Systems Programming

Chapter 3

Linkers and Loaders

Outline

- Design and implementation of linkers and loaders
  - fundamental function:
    - loading an object program into memory for execution
    - e.g.,an absolute loader for SIC machine
  - relocation and linking
    - object program representation and machine dependence
  - linking loader
  - machine independent loader features
  - linkage editors – perform linking before loading
  - dynamic linking – delaying linking until execution time

Introduction

Object program:

- contains translated instructions and data from the source program,
- specifies addresses in memory where these items are to be loaded.

Loading: brings the object program into memory for execution

Relocation: modifies the object program so that it can be loaded at an address different from the location originally specified

Linking: combines two or more separate object programs and supplies the information needed to allow references between them.

Loaders

- A loader is a system program that performs the loading function.
  - many also support relocation & linking
  - others have a separate linker and loader
- A single loader and linker exist on a system since compilers/assemblers produce object code in the same format.

Basic Loader Functions

- bringing an object program into memory
- starting its execution

Design of an Absolute Loader

- Refer to Section 2.1&2.1.1 and Figure 3.1
- Its operation is very simple
  - no linking or relocation
- Single pass operation
  - check **H** record to verify that correct program has been presented for loading
  - read each **T** record, and move object code into the indicated address in memory
  - at **E** record, jump to the specified address to begin execution of the loaded program.

- Figure 3.2

- Each byte of assembled code is given using Hex representation in character form
- As the instruction is loaded for execution, the operation code must be stored in a single byte w/Hex value.

- May prefer to store object code in binary form for obtaining more efficiency!

A Simple Bootstrap Loader

- Automatically executed when the computer is first turned on
- Loads the first program to be run: usually the O/S.
- See Figure 3.3 – A bootstrap loader for SIC/XE
  - itself begins at address 0 in memory
  - loads the O/S starting at address 80
  - Each byte of object code to be loaded is represented on device F1 as two Hex digits
  - No H or E records, no control information (eoln)
  - After all code is loaded, bootstrap jumps to address 80.
  - Subroutine GETC reads one char from device F1 and converts it from ASCII char code to the value of the hex digit that it represents

MACHINE-DEPENDENT LOADER FEATURES

- Disadvantages of absolute loader
  - actual load address must be specified
    - OK for SIC
    - problematic for an advanced machine where several independent programs run together and share memory
      - relocation is needed for efficient execution
  - difficult to use subroutine libraries (scientific and mathematical) efficiently
    - important to be able to select and load exactly those routines that are needed

- A more complex loader
  - suitable for SIC/XE and is typical of those found on most modern computers
  - supports relocation and linking
- Section 3.2.1 – hardware dependencies
- Section 3.2.2 – program linking from the loader's point of view
  - not as machine dependent as relocation
- Section 3.3 – data structures and processing logic

Relocation

Relocating loaders or relative loaders:

loaders that allow for program relocation.

Two methods for specifying relocation as part of the object program:

1. **A Modification record (Section 2.3.5) is used** to describe each part of the object code that must be changed when the program is relocated
   - Figure 3.4 (same as 2.6) – XE program -> Figure 3.5
   - Most instructions in this XE program use relative addressing, except lines 15, 35, and 65.
   - **M00000705+COPY**

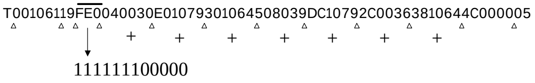
Second method

2. Modification record scheme is not well suited for use with a machine like SIC
   - SIC does not use relative addresses

     almost all instructions (except RSUB) must be modified when the program is relocated.

   No Modification records

   A **relocation bit** associated with each word of object code and they form a **bit mask**


T00106119FE0040030E01079301064508039DC10792C00363810644C000005
+ + + + + + +
111111100000

Third method

3. Hardware relocation capability is provided by some computers

   - eliminates some of the need for the loader to perform relocation

   - They keep all memory references to be relative to the user's assigned area of memory

   - Conversion takes place during execution.

Program Linking

- Section 2.3.5, Figure 2.15
  - a program w/3 control sections
  - They may be separately or together assembled
    - result is separate segments of object code after assembly
    - Figure 3.8
    - set of references to external symbols:
      - instruction operands (REF1 – REF3)
      - values of data words (REF4 – REF8)
        - » We will examine the differences in the way these identical expressions are handled within the three programs.

- REF1 –
  - PROGA – simply a reference to a label within the program: PC-relative instr.
  - PROGB/PROGC – refers to an external symbol: extended-format instr.
    - has a Modificiation record instructing the loader to add the value of LISTA to this address during linking
- REF2 –
  - same as REF1 , except value of constant

- REF3 – immediate operand whose value is to be the difference between ENDA and LISTA
  - PROGA – knows all info
  - PROGB/C – values of labels are unknown
    - must be assembled as an external reference w/two modification records

General approach

- to evaluate as much of the expression as it can and to pass the remaining terms to the loader via Modification records

- See REF4
  - PROGA – evaluate all except LISTC
  - PROGB/PROGC – no terms can be evaluated

- Fig 3.10 – three programs after loading and linking
  - REF4 – 8 has resulted in the same value
  - Figure 3.10.b – actual computation of REF4 in PROGA
- For the references that are instruction operands, the calculated values after loading do not always appear to be equal
  - because, there is an additional address calculation step involved for PC-relative instructions (BUT target addresses are the same)
    - REF1 (PROGA: 01D PC-relative, PROGB: 4040 extended format)

Algorithm and Data Structures for a Linking (Relocating) Loader

- More complicated than the absolute loader
- We use Modification records for relocation
  - so that linking and relocating functions are performed using the same mechanism
- Input: a set of object programs (control sections) that are to be linked together
  - A CS may make an external ref. to a symbol whose def does not appear until later in this input stream until later CS is read.
  - Two passes over input:
    - Pass 1: assigns addresses to all external symbols
    - Pass 2: performs the actual loading, relocation, and linking.

Main Data Structures

- **External Symbol Table** (ESTAB)
  - ~Symbol Table
  - stores the name and address of each external symbol in the set of control sections being loaded.
  - also often indicates in which CS the symbol is defined.
  - A hashed organization is typically used.
- **Program Load Address** (PROGADDR)
  - beginning address inmemory where the linked program is to be loaded.
  - supplied by the O/S
- **Control Section Address** (CSADDR)
  - starting address asigned to the CS currently being scanned by the loader
  - Its value is added to all relative addresses within the control section to convert them to actual addresses

The Algorithm

- Refer to Fig.3.11
- Pass 1:
  - concerned only w/Header and Define records
  - PROGADDR is obtained from O/S
  - CSADDR is set accordingly
  - All external symbols are entered into External Symbol Table (Fig 3.11a)
  - Starting address and length of each CS are determined

- Pass 2:
  - does actual loading, relocation, and linking
  - As each Text record is read, the object code is moved to the specified address
    - (plus the current value of CSADDR)
  - When a Modification record is encountered, the mentioned symbol's value is added or subtracted from the indicated location in memory
- Last step: transfer control to loaded program to begin execution.
  - indicated in the End record

Remark: Algorithm can be made more efficient by changing the object code but this concerns the implementors only!

MACHINE-INDEPENDENT LOADER FEATURES

- Loading and linking are often thought as O/S service functions.
  - The programmer's connection with such services is not as direct as it is with assemblers.
  - Most loaders include fewer different features than are found in a typical assembler.
- Automatic library search process for handling external references

Automatic Library Search

- allow a programmer to use standard subroutines w/o explicitly including them in the program to be loaded.
  - In most cases there is a standard system library that is used this way
- subroutines called by the program being loaded are automatically retrieved from a library as they are needed during linking.

Implementation of Search

- linking loader must keep track of external symbols that are referred to, but not defined.
  - – enter symbol from each Refer record into the external symbol table (ESTAB)
    - –> at the end of Pass 1, the symbol in table that remain undefined represent unresolved external references
  - – the loader searches the library or libraries specified for routines that contain the definitions of these symbols
    - Subroutines fetched from a library in this may may themselves contain external references.

- The process allows the programmer to override the standard subroutines in the library by supplying his or her own routines.
- The libraries to be searched by the loader ordinarily contain assembled or compiled versions of subroutines (i.e., object programs)
  - – In most cases, a special directory is used for the libraries.
  - – Directory entry points to the address of the subroutine within the file.

Loader Options

- Many loaders allow the user to specify options that modify the standard processing described above
  - – a special command language (job control language) is used for this purpose
- Examples:
  - – Most loaders allow the user to specify alternative libraries to be searched
    - LIBRARY MYLIB

LOADER DESIGN OPTIONS

- Linking loaders
  - – perform all linking and relocation at load time
- Linkage editors
  - – perform linking prior to load time, and writes linked program (executable image) into file instead of being immediately loading into memory.
  - – found on most systems in addition to linking loaders
- Dynamic linking
  - – linking function is performed at execution time
  - – uses facilities of the O/S to load and link subprograms at the tie they are first called

Linkage Editors

- A linking loader performs all linking and relocation, including automatic library search, and loads the linked program directly into memory for execution.
- A linkage editor produces a linked version of the program (load module or executable image) which is written to a file or library for later execution
  - – a simple relocating loader can be used later to load the program into memory
  - – linkage editor performs relocation of all CSs relative to the start of the linked program
  - – loading can be accomplished in one pass w/no external symbol table required.

- A linked program is generally in a form suitable for processing by a relocating loader
  - – all external references are resolved
  - – relocating is indicated by some mechanism, such as Modification record or bit mask
  - – Even though all linking has been performed, information concerning external references is often retained in the linked program
  - – This allows subsequent re-linking of the program to replace control sections, modify external references, etc.
    - if this info is not retained, then what happens?

Suitable Work Environments

- In an environment where program is to be executed many times w/o being reassembled
  - use of a linkage editor -> reduces overhead
    - resolution of external refs and library searching are performed only once

      (Compare to a linking loader!)
- In a development and testing environment
  - a linking loader is more efficient.

- Exact (executable) image
  - if the actual address at which the program will be loaded is known in advance, the linkage editor can perform all of the needed relocation

Other Functions of Linkage Editors

1- when a change is made in the source code of a subroutine, linkage editor can replace this subroutine in the linked version of the program w/o recompiling/reassembling all code

2- can be used to build packages of CSs that are generally used together

- Ex: FORTRAN has a large number of subroutines for formatted i/o with lots of cross-refs between them
- It is desirable to keep them as separate CSs for reasons of program modularity and maintainability

Using packages

- If all of the cross-refs between library routines would have to be processed individually – same set of cross-refs would need to be processed for almost every FORTRAN program linked
- A linkage editor can be used to combine the appropriate subroutines into a package
- Since package already has all of the cross-refs between subroutines resolved, these linkages would not be processed when each user's program is linked

3- allow users to specify that external refs are not to be resolved by automatic library search

- 100 programs using the I/O routines described above stored in a library
  - If all external refs are resolved, 100 copies of the package would be stored
    - wastes memory
  - Thus only the external refs bw user-written routines would be resolved, and linking loader could be used to combine the linked user routines with the package at execution time
    - involves two separate linking operations
    - saves space

Dynamic Linking (Load on call)

- We can postpone the linking function until execution time: a routine is loaded and linked to the rest of the program when it is first called
- often used to allow several executing programs to share one copy of a subroutine or library
  - E.g., run-time support routines for a high-level language like C could be stored in a dynamic link library
    - a single copy of the routines in this library could be loaded into the memory
    - All C programs currently in execution could be linked to this one copy

- Advantages over other types of linking:
  - – provides the ability to load the routines only when (and if) they are needed
    - saves time and memory space
    - Ex: a program contains correction and diagnostic routines that may not be used al all during most executions of the program
  - – avoids loading of the entire libraries for each execution
    - Ex: a user can interactively call any of the subroutines of a large mathematical and statistical library

3- Mechanisms to accomplish the actual loading and linking of a called subroutine

- – Fig.3.14
- – routines that are to be dynamically loaded must be called via an O/S service request
  - instead of executing a JSUB, the program makes a load-and-call service request to the O/S.
  - O/S loads the routine if not already loaded
  - Control is passed from O/S to the routine being called
  - When done, control returns to O/S and then to the user's calling program
    - – O/S may free memory, or wait for a while if some other call may come soon

- When dynamic linking is used, the association of an actual address with the symbolic name of the called routine is not made until the call statement is executed:
  - – in other words, binding of the name to an actual address is delayed from load time until execution time (delayed binding)

Bootstrap Loaders

- How is the loader itself loaded into memory?
  - – OS may load it in
- Then how is the O/S loaded into memory?

"Given an idle computer w/no program in memory, how do we get things started?"

- With the machine empty and idle, there is no need for program relocation
  - – we can simply specify the absolute address for whatever program is first loaded, usually the O/S.
    - we need some means of accomplishing the functions of an absolute loader
    - Operator may enter object code of absolute loader into memory
    - absolute loader program may be permanently resident on ROM and activated by a hardware signal
    - A built-in hardware function reads a fixed-length record (called bootstrap loader) from some device into memory at a fixed location – this record may contain the machine instructions to load the absolute program in
    - Control is transferred to there
    - If the loading process requires more instructions that can be read in a single record, this first record causes the reading of others, and these in turn can read still more records , hence the term bootstrap

SunOS Linkers

- Two different linkers:
  - – link editor
  - – run-time linker
- Link editor
  - – invoked in the process of compiling a program
  - – takes one or more object modules produced by assemblers and compilers and produces a single output module:

Output module can be:

- a relocatable object module – suitable for further link-editing
- a static executable – w/all symbolic references bound and ready to run
- a dynamic executable – in which some symbolic references may need to be bound at run time
- a shared object – which provides services that can be bound at run time to one or more dynamic executables

- An object module contains
  - one or more sections: instructions and data areas
  - a list of the relocation and linking operations that need to be performed
  - a symbol table that describes the symbols used in these operations
- SunOS link-editor processes object modules and usually generates a new symbol table and a new set of relocation instructions (symbols bound at run time, relocations to be performed at load time)

- Symbolic references from the input files that do not have matching definitions are processed by referring to archives or shared objects
  - An **archive** is a collection of relocatable object modules
    - A directory stored with the archive associates symbol names with the object modules that contain their definitions
  - A **shared object** is an indivisible unit that was generated by a previous link-edit operation
    - When the link-editor encounters a reference to a symbol defined in a shared object, the entire contents of the shared object become a logical part of the output file
    - Shared object is not physically included in the output file, instead the link-editor records the dependency on the shared object
- SunOS run-time linker is used to bind dynamic executables and shared objects at run time.

Lazy Binding

- After it locates and includes the necessary shared objects, the linker performs relocation and linking operations toprepare the program for execution.
  - During link-editing, calls to globally defined procedures are converted to references to a procedure linkage table
  - When a procedure is called for the first time, control is passes via this table to the run-time linker.
  - The linker looks up the actual address of the called procedure and inserts it into the linkage table
    - subsequent calls directly go to the called procedure