Relocation

Once a linker has scanned all of the input files to determine segment sizes, symbol definitions and symbol references, figured out which library modules to include, and decided where in the output address space all of the segments will go, the next stage is the heart of the linking process, relocation. We use relocation to refer both to the process of adjusting program addresses to account for non-zero segment origins, and the process of resolving references to external symbols, since the two are frequently handled together.

The linker's first pass lays out the positions of the various segments and collects the segment-relative values of all global symbols in the program. Once the linker determines the position of each segment, it potentially needs to fix up all storage addresses to reflect the new locations of the segments. On most architectures, addresses in data are absolute, while those embedded in instructions may be absolute or relative. The linker needs to fixup accordingly, as we'll discuss later.

The first pass also creates the global symbol table as described in Chapter 5. The linker also resolves stored references to global symbols to the symbols' addresses.

Hardware and software relocation

Since nearly all modern computers have hardware relocation, one might wonder why a linker or loader still does software relocation. (This question confused me when programming a PDP-6 in the late 1960s, and the situation has only gotten more complicated since then.) The answer has partly to do with performance, and partly with binding time.

Hardware relocation allows an operating system to give each process a separate address space that starts at a fixed known address, which makes program loading easier and prevents buggy programs in one address space from damaging programs in other address spaces. Software linker or loader relocation combines input files into one large file that's ready to be loaded into the address space provided by hardware relocation, frequently with no load-time fixing up at all.

On a machine like a 286 or 286 with several thousand segments, it would indeed be possible to load one routine or global datum per segment, completely doing away with software relocation, since each routine or datum starts at location zero in its segment, and all global references are handled as inter-segment references looked up in the system's segment tables and bound at runtime, Unfortunately, segment lookups are very slow, and a program that did a segment lookup for every inter-module call or global data refrence would be far slower than one linked conventionally, the main reason that nobody does that, even on systems where it is in theory possible.

Equally importantly, although runtime binding can be useful (a topic we cover in Chapter 10), most programs are better off avoiding it. For reliability reasons, program files are best bound together and addresses fixed at link time, so they hold still during debugging and remain consistent after shipping. Library "bit creep" is a chronic and very hard to debug source of program errors when a program runs using different versions of libraries than its authors anticipated. (MS Windows applications are prone to this problem due to the large number of shared libraries they use, with different versions of libraries often shipped with various applications all loaded on the same computer.) Even without the overhead of 286 style segments, dynamic linking is far slower than static linking, and there's no point in paying for it where it's not needed.

Link time and load time relocation

Many systems perform both link time and load time relocation. A linker combines a set of input file into a single output file ready to be loaded at specific address. If when the program is loaded, storage at that address isn't available, the loader has to relocate the loaded program to reflect the actual load address. On some systems including MS-DOS and MVS, every program is linked as though it will be loaded at location zero, the actual address is chosen from available storage and the program is always relocated as it's loaded. On others, notably MS Windows, programs are linked to be loaded at a fixed address which is generally available, and no load-time relocation is needed except in the unusual case that the standard address is already in use by something else. (Current versions of Windows in practice never do load-time relocation of executable programs, although they do relocate DLL shared libraries. Similarly Unix systems never relocate ELF programs although they sometimes do relocate ELF shared libraries.)

Load-time relocation is quite simple compared to link-time relocation. At link time, different addresses need to be relocated different amounts depending on the size and locations of the segments. At load time, on the other hand, the entire program is invariably treated as a single big segment for relocation purposes, and the loader needs only to adjust program addresses by the difference between the nominal and actual load addresses.

Symbol and segment relocation

The linker's first pass lays out the positions of the various segments and collects the segment-relative values of all global symbols in the program. Once the linker determines the position of each segment, it needs to adjust the stored addresses.

- Data addresses and absolute program address references within a segment need to be adjusted. For example, if a pointer refers to location 100, but the segment base is relocated to 1000, the pointer needs to be adjusted to location 1100.
- Inter-segment references need to be adjusted as well. Absolute addresses need to be adjusted to reflect the new position of the target address' segment, relative addresses need to reflect the positions of both segments.
- References to global symbols have to be resolved. If an instruction calls a routine `detonate`, and `detonate` is at offset 500 in a segment that starts at 1000, the address in that instruction has to be adjusted to refer to location 1500. The requirements of relocation and symbol resolution are slightly different. For relocation, the number of base values is fairly small, the number of segments in an input file, but the object format has to permit relocation of any address in any segment. For symbol resolution, the number of symbols is far greater, but in most cases the only action the linker needs to take with the symbol is to plug the symbol's value into word in the program.

Many linkers unify segment and symbol relocation by treating each segment as a pseudo-symbol whose value is the base of the segment. This makes segment-relative relocations a special case of symbol-relative ones.

One minor difference is that a symbol involves two addends, the base address of the segment in which the symbol resides and the offset of the symbol within that segment. Some linkers precompute all the symbol addresses before starting the relocation phase, adding the segment base to the symbol value in the symbol table. Others look up the segment base do the addition as each item is relocated. In most cases, there's no compelling reason to do it one way or the other. In a few linkers, notably those for real-mode x86 code, a single location can be addressed relative to several different segments, so the linker can only determine the address to use for a symbol in the context of an individual reference using a specified segment.

Symbol lookups

Object formats invariably treat each file's set of symbols as an array, and internally refer to the symbols using a small integer, the index in that array. This causes minor complications for the linker, as mentioned in Chapter 5, since each input file will have different indexes, as will the output if the output is relinkable. The most straightforward way to handle this is to keep an array of pointers for each input file, pointing to entries in the global symbol table.

Basic relocation techniques

Object files contain a relocation table, a list of places in each segment that need to be relocated. The linker reads in the contents of the segment, applies the relocation items, then disposes of the segment, usually by writing it to the output file. Usually but not always, relocation is a one-time operation and the resulting file can't be relocated again. Some object formats, notably the IBM 360, are relinkable and keep all the relocation data in the output file. (In the case of the 360, the output file needs to be relocated when loaded, so it has to keep all the relocation information anyway.) With Unix linkers, a linker option makes the output relinkable, and in some cases, notably shared libraries, the output always has relocation information since libraries need to be relocated when loaded as well.

In the simplest case, Figure 1, the relocation information for a segment is just a list of places in the segment that need to be relocated. As the linker processes the segment, it adds the base position of the segment to the value at each location identified by a location entry. This handles direct addressing and pointer values in memory for a single segment.

Figure 1: Simple relocation entry

address | address | address | ...

Real programs are somewhat more complicated, due to multiple segments and addressing modes. The classic Unix a.out format, Figure 2, is about the simplest that handles these issues.

Figure 2: a.out relocation entry

```
int address /* offset in text or data segment */
unsigned int r_symbolnum : 24,  /* ordinal number of add symbol */
r_pcrel :  1,  /* 1 if value should be pc-relative */
r_length :  2,  /* log base 2 of value's width */
r_extern :  1,  /* 1 if need to add symbol to value */
```

Each object file has two sets of relocation entries, one for the text segment and one for the data segment. (The bss segment is defined to be all zero, so there's nothing to relocate there.) Each relocation entry contains a bit `r_extern` to specify whether this is a segment-relative or symbol-relative entry. If the bit is clear, it's segment relative and `r_symbolnum` is actually a code for the segment, `N_TEXT` (4), `N_DATA` (6), or `N_BSS` (8). The `pc_relative` bit specifies whether the reference is absolute or relative to the current location ("program counter".)

The exact details of each relocation depend on the type and segments involved. In the discussion below, TR, DR, and BR are the relocated bases of the text, data, and bss segments, respectively.

For a pointer or direct address within the same segment, the linker adds TR or DR to the stored value already in the segment.

For a pointer or direct address from one segment to another, the linker adds the relocated base of the target segment, TR, DR, or BR to the stored value. Since a.out input files already have the target addresses in each segment relocated to the tentative segment positions in the new file, this is all that's necessary. For example, assume that in the input file, the text starts at 0 and data at 2000, and a pointer in the text segment points to offset 100 in the data segment. The in the input file, the stored pointer will have the value 2200. If the final relocated address of the data segment in the output turns out to be 15000, then DR will be 13000, and the linker will add 13000 to the existing 2200 producing a final stored value of 15200.

Some architectures have different sizes of addresses. Both the IBM 360 and Intel 386 have both 16 and 32 bit addresses, and the linkers have generally supported relocation items of both sizes. In both cases, it's up to the programmer who uses 16 bit addresses to make sure that the addresses will fit in the 16 bit fields; the linker doesn't do any more than verify that the address fits.

Instruction relocation

Relocating addresses in instructions is somewhat trickier due to the profusion of often quirky instruction formats. The a.out format described above has only two relocation formats, absolute and pc-relative, but most computer architectures require a longer list of relocation formats to handle all the instruction formats.

X86 instruction relocation

Despite the complex instruction encodings on the x86, from the linker's point of view the architecture is easy to handle because there are only two kinds of addresses the linker has to handle, direct and pc-relative. (We ignore segmentation here, as do most 32 bit linkers.) Data reference instructions can contain the 32 bit address of the target, which the linker can relocate the same as any other 32 bit data address, adding the relocated base of the segment in which the target resides.

Call and jump instructions use relative addressing, so the value in the instruction is the difference between the target address and the address of the instruction itself. For calls and jumps within the same segment, no relocation is required. For intersegment jumps the linker needs to add the relocation for the target segment and subtract that of the instruction's segment. For a jump from the text to the data segment, for example, the relocation value to apply would be DR-TR.

SPARC instruction relocation

Most architectures do not have instruction encodings as linker-friendly as the x86. The SPARC, for example, has no direct addressing, four different branch formats, and some specialized instructions used to synthesize a 32 bit address, with individual instructions only containing part of an address. The linker needs to handle all of this.

Unlike the x86, none of the SPARC instruction formats have room for a 32 bit address in the instruction itself. This means that in the input files, the target of an instruction with a relocatable memory reference can't be stored in the instruction itself since there may not be room to do so. As a result, SPARC relocation entries, Figure 3, need an extra field `r_addend` which contains the 32 bit value to which the reference is made. Since SPARC relocation can't be described as simply as x86, the various type bits are replaced by a `r_type` field that contains a code that describes the format of the relocation. Also, rather than dedicate a bit to distinguishing between segment and symbol relocations, each input file defines symbols `.text`, `.data`, and `.bss` and segment relocations refer to those symbols.

Figure 3: SPARC relocation entry

```
int r_address;   /* offset of of data to relocate */
int_byte r_index:24,/* symbol table index of symbol */
   r_type:8;   /* relocation type*/
int r_addend;  /* datum addend*/
```

The SPARC relocations divide up into three categories: absolute addresses for pointers in data, relative addresses of various sizes for branches and calls, and the special SETHI absolute address hack. Absolute addresses are handled almost the same as on the x86, the linker adds TR, DR, or BR to the stored value. In this case the addend isn't really needed, since there's room for a full address in the stored value, but the linker adds the addend to the stored value anyway for consistency.

For branches, the stored offset value is generally zero, with the addend being the relative address of the target, the difference between the target and the address of the stored value. The linker adds the appropriate relocation value to the addend to get the relocated relative address. Then it shifts the relative address right two bits, since SPARC relative addresses are stored without the low bits, checks to make sure that the shifted value will fit in the number of bits available (16, 19, 22, or 30 depending on format), masks the shifted address to that number of bits and adds it into the instruction. The 16 bit format stores 14 low bits in the low bits of the word, but the 15th and 16th bits are in bit positions 20 and 21. The linker does the appropriate shifting and masking to store those bits without modifying the intervening bits.

The special SETHI hack synthesizes a 32 bit address with a SETHI instruction, which takes a 22 bit value from the instruction and places it in the 22 high bits of a register, followed by an OR immediate to the same register which provides the low 10 bits of the address. The linker handles this with two specialized relocation modes, one of which puts the 22 high bits of the relocated address (the addend plus the appropriate relocated segment base) in the low 22 bits of the stored value, and a second mode which puts the low 10 bit of the relocated address in the low 10 bits of the stored value. Unlike the branch modes above, these relocation modes do *not* check that each value fits in the stored bits.

Relocation on other architectures use variations on the SPARC techniques, with a different relocation type for each instruction format that can address memory.

ECOFF segment relocation

Microsoft's COFF object format is an extended version of COFF which is descended from a.out, so it's not surprising that Win32 relocation bears a lot of similarities to a.out relocation. Each section in a COFF object file can have a list of relocation entries similar to a.out entries, Figure 4. A peculiarity of COFF relocation entries is that even on 32 bit machines, they're 10 bytes long, which means that on machines that require aligned data, the linker can't just load the entire relocation table into a memory array with a single read, but rather has to read and unpack entries one at a time. (COFF is old enough that saving two bytes per entry probably seemed important.) In each entry, the address is the RVA (relative virtual address) of the stored data, the index is the segment or symbol index, and the type is a machine specific relocation type. Each section of the input file has a symbol table entry with a name like `.text`, so segment relocations use the index of the symbol corresponding to the target section.

Figure 4: MS COFF relocation entry

```
int address;  /* offset of of data to relocate */
int index;  /* symbol index */
short type; /* relocation type */
```

On the x86, ECOFF relocations work much like they do in a.out. An IMAGE_REL_I386_DIR32 is a 32 bit direct address or stored pointer, and an An IMAGE_REL_I386_DIR32NB is 32 bit direct address or stored pointer relative to the base of the progam, and an IMAGE_REL_I386_REL32 is a pc-relative 32 bit address. A few other relocation types support special Windows features, mentioned later.

ECOFF supports several RISC processors including the MIPS, Alpha, and Power PC. These processors all present the same relocation issues the SPARC does, branches with limited addressing and multi-instruction sequences to synthesize a direct address. ECOFF has relocation types to handle each of those situations, along with the conventional full-word relocations.

The MIPS, for example, has a jump instruction with a 26 bit address which is shifted two bits to the left and placed in the 28 low bits of the program counter, leaving the high four bits unchanged. The relocation type IMAGE_REL_MIPS_JMPADDR relocates a branch target address. Since there's no place in the relocation item for the target address, the stored instruction already contains the unrelocated target address. To do the relocation, the linker has to reconstruct the unrelocated target address by extracting the low 26 bits of the stored instruction, shifting and masking, then add the relocated segment base for the target segment, then undo the shifting and masking to reconstruct the instruction.

The MIPS also has an equivalent of the SETHI trick. MIPS data reference instructions have 16 bit literal values, so to load an arbitrary 32 bit value one uses a LUI (load upper immediate) instruction to place the high half of an immediate value in the high 16 bits of a register, followed by an ORI (OR immediate) to place the low 16 bits in the register. The relocation types IMAGE_REL_MIPS_REFHI and IMAGE_REL_MIPS_REFLO support this trick, telling the linker to relocate the high or low half, respectively, of the target value in the relocated instruction. REFHI presents a problem though. Imagine that the target address before relocation is hex 00123456, so the stored instruction would contain 0012, the high half of the unrelocated value. Now imagine that the relocation value is 1E000. The final value will be 123456 plus 1E000 which is 141456, so the stored value will be 0014. But wait – to do this calculation, the linker needs the full value 00123456, but only the 0012 is stored. Where does it find the low half with 3456? ECOFF's answer is that the next relocation item after the REFHI is IMAGE_REL_MIPS_PAIR, in which the index contains the low half of the target for a preceding REFHI. This is arguably a better approach than using an extra addend field in each relocation item, since the PAIR item only occurs after REFHI, rather than wasting space in every item. The disadvantage is that the order of relocation items now becomes important, while it wasn't before.

ELF relocation

ELF relocation is similar to a.out and COFF relocation. ELF does rationalize the issue of relocation items with addends and those without, having two kinds of relocation sections, SNT_REL without and SHT_RELA with. In practice, all of the relocation sections are of the same type, depending on the architecture. If the architecture has room for all the addends in the object code like the x86 does, it uses REL, otherwise RELA. But in principle a compiler could save some space on architectures that need addends by putting all the relocations with zero addends, e.g., procedure references, in a SHT_REL section and the rest in a SHT_RELA.

ELF also adds some extra relocation types to handle dynamic linking and position independent code, that we discuss in Chapter 8.

OMF relocation

OMF relocation is conceptually the same as the schemes we've already looked at, although the details are quite complex. Since OMF was originally designed for use on microcomputers with limited memory and storage, the format permits relocation to take place without having to load an entire segment into memory. OMF intermixes LIDATA or LEDATA data records with FIXUPP relocation records, with each FIXUPP referring to the preceding data. Hence, the linker can read and buffer a data record, then read a following FIXUPP, apply the relocations, and write out the relocated data. FIXUPPs refer to relocation ‘‘threads’’, two-bit codes that indirectly refer to a frame, an OMF reloctation base. The linker has to track the four active frames, updating them as FIXUPP records redefine them, and using them as FIXUPP records refer to them.

Relinkable and relocatable output formats

A few formats are relinkable, which means that the output file has a symbol table and relocation information so it can be used as an input file in a subsequent link. Many formats are relocatable, which means that the output file has relocation information for load-time relocation.

For relinkable files, the linker needs to create a table of output relocation entries from the input relocation entries. Some entries can be passed through verbatim, some modified, and some discarded. Entries for segment-relative fixups in formats that don't combine segments can generally be passed through unmodified other than adjusting the segment index, since the final link will handle the relocation. In formats that do combine segments, the item's offset needs to be adjusted. For example, in linked a.out file, an incoming text segment has a segment-relative relocation at offset 400, but that segment is combined with other text segments so the code from that segment is at location 3500. Then the relocation item is modified to refer to location 3900 rather than 400.

Entries for symbol resolution can be passed through unmodified, changed to segment relocations, or discarded. If an external symbol remains undefined, the linker passes through the relocation item, possibly adjusting the offset and symbol index to reflect combined segments and the order of symbols in the output files' symbol table. If the symbol is resolved, what the linker does depends on the details of the symbol reference. If the reference is a pc-relative one within the same segment, the linker can discard the relocation entry, since the relative positions of the reference and the target won't move. If the reference is absolute or inter-segment, the relocation item turns into a segment-relative one.

For output formats that are relocatable but not relinkable, the linker discards all relocation items other than segment-relative fixups.

Other relocation formats

Although the most common format for relocation items is an array of fixups, there are a couple of other possiblities.

Chained references

For external symbol references, one surprisingly effective format is a linked list of references, with the links in the object code itself. The symbol table entry points to one reference, the word at that location points to a subsequent reference, and so forth to the final reference which has a stop value such as zero or -1. This works on architectures where address references are a full word, or at least enough bits to cover the maximum size of an object file segment. (SPARC branches, for example, have a 22 bit offset which, since instructions are aligned on four-byte boundaries, is enough to cover a 2^{24} byte section, which is a reasonable limit on a single file segment.)

This trick does not handle symbol references with offsets, which is usually an acceptable limitation for code references but a problem for data. In C, for example, one can write static initializers which point into the middle of arrays:

```
extern int a[];
static int *ap = &a[3];
```

On a 32 bit machine, the contents of `ap` are `a` plus 12. A way around this problem is either to use this technique just for code pointers, or else to use the link list for the common case of references with no offset, and something else for references with offsets.

Bit maps

On architectures like the PDP-11, Z8000, and some DSPs that use absolute addressing, code segments can end up with a lot of segment relocations since most memory reference instructions contain an address that needs to be relocated. Rather than making a list of locations to fix up, it can be more efficient to store fixups as a bit map, with one bit for every word in a segment, the bit being set if the location needs to be fixed up. On 16 bit architectures, a bit map saves space if more than 1/16 of the words in a segment need relocation; on a 32 bit architecture if more than 1/32 of the words need relocation.

Special segments

Many object formats define special segment formats that require special relocation processing.

- Windows objects have thread local storage (TLS), a special segment containing global variables that is replicated for each thread started within a process.
- IBM 360 objects have "pseudoregisters", similar to thread local storage, an area with named subchunks referred to from different input files.

- Many RISC architectures define "small" segments that are collected together into one area, with a register set at program startup to point to that area allowing direct addressing from anywhere in the program. In each of these cases, the linker needs a special relocation type or two to handle special segments.

For Windows thread local storage, the details of the relocation type(s) vary by architecture. For the x86, IMAGE_REL_I386_SECREL fixups store the target symbol's offset from the beginning of its segment. This fixup is generally an instruction with an index register that is set at runtime to point to the current thread's TLS, so the SECREL provides the offset within the TLS. For the MIPS and other RISC processors, there are both SECREL fixups to store a 32 bit value as well as SECRELLO and SECRELHI (the latter followed by a PAIR, as with REFHI) to generate section-relative addresses.

For IBM pseudoregisters, the object format adds two relocation types. One is a PR pseudoregister reference, which stores the offset of the pseudoregister, typically into two bytes in a load or store intstruction. The other is CXD, the total size of the pseudoregisters used in a program. This value is used by runtime startup code to determine how much storage to allocate for a set of pseudoregisters.

For small data segments, object formats define a relocation type such as GPREL (global pointer relocation) for MIPS or LITERAL for Alpha which stores the offset of the target date in the small data area. The linker defines a symbol like `_GP` as the base of the small data area, so that runtime startup code can load a pointer to the area into a fixed register.

Relocation special cases

Many object formats have "weak" external symbols which are treated as normal global symbols if some input file happens to define them, or zero otherwise. (See Chapter 5 for details.) These usually require no special effort in the relocation process, since the symbol is either a normal defined global, or else it's zero. Either way, references are resolved like any other symbol.

Some older object formats permitted much more complex relocation than the formats we've discussed here. The IBM 360 format, for example, each relocation item can either add or subtract the address to which it refers, and multiple relocation items can modify the same location, permitting references like `A-B` where either or both A and B are external symbols.

Some older linkers permitted arbitrarily complex relocations, with elaborate reverse polish strings representing link-time expressions to be resolved and stored into program memory. Although these schemes had great expressive power, it turned out to be power that wasn't very useful, and modern linkers have retreated to references with optional offsets.

Exercises

Why does a SPARC linker check for address overflow when relocating branch addresses, but not when doing the high and low parts of the addresses in a SETHI sequence?

In the MIPS example, a REFHI relocation item needs a following PAIR item, but a REFLO doesn't. Why not?

References to symbols that are pseudo-registers and thread local storage are resolved as offsets from the start of the segment, while normal symbol references are resolved as absolute addresses. Why?

We said that a.out and COFF relocation doesn't handle references like A-B where A and B are both global symbols. Can you come up with a way to fake it?