Linkers

Ian Lance Taylor

2007

Contents

1 Part 1

I've been working on and off on a new linker. To my surprise, I've discovered in talking about this that some people, even some computer programmers, are unfamiliar with the details of the linking process. I've decided to write some notes about linkers, with the goal of producing an essay similar to my existing one about the GNU configure and build system.

As I only have the time to write one thing a day, I'm going to do this on my blog over time, and gather the final essay together later. I believe that I may be up to five readers, and I hope y'all will accept this digression into stuff that matters. I will return to random philosophizing and minding other people's business soon enough.

1.1 A Personal Introduction

Who am I to write about linkers?

I wrote my first linker back in 1988, for the AMOS operating system which ran on Alpha Micro systems. (If you don't understand the following description, don't worry; all will be explained below). I used a single global database to register all symbols. Object files were checked into the database after they had been compiled. The link process mainly required identifying the object file holding the main function. Other objects files were pulled in by reference. I reverse engineered the object file format, which was undocumented but quite simple. The goal of all this was speed, and indeed this linker was much faster than the system one, mainly because of the speed of the database.

I wrote my second linker in 1993 and 1994. This linker was designed and prototyped by Steve Chamberlain while we both worked at Cygnus Support (later Cygnus Solutions, later part of Red Hat). This was a complete reimplementation of the BFD based linker which Steve had written a couple of years before. The primary target was a.out and COFF. Again the goal was speed, especially compared to the original BFD based linker. On SunOS 4 this linker was almost as fast as running the cat program on the input .o files.

The linker I am now working, called gold, on will be my third. It is exclusively an ELF linker. Once again, the goal is speed, in this case being faster than

my second linker. That linker has been significantly slowed down over the years by adding support for ELF and for shared libraries. This support was patched in rather than being designed in. Future plans for the new linker include support for incremental linking–which is another way of increasing speed.

There is an obvious pattern here: everybody wants linkers to be faster. This is because the job which a linker does is uninteresting. The linker is a speed bump for a developer, a process which takes a relatively long time but adds no real value. So why do we have linkers at all? That brings us to our next topic.

1.2 A Technical Introduction

What does a linker do?

It's simple: a linker converts object files into executables and shared libraries. Let's look at what that means. For cases where a linker is used, the software development process consists of writing program code in some language: e.g., C or C++ or Fortran (but typically not Java, as Java normally works differently, using a loader rather than a linker). A compiler translates this program code, which is human readable text, into into another form of human readable text known as assembly code. Assembly code is a readable form of the machine language which the computer can execute directly. An assembler is used to turn this assembly code into an object file. For completeness, I'll note that some compilers include an assembler internally, and produce an object file directly. Either way, this is where things get interesting.

In the old days, when dinosaurs roamed the data centers, many programs were complete in themselves. In those days there was generally no compiler–people wrote directly in assembly code–and the assembler actually generated an executable file which the machine could execute directly. As languages liked Fortran and Cobol started to appear, people began to think in terms of libraries of subroutines, which meant that there had to be some way to run the assembler at two different times, and combine the output into a single executable file. This required the assembler to generate a different type of output, which became known as an object file (I have no idea where this name came from). And a new program was required to combine different object files together into a single executable. This new program became known as the linker (the source of this name should be obvious).

Linkers still do the same job today. In the decades that followed, one new feature has been added: shared libraries.

2 Part 2

I'm back, and I'm still doing the linker technical introduction.

Shared libraries were invented as an optimization for virtual memory systems running many processes simultaneously. People noticed that there is a set of basic functions which appear in almost every program. Before shared libraries, in a system which runs multiple processes simultaneously, that meant that almost every process had a copy of exactly the same code. This suggested that on a virtual memory system it would be possible to arrange that code so that a single copy could be shared by every process using it. The virtual memory system would be used to map the single copy into the address space of each process which needed it. This would require less physical memory to run multiple programs, and thus yield better performance.

I believe the first implementation of shared libraries was on SVR3, based on COFF. This implementation was simple, and basically assigned each shared library a fixed portion of the virtual address space. This did not require any significant changes to the linker. However, requiring each shared library to reserve an appropriate portion of the virtual address space was inconvenient.

SunOS4 introduced a more flexible version of shared libraries, which was later picked up by SVR4. This implementation postponed some of the operation of the linker to runtime. When the program started, it would automatically run a limited version of the linker which would link the program proper with the shared libraries. The version of the linker which runs when the program starts is known as the *dynamic linker*. When it is necessary to distinguish them, I will refer to the version of the linker which creates the program as the *program linker*. This type of shared libraries was a significant change to the traditional program linker: it now had to build linking information which could be used efficiently at runtime by the dynamic linker.

That is the end of the introduction. You should now understand the basics of what a linker does. I will now turn to how it does it.

2.1 Basic Linker Data Types

The linker operates on a small number of basic data types: *symbols*, *relocations*, and *contents*. These are defined in the input object files. Here is an overview of each of these.

A symbol is basically a name and a value. Many symbols represent static objects in the original source code–that is, objects which exist in a single place for the duration of the program. For example, in an object file generated from C code, there will be a symbol for each function and for each global and static variable. The value of such a symbol is simply an offset into the contents. This type of symbol is known as a *defined* symbol. It's important not to confuse the value of the symbol representing the variable `my_global_var` with the value of `my_global_var` itself. The value of the symbol is roughly the address of the variable: the value you would get from the expression `&my_global_var` in C.

Symbols are also used to indicate a reference to a name defined in a different object file. Such a reference is known as an *undefined* symbol. There are other less commonly used types of symbols which I will describe later.

During the linking process, the linker will assign an address to each defined symbol, and will *resolve* each undefined symbol by finding a defined symbol with the same name.

A relocation is a computation to perform on the contents. Most relocations refer to a symbol and to an offset within the contents. Many relocations will also provide an additional operand, known as the *addend*. A simple, and commonly used, relocation is "set this location in the contents to the value of this symbol plus this addend". The types of computations that relocations do are inherently dependent on the architecture of the processor for which the linker is generating code. For example, RISC processors which require two or more instructions to form a memory address will have separate relocations to be used with each of those instructions; for example, "set this location in the contents to the lower 16 bits of the value of this symbol".

During the linking process, the linker will perform all of the relocation computations as directed. A relocation in an object file may refer to an undefined symbol. If the linker is unable to resolve that symbol, it will normally issue an error (but not always: for some symbol types or some relocation types an error may not be appropriate). The contents are what memory should look like during

the execution of the program. Contents have a size, an array of bytes, and a type. They contain the machine code generated by the compiler and assembler (known as *text*). They contain the values of initialized variables (*data*). They contain static unnamed data like string constants and switch tables (read-only data or *rdata*). They contain uninitialized variables, in which case the array of bytes is generally omitted and assumed to contain only zeroes (*bss*). The compiler and the assembler work hard to generate exactly the right contents, but the linker really doesn't care about them except as raw data. The linker reads the contents from each file, concatenates them all together sorted by type, applies the relocations, and writes the result into the executable file.

2.2 Basic Linker Operation

At this point we already know enough to understand the basic steps used by every linker.

- Read the input object files. Determine the length and type of the contents. Read the symbols.
- Build a symbol table containing all the symbols, linking undefined symbols to their definitions.
- Decide where all the contents should go in the output executable file, which means deciding where they should go in memory when the program runs.
- Read the contents data and the relocations. Apply the relocations to the contents. Write the result to the output file.
- Optionally write out the complete symbol table with the final values of the symbols.

3 Part 3

3.1 Address Spaces

An address space is simply a view of memory, in which each byte has an address. The linker deals with three distinct types of address space.

Every input object file is a small address space: the contents have addresses, and the symbols and relocations refer to the contents by addresses.

The output program will be placed at some location in memory when it runs. This is the output address space, which I generally refer to as using *virtual memory addresses*.

The output program will be loaded at some location in memory. This is the *load memory address*. On typical Unix systems virtual memory addresses and load memory addresses are the same. On embedded systems they are often different; for example, the initialized data (the initial contents of global or static variables) may be loaded into ROM at the load memory address, and then copied into RAM at the virtual memory address.

Shared libraries can normally be run at different virtual memory address in different processes. A shared library has a *base address* when it is created; this is often simply zero. When the dynamic linker copies the shared library into the virtual memory space of a process, it must apply relocations to adjust the shared library to run at its virtual memory address. Shared library systems minimize the number of relocations which must be applied, since they take time when starting the program.

3.2 Object File Formats

As I said above, an assembler turns human readable assembly language into an object file. An object file is a binary data file written in a format designed as input to the linker. The linker generates an executable file. This executable file is a binary data file written in a format designed as input for the operating system or the loader (this is true even when linking dynamically, as normally the operating system loads the executable before invoking the dynamic linker to begin running the program). There is no logical requirement that the object file format resemble the executable file format. However, in practice they are normally very similar.

Most object file formats define *sections*. A section typically holds memory contents, or it may be used to hold other types of data. Sections generally have a name, a type, a size, an address, and an associated array of data.

Object file formats may be classed in two general types: record oriented and section oriented.

A record oriented object file format defines a series of records of varying size. Each record starts with some special code, and may be followed by data. Reading the object file requires reading it from the begininng and processing each record. Records are used to describe symbols and sections. Relocations may be associated with sections or may be specified by other records. IEEE-695 and Mach-O are record oriented object file formats used today.

In a section oriented object file format the file header describes a section table with a specified number of sections. Symbols may appear in a separate part of the object file described by the file header, or they may appear in a special section. Relocations may be attached to sections, or they may appear in separate sections. The object file may be read by reading the section table, and then reading specific sections directly. ELF, COFF, PE, and a.out are section oriented object file formats.

Every object file format needs to be able to represent debugging information. Debugging informations is generated by the compiler and read by the debugger. In general the linker can just treat it like any other type of data. However, in practice the debugging information for a program can be larger than the actual program itself. The linker can use various techniques to reduce the amount of debugging information, thus reducing the size of the executable. This can speed up the link, but requires the linker to understand the debugging information.

The a.out object file format stores debugging information using special strings in the symbol table, known as stabs. These special strings are simply the names of symbols with a special type. This technique is also used by some variants of ECOFF, and by older versions of Mach-O.

The COFF object file format stores debugging information using special fields in the symbol table. This type information is limited, and is completely inadequate for C++. A common technique to work around these limitations is to embed stabs strings in a COFF section.

The ELF object file format stores debugging information in sections with special names. The debugging information can be stabs strings or the DWARF debugging format.

4 Part 4

4.1 Shared Libraries

We've talked a bit about what object files and executables look like, so what do shared libraries look like? I'm going to focus on ELF shared libraries as used in SVR4 (and GNU/Linux, etc.), as they are the most flexible shared library implementation and the one I know best.

Windows shared libraries, known as DLLs, are less flexible in that you have to compile code differently depending on whether it will go into a shared library or not. You also have to express symbol visibility in the source code. This is not inherently bad, and indeed ELF has picked up some of these ideas over time, but the ELF format makes more decisions at link time and is thus more powerful.

When the program linker creates a shared library, it does not yet know which virtual address that shared library will run at. In fact, in different processes, the same shared library will run at different address, depending on the decisions made by the dynamic linker. This means that shared library code must be *position independent*. More precisely, it must be position independent after the dynamic linker has finished loading it. It is always possible for the dynamic linker to convert any piece of code to run at any virtula address, given sufficient relocation information. However, performing the reloc computations must be done every time the program starts, implying that it will start more slowly. Therefore, any shared library system seeks to generate position independent code which requires a minimal number of relocations to be applied at runtime, while still running at close to the runtime efficiency of position dependent code.

An additional complexity is that ELF shared libraries were designed to be roughly equivalent to ordinary archives. This means that by default the main executable may override symbols in the shared library, such that references in the shared library will call the definition in the executable, even if the shared library also defines that same symbol. For example, an executable may define its own version of cmalloc. The C library also defines cmalloc, and the C library contains code which calls cmalloc. If the executable defines cmalloc itself, it will override the function in the C library. When some other function in the C library calls malloc, it will call the definition in the executable, not the definition in the C library.

There are thus different requirements pulling in different directions for any

specific ELF implementation. The right implementation choices will depend on the characteristics of the processor. That said, most, but not all, processors make fairly similar decisions. I will describe the common case here. An example of a processor which uses the common case is the i386; an example of a processor which make some different decisions is the PowerPC.

In the common case, code may be compiled in two different modes. By default, code is position dependent. Putting position dependent code into a shared library will cause the program linker to generate a lot of relocation information, and cause the dynamic linker to do a lot of processing at runtime. Code may also be compiled in position independent mode, typically with the `-fpic` option. Position independent code is slightly slower when it calls a non-static function or refers to a global or static variable. However, it requires much less relocation information, and thus the dynamic linker will start the program faster.

Position independent code will call non-static functions via the *Procedure Linkage Table* or *PLT*. This PLT does not exist in .o files. In a .o file, use of the PLT is indicated by a special relocation. When the program linker processes such a relocation, it will create an entry in the PLT. It will adjust the instruction such that it becomes a PC-relative call to the PLT entry. PC-relative calls are inherently position independent and thus do not require a relocation entry themselves. The program linker will create a relocation for the PLT entry which tells the dynamic linker which symbol is associated with that entry. This process reduces the number of dynamic relocations in the shared library from one per function call to one per function called.

Further, PLT entries are normally relocated lazily by the dynamic linker. On most ELF systems this laziness may be overridden by setting the LD_BIND_NOW environment variable when running the program. However, by default, the dynamic linker will not actually apply a relocation to the PLT until some code actually calls the function in question. This also speeds up startup time, in that many invocations of a program will not call every possible function. This is particularly true when considering the shared C library, which has many more function calls than any typical program will execute.

In order to make this work, the program linker initializes the PLT entries to load an index into some register or push it on the stack, and then to branch to common code. The common code calls back into the dynamic linker, which uses the index to find the appropriate PLT relocation, and uses that to find the function being called. The dynamic linker then initializes the PLT entry with the address

of the function, and then jumps to the code of the function. The next time the function is called, the PLT entry will branch directly to the function.

Before giving an example, I will talk about the other major data structure in position independent code, the *Global Offset Table* or *GOT*. This is used for global and static variables. For every reference to a global variable from position independent code, the compiler will generate a load from the GOT to get the address of the variable, followed by a second load to get the actual value of the variable. The address of the GOT will normally be held in a register, permitting efficient access. Like the PLT, the GOT does not exist in a .o file, but is created by the program linker. The program linker will create the dynamic relocations which the dynamic linker will use to initialize the GOT at runtime. Unlike the PLT, the dynamic linker always fully initializes the GOT when the program starts.

For example, on the i386, the address of the GOT is held in the register `%ebx`. This register is initialized at the entry to each function in position independent code. The initialization sequence varies from one compiler to another, but typically looks something like this:

```
call __i686.get_pc_thunk.bx
add $offset,%ebx
```

The function `__i686.get_pc_thunk.bx` simply looks like this:

```
mov (%esp),%ebx
ret
```

This sequence of instructions uses a position independent sequence to get the address at which it is running. Then is uses an offset to get the address of the GOT. Note that this requires that the GOT always be a fixed offset from the code, regardless of where the shared library is loaded. That is, the dynamic linker must load the shared library as a fixed unit; it may not load different parts at varying addresses.

Global and static variables are now read or written by first loading the address via a fixed offset from `%ebx`. The program linker will create dynamic relocations for each entry in the GOT, telling the dynamic linker how to initialize the entry. These relocations are of type `GLOB_DAT`.

For function calls, the program linker will set up a PLT entry to look like this:

```
jmp *offset(%ebx)
pushl index
```

```
jmp first_plt_entry
```

The program linker will allocate an entry in the GOT for each entry in the PLT. It will create a dynamic relocation for the GOT entry of type `JMP_SLOT`. It will initialize the GOT entry to the base address of the shared library plus the address of the second instruction in the code sequence above. When the dynamic linker does the initial lazy binding on a `JMP_SLOT` reloc, it will simply add the difference between the shared library load address and the shared library base address to the GOT entry. The effect is that the first `asm`jmp instruction will jump to the second instruction, which will push the index entry and branch to the first PLT entry. The first PLT entry is special, and looks like this:

```
pushl 4(%ebx)
jmp *8(%ebx)
```

This references the second and third entries in the GOT. The dynamic linker will initialize them to have appropriate values for a callback into the dynamic linker itself. The dynamic linker will use the index pushed by the first code sequence to find the `JMP_SLOT` relocation. When the dynamic linker determines the function to be called, it will store the address of the function into the GOT entry references by the first code sequence. Thus, the next time the function is called, the jmp instruction will branch directly to the right code.

That was a fast pass over a lot of details, but I hope that it conveys the main idea. It means that for position independent code on the i386, every call to a global function requires one extra instruction after the first time it is called. Every reference to a global or static variable requires one extra instruction. Almost every function uses four extra instructions when it starts to initialize `%ebx` (leaf functions which do not refer to any global variables do not need to initialize `%ebx`). This all has some negative impact on the program cache. This is the runtime performance penalty paid to let the dynamic linker start the program quickly.

On other processors, the details are naturally different. However, the general flavour is similar: position independent code in a shared library starts faster and runs slightly slower.

5 Part 5

5.1 Shared Libraries Redux

Yesterday I talked about how shared libraries work. I realized that I should say something about how linkers implement shared libraries. This discussion will again be ELF specific.

When the program linker puts position dependent code into a shared library, it has to copy more of the relocations from the object file into the shared library. They will become dynamic relocations computed by the dynamic linker at runtime. Some relocations do not have to be copied; for example, a PC relative relocation to a symbol which is local to shared library can be fully resolved by the program linker, and does not require a dynamic reloc. However, note that a PC relative relocation to a global symbol does require a dynamic relocation; otherwise, the main executable would not be able to override the symbol. Some relocations have to exist in the shared library, but do not need to be actual copies of the relocations in the object file; for example, a relocation which computes the absolute address of symbol which is local to the shared library can often be replaced with a `RELATIVE` reloc, which simply directs the dynamic linker to add the difference between the shared library's load address and its base address. The advantage of using a `RELATIVE` reloc is that the dynamic linker can compute it quickly at runtime, because it does not require determining the value of a symbol.

For position independent code, the program linker has a harder job. The compiler and assembler will cooperate to generate spcial relocs for position independent code. Although details differ among processors, there will typically be a `PLT` reloc and a `GOT` reloc. These relocs will direct the program linker to add an entry to the PLT or the GOT, as well as performing some computation. For example, on the i386 a function call in position independent code will generate a `R_386_PLT32` reloc. This reloc will refer to a symbol as usual. It will direct the program linker to add a PLT entry for that symbol, if one does not already exist. The computation of the reloc is then a PC-relative reference to the PLT entry. (The `32` in the name of the reloc refers to the size of the reference, which is 32 bits). Yesterday I described how on the i386 every PLT entry also has a corresponding GOT entry, so the `R_386_PLT32` reloc actually directs the program linker to create both a PLT entry and a GOT entry.

When the program linker creates an entry in the PLT or the GOT, it must

also generate a dynamic reloc to tell the dynamic linker about the entry. This will typically be a JMP_SLOT or GLOB_DAT relocation.

This all means that the program linker must keep track of the PLT entry and the GOT entry for each symbol. Initially, of course, there will be no such entries. When the linker sees a PLT or GOT reloc, it must check whether the symbol referenced by the reloc already has a PLT or GOT entry, and create one if it does not. Note that it is possible for a single symbol to have both a PLT entry and a GOT entry; this will happen for position independent code which both calls a function and also takes its address.

The dynamic linker's job for the PLT and GOT tables is to simply compute the JMP_SLOT and GLOB_DAT relocs at runtime. The main complexity here is the lazy evaluation of PLT entries which I described yesterday.

The fact that C permits taking the address of a function introduces an interesting wrinkle. In C you are permitted to take the address of a function, and you are permitted to compare that address to another function address. The problem is that if you take the address of a function in a shared library, the natural result would be to get the address of the PLT entry. After all, that is address to which a call to the function will jump. However, each shared library has its own PLT, and thus the address of a particular function would differ in each shared library. That means that comparisons of function pointers generated in different shraed libraries may be different when they should be the same. This is not a purely hypothetical problem; when I did a port which got it wrong, before I fixed the bug I saw failures in the Tcl shared library when it compared function pointers.

The fix for this bug on most processors is a special marking for a symbol which has a PLT entry but is not defined. Typically the symbol will be marked as undefined, but with a non-zero value—the value will be set to the address of the PLT entry. When the dynamic linker is searching for the value of a symbol to use for a reloc other than a JMP_SLOT reloc, if it finds such a specially marked symbol, it will use the non-zero value. This will ensure that all references to the symbol which are not function calls will use the same value. To make this work, the compiler and assembler must make sure that any reference to a function which does not involve calling it will not carry a standard PLT reloc. This special handling of function addresses needs to be implemented in both the program linker and the dynamic linker.

5.2 ELF Symbols

OK, enough about shared libraries. Let's go over ELF symbols in more detail. I'm not going to lay out the exact data structures—go to the ELF ABI for that. I'm going to take about the different fields and what they mean. Many of the different types of ELF symbols are also used by other object file formats, but I won't cover that.

An entry in an ELF symbol table has eight pieces of information: a name, a value, a size, a section, a binding, a type, a visibility, and undefined additional information (currently there are six undefined bits, though more may be added). An ELF symbol defined in a shared object may also have an associated version name.

The name is obvious.

For an ordinary defined symbol, the section is some section in the file (specifically, the symbol table entry holds an index into the section table). For an object file the value is relative to the start of the section. For an executable the value is an absolute address. For a shared library the value is relative to the base address.

For an undefined reference symbol, the section index is the special value `SHN_UNDEF` which has the value `0`. A section index of `SHN_ABS` (`0xfff1`) indicates that the value of the symbol is an absolute value, not relative to any section.

A section index of `SHN_COMMON` (`0xfff2`) indicates a common symbol. Common symbols were invented to handle Fortran common blocks, and they are also often used for uninitialized global variables in C. A common symbol has unusual semantics. Common symbols have a value of zero, but set the size field to the desired size. If one object file has a common symbol and another has a definition, the common symbol is treated as an undefined reference. If there is no definition for a common symbol, the program linker acts as though it saw a definition initialized to zero of the appropriate size. Two object files may have common symbols of different sizes, in which case the program linker will use the largest size. Implementing common symbol semantics across shared libraries is a touchy subject, somewhat helped by the recent introduction of a type for common symbols as well as a special section index (see the discussion of symbol types below).

The size of an ELF symbol, other than a common symbol, is the size of the variable or function. This is mainly used for debugging purposes.

The binding of an elf symbol is global, local, or weak. A global symbol is globally visible. A local symbol is only locally visible (e.g., a static function). Weak symbols come in two flavors. A weak undefined reference is like an ordinary undefined reference, except that it is not an error if a relocation refers to a weak undefined reference symbol which has no defining symbol. Instead, the relocation is computed as though the symbol had the value zero.

A weak defined symbol is permitted to be linked with a non-weak defined symbol of the same name without causing a multiple definition error. Historically there are two ways for the program linker to handle a weak defined symbol. On SVR4 if the program linker sees a weak defined symbol followed by a non-weak defined symbol with the same name, it will issue a multiple definition error. However, a non-weak defined symbol followed by a weak defined symbol will not cause an error. On Solaris, a weak defined symbol followed by a non-weak defined symbol is handled by causing all references to attach to the non-weak defined symbol, with no error. This difference in behaviour is due to an ambiguity in the ELF ABI which was read differently by different people. The GNU linker follows the Solaris behaviour.

The type of an ELF symbol is one of the following:

- `STT_NOTYPE`: No particular type.
- `STT_OBJECT`: A data object, such as a variable.
- `STT_FUNC`: A function.
- `STT_SECTION`: a local symbol associated with a section. This type of symbol is used to reduce the number of local symbols required, by changing all relocations against local symbols in a specific section to use the `STT_SECTION` symbol instead.
- `STT_FILE`: A special symbol whose name is the name of the source file which produced the object file.
- `STT_COMMON`: A common symbol. This is the same as setting the section index to `SHN_COMMON`, except in a shared object. The program linker will normally have allocated space for the common symbol in the shared object, so it will have a real section index. The `STT_COMMON` type tells the dynamic linker that although the symbol has a regular definition, it is a common symbol.

- `STT_TLS`: A symbol in the Thread Local Storage area. I will describe this in more detail some other day.

ELF symbol visibility was invented to provide more control over which symbols were accessible outside a shared library. The basic idea is that a symbol may be global within a shared library, but local outside the shared library.

- `STV_DEFAULT`: The usual visibility rules apply: global symbols are visible everywhere.
- `STV_INTERNAL`: The symbol is not accessible outside the current executable or shared library.
- `STV_HIDDEN`: The symbol is not visible outside the current executable or shared library, but it may be accessed indirectly, probably because some code took its address.
- `STV_PROTECTED`: The symbol is visible outside the current executable or shared object, but it may not be overridden. That is, if a protected symbol in a shared library is referenced by other code in the shared library, that other code will always reference the symbol in the shared library, even if the executable defines a symbol with the same name.

I'll described symbol versions later.

6 Part 6

So many things to talk about. Let's go back and cover relocations in some more detail, with some examples.

6.1 Relocations

As I said back in part 2, a relocation is a computation to perform on the contents. And as I said yesterday, a relocation can also direct the linker to take other actions, like creating a PLT or GOT entry. Let's take a closer look at the computation.

In general a relocation has a type, a symbol, an offset into the contents, and an addend.

From the linker's point of view, the contents are simply an uninterpreted series of bytes. A relocation changes those bytes as necessary to produce the correct final executable. For example, consider the C code `g = 0`; where g is a global variable. On the i386, the compiler will turn this into an assembly language instruction, which will most likely be `movl $0, g` (for position dependent code–position independent code would loading the address of g from the GOT). Now, the g in the C code is a global variable, and we all more or less know what that means. The g in the assembly code is not that variable. It is a symbol which holds the address of that variable.

The assembler does not know the address of the global variable g, which is another way of saying that the assembler does not know the value of the symbol g. It is the linker that is going to pick that address. So the assembler has to tell the linker that it needs to use the address of g in this instruction. The way the assembler does this is to create a relocation. We don't use a separate relocation type for each instruction; instead, each processor will have a natural set of relocation types which are appropriate for the machine architecture. Each type of relocation expresses a specific computation.

In the i386 case, the assembler will generate these bytes:

```
c7 05 00 00 00 00 00 00 00 00
```

The `c7 05` are the instruction (movl constant to address). The first four `00` bytes are the 32-bit constant `0`. The second four `00` bytes are the address. The assembler tells the linker to put the value of the symbol g into those four bytes by generating (in this case) a `R_386_32` relocation. For this relocation the symbol will be g, the offset will be to the last four bytes of the instruction, the type will be `R_386_32`, and the addend will be `0` (in the case of the i386 the addend is stored in the contents rather than in the relocation itself, but this is a detail). The type `R_386_32` expresses a specific computation, which is: put the 32-bit sum of the value of the symbol and the addend into the offset. Since for the i386 the addend is stored in the contents, this can also be expressed as: add the value of the symbol to the 32-bit field at the offset. When the linker performs this computation, the address in the instruction will be the address of the global variable g. Regardless of the details, the important point to note is that the relocation adjusts the contents by applying a specific computation selected by the type.

An example of a simple case which does use an addend would be

```
char a[10]; // A global array.
char* p = &a[1]; // In a function.
```

The assignment to p will wind up requiring a relocation for the symbol a. Here the addend will be 1, so that the resulting instruction references a + 1 rather than a + 0.

To point out how relocations are processor dependent, let's consider g = 0; on a RISC processor: the PowerPC (in 32-bit mode). In this case, multiple assembly language instructions are required:

```
li 1,0 // Set register 1 to 0
lis 9,g@ha // Load high-adjusted part of g into register 9
stw 1,g@l(9) // Store register 1 to address in register 9 plus low adjusted part g
```

The lis instruction loads a value into the upper 16 bits of register 9, setting the lower 16 bits to zero. The stw instruction adds a signed 16 bit value to register 9 to form an address, and then stores the value of register 1 at that address. The @ha part of the operand directs the assembler to generate a R_PPC_ADDR16_HA reloc. The @l produces a R_PPC_ADDR16_LO reloc. The goal of these relocs is to compute the value of the symbol g and use it as the store address.

That is enough information to determine the computations performed by these relocs. The R_PPC_ADDR16_HA reloc computes (SYMBOL >> 16) + ((SYMBOL & 0x8000) ? 1 : 0). The R_PPC_ADDR16_LO computes SYMBOL & 0xffff. The extra computation for R_PPC_ADDR16_HA is because the stw instruction adds the signed 16-bit value, which means that if the low 16 bits appears negative we have to adjust the high 16 bits accordingly. The offsets of the relocations are such that the 16-bit resulting values are stored into the appropriate parts of the machine instructions.

The specific examples of relocations I've discussed here are ELF specific, but the same sorts of relocations occur for any object file format.

The examples I've shown are for relocations which appear in an object file. As discussed in part 4, these types of relocations may also appear in a shared library, if they are copied there by the program linker. In ELF, there are also specific relocation types which never appear in object files but only appear in shared libraries or executables. These are the JMP_SLOT, GLOB_DAT, and RELATIVE relocations discussed earlier. Another type of relocation which only appears in an executable is

a COPY relocation, which I will discuss later.

6.2 Position Dependent Shared Libraries

I realized that in part 4 I forgot to say one of the important reasons that ELF shared libraries use PLT and GOT tables. The idea of a shared library is to permit mapping the same shared library into different processes. This only works at maximum efficiency if the shared library code looks the same in each process. If it does not look the same, then each process will need its own private copy, and the savings in physical memory and sharing will be lost.

As discussed in part 4, when the dynamic linker loads a shared library which contains position dependent code, it must apply a set of dynamic relocations. Those relocations will change the code in the shared library, and it will no longer be sharable.

The advantage of the PLT and GOT is that they move the relocations elsewhere, to the PLT and GOT tables themselves. Those tables can then be put into a read-write part of the shared library. This part of the shared library will be much smaller than the code. The PLT and GOT tables will be different in each process using the shared library, but the code will be the same.

7 Part 7

As we've seen, what linkers do is basically quite simple, but the details can get complicated. The complexity is because smart programmers can see small optimizations to speed up their programs a little bit, and somtimes the only place those optimizations can be implemented is the linker. Each such optimizations makes the linker a little more complicated. At the same time, of course, the linker has to run as fast as possible, since nobody wants to sit around waiting for it to finish. Today I'll talk about a classic small optimization implemented by the linker.

7.1 Thread Local Storage

I'll assume you know what a thread is. It is often useful to have a global variable which can take on a different value in each thread (if you don't see why this is

useful, just trust me on this). That is, the variable is global to the program, but the specific value is local to the thread. If thread A sets the thread local variable to 1, and thread B then sets it to 2, then code running in thread A will continue to see the value 1 for the variable while code running in thread B sees the value 2. In Posix threads this type of variable can be created via `pthread_key_create` and accessed via `pthread_getspecific` and `pthread_setspecific`.

Those functions work well enough, but making a function call for each access is awkward and inconvenient. It would be more useful if you could just declare a regular global variable and mark it as thread local. That is the idea of Thread Local Storage (TLS), which I believe was invented at Sun. On a system which supports TLS, any global (or static) variable may be annotated with `__thread`. The variable is then thread local.

Clearly this requires support from the compiler. It also requires support from the program linker and the dynamic linker. For maximum efficiency–and why do this if you aren't going to get maximum efficiency?–some kernel support is also needed. The design of TLS on ELF systems fully supports shared libraries, including having multiple shared libraries, and the executable itself, use the same name to refer to a single TLS variable. TLS variables can be initialized. Programs can take the address of a TLS variable, and pass the pointers between threads, so the address of a TLS variable is a dynamic value and must be globally unique.

How is this all implemented? First step: define different storage models for TLS variables.

- *Global Dynamic*: Fully general access to TLS variables from an executable or a shared object.
- *Local Dynamic*: Permits access to a variable which is bound locally within the executable or shared object from which it is referenced. This is true for all static TLS variables, for example. It is also true for protected symbols–I described those back in part 5.
- *Initial Executable*: Permits access to a variable which is known to be part of the TLS image of the executable. This is true for all TLS variables defined in the executable itself, and for all TLS variables in shared libraries explicitly linked with the executable. This is not true for accesses from a shared library, nor for accesses to TLS variables defined in shared libraries opened by `dlopen`.

- *Local Executable*: Permits access to TLS variables defined in the executable itself.

These storage models are defined in decreasing order of flexibility. Now, for efficiency and simplicity, a compiler which supports TLS will permit the developer to specify the appropriate TLS model to use (with gcc, this is done with the `-ftls-model` option, although the Global Dynamic and Local Dynamic models also require using `-fpic`). So, when compiling code which will be in an executable and never be in a shared library, the developer may choose to set the TLS storage model to Initial Executable.

Of course, in practice, developers often do not know where code will be used. And developers may not be aware of the intricacies of TLS models. The program linker, on the other hand, knows whether it is creating an executable or a shared library, and it knows whether the TLS variable is defined locally. So the program linker gets the job of automatically optimizing references to TLS variables when possible. These references take the form of relocations, and the linker optimizes the references by changing the code in various ways.

The program linker is also responsible for gathering all TLS variables together into a single TLS segment (I'll talk more about segments later, for now think of them as a section). The dynamic linker has to group together the TLS segments of the executable and all included shared libraries, resolve the dynamic TLS relocations, and has to build TLS segments dynamically when dlopen is used. The kernel has to make it possible for access to the TLS segments be efficient.

That was all pretty general. Let's do an example, again for i386 ELF. There are three different implementations of i386 ELF TLS; I'm going to look at the gnu implementation. Consider this trivial code:

```
__thread int i;
int foo() { return i; }
```

In global dynamic mode, this generates i386 assembler code like this:

```
leal i@TLSGD(,%ebx,1), %eax
call ___tls_get_addr@PLT
movl (%eax), %eax
```

Recall from part 4 that `%ebx` holds the address of the GOT table. The first instruction will have a `R_386_TLS_GD` relocation for the variable `i`; the relocation

will apply to the offset of the `leal` instruction. When the program linker sees this relocation, it will create two consecutive entries in the GOT table for the TLS variable `i`. The first one will get a `R_386_TLS_DTPMOD32` dynamic relocation, and the second will get a `R_386_TLS_DTPOFF32` dynamic relocation. The dynamic linker will set the `DTPMOD32` GOT entry to hold the *module ID* of the object which defines the variable. The module ID is an index within the dynamic linker's tables which identifies the executable or a specific shared library. The dynamic linker will set the `DTPOFF32` GOT entry to the offset within the TLS segment for that module. The `__tls_get_addr` function will use those values to compute the address (this function also takes care of lazy allocation of TLS variables, which is a further optimization specific to the dynamic linker). Note that `__tls_get_addr` is actually implemented by the dynamic linker itself; it follows that global dynamic TLS variables are not supported (and not necessary) in statically linked executables.

At this point you are probably wondering what is so inefficient about `pthread_getspecific`. The real advantage of TLS shows when you see what the program linker can do. The leal; call sequence shown above is canonical: the compiler will always generate the same sequence to access a TLS variable in global dynamic mode. The program linker takes advantage of that fact. If the program linker sees that the code shown above is going into an executable, it knows that the access does not have to be treated as global dynamic; it can be treated as initial executable. The program linker will actually rewrite the code to look like this:

```
movl %gs:0, %eax
subl $i@GOTTPOFF(%ebx), %eax
```

Here we see that the TLS system has coopted the `%gs` segment register, with cooperation from the operating system, to point to the TLS segment of the executable. For each processor which supports TLS, some such efficiency hack is made. Since the program linker is building the executable, it builds the TLS segment, and knows the offset of i in the segment. The `GOTTPOFF` is not a real relocation; it is created and then resolved within the program linker. It is, of course, the offset from the GOT table to the address of `i` in the TLS segment. The `movl (%eax), %eax` from the original sequence remains to actually load the value of the variable.

Actually, that is what would happen if `i` were not defined in the executable itself. In the example I showed, `i` is defined in the executable, so the program linker can actually go from a global dynamic access all the way to a local executable access. That looks like this:

```
movl %gs:0,%eax
```

```
subl $i@TPOFF,%eax
```

Here `i@TPOFF` is simply the known offset of `i` within the TLS segment. I'm not going to go into why this uses `subl` rather than `addl`; suffice it to say that this is another efficiency hack in the dynamic linker.

If you followed all that, you'll see that when an executable accesses a TLS variable which is defined in that executable, it requires two instructions to compute the address, typically followed by another one to actually load or store the value. That is significantly more efficient than calling `pthread_getspecific`. Admittedly, when a shared library accesses a TLS variable, the result is not much better than `pthread_getspecific`, but it shouldn't be any worse, either. And the code using `__thread` is much easier to write and to read.

That was a real whirlwind tour. There are three separate but related TLS implementations on i386 (known as sun, gnu, and gnu2), and 23 different relocation types are defined. I'm certainly not going to try to describe all the details; I don't know them all in any case. They all exist in the name of efficient access to the TLS variables for a given storage model.

Is TLS worth the additional complexity in the program linker and the dynamic linker? Since those tools are used for every program, and since the C standard global variable `errno` in particular can be implemented using TLS, the answer is most likely yes.

8 Part 8

8.1 ELF Segments

Earlier I said that executable file formats were normally the same as object file formats. That is true for ELF, but with a twist. In ELF, object files are composed of sections: all the data in the file is accessed via the section table. Executables and shared libraries normally contain a section table, which is used by programs like `nm`. But the operating system and the dynamic linker do not use the section table. Instead, they use the segment table, which provides an alternative view of the file.

All the contents of an ELF executable or shared library which are to be loaded into memory are contained within a segment (an object file does not have seg-

ments). A segment has a type, some flags, a file offset, a virtual address, a physical address, a file size, a memory size, and an alignment. The file offset points to a contiguous set of bytes which are the contents of the segment, the bytes to load into memory. When the operating system or the dynamic linker loads a file, it will do so by walking through the segments and loading them into memory (typically by using the `mmap` system call). All the information needed by the dynamic linker–the dynamic relocations, the dynamic symbol table, etc.–are accessed via information stored in special segments.

Although an ELF executable or shared library does not, strictly speaking, require any sections, they normally do have them. The contents of a loadable section will fall entirely within a single segment.

The program linker reads sections from the input object files. It sorts and concatenates them into sections in the output file. It maps all the loadable sections into segments in the output file. It lays out the section contents in the output file segments respecting alignment and access requirements, so that the segments may be mapped directly into memory. The sections are mapped to segments based on the access requirements: normally all the read-only sections are mapped to one segment and all the writable sections are mapped to another segment. The address of the latter segment will be set so that it starts on a separate page in memory, permitting `mmap` to set different permissions on the mapped pages.

The segment flags are a bitmask which define access requirements. The defined flags are `PF_R`, `PF_W`, and `PF_X`, which mean, respectively, that the contents must be made readable, writable, or executable.

The segment virtual address is the memory address at which the segment contents are loaded at runtime. The physical address is officially undefined, but is often used as the load address when using a system which does not use virtual memory. The file size is the size of the contents in the file. The memory size may be larger than the file size when the segment contains uninitialized data; the extra bytes will be filled with zeroes. The alignment of the segment is mainly informative, as the address is already specified.

The ELF segment types are as follows:

- `PT_NULL`: A null entry in the segment table, which is ignored.
- `PT_LOAD`: A loadable entry in the segment table. The operating system or dynamic linker load all segments of this type. All other segments with contents

will have their contents contained completely within a `PT_LOAD` segment.

- `PT_DYNAMIC`: The dynamic segment. This points to a series of dynamic tags which the dynamic linker uses to find the dynamic symbol table, dynamic relocations, and other information that it needs.
- `PT_INTERP`: The interpreter segment. This appears in an executable. The operating system uses it to find the name of the dynamic linker to run for the executable. Normally all executables will have the same interpreter name, but on some operating systems different interpreters are used in different emulation modes.
- `PT_NOTE`: A note segment. This contains system dependent note information which may be used by the operating system or the dynamic linker. On GNU/Linux systems shared libraries often have a ABI tag note which may be used to specify the minimum version of the kernel which is required for the shared library. The dynamic linker uses this when selecting among different shared libraries.
- `PT_SHLIB`: This is not used as far as I know.
- `PT_PHDR`: This indicates the address and size of the segment table. This is not too useful in practice as you have to have already found the segment table before you can find this segment.
- `PT_TLS`: The TLS segment. This holds the initial values for TLS variables.
- `PT_GNU_EH_FRAME` (`0x6474e550`): A GNU extension used to hold a sorted table of unwind information. This table is built by the GNU program linker. It is used by gcc's support library to quickly find the appropriate handler for an exception, without requiring exception frames to be registered when the program start.
- `PT_GNU_STACK` (`0x6474e551`): A GNU extension used to indicate whether the stack should be executable. This segment has no contents. The dynamic linker sets the permission of the stack in memory to the permissions of this segment.
- `PT_GNU_RELRO` (`0x6474e552`): A GNU extension which tells the dynamic linker to set the given address and size to be read-only after applying dynamic relocations. This is used for const variables which require dynamic relocations.

8.2 ELF Sections

Now that we've done segments, lets take a quick look at the details of ELF sections. ELF sections are more complicated than segments, in that there are more types of sections. Every ELF object file, and most ELF executables and shared libraries, have a table of sections. The first entry in the table, section 0, is always a null section.

ELF sections have several fields.

- Name.
- Type. I discuss section types below.
- Flags. I discuss section flags below.
- Address. This is the address of the section. In an object file this is normally zero. In an executable or shared library it is the virtual address. Since executables are normally accessed via segments, this is essentially documentation.
- File offset. This is the offset of the contents within the file.
- Size. The size of the section.
- Link. Depending on the section type, this may hold the index of another section in the section table.
- Info. The meaning of this field depends on the section type.
- Address alignment. This is the required alignment of the section. The program linker uses this when laying out the section in memory.
- Entry size. For sections which hold an array of data, this is the size of one data element.

These are the types of ELF sections which the program linker may see.

- `SHT_NULL`: A null section. Sections with this type may be ignored.
- `SHT_PROGBITS`: A section holding bits of the program. This is an ordinary section with contents.

- SHT_SYMTAB: The symbol table. This section actually holds the symbol table itself. The section contents are an array of ELF symbol structures.
- SHT_STRTAB: A string table. This type of section holds null-terminated strings. Sections of this type are used for the names of the symbols and the names of the sections themselves.
- SHT_RELA: A relocation table. The link field holds the index of the section to which these relocations apply. These relocations include addends.
- SHT_HASH: A hash table used by the dynamic linker to speed symbol lookup.
- SHT_DYNAMIC: The dynamic tags used by the dynamic linker. Normally the PT_DYNAMIC segment and the SHT_DYNAMIC section will point to the same contents.
- SHT_NOTE: A note section. This is used in system dependent ways. A loadable SHT_NOTE section will become a PT_NOTE segment.
- SHT_NOBITS: A section which takes up memory space but has no associated contents. This is used for zero-initialized data.
- SHT_REL: A relocation table, like SHT_RELA but the relocations have no addends.
- SHT_SHLIB: This is not used as far as I know.
- SHT_DYNSYM: The dynamic symbol table. Normally the DT_SYMTAB dynamic tag will point to the same contents as this section (I haven't discussed dynamic tags yet, though).
- SHT_INIT_ARRAY: This section holds a table of function addresses which should each be called at program startup time, or, for a shared library, when the library is opened by dlopen.
- SHT_FINI_ARRAY: Like SHT_INIT_ARRAY, but called at program exit time or dlclose time.
- SHT_PREINIT_ARRAY: Like SHT_INIT_ARRAY, but called before any shared libraries are initialized. Normally shared libraries initializers are run before the executable initializers. This section type may only be linked into an executable, not into a shared library.

- SHT_GROUP: This is used to group related sections together, so that the program linker may discard them as a unit when appropriate. Sections of this type may only appear in object files. The contents of this type of section are a flag word followed by a series of section indices.
- SHT_SYMTAB_SHNDX: ELF symbol table entries only provide a 16-bit field for the section index. For a file with more than 65536 sections, a section of this type is created. It holds one 32-bit word for each symbol. If a symbol's section index is SHN_XINDEX, the real section index may be found by looking in the SHT_SYMTAB_SHNDX section.
- SHT_GNU_LIBLIST (0x6ffffff7): A GNU extension used by the prelinker to hold a list of libraries found by the prelinker.
- SHT_GNU_verdef (0x6ffffffd): A Sun and GNU extension used to hold version definitions (I'll take about symbol versions at some point).
- SHT_GNU_verneed (0x6ffffffe): A Sun and GNU extension used to hold versions required from other shared libraries.
- SHT_GNU_versym (0x6fffffff): A Sun and GNU extension used to hold the versions for each symbol.

These are the types of section flags.

- SHF_WRITE: Section contains writable data.
- SHF_ALLOC: Section contains data which should be part of the loaded program image. For example, this would normally be set for a SHT_PROGBITS section and not set for a SHT_SYMTAB section.
- SHF_EXECINSTR: Section contains executable instructions.
- SHF_MERGE: Section contains constants which the program linker may merge together to save space. The compiler can use this type of section for read-only data whose address is unimportant.
- SHF_STRINGS: In conjunction with SHF_MERGE, this means that the section holds null terminated string constants which may be merged.
- SHF_INFO_LINK: This flag indicates that the info field in the section holds a section index.

- `SHF_LINK_ORDER`: This flag tells the program linker that when it combines sections, this section must appear in the same relative order as the section in the link field. This can be used to ensure that address tables are built in the expected order.
- `SHF_OS_NONCONFORMING`: If the program linker sees a section with this flag, and does not understand the type or all other flags, then it must issue an error.
- `SHF_GROUP`: This section appears in a group (see `SHT_GROUP`, above).
- `SHF_TLS`: This section holds TLS data.

9 Part 9

9.1 Symbol Versions

A shared library provides an API. Since executables are built with a specific set of header files and linked against a specific instance of the shared library, it also provides an ABI. It is desirable to be able to update the shared library independently of the executable. This permits fixing bugs in the shared library, and it also permits the shared library and the executable to be distributed separately. Sometimes an update to the shared library requires changing the API, and sometimes changing the API requires changing the ABI. When the ABI of a shared library changes, it is no longer possible to update the shared library without updating the executable. This is unfortunate.

For example, consider the system C library and the `stat` function. When file systems were upgraded to support 64-bit file offsets, it became necessary to change the type of some of the fields in the `stat` struct. This is a change in the ABI of `stat`. New versions of the system library should provide a `stat` which returns 64-bit values. But old existing executables call `stat` expecting 32-bit values. This could be addressed by using complicated macros in the system header files. But there is a better way.

The better way is symbol versions, which were introduced at Sun and extended by the GNU tools. Every shared library may define a set of symbol versions, and assign specific versions to each defined symbol. The versions and symbol assignments are done by a script passed to the program linker when creating the

shared library.

When an executable or shared library A is linked against another shared library B, and A refers to a symbol S defined in B with a specific version, the undefined dynamic symbol reference S in A is given the version of the symbol S in B. When the dynamic linker sees that A refers to a specific version of S, it will link it to that specific version in B. If B later introduces a new version of S, this will not affect A, as long as B continues to provide the old version of S.

For example, when `stat` changes, the C library would provide two versions of `stat`, one with the old version (e.g., LIBC_1.0), and one with the new version (LIBC_2.0). The new version of `stat` would be marked as the default–the program linker would use it to satisfy references to `stat` in object files. Executables linked against the old version would require the LIBC_1.0 version of `stat`, and would therefore continue to work. Note that it is even possible for both versions of `stat` to be used in a single program, accessed from different shared libraries.

As you can see, the version effectively is part of the name of the symbol. The biggest difference is that a shared library can define a specific version which is used to satisfy an unversioned reference.

Versions can also be used in an object file (this is a GNU extension to the original Sun implementation). This is useful for specifying versions without requiring a version script. When a symbol name containts the `@` character, the string before the `@` is the name of the symbol, and the string after the `@` is the version. If there are two consecutive `@` characters, then this is the default version.

9.2 Relaxation

Generally the program linker does not change the contents other than applying relocations. However, there are some optimizations which the program linker can perform at link time. One of them is relaxation.

Relaxation is inherently processor specific. It consists of optimizing code sequences which can become smaller or more efficient when final addresses are known. The most common type of relaxation is for call instructions. A processor like the m68k supports different PC relative call instructions: one with a 16-bit offset, and one with a 32-bit offset. When calling a function which is within range of the 16-bit offset, it is more efficient to use the shorter instruction. The optimiza-

tion of shrinking these instructions at link time is known as relaxation.

Relaxation is applied based on relocation entries. The linker looks for relocations which may be relaxed, and checks whether they are in range. If they are, the linker applies the relaxation, probably shrinking the size of the contents. The relaxation can normally only be done when the linker recognizes the instruction being relocated. Applying a relaxation may in turn bring other relocations within range, so relaxation is typically done in a loop until there are no more opportunities.

When the linker relaxes a relocation in the middle of a contents, it may need to adjust any PC relative references which cross the point of the relaxation. Therefore, the assembler needs to generate relocation entries for all PC relative references. When not relaxing, these relocations may not be required, as a PC relative reference within a single contents will be valid whereever the contents winds up. When relaxing, though, the linker needs to look through all the other relocations that apply to the contents, and adjust PC relatives one where appropriate. This adjustment will simply consist of recomputing the PC relative offset.

Of course it is also possible to apply relaxations which do not change the size of the contents. For example, on the MIPS the position independent calling sequence is normally to load the address of the function into the `$25` register and then to do an indirect call through the register. When the target of the call is within the 18-bit range of the branch-and-call instruction, it is normally more efficient to use branch-and-call, since then the processor does not have to wait for the load of `$25` to complete before starting the call. This relaxation changes the instruction sequence without changing the size.

More tomorrow. I apologize for the haphazard arrangement of these linker notes. I'm just writing about ideas as I think of them, rather than being organized about that. If I do collect these notes into an essay, I'll try to make them more structured.

10 Part 10

10.1 Parallel Linking

It is possible to parallelize the linking process somewhat. This can help hide I/O latency and can take better advantage of modern multi-core systems. My intention with gold is to use these ideas to speed up the linking process.

The first area which can be parallelized is reading the symbols and relocation entries of all the input files. The symbols must be processed in order; otherwise, it will be difficult for the linker to resolve multiple definitions correctly. In particular all the symbols which are used before an archive must be fully processed before the archive is processed, or the linker won't know which members of the archive to include in the link (I guess I haven't talked about archives yet). However, despite these ordering requirements, it can be beneficial to do the actual I/O in parallel.

After all the symbols and relocations have been read, the linker must complete the layout of all the input contents. Most of this can not be done in parallel, as setting the location of one type of contents requires knowing the size of all the preceding types of contents. While doing the layout, the linker can determine the final location in the output file of all the data which needs to be written out.

After layout is complete, the process of reading the contents, applying relocations, and writing the contents to the output file can be fully parallelized. Each input file can be processed separately.

Since the final size of the output file is known after the layout phase, it is possible to use `mmap` for the output file. When not doing relaxation, it is then possible to read the input contents directly into place in the output file, and to relocation them in place. This reduces the number of system calls required, and ideally will permit the operating system to do optimal disk I/O for the output file.

11 Part 11

11.1 Archives

Archives are a traditional Unix package format. They are created by the ar program, and they are normally named with a `.a` extension. Archives are passed to a Unix linker with the `-l` option.

Although the `ar` program is capable of creating an archive from any type of file, it is normally used to put object files into an archive. When it is used in this way, it creates a symbol table for the archive. The symbol table lists all the symbols defined by any object file in the archive, and for each symbol indicates which object file defines it. Originally the symbol table was created by the `ranlib` program, but these days it is always created by ar by default (despite this, many Makefiles continue to run ranlib unnecessarily).

When the linker sees an archive, it looks at the archive's symbol table. For each symbol the linker checks whether it has seen an undefined reference to that symbol without seeing a definition. If that is the case, it pulls the object file out of the archive and includes it in the link. In other words, the linker pulls in all the object files which defines symbols which are referenced but not yet defined.

This operation repeats until no more symbols can be defined by the archive. This permits object files in an archive to refer to symbols defined by other object files in the same archive, without worrying about the order in which they appear.

Note that the linker considers an archive in its position on the command line relative to other object files and archives. If an object file appears after an archive on the command line, that archive will not be used to defined symbols referenced by the object file.

In general the linker will not include archives if they provide a definition for a common symbol. You will recall that if the linker sees a common symbol followed by a defined symbol with the same name, it will treat the common symbol as an undefined reference. That will only happen if there is some other reason to include the defined symbol in the link; the defined symbol will not be pulled in from the archive.

There was an interesting twist for common symbols in archives on old a.out-based SunOS systems. If the linker saw a common symbol, and then saw a com-

mon symbol in an archive, it would not include the object file from the archive, but it would change the size of the common symbol to the size in the archive if that were larger than the current size. The C library relied on this behaviour when implementing the `stdin` variable.

12 Part 12

I apologize for the pause in posts. We moved over the weekend. Last Friday at&t told me that the new DSL was working at our new house. However, it did not actually start working outside the house until Wednesday. Then a problem with the internal wiring meant that it was not working inside the house until today. I am now finally back online at home.

12.1 Symbol Resolution

I find that symbol resolution is one of the trickier aspects of a linker. Symbol resolution is what the linker does the second and subsequent times that it sees a particular symbol. I've already touched on the topic in a few previous entries, but let's look at it in a bit more depth.

Some symbols are local to a specific object files. We can ignore these for the purposes of symbol resolution, as by definition the linker will never see them more than once. In ELF these are the symbols with a binding of `STB_LOCAL`.

In general, symbols are resolved by name: every symbol with the same name is the same entity. We've already seen a few exceptions to that general rule. A symbol can have a version: two symbols with the same name but different versions are different symbols. A symbol can have non-default visibility: a symbol with hidden visibility in one shared library is not the same as a symbol with the same name in a different shared library.

The characteristics of a symbol which matter for resolution are:

- The symbol name
- The symbol version.

- Whether the symbol is the default version or not.
- Whether the symbol is a definition or a reference or a common symbol.
- The symbol visibility.
- Whether the symbol is weak or strong (i.e., non-weak).
- Whether the symbol is defined in a regular object file being included in the output, or in a shared library.
- Whether the symbol is thread local.
- Whether the symbol refers to a function or a variable.

The goal of symbol resolution is to determine the final value of the symbol. After all symbols are resolved, we should know the specific object file or shared library which defines the symbol, and we should know the symbol's type, size, etc. It is possible that some symbols will remain undefined after all the symbol tables have been read; in general this is only an error if some relocation refers to that symbol.

At this point I'd like to present a simple algorithm for symbol resolution, but I don't think I can. I'll try to hit all the high points, though. Let's assume that we have two symbols with the same name. Let's call the symbol we saw first A and the new symbol B. (I'm going to ignore symbol visibility in the algorithm below; the effects of visibility should be obvious, I hope.)

1. If A has a version:
    - If B has a version different from A, they are actually different symbols.
    - If B has the same version as A, they are the same symbol; carry on.
    - If B does not have a version, and A is the default version of the symbol, they are the same symbol; carry on.
    - Otherwise B is probably a different symbol. But note that if A and B are both undefined references, then it is possible that A refers to the default version of the symbol but we don't yet know that. In that case, if B does not have a version, A and B really are the same symbol. We can't tell until we see the actual definition.
2. If A does not have a version:

- If B does not have a version, they are the same symbol; carry on.
- If B has a version, and it is the default version, they are the same symbol; carry on.
- Otherwise, B is probably a different symbol, as above.

3. If A is thread local and B is not, or vice-versa, then we have an error.
4. If A is an undefined reference:

   - If B is an undefined reference, then we can complete the resolution, and more or less ignore B.
   - If B is a definition or a common symbol, then we can resolve A to B.

5. If A is a strong definition in an object file:

   - If B is an undefined reference, then we resolve B to A.
   - If B is a strong definition in an object file, then we have a multiple definition error.
   - If B is a weak definition in an object file, then A overrides B. In effect, B is ignored.
   - If B is a common symbol, then we treat B as an undefined reference.
   - If B is a definition in a shared library, then A overrides B. The dynamic linker will change all references to B in the shared library to refer to A instead.

6. If A is a weak definition in an object file, we act just like the strong definition case, with one exception: if B is a strong definition in an object file. In the original SVR4 linker, this case was treated as a multiple definition error. In the Solaris and GNU linkers, this case is handled by letting B override A.
7. If A is a common symbol in an object file:

   - If B is a common symbol, we set the size of A to be the maximum of the size of A and the size of B, and then treat B as an undefined reference.
   - If B is a definition in a shared library with function type, then A overrides B (this oddball case is required to correctly handle some Unix system libraries).
   - Otherwise, we treat A as an undefined reference.

8. If A is a definition in a shared library, then if B is a definition in a regular object (strong or weak), it overrides A. Otherwise we act as though A were defined in an object file.

9. If A is a common symbol in a shared library, we have a funny case. Symbols in shared libraries must have addresses, so they can't be common in the same sense as symbols in an object file. But ELF does permit symbols in a shared library to have the type `STT_COMMON` (this is a relatively recent addition). For purposes of symbol resolution, if A is a common symbol in a shared library, we still treat it as a definition, unless B is also a common symbol. In the latter case, B overrides A, and the size of B is set to the maximum of the size of A and the size of B.

I hope I got all that right.

13 Part 13

13.1 Symbol Versions Redux

I've talked about symbol versions from the linker's point of view. I think it's worth discussing them a bit from the user's point of view.

As I've discussed before, symbol versions are an ELF extension designed to solve a specific problem: making it possible to upgrade a shared library without changing existing executables. That is, they provide backward compatibility for shared libraries. There are a number of related problems which symbol versions do not solve. They do not provide forward compatibility for shared libraries: if you upgrade your executable, you may need to upgrade your shared library also (it would be nice to have a feature to build your executable against an older version of the shared library, but that is difficult to implement in practice). They only work at the shared library interface: they do not help with a change to the ABI of a system call, which is at the kernel interface. They do not help with the problem of sharing incompatible versions of a shared library, as may happen when a complex application is built out of several different existing shared libraries which have incompatible dependencies.

Despite these limitations, shared library backward compatibility is an important issue. Using symbol versions to ensure backward compatibility requires a

careful and rigorous approach. You must start by applying a version to every symbol. If a symbol in the shared library does not have a version, then it is impossible to change it in a backward compatible fashion. Then you must pay close attention to the ABI of every symbol. If the ABI of a symbol changes for any reason, you must provide a copy which implements the old ABI. That copy should be marked with the original version. The new symbol must be given a new version.

The ABI of a symbol can change in a number of ways. Any change to the parameter types or the return type of a function is an ABI change. Any change in the type of a variable is an ABI change. If a parameter or a return type is a struct or class, then any change in the type of any field is an ABI change–i.e., if a field in a struct points to another struct, and that struct changes, the ABI has changed. If a function is defined to return an instance of an enum, and a new value is added to the enum, that is an ABI change. In other words, even minor changes can be ABI changes. The question you need to ask is: can existing code which has already been compiled continue to use the new symbol with no change? If the answer is no, you have an ABI change, and you must define a new symbol version.

You must be very careful when writing the symbol implementing the old ABI, if you don't just copy the existing code. You must be certain that it really does implement the old ABI.

There are some special challenges when using C++. Adding a new virtual method to a class can be an ABI change for any function which uses that class. Providing the backward compatible version of the class in such a situation is very awkward–there is no natural way to specify the name and version to use for the virtual table or the RTTI information for the old version.

Naturally, you must never delete any symbols.

Getting all the details correct, and verifying that you got them correct, requires great attention to detail. Unfortunately, I don't know of any tools to help people write correct version scripts, or to verify them. Still, if implemented correctly, the results are good: existing executables will continue to run.

13.2 Static Linking vs. Dynamic Linking

There is, of course, another way to ensure that existing executables will continue to run: link them statically, without using any shared libraries. That will limit their

ABI issues to the kernel interface, which is normally significantly smaller than the library interface.

There is a performance tradeoff with static linking. A statically linked program does not get the benefit of sharing libraries with other programs executing at the same time. On the other hand, a statically linked program does not have to pay the performance penalty of position independent code when executing within the library.

Upgrading the shared library is only possible with dynamic linking. Such an upgrade can provide bug fixes and better performance. Also, the dynamic linker can select a version of the shared library appropriate for the specific platform, which can also help performance.

Static linking permits more reliable testing of the program. You only need to worry about kernel changes, not about shared library changes.

Some people argue that dynamic linking is always superior. I think there are benefits on both sides, and which choice is best depends on the specific circumstances.

14 Part 14

14.1 Link Time Optimization

I've already mentioned some optimizations which are peculiar to the linker: relaxation and garbage collection of unwanted sections. There is another class of optimizations which occur at link time, but are really related to the compiler. The general name for these optimizations is *link time optimization* or *whole program optimization*.

The general idea is that the compiler optimization passes are run at link time. The advantage of running them at link time is that the compiler can then see the entire program. This permits the compiler to perform optimizations which can not be done when sources files are compiled separately. The most obvious such optimization is inlining functions across source files. Another is optimizing the calling sequence for simple function—e.g., passing more parameters in registers, or knowing that the function will not clobber all registers; this can only be done

when the compiler can see all callers of the function. Experience shows that these and other optimizations can bring significant performance benefits.

Generally these optimizations are implemented by having the compiler write a version of its intermediate representation into the object file, or into some parallel file. The intermediate representation will be the parsed version of the source file, and may already have had some local optimizations applied. Sometimes the object file contains only the compiler intermediate representation, sometimes it also contains the usual object code. In the former case link time optimization is required, in the latter case it is optional.

I know of two typical ways to implement link time optimization. The first approach is for the compiler to provide a pre-linker. The pre-linker examines the object files looking for stored intermediate representation. When it finds some, it runs the link time optimization passes. The second approach is for the linker proper to call back into the compiler when it finds intermediate representation. This is generally done via some sort of plugin API.

Although these optimizations happen at link time, they are not part of the linker proper, at least not as I defined it. When the compiler reads the stored intermediate representation, it will eventually generate an object file, one way or another. The linker proper will then process that object file as usual. These optimizations should be thought of as part of the compiler.

14.2 Initialization Code

C++ permits globals variables to have constructors and destructors. The global constructors must be run before `main` starts, and the global destructors must be run after `exit` is called. Making this work requires the compiler and the linker to cooperate.

The a.out object file format is rarely used these days, but the GNU a.out linker has an interesting extension. In a.out symbols have a one byte type field. This encodes a bunch of debugging information, and also the section in which the symbol is defined. The a.out object file format only supports three sections–text, data, and bss. Four symbol types are defined as sets: text set, data set, bss set, and absolute set. A symbol with a set type is permitted to be defined multiple times. The GNU linker will not give a multiple definition error, but will instead build a table with all the values of the symbol. The table will start with one word holding

the number of entries, and will end with a zero word. In the output file the set symbol will be defined as the address of the start of the table.

For each C++ global constructor, the compiler would generate a symbol named `__CTOR_LIST__` with the text set type. The value of the symbol in the object file would be the global constructor function. The linker would gather together all the `__CTOR_LIST__` functions into a table. The startup code supplied by the compiler would walk down the `__CTOR_LIST__` table and call each function. Global destructors were handled similarly, with the name `__DTOR_LIST__`.

Anyhow, so much for a.out. In ELF, global constructors are handled in a fairly similar way, but without using magic symbol types. I'll describe what gcc does. An object file which defines a global constructor will include a `.ctors` section. The compiler will arrange to link special object files at the very start and very end of the link. The one at the start of the link will define a symbol for the `.ctors` section; that symbol will wind up at the start of the section. The one at the end of the link will define a symbol for the end of the `.ctors` section. The compiler startup code will walk between the two symbols, calling the constructors. Global destructors work similarly, in a `.dtors` section.

ELF shared libraries work similarly. When the dynamic linker loads a shared library, it will call the function at the `DT_INIT` tag if there is one. By convention the ELF program linker will set this to the function named `_init`, if there is one. Similarly the `DT_FINI` tag is called when a shared library is unloaded, and the program linker will set this to the function named `_fini`.

As I mentioned earlier, three are also `DT_INIT_ARRAY`, `DT_PREINIT_ARRAY`, and `DT_FINI_ARRAY` tags, which are set based on the `SHT_INIT_ARRAY`, `SHT_PREINIT_ARRAY`, and `SHT_FINI_ARRAY` section types. This is a newer approach in ELF, and does not require relying on special symbol names.

15 Part 15

15.1 COMDAT sections

In C++ there are several constructs which do not clearly live in a single place. Examples are inline functions defined in a header file, virtual tables, and typeinfo objects. There must be only a single instance of each of these constructs in the

final linked program (actually we could probably get away with multiple copies of a virtual table, but the others must be unique since it is possible to take their address). Unfortunately, there is not necessarily a single object file in which they should be generated. These types of constructs are sometimes described as having *vague linkage*.

Linkers implement these features by using *COMDAT* sections (there may be other approaches, but this is the only I know of). COMDAT sections are a special type of section. Each COMDAT section has a special string. When the linker sees multiple COMDAT sections with the same special string, it will only keep one of them.

For example, when the C++ compiler sees an inline function `f1` defined in a header file, but the compiler is unable to inline the function in all uses (perhaps because something takes the address of the function), the compiler will emit `f1` in a COMDAT section associated with the string `f1`. After the linker sees a COMDAT section `f1`, it will discard all subsequent `f1` COMDAT sections.

This obviously raises the possibility that there will be two entirely different inline functions named `f1`, defined in different header files. This would be an invalid C++ program, violating the One Definition Rule (often abbreviated ODR). Unfortunately, if no source file included both header files, the compiler would be unable to diagnose the error. And, unfortunately, the linker would simply discard the duplicate COMDAT sections, and would not notice the error either. This is an area where some improvements are needed (at least in the GNU tools; I don't know whether any other tools diagnose this error correctly).

The Microsoft PE object file format provides COMDAT sections. These sections can be marked so that duplicate COMDAT sections which do not have identical contents cause an error. That is not as helpful as it seems, as different compiler options may cause valid duplicates to have different contents. The string associated with a COMDAT section is stored in the symbol table.

Before I learned about the Microsoft PE format, I introduced a different type of COMDAT sections into the GNU ELF linker, following a suggestion from Jason Merrill. Any section whose name starts with ".gnu.linkonce." is a COMDAT section. The associated string is simply the section name itself. Thus the inline function `f1` would be put into the section ".gnu.linkonce.f1". This simple implementation works well enough, but it has a flaw in that some functions require data in multiple sections; e.g., the instructions may be in one section and associ-

ated static data may be in another section. Since different instances of the inline function may be compiled differently, the linker can not reliably and consistently discard duplicate data (I don't know how the Microsoft linker handles this problem).

Recent versions of ELF introduce section groups. These implement an officially sanctioned version of COMDAT in ELF, and avoid the problem of ".gnu.linkonce" sections. I described these briefly in an earlier blog entry. A special section of type `SHT_GROUP` contains a list of section indices in the group. The group is retained or discarded as a whole. The string associated with the group is found in the symbol table. Putting the string in the symbol table makes it awkward to retrieve, but since the string is generally the name of a symbol it means that the string only needs to be stored once in the object file; this is a minor optimization for C++ in which symbol names may be very long.

16 Part 16

16.1 C++ Template Instantiation

There is still more C++ fun at link time, though somewhat less related to the linker proper. A C++ program can declare templates, and instantiate them with specific types. Ideally those specific instantiations will only appear once in a program, not once per source file which instantiates the templates. There are a few ways to make this work.

For object file formats which support COMDAT and vague linkage, which I described yesterday, the simplest and most reliable mechanism is for the compiler to generate all the template instantiations required for a source file and put them into the object file. They should be marked as COMDAT, so that the linker discards all but one copy. This ensures that all template instantiations will be available at link time, and that the executable will have only one copy. This is what gcc does by default for systems which support it. The obvious disadvantages are the time required to compile all the duplicate template instantiations and the space they take up in the object files. This is sometimes called the Borland model, as this is what Borland's C++ compiler did.

Another approach is to not generate any of the template instantiations at compile time. Instead, when linking, if we need a template instantiation which is not

found, invoke the compiler to build it. This can be done either by running the linker and looking for error messages or by using a linker plugin to handle an undefined symbol error. The difficulties with this approach are to find the source code to compile and to find the right options to pass to the compiler. Typically the source code is placed into a repository file of some sort at compile time, so that it is available at link time. The complexities of getting the compilation steps right are why this approach is not the default. When it works, though, it can be faster than the duplicate instantiation approach. This is sometimes called the Cfront model.

gcc also supports explicit template instantiation, which can be used to control exactly where templates are instantiated. This approach can work if you have complete control over your source code base, and can instantiate all required templates in some central place. This approach is used for gcc's C++ library, `libstdc++`.

C++ defines a keyword `export` which is supposed to permit exporting template definitions in such a way that they can be read back in by the compiler. gcc does not support this keyword. If it worked, it could be a slightly more reliable way of using a repository when using the Cfront model.

16.2 Exception Frames

C++ and other languages support exceptions. When an exception is thrown in one function and caught in another, the program needs to reset the stack pointer and registers to the point where the exception is caught. While resetting the stack pointer, the program needs to identify all local variables in the part of the stack being discarded, and run their destructors if any. This process is known as *unwinding* the stack.

The information needed to unwind the stack is normally stored in tables in the program. Supporting library code is used to read the tables and perform the necessary operations. I'm not going to describe the details of those tables here. However, there is a linker optimization which applies to them.

The support libraries need to be able to find the exception tables at runtime when an exception occurs. An exception can be thrown in one shared library and caught in a different shared library, so finding all the required exception tables can be a nontrivial operation. One approach that can be used is to register the exception tables at program startup time or shared library load time. The registration

can be done at the right time using the global constructor mechanism.

However, this approach imposes a runtime cost for exceptions, in that it takes longer for the program to start. Therefore, this is not ideal. The linker can optimize this by building tables which can be used to find the exception tables. The tables built by the GNU linker are sorted for fast lookup by the runtime library. The tables are put into a `PT_GNU_EH_FRAME` segment. The supporting libraries then need a way to look up a segment of this type. This is done via the `dl_iterate_phdr` API provided by the GNU dynamic linker.

Note that if the compiler believes that the linker will generate a `PT_GNU_EH_FRAME` segment, it won't generate the startup code to register the exception tables. Thus the linker must not fail to create this segment.

Since the GNU linker needs to look at the exception tables in order to generate the PT_GNU_EH_FRAME segment, it will also optimize by discarding duplicate exception table information.

I know this is section is rather short on details. I hope the general idea is clear.

17 Part 17

17.1 Warning Symbols

The GNU linker supports a weird extension to ELF used to issue warnings when symbols are referenced at link time. This was originally implemented for a.out using a special symbol type. For ELF, I implemented it using a special section name.

If you create a section named `.gnu.warning.SYMBOL`, then if and when the linker sees an undefined reference to `SYMBOL`, it will issue a warning. The warning is triggered by seeing an undefined symbol with the right name in an object file. Unlike the warning about an undefined symbol, it is not triggered by seeing a relocation entry. The text of the warning is simply the contents of the `.gnu.warning.SYMBOL` section.

The GNU C library uses this feature to warn about references to symbols like `gets` which are required by standards but are generally considered to be unsafe.

This is done by creating a section named `.gnu.warning.gets` in the same object file which defines `gets`.

The GNU linker also supports another type of warning, triggered by sections named `.gnu.warning` (without the symbol name). If an object file with a section of that name is included in the link, the linker will issue a warning. Again, the text of the warning is simply the contents of the `.gnu.warning` section. I don't know if anybody actually uses this feature.

18 Part 18

18.1 Incremental Linking

Often a programmer will make change a single source file and recompile and relink the application. A standard linker will need to read all the input objects and libraries in order to regenerate the executable with the change. For a large application, this is a lot of work. If only one input object file changed, it is a lot more work than really needs to be done. One solution is to use an *incremental linker*. An incremental linker makes incremental changes to an existing executable or shared library, rather than rebuilding them from scratch.

I've never actually written or worked on an incremental linker, but the general idea is straightforward enough. When the linker writes the output file, it must attach additional information.

- The linker must create a mapping of object files to areas in the output file, so that an incremental link will know what to remove when replacing an object file.
- The linker must retain all the relocations for each input object which refer to symbols defined in other objects, so that it can reprocess them when symbols change. The linker should store the relocations mapped by symbol, so that it can quickly find the relevant relocations.
- The linker should leave extra space in the text and data segments, to allow for object files to grow to a limited extent without requiring rewriting the whole executable. It must keep a map of where this extra space is, as it will tend to move over time over the course of incremental links.

- The linker should keep a list of object file timestamps in the output file, so that it can quickly determine which objects have changed.

With this information, the linker can identify which object files have changed since the last time the output file was linked, and replace them in the existing output file. When an object file changes, the linker can identify all the relocations which refer to symbols defined in the object file, and reprocess them.

When an object file gets too large to fit in the available space in a text or data segment, then the linker has the option of creating additional text or data segments at different addresses. This requires some care to ensure that the new code does not collide with the heap, depending upon how the local `malloc` implementation works. Alternatively, the incremental linker could fall back on doing a full link, and allocating more space again.

Incremental linking can greatly speed up the edit/compile/debug cycle. Unfortunately it is not implemented in most common linkers. Of course an incremental link is not equivalent to a final link, and in particular some linker optimizations are difficult to implement while acting incrementally. An incremental link is really only suitable for use during the development cycle, which is course the time when the speed of the linker is most important.

19 Part 19

I've pretty much run out of linker topics. Unless I think of something new, ll I'make tomorrow's post be the last one, for a total of 20.

`__start` and `__stop` Symbols

A quick note about another GNU linker extension. If the linker sees a section in the output file which can be part of a C variable nam–the name contains only alphanumeric characters or underscore–the linker will automatically define symbols marking the start and stop of the section. Note that this is not true of most section names, as by convention most section names start with a period. But the name of a section can be any string; it doesn't have to start with a period. And when that happens for section NAME, the GNU linker will define the symbols `__start_NAME` and `__stop_NAME` to the address of the beginning and the end of section, respectively.

This is convenient for collecting some information in several different object files, and then referring to it in the code. For example, the GNU C library uses this to keep a list of functions which may be called to free memory. The `__start` and `__stop` symbols are used to walk through the list.

In C code, these symbols should be declared as something like `extern char __start_NAME[]`. For an extern array the value of the symbol and the value of the variable are the same.

19.1 Byte Swapping

The new linker I am working on, gold, is written in C++. One of the attractions was to use template specialization to do efficient byte swapping. Any linker which can be used in a cross-compiler needs to be able to swap bytes when writing them out, in order to generate code for a big-endian system while running on a little-endian system, or vice-versa. The GNU linker always stores data into memory a byte at a time, which is unnecessary for a native linker. Measurements from a few years ago showed that this took about 5% of the linker's CPU time. Since the native linker is by far the most common case, it is worth avoiding this penalty.

In C++, this can be done using templates and template specialization. The idea is to write a template for writing out the data. Then provide two specializations of the template, one for a linker of the same endianness and one for a linker of the opposite endianness. Then pick the one to use at compile time. The code looks this; I'm only showing the 16-bit case for simplicity.

```
// Endian simply indicates whether the host is big endian or not.

struct Endian
{
    public:
    // Used for template specializations.
    static const bool host_big_endian = __BYTE_ORDER == __BIG_ENDIAN;
};

// Valtype_base is a template based on size (8, 16, 32, 64) which
// defines the type Valtype as the unsigned integer of the specified
// size.

template
struct Valtype_base;
```

```
template<>
struct Valtype_base<16>
{
    typedef uint16_t Valtype;
};

// Convert_endian is a template based on size and on whether the host
// and target have the same endianness. It defines the type Valtype
// as Valtype_base does, and also defines a function convert_host
// which takes an argument of type Valtype and returns the same value,
// but swapped if the host and target have different endianness.

template
struct Convert_endian;

template
struct Convert_endian
{
    typedef typename Valtype_base::Valtype Valtype;

    static inline Valtype
    convert_host(Valtype v) { return v; }
};

template<>
struct Convert_endian<16, false>
{
    typedef Valtype_base<16>::Valtype Valtype;

    static inline Valtype
    convert_host(Valtype v) { return bswap_16(v); }
};

// Convert is a template based on size and on whether the target is
// big endian. It defines Valtype and convert_host like
// Convert_endian. That is, it is just like Convert_endian except in
// the meaning of the second template parameter.

template
struct Convert
{
    typedef typename Valtype_base::Valtype Valtype;

    static inline Valtype
```

```
    convert_host(Valtype v)
    {
        return Convert_endian
        ::convert_host(v);
    }
};

// Swap is a template based on size and on whether the target is big
// endian. It defines the type Valtype and the functions readval and
// writeval. The functions read and write values of the appropriate
// size out of buffers, swapping them if necessary.

template
struct Swap
{
    typedef typename Valtype_base::Valtype Valtype;

    static inline Valtype
    readval(const Valtype* wv) { return Convert::convert_host(*wv); }

    static inline void
    writeval(Valtype* wv, Valtype v) { *wv = Convert::convert_host(v); }
};
```

Now, for example, the linker reads a 16-bit big-endian value using `Swap<16,true>::readval`. This works because the linker always knows how much data to swap in, and it always knows whether it is reading big- or little-endian data.

20 Part 20

This will be my last blog posting on linkers for the time being. Tomorrow my blog will return to its usual trivialities. People who are specifically interested in linker information are warned to stop reading with this post.

I'll close the series with a short update on gold, the new linker I've been working on. It currently (September 25, 2007) can create executables. It can not create shared libraries or relocateable objects. It has very limited support for linker scripts–enough to read `/usr/lib/libc.so` on a GNU/Linux system. It doesn't have any interesting new features at this point. It only supports x86. The focus to date has been entirely on speed. It is written to be multi-threaded, but the thread-

ing support has not been hooked in yet.

By way of example, when linking a 900M C++ executable, the GNU linker (version 2.16.91 20060118 on an Ubuntu based system) took 700 seconds of user time, 24 seconds of system time, and 16 minutes of wall time. gold took 7 seconds of user time, 3 seconds of system time, and 30 seconds of wall time. So while I can't promise that it will stay as fast as all features are added, it's in a pretty good position at the moment.

I'm the main developer on gold, but I'm not the only person working on it. A few other people are also making improvements.

The goal is to release gold as a free program, ideally as part of the GNU binutils. I want it to be more nearly feature complete before doing this, though. It needs to at least support `-shared` and `-r`. I doubt gold will ever support all of the features of the GNU linker. I doubt it will ever support the full GNU linker script language, although I do plan to support enough to link the Linux kernel.

Future plans for gold, once it actually works, include incremental linking and more far-reaching speed improvements.