15-213
“The course that gives CMU its Zip!”

Machine-Level Programming III: Procedures
Sept. 21, 2004

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 "Bottom"

Increasing Addresses

Stack Grows Down

Stack "Top"

Stack Pointer `%esp`

+4
Stack Operation Examples

pushl %eax

popl %edx

0x110
0x10c
0x108
0x104

%esp
%eax
%edx

%eax 213
%edx 555
%esp 0x108

%eax 213
%edx 555
%esp 0x104

%eax 213
%edx 213
%esp 0x108

15-213, F’04
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 Example

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

call 8048b90

%esp 0x108
0x108 123
0x10c
0x110

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

%eip 0x804854e
%eip is program counter

%eip 0x8048b90
Procedure Return Example

8048591: c3 ret

%esp 0x104 %esp 0x108
%eip 0x8048591 %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();
  
}
```

- Procedure amI recursive

Call Chain

```
yoo
  number
  amI
  amI

who
  amI

amI
```

15-213, F’04
## Stack Frames

### Contents
- Local variables
- Return information
- Temporary space

### Management
- Space allocated when enter procedure
  - “Set-up” code
- Dealocated when return
  - “Finish” code

### Pointers
- Stack pointer `%esp` indicates stack top
- Frame pointer `%ebp` indicates start of current frame
Stack Operation

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

Call Chain

- `yoo`

Frame Pointer `%ebp`

Stack Pointer `%esp`

15-213, F’04
Stack Operation

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

Call Chain

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

-1 3– 15-213, F’04
Stack Operation

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

Call Chain

- `yoo`
- `who`
- `amI`

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

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

Call Chain

- `amI` pointer
- `%ebp` frame pointer
- `%esp` stack pointer

Frame Pointer

Stack Pointer
Stack Operation

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

Call Chain

- `yoo`
- `who`
- `amI`
- `amI`
- `amI`
- `amI`
- `amI`

Frame Pointer: `%ebp`

Stack Pointer: `%esp`
Stack Operation

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

Call Chain

- yoo
- who
- amI
- amI
- amI
- amI

Frames:
- Frame Pointer: `%ebp`
- Stack Pointer: `%esp`
Stack Operation

Call Chain

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

Frame Pointer %ebp
Stack Pointer %esp
Stack Operation

誰(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

– 19 –
Stack Operation

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

Call Chain

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

- amI
- who
- yoo

- 15-213, F’04
Stack Operation

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

Call Chain

- Frame Pointer: \%ebp
- Stack Pointer: \%esp

Yoo

Who

AmI

AmI

AmI
Stack Operation

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

Call Chain

```
yoo  
  who
  amI amI
    amI
```

Stack Pointer  %esp

Frame Pointer  %ebp

yoo

---
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

%esp
&zip2
&zip1
Rtn adr
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 %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
swap Setup #1

Entering Stack

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

Resulting Stack

\[
\begin{align*}
\text{\%ebp} & \\
\text{\&zip2} & \quad \text{\%esp} \\
\text{\&zip1} & \\
\text{Rtn adr} & \\
\text{Old \%ebp} & \\
\text{YP} & \\
\text{xp} & \\
\text{Rtn adr} & 
\end{align*}
\]

swap:

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

Entering Stack

\[
\begin{align*}
\text{\&zip2} \\
\text{\&zipl} \\
\text{Rtn adr}
\end{align*}
\]

\%ebp

\%esp

Resulting Stack

\[
\begin{align*}
\text{YP} \\
\text{xp} \\
\text{Rtn adr}
\end{align*}
\]

Old \%ebp

\%ebp

\%esp

---

swap:

- pushl \%ebp
- movl \%esp,\%ebp
- pushl \%ebx

---
swap Setup #3

Entering Stack

\[
\begin{align*}
\text{\textbf{\&zip2}} \\
\text{\textbf{\&zipl}} \\
\text{Rtn adr} \\
\end{align*}
\]

\%ebp
\%esp

Resulting Stack

\[
\begin{align*}
\text{\textbf{YP}} \\
\text{\textbf{xp}} \\
\text{Rtn adr} \\
\text{Old \%ebp} \\
\text{Old \%ebx} \\
\end{align*}
\]

\%ebp
\%esp

swap:

\begin{verbatim}
pushl \%ebp
movl \%esp,\%ebp
pushl \%ebx
\end{verbatim}
Effect of swap Setup

Entering Stack

Resulting Stack

Offset
(relative to %ebp)

• • •

%ebp

• • •

%ebp

&zip2 12

&zip1 8

Rtn adr 4

Rtn adr

Old %ebp 0

Old %ebx

movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx # get xp

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>Rtn adr</th>
<th>Old %ebp</th>
<th>Old %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</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>
</tr>
<tr>
<td>4</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>
</tr>
<tr>
<td>-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

swap’s Stack

<table>
<thead>
<tr>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>Rtn adr</th>
<th>Old %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program:

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
swap Finish #3

swap’s Stack

Offset
12
8
4
0

wap's Stack

Offset
12
8
4

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

```assembly
yoo:
  ...
  movl $15213, %edx
  call who
  addl %edx, %eax
  ...
  ret

who:
  ...
  movl 8(%ebp), %edx
  addl $91125, %edx
  ...
  ret
```

- Contents of register `%edx` overwritten by `who`
Register Saving Conventions

When procedure \texttt{you} calls \texttt{who}:
- \texttt{you} is the \textit{caller}, \texttt{who} is the \textit{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

```
globl rfact
.type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
.align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```
Rfact Stack Setup

Entering Stack

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

%ebp
%esp
x
Rtn adr

%ebp
%esp
pre %ebp
pre %ebx

0
-4
4
8

Caller
Callee

Caller

Rtn adr
Old %ebp
Old %ebx

4
-4

-3 8 – 15 – 213, F’04
int rfact(int x)
{
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1) ;
    return rval * x;
}

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

pushl %eax

call rfact

Rtn adr

%x

Old %ebp
Old %ebx

%x-1

x

Rtn adr

%x

Old %ebp
Old %ebx

%x-1

%x

Rtn adr

%x

Old %ebp
Old %ebx

%-1

%x

Rtn adr

%x

Old %ebp
Old %ebx

%-1

%x

Rtn adr

%-1

%x

Rtn adr

%-1

%x
Rfact Result

Return from Call

Assume that \( rfact(x-1) \) 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 \textit{sfact}

\begin{verbatim}
.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
\end{verbatim}

Using Stack for Local Variable

\begin{itemize}
  \item Variable \texttt{val} must be stored on stack
    \begin{itemize}
      \item Need to create pointer to it
    \end{itemize}
  \item Compute pointer as $-4(\%ebp)$
  \item Push on stack as second argument
\end{itemize}

\begin{verbatim}
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
\end{verbatim}
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

```
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
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