The Stack & Procedures
CSE 351 Winter 2018

Instructor:
Mark Wyse

Teaching Assistants:
Kevin Bi
Parker DeWilde
Emily Furst
Sarah House
Waylon Huang
Vinny Palaniappan

http://xkcd.com/648/
Administrative

- Homework 2 (x86) due tonight
- Lab 2 due Friday (2/2)
- Homework 3 released today
  - On midterm material, but due after the midterm (2/9)

- Midterm (2/5, in-class)
  - Find a study group! Study practice problems and past exams
  - Must bring your UW Student ID to the exam!
  - Topics are Lectures 1 – 12, Ch 1.0 – 3.7
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:** $%rsp$
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` (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?
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 *(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)

00000000000400540 <multstore>:

•

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

•

•

00000000000400550 <mult2>:

400550: movq %rdi,%rax

•

•

400557: ret

%rip 0x400544

%rsp 0x120

0x120

0x128

0x130
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)

0x120 0x128 0x130

0x400549

0x118

%rip 0x400557

%rsp 0x118

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
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;
}
```

```assembly
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;
}
```

```assembly
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

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

Stack

```
main

yoo
```

• %rbp
• %rsp

```
yoo

who

amI

amI

amI
```

who();
2) Call to who

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

Stack:
- yoo
- who
- amI
- amI
- rbp
- rsp
3) Call to `amI(1)`

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

Stack

```
yoo
who
amI
amI
%rbp
%rsp
amI_1
```
4) Recursive call to `amI` (2)

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

Stack

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

 `%rbp` -> `%rsp`
5) (another) Recursive call to `amI (3)`

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

Stack
```

%rbp

%rsp
6) Return from (another) recursive call to amI

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

Stack

```
yoo
   who
   amI
```

```
%rbp
%rsp
amI_1
amI_2
amI_3
```
7) Return from recursive call to `amI`
8) Return from call to `amI`
9) (second) Call to amI (4)
10) Return from (second) call to amI
11) Return from call to `who`

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

Stack Diagram:
- `%rbp` points to `yoo`
- `%rsp` points to `who`
- `amI` pointers within the stack frame
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)
  - 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)
Example: increment

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st arg (p)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd arg (val), y</td>
</tr>
<tr>
<td>%rax</td>
<td>x, return value</td>
</tr>
</tbody>
</table>
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 instruction immediately following the call to "call_incr"
  - Shown here as `main`, but could be anything)
  - Pushed onto stack by `call call_incr`

Initial Stack Structure

```
   ↓ %rsp
Return addr <main+8>
```

- Return address on stack is the address of instruction immediately following the call to "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

**Stack Structure**

- Return addr `<main+8>`
- `351` ← `%rsp+8`
- `Unused` ← `%rsp`

**Allocate space for local vars**

```
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>...</th>
<th>Return addr &lt;main+8&gt;</th>
<th>351</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>§rsp+8</td>
<td>§rsp+8</td>
<td></td>
<td></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`

### 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;
}
```

Stack Structure

- Return addr <main+8>
  - 351
  - Unused
- Return addr <call_incr+?> ← %rsp

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

```
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
```
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
- Return addr `<call_incr+?>` ← %rsp

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

**Increment:**

```asm
movq (%rdi), %rax  # x = *p
addq %rax, %rsi   # y = x+100
movq %rsi, (%rdi) # *p = y
ret
```

**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>
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

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

```c
long call_incr() {
    long v1 = 351;
    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>451</td>
</tr>
<tr>
<td>%rax</td>
<td>451+351</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`

- **Update %rax to contain v1+v2**

- **Return addr <main+8>**
- `%rsp+8`
Procedure Call Example (step 7)

```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>
- 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>
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
```

---

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

---

<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>
Procedure Call Example (step 9)

```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`

**Final Stack Structure**

- 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)

<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>