The Stack & Procedures
CSE 351 Summer 2019

Instructor: Sam Wolfson

Teaching Assistants:
Rehaan Bhimani
Corbin Modica
Daniel Hsu
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
kar *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
110000011111101000011111
```

OS:

- Windows 10
- OS X Yosemite

Memory & data
- Integers & floats

x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Processes
- Virtual memory
- Memory & caches
- 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
   - De-allocate upon return

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

```c
P(...) {
    
    y = Q(x);
    print(y)
    
}

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

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

- **Memory Addresses**
  - High Addresses: $2^{N-1}$
  - Low Addresses: 0

- **Stack**
  - Local variables; procedure context

- **Dynamic Data (Heap)**
  - Variables allocated with `new` or `malloc`

- **Static Data**
  - Static variables (including global variables (C))

- **Literals**
  - Large constants (e.g. "example")

- **Instructions**
  - Program code
Memory Permissions

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

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

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

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

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

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 Diagram](image-url)
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`
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;
}
```

```c
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:

  ```
  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

Memory
Procedure Call Example (step 2)

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

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

Memory

%rip 0x400550
%rsp 0x118

0x118
0x120
0x128
0x130
Procedure Return Example (step 1)

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

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

Memory

0x400549
0x130
0x128
0x120
0x118
0x400549
0x118
120
0x400557
0x400549
Procedure Return Example (step 2)

00000000000400540 <multstore>:
  .
  .
  400544: call 400550 <mult2>
  400549: movq %rax,(%rbx)
  .
  .

00000000000400550 <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 (Memory)
- • • •
- Arg n
- • • •
- Arg 8
- Arg 7

• Only allocate stack space when needed
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;
}
```

### Mult2 Assembly Code
```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550: movq %rdi,%rax  # a
  400553: imulq %rsi,%rax # a * b
  400557: ret  # Return
```

### Multstore Assembly Code
```
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
```

[CSE 351, Summer 2019]

L11: The Stack & Procedures
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

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

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

amI (...)
{
  
  if (...){
    amI ()
  }
  
}

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

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

Stack diagram showing the call to `yoo` and the function `who` with arguments `ami ami ami ami`.
2) Call to who

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

```
Stack
```

```
yoo
```

```
who
```

```
amI
```

```
amI
```

```
%rbp
```

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

Stack

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

%rbp
%rsp
6) Return from (another) recursive call to `amI`
7) Return from recursive call to `amI`
8) Return from call to `amI`

```c
8)  Return from call to amI

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

```
Stack

yoo

who

amI  amI

%rbp

%rsp

amI_1

amI_2

amI_3
```
9) (second) Call to `amI (4)`

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

stack
%rbp
%rsp

Stack
```
10) Return from (second) call to `amI`

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

Stack:
```
yoo
who
%rbp
%rsp
amI
amI
amI
```

Return from (second) call to `amI`.
11) Return from call to who

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

```
        main
            ↓
              yoo

        who
            ↑
              ami

        ami
            ↑
              ami
```

---

**Stack**

- yoo
- who
- ami
- ami

**Total frames:** 7
**Max depth:** 6
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)
Peer Instruction Question

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

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

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

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

A. 1  B. 2  C. 3  D. 4

Vote only on 3rd question at http://pollev.com/wolfson