CSCI-UA.0201-003

Computer Systems Organization

Lecture 10: Machine-Level Programming III: IA32 Procedures

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
Switch uses jump tables

Switch statement

```
switch(x) {
  case 0:
    Block 0
  case 1:
    Block 1
  • • •
  case n-1:
    Block n-1
}
```

Jump Table

- jtab: Targ0
- Targ1
- • • •
- Targn-1

Jump Targets

- Targ0: Code Block 0
- Targ1: Code Block 1
- ••
- Targn-1: Code Block n-1

Approximate Translation

```
target = JTab[x];
goto *target;
```
Switch Statement Example (IA32)

```c
long switch_eg(long x, long y, long z) {
    switch(x) {
    case 1: ....
    default: ....
    }
}
```

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]
...
```
Switch Statement Example (IA32)

```
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]
```

Indirect jump to effective address
.L7 + eax*4

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

Read Only

Base address
Target addr 4 bytes
Jump Table

Jump table

```assembly
.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
```

```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:      // .L5
    case 6:      // .L6
        w -= z;
        break;
    default:     // .L2
        w = 2;
}
```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>signed-ness</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>JNO</td>
<td>Jump if not overflow</td>
<td></td>
<td>OF = 0</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if sign</td>
<td></td>
<td>SF = 1</td>
</tr>
<tr>
<td>JNS</td>
<td>Jump if not sign</td>
<td></td>
<td>SF = 0</td>
</tr>
<tr>
<td>JE</td>
<td>Jump if equal</td>
<td></td>
<td>ZF = 1</td>
</tr>
<tr>
<td>JZ</td>
<td>Jump if zero</td>
<td></td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JNE</td>
<td>Jump if not equal</td>
<td></td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JNZ</td>
<td>Jump if not zero</td>
<td></td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if below</td>
<td></td>
<td>CF = 1</td>
</tr>
<tr>
<td>JNAE</td>
<td>Jump if not above or equal</td>
<td></td>
<td>CF = 0</td>
</tr>
<tr>
<td>JC</td>
<td>Jump if carry</td>
<td></td>
<td>CF = 0</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if not below</td>
<td></td>
<td>CF = 0</td>
</tr>
<tr>
<td>JAE</td>
<td>Jump if above or equal</td>
<td></td>
<td>CF = 0</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if not carry</td>
<td></td>
<td>CF = 0</td>
</tr>
<tr>
<td>JBE</td>
<td>Jump if below or equal</td>
<td></td>
<td>CF = 1 or ZF = 1</td>
</tr>
<tr>
<td>JNA</td>
<td>Jump if not above</td>
<td></td>
<td>CF = 1 or ZF = 1</td>
</tr>
<tr>
<td>JA</td>
<td>Jump if above</td>
<td></td>
<td>CF = 0 and ZF = 0</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if not below or equal</td>
<td></td>
<td>CF = 0 and ZF = 0</td>
</tr>
<tr>
<td>JL</td>
<td>Jump if less</td>
<td>signed</td>
<td>SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JNGE</td>
<td>Jump if not greater or equal</td>
<td></td>
<td>SF = OF</td>
</tr>
<tr>
<td>JGE</td>
<td>Jump if greater or equal</td>
<td>signed</td>
<td>SF = OF</td>
</tr>
<tr>
<td>JNL</td>
<td>Jump if not less</td>
<td>signed</td>
<td>ZF = 1 or SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JLE</td>
<td>Jump if not greater</td>
<td>signed</td>
<td>ZF = 1 or SF &lt;&gt; OF</td>
</tr>
<tr>
<td>JNG</td>
<td>Jump if less or equal</td>
<td>signed</td>
<td>ZF = 0 and SF = OF</td>
</tr>
<tr>
<td>JNL</td>
<td>Jump if not less or equal</td>
<td>signed</td>
<td>ZF = 0 and SF = OF</td>
</tr>
<tr>
<td>JF</td>
<td>Jump if parity</td>
<td></td>
<td>PF = 1</td>
</tr>
<tr>
<td>JFE</td>
<td>Jump if parity even</td>
<td></td>
<td>PF = 0</td>
</tr>
<tr>
<td>JNP</td>
<td>Jump if not parity</td>
<td></td>
<td>PF = 0</td>
</tr>
<tr>
<td>JPO</td>
<td>Jump if parity odd</td>
<td></td>
<td>PF = 0</td>
</tr>
<tr>
<td>JCEZ</td>
<td>Jump if %CX register is 0</td>
<td></td>
<td>%CX = 0</td>
</tr>
<tr>
<td>JECXZ</td>
<td>Jump if %ECX register is 0</td>
<td></td>
<td>%ECX = 0</td>
</tr>
</tbody>
</table>

Source: [http://www.unixwiz.net/techtips/x86-jumps.html](http://www.unixwiz.net/techtips/x86-jumps.html)
IA32 Object Code

Assembly Code

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

Disassembled Object Code

08048410 <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, inspect using GDB
  – (gdb) x/7xw 0x8048660

• Examine 7 hexadecimal format “words” (4-bytes each)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048660</td>
<td>0x8048422</td>
<td>0</td>
</tr>
<tr>
<td>0x8048664</td>
<td>0x8048432</td>
<td>1</td>
</tr>
<tr>
<td>0x8048668</td>
<td>0x804843b</td>
<td>2</td>
</tr>
<tr>
<td>0x804866c</td>
<td>0x8048429</td>
<td>3</td>
</tr>
<tr>
<td>0x8048670</td>
<td>0x8048422</td>
<td>4</td>
</tr>
<tr>
<td>0x8048674</td>
<td>0x804844b</td>
<td>5</td>
</tr>
<tr>
<td>0x8048678</td>
<td>0x804844b</td>
<td>6</td>
</tr>
</tbody>
</table>
Matching Disassembled Targets

<table>
<thead>
<tr>
<th>Value</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048422</td>
<td>mov $0x2,%eax</td>
<td>8048422</td>
</tr>
<tr>
<td>0x8048432</td>
<td>jmp 8048453 &lt;switch_eg+0x43&gt;</td>
<td>8048427</td>
</tr>
<tr>
<td>0x804843b</td>
<td>mov $0x1,%eax</td>
<td>8048429</td>
</tr>
<tr>
<td>0x8048429</td>
<td>xchg %ax,%ax</td>
<td>804842e</td>
</tr>
<tr>
<td>0x8048430</td>
<td>jmp 8048446 &lt;switch_eg+0x36&gt;</td>
<td>8048432</td>
</tr>
<tr>
<td>0x804843b</td>
<td>mov 0x10(ebp),%eax</td>
<td>8048435</td>
</tr>
<tr>
<td>0x8048439</td>
<td>imul 0xc(ebp),%eax</td>
<td>8048439</td>
</tr>
<tr>
<td>0x804844b</td>
<td>jmp 8048453 &lt;switch_eg+0x43&gt;</td>
<td>804843b</td>
</tr>
<tr>
<td>0x8048433</td>
<td>mov 0xc(ebp),%edx</td>
<td>804843b</td>
</tr>
<tr>
<td>0x804843e</td>
<td>mov %edx,%eax</td>
<td>804843e</td>
</tr>
<tr>
<td>0x8048440</td>
<td>sar $0x1f,%edx</td>
<td>8048440</td>
</tr>
<tr>
<td>0x8048443</td>
<td>idivl 0x10(ebp)</td>
<td>8048443</td>
</tr>
<tr>
<td>0x8048446</td>
<td>add 0x10(ebp),%eax</td>
<td>8048446</td>
</tr>
<tr>
<td>0x8048449</td>
<td>jmp 8048453 &lt;switch_eg+0x43&gt;</td>
<td>8048449</td>
</tr>
<tr>
<td>0x804844b</td>
<td>mov $0x1,%eax</td>
<td>804844b</td>
</tr>
<tr>
<td>0x8048450</td>
<td>sub 0x10(ebp),%eax</td>
<td>8048450</td>
</tr>
<tr>
<td>0x8048453</td>
<td>pop %ebp</td>
<td>8048453</td>
</tr>
<tr>
<td>0x8048454</td>
<td>ret</td>
<td>8048454</td>
</tr>
</tbody>
</table>
IA32 Stack

- Region of memory managed with stack discipline (i.e. LIFO)
- Register %esp contains address of “top” element

Stack Pointer: %esp

Stack Grows Down

Increasing Addresses

Bottom

Top
IA32 Stack: Push

• `pushl src`
  1. Fetch operand at `Src`
  2. Decrement `%esp` by 4
  3. Write operand at address given by `%esp`

Stack Pointer: `%esp`
IA32 Stack: Pop

- `popl dst`
  1. Write operand at address given by `%esp` to `dst`
  2. Increment `%esp` by 4

Stack Pointer: `%esp`
Procedure Control Flow

• Use stack to support procedure call and return
  – call *label_of_procedure*
    • Push return address on stack
    • Jump to *label*
  – *ret*
    • Pop address from stack
    • Jump to address

• Return address:
  – Address of the next instruction right after call
  – Example:
    
```
804854e:   e8 3d 06 00 00    call   8048b90 <main>
8048553:   50
```
  – Return address = 0x8048553
Procedure Call Example

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

0x8048b90:

call 8048b90

0x110
0x10c
0x108 123

%esp 0x108
%eip 0x804854e

0x110
0x10c
0x108 123
0x104 0x8048553

%esp 0x104
%eip 0x8048b90

%eip: program counter
# Procedure Return Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x104</td>
<td>c3</td>
<td>0x8048553</td>
<td>ret</td>
</tr>
<tr>
<td>0x108</td>
<td>123</td>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td>0x110</td>
<td></td>
<td>0x104</td>
<td>0x8048553</td>
</tr>
</tbody>
</table>

%eip: program counter
Using stack for procedure invocation

• In addition to return address, stack also stores
  – Arguments
  – local variables
  – Scratch space

• Stack allocated in *Frames*
  – state for single procedure instantiation

• Stack is well suited for procedure invocation
  – State for given procedure needed during procedure execution (from call to return).
  – Callee returns before caller does (last-in-first-out)
Call Chain Example

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
Example

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

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

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

Stack

```plaintext
yoo(...) {
    who(...) {
        amI(...) {
            amI(...) {
                amI();
                ...;
            }
            ...>
        }
        ...;
    }
    ...;
}
```

```plaintext
yoo

who

amI

%ebp

%esp
```
Example

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

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

%ebp

%esp

Stack

yoo

who

amI

amI

amI

amI

amI
```
Example

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

Stack

```
%ebp
%esp
```

```
yoo
 who
 amI
```
Example

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

$ who(...) {
  ...
  amI();
  ...
}
```

Stack:
- yoo
- who

Variables:
- %ebp
- %esp
Example

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

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

```c
int course1 = 15213;
int course2 = 18243;

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

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

swap:

```
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
swap Setup #2

Entering Stack

Resulting Stack

\[
\text{swap:} \quad 
\begin{align*}
\text{pushl} & \quad \%\text{ebp} \\
\text{movl} & \quad \%\text{esp}, \%\text{ebp} \\
\text{pushl} & \quad \%\text{ebx}
\end{align*}
\]
swap Setup #3

Entering Stack

\[
\begin{array}{c}
& \text{\&\text{course2}} \\
& \text{\&\text{course1}} \\
\text{Rtn adr} & \leftarrow \text{%esp} \\
\end{array}
\]

Resulting Stack

\[
\begin{array}{c}
\text{YP} \\
\text{xp} \\
\text{Rtn adr} \\
\text{Old \%ebp} \\
\text{Old \%ebx} \\
\end{array}
\]

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

Entering Stack

Resulting Stack

\[
\begin{align*}
\text{movl} & \ 8(\%ebp),\%edx & \# \text{get xp} \\
\text{movl} & \ 12(\%ebp),\%ecx & \# \text{get yp} \\
\ldots
\end{align*}
\]

Offset relative to %ebp

%ebp

%esp

%ebp

%esp

%ebp
Carnegie Mellon

swap Finish

Stack Before Finish

Resulting Stack

Saved and restored register %ebx, but not %eax, %ecx, %edx
Register Saving Conventions

- When procedure *yoo* calls *who*:
  - *yoo* is the *caller*
  - *who* is the *callee*

- Callee might use registers for temporary storage?

  ![Code Snippet](image)

  - Contents of register `%edx` overwritten by *who*
  - This could be trouble ➔ something should be done!
    - Need some coordination
Register Saving Conventions

• Register saving Conventions
  – “Caller Save”
    • Caller saves temporary values in its stack frame before the call
  – “Callee Save”
    • Callee saves temporary values in its stack frame before using
# IA32/Linux+Windows Register Usage

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax, %edx, %ecx</td>
<td>Caller saves prior to call if values are used later</td>
</tr>
<tr>
<td>%eax</td>
<td>also used to return integer value</td>
</tr>
<tr>
<td>%ebx, %esi, %edi</td>
<td>Callee saves if wants to use them</td>
</tr>
<tr>
<td>%esp, %ebp</td>
<td>special form of callee save</td>
</tr>
<tr>
<td>%esp</td>
<td>Restored to original values upon exit from procedure</td>
</tr>
</tbody>
</table>

### Caller-Save Temporaries
- %eax
- %edx
- %ecx

### Callee-Save Temporaries
- %ebx
- %esi
- %edi

### Special
- %esp
- %ebp
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

```
pushl %ebp
movl %esp,%ebp
pushl %ebx
subl $4,%esp
movl 8(%ebp),%ebx
movl $0,%eax
cmpl%eax,%ebx
je .L3
movl %ebx,%eax
shrl $1,%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 Call #1

/* 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);
}

• Actions
- If x == 0 (%eax),
  return

  %ebx  x

• • •
  movl  $0,%eax
cmpl  %eax,%ebx
  je    .L3
  • • •
  .L3:
  • • •
  ret
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
- **Effect after calling `pcount_r`**
  - `%eax` set to function result
  - `%ebx` still has value of `x`

```
movl %ebx,%eax
shrl $1,%eax
movl %eax,(%esp)
call pcount_r
```

```
%ebp %ebp
%esp %esp
%ebx x >> 1
%eax
```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return (x & 1) + pcount_r(x >> 1);
}

• After calling pcount_r
  – %eax holds value from recursive call
  – %ebx holds x

• Actions
  – Compute (x & 1) + computed value

• Effect
  – %eax set to function result

movl %ebx,%edx
andl $1,%edx
leal (%edx,%eax),%eax

• • •
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 %ebx,%ebp
  - Restore %esp

L3:
  ...%ebp
  addl $4,%esp
  popl %ebx
  popl %ebp
  ret

Rtn adr
Old %ebp
Old %ebx
%esp
%ebp
%esp
%ebp
%ebx
Old %ebx
/* Compute x + 3 */
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}

/* 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
- 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

- Use leal instruction to compute address of localx

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

Middle part of add3

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

- Retrieve localx from stack as return value

```c
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
```

```
movl %ebp, %esp
popl %ebp
```
Conclusions

• Important Points
  – Stack is the data structure for procedure invocation
    • If P calls Q, then Q returns before P
• Recursion handled by normal calling conventions
• Pointers are addresses of values
  – On stack or elsewhere (global, heap)