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, x86-64 e MIPS (RISC)
6. Acesso e manipulação de dados estruturados

Estrutura de uma função (/ procedimento)

- **função *versus* procedimento**
  - o nome duma função é usado como se fosse uma variável
  - uma função devolve um valor, um procedimento não
- **a 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
- **a menos visível em HLL** (gestão do contexto da função)**:**
  - variáveis locais (propriedades)
  - variáveis externas e globais (localização e acesso)
  - parâm's/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 ...


```
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

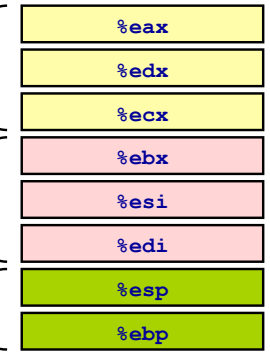
- **invocação** e regresso
  - instrução de salto, mas salvaguarda endereço de regresso
    - em registo (RISC; aninhamento / recursividade ?)
    - em memória/na *stack* (IA-32; aninhamento / recursividade ?)
- invocação e **regresso**
  - instrução de salto para o endereço de regresso
- **salvaguarda & 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 dos registos (de inteiros)

– Três do tipo *caller-save* — ***Caller-Save***
`%eax, %edx, %ecx`
- *save/restore*: função chamadora

– Três do tipo *callee-save* — ***Callee-Save***
`%ebx, %esi, %edi`
- *save/restore*: função chamada

– Dois apontadores (para a *stack*) — ***Pointers***
`%esp, %ebp`
- topo da *stack*, base/referência na *stack*



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

Análise do código de gestão de uma função

- invocação e regresso
  - instrução de salto, mas salvaguarda 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*, em *activation record* ou *frame*)
  - reserva/libertação de espaço para variáveis locais
  - atualização/recuperação do *frame pointer* (IA-32...)

Análise de exemplos

– **revisão do exemplo `swap`**
- análise das fases: 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 ...


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

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

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

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

Arranque | **Corpo** | **Término**


```
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

`call_swap` `swap`

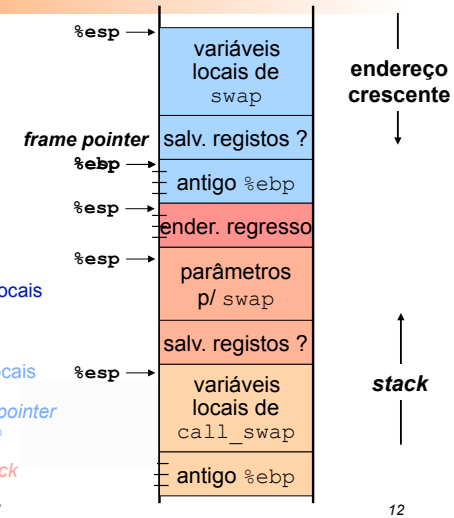
1. **Antes de invocar `swap`**
2. **Preparação p/ invocar `swap`**
   - salvaguardar registos?
   - passagem de parâmetros
3. **Invocar `swap`**
   - e guardar endereço de regresso

1. Início de `swap`
- atualizar *frame pointer*
- salvaguardar registos ?
- reservar espaço p/ var locais

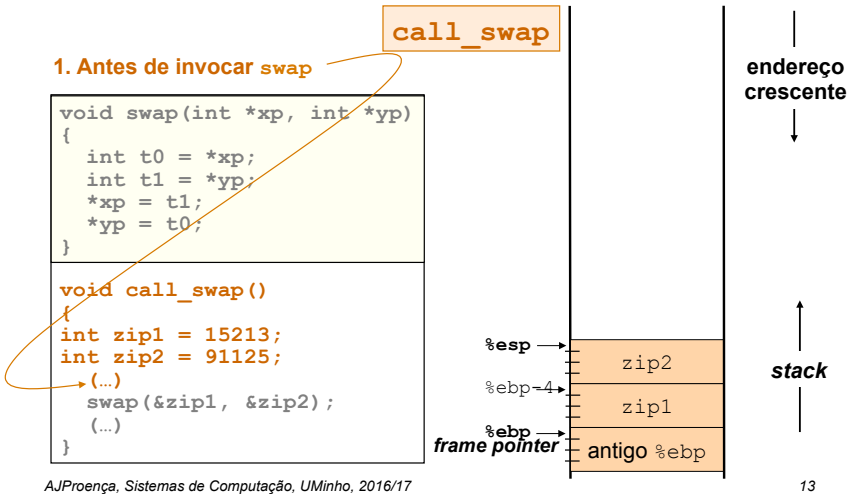
2. Corpo de `swap`

3. Término de `swap` ...
- libertar espaço de var locais
- recuperar registos ?
- recuperar antigo *frame pointer*
- regressar a `call_swap`

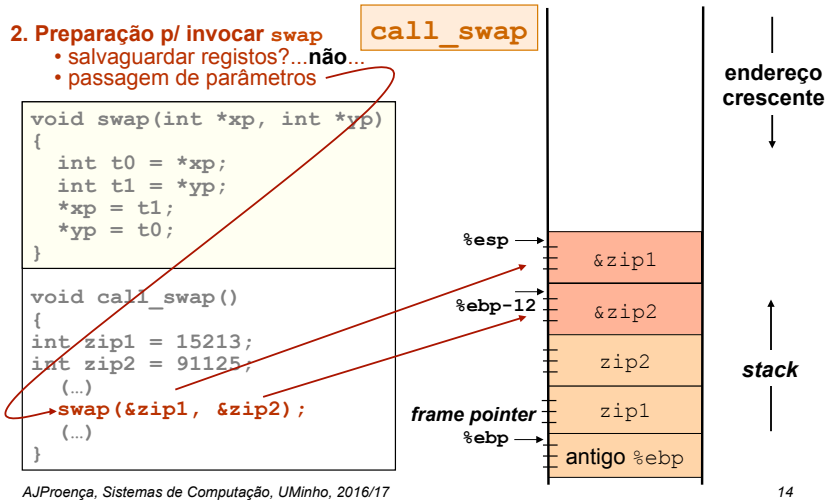
4. **Terminar invocação de `swap`...**
   - libertar espaço com parâmetros na *stack*
   - recuperar registos?



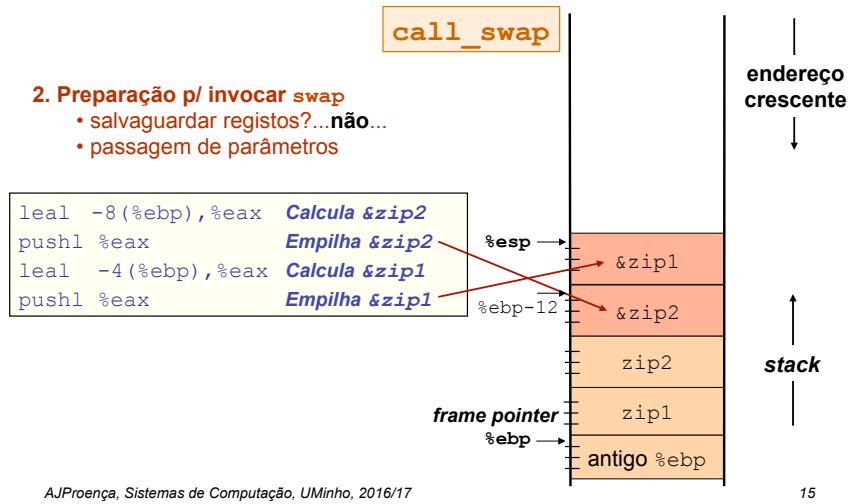
Evolução da stack, no IA-32 (1)



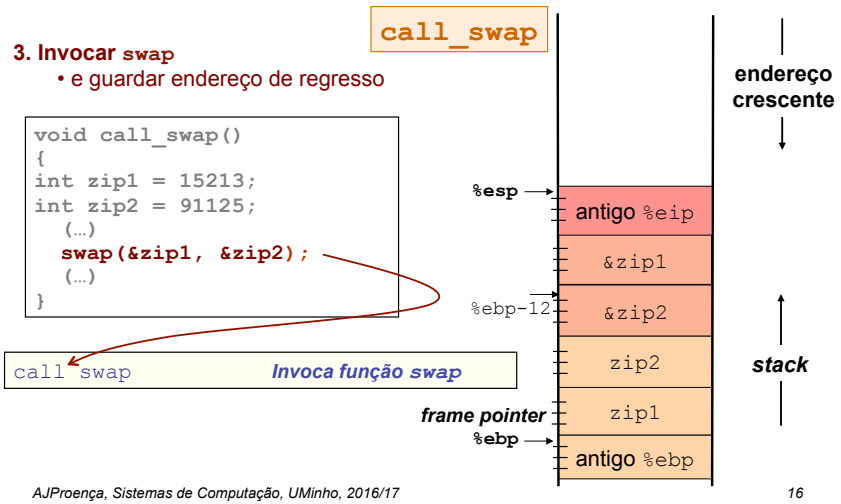
Evolução da stack, no IA-32 (2)



Evolução da stack, no IA-32 (3)



Evolução da stack, no IA-32 (4)

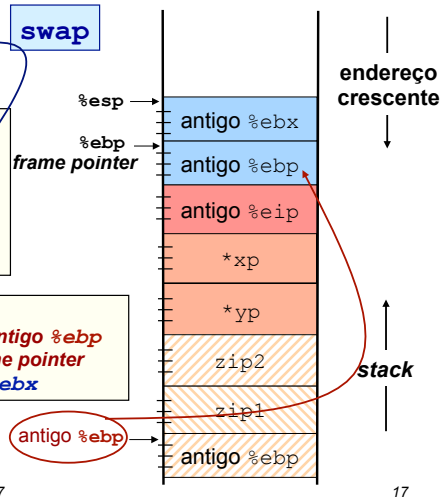


Evolução da stack, no IA-32 (5)

1. Início de `swap`
- atualizar *frame pointer*
- salvaguardar 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          Salvaguarda antigo %ebp
  movl   %esp,%ebp     Faz %ebp frame pointer
  pushl  %ebx          Salvaguarda %ebx
```

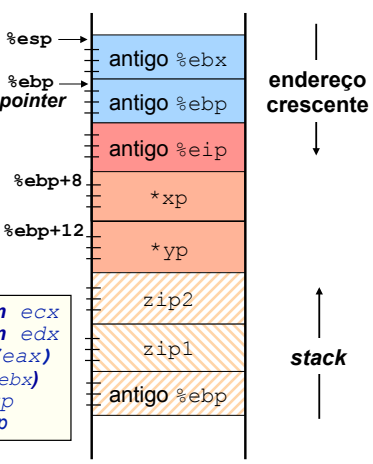


Evolução da stack, no IA-32 (6)

2. Corpo de `swap`

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

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

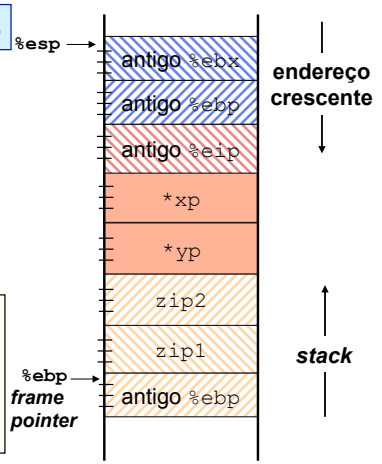


Evolução da stack, no IA-32 (7)

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
```

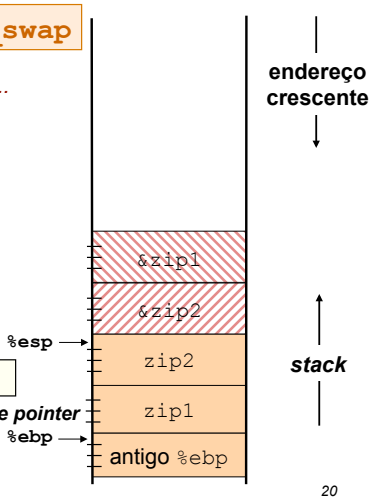


Evolução da stack, no IA-32 (8)

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
```



Análise de exemplos

- revisão do exemplo `swap`
  - análise das fases: 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 ...


```
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_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

```
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

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);
}
```


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          Atualiza frame pointer

   subl   $12, %esp           Reserva espaço na stack para 3 int's
   movl   %ebx, -8(%ebp)      Salvaguarda os 2 reg's que vão ser usados;
   movl   %esi, -4(%ebp)          de notar a forma de usar a stack...

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


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       Coloca o argumento n em %esi
   movl   $1, %eax            Coloca já o valor a devolver em %eax
   cmpl   $2, %esi            Compara n:2
   jle    L1                  Se n<=2, salta para o fim
   leal   -2(%esi), %eax      Se não, ...
   ...
L1:
   movl   -8(%ebp), %ebx
```

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              Se n<=2, salta para o fim
   leal  -2(%esi), %eax  Se não, ... calcula n-2, e...
   movl  %eax, (%esp)    ... coloca-o no topo da stack (argumento)
   call  _fib_rec        Invoca a função fib_rec e ...
   movl  %eax, %ebx      ... guarda o valor de prev_val em %ebx
   leal  -1(%esi), %eax
   ...
```


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     Calcula n-1, e...
   movl  %eax, (%esp)       ... coloca-o no topo da stack (argumento)
   call  _fib_rec           Chama de novo a função fib_rec
   leal  (%eax,%ebx), %eax
   ...
```


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: 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

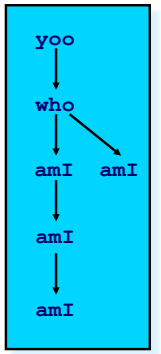
```
yoo(...)
{
   •
   •
   who();
   •
   •
}
```

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

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

Função **amI** é recursiva

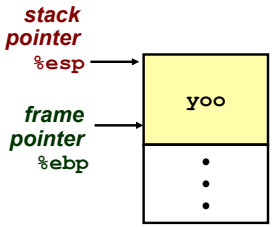
Cadeia de *Call*



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

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

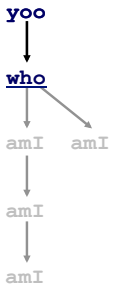
Cadeia de *Call*

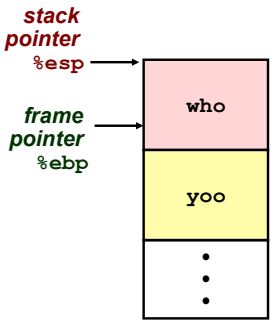


Exemplo de cadeia de invocações no IA-32 (3)

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

Cadeia de *Call*

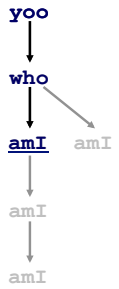


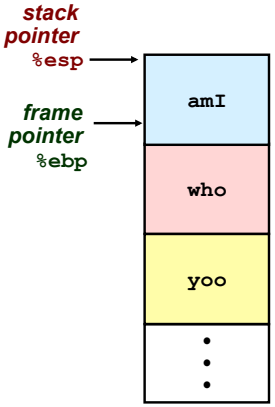


Exemplo de cadeia de invocações no IA-32 (4)

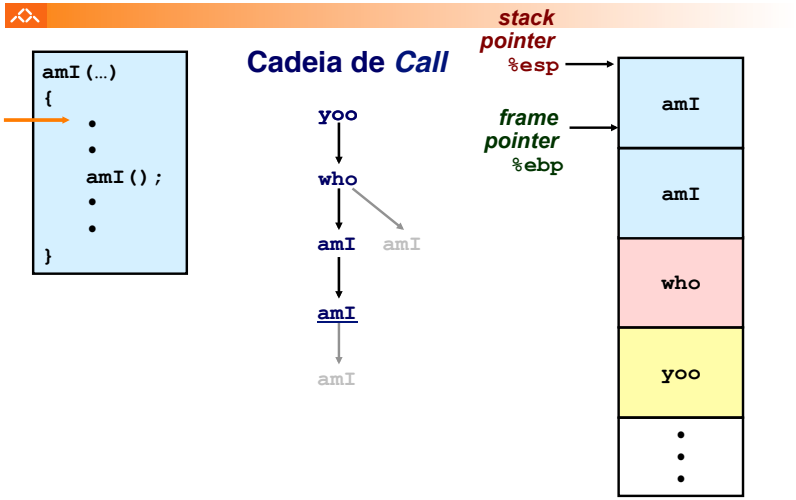
```
amI(...)
{
   •
   •
   amI();
   •
   •
}
```

Cadeia de *Call*

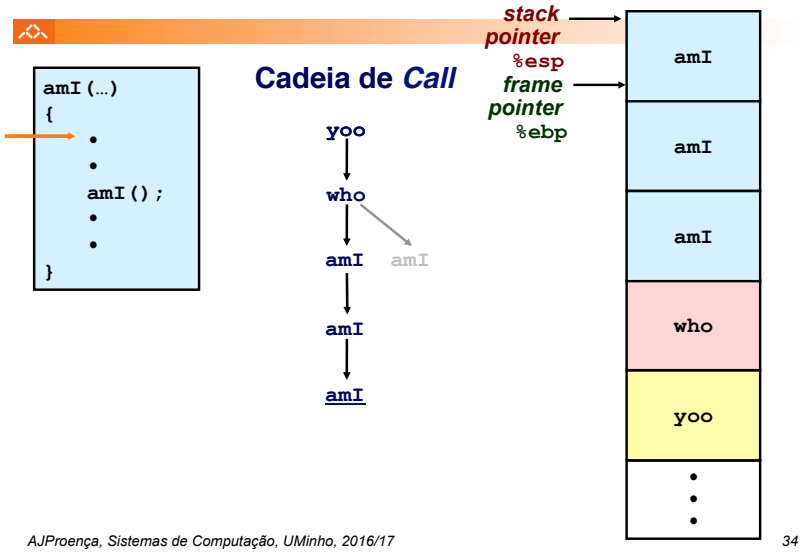




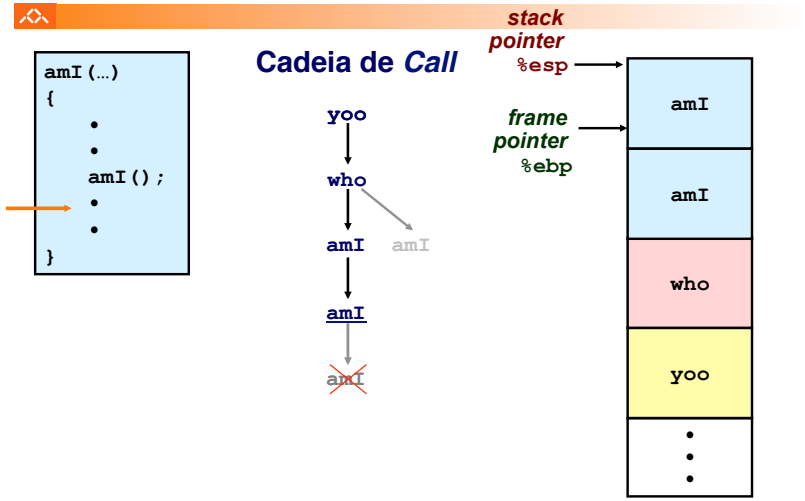
Exemplo de cadeia de invocações no IA-32 (5)



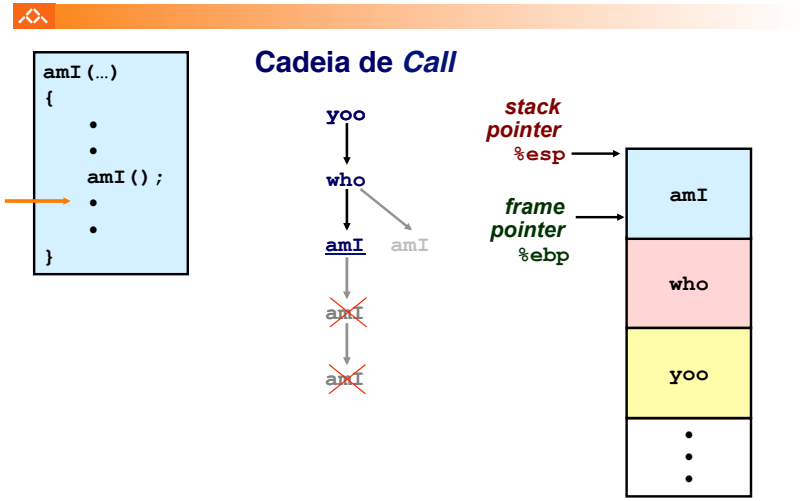
Exemplo de cadeia de invocações no IA-32 (6)



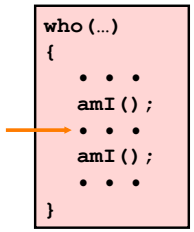
Exemplo de cadeia de invocações no IA-32 (7)



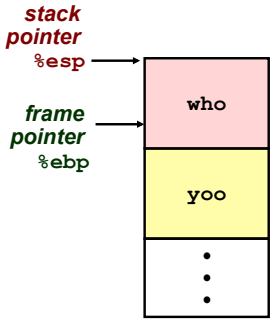
Exemplo de cadeia de invocações no IA-32 (8)



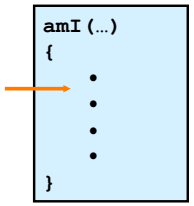
Exemplo de cadeia de invocações no IA-32 (9)



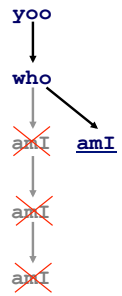
Cadeia de *Call*

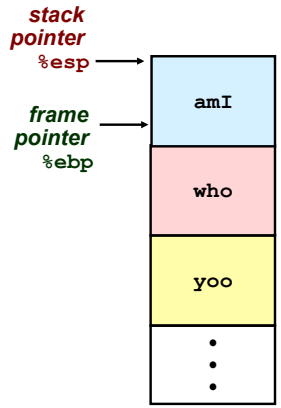


Exemplo de cadeia de invocações no IA-32 (10)

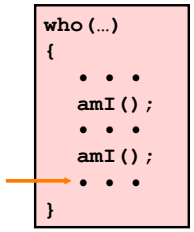


Cadeia de *Call*

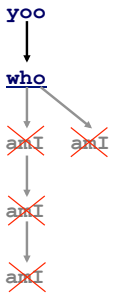


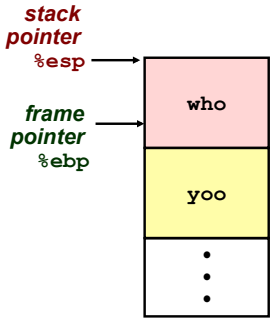


Exemplo de cadeia de invocações no IA-32 (11)

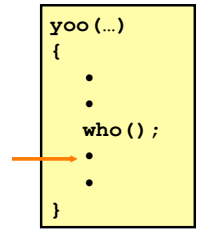


Cadeia de *Call*





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



Cadeia de *Call*