CS 31: Intro to Systems
Functions and the Stack

Kevin Webb
Swarthmore College
February 23, 2016
Reading Quiz
Overview

• Stack data structure, applied to memory

• Behavior of function calls

• Storage of function data, at IA32 level
“A” Stack

• A stack is a basic data structure
  – Last in, first out behavior (LIFO)
  – Two operations
    • Push (add item to top of stack)
    • Pop (remove item from top of stack)
“The” Stack

• Apply stack data structure to memory
  – Store local (automatic) variables
  – Maintain state for functions (e.g., where to return)

• Organized into units called frames
  – One frame represents all of the information for one function.
  – Sometimes called activation records
Memory Model

- Starts at the highest memory addresses, grows into lower addresses.
Stack Frames

• As functions get called, new frames added to stack.

• Example: Lab 4
  – main calls get_values()
  – get_values calls read_float()
  – read_float calls I/O library
Stack Frames

• As functions return, frames removed from stack.

• Example: Lab 4
  – I/O library returns to read_float
  – read_float returns to get_values
  – get_values returns to main

All of this stack growing/shrinking happens automatically (from the programmer’s perspective).
What is responsible for creating and removing stack frames?

A. The user

B. The compiler

C. C library code

D. The operating system

E. Something / someone else

Insight: EVERY function needs a stack frame. Creating / destroying a stack frame is a (mostly) generic procedure.
Stack Frame Contents

• What needs to be stored in a stack frame?
  – Alternatively: What *must* a function know / access?

  – Hint: At least 5 things

```
read_float
get_values
main
```

0xFFFFFFFF
Stack Frame Contents

• What needs to be stored in a stack frame?
  – Alternatively: What *must* a function know / access?

• Local variables

```
read_float
get_values
main
0xFFFFFFFF
```
Local Variables

If the programmer says:

```c
int x = 0;
```

Where should `x` be stored?

(Recall basic stack data structure)

Which memory address is that?
How should we determine the address to use for storing a new local variable?

A. The programmer specifies the variable location.

B. The CPU stores the location of the current stack frame.

C. The operating system keeps track of the top of the stack.

D. The compiler knows / determines where the local data for each function will be as it generates code.

E. The address is determined some other way.
Program Characteristics

• Compile time (static)
  – Information that is known by analyzing your program
  – Independent of the machine and inputs

• Run time (dynamic)
  – Information that isn’t known until program is running
  – Depends on machine characteristics and user input
The Compiler Can...

• Perform type checking.

• Determine how much space you need on the stack to store local variables.

• Insert IA32 instructions for you to set up the stack for function calls.
  – Create stack frames on function call
  – Restore stack to previous state on function return
Local Variables

• Compiler can allocate $N$ bytes on the stack by subtracting $N$ from the “stack pointer”: $\%esp$
The Compiler Can’t…

• Predict user input.

```c
int main() {
    int decision = [read user input];
    if (x > 5) {
        funcA(x);
    } else {
        funcB();
    }
}
```

![Diagram of main function with input 'x']
The Compiler Can’t...

• Predict user input.

```c
int main() {
    int decision = [read user input];
    if (x > 5) {
        funcA(x);
    } else {
        funcB();
    }
}
```
The Compiler Can’t…

• Predict user input.

• Can’t assume a function will always be at a certain address on the stack.

Alternative: create stack frames relative to the current (dynamic) state of the stack.
Stack Frame Location

• Where in memory is the current stack frame?

[Diagram showing stack frames with labels for current top and bottom of stack, and function labels (main, function 1, function 2), along with a memory address (0xFFFFFFFF)]
Recall: IA32 Registers

- Information about currently executing program

- General purpose registers
  - %eax
  - %ecx
  - %edx
  - %ebx
  - %esi
  - %edi
  - %esp
  - %ebp

- Current stack top
- Current stack frame

- Instruction pointer (PC): %eip

- Condition codes: CF, ZF, SF, OF
Stack Frame Location

- Where in memory is the current stack frame?

- Maintain invariant:
  - The current function’s stack frame is always between the addresses stored in %esp and %ebp

- %esp: stack pointer
- %ebp: frame pointer (base pointer)
Stack Frame Location

• Compiler ensures that this invariant holds.
  – We’ll see how a bit later.

• This is why all local variables we’ve seen in IA32 are relative to %ebp or %esp!
How would we implement pushing x to the top of the stack in IA32?

A. Increment `%esp`
   Store x at (%esp)

B. Store x at (%esp)
   Increment `%esp`

C. Decrement `%esp`
   Store x at (%esp)

D. Store x at (%esp)
   Decrement `%esp`

E. Copy `%esp` to `%ebp`
   Store x at (%ebp)
Push & Pop

- IA32 provides convenient instructions:
  - `pushl src`
    - Move stack pointer up by 4 bytes
    - Copy ‘src’ to current top of stack
  - `popl dst`
    - Copy current top of stack to ‘dst’
    - Move stack pointer down 4 bytes

- `src` and `dst` are the contents of any register
Local Variables

• More generally, we can make space on the stack for N bytes by subtracting N from %esp
Local Variables

• More generally, we can make space on the stack for N bytes by subtracting N from $\%esp$
• When we’re done, free the space by adding N back to $\%esp$
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
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
Stack Frame Relationships

• If function 1 calls function 2:
  – function 1 is the caller
  – function 2 is the callee

• With respect to main:
  – main is the caller
  – function 1 is the callee
Where should we store all this stuff?

A. In registers
B. On the heap
C. In the caller’s stack frame
D. In the callee’s stack frame
E. Somewhere else

Previous stack frame base address
Function arguments
Return value
Return address
Calling Convention

• You could store this stuff wherever you want!
  – The hardware does NOT care.
  – What matters: everyone agrees on where to find the necessary data.

• Calling convention: agreed upon system for exchanging data between caller and callee
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)
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)
Return Value

• If the callee function produces a result, the caller can find it in %eax

• We saw this when we wrote our while loop:
  – Copy the result to %eax before we finished up
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)
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %esp and %ebp

• Must adjust %esp, %ebp on call / return.
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %esp and %ebp

• Immediately upon calling a function:
  1. pushl %ebp
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %esp and %ebp

• Immediately upon calling a function:
  1. pushl %ebp
  2. Set %ebp = %esp
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %esp and %ebp

• Immediately upon calling a function:
  1. pushl %ebp
  2. Set %ebp = %esp
  3. Subtract N from %esp

Callee can now execute.
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %esp and %ebp

• To return, reverse this:
Frame Pointer

- Must maintain invariant:
  - The current function’s stack frame is always between the addresses stored in %esp and %ebp

- To return, reverse this:
  1. set %esp = %ebp
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %esp and %ebp

• To return, reverse this:
  1. set %esp = %ebp
  2. popl %ebp
Frame Pointer

• Must maintain invariant:
  – The current function’s stack frame is always between the addresses stored in %esp and %ebp

• To return, reverse this:
  1. set %esp = %ebp
  2. popl %ebp

Back to where we started.

IA32 has another convenience instruction for this: leave
Recall: Assembly While Loop

sum_function:
  pushl %ebp
  movl %esp, %ebp

# Your code here

movl $10, %eax
leave
ret

Set up the stack frame for this function.

Store return value in %eax.

Restore caller’s %esp, %ebp.
Recap

• The stack memory region keeps state for the sequence of function calls we’ve made

• The state for one function is a *stack frame*

• If function A calls function B:
  – function A is the *caller*
  – function B is the *callee*
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
Dynamic Stack Accounting

- Dedicate CPU registers for stack bookkeeping
  - `%esp` (stack pointer): Top of current stack frame
  - `%ebp` (frame pointer): Base of current stack frame

- Compiler maintains these pointers by inserting instructions on function call/return.
  - It doesn’t know (or care about) the exact addresses they point to.

![Diagram showing `%esp` and `%ebp` pointers to the current stack frame]
Calling Convention

• Agreed upon method for exchanging data between caller and callee.
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)
Frame Pointer: Function Call

Initial state

- %esp
- %ebp

Push %ebp (store caller’s frame pointer)

- %esp
- %ebp

movl %esp, %ebp (establish callee’s frame pointer)

- %esp
- %ebp

subl $SIZE, %esp (allocate space for callee’s locals)

- %esp
- %ebp

Callee

Caller’s %ebp value

...
Frame Pointer: Function Return

IA32 provides a convenience instruction that does all of this: `leave`
Lab 4: swap.s

swap:
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    # Your code here
    leave
    ret
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

0x0

- Operating system
- Text
- Data
- Heap

Stack

0xF0000000

Function A

funcA:
... 
call funcB 
...

Function B

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

...
Program Counter

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

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

What do we do now?

Follow PC, fetch instruction:

addl $5, %ecx
What do we do now?

Follow PC, fetch instruction:

\[
\text{addl } $5, \%ecx
\]

Update PC to next instruction.

Execute the \text{addl}. 

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

Text Memory Region

\[
\begin{align*}
\text{funcA:} \\
&\text{addl } $5, \%ecx \\
&\text{movl } \%ecx, -4(\%ebp) \\
&\text{...} \\
&\text{call funcB} \\
&\text{addl } \%eax, \%ecx \\
&\text{...} \\
\text{funcB:} \\
&\text{pushl } \%ebp \\
&\text{movl } \%esp, \%ebp \\
&\text{...} \\
&\text{movl } $10, \%eax \\
&\text{leave} \\
&\text{ret}
\end{align*}
\]
Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)

What do we do now?
Follow PC, fetch instruction:

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

**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
```
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)

Update PC to next instruction.
Execute the movl.

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?

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: Jump

• On a jump:
  – Check condition codes
  – Set PC to execute elsewhere (not next instruction)

• Do we ever need to go back to the instruction after the jump?

Maybe (and if so, we’d have a label to jump back to), but usually not.
Changing the PC: Functions

What we’d like this to do:

```
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:
Set up function B’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:

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
```
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.
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`.)

In words: `call`  
In instructions: `call`

In words: `ret`  
In instructions: `ret`
Functions and the Stack

Executing instruction:
\texttt{call funcB}

PC points to \texttt{next instruction}

\texttt{Text Memory Region}

\begin{verbatim}
funcA:
  addl $5, %ecx
  movl %ecx, -4(%ebp)
  ...
  call funcB
  addl %eax, %ecx
  ...

funcB:
  pushl %ebp
  movl %esp, %ebp
  ...
  movl $10, %eax
  leave
  ret
\end{verbatim}
Functions and the Stack

1. pushl %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

Function A
...
Functions and the Stack

Program Counter (%eip)

Stack Memory Region

Function B

Stored PC in funcA

Function A

...
Functions and the Stack

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

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
```
Functions and the Stack

6. (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. (execute `funcB`)
4. restore stack
5. `popl %eip`
6. (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  \(\text{call}\)
2. jump funcB  \(\text{call}\)
3. (execute funcB)  \(\text{call}\)
4. restore stack  \(\text{leave}\)
5. popl %eip  \(\text{ret}\)
6. (resume funcA)  \(\text{ret}\)

Stack Memory Region

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)
We know we’re going to place arguments on the stack, in the caller’s frame. Should they go above or below the return address?

A. Above

B. Below

C. Somewhere else
# 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 |
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.
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.
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:

```
main
%esp ➔
%ebp ➔ 0xFFFFFFFF
```
How would we translate this to IA32? What should be on the stack?

main: func: Stack
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
Midterm

• The midterm will cover material up to here.

• The exam will be in class on Thursday, March 3