```
char* nem2 = new char [ 1048 ];          // C++ style
```

The *calloc* function is similar to *malloc*, except that it guarantees that the allocated memory is initialized with the zero byte.

The *free* function is similar to the *delete* operator, and they both deallocate dynamic memory. For example, the following statement discards dynamic memory referenced by a variable called *mem1*:

```
free ( mem1 );
```

The *realloc* function is used to adjust the size of any dynamic memory allocated by *malloc* or *calloc*. It is very useful in managing a dynamic array that may change size in the course of a process execution. For example, a program that stores any user input from the standard input has no way of knowing ahead of time how many lines of data will be obtained from a user. A dynamic array may be used in this situation to allocate just the exact amount of memory, via the *realloc* function, to store the user input at any time.

Note that in the above example, the process may also use a linked-list or a fixed-size array to store the user's input. The drawbacks of the linked-list approach, as compared to using an array (static or dynamic), are: Because each linked-list record requires storage for a next pointer, it consumes more memory that does an array with the same number of entries. Furthermore, linked-lists setup and traversal are generally more time-consuming that are accessing data from arrays. The drawback of using a fixed-size array as compared to using a dynamic array is that it requires preallocation of all the memory to store the maximum allowed input data; thus, it is not as efficient in memory usage as dynamic arrays. Moreover, this approach imposes a maximum amount of input data allowed and may be too restrictive to users. Using a dynamic array, no such limit is necessary. In summary, dynamic arrays are preferred over linked-lists when the data stored are accessed frequently and when the order of the stored data does not change. Also dynamic arrays are preferred over static arrays if the size of an array may change (increase or decrease) over time and if it is not practical to set an upper limit on the size of an array.

The *realloc* function takes two arguments.The first argument, *old_memp*, contains an address of a dynamic memory region previously allocated. The second argument, *new_size*, is the size, in bytes, of a new dynamic memory region to be returned. The *new_size* value may be larger or smaller than the size of the old memory region as referenced by *old_memp*. The *realloc* function attempts to adjust the old dynamic memory region size to be *new_size*. If this cannot be achieved, a new dynamic memory region of size *new_size* is allocated. The function then copies data from the old memory region to the new memory region, up to the maximum size of the old or the new memory region (whichever is less), and the old memory region is deallocated by the function. If the new memory region size is larger than the old one, the content of the memory in the new one that is not initialized by the old memory data is undefined.

The following *malloc.C* program depicts the use of *realloc* to store a user's input data from the standard input:

```
#include <iostream.h>
#include <string.h>
#include <malloc.h>
int main()
{
        char** inList = 0, buf[256];
        int numIn = 0, max_size = 0;
        /* get all input lines from a user */
        while (cin.getline(buf, sizeof(buf) )) {
           if ( ++numIn > max_size )  {
              max_size += 2;
              if ( !inList )
                 inList = (char**)calloc( max_size,sizeof(char*) );
              else
                 inList = (char**)realloc( inList,sizeof(char*)*max_size );
           }
           /* store an input line */
           inList[numIn-1] = (char*)malloc( strlen( buf )+1 );
           strcpy( inList[numIn-1], buf );
        }
        /* now print back all input lines from a user */
        while ( --numIn >= 0 )   {
           cout << numIn << ": " << inList[numIn] << endl;
           free( inList[numIn] );
        }
        free ( inList );
        return 0;
}
```

The above sample program reads input lines from the standard input. As it reads each input line it stores it into the *inList* array. The size of the *inList* array is adjusted dynamically based on the actual number of input lines read. The *max_size* and *numIn* variables contain the current size of the *InList* array and the number of input lines actually read, respectively. If the *numIn* value is the same as that of *max_size* and a new input line is read, the *max_size* value is increased by two, and the *inList* array size is enlarged by two more entries via the *realloc* function call.

After all input lines are read and end-of-file is encountered from the standard input, the sample program prints all saved lines to the standard output, in an order reverse to that found and deallocates all dynamic memory along the way.

Note that in the sample program, when the *inList* variable is first allocated, it is done via a *calloc* call. This is because if one attempts to allocate dynamic memory via *realloc* as:

```
char* memp = (char*)realloc( 0, new_size );
```

on some UNIX systems the *realloc* function will segmentation fault (e.g., Sun OS 4.1.x), whereas on other systems the function will work. Thus, to ensure portability, one should avoid assigning a NULL as the first argument value to any *realloc* call.

Finally, the dynamic memory allocated via *malloc, calloc,* and *realloc* may be managed differently than that allocated by the *new* operator. This means that memory allocated by *malloc, calloc,* or *realloc* should be deallocated via the *free* function only, and that allocated by the *new* operator should be discarded by the *delete* operator.

4.6 <time.h>

The <time.h> header declares a set of functions for system clock query. They can be used to obtain the local and the Universal Standard Time (UTC) date/time, as well as statistics of cpu (central processing unit) uses of processes.

The functions declared in the <time.h> header are:

time_t	***time***	(**time_t*** timv);
const char*	***ctime***	(**const time_t*** timv);
struct tm*	***localtime***	(**const time_t*** timv);
struct tm*	***gmtime***	(**const time_t*** timv);
const char*	***asctime***	(**const tm*** tm_p);
time_t	***mktime***	(**struct tm*** tm_p);
clock_t	***clock***	(**void**);

The *time* function returns the number of seconds elapsed since the official birthday of UNIX: January 1, 1970. The *time_t*-typed result is passed via the *timv* argument and via the function's return value if it is not NULL.

The *ctime* function returns the local date and time in the following example format:

```
"Sun Sept. 16 01:03:52 1995\n"
```

The *ctime* function is almost always used with the *time* function to obtain the local date/time:

```
time_t timv = time( 0 );
cout << "local time: " << ctime ( &timv );
```

The *localtime* and *gmtime* functions take an address of a *time_t*-typed variable and returns an address of a read-only *struct tm*-typed record. The *time_t*-typed input variable should be set by a prior time function call, and the returned *struct tm*-typed record contains the local and UTC date/time information, respectively. The *struct tm*-typed record may be passed to the *asctime* function to obtain a character string of the time stamp in the same format as that of the *ctime* returned value. The *struct tm* data type is defined in the <time.h> header.

The following statements illustrate the use of these functions:

```
#include <time.h>
time_t timv = time( 0 );
struct tm *local_tm = localtime( &timv );
struct tm *gm_tm = gmtime( &timv );
cout << "local time stamp: " << asctime( local_tm );
cout << "UTC time stamp: " << asctime ( gm_tm );
```

The following function returns the time stamp of any number of hours before or after the current time:

```
const char* time_stamp( long offset_hours )
{
      time_t timv = time ( 0 );              // get current time
      timv += offset_hours * 60 * 60;        // covert offset to sec
      return ctime( & timv );                // return offset time stamp
}
```

The *mktime* function is the inverse of the *localtime* and *gmtime* functions. It takes an address of a *struct tm*-typed record and returns a *time_t* value for it. Furthermore, the *mktime* functions normalize the data in the input argument to make them within range. The function is useful in getting the time stamp of any arbitrary date from 00:00:00 UTC, January 1, 1970 to 03:14:07 UTC, January 19, 2038, inclusively.

For example, the following program depicts the day of the week for April 5, 1999:

```
#include <iostream.h>
#include <time.h>
static struct tm time_str;                          // initializes all fields to 0
main()
{
    time_t tmv;
    time_str.tm_year     = 1999 - 1900;             // year = 1999
    time_str.tm_mon      = 4 - 1;                   // mon = April
    time_str.tm_mday     = 5;                       // day = 5
    if ( ( tmv=mktime(&time_str )) != -1 )     {
        timv[3] = NULL;
        cout << timv << endl;                       // should print "Mon"
    }
}
```

The definition of the *clock* function's return value may vary on different systems. The ANSI C standard defines the *clock* function returns the number of microseconds elapsed since a calling process started executing. However, on some UNIX systems, the *clock* function's return value is the number of microseconds elapsed since the process first called the *clock* function. Users should consult their system programmer's reference manual or the *clock* man page on their system for the exact definition.

However, the following *clock.C* program illustrates the correct use of the *clock* function to monitor process executing time, regardless of how the function is implemented on a given system:

```
#include <iostream.h>
#include <time.h>

main()
{
    time_t clock_tick = CLOCKS_PER_SECOND;
    clock_t start_time = clock();              // start timer

    /* do the normal work of the process ...*/
```

```
        clock_t elapsed_time = clock() - start_time;
        cout << "Run time: " << (elapsed_time / clock_tick) << endl;
    }
```

4.7 <assert.h>

The <assert.h> header declares a macro that can be used to assert some conditions in a process that should always be true. If, however, an assertion error occurs during a process execution, the macro flags an error message to the standard error port and indicates that physical line in which the source file assertion failed. After that, the macro aborts the process.

Thus, the *assert* macro can be a substantial time saver in helping users to debug their programs for checking "should not have occurred" conditions. Furthermore, the assert macros can be taken out in a released product just by specifying the -DNDEBUG switch when compiling the source code of the product

The following *assert.C* example statements illustrate the use of the *assert* macro:

```
#include <fstream.h>
#include <string.h>
#include <assert.h>
int main( int argc, char* argv[] )
{
    assert ( argc > 1 );                         // should have 1 arg
    ifstream ifs( argv[1] );
    assert( ifs.good() );                        // should be opened OK
    char *nam = new char[strlen(argv[1])+1];
    assert( nam );                               // should not be NULL
}
```

When the *assert.C* program is compiled and run with no argument, the console output is:

```
%    cc assert.C -o assert ; assert
Assertion failed: file "assert.C", line 5
```

The *assert* macro is defined in the <assert.h> as:

```
#ifndef NDEBUG
#define assert(ex) { if (!(ex)) { \
    fprintf(stderr,"Assertion failed: file: \"%s\", line %d\n", \
                __FILE__, __LINE__ ); exit(1); }
#endif
```

Note that in the above the *assert* is a macro and can be compiled away from users' program by defining the *NDEBUG* manifested constant.

4.8 <stdarg.h>

The <stdarg.h> header declares a set of macros that users may use to define variable argument functions. Examples of variable argument functions are *printf* and *scanf* in C. These functions may be called with one or more actual arguments, and the functions must get all those arguments to function correctly. This is accomplished by making use of the macros defined in the <stdarg.h> header.

The <stdarg.h> header defines the following macros:

```
#define va_start(ap,parm) (ap) = (char*)(&(parm) + 1)
#define va_arg(ap,type) ((type*)((char*)(ap) += sizeof(type)))[-1]
#define va_end(ap)
```

To use the above macros, a function must have one well-defined argument in its prototype. The *va_start* macro is called to set the *ap* argument to contain the location of the run-time stack where the next argument value after *parm* (which is the last known argument of the calling function) resides. The macro does that by taking the address of *parm* and adding the byte offset of the data to which *parm* points. This gives the address of the next function argument value after *parm*.

The *va_arg* is called to extract the next argument value in the run-time stack. For this to work, the caller must know the data type of the next argument on the stack. The *va_arg* macro does two things for each call:

- Advances *ap* to point to a stack location after the next argument to be returned
- Returns the next argument on the stack

For the first task, the operation:

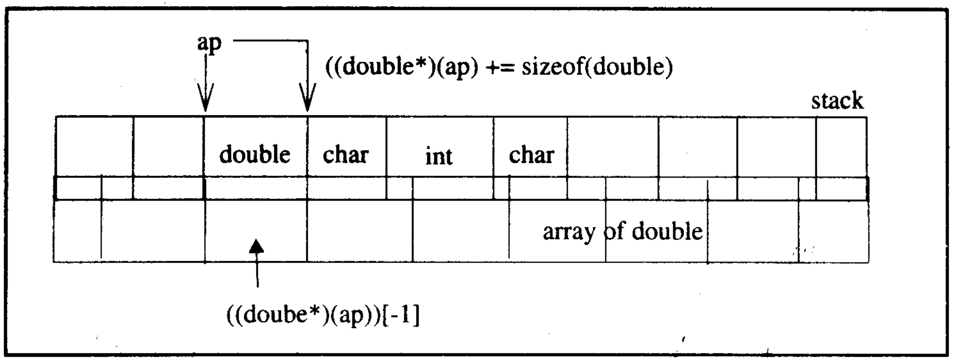
```
(char*)(ap) += sizeof(type)
```

typecasts *ap* to be a character pointer, then adds the size of the next argument data. This, in effect, advances *ap* to point to the address of the argument after the one to be fetched.

The second task is accomplished via the operation:

```
((type*)( (char*)(ap) += sizeof(type)) )[-1]
```

The above operation typecasts the *ap* that has been newly advanced to be an array of *type*. The *-1* index causes the next argument data on the stack to be returned to the caller. The following diagram illustrates the operations of *va_arg*; the next argument to be returned is assumed to be of type double:



The *va_end* is currently an NOP macro. It is defined to match the *va_start* macro and as a place-holder for any future extension of the *stdarg.h* functionality.

The following *pritnf.C* program contains the *my_printf* function, which emulates the *printf* function:

```
#include <iostream.h>
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <floatingpoint.h>

#define SHOW(s) fputs(s,stdout), cnt+=strlen(s)
```

```
/* my version of printf */
int my_printf ( const char* format, ...)
{
    char *tokp, ifmt[256], *hdr = ifmt, buf[256];
    int  cnt = 0;
    va_list pa;
    strcpy(ifmt,format);                // use local copy of the input text
    va_start(pa,format);                // pa points to args on stack

    while ( tokp=strchr( hdr,'%' ) )    {  // search for the '%' character
      *tokp++ = '\0';
      SHOW( hdr );                       // show leading text up to '%'
      if ( strchr( "dfgeisc%",*tokp )) {  // do if a legal format spec
        switch (*tokp)    {
          case 'd':                      // %i, %d
          case 'i':
                gconvert( (double)va_arg(pa,int),sizeof(buf),0,buf );
                break;
          case 's':                      // %s
                strcpy( buf,va_arg(pa,char*) );
                break;
          case 'c':                      // %c
                buf[0] = va_arg( pa,char );
                buf[1] = '\0';
                break;
          case 'f':                      // %f
                gconvert( va_arg( pa,double),8,1,buf );
                break;
          case 'g':                      // %g
                gconvert( va_arg( pa,double ),8,0,buf );
                break;
          case '%':                      // %%
                strcpy( buf,"%" );
                break;
        }
        SHOW(buf);                       // show the extracted argument
      }
      else    {                          // Show the character as is
          putchar( *tokp );
```

```
                cnt++;
            }
            hdr = tokp + 1;
        }
        SHOW(hdr);                          // show any last trailing text
        va_end( pa );
        return cnt;                         // return the no. of char. printed
}

int main()
{
        int cnt = my_printf( "Hello %% %s %zZZ\n", "world" );
        cout << "No: char: " << cnt << endl;
        cnt = my_printf( "There are %d days in %c year\n", 365, 'A' );
        cout << "No. char: " << cnt << endl;
        cnt = my_printf( "%g x %i = %f\n", 8.8, 8, 8.8*8 );
        cout << "No. char: " << cnt << endl;
        return 0;
}
```

In the above program, the *gconvert* function converts a double-type value to a character string format. This is a non-ANSI C standard function but is commonly available on most UNIX systems. The function prototype of *gconvert* is:

```
char* gconvert ( double dval, int ndigits, int trailing, char* buf );
```

The *dval* argument to *gconvert* contains the double-type value to be converted, and the *buf* argument specifies a user-defined buffer where the converted character string is placed. The *ndigits* argument defines the maximum number of significant digits that the *buf* argument may hold, and the *trailing* argument value may be 0 or 1, which determines whether or not any trailing decimal point or zero should be discarded. The function returns the buffer address as referenced in the *buf* argument.

The compilation and sample output of this test program is:

```
%    cc printf.c -o printf
%    printf
Hello % world zZZ
```

```
No. of char: 18
There are 365 days in A year
No. of char: 29
8.8 x 8 = 70.400000
No. of char: 20
```

Associated with the *va_arg* macros are the *vfprintf, vsprint,* and *vprintf* functions. These are similar to the *fprintf, sprintf,* and *printf* functions, respectively, except that they take *ap* as a pointer to the actual arguments of callers. These functions' prototypes are:

```
int    vprintf   ( const char* format, va_list ap );
int    vsprintf  ( char* buf, const char* format, va_list ap );
int    vfprintf  ( FILE* fp, const char* format, va_list ap );
```

These functions can be used to write a general message-reporting function:

```
/* source file: test_vfprintf.c */
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
typedef enum { INFO, WARN, ERROR, FATAL } MSG_TYPE_ENUM;
static int numErr, numWarn, numInfo;

void msg ( const MSG_TYPE_ENUM mtype, const char* format, ... )
{
    switch (mtype)     {
      case INFO:  numInfo++;
                  break;
      case WARN:  numWarn++;
                  fputs( "Warning: ",stderr );
                  break;
      case ERROR:numErr++;
                  fputs( "Error: ",stderr );
                  break;
      case FATAL: fputs( "Fatal: ", stderr );
                  break;
    }
    va_list pa;
    va_start( format, ap );
```

```
        vfprintf( stderr, format, pa );
        va_end( ap );
        if (mtype == FATAL ) exit( 2 );
}
/* Tets program for the msg function */
int main()
{
        msg( INFO, "Hello %% %s %%\n", "world" );
        msg( WARN, "There are %d days in %c year\n", 265, "A" );
        msg ( ERROR, "%g x %i = %f\n", 8.8, 8, 8.8*8 );
        msg( FATAL, "Bye-bye\n" );
}
```

The compilation and execution of this test program are:

```
%       cc test_vfprintf.c -o test_vfprintf
%       test_vfprintf
Hello % world %
Warning: There are 365 days in A year
Error: 8.8 x 8 = 70.400000
Fatal: Bye-bye
```

4.9 Command Line Arguments and Switches

The *getopt* function that is declared in the <stdlib.h> header may be used to implement programs that accept UNIX-style command line switches and arguments. Specifically, such programs' invocation syntax must be:

```
<program_name> [ -<switch> ... ] [ <argument> ... ]
```

All switches (or options) to a program must each begin with a "-" character and then a single letter (e.g., *-o*). Switch letters are case-sensitive. Multiple switches may be stacked such that: *-a -b* switches may be specified as *-ab* or *-ba*. A switch may be followed by an optional argument (e.g., *-o a.out*). If two or more switches are stacked, only the last switch specified may accept an argument. For example: *-o a.out -O* may be specified as *-Oo a.out*, but not as *-oO a.out*.

No switches may be specified after nonswitch arguments to a program. Thus the following invocation is incorrect, as the nonswitch argument */usr/prog/test.c* is specified before the *-o* switch.

```
%    a.out    -l    /usr/prog/test.c -o abc
```

If a program's invocation follows the above rules, the program may use the *getopt* function to extract switches and any of their associated arguments from the command line. The use of this is shown later on.

The *getopt* function and its associated global variables *opterr*, *optarg*, and *optind* are declared in the <stdlib.h> header:

```
extern int      optind, opterr;
extern char*    optarg;

int getopt ( int argc, char *const* argv[], const char* optstr);
```

The first two arguments to a *getopt* function call are the *argc* and *argv* variables of the *main* function, respectively. The *optstr* argument contains a list of switch letters that are legal to the program. The function scans the *argv* vector and looks for switches that are defined in *optstr*. For each call of *getopt*, the function returns a switch letter that is found in *argv* and is defined in *optstr*. If a switch is specified as *<switch_letter>:* in *optstr*, then the switch, if specified, must be accompanied by an argument, and that argument can be obtained via the *optarg* global pointer.

If a switch is found in *argv* but is not listed in *optstr*, the *getopt* function will flag an error message to the standard error port, and the function returns the "?" character. However, if a user sets the *opterr* global variable to be nonzero before calling *getopt*, the function will be silent for subsequent illegal switches found in *argv*.

Finally, when no more switches are found in *argv*, the *getopt* function returns the EOF value, and *optind* is set to point to the entry in *argv* where the first non-switch command line argument is stored. If *optind* is same as *argc*, there are no nonswitch arguments to a program.

The following *test_getopt.C* program accepts the *-a*, *-b*, and *-o* switches. If the *-o* switch is specified, there should also be a file name specified with it:

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
static char* outfile;
static int a_flag, b_flag;
```

```
int main( int argc, char* argv[] )
{
      int ch;
      while ( (ch=getopt( argc, argv,"o:ab" )) != EOF )
         switch ( ch )    {
            case 'a':     a_flag = 1;              // found -a
                          break;
            case 'b':     b_flag = 1;              // found -b
                          break;
            case 'o':     outfile = new char[strlen(optarg)+1];
                          strcpy(outfile,optarg);  // found -o <file>
                          break;
            case '?':     /* let getopt flags an eror message */
            default:      break;                   // an illegal switch
      }
      /* no more switches. Scan the rest of non-switch arguments */
      for ( ; optind < argc; optind++ )
         cout << " non-switch argument: " << argv[optind] << endl;

      return 0;
}
```

The compilation and sample runs of the *test_getopt.C* programs are:

```
%   CC  test_getopt.c   -o      test_getopt
%   test_getopt
%   test_getopt  -abo   xyz     /etc/hosts
non-switch argument: /etc/hosts
%   test_getopt  -xay   -bz     /usr/lib/libc.a
a.out: illegal option -- x
a.out: illegal option -- y
a.out: illegal option -- z
 non-switch argument: /usr/lib/libc.a
```

The limitations of *getopt* are: (1) switches must use single letter only; (2) switches must either have associated arguments or none; users cannot define switches that may optionally accept argument; (3) the function does not check the data type of switch arguments; (4) users may not specify mutually exclusive switches.

Despite the above limitations, *getopt* is valuable in saving users program development and debug time, and ensuring that users' programs follow the UNIX invocation convention

4.10 <setjmp.h>

The <setjmp.h> header declares a set of functions that allow a process to do *goto* from one function to another. Recalling a C *goto* statement only allows a process to transfer process flow from one statement to another within the same function. The functions defined in the <setjmp.h> header eliminate this restriction. These functions should be used sparingly. For example, if an error is detected in a deeply recursive function, it makes sense to report the error and then do a *goto* to the main function, so as to start the processing over again. This is what a UNIX shell does if an error is detected in one of its subshells. In this circumstance, the <setjmp.h> functions offer efficient error recovery and save users from adding layers and layers of error-checking code for error recovery. However, like the problem of using *goto*, if these functions are used without discipline in a program, it will become difficult for users to track program flow.

The <setjmp.h> header defines the following functions:

```
int setjmp ( jmp_buf loc );
void longjmp ( jmp_buf loc, int val );
```

The *setjmp* function records a location in a program code where the future *goto* (via the *longjmp* call) will return. The *jmp_buf* data type is defined in the <setjmp.h> header, and the *loc* argument records the location of the *setjmp* statement. If a user wishes to define multiple locations in a program where the future *longjmp* call can return, each location must be recorded in a *jmp_buf*-typed variable and is set by a *setjmp* call.

The *setjmp* function always returns zero when it is called directly in a process.

The *longjmp* function is called to transfer a program flow to a location that was stored in the *loc* argument. The program code marked by *loc* must be in a function that is among the callers of the current function. When the process is "jumping" to that target function, all the stack space used by the current function and its callers up to the target function are discarded by the *longjmp* function. The process resumes execution by reexecuting the *setjmp* statement in the target function that is marked by *loc*. The return value of the *setjmp* function is the *val* value, as specified in the *longjmp* function call. The *val* value should be nonzero (if it is zero, it is set to one by the *setjmp* function) so that it can be used to indicate where and why the *longjmp* function was invoked, and a process can do error-handling accordingly.

The following *test_setjmp.C* program illustrates the use of *setjmp* and *longjmp* functions:

```
/* source file name: test_setjmp.C */
#include <iostream.h>
#include <setjmp.h>
static jmp_buf loc;
int main()
{
      int retcode, foo();

      if ( (retcode=setjmp( loc )) != 0 )    {          // error recovery
         cerr << "Get here from longjmp. retcode=" << retcode << endl;
         return 1;
      }
      /* normal flow of program */
      cout << "Program continue after setting loc via setjmp...\n";
      foo();
      cout << "Should never get here ....\n";
      return 1;
}
int foo()
{
      cout << "Enter foo. Now call longjmp....\n";
      longjmp (loc, 5);
      cout << "Should never gets here....\n";
      return 2;
}
```

The compilation and output of the *test_setjmp.C* program are:

```
%    cc test_setjmp.c    -o test_setjmp
%    test_setjmp
Program continue after setting loc via setjmp...
Enter foo. Now call longjmp....
Get here from longjmp. retcode=5
```

4.11 <pwd.h>

The <pwd.h> defines a set of functions for users to obtain user accountant information, as specified in the UNIX */etc/passwd* file. The functions defined in the <pwd.h> header are:

```
const struct passwd*  getpwnam ( const char* user_name);
const struct passwd* getpwuid( const int uid );
int setpwent ( void );
const struct passwd* getpwent ( void );
int endpwent ( void );
const struct passwd* fgetpwent ( FILE * fptr );
```

The *struct passwd* data type is defined in the <pwd.h> as:

```
struct passwd
{
        char*           pw_name;        // user's login name
        char*           pw_passwd:      // encrypted password
        int             pw_uid;         // user's user ID
        int             pw_gid;         // user's group ID
        char*           pw_age;         // password aging info
        char*           pw_comment;     // general user's info
        char*           pw_dir;         // user's home directory
        char*           pw_shell;       // user's login shell
};
```

Each *struct passwd* record contains data from one line of the */etc/password* file. This contains the account information of one user on a UNIX system. Specifically, the information consists of a user's login name, assigned user ID, group ID, login shell, home directory, and login password (in encrypted form), etc.

The *getpwnam* function takes a user login name as argument and returns a pointer to a *struct passwd*-typed record that contains that user's information if the user is defined on the system on which this function call is made. Conversely, it returns a NULL pointer if the given user name is invalid.

The following statement depicts the home directory of a user name *joe*:

```
struct passwd *pwd = getpwnam( "joe" );
if ( !pwd )
     cerr << "'joe' is not a valid user on this system\n";
else
     cout << pwd->pw_name << ", home=" << pwd->pw_dir << endl;
```

The *getpwuid* function takes a user *UID* as argument, and returns a pointer to a *struct passwd*-typed record that contains the user's information if the user is defined on the system on which this function call is made. Again, it returns a NULL pointer if the given user *UID* is invalid.

The following statement depicts the user name and login shell of a user whose *UID* is 15:

```
struct passwd *pwd = getpwuid( 15 );
if ( !pwd )
     cerr << "'15' is not a valid UID on this system\n";
else
     cout << pwd->pw_name << ", shell=" << pwd->pw_shell << endl;
```

The *setpwent* function resets an internal file pointer to point to the beginning of the */etc/passwd* file. The *getpwent* function returns a pointer to a *struct passwd*-type record, which contains the next entry of the */etc/passwd* file. When all entries in a */etc/passwd* are scanned by the *getpwent* function, it returns a NULL pointer to indicate end-of-file. The *endpwent* function is called to close the internal file pointer, which references the */etc/passwd* file.

The following *test_pwd.C* program dumps out all defined users and their *UID* and *GID* information to the standard output:

```
#include <iostream.h>
#include <pwd.h>
int main()
{
     setpwent();
     for ( struct passwd *pwd; pwd=getpwent(); )
        cout << pwd->pw_name << ", UID: " << pwd->pw_uid
             << ", GID: " << pwd->pw_gid << endl;
     endpwent();
     return 0;
}
```

Finally, the *fgetpwent* function is like the *getpwent*, except here users supply a file pointer that references a file having the same syntax of the */etc/passwd* file. The function returns an user accountant data at the next entry in the given file. The *setpwent* and *endpwent* functions are not used with this *fgetpwent* function.

4.12 <grp.h>

The <grp.h> defines a set of functions for users to obtain group accountant information as specified in the UNIX */etc/group* file. The functions defined in the <group.h> header are:

```
const struct group*    getgrnam ( const char* group_name);
const struct group* getgrgid( const int gid );
int setgrent ( void );
const struct group* getgrent ( void );
int endgrent ( void );
const struct group* fgetgrent ( FILE * fptr );
```

The *struct group* data type is defined in the <grp.h> header as:

```
struct group
{
        char*           gr_name;        // group name
        char*           gr_passwd:      // group encrypted password
        int             gr_gid;         // group ID
        char**          gr_comment;     // group member names
};
```

Each *struct group* record contains data from one line of the */etc/group* file. This contains the account information of one group on a UNIX system. Specifically, the information consists of a group name, assigned group ID, and a list of those user names that are members of the group.

The *getgrnam* function takes a group name as argument and returns a pointer to a *struct group*-typed record that contains that group's information if the group is defined on the system on which this function call is made. It returns a NULL pointer if the given group name is invalid.

The following statement depicts the group ID of a group name *developer*:

```
struct group *grp = getgrnam( "developer" );
if ( !grp )
        cerr << "'developer' is not a valid group on this system\n";
else
        cout << grp->gr_name << ", GID=" << grp->gr_gid << endl;
```

The *getgrgid* function takes a group ID as argument and returns a pointer to a *struct group*-typed record that contains that group's information if the group is defined on the system on which this function call is made. On the contrary, it returns a NULL pointer if the given group ID is invalid.

The following statement depicts the group members of a group whose *GID* is 200:

```
struct group *grp = getgrgid( 200 );
if ( !grp )
        cerr << "'200' is not a valid GID on this system\n";
else for ( int i=0; grp->pw_comment && grp->pw_comment[i]; i++ )
        cout << grp->gr_comment[i] << endl;
```

The *getgrent* function resets an internal file pointer to point to the beginning of the */etc/group* file. The *getgrwent* function returns a pointer to a *struct group*-type record that contains information at the next entry of the */etc/group* file. When all entries in a */etc/group* are scanned by the *getgrwent* function, it returns a NULL pointer to indicate end-of-file. The *endgrent* function is called to close the internal file pointer, which references the */etc/group* file.

The following *test_grp.C* program dumps all defined groups and their *GID* information to the standard output:

```
#include <iostream.h>
#include <grp.h>
int main()
{
        setgrent();
        for ( struct group*grp; grp=getgrent(); )
            cout << grp->gr_name << ", GID: " << grp->gr_gid << endl;
        endgrent();
        return 0;
}
```

Finally, the *fgetgrent* function is like the *getgrent*, except here users supply a file pointer that references a file having the same syntax of the */etc/group* file. The function returns a group accountant data at the next entry in the given file. The *setgrent* and *endgrent* functions are not used with this *fgetgrent* function.

4.13 <crypt.h>

The <crypt.h> header declares a set of functions for data encryption and decryption. These are important functions for maintaining system security. For example, user passwords and system data files that need high security must be encrypted so that no unauthorized person can easily find out what they are. Furthermore, authorized persons must know the secret keys to decrypt these objects so that they can read and modify them.

The <crypt.h> header declares the following functions:

```
char*   crypt    (const char* key, const char* salt );
void    setkey   ( const char salt[64] );
void    encrypt  ( char key[64], const int flag );
```

The *crypt* function is used on UNIX systems to encrypt user passwords and to check for user login password validity. The function takes two arguments. The first argument, *key*, is a user-defined password. The second argument, *salt*, is used to encode the resultant encrypted string. The *salt* argument value consists of two characters from the following character set:

'a' to 'z', 'A' to 'Z', '0' to '9', or '/'

If the function is called by the *password* process on UNIX to encode a new user password, the process supplies a randomly generated *salt* value. The resultant encoded string is in the format of:

<salt><encrypted password string>

Then, when a user attempts to login to a system by supplying a user name and a password, the login process checks the authentication of the user as follows:

```
#include <iostream.h>
#include <crypt.h>
#include <pwd.h>
#include <string.h>
```

```
int check_login( const char* user_name, const char* password )
{
      struct passwd* pw;
      if ( !(pw=getpwnam( user_name )) ) {
         cerr << "Invalid login name: " << user_name << endl;
         return 0;                    // authentication fails
      }
      char* new_pw = crypt( password,pw->pw_passwd );
      if ( strcmp( new_pw,pw->pw_passwd ) ) {
         cerr << "Invalid password: " << password << endl;
         return 0;                    // authentication fails
      }
      /* both user name and password are valid */
      return 1;
}
```

In the above example, the function is called to ensure that the given user login name and password are valid. The function returns 1 if they are valid, 0 otherwise. The function calls the *getpwnam* function to convert a given user name to a pointer to the *struct passwd*-typed record. If the user name is valid, the *getpwnam* function returns a non-NULL value, otherwise, it returns NULL value, and the *check_login* function returns a failure status.

After a *struct passwd*-typed record is obtained, the *check_login* function calls *crypt* to encrypt the given password. The *salt* supplied to the *crypt* call is the *pw_passwd* field of the *struct passwd*-type record. This is the encrypted password of the user, and the first two characters of this string are the *salt* that was used to generate the encrypted password. The return value of *crypt* is an encrypted password string, and it is compared against the *pw_passwd* value. If they match, the given password is valid, otherwise, the *check_login* function returns a failure status.

As can be seen from the above example, the *crypt* function does not decrypt strings. However, it can be used to encrypt a new string and then compare that against an old string to verify the old string content. It is important to note that the first two characters of the old encrypted string are the *salt* used to generate it. If the new string is the same as the old one prior to encryption and is encrypted with a different salt value, the resultant encrypted new string will be different than the old one.

The *setkey* and *encrypt* functions perform a function similar to *crypt*, except they use the National Bureau of Standards (NBS) data encryption standard (DES) algorithm, which is more secure than the algorithm used by *crypt*. The *setkey* function argument is a character array of 64 entries. Each of these entries should contain an integer value of either 1 or 0, which is one bit of an eight byte *salt* value. The *encrypt* function first argument *key* is a character array of 64 entries, each of these entries contains one bit of an eight byte *key* value to be encrypted (if the third argument *flag* value is 0) or decrypted (if the *flag* value is 1). The resultant encrypted or decrypted string is passed back to the caller in the same *key* argument. The *encrypt* function can process up to only eight characters on each call.

4.14 Summary

This chapter described the ANSI C library functions and some UNIX-specific C library functions. These functions are not covered by the C++ standard classes or by the UNIX and POSIX APIs. Thus, by knowing these functions, users may make use of them to save application development time and to ensure high portability and low maintenance of their end products. Examples were included in this chapter to illustrate the uses of some of these C library functions.

As mentioned in the beginning of the chapter, the limitations of these C library functions are that they do not provide enough functions for users to develop system-level applications. Users must use UNIX and POSIX APIs to create such applications. The remainder of the book is devoted to describing the UNIX and POSIX APIs and the advanced usage of them for system-level programming. In addition to that, examples will be shown on how to use these APIs to implement some of the standard C library functions described in this chapter.

UNIX and POSIX APIs

Unix systems provide a set of application programming interface functions (commonly known as *system calls*) which may be called by users' programs to perform system-specific functions. These functions allow users' applications to directly manipulate system objects such as files and processes that cannot be done by using just standard C library functions. Furthermore, many of the UNIX commands, C library functions, and C++ standard classes (e.g., the iostream class) call these APIs to perform the actual work advertised. Thus, users may use these APIs directly to by-pass the overhead of calling the C library functions and C++ standard classes, or to create their own versions of the UNIX commands, C library functions and C++ classes.

Most UNIX systems provioe a common set of APIs to perform the following functions:

- Determine system configuration and user informatior
- Files Manipulation
- Processes creation and control
- Interprocess communication
- Network communication

Most UNIX APIs access their UNIX kernel's internal resources. Thus, when one of these APIs is invoked by a process (a process is a user's program under execution), the execution context of the process is switched by the kernel from a user mode to a kernel mode. A user mode is the normal execution context of any user process, and it allows the process to access its process-specific data only. A kernel mode is a protective execution environment that allows a user process to access kernel's data in a restricted manner. When the API execution completes, the user process is switched back to the user mode. This context switching for each API call ensures that processes access kernel's data in a controlled manner, and minimizes any chance of a run-away user application may damage an entire system. In general, calling an API is more time-consuming than calling a user function due to the context switching. Thus, for those time-critical applications, users should call their system APIs only if it is absolute necessary.

5.1 The POSIX APIs

Most POSIX.1 and POSIX.1b APIs are derived from UNIX APIs. However, the POSIX committees do create their own APIs when there is perceived deficiency of the UNIX APIs. For example, the POSIX.1b committee creates a new set of APIs for interprocess communication using messages, shared memory, and semaphores. There are equivalent constructs for messages, shared memory, and semaphores in System V UNIX, but the latter constructs use a nonpathname-naming scheme to identify these IPC facilities, and processes cannot use these IPCs to communicate across a LAN. Thus, the POSIX.1b committee created a different version of messages, shared memory, and semaphores that eliminated these short-coming.

In general, POSIX APIs uses and behaviors are similar to those of UNIX APIs. However, users' programs should define the _POSIX_SOURCE (for POSIX.1 APIs) and/or _POSIX_C_SOURCE (for both POSIX.1 and POSIX.1b APIs) in their programs to enable the POSIX APIs declarations in header files that they include.

5.2 The UNIX and POSIX Development Environment

The <unistd.h> header declares some commonly used POSIX.1 and UNIX APIs. There is also a set of API-specific headers placed under the <sys> directory (on a UNIX system it is the /usr/include/sys directory). These <sys/...> headers declare special data types for data objects manipulated by both the APIs and by users' processes. In addition to these, the <stdio.h> header declares the *perror* function, which may be called by a user process whenever an API execution fails. The *perror* function prints a system-defined diagnostic message for any failure incurred by the API.

Most of the POSIX.1, POSIX.1b, and UNIX API object code is stored in the *libc.a* and *libc.so* libraries. Thus, no special compile switch need be specified to indicate which archive or shared library stores the API object code. However, some network communication APIs' object code is stored in special libraries on some systems (e.g., the socket APIs are stored in *libsocket.a* and *libsocket.so* libraries on Sun Mircosystems Solaris 2.x system). Thus, users should consult their system programmer's reference manuals for the special header and library needed for the APIs they use on their systems.

5.3 API Common Characteristics

Although the POSIX and UNIX APIs perform diverse system functions on behalf of users, most of them returns an integer value which indicates the termination status of their execution. Specifically, if a API returns a -1 value, it means the API's execution has failed, and the global variable *errno* (which is declared in the <errno.h> header) is set with an error code. A user process may call the *perror* function to print a diagnostic message of the failure to the standard output, or it may call the *strerror* function and gives it *errno* as the actual argument value, the *strerror* function returns a diagnostic message string and the user process may print that message in its preferred way (e.g., output to a error log file).

The possible error status codes that may be assigned to *errno* by any API are defined in the <errno.h> header. When a user prints the man page of a API, it usually shows the possible error codes that may be assigned to *errno* by the API, and the reason why. Since this information is readily available to users and they may be different on different systems, this book will not describe the *errno* values for individual API in any details. However, the following is a list of commonly occur error status codes and their meanings:

Error status code	**Meaning**
EACCESS	A process does not have access permission to perform an operation via a API
EPERM	A API was aborted because the calling process does not have the superuser privilege
ENOENT	An invalid file name was specified to an API
BADF	A API was called with an invalid file descriptor
EINTR	A API execution was aborted due to a signal interruption (see Chapter 9 for the explanation of signal interruption)
EAGAIN	A API was aborted because some system resource it requested was temporarily unavailable. The API should be called again later.
ENOMEM	A API was aborted because it could not allocate dynamic memory
EIO	I/O error occurred in a API execution
EPIPE	A API attempted to write data to a pipe which has no reader
EFAULT	A API was passed an invalid address in one of its arguments
ENOEXEC	A API could not execute a program via one of the exec API
ECHILD	A process does not have any child process which it can wait on

If an API execution is successful, it returns either a zero value or a pointer to some data record where user-requested information is stored.

5.4 Summary

This chapter gives an overview of UNIX and POSIX APIs and describes the common uses and characteristics of these APIs. These APIs are powerful, and they enable users to develop advanced system programs that manipulate system objects (e.g., files and processes) in more ways than can be done via the standard C library functions and C++ classes alone. Furthermore, users may use these APIs to create their own library or C++ classes or their own versions of shell commands to augment those supplied by a system. However, most of these APIs involve context switching of user processes between user mode and kernel mode: thus, this is a time penalty in using these APIs.

The rest of the book examines the UNIX and POSIX APIs in more detail. The APIs are for file manipulation, process manipulation, interprocess communication, and remote procedure call. Examples will be shown on how to use these APIs to construct user's own versions of C library functions and UNIX shell commands, and also to create C++ classes to make abstract data types for system objects such as processes and for system functions like interprocess communication.

CHAPTER 6

UNIX Files

Files are the building blocks of any operating system, as most operations in a system invariably deal with files. When you execute a command in UNIX, the UNIX kernel fetches the corresponding executable file from a file system, loads its instruction text to memory, and creates a process to execute the command on your behalf. Furthermore, in the course of execution, a process may read from or write to files. All these operations involve files. Thus, the design of an operating system always begins with an efficient file management system.

Files in UNIX and POSIX systems cover a wide range of file types. These include text files, binary files, directory files, and device files. Furthermore, UNIX and POSIX systems provide a set of common system interfaces to files, such that they can be handled in a consistent manner by application programs. This, in turn, simplifies the task of developing application programs on those systems.

This chapter will explore the various file types in UNIX and POSIX systems and will show how they are created and used. Moreover, there is a set of common file attributes that an operating system keeps for each file in the system -- these attributes and their uses are explained in detail. Finally, the UNIX System V kernel and process-specific data structures used to support file manipulation are described to tie in the system call interface for files. The UNIX and POSIX.1 system calls for file handling are discussed in the next chapter.

6.1 File Types

A file in a UNIX or POSIX system may be one of the following types:

- Regular file
- Directory file
- FIFO file
- Character device file
- Block device file

A ***regular file*** may be either a text file or a binary file. UNIX and POSIX systems do not make any distinction between these two file types, and both may be "executable", provided that the execution rights of these files are set and that these files may be read or written to by users with the appropriate access permission.

Regular files may be created, browsed through, and modified by various means such as text editors or compilers, and they can be removed by specific system commands (e.g., *rm* in UNIX).

A ***directory file*** is like a file folder that contains other files, including subdirectory files. It provides a means for users to organize their files into some hierarchical structure based on file relationship or uses. For example, the UNIX */bin* directory contains all system executable-programs, such as *cat, rm, sort,* etc.

A directory may be created in UNIX by the *mkdir* command. The following UNIX command will create the */usr/foo/xyz* directory if it does not exist:

```
mkdir   /usr/foo/xyz
```

A UNIX directory is considered empty if it contains no other files except the "." and ".." files, and it may be removed via the *rmdir* command. The following UNIX command removes the */usr/foo/xyz* directory if it exists:

```
rmdir   /usr/foo/xyz
```

The content of a directory file may be displayed in UNIX by the *ls* command.

A ***block device file*** represents a physical device that transmits data a block at a time. Examples of block devices are hard disk drives and floppy disk drives. A ***character device file***, on the other hand, represents a physical device that transmits data in a character-based manner. Examples of character devices are line printers, modems, and consoles. A physical device may have both block and character device files representing it for different access methods. For example, a character device file for a hard disk is used to do raw (nonblocking) data transfer between a process and the disk.

An application program may perform read and write operations on a device file in the same manner as on a regular file, and the operating system will automatically invoke an appropriate device driver function to perform the actual data transfer between the physical device and the application.

Note that a physical device may have both a character and a block device file refer to it, so that an application program may choose to transfer data with that device by either a character-based (via the character device file) or block-based (via the block device file) method.

A device file is created in UNIX via the *mknod* command. The following UNIX command creates a character device file with the name */dev/cdsk0*, and the major and minor numbers of the device file are 115 and 5, respectively. The argument *c* specifies that the file to be created is a character device file:

```
mknod   /dev/cdsk   c   115   5
```

A major device number is an index to a kernel table that contains the addresses of all device driver functions known to the system. Whenever a process reads data from or writes data to a device file, the kernel uses the device file's major number to select and invoke a device driver function to carry out the actual data transfer with a physical device. A minor device number is an integer value to be passed as an argument to a device driver function when it is called. The minor device number tells the device driver function what actual physical device it is talking to (a driver function may serve multiple physical device types), and the I/O buffering scheme to be used for data transfer.

Device driver functions are supplied either by physical device vendors or by operating system vendors. Whenever a device driver function is installed to a system, the operating system kernel will require reconfiguration. This scheme allows an operating system to be extended at any customer site to handle any new device type preferred by users.

A block device file is also created in UNIX by the *mknod* command, except that the second argument to the *mknod* command will be *b* instead of *c*. The *b* argument specifies that the file to be created is a block device file. The following command creates a */dev/bdsk* block device file with the major and minor device numbers of 287 and 101, respectively:

```
mknod   /dev/bdsk   b   287   101
```

In UNIX, *mknod* must be invoked through superuser privileges. Furthermore, it is conventional in UNIX to put all device files in either the */dev* directory or a subdirectory beneath it.

A ***FIFO file*** is a special pipe device file which provides a temporary buffer for two or more processes to communicate by writing data to and reading data from the buffer. Unlike regular files, however, the size of the buffer associated with a FIFO file is fixed to PIPE_BUF. (PIPE_BUF and its POSIX.1 minimum value, _POSIX_PIPE_BUF, are defined in the <limits.h> header). A process may write more than PIPE_BUF bytes of data to a FIFO file, but it may be blocked when the file buffer is filled. In this case the process must wait for a reader process to read data from the pipe and make room for the write operation to complete. Finally, data in the buffer is accessed in a first-in-first-out manner, hence the file is called a FIFO.

The buffer associated with a FIFO file is allocated when the first process opens the FIFO file for read or write. The buffer is discarded when all processes which are connected to the FIFO close their references (e.g., stream pointers) to the FIFO file. Thus the data stored in a FIFO buffer is temporary; they last as long as there is one process which has a direct connection to the FIFO file for data access.

A FIFO file may be created in UNIX via the *mkfifo* command. The following command creates a FIFO file called */usr/prog/fifo_pipe* if it does not exist:

```
mkfifo    /usr/prog/fifo_pipe
```

In some early versions of UNIX (e.g., UNIX System V.3), FIFO files were created via the *mknod* command. The following UNIX command creates the */usr/prog/fifo_pipe* FIFO file if it does not exist:

```
mknod    /usr/prog/fifo_pipe    p
```

The UNIX System V.4 supports both the *mknod* and *mkfifo* commands, whereas BSD UNIX supports only the *mkfifo* command to create FIFO files.

A FIFO file may be removed like any regular file. Thus FIFO files can be removed in UNIX via the *rm* command.

Beside the above file types, BSD UNIX and UNIX System V.4 also define a ***symbolic link file*** type. A symbolic link file contains a path name which references another file in either the local or a remote file system. POSIX.1 does not yet support symbolic link file type, although it has been proposed to be added to the standard in a future revision.

A symbolic link may be created in UNIX via the *ln* command. The following command creates a symbolic link */usr/mary/slink* which references the file */usr/jose/original*. The *cat* command which follows will print the content of the */usr/jose/original* file:

```
ln   -s   /usr/jose/original      /usr/mary/slink
cat  -n   /usr/mary/slink
```

The path name referenced by a symbolic link may be depicted in UNIX via the *ls -l* command on the symbolic link file. The following command will show that */usr/mary/slink* is a symbolic link to the */usr/jose/original* file:

```
%   ls   -l   /usr/mary/slink
sr--r--r--  1   terry     20 Aug 20, 1994    slink -> /usr/jose/original
%
```

It is possible to create a symbolic link to reference another symbolic link. When symbolic links are supplied as arguments to the UNIX commands *vi, cat, more, head, tail,* etc., these commands will dereference the symbolic links to access the actual files that the links reference. However, the UNIX commands *rm, mv,* and *chmod* will operate only on the symbolic link arguments directly and not on the files that they reference.

6.2 The UNIX and POSIX File Systems

Files in UNIX or POSIX systems are stored in a tree-like hierarchical file system. The root of a file system is the root directory, denoted by the "/" character. Each intermediate node in a file system tree is a directory file. The leaf nodes of a file system tree are either empty directory files or other types of files.

The absolute path name of a file consists of the names of all the directories, specified in the descending order of the directory hierarchy, starting from "/," that are ancestors of the file. Directory names are delimited by the "/" characters in a path name. For example, if the path name of a file is */usr/xyz/a.out*, it means that the file *a.out* is located in a directory called *xyz*, and the *xyz* directory is, in turn, stored in the *usr* directory. Furthermore, the *usr* directory is in the "/" directory.

A relative path name may consist of the "." and ".." characters. These are references to the current and parent directories, respectively. For example, the path name *../../.login* denotes a file called *.login*, which may be found in a directory two levels up from the current directory. Although POSIX.1 does not require a directory file to contain "." and ".." files, it does specify that relative path names with "." and ".." characters be interpreted in the same manner as in UNIX.

A file name may not exceed NAME_MAX characters, and the total number of characters of a path name may not exceed PATH_MAX. The POSIX.1-defined minimum values for

NAME_MAX and PATH_MAX are _POSIX_NAME_MAX and _POSIX_PATH_MAX, respectively. These are all defined in the <limits.h> header.

Furthermore, POSIX.1 specifies the following character set is to be supported by all POSIX.1-compliant operating systems as legal file name characters. This means application programs that are to be ported to POSIX.1 and UNIX systems should manipulate files with names in the following character set only:

```
A to Z     a to z     0 to 9
```

The path name of a file is called a *hard link*. A file may be referenced by more than one path name if a user creates one or more hard links to the file using the UNIX *ln* command. For example, the following UNIX command creates a new hard link */usr/prog/new/n1* for the file */usr/foo/path1*. After the *ln* command, the file can be referenced by either path name.

```
ln     /usr/foo/path1     /usr/prog/new/n1
```

Note that if the *-s* option is specified in the above command, the /usr/prog/n1 will be a symbolic link instead of a hard link. The differences between hard and symbolic links will be explained in Chapter 7.

The following files are commonly defined in most UNIX systems, although they are not mandated by POSIX.1:

File	**Use**
/etc	Stores system administrative files and programs
/etc/passwd	Stores all user information
/etc/shadow	Stores user passwords (For UNIX System V only)
/etc/group	Stores all group information
/bin	Stores all the system programs like cat, rm, cp, etc.
/dev	Stores all character and block device files
/usr/include	Stores standard header files
/usr/lib	Stores standard libraries
/tmp	Stores temporary files created by programs

6.3 The UNIX and POSIX File Attributes

Both UNIX and POSIX.1 maintain a set of common attributes for each file in a file system. These attributes and the data they specify are:

Attribute	**Value meaning**
file type	Type of file
access permission	The file access permission fcr owner, group, and others
Hard link count	Number of hard links of a file
UID	The file owner user ID
GID	The file group ID
file size	The file size in bytes
last access time	The time the file was last accessed
last modify time	The time the file was last modified
last change time	The time the file access permission, UID, GID, or hard link count was last changed
inode number	The system inode number of the file
file system ID	The file system ID where the file is stored

Most of the above information can be depicted in UNIX by the *ls -l* command on any files.

The above attributes are essential for the kernel to manage files. For example, when a user attempts to access a file, the kernel matches the user's UID and GID against those of the file to determine which category (user, group, or others) of access permission should be used for the access privileges of the user. Furthermore, the last modification time of files is used by the UNIX *make* utility to determine which source files are newer than their corresponding executable files and require recompilation.

Although the above information is stored for all file types, not all file types make use of the information. For example, the file size attribute has no meaning for character and block device files.

In addition to the above attributes, UNIX systems also store the major and minor device numbers for each device file. In POSIX.1, the support of device files is implementation-dependent; thus, it does not specify major and minor device numbers as standard attributes for device files.

All the above attributes are assigned by the kernel to a file when it is created. Some of these attributes will stay unchanged for the entire life of the file, whereas others may change as the file is being used. The attributes that are constant for any file are:

- File type
- File inode number

- File system ID
- Major and minor device number (for device files on UNIX systems only)

The other attributes are changed by the following UNIX commands or system calls:

UNIX command	System call	Attributes changed
chmod	chmod	Changes access permission, last change time
chown	chown	Changes UID, last change time
chgrp	chown	Changes GID, last change time
touch	utime	Changes last access time, modification time
ln	link	Increases hard link count
rm	unlink	Decreases hard link count. If the hard link count is zero, the file will be removed from the file system
vi, emac		Changes file size, last access time, last modification time

6.4 Inodes in UNIX System V

Two of the file attributes which were mentioned but not explained in the above are the inode number and the file system ID. One may also notice that file names are not part of the attributes which an operating system keeps for files. This section will use UNIX System V as the context to give answers to all these puzzles.

In UNIX System V, a file system has an inode table which keeps tracks of all files. Each entry of the inode table is an inode record which contains all the attributes of a file, including an unique inode number and the physical disk address where the data of the file is stored. Thus if a kernel needs to access information of a file with an inode number of, say 15, it will scan the inode table to find an entry which contains an inode number of 15, in order to access the necessary data. Since an operating system may have access to multiple file systems at one time (they are connected to the operating system via the *mount* system command, and each is assigned an unique file system ID), and an inode number is unique within a file system only, a file inode record is identified by a file system ID and an inode number.

An operating system does not keep the name of a file in its inode record, because the mapping of file names to inode numbers is done via directory files. Specifically, a directory file contains a list of names and their respective inode numbers for all files stored in that directory. For example, if a directory *foo* contains files *xyz*, *a.out*, and *xyz_ln1*, where *xyz_ln1* is a hard link of *xyz*, the content of the directory *foo* is shown in Figure 6.1 (most implementation-dependent data is omitted).

To access a file, for example /usr/joe, the UNIX kernel always knows the "/" directory inode number of any process (it is kept in a process U-area and may be changed via the *chdir* system call). It will scan the "/" directory file (via the "/" inode record) to find the inode number of the *usr* file. Once it gets the *usr* file inode number, it checks that the calling process has permission to search the *usr* directory and accesses the content of the *usr* file. It then looks for the inode number of the *joe* file.

Whenever a new file is created in a directory, the UNIX kernel allocates a new entry in the inode table to store the information of the new file. Moreover, it will assign a unique inode number to the file and add the new file name and inode number to the directory file that contains it.

inode number	*file name*
115	
89	..
201	*xyz*
346	*a.out*
201	*xyz_ln1*

Figure 6.1 A sample directory file content

Inode numbers and file system IDs are defined in POSIX.1, but the uses of these attributes are implementation-dependent. Inode tables are kept in their file systems on disk, but the UNIX kernel maintains an in-memory inode table to contain a copy of the recently accessed inode records.

6.5 Application Program Interface to Files

Both UNIX and POSIX systems provide an application interface similar to files, as follows:

- Files are identified by path names
- Files must be created before they can be used. The UNIX commands and corresponding system calls to create various types of files are:

File type	UNIX command	UNIX and POSIX.1 system call
Regular files	vi, ex, etc.	open, creat
Directory files	mkdir	mkdir, mknod
FIFO files	mkfifo	mkfifo, mknod
Device files	mknod	mknod
Symbolic links	ln -s	symlink

- Files must be opened before they can be accessed by application programs. UNIX and POSIX.1 define the *open* API, which can be used to open any files. The *open* function returns an integer file descriptor, which is a file handle to be used in other system calls to manipulate the open file
- A process may open at most OPEN_MAX files of any types at any one time. The OPEN_MAX and its POSIX.1-defined minimum value _POSIX_OPEN_MAX are defined in the <limits.h> header
- The *read* and *write* system calls can be used to read data from and write data to opened files
- File attributes can be queried by the *stat* or *fstat* system call
- File attributes can be changed by the *chmod*, *chown*, *utime*, and *link* system calls
- File hard links can be removed by the *unlink* system call

To facilitate the query of file attributes by application programs, UNIX and POSIX.1 define a *struct stat* data type in the <sys/stat.h> header. A *struct stat* record contains all the user-visible attributes of any file being queried, and it is assigned and returned by the *stat* or *fstat* function. The POSIX.1 declaration of the *struct stat* type is:

```
struct stat
{
        dev_t     st_dev;       /* file system ID */
        ino_t     st_ino;       /* File inode number */
        mode_t    st_mode;      /* Contains file type and access flags */
        nlink_t   st_nlink;     /* Hard link count */
        uid_t     st_uid;       /* File user ID */
        gid_t     st_gid;       /* File group ID */
        dev_t     st_rdev;      /* Contains major and minor device numbers */
        off_t     st_size;      /* File size in number of bytes */
        time_t    st_atime;     /* Last access time  /
        time_t    st_mtime;     /* Last modification time */
        time_t    st_ctime;     /* Last status change time */
};
```

If the path name (or file descriptor) of a symbolic link file is passed as an argument to a *stat* (or *fstat*) system call, the function will resolve the link reference and show the attributes of the actual file to which the link refers. To query the attributes of a symbolic link file itself, one can use the *lstat* system call instead. Because symbolic link files are not yet supported by POSIX.1, the *lstat* system call is also not a POSIX.1 standard.

6.6 UNIX Kernel Support for Files

In UNIX System V.3, the kernel has a file table that keeps track of all opened files in the system. There is also an inode table that contains a copy of the file inodes most recently accessed.

When a user executes a command, a process is created by the kernel to carry out the command execution. The process has its own data structure which, among other things, is a file descriptor table. The file descriptor table has OPEN_MAX entries, and it records all files opened by the process. Whenever the process calls the *open* function to open a file for read and/or write, the kernel will resolve the path name to the file inode. If the file inode is not found or the process lacks appropriate permissions to access the inode data, the *open* call fails and returns a -1 to the process. If, however, the file inode is accessible to the process, the kernel will proceed to establish a path from an entry in the file descriptor table, through a file table, onto the inode for the file being opened. The process for that is as follows:

1. The kernel will search the process file descriptor table and look for the first unused entry. If an entry is found, that entry will be designated to reference the file. The index to the entry will be returned to the process (via the return value of the *open* function) as the file descriptor of the opened file.

2. The kernel will scan the file table in its kernel space to find an unused entry that can be assigned to reference the file.

   If an unused entry is found, the following events will occur:

   a. The process's file descriptor table entry will be set to point to this file table entry.
   b. The file table entry will be set to point to the inode table entry where the inode record of the file is stored.
   c. The file table entry will contain the current file pointer of the open file. This is an offset from the beginning of the file where the next read or write operation will occur .
   d. The file table entry will contain an open mode that specifies that the file is opened for read-only, write-only, or read and write, etc. The open mode is specified from the calling process as an argument to the *open* function call.
   e. The reference count in the file table entry is set to 1. The reference count keeps

track of how many file descriptors from any process are referencing the entry.

f. The reference count of the in-memory inode of the file is increased by 1. This count specifies how many file table entries are pointing to that inode.

If either (1) or (2) fails, the *open* function will return with a -1 failure status, and no file descriptor table or file table entry will be allocated.

Figure 6.2 shows a process's file descriptor table, the kernel file table, and the inode table after the process has opened three files: *xyz* for read-only, *abc* for read-write, and *abc* again for write-only.

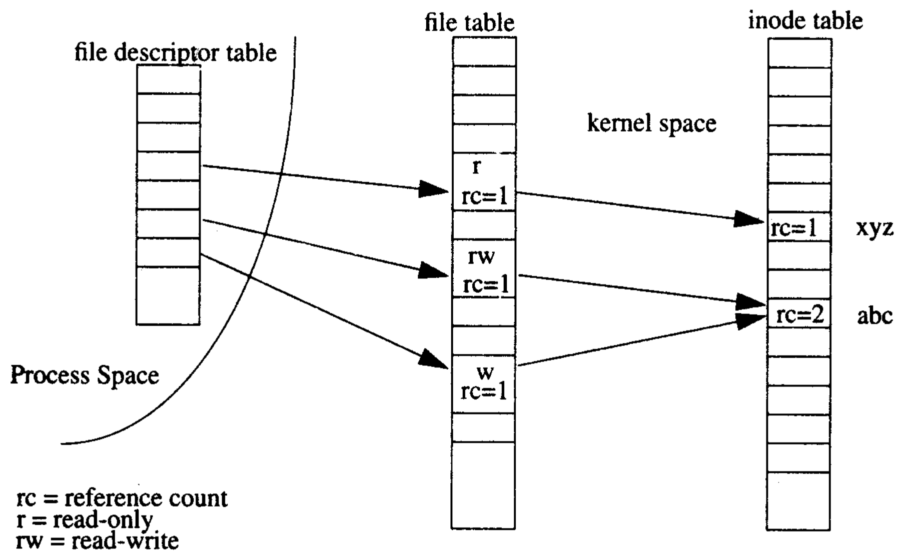


Figure 6.2 Data Structure for File Manipulation

Note that the reference count of an allocated file table entry is usually 1, but a process may use the *dup* (or *dup2*) function to make multiple file descriptor table entries point to the same file table entry. Alternatively, the process may call the *fork* function to create a child process, such that the child and parent process file table entries are pointing to corresponding file table entries at the same time. All these will cause a file table entry reference count to be larger than 1. The *dup, dup2*, and *fork* functions and their uses will be explained in more detail in Chapter 8.

The reference count in a file inode record specifies how many file table entries are pointing to the file inode record. If the count is not zero, it means that one or more processes are currently opening the file for access.

Once an *open* call succeeds, the process can use the returned file descriptor for future reference. Specifically, when the process attempts to read (or write) data from the file, it will use the file descriptor as the first argument to the *read* (or *write*) system call. The kernel will use the file descriptor to index the process's file descriptor table to find the file table entry of the opened file. It then checks the file table entry data to make sure that the file is opened with the appropriate mode to allow the requested read (or write) operation.

If the read (or write) operation is found compatible with the file's open mode, the kernel will use the pointer specified in the file table entry to access the file's inode record (as stored in the inode table). Furthermore, it will use the file pointer stored in the file table entry to determine where the read (or write) operation should occur in the file. Finally, the kernel checks the file's file type in the inode record and invokes an appropriate driver function to initiate the actual data transfer with a physical device.

If a process calls the *lseek* system call to change the file pointer to a different offset for the next read (or write) operation, the kernel will use the file descriptor to index the process file descriptor table to find the pointer to the file table entry. The kernel then accesses the file table entry to get the pointer to the file's inode record. It then checks that the file is not a character device file, a FIFO file, or a symbolic link file, as these files allow only sequential read and write operations. If the file type is compatible with *lseek*, the kernel will change the file pointer in the file table entry according to the value specified in the *lseek* arguments.

When a process calls the *close* function to close an opened file, the sequence of events are as follows:

1. The kernel sets the corresponding file descriptor table entry to be unused.
2. It decrements the reference count in the corresponding file table entry by 1. If the reference count is still non-zero, go to 6.
3. The file table entry is marked as unused.
4. The reference count in the corresponding file inode table entry is decrement by one. If the count is still nonzero, go to 6.
5. If the hard-link count of the inode is not zero, it returns to the caller with a success status. Otherwise it marks the inode table entry as unused and deallocates all the physical disk storage of the file, as all the file path names have been removed by some process.
6. It returns to the process with a 0 (success) status.

6.7 Relationship of C Stream Pointers and File Descriptors

C stream pointers (FILE*) are allocated via the *fopen* C function call. A stream pointer is more efficient to use for applications doing extensive sequential read from or write to files, as the C library functions perform I/O buffering with streams. On the other hand, a file descriptor, allocated by an *open* system call, is more efficient for applications that do frequent random access of file data, and I/O buffering is not desired. Another difference between the two is stream pointers is supported on all operating systems, such as VMS, CMS, DOS, and UNIX, that provide C compilers. File descriptors are used only in UNIX and POSIX.1-compliant systems; thus, programs that use stream pointers are more portable than are those using file descriptors.

To support stream pointers, each UNIX process has a fixed-size stream table with OPEN_MAX entries. Each entry is of type FILE and contains working data from an open file. Data stored in a FILE record includes a buffer for I/O data buffering, the file I/O error status, and an end-of-file flag, etc. When *fopen* is called, it scans the calling process FILE table to find an unused entry, then assigns this entry to reference the file and returns the address of this entry (FILE*) as the stream pointer for the file. Furthermore, in UNIX, the *fopen* function calls the *open* function to perform the actual file opening, and a FILE record contains a file descriptor for the open file. One can extract the file descriptor associated with a stream pointer via the *fileno* macro, which is declared in the <stdio.h> header:

```
int fileno ( FILE* stream_pointer );
```

Thus, if a process calls *fopen* to open a file for access, there will be an entry in the process FILE table and an entry in the process's file descriptor table being used to reference the file. If the process calls *open* to open the file, only an entry in the process's file descriptor table is assigned to reference the file. However, one can convert a file descriptor to a stream pointer via the *fdopen* C library function:

```
FILE* fdopen ( int file_descriptor, char * open_mode );
```

The *fdopen* function has an action similar to the *fopen* function, namely, it assign a process FILE table entry to reference the file, records the file descriptor value in the entry, and returns the address of the entry to the caller.

After either the *fileno* or *fdopen* call, the process may reference the file via either the stream pointer or the file descriptor. Other C library functions for files also rely on the operating system APIs to perform the actual functions. The following lists some C library functions and the underlying UNIX APIs they use to perform their functions:

C Library function	UNIX system call used
fopen	open
fread, fgetc, fscanf, fgets	read
fwrite, fputc, fprintf, fputs	write
fseek, ftell, frewind	lseek
fclose	close

6.8 Directory Files

A directory is a record-oriented file. Each record contains the information of a file residing in that directory The record data type is *struct dirent* in UNIX System V and POSIX.1, and *struct direct* in BSD UNIX. The record content is implementation-dependent, but in UNIX and POSIX systems they all contain two essential member fields: a file name and an inode number. The usage of directory files is to map file names to their corresponding inode numbers so that an operating system can resolve any file path name to locate its inode record.

Although an application can use the *open, read, write, lseek,* and *close* system calls to manipulate directory files, UNIX and POSIX.1 define a set of portable functions to open, browse, and close directory files. They are built on top of the *open, read, write,* and *close* system calls and are defined in <dirent.h> for UNIX System V and POSIX.1-compliant systems or in <sys/dir.h> for BSD UNIX:

Directory function	Purpose
opendir	Opens a directory file
readdir	Reads the next record from file
closedir	Closes a directory file
rewinddir	Sets file pointer to beginning of file

The *opendir* function returns a handle of type DIR*. It is analogous to the FILE* handle for a C stream file. The handle is used in the *readdir, rewinddir,* and *closedir* function calls to specify which directory file to manipulate.

Besides the above functions, UNIX systems also define the *telldir* and *seekdir* functions for random access of different records in a directory file. These functions are not POSIX.1 standard, and they are analogous to the *ftell* and *fseek* C library functions, respectively.

If a process adds or deletes a file in a directory file while another process has opened the file via the *opendir*, it is implementation-dependent as to whether the latter process will see the new changes via the *readdir* function. However, if the latter process does a *rewinddir* and then reads the directory via the *readdir*, according to POSIX.1, it should read the latest content of the directory file.

6.9 Hard and Symbolic Links

A hard link is a UNIX path name for a file. Most UNIX files have only one hard link. However, users may create additional hard links for files via the *ln* command. For example. the following command creates a new link call */usr/joe/book.new* for the file */usr/mary/fun.doc*:

```
ln    /usr/mary/fun.doc        /usr/joe/book.new
```

After the above command, users may refer to the same file by either */usr/joe/book.new* or */usr/mary/fun.doc*.

Symbolic links can be created in the same manner as hard links, except that you must specify the *-s* option to the *ln* command. Thus, using the above example, you can create */usr/joe/book.new* as a symbolic link instead of a hard link with the following command:

```
ln    -s  /usr/mary/fun.doc        /usr/joe/book.new
```

Symbolic links or hard links are used to provide alternative means of referencing files. For example, you are at the */usr/jose/proj/doc* directory, and you are constantly browsing the file */usr/include/sys/unistd.h*. Thus, rather than specifying the full path name */usr/include/sys/unistd.h* every time you reference it, you could define a link to that file as follows:

```
ln    /usr/include/sys/unistd.h    uniref
```

From now on, you can refer to that file as *uniref*. Thus links facilitate users in referencing files.

ln differs from the *cp* command in that *cp* creates a duplicated copy of a file to another file with a different path name, whereas *ln* primarily creates a new directory entry to reference a file. For example, given the following command:

```
ln    /usr/jose/abc    /usr/mary/xyz
```

the directory files */usr/jose* and */usr/mary* will contain:

inode number	*filename*
115	.
89	..
201	***abc***
346	*a.out*

/usr/jose

inode number	*filename*
515.	.
989	..
201	***xyz***
146	*fun.c*

/usr/mary

Note that both the */usr/jose/abc* and */usr/mary/xyz* refer to the same inode number, 201. Thus there is no new file created. If, however, we use the *ln -s* or the *cp* command to create the */usr/mary/xyz* file, a new inode will be created, and the directory files of */usr/jose* and */ usr/mary* will look like the following:

inode number	*file name*
115	.
89	..
201	***abc***
346	*a.out*

/usr/jose

inode number	*file name*
515	.
989	..
345	***xyz***
146	*fun.c*

/usr/mary

If the */usr/mary/xyz* file was created by the *cp* command, its data content will be identical to that of */usr/jose/abc*, and the two files will be separate objects in the file system. However, if the */usr/mary/xyz* file was created by the *ln -s* command, then the file data will consist only of the path name */usr/mary/abc*.

Thus, *ln* helps save disk space over *cp* by not creating duplicated copies of files. Moreover, whenever a user makes changes to a link (hard or symbolic) of a file, the changes are visible from all the other links of the file. This is not true for files created by *cp*, as the duplicated file is a separate object from the original.

Hard links are used in all versions of UNIX. The limitations of hard links are:

* Users cannot create hard links for directories, unless they have superuser (root) privileges. This is to prevent users from creating cyclic links in a file system. An example of a cyclic link is like the following command:

```
ln    /usr/jose/text/unix_link    /usr/jose
```

If this command succeeds, then whenever a user does a *ls -R /usr/jose*, the *ls* command will run into an infinite loop in displaying, recursively, the subdirectory tree of */usr/jose*. UNIX allows the superuser to create hard links on directories with the assumption that a supervisor will not make this kind of mistake

Users cannot create hard links on a file system that references files on a different system. This is because a hard link is just a directory entry (in a directory file that stores the new link) to reference the same inode number as the original link, but inode numbers are unique only within a file system (hard links cannot be used to reference files on remote file systems)

To overcome the above limitations, BSD UNIX invented the symbolic link concept. A symbolic link can reference a file on any file system because its data is a path name, and an operating system kernel can resolve path names to locate files in either local or remote file systems. Furthermore, users are allowed to create symbolic links to directories, as the kernel can detect cyclic directories caused by symbolic links. Thus, there will be no infinite loops in directory traversal. Symbolic links are supported in the UNIX System V.4, but not by POSIX.1.

The following table summarizes the differences between symbolic and hard links:

Hard Link	Symbolic Link
Does not create a new inode	Create a new inode
Can not link directories, unless this is done by root	Can link directories
Can not link files across file systems	Can link files across file systems
Increase the hard link count of the linked inode	Does not change the hard link count of the linked inode

6.10 Summary

This chapter describes the UNIX and POSIX file systems and the different file types in the systems. It also depicts how these various files are created and used. Furthermore, the UNIX System V system-wide and per-process data structures that are used to support file manipulation and the application program interfaces for files are covered. The objective of this chapter is to familiarize readers with the UNIX file structures so that they can understand why the UNIX and POSIX system calls were created, how they work, and what their applications for users are.

The next chapter will describe the UNIX and POSIX file APIs in more detail.

CHAPTER 7

UNIX File APIs

This chapter describes how the UNIX and POSIX applications interface with files. After reading this chapter, readers should be able to write programs that perform the following functions on any type of files in a UNIX or POSIX system:

- Create files
- Open files
- Transfer data to and from files
- Close files
- Remove files
- Query file attributes
- Change file attributes
- Truncate files

To illustrate the application of UNIX and POSIX.1 APIs for files, some C++ programs are depicted to show the implementation of the UNIX commands *ls, mv, chmod, chown*, and *touch* based on these APIs. Furthermore, this chapter defines a C++ class called *file*. This *file* class inherits all the properties of the C++ *fstream* class, and it has additional member functions to create objects of any file type, as well as to display and change file object attributes.

Readers are assumed to have read the last chapter to become familiar with the UNIX and POSIX file structures, as they are fundamental to a sound understanding of the use and application of the file APIs described in this chapter.

General File APIs

s explained in the last chapter, files in a UNIX or POSIX system may be one of the ing types:

- Regular file
- Directory file
- FIFO file
- Character device file
- Block device file
- Symbolic link file

There are special APIs to create these different types of files. These APIs will be described in later sections. However, there is a set of generic APIs that can be used to manipulate more than one type of files. These APIs are:

API	Use
open	Opens a file for data access
read	Reads data from a file
write	Writes data to a file
lseek	Allows random access of data in a file
close	Terminates the connection to a file
stat, fstat	Queries attributes of a file
chmod	Changes access permission of a file
chown	Changes UID and/or GID of a file
utime	Changes last modification and access time stamps of a file
link	Creates a hard link to a file
unlink	Deletes a hard link of a file
umask	Sets default file creation mask

These general APIs are explained as follows.

7.1.1 open

The *open* function establishes a connection between a process (a process is an application program under execution) and a file. If can be used to create brand new files. Furthermore, after a file is created any process can call the *open* function to get a file descriptor to refer to the file. The file descriptor is used in the *read* and *write* system calls to access the file content.

The prototype of the *open* function is:

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

int open ( const char *path_name, int access_mode, mode_t permission );
```

The first argument *path_name* is the path name of a file. This may be an absolute path name (a character string begins with the "/" character) or a relative path name (a character string does not begin with the "/" character). If a given *path_name* is a symbolic link, the function will resolve the link reference (and recursively, if the symbolic link refers to another symbolic link) to a nonsymbolic link file to which the link refers.

The second argument *access_mode* is an integer value that specifies how the file is to be accessed by the calling process. The *access_mode* value should be one of the following manifested constants as defined in the <fcntl.h> header:

Access mode flag	**Use**
O_RDONLY	Opens the file for read-only
O_WRONLY	Opens the file for write-only
O_RDWR	Opens the file for read and write

Furthermore, one or more of the following modifier flags can be specified by bitwise-ORing them with one of the above access mode flags to alter the access mechanism of the file:

Access modifier flag	**Use**
O_APPEND	Appends data to the end of the file
O_CREAT	Creates the file if it does not exist
O_EXCL	Used with the O_CREAT flag only. This flag causes *open* to fail if the named file already exists
O_TRUNC	If the file exists, discards the file content and sets the file size to zero bytes
O_NONBLOCK	Specifies that any subsequent read or write on the file should be nonblocking
O_NOCTTY	Specifies not to use the named terminal device file as the calling process control terminal

To illustrate the use of the above flags, the following example statement opens a file called */usr/xyz/textbook* for read and write in append mode:

```
int fdesc = open ( "/usr/xyz/textbook",O_RDWR|O_APPEND, 0 );
```

If a file is to be opened for read-only, the file should already exist and no other modifier flags can be used.

If a file is opened for write-only or read-write, any modifier flags can be specified. However, O_APPEND, O_TRUNC, O_CREAT, and O_EXCL are applicable to regular files only, whereas O_NONBLOCK is for FIFO and device files only, and O_NOCTTY is for terminal device files only.

The O_APPEND flag specifies that data written to a named file will be appended at the end of the file. If this is not specified, data can be written to anywhere in the file.

The O_TRUNC flag specifies that if a named file already exists, the *open* function should discard its content. If this is not specified, current data in the file will not be altered by the *open* function.

The O_CREAT flag specifies that if a named file does not exist, the *open* function should create it. If a named file does exist, the O_CREAT flag has no effect on the *open* function. However, if the named file does not exist and the O_CREAT file is not specified, *open* will abort with a failure return status. The O_EXCL flag, if used, must be accompanied by the O_CREAT flag. When both the O_CREAT and O_EXCL flags are specified, the *open* function will fail if the named file exists. Thus, the O_EXCL flag is used to ensure that the *open* call creates a new file.

The O_NONBLOCK flag specifies that if the *open* and any subsequent *read* or *write* function calls on a named file will block a calling process, the kernel should abort the functions immediately and return to the process with a proper status value. For example, a process is normally blocked on reading an empty pipe (*pipe* is described in Section 7.5) or on writing to a pipe that is full. The O_NONBLOCK flag may be used to specify that such read and write operations are nonblocking. In System V.3, O_NDELAY is defined instead of O_NONBLOCK; the two flags have similar uses, but their behaviors are not identical. These are covered in more detail in Section 7.5.

The O_NOCTTY flag is defined in POSIX.1. It specifies that if a process has no controlling terminal and it opens a terminal device file, that terminal will not be the controlling terminal of the process. If this flag is not set, it is implementation-dependent as to whether the terminal will become the process controlling terminal. Note that in UNIX System V.3 where the O_NOCTTY is undefined, if a process has no controlling terminal, the *open* call will automatically establish the first terminal device file opened as the controlling terminal.

The *permission* argument is required only if the O_CREAT flag is set in the *access_mode* argument. It specifies the access permission of the file for its owner, group member, and all other people. Its data type is *int* in UNIX System V (V.3 and earlier), and its value is usually specified as an octal integer literal, such as 0764. Specifically, the left-most, middle, and right-most digits of an octal value specify the access permission for owner, group, and others, respectively. Furthermore, in each octal digit, the left-most, middle, and right-most bits specify the read, write, and execute permission, respectively. The value of each bit is either 1, which means a right is granted, or zero, for no such right. Thus, the 0764 value manes that the new file's owner has read-write-execute permission, group members have read-write permission, and others have read-only permission.

POSIX.1 defines the *permission* data type as *mode_t*, and its value should be constructed based on the manifested constants defined in the <sys/stat.h> header. These manifested constants are aliases to the octal integer values used in UNIX System V. For example, the 0764 *permission* value should be specified as:

```
S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH
```

Actually, a *permission* value specified in an *open* call is modified by its calling process *umask* value. An umask value specifies some access rights to be masked off (or taken away) automatically on any files created by the process. A process umask is inherited from its parent process, and its value can be queried or changed by the *umask* system call. The function prototype of the *umask* API is:

```
mode_t      umask ( mode_t      new_umask );
```

The *umask* function takes a new umask value as an argument. This new umask value will be used by the calling process from then on, and the function returns the old umask value. For example, the following statement assigns the current umask value to the variable *old_mask*, and sets the new umask value to "no execute for group" and "no write-execute for others":

```
mode_t    old_mask = umask ( S_IXGRP | S_IWOTH | S_IXOTH );
```

The *open* function takes its *permission* argument value and bitwise-ANDs it with the one's complement of the calling process umask value,. Thus, the final access permission to be assigned to any new file created is:

```
actual_permission = permission & ~umask_value
```

Thus bits which are set in an umask value mean that the corresponding access rights are to be taken off of any newly created files. For example, if *open* is called in System V.3 to create a file called */usr/mary/show_ex* with a permission of 0557, and the umask of the calling process is 031, then the actual access permission assigned to the newly created file is:

```
actual_permission = 0557 & (~031) = 0546
```

The return value of the *open* function is -1 if the API fails and *errno* contains an error status value. If the API succeeds, the return value is a file descriptor that can be used to reference the file in other system calls. The file descriptor value should be between 0 and OPEN_MAX-1, inclusively.

7.1.2 creat

The *creat* system call is used to create new regular files. Its prototype is:

```
#include <sys/types.h>
#include <unistd.h>
int creat ( const char* path_name, mode_t mode );
```

The *path_name* argument is the path name of a file to be created. The *mode* argument is the same as that for the *open* API. However, since the O_CREAT flag was added, the *open* API can be used to both create and open regular files. Thus, the *creat* API has become obsolete. It is retained for backward-compatibility with early versions of UNIX. The *creat* function can be implemented using the *open* function as:

```
#define creat(path_name,mode)
        open (path_name, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

7.1.3 read

The *read* function fetches a fixed size block of data from a file referenced by a given file descriptor. The function prototype of the *read* function is:

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

ssize_t read ( int fdesc, void* buf, size_t size );
```

The first argument, *fdesc*, is an integer file descriptor that refers to an opened file. The second argument, *buf*, is the address of a buffer holding any data read. The third argument, *size*, specifies how many bytes of data are to be read from the file. The *size_t* data type is defined in the <sys/types.h> header and should be the same as *unsigned int*.

Note that *read* can read text or binary files. This is why the data type of *buf* is a universal pointer (*void**). For example, the following code fragment reads, sequentially, one or more record of *struct sample*-typed data from a file called *dbase*:

```
struct sample { int x; double y; char* z; } varX;
int fd = open ("dbase", O_RDONLY);
while (read(fd,&varX,sizeof(varX)) > 0)
        /* process data stored in varX*/
```

The return value of *read* is the number of bytes of data successfully read and stored in the *buf* argument. It should normally be equal to the *size* value. However, if a file contains less than *size* bytes of data remaining to be read, the return value of *read* will be less than that of *size*. Furthermore, if end-of-file is reached, *read* will return a zero value.

Because *ssize_t* is usually defined as *int* in the <sys/types.h> header, users should not set *size* to exceed INT_MAX in any *read* function call. This ensures that the function return value can reflect the actual number of bytes read.

If a *read* function call is interrupted by a caught signal (signals are explained in Chapter 9) and the operating system does not restart the system call automatically, POSIX.1 allows two possible behaviors of the *read* function. The first one is the same as in UNIX System V.3, where the *read* function will return a -1 value, *errno* will be set to EINTR, and all the data read in the call will be discarded (hence, the process cannot recover the data). The second behavior is mandated by the POSIX.1 FIPS standard, which specifies that the *read* function will return the number of bytes of data read prior to the signal interruption. This allows a process to continue reading the file.

In BSD UNIX where the kernel automatically restarts any system call after a signal interruption, the return value of *read* will be the same as that in a normal execution. In UNIX System V.4, the user can specify, on a per-signal basis, whether the kernel will restart any system call that is interrupted by each signal. Thus, the *read* function behavior may be similar to that of BSD UNIX for restartable signals, or to that of either UNIX System V.3 or POSIX.1 FIPS systems for nonrestartable signals.

The *read* function may block a calling process execution if it is reading a FIFO or a device file and data is not yet available to satisfy the read request. Users may specify the O_NONBLOCK or O_NDELAY flags on a file descriptor to request nonblocking read operations on the corresponding file. The behavior of the *read* function on these special files will be described in detailed in the FIFO and device file API sections.

7.1.4 write

The *write* function puts a fixed size block of data to a file referenced by a given file descriptor. Its operation is opposite to that of the *read* function. Its prototype is:

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

ssize_t write ( int fdesc, const void* buf, size_t size );
```

The first argument, *fdesc*, is an integer file descriptor that refers to an opened file. The second argument, *buf*, is the address of a buffer which contains data to be written to the file. The third argument, *size*, specifies how many bytes of data are in the *buf* argument.

Like the *read* API, *write* can write text or binary files. This is why the data type of *buf* is a universal pointer (*void**). For example, the following code fragment writes ten records of *struct sample*-typed data to a file called *dbase2*:

```
struct sample { int x; double y; char* z; } varX[10 ];
int fd = open ( "dbase2", O_WRONLY );

/* initialize varX array here... */

write( fd, (void*)varX, sizeof varX );
```

The return value of *write* is the number of bytes of data successfully written to a file. It should normally be equal to the *size* value. However, if the write will cause the file size to exceed a system imposed limit or if the file system disk is full, the return value of *write* will be the actual number of bytes written before the function was aborted.

The handling of signal interruption by the *write* function is the same as that for the *read* function: If a signal arrives during a *write* function call and the operating system does not restart the system call automatically, the *write* function may either return a -1 value and set *errno* to EINTR (the System V method) or return the number of bytes of data written prior to the signal interruption. The latter behavior is mandated by the POSIX.1 FIPS standard.

Like the *read* function, in UNIX System V.4 a user can specify, on a per-signal basis, whether the kernel will restart any system call that is interrupted by each signal. Thus, the *write* function behavior may be similar to that of BSD UNIX for restartable signals (and the *write* function return value is the same as that in a normal execution) or to that of either UNIX System V.3 or POSIX.1 FIPS systems for nonrestartable signals.

Finally, the *write* function may perform nonblocking operation if the O_NONBLOCK or O_NDELAY flags are set on the *fdesc* argument to the function. This is the same for the *read* function.

7.1.5 close

The *close* function disconnects a file from a process. The function prototype of the *close* function is:

```
#include <unistd.h>
int close ( int fdesc );
```

The argument *fdesc* is an integer file descriptor that refers to an opened file. The return value of *close* is zero if the call succeeds, or -1 if it fails, and *errno* contains an error code.

The *close* function frees unused file descriptors so that they can be reused to reference other files. This is important, as a process may open up to OPEN_MAX files at any one time, and the *close* function allows a process to reuse file descriptors to access more than OPEN_MAX files in the course of its execution.

Furthermore, the *close* function will deallocate system resources (e.g., file table entries and memory buffer allocated to hold read/write file data) that are dedicated to support the operation of file descriptors. This reduces the memory requirement of a process.

If a process terminates without closing all the files it has opened, the kernel will close those files for the process.

The *iostream* class defines a *close* member function to close a file associated with an iostream object. This member function may be implemented using the *close* API as follows:

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

int iostream::close() {return close(this->fileno());}
```

7.1.6 fcntl

The *fcntl* function helps a user to query or set access control flags and the *close-on-exec* flag of any file descriptor. Users can also use *fcntl* to assign multiple file descriptors to reference the same file. The prototype of the *fcntl* function is:

```
#include <fcntl.h>
int fcntl ( int fdesc, int cmd, ... );
```

The *cmd* argument specifies which operations to perform on a file referenced by the *fdesc* argument. A third argument value, which may be specified after *cmd,* is dependent on the actual *cmd* value. The possible *cmd* values are defined in the <fcntl.h> header. These values and their uses are:

***cmd* value**	**Use**
F_GETFL	Returns the access control flags of a file descriptor *fdesc*
F_SETFL	Sets or clears access control flags that are specified in the third argument to *fcntl.* The allowed access control flags are O_APPEND and O_NONBLOCK (or O_NDELAY in non-POSIX-compliant UNIX)
F_GETFD	Returns the *close-on-exec* flag of a file referenced by *fdesc.* If a return value is zero, the flag is off; otherwise, the return value is nonzero and the flag is on. The *close-on-exec* flag of a newly opened file is off by default
F_SETFD	Sets or clears the *close-on-exec* flag of a file descriptor *fdesc.* The third argument to *fcntl* is an integer value, which is 0 to clear the flag, or 1 to set the flag
F_DUPFD	Duplicates the file descriptor *fdesc* with another file descriptor. The third argument to *fcntl* is an integer value which specifies that the duplicated file descriptor must be greater than or equal to that value. The return value of *fcntl,* in this case, is the duplicated file descriptor

The *fcntl* function is useful in changing the access control flag of a file descriptor. For example, after a file is opened for blocking read-write access and the process needs to change the access to nonblocking and in write-append mode, it can call *fcntl* on the file's descriptor as:

```
int cur_flags = fcntl(fdesc, F_GETFL);
int rc = fcntl (fdesc, F_SETFL, cur_flag | O_APPEND | O_NONBLOCK);
```

The *close-on-exec* flag of a file descriptor specifies that if the process that owns the descriptor calls the *exec* API to execute a different program, the file descriptor should be closed by the kernel before the new program runs (if the flag is on) or not (if the flag is off). The *exec* API and the *close-on-exec* flag are explained in more detail in Chapter 8. The following example reports the *close-on-exec* flag of a file descriptor *fdesc*, sets it to on afterward:

```
cout << fdesc << " close-on-exec: " << fcntl( fdesc, F_GETFD ) << endl;
(void)fcntl( fdesc, F_SETFD, 1);              // turn-on close-on-exec flag
```

The *fcntl* function can also be used to duplicate a file descriptor *fdesc* with another file descriptor. The results are two file descriptors referencing the same file with the same access mode (read and/or write, blocking or nonblocking access, etc.) and sharing the same file pointer to read or write the file. This is useful in the redirection of the standard input or output to reference a file instead. For example, the following statements change the standard input of a process to a file called *FOO*:

```
int fdesc = open("FOO", O_RDONLY);                    // open FOO for read
close(0);                                             // close standard input
if (fcntl(fdesc,F_DUPFD, 0)==-1) perror("fcntl");     // stdin from FOO now
char buf[256];
int rc = read(0,buf,256);                             // read data from FOO
```

The *dup* and *dup2* functions in UNIX perform the same file duplication function as *fcntl*. They can be implemented using *fcntl* as:

```
#define   dup(fdesc)          fcntl(fdesc,F_DUPFD,0)
#define   dup2(fdesc1, fd2)   close(fd2), fcntl(fdesc,F_DUPFD,fd2)
```

The *dup* function duplicates a file descriptor *fdesc* with the lowest unused file descriptor of a calling process. The *dup2* function will duplicate a file descriptor *fdesc* using a *fd2* file descriptor, regardless of whether *fd2* is used to reference another file.

File duplication and standard input or output redirection are described in more detail in the next chapter.

The return value of *fcntl* is dependent on the *cmd* value, but it is -1 if the function fails. Possible failures may be due to the specification of an invalid *fdesc* or *cmd*.

7.1.7 lseek

The *read* and *write* system calls are always relative to the current offset within a file. The *lseek* system call can be used to change the file offset to a different value. Thus, *lseek* allows a process to perform random access of data on any opened file. *Lseek* is incompatible with FIFO files, character device files, and symbolic link files.

The prototype of the *lseek* function is:

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek ( int fdesc, off_t pos, int whence );
```

The first argument, *fdesc,* is an integer file descriptor that refers to an opened file. The second argument, *pos,* specifies a byte offset to be added to a reference location in deriving the new file offset value. The reference location is specified by the *whence* argument. The possible values of *whence* and the corresponding file reference locations are:

***whence* value**	**Reference location**
SEEK_CUR	Current file pointer address
SEEK_SET	The beginning of a file
SEEK_END	The end of a file

The SEEK_CUR, SEEK_SET, and SEEK_END are defined in the <unistd.h> header. Note that it is illegal to specify a negative *pos* value with the *whence* value set to SEEK_SET, as this will cause the function to assign a negative file offset. Furthermore, if an *lseek* call will result in a new file offset that is beyond the current end-of-file, two outcomes possible are: If a file is opened for read-only, *lseek* will fail; if, however, a file is opened for write access, *lseek* will succeed, and it will extend the file size up to the new file offset address. Furthermore, the data between the end-of-file and the new file offset address will be initialized with NULL characters.

The return value of *lseek* is the new file offset address where the next read or write operation will occur, or -1 if the *lseek* call fails.

The *iostream* class defines the *tellg* and *seekg* functions to allow users to do random data access of any iostream object. These functions may be implemented using the *lseek* API as follows:

```
#include <iostream.h>
#include <sys/types.h>
```

```
#include <unistd.h>

streampos iostream::tellg()
{
     return (streampos)lseek( this->fileno(), (off_t)0, SEEK_CUR );
}

iostream&iostream::seekg( streampos pos, seek_dir ref_loc )
{
     if (ref_loc == ios::beg )
        (void)lseek( this->fileno(), (off_t)pos, SEEK_SET);
     else if ( ref_loc == ios::cur )
        (void)lseek( this->fileno(), (off_t)pos, SEEK_CUR );
     else if ( ref_loc == ios::end )
        (void)lseek( this->fileno(), (off_t)pos, SEEK_END );
     return *this;
}
```

The *iostream::tellg* simply calls *lseek* to return the current file pointer associated with an iostream object. The file descriptor of an iostream object is obtained from the *fileno* member function. Note that *streampos* and *off_t* are the same as the *long* data type.

The *iostream::seekg* also relies on *lseek* to alter the file pointer associated with an iostream object. The arguments to *iostream::seekg* are a file offset and a reference location for the offset. There is a one-to-one mapping of the *seek_dir* values to the *whence* values used by *lseek*:

***seek_dir* value**	**lseek whence value**
ios::beg	SEEK_SET
ios::cur	SEEK_CUR
ios::end	SEEK_END

Thus, the *iostream::seekg* function simply converts a *seek_dir* value to an *lseek whence* value and calls *lseek* to change an iostream object file pointer according to the *pos* value. The file descriptor of an iostream object is obtained from the *fileno* member function.

7.1.8 link

The *link* function creates a new link for an existing file. This function does not create a new file. Rather, it creates a new path name for an existing file.

The prototype of the *link* function is:

```
#include <unistd.h>
int link ( const char* cur_link, const char* new_link );
```

The first argument, *cur_link,* is a path name of an existing file. The second argument, *new_link,* is a new path name to be assigned to the same file. If this call succeeds, the hard link count attribute of the file will be increased by 1.

In UNIX, *link* cannot be used to create hard links across file systems. Furthermore, *link* cannot be used on directory files unless it is called by a process that has superuser privileges.

The UNIX *ln* command is implemented using the *link* API. A simple version of the *ln* program, that does not support the *-s* (for creating a symbolic link) option, is as follows:

```
/* test_ln.C */
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>

int main ( int argc, char* argv[] )
{
   if ( argc!=3 ) {
      cerr << "usage: " << argv[0] << " <src_file> <dest_file>\n";
      return 0;
   }
   if ( link ( argv[1], argv[2 ] ) == -1 )  {
      perror( "link" );
      return 1;
   }
   return 0;
}
```

7.1.9 unlink

The *unlink* function deletes a link of an existing file. This function decreases the hard link count attributes of the named file, and removes the file name entry of the link from a directory file. If this function succeeds, the file can no longer be reference by that link. A file is removed from the file system when its hard link count is zero and no process has any file descriptor referencing that file.

The prototype of the *unlink* function is:

```
#include <unistd.h>
int unlink ( const char* cur_link );
```

The argument *cur_link* is a path name that references an existing file. The return value is 0 if the call succeeds or -1 if it fails. Some possible causes of failure may be that the *cur_link* is invalid (no file exists with that name), the calling process lacks access permission to remove that path name, or the function is interrupted by a signal.

In UNIX, *unlink* cannot be used to remove a directory file unless the calling process has the superuser privilege.

ANSI C defines the *remove* function which does the similar unlink operation. Furthermore, if the argument to the *remove* function is an empty directory, it will remove that directory. The prototype of the *rename* function is:

```
#include <stdio.h>
int rename ( const char* old_path_name, const char* new_path_name );
```

Both the *link* and the *rename* functions fail when the new link to be created is in a different file system (or disk partition) than the original file.

The UNIX *mv* command can be implemented using the *link* and *unlink* APIs. A simple version of the *mv* program is as follows:

```
#include <iostream.h>
#include <unistd.h>
#include <string.h>

int main ( int argc, char* argv[] )
{
    if (argc!=3 || !strcmp( argv[1], argv[2]) )
        cerr << "usage: " << argv[0] << " <old_link> <new_link>\n";
    else if (link (argv[1], argv[2])==0)
        return unlink(argv[1]);
    return -1;
}
```

The above program takes two command line arguments: *old_link* and *new_link*. It first checks that *old_link* and *new_link* are different path names; otherwise, the program will simply exit, as there is nothing to change. The program calls *link* to set up *new_link* as a new reference to *old_link*. If *link* fails, the program will return -1 as an error status; otherwise, it will call *unlink* to remove the *old_link*, and its return value is that of the *unlink* call.

7.1.10 stat, fstat

The *stat* and *fstat* functions retrieve the file attributes of a given file. The difference between the two functions is that the first argument of *stat* is a file path name, whereas the first argument of *fstat* is a file descriptor. The prototypes of the *stat* and *fstat* functions are:

```
#include <sys/stat.h>
#include <unistd.h>

int stat (const char* path_name, struct stat* statv);
int fstat (const int fdesc, struct stat* statv);
```

The second argument to *stat* and *fstat* is the address of a *struct stat*-typed variable. The *struct stat* data type is defined in the <sys/stat.h> header,. Its declaration, which is similar in UNIX and POSIX.1, is as follows:

```
struct stat
{
   dev_ts      t_dev;       /* file system ID */
   ino_t       st_ino;      /* File inode number */
   mode_t      st_mode;     /* Contains file type and access flags */
   nlink_t     st_nlink;    /* Hard link count */
   uid_t       st_uid;      /* File user ID */
   gid_t       st_gid;      /* File group ID */
   dev_t       st_rdev;     /* Contains major and minor device numbers */
   off_t       st_size;     /* File size in number of bytes */
   time_t      st_atime;    /* Last access time */
   time_t      st_mtime;    /* Last modification time */
   time_t-     st_ctime;    /* Last status change time */
};
```

The return value of both functions is 0 if they succeed or -1 if they fail, and *errno* contains an error status code. Possible failures of these functions may be that a given file path name (for *stat*) or file descriptor (for *fstat*) is invalid, the calling process lacks permission to access the file, or the functions are interrupted by a signal.

If a path name argument specified to *stat* is a symbolic link file, *stat* will resolve the link(s) and access the nonsymbolic link file that is being pointed at. This is the same behavior as the *open* API. Thus, *stat* and *fstat* cannot be used to obtain attributes of symbolic link files themselves. To remedy this problem, BSD UNIX invented the *lstat* API. The *lstat* function prototype is the same as that of *stat*:

```
int      lstat ( const char* path_name, struct stat* statv );
```

Lstat behaves just like *stat* for nonsymbolic link files. However, if a *path_name* argument to *lstat* is a symbolic link file, *lstat* will return the symbolic link file attributes, not the file it refers to. *Lstat* is also available in UNIX System V.3 and V.4, but undefined by POSIX.1.

The UNIX *ls* command is implemented based on the *stat* API. Specifically, the *-l* option of *ls* depicts the *struct stat* data of any file to be listed. The following *test_ls.C* program emulates the UNIX *ls -l* command:

```
/* test_ls.C: program to emulate the UNIX ls -l command */
#include <iostream.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>

static char xtbl[10] = "rwxrwxrwx";
#ifndef MAJOR
#define    MINOR_BITS       8
#define    MAJOR(dev)       ( (unsigend)dev >> MINOR_BITS )
#define    MINOR(dev)       ( dev & MINOR_BITS )
#endif

/* Show file type at column 1 of an output line */
static void display_file_type ( ostream& ofs, int st_mode )
{
  switch (st_mode&S_IFMT)   {
    case S_IFDIR:    ofs << 'd'; return;       /* directory file */
    case S_IFCHR:    ofs << 'c'; return;       /* character device file */
    case S_IFBLK:    ofs << 'b'; return;       /* block device file */
    case S_IFREG:    ofs << '-'; return;       /* regular file */
    case S_IFLNK:    ofs << 'l'; return;       /* symbolic link file */
    case S_IFIFO:    ofs << 'p'; return;       /* FIFO file */
  }
}
```

```
/* Show access perm for owner, group, others, and any special flags */
static void display_access_perm ( ostream& ofs, int st_mode )
{
   char amode[10];
   for (int i=0, j= (1 << 8); i < 9; i++, j>>=1)
         amode[i] = (st_mode&j) ? xtbl[i] : '-';         /* set access perm */
   if (st_mode&S_ISUID) amode[2] = (amode[2]=='x') ? 'S' : 's';
   if (st_mode&S_ISGID) amode[5] = (amode[5]=='x') ? 'G' : 'g';
   if (st_mode&S_ISVTX) amode[8] = (amode[8]=='x') ? 'T' : 't';
   ofs << amode << ' ';
}

/* List attributes of one file */
static void long_list (ostream& ofs, char* path_name)
{
   struct stat statv;
   struct group*gr_p;
   struct passwd*pw_p;

   if (lstat(path_name,&statv))   {
      cerr << "Invalid path name: " << path_name << endl;
      return;
   }

   display_file_type( ofs, statv.st_mode );
   display_access_perm( ofs, statv.st_mode );
   ofs << statv.st_nlink;                          /* display hard link count */

   gr_p = getgrgid(statv.st_gid);                  /* GID to group name */
   pw_p = getpwuid(statv.st_uid);                  /* convert UID to user name */
   ofs   << ' ' << ( pw_p->pw_name ? pw_p->pw_name : statv.st_uid )
         << ' ' << ( gr_p->gr_name ? gr_gr_name : statv.st_gid ) << ' ';

   if ((statv.st_mode&S_IFMT) == S_IFCHR ||
            ( statv.st_mode&S_IFMT)==S_IFBLK )
         ofs << MAJOR( statv.st_rdev ) << ',' << MINOR( statv.st_rdev );
   else  ofs << statv.st_size;             /* show file size or major/minor no. */

   ofs << ' ' << ctime( &statv.st_mtime ); /* print last modification time */
   ofs << ' ' << path_name << endl;        /* show file name */
}

/* Main loop to display file attributes one file at a time */
int main ( int argc, char* argv[] )
{
   if ( argc==1 )
```

```
        cerr << "usage: " << argv[0] << " <file path name> ...\n";
    else while (--argc >= 1)  long_list( cout, *++argv );
    return 0;
}
```

The above program takes one or more file path names as arguments. For each path name it calls *long_list* to display the attributes of the named file in the UNIX *ls -l* format. Specifically, each file attributes are depicted in one physical line with the following data:

- The first field (column 1) is a one-character code to depict the file type
- The second field (columns 2-4) has owner read, write, and execute access rights
- The third field (columns 5-7) has group read, write, and execute access rights
- The fourth field (columns 8-10) has other read, write, and execute access rights
- The fifth field is the file hard link count
- The sixth field is the file user name
- The seventh field is the file group name
- *The eighth field is the file size in number of bytes, or the major and minor device numbers, if this is a character or block device file
- The ninth field is the file last modification time stamp
- The tenth field is the file name

The *st_mode* variable in a *struct stat* record stores several attributes: file type, owner access rights, group access rights, other access rights, a *set-UID* flag, a *set-GID* flag, and a *sticky bit*. There are manifested constants defined in the <sys/stat.h> header to aid the extraction of these various fields, as shown in the above program.

The encoding of a file type to a single character code follows the UNIX *ls -l* convention: *d* stands for a directory file type, *c* stands for a character device file type, *b* stands for a block device file type, - stands for a regular file type, *p* stands for a FIFO file type, and *l* stands for a symbolic link file type.

The access permission for any category of people is always depicted in the read, write, and execute order, and by the *r*, *w* and *x* characters, respectively. A - in place of any *r*, *w*, or *x* character means that there is no read, write, or execute right for a category of people.

The *set-UID, set-GID,* and *sticky* flags are UNIX-specific. If the *set-UID* flag of an executable file is on, the effective user ID of any process created by executing that file will be the same as the file user ID. Thus, if a file user ID is zero (the superuser ID in UNIX), the corresponding process will have the superuser privileges. Similarly, if a file *set-GID* flag is on then the effective group ID of any process created by executing that file will be the same as the file group ID. The effective user ID and group ID of a process are used to determine the access permission of the process to any file. Specifically, the kernel first checks a process effective user ID against a file user ID. If they match, the process will be given the file owner access permission. If the process effective user ID does not match a file user ID but its effective group ID matches that of the file, then the file group access rights are applied to the process. Finally, if neither match, the "others" access permission will be used.

The *set-UID* and *set-GID* flags are useful on some UNIX programs, such as the *passwd*. Processes of these programs need to have a superuser privilege to perform their jobs (for *passwd* to alter the */etc/passwd* or */etc/shadow* file to change user password). Thus, by setting the *set-UID* flag of these programs, users who execute these programs can get their work done as if the superuser were there to help them.

The effective user ID and effective group ID are also used when a process creates a file: The file user ID will be assigned that of the process, and its group ID will be assigned in some system-dependent means: In UNIX System V.3, the file group ID will be assigned that of the process; but in BSD UNIX, the file group ID will be set to the group ID of the directory which contains that file. POSIX.1 permits both System V.3 and BSD methods of assigning file group ID. In UNIX System V.4, a new file group ID is assigned the group ID of the directory that contains it (BSD method) if the *set-GID* flag of the directory is on. Otherwise, the file is assigned the effective group ID of the process that creates it (System V.3 method).

If a *sticky* flag of an executable file is set, after a process of that program terminates, its text (instruction code) will stay resident in the computer's swap memory. Consequently, next time the program is executed, the kernel can start up the process faster. The *sticky* flag is reserved for frequently used programs, such as the UNIX shell or *vi* (the visual editor) programs. A *sticky* flag can be set or reset on files by the superuser only.

The user and group names of any file are supported in UNIX but not required by POSIX.1. The function *getpwuid* converts a user ID to a user name. Similarly, the *getgrgid* converts a group ID to a group name. These two functions are defined in the <pwd.h> and <grp.h> headers, respectively.

The file size is depicted for files, directories, and named pipes. For a device file, the *long_list* function shows the major and minor device numbers of the file. These device numbers are extracted from the *st_rdev* field of the *struct stat* record. Some UNIX systems supply macros called MAJOR and MINOR (defined in the <sys/stat.h> header) to render portable access of these two numbers. If they are not defined by the system, the sample program defines them explicitly. The MINOR_BITS is the number of least significant bits in the *st_rdev* field used to store the minor device number (most UNIX systems use 8 bits), and the rest of higher order bits in the *st_rdev* field store the major device number.

The last two fields are the last modification time stamp and the file name. These are obtained from the *st_mtime* field of *statv* and the *path_name* arguments, respectively.

A sample output of the above program is:

```
% a.out      /etc/motd  /dev/fd0   /usr/bin
-rw-r-xrwx   1     joe       unix       25    July 5, 1994     /etc/motd
crw-r--r-x   2     mary      admin      30    June 25, 1994   /dev/fd0
drwxr-xr--   1     terry     sharp      15    Oct. 16, 1993   /usr/bin
%
```

7.1.11 access

The *access* function checks the existence and/or access permission of user to a named file. The prototype of the *access* function is:

```
#include <unistd.h>

int access ( const char* path_name, int flag );
```

The *path_name* argument is the path name of a file. The *flag* argument contains one or more of the following bit-flags, which are defined in the <unistd.h> header:

Bit flag	**Use**
F_OK	Checks whether a named file exists
R_OK	Checks whether a calling process has read permission
W_OK	Checks whether a calling process has write permission
X_OK	Checks whether a calling process has execute permission

The *flag* argument value to an *access* call is composed by bitwise-ORing one or more of the above bit-flags. For example, the following statement checks whether a user has read and write permissions on a file called */usr/foo/access.doc*:

```
int    rc = access( "/usr/foo/access.doc", R_OK|W_OK );
```

If a *flag* value is *F_OK*, the function returns 0 if the *path_name* file exists and -1 otherwise.

If a *flag* value is any combination of *R_OK*, *W_OK*, and *X_OK*, the *access* function uses the calling process real user ID and real group ID to check against the file user ID and group ID. This determines the appropriate category (owner, group, or others) of access permissions in checking against the actual value of *flag*. The function returns 0 if all the requested permission is permitted, and -1 otherwise.

The following *test_access.C* program uses *access* to determine, for each command line argument, whether a named file exists. If a named file does not exist, it will be created and initialized with a character string *"Hello world."* However, if a named file does exists, the program will simply read data from the file:

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

int main ( int argc, char* argv[] )
{
   char buf[256];
   int   fdesc, len;
   while ( --argc > 0) {
      if (access( *++argv, F_OK ) ) {               // a brand new file
         fdesc = open( *argv, O_WRONLY|O_CREAT, 0744 );
         write( fdesc, "Hello world\n", 12 );
      } else   {                                    // file exists, read data
         fdesc = open( *argv, O_RDONLY );
         while ( len=read( fdesc, buf, 256 ) )
              write( 1, buf, len );
      }
      close ( fdesc );
   }  /* for eacn command line argument */
}
```

The above simple program may be used as a base for a database management program: If a new database is to be created, the program will initialize the database file with some startup bookkeeping data; whereas if a database file already exists, the program will read some startup data from the file to check for version compatibility between the program and the database file, etc.

7.1.12 chmod, fchmod

The *chmod* and *fchmod* functions change file access permissions for owner, group, and others, as well as the *set-UID, set-GID,* and *sticky* flags. A process that calls one of these functions should have the effective user ID of either the superuser or the owner of the file. The UNIX *chmod* commands is implemented based on the *chmod* API.

The prototypes of the *chmod* and *fchmod* functions are:

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

int chmod ( const char* path_name, mode_t flag );
int fchmod ( int fdesc, mode_t flag );
```

The *path_name* argument of *chmod* is the path name of a file, whereas the *fdesc* argument of *fchmod* is the file descriptor of a file. The *flag* argument contains the new access permission and any special flags to be set on the file. The *flag* value is the same as that used in the *open* API: It can be specified as an octal integer value in UNIX, or constructed from the manifested constants defined in the <sys/stat.h> header. For example, the following function turns on the *set-UID* flag, removes group write permission and others read and execute permission on a file named */usr/joe/funny.book*:

```
/* chmod.C */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

void change_mode()
{
   struct stat       statv;
   int               lag = ( S_IWGRP I S_IROTH I S_IXOTH );
   if (stat( "/usr/joe/funny.book", &statv ))
      perror("stat");
   else {
      flag = (statv.st_mode & ~flag) I S_ISUID;
      if (chmod ( "usr/joe/funny.book", flag ))
            perror("chmod");
   }
}
```

The above program first calls *stat* to get the file */usr/joe/funny.book* current access permission, then it masks off group write permission and others read and execute permission from the *statv.st_mode*. Then it sets the *set-UID* flag in the *statv.st_mode*. All the other existing flags are unmodified. The final *flag* value is passed to *chmod* to carry out the changes on the file. If either the *chmod* or *stat* call fails, the program will call *perror* to print a diagnostic message.

Note that, unlike the *open* API, the access permission specified in the *flag* argument of *chmod* is not modified by the calling process umask.

7.1.13 chown, fchown, lchown

The *chown* and *fchown* functions change the user ID and group ID of files. They differ only in their first argument which refer to a file by either a path name or a file descriptor. The UNIX *chown* and *chgrp* commands are implemented based on these APIs. The *lchown* function is similar to the *chown* function, except that when the *path_name* argument is a symbolic link file, the *lchown* function changes the ownership of the symbolic link file, whereas the *chown* function changes the ownership of the file to which the symbolic link refers.

The function prototypes of these functions are:

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

int chown ( const char* path_name, uid_t uid, gid_t gid );
int fchown ( int fdesc, uid_t uid, gid_t gid );
int lchown ( const char* path_name, uid_t uid, gid_t gid );
```

The *path_name* argument is the path name of a file. The *uid* argument specifies the new user ID to be assigned to the file. The *gid* argument specifies the new group ID to be assigned to the file. If the actual value of the *uid* or *gid* argument is -1, the corresponding ID of the file is not changed.

In BSD UNIX, only a process with superuser privilege can use these functions to change any file user ID and group ID. However, if a process effective user ID matches a file user ID and its effective group ID or one of its supplementary group IDs match the file group ID, the process can change the file group ID only.

In UNIX System V, a process whose effective user ID matches either the user ID of a file or the user ID of a superuser can change the file user ID and group ID.

POSIX1. specifies that if the _POSIX_CHOWN_RESTRICTED variable is defined with a non -1 value, *chown* should behave as in BSD UNIX. However, if the _POSIX_CHOWN_RESTRICTED variable is undefined, *chown* should behave as in UNIX System V.

If *chown* is called by a process that has no superuser privileges and it succeeds on a file, it will clear the file *set-UID* and *set-GID* flags. This is to prevent users from creating programs with ownership assigned to someone else (e.g., the superuser) and then executing those programs with the new owner's privileges.

If *chown* is called by a process with the effective UID of a superuser it is implementation-dependent as to how *chown* will treat the *set-UID* and *set-GID* flags of files it modifies. In UNIX System V.3 and V.4, those flags are kept intact.

The following *test_chown.C* program implements the UNIX *chown* program:

```
#include <iostream.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>

int main (int argc, char* argv[])
{
   if (argc < 3)  {
      cerr << "Usage: " << argv[0] << ' <usr_name> <file> ...\n";
      return 1;
   }

   struct passwd *pwd = getpwuid( argv[1] );  /* convert user name to UID */
   uid_t             UID = pwd ? pwd->pw_uid : -1;
   struct   stat     statv:

   if (UID == (uid_t)-1)
      cerr<< "Invalid user name\n";

   else for (int i=2; i < argc; i++)                /* do for each file specified */
      if (stat(argv[i],&statv)==0)  {
         if (chown( argv[i], UID, statv.st_gid ) )
            perror( "chown" );
      } else perror( "stat" );

   return 0;
}
```

This program takes at least two command line arguments: the first one is a user name to be assigned to files, and the second and any subsequent arguments are file path names. The program first converts a given user name to a user ID via the *getpwuid* function. If that succeeds, the program processes each named file as follows: it calls *stat to* get the file group ID, then calls *chown* to change the file user ID. If either the *stat* or *chown* API fails, *perror* will be called to print a diagnostic message.

7.1.14 utime

The *utime* function modifies the access and modification time stamps of a file. The prototype of the *utime* function is:

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

int utime ( const char* path_name, struct utimbuf* times );
```

The *path_name* argument is the path name of a file. The *times* argument specifies the new access time and modification time for the file. The *struct utimbuf* is defined in the <utime.h> header as:

```
struct utimbuf
{
        time_t    actime;              /* access time */
        time_t    modtime              /* modification time */
};
```

POSIX.1 defines the *struct utimbuf* in the <utime.h> header, whereas UNIX System V defines it in the <sys/types.h> header. The *time_t* data type is the same as *unsigned long*, and its data is the number of seconds elapsed since the birthday of UNIX: 12 AM, January 1, 1970 UTC (Universal Time Coordinate).

If *times* is specified as 0, the API will set the named file access time and modification time to the current time. This requires that the calling process have write access to the named file, its effective user ID match either the file user ID or that of the superuser.

If *times* is an address of a variable of type *struct utimbuf*, the API will set the file access time and modification time according to the values specified in the variable. This requires that the calling process effective UID either match the file UID or be the same as a superuser.

The return value of *utime* is 0 if it succeeds or -1 if it fails. Possible failures of the API may be: The *path_name* argument is invalid, the process has no access permission and ownership to a named file, or the *times* argument has an invalid address.

The following *test_touch.C* program uses the *utime* function to change the access and modification time stamps of files. The time stamp to set is also defined by users:

```
/* Usage: a.out <offset in seconds> <file> ... */
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <utime.h>
#include <time.h>

int main ( int argc, char* argv[] )
{
   struct utimbuf   times;
   int              offset;

   if (argc < 3 || sscanf( argv[1], "%d", &offset) != 1) {
      cerr << "usage: " << argv[0] << " <offset> <file> ...\n";
      return 1;
   }

   /* new time is current time + offset in seconds */
   times.actime = times.modtime = time( 0 ) + offset;

   for ( -i=1; i < argc; i ++)                /* touch each named file */
      if ( utime ( argv[i], &times ) ) perror( "utime" );
   return 0;
}
```

The above program defines a variable called *times* of type *struct utimbuf* and initializes it with the current time value (as obtained from the *time* function call) plus a user-specified offset time (in seconds). The subsequent command line arguments to the program should be one or more file path names. For each of these files, the program will call *utime* to update the file access time and modification time. If any *utime* call fails, the program will call *perror* to print a diagnostic message.

7.2 File and Record Locking

UNIX systems allow multiple processes to read and write the same file concurrently. This provides a means for data sharing among processes, but it also renders difficulty for any process in determining when data in a file can be overridden by another process. This is especially important for applications like a database manager, where no other process can write or read a file while a process is accessing a database file. To remedy this drawback, UNIX and POSIX systems support a file-locking mechanism. File locking is applicable only for regular files. It allows a process to impose a lock on a file so that other processes can not modify the file until it is unlocked by the process.

Specifically, a process can impose a write lock or a read lock on either a portion of a file or an entire file. The difference between write locks and read locks is that when a write lock is set, it prevents other processes from setting any overlapping read or write locks on the locked region of a file. On the other hand, when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region of a file. It does, however, allow overlapping read locks to be set on the file by other processes. Thus, the intention of a write lock is to prevent other processes from both reading and writing the locked region while the process that sets the lock is modifying the region. A write lock is also known as an exclusive lock. The use of a read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region. Other processes are allowed to lock and read data from the locked regions. Hence, a read lock is also called a *shared lock*.

Furthermore, file locks are mandatory if they are enforced by an operating system kernel. If a mandatory exclusive lock is set on a file, no process can use the *read* or *write* system calls to access data on the locked region. Similarly, if a mandatory shared lock is set on a region of a file, no process can use the *write* system call to modify the locked region. These mechanisms can be used to synchronize reading and writing of shared files by multiple processes: If a process locks up a file, other processes that attempts to write to the locked regions are blocked until the former process releases its lock. However, mandatory locks may cause problems: If a runaway process sets a mandatory exclusive lock on a file and never unlocks it, no other processes can access the locked region of the file until either the runaway process is killed or the system is rebooted. System V.3 and V.4 support mandatory locks, but BSD UNIX and POSIX systems do not.

If a file lock is not mandatory, it is an advisory lock. An advisory lock is not enforced by a kernel at the system call level. This means that even though a lock (read or write) may be set on a file, other processes can still use the *read* or *write* APIs to access the file. To make use of advisory locks, processes that manipulate the same file must cooperate such that they follow this procedure for every read or write operation to the file:

- Try to set a lock at the region to be accessed. If this fails, a process can either wait for the lock request to become successful or go do something else and try to lock the file again later
- After a lock is acquired successfully, read or write the locked region
- Release the lock

By always attempting to set an advisory lock on a region of a file to be worked on, a process will not violate any lock protection set by other processes on the same area, and other processes will not modify that area while the lock in imposed. A process should always release any lock that it imposes on a file as soon as it is done, so that other processes can access the now unlocked region. An advisory lock is considered safe, as no runaway processes can lock up any file forcefully, and other processes can go ahead and read or write to a file after a fixed number of failed attempts to lock the file.

The drawback of advisory locks are that programs that create processes to share files must follow the above file locking procedure to be cooperative. This may be difficult to control when programs are obtained from different sources (e.g., from different software vendors). All UNIX and POSIX systems support advisory locks.

UNIX System V and POSIX.1 use the *fcntl* API for file locking. Specifically, the API can be used to impose read or write locks on either a segment or an entire file. The *fcntl* API in BSD UNIX 4.2 and 4.3 does not support the file locking option. The prototype of the *fcntl* API is:

```
#include <fcntl.h>

int fcntl ( int fdesc, int cmd_flag, ... );
```

The *fdesc* argument is a file descriptor for a file to be processed. The *cmd_flag* argument defines which operation is to be performed. The possible *cmd_flag* values are defined in the <fcntl.h> header. The specific values for file locking and their uses are:

cmd_flag	**Use**
F_SETLK	Sets a file lock. Do not block if this cannot succeed immediately
F_SETLKW	Sets a file lock and blocks the calling process until the lock is acquired
F_GETLK	Queries as to which process locked a specified region of a file

For file locking, the third argument to *fcntl* is an address of a *struct flock*-typed variable. This variable specifies a region of a file where the lock is to be set, unset, or queried. The *struct flock* is declared in the <fcntl.h> as:

```
struct flock
{
   short   l_type;      /* what lock to be set or to unlock file */
   short   l_whence;    /* a reference address for the next field */
   off_t   l_start;     /* offset from the l_whence reference address */
   off_t   l_len;       /* how many bytes in the locked region */
   pid_t   l_pid;       /* PID of a process which has locked the file */
};
```

The *l_type* field specifies the lock type to be set or unset. The possible values, which are defined in the <fcntl.h> header, and their uses are:

l_type **value**	**Use**
F_RDLCK	Sets a a read (shared) lock on a specified region
F_WRLCK	Sets a write (exclusive) lock on a specified region
F_UNLCK	Unlocks a specified region

I he *l_whence, l_start,* and *l_len* define a region of a file to be locked or unlocked. This is similar to the *lseek* API, where the *l_whence* field defines a reference address to which the *l_start* byte offset value is added. The possible values of *l_whence* and their uses are:

l_whence **value**	**Use**
SEEK_CUR	The *l_start* value is added to the current file pointer address
SEEK_SET	The *l_start* value is added to byte 0 of the file
SEEK_END	The *l_start* value is added to the end (current size) of the file

The *l_len* specifies the size of a locked region beginning from the start address as defined by *l_whence* and *l_start*. If *l_len* is a positive number greater than 0, it is the length of the locked region in number of bytes. If *l_len* is 0, the locked region extends from its start address to a system-imposed limit on the maximum size of any file. This means that as the file size increases, the lock also applies to the extended file region. The *l_len* cannot have a negative value.

A *struct flock*-typed variable is defined and set by a process before it is passed to a *fcntl* call. If the *cmd_arg* of the *fcntl* call is F_SETLK or F_SETLKW, the variable defines a region of a file to be locked or unlocked. If, however, the *cmd_arg* is F_GETLK, the variable is used as both an input and a return variable. Specifically, when *fcntl* is called, the variable specifies a region of a file where lock status is queried. Then, when *fcntl* returns, the variable contains the region of the file that is locked and the ID of the process that owns the locked region. The process ID is returned via the *l_pid* field of the variable.

Note that if a process sets a read lock on a file, for example from address 0 to 256, then sets a write lock on the file from address 0 to 512, the process will own only one write lock on the file from 0 to 512. The previous read lock from 0 to 256 is now covered by the write lock, and the process does not own two locks on the region from 0 to 256. This process is called *lock promotion*. Furthermore, if the process now unlocks the file from 128 to 480, it will own two write locks on the file: one from 0 to 127 and the other from 481 to 512. This process is called *lock splitting*.

A lock set by the *fcntl* API is an advisory lock. POSIX.1 does not support mandatory locks. UNIX System V.3 and V.4, however, permit users to set mandatory locks via *fcntl*. This is achieved by setting the following attributes of a file, and thereafter any locks set by *fcntl* on that file will be mandatory locks:

- Turn on the *set-GID* flag of the file
- Turn off the group execute right of the file

Alternatively, the *chmod* command in UNIX System V.3 and V.4 may also be used to specify that any read or write locks set on a file are mandatory. The *chmod* command syntax is:

```
chmod  a+l  <file_name>
```

All file locks set by a process will be unlocked when the process terminates. Furthermore, if a process locks a file and then creates a child process via *fork* (the *fork* API is explained in the next chapter), the child process will not inherit the file lock.

The return value of *fcntl* is 0 if it succeeds or -1 if it fails. Possible causes of failure may be that the file descriptor is invalid, the requested region to be locked or unlocked conflicts with locks set by another process, the third argument contains invalid data, or the system-tunable limit on the maximum number of record locks per file has been reached.

The following *file_lock.C* program illustrates a use of *fcntl* for file locking:

```
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

int main (int argc, cnar* argv[])
{
   struct flock          fvar;
   int                   fdesc;
   while (--argc > 0) {                   /* do the following for each file */
      if (( fdesc=open(*++argv,O_RDWR ))==-1 ) {
         perror("open"); continue;
      }
      fvar.l_type        = F_WRLCK;
      fvar.l_whence      = SEEK_SET;
      fvar.l_start       = 0;
      fvar.l_len         = 0;
      /* Attempt to set an exclusive (write) lock on the entire file */
      while (fcntl(fdesc, F_SETLK,&fvar)==-1) {
         /* Set lock fails, find out who has locked the file */
         while (fcntl(fdesc,F_GETLK,&fvar)!=-1 &&
                fvar.l_type!=F_UNLCK) {
                cout << *argv << " locked by " << fvar.l_pid
                          << " from " << fvar.l_start << " for "
                          << fvar.l_len << " byte for " <<
                          (fvar.l_type==F_WRLCK ? 'w' : 'r') << endl;
                if (!fvar.l_len) break;
                fvar.l_start   += fvar.l_len;
                fvar.l_len     = 0;
            } /* while there are locks set by other processes */
        } /* while set lock un-successful */

        /* Lock the file OK. Now process data in the file */
        ...
        /* Now unlock the entire file */
        fvar.l_type         = F_UNLCK;
        fvar.l_whence       = SEEK_SET;
        fvar.l_start        = 0;
        fvar.l_len          = 0;
        if (fcntl(fdesc, F_SETLKW,&fvar)==-1) perror("fcntl");
    }
    return 0;
} /* main */
```

The above program takes one or more path names as arguments. For each file specified, the program attempts to set an advisory lock on the entire file via *fcntl*. If the *fcntl* call fails, the program scans the file to list all lock information to the standard output. Specifically, for each locked region, the program reports:

- The file path name
- The process ID that locks that region
- The start address of the locked region
- The length of the locked region
- Whether the lock is exclusive (*w*) or shared (*r*)

The program loops repeatedly until a *fcntl* call succeeds in locking the file. After that, the program will process the file in some way, then it calls *fcntl* again to unlock the file.

7.3 Directory File APIs

Directory files in UNIX and POSIX systems are used to aid users in organizing their files into some structure based on the specific use of files (e.g., a user may store all C++ source of a program under the */usr/<program_name>/C* directory). They are also used by the operating system to convert file path names to their inode numbers.

Directory files are created in BSD UNIX and POSIX.1 by the *mkdir* API:

```
#include <sys/stat.h>
#include <unistd.h>

int mkdir ( const char* path_name, mode_t mode );
```

The *path_name* argument is the path name of a directory file to be created. The *mode* argument specifies the access permission for the owner, group, and others to be assigned to the file. Like in the *open* API, the *mode* value is modified by the calling process umask.

The return value of *mkdir* is 0 if it succeeds or -1 if it fails. Possible causes of failure may be: The *path_name* is invalid, the calling process lacks permission to create the specified directory, or the *mode* argument is invalid.

UNIX System V.3 uses the *mknod* API to create directory files. UNIX System V.4 supports both the *mkdir* and *mknod* APIs for creating directory files. The difference between the two APIs is that a directory created by *mknod* does not contain the "." and ".." links; thus, it is not usable until those links are explicitly created by a user. On the other hand, a directory created by *mkdir* has the "." and ".." links created in one atomic operation, and it is ready to be used. In general, *mknod* should not be used to create directories. Furthermore, on systems that do not support the *mkdir* API, one can still create directories via the *system* API:

```
char     syscmd[ 256 ];
sprintf( syscmd, "mkdir    %s", <directory_name> );
if ( system ( syscmd ) == -1)     perror( "mkdir");
```

A newly created directory has its user ID set to the effective user ID of the process that creates it, and the directory group ID will be set to either the effective group ID of the calling process or the group ID of the parent directory that hosts the new directory (in the same manner as for regular files).

A directory file is a record-oriented file, where each record stores a file name and the inode number of a file that resides in that directory. However, the directory record structure is different on different file systems. For example, UNIX system V directory records are fixed-size, whereas BSD UNIX directory records are of variable size. To allow a process to scan directories in a file system-independent manner, a directory record is defined as *struct dirent* in the <dirent.h> header for UNIX System V and POSIX.1, and as *struct direct* in the <sys/dir.h> header in BSD UNIX 4.2 and 4.3. The *struct dirent* and *struct direct* data types have one common field, *d_name*, which is a character array that contains the name of a file residing in a directory. Furthermore, the following portable functions are defined for directory file browsing. These functions are defined in both the <dirent.h> and <sys/dir.h> headers.

```
#include <sys/types.h>
#if defined (BSD) && !_POSIX_SOURCE
   #include <sys/dir.h>
   typedef struct direct Dirent;
#else
   #include <dirent.h>
   typedef struct dirent Dirent;
#endif

DIR* opendir (const char* path_name);
Dirent* readdir (DIR* dir_fdesc);
int closedir (DIR* dir_fdesc);
void rewinddir (DIR* dir_fdesc);
```

The uses of these functions are:

Function	**Use**
opendir	Opens a directory file for read-only. Returns a file handle DIR* for future reference of the file
readdir	Reads a record from a directory file referenced by *dir_fdesc* and returns that record information
closedir	Closes a directory file referenced by *dir_fdesc*
rewinddir	Resets the file pointer to the beginning of the directory file referenced by *dir_fdesc*. The next call to *readdir* will read the first record from the file

The *opendir* function is analogous to the *open* API. It takes a directory file path name as an argument and opens the file for read-only. The function returns a DIR* file handler which has a use similar to that of FILE* value returned from *fopen*. The DIR data structure is defined in the <dirent.h> or <sys/dir.h> header.

The *readdir* function reads the next directory record from a directory file referenced by the *dir_fdesc* argument. The *dir_fdesc* value is the DIR* return value from an *opendir* call. The function returns the address of a *struct dirent* or *struct direct* record, which stores the file name of a file entry in the directory. When *readdir* is called after the *opendir* or *rewinddir* API, it will return the first data record from the file, on the next call it will return the second data record in the file, etc. When *readdir* has scanned all records in a directory file, it will return a zero value to indicate that end-of-file has been reached. Note that a data type called *Dirent* was defined to be either *struct dirent* (for POSIX and System V UNIX) or *struct direct* (for BSD UNIX in non-POSIX conformance mode). In this way, any application that calls *readdir* can treat the return value uniformly as *Dirent* in any system.

The *closedir* function is analogous to the *close* API. It terminates the connection between the *dir_fdesc* handler and a directory file.

The *rewinddir* function resets a file pointer associated with a *dir_fdesc*, so that if *readdir* is called again, it will scan the directory file (referenced by the *dir_fdesc*) from the beginning.

UNIX systems (System V and BSD UNIX) have defined additional functions for random access of directory file records. These functions are not supported by POSIX.1:

Function	**Use**
telldir	Returns the file pointer of a given *dir_fdesc*
seekdir	Changes the file pointer of a given *dir_fdesc* to a specified address

Directory files are removed by the *rmdir* API. Users may also use the *unlink* API to remove directories, provided they have superuser privileges. These APIs require that the directories to be removed be empty, in that they contain no files other than the "." and ".." links. The prototype of *rmdir* function is:

```
#include <unistd.h>

int rmdir (const char* path_name);
```

The following *list_dir.C* program illustrates uses of the *mkdir, opendir, readdir, closedir,* and *rmdir* APIs:

```
#include <iostream.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>
#if defined (BSD) && !_POSIX_SOURCE
   #include <sys/dir.h>
   typedef struct direct Dirent;
#else
   #include <dirent.h>
   typedef struct dirent Dirent;
#endif
```

```
int main ( int argc, char* argv[] )
{
   Dirent*    dp;
   DIR*       dir_fdesc;

   while ( --argc > 0 ) {                      /* do the following for each file */
      if ( !(dir_fdesc = opendir( *++argv ) )) {
         if (mkdir( *argv, S_IRWXU|S_IRWXG|S_IRWXO ) == -1 )
            perror( "opendir" );
         continue;
      }

      /* scan each directory file twice */
      for ( int i=0; i < 2; i++ ) {
         for ( int cnt=0; dp=readdir( dir_fdesc ); ) {
            if (i) cout << dp->d_name << endl;
            if (strcmp( dp->d_name, "." ) && strcmp( dp->d_name, ".." ) )
                  cnt++;                       /* count how many files in directory*/
         }
         if (!cnt) { rmdir( *argv ); break; }  /* empty directory */
         rewinddir( dir_fdesc );               /* reset pointer for second round */
      }
      closedir( dir_fdesc );
   }                                           /* for each file */
}                                              /* main */
```

The above program takes one or more directory file path names as arguments. For each argument, the program opens it via the *opendir* and stores the file handler in the *dir_fdesc* variable. If the *opendir* call fails, the program assumes the directory does not exist and attempts to create it via the *mkdir* API. If, however, the *opendir* succeeds, the program scans the directory file using the *readdir* API and determines the number of files, excluding "." and ".." files, in the directory. After this is done, it removes the directory, via the *rmdir* API, if it is empty. If a directory is not empty, the program resets the file pointer associated with the *dir_fdesc* via the *rewinddir*, it then scans the directory file a second time and echoes all file names in that directory to the standard output. When the second round of directory scanning is completed, the program closes the *dir_fdesc* with the *closedir* API.

7.4 Device File APIs

Device files are used to interface physical devices (e.g, console, modem, floppy drive) with application programs. Specifically, when a process reads or writes to a device file, the kernel uses the major and minor device numbers of a file to select a device driver function to carry out the actual data transfer. Device files may be character-based or block-based.

Device file support is implementation-dependent. POSIX.1 does not specify how device files are to be created. UNIX systems define the *mknod* API to create device files:

```
#include <sys/stat.h>
#include <unistd.h>

int mknod (const char* path_name, mode_t mode, int device_id);
```

The *path_name* argument is the path name of a device file to be created. The *mode* argument specifies the access permission, for the owner, group, and others, to be assigned to the file, as well as the S_IFCHR or S_IFBLK flag. The latter flag is used to indicate whether this is a character or block device file. Access permission is modified by the calling process umask. Finally, the *device_id* contains the major and minor device numbers and is constructed in most UNIX systems as follows: The lowest byte of a *device_id* is set to a minor device number and the next byte is set to the major device number. For example, to create a block device file called *SCSI5* with major and minor numbers of 15 and 3, respectively, and access rights of read-write-execute for everyone, the *mknod* system call is:

```
mknod("SCSI5", S_IFBLK|S_IRWXU|S_IRWXG|S_IRWXO, (15<<8) | 3);
```

Note that in UNIX System V.4, the major and minor device numbers are extended to fourteen and eighteen bits, respectively. The major and minor device numbers are used as follows: When a process reads from or writes to a device file, the file's major device number is used to locate and invoke a device driver function that does the actual data transmission with the physical device. The minor device number is an argument being passed to the device driver function when it is invoked. This is needed because a device driver function may be used for different types of device, and the minor device number specifies the parameters (e.g. buffer size) to be used for a particular device type.

The *mknod* API must be called by a process with superuser privileges. The user ID and group ID attributes of a device file are assigned in the same manner as for regular files. The file size attribute of any device file has no meaningful use.

The return value of *mknod* is 0 if it succeeds or -1 if it fails. Possible failures include: The path name specified is invalid, the process lacks permission to create a device file, or the *mode* argument is invalid.

Once a device file is created, any process may use the *open* API to connect to the file. It can then use *read, write, stat,* and *close* APIs to manipulate the file. *lseek* is applicable to block device files but not to character device files. A device file may be removed via the *unlink* API.

When a process calls *open* to establish connection with a device file, it may specify the O_NONBLOCK and O_NOCTTY flags that are defined by POSIX.1. Uses of these flags are depicted in the following.

In UNIX, if a calling process has no controlling terminal and it opens a character device file, the kernel will set this device file as the controlling terminal of the process. However, if the O_NOCTTY flag is set in the *open* call, such action will be suppressed.

The O_NONBLOCK flag specifies that the *open* call and any subsequent *read* or *write* calls to a device file should be nonblocking to the process.

Only privileged users (e.g., the superuser in UNIX) may use the *mknod* API to create device files. All other users may read and write device files as if they were regular files, subjected to the access permissions set on those device files.

The following *test_mknod.C* program illustrates use of the *mknod, open, read, write,* and *close* APIs on a block device file.

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

int main( int argc, char* argv[] )
{
   if ( argc != 4 ) {
      cout << “usage: “ << argv[0] << “ <file> <major no> <minor no>\n”;
      return 0;
   }
   int major = atoi( argv[2]), minor = atoi( argv[3] );
   (void)mknod( argv[1], S_IFCHR|S_IRWXU|S_IRWXG|S_IRWXO,
         ( major <<8 ) | minor );

   int rc=1, fd = open( argv[1], O_RDWR|O_NONBLOCK|O_NOCTTY );
   char buf[256];

   while ( rc && fd != -1 )
      if (( rc = read( fd, buf, sizeof( buf )) ) < 0 )
            perror( “read” );
      else if ( rc ) cout << buf << endl;
   close(fd);
} /* main */
```

This program takes three arguments: the first one is a device file name, the second one a major device number, and the last one a minor device number. The program will use these arguments to create a character device file via *mknod*. The program opens the file for read-write and sets the O_NONBLOCK and NO_CTTY flags. It then reads data from the device file and echoes data to the standard output. When end-of-file is encountered, the program closes the file descriptor associated with the device file and terminates.

Users should notice that the treatment of device files is almost identical to that of regular files, the only differences are the ways device files are created and the fact that *lseek* is not applicable for character device files.

7.5 FIFO File APIs

FIFO files are also known as *named pipes*. They are special *pipe device* files used for interprocess communication. Specifically, any process can attach to a FIFO file to read, write, or read-write data. Data written to a FIFO file are stored in a fixed-size (PIPE_BUF, as defined in the <limits.h> header) buffer and are retrieved in a first-in-first-out (FIFO) order.

BSD UNIX and POSIX.1 define the *mkfifo* API to create FIFO files:

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

int mkfifo ( const char* path_name, mode_t mode );
```

The *path_name* argument is the path name of a FIFO file to be created. The *mode* argument specifies the access permission, for user, group, and others, to be assigned to the file, as well as the S_IFIFO flag to indicate that this is a FIFO file. Access permission is modified by the calling process umask. The user ID and group ID attributes of a FIFO file are assigned in the same manner as for regular files.

For example, to create a FIFO file called *FIFO5* with access permission of read-write-execute for everyone, the *mkfifo* call is:

```
mkfifo( "FIFO5", S_IFIFO | S_IRWXU | S_IRWXG | S_IRWXO );
```

The return value of *mkfifo* is 0 if it succeeds or -1 if it fails. Possible failures may be: The path name specified is invalid, a process lacks permission to create the file, or an invalid *mode* argument is specified.

UNIX System V.3 uses the *mknod* API to create FIFO files. However, UNIX System V.4 supports the *mkfifo* API.

Once a FIFO file is created, any process may use the *open* API to connect to the file,. It can then use *read, write, stat,* and *close* APIs to manipulate the file. *lseek* is not applicable to FIFO files. A FIFO file may be removed via the *unlink* API.

When a process opens a FIFO file for read-only, the kernel will block the process until there is another process that opens the same file for write. Similarly, if a process opens a FIFO for write, it will be blocked until another process opens the FIFO file for read. This provides a method for processes synchronization. Furthermore, if a process writes to a FIFO that is full, the process will be blocked until another process has read data from the FIFO to make room for new data in the FIFO. Conversely, if a process attempts to read data from a FIFO that is empty, the process will be blocked until another process writes data to the FIFO.

If a process does not desire to be blocked by a FIFO file, it can specify the O_NONBLOCK flag in the *open* call to the FIFO file. With this flag, the *open* API will not block the process even though there is no process attached to the other end of the FIFO file. Furthermore, if the process subsequently calls the *read* or *write* API on the FIFO file and data is not ready for transfer, these functions will return immediately with a -1 value, and set error to EAGAIN. Thus, the process can continue to do something else and try these operations later. UNIX System V defines the O_NDELAY flag which has is similar to the O_NONBLOCK flag. The difference with the O_NDELAY flag is that the *read* and *write* functions will return a zero value when they are supposed to block a process. In this case.it is difficult to differentiate between an end-of-file condition and an empty one (where there is the possibility of more being written). UNIX System V.4 supports both the O_NDELAY and O_NONBLOCK flags.

Another special thing about FIFO files is that if a process writes to a FIFO file that has no other process attached to it for read, the kernel will send a SIGPIPE signal (signals are described in Chapter 9) to the process to notify it of the illegal operation. Furthermore, if a process reads a FIFO file that has no process attached to its write end, the process will read the remaining data in the FIFO and then an end-of-file indicator. Thus, if two processes are to communicate via a FIFO file, it is important that the writer process closes its file descriptor when it is done, so that the reader process can see the end-of-file condition.

It is possible for a process to open a FIFO file for both read and write. POSIX.1 does not specify how the kernel should handle this, but in UNIX systems the process will not be blocked by the *open* call. The process can use the file descriptor returned from the *open* API to read and write data with the FIFO file.

The following *test_fifo.C* example illustrates use of *mkfifo, open, read, write,* an *close* APIs for a FIFO file:

```
#include <iostream.h>
#include <stdio.h>
```

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <errno.h>

int main( int argc, char* argv[] )
{
   if ( argc != 2 && argc != 3 ) {
      cout << "usage: " << argv[0] << " <file> [<arg>]\n";
      return 0;
   }
   int      fd;
   char     buf[256];
   (void)mkfifo( argv[1], S_IFIFO|S_IRWXU|S_IRWXG|S_IRWXO );
   if (argc==2) {                          /* reader process */
         fd = open( argv[1],O_RDONLY|O_NONBLOCK );
         while ( read( fd, buf, sizeof( buf ) )==-1 && errno==EAGAIN )
               sleep( 1 );
         while ( read( fd, buf, sizeof( buf ) ) > 0 )
               cout << buf << endl;
   } else   {                              /* writer process */
         fd = open( argv[1], O_WRONLY );
         write( fd, argv[2], strlen( argv[2] ) );
   }
   close(fd);
}
```

The above program takes one or two arguments. The first argument is the name of a FIFO file to be used. The program calls *mkfifo* to create the FIFO file if it does not exist. It then checks whether it has one or two arguments. If it has one argument, it will open the FIFO file for read-only. It will then read all data from the FIFO file and echo them to the standard output. However, if the process has two arguments, it will open the FIFO file for write and will write the second argument to the FIFO file. Thus, this program can be run twice to create two processes that communicate through a FIFO file. Assuming the program has been compiled into an executable file call *a.out*, the sample run of this program is:

```
%    a.out FF64                      # create a reader process
%    a.out FF64 "Hello world"        # create a writer process
Hello world                          # reader process output
```

Another method to create FIFO files for interprocess communication is to use the *pipe* API:

```
#include <unistd.h>
int pipe ( int fds[2] );
```

The *pipe* API creates the same FIFO file as does *mkfifo*. However, the FIFO file created by the *pipe* API is transient: There is no file created in a file system to associate with the FIFO file, and it will be discarded by the kernel once all processes close their file descriptors that reference the FIFO. The uses of the *fds* argument are: *fds[0]* is a file descriptor to read data from the FIFO file, and *fds[1]* is a file descriptor to write data to the FIFO file. Because the FIFO file cannot be referenced by a path name, its use is restricted to processes that are related: The FIFO file is created by a parent process, which then creates one or more child processes; these child processes inherit the FIFO file descriptors from the parent, and they can communicate among themselves, and with the parent, via the FIFO file. The restrictive use of FIFO files created by the *pipe* API caused the invention of named pipes, so that unrelated processes can communicate using FIFO files.

7.6 Symbolic Link File APIs

Symbolic links are defined in BSD UNIX 4.2 and used in BSD 4.3, System V.3 and V.4. Symbolic links are developed to overcome several shortcomings of hard links:

- Symbolic links can link files across file systems
- Symbolic links can link directory files
- Symbolic links always reference the latest version of the files to which they link

The last point is the major advantage of symbolic links over hard links. For example, suppose a user creates a file called */usr/go/test1* and a hard link to it called */usr/joe/hdlnk*:

```
ln      /usr/go/test1       /usr/joe/hdlnk
```

If the user deletes the */usr/go/test1*, the file is now referenced by */usr/joe/hdlnk* only. However, if the user then creates a file called */usr/go/test1*, which is a file totally different than */usr/joe/hdlnk*, the */usr/joe/hdlnk* will still refer to the old file, whereas the */usr/go/test1* now refers to the new file. Thus, hard links can be broken by removal of one or more links.

In the above example if a symbolic link is used instead, the link will not be broken. Specifically, after a symbolic link called */usr/joe/synlnk* is created as:

```
ln  -s  /usr/go/test1     /usr/joe/symlnk
```

If the user deletes the */usr/go/test1*, the */usr/joe/symlnk* will refer to a nonexistent file, and any operations on that link (*cat, more, sort,* etc.) will fail. However, if the user creates the new */usr/go/test1*, the */usr/joe/symlnk* will automatically refer to this new file, and the link is reestablished again.

Symbolic links are being proposed to be included the POSIX.1 standard. BSD UNIX defines the following APIs for symbolic links manipulation:

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

int symlink ( const char* org_link, const char* sym_link );
int readlink ( const char* sym_link, char* buf, int size );
int lstat ( const char* sym_link, struct stat* statv );
```

The *org_link* and *sym_link* arguments to a *symlink* call specify the original file path name and the symbolic link path name to be created. For example, to create a symbolic link called */usr/joe/lnk* for a file called */usr/go/test1*, the *symlink* call will be:

```
symlink ( "/usr/go/test1", "usr/joe/lnk" );
```

This syntax is the same as that of the *link* API. The return value of *symlink* is 0 if it succeeds or -1 if it fails. Possible causes of failure are: The path name specified is illegal, the *sym_link* file already exists, or the calling process lacks permission to create the new file.

To query the path name to which a symbolic link refers, users must use the *readlink* API. This is necessary because the *open* API automatically resolves any symbolic link to the actual file to which a link refers, and it will connect a calling process to the actual nonlink file. The arguments to the *readlink* API are: *sym_link* is the path name of a symbolic link, *buf* is a character array buffer that holds the return path name referenced by the link, and *size* specifies the maximum capacity (in number of bytes) of the *buf* argument. The return value of *readlink* is -1 if it fails or the actual number of characters of a path name that is placed in the *buf* argument. Possible causes of failure for *readlink* are: The *sym_link* path name is not a symbolic link, the *buf* argument is an illegal address, or a calling process lacks permission to access the symbolic link file.

The following function takes a symbolic link path name as argument, and it will call *readlink* repeatedly to resolve all links to the file. The *while* loop terminates when *readlink* returns -1, and the *buf* variable contains the nonlink file path name, which is then printed to the standard output:

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>

int resolve_link( const char* sym_link )
{
   char*   buf[256], tname[256];
   strcpy(tname,sym_link);
   while ( readlink( tname, buf, sizeof( buf ) ) > 0 )
      strcpy( tname, buf );
   cout <<sym_link << " => " <<buf << endl;
}
```

The *lstat* function is used to query the file attributes of symbolic links. This is needed as the *stat* and *fstat* functions show only nonsymbolic link file attributes. The function prototype and return values of *lstat* are the same as those of *stat*. Furthermore, *lstat* can be used on non-symbolic link files, and it behaves like *stat*. The UNIX *ls -l* command uses *lstat* to display information of all file type, including symbolic links.

The following *test_symln.C* program emulates the UNIX *ln* command. The main function of the program is to create a link to a file. The names of the original file and new link are specified as the arguments to the program, and if the *-s* option is not specified, the program will create a hard link. Otherwise, it will create a symbolic link:

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>

/* Emulate the UNIX ln command */
int main (int argc, char* argv[])
{
   char*   buf[256], tname[256];
   if ((argc< 3 && argc > 4) || (argc==4 && strcmp(argv[1],"-s"))) {
      cout << "usage: " << argv[0] << " [-s] <orig_file> <new_link>\n";
      return 1;
   }
   if (argc==4)
      return symlink( argv[2], argv[3]);/* create a symbolic link */
   else
      return link(argv[1], argv[2]);/* create a hard link */
}
```

7.7 General File Class

The C++ *fstream* class can be used to define objects that represent files in a file system. Specifically, the *fstream* class contains member functions like *open, close, read, write, tellg,* and *seekg,* which are based on the *open, read, write,* and *lseek* APIs. Thus, any application program can define *fstream* class objects associated with files for read and write.

However, *fstream* does not provide any means for users to perform *stat, chmod, chown, utime,* and *link* functions on its objects. It also does not create any files other than regular files. Thus *fstream* does not encapsulate the complete POSIX and UNIX systems file object functions.

To remedy the *fstream* class deficiency, a new *filebase* class is defined below, which incorporates the *fstream* class properties and additional functions to allow users to get or change object file attributes and to create hard links:

```
/* filebase.h */
#ifndef FILEBASE_H
#define FILEBASE_H

#include<iostream.h>
#include <fstream.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
#include<utime.h>
#include <fcntl.h>
#include <string.h>

typedef enum {  REG_FILE='r', DIR_FILE='d', CHAR_FILE='c',
                BLK_FILE='b', PIPE_FILE='p', SYM_FILE='s',
                UNKNOWN_FILE='?'}    FILE_TYPE_ENUM;

/* A base class to encapsulate POSIX and UNIX file objects' properties */
class filebase:     public fstream
{
  protected:
    char*filename;
      friend ostream& operator<<( ostream& os, filebase& fobj )
    {
      /* display file attributes in UNIX ls -l command output format */
      return os;
    };
  public:
    filebase()          { filename = 0; };
```

```
        filebase( const char* fn, int flags, int prot=filebuf::openprot )
              : fstream( fn, flags, prot )
        {
           filename = new char[strlen(fn)+1];
           strcpy(filename,fn);
        };
        virtual ~filebase()  { delete filename; };
        virtual int create( const char* fn, mode_t mode )
                            {      return ::creat ( fn, mode );              };
        int fileno()        {      return rdbuf()->fd();                     };
        int chmod( mode_t mode )
                            {      return ::chmod ( filename, mode );        };
        int chown( uid_t uid, gid_t gid )
                            {      return ::chown( filename, uid, gid );     };
        int utime( const struct utimbuf *timbuf_Ptr )
                            {      return ::utime( filename,timbuf_Ptr );    };
        int link( const char* new_link )
                            {      return ::link( filename, new_link );      };
        virtual int remove( ){     return ::unlink( filename );              };

        // Query a filebase object's file type
        FILE_TYPE_ENUM file_type()
        {
           struct statstatv;
           if (stat( filename,&statv )==0 )
              switch ( statv.st_mode & S_IFMT)  {
                 case S_IFREG: return REG_FILE;          // regular file
                 case S_IFDIR:  return DIR_FILE;         // directory file
                 case S_IFCHR: return CHAR_FILE;         // char device file
                 case S_IFIFO:  return PIPE_FILE;        // block device file
                 case S_IFLNK:  return SYM_FILE;         // symbolic link file
              }
           return UNKNOWN_FILE;
        };
};
#endif  /* filebase.h */
```

The *filebase* constructor passes its arguments to its *fstream* superclass constructor for connection with an object to a file named by *fn* with the specified access mode. It then allocates a dynamic buffer to hold the *fn* path name in a *filename* private variable. The *filename* variable is used in other member functions like *chmod*, *link*, and *remove*, etc.

The *filebase* destructor deallocates the *filename* dynamic buffer. It then uses its *fstream* destructor to disconnect an object from a file named by *filename*.

The *fileno* member function returns the file descriptor of a file managed by a filebase object.

The *chmod* member function aids users in changing access permission of a file connected to a *filebase* object. The argument *mode* is the same as that for the POSIX.1 *chmod* function and the *filename* variable specifies which file access permission is to be changed.

The *chown* member function aids users in changing the user ID and group ID of a file connected to a *filebase* object. The arguments *uid* and *gid* are the same as that for the POSIX.1 *chown* function, and the *filename* variable specifies which file user IDs and group IDs are to be changed.

The *utime* member function helps users change the access and modification time stamps of a file connected to a *filebase* object. The argument *timbuf_Ptr* is the same as that for the POSIX.1 *utime* function, and the *filename* variable specifies which file access and modification time stamps are to be changed.

The *link* member function allows users to create a hard link to a file connected to a *filebase* object. The argument *new_link* contains the new link name. The original link of a file connected to an object is obtained from the *filename* variable of the object. This function calls the POSIX.1 *link* function to create a new hard link.

The *create* member function creates a file with a given name. This function calls the *creat* API to create a named file in a file system, and it is applicable only for regular files.

The *remove* member function removes a link referenced by the *filename* variable of an object. If this call succeeds, no processes can reference the file with that path name.

The overloaded "<<" operator of *ostream* is used to display the file attributes of a file connected to a *filebase* object. The function may call the *long_list* function (as depicted in Section 7.1.10) to query and display the file properties named by the *filename* private variable.

The *file_type* function determines the file type of any *filebase* object. It calls the *stat* API on a given *filename* and determine the file type of a corresponding object.The function returns an enumerator based on the FILE_TYPE_ENUM data type. If a *stat* call fails or the file type of an object is undetermined, the function returns an UNKNOWN_FILE value. Otherwise, it returns one of the REG_FILE, DIR_FILE, etc. enumerators as return value.

To illustrate the use of the *filebase* class, the following *test_filebase.C* program defines a *filebase* object called *rfile*, to be associated with a file called */usr/text/unix.doc* for read. Furthermore, the program displays object file attributes to the standard output and changes the file user ID and group ID to 15 and 30, respectively. Furthermore, it changes the file access and modification times to the current time, and then creates a hard link called */home/jon/hdlnk*. Finally, it removes the original link, */usr/text/unix.doc*. The *test filebase.C* program is:

```
#include "filebase.h"
int main()
{                                                    // Example for filebase
    filebase    rfile("/usr/text/unix.doc",ios::in):  // define an object
    cout << rfile << endl;                           // display file attributes
    rfile.chown(15, 30);                             // change UID and GID
    rfile.utime(0);                                  // touch time stamp
    rfile.link("/home/jon/hdlnk");                   // create a hard link
    rfile.remove();                                  // remove the old link
}
```

The *filebase* class defines generic functions for all POSIX and UNIX file types. However, it does not provide means for creating nonregular files or supporting file type-specific operations (e.g., file locking). To remedy these drawbacks, the following sections describe new subclasses of *filebase* class which provide complete data encapsulation for different UNIX and POSIX file types.

7.8 *Regfile* Class for Regular Files

The *filebase* class encapsulates most of the properties and functions needed to represent regular file objects in POSIX and UNIX systems except file locking. The *regfile* class is defined as a subclass of *filebase*, but also contains file locking functions. Thus, objects of this *regfile* class can do all regular file operations permitted in POSIX and UNIX systems.

The *regfile* class is defined as follows:

```
#ifndef REGFILE_H                 /* This is regfile.h header */
#define REGFILE_H
#include "filebase.h"
/* A class to encapsulate POSIX and UNIX regular file objects' properties */
class regfile  :  public filebase
{
    public:
        regfile ( const char* fn, int mode, int prot ) : filebase( fn, mode, prot )
                                    {};
        ~regfile()                  {};
        int lock( int lck_type, off_t len, int cmd = F_SETLK )
                                    {      struct flock flck;
                                           if (( lck_type&ios::in ) == ios::in )
                                                flck.l_type = F_RDLCK;
                                           else if ((lck_type &
                                                    ios::ou t ) ==ios::out )
                                                    flck.l_type = F_WRLCK;
                                            else return -1;
                                            flck.l_whence = SEEK_CUR;
                                            flck.l_start = (off_t)0;
                                            flck.l_len = len;
                                            return fcntl( fileno(), cmd, &flck );
                                    };
        int lockw( int lck_type, off_t len )
                                    {       return
                                              lock( lck_type, len, F_SETLKW );
                                    };
        int unlock( off_t len )     {       struct flock flck;
                                            flck.l_type = F_UNLCK;
                                            flck.l_whence = SEEK_CUR;
                                            flck.l_start = (off_t)0;
                                            flck.l_len = len;
                                            return fcntl(fileno(),F_SETLK,&flck );
                                    };
        int getlock( int lck_type, off_t len, struct flock& flck )
                                    {       if ( ( lck_type&ios::in ) == ios::in )
                                                    flck.l_type = F_RDLCK;
                                            else if (( lck_type &
                                                           ios::out)==ios::out )
                                                    flck.l_type = F_WRLCK;
                                            else return -1;
                                            flck.l_whence = SEEK_CUR;
                                            flck.l_start = (off_t)0;
                                            flck.l_len = len;
                                            return fcntl(fileno(),F_GETLK,&flck );
                                    };
};
#endif                  /* regfile.h */
```

The *regfile::lock* function can set read lock (*lck_type* is *ios::in*) or write lock (*lck_type* is *ios::out*) on a region of a file associated with a *regfile* object. The starting address of a lock region is the current file pointer of a file and can be set by the *fstream::seekg* function. The size of a lock region, in number of bytes, is specified via the *len* argument. The *regfile::lock* function is nonblocking by default. If users specify the *cmd* argument as F_SETLKW or if it is called from the *regfile::lockw* function, the function is blocking. The return value of *regfile::lock* is the same as that of *fcntl* in file locking operation.

The *regfile::lockw* function is a wrapper over the *regfile::lock* function, and it locks files in blocking mode. The return value of *regfile::lockw* function is the same as that of the *regfile::lock* function.

The *regfile::unlock* function unlocks a region of a file associated with a *regfile* object. The starting address of an unlock region is the current file pointer of a file and can be set by the *fstream::seekg* function. The size of an unlock region, in number of bytes, is specified via the *len* argument. The *regfile::unlock* function is nonblocking and its return value is the same as that of *fcntl* in file unlocking mode.

The *regfile::getlock* function queries lock information for a region of a file associated with a *regfile* object. The *lck_type* argument specifies whether read lock (*lck_type* is *ios::in*) or write lock (*lck_type* is *ios::out*) information is sought. The starting address of a region is the current file pointer of a file and can be set by the *fstream::seekg* function. The size of the lock region to be queried, in number of bytes, is specified via the *len* argument. The *regfile::getlock* function is nonblocking, and its return value is the same as that of *fcntl*. If a *regfile::getlock* function call succeeds, the *flck* argument will contain the lock information of a file region.

The *regfile* class provides a complete encapsulation of all UNIX and POSIX regular files functions. It also simplifies the file locking and unlocking APIs such that users need to know only the *fstream* class interface to use the *regfile::lock, regfile::unlock,* and *regfile::getlock* functions. The following *test_regfile.C* program illustrates how to use a *regfile* object to create a temporary file called *foo*, locks the entire file for write, initializes its content with data from the */etc/passwd* file, unlocks the first 10 bytes of the file, and then removes the file:

```
#include "regfile.h"
int main()
{                                              // Example for regfile
   ifstream ifs ("/etc/passwd");
   charbuf[256];
   regfilerfile("foo",ios::out | ios::in);     // define a regfile object
   rfile.lock(ios::out,0);                     // set write lock for entire file
   while (ifs.getline(buf,256)) rfile << buf << endl;
   rfile.seekg(0,ios::beg);                    // set file pointer to beginning of file
   rfile.unlock(10);                           // unlock the first ten byte of file
   rfile.remove();                             // remove the file
}
```

7.9 *dirfile* Class for Directory Files

The *dirfile* class is defined to encapsulate all UNIX and POSIX directory file functions. Specifically, the *dirfile* class defines the *create, open, read, tellg, seekg, close,* and *remove* functions that use the UNIX and POSIX directory file-specific APIs.

The *dirfile* class definition is:

```
#ifndef DIRFILE_H                   /* This is dirfile.h header */
#define DIRFILE_H
#include <dirent.h>
#include  <string.h>
class dirfile
{
    DIR      *dir_Ptr;
    char     *filename;
  public:
    dirfile( const char* fn )
                                {     dir_Ptr = opendir( fn );
                                      filename = strdup(fn);
                                };
    ~dirfile()                  {     if (dir_Ptr) close();
                                      delete(filename);
                                };
    int close()                 {     if (dir_Ptr) closedir( dir_Pt r);     };
    int create( const char* fn, mode_t prot )
                                {     return mkdir( fn, prot );             };
    int open( const char* fn )  {     dir_Ptr=opendir( fn );
                                      return dir_Ptr ? 0 : -1;
                                };
    int read( char* buf, int size )
                                {
                                      struct dirent *dp = readdir( dir_Ptr );
                                      if (dp)
                                          strncpy( buf, dp->d_name,size );
                                      return dp ? strlen( dp->d_name ) : -1;
                                };
    off_t tellg()               {     return telldir( dir_Ptr );            };
    void seekg( streampos ofs, seek_dir d )
                                {     seekdir( dir_Ptr, ofs );              };
    int remove()                {     return rmdir( filename );             };
};
#endif                    /* dirfile.h */
```

The *dirfile* class uses the *mkdir* API to create directory files. Furthermore, it uses the *opendir* and *readdir* API to open and read directory files, respectively. To users, a *dirfile* object can be treated similarly to a regular file object. The only difference is that a *dirfile* object does nothing for write operations. The *dirfile::tellg* and *dirfile::seekg* functions use the UNIX-specific *telldir* and *seekdir* APIs to support random access of any entry in directory files. Finally, the *dirfile::close* function uses *closedir* to close a directory file, and the *dirfile::remove* function uses the *rmdir* API to remove a directory file from a file system.

The following *test_dirfile.C* program creates the */usr/lck/dir.ex* directory using the *dirfile* class. It then opens the */etc* directory and echoes all files in that directory:

```
#include "dirfile.h"
int main()
{                                               // Example for dirfile
    dirfile ndir, edir( "/etc" );               // create a dirfile object to /etc
    ndir.create( "/usr/lck/dir.ex" );           // create /usr/lck/dir.ex
    char buf[ 256 ];
    while ( edir.read( buf, 256 ) )             // echo files in the /etc dir
       cout << buf << end;
    edir.close();                               // close a directory file
}
```

7.10 FIFO File Class

A FIFO file object differs from a *filebase* object in that a FIFO file is created, and the *tellg* and *seekg* functions are invalid for FIFO file objects. The following *pipefile* class encapsulates all the FIFO file type properties:

```
#ifndef PIPEFILE_H    /* This is pipefile.h */
#define PIPEFILE_H
#include "filebase.h"

/* A class to encapsulate POSIX and UNIX FIFO file objects' properties */
class pipefile :  public filebase
{
   public:
      pipefile( const char* fn, int flags, int prot ) : filebase( fn, flags, prot )
                                {};
      int create( const char* fn, mode_t prot )
                                {    return mkfifo( fn, prot );         };
      streampos tellg()         {    return ( streampos ) - 1;          };
};
#endif            /* pipefile.h */
```

The following *test_pipefile.C* program creates a FIFO file called *FIFO*, opens it for read (if *argc* is 1) or write (if *argc* is greater than 1). It reads or writes data via the FIFO file, then closes the FIFO file:

```
#include "pipefile.h"

int main( int argc, char* argv[] )
{                                                   // Example for pipefile
    pipefile nfifo( "FIFO", argc==1 ? ios::in : ios::out, 0755 );
    if ( argc > 1 )     {                           // writer process
            cout << "writer process write: " << argv[1] << endl;
            nfifo.write( argv[1],strlen(argv[1])+1 ); // write data to FIFO
    } else {                                        // reader process
            char buf[256];
            nfifo.read(buf,256);                    //read data from FIFO
            cout << "read from FIFO: " << buf << endl;
    }
    nfifo.close();                                  // close FIFO file
}
```

The program can be run twice to create two processes that communicate through the FIFO file. Specifically, one program is run with no command line argument and creates a reader process. Then the program is run again with a command line argument that creates a writer process. When both processes are created, the writer process will write its command line argument to the FIFO file, and the reader process will read that argument from the FIFO file and echo it to standard output. The following is a sample execution of the program, assuming that the program has been compiled to a file called *test_pipefile*:

```
%    CC -o test_pipefile test_pipefile.C
%    test_pipefile &                # create a reader process
%    est_pipefile "hello"           # create a writer process
writer process write: hello         # output from the writer process
read from FIFO: hello               # output from the reader process
```

7.11 Device File Class

A device file object has most of the properties of a regular file object except in the way that the device file object is created. Also the *tellg*, *seekg*, *lock*, *lockw*, *unlock*, and *getlock* functions are invalid for any character-based device file objects. The following *devfile* class encapsulates all the UNIX device files' properties:

```
#ifndef DEVFILE_H              /* This is devfile.h header */
#define DEVFILE_H
#include "regfile.h"
```

```
class devfile  :  public regfile
{
   public:
      devfile( const char* fn, int flags, int prot) : regfile(fn,flags,prot )   {};
      int create( const char* fn, mode_t prot, int major_no, int minor_no,
                               char type='c' )
      {      if (type=='c')
                   return mknod( fn, S_IFCHR | prot,
                                  (major_no << 8) | minor_no );
             else  return mknod( fn, S_IFBLK | prot,
                                  (major_no << 8) | minor_no );
      };
      streampos tellg()            {     if (file_type()==CHAR_FILE)
                                              return (streampos)-1;
                                         else return fstream::tellg();
                                   };
      istream seekg( streampos ofs, seek_dir d )
                                   {     if (file_type()!=CHAR_FILE)
                                              fstream::seekg( ofs,d );
                                         return *this;
                                   };
      int lock( int lck_type, off_t len, int cmd=F_SETLK )
                                   {     if (file_type()!=CHAR_FILE)
                                            return
                                              regfile::lock( lck_type,len,cmd );
                                         else return -1;
                                   };
      int lockw( int lck_type, off_t len )
                                   {     if (file_type()!=CHAR_FILE)
                                            return
                                              regfile::lockw( lck_type,len );
                                         else return -1;
                                   };
      int unlock( off_t len )      {     if (file_type()!=CHAR_FILE)
                                            return regfile::unlock( len );
                                         else return -1;
                                   };
      int getlock( int lck_type, off_t len,, struct flock& flck )
                                   {     if (file_type()!=CHAR_FILE)
                                            return
                                            regfile::getlock( lck_type,len,flck );
                                         else return -1;
                                   };
};
#endif                  /* devfile.h */
```