Machine-Level Programming III: Switch Statements and IA32 Procedures

CS 485: Systems Programming
Fall 2015

Instructor:
James Griffioen

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/public/instructors.html)
Today

- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers
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
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 ();
    •
    •
}
```

Stack

```
%ebp
%esp
yoo
```

Diagram showing stack allocation and function call structure.
Example

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

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

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

who() {
    ami();
}

ami() {
    ami();
    ami();
    ami();
}
```

Stack

- yoo
- who
- ami
- ami
- %ebp
- %esp
Example

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

Stack

```c
tack
```
Example

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

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

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

Stack

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

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

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

Stack

```
Stack:

%ebp

yoo

who

amI

%esp

amI
```
Example

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

Stack

```
yoo
%ebp

who
%esp

amI
amI
amI
```
Example

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

void who(...)
{
    • • •
    ami()
    • • •
}

void ami(...)
{
    •
    •
}
```

Stack
```
%ebp
%esp
yoo
who
ami
```
Example

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

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

Stack

```
%ebp

yoo

who

%esp
```

...
Example

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

Stack

```c
+yoo
%ebp
%esp
```
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 “Top”

Increasing Addresses

Stack Grows Down
IA32 Stack: Push

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

Stack Pointer: $\%esp$
IA32 Stack: Pop

Stack Pointer: %esp

Stack "Top"

Stack "Bottom"

Increasing Addresses

Stack Grows Down

%esp
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

Before Call

<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>
</tbody>
</table>

%esp 0x108
%eip 0x804854e

call 8048b90

After Call

<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
%eip 0x8048b90

call 8048b90
Procedure Return Example

Before ret

<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
%eip: 0x8048591

After ret

<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: 0x108
%eip: 0x8048553

%eip: program counter
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

Diagram:
- Caller Frame
  - Arguments
  - Return Addr
  - Old %ebp
  - Saved Registers + Local Variables
  - Argument Build
- Frame pointer %ebp
- Stack pointer %esp
Revisiting swap

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

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

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

Calling swap from call_swap

```
Resulting Stack

<table>
<thead>
<tr>
<th>%esp</th>
<th>subl</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>call</td>
</tr>
</tbody>
</table>

| &course 2 | |
| &course 1 | |
| Rtn adr | |
```
Revisiting swap

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

**swap**: 
- **Set Up**
  - `pushl %ebp`
  - `movl %esp, %ebp`
  - `pushl %ebx`
- **Body**
  - `movl 8(%ebp), %edx`
  - `movl 12(%ebp), %ecx`
  - `movl (%edx), %ebx`
  - `movl (%ecx), %eax`
  - `movl %eax, (%edx)`
  - `movl %ebx, (%ecx)`
- **Finish**
  - `popl %ebx`
  - `popl %ebp`
  - `ret`
swap Setup #1

Entering Stack

Resulting Stack

---

swap:

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

Entering Stack

\[
\begin{array}{c}
\vdots \\
\vdots \\
\vdots \\
\text{\&course2} \\
\text{\&course1} \\
\text{Rtn adr} \\
\end{array}
\]

Resulting Stack

\[
\begin{array}{c}
\vdots \\
\vdots \\
\vdots \\
\text{yp} \\
\text{xp} \\
\text{Rtn adr} \\
\text{Old \&ebp} \\
\end{array}
\]

\[
\text{swap:} \\
\quad \text{pushl \&ebp} \\
\quad \text{movl \&esp,\&ebp} \\
\quad \text{pushl \&ebx}
\]
swap Setup #3

Entering Stack

\[
\begin{array}{c}
\vdots \\
\vdots \\
\vdots \\
& \text{\&course2} \\
& \text{\&course1} \\
\text{Rtn adr} \\
\end{array}
\]

%ebp
%esp

Resulting Stack

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

%ebp
%esp

\[
swap: \\
pushl %ebp \\
movl %esp,%ebp \\
pushl %ebx
\]
**swap Body**

### Entering Stack

- `&course2`
- `&course1`
- `Rtn adr`

### Resulting Stack

- `%ebp`
- `%esp`

**Offset relative to %ebp**

- `%ebp`: 4
- `%esp`: 8
- `&course2`: 12

**Code**

```
movl 8(%ebp),%edx  # get xp
movl 12(%ebp),%ecx  # get yp
...  
```
swap Finish

Stack Before Finish

Resulting Stack

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

08048384 <swap>:

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

Calling Code

80483b4:  \text{movl} \ $0x8049658,0x4(\%esp) \ # \ Copy \ &\text{course2}
80483bc:  \text{movl} \ $0x8049654,(\%esp) \ # \ Copy \ &\text{course1}
80483c3:  \text{call} \ 8048384 \ <\text{swap}> \ # \ Call \ swap
80483c8:  \text{leave} \ # \ Prepare \ to \ return
80483c9:  \text{ret} \ # \ Return
Today

- **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?

```
\begin{tabular}{ll}
\textbf{yoo:} & \textbf{who:} \\
    & \\
    & movl $15213, \%edx \\
    & call who \\
    & addl \%edx, \%eax \\
    & \ldots \\
    & ret \\
    & \\
    & movl 8(\%ebp), \%edx \\
    & addl $18243, \%edx \\
    & \ldots \\
    & ret \\
\end{tabular}
```

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

- When procedure *you* calls *who*:
  - *you* 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
X86-64 Registers – differences from IA32

- **Register Names:**
  - Register names start with `r` instead of `e`. For example the `%eax` register becomes `%rax`

- **New Registers:**
  - There are 8 additional registers identified as `%r8`, `%r9`, `%r10`, ..., `%r15`.

- **Working with 8, 16, and 32 bits:**
  - To only use part of a register you can add a suffix to a register name. For example `%r8d` refers to the low order 32 bits. Other suffixes are `b` (8 bits), `w` (16 bits), and `d` (32 bits).

- **Passing Parameters:**
  - The first (up to) 6 arguments to a procedure are passed in registers, not on the stack. In particular, they are passed in reverse order (i.e., left to right in the function call) in registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`. If there are more than 6 parameters, they are passed on the stack in call order (from right to left).

- **Calling Conventions:**
  - Registers `%rbp`, `%rbx`, `%r12`, `%r13`, `%r14`, and `%r15` are callee-saves registers. All other registers are caller-saves.
Today

- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers
Recursive Function

/* 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 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
  ...

\[
\begin{array}{c}
\%ebx \\
x \\
Rtn adr \\
Old %ebp \\
Old %ebx
\end{array}
\]
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

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

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

```assembly
... 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 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
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

```
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

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

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

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

- Retrieve localx from stack as return value

Final part of add3

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