Today

- Lab 1 grades out later today (?)
  - Everybody did well.
  - One common issue: losing bits by casting `int*` to `int`.
    - Worked OK for `...Size`, but not `withinSameBlock/withinArray`. Why?

```
... 00000000 00000001 00000000 00000000 00000000 00000004   int*
     00000000 00000000 00000000 00000000 00000004   int
```

- Lab 2 due Wednesday.

- More on procedures, stacks, maybe data structures
  - Slides not posted yet, sorry...

- What is a stack?
  - And how about a stack `frame`?
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;
}
```

Calling `swap` from `call_swap`

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

Resulting Stack

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

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

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

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

\[
\begin{align*}
\text{call\_swap\ frame} \\
&\hspace{1cm} \text{\%ebp} \\
&\hspace{2cm} \ldots \\
&\hspace{3cm} \ldots \\
&\hspace{4cm} \ldots \\
&\hspace{5cm} \&\text{zip2} \\
&\hspace{6cm} \&\text{zipl} \\
&\hspace{7cm} \text{Rtn\ adr} \\
&\hspace{8cm} \%\text{esp} \\
\end{align*}
\]

Resulting Stack?

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

Set Up
swap Setup #1

Entering Stack

Resulting Stack

swap:

\[
\begin{align*}
\text{pushl } & \%\text{ebp} \\
\text{movl } & \%\text{esp}, \%\text{ebp} \\
\text{pushl } & \%\text{ebx}
\end{align*}
\]

Set Up
swap Setup #2

Entering Stack

Resulting Stack

swap:

\[
\begin{align*}
pushl & \%ebp \\
movl & \%esp, \%ebp \\
pushl & \%ebx \\
\end{align*}
\]
swap Setup #3

Entering Stack

Resulting Stack

\[
\text{call_swap frame} \quad \begin{cases} 
\&\text{zip2} \\
\&\text{zip1} \\
\text{Rtn adr} \\
\end{cases} \quad %\text{ebp} \quad %\text{esp}
\]

\[
\begin{array}{c}
\text{swap:} \\
\text{pushl} \ %\text{ebp} \\
\text{movl} \ %\text{esp},%\text{ebp} \\
\text{pushl} \ %\text{ebx} \\
\end{array} \quad \text{Set Up}
\]

\[
\begin{cases} 
\text{YP} \\
\text{xp} \\
\text{Rtn adr} \\
\end{cases} \quad \begin{cases} 
\text{Old } %\text{ebp} \\
\text{Old } %\text{ebx} \\
\end{cases} \quad %\text{ebp} \quad %\text{esp}
\]
swap Body

Entering Stack

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

Resulting Stack

\[
\begin{array}{c}
\bullet \\
\bullet \\
\bullet \\
& \%ebp \\
& \%esp \\
12 & \text{YP} \\
8 & \text{xp} \\
4 & \text{Rtn adr} \\
\text{Old \%ebp} & \%ebp \\
\text{Old \%ebx} & \%esp
\end{array}
\]

\[
\text{Offset relative to new \%ebp}
\]

\[
\text{movl } 12(\%ebp),\%ecx \# \text{get yp}
\]
\[
\text{movl } 8(\%ebp),\%edx \# \text{get xp}
\]

\[
\ldots
\]

Body
swap Finish #1

Finishing Stack

Resulting Stack

```
• • •
  %ebp
  %esp

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

```
swap Finish #2

Finishing Stack

Resulting Stack

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

Finish
swap Finish #3

Finishing Stack

Resulting Stack

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

Finish
swap Finish #4

Finishing Stack

Resulting Stack

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

Finish
Disassembled swap

080483a4 <swap>:

- 55  push %ebp
- e5  mov %esp,%ebp
- 53  push %ebx
- 8b 55 08  mov 0x8(%ebp),%edx
- 8b 4d 0c  mov 0xc(%ebp),%ecx
- 8b 1a  mov (%edx),%ebx
- 8b 01  mov (%ecx),%eax
- 89 02  mov %eax,(%edx)
- 89 19  mov %ebx,(%ecx)
- 5b  pop %ebx
- c9  leave
- c3  ret

Calling Code

- e8 96 ff ff ff  call 80483a4 <swap>
- 8b 45 f8  mov 0xfffffffff8(%ebp),%eax

relative address (little endian)
swap Finish #4

Finishing Stack

- YP
- xp
- Rtn adr
- Old %ebp
- Old %ebx

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

Resulting Stack

- %ebp
- %esp

Observation

- Saved & restored register \%ebx
- but not \%eax, \%ecx, or \%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?

  ```
  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 `you` calls `who`:
  - `you` is the *caller*
  - `who` is the *callee*

- Can a register be used for temporary storage?

- Conventions
  - "*Caller Save*"
    - Caller saves temporary values in its frame before calling
  - "*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;
}
```

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;
}
```

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) \hspace{1cm} val = 1
s_helper(3, &val)

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) 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)    val = 6.

Pass pointer to update location
Creating & Initializing Pointer

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

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

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
```
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**

Stack at time of call:

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>Rtn adr</th>
<th>Old %ebp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-4</td>
<td>0</td>
<td>-4</td>
<td>-4</td>
</tr>
<tr>
<td></td>
<td>-8</td>
<td>0</td>
<td>-8</td>
<td>-8</td>
</tr>
<tr>
<td></td>
<td>-12</td>
<td>0</td>
<td>-12</td>
<td>-12</td>
</tr>
<tr>
<td></td>
<td>-16</td>
<td>0</td>
<td>-16</td>
<td>-16</td>
</tr>
</tbody>
</table>

**Calling s_helper from sfact**

```assembly
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

- **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:**
  - Caller Frame
  - Arguments
    - Returns Addr
    - Old `%ebp`
    - Saved Registers + Local Variables
    - Argument Build
  - `%ebp` → `%esp`
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><code>%rax</code></td>
<td>Return value</td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>Argument #4</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>Argument #3</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>Argument #2</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>Argument #1</td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td>Stack pointer</td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%r8</code></td>
<td>Argument #5</td>
</tr>
<tr>
<td><code>%r9</code></td>
<td>Argument #6</td>
</tr>
<tr>
<td><code>%r10</code></td>
<td>Caller saved</td>
</tr>
<tr>
<td><code>%r11</code></td>
<td>Caller Saved</td>
</tr>
<tr>
<td><code>%r12</code></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%r13</code></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%r14</code></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%r15</code></td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
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):

\[
\begin{align*}
&\text{movq } (\%rdi), \%rdx \\
&\text{movq } (\%rsi), \%rax \\
&\text{movq } \%rax, (\%rdi) \\
&\text{movq } \%rdx, (\%rsi) \\
&\text{ret }
\end{align*}
\]

- **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”
- \texttt{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

2^N-1

Stack

Dynamic Data (Heap)

Static Data

Literals

Instructions

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

%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:

- 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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x4</td>
<td>x3</td>
<td>x2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

%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:

```
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

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

<table>
<thead>
<tr>
<th>Return address to caller of call_proc</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arg 7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

%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:

- 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

note sizes
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>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
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);
}
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
```
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