Today
- IA32 stack discipline
- Register saving conventions
- Creating pointers to local variables

Next time
- Structured data
IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register $\%esp$ indicates lowest stack address
  - address of top element
IA32 Stack pushing

- **Pushing**
  - `pushl Src`
  - Fetch operand at `Src`
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`
IA32 Stack popping

- **Popping**
  - `popl Dest`
  - Read operand at address given by `%esp`
  - Increment `%esp` by 4
  - Write to `Dest`
Stack operation examples

Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
<td>0x108</td>
</tr>
<tr>
<td>0x10c</td>
<td>123</td>
</tr>
<tr>
<td>0x108</td>
<td>213</td>
</tr>
</tbody>
</table>

Push operation: `pushl %eax`

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
<td>0x108</td>
</tr>
<tr>
<td>0x10c</td>
<td>123</td>
</tr>
<tr>
<td>0x108</td>
<td>213</td>
</tr>
</tbody>
</table>

Pop operation: `popl %edx`

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
<td>0x108</td>
</tr>
<tr>
<td>0x10c</td>
<td>123</td>
</tr>
<tr>
<td>0x108</td>
<td>213</td>
</tr>
</tbody>
</table>

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>213</td>
</tr>
<tr>
<td>%edx</td>
<td>555</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>213</td>
</tr>
<tr>
<td>%edx</td>
<td>555</td>
</tr>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>213</td>
</tr>
<tr>
<td>%edx</td>
<td>213</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>
Procedure control flow

- Use stack to support procedure call and return

- Procedure call
  
  ```
  call label  
  Push return address on stack; jump to label
  call *Operand  Similar, but indirect
  ```

- Procedure return
  
  - `leave` Prepare stack for return
  - `ret` Pop address from stack; jump there
    (stack should be ready)

- Return address value
  
  - Address of instruction immediately following `call`
  - Example from disassembly

<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>

Return address: 0x8048553
Procedure call example

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x804854e</td>
<td><code>e8 3d 06 00 00</code></td>
<td>Call <code>0x8048b90 &lt;main&gt;</code></td>
</tr>
<tr>
<td>0x8048553</td>
<td><code>50</code></td>
<td>Pushl <code>%eax</code></td>
</tr>
</tbody>
</table>

- `eip` is the program counter.
- `%esp` points to the top of the stack.

Call `0x8048b90`:
- Push return address on stack
- Jump to `0x8048b90`
Procedure return example

%esp is program counter

ret

Pop address from stack
Jump to address
Stack-based languages

- Languages that support recursion
  - e.g., C, Pascal, Java
  - Code must be "reentrant"
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer

- Stack discipline
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does

- Stack allocated in *frames*
  - State of a single procedure instantiation
Call chain example

Code structure

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

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

amI(...) {
    •
    •
    •
}
```

Call Chain

Procedure `amI` recursive
Stack frames

- **Contents**
  - Local variables
  - Return information
  - Temporary space

- **Management**
  - Space allocated when enter procedure
    - “Set-up” code
  - Deallocated when return
    - “Finish” code

- **Pointers**
  - Stack pointer `%esp` indicates stack top
  - Frame pointer `%ebp` indicates start of current frame
Stack operation

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

Call Chain

```
Frame Pointer
%ebp

Stack Pointer
%esp
```

yoo

12
Stack operation

```
yoo(...) {
  •

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

Call Chain

- Frame Pointer `%ebp`
- Stack Pointer `%esp`
Stack operation

Call Chain

yoo(…)
{
  •
  who(…)
  {
    • • •
    amI(...)
    {
      •
      •
      •
      amI();
      •
      •
    }
  }
}

who

yoo

Frame Pointer
%ebp

Stack Pointer
%esp

amI
Stack operation

Call Chain

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

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

Call Chain

```
yoo
  who
    amI
      amI
        amI
```

Frame Pointer %ebp
Stack Pointer %esp
Stack operation

Call Chain

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

Frame Pointer
%ebp

Stack Pointer
%esp
Stack operation

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

Call Chain

- Frame Pointer %ebp
- Stack Pointer %esp

Diagram:
- yoo
- who
- amI
Stack operation

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

Call Chain

```
| Frame Pointer |
|----|---|
| %ebp | yoo |
| Stack Pointer |
| %esp | who |
```

Call Stack:
- `yoo` returns
- `who` called
- `amI` called
- `amI` called
- `amI` called
- `who` called
- `amI` called
- `amI` called
- `amI` called
- `yoo` called
Stack operation

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

Call Chain

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

- Frame Pointer: `%ebp`
- Stack Pointer: `%esp`
Stack operation

```
void yoo(...) {
    ....
}

void who(...) {
    ....
    ami();
    ami();
    ami();
    ami();
}
```

Call Chain

- `yoo`
- `who`
- `ami`
- `ami`
- `ami`
- `ami`

Frame Pointer `%ebp`
Stack Pointer `%esp`
Stack operation

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

Call Chain

- `yoo` function call
- `who` function call
- `amI` function call
- `amI` function call

Frame Pointer
- `%ebp`

Stack Pointer
- `%esp`

Diagram showing the call chain and stack operation.
IA32/Linux stack frame

- Current stack frame (“Top” to Bottom)
  - Parameters for function about to call
    - “Argument build”
  - Local variables
    - If can’t keep in registers
  - Saved register context
  - Old frame pointer

- Caller stack frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call
Revisiting swap

```c
int zip1 = 15213;
int zip2 = 91125;

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

void swap(int *xp, int *yp)
...
```

**Calling swap from call_swap**

```
call_swap:
    pushl  %ebp
    movl  %esp, %ebp
    subl  $8, %esp
    movl  $zip2, 4(%esp)
    movl  $zip1, (%esp)
    call  swap
    ...
```

**Resulting Stack**

- `%esp`
- `Rtn adr`
- `zip1`
- `&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 8(%ebp),%edx
    movl 12(%ebp),%ecx
    movl (%edx),%ebx
    movl (%ecx),%eax
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    popl %ebx
    popl %ebp
    ret
```

Set Up

Body

Finish
swap Setup #1

**Entering Stack**

- `%ebp` pointing to `&zip1`
- `%esp` pointing to `Rtn adr`
- Stack contents: 
  - `. . .`
  - `&zip2`
  - `&zip1`
  - `Rtn adr`

**Resulting Stack**

- `%ebp` pointing to `Old %ebp`
- `%esp` pointing to `Rtn adr`
- Stack contents: 
  - `. . .`
  - `yp`
  - `xp`
  - `Rtn adr`
  - `Old %ebp`

---

**swap:**

- `pushl %ebp`  
  - Save old `%ebp`
- `movl %esp,%ebp`  
  - Set `%ebp` as frame pointer
- `pushl %ebx`  
  - Save `%ebx`
swap Setup #2

Entering Stack

```
%ebp

&zip2
&zip1
Rtn adr

%esp
```

Resulting Stack

```
%ebp

YP
xp
Rtn adr

Old %ebp

%esp
```

swap:
- `pushl %ebp` Save old %ebp
- `movl %esp,%ebp` Set %ebp as frame pointer
- `pushl %ebx` Save %ebx
swap Setup #3

**Entering Stack**

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

**Resulting Stack**

- `%ebp` entry point
- `%esp`
- `YP`
- `xp`
- `Rtn adr`
- `Old %ebp`
- `Old %ebx`

**swap**:

- `pushl %ebp`  
  *Save old %ebp*
- `movl %esp,%ebp`  
  *Set %ebp as frame pointer*
- `pushl %ebx`  
  *Save %ebx*
Effect of swap setup

### Entering Stack

- **%ebp**
- &zip2
- &zip1
- Rtn adr

### Resulting Stack

- Offset (relative to %ebp)
  - 12: yp
  - 8: xp
  - 4: Rtn adr
  - 0: Old %ebp
  - Old %ebx

### Body

```
movl 12(%ebp),%ecx  # get yp
movl 8(%ebp),%edx  # get xp
... ...
```
Observation
  - Saved & restored register %ebx
swap Finish #2

- Restore saved \%ebp and ... set stack ptr to end of caller’s frame

```
popl %ebx
popl %ebp
ret
```

Pop address from stack & jump there
swap Finish #3

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

```assembly
popl %ebx  
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?

  - Contents of register `%edx` overwritten by `who`
Register saving conventions

- When procedure \texttt{yoo} calls \texttt{who}:
  - \texttt{yoo} is the \textit{caller}, \texttt{who} is the \textit{callee}

- Can a 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
IA32/Linux register usage

- **Integer registers**
  - Two have special uses
    - %ebp, %esp
  - Three managed as callee-save
    - %ebx, %esi, %edi
    - Old values saved on stack prior to using
  - Three managed as caller-save
    - %eax, %edx, %ecx
    - Do what you please, but expect any callee to do so, as well
  - Register %eax also stores returned value
Recursive factorial

int rfact(int x) {
    int result;
    if (x <= 1) {
        result = 1;
    } else {
        result = n* rfact(x-1);
    }
    return result;
}

Registers
- %eax used without first saving
- %ebx used, but save at beginning & restore at end

.globl rfact
.type rfact, @function
rfact:
pushl %ebp
movl %esp, %ebp
push %ebx
subl $4, %esp
movl 8(%ebp), %ebx
movl $1, %eax
cmpl $1, %ebx
jle .L3
leal -1(%ebx), %eax
cmovl %eax, (%esp)
call rfact
imull %ebx, %eax
.L3:
addl $4, %esp
popl %ebx
popl %ebp
ret
Recursive factorial

```plaintext
.globl rfact
.type rfact, @function

rfact:
    pushl %ebp             # Save old %ebp
    movl %esp, %ebp        # Set %ebp as frame pointer
    pushl %ebx             # Save old %ebx
    subl $4, %esp          # Allocate 4B in stack
    movl 8(%ebp), %ebx    # Get x
    movl $1, %eax          # result=1
    cmpl $1, %ebx          # Compare n:1
    jle .L3                # If <= goto done
    leal -1(%ebx), %eax    # Compute n - 1
    movl %eax, (%esp)      # Store it at top of stack
    call rfact             # Call rfact
    imull %ebx, %eax       # Compute result = return * n
    .L3:                    # done
    addl $4, %esp          # Deallocate 4B
    popl %ebx              # Restore %ebx
    popl %ebp              # Restore %ebp
    ret                    # Return result
```

```c
int rfact(int x)
{
    int result;
    if (x <= 1) {
        result = 1;
    } else {
        result = n* rfact(x-1);
    }
    return result;
}
```
Rfact stack setup

Entering Stack

rfact:

pushl %ebp
movl %esp, %ebp
pushl %ebx
Rfact body

• Registers
  - %ebx  Stored value of x
  - %eax
    - Temporary value of \( x-1 \)
    - Returned value from \( rfact(x-1) \)
    - Returned value from this call

```c
int rfact(int x) {
    int result;
    if (x <= 1)
        result = 1;
    else
        result = n* rfact(x-1);
    return result;
}
```
Rfact recursion

leal -1(%ebx),%eax

pushl %eax

call rfact

...
Return from Call

Assume that \( rfact(x-1) \) returns \((x-1)!\) in register %eax
Rfact completion

```
addl $4, %esp
popl %ebx
popl %ebp
ret
```
The 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
- Can be 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