SWITCH STATEMENTS AND IA32 PROCEDURES

Spring 2014

Euiseong Seo
(euiseong@gmail.com)
Today

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

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

Jump Table

- jtab:
  - Targ0
  - Targ1
  - Targ2
  - Targn-1

Jump Targets

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

Approximate Translation

```java
target = JTab[x];
goto *target;
```
long switch_eg(long x, long y, long z) {
    long w = 1;
    switch(x) {
        ...
    }
    return w;
}

What range of values takes default?

Note that w not initialized here
long switch_eg(long x, long y, long z) {
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}

Jump table

.section .rodata
.align 4
.L7:
  .long .L2 # x = 0
  .long .L3 # x = 1
  .long .L4 # x = 2
  .long .L5 # x = 3
  .long .L2 # x = 4
  .long .L6 # x = 5
  .long .L6 # x = 6

Setup:

    switch_eg:
      pushl %ebp      # Setup
      movl %esp, %ebp # Setup
      movl 8(%ebp), %eax # eax = x
      cmpl $6, %eax   # Compare x:6
      ja .L2          # If unsigned > goto default
      jmp *+.L7(,%eax,4) # Goto *JTab[x]
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 $0 \leq x \leq 6$
Jump table

```
.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
  case 6:   // .L6
    w -= z;
    break;
  default:  // .L2
    w = 2;
}
```
long w = 1;
    . . .
switch(x) {
    . . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
    . . .
}
switch(x) {
  case 1:     // .L3
    w = y*z;
    break;
    . . .
  case 3:     // .L5
    w += z;
    break;
    . . .
  default:    // .L2
    w = 2;
}
switch(x) {
    ... 
    case 2:  // .L4
        w = y/z;
        /* Fall Through */
    merge:    // .L9
        w += z;
        break;
    case 5:
    case 6:  // .L6
        w -= z;
        break;
}
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

```assembly
.L8:    # done:
    popl  %ebp
    ret
```
Same general idea, adapted to 64-bit code

Table entries 64 bits (pointers)

Cases use revised code

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

Jump Table

```
.section .rodata
.align 8
.L7:
    .quad .L2  # x = 0
    .quad .L3  # x = 1
    .quad .L4  # x = 2
    .quad .L5  # x = 3
    .quad .L2  # x = 4
    .quad .L6  # x = 5
    .quad .L6  # x = 6
```

```
.L3:
    movq  %rdx, %rax
    imulq %rsi, %rax
    ret
```
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

Stack Pointer: $\%esp$

Stack “Bottom”

Increasing Addresses

Stack Grows Down

Stack “Top”
pushl Src
- Fetch operand at Src
- Decrement %esp by 4
- Write operand at address given by %esp
IA32 Stack: Pop

Stack Pointer: `%esp`

Stack "Top"

Stack "Bottom"

Increasing Addresses

Stack Grows Down

+C4
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>804854e</td>
<td>e8 3d 06 00 00</td>
<td>call 8048b90 &lt;main&gt;</td>
</tr>
<tr>
<td>8048553</td>
<td>50</td>
<td>pushl %eax</td>
</tr>
</tbody>
</table>

- Return address = 0x8048553

- **Procedure return:** `ret`
  - Pop address from stack
  - Jump to address
**Procedure Call Example**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x108</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td>0x8048553</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **%esp**: 0x108
- **%eip**: 0x804854e

---

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

call 8048b90
```

- **%esp**: 0x108
- **%eip**: 0x804854e

---

```
8048553:  50       pushl    %eax
```

- **%esp**: 0x108
- **%eip**: 0x8048b90

---

```
0x8048b90:
```

- **%esp**: 0x104
- **%eip**: 0x8048b90
Procedure Return Example

8048591: c3 ret

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x104</td>
<td>0x8048591</td>
</tr>
<tr>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td>0x10c</td>
</tr>
</tbody>
</table>

%esp: 0x104
%eip: 0x8048591

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
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

Frame Pointer: `%ebp`

Stack Pointer: `%esp`

Previous Frame

Frame for proc

Stack “Top”
```c
yoo(...) {
    •
    •
    who();
    •
    •
}
```
**Example**

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

**Stack**

- `yoo`
- `%ebp`
- `who`
- `%esp`
```plaintext
who(…)
{
  • • • 
amI();
  • • •
ails();
  • • •
}
```
Example

```c
void who() {
    amI(...) {
        amI(...) {
            amI();
        }
    }
}
```

Stack

```
%ebp
%esp
```
Example

```cpp
yoo(
{
  who(...)
  {
    amI(...)
    {
      •
      •
      amI();
      •
      •
    }
  }
}
)
```

Stack

- yoo
- who
- amI
- %ebp
- %esp
```c
void yoo() {
    who(...) {
        • • •
        ami();
        • • •
        ami();
        • • •
    }
}
```
Example

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

Stack

- yoo
- who
- amI
- %ebp
- %esp

Diagram of stack and function calls.
```c
yoo() {
    who(...) {
        • • •
        ami();
        • • •
        ami();
        • • •
    }
}
```
yoo(...) {
    •
    •
    who();
    •
    •
}
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
int course1 = 15213;
int course2 = 18243;

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

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

call_swap:
    ...
    subl $8, %esp
    movl $course2, 4(%esp)
    movl $course1, (%esp)
    call swap
    ...

Calling swap from call_swap

Resulting Stack
Revisiting swap

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

```
swap:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    movl 8(%ebp), %edx
    movl 12(%ebp), %ecx
    movl (%edx), %ebx
    movl (%ecx), %eax
    movl %eax, (%edx)
    movl %ebx, (%ecx)
    popl %ebx
    popl %ebp
    ret
```
swap Setup #1

Entering Stack

Resulting Stack

\[
\begin{align*}
\text{swap:} & \quad \text{pushl } \%\text{ebp} \\
& \quad \text{movl } \%\text{esp}, \%\text{ebp} \\
& \quad \text{pushl } \%\text{ebx}
\end{align*}
\]
swap:

pushl %ebp
movl %esp,%ebp
pushl %ebx
swap:

pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 8(%ebp),%edx  # get xp
movl 12(%ebp),%ecx  # get yp

\[\ldots\]
Observation

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

08048384 <swap>:

```
8048384:  55               push %ebp
8048385:  89 e5           mov %esp,%ebp
8048387:  53               push %ebx
8048388:  8b 55 08         mov 0x8(%ebp),%edx
804838b:  8b 4d 0c         mov 0xc(%ebp),%ecx
804838e:  8b 1a           mov (%edx),%ebx
8048390:  8b 01           mov (%ecx),%eax
8048392:  89 02           mov %eax,(%edx)
8048394:  89 19           mov %ebx,(%ecx)
8048396:  5b             pop %ebx
8048397:  5d             pop %ebp
8048398:  c3             ret
```

Calling Code

```
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
```
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?

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

- Switch statements
- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

pcount_r:
pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    subl $4, %esp
    movl 8(%ebp), %ebx
    movl $0, %eax
    testl %ebx, %ebx
    je .L3
    movl %ebx, %eax
    shrl %eax
    movl %eax, (%esp)
call pcount_r
    movl %ebx, %edx
    andl $1, %edx
    leal (%edx,%eax), %eax
.L3:
    addl $4, %esp
    popl %ebx
    popl %ebp
    ret

_registers

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

pcount_r:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    subl $4, %esp
    movl 8(%ebp), %ebx
    • • •
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}

.. .
movl $0, %eax
testl %ebx, %ebx
je .L3
   .. .
.L3:
   . . .
ret

▶ Actions
   • If x == 0, return
     • with %eax set to 0
```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return
        (x & 1) + pcount_r(x >> 1);
}
```

### Actions
- Store \(x \gg 1\) on stack
- Make recursive call

### Effect
- \(\%eax\) set to function result
- \(\%ebx\) still has value of \(x\)
/* 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 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:
addl$4, %esp
popl%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
Generating Pointer

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

Referencing Pointer

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

- add3 creates pointer and passes it to incrk
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

add3:
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp  # Alloc. 24 bytes
    movl 8(%ebp), %eax
    movl %eax, -4(%ebp) # Set localx to x
## 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`

### Middle part of add3

<table>
<thead>
<tr>
<th>Address (Bytes)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>localx</td>
</tr>
<tr>
<td>-8</td>
<td>Unused</td>
</tr>
<tr>
<td>-12</td>
<td>-4</td>
</tr>
<tr>
<td>-16</td>
<td>0</td>
</tr>
<tr>
<td>-20</td>
<td>-12</td>
</tr>
<tr>
<td>-24</td>
<td>-16</td>
</tr>
</tbody>
</table>

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

%ebp

%esp

%esp+4
int add3(int x) {
    int localx = x;
    incr(&localx, 3);
    return localx;
}

Retrieving local variable

Final part of add3

movl -4(%ebp), %eax
leave
ret

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

**Diagram:**
- **Caller Frame**
  - Arguments
  - Return Addr
  - Saved Registers
  + Local Variables
  - Argument Build
  - %ebp → Old %ebp
  - %esp →