Assembly III: Procedures

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
IA-32 Stack (1)

- **Characteristics**
  - Region of memory managed with stack discipline
  - Grows toward lower addresses
  - Register `%esp` indicates lowest stack address
    - address of top element

![Stack Diagram]

- Stack Pointer `%esp`
- Stack “Bottom”
- Increasing Addresses
- Stack Grows Down
- Stack “Top”
IA-32 Stack (2)

- **Pushing**
  - `pushl Src`
  - Fetch operand at `Src`
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`
IA-32 Stack (3)

- **Popping**
  - `popl Dest`
  - Read operand at address given by `%esp`
  - Increment `%esp` by 4
  - Write to `Dest`
IA-32 Stack (4)

- Stack operation examples

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

pushl %eax

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x108</td>
<td>213</td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
</tbody>
</table>

popl %edx

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

```
<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>213</td>
</tr>
<tr>
<td>%edx</td>
<td>555</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

%esp

%esp

%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 value
    - Address of instruction beyond call

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

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

- `0x8048553` is the address of the pushl instruction.
- `0x8048b90` is the address of the main function.

#### Example Code

```assembly
804854e: `e8 3d 06 00 00`
8048553: `50`
```

#### Memory Stack

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

- `%esp` points to the top of the stack at `0x108`.
- `%eip` points to the address `0x804854e`.
- `%eip` is the program counter.

---

SSE2030: Introduction to Computer Systems | Fall 2012 | Jin-Soo Kim (jinsookim@skku.edu)
Procedure Return Example

8048591:  c3  ret

%esp  0x104
%eip  0x8048591

%esp  0x104
%eip  0x8048591

%esp  0x108
%eip  123

%esp  0x108
%eip  123

ret

%esp  0x10c
%eip  0x110

%esp  0x110
%eip  0x10c

0x8048553

%eip is program counter
Stack-based Languages

- Languages that support recursion
  - e.g., C, Pascal, Java, etc.
  - 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 (1)

Code Structure

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

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

amI(...) {
  •
  •
  amI();
  •
}
```

- Procedure `amI` recursive

Call Chain

```plaintext
yoo
  └─ who
    └─ amI
      └─ amI
```

SSE2030: Introduction to Computer Systems  |  Fall 2012  |  Jin-Soo Kim (jinsookim@skku.edu)
Stack Frames (2)

- **Contents**
  - Return information
  - Arguments
  - Local variables & temp space

- **Management**
  - Space allocated when enter procedure  
    - "set-up" code
  - Deallocated when return  
    - "finish" code

- **Pointers**
  - Stack pointer `%esp` indicates stack top
  - Frame pointer `%ebp` indicates start of current frame
Stack Frames (3)

Call Chain

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

Frame Pointer
%ebp

Stack Pointer
%esp

yoo
Stack Frames (4)

Call Chain

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

Frame Pointer
%ebp

Stack Pointer
%esp

yoo

who

yoo
Stack Frames (5)

Call Chain

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

```
Frame Pointer %ebp
Stack Pointer %esp
```

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

```
who
```

```
yoo
```

```
: ...
```
Stack Frames (6)

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

Call Chain

```
Call Chain
```

```
\[
\begin{align*}
\text{amI}() & \rightarrow \text{who} \rightarrow \text{yoo} \\
\text{yoo} & \rightarrow \text{Frame Pointer} \rightarrow \%\text{ebp} \\
\text{who} & \rightarrow \text{Frame Pointer} \rightarrow \%\text{esp} \\
\text{amI} & \\
\text{amI} & \\
\end{align*}
\]
```
Stack Frames (7)

Call Chain

amI(...) {
  •
  •
  amI();
  •
  •
}

Frame Pointer
%ebp

Stack Pointer
%esp

yoo
who
amI
amI
Stack Frames (8)

Call Chain

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

Stack Pointer
%esp

Frame Pointer
%ebp

yoo
who
amI
amI
amI

Stack Frames (9)

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

Call Chain

```
yoo
  who
  amI
  amI
```

Frame Pointer %ebp
Stack Pointer %esp

```
Stack Frames (10)

Call Chain

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

Frame Pointer
%ebp

Stack Pointer
%esp

yoo

who

amI

amI

amI

yoo

who
Stack Frames (11)

Call Chain

amI(...) {
  •
  •
  •
}

Stack Pointer
%esp

Frame Pointer
%ebp

yoo
who
amI
amI
amI
amI

Stack Pointer
%esp
Stack Frames (12)

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

Call Chain

yoo

who

amI

amI

amI

Stack Pointer
%esp

Frame Pointer
%ebp

who

yoo
Stack Frames (13)

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

Call Chain

- `who();`
- `amI`
- `amI`
- `amI`
- `amI`
- `amI`
- `yoo`

Frame Pointer `%ebp`

Stack Pointer `%esp`
IA-32/Linux Stack Frame

- **Current stack frame (“Top” to Bottom)**
  - Parameters for function about to call
    - “Argument build”
  - 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 (1)

```c
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
    swap(&zip1, &zip2);
}

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

Calling swap from call_swap

call_swap:
    • • •
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
    • • •

Resulting Stack

```plaintext
<table>
<thead>
<tr>
<th>Rtn adr</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;zip2</td>
<td></td>
</tr>
<tr>
<td>&amp;zip1</td>
<td></td>
</tr>
</tbody>
</table>
```

Calling from call_swap
Revisiting swap (2)

```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 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Body

Setup

Finish
Swap Setup (1)

- **Entering Stack**
  - %ebp
  - &zip2
  - &zip1
  - Rtn adr

- **Resulting Stack**
  - %ebp
  - yp
  - xp
  - Rtn adr
  - Old %ebp

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

```assembly
swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
Swap Setup (2)

**Entering Stack**

- `%ebp`
- `&zip2`
- `&zip1`
- `Rtn adr`

**Resulting Stack**

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

---

**Code Snippet:**

```
swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
Swap Setup (3)

Entering Stack

\[
\begin{align*}
& \bullet \\
& \bullet \\
& \bullet \\
& & & \%ebp \\
& & & \%esp \\
& & & \%ebp \\
& & & \%esp \\
& & & \%ebp \\
& & & \%esp \\
\end{align*}
\]

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

Resulting Stack

\[
\begin{align*}
& \bullet \\
& \bullet \\
& \bullet \\
& & & \%ebp \\
& & & \%esp \\
& & & \%ebp \\
& & & \%esp \\
& & & \%ebp \\
& & & \%esp \\
& & & \%ebp \\
& & & \%esp \\
& & \text{Rtn adr} \\
& & \text{Rtn adr} \\
& & \text{Rtn adr} \\
& & \text{Rtn adr} \\
& & \text{Rtn adr} \\
& & \text{Rtn adr} \\
& & \text{Rtn adr} \\
& & \text{Rtn adr} \\
& & \text{Rtn adr} \\
& & \text{Old } \%ebp \\
& & \text{Old } \%ebp \\
\end{align*}
\]
Effect of swap Setup

Entering Stack

Resulting Stack

Body

movl 12(%ebp),%ecx  # get yp
movl 8(%ebp),%edx  # get xp
...
• **Observation**
  - Saved & restored register `%ebx`
swap Finish (2)

swap’s Stack

Offset

12 yp
8 xp
4 Rtn adr
0 Old %ebp
-4 Old %ebx

movl –4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
swap Finish (3)

swap’s Stack

```
        movl -4(%ebp),%ebx
        movl %ebp,%esp
        popl %ebp
        ret
```

Old %ebp

Old %ebx

Offset

-4

0

4

8

12
**Observation**

- Saved & restored register `%ebx`
- Didn’t do so for `%eax`, `%ecx`, or `%edx`
Register Saving Conventions (1)

- 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 $91125, %edx
  • • •
  ret
```

- Contents of register `%edx` overwritten by `who`
Register Saving Conventions (2)

- Conventions
  - “Caller save”
    - Caller saves temporary in its frame before calling
  - “Callee save”
    - Callee saves temporary in its frame before using
IA-32/Linux Register Usage

- **Integer registers**
  - Two have special uses:
    - `%ebp`, `%esp`
  - Three managed as callee-save:
    - `%ebx`, `%esi`, `%edi`
    - Old values saved on stack prior to using
  - Three managed as caller-save:
    - `%eax`, `%edx`, `%ecx`
    - Do what you please, but expect any callee to do so, as well
  - Register `%eax` also stores returned value
Recursive Factorial: rfact

- Registers
  - `%eax` used without first saving
  - `%ebx` used, but save at beginning & restore at end

```c
int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}
```

```
rfact:
pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```
rfact Stack Setup

Entering Stack

```
rfact:
pushl %ebp
movl %esp,%ebp
pushl %ebx
```

SSE2030: Introduction to Computer Systems | Fall 2012 | Jin-Soo Kim (jinsookim@skku.edu)
### Registers

- **%ebx**: stored value of \( x \)
- **%eax**
  - Temporary value of \( x-1 \)
  - Returned value from \( \text{rfact}(x-1) \)
  - Returned value from this call

### Recursion

```assembly
movl 8(%ebp),%ebx       # ebx = x
cmp $1,%ebx             # Compare x : 1
jle .L78                # If <= goto Term
leal -1(%ebx),%eax     # eax = x - 1
pushl %eax              # Push x - 1
call rfact             # rfact(x - 1)
imull %ebx,%eax        # rval * x
jmp .L79                # Goto done
.L78:
    mov $1,%eax        # Term:
    jmp .L79           # return val = 1
.L79:
```

```c
int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x - 1);
    return rval * x;
}
```
rfact Recursion

leal -1(%ebx),%eax

pushl %eax

call rfact

x
Rtn adr
Old %ebp
Old %ebx

%eax
x-1
%ebx
x

%eax
x-1
%ebx
x

%eax
x-1
%ebx
x

%eax
x-1
%ebx
x

%eax
x-1
%ebx
x

%eax
x-1
%ebx
x

%eax
x-1
%ebx
x

%eax
x-1
%ebx
x

%eax
x-1
%ebx
x
Assume that \( \text{rfact}(x-1) \) returns \( (x-1)! \) in register \%eax.
rfact Completion

![Diagram of rfact Completion]

```assembly
# rfact Completion

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
Summary

- **The stack makes recursion work**
  - Private storage for each instance of procedure call
    - Instantiations don’t clobber each other
    - Addressing of locals + arguments can be relative to stack positions
  - Can be managed by stack discipline
    - Procedures return in inverse order of calls

- **Procedures = Instructions + Conventions**
  - Call / Ret instructions
  - Register usage conventions
    - Caller / Callee save
    - %ebp and %esp
  - Stack frame organization conventions