Program linking and object files

kaneton people

January 14, 2012

Outline

Outline

Overview

This course targets **executable formats** in an operating system.

This course will answer questions such as: how source-code is turned into object-code and what are the steps from source-code compiling to runtime execution.

Assumptions

A good knowledge of **ABI** (Application Binary Interface) should be helpful. This course will deal with stack, arguments passing, local and gloable variables, *etc.*

Having, at least once in a lifetime, taken a look at a *.o* file should make things much clearer.

Outline

Von Neumann's architecture :: Basics

Todays code and data programs structures are a direct application of the **Von Neumann's architecture**.

This architecture falls into four main parts:

- The ALU (Arithmetic Logic Unit) : Achieve basic operations such as addition and incrementation.
- The Control Unit : In charge of scheduling instructions.
- The memory : Contains both program and data.
- Inputs and outputs.

Von Neumann's architecture :: Advanced concepts

A program is read from a mass memory (*i.e.* non volatile storage such as tape and hard-drive), and is loaded into the execution memory. This is the main difference with the former Harvard architecture where code and data are physically splitted.

Initially, regarding code as data allowed code modification at runtime. For instance, a loop instruction was buit from an opcode plus the index. Later, with the rise of registers inside processors, this feature became deprecated.

Nevertheless, this architecture endures for a quite obvious reason: it permits the use of **compilers**.

Outline

The linking phase :: The linker

Linking is the process of combining various pieces of code and data together to form a single executable that can be loaded in memory.

The basic concepts of linking remain the same, regardless of the operating system, processor architecture or object file format being used.

The linker is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

The linking phase :: Symbol resolution

Assemblers and compilers produce **relocatable object files**. Relocatable means functions and variables are not binded to any address. In a relocatable object file, address are symbols (i.e. strings like assembly line: *call foo*).

Linkers combine these object files together to generate executable object files, by turning these symbols into address. In other words, the linker assigns runtime addresses to each section and each symbol. At this point, the code (functions) and data (static and global variables) will have unique **runtime addresses**.

During the link-editing of an object, any relocation information supplied with the input relocatable objects is applied to the output file.However, when creating a dynamic executable or shared object, many of the relocations cannot be completed at link-edit time. These relocations require logical addresses that are known only when the objects are loaded into memory. In these cases, the link-editor generates new relocation records as part of the output file image. The loader (runtime linker) must then process these new relocation records.

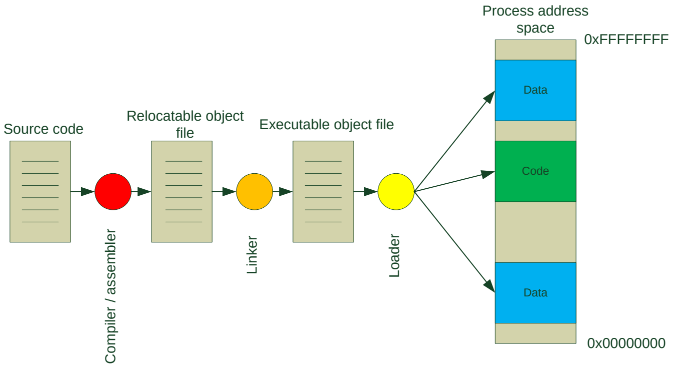
The loading phase

Program loading refers to copying a program image from hard disk to the main memory in order to put the program in a ready-to-run state.

The linker creates a program image, the loader is in charge to load this image in the system memory (virtual memory the system is running in protected mode), and to manage all other images present.

In some cases, program loading also might involve allocating storage space or mapping virtual addresses to disk pages.

Build cycle


Source code
Compiler / assembler
Relocatable object file
Linker
Executable object file
Loader
Process address space
0xFFFFFFFF
Data
Code
Data
0x00000000

Outline

Object file :: Executable format

An object file format is a computer file format used for the storage of object code and related data typically produced by a compiler or assembler.

There are many different object file formats; originally each type of computer had its own unique format, but with the advent of Unix and other portable operating systems, some formats, such as COFF and ELF, have been defined and used on different kinds of systems.

- ELF (modern UNIX, Linux, Solaris, *etc.*)
- COFF (System V)
- PE (stands for Portable Executable, Windows NT)
- DWARF
- a.out

Object file :: File structure

Most object file formats are structured as blocks of data, each block containing a certain type of data. These blocks can be paged in as needed by the virtual memory system, needing no further processing to be ready to use.

The linker, with the help of a **linker script**, builds executable object files.

The simplest object file format is the DOS *.COM* format, which is simply a file of raw bytes that is always loaded at a fixed location. Other formats are more elaborate: they may contains relocation and debugging information (COFF, ELF, *etc.*).

Types of data supported by typical object file formats:

- BSS (*Block Started by Symbol*)
- Text segement
- Data segment

Outline

Memory block :: The text segment

A.k.a text segment, text, code, text section (mis-use), code section (mis-use), *etc.*

Refers to a portion of an object file that contains executable instructions (*i.e.* the machine-code). This is why object files are usually called binaries.

Text segment is the only essential element in an object file.

It has a fixed size and is usually read-only. If the text section is not read-only, then the particular architecture allows self-modifying code.

As a memory region, a code segment resides in the lower parts of memory or at its very bottom, in order to prevent heap and stack overflows from overwriting it.

Memory block :: The data segment

The data segment in an object file contains the global variables that have been initialized by the programmer. It has a fixed size, since all of the data in this section is set by the programmer before the program is loaded.

However, it is not read-only, since the values of the variables can be altered at runtime. This is in contrast to the Rodata (constant, read-only data) section, as well as the code segment (also known as text segment).

Memory block :: The bss

It is often referred to as the ”bss section” or ”bss segment”. “bss” stands for Block Started by Symbol.

The “bss” contains uninitialized static variables. The program loader initializes the memory allocated for the bss section when it loads the program (usually, the bss segment is zero-filled).

For embedded kernel developpment, the “bss” is usually used to reserve memory space for the heap and the stack.

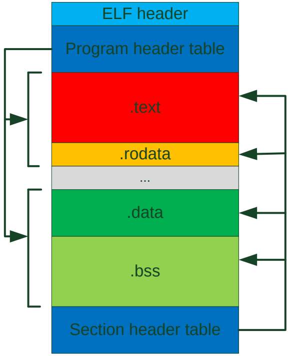
Outline

Case studies :: The ELF object file format

The segments contain information that is necessary for runtime execution of the file, while sections contain important data for linking and relocation. Sections are defined in the linker script, while segments are the products of object-code and a linker script.

Unlike many proprietary executable file formats, ELF is very flexible and extensible,and it is not bound to any particular processor or architecture. This has allowed it to be adopted by many different operating systems on many different platforms.

Case studies :: The ELF object file format



Case studies :: Writing a linker script for the WindRiver linker

```
$cat basic.c
#pragma use_section .stack
int stack[256];

#pragma use_section .code
int main()
{
  return 0;
}
```

Case studies :: Writing a linker script for the WindRiver linker

```
MEMORY
{
  flash: org = 0xF0003000, len = 0x20000
  sram: org = 0x00023000, len = 0x10000
}

SECTIONS
{
  GROUP : {
    .code : {
      *(.text)
    }
  } > flash

  GROUP : {
    .var : {
      *(.data)
    }

    .stack : {
      *(.bss)
    }
  } > sram
}

$ dld -o basic.bin -l basic.o basic.dld
```

Case studies :: Writing a linker script for the WindRiver linker

The linker command outputs a link map, referencing for each functions and variables its address.

```
$ dld -o basic.bin -l basic.o basic.dld > basic.map
```

A map file looks like:

```
g_task_pending        0x00023000      0x00000010
g_an_int              0x00023010      0x00000004

SC_krnTaskStart       0xf0003000      0x00000400
SC_krnTaskStop        0xf0003400      0x00000200
SC_krnTaskDelete      0xf0003600      0x00000800
...
```

Outline

Exercise :: 1

The following source code is linked as a UNIX standard object file.

```
#include <stdio.h>

int foo;
int bar = 90;

int main()
{
  int local1;
  int local2 = 67;

  foo = 4;
  printf("%d %d %d", local2, foo, bar);

  return 0;
}
```

1. What is the UNIX standard object file format?
2. How many sections will be used?
3. How many segments will be produced?
4. Where these functions and variables will be located in the object file?

Exercise :: 2

Loading this code as an ELF object file, what will be the output?

```
#include <stdio.h>

int bar = 90;

int main()
{
  printf("%d\n", bar);

  return 0;
}
```

Exercise :: 3

Loading this code as a raw binary object file, what will be the output?

```
#include <stdio.h>

int bar = 90;

int main()
{
  printf("%d\n", bar);

  return 0;
}
```

Outline

Conclusion

In this course, basic concepts of linking phase has been shown. Besides linking tool, modern operating systems using virtual memory and paging may need extra tools like loaders, although they are not mandatory.

Much more theory on linkers and loaders is available. Take a look at whitepapers on the subjects. Advanced linking concepts :

- Relocation algorithms
- Relaxation
- Library
- Dynamic linking
- *etc.*

Outline

- HP-UX Linker and Libraries User's Guide, HP 9000 Computers, HP
- Linkers and Libraries Guide, Sun Microsystems