机器级编程 2
Machine-Level Programming II

课 程 名 ：计算机系统

第 4 讲 （2025 年 4 月 28 日）

主 讲 人 ：杜海文

本课内容

- `switch` 语句
- 过程
  - 栈结构
  - 调用约定
    - 控制传递
    - 数据传递
    - 管理局部数据

Switch 语句示例

- （注：本例为教材§3.6.8 例程，可在本地复现）
- 多重 case 标号
  - 本例：104, 106
- 穿透fall through分支
  - 本例：102
- 缺少的分支
  - 本例：101, 105

```
long switch_eg1(long x, long n,
                long *dest)
{
    long val = x;
    switch (n) {
    case 100:
        val *= 13;
        break;
    case 102:
        val += 10;
    case 103:
        val += 11;
        break;
    case 104:
    case 106:
        val *= val;
        break;
    default:
        val = 0;
    }
    *dest = val;
}
```

跳转表jump table结构

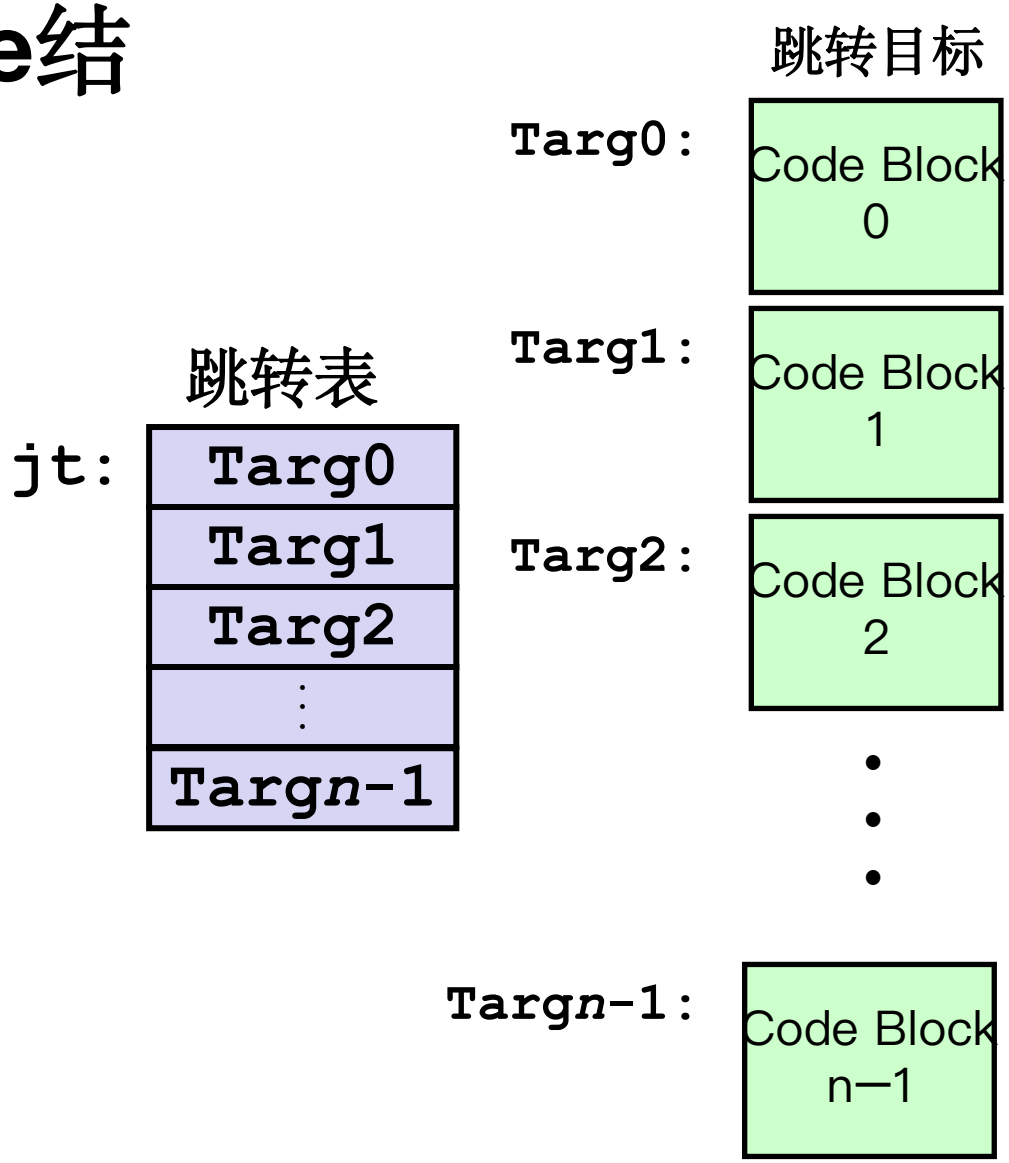
Switch 形式

```
switch (x) {
case val_0:
     Block 0
case val_1:
     Block 1
. . .
case val_n-1:
     Block n–1
}
```

翻译

```
goto *jt[x];
```

非标准 C 功能见 GCC 扩展



Switch 语句示例 1 (续)

```
long switch_eg1(long x, long n, long *dest)
{
    long val = x;
    switch (n) {
    . . .
    }
    *dest = val;
}
```

寄存器	取值
%rdi	Argument x
%rsi	Argument n
%rdx	Argument dest
%rax	Return value

准备工作:

```
switch_eg1:
    subq    $100, %rsi
    cmpq    $6, %rsi          # n-100 : 6
    ja      .L8
    jmp     *.L4(,%rsi,8)
```

执行 default 分支的 n 的取值范围是?

Switch 语句示例 1 (续)

```
long switch_eg1(long x, long n,
                long *dest)
{
    long val = x;
    switch (n) {
    . . .
    }
    *dest = val;
}
```

跳转表

```
  .section    .rodata
  .align 8
.L4:
  .quad .L3   # case 100
  .quad .L8
  .quad .L5   #      102
  .quad .L6   #      103
  .quad .L7   #      104
  .quad .L8
  .quad .L7   #      106
```

准备工作:

```
    switch_eg1:
        subq    $100, %rsi
        cmpq    $6, %rsi          # n-100 : 6
        ja      .L8               # Use default
        jmp     *.L4(,%rsi,8)     # goto *jt[n-100]
```

间接跳转

汇编代码说明

- 表结构
  - 每项占 8 字节
  - 以 .L4 为基地址
- 无条件转移
  - 直接：jmp .L8
    - 转移目标由 .L8 指明
  - 间接：jmp *.L4(,%rsi,8)
    - 跳转表首地址：.L4
    - 表索引号乘以 8 (每个地址占 8 字节)
    - 从地址 .L4 + (n-100)*8 处获取到转移目标地址
    - 仅对 100 ≤ n ≤ 106 作上述处理

.rodata 节
详见第 7 章

跳转表

```
  .section    .rodata
  .align 8
.L4:
  .quad .L3     # n = 100
  .quad .L8
  .quad .L5     #     102
  .quad .L6     #     103
  .quad .L7     #     104
  .quad .L8
  .quad .L7     #     106
```

各伪操作详见
as 汇编器手册

跳转表

```
  .section   .rodata
  .align 8
.L4:
  .quad  .L3 # n = 100
  .quad  .L8
  .quad  .L5 #     102
  .quad  .L6 #     103
  .quad  .L7 #     104
  .quad  .L8
  .quad  .L7 #     106
```

```
    switch (n) {
    case 100:          // .L3
        val *= 13;
        break;
    case 102:          // .L5
        val += 10;
        /* fall through */
    case 103:          // .L6
        val += 11;
        break;
    case 104:
    case 106:          // .L7
        val *= val;
        break;
    default:           // .L8
        val = 0;
    }
    *dest = val;
```

各代码块说明 (n == 100)

```
    switch (n) {
    case 100:           // .L3
        val *= 13;
        break;
    . . .
    }
```

寄存器	取值
%rdi	Argument x
%rsi	Argument n
%rdx	Argument dest
%rax	Return value

```
.L3:
   leaq    (%rdi,%rdi,2), %rax      # 3*x
   leaq    (%rdi,%rax,4), %rdi      # val = 13*x
   jmp     .L2                      # Goto done
```

各代码块说明 (n == 102, 103)

```
    switch (n) {
    . . .
    case 102:
        val += 10;
        /* fall through */
    case 103:
        val += 11;
        break;
    . . .
    }
```

```
.L5:                            # Case 102
   addq    $10, %rdi            # x = x + 10
.L6:                            # Case 103
   addq    $11, %rdi            # val = x + 11
   jmp     .L2                  # Goto done
```

寄存器	取值
%rdi	Argument x
%rsi	Argument n
%rdx	Argument dest
%rax	Return value

各代码块说明 (x == 104, 106, default)

```
    switch (n) {
    . . .
    case 104:          // .L7
    case 106:          // .L7
        val *= val;
        break;
    default:           // .L8
        val = 0;
    }
```

```
.L7:                        # Case
104,106
   imulq   %rdi, %rdi       # val = x * x
   jmp     .L2              # Goto done
.L8:                        # Default:
   movl    $0, %edi         # val = 0
.L2:
   movq    %rdi, (%rdx)
   ret
```

寄存器	取值
%rdi	Argument x
%rsi	Argument n
%rdx	Argument dest
%rax	Return value

Switch 语句示例 2

- （注：本例为原课件例程，不易在本地复现）
- 多重 case 标号
  - 本例：5 和 6
- 穿透fall through分支
  - 本例：2
- 缺少的分支
  - 本例：4

```
long switch_eg2
    (long x, long y, long z)
{
    long w = 1;
    switch (x) {
    case 1:
        w = y * z;
        break;
    case 2:
        w = y / z;
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

跳转表jump table结构

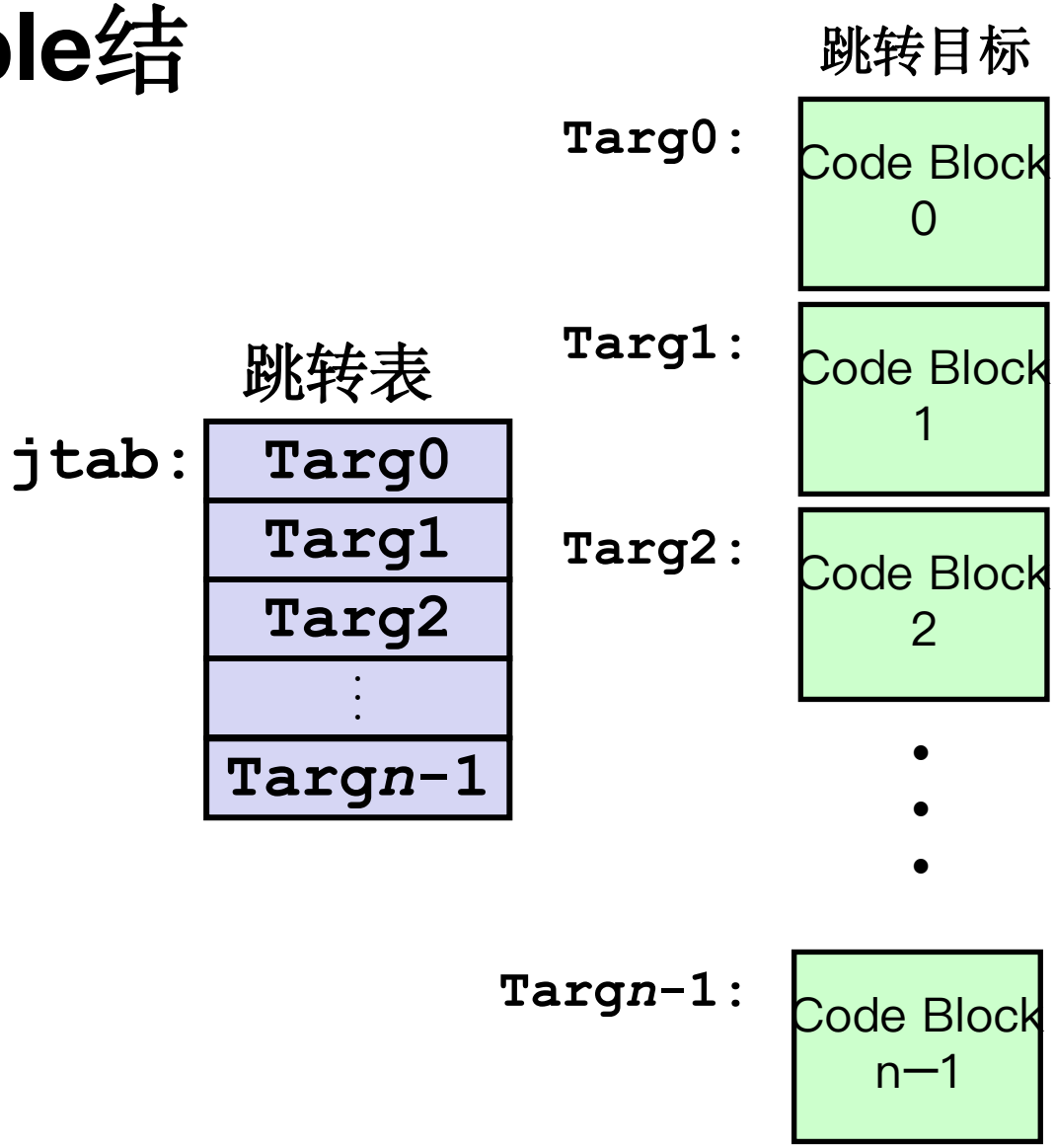
Switch 形式

```
switch (x) {
case val_0:
     Block 0
case val_1:
     Block 1
. . .
case val_n-1:
     Block n—1
}
```

翻译

```
goto *jtab[x];
```

非标准 C 功能见 GCC 扩展



Switch 语句示例（续）

```
long switch_eg2(long x, long y, long z)
{
    long w = 1;
    switch (x) {
    . . .
    }
    return w;
}
```

寄存器	取值
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

准备工作:

```
switch_eg2:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8
    jmp     *.L4(,%rdi,8)
```

注意：此例中 w 并未初始化

执行 default 分支的 x 取值范围？

Switch 语句示例（续）

```
long switch_eg2
    (long x, long y, long z)
{
    long w = 1;
    switch (x) {
    . . .
    }
    return w;
}
```

跳转表

```
  .section    .rodata
  .align 8
.L4:
  .quad .L8    # x = 0
  .quad .L3    #     1
  .quad .L5    #     2
  .quad .L9    #     3
  .quad .L8    #     4
  .quad .L7    #     5
  .quad .L7    #     6
```

准备工作:

```
    switch_eg2:
        movq    %rdx, %rcx
        cmpq    $6, %rdi        # x:6
        ja      .L8             # Use default
        jmp     *.L4(,%rdi,8)   # goto *jtab[x]
```

间接跳转 →

汇编代码说明

- 表结构
  - 每项占 8 字节
  - 以 .L4 为基地址

- 无条件转移
  - 直接：jmp .L8
    - 转移目标由 .L8 指明
  - 间接：jmp *.L4(,%rdi,8)
    - 跳转表首地址：.L4
    - 表索引号乘以 8 (每个地址占 8 字节)
    - 从地址 .L4 + x*8 处获取到转移目标地址
    - 仅对 0 ≤ x ≤ 6 作上述处理

.rodata 节
详见第 7 章

跳转表

```
   .section      .rodata
   .align 8
.L4:
   .quad .L8     # x = 0
   .quad .L3     #     1
   .quad .L5     #     2
   .quad .L9     #     3
   .quad .L8     #     4
   .quad .L7     #     5
   .quad .L7     #     6
```

各伪操作详见
as 汇编器手册

跳转表

```
  .section   .rodata
  .align 8
.L4:
  .quad   .L8  # x = 0
  .quad   .L3  #     1
  .quad   .L5  #     2
  .quad   .L9  #     3
  .quad   .L8  #     4
  .quad   .L7  #     5
  .quad   .L7  #     6
```

```
    switch (x) {
    case 1:       // .L3
        w = y * z;
        break;
    case 2:       // .L5
        w = y / z;
        /* fall through */
    case 3:       // .L9
        w += z;
        break;
    case 5:
    case 6:       // .L7
        w -= z;
        break;
    default:      // .L8
        w = 2;
    }
```

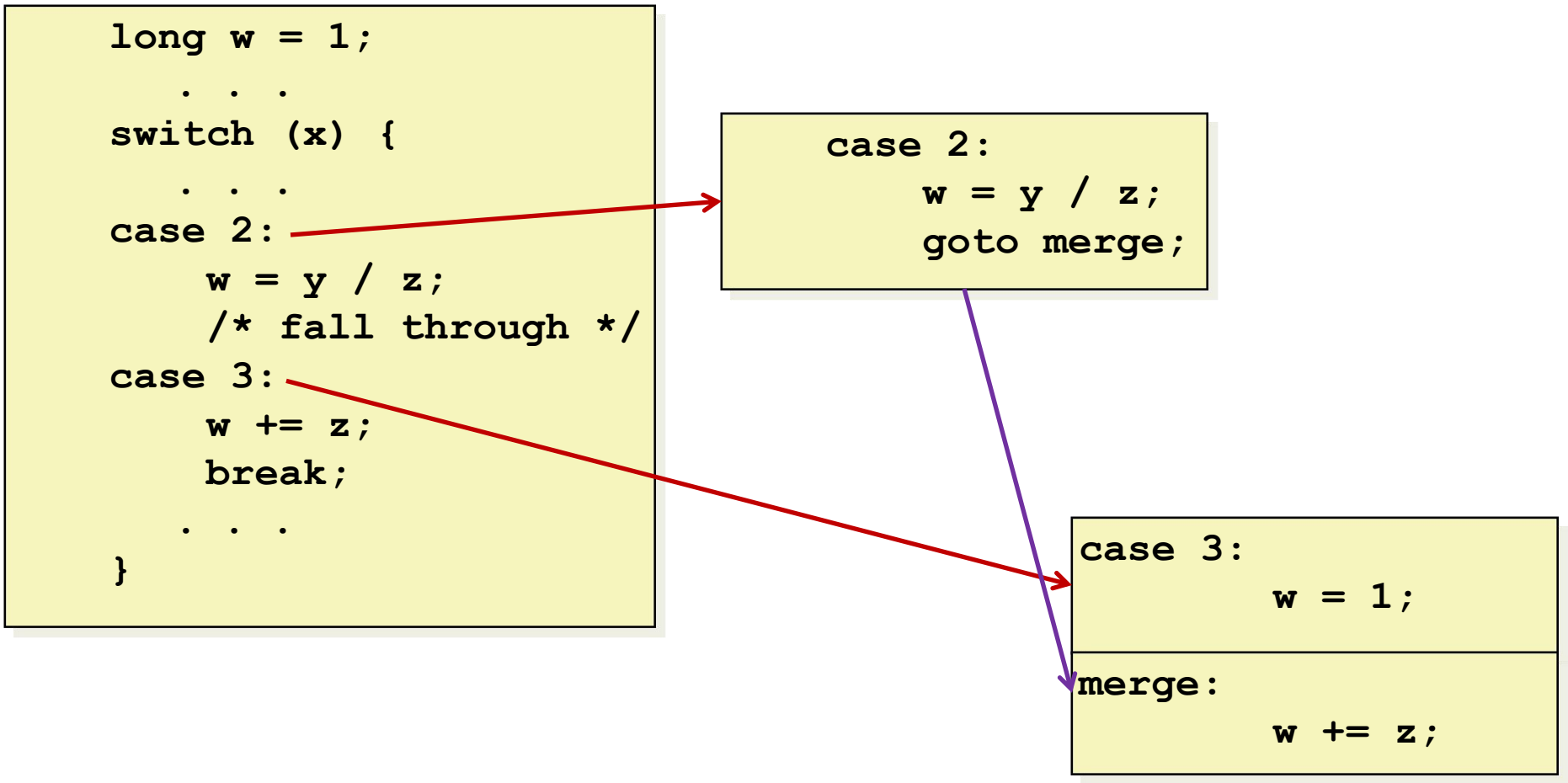
各代码块说明 (x == 1)

```
switch (x) {
case 1:      // .L3
    w = y * z;
    break;
. . .
}
```

```
.L3:
   movq    %rsi, %rax   # y
   imulq   %rdx, %rax   # y*z
   ret
```

寄存器	取值
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

处理自然下落fall-through



各代码块说明 (x == 2, x == 3)

```
    long w = 1;
       . . .
    switch (x) {
       . . .
    case 2:
        w = y / z;
        /* fall through */
    case 3:
        w += z;
        break;
       . . .
    }
```

```
.L5:                        # Case 2
   movq    %rsi, %rax
   cqto                     # Figure 3.12
   idivq   %rcx             #  y/z
   jmp     .L6              #  goto merge
.L9:                        # Case 3
   movl    $1, %eax         #  w = 1
.L6:                        # Merge:
   addq    %rcx, %rax       #  w += z
   ret
```

寄存器	取值
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

各代码块说明 (x == 5, 6, default)

```
  switch (x) {
  . . .
  case 5:  // .L7
  case 6:  // .L7
      w -= z;
      break;
  default: // .L8
      w = 2;
}
```

```
.L7:                    # Case 5,6
  movl   $1, %eax       #  w = 1
  subq   %rdx, %rax     #  w -= z
  ret
.L8:                    # Default:
  movl   $2, %eax       #  2
  ret
```

寄存器	取值
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

小结

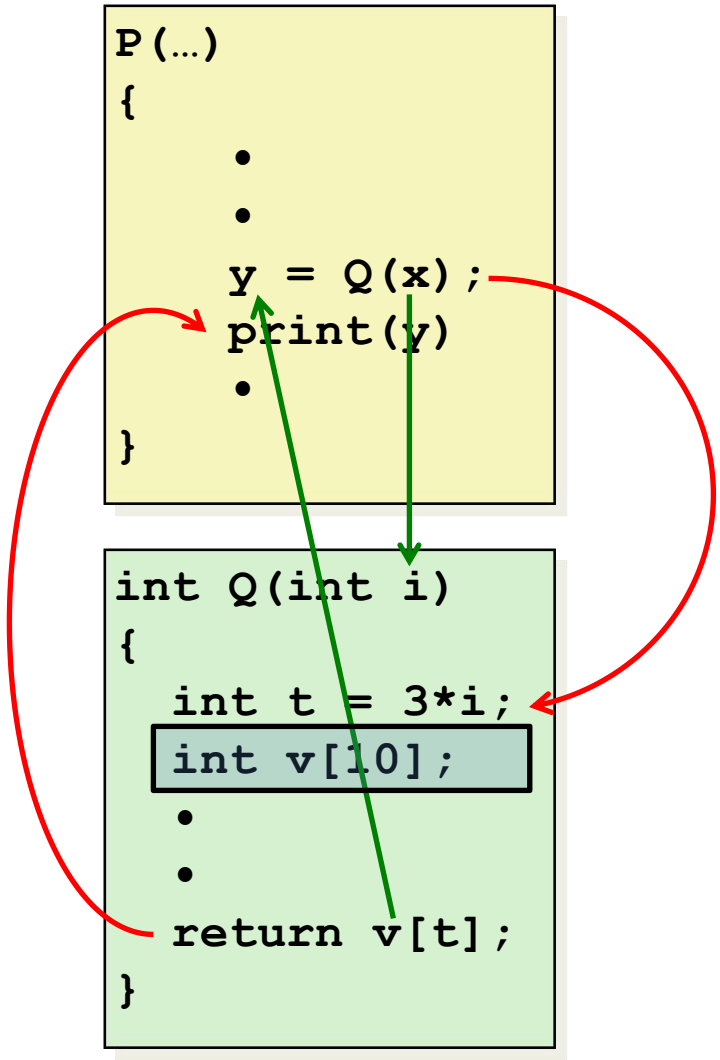
- C 控制结构
  - `if-then-else`
  - `do-while`
  - `while, for`
  - `switch`
- 汇编级控制结构
  - 条件转移conditional jump
  - 条件传送conditional move
  - 间接转移indirect jump（通过跳转表实现）
  - 编译器生成代码序列，实现更加复杂的控制
- 标准的实现方法
  - 将循环转化为do-while 或 jump-to-middle 形式
  - 分支较多的 `switch` 语句使用跳转表实现
  - 分支稀疏的 `switch` 语句可采用决策树（`if-elseif-elseif-else`）

本课内容

- `switch` 语句
- **过程**
  - **栈结构**
  - 调用约定
    - 控制传递
    - 数据传递
    - 管理局部数据

过程procedure采用的机制

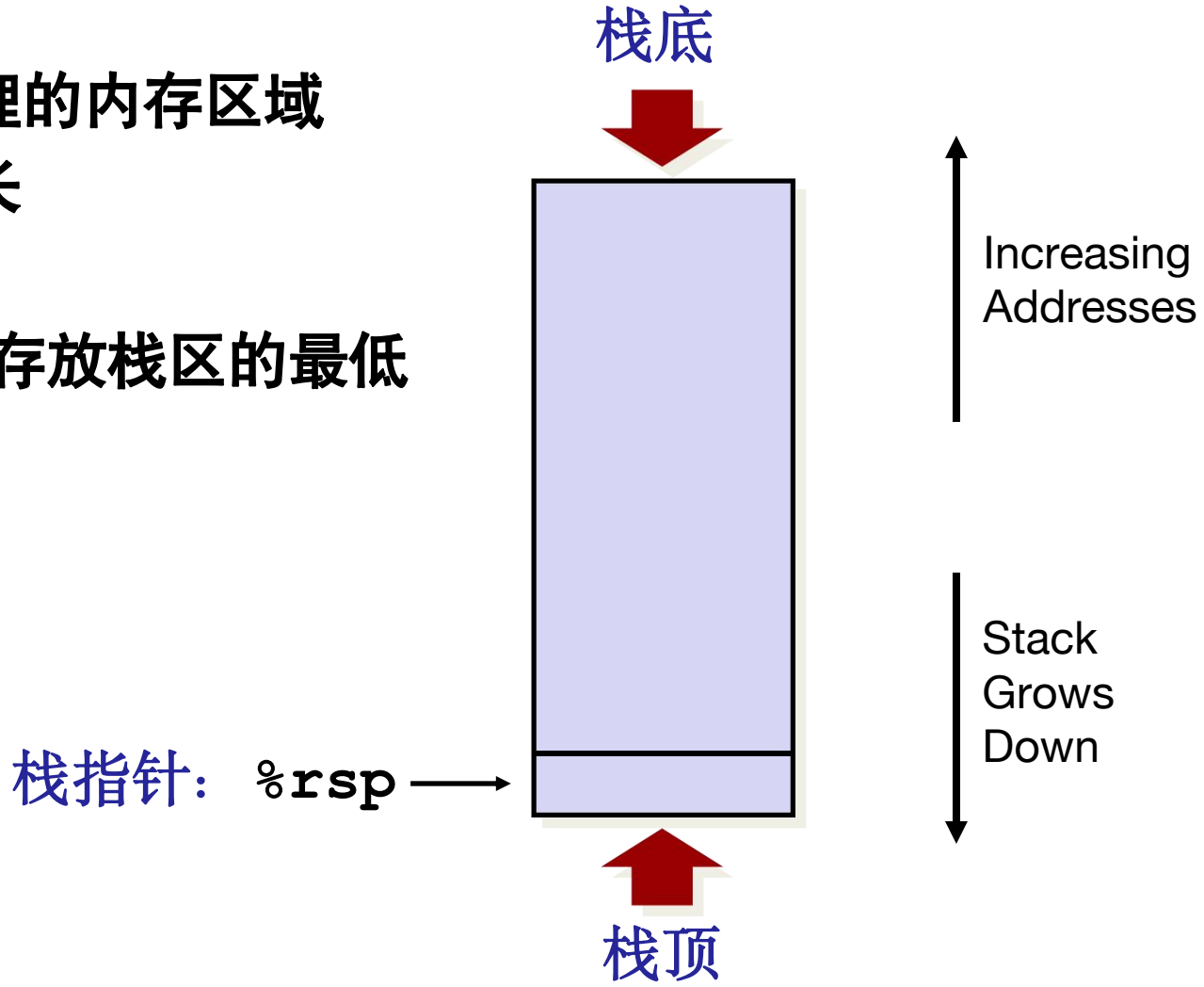
- 控制传递
  - 转移至过程代码的起始处
  - 过程执行完毕后回到正确的返回点
- 数据传递
  - 过程参数
  - 返回值
- 内存管理
  - 过程执行阶段分配内存
  - 过程返回时释放内存
- 所有方式方法均通过机器指令实现
- x86-64 实现过程调用时采取按需开销 （非必要不花销）策略



x86-64 的栈管理

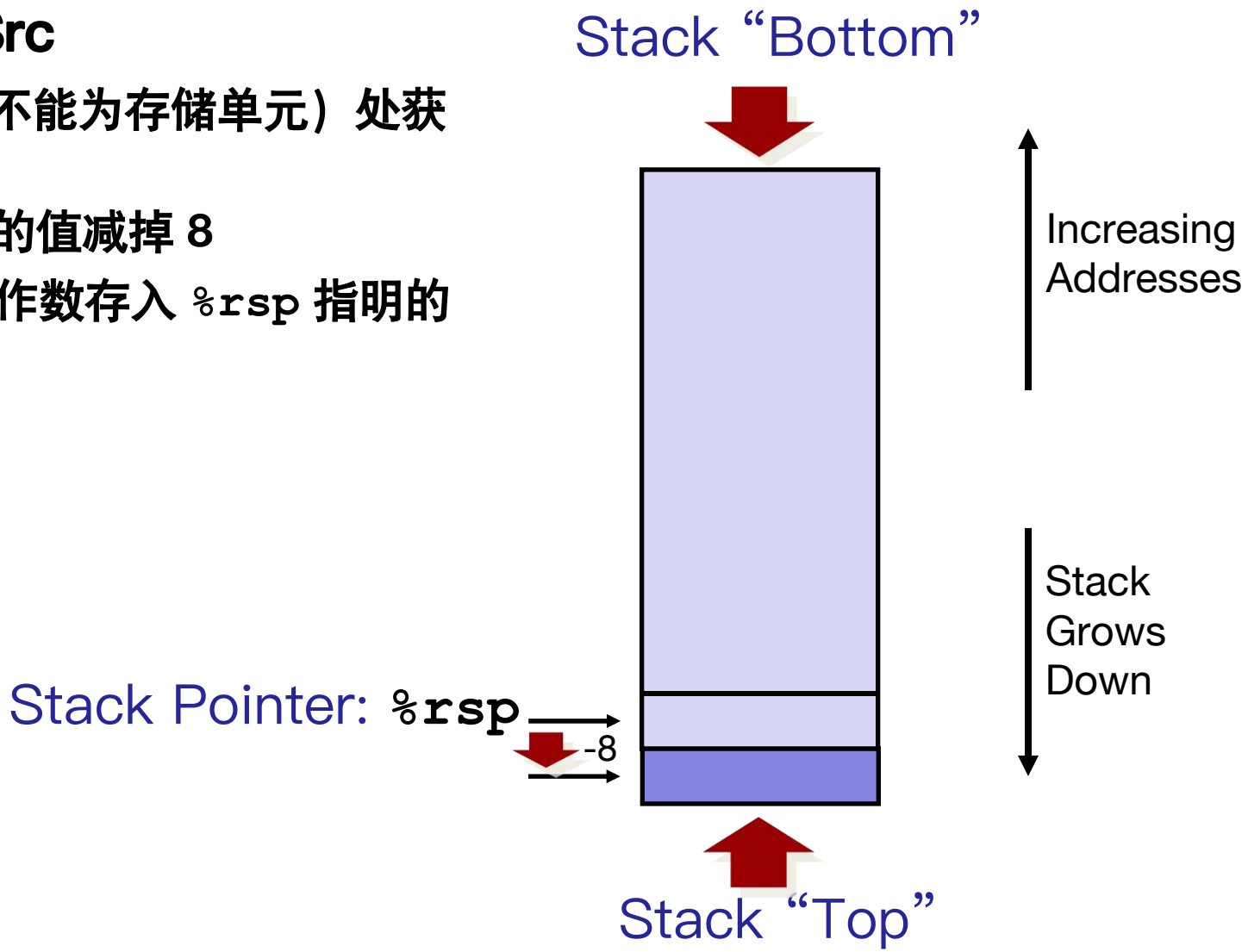
- 按照栈的规则管理的内存区域
- 向低地址方向生长

- 寄存器 `%rsp` 中存放栈区的最低地址
  - 栈顶元素的地址



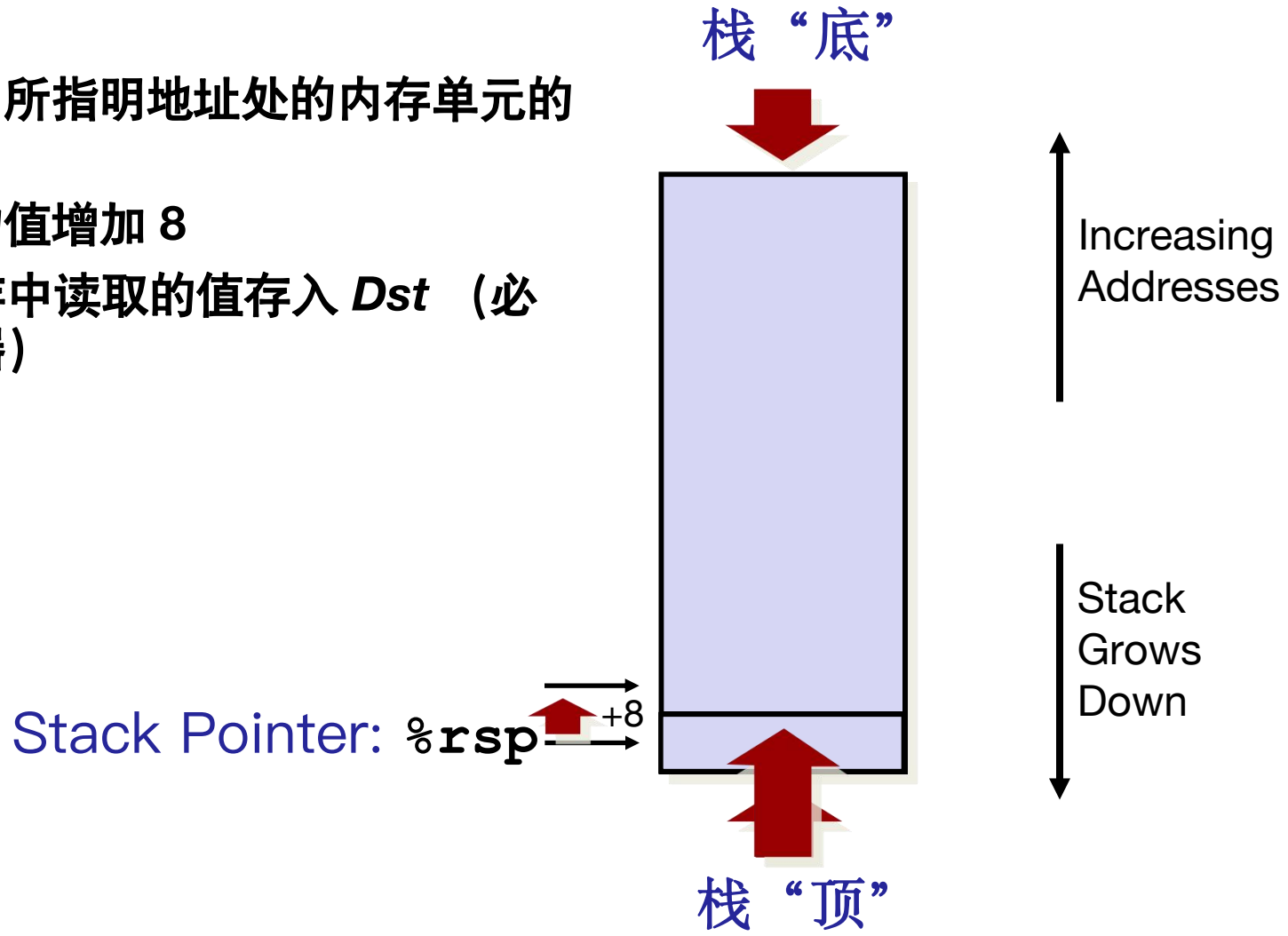
x86-64 的栈管理：入栈push

- `pushq` Src
  - 从 *Src*（不能为存储单元）处获取操作数
  - 将 `%rsp` 的值减掉 8
  - 将上述操作数存入 `%rsp` 指明的地址



x86-64 的栈管理： 出栈pop

- `popq` Dst
  - 读取 `%rsp` 所指明地址处的内存单元的内容
  - 将 `%rsp` 的值增加 8
  - 将上述内存中读取的值存入 *Dst* （必须是寄存器）



本课内容

- `switch` 语句
- **过程**
  - 栈结构
  - **调用约定**
    - **控制传递**
    - 数据传递
    - 管理局部数据

示例代码

```
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
```

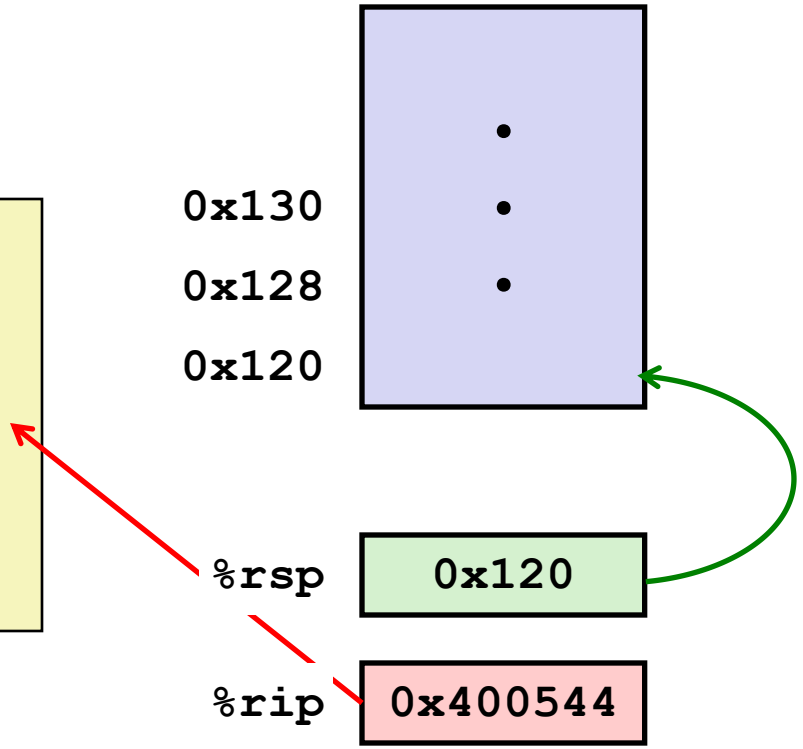
过程控制流

- 利用栈帮助实现过程的调用和返回
- 过程调用：`call` *label*
  - 将返回地址压栈
  - 转移至 label
- 返回地址：
  - 紧跟在 `call` 指令之后的下一条指令的地址
  - Example from disassembly
- 过程返回：`ret`
  - 将返回地址从栈顶弹出
  - 转移至该地址处继续执行

控制流示例 #1

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

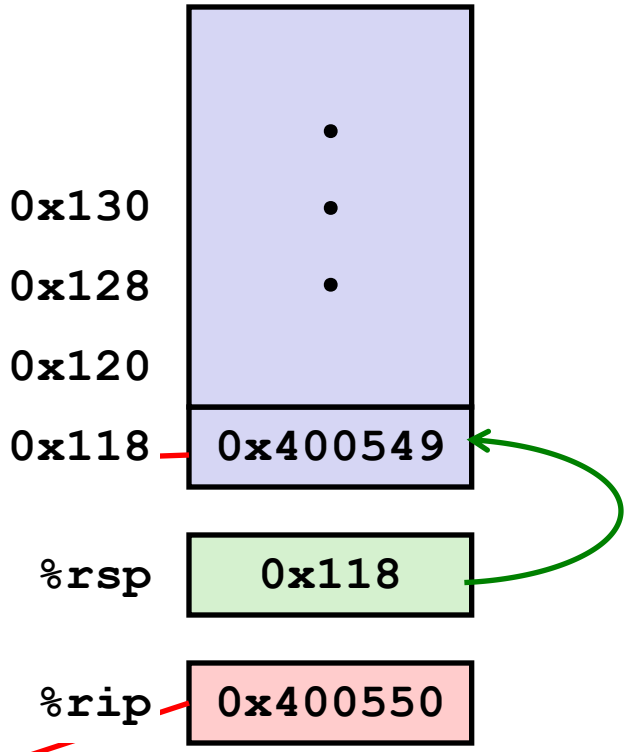
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



控制流示例 #2

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

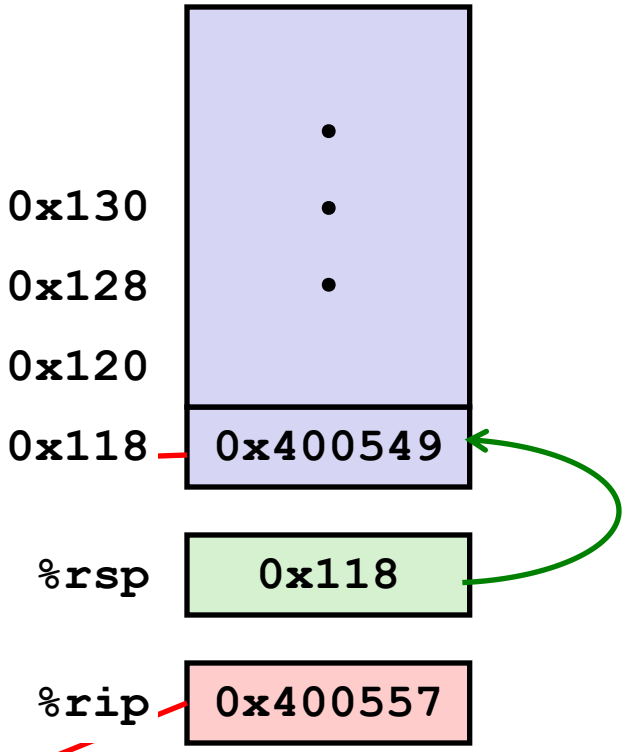
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



控制流示例 #3

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

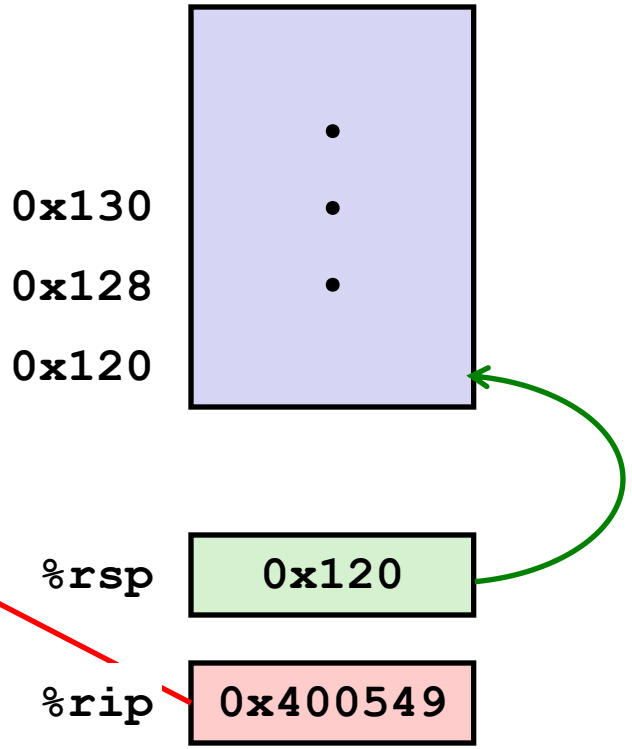
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



控制流示例 #4

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

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



本课内容

- `switch` 语句
- **过程**
  - 栈结构
  - **过程调用的相关约定**
    - 控制传递
    - **数据传递**
    - 管理局部数据

过程数据流

寄存器

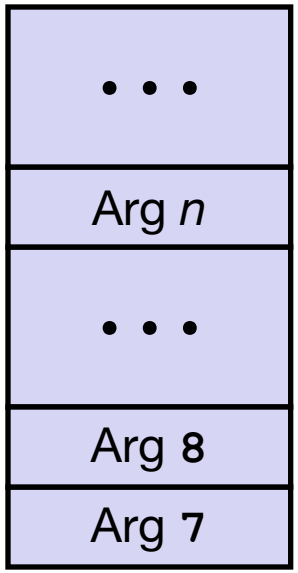
- 前 6 个参数

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

- 返回值

%rax

栈



- 仅在需要的时候分配栈空间

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
```

本课内容

- `switch` 语句
- **过程**
  - **栈结构**
  - **调用约定**
    - 控制传递
    - 数据传递
    - **管理局部数据**

基于栈的编程语言

- **支持递归的语言**
  - **例：C、Pascal、Java**
  - **代码必须可以“重入”**
    - 同一过程的多个实例同时运行
  - **需要有地方存储每个运行实例的当前状态**
    - 参数
    - 局部变量
    - 返回指针return pointer
- **栈规则**
  - **一个过程的状态仅持续有限时间**
    - 从被调用时起，到返回时止
  - **被调过程先于主调过程返回**
- **栈以帧的形式分配**
  - **单个过程实例的状态**

调用链示例

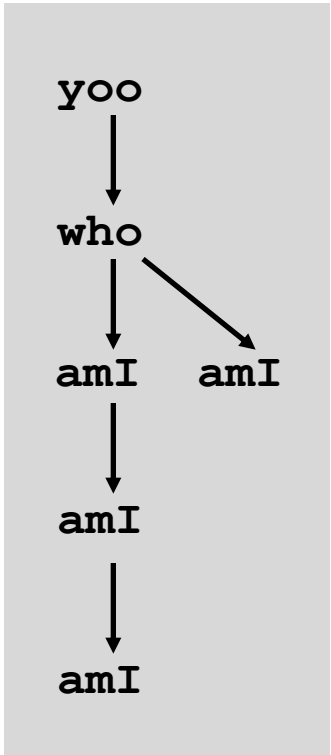
```
yoo(...)
{
      •
      •
      who();
      •
      •
}
```

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

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

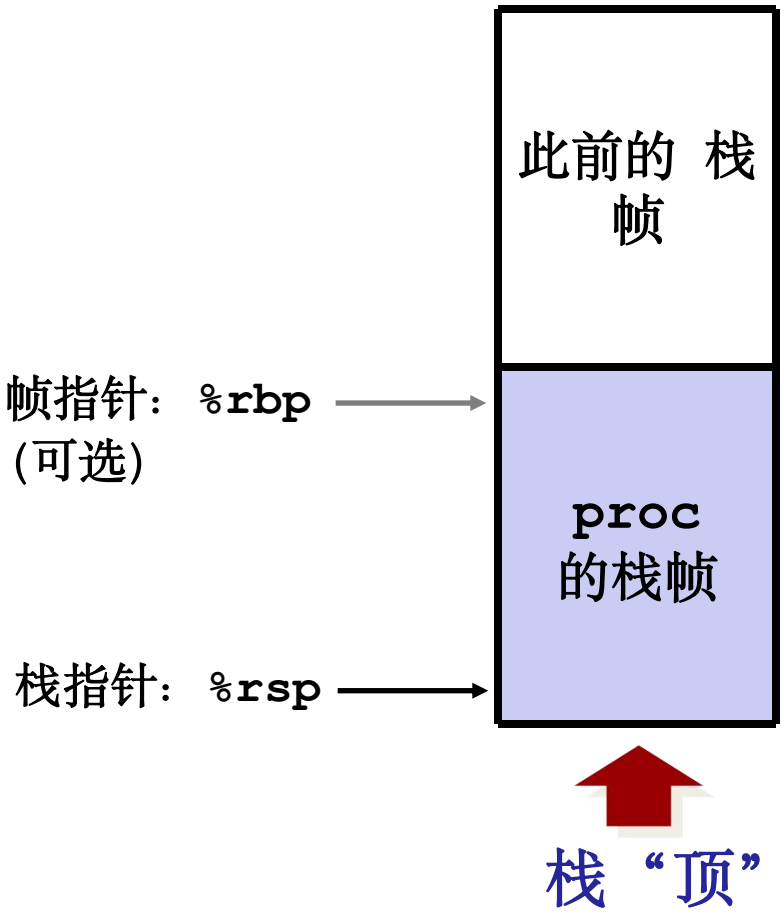
`amI()` is recursive

Example
Call Chain

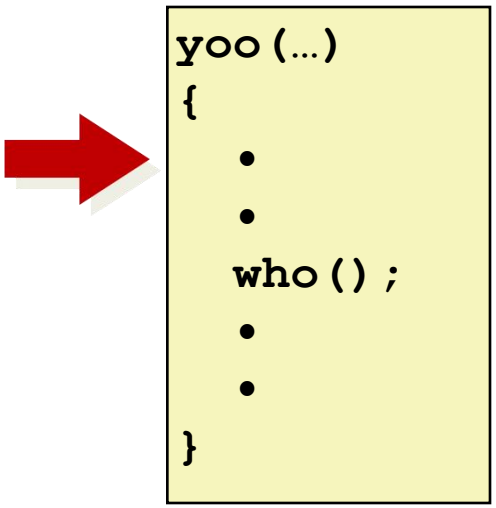


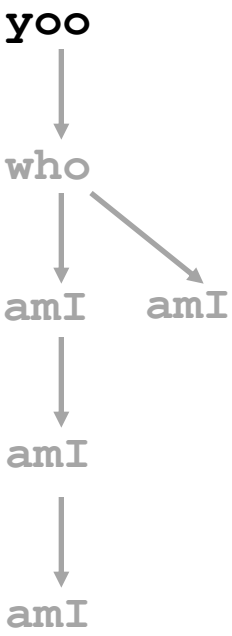
栈帧Stack Frames

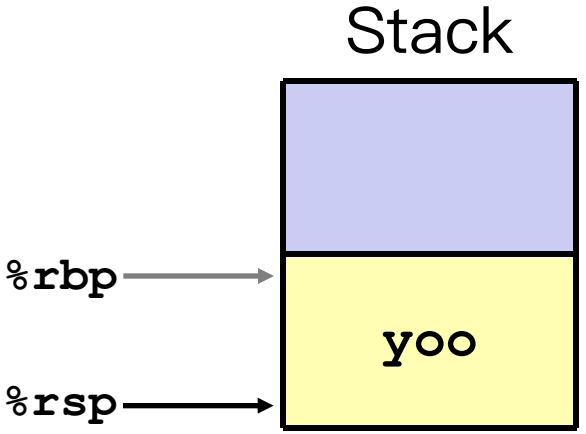
- 内容
  - 返回信息
  - 本地存储（按需分配）
  - 临时空间（按需分配）
- 管理
  - 进入过程时完成空间分配
    - “Set-up” code
    - 包括 `call` 指令的压栈操作
  - 从过程返回时释放空间
    - “Finish” code
    - 包括由 `ret` 指令执行的 pop 操作



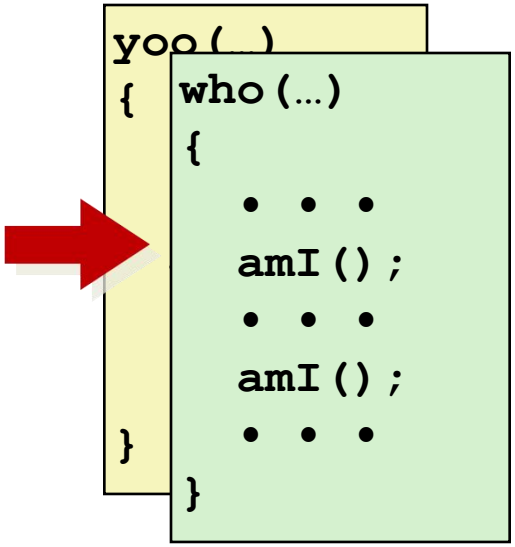
Example

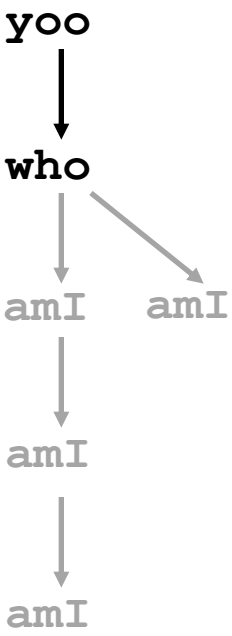


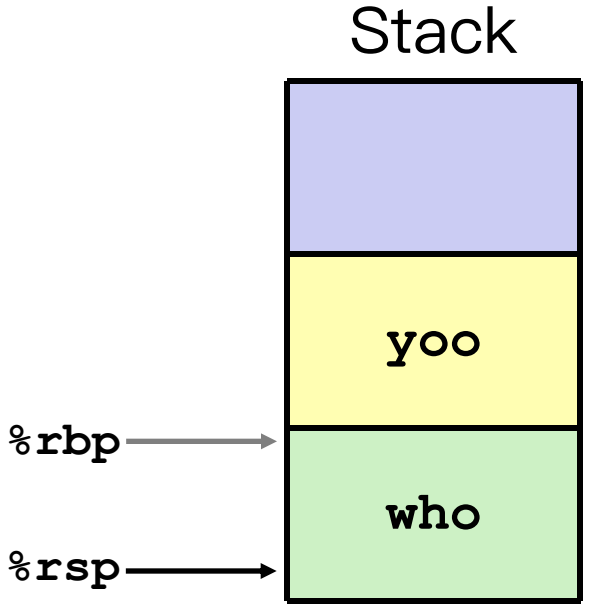




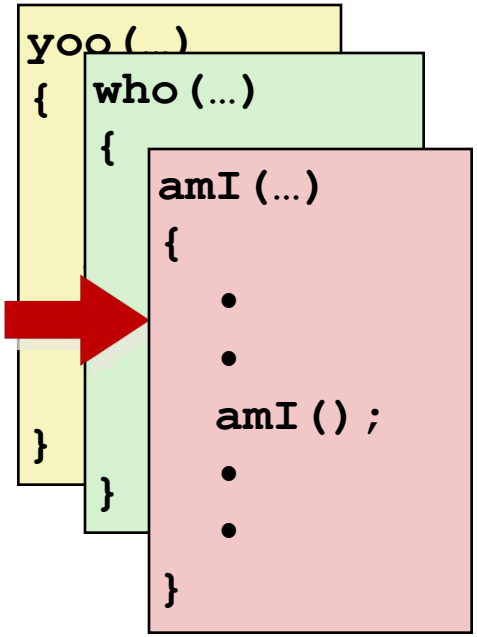
Example

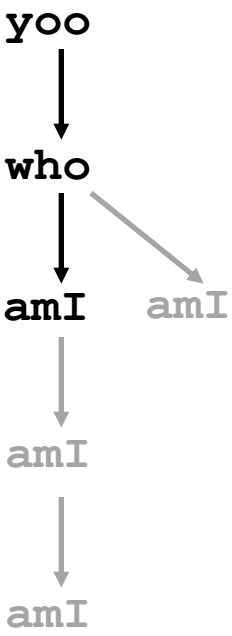


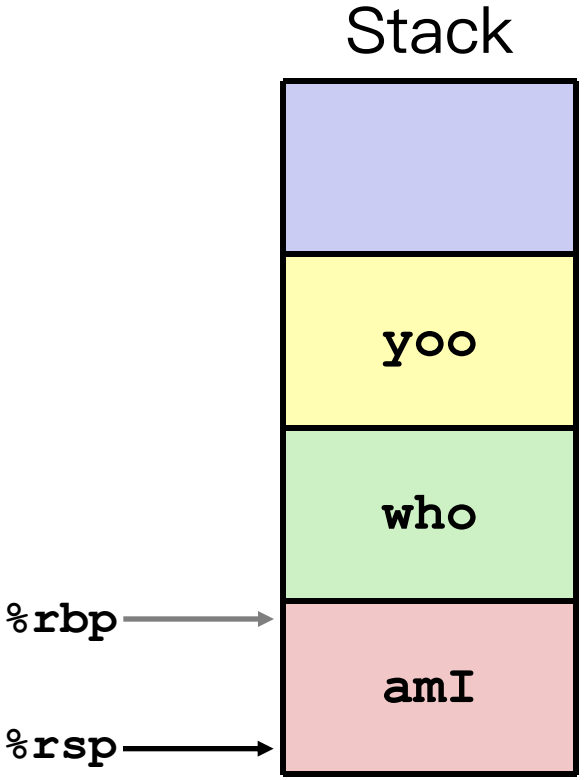




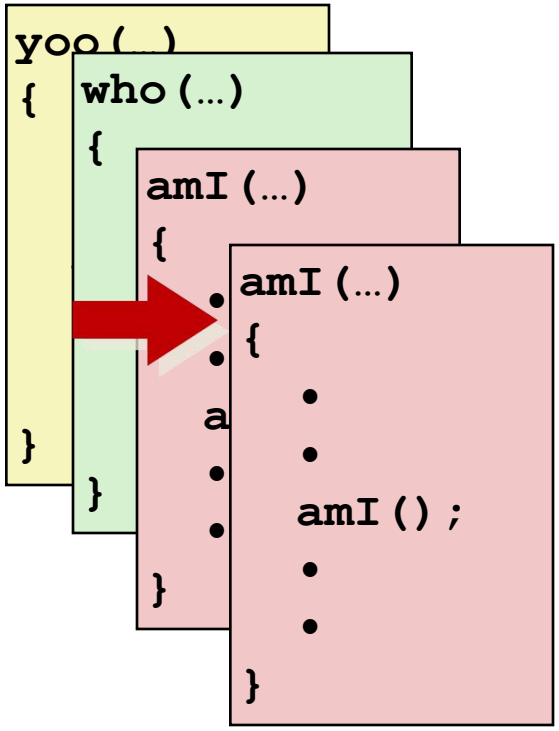
Example

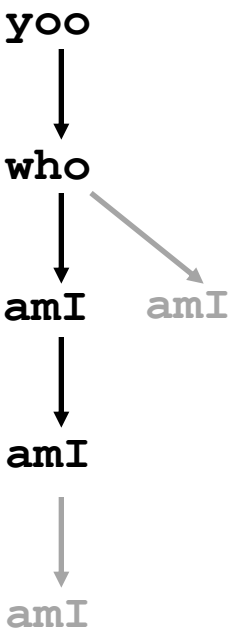

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

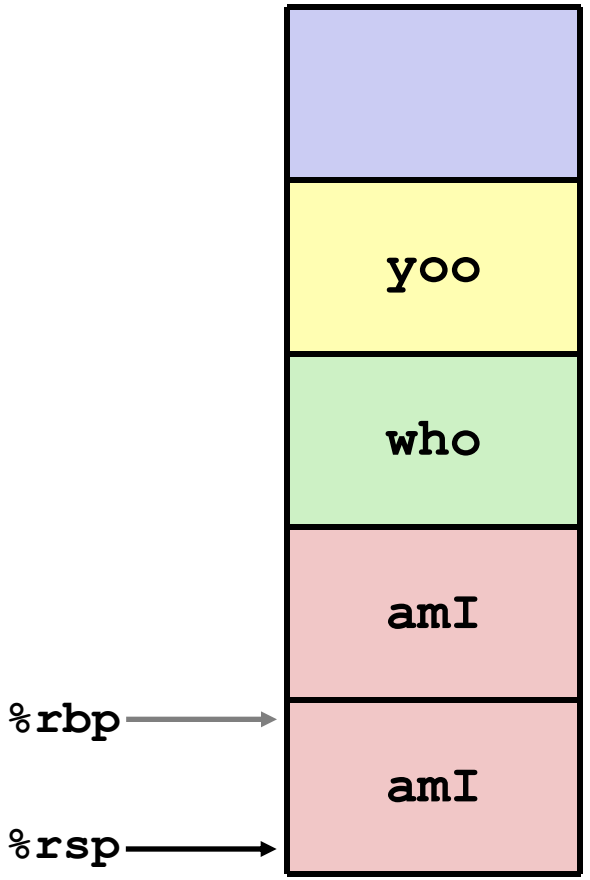

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

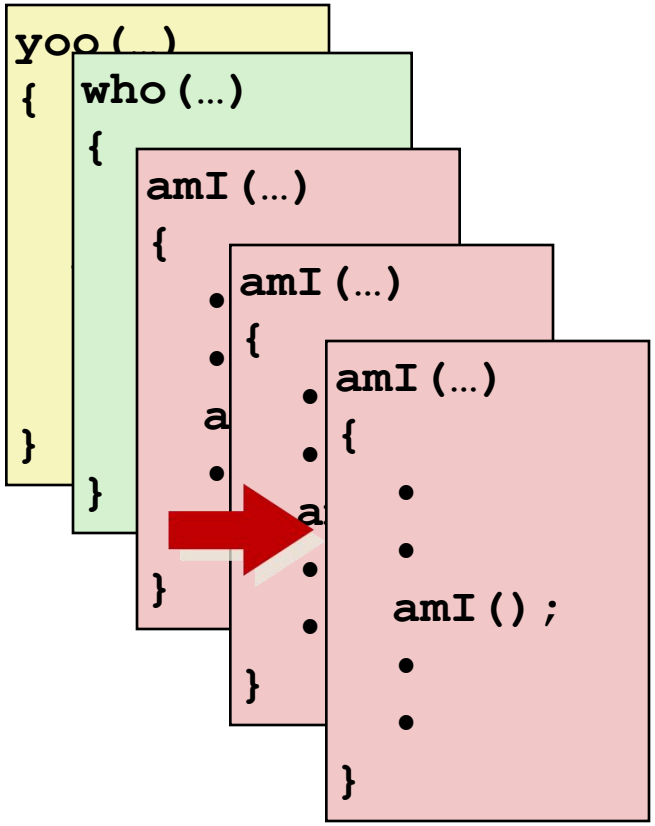
Example

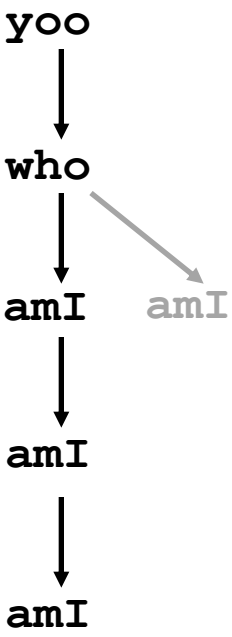

yoo(...)
{
who(...)
{
amI(...)
{
amI(...)
{
amI();
}
}
}
}

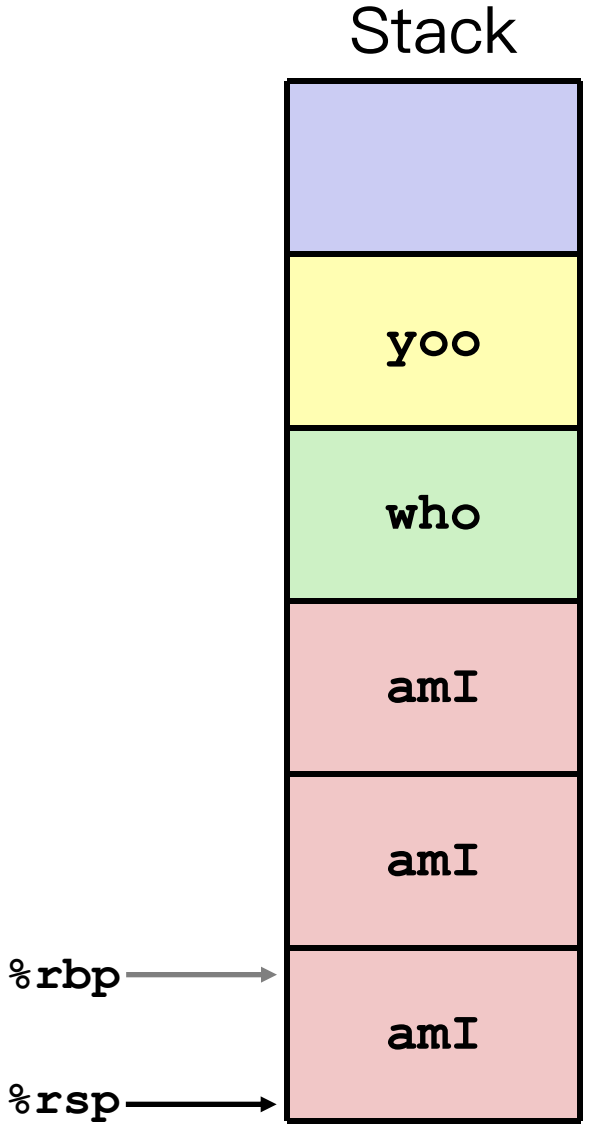

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

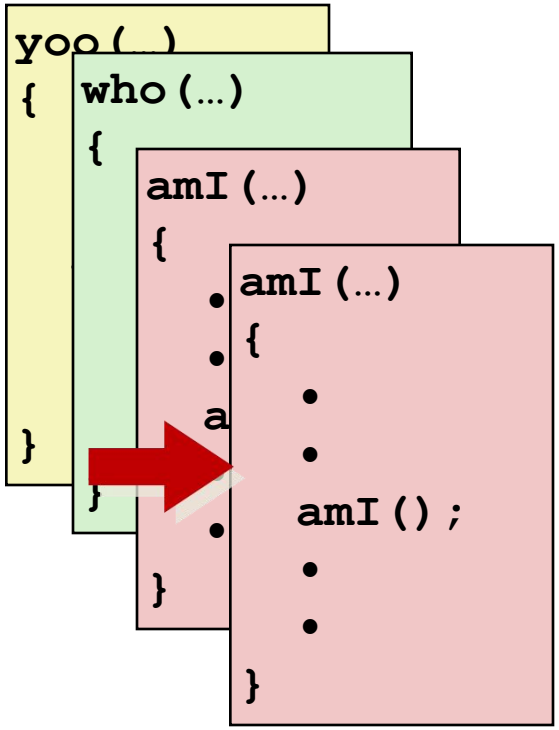
Example

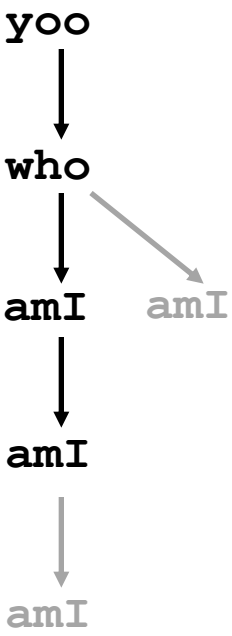

yoo (...)
{
who (...)
{
amI (...)
{
amI (...)
{
amI (...)
{
amI();
}

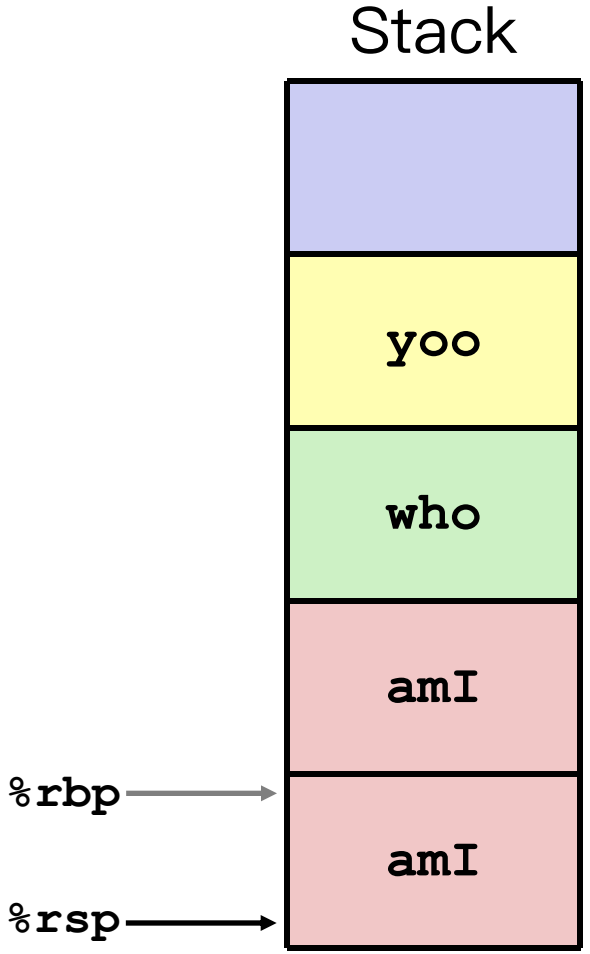

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

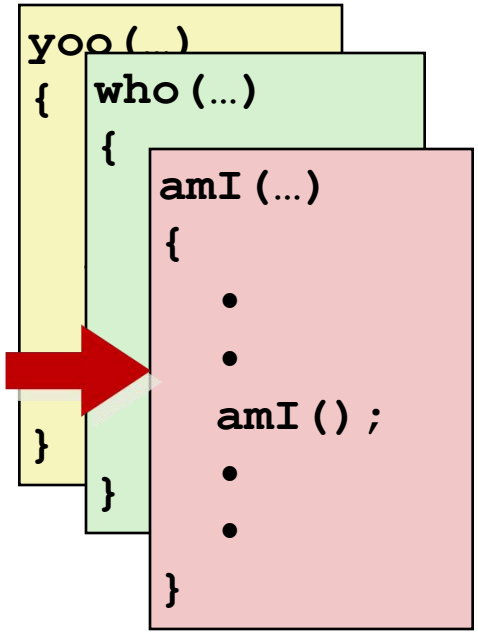
Example

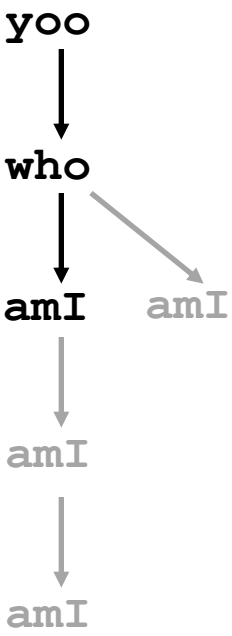

yoo(...)
{
who(...)
{
amI(...)
{
amI(...)
{
amI();
}

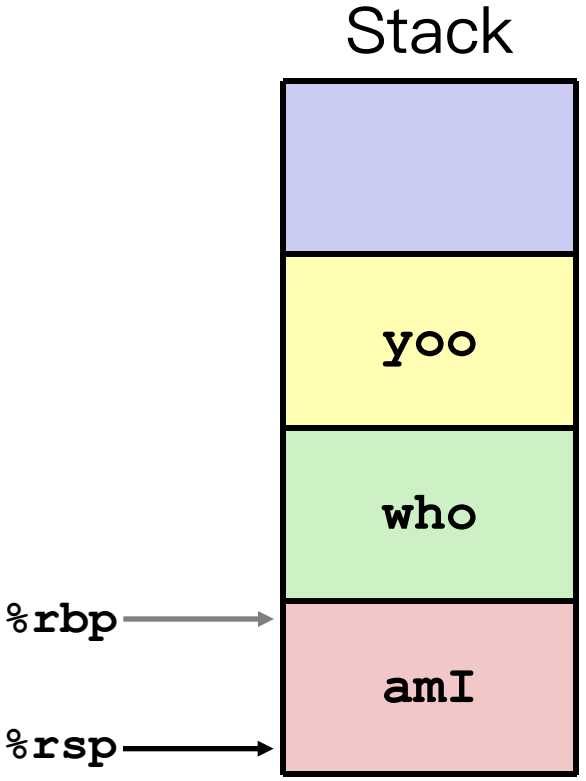

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

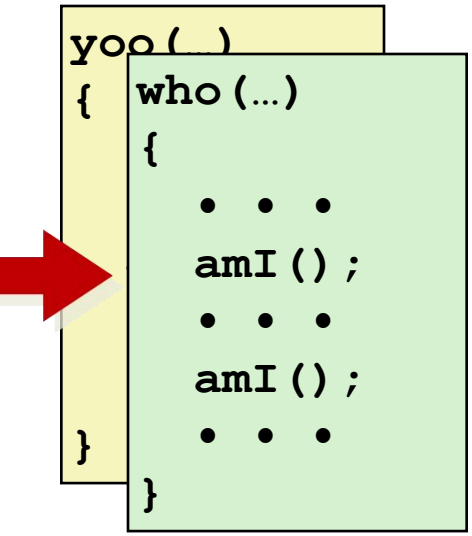
Example

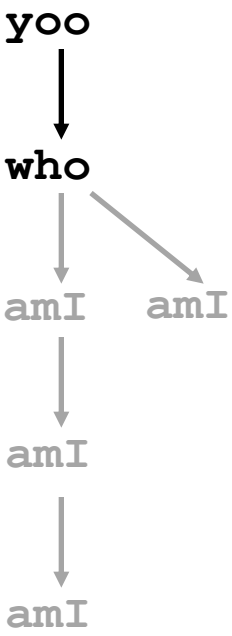

yoo(...)
{
who(...)
{
amI(...)
{
•
•
amI();
•
•
}
}
}

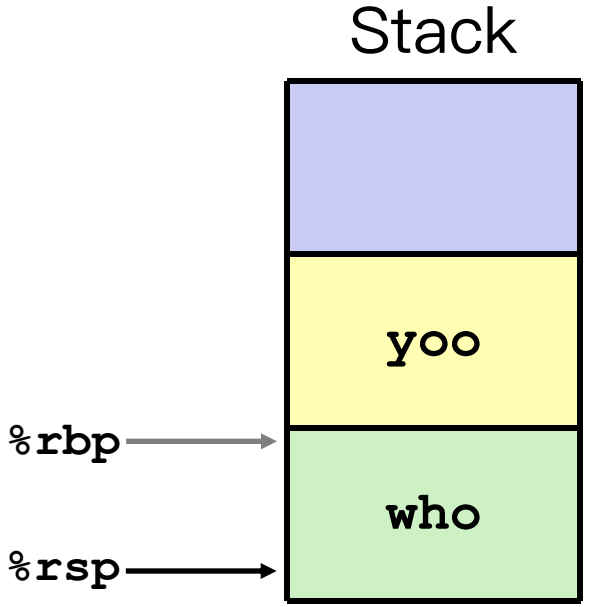

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

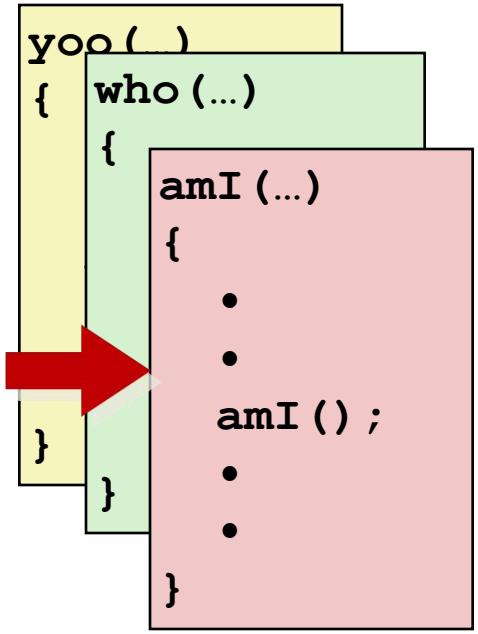
Example

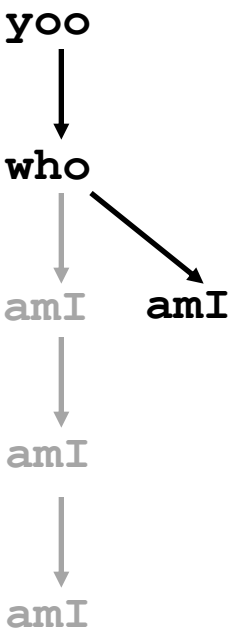


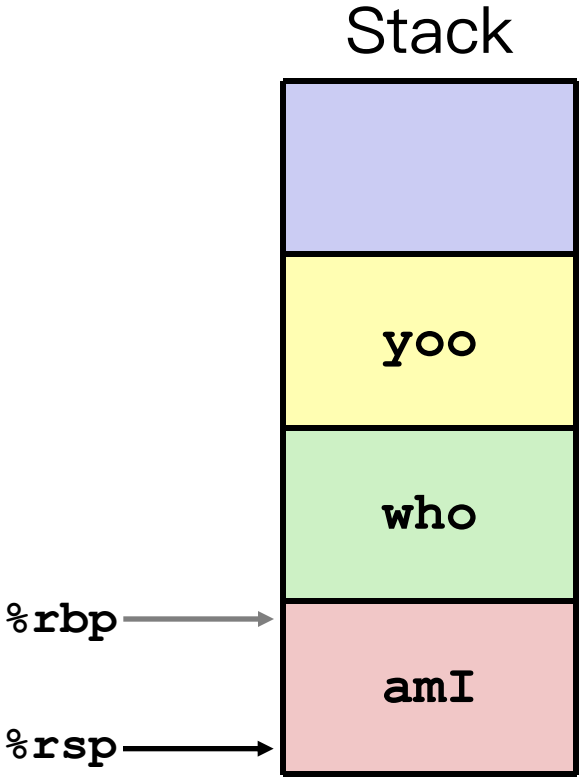




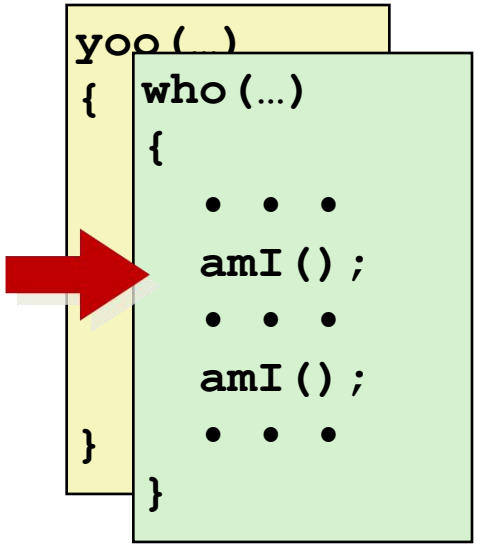
Example

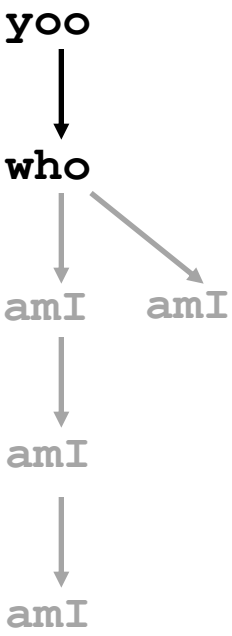

yoo(...)
{
who(...)
{
amI(...)
{
•
•
amI();
•
•
}
}
}

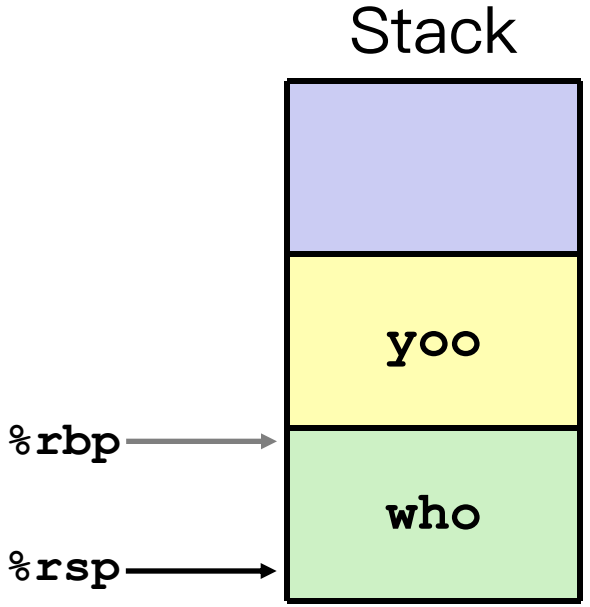

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

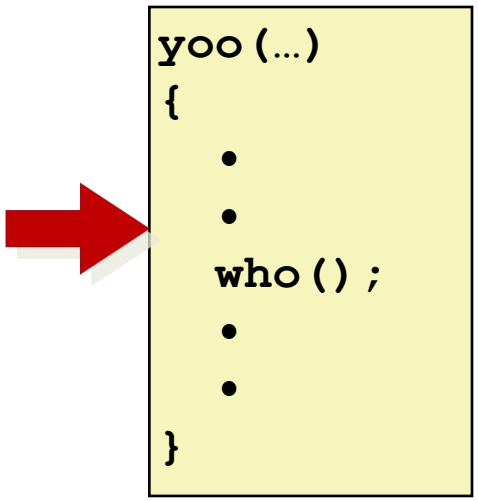
Example

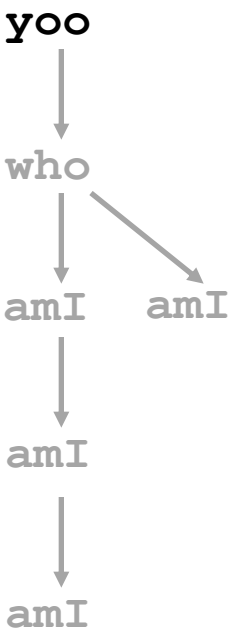


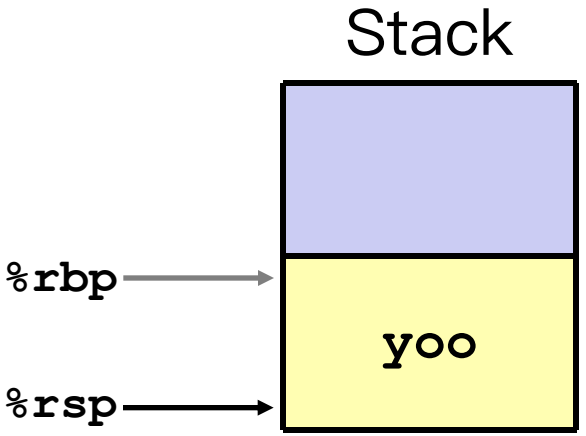




Example

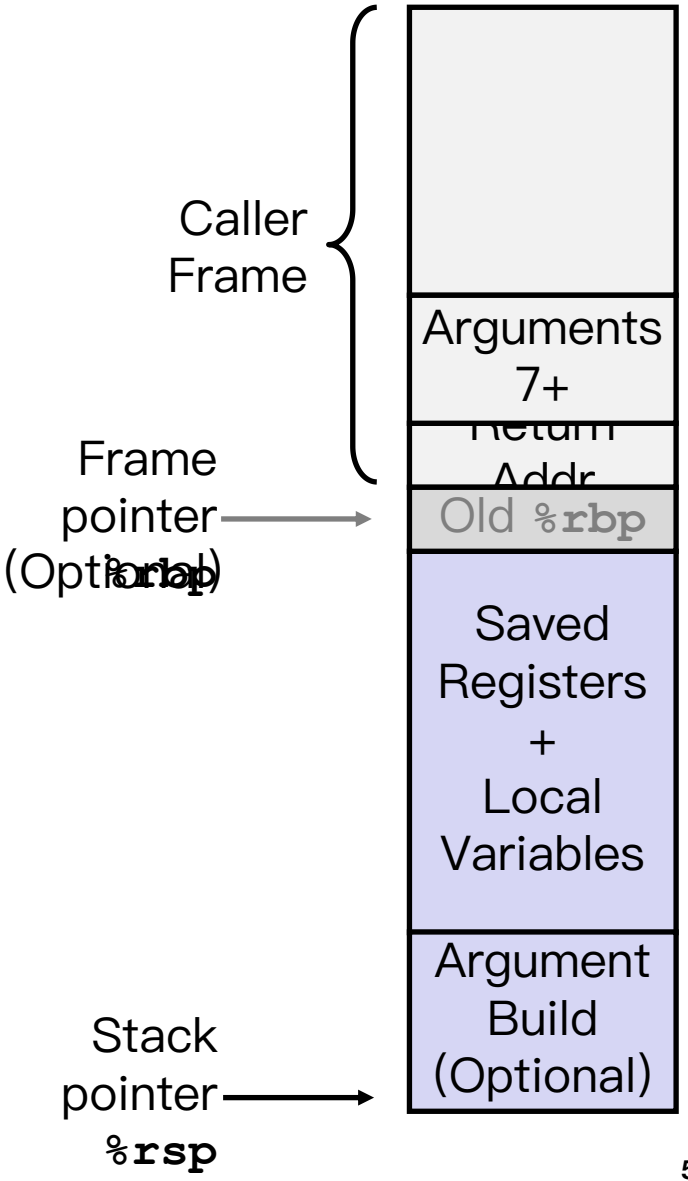






x86-64/Linux 的栈帧

- 当前栈帧（自栈顶向栈底）
  - “参数构造”——其此后将要调用的函数的参数（可选）
  - 局部变量（如果不能由寄存器承担）
  - 保存的寄存器上下文
  - 旧的帧指针（可选）
- 主调过程的栈帧
  - 返回地址
    - Pushed by `call` instruction
  - 此次调用的参数（不含前 6 个）



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
```

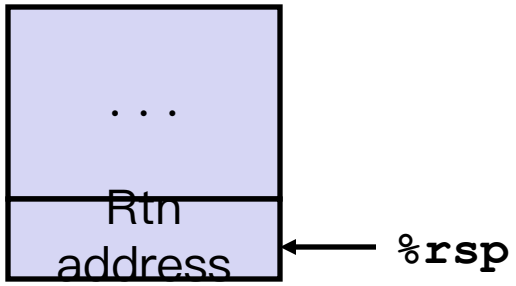
寄存器	取值
`%rdi`	Argument `p`
`%rsi`	Argument `val`, `y`
`%rax`	`x`, Return value

实例：调用 incr #1

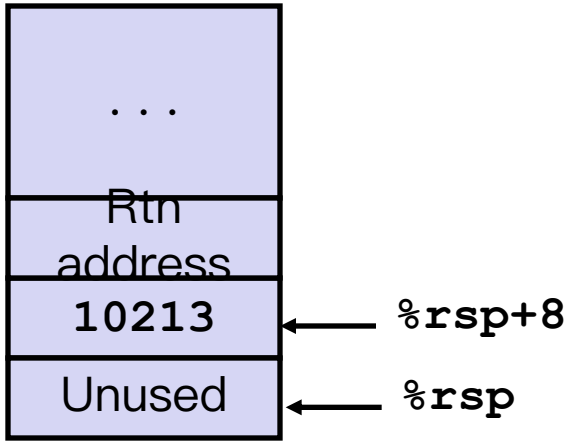
```
long call_incr()
{
    long v1 = 10213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $10213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



Resulting Stack Structure

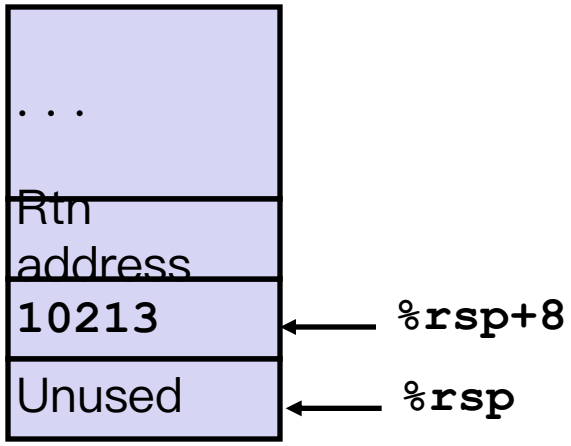


实例：调用 incr #2

```
long call_incr() {
    long v1 = 10213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $10213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



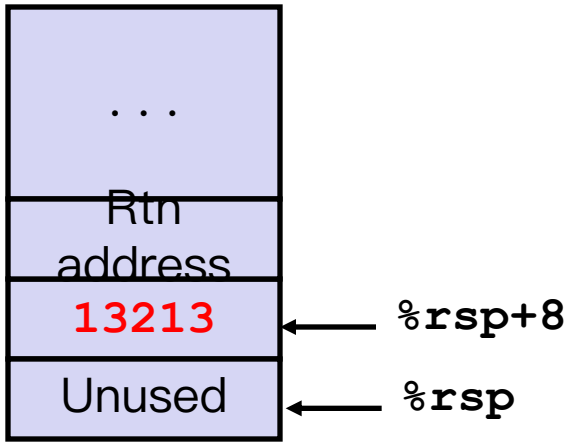
寄存器	取值
%rdi	&v1
%rsi	3000

实例：调用 incr #3

```
long call_incr() {
    long v1 = 10213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $10213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



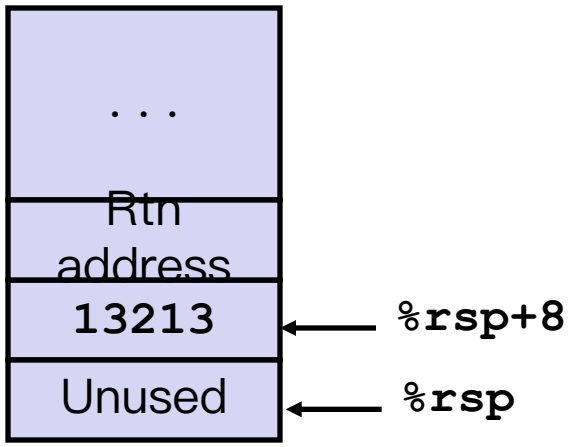
寄存器	取值
%rdi	&v1
%rsi	3000

实例：调用 incr #4

```
long call_incr() {
    long v1 = 10213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

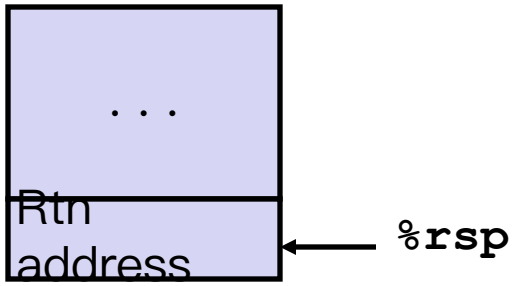
```
call_incr:
  subq    $16, %rsp
  movq    $10213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



寄存器	取值
%rax	Return value

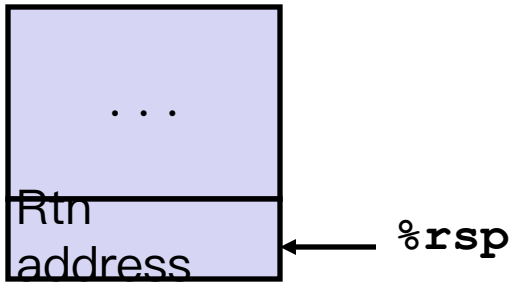
Updated Stack Structure



实例：调用 incr #5

```
long call_incr() {
    long v1 = 10213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

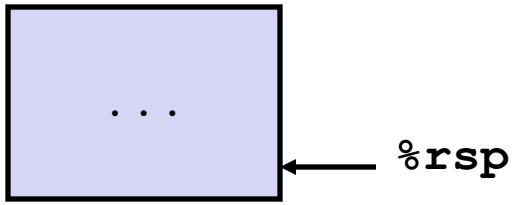
Updated Stack Structure




```
call_incr:
  subq    $16, %rsp
  movq    $10213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

寄存器	取值
%rax	Return value

Final Stack Structure



寄存器保存约定

- 当 yoo 调用 who 时：
  - yoo 为主调过程caller
  - who 为被调过程callee
- 寄存器可否用于临时性存储?

```
yoo:
    • • •
    movq $10213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $13213, %rdx
    • • •
    ret
```

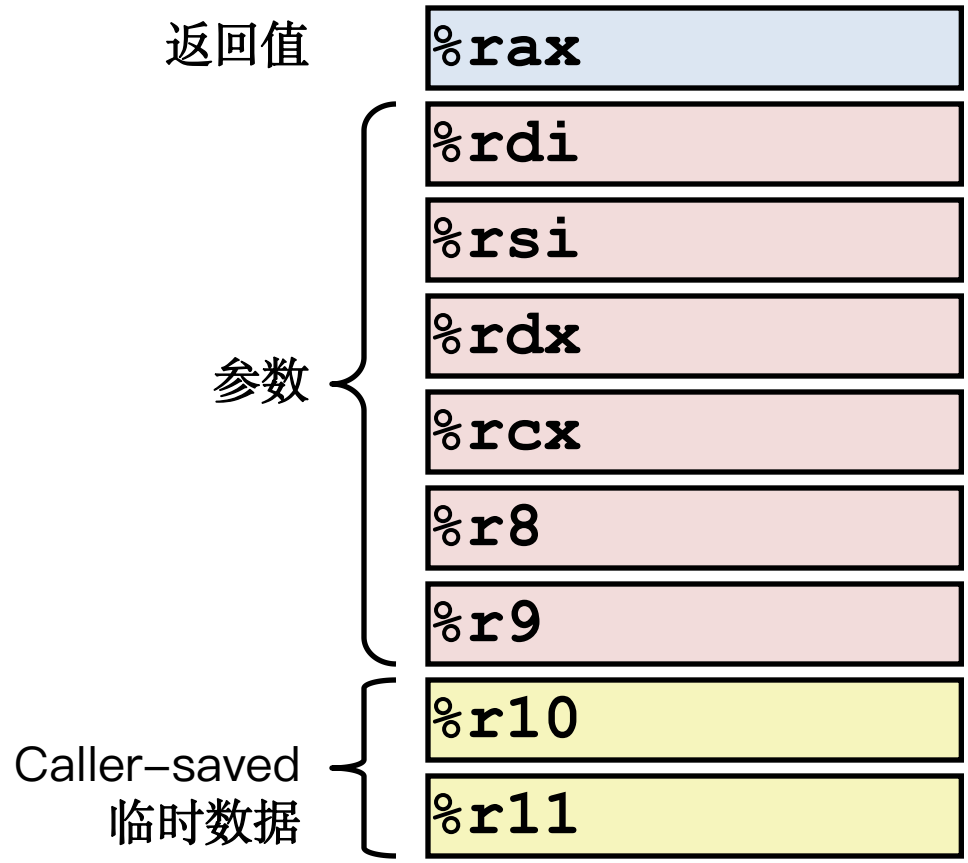
- 如上所示，寄存器 %rdx 的内容被 who 改写
- 解决这一潜在问题 ➔ 有些工作要做（课本勘误：参考文献[77]链接失效）
  - Need some coordination——ABI 的作用（gABI 和 x86-64-psABI 文档已共享）

寄存器保存约定

- 当 yoo 调用 who 时：
  - yoo 为主调过程caller
  - who 为被调过程callee
- 寄存器可否用于临时性存储?
- 约定
  - "主调方保存Caller Saved"
    - 主调方在调用之前将临时数值保存在其栈帧中
  - "被调方保存Callee Saved"
    - 被调方在使用某（些）数值之前先将其保存在自己的栈帧中
    - 被调方在返回主调方之前要将callee saved数值恢复

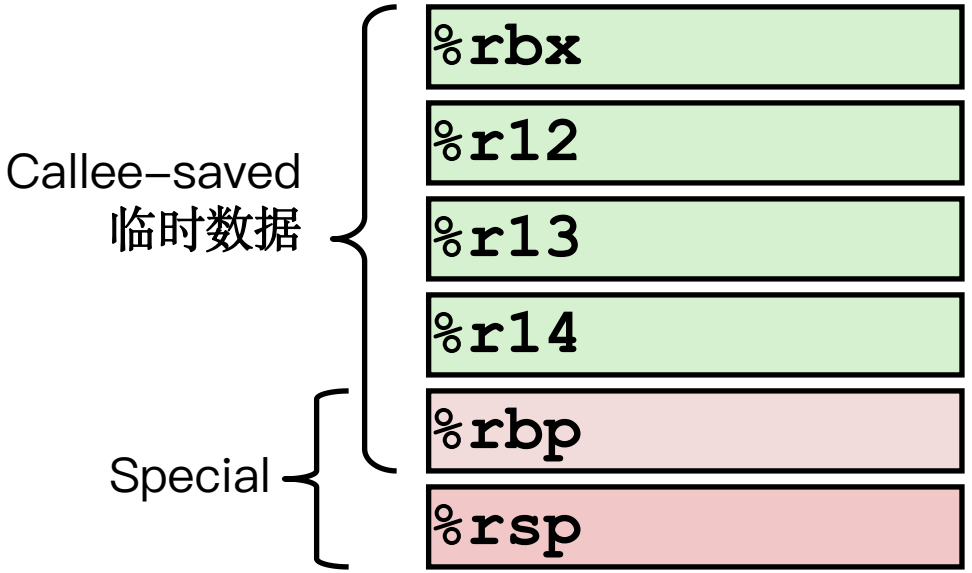
x86-64/Linux 的寄存器运用 #1

- %rax
  - 返回值
  - 也是 caller-saved
  - 可被过程修改
- %rdi, ..., %r9
  - 参数
  - 也是 caller-saved
  - 可被过程修改
- %r10, %r11
  - Caller-saved
  - 可被过程修改



x86-64/Linux 的寄存器运用 #2

- `%rbx`、`%r12`、`%r13`、`%r14`
  - Callee-saved
  - 由被调方负责保存并恢复
- `%rbp`
  - Callee-saved
  - 由被调方负责保存并恢复
  - 可被用于帧指针
  - 可以混用mix & match
- `%rsp`
  - Callee saved 的特殊形式
  - 从被调过程返回时，要求恢复原样

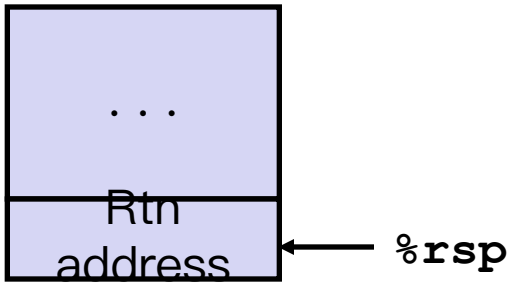


Callee-Saved 示例 #1

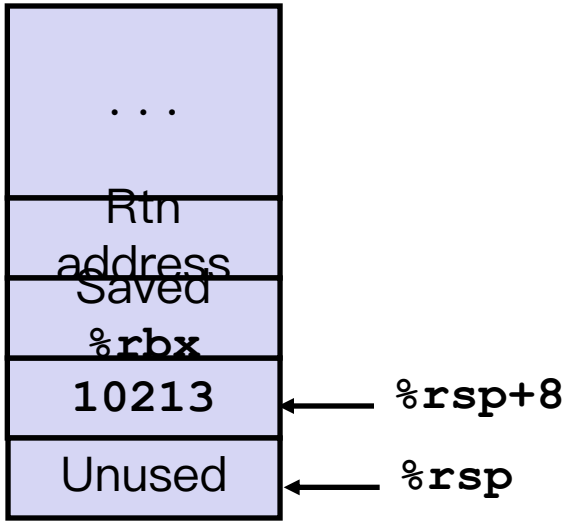
```
long call_incr2(long x)
{
    long v1 = 10213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $10213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

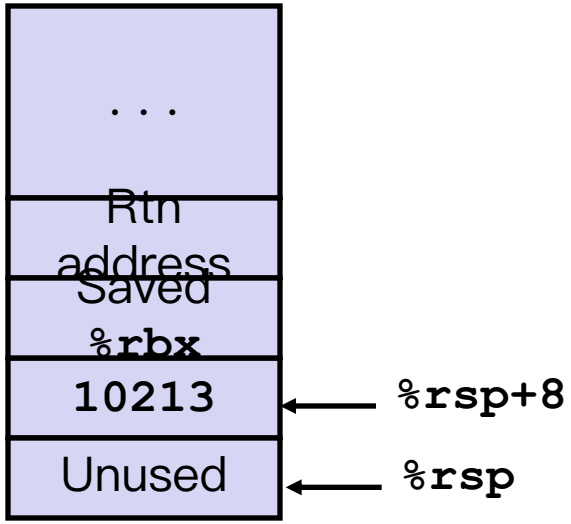


Callee-Saved 示例 #2

```
long call_incr2(long x)
{
    long v1 = 10213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $10213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Resulting Stack Structure



Pre-return Stack Structure