Loaders and Linkers

- Object Program contains the information:
  - Translated instructions and data values
  - Addresses where these items are to be loaded
- Three things to do for executing object programs
  - Loading
  - Relocation
  - Linking
- So loader/linker
- Object programs are same no matter whether generated from assembler or compiler.
- For simplicity, just call loader (for loader, linker or both)
- As the same presentation steps and similar examples as previous chapter
  - Basic functions
  - machine-dependent features
  - Typical machine-independent features
  - Design options
  - Implementation examples

Basic functions --absolute loader

- loading of an absolute program (Figure 3.1)
- Things a loader to do
  - Check Header record
  - Read Text records and put in proper memory locations
  - When End record is encountered, jump to start
- Algorithm for an absolute loader (Figure 3.2)
  - In object program, each byte is in two characters.
  - For example, 14 is two characters 1 and 4 (i.e., ASCII 31 and 34), but needs to be converted to 14
    - 31-30 →1, 34-30→4, and then 1 and 4 are combined to form byte 14.
  - Of course, object program can be stored in binary form.
    - Two issues to remember:
      - May not be readable by people
      - May conflict with some control codes.
- No relocation and linking functions.

A bootstrap loader

- When a computer first starts or turns on:
  - Bootstrap loader (a very short program)
  - In ROM with a specific address such as 0.
  - Automatically run from the address.
  - Load OS at a specific address such as 80 and jump to 80 to execute OS after finishing loading.
- Or a hardware initiated reading of a record of bootstrap loader, which reads the next record, …, read the following codes such as OS or a standard lone program that can run without OS.
- Bootstrap loader for SIC/XE (Figure 3.3)
  - Go through the program carefully and line-by-line.
  - no error check here.

Machine-dependent features ---re-locatable program loader

- Multiple programs sharing memory requires relocatable.
- Library functions also require relocatable.
- Have relocation and linking capability.
- Relocation and linking are closed related, so discussed together.
  - In particular, linking requires relocation of subroutines linked together.

Program relocation

- Example of SIC/XE program from Figure 2.6 (Figure 3.4)
- Object Program with relocation by Modification Record (Figure 3.5)
- Modification record
  - A perfect way to implement relocation.
  - Just add the start address of the loaded program in to the instruction indicated by each Modification record.
- If without relative addressing, most instructions need Modification record
  - For each word in object program, there is a relocation bit indicating whether it needs relocation.
  - Collection of Relocation bits form relocation mask.
  - Word (or 3 bytes) alignment because of the mask.
- Relocatable program for a standard SIC program (Figure 3.6)
- Object Program with relocation by bit mask (Figure 3.7)
  - The forth Text record has less than 12 words (i.e., 10 words), but a new Text record is needed, because of the alignment of word.
- Hardware relocation:
  - For relative addressing, hardware will automatically add the offset in the instruction with some segment register during the program execution.
  - However, the linking makes it necessary for the loader to do relocation.

Program linking

- Recall Independent program units (control sections) in Chapter 2
  - External definition and reference.
  - Define record and Refer record
  - Revised Modification record.
- Sample program illustrating relocation and linking (Figure 3.8) Figure 3.8 (Cont'd)
  - Three Control Sections
  - Same set of references to external symbols
  - Three in instructions (REF1 to REF3), and five are Data Words (REF4 to REF8)
- Object program corresponding to Figure 3.8 (Figure 3.9) Figure 3.9 (cont'd)
  - First reference to LISTA in REF1:
    - In PROGA, not external, so assembled as normal, PC relative.
    - In PROGB (and PROGC as well), LISTA is external, so address is set to 0 and a Modification record is created.
  - Second reference to LISTB (+4) in REF2
    - In PROGA (and PROGC as well), LISTB is external, so 4 is stored in address field and a Modification record is created. But in PROGB, it is local.
  - Third reference to #ENDA-LISTA,
    - In PROGA, immediate value is computed and put in the instruction
    - In PROGB (and PROGC as well), both ENDA and LISTA are external, so two Modification record are generated.
  - Fourth reference REF4 WORD ENDA-LISTA+LISTC
    - In PROGA, both ENDA and LISTA are known, so evaluated and 00014 is stored as value of REF4 and a MODIFICATION record for LISTC is generated.
    - In PROGB, all are external, so three Modification records are generated and 00000 is set as the value of REF4.
    - In PROGC, LISTC is local so, 30 (the relative address of LISTC relative to the Starting address of PROGC) is stored, and three Modification records (for Adding the starting address of PROGC, adding ENDA address and subtracting LISTA address) are generated.
    - So a simple reference in PROGA, a complicated reference in PROGB, and a relocation and external reference in PROGC.

Program linking (cont.)

- Program in memory for Figure 3.8 after loading and linking (Figure 3.10 (a))
  - PROGA starting at 4000
  - REF4 to REF8 have same values in all three programs.
  - However, for references that are operands of instructions, the results are not always same, e.g. 01D for REF1 in PROGA and 4040 for REF1 in PROGB.
  - because of PC relative or base relative, but, the TA should be the same.
  - In PROGA, REF1 is PC relative, Disp is 01D. When the instruction is executed, TA= 4023 (PC)+01D= 4040.
  - Furthermore, no relocation is needed, since PC always contains the actual address of the next instruction.
  - In PROGB, the extended format is used with actual address, so 4040 is the operand value.
- Relocation and linking operation performed on REF4 from Program A (Figure 3.10(b))
  - PROGA starts at 4000,
  - REF4 locates at 4054 (the relative address of REF4 is 54) and with initial value 00014 from Text record.
  - The address of LISTC is 4112 (the starting address 40E2 of PRGOC + relative address 30 of LISTC) is added to REF4 by the Modification record.
  - So the result is 00014 + 4112 is 4126.
  - Similar in PROGB, 0000 (initial value)+ 4054 (ENDA)- 4040 (LISTA)+ 4112 (LISTC) =4126 (by three Modification records).

Algorithms and Data structures for a linking loader

- Modification record:
  - Make the linking and relocation to be processed similarly.
- Two passes:
  - Assign addresses to all external symbols
  - Perform actual loading, relocation, and linking.
- Data structures used by linking loaders
  - External symbol tab: ESTAB
    - Name, address, and control session belonging to
    - Also control sessions (name, address, length)
    - Hash table
    - ESTAB with control sessions and external symbols
  - Program load address : PROGADDR
    - The beginning address in memory where the linked program is to be loaded.
    - Supplied to loader by OS
  - Control session address: CSADDR
    - The starting address of the current control session.
    - Added to all relative addresses in the control session to get actual addresses.
  - Execution starting address: EXECADDR.

Algorithms and Data structures for a linking loader (cont.)

- Algorithms for Pass 1 of a linking loader (Figure 3.11(a)) Pass 2 (Figure 3.11(b))
  - In Pass 1:
    - Just process Header and Define records
    - PROGADDR is initialized from OS, also becoming the starting address of the first control session (CSADDR)
    - Control session is entered ESTAB with name from Header record, address from CSADDR
    - For each symbol in Define record, enter to ESTAB with name, address (value in the Define record +CSADDR)
    - When End record is encountered, CSLTH (given in Header record) is added to CSADDR, which become CSADDR of the next control session.
  - In pass 2:
    - May print a load map for all control sessions and symbols, useful in program debugging
    - Actually loading, relocation, and linking
    - CSADDR is used as in Pass 1
    - For each Text record, the object code is placed to the address (=the address in the instruction + CSADDR).
    - For each Modification record, from the name, looking for ESTAB to get the address (in another control session), then add or subtract from the indicated location.
  - Transfer the control to the starting address indicated by EXECADDR
- Object program corresponding to Fig3.8 using reference number for code modification (Figure 3.12) Figure 3.12 (cont'd)
  - Given a reference number in Refer record by assembler
  - Using the reference number in the Modification record by assembler
  - The loader stores the addresses of external references in an array indexed by their reference numbers when processing Refer records
  - Get the address of a reference number from the array when processing Modification records.

Machine independent features --Automatic Library Search

- Loader and linking are generally considered as OS features.
- Standard library such as mathematical or statistics
- By default or by parameters to allow multiple libraries to be linked from.
- At the end of pass 1, the undefined symbols in ESTAB are considered as from Libraries.
- So search the libraries containing the external definitions of these symbols and fetch the code to link.
- Maybe nested references to external symbols, so repeat search in libraries.
- If still undefined symbols, errors are reported.
- The above process allows to override standard subroutines in library.
- Directory for the symbols in libraries to accelerate the search of external references.
  - Can even be put in the memory by OS for fast and frequent search.
- Same for external references to DATA items, if any.

Machine independent features --Loader options (Parameters)

- INCLUDE program-name (library name)
  - Include another program in the library for linking
- DELETE csect-name
  - Delete a control session from the input program
- CHANGE name1, name2
  - Change name1 to name2
- Example:
  - Three control sessions: COPY, RDREC and WRREC
  - COPY is the main program for loader.
    - INCLUDE READ (UTLIB)
    - INCLUDE WRITE (UTLIB)
    - DELETE RDREC,WRREC,
    - CHANGE RDREC, READ
    - CHANGE WRREC, WRITE
- LIBRARY mylib
  - Include my own library, which is searched before standard libraries.
- Output options, such as the load map output by the loader
- Other options
  - Such as the starting execution point of the program.

Loader design options --Linkage editor

- performing linking prior to load time
  - In contrast, Linking loader: linking and loading together
  - Processing of an object program using (a) Linking loader and (b) Linkage editor
  - linking the object codes together to generate a linked program (called load module or executable image).
  - Performing the relocation of all control sections, related to the start of the linked program without any external references.
    - Simple one-pass relocating loader: the only modification is the addition of an actual load address to relative values.
  - The linked program can be executed many times with efficiency.
  - The linked program generally contains information that allows the linkage editor to re-link/replace some control sections which are modified for error correction or efficiency improvement.
  - Build package of subroutines for later use.

Loader design Options
--Dynamic linking

- Linking is performed during execution.
  - A subroutine is loaded and linked to the rest of the program when it is first called.
  - Allow several programs to share one copy of a subroutine.
  - Examples and advantages:
    - C dynamic linking library used by all running C programs.
    - In object-oriented systems, allows the implementation of an object and its methods to be determined at the time the program is running.
    - Allows the implementation of a method to dynamically change
    - Allows sharing an object by multiple programs.
    - Efficiency in the sense that only needed subroutines are linked during current running.

Dynamic linking

- One way is by OS service request or to say, by a dynamic loader which is part of OS and kept in memory.
- Loading and Calling of a subroutine using Dynamic linking (Figure 3.14)
  - Service request with subroutine name as parameter
  - OS checks whether the subroutine was already loaded in memory
    - If not, search the user-specified or system libraries and load it
  - The control is transferred to the subroutine.
  - Return to OS after the subroutine is completed and then return to the caller.
    - OS may releases the memory of this subroutine or keeps it for a while, so that a near future call will have a hit without loading.
  - Association/binding of actual address with the symbolic name is delayed to execution time.
    - More flexibility
    - So overhead, including OS intervention.

Implementation Examples --MS-DOS linker

- Complier/Assembler:
  - Source Program →Object module (.obj)
  - MS-DOS object module (Figure 3.15)
    - LEDATA similar to Text record, LIDATA: repeated records.
    - FIXUP similar to Modification record.
- Linker (Linkage editor):
  - Object codes → executable (.exe).
  - Pass 1 of Two passes
    - computing starting address of each segment,
    - segments of same name and same class from different modules are combined
    - segments of same name but different classes from different modules are concatenated.
    - Constructing a symbol table associating address with each segment and external symbol
    - Searching library for any unsolved undefined symbol, if possible.
  - Pass 2 of the two pass linkage editor
    - Extracting the translated instructions and data from object modules and building an image of the executable program in memory
    - The executable is organized by segment, not by the order of the object modules
    - Memory image allows easy rearrangement caused by combination and concatenation
    - Temporary disk file may be used if memory is not enough.
    - LEDATA/LIDATA and corresponding FIXUP are processed (placed into memory in binary format). Repeated data in LIDATA is expanded
    - relocation within a segment (caused by combination and concatenation) is performed and external reference is resolved.
    - Relocation related to starting of a segment is added to a table of segment fixup, which is used for relocation when loaded.
    - Write it to .exe file, containing segment fixups, information about memory requirement, entry points, and the initial contents for registers CS and SP.
- When .exe file is typed, OS (a loader in OS) loads the file to memory to execute.

Implementation Examples --SunOS linkers

- Two linkers
- Link-editor:
  - Take one or more object modules (by assembler or compiler) and produce one single output module as
    - A relocatable object module, suitable for further link-editor
    - A static executable, all symbols are bound and ready to run
    - A dynamic executable, with some symbols being bounded at run time
    - A shared object, that can be bound at run time to other dynamic executables by run-time linker
- Run-time linker:
  - Bind the shared objects with a dynamic executable, also check dependency among shared objects.
  - Load, re-location, and linking.
  - Lazing binding:
    - During link-editing, calls to global defined procedures are converted to references to a procedure linkage table
    - When a procedure is called first time at run time, it is passed to run-time linker via this table, the linker looks the actual address and place it in the call. Then in the future, the call directly goes to the procedure without via the table. Some how similar to dynamic linking.

Implementation Examples --Cray MPP linker

- Recall: parallel architecture consisting of a large of inter-connected processing elements (PE).
- Share programs and data
  - Example of data sharing among PEs (Figure 3.16)
- When loaded, each PE gets a copy of executable, its private data and its portion of shared data.
  - T3E program loaded on multiple PEs (Figure 3.17)
- Each PE can access the memory in all other PEs, but the access is faster to its local memory.
- Parallel processing/programming.

Summary

- Loader, Linker, Linking loader,
  - Linkage editor, simple loader, dynamic linking, absolute loader, bootstrap loader.
- Functions
  - Pass 1: Assign addresses to external symbols
  - Pass 2: loading, relocation, linking
- Data Structures: ESTAB, PROGADDR, CSADDR, EXECADDR
- Library search and linking
- Linking options
- Dynamic linking