Chapter 8
Loading and overlays

$Revision: 2.3 $
$Date: 1999/06/15 03:30:36 $

Loading is the process of bringing a program into main memory so it can run. In this chapter we look at the loading process, concentrating on loading programs that have already been linked. Many systems used to have linking loaders that combined the linking and loading process, but those have now practically disappeared, with the only one I know of on current hardware being on MVS and the dynamic linkers we'll cover in chapter 10. Linking loaders weren't all that different from plain linkers, with the primary and obvious difference being that the output was left in memory rather than placed in a file.

Basic loading

We touched on most of the basics of loading in Chapter 3, in the context of object file design. Loading is a little different depending on whether a program is loaded by mapping into a process address space via the virtual memory system or just read in using normal I/O calls.

On most modern systems, each program is loaded into a fresh address space, which means that all programs are loaded at a known fixed address, and can be linked for that address. In that case, loading is pretty simple:

- Read enough header information from the object file to find out how much address space is needed.
- Allocate that address space, in separate segments if the object format has separate segments.
- Read the program into the segments in the address space.
- Zero out any bss space at the end of the program if the virtual memory system doesn't do so automatically.
- Create a stack segment if the architecture needs one.

- Set up any runtime information such as program arguments or environment variables.
- Start the program.
  If the program isn't mapped through the virtual memory system, reading in the object file just means reading in the file with normal "read" system calls. On systems which support shared read-only code segments, the system needs to check whether there's already a copy of the code segment loaded in and use that rather than making another copy.

On systems that do memory mapping, the process is slightly more complicated. The system loader has to create the segments, then arrange to map the file pages into the segments with appropriate permissions, read-only (RO) or copy-on-write (COW). In some cases, the same page is double mapped at the end of one segment and the beginning of the next, RO in one and COW in the other, in formats like compact Unix a.out. The data segment is generally contiguous with the bss segment, so the loader has to zero out the part of the last page after the end of the data (since the disk version usually has symbols or something else there), and allocate enough zero pages following the data to cover the bss segment.

Basic loading, with relocation

A few systems still do load time relocation for executables, and many do load time relocation of shared libraries. Some, like MS-DOS, lack usable hardware relocation. Others, like MVS, have hardware relocation but are descended from systems that didn't have it. Some have hardware relocation but can load multiple executable programs and shared libraries into the same address space, so linkers can't count on having specific addresses available.

As discussed in Chapter 7, load-time relocation is far simpler than link-time relocation, because the entire program is relocated as a unit. If, for example, the program is linked as though it would be loaded at location zero, but is in fact loaded at location 15000, all of the places in the program that require fixups will get 15000 added. After reading the program into memory, the loader consults the relocation items in the object file and fixes up the memory locations to which the items point.

Load-time relocation can present a performance problem, because code loaded at different virtual addresses can't usually be shared between address spaces, since the fixups for each address space are different. One approach, used by MVS, and to some extent by Windows and AIX is to create a shared memory area present in multiple address spaces and load oft-used programs into that. (MVS calls this this link pack area.) This has the problem that different processes don't get separate copies of writable data, so the application has to be written to allocate all of its writable storage explicitly.

Position-independent code

One popular solution to the dilemma of loading the same program at different addresses is position independent code (PIC). The idea is simple, separate the code from the data and generate code that won't change regardless of the address at which it's loaded. That way the code can be shared among all processes, with only data pages being private to each process.

This is a surprisingly old idea. TSS/360 used it in 1966, and I don't believe it was original there. (TSS was notoriously buggy, but I can report from personal experience that the PIC features really worked.)

On modern architectures, it's not difficult to generate PIC executable code. Jumps and branches are generally either PC-relative or relative to a base register set at runtime, so no load-time relocation is required for them. The problem is with data addressing. The code can't contain any direct data addresses, since those would be relocatable and wouldn't be PIC. The usual solution is to create a table of data addresses in a data page and keep a pointer to that table in a register, so the code can use indexed addressing relative to that register to pick up the data. This works at the cost of an extra indirection for each data reference, but there's still the question of how to get the initial data address into the register. ,

TSS/360 position independent code

TSS took a brute-force approach. Every routine had two addresses, the address of the code, known as the V-con (short for V style address constant, which even non-PIC code needed) and the address of the data,

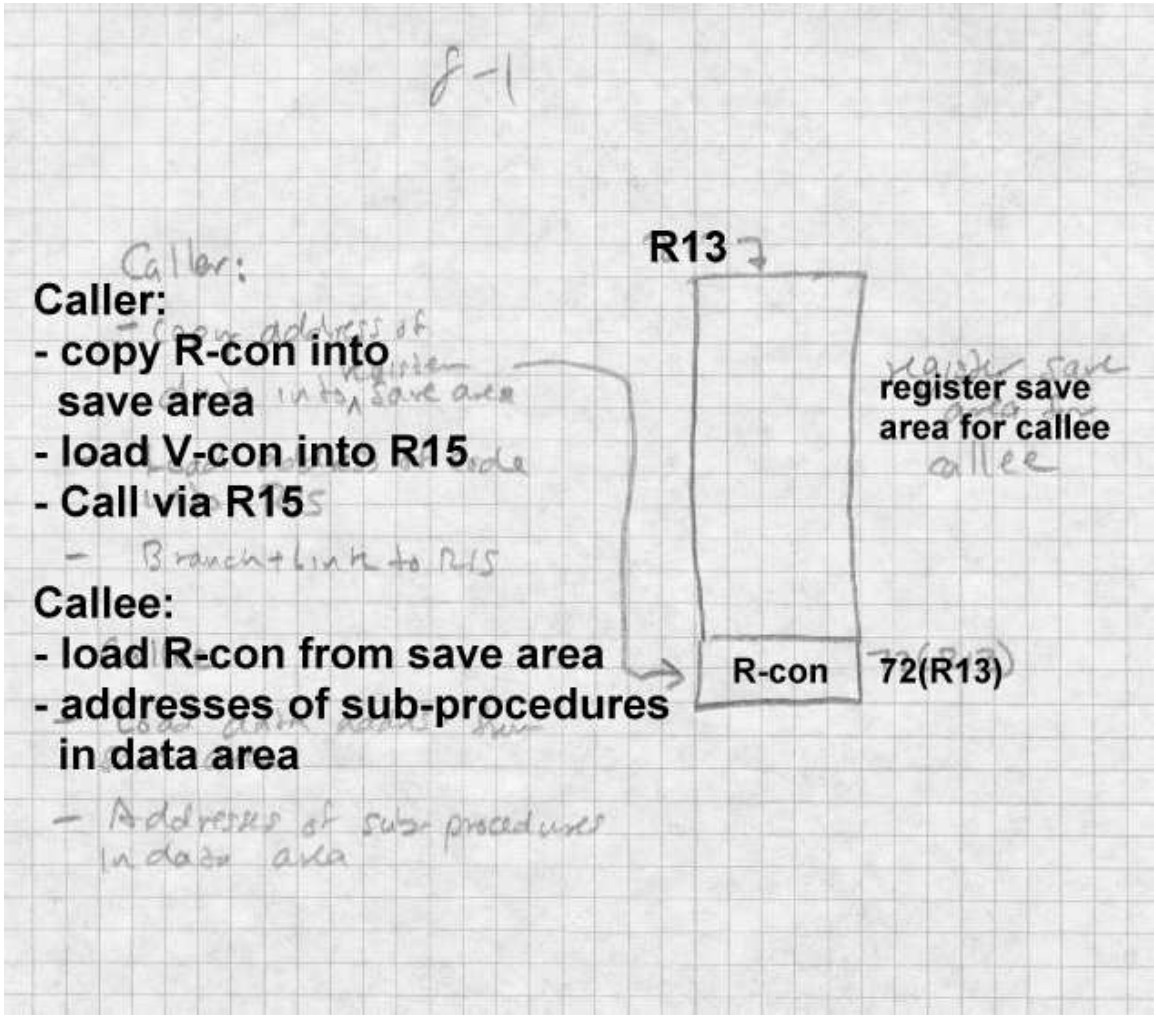
known as the R-con. The standard OS/360 calling sequence requires that the caller provide an 18 word register save area pointed to by register 13. TSS extended the save area to 19 words and required that the caller place callee's R-con into that 19th word before making the call, Figure 1. Each routine had in its data segment the V-cons and R-cons for all of the routines that it called, and stored the appropriate R-con into the outgoing save area before each call. The main routine in a program received a save area from the operating system which provided the initial R-con.

Figure 8-1: TSS style two-address procedure call

TSS style with R-con in the save area

```
Caller:
- copy R-con into
  save area
- load V-con into R15
- Call via R15

Callee:
- load R-con from save area
- addresses of sub-procedures
  in data area
```



This scheme worked, but is poorly suited for modern systems. For one thing, copying the R-cons made the calling sequence bulky. For another, it made procedure pointers two words, which didn't matter in the 1960s but is an issue now since in programs written in C, all pointers have to be the same size. (The C standard doesn't mandate it, but far too much existing C code assumes it to do anything else.)

Per-routine pointer tables

A simple modification used in some Unix systems is to treat the address of a procedure's data as the address of the procedure, and to place a pointer to the procedure's code at that address, Figure 2. To call a procedure, the caller loads the data address into an agreed data pointer register, then loads the code address from the location pointed to by the data pointer into a scratch register and calls the routine. This is easy to implement, and has adequate if not fabulous performance.

Figure 8-2: Code via data pointers

[ROMP style data table with code pointer at the beginning.]

```
Caller:
- Load pointer table
  address into RP
- Load code address from
  0(RP) into RC
- Call via RC

Callee:
- RP points to pointer
  table
- Table has addresses of
  pointer tables for
  sub-procedures
```

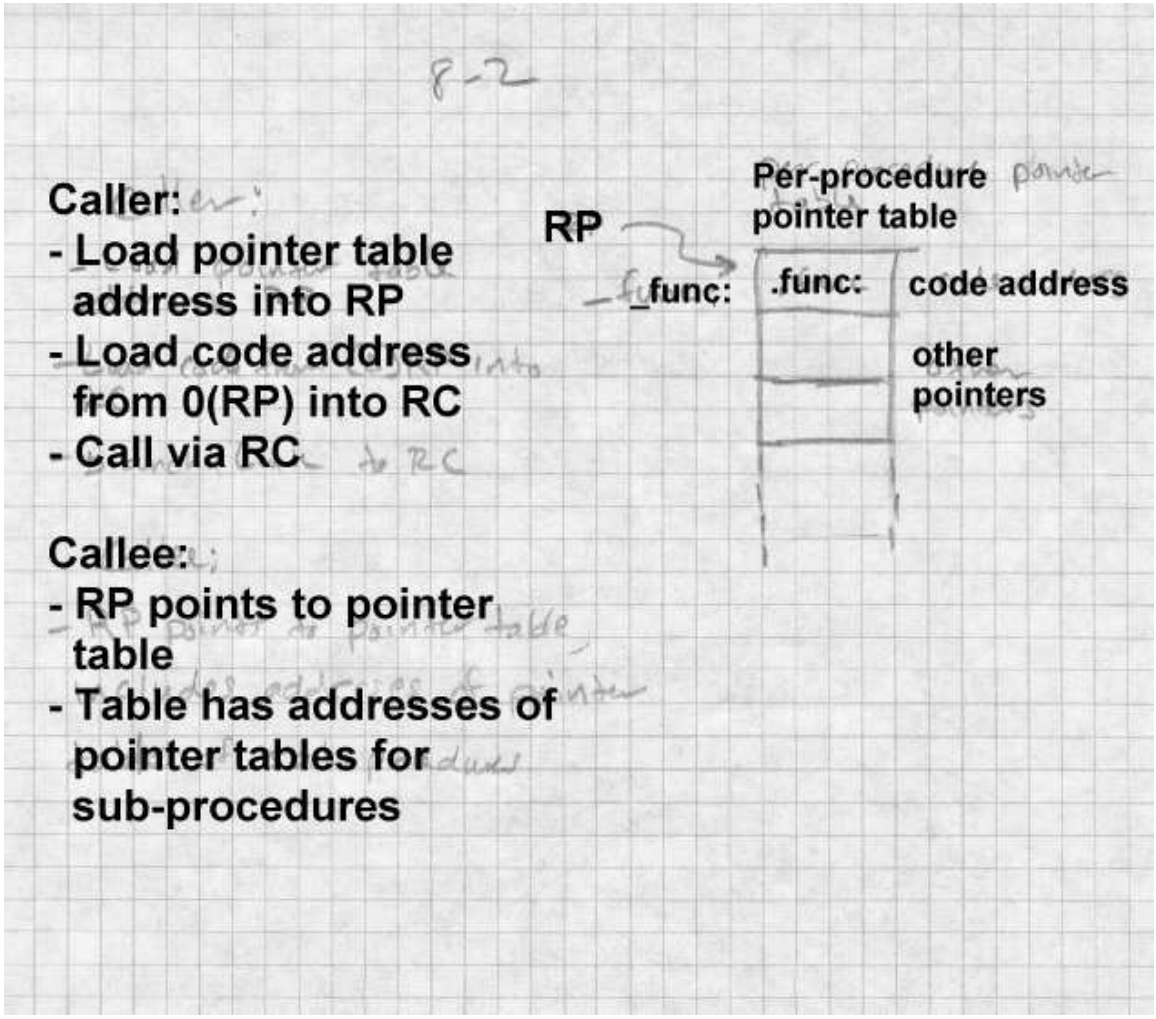


Table of Contents

IBM's AIX uses a more sophisticated version of this scheme. AIX programs group routines into *modules* with a module typically being the object code generated from a single C or C++ source file or a group of related source files. The data segment of each module contains a table of con-

tents (TOC), which contains the combined pointer tables for all of the routines in the module as well as some of the small static data for the routines. Register 2 always contains the address of TOC for the current module, permitting direct access to the static data in the TOC, and indirect addressing of code and data to which the TOC contains pointers. Calls within a single module are a single "call" instruction, since the caller and callee share the same TOC. Inter-module calls have to switch TOCs before the call and switch back afterwards.

Compilers generate all calls as a call instruction, followed by a placeholder no-op instruction, which is correct for intra-module calls. When the linker encounters an inter-module call, it generates a routine called a global linkage or *glink* at the end of the module's text segment. The glink saves the caller's TOC on the stack, loads the callee's TOC and address from pointers in the the caller's TOC, then jumps to the routine. The linker redirects each inter-module call to the glink for the called routine, and patches the following no-op to a load instruction that restores the TOC from the stack. Procedure pointers are pointers to a TOC/code pair, and calls through a pointer use a generic glink routine that uses the TOC and code address the pointer points to.

This scheme makes intra-module calls as fast as possible. Inter-module calls returns are slowed somewhat by the detour through the glink routine, but the slowdown is small compared to some of the alternatives we'll see in a moment.

ELF position independent code

Unix System V Release 4 (SVR4) introduced a PIC scheme similar to the TOC scheme for its ELF shared libraries. The SVR4 scheme is now universally used by systems that use ELF executables, Figure 3. It has the advantage of returning to the normal convention that the address of a procedure is the address of the code for the procedure, regardless of whether one is calling PIC code, found in shared ELF libraries, or non-PIC code, found in regular ELF executables, at the cost of somewhat more per-routine overhead than the TOC scheme's.

Its designers noticed that an ELF executable consists of a group of code pages followed by a group of data pages, and regardless of where in the address space the program is loaded, the offset from the code to the data doesn't change. So if the code can load its own address into a register, the data will be at a known distance from that address, and references to data in the program's own data segment can use efficient based addressing with fixed offsets.

The linker creates a global offset table (GOT) containing pointers to all of the global data that the executable file addresses. (Each shared library has its own GOT, and if the main program were compiled with PIC, which it normally isn't, it would have a GOT as well.) Since the linker creates the GOT, there is only one pointer per ELF executable for each datum regardless of how many routines in the executable refer to it.

If a procedure needs to refer to global or static data, it's up to the procedure itself to load up the address of the GOT. The details vary by architecture, but the 386 code is typical:

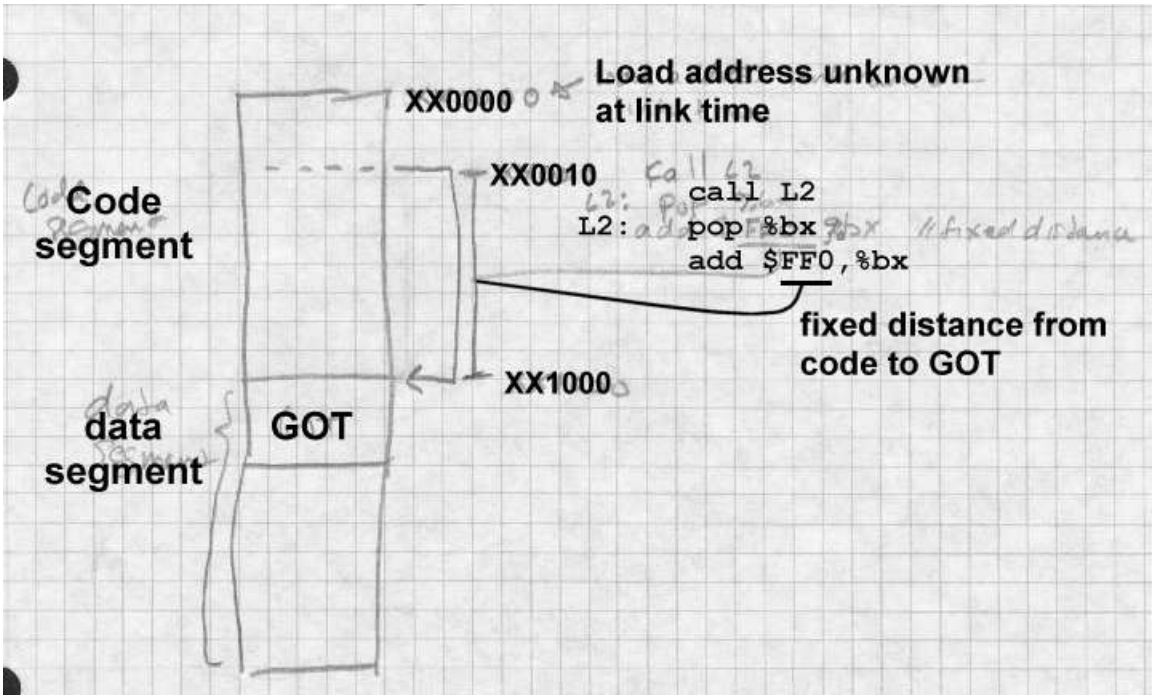
```
  call .L2;; push PC in on the stack
.L2:
  popl %ebx ;; PC into register EBX
  addl $_GLOBAL_OFFSET_TABLE_+[.-.L2],%ebx;; adjust ebx to GOT address
```

It consists of a call instruction to the immediately following location, which has the effect of pushing the PC on the stack but not jumping, then a pop to get the saved PC in a register and an add immediate of the difference between the address the GOT and address the target of the call. In an object file generated by a compiler, there's a special R_386_GOTPC relocation item for the operand of the addl instruction. It tells the linker to substitute in the offset from the current instruction to the base address of the GOT, and also serves as a flag to the linker to build a GOT in the output file. In the output file, there's no relocation needed for the instruction since the distance from the addl to the GOT is fixed.

Figure 8-3: PIC code and data with fixed offsets

picture of code page showing constant offset to data even

though loaded at different addresses in different address spaces.



Once the GOT register is loaded, code can reference local static data using the GOT register as a base register, since the distance from a static datum in the program's data segment to the GOT is fixed at link tine. Addresses of global data aren't bound until the program is loaded (see Chapter 10), so to reference global data, code has to load a pointer to the data from the GOT and then deference the pointer. This extra memory reference makes programs somewhat slower, although it's a cost that most programmers are willing to pay for the convenience of dynamically linked libraries. Speed critical code can use static shared libraries (Chapter 9) or no shared libraries at all.

To support PIC, ELF defines a handful of special relocation types for code that uses the GOT in addition R_386_GOTPC or its equivalent. The exact types are architecture-specific, but the x86 is typical:

- `R_386_GOT32`: The relative location of the slot in the GOT where the linker has placed a pointer to the given symbol. Used for indirectly referenced global data.
- `R_386_GOTOFF`: The distance from the base of the GOT to the given symbol or address. Used to address static data relative to the GOT.
- `R_386_RELATIVE`: Used to mark data addresses in a PIC shared library that need to be relocated at load time.

For example, consider this scrap of C code:

```
static int a;/* static variable */
extern int b;/* global variable */
...
a = 1; b= 2;
```

Variable `a` is allocated in the bss segment of the object file, which means it is at a known fixed distance from the GOT. Object code can reference this variable directly, using the ebx as a base register and a GOT-relative offset:

```
movl $1,a@GOTOFF(%ebx);; R_386_GOTOFF reference to variable "a"
```

Variable `b` is global, and its location may not be known until runtime if it turns out to be in a different ELF library or executable. In this case, the object code references a pointer to `b` which the linker creates in the GOT:

```
movl b@GOT(%ebx),%eax;; R_386_GOT32 ref to address of variable "b"
movl $2,(%eax)
```

Note that the compiler only creates the R_386_GOT32 reference, and it's up to the linker to collect all such references and make slots for them in the GOT.

Finally, ELF shared libraries contain R_386_RELATIVE relocation entries that the runtime loader, part of the dynamic linker we examine in Chapter 10, uses to do loadtime relocaion. Since the text in shared libraries is in-

variably PIC, there's no relocation entries for the code, but data can't be PIC, so there is a relocation entry for every pointer in the data segment. (Actually, you can build a shared library with non-PIC code, in which case there will be relocation entries for the text as well, although almost nobody does that since it makes the text non-sharable.)

PIC costs and benefits

The advantages of PIC are straighforward; it makes it possible to load code without having to do load-time relocation, and to share memory pages of code among processes even though they don't all have the same address space allocated. The possible disadvantages are slowdowns at load time, in procedure calls, in function prolog and epilog, and overall slower code.

At load time, although the code segment of a PIC file needn't be relocated, the data segment does. In large libraries, the TOC or GOT can be very large and it can take a long time to resolve all the entries. This is as much a problem with dynamic linking, which we'll address in Chapter 10, as with PIC. Handling R_386_RELATIVE items or the equivalent to relocate GOT pointers to data in the same executable is fairly fast, but the problem is that many GOT entries point to data in other executables and require a symbol table lookup to resolve.

Calls in ELF executables are usually dynamically linked, even calls within the same library, which adds significant overhead. We revisit this in Chapter 10.

Function prolog and epilogs in ELF files are quite slow. They have to save and restore the GOT register, ebx in the x86, and the dummy call and pop to get the program counter into a register are quite slow. From a performance viewpoint, the TOC approach used in AIX wins here, since each procedure can assume that its TOC register is already set at procedure entry.

Finally, PIC code is bigger and slower than non-PIC. The slowdown varies greatly by architectures. On RISC systems with plenty of registers and no direct addressing, the loss of one register to be the TOC or GOT pointer isn't significant, and lacking direct addressing they need a constant

pool of some sort anyway. The worst case is on the x86. It only has six registers, so losing one of them to be the GOT pointer can make code significantly worse. Since the x86 does have direct addressing, a reference to external data that would be a simple MOV or ADD instruction in non-PIC code turns into a load of the address followed by the MOV or ADD, which both adds an extra memory reference and uses yet another precious register for the temporary pointer.

Particularly on x86 systems, the performance loss in PIC code is significant in speed-critical tasks, enough so that some systems retreat to a sort-of-PIC approach for shared libraries. We'll revisit this issue in the next two chapters.

Bootstrap loading

The discussions of loading up to this point have all presumed that there's already an operating system or at least a program loader resident in the computer to load the program of interest. The chain of programs being loaded by other programs has to start somewhere, so the obvious question is how is the first program loaded into the computer?

In modern computers, the first program the computer runs after a hardware reset invariably is stored in a ROM known as tbe bootstrap ROM. as in "pulling one's self up by the bootstraps." When the CPU is powered on or reset, it sets its registers to a known state. On x86 systems, for example, the reset sequence jumps to the address 16 bytes below the top of the system's address space. The bootstrap ROM occupies the top 64K of the address space and ROM code then starts up the computer. On IBM-compatible x86 systems, the boot ROM code reads the first block of the floppy disk into memory, or if that fails the first block of the first hard disk, into memory location zero and jumps to location zero. The program in block zero in turn loads a slightly larger operating system boot program from a known place on the disk into memory, and jumps to that program which in turn loads in the operating system and starts it. (There can be even more steps, e.g., a boot manager that decides from which disk partition to read the operating system boot program, but the sequence of increasingly capable loaders remains.)

Why not just load the operating system directly? Because you can't fit an operating system loader into 512 bytes. The first level loader typically is only able to load a single-segment program from a file with a fixed name in the top-level directory of the boot disk. The operating system loader contains more sophisticated code that can read and interpret a configuration file, uncompress a compressed operating system executable, address large amounts of memory (on an x86 the loader usually runs in real mode which means that it's tricky to address more than 1MB of memory.) The full operating system can turn on the virtual memory system, loads the drivers it needs, and then proceed to run user-level programs.

Many Unix systems use a similar bootstrap process to get user-mode programs running. The kernel creates a process, then stuffs a tiny little program, only a few dozen bytes long, into that process. The tiny program executes a system call that runs /etc/init, the user mode initialization program that in turn runs configuration files and starts the daemons and login programs that a running system needs.

None of this matters much to the application level programmer, but it becomes more interesting if you want to write programs that run on the bare hardware of the machine, since then you need to arrange to intercept the bootstrap sequence somewhere and run your program rather than the usual operating system. Some systems make this quite easy (just stick the name of your program in AUTOEXEC.BAT and reboot Windows 95, for example), others make it nearly impossible. It also presents opportunities for customized systems. For example, a single-application system could be built over a Unix kernel by naming the application /etc/init.

Tree structured overlays

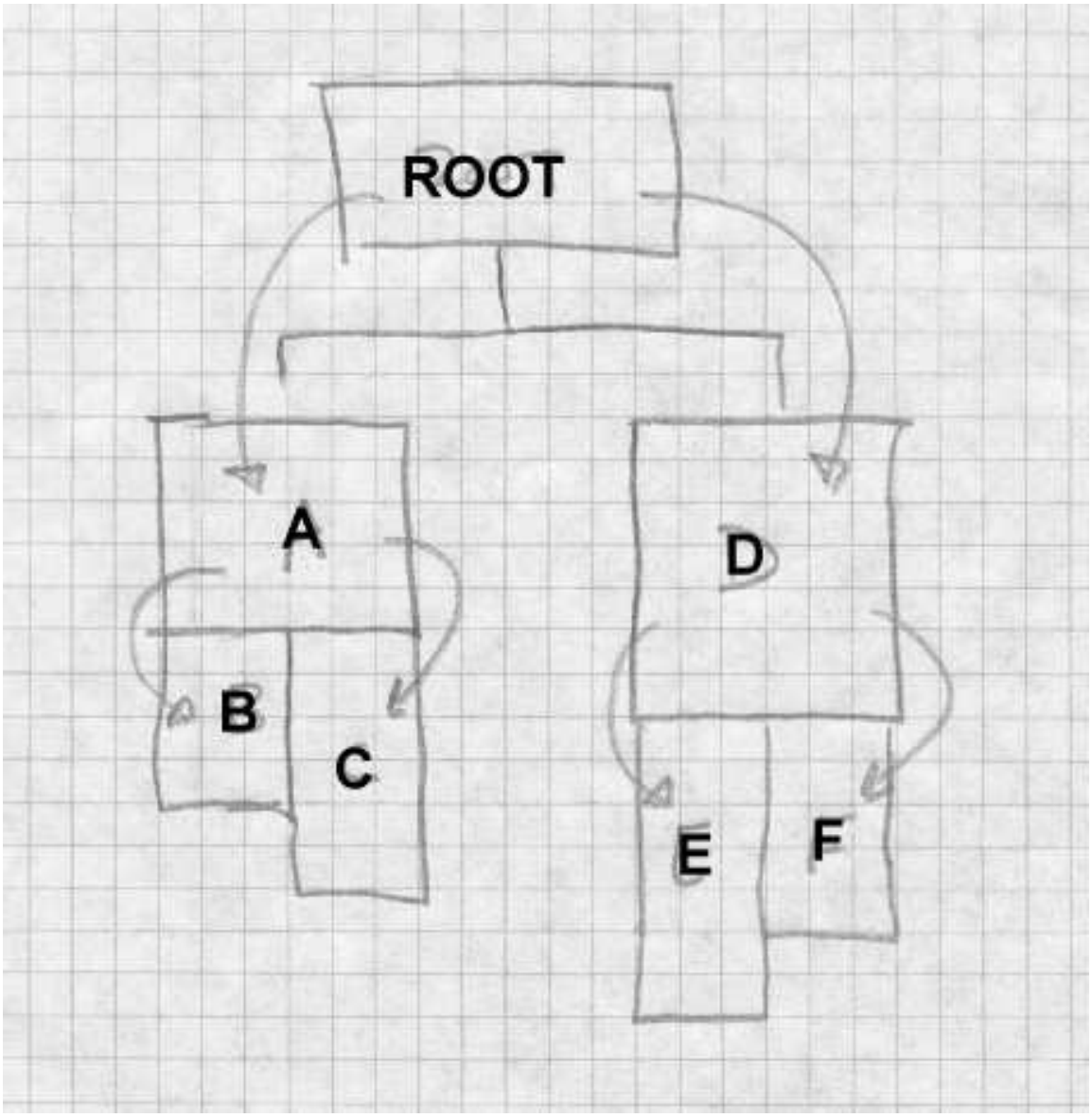
We close this chapter with a description of tree-structured overlays, a widely used scheme in the days before virtual memory to fit programs into memories smaller than the programs. Overlays are another technique that dates back to before 1960, and are still in use in some memory-constrained environments. Several MS-DOS linkers in the 1980 supported them in a form nearly identical to that used 25 years earlier on mainframe computers. Although overlays are now little used on conventional architectures, the techniques that linkers use to create and manage overlays remain inter-

esting. Also, the inter-segment call tricks developed for overlays point the way to dynamic linking. In environments like DSPs with constrained program address spaces, overlay techniques can be a good way to squeeze programs in, especially since overlay managers tend to be small. The OS/360 overlay manager is only about 500 bytes, and I once wrote one for a graphics processor with a 512 word address space that used only a dozen words or so.

Overlaid programs divide the code into a tree of segments, such as the one in Figure 4.

Figure 8-4: A typical overlay tree

ROOT calls A and D. A calls B and C, D calls E and F.



The programmer manually assigns object files or individual object code segments to overlay segments. Sibling segments in the overlay tree share

the same memory. In the example, segments A and D share the same memory, B and C share the same memory, and E and F share the same memory. The sequence of segments that lead to a specific segment is called a path, so the path for E includes the root, D, and E.

When the program starts, the system loads the root segment which contains the entry point of the program. Each time a routine makes a "downward" inter-segment call, the overlay manager ensures that the path to the call target is loaded. For example, if the root calls a routine in segment A, the overlay manager loads section A if it's not already loaded. If a routine in A calls a routine in B the manager has to ensure that B is loaded, and if a routine in the root calls a routine in B, the manager ensures that both A and B are loaded. Upwards calls don't require any linker help, since the entire path from the root is already loaded.

Calls across the tree are known as *exclusive* calls and are usually considered to be an error since it's not possible to return. Overlay linkers let the programmer force exclusive calls for situations where the called routine is known not to return.

Defining overlays

Overlay linkers created overlaid executables from ordinary input object files. The objects don't contain any overlay instructions, Intstead, the programmer specifies the overlay structure with a command language that the linker reads and interprets. Figure 5 shows the same overlay structure as before, with the names of the routines loaded into each segment.

Figure 8-5: A typical overlay tree

ROOT contains rob and rick
calls A with aaron and andy and D.
A calls B (bill and betty) and C (chris), D (dick, dot) calls E (edgar) and F (fran).

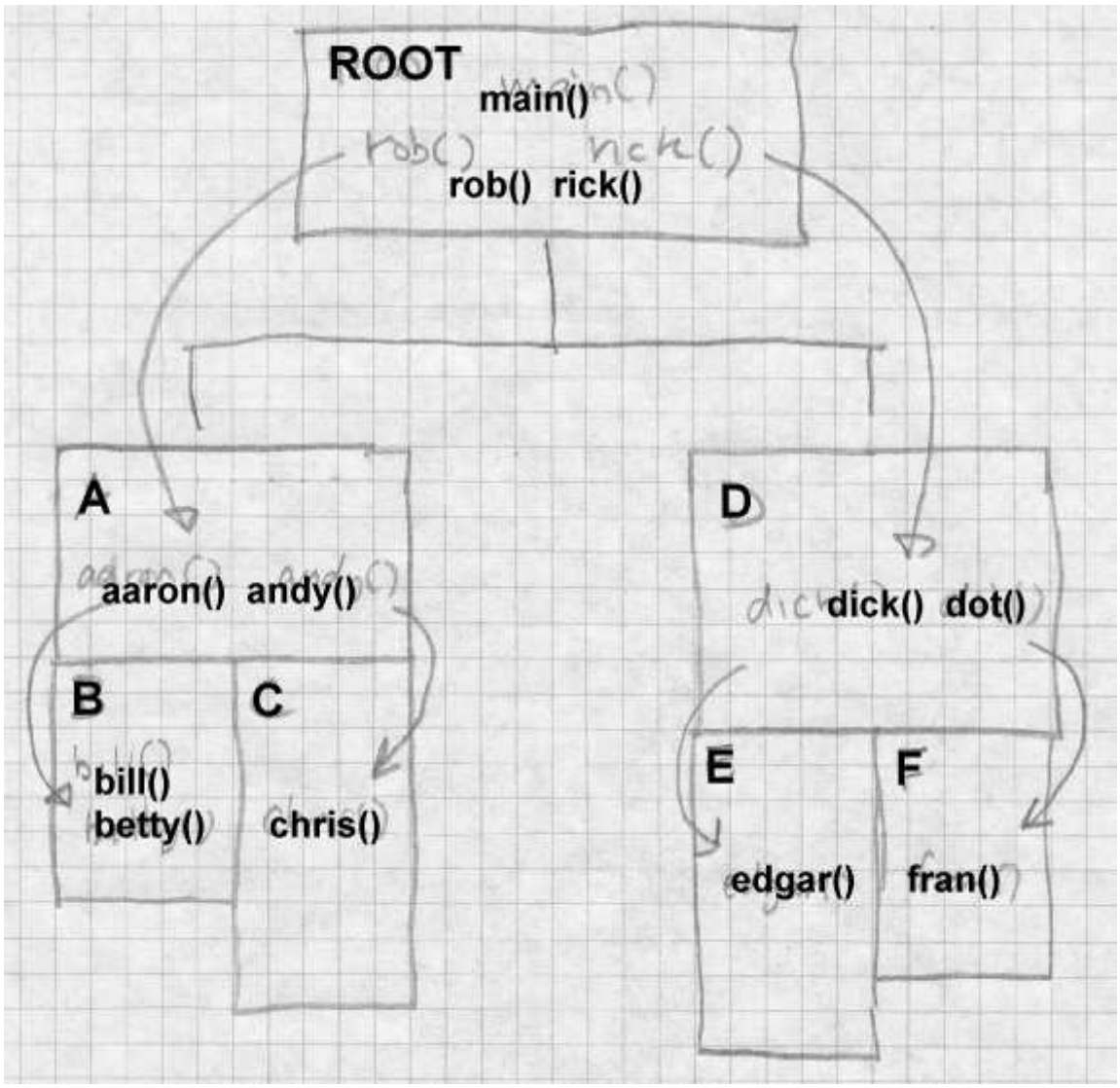


Figure 6 shows the linker commands that one might give to the IBM 360 linker to create this structure. Spacing doesn't matter, so we've indented the commands to show the tree structure. OVERLAY commands define the beginning of each segment; commands with the same overlay name

define segments that overlay each other. Hence the first OVERLAY AD defines segment A, and the second defines segmnt D. Overlay segments are defined in a depth first left to right tree walk. INCLUDE commands name logical files for the linker to read.

Figure 8-6: Linker commands

```
INCLUDE ROB
INCLUDE RICK
OVERLAY AD
  INCLUDE AARON, ANDY
  OVERLAY BC
    INCLUDE BILL, BETTY
  OVERLAY BC
    INCLUDE CHRIS
OVERLAY AD
  INCLUDE DICK, DOT
  OVERLAY EF
    INCLUDE EDGAR
  OVERLAY EF
    INCLUDE FRAN
```

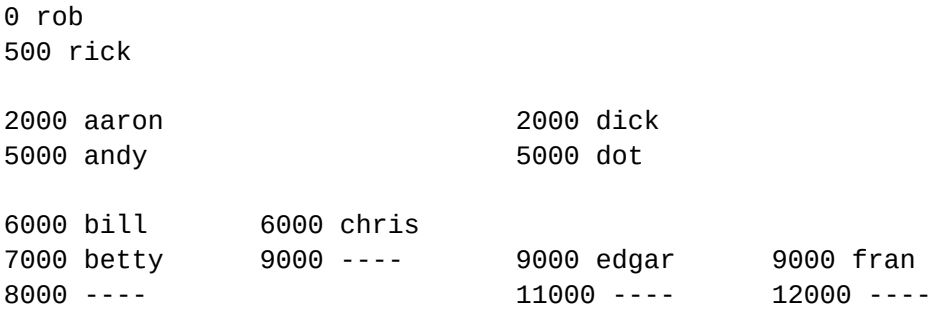
It's up to the programmer to lay out overlays to be space effiecent. The storage allocated for each segment is the maximum length of any of the segments that occupy the same space. For example, assume that the file lengths in decimal are as follows.

Name	Size
rob	500
rick	1500
aaron	3000
andy	1000
bill	1000
betty	1000

chris 3000
dick 3000
dot 4000
edgar 2000
fran 3000

The storage allocation, looks like Figure 7. Each segment starts immediately after the preceding segment in the path, and the total program size is the length of the longest path. This program is fairly well balanced, with the longest path being 11500 and the shortest being 8000. Juggling the overlay structure to find one that is as compact as possible while still being valid (no exclusive calls) and reasonably efficient is a black art requiring considerable trial and error. Since the overlays are defined entirely in the linker, each trial requires a relink but no recompilation.

Figure 8-7: Overlay storage layout



Implementation of overlays

The implementation of overlays is surprisingly simple. Once the linker determines the layout of the segments, relocates the code in each segment appropriately based on the memory location of the segment. The linker needs to create a segment table which goes in the root segment, and, in each segment, glue code for each routine that is the target of a downward call from that segment.

The segment table, Figure 8, lists each segment, a flag to note if the segment is loaded, the segment's path. and information needed to load the segment from disk.

Figure 8-8: Idealized segment table

```
struct segtab {
  struct segtab *path;// preceding segment in path
  boolean ispresent;// true if this segment is loaded
  int memoffset; // relative load address
  int diskoffset;  // location in executable
  int size;   // segment size
} segtab[];
```

The linker interposes the glue code in front of each downward call so the overlay manager can ensure that the required segment(s) are loaded. Segments can use glue code in higher level but not lower level routines. For example, if routines in the root call aaron, dick, and betty, the root needs glue code for each of those three symbols. If segment A contains calls to bill, betty, and chris, A needs glue code for bill and chris, but can use the glue for betty already present in the root. All downward calls (which are to global symbols) are resolved to glue code, Figure 9, rather than to the actual routine. The glue code has to save any registers it changes, since it has to be transparent to the calling and called routine, then jump into the overlay manager, providing the address of the real routine and an indication of which segment that address is in. Here we use a pointer, but an index into the segtab array would work as well.

Figure 8-9: Idealized glue code for x86

```
glue'betty: call load_overlay
  .long betty // address of real routine
  .long segtab+N // address of segment B's segtab
```

At runtime, the system loads in the root segment and starts it. At each downward call, the glue code calls the overlay manager. The manager checks the target segment's status. If the segment is present, the manager just jumps to the real routine. If the segment is not present, the manager loads the target segment and any unloaded preceding segments in the path, marks any conflicting segments as not present, marks the newly loaded segments as present, and jumps.

Overlay fine points

As always, details make elegant tree structured overlays messier than they might be.

Data

We've been talking about structuring code overlays, without any consideration of where the data goes. Individual routines may have private data loaded into the segments with the routines, but any data that has to be remembered from one call to the next needs to be promoted high enough in the tree that it won't get unloaded and reloaded, which would lose any changes made. In practice, it means that most global data usually ends up in the root. When Fortran programs are overlaid, overlay linkers can position common blocks appropriately to be used as communication areas. For example, if dick calls edgar and fran, and the latter two both refer to a common block, that block has to reside in segment D to be a communication area.

Duplicated code

Frequently the overall structure of an overlaid program can be improved by duplicating code. In our example, imagine that chris and edgar both call a routine called greg which is 500 bytes long. A single copy of greg would have to go in the root, increasing the total loaded size of the program, since placing it anywhere else in the tree would require a forbidden exclusive call from either chris or edgar. On the other hand, if both segments C and E include copies of greg, the overall loaded size of the program doesn't increase, since the end of segment C would grow from 9000

to 9500 and of E from 11000 to 11500, both still smaller than the 12000 bytes that F requires.

Multiple regions

Frequently, a program's calling structure doesn't map very well to a single tree. Overlay systems handle multiple code regions, with a separate overlay tree in each region. Calls between regions always go through glue code. The IBM linker supports up to four regions, although in my experience I never found a use for more than two.

Overlay summary

Even though overlays have been rendered largely obsolete by virtual memory, they remain of historical interest because were the first significant use of link-time code generation and modification. They require a great deal of manual programmer work to design and specify the overlay structure, generally with a lot of trial and error ‘‘digital origami’’, but they were a very effective way to squeeze a large program into limited memory.

Overlays originated the important technique of "wrapping" call instructions in the linker to turn a simple procedure call into one that did more work, in this case, loading the required overlay. Linkers have used wrapping in a variety of ways. The most important is dynamic linking, which we cover in chapter 10, to link to a called routine in a library that may not have been loaded yet. Wrapping is also useful for testing and debugging, to insert checking or validation code in front of a suspect routine without changing or recompiling the source file.

Exercises

Compile some small C routines with PIC and non-PIC code. How much slower is the PIC code than non-PIC? Is it enough slower to be worth having non-PIC versions of libraries for programmers in a hurry?

In the overlay example, assume that dick and dot each call both edgar and fran, but dick and dot don't call each other. Restructure the overlay so that dick and dot share the same space, and adjust the structure so that the call tree still works. How much space does the overlaid program take now?

In the overlay segment table, there's no explicit marking of conflicting segments. When the overlay manager loads a segment and the segment's path, how does the manager determine what segments to mark as not present?

In an overlaid program with no exclusive calls, is it possible that a series of calls could end up jumping to unloaded code anyway? In the example above, what happens if rob calls bill, which calls aaron, which calls chris, then the routines all return? How hard would it be for the linker or overlay manager to detect or prevent that problem?

Project

Project 8-1: Add a feature to the linker to "wrap" routines. Create a linker switch

```
-w name
```

that wraps the given routine. Change all references in the program to the named routine to be references to `wrap_name`. (Be sure not to miss internal references within the segment in which the name is defined.) Change the name of the routine to `real_name`. This lets the programmer write a wrapper routine called `wrap_name` that can call the original routine as `real_name`.

Project 8-2: Starting the linker skeleton from chapter 3, write a tool that modifies an object file to wrap a name. That is, references to `name` turn into external references to `wrap_name`, and the existing routine is renamed `real_name`. Why would one want to use such a program rather than building the feature into the linker. (Hint: consider the case where you're not the author or maintainer of the linker.)

Project 8-3: Add support to the linker to produce executables with position-independent code We add a few new four-byte relocation types:

```
loc seg ref GA4
loc seg ref GP4
loc seg ref GR4
loc seg ref ER4
```

The types are:

- `GA4`: (GOT address) At location loc, store the distance to the GOT.
- `GP4`: (GOT pointer) Put a pointer to symbol ref in the GOT, and at location loc, store the GOT-relative offset of that pointer.
- `GR4`: (GOT relative) Location loc contains an address in segment ref. Replace that with the offset from the beginning of the GOT to that address.
- `ER4`: (Executable relative) Location loc contains an address relative to the beginning of the executable. The ref field is ignored.

In your linker's first pass, look for GP4 relocation entries, build a GOT segment with all the required pointers, and allocate the GOT segment just before the data and BSS segments. In the second pass, handle the GA4, GP4, and GR4 entries. In the output file, create ER4 relocation entries for any data that would have to be relocated if the output file were loaded at other than its nominal address. This would include anything marked by an A4 or AS4 relocation entry in the input. (Hint: Don't forget the GOT.)