15-213
“The course that gives CMU its Zip!”

Machine-Level Programming III: Procedures
Sept. 17, 2002

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

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

Stack Pointer
%esp

Stack “Bottom”

Increasing Addresses

Stack Grows Down

Stack “Top”
Pushing

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

Stack Grows Down
Stack “Top”
Stack “Bottom”
Increasing Addresses

Stack Pointer `%esp`

-4
IA32 Stack Popping

Popping

- `popl Dest`
- Read operand at address given by `%esp`
- Increment `%esp` by 4
- Write to `Dest`

Diagram:

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

Image
Stack Operation Examples

```
Stack Operation Examples

<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>pushl %eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x10c</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>popl %edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x10c</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x104</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`

Return address value

- Address of instruction beyond `call`
- Example from disassembly
  - `804854e: e8 3d 06 00 00 call 8048b90 <main>`
  - `8048553: 50 pushl %eax`
    - Return address = 0x8048553

Procedure return:

- `ret` Pop address from stack; Jump to address
**Procedure Call Example**

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

0x804854e
8048553:  50

0x108 0x10c
0x108 123
0x104 0x8048553

call  8048b90

%esp  0x108
%esp  0x104
%eip  0x804854e
%eip  0x8048b90

%eip is program counter
```
**Procedure Return Example**

8048591:  c3  

```plaintext
ret
```

```
8048553:  c3  

ret
```

%esp  |  0x104  |  %esp  |  0x108  |
-----|---------|--------|---------|
%eip  |  0x8048591  |  %eip  |  0x8048553  |

%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
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

```
void who() {
    // stack frame
    // call chain
    // yoo
}
```
Stack Operation

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

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

yoo

who

who(É)
{
  ...
  amI();
  ...
  amI();
  ...
}
Stack Operation

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

Call Chain

```
  yoo
    .
    .
    who
      .
      .
      amI
```

Frame Pointer:
- %ebp

Stack Pointer:
- %esp
Stack Operation

```
ami(...) {
    
    ami();

}
```

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

ami
ami
ami
Stack Operation

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

Call Chain

- yoo
- who
- amI
- amI
- amI
- amI
- amI
- amI

Frame Pointer
- %ebp
Stack Pointer
- %esp
Stack Operation

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

Call Chain

```
ad
  who
  ami
  ami
  ami
```

Frame Pointer

Stack Pointer
Stack Operation

Call Chain

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

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

Call Chain

amI

Frame Pointer

Stack Pointer

yoo

who
Stack Operation

```
ami(...) {
    .
    .
    .
}
```

Call Chain

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

Frame Pointer %ebp
Stack Pointer %esp
Stack Operation

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

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

yoo

who

amI

amI

amI

amI
Stack Operation

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

Call Chain

```
Frame Pointer %ebp
Stack Pointer %esp
yoo
```

```
who
```

```
amI
```

```
amI
```
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)
{
    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

```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 %ebp,%esp
popl %ebp
ret
```

Set Up

Body

Finish
swap Setup #1

Entering Stack

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

Resulting Stack

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

\[
\begin{align*}
\text{wap:} & \\
\text{pushl \%ebp} & \\
\text{movl \%esp,\%ebp} & \\
\text{pushl \%ebx}
\end{align*}
\]
swap Setup #2

Entering Stack

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

\[
\begin{array}{c}
& \text{\%esp} \\
\end{array}
\]

\text{\textbf{swap:}}

\begin{verbatim}
pushl \%ebp
movl \%esp,\%ebp
pushl \%ebx
\end{verbatim}

Resulting Stack

\[
\begin{array}{c}
& \\text{\%ebp} \\
& \\text{\%esp} \\
\text{\%ebp} \\
& \\text{\%esp} \\
\text{Rtn adr} \\
\text{Old \%ebp} \\
\text{xp} \\
\text{yp} \\
\end{array}
\]
**swap Setup #3**

**Entering Stack**

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

**Resulting Stack**

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

swap:

```assembly
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
Effect of swap Setup

Entering Stack

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

Resulting Stack

\begin{align*}
\text{Offset (relative to \%ebp)} & \\
4 & \quad \text{\%esp} \\
8 & \quad \text{\%ebp} \\
12 & \quad \text{\%ebp}
\end{align*}

\begin{align*}
\text{Body} & \\
\text{movl} & \quad 12(\%\text{ebp}),\%\text{ecx} \ # \text{get yp} \\
\text{movl} & \quad 8(\%\text{ebp}),\%\text{edx} \ # \text{get xp} \\
\ldots & \\
\end{align*}
**swap Finish #1**

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

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

```
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
xp
yp
%ebp
%esp

swap’s Stack

Offset
12
8
4

%ebp
%esp
Rtn adr
xp
yp

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

- Saved & restored register %ebx
- Didn't do so for %eax, %ecx, or %edx

movl -4(%ebp),%ebx
movl %ebp,%esp
poopl %ebp
ret
Register Saving Conventions

When procedure `yoo` calls `who`:

- `yoo` is the caller, `who` is the callee

Can Register be Used for Temporary Storage?

```
yoo:
    ...
    movl $15213, %edx
    call who
    addl %edx, %eax
    ...
    ret

who:
    ...
    movl 8(%ebp), %edx
    addl $91125, %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 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
int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}

Recursive Factorial

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

movl 8(%ebp),%ebx      # ebx = x
cmpl $1,%ebx           # Compare x : 1
jle .L78               # If <= goto Term
leal -1(%ebx),%eax     # eax = x-1
pushl %eax             # Push x-1
call rfact             # rfact(x-1)
imull %ebx,%eax        # rval * x
jmp .L79               # Goto done

.L78:                    # Term:
    movl $1,%eax        # return val = 1

.L79:                    # Done:

%ebx  Stored value of x
%eax

- Temporary value of x-1
- Returned value from rfact(x-1)
- Returned value from this call
Rfact Recursion

leal -1(%ebx),%eax

Rtn adr
Old %ebp
Old %ebx

%eax  x-1
%ebx  x

call rfact

pushl %eax

%eax  x-1
%ebx  x

Rtn adr
Old %ebp
Old %ebx

%eax  x-1
%ebx  x

Rtn adr
Old %ebp
Old %ebx

%eax  x-1
%ebx  x

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

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
void s_helper(int x, int *accum) {
    if (x <= 1) return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper(x-1, accum);
    }
}

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

- Pass pointer to update location
Creating & Initializing Pointer

Initial part of \texttt{sfact}

\begin{verbatim}
_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
\end{verbatim}

Using Stack for Local Variable

- Variable \texttt{val} must be stored on stack
  - Need to create pointer to it
- Compute pointer as \texttt{-4 (\%ebp)}
- Push on stack as second argument

\begin{verbatim}
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
\end{verbatim}
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}

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
Using Pointer

```c
void s_helper (int x, int *accum) {
    int z = *accum * x;
    *accum = z;
}
```

- **Register `%ecx` holds `x`**
- **Register `%edx` holds pointer to `accum`**
  - Use access `%edx` to reference memory
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