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

Machine-Level Programming III:
Procedures
Sept. 14, 1999

Topics
• IA32 stack
• Stack-based languages
• Stack frames
• Register saving conventions
• Creating pointers to local variables
IA32 Stack

- Region of memory managed with stack discipline
- Register %esp indicates lowest allocated position in stack
  - i.e., address of top element

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

Popping
- popl  *Dest*
  - Read operand at address given by %esp
  - Increment %esp by 4
  - Write to *Dest*
### Stack Operation Examples

<table>
<thead>
<tr>
<th>%esp</th>
<th>%eax</th>
<th>%edx</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x108</td>
<td>123</td>
<td>213</td>
<td>0x104</td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**pushl %eax**

<table>
<thead>
<tr>
<th>%esp</th>
<th>%eax</th>
<th>%edx</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
<td>123</td>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**popl %edx**

<table>
<thead>
<tr>
<th>%esp</th>
<th>%eax</th>
<th>%edx</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x108</td>
<td>123</td>
<td></td>
<td>0x108</td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

%eax: 213
%edx: 555
%esp: 0x108
Procedure Control Flow

Use stack to support procedure call and return

Procedure call:

\[
\text{call } label \quad \text{Push return address on stack; Jump to } label
\]

Return address value

- Address of instruction beyond call
- Example from disassembly

\[
\begin{align*}
804854e: & \quad \text{e8 3d 06 00 00 call 8048b90 <main>} \\
8048553: & \quad 50 \quad \text{pushl %eax}
\end{align*}
\]

- Return address = 0x8048553

Procedure return:

- \text{ret} \quad \text{Pop address from stack; Jump to address}
Procedure Call / Return Example

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

%esp  0x108
%esp  0x10c
%esp  0x110
%eip  0x804854e
%eip  0x8048553
%eip  0x8048b90
%eip  0x8048553
%eip  0x8048b90
%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();
  •
  •
}
```

- Procedure `amI` recursive

Call Chain

```plaintext
yoo
  ↓
who
  ↓
amI
  ↓
amI
  ↓
amI
```
IA32 Stack Structure

Stack Growth
- Toward lower addresses

Stack Pointer
- Address of next available location in stack
- Use register %esp

Frame Pointer
- Start of current stack frame
- Use register %ebp

Stack Grows

Increasing Addresses
IA32/Linux Stack Frame

Callee Stack Frame ("Top" to Bottom)
- Parameters for called functions
- 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;
}
```

call_swap:
    ...
    pushl $zip2
    pushl $zip1
    call swap
    ...

Resulting Stack

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%esp</td>
</tr>
<tr>
<td>Return Address</td>
<td></td>
</tr>
<tr>
<td>&amp;zip1</td>
<td></td>
</tr>
<tr>
<td>&amp;zip2</td>
<td></td>
</tr>
</tbody>
</table>
```

class07.ppt
Revisiting \texttt{swap}

\begin{verbatim}
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}
swap Setup

Entering Stack

Resulting Stack

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

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

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

- Surmised by looking at code examples

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: %ebx, %esi, %edi
  - 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;
}
```

Complete Assembly

- Assembler directives
  - Lines beginning with “."  
  - Not of concern to us
- Labels
  - .Lxx
- Actual instructions

```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 %ebx,%esp
        popl %ebp
        ret
```
Rfact Stack Setup

Entering Stack

\[\text{rfact:}\]
\[
pushl \%ebp \\
movl \%esp,\%ebp \\
pushl \%ebx
\]

\begin{verbatim}
x
Rtn adr
\end{verbatim}

\begin{verbatim}
8
4
0
-4
\end{verbatim}

\begin{verbatim}
x
Rtn adr
Old \%ebp
Old \%ebx
\end{verbatim}

class07.ppt
### Rfact Body

```assembly
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:
```

### 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 rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1);
    return rval * x;
}
```
Rfact Recursion

\texttt{leal \textasciitilde1(\%ebx),\%eax}

\begin{center}
\begin{tabular}{|c|c|}
\hline
\%eax & \texttt{x-1} \\
\%ebx & \texttt{x} \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|c|}
\hline
\%eax & \texttt{x-1} \\
\%ebx & \texttt{x} \\
\hline
\end{tabular}
\end{center}

pushl \%eax

\begin{center}
\begin{tabular}{|c|c|}
\hline
\%eax & \texttt{x-1} \\
\%ebx & \texttt{x} \\
\hline
\end{tabular}
\end{center}

call rfact
Rfact Result

Return from Call

\[
\begin{array}{c|c}
\text{Rtn adr} & \%ebp \\
\hline
\text{Old \%ebp} & \%ebp \\
\text{Old \%ebx} & \%esp \\
\hline
x-1 & \%esp \\
\end{array}
\]

\[
\begin{array}{c|c}
\%eax & (x-1)! \\
\%ebx & x \\
\end{array}
\]

\[
\begin{array}{c|c}
\%eax & x! \\
\%ebx & x \\
\end{array}
\]

\[
imull \%ebx,\%eax
\]
Rfact Completion

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

Table:

<table>
<thead>
<tr>
<th>Base Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td>0</td>
</tr>
<tr>
<td>%ebx</td>
<td>Old</td>
</tr>
<tr>
<td>x-1</td>
<td>%esp</td>
</tr>
</tbody>
</table>

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

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);
    }
}
```

Top-Level Call

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

- Pass pointer to update location
- Uses tail recursion
  - But GCC only partially optimizes it
Creating & Initializing Pointer

Initial part of `sfact`

```
_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
```

Using Stack for Local Variable

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

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

Calling s_helper from sfact

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

Stack at time of call

-8
-4
0
4
8

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rtn adr</td>
</tr>
<tr>
<td>Old %ebp</td>
</tr>
<tr>
<td>val = 1</td>
</tr>
<tr>
<td>Unused</td>
</tr>
<tr>
<td>&amp;val</td>
</tr>
</tbody>
</table>

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 `accum`
  - Use access `%edx` to reference memory
Multi-Way Recursion

```c
int r_prod
  (int from, int to)
{
  int middle;
  int prodA, prodB;
  if (from >= to)
    return from;
  middle = (from + to) >> 1;
  prodA = r_prod(from, middle);
  prodB = r_prod(middle+1, to);
  return prodA * prodB;
}
```

- Compute product \( x \times (x+1) \times \ldots \times (y-1) \times y \)
- Split into two ranges:
  - Left: \( x \times (x+1) \times \ldots \times (m-1) \times m \)
  - Right: \( (m+1) \times \ldots \times (y-1) \times y \)

\[
m = \lfloor (x+y)/2 \rfloor
\]

- No real advantage algorithmically

Top-Level Call

```c
int bfact(int x)
{
  return r_prod(1, x);
}
```
Binary Splitting Example

\[ \text{bfact}(6) \]

\[ \text{r\_prod}(1,6) \]

\[ \text{r\_prod}(1,3) \quad \text{r\_prod}(4,6) \]

\[ \text{r\_prod}(1,2) \quad \text{r\_prod}(3,3) \]

\[ \text{r\_prod}(1,1) \quad \text{r\_prod}(2,2) \]

\[ \text{r\_prod}(4,5) \quad \text{r\_prod}(6,6) \]

\[ 1 \quad 2 \quad 4 \quad 5 \]

\[ 2 \quad 6 \]

\[ 120 \]

\[ 720 \]

\[ 6 \quad 120 \]

\[ 20 \quad 6 \]

\[ 4 \quad 5 \]
Multi-Way Recursive Code

Stack Frame

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>from</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>to</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Rtn Adr</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Old $ebp</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td>Old $edi</td>
<td></td>
</tr>
<tr>
<td>-8</td>
<td>Old $esi</td>
<td></td>
</tr>
<tr>
<td>-12</td>
<td>Old $ebx</td>
<td></td>
</tr>
</tbody>
</table>

$eax
from
return values

Callee Save Regs.

$ebx middle
$edi to
$esi prodA

_r_prod:

• • • # Setup
movl  8(%ebp),%eax # eax = from
movl 12(%ebp),%edi # edi = to
cmpl  %edi,%eax   # from : to
jge L8             # if >= goto done
leal (%edi,%eax),%ebx # from + to
sarl $1,%ebx       # middle
pushl %ebx          # 2nd arg: middle
pushl %eax          # 1st arg: from
call _r_prod        # 1st call
pushl %edi          # 2nd arg: to
movl %eax,%esi      # esi = ProdA
incl  %ebx           # middle + 1
pushl %ebx           # ... 1st arg
call _r_prod         # 2nd call
imull %eax,%esi      # ProdA * ProdB
movl %esi,%eax       # Return value

L8:                  # done:
• • •                   # Finish
Multi-Way Recursive Code Finish

Stack
- After making recursive calls, still has two arguments on top

Finishing Code
- Moves stack pointer to start of saved register area
- Pops registers
Mutual Recursion

Top-Level Call

```c
int lrfact(int x)
{
    int left = 1;
    return
    left_prod(&left, &x);
}
```

```c
int left_prod
(int *leftp, int *rightp)
{
    int left = *leftp;
    if (left >= *rightp)
        return left;
    else {
        int plus1 = left+1;
        return left *
        right_prod(&plus1, rightp);
    }
}
```

```c
int right_prod
(int *leftp, int *rightp)
{
    int right = *rightp;
    if (*leftp == right)
        return right;
    else {
        int minus1 = right-1;
        return right *
        left Prod(leftp, &minus1);
    }
}
```
Mutually Recursive Execution Example

Calling

- Recursive routines pass two arguments
  - Pointer to own local variable
  - Pointer to caller’s local variable
Implementation of lrfact

Call to Recursive Routine

int left = 1;
return left_prod(&left, &x);

Code for Call

leal 8(%ebp),%edx # edx = &x
pushl %edx # push &x
leal -4(%ebp),%eax# eax = &left
pushl %eax # push &left
call _left_prod # Call

Stack at time of call
Implementation of `left_prod`

Call to Recursive Routine

```c
int plus1 = left + 1;
return left *
    right_prod(&plus1, rightp);
```

Stack at time of call

```
-16
-8
0
4
8
12

%esp
%ebp
%ebx
%edx
%eax

rightp
leftp
Rtn adr
Old %ebp
Unused
&plus1
&plus1
```
Tail Recursion

Tail Recursive Procedure

```c
int t_helper
    (int x, int val)
{
    if (x <= 1)
        return val;
    return
t_helper(x-1, val*x);
}
```

General Form

```c
t_helper(x, val)
{
    ...  
    return
t_helper(Xexpr, Vexpr)
}
```

Top-Level Call

```c
int tfact(int x)
{
    return t_helper(x, 1);
}
```

Form
- Directly return value returned by recursive call

Consequence
- Can convert into loop
## Removing Tail Recursion

### Optimized General Form

```c
int t_helper(int x, int val)
{
    start:
    
    val = Vexpr;
    x = Xexpr;
    goto start;
}
```

### Resulting Code

```c
int t_helper
    (int x, int val)
{
    start:
    
    if (x <= 1)
    
        return val;
    
    val = val*x;
    x = x-1;
    goto start;
}
```

### Effect of Optimization

- Turn recursive chain into single procedure
- No stack frame needed
- Constant space requirement
  - Vs. linear for recursive version
Generated Code for Tail Recursive Proc.

Optimized Form

```c
int t_helper
    (int x, int val)
{
    start:
        if (x <= 1)
            return val;
        val = val*x;
        x = x-1;
    goto start;
}
```

Code for Loop

```
# %edx = x
# %ecx = val
L53:
    cmpl $1,%edx # x : 1
    jle L52 # if <= goto done
    movl %edx,%eax # eax = x
    imull %ecx,%eax # eax = val * x
    decl %edx # x--
    movl %eax,%ecx # val = val * x
    jmp L53 # goto start
L52:
    # done:
```

Registers

- $edx  x
- $ecx  val
Main Ideas

Stack Provides Storage for Procedure Instantiation
  • Save state
  • Local variables
  • Any variable for which must create pointer

Assembly Code Must Manage Stack
  • Allocate / deallocate by decrementing / incrementing stack pointer
  • Saving / restoring register state

Stack Adequate for All Forms of Recursion
  • Multi-way
  • Mutual