CS 31: Intro to Systems  
Functions and the Stack  

Martin Gagne  
Swarthmore College  
February 23, 2016
Late policy:
• you do not have to send me an email to inform me of a late submission before the deadline
• you have to send me an email when you submit your late work

Please do not procrastinate on labs, they won’t get any easier

Use Piazza!
IA32 Calling Convention (gcc)

- In register %eax:
  - The return value

- In the callee’s stack frame:
  - The caller’s %ebp value (previous frame pointer)

- In the caller’s frame (shared with callee):
  - Function arguments
  - Return address (saved PC value)
Instructions in Memory

```
funcA:
    ...
    call funcB
    ...

funcB:
    pushl %ebp
    movl %esp, %ebp
    ...
```

Diagram:
- **Operating system**
- **Text**
- **Data**
- **Heap**
- **Stack**

Function call between **Function A** and **Function B**.
Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)

What do we do now?

Follow PC, fetch instruction:

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

```
funcB:
pushl %ebp
movl %esp, %ebp
...  
movl $10, %eax
leave
ret
```
Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)

What do we do now?

Follow PC, fetch instruction:

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

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

Update PC to next instruction.

Execute the `addl`.
Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)

What do we do now?

Follow PC, fetch instruction:

```
movl $ecx, -4(%ebp)
```

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
```
Program Counter

Recall: PC stores the address of the next instruction. (A pointer to the next instruction.)

What do we do now?

Follow PC, fetch instruction:

\[
\text{movl } $\text{ecx}, -4(\%\text{ebp})
\]

Update PC to next instruction.

Execute the \text{movl}.
Program Counter

Recall: PC stores the address of the next instruction. (A pointer to the next instruction.)

What do we do now?

Keep executing in a straight line downwards like this until:

We hit a jump instruction.
We call a function.

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
```
Changing the PC: Functions

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

FuncB:
pushl %ebp
movl %esp, %ebp
...
movl $10, %eax
leave
ret

What we’d like this to do:
Changing the PC: Functions

What we’d like this to do:

Set up function B’s stack.
Changing the PC: Functions

What we’d like this to do:

Set up function B’s stack.

Execute the body of B, produce result (stored in %eax).
Changing the PC: Functions

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
```
Changing the PC: Functions

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!
Like `push`, `pop`, and `leave`, `call` and `ret` are convenience instructions. What should they do to support the PC-changing behavior we need? (The PC is `%eip`.)

```
   call
```

In words:  

```
   ret
```

In words:  

```
In instructions:
```

In instructions:  

Functions and the Stack

Executing instruction:
call funcB

PC points to next instruction

Stack Memory Region

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
Functions and the Stack

1. pushl %eip

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

funcB:
  pushl %ebp
  movl %esp, %ebp
  ...
  movl $10, %eax
  leave
  ret
```
Functions and the Stack

1. pushl %eip
2. jump funcB
3. create stack frame
4. (execute funcB)

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
```
Functions and the Stack

1. pushl %eip
2. jump funcB
3. create stack frame
4. (execute funcB)
5. restore stack
6. popl %eip

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
```
Functions and the Stack

7. (resume funcA)

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

funcB:
  pushl %ebp
  movl %esp, %ebp
  ...
  movl $10, %eax
  leave
  ret
```
Functions and the Stack

1. pushl %eip
2. jump funcB
3. create stack frame
4. (execute funcB)
5. restore stack
6. popl %eip
7. (resume funcA)

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
Functions and the Stack

1. pushl %eip          call
2. jump funcB
3. create stack frame
4. (execute funcB)
5. restore stack      leave
6. popl %eip          ret
7. (resume funcA)

Return address:
Address of the instruction we should jump back to when we finish (return from) the currently executing function.
### IA32 Stack / Function Call Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Assembly 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 |
IA32 Calling Convention (gcc)

• In register %eax:
  – The return value

• In the callee’s stack frame:
  – The caller’s %ebp value (previous frame pointer)

• In the caller’s frame (shared with callee):
  – Function arguments
  – Return address (saved PC value)
On the stack between the caller’s and the callee’s local variables…

- 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
- return address

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

E: some other order.
Putting it all together…

Callee’s frame.
- Callee’s local variables.
- Caller’s Frame Pointer
  - Return Address
  - First Argument to Callee
    - …
  - Final Argument to Callee
- Caller’s local variables.

Caller’s frame.
- …

Shared by caller and callee.

Older stack frames.
- …
Arguments

• Arguments to the callee are stored just underneath the return address.

• Does it matter what order we store the arguments in?

• Not really, as long as we’re consistent (follow conventions).

This is why arguments can be found at positive offsets relative to %ebp.
How would we translate this to IA32? What should be on the stack?

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

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

Assume the stack initially looks like:

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

- What needs to be stored in a stack frame?
  - Alternatively: What must a function know?

- 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
  – callee free to trash these, 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
  – caller can assume these will be preserved

This is why I’ve told you to only use these three registers.
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
  \(-4 (\%ecx, \%edx, 4)\)
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:
\[
\text{float}_{\text{arr}}[i] = 9;
\]

Translates to:
\[
\text{movl } -8(\%ebp), \%edx
\]
Example

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

User says:
float_arr[i] = 9;

Translates to:
\text{movl} -8(\%ebp), \%edx
Example

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

User says:

```c
float_arr[i] = 9;
```

Translates to:

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

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Registers:

- `%ecx` 0x0824
- `%edx` 2

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

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-------</td>
</tr>
</tbody>
</table>
Example

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

User says:

\[
\text{float\_arr}[i] = 9;
\]

Translates to:

\[
\text{movl } -8(\%ebp), \%edx \\
\text{movl } $9, (\%ecx, \%edx, 4)
\]

\[
0x0824 + (2 \times 4) + 0 \\
0x0824 + 8 = 0x082C
\]
What is the final state after this code?

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

(Initial state)

Registers:
- %eax 0x2464
- %ecx 0x246C
- %edx 7

Memory:

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2464: 5</td>
</tr>
<tr>
<td>0x2468: 1</td>
</tr>
<tr>
<td>0x246C: 42</td>
</tr>
<tr>
<td>0x2470: 3</td>
</tr>
<tr>
<td>0x2474: 9</td>
</tr>
</tbody>
</table>
Translate this array access to IA32

```c
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
```
Midterm

• The midterm will cover material up to here.

• The exam will be in class on Thursday, March 2