Procedures & The Stack I
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun

http://xkcd.com/1270/
Administrivia

- Homework 1 due today
- Lab 2 due next Friday

**Midterm** on Nov. 2 in lecture

- Changed my mind – you get 1 *handwritten* cheat sheet
- Still get [reference sheet](#), which has been updated (floats)
- I will attempt to release old exam problems I’ve written that are relevant to this midterm (harder than what you’ll see)
- Historically my exams have averages of 65-70%

**Midterm review session: 5-7pm on Monday, Oct. 31**

- Look for additional staff office hours as well
Roadmap

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

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

Machine code:
```
0111010000011000
100011010000010000000010100010011100001011000001111110
```

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

Computer system:

OS:
- Windows 8
- Mac

Memory & data
- Integers & floats
- Machine code & C
- x86 assembly

Procedures & stacks
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C
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
Questions to answer about procedures

- How do I pass arguments to a procedure?
- How do I get a return value from a procedure?
- Where do I put local variables?
- When a function returns, how does it know where to return?

To answer some of these questions, we need a call stack...
Procedures

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

- Stack
  - Local variables; procedure context
- Dynamic Data (Heap)
  - Variables allocated with `new` or `malloc`
- Static Data
  - Static variables (including global variables (C))
- Literals
- Instructions

Memory Addresses

High Addresses

Low Addresses

2^{N-1}

Addresses

0

Large constants (e.g., “example”)
## Memory Permissions

<table>
<thead>
<tr>
<th>Layer</th>
<th>Permissions</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Writable; not executable</td>
<td>Managed “automatically” (by compiler)</td>
</tr>
<tr>
<td>Dynamic Data (Heap)</td>
<td>Writable; not executable</td>
<td>Managed by programmer</td>
</tr>
<tr>
<td>Static Data</td>
<td>Read-only; not executable</td>
<td>Initialized when process starts</td>
</tr>
<tr>
<td>Literals</td>
<td>Read-only; executable</td>
<td>Initialized when process starts</td>
</tr>
<tr>
<td>Instructions</td>
<td>Read-only; executable</td>
<td>Initialized when process starts</td>
</tr>
</tbody>
</table>

**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
  - $\%rsp = \text{address of top element, the most-recently-pushed item that is not-yet-popped}$

Stack Pointer: $\%rsp$

![Stack Diagram]
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 \(\rightarrow\) -8
x86-64 Stack: Pop

- `popq dst`
  - Load value at address given by `%rsp`
  - Store value at `dst` (must be register)
  - **Increment** `%rsp` by 8

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

Stack Pointer: `%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 Examples

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
  2) Jump to `label`
- Return address:
  - Address of instruction immediately after `call` instruction
  - Example from disassembly:
    
    ```
    400544: callq 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)

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

0x120
0x128
0x120

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

%rip 0x400544
%rsp 0x120
Procedure **Call Example (step 2)**

```
0000000000400540 <multstore>:
    .
    .
400544: call 400550 <mult2>
400549: movq %rax,(%rbx)
    .
    .
400557: ret
```

```
0000000000400544: call 400550 <mult2>
400549: movq %rax,(%rbx)
    .
    .
400557: ret
```
Procedure Return Example (step 1)

0000000000400540 <multstore>:
  
  400544: call 400550 <mult2>
  400549: movq %rax, (%rbx)
  
  400550:  movq %rdi, %rax
  
  400557: ret

0x400549
Procedure Return Example (step 2)

```assembly
0000000000400540 <multstore>:

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

0000000000400550 <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
  - %rdi
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9

- Return value
  - %rax

Stack – in Memory!

- Only allocate stack space when needed

Diary’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;
}
```

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

```
00000000000400540 <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
    ...
```

```
00000000000400550 <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 pointer

- 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

Example Call Chain

Procedure `amI` is recursive (calls itself)

```
yoo (...) {
  •
  • who();
  •
}

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

amI(...) {
  •
  if(...){
    amI()
  }
  •
}
1) Call to yoo

```cpp
yoo (...) {
    •
    •
    who();
    •
}
```
2) Call to who

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

Stack

- `%rbp`
- `%rsp`

- `who`
- `yoo`

Flow chart:
- `yoo` to `who`
- `who` calls `amI`
- `amI` calls `amI`
- `amI` calls `amI`
3) Call to amI (1)

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

Stack:

```
yoo
who
amI
amI
```

Pointer:

```
%rbp
%rsp
```
4) Recursive call to amI (2)
5) (another) Recursive call to `amI (3)`

```c
yoo(…)
{
    who(…)
    {
        amI(…)
        {
            amI(…)
            {
                if()
                amI()
            }
        }
    }
}
```
6) **Return from (another) recursive call to `amI`**

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

```
yoo
  who
    amI
      amI
        if/
```

Stack:
- `yoo`
- `who`
- `amI_1`
- `amI_2`
- `amI_3`

```c
%rbp
%rsp
```
7) Return from recursive call to amI
8) Return from call to `amI`

```c
yoo(…)
{
    who(…);
    amI();
    amI();
}
```

Diagram:
- `yoo` function
- `who` function
- `amI` function
- Stack diagram with `%rbp` and `%rsp`
9) (second) Call to `amI(4)`
10) Return from (second) call to `amI`
11) Return from call to who

```c
yoo (...) {
  .
  .
  who();
  .
}
```
x86-64/Linux Stack Frame

- **Caller’s Stack Frame**
  - Extra arguments (if > 6 args) for this call
  - Return address
    - Pushed by `call` instruction

- **Current/Callee Stack Frame**
  - Old frame pointer (optional)
  - Saved register context (when reusing registers)
  - “Argument build” area (If callee needs to call another function - parameters for function about to call, if needed)
Peer Instruction Question

Answer the following questions about when `main()` is run (assume \texttt{x} and \texttt{y} stored on the Stack):

- How many total stack frames are created?
- What is the maximum *depth* (# of frames) of the Stack?
- *Higher/larger address*: \texttt{x} or \texttt{y}?

```c
int main() {  
    int i, x = 0;  
    for (i=0; i<3; i++)  
        x = randSum(x);  
    printf("x = %d\n", x);  
    return 0;  
}
```

```c
int randSum(int n) {  
    int y = rand()%20;  
    return n+y;  
}
```