The Stack & Procedures
CSE 351 Summer 2019

Instructor:
Sam Wolfson

Teaching Assistants:
Rehaan Bhimani
Corbin Modica

Daniel Hsu

corn
stack
popcorn
Administrivia

- Homework 2 due TONIGHT Wednesday (7/17)
- Lab 2 (x86-64) due Monday (7/22)
- Homework 3, coming soon
  - On midterm material, but due after the midterm
- Section tomorrow on Assembly and GDB
  - Bring your laptops!
- Midterm (Fri 7/26, 10:50-11:50am)
  - You are allowed one double-sided, handwritten 8.5x11” sheet of notes
  - Find a study group! Look at past exams!
  - Review session: Wednesday, 7/24, 4-6pm, location TBD
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:

```
011101000011000
100011010000010000000010
1000100111000010
1100000111110100001111
```

OS:

- Windows 10
- OS X Yosemite

Memory & data
Integers & floats
x86 assembly

Procedures & stacks
Executables
Arrays & structs
Processes
Virtual memory
Memory allocation
Java vs. C

Computer system:
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
   - De-allocate 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

Memory Addresses

Low Addresses

High Addresses

2^{N-1}

Stack

Dynamic Data (Heap)

Static Data

Literals

Instructions

local variables; procedure context

variables allocated with new or malloc

static variables (including global variables (C))

large constants (e.g. “example”)

program code

Instructions

Literals

Static Data

Dynamic Data (Heap)

Stack

Memory Addresses

Low Addresses

High Addresses

2^{N-1}
Memory Permissions

- **Stack**: Managed “automatically” (by compiler)
  - writable; not executable

- **Dynamic Data (Heap)**: Managed by programmer
  - writable; not executable

- **Static Data**: Initialized when process starts
  - writable; not executable

- **Literals**: Initialized when process starts
  - read-only; not executable

- **Instructions**: Initialized when process starts
  - read-only; executable

Segmentation faults?
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

Stack Pointer: %rsp

Stack “Bottom”

Stack Grows Down

Low Addresses

0x00…00

Stack “Top”

High Addresses

Increasing Addresses
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 "Top"

Stack "Bottom"

High Addresses

Increasing Addresses

Stack Grows Down

Low Addresses

0x00...00
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)
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?
Code Example (Preview)

```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;
}
```

Compiler Explorer: https://godbolt.org/g/cKKDZn

```
0000000000400540 <multstore>:
    400540: push %rbx       # Save %rbx
    400541: movq %rdx,%rbx   # Save dest
    400544: call 400550 <mult2> # mult2(x,y)
    400549: movq %rax,(%rbx) # Save at dest
    40054c: pop %rbx         # Restore %rbx
    40054d: ret              # Return

0000000000400550 <mult2>:
    400550: movq %rdi,%rax   # a
    400553: imulq %rsi,%rax  # a * b
    400557: ret              # Return
```
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:
    
    | Address     | Instruction               |
    |-------------|---------------------------|
    | 400544:     | `call 400550 <mult2>`    |
    | 400549:     | `movq %rax, (%rbx)`      |
    |             | Return address = 0x400549 |

- **Procedure return:** `ret`
  - 1) Pop return address from stack
  - 2) Jump to address
Procedure Call Example (step 1)

00000000000400540 <multstore>:

00000000000400544: call 400550 <mult2>
00000000000400549: movq %rax,(%rbx)

00000000000400550 <mult2>:

00000000000400554: movq %rdi,%rax
00000000000400557: ret
Procedure Call Example (step 2)

00000000000400540 <multstore>:

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

00000000000400550 <mult2>:

400550: movq %rdi,%rax

0x118 %rsp
0x400549
0x400550 %rip

Memory

Procedure Call Example (step 2)
Procedure Return Example (step 1)

0000000000400540 <multstore>:

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

0000000000400550 <mult2>:

400550: movq %rdi,%rax

0x400549

Memory

%rsp 0x118
%rip 0x400557

0x118
0x120
0x128
0x130
Procedure Return Example (step 2)

0000000000400540 <multstore>:

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

0000000000400550 <mult2>:

400550: movq %rdi,%rax

Procedure Return Example (step 2)
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 $8 9
  - Arg n
  - Arg 8
  - Arg 7
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;
}
```

```asm
 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
    ...
```

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

```asm
 0000000000400550 <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 `amI` is recursive (calls itself)
1) Call to `yoo`

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

Stack

```plaintext
yoo

who

ami

ami

ami

%rbp

%rsp
```
2) Call to who

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

Stack

```
%rbp
%rsp
```

```
yoo
who
amI
amI
amI
```
3) Call to `amI (1)`

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

Stack:
- `yoo`
- `who`
- `amI`
- `amI_1`
4) Recursive call to `amI` (2)

```plaintext
yoo(...) {
  who(...) {
    amI(...) {
      if() {
        amI()
      }
    }
  }
}
```

Stack:
- `yoo`
- `who`
- `amI_1`
- `amI_2`

%rbp → %rsp
5) (another) Recursive call to \texttt{amI (3)}

```c
yoo (...)[
    { who (...)[
        { amI (...)[
            { amI (...)[
                { amI ()[
                    if (){
                        amI()
                    }
                ]
            ]
        ]
    ]
```
6) Return from (another) recursive call to `amI`

```
yoo(…)
{
    who(…)
    {
        amI(…)
        {
            if(){
                amI()
            }
        }
    }
}
yoo
who
amI
amI
amI
Stack
yoo
who
amI$_1$
amI$_2$
amI$_3$
%rbp
%rsp
```
7) **Return from recursive call to `amI`**

```plaintext
yoo (...) {
  who(...) {
    amI(...) {
      .
      if() {
        amI()
      }
    }
    .
  }
}
```

**Stack**

- `yoo`
- `who`
- `amI_1`
- `amI_2`
- `amI_3`

- `%rbp` (Right Base Pointer)
- `%rsp` (Right Stack Pointer)
8) Return from call to amI

```c
yoo (...) {
  who(...) {
    • amI();
    • amI();
  }
}
```

Stack

```plaintext
yoo
who
amI
%rbp

%rsp

amI

amI1

amI2

amI3
```
9) (second) Call to `amI (4)`

```
yoo (…)
{   who (...)
    {       amI (...)
        {   •
            if (){
                amI ()
            }
        }
        •
    }
}
```

Stack:

```
yoo
      who
        amI
          amI
            amI
                %rbp
                %rsp
```

```
amI_4
amI_2
amI_3
```
10) Return from (second) call to `amI`

```
yoo(...)
{
who(...)
{

  •
  amI();

  •
  amI();

}
}
```
11) Return from call to \texttt{who}

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

Stack:

\[
\begin{array}{c}
\text{yoo} \\
\text{who} \\
\text{amI} \\
\text{amI} \\
\hline
%rbp \\
%rsp \\
\end{array}
\]
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)