Machine-Level Programming III: Procedures

CSci 211: Machine Architecture and Organization
Lecture #11 February 13th, 2015
Your Instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O’Hallaron, Antonia Zhai

Today
IA 32 Procedures
- Stack Structure
- Calling Conventions
- Illustrations of Recursion & 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

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
        ```assembly
        804854e: e8 3d 06 00 00 call 8048b90 <main>
        8048553: 50 pushl %eax
        8048556: 8b 05 50 00 00
        ```
      - Return address = 0x8048553
    - Procedure return: ret
      - Pop address from stack
      - Jump to address
**Procedure Call Example**

```
0x804854e:  8048b90 <main>  pushl %esp
call  8048b90 <main>
0x8048553:  50               pushl %eax
```

```
0x108 0x10c 0x110
0x104 0x8048553
%esp: 0x108 %esp
%eip: 0x804854e %eip
```

**Procedure Return Example**

```
0x8048591:  c3               ret
```

```
0x108 0x10c 0x110
0x104 0x8048591
%esp: 0x108 %esp
%eip: 0x8048553 %eip
```

**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**

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

```
who(...)
  • • • •
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
IA32/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build."
  - Parameters for function about to call
  - 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 course1 = 15213;
int course2 = 18243;

void call_swap() {
    swap(&course1, &course2);
}
```

Calling `swap` from `call_swap`

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

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

### swap Setup #1

**Entering Stack**

- `%ebp`
- `%esp`
- `%ebx`
- `%eax`
- `%edx`
- `%ecx`
- `%ebp`
- `%esp`

**Resulting Stack**

- `%ebp`
- `%esp`
- `%ebx`
- `%eax`
- `%ecx`
- `%edx`
- `%ebp`
- `%esp`

### swap Setup #2

**Entering Stack**

- `%ebp`
- `%esp`
- `%ebx`
- `%eax`
- `%edx`
- `%ecx`
- `%ebp`
- `%esp`

**Resulting Stack**

- `%ebp`
- `%esp`
- `%ebx`
- `%eax`
- `%ecx`
- `%edx`
- `%ebp`
- `%esp`

### swap Setup #3

**Entering Stack**

- `%ebp`
- `%esp`
- `%ebx`
- `%eax`
- `%edx`
- `%ecx`
- `%ebp`
- `%esp`

**Resulting Stack**

- `%ebp`
- `%esp`
- `%ebx`
- `%eax`
- `%ecx`
- `%edx`
- `%ebp`
- `%esp`

### swap Body

**Entering Stack**

- `%ebp`
- `%esp`
- `%ebx`
- `%eax`
- `%edx`
- `%ecx`
- `%ebp`
- `%esp`

**Resulting Stack**

- `%ebp`
- `%esp`
- `%ebx`
- `%eax`
- `%ecx`
- `%edx`
- `%ebp`
- `%esp`

### swap Finish

**Stack Before Finish**

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

**Resulting Stack**

- `%ebp`
- `%esp`
- `%ebx`
- `%eax`
- `%ecx`
- `%edx`
- `%ebp`
- `%esp`

**Observation**

- Saved and restored register `%ebx`
- Not so for `%eax`, `%ecx`, `%edx`
**Disassembled swap**

```
08048384 <swap>:                       push %ebp
  8048385:  89 e5             mov %ebp,%esp
  8048387:  53               push %ebx
  8048388:  8b 55 08         mov 0x8(%ebp),%edx
  804838a:  8b 1a             mov (%edx),%ebx
  804838c:  8b 01             mov (%ecx),%eax
  804838e:  8b 55 08         mov 0x8(%ebp),%edx
  8048390:  8b 01             mov (%ecx),%eax
  8048392:  89 02             mov %eax,(%edx)
  8048394:  89 19             mov %ebx,(%ecx)
  8048396:  5b               pop %ebx
  8048397:  5d               pop %ebp
  8048398:  c3               ret

080483b4:  movl $0x8049658,0x4(%esp)  mov $0x8049654,(%esp)  call 08048384 <swap>  leave  ret
```

**Register Saving Conventions**

- When procedure `yoo` calls `who`:
  - `yoo` is the caller
  - `who` is the callee
- Can register be used for temporary storage?
  - When `yoo` calls `who`:
    - `yoo` is the caller
    - `who` is the callee
- Can register be used for temporary storage?
  - Contents of register `%edx` overwritten by `who`
  - This could be trouble => something should be done!
  - Need some coordination

**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

**Today**

- **IA 32 Procedures**
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers
- **X86-64 Procedures**
Recursive Function

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return (x & 1) + pcount_r(x >> 1);
}
```

**Registers**
- `%eax`, `%edx` used without first saving
- `%ebx` used, but saved at beginning & restored at end

Recursive Call #1

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return (x & 1) + pcount_r(x >> 1);
}
```

**Actions**
- Save old value of `%ebx` on stack
- Allocate space for argument to recursive call
- Store `x` in `%ebx`

```
pcount_r:
pushl %ebp
    movl %esp, %ebp
pushl %ebx
    subl $4, %esp
    movl 8(%ebp), %ebx
movl $0, %eax
    testl %ebx, %ebx
    je .L3
    movl %ebx, %eax
    shrl %eax
    movl %eax, (%esp)
call pcount_r
    movl %ebx, %edx
    andl $1, %edx
    leal (%edx,%eax), %eax
.L3:
    addl $4, %esp
    popl %ebx
    popl %ebp
    ret
```

Recursive Call #2

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return (x & 1) + pcount_r(x >> 1);
}
```

**Actions**
- If `x == 0`, return with `%eax` set to 0

```
%ebx x
```

Recursive Call #3

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return (x & 1) + pcount_r(x >> 1);
}
```

**Actions**
- Store `x >> 1` on stack
- Make recursive call

```
%ebx x >> 1
```

Recursive Call #4

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return (x & 1) + pcount_r(x >> 1);
}
```

**Assume**
- `%eax` holds value from recursive call
- `%ebx` holds `x`

**Actions**
- Compute `(x & 1)` + computed value

```
%eax x
```

Recursive Call #5

```c
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return (x & 1) + pcount_r(x >> 1);
}
```

**Actions**
- Restore values of `%ebx` and `%ebp`
- Restore `%esp`

```
%ebx x
```

2/11/2015
Observations About Recursion

- **Handled Without Special Consideration**
  - Stack frames mean that each function call has private storage
  - Saved registers & local variables
  - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another’s data
  - Stack discipline follows call / return pattern
  - If P calls Q, then Q returns before P
  - Last-In, First-Out
- **Also works for mutual recursion**
  - P calls Q; Q calls P

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;
}
```

Creating Pointer as Argument

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

Retrieving local variable

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

IA 32 Procedure Summary

- **Important Points**
  - Stack is the right data structure for procedure call / return
  - If P calls Q, then Q returns before P
  - Recursion (& mutual recursion) handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in %eax
- **Pointers are addresses of values**
  - On stack or global

[Diagram of IA 32 Procedure Summary]
Today

- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

- X86-64 Procedures

### X86-64 Integer Registers

#### Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</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-r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

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

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

### 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)
- Avoiding stack
  - Can hold all local information in registers

### X86-64 Locals in the Red Zone

```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];
}
```

- Avoiding Stack Pointer Change
  - Can hold all information within small window beyond stack pointer

### x86-64-64 Integer Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%rax</td>
</tr>
<tr>
<td>%rbx</td>
<td>%rbx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%rcx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%rdx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%rsi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%rdi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%rsp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%rbp</td>
</tr>
<tr>
<td>%r8-r15</td>
<td>%r8-r15</td>
</tr>
</tbody>
</table>

- 64-bit pointers
x86-64 NonLeaf without Stack Frame

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

// No values held while swap being invoked
// No callee save registers needed
// rep instruction inserted as no-op
// Based on recommendation from AMD
```

Understanding x86-64 Stack Frame

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

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

Interesting Features of Stack Frame

- Allocate entire frame at once
  - All stack accesses can be relative to %rsp
  - Can delay allocation, since safe to temporarily use red zone

- Simple deallocation
  - Increment stack pointer
  - No base/frame pointer needed

x86-64 Stack Frame Example

```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]*a[i+1]);
}
```

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

x86-64 Procedure Summary

- Heavy use of registers
  - Parameter passing
  - More temporaries since more registers

- Minimal use of stack
  - Sometimes none
  - Allocate/deallocate entire block

- Many optimization choices (tricky to read)
  - What kind of stack frame to use
  - Various allocation techniques