Machine-Level Representation

CSCI 2021: Machine Architecture and Organization

Antonia Zhai
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~zhai

With Slides from Bryant and O’Hallaron

Procedure Calls

With Slides from Bryant and O’Hallaron
IA-32 Stack

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

Stack Pointer
%esp

Stack “Bottom”

0xffffffff

Increasing Addresses

Stack Grows Down

IA-32 Stack Pushing

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

Stack “Top”

0x00000000

Stack Grows Down

Increasing Addresses

Stack “Bottom”

With Slides from Bryant and O’Hallaron
IA-32 Stack Popping

Popping
- popl Dest
- Read operand at address given by %esp
- Increment %esp by 4
- Write to Dest

Stack Operation Examples

With Slides from Bryant and O'Hallaron
Procedure Control Flow

Use stack to support procedure call and return

Procedure call:
\[
\text{call } \text{label} \quad \text{Push return address on stack; Jump to } \text{label}
\]

Return address value

- Address of instruction beyond call
- Example from disassembly

\[
\begin{align*}
804854e: & \quad \text{e8 3d 06 00 00 call 8048b90 <main>} \\
8048553: & \quad \text{50 pushl %eax}
\end{align*}
\]

Return address = 0x8048553

Procedure return:
\[
\text{ret} \quad \text{Pop address from stack; Jump to address}
\]

Procedure Call Example

\[
\begin{align*}
804854e: & \quad \text{e8 3d 06 00 00 call 8048b90 <main>} \\
8048553: & \quad \text{50 pushl %eax}
\end{align*}
\]

%esp 0x108
%esp 0x104
%eip 0x804854e
%eip 0x8048b90

%eip is program counter

With Slides from Bryant and O'Hallaron
Procedure Return Example

Stack before call

%esp 0x108
%eip 0x8048543
%esp 0x104
%eip 0x104
%esp 0x10c
%eip 0x10c
%esp 0x108
%eip 0x108
%esp 0x104
%eip 0x8048553

%eip is program counter

Call Chain Example

Code Structure

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

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

Procedure amI is recursive

Call Chain

yoo
  ↓ who
  ↓ amI
  ↓ amI
  ↓ amI

2/10/15  CSCI 2021  10

With Slides from Bryant and O'Hallaron
Stack Frames

Contents
- Local variables
- Return information
- Temporary 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

Example

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

Stack
Example

Stack

with Slides from Bryant and O’Hallaron
Example

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

Example

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

Stack

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

With Slides from Bryant and O'Hallaron
Example

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

Example

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

Stack

With Slides from Bryant and O'Hallaron

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

With Slides from Bryant and O'Hallaron

```c
yoo
who
%ebp
%esp

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

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

IA32/Linux Stack Frame

With Slides from Bryant and O’Hallaron
With Slides from Bryant and O'Hallaron

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:
    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

Resulting Stack

- %esp
- %ebp
- &course2
- &course1

- subl $8, %esp
- movl $course2, 4(%esp)
- movl $course1, (%esp)
- call swap

• • •

Subl $8, %esp
Movl $course2, 4(%esp)
Movl $course1, (%esp)
call swap

• • •

With Slides from Bryant and O'Hallaron
**swap Setup #1**

**Entering Stack**

- \%ebp
- \%esp
- \$course2
- \$course1
- Rtn adr

**Resulting Stack**

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

**swap:**

```assembly
pushl \%ebp
movl \%esp, \%ebp
pushl \%ebx
```

With Slides from Bryant and O'Hallaron

---

**swap Setup #2**

**Entering Stack**

- \%ebp
- \%esp
- \$course2
- \$course1
- Rtn adr

**Resulting Stack**

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

**swap:**

```assembly
pushl \%ebp
movl \%esp, \%ebp
pushl \%ebx
```

With Slides from Bryant and O'Hallaron
swap Setup #3

Entering Stack

\[
\begin{array}{c}
%ebp \\

\text{\&course2} \\
\text{\&course1} \\
\text{Rtn adr}
\end{array}
\]

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

Resulting Stack

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

Effect of swap Setup

Entering Stack

\[
\begin{array}{c}
%ebp \\

\text{\&course2} \\
\text{\&course1} \\
\text{Rtn adr}
\end{array}
\]

\[
\begin{array}{c}
\text{Offset (relative to } \%ebp) \\
12 \\
8 \\
4 \\
0
\end{array}
\]

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

Resulting Stack

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

\[
\begin{array}{c}
\text{\%ebp} \\
\text{\%ebx} \\
\text{\%esp}
\end{array}
\]
swap Body

Entering Stack

Resulting Stack

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

\* Observation
\* Saved & restored register %ebx

swap Finish #1

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

With Slides from Bryant and O'Hallaron

swap Finish #3

With Slides from Bryant and O'Hallaron
With Slides from Bryant and O'Hallaron

### Disassembled swap

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Machine Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048384:</td>
<td>55</td>
<td>push %ebp</td>
<td></td>
</tr>
<tr>
<td>8048385:</td>
<td>89 a5</td>
<td>mov %esp,%ebp</td>
<td></td>
</tr>
<tr>
<td>8048387:</td>
<td>53</td>
<td>push %ebx</td>
<td></td>
</tr>
<tr>
<td>8048388:</td>
<td>8b 55 08</td>
<td>mov 0x8(%ebp),%edx</td>
<td></td>
</tr>
<tr>
<td>804838b:</td>
<td>8b 4d 0c</td>
<td>mov 0xc(%ebp),%ecx</td>
<td></td>
</tr>
<tr>
<td>804838e:</td>
<td>8b 1a</td>
<td>mov (%edx),%ebx</td>
<td></td>
</tr>
<tr>
<td>8048390:</td>
<td>8b 01</td>
<td>mov (%ecx),%eax</td>
<td></td>
</tr>
<tr>
<td>8048392:</td>
<td>89 02</td>
<td>mov %eax,(%edx)</td>
<td></td>
</tr>
<tr>
<td>8048394:</td>
<td>89 19</td>
<td>mov %ebx,(%ecx)</td>
<td></td>
</tr>
<tr>
<td>8048396:</td>
<td>5b</td>
<td>pop %ebx</td>
<td></td>
</tr>
<tr>
<td>8048397:</td>
<td>5d</td>
<td>pop %ebp</td>
<td></td>
</tr>
<tr>
<td>8048398:</td>
<td>c3</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

**Calling Code**

- `movl $0x8049658,0x4(%esp)` # Copy &course2
- `movl $0x8049654,(%esp)` # Copy &course1
- `call 8048384 <swap>` # Call swap
- `leave` # Prepare to return
- `ret` # Return

### Stack Diagram

**swap's Stack**

- Offset
- yp
- xp
- Rtn adr

**Exiting Stack**

- %esp
- &zip2
- &zip1

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

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

- When procedure `yoo` calls `who`:
  - `yoo` is the caller, `who` is the callee
- Can Register be Used for Temporary Storage?

```assembly
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`
- 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 in its frame before calling
  - "Callee Save"
    - Callee saves temporary in its frame before using

Do you have to follow conventions?

2/10/15  CSCI 2021  38

With Slides from Bryant and O'Hallaron
IA32/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

2/10/15    CSCI 2021    39
With Slides from Bryant and O'Hallaron

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

pcount_r:

```
pushl \%ebp
movl \%esp, \%ebp
pushl \%ebx
subl $4, \%esp
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
lea (\%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

/* 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
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
Recursive Call #5

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

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

```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`

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

---

With Slides from Bryant and O’Hallaron
Creating Pointer as Argument

```c
int add3(int x) {
    int localx = x;
    incr($localx, 3);
    return localx;
}
```

- Use leal instruction to compute address of `localx`

Middle part of `add3`

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

Retrieving local variable

```c
int add3(int x) {
    int localx = x;
    incr($localx, 3);
    return localx;
}
```

- Retrieve `localx` from stack as return value

Final part of `add3`

```c
    movl -4(%ebp), %eax  # Return val = localx
    leave
    ret
```

What happens with C code "return &x"? Can you return pointer to a local variable?
Returning the address of a local variable

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

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 (and 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

With Slides from Bryant and O’Hallaron