CS1101: Lecture 40

Linking & Loading

Dr. Barry O'Sullivan
b.osullivan@cs.ucc.ie

Course Homepage
http://www.cs.ucc.ie/~osullb/cs1101

Lecture Outline

- Introduction
- Linking & Loading
- The Complete Translation Process
- Object Files and Executables
- Tasks Performed by the Linker
- Example: Loaded & Linked Program
- The Relocation Problem
- The External Reference Problem
- Structure of an Object Module
- A Final Word...
- **Reading**: Tanenbaum, Chapter 7, Section 4.

Introduction

- Most programs consist of more than one procedure.
- Compilers and assemblers generally translate one procedure at a time and put the translated output on disk.
- Before the program can be run, all the translated procedures must be found and linked together properly.
- If virtual memory is not available, the linked program must be explicitly loaded into main memory as well.
- Programs that perform these functions are called by various names, including linker, linking loader, and linkage editor.

Linking & Loading

- The complete translation of a source program requires two steps:
  1. Compilation or assembly of the source procedures.
  2. Linking of the object modules.
- The first step is performed by the compiler or assembler
- The second one is performed by the linker.

The Complete Translation Process

Figure 7-13. Generation of an executable binary program from a collection of independently translated source procedures requires using a linker.

Object Files and Executables

- The translation from source procedure to object module represents a change of level because the source language and target language have different instructions and notation.
- The linking process, however, does not represent a change of level, since both the linker's input and the linker's output are programs for the same virtual machine.
- The linker's function is to collect procedures translated separately and link them together to be run as a unit called an executable binary program.
- On MS-DOS, Windows 95/98, and NT the object modules have extension obj and the executable binary programs have extension exe.
- On UNIX, the object modules have extension $.o$; executable binary programs have no extension.

Tasks Performed by the Linker

- At the start of pass one of the assembly process, the instruction location counter is set to 0.
- Thus, we assume that the object module will be located at (virtual) address 0 during execution.
- In order to run the program, the linker brings the object modules into main memory to form the image of the executable binary program
- The idea is to make an exact image of the executable program's virtual address space inside the linker and position all the object modules at their correct locations.
- If there is not enough (virtual) memory to form the image, a disk file can be used.

Example: Loaded & Linked Program

Figure 7-14. Each module has its own address space, starting at 0.

The Relocation Problem

- Consider what would happen if execution began with the instruction at the beginning of module A.
- The program would not branch to the MOVE instruction as it should, because that instruction is now at 300.
- This problem, called the **relocation problem**, occurs because each object module in Fig. 7-14 represents a separate address space.

The External Reference Problem

- The procedure call instructions will not work either.
- At address 400, the programmer had intended to call object module B, but because each procedure is translated by itself, the assembler has no way of knowing what address to insert into the CALL B instruction.
- The address of object module B is not known until linking time.
- This problem is called the **external reference problem**.
- Both of these problems can be solved by the linker.

Tasks Performed by the Linker

- The linker merges the separate address spaces of the object modules into a single linear address space in the following steps:
  1. It constructs a table of all the object modules and their lengths.
  2. Based on this table, it assigns a starting address to each object module.
  3. It finds all the instructions that reference memory adds to each a relocation constant equal to the starting address of its module.
  4. It finds all the instructions that reference other procedures and inserts the address of these procedures in place.

An Example Object Table

- Below is the object module table constructed in step 1 for the modules of Fig. 7-15.
- It gives the name, length, and starting address of each module.

Module	Length	Starting address
A	400	100
B	600	500
c	500	1100
D	300	1600

Structure of an Object Module

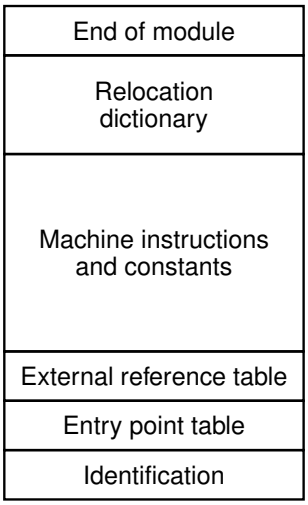

Figure 7-16. The internal structure of an object module produced by a translator.

Structure of an Object Module

- Object modules often contain six parts.
- The first part contains the name of the module, certain information needed by the linker, such as the lengths of the various parts of the module, and sometimes the assembly date.
- The second part of the object module is a list of the symbols defined in the module that other modules may reference, together with their values.
- The third part of the object module consists of a list of the symbols that are used in the module but which are defined in other modules, along with a list of which machine instructions use which symbols.

Structure of an Object Module

- The fourth part of the object module is the assembled code and constants. This part of the object module is the only one that will be loaded into memory to be executed.
- The fifth part of the object module is the relocation dictionary.
- The sixth part is an end-of-module indication, sometimes a checksum to catch errors made while reading the module, and the address at which to begin execution.

A Final Word...

- Most linkers require two passes.
- On **pass one** the linker reads all the object modules and builds up a table of module names and lengths, and a global symbol table consisting of all entry points and external references.
- On **pass two** the object modules are read, relocated, and linked one module at a time.