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

pushl %eax

popl %edx

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

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

%eax | 213
%edx | 213
%esp | 0x108
Procedure Control Flow

Use stack to support procedure call and return

Procedure call:

\texttt{call label} \hspace{1cm} \text{Push return address on stack; Jump to \textit{label}}

Return address value

- Address of instruction beyond \texttt{call}
- Example from disassembly

\begin{verbatim}
804854e: e8 3d 06 00 00 \hspace{1cm} call 8048b90 <main>
8048553: 50
\end{verbatim}

- Return address = 0x8048553

Procedure return:

- \texttt{ret} \hspace{1cm} \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 0x104 %esp 0x108
%eip 0x804854e %eip 0x8048b90 %eip 0x8048553

%esp 0x110 %esp 0x110 %esp 0x110
%eip 0x10c %eip 0x10c %eip 0x10c

0x108 123 0x108 123 0x108 123
0x104 0x8048553 0x104 0x104 0x104 0x104

%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

```c
yoo(\ldots)
{
  
  who();
  
}

who(\ldots)
{
  
  amI();
  
}

amI(\ldots)
{
  
  amI();
  
}

• Procedure amI recursive
```

Call Chain

```
yoo
\arrow{\downarrow}
who
\arrow{\downarrow}
amI
\arrow{\downarrow}
amI
\arrow{\downarrow}
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;
}
```

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

Resulting Stack

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
&zip2
&zip1
Rtn adr
%esp
```
Revisiting swap

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

Set Up

Body

Finish
swap Setup

Entering Stack

Resulting Stack

Swap:

pushl %ebp
movl %esp, %ebp
pushl %ebx

Offset

&zip2
&zip1
Rtn adr

%ebp
%esp

0
4
8
12

%ebp
%esp

Rtn adr
Old %ebp
Old %ebx

%ebp
%esp

yp
xp

class07.ppt
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?

<table>
<thead>
<tr>
<th>yoo:</th>
<th>who:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• • • movl $15213, %edx</td>
<td>• • • movl 8(%ebp), %edx</td>
</tr>
<tr>
<td>call who</td>
<td>addl $91125, %edx</td>
</tr>
<tr>
<td>addl %edx, %eax</td>
<td>• • •</td>
</tr>
<tr>
<td>• • • ret</td>
<td>ret</td>
</tr>
</tbody>
</table>

- 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
Rfact Stack Setup

Entering Stack

 Caller

 x
---
Rtn adr

%esp

Callee

 8
---
 4
---
 0
---
-4

 x
---
Rtn adr

 Old %ebp
---
 Old %ebx

Caller

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

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

Registers

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

```
    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:
```
Rfact Recursion

leal -1(%ebx),%eax

pushl %eax

call rfact
Rfact Result

Return from Call

<table>
<thead>
<tr>
<th>x</th>
<th>Rtn adr</th>
<th>Old %ebp</th>
<th>Old %ebx</th>
<th>x-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| %eax | (x-1)! |
| %ebx | x      |

imull %ebx, %eax

<table>
<thead>
<tr>
<th>x</th>
<th>Rtn adr</th>
<th>Old %ebp</th>
<th>Old %ebx</th>
<th>x-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| %eax | x!    |
| %ebx | x     |

class07.ppt
Rfact Completion

Rtn adr

Old %ebp

Old %ebx

x

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

x

%esp

%eax

x!

%ebx

Old %ebx

x-1

%esp

%ebp

%esp

%ebp

%esp

%ebp
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
- Uses tail recursion
  - But GCC only partially optimizes it
Creating & Initializing Pointer

Initial part of sfact

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

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
Passing Pointer

Calling s_helper from sfact

```c
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

Stack at time of call

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>val = 1</td>
</tr>
<tr>
<td>-8</td>
<td>Unused</td>
</tr>
<tr>
<td>-12</td>
<td></td>
</tr>
<tr>
<td>-16</td>
<td>&amp;val</td>
</tr>
<tr>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

class07.ppt
Using Pointer

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

```
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

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

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

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

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

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

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

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

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

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

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

\[ \text{r\_prod}(5, 5) \]
Multi-Way Recursive Code

Stack Frame

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

_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

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

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);
    }
}

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

```
left: 1
x: 4
```

```
left_prod( 1, 2 )
plus1: 2
```

```
right_prod( 1, 3 )
minus1: 3
```

```
left_prod( 1, 3 )
plus1: 3
```

```
right_prod( 1, 3 )
```

```
lrfact( 4 )
```

```
24
```

```
24
```

```
24
```

```
6
```

```
3
```
Implementation of \texttt{lrfact}

Call to Recursive Routine

\begin{verbatim}
int left = 1;
return left_prod(&left, &x);
\end{verbatim}

Code for Call

\begin{verbatim}
leal 8(%ebp),%edx # edx = &x
pushl %edx # push &x
leal -4(%ebp),%eax# eax = &left
pushl %eax # push &left
call _left_prod # Call
\end{verbatim}
Implementation of left_prod

Call to Recursive Routine

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

```
# %ebx holds left
# %edx holds rightp
leal 1(%ebx),%ecx # left+1
movl %ecx,-4(%ebp) # Store in plus1
pushl %edx # Push rightp
leal -4(%ebp),%eax # &plus1
pushl %eax # Push &plus1
call _right_prod # Call
```
Tail Recursion

Tail Recursive Procedure

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

General Form

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

Top-Level Call

```cpp
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