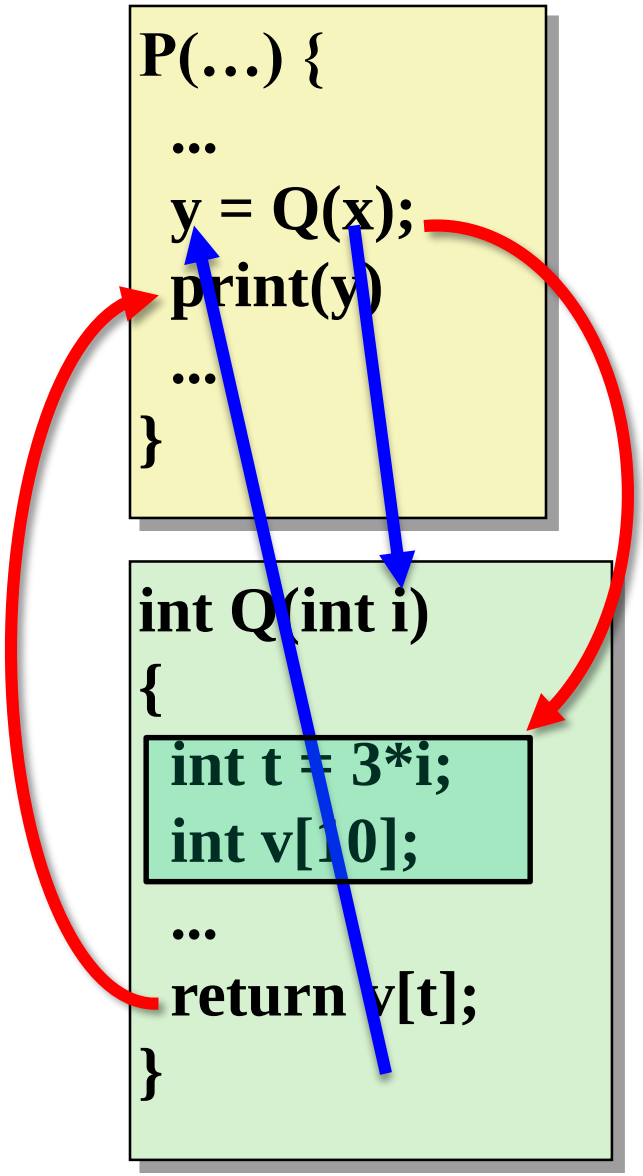
程序的机器级表示Ⅲ：过程

教师：郑贵滨
计算机科学与技术学院
哈尔滨工业大学

过程的机制

- 传递控制
  - 调用：转到过程代码的起始处
  - 结束：回到返回点
- 传递数据
  - 过程参数
  - 返回值
- 内存管理
  - 过程运行期间申请
  - 返回时解除分配
- 该机制全部由机器指令实现
- x86-64 过程的实现只是使用了这些机制

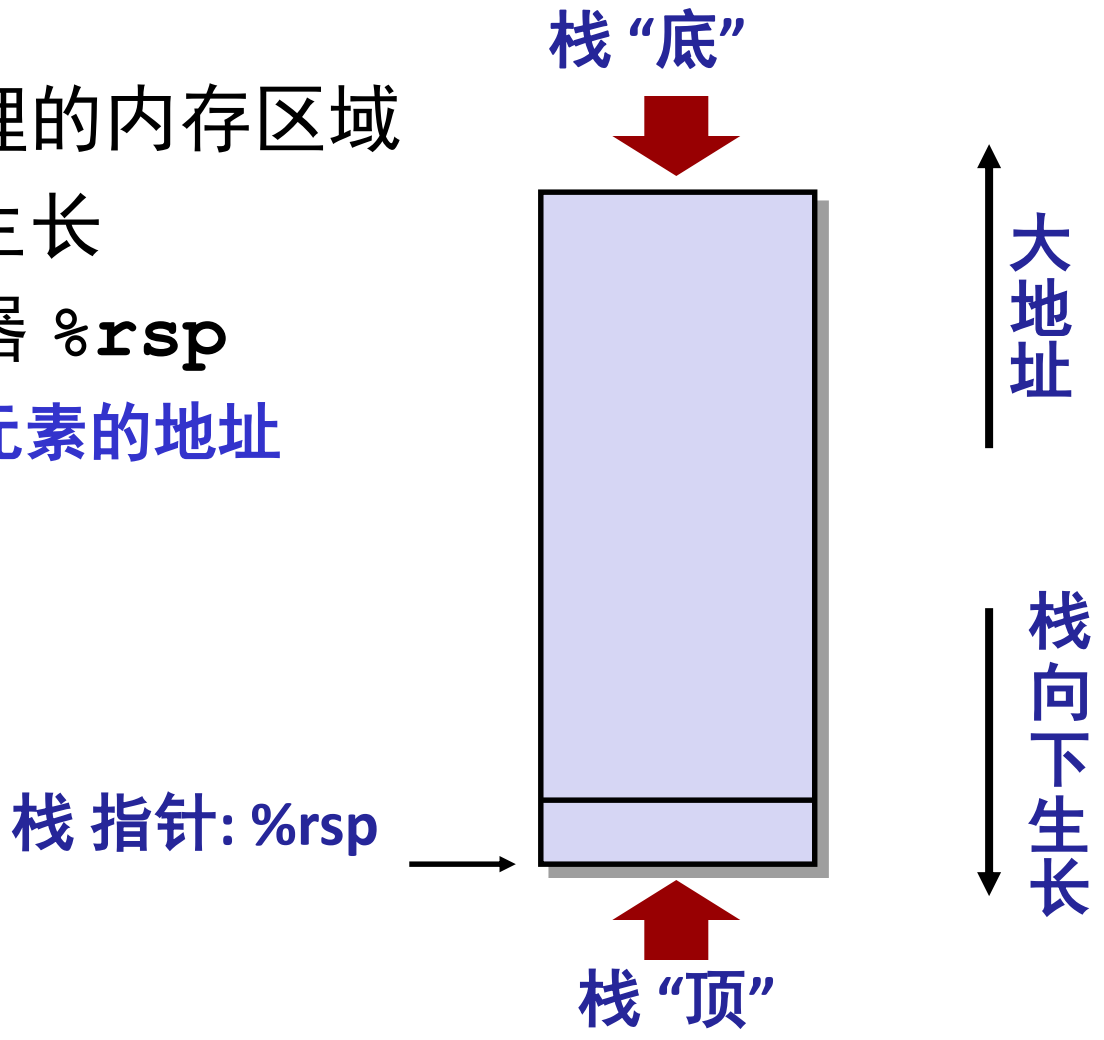


主要内容

- 过程
  - **栈结构**
  - 调用约定
    - 传递控制
    - 传递数据
    - 管理局部数据
  - 递归

x86-64 栈

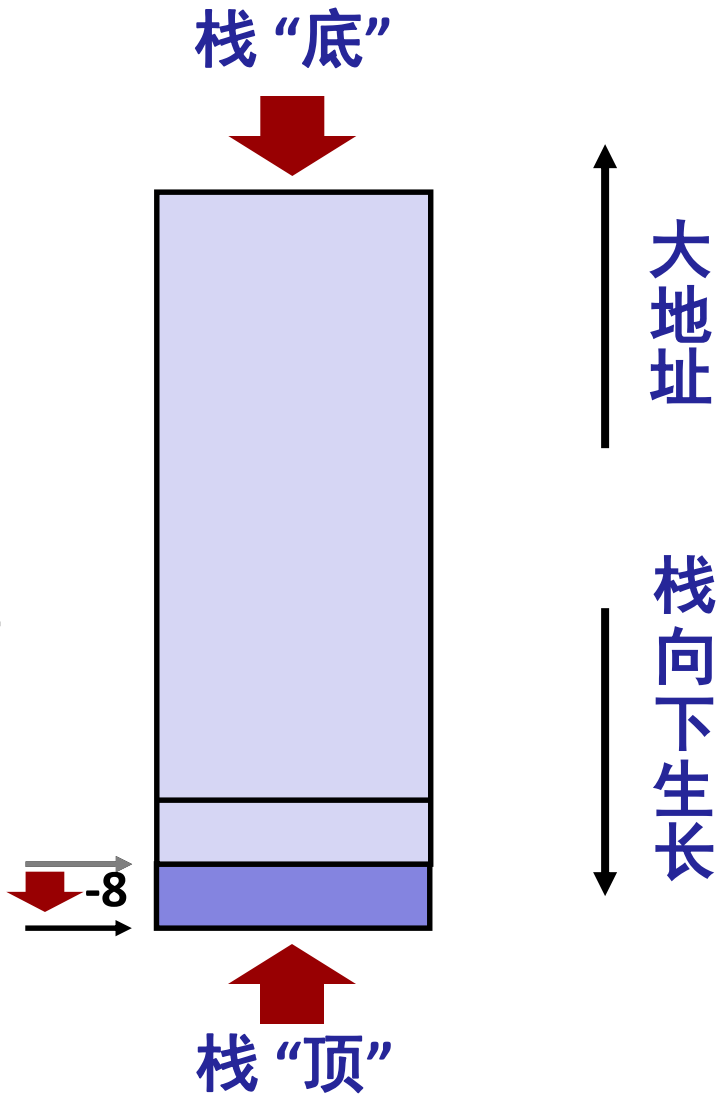
- 使用栈规则管理的内存区域
- 向低地址方向生长
- 栈指针： 寄存器 `%rsp`
  - 保存 栈 “顶”元素的地址



x86-64 入栈指令: push

- 入栈指令 pushq
  - 格式:

  pushq Src
  - 从Src取操作数
  - 将%rsp减8
  - 将操作数写到%rsp指向的地址

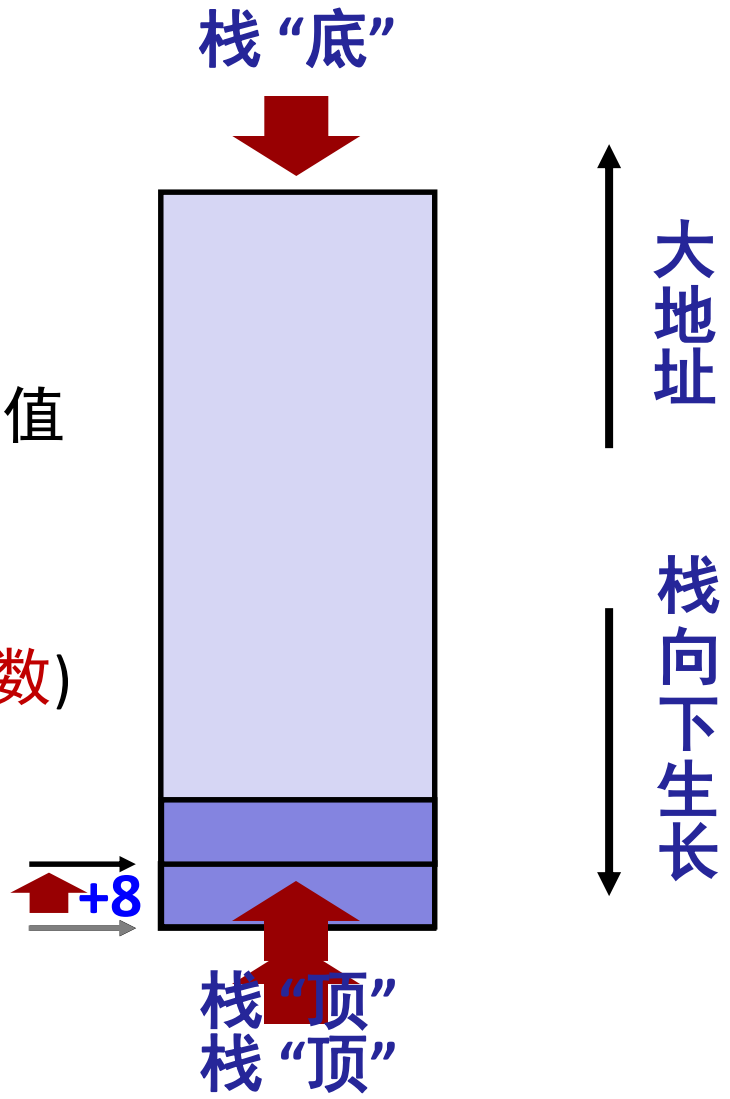


x86-64 出栈指令: pop

- 出栈指令 popq
  - 格式:

  popq Dst

  - 从%rsp中保存的地址值读取数值
  - 将 %rsp加 8
  - 将数值保存到Dst
    (dst必须是寄存器或内存操作数)



主要内容

- 过程
  - 栈结构
  - 调用约定
    - 传递控制
    - 传递数据
    - 管理局部数据
  - 递归与指针的解释

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

- 栈：支持过程的调用、返回
- 过程调用

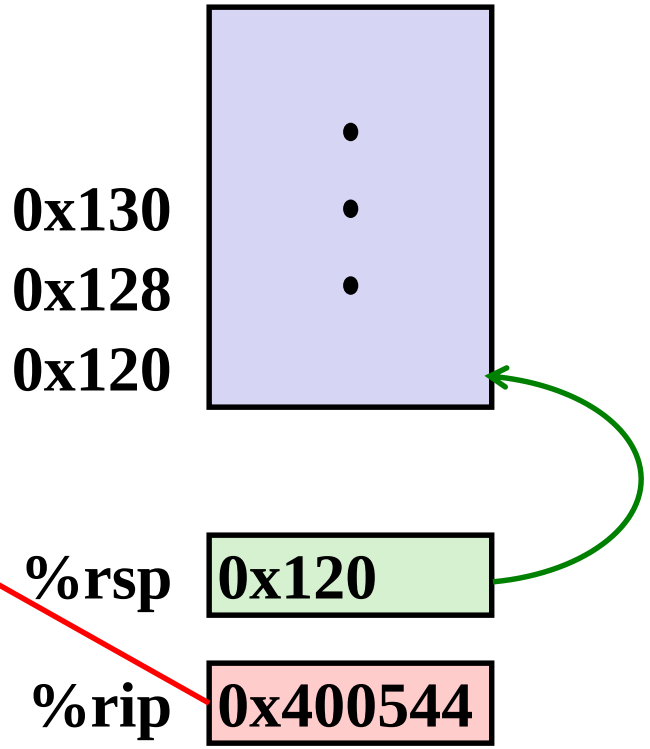
  `call` *func_label*
  - 返回地址入栈(Push)
  - 跳转到*func_label* (函数名字就是函数代码段的起始地址)
- 返回地址:
  - 紧随call指令的下一条指令的地址 (考虑PC——RIP的含义)
- 过程返回

  `ret`
  - 从栈中弹出返回地址(pop)
  - 跳转到返回地址

控制流—1/4

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

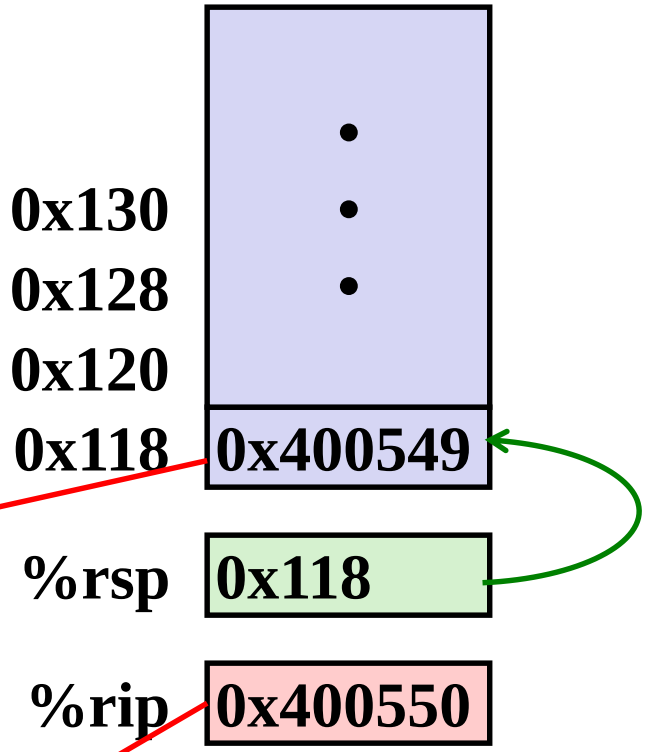
```
0000000000400550 <mult2>:
 400550:  mov    %rdi,%rax
                    •
                    •
 400557:  retq
```



控制流 —2/4

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

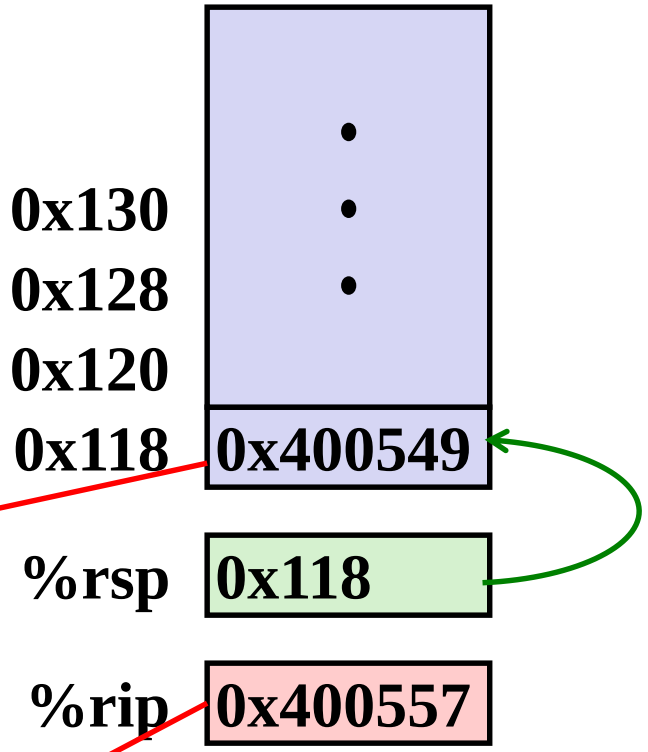
```
0000000000400550 <mult2>:
 400550:  mov    %rdi,%rax
                    •
                    •
 400557:  retq
```



控制流 —3/4

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

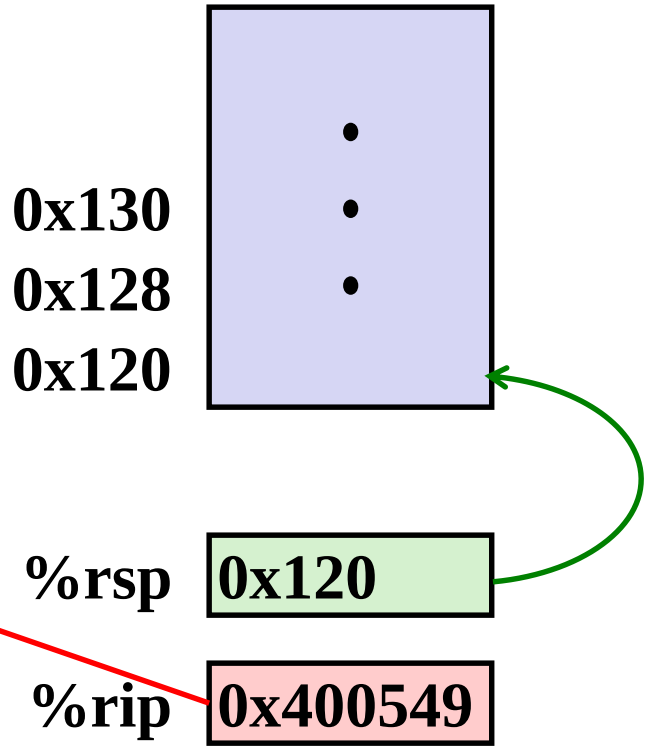
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
                    •
                    •
  400557:  retq
```



控制流 —4/4

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

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

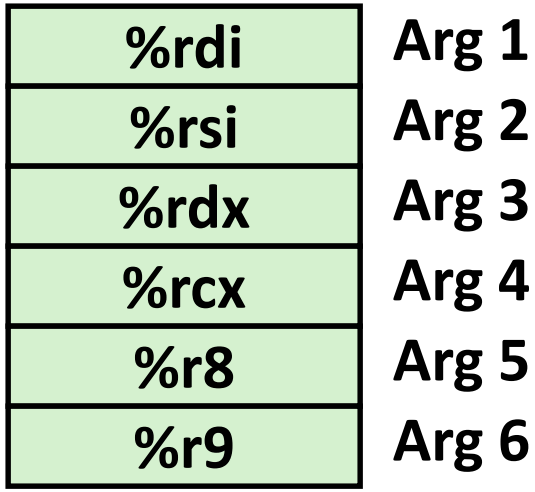


主要内容

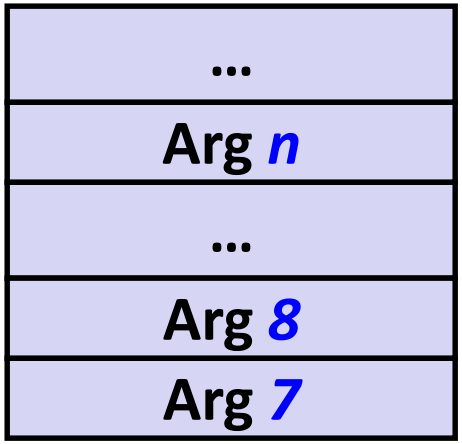
- 过程
  - 栈结构
  - 调用约定
    - 传递控制
    - **传递数据**
    - 管理局部数据
  - 递归与指针的解释

过程数据流

- 参数传递
  - 前6个参数用寄存器



  - 其余参数用栈（注意顺序）



- 返回值

- 局部变量：仅在需要时申请栈空间

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
```

主要内容

- 过程
  - 栈结构
  - 调用约定
    - 传递控制
    - 传递数据
    - **管理局部数据**
  - 递归

基于栈的语言

- 支持递归的语言
  - C、Pascal、Java
  - 代码必须可重入的“*Reentrant*”
    - 过程/函数有多个并发实例(simultaneous instantiations)
  - 需要保存每个实例的状态
    - 参数
    - 局部变量
    - 返回地址
- 栈的使用原则
  - 有限时间内，保存给定程序的状态：从调用的发生到返回
  - 被调用者先于调用者返回
- 栈分配单位——帧，栈中单个过程实例的状态数据

调用链示例

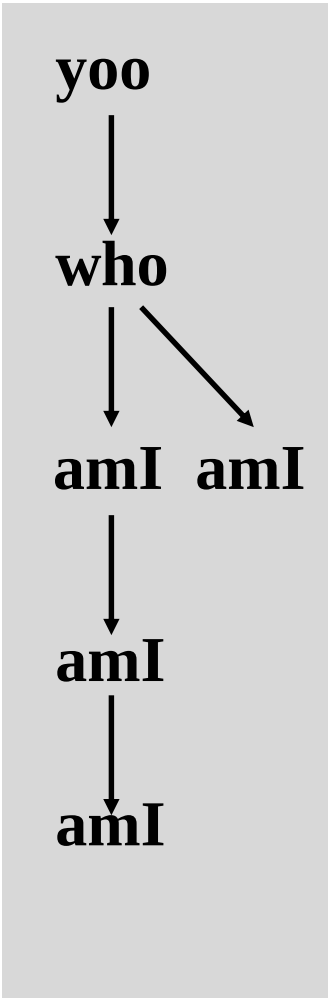
```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

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

过程 amI() 是递归的

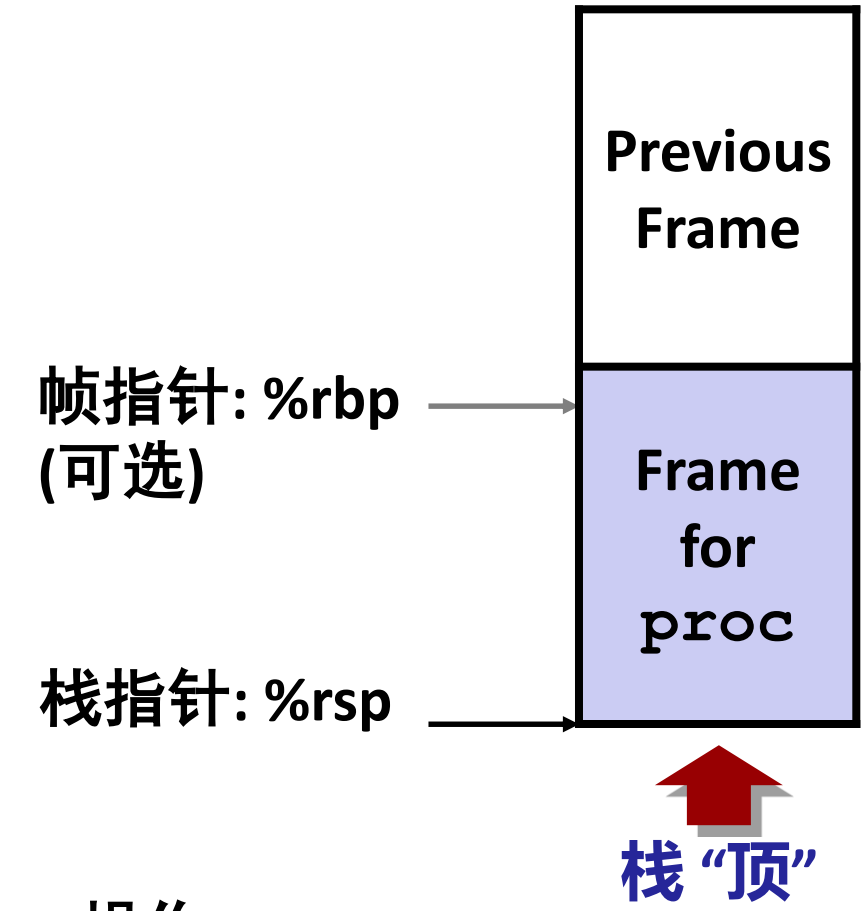
```
amI(…)
{
  …
  amI();
  …
}
```

调用链示例

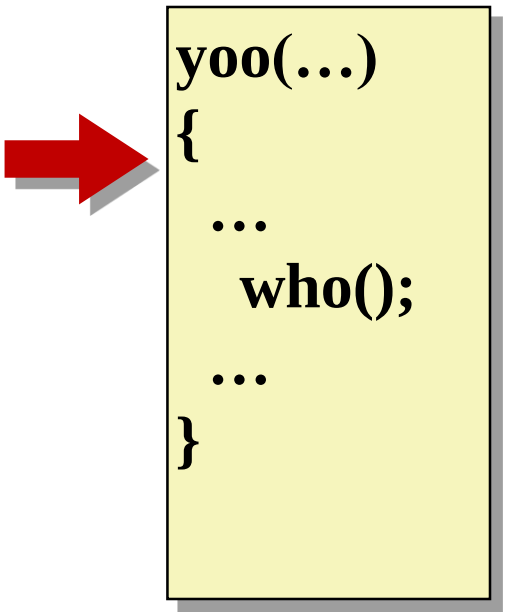


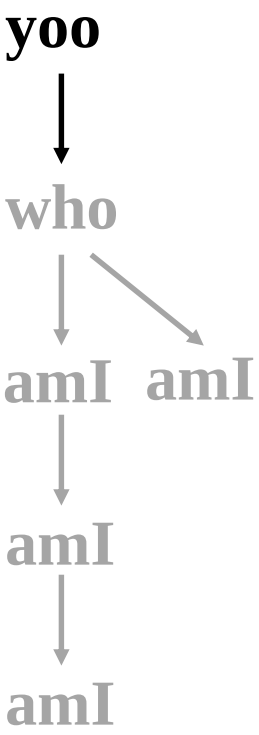
栈帧

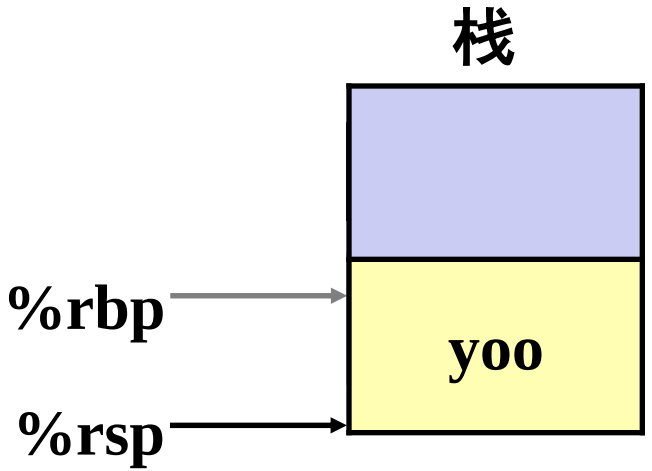
- 内容
  - 返回信息
  - 局部存储(如需要)
  - 临时空间(如需要)
- 管理
  - 进入过程时申请空间
    - 生成代码——构建栈帧
    - 包括**call指令产生的push操作**
  - 当返回时解除申请
    - 结束代码——清理栈帧
    - 包括ret指令产生的pop操作



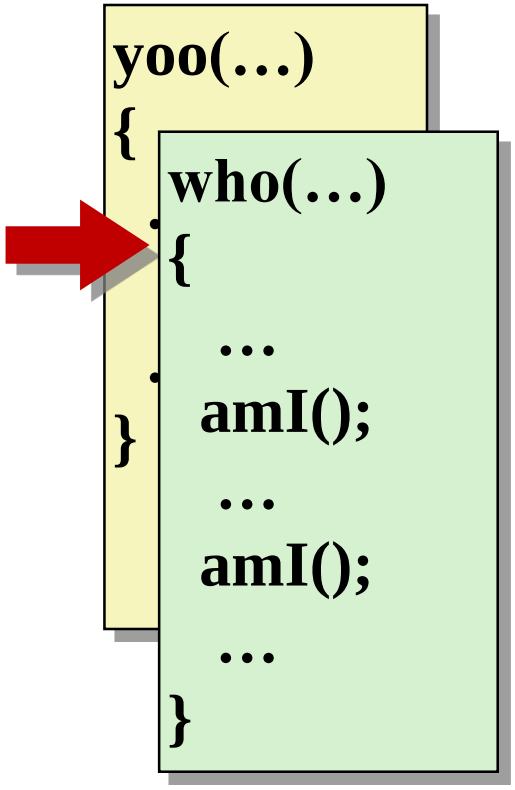
栈帧示例

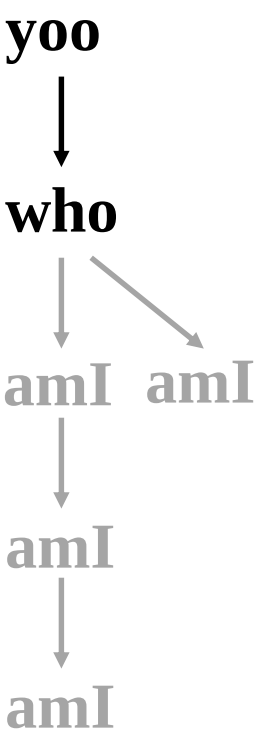


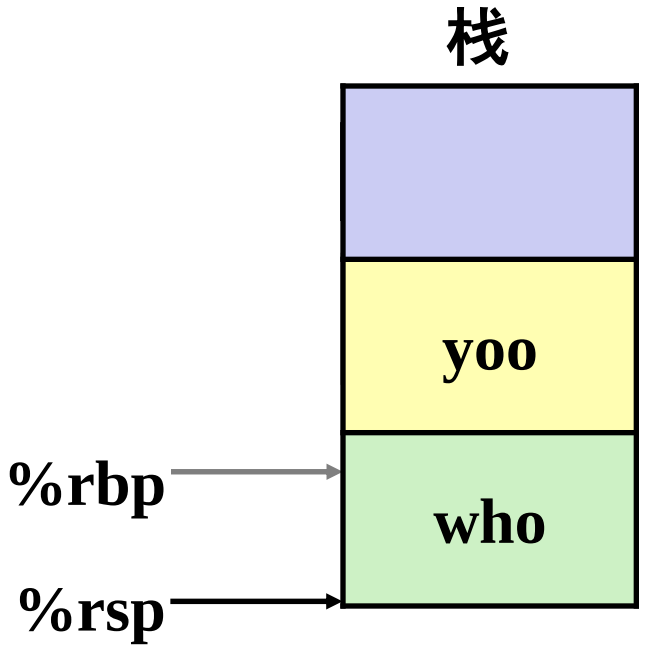




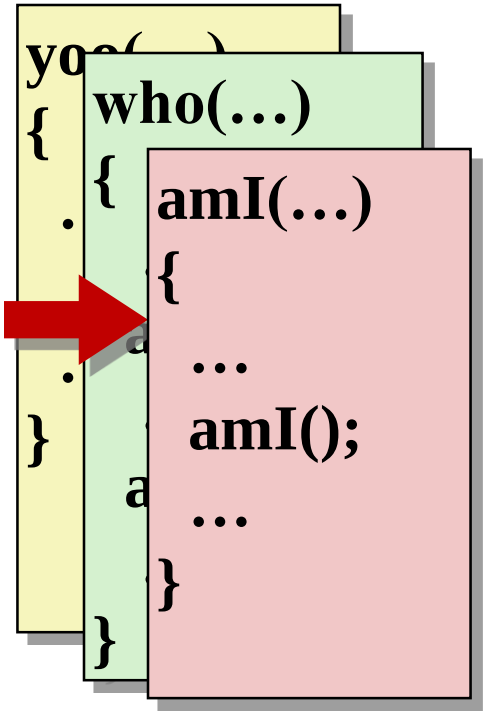
栈帧示例

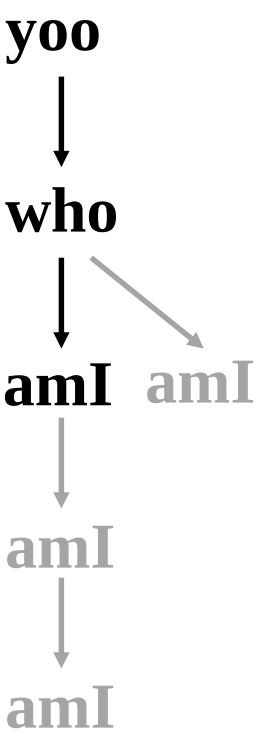


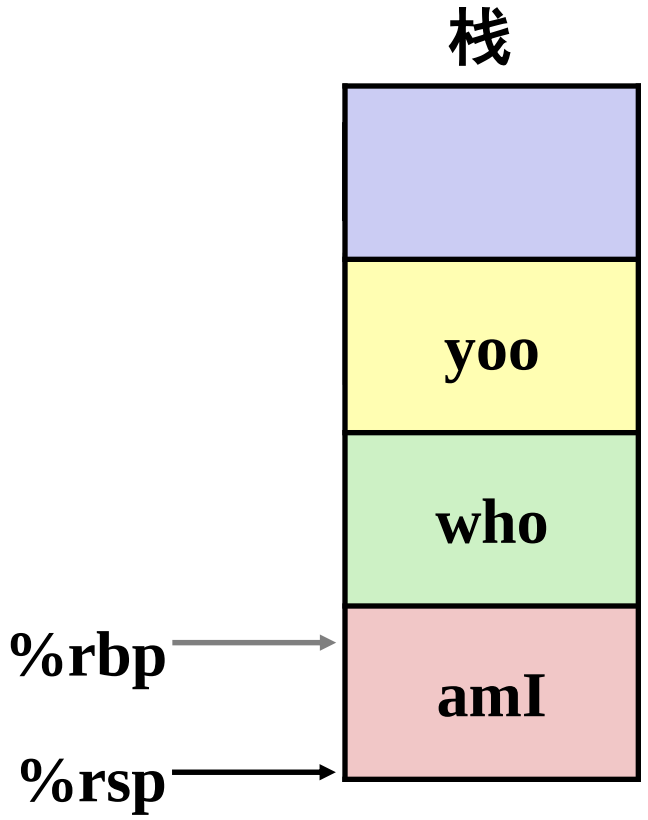




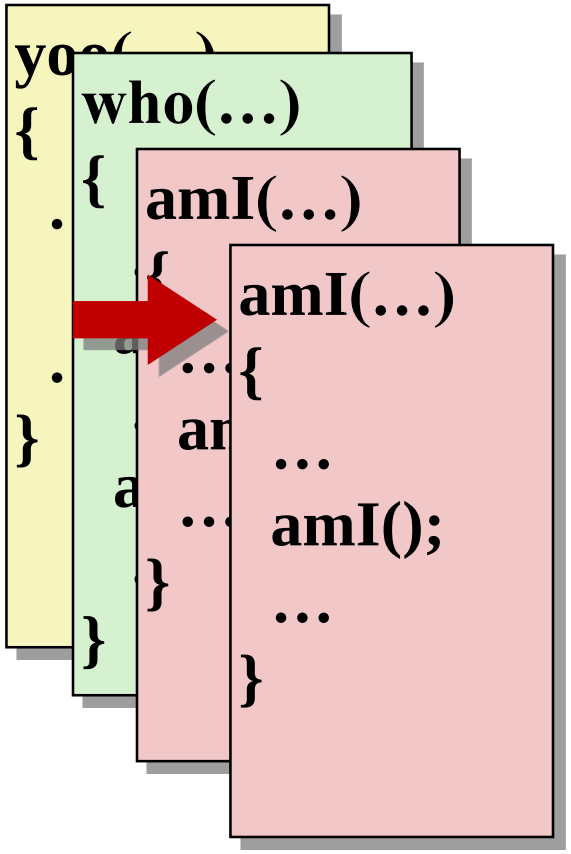
栈帧示例

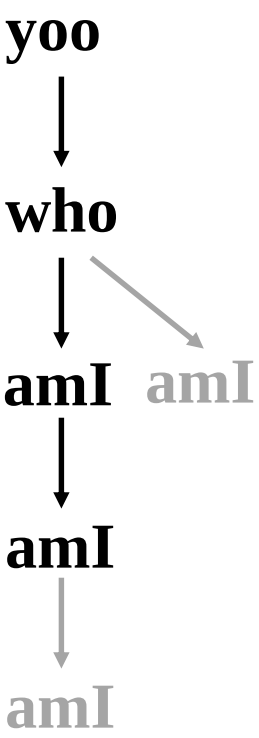


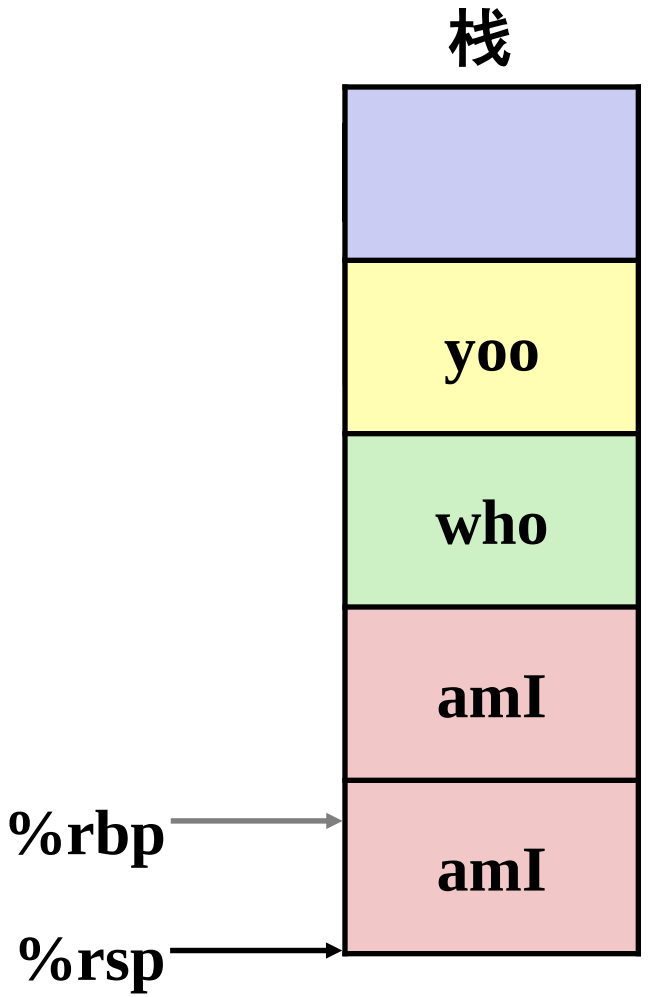




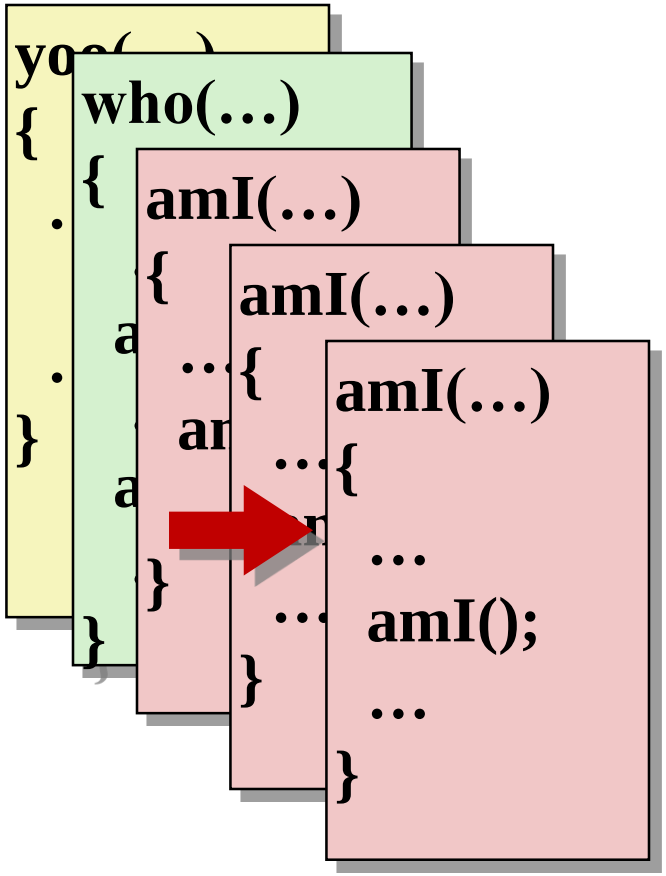
栈帧示例

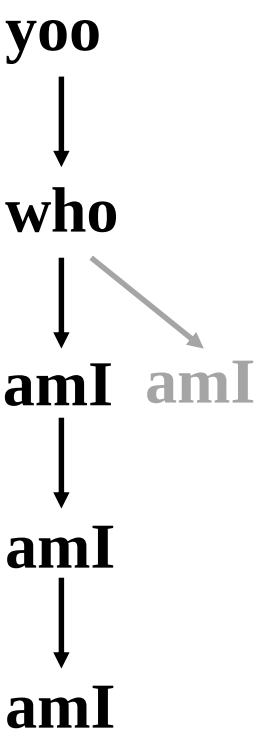


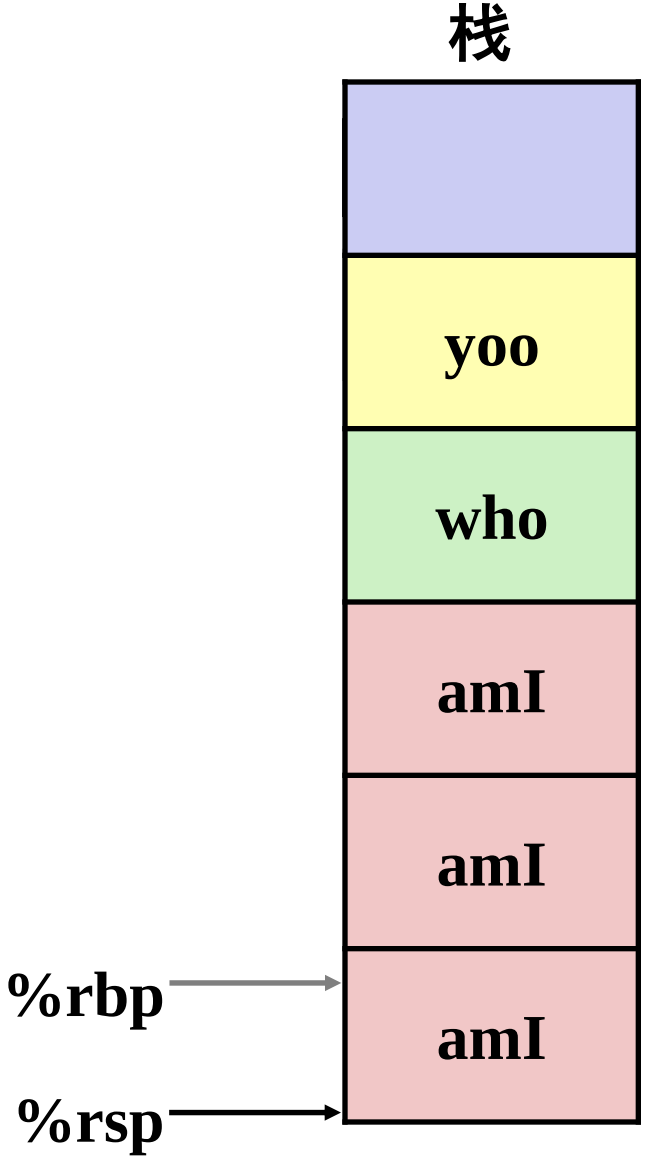




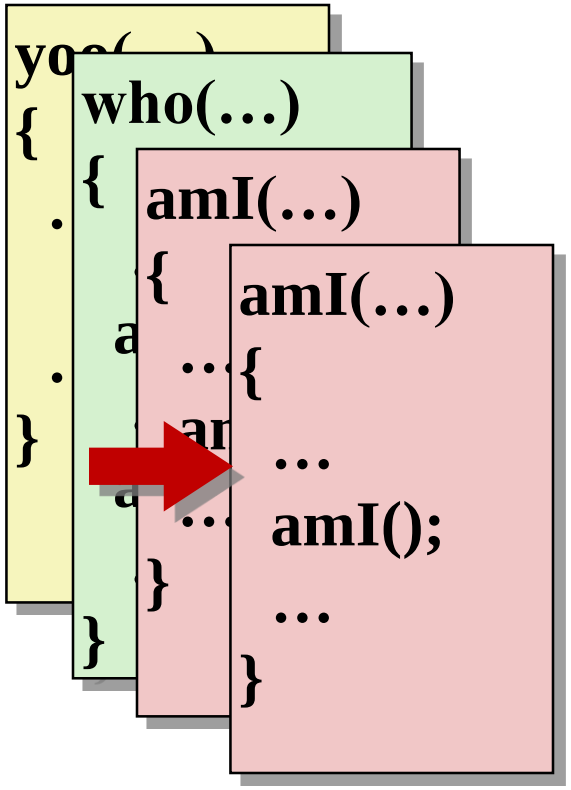
栈帧示例

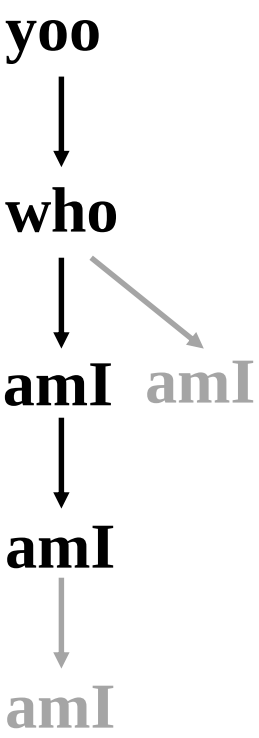

who(…)
{
amI(…)
{
amI(…)
{
amI(…)
{
…
amI();
…
}

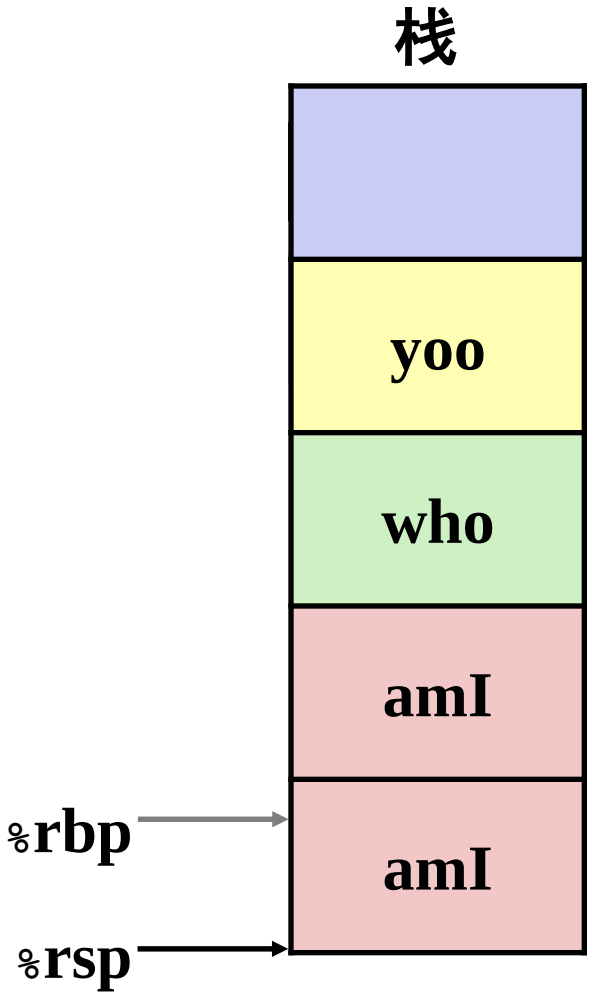

yoo
who
amI
amI
amI
amI


栈
yoo
who
amI
amI
amI
%rbp
%rsp

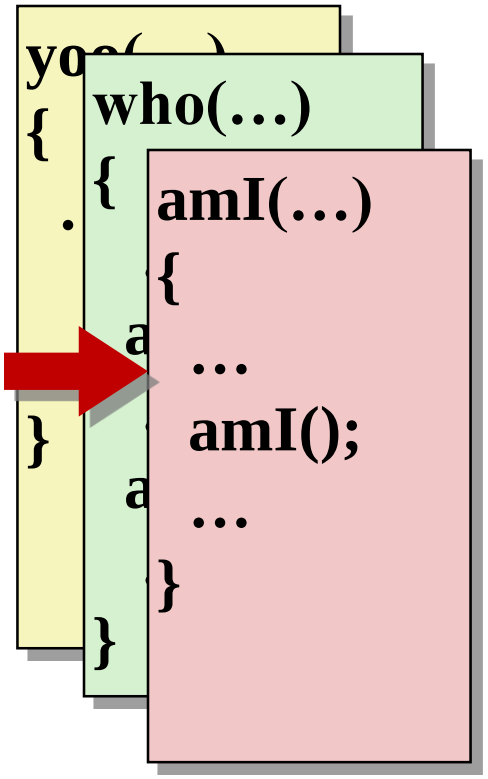
栈帧示例

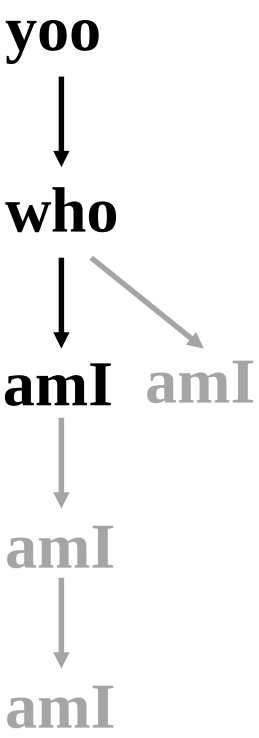


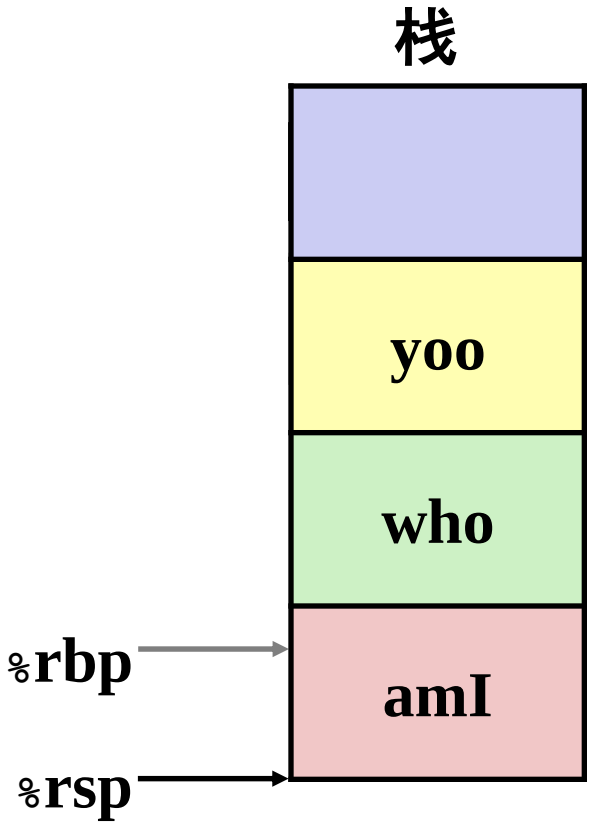




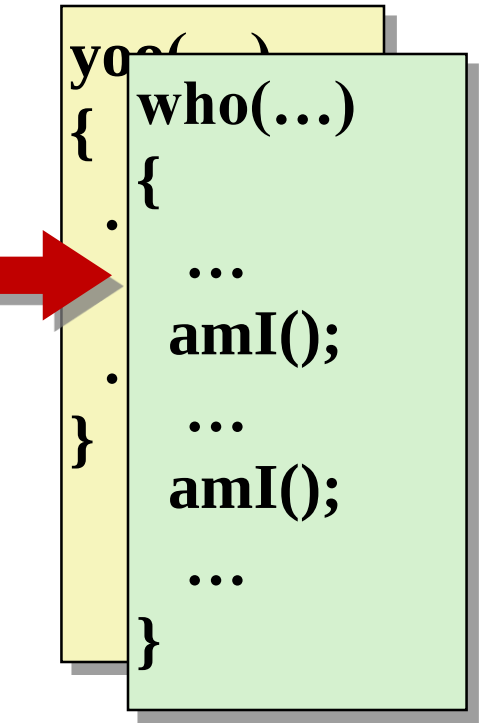
栈帧示例

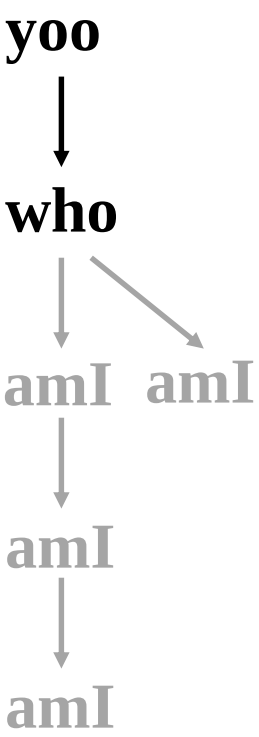


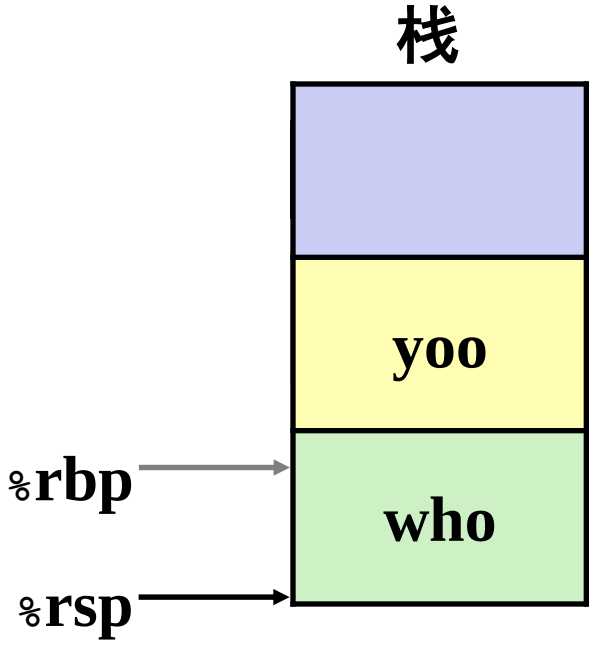




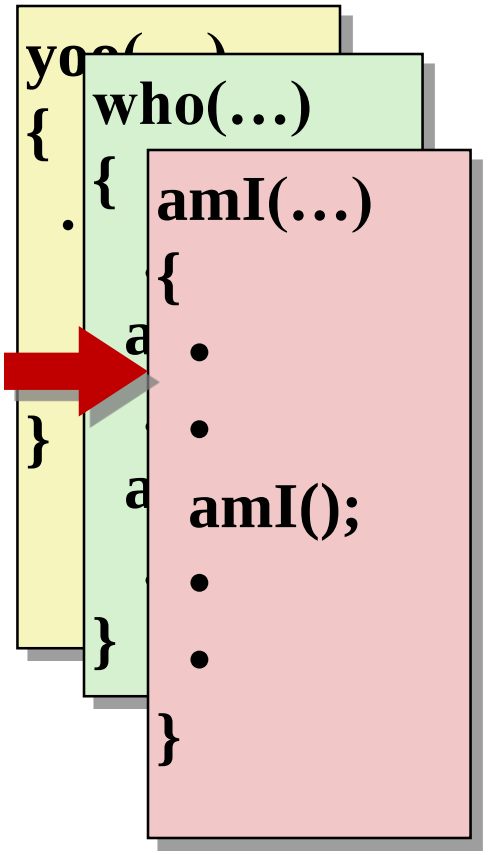
栈帧示例

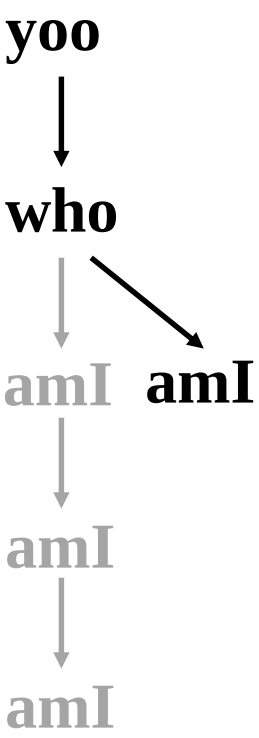


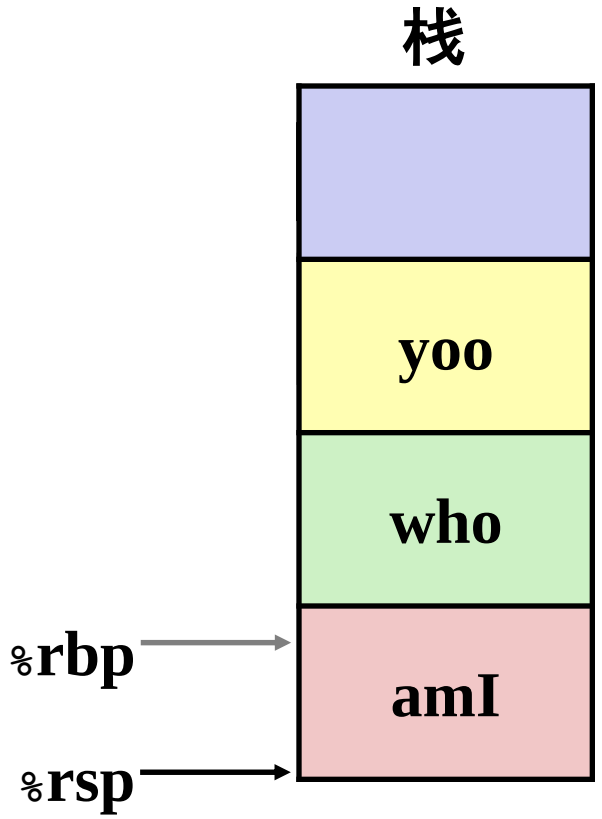




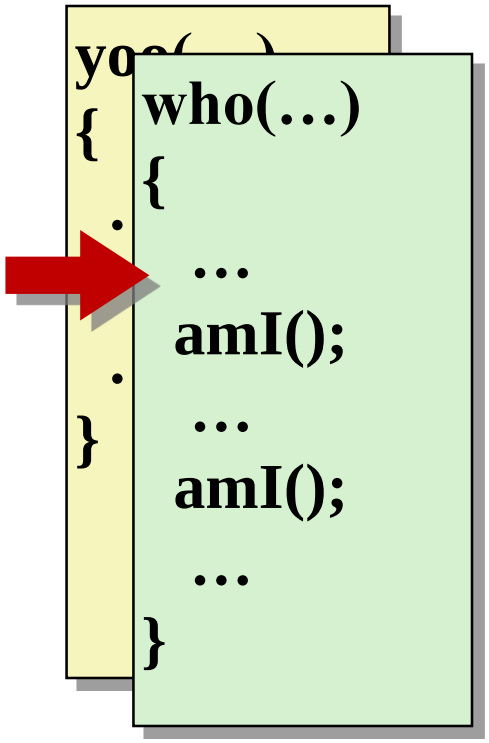
栈帧示例

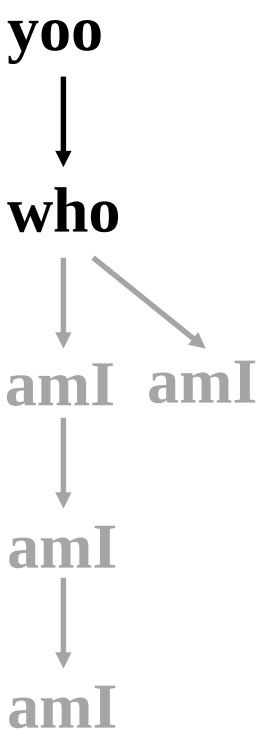


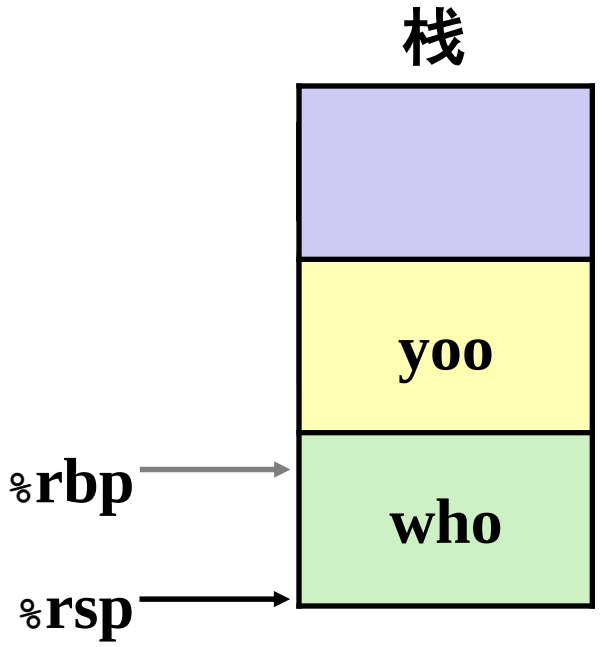




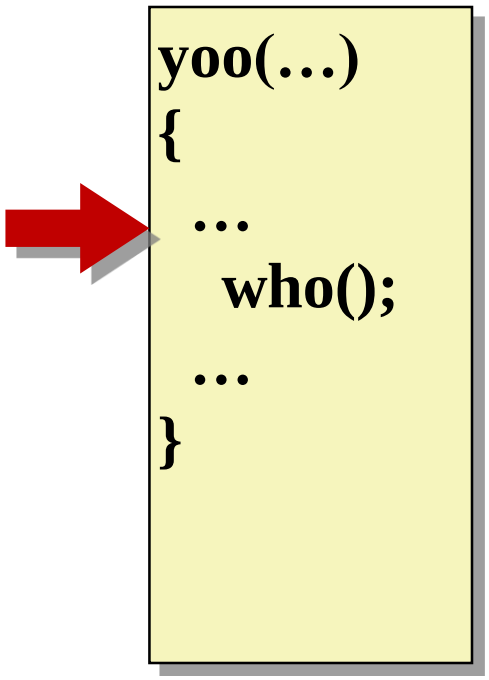
栈帧示例

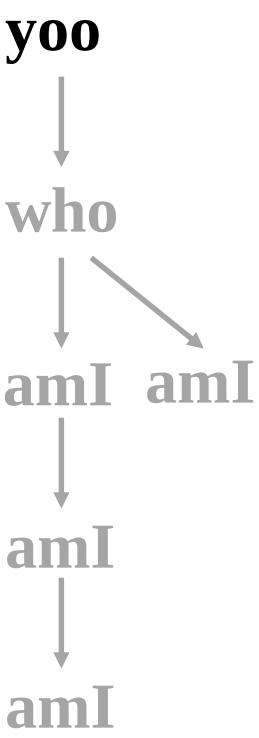


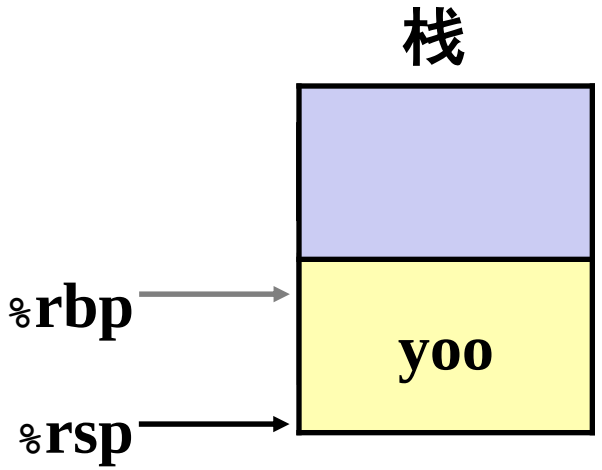




栈帧示例

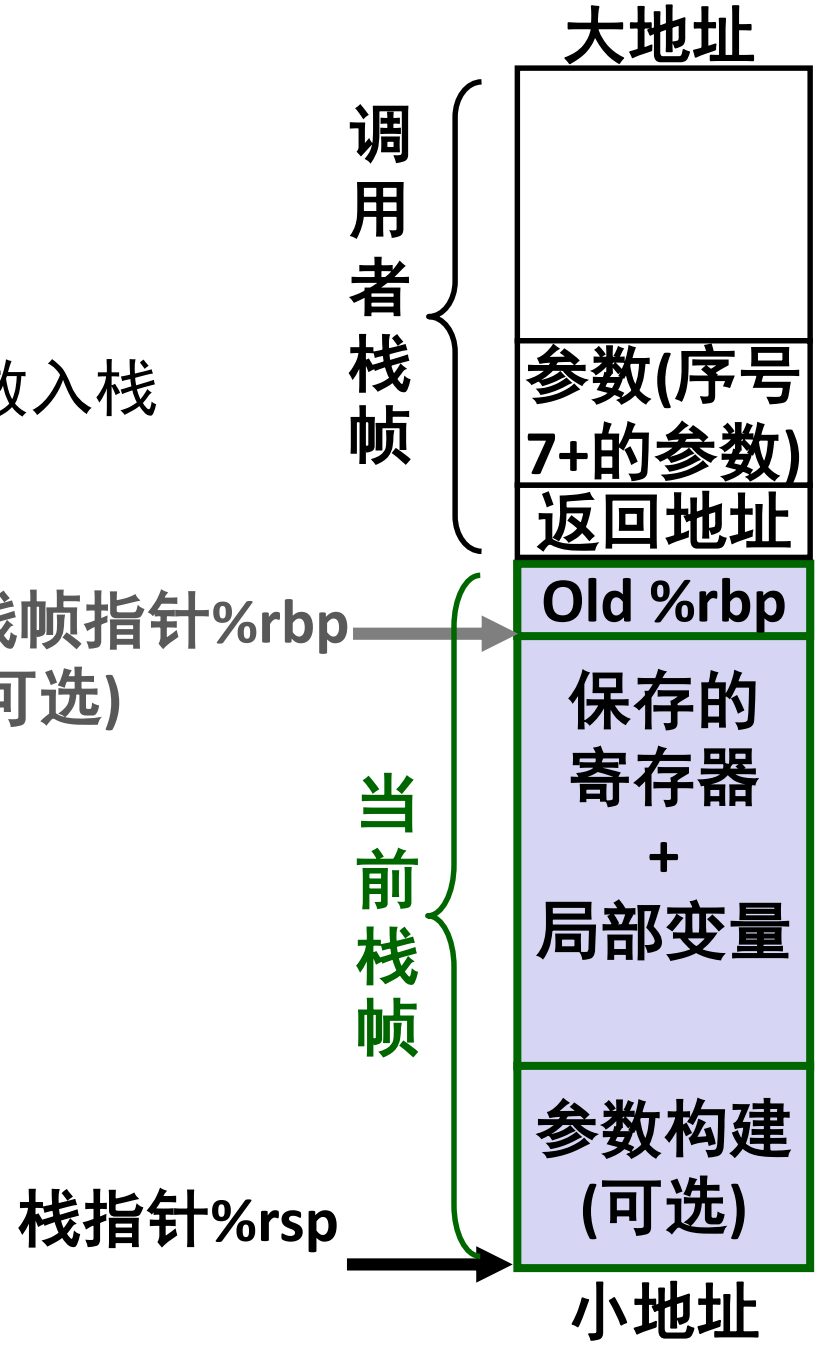






x86-64/Linux 栈帧

- 调用者栈帧
  - “参数建立”：函数调用所需参数入栈
  - 返回地址
    - 由call指令压入栈
- 当前栈帧
  - 旧栈帧指针 (可选)
  - 保存的寄存器内容
  - 局部变量
    如果不能用寄存器实现，则在栈中实现。
  - “参数建立”：把即将调用的函数所需参数入栈。



函数实例: 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
```

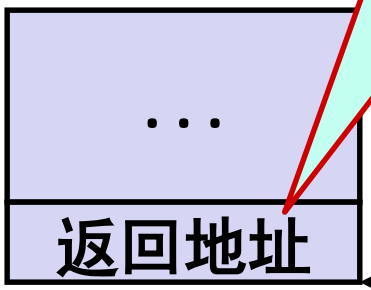
寄存器	用途
%rdi	参数 p
%rsi	参数 val, y
%rax	x, 返回值

用寄存器实现局部变量x
没有单独实现C函数中的局部变量 y

函数调用的栈结构实例：1/5

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

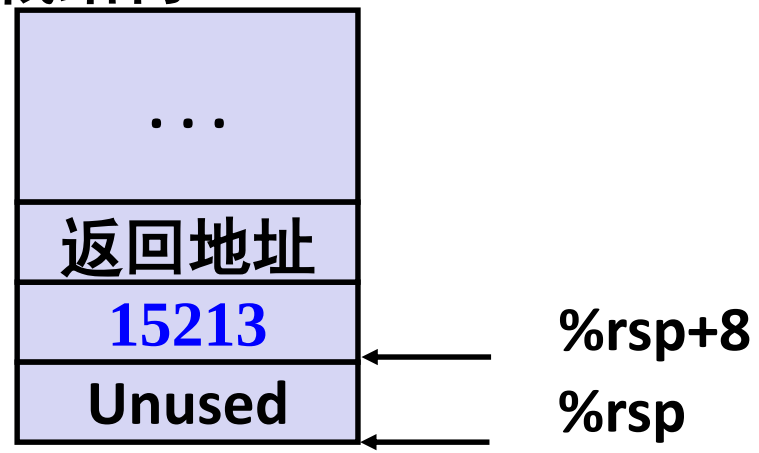
函数call_incr()的起始的栈结构




```
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
```

局部变量v1

栈结构

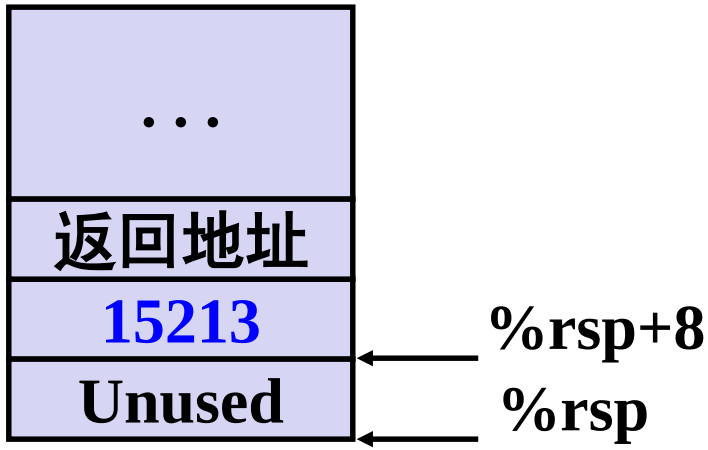


函数调用的栈结构实例：2/5

```
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
```

栈结构



寄存器	用途
%rdi	&v1
%rsi	3000

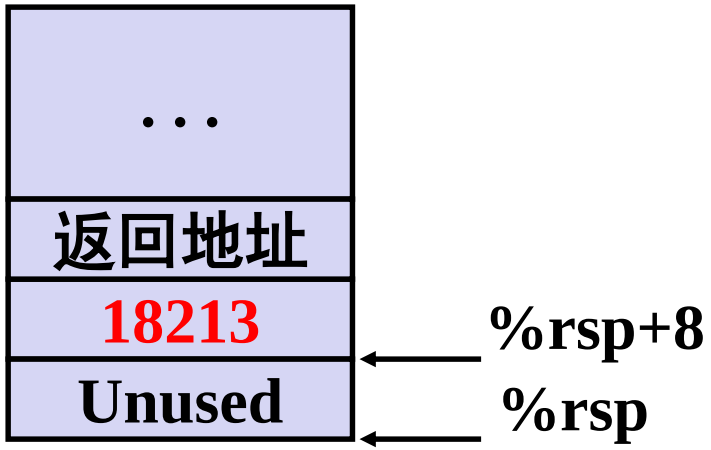
函数调用的栈结构实例：3/5

```
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
```

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

call incr之后的栈结构



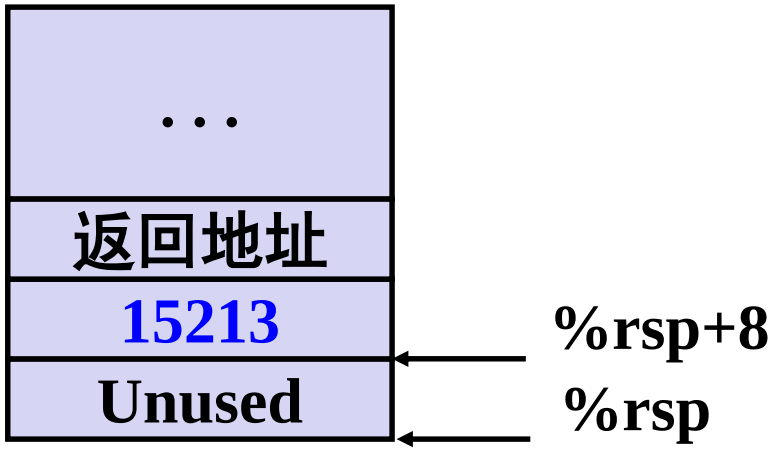
寄存器	用途
%rdi	&v1
%rsi	3000

函数调用的栈结构实例：4/5

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

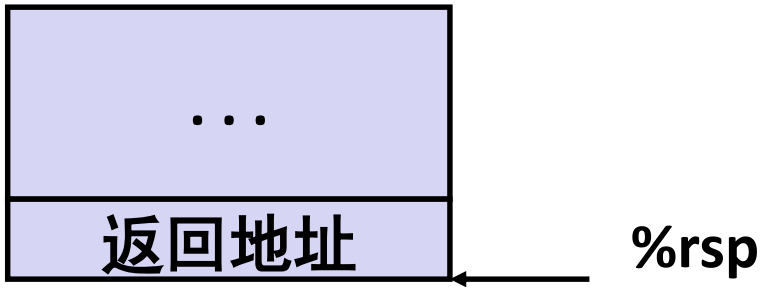
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

栈结构



寄存器	用途
%rax	返回值

更新后的栈结构

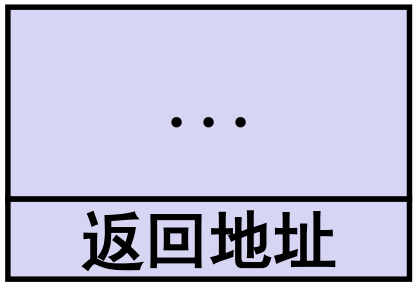


函数调用的栈结构实例：5/5

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

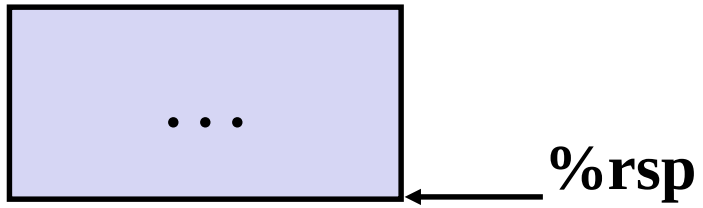
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

更新后的栈结构



寄存器	用途
%rax	返回值

ret运行之后的栈结构



寄存器保存约定

- 当过程 **yoo**调用**who**时:
  - **yoo**是调用者(caller)
  - **who**是被调用者(callee)
- 寄存器能否用于临时存储?

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

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

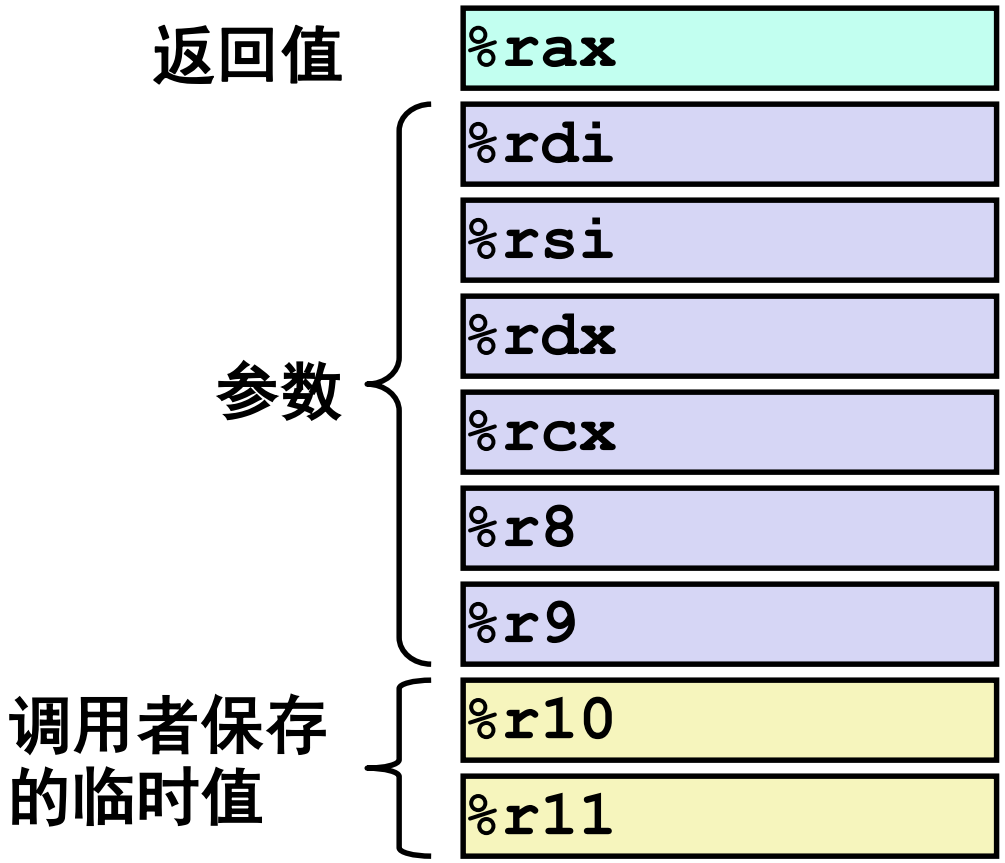
  - 寄存器**%rdx**的内容被 **who**覆盖写了
  - 这样会有问题，如果解决?
    - 需要调用者(caller)和被调用者(callee)之间的协调

寄存器保存约定

- 当过程 **yoo**调用**who**时:
  - **yoo**是调用者(caller)
  - **who**是被调用者(callee)
- 寄存器能否用于临时存储？
- 约定——谁来保存的问题
  - 调用者保存***"Caller Saved"***
    - 调用者在调用前，在它的栈帧中保存临时值(寄存器)
  - 被调用者保存***"Callee Saved"***
    - 被调用者要先在自己的栈帧中保存，然后再使用(寄存器)
    - 返回到调用者之前，恢复这些保存的值

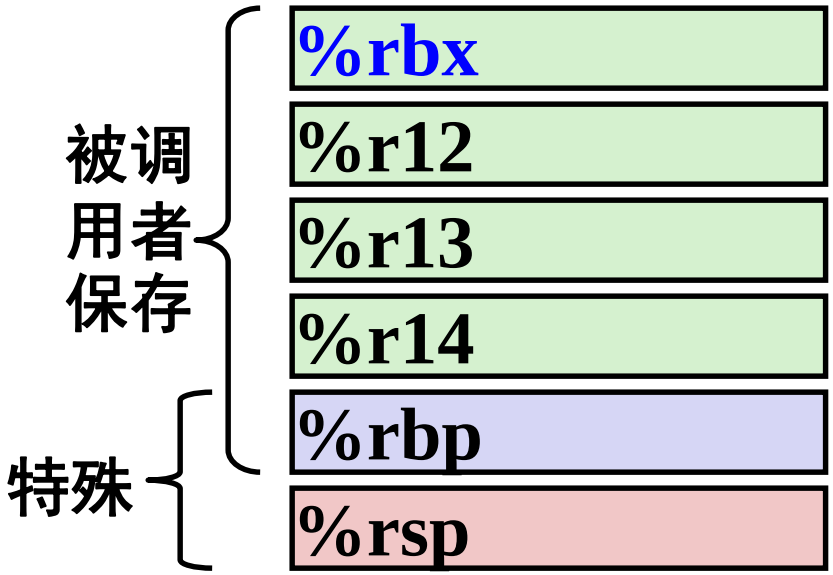
x86-64 Linux的寄存器用法#1

- %rax
  - 返回值
  - 调用者保存
  - 被调用过程可修改
- %rdi, …, %r9
  - 传递函数参数
  - 调用者保存
  - 被调用过程可修改
- %r10, %r11
  - 调用者保存
  - 被调用过程可修改



x86-64 Linux的寄存器用法#2

- `%rbx`, `%r12`, `%r13`, `%r14`
  - 被调用者保存并恢复
- `%rbp`
  - 被调用者保存并恢复
  - 可用作栈帧指针
- `%rsp`
  - 被调用者保存的特殊形式
  - 在离开过程时，恢复为原始值(CALL之前的值)
- 压入栈中的参数，谁来清除?
  - 编程语言的约定/权利范围
  - C语言：调用者清除，简洁的方式：add $32, %rsp

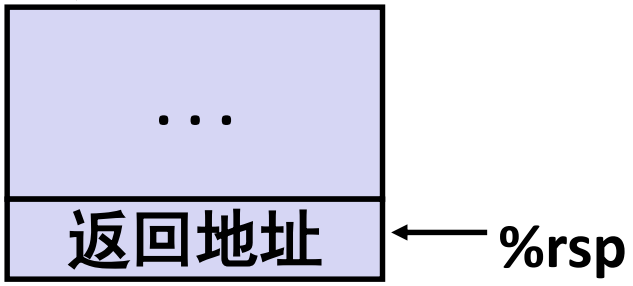


被调用者保存——实例#1

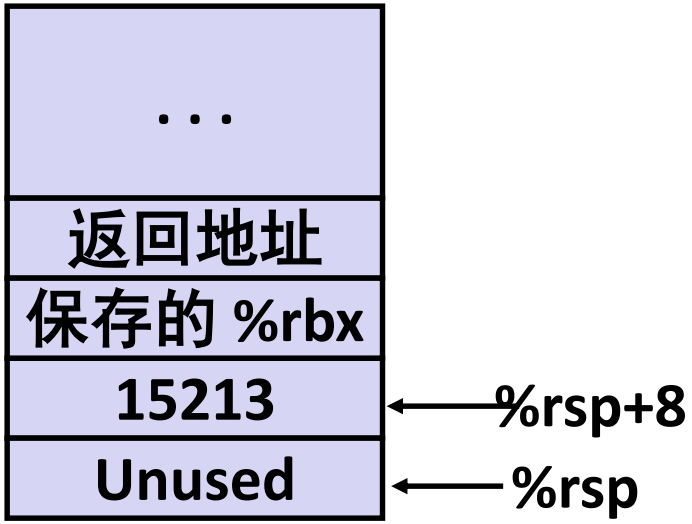
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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
```

起始的栈结构



运行中的栈结构

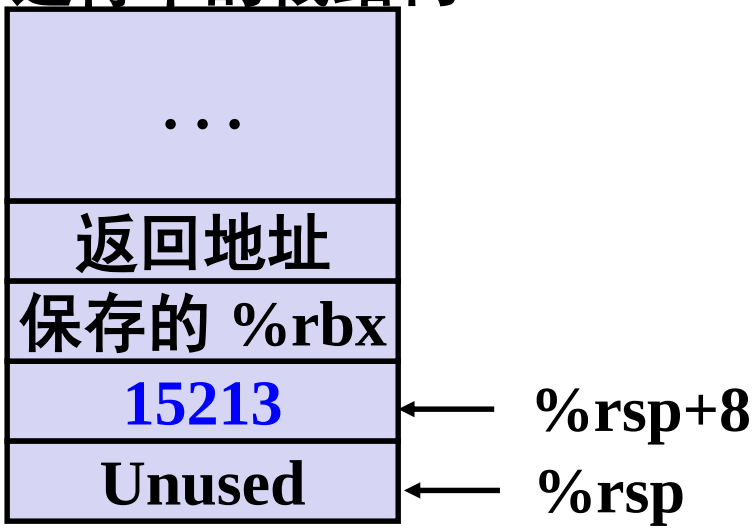


被调用者保存——实例#2

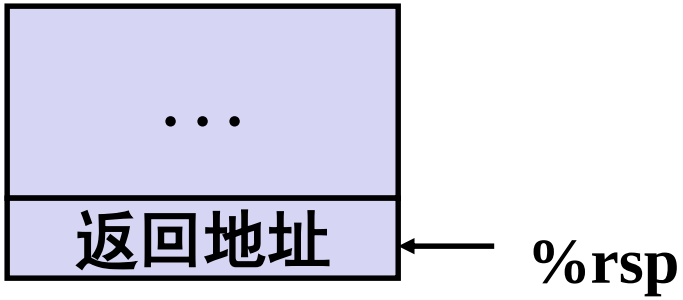
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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
```

运行中的栈结构



ret执行前的栈结构



主要内容

- 过程
  - 栈结构
  - 调用约定
    - 传递控制
    - 传递数据
    - 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 # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数的终止条件

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

寄存器	用途	类型
%rdi	x	参数
%rax	返回值	返回值

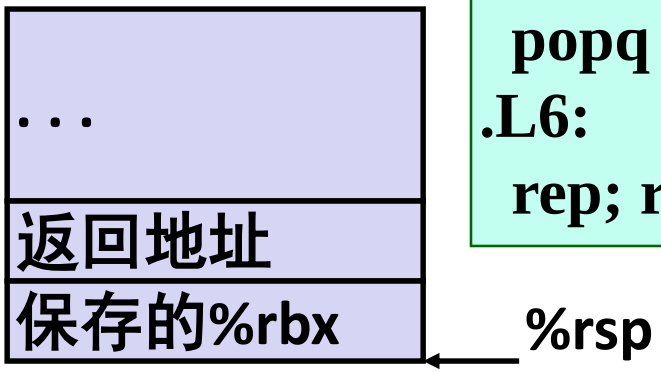
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数的寄存器保存

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

寄存器	用途	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 # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```



递归函数的调用创建

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

寄存器	用途	类型
%rdi	x >> 1	Rec. 参数
%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	用途	类型
%rbx	x & 1	被调用者保存
%rax	递归调用的返回值	

```
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	用途	类型
%rbx	x & 1	被调用者保存
%rax	返回值	

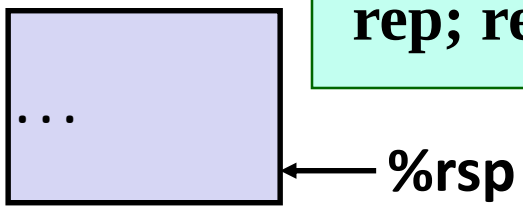
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

递归函数的完成

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

寄存器	用途	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 # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```



递归的观察

- 递归无需特殊的处理
  - 栈帧意味着每个函数调用有私有的存储
    - 保存的寄存器 、局部变量
    - 保存的返回地址
  - 寄存器保存约定:防止函数调用损毁其他函数/调用的数据
    - 除非C代码明确地这样做（如缓冲区溢出）。
  - 栈的使用原则：遵循 调用/返回 模式
    - 如 P调用Q, 然后Q 在P结束之前返回
    - 后入、先出
- 对互递归同样有效
  - P调用Q; Q调用P

完整汇编函数代码

- 使用C风格的函数，计算圆的面积：
  "functest3.s"

C语言函数传参、栈平衡1/2

```
long sum8para(long x1,long x2,long x3,long x4,
              long x5,long x6,long x7,long x8)
{
   long t = x1+x2+x3+x4+x5+x6+x7+x8;
    return t;
}

int main() {
 long d=0;
 d=sum8para(1,2,3,4,5,6,7,8);
 return 0;
}
```

编译指令：gcc -S -Og main.c -o main64.s

```
// main64.s
…
pushq $8
pushq $7
movl  $6, %r9d
movl  $5, %r8d
movl  $4, %ecx
movl  $3, %edx
movl  $2, %esi
movl  $1, %edi
call  sum8para
addq  $16, %rsp
…
```

C语言函数传参、栈平衡2/2

```
long sum8para(long x1,long x2,long x3,long x4,
              long x5,long x6,long x7,long x8)
{
   long t = x1+x2+x3+x4+x5+x6+x7+x8;
    return t;
}

int main() {
   long d=0;
   d=sum8para(1,2,3,4,5,6,7,8);
   return 0;
}
```

编译指令：gcc -S -m32 -Og main.c -o main32.s

```
// main32.s
…
pushl  $8
pushl  $7
pushl  $6
pushl  $5
pushl  $4
pushl  $3
pushl  $2
pushl  $1
call   sum8para
addl   $32, %esp
…
```

x86-64 过程总结

- 要点
  - 栈是实现过程调用/返回所依赖的数据结构
  - 如P调用Q，则Q先返回P后返回
- 用正常调用约定处理递归(互递归)
  - 可安全保存数值的地方：栈帧、被调用者保存的寄存器
  - 函数调用前将参数置于栈顶
  - 返回结果在 %**rax**中
- 指针就是数值的地址:
  - 在栈中的或是全局的