The Stack & Procedures
CSE 351 Autumn 2017

Instructor:
Justin Hsia

Teaching Assistants:
Lucas Wotton
Michael Zhang
Parker DeWilde
Ryan Wong
Sam Gehman
Sam Wolfson
Savanna Yee
Vinny Palaniappan

http://xkcd.com/648/
Administrivia

- Homework 2 due tonight
- Lab 2 due next Friday (10/27)
  - Ideally want to finish well before the midterm
- Homework 3 released next week
  - On midterm material, but due after the midterm

- **Midterm** (10/30, 5-6:30pm, KNE 120)
  - Reference sheet + 1 *handwritten* cheat sheet
  - Find a study group! Look at past exams!
  - Average is typically around 70%
- **Review session** (10/27) in EEB 105 from 5:30-7:30pm
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
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:**

1. Move `%rsp` down (subtract)
2. Store src at `%rsp`

High 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` (must be register)
  - *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/g/cKKDZn

---

**mult2**

```
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
```

**multstore**

```
0000000000400550 <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
```

**mult2**

```
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` *(special push)*
  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` *(special push)*
  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` *(special pop)*
  1) Pop return address from stack *(read ret addr at %rsp into %rip)*
  2) Jump to address *(move %rsp up)*

```
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

%rip 0x400550

%rsp 0x118

0x400549 0x118

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

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

0000000000400550 <mult2>:
- 
400550: `movq %rdi,%rax`
- 
- 
400557: `ret`
Procedure **Return Example** *(step 2)*

```
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 (Memory)

- Only allocate stack space when needed

Diane’s Silk Dress Costs $89

High Addresses

Low Addresses

0x00...00
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
long mult2(long a, long b)
{
    long s = a * b;
    return s;
}
```

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

```
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
    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

Procedure \texttt{amI} is recursive
(calls itself)
1) Call to `yoo`

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

![Stack Diagram](image)

- `yoo` is a procedure.
- `who()` is called within the `yoo` procedure.
- The stack frame includes pointers `%rbp` and `%rsp`.
- The call to `who()` is shown at the top of the stack frame.
- The stack frame includes the `yoo` procedure.

Note: The diagram illustrates the calling context and stack layout for the function call to `yoo`.
2) Call to `who`

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

Stack

```
Stack  
      
%rbp  
% rsp

"create" frame by manipulating %rsp
```
3) Call to `amI` (1)

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

Stack

```
yoo
who
amI
amI
%
rbp
%
rsp
amI₁
```
4) Recursive call to `amI (2)`

The diagram illustrates the recursive call to `amI` within the `yoo` function. The stack trace shows the call stack for the function calls, with `yoo`, `who`, `amI`, and `amI_1` at the top of the stack, indicating the recursive nature of the `amI` function call.
5) (another) Recursive call to `amI` (3)

```plaintext
yoo(…)
{  
  who(…)
  {  
    amI(…)
    {  
      amI(…)
      {  
        if(){
          amI()
        }
      }
    }
  }
}
```
6) Return from (another) recursive call to \texttt{amI}

```
yoo(...)
{
    who(...)
    {
        amI(...)
        {
            amI(...)
            {
                if()
                amI()
            }
        }
    }
    amI();
}
```
7) Return from recursive call to \texttt{amI}
8) Return from call to `amI`

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

Stack

- `yoo`
- `who`
- `amI`
- `amI`
- `amI1`
- `amI2`
- `amI3`

- `%rbp`
- `%rsp`

(new stack frame overwrites old data)
9) (second) Call to `amI (4)`

The call to `amI` at line 9 is shown in the stack diagram. The stack frame for `amI` contains arguments and local variables. The call is made from the `who` function, which itself is called from `yoo`. The stack frame for `amI` includes the return address `%rbp` and `%rsp`, indicating the state of the stack before the function call.

The diagram illustrates the recursive nature of the `amI` function, with the recursive call `amI()` shown within the `if` block, leading to another call to `amI`.

The stack frame for `yoo` shows variables and the previous stack frames are also depicted, reflecting the call stack as the function calls unfold and return values are pushed and popped from the stack.
10) Return from (second) call to `amI`
11) Return from call to who

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

call chain: main (0)
```
yoo (2)
  who (3)
    ami (5)
      ami (6)
```

Stack
```
main
  yoo
    who
      ami
        ami

%rbp
%rsp
```

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)
Peer Instruction Question

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

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

```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;
}
```

Vote only on 3rd question at [http://PollEv.com/justinh](http://PollEv.com/justinh)
Example: 

```c
long increment(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

Register Use(s)

- `%rdi`: 1st arg (`p`)
- `%rsi`: 2nd arg (`val`), `y`
- `%rax`: `x`, return value

**Increment:**

- `movq (%rdi), %rax`  # `x=*p`
- `addq %rax, %rsi`  # `y=x+val`
- `movq %rsi, (%rdi)`  # `*p=y`
- `ret`

The code is written this way to correspond to assembly language. The assembly code for the increment function is:

```
movq (%rdi), %rax
addq %rax, %rsi
movq %rsi, (%rdi)
ret
```

Adding `val` to `x` and store it at `p`.

Incr. C and incr. S posted on website so you can step through this example in gdb.

```c
increment: movq (%rdi), %rax  
addq %rax, %rsi  
movq %rsi, (%rdi)  
ret```

```
movq (%rdi), %rax
addq %rax, %rsi
movq %rsi, (%rdi)
ret
```
Procedure Call Example (initial state)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

- **Return address on stack** is the address of the instruction immediately following the call to `call_incr`
  - Shown here as `main`, but could be anything
  - Pushed onto stack by `call call_incr`

Return address on stack is the address of instruction immediately **following** the call to “call_incr”
Procedure Call Example (step 1)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

- Setup space for local variables
  - Only `v1` needs space on the stack
- Compiler allocated extra space
  - Often does this for a variety of reasons, including alignment

### Stack Structure

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return addr &lt;main+8&gt;</td>
<td>351</td>
</tr>
<tr>
<td>Unused</td>
<td></td>
</tr>
</tbody>
</table>

Allocate space for local vars “manual push”

- Setup space for local variables
  - Only `v1` needs space on the stack
- Compiler allocated extra space
  - Often does this for a variety of reasons, including alignment
Procedure Call Example (step 2)

```c
long call_incr() {
    long v1 = 351;
    %rdi, %rsi;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**Stack Structure**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>100</td>
</tr>
</tbody>
</table>

Aside: movl is used because 100 is a small positive value that fits in 32 bits. High order bits of rsi get set to zero automatically. It takes one less byte to encode a movl than a movq.
Procedure Call Example (step 3)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

- **State while inside** `increment`
  - **Return address** on top of stack is address of the `addq` instruction immediately following call to `increment`
Procedure Call Example (step 4)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

**Stack Structure**

```
Return addr <main+8>
+----------------+
| 451            |
| Unused         |
+----------------+
Call addr <call_incr+>
```

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>351</td>
</tr>
</tbody>
</table>

**Increment:**

1. `movq (%rdi), %rax` # x = *p
2. `addq %rax, %rsi` # y = x+100
3. `movq %rsi, (%rdi)` # *p = y

**State while inside increment**

- **After** code in body has been executed

- **Return addr**
  - <main+8>
  - <call_incr+>
Procedure Call Example (step 5)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

- After returning from call to `increment`
  - Registers and memory have been modified and return address has been popped off stack

Stack Structure

- Return addr `<main+8>`
- `call_incr` call
- `%rsp+8`
- `%rsp`

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>&amp;v1</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>451</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>351 (v2)</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 6)

long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}

Stack Structure

<table>
<thead>
<tr>
<th>Return addr</th>
<th>&lt;main+8&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp+8</td>
<td>451</td>
</tr>
<tr>
<td>Unused</td>
<td></td>
</tr>
</tbody>
</table>

Update %rax to contain v1+v2

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>451+351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 7)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**Stack Structure**

- Return addr <main+8>
- 451
- Unused

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>

De-allocate space for local vars
(make sure %rsp points to return addr before ret)
Procedure Call Example (step 8)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

**Stack Structure**

- Return addr <main+8>
- `%rsp`
- Popped off stack into `%rip` by `ret`

- **State just before** returning from call to `call_incr`

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>&amp;v1</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>451</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>802</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 9)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

- State immediately after returning from call to `call_incr`
  - Return addr has been popped off stack
  - Control has returned to the instruction immediately following the call to `call_incr` (not shown here)

```
call_incr:
    subq $16, %rsp
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq 8(%rsp), %rax
    addq $16, %rsp
    ret
```