# The Stack & Procedures

CSE 351 Spring 2021

**Instructor:** Ruth Anderson  
**Teaching Assistants:**  
- Allen Aby  
- Catherine Guevara  
- Diya Joy  
- Aman Mohammed  
- Neil Ryan  
- Amy Xu  
- Joy Dang  
- Corinne Herzog  
- Jim Limprasert  
- Monty Nitschke  
- Alex Saveau  
- Alena Dickmann  
- Ian Hsiao  
- Armin Magness  
- Allie Pfleger  
- Sanjana Sridhar

---

**Cartoon:**

![Cartoon 1](http://xkcd.com/571/)

---
Administrivia

- **Unit Summary #1** due this Friday (4/23) on Gradescope
  - 3 Tasks – separate submission for each task

- **Lab 2 (x86-64)** due next Friday (4/30)
  - Learn to read x86-64 assembly and use GDB
  - Optional GDB Tutorial on Ed Lessons

- **Questions Docs**: Use @uw google account to access!!
  - [https://tinyurl.com/CSE351-21sp-Questions](https://tinyurl.com/CSE351-21sp-Questions)
Roadmap

C:
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();

Assembly language:
get_mpg:
    pushq %rbp
    movq %rsp, %rbp
    ...
    popq %rbp
    ret

Machine code:
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000001111

Computer system:

OS:
Windows 10
OS X Yosemite

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Reading Review

- **Terminology:**
  - Stack, Heap, Static Data, Literals, Code
  - Stack pointer (%rsp), push, pop
  - Caller, callee, return address, call, ret
    - Return value: %rax
    - Arguments: %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - Stack frames and stack discipline
Review Questions

- How does the stack change after executing the following instructions?
  ```
  pushq %rbp
  subq $0x18, %rsp
  ```
  - The stack grows by 32B.
  - `%rbp` is added to the stack.
  - The lower address is decreased by 0x18 = 24.

- For the following function, which registers do we know *must* be used?
  ```
  void* memset(void* ptr, int value, size_t num);
  ```
  - The return value is in `%rax`.
  - Arguments are in `%rdi`, `%rsi`, and `%rdx`.
  - `%rsi` is changed by `call` and `ret`.
  - `%rip` is changed while executing instructions.
Mechanisms required for procedures

1) Passing control
   - To beginning of procedure code
   - Back to return point

2) Passing data
   - Procedure arguments
   - Return value

3) Memory management
   - Allocate during procedure execution
   - Deallocate upon return

   ❖ All implemented with machine instructions!
   - An x86-64 procedure uses only those mechanisms required for that procedure
Procedures

- **Stack Structure**
- **Calling Conventions**
  - Passing control
  - Passing data
  - Managing local data
- **Register Saving Conventions**
- **Illustration of Recursion**
Simplified Memory Layout

Address Space:

- Stack
- Dynamic Data (Heap)
- Static Data
- Literals
- Instructions

What Goes Here:
- local variables and procedure context
- variables allocated with `new` or `malloc`
- `static` variables (including global variables)
- large literals/constants (`e.g., “example”`)
Memory Management

Address Space:

- **Stack**: Managed “automatically” (by compiler/assembly)
- **Dynamic Data (Heap)**: Managed “dynamically” (by programmer)
- **Static Data**: Managed “statically” (initialized when process starts)
- **Literals**: Managed “statically” (initialized when process starts)
- **Instructions**: Managed “statically” (initialized when process starts)

Who’s Responsible:

- **Stack**: Managed “automatically” (by compiler/assembly)
- **Dynamic Data (Heap)**: Managed “dynamically” (by programmer)
- **Static Data**: Managed “statically” (initialized when process starts)
- **Literals**: Managed “statically” (initialized when process starts)
- **Instructions**: Managed “statically” (initialized when process starts)
Memory Permissions

- Segmentation faults: impermissible memory access

<table>
<thead>
<tr>
<th>Address Space:</th>
<th>Permissions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>writable; not executable</td>
</tr>
<tr>
<td>Dynamic Data (Heap)</td>
<td>writable; not executable</td>
</tr>
<tr>
<td>Static Data</td>
<td>writable; not executable</td>
</tr>
<tr>
<td>Literals</td>
<td>read-only; not executable</td>
</tr>
<tr>
<td>Instructions</td>
<td>read-only; executable</td>
</tr>
</tbody>
</table>
x86-64 Stack

- Region of memory managed with stack “discipline”
  - Grows toward lower addresses
  - Customarily shown “upside-down”

- Register `%rsp` contains *lowest* stack address
  - `%rsp` = address of *top* element, the most-recently-pushed item that is not yet-popped

Last In, First Out (LIFO)

- Stack Pointer: `%rsp`
x86-64 Stack: Push

- pushq src
  - Fetch operand at src
    - Src can be reg, memory, immediate
  - Decrement %rsp by 8
  - Store value at address given by %rsp

Example:
- pushq %rcx
  - Adjust %rsp and store contents of %rcx on the stack

Stack Pointer: %rsp

1. move %rsp down (subtract) new %rsp
2. store src at %rsp

Stack “Bottom”
x86-64 Stack: Pop

- `popq dst`  
  - Load value at address given by `%rsp`  
  - Store value at `dst`  
  - **Increment** `%rsp` by 8

- **Example:**  
  - `popq %rcx`  
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`

Those bits are still there; we’re just not using them.
Procedures

- Stack Structure
- **Calling Conventions**
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Procedure Call Overview

- **Callee** must know where to find args
- **Callee** must know where to find *return address*
- **Caller** must know where to find *return value*
- **Caller** and **Callee** run on same CPU, so use the same registers
  - How do we deal with register reuse?
- Unneeded steps can be skipped (e.g. no arguments)
The convention of where to leave/find things is called the calling convention (or procedure call linkage)

- Details vary between systems
- We will see the convention for x86-64/Linux in detail
- What could happen if our program didn’t follow these conventions?
**Code Example (Preview)**

```c
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```c
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

**Compiler Explorer:**
https://godbolt.org/z/ndro9E
Procedure Control Flow

- Use stack to support procedure call and return

Procedure call: \texttt{call label} \quad \text{(special push)}

1) Push return address on stack \texttt{(why? which address?)}
2) Jump to \texttt{label}

\begin{itemize}
  \item [1)]  \texttt{\textcolor{red}{(1)} move %rsp down}
  \item [2)]  \texttt{\textcolor{red}{(2)} store ret addr at %rsp}
  \item [3)]  \texttt{\textcolor{red}{(3)} label \rightarrow %rip}
\end{itemize}
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** call label
  1) Push return address on stack (*why? which address?*)
  2) Jump to label
- **Return address:**
  - Address of instruction immediately after `call` instruction
  - Example from disassembly:
    
    ```
    400544: call 400550 <mult2>
    400549: movq %rax,(%rbx)
    ```
    
    Return address = 0x400549
- **Procedure return:** ret  
  1) Pop return address from stack
  2) Jump to address

next instruction happens to be a move, but could be anything
Procedure **Call Example** (step 1)

```
0000000000400540 <multstore>:
  ...
  400544: call 400550 <mult2>
  400549: movq %rax, (%rbx)
  ...

0000000000400550 <mult2>:
  400550: movq %rdi, %rax
  ...
  400557: ret
```

1. Push return address on stack
2. Jump to label
Procedure **Call Example** (step 2)

```
0000000000400540 <multstore>:
  ...
  400544: call 400550 <mult2>
  400549: movq %rax,(%rbx)
  ...

0000000000400550 <mult2>:
  400550: movq %rdi,%rax
  400557: ret
```

Diagram:
- `%rsp` at 0x118
- `%rip` at 0x400550
Procedure **Return Example** (step 1)

00000000000400540 <multstore>:

```
   400544: call 400550 <mult2>
   400549: movq %rax, (%rbx)
```

00000000000400550 <mult2>:

```
   400550: movq %rdi, %rax
   400557: ret
```

1. Pop `ret` addr off stack
2. Jump to `ret` addr
**Procedure Return Example** (step 2)

000000000000400540 <multstore>:
  
  400544: call 400550 <mult2>
  400549: movq %rax, (%rbx)
  
000000000000400550 <mult2>:
  
  400550: movq %rdi, %rax
  
  400557: ret
Procedures

- Stack Structure
- **Calling Conventions**
  - Passing control
  - **Passing data**
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion
Procedure Data Flow

Registers (NOT in Memory)
- First 6 arguments
  1. %rdi
  2. %rsi
  3. %rdx
  4. %rcx
  5. %r8
  6. %r9

- Return value
  - %rax

Stack (Memory)
- Only allocate stack space when needed
  - High Addresses
  - Low Addresses

Diane’s Silk Dress Costs $89

- Only allocate stack space when needed
  - %rdi
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9
  - %rax
x86-64 Return Values

- By convention, values returned by procedures are placed in %rax
  - Choice of %rax is arbitrary

1) **Caller** must make sure to save the contents of %rax before calling a **callee** that returns a value
  - Part of register-saving convention

2) **Callee** places return value into %rax
   - Any type that can fit in 8 bytes – integer, float, pointer, etc.
   - For return values greater than 8 bytes, best to return a *pointer* to them

3) Upon return, **caller** finds the return value in %rax
Data Flow Examples

```c
void multstore( long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}

long mult2( long a, long b) {
    long s = a * b;
    return s;
}
```

```
0000000000400540 <multstore>:
    # x in %rdi, y in %rsi, dest in %rdx
    ...
    400541: movq %rdx,%rbx         # "Save" dest
    400544: call 400550 <mult2>    # mult2(x,y)
    # t in %rax
    400549: movq %rax,(%rbx)      # Save at dest
    ...

0000000000400550 <mult2>:
    # a in %rdi, b in %rsi
    400550: movq %rdi,%rax         # a
    400553: imulq %rsi,%rax       # a * b
    # s in %rax
    400556: ret                    # Return
```

Procedures

- Stack Structure

- **Calling Conventions**
  - Passing control
  - Passing data
  - Managing local data

- Register Saving Conventions

- Illustration of Recursion
Stack-Based Languages

- Languages that support recursion
  - *e.g.* C, Java, most modern languages
  - Code must be *re-entrant*
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store *state* of each instantiation
    - Arguments, local variables, return address

- Stack allocated in *frames*
  - State for a single procedure instantiation

- Stack discipline
  - State for a given procedure needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does
Call Chain Example

Procedure `amI` is recursive
(calls itself)
1) Call to whoa

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

The diagram shows the stack frame for the `whoa` procedure with the frame pointer `%rbp` and the return stack pointer `%rsp`. The call to `whoa` is depicted with arrows indicating the procedure call and return. The stack frame for the `main` procedure is also shown, with the `whoa` procedure nested inside it.
2) Call to who

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

Stack

```
%rbp

%rsp

“create” frame by manipulating %rsp

whoa

who

amI

amI

amI

amI

whoa

who

amI

amI

amI

amI
```
3) Call to `amI (1)`
4) Recursive call to `amI` (2)
5) (another) **Recursive call to amI (3)**

```
whoa(...) {
  who(...) {
    amI(...) {
      amI(...) {
        if() {
          amI()
        }
      }
    }
  }
}
```

[Stack diagram]

```
whoa

who

amI

amI

amI

%rbp

%rsp
```
6) Return from \(\text{(another) recursive call to amI}\)

```plaintext
whoa(…)
{
  who(…)
  {
    amI(…)
    {
      amI(…)
      {
        amI()
        if()
        amI()
      }
      .
    }
    .
  }
  .
}
```
7) **Return from recursive call to amI**

```
whoa(...)
{
  who(...)
  {
    amI(...)
    {
      •
      if(){
        amI()
      }
    }
  } 
} 
```

Stack:

- whoa
- who
- amI
- %rbp
- %rsp
- amI₁
- amI₂
- amI₃
8) Return from call to `amI`

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

Stack:

- `whoa`
- `who`
- `amI` (new stack frame overwrites old data)

Registers:

- `%rbp`
- `%rsp`
9) (second) **Call to amI (4)**

```
whoa(…)
{
  who(…)
  {
    amI(…)
    {
      •
      if(){
        amI()
      }
    }
    •
  }
}
```
10) Return from (second) call to \texttt{amI}

\texttt{whoa}(\ldots)
{
  \texttt{who}(\ldots)
  {
    .
    \texttt{amI}();
    .
    \texttt{amI}();
  }
}

\begin{center}
\textbf{Stack}
\end{center}

\begin{tikzpicture}
  \node[rectangle, fill=green!20] at (0,0) {
    \texttt{whoa}
  };
  \node[rectangle, fill=green!20] at (0,-1) {
    \texttt{who}
  };
  \node[rectangle, fill=green!20] at (0,-2) {
    \texttt{amI}
  };
  \node[rectangle, fill=green!20] at (0,-3) {
    \texttt{amI}
  };
  \node[rectangle, fill=green!20] at (0,-4) {
    \texttt{amI}
  };
  \node[rectangle, fill=green!20] at (0,-5) {
    \%rbp
  };
  \node[rectangle, fill=green!20] at (0,-6) {
    \%rsp
  };
  \node[rectangle, fill=green!20] at (0,-7) {
    \texttt{amI}_4
  };
  \node[rectangle, fill=green!20] at (0,-8) {
    \texttt{amI}_2
  };
  \node[rectangle, fill=green!20] at (0,-9) {
    \texttt{amI}_3
  };
\end{tikzpicture}
11) Return from call to who

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

Stack

```
call chain: main (1)

whoa (2)
who (3)
amI (4)
amI (5)
amI (6)
```

Total stack frames created: 7
Maximum stack depth: 6 frames
x86-64/Linux Stack Frame

- **Caller’s Stack Frame**
  - Extra arguments (if > 6 args) for this call

- **Current/Callee Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Old frame pointer (optional)
  - Saved register context (when reusing registers)
  - Local variables (If can’t be kept in registers)
  - “Argument build” area (If callee needs to call another function - parameters for function about to call, if needed)