Machine-Level Programming III: Procedures

Kai Shen

Outline: 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
Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: `call label`
  - Push return address on stack
  - Jump to `label`
- Procedure return: `ret`
  - Pop address from stack
  - Jump to address

Example from disassembly

```
804854e: e8 3d 06 00 00 call   8048b90 <main>
8048553: 50             pushl %eax
```

Return address = 0x8048553

```
8048591: c3             ret
```

Procedure Call Example

```
804854e:   e8 3d 06 00 00 call   8048b90 <main>
8048553:   50             pushl %eax
```

Procedure Return Example

```
8048591:   c3             ret
```

Stack:

- `%esp` (Stack Pointer)
- `%eip` (Program Counter)

Stack Grows Down

Stack “Top”

Increasing Addresses

Stack “Bottom”

Procedure Call Example

```
0x110 0x110 0x10c 0x108 0x104 0x8048553
%esp 0x108 %esp 0x104 0x804854e
%eip 0x804854e %eip 0x8048b90
before call 8048b90
```

Procedure Return Example

```
0x110 0x8048553 0x10c 0x108 0x104
%esp 0x104 %esp 0x108
%eip 0x8048591 %eip 0x8048553
before ret
```

%esp: Stack Pointer
%eip: Program Counter
**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

**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

Stack

Example

Stack
Example

```c
struct {
    int who(...);
    ...
    ami();
    ...
};
```

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

Calling swap from call_swap

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

```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;
}
```

**Setup**

- `Set Up`
  - `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)`

**Body**

- `movl %esp, %ebp`
- `pushl %ebx`
- `movl %edx, %ebx`
- `movl (%ecx), %eax`
- `movl %eax, (%edx)`
- `popl %ebx`
- `popl %ebp`
- `ret`

**Finish**

- `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 $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`
### 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**
  - Caller saves if wants to use them
- **%esp, %ebp**
  - Special form of callee save
  - Restored to original values upon exit from procedure

### 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

### Outline: Procedures

- Stack Structure & Calling Conventions
- Illustrations of Recursion & 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;
}
```

- **add3** creates pointer and passes it to **incrk**
Creating and Initializing Local Variable

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

- Variable `localx` must be stored on stack
- Because: Need to create pointer to it
- Compute pointer as -4(%ebp)

Creating Pointer as Argument

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

- Use leal instruction to compute address of `localx`

Retrieving local variable

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

- Retrieve `localx` from stack as return value

Pointer Passing

Generating Pointer

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

Referencing Pointer

- Can callee return a pointer to its local variable back to caller?
Outline: Procedures

- Stack Structure & Calling Conventions
- Illustrations of Recursion & Pointers
- X86-64 Procedures

x86-64 Registers

- %rax %eax %r8 %r8d
- %rbx %ebx %r9 %r9d
- %rcx %ecx %r10 %r10d
- %rdx %edx %r11 %r11d
- %rsi %esi %r12 %r12d
- %rdi %edi %r13 %r13d
- %rsp %esp %r14 %r14d
- %rbp %ebp %r15 %r15d

- 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
- %rsp: Stack pointer
- %rbp: Callee saved
- %r8: Argument #5
- %r9: Argument #6
- %r10: Caller saved
- %r11: Caller saved
- %r12: 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/%rbp
  - What is the problem?
- 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

- 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 NonLeaf without Stack Frame

- No values held while swap being invoked
- No callee save registers needed

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

### x86-64 Stack Frame Example

- If need callee-save registers, must set up stack frame to save them
- Why not try to use caller-save registers?

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

```c
/* 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]);
}
```
Understanding x86-64 Stack Frame

```assembly
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 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 can be relative to %rsp
    - Allocate entire frame at once
    - Simpler deallocation
- Can delay allocation, since safe to temporarily use red zone

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.