The *devfile::create* function creates a device file via the *mknod* API. The *type* argument specifies that a character (*type* value is *c*) or a block (*type* value is *b*) device file is to be created. The major and minor device numbers, access permission, and the file name of a device file are also specified via the *major_no, minor_no, prot,* and *fn* arguments. The return value of the *devfile::create* function is that of the *mknod* API.

After a device file is created, it can be opened, read, written to, or closed like regular files. Thus, *devfile* inherits all the *fstream* functions for data access to its objects. However, random data access is illegal for character device files. Thus, the *devfile::tellg* and *devfile::seekg* functions work for block device files only.

The following *test_devfile.C* program creates a character device file called */dev/tty* with major and minor device numbers of 30 and 15, respectively. It then opens it for write., writes data to it, and, finally, closes the file.

```
#include "devfile.h"
int main()
{                                          // Example for devfile
    devfile ndev]("/dev/tty", ios::out,0777); // open the device file for write
    ndev << "This is a sample output string\n";// write data to the file
    ndev.close();                          // close the device file
}
```

7.12 Symbolic Link File Class

A symbolic link file object differs from a *filebase* object in the way it is created. Also, a new member function called *ref_path* is provided to depict the path name of a file to which the symbolic link object refers. The following *symfile* class encapsulates all UNIX symbolic link file type properties:

```
#ifndef SYMFILE_H              /* This is symfile.h header */
#define SYMFILE_H
#include "filebase.h"
/* A class to encapsulate UNIX symbolic link file objects' properties */
class symfile : public filebase
{
  public:
    symfil()                   {};
    ~symfile()                 {};
```

```
        int setlink( const char* old_link, const char* new_link )
                                    {     filename=new
                                                 char[strlen(new_link)+1];
                                          strcpy(filename,new_link);
                                          return symlink(old_link,new_link);
                                    };
        void open( int mode )       {     fstream::open(filename,mode);  };
        const char* ref_path()      {     static char buf[256];
                                          if (readlink(filename,buf,256))
                                               return buf;
                                          else return (char*)-1;
                                    };
};
#endif                  /* symfile.h */
```

The *symfile::create* function creates a symbolic link file via the *symlink* API. The *new_link* argument is a new symbolic link file path name to be created, and the *old_link* is the path name of the original link. The return value of the *symfile::create* function is that of the *symlink* API.

After a symbolic link file is created, it can be opened, read, written to, or closed like any regular files. Thus, *symfile* inherits all the *fstream* functions for data access to its objects. However, all these operations occur on the nonlink file to which the symbolic link refers. Furthermore, the *symfile::ref_path* function is defined for users to query the path name of a nonlink file to which a *symfile* object references.

The following *test_symfile.C* program creates a symbolic link file call */usr/xyz/sym.lnk* that references a file called */usr/file/chap10.* It opens the symbolic link file and reads it content. Then, it echoes the link reference path name, and closes the file:

```
#include "symfile.h"
int main()
{                                                   // Examnle for symfile
     char          buf[256];
     symfile nsym;
     nsym.setlink("/usr/file/chap10","/usr/xyz/sym.lnk");
     nsym.open( ios:in );
     while(nsym.getline(buf,256))
          cout << buf << endl;                      // read /usr/file/chap10
     cout << nsym.ref_path() << endl;               // echo "/usr/file/chap10"
     nsym.close();                                  // close the symbolic link file
}
```

7.13 File Listing Program

An an example to illustrate the use of the *filebase* class and its subclasses, the following *lstdir.C* program reimplements the UNIX *ls* command to list file attributes of all path name arguments specified for the program. Furthermore, if an argument is a directory the program will list the file attributes of all files in that directory and any subdirectories underneath it. If a file is a symbolic link, the program will echo the path name to which the link refers. Thus, the program behaves like the UNIX *ls -lR* command.

```
#include "filebase.h"
#include "symfile.h"
#include "dirfile.h"

void show_list( ostream& ofs, const char* fname, int deep);

/* this is defined in test_ls.C. Section 7.1.10 */
extern void long_list( ostream& ofs, char* fn );

/* program to implement the UNIX ls -lR command */
void show_dir( ostream& ofs, const char* fname )
{
      dirfile       dirObj(fname);
      char          buf[256];
      ofs << "\nDirectory: " << fname << ":\n";
      while (dirObj.read(buf,256))            // show all files in a directory
                    show_list(ofs, buf,0);
      dirObj.seekg(0);                         // reset file pointer to beginning
      while (dirObj.read(buf,256))    {        // Look for directory files
            filebase fObj(buf,ios::in,0755);   // define a filebase object
            if (fObj.file_type==DIR_FILE)      // show sub-dir info
                    show_dir(ofs,buf);
            fObj.close();
      }
      dirObj.close();
}

void show_list( ostream& ofs, const char* fname, int deep)
{
      long_list( ofs, fname);
```

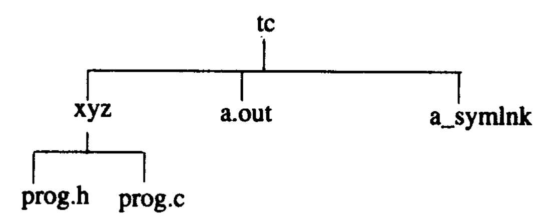
```
        filebase    fobj(fname,ios::in,0755);
        if (fobj.file_type()==SYM_FILE)    {        // symbolic link file
           symfile  *symObj = (symfile*)fobj;       // define a symfile object
           ofs << " -> " << symObj->ref_path() << endl;// show reference path
        }
        else if (fobj.file_type() == DIR_FILE && deep)// directory file
           show_dir( ofs, fname );                  // show directory content
    }

    int main( int argc, char* argv[])
    {
        while (--argc > 0) show_list( cout , *++argv, 1);
        return 0;
    }
```

The above program is similar to the program in Section 7.1.10, except that it echoes symbolic link references and lists directory contents recursively. The *main* function processes each command line argument by calling the *show_file* function to display file attributes of each argument to the standard output. The *deep* argument of *show_file* specifies, if its value is nonzero, that when a *fname* argument is a directory file, *show_file* should call the *show_dir* function to list the directory content. When *show_file* is called from *main*, the actual value of *deep* is set to 1.

The *show_dir* function is called to display the content of a directory file. It first creates a *dirfile* object to be associated with a directory whose name is specified by *fname*. It then calls *show_file* to list the file attributes of each file in that directory. Note that in this first pass, *show_file* is called with the actual argument of *deep* set to zero, so that *show_file* will not traverse any subdirectory by calling *show_dir*. After the first pass has completed, *show_dir* will scan a given directory a second time and looks for subdirectory files. For each subdirectory file found, *show_dir* will call itself recursively to list the content of the sub-directory. This is the same behavior as the UNIX *ls -lR* command.

For example, look at the following directory structure:



Assuming the above program has been compiled to an executable file called *lstdir*, the following commands and their expected output are:

```
%          CC -o lstdir lstdir.C
%          lstdir     tc
-r-x-r-xr--x    1  util      class       1234 Dec 8, 1993  a.out
prwxrwxrwx  1  util      class       122   Apr 11, 1994 a_symlink
-> /usr/dsg/unix.lnk
drwxr-x--x     1  util      class       234   Jan 17, 1994 xyz

Directry: xyz
-rw-r--r--       1  util      class       814   Dec 18, 1993prog.c
prwxrwxrwx  1  util      class       112   May 21, 1994prog.h
```

7.14 Summary

This chapter depicts the UNIX and POSIX file APIs. These APIs are used to create, open, read, write, and close all types of files in a system: regular, directory, device, FIFO, and symbolic link files. Furthermore, a set of C++ classes are defined to encapsulate the properties and functions of all file types, so that users can use these classes to manipulate files with the same interface as the *iostream* class. These classes are portable on all UNIX and POSIX systems, except for the symbolic link file class (*symfile*), which is not yet defined in the POSIX.1 standard

The inheritance hierarchy of all the file classes defined in the chapter is:

$$\begin{tikzcd}
 & \text{iostream} \arrow[d] & & \\
 & \text{fstream} \arrow[d] & & \\
 & \text{filebase} \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
\text{regfile} \arrow[d] & \text{dirfile} & \text{pipefile} & \text{symfile} \\
\text{devfile} & & &
\end{tikzcd}$$

File objects are manipulated by processes in an operating system. The UNIX and POSIX APIs for process creation and control are described in the next chapter.

UNIX Processes

A process is a program (e.g., *a.out*) under execution in a UNIX or POSIX system. For example, a UNIX shell is a process that is created when a user logs on to a system. Moreover, when a user enters a command *cat foo* to a shell prompt, the shell creates a new process. In UNIX terminology, this is a child process, which executes the *cat* command on behalf of the user. When a process creates a *child process*, it becomes the parent process of the child. The child process inherits many attributes from its parent process, and it is scheduled by the UNIX kernel to run independently from its parent.

By being able to create multiple processes that run concurrently, an operating system can serve multiple users and perform multiple tasks concurrently. Thus, process creation and management are the cornerstone of a multiuser and multitasking operating system such as UNIX. Furthermore, the advantages of allowing any process to create new processes in its course of execution are:

1. Any user can create multitasking applications.

2. Because a child process executes in its own virtual address space, its success or failure in execution will not affect its parent. A parent process can also query the exit status and run-time statistics of its child process after it has terminated.

3. It is very common for a process to create a child process that will execute a new program (e.g., the *spell* program). This allows users to write programs that can call on any other program to extend their functionality without the need to incorporate any new source code.

This chapter will explain the UNIX kernel data structures that support process creation and execution, the system call interface for process management, and a set of examples to demonstrate multitasking programs in UNIX.

8.1 UNIX Kernel Support for Processes

The data structure and execution of processes are dependent on operating system implementation. In the following, the process data structure and operating system support in the UNIX System V will be described as an illustration.

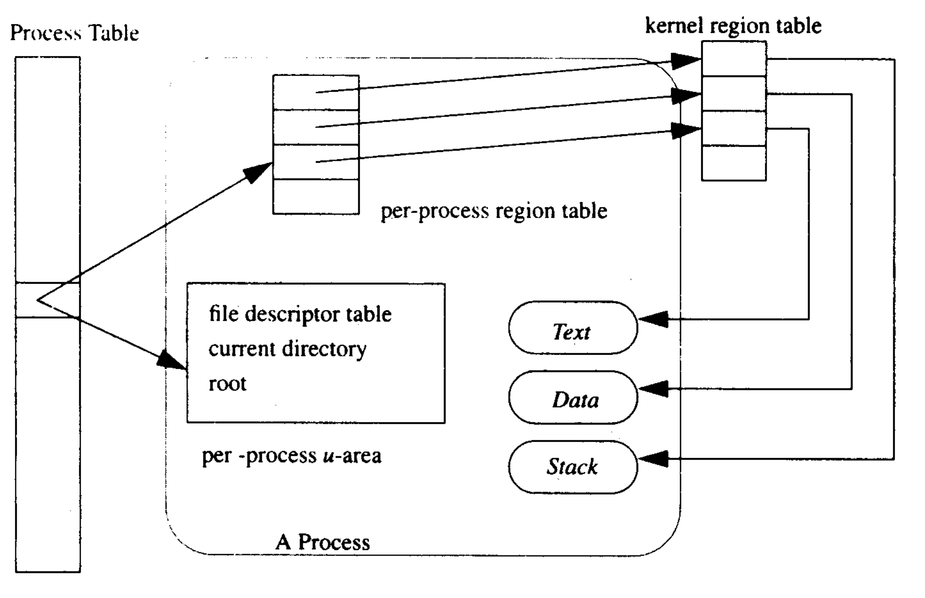


Figure 8.1 A UNIX process data structure

As shown in Figure 8.1, a UNIX process consists minimally of a text segment, a data segment, and a stack segment. A segment is an area of memory that is managed by the system as a unit. A text segment contains the program text of a process in machine-executable instruction code format. A data segment contains static and global variables and their corresponding data. A stack segment contains a run-time stack. A stack provides storage for function arguments, automatic variables, and return addresses of all active functions for a process at any time.

A UNIX kernel has a Process Table that keeps track of all active processes. Some of the processes belong to the kernel. They are called *system processes*. The majority of processes are associated with the users who are logged in. Each entry in the Process Table contains pointers to the text, data, stack segments and the *U*-area of a process. The *U*-area is an extension of a Process Table entry and contains other process-specific data, such as the file descriptor table, current root and working directory inode numbers, and a set of system-imposed process resource limits, etc.

All processes in a UNIX system, except the very first process (process 0) which is created by the system boot code, are created via the *fork* system call. After a *fork* system call, both the parent and child processes resume execution at the return of the *fork* function.

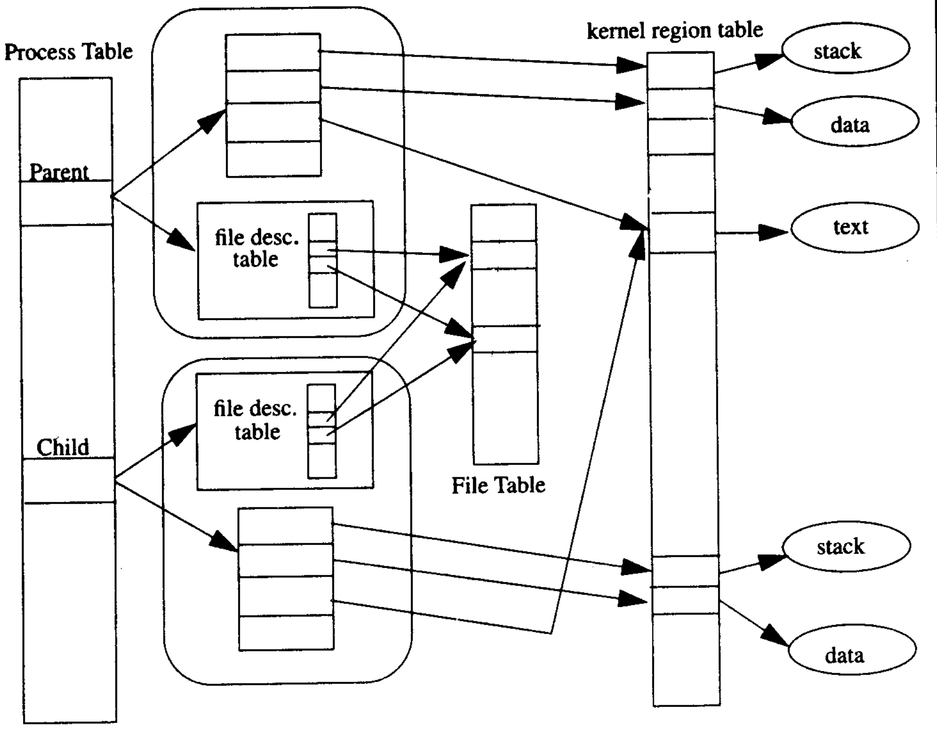


Figure 8.2 Data Structure of Parent and Chid Processes after Fork

As shown in Figure 8.2, when a process is created by *fork*, it contains duplicated copies of the text, data, and stack segments of its parent. Also, it has a file descriptor table that contains references to the same opened files as its parent, such that they both share the same file pointer to each opened file. Furthermore, the process is assigned the following attributes which are either inherited from its parent or set by the kernel:

- A real user identification number (rUID): the user ID of a user who created the parent process. This is used by the kernel to keep track of who creates which processes on a system

- A real group identification number (rGID): the group ID of a user who created the parent process. This is used by the kernel to keep track of which group creates which processes on a system
- An effective user identification number (eUID): this is normally the same as the real UID, except when the file that was executed to create the process has its *set-UID* flag turned on (via the *chmod* command or API). In that case, the process eUID will take on the UID of the file. This allows the process to access and create files with the same privileges as the program file owner
- An effective group identification number (eGID): this is normally the same as the real GID, except when the file which was executed to create the process has its *set-GID* flag turned on (via the *chmod* command or API). In that case, the process eGID will take on the GID of the file This allows the process to access and create files with the same privileges as the group to which the program file belongs
- *Saved set-UID* and *saved set-GID*: these are the assigned eUID and eGID, respectively, of the process
- Process group identification number (PGID) and session identification number (SID): these identify the process group and session of which the process is member
- Supplementary group identification numbers: this is a set of additional group IDs for a user who created the process
- Current directory: this is the reference (inode number) to a working directory file
- Root directory: this is the reference (inode number) to a root directory file
- Signal handling: the signal handling settings. See the next chapter for an explanation of signals
- Signal mask: a signal mask that specifies which signals are to be blocked
- Umask: a file mode mask that is used in creation of files to specify which accession rights should be taken out
- Nice value: the process scheduling priority value
- Controlling terminal: the controlling terminal of the process

In addition to the above attributes, the following attributes are different between the parent and child processes:

- Process identification number (PID): an integer identification number that is unique per process in an entire operating system
- Parent process identification number (PPID): the parent process PID
- Pending signals: the set of signals that are pending delivery to the parent process. This is reset to none in the child process
- Alarm clock time: the process alarm clock time (as set by the *alarm* system call) is reset to zero in the child process
- File locks: the set of file locks owned by the parent process is not inherited by the child process

After *fork*, a parent process may choose to suspend its execution until its child process terminates by calling the *wait* or *waitpid* system call, or it may continue execution independently of its child process. In the latter case, the parent process may use the *signal* or *sigaction* function (as described in Chapter 9) to detect or ignore the child process termination.

A process terminates its execution by calling the *_exit* system call. The argument to the *_exit* call is the exit status code of the process. By convention, an exit status code of zero means that the process has completed its execution successfully, and any nonzero exit code indicates failure has occurred.

A process can execute a different program by calling the *exec* system call. If the call succeeds, the kernel will replace the process's existing text, data, and stack segments with a new set that represents the new program to be executed. However, the process is still the same process (the process ID and parent process ID are the same), and its file descriptor table and opened directory streams remain mostly the same (except that those file descriptors which have their *close-on-exec* flag set via the *fcntl* system call will be closed upon *exec*'ing). Thus, calling *exec* is like a person changing jobs. After the change, the person still has the same name and personal identifications, but is now working on a different job than before.

When the *exec*'ed program completes its execution, it terminates the process. The exit status code of the program may be polled by the process's parent via the *wait* or *waitpid* function.

fork and *exec* are commonly used together to spawn a subprocess to execute a different program. For example, an UNIX shell executes each user command by calling *fork* and *exec* to execute the requested command in a child process. The advantages of this method are:

- A process can create multiple processes to execute multiple programs concurrently
- Because each child process executes in its own virtual address space, the parent process is not affected by the execution status of its child process

Two or more related processes (parent to child, or child to child with the same parent) may communicate with others by setting up unnamed pipes among them. For unrelated processes, they can communicate using named pipe or interprocess communication methods, as described in Chapter 10.

8.2 Process APIs

8.2.1 fork, vfork

The *fork* system call is used to create a child process. The function prototype of *fork* is:

```
#ifdef _POSIX_SOURCE
#include <sys/stdtypes.h>
#else
#include <sys/types.h>
#endif

pid_t          fork ( void );
```

The *fork* function takes no arguments, and it returns a value of type *pid_t* (defined in <sys/types.h>). The result of the call may be one of the following:

- The call succeeds. A child process is created, and the function returns the child process ID to the parent. The child process receives a zero return value from *fork*
- The call fails. No child process is created, and the function sets *errno* with an error code and returns a -1 value

The common causes of *fork* failure and the corresponding *errno* values are:

***Errno* value**	**Meaning**
ENOMEM	There is insufficient memory to create the new process
EAGAIN	The number of processes currently existing in a system exceeds a system-imposed limit, so try the call again later

There are system-turnable limits on the maximum number of processes that can be created by a single user (CHILD_MAX) and the maximum number of processes that can exist concurrently system-wide (MAXPID). If either of these limits is exceeded when *fork* is called, the function will return a failure status. The MAXPID and CHILD_MAX symbols are defined in the <sys/param.h> and <limits.h> headers, respectively. Furthermore, a process may obtain the CHILD_MAX value via the *sysconf* function:

```
int child_max = sysconf ( _SC_CHILD_MAX );
```

If a *fork* call succeeds, a child process is created. The data structure of the parent and child processes after *fork* are shown in Figure 8.2. Both the child and the parent process will be scheduled by the UNIX kernel to run independently, and the order of which process will run first is implementation-dependent. Furthermore, both processes will resume their execution at the return of the *fork* call. After the *fork* call, the return value is used to distinguish whether a process is the parent or the child. In this way, the parent and child processes can do different tasks concurrently.

The following *test_fork.C* program illustrates the use of *fork*. The parent process invokes *fork* to create a child process. If *fork* returns -1, the system call fails, and the parent process calls *perror* to print a diagnostic message to the standard error. On the other hand, if *fork* succeeds, the child process, when executed, will print the message *Child process created* to the standard output. It then terminates itself via the *return* statement. Meanwhile, the parent will print the message *Parent process after fork* and will then quit.

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
     pid_t    child_pid;
     cout << "PID: " << getpid() << ", parent: " << getppid() << endl;
     switch (chidl_pid=fork()){
        case (pid_t)-1:
                perror("fork");              /* fork fails */
                break;
        case (pid_t)0:
                cout << "Child created: PID: " << getpid()
                     << ", parent: " << getppid() << endl;
                exit(0);
        default: cout << "Parent after fork. PID: " << getpid()
                     << ", child PID: " << child_pid << endl;
     }
     return 0;
}
```

The sample outputs of this program, when executed, may be:

```
%    CC -o test_fork test_fork.C
%    test_fork
PID: 234, parent: 123
Child created: PID: 645, parent: 234
Parent after fork. PID: 234, child PID: 645
```

An alternative API to *fork* is *vfork*, which has the same signature as does *fork*:

```
pid_t    vfork    (void);
```

vfork has the similar function as *fork*, and it returns the same possible values as does *fork*. It is available in BSD UNIX and System V.4. However, it is not a POSIX.1 standard. The idea of *vfork* is that many programs call *exec* (in child processes) right after *fork*. Thus, it will improve the system efficiency if the kernel does not create a separate virtual address space for the child process until *exec* is executed. This is what happens in *vfork*: After the function is called, the kernel suspends the execution of the parent process while the child process is executing in the parent's virtual address space. When the child process calls *exec* or *_exit*, the parent will resume execution, and the child process will either get its own virtual address space after *exec* or will terminate via the *_exit* call.

vfork is unsafe to use, because if the child process modifies any data of the parent (e.g., closes files or modifies variables) before it calls *exec* or *_exit*, those changes will remain when the parent process resumes execution. This may cause unexpected behavior in the parent. Furthermore, the child should not call *exit* or return to any calling function, because this will cause the parent's stream files being closed or modify the parent run-time stack, respectively.

The latest UNIX systems (e.g., System V.4) have improved on the efficiency of *fork* by allowing parent and child processes to share a common virtual address space until the child calls either the *exec* or *_exit* function. If either the parent or the child modifies any data in the shared virtual address space, the kernel will create new memory pages that cover the virtual address space modified. Thus, the process that made changes will reference the new memory pages with the modified data, whereas the counterpart process will continue referencing the old memory pages. This process is called *copy-on-write*, and it renders *fork* execution efficiency comparable to that of *vfork*. Thus, *vfork* should be used in porting old applications to the new UNIX systems only.

8.2.2 _exit

The *_exit* system call terminates a process. Specifically, the API will cause the calling process data segment, stack segments, and *U*-area to be deallocated and all the open file descriptors to be closed. However, the Process Table slot entry for this process is still intact so that the process exit status and its execution statistics (e.g., total execution time, number of I/O blocks transferred, etc.) are recorded therein. The process is now called a *zombie process*, as it can no longer be scheduled to run. The data stored in the Process Table entry can be retrieved by the process parent via the *wait* or *waitpid* system call. These APIs will also deallocate the child Process Table entry.

If a process *forks* a child process and terminates before the child, the child process will be assigned by the kernel to be adopted by the *init* process (this is the second process created after a UNIX system is booted. Its process ID is always 1). When the child process terminates, its Process Table slot will be cleaned up by the *init process*.

The function prototype of the *_exit* function is:

```
#include <unistd.h>

void        _exit        (int exit_code);
```

The integer argument to *_exit* is a process exit status code. Only the lower 8 bits of the exit code are passed to a parent process. By convention, an exit status of zero indicates that the process terminated successfully, and a nonzero exit status indicates a failed termination. In some UNIX systems, the manifested constants EXIT_SUCCESS and EXIT_FAILURE are defined in the <stdio.h> header and can be used as actual arguments to *_exit* for the success and failure exit status values, respectively.

The *_exit* function never fails, and there is no return value.

The C library function *exit* is a wrapper over *_exit*. Specifically, *exit* will first flush and close all opened streams of the calling process. It will then call any functions that were registered via the *atexit* function (in an order reverse to that in which functions were registered via the *atexit* function) and, finally call *_exit* to terminate the process.

The following *test_exit.C* program illustrates the use of *_exit*. When the program is run, it declares its existence and then terminates itself via the *_exit* call. It passes a 0 exit status value to indicate that its execution has been completed successfully.

```
#include <iostream.h>
#include <unistd.h>
int main()
{
    cout << "Test program for _exit" << endl;
    _exit(0);
}
```

After this program is run, users can test the exit status of this program via the *status* (in C shell) or *$?* (in Bourne shell) shell variable. The output of this program may be:

```
%   CC -o test_exit test_exit.C ; test_exit
Test program for _exit
%   echo $status
0
```

8.2.3 wait, waitpid

The *wait* and *waitpid* system calls are used by a parent process to wait for its child process to terminate and to retrieve the child exit status (assigned by the child via *_exit*). Furthermore, these calls will deallocate the Process Table slot of the child process, so that the slot can be reused by a new process. The prototypes of these functions are:

```
#include <sys/wait.h>

pid_t     wait      (int *status_p);
pid_t     waitpid   (pid_t child_pid, int* status_p, int options);
```

The *wait* function will suspend the parent process until either a signal is sent to the process or one of its child processes terminates or is stopped (and its status has not yet been reported). If a child process has already terminated or has been stopped prior to a *wait* call, *wait* returns immediately with the child exit status (via *status_p*), and the function return value is the child PID. If, however, a parent has no unwaited-for child processes or if it is interrupted by a signal while executing *wait*, the function will return a -1 value, and *errno* will contain an error code. Note that if a parent process has spawned more than one child process, the *wait* call will wait for any one of these child processes to terminate.

The *waitpid* function is a more general function than is *wait*. Like *wait*, *waitpid* will collect a child process exit code and PID upon its termination. However, with *waitpid*, the caller has an option to specify which child process to wait for by specifying one of the following values for the *child_pid* argument:

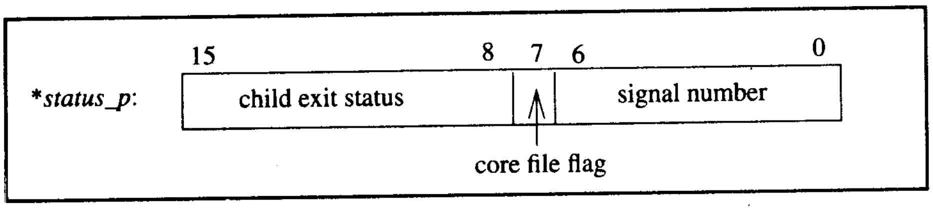
Actual value for *child_pid*	**Meaning**
A child process ID	Waits for the child with that PID
-1	Waits for any child
0	Waits for any child in the same process group as the parent
A negative value but not -1	Waits for any child whose process group ID is the absolute value of *child_pid*

Furthermore, the caller can direct *waitpid* to be either blocking or nonblocking and to wait for any child that is or is not stopped due to job control. These are specified via the *options* argument. Specifically, if the WNOHANG flag (defined in <sys/wait.h>) is set in an *options* value, the call will be nonblocking (that is, the function will return immediately if there is no child that satisfies the wait criteria). Otherwise, the call is blocking and the parent will be suspended as in a *wait* call. Furthermore, if the WNOTRACED flag is set in the *options* value, the function will also wait for a child that is stopped (but its status has not been reported before) due to job control.

If the actual value to a *status_p* argument of either a *wait* or *waitpid* call is NULL, no child exit status is to be queried. However, if the actual value is an address of an integer-typed variable, the function will assign an exit status code (specified via the *_exit* API) to this variable. The parent can then check the exit status code with the following macros as defined in <sys/wait.h>:

Macro	**Use**
WIFEXITED(*status_p)	Returns a nonzero value if a child was terminated via an *_exit* call, and zero otherwise
WEXITSTATUS(*status_p)	Returns a child exit code that was assigned to an *_exit* call. This should be called only if WIFEXITED returns a nonzero value
WIFSIGNALED(*status_p)	Returns a nonzero value if a child was terminated due to signal interruption
WTERMSIG(*status_p)	Returns the signal number that had terminated a child process. This should be called only if WIFSIGNALED returns a nonzero value
WIFSTOPPED(*status_p)	Returns a nonzero value if a child process has been stopped due to job control
WSTOPSIG(*status_p)	Returns the signal number that had stopped a child process. This should be called only if WIFSTOPPED returns a nonzero value

In some versions of UNIX, where the above macros are undefined, the above information can be obtained directly from **status_p*. Specifically, the seven least significant bits (bit 0 to bit 6) of **status_p* are zero if a child was terminated via *_exit* or a signal number that terminated the child. The eighth bit of **status_p* is 1 if a core file has been generated due to signal interruption of the child, or 0 otherwise. Furthermore, if the child was terminated via *_exit*, bit 8 to bit 15 of **status_p* is the child exit code that was passed via *_exit*. The following figure illustrates the use of the **status_p* data bits:



In BSD UNIX, the *status_p* argument is of type *union wait**, where *union wait* is defined in <sys/wait.h>. It is a union of an integer variable and a set of bit-fields. The bit-fields are used to extract the same status information as the above macros.

If the return value of either *wait* or *waitpid* is a positive integer value, it is the child PID. Otherwise, it is (*pid_t*)-1 and signifies that either no child satisfied the wait criteria or the function was interrupted by a caught signal. Here, *errno* may be assigned one of the following values:

***Errno* value**	**Meaning**
EINTR	*Wait* or *waitpid* returns because the system call was interrupted by a signal
ECHILD	For *wait*, it means the calling process has no unwaited-for child process
	For *waitpid*, it means either the *child_pid* value is illegal or the process cannot be in a state as defined by the *options* value
EFAULT	The *status_p* argument points to an illegal address
EINVAL	The *options* value is illegal

Both *wait* and *waitpid* are POSIX.1 standard. *waitpid* is not available in BSD UNIX 4.3, System V.3 and their older versions.

The following *test_waitpid.C* program illustrates use of the *waitpid* API:

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

int main()
{
    pid_t          child_pid, pid;
    int            status;
    switch (child_pid=fork()) {
      case (pid_t)-1:perror("fork");              /* fork fails */
                     break;
      case (pid_t)0:cout << "Child process created\n";
                     _exit(15);                   /* terminate child */
      default:       cout << "Parent process after fork\n";
```

```
                        pid = waitpid(child_pid,&status,WNOTRACED);
        }
        if WIFEXITED(status))
            cerr << child_pid << " exits: " <<WEXITSTATUS(status) << endl;
        else if WIFSTOPPED(status))
            cerr << child_pid << " stopped by: " <WSTOPSIG(status)
                 << endl;
        else if WIFSIGNALED(status))
            cerr << child_pid << " killed by: " <<WTERMSIG(status) << endl;
        else perror("waitpid");
        _exit(0);
    }
```

This simple program forks a child process that acknowledges its creation and then terminates with an exit status of 15. Meanwhile, the parent suspends its execution via the *waitpid* call. The parent process resumes execution after the child has terminated and the *status* and *child_pid* variables of the parent process are assigned the child's exit code and process ID. The parent uses the macros defined in <sys/wait.h> to determine the execution status of the child in the following order:

- If WIFEXITED returns a nonzero value, the child was terminated via the *_exit* call, and the parent extracts the child's exit code (which is 15 in this example) via the WEXITSTATUS macro. It then prints the value to standard error port

- If WIFEXITED returns a zero value and WIFSTOPPED returns a nonzero value, the child was stopped by a signal. The parent extracts the signal number via the WSTOPSIG macro and prints the value to standard error port

- If both WIFEXITED and WIFSTOPPED return a zero value and WIFSIGNALED returns a nonzero value, the child was terminated by an uncaught signal. The parent extracts the signal number via the WTERMSIG macro and prints the value to standard error port

- If WIFEXITED, WIFSTOPPED, and WIFSIGNALED all return a zero value, either the parent has no child processes or the *waitpid* call was interrupted by a signal. Thus, the parent calls the *perror* function to print detailed diagnostics for the failure

The output of this program may be:

```
%    CC -o test_waitpid test_waitpid.C
%    test_waitpid
```

```
Child process created
Parent process after fork
1354 exits: 15
%
```

8.2.4 exec

The *exec* system call causes a calling process to change its context and execute a different program. There are six versions of the *exec* system call. They all have the same function but they differ from each other in their argument lists.

The prototypes of the *exec* functions are:

```
#include <unistd.h>

int   execl     (const char* path, const char* arg, ...);
int   execlp    (const char* file, const char* arg, ...);
int   execle    (const char* path, const char* arg, ..., const char** env);
int   execv     (const char* path, const char** argv, ...);
int   execvp    (const char* file, const char** argv, ...);
int   execve    (const char* path, const char** argv, ..., const char **env);
```

The first argument to the function is either the path name or a file name of a program to be executed. If the call succeeds, the calling process instruction and data memory are overlaid with the new program instruction text and data. The process starts execution at the beginning of the new program. Furthermore, when the new program completes execution, the process is terminated and its exit code will be passed back to its parent process. Note that, whereas *fork* creates a child process that runs independently of its parent, *exec* does not create a new process, but rather it changes the calling process context to execute a different program.

An *exec* call may fail if the program to be executed cannot be accessed or has no execution rights. Furthermore, the program named in the first argument of an *exec* call should be an executable file (i.e., in *a.out* format). However, it is possible in UNIX to specify a shell script name to the *execlp* and *execvp* calls, so that the UNIX kernel will execute a Bourne shell (*/bin/sh*) to interpret the shell script. Because POSIX.1 does not have the notion of shells, it is illegal to use *execlp* or *execvp* to execute shell scripts. This is really not a problem, as users can always *exec* a shell and supply it with the name of a shell script that he or she wishes to execute.

The *p* suffix of *execlp* and *execvp* specifies that if the actual value of a *file* argument does not begin with a "/", the functions will use the shell PATH environment variable to search for the file to be executed. For all the other *exec* functions, the actual value of their first argument should be the path name of any file to be executed.

The *arg* or *argv* arguments are arguments for an *exec*'ed program. They are mapped to the *argv* variable of the *main* function of the new program. For the *execl, execlp,* and *execle* functions, the *arg* argument is mapped to *argv[0]*, the value after *arg* will be mapped to *argv[1]*, and so on. The argument list specified in the *exec* call must be terminated by a NULL value to tell the function where to stop looking for argument values. For the *execv, execvp,* and *execve* functions, the *argv* argument is a vector of character strings, each string being one argument value. The *argv* argument is mapped directly to the *argv* variable of the *main* function of the new program. Thus, the *l* character in an *exec* function name specifies that the argument values are listed in each call, whereas the *v* character in an *exec* function name signifies that the arguments are passed in a vector format.

Note that one must supply at least two argument values to each *exec* call. The first value (*arg* or *argv[0]*) is the name of a program to be *exec*'ed and is mapped to *argv[0]* of the *main* function of the new program. The second mandatory argument is the NULL value that terminates the argument list (for *execl, execlp* and *execle*) or the argument vectors (for *execv, execvp* and *execve*).

The *e* suffix of an *exec* function (*execle* or *execve*) specifies that the last argument (*env*) to a function call is a vector of character strings. Here, each string defines one environment variable and its value in a Bourne shell format:

```
<environment_variable_name>=<value>
```

The last entry of *env* must be a NULL value to signal the end of a vector list. In non-ANSI C environment, *env* will be assigned to the third parameter of the *main* function in the *exec*'ed program. In an ANSI C environment, the *main* function can have only two arguments (namely, *argc* and *argv*), and *env* will be mapped to the *environ* global variable of the *exec*'ed program. For the *execl, execlp, execv,* and *execvp* functions, the *environ* global variable is unchanged in the process by the *exec* call (note that the *environ* variable may be updated using the *putenv* function).

If an *exec* call succeeds, the original process text, data, and stack segments are replaced by new segments for an *exec*'ed program. However, the file descriptor table of the process remains unchanged. Those file descriptors whose *close-on-exec* flags were set (by the *fcntl* system call) will be closed before a new program runs. Furthermore, the following process attributes may be changed when the process executes an *exec*'ed program:

- Effective UID: this is changed if an *exec*'ed program file has its *set-UID* flag set
- Effective GID: this is changed if an *exec*'ed program file has its *set-GID* flag set
- Saved set-UID: this is changed if an *exec*'ed program file has its *set-UID* flag set

- Signal handling: signals that are set up to be caught in a process are reset to accept their default actions when the process *exec*'ed a new program. The user-defined signal handler functions are not present in the *exec*'ed program

Most programs call *exec* in a child process because it is desirable to continue a parent process execution after an *exec* call. However, *fork* and *exec* are implemented as two separate functions for the following reasons:

- It is simpler to implement *fork* and *exec* separately
- It is possible for a program to call *exec* without *fork*, or *fork* without *exec*. This renders flexibility in the use of these functions
- Many programs will do some operations in child processes, such as redirect standard I/O to files, before they call *exec*. This is made possible by separating the *fork* and *exec* APIs

The following *test_exec.C* program illustrates use of the *exec* API:

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int System( const char *cmd)          // emulate the C system function
{
    pid_t   pid;
    int     status;
    switch (pid=fork())      {
      case -1:      return -1;
      case 0:       execl("/bin/sh", "sh", "-c", cmd, 0);
                    perror("execl");
                    exit(errno);
    }
    if (waitpid(pid,&status,0)==pid && WIFEXITED(status))
      return WEXITSTATUS(status);
    return -1;
}
```

```
int main()
{
        int         rc = 0:
        char        buf[256];
        do  {
                    cout << "sh> " << flush;
                    if (!gets(buf)) break;
                    rc = System(buf);
        } while (!rc);
        exit(rc);
}
```

The above program is a simple UNIX shell program. It prompts users to enter shell commands from standard input and executes each command via the *System* function. The program terminates when either user enters end-of-file (<ctrl-D>) at a "shell" prompt or the return status of a *System* call is nonzero. The program differs from a UNIX shell in that its does not support the *cd* command and manipulation of shell variables.

The *system* function emulates the C library function *system*. Specifically, the *system* function prototype is:

```
int        system  (const char* cmd);
```

Both functions invoke a Bourne shell (*/bin/sh*) to interpret and execute a shell command that is specified via the argument *cmd*. A command may consist of a simple shell command or a series of shell commands separated by semicolons or pipes. Furthermore, input and/or output redirections may be specified with the commands.

The *System* function calls *fork* to create a child process. The child process, in turn, calls *execlp* to execute a Bourne shell program (*/bin/sh*) with the *-c* and *cmd* as arguments. The *-c* option instructs the Bourne shell to interpret and execute the *cmd* arguments as if they were entered at the shell level. After *cmd* is executed, the child process is terminated and the exit status of the Bourne shell is passed to the parent process, which calls the *System* function.

Note that the *System* function calls *waitpid* to specifically wait for the child that it forked. This is important, as the *System* function may be called by a process that forked a child process before calling *System*; thus, the *System* function would wait only for child processes forked by it and not those created by the calling process.

When the *waitpid* returns, the *System* function checks that: (1) the return PID matches that of the child process that it forked; and (2) the child was terminated via *_exit*. If both conditions are true, the *System* function returns the child exit code. Otherwise, it returns a -1 to indicate failure status.

The *system* library function is similar to the *System* function, except that the former will include signal handling and set the *errno* variable with an error code when the *waitpid* call fails.

The sample output of the program may be:

```
%    CC -o test_exec test_exec.C
%    test_exec
sh>  date; pwd
Sat Jan 15 18:09:53 PST 1994
/home/terry/sample
sh> echo "Hello world"  |  wc > foo;  cat foo
1   2   12
sh> ^D
```

8.2.5 pipe

The *pipe* system call creates a communication channel between two related processes (for example, between a parent process and a child process, or between two sibling processes with a same parent). Specifically, the function creates a pipe device file that serves as a temporary buffer for a calling process to read and write data with another process. The pipe device file has no assigned name in any file system; thus, it is called an *unnamed pipe*. A pipe is deallocated once all processes close their file descriptors referencing the pipe.

```
#include <unistd.h>

int          pipe    ( int      fifo[2] );
```

The *fifo* argument is an array of two integers that are assigned by the *pipe* API. On most UNIX systems, a pipe is unidirectional in that *fifo[0]* is a file descriptor that a process can use to read data from the pipe, and *fifo[1]* is a different file descriptor that a process can use to write data to the pipe. However, in UNIX System V.4, a pipe is bidirectional and both the *fifo[0]* and *fifo[1]* descriptors may be used for reading and writing data via the pipe. POSIX.1 supports both the traditional UNIX and System V.4 pipe models by not specifying the exact uses of the pipe descriptors. Applications that desire portability on all UNIX and POSIX systems should use pipes as if they were unidirectional only.

Data stored in a pipe is accessed sequentially in a first-in-first-out manner. A process cannot use *lseek* to do random data access of a pipe. Data is consumed from a pipe once it is read.

The common method is to set up a communication channel between processes via *pipe*:

- *Parent and child processes*: the parent calls *pipe* to create a pipe, then forks a child. Since the child has a copy of the parent file descriptors, the parent and child can communicate through the pipe via their respective *fifo[0]* and *fifo[1]* descriptors

- *Sibling child processes*: the parent calls *pipe* to create a pipe, then forks two or more child processes. The child processes can communicate through the pipe via their respective *fifo[0]* and *fifo[1]* descriptors

Because the buffer associated with a pipe device file has a finite size (PIPE_BUF), a pipe already filled with data when a process tries to write to it will be blocked by the kernel until another process reads sufficient data from the pipe to make room for the blocked process to succeed in the write operation. Conversely, if a pipe is empty and a process tries to read data from a pipe, it will be blocked until another process writes data into the pipe. These blocking mechanisms can be used to synchronize the execution of two (or more) processes.

There is no limit on how many processes can concurrently attach to either end of a pipe. However, if two or more processes are writing data to a pipe simultaneously, each process can write, at most, PIPE_BUF bytes of contiguous data into the pipe at a time. Consider that when a process (for example, *A*) writes *X* bytes of data into a pipe, there is no room for *Y* bytes in the pipe. If *X* is larger than *Y* only the first *Y* bytes of data are written into the pipe, and the process is blocked. Another process (for example, *B*) runs and there is room in the pipe (due to a third process reading data from the pipe), and *B* writes data into the pipe. Then, when process *A* resumes running, it writes the remaining *X-Y* bytes of data into the pipe. The end result is that data in the pipe is interlaced between the two processes. Similarly, if two (or more) processes attempt to read data from a pipe concurrently, it may happen that each process reads only a portion of the desired data from the pipe.

To avoid the above drawbacks, it is conventional to set up a pipe as unidirectional communication channel between only two processes, such that one process will be designated as the sender of the pipe and the other process designated as the receiver of the pipe. If two processes, fro example, *A* and *B*, need a bidirectional communication channel, they will create two pipes: one for process *A* to write data to process *B*, and vice versa.

If there is no file descriptor in the process to reference the write-end of a pipe, the pipe write-end is considered "close" and any process attempts to read data from the pipe will receive the remaining data. However, once all data in the pipe is consumed, a process that attempts to read more data from the pipe will receive an end-of-file (the *read* system call returns a 0 return value) indicator. On the other hand, if no file descriptor references the read-end of a pipe, and the process attempts to write data into the pipe, it will receive the SIGPIPE (broken pipe) signal from the kernel. This is because no data written to the pipe can be retrieved by the process; thus, the write operation is considered illegal. The process that does the write will be penalized by the signal (the default action of the signal is to abort the process).

pipe is used by the UNIX shell to implement the command pipe ("|") for connecting the standard output of one process to the standard input of another process. It is also used in implementation of the *popen* and *pclose* C library functions. The implementation of the *popen* and *pclose* functions is described in the next section.

The return value of *pipe* may be 0 if the call succeeds or -1 if it fails. The possible *errno* values assigned by the API and their meanings are:

***Errno* value**	**Meaning**
EFAULT	The fifo argument is illegal
ENFILE	The system file table is full

The following *test_pipe.C* program shows the use of *pipe*:

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main()
{
     pid_t child_pid;
     int   fifo[2], status;
     char  buf[80];

     if ( pipe(fifo) == -1 ) perror( "pipe" ), exit( 1 );
     switch ( child_pid = fork() ) {
        case -1:        perror( "fork" );
                        exit( 2 );
        case 0:         close( fifo[0] );            /* child process */
                        sprintf( buf, "Child %d executed\n", getpid() );
                        write( fifo[1], buf, strlen(buf)) ;
                        close( fifo[1] );
                        exit(0);
        }
        close( fifo[1 ] );                              /* parent process */
        while ( read( fifo[0], buf, 80) ) cout << buf << endl;
        close( fifo[0] );
        if ( waitpid(child_pid,&status,0)==child_pid && WIFEXITED(status) )
            return WEXITSTATUS( status );
        return 3;
}
```

This program shows a simple use of *pipe*. The parent process calls *pipe* to allocate a pipe device file. It then calls *fork* to create a child process. Both the parent and child processes can access the pipe via their own copy of the *fifo* variable.

In the example, the child process is designated as the sender of message to the parent, writing the message *Child <child_pid> executed* to the pipe via the *fifo[1]* descriptor. The *getpid* system call returns the child PID value. After the write, the child terminates via *exit* with a zero exit code.

The parent process is designated the receiver of the pipe, and it reads the child's message from the pipe via the *fifo[0]* descriptor. Note that in the parent process, it first closes the *fifo[1]* descriptor before it goes into a loop to read data from the pipe. This is to ensure that when the child process closes its *fifo[1]* descriptor (after a write), the write end of the pipe will be closed. The parent will eventually receive the end-of-file indicator after if has read all messages from the child process. If the parent does not close *fifo[1]* before it does the read loop, the parent will eventually be suspended in the *read* system call once it has read all data from the pipe (the pipe's write end is still opened as referenced by the parent's *fifo[1]*, and the end-of-file indicator will not be seen).

As a general rule, the reader process should always close the pipe write-end file descriptor before it reads data from the pipe. Similarly, the sender process should always close the pipe write-end descriptor after it finishes writing data to the pipe. This renders the reader process to detect the end-of-file situation.

After the parent process exits from the read loop, it calls *waitpid* to collect the child exit status and terminates with either the child exit code (if the child has terminated via *_exit*) or a failure exit code of 3.

The output of the program may be:

```
% CC test_pipe.C -o test_pipe;    test_pipe
Child 1234 executed
```

8.2.6 I/O Redirection

In UNIX, a process can use the C library function *freopen* to change its standard input and/or standard output ports to refer to text files instead of the console. For example, the following statements will change the process standard output to the file *foo*, so that the *printf* statement will write the message *Greeting message to foo* to the file:

```
FILE *fptr = freopen("foo","w",stdout);
printf("Greeting message to foo\n");
```

Similarly, the following statements will change the process standard input port to the file *foo*, dumping the entire file content to the standard output:

```
char buf[256];
FILE *fptr = freopen("foo","r",stdin);
while (gets(buf)) puts (buf);
```

The *freopen* function actually relies on the *open* and *dup2* system calls to do redirection of either standard input or output. Thus, to redirect the standard input of a process from the file *src_stream*, the following can be done:

```
#include <unistd.h>
int fd = open("src_stream",O_RDONLY);
if (fd!=-1) dup2(fd,STDIN_FILENO), close(fd);
```

The above statements first open the *src_stream* file for read-only, and the *fd* file description references the opened file. If the *open* call succeeds (*fd* value is not -1), the *dup2* function is called to force the STDIN_FILENO (which is defined in *<unistd.h>* header, and is the standard input file descriptor value) to reference the *src_stream* file, then the *fd* descriptor is discarded via the *close* system call. The result of all this is that the *src_stream* file is now referenced by the STDIN_FILENO descriptor of the process.

Similar system calls can also be used to change the standard output of a process to a file:

```
#include <unistd.h>
int fd = open("dest_stream",O_WRONLY | O_CREAT | O_TRUNC,0644);
if (fd!=-1) dup2(fd,STDOUT_FILENO), close(fd);
```

After the above statements, any data written to the process' standard output, via the STDOUT_FILENO, will be written to the file *dest-file*.

The *freopen* function can be implemented as follows:

```
FILE * freopen (const char* file_name, const char *mode,
                    FILE *old_fstream)
{
   if (strcmp(mode,"r") && strcmp(mode,"w"))
        return NULL;                          /* invalid mode */
   int fd = open(file_name,*mode=='r' ? O_RDONLY :
                        O_WRONLY|O_CREAT | O_TRUNC,0644);
   if (fd == -1) return NULL;
   if (!old_stream) return fdopen(fd, mode);
   fflush(old_fstream);
   int fd2 = dup2(fd, fileno(old_fstream));
   close(fd);
   return (fd2 == -1) ? NULL : old_fstream;
}
```

In the above function, if the *mode* argument value is not *"r"* or *"w"* the function returns a NULL stream pointer, as the function does not support other access modes. Furthermore, if the file named by the *file_name* argument cannot be opened with the specified mode, the function will also return a NULL stream pointer. If the *open* call succeeds and the *old_fstream* argument is NULL, there is no old stream to redirect. The function will just convert the *fd* file descriptor to a stream pointer via the *fdopen* function and return it to the caller.

If, however, the *old_fstream* is not NULL, the function will first flush all data stored in that stream's I/O buffer via the *fflush* function call, then it will use *dup2* to force the file descriptor associated with the *old_fstream* to refer to the opened file. Note that it is invalid to use *fclose* to flush and close the *old_fstream* here. *Freopen* needs to reuse the FILE record referenced by *old_fstream* for the new file, but *fclose* will deallocate the FILE record. The *fileno* macro is defined in the <stdio.h> header. It returns a file descriptor associated with a given stream pointer.

After *dup2*, the function closes *fd*, as it is no longer needed, and returns either the *old_fstream*, which now references the new file, or NULL, if the *dup2* call fails.

8.2.6.1 UNIX I/O Redirection

The UNIX shell input redirection (<) and output redirection (>) constructs can be implemented with the same concept as the above, except that the redirection operation will be done before a child process calls *execs* to a shell executing a user command. For example, the following program implements the UNIX shell command:

```
%    sort <  foo > results
```

The following program illustrates a mean for standard input and output redirection:

```
#include <unistd.h>
int main()
{
      int fd, fd2;
      switch ( fork() )  {
         case -1:      perror( "fork" ), break;

         case 0:       if ( (fd = open("foo", O_RDONLY))==-1 ||
                                 (fd2=open("results", O_WRONLY
                                 O_CREAT| O_TRUNC ,0644)) == -1 ) {
                            perror( "open" );
                            _exit( 1 );
                       }

                       /* set standard input from "foo" */
                       if ( dup2(fd,STDIN_FILENO) == -1 )  _exit( 5 );

                       /* set standard output to "result" */
                       if ( dup2(fd2,STDOUT_FILENO) == -1 )  _exit( 6 );

                       close( fd );
                       close( fd2 );
                       execlp( "sort", "sort", 0 );
                       perror( "execlp" ;
                       _exit( 8 );
      }
      return 0;
}
```

The above program forks a child process to execute the *sort* command. After the child process is created, it redirects its standard input to be from the file *foo* and its standard output to the file *results*. If both *open* calls succeed, the child process calls *exec* to execute the *sort* command. Because there is no argument specified to the *sort* command, it will take data from its standard input (the file *foo*). The sorted data are written to the process standard output, which is now the *result* file.

It is possible to redirect standard input and/or output ports in a parent process before *fork* and *exec*. The difference in a child process is that after *fork*, the parent will not use the redirected port(s) or restore the redirected port(s) to its original source (e.g., */dev/tty*). It can then be used in the same way as before *fork*. It is, therefore, easier to redirect standard input and/or output in a child process just before an *exec* system call.