Lecture 5

<2016-04-11 Mon>

Contents

1 x86-64 Stack

```
     || stack bottom ||
     ||      .       ||       .
     ||      |       ||      / \
     ||      |       ||       |
     ||     \ /      ||       |
     ||      .       ||       .
     ||    stack     ||    address
stack ||    grows     ||   increases
```

```
||    down      ||        .
||      .       ||        .
||      |       ||       / \
||      |       ||        |
||     \ /      ||        |
||      .       ||        .
||  stack top   ||  <== stack pointer %rsp
```

- `pushq src`
  - fetch operand (register) at src
  - decrement `%rsp` by 8
  - write operand at address given by `%rsp`
  - grow stack
- `popq dest`
  - read value (stack) at address given by `%rsp`
  - increment `%rsp` by 8
  - store value at dest
  - shrink stack

2 Calling Convention

2.1 Passing Control

2.1.1 example

```
void multistore(long x, long y, long *dest) {
  long t = mult2(x, y);
  *dest = t;
}

long mult2(long a, long b) {
  long s = a * b;
  return s;
}
```

register	variable
`%rbx`	`t`
`%rdi`	`x`
`%rsi`	`y`
`%rdx`	`dest`

```
multistore:
      push     %rbx             ;save %rbx
      mov      %rdx, %rbx       ;%rdx is the 3rd argument dest; save dest
      callq    <mult2>          ;mult2(x, y)
      mov      %rax, (%rbx)     ;write return value of mult2 to address %rbx
      pop      %rbx             ;restore %rbx
      retq

mult2:
      mov      %rdi, %rax       ;a
      imul     %rsi, %rax       ;a *= b
      retq
```

2.1.2 Procedure Control Flow

- procedure call: `call label`
  - push return address on stack
  - jump to `label`
- return address (jump to address after `ret` instruction)
  - address of the next instruction right after call
- procedure return: `ret`
  - pop address from stack
  - jump to return address

2.1.3 example

```
multistore:
      ...
      callq    <mult2>          ;push return address on stack, jump to <mult2>
      mov      %rax, (%rbx)
      ...
```

```
mult2:
      mov       %rdi, %rax
      ...
      retq                        ;jump to return address
```

2.2 Passing Data

usage	register
1st argument	`%rdi`
2nd argument	`%rsi`
3rd argument	`%rdx`
4th argument	`%rcx`
5th argument	`%r8`
6th argument	`%r9`
return value	`%rax`

additional arguments are allocated on the stack

2.3 Managing Local Data

- languages that support recursion e.g. C, Pascal, Java
  - code must be "Reentrant"
    - multiple simultaneous instantiation of single procedure
  - need 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

2.3.1 Stack Frame

- contents
  - return information
  - local storage
  - temporary space
- management
  - space allocated when enter procedure
    - "set-up" code
    - includes push by `call` instruction
  - deallocated when return
    - "finish" code
    - includes pop by `ret` instruction

2.3.2 x86-64/Linux Stack Frame

- current stack frame (callee)
  - in sequence of "top" to "bottom"
  - parameters for function about to call
  - local variables (if can't keep in registers)
  - saved register contents
  - old frame pointer (optional)
- caller stack frame
  - return address
    - pushed by `call` instruction
  - arguments for this call

```
        || stack bottom ||
        ||              ||
        ||              ||
caller  || arguments 7+ ||
frame   ||--------------||
        || return addr  ||
```

```
        ||--------------||
        ||  old %rbp    ||  <== frame pointer (%rbp) (optional)
--------||--------------||
        ||              ||
        ||    saved     ||
        ||  registers   ||
        ||      +       ||
callee  ||    local     ||
frame   ||  variables   ||
        ||--------------||
        ||  argument    ||      (optional)
        ||    build     ||
        ||              ||  <== stack pointer %rsp
```

2.3.3 example

```
long incr(long *p, long val) {
  long x = *p;
  long y = x + val;
  *p = y;
  return x;
}
```

register	variable
%rdi	p
%rsi	val , y
%rax	x , return value

```
incr:
      movq      (%rdi), %rax    ;x = *p
      addq      %rax, %rsi      ;val += x  (y = x + val)
      movq      %rsi, (%rdi)    ;*p = y
      ret
```

1. calling `incr`

   ```
   long call_incr() {
     long v1 = 15213;
     long v2 = incr(&v1, 3000);
     return v1 + v2;
   }
   ```

register	variable
%rdi	first argument passed to incr , &v1
%rsi	second argument passed to incr , 3000

```
call_incr:
      subq     $16, %rsp        ;reserve space for temporary variable (15213) and
      movq     $15213, 8(%rsp)  ;write 15213 to address 8+%rsp (&v1)
      movl     $3000, %rsi      ;write 3000 as the second argument passed to incr
      leaq     8(%rsp), %rdi    ;write address of 8+%rsp (&v1) as the first argume
      call     incr
      addq     8(%rsp), %rax    ;%rax += v1
      addq     $16, %rsp        ;deallocate space previously reserved
      ret
```

2.3.4 Register Saving Conventions

- caller
- callee

```
caller:
      ...
      movq     $15213, %rdx
      call     callee
      addq     %rdx, %rax      ;contents of register overwritten by callee
      ...                      ;THIS COULD BE TROUBLE
      ret

callee:
      ...
      subq     $18213, %rdx    ;contents of register overwritten by callee
      ...
      ret
```

- conventions
  - caller saved
    - caller saves temporary values in its frame before the call
  - callee saved
    - callee saves temporary values in its frame before using

  * callee restores them before returning to caller

1. x86-64 Linux Register Usage

   - caller saved
     - `%rax`
       * return value
       * can be modified by procedure (callee)
     - `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
       * arguments (first 6)
       * can be modified by procedure (callee)
     - `%r10`, `%r11`
       * can be modified by procedure (callee)
   - callee saved
     - `%rbx`, `%r12`, `%r13`, `%r14`
       * callee must save and restore
     - `%rbp`
       * callee must save & restore
       * maybe used as frame pointer
       * can mix & match
     - `%rsp`
       * special form of callee save
       * restored to original value upon exit from procedure (callee)

register	usage	caller/callee saved
%rax	return value	caller saved
%rdi %rsi %rdx %rcx %r8 %r9	arguments	caller saved
%r10 %r11	temporaries	caller saved
%rbx %r12 %r13 %r14	temporaries	callee saved
%rbp	(frame pointer)	callee saved
%rsp	stack pointer	callee saved

(a) callee saved example

```
long call_incr2(long x) {
  long v1 = 15213;
  long v2 = incr(&v1, 3000);
  return x + v2;
}

call_incr2:
      pushq     %rbx             ;save %rbx
      subq      $16, %rsp
      movq      %rdi, %rbx       ;%rbx = x
      movq      $15213, 8(%rsp)
      movl      $3000, %esi      ;second argument passed to incr
      leaq      8(%rsp), %rdi    ;first argument passed to incr
      call      incr
      addq      %rbx, %rax
      addq      $16, %rsp
      popq      %rbx             ;restore %rbx
      ret
```

2.4 Recursive Function Call

2.4.1 example

```
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x << 1);
}
```

register	lower-order 4 bytes	variable
%rdi		x, first argument
%rbx	%ebx	temporary x
%rax	%eax	return value

```
pcount_r:
      movl      $0, %eax
      testq     %rdi, %rdi      ;%rdi & %rdi (without setting destination)
      je        .L6             ;jump if zero flag is set (%rdi & %rdi == 0)
      pushq     %rbx            ;save %rbx
      movq      %rdi, %rbx      ;%rbx = x
      andl      $1, %ebx        ;%rbx &= 1
      shrq      %rdi            ;x >> 1, also as first argument
      call      pcount_r
      addq      %rbx, %rax
      popq      %rbx            ;restore %rbx
.L6:
      ret
```

2.4.2 Observation

- handled without special consideration
  - stack frame mean that each functiion has private storage
    - saved registers & local variables
    - saved return pointer
  - register saving conventions prevent one function 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