The Stack & Procedures
CSE 351 Summer 2018

Instructor:
Justin Hsia

Teaching Assistants:
Josie Lee Natalie Andreeva Teagan Horkan

http://xkcd.com/292/
Administrivia

- Homework 2 due tonight
- Lab 2 due Monday (7/16)
- Homework 3 released
  - On midterm material, but due after the midterm

- **Midterm** (7/18, in lecture)
  - Reference sheet + 1 *handwritten* cheat sheet
  - Find a study group! Look at past exams!
  - Average is typically around 70%
  - **Review session** (7/16) in EEB 045 from 5:00-6:30 pm
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:
      ```assembly
      400544: call 400550 <mult2>
      400549: movq %rax,(%rbx)
      Return address = 0x400549
      ```
  - Procedure return: `ret` (special pop)
    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
```

Stack (Memory)

```
0x130
0x128
0x120
0x118
```

Registers

```
%rip: 0x400544
%rsp: 0x120 0x118
```

Program counter

```
400549
```

Drawing #3

```
400550:
```

Drawing #2

```
400550:
```

Drawing #1

```
400544:
```
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
Procedure Return Example (step 2)

Procedure <mult2>

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

Procedure <mult2>

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

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)
- High Addresses
- Low Addresses
  - %rax
  - Arg 7
  - Arg 8
  - Arg n
  - • • •

- Only allocate stack space when needed
- Diane’s Silk Dress Costs $89

- 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

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;
}

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

Example Call Chain

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

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

Stack diagram:
- `main` frame
- `yoo` procedure
- `who` function
- `%rbp` register
- `%rsp` register

Diagram notes:
- Frame pointer
- Procedure call
- Return address
2) Call to who

```plaintext
yoo(…)
{
  who(…)
  {
    •
    amI();
    •
    amI();
    •
  }
}
```
3) Call to amI (1)
4) Recursive call to `amI (2)`
5) **(another) Recursive call to `amI` (3)**

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

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

```c
amI(…)
{
  if()
  amI()
}
```
6) Return from \((\text{another})\) recursive call to `amI`

```c
yoo(…)
{
  who(…)
  {
    amI(…)
    {
      amI(…)
      {
        
        if()
        {
          amI()
        }
      }
      
    }
  }
}
```
7) Return from recursive call to `amI`
8) Return from call to amI

```
8) Return from call to amI

Stack

yoo

who

%rbp
%rsp

new stack frame overwrites old data!

amI

amI

amI

amI

amI

amI

amI

amI

amI

amI

amI

amI`
9) (second) Call to amI (4)
10) Return from (second) call to **amI**
11) Return from call to `who`

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

Call chain: `main` → `yoo` → `who` → `amI` → `amI` → `amI` → `amI`

**Stack**

- `main`
- `yoo`
- `who`
- `amI_4`
- `amI_2`
- `amI_3`

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 be called, 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? \[8\]
- What is the maximum depth (# of frames) of the Stack?

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

A. 1  B. 2  C. 3  D. 4
Example: increment

```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

Incr. C and incr. s posted on website so you can step through this example in gdb
Procedure Call Example (initial state)

Return address on stack is the address of instruction immediately following the call to “call_incr”
- Shown here as main, but could be anything)
- Pushed onto stack by call call_incr
Procedure Call Example (step 1)

```
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

```
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
```
### Procedure Call Example (step 2)

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

**Stack Structure**

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

**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>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;
}
```

```assembly
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

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

- **State while inside increment**
  - **Return address** on top of stack is address of the `addq` instruction immediately following call to `increment`

<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>
<tr>
<td>%rax</td>
<td></td>
</tr>
</tbody>
</table>
Procedure Call Example (step 4)

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

```
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
```

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

Stack Structure

- State while inside `increment`
  - After code in body has been executed

```
<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>
```

- Read 351
- Add 100
- Store 451
Procedure Call Example (step 5)

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

<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 (v2)</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 6)

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

`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`

**Stack Structure**
- Return addr `<main+8>`
- `%rsp+8`
- `451` ← `%rsp`
- `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>451+351</td>
</tr>
</tbody>
</table>

Update `%rax` to contain `v1+v2`
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
- %rsp
- old %rsp

De-allocate space for local **vars**
(make sure %rsp points to return addr before ret)

<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>

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
Procedure Call Example (step 8)

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

- State just before returning from call to `call_incr`

Stack Structure:

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

Register Use(s):

<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)

**Final 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>451</td>
</tr>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>
Lab 2 Demo

- Let's look at that binary bomb!

```
objdump -d bomb > bomb_disas
// store disassembly of bomb in file
// called bomb_disas
```

In GDB:
- `stepi <#>`
  // execute the next `<#>` asm instr (stepping into function calls)
- `nexti <#>`
  // execute the next `<#>` asm instr (stepping over function calls)
- `print /<format> <expr>`
  // print value of `<expr>` in `<format>`
- `x /<format> <addr>`
  // dereference `<addr>` and print in `<format>`

Notes:
- Annoyingly, register names in `<expr>` and `<addr>` in GDB are preceded by `$`
- Use `$rsp` instead of `%rsp`
- Common format characters are:
  - `'b'` for binary
  - `'x'` for hex
  - `'s'` for string
Procedures

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

- When procedure `yoo` calls `who`:
  - `yoo` is the *caller*
  - `who` is the *callee*

- Can registers be used for temporary storage?

  ![Code Snippet]

  - No! Contents of register `%rdx` overwritten by `who`!
  - This could be trouble – something should be done. Either:
    - *Caller* should save `%rdx` before the call (and restore it after the call)
    - *Callee* should save `%rdx` before using it (and restore it before returning)
Register Saving Conventions

❖ “Caller-saved” registers
  ▪ It is the caller’s responsibility to save any important data in these registers before calling another procedure (i.e. the callee can freely change data in these registers)
  ▪ Caller saves values in its stack frame before calling Callee, then restores values after the call

❖ “Callee-saved” registers
  ▪ It is the callee’s responsibility to save any data in these registers before using the registers (i.e. the caller assumes the data will be the same across the callee procedure call)
  ▪ Callee saves values in its stack frame before using, then restores them before returning to caller
Silly Register Convention Analogy

1) Parents *(caller)* leave for the weekend and give the keys to the house to their child *(callee)*
   - Being suspicious, they put away/hid the valuables *(caller-saved)* before leaving
   - Warn child to leave the bedrooms untouched: “These rooms better look the same when we return!”

2) Child decides to throw a wild party *(computation)*, spanning the entire house
   - To avoid being disowned, child moves all of the stuff from the bedrooms to the backyard shed *(callee-saved)* before the guests trash the house
   - Child cleans up house after the party and moves stuff back to bedrooms

3) Parents return home and are satisfied with the state of the house
   - Move valuables back and continue with their lives
x86-64 Linux Register Usage, part 1

- **%rax**
  - Return value
  - Also **caller**-saved & restored
  - Can be modified by procedure

- **%rdi, ..., %r9**
  - Arguments
  - Also **caller**-saved & restored
  - Can be modified by procedure

- **%r10, %r11**
  - **Caller**-saved & restored
  - Can be modified by procedure
x86-64 Linux Register Usage, part 2

- `%rbx, %r12, %r13, %r14`
  - Callee-saved
  - Callee must save & restore

- `%rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match

- `%rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure
# x86-64 64-bit Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value - Caller saved</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4 - Caller saved</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3 - Caller saved</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2 - Caller saved</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1 - Caller saved</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>Argument #5 - Caller saved</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6 - Caller saved</td>
</tr>
<tr>
<td>%r10</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
Callee-Saved Example (step 1)

```c
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

**Initial Stack Structure**

```
...  
ret addr  
```

**Resulting Stack Structure**

```
...  
ret addr  
saved %rbx  
351  
<unused>  
```

```assembly
call_incr2:
    pushq %rbx  ; save old %rbx
    subq $16, %rsp
    movq %rdi, %rbx  ; change %rbx
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment  ; across procedure call
    addq %rbx, %rax
    addq $16, %rsp  
    popq %rbx
    ret
```
Callee-Saved Example (step 2)

```c
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

**Call**

1. pushq %rbx
2. subq $16, %rsp
3. movq %rdi, %rbx
4. movq $351, 8(%rsp)
5. movl $100, %esi
6. leaq 8(%rsp), %rdi
7. call increment
8. addq %rbx, %rax
9. addq $16, %rsp
10. popq %rbx
11. ret

**Pre-return Stack Structure**

- ret addr
- saved %rbx
- 351
- <unused>

**Memory**

- %rsp
- ...
Why Caller and Callee Saved?

- We want *one* calling convention to simply separate implementation details between caller and callee.

- In general, neither caller-save nor callee-save is “best”:
  - If caller isn’t using a register, caller-save is better
  - If callee doesn’t need a register, callee-save is better
  - If “do need to save”, callee-save generally makes smaller programs
    - Functions are called from multiple places

- So… “some of each” and compiler tries to “pick registers” that minimize amount of saving/restoring.
Register Conventions Summary

- **Caller**-saved register values need to be pushed onto the stack before making a procedure call *only if the Caller needs that value later*
  - **Callee** may change those register values

- **Callee**-saved register values need to be pushed onto the stack *only if the Callee intends to use those registers*
  - **Caller** expects unchanged values in those registers

- Don’t forget to restore/pop the values later!