System Programming

08. Machine-Level Programming III: Procedures – Why stack?

2019. Fall

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

datalab

data science laboratory

Function Call Problems (1/2)

- Calling and returning
  - How does caller function jump to callee function?
  - How does callee function jump back to the right place in caller function?
- Passing parameters
  - How does caller function pass parameters to callee function?
- Storing local variables
  - Where does callee function store its local variables?

Function Call Problems (2/2)

- Handling registers
  - How do caller and callee functions use same registers without interference?
- Returning a value
  - How does callee function send return value back to caller function?

Calling and Returning

- How does caller function jump to callee function?
    - i.e., Jump to the address of the callee's first instruction

- How does the callee function jump back to the right place in caller function?
    - i.e., Jump to the instruction immediately following the most-recently-executed call instruction

Attempted Solution: Use Jmp Instruction

- caller and callee use jmp instruction

```
P:              # Function P

      …

      jmp R   # Call R

 Rtn_point1:
```

```
R:                # Function R

      …

  jmp Rtn_point1     #Return
```

Attempted Solution: Use Jmp Instruction

- Problem: callee may be called by multiple callers

```
P:              # Function P

        …
        jmp R   # Call R
 Rtn_point1:
```

```
R:                  # Function P

        …
   jmp ???          #Return
```

```
Q:              # Function P

        …
        jmp R   # Call R
 Rtn_point2:
```

Attempted Solution: Use Register

- Attempted solution 2: Store return address in register

```
P:              # Function P

    movl $Rtn_point1, %eax
    jmp R    # Call R
 Rtn_point1:
  …
```

```
Q:              # Function P

    movl $Rtn_point1, %eax
    jmp R    # Call R
 Rtn_point2:
  . . .
```

```
R:                    # Function P

         …
 jmp *%eax   #Return
```

Special form of jmp instruction; we will not use

Attempted Solution: Use Register

- Problem: Cannot handle nested function calls

```
P:             # Function P

    movl $Rtn_point1, %eax
    jmp Q    # Call R
 Rtn_point1:
  …
```

```
R:               # Function P

      …
  jmp *%eax   #Return
```

```
Q:             # Function P

    movl $Rtn_point1, %eax
    jmp R    # Call R
 Rtn_point2:
  . . .
```

Problem if P calls Q, and Q calls R

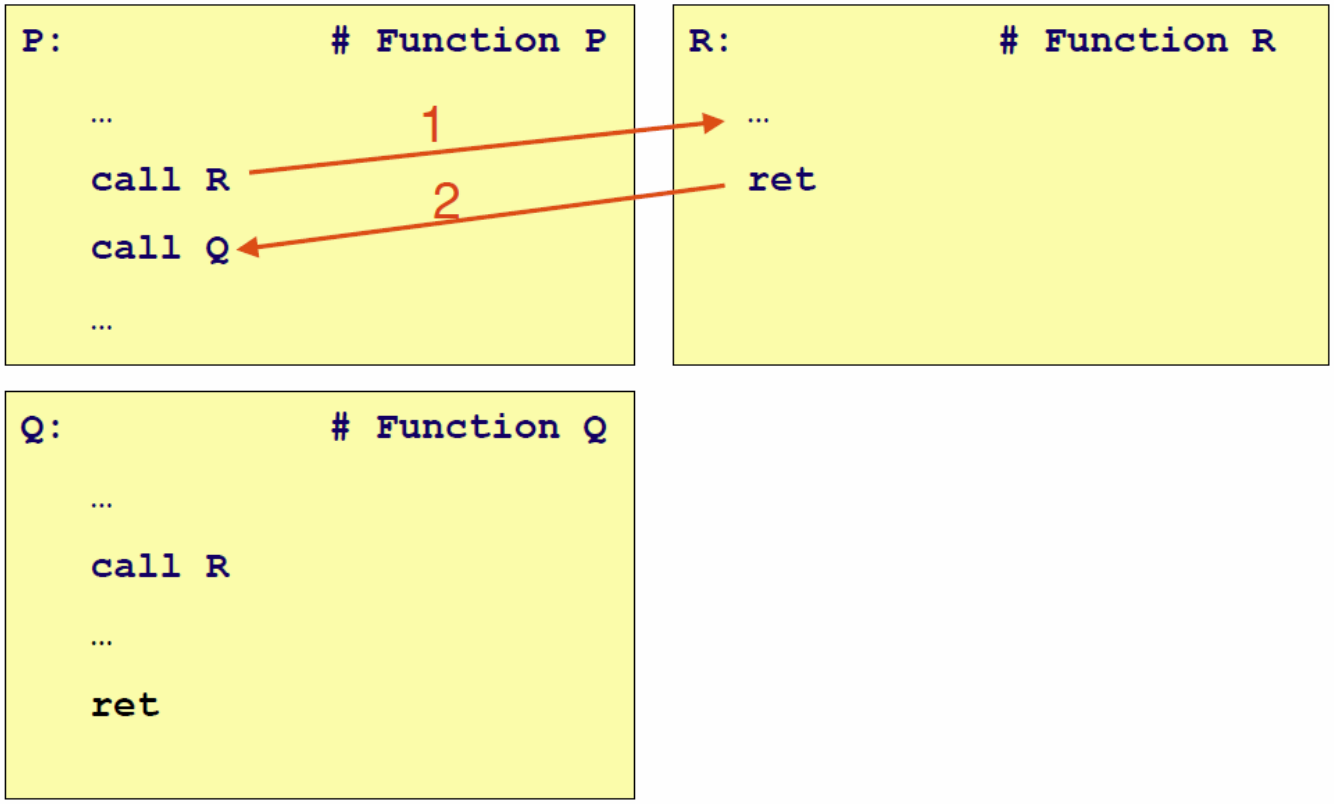
Return address for P to Q call is lost

IA-32 Solution: Use the Stack

- May need to store many return addresses
  - The number of nested functions is not known in advance
  - A return address must be saved for as long as the function invocation continues, and discarded thereafter
- Addresses used in reverse order
  - E.g., function P calls Q, which then calls R
  - Then R returns to Q which then returns to P
- Last-in-first-out data structure (stack)
  - Caller pushes return address on the stack
  - … and callee pops return address off the stack
- IA 32 solution: Use the stack via call and ret

IA-32 Call and Ret Instructions

- Ret instruction “knows” the return address

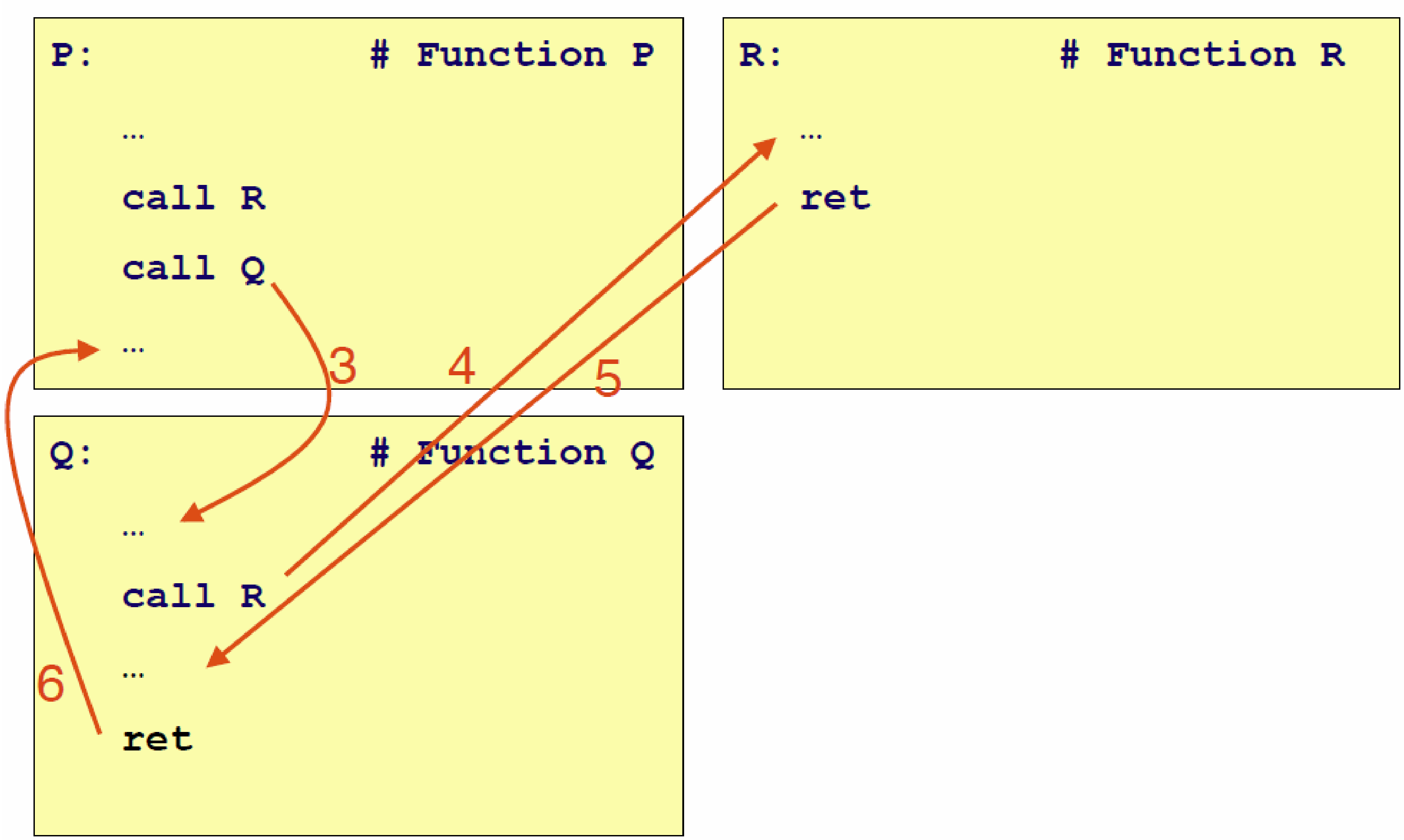


```
P:                # Function P

    …

    call R

    call Q

    …
```

```
R:                # Function R

    …

    ret
```

```
Q:                # Function Q

    …

    call R

    …

    ret
```

- Ret instruction “knows” the return address



```
P:                  # Function P

    …

    call R

    call Q

    …
```

```
Q:                  # Function Q

    …

    call R

    …

    ret
```

```
R:                  # Function R

    …

    ret
```

Implementation of Call

- ESP (stack pointer register) points to top of stack

Instruction	Effective Operations

`pushl src`	`subl $4, %esp`
	`movl src, (%esp)`

`popl dest`	`movl (%esp), dest`
	`addl $4, %esp`

Implementation of Call

- EIP (instruction pointer register) points to next instruction to be executed

Note: can’t really access EIP directly, but this is implicitly what call is doing

Instruction	Effective Operations

`pushl src`	`subl $4, %esp`
	`movl src, (%esp)`

`popl dest`	`movl (%esp), dest`
	`addl $4, %esp`

`call addr`	`pushl %eip`
	`jmp addr`

Call instruction pushes return address (old EIP) onto stack

Implementation of Ret

- Ret instruction pops stack, thus placing return address (old EIP) into EIP

Instruction	Effective Operations

`pushl src`	`subl $4, %esp`
	`movl src, (%esp)`

`popl dest`	`movl (%esp), dest`
	`addl $4, %esp`

`call addr`	`pushl %eip`
	`jmp addr`

`ret`	`pop %eip`

Note: can’t really access EIP directly, but this is implicitly what call is doing

Q&A