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
### 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>pushl %eax</th>
<th>popl %edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x104 213</td>
<td>0x104 213</td>
</tr>
<tr>
<td>0x108 123</td>
<td>0x108 123</td>
</tr>
<tr>
<td>0x10c 0</td>
<td>0x10c 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>213</td>
<td>555</td>
<td>0x108</td>
</tr>
<tr>
<td>213</td>
<td>555</td>
<td>0x104</td>
</tr>
<tr>
<td>213</td>
<td>555</td>
<td>0x108</td>
</tr>
</tbody>
</table>
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

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

0x08048553 +0x0000063d = 0x08048b90

%esp 0x108
%eip 0x804854e

%esp 0x104
%eip 0x8048b90

%eip is program counter
Procedure Return Example

8048591: c3 ret

%esp 0x104
%eip 0x8048591

%esp 0x108
%eip 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**

- `yoo` → `who` → `amI` → `amI`
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();
  •
  •
}
```
Stack Frames (4)

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

Call Chain

- **Frame Pointer** `%ebp`
- **Stack Pointer** `%esp`

- yoo
- who
- ...
- who
- yoo
Stack Frames (5)

Call Chain

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

Frame Pointer %ebp
Stack Pointer %esp

Stack Frames (5)
Stack Frames (6)

Call Chain

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

Frame Pointer
%ebp

Stack Pointer
%esp

yoo
who
amI
amI
Stack Frames (7)

Call Chain

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

Frame Pointer `%ebp`

Stack Pointer `%esp`
Stack Frames (8)

Call Chain

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

Stack Pointer: `%esp`

Frame Pointer: `%ebp`
Stack Frames (9)

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

Call Chain

Frame Pointer  
%ebp

Stack Pointer  
%esp

yoo  
who  
amI

yoo

who

amI
Stack Frames (10)

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

Call Chain

```
yoo
  ↓
who
  ↓
amI
  ↓
amI
  ↓
amI
```

Frame Pointer
%ebp

Stack Pointer
%esp

Who

Yoo
Stack Frames (11)

Call Chain

```
ami(...) {
    · · ·
}
```

Stack Pointer
%esp

Frame Pointer
%ebp

who

ami

ami

ami

ami

ami

yoo

who

ami

ami

ami

ami
Stack Frames (12)

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

Call Chain

```
who
  ↓
amI
  ↓
amI
  ↓
amI

Frame Pointer
%ebp

Stack Pointer
%esp
```

who
```c
who(…)
{  
    •
    amI();
    •
    amI();
}
```
Stack Frames (13)

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

Call Chain

Frame Pointer %ebp

Stack Pointer %esp

yoo

who

ami

ami
**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

---

**Diagram:**
- **Frame Pointer** (%ebp)
- **Return Addr**
- **Arguments**
- **Saved Registers + Local Variables**
- **Argument Build**
- **Stack Pointer** (%esp)
- **Caller Frame**

---

SSE2030: Introduction to Computer Systems | Fall 2011 | Jin-Soo Kim (jinsookim@skku.edu)
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

int zip1 = 15213;
int zip2 = 91125;

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

Calling swap from call_swap

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

Resulting Stack

%esp

&zip2
&zip1
Rtn adr
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

Setup

Body

Finish
Swap Setup (1)

Entering Stack

Resulting Stack

\[\text{swap:} \]
\[
\begin{align*}
\text{pushl} & \ %ebp \\
\text{movl} & \ %esp,%ebp \\
\text{pushl} & \ %ebx
\end{align*}
\]
Swap Setup (2)

Entering Stack

%ebp
&zip2
&zip1
Rtn adr

Resulting Stack

%esp
%ebp

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

Entering Stack

Resulting Stack

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

**Entering Stack**

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

**Resulting Stack**

```
old %ebp
old %ebx
...```
swap Finish (1)

swap’s Stack

Observation
- Saved & restored register %ebx

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

Offset

12
8
4
0
-4

yp
xp
Rtn adr
Old %ebp
Old %ebx

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

```assembly
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
.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
```

---

The diagram illustrates the stack setup process for the rfact function. The stack frame contains the following:

- **Caller**:
  - pre %ebp
  - pre %ebx
  - x
  - Rtn adr

- **Callee**:
  - Old %ebp
  - Old %ebx

The stack frame for the caller and callee is maintained with the appropriate registers and addresses, ensuring proper function execution and data management.
rfact Body

- Registers
  - `%ebx`: stored value of `x`
  - `%eax`
    - Temporary value of `x-1`
    - Returned value from `rfact(x-1)`
    - Returned value from this call

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

Recursion

```
movl 8(%ebp),%ebx          # ebx = x
cmpl $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:                      # Term:
    movl $1,%eax          # return val = 1
.L79:                      # Done:
```
rfact Recursion

leal \(-1(\%ebx),\%eax\)

pushl \%eax

call rfact

%eax \(x-1\)
%ebx \(x\)

%eax \(x\)
%ebx \(x\)

%eax \(x\)
%ebx \(x\)

%eax \(x\)
%ebx \(x\)

%eax \(x\)
%ebx \(x\)

%eax \(x\)
%ebx \(x\)

%eax \(x\)
%ebx \(x\)

%eax \(x\)
%ebx \(x\)
Assume that \( rfact(x-1) \) returns \((x-1)\)! in register %eax
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