Roadmap

C:

```c
    car *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
110000011111101000001111
```

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

Computer system:

Spring 2014
Procedures and Call Stacks

- 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 these questions, we need a call stack ...


Memory Layout

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

2^{N-1}
Memory Layout

- **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?
IA32 Call Stack

- Region of memory managed with a stack “discipline”
- Grows toward lower addresses
- Customarily shown “upside-down”

- Register %esp contains lowest stack address
  = address of “top” element

Stack Pointer: %esp

Stack “Bottom”

Increasing Addresses

Stack Grows Down

Stack “Top”
IA32 Call Stack: Push

- `pushl Src`

Stack Pointer: `%esp`

Stack "Top"

Stack "Bottom"

Increasing Addresses

Stack Grows Down
IA32 Call Stack: Push

- **pushl Src**
  - Fetch value from *Src*
  - Decrement `%esp` by 4 (*why 4?*)
  - Store value at address given by `%esp`

Stack Pointer: `%esp`

Stack "Bottom"

Stack "Top"

Increasing Addresses

Stack Grows Down

Stack Grows Down
IA32 Call Stack: Pop

- `popl Dest`

.stack Pointer: `%esp`

Stack "Bottom"

Increasing Addresses

Stack Grows Down

Stack "Top"
IA32 Call Stack: Pop

- **popl Dest**
  - Load value from address %esp
  - Write value to Dest
  - Increment %esp by 4

Stack Pointer: %esp

Stack “Top”

Stack “Bottom”

Increasing Addresses

Stack Grows Down

Spring 2014
IA32 Call Stack: Pop

- `popl Dest`
  - Load value from address `%esp`
  - Write value to `Dest`
  - Increment `%esp` by 4

Those bits are still there; we’re just not using them.
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 val
- **Caller** and **Callee** run on same CPU → use the same registers
  - So how do we deal with register reuse?

```
...<set up args>
call
<clean up args>
<find return val>
...
```

```
<create local vars>
...
<set up return val>
<destroy local vars>
return
```
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 IA32/Linux in detail
- What could happen if our program didn’t follow these conventions?
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call**: `call label`
  - Push return address on stack (*why?, and which exact address?*)
  - Jump to `label`
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call**: `call label`
  - Push return address on stack
  - Jump to `label`
- **Return address**:
  - Address of instruction after `call`
  - Example from disassembly:

```
804854e: e8 3d 06 00 00     call 8048b90 <main>
8048553: 03 45 08          addl 0x08(%ebp),%eax
```

Return address = 0x8048553

- **Procedure return**: `ret`
  - Pop return address from stack
  - Jump to address
  
  next instruction just happens to be an add, but could be anything
Procedure Call Example

804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 03 45 08 addl 0x08(%ebp),%eax

call 8048b90

%esp 0x108
%eip 0x804854e

%eip: program counter
Procedure Call Example

```
804854e:    e8 3d 06 00 00    call    8048b90 <main>
8048553:    03 45 08    addl    0x08(%ebp),%eax
```

**Procedure Call Example Diagram**

```
call 8048b90

0x110 0x10c 0x108 123
%esp 0x108
%eip 0x804854e

call 8048b90

0x110 0x10c 0x108 123
%esp 0x104
%eip 0x8048553
```

**%eip**: program counter
Procedure Call Example

```
804854e:   e8 3d 06 00 00  call  8048b90 <main>
8048553:   03 45 08  addl  0x08(%ebp),%eax
```

relative address just like jumps...
(chosen by compiler; there’s also an absolute call)

%esp: program counter

%esp  0x108
%eip  0x804854e
%esp  0x104
%eip  0x8048553

0x108   123
0x10c   123
0x110   0
0x104   0x8048553
0x10c   0
0x110   0

call  804854e
addl  0x08(%ebp),%eax
```
Procedure Return Example

8048591: c3 ret

ret

0x110
0x10c
0x108 123
0x104 0x8048553

%esp 0x104
%eip 0x8048591

%eip: program counter
Procedure Return Example

8048591: c3 ret

%esp 0x104
%eip 0x8048591

0x104 0x108 0x10c
0x8048553
123

%esp 0x104
%eip 0x8048591

0x110 0x110
0x10c 0x10c
0x108 0x108
0x8048553 0x8048553
123

%esp 0x104
%eip 0x8048591

ret
Procedure Return Example

8048591: c3 ret

%esp 0x104
%eip 0x8048591

0x104 0x8048553
0x108 123
0x10c
0x110

%esp 0x104
%eip 0x8048591

0x104 0x8048553
0x108 123
0x10c
0x110

%esp 0x104
%eip 0x8048591

ret

%eip: program counter
Procedure Return Example

8048591: c3 ret

%esp %eip
0x104 0x8048553
0x108 123
0x10c 0x110
0x104 0x8048553
0x110 0x10c
%esp %esp
0x104 0x108
%eip %eip
0x8048591 0x8048553
ret ret
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 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

- **Stack allocated in frames**
  - State for a single procedure instantiation
Call Chain Example

Procedure `amI` is recursive (calls itself)
Stack Frames

Contents
- Local variables
- Function arguments
- Return information
- Temporary space

Management
- Space allocated when procedure is entered
  - “Set-up” code
- Space deallocated upon return
  - “Finish” code
Example

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

Stack

```
%ebp

%esp

yoo
```

Procedures and Stacks
Example

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

Stack

```
%ebp
%esp
yoo
who
```

Procedure and Stacks
Example

```c
amI(...) {
  .
  .
  amI();
  .
  .
}
```

Stack

```
%ebp
%esp
yoo
who
amI
amI
amI
```

Procedures and Stacks
Example

```c
amI(...) {
  .
  .
  amI();
  .
  .
}
```

Stack

```plaintext
  %ebp
  %esp
  amI
  who
  yoo
  amI
  amI
  amI
  amI
```
Example

```c
amI(...) {
    ·
    ·
    amI();
    ·
    ·
}
```

Stack

```
%ebp
%esp
amI
amI
amI
yoo
who
```

Spring 2014
Procedures and Stacks
Example

```
void amI(...) {
    ...
    amI();
    ...
}
```

```
Stack

yoo
who
amI
amI
amI
%ebp
%esp
```
Example

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

Stack

```
%ebp

%esp
```

```c
yoo

who

amI

amI
```

Spring 2014

Procedures and Stacks
Example

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

Stack

```
Stack:

  who
  %esp
  %ebp

  yoo
```
Example

```
ami(...) {
    ...
    ...
    ...
}
```

Stack

```
%ebp
%esp

yoo
who
ami
ami
ami
ami
```
Example

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

Stack

yoo

%ebp

%esp

amI

amI

amI

amI

who

Example

yoo

%ebp

%esp

amI

amI

amI

amI

who

Stack
Example

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

How did we remember where to point `%ebp` when returning?
IA32/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build" area
    (parameters for function about to be called)
  - Local variables
    (if can’t be kept in registers)
  - Saved register context
    (when reusing registers)
  - Old frame pointer (for caller)

- **Caller’s Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call
Revisiting swap

```c
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
    swap(&zip1, &zip2);
}

void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```
Revisiting swap

```c
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
    swap(&zip1, &zip2);
}
```

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Calling swap from call_swap

```c
call_swap:
    ... ...
    pushl $zip2   # Global Var
    pushl $zip1   # Global Var
    call swap
    ... ...
```

we know the address
**Revisiting swap**

```c
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
    swap(&zip1, &zip2);
}

void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

**Calling swap from call_swap**

```c
call_swap:
    pushl $zip2     # Global Var
    pushl $zip1     # Global Var
    call swap
    ...
```

**Resulting Stack**

```
      ...
      &zip2
      &zip1
      Rtn adr
      %esp
```
Revisiting `swap`

```c
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Body

Set Up

Finish
swap Setup #1

Entering Stack

\[\begin{array}{l}
\%ebp \\
\hline
\&\text{zip2} \\
\&\text{zipl} \\
\text{Rtn adr} \\
\%esp
\end{array}\]

Resulting Stack?

\[\text{Set Up}\]

\text{swap:}\]
\begin{align*}
&\text{pushl } \%ebp \\
&\text{movl } \%esp,\%ebp \\
&\text{pushl } \%ebx
\end{align*}
swap Setup #1

**Entering Stack**

- `%ebp`
- `%esp`
- `&zip2`
- `&zip1`
- `Rtn adr`

**Resulting Stack**

- `%ebp`
- `%esp`
- `YP`
- `xp`
- `Rtn adr`
- `Old %ebp`

---

**swap:**

- `pushl %ebp`  
- `movl %esp,%ebp`  
- `pushl %ebx`

**Set Up**
swap Setup #2

Entering Stack

Resulting Stack

swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx

Set Up

ResulJng Stack

43

Procedures and Stacks
swap Setup #3

Entering Stack

Resulting Stack

swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx

Set Up
swap Body

Entering Stack

Resulting Stack

Offset relative to new %ebp

movl 12(%ebp),%ecx  # get yp
movl 8(%ebp),%edx  # get xp

Body
swap Finish #1

Finishing Stack

Resulting Stack?

```asm
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Spring 2014

Procedures and Stacks
swap Finish #1

Finishing Stack

Resulting Stack

Observation: Saved and restored register %ebx
swap Finish #2

Finishing Stack

Resulting Stack

movl  -4(%ebp),%ebx
movl  %ebp,%esp
popl  %ebp
ret

Finish
swap Finish #3

Finishing Stack

```
old %ebp
old %ebx
Rtn adr
xp
YP
```

Resulting Stack

```
%esp
Rtn adr
xp
YP
```

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Finish
swap Finish #4

Finishing Stack

Resulting Stack

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret

Finish
Disassembled swap

080483a4 <swap>:

```
080483a4:   55          push   %ebp
080483a5:   89 e5      mov   %esp,%ebp
080483a7:   53          push   %ebx
080483a8:   8b 55 08   mov   0x8(%ebp),%edx
080483ab:   8b 4d 0c   mov   0xc(%ebp),%ecx
080483ae:   8b 1a      mov   (%edx),%ebx
080483b0:   8b 01      mov   (%ecx),%eax
080483b2:   89 02      mov   %eax,(%edx)
080483b4:   89 19      mov   %ebx,(%ecx)
080483b6:   5b          pop    %ebx
080483b7:   c9          leave
080483b8:   c3          ret
```

Calling Code

```
8048409:   e8 96 ff ff ff   call 80483a4 <swap>
804840e:   8b 45 f8      mov   0xffffffff8(%ebp),%eax
```

relative address (little endian)
swap Finish #4

Finishing Stack

\[
\begin{align*}
\text{Old} & \% \text{ebp} \\
\text{Old} & \% \text{ebx} \\
\text{Rtn adr} & \\
xp & \\
\text{YP} & \\
\end{align*}
\]

\[
\begin{align*}
\text{movl} & \ -4(\% \text{ebp}), \% \text{ebx} \\
\text{movl} & \ % \text{ebp}, % \text{esp} \\
\text{popl} & \ % \text{ebp} \\
\text{ret} & \\
\end{align*}
\]

Resulting Stack

\[
\begin{align*}
\text{\%esp} & \\
xp & \\
\text{YP} & \\
\end{align*}
\]

\[
\begin{align*}
\text{\%ebp} & \\
\end{align*}
\]

- Observation
  - Saved & restored register \( %\text{ebx} \)
  - but not \( %\text{eax}, %\text{ecx}, \text{or}\ %\text{edx} \)
Register Saving Conventions

■ When procedure yoo calls who:
  - yoo is the caller
  - who is the callee

■ Can a register be used for temporary storage?

```assembly
yoo:
  . . .
  movl $12345, %edx
  call who
  addl %edx, %eax
  . . .
  ret

who:
  . . .
  movl 8(%ebp), %edx
  addl $98195, %edx
  . . .
  ret
```

- Contents of register %edx overwritten by who
Register Saving Conventions

■ When procedure $yoo$ calls $who$:
  - $yoo$ is the $\text{caller}$
  - $who$ is the $\text{callee}$

■ Can a register be used for temporary storage?

■ Conventions
  - “$\text{Caller Save}$”
    - Caller saves temporary values in its frame before calling
  - “$\text{Callee Save}$”
    - Callee saves temporary values in its frame before using
IA32/Linux Register Usage

- **%eax, %edx, %ecx**
  - Caller saves prior to call if values are used later

- **%eax**
  - also used to return integer value

- **%ebx, %esi, %edi**
  - Callee saves if wants to use them

- **%esp, %ebp**
  - special form of callee save – restored to original values upon exit from procedure
Example: Pointers to Local Variables

Top-Level Call

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

```c
int sfact(3)
```

Recursive Procedure

```c
void s_helper
    (int x, int *accum)
{
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper(x-1,accum);
    }
}
```

Pass pointer to update location
Example: Pointers to Local Variables

Top-Level Call

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

```c
int sfact(3)
```

```c
s_helper(3, &val)
```

Recursive Procedure

```c
void s_helper
    (int x, int *accum)
{
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper (x-1,accum);
    }
}
```

Pass pointer to update location

Spring 2014 Procedures and Stacks
Example: Pointers to Local Variables

Top-Level Call

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

sfact(3) val = 1
s_helper(3, &val) val = 3
s_helper(2, &val)

Recursive Procedure

```c
void s_helper
    (int x, int *accum)
{
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper (x-1,accum);
    }
}
```

Pass pointer to update location
Example: Pointers to Local Variables

Top-Level Call

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

Recursive Procedure

```c
void s_helper
    (int x, int *accum)
{
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper (x-1,accum);
    }
}
```

sfact(3) val = 1
s_helper(3, &val) val = 3
s_helper(2, &val) val = 6
s_helper(1, &val).

Pass pointer to update location
Creating & Initializing Pointer

```
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- Variable `val` must be stored on stack
  - Because: Need to create pointer to it
- Compute pointer as \(-4\) (%ebp)
- Push on stack as second argument

**Initial part of sfact**

```
_sfact:
    pushl %ebp  # Save %ebp
    movl %esp,%ebp  # Set %ebp
    subl $16,%esp  # Add 16 bytes
    movl 8(%ebp),%edx  # edx = x
    movl $1,-4(%ebp)  # val = 1
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td>val = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-12</td>
<td>Unused</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

%esp: |%ebp  |
%esp  |%esp  |
%esp  |%esp  |
%esp  |%esp  |
%esp  |%esp  |
%esp  |%esp  |
%esp  |%esp  |
%esp  |%esp  |
%esp  |%esp  |
%esp  |%esp  |
%esp  |%esp  |
Passing Pointer

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- **Variable val must be stored on stack**
  - Because: Need to create pointer to it
- **Compute pointer as \(-4 (\%ebp)\)**
- **Push on stack as second argument**

**Calling s_helper from sfact**

- `leal -4(\%ebp),%eax` # Compute &val
- `pushl %eax` # Push on stack
- `pushl %edx` # Push x
- `call s_helper` # call
- `movl -4(\%ebp),%eax` # Return val
  - `• • •` # Finish

**Stack at time of call:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(8)</td>
<td>(x)</td>
</tr>
<tr>
<td>(4)</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>(0)</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>(-4)</td>
<td>val=x!</td>
</tr>
<tr>
<td>(-8)</td>
<td>Unused</td>
</tr>
<tr>
<td>(-12)</td>
<td>&amp;val</td>
</tr>
<tr>
<td>(-16)</td>
<td>(x)</td>
</tr>
</tbody>
</table>

- \%ebp
- \%esp
IA 32 Procedure Summary

- **Important points:**
  - IA32 procedures are a combination of *instructions* and *conventions*
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P

- **Recursion handled by normal calling conventions**
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result returned in $%eax$

![Diagram of caller frame with arguments, saved registers, and local variables.]

**Spring 2014**  Procedures and Stacks
x86-64 Procedure Calling Convention

- Doubling of registers makes us less dependent on stack
  - Store argument in registers
  - Store temporary variables in registers

- What do we do if we have too many arguments or too many temporary variables?
## x86-64 64-bit Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</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>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</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>
Revisiting swap, IA32 vs. x86-64 versions

swap:

pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret

swap (64-bit long ints):

movq (%rdi),%rdx
movq (%rsi),%rax
movq %rax,(%rdi)
movq %rdx,(%rsi)
ret

- Arguments passed in registers
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers

- No stack operations required (except ret)

- Avoiding stack
  - Can hold all local information in registers
X86-64 procedure call highlights

- **Arguments (up to first 6) in registers**
  - Faster to get these values from registers than from stack in memory

- **Local variables also in registers (if there is room)**

- **Registers still designated “caller-saved” or “callee-saved”**

- **callq instruction stores 64-bit return address on stack**
  - Address pushed onto stack, decrementing %rsp by 8

- **No frame pointer**
  - All references to stack frame made relative to %rsp; eliminates need to update %ebp/%rbp, which is now available for general-purpose use

- **Functions can access memory up to 128 bytes beyond %rsp: the “red zone”**
  - Can store some temps on stack without altering %rsp
x86-64 Memory Layout

- **Instructions**
- **Literals**
- **Static Data**
- **Dynamic Data (Heap)**
- **Stack**

- **128-byte red zone**
  - space lower than the stack pointer that procedures can use for data not needed across procedure calls

- Optimization to avoid extra `%rsp` updates

---

---
x86-64 Stack Frames

- Often (ideally), x86-64 functions need no stack frame at all
  - Just a return address is pushed onto the stack when a function call is made

- A function *does* need a stack frame when it:
  - Has too many local variables to hold in registers
  - Has local variables that are arrays or structs
  - Uses the address-of operator (&) to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Needs to save the state of caller-save registers before calling a procedure
  - Needs to save the state of callee-save registers before modifying them
Example

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call_proc:
subq $32,%rsp
movq $1,16(%rsp)
movl $2,24(%rsp)
movw $3,28(%rsp)
movb $4,31(%rsp)
•••

Return address to caller of call_proc

NB: Details may vary depending on compiler.
Example

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2,
         x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call_proc:
```
subq $32,%rsp
movq $1,16(%rsp)
movl $2,24(%rsp)
movw $3,28(%rsp)
movb $4,31(%rsp)
• • •
```

Return address to caller of call_proc

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
<th>x1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>%rsp</td>
</tr>
</tbody>
</table>

Spring 2014
Procedures and Stacks
Example

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call_proc:

```assembly
leaq 24(%rsp),%rcx
leaq 16(%rsp),%rsi
leaq 31(%rsp),%rax
movq %rax,8(%rsp)
movl $4,(%rsp)
leaq 28(%rsp),%r9
movl $3,%r8d
movl $2,%edx
movq $1,%rdi
call proc
```

Return address to caller of call_proc

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arg 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arg 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Arguments passed in (in order): rdi, rsi, rdx, rcx, r8, r9

%rsp
Example

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2,
         x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call_proc:
```assembly
gleaq 24(%rsp),%rcx
leaq 16(%rsp),%rsi
leaq 31(%rsp),%rax
movq %rax,8(%rsp)
movl $4,(%rsp)
gleaq 28(%rsp),%r9
movl $3,%r8d
movl $2,%edx
movq $1,%rdi
call proc
```

Arguments passed in (in order): rdi, rsi, rdx, rcx, r8, r9

<table>
<thead>
<tr>
<th>Return address to caller of call_proc</th>
<th>Arg 8</th>
<th>Arg 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>x4</td>
<td>x3</td>
<td>x2</td>
</tr>
<tr>
<td>x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2,
         x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call_proc:
  • • •
  leaq 24(%rsp),%rcx
  leaq 16(%rsp),%rsi
  leaq 31(%rsp),%rax
  movq %rax,8(%rsp)
  movl $4,(%rsp)
  leaq 28(%rsp),%r9
  movl $3,%r8d
  movl $2,%edx
  movq $1,%rdi
  call proc
  • • •

Return address to caller of call_proc
<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>x1</td>
</tr>
<tr>
<td></td>
<td>Arg 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arg 7</td>
<td></td>
</tr>
</tbody>
</table>

Return address to line after call to proc
%rsp

Arg 8
Arg 7
Example

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2,
         x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call_proc:

- movswl 28(%rsp),%eax
- movsbl 31(%rsp),%edx
- subl %edx,%eax
- cltq
- movslq 24(%rsp),%rdx
- addq 16(%rsp),%rdx
- imulq %rdx,%rax
- addq $32,%rsp
- ret

Return address to caller of call_proc

<table>
<thead>
<tr>
<th></th>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arg 8</td>
<td>%rsp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arg 7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2,
         x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call_proc:

```
    movswl 28(%rsp),%eax
    movsbl 31(%rsp),%edx
    subl   %edx,%eax
    cltq
    movslq 24(%rsp),%rdx
    addq   16(%rsp),%rdx
    imulq  %rdx,%rax
    addq   $32,%rsp
    ret
```

Return address to caller of call_proc
x86-64 Procedure Summary

- Heavy use of registers (faster than using stack in memory)
  - Parameter passing
  - More temporaries since more registers

- Minimal use of stack
  - Sometimes none
  - When needed, allocate/deallocate entire frame at once
  - No more frame pointer: address relative to stack pointer

- More room for compiler optimizations
  - Prefer to store data in registers rather than memory
  - Minimize modifications to stack pointer