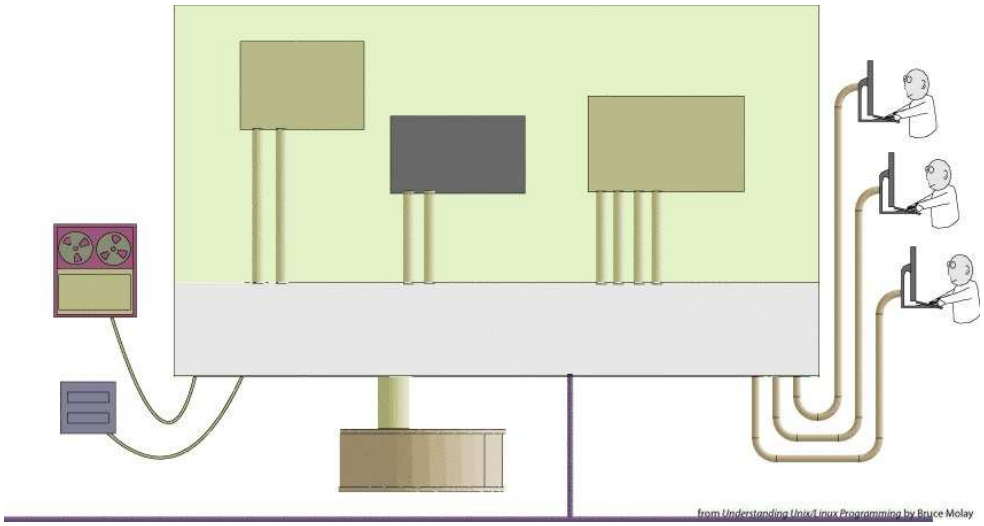
Objectives

Summary: In this chapter, we shall write a version of the who utility. In the process, we shall learn about

- on-line documentation
- the Unix file interface: open, read, write, lseek, close
- files decriptors
- kernel mode, user mode
- use utmp file to find list of current users
- detecting and reporting errors in system calls

In Unix system, there may be multiple users logged on at the same time. To know who else is using the computer, we use command *who*.



In Unix, each command is actually a program, usually written in C. Commands are usually put in directories such as */bin/ /sbin/ /usr/local/bin*.

How does who work

- What does it do?
- How does it do it?
- How can I learn about the details?

How does who work

Output running *who* on nova.kettering.edu

cwu	pts/2	Jan 28 17:57	(adsl-69-209-140-226.dsl.sfldmi.ameritech.net)
ellis	pts/4	Jan 17 11:47	(ellis-xp.kettering.edu)
ehynes	pts/5	Jan 28 17:46	(portmas031.kettering.edu)
thajek	pts/6	Jan 23 08:55	(morpheus.kettering.edu)
kpalmer	pts/7	Jan 5 12:49	(kip.kettering.edu)
jhuggins	pts/8	Jan 28 13:21	(24-236-238-57.dhcp.bycy.mi.charter.com)
ellis	pts/9	Jan 24 10:12	(adsl-67-38-2-40.dsl.sfldmi.ameritech.net)
ellis	pts/12	Jan 12 17:32	(ellis-xp.kettering.edu)
vand0215	pts/13	Jan 28 16:23	(70-134-57-81.ded.swbell.net)
jsalacus	pts/14	Jan 28 15:19	(jsalacus-xp.kettering.edu)
kpalmer	pts/16	Jan 5 12:45	(kip.kettering.edu)
ellis	pts/1	Jan 26 09:36	(adsl-67-38-2-40.dsl.sfldmi.ameritech.net)

We see that *who* shows the following information of each user:
logname, terminal, time, from where

How does who work

To learn more about Unix commands, try

- read the manual
  man who
- search the manual
  man -k utmp —more
- read the .h files in /usr/include
  more /usr/include/utmp.h
- follow the "See Also" links

How does who work–*man who*

```
NAME
       who - show who is logged on

SYNOPSIS
       who [OPTION]... [ FILE | ARG1 ARG2 ]
DESCRIPTION
...
 If  FILE is not specified, use /var/run/utmp.
...
```

How does who work–utemp structure

```
struct utmp {
    short ut_type;              /* type of login */
    pid_t ut_pid;               /* pid of login process */
    char ut_line[UT_LINESIZE];  /* device name of tty - "/dev/" */
    char ut_id[4];              /* init id or abbrev. ttyname */
    char ut_user[UT_NAMESIZE];  /* user name */
    char ut_host[UT_HOSTSIZE];  /* hostname for remote login */
    struct exit_status {
    short int e_termination;    /* process termination status.  */
    short int e_exit;           /* process exit status.  */
    }; ut_exit;                 /* The exit status of a process
                                   marked as DEAD_PROCESS. */
    time_t ut_time;       /* time entry was made.  */
};
```

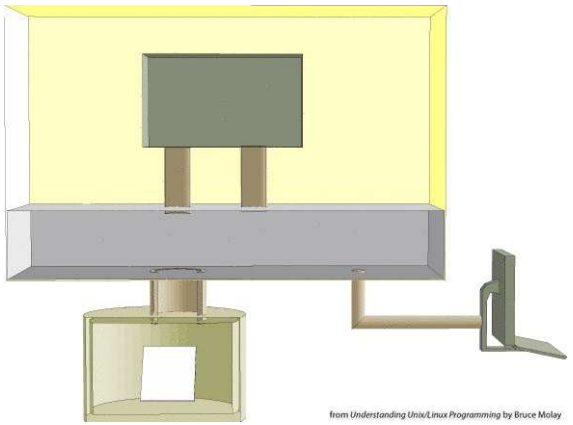
UT_LINESIZE, UT_NAMESIZE, UT_HOSTSIZE are three constants. In Linux OS, they are 32, 32 and 256. Due to the various implementation Unix, the definition of utmp might be different, however, you shall not have problem in accessing the utmp structure using the above definition.

How does who work

Answer

who works by:

- Open utmp
- read record
- display info
- closefile



Can I write *who*

- Read structs from a file
- Display the information stored in a struct

Open a file: open

- **Include** #include ⟨fcntl.h⟩
- **purpose** Create a connection to a file
- **Usage** int fd = open (char *name, int how)
- **Args** name: name of a file, how: O_RDONLY, O_WRONGLY, or O_RDWR.
- **Returns** -1: on error, int: on success

Read data from a file: read

- **Include** #include ⟨unistd.h⟩
- **purpose** Transfer up to qty bytes from fd to buf
- **Usage** ssize_t numread = read (int fd, void *buf, size_t qty)
- **Args** fd: source of data, buf: distination for data, qty: number of bytes to transfer.
- **Returns** -1: on error, numread: on success

close a file: close

- **Include** #include ⟨unistd.h⟩
- **purpose** Close a file
- **Usage** int result = close (int fd)
- **Args** fd: file descriptor
- **Returns** -1: on error, 0: on success

First version of who program

```
/* who1.c  - a first version of the who program
 *               open, read UTMP file, and show results
 */
#include <stdio.h>
#include <utmp.h>
#include <fcntl.h>
#include <unistd.h>

#define SHOWHOST /* include remote machine on output */

int main()
{
        struct utmp     current_record; /* read info into here       */
        int             utmpfd;         /* read from this descriptor */
        int             reclen = sizeof(current_record);

        if ( (utmpfd = open(UTMP_FILE, O_RDONLY)) == -1 ){
                perror( UTMP_FILE );    /* UTMP_FILE is in utmp.h    */
                exit(1);
        }

        while ( read(utmpfd, &current_record, reclen) == reclen )
                show_info(&current_record);
```

```
                close(utmpfd);
                return 0;               /* went ok */
}
/*
 *  show info()
 * displays contents of the utmp struct in human readable form
 * *note* these sizes should not be hardwired
 */
show_info( struct utmp *utbufp )
{
        printf("%-8.8s", utbufp->ut_name);      /* the logname */
        printf(" ");                            /* a space */
        printf("%-8.8s", utbufp->ut_line);      /* the tty */
        printf(" ");                            /* a space */
        printf("%10ld", utbufp->ut_time);       /* login time */
        printf(" ");                            /* a space */
#ifdef SHOWHOST
        printf("(%s)", utbufp->ut_host);        /* the host */
#endif
        printf("\n");                           /* newline */
}
```

```
$ cc who1.c -o who1
                        1138461189 ()
  reboot                1138461189 ()
  runlevel              1138461189 ()
                        1138461217 ()
  LOGIN     tty1        1138461217 ()
  LOGIN     tty2        1138461219 ()
  LOGIN     tty3        1138461217 ()
  LOGIN     tty4        1138461217 ()
  LOGIN     tty5        1138461217 ()
  LOGIN     tty6        1138461218 ()
                        1138461217 ()
  wch       :0          1138461239 ()
  wch       pts/1       1138461980 (:0.0)
  wch       pts/2       1138479409 ()
  wch       pts/3       1138497458 (:0.0)
```

Improvement needed

- Suppress blank record
- Get the log-in times correct

Suppress blank record

check ut_type(see utmp.h)

/usr/include/utmp.h contains the following definition

```
/* Values for the 'ut_type' field of a 'struct utmp'. */
          #define UT_UNKNOWN            0
          #define RUN_LVL               1
          #define BOOT_TIME             2
          #define NEW_TIME              3
          #define OLD_TIME              4
          #define INIT_PROCESS          5
          #define LOGIN_PROCESS         6
          #define USER_PROCESS          7
          #define DEAD_PROCESS          8
          #define ACCOUNTING            9
```

Suppress blank record

A modified version of *show_info*

```
/*
 *  show info()
 * displays contents of the utmp struct in human readable form
 * *note* these sizes should not be hardwired
 */
show_info( struct utmp *utbufp )
{
        if( utbufp->ut_type != USER_PROCESS)
             return;
        printf("%-8.8s", utbufp->ut_name);      /* the logname */
        printf(" ");                            /* a space */
        printf("%-8.8s", utbufp->ut_line);      /* the tty */
        printf(" ");                            /* a space */
        printf("%10ld", utbufp->ut_time);       /* login time */
        printf(" ");                            /* a space */
#ifdef SHOWHOST
        printf("(%s)", utbufp->ut_host);        /* the host */
#endif
        printf("\n");                           /* newline */
}
```

Formate time

Unix stores time as the number of seconds since the midnight, Jan 1, 1970, GMT. The numbers we see on the output of our 'who' is the number of second elapsed. There is a system call 'ctime' to convert Unix representation of time to a human readable form.

```
man 3 ctime
NAME
       asctime,   ctime,   gmtime,   localtime,  mktime,  asctime_r,  ctime_r,
       gmtime_r, localtime_r - transform date and time to broken-down time  or
       ASCII

SYNOPSIS
       #include <time.h>

       char *asctime(const struct tm *tm);
       char *asctime_r(const struct tm *tm, char *buf);

       char *ctime(const time_t *timep);
       char *ctime_r(const time_t *timep, char *buf);

       struct tm *gmtime(const time_t *timep);
```

```
       struct tm *gmtime_r(const time_t *timep, struct tm *result);

       struct tm *localtime(const time_t *timep);
       struct tm *localtime_r(const time_t *timep, struct tm *result);

       time_t mktime(struct tm *tm);
......
DESCRIPTION
       The ctime(), gmtime() and localtime() functions all take an argument of
       data type time_t which represents calendar time.  When  interpreted  as
       an  absolute  time  value,  it represents the number of seconds elapsed
       since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC).
......
       The  call ctime(t) is equivalent to asctime(localtime(t)).  It converts
       the calendar time t into a string of the form

              "Wed Jun 30 21:49:08 1993\n"
```

Second version who2.c

```
/* who2.c  - read /etc/utmp and list info therein
 *         - suppresses empty records
 *         - formats time nicely
 */
#include        <stdio.h>
#include        <unistd.h>
#include        <utmp.h>
#include        <fcntl.h>
#include        <time.h>

/* #define      SHOWHOST */

void showtime(long);
void show_info(struct utmp *);

int main()
{
        struct utmp     utbuf;          /* read info into here */
        int             utmpfd;         /* read from this descriptor */

        if ( (utmpfd = open(UTMP_FILE, O_RDONLY)) == -1 ){
                perror(UTMP_FILE);
                exit(1);
```

```
        }

        while( read(utmpfd, &utbuf, sizeof(utbuf)) == sizeof(utbuf) )
                show_info( &utbuf );
        close(utmpfd);
        return 0;
}
/*
 *      show info()
 *                      displays the contents of the utmp struct
 *                      in human readable form
 *                      * displays nothing if record has no user name
 */
void show_info( struct utmp *utbufp )
{
        if ( utbufp->ut_type != USER_PROCESS )
                return;

        printf("%-8.8s", utbufp->ut_name);      /* the logname  */
        printf(" ");                            /* a space      */
        printf("%-8.8s", utbufp->ut_line);      /* the tty      */
        printf(" ");                            /* a space      */
        showtime( utbufp->ut_time );            /* display time */
#ifdef SHOWHOST
```

```
        if ( utbufp->ut_host[0] != '\0' )
                printf(" (%s)", utbufp->ut_host);/* the host    */
#endif
        printf("\n");                           /* newline      */
}

void showtime( long timeval )
/*
 *      displays time in a format fit for human consumption
 *      uses ctime to build a string then picks parts out of it
 *      Note: %12.12s prints a string 12 chars wide and LIMITS
 *      it to 12chars.
 */
{
        char    *cp;                    /* to hold address of time      */

        cp = ctime(&timeval);           /* convert time to string       */
                                        /* string looks like            */
                                        /* Mon Feb  4 00:46:40 EST 1991 */
                                        /* 0123456789012345.            */
        printf("%12.12s", cp+4 );       /* pick 12 chars from pos 4     */
}
```

Output of who2.c

```
[wch@localhost]$ ./who2
wch      :0       Feb 19 10:49
wch      pts/1    Feb 19 11:30
wch      pts/2    Feb 19 15:17

[wch@localhost]$ who
wch      :0            Feb 19 10:49
wch      pts/1         Feb 19 11:30 (:0.0)
wch      pts/2         Feb 19 15:17
```

Project 2: cp (read and write)

In 'who', we read from a file. How do we write to a file? Let's explore a real example

cp source-file target-file

(1) what does cp do?

ANS: creates or truncates target-file, then writes the content of sourcefile to it.

(2) How does cp creat and write?

ANS: search the manual for the answers

Creating a file–creat

- **Include** #include ⟨fcntl.h ⟩
- **purpose** Create or zero a file
- **Usage** int fd = create (char *filename, mode_t mode)
- **Args** filename: the name of the file, mode: access permission
- **Returns** -1: on error, fd: on success

Creating a file–write

- **Include** #include ⟨unistd.h ⟩
- **purpose** Write data from memory to a file
- **Usage** ssize_t result =write(int fd, void *buf, size_t amt)
- **Args** fd: a file descriptor, buf: an array, amt: how many bytes to write
- **Returns** -1: on error, num written: on success

The logic of writting Ucopy.c

- step 1: open sourcefile
- step 2: create copyfile
- step 3: read source to buffer
- step 4: write buffer to copy
- step 5: repeat step3 and step4 until meets end of file.
- step 6: close

The logic of writting Ucopy.c

- step 1: open sourcefile
- step 2: create copyfile
- step 3: read source to buffer
- step 4: write buffer to copy
- step 5: repeat step3 and step4 until meets end of file.
- step 6: close

cp1.c

```
/** cp1.c
 *      version 1 of cp - uses read and write with tunable buffer size
 *
 *      usage: cp1 src dest
 */
#include        <stdio.h>
#include        <unistd.h>
#include        <fcntl.h>

#define BUFFERSIZE      4096
#define COPYMODE        0644

void oops(char *, char *);

main(int ac, char *av[])
{
        int     in_fd, out_fd, n_chars;
        char    buf[BUFFERSIZE];
                                                /* check args  */
        if ( ac != 3 ){
                fprintf( stderr, "usage: %s source destination\n", *av);
                exit(1);
        }
```

```
                                                /* open files */
        if ( (in_fd=open(av[1], O_RDONLY)) == -1 )
                oops("Cannot open ", av[1]);
        if ( (out_fd=creat( av[2], COPYMODE)) == -1 )
                oops( "Cannot creat", av[2]);
                                                /* copy files */
        while ( (n_chars = read(in_fd , buf, BUFFERSIZE)) > 0 )
                if ( write( out_fd, buf, n_chars ) != n_chars )
                        oops("Write error to ", av[2]);
        if ( n_chars == -1 )
                        oops("Read error from ", av[1]);
                                                /* close files */
        if ( close(in_fd) == -1 || close(out_fd) == -1 )
                oops("Error closing files","");
}

void oops(char *s1, char *s2)
{
        fprintf(stderr,"Error: %s ", s1);
        perror(s2);
        exit(1);
}
```

Does Buffer size matter

In the previous program, there is a BUFFERSIZE constant. Does the value of BUFFERSIZE matter? Yes, it matters!

Example

Filesize = 2500 bytes
if buffer = 100 bytes
$\Rightarrow$ 25 read() and 25 write() calls

if buffer = 1000 bytes
$\Rightarrow$ 3 read() and 3 write() calls

Important Idea

A system call is resource expensive. It runs various kernel functions, and it also requires a shift from USER MODE to KERNEL MODE and back. This shift takes time. Thus, we should try to minimize the number of system calls.

Does this mean who2.c is inefficient?

yes! Making one system call for each line of output makes as much sense as buying pizza by the slice or eggs one at a time

Better idea: Read in a bunch of records at a time and then, as with eggs in a carton, take them one by one

More efficient version of our 'who'

```
/* utmplib.c  - functions to buffer reads from utmp file
 *
 *      functions are
 *              utmp_open( filename )   - open file
 *                      returns -1 on error
 *              utmp_next( )            - return pointer to next struct
 *                      returns NULL on eof
 *              utmp_close()            - close file
 *
 *      reads NRECS per read and then doles them out from the buffer
 */
#include        <stdio.h>
#include        <fcntl.h>
#include        <sys/types.h>
#include        <utmp.h>

#define NRECS   16
#define NULLUT  ((struct utmp *)NULL)
#define UTSIZE  (sizeof(struct utmp))

static  char    utmpbuf[NRECS * UTSIZE];                /* storage      */
static  int     num_recs;                               /* num stored   */
static  int     cur_rec;                                /* next to go   */
```

```
static  int     fd_utmp = -1;                           /* read from    */

utmp_open( char *filename )
{
        fd_utmp = open( filename, O_RDONLY );           /* open it      */
        cur_rec = num_recs = 0;                         /* no recs yet  */
        return fd_utmp;                                 /* report       */
}

struct utmp *utmp_next()
{
        struct utmp *recp;

        if ( fd_utmp == -1 )                            /* error ?      */
                return NULLUT;
        if ( cur_rec==num_recs && utmp_reload()==0 )    /* any more ?   */
                return NULLUT;
                                        /* get address of next record    */
        recp = ( struct utmp *) &utmpbuf[cur_rec * UTSIZE];
        cur_rec++;
        return recp;
}

int utmp_reload()
```

```
/*
 *      read next bunch of records into buffer
 */
{
        int     amt_read;
                                                /* read them in         */
        amt_read = read( fd_utmp , utmpbuf, NRECS * UTSIZE );
                                                /* how many did we get? */
        num_recs = amt_read/UTSIZE;
                                                /* reset pointer        */
        cur_rec  = 0;
        return num_recs;
}

utmp_close()
{
        if ( fd_utmp != -1 )                    /* don't close if not   */
                close( fd_utmp );               /* open                 */
}
```

More efficient version of our 'who'

```
/* who3.c - who with buffered reads
 *        - surpresses empty records
 *        - formats time nicely
 *        - buffers input (using utmplib)
 */
#include        <stdio.h>
#include        <sys/types.h>
#include        <utmp.h>
#include        <fcntl.h>
#include        <time.h>

#define SHOWHOST

void show_info(struct utmp *);
void showtime(time_t);

int main()
{
        struct utmp *utbufp,            /* holds pointer to next rec */
                    *utmp_next();       /* returns pointer to next */

        if ( utmp_open( UTMP_FILE ) == -1 ){
                perror(UTMP_FILE);
```

```
                exit(1);
        }
        while ( ( utbufp = utmp_next() ) != ((struct utmp *) NULL) )
                show_info( utbufp );
        utmp_close( );
        return 0;
}
/*
 *      show info()
 *                      displays the contents of the utmp struct
 *                      in human readable form
 *                      * displays nothing if record has no user name
 */
void show_info( struct utmp *utbufp )
{
        if ( utbufp->ut_type != USER_PROCESS )
                 return;

        printf("%-8.8s", utbufp->ut_name);              /* the logname */
        printf(" ");                                    /* a space */
        printf("%-8.8s", utbufp->ut_line);              /* the tty */
        printf(" ");                                    /* a space */
        showtime( utbufp->ut_time );                    /* display time */
#ifdef SHOWHOST
```

```
        if ( utbufp->ut_host[0] != '\0' )
                printf(" (%s)", utbufp->ut_host);       /* the host */
#endif
        printf("\n");                                   /* newline */
}

void showtime( time_t timeval )
/*
 *      displays time in a format fit for human consumption
 *      uses ctime to build a string then picks parts out of it
 *      Note: %12.12s prints a string 12 chars wide and LIMITS
 *      it to 12chars.
 */
{
        char    *ctime();               /* convert long to ascii        */
        char    *cp;                    /* to hold address of time      */

        cp = ctime( &timeval );         /* convert time to string       */
                                        /* string looks like            */
                                        /* Mon Feb  4 00:46:40 EST 1991 */
                                        /* 0123456789012345.            */
        printf("%12.12s", cp+4 );       /* pick 12 chars from pos 4     */
}
```

If buffering is so smart, why doesn't kernel do it?

It does!

The kernel keeps copies of disk blocks in memory. It writes those blocks to disk now and then. Then read() call actually copies data from kernel buffers not from the disk.

If the machine is suddently shut off, the kernel may not have enough time to write all block in memory back to disk.

Consequences of Buffering

- Faster "disk" I/O
- Optimize disk reads and writes
- Need to sync disks before shutdown

Logging out: How it works?

When you logged out, the record in utmp is changed. It is done as follows:

- 1. Open the utmp file

```
fd = open(UTMP_FILE, O_RDWR);
```

- 2. Read the utmp file until it finds the record for your terminal

```
while(read(fd, rec, utmplen) == utmplen)      /* get next record */
      if( strcmp(rec.ut_line, myline) == 0)   /* what, my line */
            revise_entry();                   /* remove my name */
```

- 3. Write a revised utmp record in its place

```
lseek() system call
```

- 4. Close the utmp file

```
close(fd)
```

Change the current position in a file: lseek

- **Include**
  #include ⟨ sys/types.h ⟩
  #include ⟨ unistd.h ⟩
- **purpose** Seek file pointer to specified offset in file
- **Usage** off_t oldpos = lseek (int fd, off_t dist, int base)
- **Args**
  - fd: file descriptor
  - dist: a distance in bytes
  - base: SEEK_SET, SEEK_CUR, SEEK_END
- **Returns** -1 on error, or the previous position in the file

Code to log out from a terminal

```
/*
* logout_tty(char *line)
*  marks a utmp record as logged out
*  does not blank username or remote host
*  returns -1 on error, 0 on success
*/
int logout_tty(char *line)
{
      int         fd;
      struct utmp rec;
      int         len = sizeof(struct utmp);
      int         retval = -1;                        /* pessimism */

      if( (fd = open(UTMP_FILE, O_RDWR)) == -1)       /* open file */
          return -1;
      /* search and replace */
      while ( read(fd, &rec, len) == len)
         if( strncmp (rec.ut_line, line, sizeof(rec.utline))==0)
         {
            rec.ut_type = DEAD_PROCESS;               /* set type */
            if( time( &rec.ut_time) != -1 )           /* and time */
               if ( lseek(fd, -len, SEEK_CUR) != -1) /* back up */
                  if ( write(fd, &rec, len) == len ) /* update */
```

```
                    retval = 0;                         /* success! */
                break;
             }
          /* close the file */
          if ( close(fd) == -1 )
             retval = -1;
          return retval;
}
```

Error handling

```
NAME
       errno - number of last error

SYNOPSIS
       #include <errno.h>

       extern int errno;

DESCRIPTION
       The  integer  errno is set by system calls (and some library functions) to
       indicate what went wrong.  Its value is significant  only  when  the  call
       returned  an  error (usually -1), and a library function that does succeed
       is allowed to change errno.

       Sometimes, when -1 is also a legal return value  one  has  to  zero  errno
       before the call in order to detect possible errors.

       errno  is  defined by the ISO C standard to be a modifiable lvalue of type
       int, and must not be explicitly declared; errno may be a macro.  errno  is
       thread-local;  setting  it  in one thread does not affect its value in any
       other thread.

       Valid error numbers are all non-zero; errno is never set to  zero  by  any
```

```
       library function.  All the error names specified by POSIX.1 must have dis-
       tinct values.
```

A list of erro numbers can be obtained by command ‘man 3 errno’

Error handling

```
int sample ()
{
   int fd;
   fd = open("file", O_RDONLY);
   if( fd == -1 )
   {
      perror("Cannot open file");
      return;
   }
   ...
}
```

Common mistake in error handling

A common mistake is to do

```
if (somecall() == -1) {
    printf("somecall() failed\n");
    if (errno == ...) { ... }
}
```

where errno no longer needs to have the value it had upon return from somecall(). If the value of errno should be preserved across a library call, it must be saved:

```
if (somecall() == -1) {
    int errsv = errno;
    printf("somecall() failed\n");
    if (errsv == ...) { ... }
}
```

To make errno more reliable, it is better to set errno to zero before a system call after which you check the value of errno.