3. APPLICATION PROGRAMMING

Introduction

This chapter deals with programming where the objective is to produce sets of programs (applications) that will run on the SYSTEM V/68 operating system.

The chapter begins with a discussion of how the ground rules change as you move up the scale from writing programs that are essentially for your own private use (we have called this single-user programming) to working as a member of a programming team developing an application that is to be turned over to others to use.

There is a section on how the criteria for selecting appropriate programming languages may be influenced by the requirements of the application.

The next three sections of the chapter deal with a number of loosely related topics that are of importance to programmers working in the application development environment. Most of these mirror topics that were discussed in Chapter 2, "Programming Basics," but here we try to point out aspects of the subject that are particularly pertinent to application programming. They are covered under the following headings:

Advanced Programming	deals with such topics as file and record locking, interprocess communication, and the programming of terminal screens.
Support Tools	covers the common object file format, link editor directives, shared libraries, **sdb**, and **lint**.
Project Control Tools	includes some discussion of **make** and the Source Code Control System (SCCS).

The chapter concludes with a description of a sample application called **liber** that uses several of the components described in earlier portions of the chapter.

Application Programming Considerations

The characteristics of the application programming environment that make it different from single-user programming have at their base the need for interaction and for sharing of information.

Numbers

Perhaps the most obvious difference between application programming and single-user programming is in the quantities of the components. Not only are applications generally developed by teams of programmers, but the number of separate modules of code can grow into the hundreds on even a simple application.

When more than one programmer works on a project, there is a need to share such information as:

- the operation of each function
- the number, identity, and type of arguments expected by a function
- if pointers are passed to a function, whether the objects being pointed are to modified by the called function and what the lifetime of the pointed-to object is.
- the data type returned by a function

In an application, there is an odds-on possibility that the same function can be used in many different programs by many different programmers. The object code should be kept in a library accessible to anyone on the project who needs it.

Portability

When you are working on a program to be used on a single model of a computer, your concerns about portability are minimal. In application development, on the other hand, it is often a desirable objective to produce code that will run on many different operating systems. Some of the things that affect portability will be touched on later in this chapter.

Documentation

A single-user program has modest needs for documentation. There should be enough for the program's creator to recall how to use the program and what the intent was in portions of the code.

On an application development project there is a significant need for two types of internal documentation:

- comments throughout the source code that enable successor programmers to understand easily what is happening in the code. Applications can be expected to have a useful life of 5 or more years and frequently need to be modified during that time. It is not realistic to expect that the person who wrote the program will always be available to make modifications. Even if

that does happen, the comments will make the maintenance job a lot easier.

- hard-copy descriptions of functions should be available to all members of an application development team. Without them it is difficult to keep track of available modules, which can result in a function's being needlessly written again.

Unless end users have clear, readily available instructions on how to install and use an application, they either will not do it at all (if that is an option) or they will do it improperly.

Language Selection

This section presents some of the considerations that influence the selection of programming languages and describes two of the special-purpose languages that are part of the SYSTEM V/68 environment.

Influences

In single-user programming, the choice of language is often a matter of personal preference; a language is chosen because it is the one the programmer feels most comfortable with.

An additional set of considerations comes into play when a language must be chosen for an application development project.

Is there an existing standard within the organization that should be observed?

A firm may decide to emphasize one language because a good supply of programmers familiar with the language is available.

Does one language have better facilities for handling the algorithms involved in the application?

One would like to see all language selection based on such objective criteria, but it is often necessary to balance this against the skills of the organization.

Is there an inherent compatibility between the language and the operating system?

This is sometimes the impetus behind selecting C for programs destined for SYSTEM V/68.

Are there existing tools that can be used?

If parsing of input lines is an important phase of the application, perhaps a parser generator such as **yacc** should be employed to develop what the application needs.

Does the application integrate other software into the whole package?

If, for example, a package is to be built around an existing data base management system, there may be constraints on the variety of languages the data base management system can accommodate.

Special-Purpose Languages

The operating system contains several tools that can be included in the category of special-purpose languages. Three that are especially interesting are **awk**, **lex**, and **yacc**.

The awk Utility

The **awk** utility scans an ASCII input file record by record, looking for matches to specific patterns. When a match is found, an action is taken. Patterns and their accompanying actions are contained in a specification file referred to as the program.

The program can be made up of a number of statements. However, since each statement has the potential for causing a complex action, most **awk** programs contain only a few statements. The set of statements may include definitions of the pattern that separates one record from another (a newline character, for example) and of what separates one field of a record from the next (white space, for example). It may also include actions to be performed before the first record of the input file is read, and other actions to be performed after the final record has been read. All statements between are evaluated, in order, for each record in the input file. To paraphrase the action of a simple **awk** program, it would go something like this:

Look through the input file.
Every time you see this specific pattern, do this action.

A more complex **awk** program might be paraphrased like this:

First do some initialization.
Then, look through the input file.
Every time you see this specific pattern, do this action.
Every time you see this other pattern, do another action.
After all the records have been read, do these final things.

The directions for finding the patterns and for describing the actions can get pretty complicated, but the essential idea is as simple as the two sets of statements above.

One of the strong points of **awk** is that once you are familiar with the language syntax, programs can be written quickly. They don't always run very fast, however, so they are seldom appropriate if you want to run the same program repeatedly on a large quantities of records. In such a case, it is likely to be better to translate the program to a compiled language.

Using awk

One typical use of **awk** would be to extract information from a file and print it out in a report. Another might be to pull fields from records in an input file, arrange them in a different order and pass the resulting rearranged data to a function that adds records to your data base. The sample application at the end of this chapter contains an example of a use of **awk**.

The manual page for **awk** is in Section (1) of the *User's Reference Manual*. Chapter 4 in Part 2 of this guide contains a description of the **awk** syntax and gives examples of ways in which **awk** may be used.

The lex and yacc Utilities

The **lex** and **yacc** utilities are often described together because they perform complementary parts of what can be viewed as a single task: making sense out of input. The two utilities also share the common characteristic of producing source code for C language subroutines from specifications that appear on the surface to be similar.

Recognizing input is a recurring problem in programming. Input can be from various sources. In a language compiler, for example, the input is normally contained in a file of source language statements. The operating system shell language most often receives its input from a person keying in commands from a terminal. Frequently, information coming out of one program is fed into another where it must be evaluated.

The process of input recognition can be subdivided into two tasks: lexical analysis and parsing. That's where **lex** and **yacc** come in. In both utilities, the

specifications cause the generation of C language subroutines that deal with streams of characters. The **lex** utility generates subroutines that do lexical analysis, while **yacc** generates subroutines that do parsing.

To describe those two tasks in dictionary terms:

> Lexical analysis has to do with identifying the words or vocabulary of a language as distinguished from its grammar or structure.
>
> Parsing is the act of describing units of the language grammatically. Students in elementary school are often taught to do this with sentence diagrams.

Of course, the important thing to remember here is that in each case the rules for our lexical analysis or parsing are those we set down ourselves in the **lex** or **yacc** specifications. Because of this, the dividing line between lexical analysis and parsing sometimes becomes fuzzy.

The fact that **lex** and **yacc** produce C language source code means that these parts of what may be a large programming project can be separately maintained. The generated source code is processed by the C compiler to produce an object file. The object file can be link-edited with others to produce programs that perform whatever process follows from the recognition of the input.

Using lex

A **lex** subroutine scans a stream of input characters and waves a flag each time it identifies something that matches one or another of its rules. The waved flag is referred to as a "token." The rules are stated in a format that closely resembles the one used by the operating system text editor for regular expressions. For example, the notation:

```
[ \t]+
```

describes a rule that recognizes a string of one or more blanks or tabs (without mentioning any action to be taken). A more complete statement of the rule might have this notation:

```
[ \t]+        ;
```

which, in effect, says to ignore white space. The statement carries this meaning because no action is specified for when a string of one or more blanks or tabs is recognized. The semicolon marks the end of the statement. Another rule, one that does take some action, could be stated like this:

```
[0-9]+      {
        i = atoi(yytext);
        return(NBR);
        }
```


This rule depends on several things:

NBR must have been defined as a token in an earlier part of the **lex** source code called the declaration section. (It may be in a header file which is **#include**'d in the declaration section.)

The **i** is declared as an **extern int** in the declaration section.

A characteristic of **lex** is that things it finds are made available in a character string called **yytext**.

Actions can make use of standard C syntax. Here, the standard C subroutine, **atoi**, is used to convert the string to an integer.

What this rule boils down to is **lex** saying, "Hey, I found the kind of token we call NBR, and its value is now in **i**."

To review the steps of the process:

1. The **lex** specification statements are processed by the **lex** utility to produce a file called **lex.yy.c**. (This is the standard name for a file generated by **lex**, just as **a.out** is the standard name for the executable file generated by the link editor.)
2. The **lex.yy.c** file is transformed by the C compiler (with a **–c** option) into an object file called **lex.yy.o** that contains a subroutine called **yylex()**.
3. The **lex.yy.o** file is link-edited with other subroutines. Presumably, one of those subroutines will call **yylex()** with a statement such as:

   ```
   while((token = yylex()) != 0)
   ```

   and other subroutines (or even **main**) will deal with what comes back.

The manual page for **lex** is in Section (1) of the *Programmer's Reference Manual*. A tutorial on **lex** is in Chapter 5 in Part 2 of this guide.

Using yacc

Subroutines using **yacc** are produced by pretty much the same series of steps as **lex**:

1. The **yacc** specification is processed by the **yacc** utility to produce a file called **y.tab.c**.
2. The **y.tab.c** file is compiled by the C compiler, producing an object file, **y.tab.o**, that contains the subroutine **yyparse()**. A significant difference is that **yyparse()** calls a subroutine called **yylex()** to perform lexical analysis.
3. The object file **y.tab.o** may be link-edited with other subroutines, one of which will be called **yylex()**.

There are two things worth noting about this sequence:

1. The parser generated by the **yacc** specifications calls a lexical analyzer to scan the input stream and return tokens.
2. While the lexical analyzer is called by the same name as one produced by **lex**, it does not have to be the product of a **lex** specification. It can be any subroutine that does the lexical analysis.

What really differentiates these two utilities is the format for their rules. As noted above, **lex** rules are regular expressions like those used by the operating system's editors. Rules for **yacc** are chains of definitions and alternative definitions, written in Backus-Naur form, accompanied by actions. The rules may refer to other rules defined farther on in the specification. Actions are sequences of C language statements enclosed in braces. They frequently contain numbered variables that enable you to reference values associated with parts of the rules. For example:

```
%token NUMBER
%%
expr    : numb              { $$ = $1; }
        | expr '+' expr             { $$ = $1 + $3; }
        | expr '-' expr             { $$ = $1 - $3; }
        | expr '*' expr             { $$ = $1 * $3; }
        | expr '/' expr             { $$ = $1 / $3; }
        | '(' expr ')'              { $$ = $2; }
        ;
numb    : NUMBER            { $$ = $1; }
        ;
```

This fragment of a **yacc** specification shows:

- NUMBER identified as a token in the declaration section
- the start of the rules section indicated by the pair of percent signs
- alternate definitions for *expr* separated by the | sign and terminated by the semicolon
- actions to be taken when a rule is matched
- within actions, numbered variables used to represent components of the rule:

  $$ means the value to be returned as the value of the whole rule

  $*n* means the value associated with the *n*th component of the rule, counting from the left

- *numb* defined as meaning the token NUMBER. This is a trivial example that illustrates that one rule can be referenced within another, as well as within itself.

As with **lex**, the compiled **yacc** object file will generally be link-edited with other subroutines that handle processing that takes place after — or even ahead of — the parsing.

The manual page for **yacc** is in Section (1) of the *Programmer's Reference Manual.* Chapter 6 of this guide contains a detailed description of **yacc.**

Advanced Programming Tools

Chapter 2 described the use of such basic elements of programming as the standard I/O library, header files, system calls, and subroutines in the SYSTEM V/68 environment. This section introduces tools that are more apt to be used by members of an application development team than by a single-user programmer. This section contains material on the following topics:

- memory management
- file and record locking
- interprocess communication
- programming terminal screens

Memory Management

There are situations where a program needs to ask the operating system for blocks of memory. It may be, for example, that some records have been extracted from a data base and need to be held for further processing. Rather than writing them out to a file on secondary storage and then reading them back in again, it is likely to be a great deal more efficient to hold them in memory for the duration of the process. (This is not to ignore the possibility that portions of memory may be paged out before the program is finished; but such an occurrence is not pertinent to this discussion.)

There are two C language subroutines available for acquiring blocks of memory, both called **malloc**. One of them is **malloc**(3C); the other is **malloc**(3X). Each has several related commands that do specialized tasks in the same area. These commands are:

- **free**—to inform the system that space is being relinquished
- **realloc**—to change the size and possibly move the block
- **calloc**—to allocate space for an array and initialize it to zeros

In addition, **malloc**(3X) has a function, **mallopt**, and a structure, **mallinfo**. The **mallopt** function provides control over the space allocation algorithm. The **mallinfo** structure provides the program with information about the usage of the allocated space.

The **malloc**(3X) subroutine runs faster than the other version. To load it, you specify:

-lmalloc

on the **cc**(1) or **ld**(1) command line to direct the link editor to the proper library. When you use **malloc**(3X), your program should contain the statement:

```
#include <malloc.h>
```

where the values for **mallopt** options are defined.

See the *Programmer's Reference Manual* for the formal definitions of the two **malloc**s.

File and Record Locking

The provision for locking files, or portions of files, is primarily used to prevent the sort of error that can occur when two or more users of a file try to update information at the same time. The classic example is the airlines reservation system where two ticket agents each assign a passenger to Seat A, Row 5 on the 5 o'clock flight to Detroit. A locking mechanism is designed to prevent such

mishaps by blocking Agent B from even seeing the seat assignment file until Agent A's transaction is complete.

File locking and record locking are really the same thing, except that file locking implies the whole file is affected, while record locking means that only a specified portion of the file is locked. (Remember, in the operating system, file structure is undefined; a record is a concept of the programs that use the file.)

Two types of locks are available: "read" locks and "write" locks. If a process places a read lock on a file, other processes can also read the file but all are prevented from writing to it (that is, changing any of the data). If a process places a write lock on a file, no other processes can read *or* write in the file until the lock is removed. Write locks are also known as "exclusive locks." The term "shared lock" is sometimes applied to read locks.

Another distinction needs to be made between "mandatory" and "advisory" locking. Mandatory locking means that the discipline is enforced automatically for the system calls that read, write, or create files. This is done through a permission flag established by the file's owner (or the super-user). Advisory locking means that the processes that use the file take the responsibility for setting and removing locks as needed.

Thus mandatory locking may sound like a simpler and better deal, but it isn't so. The principal weakness in the mandatory method is that the lock is in place only while the single system call is being made. It is extremely common for a single transaction to require a series of reads and writes before it can be considered complete. In cases like this, the term "atomic" is used to describe a transaction that must be viewed as an indivisible unit. The preferred way to manage locking in such a circumstance is to make sure that the lock is in place before any I/O starts, and that the lock is not removed until the transaction is done. That calls for locking of the advisory variety.

Using File and Record Locking

The system call for file and record locking is **fcntl**(2). Programs should include the line:

```
#include <fcntl.h>
```

to bring in the header file shown in Figure 3-1.

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */
#define O_RDONLY 0
#define O_WRONLY 1
#define O_RDWR   2
#define O_NDELAY 04     /* Non-blocking I/O */
#define O_APPEND 010    /* append (writes guaranteed at the end) */
#define O_SYNC          020/* synchronous write option */
/* Flag values accessible only to open(2) */
#define O_CREAT  00400  /* open with file create (uses third open arg)*/
#define O_TRUNC  01000  /* open with truncation */
#define O_EXCL   02000  /* exclusive open */
/* fcntl(2) requests */
#define F_DUPFD  0      /* Duplicate fildes */
#define F_GETFD  1      /* Get fildes flags */
#define F_SETFD  2      /* Set fildes flags */
#define F_GETFL  3      /* Get file flags */
#define F_SETFL  4      /* Set file flags */
#define F_GETLK  5      /* Get file lock */
#define F_SETLK  6      /* Set file lock */
#define F_SETLKW 7      /* Set file lock and wait */
#define F_CHKFL  8      /* Check legality of file flag changes */
/* file segment locking set data type - information passed to system by user
struct flock {
        short    l_type;
        short    l_whence;
        long     l_start;
        long     l_len; /* len = 0 means until end of file */
        short    l_sysid;
        short    l_pid;
};
/* file segment locking types */
        /* Read lock */
#define F_RDLCK  01
        /* Write lock */
#define F_WRLCK  02
        /* Remove lock(s) */
#define F_UNLCK  03
```

Figure 3-1. The **fcntl.h** Header File

The format of the **fcntl**(2) system call is:

```
int fcntl(fildes, cmd, arg)
int fildes, cmd, arg;
```

The *fildes* is the file descriptor returned by the **open** system call. In addition to defining tags that are used as the commands on **fcntl** system calls, **fcntl.h**

includes the declaration for a *struct flock* that is used to pass values that control where locks are to be placed.

The lockf Subroutine

A subroutine, **lockf**(3), can also be used to lock sections of a file or an entire file. The format of **lockf** is:

```
#include <unistd.h>

int lockf (fildes, function, size)
int fildes, function;
long size;
```

The *fildes* is the file descriptor; *function* is one of four control values defined in **unistd.h** that let you lock, unlock, test and lock, or simply test to see if a lock is already in place. *size* is the number of contiguous bytes to be locked or unlocked. The section of contiguous bytes can be either forward or backward from the current offset in the file. (You can arrange to be somewhere in the middle of the file by using the **lseek**(2) system call.)

There is an example of file and record locking in the sample application at the end of this chapter. The manual pages that apply to this facility are **fcntl**(2), **fcntl**(5), **lockf**(3), and **chmod**(2) in the *Programmer's Reference Manual*. Chapter 7 in Part 2 of this guide is a detailed discussion of the subject, with examples.

Interprocess Communications

Chapter 2 described **fork**ing and **exec**ing as methods of communicating between processes. Business applications often need more sophisticated methods. For example, in applications where fast response is critical, a number of processes may be brought up at the start of a business day so that they are constantly available to handle transactions on demand. This cuts out initialization time that can add seconds to the time required to deal with the transaction. In transaction driven systems, the normal mode of processing is to have all the components of the application standing by waiting for an indication that there is work to do.

To meet requirements of this type, the operating system offers a set of nine system calls and their accompanying header files, all under the umbrella name of Interprocess Communications (IPC).

The IPC system calls come in sets of three: one set each for messages, semaphores, and shared memory. These three terms define three different styles of communication between processes.

messages	Communication is in the form of data stored in a buffer. The buffer can be either sent or received.
semaphores	Communication is in the form of positive integers with a value between 0 and 32,767. Semaphores may be contained in an array whose size is determined by the system administrator. The default maximum size for the array is 25.
shared memory	Communication takes place through a common area of main memory. One or more processes can attach a segment of memory and, therefore, can share whatever data is placed there.

The sets of IPC system calls are:

msgget	**semget**	**shmget**
msgctl	**semctl**	**shmctl**
msgop	**semop**	**shmop**

IPC get Calls

The **get** calls each return to the calling program an identifier for the type of IPC facility that is being requested.

IPC ctl Calls

The **ctl** calls provide a variety of control operations that include obtaining (IPC_STAT), setting (IPC_SET), and removing (IPC_RMID) the values in data structures associated with the identifiers picked up by the **get** calls.

IPC op Calls

The **op** manual pages describe calls that are used to accomplish the particular operations characteristic of the type of IPC facility being used. The **msgop** process has calls that send or receive messages. The **semop** process (the only one of the three that is actually the name of a system call) is used to increment or decrement the value of a semaphore, among other functions. The **shmop** process has calls that attach or detach shared memory segments.

The sample application at the end of this chapter includes an example of the use of some IPC features. The system calls are all located in Section (2) of the *Programmer's Reference Manual*. Don't overlook **intro**(2). It includes descriptions of the data structures that are used by IPC facilities. A detailed description of IPC, with many code examples that use the IPC system calls, is in Chapter 9.

Programming Terminal Screens

The facility for setting up terminal screens to meet the needs of your application is provided by two parts of the operating system. The first of these, **terminfo**, is a data base of compiled entries that describe the capabilities of terminals and the way they perform various operations.

The **terminfo** data base normally begins at the directory **/usr/lib/terminfo**. The members of this directory are themselves directories, generally with single-character names that are the first character in the name of the terminal. The compiled files of operating characteristics are at the next level down the hierarchy. For example, the entry for a Teletype 5425 is located in both the file **/usr/lib/terminfo/5/5425** and the file **/usr/lib/terminfo/t/tty5425**.

Describing the capabilities of a terminal can be a painstaking task. A good selection of terminal entries is included in the **terminfo** data base that comes with your computer. However, if you have a type of terminal that is not already described in the data base, the best way to proceed is to find a description of a terminal that comes close to having the same capabilities as yours and building on that description. There is a routine (**setupterm**) in **curses**(3X) that can be used to print out descriptions from the data base. Once you have worked out the code that describes the capabilities of your terminal, you use the **tic**(1M) command to compile the entry and add it to the data base.

The curses Package

After you have made sure that the operating capabilities of your terminal are a part of the **terminfo** data base, you can use the routines that make up the **curses**(3X) package to create and manage screens for your application.

The **curses** library includes functions to:

- define portions of your terminal screen as windows
- define pads that extend beyond the borders of your physical terminal screen and let you see portions of the pad on your terminal
- read input from a terminal screen into a program
- write output from a program to your terminal screen
- manipulate the information in a window in a virtual screen area and then send it to your physical screen

The sample application at the end of this chapter shows how you might use **curses** routines. Chapter 10 in Part 2 of this guide contains a tutorial on the subject. The manual pages for **curses** are in Section (3X), and those for **terminfo** are in Section (4) of the *Programmer's Reference Manual.*

3

Programming Support Tools

This section covers operating system components that, although part of the programming environment, have a highly specialized use. Among them are such things as:

- link edit command language
- Common Object File Format
- libraries
- Symbolic Debugger
- **lint** as a portability tool

Link Editor Command Language

The link editor command language is for use when the default arrangement of the **ld** output will not do the job. (The default locations for the standard Common Object File Format sections are described in **a.out**(4) in the *Programmer's Reference Manual.*)

When an **a.out** file is loaded into memory for execution, the text segment starts at location 0 and the data section starts at the next segment boundary after the end of the text (typically 0x400000). The stack begins at 1FFFFFF and grows to lower memory addresses. Note that these numbers may vary in different hardware configurations.

The link editor command language provides directives for describing different arrangements. The two major types of link editor directives are MEMORY and SECTIONS. MEMORY directives can be used to define the boundaries of configured and unconfigured sections of memory within a machine, to name sections, and to assign specific attributes (read, write, execute, and initialize) to portions of memory. SECTIONS directives, among many other functions, can be used to bind sections of the object file to specific addresses within the configured portions of memory.

Why would you want to be able to do those things? Well, in most cases you don't have to worry about it. The need to control the link editor output becomes more urgent under two (possibly related) sets of circumstances.

1. Your application is large and consists of numerous object files.
2. The hardware that your application is to run on is tight for space.

Chapter 12 in Part 2 of this guide gives a detailed description of the link editor command language.

Common Object File Format

A knowledge of COFF is fundamental to using the link editor command language. It is also good background knowledge for tasks such as:

- setting up archive libraries or shared libraries
- using the Symbolic Debugger

The following system header files contain definitions of data structures of parts of the Common Object File Format:

<syms.h>	symbol table format
<linenum.h>	line number entries
<ldfcn.h>	COFF access routines
<filehdr.h>	file header for a common object file
<a.out.h>	common assembler and link editor output
<scnhdr.h>	section header for a common object file
<reloc.h>	relocation information for a common object file
<storclass.h>	storage classes for common object files

The object file access routines are described below under the heading "The Object File Library."

Chapter 11 in Part 2 of this guide gives a detailed description of COFF.

Libraries

A library is a collection of related object files and/or declarations that simplify programming effort. Programming groups involved in the development of applications often find it convenient to establish private libraries. For example, an application with a number of programs using a common data base can keep the I/O routines in a library that is searched at link edit time.

Prior to Release 3 the libraries, whether system supplied or application developed, were collections of common object format files stored in an archive (*filename*.**a**) file that was searched by the link editor to resolve references. Files in the archive that were needed to satisfy unresolved references became a part of the resulting executable.

Beginning with Release 3, shared libraries are supported. Shared libraries resemble archive libraries in that they are collections of object files that are acted upon by the link editor. The difference, however, is that shared libraries perform a static linking between the file in the library and the executable that is the output of **ld**. The result is a saving of space, because all executables that need a file from the library share a single copy. Shared libraries are covered later in this section.

Chapter 2 described many of the functions that are found in the standard C library, **libc.a**. The next two sections describe two other libraries: the object file library and the math library.

The Object File Library

The object file library provides functions for the access and manipulation of object files. Some functions locate portions of an object file such as the symbol table, the file header, sections, and line number entries associated with a function. Other functions read these types of entries into memory. The need to work at this level of detail with object files occurs most often in the development of new tools that manipulate object files. For a description of the format of an object file, see "The Common Object File Format" in Chapter 11. This library consists of several portions. The functions reside in **/lib/libld.a**. They are loaded during the compilation of a C language program by the **-l** command line option, which causes the link editor to search the object file library:

cc *file* **–lld**

The argument **-lld** must appear after all files that reference functions in **libld.a**.

The following header files must be included in the source code:

```
#include <stdio.h>
#include <a.out.h>
#include <ldfcn.h>
```

Function	Reference	Brief Description
ldaclose	**ldclose**(3X)	Close object file being processed.
ldahread	**ldahread**(3X)	Read archive header.
ldaopen	**ldopen**(3X)	Open object file for reading.
ldclose	**ldclose**(3X)	Close object file being processed.
ldfhread	**ldfhread**(3X)	Read file header of object file being processed.
ldgetname	**ldgetname**(3X)	Retrieve the name of an object file symbol table entry.
ldlinit	**ldlread**(3X)	Prepare object file for reading line number entries via **ldlitem**.
ldlitem	**ldlread**(3X)	Read line number entry from object file after **ldlinit**.
ldlread	**ldlread**(3X)	Read line number entry from object file.
ldlseek	**ldlseek**(3X)	Seeks to the line number entries of the object file being processed.
ldnlseek	**ldlseek**(3X)	Seeks to the line number entries of the object file being processed given the name of a section.
ldnrseek	**ldrseek**(3X)	Seeks to the relocation entries of the object file being processed given the name of a section.

Function	Reference	Brief Description
ldnshread	**ldshread**(3X)	Read section header of the named section of the object file being processed.
ldnsseek	**ldsseek**(3X)	Seeks to the section of the object file being processed given the name of a section.
ldohseek	**ldohseek**(3X)	Seeks to the optional file header of the object file being processed.
ldopen	**ldopen**(3X)	Open object file for reading.
ldrseek	**ldrseek**(3X)	Seeks to the relocation entries of the object file being processed.
ldshread	**ldshread**(3X)	Read section header of an object file being processed.
ldsseek	**ldsseek**(3X)	Seeks to the section of the object file being processed.
ldtbindex	**ldtbindex**(3X)	Returns the long index of the symbol table entry at the current position of the object file being processed.
ldtbread	**ldtbread**(3X)	Reads a specific symbol table entry of the object file being processed.
ldtbseek	**ldtbseek**(3X)	Seeks to the symbol table of the object file being processed.

Function	Reference	Brief Description
sgetl	**sputl**(3X)	Access long integer data in a machine independent format.
sputl	**sputl**(3X)	Translate a long integer into a machine independent format.

Common Object File Interface Macros (ldfcn.h)

The interface between the calling program and the object file access routines is based on the defined type LDFILE, which is in the header file **ldfcn.h** (see **ldfcn**(4)). The primary purpose of this structure is to provide uniform access to both simple object files and to object files that are members of an archive file.

The function **ldopen**(3X) allocates and initializes the LDFILE structure and returns a pointer to the structure. The fields of the LDFILE structure may be accessed individually through the following macros:

- The TYPE macro, which returns the magic number of the file. The number is used to distinguish between archive files and object files that are not part of an archive.
- The IOPTR macro, which returns the file pointer. The pointer was opened by **ldopen**(3X) and is used by the input/output functions of the C library.
- The OFFSET macro, which returns the file address of the beginning of the object file. This value is non-zero only if the object file is a member of the archive file.
- The HEADER macro, which accesses the file header structure of the object file.

Additional macros are provided to access an object file. These macros parallel the input/output functions in the C library; each macro translates a reference to an LDFILE structure into a reference to its file descriptor field. The available macros are described in **ldfcn**(4) in the *Programmer's Reference Manual.*

The Math Library

The math library package consists of functions and a header file. The functions are located and loaded during the compilation of a C language program by the **–l** option on a command line, as follows:

cc *file* **–lm**

This option causes the link editor to search the math library, **libm.a**. In addition to the request to load the functions, the header file of the math library should be included in the program being compiled. This is accomplished by including the line:


```
#include <math.h>
```

near the beginning of each file that uses the routines.

The functions are grouped into the following categories:

- trigonometric functions
- Bessel functions
- hyperbolic functions
- miscellaneous functions

Trigonometric Functions

These functions compute angles (in radian measure), sines, cosines, and tangents. All these values are expressed in double precision.

Function	Reference	Brief Description
acos	**trig**(3M)	Return arc cosine.
asin	**trig**(3M)	Return arc sine.
atan	**trig**(3M)	Return arc tangent.
atan2	**trig**(3M)	Return arc tangent of a ratio.
cos	**trig**(3M)	Return cosine.
sin	**trig**(3M)	Return sine.
tan	**trig**(3M)	Return tangent.

Bessel Functions

These functions calculate Bessel functions of the first and second kinds of several orders for real values. The Bessel functions are **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**. The functions are located in section **bessel**(3M).

Hyperbolic Functions

These functions compute the hyperbolic sine, cosine, and tangent for real values.

Function	Reference	Brief Description
cosh	**sinh**(3M)	Return hyperbolic cosine.
sinh	**sinh**(3M)	Return hyperbolic sine.
tanh	**sinh**(3M)	Return hyperbolic tangent.

Miscellaneous Functions

These functions cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several are provided to truncate the integer portion of double-precision numbers.

Function	Reference	Brief Description
ceil	**floor**(3M)	Returns the smallest integer not less than a given value.
exp	**exp**(3M)	Returns the exponential function of a given value.
fabs	**floor**(3M)	Returns the absolute value of a given value.
floor	**floor**(3M)	Returns the largest integer not greater than a given value.
fmod	**floor**(3M)	Returns the remainder produced by the division of two given values.
gamma	**gamma**(3M)	Returns the natural log of the absolute value of the result of applying the gamma function to a given value.

Function	Reference	Brief Description
hypot	**hypot**(3M)	Returns the square root of the sum of the squares of two numbers.
log	**exp**(3M)	Returns the natural logarithm of a given value.
log10	**exp**(3M)	Returns the logarithm base ten of a given value.
matherr	**matherr**(3M)	Error-handling function.
pow	**exp**(3M)	Returns the result of a given value raised to another given value.
sqrt	**exp**(3M)	Returns the square root of a given value.

Shared Libraries

As noted above, beginning with Release 3, shared libraries are supported. Not only are some system libraries (**libc** and the networking library) available in both archive and shared library form, but applications have the option of creating private application shared libraries as well.

Shared libraries are desirable because they save space, both on disk and in memory. With an archive library, when the link editor goes to the archive to resolve a reference it takes a copy of the object file that it needs for the resolution and binds it into the **a.out** file. From that point on, the copied file is a part of the executable, whether it is in memory to be run or sitting in secondary storage. If you have many executables that use, say, **printf** (which requires much of the standard I/O library) you can be talking about a sizeable amount of space.

With a shared library, the link editor does not copy code into the executable files. When the operating system starts a process that uses a shared library, it maps the shared library contents into the address space of the process. Only one copy of the shared code exists, and many processes can use it at the same time.

This fundamental difference between archives and shared libraries has another significant aspect. When code in an archive library is modified, existing executables are unaffected. They continue using the older version until they are link-edited again. When code in a shared library is modified, all programs that share that code use the new version the next time they are executed.

Each process that uses shared library code gets its own copy of the entire data region of the library. It is actually only the text region that is really shared; so shared libraries may add space to executing **a.out**'s, even though the chances are good that they will cause more shrinkage than expansion. What this means is that when there is a choice between using a shared library and an archive, you shouldn't use the shared library unless it saves space. If you were using a shared **libc** to access only **strcmp**, for example, you would pick up more in shared library data than you would save by sharing the text.

The answer to this problem, and to others that are somewhat more complex, is to assign the responsibility for shared libraries to a central person or group within the application. The shared library developer should be the one to resolve questions of when to use shared and when to use archive system libraries. If a private library is to be built for your application, one person or organization should be responsible for its development and maintenance.

The sample application at the end of this chapter includes an example of the use of a shared library. Chapter 8 in Part 2 of this guide describes how shared libraries are built and maintained.

Symbolic Debugger

The use of **sdb** was mentioned briefly in Chapter 2. In this section we want to say a few words about **sdb** within the context of an application development project.

The **sdb** program operates on a process. It enables a programmer to find errors in the code. It is a tool a programmer might use while coding and unit testing a program, to make sure it runs according to its design. The **sdb** program would normally be used before the program is turned over, along with the rest of the application, to testers. During this phase of the application development cycle, programs are compiled with the **-g** option of **cc** to facilitate the use of the debugger. The symbol table should not be stripped from the object file. Once the programmer is satisfied that the program is error-free, **strip**(1) can be used to reduce the file storage overhead taken by the file.

If the application uses a private shared library, the possibility arises that a program bug may be located in a file residing in the shared library. Dealing with a problem of this sort calls for coordination by the administrator of the shared library. Any change to an object file that is part of a shared library means the change affects all processes that use that file. One program's bug may be another program's feature.

Chapter 15 in Part 2 of this guide contains information on how to use **sdb**. The

manual page is in Section (1) of the *Programmer's Reference Manual.*

lint as a Portability Tool

Generally speaking, it is desirable for a compiler to run fast. Most C compilers, therefore, let some things go unflagged so long as the language syntax is observed statement by statement. This sometimes means that while your program may run, the output will have some surprises. It also sometimes means that while the program may run on the machine on which the compilation system runs, you may have difficulty in running it on some other machine.

That's where **lint** comes in. The **lint** command produces comments about inconsistencies in the code. The types of anomalies flagged by **lint** are:

- cases of disagreement between the type of value expected from a called function and the value actually returned
- disagreement between the types and number of arguments expected by a function and what the function actually receives
- inconsistencies that might prove to be bugs
- things that might cause portability problems

Here is an example of a portability problem that would be caught by **lint.**

Code such as this would get by most compilers:

```
int i = lseek(fdes, offset, whence)
```

However, **lseek** returns a long integer representing the address of a location in the file. On a machine with a 16-bit integer and a bigger **long int**, the long integer value would produce incorrect results because **i** would contain only the last 16 bits of the value returned.

Chapter 16 in Part 2 of this guide contains a description of **lint** with examples of the kinds of conditions it uncovers. The manual page is in Section (1) of the *Programmer's Reference Manual.*

Project Control Tools

Volumes have been written on the subject of project control. It is an item of top priority for the managers of any application development team. Two operating system tools that can play a role in project control are described in this section.

The make Command

The **make** command is extremely useful for keeping track of what object files need to be recompiled as changes are made to source code files in an application development project. One of the characteristics of C programs is that they are made up of many small pieces, each in its own object file, that are link-edited together to form an executable file. Quite a few of the operating system tools are devoted to supporting that style of program architecture. For example, archive libraries, shared libraries and even the fact that the **cc** command accepts **.o** files as well as **.c** files (and that it can stop short of the **ld** step and produce **.o** files instead of an **a.out**) are all important elements of modular architecture. The two main advantages of this type of programming are that:

- A file that performs a given function can be reused in any program that needs it.
- The whole program does not have to be recompiled when one function is changed.

A consequence of the proliferation of object files is an increased difficulty in keeping track of what does and what does not need to be recompiled. The **make** command is designed to help deal with this problem. You use **make** by describing in a specification file, called **makefile**, the relationship (that is, the dependencies) between the different files of your program. Having done that, you conclude a session in which possibly a number of your source code files have been changed by running the **make** command. The **make** command takes care of generating a new **a.out** by comparing the time-last-changed of your source code files with the dependency rules you have given it.

The **make** command is able to work with files in archive libraries or under control of the Source Code Control System (SCCS).

Where to Find More Information

The **make**(1) manual page is contained in the *Programmer's Reference Manual.* Chapter 13 in Part 2 of this guide gives a complete description of how to use **make.**

SCCS

SCCS is an acronym for Source Code Control System. The system consists of a set of 14 commands used to track evolving versions of files. Its use is not limited to source code; any text files can be handled, so an application's documentation can also be put under control of SCCS. SCCS can:

- store and retrieve files under its control
- allow no more than a single copy of a file to be edited at one time
- provide an audit trail of changes to files
- reconstruct any earlier version of a file that may be wanted

SCCS files are stored in a special coded format. Only through commands that are part of the SCCS package can files be made available in a user's directory for editing, compiling, etc. From the point at which a file is first placed under SCCS control, only changes to the original version are stored. For example, let's say that the program **restate**, which was used in several examples in Chapter 2, was controlled by SCCS. One of the original pieces of that program is a file called **oppty.c** that looks like this:

```
                        /* Opportunity Cost -- oppty.c */
#include "recdef.h"

float
oppty(ps)
struct rec *ps;
{
        return(ps->i/12 * ps->t * ps->dp);
}
```

If you decide to add a message to this funtion, you might change the file like this:

```
                        /* Opportunity Cost -- oppty.c */
#include "recdef.h"
#include <stdio.h>

float
oppty(ps)
struct rec *ps;
{
        (void) fprintf(stderr, "Opportunity calling\n");
        return(ps->i/12 * ps->t * ps->dp);
}
```

SCCS saves only the two new lines from the second version, with a coded notation that shows where in the text the two lines belong. It also includes a note of the version number, lines deleted, lines inserted, total lines in the file, the date and time of the change and the login id of the person making the change.

Chapter 14 in Part 2 of this guide is an SCCS user's guide. SCCS commands are in Section (1) of the *Programmer's Reference Manual.*

liber, A Library System

The example on the following pages illustrates the use of operating system programming tools in the development of an application. The system is known as **liber**. The early stages of system development, we assume, have already been completed; feasibility studies have been done, the preliminary design is described in the coming paragraphs. We are going to stop short of producing a complete detailed design and module specifications for our system. You will have to accept that these exist. In using portions of the system for examples of the topics covered in this chapter, we will work from these virtual specifications.

We make no claim as to the efficacy of this design. Its sole purpose is to provide some passably realistic examples of operating system programming tools in use.

The **liber** system is a system for keeping track of the books in a library. The hardware consists of a single computer with terminals throughout the library. One terminal is used for adding new books to the data base. Others are used for checking out books and as electronic card catalogs.

The design of the system calls for it to be brought up at the beginning of the day and to remain running while the library is in operation. The system has one master index that contains the unique identifier of each title in the library. When

the system is running, the index resides in memory. Semaphores are used to control access to the index.

The pages that follow show fragments of some of the system's programs to illustrate how they work together. The startup program performs the system initialization; it opens the semaphores and shared memory, reads the index into the shared memory, and kicks off the other programs. The id numbers for the shared memory and semaphores (**shmid**, **wrtsem**, and **rdsem**) are read from a file during initialization. The programs all share the in-memory index. They attach it with the following code:

```
/* attach shared memory for index */
if ((int)(index = (INDEX *) shmat(shmid, NULL, 0)) == -1)
{
      (void) fprintf(stderr, "shmat failed: %d\n", errno);
      exit(1);
}
```

Of the programs shown, **add-books** is the only one that alters the index. The semaphores are used to ensure that no other programs will try to read the index while **add-books** is altering it. The checkout program locks the file record for the book, so that each copy being checked out is recorded separately and so that the book cannot be checked out at two different checkout stations at the same time.

The program fragments on the following pages do not provide any details on the structure of the index or the book records in the data base.

```
                /* liber.h - header file for the
                 *           library system.
                 */
typedef ... INDEX;/* data structure for book file index */
typedef struct {/* type of records in book file */
    char title[30];
    char author[30];
    .
    .
    .
} BOOK;
int shmid;
int wrtsem;
int rdsem;
INDEX *index;

int book_file;
BOOK book_buf;
/*  startup program*/

/*
 * 1. Open shared memory for file index and read it in.
 * 2. Open two semaphores for providing exclusive write access to index.
 * 3. Stash id's for shared memory segment and semaphores in a file
 *    where they can be accessed by the programs.
 * 4. Start programs:  add-books, card-catalog, and checkout running
 *    on the various terminals throughout the library.
 */

#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/sem.h>
#include"liber.h"

void exit();
extern int errno;

key_t key;
int shmid;
int wrtsem;
int rdsem;
FILE *ipc_file;

main()
{
    .
    .
    .
```

```
    if ((shmid = shmget(key, sizeof(INDEX), IPC_CREAT | 0666)) == -1)
    {
        (void) fprintf(stderr, "startup: shmget failed: errno=%d\n", errno);
        exit(1);
    }
    if ((wrtsem = semget(key, 1, IPC_CREAT | 0666)) == -1)
    {
        (void) fprintf(stderr, "startup: semget failed: errno=%d\n", errno);
        exit(1);
    }
    if ((rdsem = semget(key, 1, IPC_CREAT | 0666)) == -1)
    {
        (void) fprintf(stderr, "startup: semget failed: errno=%d\n", errno);
        exit(1);
    }
    (void) fprintf(ipc_file, "%d\n%d\n%d\n", shmid, wrtsem, rdsem);

    /*
     * Start the add-books program running on the terminal in the
     * basement.  Start the checkout and card-catalog programs
     * running on the various other terminals throughout the library.
     */
    .
    .
    .
}

/*  card-catalog program*/

/*
 * 1. Read screen for author and title.
 * 2. Use semaphores to prevent reading index while it is being written.
 * 3. Use index to get position of book record in book file.
 * 4. Print book record on screen or indicate book was not found.
 * 5. Go to 1.
 */

#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/ipc.h>
#include        <sys/sem.h>
#include<fcntl.h>
#include"liber.h"

void exit();
extern int errno;
struct sembuf sop[1];

main() {
    .
    .
    .
```

```
    while (1)
    {
        /*
         * Read author/title/subject information from screen.
         */

        /*
         * Wait for write semaphore to reach 0 (index not being written).
         */
        sop[0].sem_op = 1;
        if (semop(wrtsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }
        /*
         * Increment read semaphore so potential writer will wait
         * for us to finish reading the index.
         */
        sop[0].sem_op = 0;
        if (semop(rdsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }

        /* Use index to find file pointer(s) for book(s) */

        /* Decrement read semaphore */
        sop[0].sem_op = -1;
        if (semop(rdsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }

        /*
         * Now we use the file pointers found in the index to
         * read the book file.  Then we print the information
         * on the book(s) to the screen.
         */
    } /* while */
}
/*  checkout program*/

/*
 * 1. Read screen for Dewey Decimal number of book to be checked out.
 * 2. Use semaphores to prevent reading index while it is being written.
 * 3. Use index to get position of book record in book file.
 * 4. If book not found print message on screen, otherwise lock
 *    book record and read.
 * 5. If book already checked out print message on screen, otherwise
```

```
 *    mark record "checked out" and write back to book file.
 * 6. Unlock book record.
 * 7. Go to 1.
 */

#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/ipc.h>
#include        <sys/sem.h>
#include<fcntl.h>
#include"liber.h"

void exit();
long lseek();
extern int errno;
struct flock flk;
struct sembuf sop[1];
long bookpos;

main()
{
    .
    .
    .
    while (1)
    {
        /*
         * Read Dewey Decimal number from screen.
         */
        /*
         * Wait for write semaphore to reach 0 (index not being written).
         */
        sop[0].sem_flg = 0;
        sop[0].sem_op = 0;
        if (semop(wrtsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }
        /*
         * Increment read semaphore so potential writer will wait
         * for us to finish reading the index.
         */
        sop[0].sem_op = 1;
        if (semop(rdsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }

        /*
         * Now we can use the index to find the book's record position.
```

```
         * Assign this value to "bookpos".
         */

        /* Decrement read semaphore */
        sop[0].sem_op = -1;
        if (semop(rdsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }

        /* Lock the book's record in book file, read the record. */
        flk.l_type = F_WRLCK;
        flk.l_whence = 0;
        flk.l_start = bookpos;
        flk.l_len = sizeof(BOOK);
        if (fcntl(book_file, F_SETLKW, &flk) == -1)
        {
                (void) fprintf(stderr, "trouble locking: %d\n", errno);
                exit(1);
        }
        if (lseek(book_file, bookpos, 0) == -1)
        {
                Error processing for lseek;
        }
        if (read(book_file, &book_buf, sizeof(BOOK)) == -1)
        {
                Error processing for read;
        }

        /*
         * If the book is checked out inform the client, otherwise
         * mark the book's record as checked out and write it
         * back into the book file.
         */

        /* Unlock the book's record in book file. */
        flk.l_type = F_UNLCK;
        if (fcntl(book_file, F_SETLK, &flk) == -1)
        {
                (void) fprintf(stderr, "trouble unlocking: %d\n", errno);
                exit(1);
        }
    } /* while */
}

/*  add-books program*/

/*
 * 1. Read a new book entry from screen.
 * 2. Insert book in book file.
 * 3. Use semaphore "wrtsem" to block new readers.
```

```
 * 4. Wait for semaphore "rdsem" to reach 0.
 * 5. Insert book into index.
 * 6. Decrement wrtsem.
 * 7. Go to 1.
 */
#include <stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include"liber.h"

void exit();
extern int errno;
struct sembuf sop[1];
BOOK bookbuf;

main()
{
    .
    .
    .
    for (;;)
    {

        /*
         * Read information on new book from screen.
         */

        addscr(&bookbuf);

        /* write new record at the end of the bookfile.
         * Code not shown, but
         * addscr() returns a 1 if title information has
         * been entered, 0 if not.
         */

        /*
         * Increment write semaphore, blocking new readers from
         * accessing the index.
         */
        sop[0].sem_flg = 0;
        sop[0].sem_op = 1;
        if (semop(wrtsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }
        /*
         * Wait for read semaphore to reach 0 (all readers to finish
         * using the index).
         */
        sop[0].sem_op = 0;
```

```
        if (semop(rdsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }
        /*
         * Now that we have exclusive access to the index we
         * insert our new book with its file pointer.
         */

        /* Decrement write semaphore, permitting readers to read index. */
        sop[0].sem_op = -1;
        if (semop(wrtsem, sop, 1) == -1)
        {
                (void) fprintf(stderr, "semop failed: %d\n", errno);
                exit(1);
        }
    } /* for */
    .
    .
    .
}
```

The following example, **addscr()**, illustrates two significant points about **curses** screens:

1. Information read in from a **curses** window can be stored in fields that are part of a structure defined in the header file for the application.
2. The address of the structure can be passed from another function where the record is processed.

```
                /*  addscr is called from add-books.
                 *  The user is prompted for title
                 *  information.
                 */
#include <curses.h>

WINDOW *cmdwin;

addscr(bb)
struct BOOK *bb;
{
    int c;

    initscr();
    nonl();
    noecho();
    cbreak();

    cmdwin = newwin(6, 40, 3, 20);
    mvprintw(0, 0, "This screen is for adding titles to the data base");
    mvprintw(1, 0, "Enter  a  to add;  q  to quit: ");
    refresh();
    for (;;)
    {
        refresh();
        c = getch();
        switch (c) {
          case 'a':
                werase(cmdwin);
                box(cmdwin, '|', '-');
                mvwprintw(cmdwin, 1, 1, "Enter title: ");
                wmove(cmdwin, 2, 1);
                echo();
                wrefresh(cmdwin);
                wgetstr(cmdwin, bb->title);
                noecho();
                werase(cmdwin);
                box(cmdwin, '|', '-');
                mvwprintw(cmdwin, 1, 1, "Enter author: ");
                wmove(cmdwin, 2, 1);
                echo();
                wrefresh(cmdwin);
                wgetstr(cmdwin, bb->author);
                noecho();
                werase(cmdwin);
                wrefresh(cmdwin);
                endwin();
                return(1);
          case 'q':
                erase();
```

```
                endwin();
                return(0);
            }
        }
}

#
# Makefile for liber library system
#

CC = cc
CFLAGS = -O
all: startup add-books checkout card-catalog

startup: liber.h startup.c
    $(CC) $(CFLAGS) -o startup startup.c

add-books: add-books.o addscr.o
    $(CC) $(CFLAGS) -o add-books add-books.o addscr.o

add-books.o: liber.h

checkout: liber.h checkout.c
    $(CC) $(CFLAGS) -o checkout checkout.c

card-catalog: liber.h card-catalog.c
    $(CC) $(CFLAGS) -o card-catalog card-catalog.c
```

3