概述

计算机系统基础

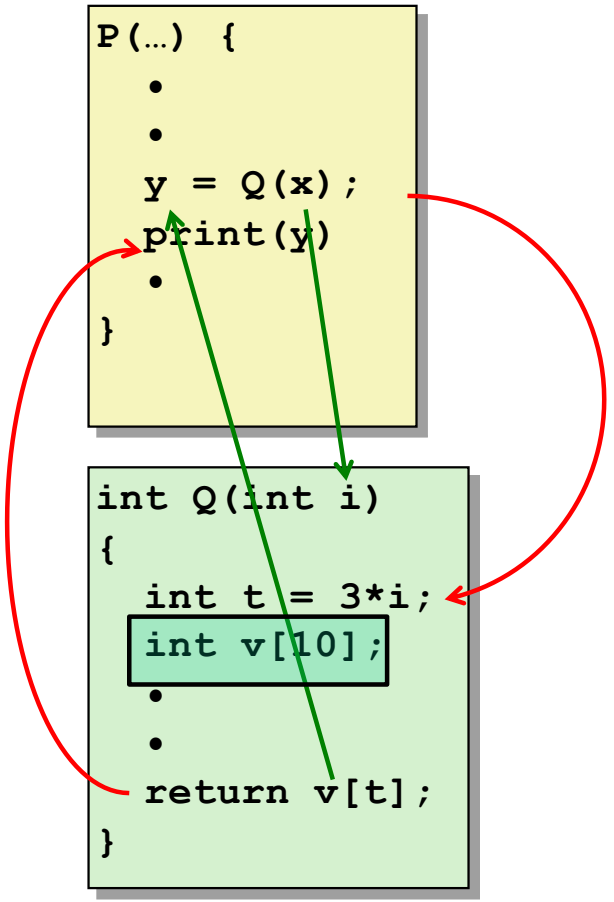
任课教师：
龚奕利
yiligong@whu.edu.cn

本节内容

- 过程
  - 栈的结构
  - 调用的惯例
    - 控制转移
    - 数据传送
    - 管理局部数据
  - 递归

过程调用的机制

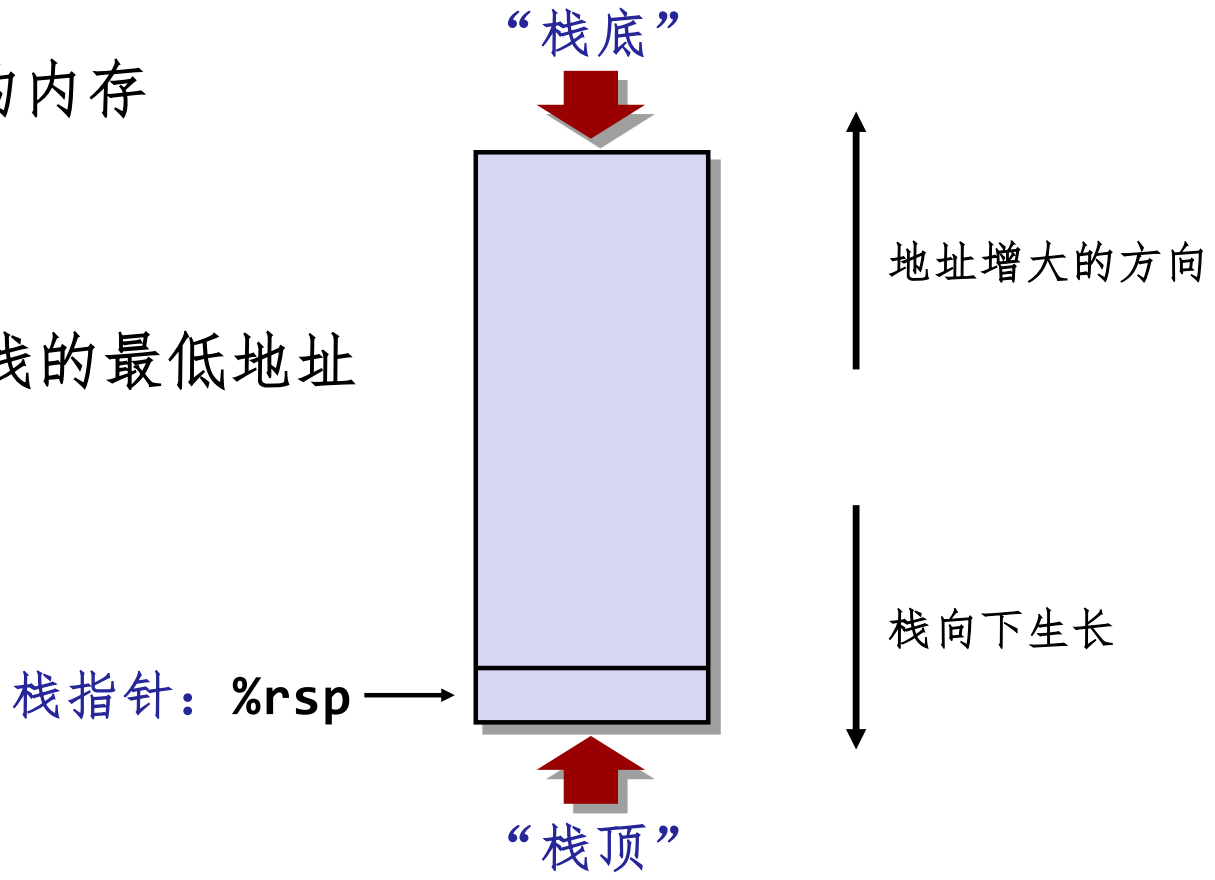
- 控制转移
  - 进入到过程代码的开始
  - 返回到返回点
- 数据传送
  - 过程参数
  - 返回值
- 内存管理
  - 程序执行时分配
  - 返回时释放
- 这些机制均由机器指令实现
- **x86-64** 的过程实现只使用了必须的机制



x86-64 的栈

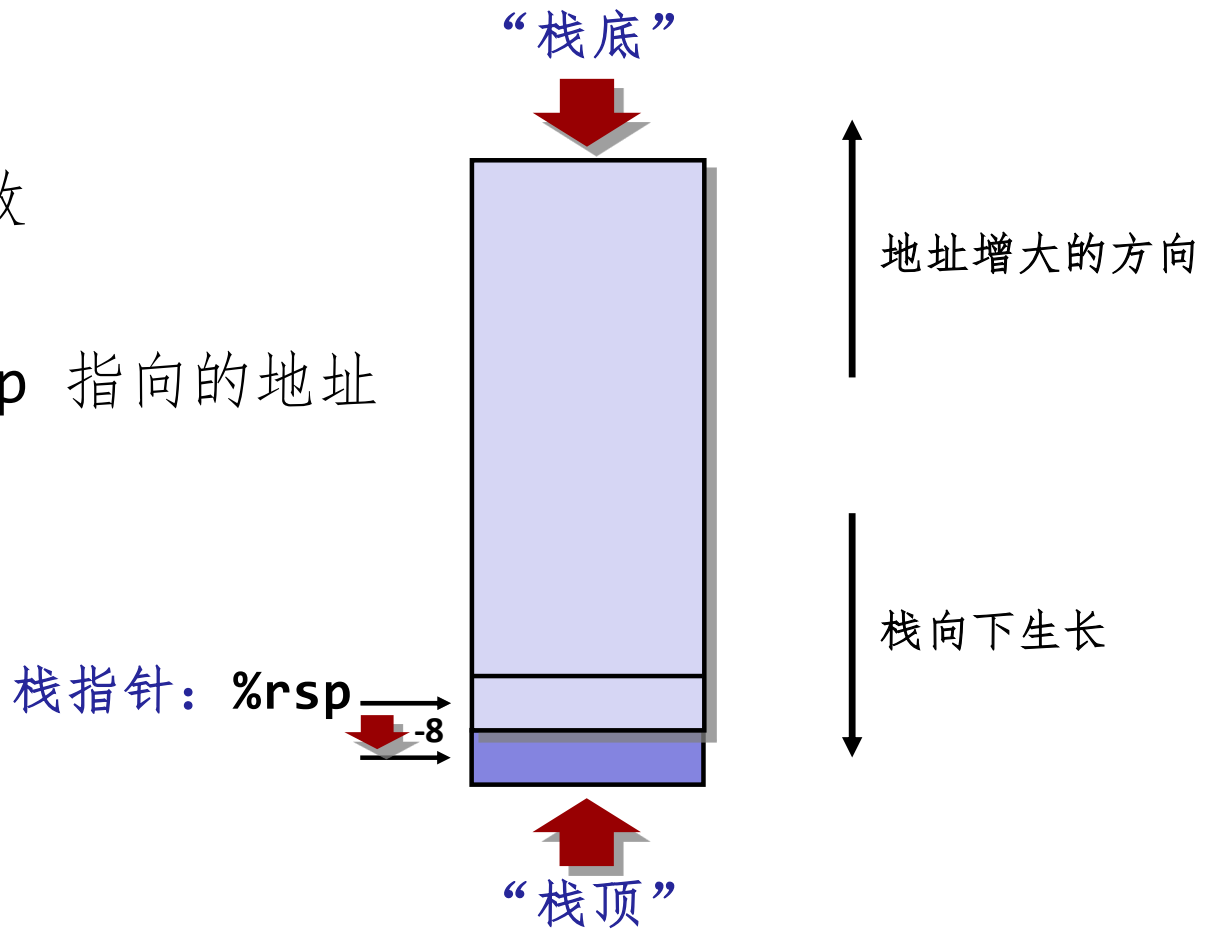
- 使用栈的规则管理的内存
- 向低地址生长

- 寄存器 %rsp 保存栈的最低地址
  - “栈顶”元素的地址



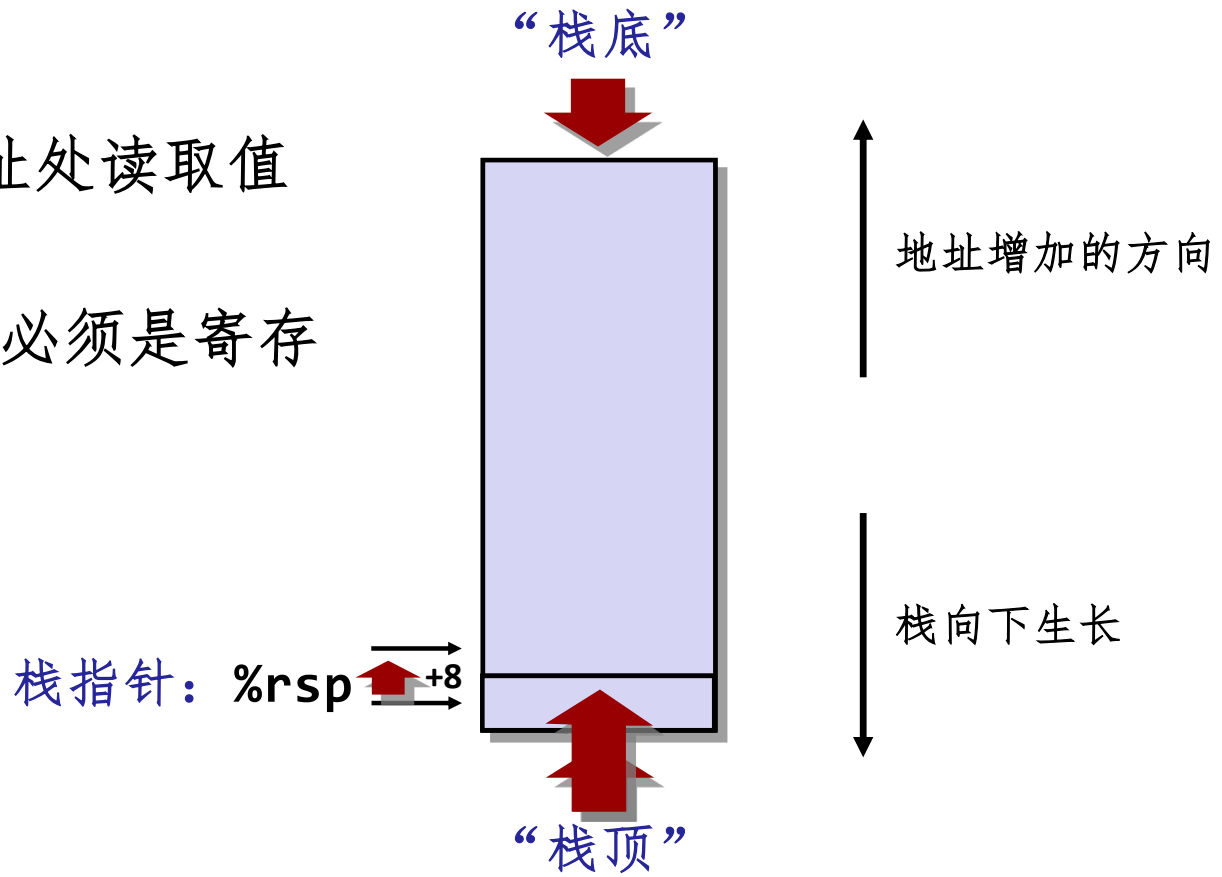
x86-64 的栈：Push

- pushq Src
  - 从 Src 获取操作数
  - %rsp 的值减8
  - 将操作数写入 %rsp 指向的地址



x86-64 的栈：Pop

- popq Dest
  - 从 %rsp 指向的地址处读取值
  - %rsp 的值加8
  - 将值存储在 Dest（必须是寄存器）中



本节内容

- 过程
  - 栈的结构
  - 调用的惯例
    - 控制转移
    - 数据传送
    - 管理局部数据
  - 递归

过程调用代码示例

```
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
```

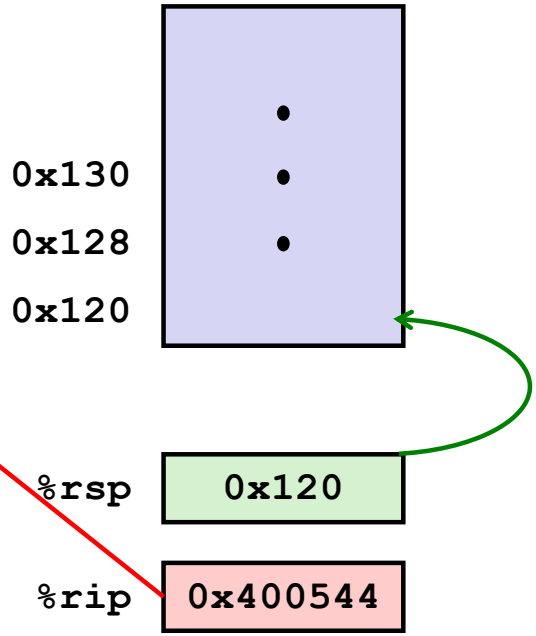
过程控制流

- 用栈支持过程调用和返回
- 过程调用：**call label**
  - 将返回地址压入栈中
  - 跳转到 label 处
- 返回地址
  - 紧接着调用后的下一条指令的地址
- 过程返回：**ret**
  - 从栈中弹出地址
  - 跳转到该地址处

控制流示例 #1

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

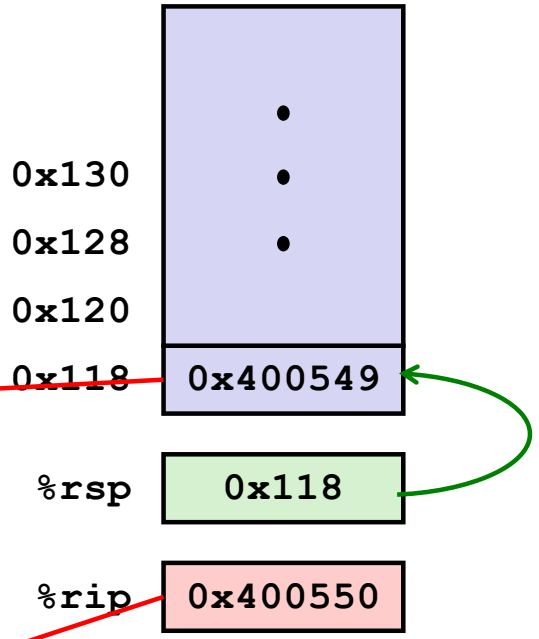
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



控制流示例 #2

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

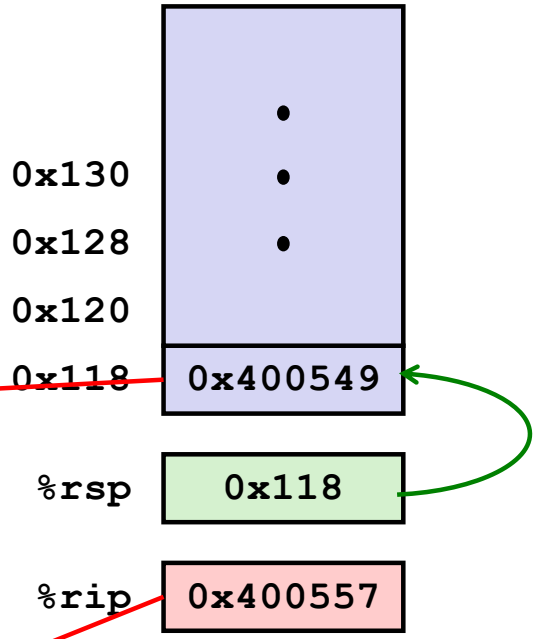
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



控制流示例 #3

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

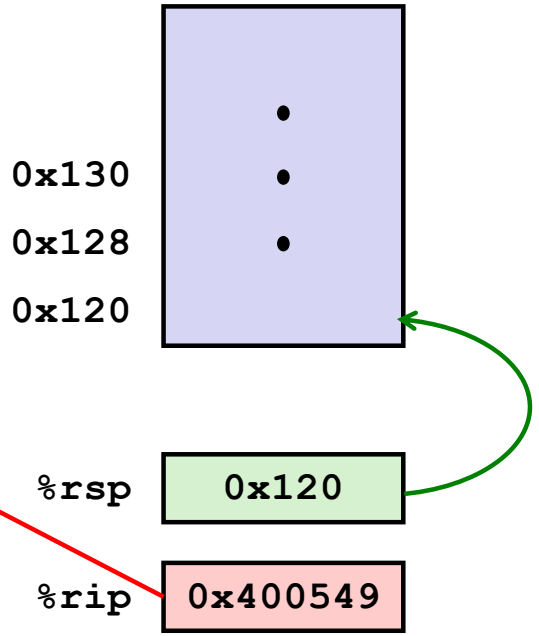
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



控制流示例 #4

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

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



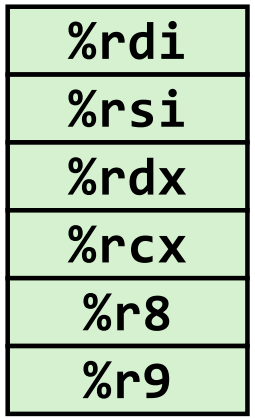
本节内容

- 过程
  - 栈的结构
  - 调用的惯例
    - 控制转移
    - 数据传送
    - 管理局部数据
  - 递归

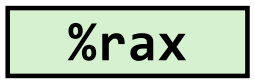
过程调用数据流

寄存器

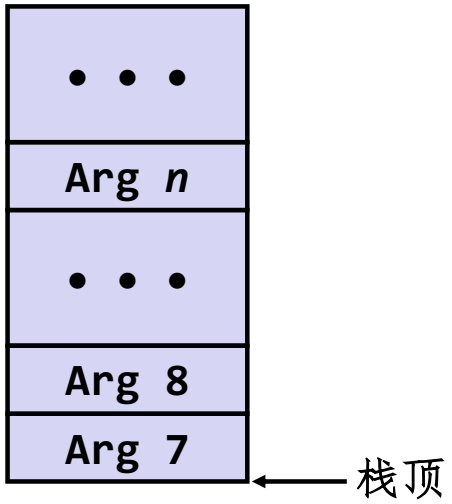
- 前6个变量



- 返回值



栈



- 通过栈传递参数时，每个数据的大小都会向上取整到8的倍数
- 只有当需要时才分配栈的空间

过程调用数据流示例

```
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
```

本节内容

- 过程
  - 栈的结构
  - 调用的惯例
    - 控制转移
    - 数据传送
    - 管理局部数据
  - 递归

基于栈的语言

- 支持递归的语言
  - 例如：C, Pascal, Java
  - 代码需要是“可重入的”
    - 单个过程同时有多个实例
  - 需要空间存储每一个实例的状态
    - 参数
    - 局部变量
    - 返回地址
- 栈规则
  - 某个过程的状态只需要保存有限的时间
    - 从被调用到返回
  - 被调用者在调用者之前返回
- 栈以栈帧的形式进行分配
  - 单个过程实例的状态

斐波那契序列：

```
int fib(int n)
{
    if (n == 1 || n == 2)
        return 1;
    return fib(n-1)+fib(n-2);
}
```

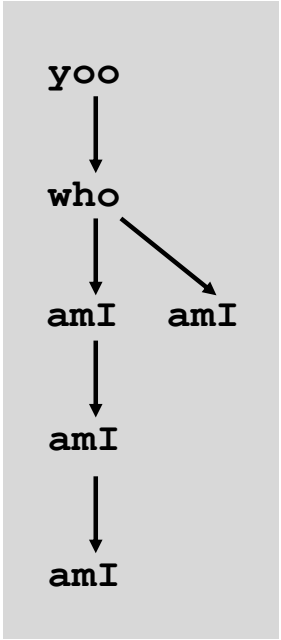
调用链示例

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

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

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

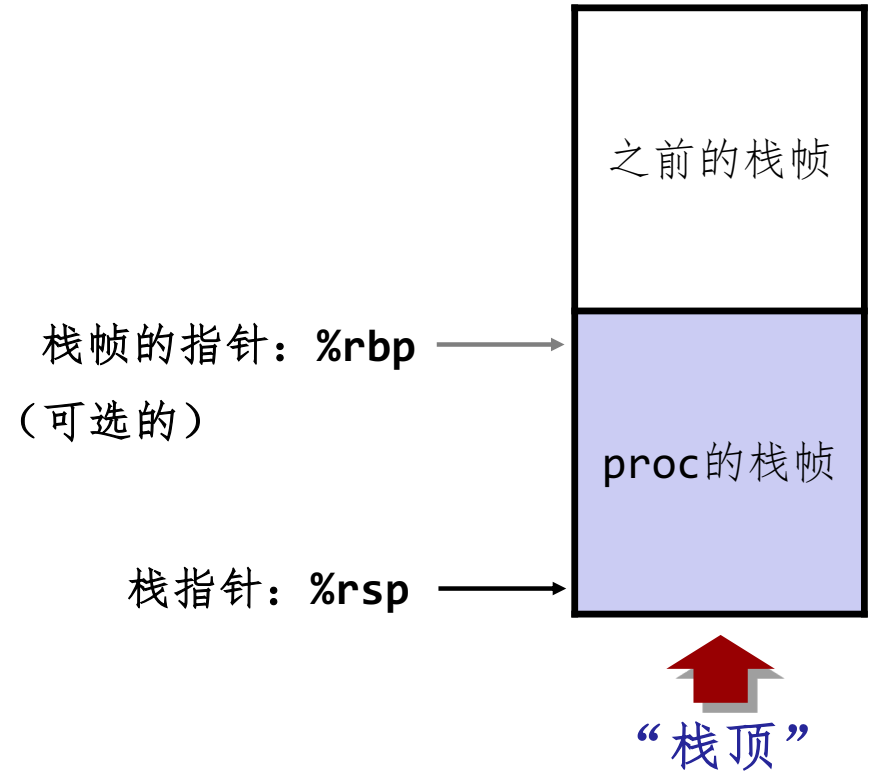
调用链



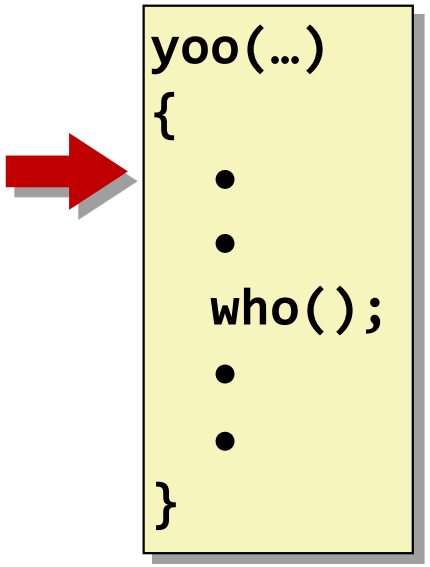
程序amI()是递归的

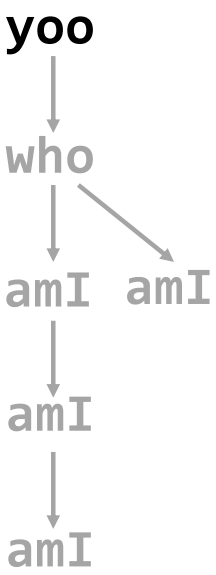
栈帧

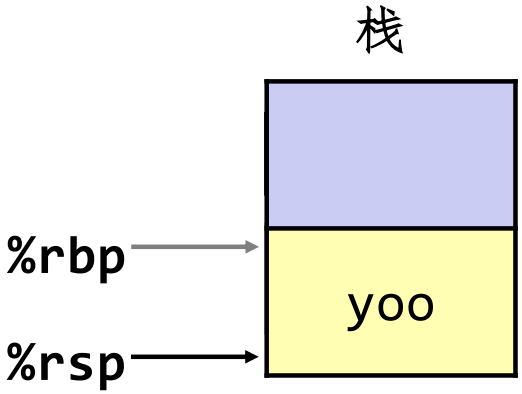
- 内容
  - 返回信息
  - 局部存储（如果需要）
  - 临时空间（如果需要）
- 管理
  - 进入过程时分配的空间
    - “准备工作” 代码
    - 包括 **call** 指令压入的数据
  - 返回时释放空间
    - “结束工作” 代码
    - 包括 **ret** 指令弹出的数据



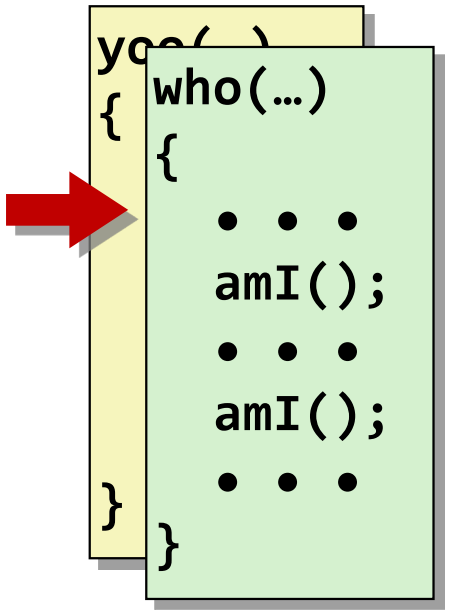
示例

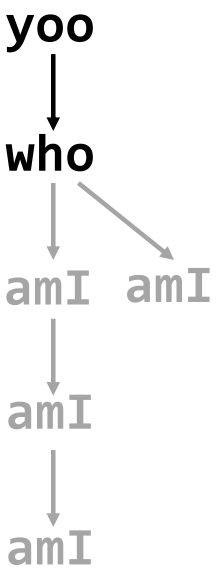


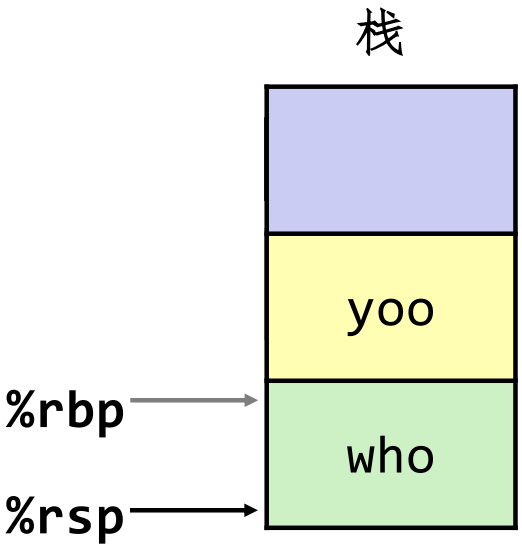




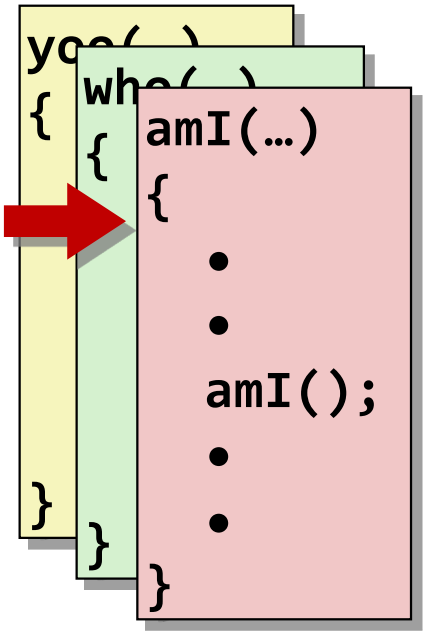
示例

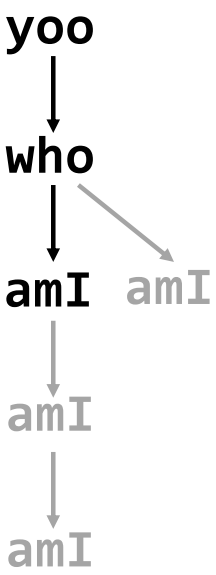


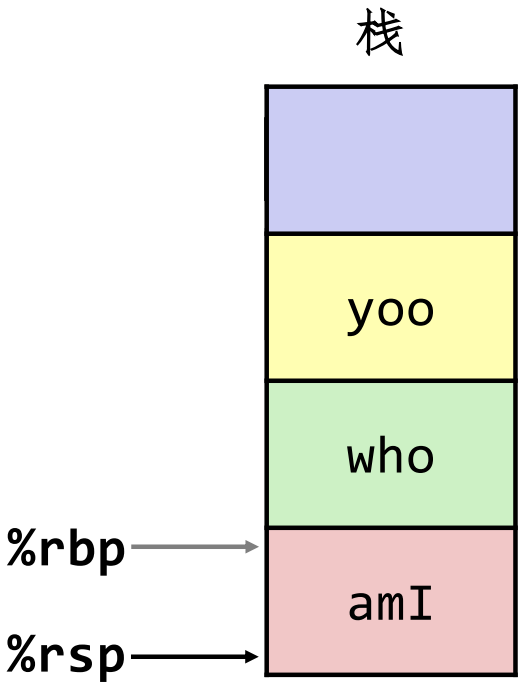




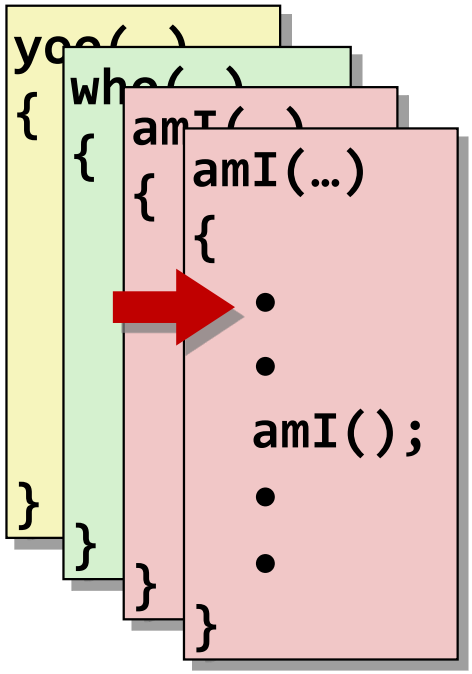
示例

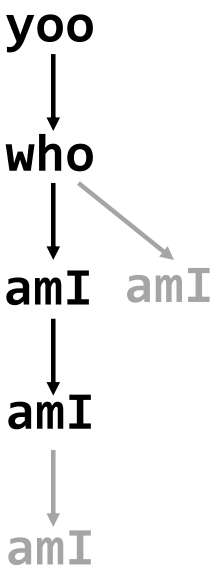


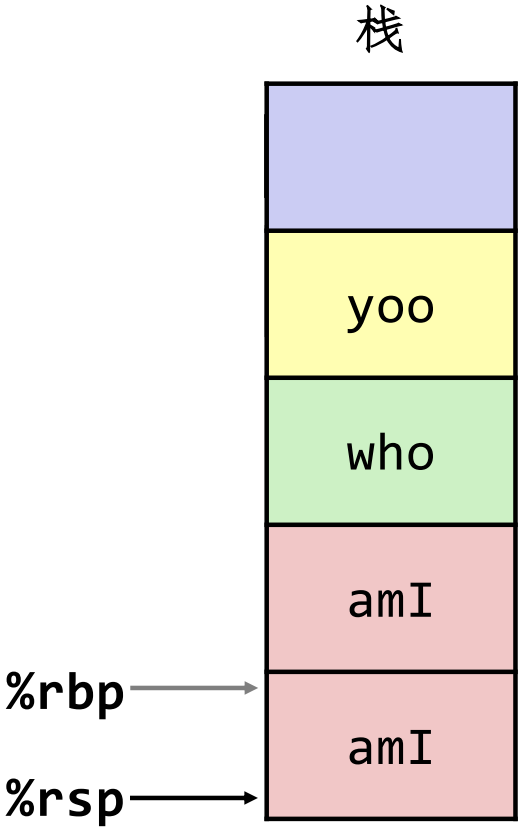




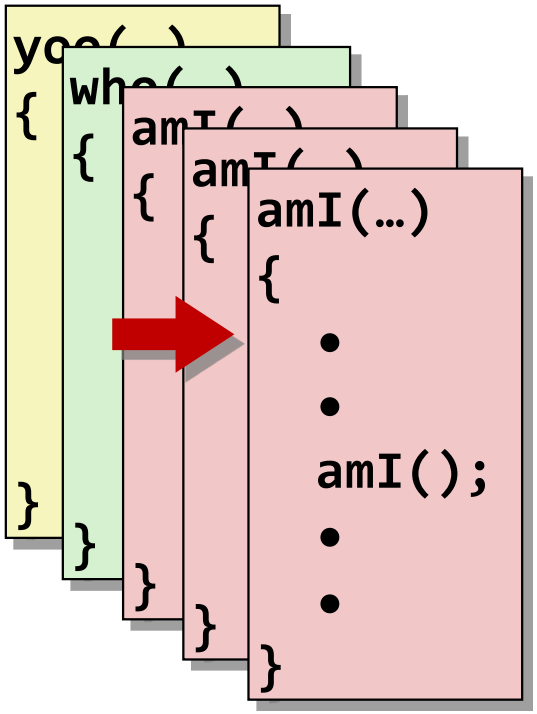
示例

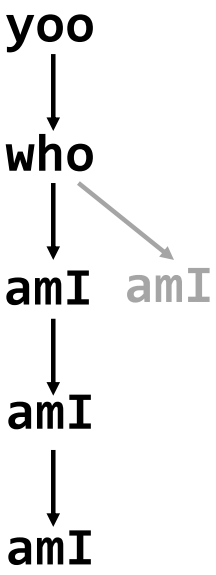


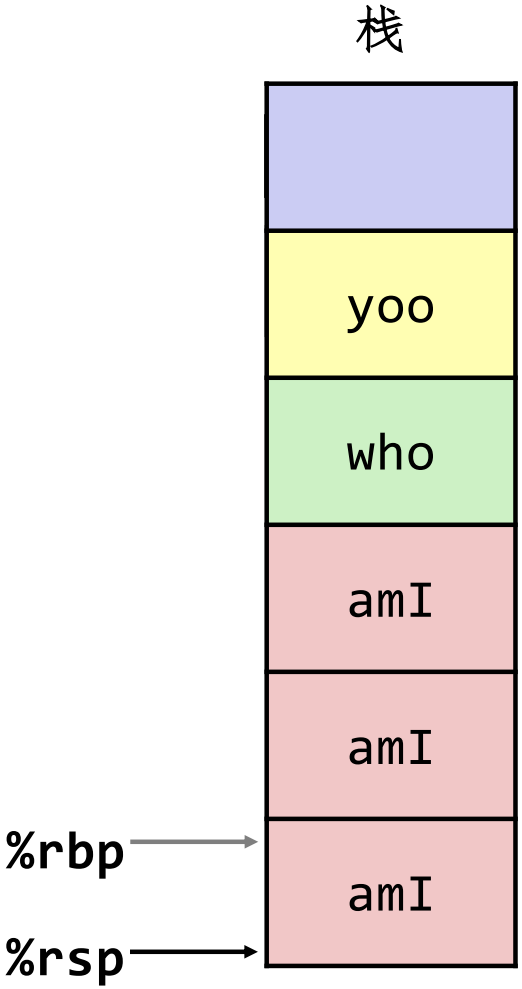




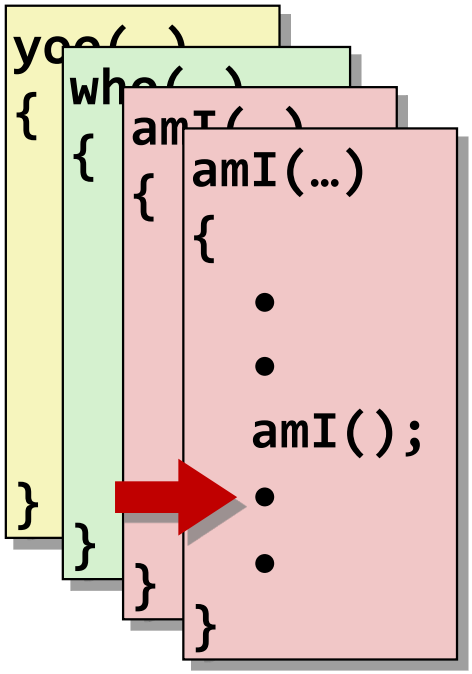
示例

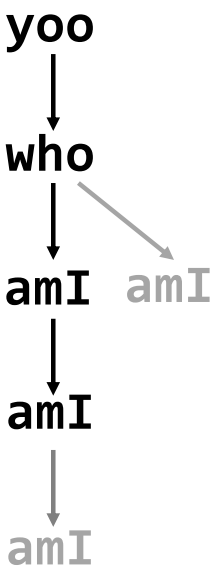


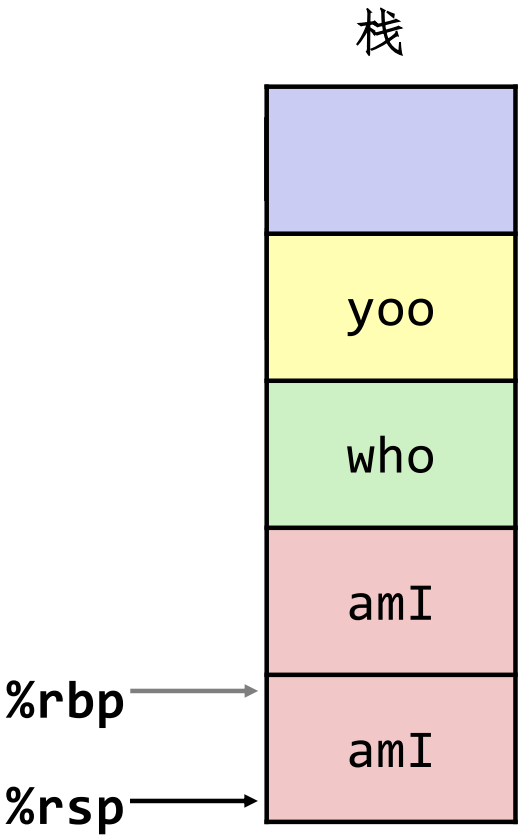




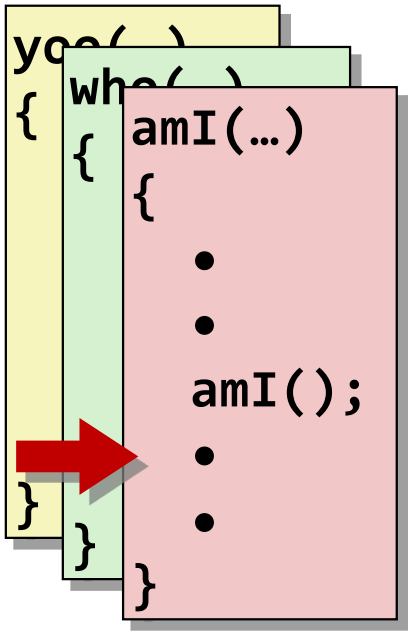
示例

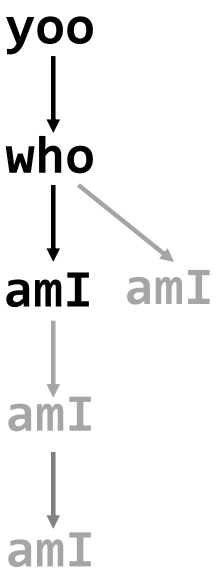


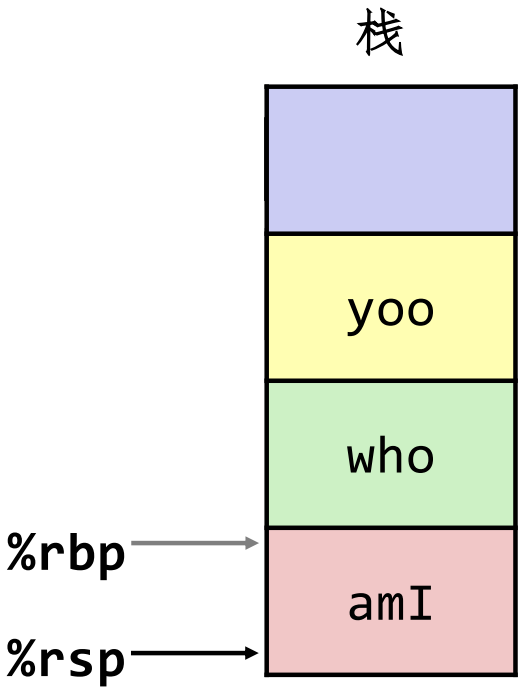




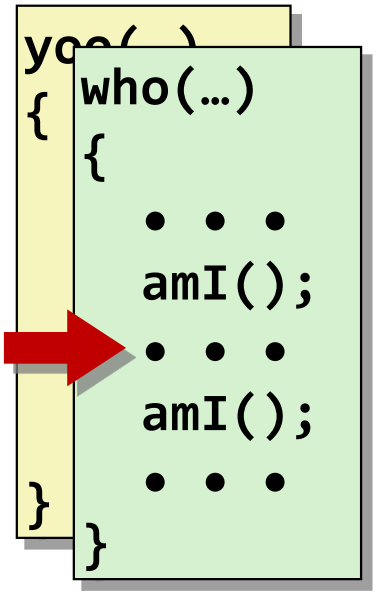
示例

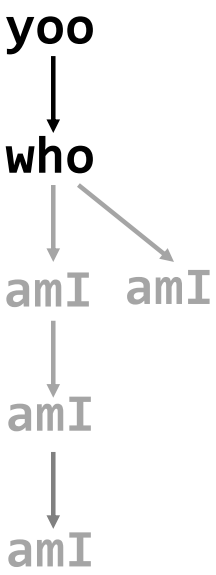


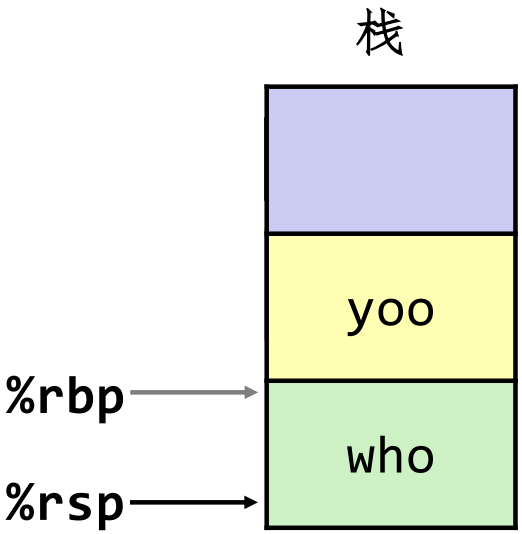




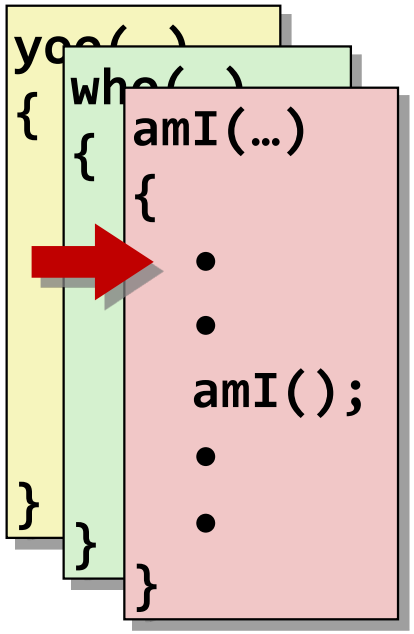
示例

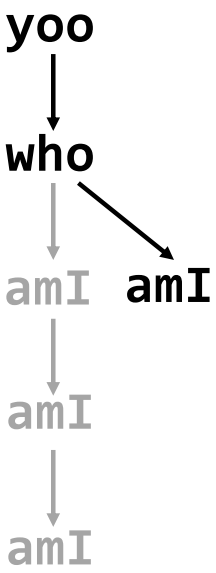


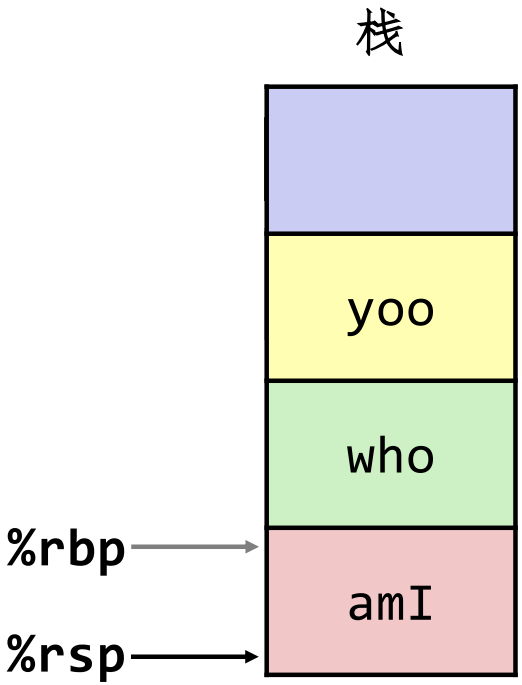




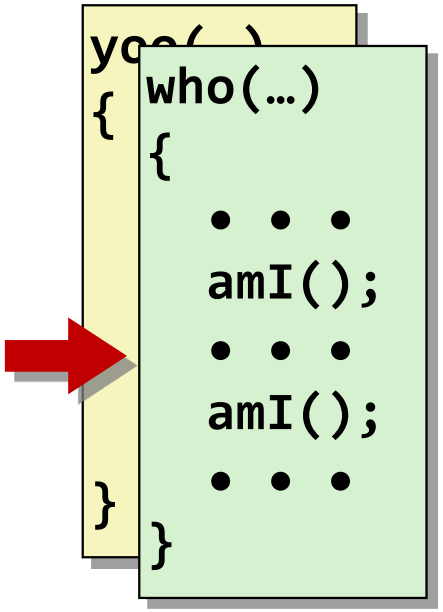
示例

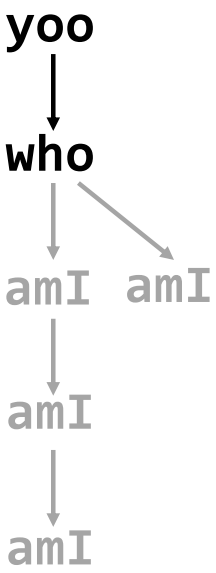


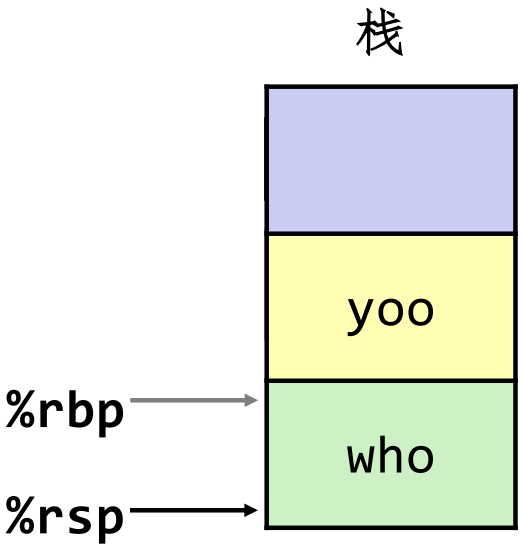




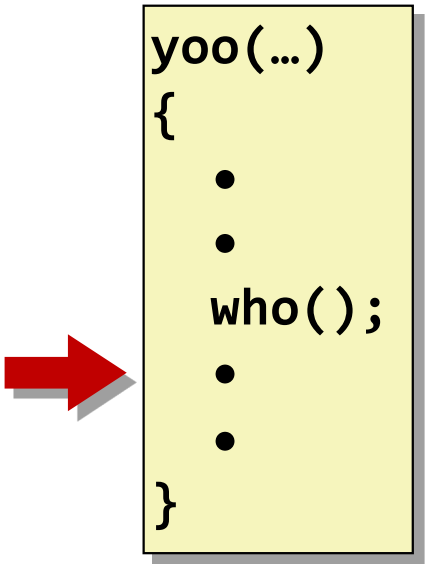
示例

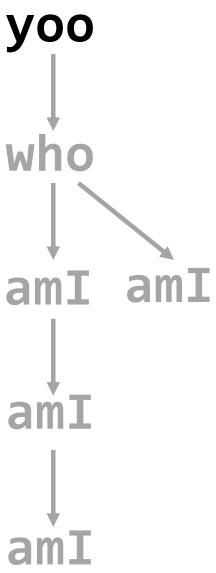


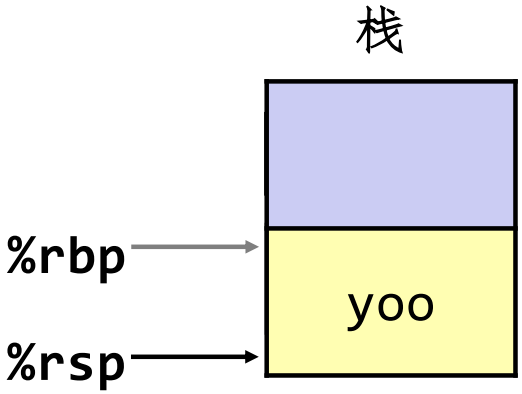




示例

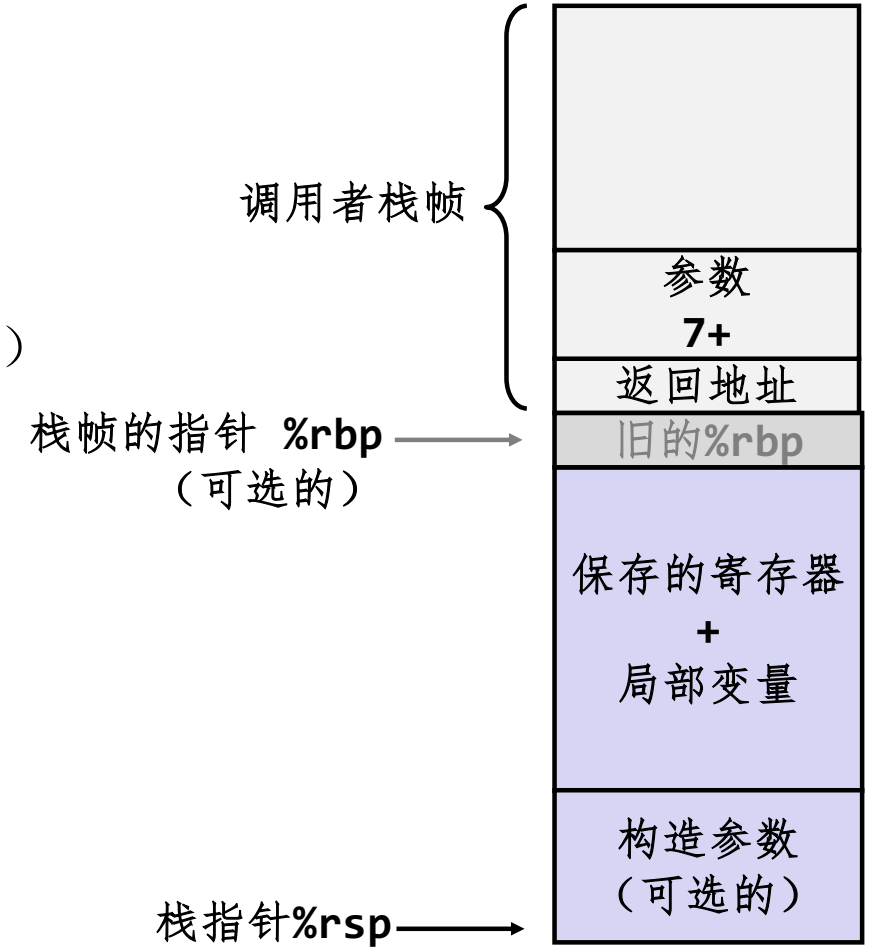






x86-64/Linux 栈帧

- 当前栈帧（从栈顶到栈底）
  - 参数构造区
    为调用的函数准备变量
  - 局部变量（如果在寄存器中不能存储）
  - 保存的寄存器
  - 旧栈帧的指针（可选择）
- 调用者的栈帧
  - 返回地址
    - 被call指令push入栈
  - 为调用函数准备的参数



示例：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
```

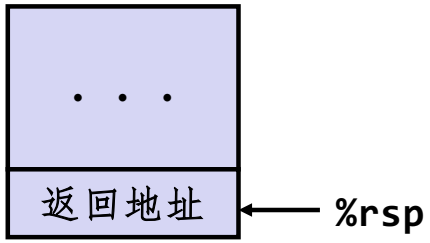
寄存器	值
%rdi	变量 p
%rsi	变量 val, y
%rax	返回值 x

示例：调用incr #1

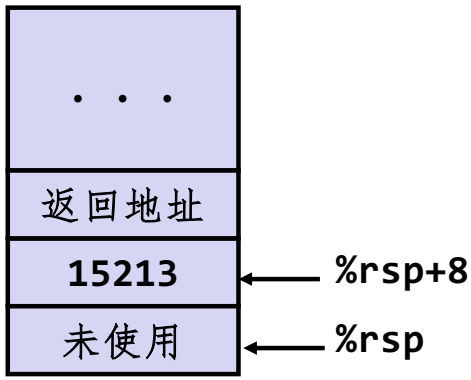
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
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
```

初始的栈结构



现在的栈结构

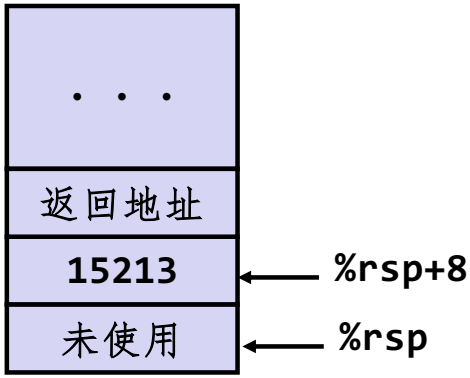


示例：调用incr #2

```
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
```

栈的结构



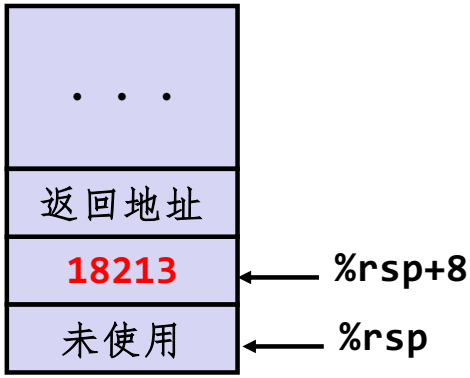
寄存器	值
%rdi	&v1
%rsi	3000

示例：调用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
```

栈的结构



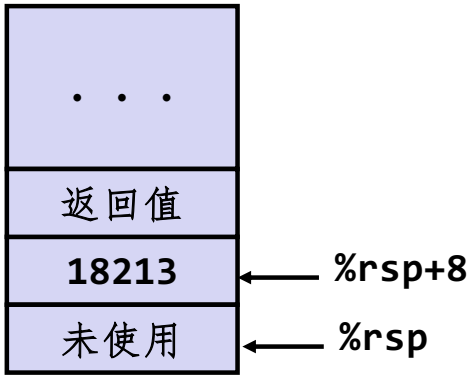
寄存器	值
%rdi	&v1
%rsi	3000

示例：调用incr #4

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

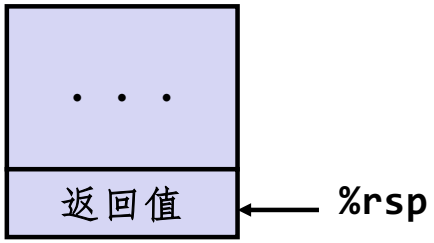
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

栈的结构



寄存器	值
%rax	返回值

更新后的栈结构

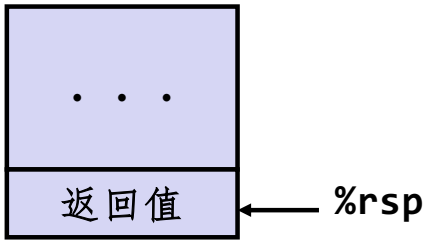


示例：调用incr #5

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

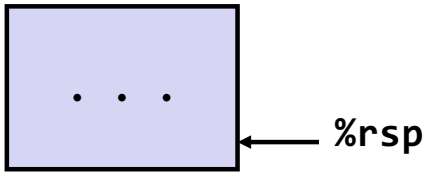
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

更新后的栈结构



寄存器	值
%rax	返回值

最后的栈结构



寄存器使用惯例

- 当程序 **yoo** 调用 **who** 时：
  - yoo 是调用者
  - who 是被调用者
- 寄存器可以用于临时存储吗？

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

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

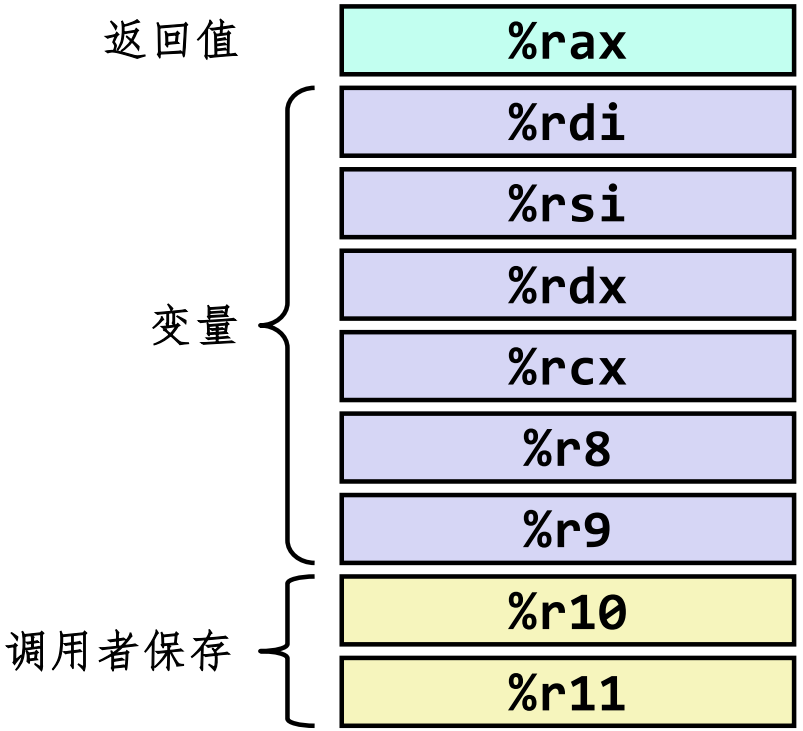
  - 寄存器 %rdx 的内容会被 who 重写
  - 这是相当麻烦的 → 我们需要采取措施！
    - 需要协调调用者和被调用者

寄存器使用惯例

- 当程序 **yoo** 调用 **who** 时：
  - yoo 是调用者
  - who 是被调用者
- 寄存器可以用于临时存储吗？
- 惯例
  - “调用者保存”
    - 函数调用前，调用者要在自己的栈帧里保存临时值
  - “被调用者保存”
    - 被调用者在使用寄存器值之前，在自己的栈帧中存储这些临时值
    - 被调用者在返回调用者之前，恢复这些值

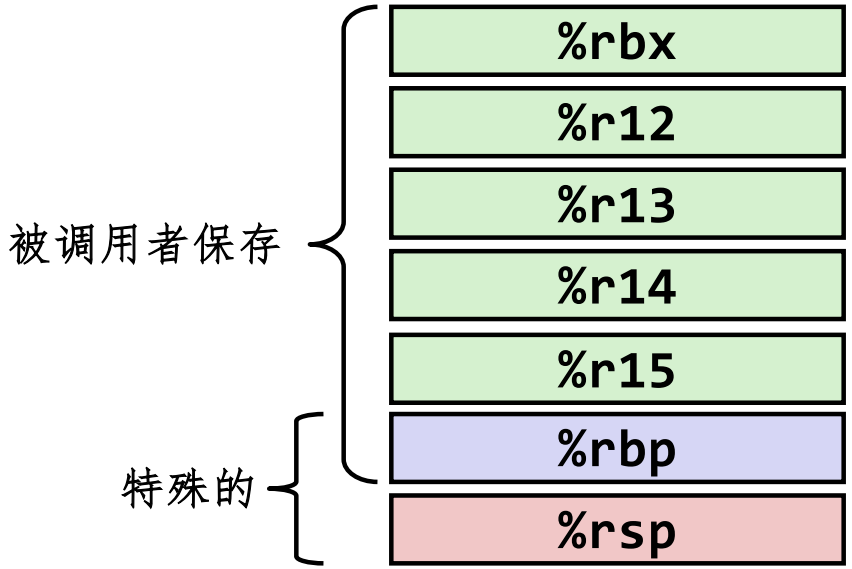
x86-64 Linux 寄存器的使用 #1

- %rax
  - 返回值
  - 调用者保存
  - 程序可以修改
- %rdi, ..., %r9
  - 变量
  - 调用者保存
  - 程序可以修改
- %r10, %r11
  - 调用者保存
  - 程序可以修改



x86-64 Linux 寄存器的保存 #2

- %rbx, %r12, %r13, %r14,%r15
  - 被调用者保存
  - 被调用者必须保存并恢复值
- %rbp
  - 被调用者保存
  - 被调用者必须保存并恢复值
  - 可以被用作帧指针
- %rsp
  - 一种特殊的被调用者保存寄存器
  - 在退出过程时恢复成原来的值

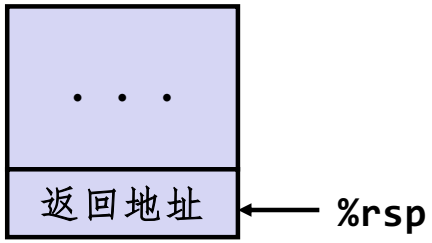


被调用者保存示例 #1

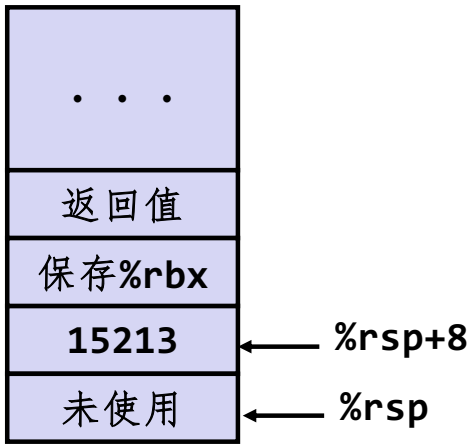
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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
```

初始的栈结构



当前的栈结构

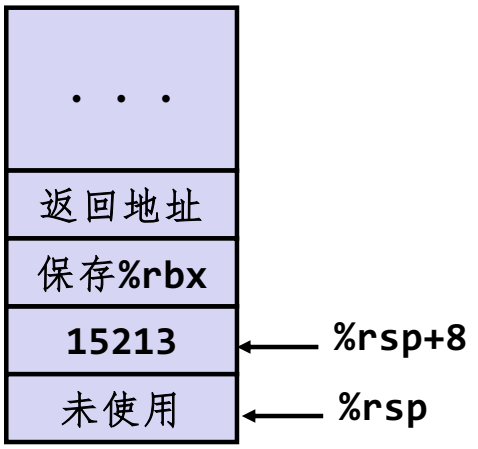


被调用者保存示例 #2

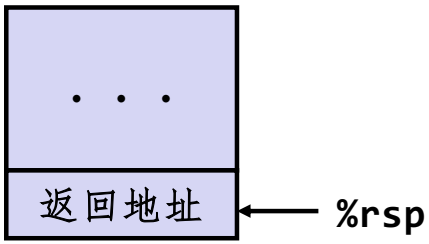
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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
```

当前的栈结构



返回前的栈结构



本节内容

- 过程
  - 栈的结构
  - 调用的惯例
    - 控制转移
    - 数据传送
    - 管理局部数据
  - 递归

递归函数

```
/* 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
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数终止情况

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

寄存器	值	类型
%rdi	x	参数
%rax	返回值	返回值

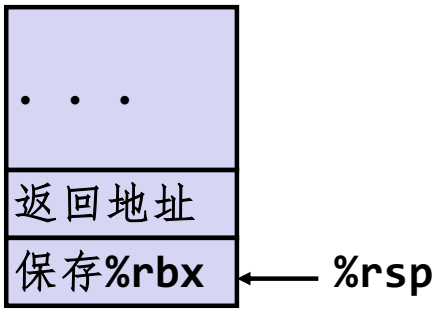
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数保存寄存器

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

寄存器	值	类型
%rdi	x	参数

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



递归函数调用准备

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

寄存器	值	类型
%rdi	x >> 1	递归参数
%rbx	x & 1	被调用者保存

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

递归函数调用

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

寄存器	值	类型
%rbx	x & 1	被调用者保存
%rax	递归调用返回值	

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

递归函数结果

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

寄存器	值	类型
%rbx	x & 1	被调用者保存
%rax	返回值	

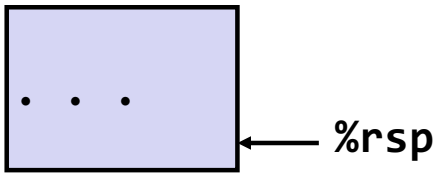
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数完成

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

寄存器	值	类型
%rax	返回值	返回值

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



有关递归的观察

- 对递归不需要特殊处理
  - 栈帧意味着每一次函数调用都有私有的空间
    - 保存的寄存器和局部变量
    - 保存的返回地址
  - 寄存器保存惯例阻止一次函数调用篡改另一次调用的数据
    - 除非 C 代码显式地这样做（例如，后面会谈到的缓冲区溢出）
  - 栈规则与过程调用/返回模式保持一致
    - 如果 P 调用 Q，那么 Q 在 P 之前返回
    - 后进先出
- 相互递归适用
  - P调用Q；Q调用P

x86-64 过程总结

- 重点
  - 栈是保存过程调用、返回的不二之选
    - 如果 P 调用 Q，则 Q 在 P 之前返回
- 一般的调用惯例就解决了递归（相互递归）
  - 可以安全地把数据存储在局部栈帧和被调用者保存寄存器中
  - 参数置于栈顶
  - 返回值放在%rax
- 指针是值的地址
  - 值存储在栈上或为全局变量