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...

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

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

Assembly language:
```assembly
pushq %rbp
movq %rsp, %rbp
...popq %rbp
ret
```

Machine code:
```
0111010000011000
100011010000010000000010
100010111100010
110000011111101000011111
```

OS: 
- Windows
- Mac

Procedures & stacks
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

Data & addressing
- Integers & floats
- Machine code & C
- x86 assembly programming
**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**
  - Writable; not executable
  - Initialized when process starts

- **Instructions**
  - Read-only; executable
  - Initialized when process starts

**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*

- Fetch value from `Src`
- Decrement `%esp` by 4 *(why 4?)*
- Store value at address given by `%esp`
IA32 Call Stack: Pop

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

Stack Pointer: `%esp`

Stack “Bottom”

Increasing Addresses

Stack Grows Down

Stack “Top”

Those bits are still there; we’re just not using them.

IA32 Call Stack: Pop

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

Stack Pointer: `%esp`

Stack “Bottom”

Increasing Addresses

Stack Grows Down

Stack “Top”

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

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`

- **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 Call Example

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

call 8048b90

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

%esp 0x108
%eip 0x804854e

%eip: program counter

---

Procedure Call Example

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

call 8048b90

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

%esp 0x108
%eip 0x804854e

%eip: program counter

---

Procedure Call Example

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

call 8048b90

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

%esp 0x108
%eip 0x804854e

%eip: program counter

---

Procedure Call Example

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

call 8048b90

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

%esp 0x108
%eip 0x804854e

%eip: program counter
Procedure Call Example

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

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

%eip: program counter

Procedure Return Example

```
8048591: c3 ret
```

%eip: program counter
Procedure Return Example

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

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

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

Example

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

Example

```
stack
```

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

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

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;
}
```

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;
}
```
**swap Setup #1**

**Entering Stack**
- %ebp
- %esp
- $zip2
- $zip1
- Rtn adr

**Resulting Stack**
- %ebp
- %esp

**Swap**
- pushl %ebp
- movl %esp, %ebp
- pushl %ebx

**Set Up**

---

**swap Setup #2**

**Entering Stack**
- %ebp
- %esp
- $zip2
- $zip1
- Rtn adr

**Resulting Stack**
- %ebp
- %esp
- %ebp
- %esp
- Old %ebp

**Swap**
- pushl %ebp
- movl %esp, %ebp
- pushl %ebx

**Set Up**

---

**swap Setup #3**

**Entering Stack**
- %ebp
- %esp
- $zip2
- $zip1
- Rtn adr

**Resulting Stack**
- %ebp
- %esp
- %ebp
- %esp
- Old %ebp
- Old %ebx

**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

swap’s Stack

Resulting Stack

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

Finish

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
Disassembled swap

```
080483a4 <swap>:
  80483a4:  55          push   %ebp
  80483a5:   89 e5       mov     %esp,%ebp
  80483a7:   53          push   %ebx
  80483a8:   8b 55 08    mov     0x8(%ebp),%edx
  80483ab:   8b 4d 0c    mov     0xc(%ebp),%ecx
  80483ae:   8b 1a       mov     (%edx),%ebx
  80483b0:   8b 01       mov     (%ecx),%eax
  80483b2:   89 02       mov     %eax,(%edx)
  80483b4:   89 19       mov     %ebx,(%ecx)
  80483b6:   5b          pop     %ebx
  80483b7:   c9          leave
  80483b8:   c3          ret
```

Calling Code
```
8048409:  e8 96 ff ff ff  call 80483a4 <swap>
804840e:  8b 45 ff       mov     0xffffffff(0x4f18),%eax
```

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?

  - Contents of register `%edx` overwritten by `who`

  ```
  yoo:  
  • • •
  movl $12345, %edx  
  call who  
  addl %edx, %eax  
  • • •
  ret
  
  who:  
  • • •
  movl 8(%ebp), %edx  
  addl $98195, %edx
  • • •
  ret
  ```

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;
}
```

### 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;
    sHelper(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
sHelper(3, &val)

Pass pointer to update location

Example: Pointers to Local Variables

Top-Level Call
```c
int sfact(int x)
{
    int val = 1;
    sHelper(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
sHelper(3, &val) val = 3
sHelper(2, &val)

Pass pointer to update location

Example: Pointers to Local Variables

Top-Level Call
```c
int sfact(int x)
{
    int val = 1;
    sHelper(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
sHelper(3, &val)

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
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>Offset</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>-16</td>
<td><code>%esp</code></td>
</tr>
<tr>
<td>-12</td>
<td>Unused</td>
</tr>
<tr>
<td>-8</td>
<td>`%ebp</td>
</tr>
<tr>
<td>0</td>
<td>Old <code>%ebp</code></td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>8</td>
<td><code>x</code></td>
</tr>
</tbody>
</table>

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`