Machine Representation of Programs: Procedures

Instructors: Sanjeev Setia

IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register $\%\text{esp}$ contains lowest stack address = address of “top” element

Stack Pointer: $\%\text{esp}$

Stack “Bottom”

Increasing Addresses

Stack Grows Down

Stack “Top”
### IA32 Stack: Push

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

![Stack Diagram for Push](image)

- **Stack Pointer:** `%esp`
- **Stack “Bottom”**
- **Stack “Top”**
- **Increasing Addresses**
- **Stack Grows Down**

### IA32 Stack: Pop

- **popl** *Dest*
  - Read operand at address `%esp`
  - Increment `%esp` by 4
  - Write operand to *Dest*

![Stack Diagram for Pop](image)

- **Stack Pointer:** `%esp`
- **Stack “Bottom”**
- **Stack “Top”**
- **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 instruction beyond 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
```

%esp: 0x108
%esp: 0x104
%esp: 0x804854e
%esp: 0x8048553
%esp: 0x8048b90
%eip: program counter
Procedure Return Example

8048591: c3 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
**Call Chain Example**

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

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

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

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();
    •
    •
}
```

Example

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

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

Example

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

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

Stack

```
%ebp
%esp
```

Example

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

Stack

```
%ebp
%esp
```
Example

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

Example

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

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

Example

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

```c
yoo(...) {
    •
    •
    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
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

```c
call_swap:
    * * *
    pushl $zip2  # Global Var
    pushl $zip1  # Global Var
    call swap
    * * *
```

Resulting Stack

```
        %esp
global Var
        %esp
        15213
        91125
Rtn adr
```

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 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
```
swap Setup #1

Entering Stack

\[
\begin{array}{c}
\vdots \\
\%ebp \\
%esp \\
&\text{zip2} \\
&\text{zip1} \\
\text{Rtn adr}
\end{array}
\]

Resulting Stack

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

\text{swap:}
\begin{align*}
\text{pushl}\ %\text{ebp} \\
\text{movl}\ %\text{esp},%\text{ebp} \\
\text{pushl}\ %\text{ebx}
\end{align*}
**swap Setup #1**

**Entering Stack**

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

**Resulting Stack**

- `%ebp`
- `%esp`
- `%zip2`
- `%zip1`
- `Rtn adr`
- `Old %ebp`

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

---

**swap Setup #1**

**Entering Stack**

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

**Resulting Stack**

- `%ebp`
- `%esp`
- `%zip2`
- `%zip1`
- `Rtn adr`
- `Old %ebp`

```
swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
### swap Setup #1

**Entering Stack**

- %ebp

**Resulting Stack**

- %ebp


text:

- \(\text{movl } 12(\%ebp), \%ecx \# \text{ get } yp\)
- \(\text{movl } 8(\%ebp), \%edx \# \text{ get } xp\)

### swap Finish #1

**swap’s Stack**

- \(\text{movl } -4(\%ebp), \%ebx\)
- \(\text{movl } \%ebp, \%esp\)
- \(\text{popl } \%ebp\)
- \(\text{ret}\)

**Resulting Stack**

- \(\text{Observation: Saved and restored register } \%ebx\)
swap Finish #2

swap’s Stack

Resulting Stack

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
swap Finish #2

swap’s Stack

\[
\begin{align*}
\text{Old } %ebp & \quad \text{Old } %ebx \\
%ebp & \quad %esp \\
\text{xp} & \quad Rtn \text{ adr} \\
\text{YP} & \\
\ldots &
\end{align*}
\]

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

swap Finish #3

Resulting Stack

\[
\begin{align*}
%ebp & \quad %esp \\
\text{xp} & \quad Rtn \text{ adr} \\
\text{YP} & \\
\ldots &
\end{align*}
\]
swap Finish #4

• swap’s Stack

• • •

| YP | XP |
|-----------------|

| Rtn adr |

| Old %ebp |

| Old %ebx |

• %ebp

• %esp

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

• Observation
- Saved & restored register %ebx
- Didn’t do so for %eax, %ecx, or %edx
Disassembled swap

80483a4 <swap>:
80483a4:   55          push   %ebp
80483a5:   89 e5       mov    %esp,%ebp
80483a7:   53          push   %ebx
80483a8:   8b 55 08    mov    0x8(%ebp),%edx
80483ab:   8b 4d 0c    mov    0xc(%ebp),%ecx
80483ae:   8b 1a       mov    (%edx),%ebx
80483b0:   8b 01       mov    (%ecx),%eax
80483b2:   89 02       mov    %eax,(%edx)
80483b4:   89 19       mov    %ebx,(%ecx)
80483b6:   5b          pop    %ebx
80483b7:   c9          leave
80483b8:   c3          ret

Calling Code
8048409:   e8 96 ff ff ff   call 80483a4 <swap>
804840e:   8b 45 f8         mov  0xfffffff8(%ebp),%eax

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 $91125, %edx
  - • • •
    ret

- Contents of register %edx overwritten by who
Register Saving Conventions

- When procedure \textit{yoo} calls \textit{who}:
  - \textit{yoo} is the \textit{caller}
  - \textit{who} is the \textit{callee}

- Can register be used for temporary storage?
- Conventions
  - "\textit{Caller Save}"
    - Caller saves temporary in its frame before calling
  - "\textit{Callee Save}"
    - Callee saves temporary in its frame before using

IA32/Linux Register Usage

- \texttt{%eax, %edx, %ecx}
  - Caller saves prior to call if values are used later
- \texttt{%eax}
  - also used to return integer value
- \texttt{%ebx, %esi, %edi}
  - Callee saves if wants to use them
- \texttt{%esp, %ebp}
  - special
Recursive Factorial

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

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

Pointer Code

**Recursive Procedure**

```c
void s_helper(
    int x, int *accum)
{
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper(x-1, accum);
    }
}
```

**Top-Level Call**

```c
int sfact(int x) {
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- **Pass pointer to update location**
Creating & Initializing Pointer

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- Variable `val` must be stored on stack
  - Because: Need to create pointer to it
- Compute pointer as `-4(%ebp)`
- Push on stack as second argument

Initial part of `sfact`

```
_sfact:
    pushl %ebp    # Save %ebp
    movl %esp,%ebp # Set %ebp
    subl $16,%esp # Add 16 bytes
    movl 8(%ebp),%edx # edx = x
    movl $1,-4(%ebp) # val = 1
```

---

Creating & Initializing Pointer

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- Variable `val` must be stored on stack
  - Because: Need to create pointer to it
- Compute pointer as `-4(%ebp)`
- Push on stack as second argument
Passing Pointer

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

Calling `s_helper` from `sfact`

```
leal -4(%ebp),%eax  # Compute &val
pushl %eax          # Push on stack
pushl %edx          # Push x
    call s_helper    # call
movl -4(%ebp),%eax  # Return val
    * * *            # Finish
```

Stack at time of call

```
<table>
<thead>
<tr>
<th>8</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Rtn adr</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>val=x!</td>
<td>Unused</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>
</tbody>
</table>
```

Call Stack Diagram:

1. `leal -4(%ebp),%eax` (Compute &val)
2. `pushl %eax` (Push on stack)
3. `pushl %edx` (Push x)
4. `call s_helper` (call)
5. `movl -4(%ebp),%eax` (Return val)
6. `* * *` (Finish)

val = x!
IA 32 Procedure 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
  - Managed by stack discipline
    - Procedures return in inverse order of calls

- IA32 Procedures Combination of Instructions + Conventions
  - Call / Ret instructions
  - Register usage conventions
    - Caller / Callee save
    - %ebp and %esp
  - Stack frame organization conventions