Machine-Level Programming III: Switch Statements and IA32 Procedures

15-213/18-213: Introduction to Computer Systems
7th Lecture, 5 February 2013

Instructors:
Anthony Rowe, Seth Goldstein and Gregory Kesden

Switch Statement Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

```c
long switch_eg (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;
}
```

Today

- Switch statements
- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

Jump Table Structure

**Switch Form**

```c
switch(x) {
  case val_0:  Block 0
  case val_1:  Block 1
  . . .
  case val_n-1: Block n-1
}
```

**Jump Table**

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

**Jump Targets**

```
  Targ0:  Code Block 0
  Targ1:  Code Block 1
  Targ2:  Code Block 2
  . . .
  Targn-1: Code Block n-1
```
Switch Statement Example (IA32)

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

Setup:
```
carnegie Mellon
```

Assembly Setup Explanation

- **Table Structure**
  - Each target requires 4 bytes
  - Base address at .L7

- **Jumping**
  - **Direct**: jmp .L2
  - Jump target is denoted by label .L2

  - **Indirect**: jmp * .L7(,%eax,4)
  - Start of jump table: .L7

- Must scale by factor of 4 (labels have 32-bits = 4 bytes on IA32)
- Fetch target from effective address .L7 + eax * 4
  - Only for 0 ≤ x ≤ 6

Jump Table

```c
switch(x) {
    case 1:      // .L3
        w = y*z;
        break;
    case 2:      // .L4
        w = y/z;
        /* Fall Through */
    case 3:      // .L5
        w += z;
        break;
    case 5:      // .L6
        w -= z;
        break;
    default:     // .L2
        w = 2;
}
```
Handling Fall-Through

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

Code Blocks (Partial)

```c
switch(x) {
  case 1: // .L3
    w = y*z;
    break;
  ...;
case 3: // .L5
    w += z;
    break;
    ...;
default: // .L2
    w = 2;
}
```

Code Blocks (Rest)

```c
switch(x) {
  ...
case 2: // .L4
  w = y/z;
  /* Fall Through */
  merge: // .L9
    w = z;
    break;
case 5:
case 6: // .L6
  w = z;
  break;
}
```

Switch Code (Finish)

```c
return w;
```

Noteworthy Features
- Jump table avoids sequencing through cases
  - Constant time, rather than linear
- Use jump table to handle holes and duplicate tags
- Use program sequencing to handle fall-through
- Don’t initialize `w = 1` unless really need it
x86-64 Switch Implementation

- Same general idea, adapted to 64-bit code
- Table entries 64 bits (pointers)
- Cases use revised code

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

Jump Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08048660</td>
<td>0x08048422</td>
</tr>
<tr>
<td>0x08048664</td>
<td>0x08048432</td>
</tr>
<tr>
<td>0x08048668</td>
<td>0x0804843b</td>
</tr>
<tr>
<td>0x0804866c</td>
<td>0x08048429</td>
</tr>
<tr>
<td>0x08048670</td>
<td>0x08048422</td>
</tr>
<tr>
<td>0x08048674</td>
<td>0x0804844b</td>
</tr>
<tr>
<td>0x08048678</td>
<td>0x0804844b</td>
</tr>
</tbody>
</table>

IA32 Object Code

- Setup
  - Label .L2 becomes address 0x08048422
  - Label .L7 becomes address 0x08048660

Assembly Code

```assembly
switch_eg:
    ... 
    ja .L2 # If unsigned > goto default
    jmp *.L7(%eax,4) # Goto *JTab[x]
```

Disassembled Object Code

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08048410</td>
<td>0x08048419</td>
</tr>
<tr>
<td>0x0804841b</td>
<td>0x0804841f</td>
</tr>
</tbody>
</table>

IA32 Object Code (cont.)

- Jump Table
  - Doesn’t show up in disassembled code
  - Can inspect using GDB
  - gdb switch
  - (gdb) x/1xw 0x8048660
    - Examine 7 hexadecimal format “words” (4-bytes each)
    - Use command “help x” to get format documentation

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08048660</td>
<td>0x08048422</td>
</tr>
<tr>
<td>0x08048664</td>
<td>0x08048432</td>
</tr>
<tr>
<td>0x08048668</td>
<td>0x0804843b</td>
</tr>
<tr>
<td>0x0804866c</td>
<td>0x08048429</td>
</tr>
<tr>
<td>0x08048670</td>
<td>0x08048422</td>
</tr>
<tr>
<td>0x08048674</td>
<td>0x0804844b</td>
</tr>
<tr>
<td>0x08048678</td>
<td>0x0804844b</td>
</tr>
</tbody>
</table>

IA32 Object Code (cont.)

- Deciphering Jump Table

```
0x8048660: 0x8048422 0x8048432 0x804843b 0x8048429
0x8048670: 0x8048422 0x804844b 0x804844b
```
Disassembled Targets

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048422</td>
<td>b8 02 00 00 00</td>
<td>mov $0x2, %eax</td>
</tr>
<tr>
<td>0x8048427</td>
<td>ab 2a</td>
<td>jmp 0x8048453 &lt;switch_eg+0x43&gt;</td>
</tr>
<tr>
<td>0x804842a</td>
<td>66 80</td>
<td>scchg far, %eax # noop</td>
</tr>
<tr>
<td>0x8048430</td>
<td>ab 34</td>
<td>jmp 0x8048446 &lt;switch_eg+0x36&gt;</td>
</tr>
<tr>
<td>0x8048432</td>
<td>8b 45 10</td>
<td>mov 0x10(%ebp), %eax</td>
</tr>
<tr>
<td>0x8048435</td>
<td>0f af 45 0c</td>
<td>imul 0xc(%ebp), %eax</td>
</tr>
<tr>
<td>0x8048439</td>
<td>eb 18</td>
<td>jmp 0x8048453 &lt;switch_eg+0x43&gt;</td>
</tr>
<tr>
<td>0x804843b</td>
<td>8b 55 0c</td>
<td>mov 0xc(%ebp), %edx</td>
</tr>
<tr>
<td>0x804843e</td>
<td>89 d0</td>
<td>mov %edx, %eax</td>
</tr>
<tr>
<td>0x8048440</td>
<td>c1 fa 1f</td>
<td>sar $0x1f, %edx</td>
</tr>
<tr>
<td>0x8048443</td>
<td>f7 7d 10</td>
<td>idivl 0x10(%ebp)</td>
</tr>
<tr>
<td>0x8048446</td>
<td>03 45 10</td>
<td>add 0x10(%ebp), %eax</td>
</tr>
<tr>
<td>0x8048449</td>
<td>db 08</td>
<td>jmp 0x8048453 &lt;switch_eg+0x43&gt;</td>
</tr>
<tr>
<td>0x804844c</td>
<td>b8 01 00 00 00</td>
<td>mov $0x1, %eax</td>
</tr>
<tr>
<td>0x8048450</td>
<td>2b 45 10</td>
<td>sub 0x10(%ebp), %eax</td>
</tr>
<tr>
<td>0x8048453</td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0x8048454</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

Matching Disassembled Targets

<table>
<thead>
<tr>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048422</td>
<td>mov $0x2, %eax</td>
</tr>
<tr>
<td>0x8048427</td>
<td>jmp 0x8048453 &lt;switch_eg+0x43&gt;</td>
</tr>
<tr>
<td>0x804842a</td>
<td>scchg far, %eax # noop</td>
</tr>
<tr>
<td>0x8048430</td>
<td>jmp 0x8048446 &lt;switch_eg+0x36&gt;</td>
</tr>
<tr>
<td>0x8048432</td>
<td>mov 0x10(%ebp), %eax</td>
</tr>
<tr>
<td>0x8048435</td>
<td>imul 0xc(%ebp), %eax</td>
</tr>
<tr>
<td>0x8048439</td>
<td>jmp 0x8048453 &lt;switch_eg+0x43&gt;</td>
</tr>
<tr>
<td>0x804843b</td>
<td>mov 0xc(%ebp), %edx</td>
</tr>
<tr>
<td>0x804843e</td>
<td>mov %edx, %eax</td>
</tr>
<tr>
<td>0x8048440</td>
<td>sar $0x1f, %edx</td>
</tr>
<tr>
<td>0x8048443</td>
<td>idivl 0x10(%ebp)</td>
</tr>
<tr>
<td>0x8048446</td>
<td>add 0x10(%ebp), %eax</td>
</tr>
<tr>
<td>0x8048449</td>
<td>jmp 0x8048453 &lt;switch_eg+0x43&gt;</td>
</tr>
<tr>
<td>0x804844c</td>
<td>mov $0x1, %eax</td>
</tr>
<tr>
<td>0x8048450</td>
<td>sub 0x10(%ebp), %eax</td>
</tr>
<tr>
<td>0x8048453</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0x8048454</td>
<td>ret</td>
</tr>
</tbody>
</table>

Summarizing

- **C Control**
  - if-then-else
  - do-while
  - while, for
  - switch

- **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler generates code sequence to implement more complex control

- **Standard Techniques**
  - Loops converted to do-while form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees

Today

- **Switch statements**
- **IA 32 Procedures**
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers
**IA32 Stack**
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` contains lowest stack address
  - address of "top" element

**IA32 Stack: Push**
- `pushl` `Src`
  - Fetch operand at `Src`
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`

**IA32 Stack: Pop**
- `popl` `Dst`
  - Read operand at address given by `%esp`
  - Put into `Dst`
  - Increment `%esp` by 4

**Procedure Control Flow**
- Use stack to support procedure call and return
- **Procedure call:** `call label`
  - Push return address on stack
  - Jump to label
- **Return address:**
  - Address of the next instruction right after call
  - Example from disassembly:
    ```
    804854e: e8 3d 06 00 00 call 8048b90 <main>
    8048553: 50 pushl %eax
    ```
    - Return address: `0x8048553`
- **Procedure return:** `ret`
  - Pop address from stack
  - Jump to address
Procedure Call Example

Stack-Based Languages
- Languages that support recursion
  - e.g., C, Pascal, Java
  - Code must be "Reentrant"
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer
- Stack discipline
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
- Stack allocated in Frames
  - state for single procedure instantiation

Procedure Return Example

Call Chain Example

Procedure ami () is recursive
Stack Frames

- Contents
  - Local variables
  - Return information
  - Temporary space

- Management
  - Space allocated when enter procedure
    - “Set-up” code
  - Deallocated when return
    - “Finish” code

Example

```c
yoo(...) {
  ...
  who();
  ...
}
```

Example

```c
who(...) {
  ...
  amI();
  ...
  ...
```
IA32/Linux Stack Frame

- Current Stack Frame (“Top” to Bottom)
  - “Argument build:” Parameters for function about to call
  - Local variables
    - If can’t keep in registers
  - Saved register context
  - Old frame pointer

- Caller Stack Frame
  - Return address
    - Pushed by call instruction
  - Arguments for this call

Revisiting swap

void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}

Revisiting swap

void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}

void call_swap() {
  swap(&course1, &course2);
}

Calling swap from call_swap

void call_swap() {
  subl $8, %esp
  movl $course2, 4(%esp)
  movl $course1, (%esp)
  call swap
}

void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}

swap Setup #1

void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}

void call_swap() {
  subl $8, %esp
  movl $course2, 4(%esp)
  movl $course1, (%esp)
  call swap
}

void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}

Calling swap from call_swap

void call_swap() {
  subl $8, %esp
  movl $course2, 4(%esp)
  movl $course1, (%esp)
  call swap
}

void swap(int *xp, int *yp) {
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}

void call_swap() {
  subl $8, %esp
  movl $course2, 4(%esp)
  movl $course1, (%esp)
  call swap
}
### swap Setup #2

**Entering Stack**

- `%ebp`
- `%esp`
- `&course2`
- `&course1`
- `Rtn adr`

**Resulting Stack**

- `%ebp`
- `%esp`
- `yp`
- `xp`
- `Rtn adr`
- `Old %ebp`
- `Old %esp`

**swap**

- `pushl %ebp`
- `movl %esp,%ebp`
- `pushl %ebx`

**Observation**

- Saved and restored register `%ebx`
- Not so for `%eax`, `%ecx`, `%edx`
Disassembled swap

08048394 <swap>:

08048384:  89 e5  mov  %esp, %ebp
08048385:  53     push %ebp
08048386:  8b 55 08  mov  0x8(%ebp), %edx
08048388:  8b 4d 0c  mov  0xc(%ebp), %ecx
08048389:  8b 1a  mov  (%edx), %ebx
0804838b:  8b 01  mov  (%ecx), %eax
0804838d:  89 02  mov  %eax, (%edx)
0804838f:  89 19  mov  %ebx, (%ecx)
08048390:  5b     pop  %ebx
08048391:  5d     pop  %ebp
08048392:  c3     ret

080483b4:  movl $0x8049658, 0x4(%esp)  # Copy &course2
080483bc:  movl $0x8049654, (%esp)  # Copy &course1
080483c0:  moveal 8048384 <swap>  # Call swap
080483c2:  leave
080483c3:  ret

Register Saving Conventions

- **When procedure yoo calls who:**
  - yoo is the **caller**
  - who is the **callee**
- Can Register be used for temporary storage?

  **yoo:**
  
  • • •
  movl $15213, %edx
  call who
  addl %edx, %eax
  • • •

  **who:**
  
  • • •
  movl 8(%ebp), %edx
  addl $18243, %edx
  • • •
  ret

  - Contents of register %edx overwritten by who
  - This could be trouble → something should be done!
  - Need some coordination

Today

- Switch statements
- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

Register Saving Conventions

- **When procedure yoo calls who:**
  - yoo is the **caller**
  - who is the **callee**
- Can Register be used for temporary storage?
- **Conventions**
  - "Caller Save"
    - Caller saves temporary values in its frame before the call
  - "Callee Save"
    - Callee saves temporary values in its frame before using
IA32/Linux+Windows Register Usage

- `%eax`, `%edx`, `%ecx`
  - Caller saves prior to call if values are used later
- `%eax`
  - also used to return integer value
- `%ebx`, `%esi`, `%edi`
  - Callee saves if wants to use them
- `%esp`, `%ebp`
  - special form of callee save
  - Restored to original values upon exit from procedure

Recursive Function

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

Registers

- `%eax`, `%edx` used without first saving
- `%ebx` used, but saved at beginning & restored at end

Recursive Call #1

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

Actions

- Save old value of `%ebx` on stack
- Allocate space for argument to recursive call
- Store x in `%ebx`

Today

- Switch statements
- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers
Recursive Call #2

/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}

- Actions
  - If x == 0, return
    - with %eax set to 0

Recursive Call #3

/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}

- Actions
  - Store x >> 1 on stack
  - Make recursive call
- Effect
  - %eax set to function result
  - %ebx still has value of x

Recursive Call #4

/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}

- Assume
  - %eax holds value from recursive call
  - %ebx holds x
- Actions
  - Compute (x & 1) + computed value
- Effect
  - %eax set to function result

Recursive Call #5

/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}

- Actions
  - Restore values of %ebx and %ebp
  - Restore %esp

L3: add $4, %esp
    pop %ebx
    popl %ebp
    ret
Observations About Recursion

- **Handled Without Special Consideration**
  - Stack frames mean that each function call has private storage
  - Saved registers & local variables
  - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another’s data
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
  - Also works for mutual recursion
    - P calls Q; Q calls P

- **Stack frames** mean that each function call has private storage
- **Saved registers** & local variables
- **Saved return pointer**
- **Register saving conventions** prevent one function call from corrupting another’s data
- **Stack discipline** follows call / return pattern
  - If P calls Q, then Q returns before P
  - **Last-In, First-Out**
- Also works for mutual recursion
  - P calls Q; Q calls P

---

Pointer Code

**Generating Pointer**

```c
/* Compute x + 3 */
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

**Referencing Pointer**

```c
/* Increment value by k */
void incrk(int *ip, int k) {
    *ip += k;
}
```

- `add3` creates pointer and passes it to `incrk`

---

Creating and Initializing Local Variable

```c
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

- Variable `localx` must be stored on stack
  - Because: Need to create pointer to it
- Compute pointer as `-4(%ebp)`

**First part of add3**

- `pushl %ebp`
- `movl %esp, %ebp`
- `subl $24, %esp` # Alloc. 24 bytes
- `movl 8(%ebp), %eax`
- `movl %eax, -4(%ebp)# Set localx to x`

**Middle part of add3**

- `movl $3, 4(%esp) # 2nd arg = 3`
- `lea1 -4(%ebp), %eax# localx`
- `movl %eax, (%esp)# 1st arg = &localx`
- `call incrk`

---

Creating Pointer as Argument

```c
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

- Use `leal` instruction to compute address of `localx`
Retrieving local variable

```c
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

Final part of `add3`

```assembly
movl -4(%ebp), %eax # Return val= localx
leave
ret
```

IA 32 Procedure Summary

- **Important Points**
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
  - Recursion (& mutual recursion) handled by normal calling conventions
    - Can safely store values in local stack frame and in callee-saved registers
    - Put function arguments at top of stack
    - Result return in %eax
  - Pointers are addresses of values
    - On stack or global