Switch Statements and IA32 Procedures

Spring 2015

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;
}
## Jump Table Structure

### Switch Form

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

### Jump Table

<table>
<thead>
<tr>
<th>jtab:</th>
<th>Targ0</th>
<th>Targ1</th>
<th>Targ2</th>
<th>Targ_{n-1}</th>
</tr>
</thead>
</table>

### 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;
```
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 .L6 # 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]
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 $0 \leq x \leq 6$

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
```
Jump table

```
s.switch .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

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:
    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

```
return w;
```

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

Table entries 64 bits (pointers)

Cases use revised code

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

Jump Table

```assembly
.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
```

```assembly
.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”

Stack Grows Down

Increasing Addresses

Stack “Top”
IA32 Stack: Push

pushl Src
- Fetch operand at Src
- Decrement %esp by 4
- Write operand at address given by %esp

Stack Pointer: %esp

Stack “Top”

Stack “Bottom”

Stack Grows Down

Increasing Addresses
IA32 Stack: Pop

Stack Pointer: %esp +4

Stack "Top"

Stack "Bottom"

Increasing Addresses

Stack Grows Down
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**

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

---

```
call 8048b90
```

```
0x108 0x10c 0x110
%esp  123
%eip  0x804854e
```

```
0x108 0x10c 0x110
%esp  123
%eip  0x8048553
```

```
0x108 0x104
%esp  0x8048553
%eip  0x8048b90
```

---

%esp: stack pointer
%eip: program counter
## Procedure Return Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td>0x104</td>
<td>0x8048553</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td>0x104</td>
<td>0x8048553</td>
</tr>
</tbody>
</table>

%esp: 0x104  %esp: 0x108
%eip: 0x8048591  %eip: 0x8048553
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
Example Call Chain

Procedure \texttt{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
```c
yoo(...) {
    •
    •
    who();
    •
    •
}
```
Example

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

Stack

- `yoo`
- `%ebp`
- `%esp`
- `who`
Example

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

Stack

\[
\begin{align*}
yoo(\ldots) \{ & \text{who(\ldots)} \\
& \{ \text{ami(\ldots)} \\
& \{ \text{ami(\ldots)} \} \} \\
& \text{ami()} \}; \\
& \ldots \\
& \ldots \\
& \} \\
\end{align*}
\]
Example

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

Stack

```
%ebp ➔
%esp ➔
yoo ➔
who ➔
amI ➔
```
Example

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

```
Stack

yoo

%ebp

who

%esp

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

Stack

yoo
who
amI
%ebp
%esp
Example

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

Stack

- yoo
- who
- %ebp
- %esp
```c
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
Revisiting swap

```c
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;
}
```

Calling swap from call_swap

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

Resulting Stack

```
%esp

subl
call

Rtn adr
&course 2
&course 1
%esp
%esp
%esp
```
Revisiting swap

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

```assembly
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
```

Set Up

Body

Finish
**swap Setup #1**

**Entering Stack**

- &course2
- &course1
- Rtn adr

**Resulting Stack**

- %ebp
- %esp

---

**swap:**

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

Entering Stack

Resulting Stack

%ebp

%esp

%ebp

%esp

&course2
&course1
Rtn adr

YP
xp
Rtn adr
Old %ebp

Sungkyunkwan University
swap:

```assembly
pushl %ebp
movl %esp, %ebp
pushl %ebx
```
\textbf{swap BODY}

\begin{itemize}
  \item \textbf{Entering Stack}:
    \begin{itemize}
      \item \&course2
      \item \&course1
      \item Rtn adr
    \end{itemize}

  \item \textbf{Resulting Stack}:
    \begin{itemize}
      \item \%ebp
      \item yp
      \item xp
      \item Rtn adr
    \end{itemize}

\end{itemize}

\textit{Offset relative to \%ebp}:

\begin{itemize}
  \item 12
  \item 8
  \item 4
\end{itemize}

\begin{verbatim}
  movl 8(\%ebp),\%edx  \# get xp
  movl 12(\%ebp),\%ecx  \# get yp
  \ldots
\end{verbatim}
# Swap Finish

**Stack Before Finish**

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

**Resulting Stack**

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

- popl %ebx
- popl %ebp

## 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?

**yoo**:
- ...  
  - movl $15213, %edx  
  - call who  
  - addl %edx, %eax  
  - ...  
  - ret

**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
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);
}

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

```
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
```
/* 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
	ret

.L3:
	ret

Actions

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

---

movl %ebx, %eax
shrl %eax
movl %eax, (%esp)
call pcount_r

---
/* 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

```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`:

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

```
<table>
<thead>
<tr>
<th>Rank</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>x</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>4</td>
<td>Old %ebp</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td>localx</td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-24</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- `%ebp` is used for saving the return address.
- `%esp` is used for saving the stack pointer.
- Local variables are stored on the stack.
- The `incrk` function increments the stack pointer by 4 bytes.

This code snippet demonstrates creating and initializing a local variable `localx` within a function `add3`. The local variable is stored on the stack, and its memory address is computed using the stack pointer. The first part of the `add3` function includes assembly code for saving the stack pointer, allocating memory, and initializing the local variable.
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}

Use leal instruction to compute address of localx

Middle part of add3

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

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td>localx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td>Unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Final part of add3
movl -4(%ebp), %eax  # Return val= localx
leave
ret

Retrieve localx from stack as return value

<table>
<thead>
<tr>
<th>x</th>
<th>Rtn adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>4</td>
<td>%ebp</td>
</tr>
<tr>
<td>0</td>
<td>Unused</td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td></td>
</tr>
<tr>
<td>-12</td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td></td>
</tr>
<tr>
<td>-20</td>
<td></td>
</tr>
<tr>
<td>-24</td>
<td></td>
</tr>
</tbody>
</table>
### 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