



Estrutura do tema ISA do IA-32

1. Desenvolvimento de programas no IA-32 em Linux
2. Acesso a operandos e operações
3. Suporte a estruturas de controlo
- 4. Suporte à invocação/regresso de funções**
5. Análise comparativa: IA-32 vs. x86-64 e RISC (MIPS e ARM)
6. Acesso e manipulação de dados estruturados



Estrutura de uma função (/ procedimento)

- **função *versus* procedimento** (ou ainda **rotina**, em Fortran)
 - o nome duma função é usado como se fosse uma variável
 - uma função devolve um valor, um procedimento não
- **parte visível ao programador em HLL**
 - o código do corpo da função
 - a passagem de parâmetros/argumentos para a função ...
... e o valor devolvido pela função
 - o alcance das variáveis: locais, externas ou globais
- **parte não visível em HLL** (gestão do contexto da função)
 - variáveis locais (propriedades)
 - variáveis externas e globais (localização e acesso)
 - parâms / argum's e valor a devolver pela função (propriedades)
 - gestão do contexto (controlo & dados)



Análise do contexto de uma função

– propriedades das variáveis locais:

- visíveis apenas durante a execução da função
- deve suportar aninhamento e recursividade
- localização ideal (escalares): em registo, se os houver...
- localização no código em IA-32: em registo, enquanto houver...

– variáveis externas e globais:

- externas: valor ou localização expressa na lista de argumentos
- globais: localização definida pelo *linker & loader* (IA-32: na memória)

– propriedades dos parâmetros / arg's (só de entrada em C):

- por valor (c^{te} ou valor da variável) ou por referência (localização da variável)
- designação independente (f. chamadora / f. chamada): →
- deve ...
- localização ideal: ...
- localização no código em IA-32: ...

– valor a devolver pela função:

- é ...
- localização: ...

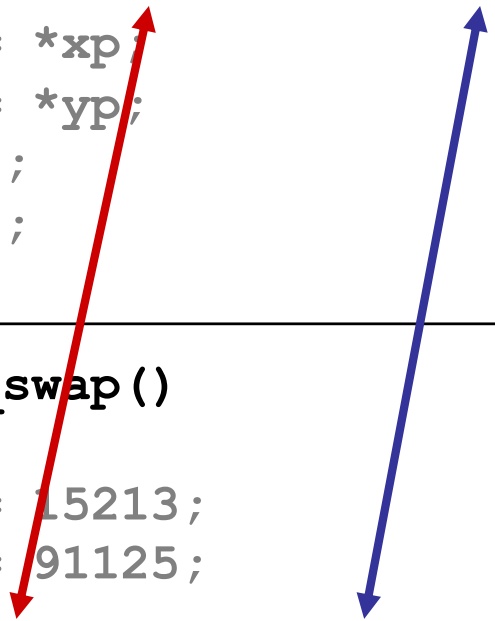
– gestão do contexto ...

Designação independente dos parâmetros



```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
void call_swap()
{
    int zip1 = 15213;
    int zip2 = 91125;
    (...)
    swap(&zip1, &zip2);
    (...)
}
```





Análise do contexto de uma função

- **propriedades das variáveis locais:**
 - visíveis apenas durante a execução da função
 - deve suportar aninhamento e recursividade
 - localização ideal (escalares): em registo, se os houver...
 - localização no código em IA-32: em registo, enquanto houver...
- **variáveis externas e globais:**
 - externas: valor ou localização expressa na lista de argumentos
 - globais: localização definida pelo *linker & loader* (IA-32: na memória)
- **propriedades dos parâmetros/arg's** (só de entrada em C):
 - por valor (c^{te} ou valor da variável) ou por referência (localização da variável)
 - designação independente (f. chamadora / f. chamada)
 - deve suportar aninhamento e recursividade
 - localização ideal: em registo, se os houver; mas...
 - localização no código em IA-32: na memória (na *stack*)
- **valor a devolver pela função:**
 - é uma quantidade escalar, do tipo inteiro, real ou apontador
 - localização: em registo (IA-32: `int` no registo `eax` e/ou `edx`)
- **gestão do contexto** (controlo & dados) ...



Análise do código de gestão de uma função

- **invocação e regresso**
 - instrução de salto, c/ salvaguarda IP (endereço de regresso)
 - em registo (RISC; aninhamento / recursividade ?)
 - em memória/na *stack* (IA-32; aninhamento / recursividade ?)
- **invocação e regresso**
 - instrução de salto para o endereço de regresso
- **salvaguarda & recuperação de registos** (na *stack*)
 - função chamadora ? (nenhum/ alguns/ todos ? RISC/IA-32 ?)
 - função chamada? (nenhum/ alguns/ todos ? RISC/IA-32 ?)
- **gestão do contexto ...**

Utilização de registos em funções: regras seguidas pelos compiladores para IA-32



Utilização dos registos (de inteiros)

–Três do tipo *caller-save*

%eax, %edx, %ecx

- *save/restore*: função chamadora

–Três do tipo *callee-save*

%ebx, %esi, %edi

- *save/restore*: função chamada

–Dois apontadores (para a *stack*)

%esp, %ebp

- topo da *stack*, base/referência na *stack*

Caller-Save

Callee-Save

Pointers

%eax

%edx

%ecx

%ebx

%esi

%edi

%esp

%ebp

Nota: valor a devolver pela função vai em **%eax**



Análise do código de gestão de uma função

- **invocação e regresso**
 - instrução de salto, c/ salvaguarda IP (endereço de regresso)
 - em registo (RISC; aninhamento / recursividade ?)
 - em memória/na *stack* (IA-32; aninhamento / recursividade ?)
- **invocação e regresso**
 - instrução de salto para o endereço de regresso
- **salvaguarda & recuperação de registos** (na *stack*)
 - função chamadora ? (nenhum/ alguns/ todos ? RISC/IA-32 ?)
 - função chamada? (nenhum/ alguns/ todos ? RISC/IA-32 ?)
- **gestão do contexto** (em *stack frame* ou *activation record*)
 - reserva/libertação de espaço para variáveis locais
 - atualização/recuperação do *frame pointer* (IA-32...)



Análise de exemplos

– revisão do exemplo swap

- análise das fases: arranque/inicialização, corpo, término
- análise dos contextos (IA-32)
- evolução dos contextos na *stack* (IA-32)

– evolução de um exemplo: Fibonacci

- análise ...

– aninhamento e recursividade

- evolução ...

Análise das fases em swap, no IA-32 (fig. já apresentada)



```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Código C

swap:

```
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx
```

Arranque

```
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax, (%edx)
    movl %ebx, (%ecx)
```

Corpo

```
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Término

Assembly

Análise dos contextos em swap, no IA-32



```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

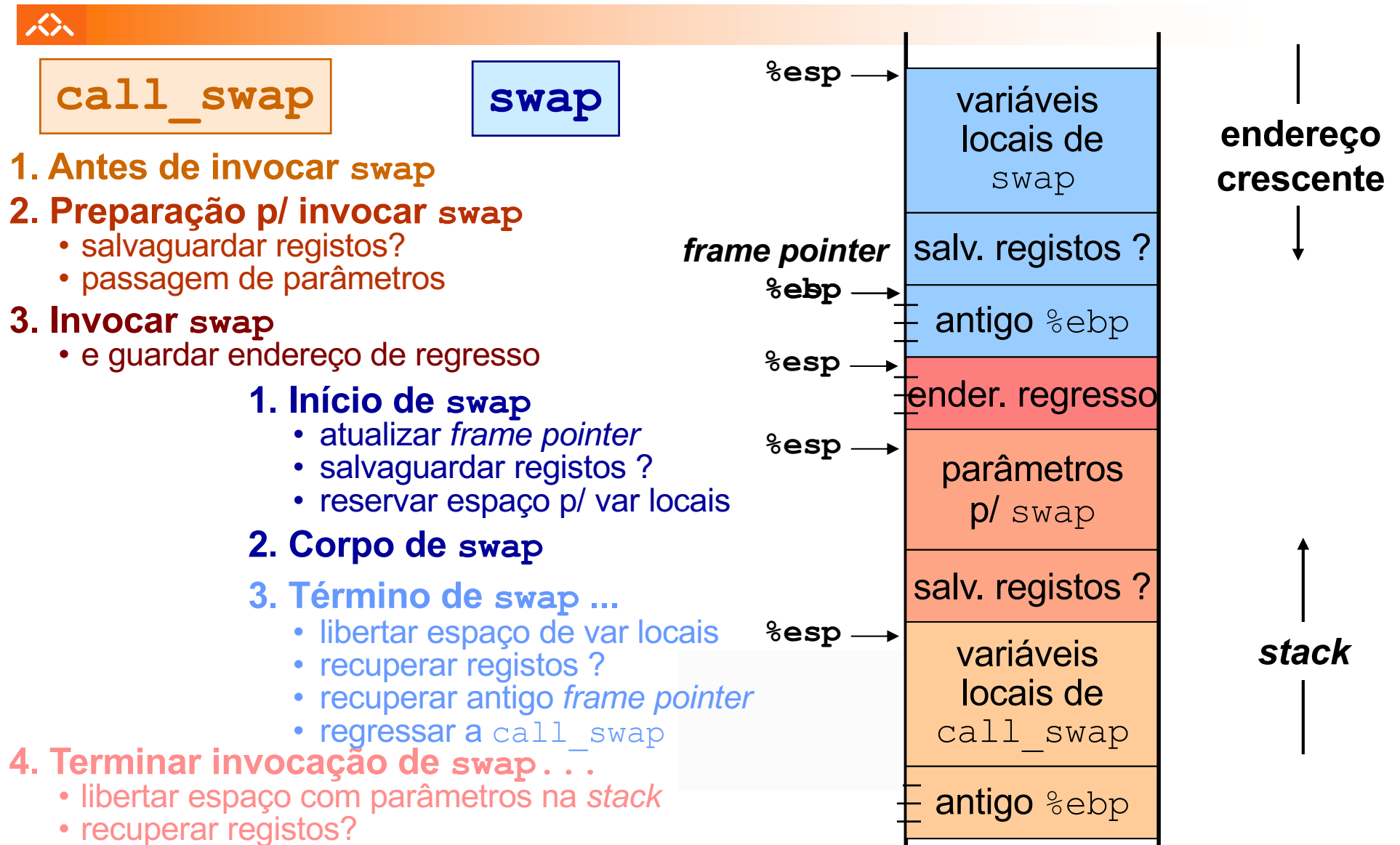
```
void call_swap()
{
    int zip1 = 15213;
    int zip2 = 91125;
    (...)
    swap(&zip1, &zip2);
    (...)
}
```

- em `call_swap`
- na invocação de `swap`
- na execução de `swap`
- no regresso a `call_swap`

Que contextos (IA-32)?

- passagem de parâmetros
 - via *stack*
- espaço para variáveis locais
 - na *stack*
- info de suporte à gestão (*stack*)
 - endereço de regresso
 - apontador para a *stack frame*
 - salvaguarda de registos

Construção do contexto na stack, no IA-32



Evolução da stack, no IA-32 (1)

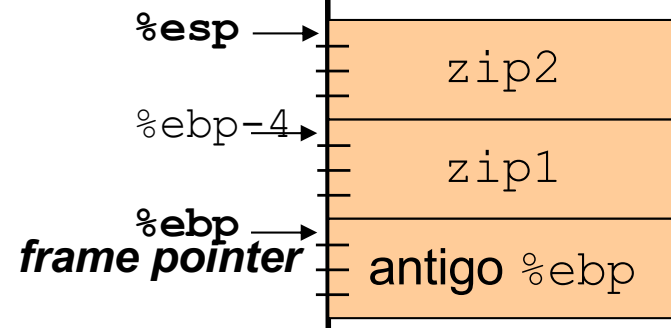


call_swap

1. Antes de invocar swap

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
void call_swap()
{
    int zip1 = 15213;
    int zip2 = 91125;
    (...)
    swap(&zip1, &zip2);
    (...)
}
```



endereço
crescente

stack

Evolução da stack, no IA-32 (2)



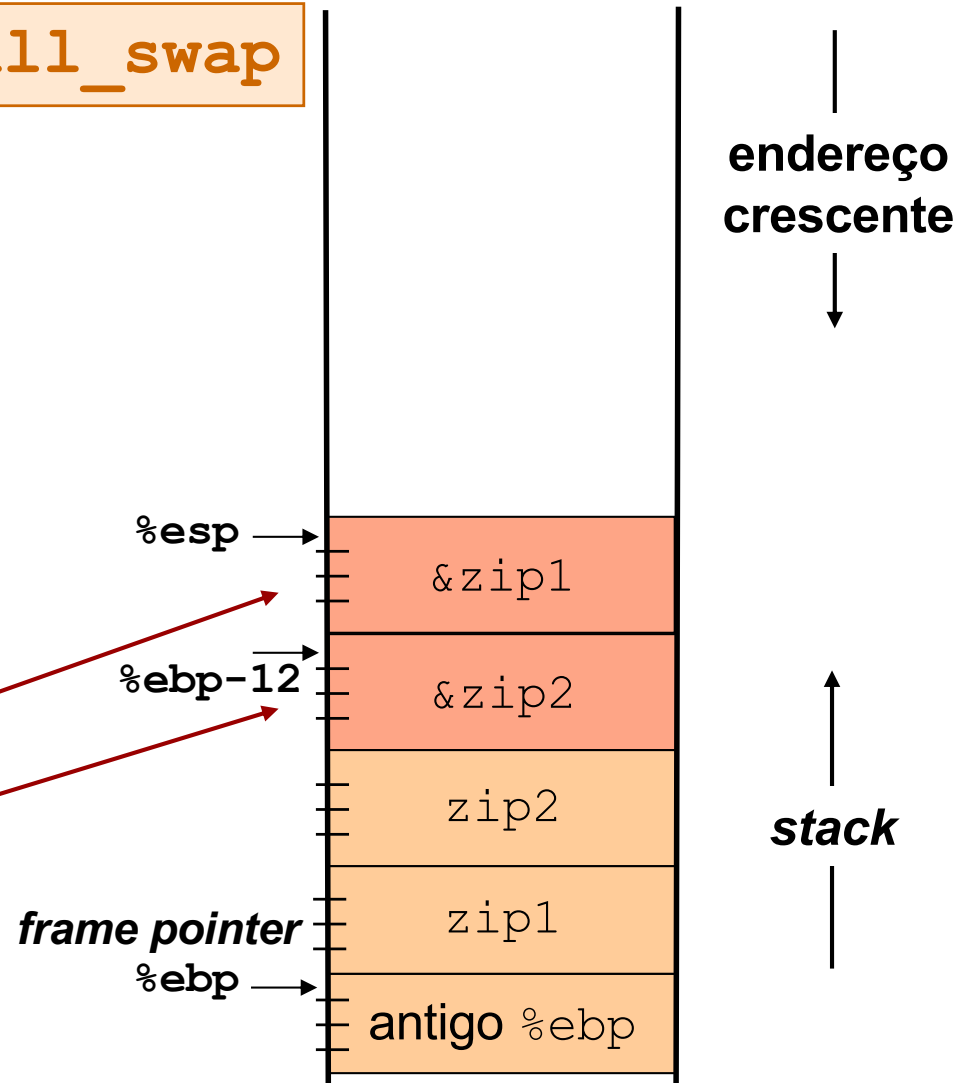
2. Preparação p/ invocar swap

- salvaguardar registos?...**não**...
- passagem de parâmetros

call_swap

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
void call_swap()
{
    int zip1 = 15213;
    int zip2 = 91125;
    (...)
    swap(&zip1, &zip2);
    (...)
}
```



Evolução da stack, no IA-32 (3)

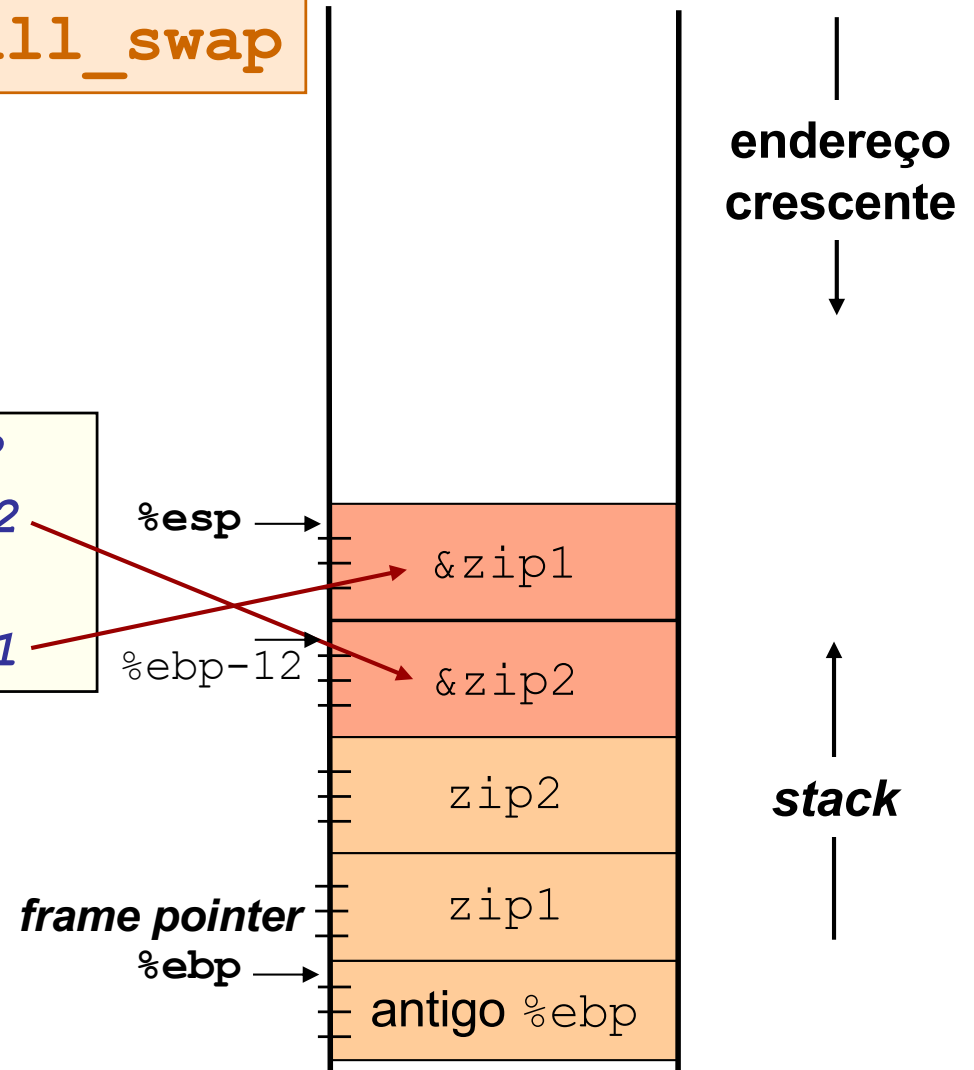


call_swap

2. Preparação p/ invocar swap

- salvar guardar registos?...**não**...
- passagem de parâmetros

```
leal  -8(%ebp), %eax  Calcula &zip2  
pushl %eax            Empilha &zip2  
leal  -4(%ebp), %eax  Calcula &zip1  
pushl %eax            Empilha &zip1
```



Evolução da stack, no IA-32 (4)



3. Invocar swap

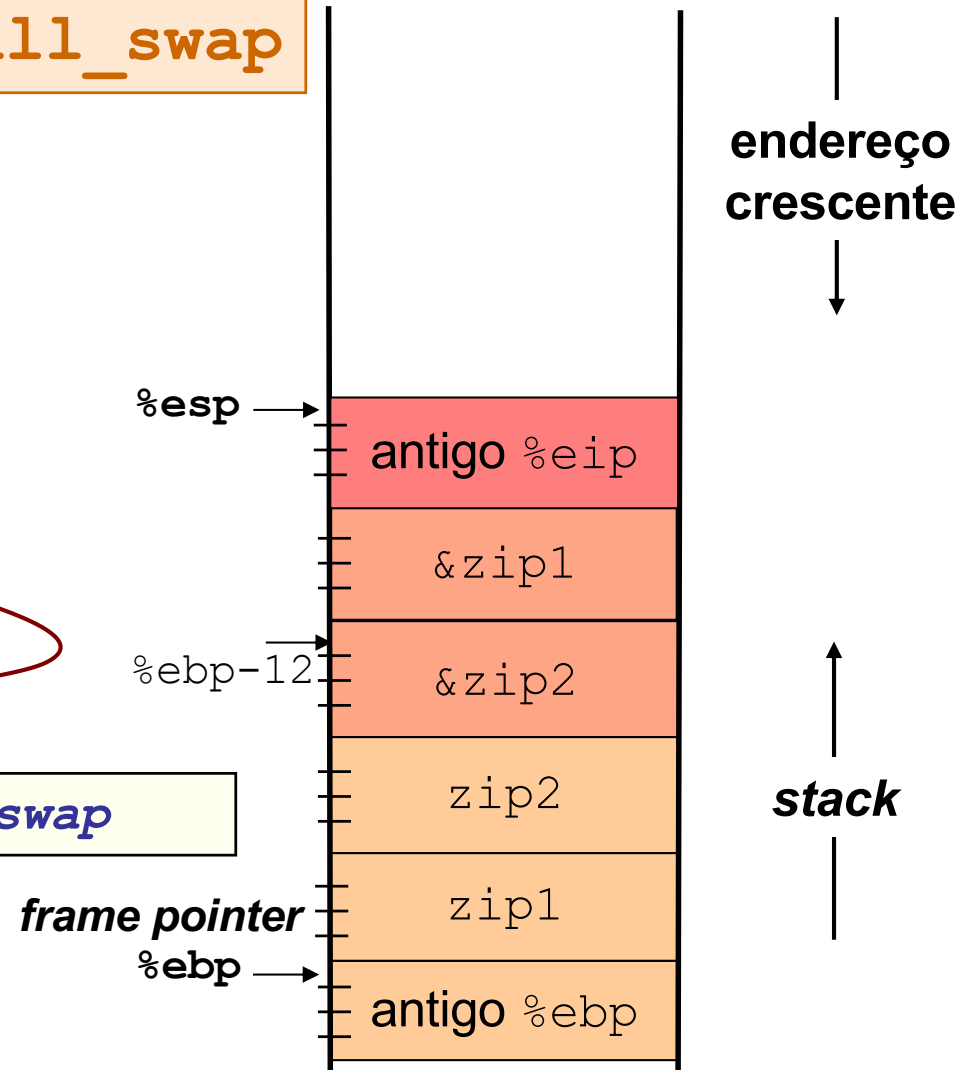
- e guardar endereço de regresso

```
void call_swap()  
{  
  int zip1 = 15213;  
  int zip2 = 91125;  
  (...)  
  swap(&zip1, &zip2);  
  (...)  
}
```

call swap

Invoca função swap

call_swap



Evolução da stack, no IA-32 (5)



1. Início de swap

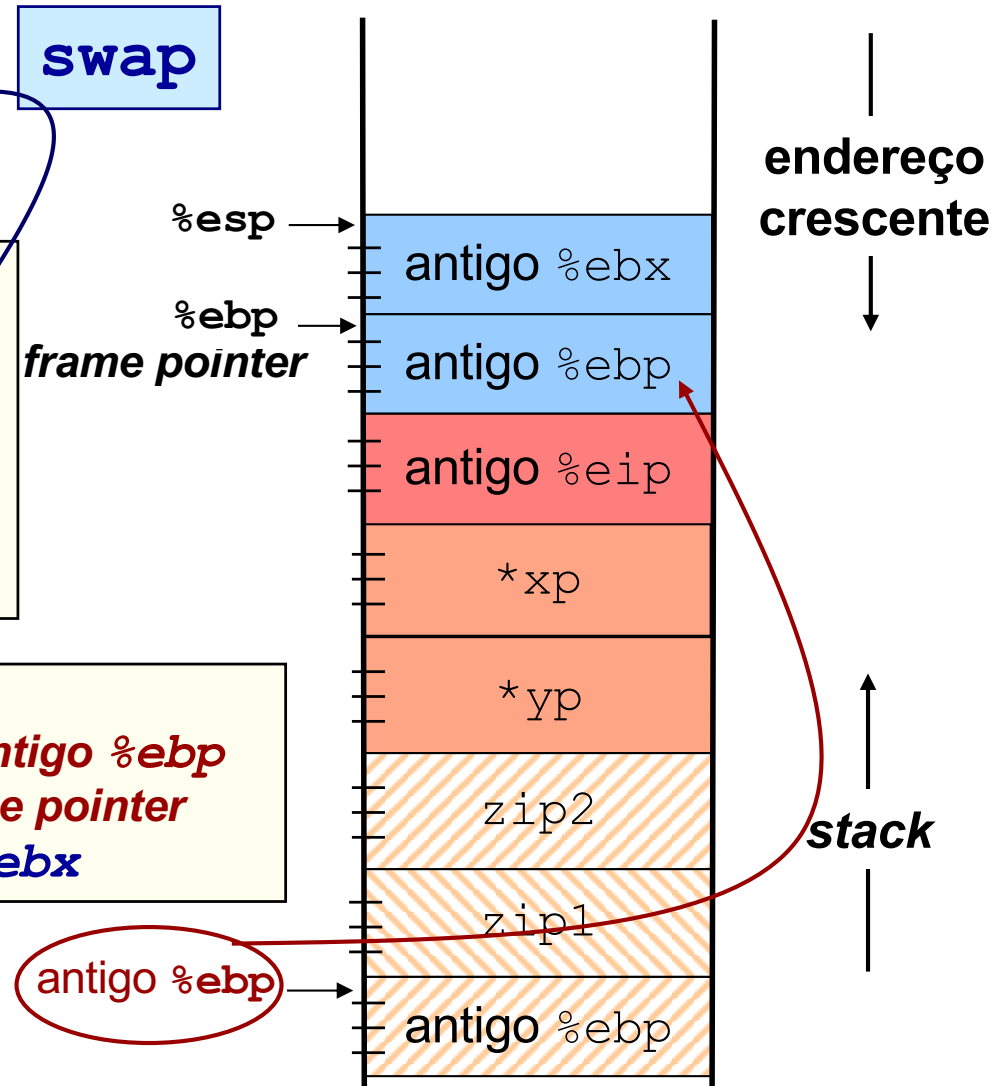
- atualizar *frame pointer*
- salvar registos
- reservar espaço p/ locais...*não...*

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    pushl    %ebp
    movl     %esp, %ebp
    pushl    %ebx
```

Salvaguarda antigo %ebp
Faz %ebp frame pointer
Salvaguarda %ebx

swap



Evolução da stack, no IA-32 (6)

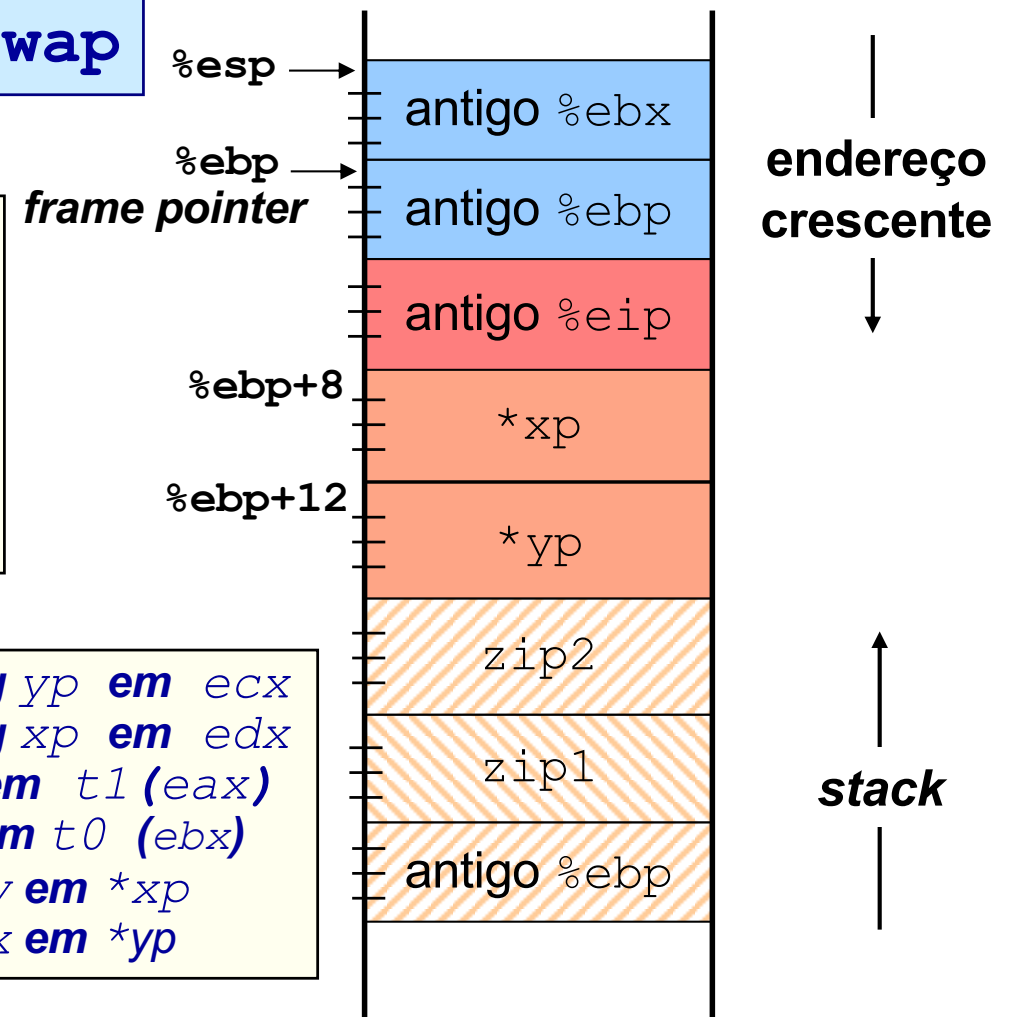


2. Corpo de swap

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

<code>movl 12(%ebp), %ecx</code>	Carrega arg yp em ecx
<code>movl 8(%ebp), %edx</code>	Carrega arg xp em edx
<code>movl (%ecx), %eax</code>	Coloca y em t1 (eax)
<code>movl (%edx), %ebx</code>	Coloca x em t0 (ebx)
<code>movl %eax, (%edx)</code>	Armazena y em *xp
<code>movl %ebx, (%ecx)</code>	Armazena x em *yp

swap



Evolução da stack, no IA-32 (7)

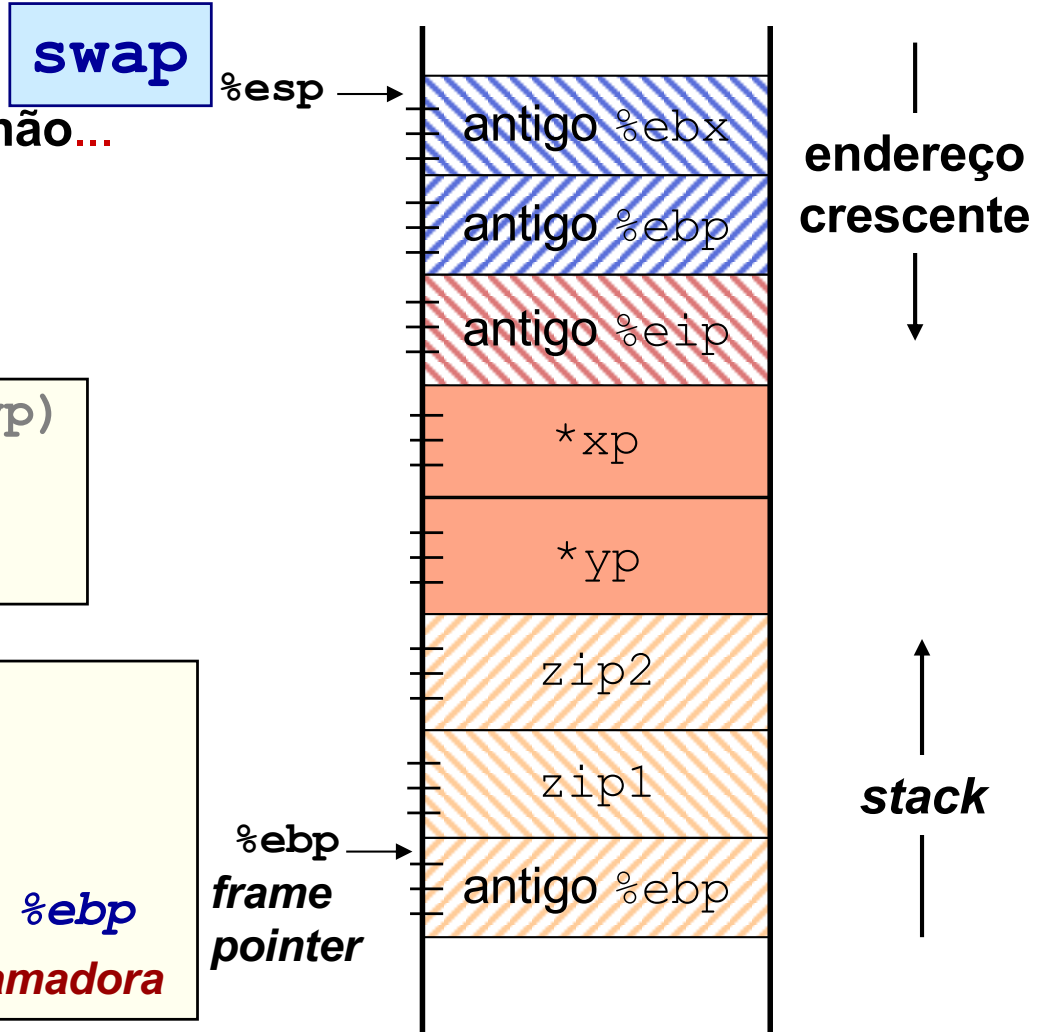


3. Término de swap ...

- libertar espaço de var locais...**não**...
- recuperar registos
- recuperar antigo *frame pointer*
- regressar a *call_swap*

```
void swap(int *xp, int *yp)
{
    (...)
}
```

```
popl %ebx          Recupera %ebx
movl %ebp, %esp    Recupera %esp
popl %ebp          Recupera %ebp
ou
leave             Recupera %esp, %ebp
ret               Regressa à f. chamadora
```



Evolução da stack, no IA-32 (8)



call_swap

4. Terminar invocação de swap...

- libertar espaço de parâmetros na *stack*...
- recuperar registos?...**não**...

```
void call_swap()
{
  int zip1 = 15213;
  int zip2 = 91125;
  (...)
  swap(&zip1, &zip2);
  (...)
}
```

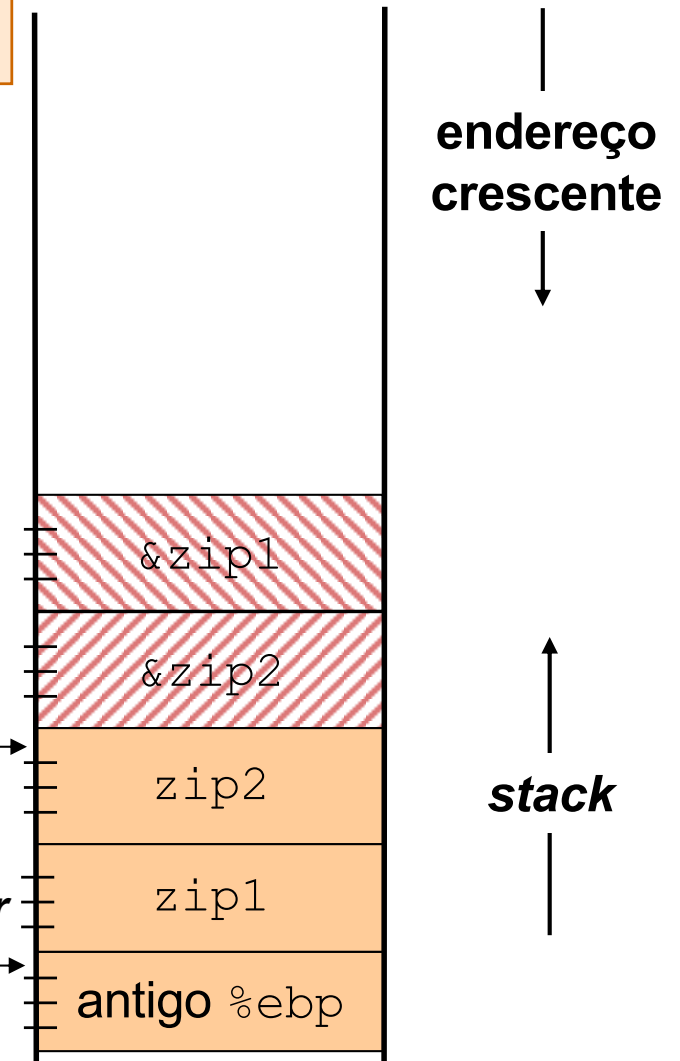
addl \$8,%esp

Atualiza stack pointer

frame pointer

%ebp →

%esp →





Análise de exemplos

– revisão do exemplo swap

- análise das fases: arranque/inicialização, corpo, término
- análise dos contextos (IA-32)
- evolução dos contextos na *stack* (IA-32)

– evolução de um exemplo: Fibonacci

- análise de uma compilação do gcc

– aninhamento e recursividade

- evolução ...

A série de Fibonacci no IA-32 (1)

```
int fib_dw(int n)
{
    int i = 0;
    int val = 0;
    int nval = 1;

    do {
        int t = val + nval;
        val = nval;
        nval = t;
        i++;
    } while (i < n);

    return val;
}
```

do-while

```
int fib_f(int n)
{
    int i;
    int val = 1;
    int nval = 1;

    for (i=1; i<n; i++) {
        int t = val + nval;
        val = nval;
        nval = t;
    }

    return val;
}
```

for

```
int fib_w(int n)
{
    int i = 1;
    int val = 1;
    int nval = 1;

    while (i < n) {
        int t = val + nval;
        val = nval;
        nval = t;
        i++;
    }

    return val;
}
```

while

função recursiva

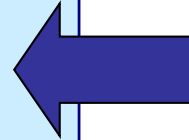
```
int fib_rec (int n)
{
    int prev_val, val;
    if (n ≤ 2)
        return (1);
    prev_val = fib_rec (n-2);
    val = fib_rec (n-1);
    return (prev_val+val);
}
```

A série de Fibonacci no IA-32 (2)



função recursiva

```
int fib_rec (int n)
{
    int prev_val, val;
    if (n ≤ 2)
        return (1);
    prev_val = fib_rec (n-2);
    val = fib_rec (n-1);
    return (prev_val+val);
}
```



_fib_rec:

```
    pushl %ebp
    movl  %esp, %ebp

    subl  $12, %esp
    movl  %ebx, -8(%ebp)
    movl  %esi, -4(%ebp)

    movl  8(%ebp), %esi
```

Atualiza frame pointer

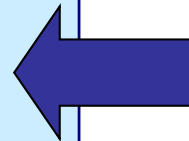
*Reserva espaço na stack para 3 int's
Salvaguarda os 2 reg's que vão ser usados;
de notar a forma de usar a stack...*

A série de Fibonacci no IA-32 (3)



função recursiva

```
int fib_rec (int n)
{
    int prev_val, val;
    if (n ≤ 2)
        return (1);
    prev_val = fib_rec (n-2);
    val = fib_rec (n-1);
    return (prev_val+val);
}
```



```
...
movl    %esi, -4(%ebp)
movl    8(%ebp), %esi
movl    $1, %eax
cmpl    $2, %esi
jle     L1
leal    -2(%esi), %eax
...
L1:
movl    -8(%ebp), %ebx
```

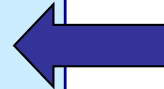
Coloca o argumento n em %esi
Coloca já o valor a devolver em %eax
Compara n:2
Se n ≤ 2, salta para o fim
Se não, ...

A série de Fibonacci no IA-32 (4)



função recursiva

```
int fib_rec (int n)
{
    int prev_val, val;
    if (n ≤ 2)
        return (1);
    prev_val = fib_rec (n-2);
    val = fib_rec (n-1);
    return (prev_val+val);
}
```



```
...
jle     L1
leal    -2(%esi), %eax
movl    %eax, (%esp)
call    fib_rec
movl    %eax, %ebx
leal    -1(%esi), %eax
...
```

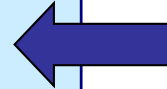
Se $n \leq 2$, salta para o fim
Se não, ... **calcula $n-2$, e...**
... **coloca-o no topo da stack (argumento)**
Invoca a função `fib_rec` e ...
... **guarda o valor de `prev_val` em `%ebx`**

A série de Fibonacci no IA-32 (5)



função recursiva

```
int fib_rec (int n)
{
    int prev_val, val;
    if (n ≤ 2)
        return (1);
    prev_val = fib_rec (n-2);
    val = fib_rec (n-1);
    return (prev_val+val);
}
```



```
...
movl    %eax, %ebx
leal    -1(%esi), %eax
movl    %eax, (%esp)
call    _fib_rec
leal    (%eax,%ebx), %eax
...
```

Calcula n-1, e...

... coloca-o no topo da stack (argumento)

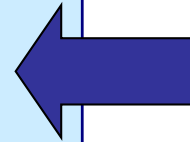
Chama de novo a função fib_rec

A série de Fibonacci no IA-32 (6)



função recursiva

```
int fib_rec (int n)
{
    int prev_val, val;
    if (n ≤ 2)
        return (1);
    prev_val = fib_rec (n-2);
    val = fib_rec (n-1);
    return (prev_val+val);
}
```



```
...
call    fib_rec
leal    (%eax,%ebx), %eax    Calcula e coloca em %eax o valor a devolver

L1:
movl    -8(%ebp), %ebx
movl    -4(%ebp), %esi       Recupera o valor dos 2 reg's usados
movl    %ebp, %esp          Atualiza o valor do stack pointer
popl    %ebp                Recupera o valor anterior do frame pointer
ret
```



Análise de exemplos

– revisão do exemplo swap

- análise das fases: arranque/inicialização, corpo, término
- análise dos contextos (IA-32)
- evolução dos contextos na *stack* (IA-32)

– evolução de um exemplo: Fibonacci

- análise de uma compilação do gcc

– aninhamento e recursividade

- evolução dos contextos na *stack*

Exemplo de cadeia de invocações no IA-32 (1)



Estrutura do código

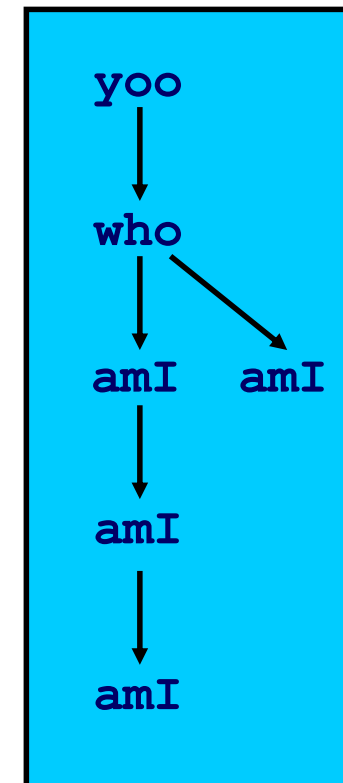
```
yoo (...)  
{  
  .  
  .  
  who () ;  
  .  
  .  
}
```

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

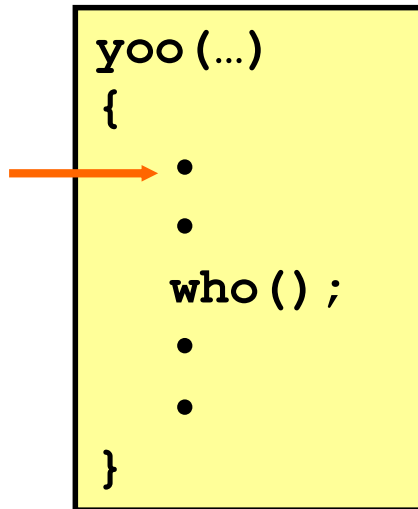
```
amI (...)  
{  
  .  
  .  
  amI () ;  
  .  
  .  
}
```

Função **amI** é recursiva

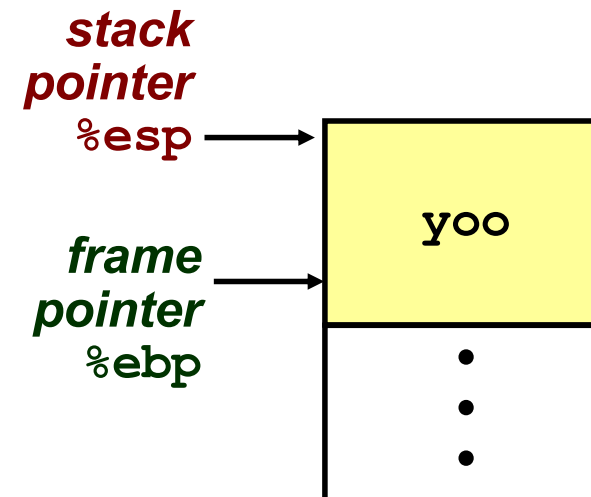
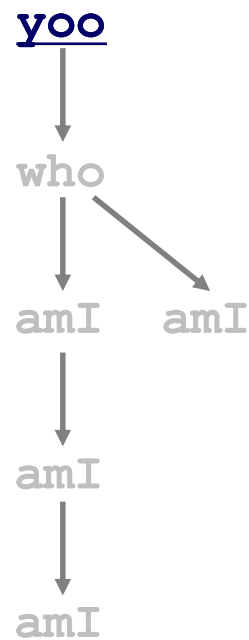
Cadeia de *call*



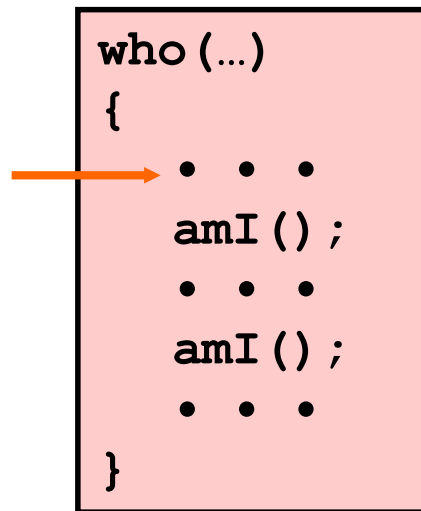
Exemplo de cadeia de invocações no IA-32 (2)



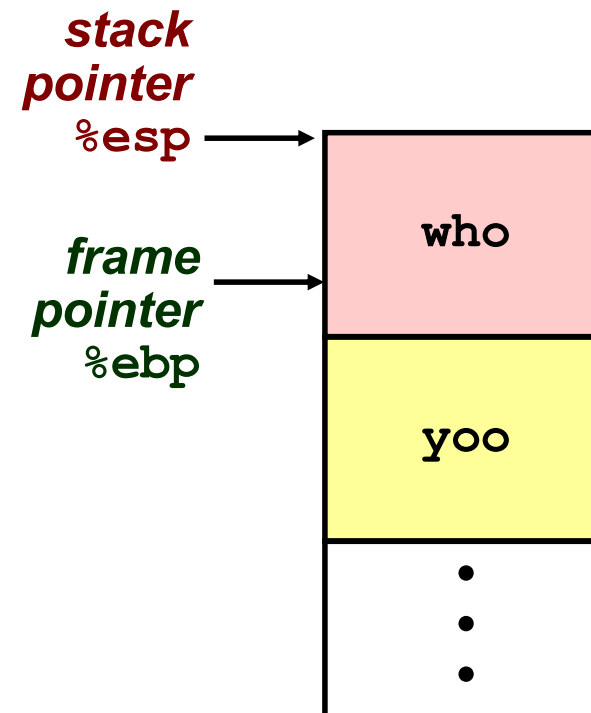
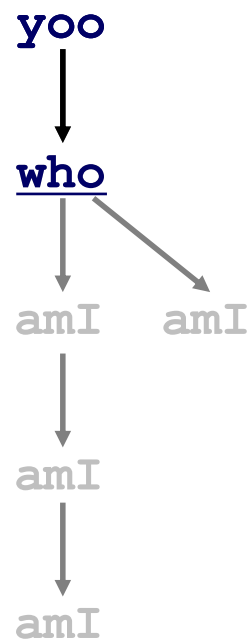
Cadeia de *call*



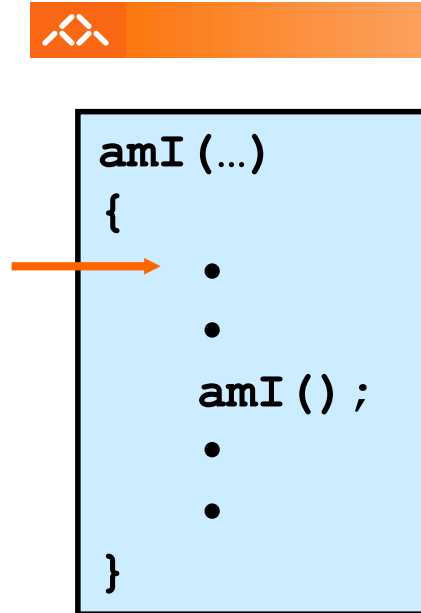
Exemplo de cadeia de invocações no IA-32 (3)



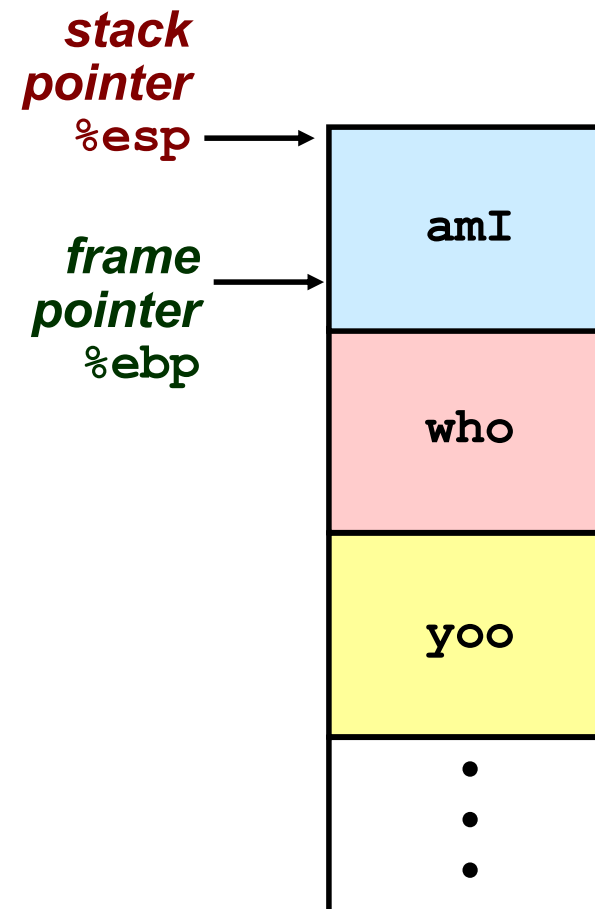
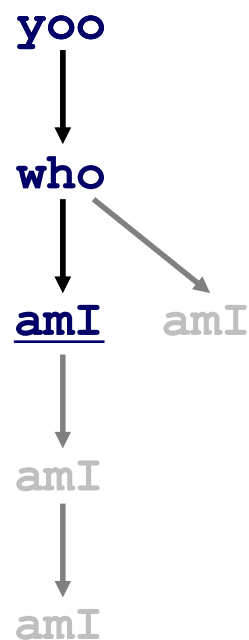
Cadeia de *call*



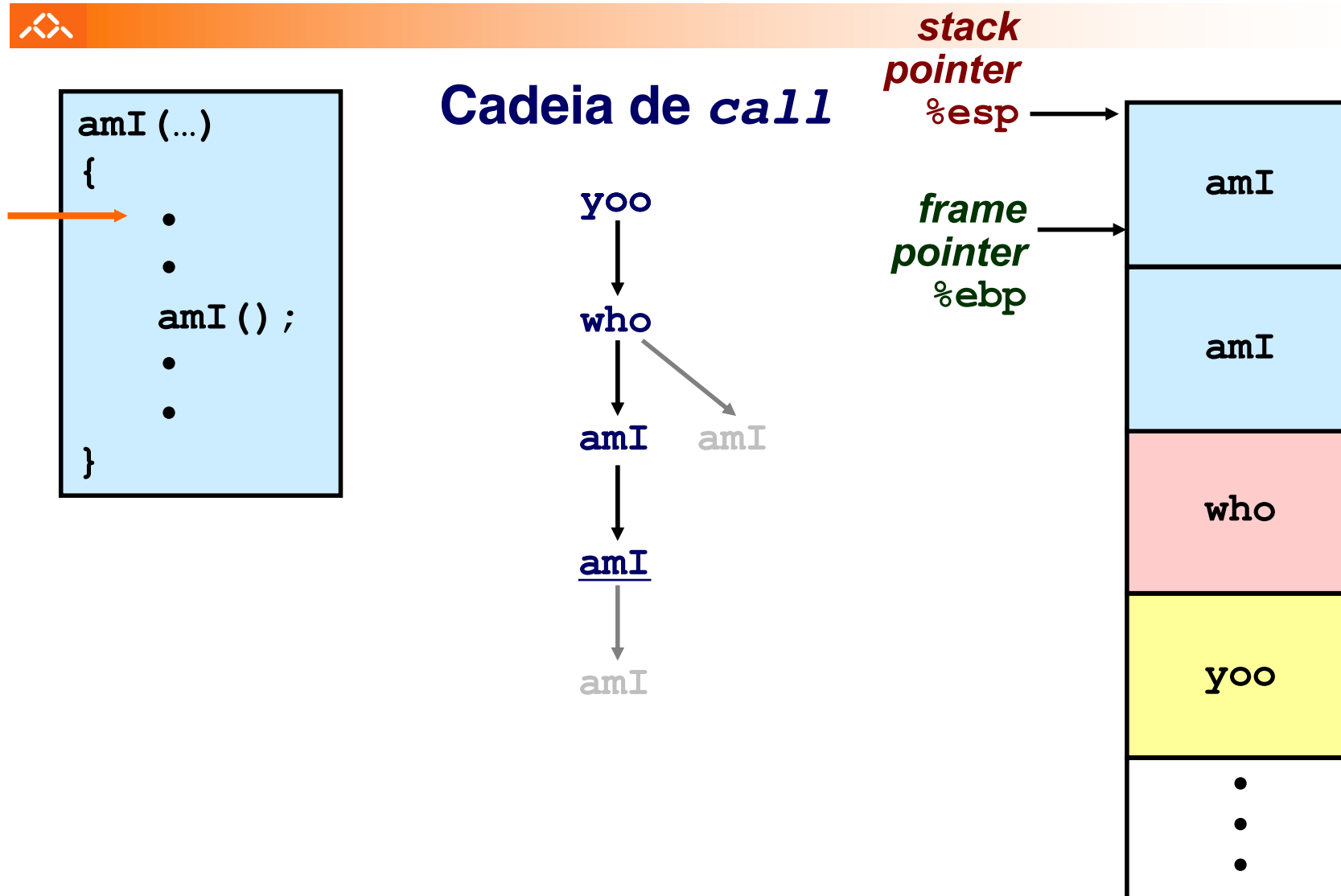
Exemplo de cadeia de invocações no IA-32 (4)



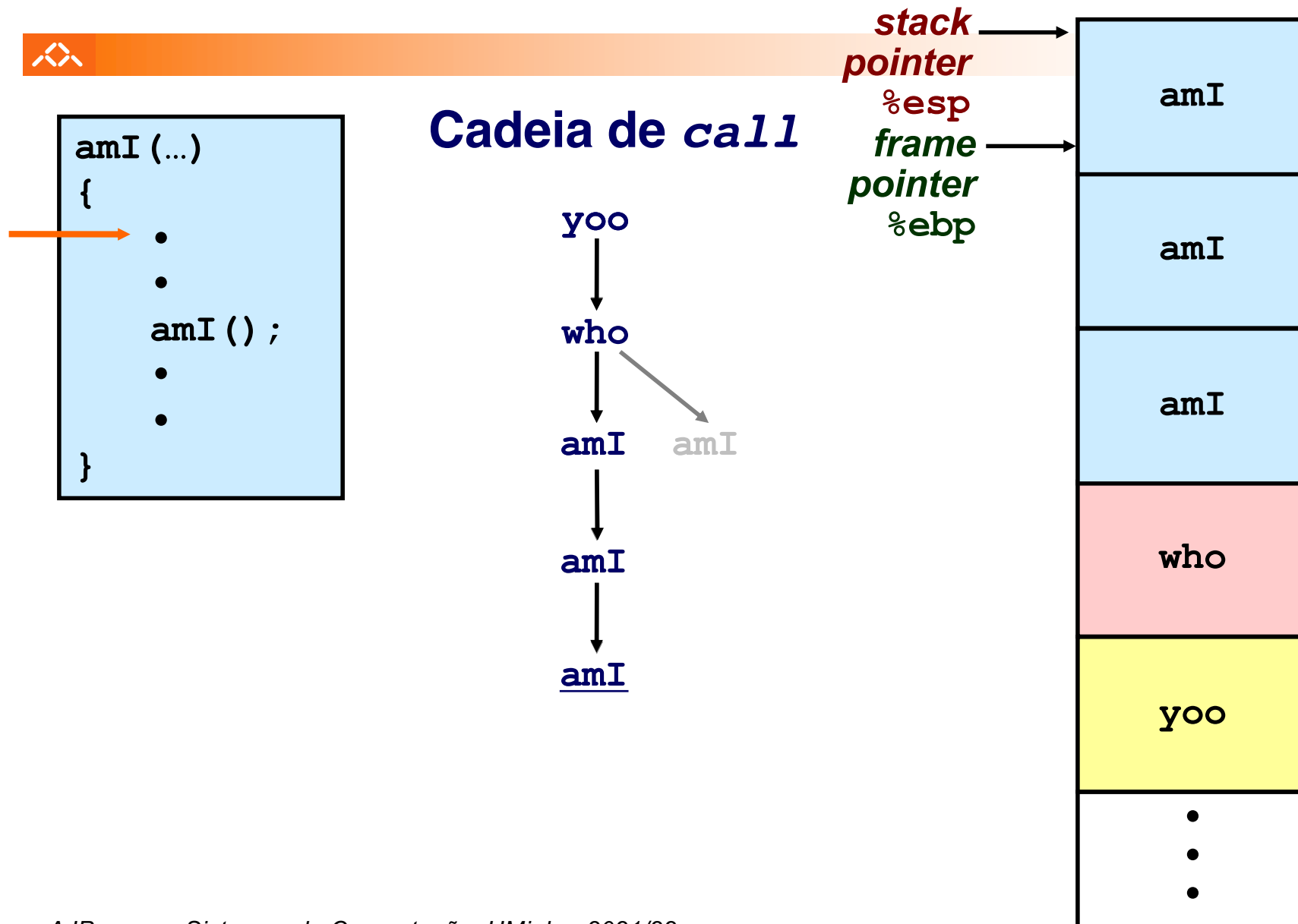
Cadeia de *call*



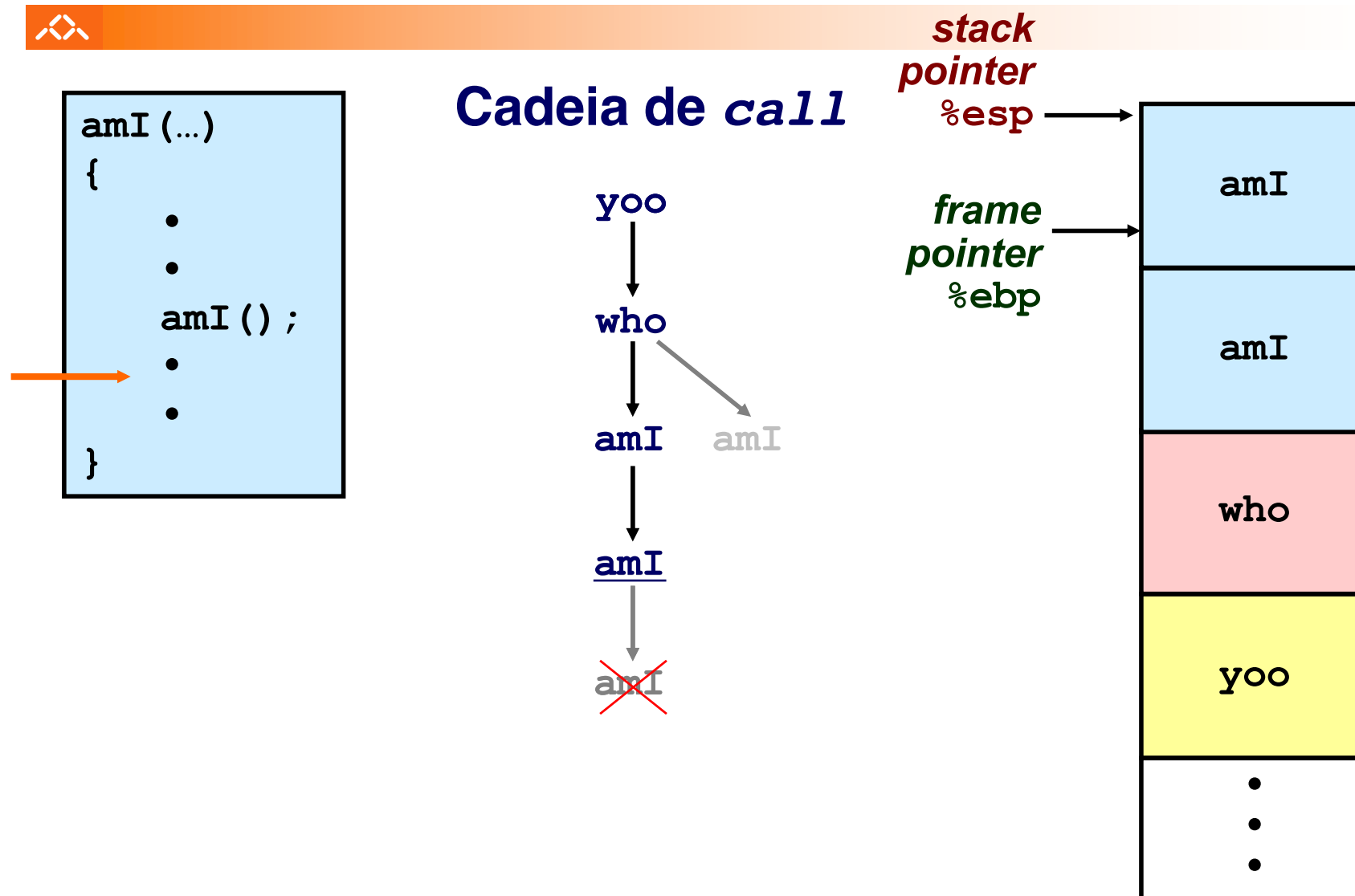
Exemplo de cadeia de invocações no IA-32 (5)



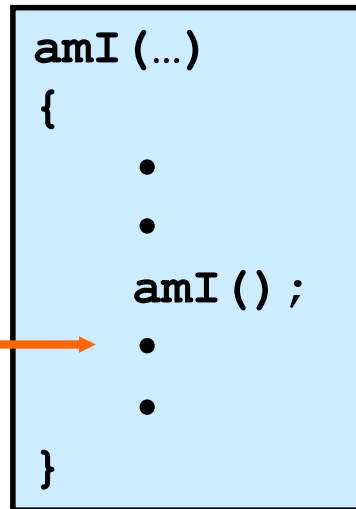
Exemplo de cadeia de invocações no IA-32 (6)



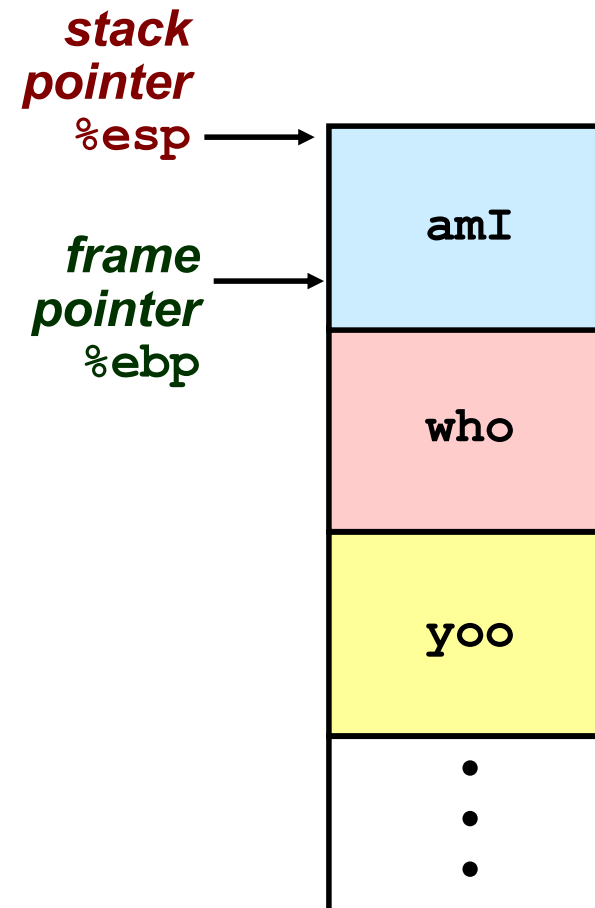
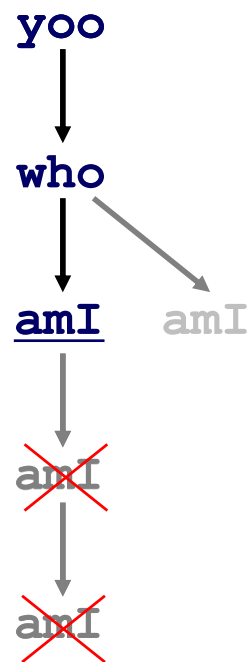
Exemplo de cadeia de invocações no IA-32 (7)



Exemplo de cadeia de invocações no IA-32 (8)



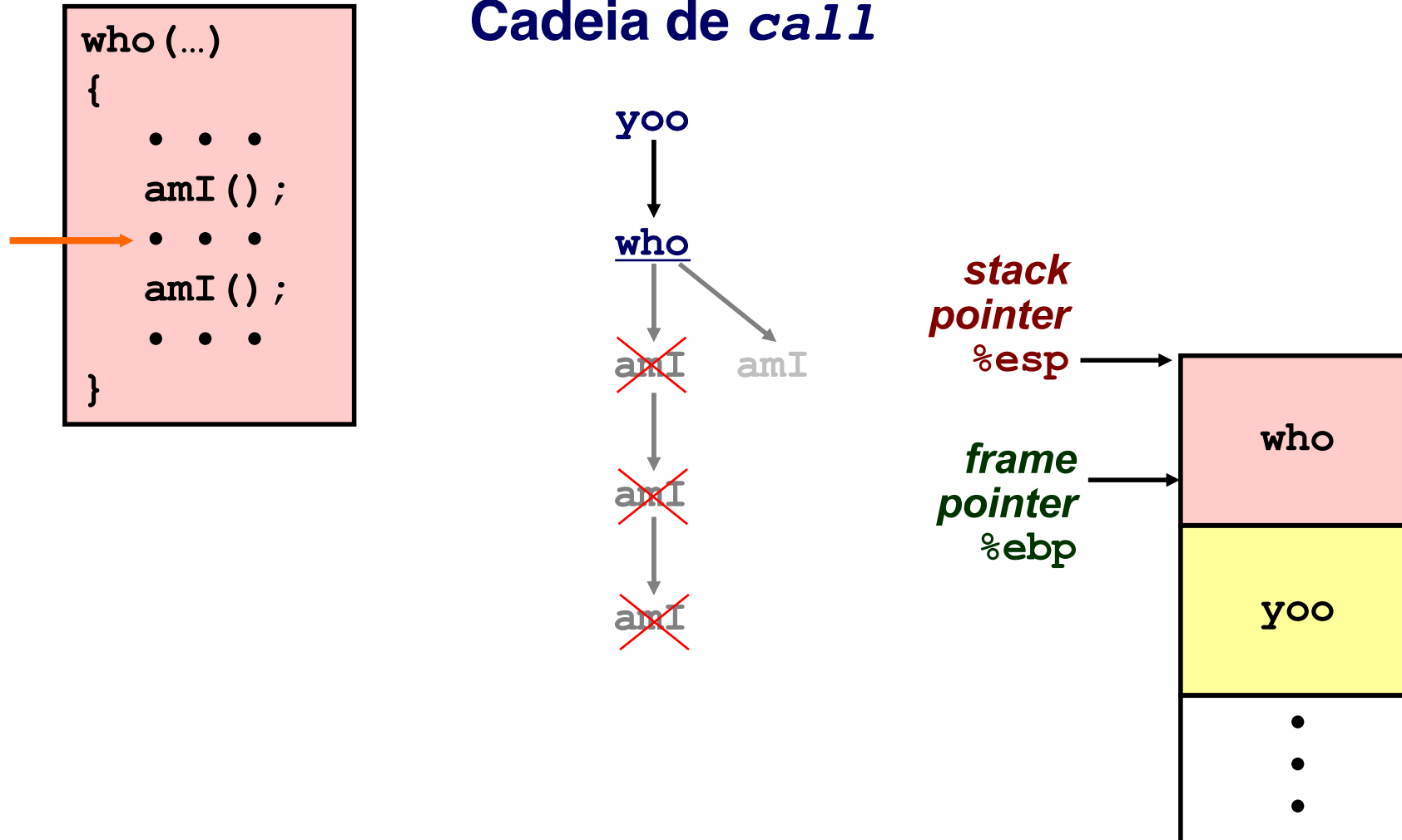
Cadeia de *call*



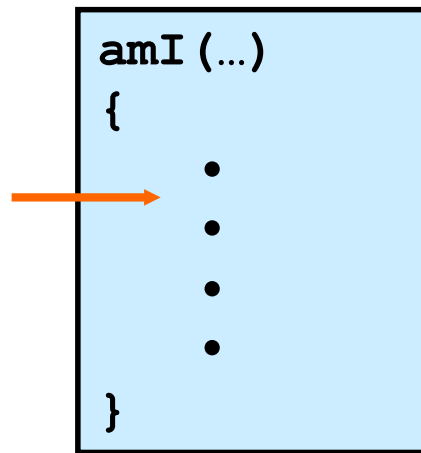
Exemplo de cadeia de invocações no IA-32 (9)



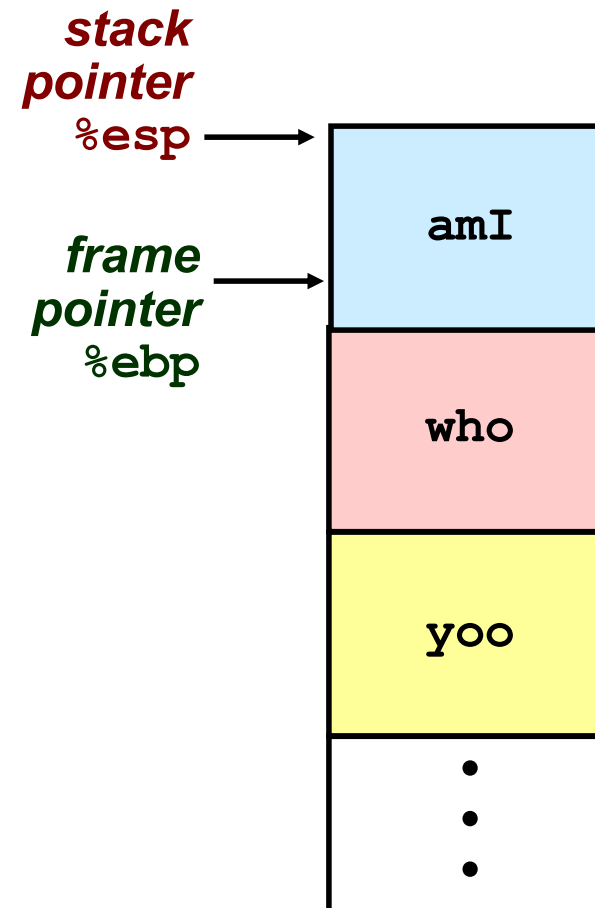
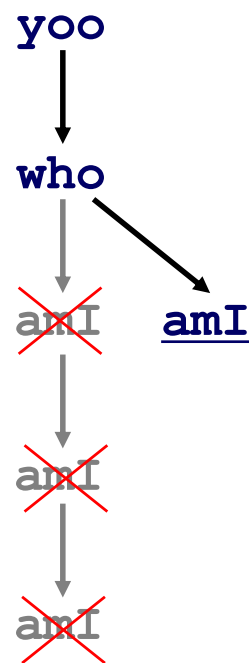
Cadeia de *call*



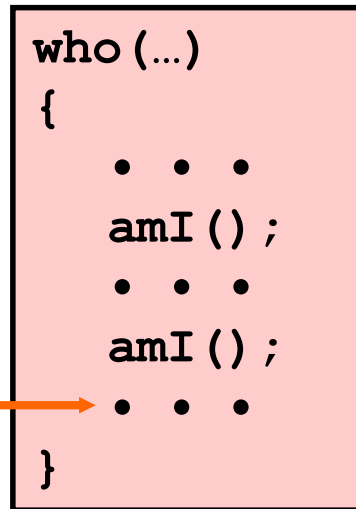
Exemplo de cadeia de invocações no IA-32 (10)



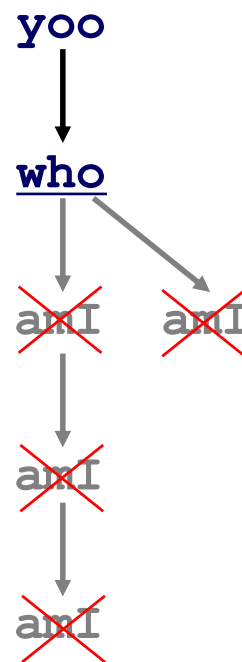
Cadeia de *call*



Exemplo de cadeia de invocações no IA-32 (11)

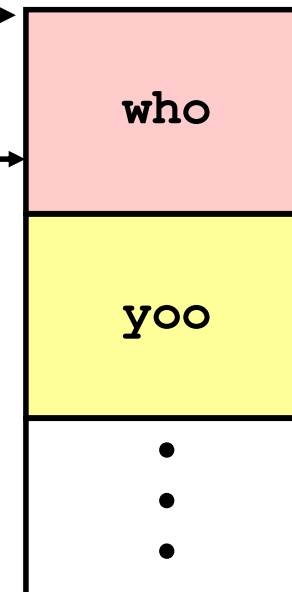


Cadeia de *call*



**stack
pointer
%esp**

**frame
pointer
%ebp**



Exemplo de cadeia de invocações no IA-32 (12)



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

Cadeia de *call*

