CSc 453
Linking and Loading

Saumya Debray
The University of Arizona
Tucson

Tasks in Executing a Program

1. Compilation and assembly.
   - Translate source program to machine language.

     The result may still not be suitable for execution, because of unresolved references to external and library routines.
2. Linking.
   - Bring together the binaries of separately compiled modules.
   - Search libraries and resolve external references.
3. Loading.
   - Bring an object program into memory for execution.

     Allocate memory, initialize environment, maybe fix up addresses.

Contents of an Object File

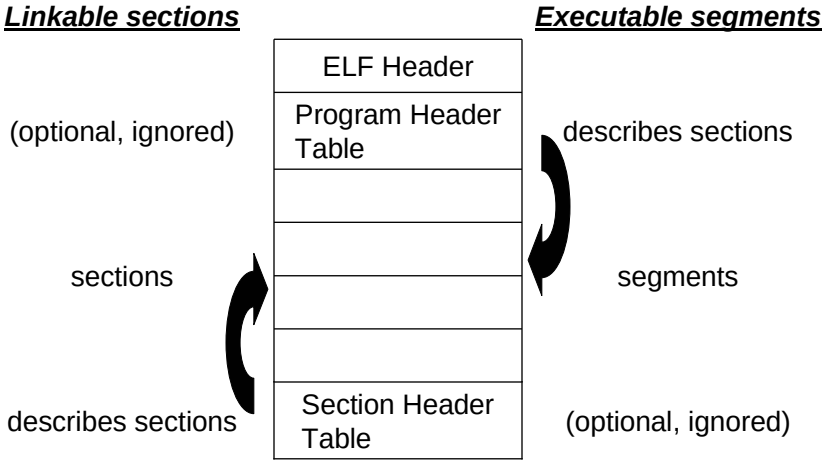
- Header information

  Overall information about the file and its contents.
- Object code and data
- Relocations (may be omitted in executables)

  Information to help fix up the object code during linking.
- Symbol table (optional)

  Information about symbols defined in this module and symbols to be imported from other modules.
- Debugging information (optional)

Example: ELF Files (x86/Linux)



ELF Files: cont'd

ELF Header structure

16 bytes	ELF file identifying information (magic no., addr size, byte order)
2 bytes	object file type (relocatable, executable, shared object, etc.)
2 bytes	machine info
4 bytes	object file version
4 bytes	entry point (address where execution begins)
4 bytes	offset of program header table
4 bytes	offset of section header table
4 bytes	processor-specific flags
2 bytes	ELF header size (in bytes)
2 bytes	size of each entry in program header table
2 bytes	no. of entries in program header table
2 bytes	size of section header table (in bytes)
2 bytes	no. of entries in section header table
2 bytes	index of section name string table in in the section header table

Elf Files: cont'd

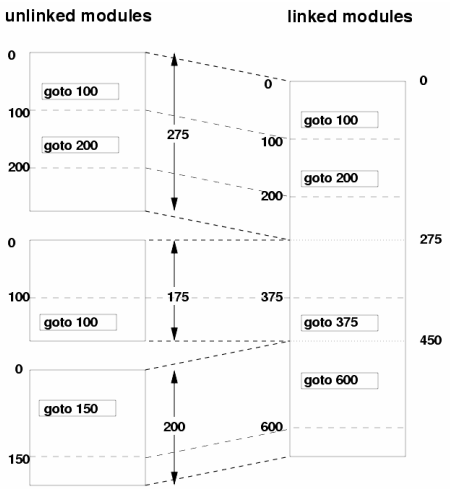
Section Header structure

4 bytes	section name (".text", ".data", ".rodata", etc.), given as an index into the section header string table section
4 bytes	section type (specifies section contents and semantics)
4 bytes	assorted section flags
4 bytes	the address within a process where the section should begin (if the section actually appears in the executing process)
4 bytes	byte offset from the beginning of the file to the first byte of the section
4 bytes	section size (in bytes)
4 bytes	index link (special information, depending on section type)
4 bytes	special information, depending on section type
4 bytes	address alignment constraints for the section, if any
4 bytes	size of each entry in the section, for sections with fixed-size entries (e.g., symbol table)

Linker Functions 1: Fixing Addresses

- Addresses in an object file are usually relative to the start of the code or data segment in that file.
- When different object files are combined:
  - The same kind of segments (text, data, read-only data, etc.) from the different object files get merged.
  - Addresses have to be “fixed up” to account for this merging.
  - The fixing up is done by the linker, using information embedded in the executable for this purpose (“relocations”).

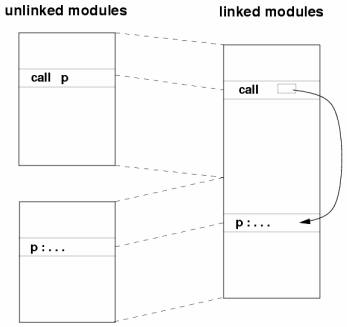
Relocation: Example



Linker Function 2: Symbol Resolution

Suppose:

- module *B* defines a symbol x;
- module *A* refers to x.



The linker must:

1. determine the location of x in the object module obtained from merging *A* and *B*; and
2. modify references to x (in both *A* and *B*) to refer to this location.

Information for Symbol Resolution

Each linkable module contains a *symbol table*, whose contents include:

- Global symbols defined (maybe referenced) in the module.
- Global symbols referenced but not defined in the module (these are generally called *externals*).
- Segment names (e.g., *text*, *data*, *rodata*).

  These are usually considered to be global symbols defined to be at the beginning of the segment.
- Non-global symbols and line number information (optional), for debuggers.

Actions Performed by a Linker

Usually, linkers make two passes:

- *Pass 1*:
  - Collect information about each of the object modules being linked.
- *Pass 2*:
  - Construct the output, carrying out address relocation and symbol resolution using the information collected in Pass 1.

Linker Actions: Pass 1

1. Construct a table of all the object modules and their lengths.
2. Based on this table, assign a *load address* to each module.
3. For each module:
   - Read in its symbol table into a *global symbol table* in the linker.
   - Determine the address of each symbol defined in the module in the output:

     Use the symbol value together with the module load address.

Linker Actions: Pass 2

Copy the object modules in the order of their load addresses:

1. *Address relocation*:
   - find each instruction that contains a memory address;
   - to each such address, add a *relocation constant* equal to the load address for its module.
2. *External symbol resolution*:
   - For each instruction that references an external object, insert the actual address for that object.

Relocation Example: ELF (x86/Linux)

- ELF relocation entries take one of two forms:

```
typedef struct {
  Addr32  offset;
  Word32  info;
}
```

```
typedef struct {
  Addr32  offset;
  Word32  info;
  SignedWord32 addend;
}
```

  - **`offset`** : specifies the location where to apply the relocation action.
  - **`info`** : gives the symbol table entry w.r.t. which the relocation should be made, and the type of relocation to apply.
    E.g.: for a **call** instruction, the **`info`** field gives the index of the callee.
  - **`addend`** : a value to be added explicitly during relocation.
- Depending on the architecture, one form or the other may be more convenient.

Loading

Programs are usually loaded at a fixed address in a fresh address space (so can be linked for that address).

In such systems, loading involves the following actions:

1. determine how much address space is needed from the object file header;
2. allocate that address space;
3. read the program into the segments in the address space;
4. zero out any uninitialized data (".bss" segment) if not done automatically by the virtual memory system.
5. create a stack segment;
6. set up any runtime information, e.g., program arguments or environment variables.
7. start the program executing.

Position-Independent Code (PIC)

- If the load address for a program is not fixed (e.g., shared libraries), we use *position independent code*.
- Basic idea: separate code from data; generate code that doesn't depend on where it is loaded.
- PC-relative addressing can give position-independent code references.

  This may not be enough, e.g.: data references, instruction peculiarities (e.g., **call** *instruction in Intel x86) may not permit the use of PC-relative addressing.*

PIC (cont'd): ELF Files

- ELF executable file characteristics:
  - data pages follow code pages;
  - the offset from the code to the data does not depend on where the program is loaded.
- The linker creates a *global offset table* (GOT) that contains offsets to all global data used.
- If a program can load its own address into a register, it can then use a fixed offset to access the GOT, and thence the data.

PIC code on ELF: cont'd

Code to figure out its own address (x86):

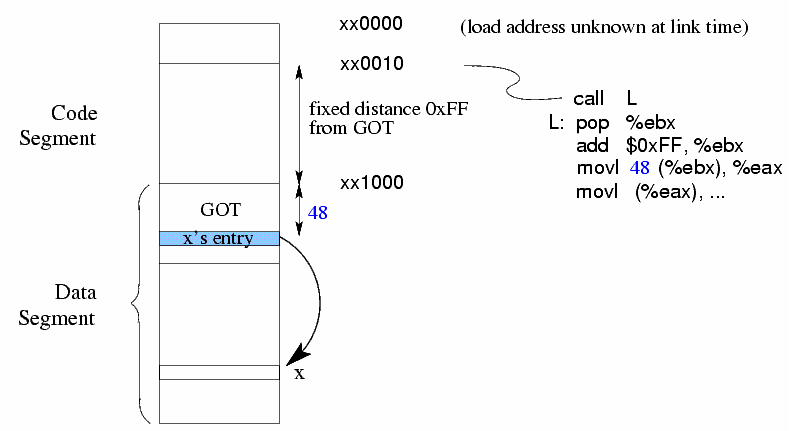
```
      call  L     /* push address of next instruction on stack */
L:    pop %ebx    /* pop address of this instruction into %ebx */
```

Accessing a global variable x in PIC:

1. GOT has an entry, say at position k, for x. The dynamic linker fills in the address of x into this entry at load time.
2. Compute "my address" into a register, say %ebx (above);
3. %ebx += offset_to_GOT; /* fixed for a given program */
4. %eax = contents of location k(%ebx) /* %eax = addr. of x */
5. access memory location pointed at by %eax;

PIC on ELF: Example

(Based on *Linkers and Loaders*, by J. R. Levine (Morgan Kaufman, 2000))



PIC: Advantages and Disadvantages

Advantages:

- Code does not have to be relocated when loaded.
  (However, data still need to be relocated.)
- Different processes can share the memory pages of code, even if they don't have the same address space allocated.

Disadvantages:

- GOT needs to be relocated at load time.
  In big libraries, GOT can be very large, so this may be slow.
- PIC code is bigger and slower than non-PIC code.
  The slowdown is architecture dependent (in an architecture with few registers, using one to hold GOT address can affect code quality significantly.)

Shared Libraries

- Have a single copy of the library that is used by all running programs.
- Saves (disk and memory) space by avoiding replication of library code.
- Virtual memory management in the OS allows different processes to share "read-only" pages, e.g., text and read-only data.
  - *This lets us get by with a single physical-memory copy of shared library code.*

Shared Libraries: cont'd

- At link time, the linker:
  - Searches a (specified) set of libraries, in some fixed order, to find modules that resolve any undefined external symbols.
  - puts a list of libraries containing such modules into the executable.
- At load time, the startup code:
  - finds these libraries;
  - maps them into the program's address space;
  - carries out library-specific initialization.
- Startup code may be in the OS, in the executable, or in a special dynamic linker.

Statically Linked Shared Libraries

- Program and data are bound to executables at link time.
- Each library is pre-allocated an appropriate amount of address space.
- The system has a master table of shared-library address space:
  - libraries start somewhere far away from application code, e.g., at 0x60000000 on Linux;
  - read-only portions of the libraries can be shared between processes.

Dynamic Linking

- Defers much of the linking process until the program starts running.
- Easier to create, update than statically linked shared libraries.
- Has higher runtime performance cost than statically linked libraries:
  - Much of the linking process has to be redone each time a program runs.
  - Every dynamically linked symbol has to be looked up in the symbol table and resolved at runtime.

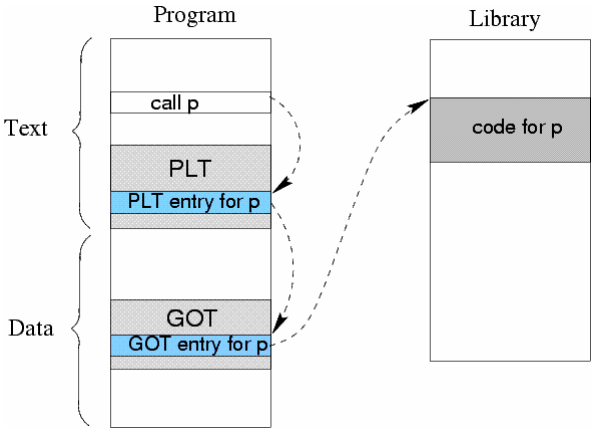
Dynamic Linking: Basic Mechanism

- A reference to a dynamically linked procedure p is mapped to code that invokes a *handler*.
- At runtime, when p is called, the handler gets executed:
  - The handler checks to see whether p has been loaded already (due to some other reference);
  - if so, the current reference is linked in, and execution continues normally.
  - otherwise, the code for p is loaded and linked in.

Dynamic Linking: ELF Files

- ELF shared libraries use PIC (position independent code), so text sections do not need relocation.
- Data references use a GOT:
  - each global symbol has a relocatable pointer to it in the GOT;
  - the dynamic linker relocates these pointers.
- We still need to invoke the dynamic linker on the first reference to a dynamically linked procedure.
  - Done using a *procedure linkage table* (PLT);
  - PLT adds a level of indirection for function calls (analogous to the GOT for data references).

ELF Dynamic Linking: PLT and GOT

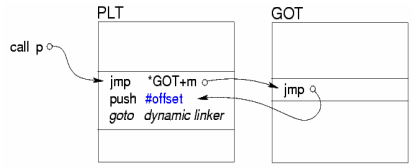


ELF Dynamic Linking: Lazy Linkage

- Initially, GOT entry points to PLT code that invokes the dynamic linker.

  offset identifies both the symbol being resolved and the corresponding GOT entry.
- The dynamic linker looks up the symbol value and updates the GOT entry.
- Subsequent calls bypass dynamic linker, go directly to callee.
- This reduces program startup time. Also, routines that are never called are not resolved.

Before:



After: