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

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

Expressing with Goto Code
❖ C allows goto as means of transferring control (jump)
  ❦ Closer to assembly programming style
  ❦ Generally considered bad coding style

Compiling Loops
❖ 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

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 code>
}

x86 Control Flow

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

Jump Table Structure

Switch Form
switch (x) {
case val_0:
    Block0
    ...
case val_n-1:
    Block n-1
}

Jump Targets
Targ0 -> Code Block 0
Targ1 -> Code Block 1
Targn-1 -> Code Block n-1

Approximate Translation
Target = JTab[x];
goto target;

Jump Table Structure

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

Use the jump table when x ≤ 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;
}
```

Jump table

```c
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
- 8-byte memory alignment
- B-byte memory alignment
- Switch(x) {
  case 1: // .L3
      w = y/z;
  break;
  case 2: // .L5
      w = y/z;
  /* Fall Through */
  case 3: // .L9
      w = 2;
  break;
  case 5: // .L7
      w = 2;
  break;
  default: // .L8
      w = 2;
}

Handling Fall-Through

```c
long w = 1;
switch (x) {
    . . .
    case 1: // .L3
        w = y/z;
    break;
    . . .
    case 2: // .L5
        w = y/z;
    /* Fall Through */
    case 3: // .L9
        w = 2;
    break;
    . . .
}
```

More complicated choice than "just fall-through" forced by "migration" of w = 1;
- Example compilation trade-off
Code Blocks (x == 2, x == 3)

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

Code Blocks (rest)

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

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
Memory Permissions

- **Stack**: Managed "automatically" (by compiler)
- **Dynamic Data (Heap)**: Managed by programmer
- **Static Data**: Initialized when process starts
- **Literals**: Initialized when process starts
- **Instructions**: Initialized when process starts
- **Heap**: Managed by programmer, writable; not executable

---

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

**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

**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.