x86-64 Programming III & The Stack
CSE 351 Autumn 2017

Instructor:
Justin Hsia

Teaching Assistants:
Lucas Wotton
Michael Zhang
Parker DeWilde
Ryan Wong
Sam Gehman
Sam Wolfson
Savanna Yee
Vinny Palaniappan

http://xkcd.com/1652/
Administrivia

- Homework 2 due Friday (10/20)
- Lab 2 due next Friday (10/27)

- Section tomorrow on Assembly and GDB
  - Bring your laptops!

- Midterm: 10/30, 5pm in KNE 120
  - You will be provided a fresh reference sheet
  - You get 1 handwritten, double-sided cheat sheet (letter)
  - Midterm Clobber Policy: replace midterm score with score on midterm portion of the final if you “do better”
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- Switches
C allows `goto` as means of transferring control (jump)

- Closer to assembly programming style
- Generally considered bad coding style
Compiling Loops

C/Java code:
```
while ( sum != 0 ) {
    <loop body>
}
```

Assembly code:
```
loopTop:    testq  %rax, %rax
            je     loopDone
            <loop body code>
            jmp   loopTop
loopDone:
```

- Other loops compiled similarly
  - Will show variations and complications in coming slides, but may skip a few examples in the interest of time

- Most important to consider:
  - When should conditionals be evaluated? (while vs. do-while)
  - How much jumping is involved?
Compiling Loops

What are the Goto versions of the following?

- Do...while: Test and Body
- For loop: Init, Test, Update, and Body
Compiling Loops

While Loop:

C:
while ( sum != 0 ) {
    <loop body>
}

Do-while Loop:

C:
do {
    <loop body>
} while ( sum != 0 )

While Loop (ver. 2):

C:
while ( sum != 0 ) {
    <loop body>
}

x86-64:

loopTop:
    testq %rax, %rax
    je loopDone
    <loop body code>
    jmp loopTop

loopDone:

all jump instructions update the program counter (rip)

x86-64 (ver. 2):

loopTop:
    <loop body code>
    testq %rax, %rax
    jne loopTop

loopDone:

7
For Loop → While Loop

For Version

```plaintext
for (Init; Test; Update) 
  Body
```

While Version

```plaintext
Init;
while (Test) { 
  Body 
  Update;
}
```

Caveat: C and Java have break and continue

- Conversion works fine for break
  - Jump to same label as loop exit condition
- But not continue: would skip doing Update, which it should do with for-loops
  - Introduce new label at Update
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- Switches
Switch Statement

Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4
- Implemented with:
  - Jump table
  - Indirect jump instruction

```
long switch_ex (long x, long y, long z)
{
    long w = 1;
    switch (x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```
Jump Table Structure

### Switch Form

```java
switch (x) {
    case val_0:
        Block 0
    case val_1:
        Block 1
    • • •
    case val_n-1:
        Block n–1
}
```

### Jump Table

- **Jump Table**: `Targ0`, `Targ1`, `Targ2`, ..., `Targn-1`
- **Jump Targets**: `Targ0`, `Targ1`, `Targ2`, ..., `Targn-1`

```java
JTab[] = target = JTab[x];
goto target;
```

- **Approximate Translation**
- **Jump Table**: Addresses (8 bytes wide)
- **Jump Targets**
- **Like an array of pointers**
Jump Table Structure

C code:

```
switch (x) {
    case 1: <some code>
            break;
    case 2: <some code>
    case 3: <some code>
            break;
    case 5:
    case 6: <some code>
            break;
    default: <some code>
}
```

Use the jump table when $x \leq 6$:

```
if (x <= 6)
    target = JTab[x];
    goto target;
else
    goto default;
```
Switch Statement Example

```c
long switch_ex(long x, long y, long z) {
    long w = 1;
    switch (x) {
        . . .
    }
    return w;
}
```

### Register Use(s)
- `%rdi` 1st argument (x)
- `%rsi` 2nd argument (y)
- `%rdx` 3rd argument (z)
- `%rax` Return value

### Note compiler chose to not initialize `w`

### Take a look!
- [https://godbolt.org/g/DnOmXb](https://godbolt.org/g/DnOmXb)
Switch Statement Example

```c
long switch_ex(long x, long y, long z) {
    long w = 1;
    switch (x) {
        ...
    }
    return w;
}
```

### Jump table

`.section .rodata
.align 8
.L4:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L5 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L7 # x = 5
.quad .L7 # x = 6

Switch Example:
```c
movq %rdx, %rcx
cmpq $6, %rdi # x:6
ja .L8 # default
jmp *._L4(,%rdi,8) # jump table
```
Assembly Setup Explanation

- **Table Structure**
  - Each target requires 8 bytes (address)
  - Base address at `.L4`

- **Direct jump:** `jmp .L8`
  - Jump target is denoted by label `.L8`

- **Indirect jump:** `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective address `.L4 + x*8`
    - Only for $0 \leq x \leq 6$

---

**Jump table**

```
.section .rodata
.align 8
.L4:
    .quad .L8  # x = 0
    .quad .L3  # x = 1
    .quad .L5  # x = 2
    .quad .L9  # x = 3
    .quad .L8  # x = 4
    .quad .L7  # x = 5
    .quad .L7  # x = 6
```
Jump Table

declaring data, not instructions

Jump table

this data is 64-bits wide

8-byte memory alignment

`.section .rodata
    .align 8

.L4:
    .quad .L8 # x = 0
    .quad .L3 # x = 1
    .quad .L5 # x = 2
    .quad .L9 # x = 3
    .quad .L8 # x = 4
    .quad .L7 # x = 5
    .quad .L7 # x = 6

switch(x) {
    case 1:     // .L3
        w = y*z;
        break;
    case 2:      // .L5
        w = y/z;
        /* Fall Through */
    case 3:      // .L9
        w += z;
        break;
    case 5:
    case 6:      // .L7
        w -= z;
        break;
    default:     // .L8
        w = 2;
}

8-byte memory alignment

this data is 64-bits wide
Code Blocks (x == 1)

```
switch(x) {
    case 1:    // .L3
        w = y*z;
        break;
    ...
}
```

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

### Code Block .L3:

```
movq %rsi, %rax  # y
imulq %rdx, %rax  # y*z
ret
```
Handling Fall-Through

```c
long w = 1;

... switch (x) {
    ... case 2: // .L5
        w = y/z;
        /* Fall Through */
    case 3: // .L9
        w += z;
        break;
    ... }
```

More complicated choice than “just fall-through” forced by “migration” of `w = 1`;

- Example compilation trade-off

```c
case 2:
    w = y/z;
    goto merge;
```

```c
case 3:
    w = 1;
merge:
    w += z;
```
Code Blocks (x == 2, x == 3)

```c
long w = 1;
    . . .
switch (x) {
    . . .
    case 2: // .L5
        w = y/z;
        /* Fall Through */
    case 3: // .L9
        w += z;
        break;
    . . .
}
```

```
.L5:
    movq %rsi, %rax # y in rax
    cqto # Div prep
    idivq %rcx # y/z
    jmp .L6 # goto merge
.L9:
    movl $1, %eax # w = 1
.L6:
    addq %rcx, %rax # w += z
    ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Code Blocks (rest)

```c
switch (x) {
    ...
    case 5: // .L7
    case 6: // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1\text{st} argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2\text{nd} argument (y)</td>
</tr>
<tr>
<td>%rdx</td>
<td>3\text{rd} argument (z)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Roadmap

C:

car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:

Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly language:

get_mpg:

pushq %rbp
movq %rsp, %rbp
...
popq %rbp
ret

Machine code:

0111010000011000
10001101000000001000011111
10001100000010
1000110111000010
110000011111101000011111

OS:

Windows 10
OS X Yosemite

Computer system:

CPU
Memory
Virtual memory
Memory allocation
Java vs. C

Memory & data
Integers & floats
x86 assembly

Procedures & stacks
Executables
Arrays & structs
Processes
Virtual memory
Memory & caches
Java vs. C
Mechanisms required for procedures

1) Passing control
   - To beginning of procedure code
   - Back to return point

2) Passing data
   - Procedure arguments
   - Return value

3) Memory management
   - Allocate during procedure execution
   - Deallocate upon return

- All implemented with machine instructions!
  - An x86-64 procedure uses only those mechanisms required for that procedure

```c
int Q(int i) {
    int t = 3*i;
    int v[10];
    ...
    return v[t];
}
```

```c
P(...) {
    ...
    y = Q(x);
    print(y)
    ...
}
```
Procedures

- **Stack Structure**
- **Calling Conventions**
  - Passing control
  - Passing data
  - Managing local data
- **Register Saving Conventions**
- **Illustration of Recursion**
Simplified Memory Layout

- **Static Data**: static variables (including global variables (C))
- **Dynamic Data (Heap)**: variables allocated with `new` or `malloc`
- **Literals**: large constants (e.g., “example”)
- **Instructions/Code**: program code
- **Stack**: local variables; procedure context

![Memory Address Diagram]

- **High Addresses**: 0xF...F
- **Low Addresses**: 0x0...0
- **$2^N - 1$**: dynamic over process execution
  - Fixed in size
Memory Permissions

- **Stack**: Writable; not executable
  - Managed "automatically" (by compiler)
  - Grow towards each other to maximize use of space

- **Dynamic Data (Heap)**: Writable; not executable
  - Managed by programmer

- **Static Data**: Writable; not executable
  - Initialized when process starts

- **Literals**: Read-only; not executable
  - Initialized when process starts

- **Instructions**: Read-only; executable
  - Initialized when process starts

Segmentation faults? Accessing memory in a way that you are not allowed to.
x86-64 Stack

- Region of memory managed with stack “discipline”
  - Grows toward lower addresses
  - Customarily shown “upside-down”

- Register `%rsp` contains lowest stack address
  - `%rsp` = address of top element, the most-recently-pushed item that is not-yet-popped

Last In, First Out (LIFO)

Stack Pointer: `%rsp`

Stack “Bottom”

Stack “Top”

High Addresses

Increasing Addresses

Stack Grows Down

Low Addresses

0x00...00
x86-64 Stack: Push

- **pushq** `src`  
  - Fetch operand at `src`  
    - `Src` can be reg, memory, immediate  
  - **Decrement** `%rsp` by 8  
  - Store value at address given by `%rsp`

- **Example:**
  - `pushq %rcx`  
  - Adjust `%rsp` and store contents of `%rcx` on the stack

**Stack Pointer:** `%rsp`  

1. Move `%rsp` down (subtract)  
2. Store `src` at `%rsp`
x86-64 Stack: Pop

- `popq dst`
  - Load value at address given by `%rsp`
  - Store value at `dst` (must be register)
  - *Increment* `%rsp` by 8

- Example:
  - `popq %rcx`
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`

Those bits are still there; we’re just not using them.