Machine-Level Programming III: Procedures

Kai Shen

Outline: Procedures
- Stack Structure & Calling Conventions
- Pointers
- X86-64 Procedures

IA32 Stack
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %esp contains lowest stack address
  - address of “top” element

IA32 Stack: Push
- pushl $src
  - Fetch operand at $src
  - Decrement %esp by 4
  - Write operand at address given by %esp

Stack Pointer: %esp
Stack “Bottom”
Increasing Addresses
Stack Grows Down
Stack “Top”

Stack Pointer: %esp
Stack “Bottom”
Increasing Addresses
Stack Grows Down
Stack “Top”
Stack

Grows Down

Increasing Addresses

IA32 Stack: Pop

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:
    - Address of the next instruction right after 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

Procedure Return Example
Stack Discipline

- 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
    - Why not registers (not enough registers, addressed variables)
- Stack allocated in Frames
  - state for single procedure instantiation
- Stack discipline
  - One active procedure instantiation at a time (for a thread of execution)
  - Last called returns first
  - State for given procedure needed from its call to its return

Call Chain Example

Example Call Chain

Procedure ami() is recursive

Stack Frames

- Contents
  - Arguments
  - Local variables
  - Temporary space
- Management
  - Space allocated when enter procedure
    - “Set-up” code
  - Deallocated when return
    - “Finish” code

Example
Example

Stack

Example

Stack
Example

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

Stack

```c
%ebp
%esp
Stack
```
Example 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 8(%ebp), %edx
movl 12(%ebp), %ecx
movl (%edx), %ebx
movl (%ecx), %eax
movl %eax, (%edx)
movl %ebx, (%ecx)
popl %ebx
popl %ebp
ret
```

swap Setup #1

**Entering Stack**
```
*  
*  
*  
%course2  
%course1  
Rtn adr  
```

**Resulting Stack**
```
%ebp  
%esp  
```

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

```

swap Setup #2

**Entering Stack**
```
*  
*  
*  
%course2  
%course1  
Rtn adr  
```

**Resulting Stack**
```
%ebp  
%esp  
```

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

```

swap Setup #3

**Entering Stack**
```
*  
*  
*  
%course2  
%course1  
Rtn adr  
```

**Resulting Stack**
```
%ebp  
%esp  
```

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

```
**swap Body**

Entering Stack

- %ebp
- %ecx
- %edx
- %ebx
- %esp
- %ebp
- %esp

Resulting Stack

- %ebp
- %ecx
- %edx
- %ebp
- %esp

Offset relative to %ebp

\[
\begin{align*}
\text{Offset} & = 12 \\
\text{Offset} & = 8 \\
\text{Offset} & = 4 \\
\end{align*}
\]

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

**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 $18243, %edx  
  - ...  
  - ret  

- Contents of register %edx overwritten by **who**
- This could be trouble — something should be done!
  - Need some coordination

**Observation**
- Saved and restored register %ebx
- Not so for %eax, %ecx, %edx

**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 values in its frame before the call
  - "**Callee Save**"
    - Callee saves temporary values in its frame before using
IA32/Linux+Windows Register Usage

- %eax, %edx, %ecx
  - caller saves prior to call if values are used later
- %eax
  - also used to return integer value
- %ebx, %esi, %edi
  - callee saves if wants to use them
- %esp, %ebp
  - special form of callee save
  - restored to original values upon exit from procedure

Outline: Procedures

- Stack Structure & Calling Conventions
- Pointers
- X86-64 Procedures

Pointer Code

Generating Pointer

```c
/* Compute x + 3 */
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

Referencing Pointer

```c
/* Increment value by k */
void incrk(int *ip, int k) {
    *ip += k;
}
```

Creating and Initializing Local Variable

```c
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

First part of add3

```c
add3:
    pushl %ebp
    movl %esp, %ebp # Alloc. 24 bytes
    subl $24, %esp
    movl %eax, -4(%ebp) # Set localx to x
```

```
<table>
<thead>
<tr>
<th>x</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rtn addr</td>
<td>%ebp</td>
</tr>
<tr>
<td>Old %ebp</td>
<td>%ebp</td>
</tr>
</tbody>
</table>
```
Creating Pointer as Argument

```c
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

Middle part of add3

```
movl $3, 4(%esp)  # 2nd arg = 3
leal -4(%ebp), %eax  # &localx
movl %eax, (%esp)  # 1st arg = &localx
```

call incrk

```
movl -4(%ebp), %eax  # Return val = localx
```

Retrieving local variable

```c
int add3(int x) {
    int localx = x;
    incrk(&localx, 3);
    return localx;
}
```

Final part of add3

```
movl -4(%ebp), %eax  # Return value = localx
leave
```

Outline: Procedures

- Stack Structure & Calling Conventions
- Pointers
- X86-64 Procedures
### x86-64 Registers

- **%rax**
- **%rbx**
- **%rcx**
- **%rdx**
- **%rsi**
- **%rdi**
- **%rsp**
- **%rbp**

#### Notes
- Twice the number of registers, accessible as 8, 16, 32, 64 bits

### x86-64 Registers: Usage Conventions

- **%rax**
  - Return value
- **%rbx**
  - Callee saved
- **%rcx**
  - Argument #4
- **%rdx**
  - Argument #3
- **%rsi**
  - Argument #2
- **%rdi**
  - Argument #1
- **%esp**
  - Stack pointer
- **%rbp**
  - 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

- Other Registers
  - 6 callee saved
  - 2 caller saved
  - 1 return value (also usable as caller saved)
  - 1 special (stack pointer)

- All references to stack frame via stack pointer
  - Eliminates need to update %ebp/%rbp
  - Why did we need %ebp in the first place?

### x86-64 Long Swap

- void swap_l(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 (except ret)
  - Avoiding stack
    - Can hold all local information in registers

- rtn addr
- %rsp
- No stack frame
x86-64 Leaf Locals in the Red Zone

- 128 bytes beyond the stack top to be usable by current function
- Save stack pointer change

```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];
}
```

x86-64 Stack Frame Example

```c
/* Swap a[i] & a[i+1] */
void swap_a(long *xp, long *yp) {
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

Understanding x86-64 Stack Frame

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

x86-64 NonLeaf without Stack Frame

```c
/* Swap a[i] & a[i+1] */
void swap_a(long *xp, long *yp) {
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```
Interesting Features of x86-64 Procedures

- Heavy use of registers
  - Parameter passing, more register-based temporaries
  - Minimal use of stack (sometimes none)
- Got rid of frame pointer
  - All stack accesses must be relative to %rsp
    - Allocate entire frame at once
  - Free one more register for general use, eliminate stack operation on frame pointer save/restore

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.