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

Machine-Level Programming III: Procedures
Jan. 31, 2008

IA32
- stack discipline
- Register saving conventions
- Creating pointers to local variables

x86-64
- Argument passing in registers
- Minimizing stack usage
- Using stack pointer as only reference

class06.ppt
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 Grows Down

Stack "Top"
IA32 Stack Pushing

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`

**IA32 Stack Popping**

- Stack “Top”
- Stack “Bottom”
- Stack Grows Down
- Increasing Addresses

Stack Pointer `%esp`
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

%esp  0x108
%eip  0x804854e

%esp  0x104
%eip  0x8048b90

%eip  is program counter
Procedure Return Example

8048591: c3 ret

%esp %eip
0x104 0x8048591
0x108 123
0x10c 0x8048553
0x110

%esp %eip
0x104 0x8048591
0x108 123
0x10c 0x8048553
0x110

%esp
0x104
%eip
0x8048591
%esp
0x108
%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

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

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

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

- Procedure `amI` recursive

Call Chain

```
yoo
  
who
  
  amI  amI
    
    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
Stack Operation

Call Chain

```c
yoo (...) {
    •
    •
    who();
    •
    •
}
```
Stack Operation

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

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

yoo
who

yoo
who
Stack Operation

Call Chain

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

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

Call Chain

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

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

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

Call Chain

```
yoo
  who
  amI
  amI
  amI
```

Frame Pointer

```
%ebp
```

Stack Pointer

```
%esp
```
Stack Operation

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

Call Chain

Frame Pointer %ebp
Stack Pointer %esp

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

```
void amI(...) {
    // Stack frame
    amI();
    // Call chain
    amI();
    amI();
}
```

Call Chain

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

Diagram:

- Stack frame
- Call chain
- Frame Pointer
- Stack Pointer
Stack Operation

Call Chain

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

Frame Pointer
%ebp

Stack Pointer
%esp

yoo

who

amI

amI

amI

amI
Stack Operation

Call Chain

```
amI(...) {
  
  
  
}
```
Stack Operation

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

Call Chain

yoo  
→  
who  
→  
amI  
→  
amI

Frame Pointer  
%ebp

Stack Pointer  
%esp

yoo

who
Stack Operation

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

  
}
```

Call Chain

```
yoo

who

amI

amI

amI

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

int zip1 = 15213;
int zip2 = 91125;

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

call_swap:
    ...
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
    ...

Resulting Stack

%esp
&zip2
&zip1
Rtn adr
Revisiting swap

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

Set Up

Body

Finish
swap Setup #1

Entering Stack

Resulting Stack

swap:

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

%ebp

%esp

%ebp

%esp

&zip2

&zipl

Rtn adr

Rtn adr

Old %ebp

YP
xp

•

•

•

•

•

•
swap Setup #2

Entering Stack

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

Resulting Stack

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

swap:
\[
\begin{align*}
pushl \ \%ebp \\
movl \ \%esp,\%ebp \\
pushl \ \%ebx
\end{align*}
\]
**swap Setup #3**

**Entering Stack**

- \%ebp
- \&zip2
- \&zip1
- Rtn adr

**Resulting Stack**

- YP
- xp
- Rtn adr
- Old \%ebp
- Old \%ebx

**swap:**

```assembly
pushl \%ebp
movl \%esp,\%ebp
pushl \%ebx
```
**Effect of swap Setup**

**Entering Stack**

- &zip2
- &zipl
- Rtn adr

**Resulting Stack**

- Old %ebp
- Old %ebx

**Offset (relative to %ebp)**

- &zip2: 12
- &zipl: 8
- Rtn adr: 4
- %esp: 0

**Body**

```plaintext
movl 12(%ebp),%ecx  # get yp
movl 8(%ebp),%edx   # get xp
. . .
```
### swap Finish #1

**swap’s Stack**

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

**Observation**

- Saved & restored register `%ebx`

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

swap’s Stack

Offset
12
8
4
0
-4

yp
xp
Rtn adr
Old %ebp
Old %ebx

%ebp
%esp

swap’s Stack

Offset
12
8
4
0

yp
xp
Rtn adr
Old %ebp

%ebp
%esp

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

swap’s Stack

Offset
12
8
4
0

Rtn adr
Old %ebp
%ebp
%esp

swap’s Stack

Offset
12
8
4

Rtn adr

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

Observation

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

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

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

Can Register be Used for Temporary Storage?

**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{yoo} calls \texttt{who}:
\begin{itemize}
\item \texttt{yoo} is the \textit{caller}, \texttt{who} is the \textit{callee}
\end{itemize}

Can Register be Used for Temporary Storage?

Conventions
\begin{itemize}
\item “Caller Save”
  \begin{itemize}
  \item Caller saves temporary in its frame before calling
  \end{itemize}
\item “Callee Save”
  \begin{itemize}
  \item Callee saves temporary in its frame before using
  \end{itemize}
\end{itemize}
IA32/Linux Register Usage

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

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

pre %ebp
pre %ebx
x
Rtn adr

%ebp
%esp

Caller

pre %ebp
pre %ebx
x
Rtn adr

%ebp
%esp

Old %ebp
Old %ebx

Callee

0
4
8

-4
4
8

15-213, S’08
int rfact(int x) {
    int rval;
    if (x <= 1)
        return 1;
    rval = rfact(x-1) ;
    return rval * x;
}

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

%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

%eax  x-1
%ebx  x
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;
}
Creating & Initializing Pointer

Initial part of sfact

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

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
Passing Pointer

Calling \texttt{s\_helper} from \texttt{sfact}

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

\begin{verbatim}
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
\end{verbatim}

Stack at time of call:

- 8: x
- 4: Rtn adr
- 0: Old %ebp
- 4: val = x!
- -4: Unused
- -8: %ebp
- -12: &val
- -16: x
- -20: %esp

\[\text{leal } -4(\%ebp),\%eax \quad \text{# Compute } \&\text{val}\]
\[\text{pushl } \%eax \quad \text{# Push on stack}\]
\[\text{pushl } \%edx \quad \text{# Push } x\]
\[\text{call } s\_\text{helper} \quad \text{# call}\]
\[\text{movl } -4(\%ebp),\%eax \quad \text{# Return } \text{val}\]
\[\ldots \text{# Finish}\]
Using Pointer

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

- **Register** `%ecx` holds `x`
- **Register** `%edx` holds `accum`
  - Assume memory initially has value `V`
  - Use access `%edx` to reference memory

```assembly
  movl %ecx,%eax  # z = x
  imull (%edx),%eax # z *= *accum
  movl %eax,(%edx) # *accum = z
  ...
```
IA 32 Procedure 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
### x86-64 General Purpose Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Twice the number of registers
- Accessible as 8, 16, 32, or 64 bits
### x86-64 Register Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>Return Value</td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td>Callee Saved</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>Argument #4</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>Argument #3</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>Argument #2</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>Argument #1</td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td>Stack Pointer</td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td>Callee Saved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%r8</code></td>
<td>Argument #5</td>
</tr>
<tr>
<td><code>%r9</code></td>
<td>Argument #6</td>
</tr>
<tr>
<td><code>%r10</code></td>
<td>Callee Saved</td>
</tr>
<tr>
<td><code>%r11</code></td>
<td>Used for linking</td>
</tr>
<tr>
<td><code>%r12</code></td>
<td>C: Callee Saved</td>
</tr>
<tr>
<td><code>%r13</code></td>
<td>Callee Saved</td>
</tr>
<tr>
<td><code>%r14</code></td>
<td>Callee Saved</td>
</tr>
<tr>
<td><code>%r15</code></td>
<td>Callee Saved</td>
</tr>
</tbody>
</table>
x86-64 Registers

Arguments passed to functions via registers
- If more than 6 integral parameters, then pass rest on stack
- These registers can be used as caller-saved as well

All References to Stack Frame via Stack Pointer
- Eliminates need to update %ebp

Other Registers
- 6+1 callee saved
- 2 or 3 have special uses
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}

swap:
movq (%rdi), %rdx
movq (%rsi), %rax
movq %rax, (%rdi)
movq %rdx, (%rsi)
ret

- Operands passed in registers
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- No stack operations required

Avoiding Stack
- Can hold all local information in registers
x86-64 Locals in the Red Zone

Avoiding Stack Pointer Change

- Can hold all information within small window beyond stack pointer

```c
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```assembly
swap_a:
    movq (%rdi), %rax
    movq %rax, -24(%rsp)
    movq (%rsi), %rax
    movq %rax, -16(%rsp)
    movq -16(%rsp), %rax
    movq %rax, (%rdi)
    movq -24(%rsp), %rax
    movq %rax, (%rsi)
    ret
```

```
rtn Ptr
unused
loc[1]
loc[0]
```

%rsp
long scount = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_se
  (long a[], int i)
{
  swap(&a[i], &a[i+1]);
  scount++;
}

swap_ele_se:
  movslq %esi,%rsi          # Sign extend i
  leaq  (%rdi,%rsi,8), %rdi # &a[i]
  leaq   8(%rdi), %rsi      # &a[i+1]
  call  swap     # swap()
  incq scount(%rip)        # scount++;
 ret
When swap executes `ret`, it will return from `swap_ele`

Possible since `swap` is a “tail call”

```c
long scount = 0;
/* Swap a[i] & a[i+1] */
void swap_ele
  (long a[], int i)
{
  swap(&a[i], &a[i+1]);
}

swap_ele:
  movslq %esi,%rsi            # Sign extend i
  leaq (%rdi,%rsi,8), %rdi   # &a[i]
  leaq 8(%rdi), %rsi        # &a[i+1]
  jmp   swap                # swap()
```
x86-64 Stack Frame Example

long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
  (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += a[i];
}

- Keeps values of a and i in callee save registers
- Must set up stack frame to save these registers
Understanding x86-64 Stack Frame

swap_ele_su:

- movq %rbx, -16(%rsp)  # Save %rbx
- movslq %esi, %rbx    # Extend & save i
- movq %r12, -8(%rsp)  # Save %r12
- movq %rdi, %r12      # Save a
- leaq (%rdi, %rbx, 8), %rdi  # &a[i]
- subq $16, %rsp       # Allocate stack frame
- leaq 8(%rdi), %rsi  # &a[i+1]
- call swap           # swap()
- movq (%r12, %rbx, 8), %rax  # a[i]
- addq %rax, sum(%rip) # sum += a[i]
- movq (%rsp), %rbx    # Restore %rbx
- movq 8(%rsp), %r12  # Restore %r12
- addq $16, %rsp      # Deallocate stack frame
- ret
Stack Operations

movq %rbx, -16(%rsp)  # Save %rbx
movq %r12, -8(%rsp)   # Save %r12
subq $16, %rsp       # Allocate stack frame

movq (%rsp), %rbx    # Restore %rbx
movq 8(%rsp), %r12   # Restore %r12
addq $16, %rsp       # Deallocate stack frame
Interesting Features of Stack Frame

Allocate Entire Frame at Once
- All stack accesses can be relative to `%rsp`
- Do by decrementing stack pointer
- Can delay allocation, since safe to temporarily use red zone

Simple Deallocation
- Increment stack pointer
x86-64 Procedure Summary

Heavy Use of Registers
- Parameter passing
- More temporaries

Minimal Use of Stack
- Sometimes none
- Allocate/deallocate entire block

Many Tricky Optimizations
- What kind of stack frame to use
- Calling with jump
- Various allocation techniques