Machine-Level Programming IV: x86-64 Procedures, Data

Andrew Case

Slides adapted from Jinyang Li, Randy Bryant & Dave O’Hallaron
Today

- Procedures (x86-64)
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
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`/`%rbp`
### x86-64 Integer Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</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>
<tr>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%r10</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
x86-64 Long Swap

```c
void swap_l(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 (except ret)**
  - Can hold all local information in registers

---

```assembly
swap:
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    ret
```
x86-64 Leaf call without Stack Frame

/* 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
- 128 bytes after the %rsp is scratch space for functions (red zone).
- Only good for leaf-functions

```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
```
x86-64 NonLeaf call without Stack Frame

/* 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 8(%rdi,%rsi,8), %rax # &a[i+1]
    leaq (%rdi,%rsi,8), %rdi  # &a[i] (1st arg)
    movq %rax, %rsi          # (2nd arg)
    call swap
    ret

- Data stored in registers instead of stack
# x86-64 Stack Frame Example

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

- Keeps values of &a[i] and &a[i+1] in callee save registers
- Must set up stack frame to save these registers

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

Swap_and_sum:

```
movq   %rbx, -16(%rsp)  # Save %rbx
movq   %rbp, -8(%rsp)   # Save %rbp
subq   $16, %rsp        # Allocate stack frame
movslq %esi,%rax        # Extend i
leaq   8(%rdi,%rax,8), %rbx  # &a[i+1] (callee save)
leaq   (%rdi,%rax,8), %rbp  # &a[i]   (callee save)
movq   %rbx, %rsi        # 2nd argument
movq   %rbp, %rdi        # 1st argument
call   swap              # Get a[i+1]
imulq  (%rbp), %rax      # Multiply by a[i]
addq   %rax, sum(%rip)   # Add to sum
movq   (%rsp), %rbx      # Restore %rbx
movq   8(%rsp), %rbp     # Restore %rbp
addq   $16, %rsp         # Deallocate frame
ret
```
Understanding x86-64 Stack Frame

```
movq %rbx, -16(%rsp)  # Save %rbx
movq %rbp, -8(%rsp)   # Save %rbp

subq $16, %rsp       # Allocate stack frame

movq (%rsp), %rbx    # Restore %rbx
movq 8(%rsp), %rbp   # Restore %rbp
addq $16, %rsp       # Deallocate 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
  - No base/frame pointer needed
x86-64 Procedure Summary

- **Heavy use of registers**
  - Parameter passing
  - More temporaries using registers

- **Minimal use of stack**
  - Sometimes none
  - Allocate/deallocate entire frame at once
  - All stack accesses can be relative to %rsp
  - Usually no base/frame pointer needed
Today

- Procedures (x86-64)
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
Basic Data Types

- **Integral**
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on instructions used

<table>
<thead>
<tr>
<th>Intel</th>
<th>ASM</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>b</td>
<td>1</td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>word</td>
<td>w</td>
<td>2</td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>double word</td>
<td>l</td>
<td>4</td>
<td>[unsigned] int</td>
</tr>
<tr>
<td>quad word</td>
<td>q</td>
<td>8</td>
<td>[unsigned] long int (x86-64)</td>
</tr>
</tbody>
</table>

- **Floating Point**
  - Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>Intel</th>
<th>ASM</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>s</td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td>Double</td>
<td>l</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>Extended</td>
<td>t</td>
<td>10/12/16</td>
<td>long double</td>
</tr>
</tbody>
</table>
Array Allocation

- **Basic Principle**
  
  $T \ A[L];$
  
  - Array of data type $T$ and length $L$
  - Contiguously allocated region of $L \times \text{sizeof}(T)$ bytes

```
char string[12];
```

```
int val[5];
```

```
double a[3];
```

```
char *p[3];
```

IA32

x86-64
## Array Access

### Basic Principle

*Type* $\text{A}[\text{Length}]$;
- Identifier $\text{A}$ can be used as a pointer to array element 0: Type $T^*$
- $x$ is the address of $\text{A}$

```c
int val[5];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{val}[4]$</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>$\text{val}$</td>
<td>int *</td>
<td>$x$</td>
</tr>
<tr>
<td>$\text{val}+1$</td>
<td>int *</td>
<td>$x+4$</td>
</tr>
<tr>
<td>$&amp;\text{val}[2]$</td>
<td>int *</td>
<td>$x+8$</td>
</tr>
<tr>
<td>$\text{val}[5]$</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>$(\text{val}+1)$</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>$\text{val} + i$</td>
<td>int *</td>
<td>$x+4i$</td>
</tr>
</tbody>
</table>
Array Example

```c
#define ZLEN 5
typedef int zip[ZLEN];

zip nyu = { 1, 0, 0, 0, 3 };
zip mit = { 0, 2, 1, 3, 9 };
```

- Declaration “zip nyu” equivalent to “int nyu[5]”
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
Array Accessing Example

int get_digit(zip z, int digit)
{
    return z[digit];
}

IA32

# %edx = z
# %eax = digit
movl (%edx,%eax,4),%eax  # z[digit]

- Store starting address of array in %edx (base)
- Store array index in %eax (index)
- Size of data type in multiplier
- Use memory reference (%edx,%eax,4): results in:
  %edx + 4*%eax
  (base pointer + index*size)
### Array Loop Example (IA32)

```c
void zip_inc(zip z) {
    int i;
    for (i = 0; i < ZIP_LEN; i++)
        z[i]++;
}
```

```assembly
# edx = z
movl $0, %eax        # %eax = i
.L4:
    # loop:
    addl $1, (%edx,%eax,4) # z[i]++
    addl $1, %eax        # i++
    cmpl $5, %eax        # i:5 (ZIP_LEN)
    jne .L4              # if !=, goto loop
```
2D Array: Example

```c
int zips[4][5] =
    {{1, 0, 0, 0, 3},
     {1, 0, 0, 1, 0 },
     {1, 0, 0, 1, 1 },
     {1, 0, 0, 0, 5 }};
```

- 2D array “Row-Major” ordering of all elements
  ```c
  int A[R][C];
  ```
  - `A[i][j] == *(A+i*C+j)`
2D Array Element Access Code

```c
int get_zip_digit(int index, int digit)
{
    return zips[index][digit];
}
```

```asm
movl 8(%ebp), %eax       # index
leal (%eax,%eax,4), %eax # 5*index
addl 12(%ebp), %eax     # 5*index+digit
movl zips(,%eax,4), %eax # offset 4*(5*index+digit)
```

- **Array Elements**
  - `zips[index][digit]` is `int`
  - Address: `zips + 20*index + 4*digit`
Array of pointers: Example

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of int’s

```c
#define UCOUNT 3
int *univ[UCOUNT] = {mit, nyu, cmu};
```

```c
zip nyu = { 1, 0, 0, 0, 3 };
zip mit = { 0, 2, 1, 3, 9 };
zip cmu = { 1, 5, 2, 1, 3};
```
Element Access in Multi-Level Array

```c
int get_univ_digit (int index, int digit)
{
    return univ[index][digit];
}
```

- **Computation (IA32)**
  - Element access `Mem[Mem[univ+4*index]+4*dig]`
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

- **Example Assembly Code**
  - `movl  8(%ebp), %eax  # index`
  - `movl  univ(,%eax,4), %edx  # p = univ[index]`
  - `movl  12(%ebp), %eax  # dig`
  - `movl  (%edx,%eax,4), %eax  # p[digit]`
Array Element Accesses

Nested array

```c
def int get_pgh_digit(int index, int dig)
{
    return pgh[index][dig];
}
```

Multi-level array

```c
def int get_univ_digit(int index, int dig)
{
    return univ[index][dig];
}
```

Accesses looks similar in C, but addresses very different:

```c
Mem[pgh+20*index+4*dig]
Mem[Mem[univ+4*index]+4*dig]
```
Today

- Procedures (x86-64)
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
Structure Allocation

```
struct rec {
    int a[3];
    int i;
    struct rec *n;
};
```

**Concept**

- Contiguously-allocated region of memory
- Offset of each struct member determined at compile time
- Refer to members within structure by names
- Members may be of different types

**Memory Layout**

<table>
<thead>
<tr>
<th>a</th>
<th>i</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
Structure Access

```c
struct rec {
    int a[3];
    int i;
    struct rec *n;
};
```

### Accessing Structure Member
- Pointer indicates first byte of structure
- Access elements with offsets

```c
void set_i(struct rec *r, int val)
{
    r->i = val;
}
```

### IA32 Assembly
```
# %edx = val
# %eax = r
movl %edx, 12(%eax)  # Mem[r+12] = val
```
Following Linked List

```c
void set_all_a_i_values (struct rec *r, int val) {
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->n;
    }
}
```

```
struct rec {
    int a[3];
    int i;
    struct rec *n;
};
```

Element i

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>r</td>
</tr>
<tr>
<td>%ecx</td>
<td>val</td>
</tr>
</tbody>
</table>

.L17:

```asm
# loop:
movl 12(%edx), %eax    # r->i
movl %ecx, (%edx,%eax,4) # r->a[i] = val
movl 16(%edx), %edx    # r = r->n
testl %edx, %edx        # Test r
jne .L17                # If != 0 goto loop
```
Summary

- **Procedures in x86-64**
  - Stack frame is relative to stack pointer
  - Parameters passed in registers

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

- **Structures**
  - Allocation
  - Access