Machine Programming Procedures

CSC 235 - Computer Organization

References

- Slides adapted from CMU

Outline

- Procedures
    - Mechanisms
    - Stack Structure
    - Calling Conventions
        - Passing Control
        - Passing Data
        - Managing local data
    - Illustration of Recursion

Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return values
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return

Mechanisms in Procedures (continued)

- Mechanisms all implemented with machine instructions, but the choices are determined by designers. These choices make up the Application Binary Interface (ABI).
- x86-64 implementation of a procedure uses only those mechanisms required

x86-64 Stack

- Region of memory managed with stack discipline
  - Memory viewed as array of bytes
  - Different regions have different purposes
  - (Like ABI, a policy decision)
- Grows toward lower addresses
- The `%rsp` register contains the lowest stack address (“top” of stack)

x86-64 Stack: Push

- Syntax: `pushq` *Src*
- Semantics:
  - Fetch operand at *Src*
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`

x86-64 Stack: Pop

- Syntax: `popq` *Dest*
- Semantics:
    - Read value at address given by `%rsp`
    - Increment `%rsp` by 8
    - Store value at *Dest* (usually a register)
    - Note that the memory does not change, only the value of `%rsp`

x86-64 Stack Example

Initially

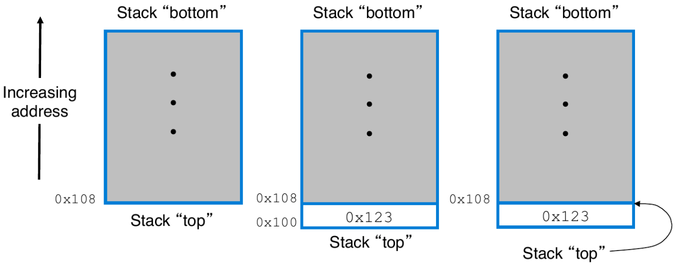
%rax	0x123
%rdx	0
%rsp	0x108

`pushq %rax`

%rax	0x123
%rdx	0
%rsp	0x100

`popq %rdx`

%rax	0x123
%rdx	0x123
%rsp	0x108



Code Examples

- C code

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

- Assembly

```
multstore:
    push    %rbx            # save %rbx
    mov     %rdx, %rbx      # save dest
    callq   mult2           # mult2(x, y)
    mov     %rax, (%rbx)    # save at dest
    pop     %rbx            # restore %rbx
    retq                    # return
```

Code Examples

- C code

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

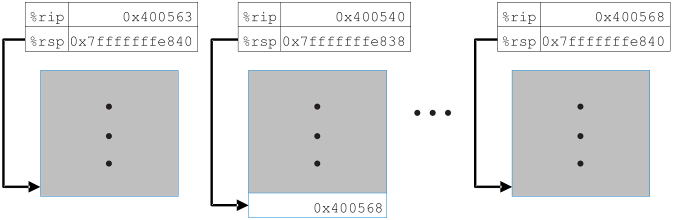
- Assembly

```
mult2:
    mov     %rdi, %rax   # a
    imul    %rsi, %rax   # a * b
    retq                 # return
```

Procedure Control Flow

- Use stack to support prodecure 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
- Procedure return: `ret`
    - Pop address from stack
    - Jump to address

Procedure Control Flow Example



(a) Executing `call`

(b) After `call`

(c) After `ret`

Procedure Data Flow

- The first six integer or pointer parameters are passed in registers:
    1. `%rdi`
    2. `%rsi`
    3. `%rdx`
    4. `%rcx`
    5. `%r8`
    6. `%r9`
- Subsequent parameters (or parameters larger than 64 bits) should be pushed onto the stack, with the first argument topmost.
- Return value in `%rax`

Stack-Based Languages

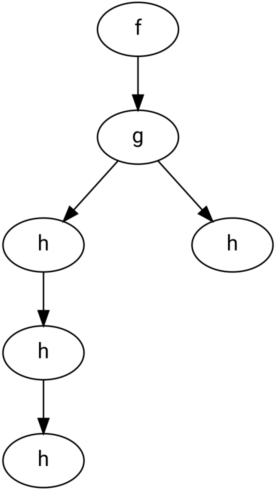
- Languages that support recursion
    - Code must be “reentrant”
        - Multiple simultaneous instantiations of single procedure
    - Need some place to store state of each instantiation
        - Arguments
        - Local variables
        - Return pointer

Stack-Based Languages (continued)

- Stack discipline
    - State for a given procedure needed for limited time
        - From when called to when returned
    - Callee returns before caller does
- Stack allocated in frames (activation records)
    - State for single procedure instantiation

Call Chain Example

- `f`: calls `g`
- `g`: calls `h` twice
- `h`: recursive



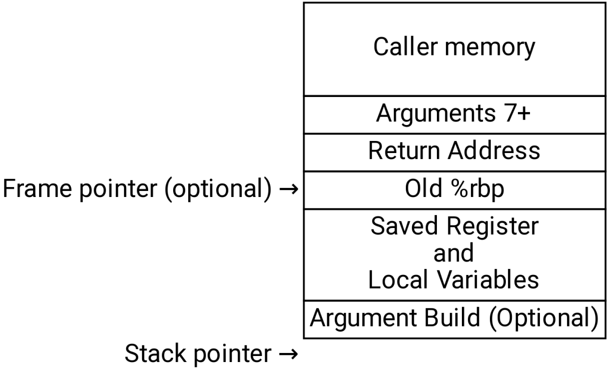
Stack Frames

- Contents
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)
- Management
  - Space allocated when procedure is entered
    - “set-up” code
    - Includes push by `call` instruction
  - Deallocated when returned from procedure
    - “finish” code
    - Includes pop by `ret` instruction

x86-64/Linux Stack Frame

- Current stack frame ("top" to bottom)
  - "Argument build:" parameters for function about to call
  - Local variables if cannot keep in registers
  - Saved register context
  - Old frame pointer (optional)
- Caller stack frame
  - Return address (pushed by `call` instruction)
  - Arguments for this call

x86-64/Linux Stack Frame



Caller memory

Arguments 7+

Return Address

Frame pointer (optional) → Old %rbp

Saved Register
and
Local Variables

Argument Build (Optional)

Stack pointer →

Example: incr

- C code

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

- Assembly code

```
incr:
    movq    (%rdi), %rax
    addq    %rax, %rsi
    movq    %rsi, (%rdi)
    ret
```

Register Saving Conventions

- When procedure `foo` calls `bar`:
    - `foo` is the caller
    - `bar` is the callee
- 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

x86-64 Linux Register Usage

- `%rax`
  - Return value
  - Caller-saved, can be modified by procedure
- `%rdi`, . . ., `%r9`
  - Arguments
  - Caller-saved, can be modified by procedure
- `%r10`, `%r11`
  - Caller-saved, can be modified by procedure

x86-64 Linux Register Usage

- `%rbx`, `%r12`, `%r13`, `%r14`
  - Callee-saved, callee must save and restore
- `%rbp`
  - Callee-saved, callee must save and restore
  - May be used as frame pointer
  - Can mix and match
- `%rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure

Recursive Function Example

- C code

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

Recursive Function Example

- Assembly

```
pcount_r
    movl    $0, %eax    # base case
    testq   %rdi, %rdi  # |
    je      .L6         # |
    pushq   %rbx        # caller save
    movq    %rdi, %rbx  # set up call
    andl    $1, %ebx    # | x & 1
    shrq    %rdi        # | x >> 1
    call    pcount_r    # recursive call
    addq    %rbx, %rax  # result
    popq    %rbx        # function completion
.L6:
    rep; ret            # base case
```

Observations About Recursion

- Handled without special consideration
  - Stack frames mean that each function call has private storage
  - Register saving conventions prevent one function call from corrupting another's data
  - Stack discipline follows call/return pattern
- Also works for mutual recursion

x86-64 Procedure Summary

- Important Points
  - Stack is the correct data structure for procedure call/return
  - If P calls Q, then Q returns before P
- 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
  - Return result in `%rax`
- Pointers are addresses of values