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

Machine-Level Programming III:
Procedures
Sept. 11, 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
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

- \texttt{pushl } \textit{Src}
- Fetch operand at \textit{Src}
- Decrement \%\text{esp} by 4
- Write operand at address given by \%\text{esp}

Stack "Top"

Stack "Bottom"

Stack Grows Down

Stack "Top"

Stack "Bottom"

Increasing Addresses

Stack Pointer \%\text{esp}

-4

-3-
**IA32 Stack Popping**

**Popping**
- `popl Dest`
- Read operand at address given by `%esp`
- Increment `%esp` by 4
- Write to `Dest`
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 %esp 0x110
%esp 0x10c %esp 0x110
%esp 0x108 123 %esp 0x10c 123
%esp 0x104 0x8048553

%eip 0x804854e %eip 0x8048b90

%eip is program counter
# Procedure Return Example

8048591: c3  

\[
\begin{array}{c}
0x110 \\
0x10c \\
0x108 \\
0x104
\end{array}
\quad
\begin{array}{c}
0x110 \\
0x10c \\
0x108 \\
0x104
\end{array}
\]

\[
\begin{array}{c}
0x8048553 \\
123 \\
0x8048553
\end{array}
\quad
\begin{array}{c}
0x8048553 \\
123 \\
0x8048553
\end{array}
\]

%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();
  
}
```

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

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

- **Procedure amI recursive**

Call Chain

- Procedure amI recursive
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

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

Call Chain

Stack Pointer
%esp

Frame Pointer
%ebp

who

yoo

who
Stack Operation

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

Call Chain

- Frame Pointer %ebp
- Stack Pointer %esp

```
Call Chain:
- yoo
- who
- amI
```

- 13 – 15-213, F’08
Stack Operation

Call Chain

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

Frame Pointer %ebp
Stack Pointer %esp
Stack Operation

```
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

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

Call Chain

- **Frame Pointer** %ebp
- **Stack Pointer** %esp

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

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

Call Chain

yoo

who

amI

amI

Frame Pointer
%ebp

Stack Pointer
%esp
Stack Operation

Call Chain

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

Frame Pointer
%ebp

Stack Pointer
%esp

yoo
who
amI
amI
Stack Operation

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

Call Chain

![Call Chain Diagram](attachment:call-chain-diagram.png)
Stack Operation

```c
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

Diagram:
- Frame Pointer (%ebp)
- Stack Pointer (%esp)
- Caller Frame
- Arguments
  - Return Address
  - Old %ebp
- Saved Registers + Local Variables
- Argument Build

- 22 -
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:

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

Resulting Stack

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

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

```assembly
call swap

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

%ebp

%esp

%esp

%esp

%esp

&zip2

&zip1

Rtn adr

Rtn adr

Old %ebp

YP

xp

Rtn adr
swap Setup #2

Entering Stack

Resulting Stack

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

Entering Stack

Resulting Stack

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

%ebp

%esp

&zip2
&zip1
Rtn adr

%ebp

%esp

Yp
Xp
Rtn adr

Old %ebp
Old %ebx

%ebp

%esp
Effect of swap Setup

Entering Stack

Resulting Stack

Offset (relative to %ebp)

12
8
4
0

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

Body
swap Finish #1

swap’s Stack

Offset
12 yp
8 xp
4 Rtn adr
0 Old %ebp
-4 Old %ebx

Observation
- Saved & restored register %ebx

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

swap’s Stack

 Offset  yp  xp  Rtn adr  Old %ebp  Old %ebx
 12     .    .    .       .      .
  8     .    .    .       .      .
  4     .    .    .       .      .
  0     .    .    .       .      .
 -4    .    .    .       .      .

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

swap’s Stack

Offset
12
8
4
0

Old %ebp

%ebp

%esp

Rtn adr

xp

yp

swap’s Stack

Offset
12
8
4

%ebp

%esp

Rtn adr

xp

yp

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
Disassembled swap

080483a4 <swap>:
  80483a4:  55          push %ebp
  80483a5:  89 e5       mov %esp,%ebp
  80483a7:  53          push %ebx
  80483a8:  8b 55 08    mov 0x8(%ebp),%edx
  80483ab:  8b 4d 0c    mov 0xc(%ebp),%ecx
  80483ae:  8b 1a       mov (%edx),%ebx
  80483b0:  8b 01       mov (%ecx),%eax
  80483b2:  89 02       mov %eax,(%edx)
  80483b4:  89 19       mov %ebx,(%ecx)
  80483b6:  5b          pop %ebx
  80483b7:  c9          leave
  80483b8:  c3          ret

Calling Code

  8048409:  e8 96 ff ff ff           call 80483a4 <swap>
  804840e:  8b 45 f8               mov 0xfffffffff8(%ebp),%eax
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 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


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

Recursive Factorial

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:

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

```asm
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 \texttt{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

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

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

Calling `s_helper` from `sfact`

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

Stack at time of call:

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

- Twice the number of registers
- Accessible as 8, 16, 32, or 64 bits

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>
## x86-64 Register Conventions

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

### Argument #5
- %r8  - Argument #5
- %r9  - Argument #6
- %r10 - Callee Saved
- %r11 - Used for linking
- %r12 - C: Callee Saved
- %r13 - Callee Saved
- %r14 - Callee Saved
- %r15 - Callee Saved
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
x86-64 Long Swap

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
  %rsp
  -8    unused
  -16   loc[1]
  -24   loc[0]
```
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

- No values held while swap being invoked
- No callee save registers needed
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]);
}

void 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

Keeps values of a and i in callee save registers
Must set up stack frame to save these registers

```c
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];
}
```

```assembly
swap_ele_su:
  movq %rbx, -16(%rsp)
  movslq %esi,%rbx
  movq %r12, -8(%rsp)
  movq %rdi, %r12
  leaq (%rdi,%rbx,8), %rdi
  subq $16, %rsp
  leaq 8(%rdi), %rsi
  call swap
  movq (%r12,%rbx,8), %rax
  addq %rax, sum(%rip)
  movq (%rsp), %rbx
  movq 8(%rsp), %r12
  addq $16, %rsp
  ret
```
Understanding x86-64 Stack Frame

swap_ele_su:

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

\[
\begin{align*}
\text{movq} & \quad \%rbx, -16(\%rsp) \quad \# \text{Save } \%rbx \\
\text{movq} & \quad \%r12, -8(\%rsp) \quad \# \text{Save } \%r12 \\
\text{subq} & \quad 16, \%rsp \quad \# \text{Allocate stack frame} \\
\text{movq} & \quad (\%rsp), \%rbx \quad \# \text{Restore } \%rbx \\
\text{movq} & \quad 8(\%rsp), \%r12 \quad \# \text{Restore } \%r12 \\
\text{addq} & \quad 16, \%rsp \quad \# \text{Deallocation stack frame}
\end{align*}
\]
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