Computer Systems
CSE 410 Spring 2012
7 – Procedures, parameters, and the call stack (rev. with x86-64 details)
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?

To answer these questions, we need a *call stack* ...
Memory Layout

- **Stack**
  - Local variables

- **Dynamic Data (Heap)**
  - new'ed variables

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

- **Literals**
  - literals (e.g., “example”)

- **Instructions**
Memory Layout

- **Stack**
  - “Automatic” lifetime; mutable
  - Writable; not executable

- **Dynamic Data (Heap)**
  - Programmer controlled lifetime; mutable
  - Writable; not executable

- **Static Data**
  - Execution lifetime; mutable
  - Writable; not executable

- **Literals**
  - Execution lifetime; immutable
  - Read-only; not executable

- **Instructions**
  - Execution lifetime; immutable
  - Read-only; executable
IA32 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 Stack: Push

- `pushl Src`

Stack Pointer: `%esp`

Stack "Bottom"

Stack Grows Down

Increasing Addresses

Stack "Top"
IA32 Stack: Push

- `pushl Src`
  - Fetch operand at `Src`
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`
IA32 Stack: Pop

- popl Dest

Stack Pointer: %esp

Increasing Addresses

Stack Grows Down

Stack “Bottom”

Stack “Top”
IA32 Stack: Pop

- `popl Dest`
  - Read operand at address `%esp`
  - Increment `%esp` by 4
  - Write operand to `Dest`
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
  - Might need to save registers used by **Callee**
Procedure Call Overview

The convention of where to leave/find things is called the procedure call linkage

- Details vary between systems
- We will see the convention for IA32/Linux in detail
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call label`
  - Push return address on stack
  - 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 beyond `call`
  - Example from disassembly

  ```
  804854e:   e8 3d 06 00 00   call  8048b90 <main>
  8048553:   50                pushl  %eax
  ```
  - Return address = 0x8048553

- **Procedure return:** `ret`
  - Pop address from stack
  - Jump to address
**Procedure Call Example**

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>804854e</td>
<td><code>e8 3d 06 00 00</code></td>
<td><code>call 8048b90 &lt;main&gt;</code></td>
</tr>
<tr>
<td>8048553</td>
<td><code>50</code></td>
<td><code>pushl %eax</code></td>
</tr>
</tbody>
</table>

```
804854e: e8 3d 06 00 00  call  8048b90 <main>
8048553: 50       pushl %eax
```

```
call  8048b90
```

- `%esp`: Top of stack
- `%eip`: Program counter

---

18 April 2012

Procedures and Stacks
Procedure Call Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>804854e:</td>
<td>e8 3d 06 00 00</td>
<td>call 8048b90 &lt;main&gt;</td>
</tr>
<tr>
<td>8048553:</td>
<td>50</td>
<td>pushl %eax</td>
</tr>
</tbody>
</table>

- %esp
- %eip
- Program Counter

Program counter:

- 0x110
- 0x10c
- 0x108
- 0x104
- 0x108

Stack:

- 123
Procedure Call Example

```
804854e:  e8 3d 06 00 00    call  8048b90 <main>
8048553:  50            pushl  %eax
```

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

0x104
%eip
0x100 123
%esp
```

`%eip`: program counter
Procedure Call Example

804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50 pushl %eax

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

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

call 8048b90

%eip: program counter
Procedure Call Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>804854e</td>
<td>e8 3d 06 00 00</td>
<td>call 8048b90 &lt;main&gt;</td>
</tr>
<tr>
<td>8048553</td>
<td>50</td>
<td>pushl %eax</td>
</tr>
</tbody>
</table>

**%esp** 0x108  
**%eip** 0x804854e

0x108 123  
0x10c 0x8048553  
0x110

**%esp** 0x104

%eip: program counter

0x804854e + 0x000063d

0x8048b90
Procedure Return Example

<table>
<thead>
<tr>
<th>%esp</th>
<th>%eip</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x104</td>
<td>0x8048591</td>
</tr>
<tr>
<td>0x108</td>
<td>0x8048553</td>
</tr>
<tr>
<td>0x10c</td>
<td>123</td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
</tbody>
</table>

%eip: program counter

%esp: %eip
Procedure Return Example

8048591:  c3  ret

%esp  0x104  %eip  0x8048591
0x108  123
0x10c
0x110
0x110  123
0x104  0x8048553
0x10c
0x108
0x110  0x8048553
0x104
0x10c
0x108
0x110
%esp  0x104  %esp  0x104
%eip  0x8048591  %eip  0x8048591
%eip:  program counter

8048591:  c3  ret

%esp  0x104  %eip  0x8048591
0x108  123
0x10c
0x110
0x110  123
0x104  0x8048553
0x10c
0x108
0x110  0x8048553
0x104
0x10c
0x108
0x110
%esp  0x104  %esp  0x104
%eip  0x8048591  %eip  0x8048591
%eip:  program counter

8048591:  c3  ret

%esp  0x104  %eip  0x8048591
0x108  123
0x10c
0x110
0x110  123
0x104  0x8048553
0x10c
0x108
0x110  0x8048553
0x104
0x10c
0x108
0x110
%esp  0x104  %esp  0x104
%eip  0x8048591  %eip  0x8048591
%eip:  program counter
# Procedure Return Example

<table>
<thead>
<tr>
<th>Value</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x104</td>
<td>%esp</td>
</tr>
<tr>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
</tbody>
</table>

The value at `%esp` is 0x104, and the value at `%eip` is 0x8048591.

<table>
<thead>
<tr>
<th>Value</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
</tbody>
</table>

The value at `%esp` is 0x104, and the value at `%eip` is 0x8048591.

The code at 0x8048591 is:
```
c3
```

The value at `%esp` is 0x104, and the value at `%eip` is 0x8048591.

The return instruction is located at 0x8048553.

---

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

8048591:  c3  ret

ret

%esp  0x104
%eip  0x8048591

%esp  0x108
%eip  0x8048591

%esp  0x10c
%eip  0x8048553

%esp  0x110
%eip  0x8048553

%esp  0x108
%eip  0x8048553

%esp  0x10c
%eip  0x8048553

%esp  0x110
%eip  0x8048553

%eip: program counter
Stack-Based Languages

- **Languages that support recursion**
  - e.g., C, Pascal, Java
  - Code must be *re-entrant*
    - Multiple simultaneous instantiations of single procedure
      - What would happen if code could not be reentrant?
  - 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
  - Return information
  - Temporary space

- **Management?**
Stack Frames

Contents
- Local variables
- Return information
- Temporary space

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

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

Stack

```
%ebp
%esp
```

Procedures and Stacks
Example

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

Stack

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

`amI (...)`

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

Stack

```plaintext
%ebp
%esp

yoo
who
amI
```


Example

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

Stack

```
%ebp
%esp
```
Example

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

Stack

```
%ebp
%esp
```

Procedures and Stacks
Example

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

Stack

```
%ebp
%esp

yoo
who
amI
amI
amI
```
Example

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

Stack

```
%ebp
|
|-- yoo
|
|-- who
|
|-- amI
|
|-- amI
|
|-- amI
```

who
Example

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

Stack

```
Stack

who

yoo

%ebp

%esp
```
Example

```
amI(...) {
    ...amI
    ...amI
    ...amI
}
```
Example

who (...) 
{ 
  • • • 
  ami(); 
  • • • 
  ami(); 
  • • • 
}

Stack

yoo

who

ami  ami

ami

ami

Stack

yoo

who

%ebp

%esp
Example

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

Stack

```
%ebp
%esp
```

```
yoo
who
  amI  amI
    amI
      amI
```
### IA32/Linux Stack Frame

#### Current Stack Frame (“Top” to Bottom)
- Old frame pointer
- Local variables
  - If can’t be just kept in registers
- Saved register context
  - When reusing registers
- “Argument build area”
  - Parameters for function about to be called

#### Caller 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);
}

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
    ...
```
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

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

Resulting Stack

Rtn adr  %esp
&zip1
&zip2
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
```
swap Setup #1

Entering Stack

Resulting Stack?

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

Entering Stack

Resulting Stack

```
swap:
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
swap Setup #1

Entering Stack:

Resulting Stack:

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

Entering Stack

\[
\begin{align*}
&\text{\&zip2} \\
&\text{\&zip1} \\
&Rtn\adr
\end{align*}
\]

\[
\begin{align*}
%ebp &\leftarrow \text{\&zip2} \\
%esp &\leftarrow \text{\&zip1} \\
%esp &\leftarrow Rtn\adr
\end{align*}
\]

Resulting Stack

\[
\begin{align*}
&\text{\&zip2} \\
&\text{\&zip1} \\
&Rtn\adr \\
&\text{Old }%ebp \\
&\text{Old }%ebx
\end{align*}
\]

\[
\begin{align*}
%ebp &\leftarrow \text{\&zip2} \\
%esp &\leftarrow \text{\&zip1} \\
%esp &\leftarrow Rtn\adr \\
%esp &\leftarrow \text{Old }%ebp \\
%esp &\leftarrow \text{Old }%ebx
\end{align*}
\]

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

Entering Stack

Resulting Stack

\[
\begin{align*}
\text{movl } 12(\%ebp),\%ecx & \ # \text{ get } yp \\
\text{movl } 8(\%ebp),\%edx & \ # \text{ get } xp \\
\ldots
\end{align*}
\]
swap Finish #1

swap’s Stack

Resulting Stack?

```
movl  -4(%ebp),%ebx
movl  %ebp,%esp
popl  %ebp
ret
```
**swap Finish #1**

**swap’s Stack**

```
• • •
yp
xp
Rtn adr
Old %ebp
Old %ebx
```

**Resulting Stack**

```
• • •
yp
xp
Rtn adr
Old %ebp
Old %ebx
```

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

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

swap’s Stack

Resulting Stack

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
swap Finish #3

**swap’s Stack**

- 
- 
- 
- \(\text{yp}\)
- \(\text{xp}\)
- \(\text{Rtn adr}\)
- \(\text{Old } \%\text{ebp}\)
- \(\text{Old } \%\text{ebx}\)

**Resulting Stack**

- 
- 
- 
- \(\%\text{ebp}\)
- \(\%\text{esp}\)
- \(\text{yp}\)
- \(\text{xp}\)
- \(\text{Rtn adr}\)

```
movl -4(\%ebp),\%ebx
movl \%ebp,\%esp
popl \%ebp
ret
```
swap Finish #4

Observation
- Saved & restored register %ebx
- Didn’t do so for %eax, %ecx, or %edx

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
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:  8b 02       mov    %eax,(%edx)
080483b4:  8b 19       mov    %ebx,(%ecx)
080483b6:  5b          pop    %ebx
080483b7:  c9          leave
080483b8:  c3          ret

Calling Code

08048409:  e8 96 ff ff ff ff    call 080483a4 <swap>
0804840e:  8b 45 f8        mov    0xfffffffff8(%ebp),%eax

0x0804840e + 0xffffffff96 = 0x080483a4
swap Finish #4

swap’ s Stack

Resulting Stack

- Observation
  - Saved & restored register %ebx
  - Didn’t do so for %eax, %ecx, or %edx

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

- When should you save them?

- When should you not save them?
  - Why not save all of them?
Register Saving Conventions

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

- Can register be used for temporary storage?

- Conventions
  - *“Caller Save”*
    - Caller saves temporary in its frame before calling
  - *“Callee Save”*
    - Callee saves temporary in its frame before using

- Why do we have these conventions?
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
Recursive Factorial

```c
int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}
```

rfact:
```
pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 8(%ebp),%ebx
cmpl $1,%ebx
jle .L78
leal -1(%ebx),%eax
pushl %eax
call rfact
imull %ebx,%eax
jmp .L79
.align 4

.L78:
    movl $1,%eax

.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```
Recursive Factorial

```c
int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x - 1);
    return rval * x;
}
```

**Registers**

- `%ebx` used, but saved at beginning & restored at end
- `%eax` used without first saving
  - expect caller to save
  - pushed onto stack as parameter for next call
  - used for return value

**rfact:**

```assembly
pushl  %ebp
movl  %esp,%ebp
pushl  %ebx
movl  8(%ebp),%ebx
cmpl  $1,%ebx
jle  .L78
leal  -1(%ebx),%eax
pushl  %eax
call  rfact
imull  %ebx,%eax
jmp   .L79
.align  4
.L78:
movl  $1,%eax
.L79:
movl  -4(%ebp),%ebx
movl  %ebp,%esp
popl  %ebp
ret
```
void s_helper (int x, int *accum) {
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper (x-1, accum);
    }
}

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

- Pass pointer to update location
Creating & Initializing 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 $-4 \cdot \%ebp$
- 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
```

```
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
| 8 | 4 | 0 | -4 | -8 | -12 | -16 |
+---+---+---+---+---+---+---+---+
|   |   |   | Rtn adr | Old %ebp | val = 1 | Unused |
+---+---+---+---------+----------+----------+--------+
| @%esp | %ebp | %ebp | %ebp | %ebp | %ebp |
+---+---+---+---+---+---+---+
```
Passing Pointer

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

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
```
IA 32 Procedure Summary

- Stack makes recursion work
  - Private storage for each *instance* of procedure call
    - Instantiations don’t clobber each other
    - Addressing of locals + arguments can be relative to stack positions
  - Managed by stack discipline
    - Procedures return in inverse order of calls

- IA32 procedures
  *Combination of Instructions + Conventions*
  - call / ret instructions
  - Register usage conventions
    - caller / callee save
    - `%ebp` and `%esp`
  - Stack frame organization conventions
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>Description</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>Description</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, again

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

```assembly
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
```

- **Set Up**
- **Body**
- **Finish**
Revisiting swap, IA32 vs. x86-64 versions

swap:
\[
\begin{align*}
\text{pushl } & \%ebp \\
\text{movl } & \%esp, \%ebp \\
\text{pushl } & \%ebx \\
\text{movl } & 12(\%ebp), \%ecx \\
\text{movl } & 8(\%ebp), \%edx \\
\text{movl } & (\%ecx), \%eax \\
\text{movl } & (\%edx), \%ebx \\
\text{movl } & \%eax, (\%edx) \\
\text{movl } & \%ebx, (\%ecx) \\
\text{movl } & -4(\%ebp), \%ebx \\
\text{movl } & \%ebp, \%esp \\
\text{popl } & \%ebp \\
\text{ret }
\end{align*}
\]

swap (64-bit long ints):
\[
\begin{align*}
\text{movq } & (\%rdi), \%rdx \\
\text{movq } & (\%rsi), \%rax \\
\text{movq } & \%rax, (\%rdi) \\
\text{movq } & \%rdx, (\%rsi) \\
\text{ret }
\end{align*}
\]

- Operands 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

- **Callq instructions stores 64-bit address on stack**
  - Address pushed onto stack decrementing rsp by 8

- **Local variables also in registers (if there is room)**
  - Eliminates stack accesses and need to allocate stack space

- **Functions can access storage on stack up to 128 beyond rsp**
  - Can store some temps on stack without altering rsp

- **No frame pointer**
  - All references to stack made relative to rsp, no need to restore base ptr

- **Some registers designated “callee-saved”**
  - Must to restored by callee before ret
Passing differently sized arguments

- Use registers `%edi, %esi, %edx, ...` for 32-bit arguments
- Use registers `%di, %si, %dx, ...` for 16-bit arguments
- Use registers `%dil, %sil, %dl, ...` for 8-bit arguments

Useful instructions:
- `movzbl`: move byte to low-end of long zero-filled
- `movslq`: move long to low-end of quad sign-extended
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
subq $32,%rsp
movq $1,16(%rsp)
movl $2,24(%rsp)
movw $3,28(%rsp)
movb $4,31(%rsp)
...
```

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

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>x1</td>
<td></td>
</tr>
</tbody>
</table>
---|----|----|
|    |    |    |
|    |    |    |
|    |    |    |
|    |    |    |%rsp-32
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
movl  $1,%edi
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>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

|x1
| Arg 8
| Arg 7

%rsp (old rsp - 32)
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:

```asm
    ; 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
    movl  $1,%edi
    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>x1</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>

Return address to line after call to proc

%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:
```
    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>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arg 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arg 7</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:

```assembly
    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 (best case, less memory references)
  - Address relative to stack pointer when necessary
  - No more base pointer
  - Allocate/deallocate entire block

- Many optimizations
  - What kind of stack frame to use
  - Various allocation techniques