Symbol	.symtab entry?	Symbol type	Module where defined	Section
buf				
bufp0				
bufp1				
swap				
temp				

7.6 Symbol Resolution

The linker resolves symbol references by associating each reference with exactly one symbol definition from the symbol tables of its input relocatable object files. Symbol resolution is straightforward for references to local symbols that are defined in the same module as the reference. The compiler allows only one definition of each local symbol per module. The compiler also ensures that static local variables, which get local linker symbols, have unique names.

Resolving references to global symbols, however, is trickier. When the compiler encounters a symbol (either a variable or function name) that is not defined in the current module, it assumes that it is defined in some other module, generates a linker symbol table entry, and leaves it for the linker to handle. If the linker is unable to find a definition for the referenced symbol in any of its input modules, it prints an (often cryptic) error message and terminates. For example, if we try to compile and link the following source file on a Linux machine,

```
1   void foo(void);
2
3   int main() {
4       foo();
5       return 0;
6   }
```

then the compiler runs without a hitch, but the linker terminates when it cannot resolve the reference to `foo`:

```
linux> gcc -Wall -Og -o linkerror linkerror.c
/tmp/ccSz5uti.o: In function ‘main':
/tmp/ccSz5uti.o(.text+0x7): undefined reference to ‘foo'
```

Symbol resolution for global symbols is also tricky because multiple object modules might define global symbols with the same name. In this case, the linker must either flag an error or somehow choose one of the definitions and discard the rest. The approach adopted by Linux systems involves cooperation between the compiler, assembler, and linker and can introduce some baffling bugs to the unwary programmer.

Aside Mangling of linker symbols in C++ and Java

Both C++ and Java allow overloaded methods that have the same name in the source code but different parameter lists. So how does the linker tell the difference between these different overloaded functions? Overloaded functions in C++ and Java work because the compiler encodes each unique method and parameter list combination into a unique name for the linker. This encoding process is called *mangling*, and the inverse process is known as *demangling*.

Happily, C++ and Java use compatible mangling schemes. A mangled class name consists of the integer number of characters in the name followed by the original name. For example, the class `Foo` is encoded as `3Foo`. A method is encoded as the original method name, followed by `__`, followed by the mangled class name, followed by single letter encodings of each argument. For example, `Foo::bar(int, long)` is encoded as `bar__3Fooil`. Similar schemes are used to mangle global variable and template names.

7.6.1 How Linkers Resolve Duplicate Symbol Names

The input to the linker is a collection of relocatable object modules. Each of these modules defines a set of symbols, some of which are local (visible only to the module that defines it), and some of which are global (visible to other modules). What happens if multiple modules define global symbols with the same name? Here is the approach that Linux compilation systems use.

At compile time, the compiler exports each global symbol to the assembler as either *strong* or *weak*, and the assembler encodes this information implicitly in the symbol table of the relocatable object file. Functions and initialized global variables get strong symbols. Uninitialized global variables get weak symbols.

Given this notion of strong and weak symbols, Linux linkers use the following rules for dealing with duplicate symbol names:

Rule 1. Multiple strong symbols with the same name are not allowed.

Rule 2. Given a strong symbol and multiple weak symbols with the same name, choose the strong symbol.

Rule 3. Given multiple weak symbols with the same name, choose any of the weak symbols.

For example, suppose we attempt to compile and link the following two C modules:

```
1  /* foo1.c */
2  int main()
3  {
4      return 0;
5  }
```

```
1  /* bar1.c */
2  int main()
3  {
4      return 0;
5  }
```

In this case, the linker will generate an error message because the strong symbol `main` is defined multiple times (rule 1):

```
linux> gcc foo1.c bar1.c
/tmp/ccq2Uxnd.o: In function ‘main':
bar1.c:(.text+0x0): multiple definition of ‘main'
```

Similarly, the linker will generate an error message for the following modules because the strong symbol `x` is defined twice (rule 1):

```
1   /* foo2.c */
2   int x = 15213;
3
4   int main()
5   {
6       return 0;
7   }
```

```
1   /* bar2.c */
2   int x = 15213;
3
4   void f()
5   {
6   }
```

However, if `x` is uninitialized in one module, then the linker will quietly choose the strong symbol defined in the other (rule 2):

```
1   /* foo3.c */
2   #include <stdio.h>
3   void f(void);
4
5   int x = 15213;
6
7   int main()
8   {
9       f();
10      printf("x = %d\n", x);
11      return 0;
12  }
```

```
1   /* bar3.c */
2   int x;
3
4   void f()
5   {
6       x = 15212;
7   }
```

At run time, function `f` changes the value of `x` from 15213 to 15212, which might come as an unwelcome surprise to the author of function `main`! Notice that the linker normally gives no indication that it has detected multiple definitions of `x`:

```
linux> gcc -o foobar3 foo3.c bar3.c
linux> ./foobar3
x = 15212
```

The same thing can happen if there are two weak definitions of `x` (rule 3):

```
1  /* foo4.c */
2  #include <stdio.h>
3  void f(void);
4
5  int x;
6
7  int main()
8  {
9      x = 15213;
10     f();
11     printf("x = %d\n", x);
12     return 0;
13 }
```

```
1  /* bar4.c */
2  int x;
3
4  void f()
5  {
6      x = 15212;
7  }
```

The application of rules 2 and 3 can introduce some insidious run-time bugs that are incomprehensible to the unwary programmer, especially if the duplicate symbol definitions have different types. Consider the following example, in which `x` is inadvertently defined as an `int` in one module and a `double` in another:

```
1  /* foo5.c */
2  #include <stdio.h>
3  void f(void);
4
5  int y = 15212;
6  int x = 15213;
7
8  int main()
9  {
10     f();
```

```
11      printf("x = 0x%x y = 0x%x \n",
12             x, y);
13      return 0;
14  }
```

```
1   /* bar5.c */
2   double x;
3
4   void f()
5   {
6       x = -0.0;
7   }
```

On an x86-64/Linux machine, doubles are 8 bytes and ints are 4 bytes. On our system, the address of x is 0x601020 and the address of y is 0x601024. Thus, the assignment x = -0.0 in line 6 of bar5.c will overwrite the memory locations for x and y (lines 5 and 6 in foo5.c) with the double-precision floating-point representation of negative zero!

```
linux> gcc -Wall -Og -o foobar5 foo5.c bar5.c
/usr/bin/ld: Warning: alignment 4 of symbol `x' in /tmp/cclUFK5g.o
is smaller than 8 in /tmp/ccbTLcb9.o
linux> ./foobar5
x = 0x0 y = 0x80000000
```

This is a subtle and nasty bug, especially because it triggers only a warning from the linker, and because it typically manifests itself much later in the execution of the program, far away from where the error occurred. In a large system with hundreds of modules, a bug of this kind is extremely hard to fix, especially because many programmers are not aware of how linkers work, and because they often ignore compiler warnings. When in doubt, invoke the linker with a flag such as the GCC -fno-common flag, which triggers an error if it encounters multiply-defined global symbols. Or use the -Werror option, which turns all warnings into errors.

In Section 7.5, we saw how the compiler assigns symbols to COMMON and .bss using a seemingly arbitrary convention. Actually, this convention is due to the fact that in some cases the linker allows multiple modules to define global symbols with the same name. When the compiler is translating some module and encounters a weak global symbol, say, x, it does not know if other modules also define x, and if so, it cannot predict which of the multiple instances of x the linker might choose. So the compiler defers the decision to the linker by assigning x to COMMON. On the other hand, if x is initialized to zero, then it is a strong symbol (and thus must be unique by rule 2), so the compiler can confidently assign it to .bss. Similarly, static symbols are unique by construction, so the compiler can confidently assign them to either .data or .bss.

Practice Problem 7.2 (solution page 754)

In this problem, let REF(x.i) → DEF(x.k) denote that the linker will associate an arbitrary reference to symbol x in module i to the definition of x in module k. For each example that follows, use this notation to indicate how the linker would resolve references to the multiply-defined symbol in each module. If there is a link-time error (rule 1), write "ERROR". If the linker arbitrarily chooses one of the definitions (rule 3), write "UNKNOWN".

A.
```
/* Module 1 */          /* Module 2 */
int main()              int main;
{                       int p2()
}                       {
                        }
```

(a) REF(main.1) → DEF(_____._____)

(b) REF(main.2) → DEF(_____._____)

B.
```
/* Module 1 */          /* Module 2 */
void main()             int main = 1;
{                       int p2()
}                       {
                        }
```

(a) REF(main.1) → DEF(_____._____)

(b) REF(main.2) → DEF(_____._____)

C.
```
/* Module 1 */          /* Module 2 */
int x;                  double x = 1.0;
void main()             int p2()
{                       {
}                       }
```

(a) REF(x.1) → DEF(_____._____)

(b) REF(x.2) → DEF(_____._____)

7.6.2 Linking with Static Libraries

So far, we have assumed that the linker reads a collection of relocatable object files and links them together into an output executable file. In practice, all compilation systems provide a mechanism for packaging related object modules into a single file called a *static library*, which can then be supplied as input to the linker. When it builds the output executable, the linker copies only the object modules in the library that are referenced by the application program.

Why do systems support the notion of libraries? Consider ISO C99, which defines an extensive collection of standard I/O, string manipulation, and integer math functions such as `atoi`, `printf`, `scanf`, `strcpy`, and `rand`. They are available

to every C program in the `libc.a` library. ISO C99 also defines an extensive collection of floating-point math functions such as `sin`, `cos`, and `sqrt` in the `libm.a` library.

Consider the different approaches that compiler developers might use to provide these functions to users without the benefit of static libraries. One approach would be to have the compiler recognize calls to the standard functions and to generate the appropriate code directly. Pascal, which provides a small set of standard functions, takes this approach, but it is not feasible for C, because of the large number of standard functions defined by the C standard. It would add significant complexity to the compiler and would require a new compiler version each time a function was added, deleted, or modified. To application programmers, however, this approach would be quite convenient because the standard functions would always be available.

Another approach would be to put all of the standard C functions in a single relocatable object module, say, `libc.o`, that application programmers could link into their executables:

```
linux> gcc main.c /usr/lib/libc.o
```

This approach has the advantage that it would decouple the implementation of the standard functions from the implementation of the compiler, and would still be reasonably convenient for programmers. However, a big disadvantage is that every executable file in a system would now contain a complete copy of the collection of standard functions, which would be extremely wasteful of disk space. (On our system, `libc.a` is about 5 MB and `libm.a` is about 2 MB.) Worse, each running program would now contain its own copy of these functions in memory, which would be extremely wasteful of memory. Another big disadvantage is that any change to any standard function, no matter how small, would require the library developer to recompile the entire source file, a time-consuming operation that would complicate the development and maintenance of the standard functions.

We could address some of these problems by creating a separate relocatable file for each standard function and storing them in a well-known directory. However, this approach would require application programmers to explicitly link the appropriate object modules into their executables, a process that would be error prone and time consuming:

```
linux> gcc main.c /usr/lib/printf.o /usr/lib/scanf.o ...
```

The notion of a static library was developed to resolve the disadvantages of these various approaches. Related functions can be compiled into separate object modules and then packaged in a single static library file. Application programs can then use any of the functions defined in the library by specifying a single filename on the command line. For example, a program that uses functions from the C standard library and the math library could be compiled and linked with a command of the form

```
linux> gcc main.c /usr/lib/libm.a /usr/lib/libc.a
```

(a) `addvec.o` *code/link/addvec.c*

```
1   int addcnt = 0;
2
3   void addvec(int *x, int *y,
4               int *z, int n)
5   {
6       int i;
7
8       addcnt++;
9
10      for (i = 0; i < n; i++)
11          z[i] = x[i] + y[i];
12  }
```

code/link/addvec.c

(b) `multvec.o` *code/link/multvec.c*

```
1   int multcnt = 0;
2
3   void multvec(int *x, int *y,
4                int *z, int n)
5   {
6       int i;
7
8       multcnt++;
9
10      for (i = 0; i < n; i++)
11          z[i] = x[i] * y[i];
12  }
```

code/link/multvec.c

Figure 7.6 Member object files in the `libvector` library.

At link time, the linker will only copy the object modules that are referenced by the program, which reduces the size of the executable on disk and in memory. On the other hand, the application programmer only needs to include the names of a few library files. (In fact, C compiler drivers always pass `libc.a` to the linker, so the reference to `libc.a` mentioned previously is unnecessary.)

On Linux systems, static libraries are stored on disk in a particular file format known as an *archive*. An archive is a collection of concatenated relocatable object files, with a header that describes the size and location of each member object file. Archive filenames are denoted with the `.a` suffix.

To make our discussion of libraries concrete, consider the pair of vector routines in Figure 7.6. Each routine, defined in its own object module, performs a vector operation on a pair of input vectors and stores the result in an output vector. As a side effect, each routine records the number of times it has been called by incrementing a global variable. (This will be useful when we explain the idea of position-independent code in Section 7.12.)

To create a static library of these functions, we would use the AR tool as follows:

```
linux> gcc -c addvec.c multvec.c
linux> ar rcs libvector.a addvec.o multvec.o
```

To use the library, we might write an application such as `main2.c` in Figure 7.7, which invokes the `addvec` library routine. The include (or header) file `vector.h` defines the function prototypes for the routines in `libvector.a`,

To build the executable, we would compile and link the input files `main2.o` and `libvector.a`:

```
linux> gcc -c main2.c
linux> gcc -static -o prog2c main2.o ./libvector.a
```

code/link/main2.c

```
1   #include <stdio.h>
2   #include "vector.h"
3
4   int x[2] = {1, 2};
5   int y[2] = {3, 4};
6   int z[2];
7
8   int main()
9   {
10      addvec(x, y, z, 2);
11      printf("z = [%d %d]\n", z[0], z[1]);
12      return 0;
13  }
```

code/link/main2.c

Figure 7.7 Example program 2. This program invokes a function in the `libvector` library.

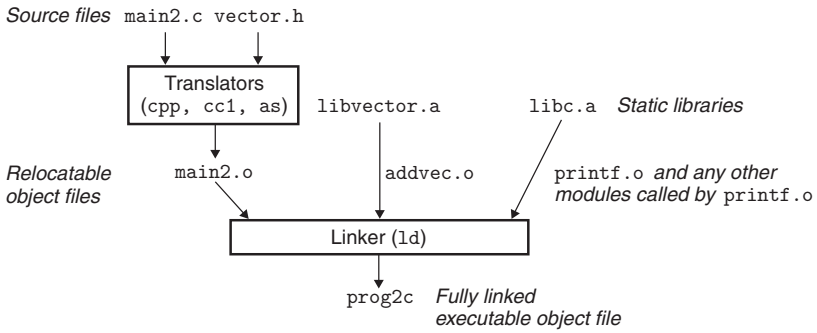

Figure 7.8 Linking with static libraries.

or equivalently,

```
linux> gcc -c main2.c
linux> gcc -static -o prog2c main2.o -L. -lvector
```

Figure 7.8 summarizes the activity of the linker. The `-static` argument tells the compiler driver that the linker should build a fully linked executable object file that can be loaded into memory and run without any further linking at load time. The `-lvector` argument is a shorthand for `libvector.a`, and the `-L.` argument tells the linker to look for `libvector.a` in the current directory.

When the linker runs, it determines that the `addvec` symbol defined by `addvec.o` is referenced by `main2.o`, so it copies `addvec.o` into the executable.

Since the program doesn't reference any symbols defined by `multvec.o`, the linker does *not* copy this module into the executable. The linker also copies the `printf.o` module from `libc.a`, along with a number of other modules from the C run-time system.

7.6.3 How Linkers Use Static Libraries to Resolve References

While static libraries are useful, they are also a source of confusion to programmers because of the way the Linux linker uses them to resolve external references. During the symbol resolution phase, the linker scans the relocatable object files and archives left to right in the same sequential order that they appear on the compiler driver's command line. (The driver automatically translates any `.c` files on the command line into `.o` files.) During this scan, the linker maintains a set E of relocatable object files that will be merged to form the executable, a set U of unresolved symbols (i.e., symbols referred to but not yet defined), and a set D of symbols that have been defined in previous input files. Initially, E, U, and D are empty.

- For each input file f on the command line, the linker determines if f is an object file or an archive. If f is an object file, the linker adds f to E, updates U and D to reflect the symbol definitions and references in f, and proceeds to the next input file.
- If f is an archive, the linker attempts to match the unresolved symbols in U against the symbols defined by the members of the archive. If some archive member m defines a symbol that resolves a reference in U, then m is added to E, and the linker updates U and D to reflect the symbol definitions and references in m. This process iterates over the member object files in the archive until a fixed point is reached where U and D no longer change. At this point, any member object files not contained in E are simply discarded and the linker proceeds to the next input file.
- If U is nonempty when the linker finishes scanning the input files on the command line, it prints an error and terminates. Otherwise, it merges and relocates the object files in E to build the output executable file.

Unfortunately, this algorithm can result in some baffling link-time errors because the ordering of libraries and object files on the command line is significant. If the library that defines a symbol appears on the command line before the object file that references that symbol, then the reference will not be resolved and linking will fail. For example, consider the following:

```
linux> gcc -static ./libvector.a main2.c
/tmp/cc9XH6Rp.o: In function ‘main':
/tmp/cc9XH6Rp.o(.text+0x18): undefined reference to ‘addvec'
```

What happened? When `libvector.a` is processed, U is empty, so no member object files from `libvector.a` are added to E. Thus, the reference to `addvec` is never resolved and the linker emits an error message and terminates.

The general rule for libraries is to place them at the end of the command line. If the members of the different libraries are independent, in that no member references a symbol defined by another member, then the libraries can be placed at the end of the command line in any order. If, on the other hand, the libraries are not independent, then they must be ordered so that for each symbol *s* that is referenced externally by a member of an archive, at least one definition of *s* follows a reference to *s* on the command line. For example, suppose `foo.c` calls functions in `libx.a` and `libz.a` that call functions in `liby.a`. Then `libx.a` and `libz.a` must precede `liby.a` on the command line:

```
linux> gcc foo.c libx.a libz.a liby.a
```

Libraries can be repeated on the command line if necessary to satisfy the dependence requirements. For example, suppose `foo.c` calls a function in `libx.a` that calls a function in `liby.a` that calls a function in `libx.a`. Then `libx.a` must be repeated on the command line:

```
linux> gcc foo.c libx.a liby.a libx.a
```

Alternatively, we could combine `libx.a` and `liby.a` into a single archive.

Practice Problem 7.3 **(solution page 754)**

Let `a` and `b` denote object modules or static libraries in the current directory, and let `a`→`b` denote that `a` depends on `b`, in the sense that `b` defines a symbol that is referenced by `a`. For each of the following scenarios, show the minimal command line (i.e., one with the least number of object file and library arguments) that will allow the static linker to resolve all symbol references.

A. `p.o` → `libx.a`

B. `p.o` → `libx.a` → `liby.a`

C. `p.o` → `libx.a` → `liby.a` *and* `liby.a` → `libx.a` → `p.o`

7.7 Relocation

Once the linker has completed the symbol resolution step, it has associated each symbol reference in the code with exactly one symbol definition (i.e., a symbol table entry in one of its input object modules). At this point, the linker knows the exact sizes of the code and data sections in its input object modules. It is now ready to begin the relocation step, where it merges the input modules and assigns run-time addresses to each symbol. Relocation consists of two steps:

1. *Relocating sections and symbol definitions.* In this step, the linker merges all sections of the same type into a new aggregate section of the same type. For example, the `.data` sections from the input modules are all merged into one section that will become the `.data` section for the output executable object