Machine-Level Programming III - Procedures

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/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

---

Diagram details:
- Stack Pointer (%esp)
- Frame Pointer (%ebp)
- Return Address
- Arguments
- Saved Registers + Local Variables
- Argument Build
- Caller Frame
- Old %ebp
**IA32 Stack pushing**

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

Stack Grows Down

Increasing Addresses

Stack “Bottom”

Stack “Top”

Stack Pointer

-4
IA32 Stack popping

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

%esp  %eax  %edx
0x108  123  555
0x10c
0x110

%esp  %eax  %edx
0x108  123  555
0x10c
0x110

%esp  %eax  %edx
0x104  213
0x108
0x110

%esp  %eax  %edx
0x104  213
0x108
0x110

%esp  %eax  %edx
0x108
0x10c
0x110

%esp  %eax  %edx
0x108
0x10c
0x110

%esp  %eax  %edx
0x104
0x108
0x110

%esp  %eax  %edx
0x104
0x108
0x110

%esp  %eax  %edx
0x104
0x108
0x110

%esp  %eax  %edx
0x104
0x108
0x110

%esp  %eax  %edx
0x104
0x108
0x110

%esp  %eax  %edx
0x104
0x108
0x110
Stack operation examples

pushl %eax

%eax 213
%edx 555
%esp 0x108

0x108 123
0x10c 0x10
0x110 0x10

%eax 213
%edx 555
%esp 0x108

0x108 213
0x10c 123
0x110 213

%eax 213
%edx 555
%esp 0x104

0x108 213
0x10c 123
0x110 213
0x104 213
Stack operation examples

```
pushl %eax
```

Before:

<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>
</tbody>
</table>

After:

<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>213</td>
</tr>
<tr>
<td>0x110</td>
<td>213</td>
</tr>
</tbody>
</table>

- `%eax` starts with 0x104
- `%edx` starts with 0x10c
- `%esp` starts with 0x108
Stack operation examples

```
pushl %eax
```

Initial state:
```
%esp
%eax
%edx
```
```
0x104
555
213
```
```
0x108
123
```
```
0x10c
```
```
0x110
```

After pushl %eax:
```
%esp
%eax
%edx
```
```
0x104
213
```
```
0x108
123
```
```
0x10c
```
```
0x110
```

```
%eax
%edx
%esp
```
```
213
555
0x108
```

```
%eax
%edx
%esp
```
```
213
555
0x104
```

```
%eax
%edx
%esp
```
```
213
555
0x104
```
Stack operation examples

pushl %eax

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

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

<table>
<thead>
<tr>
<th>%eax</th>
<th>%edx</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>213</td>
<td>555</td>
<td>0x104</td>
</tr>
</tbody>
</table>
Stack operation examples

```
pushl %eax

%eax  213
%edx  555
%esp  0x108

0x108  123
0x10c  213
0x110   

popl %edx

%eax  213
%edx  555
%esp  0x104

0x108  123
0x10c  213
0x110   
```
Stack operation examples

```
# Initial state
%esp       %eax       %edx
0x108       123
0x10c       0x110

# After `pushl %eax`
%esp       %eax       %edx
0x108       123       213
0x10c       0x110

# After `popl %edx`
%esp       %eax       %edx
0x108       213       555
0x10c       0x110
```

```
%esp  %eax  %edx
0x108  213
0x10c

%esp  %eax  %edx
0x108  555
0x10c

%esp  %eax  %edx
0x108  0x104
0x10c
```
Stack operation examples

- **pushl %eax**
  - 0x108: 123
  - 0x104: 213

- **popl %edx**
  - 0x108: 123
  - 0x104: 213

**Variables:**
- %eax: 213
- %edx: 555
- %esp: 0x108

Sunday, October 16, 2011
Procedure control flow: call

- Use stack to support procedure call and return
- Procedure call
  ```
  call label  \text{Push return address on stack; Jump to label}
  call *Operand  \text{Indirect call/jump}
  ```
- Return address value
  - Address of instruction immediately following call
  - Example from disassembly
    ```
    804854e: e8 3d 06 00 00  call  8048b90
    <main>
    8048553: 50  \text{pushl}  %eax
    ```
  - Return address = $0x8048553$
Procedure control flow: return

Procedure return

- **leave** Prepare stack for return; equivalent to
  - `movl %ebp, %esp`
  - `popl %ebp`

- **ret** Pop address from stack; Jump to address (after stack is ready)
Procedure call example

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

pushl  %eax

%esp  0x108
0x110
0x10c
0x108  123
0x104
%eip  0x804854e
%eip  is program counter

%esp  0x108
0x110
0x10c
0x108  123
0x104
%eip  0x804854e
Procedure call example

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

0x108 0x10c 0x110
0x108 123
0x104

%esp 0x108
%eip 0x804854e

%esp 0x108
%eip 0x804854e

%eip is program counter
Procedure call example

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

0x108
0x10c
0x110
0x108
123

%esp  0x108
%esp  0x110
%eip  0x804854e
%eip  0x804854e

%eip is program counter
Procedure call example

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

Call 8048b90

%esp 0x108 %esp 0x110
%eip 0x804854e %eip 0x804854e

0x108 123 0x10c 123
0x104 0x8048553

%eip is program counter
Procedure call example

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

%eip is program counter
Procedure return example

8048591: c3  ret

%esp  0x104
%esp  0x104
%esp  0x08048591
%esp  0x08048591

0x110  0x10c  0x108  0x110  0x10c  0x108  0x104
123     123     123     123     123     123     123
0x08048553 0x08048553

%eip  0x8048591
%eip  0x8048591

%eip is program counter

Sunday, October 16, 2011
Procedure return example

8048591: c3

ret

ret

%esp 0x104
%eip 0x8048591
%esp 0x104
%eip 0x8048591

%esp 0x110
%eip 0x8048553
%esp 0x110
%eip 0x8048553

0x108 123
0x108 123

%eip is program counter
Procedure return example

8048591: c3

ret

ret

%esp  0x104
%esp  0x104

%eip  0x8048591
%eip  0x8048591

%eip is program counter
Procedure return example

8048591:  c3  

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

ret  

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

ret  

%eip  is program counter
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 for single procedure instantiation
Call chain example

Code structure

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

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

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

Procedure `amI` recursive

Call Chain

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

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

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

Call Chain

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

Diagram:
- Stack operation
- Call Chain
- Frame Pointer: `%ebp`
- Stack Pointer: `%esp`

Sunday, October 16, 2011
amI(...) 
{
    •
    •
    amI();
    •
    •
}
Stack operation

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

Call Chain

```
yoo
who
amI
amI
```

Frame Pointer
%ebp

Stack Pointer
%esp

Sunday, October 16, 2011
Stack operation

Call Chain

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

Stack Pointer %esp

Frame Pointer %ebp

yoo

who

amI

amI

amI

amI

amI

amI

Sunday, October 16, 2011
amI(...)  
{}  
  ...  
  amI();  
  ...  
⟩

Call Chain

Frame Pointer  
%ebp

Stack Pointer  
%esp

yoo  
who  
amI  
amI  
amI  
amI

Sunday, October 16, 2011
Stack operation

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

Call Chain

```
amI
    
who
    
yoo
```

Frame Pointer

```
%ebp
```

Stack Pointer

```
%esp
```

Sunday, October 16, 2011
Stack operation

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

Call Chain

Stack operation
Stack operation

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

Call Chain

```
yoo
  ▼ who
    ▼ amI
        ▼ amI
            ▼ amI
                ▼ amI
```

Frame Pointer
- `%ebp`

Stack Pointer
- `%esp`

Sunday, October 16, 2011
Stack operation

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

Call Chain

- `yoo` calls `who`
- `who` calls `amI`
- `amI` calls `amI`
- `amI` calls `amI`
- `amI` calls `amI`

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

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

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

Sunday, October 16, 2011
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

int zip1 = 15213;
int zip2 = 91125;

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 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

\[
\text{pushl } \%ebp \\
\text{movl } \%esp, \%ebp \\
\text{pushl } \%ebx
\]
swap Setup #1

Entering Stack

Resulting Stack

swap:

pushl %ebp
movl %esp,%ebp
pushl %ebx
swap Setup #2

Entering Stack

resulting Stack

swap:

pushl %ebp
movl %esp,%ebp
pushl %ebx
swap Setup #3

Entering Stack

Resulting Stack

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

Entering Stack

Resulting Stack

Offset (relative to %ebp)

%ebp

%ebp

%ebp

%ebp

%ebp

%esp

%esp

%esp

%esp

movl 12(%ebp),%ecx  # get yp
movl 8(%ebp),%edx  # get xp

Body

movl 12(%ebp),%ecx  # get yp
movl 8(%ebp),%edx  # get xp

Body
**swap Finish #1**

**swap’s Stack**

- **Observation**
  - Saved & restored register `%ebx`

```assembly
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
swap Finish #2

swap’s Stack

Offset
12
8
4
0
-4

yp
xp
Rtn adr
Old %ebp
Old %ebx

swap’s Stack

Offset
12
8
4
0

yp
xp
Rtn adr
Old %ebp
Old %ebp

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

```
swap's Stack

Offset
12  yp
8   xp
4   Rtn adr
0   Old %ebp

swap's Stack

Offset
12  yp
8   xp
4   Rtn adr

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

swap’s Stack

Offset

12
8
4
0

Old %ebp

Rtn adr

yp
xp

swap’s Stack

Offset

12
8
4

Rtn adr

%ebp

%esp

%ebp

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

Pop address from stack & jump there
swap Finish #3

swap’s Stack

Offset
12
8
4
0

yp
xp
Rtn adr
Old %ebp

%ebp
%esp

swap’s Stack

Offset
12
8
4

yp
xp
Rtn adr

%ebp
%esp

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
swap’s Stack

Offset
12  yp
8   xp
4   Rtn adr

Exiting 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 \texttt{yoo} calls \texttt{who}:
  – \texttt{yoo} is the \texttt{caller}, \texttt{who} is the \texttt{callee}

• Can register be used for temporary storage?

\begin{align*}
\text{yoo:} & \quad \text{who:} \\
\text{• • •} & \quad \text{• • •} \\
\text{movl } 15213, \%edx & \quad \text{movl } 8(\%ebp), \%edx \\
\text{call } \text{who} & \quad \text{call } \text{who} \\
\text{addl } \%edx, \%eax & \quad \text{addl } 91125, \%edx \\
\text{• • •} & \quad \text{• • •} \\
\text{ret} & \quad \text{ret}
\end{align*}

– Contents of register \%edx overwritten by \texttt{who}
Register saving conventions

- **When procedure** you calls who:
  - you 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
**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

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

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

```assembly
.globl rfact
    .type rfact,@function
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
  - `%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
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

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

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

Entering Stack

rfact:
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx

Sunday, October 16, 2011
Rfact stack setup

Entering Stack

rfact:

\[
pushl \%ebp
\]

\[
movl \%esp,\%ebp
\]

\[
pushl \%ebx
\]
Rfact stack setup

Entering Stack

rfact:
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx
Rfact stack setup

rfact:
pushl %ebp
movl %esp,%ebp
pushl %ebx

Entering Stack
Rfact stack setup

Entering Stack

rfact:
pushl %ebp
movl %esp,%ebp
pushl %ebx
Rfact stack setup

Entering Stack

```
rfact:
pushl %ebp
movl %esp,%ebp
pushl %ebx
```

Sunday, October 16, 2011
int rfact(int x) {
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}

Register Usage:
- %ebx: Stored value of x
- %eax: Temporary value of x-1
- Returned value from rfact(x-1)
- Returned value from this call
Rfact recursion

x
Rtn adr
Old %ebp
Old %ebx

%eax
%ebx
x

%eax
x-1
%ebx
x

%eax
x
%ebx

Rtn adr
Old %ebp
Old %ebx

%eax
x-1
%ebx
x

%eax
x
%ebx

Sunday, October 16, 2011
Rfact recursion

leal -1(%ebx),%eax

```
x
Rtn adr
Old %ebp
Old %ebx
```

```
x-1
%eax

%ebx
```

```
x
Rtn adr
Old %ebp
Old %ebx
```

```
x-1
%eax

%ebx
```

```
x
Rtn adr
Old %ebp
Old %ebx
```

```
x-1
%eax

%ebx
```
Rfact recursion

leal -1(%ebx),%eax

diagram showing the recursive call stack with
- `x`: register holding the current value
- `Rtn adr`: return address
- `%ebp`: base pointer
- `%esp`: stack pointer
- `%eax`: temporary register holding `x-1`
- `%ebx`: temporary register holding `x`

```
pushl %eax
```

```
elleal -1(%ebx),%eax
```

```
x
Rtn adr
Old %ebp
Old %ebx
%x
```

```
x
Rtn adr
Old %ebp
Old %ebx
x-1
```

```
x
Rtn adr
Old %ebp
Old %ebx
x-1
```

```
x
Rtn adr
Old %ebp
Old %ebx
x
```

```
x
Rtn adr
Old %ebp
Old %ebx
x-1
```

```
x
Rtn adr
Old %ebp
Old %ebx
x
```

```
x-1
```

```
x
```
Rfact recursion

leal -1(%ebx),%eax

pushl %eax

call rfact

Sunday, October 16, 2011
Assume that $r\text{fact}(x-1)$ returns $(x-1)!$ in register $%eax$
Rfact result

Return from Call

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

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

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

---

Sunday, October 16, 2011
Rfact completion

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

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

```
old %ebp
0 4
-4
-8

old %ebx
0
-8

pre %ebp
8
4
0

x-1
x

x
%ebx
%ebp
%eax
%esp
pre %ebp
pre %ebx
Rtn adr
Old %ebp
4
8
0

%ebp
%esp

%ebx
Old %ebx
```

```
old %ebp
0 4
-4
-8

old %ebx
0
-8

pre %ebp
8
4
0

x-1
x

x!
%eax
%ebx
Old %ebx
```

Sunday, October 16, 2011
Rfact completion

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

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

- 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