Assemblers and Linkers

Goals for this Lecture

- Help you to learn about:
  - IA-32 machine language
  - The assembly and linking processes

Why Learn Machine Language

- Machine language is the last stop on the “language levels” tour
- A power programmer knows about the relationship between assembly language and machine language
- A systems programmer knows how an assembler translates assembly language to machine language
Part 1: Machine Language

IA-32 Machine Language

- IA-32 machine language
  - Difficult to generalize about IA-32 instruction format
  - Many (most!) instructions are exceptions to the rules
  - Generally, instructions use the following format shown in following slides
- We'll go over
  - The format of instructions
  - Two example instructions
- Just to give a sense of how it works…

IA-32 Instruction Format

- Instruction prefixes
  - Up to 4 prefixes of 1 byte each (optional)
- Opcode
  - 1, 2, or 3 bytes of required
- ModR/M
  - 1 byte of required
- SIB
  - 0, 1, 2, or 4 bytes
- Displacement
  - 0, 1, 2, or 4 bytes
- Immediate
  - 0, 1, 2, or 4 bytes

Instruction prefix
- Sometimes a repeat count
- Rarely used; don’t be concerned
IA-32 Instruction Format (cont.)

Instruction prefixes
Up to 4 prefixes of 1 byte each (optional)

Opcode
- Specifies which operation should be performed
- Add, move, call, etc.

ModR/M
- Specifies types of operands (immediate, register, memory)
- Specifies sizes of operands (byte, word, long)
- Sometimes denotes a register:
  000 = EAX/AL; 011 = EBX/BL; 001 = ECX/CL; 010 = EDX/DL;
  110 = ESI/DH; 111 = EDI/BH; 101 = EBP/CH; 110 = ESP/AH
- Sometimes contains an extension of the opcode

SIB
- Used when one of the operands is a memory operand
  that uses a scale, an index register, and/or a base register
IA-32 Instruction Format (cont.)

- Instruction prefixes: Up to 4 prefixes of 1 byte each (optional)
- Opcode: 1, 2, or 3 byte opcode
- ModR/M: 1 byte (if required)
- SIB: 0, 1, 2, or 4 bytes
- Displacement: 0, 1, 2, or 4 bytes
- Immediate: 0, 1, 2, or 4 bytes

Displacement
- Used in jump and call instructions
- Indicates the displacement between the destination instruction and the jump/call instruction
- More precisely, indicates:
  \[ \text{[addr of destination instr]} - \text{[addr of instr following the jump/call]} \]
- Uses little-endian byte order

Immediate
- Specifies an immediate operand
- Uses little-endian byte order

Example: Push on to Stack

- Assembly language:
  ```assembly
  pushl %edx
  ```

- Machine code:
  - IA32 has a separate opcode for push for each register operand
    - 50: pushl %eax
    - 51: pushl %ecx
    - 52: pushl %edx
    - ... 
  - Results in a one-byte instruction

- Observe: sometimes one assembly language instruction can map to a group of different opcodes
Example: Load Effective Address

- **Assembly language:**
  
  \[
  \text{leal} \left(\%eax,\%eax,4\right), \%eax
  \]

- **Machine code:**
  
  - Byte 1: 8D (opcode for “load effective address”)  \[\text{1000 1101}\]
  
  - Byte 2: 04 (dest \%eax, with scale-index-base)  \[\text{000 0100}\]
  
  - Byte 3: 80 (scale=4, index=%eax, base=%eax)  \[\text{000 0000}\]

  Load the address \%eax + 4 * \%eax into register \%eax

CISC and RISC

- **IA-32** machine language instructions are complex

- **IA-32** is a
  
  - Complex Instruction Set Computer (CISC)

- **Alternative**:
  
  - Reduced Instruction Set Computer (RISC)

Characteristics of CISC and RISC

- **CISC**
  
  - Many instructions
  
  - Many addressing modes (direct, indirect, indexed, base-pointer)
  
  - Hardware interpretation is complex
  
  - Few instructions required to accomplish a given job (expressive)
  
  - Example: IA-32

- **RISC**
  
  - Few instructions
  
  - Few addressing modes (typically only direct and indirect)
  
  - Hardware interpretation is simple
  
  - Many instructions required to accomplish a given job (not expressive)
  
  - Examples: MIPS, SPARC
**Brief History of CISC and RISC**

- **Stage 1:** Programmers write assembly language
  - Important that assembly/machine language be expressive
  - CISC dominates (esp. Intel)

- **Stage 2:** Programmers write high-level language
  - Not important that assembly/machine language be expressive; the compiler generates it
  - Important that compilers work well \( \rightarrow \) assembly/machine language should be simple
  - RISC takes a foothold (but CISC, esp. Intel, persists)

- **Stage 3:** Compilers get smarter
  - Less important that assembly/machine language be simple
  - Much motivation for RISC disappears
  - CISC (esp. Intel) dominates the computing world

---

**Part 2: The Assembly Process**

**The Build/Execute Process**

- **myprog.c**
- **myprog.s**
- **myprog.o**
- **libraries**

Covered in COS 320: Compiling Techniques

Covered here
Two Aspects of the Assembler/Linker

- Translating each instruction
  - Mapping an assembly-language instruction
  - ... into the corresponding machine-language instruction
- Dealing with references across instructions
  - Jumps to other locations in same chunk of code
  - Accesses a global variable by the name of its memory location
  - Calling to and returning from functions defined in other code

```c
main:
pushl %ebp
movl %esp, %ebp
call getchar
cmpl $'A', %eax
jne skip
pushl $msg
call printf
addl $4, %esp
skip:
movl $0, %eax
movl %ebp, %esp
popl %ebp
ret
```

References Across Instructions

- Many instructions can be assembled independently
  - pushl %edx
  - leal (%eax, %eax, 4), %eax
  - movl $0, %eax
  - addl %ebx, %ecx
- But, some make references to other data or code
  - jne skip
  - pushl $msg
  - call printf
- Need to fill in those references
  - To generate a final executable binary

```c
main:
pushl %ebp
movl %esp, %ebp
call getchar
cmpl $'A', %eax
jne skip
pushl $msg
call printf
addl $4, %esp
skip:
movl $0, %eax
movl %ebp, %esp
popl %ebp
ret
```

The Forward Reference Problem

- Problem
  - Any assembler must deal with the forward reference problem

```c
jmp mylabel
    mylabel:...
```

- Assembler must generate machine language code for "jmp mylabel"
  - But assembler hasn’t yet seen the definition of mylabel
  - i.e., the jmp instruction contains a forward reference to mylabel
The Forward Reference Solution

• Solution
  • Assembler performs 2 passes over assembly language program

• Different assemblers perform different tasks in each pass

• One straightforward design...

Assembler Passes

• Pass 1
  • Assembler traverses assembly program to create...
  • Symbol table
    • Key: label
    • Value: information about label
      • Label name, which section, what offset within that section, ...

• Pass 2
  • Assembler traverses assembly program again to create...
    • RODATA section
    • DATA section
    • BSS section
    • TEXT section
    • Relocation record section
      • Each relocation record indicates an area that the linker must patch

An Example Program

• A simple (nonsensical) program:

```c
#include <stdio.h>
int main(void) {
  if (getchar() == 'A')
    printf("Hi\n");
  return 0;
}
```

• Let's consider how the assembler handles that program...

```assembly
.include "stdio.h"
main:
  pushl temp
  movl temp, temp
  call getchar
  cmp $'A', %eax
  je skip
  pushl $msg
  call printf
  addl $4, %esp

skip:
  movl $0, %eax
  popl %ebp
  ret
```
Assembler Data Structures (1)

- Symbol Table

<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local?</th>
<th>Seq#</th>
</tr>
</thead>
</table>

- Relocation Records

<table>
<thead>
<tr>
<th>Section</th>
<th>Offset</th>
<th>Rel Type</th>
<th>Seq#</th>
</tr>
</thead>
</table>

- RODATA Section (location counter: 0)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
</table>

- TEXT Section (location counter: 0)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
</table>

- No DATA or BSS section in this program
- Initially all sections are empty

Assembler Pass 1

```
section .rodata
.msg: .asciz  "Hi"

section .text
.globl  main
main:
pushl   %ebp
movl    %esp, %ebp
call    getchar
cmpl    $'A', %eax
jne     skip
pushl   $msg
call    printf
addl    $4, %esp
skip:
movl    $0, %eax
movl    %ebp, %esp
popl    %ebp
ret
```

Assembler adds binding to Symbol Table…

Assembler notes that the current section is RODATA

Assembler adds binding to Symbol…

Assembler Data Structures (2)

- Symbol Table

<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local?</th>
<th>Seq#</th>
</tr>
</thead>
</table>

- RODATA Section (location counter: 0)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
</table>

- TEXT Section (location counter: 0)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
</table>

- msg marks a spot in the RODATA section at offset 0
- msg is a local label
- Assign msg sequence number 0
Assembler Pass 1 (cont.)

```asm
section ".rodata"
msg: .asciz  "Hi"

section ".text"
.globl main
main:
pushl %ebp
movl %esp, %ebp
call getchar
cmpl $'A', %eax
jne     skip
pushl $msg
call printf
addl $4, %esp
skip:
movl $0, %eax
movl %ebp, %esp
popl %ebp
ret
```

Assembler increments RODATA section location counter by byte count of the string (4)...

Assembler Data Structures (3)

- **Symbol Table**
  - Label | Section | Offset | Local? | Seq#
  - `msg` | RODATA | 0 | local | 0

- **Relocation Records**
  - `(Same)`
  - RODATA Section (location counter: 0)
  - `(Same)`
  - TEXT Section (location counter: 0)
  - `(Same)`

- **RODATA location counter now is 4**
- If another label were defined at this point, it would mark a spot in RODATA at offset 4

Assembler Pass 1 (cont.)

Assembler notes that current section is TEXT
Assembler adds binding to Symbol Table...
Assembler does nothing
Assembler Data Structures (4)

- Symbol Table

<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>RODATA</td>
<td>0</td>
<td>local</td>
<td>0</td>
</tr>
<tr>
<td>main</td>
<td>TEXT</td>
<td>0</td>
<td>local</td>
<td>1</td>
</tr>
</tbody>
</table>

- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 0)
  - (Same)

- main marks a spot in the TEXT section at offset 0
- main is a local label (assembler will discover otherwise in Pass 2)
- Assign main sequence number 1

Assembler Pass 1 (cont.)

```
.rodata
msg: .asciz  "Hi"

.text
.globl  main
main:
pushl   %ebp
movl    %esp, %ebp
call    getchar
cmpl    $'A', %eax
jne     skip
pushl   $msg
call    printf
addl    $4, %esp
skip:
movl    $0, %eax
movl    %ebp, %esp
popl    %ebp
ret
```

- Assembler increments TEXT section location counter by the length of each instruction...

Assembler Data Structures (5)

- Symbol Table

<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>RODATA</td>
<td>0</td>
<td>local</td>
<td>0</td>
</tr>
<tr>
<td>main</td>
<td>TEXT</td>
<td>0</td>
<td>local</td>
<td>1</td>
</tr>
</tbody>
</table>

- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 6)
  - (Same)

- TEXT location counter now is 26
- If another label were defined at this point, it would mark a spot in TEXT at offset 26
Assembler Data Structures (6)

- Symbol Table
  - Label | Section | Offset | Local? | Seq#
  - msg   | .rodata | 0      | local  | 0
  - main  | .text   | 0      | local  | 1
  - skip  | .text   | 26     | local  | 2
  
- Relocation Records
  - (Same)
  - .rodata Section (location counter: 4)
  - (Same)
  - .text Section (location counter: 26)
  - (Same)

Assembler Pass 1 (cont.)

```assembly
.section .rodata
msg:
  .asciz "Hi"

.section .text
.globl main
main:
  pushl %ebp
  movl %esp, %ebp
  call getchar
  cmpl $'A', %eax
  jne skip
  pushl $msg
  call printf
  addl $4, %esp
skip:
  movl $0, %eax
  movl %ebp, %esp
  popl %ebp
  ret
```

Assembler increments TEXT section location counter by the length of each instruction.

Assembler adds binding to Symbol Table.
Assembler Data Structures (7)

- Symbol Table
  - Label | Section | Offset | Local? | Seq#
  - msg | RODATA | 0 | local | 0
  - main | TEXT | 0 | local | 1
  - skip | TEXT | 26 | local | 2

- Relocation Records
  - (Same)
  - RODATA Section (location counter: 4)
    - (Same)
  - TEXT Section (location counter: 35)
    - (Same)

- TEXT location counter now is 35
- If another label were defined at this point, it would mark a spot in TEXT at offset 35

From Assembler Pass 1 to Pass 2

- End of Pass 1
  - Assembler has (partially) created Symbol Table
  - So assembler now knows which location each label marks

- Beginning of Pass 2
  - Assembler resets all section location counters…

Assembler Data Structures (8)

- Symbol Table
  - Label | Section | Offset | Local? | Seq#
  - msg | RODATA | 0 | local | 0
  - main | TEXT | 0 | local | 1
  - skip | TEXT | 26 | local | 2

- Relocation Records
  - (Same)
  - RODATA Section (location counter: 0)
    - (Same)
  - TEXT Section (location counter: 0)
    - (Same)

- Location counters reset to 0
Assembler Pass 2

```
Assembler notes that the current section is RODATA
Assembler does nothing
Assembler places bytes in RODATA section, and increments location counter...
```

Assembler Data Structures (9)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - Location counter incremented to 4
  - RODATA section contains the bytes comprising the string
- TEXT Section (location counter: 6)
  - (Same)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents (hex)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
<td>ASCII code for 'H'</td>
</tr>
<tr>
<td>1</td>
<td>69</td>
<td>ASCII code for 'i'</td>
</tr>
</tbody>
</table>
| 2      | 0A             | ASCII code for '
'  |
| 3      | 00             | ASCII code for null char |

Assembler Pass 2 (cont.)

```
Assembler notes that the current section is TEXT
Assembler updates Symbol Table...
```
Assembler Data Structures (10)

- Symbol Table
  - | Label | Section | Offset | Local | Seq#
  - | msg | .rodata | 0 | local | 0
  - | main | .text | 0 | global | 1
  - | skip | .text | 26 | local | 2

- Relocation Records
  - (Same)
- .rodata Section (location counter: 4)
  - (Same)
- .text Section (location counter: 0)
  - (Same)

Label | Section | Offset | Local | Seq#
---|--------|--------|-------|-----
msg | .rodata | 0 | local | 0
main | .text | 0 | global | 1
skip | .text | 26 | local | 2

- main is a global label

Assembler Pass 2 (cont.)

- .section "rodata"
  - .ascii "Hi"
  - .section "text"
    - .globl main
    - main:
      - pushl %ebp
      - movl %esp, %ebp
      - call getchar
      - cmpb $'A', %eax
      - jne skip
      - pushl $msg
      - call printf
      - addl $4, %esp
    - skip:
      - movl $0, %eax
      - movl %ebp, %esp
      - pushl %ebp
      - ret

Assembler does nothing
Assembler generates machine language code in current (TEXT) section...

Assembler Data Structures (11)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- .rodata Section (location counter: 4)
  - (Same)
- .text Section (location counter: 1)

Offset | Contents | Explanation
---|---------|-------------
0 | pushl %ebp | push %ebp
00010000 | pushl %ebp | push %ebp
Assembler Pass 2 (cont.)

```
Assembler generates machine language code in current (TEXT) section...
```

Assembler Data Structures (12)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 3)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>89 E5</td>
<td>movl %esp,%ebp</td>
</tr>
<tr>
<td>10001001 11 100 101</td>
<td>This is a &quot;movl&quot; instruction whose source operand</td>
<td>The M field designates a register. The source register is ESP. The destination register is EBP.</td>
</tr>
</tbody>
</table>
Assembler Data Structures (12)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 8)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-7</td>
<td>E8 ????????</td>
<td>call getchar</td>
</tr>
</tbody>
</table>

This is a "call" instruction with a 4-byte immediate operand. This is a "call" instruction with a 4-byte immediate operand. This is a "call" instruction with a 4-byte immediate operand. This is a "call" instruction with a 4-byte immediate operand. This is a "call" instruction with a 4-byte immediate operand. This is a "call" instruction with a 4-byte immediate operand. This is a "call" instruction with a 4-byte immediate operand. This is a "call" instruction with a 4-byte immediate operand.

Assembler looks in Symbol Table to find offset of getchar. getchar is not in Symbol Table. Assembler cannot compute displacement that belongs at offset 4. So...

Assembler Data Structures (13)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 8)
  - (Same)

- Assembler adds getchar to Symbol Table.

- Then...

Assembler Data Structures (14)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 8)
  - (Same)

- Assembler generates a relocation record, thus asking linker to patch code.

Dear Linker,
Please patch the TEXT section at offset 4. Do a "displacement" type of patch. The patch is with respect to the label whose seq number is 3 (i.e. getchar).

Sincerely,
Assembler
Assembler Pass 2 (cont.)

```
section '.rodata'
msg:
.asciz  "Hi

section '.text'
globl  main
main:
pushl  %ebp
movl    %esp, %ebp
call    getchar
cmpl    $'A', %eax
jne     skip
pushl   $msg
call    printf
addl    $4, %esp
skip:
movl    $0, %eax
movl    %ebp, %esp
popl    %ebp
ret
```

Assembler generates machine language code in current (TEXT) section...

Assembler Data Structures (15)

- Symbol Table
- (Same)
- Relocation Records
- (Same)
- RDATA Section (location counter: 4)
- (Same)
- TEXT Section (location counter: 11)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-10</td>
<td>83 F8 41</td>
<td>cmpl %'A',%eax</td>
</tr>
</tbody>
</table>

This is some "l" instruction that has a 1 byte immediate operand
The M field designates a register
This is a "cmp" instruction
The destination register is EAX
The immediate operand is 'A'

Assembler Pass 2 (cont.)

```
section '.rodata'
msg:
.asciz  "Hi

section '.text'
globl  main
main:
pushl  %ebp
movl    %esp, %ebp
call    getchar
cmpl    $'A', %eax
jne     skip
pushl   $msg
call    printf
addl    $4, %esp
skip:
movl    $0, %eax
movl    %ebp, %esp
popl    %ebp
ret
```

Assembler generates machine language code in current (TEXT) section...
Assembler Data Structures (16)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 13)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>11-12</td>
<td>75 0D</td>
<td>This is a jne instruction that has a 1 byte immediate operand. The displacement between the destination instr. and the next instr. is 13.</td>
</tr>
</tbody>
</table>

Assembler Pass 2 (cont.)

```
section "rodata"
msg: .asciz "Hi"

section "text"
globl main
main:
pushl %ebp
movl %esp, %ebp
call getchar
cmpl $'A', %eax
jne skip
pushl $msg
call printf
addl $4, %esp
skip:
movl $0, %eax
movl %ebp, %esp
popl %ebp
ret
```

Assembler Data Structures (16)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 13)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>13-17</td>
<td>68 ???????</td>
<td>This is a pushl instruction with a 4 byte immediate operand. This is the data to be pushed.</td>
</tr>
</tbody>
</table>

Assembler knows offset of msg (0) within RODATA section.
But assembler does not know location RODATA section.
So assembler does not know location of msg.
So...
Dear Linker,

Please patch the TEXT section at offset 14. Do an "absolute" type of patch. The patch is with respect to the label whose seq number is 0 (i.e. msg).

Sincerely,
Assembler

---

Assembler Pass 2 (cont.)

```assembly
section "rodata"
msg: .asciz  "Hi\n"

section "text"
.globl  main
main:
    pushl   %ebp
    movl    %esp, %ebp
    call    getchar
    cmpl    $'A', %eax
    jne     skip
    pushl   $msg
    call    printf
    addl    $4, %esp
skip:
    movl    $0, %eax
    movl    %ebp, %esp
    popl    %ebp
    ret
```

Assembler generates machine language code in current (TEXT) section...

---

Assembler Data Structures (17)

- Symbol Table
  - (Same)
- Relocation Records
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 18)
  - (Same)

<table>
<thead>
<tr>
<th>Section</th>
<th>Offset</th>
<th>Rel Type</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXT</td>
<td>14</td>
<td>absolute</td>
<td></td>
</tr>
</tbody>
</table>

Assembler generates a relocation record, thus asking linker to patch code.

---

Assembler Data Structures (18)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 23)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>18-22</td>
<td>E8 ???????</td>
<td>call printf</td>
</tr>
<tr>
<td>11101000</td>
<td>???????????????????????????????</td>
<td>This is a &quot;call&quot; instruction with a 4-byte immediate operand</td>
</tr>
<tr>
<td></td>
<td>This the displacement</td>
<td></td>
</tr>
</tbody>
</table>

Assembler looks in Symbol Table to find offset of printf: printf is not in Symbol Table. Assembler cannot compute displacement that belongs at offset 19. So...

---
Assembler Data Structures (19)

- Symbol Table
<table>
<thead>
<tr>
<th>Label</th>
<th>Section</th>
<th>Offset</th>
<th>Local?</th>
<th>Seq</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>RODATA</td>
<td>0</td>
<td>local</td>
<td>0</td>
</tr>
<tr>
<td>main</td>
<td>TEXT</td>
<td>0</td>
<td>global</td>
<td>1</td>
</tr>
<tr>
<td>skip</td>
<td>TEXT</td>
<td>20</td>
<td>local</td>
<td>2</td>
</tr>
<tr>
<td>printf</td>
<td>TEXT</td>
<td>26</td>
<td>global</td>
<td>3</td>
</tr>
<tr>
<td>main</td>
<td>TEXT</td>
<td>30</td>
<td>global</td>
<td>4</td>
</tr>
</tbody>
</table>

- Relocation Records
  - (Same)
  - RODATA Section (location counter: 4)
    - (Same)
  - TEXT Section (location counter: 23)
    - (Same)

Assembler adds printf to Symbol Table
Then...

Assembler Data Structures (20)

- Symbol Table
  - (Same)
- Relocation Records
  - (Same)
- RODATA Section (location counter: 4)
  - (Same)
- TEXT Section (location counter: 8)
  - (Same)

Assembler generates a relocation record, thus asking linker to patch code

Dear Linker,
Please patch the TEXT section at offset 19. Do a "displacement" type of patch. The patch is with respect to the label whose seq number is 4 (i.e. printf).
Sincerely,
Assembler

Assembler Pass 2 (cont.)

- .section ".rodata"
- .asciz "Hi"
- .section ".text"
- .globl main
- pushl $ebp
  movl $esp, %ebp
- call getchar
  cmpl $'A', %eax
- jne skip
  pushl $msg
  call printf
  addl $4, %esp
- skip:
- movl $0, %eax
- movl %eax, $temp
- pushl $temp
- call printf
  movl %ebp, %esp
  popl %ebp
  ret

Assembler generates machine language code in current (TEXT) section...

Assembler ignores
Assembler Data Structures (21)

- Symbol Table, Relocation Records, RODATA Section
- TEXT Section (location counter: 31)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>23-25</td>
<td>83 C4 04</td>
<td>addl $4,%esp</td>
</tr>
<tr>
<td>10000011 11 000 100 00000100</td>
<td>This is some &quot;l&quot; instruction that has a 1 byte immediate operand</td>
<td></td>
</tr>
<tr>
<td>The M field designates a register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>This is an &quot;add&quot; instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The destination register is ESP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The immediate operand is $4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>26-30</td>
<td>B8 00000000</td>
<td>movl $0,%eax</td>
</tr>
<tr>
<td>10111000 00000000000000000000000000000000</td>
<td>This is an instruction of the form &quot;movl 4-byte immediate, %eax&quot;</td>
<td></td>
</tr>
<tr>
<td>The immediate operand is $0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assembler Data Structures (22)

- Symbol Table, Relocation Records, RODATA Section
- TEXT Section (location counter: 35)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-32</td>
<td>89 EC</td>
<td>movl %ebp,%esp</td>
</tr>
<tr>
<td>10001001 11 101 100</td>
<td>This is a &quot;movl&quot; instruction whose source operand is a register</td>
<td></td>
</tr>
<tr>
<td>The M field designates a register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The source register is EBP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The destination register is ESP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>5D</td>
<td>popl %ebp</td>
</tr>
<tr>
<td>01011101</td>
<td>This is a &quot;popl %ebp&quot; instruction</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>C3</td>
<td>ret</td>
</tr>
<tr>
<td>11000011</td>
<td>This is a &quot;ret&quot; instruction</td>
<td></td>
</tr>
</tbody>
</table>

From Assembler to Linker

- Assembler writes its data structures to .o file
- Linker:
  - Reads .o file
  - Works in two phases: resolution and relocation
Linker Resolution

• Resolution
  • Linker resolves references
  • For this program, linker:
    • Notes that Symbol Table contains undefined labels
    • getchar and printf
    • Fetches, from libc.a, machine language code defining getchar and printf
    • Adds that code to TEXT section
    • (May add code to other sections too)
    • Updates Symbol Table to note offsets of getchar and printf
    • Adds column to Symbol Table to note addresses of all labels

Linker Relocation

• Relocation
  • Linker patches (“relocates”) code
  • Linker traverses relocation records, patching code as specified
  • For this program

<table>
<thead>
<tr>
<th>Section</th>
<th>Offset</th>
<th>Rel Type</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXT</td>
<td>4</td>
<td>displacement</td>
<td>3</td>
</tr>
<tr>
<td>TEXT</td>
<td>14</td>
<td>absolute</td>
<td>0</td>
</tr>
<tr>
<td>TEXT</td>
<td>19</td>
<td>displacement</td>
<td>4</td>
</tr>
</tbody>
</table>

• Linker looks up offset of getchar
• Linker computes:
  [offset of getchar] – 8
• Linker places difference in TEXT section at offset 4

Linker Relocation (cont.)

• For this program

<table>
<thead>
<tr>
<th>Section</th>
<th>Offset</th>
<th>Rel Type</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXT</td>
<td>4</td>
<td>displacement</td>
<td>3</td>
</tr>
<tr>
<td>TEXT</td>
<td>14</td>
<td>absolute</td>
<td>0</td>
</tr>
<tr>
<td>TEXT</td>
<td>19</td>
<td>displacement</td>
<td>4</td>
</tr>
</tbody>
</table>

• Linker looks up addr of msg
• Linker places addr in TEXT section at offset 12
### Linker Relocation (cont.)

- For this program

<table>
<thead>
<tr>
<th>Section</th>
<th>Offset</th>
<th>Rel Type</th>
<th>Seq#</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXT</td>
<td>4</td>
<td>displacement</td>
<td>2</td>
</tr>
<tr>
<td>TEXT</td>
<td>14</td>
<td>absolute</td>
<td>0</td>
</tr>
<tr>
<td>TEXT</td>
<td>19</td>
<td>displacement</td>
<td>4</td>
</tr>
</tbody>
</table>

- Linker looks up offset of printf
- Linker computes:
  
  (offset of printf) \(-\) 23
- Linker places difference in TEXT section at offset 19

### Linker Finishes

- Linker writes resulting TEXT, RDATA, DATA, BSS sections to executable binary file

### ELF: Executable and Linking Format

- Unix format of object and executable files
  - Output by the assembler
  - Input and output of linker

<table>
<thead>
<tr>
<th>optional for .o files</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF Header</td>
</tr>
<tr>
<td>Program Hdr Table</td>
</tr>
<tr>
<td>Section 1</td>
</tr>
<tr>
<td>Section n</td>
</tr>
<tr>
<td>optional for a.out files</td>
</tr>
<tr>
<td>Section Hdr Table</td>
</tr>
</tbody>
</table>

---

24
Conclusions

- **Assembler:** reads assembly language file
  - Pass 1: Generates Symbol Table
  - Contains info about labels
  - Pass 2: Uses Symbol Table to generate code
  - TEXT, RODATA, DATA, BSS sections
  - Relocation Records
  - Writes object file (ELF)

- **Linker:** reads object files
  - Resolution: Resolves references to make Symbol Table complete
  - Relocation: Uses Symbol Table and Relocation Records to patch code
  - Writes executable binary file (ELF)

Appendix: Generating Machine Lang

- **Hint for Buffer Overrun assignment...**

- Given an assembly language instruction, how can you find the machine language equivalent?

- **Option 1:** Consult IA-32 reference manuals
  - See course Web pages for links to the manuals

- **Option 2:**
  - Compose an assembly language program that contains the given assembly language instruction
  - Then use gdb...
Appendix: Generating Machine Lang

• Using gdb

```bash
$ gcc detecta.s –o detecta
$ gdb detecta
(gdb) x/12i main
0x80483b4 <main>:       push   %ebp
0x80483b5 <main+1>:     mov    %esp,%ebp
0x80483b7 <main+3>:     call   0x8048298 <getchar@plt>
0x80483bc <main+8>:     cmp    $0x41,%eax
0x80483bf <main+11>:    jne    0x80483ce <skip>
0x80483c1 <main+13>:    push   $0x80484b0
0x80483c6 <main+18>:    call   0x80482c8 <printf@plt>
0x80483cb <main+23>:    add    $0x4,%esp
0x80483ce <skip>:       mov    $0x0,%eax
0x80483d3 <skip+5>:     mov    %ebp,%esp
0x80483d5 <skip+7>:     pop    %ebp
0x80483d6 <skip+8>:     ret
```

Build program; run gdb from shell

Issue `x/i` command to examine memory as instructions

Issue `x/b` command to examine memory as raw bytes

Appendix: Generating Machine Lang

• Option 3:
  - Compose an assembly language program that contains the given assembly language instruction
  - Then use objdump – a special purpose tool...

Appendix: Generating Machine Lang

• Using objdump

```bash
$ gcc detecta.s –o detecta
$ objdump –d detecta
```

Disassembly of section .text:

```
080483b4 <main>:
  80483b4:       55                      push   %ebp
  80483b5:       89 e5                   mov    %esp,%ebp
  80483b7:       e8 dc fe ff ff          call   8048298 <getchar@plt>
  80483bc:       83 f8 41                cmp    $0x41,%eax
  80483bf:       75 0d                   jne    80483ce <skip>
  80483c1:       68 b0 84 04 08          push   $0x80484b0
  80483c6:       e8 fd fe ff ff          call   80482c8 <printf@plt>
  80483cb:       83 c4 04                add    $0x4,%esp
  80483ce <skip>:
  80483ce:       b8 00 00 00 00          mov    $0x0,%eax
  80483d3:       89 ec                   mov    %ebp,%esp
  80483d5:       5d                      pop    %ebp
  80483d6:       c3                      ret
```

Build program; run objdump

Machine language
Assembly language