CS 31: Intro to Systems C Programming
L11-12: Functions & The Stack

Vasanta Chaganti & Kevin Webb
Swarthmore College
October 10 - 24, 2023
Today

• Functions & the stack
• Stack data structure, applied to memory

• Behavior of function calls

• Storage of function data, at assembly level
Overview

- Stack data structure, applied to memory

- Behavior of function calls

- Storage of function data, at assembly 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)

Push (add data item)  
Newest data

Pop (remove and return item)

Oldest data
“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.
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?

• Local variables

![Diagram of stack frame contents]

- 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?

```
function 2

function 1

main

0xFFFFFFFF
```

```
0x????????
```

```
X goes here
```
How should we determine the addresses (or location in memory) to use for storing the stack frames?

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.
How should we determine the addresses (or location in memory) to use for store the stack frames?

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 assembly 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
The Compiler Can...allocate space for local variables

Compiler can allocate N bytes on the stack by subtracting N from the stack pointer: (rsp)
The Compiler Can’t…Predict User Input

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

int main(void) {
    int decision = [read user input];
    if (decision > 5) {
        funcA();
    } 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.
Recall: x86_64 Register Conventions

- Working memory for currently executing program
  - Address of next instruction to execute ( `%rip` )
  
  - Location of runtime stack (%rbp, %rsp)
  
  - Temporary data ( `%rax - %r15` )

  - Status of recent ALU tests ( CF, ZF, SF, OF )

<table>
<thead>
<tr>
<th>%rax</th>
<th>%r8</th>
<th>%r14</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%r9</td>
<td>%r15</td>
</tr>
<tr>
<td>%rcx</td>
<td>%r10</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>%r11</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>%r12</td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td>%r13</td>
<td></td>
</tr>
</tbody>
</table>

- General purpose registers

- Current stack top
- Current stack frame
- Program Counter (PC)
- Condition codes (flags)
Stack Frame Location

Where in memory is the current stack frame?

current top of the stack

function 2

function 1

main

0xFFFFFFFF
Stack Frame Location

Where in memory is the current stack frame?

- **rsp**: stack pointer
- **rbp**: frame pointer (base pointer)

**Invariant:**
The current function’s stack frame is always between the addresses stored in `rsp` and `rbp`.
Stack Frame Location

- Compiler ensures that this invariant holds.
- This is why all local variables are relative to \texttt{rbp} or \texttt{rsp}!

\begin{itemize}
  \item \texttt{rbp}
  \item \texttt{rsp}
\end{itemize}

\textbf{Invariant:}
The current function’s stack frame is always between the addresses stored in \texttt{rsp} and \texttt{rbp}.
How would we implement pushing a new variable (x) to the top of the stack in x86_64?

A. Increment rsp
   Store x at (rsp)

B. Store x at (rsp)
   Increment rsp

C. Decrement rsp
   Store x at (rsp)

D. Store x at (rsp)
   Decrement rsp

E. Copy rsp to rbp
   Store x at rbp
How would we implement pushing x to the top of the stack in x86_64?

A. Increment rsp
   Store x at (rsp)

B. Store x at (rsp)
   Increment rsp

C. Decrement rsp
   Store x at (rsp)

D. Store x at (rsp)
   Decrement rsp

E. Copy rsp to rbp
   Store x at rbp
Local Variables

• Generally, we can make space on the stack for N bytes by:
  – subtracting N from rsp
Local Variables

- When we’re done, free the space by adding N back to $\text{rsp}$
  \[ \text{rsp} + N \]
Stack Frame Contents

What needs to be stored in a stack frame? 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? **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 the following:

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

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

• When possible, keep values in registers (why?)
  – Accessing registers is faster than memory (stack)
The function’s **return value**: In register %rax

The caller’s %rbp value (caller’s **saved frame pointer**)
– Placed on the stack in the callee’s stack frame

The **return address** (saved PC value to resume execution on return)
– Placed on the stack in the caller’s stack frame

**Arguments** passed to a function:
– First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
– Any additional arguments stored on the caller’s stack frame (shared with callee)
Top of the Stack

- Space for local & temporary vars, & saved register values
- saved %rbp
  (Caller’s stack frame)

Return address
(saved program counter)

- parameter 7
  (first six parameters passed as registers: rdi, rsi, rdx, rcx, r8, r9)
  :
  :
- parameter n
  :
  :

Both caller & callee can access these:
- push %rip (PC)
- push input arguments to callee

Bottom of Stack

- Call
  - Lower memory address
- Return
  - Higher memory address

Callee’s Frame or Active Frame (current frame in execution)

Caller’s Frame

Earlier Stack Frames
x86_64 Calling Convention

• The function’s **return value**: In register %rax

• The caller’s %rbp value (caller’s saved frame pointer)
  – Placed on the stack in the callee’s stack frame

• The **return address** (saved PC value to resume execution on return)
  – Placed on the stack in the caller’s stack frame

• Arguments passed to a function:
  – First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  – Any additional arguments stored on the caller’s stack frame (shared with callee)
Return Value

• If the callee function produces a result, the caller can find it in $rax$

• You’ll see this at the end of every function call in the maze lab, if there is a return value.
  – Copy the result to $rax$ before we finishing up
Dynamic Stack Accounting

• Dedicate CPU registers for stack bookkeeping
  – %rsp (stack pointer): Top of current stack frame
  – %rbp (frame pointer): Base of current stack frame

• Compiler maintains these pointers
  – Does the compiler know the exact address they point to?
  – Compiler doesn’t know or care! (job of the OS to figure that out)

• To the compiler: every variable access is relative to %rsp and %rbp!
Compiler: updates to rsp/rbp on function call/return

invariant: The current function’s stack frame is always between the addresses stored in rsp and rbp
Compiler: Upon a new Function Call.

Immediately upon calling a new function:

1. push current %rbp
Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp

invariant:
The current function’s stack frame is always between the addresses stored in rsp and rbp
Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp
Compiler: Upon a new Function Call.

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp
3. Subtract N from %rsp

invariant:
The current function’s stack frame is always between the addresses stored in rsp and rbp
Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp
3. Subtract N from %rsp

invariant: The current function’s stack frame is always between the addresses stored in rsp and rbp

Callee can now execute.
Compiler: Returning from a function call..

Returning from a function:
1. Set %rsp = %rbp

invariant:
The current function’s stack frame is always between the addresses stored in rsp and rbp
Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp (callee stack frame no longer exists)

---

invariant:
The current function’s stack frame is always between the addresses stored in_rsp and_rbp
Compiler: Returning from a function call..

Returning from a function:
1. Set %rsp = %rbp (callee stack frame no longer exists)
2. pop %rbp

**Invariant:**
The current function’s stack frame is always between the addresses stored in rsp and rbp
Compiler: Returning from a function call..

Returning from a function:
1. Set %rsp = %rbp
2. pop %rbp
   - pop caller’s %rbp off the stack and set it to the value of %rbp
   - decrement %rsp

X86_64 has another convenience instruction for this: leaveq

invariant:
The current function’s stack frame is always between the addresses stored in %rsp and %rbp
Compiler: Returning from a function call..

Returning from a function:
1. Set \%rsp = \%rbp
2. pop \%rbp
   - pop caller’s \%rbp off the stack and set it to the value of \%rbp
   - decrement \%rsp

invariant:
The current function’s stack frame is always between the addresses stored in \%rsp and \%rbp

Back to where we started

\%rsp
\%rbp

caller stack frame

...
x86 Calling Conventions: Function Call

Initial state

push %rbp (store caller’s base pointer)

mov %rsp, %rbp (establish callee’s frame pointer)

sub $SIZE, %rsp (allocate space for callee’s locals)
x86 Calling Conventions: Function Return

x86_64 provides a convenience instruction that does all of this: `leaveq`

- `mov %rbp, %rsp` (restore caller’s stack pointer)
- `pop %rbp` (restore caller’s frame pointer)

we want to restore the caller’s frame
x86_64 Calling Convention

• The function’s return value:
  – In register %rax

• The caller’s %rbp value (caller’s saved frame pointer)
  – Placed on the stack in the callee’s stack frame

• The return address (saved PC value to resume execution on return)
  – Placed on the stack in the caller’s stack frame

• Arguments passed to a function:
  – First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  – Any additional arguments stored on the caller’s stack frame (shared with callee)
funcA:
...
callq funcB
...

funcB:
push %rbp
mov %rsp, %rbp
...

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

What do we do now?

Follow PC, fetch instruction:

```
add $5, %rcx
```
Recall: PC stores the address of the next instruction. (A pointer to the next instruction.)

What do we do now?

Follow PC, fetch instruction:

```
add $5, %rcx
```

Update PC to next instruction.

Execute the `addl`.

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...
funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
Recall: PC stores the address of the next instruction. (A pointer to the next instruction.)

What do we do now?

Follow PC, fetch instruction:

\[ \text{mov } $\text{rcx}, -8(\%\text{rbp}) \]
Recall: PC stores the address of the next instruction. (A pointer to the next instruction.)

What do we do now?

Follow PC, fetch instruction:

```
mov $rcx, -8(%rbp)
```

Update PC to next instruction.

Execute the mov.
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:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...
funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
Changing the PC: Jump

• On a (non-function call) jump:
  – Check condition codes
  – Set PC to execute elsewhere (usually not the 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:

Text Memory Region

```
funcA:
  add $5, %rcx
  mov %rcx, -8(%rbp)
  ...
  callq funcB
  add %rax, %rcx
  ...
  
  funcB:
  push %rbp
  mov %rsp, %rbp
  ...
  mov $10, %rax
  leaveq
  retq
```
Changing the PC: Functions

What we’d like this to do:

Set up function B’s stack.

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...
funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
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 %rax).

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...

funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
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 %rax).

Restore function A’s stack.

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...
funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
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!

Text Memory Region

```
funcA:
  add $5, %rcx
  mov %rcx, -8(%rbp)
  ...
  callq funcB
  add %rax, %rcx
  ...

funcB:
  push %rbp
  mov %rsp, %rbp
  ...
  mov $10, %rax
  leaveq
  retq
```
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 %rip.)

**call**

In words:  
In instructions:  

**ret**

In words:  
In instructions: 
Functions and the Stack

Executing instruction:
callq funcB

PC points to next instruction

Text Memory Region

funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...

funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
Functions and the Stack

1. `push %rip`

Text Memory Region

```
funcA:  
  add $5, %rcx
  mov %rcx, -8(%rbp)
  ...
  callq funcB
  add %rax, %rcx
  ...

funcB:  
  push %rbp
  mov %rsp, %rbp
  ...
  mov $10, %rax
  leaveq
  retq
```
Functions and the Stack

1. push %rip
2. jump funcB
3. (execute funcB)

Text Memory Region

```
funcA:
  add $5, %rcx
  mov %rcx, -8(%rbp)
  ...  
  callq funcB
  add %rax, %rcx
  ...  
funcB:
  push %rbp
  mov %rsp, %rbp
  ...  
  mov $10, %rax
  leaveq
  retq
```
Functions and the Stack

1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop %rip

Stack Memory Region

Stored PC in funcA
(Address of instruction: add %rax, %rcx)
Functions and the Stack

6. (resume funcA)

```assembly
funcA:
    add $5, %rcx
    mov %rcx, -8(%rbp)
    ...
    callq funcB
    add %rax, %rcx
    ...

funcB:
    push %rbp
    mov %rsp, %rbp
    ...
    mov $10, %rax
    leaveq
    retq
```
Recap: PC upon a Function Call

1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop %rip
6. (resume funcA)

Text Memory Region

```c
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
...
callq funcB
add %rax, %rcx
...

funcB:
push %rbp
mov %rsp, %rbp
...
mov $10, %rax
leaveq
retq
```
Functions and the Stack

1. push %rip  
2. jump funcB  
3. (execute funcB)  
4. restore stack  
5. pop %rip  
6. (resume funcA)

Return address:
Address of the instruction we should jump back to when we finish (return from) the currently executing function.

Stack Memory Region

Stored PC in funcA  
(Address of instruction: add %rax, %rcx)

Function A

...
# x86_64 Stack / Function Call Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Assembly Code</th>
</tr>
</thead>
</table>
| **push**    | Create space on the stack and place the source there. | `sub $8, %rsp`  
`mov src, (%rsp)` |
| **pop**     | Remove the top item off the stack and store it at the destination. | `mov (%rsp), dst`  
`add $8, %rsp` |
| **callq**   | 1. Push return address on stack  
2. Jump to start of function | `push %rip`  
`jmp target` |
| **leaveq**  | Prepare the stack for return (restoring caller’s stack frame) | `mov %rbp, %rsp`  
`pop %rbp` |
| **retq**    | Return to the caller, PC ← saved PC (pop return address off the stack into PC (rip)) | `pop %rip` |
• The function’s return value:
  – In register %rax

• The caller’s %rbp value (caller’s saved frame pointer)
  – Placed on the stack in the callee’s stack frame

• The return address (saved PC value to resume execution on return)
  – Placed on the stack in the caller’s stack frame

• Arguments passed to a function:
  – First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  – Any additional arguments stored on the caller’s stack frame (shared with callee)
Function Arguments

• Most functions don’t receive more than 6 arguments, so x86_64 can simply use registers most of the time.

• If we do have more than 6 arguments though (e.g., perhaps a printf with lots of placeholders), we can’t fit them all in registers.

• In that case, we need to store the extra arguments on the stack. By convention, they go in the caller’s stack frame.
If we need to place arguments in the caller’s stack frame, should they go above or below the return address?

A. Above
B. Below
C. It doesn’t matter
D. Somewhere else
If we need to place arguments in the caller’s stack frame, should they go above or below the return address?

A. Above

B. Below

C. It doesn’t matter

D. Somewhere else

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Above</td>
<td>Above</td>
</tr>
<tr>
<td>Return Address</td>
<td>Below</td>
</tr>
<tr>
<td>Caller</td>
<td></td>
</tr>
</tbody>
</table>
# x86_64 Stack / Function Call Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Assembly Code</th>
</tr>
</thead>
</table>
| **push**    | Create space on the stack and place the source there. | sub $8, %rsp  
mov src, (%rsp) |
| **pop**     | Remove the top item off the stack and store it at the destination. | mov (%rsp), dst  
add $8, %rsp |
| **callq**   | 1. Push return address on stack  
2. Jump to start of function | push %rip  
jmp target |
| **leaveq**  | Prepare the stack for return (restoring caller’s stack frame) | mov %rbp, %rsp  
pop %rbp |
| **retq**    | Return to the caller, PC ← saved PC (pop return address off the stack into PC (rip)) | pop %rip |
Arguments

- Extra 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 %rbp.
Top of the Stack

Stack Pointer %rsp

Frame or Base Pointer %rbp

both caller & callee can access these:
- push %rip (PC)
- push input arguments to callee

Caller’s Frame

Callee’s Frame or Active Frame (current frame in execution)

Caller’s Frame

Earlier Stack Frames

Bottom of Stack

Space for local & temporary vars, & saved register values

saved %rbp (Caller’s stack frame)

Return address (saved program counter)

parameter 7
(first six parameters passed as registers: rdi, rsi, rdx, rcx, r8, r9)

parameter n

higher memory address

call

lower memory address

return
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

Diagram:

```
  main
   ^
  |    function 1
  |         ^
  |         |    function 2
  |         |         ^
  |         |         |
  |         |         |
  |         |         |
  |         |         |
```
Saving Registers

- Registers are a relatively 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...

• Divide registers into two groups:

  Caller-saved: %rax, %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11
     Caller must save them prior to calling callee
     callee free to trash these,
     Caller will restore if needed

  Callee-saved: %rbx, %r12, %r13, %r14, %r15
     Callee must save them first, and restore
     them before returning
     Caller can assume these will be preserved
Running Out of Registers

- Some computations require more than 16 general-purpose 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
  - This is getting to the limits of CS 31!
    - take CS 75 (compilers) for more details.
Up next…

• Connecting Arrays, Structs, and Pointers with assembly