Insper

Sistemas Hardware-Software

Aula 7 – Programação em nível de máquina (III)

2019 – Engenharia

Igor Montagner, Fábio Ayres <igorsm1@insper.edu.br>

Aula passadas

- Operações aritméticas
- Acessos a memória
- Chamadas de funções
- Expressões booleanas e pulos condicionais

Instruções de comparação

Permitem preencher os códigos de condição sem modificar os registradores:

- Instrução `cmp A, B`
  - Compara valores A e B
  - Funciona como `sub A, B` sem gravar resultado no destino

Flag set?	Significado
CF	Carry-out em **B – A**
ZF	**B == A**
SF	**(B – A) < 0** (quando interpretado como signed)
OF	Overflow de complemento-de-2: **(A > 0 && B < 0 && (B – A) < 0) || **(A < 0 && B > 0 && (B – A) > 0)**

Instruções de comparação

- Instrução `test A, B`
  - Testa o resultado de **A & B**
  - Funciona como `and A, B` sem gravar resultado no destino
  - Útil para checar um dos valores, usando o outro como máscara
  - Normalmente usado com A e B sendo o mesmo registrador, ou seja: `test %rdi, %rdi`

Flag set?	Significado
ZF	**A & B == 0**
SF	**A & B < 0** (quando interpretado como signed)

Acessando os códigos de condição

Instrução	Condição	Descrição
sete	ZF	Equal /Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	(signed) Negativo
setns	~SF	(signed) Não-negativo
setl	(SF^OF)	(signed) Less than
setle	(SF^OF)|ZF	(signed) Less than or Equal
setge	~(SF^OF)	(signed) Greater than or Equal
setg	~(SF^OF) & ~ZF	(signed) Greater than
setb	CF	(unsigned) Below
seta	~CF & ~ZF	(unsigned) Above

Desvios (ou saltos) condicionais

Instrução	Condição	Descrição
`jmp`	`1`	Incondicional
`je`	`ZF`	Equal /Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	(signed) Negativo
`jns`	`~SF`	(signed) Não-negativo
`jl`	`(SF^OF)`	(signed) Less than
`jle`	`(SF^OF)|ZF`	(signed) Less than or Equal
`jge`	`~(SF^OF)`	(signed) Greater than or Equal
`jg`	`~(SF^OF) & ~ZF`	(signed) Greater than
`jb`	`CF`	(unsigned) Below
`ja`	`~CF & ~ZF`	(unsigned) Above

O par de comandos **if-goto**

O par de comandos if-goto é equivalente às instruções cmp/test seguidas de um jump condicional

```
cmp 0x4, %rdi
jle label
(bloco 1)
label:
...
```

```
if (a <= 4) {
   goto
label;
}
(bloco1)
label:
. . .
```

Vamos chamar código **C** que use somente `if-goto` de **gotoC**!

Exercício da aula anterior

```
0000000000000000 <ex6>:
0:   48 39 f7                cmp    %rsi,%rdi
3:   7e 03                   jle    8 <ex6+0x8>
5:   48 89 fe                mov    %rdi,%rsi
8:   48 85 ff                test   %rdi,%rdi
b:   7e 03                   jle    10 <ex6+0x10>
d:   48 f7 de                neg    %rsi
10:  89 f0                   mov    %esi,%eax
12:  c3                      retq
```

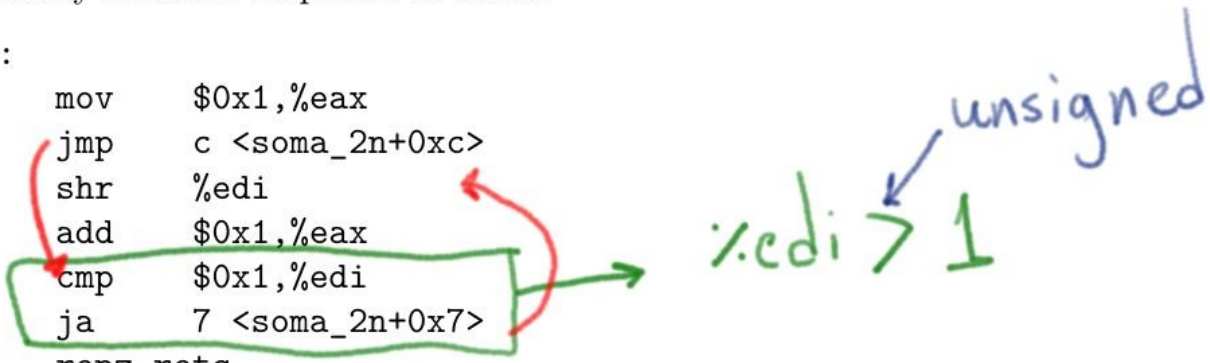
Loops

Façam exercício 1 do handout!

Resultado 1 – setas e comparação

Exercício 1: veja o código assembly abaixo e responda os itens.

```
0000000000000000 <soma_2n>:
0:   b8 01 00 00 00          mov    $0x1,%eax
5:   eb 05                   jmp    c <soma_2n+0xc>
7:   d1 ef                   shr    %edi
9:   83 c0 01                add    $0x1,%eax
c:   83 ff 01                cmp    $0x1,%edi
f:   77 f6                   ja     7 <soma_2n+0x7>
11:  f3 c3                   repz retq
```



1. Localize no código acima as instruções de saltos (`jmp` ou condicionais `j*`). Desenhe setas indicando para qual linha do código elas pulam.

Resultado 2 – versão if-goto

Exercício 1: veja o código assembly abaixo e responda os itens.

```
0000000000000000 <soma_2n>:
0:   b8 01 00 00 00          mov    $0x1,%eax
5:   eb 05                   jmp    c <soma_2n+0xc>
7:   d1 ef                   shr    %edi
9:   83 c0 01                add    $0x1,%eax
c:   83 ff 01                cmp    $0x1,%edi
f:   77 f6                   ja     7 <soma_2n+0x7>
11:  f3 c3                   repz retq
```

%edi → unsigned int a;

```
int r=1;
goto Lc;
L7:
  a=a>>1;
  r+=1;
Lc: if(a>1) goto L7;
return r;
```

1. Localize no código acima as instruções de saltos (`jmp` ou condicionais `j*`). Desenhe setas indicando para qual linha do código elas pulam.

Resultado 3 - versão final

Exercício 1: veja o código assembly abaixo e responda os itens.

```
0000000000000000 <soma_2n>:
0:   b8 01 00 00 00          mov    $0x1,%eax
5:   eb 05                   jmp    c <soma_2n+0xc>
7:   d1 ef                   shr    %edi
9:   83 c0 01                add    $0x1,%eax
c:   83 ff 01                cmp    $0x1,%edi
f:   77 f6                   ja     7 <soma_2n+0x7>
11:  f3 c3                   repz retq
```

```
int soma_2n(unsigned int a){
    int r=1;
    while(a>1){
        a = a/2;
        r++;
    }
    return r;
}
```

1. Localize no código acima as instruções de saltos (`jmp` ou condicionais `j*`). Desenhe setas indicando para qual linha do código elas pulam.

while

While version

```
while (Test)
  Body
```

Goto Version

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

while

```
long foo_while(long n) {
  long sum = 0;

  while (n > 0) {
    sum += n;
    n--;
  }

  sum *= sum;
  return sum;
}
```

```
long foo_while_goto_1(long n) {
  long sum = 0;

  goto test;

loop:
  sum += n;
  n--;

test:
  if (n > 0)
    goto loop;

  sum *= sum;
  return sum;
}
```

while

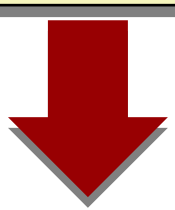
```
long foo_while_goto_1(long n) {
  long sum = 0;

  goto test;

loop:
  sum += n;
  n--;

test:
  if (n > 0)
    goto loop;

  sum *= sum;
  return sum;
}
```

```
0000000000000044 <foo_while_goto_1>:
  44:   mov    $0x0,%eax
  49:   jmp    52 <foo_while_goto_1+0xe>
  4b:   add    %rdi,%rax
  4e:   sub    $0x1,%rdi
  52:   test   %rdi,%rdi
  55:   jg     4b <foo_while_goto_1+0x7>
  57:   imul   %rax,%rax
  5b:   retq
```

`while`, versão 2

While version

```
while (Test)
  Body
```



Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

while

```
long foo_while(long n) {
  long sum = 0;

  while (n > 0) {
    sum += n;
    n--;
  }

  sum *= sum;
  return sum;
}
```

```
long foo_while_do_while(long n) {
  long sum = 0;

  if (n <= 0)
    goto done;

  do {
    sum += n;
    n--;
  } while (n > 0);

done:
  sum *= sum;
  return sum;
}
```

while

```
long foo_while_do_while(long n) {
  long sum = 0;

  if (n <= 0)
    goto done;

  do {
    sum += n;
    n--;
  } while (n > 0);

done:
  sum *= sum;
  return sum;
}
```

```
long foo_while_goto_2(long n) {
  long sum = 0;

  if (n <= 0)
    goto done;

loop:
  sum += n;
  n--;

  if (n > 0)
    goto loop;

done:
  sum *= sum;
  return sum;
}
```

while

```
long foo_while_goto_2(long n) {
  long sum = 0;

  if (n <= 0)
    goto done;

loop:
  sum += n;
  n--;

  if (n > 0)
    goto loop;

done:
  sum *= sum;
  return sum;
}
```

```
000000000000007e <foo_while_goto_2>:
  7e:   test   %rdi,%rdi
  81:   jle    96 <foo_while_goto_2+0x18>
  83:   mov    $0x0,%eax
  88:   add    %rdi,%rax
  8b:   sub    $0x1,%rdi
  8f:   test   %rdi,%rdi
  92:   jg     88 <foo_while_goto_2+0xa>
  94:   jmp    9b <foo_while_goto_2+0x1d>
  96:   mov    $0x0,%eax
  9b:   imul   %rax,%rax
  9f:   retq
```

for

For Version

```
for (Init; Test; Update )
        Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update;
}
```

for

```
long foo_for(long n) {
  long sum;

  for (sum = 0; n > 0; n--) {
    sum += n;
  }

  sum *= sum;
  return sum;
}
```

```
long foo_while(long n) {
  long sum = 0;

  while (n > 0) {
    sum += n;
    n--;
  }

  sum *= sum;
  return sum;
}
```

for

while

```
000000000000002c <foo_while>:
  2c:    mov    $0x0,%eax
  31:    jmp    3a <foo_while+0xe>
  33:    add    %rdi,%rax
  36:    sub    $0x1,%rdi
  3a:    test   %rdi,%rdi
  3d:    jg     33 <foo_while+0x7>
  3f:    imul   %rax,%rax
  43:    retq
```

for

```
00000000000000a0 <foo_for>:
  a0:    mov    $0x0,%eax
  a5:    jmp    ae <foo_for+0xe>
  a7:    add    %rdi,%rax
  aa:    sub    $0x1,%rdi
  ae:    test   %rdi,%rdi
  b1:    jg     a7 <foo_for+0x7>
  b3:    imul   %rax,%rax
  b7:    retq
```

Loops

Faça o exercício 2 do handout!

Quem acabar pode fazer os extras!

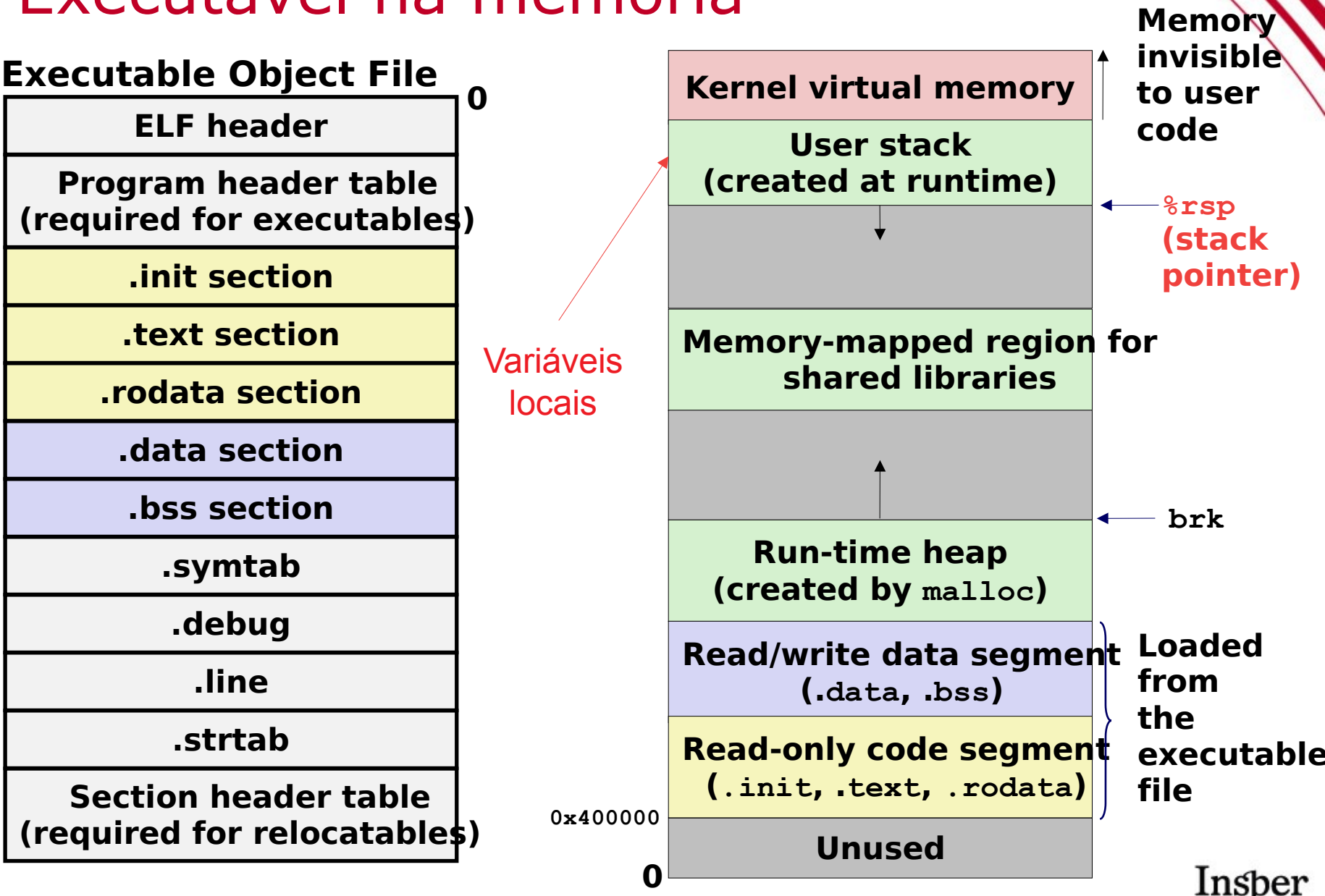
30 minutos

Variáveis locais

- Na maioria do tempo são colocadas em registradores
- Se não for possível colocamos na pilha (memória)
- Topo da pilha está armazenado em %rsp
- Sabemos acessar memória de maneira relativa a %rsp

$0xF(%rsp)

Executável na memória



Criando variáveis locais

```
sub $0x10, %rsp

. . .
mov 0x4(%rsp), %rdx
. . .
mov %rdx, 0x4(%rsp)
. . .


add $0x10, %rsp
```

- Subtrair de %rsp equivale a empilhar, somar equivale a desempilhar
- Não existe suporte para operações memória-memória

- No fim da função deletamos todas as variáveis locais

Atividade

- Parte 2 do handout

Insper

www.insper.edu.br

do-while

C Code

```
do
   Body
   while (Test);
```

Goto Version

```
loop:
   Body
   if (Test)
      goto loop
```

Body:

```
{
   Statement1;
   Statement2;
      …
   Statementn;
}
```

do-while

```
long foo_do_while(long n) {
  long sum = 0;

  do {
    sum += n;
    n--;
  } while (n > 0);

  sum *= sum;
  return sum;
}
```

```
long foo_do_while_goto(long n) {
  long sum = 0;

loop:
  sum += n;
  n--;

  if (n > 0)
    goto loop;

  sum *= sum;
  return sum;
```

do-while

```
long foo_do_while_goto(long n) {
  long sum = 0;

loop:
  sum += n;
  n--;

  if (n > 0)
    goto loop;

  sum *= sum;
  return sum;
}
```

```
0000000000000016 <foo_do_while_goto>:
  16:    mov    $0x0,%eax
  1b:    add    %rdi,%rax
  1e:    sub    $0x1,%rdi
  22:    test   %rdi,%rdi
  25:    jg     1b <foo_do_while_goto+0x5>
  27:    imul   %rax,%rax
  2b:    retq
```