The Stack & Procedures
CSE 351 Spring 2021

<table>
<thead>
<tr>
<th>Instructor:</th>
<th>Teaching Assistants:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ruth Anderson</td>
<td>Allen Aby</td>
</tr>
<tr>
<td></td>
<td>Catherine Guevara</td>
</tr>
<tr>
<td></td>
<td>Diya Joy</td>
</tr>
<tr>
<td></td>
<td>Aman Mohammed</td>
</tr>
<tr>
<td></td>
<td>Neil Ryan</td>
</tr>
<tr>
<td></td>
<td>Amy Xu</td>
</tr>
<tr>
<td></td>
<td>Joy Dang</td>
</tr>
<tr>
<td></td>
<td>Corinne Herzog</td>
</tr>
<tr>
<td></td>
<td>Jim Limprasert</td>
</tr>
<tr>
<td></td>
<td>Monty Nitschke</td>
</tr>
<tr>
<td></td>
<td>Alex Saveau</td>
</tr>
<tr>
<td></td>
<td>Alena Dickmann</td>
</tr>
<tr>
<td></td>
<td>Ian Hsiao</td>
</tr>
<tr>
<td></td>
<td>Armin Magness</td>
</tr>
<tr>
<td></td>
<td>Allie Pfleger</td>
</tr>
<tr>
<td></td>
<td>Sanjana Sridhar</td>
</tr>
</tbody>
</table>

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:

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

Java:

```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
11000001111110101000011111
```

OS:

- Windows 10
- OS X Yosemite

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Processes
Virtual memory
Memory & caches
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`

- For the following function, which registers do we know must be used?
  - `void* memset(void* ptr, int value, size_t num);`
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

int Q(int i) {
    int t = 3*i;
    int v[10];
    ...
    return v[t];
}

P(...) {
    ...
    y = Q(x);
    print(y)
    ...
}
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”)**
- **program code**
Memory Management

Address Space:

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

Who’s Responsible:

- Managed “automatically” (by compiler/assembly)
- Managed “dynamically” (by programmer)
- Managed “statically” (initialized when process starts)
- Managed “statically” (initialized when process starts)
- Managed “statically” (initialized when process starts)
Memory Permissions

- Segmentation faults: impermissible memory access
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 = \text{address of top element, the most-rec}}{}

Stack Pointer: $\%rsp$

- Stack “Top”
- Stack “Bottom”
- High Addresses
- Increasing Addresses
- Stack Grows Down
- Low Addresses 0x00...00
**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` 

Stack Pointer: `%rsp` 

Stack “Bottom” 

Stack “Top”
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`

Stack Pointer: `%rsp` + 8

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)*
Procedure Call Overview

- 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?
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;
}
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*
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
Procedure Call Example (step 2)

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

0000000000400550 <mult2>:
  
  400550: movq %rdi,%rax
  
  400557: ret
Procedure Return Example (step 1)

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

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

%rip 0x400557
%rsp 0x118
Procedure Return Example (step 2)

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

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

%rip 0x400549
%rsp 0x120
0x120
0x128
0x130
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
  - %rdi
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9

- Return value
  - %rax

Stack (Memory)

- Only allocate stack space when needed

High Addresses

Low Addresses

0x00...00

Diane’s Silk Dress Costs $89
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;
}
```

```assembly
000000000000400540 <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
    ...

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

```assembly
000000000000400550 <mult2>:
    # a in %rdi, b in %rsi
400550: movq %rdi,%rax   # a
400553: imulq %rsi,%rax  # a * b
    # s in %rax
400557: 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 \texttt{amI} is recursive (calls itself)
1) Call to whoa

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

Stack:

```
whoa
/
|
who
/
|
ami ami
/
|
ami ami
```

- %rbp
- %rsp
2) Call to who

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

Diagram:
- `whoa` on top, with `who` inside it.
- `ami()` calls are stacked below.
- Stack diagram shows `whoa` and `who` with `ami()` calls.

```
Stack

whoa

who

ami

ami

ami

%rbp

%rsp
```
3) Call to `amI (1)`

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

### Stack

```
whoa

who

amI

amI

%rbp

%rsp

amI_1
```
4) Recursive call to `amI(2)`

```plaintext
whoa(…)
{
  who(…)
  {
    amI(…)
    {
      amI(…)
      {
        if(){
          amI()
        }
      }
    }
  }
}
```
5) (another) Recursive call to `amI(3)`
6) Return from (another) recursive call to amI
7) Return from recursive call to `amI`
8) Return from call to amI

whoa (…)
{
  who (…)
  {
    •
    •
    •
    amI();
  }
  amI();
}

Stack

%rbp

%rsp

amI1

amI2

amI3
9) (second) Call to `amI (4)`
10) Return from (second) call to amI

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

Stack

```
whoa

who

amI  amI

%rbp

%rsp

amI

amI

amI4

amI2

amI3
```
11) Return from call to who

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

Stack

- `whoa`
- `who`
- `amI`
- `amI`
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)