IA-32 Assembly Programming and Tools

Wei Wang
Why Learn Assembly Programming and their tools?

- Viruses are written in assembly language
  - You should be able to read assembly codes
  - The best way to understand assembly code is to write your own

- The tools are crucial for
  - Programming and compiling
  - Debugging
  - Analyze malicious code
Road Map

- x86 assembly programming
  - One example
  - Compilation
  - Summary of main points
- Assembly Tools
  - Disassembler
  - Compiler
  - Debugger
- Reference documents
- Summary
x86 Assembly Programming

• You almost know everything already
  – Essentially instructions
  – Intel or ATT syntax
  – Calling convention

• The other things you need:
  – Data section
  – Text section
  – Labels
x86 Assembly Programming: Learn From Example

• An example – write the following function in assembly language
  – `compute_print(int a, int b)`
  – Compute $a^2 - b^2$, and output the result with printf

• Two instructions to use
  – `mull src; %eax = %eax * src`
  – `subl src, dest; dest = dest – src`

• Let's write it (in AT&T syntax)
Data Section: x86 Data Declarations

- Global variables are declared in data section
- Give name, type, optional initialization:

```
.section .data
.globl d
d:
    .long 10 ; int d = 10
.globl s
s:
    .string "hello %d \n" ; new string
```
Data Section: x86 Data Declarations

• Accessing global variables:
  – If value is used, put the name of the variable directly
    • E.g., push d
  – If address is used, put the $ before the variable name
    • E.g., push $s
The Text Section: Where Codes are Written

- The text section is for assembly statements:
  - Comments start with “#”

- Label your function and assembly code

```assembly
.extern printf
.text
.globl main
main:
  push d
  push $s
  call printf  # printf(s, d)
.L0:
  add $8, %esp
  jmp .L0
```
Compiling x86 Assembly Codes

• Put your source code in a “.s” file
  – It should be a pure text file

• Compile with GCC
  - gcc  -m32  -c your_source.s  -o your_object.o
  - gcc  -m32 your_object.o  -o your_exec

• You should know about object files and executable files
Road Map

- x86 assembly programming
  - One example
  - Compiling
  - Summary of main points

- Assembly Tools
  - Disassembler
  - Compiler
  - Debugger

- Reference documents

- Summary
To Convert C Code to Assembly code

- GCC:
  - gcc -m32 -S -c c_source_file.c -o assem_output.s
Disassembler: Disassemble a Binary or Object File

- On Linux: objdump (dumpbin on Windows)
  - To show text section:
    `objdump -d your_binary`

<table>
<thead>
<tr>
<th>Instruction Addresses</th>
<th>Binary Instructions</th>
<th>Assembly Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0804848c &lt;compute_print&gt;:</td>
<td>804848c: 55 push %ebp</td>
<td>push %ebp</td>
</tr>
<tr>
<td></td>
<td>804848d: 89 e5 mov %esp,%ebp</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td></td>
<td>804848f: 83 ec 08 sub $0x8,%esp</td>
<td>sub $0x8,%esp</td>
</tr>
</tbody>
</table>
Inspect Execution or Debugging: GDB

• Compile with -g option:
  - `gcc -m32 -g your_source.c -o your_exec`
  - Only if you need to check C source code, or check functions and variables by name

• Start debugging
  - Debug a binary: `gdb your_exec`
  - Debug a running process: `gdb -p pid`

• Quit GDB
  - `quit`
Inspect Execution or Debugging: GDB for C Programs

- Set break points
  - `break function_name`

- Run and stop program
  - Start execution: `run params`
  - Stop execution: `kill`

- Execute next C statement
  - Next C statement: `next` or `n`
  - Step into a function: `step` or `s`
Inspect Execution or Debugging: GDB for C Programs cont'd

• Show variable value:
  - `print var or p var`

• Show register value:
  - `print $reg or p $reg`
  - E.g., `print $eax`

• View current functions
  - `backtrace or bt`
Inspect Execution or Debugging: GDB for Assembly Programs

• No need to compile with -g
• Start debugging like when you're debugging C programs
• Run and stop like when you're debugging C programs
• Set break points:
  - break *address
  - E.g., break *0x0804848c
Inspect Execution or Debugging: GDB for Assembly Programs cont'd

• Tell GDB to show assembly code
  – set disassemble-next-line on

• Execute next assembly statement:
  – Next instruction: nexti or ni
  – Step into a call instruction: steipi or si

• Show assembly code:
  – x/ni address or function or pc (program counter)
  – n is the number of instructions to display
  – e.g., x/5i 0x08048430
  – e.g., x/8i compute_print
  – e.g., x/12i $pc # this command shows next instruction to execution
Inspect Execution or Debugging: GDB for Assembly Programs cont'd

- Print registers values like you are debugging C programs
- Examine memory:
  - `x/nx address or $reg`
  - e.g. `x/4x $esp`
  - e.g. `x/8x $ebp+8`
Reference Documents

- A Tiny Guide to Programming in 32-bit x86 Assembly Language
  - For general ideas
  - The Intel syntax won’t work with gcc
- From C to Assembly Language
  - For GCC assembly formats
  - [http://tldp.org/LDP/LGNET/94/ramankutty.html](http://tldp.org/LDP/LGNET/94/ramankutty.html)
- Use GCC’s assembly output functionality to learn assembly programming
Reference Documents

- Manual pages for objdump and gcc
  - `man objdump`
  - `man gcc`

- GDB documentation – “Debugging with GDB)”
  - Ch 6. Examining the stack
  - Ch 8.5 Examining Memory
  - Ch 9.6 Source and Machine Code

- GDB quick doc – “RMS’s gdb Debugger Tutorial”
Summary

- Assembly Programming
  - data section
  - text sections
  - Labels
- Tools for examining binaries and executions
  - Disassembler: objdump
  - Translate C to assembly: GCC
  - Inspect execution & debug: GDB