate或Stop的时候，SIGCHLD会发送给它的父进程。缺省情况下该Signal会被忽略 [忽略]

SIGCONT 当被stop的进程恢复运行的时候，自动发送 [继续/忽略]

SIGEMT 和实现相关的硬件异常 [终止+core]

SIGFPE 数学相关的异常，如被0除，浮点溢出，等等 [终止+core]

SIGFREEZE Solaris专用，Hiberate或者Suspended时候发送 [忽略]

SIGHUP 发送给具有Terminal的Controlling Process，当terminal被disconnect时候发送-----连接断开 [终止]

SIGILL 非法指令异常 [终止+core]

SIGINFO BSD signal。由Status Key产生，通常是CTRL+T。发送给所有Foreground Group的进程 [忽略]

SIGINT 由Interrupt Key产生，通常是CTRL+C或者DELETE。发送给所有ForeGround Group的进程 [终止]

SIGIO 异步IO事件 [忽略/终止]

SIGIOT 实现相关的硬件异常，一般对应SIGABRT [终止+core]

SIGKILL 无法处理和忽略。中止某个进程 [终止]

SIGLWP 由Solaris Thread Libray内部使用 [忽略]

SIGPIPE 在reader中止之后写Pipe的时候发送 [终止]

SIGPOLL 当某个事件发送给Pollable Device的时候发送 [终止]

SIGPROF Setitimer指定的Profiling Interval Timer所产生 [终止]

SIGPWR 和系统相关。和UPS相关。电源失效/重起动 [终止/忽略]

SIGQUIT 输入Quit Key的时候（CTRL+\）发送给所有Foreground Group的进程[终+core]

SIGSEGV 非法内存访问 [终止+core]

SIGSTKFLT Linux专用，数学协处理器的栈异常 [终止]

SIGSTOP 中止进程。无法处理和忽略。 [暂停进程]

SIGSYS 非法系统调用 [终止+core]

SIGTERM 请求中止进程，kill命令缺省发送 [终止]

SIGTHAW Solaris专用，从Suspend恢复时候发送 [忽略]

SIGTRAP 实现相关的硬件异常。一般是调试异常 [终止+core]

SIGTSTP Suspend Key，一般是Ctrl+Z。发送给所有Foreground Group的进程[暂停进程]

SIGTTIN 当Background Group的进程尝试读取Terminal的时候发送 [暂停进程]

SIGTTOU 当Background Group的进程尝试写Terminal的时候发送 [暂停进程]

SIGURG 当out-of-band data接收的时候可能发送 [忽略]

SIGUSR1 用户自定义signal 1 [终止]

SIGUSR2 用户自定义signal 2 [终止]

SIGVTALRM setitimer函数设置的Virtual Interval Timer超时的时候 [终止

SIGWAITING Solaris Thread Librar

y内部实现专用 [忽略

SIGWINCH 当Terminal的窗口大小改变的时候，发送给Foreground Group的所有进程[忽略]

SIGXCPU 当CPU时间限制超时的时候 [终止+core/忽略]

SIGXFSZ 进程超过文件大小限制 [终止+core/忽略]

SIGXRES Solaris专用，进程超过资源限制的时候发送 [忽略]

不产生core文件的几个条件（1）进程是设置用户ID的，而且当前用户并非程序文件的所有者（2）进程是设置组ID的，而且当前用户并非该程序文件的组所有者（3）用户没有写当前工作目录的权限（4）文件已存在，而且用户对该文件没有写权限（5）文件太大。

2.#include void (*signal(int signo, void(*func)(int)))(int);

//捕捉信号并处理，返回之前的signal处理函数，错误返回SIG_ERR

signo：信号名

Func：函数地址，或者是SIG_IGN （忽略Signal）或SIG_DFL（缺省行为）。原型为：void (*)(int)。不过很多UNIX的实现也会传入一些和实现相关的参数. 如果用exec创建子进程，那么子进程的所有Signal状态是缺省或者忽略：1. 如果父进程忽略了某个Signal，那么父进程用exec创建子进程时子进程的这个Signal的状态也是忽略2. 如果在父进程的某个Signal注册了Signal处理函数的话，那么这个子进程的该Signal的状态会恢复成缺省状态，因为注册的函数地址肯定在另外一个进程中无法调用。如果用fork创建子进程，那么子进程会继承所有父进程的Signal状态。

3. int kill(pit_t pid, int signo);//发送信号给进程或进程组

pid的值>0 将该信号发送给进程ID为pid的进程

pid==0 将该信号发送给与发送进程属于同一进程组的所有进程，且发送进程具有向其发信号的权限

pid < 0 IDpidp>

pid == -1将该信号发送给发送进程有权限向他们发送信号的系统上的所有进程

int raise(int signo);//进程给自身发送信号；成功返回0，错误返回-1

4. unsigned int alarm(unsigned int second);//返回前一个alarm的剩余时间值，单位秒

SIGALRM的缺省行为是中止进程。同一个进程只能有一个alarm。调用alarm函数会取消前一个alarm，并返回前一个alarm的剩余时间。如果second值为0，则只是取消前一个alarm。

int pause(void);//Pause函数暂停当前进程运行，知道某个信号被捕捉signal被catch

5. int sigemptyset(sigset_t *set); 初始化sigset_t，所有signal都不包括在内

int sigfillset(sigset_t *set); 初始化sigset_t，包括所有signal

int sigaddset(sigset_t *set, int signo); 添加signal

int sigdelset(sigset_t *set, int signo); 删除signal

//成功返回0，错误返回-1

int sigismember(sigset_t

*set, int signo);

//返回1如果是，不是则返回0

6. int sigprocmask(int how, const sigset_t *restrict set, sig_set *restrict oset);

//成功返回0，错误返回-1； how参数的意义如下：

SIG_BLOCK 在原来的mask基础上Block signal set

SIG_UNBLOCK 在原来的mask基础上Unblock signal set

SIG_SETMASK 设置mask为指定的signal set

set：signal set，如果=NULL则忽略how参数（again，又是一个不太合适的设计，不利于差错和记忆）

oset：返回当前process的signal mask

7. int sigpending(sigset_t *set);//成功返回0，错误返回-1 函数返回当前被block而没有deliver的

signal的集合到set中

8.int sigaction(int signo, const struct sigaction *restrict act, struct sigaction *restrict oact);// 检查或修改与指定信号相关联的处理动作（或同时执行两种操作）

struct sigaction {

void (*sa_handler)(int);// signal处理函数

sigset_t sa_mask;

int sa_flags;

void (*sa_sigaction)(int, siginfo_t *, void *)

};// sa_mask：当signal函数被调用的时候，自动block这些signal。当signal处理函数返回的时候，mask会恢复到初始值。同时，操作系统也会自动把该signal加到mask中，防止重入；
sa_flags：定义如下：

SA_INTERRUPT 被该signal中断的系统调用不会自动重启

SA_NOCLDSTOP 如果signo是SIGCHLD，当子进程stop的时候不产生该signal

SA_NOCLDWAIT 如果signo是SIGCHLD，子进程不会变成zombie。当调用wait的时候，调用进程会一直等待直到所有子进程结束

SA_NODEFER 在signal处理函数中系统不会自动block该signal

SA_ONSTACK 如果调用了sigaltstack指定一个额外的stack，那么该signal发送的时候，进程处于此stack中

SA_RESETHAND 恢复为初始化值，SA_SIGINFO flag也会被清除

SA_RESTART 被该signal中断的系统调用会自动重启

SA_SIGINFO 为signal处理函数提供额外信息

9. int sigsetjmp(sigjmp_buf env, int savemask); //保存信号屏蔽字

//直接调用返回0，从siglongjmp返回的时候返回非0

void siglongjmp(sigjmp_buf env, int val);//恢复信号屏蔽字

//如果sigsetjmp的savemask为非0，则sigsetjmp会在env参数中记住当前process的signal mask。一般情况下如果和signal打交道的话都需要调用这两个函数。

10.int sigsuspend(const sigset_t *sigmask);//将信号屏蔽字设置为sigmask指定的值。在捕捉到一个信号或发生了一个会终止该进程的信号前，该进程被挂起。如果捕捉到一个信号而且从该信号处理程序返回，则sigsuspend返回并将进程的信号屏蔽字设置为调用sigsuspend前的值 ； 返回-1并将errno设置为EINTR

11.void abort(void);//使异常程序终止

12. System函数除了会执行可执行文件创建子进程之外，还会设置sigmask为忽 略
SIGINT，SIGQUIT并

Block SIGCHLD。1. 忽略SIGINT和SIGQUIT的原因是，只有子进程应该处理这两个signal，而父进程无需处理2. Block SIGCHLD的原因是，System函数需要知道子进程的结束，而父进程不应该先提前知道，以免提前调用wait函数使得System函数本身无法获得进程的退出值

13.unsigned int sleep(unsignes int seconds);//休眠函数，返回0或未休眠的秒数

14.void gsignal(int signo, const char *msg);//字符串msg（通常是程序名）输出到标准错错文件，后接一个冒号个一个空格再接着对该信号的说明，最后是一个换行符，类似于perror

15.char *strsignal(int signo);//给出一个信号，strsignal将返回说明该信号的字符串，应用程序可用该字符串打印关于收到信号的出错信息。类似于strerror