Machine-Level Programming III: Procedures
Feb. 8, 2000

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

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 / Return Example**

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

  %esp %eip
0x110 0x110
0x10c 0x10c
0x108 123   0x108 123
0x104 0x8048553

does not fit

call 8048b90 %esp
ret

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

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

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

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

); End of Calling Chain

```

- Procedure amI recursive

---

**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 Allocated in Frames**
- state for single procedure instantiation
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

Callee Stack Frame

Caller Frame
- Arguments
- Return Addr
- Old %ebp

Frame Pointer (%ebp)

Saved Registers

Local Variables

Argument Build

Stack Pointer (%esp)

Revisiting swap

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

void call_swap()
{
  swap(&zip1, &zip2);
}

int zip1 = 15213;
int zip2 = 91125;
call_swap;

Revisiting swap

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

Revisiting swap

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

void call_swap()
{
  swap(&zip1, &zip2);
  call_swap;
}

int zip1 = 15213;
int zip2 = 91125;

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

void call_swap()
{
  swap(&zip1, &zip2);
  call_swap;
}

int zip1 = 15213;
int zip2 = 91125;
void call_swap()
{
  swap(&zip1, &zip2);
  call_swap;
}

Revisiting swap

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

void call_swap()
{
  swap(&zip1, &zip2);
  call_swap;
}

int zip1 = 15213;
int zip2 = 91125;
void call_swap()
{
  swap(&zip1, &zip2);
  call_swap;
}

int zip1 = 15213;
int zip2 = 91125;
void call_swap()
{
  swap(&zip1, &zip2);
  call_swap;
}
**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>Procedure</th>
<th>Contents of register %edx overwritten by whom</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>yoo</code></td>
<td>%edx</td>
</tr>
<tr>
<td><code>who</code></td>
<td>%ebx</td>
</tr>
</tbody>
</table>

**Conventions**
- "Caller Save" - Caller saves temporary in its frame before calling
- "Callee Save" - Callee saves temporary in its frame before using

**Recursive Factorial**

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

**Complete Assembly**

```assembly
.globl rfact
.rfact.@function
.rfact:
pushl %ebp
movl %esp,%ebp
movl $15213, %edx
call who
movl $91125, %edx
ret
```

**IA32/Linux Register Usage**

- Surmised by looking at code examples

**Integer Registers**
- Two have special uses %ebp, %esp
- Three managed as callee-save %eax, %esi, %edi
- Old values saved on stack prior to using %eax, %esi, %edi
- 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 %eax

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

**Recursive Factorial**

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

**Complete Assembly**

```assembly
.globl rfact
.rfact.@function
.rfact:
pushl %ebp
movl %esp,%ebp
movl $15213, %edx
call who
movl $91125, %edx
ret
```
Rfact Stack Setup

Entering Stack

Rtn adr
%esp

x

Rfact:

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

Rfact Body

x
硖

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

Rfact Recursion

leal -1(%ebx), %eax

x
硖

pushl %eax

x
硖

call rfact

x
硖

Rfact Result

Return from Call

imull %ebx, %eax

x
硖

imull %ebx, %eax

x
硖

Return val = 1
Rfact Completion

```
8  x
4  Rtn adr
0  %ebp
-4  Old %ebp
-8  x-1
%eax  x!
%ebx  x

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

Pointer Code

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

Passing Pointer
```
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
* * *
```

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
- Need to create pointer to it
- Compute pointer as -4(%ebp)
- Push on stack as second argument
```

Passing Pointer
```
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
* * *
```

Stack at time of call
```
8  x
4  Rtn adr
0  %ebp
-4  Old %ebp
-8  val = 1
12  Unused
-16  %esp
```

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;
}
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 $x \cdot (x+1) \cdot \ldots \cdot (y-1) \cdot y$
- Split into two ranges:
  - Left: $x \cdot (x+1) \cdot \ldots \cdot (m-1) \cdot m$
  - Right: $(m+1) \cdot \ldots \cdot (y-1) \cdot y$
- No real advantage algorithmically

Multi-Way Recursive Code

```
_stack_frame:
    _r_prod:
    
    ; Setup
    movl (%ebp), %eax # eax = from
    movl %edi, %edi # edi = to
    cmpl %edi, %eax # from : to
    jge L8 # if >= goto done
    ; Split into two ranges:
    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 # 1st 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
```

Binary Splitting Example

```
bfact(6)
    \[ r_{prod}(1,6) \]
    \[ \frac{1}{2} \cdot 3 \cdot 4 \cdot 5 \cdot 6 \]
    \[ = 720 \]
```

Top-Level Call

```c
int bfact(int x)
{
    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;
}
```
Multi-Way Recursive Code Finish

Stack
- After making recursive calls, still has two arguments on stack
- Moves stack pointer to start of saved register area
- Pops registers

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);
}
```

```
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);
    }
}
```

Implementation of lrfact

Call to Recursive Routine

- Pointer to own local variable
- Pointer to caller’s local variable

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

```
leal -12(%ebp),%esp # Set Stack Ptr
popl %ebx # Restore %ebx
popl %esi # Restore %esi
popl %edi # Restore %edi
movl %ebp, %esp # Restore %esp
popl %ebp # Restore %ebp
```

```
leal 8(%ebp),%edx # edx = &x
pushl %edx # push &x
```

```
leal -4(%ebp),%eax # eax = &left
pushl %eax # push &left
```

```
call _left_prod # Call
```

```
leal -12(%ebp),%esp # Set Stack Ptr
popl %ebx # Restore %ebx
popl %esi # Restore %esi
popl %edi # Restore %edi
movl %ebp, %esp # Restore %esp
popl %ebp # Restore %ebp
```

```
ret # Return
```

```
```
```
```
```
### Implementation of left_prod

#### Call to Recursive Routine

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

#### Stack at time of call

- `left` at `%ebp`
- `right` at `%ebp + 4`
- `Rtn adr` at `%ebp + 8`
- `Old %ebp` at `%ebp + 12`
- `plus1` at `%ebp + 16`
- `rightp` at `%ebp + 20`
- `&plus1` at `%ebp + 24`

- `%ebx` holds `left`
- `%edx` holds `rightp`

#### Call 

- `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

#### General Form

```c
t_helper(x, val) {
    • • •
    return t_helper(x-1, val*x);
}
```

#### Form

- Directly return value returned by recursive call

#### Consequence

- Can convert into loop

### Removing Tail Recursion

#### Optimized General Form

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

#### Resulting Code

```c
int t_helper;
    (int x, int val) {
        start:
        • • •
        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

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

#### Registers

- `$edx x$
- `$ecx val$

---

**Page 9**
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