程序的机器级表示：过程

Machine-Level Programming : Procedures

本章内容

Topic

- 栈的结构

  Stack Structure

- 过程调用规范

  Calling Conventions

- 递归

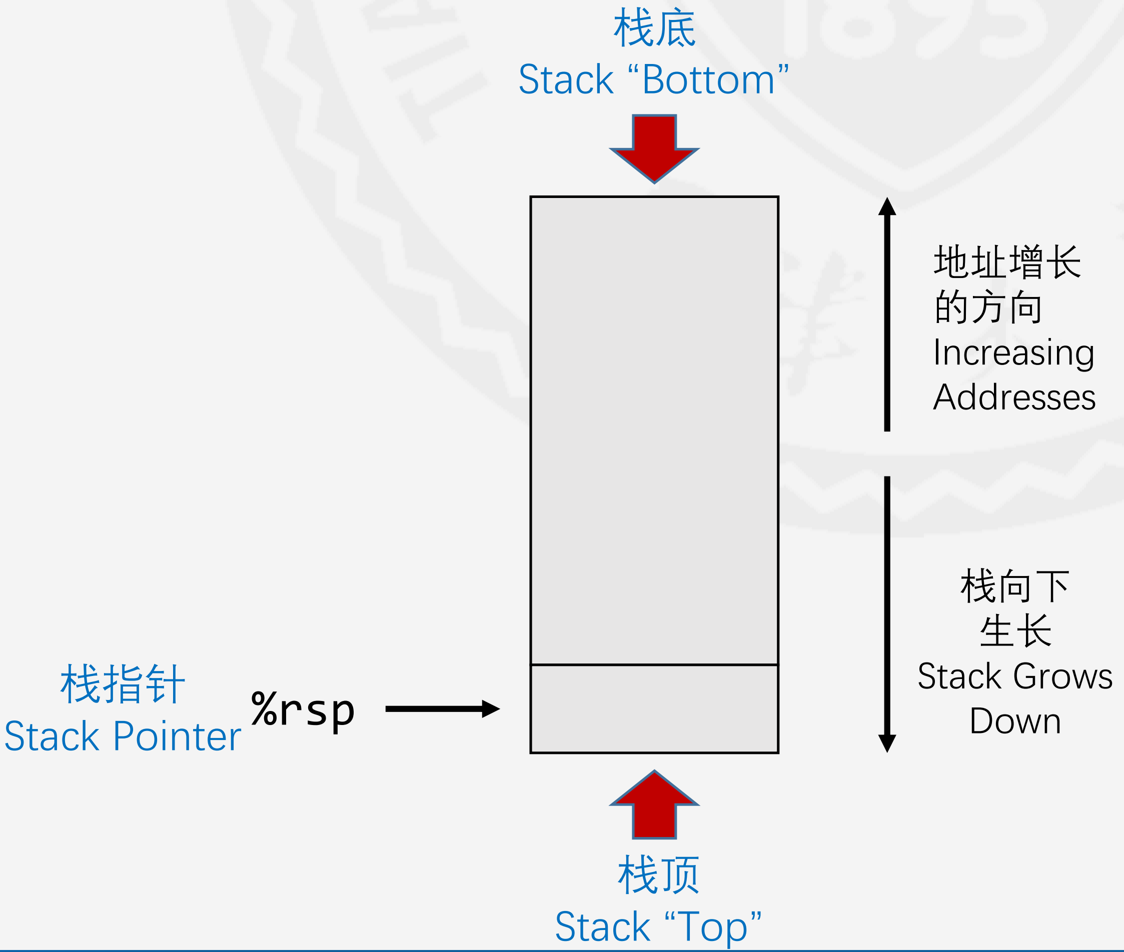
  Recursion

栈的结构

Stack Structure

x86-64的栈
x86-64 Stack

- 使用栈的规则管理内存区域
  Region of memory managed with stack discipline
- 向低地址方向生长
  Grows toward lower addresses
- 寄存器 `%rsp` 栈的最低地址
  Register `%rsp` contains lowest stack address

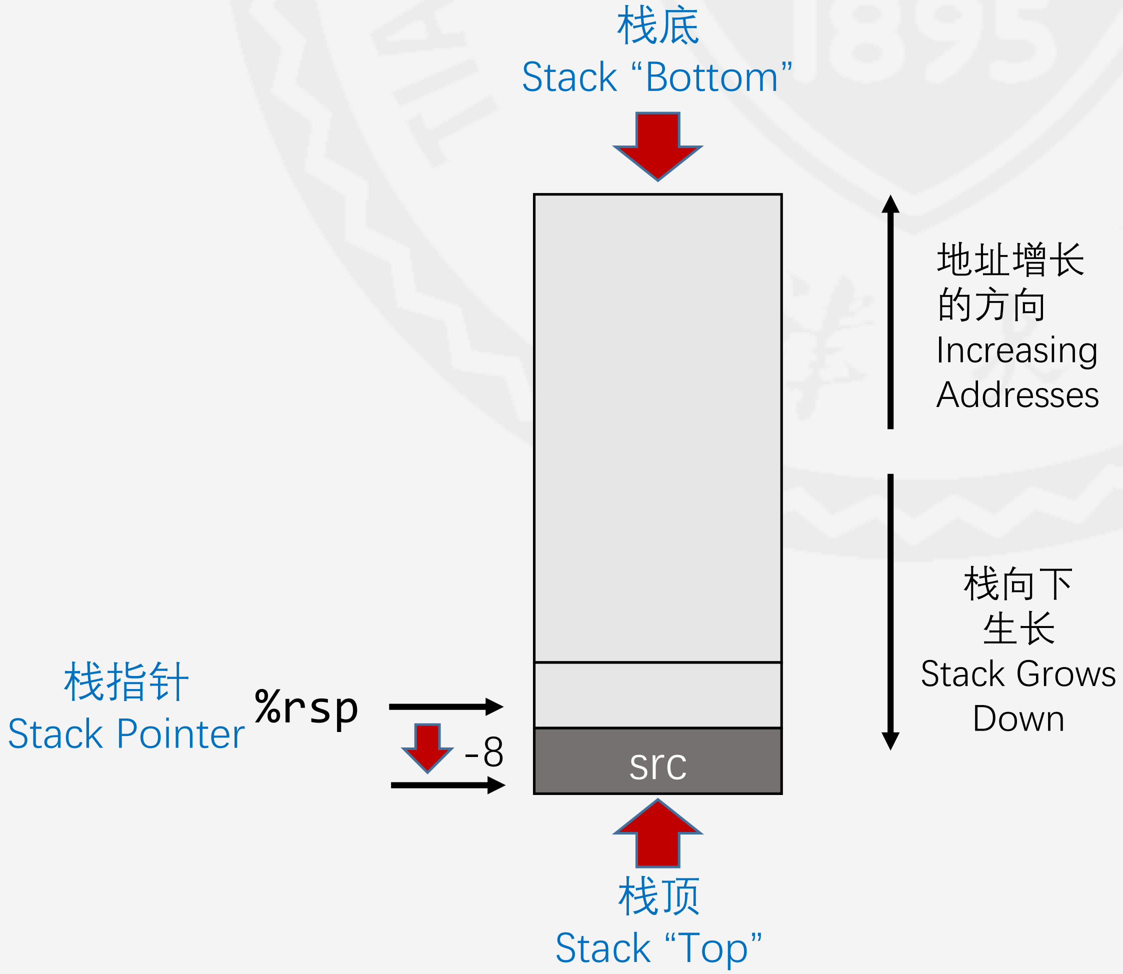


栈的结构

Stack Structure

x86-64的栈：入栈
x86-64 Stack: Push

- `pushq src`
  1. 从 `src` 中取出操作数
     Fetch operand at `src`
  2. `%rsp` 减 8
     Decrement `%rsp` by 8
  3. 将操作数的值写入`%rsp`指向的地址
     Write operand at address given by `%rsp`

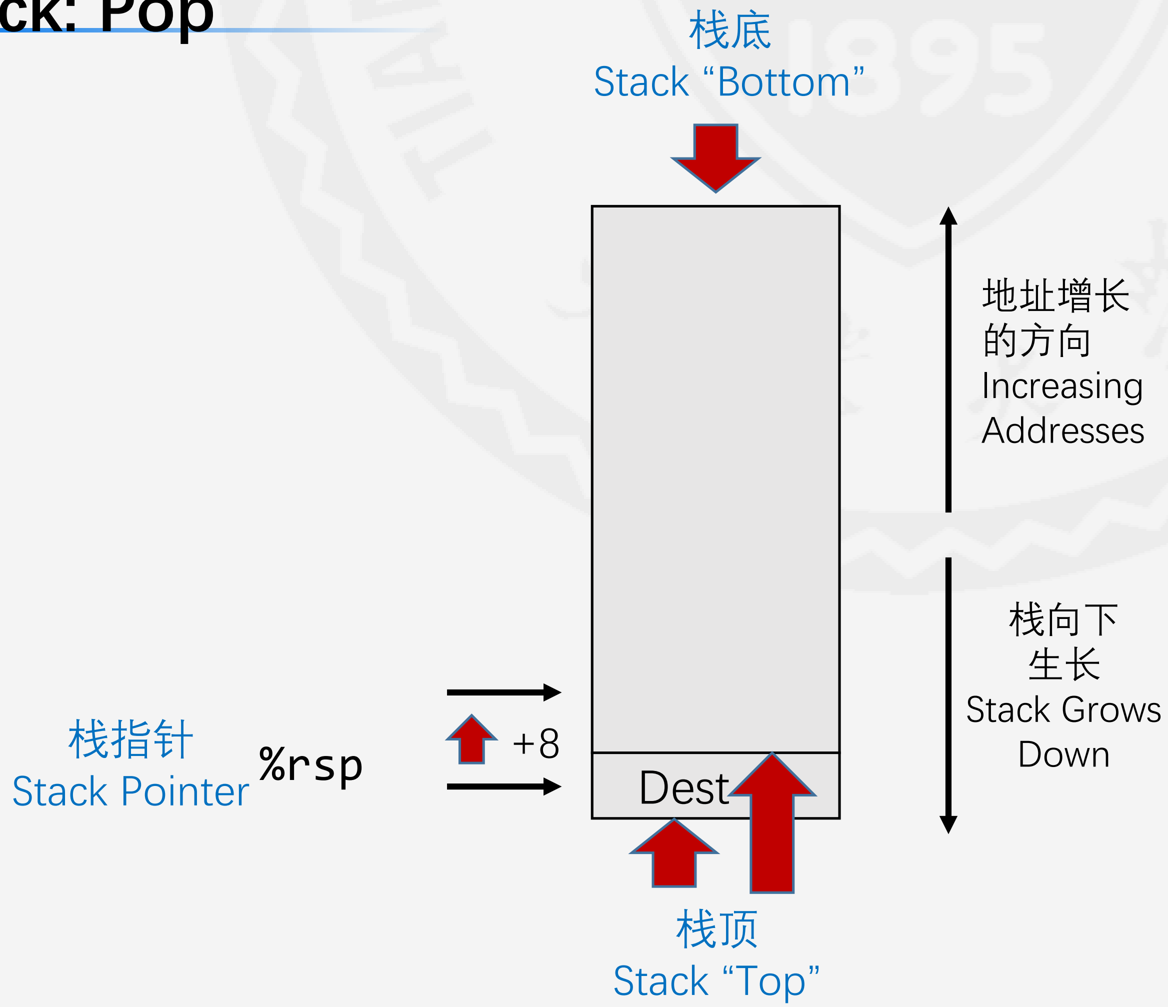


栈的结构

Stack Structure

x86-64的栈：出栈
x86-64 Stack: Pop

- `popq dest`
  1. 从 `%rsp` 指向的地址中取出值
     Fetch value in `%rsp`
  2. 将值写入 `dest`
     Write value to operand `dest`
  3. `%rsp` 加 8
     Increment %rsp by 8



本章内容

Topic

- 栈的结构

  Stack Structure

- 过程调用规范

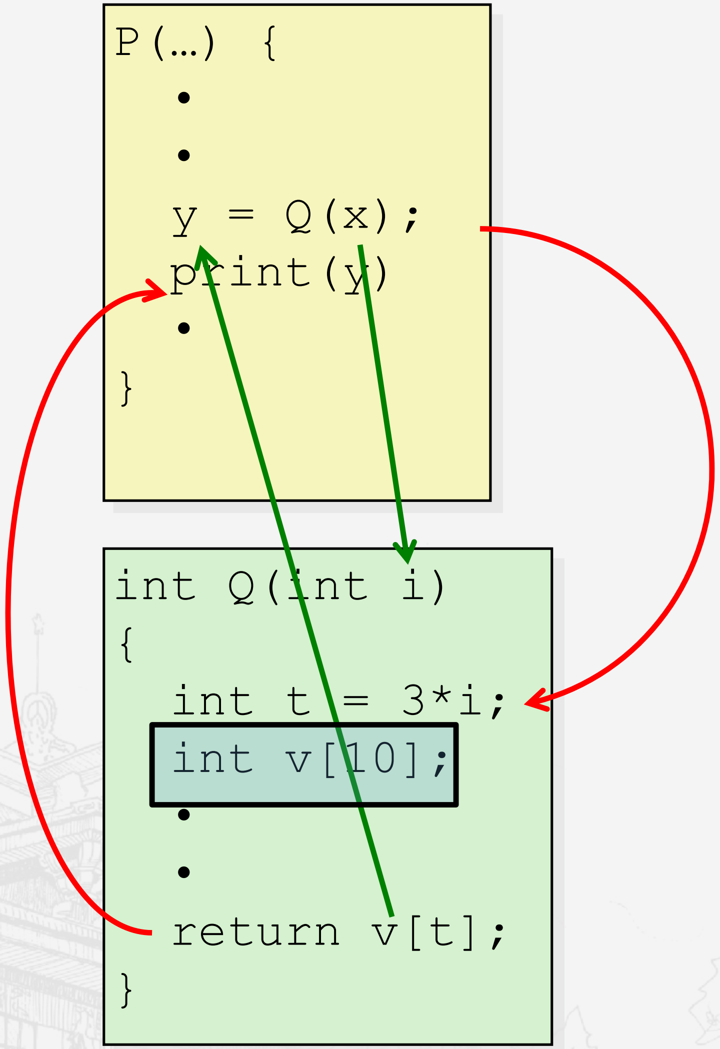
  Calling Conventions

- 递归

  Recursion

过程调用规范

Calling Conventions



- 控制流转移
  Passing control flow
  - 跳转至过程代码的开始位置（过程调用时）
    To beginning of procedure code
  - 跳转至返回点（过程返回时）
    Back to return point
- 数据传递
  Passing data
  - 过程的参数
    Procedure arguments
  - 返回值
    Return value
- 存储管理
  Memory management
  - 在过程执行时分配存储空间
    Allocate during procedure execution
  - 在过程返回时回收空间
    Deallocate upon return
- 所有过程调用的都是通过机器指令实现的（软件实现而非硬件实现）
  Mechanisms all implemented with machine instructions
- x86-64在实现某个过程时，只选择实现当前过程所需要的机制
  x86-64 implementation of a procedure uses only those mechanisms required

本章内容

Topic

- 栈的结构
  Stack Structure
- 过程调用规范
  Calling Conventions
  - 控制流转移
    Passing Control Flow
  - 数据传递
    Passing Data
  - 存储管理
    Memory Management
- 递归
  Recursion

过程调用规范

Passing Control

过程控制流
Procedure Control Flow

- 使用栈支持过程的调用和返回
  Use stack to support procedure call and return
- **过程调用：** `call label`
  Procedure call: `call label`
  - 将返回地址压入栈
    Push return address on stack
  - 跳转至 `label` 标签
    Jump to `label`
- 返回地址
  Return address:
  - 紧接call指令的指令所在地址
    Address of the next instruction right after call

- **过程返回：** `ret`
  Procedure return: `ret`
  - 从栈中弹出返回地址
    Pop address from stack
  - 跳转至返回地址
    Jump to address

举例：控制流转移
Passing Control Flow Examples

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

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
```

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

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax    # a * b
  400557:  retq                # Return
```

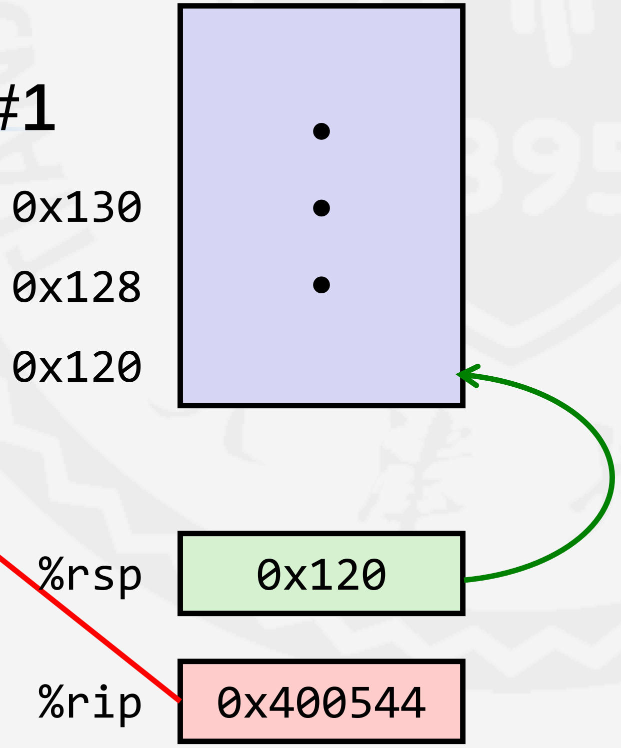
过程调用规范

Passing Control

举例：控制流转移 #1
Passing Control Flow Examples #1

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



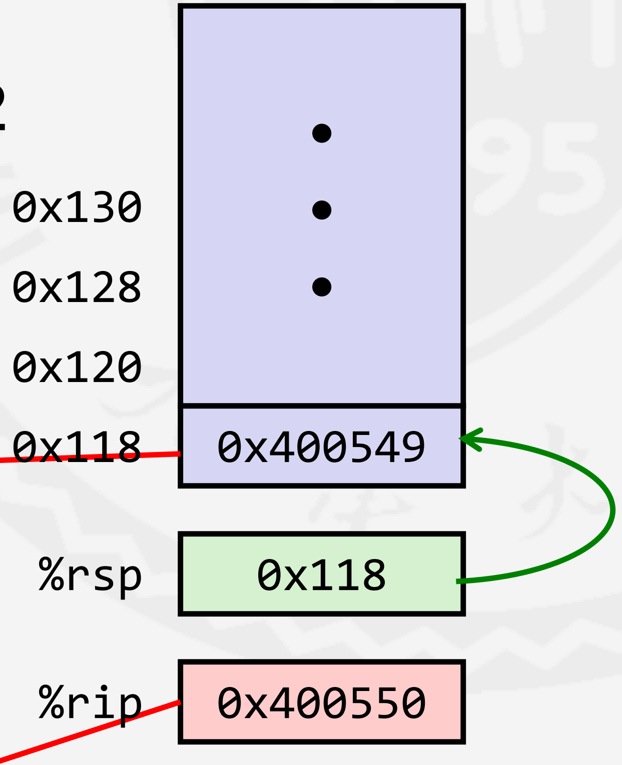
过程调用规范

Passing Control

举例：控制流转移 #2
Passing Control Flow Examples #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



过程调用规范

Passing Control

举例：控制流转移 #3
Passing Control Flow Examples #3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

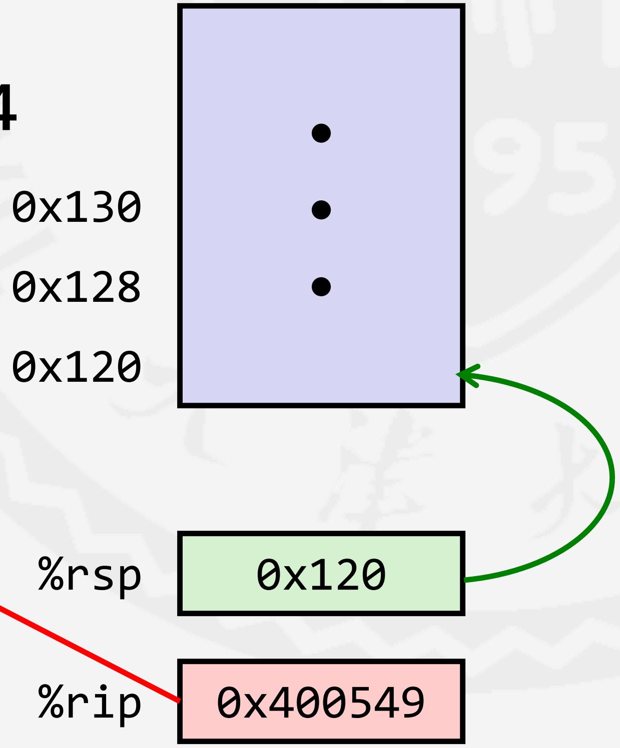
0x130	•
0x128	•
0x120	•
0x118	0x400549

%rsp	0x118
%rip	0x400557

举例：控制流转移 #4
Passing Control Flow Examples #4

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



本章内容

Topic

- 栈的结构

  Stack Structure

- 过程调用规范

  Calling Conventions

  - 控制流转移

    Passing Control Flow

  - 数据传递

    Passing Data

  - 存储管理

    Memory Management

- 递归

  Recursion

过程调用规范

Passing Control

过程数据流
Procedure Data Flow

- 前6个参数通过寄存器传递

  First 6 arguments passed by registers

%rdi
%rsi
%rdx
%rcx
%r8
%r9

- 返回值

  Return value

%rax

- 剩余参数通过栈传递
- Other arguments passed by Stack

• • •
Arg *n*
• • •
Arg 8
Arg 7

- 只有需要的时候栈才会分配空间

  Only allocate stack space when needed

举例：过程数据流
Data Flow Example

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

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  • • •
  400541: mov    %rdx,%rbx        # Save dest
  400544: callq  400550 <mult2>   # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)      # Save at dest
  • • •
```

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

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax   # a
  400553:  imul   %rsi,%rax   # a * b
  # s in %rax
  400557:  retq               # Return
```

本章内容

Topic

- 栈的结构
  Stack Structure
- 过程调用规范
  Calling Conventions
  - 控制流转移
    Passing Control Flow
  - 数据传递
    Passing Data
  - 存储管理
    Memory Management
- 递归
  Recursion

过程调用规范

Passing Control

基于栈的程序设计语言
Stack-Based Languages

- 支持递归的编程语言
  Languages that support recursion
  - e.g., C, Pascal, Java ,C#
  - 代码必须是可重入的
    Code must be “Reentrant”
    - 一个过程可以有多个实例
      Multiple simultaneous instantiations of single procedure
  - 需要空间存储每个过程实例的状态
    Need some place to store state of each instantiation
    - 参数
      Arguments
    - 局部变量
      Local variables
    - 返回地址
      Return address

- 栈的规则
  Stack discipline
  - 每个过程的状态只需要保存有限的时间
    State for given procedure needed for limited time
  - 从调用开始至返回结束
    From when called to when return
  - 被调用者先于调用者返回
    Callee returns before caller does
- 栈空间以**帧**的方式进行分配
  Stack allocated in **Frames**
  - 存储每个过程实例的状态
    state for single procedure instantiation

过程调用规范

Passing Control

举例：调用链
Call Chain Example

```
yoo(…)
{
    .
    .
    who();
    .
    .
}
```

```
who(…)
{
    ...
    amI();
    ...
    amI();
    ...
}
```

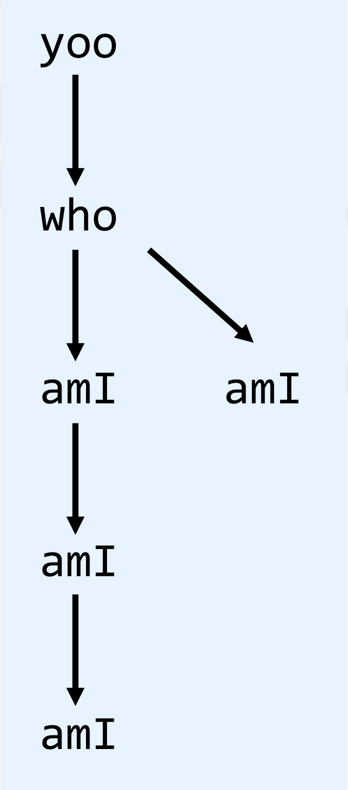
```
amI(…)
{
    .
    .
    amI();
    .
    .
}
```

amI() 是递归的

Procedure `amI()` is recursive

调用链示意

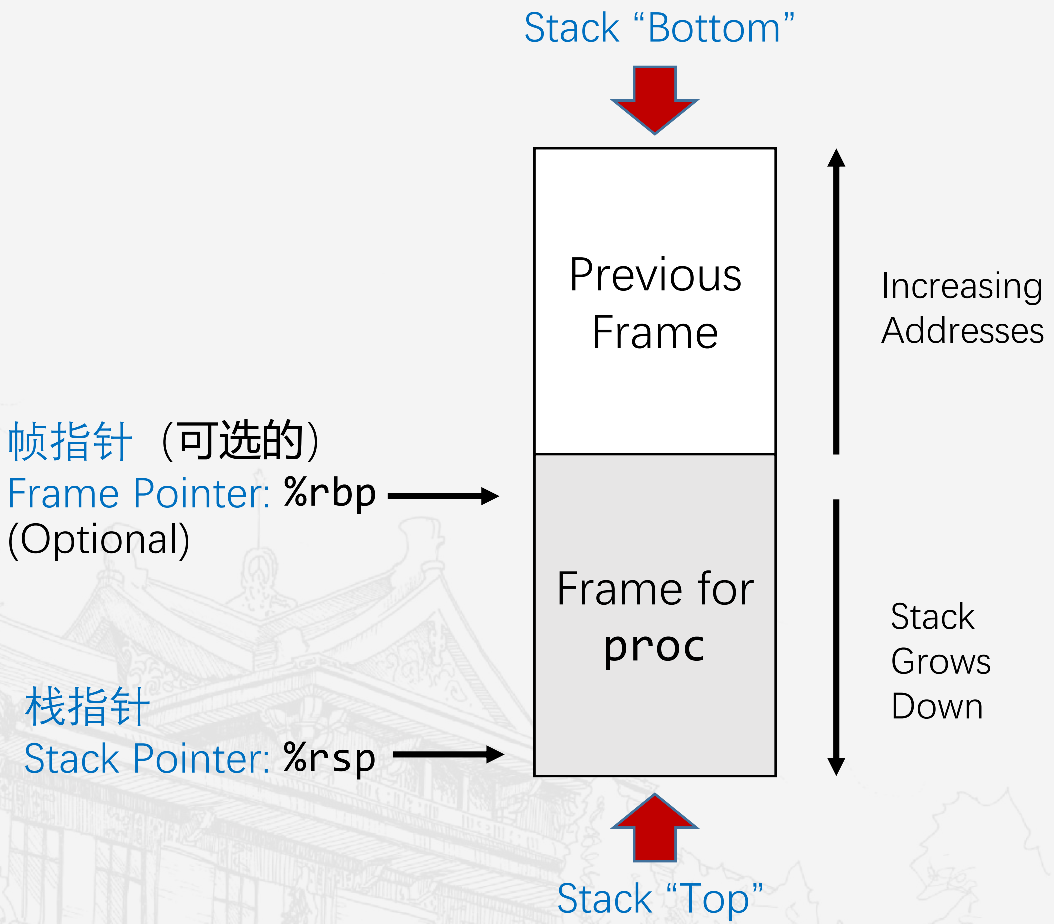
Call Chain



过程调用规范

Passing Control

栈帧
Stack Frames



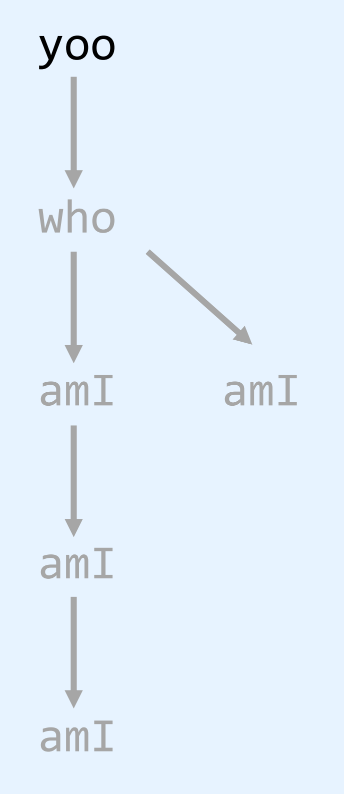
- 内容
  Contents
  - 返回地址
    Return address
  - 局部变量 （如果需要）
    Local variables (if needed)
  - 参数（如果需要）
    Other Arguments (if needed)
  - 其他临时空间（如果需要）
    Temporary space (if needed)
- 管理
  Management
  - 进入过程后，分配栈帧空间
    Space allocated when enter procedure
    - “建立”代码
      “Set-up” code
    - 包括 call指令中的入栈操作
      Includes push by call instruction
  - 过程返回前，释放栈帧空间
    Deallocated when return
    - “结束”代码
      “Finish” code
    - 包括 ret 指令中的出栈操作
      Includes pop by ret instruction

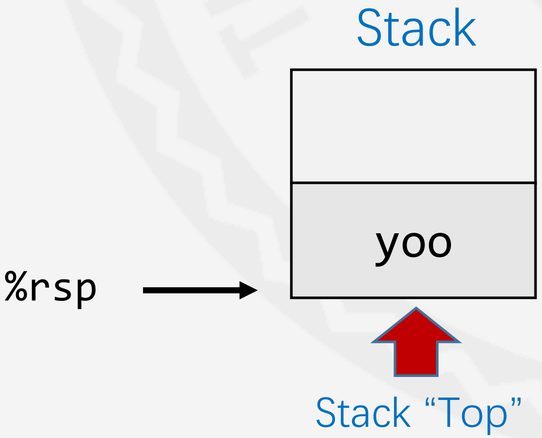
过程调用规范

Passing Control

举例：调用链
Call Chain Example

```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

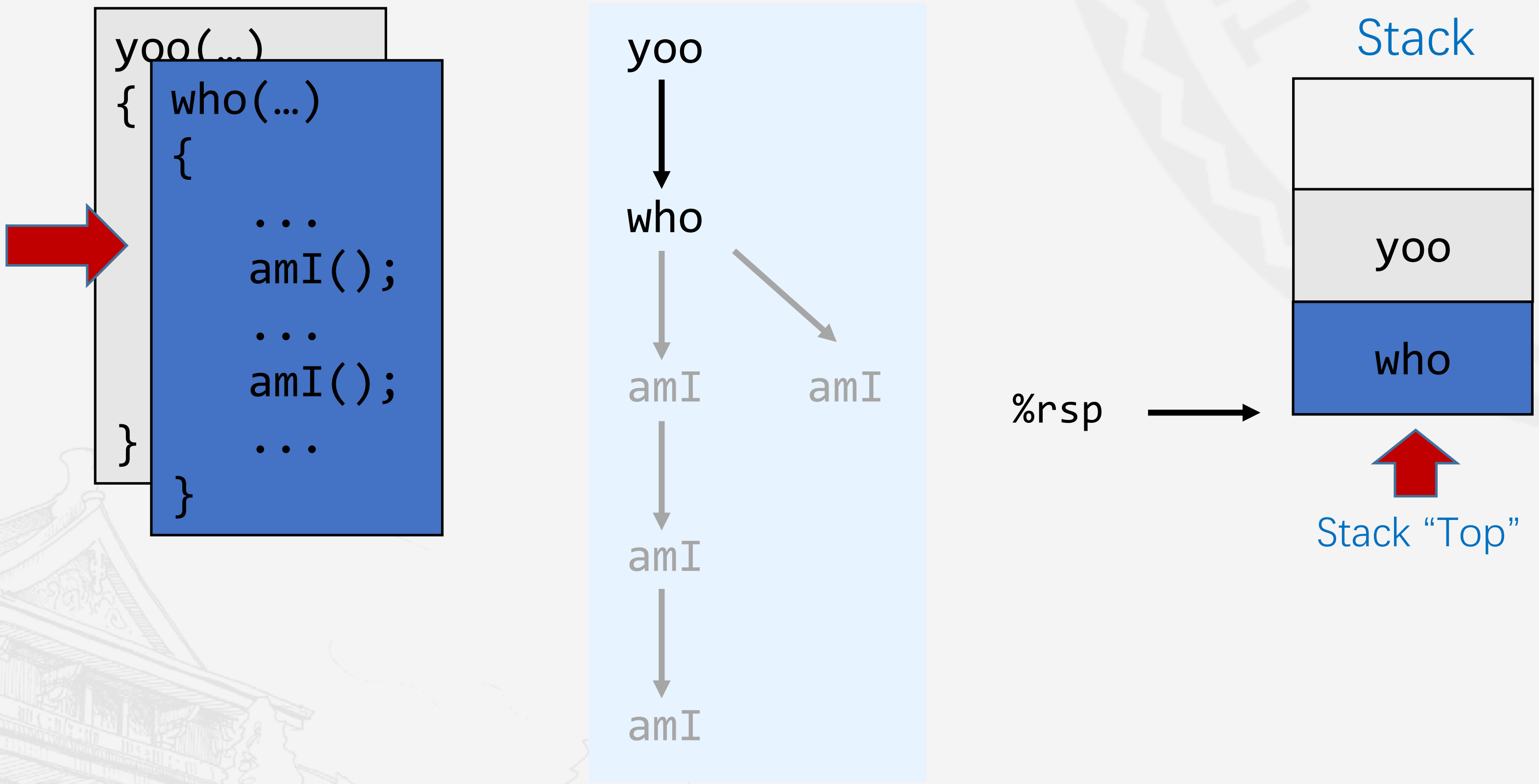




过程调用规范

Passing Control

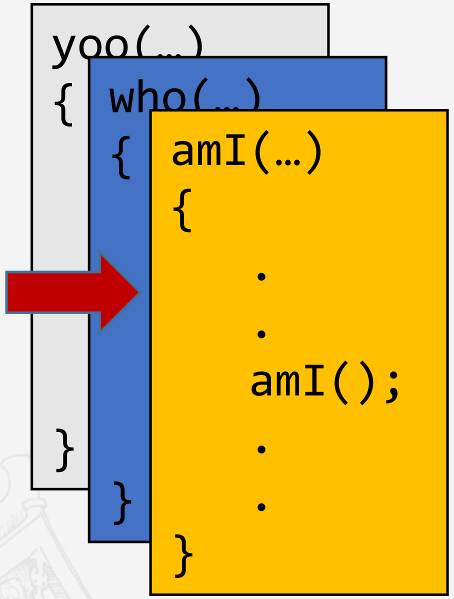
举例：调用链
Call Chain Example

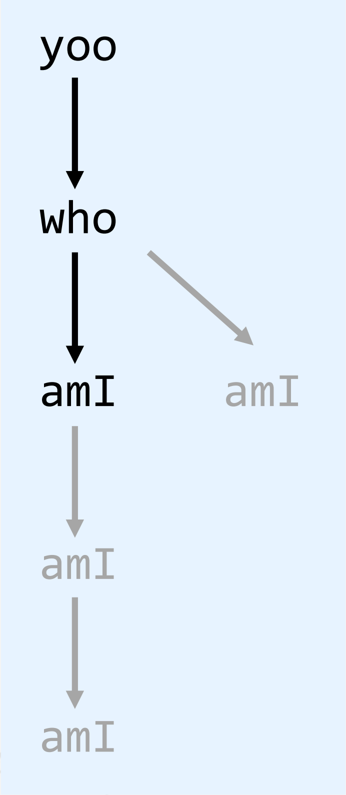


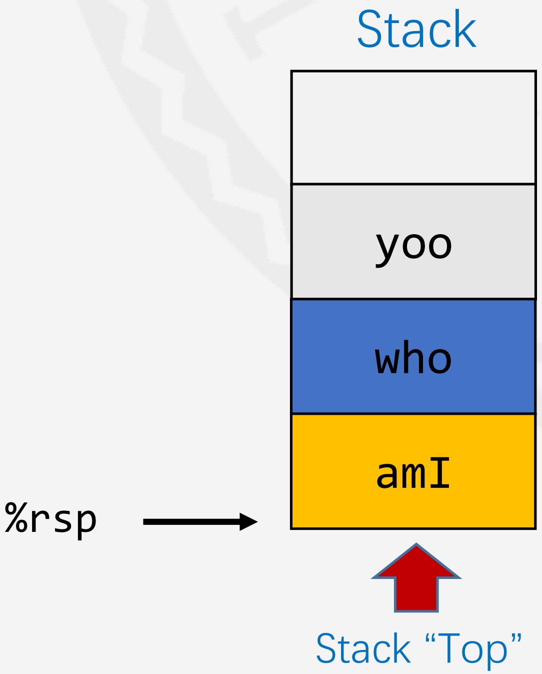
过程调用规范

Passing Control

举例：调用链
Call Chain Example



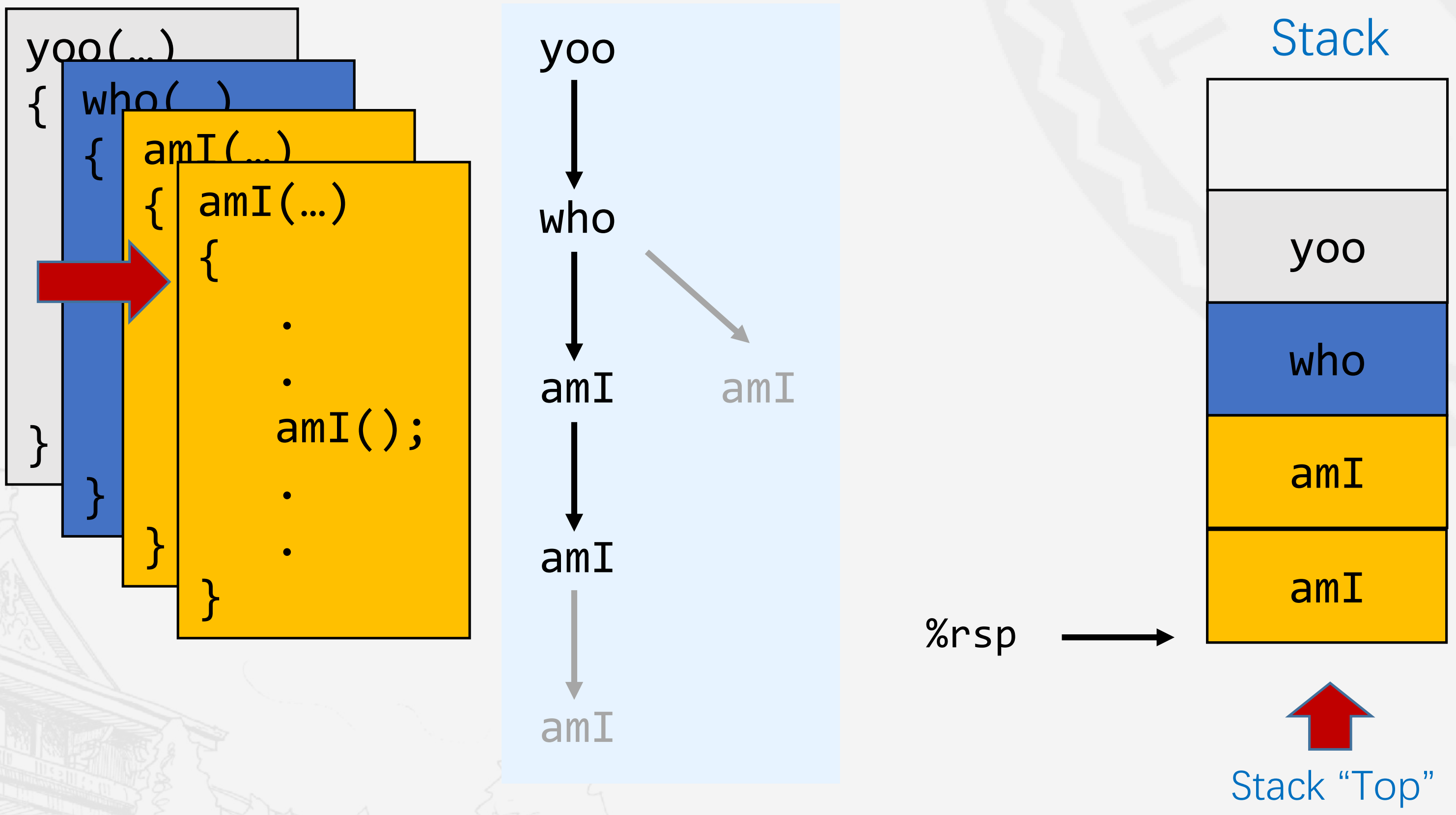




过程调用规范

Passing Control

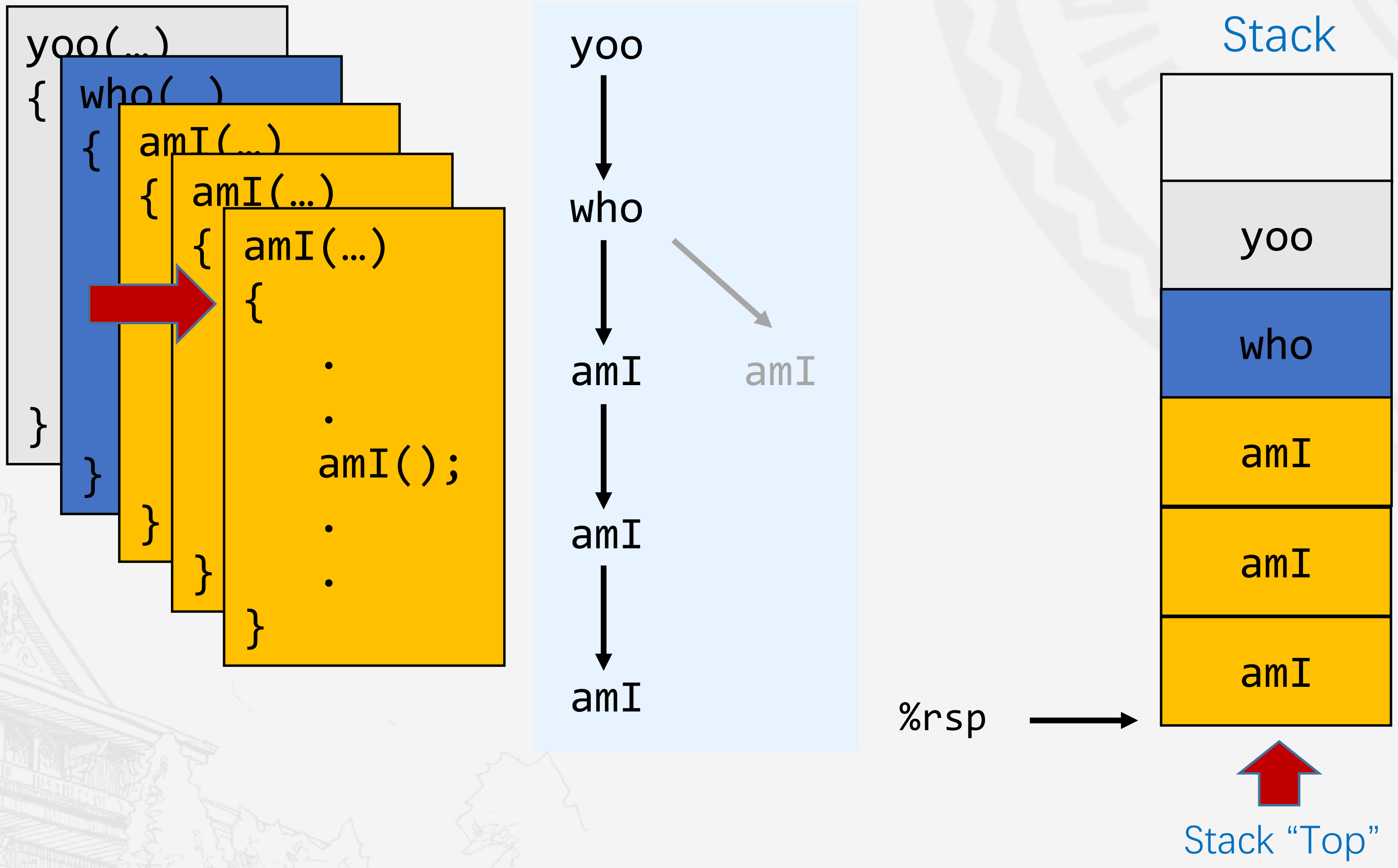
举例：调用链
Call Chain Example



过程调用规范

Passing Control

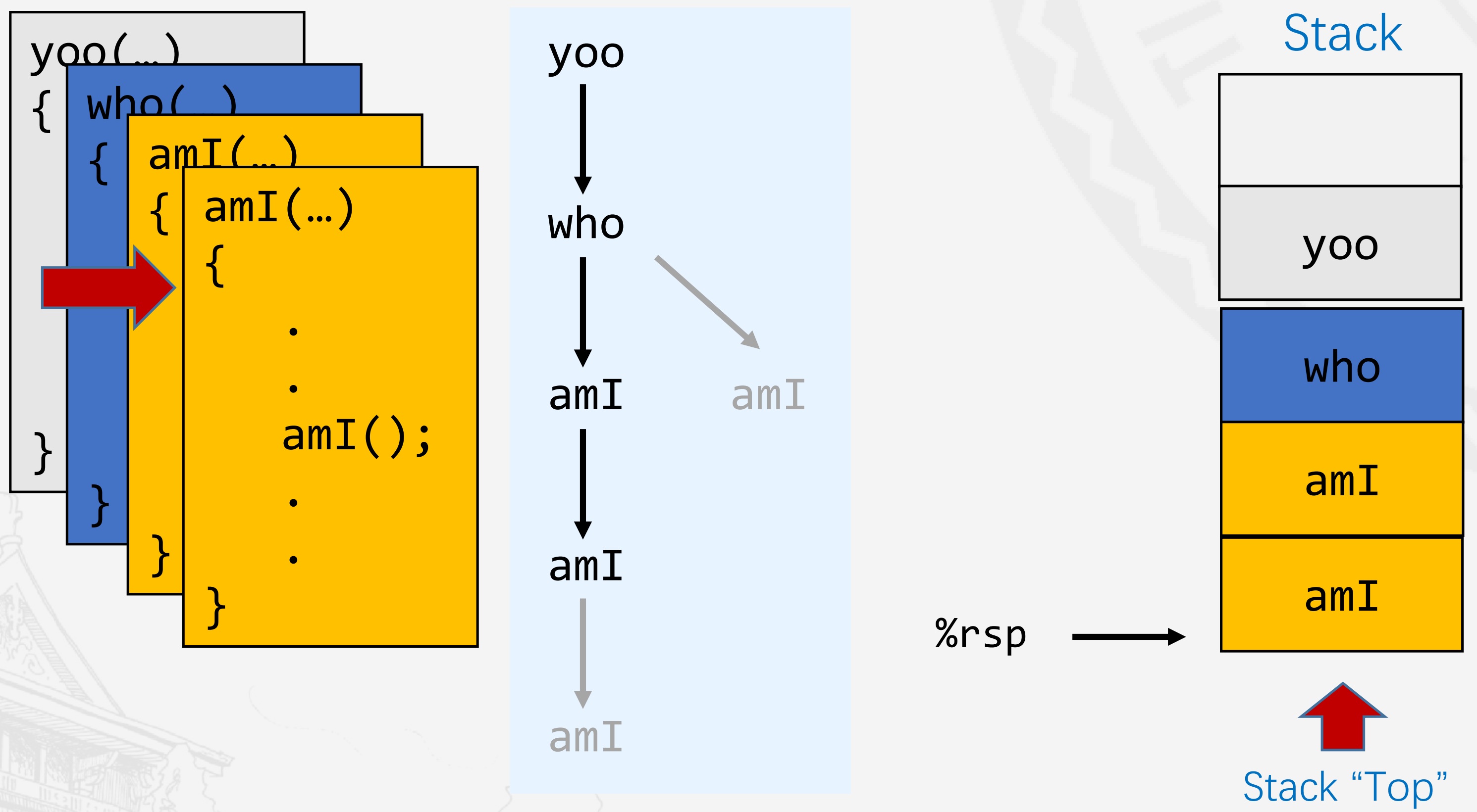
举例：调用链
Call Chain Example



过程调用规范

Passing Control

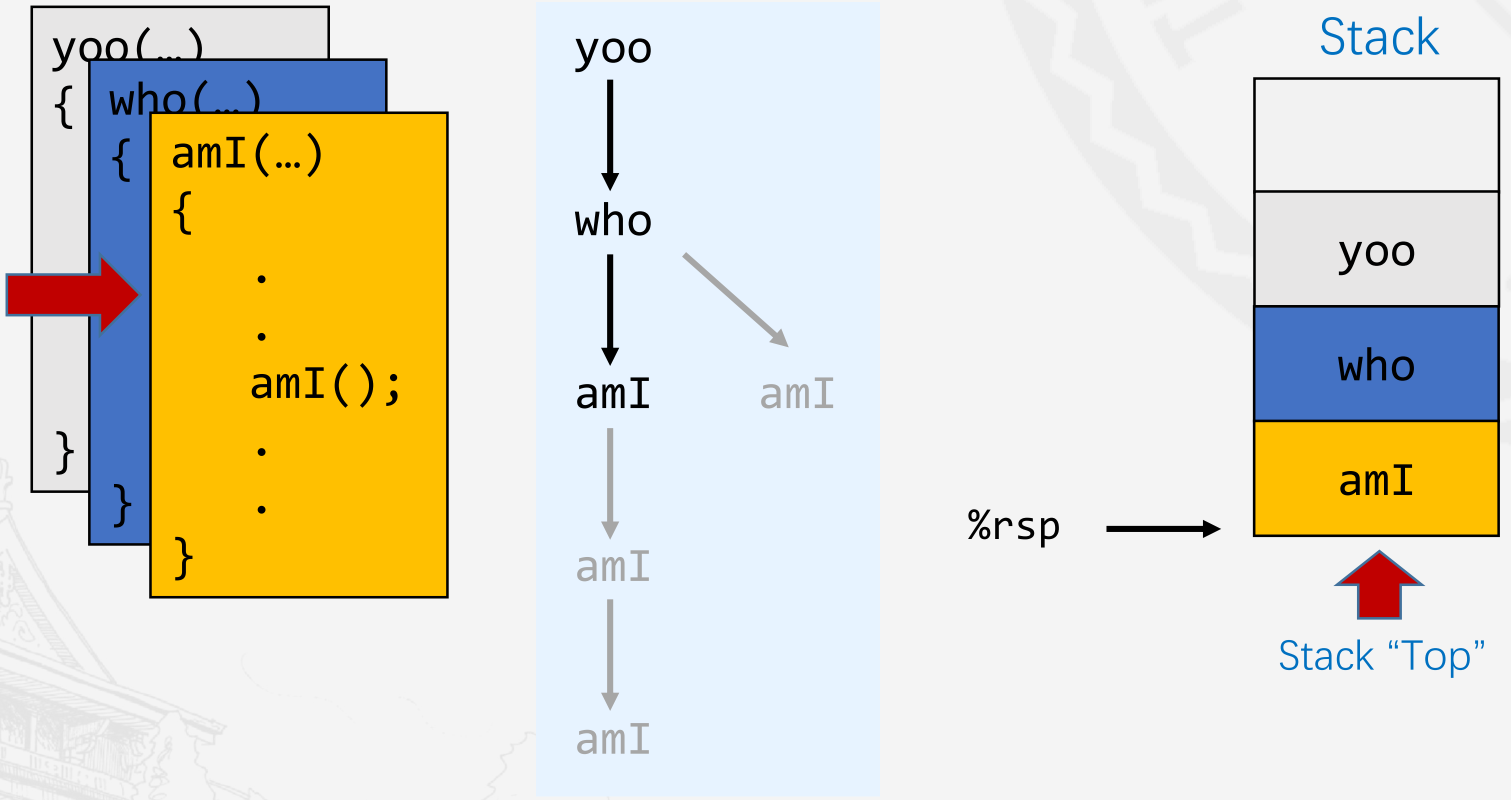
举例：调用链
Call Chain Example



过程调用规范

Passing Control

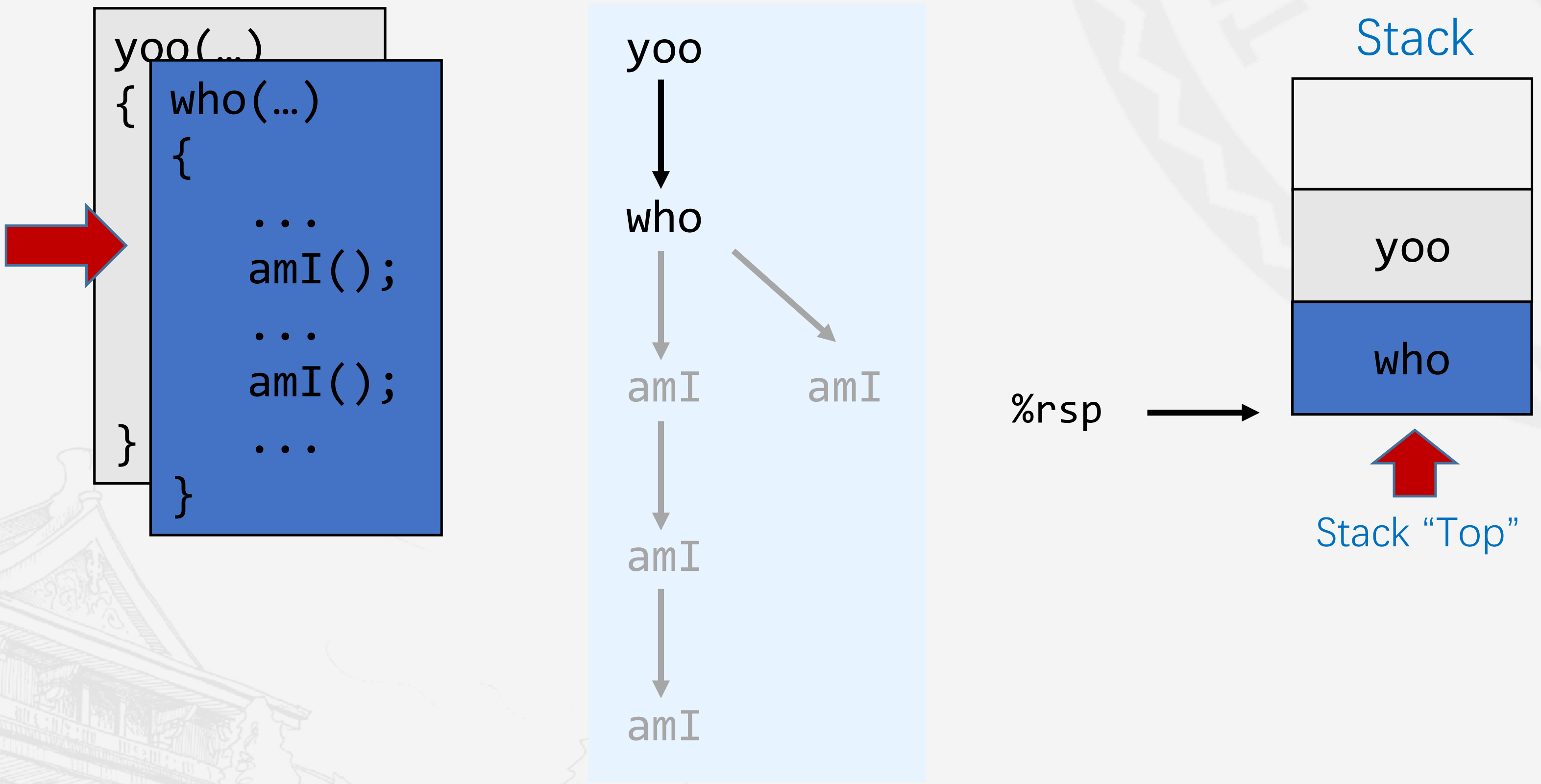
举例：调用链
Call Chain Example



过程调用规范

Passing Control

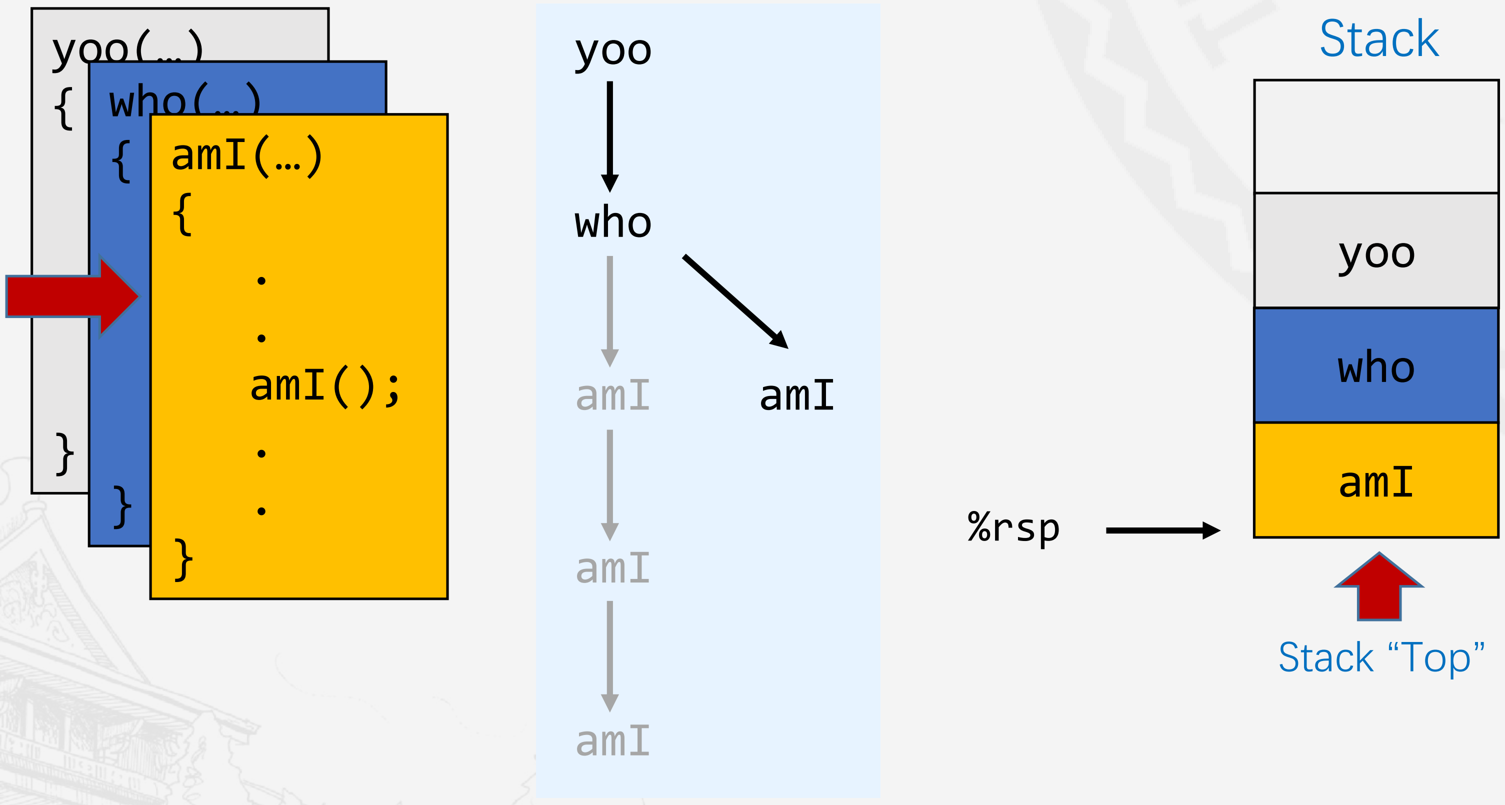
举例：调用链
Call Chain Example



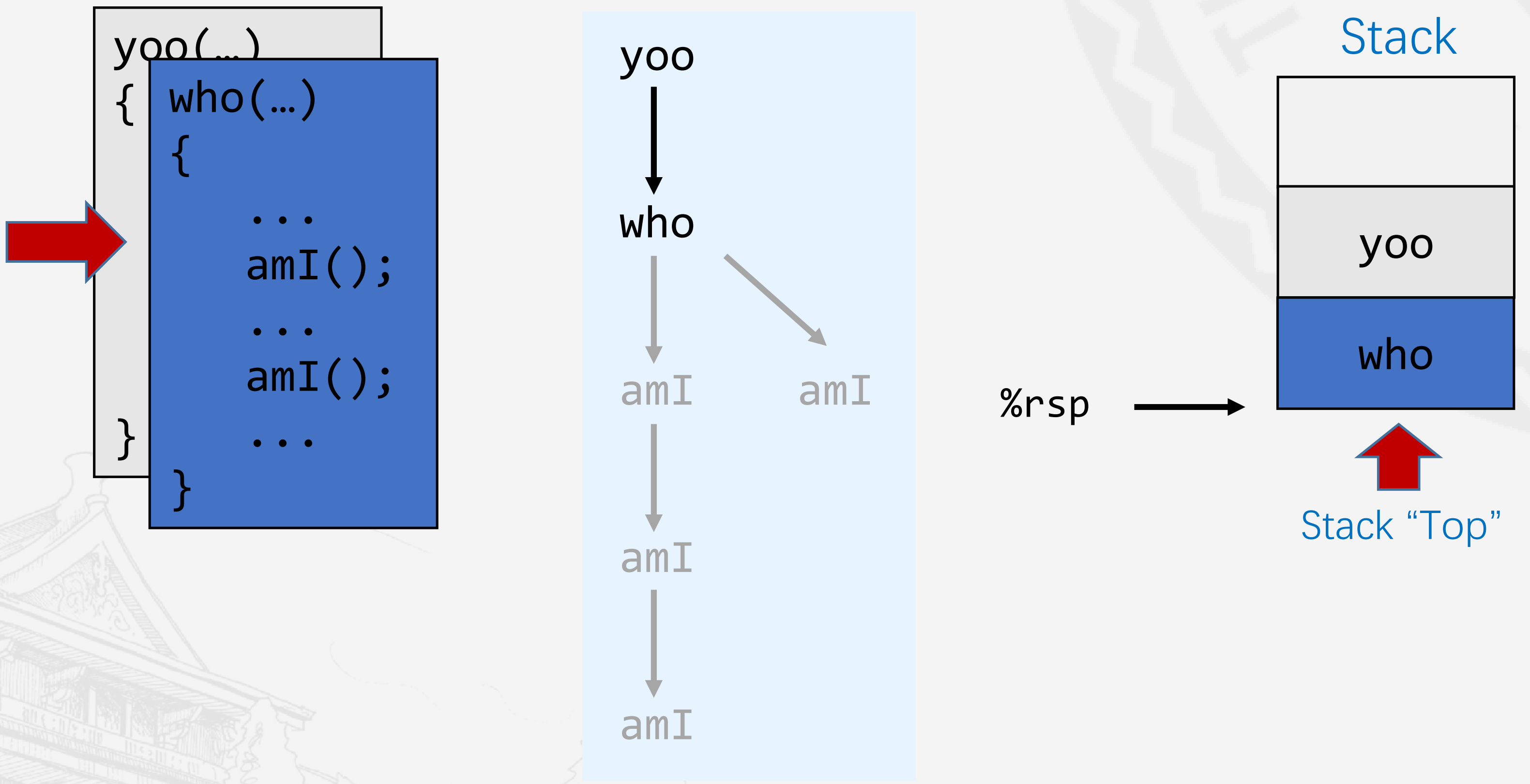
过程调用规范

Passing Control

举例：调用链
Call Chain Example



举例：调用链
Call Chain Example

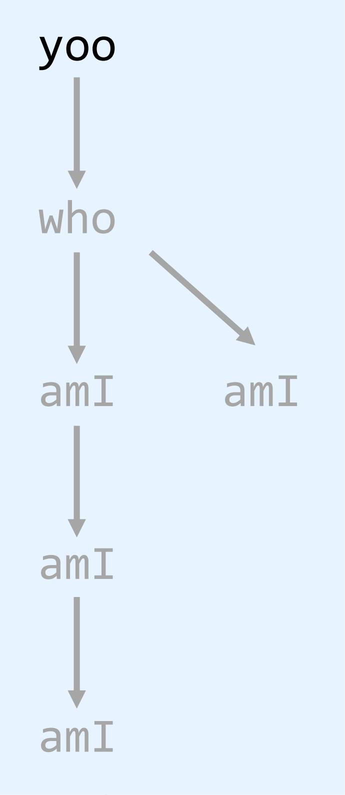


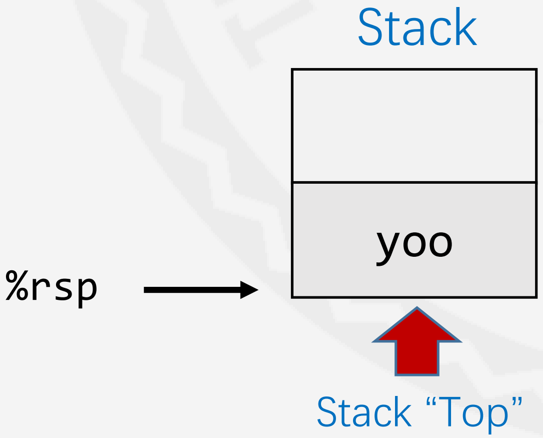
过程调用规范

Passing Control

举例：调用链
Call Chain Example

```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```



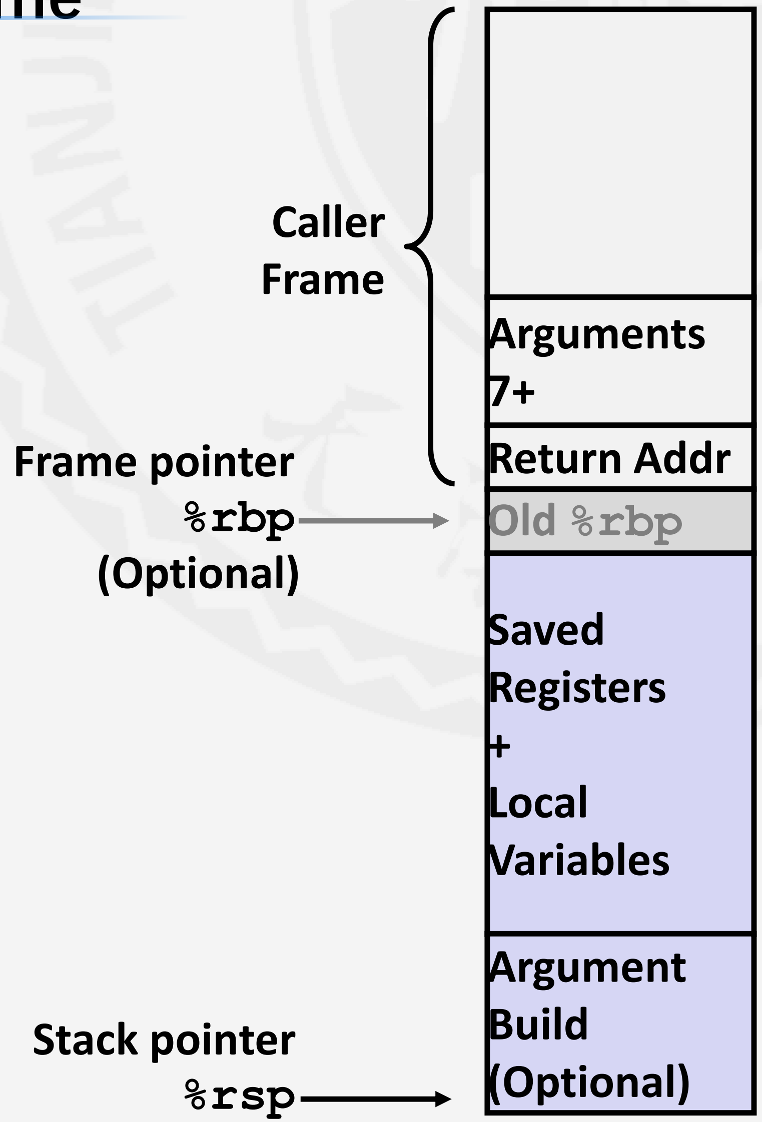


过程调用规范

Passing Control

x86-64/Linux 栈帧
x86-64/Linux Stack Frame

- 当前栈帧（从顶至底）
  Contents
  - 参数：即将要调用的函数的参数
    “Argument build:” Parameters for function about to call
  - 本地变量：在寄存器中保存不下的
    Local variables：If can’t keep in registers
  - 保存的寄存器上下文信息
    Saved register context
  - 指向调用者的栈帧底部的指针 （可选）
    Old frame pointer (optional)
- 调用者栈帧
  Caller Stack Frame
  - 返回地址
    Return address
    - 调用call指令时入栈
      Pushed by call instruction
  - 本次调用的参数
    Arguments for this call



过程调用规范

Passing Control

举例：incr 过程
Example: incr

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

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

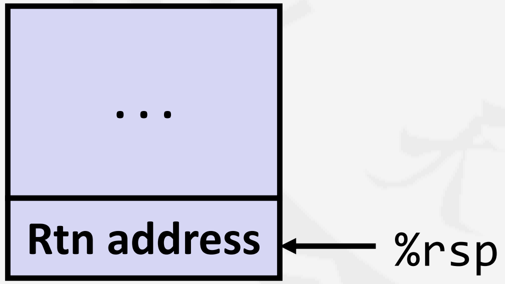
Register	Use(s)
%rdi	Argument **p**
%rsi	Argument **val**, **y**
%rax	**x**, Return value

举例：调用 incr #1
Example: Calling incr #1

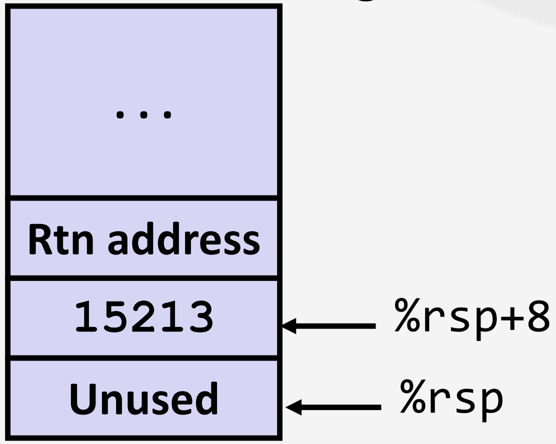
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

栈的初始结构
Initial Stack Structure



指令执行后的栈的结构
Resulting Stack Structure



过程调用规范

Passing Control

举例：调用 incr #2
Example: Calling incr #2

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

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure

...	
Rtn address	
15213	← %rsp+8
Unused	← %rsp

Register	Use(s)
%rdi	&v1
%rsi	3000

过程调用规范

Passing Control

举例：调用 incr #3
Example: Calling incr #3

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

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure

Stack	
...	
Rtn address	
18213	← %rsp+8
Unused	← %rsp

Register	Use(s)
%rdi	&v1
%rsi	3000

过程调用规范

Passing Control

举例：调用 incr #4
Example: Calling incr #4

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

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure

...	
Rtn address	
18213	← %rsp+8
Unused	← %rsp

Register	Use(s)
%rax	Return value

更新后栈的结构

Updated Stack Structure

...	
Rtn address	← %rsp

过程调用规范

Passing Control

举例：调用 incr #5
Example: Calling incr #5

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

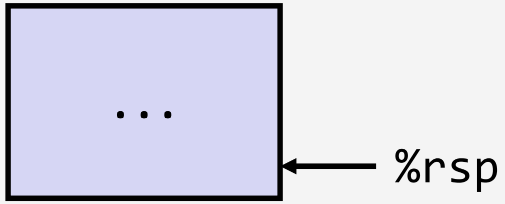
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure

Register	Use(s)
%rax	Return value

栈的最终结构

Final Stack Structure



寄存器使用惯例
Register Saving Conventions

- 寄存器可以用做临时存储吗?
  Can register be used for temporary storage?

```
yoo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

- %rdx 的内容会被 who 覆盖
  Contents of register %rdx overwritten by who
- 这会引发逻辑错误 ➙ 需要做点什么
  This could be trouble ➙ something should be done!
  - 建立一种使用寄存器的协调机制
    Need some coordination

过程调用规范

Passing Control

寄存器使用惯例
Register Saving Conventions

- 当 yoo 调用 who:
  When procedure yoo calls who:
  - yoo是调用者
    yoo is the caller
  - Who是被调用者
    who is the callee

- **“调用者保护”**
  “Caller Saved”
  - 在调用前，调用者把临时数据保存到自己的栈帧中
    Caller saves temporary values in its frame before the call
- **“被调用者保护”**
  “Callee Saved”
  - 被调用者在使用寄存器前将降临时数据保存至自己的栈帧中
    Callee saves temporary values in its frame before using
  - 在被调用者返回前恢复这些数据
    Callee restores them before returning to caller

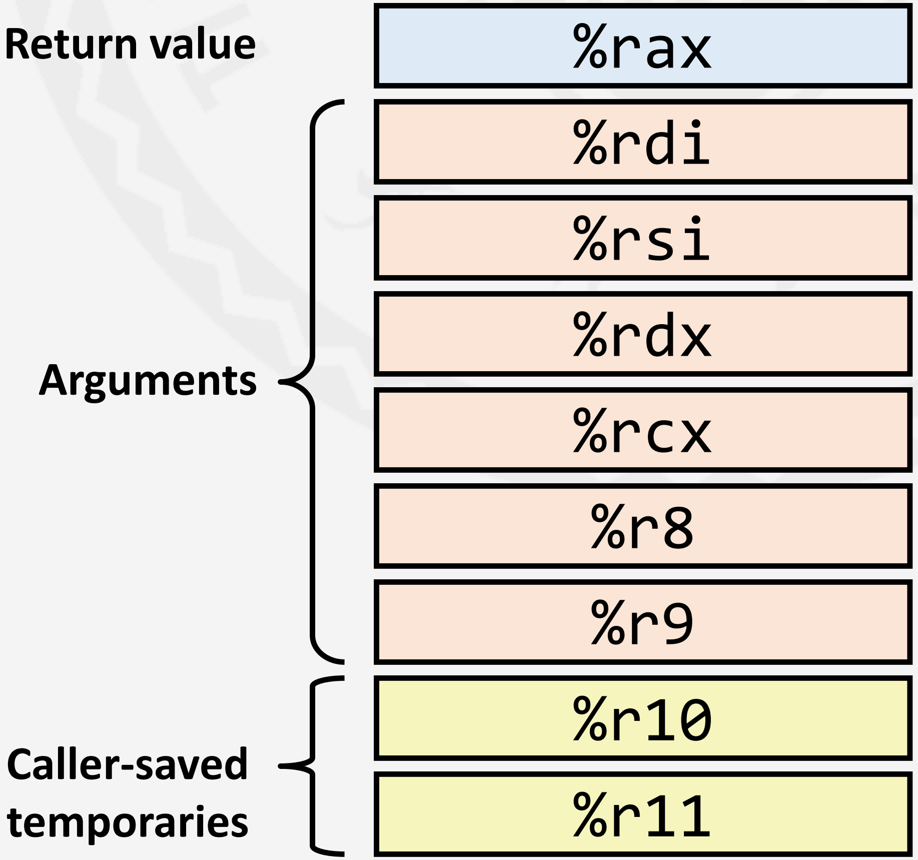
过程调用规范

Passing Control

x86-64 Linux 中寄存器的用途 #1
x86-64 Linux Register Usage #1

- %rax, %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11
  - 调用者保护
    Caller-saved
  - 在过程中值可能被修改
    Can be modified by procedure

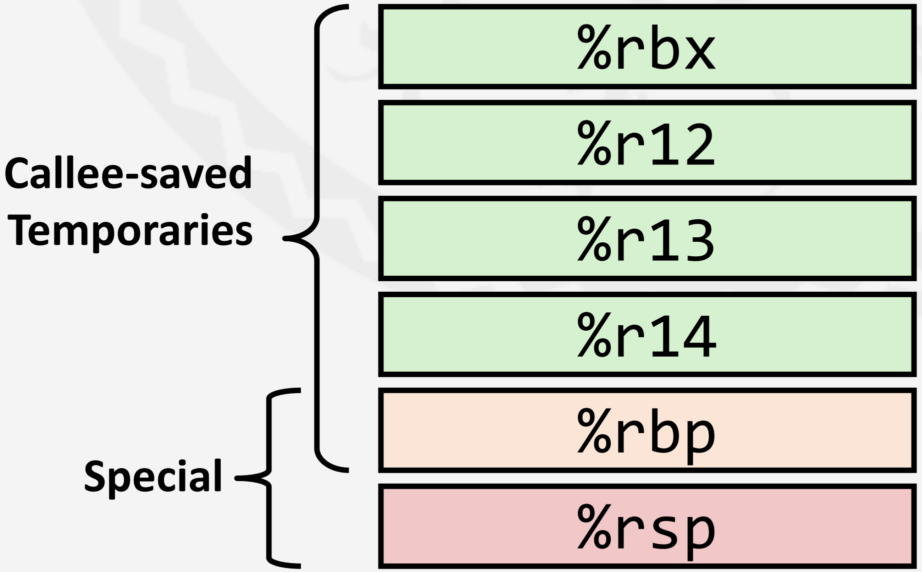
- %rax
  - 返回值
    Return value
- %rdi, ..., %r9
  - 参数
    Arguments



x86-64 Linux 中寄存器的用途 #2
x86-64 Linux Register Usage #2

- `%rbx, %r12, %r13, %r14, %rbp, %rsp`
  - 被调用者保护
    Callee-saved
  - 被调用者必须保护和恢复（如果被调用者使用）
    Callee must save & restore

- `%rbp`
  - 可以用于存储指向栈底指针
    May be used as frame pointer
  - 也可用用作普通临时数据的存储
    Arguments
- `%rsp`
  - 特殊的调用者保护形式
    Special form of callee save
  - 过程返回前恢复
    Restored to original value upon exit from procedure



过程调用规范

Passing Control

举例：被调用者保护 #1
Callee-Saved Example #1

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

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure

Stack	
...	
Rtn address	← %rsp

Resulting Stack Structure

Stack	
...	
Rtn address	
Saved %rbx	
15213	← %rsp+8
Unused	← %rsp

过程调用规范

Passing Control

举例：被调用者保护 #2
Callee-Saved Example #2

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

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Resulting Stack Structure

Stack	
. . .	
Rtn address	
Saved %rbx	
15213	← %rsp+8
Unused	← %rsp

Pre-return Stack Structure

Stack	
. . .	
Rtn address	← %rsp

本章内容

Topic

- 栈的结构

  Stack Structure

- 过程调用规范

  Calling Conventions

- 递归

  Recursion

递归函数
Recursive Function

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

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数的出口
Recursive Function Terminal Case

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

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数的寄存器保护
Recursive Function Register Save

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

Register	Use(s)	Type
%rdi	x	Argument

...
Rtn address
Saved %rbx ← %rsp

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数的调用前的准备
Recursive Function Call Setup

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

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数的调用
Recursive Function Call

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

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归

Recursion

递归函数的结果
Recursive Function Result

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

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

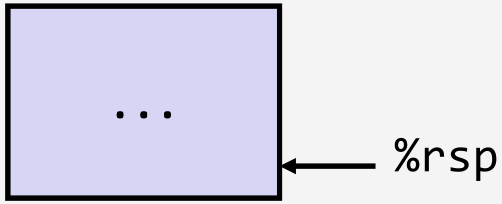
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数的结束
Recursive Function Completion

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

Register	Use(s)	Type
%rax	Return value	Return value

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```



对递归函数的观察
Observations About Recursion

- 和普通函数调用相比并没有什么特殊的处理
  Handled Without Special Consideration
  - 栈帧：每次函数调用都会分配一个私有的存储空间
    Stack frames mean that each function call has private storage
    - 保存寄存器和局部变量
      Saved registers & local variables
    - 保存返回地址
      Saved return pointer
  - 寄存器使用惯例保证了一次函数调用不会破坏其他函数的数据
    Register saving conventions prevent one function call from corrupting another's data
  - 栈与过程调用返回在工作模式上完美契合
    Stack discipline follows call / return pattern
    - 如果 P 调用 Q，则 Q 先于 P 返回
      If P calls Q, then Q returns before P
    - 后进先出
      Last-In, First-Out

- 同样也适用于相互递归
  Also works for mutual recursion
  - P 调用 Q；Q 调用 P
    P calls Q; Q calls P

X86-64 过程总结
x86-64 Procedure Summary

- 要点
  Important Points
- 对于过程的调用与返回，栈是一种恰当的数据结构
  Stack is the right data structure for procedure call / return
  - 如果 P 调用 Q，则 Q 先于 P 返回
    If P calls Q, then Q returns before P
- 递归和普通函数调用的处理方式相同
  Recursion (& mutual recursion) handled by normal calling conventions
- 指针是地址的值
  Pointers are addresses of values