15-213

Machine-Level Programming III: Procedures
Sept. 17, 2007

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

- `pushl Src`
- Fetch operand at `Src`
- Decrement `%esp` by 4
- Write operand at address given by `%esp`

Stack Grows Down

Stack "Bottom"

Increasing Addresses

Stack "Top"

Stack Pointer `%esp`
IA32 Stack Popping

Popping
- `popl Dest`
- Read operand at address given by `%esp`
- Increment `%esp` by 4
- Write to `Dest`

Stack Pointer `%esp`

Stack "Top" +4

Stack Grows Down

Increasing Addresses

Stack "Bottom"
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:**

```plaintext
    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

call  8048b90

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>123</td>
</tr>
<tr>
<td>0x104</td>
<td>0x8048553</td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x804854e</td>
<td></td>
</tr>
<tr>
<td>0x8048b90</td>
<td></td>
</tr>
</tbody>
</table>

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

%eip is program counter
Procedure Return Example

8048591: c3  ret

%esp 0x104  0x8048553
%eip 0x8048591

%esp 0x104  0x100
0x100  123
0x108  0x8048553
0x110  0x8048591

ret

%esp 0x108  0x100
0x100  123
0x108  0x8048553
0x110  0x8048591

%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();
    . . .
    amI();
}
```

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

- Procedure `amI` recursive

Call Chain

```
yoo
  who
    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

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

Frame Pointer %ebp
Stack Pointer %esp

yoo
Stack Operation

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

Call Chain

Stack Operation

- Frame Pointer
  - ebp
- Stack Pointer
  - esp

- yoo
  - who

- yoo
  - who
Stack Operation

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

Call Chain

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

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

Call Chain

```c
void amI(...) {
    // Code...
    amI();
    // Code...
}
```

Frame Pointer %ebp
Stack Pointer %esp
Stack Operation

amI(...) {
  ●
  ●
  amI();
  ●
}

Call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

yoo
who
amI
amI
amI
amI
amI
Stack Operation

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

Call Chain

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

```
  .
  .
  yoo
  who
  amI
  amI
  amI
```
Stack Operation

Call Chain

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

Frame Pointer %ebp

Stack Pointer %esp

yoo

who
Stack Operation

Call Chain

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

Frame Pointer
%ebp

Stack Pointer
%esp

yoo

who

amI

amI

amI
Stack Operation

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

Call Chain

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

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

call Chain

Frame Pointer
%ebp

Stack Pointer
%esp

yoo

who

amI amI

amI

amI
Stack Operation

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

Call Chain

```
<table>
<thead>
<tr>
<th>Frame Pointer</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Pointer</td>
<td>%esp</td>
</tr>
</tbody>
</table>
```

- `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:

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

Resulting Stack

```
Rtn adr
&zip1
&zip2

%esp
```
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

- `%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 #2**

**Entering Stack**

- `&zip2`
- `&zip1`
- `Rtn adr`

- `%ebp` (current stack pointer)

**Resulting Stack**

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

- `%ebp` (old stack pointer)
- `%esp` (new stack pointer)

**swap:**

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

Entering Stack

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

Resulting Stack

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

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

Entering Stack

Resulting Stack

\[
\begin{align*}
\text{Offset} &\quad (\text{relative to} \ %\text{ebp}) \\
&\quad \%\text{ebp} \\
&\quad \%\text{esp} \\
&\quad \%\text{ebx} \\
&\quad \%\text{ebp} \\
\end{align*}
\]

\[
\begin{align*}
\&\text{zip}2 &\quad 12 \\
\&\text{zip}1 &\quad 8 \\
\text{Rtn adr} &\quad 4 \\
\end{align*}
\]

\[
\begin{align*}
\&\text{zip}2 &\quad \text{yp} \\
\&\text{zip}1 &\quad \text{xp} \\
\text{Rtn adr} &\quad \%\text{ebp} \\
\%\text{ebp} &\quad \%\text{esp} \\
\%\text{ebp} &\quad \%\text{esp} \\
\end{align*}
\]

Body

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

- Saved & restored register %ebx
swap Finish #2

swap’s Stack

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

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

swap’s Stack

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

%ebp
%esp
swap Finish #3

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

swap’s Stack

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

swap’s Stack

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

%ebp
%esp
%ebp
%esp

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
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{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
Rfact Stack Setup

Entering Stack

```
rfact:
pushl %ebp
movl %esp,%ebp
pushl %ebx
```
Recursion

Registers

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:
Rfact Recursion

`leal -1(%ebx),%eax`

```
x
Rtn adr
Old %ebp
Old %ebx
```

```
x
Rtn adr
Old %ebp
Old %ebx
```

```
x-1
```

```
%eax
%ebx
```

```
x-1
```

```
%eax
%ebx
```

```
x
Rtn adr
Old %ebp
Old %ebx
```

```
x
Rtn adr
Old %ebp
Old %ebx
```

```
x-1
```

```
%eax
%ebx
```

```
x-1
```

```
%eax
%ebx
```

```
x
Rtn adr
Old %ebp
Old %ebx
```

```
x
Rtn adr
Old %ebp
Old %ebx
```

```
x-1
```

```
%eax
%ebx
```

```
x-1
```

```
%eax
%ebx
```

```
x
Rtn adr
Old %ebp
Old %ebx
```

```
x
Rtn adr
Old %ebp
Old %ebx
```

```
x-1
```

```
%eax
%ebx
```

```
x-1
```

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

\[
\begin{align*}
%eax & \quad (x-1)! \\
%ebx & \quad x
\end{align*}
\]
Rfact Completion

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
recursive procedure
void s_helper (int x, int *accum)
{
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper (x-1, accum);
    }
}

Top-Level Call

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

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

Stack at time of call:

- `x`
- `val = x!`
- Unused
- `&val`
- `%ebp`
- `%esp`

Machine code:

```assembly
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

```assembly
    movl %ecx,%eax  # z = x
    imull (%edx),%eax  # z *= *accum
    movl %eax,(%edx)  # *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 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
# 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
x86-64 Long Swap

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

- 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

```assembly
swa:
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    ret
```
x86-64 Locals in the Red Zone

/* 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];
}

Avoiding Stack Pointer Change

- Can hold all information within small window beyond stack pointer

swap_a:
    movq (%rdi), %rax
    movq %rax, -24(%rsp)
    movq (%rsi), %rax
    movq %rax, -16(%rsp)
    movq -16(%rsp), %rax
    movq %rax, (%edi)
    movq -24(%rsp), %rax
    movq %rax, (%esi)
    ret
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
x86-64 Call using Jump

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

- When swap executes `ret`, it will return from `swap_ele`
- Possible since swap is a “tail call”

```
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
```

```
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:

```assembly
    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

```assembly
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