Assemblers, Linkers, and Loaders

Hakim Weatherspoon

CS 3410, Spring 2013

Computer Science

Cornell University

See: P&H Appendix B.3-4 and 2.12
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
Upcoming agenda

- PA2 Design Doc due *yesterday*, Monday, March 11\(^{th}\)
- HW3 due *this* Wednesday, March 13\(^{th}\)
- PA2 Work-in-Progress circuit due *before* spring break

- **Spring break**: Saturday, March 16\(^{th}\) to Sunday, March 24\(^{th}\)

- Prelim2 Thursday, March 28\(^{th}\), right after spring break

- PA2 due Thursday, April 4\(^{th}\)
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?
Compiler, Assembler, Linker, Loader

C source files

Compiler

Calc.c

Math.c

Assembly files

Assembler

Calc.s

Math.s

Io.s

Calc.o

Math.o

Io.o

Obj files

Linker

Libc.o

Libm.o

Executable program

Loader

Calc.exe

Exists on disk

Executing in Memory process

Loader

Program
Anatomy of an executing program

- System reserved
- Stack
- Dynamic data (heap)
- Static data
- Code (text)
- System reserved
Example: Review of Program Layout

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

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

```c
global variable: pi
entry point: prompt
entry point: print
entry point: malloc
```

```
stack
```

```
dynamic data (heap) v
```

```
pi
"result %d"
static data "enter x" "enter y"
```

```
code (text)
```

```
tnorm
abs
main
```

```
system reserved
```

```
system reserved
```

```
stack v c
```

```
dynamic data (heap) v
```

```
code (text)
tnorm
abs
main
```

```
system reserved
```

```
system reserved
```
Anatomy of an executing program

- **Instruction Fetch (IF/ID)**: Fetches the instruction from memory.
- **Instruction Decode (ID/EX)**: Decodes the fetched instruction.
- **Execute (EX/MEM)**: Performs the operation specified by the instruction.
- **Memory (MEM/WB)**: Retrieves operands from memory and writes results back.

**Code Stored in Memory (also, data and stack)**:
- $0$ (zero)
- $1$ ($at$)
- register $29$ ($sp$)
- $31$ ($ra$)

**Register File (alu)**: Contains registers for holding data.

**Control (ctrl)**: Manages the flow of the program.

**Forward Unit**: Detects hazards and advances the program accordingly.

**Jump/Branch Targets**: Computes the target address for jumps or branches.

**System Reserved and Stack, Data, Code**: Stored in memory.

**alu (ALU)**: Performs arithmetic and logical operations.

**PC (Program Counter)**: Points to the next instruction to be fetched.

**Write-Back**: Writes back the results to memory.
Output of assembler is an object file.

- Binary machine code, but not executable
- How does assembler handle forward 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
Big Picture: Assembling file separately

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
Symbols and References

Global labels: Externally visible “exported” symbols
- Can be referenced from other object files
- Exported functions, global variables

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

Examples:
- e.g. pi (from a couple of slides ago)
- 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
#include <stdio.h>

#define PI 3.14159

int main() {
    int pi = 3;
    int e = 2;
    static int randomval = 7;

    extern char *username;
    extern int printf(char *str, ...);

    int square(int x) { ... }
    static int is_prime(int x) { ... }
    int pick_prime() { ... }
    int pick_random() {
        return randomval;
    }

    // Compilation steps
    gcc -S math.c
    gcc -c math.s
    objdump --disassemble math.o
    objdump --syms math.o

    return 0;
}
```
Objdump disassembly

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)

0x00000000 <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

0x00000028 <square>:
  28: 27bdfff8  addiu  sp,sp,-8
  2c: afbe0000  sw  s8,0(sp)
  30: 03a0f021  move  s8,sp
  34: afc40008  sw  a0,8(s8)

symbol

prolog

resolved (fixed) later

body

epilog
### Objdump symbols

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

SYMBOL TABLE:

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Section</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>l</td>
<td>.text</td>
<td>00000000 math.c</td>
</tr>
<tr>
<td>00000000</td>
<td>d</td>
<td>.text</td>
<td>00000000 .text</td>
</tr>
<tr>
<td>00000000</td>
<td>d</td>
<td>.data</td>
<td>00000000 .data</td>
</tr>
<tr>
<td>00000000</td>
<td>d</td>
<td>.bss</td>
<td>00000000 .bss</td>
</tr>
<tr>
<td>00000000</td>
<td>d</td>
<td>.mdebug.abi32</td>
<td>00000000 .mdebug.abi32</td>
</tr>
<tr>
<td>00000008</td>
<td>O</td>
<td>.data</td>
<td>00000004 randomval</td>
</tr>
<tr>
<td>00000060</td>
<td>F</td>
<td>.text</td>
<td>00000028 is_prime</td>
</tr>
<tr>
<td>00000000</td>
<td>d</td>
<td>.rodata</td>
<td>00000000 .rodata</td>
</tr>
<tr>
<td>00000000</td>
<td>d</td>
<td>.comment</td>
<td>00000000 .comment</td>
</tr>
<tr>
<td>00000000</td>
<td>O</td>
<td>.data</td>
<td>00000004 pi</td>
</tr>
<tr>
<td>00000004</td>
<td>O</td>
<td>.data</td>
<td>00000004 e</td>
</tr>
<tr>
<td>00000000</td>
<td>F</td>
<td>.text</td>
<td>00000028 pick_random</td>
</tr>
<tr>
<td>00000028</td>
<td>F</td>
<td>.text</td>
<td>00000038 square</td>
</tr>
<tr>
<td>00000088</td>
<td>F</td>
<td>.text</td>
<td>0000004c pick_prime</td>
</tr>
<tr>
<td>00000000</td>
<td><em>UND</em></td>
<td></td>
<td>00000000 username</td>
</tr>
<tr>
<td>00000000</td>
<td><em>UND</em></td>
<td></td>
<td>00000000 printf</td>
</tr>
</tbody>
</table>
```
```
Objdump symbols

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

SYMBOL TABLE:
<table>
<thead>
<tr>
<th>Address</th>
<th>Segment</th>
<th>Type</th>
<th>Segment Size</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>l</td>
<td>df</td>
<td><em>ABS</em></td>
<td>00000000 math.c</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d</td>
<td>.text</td>
<td>00000000 .text</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d</td>
<td>.data</td>
<td>00000000 .data</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d</td>
<td>.bss</td>
<td>00000000 .bss</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d</td>
<td>.mdebug.abi32</td>
<td>00000000 .mdebug.abi32</td>
</tr>
<tr>
<td>00000008</td>
<td>l</td>
<td>O</td>
<td>.data</td>
<td>000000004 randomval</td>
</tr>
<tr>
<td>00000060</td>
<td>l</td>
<td>F</td>
<td>.text</td>
<td>000000028 is_prime</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d</td>
<td>.rodata</td>
<td>000000000 .rodata</td>
</tr>
<tr>
<td>00000000</td>
<td>l</td>
<td>d</td>
<td>.comment</td>
<td>000000000 .comment</td>
</tr>
<tr>
<td>00000000</td>
<td>g</td>
<td>O</td>
<td>.data</td>
<td>000000004 pi</td>
</tr>
<tr>
<td>00000004</td>
<td>g</td>
<td>O</td>
<td>.data</td>
<td>000000004 e</td>
</tr>
<tr>
<td>00000000</td>
<td>g</td>
<td>F</td>
<td>.text</td>
<td>000000028 pick_random</td>
</tr>
<tr>
<td>00000028</td>
<td>g</td>
<td>F</td>
<td>.text</td>
<td>000000038 square</td>
</tr>
<tr>
<td>00000088</td>
<td>g</td>
<td>F</td>
<td>.text</td>
<td>00000004c pick_prime</td>
</tr>
<tr>
<td>00000000</td>
<td>g</td>
<td>O</td>
<td>.data</td>
<td>000000000 username</td>
</tr>
<tr>
<td>00000000</td>
<td>g</td>
<td>O</td>
<td>.data</td>
<td>000000000 printf</td>
</tr>
</tbody>
</table>
```

Address:
- l: local
- g: global
- f: func
- O: obj
- *UND*: external reference

Segment:
- df: *ABS*
- d: .text
- d: .data
- d: .bss
- d: .mdebug.abi32
- O: .data
- F: .text
- d: .rodata
- d: .comment
- O: .data
- F: .text
- F: .text
- F: .text
- O: .data
- O: .data

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

Function:
- pick_random
- square
- pick_prime
- username
- printf
Q: Why separate compile/assemble and linking steps?
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
Linker Example

**Steps**

1) Find UND symbols in symbol table

2) Relocate segments that collide

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

External references need to be resolved (fixed)
**Linker Example**

- **main.o**
  - 00 T main
  - 00 D pi
  - *UND* printf
  - *UND* pi
  - 40, JL, printf
  - 4C, LW/gp, pi
  - 50, JL, square

- **math.o**
  - 21032040
  - 0C000000
  - 1b301402
  - 3C040000
  - 34040000

- **printf.o**
  - ...}

- **calc.exe**
  - 21032040
  - 0C40023C
  - 1b301402
  - 3C040000
  - 34040004
  - 0C40023C
  - 21035000
  - 1b80050C
  - 8C043004
  - 0C400200

- **Relocation info**
  - LW $4, -32764($gp)
  - $4 = pi

- **Symbol tbl**
  - main
  - unwind
  - printf
  - pi
  - uname

- **Entry**
  - 0040 0100
  - text: 0040 0000
  - data: 1000 0000

- **Linker Example**
  - LA uname
  - LUI 1000
  - ORI 0004
  - JAL printf
  - JAL square

- **Relocation info**
  - LW $4, -32764($gp)
  - $4 = pi
  - JAL square

- **Symbol tbl**
  - main
  - unwind
  - printf
  - pi
  - uname

- **Entry**
  - 0040 0100
  - text: 0040 0000
  - data: 1000 0000

- **Linker Example**
  - LA uname
  - LUI 1000
  - ORI 0004
  - JAL printf
  - JAL square

- **Symbol tbl**
  - main
  - unwind
  - printf
  - pi
  - uname

- **Entry**
  - 0040 0100
  - text: 0040 0000
  - data: 1000 0000

- **Linker Example**
  - LA uname
  - LUI 1000
  - ORI 0004
  - JAL printf
  - JAL square

- **Symbol tbl**
  - main
  - unwind
  - printf
  - pi
  - uname

- **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

calc.exe

Executable program

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!

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, ....
Q: But every program still contains part of library!
## Direct Function Calls

<table>
<thead>
<tr>
<th>Direct call:</th>
<th>Drawbacks:</th>
</tr>
</thead>
<tbody>
<tr>
<td>004000010 &lt;main&gt;:</td>
<td>Linker or loader must edit every use of a symbol (call site, global var use, ...)</td>
</tr>
<tr>
<td>...</td>
<td>Idea:</td>
</tr>
<tr>
<td>jal 0x004000330</td>
<td>Put all symbols in a single “global offset table”</td>
</tr>
<tr>
<td>...</td>
<td>Code does lookup as needed</td>
</tr>
<tr>
<td>jal 0x00400620</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>jal 0x00400330</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>004000330 &lt;printf&gt;:</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>004000620 &lt;gets&gt;:</td>
<td></td>
</tr>
</tbody>
</table>
Indirect Function Calls

Indirect call:

00400010 <main>:
  ...
  jal 0x00400330
  ...
  jal 0x00400620
  ...
  jal 0x00400330
  ...
  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>:

... lw $t9,-32704($gp)
jal $t9
...
```

```
00400620 <gets>:

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

# data segment
GOT: global offset table

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</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>

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

printf = 4+(-32712)+$gp
gets = 8+(-32712)+$gp
Indirect Function Calls

Indirect call:

00400010 <main>:

```
... lw $t9,-32708($gp)      # data segment
     jalr $t9
     ...
lw $t9,-32704($gp)      # global offset table
ejalr $t9
     ...
```

00400330 <printf>:

```
... lw $t9,-32708($gp)      # to be loaded
     jalr $t9
     ...
```

00400620 <gets>:

```
... lw $t9,-32708($gp)      # at -32712($gp)
ejalr $t9
```

# data segment

```
.got
.word  0x00400010 # main
.word  0x00400330 # printf
.word  0x00400620 # gets
```

# global offset table

```
.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>:
    ...
    # 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
```
Big Picture

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

calc.exe

Executing in Memory
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
Upcoming agenda

- PA2 Design Doc due *yesterday*, Monday, March 11\textsuperscript{th}
- HW3 due *this* Wednesday, March 13\textsuperscript{th}
- PA2 Work-in-Progress circuit due *before* spring break

- **Spring break:** Saturday, March 16\textsuperscript{th} to Sunday, March 24\textsuperscript{th}

- Prelim2 Thursday, March 28\textsuperscript{th}, right after spring break

- PA2 due Thursday, April 4\textsuperscript{th}
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