Assemblers, Linkers, and Loaders

Prof. Kavita Bala and Prof. Hakim Weatherspoon

CS 3410, Spring 2014

Computer Science

Cornell University

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

- **PA2 Work-in-Progress** due yesterday, Monday, March 17th
- **PA2** due next week, Thursday, March 27th
- **HW2** available, due before Prelim2 in April
- **Spring break:** Saturday, March 29th to Sunday, April 6th
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, write 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?
C source files → Compiler → obj files
assembly files → Assembler → obj files
obj files → linker → executable program
Executable program exists on disk → loader → process exists on disk
Executing in Memory → Executing in Memory
Anatomy of an executing program:

0xfffffffffc: system reserved
0x80000000
0x7ffffffffc: stack
0x10000000
0x100000000: dynamic data (heap)
0x00400000: static data
0x00400000: code (text)
0x00000000: system reserved

.top

 bottoms
vector* v = malloc(8);
v->x = prompt("enter x");
v->y = prompt("enter y");
int c = pi + tnorm(v);
print("result %d", 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
Anatomy of an executing program

Code Stored in Memory (also, data and stack)

$0$ (zero)
$1$ ($at$)
$29$ ($sp$)
$31$ ($ra$)

PC

Instruction Fetch

new pc

Instruction Decode

extend

detect hazard

compute jump/branch targets

Instruction Execution

forward unit

alu

Memory

MEM/ WB

Write-Back

Stack, Data, Code Stored in Memory

$0$ (zero)
$1$ ($at$)
$29$ ($sp$)
$31$ ($ra$)
Output of assembler is a object files

- Binary machine code, but not executable
- How does assembler handle forward references?

Big Picture: Assembling file separately
.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, 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
  00010100010001000100000000000001
  00000000000000000000000000000000
  00100100011000100000000000000010`
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
```c
int pi = 3;  // global
int e = 2;   // external
static int randomval = 7;  // local (to current file)
extern char *username;  // external (defined in another file)
extern int printf(char *str, ...);  // external (defined in another file)
int square(int x) { ... }  // local
static int is_prime(int x) { ... }  // local
int pick_prime() { ... }  // global
int pick_random() {
    return randomval;
}
```
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)
Objdump disassembly

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: 27bdff8  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)
Objdump symbols

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

SYMBOL TABLE:
00000000 l  df  *ABS*                         00000000 math.c
00000000 l  d   .text                          00000000 .text
00000000 l  d   .data                          00000000 .data
00000000 l  d   .bss                           00000000 .bss
00000000 l  d   .mdebug.abi32                  00000000 .mdebug.abi32
00000008 l  O   .data                          00000004 randomval
00000060 l  F   .text                          00000028 is_prime
00000000 l  d   .rodata                        00000000 .rodata
00000000 l  d   .comment                      00000000 .comment
00000000 g   O   .data                          00000004 pi
00000004 g   O   .data                          00000004 e
00000000 g   F   .text                          00000028 pick_random
00000028 g   F   .text                          00000038 square
00000088 g   F   .text                          0000004c pick_prime
00000000         *UND*                           00000000 username
00000000         *UND*                           00000000 printf
```
<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Origin</th>
<th>Segment</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>l</td>
<td>df</td>
<td>.text</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d</td>
<td>.text</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d</td>
<td>.data</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d</td>
<td>.bss</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d</td>
<td>.mdebug.abi32</td>
<td>00000000</td>
</tr>
<tr>
<td>00000008</td>
<td>l</td>
<td>O</td>
<td>.data</td>
<td>00000004</td>
</tr>
<tr>
<td>00000060</td>
<td>l</td>
<td>F</td>
<td>.text</td>
<td>00000028</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d</td>
<td>.rodata</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d</td>
<td>.comment</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>g</td>
<td>O</td>
<td>.data</td>
<td>00000004</td>
</tr>
<tr>
<td>00000004</td>
<td>g</td>
<td>O</td>
<td>.data</td>
<td>00000004</td>
</tr>
<tr>
<td>00000000</td>
<td>g</td>
<td>F</td>
<td>.text</td>
<td>00000028</td>
</tr>
<tr>
<td>00000028</td>
<td>g</td>
<td>F</td>
<td>.text</td>
<td>00000038</td>
</tr>
<tr>
<td>00000088</td>
<td>g</td>
<td>F</td>
<td>.text</td>
<td>0000004c</td>
</tr>
<tr>
<td>00000000</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address: l: local, g: global, segment: size, Static local func @ addr=0x60 size=0x28 bytes
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?
Big Picture

calc.c ➔ calc.s ➔ calc.o
math.c ➔ math.s ➔ math.o
io.s ➔ io.o

libc.o
libm.o

calc.exe

linker

Executing in Memory
**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`
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
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  <main>:
  ...
  jal 0x00400330
  ...
  jal 0x00400620
  ...
  jal 0x00400330
  ...
  00400330  <printf>:
  ...
  00400620  <gets>:
  ...

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:

```
00400010 <main>:
    ...
    jal 0x004000330
    ...
    jal 0x00400620
    ...
    jal 0x004000330
    ...
00400330 <printf>:
    ...
00400620 <gets>:
    ...
```

GOT: global offset table

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400010</td>
<td>main</td>
</tr>
<tr>
<td>0x00400330</td>
<td>printf</td>
</tr>
<tr>
<td>0x00400620</td>
<td>gets</td>
</tr>
</tbody>
</table>
```
Indirect Function Calls

Indirect call: 

00400010 <main>:

... 

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

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

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

00400330 <printf>:

... 

00400620 <gets>:

... 

GOT: global offset table

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00400010 # main</td>
</tr>
<tr>
<td>4</td>
<td>0x00400330 # printf</td>
</tr>
<tr>
<td>8</td>
<td>0x00400620 # gets</td>
</tr>
</tbody>
</table>

# data segment

# global offset table

# to be loaded

# at -32712($gp)

# printf = 4+(-32712)+$gp

# gets = 8+(-32712)+$gp
Indirect call:

```
00400010 <main>:
   ...  
   lw $t9,-32708($gp)
   jalr $t9
   ...  
   lw $t9,-32704($gp)
   jalr $t9
   ...  
   lw $t9,-32708($gp)
   jalr $t9
   ...  
```

```
00400330 <printf>:
   ...  
00400620 <gets>:
   ...  
```

Indirect Function Calls

```
# 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
```
Dynamic Linking

Indirect call with on-demand dynamic linking:

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>:
  ...
  0x400888 # 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