Lecture 25: Assembly

CSE 374: Intermediate Programming Concepts and Tools

Lecture Participation Poll #25

Log onto pollev.com/cse374
Or
Text CSE374 to 22333
Administrivia

- HW 4 posted -> Extra credit due date Thursday Dec 3rd
- HW 5 (final HW) coming later today
- HW 6 extra credit releasing next week
- 2 more exercises coming – 1 later today, 1 next week
- Final review assignment will release last week of quarter
- End of quarter due date Wednesday December 16th @ 9pm

THANK YOU FOR YOUR PATIENCE
Review: General Memory Layout

- **Stack**
  - Local variables (procedure context)

- **Heap**
  - Dynamically allocated as needed
  - malloc(), calloc(), new, ...

- **Statically allocated Data**
  - Read/write: global variables (Static Data)
  - Read-only: string literals (Literals)

- **Code/Instructions**
  - Executable machine instructions
  - Read-only
Where does everything go?

```c
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */

int global = 0;

int useless() { return 0; }

int main()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8); /* 256 B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8); /* 256 B */
    /* Some print statements ... */
}
```
Hardware Software Interface

**Source code**
- Different applications or algorithms

- **C Language**
  - Program A
  - Program B
  - *Your program*

**Compiler**
- Perform optimizations, generate instructions

- GCC
- Clang

**Architecture**
- Instruction set

- x86-64

**Hardware**
- Different implementations

- Intel Pentium 4
- Intel Core 2
- Intel Core i7
- **AMD Opteron**
- **AMD Athlon**

- ARMv8 (AArch64/A64)

- ARM Cortex-A53
- Apple A7
From Human to Computer

- C /C++ is translated directly into assembly by compiler
  - Other languages may be translated into another form
  - Java is translated into an assembly-like form, which is then run by the Java interpreter/runtime
  - The Java runtime is executing assembly instructions!
  - Some languages are directly interpreted without being translated into another form
    - Most Bash implementations will directly interpret the commands without compiling
    - Python can do either. It can be used as an interpreter or compile scripts

- Assembler translates assembly into machine code

```c
#include <stdio.h>

int main()
{
    char name[20];
    ...
    return 0;
}
```

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Assembly</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>push ebp</td>
<td>mov ebp, esp</td>
<td>83 ec 08</td>
</tr>
<tr>
<td>sub esp, 0C0h</td>
<td></td>
<td>83 e4 f0</td>
</tr>
<tr>
<td></td>
<td>b8 00 00 00 00</td>
<td>83 c0 0f</td>
</tr>
</tbody>
</table>
Computer Architecture

**Instruction Set Architecture (ISA):** The "programming language" of the processor, the syntax and language of how to give commands to the processor.
- There are a set of ISAs that are supported by a larger collection of microarchitectures
  - Ex: x86, ARM ISA, TI DSPs ISA
The ISA defines:
- The system’s state (e.g. registers, memory, program counter)
- The instructions the CPU can execute
- The effect that each of these instructions will have on the system state

**Microarchitecture:** The way a specific processor executes a given ISA based on the processor’s design.
- The Microarchitecture defines how the data (data path) moves through the parts of the processor (control path), often represented as a data flow diagram.
- microarchitecture dictates the flow of instructions through items within the processor such as logic gates, registers, Arithmetic Logic Units (ALUs)
Mainstream ISAs

<table>
<thead>
<tr>
<th>x86</th>
<th>ARM architectures</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Designer</strong></td>
<td>Intel, AMD</td>
<td>ARM Holdings</td>
</tr>
<tr>
<td><strong>Bits</strong></td>
<td>16-bit, 32-bit and 64-bit</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td><strong>Introduced</strong></td>
<td>1978 (16-bit), 1985 (32-bit), 2003 (64-bit)</td>
<td>1985; 31 years ago</td>
</tr>
<tr>
<td><strong>Design</strong></td>
<td>CISC</td>
<td>RISC</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Register-memory</td>
<td>Register-Register</td>
</tr>
<tr>
<td><strong>Encoding</strong></td>
<td>Variable (1 to 15 bytes)</td>
<td>AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility [1]</td>
</tr>
<tr>
<td><strong>Endianness</strong></td>
<td>Little</td>
<td>Bi</td>
</tr>
</tbody>
</table>

Macbooks & PCs  
(Core i3, i5, i7, M)  
x86-64 instruction set

Smartphone (and similar) devices  
(iPhone, iPad, Raspberry Pi)  
ARM instruction set

Digital home & networking  
(Blu-ray, Playstation 2)  
MIPS instruction set
So... who writes assembly?

- Chances are, you’ll never write a program in assembly!
  - BUT understanding assembly is the key to the machine-level execution model.

- Some use cases for assembly:
  - When working in embedded where you can’t trust the compiler to reduce program size as efficiently as possible
  - When special purpose subroutines are required that are not possible in higher level languages
  - Behavior of programs in the presence of bugs
    - When high-level language model breaks down
  - Tuning program performance
  - Implementing systems software
  - Fighting malicious software
    - Distributed software is in binary form
Assembly Programmer’s View

- **Programmer-visible state**
  - PC: the Program Counter (%rip in x86-64)
    - Address of next instruction
  - Named registers
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
Registers

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)

- Registers have *names*, not *addresses*
  - In assembly, they start with % (e.g. %rsi)

- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but especially x86

Memory

- Addresses (EX: 0x7FFFD024C3DC)
- Big ~ 8 GiB
- Slow ~50-100 ns
- Dynamic - Can “grow” as needed while program runs

Registers

- Names (EX: %rdi)
- Small - (16 x 8 B) = 128 B
- Fast - sub-nanosecond timescale
- Static - fixed number in hardware
Assembly Instructions Basics

Assembly instructions fall into one of 3 categories:

- **Transfer data** between memory and register
  - Load data from memory into register
    - \(%\text{reg} = \text{Mem}[\text{address}]\)
  - Store register data into memory
    - \(\text{Mem}[\text{address}] = %\text{reg}\)

- **Perform arithmetic** operation on register or memory data
  - \(c = a + b; \quad z = x << y; \quad i = h \& g;\)

- **Control flow**: what instruction to execute next
  - Unconditional jumps to/from procedures
  - Conditional branches

Items in Assembly fall into one of 3 operand categories:

- **Immediate**: Constant integer data
  - Examples: \(0x400, \$-533\)
  - Like C literal, but prefixed with \(\$\)
  - Encoded with 1, 2, 4, or 8 bytes

- **Register**: 1 of 16 integer registers
  - Examples: \(%\text{rax}, %\text{r13}\)

- **Memory**: Consecutive bytes of memory at a computed address
  - Simplest example: \((%\text{rax})\)
Example: Moving Data

- General form: `mov_ source, destination`
  - Missing letter (_) specifies size of operands
  - Lots of these in typical code

Examples:

- `movb src, dst`
  - Move 1-byte “byte”

- `movw src, dst`
  - Move 2-byte “word”

- `movl src, dst`
  - Move 4-byte “long word”

- `movq src, dst`
  - Move 8-byte “quad word”

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td><code>movq $0x4, %rax</code></td>
<td><code>rax = 4;</code></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td><code>movq $-147, (%rax)</code></td>
<td><code>*rax = -147;</code></td>
</tr>
<tr>
<td>Reg</td>
<td>Mem</td>
<td><code>movq %rax, (%rdx)</code></td>
<td><code>*rdx = rax;</code></td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td><code>movq (%rax), %rdx</code></td>
<td><code>rdx = *rax;</code></td>
</tr>
</tbody>
</table>

Assume we have two variables called `rax` and `rdx`. Which assembly instruction does `*rdx = rax`?

1. `movq %rdx, %rax`  
2. `movq (%rdx), %rax`  
3. `movq %rax, (%rdx)`  
4. `movq (%rax), %rdx`
Example: Arithmetic Operations

```c
long simple_arith(long x, long y) {
    long t1 = x + y;
    long t2 = t1 * 3;
    return t2;
}
```

Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1\textsuperscript{st} argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2\textsuperscript{nd} argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

```plaintext
y += x;
y *= 3;
long r = y;
return r;
```
Example: swap()

```c
void swap(long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

**Registers**
- `%rdi`
- `%rsi`
- `%rax`
- `%rdx`

**Memory**

**Register** | **Variable**
---|---
 `%rdi` | `xp`
 `%rsi` | `yp`
 `%rax` | `t0`
 `%rdx` | `t1`
Example: swap()

```plaintext
swap:
  movq (%rdi), %rax  # t0 = *xp
  movq (%rsi), %rdx  # t1 = *yp
  movq %rdx, (%rdi)  # *xp = t1
  movq %rax, (%rsi)  # *yp = t0
  ret
```
Example: swap()

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td>123</td>
</tr>
<tr>
<td>%rdx</td>
<td>456</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>456</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>123</td>
</tr>
</tbody>
</table>

swap:

```
movq (%rdi), %rax  # t0 = *xp
movq (%rsi), %rdx  # t1 = *yp
movq %rdx, (%rdi)  # *xp = t1
movq %rax, (%rsi)  # *yp = t0
ret
```
Where does everything go?

```c
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */

int global = 0;

int useless() { return 0; }

int main()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8); /* 256 B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8); /* 256 B */
    /* Some print statements ... */
}
```
Buffer Overflow

- A buffer is an array used to temporarily store data
  - You’ve probably seen “video buffering...”
  - The video is being written into a buffer before being played
  - Buffers can also store user input

- C does not check array bounds
  - Many Unix/Linux/C functions don’t check argument sizes
  - Allows overflowing (writing past the end) of buffers (arrays)

- “Buffer Overflow” = Writing past the end of an array

- Characteristics of the traditional Linux memory layout provide opportunities for malicious programs
  - Stack grows “backwards” in memory
  - Data and instructions both stored in the same memory
Buffer Overflow

- Stack grows \textit{down} towards lower addresses
- Buffer grows \textit{up} towards higher addresses
- If we write past the end of the array, we overwrite data on the stack!

Enter input: hello
-> no overflow

Enter input: helloabcdef
-> overflow!
What happens when there is an overflow?

- Buffer overflows on the stack can overwrite “interesting” data
  - Attackers just choose the right inputs

- Simplest form (sometimes called “stack smashing”)
  - Unchecked length on string input into bounded array causes overwriting of stack data
  - Try to change the return address of the current procedure

- Why is this a big deal?
  - It was the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance

Enter input: helloabcdef

We’ve lost our way!
Lost address of function pointer telling us which instruction to return to
Malicious Buffer Overflow – Code Injection

- Buffer overflow bugs can allow attackers to execute arbitrary code on victim machines
  - Distressingly common in real programs
- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When bar() executes ret, will jump to exploit code

```
void foo(){
    bar();
    A:... return address A
}
int bar() {
    char buf[64];
    gets(buf);
    ...
    return ...
;
}
```

Examples

- Original "Internet worm" (1988)
  - Early versions of the finger server (fingerd) used gets() to read the argument sent by the client: finger droh@cs.cmu.edu
  - Worm attacked fingerd server with phony argument:
    - finger "exploit-code padding new-return-addr"
    - Exploit code executed a root shell on the victim machine with a direct connection to the attacker
  - Robert Morris is now a professor at MIT, first person convicted under the ‘86 Computer Fraud and Abuse Act

- Heartbleed (2014, affected 17% of servers)
  - Buffer over-read in OpenSSL
  - "Heartbeat" packet
    - Specifies length of message and server echoes it back
    - Library just "trusted" this length
    - Allowed attackers to read contents of memory anywhere they wanted
  - Est. 17% of Internet affected
  - Similar issue in Cloudbleed (2017)
Protect Your Code!

- Employ system-level protections
  - Code on the Stack is not executable
  - Randomized Stack offsets

- Avoid overflow vulnerabilities
  - Use library routines that limit string lengths
  - Use a language that makes them impossible

- Have compiler use "stack canaries"
  - place special value ("canary") on stack just beyond buffer
System Level Protections

- **Non-executable code segments**
  - In traditional x86, can mark region of memory as either “read-only” or “writeable”
    - Can execute anything readable
  - x86-64 added explicit “execute” permission
  - Stack marked as non-executable
    - Do *NOT* execute code in Stack, Static Data, or Heap regions
    - Hardware support needed
  - Works well, but can’t always use it
  - Many embedded devices *do not* have this protection
    - Cars
    - Smart homes
    - Pacemakers
  - Some exploits still work!

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
    - Addresses will vary from one run to another
    - Makes it difficult for hacker to predict beginning of inserted code
Avoid Overflow Vulnerabilities

- Use library routines that limit string lengths
  - fgets instead of gets (2nd argument to fgets sets limit)
  - strncpy instead of strcpy
  - Don’t use scanf with %s conversion specification
    - Use fgets to read the string
    - Or use %ns where n is a suitable integer

- Alternatively, don’t use C - use a language that does array index bounds check
  - Buffer overflow is impossible in Java
    - ArrayIndexOutOfBoundsException
  - Rust language was designed with security in mind
    - Panics on index out of bounds, plus more protections

```c
/* Echo Line */
void echo()
{
    char buf[8]; /* Way too small! */
    fgets(buf, 8, stdin);
    puts(buf);
}
```
Stack Canaries

- **Basic Idea:** place special value ("canary") on stack just beyond buffer
  - *Secret* value that is randomized before main()
  - Placed between buffer and return address
  - Check for corruption before exiting function

- **GCC implementation**
  - `-fstack-protector`

```
unix> ./buf
Enter string: 12345678
12345678
unix> ./buf
Enter string: 123456789
*** stack smashing detected ***
```
Questions