Machine-Level Programming III: Procedures
Sept. 16, 2003

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

Stack "Bottom"

Increasing Addresses

Stack “Top”

Stack Grows Down

Stack Pointer %esp

-2-
IA32 Stack Pushing

Pushing

- `pushl Src`
- Fetch operand at `Src`
- Decrements `%esp` by 4
- Writes 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

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:**

```plaintext
call label  Push return address on stack; Jump to label
```

**Return address value**

- Address of instruction beyond `call`
- Example from disassembly
  
  ```plaintext
  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

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

call 8048b90

%esp 0x108 %esp 0x104
%eip 0x804854e %eip 0x8048b90

%eip is program counter
Procedure Return Example

8048591: c3

ret

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

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

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

- Procedure amI recursive

Call Chain
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
Stack Operation

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

Call Chain

```
Frame Pointer %ebp

Stack Pointer %esp
```

- 12 -
Stack Operation

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

Call Chain

- Frame Pointer `%ebp`
- Stack Pointer `%esp`
- who
  - yoo

- who
  - yoo
  - ...

15-213, F'03
Stack Operation

```
myI (...) {
  ...
  myI(k);
  ...
}
```

Call Chain

```
  myI
  who
  myI
```

Frame Pointer %ebp
Stack Pointer %esp
Stack Operation

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

Call Chain

```
Frame Pointer
%ebp
```

```
Stack Pointer
%esp
```

```
 Frame Pointer
%ebp
```

```
Stack Pointer
%esp
```

```
Frame Pointer
%ebp
```

```
Stack Pointer
%esp
```

```
Frame Pointer
%ebp
```

```
Stack Pointer
%esp
```

```
Frame Pointer
%ebp
```

```
Stack Pointer
%esp
```

```
Frame Pointer
%ebp
```

```
Stack Pointer
%esp
```

```
Frame Pointer
%ebp
```

```
Stack Pointer
%esp
```

```
Frame Pointer
%ebp
```

```
Stack Pointer
%esp
```

```
Frame Pointer
%ebp
```

```
Stack Pointer
%esp
```

```
Frame Pointer
%ebp
```

```
Stack Pointer
%esp
```

```
Frame Pointer
%ebp
```

```
Stack Pointer
%esp
```

```
Frame Pointer
%ebp
```

```
Stack Pointer
%esp
```

```
Frame Pointer
%ebp
```

```
Stack Pointer
%esp
```
Stack Operation

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

Call Chain

```
Frame Pointer %ebp
Stack Pointer %esp
```
Stack Operation

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

Call Chain

- `yoo`
- `who`
- `amI`
- `amI`
- `amI`
- `amI`
- `Frame Pointer %ebp`
- `Stack Pointer %esp`
Stack Operation

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

Call Chain

```
Frame Pointer %ebp
Stack Pointer %esp
```
**Stack Operation**

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

**Call Chain**

- Frame Pointer
- Stack Pointer

- yoo
- who
- amI
- amI
- amI
Stack Operation

amI (...)  
{  
  •  
  •  
  •  
}  

Call Chain

Frame Pointer  
%ebp  

Stack Pointer  
%esp  

yoo  

who  

amI  

yoo  

who  

amI  

amI  

amI  

amI
Stack Operation

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

Call Chain

```
Frame Pointer %ebp
Stack Pointer %esp
```

- 21 -
Stack Operation

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

Call Chain

```
yoo
   who
   amI
   amI
   amI
   amI

Frame Pointer %ebp
Stack Pointer %esp
```
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

```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;
}
```

Calling swap from call_swap

call_swap:

```
    pushl $zip2   # Global Var
    pushl $zip1   # Global Var
    call swap
```

Resulting Stack

- `&zip2`
- `&zip1`
- Rtn adr
- `%esp`
Revisiting swap

```c
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
```
swap Setup #1

Entering Stack

Resulting Stack

\[
\begin{align*}
\text{Enterin} & \quad \text{Rtn addr} \\
\text{Stack} & \quad \%ebp \\
\text{\&zip2} & \quad \%esp \\
\text{\&zip1} & \\
\text{Rtn addr} & \quad \%ebp \\
\text{\%ebp} & \quad \%esp
\end{align*}
\]

\[
\begin{align*}
\text{Resulting} & \quad \text{Rtn addr} \\
\text{Stack} & \quad \%ebp \\
\text{\&zip2} & \quad \%esp \\
\text{\&zip1} & \quad \%esp \\
\text{Rtn addr} & \quad \%ebp \\
\text{\%ebp} & \quad \%esp
\end{align*}
\]

swap:

\[
\begin{align*}
pushl & \%ebp \\
movl & \%esp,\%ebp \\
pushl & \%ebx
\end{align*}
\]
**swap Setup #2**

**Entering Stack**

- `%ebp`  
- `%esp`

- &zip2
- &zipl
- Rtn adr

**Resulting Stack**

- `%ebp`  
- `%esp`

- YP
- xp
- Rtn adr
- Old `%ebp`

**swap:**

- `pushl %ebp`
- `movl %esp,%ebp`
- `pushl %ebx`
swap Setup #3

Entering Stack

\[
\begin{array}{c}
\vdots \\
& & \%ebp \\
& & \%esp \\
& & \%esp \\
& & \&zip2 \\
& & \&zipl \\
Rtn adr \\
\end{array}
\]

Resulting Stack

\[
\begin{array}{c}
\vdots \\
\vdots \\
& & \%esp \\
& & \%ebp \\
& & \%ebp \\
& & \%ebp \\
Rtn adr \\
\end{array}
\]

\[
\text{swap:} \\
pushl \%ebp \\
movl \%esp, \%ebp \\
pushl \%ebx
\]
Effect of swap Setup

Entering Stack

\[ \text{\%ebp} \]

Resulting Stack

\[ \text{\%ebp} \]
\[ \text{\%esp} \]
\[ \text{\%esp} \]
\[ \text{\%ebp} \]
\[ \text{\%esp} \]

\begin{align*}
\text{movl} & 12(\%ebp), \%ecx \ # \ get \ \text{yp} \\
\text{movl} & 8(\%ebp), \%edx \ # \ get \ \text{xp} \\
\ldots
\end{align*}

Body
swap Finish #1

Observation
- Saved & restored register %ebx

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
swap Finish #2

swap’s Stack

Offset
12
8
4
0
-4

Old %ebp
Old %ebx

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

swap’s Stack

Offset
12
8
4
0

Old %ebp
%ebp
%esp
%esp

15-213, F’03
swap Finish #3

swap’s Stack

Offset
12
8
4
0

Old %ebp

%ebp

%esp

Offset
12
8
4

%ebp

%esp

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
**swap Finish #4**

**swap’s Stack**

- Offset
  - 12: yp
  - 8: xp
  - 4: Rtn adr

**Exiting Stack**

- &zip1
- &zip2

---

**Observation**

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

---

```c
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
Register Saving Conventions

When procedure \texttt{yoo} calls \texttt{who}:

- \texttt{yoo} is the \textit{caller}, \texttt{who} is the \textit{callee}

Can Register be Used for Temporary Storage?

\texttt{yoo}:

\begin{verbatim}
    ... 
    movl $15213, %edx
    call who
    addl %edx, %eax
    ... 
    ret
\end{verbatim}

\texttt{who}:

\begin{verbatim}
    ... 
    movl 8(%ebp), %edx
    addl $91125, %edx
    ... 
    ret
\end{verbatim}

- Contents of register \%edx overwritten by \texttt{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
**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
  - %eax, %edx, %ecx
  - 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;
}
```

Registers

- `%eax` used without first saving
- `%ebx` used, but save at beginning & restore at end
Rfact Stack Setup

Entering Stack

rfact:
  pushl %ebp
  movl %esp,%ebp
  pushl %ebx
int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1) ;
    return rval * x;
}

movl 8(%ebp),%ebx  # ebx = x
cmp $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:

• Temporary value of x-1
• Returned value from rfact(x-1)
• Returned value from this call

%ebx  Stored value of x
%eax
Rfact Recursion

leal -1(%ebx),%eax

x
Rtn adr
Old %ebp
Old %ebx

%eax x-1
%ebx x

pushl %eax

x
Rtn adr
Old %ebp
Old %ebx
x-1

%eax x-1
%ebx x

Rtn adr

%ebp
%esp

leal -1(%ebx),%eax

%eax x-1
%ebx x

Rtn adr

%ebp
%esp

call rfact

%eax x-1
%ebx x

Rtn adr

%ebp
%esp
Return from Call

Assume that \texttt{rfact}(x-1) returns \((x-1)!\) in register %eax
Rfact Completion

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

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

Calling `s_helper` from `sfact`

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

Stack at time of call:

```
  8       x
  4       Rtn adr
  0       Old %ebp
-4       val = x!
-8       Unused
-12      &val
-16      x
```

- 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

- 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

- 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 pointer to accum
  - Use access (%edx) to reference memory
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