

6. Instruction Set Architecture – Translation software: linker, loader, etc.

EECS 370 – Introduction to Computer Organization – Winter 2009

Prof. Todd Austin & Prof. Marios Papaefthymiou

EECS Department
University of Michigan, Ann Arbor, USA

© T. Austin & M. Papaefthymiou, 2009

The material in this presentation cannot be
copied in any form without our written permission

Announcements

□ Programming assignment and homeworks

- HW1 due today by 1:30pm at CSE 4745
- Project 1 due Fri 2/6
- HW2 going out today, due Tue 2/10

□ Lecture today:

- Finish caller/callee
- Compiler, linker, loader

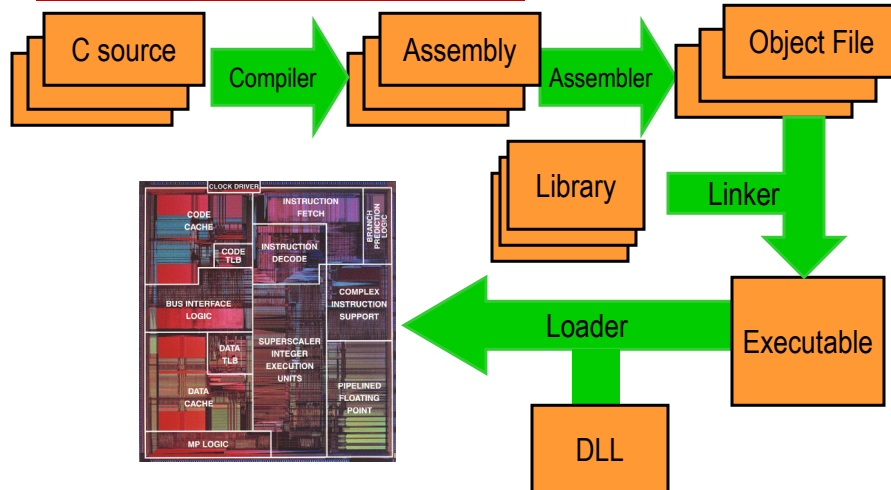
Clarification

- ❑ MIPS and branch offsets
 - Should offset values be a multiple of 4? Or 1?
 - In **assembly**: depends on the assembler
 - In **binary**: the datapath attaches “00” to the right of the immediate (see green card) – that way you can specify larger displacements

Instruction Set Architecture (ISA) Design Lectures

- ❑ Lecture 2: Storage types and addressing modes
- ❑ Lecture 3: MIPS architecture
- ❑ Lecture 4 and 5: Calling functions / passing arguments
- ❑ **Lecture 6: Translation software; libraries, VMs**

Source Code to Execution



Compiler

- ❑ C → assembly code
- ❑ 2 major parts
 - Frontend (EECS 483)
 - Parsing C syntax
 - Source-level analysis
 - Backend (EECS 583)
 - Machine independent assembly → optimizations
 - Binding variables/instructions to processor resources

Assembler

- ❑ Converts assembly (.s file) to machine code or object file (.o file)
- ❑ Generally a simple translation
 - Most assembly instructions map to 1 machine code instruction
 - Pseudo-instructions map to 1 or more machine code instructions
 - Assembler directives tell the assembler to do something
 - Allocate space for a variable/array
 - Associate a symbol with a value in the symbol table

Assembly Issues

- ❑ Reference to a label in another file
 - Data or jump address
 - Only global labels can be referenced from another file
- ❑ Assume start at fixed address (say 0) for each procedure
 - But what about multiple procedures?
- ❑ Machine code is **NOT** executable!

Unix object file format

Object files contain more than just machine code instructions!

Header: (this is an object file) contains sizes of other parts

Text: machine code

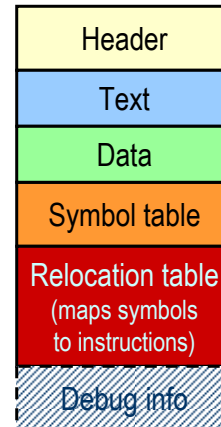
Data: global and static data

Symbol table: symbols and values

Relocation table: references to addresses that may change

Debug info: mapping of object back to source (only exists when debugging options are turned on)

Object code format

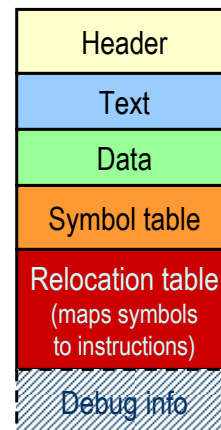


Unix object file format (2)

Object code format

Header

- size of other pieces in file
- size of text segment
- size of static segment
- size of symbol table
- size of relocation table



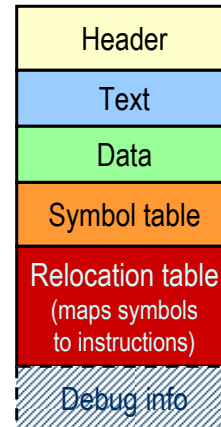
Unix object file format (3)

Text segment

- machine code

By default this segment is assumed to be read-only and that is enforced by the OS.

Object code format



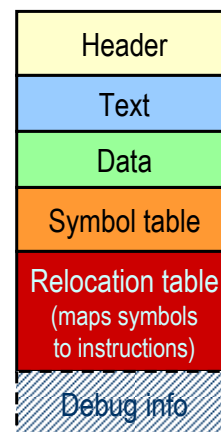
Unix object file format (4)

Data segment (Initialized static segment)

- values of initialized globals
- values of initialized static locals

Doesn't contain un-initialized data.
Just keep track of how much memory is
needed for un-initialized data.

Object code format

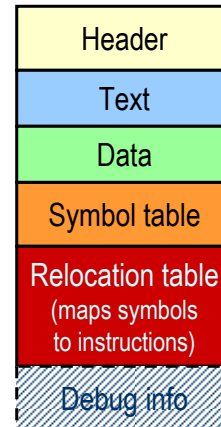


Unix object file format (5)

Symbol table:

- It is used by the linker to bind public entities within this object file (function calls and globals)
- Maps string symbol names to values (addresses or constants)
- Associates addresses with global labels. Also lists unresolved labels.

Object code format



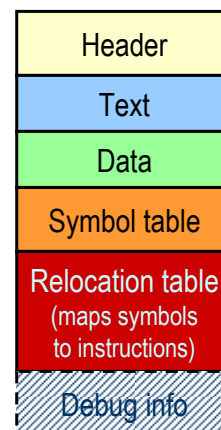
Unix object file format (6)

Relocation table :

identifies instructions and data words that rely on absolute addresses. These references must change if portions of program are moved in memory.

Used by linker to update symbol uses (e.g., branch target addresses)

Object code format



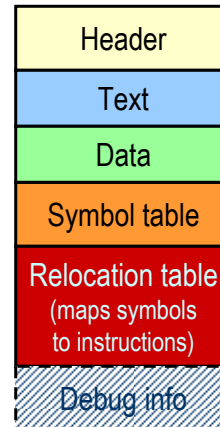
Unix object file format (7)

Debug info (optional) :

Contains info on where variables are in stack frames and in the global space, types of those variables, source code line numbers, etc..

Debuggers use this information to access debugging info at runtime

Object code format



Object file - example

Header	Name	foo	
	Text size	0x100	
	Data size	0x20	
Text	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal B	
	...		
Data	0	X	
	...		
Symbol table	Label	Address	
	X	0	
	B	-	
Reloc table	Addr	Instruction type	Dependency
	0	lw	X
	4	jal	B

Linker

- ❑ Stitches independently created object files into a single executable file (i.e., a.out)
 - Libraries are just special object files.
 - You create new libraries by making lots of object files (for the components of the library) and combining them (see ar and ranlib on Unix machines).

- ❑ Resolve cross-file references to labels
 - Make sure there are no undefined labels

Linker - continued

- ❑ Determine the memory locations the code and data of each file will occupy
 - Each function could be assembled on its own
 - Thus the relative placement of code/data is not known up to this point
 - **Must relocate absolute references to reflect placement by the linker**

- ❑ Executable file contains no relocation info or symbol table – these just used by assembler/linker

Example Executable File

Header	Text size	0x200
	Data size	0x40
Text	Address	Instruction
	0x0040 0000	lw \$a0, 0x8000 (\$gp)
	0x0040 0004	jal 0x400100
	...	
	0x0040 0100	sw \$a1, 0x8020 (\$gp)
	0x0040 0104	jal 0x400000
Data	0x1000 0000	X
	...	
	0x1000 0020	Y
	...	

Class Problem 1

In the object file for file1 and file2, what is in the symbol table and the relocation table?

```

file1.c/s
int a;
void foo(int b)
{
    int d[20];
    load a;
    load e;
    L1: load d;
    ...
    beq a e L1;
    jal bar
}
    
```

```

file2.c/s
int e;
void bar()
{
    int f;
    ...
    load f;
    load a;
    ...
    jal printf
}
    
```

Loader

- ❑ Executable file is sitting on the disk
- ❑ Puts the executable file code image into memory and asks the operating system to schedule it as a new process.
 - This used to be fairly straightforward.
 - Times are changing; it is not simple anymore.
 - We now delay some of the linking to load time.
 - Some systems even delay some code optimization (usually a compiler job) to load time.
 - Loaders must deal with more sophisticated operating systems

Trends in Software Systems

- ❑ Programmers are expensive.
- ❑ Applications are more sophisticated.
 - Pop-down menus, streaming video, etc
- ❑ Application programmers rely more on library code to make high quality apps while reducing development time.
 - This means that more of the executable is library code
 - Why not keep those shared library routines in memory and link an object file at load time? (DLLs)
 - Executable files are smaller (not very important)
 - Updating library routines is easy (a new voice activated pop-down menu).

Next Topic

- ❑ We've been looking at the instruction set
 - How to tell the processor what to do
- ❑ Next is the real **hardware!!!**
 - Basic hardware building blocks
 - Architecture of a simple processor
 - Read Chapter 5 and Appendix B