Assemblers, Linkers, and Loaders

Prof. Hakim Weatherspoon
CS 3410, Spring 2015
Computer Science
Cornell University

See: P&H Appendix A.1-2, A.3-4 and 2.12
Administrivia

Upcoming agenda

- PA2 Work-in-Progress due yesterday, Monday, March 16th
- PA2 due next week, Thursday, March 26th
- HW2 available later today, due before Prelim2 in April
- Spring break: Saturday, March 28th to Sunday, April 5th
Academic Integrity

All submitted work must be your own

• OK to study together, but do NOT share soln’s
  e.g. CANNOT email soln, look at screen, writ soln for others
• Cite your (online) sources
• “Crowd sourcing” your problem/soln same as copying

Project groups submit joint work

• Same rules apply to projects at the group level
• Cannot use of someone else’s soln

Closed-book exams, no calculators

• Stressed? Tempted? Lost?
  • Come see me before due date!

Plagiarism in any form will not be tolerated
Academic Integrity

“Black Board” Collaboration Policy

• Can discuss approach together on a “black board”
• Leave and write up solution independently
• Do not copy solutions

Plagiarism in any form will not be tolerated
Goal for Today: Putting it all Together

Compiler output is assembly files

Assembler output is obj files

Linker joins object files into one executable

Loader brings it into memory and starts execution
Goal for Today: Putting it all Together

Compiler output is assembly files

Assembler output is obj files
  • How does the assembler resolve references/labels?
  • How does the assembler resolve external references?

Linker joins object files into one executable
  • How does the linker combine separately compiled files?
  • How does linker resolve unresolved references?
  • How does linker relocate data and code segments

Loader brings it into memory and starts execution
  • How does the loader start executing a program?
  • How does the loader handle shared libraries?
Big Picture

Assembler output is obj files
  • Not executable
  • May refer to external symbols
  • Each object file has its own address space

Linker joins these object files into one executable

Loader brings it into memory and executes
Big Picture

C source files → calc.c → calc.s → calc.o
math.c → math.s → math.o
io.s → io.o

Assembly files

obj files

Executable program

Exists on disk

Loader

Executing in Memory process

Compiler

Assembler

Linker
Anatomy of an executing program

- .text
- .data
- system reserved
- stack
- dynamic data (heap)
- static data
- code (text)
- system reserved
**Example #2: Review of Program Layout**

**calc.c**

```c
vector* v = malloc(8);
v->x = prompt("enter x");
v->y = prompt("enter y");
int c = pi + tnorm(v);
print("result %d", c);
```

**math.c**

```c
int tnorm(vector* v) {
    return abs(v->x)+abs(v->y);
}
```

**lib3410.o**

- Global variable: `pi`
- Entry point: `prompt`
- Entry point: `print`
- Entry point: `malloc`

**Diagram: Stack and Memory Layout**

- Stack: `v`, `c`
- Dynamic data (heap): `v`
- Code (text): `abs`, `tnorm`, `prompt`, `print`, `main`
- System reserved
Anatomy of an executing program

Code Stored in Memory (also, data and stack)

- System reserved
- Stack
- Data
- Code (text)
- System reserved

PC → new pc

Instruction Fetch

- $0 (zero)
- $1 ($at)
- Register file
- $29 ($sp)
- $31 ($ra)

Instruction Decode

- Extend
- Detect hazard

Instruction Fetch

IF/ID

ID/EX

EX/MEM

MEM/WB

- Forward unit
- Execute
- Memory

Write-Back

- Alu
- Control
- Memory

Compute jump/branch targets

Jump/branch targets

- Compute jump/branch targets

Data, Code

- Stacked

Data, Code, Code

- Stacked
Big Picture: Assembling file separately

Output of assembler is a object files
- Binary machine code, but not executable
- How does assembler handle forward references?

- math.c
- math.s
- math.o

.o = Linux
.obj Windows
Next Goal

How does the assembler handle local references
How does Assembler handle forward references

Two-pass assembly

• Do a pass through the whole program, allocate instructions and lay out data, thus determining addresses
• Do a second pass, emitting instructions and data, with the correct label offsets now determined

One-pass (or backpatch) assembly

• Do a pass through the whole program, emitting instructions, emit a 0 for jumps to labels not yet determined, keep track of where these instructions are
• Backpatch, fill in 0 offsets as labels are defined
How does Assembler handle forward references

Example:
- bne $1, $2, \text{L}
  sll $0, $0, 0
\text{L:} addiu $2, $3, 0x2

The assembler will change this to
- bne $1, $2, +1
  sll $0, $0, 0
  addiu $2, $3, $0x2

Final machine code
- 0X14220001 \# bne
  0x00000000 \# sll
  0x24620002 \# addiu
How does Assembler handle forward references

Example:

• bne $1, $2, L
  sll $0, $0, 0
  L: addiu $2, $3, 0x2

The assembler will change this to

• bne $1, $2, +1
  sll $0, $0, 0
  addiu $2, $3, $0x2

Final machine code

• 0X14220001 # bne
  0x00000000 # sll
  0x24620002 # addiu
How does Assembler handle forward references

Example:

• bne $1, $2, L
  sll $0, $0, 0
L: addiu $2, $3, 0x2

The assembler will change this to

• bne $1, $2, +1
  sll $0, $0, 0
  addiu $2, $3, $0x2

Final machine code

• 0X14220001 # bne
  0x00000000 # sll
  0x24620002 # addiu
Output of assembler is a object files

- Binary machine code, but not executable
- How does assembler handle forward references?
- May refer to external symbols  i.e. Need a “symbol table”
- Each object file has illusion of its own address space
  - Addresses will need to be fixed later
    e.g. .text (code) starts at addr 0x00000000
    .data starts @ addr 0x00000000
Next Goal

How does the assembler handle external references
Symbols and References

Global labels: Externally visible “exported” symbols
• Can be referenced from other object files
• Exported functions, global variables
  e.g. pi
  (from a couple of slides ago)

Local labels: Internal visible only symbols
• Only used within this object file
• static functions, static variables, loop labels, ...

  e.g.
  static foo
  static bar
  static baz

  e.g.
  $str
  $L0
  $L2
Object file

Header
• Size and position of pieces of file

Text Segment
• instructions

Data Segment
• static data (local/global vars, strings, constants)

Debugging Information
• line number → code address map, etc.

Symbol Table
• External (exported) references
• Unresolved (imported) references
Example

```
math.c

int pi = 3; global
int e = 2;
static int randomval = 7; local (to current file)
extern char *username;
extern int printf(char *str, ...);
    external (defined in another file)
int square(int x) { ... }
static int is_prime(int x) { ... } local
int pick_prime() { ... } global
int pick_random() {
    return randomval;
}
```

Compiler
```
gcc -S .. math.c
```

Assembler
```
gcc -c .. math.s
```

```
objdump --disassemble math.o
objdump --syms math.o
```
Objdump disassembly

csug01 ~$ mipsel-linux-objdump --disassemble math.o

math.o: file format elf32-tradlittlemips

Disassembly of section .text:

00000000 <pick_random>:
  0:  27bdfff8  addiu   sp,sp,-8
  4:  afbe0000  sw       s8,0(sp)
  8:  03a0f021  move    s8,sp
 c:  3c020000  lui      v0,0x0
10:  8c420008  lw        v0,8(v0)
14:  03c0e821  move    sp,s8
18:  8fbe0000  lw        s8,0(sp)
1c:  27bd0008  addiu    sp,sp,8
20:  03e00008  jr        ra
24:  00000000  nop

00000028 <square>:
  28:  27bdfff8  addiu    sp,sp,-8
  2c:  afbe0000  sw        s8,0(sp)
 30:  03a0f021  move      s8,sp
 34:  afc40008  sw        a0,8(s8)
csug01 ~$ mipsel-linux-objdump --disassemble math.o

math.o: file format elf32-tradlittlemips

Disassembly of section .text:

Address instruction Mem[8] = instruction 0x03a0f021 (move s8,sp)

00000000 <pick_random>:

0: 27bdfff8     addiu    sp,sp,-8
4: afbe0000   sw    s8,0(sp)
8: 03a0f021   move    s8,sp
12: 3c020000 lui v0,0x0
14: 8c420008 lw v0,8(v0)
16: 03c0e821 move sp,s8
18: 8fbe0000 lw s8,0(sp)
20: 27bd0008 addiu sp,sp,8
24: 03e00008 jr ra

00000028 <square>:

28: 27bdfff8     addiu    sp,sp,-8
2c: afbe0000   sw    s8,0(sp)
30: 03a0f021   move    s8,sp
34: afce0008   sw    a0,8(s8)
**Objdump symbols**

```bash
csug01 ~$ mipsel-linux-objdump --syms math.o
math.o:   file format elf32-tradlittlemips
```

**SYMBOL TABLE:**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Type</th>
<th>Symbol</th>
<th>File Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>l</td>
<td>df <em>ABS</em></td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>.text</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>.data</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>.bss</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>.mdebug.abi32</td>
<td>00000000</td>
</tr>
<tr>
<td>00000008</td>
<td>l</td>
<td>O .data</td>
<td>00000004</td>
</tr>
<tr>
<td>00000060</td>
<td>l</td>
<td>F .text</td>
<td>00000028</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d .rodata</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d .comment</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>g</td>
<td>O .data</td>
<td>00000004</td>
</tr>
<tr>
<td>00000000</td>
<td>g</td>
<td>O .data</td>
<td>00000004</td>
</tr>
<tr>
<td>00000000</td>
<td>g</td>
<td>F .text</td>
<td>00000028</td>
</tr>
<tr>
<td>00000028</td>
<td>g</td>
<td>F .text</td>
<td>00000038</td>
</tr>
<tr>
<td>00000088</td>
<td>g</td>
<td>F .text</td>
<td>0000004c</td>
</tr>
<tr>
<td>00000000</td>
<td><em>UND</em></td>
<td>username</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td><em>UND</em></td>
<td>printf</td>
<td>00000000</td>
</tr>
</tbody>
</table>
## Objdump symbols

```bash
$ mipsel-linux-objdump --syms math.o
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Segment</th>
<th>Function</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>l: local</td>
<td>df <em>ABS</em></td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l: local</td>
<td>.text</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l: local</td>
<td>.data</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l: local</td>
<td>.bss</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l: local</td>
<td>.mdebug.abi32</td>
<td>00000000</td>
</tr>
<tr>
<td>00000008</td>
<td>l: local</td>
<td>.data</td>
<td>00000004</td>
</tr>
<tr>
<td>00000060</td>
<td>l: text</td>
<td>.text</td>
<td>00000028</td>
</tr>
<tr>
<td>00000060</td>
<td>l: text</td>
<td>.rodata</td>
<td>00000000</td>
</tr>
<tr>
<td>00000060</td>
<td>l: text</td>
<td>.comment</td>
<td>00000000</td>
</tr>
<tr>
<td>00000060</td>
<td>l: text</td>
<td>.data</td>
<td>00000004</td>
</tr>
<tr>
<td>00000060</td>
<td>l: text</td>
<td>.data</td>
<td>00000004</td>
</tr>
<tr>
<td>00000060</td>
<td>l: text</td>
<td>.text</td>
<td>00000028</td>
</tr>
<tr>
<td>00000060</td>
<td>l: text</td>
<td>.text</td>
<td>00000038</td>
</tr>
<tr>
<td>00000060</td>
<td>l: text</td>
<td>.text</td>
<td>0000004c</td>
</tr>
<tr>
<td>00000060</td>
<td>l: text</td>
<td>.text</td>
<td>00000000</td>
</tr>
<tr>
<td>00000060</td>
<td>l: text</td>
<td>.text</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Static local
- func @
  - addr=0x60
  - size=0x28 bytes

External reference
- *UND*

Reference
- *UND*
Separate Compilation

Q: Why separate compile/assemble and linking steps?

a) Removes the need to recompile the whole program
b) Need to just recompile a small module
c) Separation of concern: Linker coalesces object files
d) All the above
e) None of the above
Separate Compilation

Q: Why separate compile/assemble and linking steps?

A: Separately compiling modules and linking them together obviates the need to recompile the whole program every time something changes

- Need to just recompile a small module

- A linker coalesces object files together to create a complete program
Linkers
Next Goal

How do we link together separately compiled and assembled machine object files?
Linkers

Linker combines object files into an executable file

- Relocate each object’s text and data segments
- Resolve as-yet-unresolved symbols
- Record top-level entry point in executable file

End result: a program on disk, ready to execute

- E.g. ./calc Linux
  ./calc.exe Windows
  simulate calc Class MIPS simulator
External references need to be resolved (fixed)

Steps
1) Find UND symbols in symbol table
2) Relocate segments that collide

- e.g. `uname @0x00`
- `pi @ 0x00`
- `square @ 0x00`
- `main @ 0x00`
Linker Example

main.o

.text

.entry 00 T main

.dataloc 00 D uname

*UND* printf

*UND* pi

40, JAL, printf

4C, LW/gp, pi

50, JAL, square

math.o

... 21032040

text:

.data

Entry: 3C T printf

3C T printf

printf.o

calc.exe

.text Symbol tbl Relocation info

0040 0000 0077616B

0040 0100

LUI 1000 ORI 0004

0040 0200

$4 = pi

square

LW $4, -32764($gp)

1000 0000

1000 0004

Entry: 0040 0100

text: 0040 0000

data: 1000 0000
Object file

Header
  • location of main entry point (if any)

Text Segment
  • instructions

Data Segment
  • static data (local/global vars, strings, constants)

Relocation Information
  • Instructions and data that depend on actual addresses
  • Linker patches these bits after relocating segments

Symbol Table
  • Exported and imported references

Debugging Information
Object File Formats

Unix
- a.out
- COFF: Common Object File Format
- ELF: Executable and Linking Format
- ...

Windows
- PE: Portable Executable

All support both executable and object files
Loaders and Libraries
Big Picture

calc.c -> calc.s -> calc.o
math.c -> math.s -> math.o
io.s -> io.o
libc.o
libm.o

Executable program: calc.exe

Exists on disk
Loader

Executing in Memory process
Loaders

*Loader* reads executable from disk into memory

- Initializes registers, stack, arguments to first function
- Jumps to entry-point

Part of the Operating System (OS)
Static Libraries

*Static Library*: Collection of object files (think: like a zip archive)

Q: But every program contains entire library!
A: Linker picks only object files needed to resolve undefined references at link time

e.g. libc.a contains many objects:
- `printf.o, fprintf.o, vprintf.o, sprintf.o, snprintf.o, ...`
- `read.o, write.o, open.o, close.o, mkdir.o, readdir.o, ...`
- `rand.o, exit.o, sleep.o, time.o, ...`
Shared Libraries

Q: But every program still contains part of library!
A: shared libraries

• executable files all point to single *shared library* on disk
• final linking (and relocations) done by the loader

Optimizations:

• Library compiled at fixed non-zero address
• Jump table in each program instead of relocations
• Can even patch jumps on-the-fly
Direct Function Calls

Direct call:

\[ 00400010 \ <\text{main}>: \]

\[ \ldots \]

\[ \text{jal} \ 0x00400330 \]

\[ \ldots \]

\[ \text{jal} \ 0x00400620 \]

\[ \ldots \]

\[ \text{jal} \ 0x00400330 \]

\[ \ldots \]

\[ 00400330 \ <\text{printf}>: \]

\[ \ldots \]

\[ 00400620 \ <\text{gets}>: \]

\[ \ldots \]

Drawbacks:

Linker or loader must edit every use of a symbol (call site, global var use, ...)

Idea:

Put all symbols in a single “global offset table”

Code does lookup as needed
Indirect Function Calls

Indirect call:

0x00400010 <main>:

jal 0x00400330

jal 0x00400620

jal 0x00400330

0x00400330

GOT: global offset table

0x00400010 # main
0x00400330 # printf
0x00400620 # gets

0x00400330 <printf>:

0x00400620 <gets>:
Indirect Function Calls

Indirect call:

\textbf{00400010} \texttt{<main>}: 

\begin{align*}
\text{l}w & \text{ } $t9, -32708($gp) \\
\text{j}a\text{l}r & \text{ } $t9 \\
\text{l}w & \text{ } $t9, -32704($gp) \\
\text{j}a\text{l}r & \text{ } $t9 \\
\end{align*}

\textbf{00400330} \texttt{<printf>}: 

\begin{align*}
\text{l}w & \text{ } $t9, -32708($gp) \\
\text{j}a\text{l}r & \text{ } $t9 \\
\end{align*}

\textbf{00400620} \texttt{<gets>}: 

\begin{align*}
\text{l}w & \text{ } $t9, -32708($gp) \\
\text{j}a\text{l}r & \text{ } $t9 \\
\end{align*}

\textbf{GOT: global offset table}

\begin{align*}
0 & 0x00400010 \# \text{main} \\
4 & 0x00400330 \# \text{printf} \\
8 & 0x00400620 \# \text{gets} \\
\end{align*}

\begin{align*}
\text{data segment} \\
\text{to be loaded} \\
\text{at } -32712($gp) \\
\text{printf} & = 4+(-32712)+$gp \\
\text{gets} & = 8+(-32712)+$gp
\end{align*}

Indirect Function Calls

Indirect call:

00400010 <main>:

...jal 0x00400330...jal 0x00400620...jal 0x00400330...

00400330 <printf>:

...printf...

00400620 <gets>:

...gets...

# data segment
.got

.word 0x00400010 # main

.word 0x00400330 # printf

.word 0x00400620 # gets

# global offset table
# to be loaded
# at -32712($gp)

printf = 4+(-32712)+$gp

gets = 8+(-32712)+$gp

lw $t9,-32708($gp)
jalr $t9

lw $t9,-32704($gp)
jalr $t9

lw $t9,-32708($gp)
jalr $t9

lw $t9,-32708($gp)
jalr $t9
Dynamic Linking

Indirect call with on-demand dynamic linking:

```assembly
00400010 <main>:
    ...
    # load address of prints
    # from .got[1]
    lw t9, -32708(gp)

    # now call it
    jalr t9
    ...

.got
    .word 00400888 # open
    .word 00400888 # prints
    .word 00400888 # gets
    .word 00400888 # foo
```
Dynamic Linking

Indirect call with on-demand dynamic linking:

```
00400010 <main>:
    ...
    # load address of prints
    # from .got[1]
    lw t9, -32708(gp)
    # also load the index 1
    li t8, 1
    # now call it
    jalr t9
    ...
.got
    .word 00400888 # open
    .word 00400888 # prints
    .word 00400888 # gets
    .word 00400888 # foo
```

```
00400888 <dlresolve>:
    # t9 = 0x400888
    # t8 = index of func that
    # needs to be loaded
    ...
    # load that func
    ... # t7 = loadfromdisk(t8)
    ...
    # save func’s address so
    # so next call goes direct
    ... # got[t8] = t7
    # also jump to func
    jr t7
    # it will return directly
    # to main, not here
```
Dynamic Shared Objects

Windows: dynamically loaded library (DLL)
  • PE format

Unix: dynamic shared object (DSO)
  • ELF format

Unix also supports Position Independent Code (PIC)
  – Program determines its current address whenever needed (no absolute jumps!)
  – Local data: access via offset from current PC, etc.
  – External data: indirection through Global Offset Table (GOT)
  – ... which in turn is accessed via offset from current PC
Static and Dynamic Linking

Static linking
• Big executable files (all/most of needed libraries inside)
• Don’t benefit from updates to library
• No load-time linking

Dynamic linking
• Small executable files (just point to shared library)
• Library update benefits all programs that use it
• Load-time cost to do final linking
  – But dll code is probably already in memory
  – And can do the linking incrementally, on-demand
Recap

Compiler output is assembly files

Assembler output is obj files

Linker joins object files into one executable

Loader brings it into memory and starts execution