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

- **Pushl %eax**
  - pushl %eax
  - popl %edx
- **Values:**
  - 0x10: 0x10
  - 0x10c: 0x10c
  - 0x108: 123
  - %eax: 213
  - %edx: 555
  - %esp: 0x108

Procedure Control Flow

- **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

- **Example from disassembly:**
  - 804854e: e8 3d 06 00 00 call 8048b90 <main>
  - 8048553: 50 pushl %eax
- **Values:**
  - 0x10: 0x10
  - 0x10c: 0x10c
  - 0x108: 123
  - %esp: 0x108
  - %eip: 0x804854e

- **Program Counter:** %eip is the program counter
- **Next instruction:** (bump up %eip to next instruction)
Procedure Call Example

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

0x110
0x10c
0x108 123

(%eip to next instruction)

%esp 0x108
%eip 0x8048553

%eip is the program counter

Procedure Call Example

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

0x110
0x10c
0x108 123
0x104 0x8048553

(%eip to next instruction)

%esp 0x104
%eip 0x8048553

%eip is the program counter

Procedure Call Example

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

0x110
0x10c
0x108 123
0x104 0x8048553

(%eip to next instruction)

%esp 0x104
%eip 0x8048553

%eip is the program counter

Procedure Return Example

8048591: c3 ret

8048591: c3 ret

0x110
0x10c
0x108 123
0x104 0x8048553

(%eip to next instruction)

%esp 0x104
%eip 0x80485591

%eip is the program counter
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
- Callee returns before caller does
- State for given procedure needed for limited time
  - When?

Stack Allocated in Frames
- state for single procedure instantiation

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
- Callee returns before caller does
- State for given procedure needed for limited time
  - From when called to when return

Stack Allocated in Frames
- state for single procedure instantiation
Call Tree Example

Code Structure

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

who(...) {
    ...
    ami();
    ...
    ami();
    ...
}

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

- Procedure ami recursive

Call Tree

Stack Frames

Contents
- Local variables
- Return information
- Temporary space

Management
- Space allocated when enter procedure
  - "Set-up" code (prolog)
- Deallocated when return
  - "Finish" code (epilog)

Pointers
- Stack pointer %esp indicates stack top
- Frame pointer %ebp indicates base of current frame

Stack Operation

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

Stack Operation

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

Stack Operation

```c
yoo

who

ami

Call Tree

```
Stack Operation

Call Tree

Stack Pointer

ebp

Stack Pointer

tesp

yoo

amI

Frame Pointer

tebp

yoo

amI

Who

amI

Who

frame

pointer

Stack

Pointer

Stack

Pointer

Call Tree

Stack Operation

Stack Operation

Stack Operation

Stack Operation

Copyright CMU, 2003
Stack Operation

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

Calling swap from call_swap

Resulting Stack

Revisiting swap

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

void swap(int *xp, int *yp)
{
    // Global Var
    pushl %edi
    // Global Var
    pushl %esi
    call swap
    *

void swap(int *xp, int *yp)
{
    // Global Var
    pushl %ebp
    movl %esp, %ebp
    pushl %esi
    pushl %edi
    leal (%esi, %edi, 4), %eax
    movl -4(%ebp), %edi
    movl %edi, %esi
    movl (%esi), %edi
    movl %edi, %esi
    movl -8(%ebp), %edi
    movl %edi, %esi
    movl %edi, %esp
    ret
}

int zipl = 15213;
int zip2 = 91125;
void call_swap()
{
    swap(%edi, %esi);
}

int zip1 = 15213;
int zip2 = 91125;
void call_swap()
{
    swap(%edi, %esi);
}
The image contains diagrams and text explaining the process of stack manipulation and memory management using assembly language instructions. Here's a breakdown of the key points:

**Swapping Registers**

- **Swapping Prolog #2**:
  - The diagram shows the entering stack with registers and the resulting stack after the swap operation.

- **Swapping Prolog #1**:
  - Similar to #2, but with different values in the stack.

- **Effect of swap Prolog**:
  - This section explains the effect of swap prolog on the stack, including the offset and the registers involved.

- **Swapping Prolog #3**:
  - Another example of a swap prolog with detailed explanations of the register values and stack changes.

The diagrams illustrate the movement of registers like `%ebp`, `%esp`, and `%ebx`, and how they change in the stack after the swap operations. The text accompanying these diagrams provides a step-by-step explanation of the assembly code used to achieve these changes, focusing on the `pushl`, `movl`, and `swap` instructions.
Register Saving Conventions

When procedure you calls who:
- `who` is the caller, `who` is the callee
- "Caller-Save":
  - "Caller-Save" temporary in its frame before calling
- "Callee-Save" temporary in its frame before using

Can Register be Used for Temporary Storage?
- `yooy` is the caller, `who` is the callee

Contents of register `who` overwritten by `yooy`:
- `yooy`:
  - `%edi, %esi, %edi, %esi, %edi, %esi, ...`

When procedure you calls who:
- `who` is the caller, `who` is the callee

Contents of register `who` overwritten by `yooy`:
- `yooy`:
  - `yooy %edi, %esi, %edi, %esi, %edi, %esi, ...`

SWAP: 1 mo' time

```c
void swap(int *xp, int *yp)
{
    int t = *xp;
    *xp = *yp;
    *yp = t;
}

void call_swap(void)
{
    int x = 1;
    int y = 2;
    swap(&x, &y);
    printf("%d %d\n", x, y);
}
```

Swaps:
- `pushl %ebp`
- `movl %esp, %ebp` (Temporaries)
- `addl $12, %ebp` (Temporaries)
- `movl $12, %eax`
- `call _swap`
- `movl %eax, %esp`
- `pushl %eax`
- `pushl %eax`
- `pushl %eax`
- `pushl $LC0`
- `call _printf`
- `movl %ebp, %esp`
- `popl %ebp`
- `ret`
Void callswap(void) {
    int x = 1;
    int y = 2;
    swap(&x, &y);
    printf("%d %d", x, y);
}

call swap:
push l %ebp
movl %esp, %ebp
subl $24, %esp
movl $1, -8(%ebp)
addl $-8, %esp
movl $2, -4(%ebp)
subl $24, %esp
movl $1, -8(%ebp)
addl $-8, %esp
movl $2, -4(%ebp)
lea -4(%ebp), %eax
push %eax
lea -8(%ebp), %eax
push %eax
lea _swap
movl -4(%ebp), %eax
addl $-4, %esp
push %eax
movl 8(%esp), %ebx
cmpl $1, %ebx
jle L78
lea -1(%ebx), %eax
push %eax
call rfact
imull %ebx, %eax
jmp L79
.align 4
L78:
movl $1, %eax
L79:
movl -4(%ebp), %ebx
movl %ebp, %esp
pop %ebp
ret

Recursive Factorial

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

Where is X? Where is rval?

After swap's prolog

Recursive Factorial

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

Where is X? Where is rval?
Rfact Stack Prolog

Entering Stack

Can now see that argument, \( x \), is at 8 (%ebp)

Rfact Body

\[
\begin{align*}
\text{movl} \quad & 8(\%ebp), \%ebx \quad \# \%ebx = x \\
\text{cmp} \quad & 1, \%ebx \quad \# \text{Compare} \ x : 1 \\
\text{jle} \quad & .L78 \quad \# \text{If} <= \text{goto} \ \text{Term} \\
\text{leal} \quad & -1(\%ebx), \%eax \quad \# \%eax = x-1 \\
\text{pushl} \quad & \%eax \quad \# \text{Push} \ x-1 \\
\text{call} \quad & \text{rfact} \quad \# \text{rfact}(x-1) \\
\text{imull} \quad & \%ebx, \%eax \quad \# \%rval \star \ x \\
\text{jmp} \quad & .L79 \quad \# \text{Goto done} \\
\text{movl} \quad & 1, \%eax \quad \# \text{return val} = 1 \\
\end{align*}
\]

int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1); 
    return rval \star x;
}

Registers

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

Copyright CMU, 2003
```c
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

---

```c
.globl rfact
.type rfact, @function
rfact:
    push ebp
    movl esp, ebp
    push esp
    movl $1, %ebx
    jle .L78
    leal -1(%ebx), %eax
    push %eax
    call rfact
    imull %ebx, %eax
    jmp .L79
    .align 4
.L78:
    movl $1, %eax
    # return val = 1
.L79:
```

**Registers**
- %eax Stored value of x
- %ebx Stored value of x - 1
- Temporary value of x - 1
- Returned value from rfact(x - 1)
- Returned value from this call

---

```asm
movl %ebp, %ebx # ebx = x
cmpl $1, %ebx # Compare x : 1
jle .L78 # If <= goto Term
lea -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
movl $1, %eax # return val = 1
.L78:
```

** Registers**
- %eax Stored value of x
- Temporary value of x - 1
- Returned value from rfact(x - 1)
- Returned value from this call

---

```asm
movl %ebp, %ebx # ebx = x
cmpl $1, %ebx # Compare x : 1
jle .L78 # If <= goto Term
lea -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
movl $1, %eax # return val = 1
.L79:
```

** Registers**
- %eax Stored value of x
- Temporary value of x - 1
- Returned value from rfact(x - 1)
- Returned value from this call
After Rfact Recursion?

```
leal -1(%ebx), %eax
pushl %eax
call rfact
```

- 57 -

```
leal -1(%ebx), %eax  # eax = x
pushl %eax  # Push x-1
call rfact  # rfact(x-1)
imull %ebx,%eax  # rval = x
```

- 58 -

```
%ebx has value of x
%eax has value of (x-1)!
```

Rfact Result

```
Return from Call
imull %ebx, %eax
```

- 59 -

```
Assume that rfact(x-1) returns (x-1)! in register %eax
```

Rfact Epilog

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

- 59 -

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

- 60 -

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

- 61 -

```
# Pass pointer to update location
```

Copyright CMU, 2003
Creating & Initializing Pointer

Initial part of sfact

```
_sfact:
pushl %ebp  # Save %ebp
movl %esp,%ebp  # Set %ebp
subl $16,%esp  # Add 16 bytes
movl (%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

Using Pointer

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

```

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 reverse 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
```