Object files and friends

Object files are a special kind of file which store compiled programs. Normally, you manipulate them only as part of the build process, where you can treat them as a black box: you don't need to know what they look like inside.

Sometimes, however, some aspects of the true nature of object files become apparent—for example:

- Your program bombs out with a segmentation violation, and when you check up you discover that it was trying to write to a valid storage location—so why did it bomb out? It might be that the location was in the text segment, a part of the address space that is read-only.
- You want to debug a program, and find that the debugger refuses to look at it, because it doesn't have any symbols—whatever *that* may mean.
- You recompile programs and run out of disk space—for some reason, the object files are suddenly ten times the size that they used to be.

The information in this chapter is some of the most technical in the whole book, which is why I've left it to the end. We look at a number of topics that are related only by their dependence on object files. So far, the inter-platform differences we've seen have been the result of a choice made by the software people who implemented the system. In this chapter, we come a whole lot closer to the hardware—you can almost feel the clocks tick and the pipelines fill. You definitely see instructions execute. You'll find it an interesting look below covers that are usually locked shut.

A number of programs manipulate the object files either because that's their purpose—for example, assemblers or linkers—or because they want to play tricks to install more comfortably. For example, *emacs* and TEX both write themselves out as object files during the build process. If anything goes wrong with these programs, you need to open the black box and look inside. In this chapter, we'll examine the tools that manipulate object files and some of the background information that you need to know to solve problems.

There aren't many programs that manipulate object files. The kernel uses absolute object files when creating a process—this is the most frequent use of an object file. In addition, the *assembler* creates them from assembly sources. In most UNIX systems, this is the only

program that creates object files from scratch. The *linker* or *link editor* joins object files together to form a larger object file, and debuggers access specific debugging information in the object file. These are the only programs that have intimate understanding of the object file format.

A number of smaller programs do relatively trivial things with object files:

- The archiver *ar* is normally used for archiving binary files, but it does not know very much about their contents.
- The name list display program *nm* displays the symbol table or *name list* of an object file or an archive of object files. We'll look at the symbol table in more detail on page 363.
- *size* displays size information from an object file.
- *strings* displays printable strings in an object file.
- *strip* removes unnecessary information from an object file.

In the rest of this chapter, we'll look at the following topics:

- The kernel process model that the object file supports.
- The assembler, including some of the syntax, the symbol table, relocation, and debugging symbols.
- The linker, including the way it searches libraries, and some of the problems that can occur during linking.
- The internal structure of function libraries, and how this affects what they can and cannot do.
- How *emacs* and TEX dump themselves as object files.
- How `exec` starts programs.

Object formats

The purpose of object files is to make it as easy as possible to start a process, so it makes sense to look at the process image in memory first. Modern UNIX systems run on stack-based systems with virtual memory. We touched on the concept of virtual memory in Chapter 11, *Hardware dependencies*, on page 155. Since UNIX is a multiprogramming system, it is possible for more than one process to run from a single object file. These facts have a significant influence on the way the system manages processes. For each process, the system allocates at least three segments in which program code and data is stored:

- A *text segment*, which contains the executable code of the program and read-only data. Modern systems create code where the program may not write to its text segment—it is so-called *pure text*. This has two significant advantages for the memory manager: first, all processes in the system that are run from this program can share the same text segment, which significantly reduces the memory requirements when, say, 20 copies of a shell are running. In addition, since the text is not modified, the memory management

routines never need to swap it out to disk. The copy on disk is always the same as the copy in memory. This also means that the copy on disk can be the copy in the object file: it does not take up any space in the swap partition.

Older systems also provided for *impure text* segments that could be modified by the program. This usage is obsolete, but it is still supported by modern systems.

- A *data segment*. This consists of two parts:
  - Global data that has been initialized in the program. This data can be modified, of course, so it takes up space in the swap partition, but the first time the page is referenced, the memory manager must load it from the object file.
  - *bss*[*] *data*, non-initialized global data. Since the data is not initialized, it does not need to be loaded from a file. The first time the page is referenced, the memory manager just creates an empty data page. After that, it gets paged to the swap partition in the same way as initialized data.
- A *stack segment*. Like bss data, the stack segment is not initialized, and so is not stored in the object file. Unlike any of the other segments, it does not contain any fixed addresses: at the beginning of program execution, it is almost empty, and all data stored in it is relative to the top of the stack or another stack marker. We'll look at stack organization in more detail on page 377.
- In addition, many systems have *library segments*. From the point of view of memory management, these segments are just additional text and data segments, but they are loaded at run time from another object file, the *library file*.

Older systems without virtual memory stored the data segment below the stack segment with a gap in between, the so-called *break*. The stack grew down into the break as the result of *push* or *call* instructions, and the data segment grew up into the break as the result of system calls `brk` and `sbrk` (*set break*). This additional space in the data segment is typically used for memory allocated by the library call `malloc`. With a virtual memory system, the call to `sbrk` is no longer necessary, but some versions of UNIX still require it, and all support it. Table 21-1 summarizes this information:

* The name comes from the assembler directive *bss* (Block Starting with Symbol), which was used in older assemblers to allocate uninitialized memory and allocate the address of the first word to the label of the directive. There was also a directive *bes* (Block Ending with Symbol) which allocated the address of the last word to the label.

Table 21–1: Kinds of segments

Property	Text Segment	Initialized Data	bss Data	Stack Segment
In object file	yes	yes	no	no
Access	r–x	rw–	rw–	rw–
Paged out	no	yes	yes	yes
Fixed size	yes	yes	maybe	no

Object files contain the information needed to set up these segments. Before we continue, we should be aware of a terminology change:

- The object file for a process is called a *program*.
- The images of process segments in an object file are called *sections*.

There are three main object file formats in current use:

- The *a.out* format is the oldest, and has remained essentially unchanged since the Seventh Edition. It supplies support for a text section and a data section, as well as relocation information for both sections. It is used by XENIX and BSD systems.
- The *COFF* (Common Object File Format) was introduced in System V, and offers an essentially unlimited number of segments, including library segments. It is now obsolescent, except for use in Microsoft Windows NT.
- The *ELF* (Executable and Linking Format) format was introduced for System V.4. From our point of view, it offers essentially the same features as COFF. ELF shows promise as the executable format of the future, since it greatly simplifies the use of shared libraries. Currently the Linux project is moving from a.out to ELF.

With the exception of library segments, there's not much to choose between the individual object formats, but the internal structures and the routines that handle them are very different. Let's take an *a.out* header from a BSD system as an example. The header file *sys/exec.h* defines:

```
struct exec
  {
  long  a_magic;              /* magic number */
  unsigned long a_text;       /* text segment size */
  unsigned long a_data;       /* initialized data size */
  unsigned long a_bss;        /* uninitialized data size */
  unsigned long a_syms;       /* symbol table size */
  unsigned long a_entry;      /* entry point */
  unsigned long a_trsize;     /* text relocation size */
  unsigned long a_drsize;     /* data relocation size */
  };

/* a_magic */
#define OMAGIC          0407  /* old impure format */
```

```
#define NMAGIC          0410  /* read-only text */
#define ZMAGIC          0413  /* demand load format */
#define QMAGIC          0314  /* compact demand load format */
```

This header includes:

- A *magic number*. This specifies the exact kind of file (for example, whether it is relocatable or absolute). The program *file* can interpret this magic number and report the kind of object file.
- The length of the *text section*, an image of the text segment. The text section immediately follows the header.
- The length of the *data section*, an image of the initialized global data part of the data segment—as we have seen, bss data does not need to be stored in the object file. The data section immedately follows the text section.
- The length of the bss data. Since the bss data is not initialized, no space is needed for it in the object file.
- The length of the symbol table. The symbol table itself is stored after the data section.
- The *entry point*, the address in the text segment at which execution is to start.
- The lengths of the text and data relocation tables, which are stored after the symbol table.

If you look at the above list of contents carefully, you'll notice that there are no start addresses for the segments, and there isn't even any mention of the stack segment. The start address of the text and data segments is implicit in the format, and it's frequently difficult information to figure out. On 32 bit machines, the text segment typically starts at a low address, for example `0` or `0x1000`.* The data segment may start immediately after the text segment (on the following page), or it might start at a predetermined location such as `0x40000000`. The stack segment is usually placed high in the address space. Some systems place it at `0x7fffffff`, others at `0xefffffff`. The best way to find out these addresses is to look through the address space of a typical process with a symbolic debugger.

The magic number is worth closer examination: I said that it occupies the first two bytes of the header, but in our example it is a `long`, four bytes. In fact, the magic number is used in two different contexts:

- The first two bytes in the file are reserved for the magic number in all systems. The information in these bytes should be sufficient to distinguish the architecture.
- The following two bytes may contain additional information for specific systems, but it is often set to 0.

* Why `0x1000`? It's a wonderful debugging aid for catching `NULL` pointers. If the first page of memory is not mapped, you'll get a segmentation violation or a bus error if you try to access data at that address

The assembler

Assembly is the second oldest form of programming*. It is characterized by being specific about the exact instructions that the machine executes, which makes an assembler program much more voluminous than a higher level language. Nevertheless, there is nothing difficult about it, it's just tedious.

Assembler programming involves two aspects that don't have much in common:

- The instruction set of the machine in question. The best source of information for this aspect is the hardware description of the machine. Even if you get an assembler manual for the machine, it will not be as authoratative as the hardware description.
- The syntax of the assembler. This is where the problems start: first, little documentation is available, and secondly, assembler syntax diverges greatly, and the documentation you get may not match your assembler.

The i386 is a particularly sorry example of incompatible assembler syntax. The UNIX assemblers available for the i386 (at least three of them, not even compatible with each other) use modified forms of the old UNIX *as* syntax, whereas all books about the assembler and the hardware of the i386 use a syntax related to the Microsoft assembler *MASM*. They don't even agree on such basic things as the names of the instructions and the sequence of the operands.

Although nowadays it is used almost only for assembling compiler output, *as* frequently offers features specifically intended for human programmers. In particular, most assemblers support some kind of preprocessing: they may use the macro preprocessor *m4* or the C preprocessor when processing assembler source. See the description of the flags in Appendix C, *Assembler directives and flags*, page 415, for more information.

Assembler syntax

Assembler syntax is a relatively involved topic, but there are some general rules that apply to just about every assembler. In this section, we'll see how to fight our way through an assembler listing.

- Assemblers are *line-oriented*: each instruction to the assembler is placed on a separate line.
- An instruction line consists of four parts:
  - If the optional *label* is present, the assembler assigns a value to it. For most instructions, the value is the current value of the *location counter*, the relative address of the instruction in the current section. In UNIX, if the label is present it is followed by a colon (`:`). Other assemblers frequently require that only labels start at the beginning of the line, and recognize them by this fact.

* The oldest form of programming, of course, used no computational aids whatsoever: in some form or another, the programmer wrote down direct machine code and then entered into memory with a loader or via front-panel switches. Assembly added the symbolic character to this operation.

The assembler usually translates the source file in a single pass. This means that when it encounters the name of a label that is further down in the source file, it cannot know its value or even if it exists. Some assemblers require that the name of the label be followed with the letter `b` (*backwards*) for labels that should have already been seen in the text, and `f` (*forwards)* for labels that are further down. In order to avoid ambiguity, these assemblers also require that the labels be all digits. Many other assemblers also support this syntax, so `1b` is not a good name for a label.

  - The next field is the *instruction*. In this context, *assembler instructions* are commands to the assembler, and may be either *directives*, which tell the assembler to do something not directly related to emitting code, or *machine instructions*, which emit code. In UNIX, directives frequently start with a period (`.`).
  - The third field contains the `operands` for the instruction. Depending on the instruction, they may not be required.
  - The fourth field is a comment field. It is usually delimited by a hash mark (`#`).

- The operands of instructions that take a source operand and a destination operand are usually specified in the sequence *src, dest*.
- Register names are usually preceded with a `%` sign.
- Literal values are usually preceded with a `$` sign.

For example, consider the instruction:

```
fred:   movl    $4,%eax # example
```

This instruction emits a `movl` instruction, which moves the literal* value 4 to the register `eax`. The symbol `fred` is set to the address of the instruction.

We can't go into all the details of the assembly language for all machines, but the descriptions in Appendix C, *Assembler directives and flags*, page 415, will hopefully give you enough insight to be able to read existing assembler source, though you'll need more information before you can write it. One of the few reasonably detailed *as* manuals is *Using as*, by Dean Elsner and Jay Fenlason, which is included as part of the GNU binutils distribution.

Assembler symbols

Local symbols define instruction addresses. High-level constructs in C such as `if`, `while` and `switch` require a number of `jump` (`go to`) instructions in assembler, and the compiler must generate labels for the instructions.

Local symbols are also used to label literal data constants such as strings.

Global symbols defined in the source. The word *global* has different meanings in C and assembler: in C, it is any symbol defined in the data or text segments, whether or not it is

* `movl` means "move long", not "move literal". In this particular assembler, we know that it is a literal value because of the `$` symbol, just as we know that `eax` is a register name because it is preceded by a `%` sign.

visible outside the module. In assembler, a global symbol is one that is visible outside the module.

There are a couple of points to note here:

- C local variables are generated automatically on the stack and do not retain their names after compilation. They do not have a fixed location, since their position on the stack depends on what was already on the stack when the function was called. If the function is recursive, they could even be in many different places on the stack at the same time. As a result, there is nothing that the assembler or the linker can do with the symbols, and the compiler discards them.
- There is a possibility of conflict between the local symbols generated by the compiler and global symbols declared in the program. Most compilers avoid this conflict by prepending an underscore (_) to all symbols defined in the program, and not using the underscore for local symbols. Others solve the problem by prepending local symbols with a period (.), which is not legal in a C identifier.

To see how this all works, let's take the following small program and look at different aspects of what the compiler and assembler do with it in the next few sections:

Example 21–1:

```
char global_text [] = "This is global text in the data area";
void inc (int *x, int *y)
{
  if (*x)
    (*x)++;
  else
    (*y)++;
  puts (global_text);         /* this is an external function */
  puts ("That's all, folks");
  }
```

We compile this program on a BSD/OS machine using *gcc* version 2.5.8, with maximum optimization and debugging symbols:

```
$ gcc -O2 -g -S winc.c
```

The `-S` flag tells the compiler control program to stop after running the compiler. It stores the assembly output in *winc.s*, which looks like this:

Example 21–2:

```
        .file   "winc.c"
gcc2_compiled.:
___gnu_compiled_c:
.stabs "/usr/lemis/book/porting/grot/",100,0,0,Ltext0 name of the source directory
.stabs "winc.c",100,0,0,Ltext0  name of the source file
.text                           select text section
Ltext0:                         internal label: start of text
.stabs "int:t1=r1;-2147483648;2147483647;",128,0,0,0
.stabs "char:t2=r2;0;127;",128,0,0,0
...  a whole lot of standard debugging output omitted
```

Example 21–2: (continued)

```
.stabs "void:t19=19",128,0,0,0
.globl _global_text             specify an externally defined symbol
.data                           select data section
.stabs "global_text:G20=ar1;0;36;2",32,0,1,0  debug info for global symbol
_global_text:                   variable label
        .ascii "This is global text in the data area " and text
.text                           select text section
LC0:
        .ascii "That's all, folks "
        .align 2                start on a 16 bit boundary
.globl _inc                     define the function inc to be external
_inc:                           start of function inc
        .stabd 68,0,3           debug information: start of line 3
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax
        movl 12(%ebp),%edx
        .stabd 68,0,4           debug information: start of line 4
LBB2:
        cmpl $0,(%eax)
        je L2
        .stabd 68,0,5           debug information: start of line 5
        incl (%eax)
        jmp L3
        .align 2,0x90
L2:
        .stabd 68,0,7           debug information: start of line 7
        incl (%edx)
L3:
        .stabd 68,0,8           debug information: start of line 8
        pushl $_global_text
        call _puts
        .stabd 68,0,9           debug information: start of line 9
        pushl $LC0
        call _puts
        .stabd 68,0,10          debug information: start of line 10
LBE2:
        leave
        ret
.stabs "inc:F19",36,0,3,_inc    debug information for inc
.stabs "x:p21=*1",160,0,2,8     debug information for x
.stabs "y:p21",160,0,2,1        debug information for y
.stabs "x:r21",64,0,2,0
.stabs "y:r21",64,0,2,2
.stabn 192,0,0,LBB2
.stabn 224,0,0,LBE2
```

We'll look at various aspects of this output in the next few sections. For now, we should notice:

- As advertised, the names of the global symbols `global_text`, `inc` and `puts` have been changed to `_global_text`, `_inc` and `_puts`.

- The compiler has created the local symbols `Ltext0`, `LC0`, `LBB2`, `LBE2`, `L2` and `L3`. Clearly it likes to start the names of local symbols with the letter `L`, and distinguish them with numbers at the end. But what has happened to `L1`, for example? The compiler generated it, but the optimizer optimized it away. If you compile this same program without the optimizer, the labels will all still be there.
- The compiler has assigned the local symbol `LC0` to the string `"That's all, folks"` so that the assembler can refer to it.
- The variables `x` and `y` have disappeared, since they exist only on the stack.

Relocation information

Example 21-2 shows another dilemma that afflicts the linker: the program is not complete. It refers to the external symbol `_puts`, and in addition it does not have a `main` function: the only way to use it is for a function in another object file to call `_inc`. In order to do this, we need to give the linker additional information:

- Information about the names of the external symbols that the object file references (`_puts` in our example).
- Information about symbols defined in the object file that can be referenced by other object files (`_global_text` and `_inc` in our example).
- Information about where external symbols are referenced in the object code.
- Information about where locations in the text and data segments are referenced in the object code.

Why do we need to know where *internal* locations are referenced? The linker takes the text and data sections from a large number of object files and makes a single text section and a single data section out of them. The locations of the original sections from the individual object files differ from one occasion to the next, but the addresses in the final executable must reflect the correct references. If an instruction refers to an address in the data or text section or an external symbol, the assembler can't just put the address of the item in the instruction, since the address is allocated by the linker. Instead, it places the *offset* from the beginning of the text or data section or from the external symbol into the instruction or data word, and generates a *relocation record* in the output file. These relocation records contain the following information:

- The *address* of the data that needs to be relocated. From the linker's point of view, the data may be an instruction, in which case it will need to modify only the address portion of the instruction, or it may be a pointer, in other words an indirect address.
- The *length* of the data. For a data pointer, this is the length of the pointer. For an instruction, it is the length of the address field of the instruction. Some systems have strange instruction formats, and this can become quite complicated.
- Information about the section in which the data will be located. This could be the current text or data section, or it could be a reference to an external symbol.

- For an external symbol, a pointer to information about the symbol.

Object files contain separate relocation tables for each section that can contain address data—at least the text and data sections. Referring again to Example 21-2, we see that the compiler has output `.text` and `.data` directives. These are used to tell the assembler in which section it should put the output that follows. It also supplies relocation information for the output file.

String table and name list

Amongst other things, the relocation information includes a significant number of strings. These are stored in the *string table*, which is simply a list of strings terminated with a NUL (`\0`) character. Other parts of the object file refer to strings by their offset in the string table.

As we saw in Example 21-2, the assembler has a directive (`.globl` in this example) that outputs information about externally visible symbols, such as `global_text`. Some assemblers need to be told about external references (such as `_puts` in this example), and others don't, like the GNU assembler *gas* used here. For both external definitions and external references, *gas* outputs an entry for the *symbol table* to the output file with information about the symbol. This symbol table is one of the better-known parts of an object file, and is usually called the *name list*. The structure differs strongly from one flavour of UNIX to the next, but all namelists contain the following information:

- The index of the symbol's name in the *string table*.
- The type of the symbol (undefined, absolute, text, data, bss, common).
- The *value* of the symbol, if it has one (undefined symbols don't, of course).

The library function *nlist* accesses the symbol table and returns a symbol table entry. The call is

```
#include <nlist.h>

int nlist (const char *filename, struct nlist *nl);
```

This function has confusing semantics: the symbol table structure `struct nlist` does not contain the name of the symbol. Instead, it contains a *pointer* to the name of the symbol. On disk, the symbol is located in the string list, but in your program you supply the strings in advance. For the System V.4 ELF format, the structure is

```
struct nlist
  {
  char  *n_name;              /* name of symbol */
  long  n_value;              /* value of symbol */
  short n_scnum;              /* section number */
  unsigned short  n_type;     /* type and derived type */
  char  n_sclass;             /* storage class */
  char  n_numaux;             /* number of auxiliary entries */
  };
```

To use the nlist function, you create an array of struct nlist entries with n_name set to the symbols you are looking for. The last entry contains a null string to indicate the end of the list. nlist searches the symbol table and fills in information for the symbols it finds, and sets all fields except n_name to 0 if it can't find the string.

The return value differs from one system to another:

- If *filename* doesn't exist, or if it isn't an object file, nlist returns -1.
- If all symbols were found, nlist returns 0.
- If some symbols were not found, BSD nlist returns the number of symbols not found. System V nlist still returns 0.

Examining symbol tables: the nm program

You can display the complete symbol table of an object file or an archive with the program nm. Invoke it simply with

```
$ nm filename
```

nm output is frequently used by tools such as shell scripts used during building. This can be a problem, since the format of the printout depends strongly on the object file format. In the following sections we'll look at the differences between *nm* output for *a.out*, COFF and ELF files.

nm display of a.out format

With an *a.out* file, *nm* output looks like:

```
$ nm /usr/lib/libc.a

syscall.o:                              this is the object file name
00000028 T _syscall
         U cerror

sigsuspend.o:
00000030 T _sigsuspend
         U cerror
```

The lines with the file name and colon tell you the name of the archive member (in other words, the object file) from which the following symbols come. The other lines contain a value (which may be missing if it is not defined), a type letter, and a symbol name.

That's all there is to *a.out* symbols. As we will see, *a.out* handles debugging information separately. On the other hand, this means that the type letters are reasonably easy to remember. Upper case represents global symbols, lower case represents local symbols. Table 21-2 gives an overview:

Table 21–2: a.out symbol types

Type letter	Meaning
-	symbol table entries (see the -a flag).
A	absolute symbol (not relocatable)
B	bss segment symbol
C	common symbol
D	data segment symbol
f	file name (always local)
T	text segment symbol
U	undefined

nm display of COFF format

By contrast,, COFF gives something like this:

```
$ nm /usr/lib/libc.a
Symbols from /lib/libc.a[printf.o]:    this is the object file name

Name                  Value   Class        Type         Size   Line  Section

printf.c            |        | file |                 |      |     |
DGROUP              |       0|static|                 |      |     |.data
printf              |       0|extern|                 |      |     |.text
_doprnt             |       0|extern|                 |      |     |
_iob                |       0|extern|                 |      |     |
```

These columns have the following meaning:

- *Name* is the name of the symbol.
- *Value* is the value of the symbol.
- *Class* is the *storage class* of the symbol. There are a large number of storage classes, and the System V.3 man pages don't describe them. See *Understanding and using COFF*, by Gintaras R. Gircys, for a complete list. The one that interests us is `extern` (externally defined symbols).
- In conjunction with *Class*, *Type* describes the type of symbol more accurately, when it is needed. The symbols we're looking at don't need a more accurate description.
- *Size* specifies the size of the entry. This is used in symbolic debug information.
- *Line* is line number information, which is also used for symbolic debug information.
- *Section* is the section to which the symbol belongs. In our example, the familiar `.text` and `.data` occur, but this could be any of the myriad COFF section names.

nm display of ELF format

The differences between COFF and ELF are less obvious:

```
$ nm /lib/libc.so.1

Symbols from /lib/libc.so.1:

[Index]   Value      Size    Type  Bind  Other Shndx   Name

[1]     |         0|       0|FILE |LOCL |0    |ABS    |../../libc.so.1
[2]     |       148|       0|SECT |LOCL |0    |1      |
...  skipping
[32]    |    208976|      12|OBJT |LOCL |0    |12     |libdirs
[33]    |    208972|       4|OBJT |LOCL |0    |12     |rt_dir_list
[34]    |         0|       0|FILE |LOCL |0    |ABS    |dlfcns.c
[35]    |     57456|     384|FUNC |LOCL |0    |8      |dl_delete
[36]    |     56240|      72|FUNC |LOCL |0    |8      |lmExists
[37]    |     56112|     128|FUNC |LOCL |0    |8      |appendLm
[38]    |     56320|     464|FUNC |LOCL |0    |8      |dl_makelist
[39]    |    214960|       4|OBJT |LOCL |0    |15     |dl_tail
```

These columns have the following meanings:

- *Index* is simply the index of the symbol in the symbol list.
- *Value* is the value of the symbol.
- *Size* is the size of the associated object in bytes.
- *Type* is the type of the object. This can be `NOTY` (no type specfi ed), `OBJT` (a data object), `FUNC` (executable code), `SECT` (a section name), or `FILE` (a fi le name).
- *Bind* specifi es the scope of the symbol. `GLOB` specifi es that the symbol is global in scope, `WEAK` specifi es a global symbol with lower precendene, and `LOCL` specifi es a local symbol.
- *Other* is currently unused and contains 0.
- *Shndx* may be `ABS`, specifying an absolute symbol (in other words, not relocatable), `COMMON` specifi es a bss block, and `UNDEF` specifi es a reference to an external symbol. A number in this fi eld is an index into the section table of the section to which the symbol relates. *nm* doesn't tell you which section this is, so the information is not very useful.

Problems with nm output

As we have seen, the output of *nm* depends a lot on the object fi le format. You frequently see shell scripts that use *nm* to look inside a fi le and extract information—they will go seriously wrong if the object fi le format is not what they expect. If there isn't an alternative script to look at your kind of object fi le, you will have to modify it yourself.

Debugging information

Symbolic debuggers have a problem: they relate to the object file, but they want to give the impression that they are working with the source file. For example, the program addresses that interest you are source file line numbers, not absolute addresses, and you want to refer to variables by their names, not their addresses. In addition, you expect the debugger to know the types of variables, so that when you ask the debugger to display a variable of type `char *`, it displays a string, and when you ask it to display a `float`, you get the correct numeric value.

The object file structures we have seen so far don't help too much. Information is available for global symbols, both in the text and data sections, but the type information is not detailed enough to tell the debugger whether a data variable is a `char *` or a `float`. The symbol table information contains no information at all about local variables or line numbers. In addition, the symbol table information goes away at link time, so it wouldn't be of much help anyway.

For these reasons, separate data structures for debugging information were introduced. In the *a.out* format, they have nothing to do with the rest of the file. In COFF and ELF, they are more integrated, but debugging information has one thing in common in all object formats: it is at the end of the file so that it can be easily removed when it is no longer wanted—debugging information can become *very* large. It's not uncommon to see debugging information increase the size of an executable by a factor of 5 or 10. In extreme cases, such as in libraries, it can become 50 times as big.

Frequently you'll see a *make all* that creates an executable with debugging symbols, and a *make install* that installs the same executable but removes the debugging symbols. This process is called *stripping*, and can be done by the program *strip* or by *install* with the `-s` flag. In order to do this, it makes sense for the debugging information to be in its own section at the end of the file, and this is how all object file formats solve the problem.

Debugging information is supplied in the assembler source in the form of *directives*. In Example 21-2, which is from an assembler designed to create *a.out* relocatables, this job is done by the `.stabs`, `.stabn` and `.stabd` directives. These directives are discussed in more detail in Appendix C, *Assembler directives and flags*, on page 421. Let's look at the directives in our example:

- At the beginning of the file there are a lot of `.stabs` directives defining standard data types, so many that we have omitted most of them. The compiler outputs these directives even if the data type isn't used in the program, and they're handy to have in case you want to cast to this data type when debugging.
- Throughout the file you find individual `.stabd 68` directives. These specify that the line number specified in the last parameter starts at this point in the text.
- At the end of the function `_inc`, information about the function itself and the variables associated with it appear in further `.stabs` directives.
- Finally, information about the block structure of the function appears in the `.stabn` directive.

This information is very dependent on the object file format. If you need more information, the best source is the accompanying system documentation.

The linker

You usually encounter the *linker* as the last pass of the C compiler. As the name *ld* implies, the linker was called the *loader* in the Seventh Edition, though all modern systems call it a link editor.* Traditionally, the compiler compiles object files, and then runs the linker to create an executable program. This is logical, since a single executable is always composed of multiple object files,† whereas there is a one-to-one relationship between source files and object modules.

The most important function performed by the linker is symbol resolution. To understand this, we need to define a few terms:

- The *symbol list*, sometimes called a *symbol table*, is an internal data structure where the linker stores information about all symbols whose name it has encountered. It contains the same kind of information about the symbol as we saw in `struct nlist` on page 363.
- An *undefined symbol* is only partially undefined: we know at least its name, but some part of its value is unknown.

Initially, the symbol list is empty, but every file that is included adds to the list. A number of cases can occur:

- The file refers to an undefined symbol. In this case, if the linker has not yet seen this symbol, it enters it into the symbol list and starts a list of references to it. If the symbol is already in the symbol list, the linker adds a reference to the symbol's reference list.
- The file refers to a symbol that has already been defined. In this case, the linker simply performs the required relocation.
- The file defines a symbol. There are three possibilities here:
  - If the symbol has not been encountered before, it is just added to the symbol list.
  - If the symbol is already marked as undefined, the linker updates the symbol information and performs the required relocation for each element in the reference list.
  - If the symbol is known and defined, it is now doubly defined. The linker prints an error message, and will not create an output file.

At the same time as it creates the symbol list, the linker copies data and text sections into the areas it has allocated for them. It copies each individual section to the current end of the area. The symbol list entries reflect these addresses.

* Properly, the loader was the part of the operating system that loaded a program into memory prior to execution. Once, long before the advent of UNIX, the two functions were almost synonymous.
† Even if you supply only a single object file yourself, you need the C startup code in *crt0.o* and library modules from system libraries such as *libc.a*.

Function libraries

Many of the functions you use in linking an executable program are located in *function libraries*, a kind of object file archive built by *ar*. The linker knows about the format of *ar* archives and is able to extract the object files from the archive. The resultant executable contains code from the object files specified on the command line and from the object files found in the libraries. The functions in the libraries are just like any others you may include. They run in user context, not kernel context, and are stored in libraries simply for convenience. We can consider three groups:

- The standard* C library, normally */usr/lib/libc.a*. This library contains at least the functions needed to link simple C programs. It may also contain functions not directly connected with the C language, such as network interface functions—BSD does it this way.
- Additional libraries supporting system functions not directly concerned with the C programming language. Networking functions may also fall into this category—System V does it this way.
- Libraries supporting third party packages, such as the X11 windowing system.

Library search

You can specify object files to the linker in two different ways: you specify that an object file is to be *included* in the output or that a library file is to be *searched* by specifying its name on the command line. The *library search* is one of the most powerful functions performed by the linker. Instead of including the complete library in the output file, the linker checks each object file in the library for definitions of currently undefined symbols. If this is the case, it includes the object file, and not the library. This has a number of implications:

- The linker includes only object files that define symbols referenced by the program, so the program is a lot smaller than it would be if you included the complete library.
- We don't want to include anything that isn't required, so each object file usually defines a single function. In some rare cases, it may define a small number of related functions that always get included together.
- Each object file may refer to other external symbols, so including one file in an archive may require including another one.
- If you compile a library with symbols, each single-function object file will contain debugging information for all the external information defined in the header files. This information is usually many times the size of the function text and data.
- Once the library has been searched, the linker forgets it. This has important consequences which we'll examine on page 373.

For reasons shrouded in history, you don't specify the path name of the library file—instead

* Note the lower-case use of the word *standard*. Whether or not the library conforms to the ANSI/ISO C Standard, it is a standard part of a software development system.

you tell the linker the names of directories that may contain the libraries you are looking for, and a coded representation of the library name. For example, if you want to include */opt/lib/libregex.a* in your search path, you would include `-L/opt/lib -lregex` in your compiler or linker call:

- `-L/opt/lib` tells the linker to include */opt/lib* in the list of directories to search.
- `-lregex` tells the linker to search for the file *libregex.a* in *each* of the directories to search.

This can be a problem if you have four files */usr/lib/libfoo.a*, */usr/lib/libbar.a*, */opt/lib/libfoo.a* and */opt/lib/libbar.a*, and you want to search only */opt/lib/libfoo.a* and */usr/lib/libbar.a*. In this case, you can name the libraries explicitly.

To keep the pain of linking executables down to tolerable levels, the compiler control program (usually *cc*) supplies a few library paths and specifications for free—normally the equivalent of `-L/usr/lib -lc`, which at least finds the library */usr/lib/libc.a*, and also supplies the path to all other libraries in */usr/lib*. You need only specify *additional* paths and libraries. Occasionally this behaviour is undesirable: what if you deliberately want to exclude the standard libraries, like if you're building an executable for a different version of the operating system? Some compilers give you an option to forget these libraries. For example, in *gcc* it is `-nostdlib`.

Like most aspects of UNIX, there is no complete agreement on where to store library files, but most systems come close to the following arrangement:

- */usr/lib* contains the basic system libraries as well as startup code like *crt0.o* and friends, which are bound in to supply low-level support for the C language. We'll look at this in the next section.
- Some of these files used to be stored in */lib*. Nowadays */lib* tends either not to be present or, for compatibility's sake, it is a symlink to */usr/lib*.
- System V.4 systems place BSD compatibility libraries in */usr/ucblib*[*]. Many of these functions duplicate functions in */usr/lib*.
- */usr/X11/lib*, */usr/X/lib*, */usr/lib/X11*, */usr/lib/X11R6* and others are some of the places that the X11 libraries might be hidden. This directory probably contains all the parts of X11 and related code that are on your system.

Shared libraries

Some libraries can be very big. The X11R6 version *libX11.a*, the standard X11 functions, runs to 630 kB on BSD/OS. The Motif library *libXm.a* is nearly 1.4 MB in size. This can lead to enormous executables, even if the program itself is relatively small—the "*500 kB Hello world*" syndrome. Since these functions are used in many programs, many copies of a function may be active at any one time in the system. For example, just about every program

* UCB stands for the University of California at Berkeley, the home of the Berkeley Software Distributions. You'll frequently find BSD-derived software stored in directories whose names start with the letters *ucb*.

uses the function `printf`, which with its auxiliary functions can be quite big. To combat this, modern UNIX flavours support *shared libraries*: the library itself is no smaller, but it is in memory only once.

Two different library schemes are in current use: *static shared libraries*[*] and *dynamic shared libraries*. Static shared libraries contain code which has been linked to run at a specific address, which means that you could have difficulties if your program refers to two libraries with overlapping address ranges, or if you use a different version of the library with functions at slightly different addresses. Dynamic libraries get round this problem by linking at run time, which requires a *dynamic linker*. Unless you're building shared libraries, a topic beyond the scope of this book, you don't need to worry about the difference between the two. If you do find yourself in the situation where you need to build shared libraries, your best source of information is your operating system documentation.

A shared library performs two different functions:

- When you link your program, it supplies information about the locations of the functions and data in the library. Some systems, such as SunOS 4, supply a "stub" file with a name like *libc.sa.1.9*. Since it does not contain the code of the functions, it is relatively small—on SunOS 4.1.3, it is 7996 bytes long. Other systems, such as System V.4, only supply a single library file with a name like *libc.so*. The linker only includes enough information for the dynamic loader to locate the functions in the library file at run time.
- At run time, it supplies the functions and data. On all systems, the file name is of the form *libc.so.1.9*

It's important to ensure that you use the same library to perform these two actions. If a function or a data structure changes between versions of the library, a program written for a different version may work badly or crash. This is a common problem: most programs are distributed in executable form, and thus contain preconceived notions about what the library looks like. Since we're linking the program ourselves, we should not run in to this problem. If you do run into problems, you can always fall back to static (unshared) libraries.

Other linker input

In addition to the user-specified object files and libraries, the C programming language requires a few auxiliary routines to set up its run-time environment. These are stored in one or more auxiliary object files in a place known to the compiler, usually */usr/lib*. For example, in the System V.4 example above, we can see how the compiler control program starts the linker if we use the `-v` flag to the compiler:

```
$ gcc -v -u baz -o foo foo.o -L.  -lbaz -lbar
/usr/ccs/bin/ld -V -Y P,/usr/ccs/lib:/usr/lib -Qy -o foo -u baz
/usr/ccs/lib/crt1.o /usr/ccs/lib/crti.o /usr/ccs/lib/values-Xa.o
/opt/lib/gcc-lib/i386-unknown-sysv4.2/2.5.8/crtbegin.o
-L.  -L/opt/lib/gcc-lib/i386-unknown-sysv4.2/2.5.8 -L/usr/ccs/bin
-L/usr/ccs/lib -L/opt/lib foo.o -lbaz -lbar -lgcc -lc
```

* Don't confuse static shared libraries with the term *static libraries*, which are traditional, non-shared libraries.

```
/opt/lib/gcc-lib/i386-unknown-sysv4.2/2.5.8/crtend.o
/usr/ccs/lib/crtn.o -lgcc
```

The same example in BSD/OS specifies the files */usr/lib/crt0.o*, *foo.o*, *-lbar*, *-lbaz*, *-lgcc*, *-lc* and *-lgcc*—only fractionally more readable. This example should make it clear why almost nobody starts the linker directly.

Merging relocatable files

Occasionally you want to merge a number of object files into one large file. We've seen one way of doing that: create an object file library with *ar*. You can also use the linker to create an object file. Which you choose depends on why you want to make the file. If you are creating a function library, use *ar*. As we have seen, the linker includes individual object files from the archive. It also happily includes a relocatable object created by a previous invocation of the linker, but in this case it includes the complete object, even if you don't need all the functions.

You don't often need to create relocatable objects with the linker: the only real advantage over a library is that the resultant object is smaller and links faster. If you want to do it, you specify a flag, normally `-r`. For example,

```
$ ld -r foo.o bar.o baz.o -o foobarbaz.o
```

This links the three object files *foo.o*, *bar.o* and *baz.o* and creates a new object file *foobarbaz.o* that contains all the functions and data in the three input files.

Problems with the link editor

Once you have compiled all your objects, you're still not home. There are plenty of things that can go wrong with the linkage step. In this section we'll look at some of the more common problems.

Invalid linker flags

You normally invoke the linker via the compiler rather than calling it directly. This is a good idea, as we saw in the previous section. If you have a *Makefile* with an explicit linker call, and you run into trouble with linker flags, and the system documentation doesn't help, consider replacing the linker invocation with a compiler invocation.

Invalid object files

Occasionally, the package you are building may already contain object files. It's unlikely that you can to use them, but *make* is far too simplistic to notice the difference, and the result is usually some kind of message from the linker saying that it can't figure out what kind of file this is. If you're in doubt, use the *file* command:

```
$ file *.o
gram.o:  386 executable not stripped
main.o:  ELF 32-bit LSB relocatable 80386 Version 1
scan.o:  sparc executable not stripped
```

```
util.o:  80386 COFF executable not stripped - version 30821
```

Here are four different kinds of object files in the same directory. Occasionally, you will see files like this that are there for a good reason: due to license reasons, there are no corresponding sources, and there will be one object for each architecture that the package supports. In this example, however, the file names are different enough that you can be reasonably sure that these files are junk left behind from previous builds. If the object files are still there after a *make clean*, you should remove them manually (and fix the *Makefile*).

Suboptimal link order

We have seen that the linker takes all objects it finds and puts their code and data into the code and data segments in the order in which they appear. From the point of view of logic flow, this works fine, but it can have significant performance implications on modern machines. You might find that 95% of the execution time of a program is taken up by 5% of the code. If this code is located contiguously, it will probably fit into the cache of any modern machine. If, on the other hand, it is scattered throughout memory, it will require much more cache, possibly more than the machine can supply. This can result in a dramatic drop in performance.

Most linkers do not help you much in arranging functions. The simplest way is to put one function in a file, like you do in an archive, and specify them in sequence in the linker invocation. For example, if you have five functions `foo`, `bar`, `baz`, `zot`, and `glarp`, and you have determined that you need three functions next to each other in the sequence `foo`, `glarp` and `zot`, you can invoke the linker with:

```
$ cc -o foobar foo.o glarp.o zot.o bar.o baz.o
```

Missing functions

The UNIX library mechanism works well and is reasonably standardized from one platform to the next. The main problem you are likely to encounter is that the linker can't find a function that the program references. There can be a number of reasons for this:

- The symbol may not be a function name at all, but a reference to an undefined preprocessor variable. For example, in *xfm* version 1.1, the source file *FmInfo.c* contains:

  ```
  if (S_ISDIR(mode))
    type = "Directory";
  else if (S_ISCHR(mode))
    type = "Character special file";
  else if(S_ISBLK(mode))
    type = "Block special file";
  else if(S_ISREG(mode))
    type = "Ordinary file";
  else if(S_ISSOCK(mode))
    type = "Socket";
  else if(S_ISFIFO(mode))
    type = "Pipe or FIFO special file";
  ```

  sys/stat.h defines the macros of the form `S_IS`*foo*. They test the file mode bits for specific file types. System V does not define `S_ISSOCK` (the kernel doesn't have sockets), so

a pre-ANSI compiler assumes that S_ISSOCK is a reference to an external function. The module compiles correctly, but the linker fails with an undefined reference to S_ISSOCK. The obvious solution here is conditional compilation, since S_ISSOCK is a preprocessor macro, and you can test for it directly with #ifdef:

```
        type = "Ordinary file";
#ifdef S_ISSOCK
    else if(S_ISSOCK(mode))
        type = "Socket";
#endif
    else if(S_ISFIFO(mode))
```

- The function is in a different library, and you need to specify it to the linker. A good example is the networking code we mentioned on page 369: a reference to socket will link just fine with no additional libraries on a BSD platform, but on some versions of System V.3 you will need to specify -linet, and on System V.4 and other versions of System V.3 you will need to specify -lsocket. The *findf* script can help here. It uses *nm* to output symbol information from the files specified in LIBS, and searches the output for a function definition whose name matches the parameter supplied. The search parameter is a regular expression, so you can search for a number of functions at once. For example, to search for strcasecmp and strncasecmp, you might enter:

```
$ findf str.*casecmp
/usr/lib/libc.a(strcasecmp.o):  _strcasecmp
/usr/lib/libc.a(strcasecmp.o)): _strncasecmp
/usr/lib/libc_p.a(strcasecmp.po):       _strcasecmp
/usr/lib/libc_p.a(strcasecmp.po)):      _strncasecmp
```

Because of the differences in *nm* output format, *findf* looks very different on BSD systems and on System V. You may find that you need to modify the script to work on your system. Example 21-3 shows a version for 4.4BSD:

Example 21–3:

```
LIBS="/usr/lib/lib* /usr/X11R6/lib/lib*"
nm $LIBS 2>/dev/null \
  | awk -v fun=$1 \
  '/^\// {file = $1};
  /^[^\/].*:/{member = $1};
  $3 ~ fun && $2 ~ /T/ {
  sub (":$", "", file); ;  sub (":$", "):", member); print file "(" member "\t" $3}'
```

On a system like System V.4, which uses ELF format, the corresponding script is in Example 21-4:

Example 21–4:

```
LIBS="/usr/lib/lib* /usr/X11R6/lib/lib*"
nm $LIBS 2>/dev/null \
  | sed 's:|: :g' \
  | gawk -v fun=$1 \
  '/^Symbols from/ {file = $3};
  $8 ~ fun && $4 ~ /FUNC/ { print file member "\t" $8 }'
```

Example 21–4: (continued)

Some versions of System V *awk* have difficulty with this script, which is why this version uses GNU *awk*.

- The function is written in a different language, and the internal name differs from what the compiler expected. This commonly occurs when you try to call a C function from C++ and forget to tell the C++ compiler that the called function is written in C. We discussed this in Chapter 17, *Header files*, page 285.
- The function is part of the package, but has not been compiled because a configuration parameter is set incorrectly. For example, *xpm*, a pixmap conversion program, uses `strcasecmp`. Knowing that it is not available on all platforms, the author included the function in the package, but it gets compiled only if the Makefile contains the compiler flag `-DNEED_STRCASECMP`.
- The function is supplied in a library within the package, but the *Makefile* is in error and tries to reference the library before it has built it. You wouldn't expect this ever to happen, since it is the purpose of *Makefile*s to avoid this kind of problem, but it happens often enough to be annoying. It's also not always immediately obvious that this is the cause—if you suspect that this is the reason, but are not sure, the best thing is to try to build all libraries first and see if that helps.
- The function is really not supplied in your system libraries. In this case, you will need to find an alternative. We looked at this problem in detail in Chapter 18, *Function libraries*.
- The first reference to a symbol comes after the linker has searched the library in which it is located.

Let's look at the last problem in more detail: when the linker finishes searching a library, it continues with the following file specifications. It is possible that another file later in the list will refer to an object file contained in the library which was not included in the executable. In this case, the symbol will not be found. Consider the following three files:

foo.c
```
main ()
  {
  bar ("Hello");
  }
```

bar.c
```
void bar (char *c)
  {
  baz (c);
  }
```

baz.c
```
void baz (char *c)
  {
  puts (c);
  }
```

We compile them to the corresponding object files, and then make libraries *libbar.a* and *libbaz.a*, which contain just the single object file *bar.o* and *baz.o* respectively. Then we try to link:

```
$ gcc -c foo.c
$ gcc -c bar.c
$ gcc -c baz.c
$ ar r libbaz.a baz.o
$ ar r libbar.a bar.o
$ gcc -o foo foo.o -L.  -lbaz -lbar
Undefined                       first referenced
 symbol                             in file
baz                                 ./libbar.a(bar.o)
ld: foo: fatal error: Symbol referencing errors.  No output written to foo
$ gcc -o foo foo.o -L.  -lbar -lbaz
$
```

In the first link attempt, the linker included *foo.o*, then searched *libbaz.a* and didn't find anything of interest. Then it went on to *libbar.a* and found it needed the symbol `baz`, but by that time it was too late. You can solve the problem by putting the reference `-lbar` before `-lbaz`.

This problem is not even as simple as it seems: although it's bad practice, you sometimes find that libraries contain mutual references. If *libbar.a* also contained an object file *zot.o*, and `baz` referred to it, you would have to link with:

```
$ gcc -o foo foo.o -L.  -lbar -lbaz -lbar
```

An alternative seems even more of a kludge: with the `-u` flag, the linker will enter an undefined symbol in its symbol table. In this example, we could also have written

```
$ gcc -u baz -o foo foo.o -L.  -lbaz -lbar
$
```

Dumping to object files

Some programs need to perform significant processing during initialization. For example, *emacs* macros are written in emacs LISP, and they take some time to load. Startup would be faster if they were already in memory when the program is started. The only normal way to have them in memory is to compile them in, and it's very difficult to initialize data at compile time as intricately as a program like *emacs* does it at run time.

The solution chosen is simple in concept: *emacs* does it once at run time. Then it dumps itself to disk in object file format: it copies the text section directly from its own text area, since there is no way it can be changed, and it writes the data section from its current data area, including all of what used to be bss. It doesn't need to copy the stack section, since it will be recreated on initialization.

This rather daring approach works surprisingly well as long as *emacs* knows its own object file format. UNIX doesn't provide any way to find out, since there is normally no reason why a program *should* know its own object file format. The result can be problems when porting a package like this to a system with a different object format: the port runs fine until the first

executable dumps, but the dumped executable does not have a format that the kernel can recognize.

Other programs that use this technique are *gcl* (GNU common LISP) and TEX.

Process initialization and stack frames

In Chapter 12, *Kernel dependencies*, page 168, we examined the myriad flavours of `exec`. They all pass arguments and environment information to the newly loaded program. From a C program viewpoint, the arguments are passed as a parameter to `main`, and the environment is just there for the picking. In this section we'll look more closely at what goes on between `exec` and `main`. In order to understand this, we need to look more closely at parameter passing.

Stack frames

Most modern machines have a stack-oriented architecture, even if the support is rather rudimentary in some cases. Everybody knows what a stack is, but here we'll use a more restrictive definition: a *stack* is a linear list of storage elements, each relating to a particular function invocation. These are called *stack frames*. Each stack frame contains

- The parameters with which the function was invoked.
- The address to which to return when the function is complete.
- Saved register contents.
- Variables local to the function.
- The address of the previous stack frame.

With the exception of the return address, any of these fields may be omitted.* Typical stack implementations supply two hardware registers to address the stack:

- The *stack pointer* points to the last used word of the stack.
- The *frame pointer* points to somewhere in the middle of the stack frame.

The resultant memory image looks like:

* Debuggers recognize stack frames by the frame pointer. If you don't save the frame pointer, it will still be pointing to the previous frame, so the debugger will report that you are in the previous function. This frequently happens in system call linkage functions, which typically do not save a stack linkage, or on the very first instruction of a function, before the linkage has been built. In addition, some optimizers remove the stack frame.

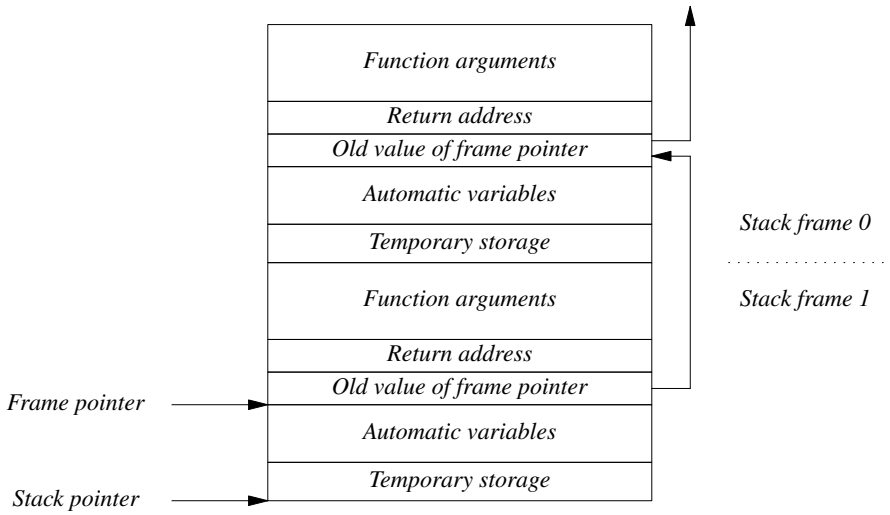

Figure 21–1. Function stack frame

The individual parts of the stack frames are built at various times. In the following sections, we'll see how the stack gets set up and freed.

Setting up the initial parameters

`exec` builds the initial stack. The exact details are implementation-dependent, but most come close to the way BSD/OS does it, so we'll look at that.

The stack is always allocated at a fixed point in memory, `0xefbfe000` in the case of BSD/OS, and grows downwards, like the stacks on almost every modern architecture. At the very top of stack, is structure with information for the *ps* program:

```
struct ps_strings
  {
  char    **ps_argv;          /* first of 0 or more argument pointers */
  int     ps_argc;            /* the number of argument pointers */
  char    **ps_envp;          /* first of 0 or more environment pointers */
  int     ps_nenv;            /* the number of environment pointers */
};
```

This structure is supplied for convenience and is not strictly necessary. Many systems, for example FreeBSD, do not define it.

Next, `exec` places on the stack all environment variable strings, followed by all the program arguments. Some systems severely limit the maximum size of these strings—we looked at the problems that that can cause in Chapter 5, *Building the package*, page 74.

After the variable strings come two sets of `NULL`-terminated pointers, the first to the environment variables, the second to the program arguments.

Finally comes the number of arguments to `main`, the well-known parameter `argc`. At this point, the stack looks like:

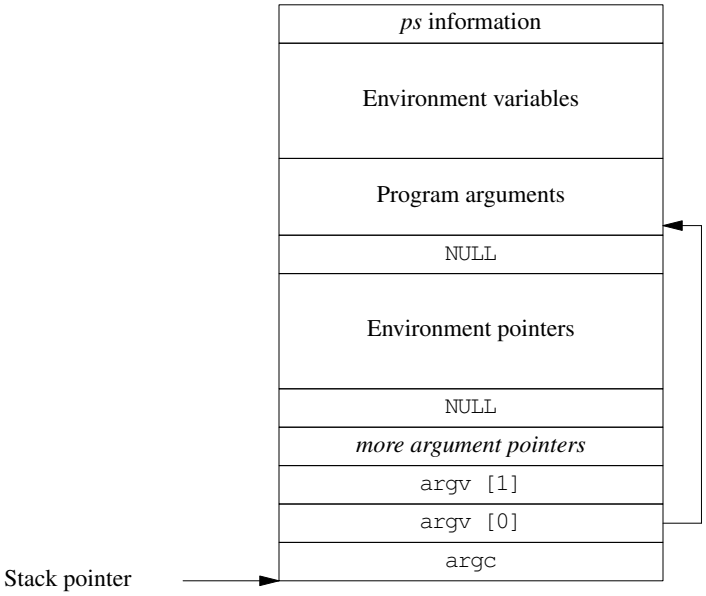

Figure 21–2. Stack frame at start of program

At this point, all the data for `main` is on the stack, but it's not quite in the form that `main` needs. In addition, there's no return address. But where could `main` return to? All this work has been done in the kernel by `exec`: we need to return to a place in the program. These problems are solved by the function `start`, the *real* beginning of the program: it calls `main`, and then calls `exit` with the return value from `main`. Before doing so, it may perform some runtime initializations. A minimal `start` function looks like this stripped down version of GNU libc *start.c*, which is the unlikely name of the source file for *crt0.o*:

```
static void start (int argc, char *argp)
{
  char **argv = &argp;        /* set up a pointer to the first argument pointer */
  __environ = &argv [argc + 1]; /* The environment starts just after argv */
  asm ("call L_init");          /* call the .init section */
  __libc_init (argc, argv, __environ); /* Do C and C++ library initializations  */
  exit (main (argc, argv, __environ)); /* Call the user program */
}
```

The `asm` directive is used for C++ initialization—we'll look at that on page 380. But what's this? `start` calls `main` with three parameters! The third is the address of the environment variable pointers. This is one of the best kept secrets in UNIX: `main` really has three parameters:

```
int main (int argc, char *argv [], char *envp []);
```

It isn't documented anywhere, but it's been there at least since the Seventh Edition and it's unlikely to go away, since there isn't really any other good place to store the environment variables.

By the time we have saved the stack linkage in `main`, the top of the stack looks like:

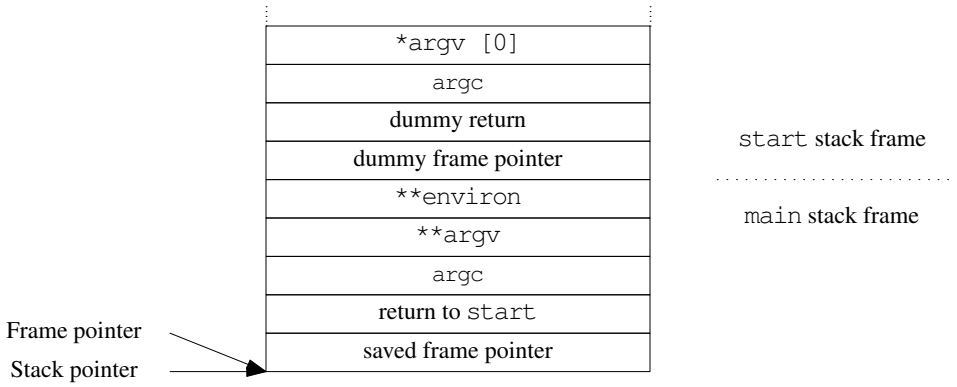

Figure 21–3. Stack frame after entering main

Initializing C++ programs

What we've seen here is not enough for C++: before entering `main`, the program may need to initialize global class instances. Since this is system library code, it can't know what global classes exist. The solution depends on the system:

- System V systems place this information in a special section, `.init`. The initialization file *crtn.o* contains a default `.init` section containing a single return instruction. If a C++ program has global initializers, it will create an `.init` section to initialize them. If any object module before *ctrn.o* has an `.init` section, it will be included before the `.init` section in *ctrn.o*. During program initialization, the function `start` calls the `.init` section to execute the global constructors—this is the purpose of the `asm` directive on page 379.
- Systems based on *a.out* formats do not have this luxury. Instead, they compile special code into `main` to call the appropriate constructors.

The difference between these two approaches can be important if you are debugging a C++ program which dies in the global constructors.

Stack growth during function calls

Now that we have an initial stack, let's see how it grows and shrinks during a function call. We'll consider the following simple C program compiled on the i386 architecture:

```
foo (int a, int b)
{
  int c = a * b;
  int d = a / b;
  printf ("%d %d0, c, d);
  }

main (int argc, char *argv [])
{
  int x = 4;
  int y = 5;
  foo (y, x);
  }
```

The assembler code for the calling sequence for `foo` in `main` is:

```
pushl -4(%ebp)          value of x
pushl -8(%ebp)          value of y
call _foo               call the function
addl $8,%esp            and remove parameters
```

Register `ebp` is the *base pointer*, which we call the frame pointer. `esp` is the stack pointer.

The `push` instructions decrement the stack pointer and then place the word values of `x` and `y` at the location to which the stack pointer now points.

The `call` instruction pushes the contents of the current instruction pointer (the address of the instruction following the `call` instruction) onto the stack, thus saving the return address, and loads the instruction pointer with the address of the function. We now have:

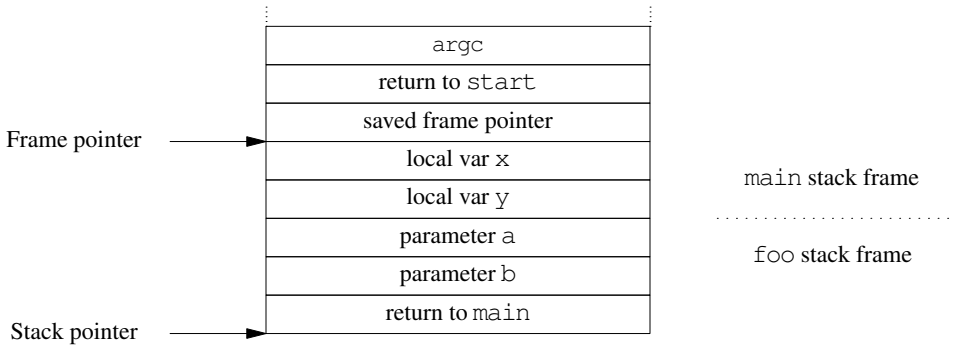

Figure 21–4. Stack frame after `call` instruction

The called function `foo` saves the frame pointer (in this architecture, the register is called *ebp*, for *extended base pointer*), and loads it with the current value of the stack pointer register *esp*.

```
_foo:   pushl %ebp              save ebp on stack
        movl %esp,%ebp          and load with current value of esp
```

At this point, the stack linkage is complete, and this is where most debuggers normally set a breakpoint when you request on to be placed at the entry to a function.

Next, `foo` creates local storage for `c` and `d`. They are each 4 bytes long, so it subtracts 8 from the *esp* register to make space for them. Finally, it saves the register *ebx*—the compiler has decided that it will need this register in this function.

```
        subl $8,%esp            create two words on stack
        pushl %ebx              and save ebx register
```

At this point, our stack is now complete and looks like the diagram on page 377:

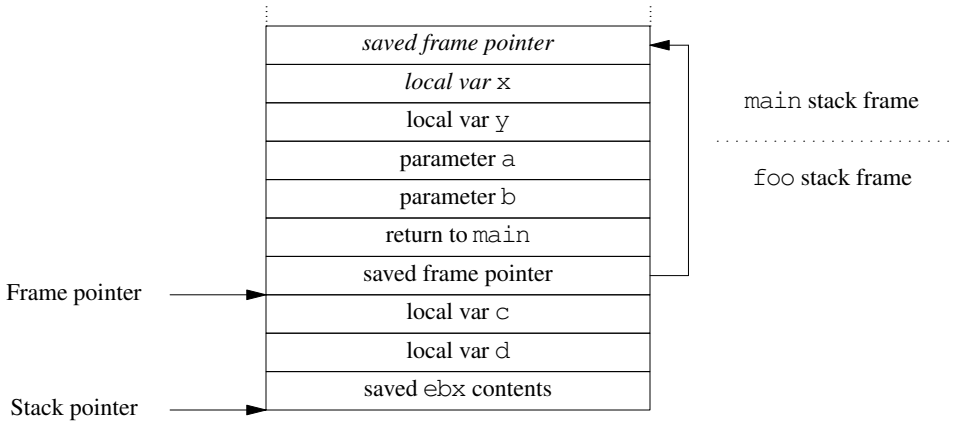

Figure 21–5. Complete stack frame after entering called function

The frame pointer isn't absolutely necessary: you can get by without it and refer to the stack pointer instead. The problem is that during the execution of the function, the compiler may save further temporary information on the stack, so it's difficult to keep track of the value of the stack pointer—that's why most architectures use a frame pointer, which *does* stay constant during the execution of the function. Some optimizers, including newer versions of *gcc*, give you the option of compiling without a stack frame. This makes debugging almost impossible.

On return from the function, the sequence is reversed:

```
        movl -12(%ebp),%ebx     and restore register ebx
        leave                   reload ebp and esp
        ret                     and return
```

The first instruction reloads the saved register *ebx*, which could be stored anywhere in the stack. This instruction does not modify the stack.

The *leave* instruction loads the stack pointer *esp* from the frame pointer *ebp*, which effectively discards the part stack below the saved *ebp* value. Then it loads *ebp* with the contents of the

word to which it points, the saved *ebp*, effectively reversing the stack linkage. The stack now looks like it did on entry.

Next, the *ret* instruction pops the return address into the instruction pointer, causing the next instruction to be fetched from the address following the *call* instruction in the calling function.

The function parameters `x` and `y` are still on the stack, so the next instruction in the calling function removes them by adding to the stack pointer:

```
addl $8,%esp            and remove parameters
```

Object Archive formats

As we have seen, object files are frequently collected into *libraries*, which from our current point of view are *archives* maintained by *ar*. One of the biggest problems with *ar* archives is that there are so many different forms. You're likely to come across the following ones:

- The so-called *common archive format*. This format starts with the magic string `!<arch>\n`. It is used both in System V.4 and in BSD versions since 4BSD.
- The *PORT5AR* format, which starts with the magic string `<ar>`. System V.3 defines it, but doesn't use it.
- The Seventh Edition archive, which starts with the magic number `0177545`. It is also used by XENIX and System V.2 systems. In System V.3 systems, *file* reports this format as `x.out randomized archive`. 4.4BSD *file* refers to it as *old PDP-11 archive*, while many System V.4 *file*s don't recognize it at all. UnixWare is one that does, and it calls it a *pdp11/pre System V ar archive*.

As long as you stick to modern systems, the only archive type you're likely to come across is the common archive format. If you do find one of the others, you should remember that it's not the archive that interests you, it's the contents. If you have another way to get the contents, like the original object or source files, you don't need to worry about the archive.