The Stack and Memory in IA32

10/6/16
Tuesday, we covered these IA32 convenience instructions...

• `pushl src`
  `subl $4, %esp`
  `movl src, (%esp)`

• `popl dst`
  `movl (%esp), dst`
  `addl $4, %esp`

• `leave`
  `%esp = %ebp`
  `popl %ebp`
Next up: call and ret

• Call jumps to the start of the callee’s instructions.
  • indicated by a label

• Ret jumps back to the next instruction of the caller.

Why don’t we just do this with jmp?
Function calls

Text Memory Region

```
funcA:
  addl $5, %ecx
  movl %ecx, -4(%ebp)
  ...
call funcB
  addl %eax, %ecx
  ...

funcB:
  pushl %ebp
  movl %esp, %ebp
  ...
  movl $10, %eax
  leave
  ret
```
Function calls

What we’d like this to do:

Set up function B’s stack.

Text Memory Region

```assembly
funcA:
  addl $5, %ecx
  movl %ecx, -4(%ebp)
  ...
  call funcB
  addl %eax, %ecx
  ...

funcB:
  pushl %ebp
  movl %esp, %ebp
  ...
  movl $10, %eax
  leave
  ret
```
Function calls

What we’d like this to do:

Set up function B’s stack.

Execute the body of B, produce result (stored in %eax).

Text Memory Region

```
funcA:
  addl $5, %ecx
  movl %ecx, -4(%ebp)
  ...  
  call funcB
  addl %eax, %ecx
  ...  

funcB:
  pushl %ebp
  movl %esp, %ebp
  ...  
  movl $10, %eax
  leave
  ret
```
Function calls

What we’d like this to do:

Set up function B’s stack.

Execute the body of B, produce result (stored in %eax).

Restore function A’s stack.

```
Text Memory Region

funcA:
  addl $5, %ecx
  movl %ecx, -4(%ebp)
  ...
  call funcB
  addl %eax, %ecx
  ...

funcB:
  pushl %ebp
  movl %esp, %ebp
  ...
  movl $10, %eax
  leave
  ret
```
Function calls

What we’d like this to do:

Return:
Go back to what we were doing before funcB started.

Unlike jumping, we intend to go back!
We need to get $\%eip$ back.

- `call` should save $\%eip$ then jump to callee.

- `ret` should restore $\%eip$ to jump back to the caller.

We could accomplish this without `call` and `ret`. They’re just convenience instructions (like `push`, `pop`, and `leave`).
Write write call and ret using other IA32 instructions.

- **call f:** save %eip then jump to the start of f.
  
  push %eip
  
  jmp f

- **ret:** restore %eip to jump back to the caller.
  
  popl %eip
### IA32 Stack / Function Call Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Example Code</th>
</tr>
</thead>
</table>
| pushl       | Create space on the stack and place the source there.                                                                                            | subl $4, %esp
              movl src, (%esp) |
| popl        | Remove the top item off the stack and store it at the destination.                                                                                | movl (%esp), dst
              addl $4, %esp |
| call        | 1. Push return address on stack  
              2. Jump to start of function                                                                                                                      | push %eip
              jmp target |
| leave       | Prepare the stack for return (restoring caller’s stack frame)                                                                                         | movl %ebp, %esp
              popl %ebp |
| ret         | Return to the caller, PC ← saved PC (pop return address off the stack into PC (eip))                                                                | popl %eip |
On the stack between the caller’s and the callee’s stack frames...

- Caller’s base pointer (to reset the stack).
- Caller’s instruction pointer (to continue execution).
- Function parameters.
What order should we store all of these things on the stack? Why?

A
- callee parameters
- return address
- caller’s base pointer

B
- return address
- caller’s base pointer
- callee parameters

C
- caller’s base pointer
- callee parameters
- callee parameters
- return address

D
- callee parameters
- caller’s base pointer
- return address

E: some other order.
Putting it all together...

- **Callee’s frame.**
  - Caller’s local variables.
  - Caller’s Frame Pointer
  - Return Address
  - First Argument to Callee
  - ... (omitted)
  - Final Argument to Callee
  - Caller’s local variables.

- **Caller’s frame.**
  - Older stack frames.
  - ... (omitted)

Shared by caller and callee.
Translate this to IA32. What should be on the stack?

```c
int add_them(int a, int b, int c) {
    return a+b+c;
}

int main() {
    add_them(1, 2, 3);
}
```

Assume the stack initially looks like:

```
%esp  main
%ebp  0xFFFFFFFF
```
Stack Frame Contents

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

- Saved registers
- Spilled temporaries
Saving Registers

• Registers are a scarce resource, but they’re fast to access. Memory is plentiful, but slower to access.

• Should the caller save its registers to free them up for the callee to use?
• Should the callee save the registers in case the caller was using them?
• Who needs more registers for temporary calculations, the caller or callee?

• Clearly the answers depend on what the functions do...
Splitting the difference...

• We can’t know the answers to those questions in advance...

• We have six general-purpose registers, let’s divide them into two groups:
  • Caller-saved: %eax, %ecx, %edx
  • Callee-saved: %ebx, %esi, %edi
Register Convention

• Caller-saved: %eax, %ecx, %edx
  • If the caller wants to preserve these registers, it must save them prior to calling callee.
  • The callee is free to trash these; the caller will restore if needed.

• Callee-saved: %ebx, %esi, %edi
  • If the callee wants to use these registers, it must save them first, and restore them before returning.
  • The caller can assume these will be preserved.

This is why lab 4 had the comment about using only %eax, %ecx, and %edx.
Running Out of Registers

• Some computations require more than six registers to store temporary values.

• *Register spilling:* The compiler will move some temporary values to memory, if necessary.
  • Values pushed onto stack, popped off later
  • No explicit variable declared by user
IA32 addressing modes

• Direct addressing (what we’ve seen so far)
  \[-4 (\%ebp)\]

• Indexed addressing
  \((\%ecx, \%edx, 4)\)
Indexed Addressing Mode

• General form:
  
  \( \text{offset}(\%\text{base}, \%\text{index}, \text{scale}) \)

• Translation: Access the memory at address...
  
  \( \text{base} + (\text{index} \times \text{scale}) + \text{offset} \)

Discussion: when would this mode be useful?
Example

Suppose \( i \) is at \( \%ebp-8 \), and equals 2.

User says:

\[
\text{float} \_\text{array}[i] = 9;
\]

Translates to:

\[
\text{movl} \ -8(\%ebp), \ %edx
\]

<table>
<thead>
<tr>
<th>Registers</th>
<th>%ecx</th>
<th>0x0824</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0824: iptr[0]</td>
</tr>
<tr>
<td>0x0828: iptr[1]</td>
</tr>
<tr>
<td>0x082C: iptr[2]</td>
</tr>
<tr>
<td>0x0830: iptr[3]</td>
</tr>
</tbody>
</table>
Example

Suppose $i$ is at $\%ebp - 8$, and equals 2.

User says:

float_arr[i] = 9;

Translates to:

movl -8(\%ebp), \%edx
Example

Suppose $i$ is at $\%ebp - 8$, and equals 2.

User says:

float_arr[i] = 9;

Translates to:

```
movl -8(%ebp), %edx
movl $9, (%ecx, %edx, 4)
```

Heap

<table>
<thead>
<tr>
<th></th>
<th>0x0824</th>
<th>0x0828</th>
<th>0x082C</th>
</tr>
</thead>
<tbody>
<tr>
<td>iptr[0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iptr[1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iptr[2]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iptr[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Registers:

<table>
<thead>
<tr>
<th>%ecx</th>
<th>0x0824</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>2</td>
</tr>
</tbody>
</table>
Example

Suppose \( i \) is at \( \%\text{ebp} - 8 \), and equals 2.

User says:

\[
\text{float}_\text{arr}[i] = 9;
\]

Translates to:

\[
\begin{align*}
\text{movl} & \ -8(\%\text{ebp}), \ %\text{edx} \\
\text{movl} & \ 9, \ (%\text{ecx}, \ %\text{edx}, \ 4)
\end{align*}
\]

\[
0x0824 + (2 * 4) + 0 = 0x082C
\]
What is the final state after this code?

```
addl $4, %eax
movl (%eax), %eax
sall $1, %eax
movl %edx, (%ecx, %eax, 2)
```

(Initial state)

<table>
<thead>
<tr>
<th>Registers:</th>
<th>Memory:</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x2464</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x246C</td>
</tr>
<tr>
<td>%edx</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2464:</td>
</tr>
<tr>
<td>0x2468:</td>
</tr>
<tr>
<td>0x246C:</td>
</tr>
<tr>
<td>0x2470:</td>
</tr>
<tr>
<td>0x2474:</td>
</tr>
</tbody>
</table>
Translate this array access to IA32

```
int *x;
x = malloc(10*sizeof(int));

...

x[i] = -12;
```

At this point, suppose that the variable `x` is stored at `%ebp+8`. And `i` is in `%edx`. Use indexed addressing to assign into the array.
The *leal* instruction

- Uses the circuitry that computes addresses.
- Doesn’t actually access memory.
- Compute an “address” and store it in a register.
- Can use the full version of indexed addressing.

```
leal offset(%base, %index, scale), dest
```

```
leal 5(%eax, %esi, 2), %edx
#put %eax + 5 + (2*%esi) in %edx
```