Machine Level Programming: Switch and Procedures

Today

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

Switch Statement Example

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

Switch Statement Example (IA32)

setup:

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

Jump Table Structure

Jump Targets

Targ0: Code Block 0
Targ1: Code Block 1
Targ2: Code Block 2
Targ-n-1: Code Block n-1

Approximate Translation

target = JTab[x];
goto *target;

Switch Statement Example (IA32)

setup:

Note that w not initialized here
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-bit address)
    - Fetch target from effective Address .L7 + eax*4 (x is stored in eax)

IA32 Object Code

- **Setup**
  - Label .L2 becomes address 0x8048442
  - Label .L7 becomes address 0x8048660

**Assembly Code**

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

**Disassembled Object Code**

```
0x8048410 <switch_eg>:
  8048419: 77 07  ja  8048422 <switch_eg+0x12>
  804841b: ff 24 85 60 86 04 08  jmp  *0x8048660(*%eax,4)
```

IA32 Object Code (cont.)

- **Jump Table**
  - Doesn’t show up in disassembled code
  - Can inspect using GDB
  - gdb switch
    - *(gdby)* /x/7xw 0x8048660
      - Examines 2 hexadecimal format *words* (4-bytes each)
      - Use command 'help x' to get format documentation

**Disassembled Targets**

```
0x8048660: 0x08048442 0x08048432 0x0804843b 0x08048429
0x8048670: 0x08048442 0x0804844b 0x0804844b
```

IA32 Object Code (cont.)

- **Deciphering Jump Table**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08048429</td>
<td>0x08048422</td>
<td>0x08048432</td>
</tr>
<tr>
<td>0x08048460</td>
<td>0x08048422</td>
<td>0x08048432</td>
</tr>
</tbody>
</table>

- **Disassembled Targets**

```
0x8048442:  ba 00 00 00  mov  $0x1a, %eax
0x8048443:  ba 00 00 00  mov  $0x1a, %eax
0x8048444:  ba 00 00 00  mov  $0x1a, %eax
0x8048445:  ba 00 00 00  mov  $0x1a, %eax
0x8048446:  ba 00 00 00  mov  $0x1a, %eax
0x8048447:  ba 00 00 00  mov  $0x1a, %eax
0x8048448:  ba 00 00 00  mov  $0x1a, %eax
0x8048449:  ba 00 00 00  mov  $0x1a, %eax
0x804844a:  ba 00 00 00  mov  $0x1a, %eax
0x804844b:  ba 00 00 00  mov  $0x1a, %eax
0x804844c:  ba 00 00 00  mov  $0x1a, %eax
0x804844d:  ba 00 00 00  mov  $0x1a, %eax
0x804844e:  ba 00 00 00  mov  $0x1a, %eax
0x804844f:  ba 00 00 00  mov  $0x1a, %eax
```

- **Disassembled Object Code**

```
0x8048410 <switch_eg>:
  0x8048419: 77 07  ja  0x8048422 <switch_eg+0x12>
  0x804841b: ff 24 85 60 86 04 08  jmp  *0x8048660(*%eax,4)
```

- **Jump Table**

```
section .rodata
.align 4
.long .12 # x = 0
.long .13 # x = 1
.long .14 # x = 2
.long .15 # x = 3
.long .16 # x = 4
.long .17 # x = 5
.long .18 # x = 6

.switch[x] { 
  case 1: // .L2
    w = y/z; break;
  case 2: // .L4
    w = y/z; break;
  /* Fall Through */
  case 3: // .L5
    w = x; break;
  case 4: // .L6
    w = x; break;
  default: // .L2
    w = 2;
}
```
Matching Disassembled Targets

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
  - Write operand at address given by $esp$ to 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
  • Return address = 0x8048553
  • Procedure return: **ret**
    - Pop address from stack
    - Jump to address

Procedure Call Example

```
804854e: e8 3d 06 00 00  
call 8048b90 <main>
8048553: 50
pushl %eax
```

Procedure Return Example

```
8048591: c3  
ret
```

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

Stack Frames

• Contents
  - Local variables
  - Return information
  - Temporary space

• Management
  - Space allocated when enter procedure
    - "Set-up" code
  - Deallocated when return
    - "Finish" code
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

Example

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

Example

Example

Example

Example
Revisiting swap

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

Calling swap from call_swap

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

Swap Setup #1

**Entering Stack**

- %ebp
- course2
- course1
- Rtn adr

**Resulting Stack**

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

**Swap**: pushl %ebp
movl %esp,%ebp
pushl %ebx

Swap Setup #2

**Entering Stack**

- %ebp
- course2
- course1
- Rtn adr

**Resulting Stack**

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

**Swap**: pushl %ebp
movl %esp,%ebp
pushl %ebx

Swap Setup #3

**Entering Stack**

- %ebp
- course2
- course1
- Rtn adr

**Resulting Stack**

- %ebp
- yp
- xp
- Rtn adr
- Old %ebp
- Old %ebx

**Swap**: pushl %ebp
movl %esp,%ebp
pushl %ebx

Swap Body

```asm
call_swap:
      push %ebp
      movl %esp,%ebp
      pushl %ebx
      subl $8, %esp
      movl $course2, 4(%ebp)
      movl $course1, (%ebp)
      call swap
      movl $course2, 4(%esp)
      movl $course1, (%esp)
      call swap
      movl $course2, (%esp)
      movl $course1, (%esp)
      popl %ebx
      popl %ebp
      ret
```
Today

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

Register Saving Conventions

• When procedure \texttt{yoo} calls \texttt{who}:
  • \texttt{yoo} is the \textit{caller}
  • \texttt{who} is the \textit{callee}
• Can register be used for temporary storage?
• Conventions
  • \textit{“Caller Save”}
    • Caller saves temporary values in its frame before the call
  • \textit{“Callee Save”}
    • Callee saves temporary values in its frame before using

Disassembled swap

\begin{verbatim}
08048384 <swap>: 8048384: 55
          push   %ebp
8048385: 89 e5
          mov    %esp,%ebp
8048387: 53
          push   %ebx
8048388: 8b 55 08
          mov    0x8(%ebp),%edx
804838a: 8b 4d 0c
          mov    0xc(%ebp),%ecx
804838c: 8b 1a
          mov    (%edx),%ebx
804838e: 8b 01
          mov    (%ecx),%eax
8048390: 89 02
          mov    %eax,(%edx)
8048392: 89 19
          mov    %ebx,(%ecx)
8048394: 5b
          pop    %ebx
8048395: 5d
          pop    %ebp
8048396: c3
          ret
80483b4: movl   $0x8049658,0x4(%esp)  # Copy &course2
80483bc: movl   $0x8049654,(%esp)  # Copy &course1
80483c3: call   8048384 <swap>  # Call swap
80483c8: leave  # Prepare to return
80483c9: ret    # Return
\end{verbatim}

Register Saving Conventions

• When procedure \texttt{yoo} calls \texttt{who}:
  • \texttt{yoo} is the \textit{caller}
  • \texttt{who} is the \textit{callee}
• Can register be used for temporary storage?
• Conventions
  • \textit{“Caller Save”}
    • Caller saves prior to call if values are used later
  • \textit{“Callee Save”}
    • Caller saves temporary values in its frame before the call
    • Callee saves if wants to use them
• \texttt{ebp}, \texttt{esp}
  • special form of callee save
  • Restored to original values upon exit from procedure
Today

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

Recursive Function

```c
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`

Recursive Call #2

```c
/* 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

```c
/* 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

Recursive Call #4

```c
/* 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

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

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

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/indirect recursion
  - P calls Q; Q calls P
  - P calls Q; Q calls R; R calls P

Creating and Initializing Local Variable

- Variable localx must be stored on stack because we need to create pointer to it
- Compute pointer as -(%ebp)

Creating Pointer as Argument

- Use leal instruction to compute address of localx

Retrieving local variable

- Retrieve localx from stack as return value
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