Software Básico

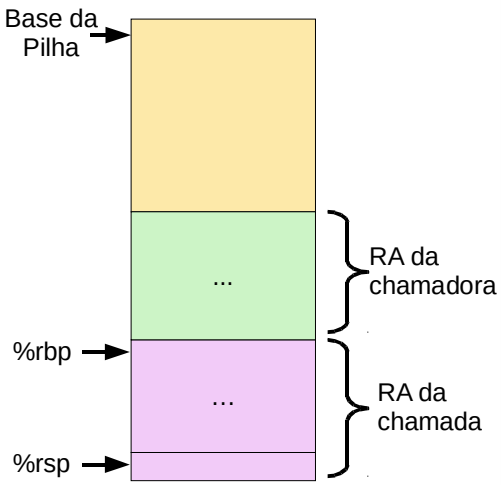
Procedimentos
Variáveis Locais

Reconhecimento

- Material produzido por:
  - Noemi Rodriguez – PUC-Rio
  - Ana Lúcia de Moura – PUC-Rio
- Adaptação
  - Bruno Silvestre – UFG

Registro de Ativação

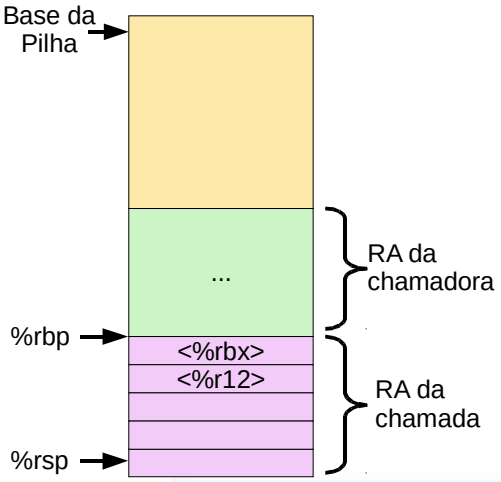
- Porção da pilha associada a uma chamada de função
- O registrador %rbp (*frame ou base pointer*) pode ser usado como base do registro de ativação
  - Acesso a elementos alocados no registro de ativação da função



Registro de Ativação: Registradores

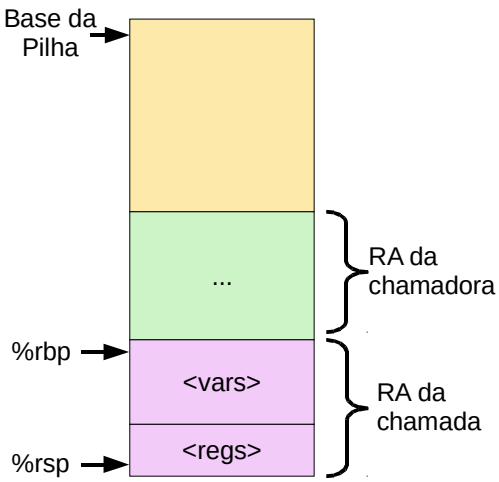
- RA é usado para guardar o valor de registradores *callee-saved*

```
foo:
  pushq %rbp
  movq  %rsp,%rbp

  /* múltiplo 16 */
  subq  $32,%rsp

  movq  %rbx, -8(%rbp)
  movq  %r12, -16(%rbp)
```



Registro de Ativação: Variáveis Locais

- RA também é usado para armazenar variáveis locais
  - Número de registradores é insuficiente
  - Arrays e estruturas



Alocação de Variáveis Locais

- Não existe convenção para a ordenação das variáveis locais no RA
  - Apenas a própria função manipula seus endereços
- Devem ser respeitadas as convenções de alinhamento
  - Variáveis escalares, arrays e estruturas
- A pilha deve permanecer alinhada em endereço múltiplo de 16

Alocação de Variáveis Escalares

- Endereço das variáveis deve ser alinhados
- Alocação de bloco de memória múltipla de 16
- Ordem não precisa ser mantida

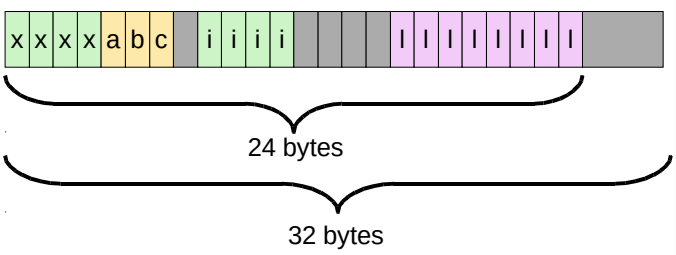
```
int foo() {
  int  x = 128;
  char a = 1;
  char b = 2;
  char c = 3;
  int  i = 1024;
  long l = 2048;
  ...
}
```

INSTITUTO DE INFORMÁTICA - UFG

Alocação de Variáveis Escalares

- Endereço das variáveis deve ser alinhados
- Alocação de bloco de memória múltipla de 16
- Ordem não precisa ser mantida

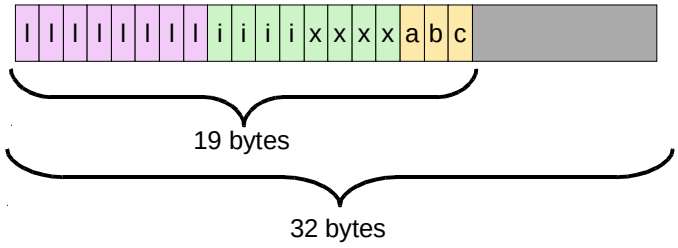
```
int foo() {
  int  x = 128;
  char a = 1;
  char b = 2;
  char c = 3;
  int  i = 1024;
  long l = 2048;
  ...
}
```



Alocação de Variáveis Escalares

- Endereço das variáveis deve ser alinhados
- Alocação de bloco de memória múltipla de 16
- Ordem não precisa ser mantida

```
int foo() {
  int  x = 128;
  char a = 1;
  char b = 2;
  char c = 3;
  int  i = 1024;
  long l = 2048;
  ...
}
```

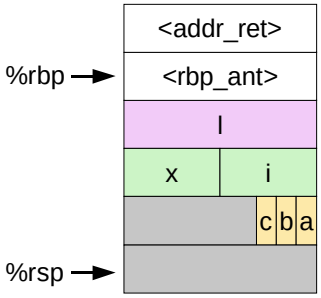


INSTITUTO DE INFORMÁTICA - UFG

Alocação de Variáveis Escalares

- Endereço das variáveis deve ser alinhados
- Alocação de bloco de memória múltipla de 16
- Ordem não precisa ser mantida

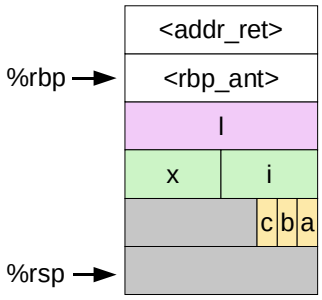
```
int foo() {
  int  x = 128;
  char a = 1;
  char b = 2;
  char c = 3;
  int  i = 1024;
  long l = 2048;
  ...
}
```



Acessando de Variáveis Escalares

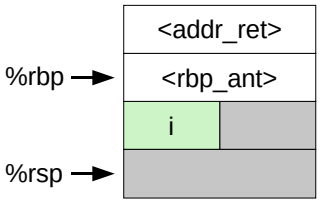
- Endereço das variáveis deve ser alinhados
- Alocação de bloco de memória múltipla de 16
- Ordem não precisa ser mantida

```
foo:
   pushq   %rbp
   movq    %rsp, %rbp
   subq    $32, %rsp
   movb      $3, -19(%rbp)  # c
   movb      $2, -18(%rbp)  # b
   movb      $1, -17(%rbp)  # a
   movl    $128, -16(%rbp)  # x
   movl   $1024, -12(%rbp)  # i
   movq   $2048,  -8(%rbp)  # l
```



Exemplo 1

```
int foo() {
  int i;
  scanf("%d", &i );
  return  i;
}
```




```
fmt: .string "%d"

foo:
   pushq     %rbp
   movq      %rsp, %rbp
   subq       $16, %rsp
   movq      $fmt, %rdi
   leaq  -8(%rbp), %rsi  # &i
   movl       $0, %eax
   call     scanf
   movl  -8(%rbp), %eax
   leave
   ret
```

Instrução “leaq”

- A instrução “leaq” resolve (calcula) o endereço final e o armazenando em um registrador
- Não há nenhum acesso à memória

```
leaq -8(%rbx), %rax          /* rax = %rbx - 8          */
leaq (%rbx, %rcx, 4), %rdi   /* rdi = %rbx + 4 * %rcx */
leaq 4( , %rcx, 2), %rax     /* rax = 4 + 2 * %rcx     */
```

Exemplo 2

- “x” deve ser uma região de memória por causa do scanf
- “n” e “s” podem ser registradores *callee-saved*
  - Não são alterados pela chamada a scanf
- Mas, se são *callee-saved*, temos que salvá-los antes de usá-los
- Resumindo, precisamos de 3 variáveis locais

```
int proc() {
  int x,n=5,s=0;
  while (n--) {
    scanf ("%d",&x);
    s += x;
  }
  return s;
}
```

Exemplo 2

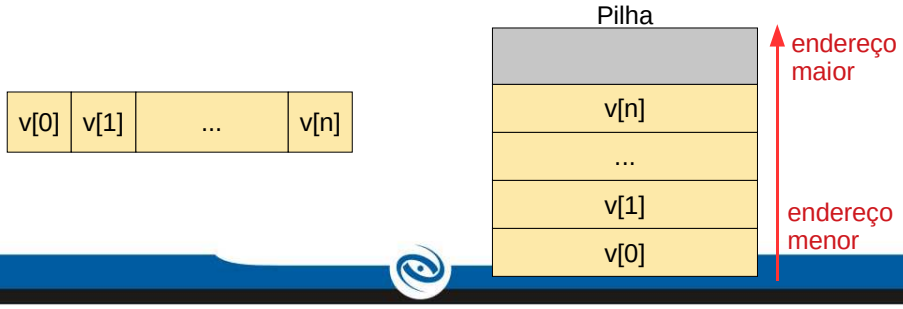
```
int proc() {
  int x,n=5,s=0;
  while (n--) {
    scanf ("%d",&x);
    s += x;
  }
  return s;
}
```

```
fmt: .string "%d"
proc:
  pushq %rbp
  movq  %rsp, %rbp
  subq   $32, %rsp
  movq  %rbx,  -8(%rbp)
  movq  %r12, -16(%rbp)
  movl    $5, %ebx
  movl    $0, %r12d
loop:
  movl %ebx, %ecx
  decl %ebx
  cmpl  $0, %ecx
  je    fim
  movq  $fmt, %rdi
  leaq  -20(%rbp), %rsi
  movl  $0, %eax
  call  scanf
  addl -20(%rbp), %r12d
  jmp  loop
fim:
  movl     %r12d, %eax
  movq  -8(%rbp), %rbx
  movq -16(%rbp), %r12
  leave
  ret
```

Arrays Locais

Arrays Locais

- O cálculo do endereços dos elementos de um array é feito a partir do seu endereço inicial
  - É esperado que &v[0] < &v[1] < … < &v[n]
  - O elemento de índice 0 é o primeiro elemento, e deve estar armazenado no menor endereço da pilha
- Lembrar do alinhamento de memória



Arrays Locais

```
void foo() {
 int  A[3];
 long B[2];
 calc(A, B);
}
```

3 * 4
2 * 8

28 bytes → 32 bytes (múltiplo de 16)

Arrays Locais

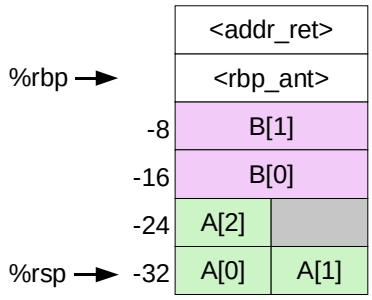
```
void foo() {
 int  A[3];
 long B[2];
 calc(A, B);
}
```

3 * 4 = 12

2 * 8 = 16

28 bytes → 32 bytes

```
foo:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $32,  %rsp

  leaq -32(%rbp), %rdi  # &A
  leaq -16(%rbp), %rsi  # &B
  call calc
```

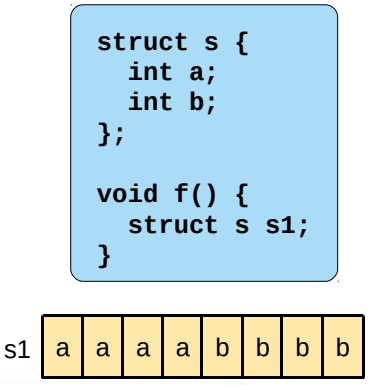


Estruturas Locais

Estruturas Locais

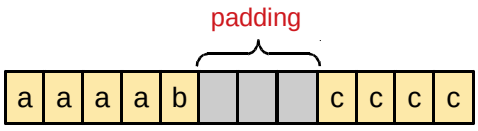
- Da mesma forma que array, o início da estrutura deve estar armazenado no menor endereço da pilha
- Lembrar do alinhamento de memória

```
struct s {
  int a;
  int b;
};

void f() {
  struct s s1;
}
```



Estruturas Locais

```
struct s {
  int  a;
  char b;
  int  c;
};

void foo() {
 struct s s1;
}
```



12 bytes → 16 bytes (múltiplo de 16)

alinhamento de 4 bytes (maior tipo “int”)

Estruturas Locais

```
void foo() {
 struct s s1;
 char *p = &s1.b;
 s1.c = 200;
}
```

```
struct s {
  int  a;
  char b;
  int  c;
};
```

```
foo:
  pushq %rbp
  movq  %rbp, %rsp
  subq  $16,  %rsp

  leaq -12(%rbp), %rsi
  movl $200, -8(%rbp)
```