Giving credit where credit is due

- Most of slides for this lecture are based on slides created by Drs. Bryant and O’Hallaron, Carnegie Mellon University.
- I have modified them and added new slides.

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

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

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

Procedures 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 8049a90 <main>
- 8048553: 50 pushl %eax
- Return address = 0x8048553

Procedure return:
- ret Pop address from stack; Jump to address

Procedure Call Example

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

Procedure Return Example

<table>
<thead>
<tr>
<th>0x110</th>
<th>0x110</th>
<th>0x110</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10c</td>
<td>0x10c</td>
<td>0x10c</td>
</tr>
<tr>
<td>0x108</td>
<td>123</td>
<td>0x108</td>
</tr>
<tr>
<td>0x108</td>
<td>0x8048553</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>0x8048553</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>0x8048553</td>
<td></td>
</tr>
</tbody>
</table>

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
you() { 
  who(); 
  amI();
} 

who() { 
  amI();
} 

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

Procedure amI recursive

Call Chain

you
  who
    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
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
  - Old frame pointer

Caller Stack Frame
- Return address
- Pushed by call instruction
- Arguments for this call

Revisiting swap

void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

swap:
    push ebp
    movl ebp, esp
    pushl skb
    movl 12(skb), skb
    movl 8(skb), skb
    movl 4(skb), skb
    movl skb, skb
    movl skb, skb
    movl -4(skb), skb
    movl skb, skb
    ret

Calling swap from call_swap
call_swap:
    ...
    pushl %ebp # Global Var
    pushl %esp # Global Var
call swap
    ...

void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

swap Setup #1

Entering Stack

Resulting Stack

swap:
    pushl skb
    movl skb, skb
    pushl skb

Revisiting swap

void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

swap Setup #2

Entering Stack

Resulting Stack

swap:
    pushl skb
    movl skb, skb
    pushl skb

swap Setup #3

Entering Stack

Resulting Stack

swap:
    pushl skb
    movl skb, skb
    pushl skb

Page 5
Effect of swap Setup

Entering Stack

Resulting Stack

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

swap Finish #1

Observation
- Saved & restored register ebx

swap Finish #2

Observation
- Saved & restored register ebx

Register Saving Conventions
When procedure yoo calls who:
yoo is the caller, who is the callee

Can Register be Used for Temporary Storage?

Contents of register edx overwritten by who
Register Saving Conventions

When procedure `yoo` calls `who`:
- `yoo` 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:
  - `tbp`, `tebp`
  - Three managed as callee-save:
    - `tib`, `tiw`, `ti0`
  - Old values saved on stack prior to using:
  - Three managed as callee-save:
    - `tis`, `tix`, `ti0`
  - Do what you please, but expect any callee to do so, as well:
  - Register `tis` also stores returned value

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

```
.rglob rfact
.type rfact, @function
.rfact:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebp
    movl ($ebp), %ebx
    cmpi $1, %ebx
    jle .L78
    leal -(tibp), %eax
    pushl %eax
    call rfact
    imull %ebx, %eax
    jmp .L79
align 4
.L78:
    movi $1, %eax
.L79:
    movl -4(%ebp), %ebx
    movl %ebx, %ebp
    popl %ebp
    ret
```

Rfact Body

```
movl $(tibp), %ebx # ebx = x
cmpi $1, %ebx # Compare x : 1
jle .L78 # if <= goto Term
lea -1(tibp), %eax # eax = x-1
pushl %eax
# Push x-1
pushl %eax
# Push rfact
imull %tibp, %eax # real * 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

Rfact Recursion

```
lea -(tibp), %eax
```

```
.x
pushl %ebp
    movl %esp, %ebp
    pushl %ebp
    movl %ebp, %tibp
    pushl %tibp
    call rfact
    imull %ebx, %eax
    jmp .L79
align 4
.L78:
    movi $1, %eax
.L79:
    movl -4(%ebp), %ebx
    movl %ebx, %ebp
    popl %ebp
    ret
```

```
.x
Rtn ad
Old tibp
Old tibx
Rtn ad
pushl %ebp
    movl %esp, %ebp
    pushl %ebp
    movl %ebp, %tibp
    pushl %tibp
    call rfact
    imull %ebx, %eax
    jmp .L79
align 4
.L78:
    movi $1, %eax
.L79:
    movl -4(%ebp), %ebx
    movl %ebx, %ebp
    popl %ebp
    ret
```

```
.x
pushl %ebp
    movl %esp, %ebp
    pushl %ebp
    movl %ebp, %tibp
    pushl %tibp
    call rfact
    imull %ebx, %eax
    jmp .L79
align 4
.L78:
    movi $1, %eax
.L79:
```
Assume that `s_fact(x)` returns `x!` in register `eax`.

- **Passing Pointer**
  - **Calling `_s_helper` from `s_fact`**
  - **Stack at time of call**

  ```
  leal -4(%ebp), %eax # Compute %eax
  push %eax # Push on stack
  push %edx # Push %edx
  call _s_helper # Call
  movl -4(%ebp), %eax # Return val
  ...
  ```

  ```
  int s_fact(int x)
  { ...
    int val = 1;  %eax
    _s_helper(x, %eax);  %edx
    return val;
  }
  ```

- **Using Pointer**
  - **Register `%eax` holds `x`**
  - **Register `%edx` holds pointer to `accum`**
  - **Use access (%edx) to reference memory**

- **Creating & Initializing Pointer**

  ```
  _sfact:
  pushl %ebp
  # Save %ebp
  movl %esp, %ebp # Set %ebp
  subl $16, %esp # Add 16 bytes
  movl %eax, %edx # %edx = %eax
  ...
  ```

  ```
  int s_fact(int x)
  { ...
    int val = 1; # %eax
    _s_helper(x, %eax); # Call
    return val;
  }
  ```

- **Passing Pointer**
  - **Calling `_s_helper` from `_sfact`**

  ```
  leal -4(%ebp), %eax # Compute %eax
  push %eax # Push on stack
  push %edx # Push %edx
  call _s_helper # Call
  movl -4(%ebp), %eax # Return val
  ...
  ```

  ```
  int s_fact(int x)
  { ...
    int val = 1;  %eax
    _s_helper(x, %eax);  %edx
    return val;
  }
  ```

- **Pointer Code**
  - **Recursive Procedure**
  - **Top-Level Call**

  ```
  void _s_helper
  (int x, int *accum)
  { ...
    if (x <= 1)
      return;
    else {
      int z = *accum * x;
      *accum = z;
      _s_helper (x-1, accum);
    }
  }
  ```

  ```
  int s_fact(int x)
  { ...
    int val = 1; # %eax
    _s_helper(x, %eax); # Call
    return val;
  }
  ```

- **Passing Pointer**
  - **Calling `_s_helper` from `_sfact`**

  ```
  leal -4(%ebp), %eax # Compute %eax
  push %eax # Push on stack
  push %edx # Push %edx
  call _s_helper # Call
  movl -4(%ebp), %eax # Return val
  ...
  ```

  ```
  int s_fact(int x)
  { ...
    int val = 1; # %eax
    _s_helper(x, %eax); # Call
    return val;
  }
  ```

- **Using Pointer**
  - **Register `%eax` holds `x`**
  - **Register `%edx` holds pointer to `accum`**
  - **Use access (%edx) to reference memory**

- **Passing Pointer**
  - **Calling `_s_helper` from `_sfact`**

  ```
  leal -4(%ebp), %eax # Compute %eax
  push %eax # Push on stack
  push %edx # Push %edx
  call _s_helper # Call
  movl -4(%ebp), %eax # Return val
  ...
  ```

  ```
  int s_fact(int x)
  { ...
    int val = 1; # %eax
    _s_helper(x, %eax); # Call
    return val;
  }
  ```
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