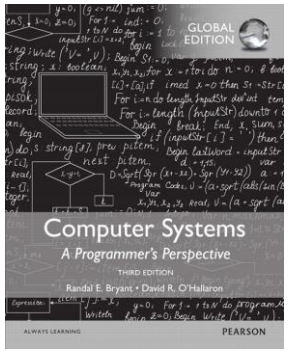


X86-64
Chamadas de funções e convenções

Bib: ComputerSystems: A Programmer's Perspective
x86-64 Machine-Level Programming (adenda ao cap. 3 do livro anterior)

Programação em Sistemas Computacionais

João Pedro Patriarca (jpatri@cc.isel.ipl.pt, joao.patriarca@isel.pt), Gabinete F.0.23 do edifício F

ISEL, ADEETC, LEIC

Agenda

- Chamada e retorno de funções
- Convenções
  - Passagem de parâmetros
  - Variáveis locais
  - Retorno
- Estrutura de dados *Stack* e *StackFrame*

Agenda

- Chamada e retorno de funções
- Convenções
  - Passagem de parâmetros
  - Variáveis locais
  - Retorno
- Estrutura de dados *Stack* e *StackFrame*

Implementação de chamadas no x86-64

- Os primeiros 6 argumentos são passados à função por via de registos, e os restantes são passados via *stack*
- A instrução `callq` empilha o endereço de retorno (valor atual do registo RIP) no *stack*
- Raramente as funções necessitam definir uma *Stack Frame*. Quando o fazem é porque não conseguem mapear todo o estado local em registos ou, chamam funções com mais de 6 argumentos
- As funções podem usar, no limite, até 128 bytes abaixo do valor atual do *stack pointer* (*rez zone*). Esta zona é válida até ser chamada outra função
- O *stack pointer* permanece com uma posição fixa durante a execução da função, ou seja, o espaço é alocado no preâmbulo da função
- Os registos definidos como *calle-saved*, quando usados, devem ser salvos e repostos no início e antes do retorno da função
- Antes da chamada de uma função o registo *stack pointer* deve apontar para um endereço múltiplo de 16 (relevante no âmbito da extensão SSE que introduz registos de 128 bits)

Instruções para chamar e retornar de uma função

Instrução		Efeito	Exemplo
`call`	*`target`* label reg mem	 push RIP; RIP += offset8(16)(32) push RIP; RIP = reg push RIP; RIP = [mem]	 `call  strcmp` `call  *%rax` `call  *table(%rsi)`
`ret`	*`[count]`*	 pop RIP pop RIP; RSP = RSP + count	 `ret` `ret  $4`

Agenda

- Chamada e retorno de funções
- Convenções
  - Passagem de parâmetros
  - Variáveis locais
  - Retorno
- Estrutura de dados *Stack* e *StackFrame*

Convenções

Passagem de parâmetros

- Os primeiros 6 parâmetros são passados em registos
- Os restantes parâmetros são passados através do *stack*
- Os parâmetros passados através do *stack* são empilhados da direita para a esquerda

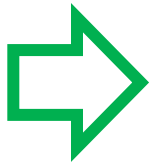
Registos convencionados para passagem de parâmetros

Número do parâmetro	Dimensão dos parâmetros (em bits)			
	64	32	16	8
1	%rdi	%edi	%di	%dil
2	%rsi	%esi	%si	%sil
3	%rdx	%edx	%dx	%dl
4	%rcx	%ecx	%cx	%cl
5	%r8	%r8d	%r8w	%r8b
6	%r9	%r9d	%r9w	%r9b

- Parâmetros com dimensão inferior a 64 bits podem ser acedidos acedendo à componente do registo adequada à dimensão do parâmetro
- O mesmo registo não pode ser usado para representar dois parâmetros ainda que o registo de 64 bits suportasse a dimensão dos dois parâmetros

Exemplo com 6 parâmetros

```
void proc(long a1, long *a1p,
          int a2, int *a2p,
          short a3, short *a3p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
}
```



```
proc:
    add     %rdi, (%rsi)
    add     %edx, (%rcx)
    add     %r8w, (%r9)
    retq
```

1º Param: a1 = rdi

2º Param: a1p = rsi

3º Param: a2 = rdx

4º Param: a2p = rcx

5º Param: a3 = r8

6º Param: a3p = r9

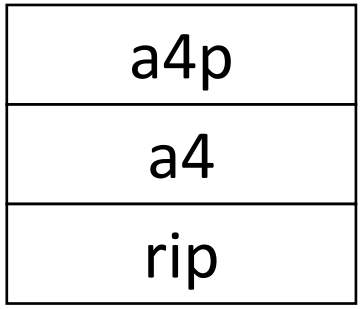
Exemplo com 8 parâmetros

```
void proc(long a1, long *a1p,
          int a2, int *a2p,
          short a3, short *a3p,
          char a4, char *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

```
proc:
    add     %rdi, (%rsi)
    add     %edx, (%rcx)
    add     %r8w, (%r9)
    mov     0x10(%rsp), %rax
    mov     0x8(%rsp), %edx
    add     %dl, (%rax)
    retq
```

1º Param: a1 = rdi
2º Param: a1p = rsi
3º Param: a2 = rdx
4º Param: a2p = rcx
5º Param: a3 = r8
6º Param: a3p = r9

8º Param:
7º Param:



rsp + 16
rsp + 8
rsp

Convenções

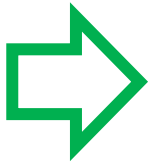
Variáveis locais

- Sempre que possível, usar registos para representar variáveis locais
- Quais os registos que devem ser usados?
  - Em funções folha, deve privilegiar-se a utilização de registos *caller saved* (não necessitam ser preservados)
  - Em funções não folha, deve privilegiar-se a utilização de registos *calle saved* sempre que o estado das variáveis prevaleça para além de chamadas a outras funções (o estado destes registos tem de ser preservado)
- Quando tem de se usar memória para representar variáveis locais?
  - Quando esgotados os registos
  - Sempre que aplicado o operador *&* (endereço de) a uma variável
  - Na definição de uma variável do tipo *array*
  - Na definição de uma variável do tipo estrutura

Registos	Convenção C
%rax	Return value
%rbx	Callee saved
%rcx	4th argument
%rdx	3rd argument
%rsi	2nd argument
%rdi	1st argument
%rbp	Callee saved
%rsp	Stack pointer
%r8	5th argument
%r9	6th argument
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

Exemplo com função folha

```
void f1_leaf_function(
    int a,
    int b,
    int *r)
{
    int v1, v2, v3;
    int v4, v5, v6;
    v1 = a * b;
    v2 = a + b;
    v3 = a / b;
    v4 = a % b;
    v5 = a << b;
    v6 = a >> b;
    *r = v1 + v2 + v3 +
         v4 + v5 + v6;
}
```



```
f1_leaf_function:
    mov    %esi, %ecx      # ecx = b
    mov    %rdx, %rsi      # rsi = r
    mov    %edi, %r8d      # r8d = a
    imul   %ecx, %r8d      # r8d = a*b
    lea    (%rdi,%rcx,1), %r10d # a+b
    mov    %edi, %eax      # eax = a
    cltd
    idiv   %ecx            # eax = a/b
    mov    %edi, %r9d      # edx = a%b
    shl    %cl,  %r9d      # r9d = a<<b
    sar    %cl,  %edi      # edi = a>>b
    add    %r10d,%r8d
    add    %eax, %r8d
    add    %edx, %r8d
    add    %r9d, %r8d
    add    %edi, %r8d
    mov    %r8d, (%rsi)
    retq
```

Exemplos com mapeamento em memória

- Os exemplos com mapeamento em memória são ilustrados depois de apresentado o tópico *Stack Frame*

Convenções

Retorno

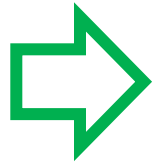
- O retorno é realizado no registo RAX sempre que a dimensão do tipo de retorno seja suportado pelo registo

Tipo de retorno	Registo convencionado
char	AL
short	AX
int	EAX
long	RAX
float	EAX
double	RAX

- Usado o par RDX:RAX quando o retorno é representado pelo intervalo 64 a 128 bits (o registo RDX contém a parte alta do retorno)
- Usado um parâmetro de saída sempre que a dimensão do tipo de retorno não seja suportada pelo par de registo RDX:RAX

Exemplo com retorno no registo RAX

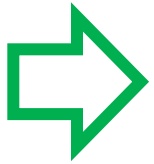
```
long return_long_rax(
    long a, int b)
{
    return a + b;
}
int return_int_rax(
    long a, int b)
{
    return (int)(a + b);
}
short return_short_rax(
    long a, int b)
{
    return (short)(a + b);
}
```



```
return_long_rax:
    movslq %esi,%rsi
    lea    (%rsi,%rdi,1),%rax
    retq

return_int_rax:
    lea    (%rsi,%rdi,1),%eax
    retq

return_short_rax:
    lea    (%rdi,%rsi,1),%eax
    retq
```

Exemplo com retorno no par de registos RDX:RAX

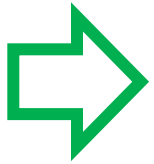
```c
typedef struct {
    long e1, e2;
} DataType128B;

DataType128B return_rdx_rax(
    long a,
    int b)
{
    DataType128B r = {a+b, a-b};
    return r;
}
```



```
return_rdx_rax:
    mov    %rdi, %rdx
    movslq %esi, %rsi
    lea    (%rsi,%rdi,1), %rax
    sub    %rsi, %rdx
    retq
```

Exemplo com retorno num parâmetro de saída

```
void return_out_parameter(
    long a,
    long b,
    long *r)
{
    int v = a + b;
    *r = v;
}
```



```
return_out_parameter:
    add     %rsi,%rdi
    mov     %rdi,(%rdx)
    retq
```

Agenda

- Chamada e retorno de funções
- Convenções
  - Passagem de parâmetros
  - Variáveis locais
  - Retorno
- Estrutura de dados *Stack* e *StackFrame*

Estrutura de dados *Stack*

- Estrutura com disciplina LIFO (*Last In First Out*)
- *Full descending*, ou seja, para empilhar, primeiro aloca, depois escreve e para desempilhar, primeiro lê, depois liberta o espaço alocado
- Registo RSP que aponta para o topo do *stack*
  - Registo usado para ler dados do *stack*
- Instrução *PUSH* para empilhar dados no *stack*
- Instrução *POP* para desempilhar dados do *stack*

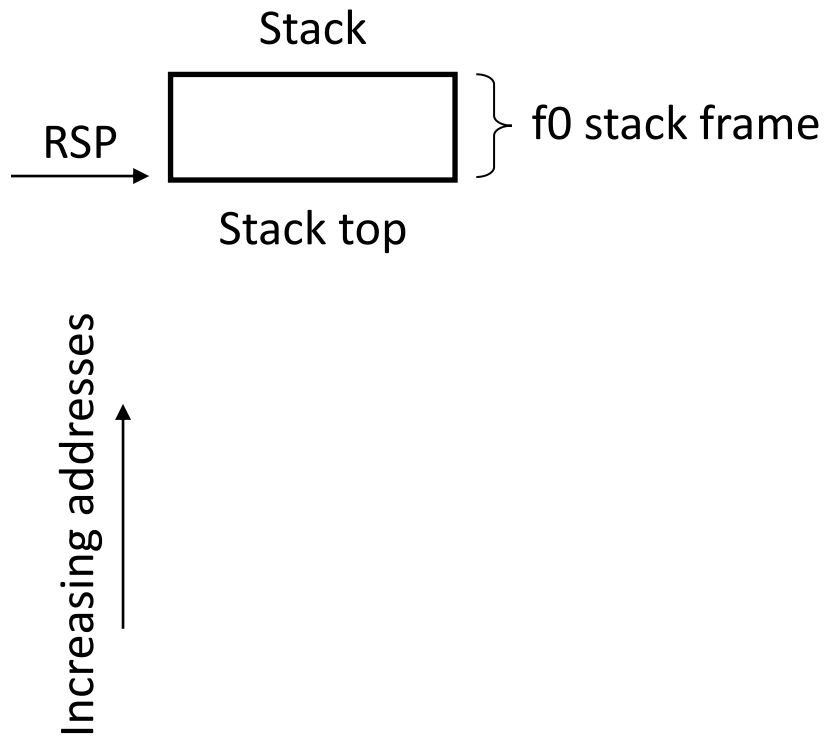
Instrução		Efeito	Exemplo
push	*S* reg64 mem64 imm64	*%rsp ⟵ %rsp - 8;* *M[%rsp] ⟵ S*	 `push %r11` `push (%rbx)` `push $0`
pop	*D* reg64 mem64	*D ⟵ M[%rsp];* *%rsp ⟵ %rsp + 8*	 `pop %r8` `pop var`

Stack Frame no x86-64

- Necessário definir um *stack frame* sempre que é necessário representar em memória dados locais de uma função
  - Quando esgotados os registos; Quando aplicado o operador *&* (endereço de) a uma variável; Na definição de uma variável do tipo *array*; Na definição de uma variável do tipo estrutura
  - Na preservação de registos *callee saved* antes de os modificar
  - Na preservação de registos *caller saved* antes de chamar uma função
  - Na passagem de argumentos a outras funções
- Definir um *stack frame* corresponde, no preâmbulo da função, subtrair o registo RSP pelo espaço necessário a alocar
  - O espaço alocado deve considerar o alinhamento para acesso às variáveis mapeadas em memória
- Durante a execução da função, o registo RSP permanece com um valor fixo
  - Altera quando é chamada outra função voltando ao mesmo valor após o retorno
- O acesso às variáveis e parâmetros é realizado indiretamente através do registo RSP
- O tempo de vida das variáveis corresponde ao tempo de execução da função, ou seja, o espaço alocado para o *stack frame* deve ser libertado no epílogo da função (antes de retornar) e corresponde a adicionar o registo RSP pelo espaço alocado

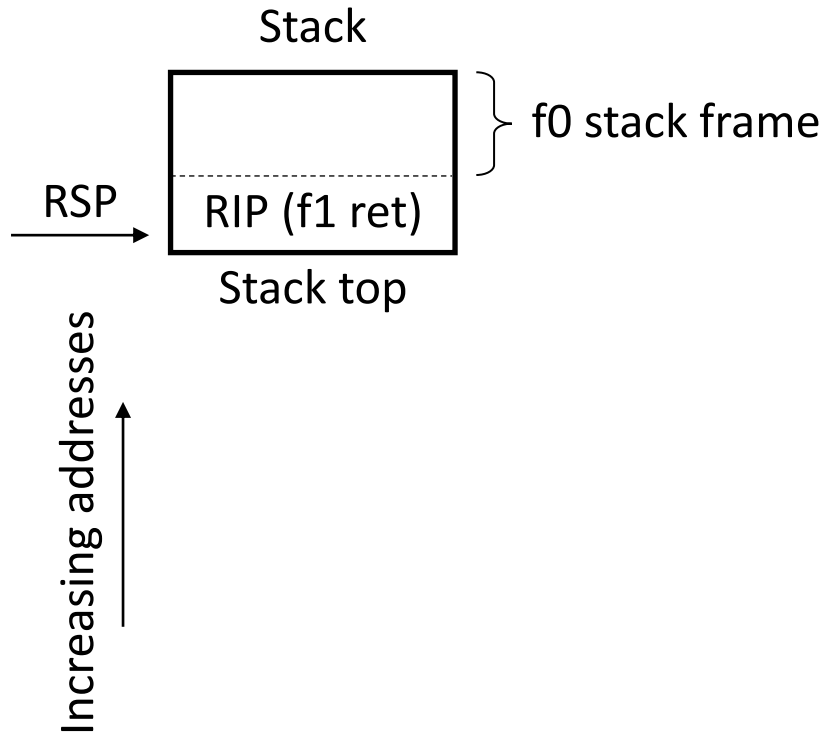
Exemplo de uma sequência de *stack frames* em memória
(1 de 6)

- A função é *f0* é chamada e aloca espaço para o seu *stack frame*



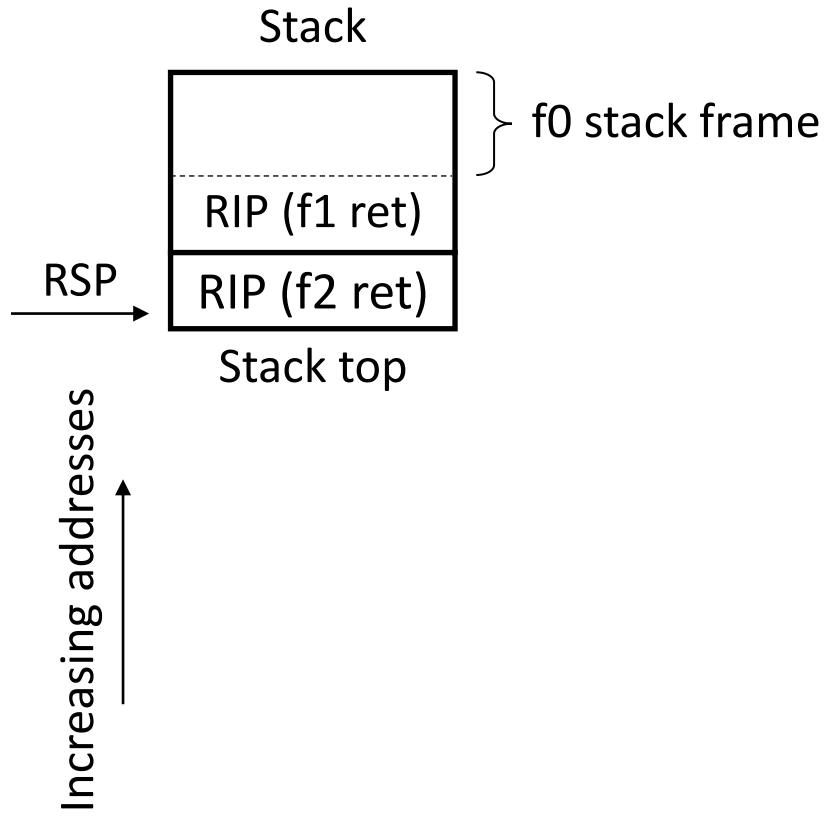
Exemplo de uma sequência de *stack frames* em memória (2 de 6)

- *f0* chama *f1* e *f1* não define *stack frame*



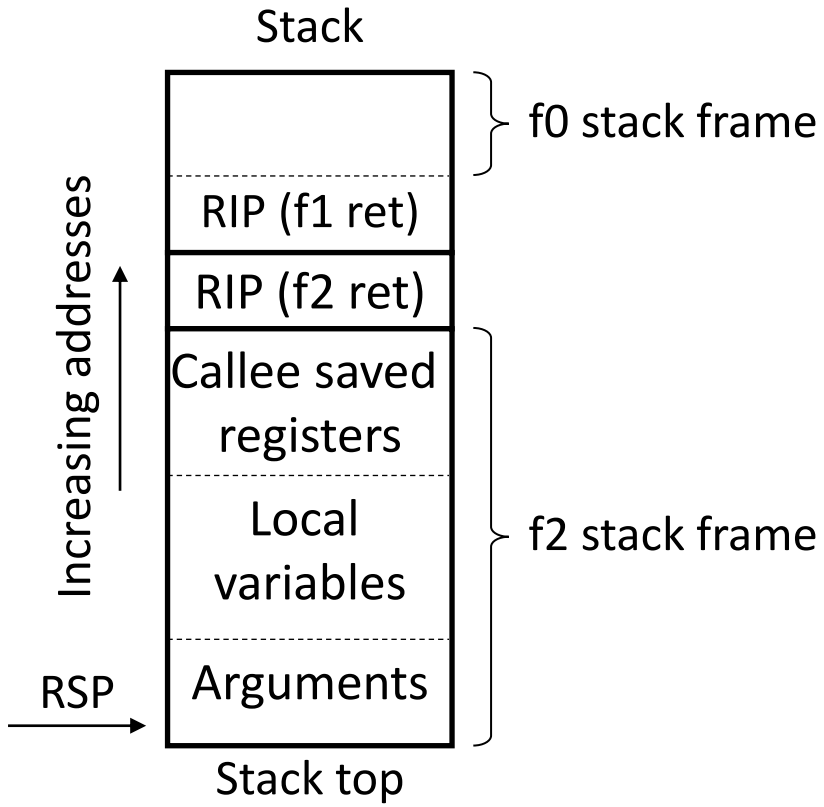
Exemplo de uma sequência de *stack frames* em memória
(3 de 6)

- *f0* chama *f1* chama *f2*



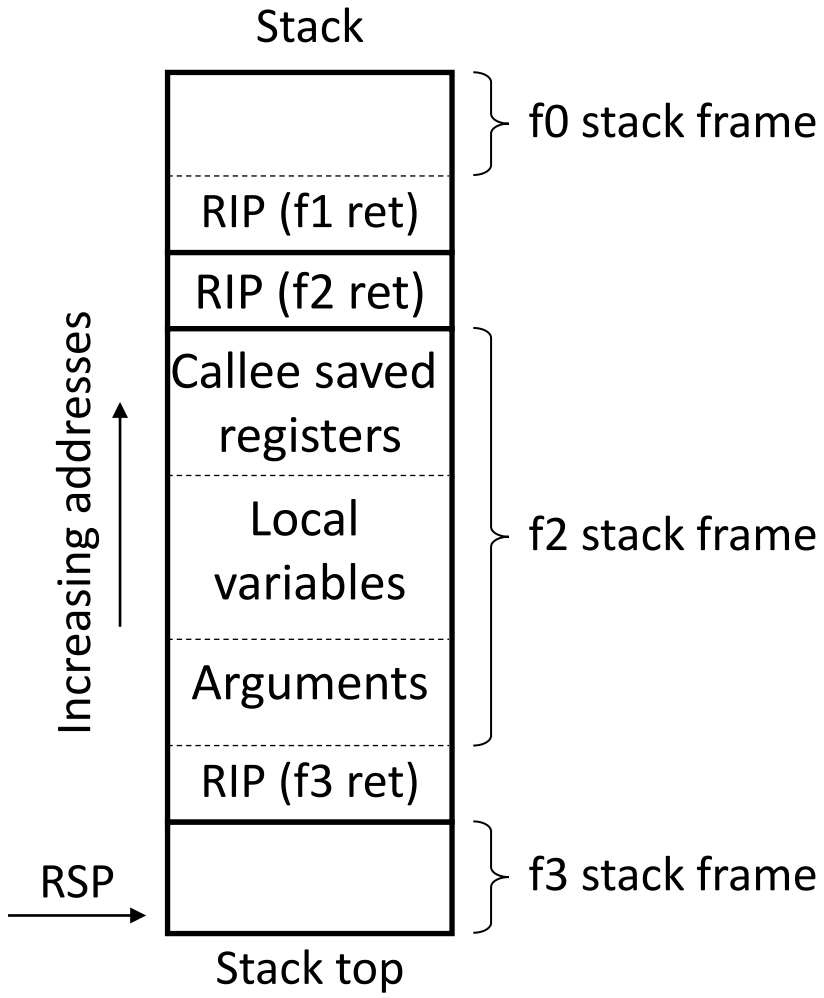
Exemplo de uma sequência de *stack frames* em memória (4 de 6)

- `f0` chama `f1` chama `f2` e define o seu *stack frame*
  1. Registos *callee saved*
  2. Variáveis locais
  3. Argumentos



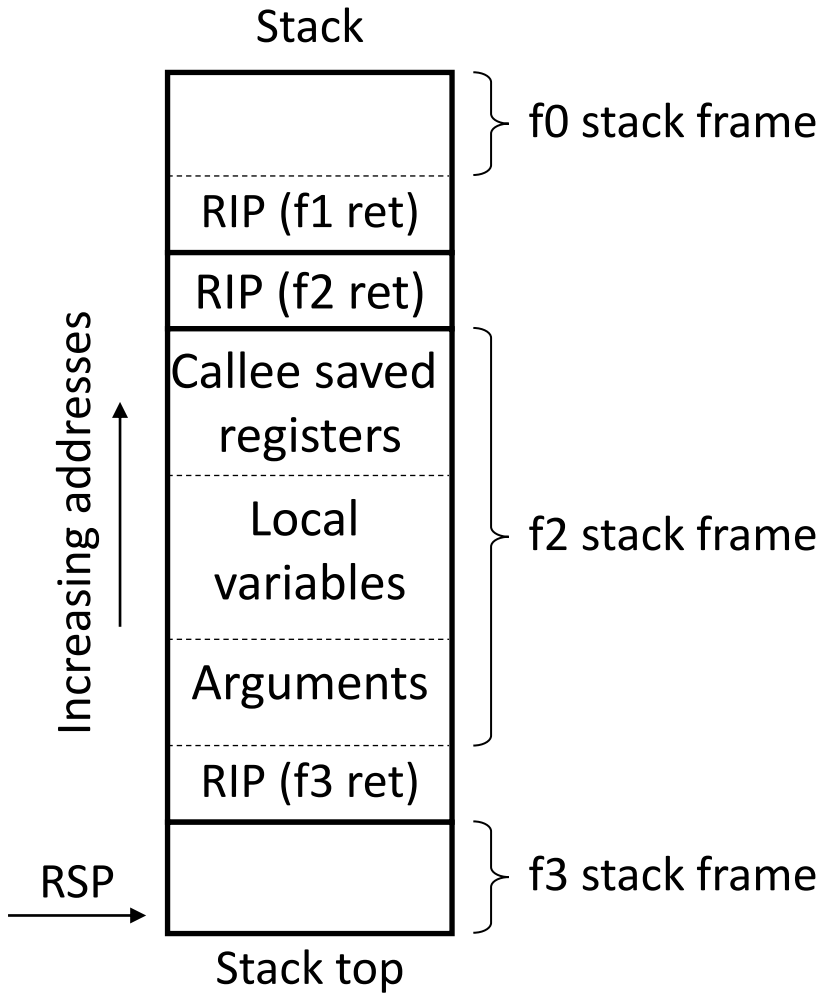
Exemplo de uma sequência de *stack frames* em memória
(5 de 6)

- *f0* chama *f1* chama *f2* chama *f3* e define o seu *stack frame*



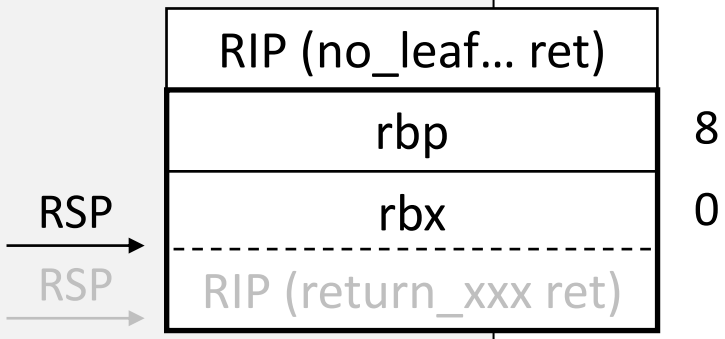
Exemplo de uma sequência de *stack frames* em memória (6 de 6)

- Sequência de chamadas
  - *f0* chama *f1* chama *f2* chama *f3*
  - Neste exemplo apenas f1 não define *stack frame*
- O endereço de retorno da função chamada pertence, por convenção, ao *stack frame* da função chamadora
- Organização dos dados dentro de um *stack frame*:
  1. Registos *callee saved*
  2. Variáveis locais
  3. Argumentos



Exemplo com função não folha

```
ulong no_leaf_function() {
    ulong  r1;
    uint   r2;
    ushort r3;
    r1 = return_long_rax(1, 2);
    r2 = return_int_rax(1, 2);
    r3 = return_short_rax(1, 2);

    return r1+r2+r3;
}
```



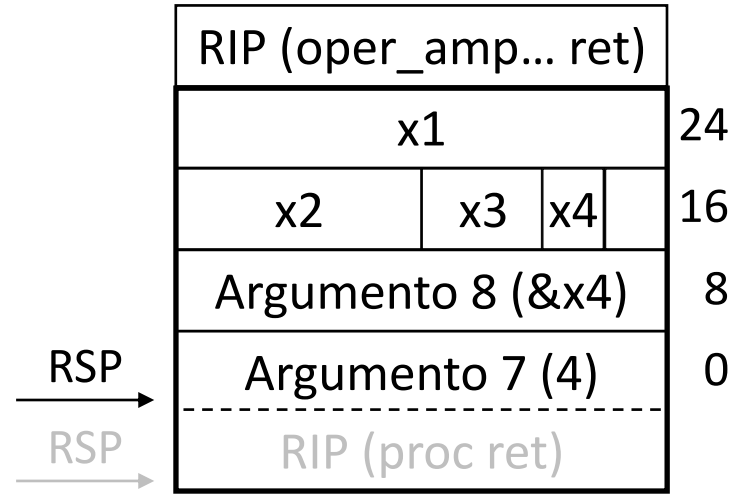

```
no_leaf_function:
    push   %rbp
    push   %rbx
    mov    $0x2, %esi
    mov    $0x1, %edi
    callq  <return_long_rax>
    mov    %rax, %rbp
    mov    $0x2, %esi
    mov    $0x1, %edi
    callq  <return_int_rax>
    mov    %eax, %ebx
    mov    $0x2, %esi
    mov    $0x1, %edi
    callq  <return_short_rax>
    movzwq %ax,  %rax
    add    %rbp, %rax
    add    %rbx, %rax
    pop    %rbx
    pop    %rbp
    retq
```

Exemplo com utilização do operador *&*

```
long oper_ampersand()
{
    long x1  = 1;
    int x2   = 2;
    short x3 = 3;
    char x4  = 4;
    proc(x1, &x1,
         x2, &x2,
         x3, &x3,
         x4, &x4
    );
    return x1 +
           x2 +
           x3 +
           x4;
}
```

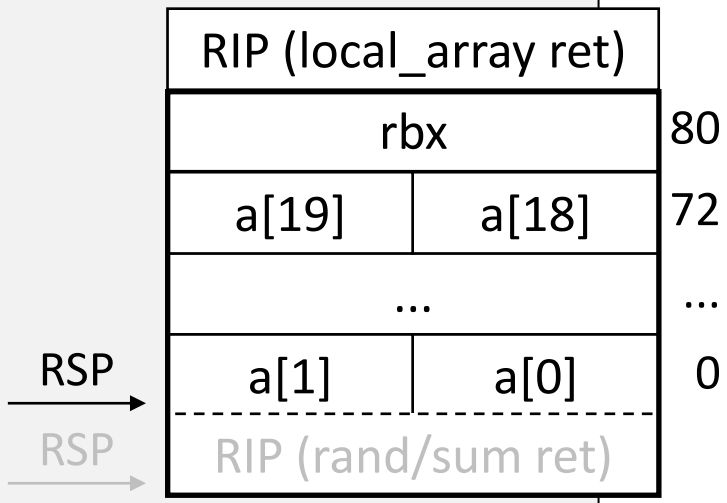
```
oper_ampersand:
    sub    32, %rsp
    movq   $1, 24(%rsp)
    movl   $2, 20(%rsp)
    movw   $3, 18(%rsp)
    movb   $4, 17(%rsp)
    lea    24(%rsp), %rsi
    lea    20(%rsp), %rcx
    lea    18(%rsp), %r9
    lea    17(%rsp), %rax
    mov    %rax, 8(%rsp)
    movq   $4, (%rsp)
    mov    $3, %r8d
    mov    $2, %edx
    mov    $1, %edi
    callq  1169 <proc>
```

```
 movslq 20(%rsp), %rdx
 add    24(%rsp), %rdx
 movswq 18(%rsp), %rax
 add    %rax, %rdx
 movsbq 17(%rsp), %rax
 add    %rdx, %rax
 add    $32, %rsp
 retq
```



Exemplo com *array*

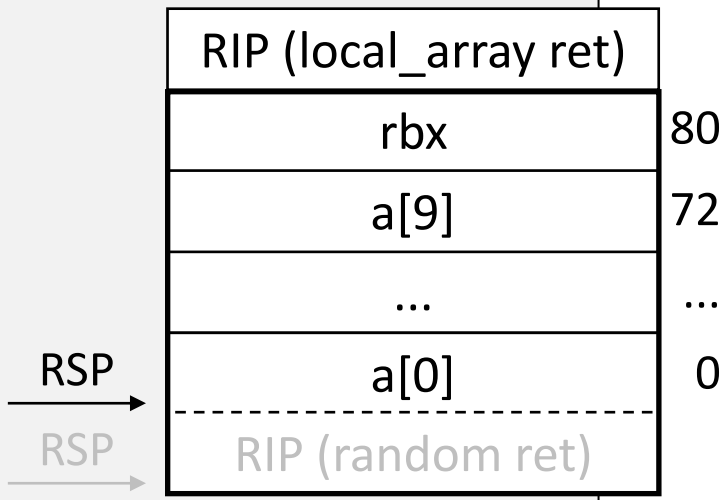
```c
int local_array() {
    int a[20];
    int sz = ARRAY_SIZE(a);
    for (int i = 0; i < sz; i++)
        a[i] = random();
    int r = sum_array(a, sz);
    return r;
}
```




```
local_array:
1475: endbr64
1479: push   %rbx
147a: sub    $80, %rsp
147e: mov    $0,  %ebx
1483: jmp    1496 <local_array+0x21>
1485: callq  1080 <random@plt>
148a: mov    %rax, %rdx
148d: movslq %ebx, %rax
1490: mov    %edx, (%rsp,%rax,4)
1493: add    $0x1, %ebx
1496: cmp    $19,  %ebx
1499: jle    1485  <local_array+0x10>
149b: mov    %rsp, %rdi
149e: mov    $20,%esi
14a3: callq  1455 <sum_array>
14a8: add    $80,  %rsp
14ac: pop    %rbx
14ad: retq
```

Exemplo com objeto estrutura

```c
typedef struct {
    ulong a[10];
} F7Struct;
ulong local_struct() {
    F7Struct st;
    for (int i = 0; i < 10; i++)
        st.a[i] = random();
    return st.a[0] - st.a[9];
}
```




```
local_struct>:
14dd: endbr64
14e1: push   %rbx
14e2: sub    $0x50, %rsp
14e6: mov    $0x0, %ebx
14eb: cmp    $0x9, %ebx
14ee: jg     1504 <local_struct+0x27>
14f0: callq  1080 <random@plt>
14f5: mov    %rax, %rdx
14f8: movslq %ebx, %rax
14fb: mov    %rdx,(%rsp,%rax,8)
14ff: add    $0x1, %ebx
1502: jmp    14eb <local_struct+0xe>
1504: mov    (%rsp), %rax
1508: sub    0x48(%rsp), %rax
150d: add    $0x50, %rsp
1511: pop    %rbx
1512: retq
```

Exercício com função não folha

- Traduza para *assembly* x64-86 a função `selection_sort` que ordena por ordem crescente um *array* de inteiros.

```
void swap(int * l, int * r) {
    int tmp = *l;
    *l = *r;
    *r = tmp;
}

void selection_sort(int a[], int size) {
    int min_idx;
    for (int i = 0; i < size-1; i++) {
        min_idx = i;
        for (int j = i+1; j < size; j++)
            if (a[j] < a[min_idx])
                min_idx = j;
        if (min_idx != i)
            swap(&a[i], &a[min_idx]);
    }
}
```