If you loved Lab 1/HW 1...

- Many authors. **HAKMEM.** 1978.


If you hated Lab 1/HW 1...

- We’ll use bit manipulations occasionally, but not like Lab 1! (phew!)

- But pointers and disassembly will be back...
Roadmap

C:

```c
#include <stdlib.h>

int main()
{
    // Allocate memory
    car *c = malloc(sizeof(car));
    c->miles = 100;
    c->gals = 17;
    float mpg = get_mpg(c);
    free(c);
    return 0;
}
```

Java:

```java
import java.util.*;

public class Car {
    private int miles;
    private int gals;

    public Car() {
        // Constructor
    }

    public void setMiles(int miles) {
        this.miles = miles;
    }

    public void setGals(int gals) {
        this.gals = gals;
    }

    public float getMPG() {
        return mpg;
    }
}
```

Assembly language:

```
get_mpg:
    pushq %rbp
    movq %rsp, %rbp
    ...
    popq %rbp
    ret
```

Machine code:

```
0111010000011000
1000110100000100
1000100111000010
1100000111111010
010000111110100001111
```

Computer system:

Windows

Mac

OS:

Data & addressing
Integers & floats
Machine code & C

x86 assembly programming

Procedures & stacks

Arrays & structs

Memory & caches

Processes

Virtual memory

Memory allocation

Java vs. C
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 ...

- Also, I got a remote thingy so I can run all over the room! Wheee!!
Memory Layout

- **Instructions**: At the bottom, containing program instructions.
- **Literals**: Above instructions, containing static data like literals (e.g., “example”).
- **Static Data**: Above literals, containing static variables (including global variables in C).
- **Dynamic Data (Heap)**: Above static data, containing dynamic data allocated with `new` or `malloc`.
- **Stack**: At the top, containing local variables and procedure context.

The diagram illustrates the memory layout with `2^N - 1` as the top boundary.
Memory Layout

- **Instructions**: read-only; executable. Initialized when process starts.
- **Literals**: writable; not executable. Managed by programmer.
- **Static Data**: writable; not executable. Managed when process starts.
- **Dynamic Data (Heap)**: writable; not executable. Managed by programmer.
- **Stack**: Managed "automatically" (by compiler).
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`  

(not extra-sensory perception)
IA32 Call Stack: Push

- `pushl Src`

Stack Pointer: `%esp`

Stack "Top" moves down as addresses increase.

Stack "Bottom" moves up as addresses increase.
IA32 Call Stack: Push

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

![Stack Diagram]

- Stack Pointer: %esp
- Stack “Top”
- Stack “Bottom”
- Increasing Addresses
- Stack Grows Down

Stack Grows Down
IA32 Call Stack: Pop

- `popl Dest`

Stack Pointer: `%esp`

Stack “Top”

Stack Grows Down

Increasing Addresses

Stack “Bottom”
IA32 Call Stack: Pop

- `popl Dest`
  - Load value from address `%esp`
  - Write value to `Dest`
  - Increment `%esp` by 4
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?
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
  - **Caller** might need to save registers that **Callee** might use
  - **Callee** might need to save registers that **Caller** has used
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:
    
    | Address | Instruction |
    |---------|-------------|
    | 804854e: | e8 3d 06 00 00 | `call 8048b90 <main>` |
    | 8048553: | 50          | `pushl %eax` |
  - Return address = 0x8048553

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

next instruction just happens to be a push could be anything
Procedure Call Example

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x804854e</td>
<td>e8 3d 06 00 00</td>
<td>call 8048b90 &lt;main&gt;</td>
</tr>
<tr>
<td>0x8048553</td>
<td>50</td>
<td>pushl %eax</td>
</tr>
</tbody>
</table>

```
call 8048b90
```

---

**%esp** 0x108

**%eip** 0x804854e

*%eip: program counter*
Procedure Call Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</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>

```
804854e: e8 3d 06 00 00
8048553: 50
```

```c
804854e:
call 8048b90

8048553:
pushl %eax
```
Procedure Call Example

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

%esp %eip

%esp  0x108    %esp  0x108

%eip  0x804854e  %eip  0x8048553

%eip:  program counter
Procedure Call Example

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

call 8048b90

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>0x8048553</td>
</tr>
</tbody>
</table>

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

%eip: program counter
Procedure Call Example

```
804854e:   e8 3d 06 00 00
8048553:   50

call 8048b90 <main>
pushl %eax
```

```
%esp 0x108
%eip 0x804854e
```

```
0x108 123
0x10c
0x110

call 8048b90

%esp 0x104
%eip 0x8048553

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

%eip: program counter
Procedure Return Example

8048591: c3 ret

%esp 0x104
%eip 0x8048591

0x104
0x108
0x10c
0x110

123

ret

%eip: program counter
Procedure Return Example

8048591: c3  ret

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

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

%eip: program counter
Procedure Return Example

8048591: c3 ret

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

%eip 0x8048553
ret

%eip 0x8048591
ret

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

%eip 0x8048553

%eip: program counter
Procedure Return Example

8048591: c3 ret

%esp
%eip

0x104
0x108
0x10c
0x110
0x8048553
123
0x8048591

ret

%esp
%eip

0x104
0x108
0x10c
0x110
0x8048553
123
0x8048591

%eip: program counter
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
```

Diagram:

```
 yoo
  ↓
 who
  ↓
 amI  amI
  ↓
 amI
  ↓
 amI
```
Example

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

![Stack Diagram]

- yoo
- %ebp
- %esp
- who
- amI
- amI
- amI
- amI
Example

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

Stack

```
%ebp

%esp

amI

who

yoo

```
Example

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

Stack

- yoo
- who
- amI
- amI
- %ebp
- %esp
Example

Example

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

Stack

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

Diagram:

- `amI` function call
- Stack frame with variables and function calls
- Arrows indicating function calls and stack movement
Example

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

Stack

- yoo
- who
- amI
- %ebp
- %esp
Example

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

Stack

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

Example

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

Stack

36
Example

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

Stack

```
%ebp
%esp
```
Example

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

Stack

```
%ebp
%esp
```

```
who
yoo
```

Example

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

How did we remember where to point `%ebp`?
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

```
Stack pointer %esp
Frame pointer %ebp
Caller Frame
Arguments
Return Addr
Old %ebp
Saved Registers + Local Variables
Argument Build
```
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
```
call_swap:
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
    ... ...
```

we know the address

not Global Var!
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

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

Revisiting \texttt{swap}

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

\texttt{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
```

\textbf{Set Up}

\textbf{Body}

\textbf{Finish}
swap Setup #1

Entering Stack

Resulting Stack?

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

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

**Entering Stack**

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

**Resulting Stack**

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

### swap:

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

**Set Up**
swap Setup #2

Entering Stack

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

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

Resulting Stack

\[
\begin{align*}
\text{\%ebp} & \\
\text{\%esp} & \\
\text{\text{Rtn adr}} & \\
\text{\text{Old \%ebp}} & \\
\text{\text{Yp}} & \\
\text{\text{Xp}}
\end{align*}
\]
swap Setup #3

Entering Stack

\[ \begin{array}{c}
\cdot \\
\cdot \\
\cdot \\
&\text{zip2} \\
&\text{zip1} \\
\text{Rtn adr} \\
\end{array} \]

\[ %ebp \]

\[ %esp \]

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

\[ \{ \text{Set Up} \] 

Resulting Stack

\[ \begin{array}{c}
\cdot \\
\cdot \\
\cdot \\
\text{yp} \\
\text{xp} \\
\text{Rtn adr} \\
\text{Old %ebp} \\
\text{Old %ebx} \\
\end{array} \]

\[ %ebp \]

\[ %esp \]
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

swap’s Stack

Resulting Stack?

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

swap’s Stack

Resulting Stack

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

swap’s Stack

Resulting Stack

why not pop? compiler’s choice

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

Finish
swap Finish #3

swap’s Stack

Resulting Stack

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

swap’s 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

```
0848409:  e8 96 ff ff ff  call 080483a4 <swap>
084840e:  8b 45 f8      mov 0xfffffffffffffff8(%ebp),%eax
```

relative address (little endian)
swap Finish #4

**swap’s Stack**

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

**Resulting Stack**

- %ebp
- %esp

**Observation**

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

```assembly
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?**

  **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 \texttt{you} calls \texttt{who}:
  - \texttt{you} is the \textit{caller}
  - \texttt{who} is the \textit{callee}

- Can a register be used for temporary storage?

- Conventions
  - \textit{“Caller Save”}
    - Caller saves temporary values in its frame before calling
  - \textit{“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} \text{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;
}
```

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) \hspace{1cm} val = 3
s_helper(2, &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;
}
```

sfact(3)          val = 1
s_helper(3, &val) val = 3
s_helper(2, &val) val = 6
s_helper(1, &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
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 (%ebp)`
- Push on stack as second argument

Initial part of `sfact`

```assembler
_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
```

<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 = 1</td>
</tr>
<tr>
<td>-8</td>
<td></td>
</tr>
<tr>
<td>-12</td>
<td>Unused</td>
</tr>
<tr>
<td>-16</td>
<td></td>
</tr>
</tbody>
</table>
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>Address</th>
<th>Variable</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></td>
</tr>
<tr>
<td>-16</td>
<td>&amp;val</td>
</tr>
<tr>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

#### 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

- **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 showing stack and registers]

- **Caller Frame**
  - %ebp
  - Old %ebp
  - Saved Registers + Local Variables
  - Argument Build
  - %esp